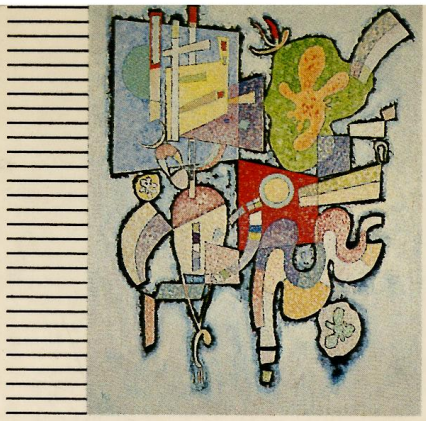
Christine Froidevaux
Marie-Claude Gaudel
Michèle Soria



TYPES DE DONNÉES ET ALGORITHMES

McGRAW-HILL

TYPES DE DONNÉES ET ALGORITHMES

Christine Froidevaux
Maître de conférences
Université Paris-Sud

Marie-Claude Gaudel
Professeur
Université Paris-Sud

Michèle Soria
Maître de conférences
Université Paris-Sud

McGRAW-HILL

Paris — Auckland — Bogota — Caracas — Hambourg — Jakarta — Lisbonne — Londres — Madrid — Mexico — Milan — Montréal — New Delhi — New York — Panama — San Juan — São Paulo — Singapour — Sydney — Tokyo — Toronto

1990

Abstraction et maîtrise de la complexité sont des aspects essentiels de l'algorithmique d'aujourd'hui. Ces concepts ne sont-ils pas remarquablement illustrés dans l'œuvre de Kandinsky ?

Source : Wassily Kandinsky, « Complexité simple », 1939, Musée national d'art moderne, Centre Georges Pompidou. Service photographique : Jacqueline Hyde, Paris.

Maquette de couverture : Françoise Rojare

ISBN : 2-7042-1217-1
ISSN : 0989-392-X

McGraw-Hill — 28, rue Beaunier — 75014 Paris

Préface

L'algorithme a pour objet la conception et l'étude d'algorithmes et de structures de données efficaces. Il s'agit d'un domaine de l'Informatique dont l'étude est indispensable dans bien des formations. Aussi de nombreux ouvrages portant sur ce thème ont-ils été publiés, en France comme à l'étranger, au cours des dernières années. Au vu de la qualité de certains d'entre eux et du renom international de plusieurs de leurs auteurs, il paraît aujourd'hui difficile d'innover en rédigeant un nouveau manuscrit sur le thème «Types de données et Algorithme».

Il importe donc particulièrement de souligner ici que le travail effectué par Christine Froidevaux, Marie-Claude Gaudel et Michèle Soria est profondément original. Son originalité tient principalement à la présentation étroitement imbriquée qui est faite de différents aspects complémentaires de la conception d'algorithmes ou de structures de données efficaces : spécification, «programmation», analyse. Chacun de ces aspects est traité avec clarté et rigueur sans éluder les difficultés.

Les auteurs, qui se placent dans le cadre des grands problèmes classiques de l'algorithmique non numérique (tri, recherche, algorithmique des graphes), présentent, dans le corps du texte ou par le biais d'exercices, une variété exceptionnellement grande de structures de données. Par ailleurs, tous les concepts importants relevant d'études de complexité (bornes inférieures, arbres de décision, technique d'oracle, bornes supérieures, optimalité, analyse en moyenne, dénombrements, etc.) sont introduits et largement utilisés.

Le travail de Christine Froidevaux, Marie-Claude Gaudel et Michèle Soria constitue donc à la fois une excellente introduction à l'Algorithmique et un précieux outil de référence.

Je suis persuadé que de nombreux lecteurs éprouveront, comme moi, un grand plaisir à lire et à étudier cet ouvrage dense, riche et stimulant.

Claude Puech

Avant-propos

L'étude des algorithmes et des types de données est un des enseignements fondamentaux en deuxième cycle d'Informatique : Licence, Maîtrise, Magistère, MIAGE, et dans les Ecoles d'Ingénieurs. Cet enseignement s'adresse à des étudiants déjà familiarisés avec la programmation. Il a pour but de leur faire connaître les algorithmes de base, tout en leur donnant de bons réflexes face à un problème à résoudre : spécification précise *a priori*, analyse rigoureuse *a posteriori* de la solution proposée.

Cet ouvrage résulte de plusieurs années d'expérience de cet enseignement. Au cours des années, son contenu a peu évolué sur le fond : les algorithmes et les types de données sont restés les mêmes. Cependant il a été sensiblement modifié quant à la forme, en fonction de l'évolution des méthodes de spécification et de programmation, et de l'introduction de nouvelles techniques d'analyse des algorithmes. Dans sa version présente, une large place a été faite à la spécification abstraite des problèmes étudiés, indépendamment des solutions possibles pour leur programmation. Ces solutions sont précisément des *algorithmes* accompagnés des *types de données* correspondants. L'analyse rigoureuse de ces algorithmes, basée sur des outils mathématiques bien compris est le sujet principal de ce livre. Cette analyse permet de comparer des algorithmes qui résolvent le même problème : d'où l'intérêt de la spécification du problème, et la nécessité d'une justification que l'algorithme résoud bien le problème posé.

La démarche suivie dans les différentes parties est donc très systématique : spécifier un problème; décrire un algorithme, éventuellement par raffinements successifs; justifier que cet algorithme résoud le problème; analyser l'algorithme puis le comparer à d'autres qui résolvent le même problème. Notre but est de montrer à quel point toutes ces étapes sont fondamentales pour l'étude des algorithmes, et reposent sur des approches qui méritent chacune d'être approfondies. Ce livre n'est pas un livre sur la spécification et l'abstraction comme Liskov & Guttag, *Abstraction and Specification in Program Development*, MIT Press, 1986, mais on y trouve la spécification d'un certain nombre de problèmes; ni un livre sur la preuve de programmes comme Gries, *The Science of Programming*, Springer Verlag, 1981, mais on y trouve des preuves; ni un livre sur les méthodes mathématiques d'analyse de la complexité des algorithmes comme Flajolet & Sedgewick, *Average-case analysis of algorithms*, Addison-Wesley, 1990, mais il

contient des analyses d'algorithmes détaillées. C'est encore moins un livre sur la construction d'algorithmes à partir de spécifications comme Anagram, *Raisonner pour programmer*, Dunod, 1986. Nous espérons cependant que notre livre incitera les lecteurs à explorer tous ces différents aspects.

L'ouvrage est organisé de la manière suivante :

• la première partie introduit le concept d'algorithme, présente les conventions pour l'expression des algorithmes et définit ce qu'est l'analyse de la complexité des algorithmes (chapitres 1 et 2). On trouve au chapitre 3 un exemple complètement traité d'analyse d'un algorithme simple : la recherche des deux plus grands éléments d'une liste d'entiers.

• la seconde partie traite de structures de données. Le concept de type abstrait est introduit au chapitre 4, et l'on présente ensuite les types de données les plus usuels : listes, ensembles, arbres, graphes (chapitres 5 à 8). Pour chacun de ces types, on donne une spécification et l'on décrit diverses implémentations, c'est-à-dire, représentations en mémoire et algorithmes d'utilisation.

Ces deux premières parties sont relativement indépendantes mais l'analyse de certains algorithmes de la deuxième partie utilise des définitions de la première partie . Les chapitres 1 à 4 introduisent les concepts fondamentaux et présentent la démarche suivie. Leur lecture est indispensable.

Le reste du livre est consacré à l'étude des principaux algorithmes et types de données pour trois classes de problèmes : recherche, tri et problèmes simples sur les graphes. Ces trois parties peuvent être étudiées indépendamment les unes des autres, mais elles font toutes appel au contenu des deux premières parties.

• Pour la recherche d'un élément dans un ensemble de données (troisième partie) on présente les méthodes classiques de recherche séquentielle et dichotomique (chapitre 9); puis des méthodes adaptées aux ensembles qui évoluent sous l'action d'adjonctions et de suppressions d'éléments, méthodes arborescentes (arbres binaires de recherche au chapitre 10, arbres équilibrés au chapitre 11), et méthodes de hachage (chapitre 12). Au chapitre 13 on compare ces différentes méthodes, qui sont étendues, pour certaines d'entre elles, à la recherche externe.

• La quatrième partie est consacrée aux méthodes de tri. Au chapitre 14 on présente le problème et des méthodes simples de résolution par sélection et par insertion; on étudie ensuite (chapitre 15) deux algorithmes plus performants, le tri rapide (quicksort) et le tri par tas (heapsort); on donne au chapitre suivant une borne inférieure du nombre de comparaisons, ce qui permet d'établir l'optimalité du tri rapide et du tri par tas; on présente aussi au chapitre 16 d'autres types de tris (tri radix...). Le chapitre 17 expose quelques méthodes de tri sur support externe : interclassement, tri et fusion.

• Dans la dernière partie, on étudie quelques questions simples sur les graphes. Le chapitre 18 illustre la démarche de description d'un algorithme par raffinements successifs, en présentant un algorithme abstrait de tri topologique, et différents algorithmes plus concrets; on étudie au chapitre 19 des propriétés de connexités; les deux derniers chapitres sont consacrés à l'étude de deux problèmes fondamentaux des graphes valués : plus courts chemins entre deux sommets, et arbres de recouvrement minimums.

Enfin on trouve en annexe une présentation des outils mathématiques utilisés pour l'analyse des algorithmes : développement asymptotique de fonctions et ordres de grandeur, séries génératrices de dénombrement et application à l'analyse en moyenne des algorithmes, et enfin techniques de résolution pour une large classe d'équations de récurrence. La lecture de l'annexe suppose une certaine pratique mathématique, mais aucune connaissance particulière n'est requise.

Chaque chapitre est accompagné de «lectures conseillées» (dans certains cas, elles sont regroupées en fin de partie). Il s'agit de livres dans lesquels le lecteur trouvera des compléments ou des développements de certains points traités.

De nombreux énoncés d'exercices sont également proposés à la fin de chaque chapitre. Nous les avons volontairement choisis simples ou de difficulté moyenne afin de permettre au lecteur isolé de tester sa bonne compréhension du texte.

Le contenu de l'ensemble de l'ouvrage correspond à un peu moins d'une année d'enseignement d'un certificat de licence ou de maîtrise, le début de l'année étant consacré à l'enseignement de la programmation. Il n'est pas obligatoire, comme cela a déjà été mentionné, de suivre linéairement la présentation du livre.

Il est impossible de remercier personnellement tous ceux ou celles qui, d'une manière ou d'une autre, ont contribué à cet ouvrage : les origines et les sources de ce livre sont multiples.

Les enseignements de Philippe Flajolet, Claude Pair et Jean Vuillemin nous ont influencées de façon certaine. Notre livre s'appuie sur d'excellents ouvrages d'algorithmique, tels que Aho, Hopcroft et Ullmann, *Data Structures and Algorithms*, Addison-Wesley, 1983, Baase, *Computer Algorithms*, Addison Wesley, 1978; Horowitz et Sahni, *Fundamentals of Computer Algorithms*, Pitman, 1978; Knuth, *The Art of Computer Programming*, volumes 1 et 3, Addison Wesley, 1968 et 1973; Pair et Gaudel, *Les structures de données et leur représentation en mémoire*, Monographie IRIA, 1979; Sedgewick, *Algorithms*, Addison Wesley 1983.

Le service d'édition de L'INRIA (SEDIS) nous a permis, il y a quelques années, de publier une première version de cet ouvrage. Qu'il en soit remercié. Les remarques de nos étudiants et de certains collègues sur cette première version ont été précieux.

Nous tenons spécialement à remercier tous nos collègues d'Orsay qui ont participé, de près ou de loin, à l'enseignement d'algorithmique et tout particulièrement :

Jean-Claude Bermond, Johny Bond, Serena Cerrito, Annie Choquet, Danièle Gardy, Anne Germa, Marie-Claude Heydemann, Bruno Marre, Maryvonne Mahéo, Michel de Rougemont, Marie-Christine Rousset et Frédéric Voisin.

Il ne faut pas oublier l'importante contribution de Pierre Legay qui, avec beaucoup de perspicacité, a mis en œuvre en Pascal la plupart des algorithmes de ce livre.

Merci à Daniel Baldit qui nous a aidées à choisir l'illustration de la couverture.

La réalisation de ce livre a enfin bénéficié de tout l'environnement matériel du Laboratoire de Recherche en Informatique du centre d'Orsay de l'Université Paris-Sud.

A propos des auteurs

Christine Froidevaux, Marie-Claude Gaudel et Michèle Soria sont toutes trois enseignants-chercheurs au Laboratoire de Recherche en Informatique de l'Université d'Orsay, Unité Associée du CNRS.

Christine Froidevaux est maître de conférences et effectue des travaux dirigés en algorithmique. Ses recherches portent sur la représentation des connaissances et la formalisation du raisonnement humain, à l'aide de logiques non classiques. Elle a publié des articles et participé à la rédaction de plusieurs livres sur ces sujets.

Marie-Claude Gaudel est professeur et a la charge de plusieurs cours d'algorithmique. Elle est l'auteur d'articles et d'ouvrages sur la théorie des types abstraits, les méthodes de spécification et de validation de logiciel, et plus généralement, le génie logiciel.

Michèle Soria est maître de conférences et effectue des cours et des travaux dirigés en algorithmique. L'objet de ses recherches est l'évaluation asymptotique de la complexité des algorithmes à l'aide de méthodes d'analyse complexe. Ses publications portent sur la complexité en moyenne des algorithmes et sur les propriétés statistiques des structures combinatoires.

Table des matières

Première partie

Introduction à l'analyse des algorithmes

Chapitre 1
Notion d'algorithme, expression des algorithmes

1. Notion d'algorithme

Selon l'*Encyclopedia Universalis* [1] un algorithme est la «spécification d'un schéma de calcul, sous forme d'une suite [finie] d'opérations élémentaires obéissant à un enchaînement déterminé».

On connaît depuis l'antiquité des algorithmes sur les nombres, comme par exemple l'algorithme d'Euclide qui permet de calculer le p.g.c.d. de deux nombres entiers. Pour le traitement de l'information, on a développé des algorithmes opérant sur des données non numériques : les algorithmes de tri, qui permettent par exemple de ranger par ordre alphabétique une suite de noms, les algorithmes de recherche d'une chaîne de caractères dans un texte, ou les algorithmes d'ordonnancement, qui permettent de décrire la coordination entre différentes tâches, nécessaire pour mener à bien un projet.

Un programme destiné à être exécuté par un ordinateur, est la plupart du temps, la description d'un algorithme dans un langage accepté par cette machine. La notion d'algorithme est donc fondamentale en informatique. Cependant le terme d'«algorithme» a en général une acception plus large que celle de «méthode de résolution d'un problème pouvant être réalisée par un ordinateur».

> *Tout d'abord, historiquement, la notion d'algorithme a précédé celle d'ordinateur : bien avant Euclide (mathématicien grec du 3e siècle de notre ère), on peut faire remonter aux Babyloniens de l'époque d'Hammurabi (1800 av J.C.) les premières formulations de règles précises pour la résolution de certains types d'équations.*
> *Le mot «algorithme» lui-même est plus récent : il vient du nom d'un mathématicien perse du 9e siècle : Abu Ja'Far Mohammed Ibn Mûsâ al-Khowâ-rismî,*

(1) Encyclopedia Universalis, Paris, édition de 1984.

dont l'ouvrage d'arithmétique utilisant les règles de calcul sur la représentation décimale des nombres, montra l'inutilité des tables et abaques, et eut une influence capitale pendant plusieurs siècles.
D'autre part, on verra dans ce livre que certains algorithmes, ceux dont le temps d'exécution ou l'encombrement en mémoire croissent de façon exponentielle en fonction de la taille des données, ne peuvent être utilisés en pratique par un ordinateur, car on ne peut leur accorder qu'un temps et une mémoire limités.

Mais il est vrai que la notion d'algorithme est très liée à l'idée de machine capable d'exécuter des opérations mathématiques élémentaires. Vers 1930, avant la construction des premiers ordinateurs, plusieurs mathématiciens (Gödel, Church, Turing, Post) ont formalisé le concept d'algorithme, ou de fonction calculable, par différents modèles abstraits à ressources infinies (Fonctions récursives, λ-calcul, Machines de Turing, Systèmes de Post, Machines à registres). La «Théorie de la calculabilité» montre que tous ces modèles sont équivalents et l'on peut alors penser (c'est la «Thèse de Church ») que l'on a bien cerné exactement la notion de fonction calculable, c'est-à-dire que toute autre définition «raisonnable» est équivalente aux précédentes.

La théorie de la calculabilité décrit et caractérise les problèmes qui peuvent être résolus par un algorithme (on parle alors de problème «décidable» ou «calculable»), et répertorie ceux qui ne le peuvent pas.

Il existe en effet des problèmes mathématiquement bien posés, mais qui sont trop complexes pour pouvoir être résolus par un algorithme. Par exemple, l'arithmétique est une théorie dont les énoncés sont complètement formalisés dans un langage bien précis, mais il est démontré qu'il ne peut pas exister d'algorithme général pour décider si une assertion arithmétique quelconque est vraie ou fausse.

Par ailleurs, il existe des fonctions non calculables; pour s'en persuader, il suffit de considérer toutes les fonctions possibles des entiers dans les entiers : cet ensemble n'est pas dénombrable. Par contre, l'ensemble des fonctions calculables l'est, puisqu'à toute fonction de ce type on sait associer un algorithme qui est un texte de longueur finie sur un langage de cardinalité finie.

Un exemple fondamental de problème non décidable est le «*Problème de l'arrêt* » : il est démontré qu'il ne peut pas exister d'algorithme général qui, pour tout programme P et toute donnée D, répondrait oui ou non à la question «P termine-t-il pour D?».

Dans ce livre, on étudie des problèmes pour lesquels il existe des algorithmes, et on analyse la complexité de ces algorithmes, c'est-à-dire les ressources nécessaires à leur exécution, en temps et en mémoire. On ne s'intéresse qu'aux algorithmes de complexité «raisonnable». En effet, il ne suffit pas de savoir qu'il existe un

algorithme pour être sûr qu'il est utilisable pratiquement. Un exemple bien connu est le *jeu d'échec.*

> En théorie, à chaque coup joué, il y a un nombre fini de mouvements possibles. Sous certaines règles, une partie se termine, d'une manière ou d'une autre, après un nombre fini de coups. On peut donc concevoir un programme qui calculerait toutes les conséquences de tous les coups possibles et qui serait meilleur que tout joueur humain. Mais la complexité du problème est telle qu'il n'est pas envisageable de mettre un tel algorithme en pratique. Il faudrait considérer de l'ordre de 10^{19} coups possibles pour décider de chaque déplacement. Rappelons que 10^{19} millisecondes est de l'ordre de 300 millions d'années.

Dans le cas d'une complexité «hors du possible», comme pour le jeu d'échec, on envisage alors des méthodes approchées, appelées des *heuristiques.* Une heuristique est une méthode qui produit la plupart du temps un résultat acceptable, pas nécessairement optimal, dans un temps raisonnable.

On donne maintenant des définitions plus précises des notions «d'algorithme» et de «machine» qui vont être à la base de ce livre.

• Un ***algorithme*** décrit un traitement sur un certain nombre, fini, de *données* (éventuellement aucune).

• Un ***algorithme*** est la composition d'un ensemble fini d'*étapes*, chaque étape étant formée d'un nombre fini d'*opérations* dont chacune est :

- *définie* de façon rigoureuse et non ambiguë;
- *effective* c'est-à-dire pouvant être effectivement réalisée par une machine : cela correspond à une action qui peut être réalisée avec un papier et un crayon en un temps fini; par exemple la division entière est une opération effective, mais pas la division avec un nombre infini de décimales.

• Quelle que soit la donnée sur laquelle il travaille, un algorithme doit toujours *se terminer* après un nombre fini d'opérations, et fournir un *résultat.*

De plus, les algorithmes que l'on considère sont *déterministes* : étant donné un algorithme, toute exécution de cet algorithme sur les mêmes données donne lieu à la même suite d'opérations.

Pour l'analyse de la complexité des algorithmes, il est aussi important d'expliciter certaines hypothèses concernant le modèle de machine sur lequel ces algorithmes seront exécutés.

Un ordinateur est une machine disposant d'une *unité de calcul* pour exécuter les opérations arithmétiques et logiques et d'une *mémoire* pour contenir et manipuler les instructions et les données, telles que :

a) On peut avoir accès à (ou ranger) un élément dans la mémoire en un temps fixe, ce qui est le cas pour les mémoires à accès aléatoire.

b) L'unité de calcul ne peut traiter qu'une seule opération à la fois. Cela signifie que l'on n'étudie pas dans ce livre, d'algorithmes comportant du parallélisme. En effet, l'analyse de ces algorithmes nécessite des techniques spécifiques.

c) Le coût de transfert des informations est négligeable devant le coût des opérations, ce qui n'est plus le cas si l'on travaille sur des machines très câblées (VLSI).

Les principes de l'analyse de la complexité des algorithmes sous ces hypothèses sont exposés au chapitre 2. Le paragraphe suivant présente tout d'abord le langage d'expression des algorithmes.

2. Expression des algorithmes

Un algorithme doit être exprimé dans un langage de programmation pour être compris et exécuté par un ordinateur. Cependant il faut bien comprendre qu'un algorithme est indépendant du langage de programmation utilisé. L'algorithme d'Euclide programmé en Pascal, en Ada ou en Lisp, reste toujours l'algorithme d'Euclide.

On donne ici quelques règles méthodologiques qui permettent une expression claire et concise des algorithmes. Ces règles s'inspirent fortement des principes de la programmation structurée.

2.1. Méthodologie

Un algorithme pour être compréhensible doit être présenté de manière *lisible*; il ne doit pas être trop long, ce qui nécessite parfois de le décomposer en *modules*; certaines formes de programmation comme les *branchements* sont à éviter; par contre, la *récursivité* permet une expression concise des algorithmes et facilite leur analyse. Enfin, le problème résolu par l'algorithme doit être clairement *spécifié*.

2.1.1. Lisibilité d'un algorithme

La lisibilité d'un algorithme peut être améliorée de manière significative par sa mise en page. Retour à la ligne et indentation peuvent être utilisés avec discernement comme le montre l'exemple ci-dessous :

Mise en page 1

```
i := 0; S := 0; while i < n do
begin i := i + 1; if T[i] > 0 then
S := S + T[i] else S := S - T[i] end;
```

Mise en page 2

```
i := 0; S := 0;
while i < n do begin
      i := i + 1;
      if T[i] > 0 then S := S + T[i]
      else S := S - T[i]
end;
```

Les commentaires peuvent également être utiles à condition qu'ils ne paraphrasent pas le programme, qu'ils soient courts et qu'ils se limitent à l'explication des idées essentielles ou des points délicats.
Par exemple, le commentaire :

{*ce programme calcule la somme des valeurs absolues des éléments de T*}

ajouté au début du programme précédent améliore sans aucun doute sa lisibilité.

2.1.2. Modularité

Il s'agit de découper l'algorithme en modules aussi indépendants que possible. Dans ce cas, *l'interface* des différents modules avec l'extérieur doit être décrite très précisément : variables d'entrée-sorties, variables globales utilisées et/ou modifiées. De plus, les liaisons des différents modules entre eux doivent apparaître clairement : quel module utilise quel autre module, quels modules partagent une même variable globale. Ces informations permettent d'avoir une vue d'ensemble de la décomposition qui a été faite.

Enfin, le rôle de chaque module doit être explicité clairement, soit sous forme de formules, soit sous forme d'une phrase en langage naturel : on donne la *spécification* de ce module. On reviendra plus loin sur cette importante notion.

2.1.3. Suppression des instructions de branchement

Il est démontré qu'il est toujours possible de programmer sans branchement (**go to**) en utilisant uniquement des boucles (**while, for**), des instructions conditionnelles (**if - then - else**) ou des procédures.

Cependant, il existe des situations où le fait de ne pas utiliser de branchements obscurcit le programme au lieu de le rendre plus compréhensible. C'est le cas lorsque l'on a des conditions multiples pour sortir d'une boucle, et que ces sorties se font depuis des endroits différents du corps de la boucle. Si on s'autorise une instruction de sortie de boucle (**exit**) ou de procédure (**return**) ce phénomène ne se produit pas. On utilise de telles instructions dans ce livre.

2.1.4. Utilisation de la récursivité

L'expression d'algorithmes sous forme récursive permet des descriptions concises, qui se prêtent bien à des démonstrations par récurrence. Le principe est d'utiliser, pour décrire l'algorithme sur une donnée d, l'algorithme lui-même appliqué à un sous-ensemble de d ou à une donnée d' plus petite.

Exemple 1

```
type cellule = record element : integer;
                      suivant : ↑cellule
               end;
     liste = ↑cellule;

function longueur(L : liste) : integer;
begin
     if L = nil then longueur := 0
     else longueur := 1 + longueur(L ↑. suivant);
end longueur;
```

Exemple 2

```
function pgcd(n, m : integer) : integer;
var r : integer;
begin
     {n est supposé supérieur ou égal à m}
     r := n − (n div m) ∗ m;
     if r = 0 then pgcd := m
     else pgcd := pgcd(m, r)
end pgcd;
```

Le premier de ces exemples utilise un type défini de manière récursive : le type cellule. Généralement, les algorithmes qui utilisent des données d'un type défini récursivement se décrivent très naturellement de manière récursive.

La conception d'algorithmes récursifs doit se faire en suivant quelques règles de bon sens : il faut s'assurer qu'on ne réapplique pas l'algorithme à des données plus grandes et qu'on a bien un test de *terminaison*, qui correspond à un cas où la donnée est suffisamment élémentaire pour être traitée directement sans réapplication de l'algorithme.

Cependant, il existe des langages de programmation qui ne permettent pas des appels récursifs de sous-programmes. De plus, la formulation récursive d'un algorithme peut parfois masquer un problème de complexité : on utilise en effet une pile «cachée» pour stocker les résultats intermédiaires.

Il est nécessaire lorsque l'on donne un algorithme récursif, d'en donner ensuite une version itérative équivalente. Dans certains cas (récursivité terminale) cette transformation est simple. Mais le cas général est très compliqué. On se contente de donner dans ce livre les règles qui s'appliquent dans les cas particuliers que l'on rencontre : parcours d'arbres ou de graphes, recherche dichotomique, tri par sélection, tri par insertion et tri rapide.

2.1.5. Spécification d'un algorithme

La spécification d'un algorithme décrit ce que l'algorithme fait, sans détailler comment il le fait.

On en a vu un exemple au §2.1.1 où le commentaire

{*ce programme calcule la somme des valeurs absolues des éléments de T*}

est une spécification. Considérons comme deuxième exemple la recherche séquentielle d'un élément dans une liste :

```
{Ce programme recherche la place d'un élément X dans une liste L.
Si X n'est pas dans la liste le résultat est zéro.
La liste est représentée par un tableau. Elle comporte n éléments.}
j := 1;
while (j ≤ n) and (L[j] ≠ X)
      do j  := j + 1;
if j > n then j  := 0;
```

La spécification de l'algorithme est donnée en italique, au début de celui-ci. Cette convention est suivie partout dans ce livre. Toute spécification doit être établie avec beaucoup de soin car la comparaison d'algorithmes n'a de sens que si les spécifications sont les mêmes, ou divergent sur des cas particuliers bien répertoriés.

On peut noter que la spécification ci-dessus est imprécise.

Est-ce que la liste peut être vide ?
Que fait-on s'il y a deux occurrences de X ?
Quels sont les données et les résultats ?

Une spécification précise serait :

{*Les données sont L et X. L est un tableau de n éléments qui représente une liste. Cette liste peut être vide (n=0).*
On recherche l'indice de l'élément X dans L.
L'algorithme termine avec la variable j égale à l'indice de la première occurrence de X dans L, s'il y en a une. Si X n'est pas dans la liste, la variable j vaut zéro à la fin de l'algorithme.}

Une spécification peut aussi comporter des formules mathématiques ou logiques comme :

$$S = \sum_{i=1}^{n} | T[i] |$$

ou $$card(L) = n, n \geq 0$$
$$X \notin L \Rightarrow j = 0$$
$$(X = L[k]) \,\&\, (\forall i \in [1, k-1]\, X \neq L[i]) \Rightarrow j = k$$

Dans le cas où elle ne comporte que des formules, on parle de ***spécification formelle.***

2.2. Conventions pour l'écriture des algorithmes

Le langage d'expression des algorithmes utilisé dans ce livre est Pascal enrichi d'un certain nombre de constructions destinées à éviter des détails de programmation qui pourraient obscurcir ce qui est essentiel dans les algorithmes décrits.

Les différences avec Pascal sont les suivantes.

1) Le **and** et le **or** sont séquentiels; cela signifie que

A **and** B est équivalent à **if not** A **then** false **else** B;
et A **or** B est équivalent à **if** A **then** true **else** B;

Le deuxième opérande n'est donc pas évalué quand le premier permet de déterminer le résultat.

2) On a une instruction d'échange entre deux variables qui se note : $X \leftrightarrow Y$. La variable X prend l'ancienne valeur de Y et Y prend l'ancienne valeur de X. Cette instruction nécessite pour être exécutée une variable auxiliaire cachée. Cela doit être pris en compte dans les calculs de complexité.

3) On peut sortir d'une boucle par l'instruction **exit.** Cette instruction fait sortir de la boucle la plus interne. Dans l'algorithme de recherche séquentielle donné précédemment (§2.1.5), on aurait pu l'utiliser de la manière suivante :

```
while j ≤ n do
    if L[j] = X then exit
                else j := j + 1;
```

4) On peut également sortir du corps d'une procédure ou d'une fonction par l'instruction **return.** Dans le cas d'une procédure, cette instruction provoque l'abandon de la procédure et la reprise de l'exécution à l'instruction qui suit l'appel.

Dans le cas d'une fonction, on précise en argument de **return** la valeur du résultat de la fonction et on reprend l'évaluation de l'expression à l'endroit où se trouve l'appel de la fonction. Par exemple, le programme de recherche séquentielle, écrit sous forme de fonction devient :

```
type liste = array [1..n] of element;
function recherche (L : liste; X : element) : integer;
var j : integer;
begin
    j := 1;
    while j ≤ n do
        if L[j] = X then return (j)
                    else j := j + 1;
    return (0);
end recherche;
```

Chapitre 2
Complexité des algorithmes

L'exécution d'un programme nécessite l'utilisation des ressources de l'ordinateur : temps de calcul pour exécuter les opérations, et occupation de la mémoire pour contenir et manipuler le programme et ses données.

L'objet de l'analyse de la complexité est de quantifier les deux grandeurs physiques «temps d'exécution» et «place mémoire», dans le but de comparer entre eux différents algorithmes qui résolvent le même problème.

Il s'agit d'abord de déterminer quelle mesure utiliser pour calculer ces deux quantités : pour un programme donné sur une machine donnée, on peut par exemple exprimer la complexité en temps (resp. en place) par le nombre de cycles machine (resp. mots mémoire) utilisés lors de l'exécution du programme, en comptant

• pour le temps : le nombre d'opérations effectuées par le programme et le temps nécessaire pour chaque opération,

• pour la place : le nombre d'instructions et le nombre de données du programme, avec le nombre de mots mémoire nécessaires pour stocker chacune d'entre elles, ainsi que le nombre de mots mémoire supplémentaires pour la manipulation des données. Ce type d'analyse conduit à des énoncés comme :

> «*L'algorithme A implémenté par le programme P sur l'ordinateur O, et exécuté sur la donnée D utilise k secondes de calcul et j bits de mémoire.*»

Un résultat de ce genre peut être une source d'information intéressante, mais le but de l'analyse de la complexité des algorithmes est d'établir des résultats plus généraux permettant d'estimer l'efficacité intrinsèque de la méthode utilisée par un algorithme, indépendamment de la machine, du langage de programmation, du compilateur et de tous les détails d'implémentation.

Le type d'énoncé que l'on souhaite produire est :

> «*Sur toute machine, et quel que soit le langage de programmation, l'algorithme A1 est meilleur que l'algorithme A2 pour les données de grande taille.*»

ou encore,

> *«L'algorithme A est optimal en nombre de comparaisons pour résoudre le problème Q.»*

On précise plus loin ce que l'on entend par «meilleur», «taille d'une donnée», «optimal»...

1. Complexité d'un algorithme

On cherche à déterminer une mesure qui rende compte de la complexité intrinsèque des algorithmes, indépendamment de l'implémentation, et permette ainsi de comparer entre eux des algorithmes.

On va pour l'instant se consacrer à l'étude de la complexité en temps, et on reviendra à la fin du paragraphe sur la complexité en place.

1.1. Mesure de la complexité en temps

Pour certains problèmes, on peut mettre en évidence une ou plusieurs *opérations* qui sont *fondamentales* au sens où le temps d'exécution d'un algorithme résolvant ce problème est toujours proportionnel au nombre de ces opérations. Il est alors possible de comparer des algorithmes traitant ce problème selon cette mesure simplifiée.

Donnons quelques exemples d'opérations fondamentales :

- pour la recherche d'un élément dans une liste en mémoire centrale : le nombre de comparaisons entre cet élément et les entrées de la liste;
- pour la recherche d'un élément sur un disque : le nombre d'accès à la mémoire secondaire;
- pour trier une liste d'éléments : on peut considérer deux opérations fondamentales : le nombre de comparaisons entre deux éléments et le nombre de déplacements d'éléments;
- pour multiplier deux matrices de nombres : le nombre de multiplications et le nombre d'additions.

Remarquons que si l'on choisit plusieurs opérations fondamentales, on doit les décompter séparément puis, si besoin est, on les affecte chacune d'un poids qui tient compte des temps d'exécution différents.

Soulignons deux points importants.

a) En faisant varier le nombre d'opérations fondamentales, on fait varier le degré de précision de l'analyse, et aussi son degré d'abstraction, i.e. d'indépendance par rapport à l'implémentation. A la limite si l'on veut faire une «microanalyse» très

précise du temps d'exécution du programme, il suffit de décider que toutes les opérations du programme sont fondamentales.

b) On a fait l'hypothèse que le temps d'exécution est proportionnel à la mesure choisie. Cependant il se peut qu'après avoir analysé quelques algorithmes en fonction d'une certaine opération choisie comme fondamentale pour le problème, on découvre des algorithmes résolvant le même problème – ou un sous-problème – par des méthodes si différentes qu'ils ne font aucune opération de ce type. Dans ce cas, il faut se restreindre à la «classe d'algorithmes» pour laquelle la mesure choisie est significative. Les algorithmes qui utilisent d'autres techniques nécessitant le choix d'autres opérations fondamentales sont étudiés séparément, et ne peuvent pas être comparés à ceux de la classe précédente (on verra un exemple de cette situation pour des algorithmes de tri).

1.2. Calcul de la complexité

Après avoir déterminé les opérations fondamentales, il s'agit de compter le nombre d'opérations de chaque type. Il n'existe pas de système complet de règles permettant de compter le nombre d'opérations en fonction de la syntaxe des algorithmes mais l'on peut faire quelques remarques.

a) Lorsque les opérations sont dans une séquence d'instructions, leurs nombres s'ajoutent.

b) Pour les branchements conditionnels, il est en général difficile de déterminer quelle branche de la condition est exécutée, et donc quelles sont les opérations à compter. Cependant, on peut majorer ce nombre d'opérations : par exemple, si $P(X)$ est le nombre d'opérations fondamentales de la construction X :

$$P\ (\textbf{if}\ C\ \textbf{then}\ I_1\ \textbf{else}\ I_2) \ \leq\ P(C) + \max\left(P(I_1),\ P(I_2)\right)$$

c) Pour les boucles, le nombre d'opérations dans la boucle est $\sum P(i)$, où i est la variable de contrôle de la boucle, et $P(i)$ le nombre d'opérations fondamentales lors de l'exécution de la $i^{ième}$ itération.

Pour évaluer les bornes de i dans la somme précédente, il faut connaître le nombre d'itérations du corps de la boucle. Ce nombre peut être défini dans l'algorithme (cas d'une «boucle **for**»), sinon il doit être calculé à partir de l'algorithme et ce calcul peut s'avérer difficile; dans ce cas on peut se contenter quelquefois d'un majorant.

d) Pour les appels de procédure ou de fonction :

• s'il n'y a pas de procédures ou de fonctions récursives, on peut toujours trouver une façon d'ordonner les procédures et fonctions de telle sorte que chacune d'entre elles n'appelle que des procédures ou fonctions dont le nombre d'opérations fondamentales a déjà été évalué;

• pour les procédures ou fonctions récursives, compter le nombre d'opérations fondamentales donne en général lieu à la résolution de relations de récurrence. En effet, le nombre $T(n)$ d'opérations dans l'appel de la procédure avec un argument de taille n s'écrit, selon la récursion, en fonction de divers $T(k)$, pour $k < n$. Par exemple, l'algorithme ci-dessous calcule la factorielle d'un entier positif.

```
function fact (n : integer) : integer;
begin
    if n = 0 then fact := 1 else fact := n * fact (n - 1)
end fact;
```

Si l'on choisit comme opération fondamentale la multiplication de deux entiers, on obtient clairement que le nombre $T(n)$ d'opérations fondamentales vérifie $T(0) = 0$ et $T(n) = T(n-1) + 1$, pour $n \geq 1$, d'où par une résolution directe de cette récurrence très simple, $T(n) = n$.

(On trouvera en annexe un exposé de certaines méthodes de résolution de relations de récurrence.)

Pour illustrer les indications ci-dessus sur la manière de déterminer la complexité d'un algorithme (on dit qu'on analyse l'algorithme), on va traiter un exemple.

Reprenons l'algorithme de recherche séquentielle d'un élément dans une liste (on a indiqué des numéros de ligne pour faciliter la compréhension de l'analyse).

```
        var L : array [1..n] of element;
            X : element;
            j : integer;
        begin
(1)         j := 1;
(2)         while (j ≤ n) and (L[j] ≠ X)
(3)               do j := j + 1;
(4)         if j > n then j := 0
        end;
```

Dans cet algorithme, les éléments significatifs pour analyser la complexité en nombre d'opérations sont les suivants :

- le nombre d'itérations,
- le nombre d'opérations par itération.

On peut remarquer que l'instruction $j := j + 1$ de la ligne (3) est dépendante de la programmation : elle disparaît si on programme l'algorithme différemment, avec une boucle **for**. Il en est de même de la comparaison $j \leq n$ de la ligne (2). On voit bien qu'il ne faut pas prendre en compte ces opérations pour l'évaluation de l'algorithme.

De plus ces instructions sont dépendantes de la structure de données choisie pour représenter la liste d'éléments (cf. chapitre 5). Avec une liste chaînée on aurait d'autres instructions : manipulations de pointeurs, test si un pointeur est égal à nil.

Les opérations significatives dans cet algorithme sont donc les comparaisons de X avec les éléments de la liste. Il y en a une par itération.

Le nombre d'itérations est égal à n si X n'est pas dans la liste, et à j, rang de la première occurrence de X, si X est dans la liste.

L'analyse se fait en établissant des *invariants de boucle*, c'est-à-dire des propriétés qui sont vraies à chaque itération, et des *conditions d'arrêt.*

a) Invariant de boucle :

(i) – au début de la première itération (ligne (2)) on a $j = 1$,
– au début de la $k^{ième}$ itération on a : $j = k$ et $\forall i,\ 1 \leq i < k,\ L[i] \neq X$.

b) Conditions d'arrêt :

(ii) si au début de la $k^{ième}$ itération de la boucle on a : $k \leq n$ et $L[k] = X$, alors on va s'arrêter avec $j = k$;
(iii) si on a $k = n + 1$, alors on va s'arrêter avec $j = 0$.

On démontre les propriétés précédentes par récurrence sur k, nombre d'itérations.

Pour $k = 1$:

- La propriété (i) est vraie.
- Pour (ii) : $1 \leq n$ et $L[1] = X$, donc on sort de la boucle, on exécute le **if-then** et j reste inchangé (car $j \leq n$) : $j = k = 1$.
- Pour (iii) : si $k = n + 1$ alors $n = 0$; la liste est vide.
  On va exécuter le **if-then.**
  Comme $j > n$ et $n = 0$, on exécute $j := 0$.

Supposons les propriétés vraies pour k. On les démontre pour $k + 1$:

- Démontrons la propriété (i) à la $(k+1)^{ième}$ itération : on avait $j = k$ au début de la $k^{ième}$ itération (hypothèse de récurrence). On a eu $k \leq n$ et $L[k] \neq X$ puisqu'on a continué la boucle.

Donc, on a exécuté $j := j + 1$. Donc, on a maintenant $j = k + 1$.

Or, on avait $L[i] \neq X,\ 1 \leq i < k$ par hypothèse de récurrence, de plus $L[k] \neq X$ donc $L[i] \neq X,\ 1 \leq i < k + 1$.

- Les propriétés (ii) et (iii) se démontrent de la même façon que pour le cas où $k = 1$.

Les propriétés (ii) et (i) impliquent que j est l'indice de la première occurrence de X dans L.

Les propriétés (iii) et (i) impliquent que si $j = 0$ à la fin de l'algorithme, il n'y a pas d'occurrence de X dans L.

On a donc démontré qu'il y a au plus n itérations, et qu'il y en a j, si j est l'indice de la première occurrence de X. Comme il y a une comparaison par itération, si $j \leq n$, on sait maintenant que cet algorithme effectue : j comparaisons si j est l'indice de la première occurrence de X, n comparaisons si X n'est pas dans la liste.

Ce premier exemple d'analyse d'algorithme met en évidence trois points essentiels :

- le choix de (ou des) l'opération(s) que l'on prend en compte doit être établi avant toute analyse et précisé dans le résultat,
- la complexité dépend de la taille des données (ici n),
- la complexité dépend, pour une taille fixée, des différentes données possibles.

Ici pour les données de taille n, la complexité varie de 1 à n. Ceci dépend des données X et L : les données où X apparaît au rang i de L correspondent à une complexité i en nombre de comparaisons; celles où X n'apparaît pas dans L, à une complexité n.

1.3. Autres critères d'évaluation

La complexité en temps n'est pas le seul critère d'évaluation d'un algorithme. D'autres critères entrent en ligne de compte, comme la place mémoire, la simplicité de l'algorithme, ou l'adéquation à certaines données.

Il se peut en effet que l'on soit intéressé principalement par la complexité en place, c'est-à-dire la quantité de mémoire nécessaire à l'exécution d'un algorithme. De même que pour la complexité en temps, on peut définir différentes mesures qui rendent compte de la complexité intrinsèque d'un algorithme avec une certaine structuration des données, indépendamment de l'implémentation. Très souvent, un algorithme plus rapide utilisera plus de place; dans le cas extrême où un algorithme est très efficace, mais utilise plus de place qu'il n'y en a en mémoire centrale, étant donné qu'il est au moins un million de fois plus lent d'accéder à une information rangée en mémoire secondaire, il vaut évidemment mieux utiliser un algorithme moins «rapide» pouvant être traité uniquement en mémoire centrale. Il est donc important de choisir un algorithme avec un bon *compromis espace-temps.*

D'autre part, la simplicité et la clarté d'un algorithme peuvent aussi être un critère de choix : un algorithme efficace mais complexe sera probablement plus difficile, donc plus long, à implanter correctement qu'un algorithme moins subtil; il faut tenir

compte de ce «temps humain», en particulier si l'algorithme doit être utilisé peu de fois.

De plus, certains algorithmes, peu performants en général, sont très efficaces sur certaines formes de données (par exemple, le tri insertion sur des listes presque triées, cf. chapitre 14).

Et enfin, pour des problèmes particuliers, d'autres critères peuvent être prépondérants : par exemple, la précision pour un algorithme numérique ou la stabilité pour un algorithme de tri (cf. chapitres 14 et 16). Il faut donc savoir faire des compromis lorsqu'on choisit un algorithme et prendre en compte le contexte dans lequel le programme sera développé et utilisé.

2. Complexité en moyenne et au pire

Il est clair, et l'exemple de la recherche séquentielle l'a bien mis en évidence, que le temps d'exécution d'un algorithme dépend de la donnée sur laquelle il opère.

• Il faut d'abord définir une mesure de taille sur les données qui reflète la quantité d'information contenue. Par exemple, si l'on additionne ou multiplie des entiers, une mesure significative est le nombre de chiffres des nombres; dans le cas de la recherche (cf. 2^e partie de ce livre) ou du tri (cf. 3^e partie de ce livre), la taille sera souvent le nombre d'éléments manipulés; dans le cas du produit de matrices carrées, on peut prendre comme mesure de la taille, la dimension de la matrice; pour les parcours d'arbres (chapitre 7) ou de graphes (chapitre 8), on peut compter le nombre de nœuds ou le nombre d'arcs...

• Pour certains algorithmes (par exemple addition ou multiplication usuelles sur les nombres entiers), le temps d'exécution ne dépend que de la taille des données; mais la plupart du temps la complexité varie aussi, pour une taille fixée des données, en fonction de la donnée elle-même.

On peut définir plusieurs quantités pour caractériser le comportement d'un algorithme sur l'ensemble D_n des données de taille n.

Notons $coût_A(d)$ la complexité en temps de l'algorithme A sur la donnée d, complexité déterminée selon les méthodes décrites au paragraphe 1.2.

Définitions : On s'intéresse à plusieurs mesures.

a) La complexité ***dans le meilleur des cas***

$$\mathrm{Min}_A(n) = \min\{coût_A(d);\ d \in D_n\}$$

b) La complexité ***dans le pire des cas***

$$\mathrm{Max}_A(n) = \max\{coût_A(d);\ d \in D_n\}$$

c) La complexité ***en moyenne***

$$\text{Moy}_A(n) = \sum_{d \in D_n} p(d) \cdot coût_A(d)$$

où $p(d)$ est la probabilité que l'on ait la donnée d en entrée de l'algorithme.

(On omet l'indice A lorsque l'algorithme en cause est évident.)

Ces définitions entraînent plusieurs remarques.

1) Les complexités dans le meilleur et dans le pire des cas donnent des indications sur les bornes extrêmales de la complexité de l'algorithme sur les données de taille n. Leur détermination nécessite en général la construction de données particulières, qui forcent l'algorithme à se comporter de façon extrêmale.

La complexité dans le pire des cas, qui donne une borne supérieure du temps d'exécution, est particulièrement utile car elle permet de donner une estimation de la taille maximale des données qui pourront être traitées par l'algorithme.

2) Les cas extrêmaux ne sont pas les plus fréquents et dans la pratique on aimerait savoir quel comportement attendre «en général» de l'algorithme, d'où l'introduction de la complexité en moyenne.

Si toutes les données sont équiprobables alors la complexité en moyenne s'exprime simplement en fonction du nombre $|D_n|$ de données de taille n :

$$\text{Moy}_A(n) = \frac{1}{|D_n|} \cdot \sum_{d \in D_n} coût_A(d)$$

Mais en général, les données n'ont pas toutes la même probabilité et la définition de la complexité en moyenne nécessite l'introduction d'un modèle probabiliste lié au problème.

Souvent (cf. l'exemple, qui suit, de la recherche séquentielle, et l'exemple de la recherche du maximum au chapitre 3) on partitionne l'ensemble D_n en regroupant les données de taille n de même coût, et on évalue la probabilité $p(D_{n,k})$ de chaque classe $D_{n,k}$ de la partition. La complexité en moyenne devient alors

$$\text{Moy}_A(n) = \sum_{D_{n,k} \subseteq D_n} p(D_{n,k}) \cdot coût_A(D_{n,k})$$

où $coût_A(D_{n,k})$ représente le coût d'une donnée quelconque de $D_{n,k}$.

En pratique, la complexité en moyenne est souvent beaucoup plus difficile à déterminer que la complexité dans le pire des cas, d'une part parce que l'analyse devient mathématiquement difficile, et d'autre part parce qu'il n'est pas toujours facile de déterminer un modèle de probabilités adéquat au problème.

3) Clairement, il existe entre la complexité en moyenne et les complexités extrêmales la relation suivante :

$$\mathrm{Min}_A(n) \leq \mathrm{Moy}_A(n) \leq \mathrm{Max}_A(n)$$

Si le comportement de l'algorithme ne dépend que de la taille des données (comme dans l'exemple de la multiplication de matrices, donné plus loin), alors ces trois quantités sont confondues. Mais en général, ce n'est pas le cas et l'on ne sait même pas si le coût moyen est plus proche du coût minimal ou du coût maximal (sauf si l'on sait déterminer les fréquences relatives des données qui correspondent à un coût minimal et de celles qui correspondent à un coût maximal).

Remarquons enfin que ce n'est pas parce qu'un algorithme est meilleur qu'un autre en moyenne, qu'il est meilleur dans le pire des cas.

Pour terminer ce paragraphe, on traite deux exemples qui illustrent les définitions introduites :

Exemple A : Multiplication de matrices carrées.

Soit $A = (a_{ij})$ et $B = (b_{ij})$ deux matrices $n \times n$ à coefficients dans $\mathbb{R}$; l'algorithme suivant calcule les coefficients (c_{ij}) de la matrice produit $C = A \times B$ selon la formule classique

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

pour i et j compris entre 1 et n.

```
type matrice = array [1..n, 1..n] of integer;
procedure multmat (a, b : matrice; var c : matrice);
var i, j, k : integer;
begin
    for i := 1 to n do
        for j := 1 to n do
            begin c[i, j] := 0;
                for k := 1 to n do
                    c[i, j] := c[i, j] + a[i, k] * b[k, j]
            end
end multmat;
```

La complexité de l'algorithme *multmat*, comptée en nombre de multiplications de réels ne dépend que de la taille des matrices :

$$\mathrm{Min}(n) = \mathrm{Moy}(n) = \mathrm{Max}(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} 1 = n^3$$

Exemple B : Recherche séquentielle.

On cherche à déterminer la complexité, en nombre de comparaisons, de l'algorithme de recherche séquentielle d'un élément dans une liste, présenté au paragraphe 1.2. D'après l'analyse faite alors, on a $\text{Max}(n) = n$ et $\text{Min}(n) = 1$.

Pour calculer $\text{Moy}(n)$ on doit se donner des probabilités sur L et X :

- soit q la probabilité que X soit dans L;
- on suppose que si X est dans L, toutes les places sont équiprobables.

On note $D_{n,i}$ pour $1 \leq i \leq n$, l'ensemble des données où X apparaît à la $i^{ième}$ place et $D_{n,0}$ l'ensemble des données où X est absent. D'après les conventions ci-dessus on a :

$$p(D_{n,i}) = q/n \qquad \text{et} \qquad p(D_{n,0}) = 1 - q$$

D'après l'analyse de l'algorithme on a :

$$\textit{coût}(D_{n,i}) = i \qquad \text{et} \qquad \textit{coût}(D_{n,0}) = n$$

On a donc :

$$\begin{aligned}\text{Moy}(n) = \sum_{i=0}^{n} p(D_{n,i}) \cdot \textit{coût}(D_{n,i}) &= (1-q) \cdot n + \sum_{i=1}^{n} i.q/n \\ &= (1-q) \cdot n + (n+1) \cdot q/2\end{aligned}$$

Si on sait que X est dans la liste, on a $q = 1$ et :

$$\text{Moy}(n) = (n+1)/2$$

Si X a une chance sur deux d'être dans la liste, on a $q = 1/2$ et :

$$\text{Moy}(n) = n/2 + (n+1)/4 = (3n+1)/4$$

3. Comparaisons de deux algorithmes; ordre de grandeur

On a déterminé la complexité d'un algorithme comme une fonction de la taille des données; il est très important de connaître la rapidité de croissance de cette fonction lorsque la taille des données croît. En effet, pour traiter un problème de petite taille la méthode employée importe peu, alors que pour un problème de grande taille, les différences de performance entre algorithmes peuvent être énormes.

Souvent, une simple approximation de la fonction de complexité suffit pour savoir si un algorithme est utilisable ou non, ou pour comparer entre eux différents algorithmes.

Par exemple, pour n grand, il est souvent secondaire de savoir si un algorithme fait $n+1$ ou $n+2$ opérations.

Parfois les constantes multiplicatives ont, elles aussi, peu d'importance : supposons que l'on ait à comparer l'algorithme A_1 de complexité $M_1(n) = n^2$ et l'algorithme A_2 de complexité $M_2(n) = 2n$. A_2 est meilleur que A_1 pour presque tous les n ($n > 2$); de même si $M_1(n) = 3n^2$ et $M_2(n) = 25n$, A_2 est meilleur que A_1 pour $n > 8$. Quelles que soient les constantes multiplicatives k_1 et k_2 telles que $M_1(n) = k_1 \cdot n^2$ et $M_2(n) = k_2 \cdot n$, l'algorithme A_2 est toujours meilleur que A_1 à partir d'un certain n, car la fonction $f(n) = n^2$ croît beaucoup plus vite que la fonction $g(n) = n$ (en effet $\lim_{n\to\infty} g(n)/f(n) = 0$).

On dit alors que l'***ordre de grandeur asymptotique*** de $f(n)$ est strictement plus grand que celui de $g(n)$.

La figure 1 met en évidence la différence de rapidité de croissance de certaines fonctions usuelles : les ordres de grandeur asymptotiques des fonctions $1, \log_2 n, n\log_2 n, n^2, n^3, 2^n$ vont en croissant strictement; ces fonctions forment une ***échelle de comparaison*** (cf. annexe).

Pour analyser la complexité [1] $M_A(n)$ d'un algorithme A, on s'attache d'abord à déterminer l'ordre de grandeur asymptotique de $M_A(n)$: on cherche dans une échelle de comparaison, éventuellement plus complète que celle qui est formée par les fonctions de la figure 1, une fonction qui a une rapidité de croissance voisine de celle de $M_A(n)$.

Supposons que l'on ait à comparer deux algorithmes A_1 et A_2 de complexités $M_{A_1}(n)$ et $M_{A_2}(n)$[2]. Si l'ordre de grandeur de $M_{A_1}(n)$ est strictement plus grand que l'ordre de grandeur de $M_{A_2}(n)$, alors on peut conclure immédiatement que A_1 est meilleur que A_2 pour n grand. Par contre, si $M_{A_1}(n)$ et $M_{A_2}(n)$ ont même ordre de grandeur asymptotique, il faut faire une analyse plus fine pour pouvoir comparer A_1 et A_2.

Pour comparer les ordres de grandeur asymptotiques des fonctions, on a l'habitude d'utiliser la notion suivante : étant donné deux fonctions f et g de $\mathbb{N}$ dans $\mathbb{R}^+$,

$$f = O(g) \text{ si et seulement si } \exists c \in \mathbb{R}^{+*},\ \exists n_0 \in \mathbb{N} \text{ tel que}$$
$$\forall n > n_0,\ f(n) \leq c.g(n)$$

(1) Il s'agira suivant les cas de $\text{Max}_A(n)$ ou $\text{Moy}_A(n)$ ou $\text{Min}_A(n)$.

(2) Il est clair que pour comparer des algorithmes, on utilise la même mesure de complexité (quelle qu'elle soit) sur chacun d'entre eux.

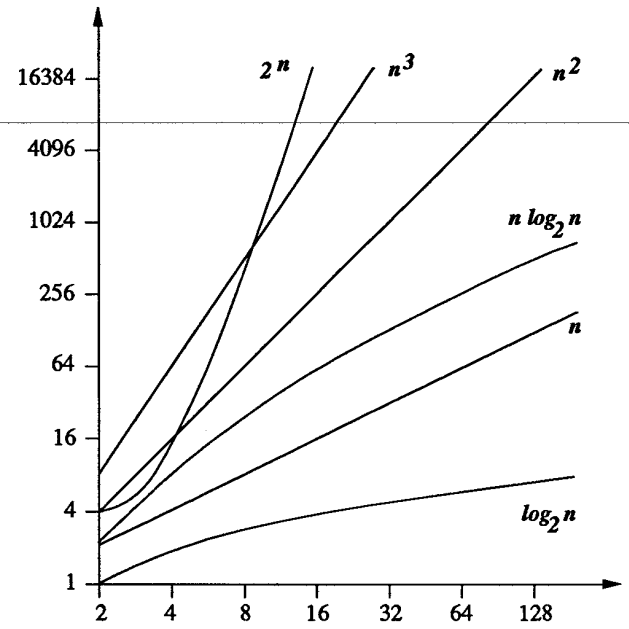


Figure 1. Rapidités de croissance comparées de certaines fonctions usuelles.

Ainsi $f = O(g)$ [3] veut dire que l'ordre de grandeur asymptotique de f est inférieur ou égal à celui de g, on dit aussi que f est ***dominée asymptotiquement*** par g ; par exemple $2n = O(n^2)$, mais aussi $2n = O(n)$ [4].

Cette notion qui donne un majorant de l'ordre de grandeur asymptotique de f, est très utile pour de nombreuses applications, mais elle n'est pas suffisante pour comparer entre elles les performances des différents algorithmes, car il faut connaître les ordres de grandeurs exacts, et non des majorants. Lorsque l'on dit que la complexité $M_A(n)$ d'un algorithme A est en $h(n)$, on veut dire que son ordre de grandeur asymptotique est *exactement* $h(n)$ (i.e. $h(n)$ est le plus petit majorant).

On est donc amené à introduire la notion suivante :

$$f = \Theta(g) \text{ si et seulement si } f = O(g) \text{ et } g = O(f)$$

c'est-à-dire qu'il existe deux réels positifs c et d, et un entier n_0 tels que :

$$\forall n > n_0,\ d \cdot g(n) \leq f(n) \leq c \cdot g(n)$$

On dit que f et g ont ***même ordre de grandeur asymptotique.***

[3] $f = O(g)$ se prononce « f égale grand O de g ». On peut aussi dire « f est en grand O de g ».

[4] Pour une étude plus poussée de cette notion on se reportera à l'annexe «Outils mathématiques».

La notion Θ [5] est plus précise que la notion O. Par exemple, $2n = \Theta(n)$, mais $2n$ n'est pas en $\Theta(n^2)$. Cependant, dans la plupart des ouvrages d'algorithmique, les résultats des analyses sont mis sous la forme $O(f(n))$, alors qu'un décompte précis des opérations fondamentales permet souvent de conclure que la complexité est exactement $\Theta(f(n))$. Par exemple, on dit souvent que le tri par tas (chapitre 15) est en $O(n \log n)$, alors qu'en fait il est en $\Theta(n \log n)$.

Soulignons un point fondamental : les définitions de O et Θ reposent sur l'existence de certaines constantes finies, mais il n'est rien précisé sur la *valeur* de ces constantes. Cela n'a pas d'importance pour obtenir des *résultats asymptotiques* lorsque les fonctions ont des *ordres de grandeur différents* :

par exemple si $f(n) = 2n$ et $g(n) = n^2$, alors $f(n) < g(n)$ pour $n > 2$

si $f(n) = 10000n$ et $g(n) = n^2$, alors $f(n) < g(n)$ pour $n > 10^4$.

Ainsi, si l'ordre de grandeur de f est plus petit que celui de g alors il existe un seuil à partir duquel la valeur de f est c fois plus petite que celle de g, mais on ne sait pas quel est ce seuil.

Par contre, si f et g ont même ordre de grandeur, il devient beaucoup plus difficile de les comparer : la détermination des constantes, et éventuellement des termes d'ordre inférieur nécessite en général des techniques mathématiques beaucoup plus complexes. Il faut bien être conscient de ce que l'obtention de résultats tels que : «l'algorithme A va deux fois plus vite que l'algorithme B sur un ordinateur standard», est en général très difficile, voire impossible.

La notion d'ordre de grandeur de la complexité des algorithmes a une grande importance pratique. Supposons que l'on dispose pour résoudre un problème donné de sept algorithmes dont les complexités dans le cas le pire ont respectivement pour ordre de grandeur 1 (c'est-à-dire une fonction constante, qui ne dépend pas de la taille des données), $\log_2 n$ (c'est-à-dire une fonction logarithmique), n (c'est-à-dire une fonction linéaire), $n.\log_2 n$, n^2 (c'est-à-dire une fonction polynômiale d'ordre 2), n^3 (c'est-à-dire une fonction polynômiale d'ordre 3), 2^n (c'est-à-dire une fonction exponentielle).

Le tableau A donne une estimation du temps d'exécution de chacun de ces algorithmes pour différentes tailles n des données du problème sur un ordinateur pouvant effectuer 10^6 opérations par seconde. Il montre bien que, plus la taille des données est grande, plus les écarts entre les différents temps d'exécution se creusent.

Le tableau B donne une estimation de la taille maximale des données que l'on peut traiter par chacun des algorithmes en un temps d'exécution fixé (et toujours sur un ordinateur effectuant 10^6 opérations par seconde).

---

[5] $f = \Theta(g)$ se prononce «f égale theta de g».

Tableau A. Temps d'exécution.

Complexité / Taille	1	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
$n = 10^2$	$\simeq 1\mu s$	6,6 μs	0,1 ms	0,6 ms	10 ms	1 s	4×10^{16} a
$n = 10^3$	$\simeq 1\mu s$	9,9 μs	1 ms	9,9 ms	1 s	16,6 mn	∞
$n = 10^4$	$\simeq 1\mu s$	13,3 μs	10 ms	0,1 s	100 s	11,5 j	∞
$n = 10^5$	$\simeq 1\mu s$	16,6 μs	0,1 s	1,6 s	2,7 h	31,7 a	∞
$n = 10^6$	$\simeq 1\mu s$	19,9 μs	1 s	19,9 s	11,5 j	$31,7 \times 10^3$ a	∞

N.B. On a noté «∞» lorsque la valeur dépasse 10^{100}.

Tableau B. Taille maximum des données.

Complexité / Temps calcul	1	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
1s	∞	∞	10^6	63×10^3	10^3	100	19
1 mn	∞	∞	6×10^7	28×10^5	77×10^2	390	25
1 h	∞	∞	36×10^8	13×10^7	60×10^3	15×10^2	31
1 jour	∞	∞	86×10^9	27×10^8	29×10^4	44×10^2	36

D'après ces deux tableaux, il est clair que certains algorithmes sont utilisables pour résoudre des problèmes sur ordinateurs, et que d'autres ne sont pas, ou peu utilisables.

Les algorithmes utilisables pour des données de grande taille sont ceux qui s'exécutent en temps :

- constant (c'est le cas de la complexité en moyenne de certaines méthodes de hachage présentées au chapitre 12);
- logarithmique (par exemple la recherche dichotomique présentée au chapitre 9 ou les opérations sur les arbres binaires de recherche présentées au chapitre 10);
- linéaire (par exemple, la recherche séquentielle, vue précédemment, d'un élément dans un tableau non trié);
- $n.\log n$ (par exemple les bons algorithmes de tri présentés au chapitre 15).

Les algorithmes qui prennent un temps polynômial, c'est-à-dire en $\Theta(n^k)$ avec $k > 0$, ne sont vraiment utilisables que pour $k < 2$. Lorsque $2 \leq k \leq 3$, on ne peut traiter que des problèmes de taille moyenne, et lorsque k dépasse 3 on ne peut traiter que des petits problèmes.

Les algorithmes en temps exponentiel, c'est-à-dire en $\Theta(2^n)$ par exemple, sont à peu près inutilisables, sauf pour des problèmes de très petite taille. Ce sont de tels algorithmes que l'on a qualifiés d'inefficaces dans l'introduction.

Tableau C. Evolutions mutuelles du temps et de la taille des données.

Complexité	1	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
Evolution du temps quand la taille est multipliée par 10	t	t+3,32	$10\times t$	$(10+\varepsilon)\times t$	$100\times t$	$1000\times t$	t^{10}
Evolution de la taille quand le temps est multiplié par 10	∞	n^{10}	$10\times n$	$(10-\varepsilon)\times n$	$3{,}16\times n$	$2{,}15\times n$	n+3,32

Le tableau C, enfin, montre comment la taille des données et le temps d'exécution varient en fonction l'un de l'autre. On voit en particulier que si l'on multiplie par 10 la vitesse de calcul de l'ordinateur, on ne modifie quasiment pas la taille maximale des données que l'on peut traiter avec un algorithme exponentiel, alors que l'on multiplie évidemment par 10 la taille des données traitables par un algorithme linéaire. Il est donc toujours d'actualité de rechercher des algorithmes efficaces, même si les progrès technologiques accroissent les performances du matériel !

4. Optimalité

On vient de voir comment comparer des algorithmes. Supposons maintenant que l'on dispose d'un algorithme A pour résoudre un problème donné, il est alors naturel de se demander si l'on peut trouver un algorithme «meilleur» que A; il est donc intéressant de connaître la complexité du meilleur algorithme possible pour traiter un problème.

Soit un problème P, on considère la classe C de tous les algorithmes résolvant P, qui utilisent des opérations d'un certain type, et dont les données sont organisées d'une certaine manière (notons que l'on ne connaît pas nécessairement tous les algorithmes de la classe). On se donne également une mesure de complexité, le nombre d'opérations fondamentales (évalué soit dans le pire des cas, soit en moyenne).

On définit la *complexité optimale* de la classe C, comme la borne inférieure des complexités des algorithmes de la classe. Un algorithme A de C est dit optimal si sa complexité est égale à la complexité optimale de C, que l'on note $M_{\text{opt}}(n)$. Par conséquent, un algorithme A de la classe C est optimal, s'il n'existe pas d'algorithme B dans C qui résolve le problème P en moins d'opérations que A.

Parfois, on ne peut pas déterminer $M_{\text{opt}}(n)$ avec précision, mais on peut connaître *l'ordre de grandeur* de la complexité optimale de la classe. Dans ce cas, un

algorithme A de la classe C est dit optimal si sa complexité est d'ordre de grandeur inférieur ou égal à la complexité de tout algorithme de classe C :

$$\forall B \in C,\ M_A = O(M_B)$$

L'ordre de grandeur de $M_{\text{opt}}(n)$ est alors $\Theta(M_A)$. Il est clair que, dans ce cas, plusieurs algorithmes de même ordre de complexité peuvent être optimaux.

Les techniques de démonstration d'optimalité dépendent énormément du problème et de la classe d'algorithmes étudiés et il n'existe pas de méthode générale pour établir des résultats d'optimalité. Dans ce livre, on en établit plusieurs, pour les problèmes suivants : recherche des deux plus grands éléments d'un tableau, recherche dichotomique, tri par comparaisons. Les techniques utilisées sont présentées dans le chapitre 3.

Il faut savoir que beaucoup de problèmes d'optimalité sont difficiles et qu'un grand nombre n'est pas encore résolu. Prenons l'exemple du produit de deux matrices, réalisé par des algorithmes utilisant les opérations arithmétiques classiques ($+$, $-$, $*$, $/$). On mesure la complexité en prenant la multiplication comme opération fondamentale. On a vu que l'algorithme classique de multiplication de matrices utilise n^3 multiplications; d'autre part, il est établi que la résolution du problème de la multiplication de deux matrices $n \times n$ nécessite au moins n^2 multiplications. Cependant, on ne connaît pas d'algorithme résolvant le problème avec seulement n^2 multiplications, et on ne sait pas s'il en existe. Déterminer la complexité du meilleur algorithme possible pour ce problème est actuellement un problème ouvert.

Chapitre 3
Etude d'un exemple, optimalité

Ce chapitre est consacré au traitement complet de l'analyse de la complexité de deux algorithmes.

Considérons le problème de la recherche des deux plus grands éléments d'une liste L d'entiers. Nous le décomposons en deux sous-problèmes :

1) la recherche du plus grand élément d'une liste,
2) la recherche du deuxième plus grand élément d'une liste.

1. Recherche du plus grand élément d'une liste

On considère une liste d'entiers représentée par un tableau et il s'agit d'en trouver le plus grand élément. On donne ci-dessous la spécification de ce problème et un algorithme simple le résolvant.

```
procedure max(L : array [1..n] of integer; var M : integer);
{la donnée est une liste non vide de n entiers distincts non ordonnés que l'on
représente par un tableau L. On recherche le plus grand entier de L. Le résul-
tat est M qui représente le maximum de L}
var j : integer;
begin
(1)  M := L[1];
(2)  for j := 2 to n do
(3)      if L[j] > M then M := L[j]
end max;
```

Dans cet algorithme, on peut choisir deux opérations fondamentales différentes, qui sont toutes deux significatives pour le problème. On peut compter le nombre $A(n)$ de tests à la ligne (3), c'est-à-dire, le nombre de comparaisons entre éléments du tableau; et l'on peut aussi compter le nombre $B(n)$ d'affectations (on dit aussi déplacement, ou transfert) aux lignes (1) et (3), c'est-à-dire, le nombre de mises à

jour de M, initialisation comprise. La complexité en temps dépend à la fois de $A(n)$ et de $B(n)$. On évalue maintenant ces deux quantités.

1.1. Complexité en nombre de comparaisons

1.1.1. Complexité de l'algorithme en nombre de comparaisons

On voit immédiatement que le nombre de tests à la ligne (3) est égal au nombre d'itérations dans la boucle **for**, soit $A(n) = n - 1$. Remarquons que ce nombre est indépendant de la liste traitée. Quelle que soit la mesure d'évaluation considérée, la complexité est donc la même :

$$\text{Min}_A(n) = \text{Moy}_A(n) = \text{Max}_A(n) = n - 1$$

(Notons qu'ici l'indice A renvoie non pas à un algorithme noté A mais rappelle que l'opération fondamentale choisie est le nombre de tests.)

1.1.2. Optimalité en nombre de comparaisons

On connaît donc un algorithme qui trouve, pour tout tableau de n éléments, le maximum en $n - 1$ comparaisons. On peut se demander si ce nombre peut être amélioré. On va montrer en fait que cet algorithme est optimal.

Considérons la classe C des algorithmes qui résolvent le problème de la recherche du maximum de n éléments en utilisant comme critère de décision, les comparaisons entre éléments. (Ces algorithmes peuvent aussi effectuer des déplacements, mais aucune opération arithmétique n'est autorisée.) On montre, par une analyse directe du problème, que tout algorithme de C effectue au moins $n-1$ comparaisons, quelle que soit la donnée sur laquelle il travaille. Prouvons pour cela le résultat suivant.

Lemme 1 : Etant donné un algorithme E de C et un tableau quelconque, tout élément autre que le maximum est comparé au moins une fois avec un élément qui lui est plus grand.

> *Preuve :* Soit i_0 l'indice du tableau L où se trouve le maximum M, résultat de l'algorithme E. Raisonnons par l'absurde en supposant qu'il existe $j_0 \neq i_0$ tel que $L[j_0]$ n'a pas été comparé avec un élément plus grand que lui. $L[j_0]$ n'a donc pas été comparé avec le maximum $L[i_0]$. Construisons alors le tableau L' identique à L, sauf pour l'indice j_0, où il vaut $M + 1$.
>
> L'algorithme E effectuera les mêmes comparaisons sur L' que sur L, et par suite ne comparera pas $L'[j_0]$ et $L'[i_0]$. Il donnera donc comme maximum $L'[i_0]$ et sera incorrect. D'où la contradiction. □

Il résulte du lemme qu'il est impossible de déterminer l'élément maximum d'un tableau quelconque de n éléments en moins de $n - 1$ comparaisons. L'algorithme du paragraphe 1 est donc optimal en nombre de comparaisons.

1.2. Complexité en nombre d'affectations

Il s'agit ici d'évaluer le nombre $B(n)$ d'opérations d'affectations de M : initialisation à la ligne (1) de l'algorithme, et mises à jour à la ligne (3). Ce nombre de mises à jour est égal au nombre de fois où le test de la ligne (3) est positif : il varie selon la liste sur laquelle s'exécute l'algorithme.

Il est fondamental pour la suite de l'étude de remarquer que $B(n)$ est en fait le nombre de maximums provisoires rencontrés à la lecture de gauche à droite du tableau L. Par exemple, si on lit de gauche à droite le tableau [10, 8, 20, 15, 23, 30, 18, 25] on a les maximums provisoires suivants : 10, 20, 23, 30.

1.2.1. Bornes extrêmes

Lorsque le tableau est trié en ordre croissant, chaque élément rencontré est un maximum provisoire de gauche à droite (en abrégé MPGD); cette configuration donne un nombre maximal de déplacements :

$$\mathrm{Max}_B(n) = n$$

Par ailleurs, pour toutes les listes de données qui commencent par leur plus grand élément (et peu importe l'ordre des autres éléments), l'algorithme ne fait qu'une seule affectation de M, à l'initialisation. Ainsi :

$$\mathrm{Min}_B(n) = 1$$

La détermination des complexités dans le pire et dans le meilleur des cas nous donne un encadrement très large pour la complexité en moyenne :

$$1 \leq \mathrm{Moy}_B(n) \leq n$$

Cependant, intuitivement, il est beaucoup moins probable d'avoir un tableau trié en ordre croissant, qu'un tableau dont le plus grand élément se trouve à la première place (sous l'hypothèse que tous les tableaux sont équiprobables). Cet argument suggère que la fonction $\mathrm{Moy}_B(n)$ a un comportement à l'infini plus proche de la fonction 1 que de la fonction n. Ceci est confirmé par l'étude précise de la valeur de $\mathrm{Moy}_B(n)$ faite au paragraphe suivant.

1.2.2. Complexité en moyenne

Reprenons la définition du chapitre 2 :

$$\mathrm{Moy}_B(n) = \sum_{d \in D_n} p(d)\ coût_B(d)$$

A priori, D_n désigne ici l'ensemble (infini) de toutes les valeurs possibles d'un tableau de n éléments. Cependant, comme d'une part, seul *l'ordre* entre les valeurs

des éléments du tableau est important, (et non pas les valeurs elles-mêmes), et que de plus tous les éléments sont *distincts*, on ne restreint pas le problème en prenant pour D_n l'ensemble des tableaux qui sont des permutations de $[1...n]$. Le nombre $coût_B(d)$ est le nombre de MPGD de la permutation d qui correspond à la configuration du tableau. La figure 1 est une représentation graphique de la permutation $d = (2, 1, 5, 3, 6, 8, 4, 7)$ qui possède quatre MPGD.

On note $p(d)$ la probabilité d'avoir la permutation d. Comme on considère que toutes les permutations sont équiprobables, on a $p(d) = 1/n!$.

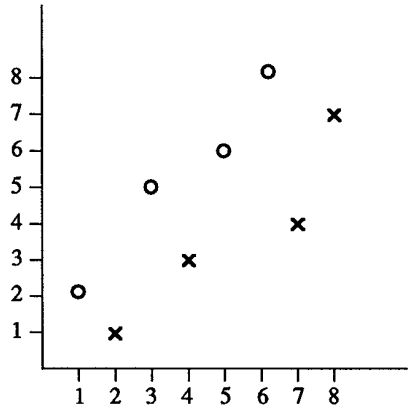


Figure 1. *coût(d) = 4*
(les MPGD sont marqués par des o).

L'évaluation de $\text{Moy}_B(n)$ nécessite quelques lemmes.

Lemme 2 : Soit $P_{n,k}$ le nombre de permutations ayant k MPGD, on a pour $n > 0$

$$\text{Moy}_B(n) = \sum_{k=1}^{n} k \frac{P_{n,k}}{n!}$$

Preuve : Partitionnons l'ensemble D_n en sous-ensembles $D_{n,k}$ tels que :

$D_{n,k} = \{\text{permutations de } [1..n] \text{ ayant } k \text{ MPGD}\}$, pour $k = 1, ..., n$.

Remarquons que le nombre de MPGD est inférieur ou égal à n. D'après la formule vue au chapitre 2, on a :

$$\text{Moy}_B(n) = \sum_{D_{n,k} \subseteq D_n} p(D_{n,k})\ coût_B(D_{n,k})$$

Posons $P_{n,k} = |D_{n,k}|$; alors $p(D_{n,k}) = P_{n,k}/n!$, en raison de l'équiprobabilité des permutations. D'autre part, par définition de $D_{n,k}$, $coût_B(D_{n,k}) = k$.

D'où le résultat. □

Il est difficile d'évaluer directement $P_{n,k}$. On essaie donc d'établir une relation de récurrence.

Lemme 3 : Le nombre de permutations ayant k MPGD vérifie :

$$\begin{cases} P_{1,1} & = 1, P_{n,0} = 0, \text{ pour } n \geq 1 \\ P_{n,k} & = (n-1)P_{n-1,k} + P_{n-1,k-1}, \text{pour } n \geq 2, k \geq 1 \end{cases}$$

Preuve : $P_{1,1} = |D_{1,1}| = 1$; $P_{n,0} = |D_{n,0}| = 0$ pour $n \geq 1$. Ces formules expriment que si le tableau a un seul élément, il y a une opération sur le maximum, son initialisation et que cette opération doit toujours être prise en compte, quelle que soit la taille du tableau.

Pour évaluer $P_{n,k}$ dans le cas général, on décompose l'ensemble $D_{n,k}$ en une union disjointe de sous-ensembles :

$D^i_{n,k}$ = {permutations sur $[1..n]$ ayant k MPGD sur $[1..n-1]$ et telles que $L[n] = i$}, pour $1 \leq i \leq n-1$
$D^n_{n,k}$ = {permutations sur $[1..n]$ ayant $k-1$ MPGD sur $[1..n-1]$ et telles que $L[n] = n$},

Ces n sous-ensembles étant disjoints, on a :

$$|D_{n,k}| = \sum_{i=1}^{n-1} |D^i_{n,k}| + |D^n_{n,k}|$$

Pour $1 \leq i \leq n-1$, $D^i_{n,k}$ est en bijection avec l'ensemble des permutations sur $[1..n-1]$ ayant k MPGD. D'où, pour $1 \leq i \leq n-1$:

$$|D^i_{n,k}| = P_{n-1,k}$$

D'autre part, on a $|D^n_{n,k}| = P_{n-1,k-1}$. On obtient donc la relation de récurrence suivante :

$$P_{n,k} = (n-1) \cdot P_{n-1,k} + P_{n-1,k-1}.$$

La définition de $P_{n,k}$ peut être étendue à toute valeur de k en posant $P_{n,k} = 0$ pour $k > n$. La relation de récurrence est alors vraie pour $n \geq 2$ et $k \geq 1$. □

On résoud cette relation de récurrence en utilisant la série génératrice associée aux $(P_{n,k})_{k \in \mathbb{N}}$ selon un procédé décrit en annexe.

Lemme 4 : Soit $G_n(z) = \sum_{k\geq 0} P_{n,k} \cdot z^k$; alors pour $n \geq 1$, on a :

$$G_n(z) = z.(z+1).....(z+n-1)$$

Preuve : D'après le lemme 3, $G_1(z) = z$. D'autre part, comme $P_{n,0} = 0$ pour $n \geq 1$ et $P_{n,k} = 0$ pour $k > n$, on a $G_n(z) = \sum_{k=1}^{n} P_{n,k} \cdot z^k$, pour tout $n \geq 1$.

Utilisons la relation de récurrence du lemme 3 pour calculer $G_n(z)$ pour $n \geq 2$.

$$\begin{aligned} G_n(z) &= \sum_{k=1}^{n} [(n-1)P_{n-1,k} + P_{n-1,k-1}]z^k \\ &= (n-1)\sum_{k=1}^{n} P_{n-1,k} \cdot z^k + z\sum_{k=1}^{n} P_{n-1,k-1} \cdot z^{k-1} \end{aligned}$$

Or, $P_{n-1,n} = 0$ et $P_{n-1,0} = 0$. Posant $k_1 = k-1$ dans la deuxième sommation, on obtient :

$$G_n(z) = (n-1)\sum_{k=1}^{n-1} P_{n-1,k} \cdot z^k + z\sum_{k_1=1}^{n-1} P_{n-1,k_1} \cdot z^{k_1}$$

Donc : $G_n(z) = (z+n-1).G_{n-1}(z)$, $n \geq 2$.

Cette relation de récurrence se résoud immédiatement :

$$G_n(z) = (z+n-1)(z+n-2)...(z+1)G_1(z)$$

D'où $G_n(z) = (z+n-1)(z+n-2)...(z+1)z$. □

Remarquons que $G_n(1) = \sum_{k=1}^{n} P_{n,k} = n!$ et $G'_n(1) = \sum_{k=1}^{n} k.P_{n,k}$.

La moyenne du nombre de MPGD s'exprime facilement à l'aide de la fonction génératrice $G_n(z)$ et de sa dérivée. Le résultat suivant est établi dans le paragraphe sur les séries génératrices de l'annexe :

$$\text{Moy}_B(n) = \frac{1}{n!}\sum_{k=1}^{n} k.P_{n,k} = \frac{G'_n(1)}{G_n(1)}$$

On a donc la proposition suivante.

Proposition 1 : Le nombre moyen d'affectations de M dans la procédure *max*, qui trouve le plus grand élément d'un tableau de taille n, est $\text{Moy}_B(n) = H_n$. Le nombre $H_n = 1 + \frac{1}{2} + ... + \frac{1}{n}$ est le $n^{ième}$ nombre harmonique.

Preuve : On calcule la dérivée logarithmique de $G_n(z)$.

$$\log G_n(z) = \sum_{i=0}^{n-1} \log(z+i)$$

D'où :

$$\frac{G'_n(z)}{G_n(z)} = \sum_{i=0}^{n-1} \frac{1}{z+i} \text{ et } \frac{G'_n(1)}{G_n(1)} = \sum_{i=1}^{n} \frac{1}{i}$$

On obtient donc : $\text{Moy}_B(n) = H_n$, $n^{ième}$ nombre harmonique. □

Or, H_n est d'ordre de grandeur asymptotique log$n$: $H_n = \Theta(\log n)$. Le lecteur en trouvera une preuve dans le paragraphe de l'annexe consacré à l'asymptotique.

En conclusion, on a donc montré que la complexité moyenne, en nombre d'affectations, pour la recherche du maximum dans une liste de taille n est de l'ordre de log$n$.

2. Recherche du deuxième plus grand élément d'une liste

2.1. Préliminaires

Il s'agit maintenant de trouver aussi le deuxième plus grand élément, noté S, (comme Second), d'une liste de n éléments ($n \geq 2$), non ordonnés, distincts, représentée par un tableau. Il s'agit donc de trouver les indices s et m tels que :

$$L[s] < L[m] \quad \text{avec } L[s] = S \text{ et } L[m] = M$$
$$\text{et} \quad \forall i \in \{1,...,n\} - \{s,m\},\ L[i] < L[s]$$

Commençons notre étude par une remarque importante : pour connaître le deuxième plus grand élément S, il faut connaître le maximum M. En effet, pour être le second, S doit avoir été comparé exactement une fois avec un élément plus grand que lui, à savoir M. En conséquence, tout algorithme qui trouve S doit également trouver M et donc rechercher S est équivalent à rechercher S et M. Il est alors naturel de penser à un algorithme en deux étapes pour trouver S.

1^e étape : utiliser la procédure *max* pour trouver le maximum M de la liste de n éléments.

2^e étape : supprimer M de la liste L et appliquer à nouveau la procédure *max* pour trouver le maximum de cette nouvelle liste de $n-1$ éléments. On obtient alors S.

On peut évaluer facilement le nombre de comparaisons effectuées par cet algorithme :

$n-1$ comparaisons sont requises pour la première étape,
$n-2$ pour la deuxième,

soit au total $2n-3$.

On a vu au paragraphe précédent que l'on ne peut pas faire moins de $n-1$ comparaisons pour trouver le maximum d'une liste de n éléments. On peut aussi se demander si l'on peut faire mieux que $2n-3$ comparaisons pour trouver le second. On peut effectivement améliorer l'algorithme précédent, en utilisant la méthode qui suit.

2.2. Tournoi

On adopte dans ce qui suit une terminologie de jeu : deux éléments comparés seront en fait deux joueurs qui disputent un match. Le plus grand élément sera appelé vainqueur, le plus petit, perdant. On s'intéresse ici essentiellement au nombre de comparaisons (i.e. de matches) effectuées dans les algorithmes résolvant le problème de la recherche de S.

On organise le jeu comme pour un tournoi de tennis : on fait jouer les joueurs deux par deux. Seul le vainqueur reste en lice et joue au tour suivant. A chaque tour, si le nombre de joueurs est impair, on déclare que le dernier joueur est vainqueur et il reste en lice pour le tour suivant. Notons qu'un joueur peut franchir plusieurs tours sans jouer : c'est le cas de $L[9]$ dans l'exemple de la figure 2. On continue à organiser des tours jusqu'à ce qu'il n'y ait plus qu'un seul joueur en lice : c'est le vainqueur.

On peut représenter le tournoi par un arbre binaire à n feuilles[1], dont les nœuds représentent les joueurs. Chaque nœud interne représente le vainqueur du match joué entre ses deux fils. La racine représente le vainqueur final du tournoi. La figure 2, est un exemple d'arbre binaire correspondant à un tournoi entre neuf joueurs.

On a vu que le dernier joueur en lice est le vainqueur M. Regardons à quelles places peut se trouver le second S. Il a forcément perdu une seule fois, en jouant contre M. Donc S doit être cherché parmi les joueurs qui ont perdu contre M.

(1) La terminologie sur les arbres est définie au chapitre 7.

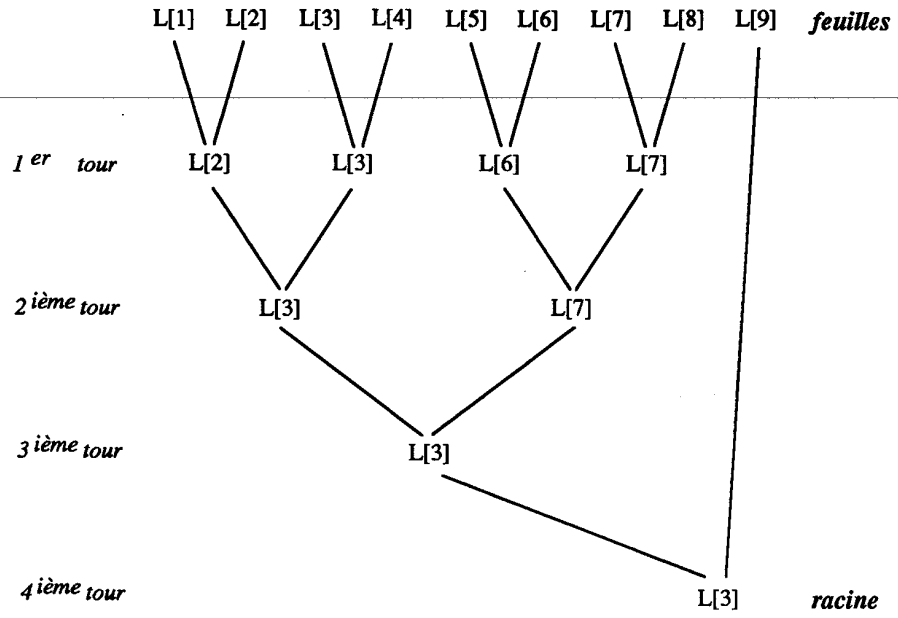


Figure 2. Tournoi avec 9 joueurs.

Sur l'arbre binaire représentant le tournoi, pour trouver les candidats au rang de second, il suffit de regarder le chemin de la racine vers la feuille correspondant au vainqueur, et de prendre pour chaque nœud interne du chemin, le fils qui n'est pas le vainqueur. L'arbre de la figure 2, par exemple, donne la liste de candidats suivante :

$$L' = \{L[9], L[7], L[2], L[4]\}$$

S est alors le plus grand élément de L'. En utilisant la procédure *max*, on trouve S en effectuant $p - 1$ comparaisons, si p est le cardinal de L'. Pour analyser la complexité du tournoi on doit donc étudier la valeur de p.

N.B. On trouvera dans les exercices des indications sur la manière de programmer l'algorithme de tournoi.

2.3. Complexité dans le pire des cas

L'algorithme présenté ci-dessus comporte deux phases : la recherche du vainqueur (à laquelle correspond la construction de l'arbre binaire) et la recherche du meilleur joueur dans la liste L' des candidats à la place de second.

Remarquons que le nombre de comparaisons effectuées lors de la première phase est égal au nombre de nœuds internes de l'arbre. Cet arbre comporte n feuilles, et chacun de ses nœuds a soit 2 fils, soit 0 fils. Il en résulte (cf. chapitre 7) qu'il contient $n-1$ nœuds internes. La première phase comporte donc $n-1$ comparaisons.

Le nombre de comparaisons de la deuxième phase vaut $p-1$, p étant la longueur du chemin issu de la racine qui va vers la feuille correspondant au vainqueur. Si le nombre n de joueurs est une puissance de 2, alors toutes les feuilles de l'arbre tournoi sont au niveau $\log_2 n$. Si le nombre de joueurs n'est pas une puissance de 2, le chemin de la racine au vainqueur est de longueur variable, selon la place du vainqueur. Puisqu'on cherche le nombre de comparaisons dans le pire des cas, il faut déterminer la profondeur h de l'arbre tournoi, c'est-à-dire la longueur (nombre de nœuds moins un) du plus long chemin allant de la racine à une feuille.

Lemme 5 : La profondeur $h(n)$ de l'arbre binaire correspondant à un tournoi avec n joueurs vaut $\lceil \log_2 n \rceil$.

Preuve : Nous prouvons le lemme en établissant par récurrence sur d que :

$$\forall d \geq 0,\ \forall n \geq 2,\ 2^d < n \leq 2^{d+1} \Rightarrow h(n) = d+1$$

Si $d = 0$ alors $n = 2$ et le tournoi se termine en un tour; $h(2) = 1 = d+1$.

Soit l'hypothèse de récurrence :

$$\forall k, 0 \leq k \leq d,\ \forall n \geq 2,\ 2^k < n \leq 2^{k+1} \Rightarrow h(n) = k+1.$$

Soit n quelconque tel que $2^{d+1} < n \leq 2^{d+2}$.

Examinons le déroulement du tournoi avec ces n joueurs. Les 2^{d+1} premiers joueurs organisent un premier tournoi partiel entre eux, qui détermine leur vainqueur M_1. Les $n - 2^{d+1}$ derniers joueurs organisent en même temps un second tournoi partiel (éventuellement sans match si $n = 2^{d+1}+1$) qui détermine leur vainqueur M_2.

Enfin M_1 joue contre M_2, et on obtient le vainqueur M du tournoi total. D'après l'hypothèse de récurrence, le sous-arbre correspondant au premier tournoi partiel est de profondeur $d+1$, tandis que celui qui correspond au deuxième tournoi partiel est de profondeur inférieure ou égale à $d+1$. L'arbre total, compte tenu du match entre M_1 et M_2, a donc une profondeur $d+2$. La récurrence est établie. D'où $h(n) = \lceil \log_2 n \rceil$. □

La deuxième phase de l'algorithme nécessite donc $\lceil \log_2 n \rceil - 1$ comparaisons. On a ainsi établi le résultat suivant :

Proposition 2 : La complexité dans le pire des cas en nombre de comparaisons pour trouver le deuxième plus grand élément parmi n éléments par l'algorithme de

tournoi est :

$$\text{Max}(n) = n + \lceil \log_2 n \rceil - 2$$

On va montrer qu'on a en fait atteint une borne inférieure du nombre de comparaisons requises dans un algorithme de recherche du deuxième plus grand élément.

2.4. Optimalité

On s'intéresse ici à la classe des algorithmes qui résolvent le problème de la recherche du deuxième plus grand élément d'un tableau de taille n, en utilisant comme seul critère de décision, les comparaisons entre éléments. Le résultat d'optimalité obtenu est que tout algorithme de cette classe effectue dans le pire des cas au moins $n + \lceil \log_2 n \rceil - 2$ comparaisons.

On présente deux méthodes différentes qui établissent ce résultat. La première utilise un arbre de décision, la seconde repose sur la constitution d'un oracle.

2.4.1. Optimalité et arbres de décision

La technique d'arbre de décision peut être utilisée pour déterminer des minorants de la complexité optimale de classes d'algorithmes qui résolvent les problèmes suivants :

- *recherche* d'un (ou plusieurs) élément(s) dans une liste d'éléments tous distincts. Le (ou les) éléments sont connus soit par leur valeur exacte, dans le cas d'une liste triée (cf. recherche dichotomique), soit par leur valeur relative dans le cas d'une liste non triée (cf. recherche du maximum, ou du $k^{ième}$ plus grand élément pour $k > 1$).
- *tri* d'une liste d'éléments tous distincts.

Pour tous ces problèmes, cette technique n'est valable que pour la classe des algorithmes qui n'utilisent que des comparaisons entre éléments (ces algorithmes peuvent bien sûr effectuer aussi des affectations mais pas d'opérations sur les valeurs des éléments).

Donnons tout d'abord la définition d'un arbre de décision. (On trouvera à la figure 3 un algorithme simple de recherche du plus grand élément et du deuxième plus grand élément, et l'arbre de décision lui correspondant.)

Définition : A tout algorithme E de la classe considérée, on associe un arbre binaire[2] appelé ***arbre de décision*** qui représente toutes les exécutions différentes de l'algorithme sur toutes les données possibles d'une certaine taille.

[2] Dans le cas de comparaisons pouvant donner plus de deux résultats, les arbres de décision ne sont pas forcément binaires.

• Les feuilles de l'arbre indiquent les résultats de ces différentes exécutions. Deux exécutions différentes peuvent donner le même résultat (cf. figure 3).

• Les nœuds internes de l'arbre représentent les opérations de comparaisons entre éléments, effectuées par cet algorithme E; en particulier, la racine correspond à la première comparaison effectuée par E.

• Pour un nœud interne correspondant à la comparaison des éléments x_i et x_j, si la comparaison effectuée entre x_i et x_j a un résultat vrai alors l'exécution de l'algorithme se poursuit dans le sous-arbre gauche de ce nœud et sinon dans son sous-arbre droit.

• A toute exécution de l'algorithme E sur une donnée correspond une branche de l'arbre. Soulignons aussi le fait que plusieurs données peuvent conduire à la même exécution (cf. figure 3).

• Le nombre de comparaisons effectuées par E pour une donnée d est donc égal à la longueur de la branche correspondant à l'exécution de E sur d.

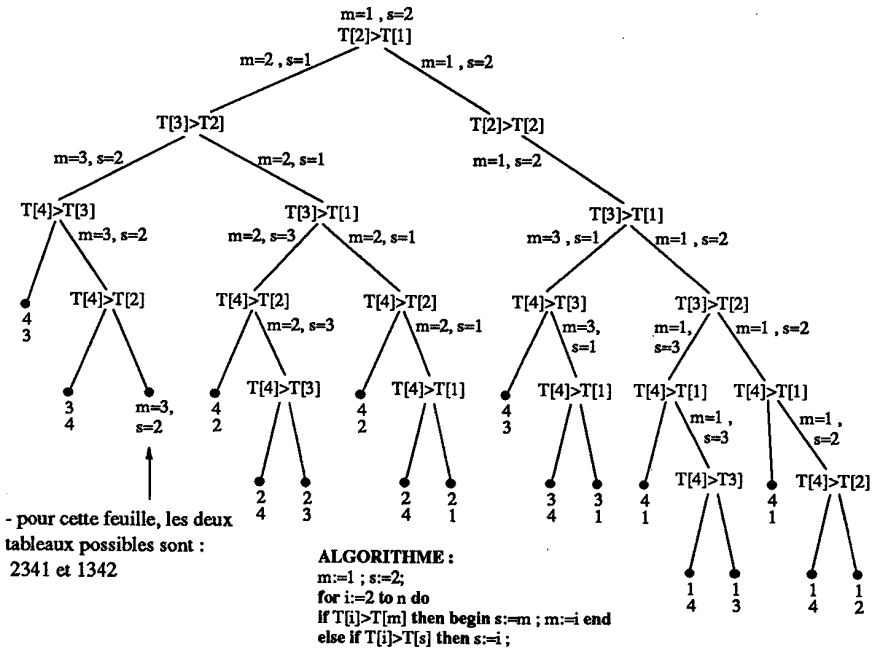


Figure 3. Exemple d'arbre de décision : $n = 4$ et T est une permutation sur $[1..4]$ (les branches correspondant à des exécutions impossibles sont omises).

La profondeur de l'arbre de décision correspondant à un algorithme donné E est donc le nombre maximum de comparaisons effectuées par E.

Considérons l'ensemble AD de tous les arbres de décision correspondant aux algorithmes de la classe C. Rappelons qu'on note $h(A)$, la profondeur d'un arbre A. On obtient que la complexité optimale de la classe dans le pire des cas, notée $M^{\text{opt}}_{\max}(n)$, vaut :

$$M^{\text{opt}}_{\max}(n) = \inf_{A \in AD} h(A)$$

En conséquence, si on connaît un minorant des profondeurs des arbres de AD, on a un minorant de $M^{\text{opt}}_{\max}(n)$.

Dans le cas de la recherche du maximum et du deuxième plus grand élément, à tout algorithme correspond un arbre de décision dont les nœuds internes indiquent les comparaisons entre éléments du tableau et les feuilles les places m et s des deux plus grands éléments M et S.

Proposition 3 : La profondeur k de tout arbre binaire de décision correspondant à un algorithme de recherche du plus grand et du deuxième plus grand élément d'un tableau non trié, de n éléments distincts, basé sur les comparaisons entre éléments est telle que :

$$k \geq n + \lceil \log_2 n \rceil - 2$$

Preuve : Soit E un algorithme de C et soit A son arbre de décision. Considérons toutes les données possibles : ce sont les $n!$ permutations. Partitionnons cet ensemble de données en n classes d'équivalence $(B_i)_{i=1,\ldots,n}$ telles que :

$B_i = \{\text{permutations sur } [1..n] \text{ telles que } M = L[i], 1 \leq i \leq n\}$.

Partitionnons également l'ensemble des feuilles de l'arbre, en n classes d'équivalence X_i telles que deux feuilles sont dans la même classe si, et seulement si, ils donnent le même indice pour le maximum :

$X_i = \{\text{feuilles telles que } m = i\}$.

Notons que $|X_i| \leq |B_i|$ car une feuille peut correspondre à plusieurs données fournissant le même résultat (s, m). Mais il y a autant de classes X_i que de classes B_i et l'ensemble des données de B_i est l'ensemble des permutations correspondant aux nœuds de X_i.

Toutes les permutations appartenant à une même classe B_i ont le même maximum $L[i]$. Donc rechercher le deuxième plus grand élément pour l'une d'entre elles (c'est-à-dire, trouver le nœud de X_i lui correspondant), revient à rechercher le plus grand élément parmi les $n-1$ éléments qui ne sont pas $L[i]$.

Sur l'arbre de décision A, le fait de ne s'intéresser qu'à B_i revient à ne

s'intéresser qu'aux branches de l'arbre qui conduisent à des nœuds de X_i.

L'ensemble de ces branches constitue un sous-arbre partiel A'_i. Par exemple, sur la figure 4, on a matérialisé les branches de A'_3.

Sur chacune des branches de A'_i, les seules comparaisons utiles pour décider du deuxième plus grand élément (i.e. pour savoir à quel nœud de X_i on aboutit), sont évidemment les comparaisons qui ne font pas intervenir le maximum $L[i]$ et les comparaisons non triviales (c'est-à-dire les comparaisons pour lesquelles l'exécution peut effectivement se poursuivre tantôt dans le sous-arbre gauche, tantôt dans le sous-arbre droit). On peut alors construire à partir de A'_i un arbre A_i dont les feuilles sont toujours les éléments de X_i, mais dont les nœuds internes ne contiennent que des comparaisons non triviales permettant de décider de S (cf. figure 4). C'est un arbre dont tout nœud a soit 0, soit 2 fils.

A_i est un arbre de décision correspondant à l'exécution de l'algorithme E sur des données de B_i, en faisant abstraction des comparaisons concernant $L[i]$ et des comparaisons triviales : c'est donc un arbre de décision pour la recherche du plus grand élément parmi $n-1$ éléments (on a exclu $M = L[i]$).

Or, on a vu qu'un algorithme de recherche du maximum pour $n-1$ éléments utilise au moins $n-2$ comparaisons. Donc, tout chemin de A_i a au moins $n-2$ nœuds. Par suite, l'arbre A_i a au moins 2^{n-2} feuilles. D'où $|X_i| \geq 2^{n-2}$, et ceci pour tout $i = 1, ..., n$.

Donc, le nombre total de feuilles est :

$$\sum_{i=1}^{n} |X_i| \geq n \cdot 2^{n-2}$$

On verra au chapitre 7 que tout arbre binaire ayant f feuilles a une profondeur supérieure ou égale à $\lceil \log_2 f \rceil$. Donc la profondeur k est telle que $k \geq \log_2(n.2^{n-2})$ i.e. $k \geq \log_2 n + n - 2$, et comme k est un entier, $k \geq \lceil \log_2 n \rceil + n - 2$. □

On peut ainsi conclure que tout algorithme de recherche des deux plus grands éléments d'un tableau de taille n, basé sur les comparaisons entre éléments, nécessite au moins $n + \lceil \log_2 n \rceil - 2$ comparaisons dans le pire des cas. On a donc trouvé un minorant de la complexité optimale dans le pire des cas. Comme ce minorant est égal à la complexité dans le pire des cas de l'algorithme de tournoi, ce minorant est la borne inférieure. L'algorithme de tournoi du paragraphe 2.2 est optimal.

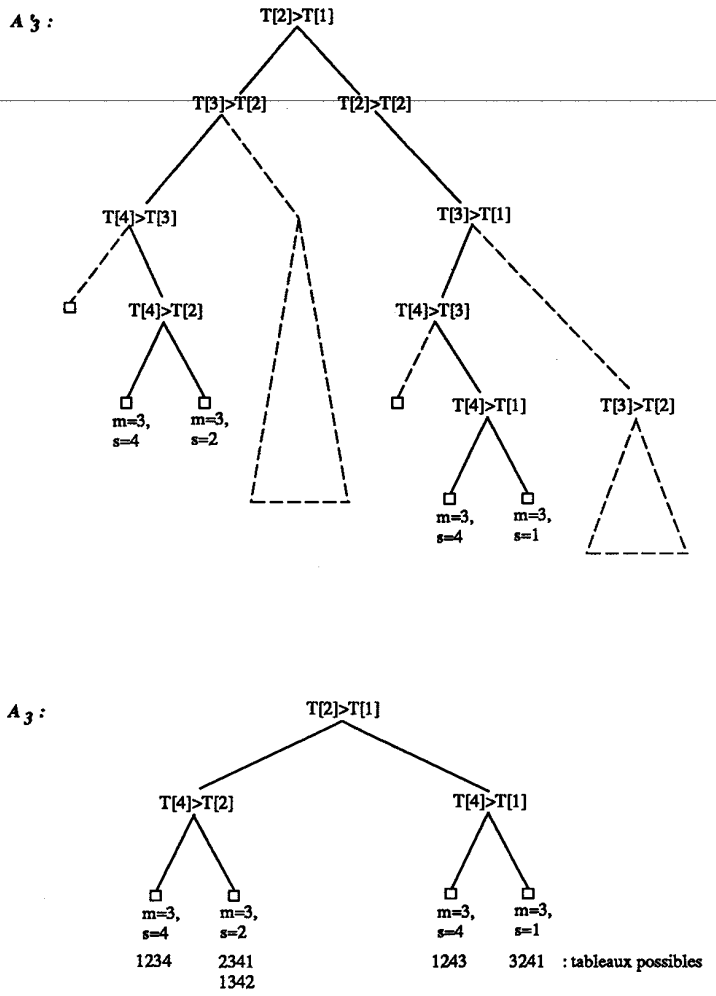


Figure 4. Arbres A'_3 et A_3.

2.4.2. Optimalité et oracle

Dans ce livre, les résultats d'optimalité sont établis en utilisant des arbres de décision. Cependant, il existe d'autres techniques pour démontrer l'optimalité, en particulier, la ***technique d'adversaire*** dite aussi de l'***oracle***. Nous donnons un exemple d'utilisation de cette technique, pour retrouver le résultat d'optimalité, dans le pire des cas, du paragraphe précédent.

Un oracle procède de la façon suivante. Pour tout algorithme d'une classe, il construit une donnée de taille n telle que la complexité de l'algorithme sur cette donnée soit supérieure ou égale à une certaine complexité $c(n)$. Ainsi, la complexité au pire de tout algorithme de la classe est supérieure ou égale à $c(n)$. Donc, tout algorithme ayant pour complexité au pire $c(n)$ est optimal dans cette classe.

Revenons au problème de la recherche du plus grand élément M et du deuxième plus grand élément S, parmi n éléments distincts. On a déjà vu (cf. lemme 1 du §1.1) que trouver M requiert que tout élément autre que M ait perdu dans une comparaison, c'est-à-dire, qu'il y ait au moins $n-1$ comparaisons avec des perdants distincts. Soit p le nombre de ces perdants distincts, qui ont été comparés à M. Pour déterminer S, il faut trouver M et il faut que chacun de ses p adversaires perdants, sauf un, ait perdu dans une comparaison autre que celle avec M. Il faut donc $p-1$ comparaisons en sus.

On reprend la terminologie du jeu exposée précédemment. On va montrer que dans le pire des cas M joue avec $\lceil \log_2 n \rceil$ joueurs au moins.

Proposition 4 : Pour tout algorithme E qui trouve M et S en organisant des matches entre joueurs, il existe une donnée (déterminée par une technique d'adversaire), telle que M joue au moins $\lceil \log_2 n \rceil$ matches contre des joueurs qui n'ont perdu qu'un match, celui contre M.

Preuve : Comme on considère tous les algorithmes de la classe, les joueurs vont jouer deux par deux, n'importe comment, et un perdant n'est pas nécessairement éliminé au tour suivant.

On cherche une donnée $L[1], ..., L[n]$, qui fait gagner un joueur le plus possible. Ce joueur sera le vainqueur, il aura la valeur maximale; il restera à montrer qu'il a joué suffisamment de matches. L'oracle détermine les résultats des matches dans ce sens et par suite les valeurs des joueurs. L'oracle observe les règles suivantes pour chaque match entre deux joueurs x et y.

Règle 1. Si x n'a jamais perdu, (un joueur n'ayant jamais joué est considéré comme n'ayant jamais perdu), et si y a déjà perdu au moins une fois, alors l'oracle déclare x vainqueur et augmente la valeur de x si nécessaire.

Règle 2. Si x et y n'ont jamais perdu, alors l'oracle déclare vainqueur celui des deux qui a gagné le plus de matches jusqu'alors. S'ils ont gagné le même nombre de matches, l'oracle choisit arbitrairement l'un des deux, et ajuste leurs valeurs en conséquence.

Règle 3. Si x et y ont perdu chacun au moins une fois, alors le vainqueur est déterminé par les valeurs déjà attribuées aux joueurs.

Il est toujours possible de choisir une donnée telle que les valeurs des joueurs soient cohérentes avec les trois règles qui viennent d'être prescrites. En effet, les

seuls vainqueurs explicitement déterminés par l'oracle, sont des joueurs invaincus. On peut donc toujours augmenter arbitrairement la valeur de tels joueurs pour qu'ils soient effectivement vainqueurs d'un match, sans contredire les résultats des matches précédents.

Voici un exemple de fonctionnement de l'oracle pour un algorithme A qui calcule le deuxième plus grand élément d'une liste de taille $n = 5$.

> Les valeurs du tableau non initialisées sont notées par un astérisque. Au départ la liste n'est pas initialisée et on a : *, *, *, *, *. Supposons que la première comparaison effectuée par A soit entre $L[2]$ et $L[1]$. Comme ces deux joueurs n'ont jamais perdu et ont gagné le même mombre de matches (à savoir 0), d'après la règle 2, on peut choisir le vainqueur arbitrairement, soit par exemple $L[2]$. Soit $L[1] = 10$ et $L[2] = 20$. La liste est alors : 10, 20, *, *, *. Maintenant A compare $L[3]$ et $L[2]$. Ce sont deux joueurs invaincus mais $L[2]$ a gagné plus de matches que $L[3]$: $L[2]$ est déclaré vainqueur, toujours d'après la règle 2. On choisit pour $L[3]$ la valeur 15. On a la liste : 10, 20, 15, *, *. A compare ensuite $L[3]$ et $L[1]$. Il s'agit de deux perdants. D'après la règle 3, le résultat du match est déterminé par leurs valeurs : $L[3] > L[1]$. Le match suivant a lieu entre $L[4]$ et $L[5]$. Ce sont deux joueurs invaincus qui ont gagné le même nombre de matches (à savoir 0) : $L[4]$ est déclaré vainqueur, par exemple. On donne à $L[4]$ la valeur 40 et à $L[5]$ la valeur 30; on obtient la liste : 10, 20, 15, 40, 30. A compare ensuite $L[4]$ et $L[2]$ qui sont deux gagnants. Comme $L[2]$ a gagné plus de matches que $L[4]$, d'après la règle 1, c'est $L[2]$ qui est déclaré vainqueur. Il faut alors réajuster les valeurs; on sait qu'on peut augmenter sans problème la valeur de $L[2]$: par exemple $L[2] = 50$. La liste est alors : 10, 50, 15, 40, 30. Supposons enfin qu'il y ait un match joué par $L[3]$ et $L[4]$. Ce sont deux joueurs qui ont tous les deux déjà perdu. D'après la règle 3, c'est $L[4]$ qui gagne.

Les résultats du jeu peuvent être représentés par un graphe orienté dont les sommets sont les joueurs. Soit x et y deux sommets, on dessine un arc de x vers y :

> i) si x gagne un match contre y, ou
> ii) si x gagne un match contre z qui a lui-même déjà gagné contre y.

A chaque match, tous les arcs qu'on ajoute sont des arcs issus du sommet correspondant au vainqueur du match. Le nœud correspondant à un joueur n'ayant pas joué le match ne peut être que l'extrémité d'un arc ajouté.

Lemme 6 : A tout joueur x invaincu qui a joué et gagné p matches contre des joueurs jusque-là invaincus, correspond dans le graphe un sommet x dont sont issus au plus $2^p - 1$ arcs.

> *Preuve du lemme 6 :* La propriété est établie par une récurrence sur p. Les arcs issus d'un sommet x sont ajoutés lorsque x gagne.

• Soit $p = 1$: x joue son premier match et gagne. Pour l'adversaire y invaincu jusque-là, d'après la règle 2, c'est aussi son premier match. Du sommet x ne part qu'un seul arc, pointant sur y, soit $2^1 - 1$ arc.

• Supposons la propriété vraie pour tout $k, 1 \leq k \leq p-1$.

Soit x un joueur invaincu qui a joué et gagné $p-1$ matches. Par hypothèse de récurrence, du sommet x partent au plus $2^{p-1} - 1$ arcs. Le joueur x joue son $p^{ième}$ match contre un joueur y invaincu jusque-là, et gagne. D'après la règle 2, y a joué et gagné jusque-là k matches, avec $k \leq p-1$. Par hypothèse de récurrence, du sommet y partent au plus $2^{p-1} - 1$ arcs. D'après i) et ii), après le $p^{ième}$ match, du sommet x partent au plus m arcs avec $m = 1 + 2^{p-1} - 1 + 2^{p-1} - 1$, soit $m = 2^p - 1$. La récurrence est établie. □

Pour terminer la preuve de la proposition 4, on utilise enfin le lemme suivant.

Lemme 7 : Si un joueur est vainqueur à la fin du jeu, alors il lui correspond dans le graphe un sommet duquel partent $n-1$ arcs vers les autres sommets.

Preuve du lemme 7 : Soit x le sommet correspondant au vainqueur M. Tous les autres joueurs ont nécessairement joué et perdu au moins une fois. Soit y un sommet quelconque distinct de x. Construisons par récurrence la suite de sommets $(y_k)_{k \geq 1}$ tels qu'il existe un arc de y_k vers y.

• Soit y_1 le premier joueur qui gagne sur y. D'après la règle 1, y_1 est nécessairement invaincu jusque-là. On dessine un arc de y_1 vers y, d'après i).

• Supposons que la suite $(y_k)_{1 \leq k \leq i}$ est construite, telle que chaque y_k est invaincu jusqu'au moment où on le met dans la suite, et il existe un arc de y_k vers y.

Soit y_{i+1} le premier joueur qui gagne sur y_i. Un tel joueur y_{i+1} existe si y_i n'est pas le vainqueur x. D'après la règle 1, y_{i+1} est nécessairement invaincu. Comme y_{i+1} gagne sur y_i, on dessine un arc de y_{i+1} vers y_i.

D'après la règle ii) de constitution du graphe et l'hypothèse de récurrence, on dessine un arc de y_{i+1} vers y. La construction par récurrence est réalisée. Comme le nombre de sommets est fini, la suite $(y_k)_{k \geq 1}$ est finie et son dernier élément y_p ne peut être que le sommet x correspondant au vainqueur. On a donc bien dessiné un arc de x vers y. De x part un arc vers chacun des $n-1$ autres sommets. Le lemme est donc prouvé. □

Poursuivons la preuve de la proposition 4.

Le vainqueur M doit avoir joué au moins p matches contre des joueurs invaincus, tels qu'après le $p^{ième}$ match, $n-1$ arcs soient issus du sommet M. On a donc

$2^p - 1 \geq n - 1$; c'est-à-dire $p \geq \log_2 n$. Comme p est un entier, on obtient :

$$p \geq \lceil \log_2 n \rceil .$$

(Fin de la preuve de la proposition 4.) □

De la proposition 4, on peut conclure que tout algorithme de recherche du maximum et du deuxième plus grand élément d'un tableau de taille n, basé sur les comparaisons entre éléments, nécessite dans le pire des cas $n + \lceil \log n \rceil - 2$ comparaisons. On retrouve le fait que l'algorithme de tournoi du paragraphe 2.2 est optimal dans le pire des cas.

Deux techniques différentes ont été présentées pour établir l'optimalité de l'algorithme de tournoi. Dans les deux cas, on n'a traité que la complexité au pire.

D'une manière générale, la technique de l'oracle ne donne un minorant que pour la complexité au pire. On verra au chapitre sur les tris que les arbres de décision permettent de minorer également la complexité en moyenne.

Exercices

1. *Programmation de l'algorithme de tournoi.* A chaque étape du tournoi, il faut garder trace de tous les éléments vainqueurs à cette étape, et pour chacun d'entre eux, il faut garder aussi la liste de tous les éléments qu'il a battus. Ainsi, à la fin du tournoi, on aura trouvé l'élément maximal, et l'on aura aussi la liste de ceux qu'il a vaincus (parmi lesquels se trouve le second). Programmer l'algorithme de tournoi de deux façons différentes.

a) En représentant les éléments par une liste chaînée circulaire, chaque élément pointant sur la liste de ses vaincus.

b) En représentant les éléments dans un tableau, et en utilisant un tableau auxiliaire pour stocker les listes des éléments vaincus par des éléments non encore vaincus (voir chapitre 5 pour la représentation de listes chaînées dans un tableau).

Analysez le nombre de comparaisons de ces deux programmes et comparez-les.

Dans les trois exercices qui suivent, on note $W_k(n)$ le nombre minimal de comparaisons dans le pire des cas, pour un algorithme qui détermine la suite ordonnée des k plus grands éléments parmi n éléments, et $V_k(n)$ le nombre minimal de comparaisons dans le pire des cas, pour un algorithme qui détermine le $k^{ième}$ plus grand élément parmi n éléments.

2. En utilisant la technique d'arbre de décision, montrer l'inégalité suivante :

$W_k(n) \geq n - k + \lceil \log_2(n.(n-1).....(n-k+2)) \rceil$, pour k tel que $1 \leq k \leq n$.

3. a) Montrer que si on a trouvé le $k^{ième}$ plus grand élément d'un tableau de n éléments, on connaît les $k-1$ éléments qui lui sont supérieurs.

b) En déduire qu'on peut trouver les k plus grands éléments d'un tableau de n éléments, sans connaître leur ordre respectif, en $V_k(n-1)+1$ comparaisons.

4. Montrer que : $W_3(n) \leq V_3(n)+1$. (Indication : on pourra utiliser l'exercice n° 3.)

5. Dessiner l'arbre de décision correspondant à l'algorithme de tournoi.

6. Soit x et y des nombres réels et n un entier. Evaluer $\lceil \lfloor n \rfloor \rceil$ et montrer que :

a) $\lfloor x \rfloor < n$ si et seulement si $x < n$

b) $n \leq \lfloor x \rfloor$ si et seulement si $n \leq x$

c) $\lceil x \rceil \leq n$ si et seulement si $x \leq n$

d) $n < \lceil x \rceil$ si et seulement si $n < x$

e) $\lfloor x \rfloor = n$ si et seulement si $x - 1 < n \leq x$

f) $\lceil x \rceil = n$ si et seulement si $x \leq n < x + 1$

g) $\lceil n/2 \rceil + \lfloor n/2 \rfloor = n$

h) $\lfloor x \rfloor + \lfloor y \rfloor \leq \lfloor x + y \rfloor$. Montrer qu'on a l'égalité si et seulement si $(x \bmod 1 + y \bmod 1) < 1$.

7. a) Montrer que pour tous entiers n et k tels que $2^{k-1} \leq n < 2^k, k = \lfloor \log_2 n \rfloor + 1$.

b) Montrer que pour tous entiers n et k tels que $2^{k-1} < n \leq 2^k, k = \lceil \log_2 n \rceil$.

c) Vérifier que $\lceil \log_2(n+1) \rceil = \lfloor \log_2 n \rfloor + 1$.

Lectures conseillées pour la première partie

Aho, Hopcroft & Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983.

Baase, *Computer Algorithms, Introduction to Design and Analysis*, Addison-Wesley, 1978.

Horowitz & Sahni, *Fundamentals of Data Structures*, Pitman, 1976.

Knuth, *The Art of Computer Programming*, Vol.1 : *Fundamental Algorithms*, Vol. 3 : *Searching and Sorting*, Addison-Wesley, 1973.

Raynal, *Algorithmes distribués et protocoles*, Eyrolles, 1989.

Sedgewick, *Algorithms*, Addison-Wesley, 1983.

Deuxième partie
Structures de données

Chapitre 4
Types abstraits

La conception d'un algorithme un peu compliqué se fait toujours en plusieurs étapes qui correspondent à des *raffinements successifs*. La première version de l'algorithme est autant que possible indépendante d'une implémentation particulière. En particulier, la représentation des données n'est pas fixée.

A ce premier niveau, les données sont considérées de manière *abstraite* : on se donne une notation pour les décrire ainsi que l'ensemble des opérations qu'on peut leur appliquer et les propriétés de ces opérations. On parle alors de ***type abstrait de données***. La conception de l'algorithme se fait en utilisant les opérations du type abstrait. Les différentes représentations du type abstrait permettent d'obtenir différentes versions de l'algorithme si le type abstrait n'est pas un type du langage de programmation que l'on veut utiliser.

Précisons maintenant à l'aide d'un exemple élémentaire ce qu'est un type abstrait. Considérons le type réel des langages de programmation : on a une convention pour écrire les constantes réelles ; on sait calculer sur les réels en utilisant +, −, *, ... dont les propriétés ne sont pas rappelées car elles sont bien connues. On peut manipuler les réels sans avoir à connaître leur représentation interne (mantisse, exposant). Le manuel du langage indique simplement la précision à utiliser pour tester l'égalité ainsi que le plus grand et le plus petit réels utilisables. Ces points, qui s'écartent des propriétés classiques des réels, sont dus à la représentation finie de ceux-ci en machine : l'utilisateur n'a pas à connaître les détails de cette représentation pour se servir des réels. Tout informaticien utilise donc le concept de type abstrait, parfois comme monsieur Jourdain qui faisait de la prose sans le savoir...

Parfois, la démarche suivie est une *démarche ascendante* : on se donne une *représentation concrète* [1] du type de données en terme d'objets du langage de programmation utilisé, ainsi que des procédures ou des fonctions correspondant aux opérations du type. A partir de là, si on pratique une « bonne » programmation qui consiste à ne plus manipuler les objets du type que par les opérations du type, on

(1) En fait, la distinction entre type abstrait de données et représentation concrète devient relative si l'on conçoit un programme par niveaux d'abstraction successifs : à chaque étape, le type abstrait devient la représentation concrète d'un nouveau type abstrait.

programme en fait avec des types abstraits. Certains langages de programmation comme ADA et CLU fournissent des supports linguistiques pour assurer cette programmation par niveaux.

Cependant, l'informaticien familiarisé avec le concept de type abstrait va pouvoir, lors de la conception d'un algorithme, appliquer une *démarche descendante* : on se donne la définition des types de données (on dit encore leur spécification), et on conçoit l'algorithme à ce niveau. On donne ensuite une représentation concrète des types et des opérations, qui peut être encore un type abstrait, et ceci jusqu'à obtenir un programme exécutable.

Les avantages de cette approche sont multiples : la conception est plus simple, puisqu'on n'a pas à prendre en compte des détails de programmation; elle est faite une fois pour toutes, quelle que soit la représentation concrète choisie ultérieurement pour le type abstrait. Dans ce livre, on applique cette démarche chaque fois que cela se justifie.

Cette approche pose le problème de la définition d'un type abstrait indépendamment de son implémentation. Pour permettre l'utilisation du type, cette définition doit être précise et non ambiguë. C'est ce problème qui va être étudié dans la suite de ce chapitre.

Il existe plusieurs manières de définir un type de données. Toutes ont en commun le concept de *signature*. La signature d'un type de données décrit la syntaxe du type (nom des opérations, type de leurs arguments) mais elle ne définit pas les *propriétés* des opérations du type. C'est par ce dernier aspect que diffèrent les méthodes de définition des types de données. Dans ce livre on a choisi de décrire les propriétés d'un type de données par des axiomes, c'est-à-dire des formules logiques. Cependant, aucune connaissance préalable sur la spécification formelle des types de données n'est requise; tout axiome est abondamment commenté, et certaines propriétés sont données seulement sous forme de commentaires si elles nécessitent des considérations théoriques trop spécialisées ou trop éloignées de notre propos.

1. Signature

La signature d'un type abstrait est la donnée :

- de noms pour un certain nombre d'ensembles de valeurs, par exemple : Booléen, Entier, Liste...; ces noms sont appelés des ***sortes*** (une sorte est un type, au sens des langages de programmation classiques; on introduit ce terme pour éviter des confusions entre *type abstrait* et ensemble de valeurs. Notons que la définition d'un type fait souvent intervenir plusieurs sortes);
- de noms d'un certain nombre d'opérations et de leurs profils; le profil précise à quels ensembles de valeurs appartiennent les arguments et le résultat d'une opération.

On donne ci-dessous un exemple de signature :

sorte Vecteur, Elément, Entier
opérations

ième	:	Vecteur × Entier → Elément
changer-ième	:	Vecteur × Entier × Elément → Vecteur
bornesup	:	Vecteur → Entier
borneinf	:	Vecteur → Entier

Il faut remarquer que cette signature donne une syntaxe du type et ne suffit pas à définir celui-ci. La relative compréhension du lecteur est due au choix des noms et est basée sur son intuition, ce qui manque pour le moins de rigueur. Pour s'en persuader il suffit de considérer la signature ci-dessous, qui est identique, au choix des noms près :

sorte R, S, T
opérations

o	:	$R \times T \rightarrow S$
p	:	$R \times T \times S \rightarrow R$
q	:	$R \rightarrow T$
v	:	$R \rightarrow T$

Il apparaît alors clairement qu'une partie de la définition manque, celle qui donne une sémantique, c'est-à-dire une signification, aux noms R, S, T, o, p, q, v. Ce point est développé au paragraphe 3. Auparavant, on explique comment on écrit une signature.

• Si le nom d'une opération apparaît tel quel dans une signature, comme o, p, q ou v ci-dessus, ce nom est utilisé comme un nom de fonction. Par exemple, si f a le profil $f : T1 \times T2 \rightarrow T3$, et qu'on l'applique aux arguments $a1$ et $a2$ de sortes $T1$ et $T2$, on écrit $f(a1, a2)$.

• Pour éviter trop d'imbrications de parenthèses et pour pouvoir retrouver des notations «habituelles » on s'autorise à préciser, en même temps que son nom, la place des arguments d'une opération. Ces places sont indiquées par des tirets. Par exemple, si on a

$- + -$	: Entier × Entier → Entier
$-!$	: Entier → Entier

on peut écrire des additions et des factorielles sur les entiers comme d'habitude, sans s'interdire pour autant les parenthèses en cas d'ambiguïté :

$i + j$ $\qquad$ $(i + j)!$ $\qquad$ $i + (j!)$

• Une opération qui n'a pas d'arguments est une constante. Par exemple, on note :

0	: → Entier
vrai	: → Booléen

Cela s'explique en considérant l'analogie entre une signature et une syntaxe : une expression syntaxiquement correcte de sorte Entier est soit un zéro, soit deux expressions de sorte Entier séparées par un +, soit une expression de sorte Entier suivie de !, etc. (la signature complète des Entiers n'a pas été donnée).

Pour conclure sur ces notations, on trouvera ci-dessous un exemple de signature pour un type bien connu, celui des booléens :

sorte Booléen
opérations

vrai	:	$\rightarrow$ Booléen
faux	:	$\rightarrow$ Booléen
$\neg -$	:	Booléen $\rightarrow$ Booléen
$- \wedge -$	:	Booléen $\times$ Booléen $\rightarrow$ Booléen
$- \vee -$	:	Booléen $\times$ Booléen $\rightarrow$ Booléen

2. Réutilisation et hiérarchie dans les types abstraits

Il serait peu pratique d'avoir à donner toute la signature et toutes les propriétés des entiers (par exemple) quand on définit les vecteurs (où les entiers servent à numéroter les éléments), les listes, les ensembles d'entiers, bref tout type dont les opérations font intervenir les entiers. La signature donnée précédemment pour les vecteurs est d'ailleurs incomplète car rien n'est dit sur les calculs qu'on peut faire sur les entiers et les éléments.

On se donne donc la possibilité, quand on définit un type, de réutiliser des types déjà définis. Par exemple pour les vecteurs, on aurait dû écrire :

sorte Vecteur
utilise Entier, Elément
opérations

ième	:	Vecteur $\times$ Entier $\rightarrow$ Elément
changer-ième	:	Vecteur $\times$ Entier $\times$ Elément $\rightarrow$ Vecteur
bornesup	:	Vecteur $\rightarrow$ Entier
borneinf	:	Vecteur $\rightarrow$ Entier

Dans ce cas, la signature du type vecteur est l'*union des signatures* des types utilisés, enrichie des nouveaux noms de sortes et d'opérations. On peut donc utiliser des opérations des types déjà définis, par exemple l'addition sur les entiers, ce qui n'était pas le cas dans la version précédente de cette signature. On se limite ici à des unions disjointes. Remarquons que la caractérisation d'une opération étant son nom *et* son profil, cela autorise cependant la *surcharge* des noms d'opérations : + sur les entiers et + sur les réels peuvent apparaître dans une même signature; ce sont des opérations différentes, car elles sont de profil différent.

On introduit par cette construction une *hiérarchie* [2] entre les types. Par exemple, le type Vecteur est au-dessus des types Entier et Elément dans cette hiérarchie. Cette hiérarchie est fondamentale pour structurer les définitions de type abstrait.

Elle permet d'introduire une classification importante parmi les sortes et les opérations d'un type de données. Dans une signature, on appelle ***sorte(s) définie(s)*** la ou les sortes correspondant aux noms de sorte nouveaux : dans l'exemple la seule sorte définie est Vecteur. On appelle ***sorte(s) prédéfinie(s)*** la ou les sortes provenant de types utilisés : ici il s'agit de Entier et Elément.

On dira qu'une opération est ***interne*** si elle rend un résultat d'une sorte définie : c'est le cas ici, de *changer-ième*. Toute valeur d'une sorte définie est le résultat d'une opération interne. On dira qu'une opération est un ***observateur*** si elle a au moins un argument d'une sorte définie et si elle rend un résultat d'une sorte prédéfinie : c'est le cas, ici, de *ième*, *bornesup* et *borneinf*. On va voir que ces définitions sont très utilisées quand on décrit les propriétés d'un type abstrait de données.

On suit ici une règle méthodologique de bon sens : quand on écrit une signature, on donne des opérations internes et des observateurs sur la sorte définie. On n'ajoute pas d'opérations sur les types prédéfinis : par exemple, dans la spécification des vecteurs, on ne définit pas de nouvelles opérations sur les entiers.

3. Description des propriétés d'un type de données

Le problème est de donner une signification (une sémantique) aux noms de la signature : sortes et opérations.

Si on emploie une approche ascendante, on n'a pas ce problème puisqu'on a donné une implémentation des sortes et opérations en terme de sortes et opérations déjà implémentées. On a complètement défini un type de données mais on peut difficilement qualifier ce type d'abstrait. Il s'agit plutôt d'une abréviation que d'une abstraction.

Si on veut définir un type abstraitement, c'est-à-dire indépendamment de ses implémentations possibles, la méthode la plus connue consiste à énoncer les propriétés des opérations sous forme d'***axiomes***. Par exemple :

$$borneinf(v) \leq i \leq bornesup(v) \Rightarrow ième(changer\text{-}ième(v, i, e),\ i) = e$$

où v, i et e sont des variables respectivement de sortes Vecteur, Entier et Elément.

[2] Il faut distinguer clairement cette hiérarchie «d'utilisation» de la hiérarchie de «conception», qui correspond aux démarches ascendante et descendante mentionnées précédemment. Dans ces derniers cas l'ordre hiérarchique est introduit par une relation qu'on pourrait appeler «est représenté par» alors qu'ici la relation est «utilisé».

Cet axiome exprime que, dans la mesure où i est compris entre les bornes d'un vecteur v, quand on construit un nouveau vecteur en donnant au $i^{ième}$ élément la valeur e, et qu'on accède ensuite au $i^{ième}$ élément de ce nouveau vecteur, on obtient e.

Cette propriété est satisfaite quelles que soient les valeurs, de sortes convenables, données aux variables.

Un autre axiome serait :

$$\textit{borneinf}(v) \leq i \leq \textit{bornesup}(v) \;\&\; \textit{borneinf}(v) \leq j \leq \textit{bornesup}(v) \;\&\; i \neq j \Rightarrow \textit{ième}(\textit{changer-ième}(v, i, e),\ j) = \textit{ième}\ (v, j)$$

Cet axiome établit que seul le $i^{ième}$ élément a changé dans le nouveau vecteur.

Ces deux propriétés doivent être satisfaites par toute implémentation du type abstrait vecteur. *La définition d'un type abstrait de données est donc composée d'une signature et d'un ensemble d'axiomes.* Les axiomes sont accompagnés de la déclaration d'un certain nombre de variables. Ce type de définition s'appelle une *définition algébrique* ou *axiomatique* d'un type abstrait. Pour abréger on parle souvent de ***types abstraits algébriques***.

Lorsque l'on écrit, ou lit, des axiomes d'un type abstrait algébrique, on est amené à se poser les questions suivantes : n'y-a-t-il pas d'axiomes contradictoires (problème de *consistance*) ? A-t-on donné un nombre suffisant d'axiomes pour décrire toutes les propriétés du type abstrait que l'on voulait spécifier au départ (problème de *complétude*) ?

Un exemple d'inconsistance serait, dans le cas des vecteurs d'entiers, de trouver une expression v de sorte Vecteur et une expression entière i telles que l'on puisse démontrer, en utilisant les axiomes, les deux propriétés :

$$\textit{ième}(v,\ i) = 0 \qquad \text{et} \qquad \textit{ième}(v,\ i) = 1$$

La notion de complétude est plus délicate et demande à être examinée avec soin quand on travaille sur des types abstraits. En mathématiques, une théorie T, et par suite le système d'axiomes qui la définit, est dite complète si elle est consistante et si pour toute formule P sans variable, on sait démontrer soit P, soit $\neg P$.

Cette notion est trop forte pour les types abstraits algébriques : elle entraîne que toute égalité de deux expressions sans variable doit être soit vraie, soit fausse. Or souvent on veut laisser une latitude aux futurs implémenteurs. Prenons l'exemple des vecteurs. Supposons que l'on applique *changer-ième* à un vecteur v avec pour arguments 5 et a, puis 10 et b. Considérons le vecteur obtenu si on change l'ordre des deux opérations. A priori on a envie de dire que les deux résultats sont égaux. Mais cela peut ne pas être vrai pour certaines implémentations : par exemple une liste chaînée des couples < indice, élément >, où *changer-ième* fait simplement une adjonction d'un nouveau couple en tête de la liste. En fait ce qui est important c'est que ces deux vecteurs, quand on leur applique *ième*, rendent toujours les mêmes résultats. Le fait qu'ils soient égaux ou non n'est pas essentiel.

Le critère utilisé pour les types abstraits algébriques est la ***complétude suffisante*** : les axiomes doivent permettre de déduire une valeur d'une sorte prédéfinie, pour toute application d'un observateur à un objet d'une sorte définie.

Comme on obtient ces objets par les opérations internes, il faut écrire des axiomes qui définissent le résultat de la composition des observateurs avec toutes les opérations internes. De tels axiomes ont été donnés pour *ième* et *changer-ième*, dans l'exemple des vecteurs.

Cependant la règle qui vient d'être énoncée doit être modulée : il existe des types de données où certaines opérations sont des fonctions partielles, non définies partout. C'est par exemple le cas du sommet d'une pile (cf. chapitre 5), qui n'est pas défini pour une pile vide, de l'accès à une place non initialisée dans le cas des tableaux Pascal (qui contiennent des vecteurs). On reformule donc la règle ci-dessus en disant que l'on doit pouvoir déduire *une valeur pour tous les observateurs sur tout objet d'une sorte définie appartenant* ***au domaine de définition*** *de cet observateur.* Le domaine de définition d'une opération partielle est défini par une ***précondition.***

Revenons à l'exemple des vecteurs. Les axiomes donnés précédemment définissent l'opération *ième* suffisamment par rapport à l'opération *changer-ième.* Cependant on n'a pas la possibilité avec la signature actuelle d'écrire une expression de sorte Vecteur sans variable : la seule opération interne est *changer-ième* qui prend en argument un vecteur... Il faut donc ajouter une opération interne qui correspond au contenu d'un vecteur non initialisé mais dont on connaît les bornes :

$$vect \ : \ \text{Entier} \times \text{Entier} \rightarrow \text{Vecteur}$$

avec les axiomes

$$borneinf(vect(i,j)) = i$$
$$bornesup(vect(i,j)) = j$$

Il n'y aura pas d'axiomes définissant $ième(vect(i,\ j),\ k)$ puisque *vect* retourne un vecteur où aucun élément n'est défini. Par contre, il faut écrire la précondition sur l'opération *ième.* Pour cela on a besoin d'une opération auxiliaire sur les vecteurs, qui permet de savoir si un élément a été associé à un certain indice :

$$init \ : \ \text{Vecteur} \times \text{Entier} \rightarrow \ \text{Booléen}$$

avec les axiomes :

$$init(vect(i,j),\ k) = \text{faux}$$
$$(borneinf(v) \leq i \leq bornesup(v)) \Rightarrow (init(changer\text{-}ième(v,\ i,\ e),\ i) = \text{vrai})$$
$$(borneinf(v) \leq i \leq bornesup(v) \quad \& \quad i \neq j) \Rightarrow$$
$$(init(changer\text{-}ième(v,\ i,\ e),\ j) = init(v,j))$$

L'opération *ième* est définie si, et seulement si :

$$borneinf(v) \leq i \leq bornesup(v) \quad \& \quad init(v,i) = \text{vrai}$$

La formule ci-dessus est appelée une précondition sur l'opération *ième*.

Il manque les définitions des observateurs *bornesup* et *borneinf* sur le résultat de l'opération interne *changer-ième* :

$$borneinf(changer\text{-}ième(v,\ i,\ e)) = borneinf(v)$$
$$bornesup(changer\text{-}ième(v,\ i,\ e)) = bornesup(v)$$

La définition finale du type Vecteur est donnée Figure 1. Elle est suffisamment complète : tout vecteur est le résultat d'une opération *vect* et d'une suite d'opérations *changer-ième*; les axiomes permettent de déduire le résultat de *init*, *bornesup* et *borneinf* dans tous les cas; pour ce qui est de *ième*, on peut établir son résultat, en

sorte Vecteur
utilise Entier, Elément
opérations

vect	:	Entier × Entier → Vecteur
changer-ième	:	Vecteur × Entier × Elément → Vecteur
ième	:	Vecteur × Entier → Elément
init	:	Vecteur × Entier → Booléen
borneinf	:	Vecteur → Entier
bornesup	:	Vecteur → Entier

précondition
ième$(v,\ i)$ **est-défini-ssi**
$borneinf(v) \leq i \leq bornesup(v)$ & $init(v,i) =$ vrai

axiomes
$borneinf(v) \leq i \leq bornesup(v) \Rightarrow$
$ième(changer\text{-}ième(v,\ i,\ e),\ i) = e$
$borneinf(v) \leq i \leq bornesup(v)$ & $borneinf(v) \leq j \leq bornesup(v)$ & $i \neq j \Rightarrow$
$ième(changer\text{-}ième(v,\ i,\ e),\ j) = ième(v,j)$

$init(vect(i,j),k) =$ faux
$borneinf(v) \leq i \leq bornesup(v) \Rightarrow$
$init(changer\text{-}ième(v,i,e),i) =$ vrai
$borneinf(v) \leq i \leq bornesup(v)$ & $i \neq j \Rightarrow$
$init(changer\text{-}ième(v,\ i,\ e),\ j) = init(v,j)$

$borneinf(vect(i,\ j)) = i$
$borneinf(changer\text{-}ième(v,\ i,\ e)) = borneinf(v)$

$bornesup(vect(i,\ j)) = j$
$bornesup(changer\text{-}ième(v,\ i,\ e)) = bornesup(v)$

avec
v : Vecteur; i,j,k : Entier; e : Elément

Figure 1. Le type abstrait algébrique Vecteur.

utilisant les axiomes, quand la précondition est satisfaite, c'est-à-dire quand une des opérations *changer-ième* a pour argument l'indice en argument de *ième.*

On prend la convention que si une opération est partielle, sa précondition, convenablement instanciée, est implicitement en prémisse de tout axiome où apparaît cette opération. Autrement dit, les axiomes doivent être satisfaits pour toutes les valeurs des variables de sortes convenables vérifiant les préconditions sur les opérations qui y apparaissent.

En conclusion, un critère pour savoir si on a écrit suffisamment d'axiomes est : peut-on déduire de ces axiomes le résultat de chaque observateur sur son domaine de définition?

De même, pour vérifier qu'on n'a pas écrit d'axiomes contradictoires, il faut vérifier que cette valeur est unique : il serait en effet gênant, comme on l'a vu, de pouvoir montrer que la borne inférieure d'un vecteur est égale à 0, et qu'elle est également égale à 1... Cela impliquerait que 0 est égal à 1, et à partir de là on pourrait démontrer n'importe quoi : on dit alors que les axiomes sont inconsistants.

4. Types abstraits et programmation

A l'issue d'une conception descendante, on veut obtenir l'algorithme dans un langage de programmation. Dans ce livre, il s'agit du langage proche de Pascal décrit au chapitre 1. Il faut donc établir une correspondance entre les types abstraits utilisés pour concevoir l'algorithme et les types du langage de programmation. On présente dans les chapitres suivants des représentations concrètes classiques pour les principaux types abstraits utilisés dans ce livre, qui sont :

1) les *structures séquentielles*, essentiellement les listes, et les cas particuliers de celles-ci que sont les piles et les files;
2) les *ensembles*, en se limitant aux cas où les éléments sont pris dans un domaine fini ou quand les ensembles contiennent peu d'éléments;
3) les *structures arborescentes* : arbres binaires et arbres planaires généraux;
4) *les structures relationnelles ou graphes.*

Auparavant, on rappelle quelques principes de la représentation des objets dans un langage de programmation du type Pascal.

Un tel langage comporte d'une part des types de base (boolean, integer, real, char) dont les objets ont une représentation en mémoire bien définie, et d'autre part des constructeurs (array, record, ↑) qui permettent de combiner entre eux les types de bases pour former des types plus complexes. Dans l'implémentation ces combinaisons correspondent à des groupements de cellules mémoires.

Il y a essentiellement deux façons de grouper les cellules : par contiguïté et par chaînage. Dans les groupements contigus on assemble des cellules qui sont côte à

côte dans la mémoire : tableaux, enregistrements. Dans les groupements chaînés on lie entre elles, à l'aide de pointeurs, des cellules situées un peu partout en mémoire. Un pointeur est une cellule dont le contenu est la désignation (l'adresse) d'une autre cellule.

On va retrouver ces deux approches pour représenter les différents types étudiés dans les chapitres suivants, et l'on montrera comment l'organisation de la mémoire influe sur la complexité des algorithmes.

Lectures conseillées pour le chapitre 4

Liskov & Guttag, *Abstraction and Specification in Program Development*, MIT Press & McGraw-Hill, 1986.

"Algebraic Specifications", chapitre 22 in : *The Software Engineer's Reference Book*, McDermid ed., Butterworth Scientific Ltd, 1990.

Chapitre 5
Structures séquentielles

On présente ici les ***listes linéaires*** qui sont la forme la plus commune d'organisation des données. On organise en liste linéaire des données qui doivent être *traitées séquentiellement.* De plus une liste est évolutive, c'est-à-dire qu'on veut pouvoir *ajouter et supprimer* des données. Deux cas particuliers des listes jouent un rôle important en informatique : les ***piles***, où les données sont ajoutées et supprimées à une même extrémité (le sommet); les ***files***, où les données sont ajoutées à une extrémité de la liste (la queue) et supprimées à l'autre extrémité (la tête). On donne dans ce chapitre différentes représentations des listes linéaires, des piles et des files, et on les compare.

1. Les listes

Définition : Une ***liste linéaire*** λ est une suite finie, éventuellement vide, d'*éléments* repérés selon leur rang dans la liste : $\lambda =< e_1, ..., e_n >$.

Soit E l'ensemble des éléments, l'ensemble $\mathbb{L}$ des listes peut se définir récursivement par :

$$\mathbb{L} = \emptyset + E \times \mathbb{L}$$

L'ordre des éléments dans une liste est fondamental. Il faut remarquer que ce n'est pas *un ordre sur les éléments*, mais un *ordre sur les places des éléments.* Ces places sont totalement ordonnées, c'est-à-dire, qu'il existe une fonction de succession, *succ*, telle que toute place est accessible en appliquant *succ* de manière répétée à partir de la première place de la liste. Pour toute place p d'une liste λ non vide on a donc :

$$\exists k \geq 0 \text{ tel que } p = succ^k(tête(\lambda)) \qquad (1)$$

où $tête(\lambda)$ indique la première place de λ. Chaque place a un *contenu*, qui est un élément.

Le nombre n d'éléments (donc de places) de λ est appelé la *longueur* de λ. Si $n = 0$ la liste ne contient aucun élément : elle est *vide*. D'autre part la fonction *succ* n'est pas définie pour la $n^{ième}$ place, ce qui revient à dire que :

$succ^n(tête(\lambda))$ n'est pas défini.

Parmi les schémas de traitement mis en œuvre par les algorithmes qui manipulent des listes, le plus usuel consiste à examiner séquentiellement, dans l'ordre des places, toutes les places d'une liste non vide pour appliquer le même traitement à leur contenu :

```
x := tête(λ); traiter(contenu(x));
for i:= 1 to longueur(λ) − 1 do begin
    x := succ(x); traiter(contenu(x))
end;
```

En plus du traitement séquentiel des éléments en suivant l'ordre des places, les opérations de base que l'on effectue sur les listes sont les suivantes :

- accéder au $k^{ième}$ élément;
- supprimer le $k^{ième}$ élément;
- insérer un nouvel élément à la $k^{ième}$ place.

1.1. Le type abstrait «Liste itérative»

Si on suit ce qui vient d'être dit, la signature du type Liste est la suivante :

sorte Liste, Place
utilise Entier, Elément
opérations

liste-vide	:	→ Liste
accès	:	Liste × Entier → Place
contenu	:	Place → Elément
longueur	:	Liste → Entier
supprimer	:	Liste × Entier → Liste
insérer	:	Liste × Entier × Elément → Liste
succ	:	Place → Place

Selon la terminologie du chapitre précédent, Liste et Place sont les sortes définies. Les opérations *accès* et *succ* sont les opérations internes de Place; *contenu* est un observateur de Place. Les opérations *liste-vide*, *supprimer* et *insérer* sont les opérations internes de Liste; *accès* et *longueur* sont les observateurs de Liste.

Les opérations ci-dessus ne sont pas définies partout. On a les préconditions suivantes :

préconditions

accès(λ, k) **est-défini-ssi** $1 \leq k \leq$ *longueur*(λ)
supprimer(λ, k) **est-défini-ssi** $1 \leq k \leq$ *longueur*(λ)
insérer(λ, k, e) **est-défini-ssi** $1 \leq k \leq$ *longueur*(λ) + 1
{k = *longueur* (λ) + 1 *correspond à l'ajout en fin de liste*}

Ces opérations satisfont les axiomes suivants, où λ est de sorte Liste, k est de sorte Entier et e est de sorte Elément :

longueur(*liste-vide*) = 0
$\lambda \neq$ *liste-vide* & $1 \leq k \leq$ *longueur*(λ) $\Rightarrow$
longueur(*supprimer*(λ, k)) = *longueur*(λ) − 1
$1 \leq k \leq$ *longueur*(λ) + 1 $\Rightarrow$
longueur(*insérer*(λ, k, e)) = *longueur*(λ) + 1

Ces trois axiomes définissent suffisamment, et de manière consistante l'opération *longueur*.

Il faudrait maintenant donner les axiomes pour l'observateur *accès*. Cependant le résultat de cet observateur n'est pas très intéressant : ce qui est vraiment caractéristique d'une liste c'est le contenu de la $i^{ième}$ place.

On se donne donc un nouvel observateur, *ième*, de profil :

ième : Liste × Entier → Elément

et tel que

ième(λ, k) = *contenu*(*accès* (λ, k))

Cette opération a la même précondition que l'opération *accès*, et vérifie les axiomes ci-dessous, où λ est de sorte Liste, k, i sont de sorte Entier et e est de sorte Elément :

$\lambda \neq$ *liste-vide* & $1 \leq k \leq$ *longueur*(λ) & $1 \leq i < k \Rightarrow$
ième(*supprimer*(λ, k), i) = *ième*(λ, i)
$\lambda \neq$ *liste-vide* & $1 \leq k \leq$ *longueur*(λ) & $k \leq i \leq$ *longueur*(λ) − 1 $\Rightarrow$
ième(*supprimer*(λ, k), i) = *ième*($\lambda, i + 1$)

$1 \leq k \leq$ *longueur*(λ) + 1 & $1 \leq i < k \Rightarrow$
ième(*insérer*(λ, k, e), i) = *ième*(λ, i)
$1 \leq k \leq$ *longueur*(λ) + 1 & $k = i \Rightarrow$
ième(*insérer*(λ, k, e), i) = e
$1 \leq k \leq$ *longueur*(λ) + 1 & $k < i \leq$ *longueur*(λ) + 1 $\Rightarrow$
ième(*insérer*(λ, k, e), i) = *ième*(λ, $i - 1$)

L'opération *ième* n'étant pas définie pour les listes vides, elle est suffisamment définie par les axiomes ci-dessus.

Enfin, on a l'axiome suivant sur les places :

$\lambda \neq$ *liste-vide* & $1 \leq k < longueur(\lambda) \Rightarrow$
$succ(accès(\lambda, k)) = accès(\lambda, k+1)$

Cet axiome définit *succ* sur toutes les places accessibles de λ pour lesquelles cette opération est définie, et implique la propriété (1) donnée au début de ce chapitre.

1.2. Le type abstrait «Liste récursive »

On aurait pu se donner une autre définition des listes en prenant comme opérations de base non plus l'accès et l'insertion à la $k^{ième}$ place, mais plutôt l'opération *tête* qui rend la première place d'une liste et l'opération *fin* qui donne la liste amputée de sa première place. On obtient alors :

sorte Liste, Place
utilise Elément, Entier
opérations

liste-vide	:	→ Liste
tête	:	Liste → Place
fin	:	Liste → Liste
cons	:	Elément × Liste → Liste
premier	:	Liste → Elément
contenu	:	Place → Elément
succ	:	Place → Place

préconditions
tête(λ) **est-défini-ssi** $\lambda \neq$ *liste-vide*
fin(λ) **est-défini-ssi** $\lambda \neq$ *liste-vide*
premier(λ) **est-défini-ssi** $\lambda \neq$ *liste-vide*

axiomes
$\lambda \neq$ *liste-vide* $\Rightarrow$ *premier*(λ) = *contenu*(*tête*(λ))
fin(*cons*(e, λ)) = λ
premier(*cons*(e, λ)) = e
$\lambda \neq$ *liste-vide* $\Rightarrow$ *succ*(*tête*(λ)) = *tête*(*fin*(λ))

avec
λ : Liste; e : Elément

Cette formulation permet de décrire plus facilement les traitements récursifs des listes alors que la précédente est plus commode pour les traitements itératifs. Au demeurant ces définitions sont très proches puisque l'on peut définir *tête*, *fin* et *cons* en terme des opérations *accès*, *insérer* et *supprimer* et que l'inverse est également vrai. Par exemple on a : *tête*(λ) = *accès*(λ, 1) (cf. exercices). On peut aussi définir les listes récursives «en arrière» en se donnant les opérations *dernier* et *début*... (cf. exercices).

1.3. Extensions du type liste

On a souvent à effectuer sur les listes des opérations plus complexes que celles vues jusqu'à présent, par exemple la concaténation de deux listes, ou la recherche d'un élément dans une liste. Ces opérations se définissent à partir des opérations de base données ci-dessus. On les appelle des *extensions* du type. Il est intéressant de constater que la concaténation se formule plus facilement pour les listes itératives que pour les listes récursives.

1.3.1. Opération de concaténation de deux listes

Cette opération a pour profil

concaténer : Liste × Liste → Liste

Elle consiste à construire une liste en mettant deux listes bout à bout. On peut la définir par les axiomes suivants, où λ et λ' sont de sorte Liste et i est de sorte Entier :

longueur(*concaténer*(λ, λ')) = *longueur*(λ) + *longueur*(λ')
$1 \leq i \leq$ *longueur*(λ) ⇒ *ième* (*concaténer*(λ, λ'), i) = *ième*(λ, i)
longueur(λ) + 1 ≤ $i \leq$ *longueur*(λ) + *longueur*(λ') ⇒
ième(*concaténer*(λ, λ'), i) = *ième*(λ', $i-$ *longueur*(λ))

Ces axiomes sont suffisamment complets puisque la valeur des observateurs *longueur* et *ième* peut être déduite pour *concaténer*(λ, λ'), si on connaît la valeur de ces observateurs pour λ et λ'.

On aurait aussi pu définir l'opération *concaténer* récursivement sur les opérations *liste-vide* et *cons* comme suit :

concaténer(*liste-vide*, λ') = λ'
concaténer(*cons*(e, λ), λ') = *cons*(e, *concaténer*(λ, λ'))

1.3.2. Opération de recherche d'un élément dans une liste

Cette opération consiste à rechercher si un élément est présent dans une liste et, dans ce cas, à retourner la place de cet élément. Le profil de cette opération est :

rechercher : Liste × Elément → Place

On prend ici la convention que cette opération n'est pas définie si l'élément recherché n'est pas présent dans la liste. Pour écrire la précondition sur *rechercher*, on se donne l'opération suivante :

est-présent : Liste × Elément → Booléen

avec les axiomes :

est-présent(*liste-vide*, e) = faux
$e = e'$ ⇒ *est-présent*(*cons*(e, λ), e') = vrai
$e \neq e'$ ⇒ *est-présent*(*cons*(e, λ), e') = *est-présent*(λ)

Moyennant quoi, on a la précondition et l'axiome suivants pour *rechercher* :

rechercher(λ, e) **est-défini-ssi** *est-présent*(λ, e) = vrai
est-présent(λ, e) = vrai ⇒ *contenu*(*rechercher*(λ, e)) = e

Il faut noter qu'en cas de répétition de e dans λ, cette spécification ne précise pas de quelle occurrence de e on retourne la place.

Si on veut préciser qu'on cherche la première occurrence comme au chapitre 2, on peut écrire, en les utilisant le type *liste récursive* (pour le type *liste itérative* voir les exercices) :

$e = e'$ ⇒ *rechercher*(*cons*(e, λ), e') = *tête*(λ)
$e \neq e'$ ⇒ *rechercher*(*cons*(e,λ), e') = *rechercher*(λ, e')

L'algorithme étudié dans le chapitre 2 est une implémentation de l'opération *rechercher* sur une représentation des listes par un tableau. On discute au paragraphe suivant les diverses représentations des listes.

2. Représentation des listes

2.1. Représentation contiguë

La liste est représentée par un tableau dont la $i^{ième}$ case est la $i^{ième}$ place de la liste. Il est clair que dans ce cas la longueur de la liste ne peut dépasser la taille du tableau, qui est surdimensionné. Pour ne pas prendre en compte tous les éléments présents dans le tableau, mais seulement ceux de la liste, on a besoin de connaître la longueur de la liste. La liste est donc représentée par un couple <tableau, entier> (voir figure 1); on a le type Pascal suivant :

```
type LISTET = record t : array [1 .. lmax] of Elément;
                     longueur : 0 .. lmax
              end;
```

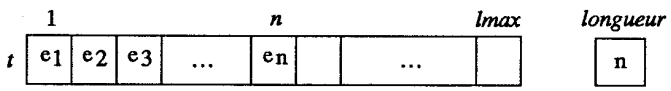


Figure 1. Représentation contiguë d'une liste.

N.B. On n'est pas obligé d'utiliser un type enregistrement : il est possible de représenter une liste par deux variables distinctes, une pour le tableau des éléments, l'autre pour la longueur. Les notations sont plus commodes dans les programmes, mais la même entité est représentée par deux objets distincts, ce qui peut être source d'erreurs.

Soit L une variable de type LISTET. Si L représente la liste vide, on a $L.longueur = 0$. L'expression $L.t[i]$ dénote, selon le contexte, l'accès à la $i^{ième}$ place de la liste représentée par L, ou le $i^{ième}$ élément de cette liste. L'opération *succ* est représentée par la succession des cases du tableau en mémoire.

Avec cette représentation il est simple d'accéder au $k^{ième}$ élément d'une liste, et de parcourir séquentiellement une liste. Il est également facile d'insérer ou de supprimer le dernier élément; cependant lorsque l'élément à insérer ou supprimer est à la $k^{ième}$ place d'une liste L, avec $k < L.longueur$, il faut déplacer tous les éléments de la place d'indice k à la place d'indice $L.longueur$ dans le tableau pour reconstituer la structure.

```
procedure supprimer (var L : LISTET; k : 1.. lmax);
var n : 0 .. lmax; i : 1 .. lmax;
begin
    n := L.longueur;
    if k ≤n then begin
        for i := k to n − 1 do L.t[i] := L.t[i + 1]; L.longueur := n − 1
    end
end supprimer;

procedure insérer (var L : LISTET; k : 1.. lmax; x : Elément);
var n : 0 .. lmax; i : 1 .. lmax;
begin
    n := L.longueur;
    if (n <lmax) and (k ≤ n + 1) then begin
        for i := n downto k do L.t[i + 1] := L.t[i];
        L.t[k] := x; L.longueur := n + 1
    end
end insérer;
```

On voit qu'une suppression demande au pire $n-1$ affectations d'éléments (suppression du premier élément) et une insertion $n+1$ (insertion à la première place), où n est le nombre d'éléments de la liste.

Cette représentation est relativement bien adaptée aux listes itératives : accès et parcours sont très efficaces; mais les adjonctions et insertions ailleurs qu'en fin de liste sont coûteuses et il faut savoir majorer la taille des listes.

Elle est par contre mal adaptée aux listes récursives : l'accès au premier élément est facile, mais il n'est pas commode de représenter les opérations *cons* et *fin*.

2.2. Représentation chaînée

On utilise des pointeurs pour chaîner entre eux les éléments successifs, et la liste est alors déterminée par l'adresse de son premier élément. Une liste est ainsi représentée par le type PASCAL suivant :

type cellules = **record** *val* : Elément; *lien* : ↑cellules **end**;
LISTEP = ↑cellules;

La figure 2 montre une liste L de type LISTEP, contenant 4 éléments.

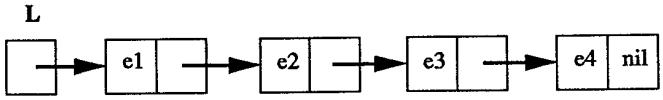


Figure 2. Liste chaînée.

La liste vide est représentée par le pointeur nil. Si la liste représentée par L n'est pas vide, la valeur de L est un pointeur qui représente *tête*(L); $L\uparrow$*.lien* représente *fin*(L); si la place p est représentée par le pointeur P, *succ*(p) est représenté par $P\uparrow$*.lien*.

Cette représentation nécessite de la place mémoire supplémentaire pour les pointeurs. Elle a l'avantage de ne pas imposer une longueur maximum sur les listes; elle permet de traiter facilement la plupart des opérations sur les listes : le parcours séquentiel, l'insertion et la suppression d'un élément à une place quelconque, la concaténation de deux listes, se font par une simple manipulation de pointeurs. Cependant le calcul de la longueur nécessite le parcours de toute la liste; de même, l'accès au $k^{ième}$ élément n'est plus direct : on doit parcourir $k-1$ pointeurs à partir de la tête de la liste pour trouver le $k^{ième}$ élément.

Quoique très simples dans leur principe, ces procédures sont souvent programmées de manière erronée, la manipulation des pointeurs devant se faire avec soin. On trouvera ci-dessous des fonctions et procédures correspondant aux opérations *longueur*, *ième*, *supprimer* et *ajouter*. On notera l'utilisation systématique de variables auxiliaires pour parcourir la liste L (l'utilisation de la variable L pour cela est une erreur fréquente qui a pour effet de perdre la partie de la liste parcourue) et la nécessité du traitement particulier des premiers éléments pour la suppression et l'insertion.

```
function longueur(L : LISTEP ) : integer;
var n : integer; P : LISTEP;
begin
    n := 0; P := L;
    while P <> nil do begin n := n + 1; P := P↑.lien end;
    return (n)
end longueur;
```

```
procedure ième (L: LISTEP; k: integer; var x: Elément; var erreur: boolean);
{erreur prend la valeur true si, et seulement si, la précondition sur ième n'est pas
satisfaite}
var i : integer; P : LISTEP;
begin
    P := L; erreur := false;
    for i := 1 to k - 1 do
        if P = nil then begin erreur := true; return end
        else P := P↑.lien;
    if P = nil then erreur := true else x := P↑.val
end ième;

procedure supprimer(var L : LISTEP; k : integer; var erreur : boolean);
var i : integer; P, PP : LISTEP;
begin
{cas où la liste est vide :}
    if L = nil then begin erreur := true; return end;
{cas particulier de la suppression du premier élément :}
    if k = 1 then begin L := L↑.lien; erreur := false; return end;
{cas général :}
    P := L; erreur := false;
    for i := 1 to k - 1 do begin
        PP := P; P := P↑.lien;
        if P = nil then begin erreur := true; return end
    end;
    PP↑.lien := P↑.lien
end supprimer;

procedure insérer(var L: LISTEP; k: integer; x: Elément; var erreur: boolean);
var i : integer; C, P, PP : LISTEP;
begin
{cas particulier de l'insertion du premier élément :}
    if k = 1 then begin
        new(C); C↑.val := x; erreur := false; C↑.lien := L; L := C; return
    end;
{cas général :}
    P := L; erreur := false;
    for i := 1 to k - 1 do begin
        if P = nil then begin erreur := true; return end;
        PP := P; P := P↑.lien
    end;
    new(C); C↑.val := x; C↑.lien := P; PP↑.lien := C
end insérer;
```

On remarque que toutes ces opérations nécessitent au pire $\Theta(n)$ affectations de pointeurs, si la liste contient n éléments.

Cependant, cette représentation permet des adjonctions et des suppressions en milieu de liste sans déplacements d'éléments. Elle permet aussi de concaténer deux listes sans recopier la deuxième.

La programmation des fonctions et procédures ci-dessus serait beaucoup plus simple si on vérifiait les préconditions avant le parcours, plutôt que pendant. Cela nécessite un parcours supplémentaire de la liste, éventuellement sur toute sa longueur. Une variante intéressante de cette représentation consiste à prendre un couple <liste chaînée, longueur> et de mettre à jour la longueur, comme dans le cas contigu, à chaque suppression ou insertion. Cela ne change pas, cependant, l'ordre de grandeur de la complexité.

Il existe d'ailleurs de nombreuses variantes de la représentation des listes à l'aide de pointeurs :

a) Il est parfois commode de définir une liste non pas comme un pointeur sur une cellule, mais par un bloc de cellules du même type que les autres cellules de la liste. Ce bloc ne contient que l'adresse du premier élément de la liste (voir figure 3). L'utilisation d'un tel bloc permet d'éviter un traitement spécial pour l'insertion et la suppression en début de liste (cf. exercices).

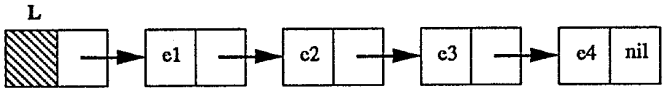


Figure 3. Liste chaînée avec tête de liste.

b) On peut aussi créer des ***listes circulaires*** : on remplace, dans la dernière place de la liste, le pointeur à nil par un pointeur vers la tête de la liste ; de plus si l'on choisit la *dernière place* comme point d'entrée dans la liste comme dans la figure 4, on retrouve la tête de liste en parcourant *un seul* lien (notons que dans la représentation classique des listes, pour avoir le premier et le dernier élément, il faut utiliser deux pointeurs si l'on veut éviter de parcourir toute la liste). Les listes circulaires sont utiles notamment pour représenter des files (on y revient au paragraphe suivant).

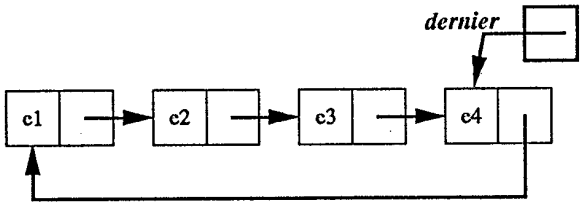


Figure 4. Liste chaînée circulaire.

c) Dans la représentation classique, le parcours des listes est orienté dans un seul sens : du premier élément vers le dernier élément. Cependant de nombreuses applications nécessitent de parcourir les listes à la fois vers l'avant et vers l'arrière, et dans ce cas on peut faciliter le traitement en rajoutant des «pointeurs arrière»

ce qui augmente évidemment la place mémoire utilisée. On obtient alors une ***liste doublement chaînée*** : chaque place comporte un pointeur vers la place suivante et un pointeur vers la place précédente. Sur la figure 5, on a représenté une liste doublement chaînée circulaire.

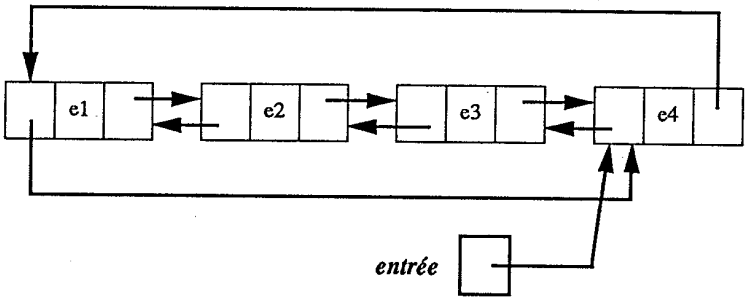


Figure 5. Liste doublement chaînée circulaire.

2.3. Simulation de pointeurs dans un tableau

Certains langages de programmation ne comportent pas de pointeurs. On peut cependant simuler dans ce cas la notion de pointeur dans un tableau, à l'aide d'entiers qui représentent l'indice des places dans le tableau ; le type PASCAL suivant décrit une mémoire où l'on représente des listes chaînées :

type MEM = **array** [1.. *max*] **of record** *val* : Elément;
suiv : integer **end**;

Une liste est alors représentée par un indice dans le tableau de type MEM.

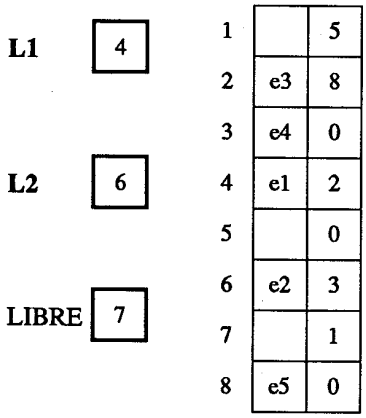


Figure 6. Listes chaînées dans un tableau.

Le choix pour *max* d'un nombre assez grand permet non seulement de faire évoluer une liste en y insérant des éléments, mais aussi de représenter plusieurs listes à l'intérieur d'un même tableau : sur la figure 6 on a représenté dans un même tableau, où *max* vaut 8, la liste $(e1, e3, e5)$ dont l'indice de tête dans le tableau est contenu dans la variable entière $L1$, et la liste $(e2, e4)$ dont l'indice de tête est dans $L2$; de plus, pour pouvoir insérer et supprimer des éléments dans les listes, il faut chaîner entre elles les cases libres du tableau : insérer dans $L1$ ou $L2$ provoque la suppression de la première place de la liste libre (ce qui correspond à un *new* de Pascal); supprimer dans $L1$ et $L2$ provoque l'adjonction de la place supprimée en tête de la liste libre (en Pascal, la restitution des zones de mémoire libres se fait soit automatiquement par un ramasse-miettes, soit explicitement par l'instruction *dispose*). Sur la figure 6, la variable *LIBRE* contient l'indice dans le tableau de la première place de la liste libre (voir aussi exercices). L'indice 0 correspond ici au pointeur *nil*.

3. Les piles et les files

Pour beaucoup d'applications, les seules opérations à effectuer sur les listes sont des insertions et des suppressions aux extrémités.

3.1. Les piles

Dans les piles les insertions et les suppressions se font à une seule extrémité, appelée ***sommet de pile***. Les piles sont aussi appelées LIFO, pour Last-In-First-Out, c'est-à-dire, dernier-entré-premier-sorti; une bonne image pour se représenter une pile est une ... pile d'assiettes : c'est en haut de la pile qu'il faut prendre ou mettre une assiette !

Les opérations sur les piles sont : tester si une pile est vide; accéder au sommet d'une pile; empiler un élément; retirer l'élément qui se trouve au sommet (dépiler).

La signature du type Pile est donc :

sorte Pile
utilise Booléen, Elément
opérations

pile-vide	:	$\rightarrow$ Pile
empiler	:	Pile $\times$ Elément $\rightarrow$ Pile
dépiler	:	Pile $\rightarrow$ Pile
sommet	:	Pile $\rightarrow$ Elément
est-vide	:	Pile $\rightarrow$ Booléen

Les opérations *dépiler* et *sommet* ne sont définies que si la pile n'est pas vide :

dépiler(*p*) **est-défini-ssi** *est-vide*(*p*) = faux
sommet(*p*) **est-défini-ssi** *est-vide*(*p*) = faux

Ces opérations vérifient les axiomes suivants, où p est de sorte Pile, et e de sorte Elément :

dépiler(*empiler*(*p*, *e*)) = *p*
sommet(*empiler*(*p*, *e*)) = *e*

D'autre part, l'opération *est-vide* vérifie :

est-vide(*pile-vide*) = vrai
est-vide(*empiler*(*p*, *e*)) = faux

Cette spécification est suffisamment complète car on sait réécrire en utilisant le deuxième axiome toute expression définie, sans variable, comportant une opération *dépiler*, en une expression équivalente ne comportant que les opérations *pile-vide* et *empiler*. Or, on peut déduire par les autres axiomes le résultat des observateurs *sommet* et *est-vide* sur toute expression sans variable, définie, ne comportant que des opérations *pile-vide* et *empiler*.

On peut utiliser pour implémenter les piles toutes les représentations étudiées pour les listes. On montre ici une représentation contiguë et une représentation chaînée, avec, pour chaque cas, la représentation correspondante des opérations du type abstrait Pile.

3.1.1. Représentation contiguë des piles

Les éléments de la pile sont rangés dans un tableau, et l'on conserve aussi l'indice du sommet de pile; on a, par exemple, le type PASCAL suivant :

```
type PILE = record som : integer;
                   elts : array [1.. lmax] of Elément
            end;
```

La pile vide est représentée par tout enregistrement dont le champ *som* contient 0. Les opérations du type abstrait sont données ci-dessous.

```
procedure pile-vide(var P : PILE);
begin
    P.som := 0
end pile-vide;
```

```
procedure empiler(var P : PILE; x : Elément);
begin
    if P.som = lmax then write ('erreur pile pleine')
    else begin P.som := P.som + 1; P.elts[P.som] := x end
end empiler;

procedure dépiler(var P : PILE);
begin
    if est-vide (P) then write ('erreur pile vide')
    else P.som := P.som - 1
end dépiler;

function sommet(P : PILE) : Elément;
begin
    if est-vide(P) then write ('erreur pile vide')
    else return (P.elts[P.som])
end sommet;

function est-vide(P : PILE) : boolean;
return (P.som = 0);
```

3.1.2. Représentation chaînée des piles

Les éléments de la pile sont chaînés entre eux, et le sommet d'une pile non vide est le premier de la liste; la pile vide est représentée par *nil*. On a le type PASCAL suivant :

```
type Elt = record val : Elément; lien : ↑Elt end;
     PILE = ↑Elt;
```

On donne ci-dessous la programmation des opérations sur les piles dans ce cas.

```
procedure pile-vide(var P : PILE);
begin
    P := nil
end pile-vide;

procedure empiler(var P : PILE; x : Elément);
var E : PILE;
begin
    new (E); E↑.val := x; E↑.lien := P; P := E
end empiler;

procedure dépiler(var P : PILE);
begin
    if est-vide(P) then write ('erreur pile vide')
    else P := P↑.lien
end dépiler;
```

```
function sommet(P : PILE) : Elément;
begin
    if est-vide(P) then write ('erreur pile vide')
    else return (P↑.val)
end sommet;

function est-vide(P : PILE ) : boolean;
return (P = nil);
```

Les en-têtes de ces procédures sont les mêmes pour les deux représentations et correspondent à peu de choses près, à la signature du type Pile. Cela permet, dans la suite de ce livre, d'utiliser des piles pour certains algorithmes sans préciser l'implémentation choisie.

La différence entre ces en-têtes et la signature du type Pile vient de l'utilisation de procédures où le même paramètre (passé par variable) est utilisé comme argument et comme résultat.

3.2. Les files

Dans le cas d'une file on fait les adjonctions à une extrémité, les accès et les suppressions à l'autre extrémité. Par analogie avec les files d'attente on dit que l'élément présent depuis le plus longtemps est le premier, on dit aussi qu'il est en tête. Les files sont aussi appelée FIFO pour First-In-First-Out, c'est-à-dire, premier-entré-premier-sorti. Les opérations sur les files sont : tester si une file est vide; accéder au premier élément de la file; ajouter un élément dans la file; retirer le premier élément de la file. La signature du type File est donc :

sorte File
utilise Booléen, Elément
opérations

file-vide	:	$\rightarrow$ File
ajouter	:	File $\times$ Elément $\rightarrow$ File
retirer	:	File $\rightarrow$ File
premier	:	File $\rightarrow$ Elément
est-vide	:	File $\rightarrow$ Booléen

Les opérations *premier* et *retirer* ne sont définies que si la file n'est pas vide :

premier(f) **est-défini-ssi** *est-vide*(f) = faux
retirer(f) **est-défini-ssi** *est-vide*(f) = faux

Naturellement, les axiomes diffèrent de ceux de la pile en ce qui concerne les opérations *premier* et *retirer*. Pour tout f de sorte File et tout e de sorte Elément, on a :

est-vide(f) = vrai $\Rightarrow$ *premier*(*ajouter*(f, e)) = e
est-vide(f) = faux $\Rightarrow$ *premier*(*ajouter*(f, e)) = *premier*(f)

est-vide(f) = vrai ⇒ *retirer*(*ajouter*(f, e)) = *file-vide*
est-vide(f) = faux ⇒ *retirer*(*ajouter*(f, e)) = *ajouter*(*retirer*(f), e)

De plus on a les axiomes suivants pour *est-vide* :

est-vide(*file-vide*) = vrai
est-vide(*ajouter*(f, e)) = faux

On peut représenter les files de manière contiguë ou de manière chaînée.

3.2.1. Représentation contiguë des files

Dans ce cas on doit conserver l'indice i du premier élément et l'indice j de la première case libre après la file. On fait progresser ces indices modulo la taille *lmax* du tableau. Le seul point délicat est la détection des débordements (voir la figure 7 et les exercices).

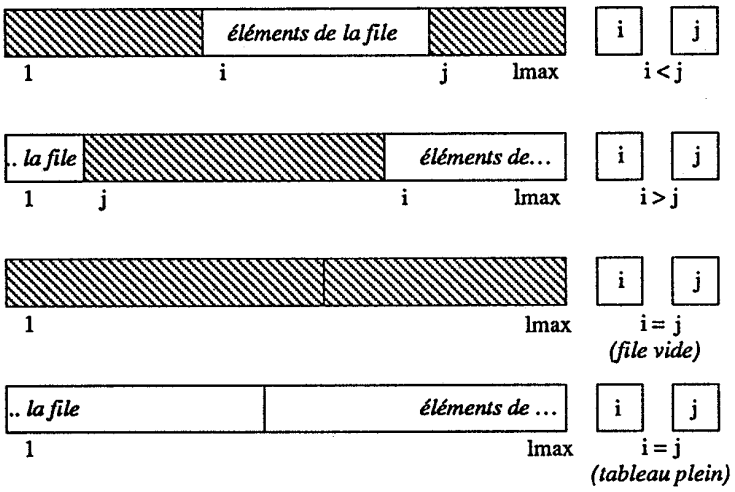


Figure 7. Etats possibles de la représentation contiguë d'une file.

3.2.2. Représentation chaînée des files

Dans le cas d'une représentation chaînée, soit on a deux pointeurs, *tête* et *dernier*, vers le premier et le dernier élément de la file, soit on utilise le pointeur qui suit le dernier élément pour repérer le premier élément. On a alors une représentation circulaire comme le montre la figure 8 b.

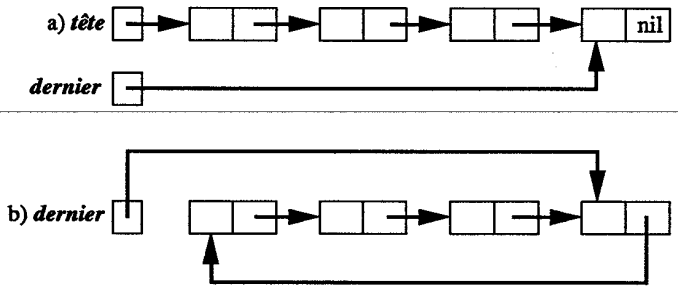


Figure 8. Représentation chaînée des files.

Les représentations contiguës des piles et des files sont très performantes en place et en temps mais il faut savoir majorer le nombre d'éléments pour pouvoir les utiliser. Les représentations chaînées prennent plus de place en mémoire mais n'imposent pas cette majoration.

La programmation des opérations sur les files, dans le cas contigu et dans le cas chaîné est laissée en exercice.

Exercices

1. On a donné deux définitions de type abstrait pour les listes : le type «liste itérative» et le type «liste récursive».

a) Montrer que les opérations *tête, fin* et *cons* peuvent se définir en termes des opérations *accès, insérer* et *supprimer.*

b) Montrer que les opérations *accès, insérer, supprimer* et *longueur* peuvent se définir en termes des opérations *tête, fin* et *cons.*

2. On peut également définir le type liste récursive «en arrière», en prenant, au lieu des opérations *tête* et *fin*, les opérations *dernier* et *début* (les autres opérations du type étant les mêmes). L'opération *début* donne, pour une liste non vide, la liste privée de son dernier élément et l'opération *dernier* donne le dernier élément d'une liste qui a au moins un élément.

a) Préciser les domaines de définition de *début* et de *dernier* et montrer que ces opérations se définissent en termes des opérations du type abstrait «liste récursive».

b) Montrer que les opérations *tête* et *fin* peuvent se définir à l'aide des opérations *début* et *dernier.*

c) Définir l'opération *concaténer* en utilisant *début* et *dernier.*

3. Définir l'opération *recherche de la première occurrence d'un élément dans une liste* pour le type «liste itérative».

4. L'opération *inverser* consiste à inverser l'ordre des éléments d'une liste. Donner des axiomes qui définissent cette opération en utilisant les opérations du type «liste récursive», puis celles du type liste récursive «en arrière».

5. a) Une autre représentation des listes par tableau consiste à implémenter différemment l'opération de suppression : à la place de l'élément détruit, on met une valeur spéciale qui ne peut pas être prise par les éléments de la liste. Définir

le type correspondant et écrire les procédures de suppression, insertion et recherche d'un élément dans une liste.

b) Une variante du type LISTEP (cf. figure 3) consiste à considérer une tête de liste. Comment les opérations sont-elles modifiées ?

c) Reprendre les questions précédentes dans le cas d'une liste triée.

6. On se donne N entiers en lecture. Ecrire une procédure qui constitue une liste chaînée de ces entiers. (On donnera une version récursive et une version itérative).

7. Soit L une liste d'entiers représentée par une liste simplement chaînée. Donner une procédure récursive qui effectue le traitement suivant : écrire sur une ligne les entiers de la liste dans l'ordre où ils figurent dans L, et sur la ligne suivante la liste des carrés dans l'ordre inverse.

8. Ecrire des programmes qui décrivent les opérations d'insertion et de suppression dans les listes pour la représentation doublement chaînée.

9. Pour jouer à AM-STRAM-GRAM, N enfants forment une ronde, choisissent l'un d'eux comme le premier, commencent à compter à partir de lui et décident que le $k^{ième}$ enfant doit quitter la ronde, puis le $2k^{ième}$, et ainsi de suite.

a) En représentant la ronde des enfants par une liste circulaire, écrire un programme qui donne la suite des enfants dans l'ordre où ils sont sortis de la ronde.

b) Reprendre le problème avec une représentation de la ronde dans un tableau.

10. On appelle *palindrome* un mot m qui se lit de la même façon de gauche à droite et de droite à gauche. Par exemple les mots «elle» et «radar» sont des palindromes. Un mot étant considéré comme une liste de caractères, trouver une représentation chaînée judicieuse pour les mots et écrire une procédure qui reconnaît si un mot est un palindrome.

11. Ecrire deux procédures de fusion de deux listes triées L_1 et L_2 sans répétition d'éléments telles que :

a) pour la première procédure, la liste résultante L est triée et chaque élément apparaît dans L autant de fois qu'il apparaît dans L_1 et dans L_2,

b) pour la seconde, la liste résultante L est triée et n'a qu'une seule occurrence de chaque élément.

12. a) Ecrire une procédure qui concatène deux listes, pour chacune des deux représentations contiguë et chaînée.

b) En choisissant une représentation chaînée, écrire une procédure qui concatène une liste de listes.

13. Ecrire une procédure qui inverse une liste chaînée sans recopier ses éléments.

14. Soit L une liste chaînée. Ecrire des procédures telles que :

a) dans la première on supprime toutes les occurrences d'un élément donné x.

b) dans la deuxième, pour chaque élément de la liste, on ne laisse que sa première occurrence.

15. On considère des polynômes de la forme : $P(x) = a_n x^{e_1} + a_{n-1} x^{e_2} + ... + a_1 x^{e_n}$, avec $e_1 > e_2 > ... > e_n \geq 0$.

a) Trouver une représentation adéquate d'un polynôme par une liste chaînée.

b) Ecrire une procédure d'addition de deux polynômes.

c) Ecrire une procédure de différentiation d'un polynôme.

d) Ecrire une procédure de multiplication de deux polynômes.

16. Un nombre entier p peut être codé en représentation binaire : $b_0 b_1 \ldots b_n$, avec $b_i = 0$ ou 1, $(1 \leq i \leq n)$ et $p = \sum_{i=0}^{n} b_i 2^{n-i}$. Ecrire une procédure *incrément* qui ajoute 1 à un entier codé en binaire.

17. Trouver comment ranger deux piles dans un seul tableau T de sorte qu'on ne puisse plus empiler sur aucune que si le tableau est plein. Ecrire une procédure *empiler(x, T, B)* qui empile l'élément x sur l'une des deux piles dans T selon la valeur du booléen B.

18. Dans cet exercice on étudie quelques propriétés des *permutations engendrables par une pile.*

a) Imaginons quatre voitures v_1, v_2, v_3 et v_4 arrivant à la queu-leu-leu sur une route à une seule voie (v_1 étant en tête), pour entrer dans un garage qui n'a qu'une seule voie. Les mouvements possibles des voitures sont : une voiture ne peut ressortir que si elle est la dernière entrée (les voitures ne peuvent sauter l'une par dessus

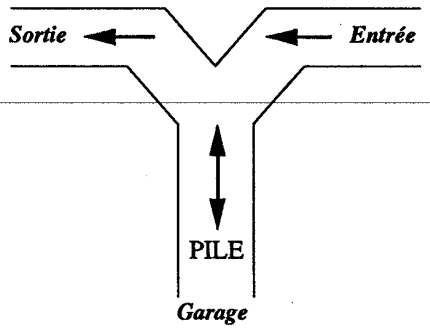


Figure 9

l'autre) et les voitures ressortent sur une autre voie que la voie d'entrée. Le garage fonctionne ainsi comme une pile. On réalise la suite d'actions suivante :

1) rentrer v_1 ; 2) rentrer v_2 ; 3) sortir v_2 ; 4) entrer v_3 ; 5) entrer v_4 ; 6) sortir v_4 ; 7) sortir v_3 ; 8) sortir v_1. Il en résulte que l'ordre de sortie des voitures n'est plus v_1, v_2, v_3, v_4 mais v_2, v_4, v_3, v_1 : on a obtenu une autre permutation. Supposant que six voitures $v_1, v_2, v_3, v_4, v_5, v_6$ se présentent dans cet ordre à l'entrée du garage (v_1 en tête), peut-on obtenir l'ordre $v_3, v_2, v_5, v_6, v_4, v_1$ (v_3 en tête) et l'ordre $v_1, v_5, v_4, v_6, v_2, v_3$ (v_1 en tête) en sortie? Dans les cas où c'est possible, comment le réaliser?

b) On peut symboliser l'action «entrer une voiture» par le symbole *E (Empiler)* et l'action «sortir une voiture» par le symbole *D (Dépiler)*. La suite des actions 1), ..., 8) peut alors être symbolisée par $EEDEEDDD$. Toutes les séquences formées de E et de D ne s'interprètent pas par une suite d'actions sur la pile. Par exemple la séquence $EDDEEDDE$ n'a pas de sens puisqu'à un moment, il faudrait dépiler une pile déjà vide. Une séquence formée de E et de D est dite *admissible* si elle contient autant de E que de D et si toutes les actions qui lui correspondent peuvent être accomplies dans l'ordre indiqué par la séquence.

Donner une règle qui caractérise les séquences admissibles, c'est-à-dire *les permutations engendrables par une pile.*

c) On note a_n le nombre de permutations de n éléments engendrables par une pile. Calculer a_n.

d) Montrer qu'il est possible d'obtenir la permutation $p_1, p_2, ..., p_n$, $(1 \leq p_i \leq n)$ à partir de $1, 2, \ldots, n$ en utilisant une pile si, et seulement si, il n'y a pas d'indices $i < j < k$ tels que $p_j < p_k < p_i$.

e) Si on utilise une file au lieu d'une pile, quelles sont les permutations de $1, 2, \ldots, n$ qui peuvent être obtenues?

19. Ecrire des procédures qui décrivent les opérations *ajouter* et *retirer* dans le cas d'une représentation des files par tableau. (On peut continuer à ajouter des éléments tant que le tableau n'est pas plein.)

20. Donner une implémentation des opérations du type File dans le cas de la représentation chaînée.

Lectures conseillées pour le chapitre 5

Aho, Hopcroft & Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983.

Wirth, *Algorithms + Data Structures = Programs*, Prentice Hall, 1976.

Chapitre 6
Ensembles

Dans le cas des structures séquentielles étudiées au chapitre précédent, la notion d'ordre des éléments appartenant à une liste ou une file ou une pile est fondamentale. Ce n'est pas toujours le cas : l'ordre des éléments peut n'avoir aucune signification et la propriété importante est la présence ou l'absence d'un élément.

On retrouve alors la notion d'***ensemble d'éléments*** : on veut pouvoir tester l'appartenance d'un élément à un ensemble, ajouter ou supprimer un élément, tester si un ensemble est vide, etc. La définition et les propriétés des ensembles sont bien connues et ne sont pas rappelées ici.

Cependant, dans certains cas on peut être amené à considérer des répétitions d'éléments : on parle alors d'***ensembles avec répétitions*** ou encore de ***multi-ensembles***. Certaines des représentations décrites dans ce chapitre sont valables seulement pour les ensembles, d'autres le sont aussi pour les ensembles avec répétitions.

1. Spécifications des ensembles

1.1. Le type abstrait Ensemble

La signature du type Ensemble comporte au moins les sortes et opérations suivantes :

sorte Ensemble
utilise Elément, Booléen
opérations

ensemble-vide	: → Ensemble
ajouter	: Elément × Ensemble → Ensemble
supprimer	: Elément × Ensemble → Ensemble
$- \in -$	: Elément × Ensemble → Booléen

On a plusieurs choix pour le comportement de l'opération *ajouter* dans le cas d'un élément déjà présent et de même pour l'opération *supprimer* dans le cas d'un élément absent : on peut émettre un message d'erreur ce qui signifie qu'*ajouter* et

supprimer sont des opérations partielles, ou on peut considérer que ces opérations sont sans effet, ou on peut faire les deux... Dans les axiomes ci-dessous, on considère qu'ajouter un élément présent ou supprimer un élément absent laisse l'ensemble inchangé.

Dans ce qui suit, x et y sont des variables de sorte Elément et e est une variable de sorte Ensemble.

axiomes

$x \in$ *ensemble-vide* = faux
$x = y \Rightarrow (x \in$ *ajouter*$(y,\ e)) =$ vrai
$x \neq y \Rightarrow (x \in$ *ajouter*$(y,\ e)) = (x \in e)$
$x = y \Rightarrow (x \in$ *supprimer*$(y,\ e)) =$ faux
$x \neq y \Rightarrow (x \in$ *supprimer*$(y,\ e)) = (x \in e)$

Cette spécification peut être enrichie par d'autres opérations, par exemple le nombre d'éléments d'un ensemble :

card : Ensemble → Entier

On a alors les axiomes :

card(ensemble-vide) = 0
$(x \in e) =$ vrai $\Rightarrow$ *card*(*ajouter*$(x,\ e)) =$ *card*(e)
$(x \in e) =$ faux $\Rightarrow$ *card*(*ajouter*$(x,\ e)) =$ *card*$(e) + 1$
$(x \in e) =$ vrai $\Rightarrow$ *card*(*supprimer*$(x,\ e)) =$ *card*$(e) - 1$
$(x \in e) =$ faux $\Rightarrow$ *card*(*supprimer*$(x,\ e)) =$ *card*(e)

On peut aussi ajouter les opérations classiques sur les ensembles : *union*, *intersection*, *complémentaire*.

1.2. Enumération d'un ensemble

Dans le cas des listes, on a vu que le traitement de tous les éléments d'une liste se fait par un parcours séquentiel des places. Dans le cas des ensembles, l'ordre de traitement des éléments n'est pas déterminé. Pour traiter tous les éléments d'un ensemble non vide, on choisit un élément de l'ensemble, on le traite, et on recommence sur l'ensemble obtenu en supprimant l'élément qui vient d'être traité, ceci tant que l'ensemble n'est pas vide.

On enrichit donc la signature des ensembles par une opération de choix d'un élément de profil

choisir : Ensemble → Elément

avec la précondition suivante

choisir(e) **est-défini-ssi** $e \neq$ *ensemble-vide*

et l'axiome

$$(\mathit{choisir}(e) \in e) = \text{vrai}$$

Le programme correspondant au traitement de tous les éléments d'un ensemble e s'écrit alors

$e' := e$;
while $e' \neq$ *ensemble-vide* **do begin**
    $x :=$ *choisir*(e'); *traiter*(x);
    $e' :=$ *supprimer*$(x,\ e')$
end;

où les opérations sur les ensembles sont représentées selon une des méthodes étudiées au paragraphe 2 de ce chapitre.

1.3. Cas des ensembles avec répétitions

Dans le cas des ensembles avec répétitions (appelés aussi multi-ensembles), on a les mêmes opérations que ci-dessus, mais certains axiomes diffèrent : en effet, le nombre d'éléments augmente quand on ajoute un élément déjà présent; de même, supprimer une occurrence d'un élément n'implique pas que cet élément n'appartient plus au multi-ensemble, puisqu'il peut être présent plusieurs fois. Il y a une opération supplémentaire qui donne le nombre d'occurrences d'un élément dans un multi-ensemble.

On a donc les axiomes suivants, où x et y sont des variables de sorte Elément et m une variable de sorte Multi-ensemble :

sorte Multi-ensemble
utilise Elément, Booléen
opérations

multi-ensemble-vide	: $\rightarrow$ Multi-ensemble
ajouter	: Elément $\times$ Multi-ensemble $\rightarrow$ Multi-ensemble
supprimer	: Elément $\times$ Multi-ensemble $\rightarrow$ Multi-ensemble
$- \in -$	: Elément $\times$ Multi-ensemble $\rightarrow$ Booléen
card	: Multi-ensemble $\rightarrow$ Entier
nb-occurrences	: Elément $\times$ Multi-ensemble $\rightarrow$ Entier

axiomes

$(x \in \mathit{multi\text{-}ensemble\text{-}vide}) = \text{faux}$
$x = y \Rightarrow (x \in \mathit{ajouter}(y,\ m)) = \text{vrai}$
$x \neq y \Rightarrow (x \in \mathit{ajouter}(y,\ m)) = (x \in m)$
$\mathit{nb\text{-}occurrences}(x,\ m) \leq 1 \Rightarrow (x \in \mathit{supprimer}(x,\ m)) = \text{faux}$
$x \neq y \Rightarrow (x \in \mathit{supprimer}(y, m)) = (x \in m)$

card(multi-ensemble-vide) = 0
card(ajouter(x, m)*)* = *card*$(m) + 1$
$(x \in m)$ = vrai $\Rightarrow$ *card(supprimer*$(x,\ m)$*)* = *card*$(m) - 1$
$(x \in m)$ = faux $\Rightarrow$ *card(supprimer*$(x,\ m)$*)* = *card*(m)

nb-occurrences$(x,$ *multi-ensemble-vide*$)$ = 0
$x = y \Rightarrow$ *nb-occurrences*$(x,$ *ajouter*$(y,\ m))$ = *nb-occurrences*$(x,\ m) + 1$
$x \neq y \Rightarrow$ *nb-occurrences*$(x,$ *ajouter*$(y, m))$ = *nb-occurrences*$(x,\ m)$
nb-occurrences$(x,\ m) = 0 \Rightarrow$ *nb-occurrences*$(x,$ *supprimer*$(x, m)) = 0$
nb-occurrences$(x,\ m) \geq 1 \Rightarrow$
nb-occurrences$(x,$ *supprimer*$(x,\ m))$ = *nb-occurrences*$(x,\ m) - 1$
$x \neq y \Rightarrow$ *nb-occurrences*$(x,$ *supprimer*$(y,\ m))$ = *nb-occurrences*$(x,\ m)$

Comme la spécification des ensembles, cette spécification pourrait être enrichie d'opérations d'*union* et *d'intersection.*

L'énumération des éléments d'un multi-ensemble se fait selon le même schéma qu'au paragraphe 1.2, puisque l'opération *supprimer* ne supprime qu'une occurrence de l'élément en argument.

2. Représentation par des tableaux de booléens

Dans le cas où l'on travaille sur des ensembles dont les éléments possibles sont en nombre fini N, il est possible de représenter tout ensemble par un tableau de N booléens :

type ENSB = **array** [1..N] **of** boolean;

Si e est de type ENSB, $e[i]$ a la valeur true si $i \in e$ et la valeur false sinon.

Le test d'appartenance d'un élément i à l'ensemble e se fait donc par un simple accès au tableau. L'adjonction d'un élément i à un ensemble e se fait en affectant la valeur true à $e[i]$, et la suppression en lui affectant la valeur false.

Ces opérations sont donc représentées de manière très efficace puisqu'elles correspondent à un accès ou à une affectation.

Cependant cette représentation est coûteuse en place mémoire, surtout si N est grand et si les ensembles considérés n'ont pas beaucoup d'éléments.

De plus, l'ensemble vide est représenté par un tableau où tous les éléments valent false. De ce fait, l'initialisation d'un ensemble à la valeur ensemble-vide demande N affectations. De même, l'énumération de tout ensemble, ou le calcul du nombre de ses éléments, demande le parcours du tableau, soit N accès.

opération du type abstrait	**programmation**
ajouter(x, e)	$e[x] :=$ true
supprimer(x, e)	$e[x] :=$ false
$x \in e$	$e[x]$
ensemble-vide	**for** $i := 1$ **to** N **do** $e[i] :=$ false;
card(e)	$c := 0$;
	for $i := 1$ **to** N **do if** $e[i]$ **then** $c := c + 1$

Figure 1. Récapitulation de la représentation des ensembles par des tableaux de booléens.

Notons que cette représentation doit être modifiée pour les multi-ensembles : dans ce cas, ce n'est plus un booléen qu'il faut avoir dans le tableau, mais un entier qui indique le nombre d'occurrences.

Lorsque le nombre des éléments possibles N est trop grand, voire infini (par exemple, des ensembles d'entiers, de mots...), mais que les ensembles qu'on manipule effectivement sont de petite taille, une des solutions possibles est d'utiliser des listes pour les représenter. Cette solution est présentée au paragraphe suivant.

Remarque : Dans le langage Pascal, il existe un type ensemble, qui est très similaire à ce qui vient d'être présenté : les éléments doivent être de type scalaire. La représentation interne est équivalente à des tableaux compressés de booléens, ayant pour indices le type des éléments.

3. Représentation par des listes

Un ensemble est représenté par une liste de ses n éléments : il y a plusieurs listes possibles puisque l'ordre des éléments n'a pas de signification.

Le test d'appartenance d'un élément x à un ensemble e représenté par une liste se fait par un parcours séquentiel de la liste, en comparant chaque élément à x. On arrête le parcours dès que la comparaison est un succès.

L'adjonction d'un élément x peut se faire de plusieurs manières : on insère l'élément x à une place quelconque de la liste; on verra plus loin que le choix judicieux de cette place dépend de la représentation de la liste. De plus, on a le choix entre faire une adjonction «paresseuse», qui ne teste pas si x est déjà présent dans la liste avant de l'insérer, ou faire ce test d'appartenance avant toute adjonction, ce qui nécessite le parcours de la liste, mais économise de la place mémoire, facilite les suppressions ultérieures, ainsi que le calcul du nombre d'éléments.

La suppression d'un élément x nécessite un parcours de la liste. Si l'adjonction est paresseuse, il faut parcourir toute la liste pour supprimer toutes les occurrences de x, sinon on peut arrêter le parcours dès que l'on a rencontré x. On verra plus loin que la représentation de la liste influe également sur certains détails de programmation de l'opération de suppression.

Pour énumérer les éléments de l'ensemble, dans le cas où on ne fait pas d'adjonction paresseuse, il suffit de traiter séquentiellement tous les éléments de la liste; si on fait de l'adjonction paresseuse, il faut suivre exactement le schéma donné au §1.2, ce qui est très inefficace puisque chaque suppression provoque un parcours de la liste.

Dans le cas des ensembles avec répétitions, cette représentation s'adapte très facilement : l'adjonction se fait comme une adjonction paresseuse dans un ensemble, mais pour la suppression on s'arrête à la première occurrence de x. Le test d'appartenance est inchangé. L'énumération se fait par un parcours de la liste.

Cependant le calcul du nombre d'occurrences d'un élément x dans un multi-ensemble nécessite le parcours de toute la liste. Dans le cas où l'efficacité de cette dernière opération est importante, il faut choisir une autre représentation : on représente alors le multi-ensemble par une liste de couples

$$< x,\ \textit{nb-occurrences}(x) >$$

où chaque élément apparaît au plus une fois. Le calcul du nombre d'occurrences se fait alors par un parcours de la liste jusqu'à ce qu'on ait trouvé x (ce qui est similaire au test d'appartenance). Cependant, l'adjonction nécessite alors également un parcours de la liste.

Dans les deux paragraphes suivants, on détaille ces opérations dans les cas où la représentation des listes se fait de manière contiguë par des tableaux, ou de manière chaînée (cf. chapitre 5).

3.1. Représentation par un tableau

Le tableau est surdimensionné. Dans le cas où on teste l'appartenance avant d'ajouter un élément, on peut donner comme dimension au tableau le nombre maximum d'éléments que l'ensemble sera amené à contenir. Si on fait de l'adjonction paresseuse, il faut surdimensionner plus. Les éléments de l'ensemble sont rangés au début du tableau. Comme dans le cas des listes, on utilise une variable pour repérer l'indice du dernier élément du tableau appartenant à l'ensemble. On a, par exemple :

type ENST = **record** t : **array** [1.. *max*] **of** Elément; *nb* : 0 .. *max* **end**;

Comme dans le cas des listes, on note que l'utilisation d'un enregistrement est facultative et que pour éviter la manipulation de variables pointées, on peut choisir

de déclarer deux variables distinctes, une de type tableau et une de type entier, pour chaque ensemble à représenter.

Dans le cas où un ensemble e est représenté par le type ci-dessus, $e.t[i]$ désigne le $i^{ième}$ élément de la liste, et l'on peut écrire la fonction ci-dessous, qui renvoie la valeur true si l'élément X appartient à la liste et la valeur false sinon.

```
function appartient(X : Elément; e : ENST ) : boolean;
{cette fonction recherche séquentiellement si l'élément X figure ou non
dans l'ensemble e}
var i : integer;
begin i := 1;
    while (i ≤ e.nb) and (X ≠ e.t[i]) do i := i + 1;
    if i = e.nb + 1 then return (false)
                    else return (true)
end appartient;
```

Remarque : Dans le programme ci-dessus, la condition d'arrêt de la boucle **while** est double : on effectue, à chaque itération, un test de fin de liste en plus de la comparaison entre éléments. Une façon élégante d'éviter ce test est d'utiliser une *sentinelle* : on rajoute à la fin de la liste (c'est-à-dire à la place $e.t[e.nb + 1]$) l'élément cherché X de telle sorte que la sortie de la boucle **while** se fait toujours sur la rencontre de l'élément X. Si c'est au rang $e.nb + 1$, c'est que l'élément X n'appartient pas à la liste initiale. On a alors la fonction ci-dessous :

```
function appartient_sent(X : Elément; e : ENST) : boolean;
var k : integer;
begin k := 1; e.t[e.nb + 1] := X;
      while X ≠ e.t[k] do k := k + 1;
      if k = e.nb + 1 then return (false)
      else return (true)
end appartient_sent;
```

Notons que dans ce cas, la constante *max* dans le type ENST ne correspond plus au nombre maximum d'éléments, mais à ce nombre plus un.

Si on teste l'appartenance de X à e avant d'ajouter, la procédure d'adjonction s'écrit :

```
procedure ajouter1(X : Elément; var e : ENST);
begin {précondition : e.nb < max}
  if not appartient(X, e) then begin e.nb := e.nb + 1; e.t[e.nb] := X end
  {appartient peut être une des deux fonctions ci-dessus ; X est inséré en fin
  de liste}
end ajouter1;
```

Dans ce cas, l'opération de suppression est simple : on cherche l'indice (unique) de X dans le tableau, s'il existe, et on met le dernier élément à cet indice. On a donc la procédure suivante :

```
procedure supprimer1(X : Elément; var e : ENST);
var i : integer;
begin
    i := 1; {NB : on pourrait utiliser là aussi X comme sentinelle}
    while (i ≤ e.nb) and (X ≠ e.t[i]) do i := i + 1;
    if X = e.t[i] then begin e.t[i] := e.t[e.nb]; e.nb := e.nb - 1 end
end supprimer1;
```

On remarque que dans ce cas $e.nb$ contient à chaque instant le nombre d'éléments de l'ensemble e.

Ce n'est pas vrai si on fait de l'adjonction paresseuse. Dans ce cas on a :

```
procedure ajouter2(X : Elément; var e : ENST);
begin {précondition : e.nb < max}
    e.nb := e.nb + 1; e.t[e.nb] := X
end ajouter2;
```

La procédure correspondant à l'opération *supprimer* doit alors parcourir toute la liste et supprimer tous les éléments égaux à X :

```
procedure supprimer2(X : Elément; var e : ENST);
var i : integer;
begin
    i := 1;
    while i ≤ e.nb do
        if X ≠ e.t[i] then i := i + 1
        else begin e.t[i] := e.t[e.nb]; e.nb := e.nb - 1 end
end supprimer2;
```

Enfin, on remarque que l'ensemble vide est représenté par tout e de type ENST tel que $e.nb = 0$. L'initialisation d'un ensemble e à la valeur *ensemble-vide* se fait donc par une simple affectation à $e.nb$ de 0.

Dans le cas d'ensembles avec répétitions le type ENST reste utilisable, à condition d'utiliser la procédure *ajouter2* (adjonction sans test d'appartenance préalable) et la procédure *supprimer1* (suppression de la première occurrence de X).

On peut également utiliser un tableau de couples <élément, nombre d'occurrences> où chaque élément apparaît au plus une fois. La programmation des opérations dans ce cas est laissée en exercice.

3.2. Analyse du nombre de comparaisons

L'opération fondamentale est ici la comparaison de deux éléments. On détermine les complexités en fonction de la longueur n de la liste (qui correspond au $e.nb$ du paragraphe précédent).

Certains des algorithmes présentés s'analysent très facilement. Par exemple, l'adjonction paresseuse *(ajouter2)* ne demande aucune comparaison, d'où son nom; par contre, l'algorithme de suppression qui lui correspond *(supprimer2)* fait n comparaisons dans tous les cas.

L'adjonction avec test d'appartenance préalable *(ajouter1)* fait le même nombre de comparaisons que ce test. L'algorithme de suppression qui lui correspond *(supprimer1)* s'arrête soit à la fin de la liste, soit dès que l'on a trouvé X, comme le test d'appartenance. L'analyse du test d'appartenance suffit donc à déterminer la complexité en nombre de comparaisons de ces algorithmes.

L'analyse de la fonction *appartient* a été faite au chapitre 2. On s'intéresse ici au cas où on utilise X comme sentinelle.

La fonction *appartient_sent* effectue k comparaisons de X avec un élément de la liste dans le cas où l'élément recherché apparaît à la place k de la liste (et pas avant), et elle effectue toujours $(n+1)$ comparaisons si l'élément cherché n'est pas dans la liste d'origine.

Pour calculer le nombre moyen de comparaisons, supposons que q_k soit la probabilité que l'élément recherché X apparaisse à la place k, pour $k = 1, .., n$ et que p soit la probabilité pour que l'élément X n'appartienne pas à la liste $(p + q_1 + ...q_n = 1)$. Le nombre moyen de comparaisons pour une recherche est alors :

$$\text{Moy}(n) = 1 \cdot q_1 + 2 \cdot q_2 + ... + n \cdot q_n + (n+1) \cdot p$$

Lorsque toutes les places de la liste où peut se trouver l'élément cherché X sont équiprobables, on a les résultats suivants :

a) si on sait que l'élément X appartient à la liste $(p = 0)$

$$\text{Moy}(n) = \sum_{1 \leq i \leq n} i \cdot (1/n) = (n+1)/2$$

b) si X a une chance sur deux d'appartenir à la liste $(p = 1/2)$

$$\text{Moy}(n) = \sum_{1 \leq i \leq n} i \cdot (1/2n) + (n+1)/2 = 3(n+1)/4$$

N.B. : Ce résultat est légèrement différent de celui du chapitre 2 à cause de l'utilisation de la sentinelle.

Par conséquent, toutes ces opérations, sauf l'adjonction paresseuse, sont en $\Theta(n)$ comparaisons au pire et en moyenne.

3.3. Représentation par une liste chaînée

Il est parfois difficile de déterminer la taille maximum de la liste qui représente un ensemble. On a déjà vu dans le chapitre sur les listes qu'une solution est d'utiliser des pointeurs pour chaîner les éléments entre eux. On va donc reprendre le type défini précédemment pour les listes, à savoir le type ENSC dans :

```
type cellules = record val : Elément; lien : ↑ cellules end;
     ENSC     = ↑ cellules;
```

Le test d'appartenance se fait de même par un parcours séquentiel de la liste :

```
function appartient(X : Elément; e : ENSC ) : boolean;
{cette fonction recherche séquentiellement si l'élément X figure ou non
dans l'ensemble e}
var p : ENSC;
begin p := e;
   while (p ≠nil) and (X ≠ p↑.val) do p := p↑.lien;
   if p =nil then return (false)
             else return (true)
end appartient;
```

Cette fonction s'obtient à partir de la fonction *appartient* du §3.1 en remplaçant i par p, la progression de i par la progression de p, et le test de fin de liste par $p \neq$nil. L'amélioration consistant à mettre X comme sentinelle à la fin de la liste nécessite d'avoir prévu un pointeur vers la fin de la liste car, sinon, il faudrait parcourir une première fois la liste pour mettre la sentinelle.

Les autres algorithmes ne sont pas non plus très différents de ceux vus pour les tableaux. On peut simplement remarquer que pour l'adjonction paresseuse, on ajoute l'élément en début de liste (sauf si on a un pointeur vers la fin de liste, par exemple à cause des sentinelles, auquel cas on peut indifféremment l'ajouter au début ou à la fin). De même, l'adjonction avec test préalable peut se faire indifféremment en fin de liste ou en début de liste.

La suppression se fait en mettant à jour des liens comme on l'a vu au chapitre 5, sans déplacer le dernier élément comme on le fait dans le cas des tableaux.

L'ensemble vide est représenté par le pointeur *nil*.

L'opération fondamentale prise en compte pour la complexité étant la comparaison entre éléments, et les programmes ne différant que par la progression et l'arrêt dans la liste, on retrouve les mêmes résultats de complexité que dans le cas des tableaux : toutes ces opérations, sauf l'adjonction paresseuse, sont en $\Theta(n)$ comparaisons au pire et en moyenne, où n est le nombre d'éléments de la liste.

Cette représentation peut être utilisée aussi pour les multi-ensembles. On peut également utiliser une liste chaînée où chaque élément apparaît au plus une fois, accompagné du nombre de ses occurrences.

4. Conclusion

On a étudié dans ce chapitre la représentation des ensembles dans le cas où le nombre d'éléments en jeu est petit. Dans tous les cas considérés l'une au moins des opérations est de complexité au pire et en moyenne :

- soit $\Theta(N)$, où N est le nombre des éléments possibles;
- soit $\Theta(n)$ où n le nombre des éléments présents dans l'ensemble (en tenant compte ou non des répétitions).

Cela signifie que ces méthodes deviennent vite inacceptables : au-delà de la centaine d'éléments.

On revient sur ce problème, comme cas particulier d'un problème plus général appelé la recherche associative, dans la troisième partie de ce livre. On y discute en particulier l'intérêt de représenter un ensemble par la liste triée de ses éléments, quand il existe une relation d'ordre sur ceux-ci. On voit également comment utiliser des arbres, ou des méthodes de partage en très petits sous-ensembles (hachage).

Exercices

1. On représente l'opération *choisir* de manière à ce qu'elle rende toujours le dernier élément ajouté et on n'autorise que la suppression de cet élément. Quel est le type de données obtenu ? Même question avec le premier élément.

2. Soit un ensemble e d'entiers compris entre 1 et N, représenté par un tableau de booléens.

a) Ecrire une procédure qui construit le sous-ensemble de e formé de tous les éléments de e qui sont des nombres premiers. On supposera qu'on dispose d'une fonction qui teste si un entier est un nombre premier.

b) Ecrire une procédure générale qui construit le sous-ensemble de e formé de tous les éléments de e qui vérifient une certaine propriété. On sait que la propriété est donnée sous la forme d'une fonction à résultat booléen.

c) Analyser cette dernière procédure.

3. Programmer et analyser l'union et l'intersection de deux ensembles :

a) dans le cas où les ensembles sont représentés par des tableaux de booléens ;

b) dans le cas où les ensembles sont représentés par des listes.

4. Programmer et analyser les opérations des multi-ensembles dans le cas où ceux-ci sont représentés par des listes chaînées où chaque élément apparaît au plus une fois, accompagné du nombre de ses occurrences.

5. Modifier la procédure *supprimer1* en utilisant X comme sentinelle.

6. Donner une version récursive du test d'appartenance dans le cas où l'ensemble est représenté par une liste chaînée. En déduire une version itérative équivalente.

7. Dessiner l'arbre de décision correspondant au test d'appartenance dans une liste de 12 éléments (indépendamment de la représentation de la liste).

Chapitre 7
Structures arborescentes

Un arbre est un ensemble de *nœuds*, organisés de façon hiérarchique, à partir d'un nœud distingué, appelé racine. La structure d'arbre est l'une des plus importantes et des plus spécifiques de l'informatique : par exemple, c'est sous forme d'arbre que sont organisés les fichiers dans des systèmes d'exploitation tels que UNIX ou MULTICS; c'est aussi sous forme d'arbres que sont représentés les programmes traités par un compilateur...

Une propriété intrinsèque de la structure d'arbre est la récursivité, et les définitions des caractéristiques des arbres, aussi bien que les algorithmes qui manipulent des arbres s'écrivent très naturellement de manière récursive.

1. Arbres binaires

Examinons tout d'abord quelques exemples simples représentés par des arbres binaires :

- les résultats d'un tournoi de tennis : la figure 1 montre qu'au premier tour, Marc a battu François, Jean a battu Jules et Luc a battu Pierre; au deuxième tour Jean a battu Marc, et Paul a battu Luc; et Jean a gagné en finale contre Paul.

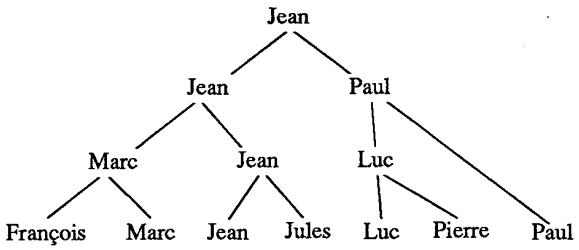


Figure 1. Tournoi de tennis.

– le pedigree d'un cheval de course : la figure 2 décrit le pedigree du cheval Zoe; son père est Tonnerre et sa mère Belle, la mère de Belle s'appelle Rose et son père Eclair...

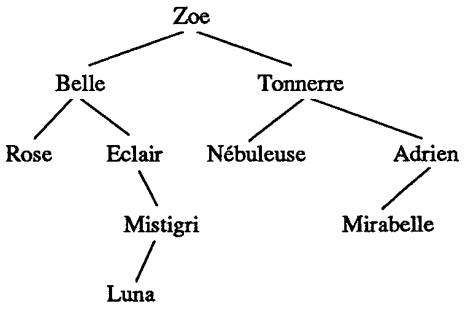


Figure 2. Pedigree de Zoe.

– une expression arithmétique dans laquelle tous les opérateurs sont binaires; l'arbre de la figure 3 code l'expression :

$$(x - (2 * y)) + ((x + (y/z)) * 3)$$

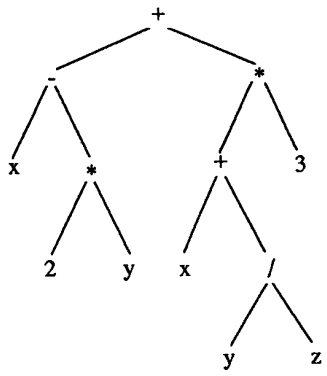


Figure 3. Expression arithmétique.

La structure d'arbre binaire est utilisée dans de très nombreuses applications informatiques; de plus les arbres binaires permettent de représenter les arbres plus généraux présentés au paragraphe 2.

1.1. Définitions

1.1.1. Le type arbre binaire

Définition 1 : Un ***arbre binaire*** est soit vide (noté Ø), soit de la forme $B =< o, B_1, B_2 >$, où B_1 et B_2 sont des arbres binaires disjoints et 'o' est un nœud appelé ***racine***.

Cette définition est récursive; elle peut s'écrire sous la forme de l'équation :

$$\mathbb{B} = Ø + < o,\ \mathbb{B},\ \mathbb{B} >$$

où $\mathbb{B}$ représente l'ensemble des arbres binaires, Ø représente l'arbre vide, et o désigne un nœud.

Remarque : Il est important de noter la **non-symétrie** gauche-droite des arbres binaires : l'arbre $< o,\ < o,\ Ø,\ Ø >,\ Ø >$ et l'arbre $< o,\ Ø,\ < o,\ Ø,\ Ø >>$ sont des arbres binaires différents. Pour pouvoir repérer les nœuds dans un arbre, il faut associer à chacun d'entre eux un nom différent. Sur la figure 4, on a représenté un arbre binaire en inscrivant auprès de chaque nœud une lettre indicée qui le désigne.

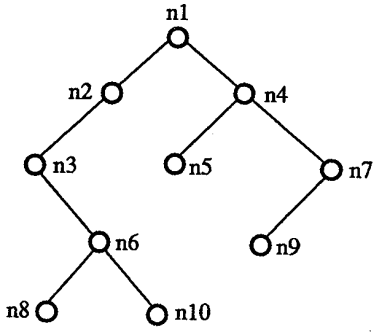


Figure 4. Représentation graphique d'un arbre binaire.

La signature du type abstrait arbre binaire est :

sorte Arbre
utilise Nœud, Elément
opérations

arbre-vide	:	→ Arbre
$< -, -, - >$	:	Nœud × Arbre × Arbre → Arbre
racine	:	Arbre → Nœud
g	:	Arbre → Arbre
d	:	Arbre → Arbre
contenu	:	Nœud → Elément

Etant donnés B_1 et B_2 des variables de sorte Arbre, et o une variable de sorte Nœud, on a les préconditions et axiomes suivants :

préconditions

racine(B_1) **est-défini-ssi** $B_1 \neq$ *arbre-vide*
$g(B_1)$ **est-défini-ssi** $B_1 \neq$ *arbre-vide*
$d(B_1)$ **est-défini-ssi** $B_1 \neq$ *arbre-vide*

axiomes

racine$(< o,\ B_1,\ B_2 >) = o$
$g(< o,\ B_1,\ B_2 >) = B_1$
$d(< o,\ B_1,\ B_2 >) = B_2$

L'opération *contenu* permet d'associer à chaque Nœud de l'Arbre une information de type Elément.

Un arbre dont les nœuds contiennent des éléments est dit ***arbre étiqueté.***

Si $B =< o,\ B_1,\ B_2 >$ est un arbre étiqueté dont la racine contient l'élément r, on note, abusivement, $B =< r,\ B_1,\ B_2 >$.

Le vocabulaire concernant les arbres informatiques est souvent emprunté à la botanique ou à la généalogie; étant donné un arbre $B =< o,\ B_1,\ B_2 >$:

• *'o'* est la ***racine*** de B.

• B_1 est le ***sous-arbre gauche*** de la racine de B, (ou, plus simplement, le sous-arbre gauche de B), et B_2 est son ***sous-arbre droit.***

• On dit que C est un ***sous-arbre*** de B si, et seulement si : $C = B$, ou $C = B_1$, ou $C = B_2$, ou C est un sous-arbre de B_1 ou de B_2.

• On appelle ***fils gauche*** (respectivement ***fils droit***) d'un nœud la racine de son sous-arbre gauche (respectivement sous-arbre droit), et l'on dit qu'il y a un ***lien gauche*** (respectivement droit) entre la racine et son fils gauche (respectivement fils droit).

• Si un nœud n_i a pour fils gauche (respectivement droit) un nœud n_j, on dit que n_i est le ***père*** de n_j (remarquons que d'après la définition 1 chaque nœud n'a qu'un seul père).

• Deux nœuds qui ont le même père sont dits ***frères.***

• Le nœud n_i est un ***ascendant*** ou un ***ancêtre*** du nœud n_j si, et seulement si, n_i est le père de n_j, ou un ascendant du père de n_j ; n_i est un ***descendant*** de n_j si, et seulement si n_i est fils de n_j, ou n_i est un descendant d'un fils de n_j.

• Tous les nœuds d'un arbre binaire ont au plus deux fils :
  – un nœud qui a deux fils est appelé ***nœud interne*** ou ***point double,***

- un nœud qui a seulement un fils gauche (respectivement droit) est dit ***point simple à gauche*** (resp. ***point simple à droite***), ou ***nœud interne au sens large***,
- un nœud sans fils est appelé ***nœud externe*** ou ***feuille***.

• On appelle ***branche*** de l'arbre B tout chemin (un chemin est une suite de nœuds consécutifs) de la racine à une feuille de B : un arbre a donc autant de branches que de feuilles.

• On appelle ***bord gauche*** (respectivement droit) de l'arbre B le chemin obtenu à partir de la racine en ne suivant que des liens gauches (respectivement droits) dans B.

1.1.2. Mesures sur les arbres

On introduit maintenant quelques opérations sur les arbres, à valeurs dans les entiers (ou dans les réels). Ces opérations seront utiles pour évaluer la complexité des algorithmes sur les arbres.

• La ***taille*** d'un arbre est le nombre de ses nœuds; on définit récursivement l'opération *taille* par :

taille(arbre-vide) = 0, et

taille$(< o,\ B_1,\ B_2 >) = 1 +$ *taille*$(B_1) +$ *taille*(B_2).

• La ***hauteur d'un nœud*** (on dit aussi ***profondeur*** ou ***niveau***) est définie récursivement de la façon suivante, étant donné x un nœud de B,

$h(x) = 0$ si x est la racine de B, et

$h(x) = 1 + h(y)$ si y est le père de x.

Ainsi la hauteur d'un nœud est comptée par le nombre de liens sur l'unique chemin de la racine à ce nœud.

• La ***hauteur*** ou ***profondeur*** d'un arbre B est :

$h(B) = \max\{h(x); x$ nœud de $B\}$

• La ***longueur de cheminement*** d'un arbre B est :

$LC(B) = \sum h(x)$, la somme étant prise sur tous les nœuds x de B.

Notons que l'on peut donner aussi une définition récursive de la hauteur et de la longueur de cheminement d'un arbre (cf. exercices).

• Dans la longueur de cheminement d'un arbre, on peut distinguer la contribution des feuilles de celle des autres nœuds :

- la ***longueur de cheminement externe*** d'un arbre B est
  $LCE(B) = \sum h(f)$, la somme étant prise sur toutes les feuilles f de B.
- la ***longueur de cheminement interne*** (au sens large) d'un arbre B est
  $LCI(B) = \sum h(x)$, la somme étant prise sur tous les nœuds internes (au sens large) x de B.

Et l'on a évidemment la relation $LC(B) = LCE(B) + LCI(B)$

Les quantités *longueurs de cheminement*, sont des sommes de profondeurs de nœuds ; en divisant chacune de ces quantités par le nombre de nœuds intervenant dans la somme, on obtient une notion de profondeur moyenne d'un nœud (quelconque, externe, ou interne au sens large).

Par exemple la ***profondeur moyenne externe*** d'un arbre B ayant f feuilles est

$$PE(B) = \frac{1}{f} LCE(B)$$

Il est important de ne pas confondre $PE(B)$ qui est la profondeur moyenne d'une feuille de B, et $h(B)$ qui est la profondeur maximale des feuilles de B.

Exemple : Sur l'arbre B de la figure 4, la racine n_1 a pour fils gauche n_2 et pour fils droit n_4 ; n_8 et n_{10} sont frères et n_6 est leur père ; les points doubles sont n_1, n_4 et n_6 ; n_8, n_9 et n_{10} sont des feuilles ; n_2 et n_7 sont des points simples à gauche, et n_3 est un point simple à droite ; le bord gauche est (n_1, n_2, n_3) ; le bord droit est (n_1, n_4, n_7) ; $(n_1, n_2, n_3, n_6, n_8)$ et $(n_1, n_2, n_3, n_6, n_{10})$ sont deux branches de l'arbre. Les hauteurs des nœuds sont $h(n_1) = 0$, $h(n_2) = h(n_4) = 1$, $h(n_3) = h(n_5) = h(n_7) = 2$, $h(n_6) = h(n_9) = 3$ et $h(n_8) = h(n_{10}) = 4$.

Caractéristiques de l'arbre : $h(B) = 4$; $LC(B) = 21$; $LCI(B) = 9$; $LCE(B) = 13$; $PE(B) = 13/4 = 3.24$.

1.1.3. Arbres binaires particuliers

On définit ici certaines formes particulières d'arbres binaires qui jouent un rôle important par la suite : les formes extrêmes d'arbres binaires (dégénérés , complets ou parfaits), et les arbres binaires sans points simples (localement complets).

• Un arbre binaire ***dégénéré*** ou ***filiforme*** est un arbre formé uniquement de points simples (voir figure 5.a).

• Un arbre binaire est dit ***complet*** s'il contient 1 nœud au niveau 0, 2 nœuds au niveau 1, 4 nœuds au niveau 2,..., 2^h nœuds au niveau h (voir figure 5.b). Dans un tel arbre on dit aussi que chaque *niveau est complètement rempli.*

Le nombre total de nœuds d'un arbre complet de hauteur h est donc $n = 1 + 2 + ... + 2^h = 2^{h+1} - 1$.

La définition suivante élargit la notion d'arbre complet aux arbres de taille quelconque.

• Un arbre binaire ***parfait*** est un arbre binaire dont tous les niveaux sont complètement remplis, sauf éventuellement le dernier niveau, et dans ce cas les nœuds (feuilles) du dernier niveau sont groupés le plus à gauche possible.

Par exemple dans la figure 6, l'arbre (a) est parfait mais les arbres (b) et (c) ne sont pas parfaits.

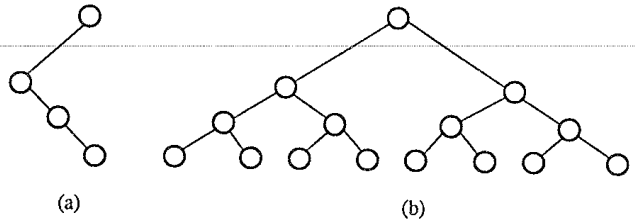


Figure 5. Arbres binaires extrêmes (a) dégénéré, (b) complet.

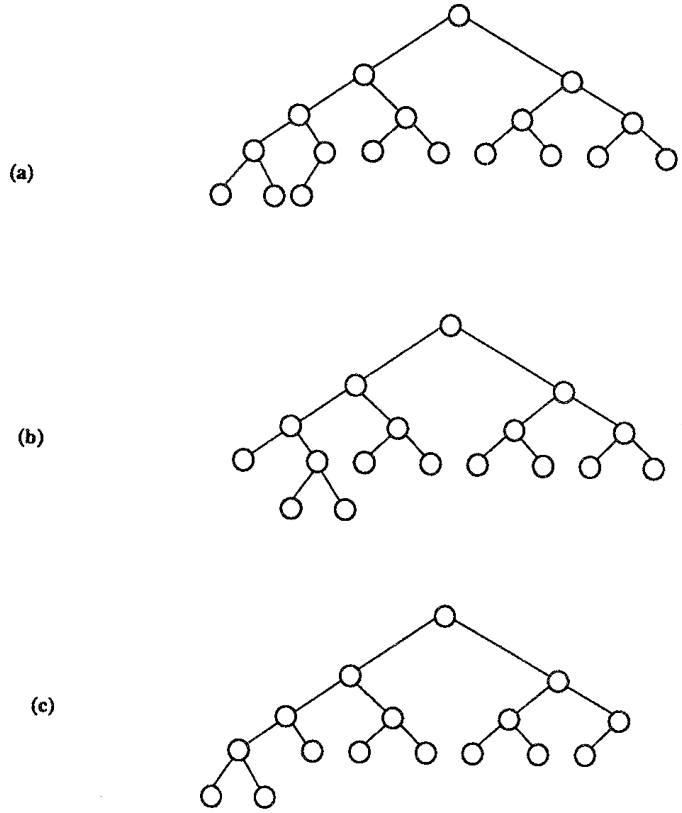


Figure 6. Arbres parfait (a), et non parfaits (b) et (c).

• Un arbre binaire ***localement complet*** est un arbre binaire non vide qui n'a pas de point simple : tous les nœuds qui ne sont pas des feuilles ont deux fils.

Par exemple, les arbres binaires des figures 1 et 3 sont localement complets, mais celui de la figure 2 ne l'est pas.

On peut aussi donner une définition récursive des arbres binaires localement complets analogue à la définition récursive des arbres binaires :

$$\mathbb{BC} =< o,\ \emptyset,\ \emptyset > + < o,\ \mathbb{BC},\ \mathbb{BC} >$$

où $\mathbb{BC}$ représente l'ensemble des arbres binaires localement complets, et o désigne un nœud.

• Un ***peigne gauche*** (respectivement ***un peigne droit***) est un arbre binaire localement complet dans lequel tout fils droit (respectivement gauche) est une feuille (voir figure 7). Si $\mathbb{PG}$ représente l'ensemble des peignes gauches, on a donc

$$\mathbb{PG} =< o,\ \emptyset,\ \emptyset > + < o,\ \mathbb{PG},\ < o,\ \emptyset,\ \emptyset >> .$$

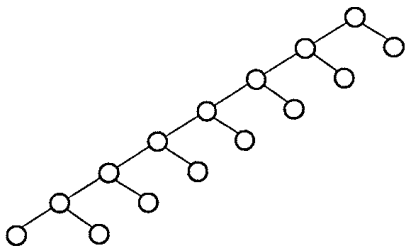


Figure 7. Arbre peigne gauche.

1.1.4. Occurrence et numérotation hiérarchique

Une façon classique de désigner un nœud dans un arbre est de lui associer un mot formé de symboles '0' et '1', qui décrit le chemin de la racine de l'arbre à ce nœud; ce mot est appelé ***occurrence*** du nœud dans l'arbre. Par définition l'occurrence de la racine d'un arbre est le mot vide ε et si un nœud de l'arbre a pour occurrence μ, son fils gauche a pour occurrence $\mu 0$, et son fils droit a pour occurrence $\mu 1$.

L'intérêt de cette notation est de permettre le codage d'un arbre par un ensemble de mots. Par exemple, l'arbre de la figure 4 est représenté par l'ensemble :

$$\{\varepsilon,\ 0,\ 1,\ 00,\ 10,\ 11,\ 001,\ 110,\ 0010,\ 0011\}$$

Il est facile de déterminer à partir de cet ensemble de mots les caractéristiques de l'arbre qu'il représente (cf. exercices).

De plus, ce codage des nœuds est lié à la numérotation en ***ordre hiérarchique***. Dans la numérotation en ordre hiérarchique des nœuds d'un arbre complet, on numérote en ordre croissant à partir de 1 tous les nœuds à partir de la racine, niveau par niveau, et de gauche à droite sur chaque niveau (voir figure 8). Ainsi un nœud numéroté i dans la numérotation en ordre hiérarchique a son fils gauche numéroté par $2i$, et son fils droit par $2i + 1$. Il en résulte que si un nœud d'un arbre complet a pour occurrence μ et pour numérotation en ordre hiérarchique i, on a la relation

$i = 2^{\lfloor \log_2 i \rfloor} + m$, où m est l'entier représenté par μ.

La preuve de ces propriétés est reportée en exercice.

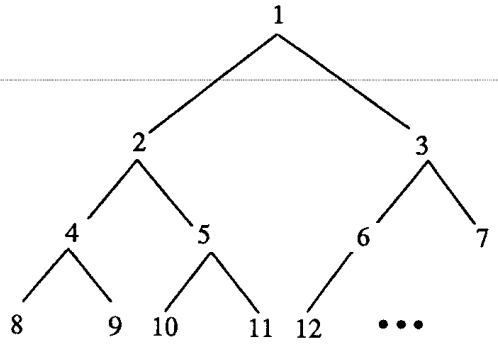


Figure 8. Ordre hiérarchique.

1.2. Propriétés fondamentales

La profondeur et la longueur de cheminement sont des quantités importantes pour l'analyse de la complexité d'un certain nombre d'algorithmes : on verra par la suite que la complexité, en temps ou en place, des algorithmes manipulant des arbres, s'exprime souvent en fonction de ces deux mesures. Il est donc intéressant de montrer les relations qui existent entre le nombre total de nœuds d'un arbre et son nombre de feuilles, sa hauteur ou sa longueur de cheminement. On donne aussi dans ce paragraphe des résultats de dénombrement d'arbres selon leur taille, qui sont utiles pour l'analyse de la complexité en moyenne.

1.2.1. Encadrements de la hauteur et de la longueur de cheminement

Lemme 1 : Pour un arbre binaire ayant n nœuds au total et de hauteur h , on a : $\lfloor \log_2 n \rfloor \leq h \leq n - 1$.

Preuve : Pour une hauteur h donnée, les arbres ayant le plus petit nombre de nœuds sont les arbres dégénérés (voir figure 5.a) ; et si B est un arbre dégénéré de hauteur h, alors sa taille est $n = h + 1$, d'où la seconde inégalité.

D'autre part, l'arbre de hauteur h ayant le plus grand nombre de nœuds est l'arbre complet (voir figure 5.b). Or, un arbre complet de hauteur h a pour taille $2^{h+1} - 1$.

Donc pour tout arbre binaire on a $n < 2^{h+1}$, ce qui implique $\log_2 n < h + 1$, d'où $\lfloor \log_2 n \rfloor \leq h$. □

Remarque : Un arbre binaire parfait avec n nœuds a pour hauteur $\lfloor \log_2 n \rfloor$. Cela provient de ce qu'un arbre binaire parfait de hauteur h contient entre 2^h nœuds et $2^{h+1} - 1$ nœuds.

Corollaire 1 : Tout arbre binaire non vide B ayant f feuilles a une hauteur $h(B)$ supérieure ou égale à $\lceil \log_2 f \rceil$.

Preuve : Montrons tout d'abord par récurrence que pour tout arbre binaire ayant n nœuds au total, dont f feuilles, on a $f \leq \frac{n+1}{2}$:

– la propriété est vraie pour l'arbre réduit à une feuille,
– supposons-la vraie pour tous les arbres binaires ayant moins de n nœuds.

Soit $B =< o,\ B1,\ B2 >$ un arbre ayant n nœuds et f feuilles. Notons n_1 et n_2 les nombres de nœuds de B_1 et de B_2, et f_1 et f_2 leurs nombres de feuilles respectifs; on a $n = n_1 + n_2 + 1$, et $f = f_1 + f_2$.

Par hypothèse de récurrence, on a $f_1 \leq \frac{n_1+1}{2}$ et $f_2 \leq \frac{n_2+1}{2}$, donc le nombre de feuilles de B est $f = f_1 + f_2 \leq \frac{n_1+n_2+2}{2} = \frac{n+1}{2}$, ce qui achève la preuve par récurrence.

Maintenant comme la fonction log est croissante sur $\mathbb{R}^+$, on a $\log_2 f \leq \log_2\left(\frac{n+1}{2}\right)$, soit $1 + \log_2 f \leq \log_2(n+1)$.

En prenant la partie entière supérieure on obtient $1 + \lceil \log_2 f \rceil \leq \lceil \log_2(n+1) \rceil$.
Il en résulte que $\lceil \log_2 f \rceil \leq \lfloor \log_2 n \rfloor$ (car $\lceil \log_2(n+1) \rceil = 1 + \lfloor \log_2 n \rfloor$).
En utilisant le lemme 1, on conclut donc que $\lceil \log_2 f \rceil \leq h(B)$. □

Lemme 2 : Soit B un arbre binaire ayant n nœuds au total, on a l'encadrement suivant pour la longueur de cheminement de B :

$$\sum_{1 \leq k \leq n} \lfloor \log_2 k \rfloor \leq LC(B) \leq \frac{n(n-1)}{2}$$

Preuve : Si l'arbre B est dégénéré, on a $LC(B) = 0 + 1 + 2 + ... + n - 1 = \frac{n(n-1)}{2}$.

A l'opposé, les arbres binaires qui minimisent la longueur de cheminement sont ceux qui ont toutes leurs feuilles situées sur deux niveaux au plus (voir exercices). On va supposer ici que les feuilles du dernier niveau sont le plus à gauche possible, et calculer la longueur de cheminement des arbres parfaits.

Soit B l'arbre parfait de taille n. Si les nœuds de B sont numérotés de 1 à n en ordre hiérarchique, alors le nœud numéroté k est à la profondeur $\lfloor \log_2 k \rfloor$; on montre cette propriété par récurrence sur la profondeur des nœuds d'un arbre parfait : le nœud numéroté 1 est à profondeur $0 = \lfloor \log_2 1 \rfloor$; supposons que les nœuds à profondeur k sont numérotés de 2^k à $2^{k+1} - 1$, et considérons le niveau

$k+1$: le nœud le plus à gauche a pour numéro $(2^{k+1}-1)+1 = 2^{k+1}$; de plus il y a 2 fois plus de nœuds qu'au niveau précédent; à profondeur $k+1$ il y a donc $2 \cdot 2^k$ nœuds, dont les numéros vont de 2^{k+1} à $2^{k+1}+2^{k+1}-1 = 2^{k+2}-1$.

Il en résulte que $LC(B) = \sum_{1 \leq k \leq n} \lfloor \log_2 k \rfloor$. □

Corollaire 2 : La longueur de cheminement d'un arbre binaire ayant n nœuds au total, est au minimum de l'ordre de $n \log_2 n$ et au maximum de l'ordre de n^2.

La preuve de ce corollaire est proposée en exercice. □

Il sera aussi utile pour la suite de connaître un minorant de la longueur de cheminement externe, et donc de la profondeur moyenne des feuilles, d'un arbre binaire ayant f feuilles.

Lemme 3 : Tout arbre binaire B ayant f feuilles a une profondeur moyenne externe $PE(B)$ supérieure ou égale à $\log_2 f$.

Preuve : On raisonne par récurrence. La propriété est vraie pour l'arbre réduit à une feuille, qui a pour profondeur 0. Supposons-la vraie pour tous les arbres ayant strictement moins de k feuilles. Soit $B =< o,\ B1,\ B2 >$ un arbre ayant k feuilles. B est de l'une des trois formes de la figure 9.

• Si B est de la forme a) ou b), alors B' est un arbre avec k feuilles, et l'on cherche à prouver la propriété sur B', ce qui la prouvera *a fortiori* sur B.

• Si B est de la forme c), ses sous-arbres gauche et droit ont strictement moins de k feuilles donc ils vérifient le lemme : si k_1 (respectivement k_2) est le nombre de feuilles de B_1 (respectivement B_2) on a $PE(B_1) \geq \log_2 k_1$ et $PE(B_2) \geq \log_2 k_2$;
or $PE(B) = 1 + \frac{k_1}{k_1+k_2}\ PE(B_1) + \frac{k_2}{k_1+k_2}\ PE(B_2)$
puisque $LCE(B) = k_1 + LCE(B_1) + k_2 + LCE(B_2)$; d'où

$$PE(B) \geq 1 + \frac{k_1}{k_1+k_2} \log_2 k_1 + \frac{k_2}{k_1+k_2} \log_2 k_2$$

Puisque $k_1 + k_2 = k$, le terme de droite de l'inéquation s'écrit :

$$q(k_2) = 1 + \frac{k-k_2}{k} \log_2 (k-k_2) + \frac{k_2}{k} \log_2 k_2$$

Pour k fixé, il est facile de montrer que $q(k_2)$ est minimum lorsque $k_2 = k/2$ (c'est en cette valeur que la dérivée s'annule), et l'on a $q(k/2) = \log_2 k$. En conséquence $PE(B) \geq \log_2 k$. □

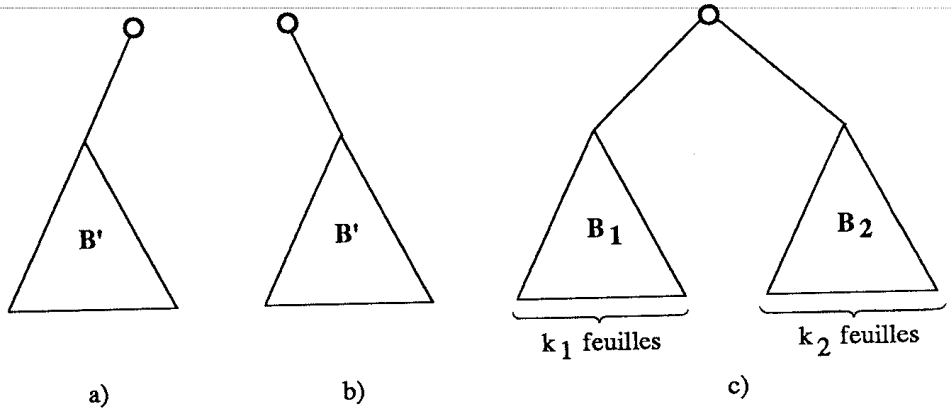


Figure 9

Remarque : Le lemme 3 et le corollaire 1 donnent des valeurs très proches pour les minorants de $h(B)$ et $PE(B)$, mais cela ne veut pas dire que les valeurs de $h(B)$ et $PE(B)$ sont très proches. Considérons l'arbre peigne B_0 de la figure 7. B_0 a 8 feuilles, donc $h(B_0) \geq 3$ d'après le corollaire1 et $PE(B_0) \geq 3$ d'après le lemme 3. Or, en fait, on calcule que $h(B_0) = 7$ et $PE(B_0) = 4.37$.

1.2.2. Propriétés des arbres binaires localement complets

Lemme 4 : Un arbre binaire localement complet ayant n nœuds internes a $(n+1)$ feuilles.

Preuve : La preuve se fait par récurrence sur le nombre de nœuds internes de l'arbre.

- La propriété est vraie pour l'arbre ayant 0 nœud interne et 1 feuille,
- Supposons-la vraie pour tous les arbres ayant moins de n nœuds internes, et soit $B =< o, B_1, B_2 >$ ayant n nœuds internes.

Notons n_1 (respectivement n_2) le nombre de nœuds internes de B_1 (respectivement B_2); on a : $n = n_1 + n_2 + 1$.

Par hypothèse de récurrence, B_1 (respectivement B_2) a $n_1 + 1$ (respectivement $n_2 + 1$) feuilles. Or les feuilles de B sont feuilles de B_1 ou de B_2, donc le nombre de feuilles de B est $(n_1 + 1) + (n_2 + 1) = n + 1$. □

Pour un arbre binaire localement complet, les longueurs de cheminement interne et externe vérifient la relation suivante.

Lemme 5 : Soit B un arbre binaire localement complet ayant n nœuds internes :

$$LCE(B) = LCI(B) + 2n.$$

La preuve est laissée en exercice. □

Par exemple, pour l'arbre binaire localement complet B de la figure 3, on a $n = 6$, $LCI(B) = 9$ et $LCE(B) = 21$.

On précise à présent le rapport entre les arbres binaires et les arbres binaires localement complets.

Définition : On appelle ***complétion locale d'un arbre binaire B*** l'opération qui consiste à compléter l'arbre B en un arbre BC, en rajoutant des feuilles de telle sorte que chaque nœud de B, interne ou externe, ait deux fils dans BC.

Par exemple sur la figure 10, l'arbre BC est obtenu par complétion locale de l'arbre B : on a noté par des carrés les feuilles de BC, qui sont les nœuds ajoutés par l'opération de complétion.

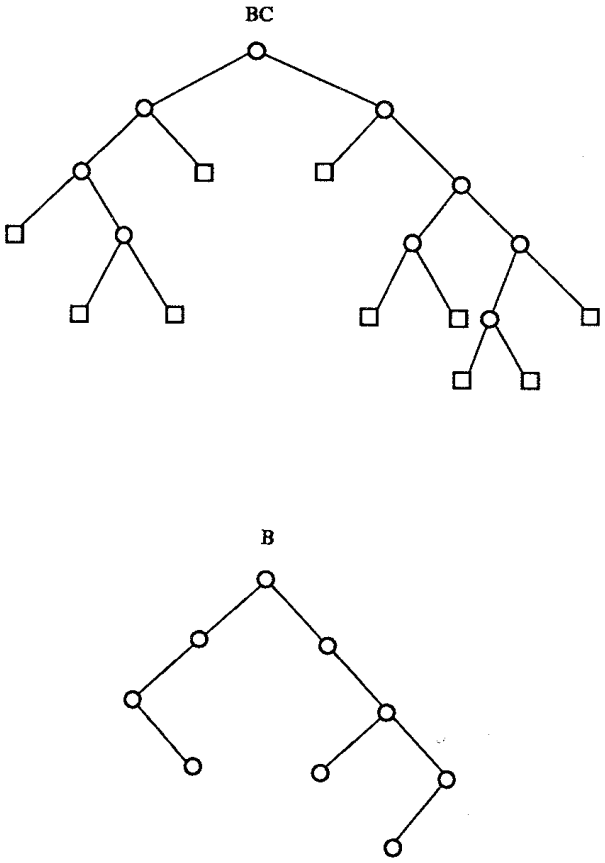


Figure 10. Complétion locale d'un arbre binaire.

Proposition 1 : Il existe une bijection entre l'ensemble $\mathbb{B}_n$ des arbres binaires ayant n nœuds, et l'ensemble $\mathbb{BC}_n$ des arbres binaires localement complets ayant $2n+1$ nœuds.

Preuve : D'après le lemme 4, si B est un arbre binaire localement complet ayant $2n+1$ nœuds, l'arbre B' obtenu en supprimant toutes les feuilles de B est un arbre binaire ayant n nœuds. Cette application est bijective : son inverse est l'opération de complétion locale. □

1.2.3. Dénombrement des arbres binaires

Pour faire une analyse en moyenne d'algorithmes opérant sur des arbres binaires, il est nécessaire de connaître le nombre b_n d'arbres binaires de taille donnée n. La figure 11 énumère les arbres binaires de taille $n = 0, 1, 2, 3$.

Proposition 2 : Le nombre d'arbres binaires de taille n est $b_n = \frac{1}{n+1}\binom{2n}{n}$.

La *preuve* de cette proposition est donnée dans le paragraphe sur les séries génératrices de l'annexe. □

Le nombre binômial, qui représente le nombre de façons de choisir n éléments parmi $2n$, a été noté ici $\binom{2n}{n}$; il est aussi quelquefois noté C_{2n}^{n}.

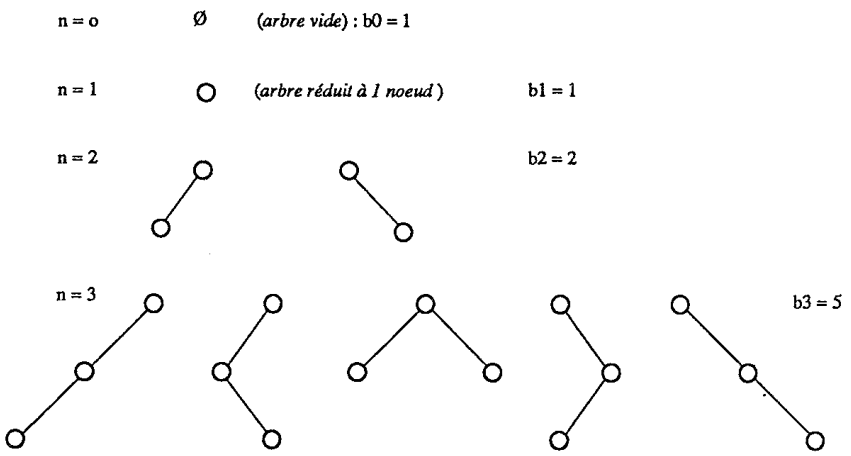


Figure 11. Enumération des arbres binaires de taille $0, 1, 2, 3$.

Comme conséquence des propositions 1 et 2, on a le corollaire suivant.

Corollaire 3 : Le nombre bc_n d'arbres binaires localement complets ayant $(2n+1)$ nœuds est $bc_n = \frac{1}{n+1}\binom{2n}{n}$.

La figure 12 montre les 5 arbres binaires localement complets avec 7 nœuds :

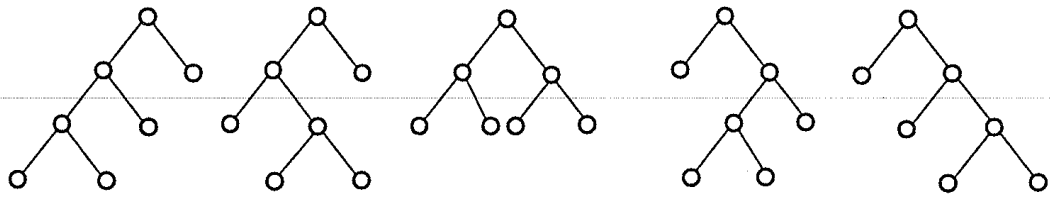


Figure 12. Arbres binaires localement complets avec 7 nœuds.

1.3. Représentation des arbres binaires

On décrit ici quelques façons d'implémenter les arbres binaires, qui seront utilisées par la suite. Il y a bien d'autres représentations possibles; dans une application donnée, c'est l'étude des opérations que l'on veut réaliser sur les arbres qui permet de déterminer la représentation la mieux adaptée au problème.

La représentation la plus naturelle reproduit la définition récursive des arbres binaires. Elle peut être réalisée en allouant la mémoire soit de façon chaînée soit de façon contiguë.

1.3.1. Utilisation de pointeurs

A chaque nœud on associe deux pointeurs, l'un vers le sous-arbre gauche, l'autre vers le sous-arbre droit, et l'arbre est déterminé par l'adresse de sa racine. Lorsque l'arbre est étiqueté, on représente dans un champ supplémentaire l'information contenue dans le nœud. Avec cette représentation, un arbre étiqueté comme celui de la figure 3 peut être défini par le type Pascal suivant :

```
type ARBRE = ↑ Nœud;
     Nœud = record val : Elément;
                   g, d : ARBRE
            end;
```

La figure 13 montre un arbre binaire A de type ARBRE. L'arbre vide ($A = nil$) est représenté par une case barrée.

Dans une telle représentation, les opérations du type abstrait Arbre donné au paragraphe 1.1 se traduisent comme suit : $A = nil$ correspond à $A =$ *arbre-vide*, $A\uparrow.val$ correspond à *contenu*(*racine*(A)), $A\uparrow.g$ correspond à $g(A)$, et $A\uparrow.d$ correspond à $d(A)$.

1.3.2. Utilisation de tableaux

Dans le cas où l'on n'a pas la possibilité d'allouer dynamiquement la mémoire, on peut simuler la représentation précédente à l'aide de tableaux : à chaque nœud de

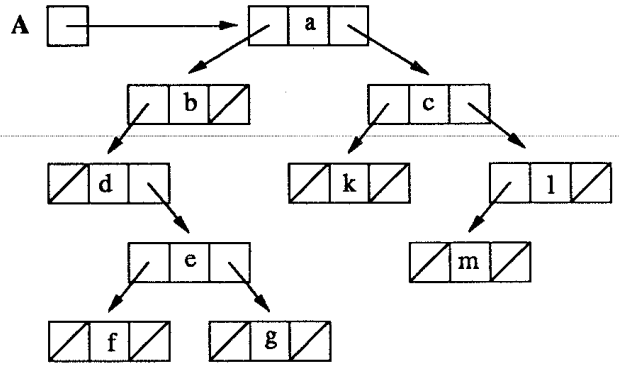


Figure 13. Représentation d'un arbre binaire à l'aide de pointeurs.

l'arbre on associe un indice dans un tableau à deux champs G et D. Le champ G (respectivement D) contient l'indice associé à la racine du sous-arbre gauche (respectivement droit). Il faut une convention pour l'indice associé à l'arbre vide. Il faut aussi connaître l'indice dans le tableau de la racine de l'arbre. De plus, lorsque l'arbre est étiqueté, on représente dans un champ supplémentaire V l'information contenue dans le nœud.

Avec cette représentation, un arbre étiqueté, contenant moins de N nœuds, peut être défini par le type Pascal ARBTAB, qui utilise le type TAB :

```
type    TAB = array [1..N] of record V : Elément;
                                     G : 0..N;
                                     D : 0..N
                              end;
      ARBTAB = record rac : 0..N;
                      tab : TAB
               end;
```

Dans ce cas, l'arbre de la figure 13 peut être représenté par la variable A de type ARBTAB, dont le champ *rac* contient 3, et le champ *tab* contient le tableau de la figure 14 ($N = 13$ est la valeur maximale de la taille des arbres qui peuvent être rangés dans ce tableau).

Voyons comment se traduisent les opérations du type abstrait Arbre sur une variable A de type ARBTAB. Par convention si $A.rac = 0$, on considère que A représente l'arbre vide. Sinon soit $r > 0$ tel que $r = A.rac$, alors $A.tab[r]$ correspond à $racine(A)$, $A.tab[r].V$ est le contenu de cette racine, et $A.tab[r].G$ et $A.tab[r].D$ sont les indices de $g(A)$ et $d(A)$.

Notons que dans la représentation indexée, l'indice associé à chaque nœud de l'arbre est arbitraire; on peut donc ajouter ou enlever des nœuds de l'arbre sans difficulté, à condition de «chaîner» entre elles les cases libres de façon analogue à ce qui

	V	G	D
1			
2	d	0	10
3	a	5	6
4	g	0	0
5	b	2	0
6	c	13	11
7			
8	f	0	0
9	m	0	0
10	e	8	4
11	l	9	0
12			
13	k	0	0

Figure 14. Représentation indexée d'un arbre binaire.

a été fait au chapitre 5 pour les listes; on conserve ainsi, dans les limites de la place réservée pour le tableau, l'un des avantages de l'allocation chaînée. Avec des déclarations de types légèrement différentes, on pourrait représenter plusieurs arbres dans le même tableau, de façon analogue à ce qui a été fait pour les listes.

1.3.3. Cas particulier : les arbres binaires parfaits

Les arbres parfaits ont une représentation séquentielle très compacte qui repose sur la numérotation en ordre hiérarchique des nœuds d'un arbre dans un parcours par niveaux (figure 8).

L'utilisation de la numérotation en ordre hiérarchique permet de coder les arbres parfaits de taille n dans un tableau de n cases, comme le montre la figure 15. Si un nœud est numéroté par i dans la numérotation hiérarchique, alors son fils gauche est numéroté par $2i$, et son fils droit par $2i+1$. Donc dans cette représentation le passage d'un nœud à un autre se traduit par un simple calcul d'indice dans le tableau, calcul qu'il est bon de mémoriser dans le schéma suivant :

$2 \leq i \leq n \Rightarrow$ le père du nœud d'indice i est à l'indice i **div** 2
$1 \leq i \leq n$ **div** $2 \Rightarrow$ le fils gauche du nœud d'indice i est en $2i$
le fils droit du nœud d'indice i est en $2i+1$

Notons que cette représentation peut être utilisée pour des arbres binaires quelconques, en laissant des cases vides dans le tableau, qui marquent la place des nœuds

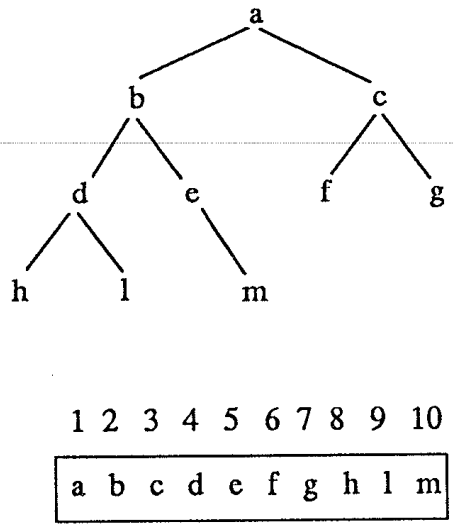


Figure 15. Représentation d'un arbre parfait dans un tableau.

possibles non présents. Mais elle perd alors son avantage de compacité : la représentation d'un arbre à n nœuds peut nécessiter jusqu'à $2^n - 1$ cases, comme le montre l'exemple de la figure 16.

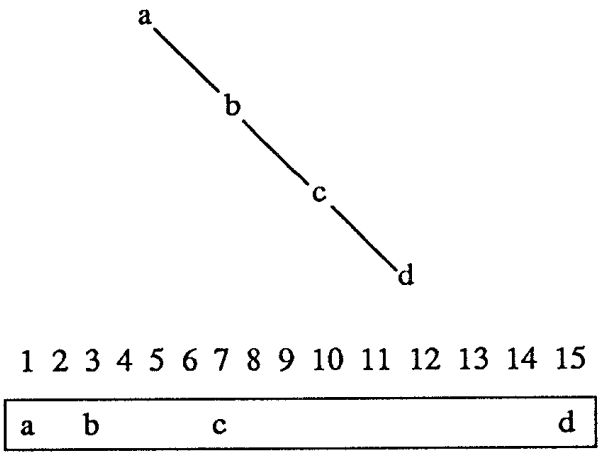


Figure 16. Représentation d'une branche droite par un tableau.

1.4. Parcours d'un arbre binaire

Une des opérations les plus fréquentes mise en oeuvre par les algorithmes qui manipulent des arbres consiste à examiner systématiquement, dans un certain ordre, chacun des nœuds de l'arbre pour y effectuer un même traitement; cette opération est appelée ***parcours*** ou ***traversée*** de l'arbre. On étudie ici un algorithme général de parcours d'un arbre binaire, que l'on appelle parcours en profondeur à main gauche

– sous forme récursive et sous forme itérative – et trois ordres classiques induits sur les nœuds.

Le ***parcours en profondeur à main gauche*** consiste à tourner autour de l'arbre en suivant le chemin indiqué sur la figure 17. Ce chemin part à gauche de la racine, et va toujours le plus à gauche possible en suivant l'arbre (notons que l'on a fait figurer les sous-arbres vides de l'arbre binaire).

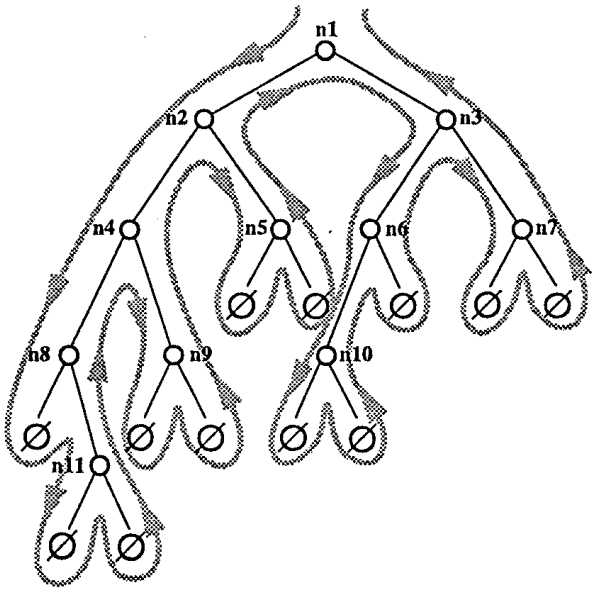


Figure 17. Parcours en profondeur à main gauche.

Dans ce parcours, chaque nœud de l'arbre est rencontré trois fois : d'abord à la descente (figure 18.a), puis en montée gauche, après son sous-arbre gauche (figure 18.b), et enfin en montée droite, après son sous-arbre droit (figure 18.c).

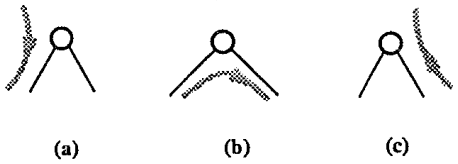


Figure 18. Rencontres des nœuds lors d'un parcours.

1.4.1. Algorithme récursif de parcours

Lors du parcours en profondeur à main gauche de l'arbre A, on peut faire correspondre à chacune des rencontres d'un nœud un traitement particulier pour ce nœud. Appelons TRAITEMENT1 (resp. TRAITEMENT2 ET TRAITEMENT3) la suite d'actions exécutées lorsqu'un nœud est rencontré pour la première (resp. deuxième et troisième) fois; de plus, envisageons pour les arbres vides un traitement

spécial : TERMINAISON. Le parcours de l'arbre est alors décrit récursivement par la procédure *Parcours* ci-dessous. Remarquons ici que la complexité, comptée en nombre de traitements de nœuds, de l'algorithme de parcours, sur un arbre comportant n nœuds , est clairement en $\Theta(n)$.

```
procedure Parcours(A : Arbre);
begin
    if A = arbre-vide then TERMINAISON
    else     begin        TRAITEMENT1;
                          Parcours(g(A));
                          TRAITEMENT2;
                          Parcours(d(A));
                          TRAITEMENT3
             end
end Parcours;
```

Le parcours en profondeur à main gauche contient comme cas particuliers trois ordres classiques d'exploration d'arbres :

1) ***ordre préfixe*** ou ***préordre*** : si TRAITEMENT2 et TRAITEMENT3 n'existent pas, alors TRAITEMENT1 est appliqué aux nœuds de l'arbre en ordre préfixe; sur l'exemple de la figure 17 cela correspond à l'ordre de traitement n_1, n_2, n_4, n_8, n_{11}, n_9, n_5, n_3, n_6, n_{10}, n_7;

2) ***ordre infixe*** ou ***symétrique*** : c'est l'ordre obtenu lorsque seul TRAITEMENT2 (et TERMINAISON) sont appliqués; sur l'exemple cela correspond à l'ordre de traitement n_8, n_{11}, n_4, n_9, n_2, n_5, n_1, n_{10}, n_6, n_3, n_7;

3) ***ordre suffixe*** ou ***postfixe*** : c'est l'ordre obtenu lorsque seul TRAITEMENT3 (et TERMINAISON) sont appliqués; sur l'exemple cela correspond à l'ordre de traitement n_{11}, n_8, n_9, n_4, n_5, n_2, n_{10}, n_6, n_7, n_3, n_1.

Remarque : On ne peut pas obtenir ainsi l'ordre hiérarchique (figure 8), car il correspond à un *parcours par niveaux* de l'arbre et non à un parcours en profondeur.

1.4.2. Arbre étiqueté représentant une expression arithmétique

Une expression arithmétique peut être représentée par un arbre binaire dont les feuilles contiennent des symboles de variables (x, y, z...) ou de constantes (2, 3...) et les nœuds internes contiennent des symboles d'opérateurs (cf. figure 19). Si l'on ne considère que des opérateurs binaires, comme +, − , *, /, l'arbre binaire est localement complet.

Pour un tel arbre, les suites des éléments en ordre préfixe, suffixe et symétrique, correspondent aux différentes représentations habituelles d'une expression arithmétique :

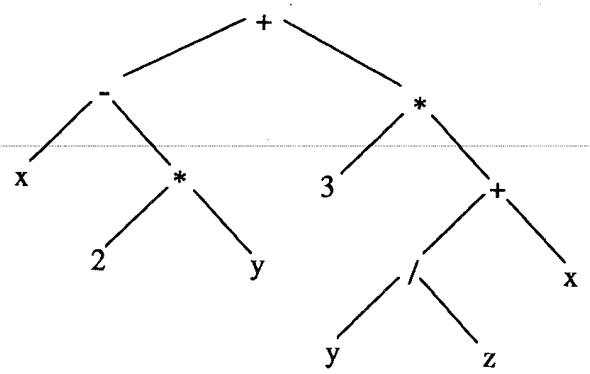


Figure 19. Arbre binaire représentant une expression arithmétique.

- en ordre préfixe, on obtient l'expression sous forme polonaise préfixée : chaque opérateur précède ses deux opérandes; ce qui donne sur l'exemple de la figure 19 : $+ - x * 2y * 3 + /yzx$.
- en ordre suffixe, on obtient l'expression sous forme polonaise postfixée : chaque opérateur est précédé par ses deux opérandes; ce qui donne pour l'exemple : $x2y * -3yz/x + *+$

Remarquons que pour les deux formes polonaises, la notation est non ambiguë, et il est donc inutile de mettre des parenthèses autour des sous-expressions.

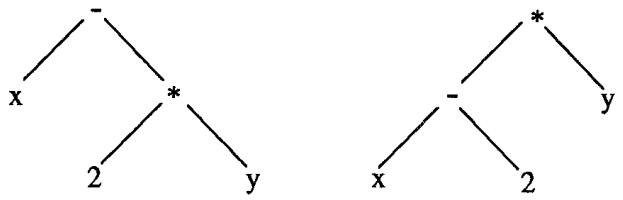


Figure 20. Ambiguïté de la notation symétrique.

Par contre la lecture des éléments de l'arbre en ordre symétrique produit la notation symétrique habituelle, mais sans parenthèses, ce qui la rend ambiguë : par exemple, l'expression $x - 2 * y$ représente les deux arbres différents de la figure 20.

L'ambiguïté peut être levée en introduisant des parenthèses. Les deux arbres de la figure 20 sont respectivement représentés par les expressions bien parenthésées $(x - (2 * y))$ et $((x - 2) * y)$. L'algorithme ci-dessous écrit l'expression symétrique bien parenthésée correspondant à un arbre binaire non vide A. On utilise l'opération *feuille* de profil

feuille : Arbre $\rightarrow$ Booléen,

et qui satisfait la propriété

feuille(A) = vrai **ssi** $A \neq$ *arbre-vide* & $g(A) =$*arbre-vide* & $d(A) =$ *arbre-vide*.

```
procedure Sym (A : Arbre);
{Cette procédure écrit l'expression symétrique bien parenthésée correspondant
à l'arbre binaire A supposé non vide}
begin
    if feuille(A) then write(contenu(racine(A)))
        else begin write('(');
                   Sym(g(A));
                   write(contenu(racine(A)));
                   Sym(d(A));
                   write (')')
             end
end Sym;
```

1.4.3. Version itérative du parcours d'arbres binaires

La procédure *Parcours* est une version récursive du parcours en profondeur à main gauche d'un arbre binaire. Dans ce parcours chaque nœud est rencontré trois fois : une première fois à la descente, et l'on effectue le TRAITEMENT1 (en abrégé T_1), une deuxième fois en remontée par la gauche et l'on effectue le TRAITEMENT2 (T_2) et une troisième fois en remontée par la droite et l'on effectue le TRAITEMENT3 (T_3). Par exemple sur l'arbre de la figure 17, si l'on note (T_i, n_k) l'action d'effectuer T_i sur le nœud n_k, le parcours général donne la suite :
(T_1, n_1), (T_1, n_2), (T_1, n_4), (T_1, n_8), (T_2, n_8), (T_1, n_{11}), (T_2, n_{11}), (T_3, n_{11}), (T_3, n_8), (T_2, n_4), (T_1, n_9), (T_2, n_9), (T_3, n_9), (T_3, n_4), (T_2, n_2), (T_1, n_5), (T_2, n_5), (T_3, n_5), (T_3, n_2), (T_2, n_1), (T_1, n_3), (T_1, n_6), (T_1, n_{10}), (T_2, n_{10}), (T_3, n_{10}), (T_2, n_6), (T_3, n_6), (T_2, n_3), (T_1, n_7), (T_2, n_7), (T_3, n_7), (T_3, n_3), (T_3, n_1).

L'examen du parcours général suggère l'algorithme itératif suivant : descendre la branche gauche de l'arbre en effectuant T_1, et en empilant les nœuds rencontrés; c'est avec ces nœuds qu'il faudra effectuer T_2 lors de la remontée par la gauche; il est donc nécessaire de les conserver. De plus l'ordre de rencontre des nœuds pour T_2 est inverse de l'ordre de rencontre pour T_1, d'où la structure de pile pour conserver l'information; ensuite lorsqu'un nœud est rencontré en remontée gauche il faut effectuer T_2, mémoriser le nœud dans une pile, puis il faut travailler sur tout son sous-arbre droit, et enfin traiter le nœud lui-même en remontée droite par T_3.

Pour éviter d'utiliser deux piles, on introduit une marque N qui vaut 1 lorsque le nœud est rencontré en descente et en remontée gauche, et 2 lorsqu'il est rencontré en remontée droite. On utilise dans l'algorithme le type Pile et les opérations *empiler*, *dépiler* et *est-vide* définies sur ce type au chapitre 5. La pile est formée d'éléments qui sont des couples (Arbre, marque). Le parcours itératif d'arbres binaires se décrit comme suit.

```
procedure Parcoursiter (A : Arbre);
var P : Pile; N : integer;
{N est une marque, positionnée au moment de la descente, qui indique au moment de la remontée si on remonte du sous-arbre gauche (N = 1) ou droit (N = 2)}
begin
    N = 1; P :=pile-vide;
    repeat
        if N= 1 then begin
            while A <> arbre-vide do begin
              TRAITEMENT1;
              P := empiler(P, (A, 1));
              A := g(A) {descente gauche}
              end;
            TERMINAISON {traitement de l'arbre vide}
        end;
        if not est-vide(P) then begin
              (A, N) := sommet(P); P := dépiler(P);
              if N = 1 then begin
              TRAITEMENT2; {remontée de la gauche}
              P := empiler(P, (A, 2));
              A := d(A) {descente droite}
              end
              else TRAITEMENT3 {remontée de la droite}
        end
    until est-vide(P)
end Parcoursiter;
```

Remarque : La méthode qui vient d'être présentée pour transformer le parcours récursif d'arbres binaires en parcours itératif peut être adaptée pour dérécursifier beaucoup d'autres algorithmes comportant deux appels récursifs; on en verra des exemples par la suite.

2. Arbres planaires généraux

On présente maintenant une structure arborescente plus large appelée ***arbre planaire général*** ou plus brièvement *arbre général* ou *arbre*, où le nombre de fils de chaque nœud n'est plus limité à deux. La figure 21 montre un exemple d'arbre planaire général.

2.1. Définitions et propriétés

Un ***arbre*** $A =< o,\ A_1, \ldots,\ A_p >$ est la donnée d'une racine et d'une liste finie, éventuellement vide (si $p = 0$), d'arbres disjoints. Une liste finie éventuellement

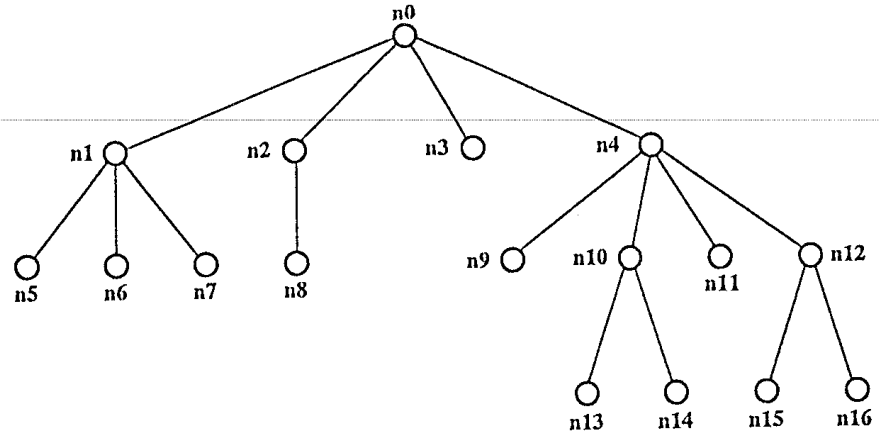


Figure 21. Arbre planaire général.

vide d'arbres disjoints est appelée une ***forêt***. Un arbre est donc obtenu en ajoutant un nœud racine à une forêt.

Ces définitions peuvent être exprimées par les équations :

$$\mathbb{A} = < o,\ \mathbb{F} >$$
$$\text{avec } \mathbb{F} = \emptyset + \mathbb{A} + < \mathbb{A},\ \mathbb{A} > + < \mathbb{A},\ \mathbb{A},\ \mathbb{A} > + \ldots$$

où $\mathbb{A}$ désigne l'ensemble des arbres planaires généraux, $\mathbb{F}$ l'ensemble des forêts et $\emptyset$ la forêt ne contenant aucun arbre.

La signature correspondant à ces définitions est la suivante :

sorte Arbregen, Forêt
utilise Nœud, Entier
opérations

cons	:	Nœud × Forêt → Arbregen
racine	:	Arbregen → Nœud
list-arbres	:	Arbregen → Forêt
forêt-vide	:	→ Forêt
ième	:	Forêt × Entier → Arbregen
nb-arbres	:	Forêt → Entier
insérer	:	Forêt × Entier × Arbregen → Forêt

Cette signature peut être complétée par d'autres opérations, par exemple *supprimer* un arbre d'une forêt, ou donner le *contenu* d'un nœud. Dans un but de simplification on se limite ici aux opérations de construction et d'accès pour les arbres et les forêts.

Etant donné des variables, o de sorte Nœud, F de sorte Forêt, A de sorte Arbregen et i de sorte Entier, on a la précondition et les axiomes suivants :

précondition

insérer(F, i, A) **est-défini-ssi** $1 \leq i \leq 1+$ *nb-arbres*(F)
{*l'opération insérer permet d'ajouter un arbre à une forêt*}

axiomes

racine(*cons*(o, F)) $= o$
list-arbres(*cons*(o, F)) $= F$
nb-arbres(*forêt-vide*) $= 0$
$1 \leq i \leq 1+$ *nb-arbres*(F) $\Rightarrow$
nb-arbres(*insérer*(F, i, A)) $=$ *nb-arbres*(F) $+ 1$
$1 \leq k < i \Rightarrow$ *ième*(*insérer*(F, i, A), k) $=$ *ième*(F, k)
$k = i \Rightarrow$ *ième*(*insérer*(F, i, A), k) $= A$
$i + 1 \leq k \leq 1 +$ *nb-arbres*(F) $\Rightarrow$ *ième*(*insérer*(F, i, A), k) $=$ *ième*($F, k-1$)

Remarque : Il est important de remarquer qu'il n'y a pas de notion gauche-droite dans les arbres : alors que $< o, < o, \emptyset, \emptyset >, \emptyset >$ et $< o, \emptyset, < o, \emptyset, \emptyset >>$ sont deux arbres binaires différents, il n'y a qu'un arbre planaire général avec deux nœuds l'arbre $< o, < o, \emptyset >>$. Une autre différence importante avec les arbres binaires, est qu'un arbre n'est jamais vide !

Le vocabulaire concernant les arbres est le même que celui présenté précédemment pour les arbres binaires à une différence près : il n'y a plus de notion gauche-droite pour les fils et pour les sous-arbres. Les fils (et les sous-arbres) d'un nœud sont ordonnés et l'on parle du premier fils, du deuxième fils, du troisième fils... et de même pour les sous-arbres. Par exemple sur la figure 21, n_9 est le premier fils de n_4, et n_{12} est le quatrième fils de n_4, n_8 est le premier fils (et fils unique) de n_2, n_{13} est le premier fils de n_{10} et n_{14} son deuxième fils.

2.2. Parcours des arbres généraux

Par extension du parcours des arbres binaires vu au paragraphe précédent, on peut définir un parcours en profondeur à main gauche des arbres généraux; dans ce parcours chaque nœud de l'arbre est rencontré une fois de plus que son nombre de fils, et l'on peut faire correspondre à chacun de ces passages un certain traitement du nœud rencontré : on note TRAITEMENT(i) (en abrégé T_i) la suite d'actions exécutées lors du $(i+1)^{ième}$ passage sur le nœud (figure 22); le premier traitement sur un nœud est noté TRAITEMENTPREF et le dernier traitement TRAITEMENTSUFF; on note TERM le traitement particulier des nœuds qui n'ont pas de fils.

La procédure *Parc* est une version récursive du parcours d'un arbre général A. Sa complexité, comptée en nombre de traitements de nœuds, est en $\Theta(n)$, si n est le nombre de nœuds de l'arbre A.

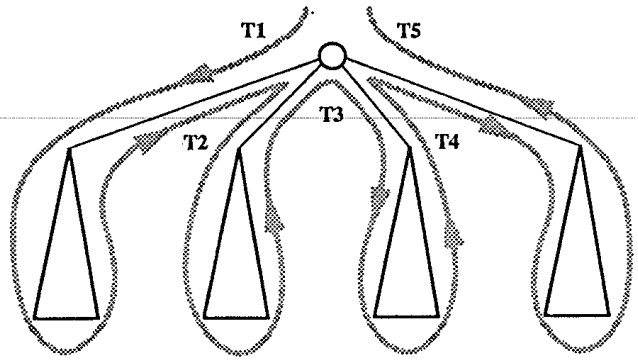


Figure 22. Parcours d'un arbre général.

L'opération *feuille*(A) teste si un arbre A est réduit à un seul nœud, c'est-à-dire si *list-arbres*(A) = *forêt-vide.*

```
procedure Parc(A : Arbregen);
var i, nb : integer;
begin
    nb := nb-arbres(list-arbres(A));
    if feuille(A) then TERM
    else begin
        TRAITEMENTPREF; {traitement avant de voir les fils}
        for i := 1 to nb - 1 do begin
            Parc(ième(list-arbres(A), i));
            TRAITEMENT(i)
        end;
        Parc(ième(list-arbres(A), nb));
        TRAITEMENTSUFF {traitement après avoir vu tous les fils}
    end
end Parc;
```

Deux ordres naturels d'exploration des arbres généraux sont inclus dans ce parcours.

1) L'***ordre préfixe***, où chaque nœud n'est pris en compte que lors du premier passage : cela revient à ne faire intervenir que TRAITEMENTPREF et TERM dans l'algorithme de parcours.

2) L'***ordre suffixe***, où chaque nœud n'est pris en compte que lors du dernier passage : dans l'algorithme de parcours cela revient à ne faire intervenir que TRAITEMENTSUFF et TERM .

Exemple : La figure 21 est la représentation graphique d'un arbre; on peut aussi représenter cet arbre linéairement à l'aide de parenthèses :

(a) $(n_o(n_1(n_5)(n_6)(n_7))(n_2(n_8))(n_3)(n_4(n_9)(n_{10}(n_{13})(n_{14}))(n_{11})(n_{12}(n_{15})(n_{16}))))$

Cette façon de noter doit être familière aux utilisateurs du langage LISP; ceux qui ont surtout l'habitude d'écrire les fonctions dans la notation mathématique usuelle préfèrent peut-être la représentation linéaire de l'arbre à l'aide de parenthèses et de virgules :

(b) $n_o(n_1(n_5,\ n_6,\ n_7),\ n_2(n_8),\ n_3,\ n_4(n_9,\ n_{10}(n_{13},\ n_{14}),\ n_{11},\ n_{12}(n_{15},\ n_{16})))$

Pour obtenir une représentation des arbres généraux sous la forme (a) il suffit que les traitements de la procédure de parcours *Parc* soient les suivants :

- TRAITEMENTPREF écrit une parenthèse ouvrante suivie du contenu de la racine de A;
- TRAITEMENTSUFF écrit une parenthèse fermante;
- le traitement, réservé aux feuilles, TERM écrit une parenthèse ouvrante, puis le nœud, puis une parenthèse fermante;
- les autres traitements ne font rien.

Pour obtenir une représentation sous la forme (b) il suffit que dans la procédure de parcours *Parc* on ait les traitements suivants :

- TRAITEMENTPREF écrit le contenu de la racine de A et une parenthèse ouvrante;
- les TRAITEMENT(i) écrivent une virgule;
- TRAITEMENTSUFF écrit une parenthèse fermante;
- le traitement réservé aux feuilles, TERM, écrit le contenu du nœud.

2.3. Représentation des arbres généraux

2.3.1. Représentations simples

Différentes façons d'implémenter un arbre sont envisageables.

On peut donner pour chaque nœud la liste de ses fils comme sur la figure 23, qui représente l'arbre de la figure 21. Cette représentation des arbres peut être utile, mais elle se prête mal à une gestion dynamique (cf. exercices).

On peut aussi décrire une représentation analogue à celle des arbres binaires : chaque nœud contient un pointeur vers chacun des sous-arbres (cf. figure 24), et éventuellement un champ pour stocker l'élément contenu dans le nœud. Sauf pour des cas particuliers d'arbres dont tous les nœuds ont à peu près le même nombre de fils, cette représentation est trop gourmande en place (c'est le nombre maximal de

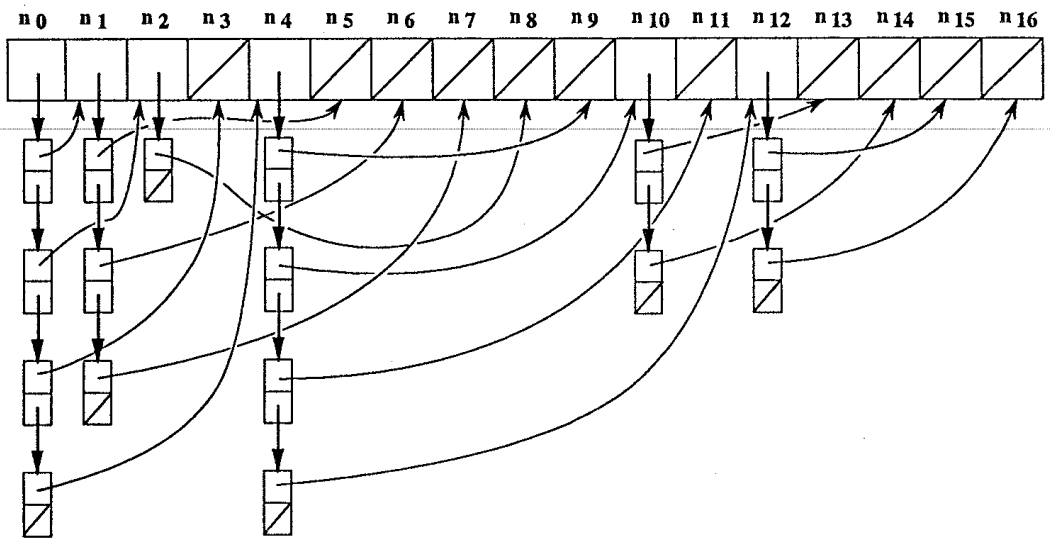


Figure 23. Représentation d'un arbre par liste de fils.

fils d'un nœud qui détermine le nombre de pointeurs dans tous les nœuds), et donc peu utilisable.

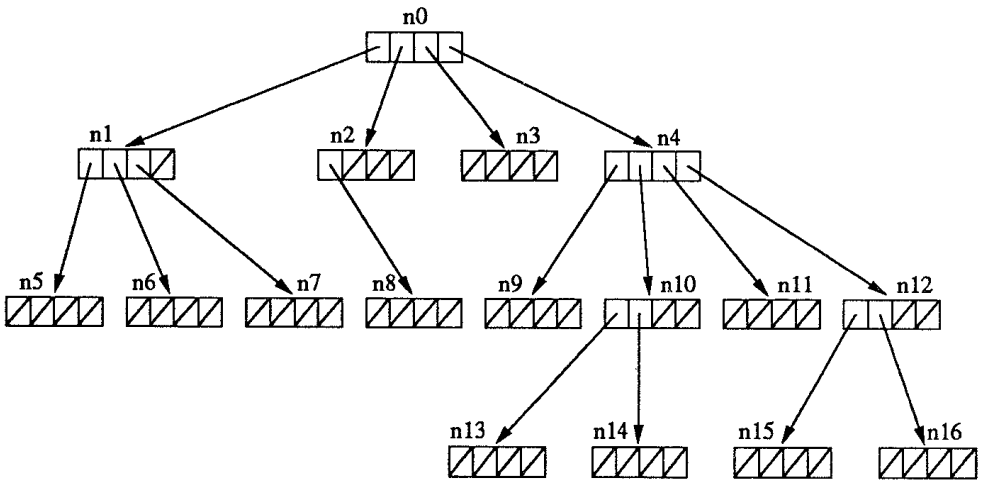


Figure 24. Représentation d'un arbre par des n-uplets de pointeurs.

La suite de ce paragraphe est consacrée à une représentation, très utilisée, des arbres généraux sous forme d'arbres binaires.

2.3.2. Représentation sous forme d'arbres binaires

On établit ici une correspondance entre les arbres planaires généraux et les arbres binaires. Rappelons qu'un arbre général est obtenu en ajoutant un nœud racine à

une forêt. De même qu'on a défini la taille d'un arbre binaire, on définit la ***taille d'un arbre général*** comme le nombre total de ses nœuds; la ***taille d'une forêt*** est le nombre total de nœuds de tous les arbres qui la composent. On a donc :

taille(forêt-vide) = 0
taille(insérer(F, i, A)) = *taille(F)* + *taille(A)*
taille(cons(o, F)) = 1 + *taille(F)*

Lemme 7 : Il existe une bijection entre les arbres généraux ayant $n + 1$ nœuds, et les forêts de taille n.

La *preuve* est évidente : on associe à l'arbre $A =< o, A_1, \ldots, A_p >$ la forêt $F =< A_1, \ldots, A_p >$; l'arbre A a un nœud de plus (sa racine) que la forêt. La figure 25 montre la forêt associée à l'arbre de la figure 21. □

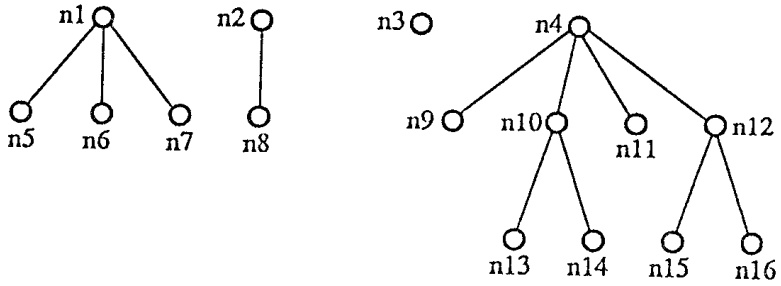


Figure 25. Forêt associée à l'arbre de la figure 21.

Montrons maintenant que l'on peut coder les forêts par des arbres binaires. La figure 26 montre l'arbre binaire associé à la forêt de la figure 25 dans la bijection dite ***bijection fils aîné-frère droit.***

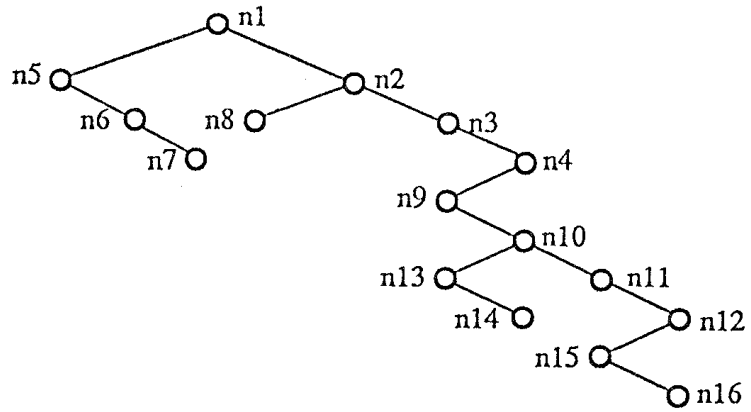


Figure 26. Arbre binaire associé à la forêt de la figure 25.

Proposition 3 : Il existe une bijection entre les forêts de taille n et les arbres binaires ayant n nœuds.

Preuve : Pour obtenir l'arbre binaire associé à une forêt, on construit pour chaque nœud un lien gauche vers son *premier fils* (fils aîné), et un lien droit vers son *frère* situé immédiatement à sa droite dans la forêt (on considère que les racines des différents arbres de la forêt sont des nœuds frères), comme le montre le schéma de la figure 27.

Inversement, pour revenir de l'arbre binaire vers la forêt qu'il représente, on effectue la transformation inverse de celle de la figure 27 : conserver les liens gauches; supprimer les liens droits, et construire des liens entre un nœud et chacun des nœuds du bord droit de son ancien sous-arbre gauche. □

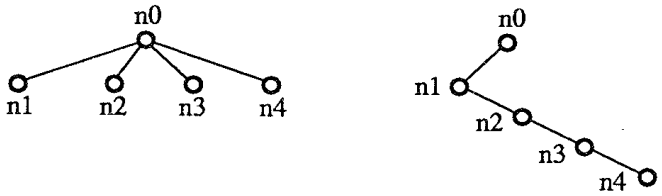


Figure 27. Bijection fils aîné-frère droit.

Les deux résultats précédents permettent maintenant :
- d'une part de dénombrer les arbres généraux (respectivement les forêts) selon leur taille,
- d'autre part de donner une bonne représentation des arbres (respectivement des forêts) en termes d'arbres binaires.

Proposition 4 : Le nombre a_{n+1} d'arbres de taille $n+1$ est :

$$a_{n+1} = \frac{1}{n+1}\binom{2n}{n}$$

La *preuve* est une conséquence immédiate du lemme 7 et des propositions 2 et 3 : il y a autant de forêts de taille n, que d'arbres binaires ayant n nœuds, et il y a autant d'arbres ayant $(n+1)$ nœuds que de forêts de taille n. □

La figure 28 montre les 5 arbres que l'on peut construire avec 4 nœuds.

Proposition 5 : Dans la bijection fils aîné-frère droit entre une forêt F et un arbre binaire B,
- le bord gauche de B est le même que le bord gauche du premier arbre de F,
- le bord droit de B représente la suite des racines des arbres de F,

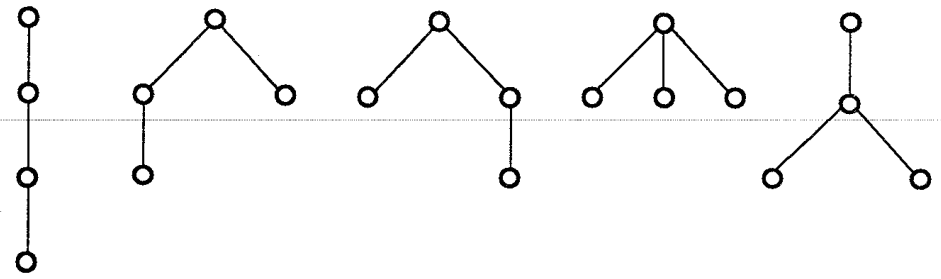


Figure 28. Les 5 arbres avec 4 nœuds.

- le parcours préfixe de B donne les nœuds dans le même ordre que le parcours préfixe de F,
- le parcours symétrique de B donne les nœuds dans le même ordre que le parcours suffixe de F.

Cette propriété, dont la preuve est laissée en exercice, montre que la représentation des arbres et des forêts par des arbres binaires est utilisable en pratique : les opérations sur les arbres ne sont en général pas difficiles à décrire sur les arbres binaires leur correspondant.

Exercices

1. Montrer que dans un arbre binaire, le nombre maximal de nœuds à profondeur k est 2^k.

2. Montrer que pour tout arbre binaire ayant f feuilles et s nœuds qui ne sont pas des feuilles, on a : $f \leq s + 1$.

3. On a vu qu'on peut représenter un nœud dans un arbre binaire par son occurrence dans l'arbre, qui est un mot de $\{0, 1\}^*$. Exprimer dans cette représentation :

a) le bord gauche et le bord droit de l'arbre;

b) le père et le fils d'un nœud;

c) la hauteur d'un nœud, la hauteur de l'arbre et la longueur de cheminement.

4. La numérotation en ordre hiérarchique d'un arbre binaire consiste à numéroter en ordre croissant à partir de 1, tous les nœuds possibles d'un arbre binaire, à partir de la racine, niveau par niveau, et de gauche à droite sur chaque niveau.

a) Montrer que si un nœud est numéroté par i dans la numérotation en ordre hiérarchique, alors son fils gauche est numéroté par $2i$, et son fils droit par $2i + 1$.

b) Ecrire une procédure qui produit la liste des nœuds d'un arbre binaire dans l'ordre hiérarchique.

c) Montrer que si un nœud a pour occurrence μ (cf. exercice précédent) et pour numérotation en ordre hiérarchique l'entier i, alors on a la relation $i = 2^{\lfloor \log_2 i \rfloor} + m$, où m est l'entier représenté par μ.

5. Montrer que pour tout arbre binaire localement complet B ayant n nœuds internes, on a la relation $LCE(B) = LCI(B) + 2n$.

6. Montrer que les arbres binaires qui minimisent la longueur de cheminement (resp. longueur de cheminement interne et longueur de cheminement externe) sont ceux qui ont toutes leurs feuilles situées sur deux niveaux au plus.

7. Soit B un arbre binaire de taille n. Le lemme 3 montre l'encadrement suivant pour la longueur de cheminement totale $LC(B)$:

$$\sum_{i=1}^{n} \lfloor \log_2 i \rfloor \leq LC(B) \leq \frac{n(n-1)}{2}$$

Etablir l'égalité : $\sum_{i=1}^{n} \lfloor \log_2 i \rfloor = 2 + (n+1)\lfloor \log_2 n \rfloor - 2^{\lfloor \log_2 n \rfloor + 1}$. En déduire que la longueur de cheminement est au minimum de l'ordre de $n \log_2 n$ et au maximum de l'ordre de n^2.

8. Ecrire pour chaque représentation d'un arbre binaire (chaînée ou contiguë)

a) une fonction qui calcule la hauteur d'un arbre binaire;

b) une fonction qui calcule la longueur de cheminement d'un arbre binaire; (on ne supposera pas connue à l'avance la taille de l'arbre);

c) une fonction qui indique pour chaque nœud de l'arbre le nombre de ses descendants.

9. a) Ecrire une fonction qui teste l'égalité de deux arbres binaires étiquetés.

b) Un arbre binaire S figure dans un arbre binaire T s'il lui est égal ou s'il est égal à un sous-arbre de T. Ecrire une fonction récursive *figure*(S, T) qui indique si S figure dans T; si c'est le cas, la fonction renvoie l'adresse du sous-arbre de T qui est égal à S et sinon renvoie *nil*.

10. Représenter un arbre binaire localement complet dans un tableau avec seulement l'adresse du fils gauche, le fils droit étant placé dans une case contiguë.

11. a) Est-il vrai que les feuilles d'un arbre binaire occurrent aux mêmes positions relatives dans les trois parcours préfixe, symétrique et suffixe?

b) Pour chacun des couples de parcours suivants trouver tous les arbres binaires dont les nœuds apparaissent exactement dans le même ordre pour les deux parcours du couple :

i) préfixe et symétrique,
ii) préfixe et suffixe,
iii) symétrique et suffixe.

12. En utilisant la représentation d'un nœud dans un arbre binaire par son occurrence (mot de $\{0,1\}^*$), caractériser chacun des trois parcours préfixe, symétrique et suffixe.

13. Montrer que si on connaît la liste des nœuds d'un arbre binaire en ordre préfixe et la liste en ordre symétrique, alors on peut reconstruire la structure de l'arbre binaire. Ce résultat est-il encore valable si on connaît seulement l'ordre préfixe et l'ordre suffixe? Ou seulement l'ordre symétrique et l'ordre suffixe?

14. Déterminer les couples d'ordres (X, Y), où X et Y sont pris parmi {préfixe, suffixe, symétrique} pour lesquels l'assertion suivante est vraie : on peut tester si un nœud m est un ascendant d'un nœud n en testant si m précède n dans l'ordre X et si m suit n dans l'ordre Y.

15. Ecrire une procédure qui construit un arbre binaire de profondeur minimum à partir de la suite de ses nœuds en ordre préfixe.

16. Donner tous les arbres binaires dont la liste des nœuds en ordre symétrique coïncide avec l'expression arithmétique : $2 + 3 * 4 + 5$.

17. Supposons que la liste des nœuds d'un arbre binaire dans l'ordre préfixe soit $n_1, n_2 \ldots, n_p$ et dans l'ordre symétrique $n_{k_1}, n_{k_2} \ldots, n_{k_p}$.

a) Montrer que la permutation $k_1, k_2, \ldots, k_p$ de $1, 2 \ldots, p$ peut être obtenue en passant $1, 2, \ldots, p$ dans une pile (cf. exercices sur les listes).

b) Inversement, montrer qu'à toute permutation $k_1, k_2 \ldots, k_p$ qui peut être engendrée par une pile on peut associer un arbre binaire tel que la liste de ses nœuds dans l'ordre préfixe soit $n_1, n_2 \ldots, n_p$ et dans l'ordre symétrique $n_{k_1}, n_{k_2} \ldots, n_{k_p}$.

18. On considère un arbre dont les nœuds internes représentent des fonctions d'arité 2, et les feuilles des constantes ou des variables. On veut obtenir l'expression arithmétique correspondante sous l'une des trois formes : préfixe, symétrique ou suffixe.

a) Donner une version de la procédure *Parcours* pour chaque forme préfixe ou suffixe.

b) Pour la forme symétrique, comment faut-il modifier la procédure *Parcours* pour obtenir une notation avec parenthèses et virgules $f(a, b)$?

19. Dérécursifier la procédure *Sym* en s'inspirant de la procédure *Parcoursiter.*

20. Ecrire dans chacun des cas suivants une procédure récursive qui construit un arbre binaire à partir de la lecture linéaire d'une expression arithmétique bien parenthésée :

a) l'expression est donnée en ordre préfixe;

b) l'expression est donnée en ordre symétrique;

c) l'expression est donnée en ordre suffixe.

21. Montrer que la complexité en temps de la procédure *Parcours* est linéaire.

22. Montrer que dans la procédure *Parcoursiter*, la hauteur de pile nécessaire est égale à la hauteur de l'arbre.

23. On définit récursivement une fonction entière ϕ sur l'ensemble des arbres binaires. Appelons $g(A)$ et $d(A)$ respectivement le sous-arbre gauche et le sous-arbre droit de A :

$$\phi(A) = \begin{cases} 0 & \text{si } A \text{ est l'arbre vide} \\ \max(\phi(g(A)),\ \phi(d(A))) & \text{si } \phi(g(A)) \neq \phi(d(A)) \\ \phi(d(A)) + 1 & \text{si } \phi(g(A)) = \phi(d(A)) \end{cases}$$

Cette fonction ϕ permet d'associer à tout arbre binaire un ***nombre*** dit ***de Strahler.***

a) Quel est le nombre de Strahler d'un arbre binaire complet de hauteur n ? Le nombre de Strahler d'un arbre binaire dégénéré de hauteur n ?

b) Donner une procédure récursive basée sur le parcours suffixe qui calcule le nombre de Strahler d'un arbre binaire.

24. Définir la structure de données qui permet de représenter l'arbre planaire de la figure 23 (représentation par listes de fils).

25. On a montré comment la procédure *Parc* de parcours à main gauche d'un arbre planaire général permet d'obtenir deux représentations linéaires avec parenthèses d'un arbre planaire général. Montrer comment on peut passer de l'une de ces représentations à l'autre.

26. a) Pour chacune des deux représentations linéaires d'un arbre général (cf. exercice précédent), écrire une procédure récursive qui à tout arbre associe sa représentation sous forme d'arbre binaire.

b) Ecrire aussi les procédures qui réalisent les transformations inverses.

27. Montrer que le nombre total n de nœuds qui ne sont pas des feuilles dans une forêt s'exprime simplement en fonction du nombre p de liens droits à *nil* dans l'arbre binaire obtenu par la bijection fils aîné-frère droit.

28. Une forêt peut être considérée comme induisant un ordre partiel «précède» tel que tout nœud d'un arbre précède tous ses descendants dans l'arbre. Une liste des nœuds de la forêt est dite en ***ordre topologique*** si l'ordre séquentiel de la liste respecte l'ordre partiel «précède». Les nœuds de la forêt sont-ils triés en ordre topologique lorsqu'ils sont listés a) en ordre préfixe b) en ordre suffixe?

29. Démontrer la proposition 5 de la représentation par arbre binaires des arbres planaires.

30. La bijection fils aîné-frère droit n'est pas «symétrique». En inversant droite et gauche, mettre en évidence une autre bijection entre les arbres binaires et les forêts. Décrire des propriétés analogues à celles qui sont énoncées dans la proposition 5.

31. Les ***arbres binômiaux*** sont définis, pour $k \geq 0$, par les règles suivantes :

$$B_0 = o, \text{ et } B_{k+1} = \mathit{cons}(o,\ \mathit{insérer}(\mathit{list\text{-}arbres}(B_k), 1, B_k))$$

L'arbre B_k est donc formé de 2^k nœuds; on dit qu'il est de *taille* 2^k. Une ***file binômiale*** F_n est une suite d'arbres binômiaux étiquetés de tailles strictement décroissantes, dont le nombre total de nœuds vaut n; il y a donc dans F_n exactement un arbre binômial étiqueté B_i pour chaque $b_i \neq 0$ dans l'écriture binaire de $n = \sum_{i \geq 0} b_i 2^i$. Par exemple, $13 = 8+4+1$, et l'on a la file binômiale F_{13} (figure 29).

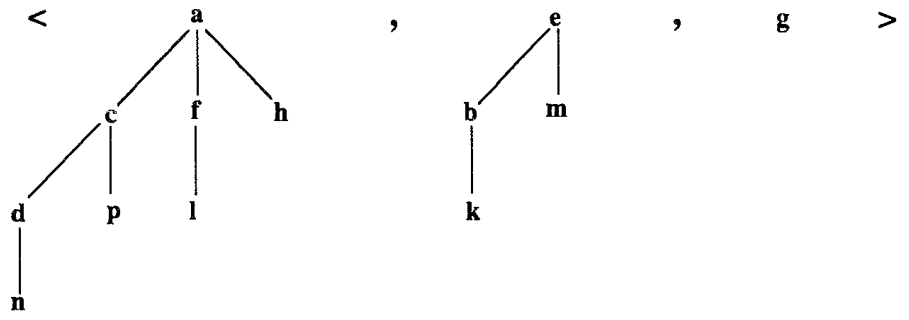


Figure 29. File binômiale.

Définissons l'algorithme de concaténation de deux files binômiales.

• La concaténation de deux arbres binômiaux B_k et B'_k est l'arbre binômial B_{k+1}, construit comme dans la définition des arbres binômiaux.

• La concaténation de deux files binômiales F_n et F_m peut alors se décrire par analogie avec l'algorithme classique d'addition des représentations binaires de n et m : la concaténation de F_n et F_m se fait en traitant par tailles croissantes les arbres binômiaux composant chacune des files, et le résultat de la concaténation de deux arbres binômiaux de même taille constitue la «retenue» pour l'étape suivante. Initialement il y a au plus deux B_0 présents, un pour chaque file; si les deux sont présents, ils sont concaténés pour constituer une retenue B_1. A l'étape k, pour $k \geq 1$, il y a au plus trois B_k présents : un pour chaque file et la retenue. S'il n'y a qu'un seul B_k présent, il constitue la $k^{ième}$ composante du résultat; s'il y a deux B_k ils sont concaténés pour former la retenue B_{k+1} de l'étape suivante; s'il y en a trois, deux d'entre eux sont concaténés pour former la retenue suivante, et le troisième constitue la composante B_k du résultat.

La procédure se termine quand l'une des files est épuisée et qu'il n'y a plus de propagation de retenue.

Le but de l'exercice est de programmer cet algorithme en utilisant la représentation fils aîné-frère droit des arbres et des forêts.

Lectures conseillées pour le chapitre 7

Knuth, *The Art of Computer Programming*, Vol.1, *Fundamental Algorithms*, Addison-Wesley, 1973.

Chapitre 8
Graphes

1. Définitions et exemples

Beaucoup de problèmes de la vie courante, tels la gestion de réseaux de communication ou l'ordonnancement de tâches, correspondent à des structures relationnelles que l'on peut modéliser par des graphes. Informellement un graphe est un ensemble d'objets, appelés *sommets*, et de *relations* entre ces sommets.

Exemple 1 : Dans une carte des liaisons aériennes, les villes sont des sommets du graphe et l'existence d'une liaison aérienne entre deux villes est la relation du graphe (figure 1).

Dans cet exemple, la relation est symétrique : on peut raisonnablement supposer que la compagnie aérienne assure des vols aller-retour. Dans ce cas on dit que le graphe est *non orienté*. Si deux sommets $s1$ et $s2$ sont en relation, on dit qu'il existe une *arête* entre $s1$ et $s2$. On peut se poser des questions du type suivant : existe-t-il un moyen d'aller d'une ville A à une ville B ? Quel est le trajet qui nécessite le moins d'escales ? etc.

Exemple 2 : Dans le graphe du flôt de contrôle d'un programme, les boîtes (instructions ou tests) sont les sommets, et les flèches indiquent les enchaînements possibles entre celles-ci.

Dans cet exemple, la relation n'est pas symétrique : $I := I + 1$ s'exécute immédiatement avant $S := S + A[I]$, mais pas l'inverse. Dans ce cas, on dit qu'on a un graphe *orienté*. Si deux sommets $s1$ et $s2$ sont en relation, on dit qu'on a un *arc* de $s1$ vers $s2$.

Exemple 3 : Dans une organisation du travail où certaines tâches doivent être exécutées avant d'autres, on peut schématiser l'ordonnancement des tâches par un graphe où les sommets sont les tâches et où il existe un arc entre deux tâches t_i

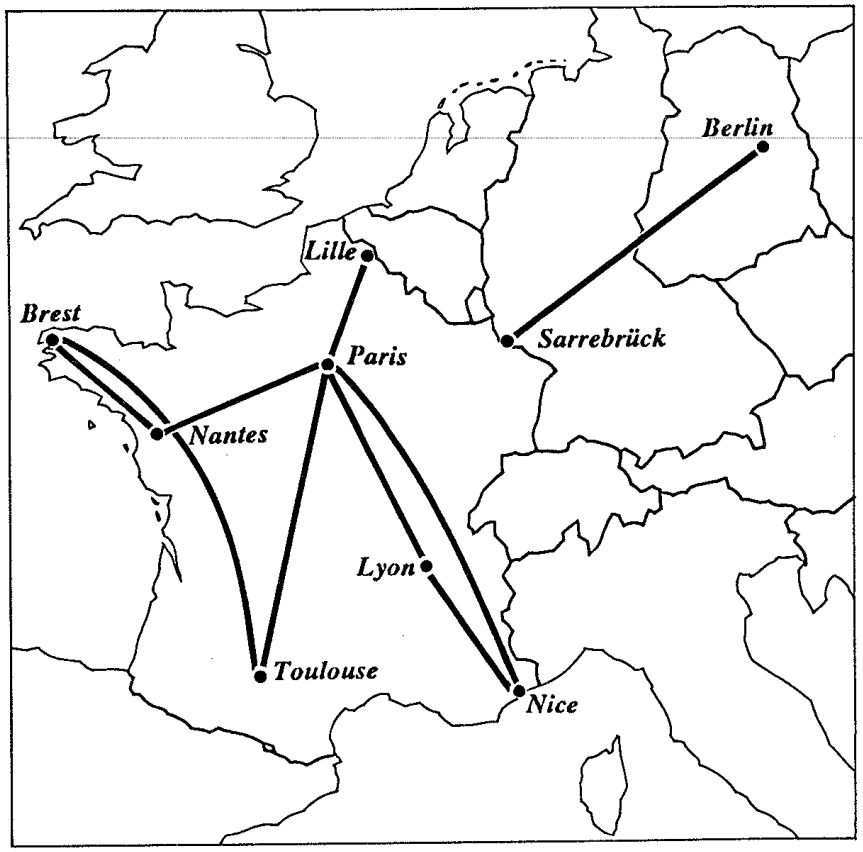


Figure 1. Graphe de liaisons aériennes.

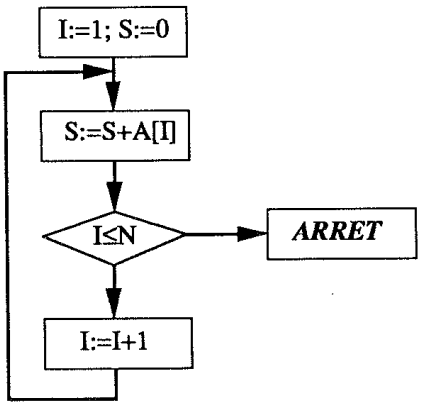


Figure 2. Graphe du flôt de contrôle d'un programme.

et t_j seulement si t_i doit être terminée juste avant d'exécuter t_j. Le graphe de la figure 3 décrit la préparation d'un curry d'agneau.

Un des traitements intéressants sur un tel graphe est un *tri topologique* qui consiste à trouver un ordre des tâches tel que toute tâche t_i soit exécutée avant toute tâche t_j s'il existe un arc de t_i à t_j.

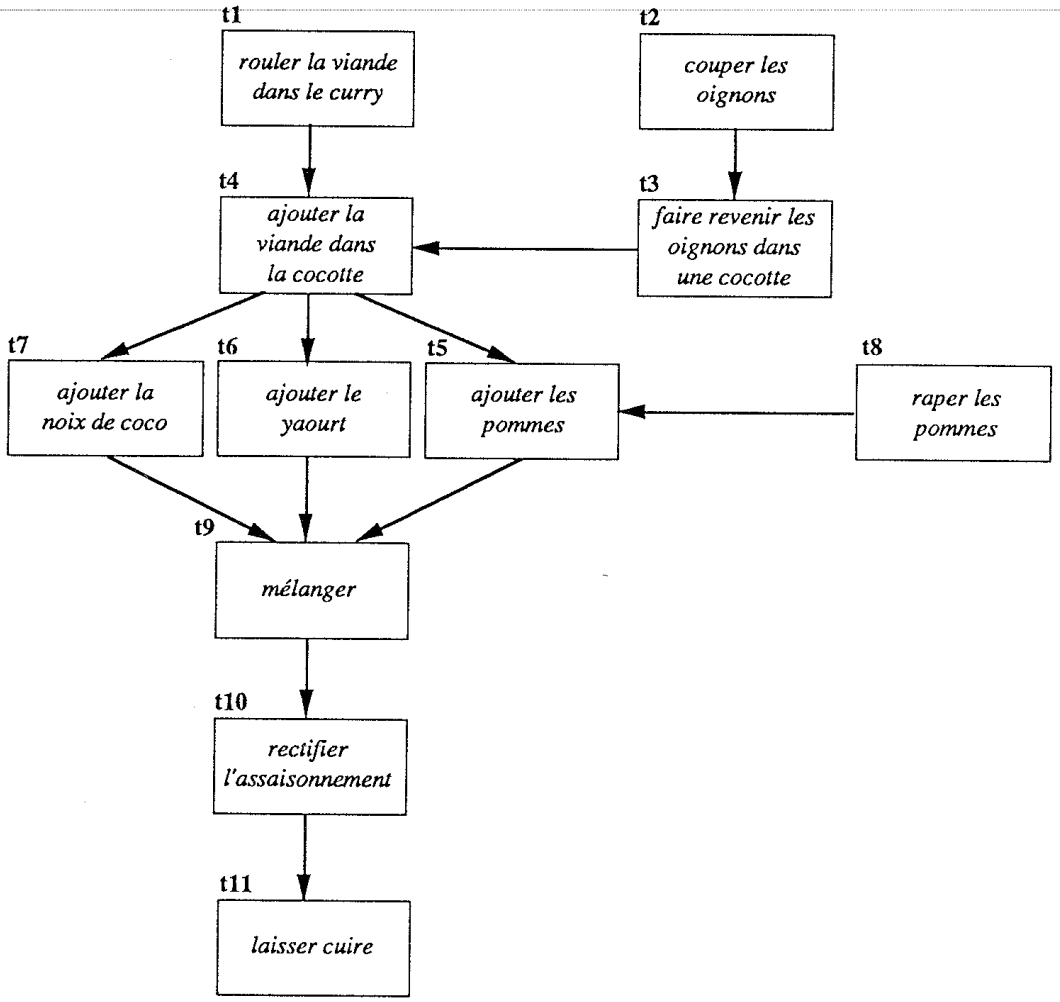


Figure 3. Exemple de graphe d'ordonnancement de tâches.

Définitions :

• Un ***graphe orienté*** G est un couple $< S, A >$, où S est un ensemble fini de sommets et où A est un ensemble fini de *paires ordonnées* de sommets, appelées ***arcs***.

• Un ***graphe non orienté*** G est un couple $< S, A >$, où S est un ensemble fini de *sommets* et où A est un ensemble fini de *paires* de sommets, appelées ***arêtes***.

• Dans de nombreux problèmes, il est naturel d'associer une valeur (on dit aussi un coût ou un poids) aux arcs ou aux arêtes du graphe. Un ***graphe valué***, orienté (resp. non orienté) est un triplet $< S, A, C >$ où S est un ensemble fini de sommets, A un ensemble fini d'arcs (resp. arêtes) et C une fonction de A dans $\mathbb{R}$ appelée fonction de coût.

On peut alors traiter des problèmes tels que : trouver le moyen le plus économique d'aller de Brest à Lyon, connaissant pour chaque ligne aérienne, le prix du billet, ce qui revient à *rechercher un plus court chemin entre deux sommets du graphe.*

Remarque 1 : La distinction entre graphes orientés et graphes non orientés n'est pas aussi catégorique qu'on pourrait le croire : quand on a un graphe orienté, il est parfois commode de ne pas tenir compte de l'orientation si le problème posé est de nature non orientée. Dans ce cas, on se contente de considérer, provisoirement, que la relation est symétrique.

Remarque 2 : Certains graphes admettent des boucles, c'est-à-dire, des arcs (ou des arêtes) qui connectent un sommet avec lui-même. On peut aussi trouver dans un graphe, deux sommets reliés par plusieurs arcs (ou arêtes) correspondant à des relations différentes (cas de *multigraphes*). On parle dans ce cas d'*arcs* (ou d'*arêtes*) *multiples*. Dans ce qui suit, on suppose que les graphes n'ont ni boucles ni arcs (ni arêtes) multiples (*graphes simples*).

2. Terminologie

Nous définissons ici le vocabulaire et les notations usuels sur les graphes.

• On note $x \rightarrow y$ l'arc (x, y) ; x est l'***extrémité initiale*** de l'arc, y est son ***extrémité terminale***. On dit que y est un ***successeur*** de x et que x est un ***prédécesseur*** de y.

De même, on note x—y l'arête $\{x, y\}$; x et y sont les deux ***extrémités*** de l'arête. Par abus de langage, on dit parfois que y est ***successeur*** de x ou que x est successeur de y.

• Soit $G = < S, A >$ un graphe. Le ***sous-graphe*** *de* G ***engendré par*** $S' \subseteq S$ est le graphe G' dont les sommets sont les éléments de S' et dont les arcs (resp. arêtes) sont les arcs (resp. arêtes) de G ayant leurs deux extrémités dans S' (cf. figure 4b). Autrement dit, on ignore les sommets de $S - S'$ ainsi que les arcs ayant au moins une extrémité dans $S - S'$.

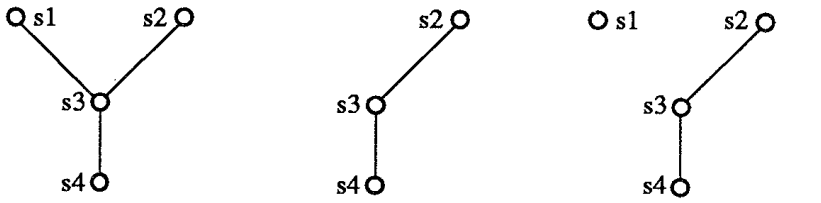
• Soit $G = < S, A >$ un graphe. Le ***graphe partiel*** *de* G ***engendré par*** $A' \subseteq A$ est le graphe $< S, A' >$ dont les sommets sont les éléments de S et dont les arcs (resp. arêtes) sont ceux de A'. Autrement dit, on élimine de G les arcs (arêtes) de $A - A'$ (cf. figure 4c).

• Deux ***arcs*** (resp. *arêtes*) d'un graphe orienté (resp. non orienté) sont dits *adjacents* s'ils ont au moins une extrémité commune.

Deux ***sommets*** d'un graphe non orienté sont dits ***adjacents*** s'il existe une arête les joignant.

Dans un graphe orienté, le *sommet* y est dit *adjacent* au sommet x s'il existe un arc $x \rightarrow y$.

• Un graphe orienté (resp. non orienté) est dit ***complet*** si pour tout couple de sommets (x, y), il existe un arc $x \rightarrow y$ (resp. une arête x—y).



a) G (non orienté) **b)** sous-graphe S'={s2, s3, s4} **c)** graphe partiel A'={(s2, s3), (s3, s4)}

Figure 4. Exemple de sous-graphe et de graphe partiel.

• Dans un graphe orienté, si un sommet x est l'extrémité initiale d'un arc $u = x \rightarrow y$, on dit que l'arc u est ***incident à*** x ***vers l'extérieur***. Le nombre d'arcs ayant leur extrémité initiale en x, se note $d^{\circ +}(x)$ et s'appelle le ***demi-degré extérieur*** de x.

On définit de même les notions d'arc ***incident vers l'intérieur*** et de ***demi-degré intérieur*** qui est noté $d^{\circ -}(x)$.

• Dans un graphe orienté (resp. non orienté), on appelle ***degré*** d'un sommet x, et on note $d^{\circ}(x)$, le nombre d'arcs (resp. d'arêtes) dont x est une extrémité. Dans le cas d'un graphe orienté, on a : $d^{\circ}(x) = d^{\circ +}(x) + d^{\circ -}(x)$, pour tout sommet x. Dans l'exemple de la figure 5, le calcul des degrés du sommet $x3$ donne : $d^{\circ +}(x3) = 2; d^{\circ -}(x3) = 3; d^{\circ}(x3) = 5$.

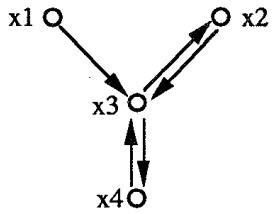


Figure 5. $d^{\circ +}(x3) = 2; d^{\circ -}(x3) = 3; d^{\circ}(x3) = 5$.

• Dans un graphe orienté G (resp. non orienté), on appelle ***chemin*** (resp. ***chaîne***) ***de longueur*** λ, une suite de $(\lambda+1)$ sommets $(s_0, s_1, ..., s_\lambda)$ tels que : pour tout i tel que $0 \leq i \leq \lambda - 1$, $s_i \rightarrow s_{i+1}$ est un arc (resp. une arête) de G. Par convention, on dit qu'il y a un chemin de longueur 0 de tout sommet vers lui-même.

On peut aussi définir de façon récursive un chemin de longueur λ ($\lambda > 0$) allant du sommet x vers le sommet y comme :

- si $\lambda = 1$, un arc de x vers y
- sinon la suite composée d'un arc de x vers un certain sommet z et d'un chemin de z vers y, de longueur $\lambda - 1$.

La définition récursive d'une chaîne de longueur λ est analogue.

• Un chemin (resp. une chaîne) est dit ***élémentaire*** s'il ne contient pas plusieurs fois le même sommet.

• Dans un graphe orienté (resp. non orienté), un chemin (resp. une chaîne) $(s_0, s_1, ..., s_\lambda)$ dont les λ arcs (resp. arêtes) sont tous distincts deux à deux et tel que les deux sommets aux extrémités du chemin (resp. de la chaîne) coïncident, est un ***circuit*** (resp. un ***cycle***).

• Un graphe orienté est dit ***fortement connexe*** si pour toute paire ordonnée de sommets distincts (u, v), il existe un chemin de u vers v et un chemin de v vers u.

Un graphe non orienté est dit ***connexe*** si pour toute paire de sommets distincts $\{u, v\}$, il existe une chaîne reliant u et v.

• On appelle ***composante fortement connexe*** d'un graphe orienté un sous-graphe fortement connexe maximal, c'est-à-dire un sous-graphe fortement connexe qui n'est pas strictement contenu dans un autre sous-graphe fortement connexe.

De même, dans un graphe non orienté, on appelle ***composante connexe*** un sous-graphe connexe maximal. Le graphe de la figure 1 a deux composantes connexes.

Arbres et arborescences en théorie des graphes

On définit en théorie des graphes des structures appelées *arbres* et *arborescences*. Ces structures sont proches de la structure de données qui a été appelée précédemment arbre planaire général. En théorie des graphes, on appelle ***arbre*** un *graphe non orienté, connexe, sans cycle.*

Un certain nombre de propriétés caractéristiques des arbres sont énoncées ci-dessous.

Proposition : Soit $G = < S, A >$ un graphe non orienté, ayant n sommets. Les propriétés suivantes sont équivalentes :

(1) G est un graphe connexe sans cycle,

(2) G est connexe et si on supprime une arête, il n'est plus connexe,
(3) G est connexe avec $(n-1)$ arêtes,
(4) G est sans cycle et en ajoutant une arête, on crée un cycle,
(5) G est sans cycle avec $(n-1)$ arêtes,
(6) tout couple de sommets de S est relié par une chaîne et une seule.

La preuve de cette proposition est laissée en exercice.

Le type de données arbre planaire général, défini au chapitre 7 correspond plutôt à la notion d'*arborescence* définie ci-dessous, car la relation père-fils est orientée.

Etant donné $G =< S, A >$ un graphe orienté, on appelle ***racine*** de G un sommet r de S tel que tout autre sommet de S puisse être atteint par un chemin d'origine r. Notons qu'une racine n'existe pas toujours : le graphe de la figure 6 n'a pas de racine.

Figure 6. Exemple de graphe qui n'est pas une arborescence.

On appelle ***arborescence*** un graphe orienté G admettant une racine, et tel que le graphe non orienté G', obtenu à partir de G en oubliant l'orientation des arcs, soit un arbre. Les propriétés caractéristiques des arborescences sont étudiées en exercice.

Rappelons que dans un arbre planaire général, les successeurs d'un nœud forment une suite ordonnée. Dans une arborescence ils forment, par contre, un ensemble et l'ordre dans lequel on accède aux éléments de cet ensemble n'est que pure convention. Par exemple les deux graphes de la figure 7 sont une même arborescence, mais si on les considère comme des arbres planaires généraux, ceux-ci sont différents.

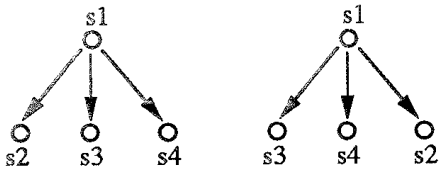


Figure 7. Deux arborescences identiques.

Inversement, si on considère un arbre planaire général comme un graphe, en faisant abstraction de l'ordre des fils de chaque nœud, on obtient une arborescence.

Dans le chapitre sur les structures arborescentes, on a donné la définition d'une forêt d'arbres planaires généraux : c'est une liste de tels arbres. On dit la même chose à propos des arbres et arborescences de la théorie des graphes, mais on ignore

l'ordre des arbres (ou arborescences) : un graphe qui est un ensemble fini d'arbres (ou d'arborescences) disjoints est appelé une ***forêt.***

3. Types abstraits Graphes

Sur les exemples proposés ci-dessus, le graphe est donné une fois pour toutes et les opérations intéressantes sont le test de l'existence d'un arc (d'une arête) entre deux sommets, ou le test de l'existence d'un sommet parmi les successeurs d'un autre sommet, etc. De plus, dans bien des cas, on a besoin d'énumérer les successeurs d'un sommet : il est donc utile de connaître le demi-degré extérieur de tout sommet et le $i^{ième}$ successeur d'un sommet *(l'ordre d'énumération étant arbitraire).*

Mais, le plus souvent, le graphe est évolutif et l'on veut pouvoir lui appliquer des opérations de base qui sont : ajout ou suppression d'un sommet, ajout ou suppression d'un arc, etc. On retrouve toutes ces opérations dans la signature des graphes.

On va avoir deux types abstraits : un pour les graphes orientés, et un pour les graphes non orientés. On commence par donner le type sommet, qui est utilisé par ces deux types abstraits.

Pour distinguer les sommets d'un graphe, on les étiquette, soit par des chaînes de caractères, soit par des numéros. C'est cette deuxième convention qui est suivie dans la spécification ci-dessous.

sorte Sommet
utilise Entier
opérations
som : Entier $\rightarrow$ Sommet
$n°$: Sommet $\rightarrow$ Entier

axiomes
$n°(som(i)) = i$, pour tout entier i

Dans la suite, quand il n'y a pas d'ambiguïté, on ne fait pas la distinction entre un sommet et son numéro : on considère que les sommets sont des entiers.

3.1. Spécification des graphes orientés

Un graphe orienté est un ensemble de sommets et un ensemble d'arcs; les arcs sont des paires ordonnées de sommets. Il y a des similarités entre ce type abstrait et celui vu pour les ensembles au chapitre 7. Il y a cependant des différences, dues essentiellement au fait qu'il faut maintenir la cohérence de deux ensembles : ajouter ou retirer un sommet peut affecter l'ensemble des arcs; ajouter ou retirer un arc peut affecter l'ensemble des sommets.

On a pris les conventions suivantes :

- quand on ajoute un sommet, celui-ci est isolé (il n'a aucun arc incident);
- quand on ajoute un arc, si les sommets adjacents à cet arc n'appartiennent pas au graphe, on les ajoute;
- par contre, quand on retire un arc, les sommets adjacents ne sont pas retirés;
- mais quand on retire un sommet, tous les arcs incidents sont supprimés;
- les opérations d'ajout d'un sommet ou d'un arc ne sont pas définies si le sommet ou l'arc est déjà présent dans le graphe concerné.

La signature du type abstrait Graphe est la suivante :

sorte Graphe {*cas orienté*}
utilise Sommet, Entier, Booléen
opérations

graphe-vide	: → Graphe
ajouter-le-sommet _ *à* _	: Sommet × Graphe → Graphe
ajouter-l'arc < _, _ > *à* _	: Sommet × Sommet × Graphe → Graphe
_ *est-un-sommet-de* _	: Sommet × Graphe → Booléen
< _, _ > *est-un-arc-de* _	: Sommet × Sommet × Graphe → Booléen
$d^{\circ+}$ *de*_ *dans*_	: Sommet × Graphe → Entier
_ *ème-succ-de* _ *dans* _	: Entier × Sommet × Graphe → Sommet
$d^{\circ-}$ *de*_ *dans*_	: Sommet × Graphe → Entier
_ *ème-pred-de* _ *dans* _	: Entier × Sommet × Graphe → Sommet
retirer-le-sommet _ *de* _	: Sommet × Graphe → Graphe
retirer-l'arc < _, _ > *de* _	: Sommet × Sommet → Graphe

Dans ce qui suit les variables s, s', s'' sont de sorte Sommet, la variable g est de sorte Graphe, les variables i, j sont de sorte Entier.

Les domaines de définition des opérations sont spécifiés par les préconditions suivantes.

préconditions

ajouter-le-sommet s à g **est-défini-ssi** s *est-un-sommet-de* g = faux
ajouter-l'arc $< s, s' >$ à g **est-défini-ssi**
$s \neq s'$ & $(< s, s' >$ *est-un-arc-de* $g) =$ faux

$d^{\circ+}$ *de* s *dans* g **est-défini-ssi** s *est-un-sommet-de* g = vrai
i *ème-succ-de* s *dans* g **est-défini-ssi**
$(s$ *est-un-sommet-de* $g) =$ vrai & $(i \leq d^{\circ+}$ *de* s *dans* $g) =$ vrai

$d^{\circ-}$ *de* s *dans* g **est-défini-ssi** s *est-un-sommet-de* g = vrai
i *ème-pred-de* s *dans* g **est-défini-ssi**
$(s$ *est-un-sommet-de* $g) =$ vrai & $(i \leq d^{\circ-}$ *de* s *dans* $g) =$ vrai

retirer-le-sommet s *de* g **est-défini-ssi** s *est-un-sommet-de* g = vrai
retirer-l'arc $< s, s' >$ *de* g **est-défini-ssi** $< s, s' >$ *est-un-arc-de* g = vrai

Les axiomes sont présentés ci-dessous dans l'ordre suivant : définitions successives des observateurs *est-un-sommet-de*, *est-un-arc-de*, $d^{\circ+}$, pour le graphe vide et pour le résultat des opérations d'ajout de sommet et d'arc; puis définitions de ces observateurs sur les opérations de suppression d'un sommet ou d'un arc.

Les axiomes caractérisant le $i^{ième}$ successeur d'un sommet (*ème-succ-de*) sont laissés en exercice, ainsi que ceux caractérisant le demi-degré intérieur ($d^{\circ-}$ *de*) et le $i^{ième}$ prédécesseur d'un sommet (*ème-pred-de*).

Rappelons que l'on a pris la convention au chapitre 4 que les axiomes ont comme prémisses implicites les préconditions, convenablement instanciées, des opérations qui y apparaissent.

3.1.1. Définition de est-un-sommet-de sur le graphe vide et les ajouts

s *est-un-sommet-de graphe-vide* = faux

Le graphe vide correspond à un ensemble vide de sommets.

$s = s' \Rightarrow s$ *est-un-sommet-de* (*ajouter-le-sommet* s' *à* $g) =$ vrai
$s \neq s' \Rightarrow s$ *est-un-sommet-de* (*ajouter-le-sommet* s' *à* $g) =$
s *est-un-sommet-de* g

$s = s' \quad \vee \quad s = s'' \Rightarrow s$ *est-un-sommet-de* (*ajouter-l'arc* $< s', s'' >$ *à* $g)$
$=$ vrai

Quand on ajoute un arc, si les sommets adjacents à cet arc n'appartiennent pas au graphe, on les ajoute.

$s \neq s' \quad \& \quad s \neq s''$
$\Rightarrow s$ *est-un-sommet-de* (*ajouter-l'arc* $< s', s'' >$ *à* $g) =$
s *est-un-sommet-de* g

3.1.2. Définition de est-un-arc-de sur le graphe vide et les ajouts

$< s, s' >$ *est-un-arc-de graphe-vide* = faux

Le graphe vide ne contient aucun arc.

$< s, s' >$ *est-un-arc-de* (*ajouter-le-sommet* s'' à $g) =$
$< s, s' >$ *est-un-arc-de* g

Quand on ajoute un sommet, celui-ci est isolé : on n'ajoute aucun arc.

$s = s''$ & $s' = s'''$
$\Rightarrow < s, s' >$ *est-un-arc-de* (*ajouter-l'arc* $< s'', s''' >$ *à* g) = vrai
$s \neq s''$ $\vee$ $s' \neq s'''$
$\Rightarrow < s, s' >$ *est-un-arc-de* (*ajouter-l'arc* $< s'', s''' >$ *à* g) =
$< s, s' >$ *est-un-arc-de* g

3.1.3. Définition du demi-degré extérieur sur les ajouts

Le graphe vide ne contenant aucun arc, d^{o+} n'est jamais défini sur le graphe vide.

$s \neq s' \Rightarrow d^{o+}$ *de* s *dans* (*ajouter-le-sommet* s' *à* g) = d^{o+} *de* s *dans* g
$s = s' \Rightarrow d^{o+}$ *de* s *dans* (*ajouter-le-sommet* s' à g) = 0
Quand on ajoute un sommet, celui-ci est isolé.

$s \neq s'$ & $s \neq s'' \Rightarrow d^{o+}$ *de* s *dans* (*ajouter-l'arc* $< s', s'' >$ *à* g) =
d^{o+} *de* s *dans* g
$s = s'$ & s *est-un-sommet-de* g = vrai
$\Rightarrow d^{o+}$ *de* s *dans* (*ajouter-l'arc* $< s', s'' >$ *à* g) = (d^{o+} *de* s *dans* g)+1
s' était déjà dans g; il a un successeur de plus.

$s = s'$ & s *est-un-sommet-de* g = faux
$\Rightarrow d^{o+}$ *de* s *dans* (*ajouter-l'arc* $< s', s'' >$ *à* g) = 1
s' n'était pas dans g; on l'ajoute; il a un successeur, s''.

$s = s''$ & s *est-un-sommet-de* g = vrai
$\Rightarrow d^{o+}$ *de* s *dans* (*ajouter-l'arc* $< s', s'' >$ *à* g) = (d^{o+} *de* s *dans* g)
s' était déjà dans g; il a le même nombre de successeurs.

$s = s''$ & s *est-un-sommet-de* g = faux
$\Rightarrow d^{o+}$ *de* s *dans* (*ajouter-l'arc* $< s', s'' >$ *à* g) = 0
s' n'était pas dans g; on l'ajoute; il n'a aucun successeur.

3.1.4. Définition des observateurs sur le résultat de la suppression d'un sommet

$s = s' \Rightarrow s$ *est-un-sommet-de* (*retirer-le-sommet* s' *de* g) = faux
$s \neq s' \Rightarrow s$ *est-un-sommet-de* (*retirer-le-sommet* s' *de* g) =
s *est-un-sommet-de* g

$s = s'' \vee s' = s'' \Rightarrow < s, s' >$ *est-un-arc-de* (*retirer-le-sommet* s'' *de* g) = faux
Quand on retire un sommet, on supprime les arcs incidents.

$s \neq s''$ & $s' \neq s''$
$\Rightarrow < s, s' >$ *est-un-arc-de* (*retirer-le-sommet* s'' *de* g) =
$< s, s' >$ *est-un-arc-de* g

$< s, s' >$ *est-un-arc-de* g = vrai
$\Rightarrow d^{o+}$ *de* s *dans* (*retirer-le-sommet* s' *de* g) = (d^{o+} *de* s *dans* g) -1

$< s, s' >$ *est-un-arc-de* g = faux & s *est-un-sommet-de* g = vrai
$\Rightarrow d^{\circ+}$ *de* s *dans* (*retirer-le-sommet* s' *de* g) = $d^{\circ+}$ *de* s *dans* g

Les autres observateurs sont laissés en exercice.

3.1.5. Définition des observateurs sur le résultat de la suppression d'un arc

s *est-un-sommet-de* (*retirer-l'arc* $< s', s'' >$ *de* g) = s *est-un-sommet de* g
Quand on retire un arc on ne retire pas de sommet.

$s = s''$ & $s' = s'''$
$\Rightarrow <s, s'>$ *est-un-arc-de* (*retirer-l'arc* $< s'', s''' >$ *de* g) = faux

$s \neq s''$ ∨ $s' \neq s'''$
$\Rightarrow <s, s'>$ *est-un-arc-de* (*retirer-l'arc* $< s'', s''' >$ *de* g) =
$< s, s' >$ *est-un-arc-de* g

$s = s' \Rightarrow d^{\circ+}$ *de* s *dans* (*retirer-l'arc* $< s', s'' >$ *de* g) = ($d^{\circ+}$ *de* s *dans* g)−1
$s \neq s' \Rightarrow d^{\circ+}$ *de* s *dans* (*retirer-l'arc* $< s', s'' >$ *de* g) = $d^{\circ+}$ *de* s *dans* g

Les autres observateurs sont laissés en exercice. Ces axiomes terminent la définition du type abstrait Graphe, dans le cas orienté.

De plus, pour des raisons de simplicité dans la description du déroulement de certains algorithmes, on prend la *convention* que les successeurs d'un sommet sont numérotés de façon croissante :

$i < j \Rightarrow n^{\circ}$ (i *ème-succ-de* s *dans* g) $< n^{\circ}$ (j *ème-succ-de* s *dans* g)

3.2. Compléments à la spécification des graphes orientés

A partir de ces opérations de base, on peut définir d'autres opérations qui sont très utilisées dans les algorithmes travaillant sur un graphe orienté :

premsucc : Sommet × Graphe → Sommet

est définie pour tout sommet s et tout graphe g tels que $d^{\circ+}$ *de* s *dans* $g \geq 1$,

et on a :

$$premsucc(s, g) = 1 \text{ ème-succ-de } s \text{ dans } g$$

succsuivant : Sommet × Sommet × Graphe → Sommet

est définie pour tout couple de sommets s, s' et tout graphe g tels que :

il existe $i, 1 \leq i < d^{\circ+}$ *de* s *dans* g, tel que $s' = i$ *ème-succ-de* s *dans* g

et on a alors :

$$succsuivant(s, s', g) = (i + 1) \text{ ème-succ-de } s \text{ dans } g$$

Par ailleurs, on a vu que certains graphes sont valués. Cela signifie qu'on a une fonction de coût, de profil (si les coûts sont des nombres réels) :

coût : Sommet × Sommet × Graphe → Réel

Cette opération est définie pour toute paire ordonnée de sommets (s, s') et tout graphe g tels que $< s, s' >$ *est-un-arc-de* g = vrai.

Il faut modifier dans ce cas le profil de l'opération d'ajout d'un arc, car il faut prévoir le coût de cet arc en opérande :

ajouter-l'arc $<$ _, _ $>$ *de-coût* _ *à* _ :
Sommet × Sommet × Réel × Graphe → Graphe.

Pour programmer certains algorithmes, il est parfois commode de prolonger l'opération de coût à toutes les paires ordonnées de sommets du graphe, y compris celles qui ne correspondent pas à un arc : on lui fait rendre pour ces paires une valeur dont on sait qu'elle ne peut être un coût.

D'autres opérations peuvent être définies à partir de la spécification ci-dessus : par exemple les opérations *nb-sommets* et *nb-arcs* dont la spécification est laissée en exercice.

Très souvent, on rencontre dans les algorithmes sur les graphes le schéma suivant de traitement des successeurs d'un sommet s : «pour chaque sommet y, successeur de s, effectuer un certain traitement sur y». Ce schéma s'écrit en utilisant les opérations de base :

for $i := 1$ **to** d^{o+} *de s dans g* **do**
traiter(*i ème-succ-de s dans g*);

3.3. Spécification des graphes non orientés

On a la signature suivante :

sorte Graphe {*cas non orienté*}
utilise Sommet, Entier, Booléen
opérations

graphe-vide	: → Graphe
ajouter-le-sommet _ *à* _	: Sommet × Graphe → Graphe
ajouter-l'arête $<$ _, _ $>$ *à* _	: Sommet × Sommet × Graphe → Graphe
_ *est-un-sommet-de* _	: Sommet × Graphe → Booléen
$<$ _, _ $>$ *est-une-arête-de* _	: Sommet × Sommet × Graphe → Booléen

d° de _ dans _	: Sommet × Graphe → Entier
ème-succ-de _ dans _	: Entier × Sommet × Graphe → Sommet
retirer-le-sommet _ de _	: Sommet × Graphe → Graphe
retirer-l'arête < _, _ > de _	: Sommet × Sommet → Graphe

On a des axiomes similaires à ceux du cas orienté en remplaçant partout *arc* par *arête* et d^{o+} par d^o. Il faut cependant modifier et ajouter des axiomes, car on doit avoir pour tous sommets s, s' et tout graphe g :

$$< s, s' > \textit{ est-une-arête-de } g =< s', s > \textit{ est-une-arête-de } g$$

Par exemple, quand on ajoute une arête $\{s, s'\}$, cela a pour conséquence que $< s, s' >$ *est-une-arête-de* $g =$ vrai et $< s', s >$ *est-une-arête-de* $g =$ vrai ; de même, le degré de s et le degré de s' sont tous deux augmentés de 1, ou initialisés à 1 s'il s'agit d'un nouveau sommet.

On définit les opérations *premsucc* et *succsuivant*, et le schéma de traitement des successeurs d'un sommet et la fonction de coût, comme dans le cas des graphes orientés.

Les axiomes de cette spécification sont laissés en exercice.

4. Représentations des graphes

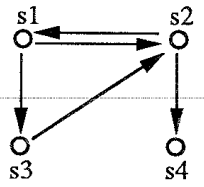
On peut représenter les graphes de plusieurs manières. On peut distinguer deux grandes classes de représentations, selon que l'on privilégie le fait qu'un graphe est un ensemble d'arcs (resp. arêtes) ou un ensemble de sommets.

4.1. Utilisation de matrices

Cette représentation correspond au cas où l'ensemble des sommets du graphe n'évolue pas ; on représente l'ensemble des arcs par un tableau de booléens (cf. chapitre 6) ; comme chaque arc est une paire ordonnée de sommets, le graphe est représenté par une matrice carrée de booléens, dite ***matrice d'adjacence***, de dimension $n \times n$ si le graphe a n sommets. On a donc le type Pascal suivant :

type GRAPHE = **array** $[1..n, 1..n]$ **of** boolean ;

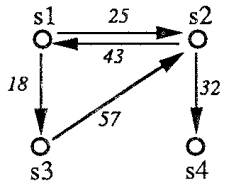
L'élément d'indices i et j de la matrice est vrai si, et seulement si, il existe un arc entre les sommets i et j. La figure 8 donne un exemple de graphe avec sa représentation sous forme matricielle.



	1	2	3	4
1	F	T	T	F
2	T	F	F	T
3	F	T	F	F
4	F	F	F	F

Figure 8. Représentation d'un graphe par une matrice de valeurs booléennes.

Dans le cas où le graphe est non orienté, la matrice est symétrique. Dans le cas où le graphe est valué, on utilise une matrice où l'élément d'indices i et j a pour valeur le poids de l'arc du sommet i au sommet j (resp. de l'arête entre les sommets i et j), si cet arc (resp. arête) existe, et sinon une valeur dont on sait qu'elle ne peut être un poids : par exemple, le plus grand entier utilisable si les poids sont des entiers bornés supérieurement. La figure 9 montre la représentation d'un graphe orienté, valué par des entiers positifs, par une matrice d'entiers où la valeur -1 indique qu'il n'y a pas d'arc entre les sommets correspondants.



-1	25	18	-1
43	-1	-1	32
-1	57	-1	-1
-1	-1	-1	-1

Figure 9. Représentation d'un graphe par la matrice des coûts de ses arcs.

La représentation matricielle est pratique pour tester l'existence d'un arc (ou d'une arête) entre deux sommets : on accède directement à l'élément de la matrice (en un temps constant). De même, il est facile d'ajouter ou de retirer un arc ou une arête. Il est également facile de parcourir tous les successeurs ou prédécesseurs d'un sommet. Pour traiter tous les successeurs du sommet i, on a le schéma suivant, où G est la matrice d'adjacence du graphe :

for $j := 1$ **to** n **do**
  if $G[i,j]$ **then** *traiter* (j);

Cependant, ce schéma demande n tests quel que soit le nombre de successeurs de i. Il en est de même du calcul de $d^{\circ +}$ ou de $d^{\circ -}$. Une consultation complète de la matrice requiert un temps d'ordre n^2, et cette représentation exige un espace mémoire de $\Theta(n^2)$ si le graphe a n sommets, quel que soit le nombre d'arcs ou d'arêtes du graphe. Cela interdit d'avoir des algorithmes d'ordre inférieur à n^2 pour des graphes de n sommets, n'ayant que peu d'arcs. Pour remédier à cet inconvénient, on utilise dans ce cas une représentation appelée «par listes d'adjacence».

4.2. Utilisation de listes d'adjacence

Une autre représentation classique des graphes consiste à représenter l'ensemble des sommets et à associer à chaque sommet la liste de ses successeurs rangés dans un certain ordre. Ces listes sont appelées ***listes d'adjacence.***

Dans le cas où l'ensemble des sommets n'évolue pas, ces listes sont accessibles à partir d'un tableau S, qui contient pour chaque sommet, un pointeur vers le début de sa liste. La figure 10 donne la représentation sous forme de listes d'adjacence du graphe de la figure 8. Le type Pascal correspondant est le suivant :

```
type adr = ↑doublet;
     doublet = record no : 1..n; suiv : adr end;
     GRAPHE = array [1..n] of adr;
```

Il est facile de prendre en compte l'existence de poids sur les arcs en ajoutant un champ au type doublet.

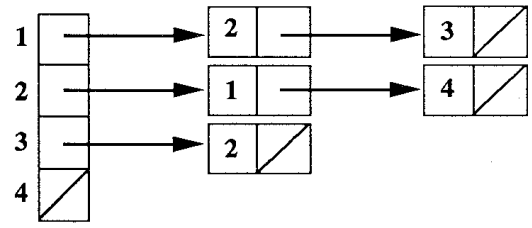


Figure 10. Représentation d'un graphe par listes d'adjacence.

L'intérêt de cette représentation est que l'espace mémoire utilisé est, pour un graphe orienté avec n sommets et p arcs, en $\Theta(n+p)$. Dans le cas d'un graphe non orienté avec p arêtes, l'espace mémoire est en $\Theta(n+2p)$. De plus, quand on a besoin de faire un traitement sur les successeurs d'un sommet s, le nombre de sommets parcourus est exactement le nombre de successeurs de s, soit $d^{o+}(s)$. Le schéma du traitement de tous les successeurs du sommet i est donné ci-dessous :

```
v := S[i];
while v ≠ nil do begin
      j := v↑. no; traiter(j); v := v↑. suiv
end;
```

Le calcul de $d^{\circ+}$ se fait suivant ce schéma.

Un algorithme qui traite tous les arcs d'un graphe de p arcs peut donc être d'ordre p.

En revanche, cette représentation présente l'inconvénient d'exiger, dans le pire des cas, un temps d'ordre n pour tester s'il existe un arc (resp. une arête) entre un sommet donné x et un sommet y (cas où la liste d'adjacence est de longueur $n-1$ et où y est en fin de liste) ou pour l'ajout d'un arc ou d'une arête (avec test de non

répétition). Enfin, on note que cette représentation, contrairement aux matrices, ne permet pas de calculer facilement les opérations relatives aux prédécesseurs ($d^{\circ -}$ et *ème-pred-de*).

Remarque 1 : Ces deux méthodes de représentation, matrice d'adjacence et listes d'adjacence, sont valables que les graphes soient orientés ou non. Dans le cas d'un graphe non orienté, on représente l'arête x—y par deux arcs $x \rightarrow y$ et $y \rightarrow x$. En conséquence, la matrice d'adjacence d'un graphe non orienté est symétrique. Dans le cas d'une représentation par listes d'adjacence, chaque fois qu'on trouve le sommet y sur la liste d'adjacence du sommet x, on trouve le sommet x sur la liste d'adjacence du sommet y (s'il y a p arêtes, on a donc $2p$ doublets).

Remarque 2 : Il existe des variantes de ces deux représentations. Par exemple, la représentation par listes d'adjacence peut se faire en utilisant un tableau pour ranger les listes chaînées, comme on l'a vu au chapitre 5 (cf. exercices). Inversement, on peut vouloir représenter le tableau des têtes de listes par une liste chaînée, dans le cas où on doit ajouter ou supprimer des sommets (cf. exercices).

Remarque 3 : On peut aussi représenter un graphe par listes d'adjacence des prédécesseurs de chaque sommet. Pour certains algorithmes, il est même parfois efficace (en temps, mais coûteux en mémoire) d'avoir à la fois la liste des successeurs et la liste des prédécesseurs de chaque sommet. Le tableau de la figure 10 contient alors deux têtes de liste.

5. Parcours de graphes

Beaucoup de problèmes sur les graphes nécessitent un examen exhaustif des sommets et des arcs (ou arêtes) du graphe. On en verra des exemples comme le tri topologique au chapitre 18 et les connexités au chapitre 19. On va étudier deux types de parcours qui correspondent à des stratégies d'exploration très générales :

• ***Le parcours en profondeur*** (en anglais depth-first search) consiste, à partir d'un sommet donné, à suivre un chemin le plus loin possible, puis à faire des retours en arrière pour reprendre tous les chemins ignorés précédemment. On verra qu'un tel parcours se conçoit naturellement de façon récursive.

• ***Le parcours en largeur*** (en anglais breadth-first search) consiste à explorer le graphe niveau par niveau, à partir d'un sommet donné. Ce parcours, quant à lui, est intrinsèquement itératif.

5.1. Parcours en profondeur : version récursive

On considère un graphe orienté dont tous les sommets sont initialement non marqués. Le parcours en profondeur consiste à choisir un sommet de départ s, à le marquer et

à suivre un chemin issu de s, aussi loin que possible, en marquant les sommets au fur et à mesure qu'on les rencontre. Lorsqu'on est en fin de chemin, on revient au dernier choix fait et on prend une autre direction. On donne ci-après la procédure récursive de parcours en profondeur *prof*. Lors de l'exécution de la procédure *prof* appelée pour le sommet s, tous les sommets qui n'ont pas été marqués avant s et qui peuvent être atteints à partir de s sont visités, et rien qu'eux.

Pour obtenir le parcours total du graphe, il faut un programme principal qui appelle la procédure pour un sommet non marqué, et qui recommence à l'issue de ce parcours tant qu'il reste des sommets non marqués.

5.1.1. Algorithme abstrait

Les programmes qui suivent font appel aux opérations et au schéma de traitement des successeurs du type abstrait donné au paragraphe 3.

Programme principal :

```
var i : integer; gr : Graphe; marque : array [1..n] of boolean;
...
begin
   for i := 1 to n do marque [i] := false;
   for i := 1 to n do
      if not (marque [i] ) then prof(i, gr, marque)
end;
```

Procédure récursive :

```
procedure prof(s : integer; G : Graphe; var M : array [1..n] of boolean);
{s est un sommet }
var j, v : integer; {v est un sommet }
begin
   M[s] := true;
1  {première rencontre de s}
   for j := 1 to d°+ de s dans G do
   begin
      v :=j ème-succ-de s dans G; {rencontre de l'arc (s,v) à l'aller}
      if not (M[v]) then prof(v, G, M); {rencontre de l'arc (s,v) au retour}
   end
2  { dernière rencontre de s}
end prof;
```

On illustre cet algorithme en montrant son fonctionnement sur l'exemple de la figure 11.

Exemple : On suppose que l'ordre d'exploration des successeurs d'un sommet du graphe de la figure 11, est l'ordre des numéros croissants.

• Si on insère dans la procédure *prof*, une instruction d'écriture de s, au moment de la première rencontre, ligne n° 1, on obtient l'impression des sommets dans l'ordre : $s1, s3, s2, s6, s5, s7, s4, s9, s8$.

• Par contre, si cette instruction d'écriture est insérée au moment de la dernière rencontre, ligne n° 2, on obtient : $s2, s6, s3, s5, s7, s1, s9, s4, s8$.

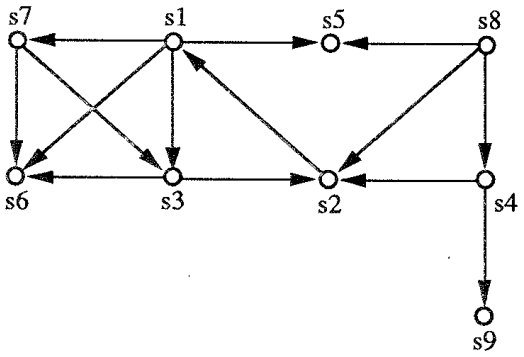


Figure 11.

Pour un parcours donné, tous les appels de *prof* se font avec les mêmes paramètres G et M. Dans tout ce qui suit, lorsqu'il n'y a pas d'ambiguïté, *prof*(s, G, M) est abrégé par *prof*(s).

L'algorithme commence par l'appel de *prof*$(s1)$ qui marque $s1$, le premier sommet du graphe puis choisit $s3$, le premier successeur de $s1$. Comme $s3$ est non marqué, il y a appel récursif de *prof*$(s3)$. Le sommet $s3$ est alors marqué, et son premier successeur $s2$ est choisi. Comme $s2$ est non marqué, il y a appel de *prof*$(s2)$. Le sommet $s2$ est marqué et son premier successeur $s1$ est choisi. Comme $s1$ est déjà marqué, et que $s2$ n'a que ce seul successeur, la procédure *prof*$(s2)$ s'arrête et la visite des sommets reprend au deuxième successeur de $s3$, le sommet $s6$; $s6$ n'ayant pas de successeur, *prof*$(s6)$ marque $s6$ et s'arrête tout de suite. Tous les successeurs de $s3$ ont été explorés, donc *prof*$(s3)$ se termine. *prof*$(s1)$ se poursuit par le choix de $s5$, qui n'est pas encore marqué. L'appel *prof*$(s5)$ marque $s5$ mais se termine tout de suite, $s5$ n'ayant pas de successeur. Le successeur suivant de $s1$ est $s6$ qui est déjà marqué ; *prof*$(s1)$ considère donc le dernier successeur $s7$ de $s1$ qui reste non marqué. La procédure *prof*$(s7)$ marque $s7$ et se termine tout de suite, les successeurs de $s7$ ayant déjà été marqués. *prof*$(s1)$ est maintenant terminée : il y a retour dans le programme principal. Le prochain sommet non encore marqué étant $s4$, il y a appel de *prof*$(s4)$. Le premier successeur de $s4$, soit $s2$ étant déjà marqué, *prof*$(s4)$ marque $s4$ et appelle *prof*$(s9)$, qui marque $s9$ et se termine ensuite. La procédure *prof*$(s4)$ étant terminée, il y a retour dans le programme principal et le sommet $s8$ est choisi. *prof*$(s8)$ marque $s8$ et se termine ensuite, les successeurs $s2, s4, s5$ de $s8$ étant déjà marqués. Tous les sommets étant marqués, l'algorithme se termine.

On donne ci-dessous deux versions de l'algorithme de parcours en profondeur selon que le graphe est représenté par matrice d'adjacence ou par listes d'adjacence.

5.1.2. Représentation par matrice d'adjacence

```
type GRAPHE = array [1..n, 1..n] of boolean;

    procedure prof(i : integer; G : GRAPHE; var M : array [1..n]
    of boolean);
    var j : integer;
    begin
        M[i] := true;
        for j := 1 to n do
            if G[i, j] and not (M[j]) then prof(j, G, M)
    end prof;
```

5.1.3. Représentation par listes d'adjacence

```
type adr = ↑doublet;
    doublet = record no : 1..n; suiv : adr end;
    GRAPHE = array [1..n] of adr;
    procedure prof(i : integer; G : GRAPHE; var M : array [1..n]
    of boolean);
    var j : integer; s : adr;
    begin
        M[i] := true;
        s := G[i];
        while s ≠ nil do begin
                j := s↑.no;
                if not (M[j]) then prof(j, G, M);
                s := s↑.suiv
        end
    end prof;
```

5.1.4. Analyse de la complexité du parcours en profondeur

Généralement, dans les algorithmes de traitement des graphes, on mesure la *complexité en fonction de deux paramètres,* d'une part le nombre n de sommets du graphe, d'autre part le nombre p d'arcs (ou d'arêtes) du graphe.

Supposons qu'on effectue un parcours en profondeur sur un graphe orienté ayant n sommets et p arcs ($p \leq n^2$). Chaque sommet est marqué une et une seule fois, si bien qu'il y a n appels de la procédure *prof.* L'examen des marques se fait exactement n fois dans le programme principal; dans la procédure *prof,* il se fait

pour chaque successeur y d'un sommet x, et ceci pour tous les sommets x. Dans le cas de la représentation par listes d'adjacence, cela revient à parcourir l'ensemble des listes de successeurs : le temps requis est en $\Theta(p)$. Dans le cas de la représentation par matrices d'adjacence, la consultation de la matrice exige un temps en $\Theta(n^2)$. En conséquence, avec une représentation par listes d'adjacence, le temps total requis est en $\Theta(\max(n,p))$, et avec une représentation par matrices d'adjacence, il est en $\Theta(n^2)$.

5.2. Parcours en profondeur : version itérative

Dans ce paragraphe, on décrit une version itérative de la procédure *prof* du parcours en profondeur d'un graphe. Cet algorithme utilise les opérations des types abstraits Graphe et Pile.

Comme l'appel récursif dans *prof* se trouve dans une boucle, il faut garder le sommet visité, pour savoir où reprendre la boucle. A la place de chaque appel de *prof* sur v, il faut empiler ce sommet v. On remarque que si deux sommets s et v se suivent dans la pile, c'est que v est un successeur de s. Lorsqu'on a épuisé la visite le long d'un chemin en profondeur des descendants d'un sommet (booléen *poursuite* mis à false), il faut revenir en arrière. Pour cela, il faut mémoriser le sommet de pile, soit v; puis dépiler pour obtenir le sommet s. Le parcours se poursuit alors en considérant le successeur de s qui vient après v, pour une visite en profondeur de ses successeurs. Pour obtenir le parcours du graphe tout entier, il faut ajouter le programme principal donné au §5.1. en remplaçant l'appel de *prof* par l'appel de *iterprof*.

```
procedure iterprof(s: integer; G: Graphe; var M: array [1..n] of boolean);
{s est un sommet}
var i, j, d : integer; poursuite : boolean; Q : Pile;
{i et j sont des sommets}
begin
   i := s; Q := pile-vide; poursuite : = false; M[i] := true;
       {1ère rencontre de i}
   Q := empiler (Q, i);
   if d°+ de i dans G <> 0 then begin
     poursuite : = true; j := premsucc(i, G)
   end;
   while not (est-vide (Q)) do begin
     while poursuite do
           if not (M[j]) then begin {j n'est pas marqué : on progresse}
             M[j] := true; { 1ère rencontre de j}
             Q := empiler (Q, j); i := j; {on passe aux successeurs}
             if d°+ de i dans G > 0 then j := premsucc(i, G)
             else poursuite : = false {pas de successeurs}
           end
```

```
                else begin {j est marqué : on regarde les autres successeurs de i}
                d := d°+ de i dans G;
                if j <> d ème-succ-de i dans G then j := succsuivant(i, j, G)
                else poursuite : = false {il n'y a plus d'autres successeurs de i}
                end;
        {retour arrière :}
        j := sommet(Q); Q := dépiler(Q); {dernière rencontre de j}
        if not (est-vide(Q)) then begin
                i := sommet(Q);
                d := d°+ de i dans G;
                if j <> d ème-succ-de i dans G then begin
                   j := succsuivant(i, j, G); poursuite : = true
                end
                {sinon on a marqué tous les successeurs de i;
                dernière rencontre de i; on continue le retour arrière}
        end
    end
end iterprof;
```

Dans le cas d'une représentation du graphe par une matrice d'adjacence ou par des listes d'adjacence, on obtient des programmes plus simples (voir exercices).

5.3. Forêt couvrante associée au parcours en profondeur

Durant le parcours en profondeur d'un graphe orienté, on distingue différents types d'arcs.

• Les arcs $x \rightarrow y$ tels que *prof*(x) appelle *prof*(y) sont nommés ***arcs couvrants.*** Les arcs couvrants constituent une ***forêt couvrante d'arborescences de recherche en profondeur.***

Soit r un sommet tel que *prof*(r) est appelée par le programme principal ; considérons le sous-graphe G' des arcs couvrants résultant de l'exécution de *prof*(r). Le graphe non orienté sous-jacent à G' est clairement connexe. Comme la procédure *prof* est appelée exactement une fois pour chaque sommet s de G' qui n'est pas r, $d^{\circ -}$ *de* s *dans* $G' = 1$ pour $s \neq r$; de plus, $d^{\circ -}$ *de* r *dans* $G' = 0$ (car *prof*(r) est appelée par le programme principal). Or, un graphe orienté G' dont le graphe non orienté sous-jacent est connexe et tel que tous ses sommets sont de demi-degré intérieur égal à 1, sauf un, qui est lui, de demi-degré intérieur nul, est une arborescence de racine r (voir exercices). Par exemple, sur la figure 12, l'arc $s1 \rightarrow s3$ est un arc couvrant.

• Les arcs dont l'extrémité terminale est un ascendant (au sens de la terminologie des arbres planaires) de l'extrémité initiale, dans la forêt sont appelés ***arcs en arrière.*** Par exemple, sur la figure 12, l'arc $s2 \rightarrow s1$ est un arc en arrière.

• Les arcs dont l'extrémité terminale est un descendant, dans la forêt, de l'extrémité initiale sont appelés ***arcs en avant.*** Par exemple, sur la figure 12, l'arc $s1 \rightarrow s6$ est un arc en avant.

• Les arcs pour lesquels il n'existe pas de chemin entre leurs extrémités dans la forêt sont appelés des ***arcs croisés.*** Sur la figure 12, $s8 \rightarrow s5$ est un arc croisé.

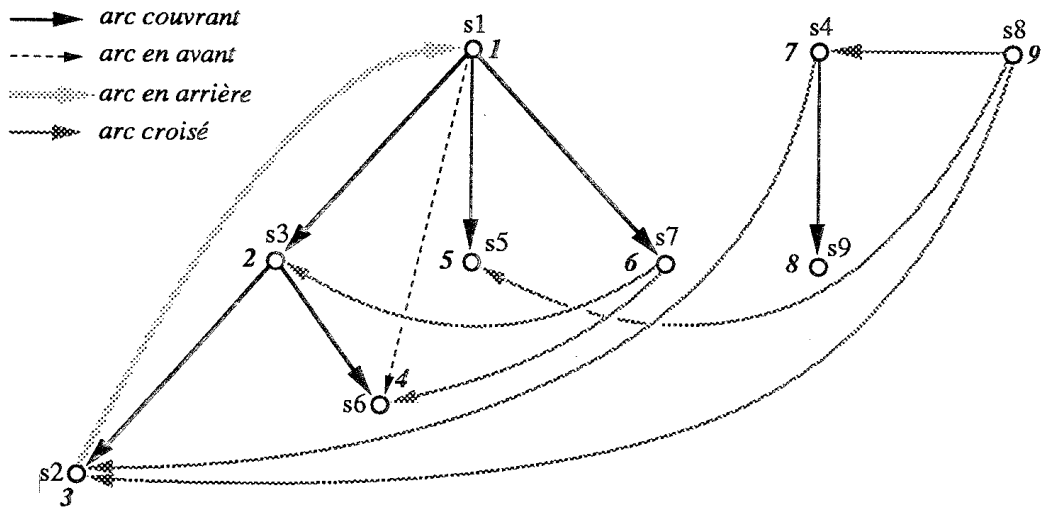


Figure 12. Forêt couvrante du graphe de la figure 11 et numérotation préfixe.

Si l'on prend la convention de dessiner les différents arcs de la forêt au fur et à mesure qu'ils sont empruntés et les différentes arborescences de la gauche vers la droite, au fur et à mesure qu'elles sont constituées par l'algorithme, les arcs croisés sont nécessairement dirigés de la droite vers la gauche.

En effet, raisonnons sur la figure 12. Supposons par exemple, qu'au lieu d'avoir un arc croisé $s7 \rightarrow s3$, on ait un arc $s3 \rightarrow s7$, et que par conséquent, $s7$ soit un successeur de $s3$. Comme l'arc $s1 \rightarrow s7$ a été dessiné après l'arc $s1 \rightarrow s3$, au moment où on visite $s3$, le sommet $s7$ n'est pas encore visité. Comme $s7$ est un successeur de $s3$, *prof*($s3$) doit comporter la visite de $s7$, et l'arc $s3 \rightarrow s7$ devrait être un arc couvrant et non pas un arc croisé.

Considérons une numérotation des sommets qui respecte l'ordre dans lequel ils sont marqués lors du parcours. Pour cela, il suffit d'ajouter dans la procédure *prof*, avant la boucle **for** (ligne 1), les instructions de comptage suivantes :

$compt[i] := cpt;$
$cpt := cpt + 1;$

et d'initialiser à 1 le compteur *cpt* dans le programme principal. La numérotation du sommet i est donnée alors par $compt[i]$. On obtient les propriétés suivantes :

- un sommet y est l'un des k descendants d'un sommet x dans la forêt si, et seulement si : $compt[x] < compt[y] \leq compt[x] + k$
- si l'arc $x \rightarrow y$ est un arc couvrant alors $compt[x] < compt[y]$
- si l'arc $x \rightarrow y$ est un arc en avant alors $compt[x] < compt[y]$
- si l'arc $x \rightarrow y$ est un arc en arrière alors $compt[x] > compt[y]$
- si l'arc $x \rightarrow y$ est un arc croisé alors $compt[x] > compt[y]$

La démonstration de ces propriétés est laissée en exercice.

Sur la figure 12 on a indiqué les valeurs de *compt* pour chaque sommet. Cette numérotation correspond à l'ordre de première rencontre des sommets, qui a été donné pour les sommets du graphe de la figure 11. Il s'agit de l'ordre préfixe sur la forêt couvrante (considérée comme une forêt d'arbres planaires). De même, l'ordre suffixe sur cette forêt couvrante correspond à l'ordre de dernière rencontre des sommets. Pour l'obtenir il faut mettre les instructions de numérotation après la boucle **for** (ligne n° 2).

5.4. Parcours en profondeur d'un graphe non orienté

L'algorithme de parcours en profondeur donné est valable aussi bien dans le cas orienté que dans le cas non orienté. La complexité de l'algorithme dans le cas non orienté est du même ordre : si le graphe a p arêtes, cela revient à considérer qu'il a $2p$ arcs. La complexité est donc en $\Theta(\max(n, 2p))$ pour une représentation par listes d'adjacence, et en $\Theta(n^2)$ pour une représentation par matrice d'adjacence.

Notons que le sens du parcours implique une orientation des arêtes : on parle donc encore d'arcs pour la forêt couvrante. La forêt couvrante associée à un parcours en profondeur d'un graphe non orienté comporte autant d'arborescences que le graphe a de composantes connexes. On ne distingue plus dans ce cas que deux types d'arcs :

- les ***arcs couvrants*** : ce sont les arcs $x \rightarrow y$ tels que *prof*(x) appelle directement *prof*(y).
- les ***arcs en arrière*** : ce sont les arcs $x \dashrightarrow y$ tels que x est un descendant de y dans la forêt. (Attention si l'arête y—x est devenue un arc couvrant, on ne doit pas considérer l'arête x—y quand on construit les arcs en arrière. L'examen de l'arête x—y n'a pas été suivi par un appel récursif de *prof*.)

Il n'y a plus d'arcs en avant, puisque les arêtes ne sont pas orientées.

Il n'y a pas non plus la notion d'arc croisé. En effet, supposons qu'il y ait un arc croisé (au sens des graphes orientés) de y vers x, c'est-à-dire que x et y ne soient pas descendants l'un de l'autre mais soient adjacents, et raisonnons par l'absurde. Si x est visité avant y, *prof*(x) est appelée avant *prof*(y) ; comme la procédure *prof*(x)

ne peut se terminer sans avoir visité tous les sommets accessibles à partir de x, nécessairement y est visité lors de $prof(x)$, si bien que y est un descendant de x dans l'arborescence; d'où la contradiction.

Dans la figure 13, on a donné le graphe non orienté associé au graphe de la figure 11 et on a dessiné la forêt couvrante obtenue par un parcours en profondeur du graphe (toujours avec la convention que les sommets et les successeurs sont choisis par ordre d'indices croissants). Comme le graphe est connexe, on obtient une seule arborescence.

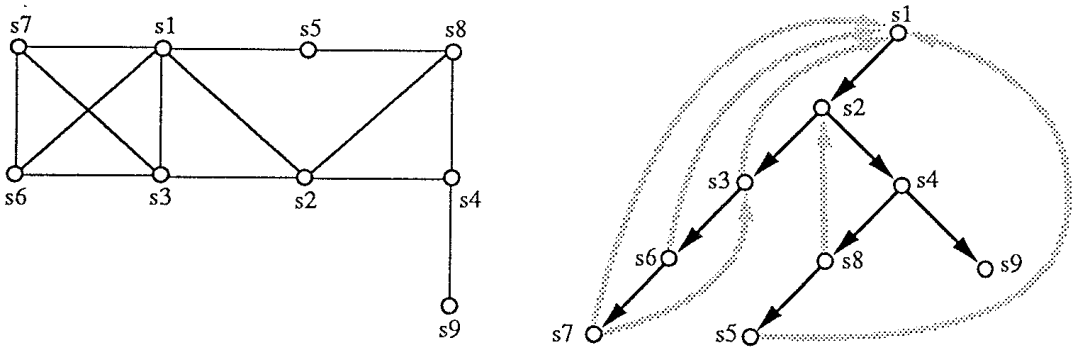


Figure 13. Exemple de forêt couvrante associée à un graphe non orienté.

5.5. Parcours en largeur

On appelle ce parcours, ***parcours en largeur***, car pour un sommet de départ s on commence par visiter tous les successeurs de s avant de visiter les autres descendants de s. Appelons distance de s à y, la longueur d'un plus court chemin issu de s et allant vers y. Le parcours en largeur consiste à visiter d'abord tous les sommets qui sont à la distance 1 de s, puis ceux qui sont à la distance 2, puis à la distance 3 et ainsi de suite.

Un parcours en largeur du graphe de la figure 11, à partir du sommet $s1$, visite dans l'ordre :

$s1$ (sommet de départ)

$s3, s5, s6, s7$ (sommets dont la distance à $s1$ vaut 1)

$s2$ (sommets non encore visités dont la distance à $s1$ vaut 2)

Pour programmer l'algorithme de parcours en largeur de façon itérative, on utilise une structure de file : en effet, lorsque à partir d'un sommet s, on visite ses successeurs non marqués, il est nécessaire de les ranger successivement dans une file

(FIFO) puisque la recherche au niveau suivant repartira de chacun des successeurs de s, à partir du premier.

L'algorithme qui suit, utilise les opérations des types abstraits donnés pour les graphes et le type File.

```
procedure larg(s : integer; G : Graphe; var M : array [1..n] of boolean);
{s est un sommet}
var v, w, i : integer; F : File;
{v et w sont des sommets}
begin
    F := file-vide; M[s] := true;
    F := ajouter(F, s);
    while not (est-vide(F)) do begin
        v := premier(F); F := retirer(F);
        for i := 1 to d^o+ de v dans G do begin
            w := i ème-succ-de v dans G;
            if not (M[w]) then begin
                M[w] := true; F := ajouter(F, w)
            end
        end
    end
end larg;
```

Comme pour la procédure *prof*, lors de l'exécution de *larg*, appelée pour s, on visite tous les sommets qui peuvent être atteints à partir de s et qui n'ont pas encore été marqués, et eux seulement.

Pour parcourir tout le graphe, il faut ajouter dans le programme principal, une boucle sur tous les sommets du graphe, analogue à celle du programme principal du parcours en profondeur. Les sommets du graphe de la figure 11 sont alors visités dans l'ordre suivant : $s1, s3, s5, s6, s7, s2, s4, s9, s8$.

Les procédures obtenues lorsque le graphe est implémenté par matrice d'adjacence ou listes d'adjacence s'écrivent aisément : elles sont laissées en exercice.

La complexité du parcours en largeur est la même que celle du parcours en profondeur.

Comme pour le parcours en profondeur, on peut associer au parcours en largeur une forêt couvrante. Cette forêt couvrante peut être utilisée, par exemple, pour déterminer les composantes connexes dans des graphes non orientés.

Selon les applications on utilise soit le parcours en profondeur (tri topologique, plus courts chemins, connexités...), soit le parcours en largeur (algorithmes de diffusion, algorithmes distribués...).

Exercices

1. a) Déterminer le nombre de graphes orientés (respectivement non orientés) ayant n sommets (en prenant en compte l'existence de boucles).

b) Reprendre les calculs dans le cas où on exclut les boucles.

c) Calculer le nombre d'arêtes d'un graphe non orienté complet de n sommets.

2. a) Soit $G =< S, A >$ un graphe non orienté de n sommets : $S = \{s_1, s_2, ..., s_n\}$. Montrer que : $\sum_{1 \leq i \leq n} d^o(s_i)$ est un nombre pair.

b) En déduire que tout graphe a un nombre pair de sommets de degré impair.

3. Démontrer la proposition caractérisant les arbres.

4. Soit $G = < S, A >$ un graphe orienté ayant n sommets. On appelle $(\mathbb{P})$ la propriété suivante :

$(\mathbb{P})$: pour tout couple de sommets x et y de S, il existe un sommet z de S d'où partent à la fois un chemin de z vers x et un chemin de z vers y.

Si G vérifie $(\mathbb{P})$, G est dit *quasi-fortement connexe.*

a) Montrer qu'un graphe admet une racine si, et seulement si, il est quasi-fortement connexe.

b) Soit G' le graphe non orienté obtenu à partir de G en oubliant l'orientation des arcs. Montrer que les assertions suivantes sont équivalentes :

(1) G admet une racine et G' est un arbre.

(2) G' est sans cycle et G vérifie $(\mathbb{P})$.

(3) G admet $n - 1$ arcs et vérifie $(\mathbb{P})$.

(4) G vérifie $(\mathbb{P})$ et si on supprime un arc quelconque, G ne vérifie plus $(\mathbb{P})$.

(5) Il existe un sommet r de G tel que tout autre sommet est atteint par un unique chemin issu de r.

(6) G' est connexe et il existe un sommet r de G tel que : $d^{\circ-}(r) = 0$ et pour tout $x \neq r, d^{\circ-}(x) = 1$.

(7) G' est sans cycle et il existe un sommet r de G tel que : $d^{\circ-}(r) = 0$ et pour tout $x \neq r, d^{\circ-}(x) = 1$.

Ces assertions caractérisent une ***arborescence.***

5. a) Ecrire les axiomes de la spécification des graphes non orientés.

b) Définir les opérations *premsucc*, *succsuivant*, et le schéma de traitement des successeurs d'un sommet.

c) Donner les axiomes pour les opérations *nb-sommets* et *nb-arêtes* pour un graphe non orienté; on rappelle leur profil :

nb-sommets, *nb-arêtes* : Graphe → Entier

	TETE
1	1
2	4
3	7
4	9

	SUCC
1	2
2	3
3	0
4	1
5	4
6	0
7	2
8	0
9	0

Figure 14. Listes d'adjacence représentées par un tableau.

6. On propose ici une autre représentation d'un graphe à l'aide de deux tableaux *TETE* et *SUCC*. La figure 14 donne la représentation du graphe de la figure 8, qui utilise ces deux tableaux. Le tableau *TETE* indique pour chaque sommet x, l'indice dans le tableau *SUCC* où commence la liste des successeurs de x. Le premier 0 qui suit *SUCC* [*TETE*[x]] indique la fin de la liste des successeurs de x.

Définir le type Pascal correspondant et discuter les avantages et les inconvénients de cette représentation. (On évaluera l'espace mémoire utilisé.)

7. a) On peut modifier la représentation par listes d'adjacence en représentant l'ensemble des sommets par une liste chaînée. On associe toujours à chaque sommet la liste de ses successeurs. Par exemple, le graphe de la figure 8 est représenté comme indiqué à la figure 15.

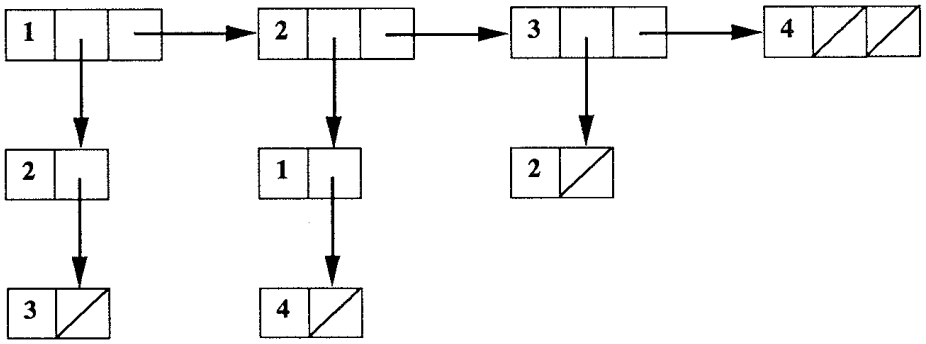


Figure 15. Ensemble des sommets représenté par une liste chaînée.

Définir le type Pascal correspondant et discuter les propriétés de cette représentation.

b) Cette dernière représentation peut être légèrement modifiée : pour un sommet donné x, au lieu d'indiquer dans sa liste de successeurs chaque sommet y qui lui est adjacent, on utilise un pointeur vers le sommet de la liste des sommets correspondant à ce successeur y. La figure 16 montre la représentation obtenue pour le graphe de la figure 8.

Définir le type Pascal correspondant et discuter les avantages et inconvénients de cette représentation.

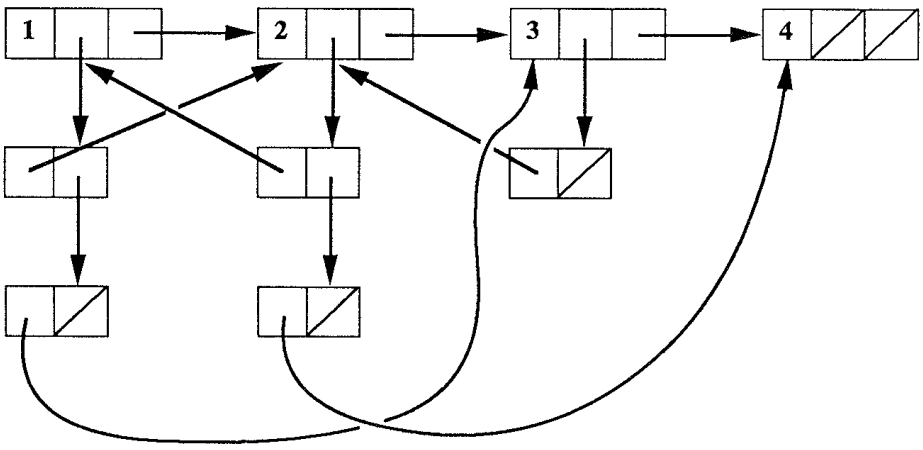


Figure 16

c) Une autre façon de représenter un graphe est de considérer son ensemble de sommets et, non pas la liste des successeurs, mais la liste des prédécesseurs.

Que deviennent la représentation par listes d'adjacence et les représentations définies dans cet exercice et dans le précédent ?

8. En supposant les successeurs ordonnés pour chaque sommet, écrire une procédure de recherche du $k^{ième}$ successeur d'un sommet (lequel n'existe pas nécessairement), pour chacune des représentations d'un graphe mentionnées au §4 ou dans les exercices 6. et 7. a) et b).

9. a) Soit deux graphes $G_1 =< S_1, A_1 >$ et $G_2 =< S_2, A_2 >$ tels que $S_1 = S_2$. On définit la *superposition* des deux graphes G_1 et G_2 comme le graphe $G =< S, A >$ tel que $S = S_1$ et $A = A_1 \cup A_2$. Ecrire une procédure de superposition de deux graphes pour chacune des représentations d'un graphe mentionnées au §4.

b) Soit deux graphes $G_1 =< S_1, A_1 >$ et $G_2 =< S_2, A_2 >$ tels que $S_1 \cap S_2 = \emptyset$ (cette hypothèse n'est pas restrictive, il suffit de renommer les sommets). On définit la *juxtaposition* des deux graphes G_1 et G_2 comme le graphe $G =< S, A >$ tel que $S = S_1 \cup S_2$ et $A = A_1 \cup A_2$. Ecrire une procédure de juxtaposition de deux graphes pour chacune des représentations d'un graphe vues au §4.

10. Un sommet s_p d'un graphe orienté s'appelle un *puits* si pour tout sommet $s_i \neq s_p$, l'arc $s_i \rightarrow s_p$ est dans le graphe et l'arc $s_p \rightarrow s_i$ n'y est pas.

a) Montrer qu'un graphe orienté peut avoir au maximum un puits.

b) Ecrire une fonction PASCAL qui détermine s'il y a un puits s_p dans un graphe donné. Si oui, elle rend s_p, sinon elle rend zéro. Quelle est la complexité de l'algorithme décrit par la fonction ?

11. Soit G le graphe de la figure 17.

a) Donner l'ordre de première (respectivement dernière) rencontre des sommets lors d'un parcours en profondeur des sommets de G.

b) Dessiner la forêt couvrante associée.

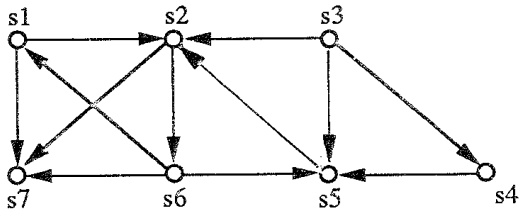


Figure 17

12. On insère dans la procédure *prof* les instructions de comptage comme au §5.3. Montrer les propriétés suivantes :

(1) un sommet y est l'un des k descendants d'un sommet x dans la forêt couvrante si, et seulement si, $compt[x] < compt[y] \leq compt[x] + k$
(2) si $x \rightarrow y$ est un arc couvrant, alors $compt[x] < compt[y]$
(3) si $x \rightarrow y$ est un arc en avant, alors $compt[x] < compt[y]$
(4) si $x \rightarrow y$ est un arc en arrière, alors $compt[x] > compt[y]$
(5) si $x \rightarrow y$ est un arc croisé, alors $compt[x] > compt[y]$.

13. Montrer qu'un graphe orienté admet un circuit si, et seulement si, dans la forêt couvrante associée à un parcours en profondeur du graphe on peut trouver un arc en arrière.

14. En choisissant une représentation des graphes par matrices d'adjacence, montrer que la procédure itérative de parcours en profondeur se simplifie et peut s'écrire comme suit :

```
procedure iterprof(s : integer; G : Graphe; var M : array [1..n]
    of boolean);
var i, j : integer; Q : PILE;
begin
   pile-vide(Q); i := s; M[i] := true; empiler(Q, i); j := 1;
   {début du parcours des successeurs de i}
   while not (est-vide(Q)) do begin
        while j ≤ n do
          if G[i, j] and not (M[j]) then begin
             M[j] := true; empiler(Q, j); i := j; j := 1
          end
          else j := j + 1;
        j := sommet(Q) + 1; dépiler(Q);
        if not (est-vide(Q)) then i := sommet(Q)
   end
end iterprof;
```

15. Dans le cas d'une représentation du graphe par un tableau S et des listes d'adjacence, on peut simplifier la procédure *iterprof* comme suit :

- on utilise un tableau de pointeurs auxiliaire L tel que $L[i]$ indique pour chaque sommet i quel est le prochain successeur à visiter. $L[i]$ est initialisé à $S[i]$ dans le programme principal ;
- le test sur *poursuite* est remplacé par le test à *nil* de $L[i]$;

- l'opération *premsucc*(i, G) se traduit par la première consultation de $L[i]$;
- l'opération *succsuivant*(i, j, G), où j est donné par $L[i]$ s'obtient en poursuivant l'exploration de la liste $L[i]$;

Récrire la procédure *iterprof* dans ce cas.

16. Ecrire une procédure qui indique si un graphe orienté admet une racine. Quelle est la complexité de la procédure ?

17. On appelle *clique* d'un graphe non orienté G tout ensemble de sommets C tel que deux sommets quelconques de C sont reliés par une arête. Montrer que s'il existe une clique, quel que soit l'endroit où on démarre dans la clique le parcours en profondeur de G, tous les sommets de la clique apparaissent sur une même branche de l'arborescence de recherche en profondeur. Sont-ils nécessairement consécutifs sur cette branche ?

18. Le but de l'exercice est de détecter l'existence d'un circuit dans un graphe orienté. On se donne le graphe G représenté par listes d'adjacence.

a) Ecrire une procédure qui construit un tableau P contenant pour chaque sommet le nombre de ses prédécesseurs. Calculer la complexité de cette procédure.

b) Dans le tableau P, on chaîne entre eux les éléments nuls en utilisant la variable *tête*, qui contient l'indice d'un élément nul du tableau. Si i est un sommet sans prédécesseur, $P[i]$ donne l'indice d'un autre sommet sans prédécesseur, précédé du signe $-$, sauf si i est le dernier sommet sans prédécesseur, auquel cas $P[i] = 0$. Par exemple, le tableau :

	4	0	3	2	0	1	0	
devient	4	**0**	3	2	**−2**	1	**−5**	avec *tête* = **7**
ou encore	4	**−5**	3	2	**−7**	1	**0**	avec *tête* = **2**

Ecrire une procédure qui réalise un tel chaînage. Quelle est sa complexité ?

c) Prouver que l'algorithme suivant est correct : «Tant que c'est possible, supprimer du graphe un sommet sans prédécesseur. Si on réussit à supprimer tous les sommets, le graphe est sans circuit». Programmer cet algorithme et évaluer sa complexité.

d) Reprendre la question c) en utilisant les différentes représentations d'un graphe données au §4 et dans l'exercice 7.

19. Ecrire la procédure *larg* dans chacun des cas suivants :

a) avec une représentation du graphe par une matrice d'adjacence;

b) avec une représentation par des listes d'adjacence;

Evaluer dans chaque cas la complexité de la procédure.

Lectures conseillées pour le chapitre 8

Berge, *Graphes et hypergraphes*, Dunod, 2^{e} édition, 1973.

Troisième partie

Algorithmes de recherche

Chapitre 9
Méthodes simples

1. Introduction

Une des utilisations les plus communes de l'informatique est le stockage de collections de données présentant des caractéristiques communes, et la recherche, parmi ces données, d'éléments satisfaisant certains critères.

Si, comme c'est souvent le cas, le nombre de données est important, la recherche, l'adjonction, et la suppression, doivent être réalisées de manière à ne pas demander un temps trop long. Ces traitements dépendent évidemment de la représentation choisie pour structurer les données, que cela soit en mémoire centrale ou en mémoire secondaire.

1.1. Quelques exemples

Les exemples qui suivent démontrent la diversité des problèmes abordés. Considérons tout d'abord l'exemple d'un annuaire téléphonique informatisé. Cet annuaire contient l'ensemble des abonnés au téléphone. Ces abonnés sont caractérisés par leur nom, leur prénom, leur adresse, leur numéro de téléphone, éventuellement leur profession, et ils sont tous distincts.

Un annuaire du téléphone est destiné (quand on l'utilise de manière classique) à permettre de trouver le numéro de téléphone d'un abonné, étant donnés son nom et éventuellement des caractéristiques complémentaires comme le prénom ou l'adresse.

Le nom joue ici un rôle particulier : la recherche dans l'annuaire n'est pas prévue pour rechercher les abonnés de prénoms Annabelle ou Gontran ou ceux qui habitent rue des Jardiniers à Asnières. Dans ce cas, on dit qu'à chaque élément est associée ***une clé*** (le nom dans cet exemple). On recherche donc les éléments dont la clé est égale à une certaine valeur : par exemple, on recherche les abonnés de nom Dupont. Cette recherche peut donner plusieurs résultats. Dans l'exemple de l'annuaire, on peut diminuer le nombre de ces résultats en précisant des caractéristiques supplémentaires.

Il est évident que l'annuaire téléphonique évolue : son organisation doit donc permettre la suppression d'abonnés et l'adjonction de nouveaux abonnés. Cependant, ces opérations étant moins fréquemment utilisées que la recherche, on choisira une représentation qui privilégie les performances de cette dernière opération.

Un autre exemple, assez différent, est celui d'un dictionnaire du français. Dans ce cas, il n'y a pas d'adjonctions ni de suppressions d'éléments (ou très rarement). Les clés sont les mots dont on recherche la définition, et il n'y a qu'un élément par clé.

A l'opposé, on peut considérer un dernier exemple où les adjonctions et les suppressions sont très fréquentes, comme la table des utilisateurs connectés, à un moment donné, à un gros système informatique du type réservation de places d'avion ou consultation de compte bancaire à distance.

1.2. Le problème

Ces exemples ont en commun le fait que le critère de recherche ne porte que sur la valeur de la clé de l'élément recherché. On appelle une telle opération une ***recherche associative.***

On pourrait en effet avoir d'autres types de recherche comme la recherche de «l'élément présent depuis le plus longtemps dans l'ensemble» ou de «l'élément dont la clé est maximale». On trouvera dans d'autres chapitres de ce livre quelques exemples de recherche non associative, comme la recherche du $k^{ième}$ plus petit élément, ou du minimum. On pourrait même vouloir faire des recherches beaucoup plus compliquées : dans le cas de l'annuaire, on pourrait vouloir la liste des médecins qui habitent des villes de plus de 10 000 habitants... Ce type de recherche, très générale, qui ne repose pas sur une identification particulière des éléments (la recherche peut se faire selon la profession ou selon le nom...) nécessite une représentation plus complexe de l'ensemble des données. On parle alors de base de données.

Ce chapitre traite uniquement de recherche associative. Le problème traité s'énonce comme suit. Etant donnée une collection C d'éléments, on veut lui appliquer les trois opérations suivantes : *rechercher* un élément dont la clé est égale à une certaine valeur c; *ajouter* un élément; *supprimer* un élément de clé c.

On a donc, en termes de spécification formelle, la signature suivante :

sorte Collection
utilise Elément, Clé
opérations

clé	:	Elément → Clé
collection-vide	:	→ Collection
ajouter	:	Elément × Collection → Collection
supprimer	:	Clé × Collection → Collection
rechercher	:	Clé × Collection → Elément

La plupart des algorithmes correspondants sont basés sur des comparaisons entre clés. Pour l'analyse de ces algorithmes, qui sont présentés dans ce chapitre et aux chapitres 10 et 11, l'opération fondamentale est donc la *comparaison entre clés.* D'autres algorithmes font intervenir aussi des calculs sur les clés (chapitre 12) et ces calculs sont pris en compte pour établir la complexité.

Lorsqu'on recherche une clé qui est présente, on parle de ***recherche positive.*** Les algorithmes de recherche peuvent varier selon qu'il existe ou non plusieurs éléments ayant la même clé. Dans ce dernier cas, il faut décider si la recherche rend toutes les solutions, ou une solution quelconque, ou une solution particulière. Dans la suite de ce chapitre on donne la version des algorithmes de recherche qui rend une solution quelconque. Si la solution recherchée est particulière, on le précise. L'adaptation de ces algorithmes, afin d'obtenir toutes les solutions, n'est en général pas trop difficile : au lieu de s'arrêter après avoir trouvé une solution, on continue l'algorithme sur le reste de la collection jusqu'à ce qu'il n'y ait plus de solution possible.

S'il n'existe pas de solution, la recherche se termine sur un échec et ne rend pas de résultat. On parle alors de ***recherche négative.*** L'opération *rechercher* est donc une opération partielle. Certains des algorithmes présentés ici impriment un diagnostic d'erreur dans le cas d'une recherche négative; d'autres donnent pour résultat une valeur spéciale (l'outil de programmation naturel serait ici la levée d'une exception, mais ce n'est pas prévu en Pascal).

La situation est analogue pour la suppression d'une clé. Dans les algorithmes présentés ici, on a pris la convention de laisser la collection inchangée lorsqu'il n'y a pas d'élément avec cette clé. Dans le cas où il existe plusieurs éléments de même clé, on supprime l'élément qui est rendu par la recherche de cette clé.

Pour exprimer ces conventions plus formellement, c'est-à-dire comme des axiomes, on ajoute à la signature deux opérations auxiliaires : $\in$, qui teste si un élément existe dans une collection; *nb-occurrences* qui indique combien d'éléments ont une clé donnée.

opérations

$- \in -$: Elément $\times$ Collection $\rightarrow$ Booléen
nb-occurrences: Clé $\times$ Collection $\rightarrow$ Entier

axiomes

$(e \in$ *collection-vide*$)$ = faux
$e = e' \Rightarrow (e \in$ *ajouter*$(e', C))$ = vrai
$e \neq e' \Rightarrow (e \in$ *ajouter*$(e', C))$ = $e \in C$

nb-occurrences(*c*, *collection-vide*) = 0
clé$(e) = c \Rightarrow$ *nb-occurrences*(*c*, *ajouter*$(e, C))$ = *nb-occurrences*$(c, C) + 1$
clé$(e) \neq c \Rightarrow$ *nb-occurrences*(*c*, *ajouter* $(e, C))$ = *nb-occurrences*(c, C)

$c = c'$ & *nb-occurrences*$(c, C) = 0 \Rightarrow$
nb-occurrences$(c$, *supprimer*$(c', C)) = 0$
$c = c'$ & *nb-occurrences*$(c, C) \geq 1 \Rightarrow$
nb-occurrences$(c$, *supprimer*$(c', C)) =$ *nb-occurrences*$(c, C) - 1$
$c \neq c' \Rightarrow$ *nb-occurrences*$(c$, *supprimer*$(c', C)) =$ *nb-occurrences*(c, C)

avec
c, c' : Clé; e, e' : Elément; C : Collection

On a alors la précondition et les axiomes suivants, qui donnent les propriétés de *rechercher* et *supprimer* :

précondition
rechercher(c, C) **est-défini-ssi** *nb-occurrences*$(c, C) \geq 1$

axiomes
clé(*rechercher*$(c, C)) = c$
(*rechercher*$(c, C) \in C$) = vrai
$e =$ *rechercher*$(c, C) \Rightarrow (e \in$ *supprimer*$(c, C))$ = faux
$e \neq$ *rechercher*$(c, C) \Rightarrow (e \in$ *supprimer*$(c, C)) = (e \in C)$

Remarque : Cette spécification n'est pas suffisamment complète au sens du chapitre 4. En effet, on ne peut pas déduire des axiomes quel élément est le résultat de rechercher. On dit simplement que sa clé est c, et qu'il appartient à la collection, ce qui correspond bien au problème posé.

Dans la suite de cette partie, plusieurs représentations des collections sont données ainsi que les algorithmes correspondants pour initialiser, *ajouter, supprimer, rechercher*. Les représentations les plus simples d'une collection utilisent une liste. Si les clés sont munies d'une relation d'ordre, cette liste peut être triée. Dans ce cas, on peut trouver des représentations arborescentes plus efficaces : arbres binaires de recherche, arbres équilibrés. On peut aussi utiliser des décompositions des clés pour partitionner l'ensemble ou même des codages qui permettent des partitions en sous-ensembles de très petites tailles (hachage). Certaines de ces méthodes sont bien adaptées au cas où l'ensemble des données n'est pas stocké en mémoire centrale mais sur un support externe (bande ou disque magnétique). On parle alors de ***recherche externe.***

Afin de faciliter l'expression des algorithmes, on considère dans la suite que *la clé d'un élément est l'élément lui-même* (dans les exemples, il s'agit d'entiers ou de chaînes de caractères). Dans ce cas, l'opération *rechercher* du type abstrait perd de son intérêt (étant donnés un élément-clé et une collection elle rend... l'élément-clé s'il est présent dans la collection). Selon les cas, on modifiera son profil de manière à ce qu'elle rende un booléen (l'élément est présent ou non) ou la place de l'élément recherché, ce qui peut être utile, par exemple, si la recherche est le prélude à une suppression.

Dans le cas où cette opération rend un booléen, et où on la renomme $\in$, on peut remarquer que la spécification des collections est la même que celle des multi-ensembles étudiés au chapitre 6.

2. Recherche séquentielle et recherche autoadaptative

La suite de ce chapitre est consacrée aux algorithmes élémentaires de recherche applicables lorsque la collection des éléments est représentée par une liste. Ces algorithmes sont exemplaires d'un certain nombre de méthodes de conception et d'analyse. Ils ont aussi un intérêt pratique non négligeable : la recherche séquentielle convient tout à fait lorsqu'il y a peu d'éléments, et elle peut être adaptée aux situations dans lesquelles il y a un grand nombre d'éléments, mais où ce sont presque toujours les mêmes éléments qui sont recherchés; lorsque la liste des éléments est triée, on peut aussi faire de la recherche séquentielle, mais on verra aux paragraphes suivants des méthodes beaucoup plus performantes.

2.1. Recherche dans une liste non triée

La façon de procéder la plus naïve consiste à comparer l'élément cherché X successivement à tous les éléments de la liste λ, selon le schéma séquentiel :

$$\text{pour } i \text{ compris entre 1 et } \textit{longueur}(\lambda), \text{comparer } X \text{ et } \textit{ième}(\lambda, i),$$

où *longueur* et *ième* sont les opérations sur le type abstrait Liste définies au chapitre 5. Ce type de recherche a déjà été étudié aux chapitres 2 et 6, et l'on a vu que quelle que soit la représentation des listes, l'une au moins des opérations de recherche, adjonction ou suppression d'un élément dans une liste qui contient n éléments nécessite toujours de l'ordre de n comparaisons.

Plus précisément on a montré que si q_i représente la probabilité que l'élément recherché X apparaisse à la place i de la liste $(i = 1, .., n)$ et si p est la probabilité pour que l'élément X n'appartienne pas à la liste $(p+q_1+...q_n = 1)$, alors le nombre moyen de comparaisons pour une recherche dans une liste avec sentinelle est :

$$\text{Moy}(n) = 1.q_1 + 2.q_2 + ... + n.q_n + (n+1).p.$$

Dans le cas d'une recherche positive (on recherche un élément dont on sait qu'il est dans la liste, c'est-à-dire $p = 0$), on a $\text{Moy}(n) = 1.q_1 + 2.q_2 + ... + n.q_n$.

L'ensemble des q_i étant fixé, cette quantité est minimum lorsque $q_1 \geq q_2 \geq ... \geq q_n$, c'est-à-dire, lorsque les éléments les plus recherchés sont en début de liste. Ainsi, quand on connaît les probabilités de recherche des éléments, on sait dans quel ordre il faut les ranger pour optimiser la recherche séquentielle.

Mais en pratique, on ne connaît pas, en général, les probabilités de recherche des différents éléments. Cependant, si l'on doit effectuer un grand nombre de recherches, il est possible de faire évoluer la liste de sorte que les éléments les plus recherchés se retrouvent en tête; on emploie alors le terme de ***recherche autoadaptative.***

Les méthodes de recherche autoadaptative consistent à réorganiser la liste des éléments au fur et à mesure des recherches, pour minimiser le nombre de comparaisons lors des recherches suivantes.

Une première possibilité est de comptabiliser en mémoire le nombre de recherches pour chaque élément en réarrangeant constamment les éléments par ordre de fréquence décroissante. Les éléments tendent alors à se stabiliser en ordre décroissant de leurs probabilités de recherche, et la liste des éléments est donc organisée de façon optimale pour la recherche séquentielle. Cette méthode a l'inconvénient d'être coûteuse en place à cause des compteurs.

On peut envisager d'autres algorithmes de recherche autoadaptative qui ne gardent aucune information sur la fréquence de recherche des clés, et donc n'utilisent aucune mémoire supplémentaire. Mentionnons deux telles méthodes.

Méthode 1 : après chaque recherche, placer l'élément recherché en tête de liste.

Méthode 2 : après chaque recherche, on fait progresser l'élément recherché d'une place vers la tête de la liste.

La première méthode est bien adaptée à la représentation de la liste des éléments par une liste chaînée alors que dans une représentation contiguë, la mise en tête de l'élément recherché nécessite de décaler d'un rang vers la fin de la liste tous les éléments qui le précèdent. Par contre, la seconde méthode est bien adaptée à une représentation par tableau : à chaque fois qu'un élément est recherché, on l'échange avec l'élément précédent.

L'analyse de ces deux méthodes est délicate (Knuth, 1973); elle montre que la méthode 2 est meilleure *asymptotiquement* que la méthode 1 : après n recherches, lorsque n tend vers l'infini, le coût moyen d'une recherche est plus petit pour la méthode 2 que pour la méthode 1. Cependant, par la méthode 1, les éléments atteignent plus vite un état stationnaire. Le choix de la méthode dépend donc, pour autant qu'on puisse le prévoir, du nombre de recherches qui seront faites sur la liste des n éléments : il est démontré que si ce nombre est inférieur ou égal à n^2 la méthode 1 est préférable, mais au-delà, les performances de la méthode 2 sont meilleures.

2.2. Recherche séquentielle dans une liste triée

Supposons à présent qu'il existe un ordre total sur les éléments, et que la liste est triée en ordre croissant. Pour rechercher un élément X, on peut encore évidemment parcourir séquentiellement la liste jusqu'à ce qu'on trouve soit X, soit un élément

strictement supérieur à X (on sait alors que l'élément X n'est pas dans la liste, puisque les éléments sont rangés par ordre croissant).

Dans le cas le pire (recherche du plus grand élément de la liste, ou recherche d'un élément plus grand que tous ceux de la liste), il faut faire n (ou $n+1$) comparaisons, comme pour une recherche séquentielle dans une liste non triée.

Examinons les performances en moyenne d'une telle méthode. Notons q_i la probabilité que X soit égal au $i^{ième}$ élément de la liste, pour $i = 1, ..., n$; et notons p_j la probabilité que X soit strictement compris entre le $j^{ième}$ et le $(j+1)^{ième}$ élément de la liste, pour $j = 1, ..., n-1$; enfin p_0 (resp. p_n) est la probabilité que X soit strictement inférieur (resp. supérieur) au premier (resp. dernier) élément de la liste .

On a clairement $p_0 + p_1 + ... + p_n + q_1 + ... + q_n = 1$.

Le nombre moyen de comparaisons pour une recherche est alors :

$$\text{Moy}(n) = \sum_{1 \le i \le n} i.q_i + \sum_{0 \le j \le n} (j+1).p_j$$

Plaçons-nous dans l'hypothèse d'équiprobabilité $p_0 = p_1 = ... = p_n$ et $q_1 = q_2 = ... = q_n$ et considérons quelques cas de figures.

• Recherche d'éléments présents dans la liste : tous les p_j sont nuls.

Alors $q_1 = q_2 = ... = q_n = 1/n$, d'où $\text{Moy}(n) = (n+1)/2$.

Dans ce cas on obtient le même nombre moyen de comparaisons que pour la recherche séquentielle dans une liste non triée, ce qui est normal car l'algorithme s'arrête différemment uniquement dans le cas où X n'appartient pas à la liste.

• Recherche d'éléments qui ne sont pas dans la liste (par exemple, en vue d'une adjonction) : tous les q_i sont nuls.

Alors $p_0 = p_1 = ... = p_n = 1/(n+1)$, d'où $\text{Moy}(n) = (n+2)/2$.

Dans ce cas on diminue de moitié le nombre moyen de comparaisons par rapport à la recherche séquentielle dans une liste non triée.

• Recherche d'un élément qui a une chance sur deux d'appartenir à la liste : c'est le cas où $\sum q_i = \sum p_j = 1/2$. On a alors

$$\text{Moy}(n) = \left(\frac{1}{2n} \sum_{1 \le i \le n} i\right) + \left(\frac{1}{2(n+1)} \sum_{0 \le j \le n} (j+1)\right) = \frac{2n+3}{4},$$

ce qui est un résultat légèrement meilleur que celui qu'on obtient dans les mêmes conditions pour une liste non triée.

On montre de même que les opérations d'adjonction et de suppression d'un élément dans une liste triée nécessitent aussi avec cette méthode de l'ordre de n comparaisons.

La méthode de recherche séquentielle dans une liste triée ne tient que très peu compte du fait que les n éléments de la liste sont rangés en ordre croissant, et sa complexité est en $\Theta(n)$ comme lorsque les éléments ne sont pas triés.

3. Recherche dichotomique

La méthode que l'on étudie maintenant, dite ***recherche par dichotomie*** ou ***recherche dichotomique***, utilise pleinement le fait que les éléments sont triés; on montre qu'avec cette méthode, on peut toujours rechercher un élément dans une liste triée en $\Theta(\log n)$ comparaisons.

3.1. Recherche d'une occurrence quelconque

Le principe de la recherche par dichotomie de l'élément X dans une liste triée L consiste à comparer X avec l'élément M du milieu de la liste L :

- si $X = M$, on a trouvé une solution, la recherche s'arrête;
- si $X > M$, il est impossible que X se trouve avant M dans la liste L, et il ne reste à traiter que la moitié (droite) de la liste L;
- de même si $X < M$, X ne peut se trouver que dans la moitié gauche de L.

On continue ainsi la recherche, en diminuant de moitié le nombre d'éléments de la liste restant à traiter, après chaque comparaison; et si on recherche X dans une liste ne contenant aucun élément, la recherche se termine sur un échec : X n'est pas dans L.

La recherche dichotomique est *une méthode de résolution par partition*; la stratégie des méthodes de résolution par partition est la suivante : pour obtenir la solution d'un problème de taille n, on le découpe en a $(1 \leq a \leq n)$ sous-problèmes analogues de taille inférieure. Ces méthodes se décrivent donc très naturellement par des algorithmes récursifs; leur analyse conduit à des équations de récurrence dites de partition (cf. Annexe).

3.1.1. Version récursive de la recherche dichotomique

En suivant le principe de la recherche dichotomique, on est naturellement amené à écrire la procédure récursive suivante :

```
procedure dicho(X : Elément; t : array [1..n] of Elément; g, d : 0..n + 1;
                var res : 0..n);

{cette procédure récursive recherche par dichotomie l'élément X dans le
tableau t dont les éléments sont triés en ordre croissant; le résultat de la
procédure est contenu dans la variable res : c'est 0 si X n'appartient pas
au tableau, et c'est i ∈ {1,.., n} si X se trouve à l'indice i du tableau}

var m : 1..n;
begin if g ≤ d then begin
        m := (g + d) div 2;
        if X = t[m] then res := m
            else if X < t[m] then dicho(X, t, g, m − 1, res)
                 else dicho(X, t, m + 1, d, res)
        end
      else res := 0
end dicho;
```

Pour réaliser une recherche sur une liste de n éléments, on appelle cette procédure avec les paramètres $g = 1$ et $d = n$. On remarque que la liste des éléments est représentée par un tableau; ceci est nécessaire car la recherche par dichotomie demande un accès direct aux éléments de la liste; une représentation à l'aide de pointeurs est donc à proscrire puisque, dans ce cas, l'accès à un élément implique le parcours de tous les éléments qui le précèdent dans la liste.

3.1.2. Justification de l'algorithme

Pour prouver que la procédure *dicho* est correcte, il faut montrer les points suivants :

a) la procédure termine toujours (soit avec $res > 0$, soit avec $res = 0$),

b) s'il existe un entier i $(1 \leq i \leq n)$ tel que $X = t[i]$ alors, après exécution de l'algorithme, on a $res > 0$ et $t[res] = X$,

c) réciproquement, si $res > 0$ alors il existe un entier i $(1 \leq i \leq n)$ tel que $X = t[i]$.

Le dernier point est évident car l'affectation à *res* d'une valeur m différente de 0 se fait seulement lorsque le test $X = t[m]$ est vrai.

Pour prouver que la procédure termine, on va montrer que les appels récursifs sont toujours en nombre fini, c'est-à-dire que la suite (g_i, d_i) des bornes des différents sous-tableaux avec lesquels on appelle récursivement la procédure *dicho* est finie. Supposons les (g_i, d_i) construits pour $0 \leq i \leq k$, avec $g_0 = 1$ et $d_0 = n$; et soit $m_k = (g_k + d_k)$ **div** 2. Plusieurs cas sont à considérer :

- si $g_k > d_k$, ou si $X = t[m_k]$ alors la procédure termine et il n'y a plus d'appel récursif;

- si $g_k \leq d_k$ et $X < t[m_k]$, il y a un appel récursif et $g_{k+1} = g_k$ et $d_{k+1} = m_k - 1$; on a alors $d_{k+1} - g_{k+1} < d_k - g_k$;
- si $g_k \leq d_k$ et $X > t[m_k]$, il y a un appel récursif et $g_{k+1} = m_k + 1$ et $d_{k+1} = d_k$; on a encore $d_{k+1} - g_{k+1} < d_k - g_k$.

Ainsi la suite des écarts $(d_i - g_i)$ est strictement décroissante. Il existe donc un entier p tel que :

- soit $g_p > d_p$, et dans ce cas la procédure termine avec *res* = 0,
- soit $X = t[m_p]$ avec $m_p = (g_p + d_p)$ **div** 2, et dans ce cas la procédure termine avec $res = m_p > 0$.

Prouvons enfin le point b).

On suppose qu'il existe un entier i $(1 \leq i \leq n)$ tel que $X = t[i]$. Considérons la suite $(g_k, d_k)_{0 \leq k \leq p}$ précédemment définie. L'entier p est l'indice d'arrêt : pour $k < p$ on a $g_k \leq d_k$ et d'autre part on a $g_p > d_p$ ou $X = t[m_p]$. Montrons par récurrence la propriété (P) suivante :

Pour tout $k, 0 \leq k \leq p$, il existe un entier i_k, $g_k \leq i_k \leq d_k$, tel que $X = t[i_k]$.

- (P) est vraie pour $k = 0$, c'est notre hypothèse de départ.
- Supposons (P) vraie pour $k < p$.

Par hypothèse de récurrence $g_k \leq d_k$; soit $m_k = (g_k + d_k)$ **div** 2.

Le cas $X = t[m_k]$ est impossible par définition de p, car $k < p$; il reste deux cas :

- soit $X < t[m_k]$, et alors $i_k < m_k$ puisque la liste est triée ; comme $g_{k+1} = g_k$ et $d_{k+1} = m_k - 1$, on a $g_{k+1} \leq i_k \leq d_{k+1}$, et $X = t[i_k]$;
- soit $X > t[m_k]$, et alors $i_k > m_k$ puisque la liste est triée ; comme $g_{k+1} = m_k + 1$ et $d_{k+1} = d_k$, on a $g_{k+1} \leq i_k \leq d_{k+1}$, et $X = t[i_k]$.

La récurrence est établie. En conséquence il existe i_p tel que $g_p \leq i_p \leq d_p$. On n'a donc pas $g_p > d_p$. Si la suite est terminée, c'est parce que $X = t[m_p]$ et $res = m_p > 0$.

3.1.3. Version itérative de la recherche dichotomique

On peut vouloir transformer, pour des raisons d'efficacité, la procédure récursive de la recherche dichotomique en une procédure itérative. La transformation est ici particulièrement simple car, que l'on soit dans le cas $X < t[m]$ ou dans le cas $X > t[m]$, l'appel récursif de *dicho* n'est suivi d'aucun autre traitement : il équivaut donc à recommencer l'exécution de la procédure en changeant les valeurs des paramètres, et ceci tant que la condition $g \leq d$ reste vraie, et que X est différent de $t[m]$. Le résultat transmis dépend de la raison pour laquelle la procédure termine : soit l'élément X est trouvé dans le tableau, et la recherche s'arrête là, en donnant

l'indice de l'élément dans le tableau, soit on sort de l'itération sans avoir trouvé X, et le résultat est 0. Enfin, il faut initialiser les paramètres à leurs valeurs pour le premier appel de la procédure récursive. On obtient ainsi la procédure itérative suivante :

```
procedure dichoiter(X : Elément; t : array [1..n] of Elément; var res : 0..n);
{version itérative de la procédure dicho}
var m : 1..n ; g, d : 0..n + 1;
begin g := 1; d := n;
      while g ≤ d do
          begin m := (g + d) div 2;
              if X = t[m] then begin res := m; return end
              else if X < t[m] then d := m − 1 else g := m + 1
          end;
      res : = 0
end dichoiter;
```

La transformation de *dicho* en *dichoiter* se fait par application d'une règle générale de transformation des procédures récursives terminales en procédures itératives. Une procédure ou une fonction est dite ***récursive terminale*** s'il n'y a aucun traitement à exécuter après chacun des appels à elle-même qu'elle contient. Pour de telles procédures, il existe un schéma de transformation général qui est donné ci-dessous. Considérons la procédure suivante

```
procedure P(U) ;
begin if C then begin D ; P(α(U)) end
      else T
end P;
```

dans laquelle U représente la liste des paramètres; C est une condition portant sur U; D est le traitement de base de la procédure (dépendant de U); $\alpha(U)$ représente la transformation des paramètres; T est le traitement de terminaison (dépendant de U).

Supposons que la procédure P soit appelée avec l'argument V; alors cet appel est équivalent au morceau de programme itératif

```
var U;
begin U := V;
      while C do begin
          D ; U := a(U)
      end;
      T
end;
```

Remarquons que cette transformation conserve l'ensemble des calculs effectués par la procédure. Il est donc indifférent d'analyser la procédure récursive ou sa version itérative (voir paragraphe 3.3).

3.2. Recherche de la première occurrence

3.2.1. Algorithme

Dans le cas où il y a plusieurs occurrences de X dans la liste, il se peut qu'on ne se contente pas d'une solution quelconque, mais qu'on recherche une solution particulière, par exemple la première occurrence de X (i.e. l'occurrence de X située le plus à gauche dans la liste); dans cette hypothèse, si au cours de la recherche on trouve un indice m tel que $t[m] = X$, il faut quand même continuer la recherche à gauche de m (m compris) pour savoir si $t[m]$ est bien la première occurrence de X dans la liste, ou s'il y a une occurrence de X à sa gauche. La recherche se poursuit jusqu'à ce que la liste restant à traiter n'ait plus qu'un seul élément : c'est bien l'occurrence de X cherchée.

Pour obtenir la première occurrence de X on modifie la procédure *dicho* comme suit :

```
procedure dichoprem(X : Elément; t : array[1..n] of Elément; g, d : 0..n + 1;
                    var res : 0..n);
var m : 1..n;
begin
   if g < d then
     begin m := (g + d) div 2;
           if X ≤ t[m] then dichoprem(X, t, g, m, res)
           else dichoprem(X, t, m + 1, d, res)
     end
   else if X = t[g] then res := g
     else res := 0
end dichoprem;
```

3.2.2. Justification de l'algorithme

Pour prouver que la procédure *dichoprem* est correcte, il faut montrer, comme pour la procédure *dicho*, les points a), b), c); mais en plus, dans le point b), il faut montrer que $t[res]$ est la première occurrence de X.

Les preuves de a) et c) sont analogues à ce qui est fait pour *dicho*.

Prouvons le deuxième point.

On suppose qu'il existe un entier i $(1 \leq i \leq n)$ tel que $X = t[i]$. On considère la suite $(g_k, d_k)_{0 \leq k \leq p}$ des bornes des différents sous-tableaux avec lesquels on appelle récursivement la procédure *dichoprem*. L'indice d'arrêt p est le premier indice tel

que $d_p \leq g_p$. On va montrer que pour tout $k, 0 \leq k \leq p$, il existe un entier i_k, $g_k \leq i_k \leq d_k$, tel que $X = t[i_k]$, et pour tout $j, 1 \leq j < g_k$, $t[j] \neq X$. La propriété est vraie pour $k = 0$, car $g_0 = 1$ et $d_0 = n$. Supposons la vraie pour $k < p$.

Par hypothèse de récurrence $g_k \leq d_k$; soit $m_k = (g_k + d_k)$ **div** 2. Comme $k < p$, g_k ne peut être égal à d_k.

Supposons que $g_k < d_k$ et $X \leq t[m_k]$; comme la liste est triée et qu'il existe une occurrence de X entre les indices g_k et d_k, il existe une occurrence de X entre g_k et m_k ; on a $g_{k+1} = g_k$ et $d_{k+1} = m_k$, et pour tout j, $1 \leq j < g_{k+1}$, $t[j] \neq X$.

Supposons à présent que $g_k < d_k$ et $X > t[m_k]$; comme la liste est triée et qu'il existe une occurrence de X entre les indices g_k et d_k, il existe une occurrence de X entre m_k et d_k ; de plus, toutes les occurrences de X sont à droite de $t[m_k]$ car t est trié ; on a donc $g_{k+1} = m_k + 1$ et $d_{k+1} = d_k$ et pour tout j, $1 \leq j < g_{k+1}$, $t[j] \neq X$.

Si p est le dernier indice, c'est que $d_p \leq g_p$; or, il existe i_p tel que $g_p \leq i_p \leq d_p$. Par suite, $g_p = i_p = d_p$, et $X = t[g_p]$. Comme $res = g_p$, on a $t[res] = X$. De plus $t[res]$ est bien la première occurrence de X puisque pour tout j, $1 \leq j < g_p$, $X \neq t[j]$.

3.3. Analyse du nombre de comparaisons

L'opération fondamentale dans la recherche dichotomique est la comparaison entre l'élément cherché et les éléments de la liste.

On étudie d'abord les équations de récurrence vérifiées par les fonctions qui représentent la complexité de la procédure *dichoprem* dans le meilleur et le pire des cas. Puis on analyse la complexité au pire, au mieux et en moyenne de la procédure *dicho* en étudiant l'arbre de décision correspondant.

3.3.1. Analyse de dichoprem

Dans la procédure *dichoprem*, le nombre de comparaisons est le même pour une recherche positive et pour une recherche négative. On montre dans ce paragraphe que le nombre de comparaison pour une recherche est toujours de l'ordre de $\log n$.

La procédure *dichoprem* étant récursive, on cherche à exprimer sa complexité sous forme d'une équation de récurrence. Pour cela, on regarde précisément comment évolue la taille du sous-tableau où se fait la recherche. Si n est la taille du tableau de départ, et que n est pair ($n = 2p$), on a $m = \lfloor (2p + 1)/2 \rfloor = p$. La taille du sous-tableau de gauche est donc $n/2$, ainsi que la taille du sous-tableau de droite. Si n est impair ($n = 2p + 1$), on a $m = \lfloor (2p + 2)/2 \rfloor = p + 1$. La taille du sous-tableau de gauche est donc $\lceil n/2 \rceil$ alors que la taille du sous-tableau de droite est $\lfloor n/2 \rfloor$. Le

nombre maximum de comparaisons pour une recherche avec la procédure *dichoprem* vérifie donc :

$$\begin{aligned}&\mathrm{Max}(1) = 1, \text{et}\\ &\mathrm{Max}(n) = 1 + \mathrm{Max}(\lceil n/2 \rceil), \text{ pour } n > 1.\end{aligned}$$

Par exemple, si X est strictement inférieur à tous les éléments du tableau, on est dans le cas le pire puisqu'on rappelle toujours la procédure sur le sous-tableau de gauche.

Montrons par récurrence que $\mathrm{Max}(n) = \lceil \log_2 n \rceil + 1$.

C'est vrai pour $n = 1$.

Supposons que c'est vrai pour tout $k < n$. On a alors :

$$\mathrm{Max}(n) = 1 + \mathrm{Max}(\lceil n/2 \rceil) = 1 + \lceil \log_2 \lceil n/2 \rceil \rceil + 1$$

- Si n est pair, on a $\lceil n/2 \rceil = n/2$, et la propriété est démontrée.
- Si n est impair, on a $\lceil n/2 \rceil = (n+1)/2$, ce qui donne $\mathrm{Max}(n) = \lceil \log_2(n+1) \rceil + 1$; mais dans ce cas, n ne peut être une puissance de 2, donc $\lceil \log_2(n+1) \rceil = \lceil \log_2 n \rceil$, ce qui montre la propriété.

L'algorithme se comporte de manière identique quelle que soit la place de l'élément cherché dans la liste. On a donc une complexité en moyenne très voisine de la complexité au pire. On ne calcule pas précisément ici la complexité en moyenne de *dichoprem*, mais on prouve qu'elle est de l'ordre de $\log n$, car la complexité dans le pire et dans le meilleur des cas sont toutes les deux de l'ordre de $\log n$.

La taille des sous-tableaux étant comprise entre $\lfloor n/2 \rfloor$ et $\lceil n/2 \rceil$, la complexité de *dichoprem* dans le meilleur des cas est définie par :

$$\begin{aligned}&\mathrm{Min}(1) = 1, \text{et}\\ &\mathrm{Min}(n) = 1 + \mathrm{Min}(\lfloor n/2 \rfloor), \text{ pour } n > 1.\end{aligned}$$

On montre par récurrence que :

$$\mathrm{Min}(n) = \lfloor \log_2 n \rfloor + 1.$$

Comme la complexité en moyenne d'un algorithme est toujours comprise entre la complexité dans le meilleur des cas et la complexité dans le pire des cas, on obtient :

$$\lfloor \log_2 n \rfloor + 1 \leq \mathrm{Moy}(n) \leq \lceil \log_2 n \rceil + 1$$

Donc la complexité moyenne de *dichoprem* vérifie $\mathrm{Moy}(n) = \Theta(\log n)$.

3.3.2. Analyse de dicho

Comme pour tous les algorithmes opérant par comparaisons, on peut illustrer le comportement de la méthode de recherche dichotomique par un arbre de décision (chapitre 3).

Par exemple, la recherche par dichotomie d'un élément dans une liste triée t de 12 éléments différents (procédures *dicho* ou *dichoiter*) est décrite par l'arbre de la figure 1 : les nœuds internes représentent les comparaisons de l'élément cherché avec les éléments $t[i]$ de la liste :

- si $X = t[i]$ on arrête la procédure ;
- si $X > t[i]$ (resp. $X < t[i]$), la procédure se poursuit en comparant X et le fils droit (resp. gauche) de $t[i]$.

Les feuilles $f_0, f_1, ..., f_{12}$ représentent les cas de terminaison sans succès : si $X < t[1]$ la procédure se termine en f_0, si $t[i] < X < t[i+1]$, pour $i = 1, ..., 11$, la procédure se termine en f_i, et si $X > t[12]$, la procédure se termine en f_{12}.

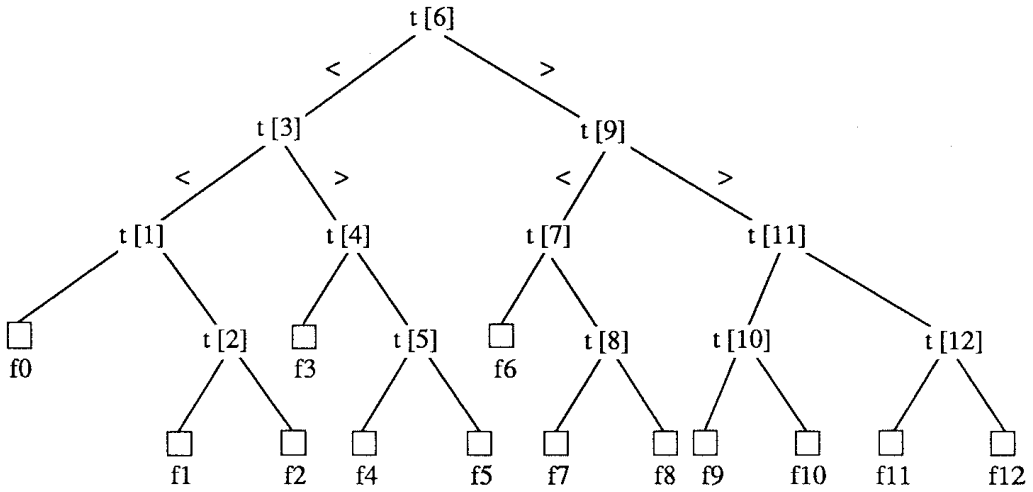


Figure 1. Recherche par dichotomie dans une liste triée de 12 éléments différents.

Il est facile, sur l'arbre de décision, de compter le nombre de comparaisons effectuées dans le cas d'une recherche positive et dans le cas d'une recherche négative ($prof(\nu)$ désigne la profondeur du nœud ν dans l'arbre, et on compte 2 comparaisons s'il y a un test $X = t[m]$, et un test $X < t[m]$) :

- pour la recherche de $X = t[i]$, le nombre de comparaisons est $1 + 2\, prof(t[i])$,
- pour la recherche d'un élément qui n'appartient pas à la liste, et qui est dans l'intervalle représenté par la feuille f_i, le nombre de comparaisons est $2\, prof(f_i)$.

Une recherche positive nécessite donc un nombre de comparaisons compris entre 1 et $2h-1$, où h est la hauteur de l'arbre de décision. Une recherche négative nécessite $2h$ comparaisons dans le cas le pire.

Notons A_n l'arbre de décision associé à l'algorithme *dicho* sur une liste de n éléments. A_n est un arbre binaire localement complet ayant n nœuds internes, et donc $n+1$ feuilles. On va montrer que A_n a toutes ses feuilles situées sur au plus deux niveaux, et on sait (cf. corollaire 1, chapitre 7) que la hauteur d'un tel arbre, sa profondeur moyenne interne et sa profondeur moyenne externe sont de l'ordre de $\log_2 n$. On en déduira donc que la complexité de l'algorithme *dicho* est en moyenne et au pire de l'ordre de $\log_2 n$.

Lemme 1 : Soit p et k tels que $n = 2^p + (k-1)$, avec $1 \leq k \leq 2^p$, alors les feuilles de l'arbre de décision A_n sont toutes à profondeur soit p, soit $p+1$.

Preuve : D'après l'algorithme *dicho*, chaque comparaison partage la liste des éléments restant à traiter en deux sous-listes *lg* et *ld* qui ont, à un élément près, la même cardinalité. Ainsi, l'arbre de décision A_n possède la propriété suivante : *en chaque nœud, les nombres de nœuds internes du sous-arbre gauche et du sous-arbre droit diffèrent au plus de 1.*

Montrons qu'un arbre qui a cette propriété vérifie le lemme 1 ; raisonnons par l'absurde : soit B un sous-arbre de A_n de taille minimum, qui a des feuilles situées à des profondeurs p_1 et p_2, distantes d'au moins deux niveaux ($p_2 \geq p_1 + 2$) ; notons G (resp. D) le sous-arbre gauche (resp. droit) de B (voir figure 2). Les feuilles de G (resp. D) sont situées sur au plus deux niveaux consécutifs, en raison de la propriété de minimalité de B. Supposons que :
1) D a une feuille au niveau p_1 et n'a donc pas de feuille au niveau p_2 ;
2) G a une feuille au niveau p_2 et n'a donc que des nœuds internes au niveau p_1.
Notons alors que D n'a pas non plus de nœud interne au niveau p_2, car sinon D aurait des feuilles au niveau p' avec $p' > p_2$, et comme D a aussi une feuille au niveau p_1, B ne serait pas le plus petit sous-arbre ayant des feuilles séparées d'au moins deux niveaux.

Il en résulte que G a un nœud interne de plus que D au niveau p_1 et que G a (au moins) un nœud interne au niveau $(p_2 - 1)$, alors que D n'en a pas (si D avait un nœud interne au niveau $(p_2 - 1)$, ce nœud donnerait naissance à un autre nœud au niveau p_2, ce qui est contraire à l'hypothèse).

Au total G a donc au moins deux nœuds internes de plus que D, ce qui n'est pas possible dans l'arbre de décision A_n. On a obtenu une contradiction : un tel sous-arbre B n'existe pas. Donc l'arbre A_n a ses feuilles sur au plus deux niveaux.
A partir du corollaire 1 du chapitre 7, on déduit que les feuilles de A_n sont aux niveaux p et $p+1$. □

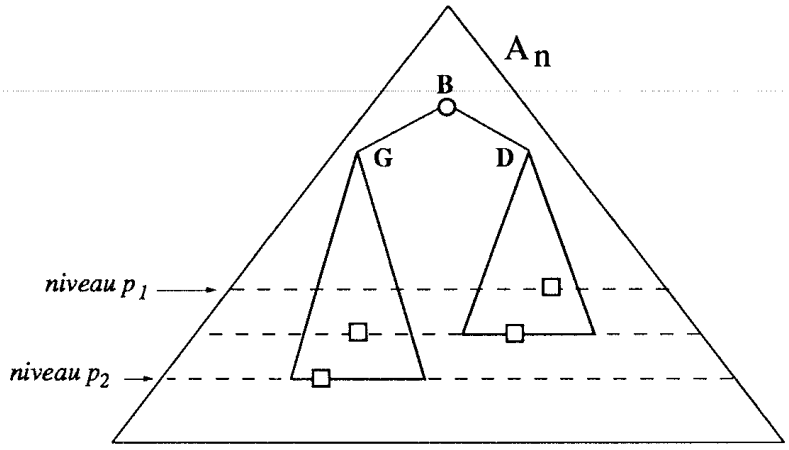


Figure 2

Lemme 2 : Soit $h(A_n)$ la hauteur de l'arbre de décision A_n, et soit $PE(A_n)$ et $PI(A_n)$ les profondeurs moyennes externe et interne de A_n, on a

$$h(A_n) = \lfloor \log_2 n \rfloor + 1$$

$$PE(A_n) = \lfloor \log_2 n \rfloor + 2 - \frac{2^{\lfloor \log_2 n \rfloor + 1}}{n+1}$$

$$PI(A_n) = \lfloor \log_2 n \rfloor + \frac{\lfloor \log_2 n \rfloor + 2}{n} - \frac{2^{\lfloor \log_2 n \rfloor + 1}}{n}$$

Preuve : Posons $p = \lfloor \log_2 n \rfloor$; d'après le lemme précédent, la profondeur maximum des feuilles est $p+1$; la hauteur de l'arbre de décision A_n est donc $h(A_n) = \lfloor \log_2 n \rfloor + 1$.

Soit $n = 2^p + k - 1$, avec $1 \leq k \leq 2^p$: k est le nombre de nœuds internes au niveau p; il y a alors $2k$ feuilles au niveau $p+1$, et $(n+1) - 2k$ feuilles au niveau p. La longueur de cheminement externe de l'arbre A_n est donc

$$LCE(A_n) = (p+1).2k + p.(n+1-2k) = 2k + p.(n+1)$$

Comme $PE(A_n) = \frac{1}{n+1} LCE(A_n)$, en remplaçant k par $n - 2^p + 1$, on obtient donc pour la profondeur moyenne externe :

$$PE(A_n) = p + 2 - \frac{2^{p+1}}{n+1}.$$

Pour obtenir la profondeur moyenne interne $PI(A_n) = \frac{1}{n}LCI(A_n)$, on utilise la formule reliant les longueurs de cheminement externe et interne dans un arbre binaire localement complet, $LCE(A_n) = LCI(A_n) + 2n$, et l'on obtient :

$$PI(A_n) = p + \frac{p+2}{n} - \frac{2^{p+1}}{n}. \qquad \square$$

Revenons à présent à la complexité de l'algorithme *dicho*. Etant donnée une liste triée de n éléments, on note $\text{Min}_{\text{Rech+}}(n)$ (resp. $\text{Moy}_{\text{Rech+}}(n)$, $\text{Max}_{\text{Rech+}}(n)$) le nombre minimum, (resp. le nombre moyen, le nombre maximum) de comparaisons pour une recherche positive; et on note $\text{Min}_{\text{Rech-}}(n)$ (resp. $\text{Moy}_{\text{Rech-}}(n)$, $\text{Max}_{\text{Rech-}}(n)$) le nombre minimum (resp. le nombre moyen, le nombre maximum) de comparaisons pour une recherche négative.

Une recherche positive se termine à la racine de l'arbre de décision dans le meilleur des cas, et dans le pire des cas sur un nœud de profondeur h (où h est la hauteur de l'arbre de décision). Une recherche négative se termine dans le meilleur des cas sur une feuille de profondeur $h-1$, et dans le pire des cas sur une feuille de profondeur h. D'après le calcul du nombre de comparaisons et le résultat concernant la hauteur de l'arbre de décision, on a donc le théorème suivant.

Théorème 1 : La procédure *dicho* de recherche, positive et négative, dans une liste triée de n éléments, a pour complexité :

$$\text{Min}_{\text{Rech+}}(n) = 1 \quad \text{et} \quad \text{Max}_{\text{Rech+}}(n) = 2\lfloor \log_2 n \rfloor + 1,$$

$$\text{Min}_{\text{Rech-}}(n) = 2\lfloor \log_2 n \rfloor \quad \text{et} \quad \text{Max}_{\text{Rech-}}(n) = 2\lfloor \log_2 n \rfloor + 2$$

Pour le calcul de la complexité en moyenne, notons q_i $(i = 1, ..., n)$ la probabilité de rechercher le $i^{ième}$ élément de la liste et p_j $(j = 1, ..., n-1)$ la probabilité de rechercher un élément compris entre le $j^{ième}$ et le $(j+1)^{ième}$ élément de la liste; et soit p_0 (resp. p_n) la probabilité de rechercher un élément plus petit que le premier élément de la liste (resp. plus grand que le dernier).

Le coût moyen d'une recherche s'exprime donc en fonction des profondeurs des nœuds internes $t[i]$ et des feuilles f_j de l'arbre de décision A_n par :

$$\text{Moy}_{\text{Rech}}(n) = \sum_{1 \le i \le n} q_i.(1 + 2.\mathit{prof}(t[i])) + \sum_{0 \le j \le n} p_j.2.\mathit{prof}(f_j)$$

Si l'on suppose que tous les éléments de la liste ont la même probabilité $1/n$ d'être cherchés, alors le coût moyen d'une recherche positive est :

$$\text{Moy}_{\text{Rech+}}(n) = \frac{1}{n} \sum_{1 \le i \le n} (1 + 2.\mathit{prof}(t[i])) = \frac{1}{n}(n + 2LCI(A_n)) = 1 + 2PI(A_n)$$

D'autre part, si l'on suppose que toutes les terminaisons en échec ont la même probabilité $1/(n+1)$, alors on peut évaluer le coût moyen d'une recherche négative :

$$\text{Moy}_{\text{Rech}-}(n) = \frac{1}{n+1} \sum_{0 \le j \le n} 2.prof(f_j) = \frac{2}{n+1} LCE(A_n) = 2PE(A_n)$$

Théorème 2 : Les nombres moyens de comparaisons nécessaires pour une recherche positive et pour une recherche négative dans une liste triée de n éléments en utilisant l'algorithme *dicho* vérifient :

$$\text{Moy}_{\text{Rech}+}(n) \sim 2\log_2 n \quad \text{et} \quad \text{Moy}_{\text{Rech}-}(n) \sim 2\log_2 n.$$

Preuve : A partir des formules données pour $PI(A_n)$ et $PE(A_n)$ dans le lemme 2 qui sont

$PE(A_n) = p + 2 - \frac{2^{p+1}}{n+1}$ et $PI(A_n) = p + \frac{p+2}{n} - \frac{2^{p+1}}{n}$, avec $p = \lfloor \log_2 n \rfloor$,

on obtient

$$PE(A_n) = \log_2 n + c1(n), \text{et } PI(A_n) = \log_2 n + c2(n),$$

où $c1(n)$ et $c2(n)$ sont des fonctions à valeurs dans un petit intervalle autour de zéro.

Ce qui prouve le théorème, d'après l'expression de $\text{Moy}_{\text{Rech}+}(n)$ et $\text{Moy}_{\text{Rech}-}(n)$. □

Remarque : Les procédures *dicho* et *dichoprem* sont toutes deux de complexité logarithmique, en moyenne et dans le pire des cas. Le facteur 2 qui les différencie vient de ce que l'on a deux comparaisons par nœud dans le cas de *dicho*.

3.4. Optimalité

La recherche par dichotomie dans une liste triée de n éléments nécessite, en moyenne comme au pire, un nombre de comparaisons de l'ordre de $\log n$. Ce résultat montre que la recherche dichotomique représente une amélioration importante par rapport à la recherche séquentielle.

Si l'on considère la classe RC des algorithmes de recherche qui procèdent uniquement par comparaisons de l'élément cherché avec ceux de la liste et qui s'arrêtent dès qu'il y a égalité, on va montrer qu'on ne peut faire mieux que par dichotomie.

Tout algorithme de recherche de la classe RC peut être décrit par un arbre binaire de décision (chapitre 3) : chaque nœud interne représente une comparaison entre l'élément cherché X et un élément $t[i]$ de la liste. Il y a trois résultats possibles pour la comparaison : ou bien $X = t[i]$, et dans ce cas l'algorithme termine; ou

bien $X < t[i]$ (resp. $X > t[i]$) et le déroulement de l'algorithme se poursuit dans le sous-arbre gauche (resp. droit) de ce nœud de comparaison.

Chaque exécution de l'algorithme est représentée dans l'arbre de décision par un chemin issu de la racine, et le nombre de comparaisons effectuées lors de cette exécution est proportionnel au nombre de nœuds sur ce chemin. L'algorithme peut se terminer avec succès sur chacun des n éléments de la liste : il y a donc au moins n nœuds internes dans l'arbre de décision.

Les algorithmes de recherche optimaux dans le cas le pire pour une recherche positive sont donc ceux dont l'arbre de décision a une hauteur minimale. Or, on a vu que les arbres binaires pour lesquels ces deux caractéristiques sont minimales sont ceux dont tous les niveaux sont complètement remplis, sauf éventuellement le dernier. Et l'arbre de décision associé à la recherche par dichotomie est bien un arbre de cette forme; donc la recherche dichotomique est optimale dans la classe RC.

La recherche d'un élément n'appartenant pas à la liste se traduit, sur l'arbre de décision associé à l'algorithme, par le parcours d'une branche entière et l'exécution aboutit, à une feuille (vide) de l'arbre de décision. Si la liste est triée, la recherche négative peut se terminer sur chacun des $(n + 1)$ intervalles entre les éléments de la liste, et l'arbre de décision a donc au moins $(n + 1)$ feuilles. Or, les arbres qui minimisent la profondeur maximale d'une feuille sont aussi ceux dont les feuilles sont situées sur deux niveaux au plus. Donc, la recherche dichotomique est optimale, dans le cas le pire, pour une recherche négative.

4. Recherche par interpolation

En pratique, quand on consulte une liste triée, on tient compte de l'élément recherché pour déterminer les éléments de la liste qui lui sont comparés : par exemple, si l'on recherche le mot «Ananas» dans un dictionnaire, on ouvrira le dictionnaire vers le début plutôt qu'au milieu. Ce principe est à la base d'une méthode de recherche dite ***recherche par interpolation.***

Dans la recherche dichotomique, la comparaison a lieu systématiquement avec l'élément $t[(g + d)$ **div** $2]$ qui est au milieu de la sous-liste $t[g]...t[d]$, et cela quel que soit l'élément cherché X. Si les éléments sont des nombres, on peut essayer d'*estimer* la place de X dans la sous-liste; sous l'hypothèse que les nombres de la sous-liste forment une progression régulière, il paraît raisonnable d'aller chercher X autour de la place p définie par $p = g + ((d - g).(X - t[g])$ **div** $(t[d] - t[g]))$, comme le montre la figure 3.

On compare donc X et $t[p]$:

- si $X = t[p]$, on a terminé;

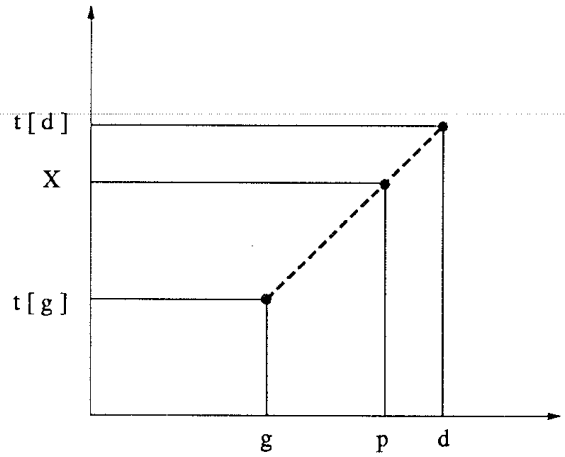


Figure 3. Recherche par interpolation.

- si $X < t[p]$, on recommence une recherche par interpolation de X sur $t[g], ..., t[p-1]$;
- si $X > t[p]$, on recommence une recherche par interpolation de X sur $t[p+1], ..., t[d]$.

Ainsi l'algorithme de recherche par interpolation se distingue de celui de la recherche dichotomique uniquement par un calcul plus compliqué des places des éléments à comparer avec X.

On peut montrer, mais l'analyse est difficile, que sous l'hypothèse d'une distribution uniforme des éléments, le nombre moyen de comparaisons pour rechercher un élément est de l'ordre de $\log_2(\log_2 n)$.

Ceci est une amélioration importante par rapport à la recherche dichotomique : sur une liste comportant 10^9, soit à peu près 2^{30} éléments, une recherche par dichotomie nécessite en moyenne $\log_2(10^9)$, soit à peu près 30 comparaisons, et une recherche par interpolation n'en demande en moyenne qu'à peine 5.

Est-ce à dire qu'il faut abandonner la recherche dichotomique pour la recherche par interpolation? Et que penser alors de «l'optimalité» de la recherche dichotomique vue précédemment?

En fait, la recherche dichotomique est optimale dans la classe des algorithmes qui n'opèrent que par comparaisons (et n'effectuent aucun calcul à partir des éléments); or, la recherche par interpolation *calcule* la place des essais de recherche à partir des éléments eux-mêmes (qui doivent être des nombres); elle n'appartient donc pas à la classe des algorithmes qui n'opèrent que par comparaisons et il n'est donc pas contradictoire de trouver un nombre moyen de comparaisons d'ordre de grandeur strictement inférieur à $\log_2 n$.

D'autre part, la recherche par interpolation n'est efficace que si les éléments augmentent de façon régulière dans la liste, et même dans ce cas, il faut compter le temps de calcul supplémentaire de la place du prochain essai; ceci explique que la recherche par interpolation n'est intéressante en pratique que lorsque le nombre d'éléments de la liste est très grand (car $\log(\log n)$ est alors beaucoup plus petit que $\log n$), ou lorsque les comparaisons sont très coûteuses en temps : c'est en particulier le cas si la liste est stockée en mémoire externe; les comparaisons nécessitent alors des accès à la mémoire externe, qui prennent un temps de l'ordre de 10^3 fois supérieur au temps d'une opération dans l'unité centrale.

Exercices

1. Ecrire des algorithmes de recherche séquentielle, itératif et récursif, dans le cas où la liste des éléments est représentée par une liste chaînée.

2. Connaissant les probabilités de recherche des éléments d'une collection, dans quel ordre faut-il ranger ces éléments pour *maximiser* le nombre de comparaisons de la recherche séquentielle ? Quel est dans ce cas le nombre moyen de comparaisons pour une recherche positive ? Comparer avec le cas où l'ordre de rangement des éléments est optimal.

3. Dessiner l'arbre de décision correspondant à l'algorithme de recherche séquentielle sur une liste de 12 éléments.

4. L'algorithme ci-dessous effectue l'interclassement des deux listes triées contenues dans les tableaux $t1$ et $t2$; le résultat est dans le tableau t ; $t1$ et $t2$ ont respectivement n et m éléments.

```
i := 1; j := 1; k := 1;
while (i ≤ n) and (j ≤ m) do begin
    if t1[i] < t2[j] then begin t[k] := t1[i]; i := i + 1 end
    else begin t[k] := t2[j]; j := j + 1 end;
    k := k + 1
end;
if i ≤ n then {on recopie la fin de t1}
    for j := i to n do begin t[k] := t1[j]; k := k + 1 end
else {on recopie la fin de t2 }
    for i := j to m do begin t[k] := t2[i]; k := k + 1 end;
```

a) Quelle est la complexité au pire, en nombre de comparaisons et en nombre de transferts d'éléments du tableau de cet algorithme ?

b) Calculer les complexités en moyenne, en prenant les conventions suivantes :

- $q = m/(m+n)$ est la probabilité que $t1[n]$ soit plus petit qu'un élément de $t2$;
- on considère que les éléments de $t2$ ont tous la même probabilité d'être le plus petit élément de $t2$ supérieur à $t1[n]$;
- de même, on considère que les éléments de $t1$ ont des probabilités égales d'être le plus petit élément de $t1$ supérieur à $t2[m]$.

c) Transformer cet algorithme en plaçant une sentinelle judicieusement choisie à la fin de $t1$ et $t2$, de manière à éviter le **if-then-else** de la fin. Que deviennent les complexités en moyenne?

Dans quel cas cet algorithme est-il aussi bon que le précédent?

5. Programmer les deux méthodes de recherche autoadaptative présentées au §4, ainsi que la méthode avec compteurs de fréquences. Comparer les performances expérimentales de ces algorithmes avec les résultats théoriques du §4, en exécutant les programmes sur des exemples judicieusement choisis.

6. Dérécursifier la procédure *dichoprem.*

7. Ecrire un algorithme de recherche dichotomique de la dernière occurrence d'un élément (le tester pour la recherche de 2 dans la liste (1, 2, 3)).

8. Ecrire un algorithme de *recherche binaire* selon le même principe que la recherche dichotomique, mais en divisant la liste des éléments en deux sous-listes de tailles respectives 1/3 et 2/3 de la taille de la liste totale. Analyser cet algorithme, et comparer avec la recherche dichotomique.

9. Ecrire un algorithme de *recherche trichotomique* d'un élément X dans une liste triée contenant n éléments, sur le principe suivant :

Comparer X avec l'élément en position $\lfloor n/3 \rfloor$ puis le comparer éventuellement avec l'élément en position $\lfloor 2n/3 \rfloor$; si l'on n'a pas trouvé X, la recherche se poursuit alors sur une liste qui contient trois fois moins d'éléments que la liste d'origine.

Analyser cet algorithme et le comparer à la recherche dichotomique.

10. Peut-on dessiner un arbre de décision correspondant à la recherche par interpolation?

11. Combien d'éléments examine-t-on lors de la recherche par interpolation de l'élément $2k - 1$ (k entier relatif) dans un tableau t qui contient les n premiers entiers pairs rangés en ordre croissant, i.e., $t[i] = 2i$ pour $i = 1, ..., n$?

Lectures conseillées pour le chapitre 9

Horowitz & Sahni, *Fundamentals of Computer Algorithms*, Pitman, 1978.

Knuth, *The Art of Computer Programming*, Vol. 3 : *Sorting and Searching*, Addison-Wesley, 1973.

Chapitre 10
Arbres binaires de recherche

On a étudié au chapitre précédent le problème de la recherche dans une collection d'éléments ordonnés entre eux : on a montré que lorsque n éléments sont représentés par une liste contiguë triée, on peut toujours effectuer une recherche en $\Theta(\log n)$ comparaisons. Mais la représentation contiguë est mal adaptée lorsque la collection évolue : une adjonction ou une suppression peuvent nécessiter $\Theta(n)$ opérations.

Lorsque la collection des éléments n'est pas figée, il paraît donc nécessaire d'utiliser une représentation chaînée, bien que cela prenne de la place supplémentaire en mémoire. Si l'on utilise une liste chaînée (cf. chapitre 5), l'adjonction d'un élément peut être effectuée en un nombre constant d'opérations, mais c'est la recherche et la suppression d'un élément qui demandent trop de temps ($\Theta(n)$ comparaisons au pire).

Quand on veut que les trois opérations – recherche, adjonction, suppression – soient effectuées avec la même efficacité, les représentations les mieux adaptées sont les structures arborescentes. Les méthodes de recherche arborescentes reposent toutes sur le même principe : la comparaison avec le (ou les) élément(s) d'un nœud de l'arbre permet d'orienter la suite de la recherche dans l'arbre. La structure fondamentale est celle d'*arbre binaire de recherche.*

1. Définition et recherche

On peut représenter une collection ordonnée de n éléments par un arbre étiqueté ayant n nœuds (chapitre 7). Chaque nœud contient un élément et la répartition des éléments dans l'arbre permet de guider la recherche en faisant des comparaisons.

1.1. Arbre binaire de recherche

Définition : Un ***arbre binaire de recherche*** est un arbre binaire étiqueté tel que pour tout nœud ν de l'arbre :

- les éléments de tous les nœuds du sous-arbre gauche de ν sont inférieurs ou égaux à l'élément contenu dans ν,
- et les éléments de tous les nœuds du sous-arbre droit de ν sont supérieurs à l'élément contenu dans ν.

Il résulte immédiatement de cette définition que le *parcours symétrique* d'un arbre binaire de recherche produit la suite des éléments *triée en ordre croissant.*

Sur la figure 1.a on a dessiné un arbre binaire de recherche qui représente l'ensemble $E = \{a, d, e, g, i, l, q, t\}$ muni de l'ordre alphabétique. Notons qu'il y a plusieurs représentations possibles d'un même ensemble par un arbre binaire de recherche : l'arbre de la figure 1.b représente aussi E.

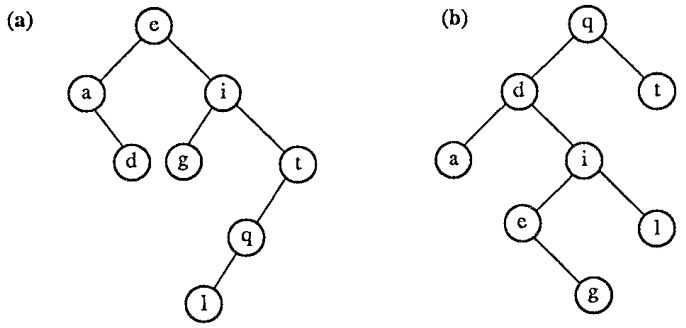


Figure 1. Arbres Binaires de Recherche.

1.2. Recherche d'un élément

Pour rechercher (une occurrence) d'un élément dans un arbre binaire de recherche, on compare cet élément au contenu de la racine :

- s'il y a égalité, on a trouvé l'élément et la recherche est terminée (*recherche positive*),
- si l'élément est plus petit (resp. plus grand) que celui de la racine, on poursuit la recherche dans le sous-arbre gauche (resp. droit); si ce sous-arbre est vide, il y a échec : l'élément n'appartient pas à l'arbre binaire de recherche initial *(recherche négative)*.

Rappelons que si $B =< o,\ G,\ D >$ est un arbre étiqueté dont la racine contient l'élément r, on note (cf. chapitre 7) $B =< r,\ G,\ D >$.

La spécification de l'opération de recherche d'un élément (que l'on suppose de sorte Elément) dans un arbre binaire de recherche s'écrit :

rechercher : Elément $\times$ Arbre $\rightarrow$ Booléen

Si G et D sont des variables de sorte Arbre, x et r des variables de sorte Elément, on a les axiomes suivants :

axiomes

rechercher(x, *arbre-vide*) = faux
$x = r \Rightarrow$ *rechercher*(x, $< r,\ G,\ D >$) = vrai
$x < r \Rightarrow$ *rechercher*(x, $< r,\ G,\ D >$) = *rechercher*(x, G)
$x > r \Rightarrow$ *rechercher*(x, $< r,\ G,\ D >$) = *rechercher*(x, D)

Une version récursive de l'algorithme de recherche est donnée dans la fonction *chercher*, qui utilise le type Pascal ARBRE.

```
type ARBRE  = ↑nœud;
       nœud = record
                val : Elément;
                g, d : ARBRE
              end;

function chercher(X : Elément; A : ARBRE) : boolean;
{cette fonction récursive recherche si un élément X appartient à un arbre binaire de recherche A ; le résultat est true si X appartient à A, et false sinon}
begin
   if A = nil then chercher := false
   else if A↑.val = X then chercher := true
        else if X < A↑.val then chercher := chercher (X, A↑.g)
             else chercher := chercher(X, A↑.d)
end chercher;
```

Cet algorithme a une forme semblable à celle de l'algorithme de recherche dichotomique; la récursion est terminale, mais la fonction renvoie un résultat. La transformation en une version itérative est laissée en exercice.

2. Adjonction d'un élément

L'adjonction d'un élément dans une collection quelle qu'en soit la représentation, se fait toujours selon le principe suivant : d'abord déterminer la place où l'on doit faire l'adjonction, puis réaliser l'adjonction à cette place. Dans un arbre binaire de recherche, l'opération d'adjonction se décompose en une étape de recherche, qui renvoie des informations sur la place où doit être inséré le nouvel élément, suivie de l'adjonction proprement dite.

2.1. Adjonction aux feuilles

Pour ajouter un élément dans un arbre binaire de recherche, on le compare au contenu de la racine pour déterminer s'il faut l'ajouter dans le sous-arbre gauche

ou le sous-arbre droit, et l'on rappelle la procédure récursivement. Le dernier appel récursif se fait sur un arbre vide et l'on crée alors à cette place le nœud contenant l'élément à ajouter. Le nouveau nœud devient donc une feuille de l'arbre binaire de recherche. Décrivons la spécification de l'opération d'adjonction aux feuilles d'un arbre binaire de recherche :

ajouter-feuille : Elément × Arbre → Arbre

Si G et D sont des variables de sorte Arbre, x et r des variables de sorte Elément, on a les axiomes suivants :

axiomes

ajouter-feuille(x, *arbre-vide*) = $< x$, *arbre-vide*, *arbre-vide* $>$
$x \leq r \Rightarrow$ *ajouter-feuille*($x, < r, G, D >$) $=< r$, *ajouter-feuille*$(x, G), D >$
$x > r \Rightarrow$ *ajouter-feuille*($x, < r, G, D >$) $=< r, G$, *ajouter-feuille*$(x, D) >$

2.1.1. Procédures d'adjonction

Les procédures *ajouterfeuille* et *ajouterfeuilleiter* sont une version récursive et une version itérative de l'adjonction aux feuilles dans un arbre binaire de recherche. Dans le programme récursif, c'est le passage par référence du paramètre A qui permet d'affecter à un de ses champs un pointeur vers le nœud créé. Dans une version itérative de l'algorithme, l'affectation de ce lien doit être explicite. La procédure *ajouterfeuilleiter* résoud le problème en testant avant chaque progression dans l'arbre si le nœud suivant sera une feuille. (Une autre façon de faire, laissée en exercice, consiste à conserver à chaque étape de la progression dans l'arbre des informations concernant le nœud précédent.)

```
procedure ajouterfeuille (X : Elément; var A : ARBRE);
{procédure récursive d'adjonction d'un élément X aux feuilles d'un arbre
binaire de recherche A}
begin
   if A = nil then begin new(A); A↑.val := X; A↑.g := nil; A↑.d :=nil end
               else if X ≤ A↑.val then ajouterfeuille(X, A↑.g)
                                  else ajouterfeuille(X, A↑.d)
end ajouterfeuille;


procedure ajouterfeuilleiter(X : Elément; var A : ARBRE);
{procédure itérative d'adjonction d'un élément X aux feuilles d'un arbre
binaire de recherche A}
var B, Aux : ARBRE;
    poursuite : boolean;
begin
    new(B); B↑.val := X; B↑.g := nil; B↑.d := nil;
    {création du nœud contenant l'élément à ajouter}
```

```
if A = nil then A := B
else begin
    Aux := A; poursuite := true;
    while poursuite do
        if X ≤ Aux↑.val then
        {on teste avant chaque progression dans l'arbre si le nœud
        suivant sera une feuille}
          if Aux↑.g = nil then begin
              Aux↑.g := B;
              poursuite := false
          end
          else Aux := Aux↑.g {on descend à gauche}
        else if Aux↑.d = nil then begin
              Aux↑.d := B;
              poursuite := false
          end
          else Aux := Aux↑.d {on descend à droite}
  end
end ajouterfeuilleiter;
```

2.1.2. *Construction d'un arbre binaire de recherche par adjonctions successives aux feuilles*

On peut construire un arbre binaire de recherche par adjonctions successives à partir de la donnée de la suite de ses éléments : par exemple, l'adjonction successive des éléments e, i, a, t, d, g, q, l conduit aux arbres binaires de recherche dessinés sur

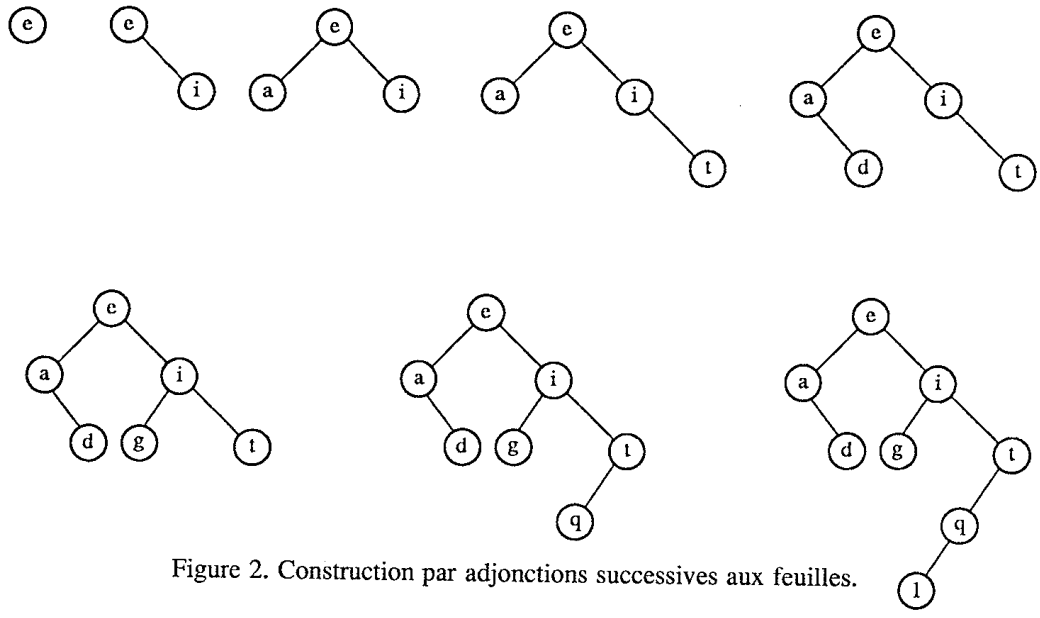


Figure 2. Construction par adjonctions successives aux feuilles.

la figure 2, et l'arbre résultant de toutes les adjonctions est celui de la figure 1.a. D'autres façons d'ordonner la suite des éléments en entrée peuvent aussi conduire au même arbre binaire de recherche résultat, par exemple, l'adjonction successive des éléments dans l'ordre e, a, d, i, g, t, q, l conduit au même arbre. On étudie en exercice le nombre de façons d'ordonner la suite des éléments en entrée qui produisent le même arbre binaire de recherche.

2.2. Adjonction à la racine

On peut ajouter un élément, non seulement aux feuilles, mais à n'importe quel niveau d'un arbre binaire de recherche, et en particulier à la racine (ceci est à rapprocher des méthodes de recherche séquentielle autoadaptative : l'adjonction à la racine peut présenter des avantages, par exemple dans le cas où c'est juste après l'adjonction d'un élément que l'on effectue le plus de recherches le concernant).

Pour ajouter un élément X à la racine d'un arbre binaire de recherche, il faut d'abord couper l'arbre binaire de recherche initial en deux arbres binaires de recherche G et D contenant respectivement tous les éléments inférieurs ou égaux à X, et tous les éléments supérieurs à X (figure 3), puis former l'arbre dont la racine contient X, et qui a pour sous-arbre gauche (resp. droit) l'arbre binaire de recherche G (resp. D). On obtient bien un arbre binaire de recherche.

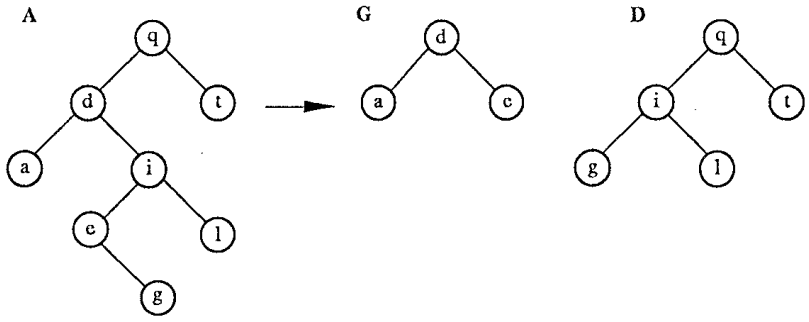


Figure 3. Coupure selon l'élément f.

C'est évidemment l'étape de *coupure* qui est la plus délicate; il est important de remarquer qu'il n'est pas nécessaire de parcourir tous les nœuds de l'arbre initial A pour former G et D. Ce sont les nœuds situés sur le chemin C suivi lors de la recherche de X qui déterminent la coupure : en effet, si un nœud de C contient un élément plus petit (resp. plus grand) que X, il vient se placer sur le bord droit de G (resp. bord gauche de D), et par la propriété d'arbre binaire de recherche, il entraîne avec lui dans G (resp. D) tout son sous-arbre gauche (resp. droit).

La spécification de l'opération d'adjonction à la racine dans un arbre binaire de recherche se décrit comme suit :

ajouter-rac : Elément × Arbre → Arbre,

Si G et D sont des variables de sorte Arbre, x et r des variables de sorte Elément, on a les axiomes suivants :

axiomes

ajouter-rac$(x,$ *arbre-vide*$) = <x,$ *arbre-vide*, *arbre-vide*$>$
racine(*ajouter-rac*$(x, <r, G, D>)) = x$
$x < r \Rightarrow$ *g*(*ajouter-rac*$(x, <r, G, D>)) =$ *g*(*ajouter-rac*$(x, G))$
d(*ajouter-rac*$(x, <r, G, D>)) = <r,$ *d*(*ajouter-rac*$(x, G)), D>$
$x \geq r \Rightarrow$ *g*(*ajouter-rac*$(x, <r, G, D>)) = <r, G,$ *g*(*ajouter-rac*$(x, D))>$
d(*ajouter-rac*$(x, <r, G, D>)) =$ *d*(*ajouter-rac*$(x, D))$

2.2.1. Procédure de coupure

La procédure *couper* réalise la coupure de l'arbre binaire de recherche A selon l'élément X ; le résultat est un couple (G, D) d'arbres binaires de recherche : G contient tous les éléments de A plus petits que X (au sens large) et D contient tous les élément de A plus grands que X. Le passage par référence des paramètres G et D permet, comme dans la procédure *ajouterfeuille*, de «raccrocher» les sous-arbres. De plus il faut passer le paramètre A par référence, car l'arbre d'origine est modifié du fait de la manipulation des pointeurs.

```
procedure couper(X : Elément; var A, G, D : ARBRE);
{procédure récursive de coupure de l'arbre binaire de recherche A, selon
l'élément X, en deux arbres binaires de recherche G et D ; G contient
tous les éléments de A inférieurs ou égaux à X et D tous les élément de A
strictement supérieurs à X}
begin
    if A = nil then begin G := nil; D := nil end
            else if X < A↑.val then begin
                                    D := A; couper(X, A↑.g, G, D↑.g)
                                    end
                                    else begin
                                    G := A; couper(X, A↑.d, G↑.d, D)
                                    end
end couper;
```

Notons que dans le cas où X est dans l'arbre A, le chemin C de coupure selon l'élément X dans la procédure *couper* est le chemin de recherche de X suivi du bord gauche du sous-arbre droit du (premier) nœud contenant X. On peut améliorer cette procédure en arrêtant la récursion dès que l'on rencontre l'élément de coupure (cf. exercices).

D'autre part, la procédure *couper* étant récursive terminale, elle peut être dérécursifiée sans pile. L'écriture de la version itérative de la coupure est renvoyée en exercice (attention au passage par référence des paramètres G et D).

2.2.2. *Justification de l'algorithme*

Il est intéressant de prouver la correction de l'algorithme de coupure, en particulier dans le cas où l'élément X est déjà dans l'arbre A.

Tout d'abord l'algorithme termine toujours : à chaque appel récursif, on descend d'un niveau dans l'arbre, et l'on arrive donc au bout d'un nombre fini d'appels (puisque la hauteur de l'arbre est finie) sur un arbre vide, où l'on s'arrête.

Ensuite raisonnons par récurrence sur la hauteur des arbres, pour montrer que l'algorithme construit un arbre binaire de recherche G qui contient tous les éléments de A inférieurs ou égaux à X et un arbre binaire de recherche D qui contient tous les éléments de A strictement supérieurs à X :

- si A est vide, alors G et D sont vides, et la propriété est vraie,
- sinon, notons r le contenu de la racine de A.

Supposons que $X < r$.

L'algorithme de coupure sur $g(A)$ construit deux arbres binaires de recherche G_0 et D_0. Montrons que si la propriété est vraie pour G_0 et D_0, elle est aussi vraie pour les arbres $G = G_0$ et $D =< r, D_0, d(A) >$ construits à cette étape.

Par hypothèse de récurrence, G_0 (resp. D_0) contient tous les éléments de $g(A)$ inférieurs ou égaux à X (resp. strictement supérieurs à X).

Puisque $G = G_0$ et que $X < r$, G est un arbre binaire de recherche qui contient tous les éléments de A inférieurs ou égaux à X.

Montrons maintenant que $D =< r, D_0, d(A) >$ est bien un arbre binaire de recherche dont tous les éléments sont strictement supérieurs à X.

Le sous-arbre gauche D_0 de D provient de $g(A)$; ses éléments sont donc inférieurs ou égaux à r. Le sous-arbre droit de D est $d(A)$, donc ses éléments sont strictement supérieurs à r. Ainsi, D est un arbre binaire de recherche. De plus il contient tous les éléments de A strictement supérieurs à X, à savoir r, les éléments de $d(A)$, et tous les éléments de $g(A)$ strictement supérieurs à X.

Le cas où $X \geq r$ se traite de façon analogue.

2.2.3. *Procédure d'adjonction à la racine*

La procédure *ajouterrac*, d'adjonction d'un élément X à la racine d'un arbre binaire de recherche A, utilise la procédure de coupure de l'arbre binaire de recherche selon cet élément.

```
procedure ajouterrac(X : Elément; var A : ARBRE);
{cet algorithme réalise l'adjonction de l'élément X à la racine de l'arbre binaire de recherche A ; il utilise la procédure de coupure d'un arbre binaire de recherche}
var R : ARBRE;
begin
    new(R); R↑.val := X;
    couper(X, A, R↑.g, R↑.d);
    A := R
end ajouterrac;
```

3. Suppression d'un élément

Pour supprimer un élément d'une collection, il faut tout d'abord déterminer sa place, puis effectuer la suppression proprement dite, qui s'accompagne éventuellement d'une réorganisation des éléments.

Dans un arbre binaire de recherche, la suppression de l'élément commence par sa recherche, et la suite dépend de la place de l'élément à supprimer dans l'arbre.

• Si c'est *un nœud sans fils*, la suppression est immédiate (c'est le cas, par exemple, pour le nœud contenant g, noté simplement g, dans l'arbre binaire de recherche de la figure 1.a).

• Si c'est un *nœud qui a un seul fils*, il suffit de le remplacer par ce fils, et l'arbre résultant reste un arbre binaire de recherche (par exemple, la suppression de t dans l'arbre de la figure 1.a donne l'arbre binaire de recherche de la figure 4.a).

• Si c'est un *nœud qui a deux fils*, il y a deux solutions pour conserver la structure d'arbre binaire de recherche (éléments en ordre croissant dans la lecture symétrique), tout en modifiant le moins possible l'arbre binaire initial :

- soit remplacer le nœud contenant l'élément à supprimer par celui qui lui est *immédiatement inférieur* (qui contient le plus grand élément de son sous-arbre gauche) : le remplacement de e par d dans l'arbre de la figure 4.a donne l'arbre de la figure 4.b,
- soit remplacer le nœud contenant l'élément à supprimer par celui qui lui est *immédiatement supérieur* (qui contient le plus petit élément de son sous-arbre-droit) : le remplacement de i par l dans l'arbre de la figure 4.a donne l'arbre de la figure 4.c.

Ces deux solutions sont équivalentes lorsque tous les éléments de l'arbre binaire de recherche sont distincts; (le lecteur est invité à étudier le cas où l'arbre binaire de recherche contient des éléments égaux). On a choisi ici de décrire l'algorithme de suppression en utilisant la première solution.

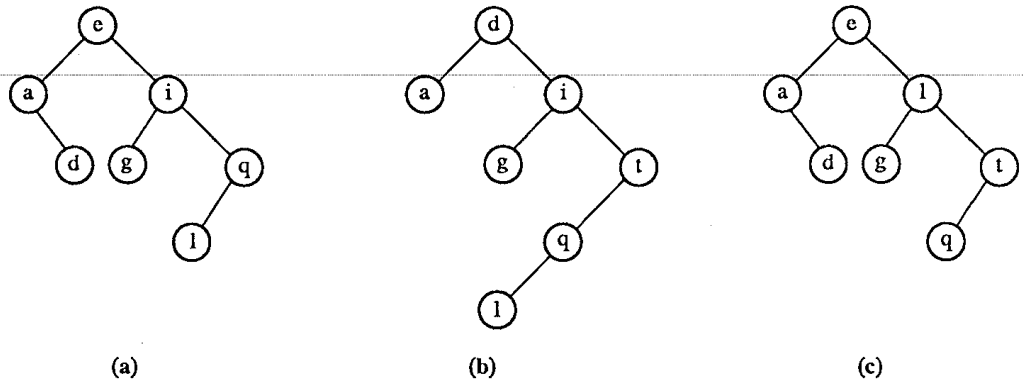


Figure 4. Suppressions dans un arbre binaire de recherche.

3.1. Spécification de la suppression

L'opération de suppression d'un élément dans un arbre binaire de recherche utilise l'opération *max*, qui renvoie l'élément maximal d'un arbre binaire de recherche, et l'opération *dmax* qui renvoie l'arbre privé de cet élément.

supprimer : Elément × Arbre → Arbre
max : Arbre → Elément
dmax : Arbre → Arbre

Si G et D sont des variables de sorte Arbre, x et r des variables de sorte Elément, on a les préconditions et axiomes suivants :

préconditions

max(G) **est-défini-ssi** $G \neq$ *arbre-vide*
dmax(G) **est-défini-ssi** $G \neq$ *arbre-vide*

axiomes

supprimer(x , *arbre-vide*) = *arbre-vide*
$x = r \Rightarrow$ *supprimer*(x, $< r, G,$ *arbre-vide*$>$) = G
$x = r$ & $D \neq$ *arbre-vide*
$\Rightarrow$ *supprimer*(x, $< r,$ *arbre-vide*, $D >$) = D
$x = r$ & $G \neq$ *arbre-vide* & $D \neq$ *arbre-vide*
$\Rightarrow$ *supprimer*($x, < r, G, D >$) = <*max*(G), *dmax*(G), $D >$
$x < r \Rightarrow$ *supprimer*($x, < r, G, D >$) = $< r,$ *supprimer*(x, G), $D >$
$x > r \Rightarrow$ *supprimer*($x, < r, G, D >$) = $< r, G,$ *supprimer*(x, D) $>$

max (< *r*, *G*, *arbre-vide*>) = *r*
dmax (< *r*, *G*, *arbre-vide*>) = *G*
D ≠ *arbre-vide* ⇒ *max*(< *r*, *G*, *D* >) = *max*(*D*)
D ≠ *arbre-vide* ⇒ *dmax*(< *r*, *G*, *D* >) =< *r*, *G*, *dmax*(*D*) >

3.2. Algorithme de suppression

La procédure *supprimerabr* utilise la procédure *supmax* de suppression du maximum dans un arbre binaire de recherche.

```
procedure supmax(var MAX : Elément; var A : ARBRE);
{cet algorithme supprime le nœud contenant l'élément maximal dans un
arbre binaire de recherche A non vide ; il donne pour résultats l'élément
maximal, et l'arbre binaire de recherche A privé du nœud contenant cet
élément}
begin
   if A↑.d =nil then begin MAX := A↑.val; A := A↑.g end
               else supmax(MAX, A↑.d)
end supmax;

procedure supprimerabr(X : Elément; var A : ARBRE);
{cet algorithme supprime un nœud contenant X dans l'arbre binaire de
recherche A ; il donne pour résultat A si X n'est pas dans A, et sinon
A privé de X ; il utilise la procédure de suppression du maximum dans
un arbre binaire de recherche}
var MAX : Elément;
begin
   if A<>nil then
        if X < A↑.val then supprimerabr(X, A↑.g)
        else if X > A↑.val then supprimerabr(X, A↑.d)
             else {A↑.val = X}
                  if A↑.g =nil then A := A↑.d
                  else if A↑.d =nil then A := A↑.g
                       else {A↑.d <> nil et A↑.g <> nil}
                            begin
                            supmax(MAX, A↑.g);
                            A↑.val := MAX
                            end
end supprimerabr;
```

4. Analyse du nombre de comparaisons

On analyse maintenant les différents algorithmes présentés dans le paragraphe précédent, dans le cas où *tous les éléments sont distincts.* On montre que leur

complexité en temps est logarithmique en moyenne, alors qu'elle est linéaire dans le pire des cas.

Les algorithmes de recherche, adjonction et suppression dans un arbre binaire de recherche opèrent par comparaisons : la comparaison entre deux éléments est l'opération fondamentale pour mesurer la complexité en temps de ces algorithmes. Or, le nombre de comparaisons faites par chacun des algorithmes peut être lu directement sur l'arbre binaire de recherche.

Pour l'algorithme de recherche :

- dans le cas d'une recherche positive terminée sur le nœud ν, le nombre de comparaisons est $2\,prof(\nu)+1$, où $prof(\nu)$ est la profondeur du nœud ν dans l'arbre (cf. chapitre 7 pour la définition de la profondeur d'un nœud dans un arbre) ;
- dans le cas d'une recherche négative terminée après le nœud ν, le nombre de comparaisons est $2(prof(\nu)+1)$.

Dans l'algorithme d'adjonction aux feuilles, il faut diviser par deux le nombre de comparaisons pour une recherche négative (il y a un test par nœud au lieu de deux).

Pour une adjonction à la racine, l'algorithme compare l'élément à insérer uniquement aux contenus des nœuds situés sur son chemin de recherche dans l'arbre ; le nombre de comparaisons effectuées est donc exactement le même que pour l'algorithme d'adjonction aux feuilles.

Pour la suppression, le nombre de comparaisons entre éléments est le même que pour la recherche de l'élément à supprimer dans l'arbre.

Remarquons que si l'on veut aussi tenir compte du temps de recherche de l'élément qui remplace l'élément supprimé, il faut ajouter la complexité de la procédure *supmax* ; dans cette procédure, le *test d'arbre vide* est l'opération fondamentale. (Dans le cas de la recherche et de l'adjonction, le nombre de tests d'arbre vide n'est pas significatif, car il est égal au nombre de comparaisons d'éléments.)

L'analyse des algorithmes de recherche, adjonction et suppression, se ramène donc à l'étude de la profondeur des nœuds dans les arbres binaires de recherche.

4.1. Cas extrêmes d'arbres binaires de recherche

On a vu au chapitre 7 que la profondeur et la profondeur moyenne d'un arbre binaire de taille n sont des quantités toujours comprises entre n et $\log_2 n$.

Dans un arbre binaire de recherche *dégénéré*, la recherche d'un élément nécessite en moyenne et dans le cas le pire, un nombre de comparaisons *linéaire* par rapport à la taille de l'arbre : par exemple, si l'on construit un arbre binaire de recherche par adjonctions successives, en insérant les éléments par ordre croissant, on obtient

un arbre dégénéré comme celui de la figure 5.a dans lequel la recherche du $n^{ième}$ élément inséré nécessite $2n - 1$ comparaisons, et la recherche d'un élément quelconque de l'arbre nécessite en moyenne un nombre de comparaisons égal à

$$\frac{1}{n} \sum_{1 \leq i \leq n} (2i - 1) = n$$

On est dans la même situation si l'on insère les éléments dans un «ordre» mal choisi, par exemple dans l'ordre f, a, e, b, d, c, comme à la figure 5.b.

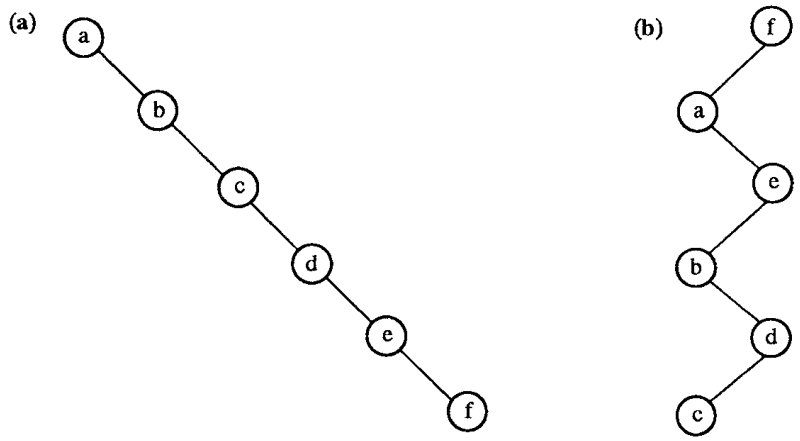


Figure 5. Arbres binaires de recherche dégénérés.

En revanche, si l'arbre binaire de recherche est «bien équilibré» (toutes les feuilles sont situées sur deux niveaux seulement), la recherche d'un élément est, en moyenne et dans le pire des cas, logarithmique par rapport à la taille de l'arbre (cf. chapitre 9, analyse de la procédure *dicho*).

4.2. Cas moyen d'arbres binaires de recherche

On montre ici que le nombre de comparaisons pour la recherche, l'adjonction ou la suppression d'un élément dans un arbre binaire de recherche de taille n est en moyenne de l'ordre de $\log n$. Il faut pour cela calculer la profondeur moyenne d'un nœud dans un arbre binaire de recherche quelconque. On s'intéresse donc à une double moyenne : moyenne sur tous les nœuds d'un arbre, qui est un arbre moyen parmi tous les arbres binaires de recherche.

4.2.1. Notion de moyenne

Les arbres binaires de recherche sont-ils en moyenne plutôt dégénérés ou plutôt équilibrés ? Il faut ici s'arrêter sur la notion de *moyenne* sur l'ensemble des arbres binaires de recherche. On considère que les arbres binaires de recherche sont obtenus uniquement par adjonctions successives aux feuilles de n éléments distincts, et

on fait l'hypothèse que les $n!$ ordres d'adjonction possibles de ces éléments sont équiprobables.

A chacun de ces ordres correspond un arbre binaire de recherche. Des ordres différents peuvent donner des arbres binaires de recherche identiques. Par exemple, les adjonctions successives de 2, 3, 1 ou de 2, 1, 3 donnent l'unique arbre de la figure 6.

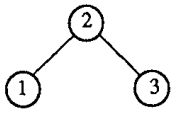


Figure 6

On a donc un ensemble avec répétitions de $n!$ arbres binaires de recherche qui sont équiprobables. On appelle ABR_n cet ensemble. Par exemple, pour $n = 3$, on a les arbres de la figure 7.

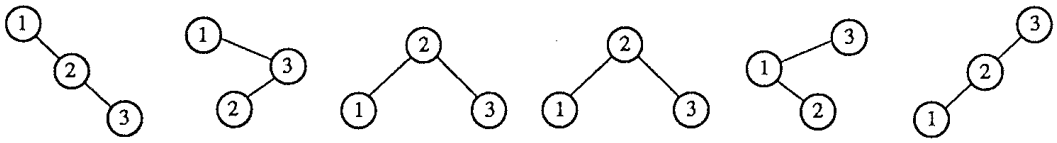


Figure 7

L'analyse qui suit montre qu'un arbre binaire de recherche est en général bien équilibré : la moyenne des profondeurs moyennes, interne et externe, sur les arbres de ABR_n est en $\Theta(\log n)$.

4.2.2. Complétion locale d'un arbre binaire de recherche

Revenons à la notion de construction d'un arbre binaire de recherche par adjonctions successives aux feuilles. Dans un arbre binaire T contenant n éléments, il y a $n+1$ possibilités d'adjonction aux feuilles : on peut matérialiser ces $n+1$ possibilités en complétant l'arbre T par des feuilles carrées , de manière à ce que tout nœud de T ait deux fils. La figure 8 montre un arbre binaire de recherche T et son complété T_c (cf. chapitre 7 pour la définition de la complétion locale d'un arbre binaire, et les propriétés associées). L'arbre T_c a n nœuds internes et $n+1$ feuilles carrées.

Si T est un arbre binaire de recherche, chaque feuille carrée du complété T_c (à l'exception de la première et de la dernière) est située, en lecture symétrique, entre deux nœuds internes n_i et n_j, et représente l'endroit où se termine la recherche négative d'un élément compris entre les contenus de n_i et de n_j. La première (resp. dernière) feuille marque l'endroit où se termine la recherche négative de tout élément inférieur (resp. supérieur) à tous les éléments de l'arbre T.

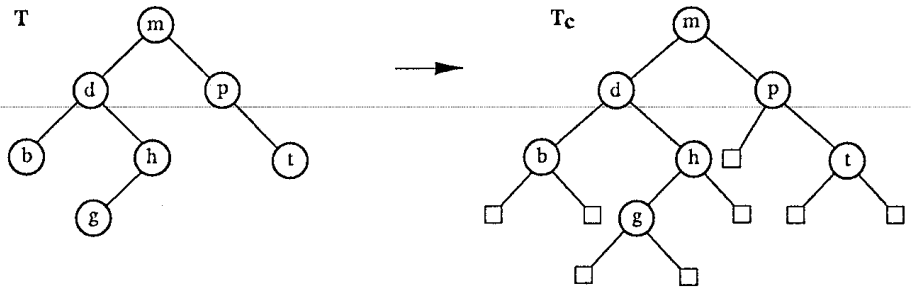


Figure 8. Complétion locale d'un arbre binaire de recherche.

4.2.3. Profondeur moyenne de recherche dans les arbres de ABR_n

Par définition, les arbres binaires de recherche contenus dans ABR_n sont tous obtenus par adjonction aux feuilles d'un élément dans un arbre binaire de recherche de $n-1$ éléments, lui-même obtenu uniquement par des adjonctions aux feuilles. Tous les ordres d'adjonction étant équiprobables, ce dernier élément peut, avec une même probabilité $1/n$, être le plus petit, le $p^{ième}$ plus petit ($p = 2, \ldots, n-1$), ou le plus grand. L'adjonction peut donc se faire en chacune des n places possibles avec une probabilité $1/n$.

Soit T un arbre de ABR_n et T_c son arbre complété. L'arbre T_c a $n+1$ feuilles. On définit la profondeur moyenne d'adjonction (ou de recherche négative) comme la profondeur moyenne externe de T_c : $PE(T_c) = \frac{1}{n+1} \cdot LCE(T_c)$.

On considère à présent PE_n, la profondeur moyenne d'adjonction (ou de recherche négative) dans un arbre de ABR_n, les $n!$ arbres binaires de recherche étant équiprobables :

$$PE_n = \frac{1}{n!} \sum_{T \in ABR_n} PE(T_c) = \frac{1}{n!} \sum_{T \in ABR_n} \frac{1}{n+1} \cdot LCE(T_c).$$

L'étude de $\sum_{T \in ABR_n} LCE(T_c)$ va permettre d'établir une relation de récurrence sur PE_n.

Soit T' un arbre binaire de recherche de $(n-1)$ éléments, et soit f une feuille de son complété T'_c sur laquelle est faite une adjonction. Soit T le résultat de l'adjonction (figure 9) :

$$\begin{aligned} LCE(T_c) &= LCE(T'_c) - prof(f) + 2(prof(f) + 1) \\ &= LCE(T'_c) + prof(f) + 2 \end{aligned}$$

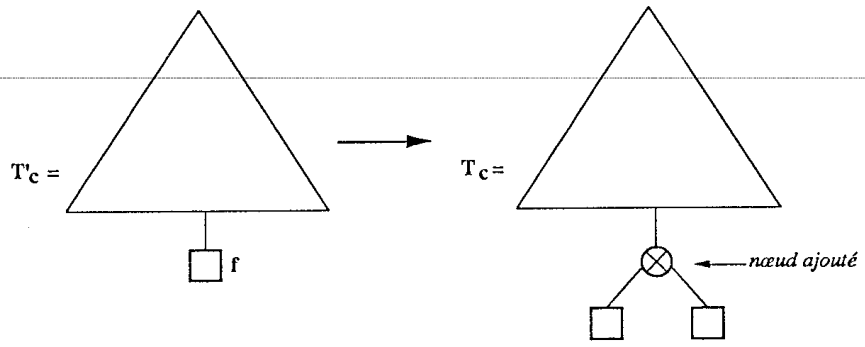


Figure 9

Par définition, chaque arbre de ABR_n provient d'un arbre de ABR_{n-1} dans lequel on a fait une adjonction aux feuilles, d'où

$$\sum_{T \in ABR_n} LCE(T_c) = \sum_{T' \in ABR_{n-1}} \left(\sum_{f \text{ feuille de } T'_c} (LCE(T'_c) + prof(f) + 2) \right)$$

or

$$\sum_{T' \in ABR_{n-1}} \left(\sum_{f \text{ feuille de } T'_c} LCE(T'_c) \right) = n \cdot \sum_{T' \in ABR_{n-1}} LCE(T'_c)$$

et

$$\sum_{T' \in ABR_{n-1}} \left(\sum_{f \text{ feuille de } T'_c} prof(f) \right) = \sum_{T' \in ABR_{n-1}} LCE(T'_c)$$

d'où

$$\sum_{T \in ABR_n} LCE(T_c) = 2n \cdot (n-1)! + (n+1) \sum_{T' \in ABR_{n-1}} LCE(T'_c)$$

En utilisant la définition de PE_n, $PE_n = \frac{1}{n!} \sum_{T \in ABR_n} \frac{1}{n+1} \cdot LCE(T_c)$, on obtient :

$$PE_n = \left(\frac{1}{n!} \sum_{T' \in ABR_{n-1}} LCE(T'_c) \right) + \frac{2}{n+1}, \text{ c'est-à-dire } PE_n = PE_{n-1} + \frac{2}{n+1}.$$

Cette relation de récurrence, avec la condition initiale $PE_1 = 1$ (pour l'arbre binaire de recherche contenant un seul élément), admet pour solution :

$$PE_n = 2H_{n+1} - 2,$$

où $H_{n+1} = \sum_{1 \leq i \leq n+1} \frac{1}{i}$ est le $(n+1)^{ième}$ nombre harmonique.

La profondeur moyenne PI_n d'un nœud interne dans un arbre de ABR_n, c'est-à-dire la *profondeur moyenne d'une recherche positive*, se calcule de manière semblable :

$$PI_n = \frac{1}{n!} \sum_{T \in ABR_n} \frac{1}{n} \cdot LCI(T_c)$$

or, puisque T_c est localement complet, $LCI(T_c) = LCE(T_c) - 2n$, d'où

$$PI_n = \left(1 + \frac{1}{n}\right) PE_n - 2 = 2\left(1 + \frac{1}{n}\right) H_{n+1} - \frac{2}{n} - 4 = 2\left(1 + \frac{1}{n}\right) H_n - 4$$

La proposition suivante résume les résultats obtenus.

Proposition 1 : Dans un arbre binaire de recherche obtenu par adjonctions successives de n éléments distincts, les $n!$ permutations possibles des éléments étant équiprobables, les nœuds externes et internes sont en moyenne à une profondeur de l'ordre de $\log n$. L'expression des profondeurs moyennes externes et internes est plus précisement :

$$PE_n = 2H_{n+1} - 2 \quad \text{et} \quad PI_n = 2\left(1 + \frac{1}{n}\right) H_n - 4\,.$$

4.3. Conclusion

On a donc montré que la complexité des algorithmes de recherche, adjonction et suppression dans un arbre binaire de recherche est en moyenne logarithmique, alors qu'elle est linéaire dans le cas le pire où l'arbre est dégénéré.

Notons $\text{Max}_{\text{rech+}}(n)$ (resp. $\text{Max}_{\text{rech-}}$, Max_{adj}, Max_{supp}) le nombre maximum de comparaisons pour une recherche positive (resp. pour une recherche négative, pour une adjonction, pour une suppression) dans un arbre binaire de recherche contenant n éléments.

La hauteur de l'arbre dégénéré étant $n - 1$, ces quatre quantités sont en $\Theta(n)$:

$\text{Max}_{\text{rech+}}(n) = \text{Max}_{\text{supp}}(n) = 2n - 1$
$\text{Max}_{\text{rech-}}(n) = 2n$
$\text{Max}_{\text{adj}}(n) = n.$

Le facteur multiplicatif deux pour la recherche et la suppression vient du fait qu'il y a dans ces deux algorithmes deux tests par nœud, et il n'y a qu'un test par nœud dans les algorithmes d'adjonction.

Notons aussi $\text{Moy}_{\text{rech+}}(n)$ (resp. $\text{Moy}_{\text{rech-}}$, Moy_{adj}, Moy_{supp}) le nombre moyen de comparaisons pour une recherche positive (resp. pour une recherche négative,

pour une adjonction, pour une suppression) dans un arbre binaire de recherche quelconque, *i.e.* obtenu par adjonctions successives de n éléments distincts, les $n!$ ordres d'adjonction étant équiprobables.

D'après la proposition 1 :

$$\text{Moy}_{\text{rech+}}(n) = \text{Moy}_{\text{supp}}(n) = 2PI_n + 1$$
$$\text{Moy}_{\text{rech-}}(n) = 2PE_n$$
$$\text{Moy}_{\text{adj}}(n) = PE_n.$$

Ces quatre quantités sont donc en $\Theta(\log n)$.

Les arbres binaires de recherche permettent de rechercher, ajouter et supprimer un élément d'une collection avec une complexité qui est en moyenne logarithmique. Cependant, la complexité au pire est linéaire. Le chapitre suivant présente des structures arborescentes pour lesquelles la complexité de ces opérations reste logarithmique dans le cas le pire.

Exercices

1. Ecrire un algorithme récursif et un algorithme itératif qui impriment la liste des éléments d'un arbre binaire de recherche en ordre croissant.

2. a) La figure 1.a représente un arbre binaire de recherche résultant de l'adjonction successive aux feuilles, à partir de l'arbre vide, des éléments e, i, g, a, t, d, q, l. Donner d'autres ordres d'entrée des éléments qui construisent le même arbre. Combien y a-t-il de permutations de l'ensemble $\{e, i, g, a, t, d, q, l\}$ qui donnent par construction l'arbre de la figure 1.a ?

b) Plus généralement, étant donné un arbre binaire de recherche T de taille n, montrer que le nombre de permutations des éléments de T qui donnent exactement T dans la construction par adjonctions successives aux feuilles, est $n!$ divisé par le produit des tailles de tous les sous-arbres de T (voir la définition de *sous-arbre* d'un arbre binaire au chapitre 7).

3. Donner une version itérative de l'algorithme d'adjonction dans un arbre binaire de recherche, en utilisant un pointeur père du pointeur courant.

4. a) Ecrire une procédure itérative de coupure d'un arbre binaire de recherche.

b) Ecrire une procédure récursive de coupure qui s'arrête dès qu'on rencontre l'élément selon lequel se fait la coupure.

5. L'arbre binaire de recherche U est la *fusion* des arbres binaires de recherche A et B, si U contient tous les éléments de A et tous les éléments de B. En utilisant la procédure de coupure, écrire une procédure de fusion de deux arbres binaires de recherche.

6. On s'intéresse ici aux arbres binaires de recherche pouvant contenir des éléments égaux de type entier que l'on veut pouvoir distinguer selon leur ordre d'entrée dans l'arbre, appelé *occurrence*.

Les opérations de base auxquelles on s'intéresse sont :
- l'ajout d'un élément,
- la suppression de la $k^{ième}$ occurrence d'un élément,
- la recherche de la $k^{ième}$ occurrence d'un élément.

On a donc les déclarations Pascal suivantes :
- **procedure** *chercher*(A : ARBRE; n, k : integer; **var** B : ARBRE);
- **procedure** *ajouter*(**var** A : ARBRE; n : integer);
- **procedure** *supprimer*(**var** A : ARBRE; n, k : integer);

où k représente l'occurrence de l'élément n, et B l'adresse de l'élément recherché ou *nil.*

On désire que les éléments égaux soient consécutifs sur un chemin de l'arbre issu de la racine. Ecrire les procédures ci-dessus de façon à ce que cette contrainte soit respectée (donner une version récursive et une version itérative).

7. Comme dans l'exercice précédent, on considère des arbres binaires de recherche avec éléments égaux distinguables par leur ordre d'arrivée, et on veut maintenant que la lecture en ordre symétrique de l'arbre binaire de recherche fournisse les éléments égaux *par ordre d'occurrences croissantes.*

a) Ecrire une procédure d'adjonction (on pourra faire l'adjonction à la racine ou aux feuilles).

b) Montrer que la lecture en ordre symétrique des arbres binaires de recherche construits par ajouts successifs à la racine fournit bien les éléments égaux par ordre d'occurrences croissantes.

c) Caractériser les nœuds correspondant à des éléments égaux.

d) Ecrire des procédures, récursive et itérative, de recherche de la $k^{ième}$ occurrence d'un élément.

e) Faire de même pour l'opération de suppression de la $k^{ième}$ occurrence d'un élément.

8. a) Deux arbres binaires de recherche sont dits *équivalents* s'ils contiennent exactement les mêmes éléments. Ecrire une fonction Pascal qui teste si deux arbres binaires de recherche sont équivalents.

b) Un arbre binaire de recherche A est *contenu* dans un arbre binaire de recherche B si tous les éléments de A sont contenus dans B. Ecrire une fonction Pascal qui teste si un arbre binaire de recherche est contenu dans un autre.

c) Un arbre binaire de recherche A est dit *de domaine plus petit* qu'un arbre binaire de recherche B si le plus petit élément de A est supérieur ou égal au plus

petit élément de B, et si le plus grand élément de A est inférieur ou égal au plus grand élément de B. Ecrire une fonction Pascal, non récursive, qui teste si un arbre binaire de recherche est de domaine plus petit qu'un autre.

9. Dans un arbre binaire de recherche construit par adjonctions successives aux feuilles, la profondeur d'un élément n'est jamais modifiée par les adjonctions ultérieures.

a) Déduire de cette remarque que :

$$PI_n = \frac{1}{n} \sum_{i=1,\ldots,n} PE_{i-1}, \text{ avec } PE_0 = 0$$

b) Retrouver la valeur de PI_n à partir de la formule précédente et de la valeur de PE_n.

10. On supprime consécutivement deux éléments dans un arbre binaire de recherche avec la procédure *supprimerabr*. Montrer que l'arbre résultant ne dépend pas de l'ordre dans lequel on a effectué les suppressions.

11. Donner une version itérative de la procédure *supmax*.

12. Analyser le nombre de comparaisons effectuées par la procédure *supmax* sur un arbre binaire de recherche de n éléments :

a) dans le cas le pire,

b) en moyenne : on pourra étudier, en suivant la même démarche que pour l'étude de la longueur moyenne de cheminement, *la longueur moyenne* du bord droit d'un arbre binaire de recherche de n éléments $Bd(n)$, qui s'exprime par

$$Bd(n) = \frac{1}{n!} \sum_{T \in ABR_n} Bd(T)$$

Rappel : le *bord droit* $Bd(T)$ d'un arbre binaire T est le chemin obtenu à partir de la racine de T en ne suivant que des liens droits.

13. On considère la méthode suivante de *recherche auto-adaptative* dans un arbre binaire de recherche qui contient les éléments $e_1, \ldots, e_n$, avec $e_1 \leq \ldots \leq e_n$.

Après toute recherche de l'élément e, si e est la racine on ne fait rien, sinon, si e est fils gauche de son père p on effectue la rotation de la figure 10, et si e est fils droit de p on effectue la rotation symétrique.

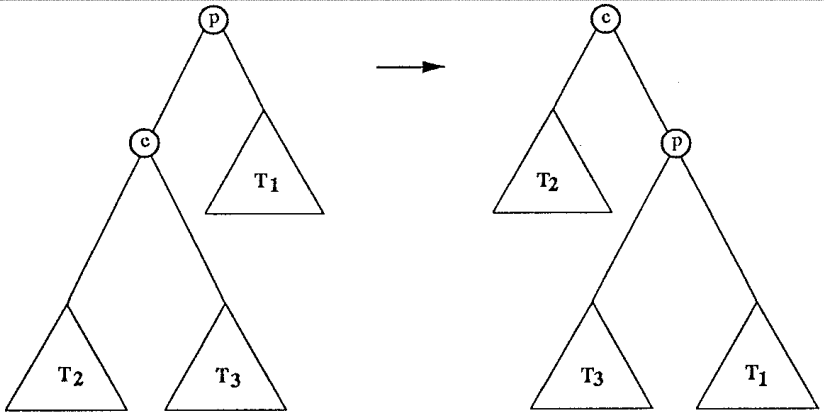


Figure 10

a) Programmer cette première méthode.

b) Programmer une autre méthode de recherche auto-adaptative, dans laquelle tout élément cherché est remonté à la racine de l'arbre par une suite de rotations.

c) Avec cette seconde méthode montrer que si e_i est inférieur à e_j, pour que e_i soit un ancêtre de e_j il faut et il suffit que

- soit e_i était au départ un ancêtre de e_j et il n'y a jamais eu de recherche de l'un quelconque des éléments $e_{i+1}, \ldots, e_j$,
- soit il n'y a pas eu de recherche de l'un quelconque des éléments $e_{i+1}, \ldots, e_j$ depuis la dernière recherche de e_i.

Quand e_j est-il un ancêtre de e_i ?

14. On a proposé dans ce chapitre un algorithme d'adjonction aux feuilles et un algorithme d'adjonction à la racine.

Ecrire un algorithme d'adjonction d'un élément à une profondeur donnée p dans un arbre binaire de recherche de hauteur supérieure à p.

15. On désire construire par adjonctions successives aux feuilles, un arbre binaire de recherche, et on connaît les probabilités de recherche des éléments. Dans quel ordre de leurs probabilités de recherche faut-il ajouter les éléments pour optimiser le temps moyen d'une recherche ?

16. Un *arbre de recherche bidimensionnel* est un arbre binaire dont les nœuds sont des enregistrements ayant deux attributs, et tel que pour une recherche ou une adjonction la discrimination se fait selon le 1^{er} champ au premier niveau de l'arbre (*i.e.* à la racine) puis selon le 2^{e} champ au 2^{e} niveau de l'arbre, puis de nouveau selon le 1^{er} champ au 3^{e} niveau, etc.

Par exemple, l'insertion successive des nœuds : (7;2), (3;4), (8;7), (9;6), (4;5), (5;2), (6;1) donne l'arbre de recherche bidimensionnel de la figure 11.

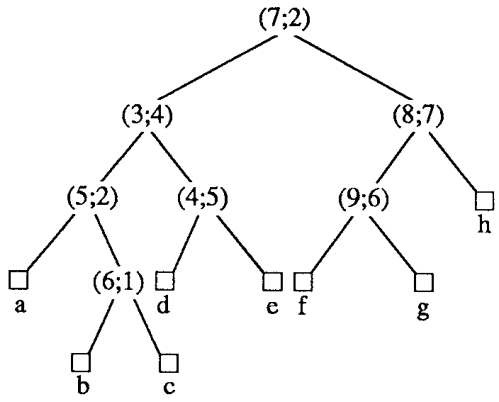


Figure 11. Arbre de recherche bidimensionnel.

a) Ecrire un algorithme d'adjonction d'un enregistrement dans un arbre de recherche bidimensionnel.

b) Ecrire un algorithme de recherche des nœuds de l'arbre dont le premier champ est minimal.

c) Ecrire un algorithme qui recherche *tous* les nœuds dont le premier (resp. le deuxième) champ contient un élément donné.

Lectures conseillées pour le chapitre 10

Knuth, *The Art of Computer Programming*, Vol. 3 : *Sorting and Searching*, Addison-Wesley, 1973.

Chapitre 11
Arbres équilibrés

On a montré, dans le chapitre précédent, que les opérations de recherche, adjonction et suppression dans un arbre binaire de recherche de n éléments se font en moyenne en $\Theta(\log n)$ comparaisons; mais dans le cas le pire, la complexité de ces opérations est en $\Theta(n)$: un arbre binaire de recherche peut en effet dégénérer en une liste. L'arbre binaire de recherche «idéal», pour lequel une recherche nécessite *au pire* $\log n$ comparaisons est un arbre complètement équilibré, *i.e.* un arbre dont les feuilles sont situées au plus sur deux niveaux. Mais un tel arbre peut être complètement désorganisé par une adjonction (ou une suppression), et c'est alors le coût de la réorganisation qui est prépondérant : l'arbre de la figure 1.a est complètement équilibré, mais il perd cette propriété après adjonction de l'élément 1; on a reconstruit dans la figure 1.b un arbre complètement équilibré contenant 1 : tous les éléments de cet arbre ont changé de place; ainsi la procédure de réorganisation après adjonction dans un arbre complètement équilibré peut être de complexité $\Theta(n)$.

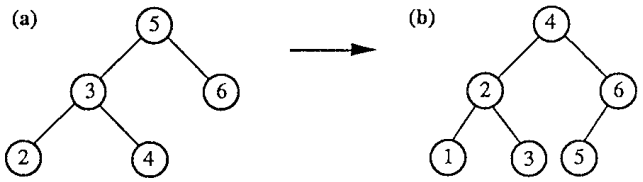


Figure 1. Réorganisation après adjonction de 1.

Pour que la réorganisation de l'arbre ne soit pas trop coûteuse, il faut assouplir les contraintes sur la forme des arbres de recherche considérés : par exemple, autoriser un léger déséquilibre en hauteur, ou autoriser les nœuds à avoir un nombre variable de fils.

On étudie dans ce chapitre diverses classes d'***arbres de recherche équilibrés***. Ces arbres sont dits équilibrés car leur hauteur est toujours une fonction logarithmique de leur nombre de nœuds; de plus, lorsqu'un arbre est «déséquilibré» après une adjonction ou une suppression, il existe des algorithmes de rééquilibrage, spécifiques à la classe à laquelle appartient l'arbre, dont la complexité est aussi logarithmique dans tous les cas. Ainsi dans chaque classe, les opérations de recherche, adjonction

et suppression sur les arbres de n éléments sont de *complexité $O(\log n)$ dans tous les cas.*

1. Rotations

Beaucoup d'algorithmes de rééquilibrage utilisent des transformations locales appelées ***rotations*** : il s'agit de faire basculer vers la droite (resp. gauche) un sous-arbre qui est trop déséquilibré vers la gauche (resp. droite). Il y a quatre types de rotation : les *rotations simples*, à droite (en abrégé *rd*, voir figure 2) et à gauche (en abrégé rg, voir figure 3), et les *rotations doubles*, rotation gauche-droite (en abrégé *rgd*, voir figure 4) et rotation droite-gauche (en abrégé *rdg*, voir figure 5).

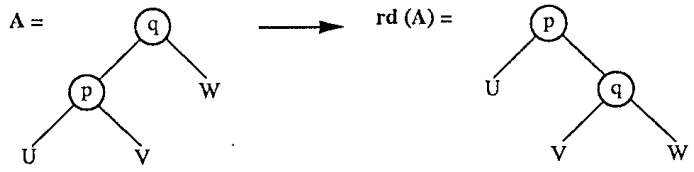


Figure 2. Rotation droite.

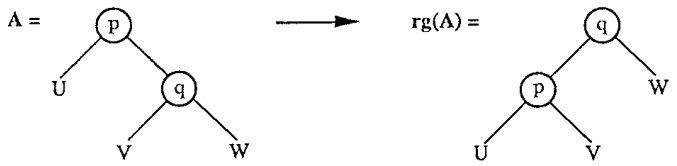


Figure 3. Rotation gauche.

Il est aisé de voir que la rotation gauche-droite sur l'arbre A (resp. droite-gauche) est composée d'une rotation gauche (resp. droite) sur le sous-arbre gauche (resp. droit) de A, suivie d'une rotation droite (resp. gauche) sur A, d'où le nom de double rotation.

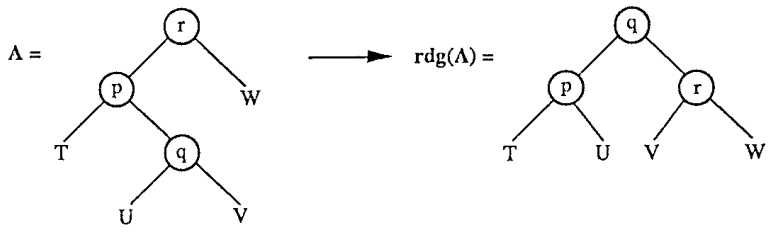


Figure 4. Rotation gauche-droite.

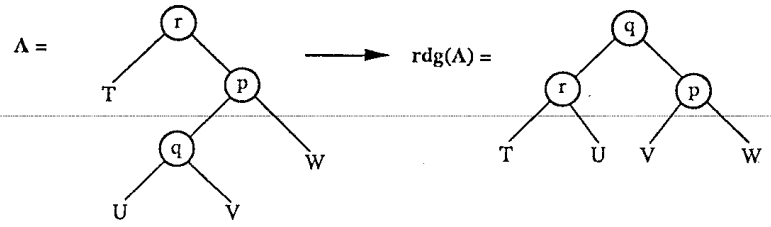


Figure 5. Rotation droite-gauche.

Ces opérations sont définies de la façon suivante :

rg : Arbre → Arbre
rd : Arbre → Arbre
rgd : Arbre → Arbre
rdg : Arbre → Arbre

Pour toutes variables A, T, U, V, W de sorte Arbre, et pour toutes variables p, q, r de sorte Nœud, on a les préconditions et les axiomes suivants :

préconditions

$rg(A)$ **est-défini-ssi** $A \neq$ *arbre-vide* & $d(A) \neq$ *arbre-vide*
$rd(A)$ **est-défini-ssi** $A \neq$ *arbre-vide* & $g(A) \neq$ *arbre-vide*
$rgd(A)$ **est-défini-ssi**
$A \neq$ *arbre-vide* & $g(A) \neq$ *arbre-vide* & $d(g(A)) \neq$ *arbre-vide*
$rdg(A)$ **est-défini-ssi**
$A \neq$ *arbre-vide* & $d(A) \neq$ *arbre-vide* & $g(d(A)) \neq$ *arbre-vide*

axiomes

$rg(< p, U, < q, V, W >>) = < q, < p, U, V >, W >$
$rd(< q, < p, U, V >, W >) = < p, U, < q, V, W >>$
$rgd(< r, < p, T, < q, U, V >>, W >) = < q, < p, T, U >, < r, V, W >>$
$rdg(< r, T, < p, < q, U, V >, W >>) = < q, < r, T, U >, < p, V, W >>$

Les procédures ci-dessous réalisent ces transformations : puisqu'une rotation gauche-droite sur A est composée d'une rotation gauche sur le sous-arbre gauche de A suivie d'une rotation droite sur A, la procédure RGD fait appel aux procédures RD et RG. Et il en est de même pour la procédure RDG. ARBRE est le type Pascal arbre binaire représenté à l'aide de pointeurs (voir chapitre 7).

```
procedure RG (var A : ARBRE);
var Aux : ARBRE;
begin Aux := A↑.d; A↑.d := Aux↑.g;
      Aux↑.g := A; A := Aux
end RG;
```

```
procedure RD (var A : ARBRE);
var Aux : ARBRE;
begin Aux := A↑.g; A↑.g := Aux↑.d;
Aux↑.d := A; A := Aux
end RD;

procedure RGD (var A : ARBRE);
begin RG(A↑.g);
RD(A)
end RGD;

procedure RDG (var A : ARBRE);
begin RD(A↑.d);
RG(A)
end RDG;
```

Remarque : Les opérations de rotation conservent la propriété d'arbre binaire de recherche. Il est immédiat de constater que la lecture symétrique de $rd(A)$(resp.$rg(A), rgd(A), rdg(A)$) donne le même résultat que la lecture symétrique de A.

2. Arbres AVL

Historiquement la première classe d'arbres équilibrés a été introduite dans les années 60 par Adelson-Velskii et Landis (d'où le nom d'AVL). En chaque nœud d'un arbre AVL, la différence des hauteurs des sous-arbres gauche et droit est égale au plus à 1 en valeur absolue. On verra que cette condition implique que la hauteur totale d'un arbre AVL est toujours inférieure à une fois et demie la hauteur d'un arbre «complètement équilibré» contenant le même nombre de nœuds. De plus, il existe des algorithmes de rééquilibrage après adjonction ou suppression qui maintiennent la propriété d'AVL en effectuant une suite de rotations sur un chemin de la racine à une feuille de l'arbre. Ainsi les opérations de recherche, adjonction et suppression ont une complexité qui reste logarithmique dans le pire des cas.

2.1. Arbres H-équilibrés

Définition : On dit qu'un arbre binaire est ***H-équilibré*** si en tout nœud de l'arbre, les hauteurs des sous-arbres gauche et droit diffèrent au plus de 1.

Soit la fonction déséquilibre sur les arbres binaires, définie récursivement par :

déséquilibre(*arbre-vide*) = 0, et
déséquilibre($< o, G, D >$) = *hauteur*(G) - *hauteur*(D)

Alors un arbre T est H-équilibré si pour tout sous-arbre S de T on a :

$$déséquilibre(S) \in \{-1, 0, 1\}$$

Exemples : Les arbres de la figure 6 sont H-équilibrés, mais l'arbre de la figure 7 ne l'est pas ; on a indiqué en chaque nœud la valeur de la fonction de déséquilibre.

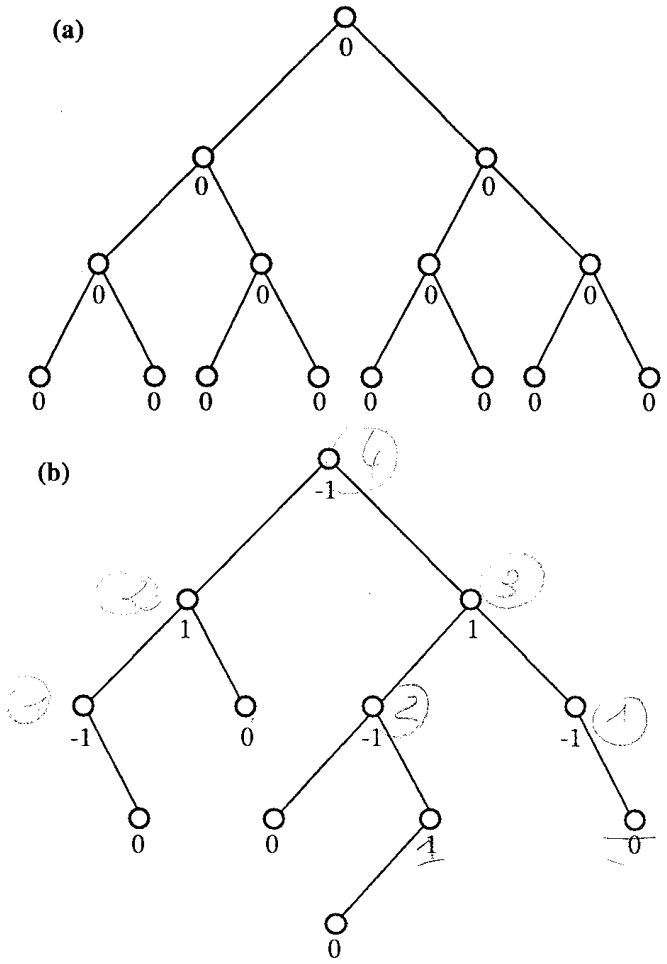


Figure 6. Arbres H-équilibrés.

La propriété suivante montre que les contraintes de déséquilibre en chaque nœud permettent de limiter la hauteur totale de l'arbre : la hauteur d'un arbre H-équilibré de n nœuds est toujours de l'ordre de $\log n$.

Propriété : Tout arbre H-équilibré ayant n nœuds a une hauteur h vérifiant :

$$\log_2(n+1) \leq h+1 < 1,44 \log_2(n+2)$$

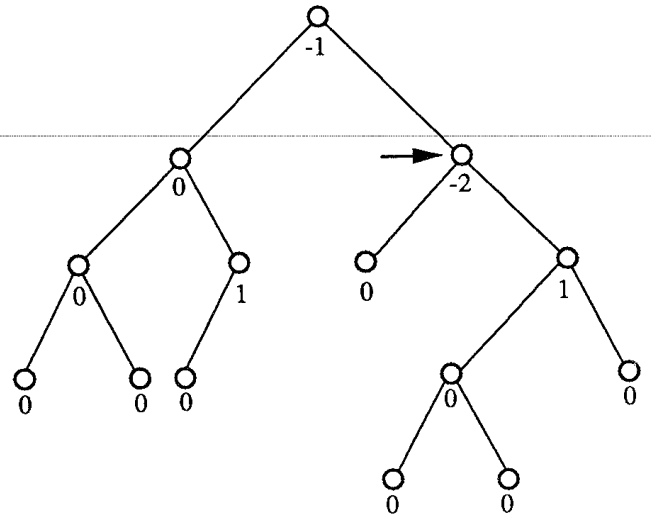


Figure 7. Arbre non H-équilibré.

Preuve :

a) Pour une hauteur donnée h, l'arbre H-équilibré contenant le plus de nœuds est celui dont tous les niveaux sont complètement remplis (déséquilibre 0 en chaque nœud). C'est un arbre complet (cf. figure 6.a), qui a donc $2^{h+1} - 1$ nœuds. Pour tout arbre H-équilibré ayant n nœuds et de hauteur h, on a $n \leq 2^{h+1} - 1$, et donc $h + 1 \geq \log_2(n + 1)$.

b) à l'opposé, pour une hauteur donnée $h \geq 1$, les arbres H-équilibrés de hauteur h contenant le moins de nœuds sont ceux pour lesquels la fonction de déséquilibre vaut 1 ou -1 dans tous les nœuds : si l'on enlève une feuille quelconque dans un tel arbre, ou bien l'arbre résultat n'est plus H-équilibré, ou bien sa hauteur a diminué de 1 (cf. figure 6.b et exercices).

Soit $N(h)$ le nombre de nœuds d'un tel arbre ; un arbre H-équilibré de hauteur h ayant le moins de nœuds possibles est formé d'une racine et de deux sous-arbres de même nature, de hauteurs respectives $h - 1$ et $h - 2$; $N(h)$ vérifie donc la relation de récurrence $N(h) = 1 + N(h - 1) + N(h - 2)$, avec les conditions initiales $N(0) = 1$ et $N(1) = 2$. Posons $F(h) = N(h) + 1$, alors l'équation précédente se récrit $F(h) = F(h - 1) + F(h - 2)$.

Cette équation de récurrence de Fibonacci est étudiée au paragraphe 3 de l'annexe (exemple 7) ; avec les conditions initiales $F(0) = 2$ et $F(1) = 3$, elle a pour solution :

$$F(h) = \frac{1}{\sqrt{5}}(\phi^{h+3} - \overline{\phi}^{h+3}), \text{avec } \phi = \frac{1+\sqrt{5}}{2} \text{et } \overline{\phi} = \frac{1-\sqrt{5}}{2}$$

On a donc : $N(h)+1 = \frac{1}{\sqrt{5}}(\phi^{h+3} - \overline{\phi}^{h+3})$

D'où, pour tout arbre H-équilibré de n nœuds et de hauteur h :

$$n+1 \geq \frac{1}{\sqrt{5}}(\phi^{h+3} - \overline{\phi}^{h+3})$$

donc $n+1 > \frac{1}{\sqrt{5}}(\phi^{h+3} - 1)$ car $-1 < \overline{\phi}^{h+3} < 1$

Il en résulte que : $\phi^{h+3} < 1 + \sqrt{5}(1+n)$

d'où
$$\begin{aligned} h+3 &< \log_\phi(1+\sqrt{5}(1+n)) \\ &< \log_\phi(\sqrt{5}(n+2)) \\ &= \frac{\log_2(n+2))}{\log_2\phi} + \log_\phi\sqrt{5} \\ &< 2 + \frac{\log_2(n+2)}{\log_2\phi}, \text{ car } \sqrt{5} < \phi^2 \end{aligned}$$

or $(1/\log_2\phi) \simeq 1,44$ et donc $h+1 < 1,44\ \log_2(n+2)$. □

Cette proposition est très importante lorsqu'on utilise les arbres H-équilibrés comme arbres binaires de recherche. On définit un ***arbre AVL*** comme étant un arbre binaire de recherche H-équilibré.

2.2. Adjonction dans les AVL

Les AVL étant des arbres binaires de recherche, on peut utiliser les méthodes vues dans le chapitre précédent pour rechercher, ajouter ou supprimer un élément. D'après la propriété précédente, *la recherche dans un AVL contenant n éléments nécessite toujours $O(\log n)$ comparaisons.* Cependant une adjonction ou une suppression dans un AVL peuvent déséquilibrer l'arbre, comme le montre la figure 8. Ainsi, après avoir ajouté (aux feuilles) ou supprimé un élément dans un AVL, il faut éventuellement le rééquilibrer, tout en conservant la structure d'arbre binaire de recherche.

On étudie maintenant des algorithmes d'adjonction et de suppression dans les AVL, dont la complexité en temps est dans tous les cas en $O(\log n)$, rééquilibrages compris.

L'exemple qui suit va permettre de comprendre les différentes stratégies de rééquilibrage dans les AVL.

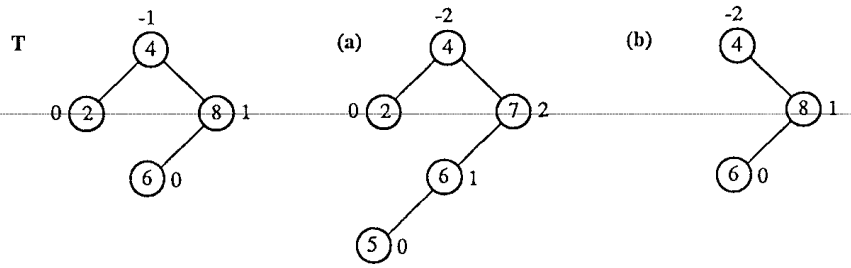


Figure 8. (a) Adjonction de 5 dans l'AVL T, et (b) suppression de 2 dans l'AVL T.

2.2.1. Exemple

Il s'agit de construire un AVL associé à la suite d'éléments 12, 3, 2, 5, 4, 7, 9, 11, 14, 10, en procédant à des adjonctions aux feuilles, et en rééquilibrant l'arbre après chaque adjonction qui l'a déséquilibré. Ainsi, le déséquilibre maximal après une adjonction est $\pm$ 2; de plus, il est clair que l'adjonction d'un élément x ne peut entraîner de déséquilibre que sur les nœuds du chemin de la racine à x. On montre plus loin que lorsqu'il y a déséquilibre après l'adjonction de x, il suffit de rééquilibrer l'arbre à partir d'un seul nœud A (dans le chemin de la racine à x, A est le dernier nœud pour lequel le déséquilibre est de $\pm$ 2).

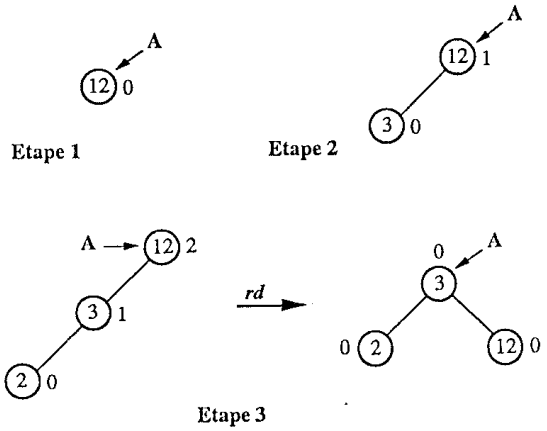


Figure 9

Voyons les différentes étapes de cette construction. Le premier rééquilibrage a lieu à l'étape 3; la transformation appliquée est une rotation droite en A, et l'arbre résultant a un déséquilibre nul en tous ses nœuds (figure 9). L'étape 5 est aussi une rotation droite sur l'arbre de racine 12; après la rotation, les nœuds 5 et 12 ont pour déséquilibre 0, et les autres nœuds conservent le déséquilibre qu'ils avaient avant l'adjonction (figure 10). L'étape 6 est une rotation gauche sur l'arbre de racine 3; le déséquilibre devient 0 pour les nœuds 3 et 5, et reste le même qu'avant l'adjonction pour les autres nœuds (figure 11). L'étape 7 est une rotation double gauche-droite

sur l'arbre de racine 12; le déséquilibre devient 0 pour les nœuds 12 et 7, et pour les autres nœuds reste le même qu'avant adjonction (figure 12).

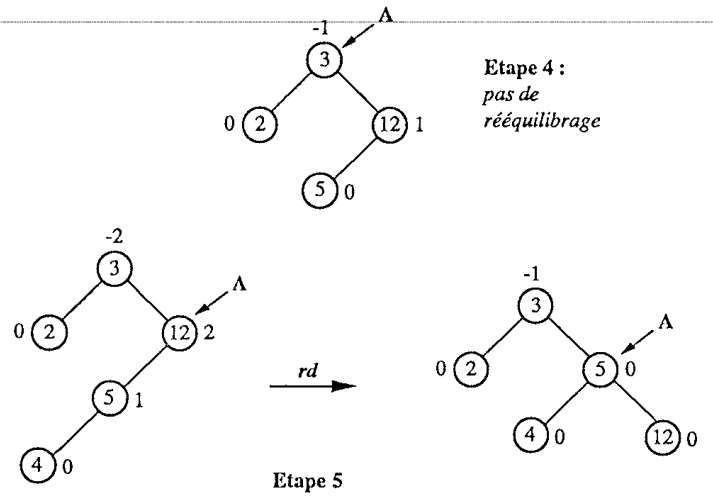


Figure 10

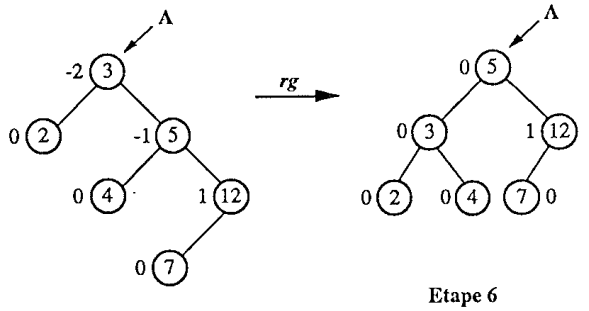


Figure 11

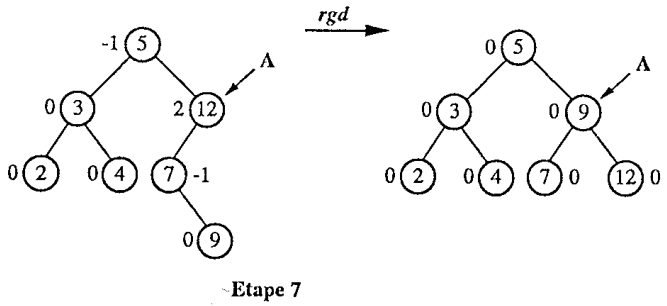
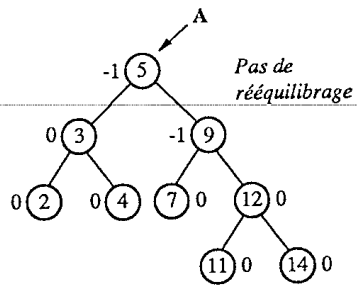
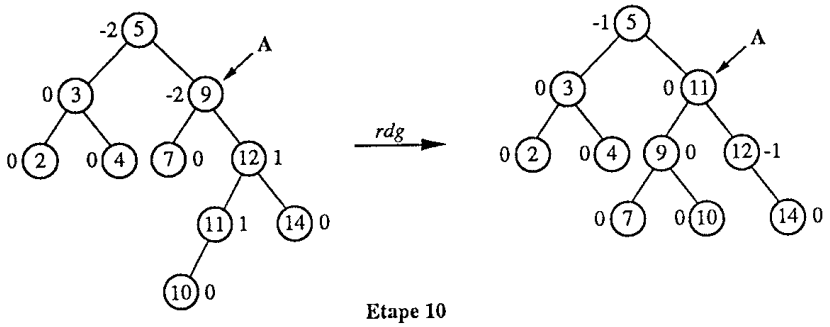


Figure 12



Etapes 8 et 9

Figure 13



Etape 10

Figure 14

Il n'y a pas de rééquilibrage pour les étapes 8 et 9 (figure 13). L'étape 10 est une double rotation droite-gauche sur l'arbre de racine 9; le déséquilibre devient 0 pour les nœuds 9 et 11; il devient -1 pour le nœud 12, et les autres nœuds conservent le même déséquilibre qu'avant l'adjonction (figure 14).

2.2.2. Principe général de rééquilibrage

L'exemple précédent a permis de dégager le principe de rééquilibrage dans les AVL qu'on explicite maintenant.

Soit $T = < r, G, D >$ un AVL; supposons que l'adjonction de l'élément x a lieu sur une feuille de G et *qu'elle fait augmenter de 1 la hauteur de* G, et que G reste un AVL (donc avant l'adjonction, le déséquilibre de G vaut 0).

1) Si le déséquilibre de T valait 0 avant l'adjonction, il vaut 1 après; T reste un AVL et sa hauteur a augmenté de 1.

2) Si le déséquilibre de T valait -1 avant l'adjonction, il vaut 0 après; T reste un AVL et sa hauteur n'est pas modifiée.

3) Si le déséquilibre de T valait +1 avant l'adjonction, il vaut +2 après : T n'est plus H-équilibré, il faut donc le restructurer en AVL.

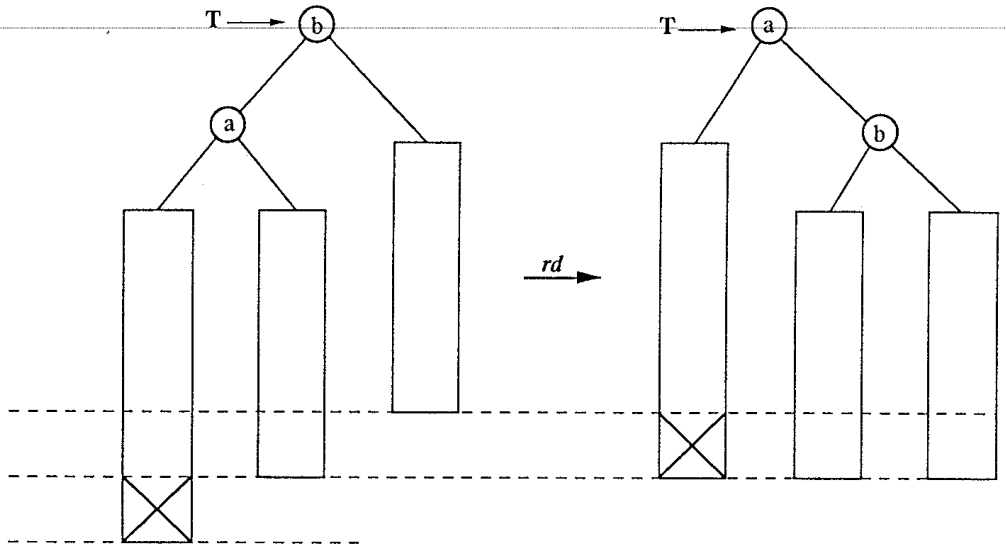


Figure 15.a. Adjonction dans un AVL.

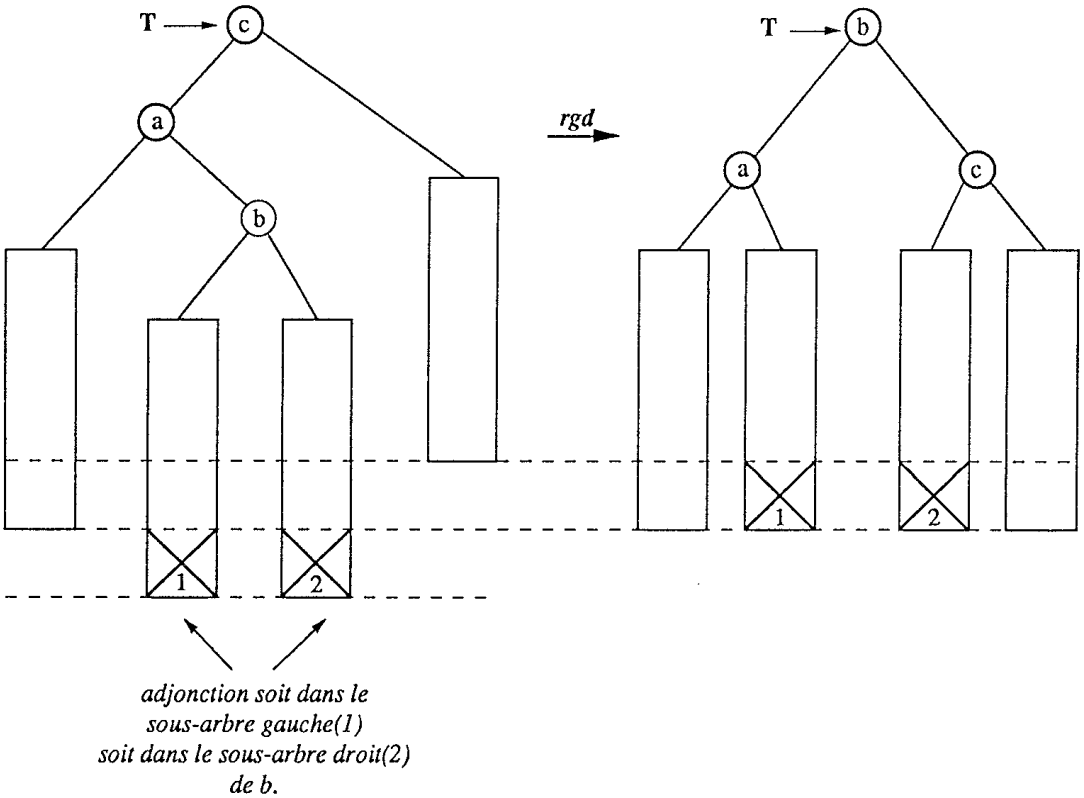


Figure 15.b. Adjonction dans un AVL.

Dans cette troisième hypothèse, il n'y a que deux cas possibles, selon que l'adjonction a lieu dans le sous-arbre gauche ou droit de G. Ces deux cas sont illustrés par la figure 15, où l'on a mis en évidence la hauteur des sous-arbres, et représenté par une croix l'augmentation de hauteur provoquée par l'adjonction. Dans le premier cas (figure 15.a), le déséquilibre de G passe de 0 à 1, et on rééquilibre T par une rotation droite. Dans le deuxième cas (figure 15.b), le déséquilibre de G passe de 0 à -1, et on rééquilibre T par une double rotation gauche-droite. Dans les deux cas, *l'arbre T obtenu après le rééquilibrage est bien un AVL (arbre binaire de recherche H-équilibré), et il retrouve exactement la hauteur qu'il avait avant l'adjonction de x.*

2.2.3. Principe de l'adjonction

Pour effectuer une adjonction dans un AVL, on procède donc comme pour l'adjonction aux feuilles d'un arbre binaire de recherche, mais il faut ensuite rééquilibrer si l'arbre obtenu n'est plus H-équilibré.

Spécification de l'adjonction dans les AVL

L'opération *ajouter-avl* utilise les opérations *rééquilibrer* et *déséquilibre*, et les opérations de rotation rg, rd, rgd, rdg définies précédemment sur les arbres .

ajouter-avl : Elément × Arbre → Arbre
rééquilibrer : Arbre → Arbre
déséquilibre : Arbre → Entier

Pour toutes variables T, G, D de sorte Arbre, et pour toutes variables x et r de sorte Elément, on a les préconditions et les axiomes suivants :

précondition
rééquilibrer(T) **est-défini-ssi** *déséquilibre*$(T) \in \{-2, -1, 0, 1, 2\}$

axiomes
ajouter-avl$(x,$ *arbre-vide*$) = x$
$x \leq r \Rightarrow$ *ajouter-avl*$(x, < r, G, D >)$ = *rééquilibrer*$(< r,$ *ajouter-avl*$(x, G), D >)$
$x > r \Rightarrow$ *ajouter-avl*$(x, < r, G, D >)$ = *rééquilibrer*$(< r, G,$ *ajouter-avl*$(x, D) >)$

déséquilibre$(T) = 0$ (ou 1 ou -1) $\Rightarrow$ *rééquilibrer*$(T) = T$
déséquilibre$(T) = +2$ & *déséquilibre*$(g(T)) = +1 \Rightarrow$ *rééquilibrer*$(T) = rd(T)$
déséquilibre$(T) = -2$ & *déséquilibre*$(d(T)) = -1 \Rightarrow$ *rééquilibrer*$(T) = rg(T)$
déséquilibre$(T) = +2$ & *déséquilibre*$(g(T)) = -1 \Rightarrow$ *rééquilibrer*$(T) = rgd(T)$
déséquilibre$(T) = -2$ & *déséquilibre*$(d(T)) = +1 \Rightarrow$ *rééquilibrer*$(T) = rdg(T)$

Il est important de remarquer que la fonction *rééquilibrer* est utilisée au plus une fois au cours de chaque adjonction.

Proposition : Toute adjonction dans un AVL nécessite au plus une rotation pour le rééquilibrer.

Preuve : Soit à ajouter l'élément x dans l'arbre T; c'est uniquement sur le chemin de la racine de T jusqu'à la feuille où l'on ajoute x qu'il peut y avoir à faire des rotations; on considère sur ce chemin le nœud le plus bas dont le déséquilibre (avant adjonction) est non nul, et l'on note A le sous-arbre enraciné en ce nœud (cf. figure 16).

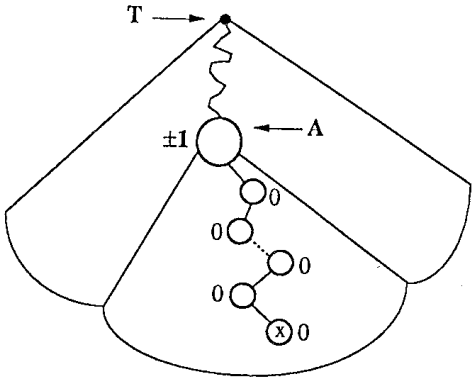


Figure 16

Or, la hauteur du sous-arbre A n'est pas modifiée par l'adjonction de x, même si l'on fait une rotation en A, c'est ce que l'on a vu dans le principe général. Donc dans l'AVL résultant de l'adjonction de x, le père de A, et tous ses ascendants ont exactement le déséquilibre qu'ils avaient avant l'adjonction de x; il n'y a donc jamais besoin d'effectuer de rotation «au-dessus» de A (ni de mise à jour du déséquilibre). □

2.2.4. Algorithme d'adjonction

Pour implanter efficacement l'algorithme d'adjonction, il faut toujours conserver la valeur du déséquilibre en chaque nœud de l'arbre. Dans la procédure *ajouteravl* on suppose que les arbres AVL ont le type suivant :

```
type AVL = ↑NŒUD;
     NŒUD = record
                deseq : -2..+2;
                val : Elément;
                g, d : AVL
            end;
```

Après chaque rotation, il faut mettre à jour la valeur du déséquilibre dans les nœuds modifiés par la rotation. Reprenons les rotations de la figure 15 : dans le premier

cas (figure 15.a), l'adjonction du nouvel élément crée un déséquilibre égal à +2 au nœud contenant b, et +1 au nœud contenant a; après avoir effectué une rotation droite, le déséquilibre du nœud contenant a et celui du nœud contenant b sont tous les deux nuls. Dans le deuxième cas (figure 15.b), le déséquilibre après adjonction vaut +2 au nœud contenant c, -1 au nœud contenant a et +1 (resp. -1) au nœud contenant b, si l'adjonction a lieu dans le sous-arbre gauche (resp. droit) du nœud contenant b; après une rotation gauche-droite, le déséquilibre devient 0 au nœud contenant b , 0 (resp. +1) au nœud contenant a, et -1 (resp. 0) au nœud contenant c.

D'après la propriété précédente, l'algorithme d'adjonction s'exprime de façon itérative : lors de la descente dans l'arbre à la recherche de la place où on doit ajouter x, on mémorise le dernier sous-arbre A pour lequel le déséquilibre est $\pm$ 1. Après avoir ajouté x à la feuille Y, c'est uniquement sur le chemin de A à Y qu'il est nécessaire de modifier les valeurs du déséquilibre. Il faut ensuite faire, le cas échéant, un rééquilibrage en A.

```
procedure, ajouteravl(var T : AVL; x : Elément );
var Y, A, P, AA, PP : AVL;
begin {création du nœud à ajouter}
      new(Y); Y↑.val := x; Y↑. deseq := 0; Y↑.g := nil; Y↑.d := nil;
      if T = nil then T := Y
      else begin
          A := T; AA := nil; P := T; PP := nil;
          {AA est le père de A; PP est le père de P}
          while P <> nil do begin
          {descente à la recherche de la feuille, en mémorisant le dernier nœud
          pointé par A dont le déséquilibre est ± 1}
              if P↑. deseq <> 0 then begin A := P; AA := PP end;
              PP := P;
              if x ≤ P↑.val then P := P↑.g else P := P↑.d
          end;
          {adjonction}
          if x ≤ PP↑.val then PP↑.g := Y else PP↑.d := Y;
          {modification du déséquilibre sur le chemin de A à Y}
          P := A;
          while P <> Y do
                    if x ≤ P↑.val then begin
                      P↑.deseq := P↑.deseq +1;
                      P := P↑.g
                    end
```

```
                else begin
                    P↑.deseq := P↑.deseq −1;
                    P := P↑.d
                end;
        {rééquilibrage}
        case A↑.deseq of
        0, +1, −1 : return;
        +2 :
            case A↑.g↑.deseq of
            +1 : begin RD(A); A↑.deseq := 0; A↑.d↑.deseq := 0 end;
            −1 : begin RGD(A);
                    caseA↑.deseq of
                    +1  : begin A↑.g↑.deseq := 0; A↑.d↑.deseq := −1 end;
                    −1  : begin A↑.g↑.deseq := +1; A↑.d↑.deseq := 0 end;
                     0  : {cas où A=Y}
                       begin A↑.g↑.deseq := 0; A↑.d↑.deseq := 0 end
                    end;
                    A↑.deseq := 0
                end
            end;
        −2 : {cas symétrique du cas +2}...
        end {fin du case};
        {mise à jour des pointeurs après une rotation}
        if AA = nil then T := A
        else if A↑.val≤ AA↑.val then AA↑.g := A else AA↑.d := A
    end {descente}
end ajouteravl;
```

Analyse de la complexité

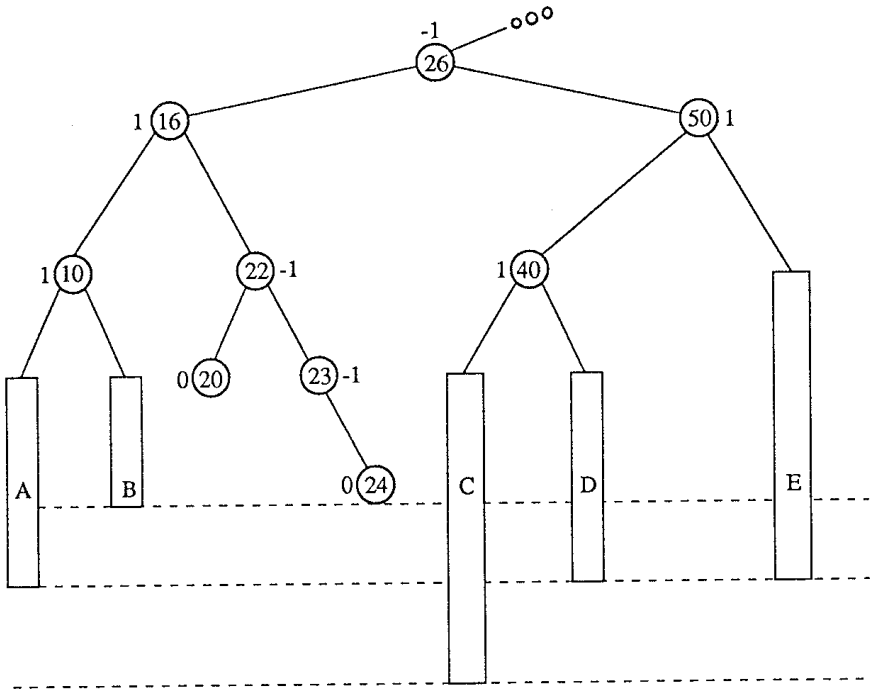
On a vu que la hauteur d'un AVL est toujours inférieure à $1.44 \log_2 n$; il est donc aisé de majorer le nombre de comparaisons nécessaires pour une adjonction, compte tenu du fait qu'il y a au plus une rotation par adjonction, et que chaque rotation n'entraîne qu'un nombre constant de comparaisons. La complexité de l'algorithme d'adjonction est donc en $\Theta(\log n)$ dans le cas le pire.

Qu'en est-il en moyenne? Considérons les AVL construits par adjonctions successives aux feuilles à partir d'une permutation de $[1, n]$, les $n!$ permutations étant équiprobables; on peut alors se poser les problèmes suivants :

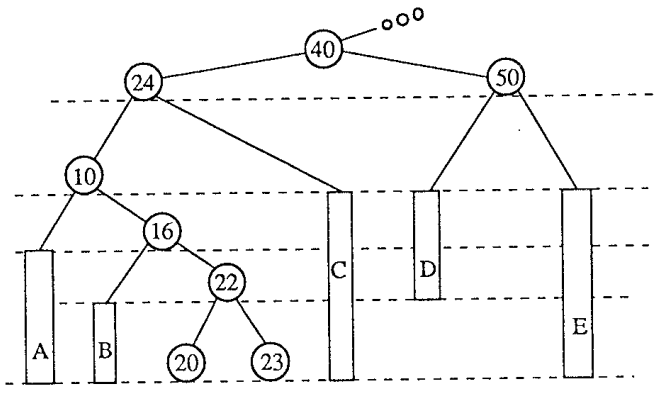
- quelle est la hauteur moyenne d'un tel AVL?
- quelle est la probabilité pour qu'une adjonction entraîne un rééquilibrage?

L'analyse mathématique de ces problèmes est très difficile, et non encore complètement résolue. Des résultats expérimentaux montrent cependant que les AVL ont un très bon comportement : leur hauteur moyenne est $\log_2 n + c$ (où c est une

constante inférieure à 1), ce qui est très proche de la hauteur des arbres complètement équilibrés; et il faut une rotation (simple ou double) en moyenne toutes les deux adjonctions.



(a) *arbre* AVL



(b) *arbre* AVL *résultant de la suppression de 26 dans l'arbre du (a)*

Figure 17

2.3. Suppression dans les AVL

Le principe de la suppression d'un élément dans un AVL est le même que pour les arbres binaires de recherche : remplacer l'élément à supprimer par l'élément de l'arbre qui lui est immédiatement inférieur. Mais l'arbre résultant d'une telle suppression peut ne plus être H-équilibré et il faut alors le réorganiser en arbre AVL. Le rééquilibrage après suppression fait intervenir les mêmes techniques que le rééquilibrage après adjonction ; mais dans le cas d'une suppression, la réorganisation de l'arbre peut nécessiter plusieurs rotations successives, sur le chemin de la feuille supprimée jusqu'à la racine. Par exemple, pour supprimer l'élément 26 sur l'AVL de la figure 17.a, on le remplace par 24 et on supprime le nœud contenant 24. Cette suppression diminue la hauteur du sous-arbre de racine 22, et le sous-arbre de racine 16 est alors trop déséquilibré. On le réorganise par une rotation droite mais cela accentue le déséquilibre au niveau immédiatement supérieur et il faut alors faire une rotation droite-gauche en 24. Les rotations peuvent ainsi remonter en cascade jusqu'à la racine de l'arbre (cf. figure 17.b).

2.3.1. Spécification de la suppression dans les AVL

L'opération *supprimer-avl* utilise l'opération *rééquilibrer* définie pour *ajouter-avl*, et les opérations *max* et *dlmax* (*max* a déjà été définie pour la suppression dans les arbres binaires de recherche, et *dlmax* est l'analogue de l'opération *dmax* sur les arbres binaires de recherche).

supprimer-avl : Elément × Arbre → Arbre
rééquilibrer : Arbre → Arbre
max : Arbre → Elément
dlmax : Arbre → Arbre

Pour toutes variables T, G, D de sorte Arbre, et pour toutes variables x et r de sorte Elément, on a les préconditions et les axiomes suivants :

préconditions
max(T) **est-défini-ssi** $T \neq$ *arbre-vide*
dlmax(T) **est-défini-ssi** $T \neq$ *arbre-vide*

axiomes
supprimer-avl(x, *arbre-vide*) = *arbre-vide*
$x = r \Rightarrow$ *supprimer-avl* ($x, < r, G$,*arbre-vide*>) = G
$x = r$ & $D \neq$ *arbre-vide*
$\Rightarrow$ *supprimer-avl*($x, < r$, *arbre-vide*, $D >$) = D
$x = r$ & $G \neq$ *arbre-vide* & $D \neq$ *arbre-vide*
$\Rightarrow$ *supprimer-avl*($x, < r, G, D >$) = *rééquilibrer*($< max(G), dlmax(G), D >$)
$x < r \Rightarrow$ *supprimer-avl*($x, < r, G, D >$) = *rééquilibrer*($< r$, *supprimer-avl*(x, G), $D >$)
$x > r \Rightarrow$ *supprimer-avl*($x, < r, G, D >$) = *rééquilibrer*($< r, G$, *supprimer-avl*(x, D)$>$)
max($< r, G$, *arbre-vide*>) = r

$dlmax(< r, G,$ *arbre-vide*$>) = G$
$D \neq$ *arbre-vide* $\Rightarrow max(< r, G, D >) = max(D)$
$D \neq$ *arbre-vide* $\Rightarrow dlmax(< r, G, D >) =$ *rééquilibrer* $(< r, G, dlmax(D) >)$

2.3.2. Algorithme de suppression

La réorganisation d'un AVL après une suppression peut entraîner des rotations en cascade sur le chemin allant de la feuille supprimée jusqu'à la racine de l'arbre; en fait les rotations se propagent de bas en haut tant que la hauteur du sous-arbre réorganisé après suppression est diminuée de 1 (c'est toujours le cas après une rotation, mais cela peut aussi être le cas sans qu'il y ait rotation, comme le montre la première étape de la suppression dans l'exemple de la figure 17). L'implémentation de l'algorithme nécessite donc de mémoriser complètement un chemin de la racine à une feuille; cela peut être fait soit de façon explicite en utilisant une pile dans une version proche de celle de l'adjonction, soit de façon implicite dans une version récursive proche de la spécification formelle (attention : après la suppression d'un élément contenu dans le nœud ν, il faut examiner la hauteur h du sous-arbre de racine ν; si h a diminué de 1, il faut éventuellement faire une rotation au niveau du père de ν). La programmation de ces deux versions est laissée en exercice.

Une suppression dans un AVL peut entraîner jusqu'à 1.5 $\log_2 n$ rotations (voir exercice sur les arbres de Fibonacci) mais la complexité reste toujours en $\Theta(\log n)$. Comme pour l'adjonction, l'analyse en moyenne reste un problème ouvert; les résultats expérimentaux montrent cependant qu'il y a seulement en moyenne une rotation pour cinq suppressions, ce qui va donc à l'encontre du sentiment intuitif qu'une suppression est plus coûteuse qu'une adjonction !

3. Arbres-2.3.4

Pour éviter les cas d'arbres de recherche dégénérés, on peut aussi faire varier le nombre de directions de recherche à partir d'un nœud. Dans un arbre binaire de recherche, chaque nœud contient un élément, et la comparaison avec l'élément cherché oriente la suite de la recherche soit dans le sous-arbre gauche, soit dans le sous-arbre droit. Dans les arbres-2.3.4 un nœud peut contenir entre un et trois éléments, rangés en ordre croissant, et le nombre de façons de poursuivre la recherche d'un élément donné, après comparaisons avec le (ou les) élément(s) contenu(s) dans un nœud est donc égal au nombre (2, 3 ou 4) d'intervalles délimités par cet (ou ces) élément(s).

Ce degré de liberté au niveau du nombre de fils des nœuds permet d'imposer une contrainte supplémentaire : toutes les feuilles d'un arbre-2.3.4 sont au même niveau; la hauteur de l'arbre est alors toujours logarithmique en fonction du nombre d'éléments qu'il contient. De plus, on a des algorithmes de rééquilibrage qui maintiennent un arbre-2.3.4 après toute adjonction ou suppression, en effectuant

une suite de rotations sur un chemin de la racine à une feuille. Toutes les opérations sur les arbres-2.3.4 se font en temps logarithmique dans le pire des cas. Enfin, il existe une implémentation efficace des arbres-2.3.4 sous la forme d'arbres binaires de recherche dits bicolores.

3.1. Recherche dans un arbre-2.3.4

3.1.1. Définitions et propriétés

Définition 1 : Un ***arbre de recherche*** est un arbre général étiqueté dont chaque nœud contient un k-uplet d'éléments distincts et ordonnés ($k = 1, 2, 3, ...$). Un nœud contenant les éléments $x_1 < x_2 < ... < x_k$ a $k+1$ sous-arbres, tels que :

- tous les éléments du premier sous-arbre sont inférieurs ou égaux à x_1,
- tous les éléments du $i^{ième}$ sous-arbre ($i = 2, ..., k$) sont strictement supérieurs à x_{i-1} et inférieurs ou égaux à x_i,
- tous les éléments du $(k+1)^{ième}$ sous-arbre sont strictement supérieurs à x_k.

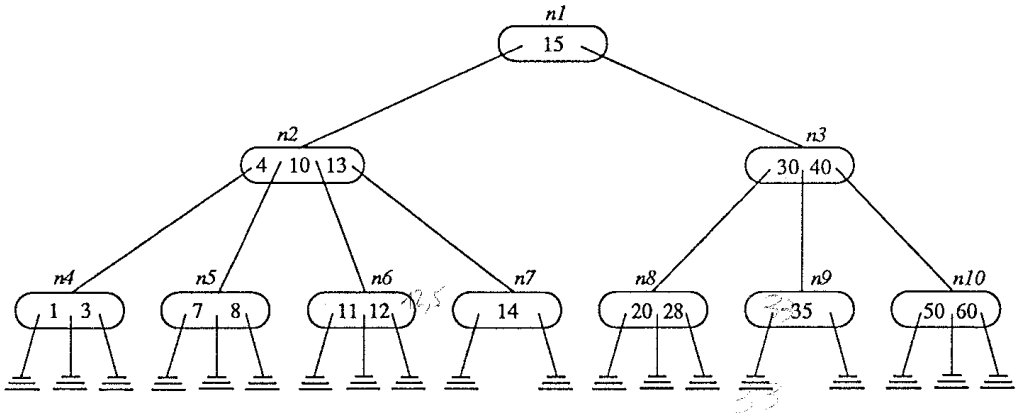


Figure 18. Arbre-2.3.4.

Dans un arbre de recherche, on appelle ***k-nœud*** un nœud contenant $(k-1)$ éléments.

Remarque : La liste L obtenue, dans le parcours en profondeur (procédure *Parc* du chapitre 7) d'un arbre de recherche, par insertion de l'élément $x_i (i = 1, ..., k-1)$ lors de la $(i+1)^{ième}$ visite du k-nœud contenant $< x_1, x_2, ..., x_{k-1} >$, est une liste croissante.

Définition 2 : Un ***arbre-2.3.4*** est un arbre de recherche dont les nœuds sont de trois types, 2-nœud ou 3-nœud ou 4-nœud, et dont toutes les feuilles sont situées au même niveau.

Exemple : La figure 18 montre un exemple d'arbre-2.3.4. Cet arbre contient 18 éléments distincts, et est constitué de 10 nœuds $n_1, n_2, ..., n_{10}$. On a marqué à l'intérieur de chaque nœud les éléments qu'il contient, rangés en ordre croissant, ainsi que les liens vers les sous-arbres. Par exemple le nœud n_2 a quatre sous-arbres : un sous-arbre qui contient tous les éléments inférieurs à 4, puis un sous-arbre qui contient les éléments supérieurs à 4 et inférieurs à 10, puis un sous-arbre qui contient les éléments compris entre 10 et 13, et enfin un sous-arbre qui contient les éléments supérieurs à 13, et inférieurs à 15.

Propriété 2 : La hauteur $h(n)$ d'un arbre-2.3.4 contenant n éléments est en $\Theta(\log\ n)$. Plus précisément, on a l'encadrement suivant :

$$\log_4(n+1) \leq h(n)+1 \leq \log_2(n+1)$$

Preuve : On considère les arbres-2.3.4 extrémaux de hauteur h donnée. L'arbre-2.3.4 qui contient le moins d'éléments est celui qui n'a que des 2-nœuds; puisque toutes les feuilles sont au même niveau, son nombre d'éléments est $2^0 + 2^1 + ... + 2^h = 2^{h+1} - 1$. A l'opposé, l'arbre-2.3.4 qui contient le plus d'éléments est celui qui n'a que des 4-nœuds, et son nombre d'éléments est 3 $(4^0 + 4^1 + ... + 4^h) = 4^{h+1} - 1$. On en déduit donc que pour tout arbre-2.3.4 contenant n éléments et de hauteur $h(n)$, on a la relation $2^{h(n)+1} - 1 \leq n \leq 4^{h(n)+1} - 1$, d'où l'on tire l'encadrement de la hauteur. □

3.1.2. Principe de la recherche dans un arbre-2.3.4

La recherche d'un élément x dans un arbre-2.3.4 M suit le même principe que la recherche dans un arbre binaire de recherche.

On compare x avec le(s) élément(s) $x_1, ..., x_i$ (i =1 ou 2 ou 3) contenus dans la racine de M ;

- s'il existe $j \in [1, i]$ tel que $x = x_j$ alors x est trouvé,
- si $x \leq x_1$, la recherche de x se poursuit dans le premier sous-arbre de M,
- si $x_j < x \leq x_{j+1}$ (pour j= 1, ..., $i-1$) la recherche de x se poursuit dans le $(j+1)^{ième}$ sous-arbre de M,
- si $x > x_i$, la recherche de x se poursuit dans le dernier sous-arbre de M,
- si la recherche se termine sur une feuille qui ne contient pas x, alors x n'appartient pas à M.

La recherche d'un élément dans un arbre-2.3.4 suit un chemin dans l'arbre, à partir de la racine vers une feuille. La complexité de l'algorithme de recherche, comptée en nombre de comparaisons entre éléments, est donc en $\Theta(\log\ n)$ dans tous les cas, d'après la propriété 2.

Dans un arbre-2.3.4, on étudie l'adjonction aux feuilles. L'adjonction d'un nouvel élément ne pose problème que si la feuille qui doit recevoir cet élément contient déjà trois éléments. Dans ce cas, il faut ajouter un nouveau nœud à l'arbre, et réorganiser en arbre-2.3.4.

3.2. Adjonction dans un arbre-2.3.4

3.2.1. Adjonction avec éclatements en remontée

L'exemple qui suit montre la construction d'un arbre-2.3.4 par adjonctions successives aux feuilles, à partir de l'arbre vide. On insère successivement les éléments : 4, 35, 10, 13, 3, 30, 15, 12, 7, 40, 20, 11, 6.

L'adjonction de 4 entraîne la création d'un 2-nœud, qui se transforme en 3-nœud avec l'adjonction de 35, puis en 4-nœud avec l'adjonction de 10 (figure 19).

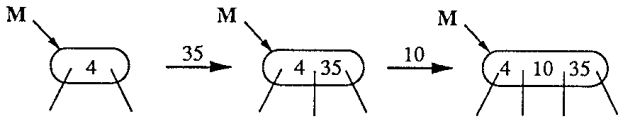


Figure 19

A cette étape, l'unique nœud de M ne peut plus contenir de nouveaux éléments. Or, du point de vue de la recherche, M est équivalent à l'arbre binaire de la figure 20.

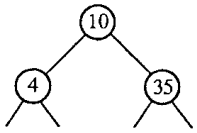


Figure 20

Cet arbre est bien un arbre-2.3.4, et il y a de la place dans ses feuilles pour de nouveaux éléments. On peut alors ajouter 13, puis 3, puis 30 (figure 21).

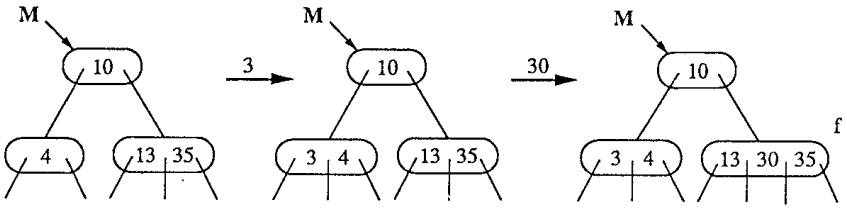


Figure 21

L'adjonction de 15 provoque l'*éclatement* de la feuille f en deux 2-nœuds contenant respectivement le plus petit et le plus grand élément de f, et l'élément médian 30

doit être ajouté au nœud père de f ; il y a alors de la place pour 15 dans le même nœud que 13 qui devient alors un 3-nœud (figure 22).

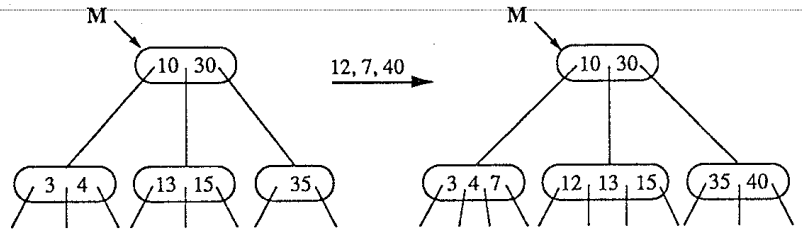


Figure 22

L'adjonction des éléments 12, 7 et 40 peut être facilement réalisée, mais l'adjonction de 20 entraîne un *éclatement* en deux de la feuille contenant 12, 13 et 15, avec *remontée* de 13 dans le nœud père (figure 23).

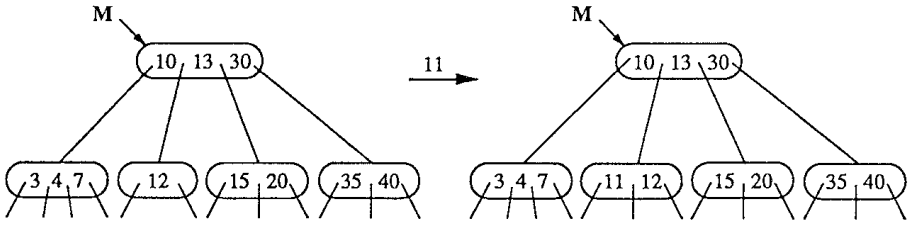


Figure 23

On peut alors ajouter 11 sans problème. L'adjonction de 6 provoque l'*éclatement* de la feuille (3,4,7), dont le père contient déjà trois éléments ; la *remontée* de 4 fait donc *éclater à son tour* la racine de l'arbre en deux nœuds et il faut aussi créer un nouveau nœud racine pour recevoir l'élément médian 13 (figure 24).

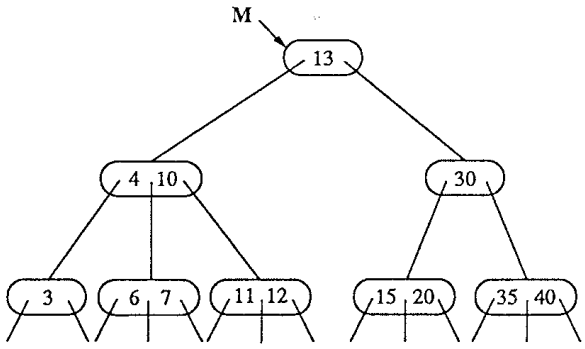


Figure 24

3.2.2. Adjonction avec éclatements à la descente

Avec la méthode d'adjonction précédente, les éclatements peuvent *remonter en cascade*, éventuellement sur toute la hauteur de l'arbre si le chemin suivi pour déterminer l'endroit de l'adjonction n'est formé que de 4-nœuds.

Pour éviter la propagation des éclatements de bas en haut, il suffit de travailler sur des arbres-2.3.4 qui ne contiennent jamais deux 4-nœuds à la suite : dans ce cas *toute adjonction provoque au plus un éclatement.* Ceci peut être réalisé en éclatant les 4-nœuds *à la descente* : lors de la recherche de la place de l'élément à ajouter, on parcourt un chemin dans l'arbre-2.3.4, à partir de la racine jusqu'à une feuille; seuls les 4-nœuds de ce chemin risquent d'éclater à la suite de l'adjonction; on prévient ce risque en les faisant éclater, au fur et à mesure de leur rencontre, *avant* de réaliser l'adjonction. Cette précaution a évidemment l'inconvénient de provoquer quelquefois des éclatements qui se révèlent inutiles. Reprenons l'exemple précédent en utilisant la méthode d'éclatement à la descente : l'arbre-2.3.4 obtenu après adjonctions successives de 4, 35, 10, 13, 3, 30, 15, 12, 7, 40 et 20 est le même qu'avec la première méthode mais lors de la recherche de la place où on doit insérer 11, on rencontre un 4-nœud (la racine) que l'on doit faire éclater à ce moment là, ce qui donne l'arbre de la figure 25.

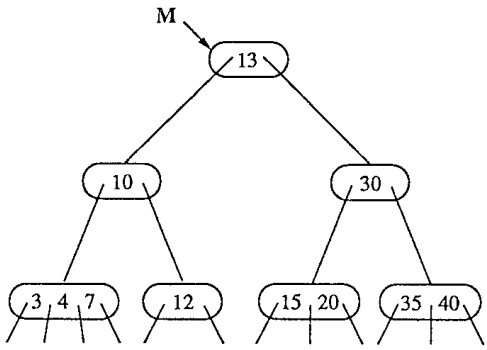


Figure 25

L'adjonction de 11 se réalise ensuite sans problème dans le nœud qui contient 12.

A cette étape, l'arbre obtenu (figure 26) n'est pas le même que celui que l'on obtient avec la méthode d'éclatements en remontée; il a une hauteur plus grande. Mais si, par exemple, on ajoute maintenant 6, il faut éclater le nœud (3,4,7) et l'on obtient finalement le même arbre que celui de la figure 24.

Dans cette méthode, tous les rééquilibrages se font pendant la descente dans l'arbre, lors de la recherche de la feuille où doit être inséré l'élément; lorsqu'on arrive sur cette feuille, il ne reste plus qu'à y ajouter effectivement l'élément.

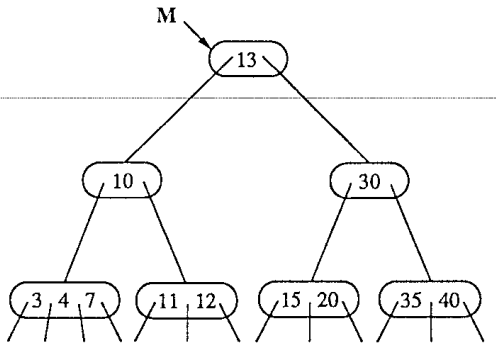
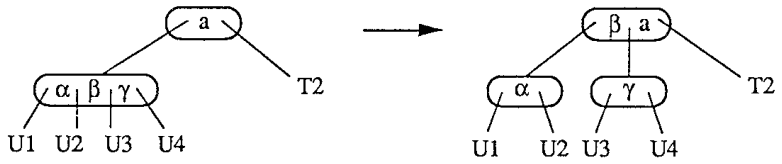
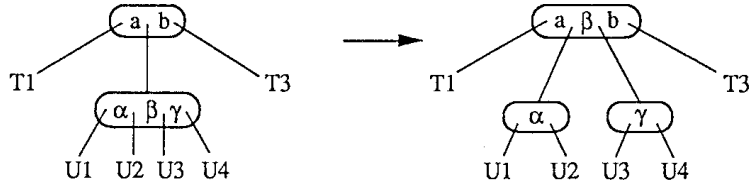


Figure 26

Les transformations de rééquilibrage à la descente sont purement locales, comme le montre la figure 27. Lorsqu'un nœud éclate, son père n'est pas un 4-nœud (sinon on aurait dû le faire éclater juste avant). L'éclatement d'un nœud consiste donc à transformer son père soit en 3-nœud, si c'est un 2-nœud, soit en 4-nœud, si c'est un 3-nœud, ce qui est réalisé en un nombre constant d'opérations. Cette transformation locale ne modifie pas la profondeur des feuilles, sauf si c'est la racine qui éclate, et dans ce cas la hauteur de l'arbre augmente d'une unité. Ainsi, la procédure d'adjonction conserve bien les propriétés d'arbre-2.3.4.



(i) *cas où le père du noeud à éclater est un 2-noeud*



(ii) *cas où le père du noeud à éclater est un 3-noeud*

Figure 27. Eclatements à la descente.

Remarque 1 : Les deux méthodes d'adjonction (éclatements en remontée ou à la descente) opèrent toujours sur un chemin de la racine à une feuille de l'arbre-2.3.4. Elles ont donc toutes les deux une complexité qui est dans tous les cas en $\Theta(\log n)$, où n est le nombre d'éléments contenus dans l'arbre-2.3.4.

La seconde méthode présente cependant plusieurs avantages pratiques par rapport à la méthode d'adjonction avec éclatements en remontée :

- on n'a jamais besoin de retraverser le chemin de recherche pour restaurer l'équilibre de l'arbre, donc l'algorithme se décrit de façon itérative sans pile ;
- l'arbre-2.3.4 est accessible en parallèle par plusieurs utilisateurs avec une synchronisation minimale, chaque mise à jour n'affectant qu'une petite portion de l'arbre.

Par contre, elle a l'inconvénient de prendre plus de place : le taux d'occupation des nœuds étant plus faible qu'avec la première méthode, il faut donc plus de nœuds, et par conséquent la hauteur de l'arbre est plus grande.

Remarque 2 : Si on voulait insérer plusieurs fois le même élément, il faudrait modifier à la fois la définition des arbres-2.3.4 (un nœud pourrait contenir plusieurs fois le même élément) et les algorithmes de recherche et d'adjonction (l'éclatement d'un nœud pourrait répartir des éléments égaux dans deux fils différents de ce nœud). On traite ici le cas où *tous les éléments sont différents.*

3.2.3. Spécification de l'adjonction avec éclatements à la descente

Dans la spécification de l'algorithme d'adjonction avec éclatements à la descente, on suppose qu'un arbre-2.3.4 est défini par la donnée de la suite des éléments $e_1 < ... < e_i$ (i = 1, 2 ou 3) contenus dans la racine, et de la liste des $i + 1$ sous-arbres de la racine. La spécification que l'on donne ici est incomplète. On a plutôt essayé d'expliquer le fonctionnement de l'algorithme, et d'illustrer tous les cas différents par des exemples. (Le lecteur rebuté par la longueur de cette spécification peut passer directement au paragraphe 4.)

sorte A234
utilise Elément, Booléen
opérations

arbre-vide	: → A234
< _, _, _ >	: Elément × A234 × A234 → A234
< (_, _), _, _, _ >	: Elément × Elément × A234 × A234 × A234 → A234
< (_, _, _), _, _, _, _ >	: Elément × Elément × Elément × A234 × A234 × A234 × A234 → A234
est-vide	: A234 → Booléen
2-fils, 3-fils, 4-fils	: A234 → Booléen
F_1, F_2, F_3, F_4	: A234 → A234
E_1, E_2, E_3	: A234 → Elément

L'opération *est-vide* rend vrai si l'arbre est vide et faux sinon.

Les opérations *2-fils*, *3-fils*, *4-fils* permettent de distinguer les arbres-2.3.4 selon que la racine est un 2-nœud, un 3-nœud ou un 4-nœud.

Par exemple, 3-fils(T) = *vrai si* T *est de la forme* $< (a,b), T_1, T_2, T_3 >$ *avec* T_1, T_2 *et* T_3 *non vides, et faux sinon.*

Les opérations F_1, F_2, F_3, F_4 permettent d'accéder aux sous-arbres de la racine, et l'on prend la convention que si la racine d'un arbre n'a que i sous-arbres alors F_j a pour résultat l'arbre vide, noté aussi $\emptyset$, pour $j > i$.

Par exemple, si T *est de la forme* $< (a,b), T_1, T_2, T_3 >$, *on a* $F_2(T) = T_2$ *et* $F_4(T) = \emptyset$.

E_1, E_2 et E_3 sont des opérations partielles qui permettent d'accéder aux éléments. $E_1(T)$ est défini pour $T \neq \emptyset$, $E_2(T)$ est défini si, et seulement si, *3-fils*(T) est vrai, et $E_3(T)$ est défini si, et seulement si, *4-fils*(T) est vrai.

Par exemple, si T *est de la forme* $< (a,b), T_1, T_2, T_3 >$, $E_2(T) = b$, *et* $E_3(T)$ *n'est pas défini.*

Dans l'algorithme d'adjonction, le cas où la racine est un 4-nœud est traité à part car l'éclatement introduit alors une nouvelle racine. On a donc besoin de deux opérations, *ajouter234* et *ajouter*.

ajouter234 : A234 × Elément → A234
ajouter : A234 × Elément → A234

L'opération *ajouter* est définie uniquement pour les arbres-2.3.4 dont la racine est un 2-nœud ou un 3-nœud. On a les axiomes suivants :

ajouter234$(\emptyset, x) = < x, \emptyset, \emptyset >$
4-fils(T) = vrai ⇒ *ajouter234*$(<(a,b,c), T_1, T_2, T_3, T_4 >, x)$
$= ajouter\ (< b, < a, T_1, T_2 >,\ < c, T_3, T_4 >>, x)$,
avec $T = < (a,b,c), T_1, T_2, T_3, T_4 >$
4-fils(T) = faux ⇒ *ajouter234*(T, x) = *ajouter*(T, x).

Dans le cas où la racine d'un arbre-2.3.4 T est un 2-nœud, si le sous-arbre où l'on doit faire l'adjonction a quatre fils, on l'éclate (figure 27-i), et on continue l'adjonction dans le «bon» sous-arbre résultant de cet éclatement. L'axiome ci-dessous décrit ce qui se passe quand on ajoute un élément dans le *premier sous-arbre* résultant de l'éclatement de $F_1(T)$:

2-fils(T) = vrai & $x < E_1(T)$ & *4-fils*$(F_1(T))$ = vrai & $x < E_2(F_1(T))$
⇒ *ajouter*$(T, x) = < (\beta, a)$, *ajouter*$(< \alpha, U_1, U_2 >, x), < \gamma, U_3, U_4 >, T_2 >$,
avec $T = < a, T_1, T_2 >$ et $T_1 = < (\alpha, \beta, \gamma), U_1, U_2, U_3, U_4 >$.

On a un axiome similaire dans le cas où l'adjonction doit se faire dans le deuxième sous-arbre résultant de l'éclatement (cas où $x > E_2(F_1(T))$). Et si c'est dans $F_2(T)$ que l'adjonction a lieu (cas où $x > E_1(T)$), les axiomes s'obtiennent de façon analogue.

Lorsque le nœud où doit se faire l'adjonction n'est pas un 4-nœud, les axiomes sont similaires à ceux de l'adjonction dans les arbres binaires de recherche, par exemple :

$$\textit{2-fils}(T) = \text{vrai} \;\&\; x < E_1(T) \;\&\; \textit{4-fils}(F_1(T)) = \text{faux} \;\&\; F_1(T) \neq \textit{arbre-vide}$$
$$\Rightarrow \textit{ajouter}(T, x) = < E_1(T), \textit{ajouter}(F_1(T), x), F_2(T) >.$$

De même, dans le cas où la racine de T est un 3-nœud, si le sous-arbre où doit avoir lieu l'adjonction a quatre fils, on l'éclate. L'axiome ci-dessous décrit le cas où $F_2(T)$ est éclaté, et où l'adjonction se fait dans le deuxième sous-arbre résultant de l'éclatement. (On a un axiome analogue lorsque l'adjonction se fait dans le premier sous-arbre.)

$$\textit{3-fils}(T) = \text{vrai} \;\&\; E_1(T) < x < E_2(T) \;\&\; \textit{4-fils}(F_2(T)) = \text{vrai} \;\&\; x > E_2(F_2(T))$$
$$\Rightarrow \textit{ajouter}(T, x) =< (a, \beta, b), T_1, < \alpha, U_1, U_2 >, \textit{ajouter}(< \gamma, U_2, U_3 >, x), T_3 >,$$

avec $T = < (a, b), T_1, T_2, T_3 >$ et $T_2 = < (\alpha, \beta, \gamma), U_1, U_2, U_3, U_4 >$.

Les cas d'éclatement de $F_1(T)$ ou de $F_3(T)$ se traitent de façon similaire.

Lorsque le sous-arbre où se fait l'adjonction n'est pas un *4-fils*, il n'y a pas d'éclatement; par exemple

$$\textit{3-fils}(T) = \text{vrai} \;\&\; E_1(T) < x < E_2(T) \;\&\; \textit{4-fils}(F_2(T)) = \text{faux} \;\&\; F_2(T) \neq \textit{arbre-vide}$$
$$\Rightarrow \textit{ajouter}(T, x) = < (E_1(T), E_2(T)), F_1(T), \textit{ajouter}(F_2(T), x), F_3(T) >$$

L'axiome suivant décrit le cas de l'adjonction dans un arbre vide :

$$\textit{ajouter234}(\emptyset, x) =< x, \emptyset, \emptyset >$$

Lorsqu'on ajoute à une feuille on a des axiomes particuliers, dont on donne quelques exemples :

$$x < a \Rightarrow \textit{ajouter}(< a, \emptyset, \emptyset >, x) =< (x, a), \emptyset, \emptyset, \emptyset >$$
$$x < a \Rightarrow \textit{ajouter}(< (a, b), \emptyset, \emptyset, \emptyset >, x) = < (x, a, b), \emptyset, \emptyset, \emptyset, \emptyset >$$

(Pour une feuille qui est un 4-nœud, le traitement est le même que dans le cas général.)

3.2.4. Représentation des arbres-2.3.4

Quand on passe à l'implantation de l'algorithme, le problème qu'on rencontre immédiatement est celui de la *représentation des différents types de nœuds.*

On peut choisir une représentation maximale : tous les nœuds sont des enregistrements à sept champs : trois champs «élément» e_i et quatre champs «pointeur» P_j (figure 28); mais cette représentation gaspille beaucoup de place pour les nœuds qui ne sont pas des 4-nœuds.

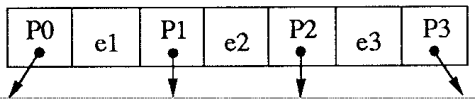
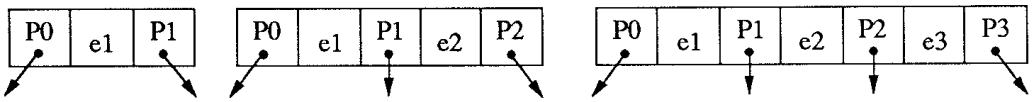


Figure 28

On peut choisir de représenter les différents types de nœuds par différents types de données (figure 29); cependant, dans ce cas, la manipulation des nœuds lors des réorganisations de l'arbre non seulement rend fastidieuse la programmation de l'algorithme, mais de plus ralentit son exécution (ce n'est d'ailleurs pas possible de façon efficace dans un langage de programmation comme Pascal).



2-noeud **3-noeud** **4-noeud**

Figure 29

On choisit ici de représenter les arbres-2.3.4 sous forme d'arbres binaires bicolores. Cela évite les inconvénients précédents. La représentation des nœuds est uniforme, ce sont des enregistrements à quatre champs : un élément, deux pointeurs, et un bit de couleur; et les transformations de rééquilibrage se réduisent à des rotations simples ou doubles, à gauche ou à droite (cf. le premier paragraphe de ce chapitre).

3.3. Arbres bicolores

3.3.1. Définition et propriétés

Les ***arbres bicolores***, inventés par Guibas et Sedgewick dans les années soixante-dix, sont des arbres binaires de recherche dont les nœuds portent une information supplémentaire : les nœuds sont soit rouges soit blancs. Les arbres bicolores sont une représentation des arbres-2.3.4.

Dans un arbre-2.3.4, on hiérarchise les éléments à l'intérieur d'un 4-nœud ou d'un 3-nœud et l'on considère alors les arbres-2.3.4 comme des arbres binaires dont les nœuds sont eux-mêmes des petits morceaux d'arbres binaires. Pour simuler cette double hiérarchie, on colorie les liens de deux façons différentes : *en rouge* (double trait) si les éléments appartiennent au même nœud de l'arbre-2.3.4 – on parle alors *d'éléments jumeaux* (sur la figure 30, les éléments a, b et c sont jumeaux) –, et *en blanc* (simple trait) si les éléments sont dans des nœuds différents de l'arbre-2.3.4 initial. En fait, la couleur peut être attachée aux nœuds plutôt qu'aux liens : tout nœud a la couleur du lien qui pointe sur lui, et un nœud est donc rouge si, et seulement si, l'élément qu'il contient est un jumeau de l'élément contenu dans son père.

Les différentes étapes de la transformation d'un arbre-2.3.4 en arbre bicolore apparaissent sur la figure 30 pour un 4-nœud, et sur la figure 31 pour un 3-nœud (il y a dans ce cas deux représentations possibles – penché à droite ou à gauche –, et les deux peuvent coexister à la suite de rotations, comme on le verra plus loin).

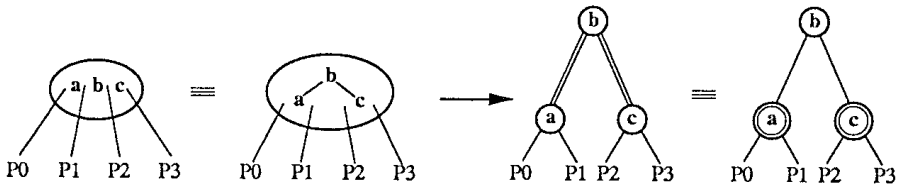
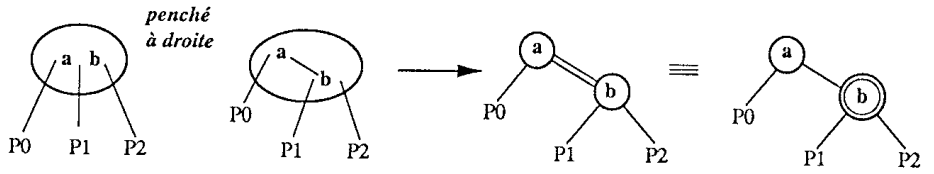
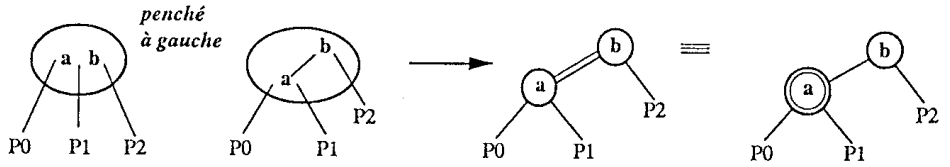


Figure 30. Transformation d'un 4-nœud.



i) *représentation d'un 3-noeud "penché à droite"*



ii) *représentation d'un 3-noeud "penché à gauche"*

Figure 31. Transformation d'un 3-nœud.

Par exemple, l'arbre-2.3.4 $M1$ de la figure 32(a) peut être représenté par l'arbre bicolore $M2$ de la figure 32(b) : en compactant les nœuds jumeaux de $M2$, on retrouve $M1$.

La hauteur de l'arbre bicolore obtenu par un tel procédé de construction est au plus égale à deux fois la hauteur de l'arbre-2.3.4 initial, augmentée de 1. En effet, par construction, il n'y a jamais deux liens rouges à la suite sur un chemin de l'arbre bicolore, et de plus, tous les chemins ont le même nombre de liens blancs, qui est égal à la hauteur $h_{2.3.4}(n)$ de l'arbre-2.3.4 initial. Or, d'après la Propriété 2 on a la relation : $\log_4(n+1) \leq h_{2.3.4}(n) + 1 \leq \log_2(n+1)$. On peut donc énoncer la propriété suivante.

Propriété 3 : Tout arbre bicolore associé à un arbre-2.3.4 contenant n éléments a une hauteur d'ordre $\Theta(\log n)$.

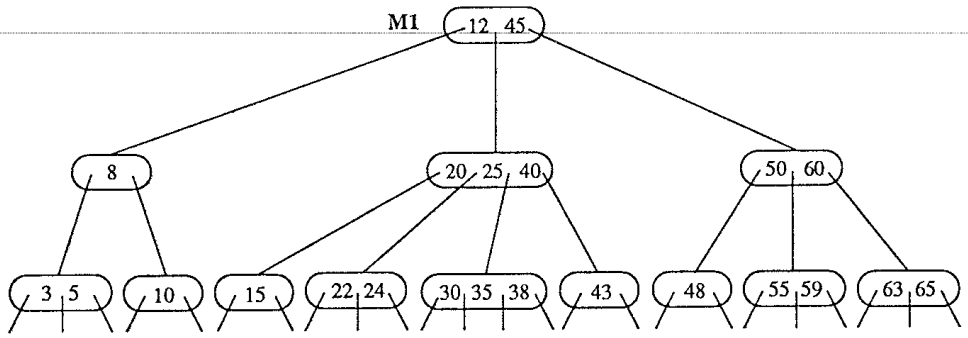


Figure 32(a). Arbre-2.3.4.

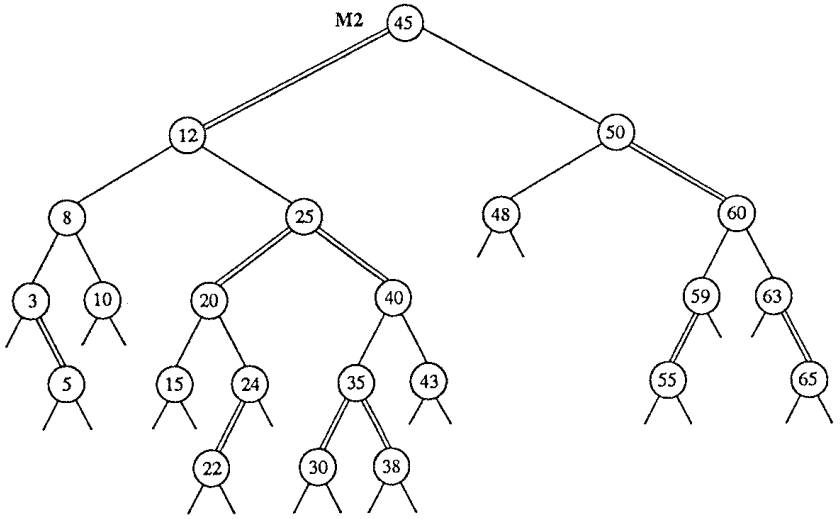


Figure 32 (b). Une représentation en arbre bicolore.

La recherche d'un élément dans un arbre bicolore se fait exactement de la même façon que dans un arbre binaire de recherche, sans tenir aucun compte des couleurs. Elle nécessite donc un nombre de comparaisons qui est en $\Theta(\log n)$ *dans le pire des cas*.

3.3.2. Adjonction dans un arbre bicolore

L'adjonction d'un élément peut être réalisée en simulant l'adjonction avec éclatements à la descente dans un arbre-2.3.4 : éclater un 4-nœud revient à inverser les couleurs des éléments jumeaux de ce nœud (figure 33).

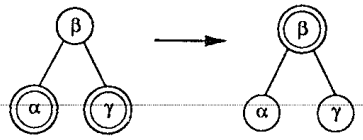


Figure 33

Cependant, il se peut que cette transformation fasse apparaître deux nœuds rouges consécutifs dans l'arbre bicolore; il faut éviter ce phénomène si l'on veut conserver la propriété de hauteur logarithmique des arbres bicolores, et pour cela on a recours à des transformations locales qui ont la forme des rotations présentées au début de ce chapitre.

Considérons les différentes configurations qui peuvent se présenter :

1) Le 4-nœud qui doit éclater est attaché à un 2-nœud (comme premier ou second fils); alors une simple inversion des couleurs est suffisante (figure 34).

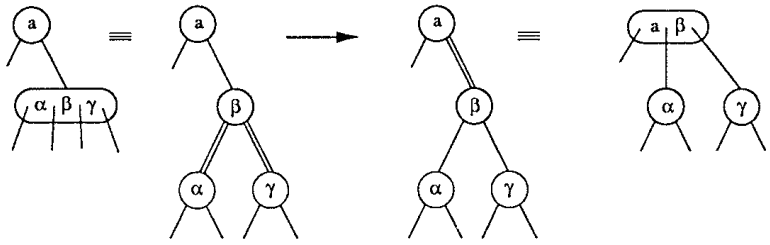


Figure 34

2) Le 4-nœud E qui doit éclater est rattaché à un 3-nœud; alors il faut distinguer plusieurs cas – on considère ici les trois cas possibles lorsque le 3-nœud est représenté «penché à droite» (figure 31-i), mais il y a trois cas symétriques lorsque le 3-nœud est représenté «penché à gauche».

a) Si E est premier fils du 3-nœud, il suffit encore d'inverser les couleurs des éléments contenus dans le 4-nœud (figure 35).

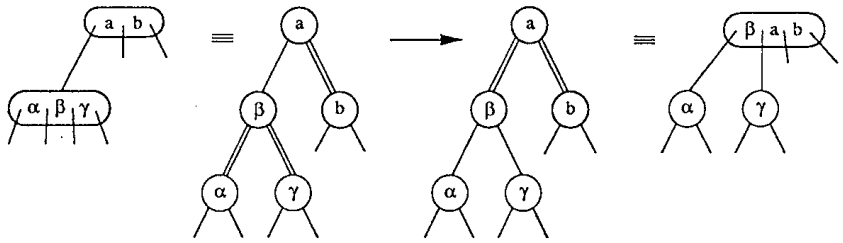


Figure 35

b) Si E est second fils du 3-nœud, l'inversion des couleurs entraîne une mauvaise disposition des éléments jumeaux a, β et b, mais pour retrouver la bonne représentation (figure 30), il suffit alors d'effectuer une rotation droite-gauche au niveau du nœud contenant a (figure 36).

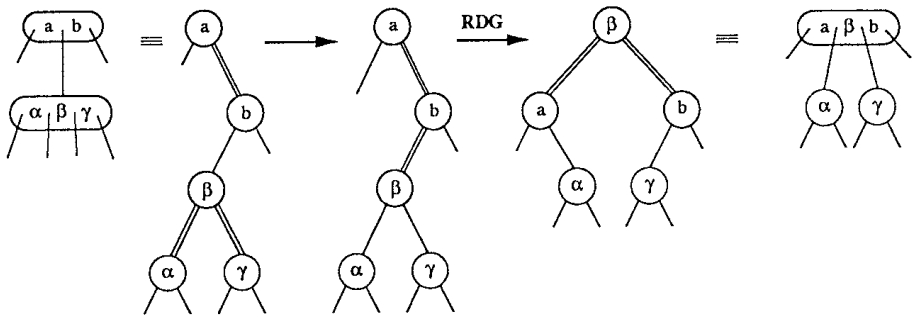


Figure 36

c) Si E est troisième fils du 3-nœud, l'inversion des couleurs entraîne encore une mauvaise disposition des éléments jumeaux a, b et β ; dans ce cas, il faut faire une rotation gauche pour rétablir l'arbre bicolore (figure 37).

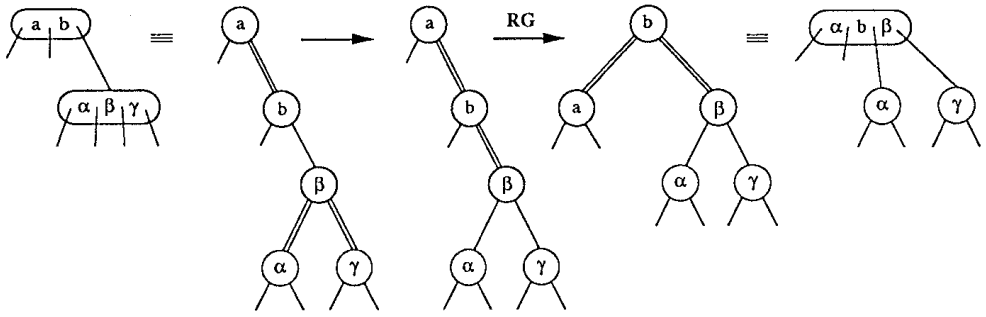


Figure 37

Algorithme d'adjonction

L'algorithme d'adjonction d'un élément dans un arbre bicolore se décrit donc informellement de la façon suivante.

• Descendre à partir de la racine à la recherche de la feuille où il faut insérer l'élément.

• Sur ce chemin, lorsqu'un nœud A a ses deux fils rouges, on inverse les couleurs de A et de ses fils; si de plus le père de A est lui aussi rouge, il faut faire une rotation au niveau du grand-père de A avant de poursuivre la descente.

• L'adjonction d'un élément se fait toujours dans une feuille qui devient un nœud rouge, puisque le nouvel élément est ajouté en tant que jumeau; dans le cas où le père du nœud ajouté est rouge, mais n'a pas de frère rouge, il faut effectuer une rotation comme en b).

Par exemple, pour ajouter 20 à l'arbre bicolore de la figure 38(a) (qui résulte de l'adjonction successive des éléments 4, 35, 10, 13, 3, 30, 15, 12, 7 et 40) on suit le chemin 10 → 30 → 13; les deux fils de 13 sont rouges, on inverse donc les couleurs (13 devient rouge et ses deux fils blancs) mais alors 13 et 30 étant rouges, il faut faire une rotation droite-gauche au niveau de 10. Puis on continue à descendre dans la branche droite de 13 : 30 → 15 et on peut alors insérer 20 à droite de 15 (figure 38(b)). Si l'on ajoute ensuite 21, dont la place est à droite de 20, il faudra faire une rotation gauche au niveau de 15.

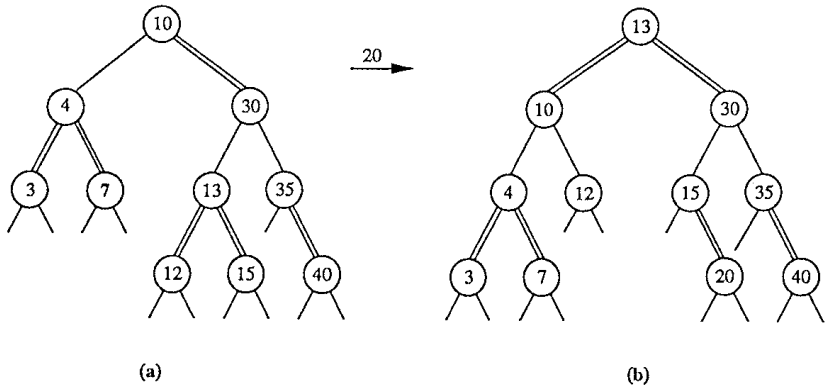


Figure 38. Adjonction dans un arbre bicolore.

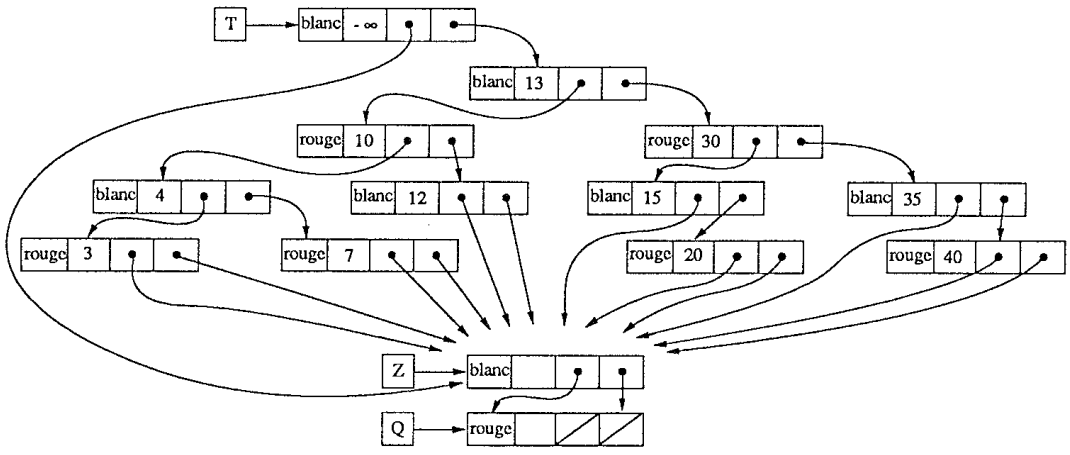


Figure 39. Représentation de l'arbre de la figure 38(b).

On va maintenant décrire une procédure de recherche dans un arbre bicolore, avec adjonction dans le cas où l'élément cherché n'est pas dans l'arbre. Un arbre bicolore est représenté par le type Pascal suivant :

```
type couleur = (blanc, rouge);
     Bic = ↑Bnœud;
     Bnœud = record
                 coul : couleur;
                 val : Elément;
                 g, d : Bic
             end;
```

Pour unifier le traitement, la procédure rech-adj-bic travaille sur un arbre bicolore avec une tête T, une sentinelle Z dont la couleur est le blanc, et un quadruplet supplémentaire pointé par Q, sous la sentinelle (figure 39).

Le quadruplet de la tête T contient dans son champ *val* un élément strictement inférieur à tous les éléments possibles; sa couleur est le blanc, son lien g pointe sur Z et son lien d pointe sur la racine de l'arbre, ou sur Z si l'arbre est vide. Son utilité est de ne pas particulariser l'adjonction dans un arbre vide.

L'utilisation de la sentinelle Z est classique : on fait pointer sur Z tous les liens de l'arbre qui sont à nil, et, de plus, avant de commencer la recherche d'un élément X, on affecte X au champ *val* de la sentinelle; de cette façon, la recherche aboutit toujours sur X et l'on n'a pas à distinguer deux cas de sortie de l'itération. De plus, on a introduit un quadruplet pointé par Q qui est rouge, et dont les liens g et d sont à nil, pour pouvoir traiter de la même façon les inversions de couleurs qui ont lieu au milieu de l'arbre et celles qui ont lieu en bas (adjonction).

```
procedure rech-adj-bic(X: Elément; Z, Q : Bic; var T, A: Bic; var B: Boolean);
{Cette procédure recherche l'élément X dans l'arbre bicolore T (avec tête et
sentinelle, et dont tous les éléments sont distincts) et, s'il n'y est pas, réalise
l'adjonction de X dans T ; elle renvoie pour résultat dans B la valeur true s'il y a
eu adjonction et false sinon, et dans tous les cas elle donne l'adresse A de X dans
l'arbre bicolore T ; Z est la sentinelle, sa couleur est le blanc et ses liens g et d
valent Q ; Q est rouge}

var P, GP, GGP : Bic;{on a besoin de trois ascendants, père, grand-père
et arrière-grand-père, pour le rééquilibrage}
begin
  A := T; Z↑.val := X; B := false; P := T; GP := T;
  repeat
  GGP := GP; GP := P; P := A;
    if X < A↑.val then A := A↑.g else A := A↑.d;
    if (A↑.g↑.coul = rouge) and (A↑.d↑.coul = rouge) then begin
      if A = Z then {adjonction}
```

```
        begin B := true; new(A);
        with A↑ do begin val := X; g := Z; d := Z; coul := rouge end;
        if X < P↑.val then P↑.g := A else P↑.d := A
        end
        else begin
        A↑.coul := rouge; A↑.g↑.coul := blanc; A↑.d↑.coul := blanc
        end;
      if P↑.coul = rouge then begin {rééquilibrage}
        {8 types de rotation-raccrochage}
      if P↑.val > GP↑.val then begin {rotation gauche ou droite-gauche}
        if A↑.val > P↑.val then RG(GP) else RDG(GP);
        if GP↑.val < GGP↑.val then GGP↑.g := GP else GGP↑.d := GP
        end
      else begin { rotation droite ou gauche- droite}
        if A↑.val < P↑.val then RD(GP) else RGD(GP);
        if GP↑.val < GGP↑.val then GGP↑.g := GP else GGP↑.d := GP
        end;
      {rétablir les couleurs après rotation}
      GP↑.coul := blanc; GP↑.g↑.coul := rouge; GP↑.d↑.coul := rouge;
      {rétablir la hiérarchie des ascendants}
        P := GP; GP := GGP;
      if X = P↑.val then A := P {A renvoie l'adresse de X dans T}
      else if X < P↑.val then A := P↑.g else A := P↑.d
      end
    end
  until X = A↑.val;
  T↑.d↑.coul := blanc {la racine de l'arbre est toujours blanche}
end rech-adj-bic;
```

Analyse de la complexité

Comme pour les AVL, les principaux critères d'évaluation de la complexité des arbres bicolores sont d'une part la forme «équilibrée» de l'arbre, ce qui donne une mesure du coût de recherche d'un élément, et d'autre part le taux de rotations-inversions de couleurs, qui mesure le coût d'une adjonction.

Puisque la hauteur d'un arbre bicolore contenant n éléments est toujours en $\Theta(\log_2 n)$, et que le nombre de rotations-inversions de couleurs est toujours inférieur à la hauteur (exercices), le coût d'une recherche ou d'une adjonction est donc toujours en $\Theta(\log n)$. Pour ce qui est du comportement moyen de ces coûts, aucune analyse complète n'a pu être réalisée jusqu'à présent. Cependant, les résultats expérimentaux montrent des performances analogues à celles des AVL : les arbres bicolores ont une hauteur moyenne égale à $\log_2 n + c$, où c est une constante positive inférieure à 1, et il y a en moyenne une rotation ou une inversion de couleurs pour deux descentes dans l'arbre.

Exercices

1. On appelle suite d'arbres de Fibonacci la suite d'arbres binaires définis de la façon récursive suivante :

$F_0 = \emptyset, F_1 =$ o

et pour $n \geq 2, F_n$ est l'arbre binaire dont le sous-arbre gauche est F_{n-1} et le sous-arbre droit F_{n-2}.

a) Construire F_6.

b) Montrer que les arbres de Fibonacci sont H-équilibrés.

c) Pour tout $n \geq 2$, calculer la hauteur (notée h_n) et le nombre de nœuds (noté N_n) de l'arbre F_n.

d) Les arbres de Fibonacci et les arbres qu'on obtient à partir d'eux par symétries, sont les arbres H-équilibrés qui ont le moins de nœuds pour une hauteur donnée. Montrer que si on supprime un nœud dans un arbre de Fibonacci de hauteur h on obtient :

- soit un arbre H-équilibré de hauteur $h - 1$,
- soit un arbre de hauteur h qui n'est plus H-équilibré.

e) Calculer la longueur de cheminement externe des arbres de Fibonacci.

Existe-t-il des arbres H-équilibrés de hauteur h dont la longueur de cheminement externe est supérieure à celle de l'arbre de Fibonacci de hauteur h ?

2. Ecrire un algorithme de suppression d'un élément dans un AVL.

3. Ecrire des algorithmes d'adjonction d'un élément dans un arbre-2.3.4, dont les nœuds sont représentés :

a) comme sur la figure 28,

b) comme sur la figure 29.

4. Il suffit d'un bit de couleur pour représenter de façon uniforme les trois types de nœuds des arbres 2.3.4. Combien de types de nœuds peut-on représenter en utilisant deux bits de couleur?

5. Lors de la construction d'un arbre bicolore par adjonctions successives, que se passe-t-il si les éléments se présentent en ordre croissant?

6. Etudier l'opération de suppression d'un élément dans les arbres bicolores.

7. Un ***arbre-2.3*** est un arbre de recherche tel que :

- chaque nœud interne a 2 fils ou 3 fils,
- toutes les feuilles sont au même niveau.

Chaque feuille de l'arbre contient un élément de l'ensemble représenté par l'arbre 2.3, et les éléments sont rangés de gauche à droite par ordre croissant dans les feuilles. Chaque nœud interne contient deux valeurs : le plus grand élément du premier sous-arbre de ce nœud, puis le plus grand élément de son second sous-arbre.

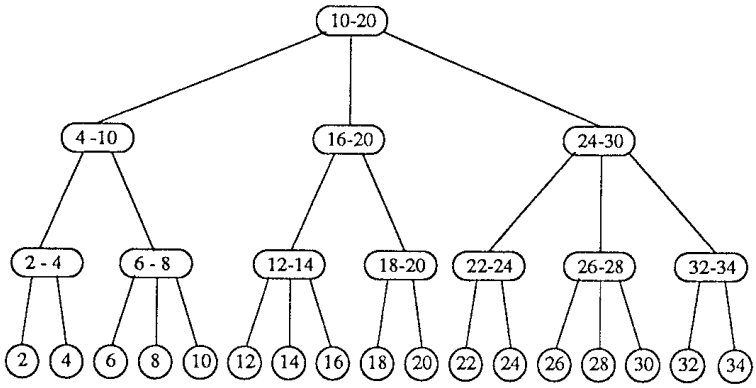


Figure 40. Arbre-2.3.

a) Etant donné un arbre-2.3 contenant i nœuds internes, n feuilles, et de hauteur h, montrer que : $2^{h+1} - 1 \leq n + i \leq (3^{h+1} - 1)/2$ et $2^h \leq n \leq 3^h$.

b) Ecrire un algorithme de recherche d'un élément dans un arbre-2.3 dont tous les éléments sont distincts.

c) Etudier des algorithmes d'adjonction et de suppression d'un élément qui conservent les propriétés d'un arbre-2.3.

Lectures conseillées pour le chapitre 11

Knuth, *The Art of Computer Programming*, Vol. 3 : *Sorting and Searching*, Addison-Wesley, 1973.

Sedgewick, *Algorithms*, Addison-Wesley, 1983.

Wirth, *Algorithms+Data Structures = Programs*, Prentice Hall, 1976.

Chapitre 12
Méthodes de hachage

Dans les méthodes arborescentes vues aux chapitres précédents, il y a un ordre sur les clés, et la place d'un élément dépend du rang de sa clé par rapport aux clés des autres éléments. Dans les méthodes de hachage, par contre, *la place d'un élément est calculée uniquement à partir de sa clé ;* ce calcul est réalisé grâce à une fonction, dite *fonction de hachage,* qui transforme directement la clé en une adresse dans une zone de mémoire contiguë ou en un indice de tableau.

Ces méthodes ont la particularité de permettre les opérations de recherche, d'adjonction et de suppression en un nombre de comparaisons qui est *en moyenne constant,* c'est-à-dire, qui ne dépend pas du nombre d'éléments de la collection !

Elles sont aussi adaptables à la recherche sur mémoire externe.

1. Principes du hachage

1.1. Exemple

Soit E l'ensemble de mots suivant :

$E = \{$serge, odile, luc, anne, annie, jean, julie, basile, paula, marcel, élise$\}$

On considère ici, comme dans les chapitres précédents, que la clé est l'élément lui-même ; un exemple de hachage consiste à associer à chaque élément e de E un nombre $h(e)$, compris entre 0 et 12, en procédant comme suit.

1) Attribuer aux lettres $a, b, c, ..., z$ les valeurs 1, 2, 3,..., 26.

2) Ajouter les valeurs des lettres de e.

3) Ajouter au nombre obtenu le nombre de lettres de e.

4) Calculer ce dernier nombre modulo 13 pour obtenir la valeur cherchée $h(e)$.

En utilisant ce procédé de calcul, on obtient :

h(serge) = (54 + 5) **mod** 13 = 7,

h(odile) = (45 + 5) **mod** 13 = 11,

h(luc) = (36 + 3) **mod** 13 = 0,

h(anne) = (34 + 4) **mod** 13 = 12,

h(annie) = (43 + 5) **mod** 13 = 9,

et de même :

h(jean) = 8, h(julie) = 10, h(basile) = 2,

h(paula) = 4, h(marcel) = 6, h(élise) = 3.

0	luc
1	
2	basile
3	élise
4	paula
5	
6	marcel
7	serge
8	jean
9	annie
10	julie
11	odile
12	anne

Figure 1. Tableau de hachage.

La figure 1 montre le tableau de hachage associé à l'exemple précédent. Dans cet exemple, la restriction de h à E est injective et l'on peut alors ranger chaque élément e à l'adresse $h(e)$.

Pour déterminer si un élément quelconque x appartient à E, il suffit de calculer $v = h(x)$.

• Si $h(x) = 1$ ou $h(x) = 5$ alors x n'est pas dans E; on remarque qu'il faut savoir reconnaître si une place est vide; cela peut se faire en initialisant le tableau par

une valeur spéciale, qui ne peut être un élément, ou en associant à chaque place du tableau un champ qui indique si cette place est vide ou occupée.

- Sinon, on compare x avec l'élément e de E tel que $h(e) = v$
  - si $x = e$ alors x est bien dans E;
  - si $x \neq e$ alors x n'est pas dans E;

Si on veut ajouter x à E, et que l'on est dans le cas ci-dessus (la place d'indice $v = h(x)$ est occupée par un élément $e \neq x$), il y a un problème du fait que h n'est plus injective : $x \neq e$ et $h(x) = h(e) = v$.

On dit alors qu'il y a une ***collision primaire*** entre x et e sur la case v. On verra plus loin qu'il n'est pas possible en général d'éviter les collisions; la fonction de hachage donne alors accès non plus à un élément, mais à un (petit) sous-ensemble d'éléments dans lequel il faut faire une recherche : on a «haché» l'ensemble des éléments en petits morceaux.

1.2. Principe général

Les méthodes de hachage sont utilisables dans les cas suivants : on doit gérer une collection C, dont les clés appartiennent à un univers U très grand; la taille probable de C est connue et est relativement petite par rapport au nombre d'éléments de U.

Par exemple, C est un ensemble de 1000 mots de 10 lettres, et U est l'ensemble de tous les mots possibles de 10 lettres : le nombre d'éléments de U est donc $26^{10} \simeq 10^{13}$.

La taille de U interdit en pratique de représenter les collections d'éléments de U en réservant une place en mémoire par clé possible. On utilise alors une ***fonction de hachage h*** qui associe à chaque clé un entier compris entre 1 et m

$$h : U \rightarrow [1, m]$$

où l'entier m est choisi en fonction de la taille prévue de la collection C.

Pour toute clé $x, h(x)$ appelée ***valeur de hachage primaire,*** donne l'indice de la place de x dans un tableau T de m éléments. Cet indice sert à vérifier si x appartient à T, à l'ajouter ou à le supprimer.

Le choix de la fonction de hachage est fondamental : il faut appliquer U de manière aussi uniforme que possible sur $[1, m]$ pour tendre vers la répartition idéale :

$$|h^{-1}(i)| = \frac{|U|}{m}$$

c'est-à-dire, en termes de probabilité :

$$\forall x \in U \text{et } \forall i \in [1, m], \text{Proba}(h(x) = i) = \frac{1}{m}$$

On dit alors que h est ***uniforme.***

De plus le calcul de la fonction de hachage doit être rapide pour ne pas affaiblir les performances de la méhode.

Pour ces raisons, la construction de la fonction de hachage est un problème délicat (qui est discuté au paragraphe suivant). Mais même avec une «bonne» fonction de hachage, *il est impossible en général d'éviter les collisions primaires* comme le montre l'argument ci-dessous.

Etant donné un ensemble E de n éléments distincts et une fonction h à valeurs dans $[1, m]$, sous l'hypothèse que h est uniforme, la probabilité pour que h soit injective est $P = \frac{1}{m^n} \cdot m \cdot (m-1)...(m-n+1)$.

La valeur de P est petite lorsque m n'est pas très grand par rapport à n.

Par exemple, si $m = 365$ et $n = 23$ alors P est inférieure à $1/2$. On donne souvent l'interprétation suivante : si l'on réunit plus de 23 personnes, il y a plus d'une chance sur deux que deux d'entre elles soient nées le même jour du même mois. En termes de hachage, cela veut dire que pour maintenir, avec une probabilité supérieure à $1/2$, un ensemble de 23 éléments sans collision, il faut un tableau de dimension supérieure à 365.

Il est donc nécessaire de savoir gérer les collisions. Les diverses méthodes de hachage que l'on présente dans la suite diffèrent par la façon dont elles résolvent les collisions; il y a deux classes de méthodes :

- *Les méthodes de résolution des collisions par chaînage* : les éléments dont la clé a la même valeur par la fonction h sont chaînés entre eux, à l'intérieur ou à l'extérieur du tableau. On parle alors de *hachage indirect.*
- *Les méthodes de résolution des collisions par calcul* : lorsqu'il y a collision, on calcule une nouvelle place dans le tableau à partir de la clé de l'élément considéré (pour le rechercher, l'ajouter ou le supprimer). On parle alors de *hachage direct.*

On peut résumer le *principe d'une recherche* dans le cas du hachage comme suit : déterminer la première place possible pour l'élément cherché, par la fonction de hachage; si cette place est vide, arrêter la recherche, sinon comparer le contenu de cette place avec l'élément cherché; si on a trouvé l'élément, arrêter la recherche, sinon passer à une autre place possible, par chaînage ou par calcul, et recommencer le même traitement pour cette nouvelle place.

De même, une *adjonction* commence par déterminer la première place possible pour l'élément cherché; si cette place est vide on y met l'élément et on s'arrête; si on a trouvé l'élément, on s'arrête, sinon on passe à une autre place possible, par chaînage ou par calcul, et on recommence.

Avec toutes ces méthodes, la recherche d'un élément nécessite en moyenne entre 1 et 5 accès (quelle que soit la taille du tableau !). Ce résultat vient de ce *qu'il y a très peu de collisions primaires* (il y a forcément des collisions, mais très peu !), et que la résolution de ces collisions n'en entraîne pas trop d'autres.

Le fait qu'il y a très peu de collisions primaires est illustré par la propriété suivante, dont la preuve est reportée en exercice : étant donné une fonction de hachage uniforme h, à valeurs dans $[1,m]$, et un tableau de m places contenant n éléments répartis selon h, soit v un indice quelconque compris entre 1 et m, on note $P(k)$ la probabilité qu'il y ait exactement k clés dont la valeur par la fonction h est v; $P(0)$ correspond à la probabilité que la place considérée soit vide ; P(k) suit une loi limite dite de Poisson : $P(k) = \mathrm{e}^{-\alpha}\frac{\alpha^k}{k!}$, où α est le rapport $\frac{n}{m}$, appelé ***taux de remplissage du tableau****.*

Cette valeur de $P(k)$ est calculée en faisant tendre m et n vers l'infini, tout en gardant leur rapport α fixe .

Le tableau suivant montre la valeur des probabilités $P(k)$ pour $k = 0,1,...,5$ dans le cas où $\alpha = 0,5$, et dans le cas où $\alpha = 1$.

α \ k	0	1	2	3	4	5
0,5	0,607	0,303	0,076	0,013	0,002	0,000
1	0,368	0,368	0,184	0,061	0,015	0,003

2. Fonctions de hachage

Une fonction de hachage transforme la clé d'un élément à ranger dans un tableau (ou une zone de mémoire) en un indice du tableau (ou en une adresse de la mémoire). Cette fonction distribue les clés de façon aléatoire, mais elle doit évidemment être déterministe : la valeur de hachage d'une clé donnée doit toujours être la même pour pouvoir retrouver l'élément correspondant ! De plus, on a vu que pour limiter le plus possible les collisions, on attend de cette fonction qu'elle soit uniforme.

Enfin, la fonction de hachage doit être facilement calculable, pour que la méthode conserve ses performances en temps.

Il est important de comprendre aussi que le choix d'une «bonne» fonction dépend de l'ensemble des clés sur lesquelles on travaille réellement : considérons la fonction qui associe à une chaîne de caractères la représentation binaire de ses deux premières lettres; cette fonction est uniforme sur l'univers des 26^{10} mots de dix lettres mais elle est mal adaptée pour traiter l'ensemble des prénoms du calendrier, ou pour gérer la table des identificateurs d'un programme FORTRAN où tous les identificateurs d'entiers doivent commencer par I, J, K, L, M, N.

On donne ici quelques principes de construction de fonctions de hachage qui permettent de dégager les techniques utiles et les pièges à éviter. Dans une application pratique, on peut combiner ces différents principes pour trouver une fonction adéquate à l'ensemble des clés traitées.

On suppose que les clés sont des mots. Elles sont représentées en mémoire par une suite de bits. Cette suite de bits peut être interprétée comme un entier. Les fonctions de hachage utilisent beaucoup cette interprétation car elle permet de passer sans calcul des clés aux entiers. Le problème est ensuite de calculer à partir de ces entiers un entier de l'intervalle $[1, m]$ ou plutôt $[0, m-1]$. Les bornes de l'intervalle sont souvent décalées de 1 pour faciliter le calcul de la fonction h : pour se ramener à l'intervalle $[1, m]$ il suffit d'ajouter 1.

Pour simplifier les calculs dans ce qui suit, on va considérer qu'on a une représentation ordinale des caractères codée sur 5 bits; un mot est représenté par la concaténation des représentations de ses caractères; on a donc les correspondances suivantes :

$$A = 00001,\ B = 00010,\ C = 00011,\ \ldots$$

$$LES = 011000010110011$$

On va alors construire une fonction h des suites de bits dans un intervalle de m entiers.

$$h : \{0,1\}^* \to [0, m-1]$$

2.1. Extraction

On extrait certains bits, ou certaines zones de bits, de la représentation binaire; si on extrait p bits, on se ramène à l'intervalle $[0, 2^p - 1]$ en les concaténant et en considérant l'entier représenté.

Exemple : Extraction du sous-mot formé par les bits numérotés 1, 2, 7 et 8 en comptant à partir de la droite, et en complétant à gauche par des zéros.

$ET = 00\mathbf{10}1101\mathbf{00}$ d'où $h(ET) = 1000 = (8)_{10}$

$OU = 01\mathbf{11}1101\mathbf{01}$ d'où $h(OU) = 1101 = (13)_{10}$

$NI = 01\mathbf{11}001\mathbf{01}$ d'où $h(NI) = 1101 = (13)_{10}$

$IL = 01\mathbf{00}101\mathbf{00}$ d'où $h(IL) = 0000 = (0)_{10}$

Cette méthode de calcul est évidemment très facile à mettre en œuvre sur ordinateur. Dans le cas où $m = 2^p$, un sous-mot formé de p bits fournit directement un indice dans le tableau. Elle n'est cependant bien adaptée que pour certains cas particuliers où l'on connaît les données à l'avance, ou alors quand on sait que certains bits ne sont pas significatifs.

En général, on constate qu'elle ne donne pas de bons résultats car la valeur de hachage ne dépend pas de la clé toute entière : *une bonne fonction de hachage doit faire intervenir tous les bits de la représentation.*

2.2. Compression

On utilise ici tous les bits de la clé pour obtenir un indice du tableau. On peut, par exemple, couper les chaînes de bits en morceaux d'égale longueur et additionner ces morceaux ; pour éviter les débordements et les retenues, on préfère en fait l'opération booléenne *ou exclusif* à l'addition.

L'avantage du *ou exclusif*, par rapport aux autres opérations logiques (*et, ou*) est qu'il ne rend pas systématiquement un nombre binaire plus petit que ses arguments (cas du *et*) ou plus grand (cas du *ou*). L'utilisation des opérations *et/ou* provoque en effet des accumulations en début/fin de tableau.

Exemple

$h(ET) = 00101$ **xor** $10100 = 10001 = (17)_{10}$

$h(OU) = 01111$ **xor** $10101 = 11010 = (26)_{10}$

$h(NI) = 01110$ **xor** $01001 = 00111 = (7)_{10}$

$h(CAR) = 00011$ **xor** 00001 **xor** $10010 = 10000 = (16)_{10}$

L'inconvénient de cette méthode est de hacher de la même façon toutes les permutations d'un même mot. Avec la fonction ci-dessus, on a : $h(CAR) = h(ARC)$.

Cela vient de ce que, lors du calcul de h, on a coupé les chaînes aux limites des représentations des caractères : *une bonne fonction de hachage doit briser les sous-chaînes de bits.* On peut résoudre le problème par des décalages : par exemple, on peut décaler circulairement vers la droite la $1^{ière}$ sous-chaîne de 1 bit, la $2^{ième}$ sous-chaîne de 2 bits, la $3^{ième}$ sous-chaîne de 3 bits... et dans ce cas :

$h(CAR)$ = 10001 **xor** 01000 **xor** 01010 = 10011 = $(19)_{10}$

et $h(ARC)$ = 10000 **xor** 10100 **xor** 01100 = 01000 = $(8)_{10}$

Ce type de technique est souvent utilisé pour réduire la taille des clés à celle d'un mot-mémoire. On peut ensuite enchaîner avec d'autres méthodes de calcul telles que celles qu'on présente maintenant.

2.3. Division

On calcule tout simplement le reste dans la division par m (m est la taille du tableau de hachage) de la valeur de la clé :

$$h(c) = c \textbf{ mod } m$$

Une telle fonction de hachage est facile et rapide à calculer. Mais elle dépend de la valeur de m : si, par exemple, m est pair, toutes les clés paires vont dans les indices pairs de la table, et toutes les clés impaires dans les indices impairs, ce qui est à éviter si la répartition des clés selon leur parité n'est pas uniforme. Et il en est de même si m a d'autres petits diviseurs. On est donc conduit à choisir m *premier* mais là encore il peut y avoir des phénomènes d'accumulation (cf. exercices).

Exemple : $m = 37$

$ET = 0010110100 = (180)_{10}$ d'où $h(ET) = 180$ **mod** 37 = 32

$OU = 0111110101 = (501)_{10}$ d'où $h(OU) = 501$ **mod** 37 = 20

2.4. Multiplication

Etant donné un nombre réel θ, tel que $0 < \theta < 1$, on construit une fonction de hachage de la façon suivante :

$$h(e) = \lfloor((e * \theta) \textbf{ mod } 1) * m\rfloor$$

C'est-à-dire que l'on fait le produit de e par θ, on garde la partie décimale (c'est ce qui est réalisé par l'opération «**mod** 1»), puis on multiplie par la taille du tableau et l'on prend la partie entière du nombre obtenu.

Exemple : $\theta = 0,6125423371$ et $m = 30$

$$h(ET) = \lfloor((180 * \theta) \textbf{ mod } 1) * m\rfloor = \lfloor(110,25762068 \textbf{ mod } 1) * 30\rfloor$$
$$= \lfloor 0,25762068 * 30\rfloor = 7$$

Avec cette méthode la taille du tableau est sans importance; par contre, la valeur de θ doit être choisie avec soin : par exemple, θ ne doit pas être pris trop près de 0 ni de 1 pour éviter les accumulations aux extrémités du tableau. On trouvera en exercice une étude des valeurs de θ à éviter. Des études théoriques montrent que les valeurs de θ qui répartissent les clés le plus uniformément sont :

$$\theta = \frac{\sqrt{5}-1}{2} \simeq 0,6180339887 \text{ et } \theta = 1 - \frac{\sqrt{5}-1}{2} \simeq 0,3819660113$$

En conclusion, il n'y a pas de fonction de hachage universelle. Une bonne fonction de hachage doit être rapide à calculer et répartir uniformément les clés, mais ces deux critères dépendent à la fois de la représentation des clés, et de *l'application à traiter.*

3. Résolution des collisions par chaînage : méthodes indirectes

On suppose à présent qu'on dispose d'une fonction de hachage uniforme et adéquate à l'application traitée. De plus, afin de simplifier la présentation des algorithmes, on identifie la clé et l'élément.

Plusieurs stratégies sont alors possibles pour résoudre les collisions. Comme on l'a dit précédemment, on peut chaîner entre eux les éléments en collision; on parle alors de ***méthodes de hachage par chaînage*** ou ***méthodes indirectes.*** Le chaînage peut être fait dans une zone de débordement à l'extérieur du tableau de hachage (méthode de hachage avec *chaînage séparé*), ou à l'intérieur même du tableau de hachage (hachage *coalescent*). Ce sont ces méthodes qui vont être étudiées maintenant.

Une autre classe de méthodes, où l'on calcule un nouvel emplacement dans le tableau de hachage lorsqu'il y a collision (on parle alors de ***méthodes de hachage par calcul*** ou ***méthodes directes***) est étudiée au paragraphe suivant.

3.1. Hachage avec chaînage séparé

Dans cette méthode, tous les éléments en collision sont chaînés entre eux à l'extérieur du tableau de hachage : le tableau contient seulement les têtes de m listes chaînées représentant chacune un ensemble d'éléments qui rendent une même valeur par la fonction de hachage. Les algorithmes de recherche, adjonction et suppression sont très proches de ceux vus au chapitre 6 pour des ensembles représentés par des listes chaînées. L'adjonction est précédée d'un test d'appartenance car il est essentiel de garder les listes aussi courtes que possibles. On prend la convention de réaliser l'adjonction en fin de liste car c'est là qu'on se trouve à l'issue du test d'appartenance; mais on pourrait aussi, bien sûr, faire l'adjonction en début de

liste. On note L_i la liste dont la tête est $T[i]$; pour rechercher, supprimer ou ajouter l'élément x on travaille avec la liste $L_{h(x)}$.

Exemple : La figure 2 montre le résultat du hachage par adjonctions successives des éléments :

e1, e2, e3, e4, e5, e6, e7, e8, e9, e10, e11, e12, e13

ayant respectivement pour valeur de hachage

3, 1, 4, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5

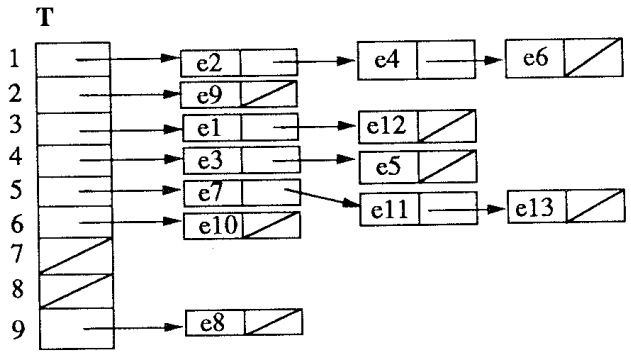


Figure 2. Hachage avec chaînage séparé.

Remarque : On pourrait représenter différemment les éléments en collision (listes chaînées triées, arbres binaires de recherche, arbres équilibrés). Ces différentes stratégies sont laissées en exercice : elles ont peu d'intérêt car, lorsque la fonction de hachage est bien choisie, on a vu qu'il est très peu probable d'avoir plus de cinq éléments en collision sur la même valeur de hachage.

3.1.1. Analyse de la complexité du hachage avec chaînage séparé

Du point de vue de la place en mémoire, cette méthode nécessite m cases pour ranger les m pointeurs vers la zone de débordement, plus autant de doublets <élément, pointeur> que d'éléments.

Les algorithmes de recherche, adjonction et suppression, font un calcul de la valeur de hachage, puis des comparaisons entre éléments en parcourant la liste ainsi déterminée. Comme pour les autres représentations vues dans les chapitres précédents, on va, dans un premier temps, analyser ces algorithmes en considérant comme fondamentale l'opération de comparaison entre éléments, sachant qu'à ces comparaisons s'ajoute le calcul de la fonction de hachage qui est indépendant du nombre n d'éléments et de la taille m du tableau. On verra ensuite que le choix de cette opération fondamentale n'est pas toujours bon dans le cas du hachage.

On considère un ensemble de n éléments distincts $e_1, ..., e_n$ représenté par hachage avec chaînage séparé à partir d'un tableau $T[1..m]$, et obtenu uniquement par des adjonctions. On étudie le coût moyen de la recherche d'un élément dans T.

3.1.2. Complexité en moyenne d'une recherche négative

On peut remarquer qu'une recherche négative a le même coût en nombre de comparaisons entre éléments que l'adjonction d'un nouvel élément car celle-ci est précédée d'un test d'appartenance et se fait en fin de liste.

La fonction h étant uniforme, il y a la même probabilité $1/m$ d'effectuer la recherche dans chacune des m listes. Soit λ_i la longueur de la liste L_i, le coût moyen d'une recherche négative dans un tableau T est :

$$Rech^-(T) = \frac{1}{m} \sum_{i=1}^{m} \lambda_i$$

En effet, on va jusqu'au bout de chaque liste.

Comme il y a en tout n éléments, on a $\sum_{i=1}^{m} \lambda_i = n$, et donc $Rech^-(T) = \frac{n}{m} = \alpha$.

Considérons maintenant le coût moyen d'une recherche négative quand on a n éléments dans un tableau T quelconque de m listes. Les m^n tableaux possibles sont équiprobables, puisque la fonction de hachage h est uniforme. Ce coût moyen s'écrit :

$$\text{Moy}_{Rech-}(m, n) = \frac{1}{m^n} \sum_{T} Rech^-(T),$$

la somme étant prise sur tous les tableaux T possibles. On en déduit que :

$$\text{Moy}_{Rech-}(m, n) = \frac{n}{m} = \alpha$$

3.1.3. Complexité en moyenne d'une recherche positive

Le tableau T étant obtenu uniquement par des adjonctions en fin de listes, la position d'un élément par rapport à la tête de sa liste ne change jamais. De ce fait, le coût d'une recherche positive est égal, en nombre de comparaisons entre éléments, au coût de l'adjonction de cet élément quand elle a eu lieu, augmenté de 1 : en effet on fait des comparaisons avec les éléments de la liste situés avant l'élément cherché, et qui étaient donc présents au moment de l'adjonction de celui-ci, et on fait une comparaison supplémentaire avec l'élément lui-même, ce qui arrête la recherche. Si l'élément recherché a été le $(i+1)^{ième}$ ajouté, son adjonction a eu le même coût

qu'une recherche négative parmi i éléments, soit en moyenne :

$$\mathrm{Moy}_{Rech-}(m,i) = \frac{i}{m}$$

On considère que les n éléments présents ont la même probabilité d'être recherchés. On en déduit :

$$\mathrm{Moy}_{Rech+}(m,n) = \frac{1}{n}\sum_{i=0}^{n-1}(\mathrm{Moy}_{Rech-}(m,i)+1)$$

d'où

$$\mathrm{Moy}_{Rech+}(m,n) = \frac{i}{n}\sum_{i=0}^{n-1}\left(\frac{i}{m}+1\right) = \frac{n(n-1)}{2nm}+1 = \frac{\alpha}{2}-\frac{1}{2m}+1.$$

On voit donc que, pour le hachage avec chaînage séparé, dans le cas où on ne fait pas de suppressions, la recherche et l'adjonction sont en $\Theta(\frac{n}{m})$ en nombre de comparaisons entre éléments, m étant le nombre de listes et n le nombre d'éléments.

3.1.4. Choix de l'opération fondamentale

Une recherche dans le cas du hachage est une suite d'étapes qui consiste à accéder à une place possible pour l'élément cherché; si cette place est vide, arrêter la recherche, sinon comparer le contenu de cette place avec l'élément cherché; si on a trouvé l'élément, arrêter la recherche, sinon passer à une autre place possible.

Le test qu'une place est vide est donc une opération importante. Dans le cas du hachage indirect, ce test de place vide est un test d'égalité de pointeur à *nil*. Le nombre des tests d'égalité de pointeur avec *nil* est égal, dans le cas d'une recherche négative ou d'une adjonction, au nombre des comparaisons entre éléments plus 1; dans le cas d'une recherche positive, il est égal au nombre de comparaisons entre éléments. Ce nombre est donc égal en moyenne à $\alpha+1$ pour une recherche négative et à $\frac{\alpha}{2}-\frac{1}{2m}+1$ pour une recherche positive.

On veut éviter de compter tantôt le nombre de comparaisons entre éléments et tantôt le nombre de tests d'égalité de pointeur à *nil*, car ces nombres peuvent différer de 1. Pour cela on considère globalement une seule opération, l'***examen d'une place*** p par rapport à un élément e : cette opération teste si la place p est vide, et si ce n'est pas vrai elle compare e et le contenu de p. C'est cette opération qui est considérée comme fondamentale dans les analyses des algorithmes du reste de ce chapitre.

Remarque : On peut vouloir calculer le coût moyen d'une recherche positive pour un tableau T comme on l'a fait pour une recherche négative.

On a alors :

$$Rech^+(T) = \frac{1}{n}\sum_{j=1}^{n} coût(e_j)$$

où $coût(e_j)$ est égal au nombre de comparaisons pour trouver e_j dans la liste $L_{h(e_j)}$ du tableau T.

En regroupant dans la somme précédente tous les éléments d'une même liste, on obtient :

$$Rech^+(T) = \frac{1}{n}\sum_{i=1}^{m}\sum_{h(e)=i} coût(e),$$

or, d'après la définition du coût , on a :

$$\sum_{h(e)=i} coût(e) = 1 + 2 + ... + \lambda_i$$

D'où :

$$Rech^+(T) = \frac{1}{n}\sum_{i=1}^{m}\frac{\lambda_i.(\lambda_i + 1)}{2}$$

On voit que le coût moyen d'une recherche positive dépend de la taille des listes (alors que le coût moyen d'une recherche négative ne dépend que de la dimension m du tableau, et du nombre d'éléments n qu'on y a insérés) ; donc ce coût dépend de la façon dont les n éléments sont distribués par la fonction h dans les m listes.

Dans le pire des cas, si les n éléments ont tous la même valeur de hachage, le coût moyen d'une recherche positive est $\frac{1}{n}.\frac{n(n+1)}{2}$, soit $\frac{n+1}{2}$. Mais on montre qu'en moyenne il est constant. En effet, sous l'hypothèse que la fonction de hachage est uniforme, il y a équiprobabilité pour les m^n répartitions possibles. La complexité en moyenne d'une recherche positive pour un hachage avec chaînage séparé s'exprime alors sous la forme

$$\text{Moy}_{Rech+}(m,n) = \frac{1}{m^n}\sum_{T} Rech^+(T),$$

la somme étant prise sur tous les tableaux T de m listes avec chaînage séparé contenant en tout n éléments.

Rappelons qu'il y a $\frac{n!}{\lambda_1!.\lambda_2!...\lambda_m!}$ façons de répartir n éléments en m sous-ensembles de tailles respectives $\lambda_1, \lambda_2, ..., \lambda_m$.

En regroupant, dans la somme précédente, les tableaux selon les tailles des listes $L_1, ..., L_m$, on obtient :

$$\text{Moy}_{Rech+}(m,n) = \frac{1}{m^n} \sum_{\lambda_1+...+\lambda_m=n} \frac{n!}{\lambda_1!.\lambda_2!...\lambda_m!}.Rech^+(T_{\lambda_1,...,\lambda_m})$$

où $T_{\lambda_1,...,\lambda_m}$ est un tableau quelconque tel que la liste L_i soit de longueur h_i (voir exercice pour un calcul direct par des séries génératrices).

3.2. Hachage coalescent

La méthode de hachage avec chaînage séparé se programme simplement lorsqu'il est possible d'allouer de la mémoire en cours d'exécution. Dans de nombreux contextes d'utilisation, une telle allocation dynamique de mémoire n'est pas possible. Il faut alors réserver *a priori* une zone contiguë de mémoire, et gérer l'ensemble des éléments à l'intérieur de cette zone de taille fixe. On considère que cette zone est un tableau de taille m. Dans ce cas, on peut diviser le tableau en deux zones : *une zone d'adresses primaires* de capacité p, et une *réserve* pour gérer les collisions, de capacité r, telles que $p + r = m$. Les entiers p et r sont fixés *a priori* et la fonction de hachage est à valeurs dans $[1, p]$. On remarque que le nombre d'éléments présents n, est inférieur ou égal à m.

Exemple : La figure 3 représente un tableau de dimension $m = 9$, dont les trois dernières cases forment la réserve : $p = 6$, $r = 3$. On a ajouté successivement les éléments :

e1, e2, e3, e4, e5, e6, e7

ayant pour valeurs de hachage respectives :

1, 4, 1, 4, 2, 1, 3

La réserve est utilisée ici par ordre d'adresses décroissantes, et l'on a chaîné en une même liste tous les éléments ayant une même valeur de hachage primaire.

Si on ajoute un nouvel élément dont la valeur de hachage est 1, 2, 3 ou 4, on ne peut pas l'ajouter car la réserve est pleine et pourtant il reste de la place dans la zone d'adresses primaires. On voit apparaître là le problème de la détermination des tailles respectives de la zone d'adresses primaires et de la réserve : si la réserve est petite, elle est trop vite remplie et l'utilisation du tableau est incomplète; si la réserve est grande, on peut mieux remplir le tableau, mais on diminue l'effet de dispersion du hachage (à la limite on peut n'avoir qu'une seule liste si la zone d'adresses primaires est réduite à une case !).

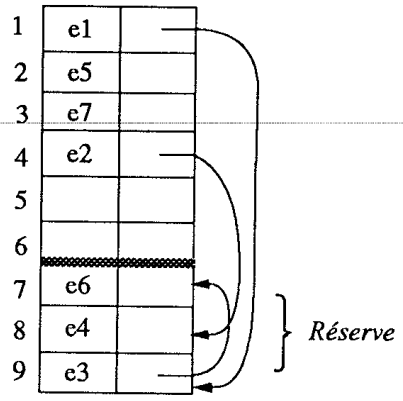


Figure 3. Hachage avec réserve m=9, r=3.

En fait, pour pouvoir remplir au mieux le tableau, on peut penser à ne pas faire de distinction entre la zone d'adresses primaires et la réserve : chacune des places peut soit être atteinte par hachage primaire, soit servir à résoudre une collision. Mais cette technique a l'inconvénient de créer des ***collisions secondaires,*** c'est-à-dire des collisions qui ne sont pas liées à la coïncidence des valeurs de la fonction de hachage.

La figure 4 montre la situation d'un tableau de taille $m = 11$ dans lequel on a ajouté les éléments :

e1, e2, e3, e4, e5, e6, e7, e8, e9, e10

de valeurs de hachage respectives :

1, 5, 2, 5, 1, 8, 11, 9, 5, 10

Si un élément est en collision, on le place (par exemple) à la première place vide à partir de la fin du tableau. L'élément e4 est en collision avec e2, on le place donc dans la place d'indice 11 ; lorsque l'élément e7, dont la valeur de hachage est 11, doit être inséré, il y a collision et on le met alors dans la première place vide à partir de la fin du tableau, c'est-à-dire la place d'indice 9, mais il faut chaîner cette place avec la place d'indice 11. Dès lors les listes des éléments correspondant aux valeurs de hachage primaire 5 et 11 sont fusionnées. C'est ce phénomène qui est à l'origine du nom donné à cette méthode : *hachage avec coalescence de listes, ou* ***hachage coalescent.*** Les collisions secondaires, dues à la coalescence des listes, rendent le temps de recherche plus long que lorsque le chaînage est à l'extérieur du tableau, mais on verra que la méthode est quand même très performante.

L'algorithme d'adjonction dans le cas du hachage coalescent est donné ci-dessous. Le tableau est constitué de m enregistrements à deux champs : un champ *val* contenant l'élément et un champ *lien* permettant les chaînages ; on suppose connue la fonction de hachage h qui est à valeurs dans $1..m$. On utilise le type Pascal suivant :

1	e1	10
2	e3	0
3		
4	e10	0
5	e2	11
6	e9	0
7	e8	6
8	e6	0
9	e7	7
10	e5	4
11	e4	9

Figure 4. Hachage coalescent.

type tab = **array** $[1..m]$ **of record** : *val* : Elément; *lien* : $0..m$ **end**;

On utilise également une fonction *vide* qui permet de tester si une place est occupée ou non :

function *vide*(T : tab; i : $1..m$) : boolean;

On fait pour l'instant abstraction de la manière dont est programmé ce test. Comme on l'a déjà dit, on peut avoir initialisé les champs *val* du tableau à une valeur particulière qui ne peut être un élément; on peut aussi avoir ajouté un champ aux enregistrements du tableau, champ qui indique si une place est vide ou occupée; dans ce cas il faut initialiser ce champ à *vide*, et lui donner la valeur occupée quand on met un élément dans une place. On revient sur ce problème dans le paragraphe 4.4.

```
procedure ajouter_HCO(x : Elément; var T : tab; var plein : boolean);
{On recherche l'élément x dans le tableau T, et dans le cas où la recherche
est négative, on l'ajoute à T ; plein est un booléen d'alarme : il prend la
valeur true si l'adjonction n'a pu se faire car le tableau est plein}
var R : 0..m; i : 1..m;
{R sert à rechercher la case libre pouvant servir à résoudre une collision}
begin
    {h est une fonction de hachage, déclarée ailleurs}
    i := h(x); R := m; plein := false;
    if vide(T, i)
    then begin {on ajoute l'élément}
        T[i].val := x; T[i].lien := 0
        {éventuellement, marquer la place d'indice i comme occupée}
    end
```

```
        else begin
                while (T[i].val <> x) and (T[i].lien <> 0) do i:= T[i].lien;
                if T[i].val = x then {élément trouvé} return;
                while (R ≥ 1) and (not vide(T, R)) do R := R − 1;
                if R ≥ 1 then begin
                        T[i].lien := R ; T[R].val := x; T[R].lien := 0
                end
                else plein :=true {le tableau est plein}
        end
end ajouter_HCO;
```

Pour des raisons de clarté, R est une variable locale initialisée à m au début de la procédure, mais il serait plus judicieux de conserver R avec le tableau T (sous forme d'un enregistrement à deux champs), cela éviterait, lorsqu'on fait une adjonction, de reparcourir à chaque fois tout le tableau à partir de la fin.

3.2.1. Analyse de la complexité du hachage coalescent (cas sans réserve)

L'évaluation des performances du hachage coalescent est assez compliquée; on peut montrer que lorsque la fonction de hachage est uniforme, le nombre moyen d'examens de places pour un ensemble de n éléments distincts dans un tableau de m cases vaut asymptotiquement, pour m et n grands, le rapport $\alpha = n/m$ restant fixé (et étant par construction du tableau inférieur à 1) :

– dans le cas d'une recherche positive

$$\text{Moy}_{Rech+}(m,n) \sim \frac{1}{8\alpha}(e^{2\alpha} - 1) + \frac{\alpha}{4} + \frac{3}{4}$$

– dans le cas d'une recherche négative

$$\text{Moy}_{Rech-}(m,n) \sim \frac{1}{4}e^{2\alpha} - \frac{\alpha}{2} + \frac{3}{4}$$

On déduit par exemple de ces formules qu'une recherche négative (respectivement positive) dans un tableau plein ($\alpha = 1$) ne coûte qu'environ 2.1 examens de places (respectivement 1.7 examens de places) en moyenne.

3.2.2. Suppression dans le cas du hachage coalescent

La suppression d'un élément dans un ensemble représenté par hachage coalescent est un problème plus compliqué que l'adjonction : considérer par exemple la suppression de l'élément e4 dans le tableau de la figure 4; cette suppression entraîne le décalage des éléments e7, e8 et e9.

En fait, on verra que *toutes les méthodes de hachage* (sauf la méthode avec chaînage séparé) sont mal adaptées aux suppressions d'éléments, celles-ci pouvant entraîner

une réorganisation importante du tableau. Plutôt que de supprimer «physiquement» un élément, on préfère *souvent marquer* sa place comme libérée. Une place libérée n'est pas une place vide : cela aurait pour effet de rompre la chaîne qui lie les éléments en collisions. On peut cependant réutiliser les cases marquées comme libérée pour l'adjonction d'un nouvel élément. On revient en détail sur ce problème dans le cas du hachage direct, au paragraphe 4.4.

4. Résolution des collisions par calcul : méthodes directes

4.1. Principe général

Dans les méthodes vues précédemment, les liens occupent de l'espace en mémoire; il peut paraître préférable d'utiliser cet espace pour ranger les éléments eux-mêmes, de façon à diminuer les risques de collision. On est alors conduit à rechercher des méthodes de résolution des collisions par *calcul* à l'intérieur du tableau. Notons que comme dans le cas du hachage coalescent, si m est la taille du tableau et n le nombre d'éléments présents, on a forcément $n \leq m$.

La réalisation de ces méthodes repose sur la construction d'une fonction, dite *fonction des essais successifs :*

$$essai : U \rightarrow \{1, 2, ..., m\}^m$$

qui associe à chaque élément x de l'univers U une permutation de $\{1, 2, ..., m\}$:

$$essai(x) = (essai_1(x),\ essai_2(x),\ ...,\ essai_m(x))$$

avec $essai_i(x) \in \{1, ..., m\}$ pour tout i, et si $i \neq j$ alors $essai_i(x) \neq essai_j(x)$.

Lorsqu'on recherche x dans le tableau $T[1..m]$, on explore successivement les places $essai_1(x)$ (valeur de hachage primaire) puis $essai_2(x)$, $essai_3(x)$... jusqu'à ce qu'on trouve x, ou qu'on arrive sur une place vide ou à la fin des m essais, auquel cas on peut conclure que x n'est pas dans le tableau. Pour faire une adjonction on suit la même démarche; si on trouve x, on ne fait rien, sinon on l'ajoute à la place vide; si on est au bout des m essais, c'est que le tableau est plein et que l'adjonction est impossible.

Les procédures de recherche et d'adjonction sont données ci-dessous pour le cas où on ne fait jamais de suppression (le cas des suppressions est étudié au §4.4). On suppose connue une fonction *essai* qui calcule les valeurs de $essai_i(x)$:

function *essai*(i : 1..m; x : Elément) : 1..m; $\{essai(i, x) = essai_i(x)\}$

On travaille sur des tableaux d'éléments :

type tab = **array** [1..m] **of** Elément;

Comme dans le cas du hachage coalescent, on se donne une fonction qui permet de tester si une place du tableau est vide :

```
function vide(T : tab ; i : 1..m) : boolean;

function rechercher_HD1(x : Elément ; T : tab) : integer;
{cette fonction a pour résultat l'indice de x dans T, si x est présent dans T,
et 0 sinon ; on ne fait jamais de suppression dans T}
var i, v : integer;
begin
    i := 1;
    while i ≤ m do begin
            v := essai(i, x); if (vide(T, v)) or (T[v] = x) then exit else i := i + 1
            end;
            if T[v] = x then {x présent} return (v)
            else {x absent} return (0)
end rechercher_HD1;

procedure ajouter_HD1(x : Elément; var T : tab; var plein : boolean);
{si x n'est pas dans T, cette procédure l'ajoute dans T à la première place
libre rencontrée au cours des essais ; si x est dans T, la procédure ne fait
rien ; si le tableau est plein, c'est que x n'a pu être ajouté, plein prend
la valeur true, sinon il vaut false ; on ne fait jamais de suppression dans T}
var i, v : integer;
begin
    i := 1; plein := false;
    while i ≤ m do begin
    v := essai(i, x); if (vide(T, v)) or (t[v] = x) then exit else i := i + 1
    end;
    if T[v] = x then {x déjà présent , on ne fait rien :} return
    else if vide(T, v) then {x absent; on le place dans le tableau :} T[v] := x
        else plein := true
end ajouter_HD;
```

Remarques :

1) Si on peut vérifier que le tableau n'est pas plein avant d'appeler la procédure d'adjonction (par exemple au moyen d'un compteur associé au tableau), celle-ci se simplifie notablement.

2) On peut aussi effectuer la recherche à l'aide d'une procédure qui indique en cas d'échec l'indice de la dernière place examinée : dans ce cas, l'adjonction se programme en appelant cette procédure et, si la recherche a été négative, on met l'élément à ajouter à la place obtenue en résultat.

La suppression dans le cas du hachage direct est présentée à la fin de ce paragraphe, après deux exemples de méthodes directes qui correspondent à des fonctions des essais successifs différentes : le hachage linéaire et le double hachage.

Les différentes méthodes de hachage direct se caractérisent par le choix de la fonction des essais successifs. Cette fonction associe à chaque élément une suite de m places dans le tableau; il y a au plus $m!$ suites différentes dans le tableau, mais la plupart des méthodes de hachage direct en utilisent en réalité beaucoup moins, plusieurs éléments pouvant partager une même suite, ou une même sous-suite : dans le hachage linéaire, qui est présenté au paragraphe suivant, on voit que la suite des places possibles d'un élément dépend uniquement de la valeur de hachage primaire de celui-ci, ce qui n'autorise que m suites différentes dans le tableau.

4.2. Hachage linéaire

Dans cette méthode, lorsqu'il y a collision sur une case d'indice v, on essaie la case d'indice $v+1$ du tableau; si on est à la fin du tableau ($v = m$), on recommence au début. Notons $a \oplus b$ l'opération qui assure cette progression circulaire (elle est définie sur $[1,m] \times [1,m]$; si $a+b \leq m$, elle vaut $a+b$; si $a+b > m$, elle vaut $a+b-m$).

Etant donnée une fonction de hachage uniforme $h : U \rightarrow [1,m]$, on fait donc la suite d'essais suivante :

$essai_1(x) = h(x)$

$essai_2(x) = h(x) \oplus 1$

...

$essai_i(x) = h(x) \oplus i - 1$

...

$essai_m(x) = h(x) \oplus m - 1$

Exemple : L'adjonction successive des éléments :

e1, e2, e3, e4, e5, e6, e7, e8, e9

de valeurs de hachage primaire respectives :

6, 4, 7, 4, 8, 2, 5, 9, 8

dans une table de 10 éléments donne pour résultat la répartition de la figure 5.

Cet exemple met en évidence deux phénomènes caractéristiques du hachage linéaire. Tout d'abord tous les éléments ayant la même adresse de hachage primaire ont aussi exactement la même séquence d'essais successifs : il n'y a aucune dispersion après le premier hachage. De plus, il y a formation de groupements dans le tableau :

après l'ajout de e5, tout élément dont la valeur de hachage primaire est comprise entre 4 et 9 ira se placer à l'adresse 9; et ainsi plus les groupements d'éléments consécutifs sont importants plus ils ont tendance à grossir encore. On remarque en effet que dans l'hypothèse où la fonction de hachage est uniforme, un groupement de k éléments a la probabilité $\frac{k+2}{m}$ de s'accroître lors d'une adjonction.

1	e9
2	e6
3	
4	e2
5	e4
6	e1
7	e3
8	e5
9	e7
10	e8

Figure 5. Exemple de hachage linéaire.

Cette méthode a cependant l'avantage de ne faire intervenir qu'une seule fonction de hachage, les essais successifs étant ensuite très simples à calculer : il suffit de remplacer dans les algorithmes donnés précédemment l'instruction $v := essai(i, x)$ par $v := v \oplus 1$ et de modifier légèrement l'initialisation et la terminaison de la boucle.

```
function rechercher_HL(x : Elément; T : tab) : integer;
{cette fonction a pour résultat l'indice de x dans T, si x est présent dans T,
et 0 sinon}
var i, v : integer;
begin
    {h est une fonction de hachage, déclarée ailleurs}
    i := 1; v := h(x);
    while i ≤ m do
        if (vide(T, v)) or (T[v] = x) then exit
        else begin i := i + 1; v := v + 1; if v > m then v := 1 end;
    if T[v] = x then {x présent} return (v)
    else {x absent} return (0)
end rechercher_HL;
```

De plus, l'analyse montre que l'algorithme est tout à fait utilisable pour des taux de remplissage pas trop élevés. Cette analyse étant complexe, on se contente ici d'en donner les résultats.

Pour une fonction de hachage uniforme, le nombre moyen d'examens de places, pour le hachage linéaire, lorsque m et n tendent vers l'infini, le rapport $\alpha = n/m$ restant fixé, vaut asymptotiquement :

– pour une recherche positive

$$\text{Moy}_{Rech+}(m,n) \sim \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

– pour une recherche négative

$$\text{Moy}_{Rech+}(m,n) \sim \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

On rappelle que par construction on a $n \leq m$, donc $\alpha \leq 1$.

Par exemple, pour un taux de remplissage $\alpha = 0{,}5$, cela donne 1,5 examen de places en moyenne pour une recherche positive, et 2,5 pour une recherche négative ; pour $\alpha = 0{,}8$, on a 3 examens de place pour une recherche positive, et 13 pour une recherche négative. Les recherches négatives sont ici nettement plus coûteuses en moyenne que les recherches positives : intuitivement cela se justifie par le fait que pour vérifier qu'un élément n'est pas dans le tableau, il faut parcourir complètement le groupement auquel il pourrait appartenir, et l'on a vu que plus un groupement est gros plus il a de chances de contenir la valeur de hachage primaire d'un élément.

4.3. Double hachage

Pour éviter la formation de groupements d'éléments contigus, il faut essayer de les disperser plus lorsqu'il y a collision.

Prenons comme suite d'essais :

$essai_i(x) = h(x) \oplus k.(i-1)$, où k est un nombre fixé.

Si k et m sont premiers entre eux, alors la suite des m essais comporte bien toutes les places du tableau. Il se reproduit cependant les mêmes phénomènes de groupements, à présent de k en k, dans le tableau. Pour éviter les groupements, il faut que l'incrément k dépende de l'élément.

On est donc amené à utiliser une deuxième fonction de hachage d, d'où le nom de ***double hachage*** donné à cette méthode. La séquence des essais est alors la suivante :

$$essai_i(x) = h(x) \oplus d(x).(i-1)$$

La fonction d doit être telle que pour tout élément x de U, la suite des m essais considère bien toutes les places du tableau ; cela n'est le cas que si $d(x)$ est premier avec m pour tout x. Cette propriété est satisfaite si on choisit :

- soit m premier, et il suffit alors que d soit à valeurs dans $[1, m-1]$
- soit $m = 2^p$, et il suffit alors que $d(x)$ soit impair pour tout x, ce que l'on peut réaliser en prenant $d(x) = 2d'(x) + 1$, où d' est une fonction à valeurs dans $[0, 2^{p-1} - 1]$.

Les algorithmes de recherche et d'adjonction dans le cas du double hachage se déduisent de ceux du cas général en changeant $v := essai(i, x)$ en $v := v \oplus d(x)$, pour $i > 1$, et en modifiant légèrement l'initialisation et la terminaison de la boucle.

On peut prouver que dans les méthodes de double hachage, les $m!$ permutations de $\{1, 2, ..., m\}$ sont équiprobables comme séquences d'essais successifs. C'est cette propriété qui permet d'analyser la complexité du double hachage.

On se place dans le cas où tous les éléments sont distincts, et où on ne fait que des adjonctions et des recherches.

Comme pour les autres méthodes de hachage, on compte les examens de places.

4.3.1. Analyse du double hachage : recherche négative

Considérons d'abord le nombre moyen d'examens de places $\text{Moy}_{Rech-}(m, n)$ pour une recherche négative par double hachage dans un tableau de m places contenant n éléments.

Soit p_r la probabilité qu'exactement r examens de places soient nécessaires pour vérifier qu'un élément est absent. Du fait de l'uniformité des fonctions de hachage h et d, et de l'équiprobabilité des $m!$ séquences d'essais successifs on a :

$p_1 = \dfrac{m-n}{m}$, probabilité de tomber sur une place vide au premier essai et donc de faire en tout un examen de place ;

$p_2 = \dfrac{n}{m} \cdot \dfrac{m-n}{m-1}$, probabilité de tomber sur une place occupée au premier essai et sur une place libre au deuxième essai et donc de faire en tout deux examens de places ;

...

$p_r = \dfrac{n}{m} \cdot \dfrac{n-1}{m-1} \cdots \dfrac{n-r+2}{m-r+2} \cdot \dfrac{m-n}{m-r+1}$, probabilité de tomber sur $r-1$ places occupées aux $r-1$ premiers essais puis sur une place libre, donc de faire en tout r examens de places.

...

Au pire, on va faire $n+1$ examens de places, puisque n places sont occupées :

$p_{n+1} = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{1}{m-n+1} \cdot \frac{m-n}{m-n}$.

D'où la valeur moyenne du nombre d'examens :

$$\text{Moy}_{Rech-}(m,n) = \sum_{r=1}^{n+1} r.p_r$$

En notant que $p_r = \frac{C_{m-r}^{n-r+1}}{C_m^n}$ (cf. exercices), on montre que :

$$\text{Moy}_{Rech-}(m,n) = \frac{m+1}{m+1-n} = \frac{1}{1-\frac{n}{m+1}} \sim \frac{1}{1-\alpha}$$

4.3.2. Cas de la recherche positive

On fait le même raisonnement que pour le hachage séparé : comme les éléments ne sont plus jamais déplacés après leur adjonction, le nombre d'examens de places pour une recherche positive d'un élément x est égal au nombre d'examens de places effectués quand on a ajouté x. Si x a été le $(i+1)^{ième}$ élément ajouté, son adjonction a été équivalente en nombre d'examens de places à une recherche négative parmi i éléments.

$$\text{Moy}_{Rech+}(m,n) = \frac{1}{n} \sum_{i=0}^{n-1} \text{Moy}_{Rech-}(m,i)$$

$$= \frac{1}{n} \sum_{i=0}^{n-1} \frac{m+1}{m+1-i} = \frac{m+1}{n} \cdot \left(\frac{1}{m+1} + \frac{1}{m} + \ldots + \frac{1}{m-n+2} \right)$$

D'où, si l'on note H_n le $n^{ième}$ nombre harmonique

$$\text{Moy}_{Rech+}(m,n) = \frac{m+1}{n} . (H_{m+1} - H_{m-n+1})$$

et comme H_n est asymptotiquement équivalent à Log n on a :

$$\text{Moy}_{Rech+}(m,n) \sim \frac{m+1}{n} \cdot \text{Log}\left(\frac{m+1}{m-n+1} \right) \sim \frac{1}{\alpha} \text{Log}\left(\frac{1}{1-\alpha} \right)$$

Pour n et m grands et $\alpha = \frac{n}{m}$, les performances du double hachage sont donc asymptotiquement égales à :

$\frac{1}{1-\alpha}$ pour le nombre moyen d'examens de places lors d'une *recherche négative,*

$\frac{1}{\alpha}\,\mathrm{Log}\left(\frac{1}{1-\alpha}\right)$ pour le nombre moyen d'examens de places lors d'une *recherche positive.*

On montre en exercice que la complexité du double hachage est équivalente asymptotiquement à celle d'une autre méthode, proposée dès les premiers temps du hachage, le *hachage avec essais uniformes.* Dans cette méthode, on suppose que l'on peut disposer d'une suite infinie $h_1, h_2...$ de fonctions de hachage qui sont uniformes et indépendantes, et l'on essaie ces fonctions l'une après l'autre. Cette méthode disperse au mieux les éléments, mais elle nécessite le calcul de plusieurs fonctions de hachage, ce qui peut être relativement coûteux d'un point de vue pratique; c'est pourquoi on lui préfère le double hachage.

4.4. Suppression dans le cas du hachage direct

Soit x un élément à supprimer, et soit $v = \mathit{essai}(i, x)$ l'indice de sa place dans le tableau. La suppression d'un élément x devrait entraîner, en toute rigueur, le décalage de tous les éléments qui sont derrière x dans la suite des éléments en collision avec lui. Or c'est très difficile à réaliser : rien ne prouve par exemple que l'élément y contenu en $v' = \mathit{essai}(i+1, x)$ est en collision avec x; v' peut très bien être la valeur de hachage primaire de y, ou correspondre à $\mathit{essai}(k, y)$...

En fait, si la suite des essais dépend de x, ce qui est le cas pour le double hachage, on n'a aucun moyen de retrouver efficacement les éléments en collision avec x.

Il n'est pas possible non plus d'indiquer que la place v est désormais vide car la rencontre d'une place vide arrête la recherche : si z est un élément tel que $v = \mathit{essai}(k, z)$, qui a été placé, suite à des collisions, à la place d'indice $\mathit{essai}(k+1, z)$, on s'arrêterait sur v à la prochaine recherche de z et on considérerait que z n'est pas dans le tableau.

La solution utilisée est, lorsqu'on a fait une suppression, de marquer la place comme libérée, ce qui est différent de vide. Une place libérée n'arrête pas la recherche, mais peut être utilisée pour une adjonction. Une place a donc trois états possible : vide, occupé ou libéré. Une solution est de considérer non plus un tableau d'éléments, mais un tableau d'enregistrements avec les déclarations de types suivantes :

```
type état = (vide, occupé, libéré);
     doublet = record marque : état; val : Elément end;
     tab = array [1..m] of doublet;
```

Le tableau est initialisé avec toute les marques à vide. La fonction *vide* est définie par :

```
function vide(T : tab; i : 1..m) : boolean;
return (T[i].marque = vide);
```

De même, on se donne les fonctions *libéré* et *occupé* qui permettent de tester si une place est libérée ou occupée.

La procédure de suppression consiste à rechercher x et à marquer sa place comme libérée :

```
procedure supprimer_HD(x : Elément; var T : tab);
{cette procédure supprime x du tableau T s'il est présent, et ne fait rien sinon}
var v : integer;
begin
    v := rechercher_HD2(x,T); if v > 0 then {x est présent}
    T[v].marque := libéré
end supprimer_HD;
```

La fonction *rechercher* est presque inchangée :

```
function rechercher_HD2(x : Elément; T : tab) : integer;
{cette fonction a pour résultat l'indice de x dans T, si x est présent dans T,
et 0 sinon}
var i, v : integer;
begin
    i := 1;
    while i ≤ m do begin
        v := essai(i, x);
        if (vide(T, v)) or ((occupé(T, v)) and (T[v].val = x))
        then exit else i := i + 1
    end;
    if (occupé(T, v)) and (T[v].val = x) then {x présent} return (v)
    else {x absent} return (0)
end rechercher_HD2;
```

Par contre, la procédure d'adjonction est modifiée de manière significative : on commence par vérifier, en allant jusqu'à la première place libre que x n'est pas présent ; puis on l'ajoute dans la première place libérée :

```
procedure ajouter_HD2(x : Elément; var T : tab; var plein : boolean);
{si x n'est pas dans T, cette procédure l'ajoute dans T à la première place
vide ou libérée rencontrée au cours des essais ; si x est dans T, la procédure
ne fait rien ; si le tableau est plein, c'est que x n'a pu être ajouté, plein
prend la valeur true, sinon il vaut false}
var i, v, lib : integer;
begin
    i := 1; plein := false; lib := 0;
```

```
        while i ≤ m do begin
              v := essai(i, x);
              if vide(T, v) then exit;
                    {x est absent; on arrête le parcours du tableau}
              if (occupé(T, v)) and (T[v].val= x) then return;
                    {x est présent, on sort de la procédure sans rien faire}
              if (libéré(T, v)) and (lib = 0) then lib := v;
                    {on mémorise l'indice de la première place libérée
                    rencontrée}
              i := i + 1 {on poursuit la boucle tant qu'on ne trouve pas x
                          ou une place vide}
        end;
        if vide (T, v) then
              if lib = 0 then begin T[v].val := x; T[v].marque := occupé end
              else begin T[lib] .val := x; T[lib].marque := occupé end
        else plein := true
  end ajouter_HD2;
```

On remarque que le coût pour les recherches d'éléments déjà présents et pour les adjonctions ne s'améliore pas quand on fait des suppressions.

5. Comparaison des méthodes de hachage étudiées

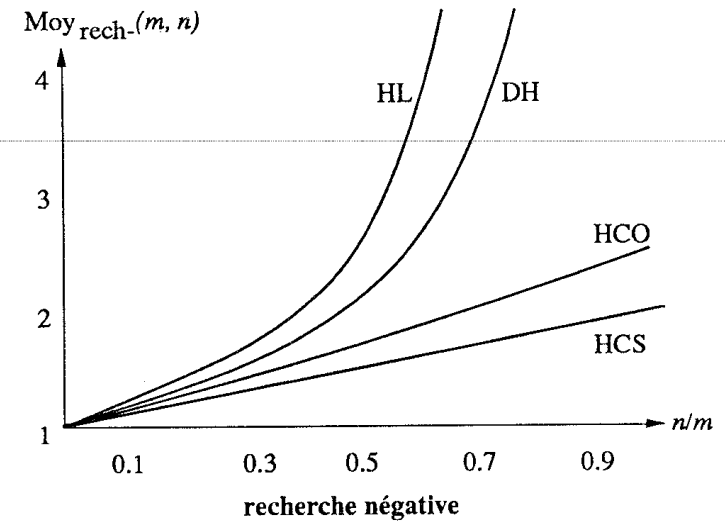
Sur les deux dessins de la figure 6, on a représenté le nombre moyen d'examens de places (pour m et n grands, et $\alpha = n/m$) pour une recherche négative et pour une recherche positive, en fonction du taux de remplissage α. Sur chaque dessin, les quatre courbes correspondent aux méthodes étudiées : hachage avec chaînage séparé (HCS), hachage coalescent (HCO), hachage linéaire (HL), et double hachage (DH).

Cette figure met en évidence plusieurs faits.

a) Pour un faible taux de remplissage, toutes les méthodes sont à peu près équivalentes.

b) La méthode la plus efficace est le hachage avec chaînage séparé; et de plus elle présente deux avantages par rapport aux autres méthodes : la suppression d'un élément est une opération simple et d'autre part le taux de remplissage peut dépasser 1. Mais elle a l'inconvénient de nécessiter une allocation dynamique de la mémoire.

c) Lorsque la zone de mémoire réservée au hachage doit être fixée *a priori*, la méthode de hachage coalescent est la plus efficace. Cependant, elle nécessite un adressage indirect qui utilise de la place mémoire; si la taille des éléments est



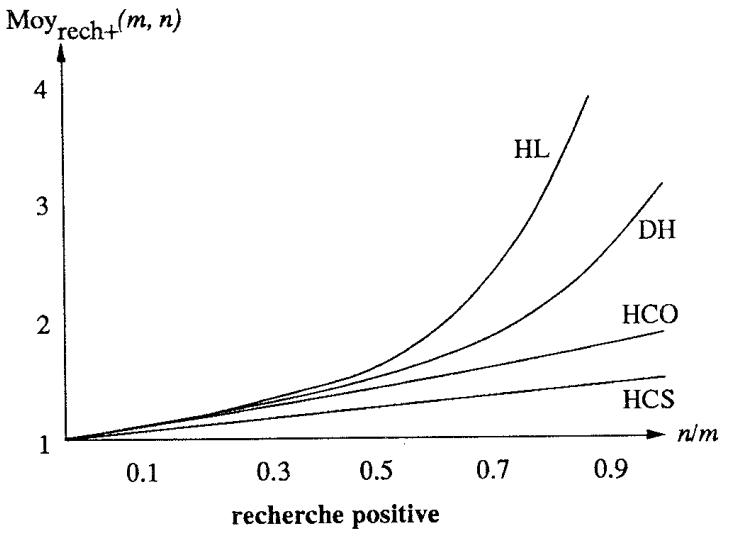


Figure 6. Comparaison des méthodes de hachage :
HL =hachage linéaire; DH =double hachage; HCO = hachage coalescent;
HCS = hachage avec chaînage séparé.

grande par rapport à la taille d'un pointeur, cela a peu d'importance, mais dans le cas contraire il vaut mieux utiliser des méthodes de hachage direct.

d) Le double hachage est meilleur que le hachage linéaire, mais il nécessite le calcul de deux fonctions de hachage.

e) C'est le hachage linéaire qui est le moins rapide mais la méthode présente l'avantage d'être simple à mettre en œuvre et ses performances sont tout à fait acceptables lorsque le taux de remplissage est faible.

Toutes ces méthodes de hachage sont très bien adaptées à la gestion d'ensembles statiques : si on connaît le nombre n d'éléments, on choisit la taille m du tableau de hachage de telle sorte que le taux de remplissage soit de l'ordre de 70 à 80 %. Mais, hormis la méthode de hachage avec chaînage séparé, elles supportent mal les adjonctions et surtout les suppressions : si le tableau est trop rempli les adjonctions entraînent trop de collisions ; les suppressions par marquage pénalisent les recherches d'éléments restant et les adjonctions ultérieures, et les suppressions physiques sont beaucoup trop coûteuses. Pour remédier à ces inconvénients on peut évidemment allouer à l'avance une très grande zone, mais cela entraîne une sous-utilisation de la mémoire. On peut aussi réorganiser périodiquement l'ensemble des éléments : quand le taux de remplissage dépasse un certain seuil, on crée un nouveau tableau plus grand, et on traite à nouveau tous les éléments présents avec une nouvelle fonction de hachage.

Ce second traitement a évidemment l'inconvénient d'interdire l'utilisation des éléments pendant un certain temps.

Exercices

1. a) Etant donnés un tableau de hachage de dimension m, et une fonction de hachage uniforme à valeurs dans $[1..m]$, montrer que si l'on doit traiter n éléments, la probabilité pour que k éléments aient une même valeur de hachage primaire donnée est :

$$P(k) = C_n^k \left(\frac{1}{m}\right)^k \cdot \left(\frac{m-1}{m}\right)^{n-k}$$

b) Montrer que si m et n tendent vers l'infini, le rapport n/m restant fixé, cette probabilité tend vers une loi de Poisson de paramètre $\alpha = n/m$, c'est-à-dire :

$$P(k) = e^{-\alpha} \frac{\alpha^k}{k!}$$

2. Quel type de fonction de hachage ne faut-il pas utiliser si on veut éviter le plus possible les collisions primaires, lorsque les éléments forment des progressions arithmétiques (comme, par exemple, les noms de variables $I1, I2, I3, DEBA, DEBB, DEBC$) ?

3. Utilisation d'une fonction de hachage par division.

Soit $r = 2^k$, où k est le nombre de bits de codage des caractères (au § 2 $r = 32$).

a) Montrer que si r **mod** $m = 1$, alors le reste dans la division par m du nombre représenté par la chaîne de bits codant un mot est égal à la somme modulo m des représentations des caractères du mot.

Peut-on alors distinguer, par leurs valeurs de hachage, les différentes permutations d'un même mot ?

b) Que se passe-t-il s'il existe un entier s tel que r^s **mod** $m = 1$?

4. Choix de θ dans la construction d'une fonction de hachage par multiplication.

Etudier les phénomènes d'accumulation dans les cas suivants :

a) $\theta = 0,99999$

b) $(r * \theta)$ **mod** 1, est voisin de 0 ou 1 (pour la définition de r, cf. exercice précédent).

5. Décrire les opérations de recherche, adjonction et suppression par la méthode de hachage avec chaînage séparé, lorsque les listes *sont triées.*

Analyser le nombre moyen de comparaisons pour une recherche positive et pour une recherche négative.

Peut-on utiliser dans ce cas la relation qui exprime le coût moyen d'une recherche positive en fonction du coût moyen d'une recherche négative ?

6. Dans cet exercice, on utilise des séries génératrices pour retrouver le coût moyen d'une recherche, positive ou négative, pour le hachage avec chaînage séparé. Soit m la taille du tableau et n le nombre d'éléments. La fonction de hachage utilisée est uniforme sur $[1, m]$.

a) Soit $P_{n,k}$ la probabilité qu'une liste donnée soit de longueur k (voir exercice n° 1). On considère la série génératrice

$$P_n(z) = \sum_{k \geq 0} P_{n,k} . z^k$$

Montrer que $P_n(z) = \left(1 + \frac{z-1}{m}\right)^n$

b) En déduire le coût moyen d'une recherche positive et le coût moyen d'une recherche négative.

7 . On considère une variante du hachage avec chaînage séparé qui consiste à ranger les clés ayant même valeur de hachage dans un arbre binaire de recherche, au lieu d'une liste.

a) Calculer la probabilité $P(k)$ pour que k éléments aient une même valeur de hachage v, dans l'hypothèse où les m^n suites d'éléments sont équiprobables (m est la taille du tableau, et n le nombre d'éléments insérés).

b) Exprimer, en fonction des $P(k)$ et des nombres harmoniques H_k, le nombre moyen d'essais :

– pour une recherche avec succès ;

– pour une recherche avec échec.

Puis expliciter les calculs de ces nombres.

On admettra alors que $\sum_{k\geq 1} P(k).H_k = \mathrm{Log}\frac{n}{m} + c + o\left(\frac{1}{m}\right)$, où $c \in \mathbb{R}$.

c) Comparer la complexité de cette méthode, en temps d'exécution et en place mémoire occupée, avec la complexité des deux méthodes (hachage avec chaînage séparé, et arbre binaire de recherche) dont elle est issue.

8. a) Dans le hachage linéaire, on peut localiser exactement tous les éléments en collision avec un élément donné. A partir de cette remarque, décrire un algorithme de suppression «physique» d'un élément dans un tableau de hachage linéaire, et étudier le nombre de transferts d'éléments que peut provoquer une suppression.

b) Reprendre le problème pour le hachage coalescent.

9. Montrer les relations suivantes utilisées pour l'analyse du double hachage :

a) $p_r = \dfrac{C_{m-r}^{n-r+1}}{C_m^n}$

b) $\sum_{r=1}^{n+1} r.p_r = m + 1 - \sum_{r=1}^{n+1} (m+1-r).p_r$

c) En déduire que $\sum_{r=1}^{n+1} r.p_r = \dfrac{m+1}{m-n+1}$

10. On représente des ensembles avec répétitions par une méthode de hachage.

a) Si on utilise une méthode de double hachage, combien fait-on d'essais pour entrer n *éléments égaux* dans un tableau vide de dimension $m > n$?

b) Que peut-on dire des méthodes de hachage dans le cas d'une application où il y a beaucoup de clés égales ?

11. La méthode de hachage dite ***hachage quadratique*** utilise la suite d'essais suivante; pour $i \geq 1$: $essai_i(x) = h_i(x)$ **mod** m, avec $h_i(x) = h(x) + (i-1)^2$, où h est une fonction de hachage uniforme à valeurs dans $[0, m-1]$.

a) Montrer que l'on peut éviter le calcul d'un «carré» en utilisant les relations de récurrence suivantes :

$$h_{i+1}(x) = h_i(x) + d_i$$

et $\quad d_{i+1} = d_i + 2$ pour $i \geq 1$,

avec $h_1(x) = h(x)$, et $d_1 = 1$.

b) Montrer que si m est un nombre premier, la suite des essais successifs balaie au moins la moitié du tableau.

12. On suppose qu'on peut disposer d'une suite infinie $h_1, h_2, \ldots$ de fonctions de hachage à valeurs dans $[1..m]$ qui sont uniformes et indépendantes; la ***méthode de hachage avec essais uniformes*** consiste à les essayer l'une après l'autre.

Soit p_r la probabilité qu'il faille exactement r essais pour trouver une case vide dans un tableau de hachage de dimension m contenant n éléments.

a) Montrer que $p_r = \alpha^{r-1}(1-\alpha)$, où $\alpha = \frac{n}{m}$ est le taux d'occupation du tableau de hachage.

b) En déduire que le nombre moyen d'essais pour une recherche négative vaut $1/1-\alpha$.

c) Montrer que le nombre moyen d'essais pour une recherche positive est $\frac{m}{n}(H_m - H_{m-n})$, où H_m est le $m^{ième}$ nombre harmonique. Déterminer la valeur asymptotique de ce coût, lorsque m et n tendent vers l'infini, α étant fixé.

13. a) Lorsqu'un tableau de hachage est plein, on peut «rehacher» ses éléments dans un tableau de dimension plus grande. On peut aussi faire du rehachage pour réorganiser un tableau lorsque beaucoup de ses éléments sont supprimés «logiquement».

Ecrire une procédure de rehachage dans un tableau construit par la méthode de hachage linéaire, qui puisse être appliquée soit pour agrandir le tableau, soit pour supprimer effectivement les éléments n'existant plus.

b) Reprendre le problème avec un tableau de hachage coalescent.

14. Soit m un entier et t un tableau d'indices $0..m-1$. Soit h une fonction de hachage définie sur le type des éléments de t et à résultats dans $[0, m-1]$. La méthode dite de ***hachage ordonné*** utilise à la fois du hachage linéaire et des comparaisons entre les valeurs des éléments : les éléments en collision sont triés par ordre décroissant.

La procédure d'adjonction correspondant à un ordre décroissant des éléments en collision est donnée ci-dessous.

a) Ecrire une procédure de suppression d'un élément dans un tableau de hachage ordonné

b) Montrer que pour une collection fixée d'éléments (m et h fixés également), le tableau obtenu par des adjonctions successives ne dépend pas de l'ordre d'ad-

jonctions des éléments. Par contre, l'ordre des adjonctions influe sur le nombre d'échanges : il existe un ordre qui permet de ne faire aucun échange, lequel ?

```
procedure ajouter_HO(var t : array [1..m] of Elément; x : Elément);
{Les données sont t et x ; si x est présent on ne l'ajoute pas , si x est absent,
on l'ajoute ; on suppose qu'il existe au moins une place vide dans t}
var v : integer; y : Elément;
begin
    {h est une fonction de hachage, déclarée ailleurs}
    v := h(x); y := x;
    while (t[v] <> y) and (not vide (t, v)) do begin
        if t[v] < y then t[v] ↔ y;
        v := v + 1 ; if v > m then v := 1
    end;
    if vide(t, v) then t[v] := y
end ajouter_HO;
```

15. On désire écrire un programme qui à la lecture d'un texte (Pascal ou Lisp... ou français) repère pour chaque identificateur les lignes où il apparaît. Un identificateur commence par une lettre et n'est composé que de lettres et de chiffres. Certains mots dits réservés ne peuvent pas être utilisés comme identificateurs.

Après la lecture, on demande d'imprimer le texte, puis la liste (triée par ordre alphabétique) des identificateurs suivis chacun de la liste des numéros de ligne où il apparaît, ces numéros de ligne étant rangés en ordre croissant.

a) Faire une analyse comparée des différentes structures de données possibles.

b) Programmer une méthode utilisant des arbres binaires de recherche, et une méthode utilisant un tableau de hachage. Analyser les résultats.

Lectures conseillées pour le chapitre 12

Knuth, *The art of Computer Programming*, vol. 3 : *Sorting and Searching*, Addison-Wesley, 1973.

Reingold & Hansen, *Data Structures*, Little Brown & C° , Boston, 1985.

Chapitre 13
Recherche externe
Conclusions sur la recherche

1. Recherche externe

On présente quelques méthodes, dites *méthodes de recherche externe*, pour gérer de très grandes collections d'éléments stockés sur mémoire secondaire, par exemple sur disque. On suppose que la mémoire secondaire est *paginée*, c'est-à-dire découpée en pages pouvant contenir chacune un certain nombre d'éléments. Une ***page*** est un bloc contigu d'informations qui sont transférées en même temps en mémoire centrale. On suppose aussi que le système de gestion de l'espace mémoire peut à tout moment, à la demande de l'utilisateur, allouer des pages en mémoire secondaire, ou inversement récupérer de l'espace libéré.

Le temps d'accès à la mémoire secondaire étant de l'ordre de mille fois supérieur au temps d'exécution d'une instruction dans l'unité centrale, c'est le nombre d'accès à la mémoire secondaire qui est prépondérant dans la complexité en temps des algorithmes de gestion de fichiers résidant en mémoire secondaire.

On cherche donc à organiser les éléments de la mémoire paginée de telle façon que les opérations de recherche, adjonction et suppression ne nécessitent que très peu de transferts de pages, au prix éventuellement d'un plus grand nombre d'opérations en unité centrale.

1.1. *B*-arbres

Les B-arbres sont une généralisation des arbres-2.3.4 présentés au chapitre 11. Un B-arbre d'ordre m est un arbre de recherche (cf. chapitre 11, §3.1) formé de nœuds qui peuvent chacun contenir jusqu'à $2m$ éléments; chaque nœud est dans une page différente du disque, et la complexité des algorithmes de recherche, adjonction et suppression d'un élément parmi n, comptée en nombre d'accès à la mémoire secondaire, est en $O(\log_m n)$.

Définition : Un ***B-arbre d'ordre m*** est un arbre de recherche dont toutes les feuilles sont situées au même niveau, et dont les nœuds sont de différents types :

- tous les nœuds, sauf la racine, sont des k-nœuds avec k compris entre $m+1$ et $2m+1$,
- la racine est un k-nœud avec k compris entre 2 et $2m+1$.

La figure 1 montre un exemple de B-arbre d'ordre 2.

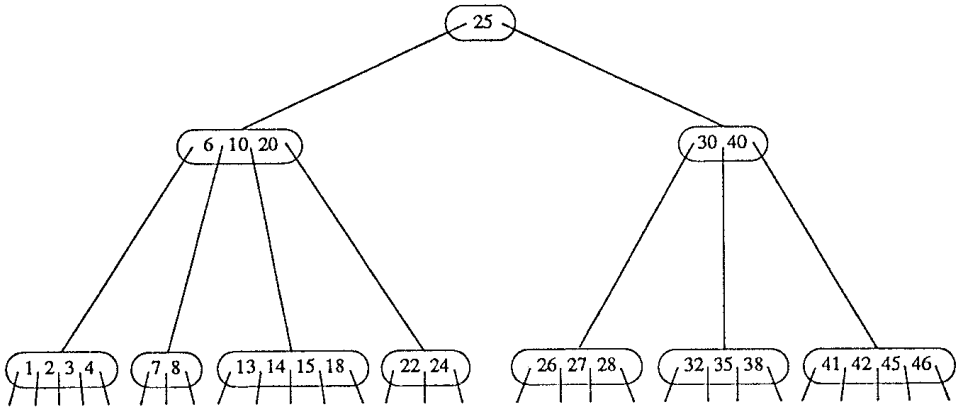


Figure 1. B-arbre d'ordre 2.

Le principe de la recherche d'un élément dans un B-arbre est le même que pour les arbres-2.3.4 : descendre dans l'arbre à partir de la racine, en s'orientant par comparaisons avec les éléments contenus dans les nœuds.

Si l'on suppose que le nœud racine est en mémoire centrale, le nombre d'accès à la mémoire secondaire lors d'une recherche est égal à la profondeur, dans le B-arbre, du nœud contenant l'élément cherché.

Or, la hauteur d'un B-arbre d'ordre m qui contient n éléments est majorée par $\log_{m+1}\left(\frac{n+1}{2}\right)$ (cf. exercice); il en résulte que le nombre d'accès à la mémoire secondaire est toujours inférieur à $\log_{m+1}\left(\frac{n+1}{2}\right)$.

On a donc intérêt à prendre m aussi grand que possible.

Le choix de m dépend du système sur lequel on travaille (taille d'un secteur du disque, quantité de données pouvant être transférée lors de chaque accès en mémoire secondaire...). En pratique, la valeur de m est le plus souvent comprise entre 50 et 400.

Cela permet de ranger un très grand nombre d'éléments dans un arbre de hauteur

très petite; par exemple si $m = 250$, un arbre de hauteur 2 peut contenir plus de 125 millions d'éléments :

- 500 éléments à la racine,
- 501×500, soit à peu près 25×10^4 éléments dans les nœuds de profondeur 1,
- $501^2 \times 500$, soit à peu près 125×10^6 éléments dans les nœuds de profondeur 2 (dans les B-arbres les niveaux les plus bas de l'arbre contiennent la quasi-totalité des éléments).

Comme pour les arbres-2.3.4, l'adjonction d'un élément ne pose problème que lorsque le nœud devant recevoir cet élément est plein, et l'on peut traiter les éclatements soit en remontée soit en descente.

- Dans le premier cas, on fait une recherche classique pour trouver la feuille où doit se faire l'adjonction; lorsque cette feuille contient déjà $2m$ éléments on l'éclate pour constituer deux nœuds de m éléments; et l'on ajoute dans le nœud père l'élément médian qui sert désormais à départager ces deux feuilles. Lorsque l'itération de ce procédé conduit à l'éclatement de la racine, on place l'élément médian dans un nouveau nœud racine, et la hauteur de l'arbre augmente alors de une unité.
- Dans le second cas, lors de la descente dans l'arbre à la recherche de l'endroit où il faut faire l'adjonction, on éclate systématiquement les nœuds qui contiennent $2m$ éléments; de cette façon l'adjonction se fait toujours dans une feuille qui n'est pas pleine, mais certains éclatements peuvent augmenter inutilement le nombre de nœuds, et la hauteur de l'arbre.

L'analyse de la complexité d'une recherche ou d'une adjonction dans un B-arbre d'ordre m repose sur l'étude de plusieurs paramètres.

La hauteur de l'arbre et le nombre d'éclatement déterminent la complexité en temps.

- *La hauteur* indique le nombre de pages auxquelles il faut accéder en lecture; elle est toujours d'ordre logarithmique (cf. exercices).
- *Le nombre d'éclatements* est fondamental car chaque éclatement nécessite l'écriture de deux pages; ce nombre est trivialement borné par la hauteur de l'arbre.

L'éclatement étant la seule façon de créer des nœuds, on montre facilement (cf. exercices) que le nombre moyen d'éclatements par adjonction, lors de la construction par adjonctions successives d'un B-arbre d'ordre m, est compris entre $\frac{1}{2m}$ et $\frac{1}{m}$. Une analyse statistique très complexe des niveaux les plus bas de l'arbre – dite analyse de frange – montre qu'il y a en moyenne 1 éclatement pour $1{,}38*m$ adjonctions dans la méthode avec éclatements en remontée. Ce résultat est confirmé par l'expérience : en fait, l'analyse de frange donne une bonne estimation car, dans un B-arbre, la quasi-totalité des éléments est située aux plus bas niveaux.

1.2. Hachage dynamique

Le principe du hachage dynamique est le suivant : le tableau de hachage est remplacé par un ***index*** en mémoire centrale; cet index, fabriqué par raffinements successifs d'une fonction de hachage, est un arbre binaire dont les feuilles contiennent des adresses en mémoire secondaire. Pour rechercher un élément x, on suit un chemin à partir de la racine de l'arbre jusqu'à une feuille, ce chemin étant déterminé par la valeur de hachage de l'élément x. La feuille sur laquelle on aboutit contient l'adresse de la page où se trouve l'élément x.

Comme la taille des pages en mémoire secondaire est limitée, si le nombre d'éléments croît, il faut allouer de nouvelles pages. On répartit ensuite les éléments dans les pages selon la fonction de hachage, en utilisant plus ou moins l'information contenue dans les valeurs de hachage des éléments, suivant qu'il y a plus ou moins d'éléments en collision. Les nouvelles pages sont référencées par de nouvelles feuilles de l'index, dont la taille croît ainsi au fur et à mesure que l'on rajoute des pages.

Si l'index est trop grand pour tenir en mémoire centrale, on le pagine lui aussi en sous-arbres, et la recherche d'un élément peut alors nécessiter plusieurs accès à la mémoire secondaire.

Exemple : La figure 2 montre le résultat de l'adjonction successive par hachage dynamique des éléments $E, X, T, F, R, N, C, L, S, G, B$.

La valeur de hachage d'une lettre est la représentation binaire de son ordinal codée sur 5 bits :

$$h(E) = 00101,\ h(X) = 11000,\ h(T) = 10100,\ h(F) = 00110,\ h(R) = 10010,$$
$$h(N) = 01110,\ h(C) = 00011,\ h(L) = 01100,\ h(S) = 10011,\ h(G) = 00111,$$
$$h(B) = 00010.$$

On a supposé que les pages en mémoire secondaire peuvent contenir au plus quatre éléments. Au départ l'index est réduit à un pointeur vers une page de la mémoire secondaire. L'adjonction des quatre premiers éléments remplit cette page. Lors de l'adjonction du cinquième élément, il faut allouer une deuxième page, et l'on distribue les éléments dans ces deux pages selon le premier bit de leur valeur de hachage. Cette distribution se fait par l'intermédiaire de l'index : on part à gauche de la racine de l'arbre si le premier bit est 0, et à droite si le premier bit est 1. On ajoute ensuite N et C dans ces deux pages. L'adjonction de L fait éclater la page correspondant au sous-arbre gauche, et l'on répartit les éléments de *cette page* selon le deuxième bit de leur valeur de hachage, par l'intermédiaire de l'index, qui a augmenté d'un nœud. Pour ajouter S on utilise seulement le premier bit de la valeur de hachage de S puisqu'on va dans la page de droite, et pour G on doit utiliser les deux premiers bits de la valeur de hachage de G, mais ces deux adjonctions ne

modifient pas l'index. Enfin, l'adjonction de B nécessite encore une nouvelle page, et un raffinement supplémentaire du hachage pour les éléments dont la valeur de hachage commence par 00.

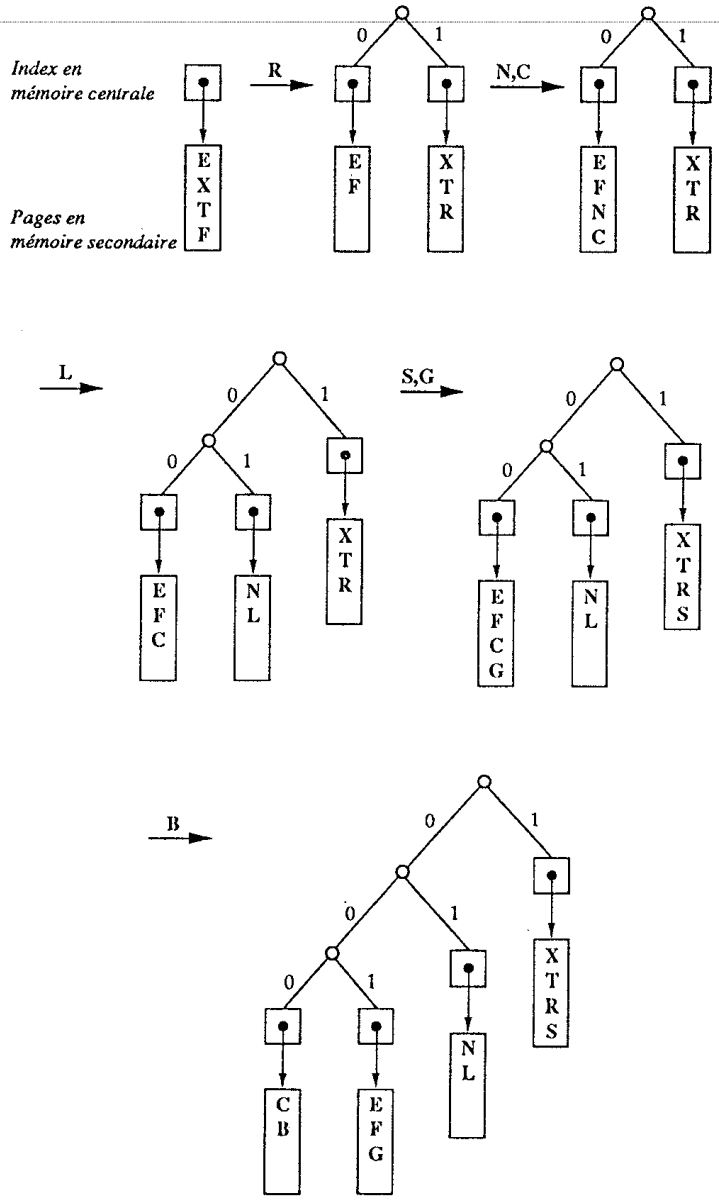


Figure 2. Hachage dynamique.

L'analyse de la complexité du hachage dynamique montre que le nombre d'accès à la mémoire secondaire est en moyenne de l'ordre de 2, quel que soit le nombre d'éléments (l'index est lui-même paginé si sa taille est trop grande). Et le taux d'occupation des pages (c'est-à-dire, le rapport entre le nombre minimal de pages

nécessaires pour ranger les éléments et le nombre de pages réellement utilisées) est de l'ordre de 70 %.

2. Conclusions sur les méthodes de recherche

On a étudié aux quatre chapitres précédents différents types de méthodes pour gérer des collections de données sur lesquelles on veut effectuer des opérations de recherche, adjonction ou suppression : méthodes séquentielles, dichotomiques, méthodes arborescentes et méthodes de hachage.

Les complexités totales en temps (comparaisons et déplacements) de ces méthodes sont résumées dans les tableaux qui suivent. La figure 3.a montre la complexité en moyenne pour une opération dans une collection de n éléments, et la figure 3.b montre la complexité dans le pire des cas.

	recherche	adjonction	suppression
méthode séquentielle	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
dichotomie	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
arbre binaire de recherche	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
arbres équilibrés	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
hachage	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

Figure 3.a. Complexité en moyenne.

	recherche	adjonction	suppression
méthode séquentielle	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
dichotomie	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
arbre binaire de recherche	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
arbres équilibrés	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
hachage	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

Figure 3.b. Complexité au pire.

Il est intéressant de mettre en évidence quelques points qui permettront éventuellement, pour une application donnée, de choisir entre ces différentes classes de méthodes.

1) La méthode dichotomique permet de rechercher un élément avec un nombre de comparaisons qui est *toujours* d'ordre inférieur ou égal à $\log n$, ce qui constitue une amélioration importante par rapport à la méthode séquentielle. La méthode dichotomique est de plus très simple à mettre en œuvre, et n'utilise que la place

mémoire nécessaire pour stocker les éléments eux- mêmes. La dichotomie dans un tableau convient donc très bien pour l'opération de recherche. Cependant elle est mal adaptée aux opérations d'adjonction et de suppression, qui peuvent nécessiter de l'ordre de n déplacements d'éléments.

2) L'introduction de structures arborescentes permet de traiter les adjonctions et les suppressions aussi bien que les recherches. La place mémoire nécessaire est plus importante à cause des pointeurs, mais elle reste proportionnelle au nombre d'éléments.

L'arbre binaire de recherche est une structure simple, sur laquelle les trois opérations sont faciles à décrire. En moyenne les arbres binaires de recherche de n éléments ont une hauteur de l'ordre de $\log n$, si bien qu'une opération d'adjonction, de suppression ou de recherche nécessite en moyenne de l'ordre de $\log n$ comparaisons. Mais il se peut qu'un arbre binaire de recherche se mette à ressembler à une liste, et chacune des opérations peut alors avoir une complexité linéaire par rapport au nombre d'éléments.

3) Avec les méthodes d'arbres équilibrés on est sûr que la hauteur de l'arbre, et donc la complexité de chaque opération, est de l'ordre de $\log n$. Mais les algorithmes d'adjonction et de suppression sont beaucoup plus complexes que dans le cas des arbres binaires de recherche à cause du rééquilibrage.

4) Les méthodes de hachage ont un coût moyen constant; elles sont donc plus performantes en moyenne que les méthodes arborescentes.

Mais dans le cas le pire, une opération de recherche, adjonction ou suppression dans un tableau de hachage contenant n éléments, peut nécessiter n comparaisons. Il y a des situations où l'on ne veut *jamais* prendre le risque d'avoir une complexité linéaire, et l'on préfère alors une méthode d'arbres équilibrés, qui assure une complexité logarithmique dans le pire des cas.

5) Les méthodes de hachage sont en général assez mal adaptées aux suppressions d'éléments, alors que dans les arbres équilibrés les suppressions ne sont pas beaucoup plus compliquées que les adjonctions.

6) Les méthodes de hachage dispersent les éléments et ne conservent pas les propriétés de voisinage; elles ne sont donc pas satisfaisantes quand on veut faire des recherches par intervalle (*i.e.* rechercher la suite des éléments compris entre deux éléments donnés), ou encore des tris. Par contre, ces types de problèmes sont faciles à traiter à partir d'une structure arborescente.

7) On a vu que les deux classes de méthodes peuvent être adaptées pour traiter de grandes collections de données sur mémoire secondaire paginée, avec un taux moyen de remplissage de la mémoire secondaire de 70 %, aussi bien dans le cas des B-arbres que du hachage dynamique.

Il est souvent important, en particulier pour les grandes collections de données, que plusieurs utilisateurs puissent travailler en même temps sur les mêmes données – on parle alors d'utilisation en accès partagé. Pour les structures arborescentes, chaque nœud est alors un goulot d'étranglement potentiel, puisque c'est l'unique point d'entrée pour tous les éléments du sous-arbre enraciné en ce nœud; pour cette raison, les protocoles d'accès partagé sont plus complexes pour les méthodes arborescentes que pour les méthodes de hachage, ce qui donne un avantage important aux méthodes de hachage dans ce type de contexte.

Exercices

1. Montrer que pour un B-arbre d'ordre m contenant n éléments et de hauteur h, on a l'encadrement suivant :

$$\log_{2m+1}(n+1) - 1 \leq h \leq \log_{m+1}\left(\frac{n+1}{2}\right)$$

2. Montrer que lors de la construction d'un B-arbre, d'ordre m, par adjonctions successives, le nombre moyen d'éclatements par adjonction, noté e, vérifie :

$$\frac{1}{2m} \leq e \leq \frac{1}{m}$$

3. Quel est le nombre maximal d'accès à la mémoire secondaire lors de l'adjonction d'un élément dans un B-arbre :

- par la méthode avec éclatements à la descente,
- par la méthode avec éclatements en remontée.

4. On peut envisager une autre méthode d'adjonction avec éclatements en remontée, dans un B-arbre, qui permet une meilleure utilisation de la mémoire secondaire : lorsqu'un nœud doit éclater on essaie de redistribuer ses éléments dans le nœud frère situé immédiatement à sa gauche ou dans le nœud frère situé immédiatement à sa droite ; et c'est seulement dans le cas où ces deux nœuds frères sont pleins que l'on fait un éclatement.

Décrire un algorithme d'adjonction utilisant ce principe et tel que la mémoire secondaire est occupée au moins aux deux tiers. Quel est pour cet algorithme le nombre maximal d'accès lors d'une adjonction ?

5. Etudier comment on peut représenter les B-arbres par des arbres binaires de recherche bicolores.

Lectures conseillées pour le chapitre 13

Sedgewick, *Algorithms*, Addison-Wesley, 1983.

Quatrième partie
Algorithmes de tri

Chapitre 14
Introduction et méthodes simples

1. Introduction à l'étude des méthodes de tri

Les méthodes de tri sont très importantes en pratique, en particulier en informatique de gestion où beaucoup d'applications consistent à trier des fichiers. De plus, elles interviennent dans beaucoup d'autres problèmes : on le verra, par exemple, dans les algorithmes sur les graphes. Le tri est également un bon exemple de problème pour lequel de nombreux algorithmes existent : l'analyse et la comparaison de ceux-ci sont donc indispensables. On a des résultats d'optimalité : borne inférieure pour la complexité au pire et en moyenne en nombre de comparaisons et existence d'algorithmes atteignant cette borne.

1.1. Spécification

La spécification du tri est la suivante : la donnée est une liste de n éléments; à chaque élément est associée une clé qui appartient à un ensemble totalement ordonné; le résultat est une liste dont les éléments sont une permutation des éléments de la liste d'origine telle que les clés soient croissantes quand on parcourt la liste séquentiellement[1].

La figure 1 présente un exemple de tri sur une liste dont chaque élément est un ensemble de renseignements sur une ville : la clé est le nom de la ville, et l'ordre sur les clés est l'ordre alphabétique.

Plus formellement, on peut décrire le tri comme un enrichissement du type abstrait Liste par l'opération :

tri : Liste → Liste

(1) Si on prend les définitions d'un dictionnaire, il s'agit d'un classement plutôt que d'un tri.

LISTE DONNÉE			LISTE RESULTAT		
JARNY	54800	9520 h.	JARNY	54800	9520 h.
YVETOT	76190	10708 h.	JARVILLE	54140	13121 h.
ROSCOFF	29211	3732 h.	MOLENE	29259	397 h.
JARVILLE	54140	13121 h.	MONSOLS	69860	764 h.
NANCY	54000	111493 h.	NANCY	54000	111493 h.
MONSOLS	69860	764 h.	PAGNY	54530	3732 h.
MOLENE	29259	397 h.	ROSCOFF	29211	3732 h.
PAGNY	54530	3732 h.	YVETOT	76190	10708 h.

Figure 1. Exemple de tri.
(Source : Petit Larousse illustré, 1982)

L'ensemble totalement ordonné auquel appartiennent les clés peut être spécifié par :

sorte Clé
utilise Booléen
opération
$\_\leq\_$: Clé × Clé → Booléen

axiomes
$x \leq x$ = vrai
$(x \leq y = \text{vrai})$ & $(y \leq x = \text{vrai}) \Rightarrow x = y$
$(x \leq y = \text{vrai})$ & $(y \leq z = \text{vrai}) \Rightarrow x \leq z = \text{vrai}$
avec
x, y, z : Clé

Comme dans le cas des problèmes de recherche, on se donne une opération nommée *clé*, qui étant donné un élément rend sa clé :

clé : Elément → Clé

Une liste est triée si les clés de ses éléments apparaissent en ordre croissant lorsqu'on la parcourt. Cela peut se spécifier en enrichissant le type "Liste Récursive" de la manière suivante :

opération
triée : Liste → Booléen
axiomes
triée(*liste-vide*) = vrai
triée(*cons*(e, *liste-vide*)) = vrai
$\lambda \neq$ *liste-vide*
$\Rightarrow$ *triée*(*cons*(e, λ)) = (*clé* (e) $\leq$ *clé*(*premier*(λ))) $\wedge$ *triée*(λ)
avec
e : Elément; λ : Liste

Un tri d'une liste λ consiste à trouver une liste λ', triée, dont les éléments sont une permutation des éléments de λ. Deux listes λ et λ' sont des permutations l'une de l'autre, si et seulement si, le nombre d'occurrences de tout élément est égal dans les deux listes. On est donc amené, pour spécifier la relation de permutation entre deux listes, à définir une opération qui donne le nombre d'occurrences d'un élément dans une liste (elle est très similaire à celle qui a été définie au chapitre 6 pour les ensembles avec répétitions). On utilise également l'opération de concaténation de deux listes vues au chapitre 5. Cela donne la spécification suivante :

opérations
perm : Liste × Liste → Booléen
nb-occ : Elément × Liste → Entier

axiomes
nb-occ(x, *liste-vide*) = 0
$x = y \Rightarrow$ *nb-occ*(x, *cons*(y, λ)) = *nb-occ*(x, λ) +1
$x \neq y \Rightarrow$ *nb-occ*(x, *cons*(y, λ)) = *nb-occ*(x, λ)
perm(*liste-vide, liste vide*) = vrai
perm(*liste-vide, cons*(x, λ)) = faux
nb-occ(x, μ) = 0 $\Rightarrow$ *perm*(*cons*(x, λ), μ) = faux
perm(*cons*(x, λ), *concat*(μ, *cons*(x, μ'))) = *perm*(λ, *concat*(μ, μ'))
avec
x, y : Elément; λ, μ, μ' : Liste;

Moyennant quoi, les axiomes que l'opération de tri doit satisfaire s'écrivent :

perm(λ, *tri*(λ)) = vrai
triée(*tri*(λ)) = vrai

1.2. Stabilité

Si plusieurs éléments ont la même clé on a plusieurs résultats possibles pour le tri. C'est le cas pour la liste de la figure 1, si on effectue un tri avec pour clé le nombre d'habitants : il se trouve que PAGNY et ROSCOFF ont le même nombre d'habitants; il y a donc deux résultats possibles, qui sont donnés dans la figure 2.

Le premier résultat de la figure 2 est le résultat d'un tri ***stable***, c'est-à-dire d'un tri qui conserve l'ordre de départ des éléments dont les clés sont égales. On verra plus loin que cette propriété est importante quand on fait des tris successifs en décomposant les clés (chapitre 16).

RESULTAT 1			RESULTAT 2		
MOLENE	29259	397 h.	MOLENE	29259	397 h.
MONSOLS	69860	764 h.	MONSOLS	69860	764 h.
ROSCOFF	29211	3732 h.	PAGNY	54530	3732 h.
PAGNY	54530	3732 h.	ROSCOFF	29211	3732 h.
JARNY	54800	9520 h.	JARNY	54800	9520 h.
YVETOT	76190	10708 h.	YVETOT	76190	10708 h.
JARVILLE	54140	13121 h.	JARVILLE	54140	13121 h.
NANCY	54000	111493 h.	NANCY	54000	111493 h.

Figure 2. Tris avec pour clé le nombre d'habitants.

1.3. Clés structurées

L'opération qui extrait une clé d'un élément peut être plus complexe que dans les exemples ci-dessus. En particulier, la clé peut être un couple de valeurs ou même un n-uplet de valeurs. Par exemple, on peut vouloir trier la liste de la figure 1 par numéro de département croissant (deux premiers chiffres du code postal) et par population décroissante. On obtient alors le résultat de la figure 3.

ROSCOFF	29211	3732 h.
MOLENE	29259	397 h.
NANCY	54000	111493 h.
JARVILLE	54140	13121 h.
JARNY	54800	9520 h.
PAGNY	54530	3732 h.
MONSOLS	69860	764 h.
YVETOT	76190	10708 h.

Figure 3. Résultat du tri selon les numéros de département croissants et les nombres d'habitants décroissants.

Dans cet exemple, toute clé c est un couple $<dept, pop>$ et on a $c \leq c'$ si, et seulement si :

$$dept < dept' \text{ ou } (dept = dept' \text{ et } pop \geq pop')$$

On dit que l'on a une ***clé structurée*** où le numéro de département est la ***sous-clé primaire*** (ou principale) et la population est la ***sous-clé secondaire.***

1.4. Organisation de la mémoire

La représentation des éléments est parfois organisée pour faciliter le tri. Par exemple, les clés peuvent être dans une zone de la mémoire particulière, chaque clé étant

accompagnée d'un pointeur vers le reste de l'élément correspondant, comme on le voit dans la figure 4. Dans ce cas, on peut effectuer le tri en permutant uniquement les clés et sans se préoccuper des éléments.

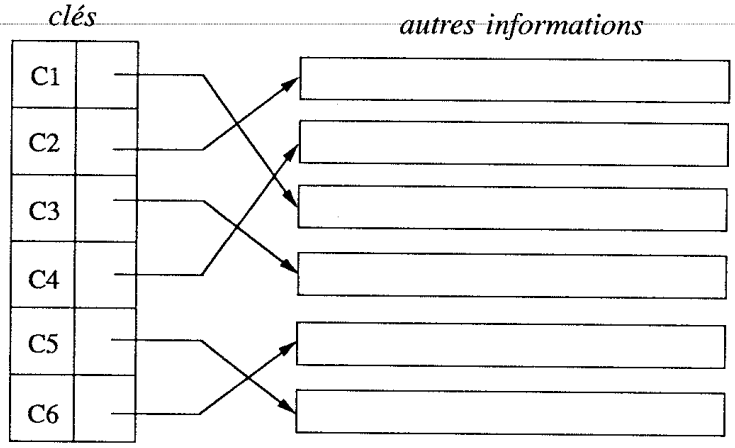


Figure 4. Représentation des clés dans une zone de mémoire distincte.

L'organisation de la mémoire est d'ailleurs un problème essentiel quand on programme un tri. Si la liste à trier ne tient pas en mémoire centrale, on est amené à employer des méthodes particulières dites de ***tri externe*** qui sont présentées au chapitre 17.

Si on peut tout faire en mémoire centrale, on dit que l'on fait du ***tri interne***, et la question se pose de savoir si on recopie le tableau (ce qui facilite bien les choses) ou si on fait du tri sur place ce qui est plus difficile, mais le plus souvent nécessaire.

Remarque : Le tri étant très utilisé, la plupart des systèmes d'exploitation et des langages de programmation comportent des programmes de tri. C'est le cas d'UNIX, LISP, COBOL, etc... Ces programmes sont souvent très efficaces, car réalisés en fonction des caractéristiques de la machine hôte et de son système. Il est donc recommandé avant de se lancer dans la programmation d'un tri, de considérer cette possibilité !

1.5. Conventions et généralités

Dans les chapitres qui suivent, on considère, pour faciliter la lecture des algorithmes, que la liste à trier est une liste d'entiers. Les clés sont les éléments eux-mêmes. De plus, on suppose en général que cette liste est représentée de manière contiguë par un tableau. Comme on l'a vu au chapitre 5, si t est ce tableau, l'expression $t[i]$ correspond au $i^{ième}$ élément de la liste si elle apparaît en partie droite d'une affectation ou comme opérande d'une comparaison ou d'une expression. Si $t[i]$ apparaît en partie gauche d'une affectation, cela correspond à l'accès à la $i^{ième}$ place de la liste.

Certaines méthodes de tri sont mieux adaptées au cas où la liste est représentée de manière chaînée, d'autres ne peuvent pas être utilisées dans ce cas : c'est alors signalé dans le texte.

Ce chapitre et les chapitres 15 et 16 présentent et étudient des algorithmes de tri pour des listes qui peuvent être contenues entièrement dans la mémoire centrale (tri interne). Il existe quantité de tels algorithmes et il n'est certainement pas utile de les énumérer tous. C'est pourquoi on se limite ici aux algorithmes les plus exemplaires, autrement dit ceux dont les autres algorithmes peuvent être considérés comme des variantes (amélioration, simplification, ou cas particulier). Beaucoup de ces variantes sont proposées en exercice.

La plupart des algorithmes étudiés effectuent deux opérations fondamentales : des *comparaisons entre clés et des transferts (ou affectations) d'éléments.* Ces algorithmes seront donc analysés et comparés en fonction du nombre de telles opérations nécessaires pour trier une liste de n éléments. Il existe cependant des algorithmes qui ne travaillent pas seulement par comparaison des clés : ils utilisent certaines propriétés de l'ensemble des clés possibles. Ces algorithmes, qui sont présentés à la fin du chapitre 16, seront donc analysés selon des critères différents.

Un deuxième critère d'analyse est la place mémoire nécessaire pour effectuer le tri d'une liste de n éléments. Ce critère est très important dans le cas du tri interne car il conditionne l'utilisation des algorithmes : les algorithmes présentés, sauf exceptions, font donc du tri sur place et utilisent un nombre constant de variables auxiliaires.

Ce chapitre présente des algorithmes de tri simples. Ces algorithmes travaillent soit par ***sélection*** du plus petit élément, suivie du tri du reste de la liste, soit par tri du début de la liste, suivi de l'***insertion*** des éléments non triés. Bien qu'ils aient des performances médiocres, ces algorithmes sont tout à fait acceptables pour trier des listes de petite taille.

2. Méthodes par sélection

Dans ces méthodes, on recherche le minimum de la liste à trier. On le met à la première place, et on recommence sur la fin de la liste (augmentée de l'élément qui se trouvait à la première place). La figure 5 donne les étapes successives d'un tri par sélection sur une liste de six éléments. On peut noter qu'après le $k^{ième}$ placement, les k plus petits éléments de la liste sont à leur place définitive.

En utilisant la spécification récursive des listes vue au chapitre 5, la spécification correspondant aux algorithmes de tri par sélection s'écrit :

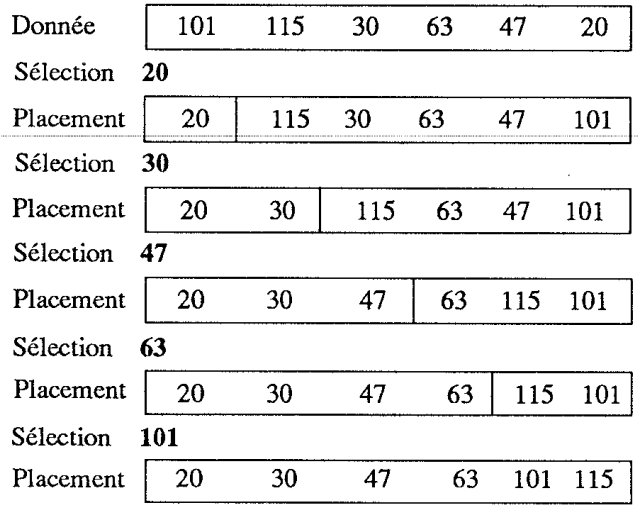


Figure 5. Exemple de tri par sélection.
Les éléments placés se trouvent à gauche de la barre verticale.

opération
tri-select : Liste → Liste
axiomes
tri-select(*liste-vide*) = *liste-vide*
$\lambda \neq$ *liste-vide* ⇒ *tri-select*(λ) = *cons*(*min*(λ), *tri-select*(*supprimer-min*(λ)))

où λ est une variable de sorte Liste et où les opérations *min* et *supprimer-min* sont définies par :

opérations
min : Liste → Elément
supprimer-min : Liste → Liste

préconditions
min(λ) **est-défini-ssi** $\lambda \neq$ *liste-vide*
supprimer-min(λ) **est-défini-ssi** $\lambda \neq$ *liste-vide*

axiomes
est-présent(*min*(λ), λ) = vrai
est-présent(e, λ) = vrai ⇒ *min*(λ) ≤ e = vrai
nb-occ(*min*(λ), *supprimer-min*(λ)) = *nb-occ*(*min*(λ), λ) -1
$e \neq$ *min*(λ) ⇒ *nb-occ*(e, *supprimer-min*(λ)) = *nb-occ*(e, λ)
avec
e : Elément; λ : Liste

On peut remarquer que cette spécification ne précise pas comment est faite la recherche du minimum. Il en est de même pour la suppression du minimum d'une

liste : tout algorithme sur une liste λ, dont le résultat est une liste où le minimum de λ apparaît une fois de moins que dans λ et où le nombre d'occurrences des autres éléments est inchangé est une implémentation correcte de *supprimer-min*(λ). Les différences entre les algorithmes de tri par sélection viennent des choix faits pour programmer les opérations *min* et *supprimer-min*.

Le fait que l'opération *tri-select* satisfait bien les axiomes établis au début de ce chapitre pour l'opération *tri* se démontre facilement et est laissé en exercice.

2.1. Sélection ordinaire

Si on recherche séquentiellement le minimum, et que l'on échange à chaque fois ce minimum avec le premier élément, on dit que l'on fait de la ***sélection ordinaire***.

2.1.1. Programmation de la sélection ordinaire

La procédure récursive *tri-select* donnée ci-dessous trie par sélection ordinaire. Si la liste à trier comporte n éléments rangés de l'indice 1 à l'indice n, le tri de la totalité de la liste est effectué par *tri-select*$(t, 1)$.

```
procedure tri-select (var t : array[1..n] of integer; i : integer);
var j, k : integer;
begin
    if i <n then begin
            {recherche séquentielle du minimum entre les indices i et n :}
            j := i;
            for k := i + 1 to n do if t[k] < t[j] then j := k;
            {placement du minimum :}
            t[j] ↔ t[i];
            {tri de la fin du tableau :}
            tri-select(t, i + 1)
    end
end tri-select;
```

On peut transformer cette version récursive du tri par sélection en une version itérative. L'appel récursif de *tri-select* est terminal : son exécution est équivalente à recommencer l'exécution de la procédure en changeant les valeurs des paramètres, et, ceci, tant que la condition $i < n$ est satisfaite. Il faut, évidemment, initialiser les paramètres aux valeurs qu'ils ont au premier appel de la procédure récursive (cf. le passage de la procédure *dicho* à la procédure *dichoiter* au chapitre 9). On obtient ainsi le programme itératif suivant (où la boucle **while** pourrait être remplacée par une boucle **for**) :

```
procedure tri-select-iter(var t : array[1..n] of integer);
var i, j, k : integer;
begin
    i := 1;
    while i < n do begin
        j := i;
        for k := i + 1 to n do if t[k] < t[j] then j := k;
        t[j] ↔ t[i];
        i := i + 1
    end
end tri-select-iter;
```

2.1.2. Analyse de la sélection ordinaire

Les deux programmes ci-dessus étant équivalents on peut raisonner indifféremment sur l'un ou l'autre pour les calculs de complexité. Si on raisonne sur la version récursive, on obtient des équations de récurrence que l'on résoud selon les méthodes présentées en annexe. Comme la plupart des méthodes de tri interne, cet algorithme effectue deux types d'opérations : des *comparaisons entre clés* et des *transferts d'éléments*. On compte le nombre de ces opérations fondamentales pour une liste de n éléments.

Nombre de comparaisons

Pour toute liste de taille n, on effectue exactement $n-1$ comparaisons pour trouver le minimum : en effet, on exécute $n-1$ itérations dans la boucle interne et à chaque itération il y a une comparaison. Cette recherche du minimum est suivie de l'exécution de l'algorithme pour une liste de $n-1$ éléments.

On a donc :

$$\begin{gathered}\mathrm{Max}_C(n) = n - 1 + \mathrm{Max}_C(n-1) \quad \text{pour } n > 1 \text{ et} \\ \mathrm{Max}_C(1) = 0\end{gathered}$$

et puisque le nombre de comparaisons ne dépend pas de la liste :

$$\mathrm{Moy}_C(n) = \mathrm{Max}_C(n).$$

L'équation de récurrence sur Max se résoud selon une méthode directe (cf. annexe) :

$$\mathrm{Max}_C(n) = \mathrm{Max}_C(1) + \sum_{i=1}^{n-1} i \;=\; \frac{n(n-1)}{2}$$

Les complexités en nombre de comparaisons pour la sélection ordinaire sont donc :

$$\mathrm{Max}_C(n) = \mathrm{Moy}_C(n) = \frac{n(n-1)}{2} = \Theta(n^2).$$

Nombre d'échanges

Là aussi la complexité au pire est égale à la complexité en moyenne puisqu'on fait un échange à chaque appel de la procédure. D'où pour le nombre d'échanges :

$$\begin{aligned} &\mathrm{Max}_E(n) = \mathrm{Moy}_E(n) \\ &\mathrm{Max}_E(n) = 1 + \mathrm{Max}_E(n-1) \quad \text{pour } n > 1 \text{ et} \\ &\mathrm{Max}_E(1) = 0 \end{aligned}$$

on obtient donc :

$$\mathrm{Max}_E(n) = \mathrm{Moy}_E(n) = n - 1.$$

Ce nombre doit être multiplié par trois pour obtenir le nombre de transferts (un échange correspond à trois transferts). En tout état de cause *les complexités au pire et en moyenne en nombre de transferts de la sélection ordinaire sont en* $\Theta(n)$.

2.2. Le tri à bulles

Il existe d'autres méthodes de tri par sélection. La plus simple à programmer est probablement le ***tri à bulles*** où on parcourt la liste en commençant par la fin, en effectuant un échange à chaque fois que l'on trouve deux éléments successifs qui ne sont pas dans le bon ordre. Sur la liste de la figure 5, l'exécution d'un tri à bulles s'effectue comme l'indique la figure 6. Les échanges y sont indiqués par le signe de double flèche.

2.2.1. Programmation du tri à bulles

Le programme s'obtient en remplaçant, dans la version itérative de la sélection ordinaire, la recherche séquentielle et le placement du minimum par un parcours qui effectue ces échanges en commençant par la fin.

A la fin du parcours descendant, le minimum de la partie de la liste qui restait à trier se trouve à la première place de cette partie, qui est $t[i]$ à la $i^{ième}$ itération (la démonstration de cette propriété est simple; elle se fait par récurrence sur la taille de la partie de la liste qui reste à trier).

Le nom de ce tri vient de ce que les éléments les plus petits (les plus "légers") remontent vers le début de la liste comme des bulles vers le haut d'un tube à essai. On peut améliorer cet algorithme de manière à éviter les parcours sans échanges (voir exercices).

```
procedure tri-bulles(var t : array[1.. n] of integer);
var i, j : integer;
begin
    i := 1;
    while i < n do begin
        for j := n downto i + 1 do
            if t[j] < t[j − 1] then t[j] ↔ t[j − 1];
        i := i + 1
    end
end tri-bulles;
```

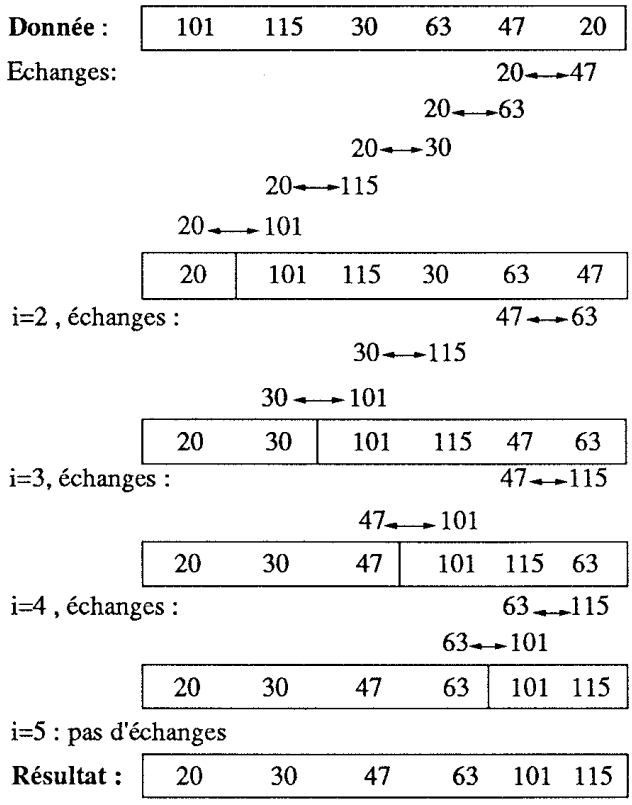


Figure 6. Exemple de tri à bulles.

2.2.2. Analyse du tri à bulles

Le *nombre de comparaisons* est le même que pour la sélection ordinaire. On a donc :

$$\mathrm{Max}_C(n) = \mathrm{Moy}_C(n) = \frac{n(n-1)}{2} = \Theta(n^2)$$

Le *nombre d'échanges* peut être *au pire* de $n-1$ au premier parcours, $n-2$ au deuxième, etc. C'est le cas si le tableau est trié en ordre strictement décroissant. On a donc :

$$\mathrm{Max}_E(n) = \sum_{i=1}^{(n-1)} (n-i) = \sum_{i=1}^{(n-1)} i = \frac{n.(n-1)}{2}$$

Pour avoir le nombre de transferts, ce nombre doit être multiplié par trois. Donc la complexité au pire en nombre de transferts du tri à bulles est en $\Theta(n^2)$.

On donne deux méthodes pour montrer que le nombre *d'échanges en moyenne* est d'ordre n^2 : la première est basée sur des dénombrements, la deuxième sur l'utilisation de séries génératrices. Dans les deux cas, on considère que tous les éléments sont différents (d'autres méthodes permettent d'obtenir un résultat du même ordre lorsque l'on donne une probabilité non nulle aux listes comportant des éléments égaux).

Calcul par dénombrement du nombre d'échanges en moyenne

Soit t de taille n et t' son miroir ($t'[1] = t[n]$, $t'[2] = t[n-1], \ldots, t'[n] = t[1]$). Si on exécute l'algorithme sur t et t', chaque paire d'éléments est échangée, soit dans t, soit dans t', jamais dans les deux. Donc pour le tri de t et t' on a, en tout, autant d'échanges qu'il y a de paires d'éléments d'indices différents, soit $n.(n-1)/2$ échanges, donc $3.n.(n-1)/2$ transferts.

Considérons l'ensemble T de tous les tableaux de taille n ne comportant pas d'éléments égaux. Supposons qu'ils sont tous équiprobables. D'après la définition de la complexité en moyenne (chapitre 2), si $coût_E(t)$ est le nombre d'échanges effectués pour trier le tableau t, et si $p(t)$ est la probabilité du tableau t, on a :

$$\mathrm{Moy}_E(n) = \sum_{t \in T} p(t).coût_E(t)$$

On peut partitionner T en T_c, ensemble des tableaux de taille n tels que $t[1] < t[n]$ et T_d, ensemble des tableaux de taille n tels que $t[1] > t[n]$. Si $t \in T_c$, alors $t' \in T_d$ et réciproquement. D'où :

$$\mathrm{Moy}_E(n) = \sum_{t \in T_c} p(t).coût_E(t) + \sum_{t' \in T_d} p(t').coût_E(t')$$

Tous les tableaux étant équiprobables, posons pour tout t, $p(t) = \pi$. On obtient :

$$\begin{aligned}\text{Moy}_E(n) &= \pi.(\sum_{t\in T_c} coût_E(t) + \sum_{t'\in T_d} coût_E(t')) \\ &= \pi.\sum_{t\in T_c} (coût_E(t) + coût_E(t')) \\ &= \pi.\sum_{t\in T_c} \frac{n.(n-1)}{2} = \pi.|T_c|.\frac{n.(n-1)}{2}\end{aligned}$$

Il existe une bijection entre T_c et T_d ; ils sont donc de même cardinalité, d'où :

$$\pi \cdot |T_c| = \frac{1}{2}$$

Par conséquent :

$$\text{Moy}_E(n) = \frac{1}{2}.\frac{n.(n-1)}{2} = \frac{n.(n-1)}{4}$$

Dans le cas où les éléments sont distincts, et où les listes sont de même probabilité on a, en nombre de transferts :

$$\text{Moy}_T(n) = \frac{3.n.(n-1)}{4} = \Theta(n^2)$$

Par conséquent, la complexité en moyenne en nombre de transferts du tri à bulles est en $\Theta(n^2)$.

Calcul par séries génératrices du nombre d'échanges en moyenne

Cette méthode fait appel aux techniques de séries génératrices introduites en annexe. Pour les algorithmes de tri par comparaisons, seul compte l'ordre relatif des éléments (comme pour l'algorithme de recherche du plus grand élément d'un tableau vu au chapitre 3). Donc dans le cas où tous les éléments sont distincts on ne restreint pas le problème en se limitant aux tableaux dont les éléments forment une permutation de l'ensemble des entiers de 1 à n.

Revenons à l'algorithme de tri à bulles : lors de l'exécution de cet algorithme, il y a un échange à chaque fois que deux éléments consécutifs ne sont pas dans le bon ordre. Le nombre total d'échanges lors de l'exécution du tri à bulles sur un tableau t est donc égal au nombre de couples (j, i) tels que $j < i$ et $t[j] > t[i]$: c'est le nombre de couples d'éléments du tableau qui ne sont pas dans le bon ordre ; avec les hypothèses rappelées ci-dessus, c'est donc le *nombre d'inversions* $Inv(\sigma)$ de la permutation σ représentée par les éléments du tableau. Si l'on considère que toutes les permutations de tableau sont équiprobables, calculer le nombre moyen

d'échanges du tri à bulles pour un tableau de n éléments revient donc à calculer le nombre moyen d'inversions $\text{Moy}_{\text{inv}}(n)$ dans une permutation de 1,...,n, toutes les $n!$ permutations étant équiprobables.

Notons Π_n l'ensemble des permutations sur $\{1, ..., n\}$.

$$\text{Moy}_{\text{inv}}(n) = \frac{1}{n!} \cdot \sum_{\sigma \in \Pi_n} Inv(\sigma)$$

Soit $I_{n,k}$ le nombre de permutations de Π_n ayant k inversions, c'est-à-dire :

$$I_{n,k} = |\{\sigma \in \Pi_n \quad \text{telles que } Inv(\sigma) = k\}|,$$

et soit $I_n(z)$ la série génératrice de dénombrement associée aux $(I_{n,k})_{k \geq 0}$.

$$I_n(z) = \sum_{k \geq 0} I_{n,k} \cdot z^k = \sum_{\sigma \in \Pi_n} z^{Inv(\sigma)}$$

Lemme : $I_n(z) = (1 + z + ... + z^{n-1}) \cdot I_{n-1}(z)$.

Preuve : Soit $\sigma = e_1 \ldots e_{n-1}$ une permutation de Π_{n-1} ; en insérant l'élément n (qui est maximal) dans toutes les positions possibles, on associe à σ, n permutations de Π_n :

$\tau_1 = n e_1 e_2 ..... e_{n-1}$ (insertion en position 1)
$\tau_2 = e_1 n e_2 ..... e_{n-1}$ (insertion en position 2)
...
$\tau_i = e_1 .... e_{i-1} n e_i .... e_{n-1}$ (insertion en position i)
...
$\tau_n = e_1 e_2 .... e_{n-1} n$ (insertion en position n)

Le nombre d'inversions de chaque permutation τ_i se calcule facilement à partir du nombre d'inversions de σ : $Inv(\tau_1) = Inv(\sigma) + (n-1)$; en effet l'insertion du plus grand élément en tête rajoute les $(n-1)$ couples $(n, e_i)_{i=1,...,n-1}$ aux couples d'éléments de σ qui ne sont pas dans le bon ordre.

De même

$$Inv(\tau_2) = Inv(\sigma) + (n-2), \ \ldots, \ Inv(\tau_{n-1}) = Inv(\sigma) + 1, \ Inv(\tau_n) = Inv(\sigma).$$

Pour calculer $I_n(z) = \sum_{\tau \in \Pi_n} z^{Inv(\tau)}$, on partitionne l'ensemble Π_n en n sous-ensembles, selon la place du plus grand élément :

$$\Pi_n = \bigcup_{i=1...n} P_{n,i}$$

où $P_{n,i}$ est l'ensemble des permutations de $\{1, ..., n\}$ telles que l'élément n est à la place i. Chaque sous-ensemble $P_{n,i}$ est en bijection avec Π_{n-1} : toute permutation τ de $P_{n,i}$ résulte de l'insertion de l'élément n à la position i dans une et une seule permutation σ de Π_{n-1}. De plus si τ résulte de l'insertion de l'élément n en position i dans σ, on a vu que $Inv(\tau) = Inv(\sigma) + (n - i)$. Il en résulte que

$$I_n(z) = \sum_{\tau \in \Pi_n} z^{Inv(\tau)} = \sum_{i=1}^{n} \left(\sum_{\tau \in P_{n,i}} z^{Inv(\tau)} \right)$$

$$= \sum_{i=1}^{n} \left(\sum_{\sigma \in \Pi_{n-1}} z^{Inv(\sigma)+(n-i)} \right) = \sum_{i=1}^{n} z^{n-i} \cdot \sum_{\sigma \in \Pi_{n-1}} z^{Inv(\sigma)}$$

d'où $I_n(z) = (1 + z + ... + z^{n-1}) \cdot I_{n-1}(z)$, ce qui démontre le lemme. □

La relation de récurrence établie dans le lemme ci-dessus, avec la condition initiale $I_1(z) = 1$ (car la permutation réduite à un seul élément 1 n'a pas d'inversion), se résoud immédiatement en :

$$I_n(z) = (1 + z).(1 + z + z^2)...(1 + z + ... + z^{n-1})$$

La moyenne $\text{Moy}_{inv}(n)$ du nombre d'inversions dans une permutation de Π_n s'exprime alors facilement à l'aide de la fonction $I_n(z)$ et de sa dérivée (cf annexe) :

$$\text{Moy}_{inv}(n) = \frac{I'_n(1)}{I_n(1)}$$

Or $\text{Log}(I_n(z)) = \sum_{i=1}^{n-1} \text{Log}(1 + z + ... + z^i)$; la dérivée logarithmique de $I_n(z)$ est donc

$$\frac{I'_n(z)}{I_n(z)} = \sum_{i=1}^{n-1} \frac{1 + 2z + \ldots + iz^{i-1}}{1 + z + \ldots + z^i}$$

d'où pour $z = 1$:

$$\frac{I'_n(1)}{I_n(1)} = \sum_{i=1}^{n-1} \frac{1 + 2 + \ldots + i}{i + 1} = \sum_{i=1}^{n-1} \frac{i(i+1)}{2(i+1)} = \frac{n(n-1)}{4}$$

On a ainsi montré que la valeur moyenne du nombre d'échanges dans le tri à bulles d'un tableau de n éléments est $\frac{n(n-1)}{4}$.

Notons que la méthode utilisée ici permet aussi de calculer d'autres caractéristiques de la distribution du nombre d'échanges (cf. annexe).

En conclusion, on voit que la complexité en nombre de transferts du tri à bulles est d'ordre supérieur à celui de la complexité de la sélection ordinaire; il a donc peu d'intérêt en pratique. Notons cependant que si la liste est triée, on ne fait aucun transfert, mais le nombre de comparaisons est inchangé.

Les deux méthodes de tri par sélection vues ici, ne conservent pas, d'un parcours à l'autre, les résultats des comparaisons effectuées. On refait donc plusieurs fois les mêmes comparaisons. Intuitivement, il apparaît qu'en stockant, sous forme d'arbre par exemple, les résultats de ces comparaisons on devrait pouvoir améliorer la complexité, de ce point de vue. On verra au chapitre 15 une méthode de tri basée sur ce principe : le tri par tas.

3. Méthodes par insertion

Dans ces méthodes, on trie successivement les premiers éléments de la liste : à la $i^{ième}$ étape on insère le $i^{ième}$ élément à son rang parmi les $i-1$ éléments précédents qui sont déjà triés entre eux. Cette méthode s'appelle aussi la *méthode du joueur de cartes*. La figure 7 donne les étapes successives d'un tel tri.

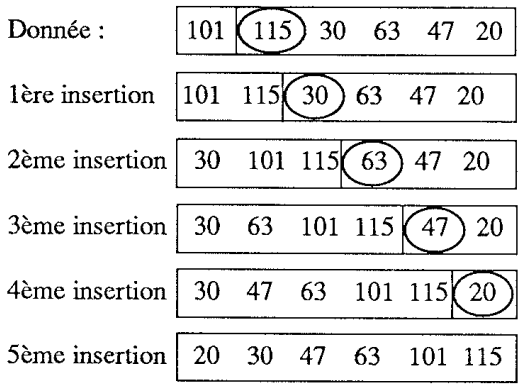


Figure 7. Exemple de tri par insertion.
La liste est triée jusqu'à la barre verticale; l'élément à insérer est encerclé.

Considérons les opérations suivantes sur les listes (déjà mentionnées au chapitre 5) : l'opération *début* donne, pour une liste non vide, la liste privée de son dernier élément; l'opération *dernier* donne le dernier élément d'une liste non vide. Soit les variables λ de sorte Liste, e de sorte Elément. La spécification d'un tri par insertion s'écrit de la manière suivante :

opérations
tri-insert : Liste → Liste
insérer : Elément × Liste → Liste

précondition
insérer(e, λ) **est-défini-ssi** *triée*(λ) = vrai

axiomes
tri-insert(*liste-vide*) = *liste-vide*
$\lambda \neq$ *liste-vide* $\Rightarrow$ *tri-insert*(λ) = *insérer*(*dernier*(λ), *tri-insert*(*début*(λ)))
triée(*insérer*(e, λ)) = vrai
perm(*cons*(e, λ), *insérer*(e, λ)) = vrai

Les algorithmes de tri par insertion se programment naturellement de manière récursive. On a différents algorithmes de tri par insertion selon la méthode employée pour rechercher le rang de l'élément à insérer parmi le début de liste déjà trié.

3.1. Insertion séquentielle

Si on recherche la place où insérer le nouvel élément de manière séquentielle, on dit que l'on fait de l'***insertion séquentielle.***

Soit t le tableau représentant la liste à trier; pour faire la $i^{ième}$ insertion, on commence la recherche de la place de $t[i]$ dans le début de tableau $t[1..i-1]$ par la fin; cela permet d'effectuer en même temps le décalage de la fin de ce sous-tableau, ce qui est nécessaire avant de faire l'insertion (voir figure 8).

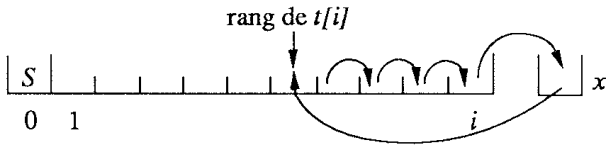


Figure 8. Principe de l'insertion séquentielle.

3.1.1. Programmation de l'insertion séquentielle

A l'indice 0, on range une *sentinelle* S, ici une valeur plus petite que tous les éléments de la liste à trier. Cela évite de tester si l'indice courant n'est pas inférieur à 1. La procédure s'écrit de la manière suivante :

```
procedure tri-insert(var t : array[0..n] of integer; i : integer);
var k, x : integer;
begin
    if i > 1 then begin
            tri-insert(t, i − 1); {tri du début de la liste}
            {recherche du rang de t[i]}
            k := i − 1; x := t[i];
            while t[k] > x do begin t[k + 1] := t[k]; k := k − 1 end;
            {on a t[k] ≤ x ; la place de x est k + 1}
            t[k + 1] := x;
    end
end tri-insert;
```

Le tri du tableau complet se fait en initialisant $t[0]$ à la valeur voulue et en appelant *tri-insert*(t, n).

La procédure *tri-insert* n'est pas récursive terminale et on ne peut donc pas utiliser la même transformation que pour le tri par sélection pour obtenir un programme itératif. Il est cependant possible de construire un tel programme en raisonnant sur ce qui se passe à l'exécution d'une procédure récursive du type suivant :

```
procedure P(X);
begin ... D; P(α); F ...
end P;
{X intervient dans D, α et F}
```

On sauvegarde, à la fin de l'exécution de D, la valeur de X sur une pile. Puis on exécute l'appel récursif, et on restaure la valeur de X avant d'exécuter F. L'utilisation d'une pile n'est pas surprenante : la dérécursivation revient à programmer le parcours de l'arbre des appels récursifs (chapitre 7).

La procédure de tri par insertion est une procédure récursive de ce type. Mais c'est un cas dégénéré puisque D est vide et que α est très simple ($\alpha = X - 1$). On va donc, pour un appel *tri-insert*(t, n), empiler successivement n, $n - 1$,..., 2, sans rien faire d'autre dans un premier temps. Puis on récupèrera sur la pile les valeurs 2, 3,..., n pour lesquelles on exécutera la fin de la procédure. Un appel de *tri-insert*(t, n) est donc équivalent à :

```
procedure tri-insert-iter(var t : array [0..n] of integer);
var k, x : integer;
begin
    for i := 2 to n do begin
            k := i − 1; x := t[i];
            while t[k] > x do begin t[k + 1] := t[k]; k := k − 1 end;
            t[k + 1] := x
    end
end tri-insert-iter;
```

N.B. : Cette équivalence est vraie en ce qui concerne le nombre de comparaisons entre clés et transferts d'éléments. Par contre, cette transformation ne conserve pas les comparaisons entre indices. Ces opérations n'étant pas prises en compte pour les calculs de complexité, on peut raisonner indifféremment sur l'un ou l'autre programme pour analyser l'insertion séquentielle.

3.1.2. Analyse de l'insertion séquentielle

Nombre de comparaisons

Dans le pire des cas, on doit faire i comparaisons après l'appel de *tri-insert*$(t, i-1)$ (ici il s'agit des comparaisons entre x et les $t[k]$, pour k allant de $i-1$ à 0). Cela est vrai pour tous les appels de *tri-insert*(t, i) si le tableau t est trié en ordre inverse. On a donc :

$$\mathrm{Max}_C(n) = \mathrm{Max}_C(n-1) + n \quad \text{pour } n > 1; \quad \text{et } \mathrm{Max}_C(1) = 0$$

d'où

$$\mathrm{Max}_C(n) = \frac{n.(n+1)}{2} - 1$$

La complexité au pire en nombre de comparaisons pour l'insertion séquentielle est donc en $\Theta(n^2)$.

Pour la complexité en moyenne, on utilise les résultats établis pour l'analyse du tri à bulles : on se restreint ici encore au cas où tous les éléments du tableau sont différents. Soit $c(t[i])$ le nombre de comparaisons pour insérer à sa place le $i^{ième}$ élément du tableau dans la sous-liste triée des $(i-1)$ premiers éléments du tableau t. L'examen de l'algorithme montre que $c(t[i])$ est égal au nombre, augmenté de 1, des éléments situés à gauche de $t[i]$ dans le tableau, et qui sont plus grands que $t[i]$.

Le nombre de comparaisons pour trier t est donc $c(t) = \sum_{i=2}^{n}(1 + INV(i))$, où $INV(i)$ représente le nombre d'indices $j < i$ tels que $t[j] > t[i]$. Si les éléments de t sont une permutation σ de $\{1, \ldots, n\}$, on a donc $c(t) = (n-1) + Inv(\sigma)$, où $Inv(\sigma)$ représente le nombre d'inversions de σ (cf. la deuxième méthode pour l'analyse du nombre moyen d'échanges dans le tri à bulles). Si l'on fait l'hypothèse que toutes les $n!$ permutations des éléments sont équiprobables, alors le nombre moyen de comparaisons pour trier un tableau de n éléments est

$$\mathrm{Moy}_C(n) = \frac{1}{n!} \sum_{\sigma \in \Pi_n} (n - 1 + Inv(\sigma)) = (n-1) + \mathrm{Moy}_{\mathrm{inv}}(n)$$

où $\mathrm{Moy}_{\mathrm{inv}}(n)$ est le nombre moyen d'inversions dans une permutation de n éléments : on a montré précédemment qu'il est égal à $\frac{n(n-1)}{4}$.

La complexité en moyenne en nombre de comparaisons pour trier n éléments par l'algorithme d'insertion séquentielle est donc en $\Theta(n^2)$. On a exactement

$$\text{Moy}_C(n) = \frac{n^2 + 3n}{4} - 1.$$

Nombre de transferts

Le nombre de transferts est lié de façon très simple au nombre de comparaisons : à chaque appel de la procédure *tri-insert* avec $i > 1$, le nombre de transferts est égal au nombre de comparaisons augmenté de 1.

Pour toute liste t de n éléments, il y a $(n-1)$ appels de tri-insert avec $i > 1$; le nombre $T(t)$ de transferts d'éléments effectués lors du tri est donc :

$$T(t) = c(t) + (n-1).$$

Le nombre de transferts pour un tableau de n éléments est donc :

$$\text{Max}_T(n) = \frac{n^2}{2} + \frac{3n}{2} - 2$$
$$\text{Moy}_T(n) = \frac{n^2}{4} + \frac{7n}{4} - 2$$

La complexité au pire et en moyenne en nombre de transferts pour l'insertion séquentielle est en $\Theta(n^2)$.

En conclusion, le tri par insertion séquentielle est donc en général plus coûteux que la sélection ordinaire, surtout au niveau des transferts. Cependant ces transferts peuvent être évités si on représente la liste de manière chaînée. Par ailleurs, si la liste est presque triée, cette méthode effectue moins de comparaisons que la sélection ordinaire (si la liste est déjà triée, on fait $n-1$ comparaisons).

3.2. Insertion dichotomique

On a vu au chapitre 9 que l'on pouvait faire de la recherche dichotomique dans une liste triée. Au cours du tri par insertion, les $i-1$ premiers éléments sont triés et on cherche le rang du $i^{ième}$. Il est donc possible d'employer cette méthode.

3.2.1. Programmation de l'insertion dichotomique

La procédure devient :

```
procedure tri-insert-dicho(var t : array[1..n] of integer; i : integer);
var j, k, x : integer;
begin
    if i > 1 then begin
        tri-insert-dicho(t, i − 1);
        if t[i − 1] > t[i] then begin
        {la fonction "rang" est donnée plus loin; elle rend l'indice où
        doit être inséré t[i] entre les indices 1 et i − 1, si t[i] < t[i − 1]}
            k := rang(t, 1, i − 1, t[i]);
            {il reste à faire les transferts :}
            x := t[i];
            for j := i − 1 downto k do t[j + 1] := t[j];
            t[k] := x
        end
    end
end tri-insert-dicho;
```

La fonction *rang* a pour résultat l'indice où doit être inséré l'élément x entre les indices p et q de t quand x est inférieur à $t[q]$:

```
function rang(t : array[1..n] of integer; p, q, x : integer) : integer;
var mil : integer;
begin
    if p = q then return (p)
    else begin
        mil := (p + q) div 2;
        if x < t[mil] then return (rang(t, p, mil, x))
                      else return (rang(t, mil+1, q, x))
    end
end rang;
```

Remarque : le test $x < t[mil]$ assure que le tri est stable : on insère toujours *après* les éléments égaux, s'il y en a.

3.2.2. Analyse de l'insertion dichotomique

Le *nombre de transferts* est le même que pour l'insertion séquentielle : c'est normal car la seule différence porte sur la méthode de calcul du rang où insérer. Cependant, on ne peut plus éviter ces transferts en travaillant sur une liste chaînée : la recherche dichotomique nécessite un accès direct à l'élément du milieu de la liste, donc une représentation contiguë.

Par contre, le *nombre de comparaisons* est amélioré. En effet, pour trier une liste de n éléments, on commence par trier les $n - 1$ premiers éléments, puis on a une comparaison entre $t[n-1]$ et $t[n]$ qui entraîne, dans le pire des cas un appel de la fonction *rang* sur une sous-liste de $n - 1$ éléments . D'une manière similaire à ce

qui a été fait au chapitre 9 sur la recherche dichotomique, on montre que l'appel de *rang* sur une liste de n éléments implique $\lceil \log n \rceil$ comparaisons entre éléments au pire. En effet, en nombre de comparaisons entre éléments, on a :

$$\mathrm{Max}_{rang}(n) = 1 + \mathrm{Max}_{rang}(\lceil n/2 \rceil) \text{ pour } n > 1 \quad \text{et} \quad \mathrm{Max}_{rang}(1) = 0$$

On a pour le tri par insertion dichotomique :

$$\mathrm{Max}_C(n) = 1 + \mathrm{Max}_C(n-1) + \mathrm{Max}_{rang}(n-1)$$

On a donc :

$$\mathrm{Max}_C(n) = \mathrm{Max}_C(n-1) + 1 + \lceil \log(n-1) \rceil \quad \text{et} \quad \mathrm{Max}_C(1) = 0$$

Soit

$$\mathrm{Max}_C(n) = \sum_{i=2}^{n} \lceil \log(i-1) \rceil + (n-1) = \sum_{i=1}^{n-1} \lceil \log i \rceil + (n-1)$$

Or $\log i \leq \lceil \log i \rceil \leq (\log i) + 1$. L'encadrement d'une somme discrète par des intégrales, pour une fonction croissante (cf. annexe) permet de montrer que :

$$\int_1^{n-1} \log x \, dx \leq \sum_{i=2}^{n-1} \log i \leq \int_2^n \log x \, dx$$

En intégrant par parties $\log(x) = \mathrm{Log}(x)/\mathrm{Log}(2)$, on obtient l'encadrement suivant, où a, b, c, d sont des constantes :

$$(n-1).\log(n-1) + a.n + b \leq \sum_{i=2}^{n-1} \log i \leq n.\log n + c.n + d$$

d'où,

$$(n-1).\log(n-1) + a.n + b \leq \sum_{i=2}^{n-1} \lceil \log i \rceil \leq n.\log n + c.n + d + n - 2$$

On en déduit :

$$(n-1).\log(n-1) + a.n + b \leq \mathrm{Max}_C(n) \leq n.\log n + c'.n + d'$$

où c' et d' sont des constantes.

Il en résulte que $\mathrm{Max}_C(n) = \Theta(n \log n)$.

Pour trouver quelles sont les listes qui correspondent à cette complexité au pire, il suffit de remarquer que maximiser le nombre de comparaisons, c'est maximiser le nombre d'appels de la fonction *rang*. Cette fonction est appelée soit sur la sous-liste de gauche, soit sur la sous-liste de droite, jusqu'à ce qu'on obtienne une liste de

longueur 1. Comme pour la procédure *dichoprem* vue au chapitre 9, maximiser le nombre d'appels de la fonction *rang*, c'est maximiser la taille des sous-listes sur lesquelles on appelle la fonction. Or, le sous-tableau de gauche est de longueur $\lceil n/2 \rceil$, celui de droite $\lfloor n/2 \rfloor$. Il faut donc toujours appeler *rang* sur la sous-liste de gauche. Une liste triée en ordre décroissant correspond donc au pire des cas.

Pour l'insertion dichotomique on a donc une complexité au pire, en nombre de comparaisons qui est en $\Theta(n \log n)$. On verra que c'est un résultat optimal pour les algorithmes de tri. On montrera aussi dans le paragraphe sur l'optimalité du tri que le nombre moyen de comparaisons de tout algorithme de tri ne peut pas être d'ordre de grandeur inférieur à $n \log n$. Donc dans le tri par insertion dichotomique, puisque le cas le pire nécessite $\Theta(n \log n)$ comparaisons, il faut aussi $\Theta(n \log n)$ comparaisons en moyenne. Malheureusement, l'insertion dichotomique a une complexité au pire et en moyenne, pour ce qui est des transferts, en $\Theta(n^2)$.

4. Conclusion

On a présenté dans ce chapitre quatre méthodes de tri simple; on donne un tableau récapitulatif de leurs complexités à la figure 9. Un autre élément de comparaison entre ces méthodes est que les méthodes de sélection donnent à l'étape i, le début du futur tableau trié. On dit que le tri est ***progressif***. Cela permet de commencer éventuellement un traitement en parallèle sur ce tableau. Cela n'est pas possible si on utilise une méthode par insertion. Par contre si le début du tableau à trier est déjà trié, il faut faire de l'insertion.

ALGORITHME	NOMBRE DE COMPARAISONS		NOMBRE DE TRANSFERTS	
	Moy	Max	Moy	Max
SELECTION ORDINAIRE	$\frac{n(n-1)}{2}$	$\frac{n(n-1)}{2}$	$3(n-1)$	$3(n-1)$
TRI A BULLES	$\frac{n(n-1)}{2}$	$\frac{n(n-1)}{2}$	$\frac{3n(n-1)}{4}$	$\frac{3n(n-1)}{2}$
INSERTION SEQUENTIELLE	$\frac{n(n+3)}{4}-1$	$\frac{n(n+1)}{2}-1$	$\frac{n(n+7)}{4}-2$	$\frac{n(n+3)}{2}-2$
INSERTION DICHOTOMIQUE	$\Theta[n\log n]$	$\Theta[n\log n]$	$\frac{n(n+7)}{4}-2$	$\frac{n(n+3)}{2}-2$

Figure 9. Complexités des méthodes de tri simples.

Exercices

1. Retrouver la complexité Max(n) du nombre de comparaisons dans le pire des cas, de la sélection ordinaire, en partant du programme itératif.

2. Montrer que dans le tri à bulles, à la fin du parcours descendant, le minimum de la partie du tableau qui restait à trier se trouve à la première place de cette partie.

3. Quelle est la complexité optimale du nombre moyen de comparaisons d'un algorithme de tri qui procède par échanges d'éléments adjacents uniquement? Comment se situe le tri à bulles par rapport à cette complexité optimale?

4. Soit $e_1, e_2, ..., e_n$ une permutation de $\{1, ..., n\}$, et soit i et j des indices tels que $i < j$ et $e_i > e_j$. Soit $e'_1, e'_2, ..., e'_n$ la permutation obtenue à partir de $e_1, e_2, ..., e_n$ en échangeant e_i et e_j. Evaluer le nombre d'inversions de $e'_1, e'_2, ..., e'_n$ par rapport à celui de $e_1, e_2, ..., e_n$.

5. Cet exercice propose d'améliorer l'algorithme de tri à bulles.

(i) On a vu que cet algorithme peut effectuer des parcours sans échange. Une première amélioration consiste à éviter ces parcours inutiles.

(ii) On peut aussi mémoriser le plus grand indice à partir duquel les échanges ne se font plus, c'est-à-dire, l'indice en dessous duquel la liste est triée.

(iii) Enfin, on remarquera une certaine dissymétrie à l'exécution de l'algorithme de tri à bulles sur les deux listes suivantes : 2, 3, 4, 5, 6, 7, 8, 1 et 8, 1, 2, 3, 4, 5, 6, 7.

Dans les deux cas, un seul élément n'est pas à sa place, mais dans le premier cas, la liste est triée à la fin du premier parcours descendant car l'élément qui n'est pas à sa place est à la fin de la liste, tandis que dans le second cas, il faut 7 parcours descendants pour obtenir une liste triée, car l'élément qui n'est pas à sa place est au début de la liste. Pour remédier à cet inconvénient, on peut alterner la direction des parcours consécutifs.

Ecrire une procédure de tri basée sur le principe du tri à bulles qui prenne en compte ces trois améliorations. Un tel tri s'appelle ***tri shaker.***

6. Le but de l'exercice est d'étudier une méthode de tri par insertion d'une liste t de n éléments, moins coûteuse que le tri par insertion séquentielle : le ***tri shell.***

Au lieu d'insérer chaque élément à sa place dans la sous-liste triée de tous les éléments qui le précèdent, on l'insère dans une sous-liste d'éléments qui le précèdent, distants d'un certain incrément *incr*, que l'on diminue au cours de passages successifs sur la liste. C'est pourquoi on appelle parfois cette méthode, ***tri par incrément décroissant.*** Au 1^{er} passage, on forme des sous-listes d'éléments distants de $n/2$ que l'on trie séparément. Au $2^{ème}$ passage, on forme des sous-listes d'éléments distants de $n/4$, que l'on trie séparément, et ainsi de suite. Par exemple, si $n = 8$, on a d'abord $n/2$ sous-listes $(t[i], t[i + n/2])(1 \leq i \leq 4)$ à trier, puis $n/4$ sous-listes $(t[i], t[i+n/4], t[i+n/2], t[i+3n/4])(1 \leq i \leq 2)$ à trier, enfin la liste des 8 éléments à trier. A chaque passage, l'incrément est divisé par deux et toutes les sous-listes sont triées; les éléments d'une même sous-liste sont triés par insertion séquentielle. Le tri s'arrête lorsque l'incrément est nul. On obtient la procédure suivante :

```
procedure trishell(var t : array [1..n] of integer);
var i, j, incr : integer;
begin
    incr := n div 2;
    while incr > 0 do begin
        for i := incr+1 to n do begin
            j := i-incr;
            while j > 0 do
                if t[j] >t[j + incr] then begin
                    t[j] ↔ t[j + incr]; j := j - incr end
                else j := 0
        end;
        incr := incr div 2
    end
end trishell;
```

a) montrer que si deux éléments $t[i]$ et $t[i + n/2^k]$ ont été échangés au $k^{ième}$ passage, alors ils restent triés par la suite.

b) exécuter la procédure *trishell* sur la suite d'entiers : 6, 8, 1, 4, 3, 7, 2, 9.

c) vérifier qu'après exécution de la procédure *trishell*, on obtient bien une liste triée.

On peut montrer qu'un tel tri a une complexité au pire en $O(n^{3/2})$. De plus, on conjecture (d'après des mesures) que si on ne prend pas comme suite d'incréments

$1, .., n/2^{i+1}, n/2^i, ...,$ *mais la suite* $1, 3, 7, 15, 31, \ldots,$ *c'est-à-dire* $1, \ldots, h_i, 2h_i+1, \ldots$ *cette complexité est en* $O(n^{1.2})$.

7. Dans cet exercice on propose une nouvelle méthode de tri, dite par ***énumération***. Soit à trier une liste t de taille n. On compare chaque clé C_i à chaque clé C_j et on détermine le nombre n_j de clés C_i telles que $C_i < C_j$, pour $j = 1, ..., n$. Le rang définitif de C_j dans la liste triée est déterminé par $n_j + 1$. On suppose que les clés sont toutes distinctes.

a) Ecrire un algorithme qui construit le tableau des n_j, puis réorganise la liste des éléments sur place.

b) Faire tourner l'algorithme sur la suite d'entiers 4, 7, 10, 3, 15, 12, 1, 8, 2, 6.

c) Evaluer la complexité au pire et en moyenne de cet algorithme en nombre de comparaisons, en nombre de transferts, et en place. Comparer cet algorithme avec d'autres algorithmes de tri.

8. On se donne un tableau de n éléments tel qu'il existe un rang p en dessous duquel le tableau est trié; $n - p$ est petit et p n'est pas connu; les éléments de la zone non triée peuvent être inférieurs à certains éléments de la zone triée. Comment utiliser ou adapter dans ce cas les méthodes de tri "simples" vues dans ce chapitre? Donner leur complexité au pire, en nombre de comparaisons, en fonction de n et de p.

Chapitre 15
Tri rapide et tri par tas

On a vu au chapitre précédent des méthodes de tri, dites simples, dont les complexités, soit en nombre de transferts, soit en nombre de comparaisons sont en $\Theta(n^2)$. Par exemple, pour le tri par insertion dichotomique, le nombre de comparaisons au pire est en $\Theta(n \log n)$, le nombre de transferts restant en $\Theta(n^2)$. Ce chapitre présente des méthodes de tri plus évoluées : le tri rapide, le tri par tas, et plus brièvement le tri fusion. Ces méthodes ont des complexités en nombres de comparaisons entre éléments en $\Theta(n \log n)$, en moyenne, et même dans le pire des cas en ce qui concerne le tri par tas et le tri fusion. Les complexités en nombres de transferts sont du même ordre ou d'ordre inférieur.

1. Le tri rapide

Le ***tri rapide*** (en anglais, quicksort) est un exemple de *tri par dichotomie* : on partage la liste à trier en deux sous-listes, telles que tous les éléments de la première soient inférieurs à tous les éléments de la seconde. On recommence jusqu'à avoir des sous-listes réduites à un élément. On pourrait utiliser, pour faire cette dichotomie, une représentation binaire des clés (si elle coïncide avec la relation d'ordre choisie !). On commencerait par faire un partage selon le bit de plus fort poids. On obtiendrait alors deux sous-listes, la première où toutes les clés commencent par un 0, la seconde par un 1. On recommencerait ensuite sur le bit suivant. On verra de telles méthodes au chapitre 17, pour le tri externe. Les méthodes étudiées ici sont basées sur cette idée de partage de la liste, mais les critères utilisés pour le partage sont indépendants de la représentation des clés.

1.1. Principe

Pour partager la liste en deux sous-listes, on choisit une des clés de la liste (par exemple celle du premier élément) et on l'utilise comme ***pivot*** : on construit une sous-liste avec les éléments dont la clé est inférieure ou égale à cette clé et une sous-liste avec les éléments dont la clé est supérieure. Comme on fait cette construction

sur place, en utilisant le tableau qui représente la liste, la frontière entre les sous-listes donne de ce fait la place définitive de l'élément dont la clé a servi de pivot. On peut donc le placer définitivement à la fin de la partition.

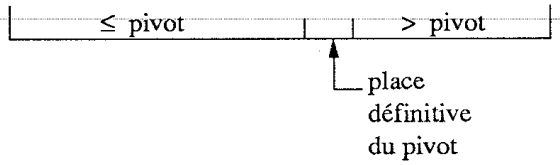


Figure 1. Partition du tableau dans le tri rapide et placement du pivot.

On recommence ensuite le même processus sur les sous-listes. Un exemple du déroulement d'un tel tri est donné figure 2, où le pivot choisi est toujours le premier élément.

Supposons que l'on ait une procédure *placer*(t, i, j, k) qui effectue la partition du sous-tableau compris entre les indices i et j en fonction du pivot $t[i]$ et rend comme résultat l'indice k où ce pivot a été placé. On considère que les éléments de la liste à trier sont des entiers et que la clé d'un élément est l'élément lui-même. La procédure de tri d'un tableau t entre les indices i et j s'écrit alors :

```
procedure tri-rapide(var t : array [1..n + 1] of integer; i, j : integer);
{on verra plus loin que t[n + 1] contient une sentinelle}
var k : integer;
begin
    if i < j then begin
            placer(t, i, j, k);
            {effectue la partition de t et le placement de t[i] en k}
            tri-rapide(t, i, k − 1);
            tri-rapide(t, k + 1, j)
    end
end tri-rapide;
```

L'appel de *tri-rapide*$(t, 1, n)$ provoque le tri de la liste complète.

1.2. Algorithme de partition et placement

La programmation de la procédure de partition et de placement ne nécessite qu'un parcours du tableau si on procède de la manière suivante : on utilise deux compteurs, l et k, qui partent des deux extrémités du tableau et vont l'un vers l'autre. Le compteur l part du début du tableau, et lorsqu'il atteint un élément dont la clé est supérieure au pivot, on arrête sa progression. De même, le compteur k part de la fin du tableau et lorsqu'il atteint un élément dont la clé est inférieure ou égale au pivot, on arrête également sa progression. On échange alors ces deux éléments. Puis on continue à faire progresser les compteurs et à faire des échanges jusqu'à ce que les valeurs des compteurs se croisent.

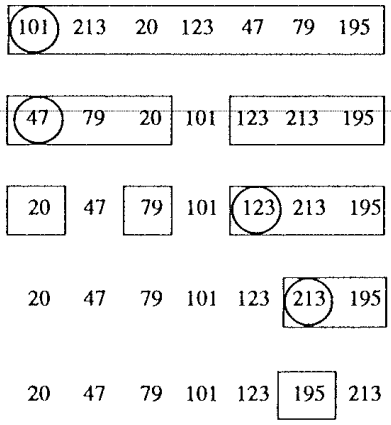


Figure 2. Exemple de tri rapide.
Les pivots sont entourés d'un cercle ; les sous-listes sont encadrées.

Si on reprend l'exemple de la figure 2, la première exécution de la procédure placer provoque l'échange de 213 et 79, puis de 47 et 123.

La programmation de cette procédure est facilitée si on considère que l'on a, à la place $t[j+1]$, une sentinelle, ici un élément x dont la clé est plus grande que celles de tous les éléments de la sous-liste à traiter. Quand on appelle *placer* sur une partie de tableau qui n'est pas en fin de tableau, cette sentinelle existe : l'élément qui se trouve à l'indice $j+1$ est un ancien pivot et tous les éléments entre i et j lui sont donc inférieurs. Il suffit donc d'ajouter un élément de clé maximum à la fin de la liste complète, c'est-à-dire à la place $t[n+1]$.

Sous cette condition, en considérant que les éléments sont des entiers et que la clé d'un élément est l'élément lui-même, la procédure *placer* s'écrit :

```
procedure placer(var t : array [1..n+1] of integer; i,j : integer;
                 var k : integer);
var l : integer;
begin
    l := i+1; k := j;
    while l ≤ k do begin
        while t[k] > t[i] do k := k-1; {t[k] ≤ t[i]}
        while t[l] ≤ t[i] do l := l+1; {t[l] > t[i]}
        if l < k then begin t[l] ↔ t[k]; l := l+1; k := k-1 end
    end;
    t[i] ↔ t[k]
end placer;
```

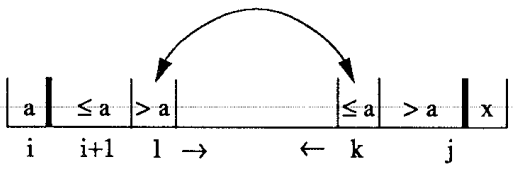
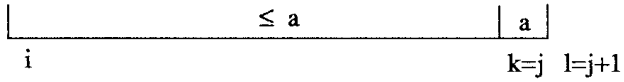


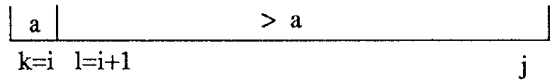
Figure 3. Partition et placement.

Il est nécessaire d'examiner avec soin le comportement dans les cas limites de cette procédure. Supposons que la valeur du pivot soit a.

Dans le cas où tous les éléments à trier sont inférieurs ou égaux à a, la variable k va garder la valeur j ; l va augmenter, et sa croissance sera stoppée par la rencontre de la sentinelle dans le troisième **while**. On n'effectue pas d'échange car on a $l = j + 1$ et $k = j$ et on sort de la boucle **while** la plus externe. On place ensuite $t[i]$ à sa place, qui est donnée par j.



Dans le cas où tous les éléments à trier sont supérieurs à a, la variable l va garder la valeur $i + 1$; k va décroître et s'arrêtera, par le test $t[k] > t[i]$ du deuxième **while**, à la valeur i ; on n'effectue pas d'échange car on a $k = i$ et $l = i + 1$ et on sort de la boucle **while** la plus externe. On échange alors $t[i]$ et $t[k]$, ce qui revient à laisser $t[i]$ à sa place puisque $k = i$.



Dans le cas général, où le pivot ne correspond pas à un maximum ou un minimum, k et l vont également se croiser : la procédure termine donc ; on montre ci-dessous que les éléments seront alors bien placés par rapport au pivot.

Considérons le dernier échange effectué au cours d'une partition, dans la boucle **while** la plus externe. Notons l_d et k_d les indices en cause : comme l'échange a lieu, on a forcément $l_d < k_d$. Après ce dernier échange, on a deux cas de figure possibles, comme le montre la figure 4.

Dans le premier cas, les derniers éléments échangés sont voisins : $k_d = l_d + 1$. Dans ce cas, l et k se croisent du fait de leurs mises à jour juste après l'échange : k vaut alors $k_d - 1$, c'est-à-dire l_d, et l vaut $l_d + 1$ c'est-à-dire k_d. De ce fait on sort de la boucle **while** la plus externe, et on échange $t[k]$, qui contient bien un élément inférieur ou égal à a, et $t[i]$.

Dans le deuxième cas, la sous-liste des éléments compris entre $t[k_d]$ et $t[l_d]$ est non vide : $k_d > l_d + 1$. Comme $t[k_d]$ et $t[l_d]$ sont les derniers éléments échangés,

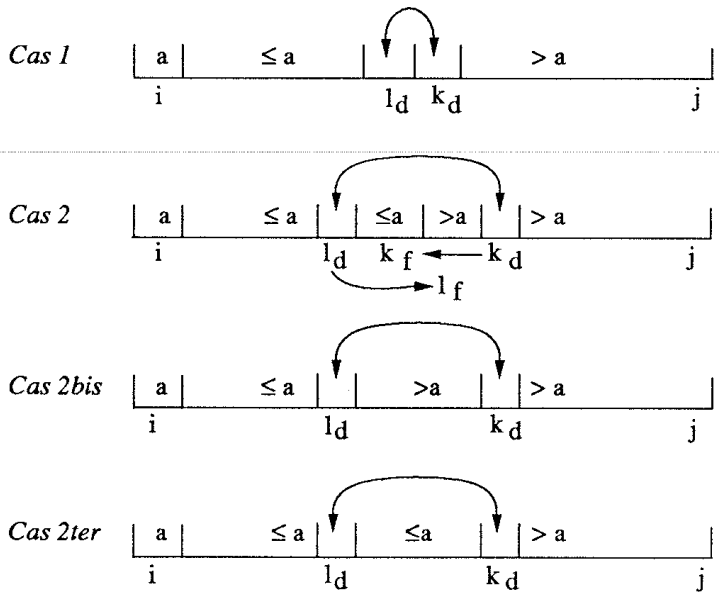


Figure 4. Cas possibles après le dernier échange de la partition.

cela signifie que cette sous-liste est déjà partitionnée : les éléments du début sont inférieurs ou égaux à a, les éléments de la fin sont supérieurs à a (voir figure 4). Après l'échange de $t[k_d]$ et $t[l_d]$, k décroît de 1 et l augmente de 1, et on exécute de nouveau la boucle **while** puisque $l \leq k$. La variable k décroît jusqu'à une valeur k_f, qui est l'indice du dernier élément inférieur ou égal à a; et l croît jusqu'à l_f, indice du premier élément supérieur à a. On a $l_f = k_f + 1$, ce qui provoque la sortie de la boucle **while**, et on échange $t[k_f]$, qui est bien inférieur ou égal à a, et $t[i]$.

Analyse de la partition et du placement

Nombre de comparaisons

Quand il y a n éléments dans la liste, la procédure *placer* compare le pivot avec les $n-1$ autres éléments. Mais certaines comparaisons peuvent être faites deux fois puisque l et k se croisent : si la procédure se termine comme dans le deuxième cas de la figure 4, $t[k]$ et $t[l]$, où k et l ont leurs valeurs finales k_f et l_f, sont comparés deux fois à $t[i]$: on a donc dans ce cas $n-1+2 = n+1$ comparaisons. On n'a cependant que $n-1$ comparaisons dans le premier cas de la figure 4 : on sort alors de la boucle la plus externe sans faire de comparaisons supplémentaires.

Nombre d'échanges

On a au pire $\lceil n/2 \rceil$ échanges (soit $3\lceil n/2 \rceil$ transferts) : c'est le cas où le pivot vient se placer au milieu du tableau et où tous les éléments qui lui sont supérieurs (resp. inférieurs ou égaux) sont en début (resp. en fin) de liste.

1.3. Analyse du tri rapide

Le graphe des appels de la procédure *tri-rapide*(t, i, j) est un arbre binaire (cf. chapitre 7). Par exemple, au tri de la figure 2 correspond l'arbre de la figure 5.

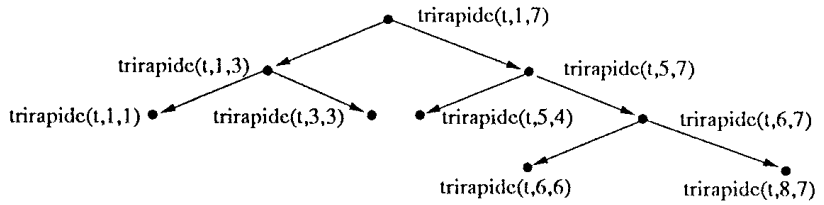


Figure 5. Arbre des appels pour le tri de la figure 2.

A chaque niveau de l'arbre correspondent des appels de la procédure *placer* qui provoquent le parcours de sous-listes dont la réunion est de taille inférieure ou égale à la taille n de la liste d'origine moins le nombre de pivots déjà placés : si on a deux sous-listes, elles contiennent en tout, au maximum, $n-1$ éléments; si on a 4 listes, elles contiennent en tout, au maximum, $n-3$ éléments; si on a p listes, elles contiennent en tout, au maximum, $n-(p-1)$ éléments. Le nombre d'éléments peut être inférieur à $n-p+1$ pour p sous-listes si on a placé des pivots aux extrémités de certaines sous-listes.

Comme on fait au pire, par sous-liste, une comparaison de plus que son nombre d'éléments, le nombre de comparaisons effectuées à chaque niveau de l'arbre des appels est donc majoré par $n+1$. Intuitivement, on voit que si l'arbre est équilibré, donc de hauteur $\Theta(\log_2 n)$, on a $\Theta(n\log_2 n)$ comparaisons; si l'arbre est déséquilibré, donc de hauteur $\Theta(n)$, on a $\Theta(n^2)$ comparaisons. C'est ce qui est démontré ci-dessous.

1.3.1. Complexité au pire en nombre de comparaisons

On a un maximum de comparaisons dans le cas où l'arbre est un "peigne" (cf. chapitre 7) car à chaque niveau, on ne place qu'un pivot et on a un nombre de niveaux maximum. Cela se produit, par exemple, si le pivot est toujours le plus petit élément, ce qui est le cas si la liste est strictement croissante et que l'on prend le premier élément comme pivot. On a alors les appels successifs :

tri-rapide$(t, 1, n)$
tri-rapide$(t, 1, 0)$; *tri-rapide*$(t, 2, n)$
tri-rapide$(t, 2, 1)$; *tri-rapide*$(t, 3, n)$
...
tri-rapide$(t, n-1, n-2)$; *tri-rapide*(t, n, n)

où les appels de *tri-rapide*$(t, i, i-1)$ ne font aucune comparaison (et aucun transfert).

Au premier niveau de l'arbre des appels, on fait $n+1$ comparaisons au pire, au deuxième n comparaisons au pire, et ainsi de suite jusqu'au niveau $n-1$ où on fait 3 comparaisons au pire. On a donc :

$$\mathrm{Max}_C(n) = \sum_{i=1}^{n-1}(n+2-i) = \sum_{j=2}^{n}(j+1) = \frac{(n+2).(n+1)}{2} - 3 = \Theta(n^2)$$

Le tri rapide a donc une complexité au pire, en nombre de comparaisons, en $\Theta(n^2)$.

Pour ce qui est des transferts, on a $n-1$ échanges soit $3(n-1)$ transferts. Mais ce cas ne correspond pas au nombre maximum de transferts : le calcul de la complexité au pire du tri rapide en nombre de transferts est laissé en exercice.

1.3.2. Complexité moyenne en nombre de comparaisons

On suppose que les n clés sont distinctes, et que l'élément pivot vient occuper la $p^{ième}$ place de la liste à trier. On fait l'hypothèse que toutes les places ont la même probabilité : on a donc une probabilité $1/n$ d'avoir le pivot à la $p^{ième}$ place, donc d'avoir deux sous-listes de tailles $p-1$ et $n-p$.

On a vu que l'exécution de *placer* sur n éléments demande $n+1$ ou $n-1$ comparaisons. On a donc, pour le nombre moyen de comparaisons $\mathrm{Moy}(n)$ effectué par la procédure *tri-rapide* sur un tableau de n éléments, l'encadrement suivant :

$$n-1+\frac{1}{n}.\sum_{p=1}^{n}(\mathrm{Moy}_C(p-1)+\mathrm{Moy}_C(n-p)) \leq \mathrm{Moy}_C(n)$$
$$\leq n+1+\frac{1}{n}.\sum_{p=1}^{n}(\mathrm{Moy}_C(p-1)+\mathrm{Moy}_C(n-p))$$

qui est valable pour $n \geq 2$. De plus, $\mathrm{Moy}_C(0) = \mathrm{Moy}_C(1) = 0$.

On a donc pour $n \geq 2$, $A(n) \leq \mathrm{Moy}_C(n) \leq B(n)$ avec $A(1) = B(1) = 0$, $A(0) = B(0) = 0$, et :

$$A(n) = n-1+\frac{1}{n}.\sum_{p=1}^{n}(A(p-1)+A(n-p))$$

$$B(n) = n+1+\frac{1}{n}.\sum_{p=1}^{n}(B(p-1)+B(n-p))$$

On remarque que :

$$\sum_{p=1}^{n}B(p-1) = \sum_{p-1}^{n}B(n-p) = \sum_{p=1}^{n-1}B(p), \text{ car } B(0)=0.$$

Il en est de même pour A. On a donc, pour $n \geq 2$, les équations :

$$B(n) = n + 1 + \frac{2}{n}.\sum_{p=1}^{n-1} B(p)$$

$$A(n) = n - 1 + \frac{2}{n}.\sum_{p=1}^{n-1} A(p)$$

On montre en annexe que ces équations peuvent se résoudre soit en utilisant des séries génératrices, soit par soustraction. On obtient :

$$B(n) = 2(n+1)\left(H_{n+1} - \frac{4}{3}\right) = 2(n+1)H_n - \frac{8}{3}n - \frac{2}{3}$$

et $$A(n) = 2(n+1)H_n - 4n$$

Il en résulte que :

$$2(n+1)H_n - 4n \leq \text{Moy}_C(n) \leq 2(n+1)H_n - \frac{8}{3}n - \frac{2}{3}$$

or $H_n \sim \text{Log}\, n$, et donc $\text{Moy}_C(n) \sim 2n\text{Log}\, n \sim 1,38n.\log_2 n$.

Le nombre de comparaisons pour trier n éléments distincts par le tri rapide est en moyenne, lorsque les $n!$ permutations de la liste de départ sont équiprobables, en $\Theta(n\log_2 n)$.

1.3.3. Coût en place de la récursivité

Le tri rapide est un exemple de programme récursif pour lequel le programme itératif équivalent nécessite l'introduction d'une pile. En fait, cette pile est présente quand on exécute l'algorithme récursif, et sa taille doit être prise en compte dans l'évaluation de la *complexité en place* de l'algorithme.

Le tri rapide correspond au schéma de procédure récursive suivant :

```
procedure P(X);
var L;
begin
    if C then begin
                D;
                P(α);
                P(β)
    end {X et L interviennent dans C, D, α et β}
end P;
```

On peut supprimer le deuxième appel de P, qui est terminal, selon le schéma de transformation utilisé pour la procédure *dicho* au chapitre 9. On obtient :

```
procedure P(X);
var L;
begin
    while C do begin
        D; P(α);
        X := β
    end
end P;
```

La suppression du premier appel est moins évidente car cet appel n'est pas terminal et la valeur des arguments ne varie pas de manière prévisible comme pour le tri par insertion séquentielle (cf. chapitre 14).

$P(\alpha)$ va être transformé en une séquence d'instructions du type : "donner à X la valeur α puis aller au début de la procédure". Pour pouvoir exécuter ensuite l'instruction $X := \beta$, il faut d'une part sauvegarder X et L dans une pile car on va avoir un arbre des appels comme dans la figure 5, et on est en train de programmer, en quelque sorte, le parcours en profondeur à main gauche de cet arbre. D'autre part, il faut prévoir un retour vers cette instruction, ou plutôt vers le début de la procédure avec les bonnes valeurs de X et de L, au moyen d'une boucle supplémentaire. Un résultat possible de cette transformation est donné ci-dessous :

```
var L; Q : PILE;
begin X := X0;
    pile-vide(Q);
    while true do begin
        while C do begin
            D ; empiler(Q, (X, L)) ; X := α
        end;
        if not est-vide(Q) then begin
            (X, L) := sommet(Q) ; dépiler(Q) ; X := β
        end
        else exit
    end
end;
```

Cette transformation donne la procédure itérative *tri-rapide-iter*, qui est équivalente à la procédure *tri-rapide* appelée avec les données $(t, 1, n)$. Le couple (i, j) correspond au X du schéma ci-dessus et k correspond à L.

Ce qui nous intéresse ici, c'est la taille à prévoir pour la pile. Cette taille est maximum quand l'arbre binaire des appels de P est de profondeur n, avec un enchaînement de n appels non terminaux : cela correspond au cas où le pivot se place chaque fois à la fin de la sous-liste.

```
procedure tri-rapide-iter(var t : array [1..n + 1] of integer);
var Q : PILE; i, j, k : integer;
begin
    i := 1; j := n; pile-vide(Q);
    while true do begin
        while i < j do begin
            placer(t, i, j, k); empiler(Q, (i, j, k)); j := k − 1
        end;
        if not est-vide(Q) then begin
            (i, j, k) := sommet(Q); dépiler(Q); i := k + 1
        end
        else exit
    end
end tri-rapide-iter;
```

On va devoir empiler n valeurs successives pour k. Ce résultat n'est pas très bon, puisqu'il établit qu'il faut prévoir n variables auxiliaires, ou plus précisément une pile de n entiers, pour trier n éléments. On peut cependant l'améliorer notablement en remarquant que seul le premier appel provoque des empilements. On a donc intérêt, pour minimiser la taille de la pile, à appeler en premier le tri rapide pour la plus petite partition.

Cela donne la version suivante de la procédure *tri-rapide* :

```
procedure tri-rapide-opt(var t : array [1..n + 1] of integer; i, j : integer);
var k : integer;
begin
    while i < j do begin
        placer(t, i, j, k);
        if j − k > k − i then begin tri-rapide-opt(t, i, k − 1);
                                    i := k + 1 end
        else begin tri-rapide-opt(t, k + 1, j); j := k − 1 end
    end
end tri-rapide-opt;
```

Dans cette version du tri rapide, la taille de la sous-liste en argument de l'appel récursif est toujours au moins divisée par deux. On a donc une profondeur des appels majorée par $\log_2 n$, ce qui majore aussi la taille de la pile auxiliaire.

1.4. Conclusion sur le tri rapide

L'algorithme présenté ici est un cas particulier du tri rapide : le choix du pivot peut être différent, et ce choix est fondamental pour améliorer le tri rapide. En *choisissant le pivot* de manière à ce qu'il ait plus de chance de se placer au milieu du tableau, on évite les cas limites pour lesquels la complexité au pire du nombre

de comparaisons est en $\Theta(n^2)$. On peut, par exemple, chercher les trois premiers éléments différents de la liste et prendre pour pivot l'élément du milieu. Dans tous les cas, le choix du pivot doit rester simple, de complexité constante. Plusieurs façons de choisir le pivot sont proposées en exercice.

On peut aussi remarquer qu'en dessous d'une certaine taille des sous-listes, le coût en temps et en place des appels récursifs devient significatif par rapport au nombre de comparaisons. En dessous d'un certain *seuil*, il devient plus avantageux d'utiliser la version itérative d'une méthode de tri simple, la sélection ordinaire par exemple. Expérimentalement on s'est aperçu que la valeur de ce seuil est de l'ordre de 10.

Le tri rapide est une méthode relativement simple à mettre en œuvre et dont la complexité, en nombre de comparaisons est en $\Theta(n\log n)$ en moyenne et en $\Theta(n^2)$ au pire. De plus, elle nécessite une pile auxiliaire de taille maximum $\log_2 n$.

On va voir dans la suite de ce chapitre qu'il existe des algorithmes de tri qui sont en $\Theta(n\log n)$ en moyenne et au pire en nombre de comparaisons, et qui ne nécessitent pas de mémoire auxiliaire.

1.5. Remarque : le tri fusion

Le tri rapide est, d'une certaine manière, une généralisation des méthodes de sélection : on transfère un élément à sa place définitive, et comme cette place n'est pas toujours la première (ce qui est le cas pour les méthodes de sélection), on recommence l'algorithme sur les deux sous-listes obtenues.

On peut avoir une démarche inverse : on partage la liste en deux sous-listes que l'on trie, et on interclasse ces deux sous-listes. On a alors un algorithme dont la structure est l'inverse du tri rapide : on a deux appels récursifs puis un traitement. Un tel tri s'appelle un ***tri fusion***. Une *fusion* ou un *interclassement* consiste à construire, à partir de deux listes triées, une liste contenant la réunion des éléments des deux listes d'origine, triée, avec conservation des répétitions. Ce problème a été vu à l'exercice 4 du chapitre 9; il est aussi étudié au paragraphe 3 du chapitre 17. Si les deux listes d'origine contiennent respectivement n et m éléments on a $n+m$ transferts et $n+m-1$ comparaisons au pire; il faut de plus $n+m$ places pour le tableau résultat.

L'inconvénient du tri fusion est qu'il nécessite une liste auxiliaire qui sert à ranger les résultats de l'interclassement. Les interclassements se font alternativement de la liste d'origine vers la liste auxiliaire ou en sens inverse. Une version récursive du tri fusion est donnée ci-dessous.

Le lecteur déjà familier avec ces méthodes remarquera que cet algorithme est légèrement différent des tris fusion donnés habituellement (Von Neumann, par exemple), à cause de sa formulation récursive.

```
procedure tri-fusion(var t : array [1..2, 1..n] of integer; i, j : integer;
                                départ : 1..2; var arrivée : 1..2);
{cette procédure trie les éléments t[départ, i] ,..., t[départ, j] et range
le résultat du tri dans les places t[arrivée, i] ,..., t[arrivée, j]}
var k, m : integer; a1, a2 : 1..2;
begin
    if i < j then begin
            k := (i + j) div 2;
            tri-fusion(t, i, k, départ, a1);
            tri-fusion(t, k + 1, j, départ, a2);
            if a1 <> a2 then {recopie de la deuxième sous-liste}
                    for m := k + 1 to j do t[a1, m] := t[a2, m];
            interclassement(t, i, j, k, a1);
            if a1 = 1 then arrivée := 2 else arrivée := 1
    end
    else if i = j then arrivée := départ
end tri-fusion;
```

{*note : l'appel de procédure interclassement*$(t, i, j, k, a1)$ *fusionne les éléments* $t[a1, i], ..., t[a1, k]$ *avec les éléments* $t[a1, k+1], ..., t[a1, j]$. *Le résultat se trouve dans* $t[2, i], ..., t[2, j]$ *si* $a1 = 1$ *et dans* $t[1, i], ..., t[1, j]$ *si* $a1 = 2$}

La procédure interclassement n'est pas donnée : on trouvera tous les éléments pour la programmer à l'exercice 4 du chapitre 9. La liste complète est triée par l'appel de *tri-fusion*$(t, 1, n, 1,$ *arrivée*$)$. Au retour de cet appel, la variable *arrivée* contient 1 ou 2 selon que le résultat du tri se trouve en $t[1, 1], ..., t[1, n]$ ou en $t[2, 1], ..., t[2, n]$.

La complexité au pire en nombre de comparaisons et de transferts de cet algorithme est meilleure que celle du tri rapide. En effet, on a pour le nombre de comparaisons :

$$\mathrm{Max}_C(1) = 0 \qquad \mathrm{Max}_C(n) = (n-1) + \mathrm{Max}_C(\lfloor n/2 \rfloor) + \mathrm{Max}_C(\lceil n/2 \rceil),$$

où $(n-1)$ est le nombre de comparaisons pour interclasser deux tableaux comportant en tout n éléments. Cette équation a pour solution (voir exercices) :

$$\mathrm{Max}_C(n) = n\lceil \log_2 n \rceil - 2^{\lceil \log_2 n \rceil} + 1$$

Le tri fusion a donc une complexité au pire en nombre de comparaisons en $\Theta(n \log n)$. La complexité au pire en nombre de transferts est laissée en exercice. Elle est aussi en $\Theta(n \log n)$.

Cependant, *cet algorithme nécessite n places auxiliaires* pour ranger les résultats des interclassements successifs, sans compter la pile due à la récursivité, qui peut d'ailleurs être évitée. Cela lui ôte tout intérêt pratique pour le tri interne. On reviendra à des algorithmes de ce type pour le tri externe, au chapitre 17.

2. Le tri par tas

Le ***tri par tas*** (en anglais, *heapsort*) est une méthode par sélection où, comme suggéré au chapitre 14, on garde, d'un parcours de la liste à l'autre, le résultat des comparaisons effectuées pour sélectionner les minimums successifs.

On utilise pour stocker ces résultats une structure de données appelée *tas* (*heap* en anglais) qui est présentée dans le paragraphe suivant. On verra ensuite que l'utilisation d'un tas permet de trier n éléments en $\Theta(n \log n)$ comparaisons au pire, ceci sans mémoire auxiliaire car on peut utiliser le tableau à trier pour représenter le tas.

2.1. Arbres partiellement ordonnés et tas

Le problème que l'on veut résoudre pour optimiser le tri par sélection est d'organiser les éléments de la liste non encore placés de manière à accéder à un minimum, et à supprimer ce minimum, efficacement. Ces deux opérations sont les seules utilisées lors d'un tri par sélection.

Le fait de restreindre l'accès et la suppression à un élément minimum de la liste permet de donner une représentation efficace des listes par des ***arbres parfaits partiellement ordonnés***. On remarque dans ce cas que l'ordre des éléments dans la liste n'a plus de signification : plutôt que la liste, on représente l'ensemble, avec répétitions, des éléments de la liste.

2.1.1. Arbres parfaits partiellement ordonnés

Définition : Un ***arbre partiellement ordonné*** est un arbre, étiqueté par des éléments d'un ensemble muni d'un ordre total, et tel que l'élément contenu dans tout nœud est inférieur ou égal aux éléments contenus dans les fils de ce nœud.

Par ailleurs, on a vu au chapitre 7 qu'un arbre parfait est un arbre binaire dont toutes les feuilles sont situées sur deux niveaux seulement, avec l'avant-dernier niveau qui est complet et les feuilles du dernier niveau qui sont regroupées le plus à gauche possible.

On peut utiliser des arbres parfaits partiellement ordonnés. Un exemple d'arbre parfait partiellement ordonné est donné à la figure 6. Si on représente une liste ou un ensemble, ou un multi-ensemble par un tel arbre, on a toujours un élément minimum à la racine : l'opération d'*accès au minimum* a donc une représentation très efficace puisqu'il suffit d'accéder à la racine de l'arbre.

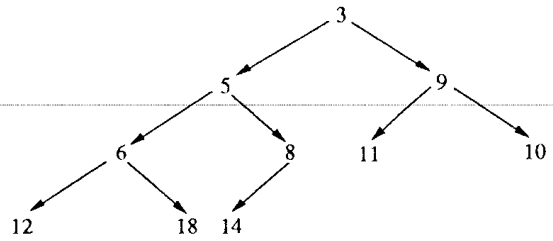


Figure 6. Arbre parfait partiellement ordonné.

L'*adjonction* d'un nouvel élément implique l'ajout d'une feuille au dernier niveau de l'arbre (si le dernier niveau est complet, il faut en commencer un autre). Dans un premier temps, on met le nouvel élément à cette feuille. De ce fait, l'arbre est toujours parfait mais peut ne plus être partiellement ordonné. Pour rétablir cette propriété, on échange le contenu de la nouvelle feuille avec le contenu de son père, puis le contenu de celui-ci avec le contenu de son père, et ainsi de suite, tant que la propriété $\mathit{contenu}(\mathit{père}(x)) \leq \mathit{contenu}(x)$ n'est pas satisfaite (voir figure 7).

Pour la *suppression du minimum* on utilise une méthode similaire : la dernière feuille du dernier niveau doit disparaître, puisqu'on a un nœud de moins. On prend donc l'élément qui est contenu dans cette feuille et on le place à la racine, à la place de l'élément à supprimer. On effectue ensuite des échanges de contenu avec le fils dont le contenu est minimum, en partant de la racine, et en descendant vers le fils avec lequel on a fait un échange, ceci tant que ce fils n'a pas un contenu inférieur à ceux de ses deux fils (ou de son seul fils) ou tant qu'il n'est pas à une feuille (voir figure 8).

Les opérations d'adjonction et de suppression peuvent nécessiter le parcours d'une branche complète de l'arbre, avec à chaque nœud des comparaisons et des échanges. La complexité au pire, en nombre de comparaisons ou d'échanges, de ces opérations, dépend de la hauteur de l'arbre.

On a vu au chapitre 7 qu'un arbre binaire parfait comportant p nœuds est de hauteur $\lfloor \log_2 p \rfloor$. Ce résultat permet d'établir que les complexités au pire, en nombre de comparaisons et d'échanges, des opérations d'adjonction et de suppression du minimum sont en $\Theta(\log_2 p)$. On montre également que la suppression d'un nœud quelconque nécessite au pire $\Theta(\log_2 p)$ comparaisons (cf. exercices).

2.1.2. Représentation par un tas

On a vu au chapitre 7 qu'un arbre binaire parfait se représente facilement par un tableau. Rappelons que si t est ce tableau, on a les règles suivantes :

- $t[1]$ est la racine ;
- $t[i \textbf{ div } 2]$ est le père de $t[i]$ pour $i > 1$;
- $t[2 * i]$ et $t[2 * i + 1]$ sont les deux fils, s'ils existent, de $t[i]$;

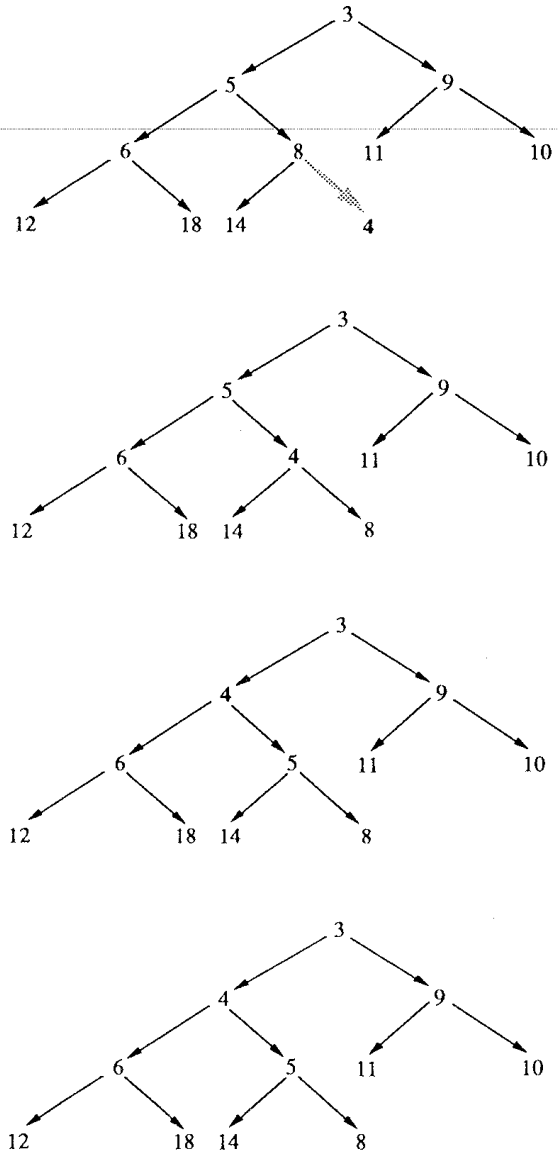


Figure 7. Adjonction de 4 à l'arbre de la figure 6.

- si p est le nombre de nœuds de l'arbre et si $2 * i = p$, $t[i]$ n'a qu'un fils, $t[p]$;
- si i est supérieur à p **div** 2, $t[i]$ est une feuille.

A titre d'exemple, l'arbre de la figure 6 est représenté par le tableau :

3 5 9 6 8 11 10 12 18 14

On appelle ***tas*** un tableau représentant un arbre parfait partiellement ordonné.

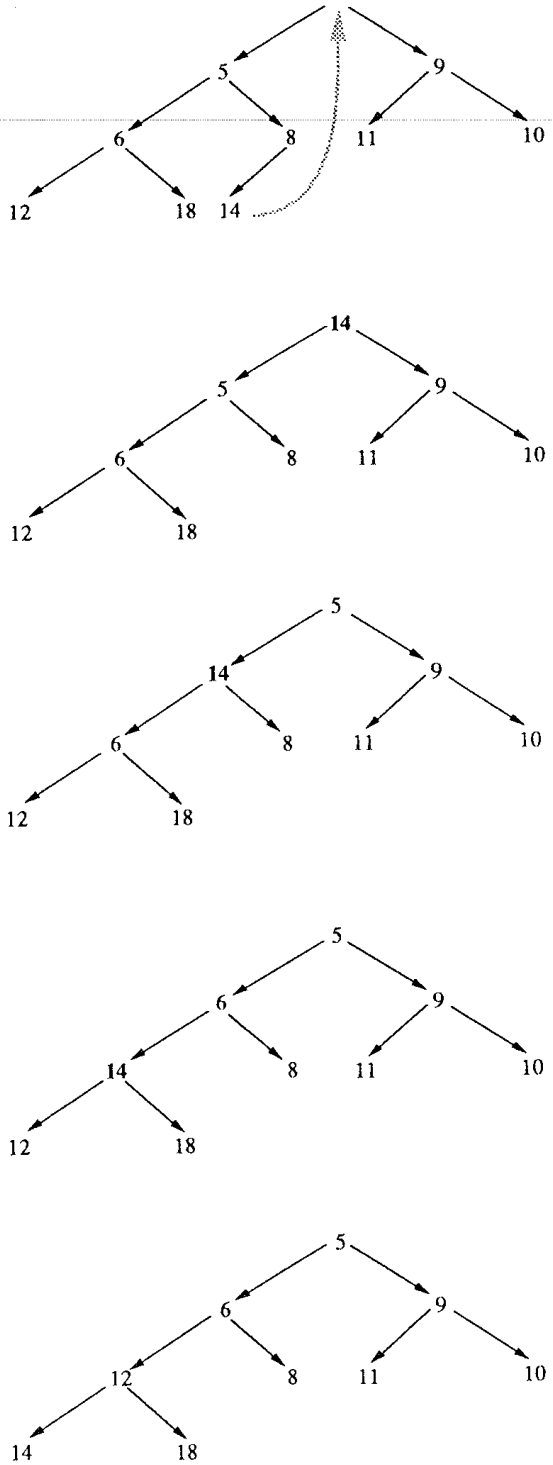


Figure 8. Suppression du minimum de l'arbre de la figure 6.

Soit n la taille du tableau t qui contient un tas de p éléments ($p \leq n$). Lorsque les éléments sont des entiers, on a la procédure d'adjonction suivante :

```
procedure ajouter(var t : array [1..n] of integer; var p : integer; x : integer);
{cette procédure ajoute l'élément x dans le tas t qui contient p éléments au
moment de l'appel; p est augmenté de 1 par la procédure}
{on suppose que p < n au moment de l'appel}
var i : integer;
begin
    {x est placé à la dernière feuille :}
    p := p + 1; t[p] := x; i := p;
    {on effectue des échanges tant que x est inférieur à son père :}
    while (i > 1) and ( t[i] < t[i div 2]) do begin
        t[i] ↔ t[i div 2]; i := i div 2
    end
end ajouter;
```

Dans le tri par sélection, la recherche du minimum est toujours suivie de sa suppression. C'est pourquoi la procédure *détruire* donnée ci-dessous effectue successivement ces deux opérations.

```
procedure détruire (var t : array [1..n] of integer; var p, min : integer);
{Cette procédure rend le minimum des p éléments contenus dans le tableau t.
Ce minimum est retourné par la variable min. La procédure réorganise le tas
après suppression de ce minimum et diminue p de 1}
{On suppose que le tas n'est pas vide au moment de l'appel : p > 0}
var i, j : integer;
begin
    {on retourne le minimum :}
    min := t[1];
    {réorganisation du tas :}
    t[1] := t[p]; p := p − 1; i := 1;
    while i ≤ (p div2) do begin {t[i] n'est pas une feuille}
    {calcul de l'indice du plus petit des deux fils de t[i] ou de son seul fils :}
    if (2 * i = p) or (t[2 * i] < t[2 * i + 1]) then j := 2 * i else j := 2 * i + 1;
    {échange si la condition d'ordre n'est pas satisfaite :}
    if t[i] > t[j] then begin t[i] ↔ t[j]; i := j end else exit
    end
end détruire;
```

2.2. Utilisation d'un tas pour le tri d'une liste

Rappelons le principe des méthodes de tri par sélection : on sélectionne successivement le minimum des n éléments de la liste, puis des $n - 1$ éléments restants, ceci jusqu'à épuisement de la liste, et à chaque fois on place l'élément sélectionné

à sa place définitive. Le tri par tas utilise, comme son nom l'indique, un tas pour effectuer la recherche du minimum de manière efficace.

2.2.1. Premières versions du tri par tas

Dans cette première version, on ignore volontairement certains problèmes d'implémentation afin de mieux faire comprendre le principe de l'algorithme. Le tri par tas comporte les étapes suivantes :

(1) construire un tas contenant les n éléments de la liste à trier par des adjonctions successives. Ceci demande, on vient de le voir, $\Theta(n \log n)$ comparaisons au pire; initialiser la future liste triée à *liste-vide*;

(2) rechercher le minimum dans ce tas et le mettre à la fin de la future liste triée, ce qui ne demande aucune comparaison;

(3) supprimer le minimum du tas et réorganiser celui-ci. Cela coûte $\Theta(\log n)$ comparaisons au pire;

(4) recommencer à l'étape 2, jusqu'à ce que le tas soit vide, c'est-à-dire, n fois.

On peut utiliser le tableau à trier pour stocker le tas. Cependant, on a vu au chapitre 7 que la gestion des indices, quand on représente un arbre parfait, est relativement simple si on met la racine à l'indice 1. On va donc utiliser le début du tableau pour stocker le tas, comme c'est indiqué à la figure 9. Cela a pour conséquence que l'on range les minimums successifs à la fin du tableau, *par ordre décroissant*. C'est cette version qui est donnée ici.

Cette deuxième version comporte les étapes suivantes :

(1) initialisation : ajouter successivement chacun des n éléments dans le tas $t[1..p]$; p augmente de 1 après chaque adjonction. A la fin on a un tas de taille $p = n$.

(2) tant que p est plus grand que 1, supprimer le minimum du tas (p décroît de 1); réorganiser le tas; ranger le minimum obtenu à la $(p+1)^{ième}$ place (voir figure 9b).

Remarque : Pour trier un tableau par ordre croissant en utilisant un tas, il faut modifier l'ordre pris en compte dans le tas ($\geq$ au lieu de $\leq$) de manière à obtenir un maximum au premier indice du tas.

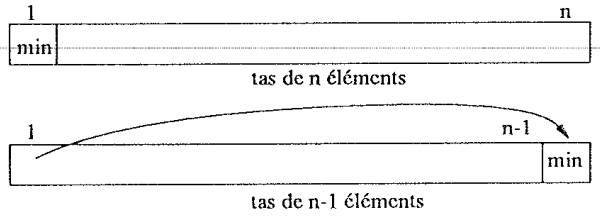


Figure 9a. Première sélection dans le tri par tas.

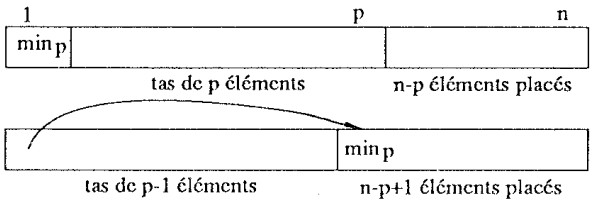


Figure 9b. Configurations successives du tableau à trier dans un tri par tas.

En utilisant les opérations d'adjonction et de suppression dans un tas, données au paragraphe 2.1, on obtient une première procédure pour le tri par tas :

```
procedure tri-par-tas1(var t : array[1..n] of integer);
var p, min : integer;
begin
    p := 0;
    while p < n do {construction du tas}
        ajouter(t, p, t[p + 1]);
        {p augmente de 1 à chaque appel de ajouter}
    while p > 1 do begin
        détruire(t, p, min);
        {p diminue de 1 à chaque appel de détruire}
        t[p + 1] := min
    end
end tri-par-tas1;
```

Cette version de l'algorithme construit le tas par n appels de la procédure *ajouter* et effectue les sélections par $n-1$ appels de la procédure *détruire*. Cela coûte de l'ordre de $\sum_{i=1}^{n} \log_2 i$ comparaisons, au pire.

On a donc un algorithme de tri sur place en $\Theta(n \log n)$ comparaisons au pire.

Le nombre total de transferts est égal au nombre d'échanges faits lors des adjonctions et des suppressions du tas, multiplié par trois, plus les $n-1$ transferts qui correspondent au placement de l'élément sélectionné. Or le nombre d'échanges dans le tas est majoré par le nombre de comparaisons et il est du même ordre de grandeur. La complexité au pire en nombre de transferts du tri par tas est donc en $\Theta(n \log n)$.

Il est possible d'améliorer la construction du tas. En effet, les n éléments étant présents dès le début de la construction, on peut utiliser au mieux les relations qui existent entre eux.

2.2.2. Tri par tas avec construction du tas en temps linéaire

Dans un tas de taille n, les éléments situés aux indices compris entre $\lfloor n/2 \rfloor + 1$ et n sont des feuilles de l'arbre parfait représenté. La construction du tas peut se faire en examinant les noeuds internes en partant du niveau le plus bas, et en vérifiant qu'ils satisfont la condition d'ordre partiel avec leurs fils. Si ce n'est pas le cas, on fait les échanges nécessaires. Cela revient à ordonner successivement des sous-arbres dont les racines sont aux indices $\lfloor n/2 \rfloor$, $\lfloor n/2 \rfloor - 1, \ldots, 1$, comme c'est montré à la figure 10.

Pour ordonner un sous-arbre, on peut être amené à descendre jusqu'à une feuille pour faire des permutations. On a la procédure suivante qui traite le sous-arbre de racine $t[i]$:

```
procedure ordonner(var t : array [1..n] of integer; i : integer);
{on suppose que les sous-arbres fils de t[i] ont déjà été traités}
var j : integer;
begin
    {recherche de l'indice du plus petit fils de t[i] :}
    if (2*i+1 > n) or (t[2*i] ≤ t[2*i+1]) then j := 2*i
                                          else j := 2*i+1;
    {échange, si nécessaire, de t[i] avec ce fils :}
    if t[j] < t[i] then begin
                        t[j] ↔ t[i];
                        {si t[j] n'est pas une feuille, il faut réordonner le sous-arbre
                        de racine t[j]}
                        if j ≤ n div 2 then ordonner(t, j)
    end
end ordonner;
```

Cette procédure étant récursive terminale, il est immédiat d'en donner une version itérative. La transformation est laissée en exercice.

Tableau d'origine: 15 12 11 7 13 9 ; $n = 6$

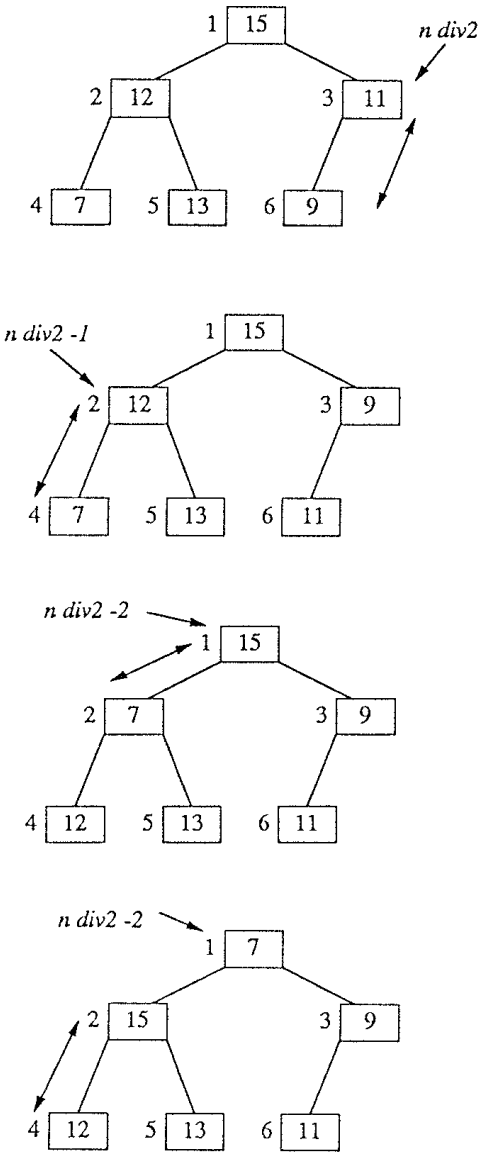


Figure 10. Réorganisation d'un tableau en arbre partiellement ordonné.

On obtient la version finale de la procédure *tri-par-tas* :

```
procedure tri-par-tas2(var t : array [1..n] of integer);
var p, min : integer;
begin
    for p := n div 2 downto 1 do ordonner(t, p);
    p := n;
    while p > 1 do begin
        détruire(t, p, min);
        t[p + 1] := min
    end
end tri-par-tas2;
```

On montre ci-dessous que la première boucle de cet algorithme effectue un nombre de comparaisons en $\Theta(n)$. Cependant, cette amélioration de la construction du tas ne change pas l'ordre de la complexité en nombre de comparaisons du tri par tas : la deuxième boucle effectue en effet $\Theta(n \log n)$ comparaisons au pire.

On a donc un algorithme de tri qui effectue $\Theta(n \log n)$ comparaisons au pire. Sa complexité en moyenne en nombre de comparaisons est du même ordre : en effet, on va voir au chapitre suivant que la complexité en moyenne, en nombre de comparaisons, de ce type d'algorithme est d'ordre au moins égal à $n \log n$.

2.2.3. Complexité linéaire de la construction du tas

Cette construction se fait par $\lfloor n/2 \rfloor$ appels de *ordonner*(t, p), où p varie de manière décroissante de $\lfloor n/2 \rfloor$ à 1. Chacun de ces appels provoque deux comparaisons entre éléments, suivies éventuellement d'un échange, suivi éventuellement d'un appel récursif de *ordonner*(t, j) où j vaut $2p$ ou $2p + 1$.

Les nombres de comparaisons et d'échanges sont proportionnels au nombre total d'appels de la procédure. On va majorer ce nombre.

Quand p est compris entre $\lfloor n/4 \rfloor + 1$ et $\lfloor n/2 \rfloor$, on n'a pas d'appel récursif car la variable j est supérieure à $\lfloor n/2 \rfloor$: on a donc un appel en tout.

Quand p est compris entre $\lfloor n/8 \rfloor + 1$ et $\lfloor n/4 \rfloor$ on a au plus un appel récursif, soit deux appels en tout.

Plus généralement, quand p est compris entre $\lfloor n/2^{i+1} \rfloor + 1$ et $\lfloor n/2^i \rfloor$, on a en tout i appels au plus : en effet, le premier appel a lieu pour p, et pour les suivants, le paramètre j est multiplié par 2 à chaque fois, s'il augmente le plus lentement possible, et on arrête quand $j > \lfloor n/2 \rfloor$. On finit avec un intervalle réduit à $\lfloor n/2^k \rfloor = 1$ où k est tel que $2^k \leq n < 2^{k+1}$.

Considérons l'intervalle $\text{Int}_i = [\, \lfloor n/2^{i+1} \rfloor + 1,\ \lfloor n/2^i \rfloor \,]$. Pour chaque valeur appartenant à cet intervalle, on a vu qu'on a i appels de la procédure. Le nombre d'éléments

de cet intervalle peut être majoré (grossièrement) par :

$$\lfloor n/2^i \rfloor - (\lfloor n/2^{i+1} \rfloor + 1) + 1 = \lfloor n/2^i \rfloor - \lfloor n/2^{i+1} \rfloor \leq n/2^i$$

De ce fait, le nombre d'appels de la procédure peut être majoré par :

$$\sum_{i=1}^{k} i.\frac{n}{2^i} \leq \frac{n}{2}.\sum_{i\geq 1} \frac{i}{2^{i-1}}$$

Or la série $\sum_{i\geq 1} i.x^{i-1}$ est la série dérivée de $\sum_{i\geq 0} x^i = \frac{1}{1-x}$, soit $\frac{1}{(1-x)^2}$.

Cette fonction vaut 4 pour $x = 1/2$. Le nombre d'appels de la procédure pour construire le tas est donc majoré par $4.(n/2)$. La construction du tas est donc en $\Theta(n)$.

Exercices

1. Parmi les algorithmes de tri suivants, indiquer ceux qui sont stables ou qui peuvent facilement être rendus stables : sélection ordinaire, tri à bulles, insertion séquentielle et dichotomique, tri rapide et tri par tas.

2. Faire tourner les algorithmes de tri par sélection, de tri par insertion, de tri rapide et de tri par tas sur la suite d'éléments 4, 7, 10, 3, 15, 12, 1, 8, 2, 6.

3. Donner un exemple qui montre qu'une permutation réorganisée par l'exécution de la procédure *partition* du tri rapide peut avoir plus d'inversions après l'exécution qu'avant .

4. Trouver un tableau t de taille n, non strictement croissant, qui correspond à la complexité au pire en nombre de comparaisons du tri rapide (indication : à chaque exécution de la procédure *placer* sur un sous-tableau $t[i..j]$ le plus grand élément du sous-tableau est $t[i]$).

5. Dans le cas où le tableau est trié en ordre strictement décroissant, dessiner l'arbre des appels récursifs de la procédure *tri-rapide*.

6. Combien de comparaisons et d'échanges fait le tri rapide pour trier n éléments égaux ?

7. Calculer la complexité au pire du nombre de transferts du tri rapide. On rappelle que le nombre d'échanges effectués par la procédure *placer* est maximum quand le pivot vient se placer au milieu du tableau, et que tous les éléments qui lui sont supérieurs sont en début de liste.

8. On peut améliorer le tri rapide en choisissant judicieusement l'élément du tableau qui sert de pivot pour la procédure *placer*(t, i, j, k).

a) Pour ne rien avoir à faire lorsque les éléments du tableau sont égaux, on peut choisir le pivot de sorte qu'il y ait un élément qui lui soit supérieur ou égal et un élément qui lui soit inférieur (si c'est possible).

Ecrire une fonction *trouve-pivot* à valeur entière qui retourne 0 si tous les éléments $t[i], ..., t[j]$ sont identiques, et sinon renvoie l'indice du plus grand des deux éléments distincts situés le plus à gauche possible. Quelles modifications ce choix de pivot entraîne-t-il pour la procédure *placer*?

b) On peut aussi choisir comme pivot de façon systématique l'élément médian des trois éléments $t[i]$, $t[(i+j)$ **div** $2]$ et $t[j]$. Combien de comparaisons fait alors la procédure *tri-rapide* dans le pire des cas pour trier n éléments? (On comptera aussi les comparaisons effectuées pour trouver le pivot.)

9. Le problème suivant est connu sous le nom de problème du *drapeau tricolore*. On se donne un tableau de n éléments qui ont chacun une couleur, soit vert, soit blanc, soit rouge. On veut obtenir un tableau avec trois zones : à gauche tous les éléments verts, au milieu tous les éléments blancs, et à droite tous les éléments rouges.

a) Ecrire une procédure qui réalise cette organisation du tableau en ne testant qu'une seule fois la couleur de chaque élément.

b) Reprendre le problème avec quatre couleurs.

10. On dispose d'une liste S non triée de n entiers deux à deux distincts, et on cherche pour k donné, $1 \leq k \leq n$, le $k^{ième}$ élément de cette liste (c'est-à-dire, l'entier tel que $k-1$ autres entiers lui soient inférieurs). On propose l'algorithme récursif suivant :

– Si $|S| = 1$, l'entier cherché est l'élément de S.

– Sinon on choisit dans S un élément s et on pose $S_1 = \{x \in S \text{ tq } x < s\}$ et $S_2 = \{x \in S \text{ tq } x > s\}$

Si $|S_1| \geq k$, on applique à nouveau l'algorithme sur S_1.

Si $|S_1| = k-1$, l'élément cherché est s.

Si $|S_1| < k-1$, on applique à nouveau l'algorithme sur S_2.

On suppose que S est représentée par un tableau $t[1..n]$ et qu'on prend pour s l'élément du tableau le plus à gauche.

a) Ecrire une procédure *sélection-du-kième* qui implémente l'algorithme (on pourra utiliser la procédure *placer*).

b) Soit $CS(n)$ le nombre moyen de comparaisons que nécessite cet algorithme quel que soit k. Etablir l'inégalité :

$$CS(n) \leq n+1+\max_{1\leq k\leq n}\{\frac{1}{n}(\sum_{i=1}^{k-1} CS(n-i)+\sum_{i=k+1}^{n} CS(i-1))\}$$

En déduire que $CS(n) \leq 4n$.

11. On se propose de chercher parmi n éléments d'une liste le plus grand et le plus petit.

a) Trouver un algorithme simple de complexité en nombre de comparaisons inférieure ou égale à $2n-3$.

b) On utilise la méthode récursive suivante : on divise la liste en deux sous-listes, on calcule le minimum et le maximum de chacune et on compare entre eux les minima et les maxima obtenus.

Programmer cette méthode et calculer sa complexité en nombre de comparaisons entre éléments. Cet algorithme est-il meilleur que le précédent?

c) Montrer que le problème ne peut être résolu en moins de $\lceil 3n/2 \rceil - 2$ comparaisons.

12. Calculer la complexité au pire en nombre de transferts de la procédure *tri-fusion.*

13. On note $\mathrm{Max}_C(n)$ la complexité au pire en nombre de comparaisons de l'algorithme de tri fusion. On a, pour tout $n \geq 2$, la relation de récurrence :

$$\mathrm{Max}_C(n) = \mathrm{Max}_C(\lceil n/2 \rceil) + \mathrm{Max}_C(\lfloor n/2 \rfloor) + n - 1 \text{ et } \mathrm{Max}_C(1) = 0$$

Etablir que : $\mathrm{Max}_C(n) = n\lceil \log_2 n \rceil - 2^{\lceil \log_2 n \rceil} + 1$ par récurrence sur n.

14. Dérécursifier l'algorithme de tri fusion.

15. Vérifier que les procédures d'adjonction et de suppression données pour les tas au paragraphe 2.1 conservent bien la propriété d'arbre partiellement ordonné.

16. On se donne un tas de n éléments représenté par un tableau t de dimension n. On veut écrire une procédure qui supprime l'élément $t[i]$ du tas, de sorte que le tableau t modifié reste un tas.

a) Ecrire la procédure dans le cas où $t[i]$ est une feuille. Donner la complexité au pire en nombre de comparaisons de cette procédure.

b) Ecrire de manière récursive la procédure dans le cas général. Cette procédure fait appel à la procédure précédente. Donner la complexité au pire en nombre de comparaisons de l'algorithme de suppression d'un noeud quelconque.

17. a) On se donne un tas de 30 éléments distincts, rangés dans un tableau t. Le plus petit élément est dans $t[1]$.

Indiquer quelles positions peut occuper :

(i) le troisième plus petit élément (c'est-à-dire, l'élément qui a exactement deux autres éléments qui lui sont inférieurs) ?

(ii) le sixième plus grand élément (c'est-à-dire, l'élément qui a exactement cinq autres éléments qui lui sont supérieurs) ?

b) On se donne un tas de n éléments distincts, rangés dans un tableau t. Indiquer quelles positions peut occuper le $i^{ième}$ plus petit élément pour $i = 2$, $i = n - 2$, $i = 3$, $i = n - 3$.

18. On se donne une liste de n éléments distincts triés en ordre décroissant qu'on veut trier en ordre croissant en utilisant le tri par tas.

a) Combien de comparaisons fait-on lors de la construction du tas ? (Regarder le cas $n = 10$.)

b) Une liste triée en ordre décroissant correspond-elle au meilleur des cas, au pire des cas, ou à un cas intermédiaire, pour la construction du tas ?

19. On propose ici une autre méthode pour réaliser la suppression du minimum dans un tas contenant des éléments distincts. On supprime l'élément à la racine, créant ainsi un "trou" qu'on fait descendre jusqu'à une feuille.

A chaque étape de la descente, on échange le trou avec le plus petit de ses deux fils. A la fin de la descente, sauf si le trou se trouve être la feuille la plus à droite du dernier niveau, l'arbre correspondant au tas n'est plus parfait. Pour le rendre parfait, on comble le trou avec l'élément contenu dans la feuille la plus à droite du dernier niveau et on fait remonter cet élément à sa place dans le tas.

Les étapes de la suppression du minimum dans le tas : 1, 2, 6, 3, 4, 11, 12, 7, 9, 8, 5, sont données à la figure 11.

a) Ecrire la procédure correspondante de suppression du minimum dans un tas.

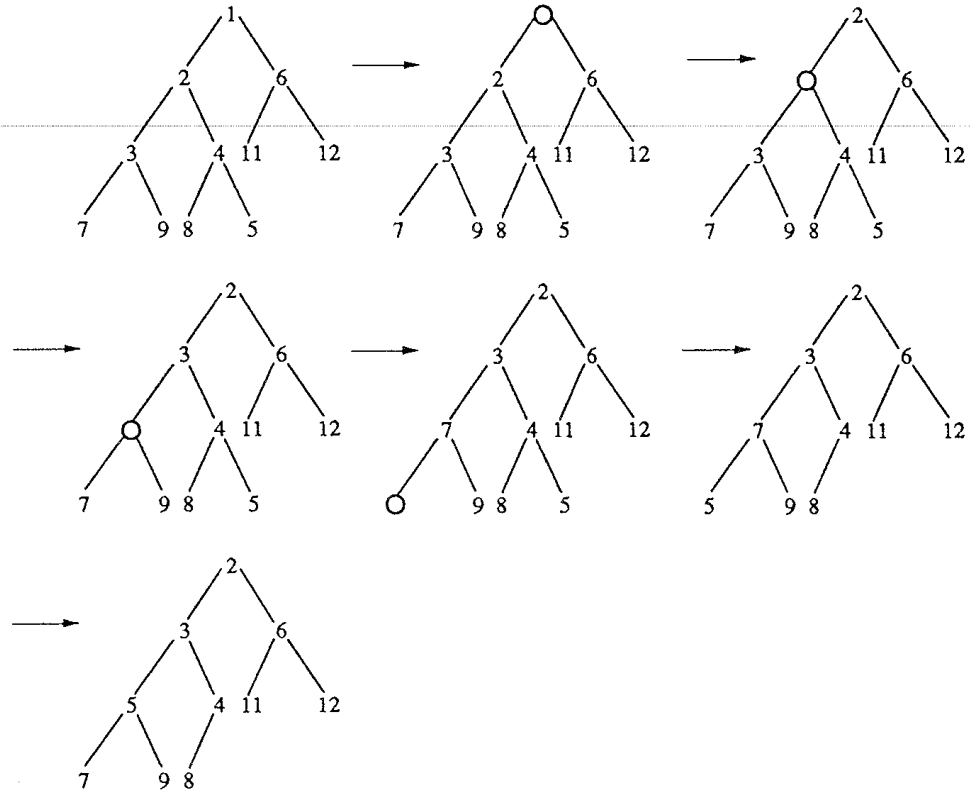


Figure 11. Variante de la suppression du minimum d'un tas.

b) Montrer que cette méthode donne le même tas que la méthode de suppression donnée au paragraphe 2.1.

c) Comparer les complexités de ces deux méthodes.

20. Soit un arbre ternaire partiellement ordonné dont tous les niveaux sont complètement remplis, sauf éventuellement le dernier et dans ce cas les feuilles manquantes sont à droite.

a) Montrer que l'on peut représenter un tel arbre par un tableau et donner le tableau représentant l'exemple. On appelle un tel tableau un ***tas ternaire***. Quels sont les indices respectifs dans le tableau, du père, des fils, des frères de l'élément situé à l'indice i ?

b) Généraliser aux tas ternaires l'algorithme de suppression du plus petit élément vu au paragraphe 2.1 pour les tas (binaires).

c) Analyser le nombre de comparaisons entre éléments faites par cet algorithme dans le cas le pire.

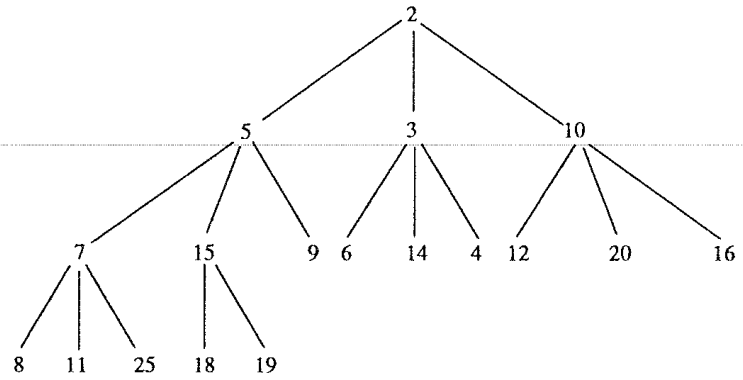


Figure 12. Exemple de tas ternaire.

Si l'on veut minimiser le nombre de comparaisons dans le cas le pire, vaut-il mieux utiliser des tas binaires ou des tas ternaires pour supprimer le plus petit élément ?

21. Pour quelles valeurs de k, $(1 \leq k \leq n)$, l'utilisation d'un tas peut-elle donner une méthode linéaire pour trouver le $k^{ième}$ plus petit élément d'une liste de n éléments ?

22. On propose une méthode de tri appelée ***tri tournoi***. On organise un tournoi des n éléments à trier (cf chapitre 3), tel que chaque nœud interne est égal au plus petit de ses deux fils.

On supprime ensuite l'élément à la racine, en le remplaçant partout dans l'arbre par un élément strictement supérieur à tous ceux de la liste à trier, et on reconstitue le tournoi, en partant des feuilles, et en mettant à chaque nœud interne le plus petit des deux fils. Ainsi n suppressions de ce type permettent de trier les n éléments.

a) Faire tourner l'algorithme sur la liste d'entiers de l'exercice n° 2.

b) Donner une procédure qui implémente l'algorithme.

c) Quelle est la complexité au pire en nombre de comparaisons de la procédure ? Sa complexité au pire en place ?

23. On dispose d'une pile de n factures de téléphone et d'une pile de p chèques destinés à payer les factures et qui portent chacun le numéro de téléphone correspondant. On veut savoir quelles factures n'ont pas été payées. On envisage trois algorithmes :

a) On prend les chèques dans l'ordre où ils se présentent et, pour chacun d'eux, on cherche la facture qui lui correspond; on enlève chèque et facture des listes données.

b) On trie les factures par ordre de numéros de téléphone croissants. Pour chaque chèque, on fait une recherche dichotomique de la facture correspondante.

c) On trie séparément factures et chèques, puis, pour i de 1 à p, on compare le $i^{ième}$ chèque avec les factures restantes. Quand on a trouvé la facture, on l'ôte ainsi que toutes celles de numéro inférieur.

Donner la complexité au pire de chaque algorithme en nombre de comparaisons. Comparer ces complexités en envisageant les grandeurs relatives de n et p.

24. On se donne une liste de n entiers distincts et un entier k ($k \leq n$). On veut obtenir les k plus petits entiers de la liste (non nécessairement ordonnés). Envisager plusieurs méthodes et comparer leurs complexités au pire et en moyenne en nombre de comparaisons (on pourra utiliser des procédures données dans les algorithmes de tri par tas et de tri rapide).

Chapitre 16
Optimalité des tris par comparaisons
Autres méthodes de tri

1. Complexité optimale pour les tris par comparaisons

Dans les chapitres précédents, différents algorithmes de tri ont été étudiés et comparés. La question qui se pose maintenant est de déterminer si l'on peut espérer trouver une méthode de tri «meilleure» que toutes les autres. Il faut évidemment préciser le(ou les) critère(s) selon le(s)quel(s) on compare les méthodes et pour quelles méthodes ce critère est significatif.

Pour pouvoir obtenir des résultats intéressants, on va se restreindre à la classe TC des algorithmes de tri qui opèrent uniquement par comparaisons deux à deux des clés des éléments à trier. De plus, on fait l'hypothèse que toutes les clés de la liste à trier sont différentes.

Dans ces conditions, on va montrer que :

La complexité, en nombre de comparaisons, aussi bien en moyenne qu'au pire, de tout algorithme de la classe TC est d'ordre de grandeur supérieur ou égal à $n.\log_2 n$.

On peut déduire de ce résultat que les algorithmes de tri rapide et de tri par tas vus au chapitre 15 sont optimaux en moyenne; le tri par tas est de plus optimal dans le cas le pire.

Remarque 1 : On peut se demander, à ce point de l'étude du tri, si la classe TC ne contient pas tous les algorithmes de tri : en effet, tous les algorithmes étudiés jusqu'ici en font partie. Cependant, certains algorithmes utilisent des propriétés de la représentation des clés. Ces algorithmes sont étudiés au paragraphe suivant de ce chapitre : dans l'exemple 1 du paragraphe 2, on trie une liste de n éléments dont les clés sont les entiers distincts 1,..., n sans faire aucune comparaison.

Remarque 2 :

On a travaillé jusqu'ici avec deux opérations fondamentales pour analyser les algorithmes de tri : les comparaisons et les transferts. Le résultat énoncé ci-dessus porte seulement sur le nombre de comparaisons. Le nombre de transferts ne doit pas être négligé pour autant : si on considère le tri par insertion dichotomique (cf. chapitre 14), il est optimal dans le pire des cas en nombre de comparaisons; mais le nombre de transferts est en $\Theta(n^2)$. Pratiquement, le temps total d'exécution de l'algorithme de tri par insertion dichotomique est donc proportionnel à n^2.

Pour établir la borne inférieure du nombre de comparaisons pour les algorithmes de la classe TC, on utilise les arbres binaires de décision. C'est l'analyse de ces arbres de décision qui permet de calculer la complexité optimale de la classe TC, en nombre de comparaisons, à la fois en moyenne et dans le pire des cas.

Les arbres de décisions ont été définis et étudiés au chapitre 3 ; ils ont été également utilisés au chapitre 9 pour montrer l'optimalité de la recherche dichotomique. La figure 1 représente l'arbre de décision associé au tri à bulles d'une liste de trois éléments x_1, x_2, x_3 initialement rangés aux places $t[1], t[2], t[3]$. Rappelons que les nœuds internes de cet arbre représentent les comparaisons effectuées par l'algorithme. Les transferts des éléments ne sont pas explicités sur l'arbre mais on en voit le résultat aux feuilles : chaque feuille représente la permutation, triée en ordre croissant, des éléments de la liste initiale; par exemple, si l'ordre relatif entre les éléments x_1, x_2 et x_3 est $x_2 > x_3$, $x_1 > x_3$ et $x_1 < x_2$, l'exécution du tri à bulles à partir de la liste $x_1x_2x_3$ aboutit, après ces trois comparaisons, à la feuille contenant la liste $x_3x_1x_2$. Le tri à bulles n'étant pas optimal, certaines comparaisons peuvent être effectuées plusieurs fois au cours d'une exécution de l'algorithme. Par exemple, si x_2 est inférieur à x_3, et x_1 inférieur à x_2, le tri à bulles teste à nouveau $x_3 < x_2$; le résultat de ce test étant forcément faux, le sous-arbre gauche de ce test est vide : il ne correspond à aucune exécution possible. L'arbre de décision du tri à bulles n'est donc pas localement complet.

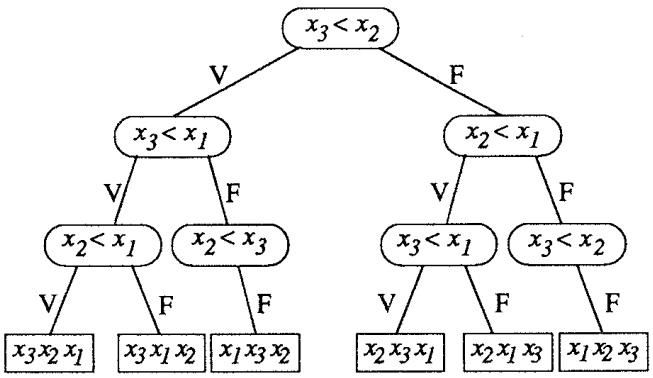


Figure 1. Arbre de décision du tri à bulles pour une liste de trois éléments.

Cet arbre a six feuilles, qui correspondent aux 3 ! permutations possibles de la liste des données. Cette observation peut être généralisée comme le montre le lemme suivant.

Lemme : L'arbre binaire de décision associé à un algorithme de tri par comparaisons sur n clés a exactement $n!$ feuilles.

> *Preuve :* Rappelons que les feuilles d'un arbre de décision indiquent les résultats des différentes exécutions de l'algorithme sur toutes les données possibles. Il y a $n!$ ordres possibles pour les n éléments qui forment la donnée d'un algorithme de tri.
>
> Ainsi chaque feuille correspond à une exécution de l'algorithme, et à chaque exécution correspond un ordre initial des éléments, donc deux permutations différentes ne peuvent apparaître dans une même feuille. Inversement, toute permutation ne peut déterminer dans l'arbre qu'un seul chemin de la racine à une feuille. L'arbre de décision a donc exactement $n!$ feuilles. □

On a vu au chapitre 3 que les complexités optimales au pire de toute classe d'algorithmes pouvant être représentés par des arbres de décision, s'expriment en fonction de la hauteur de ces arbres. On va voir ici que la complexité optimale en moyenne du tri s'exprime en fonction de la profondeur moyenne externe de ces arbres. De plus, on sait (cf. chapitre 7) que la connaissance du nombre de feuilles d'un arbre binaire permet de déterminer des minorants de sa hauteur et de sa profondeur moyenne.

Notons $TC_{\max}(n)$ (respectivement $TC_{\text{moy}}(n)$) le nombre minimal de comparaisons pour n clés, que doit effectuer dans le cas le pire (respectivement en moyenne) tout algorithme de la classe TC.

$$TC_{\max}(n) = \inf \{h(A), \text{ où } A \text{ est un arbre de décision associé à un algorithme de } TC \text{ pour une liste de } n \text{ éléments}\}$$

De plus, le nombre de comparaisons effectuées pour exécuter chacune des $n!$ permutations possibles de la liste à trier, correspond à la longueur du chemin de la racine à la feuille correspondante. Si toutes les permutations sont équiprobables, on a donc :

$$TC_{\text{moy}}(n) = \inf \{PE(A), \text{ où } A \text{ est un arbre de décision associé à un algorithme de } TC \text{ pour une liste de } n \text{ éléments}\}$$

Ces quantités sont calculées dans les deux paragraphes suivants.

1.1. Borne inférieure pour la complexité au pire

Déterminons tout d'abord un minorant de $TC_{\max}(n)$: on a vu au chapitre 7 (corollaire 1 du paragraphe 1.2) que tout arbre binaire ayant k feuilles a une hauteur

supérieure ou égale à $\lceil \log_2 k \rceil$. Il en résulte que le nombre maximum de comparaisons pour trier n éléments est toujours supérieur ou égal à $\lceil \log_2(n!) \rceil$:

$$TC_{\max}(n) \geq \lceil \log_2(n!) \rceil$$

Ce minorant est-il une borne inférieure ? Autrement dit, existe-t-il un algorithme pour lequel le minorant est atteint ? On peut se contenter d'un algorithme ayant une complexité au pire en nombre de comparaisons de même ordre de grandeur que $\lceil \log_2(n!) \rceil$: dans ce cas, on parle d'***optimalité en ordre de grandeur***; on peut aussi rechercher un algorithme effectuant, au pire, exactement $\lceil \log_2(n!) \rceil$ comparaisons : on parle alors d'***optimalité exacte.***

1.1.1. Optimalité en ordre de grandeur

Le développement asymptotique de $n!$ est donné par la formule de Stirling :

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n . \left(1 + \frac{1}{12n} + o\left(\frac{1}{n}\right)\right)$$

On en déduit

$$\log_2(n!) = \log_2 \sqrt{2\pi n} + n \log_2\left(\frac{n}{e}\right) + \log_2\left(1 + \frac{1}{12n} + o\left(\frac{1}{n}\right)\right)$$

soit

$$\log_2(n!) = n \log_2 n - n \log_2 e + \frac{1}{2}.\log_2 n + \frac{1}{2}.\log_2 2\pi + \log_2\left(1 + \frac{1}{12n} + o\left(\frac{1}{n}\right)\right)$$

et donc $\lceil \log_2(n!) \rceil = \Theta(n.\log n)$

On connaît des algorithmes de tri par comparaisons dont la complexité au pire en nombre de comparaisons est en $\Theta(n.\log n)$:

- le tri par tas,
- le tri par insertion dichotomique,
- le tri fusion.

On a donc montré que la complexité optimale, en nombre de comparaisons, dans le cas le pire, de la classe TC est en $\Theta(n.\log n)$.

Remarque : Le tri par insertion dichotomique et le tri fusion nécessitent dans le cas le pire un nombre de comparaisons équivalent exactement à $n.\log n$, alors que pour le tri par tas, le coefficient de $n.\log n$ est strictement supérieur à 1.

1.1.2. Optimalité exacte

Etudions la complexité au pire en nombre de comparaions du tri fusion : dans le pire des cas l'opération de fusion nécessite $n-1$ comparaisons, et le nombre maximal de comparaisons pour trier n éléments par l'algorithme de tri fusion vérifie la relation de récurrence :

$$F(n) = n - 1 + F(\lfloor n/2 \rfloor) + F(\lceil n/2 \rceil)$$

qui avec la condition initiale $F(1) = 0$, admet pour solution :

$$F(n) = n.\lceil \log_2 n \rceil - 2^{\lceil \log_2 n \rceil} + 1$$

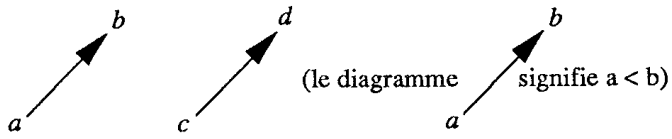
La table de la figure 2 donne les valeurs exactes de $\lceil \log_2(n!) \rceil$ et de $F(n)$ pour les premières valeurs de n. Ces valeurs coïncident pour $n = 0, 1, 2, 3, 4$, mais elles diffèrent pour $n = 5 : F(5) = 8$ et $\lceil \log_2(5!) \rceil = 7$.

n	1	2	3	4	5	6	7	8	9	10	11	12
$\lceil \log_2 (n!) \rceil$	0	1	3	5	7	10	13	16	19	22	26	29
$F(n)$	0	1	3	5	8	11	14	17	21	25	29	33

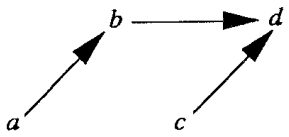
Figure 2. Borne inférieure du nombre de comparaisons pour un tri de la classe *TC* et complexité au pire, en nombre de comparaisons, du tri fusion.

Mais est-il possible de trier 5 éléments en faisant 7 comparaisons dans le pire des cas ? La réponse est oui, comme le montre la méthode suivante. Soit a, b, c, d, e les 5 éléments à trier :

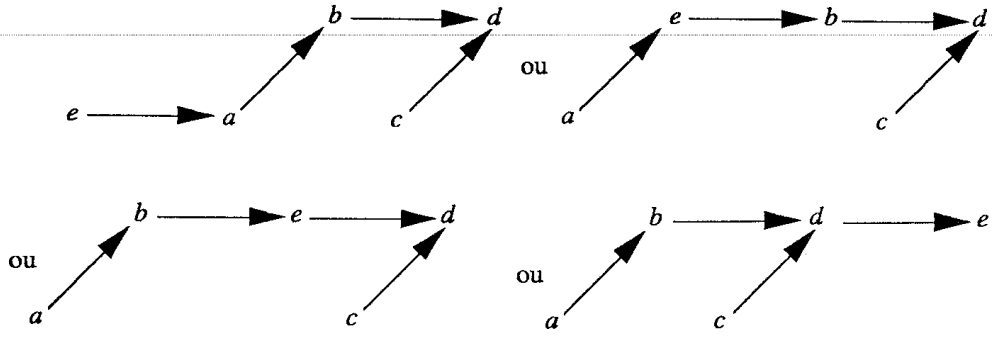
– avec deux comparaisons on ordonne deux paires d'éléments par exemple



– comparer les deux plus grands, ce qui conduit par exemple au diagramme



– insérer l'élément e à sa place dans la chaîne $a < b < d$, en comparant d'abord e et b; cela demande deux comparaisons, et conduit à l'un des quatre diagrammes suivants :



– enfin insérer c à sa place dans la chaîne des éléments plus petits que d; cela se fait toujours en deux comparaisons au plus si l'on commence par l'élément du milieu.

Cependant, il est impossible de trier 12 éléments en moins de 30 comparaisons alors que $\log_2(12!) = 29$: la preuve exhaustive de ce fait a demandé deux heures de temps calcul sur ordinateur.

Il ne peut donc pas exister de méthode de tri qui opère en $\lceil \log_2(n!) \rceil$ comparaisons pour tout n.

1.2. Borne inférieure pour la complexité en moyenne

Sur l'arbre de décision A d'un algorithme de tri par comparaisons, chaque branche correspond exactement à l'exécution de l'algorithme sur l'une des $n!$ données possibles. Les $n!$ permutations de départ étant équiprobables, le nombre moyen de comparaisons de l'algorithme est égal à $PE(A)$, la profondeur moyenne externe de l'arbre de décision.

Donc $TC_{\text{moy}}(n)$ est le minimum des $PE(A)$ pour tous les arbres de décision A associés à tous les algorithmes de TC, qui trient une liste de n éléments. Or, tout arbre binaire ayant k feuilles a une profondeur moyenne supérieure ou égale à $\log_2 k$ (lemme 3, chapitre 7, paragraphe 1.2). Il en résulte que le nombre moyen de comparaisons pour trier n éléments est toujours supérieur à $\log_2(n!)$:

$$TC_{\text{moy}}(n) \geq \log_2(n!)$$

Or, on a vu que d'après la formule de Stirling, $\log_2(n!)$ est en $\Theta(n.\log n)$. Il existe des algorithmes de tri par comparaisons dont la complexité en moyenne, en nombre de comparaisons, est de cet ordre.

• Le tri rapide a une complexité moyenne en nombre de comparaisons équivalente à $2n.\mathrm{Log}n$, soit 1,38 $n.\log_2 n$.

• Les algorithmes de tri (tri par tas, tri fusion...) dont la complexité au pire en nombre de comparaisons est en $\Theta(n.\log n)$: en effet, leur complexité en moyenne est d'ordre inférieur ou égal à leur complexité au pire, et elle est minorée par $TC_{\mathrm{moy}}(n)$.

On a donc montré que la complexité en moyenne optimale, en nombre de comparaisons, de la classe TC, est en $\Theta(n.\log n)$.

Peut-on obtenir des résultats plus précis ? On peut montrer que la valeur minimale de la longueur de cheminement externe pour un arbre binaire ayant n feuilles est

$$LCE_{\min}(n) = n.\lfloor \log_2 n \rfloor + 2.(n - 2^{\lfloor \log_2 n \rfloor})$$

La table de la figure 3 donne les valeurs de $LCE_{\min}(n!)$ pour les premières valeurs de n.

n	1	2	3	4	5	6	7
$LCE_{\min}(n!)$	0	2	16	112	832	6896	62368

Figure 3

Un algorithme serait donc exactement optimal en moyenne, s'il faisait exactement $LCE_{\min}(n!)$ comparaisons pour traiter les $n!$ données possibles, et cela pour tout entier n. Pour $n = 7$, il a été montré qu'aucune méthode de tri ne peut atteindre exactement le minorant ($LCE_{\min}(7!) = 62368$). Mais on n'a pas de résultat général. Il reste beaucoup de questions non résolues concernant d'une part la minimisation du nombre moyen de comparaisons du tri, d'autre part le lien entre les algorithmes exactement optimaux au pire et exactement optimaux en moyenne.

2. Critères de choix pour les méthodes de tri par comparaisons

On pourrait conclure de ce qui précède qu'il faut toujours utiliser le tri par tas. Mais ce n'est pas aussi simple : en effet, les résultats ci-dessus concernent seulement les nombres de comparaisons et de transferts d'éléments. Ils ne tiennent pas compte des autres opérations dues aux contrôles des boucles, aux calculs d'indices, etc. Il est donc intéressant de comparer expérimentalement les méthodes de tri.

De telles comparaisons sont toujours très difficiles à mener : les résultats vont dépendre de la machine utilisée, du compilateur. D'autres critères peuvent intervenir : on a vu, par exemple, que certains tris ont des performances très liées au fait que la liste

des éléments est presque triée; d'autres tris impliquent de très nombreux transferts, ce qui est coûteux quand les éléments sont de taille importante.

2.1. Performances expérimentales

Ce paragraphe énumère un certain nombre de résultats expérimentaux qui ont été rapportés par plusieurs auteurs.

Méthodes simples versus méthodes sophistiquées

A partir de quelle taille de la liste les méthodes simples deviennent-elles très mauvaises, ce qui justifie l'emploi de méthodes plus compliquées comme le tri rapide ou le tri par tas? Cela dépend de nombreux facteurs, mais on peut dire que pour des listes de moins d'une centaine d'éléments on ne gagne pas grand chose à utiliser des tris compliqués.

Tri rapide versus tri par tas

Expérimentalement, on constate que le tri rapide est deux à trois fois plus rapide que le tri par tas. De plus, si on prend pour pivot l'élément médian parmi le premier élément, le dernier élément et l'élément du milieu (cf. exercices du chapitre 15) le tri rapide reste meilleur que le tri par tas, même si la liste est triée!

Comparaison des méthodes simples

La méthode de sélection ordinaire est la meilleure des méthodes simples si les éléments sont de taille importante car elle minimise le nombre de transferts.

Dans le cas de listes presque triées, les méthodes par insertion sont efficaces; l'insertion dichotomique n'est pas sensiblement meilleure que l'insertion séquentielle.

Le tri à bulles donne toujours de moins bons résultats expérimentaux que les autres tris et est donc à proscrire.

2.2. Autres critères

On peut s'intéresser à deux autres critères pour comparer ces méthodes : la *stabilité et la progressivité.*

On a vu au chapitre 14 qu'un tri est stable s'il conserve l'ordre d'origine des éléments qui ont des clés égales. Les algorithmes de tri simples qui ont été donnés ici sont stables sauf la sélection ordinaire. Le tri rapide et le tri par tas ne le sont pas.

Un tri est progressif si à l'étape k du tri les k premiers éléments de la future liste sont triés. Cette dernière propriété permet de commencer un traitement sur le sous-tableau trié en parallèle avec le tri de la fin du tableau. D'une manière générale,

seuls les algorithmes de tri par sélection sont progressifs : sélection ordinaire, tri à bulles et tri par tas.

Enfin, on peut noter que les algorithmes de tri par insertion ont une troisième propriété qui peut être intéressante : on peut commencer le tri sans connaître la totalité des éléments à trier. On peut donc commencer un tel tri en parallèle avec le traitement qui produit la liste à trier.

3. Tri sur des clés de types finis

Lorsqu'on a de l'information sur les clés des éléments à trier on peut éviter certaines comparaisons. On peut même ne pas en faire du tout !

Exemple 1

Soit à trier une liste de n éléments dont les clés sont n entiers, distincts compris entre 1 et n. La liste étant représentée par le tableau t, l'algorithme suivant :

for $i := 1$ **to** n **do** $t'[clé(t[i])] := t[i]$;

trie le tableau t dans t' sans aucune comparaison et avec n transferts.

On peut améliorer cet algorithme de manière à faire du tri sur place :

for $i := 1$ **to** n **do**
  while $clé(t[i]) \neq i$ **do** $t[clé(t[i])] \leftrightarrow t[i]$;

On a $n - 1$ échanges au pire puisqu'on met à chaque fois un élément à sa place définitive et $\Theta(n)$ comparaisons entre clés.

3.1. Tri par paquets

La méthode ci-dessus se généralise au cas où le nombre de clés possibles est fini et où on a éventuellement des répétitions. On fait alors du ***tri par paquets.***

Exemple 2

Soit un paquet de 52 cartes à jouer, à trier par couleurs selon la convention

$$trèfle < carreau < cœur < pique.$$

On construit quatre paquets, un pour chaque couleur, en mettant successivement chaque carte rencontrée dans le bon paquet. On concatène ensuite ces paquets selon l'ordre des couleurs.

Soit m le nombre de valeurs possibles pour les clés et n le nombre d'éléments à trier. Cette méthode consiste à prévoir m paquets et à faire un passage sur la liste

à trier en mettant chaque élément rencontré dans le paquet qui lui correspond. On fait donc n reconnaissances de clés et n déterminations de paquet.

Pour être intéressant, ce tri doit être programmé efficacement : il est essentiel que la valeur de la clé détermine le paquet correspondant. Comme le type des clés est fini, si on programme en Pascal, on peut indexer le tableau des paquets par le type des clés. Si P est ce tableau et si $t[i].c$ est la clé du $i^{ième}$ élément de la liste à trier, l'indice du paquet où doit être mis $t[i]$ est donné par $t[i].c$. Dans d'autres langages que Pascal, on peut utiliser un adressage indexé dans P après avoir calculé le rang de la clé dans son type.

Comme on ne connaît pas en général le nombre d'éléments à prévoir dans chaque paquet, on doit utiliser des listes chaînées pour représenter les paquets. Le tableau P mentionné ci-dessus contient alors des pointeurs vers le début de ces listes. La liste triée finale est obtenue en concaténant les m listes obtenues, dans l'ordre des clés. Cela nécessite un parcours des m sous-listes (qui ont en tout n éléments) avec $m-1$ mises à jour de pointeur. Cette opération peut être améliorée si on garde pour chaque paquet dans P son adresse de tête et de fin comme on le voit à la figure 4. Dans ce cas la concaténation ne demande que m transferts de pointeurs ($m-1$ mises à jour du dernier pointeur de chaque paquet et une mise à jour du pointeur de fin de liste du 1^{er} paquet).

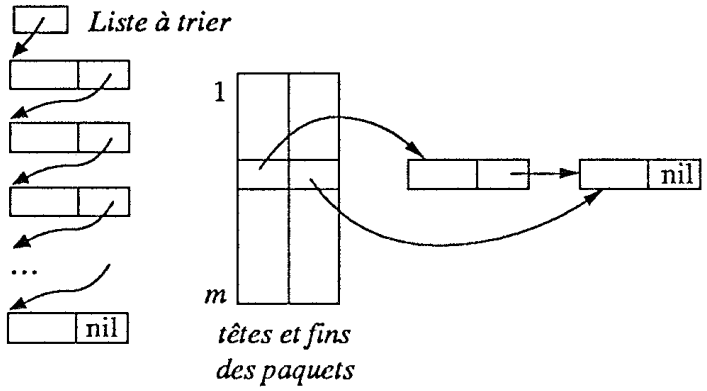


Figure 4. Représentation des paquets.

Dans tous les cas, cette méthode est coûteuse en mémoire : elle demande $2m$ pointeurs pour le tableau P et de la place pour n éléments et n pointeurs. Cependant, si la liste à trier est une liste chaînée, on peut utiliser les places de la liste pour les paquets. Dans ce cas, la méthode n'exige que $2m$ pointeurs supplémentaires.

L'existence du deuxième pointeur, vers la fin de chaque paquet, permet de faire en temps constant des adjonctions en fin de paquet. Cela rend le tri par paquets stable sans changer sa complexité. On va utiliser cette possibilité dans la suite.

3.2. Application aux clés structurées

On a vu au chapitre 14 que certains tris se font selon des clés structurées. Lorsqu'on a des clés structurées dont les sous-clés sont de types finis, on peut trier en enchaînant des tris par paquets sur chaque sous-clé.

3.2.1. Principe

Considérons tout d'abord un exemple.

Exemple 2 bis

On veut trier le paquet de cartes à jouer par couleurs *et* par valeurs des cartes

$$2 < 3 < ... < D < R < As$$

On peut faire les distributions de deux manières :

1) on crée 4 paquets correspondant aux couleurs; puis on trie les 4 paquets séparément; puis on les concatène dans l'ordre des couleurs;

2) on crée 13 paquets correspondant aux valeurs des cartes; puis on les concatène dans l'ordre des valeurs; on crée ensuite 4 paquets correspondant aux couleurs en conservant l'ordre précédent sur les valeurs et on concatène selon l'ordre des couleurs.

La première méthode nécessite que les quatre premiers paquets soient triés séparément avant de les concaténer. En effet, si ce n'est pas le cas, le deuxième tri par paquet détruit l'effet du premier comme on le voit sur l'exemple ci-dessous.

Soit la liste : $[ab\ ba\ aa\ bb]$

On veut trier cette liste par ordre alphabétique en utilisant deux tris par paquets.

Si on commence par la première lettre, et que l'on concatène les paquets obtenus, on a, en notant $[\ ]_a$ le paquet obtenu pour la lettre a :

$$[ab\ aa]_a[ba\ bb]_b = [ab\ aa\ ba\ bb]$$

Si on effectue alors un tri par paquets sur la deuxième lettre, on obtient :

$$[aa\ ba]_a[ab\ bb]_b = [aa\ ba\ ab\ bb]$$

Le résultat n'est pas trié !

Il faut donc trier les paquets séparément et leurs nombres se multiplient (on obtient 4 fois 13 paquets pour l'exemple 2 bis).

Dans la deuxième méthode, on n'a à chaque instant que le nombre de paquets correspondant à la sous-clé traitée, et si chaque étape de tri par paquets est stable on obtient finalement la liste triée. Pour la liste ci-dessus, le tri sur la deuxième lettre donne :

$$[ba\ aa]_a[ab\ bb]_b = [ba\ aa\ ab\ bb]$$

puis le tri sur la première lettre donne :

$$[aa\ ab]_a[ba\ bb]_b = [aa\ ab\ ba\ bb]$$

On effectue donc les tris par paquets successifs en commençant par le critère le moins important. Il est essentiel que le tri par *paquets* soit *stable* pour que l'effet des tris précédents soit conservé.

Remarque : Si on commence le tri par un tri par paquets sur le critère le plus important, on obtient une liste «presque triée». On peut alors enchaîner avec un tri par insertion qui, on l'a vu au chapitre 14, est efficace sur ce type de listes.

3.2.2. Ordre lexicographique

Si on a des clés structurées, avec k sous-clés, $< c_1, ..., c_k >$ de types $t_1, ..., t_k$ munis chacun d'une relation d'ordre et que l'on prend la convention que les clés sont numérotées par ordre d'importance décroissante, on dit que :

$$< a_1, ..., a_k > < < b_1, ..., b_k >$$

dans les cas suivants :

si (1) $a_1 <_1 b_1$, où $<_1$ est la relation d'ordre sur t_1

ou (2) $\exists p \leq k$ tel que $a_i = b_i$ pour $i < p$ et $a_p <_p b_p$, où $<_p$ est la relation d'ordre sur t_p.

On appelle cet ordre, ***ordre lexicographique.***

L'ordre alphabétique sur les mots de longueur k est un exemple d'ordre lexicographique.

Un tri selon l'ordre lexicographique peut se faire en enchaînant k tris selon les relations $<_k, <_{k-1}, ..., <_1$.

Remarque : La convention de numérotation des sous-clés ne doit pas faire croire que l'on traite toujours les sous-clés «de la droite vers la gauche». En effet, considérons des clés du type Pascal suivant que l'on veut trier par dates croissantes :

```
type date = record
                jour : 1..31;
                mois : (jan, fev, mar, avr, mai, jun, jul, aou, sept, oct, nov, dec);
                année : 0..2000
            end;
```

On commencera par trier sur le premier champ, puis sur le deuxième et enfin sur l'année qui est le critère le plus important.

3.2.3. Analyse de la complexité

Au départ, on a une liste de n éléments dont les clés sont de la forme $< c_1, ..., c_k >$, chaque c_i étant de type t_i fini : il y a un nombre fini m_i de valeurs de type t_i.

Supposons que la liste de départ soit chaînée. Chaque étape de tri par paquets construit une nouvelle liste chaînée. Pour la sous-clé de rang p, on distribue n éléments dans m_p paquets ce qui implique n reconnaissances de clés; on concatène ensuite ces m_p paquets, ce qui, on l'a vu, peut se faire en m_p transferts de pointeurs. L'entier p prend successivement les valeurs $k, k-1, ...1$. Le tri demande donc en tout $k.n$ reconnaissances de clés, chacune étant suivie d'un transfert dans un paquet et M transferts de pointeurs, où $M = \sum_{p=1}^{k} m_p$.

Or k et M sont des constantes. On a donc un tri de complexité linéaire en nombre de reconnaissances de clé et de transferts d'éléments, et constant en nombre de transferts de pointeurs.

3.3. Tri radix

Dans le cas où les clés sont des nombres entiers, ou des chaînes de caractères, de longueur fixe, on peut considérer chaque chiffre ou caractère comme une sous-clé.

On effectue alors k tris par paquets (k étant la longueur de la clé) avec, à chaque fois le même nombre de paquets : ce nombre est déterminé par la base de numérotation utilisée (10 en général, 2 parfois) ou par le nombre de caractères autorisés. Si m est ce nombre, cette méthode effectue $k.n$ extractions de chiffre ou de caractère, chacune étant suivie d'un transfert vers un paquet, et $k.m$ transferts de pointeurs. Elle nécessite $2m$ pointeurs.

On parle de ***tri radix*** dans ce cas car le partage de la clé en sous-clés est déterminé, dans le cas des nombres entiers, par la base (le radical) choisie pour la numérotation; les chaînes de caractères peuvent être considérées comme des nombres exprimés dans une base particulière. Ce tri est également appelé ***tri lexicographique.***

Dans le cas où les clés ne sont pas toutes de même longueur, l'ordre lexicographique est défini par :

$$< a_1, ..., a_p > < < b_1, ..., b_q >$$

si (1) $a_1 <_1 b_1$

ou (2) $\exists k \leq \min(p, q)$ tel que $a_i = b_i$ pour $i < k$ et $a_k <_k b_k$

ou (3) $p < q$ et $a_i = b_i$ pour $i = 1, ..., p$.

La propriété (3) correspond au fait que dans un dictionnaire le mot «chat» apparaît avant le mot «chaton».

Une première méthode pour généraliser le tri radix à ce cas consiste, si la plus longue clé est de longueur *lmax*, à rendre toutes les autres de cette même longueur en les complétant par un symbole spécial, plus petit que tous les autres. On peut alors utiliser la méthode précédente. Cependant, s'il n'y a que peu de clés de longueur *lmax*, les premiers passages se font sur la totalité des n éléments alors que peu d'éléments sont vraiment traités : la majorité des éléments va dans le paquet associé au symbole spécial.

Il est plus efficace, dans le cas où la longueur de chaque clé est connue, de commencer par distribuer les éléments de la liste par paquets selon la longueur de leurs clés. Puis on commence par trier seulement le paquet correspondant à *lmax*, selon le chiffre ou le caractère de rang *lmax*. On continue en faisant des tris par paquets successifs selon le chiffre ou le caractère de rang $lmax-i$ $(i = 1, ..., lmax-1)$ en ajoutant, avant de traiter le rang k, le paquet des éléments dont la clé est de longueur k. Il faut ajouter ce paquet devant la liste en cours de tri à cause de la propriété (3) de l'ordre lexicographique.

La programmation et l'analyse de ces deux méthodes sont laissées en exercice.

Exercices

1. On considère deux tableaux A et B de n éléments. On dit que A est contenu dans B si tous les éléments de A figurent au moins une fois dans B.

a) Ecrire une fonction qui, sans trier les tableaux, dit si A est contenu dans B. Quelle est la complexité de l'algorithme en nombre de comparaisons ?

b) Améliore-t-on cette complexité si l'on trie l'un des deux tableaux et si oui, préciser lequel. Améliore-t-on à nouveau la complexité si l'on trie les deux tableaux ?

2. Montrer que la valeur minimale de la longueur de cheminement externe d'un arbre binaire ayant n feuilles est : $LCE_{\min}(n) = n\lfloor \log_2 n \rfloor + 2.(n - 2^{\lfloor \log_2 n \rfloor})$ en utilisant les résultats établis au chapitre 7.

3. Soit une liste dont les n éléments sont 0 ou 1 et qui est représentée par le tableau T. Trouver un algorithme qui trie le tableau T, en utilisant $n - 1$ comparaisons entre éléments de T. Pourquoi l'algorithme a-t-il une complexité en nombre de comparaisons d'ordre strictement inférieur à $\Theta(n \log n)$?

4. Dessiner l'arbre de décision correspondant à chacun des cas suivants :

a) tri, par insertion séquentielle, de 4 éléments

b) tri, par insertion dichotomique, de 4 éléments

Donner la longueur de cheminement externe de chaque arbre.

5. Donner un arbre de décision qui trie 6 éléments, tel que tous ses nœuds externes sont sur les niveaux 10 et 11. (Indication : on pourra utiliser la méthode donnée au paragraphe 1.1 pour l'optimalité exacte.)

6. Dans cet exercice, on s'intéresse au *tri d'éléments non tous distincts*. Quand on effectue une comparaison notée $X_i : X_j$, on peut avoir l'un des trois résultats

$X_i < X_j, X_i = X_j, X_i > X_j$. On représente ces comparaisons et leurs résultats dans un arbre ternaire de décision. On utilise la branche gauche si $X_i < X_j$, celle du milieu si $X_i = X_j$ et celle de droite si $X_i > X_j$. On note P_n le nombre de résultats que l'on peut obtenir pour le tri de n éléments non tous distincts ($P_1 = 1; P_2 = 3; ...$)

a) Représenter par un arbre ternaire de décision sans comparaison redondante, le tri de 3 éléments non tous distincts x_1, x_2, x_3. Vérifier que $P_3 = 13$.

b) On note $C_{\max}(n)$ le nombre minimum de comparaisons nécessaires dans le cas le pire pour trier n éléments et déterminer toutes les relations d'égalité entre eux. Montrer que : $C_{\max}(n) \geq \log_3 P_n$.

c) Etablir la relation $P_n = \sum_{k \geq 0} C_n^k P_{n-k}$, pour $n > 0$. (On pose $P_0 = 1$.)

d) En utilisant la série génératrice exponentielle des P_n (cf. annexe)

$$P(z) = \sum_{n \geq 0} P_n \frac{z^n}{n!}$$

prouver que $P(z) = \frac{1}{2 - e^z}$.

e) On note $P(n, k)$ le nombre de partitions de n objets en exactement k classes (cf. exercice n° 34 de l'annexe). Prouver directement la relation :

$$P_n = \sum_{k>0} k!\ P(n, k), \text{ pour } n > 0$$

Retrouver à l'aide de cette relation, la valeur de $P(z)$.

7. On rappelle que $TC_{\max}(n)$ désigne le nombre minimal de comparaisons nécessaires dans le cas le pire pour trier n éléments distincts. $C_{\max}(n)$ est défini comme dans l'exercice n° 6. C'est le nombre minimal de comparaisons nécessaires dans le cas le pire pour trier n éléments non tous distincts et déterminer les relations d'égalité entre eux. Montrer que $C_{\max}(n) = TC_{\max}(n)$.

8. Dans cet exercice, les clés des éléments qu'on trie ont pour seules valeurs 0 et 1.

a) Construire un arbre ternaire de décision, sans comparaison redondante, (cf. exercice n° 6), pour trier 4 éléments X_1, X_2, X_3 et X_4. (Par exemple, si on a $X_1 < X_2$ et $X_3 < X_4$, on sait que $0 = X_1 = X_3 < X_2 = X_4 = 1$.) Quel est le nombre de résultats possibles dans le cas général (c'est-à-dire le nombre de feuilles) ?

b) Sachant que les 2^4 suites possibles de valeurs pour (X_1, X_2, X_3, X_4) sont équiprobables, utiliser l'arbre ternaire de décision précédent pour déterminer le nombre moyen de comparaisons nécessaires pour trier 4 éléments.

c) Montrer que, dans le pire des cas, le nombre de comparaisons nécessaires pour trier n éléments de valeurs 0 ou 1, est $n-1$.

9. a) Quel est le nombre minimal d'échanges qui permettent de trier la permutation 3, 7, 6, 9, 8, 1, 4, 5, 2 ?

b) En général, soit $\sigma = \sigma(1), \sigma(2), ..., \sigma(n)$ une permutation de $\{1, 2, ..., n\}$. On note $E_{\max}(n)$ le nombre minimal d'échanges pour trier σ. Montrer que $E_{\max}(n)$ est $n - c$, où c est le nombre de cycles de σ. (Un ***cycle*** dans une permutation σ est une suite $i_1, ..., i_p$ telle que $\sigma(i_1) = i_2, ..., \sigma(i_j) = i_{j+1}, ...$, et $\sigma(i_p) = i_1$. Un cycle comporte au moins deux éléments : on ne compte pas les singletons qui correspondent aux cas $\sigma(i) = i$.)

10. Tout algorithme de tri qui procède par échanges peut être représenté par un *arbre de comparaisons-échanges*, à savoir un arbre binaire dont les nœuds internes sont étiquetés par $X_i : X_j$, pour $i < j$; ce nœud est interprété comme :

si $X_i \leq X_j$, emprunter la branche gauche de l'arbre ;

si $X_i > X_j$, continuer en échangeant les éléments X_i et X_j et emprunter la branche droite de l'arbre.

Quand on arrive à un nœud externe, on a : $X_1 \leq X_2 \leq ... \leq X_n$. Dans un tel arbre de comparaisons-échanges, on spécifie donc aussi bien les comparaisons que les transferts.

a) Soit $CE_{\max}(n)$ le nombre minimal de comparaisons-échanges nécessaires pour trier n éléments, dans le pire de cas, à l'aide d'un arbre de comparaisons-échanges. Prouver que : $CE_{\max}(n) \leq TC_{\max}(n) + n - 1$.

b) Prouver que $CE_{\max}(5) = 8$.

11. Programmer le tri par paquets d'une liste chaînée dont les clés sont des entiers compris entre 0 et 63.

12. Ecrire un programme qui lit une liste dont les clés sont du type date, donné au paragraphe 3.2, et qui trie cette liste en utilisant des tris par paquets successifs.

13. Programmer le tri par ordre alphabétique d'une liste de mots de p caractères ne comportant que des lettres minuscules.

14. On considère que les clés d'une liste à trier sont des nombres binaires de longueur k. Ecrire une procédure récursive, similaire à celle du tri rapide, qui commence par

partitionner la liste selon le bit de plus fort poids des clés, puis recommence pour chaque partition obtenue, sur le bit suivant. On s'arrête quand une partition est vide, ou quand tous les bits ont été examinés.

Calculer la complexité au pire et en moyenne de cette procédure, en nombre d'extractions et de comparaisons de bits.

15. On considère les généralisations du tri radix données au paragraphe 3.3 dans le cas de chaînes de lettres minuscules de longueurs variables. Chaque clé comporte un champ qui indique sa longueur.

a) Ecrire un programme qui, étant donné la longueur maximum des clés, complète toutes les clés à cette longueur par un caractère spécial. Puis utiliser le programme de l'exercice 13. Décrire les étapes successives de l'algorithme dans le cas où 75 % des clés sont de longueur inférieure à la moitié de la longueur maximum.

b) Programmer une distribution des éléments par paquets selon la longueur des clés, suivie d'une suite de tris par paquets comme indiqué au paragraphe 3.3. Décrire les étapes successives de l'algorithme dans le cas envisagé en a) et analyser la complexité de l'algorithme dans ce cas en nombre de reconnaissances de clés et de transferts.

Lectures conseillées pour les chapitres 14, 15 et 16

Aho, Hopcroft & Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

Aho, Hopcroft & Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983.

Knuth, *The Art of Computer Programming*, Vol. 3 : *Sorting and Searching*, Addison-Wesley, 1973.

Sedgewick, *Algorithms*, Addison-Wesley, 1983.

Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, 1976.

Chapitre 17
Tri externe

1. Introduction

On utilise les méthodes de tri externe dans le cas, fréquent, où la liste à trier ne tient pas en mémoire centrale. Cette liste est donc stockée sur un support externe. Actuellement, on utilise le plus souvent des bandes ou disques magnétiques. Il y a quelques années on utilisait beaucoup les cartes perforées.

Exemple : Chaque élément de la liste a une taille de 500 caractères. On a une mémoire centrale utilisable pour le tri d'environ 500 000 caractères. Les éléments sont stockés sur bandes magnétiques et on ne connaît pas leur nombre exact à l'avance : on sait qu'il est de l'ordre de plusieurs dizaines de mille.

On rencontre ce type de problème chaque fois que l'on doit gérer un grand volume d'information, par exemple dans les problèmes de gestion ou de bases de données. *La conception et l'analyse des méthodes de tri externe font intervenir des critères différents de ceux du tri interne :* en effet les temps d'entrée-sortie sont prépondérants (de l'ordre de la milliseconde) par rapport aux temps des opérations en mémoire centrale (inférieurs à la micro-seconde et pouvant descendre jusqu'à quelques dizaines de nano-secondes). De plus les caractéristiques du système d'exploitation utilisé peuvent intervenir (organisation des mémoires tampon, lectures anticipées) ainsi, bien sûr que les contraintes liées au support externe utilisé (limitation à l'accès séquentiel sur bande magnétique; coût des mouvements de la tête de lecture sur disque qui est très variable selon les modèles utilisés).

Il n'est pas possible de considérer ici tous les cas particuliers... On va donc se limiter à des méthodes qui sont basées sur une *lecture séquentielle* sur les supports externes. Ce choix, obligatoire dans le cas de bandes magnétiques, permet dans le cas général une bonne gestion des opérations d'entrée-sortie par le système d'exploitation. Les méthodes présentées peuvent être améliorées ou revues en fonction des caractéristiques des supports externes utilisés et du système d'exploitation.

La plus ancienne méthode de tri externe est le ***tri radix externe*** : comme pour le tri radix interne, les éléments de la liste sont distribués en sous-listes, selon une décomposition de la clé.

Supposons que les clés soient formées de lettres. La distribution de la liste se fait comme dans la figure 1 (où $L_{x\ldots}$ désigne la sous-liste formée des éléments de L dont la clé commence par x).

On effectue cette distribution jusqu'à ce que chaque sous-liste tienne en mémoire. On fait alors du tri interne et on concatène toutes les sous-listes triées ainsi obtenues.

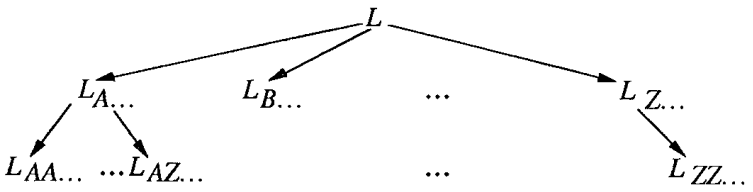


Figure 1. Tri radix externe.

Cette méthode était bien adaptée au traitement des paquets de cartes. C'est sur cette base que fonctionnent encore beaucoup de trieuses : il n'est pas très coûteux d'avoir un nombre élevé de réceptacles de sortie. Ce n'est pas le cas pour les dérouleurs de bande, ou pour les unités de disque magnétique où il faut travailler sur des unités différentes, sinon les mouvements du bras de lecture sont trop coûteux.

Dans le cas de supports magnétiques, le plus fréquent actuellement, on adopte une démarche inverse qui est appelée ***tri et fusion*** : on commence par faire du tri interne sur des sous-listes de taille aussi grande que possible; on obtient ainsi un ensemble de sous-listes triées que l'on appelle des ***monotonies***; on interclasse ces monotonies en utilisant les supports externes en entrée et en sortie – on appelle cette étape la ***fusion***; quand il ne reste qu'une seule monotonie le tri est terminé.

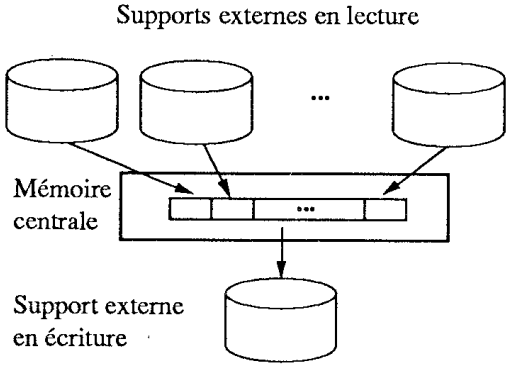


Figure 2. Fusion dans un tri externe.

Fusionner p monotonies nécessite peu de place en mémoire centrale : à chaque instant on a p éléments présents. On exécute répétitivement les opérations suivantes : écrire le plus petit de ces p éléments dans le fichier résultat de la fusion; lire un nouvel élément dans le fichier d'origine de l'élément qui vient d'être écrit.

Ce sont des méthodes de ce type qui vont être étudiées en détail dans ce chapitre. Toutes ces méthodes visent à utiliser au mieux les supports externes, à éviter la situation de blocage qui se produit si on doit fusionner deux monotonies qui se trouvent sur un même support, à minimiser le nombre de fois où un élément de la liste est lu et écrit.

Considérons un exemple de mauvaise organisation sur les supports externes. On suppose qu'à la fin des tris internes on a six monotonies, que l'on dispose de trois supports externes permettant des entrées-sorties simultanées, et que l'on a réparti les monotonies de la manière suivante; le support $S1$ contient la monotonie $M1$ suivie de la monotonie $M3$ suivie de la monotonie $M5$, le support $S2$ contient les monotonies $M2$ puis $M4$ puis $M6$, le support $S3$ est vide (réservé aux résultats des fusions); le symbole $\Downarrow$ indique la position de la tête de lecture sur chaque support.

$S1$	$\Downarrow$	$M1$	$M3$	$M5$
$S2$	$\Downarrow$	$M2$	$M4$	$M6$
$S3$	$\Downarrow$			

Après l'interclassement de $M1$ et $M2$ on a :

$S1$	$\Downarrow$	$M3$	$M5$
$S2$	$\Downarrow$	$M4$	$M6$
$S3$	$M1+M2$	$\Downarrow$	

$M1+M2$ est le résultat de la fusion de $M1$ et de $M2$. Après l'interclassement de $M3$ et $M4$ on a :

$S1$		$\Downarrow M5$
$S2$		$\Downarrow M6$
$S3$	$M1+M2$	$M3+M4\Downarrow$

Puis, après l'interclassement de $M5$ et $M6$, on a :

$S1$	$\Downarrow$		
$S2$	$\Downarrow$		
$S3$	$M1+M2$	$M3+M4$	$M5+M6\Downarrow$

On ne peut alors plus rien faire d'efficace puisque toutes les monotonies sont sur le même support : il faut faire des recopies, ce qui est coûteux en temps, quand on a beaucoup de monotonies, ou qu'elles sont longues. On a mis en évidence le problème essentiel du tri externe par tri et fusion : comment répartir les monotonies sur les supports externes ? comment organiser les fusions ? Ce problème est d'autant plus délicat qu'on ne sait pas, en général, le nombre de monotonies que l'on va obtenir après les tris internes...

Dans ce chapitre on étudie tout d'abord les méthodes particulières à la construction de monotonies. On donne ensuite l'algorithme d'interclassement de deux monotonies et sa complexité. Puis on présente deux stratégies de répartition sur les supports externes : le tri équilibré et le tri polyphasé. On étudie enfin les optimisations apportées par la lecture en arrière de certaines monotonies.

2. Construction des monotonies

Cette première étape consiste à répartir les éléments en sous-listes triées appelées *monotonies* (en anglais, run). C'est une suite de lectures, de tris internes et d'écritures des résultats de ces tris dans des fichiers.

2.1. Décomposition en sous-listes de même taille

C'est la méthode la plus simple : on remplit au maximum la mémoire centrale disponible. On choisit une méthode de tri interne qui ne nécessite pas de mémoire auxiliaire. A la fin du tri interne on écrit sur un support externe la monotonie obtenue. Dans ce cas, toutes les monotonies sont de taille égale, connue.

Exemple : Si on reprend l'exemple de l'introduction, on a des monotonies de 1000 éléments. Si le nombre d'éléments total est 100 000, cela signifie que l'on obtient 100 monotonies.

Si n éléments tiennent en mémoire centrale, et si on a N éléments en tout, la construction des monotonies nécessite N/n tris de n éléments à faire. Donc, si on prend une méthode de tri interne optimale, en $\Theta(n\log n)$ comparaisons, la construction des monotonies est en $\Theta(N\log n)$ comparaisons. Cependant, les

nombres de lectures et d'écritures sont bien plus significatifs pour évaluer le temps d'exécution : il y en a N de chaque type.

2.2. Sélection et remplacement

Cette méthode a pour but d'augmenter la taille des monotonies produites. Elle permet de réduire sensiblement le nombre de monotonies, donc le nombre de fusions à effectuer par la suite. Les monotonies obtenues ont des tailles variables.

On utilise une méthode de tri par sélection : le tri interne commence par sélectionner le minimum des éléments présents en mémoire centrale; on peut donc écrire tout de suite ce minimum dans la monotonie en cours de création; on a, de ce fait, libéré une place en mémoire centrale; on peut alors lire un nouvel élément sur le support externe où se trouve la liste globale, et le mettre à cette place.

Pour que le nouvel élément ne provoque pas la production d'une monotonie incorrecte (c'est-à-dire non triée), il faut éviter de le sélectionner s'il est plus petit que le minimum que l'on vient d'écrire. Dans ce cas, on le marque pour qu'il soit ignoré dans les sélections ultérieures. Cet élément fera partie de la monotonie suivante.

On recommence cette opération jusqu'à ce que tous les éléments en mémoire centrale soient marqués : on ferme alors la monotonie et on en crée une nouvelle.

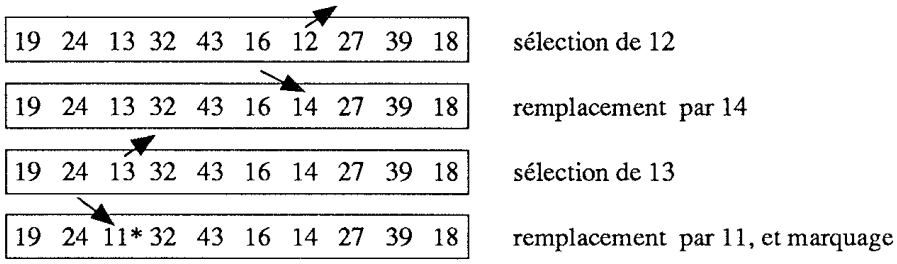


Figure 3. Exemple de sélections et remplacements.

Avant de commencer une nouvelle monotonie, il faudrait supprimer toutes les marques, c'est-à-dire parcourir la mémoire centrale ! Pour éviter cette perte de temps, on travaille «en bascule» : pour commencer, on sélectionne parmi les éléments non marqués et on marque tout élément de remplacement qui est inférieur à l'élément qui vient d'être sélectionné; pour la monotonie suivante, on sélectionne parmi les éléments marqués et on marque tout élément de remplacement qui est supérieur à l'élément qui vient d'être sélectionné, et ainsi de suite.

{*Ce programme lit les éléments d'une liste L, et les met dans des monotonies* L_k. *A la fin du programme le nombre de monotonies construites est contenu dans la variable* k. *Le programme utilise un tableau* t *de* n *éléments et un tableau* m *de* n *booléens qui indique quels sont les éléments marqués dans* t}

```
var i, k : integer; marque : boolean; min : Elément;
    t : array[1..n] of Elément; m : array[1..n] of boolean;
begin
    {initialisations :}
    for i := 1 to n do begin
        read(L, t[i]); {la liste à trier se trouve sur le support externe L}
        m[i] := false {aucun élément n'est marqué}
    end;
    marque := true;
    k := 1; {n° de la monotonie en cours}
    select(t, n, marque, min, i);
    {cette procédure trouve un minimum min de t et son indice i en ne
    considérant que les éléments non marqués, c'est-à-dire les t[j] tels que
    m[j] =not marque. Si tous les éléments sont marqués on obtient i = 0}
    {corps du programme :}
    while (not eof(L)) do begin
        while (i ≠ 0) {il existe des éléments non marqués}
                    and (not eof(L)) do begin
            write (L_k, min);{L_k est le support de la k^ième monotonie}
            read (L, t[i]); {remplacement}
            if t[i] < min then m[i] := marque;
            select(t, n, marque, min, i);
        end;
        {nouvelle monotonie ; on bascule les marques}
        k := k + 1; marque := not marque; select(t, n, marque, min, i)
    end;
    {L a été lue complètement : on trie les éléments présents en mémoire
    sauf le dernier élément écrit qui n'a pas été remplacé}
    t[i] ↔ t[n]; tri(t, n − 1); k := k + 1;
    for i := 1 to n do write (L_k, t[i]);
    {on peut alors enchaîner sur les fusions de monotonies} ...
end;
```

On peut améliorer cet algorithme en maintenant en mémoire centrale les clés organisées en tas. Dans ce cas, on a $\Theta(\log n)$ comparaisons à chaque lecture pour conserver la structure de tas et la sélection ne nécessite aucune comparaison. Globalement, si la liste à trier comporte N éléments, on a une complexité en $\Theta(N \log n)$ pour les comparaisons, comme dans la méthode précédente. Mais de toute façon, ce qui est coûteux en temps dans la construction des monotonies, ce sont les N lectures et les N écritures.

L'intérêt de cette méthode ne réside donc pas dans sa complexité, mais plutôt dans la plus grande taille des monotonies obtenues : expérimentalement, dans le cas de clés aléatoires, on observe que les monotonies ont environ deux fois la taille de celles qui peuvent être obtenues par la méthode précédente. On gagne en gros un

passage[1] pour les fusions (exactement un si les fusions se font deux à deux) donc de l'ordre de N lectures et N écritures. Si les clés sont partiellement triées, les monotonies peuvent être beaucoup plus grandes (voir exercices).

3. Interclassement (Fusion)

On donne ci-dessous l'algorithme d'interclassement de deux tableaux triés. Etant donné un tableau $t1$ de n éléments, trié, et un tableau $t2$ de m éléments, trié, il s'agit de construire un tableau t, trié, qui contient la réunion des éléments de $t1$ et $t2$ en conservant les répétitions : si x a $n1$ occurrences dans $t1$ et $n2$ occurrences dans $t2$, x a $n1+n2$ occurrences dans t. La procédure donnée ici met deux sentinelles (ici on utilise un élément *maxelt* plus grand que tous les autres) à la fin des tableaux à interclasser. Cela permet d'éviter des copies de fin de tableau quand un des tableaux est épuisé. Cette méthode se généralise sans problème au cas de p tableaux (voir exercices).

On a $m+n$ transferts, qui dans le cas d'un tri externe correspondent à des entrées-sorties : $m+n$ lectures et $m+n$ écritures. On a $m+n$ comparaisons entre éléments.

```
procedure fusion(var t1 : array[1..n + 1] of Elément;
                 var t2 : array[1..m + 1] of Elément;
                 var t : array[1..n + m] of Elément );
var i, j, k : integer;
begin
    i := 1 ; j := 1; k := 1; t1[n + 1] := maxelt; t2[m + 1] := maxelt;
    while k ≤ m + n do begin
        if t1[i] < t2[j] then begin t[k] := t1[i]; i := i + 1 end
        else begin t[k] := t2[j]; j := j + 1 end;
        k := k + 1
    end
end fusion;
```

Cet algorithme s'adapte facilement au problème de la fusion des monotonies dans un tri externe : si on a p monotonies à fusionner, on utilise un tableau de p éléments en mémoire centrale qui contient le plus petit élément non encore pris en compte de chaque monotonie (voir figure 2); lorsqu'une monotonie est terminée on met une valeur sentinelle dans la place du tableau qui lui correspond.

Il reste donc à étudier le problème de la répartition des monotonies sur les supports externes.

---

[1] Un *passage* est une suite de fusions où tous les éléments à trier sont lus une fois et une seule.

4. Une première stratégie de répartition : le tri équilibré

Cette méthode utilise un nombre pair de supports externes : unités de disque ou dérouleurs de bande. A chaque instant, si on a $2p$ supports, on utilise p supports en lecture et p supports en écriture. On équilibre systématiquement le nombre de monotonies sur les supports en écriture.

Supposons que l'on ait quatre supports, $S1, S2, S3, S4$. On en utilise deux en lecture et deux en écriture. Considérons le cas où 1000 éléments peuvent être triés en mémoire centrale et où on a 5000 éléments en tout. Le tri complet comporte les passages suivants sur l'ensemble des éléments à trier.

Passage 0 : Construction des monotonies.

Pour simplifier on convient que l'on découpe en monotonies de taille égale : on en a donc 5. La taille des monotonies est indiquée entre parenthèses.

$S1$	$M1(1000)$	$M3(1000)$	$M5(1000) \Downarrow$
$S2$	$M2(1000)$	$M4(1000) \Downarrow$	
$S3$	$\Downarrow$		
$S4$	$\Downarrow$		

Rembobinage si on utilise des bandes.

Passage 1 : Fusion, $S1$ et $S2$ servent en lecture, $S3$ et $S4$ alternativement en écriture.

$S1$		$\Downarrow$
$S2$	$\Downarrow$	
$S3$	$M1 + M2(2000)$	$M5(1000) \Downarrow$
$S4$	$M3 + M4(2000) \Downarrow$	

Rembobinage si on utilise des bandes.

Passage 2 : Fusion, $S1$ et $S2$ servent en écriture, $S3$ et $S4$ en lecture.

$S1$	$M1 + M2 + M3 + M4(4000) \Downarrow$	
$S2$	$M5(1000) \Downarrow$	
$S3$		$\Downarrow$
$S4$	$\Downarrow$	

Rembobinage

Passage 3 : Fusion, $S1$ et $S2$ servent en lecture, $S3$ et $S4$ en écriture. On obtient la liste triée sur $S3$.

Dans ce cas précis, où on a quatre supports, on dit que l'on fait du tri équilibré à deux voies. La méthode se généralise de manière évidente au cas de $2p$ supports. On parle alors de ***tri équilibré à p voies.***

L'exemple précédent avec six supports donne les configurations suivantes.

	Passage 0	Passage 1	Passage 2
$S1$	$M1\ M4$		$M1+M2+M3+M4+M5$
$S2$	$M2\ M5$		
$S3$	$M3$		
$S4$		$M1+M2+M3$	
$S5$		$M4+M5$	
$S6$			

Si on a N éléments à trier et n places en mémoire centrale, les tris internes produisent $\lceil N/n \rceil$ monotonies.

Si on fait ensuite des fusions à p voies, chaque passage divise le nombre de monotonies par p. On aura $\Theta(\log_p(N/n))$ passages. A chaque passage on a N lectures et N écritures. On a donc, pour les fusions, une complexité en $\Theta(N\log_p(N/n))$ pour les entrées-sorties et ceci dans le cas de $2p$ supports.

Cette méthode a le mérite de la simplicité. Elle est cependant loin d'être optimale : on le voit dans l'exemple ci-dessus où le support $S6$ n'est jamais utilisé, et dans l'exemple à deux voies où la monotonie $M5$ est recopiée deux fois inutilement.

5. Le tri polyphasé

Cette méthode permet d'utiliser en permanence tous les supports. Si on a q supports, à chaque instant on en a $q-1$ en lecture et 1 en écriture. La stratégie utilisée est de répartir au départ les monotonies sur $q-1$ supports, en gardant un support vide. On effectue alors des fusions en utilisant ce support en écriture. Dès qu'un support devient vide c'est lui qui est utilisé en écriture. La distribution des monotonies sur les supports doit être judicieuse pour éviter les phénomènes de blocage dont on a parlé dans l'introduction.

5.1. Principe du tri polyphasé

On appelle ***phase*** une suite de fusions où les supports gardent le même rôle : le support utilisé pour écrire reste le même. Une phase ne correspond pas à un passage sur la totalité des éléments à trier comme on le voit dans l'exemple suivant, où on a trois supports et huit monotonies au départ.

	Résultat des tris internes	Phase 1
$S1$	$M1\ M3\ M5\ M7\ M8$	$M7\ M8$
$S2$	$M2\ M4\ M6$	
$S3$		$M1+M2 \quad M3+M4 \quad M5+M6$

	Phase 2	Phase 3
$S1$		$M1+M2+M7+M5+M6$
$S2$	$M1+M2+M7 \quad M3+M4+M8$	$M3+M4+M8$
$S3$	$M5+M6$	

Figure 4. Exemple de tri polyphasé.

La phase suivante donne la liste triée sur le support $S3$. Dans le cas de dérouleurs de bandes on rembobine la bande réceptrice à la fin de chaque phase.

Les répartitions de départ qui permettent d'éviter les blocages sont basées sur les nombres de Fibonacci (cf. annexe).

Si on a trois supports et F_{n+1} monotonies, on répartit F_n monotonies sur un support, F_{n-1} sur un autre support, et le dernier support est utilisé en écriture comme indiqué dans la figure 5.

On finit par obtenir F_2 monotonies et F_1 monotonies sur deux supports, soit une monotonie sur chaque support, ce qui permet l'interclassement final.

Cette méthode se généralise à q supports pour q supérieur à 2. On va établir quelles doivent être les répartitions dans ce cas. En même temps on montrera un résultat d'optimalité.

Propriété : Le tri polyphasé permet d'interclasser un nombre maximum de monotonies en n phases.

Preuve : On se fixe n, le nombre de phases, et on cherche la stratégie d'interclassement (choix du support récepteur et du nombre de fusions pour chaque phase), et la répartition des monotonies, qui permettent d'interclasser un nombre maximum de monotonies en utilisant q supports externes. On va retrouver le tri polyphasé.

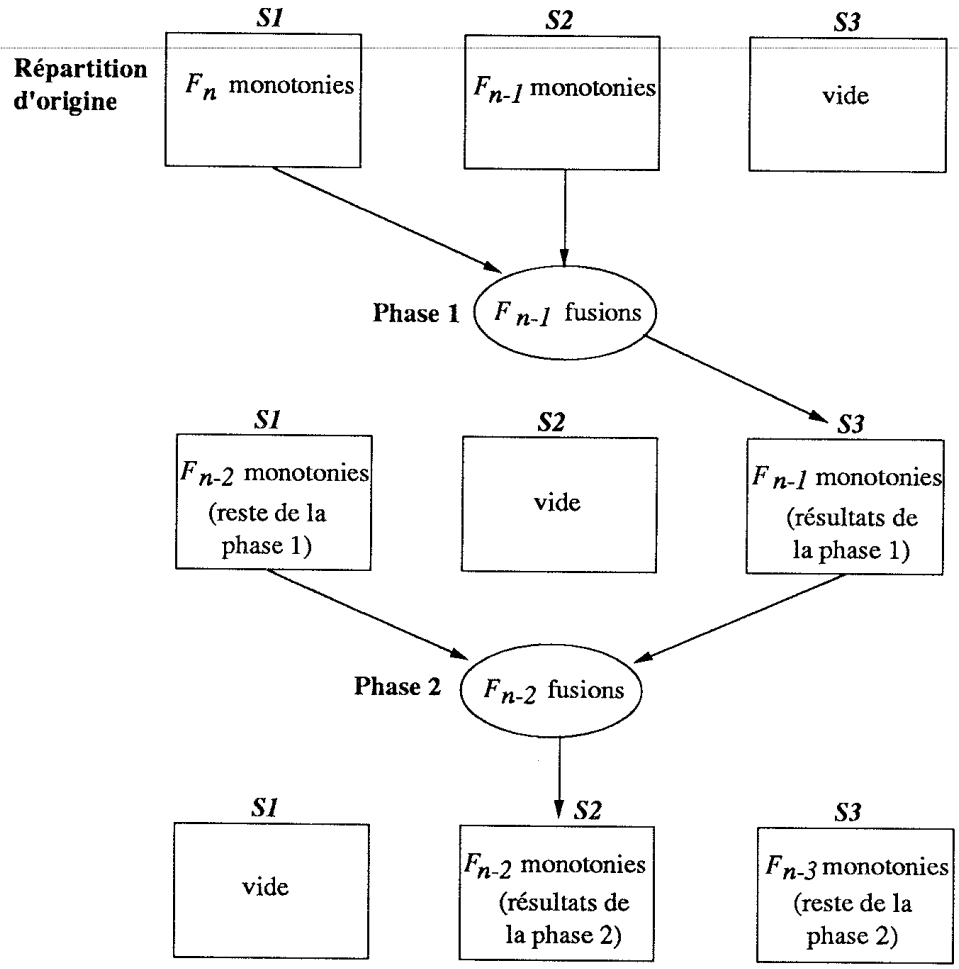


Figure 5. Tri polyphasé et nombres de Fibonacci.

1) Choix du support récepteur et du nombre de fusions à chaque phase

Soit $a_1, ..., a_q$ la répartition initiale : a_i est le nombre de monotonies sur le $i^{ième}$ support. On veut rendre maximum $\sum_{i=1}^{q} a_i$.

Soit $b_1, ..., b_q$ la répartition obtenue après la première phase.

On va supposer que pendant cette première phase, j est le numéro du récepteur; λ_i (pour $i \neq j$) est le nombre de monotonies lues sur le support numéro i et λ_j est le nombre de monotonies écrites sur le support numéro j.

On a par définition de la fusion :

$$b_j = a_j + \lambda_j \tag{1}$$
$$b_i = a_i - \lambda_i \quad \text{pour} \quad i \neq j \tag{2}$$

D'où

$$\sum_{k=1}^{q} a_k = (b_j - \lambda_j) + \sum_{\substack{i=1 \\ i \neq j}}^{q} (b_i + \lambda_i)$$
$$= \sum_{i=1}^{q} b_i + \sum_{\substack{i=1 \\ i \neq j}}^{q} \lambda_i - \lambda_j \tag{3}$$

Or on a par définition $\lambda_i \leq \lambda_j$ pour tout $i \neq j$, puisque toute monotonie lue intervient dans le résultat d'une fusion. On a aussi $\lambda_j \leq b_j$. D'où

$$\sum_{k=1}^{q} a_k \leq \sum_{i=1}^{q} b_i + (q-2)\lambda_j \leq \sum_{i=1}^{q} b_i + (q-2)b_j \tag{4}$$

On cherche à rendre maximum $\sum_{k=1}^{q} a_k$ donc à atteindre la borne supérieure donnée par (4).

Pour cela il faut exécuter la première phase de manière à avoir :

- $\lambda_i = \lambda_j$ pour tout $i \neq j$, ce qui implique d'interclasser toujours $q - 1$ monotonies, donc d'arrêter la phase dès qu'un support est vide;
- $\lambda_j = b_j$ ce qui implique que $a_j = 0$, donc que l'on écrit sur un support qui était vide au début de la phase.

On a retrouvé le fait qu'à tout instant on a $q-1$ supports en lecture et que dès qu'un support est vide, il devient le support d'écriture.

La borne supérieure donnée en (4) peut être rendue maximum en choisissant pour b_j la valeur maximum des $b_k (k = 1, ..., q)$ et pour $b_1, ..., b_q$ la répartition qui permet d'interclasser un nombre maximum de monotonies en $n - 1$ phases. Ces deux remarques vont être utilisées pour le calcul des répartitions optimales.

2) Calcul des répartitions optimales

On va calculer le nombre de monotonies sur les supports. Les permutations de supports ne sont pas significatives : c'est la même chose, pour ce qui est du nombre de monotonies fusionnées, d'avoir une monotonie sur $S1$ et $S2$, et zéro sur $S3$, ou une monotonie sur $S1$ et $S3$, et zéro sur $S2$. On ne fait donc pas de distinction entre

de telles répartitions, ce qui revient à ignorer les numéros de support et on note :

$$f_1^n \geq f_2^n \geq .... \geq f_q^n$$

les valeurs $a_1, ..., a_q$ triées. L'exposant n rappelle qu'on va effectuer n phases en tout.

De même, $f_1^i \geq ... \geq f_q^i$ est la répartition triée à partir de laquelle on peut interclasser un maximum de monotonies en i phases (les valeurs b_i obtenues après la première phase sont donc notées $f_1^{n-1} \geq ... \geq f_q^{n-1}$, lorsqu'elles sont triées).

D'après les calculs ci-dessus on a :

$$f_q^n = 0 \quad \text{puisque} \quad a_j = 0$$
$$f_1^{n-1} = b_j \quad \text{puisque} \quad b_j = \max_{i=1,...,q} b_i.$$

Par définition des a_i et b_i (cf. les relations (1) et (2)), on a pour $n > 1$:

$$f_q^n = 0$$
$$f_i^n = f_{i+1}^{n-1} + f_1^{n-1} \text{ pour } i = 1, ..., q-1.$$

En effet, à la fin de chaque étape, on a un maximum b_j de monotonies sur le support récepteur, qui était vide au départ, et on a retiré b_j monotonies de tous les autres supports. L'ordre des f_i^n est conservé pour $i < q$, mais les rangs sont décalés de 1 car b_j est la valeur maximum et de ce fait est notée f_1^{n-1} (cf. figure 6).

On a donc défini les f_i^n par récurrence sur n. Dans le cas d'une seule phase, on peut fusionner au maximum $q-1$ monotonies et en écrire une. La récurrence s'initialise donc de la manière suivante :

$$f_q^1 = 0$$
$$f_i^1 = 1 \text{ pour } i = 1, ..., q-1.$$

Dans le cas où $q = 3$, on retrouve les nombres de Fibonacci (voir exercices). On peut interclasser, dans le cas de trois supports, au plus F_{n+2} monotonies en n phases. □

Pour un nombre de supports supérieur à 3, le tableau des répartitions optimales se construit facilement. On donne à la figure 6 le début de ce tableau pour $q = 6$.

Lors de la création des monotonies, on les répartit sur les supports de manière à se rapprocher d'une répartition optimale. Puis on complète par des monotonies factices. L'algorithme de répartition est conçu de manière à équilibrer le nombre de monotonies factices sur chaque support (voir exercices).

Ces monotonies factices n'ont pas de réalité physique : on sait simplement par l'algorithme de répartition le nombre de monotonies factices pour chaque support.

Répartition / n	f_1^n	f_2^n	f_3^n	f_4^n	f_5^n	f_6^n	Total
0	1	0	0	0	0	0	1
1	1	1	1	1	1	0	5
2	2	2	2	2	1	0	9
3	4	4	4	3	2	0	17
4	8	8	7	6	4	0	33

Figure 6. Répartitions optimales pour 6 supports.

On peut donc organiser le programme de fusion de manière à ignorer un support quand une monotonie factice est en cause comme le montrent les exemples ci-dessous. Lors d'une fusion où intervient une monotonie factice on fait moins de comparaisons, ou même pas du tout dans le cas où toutes les monotonies sont factices sauf une (dans ce cas, il s'agit d'une recopie; cela se produit par exemple quand on a 5 + 1, ou 9 + 1, ou 17 + 1... monotonies dans le cas de six supports).

Pour avoir un maximum de fusions efficaces, il faut se ramener à une distribution sans monotonies factices le plus tôt possible, quand les monotonies ne sont pas trop grandes : pour cela, on considère que les monotonies factices sont les premières à interclasser ce qui est parfaitement possible puisqu'elles n'ont pas d'existence physique.

Exemple 1 : $q = 6$; 7 monotonies.

Les supports incomplets sont indiqués par un *.

Répartition initiale (se reporter à la figure 6 : il y a 2 monotonies factices) :

$M1, M6$	$M2, M7$	$M3^*$	$M4^*$	$M5$	–

Phase 1 : On interclasse seulement 3 monotonies venant de $S1, S2$ et $S5$ et on obtient :

$M6$	$M7$	$M3$	$M4$	–	$M1 + M2 + M5$

On retrouve le cas optimal pour $n = 1$.

Exemple 2 : $q = 6$; 12 monotonies.

Répartition initiale :

$M1, M6, M10^*$	$M2, M7, M11^*$	$M3, M8, M12^*$	$M4, M9^*$	$M5^*$	–

Phase 1 : On a une première fusion de monotonies factices où on ne touche pas aux supports externes; on note qu'on a une monotonie factice sur le support récepteur $S6$; on effectue ensuite une fusion normale et on obtient :

$M6, M10$	$M7, M11$	$M8, M12$	$M9$	_	$M1+M2+M3+M4+M5^*$

Phase 2 : On écrit sur $S5$, on interclasse seulement 4 monotonies prises sur $S1, S2, S3$ et $S4$ car sur $S6$ on a une monotonie factice. On obtient :

$M10$	$M11$	$M12$	_	$M6+M7+M8+M9$	$M1+M2+M3+M4+M5$

On retrouve le cas optimal pour $n = 1$.

5.2. Analyse du tri polyphasé

Le tri polyphasé permet de fusionner un maximum de monotonies en n phases d'interclassement. Il est intéressant d'étudier la complexité de ces opérations d'interclassement. Les opérations auxquelles on s'intéresse dans ce cas sont les entrées-sorties. Comme il y a autant de lectures que d'écritures au cours d'une fusion, on choisit de compter les écritures.

Le nombre d'écritures correspond à la somme, pour toutes les fusions qui sont faites, des tailles des monotonies créées.

On va se placer dans le cas suivant :

- il y a 3 supports;
- il y a N éléments à trier en tout;
- on a au départ F_{n+1} monotonies de taille égale, $t = \frac{N}{F_{n+1}}$.

On a vu que dans ce cas il y a $n-1$ phases. Au cours de la première on fait F_{n-1} fusions, au cours de la deuxième F_{n-2}, et au cours de la $i^{ième}$, F_{n-i}.

Au cours de la première phase, les monotonies résultantes sont de taille $2t$. Au cours de la $2^{ième}$ phase, on interclasse ces monotonies de taille $2t$ avec des monotonies de taille t, restant de la $1^{ère}$ phase, comme le montre le tableau ci-dessous où les tailles sont indiquées entre crochets.

Au démarrage de la $i^{ième}$ phase (c'est-à-dire à la fin de la $(i-1)^{ième}$ phase) on a sur un support F_{n-i} monotonies de $(t.F_i)$ éléments et sur un autre support F_{n-i+1} monotonies de $(t.F_{i+1})$ éléments.

Support	$S1$	$S2$	$S3$
Répartition initiale	$F_n[t]$	$F_{n-1}[t]$	0
Fin de $1^{ère}$ phase	$F_{n-2}[t]$	0	$F_{n-1}[2t]$
Fin de $2^{ème}$ phase	0	$F_{n-2}[3t]$	$F_{n-3}[2t]$
Fin de $3^{ème}$ phase etc.	$F_{n-3}[5t]$	$F_{n-4}[3t]$	0

La démonstration se fait par récurrence : la propriété est vraie pour la première phase car $F_1 = F_2 = 1$. Si elle est vraie pour la $i^{ième}$ phase, elle est vraie pour la $(i+1)^{ième}$ car on obtient F_{n-i} monotonies de taille $t.(F_i + F_{i+1}) = t.F_{i+2}$ (écritures de la $i^{ième}$ phase) et il reste F_{n-i-1} monotonies de taille $(t.F_{i+1})$.

Le nombre d'écritures de la $i^{ième}$ phase est donc : $F_{n-i}.(t.F_{i+2})$.

Or, on a $n-1$ phases. Le nombre total d'opérations est donc :

$$E = t.\sum_{i=1}^{n-1} F_{n-i}.F_{i+2}$$

Or, on sait que (voir exercices de l'annexe) :

$$\sum_{k=0}^{n} F_k.F_{n-k} = \frac{n-1}{5}.F_n + \frac{2n}{5}.F_{n-1}$$

Pour utiliser cette formule on réécrit E ainsi :

$$E = t.\sum_{i=1}^{n-1} F_{(n+2)-(i+2)}.F_{i+2}$$

Posons $k = i + 2$.

$$\begin{aligned} E &= t.\left(\sum_{k=0}^{n+2} F_k.F_{n+2-k} - F_{n+1}.F_1 - F_n.F_2 - 2.F_0.F_{n+2}\right) \\ &= t.\left(\frac{n-4}{5}.F_{n+2} + \frac{2n+4}{5}.F_{n+1}\right) \end{aligned}$$

Or, $t = \frac{N}{F_{n+1}}$. D'où :

$$E = \frac{N.(n-4)}{5F_{n+1}}F_{n+2} + \frac{N.(2n+4)}{5}$$

En utilisant le fait (voir annexe) que :

$$F_n = \frac{1}{\sqrt{5}}(\phi^n - \overline{\phi}^n)$$

on peut montrer que :

$$F_{n+2}/F_{n+1} = \phi.(1 + o(1))$$

et que

$$n + 1 = \frac{\text{Log } F_{n+1}}{\text{Log}\phi} + \frac{\text{Log } 5}{2\text{Log } \phi} + o(1)$$

D'où

$$E = N \text{ Log } F_{n+1}\left(\frac{2+\phi}{5 \text{ Log } \phi} + \frac{cte + o(1)}{\text{Log } F_{n+1}} + o(1)\right) = \Theta(N\text{Log } F_{n+1})$$

Rappelons que F_{n+1} est le nombre de monotonies obtenues à l'issue des tris internes. Si on note M ce nombre on peut dire que les fusions du tri polyphasé, dans le cas de trois supports, ont une complexité en $\Theta(N\log M)$ pour les entrées-sorties. Si le nombre de monotonies n'est pas égal à un nombre de Fibonacci, on a vu que l'on ajoute des monotonies fictives jusqu'à obtenir le nombre de Fibonacci immédiatement supérieur. Ces monotonies ne correspondent à aucune opération d'entrée-sortie. *On a donc une complexité en* $\Theta(N\log M)$ *entrées-sorties.* Cette complexité du tri polyphasé avec 3 supports est du même ordre que celle du tri équilibré avec 4 supports.

D'une manière générale un tri polyphasé avec q supports est comparable avec un tri équilibré de $2.(q-1)$ supports. Théoriquement, plus le nombre de supports augmente, plus il est intéressant de faire du tri polyphasé.

6. Les lectures en arrière

Cette technique est utilisée dans le cas de bandes magnétiques pour éviter des rembobinages. Lorsque l'on est à la fin de plusieurs bandes, on peut lire les monotonies en marche arrière et faire un interclassement dont les résultats sont en ordre inverse de l'ordre des monotonies d'origine.

Le principe de base est donc que l'on *lit en arrière* et que l'on *écrit en avant.*

6.1. Application au tri équilibré

On va noter C les monotonies croissantes et D les monotonies décroissantes. Si on part avec 8 monotonies on a les passages suivantes :

	Répartition d'origine	Résultat du passage 1	Résultat du passage 2	Résultat du passage 3
$B1$	$C1\ C3\ C5\ C7$		$C1 + C2 + C3 + C4$	
$B2$	$C2\ C4\ C6\ C8$		$C5 + C6 + C7 + C8$	
$B3$		$D1 + D2 \quad D5 + D6$		$D1 + ... + D8$
$B4$		$D3 + D4 \quad D7 + D8$		

On a un problème car le résultat est trié de manière décroissante. On pourrait envisager de produire les monotonies d'origine décroissantes. Mais cela implique de savoir au début du tri externe combien on va avoir de monotonies. On a vu que ce n'est en général pas le cas.

La seule solution est donc de prévoir, si c'est nécessaire, un rembobinage avant le dernier passage.

6.2. Application au tri polyphasé

Il faut noter que dans l'analyse du tri polyphasé on n'a pas pris en compte les temps de rembobinage dans le cas où les supports externes sont des bandes magnétiques. Or, ce temps est important et pour n phases, on a n rembobinages.

Cependant la mise en œuvre des lectures en arrière n'est pas immédiate. Considérons le cas où $q = 3$ et où on a 13 monotonies :

	Phase 1	Phase 2
$B1$	$8\ C \Downarrow$	$3\ C \Downarrow$
$B2$	$5\ C \Downarrow$	
$B3$		$5\ D \Downarrow$

A la fin de la deuxième phase, on doit fusionner des monotonies croissantes et décroissantes. Pour éviter ce phénomène, il faut alterner monotonies croissantes et décroissantes sur chaque bande.

Pour obtenir une liste croissante à la fin, la règle est de commencer une bande par une liste croissante et les autres par une liste décroissante. On montre qu'il faut que la bande commençant par une liste ascendante comporte un nombre impair de monotonies. Par définition des répartitions optimales, les autres ont alors un nombre pair de monotonies dans le cas où $q = 3$. Pour certaines valeurs de q, cette approche

élimine certaines répartitions, celles où tous les nombres sont pairs (voir exercices).

	$B1$	$B2$	$B3$
Répartition initiale	$DCDCDCDC \Downarrow$	$CDCDC \Downarrow$	
Résultat de la phase 1	$DCD \Downarrow$		$DCDCD \Downarrow$
Résultat de la phase 2		$CDC \Downarrow$	$DC \Downarrow$
Résultat de la phase 3	$DC \Downarrow$	$C \Downarrow$	
Résultat de la phase 4	$D \Downarrow$		$D \Downarrow$
Résultat final		$C \Downarrow$	

Cette méthode, certes optimale, est difficile à mettre en œuvre. Elle n'est utilisée en pratique que lorsqu'on dispose d'un générateur de tri qui effectue tous les calculs de répartition et pilote les répartitions et les fusions.

7. Conclusion

Il existe des programmes de tri polyphasé paramétrables par le format des données à trier et le nombre des supports externes. On les appelle des *générateurs de tri*. Dans le cas où on ne dispose pas d'un tel programme il faut savoir faire un choix entre le tri équilibré et le tri polyphasé. Le tri équilibré est moins efficace et nécessite au moins quatre supports, mais il a l'avantage d'être facile à mettre en œuvre.

Exercices

Dans tous les exercices, N désigne la taille de la liste à trier, et n la taille de la mémoire centrale.

1. Que donne la méthode de sélection et remplacement si la liste L à trier est déjà triée ? Si la liste L est triée en ordre décroissant ? Préciser le nombre de monotonies obtenues dans chaque cas, en fonction de N et de n.

2. Est-il vrai que la dernière monotonie produite par la méthode de sélection et remplacement ne contient jamais plus de n éléments ? Justifiez votre réponse.

3. Donner une condition nécessaire et suffisante pour que la méthode de sélection et remplacement produise exactement une monotonie.

4. a) Ecrire l'algorithme de sélection et remplacement dans le cas où on maintient en mémoire centrale les clés organisées en tas.

b) Quel est le pire des cas pour l'algorithme précédent, si on utilise un tas de taille n, pour produire des monotonies à partir d'une liste de taille N, avec $n < N$?

5. On appelle *sélection externe* la recherche du $k^{ième}$ élément dans une liste de N éléments ($k \leq N$), avec $N >> n$. Donner un algorithme qui réalise cette sélection externe.

6. Comparer les complexités au pire et en moyenne de l'algorithme d'interclassement donné à l'exercice 4 du chapitre 9 et de l'algorithme de fusion donné au paragraphe 3 de ce chapitre.

7. On veut effectuer *l'interclassement de p tableaux triés.* Ces p tableaux sont stockés dans une matrice rectangulaire T de dimension $[1..p, 1..m]$. La longueur de ces tableaux est donnée dans le tableau L de dimension $[1..p]$ dont tous les éléments

sont strictement inférieurs à m. Le tableau de numéro i comporte donc $L[i]$ éléments ; ces éléments sont suivis d'un élément supplémentaire différent et supérieur à tous les autres, noté *maxelt*. On a donc pour tout i dans $1..p$:

$$T[i, L[i] + 1] = maxelt.$$

a) Ecrire un programme qui effectue cet interclassement en utilisant un tableau de p compteurs C pour parcourir les p tableaux et un compteur k pour parcourir le tableau résultat R de dimension $[1..p * m]$.

b) Quelle est la complexité de cet algorithme ? Pourrait-on améliorer cette complexité et quels seraient les changements à apporter à cet algorithme ?

c) Le programme d'interclassement de deux tableaux donné à l'exercice 4 du chapitre 9 se généralise-t-il facilement à p tableaux ? Pourquoi et comment ?

8. Soit deux tableaux A et B contenant respectivement p et m éléments. A et B sont triés et p est beaucoup plus grand que m. On veut interclasser ces deux tableaux dans le tableau C.

a) Donner une version récursive de l'algorithme d'interclassement dichotomique *interdich*(A, B, C, p, m) qui utilise une recherche dichotomique pour placer les éléments de B par rapport à ceux de A.

b) Donner une version itérative de ce même algorithme.

c) Donner la complexité au pire et en moyenne de cet algorithme, en nombre de copies d'éléments, en fonction de m et p.

d) Calculer la complexité au pire, en nombre de comparaisons entre éléments, en fonction de m et p.

e) Que donne cet algorithme si $p = m$? Comparer avec l'algorithme d'interclassement vu au paragraphe 3.

9. On note $F(m, p)$ le nombre minimal de comparaisons nécessaires pour interclasser un tableau trié de m éléments avec un tableau trié de p éléments, en utilisant des comparaisons entre éléments.

a) Etablir la relation suivante : $\lceil \log C_{m+p}^{p} \rceil \leq F(m, p) \leq m + p - 1$.

(Indication : on pourra utiliser les arbres de décision.)

b) Montrer que $F(p, p) = 2p - 1$.

10. Soit deux tableaux triés $t1$ de taille p et $t2$ de taille m $(m \leq p)$. L'algorithme d'interclassement proposé au paragraphe 3 nécessite $m + p$ places mémoire supplémen-

taires. Montrer qu'en utilisant une zone auxiliaire au début de $t1$, seules m places mémoire supplémentaires sont nécessaires pour réaliser l'interclassement de $t1$ et de $t2$ (le contenu initial de $t1$ est détruit dans ce cas).

11. Montrer comment on peut utiliser une méthode de sélection et remplacement basée sur un tas de taille p, pour réaliser une fusion à p voies. Evaluer le gain de temps et discuter la pertinence de cette méthode.

12. Le tri équilibré se généralise au cas où on a un nombre quelconque de supports $q \geq 3$. On sépare les supports en deux groupes : un premier groupe de p supports, initialement en lecture ($1 \leq p < q$) et un deuxième groupe de $(q - p)$ supports, initialement en écriture. On répartit de façon aussi équilibrée que possible les monotonies initiales sur les p supports en lecture et de même, le nombre de monotonies sur les supports en écriture. Chaque groupe de supports est alternativement en lecture ou en écriture.

Décrire la suite des passages obtenue lorsqu'on effectue un tri équilibré généralisé de 5 monotonies de taille 1000, avec $q = 3$ et $p = 2$.

On appelle ce tri, ***tri équilibré à (p,q − p) voies***.

13. a) Montrer qu'avec a monotonies de départ le tri équilibré à $(p, q - p)$ voies fait (cf. exercice précédent) :

- $2.k$ passages si $p^k.(q-p)^{k-1} < a \leq p^k.(q-p)^k$
- $2.k + 1$ passages si $p^k.(q-p)^k < a \leq p^{k+1}(q-p)^k$

b) Exprimer exactement le nombre de passages, en fonction de a, lorsque $q = 2p$, et lorsque $q = 2p - 1$.

c) Pour quelle valeur de p, $1 \leq p < q, p.(q - p)$ est-il maximal ? En déduire la valeur optimale de p pour minimiser le nombre de passages sur un nombre de monotonies fixé, avec un tri équilibré à $(p, q - p)$ voies.

14. Combien de phases comporte le tri polyphasé à 5 supports, si les 4 supports en lecture contiennent respectivement 26, 15, 22 et 28 monotonies au départ ?

15. Montrer que les quantités f_1^n et f_2^n introduites pour le calcul des répartitions optimales du tri polyphasé (au §5), sont respectivement égales à F_{n+1} et F_n.

16. Ecrire un algorithme de répartition pour le tri polyphasé qui équilibre le nombre de monotonies factices pour chaque support.

17. Dans le cas du tri polyphasé avec q supports, montrer que les répartitions optimales correspondant aux nombres de Fibonacci vérifient la propriété suivante :

(i) si la dernière monotonie obtenue à la fin des fusions se trouve sur le support S_q (celui qui était libre au départ), alors pour tout i, $1 \leq i < q$, le nombre initial de monotonies sur S_i est impair;

(ii) si la dernière monotonie obtenue se trouve sur un support S_j avec $j \neq q$, alors pour tout i, $1 \leq i < q$ et $i \neq j$, le nombre initial de monotonies sur S_i est pair, et le nombre initial de monotonies sur S_j est impair.

18. On suppose qu'on dispose d'une répartition optimale pour le tri polyphasé avec 6 supports telle que S_1 contient un nombre impair de monotonies. On applique la méthode de lecture en arrière en alternant monotonies croissantes et décroissantes. Montrer que si S_1 comporte $CDCD...$ au départ et S_j $(1 < j \leq 6)$ comporte $DCDC...$, alors le tri polyphasé avec lecture en arrière se termine avec une seule monotonie sur S_1, et qu'elle est croissante.

(Indication : on pourra utiliser l'exercice n° 17.)

Lectures conseillées pour le chapitre 17

Knuth, *The Art of Computer Programming 3*, Vol. 3 : *Sorting and Searching*, Addison-Wesley, 1973.

Purdom & Brown, *The Analysis of Algorithms*, Holt, Rinehart & Winston, 1985.

Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, 1976.

Cinquième partie

Quelques algorithmes sur les graphes

On a vu au chapitre 8 que les problèmes qui se formulent en termes d'objets et de connexions (ou relations) entre ces objets se modélisent naturellement par des graphes, orientés ou non.

Les algorithmes sur les graphes sont extrêmement nombreux et peuvent remplir plusieurs livres à eux seuls (on trouvera quelques références à la fin de cette introduction). Il n'est donc pas possible de prétendre ici à un traitement exhaustif de ce sujet : on se limite à des problèmes généraux (ils interviennent dans de nombreux autres problèmes) pour lesquels on connaît des algorithmes utilisables.

> *En effet, il existe de nombreux problèmes sur les graphes pour lesquels on ne connaît pas d'algorithmes de complexité acceptable. Un exemple est le problème du voyageur de commerce : il s'agit de trouver dans un graphe non orienté, valué, un cycle de coût minimum qui passe par tous les sommets une fois et une seule. Un tel cycle est appelé hamiltonien. Les algorithmes exacts connus effectuent la construction de quasiment tous les cycles du graphe. Ces méthodes sont d'une complexité inacceptable : dans le cas où le graphe a n sommets et est complet, on a* $(n-1)!/2$ *cycles à construire. Mais on connaît des heuristiques pour ce problème, de complexités acceptables, qui donnent dans la majorité des cas des solutions pas trop mauvaises (c'est-à-dire, pas trop loin de l'optimum). Ces heuristiques ne sont pas étudiées ici car leur présentation impliquerait une définition rigoureuse des concepts de solution approchée et d'optimalité locale, qui sortent du cadre de ce livre.*

On va donc présenter, en utilisant la terminologie et les résultats du chapitre 8, des algorithmes considérés comme «classiques», qui permettent de traiter les problèmes suivants : effectuer un *tri topologique*; déterminer les *composantes connexes, fortement connexes* et *deux-connexes* d'un graphe; trouver un *plus court chemin* dans un graphe orienté valué; rechercher un *arbre de recouvrement minimum* dans un graphe non orienté valué. On évalue la complexité au pire de ces algorithmes en fonction de deux paramètres, le nombre d'arcs (ou d'arêtes) et le nombre de sommets du graphe.

Un des buts de cette partie est de mettre en évidence la difficulté à concevoir des algorithmes efficaces sur les graphes. Elle donne des notions de base et devrait persuader le lecteur que lorsqu'il est confronté à un problème non traité ici, il est raisonnable de rechercher dans la littérature si un algorithme (ou une heuristique) existe déjà. Attention, les complexités exponentielles guettent les inventeurs d'algorithmes sur les graphes !

Lectures conseillées pour les graphes

Aho, Hopcroft & Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983.

Aho, Hopcroft & Ullman, *The design and Analysis of Computer Algorithms*, Addison-Wesley, 1979.

Berge, *Graphes*, Gauthier-Villars, 1983.

Gondran & Minoux, *Graphes et algorithmes*, Eyrolles, 1979.

Sakarovitch, *Optimisation combinatoire - Programmation discrète*, Hermann, 1984.

Chapitre 18
Tri topologique

Dans ce chapitre, on considère des graphes orientés.

1. Spécification informelle

On a vu au chapitre 8, à la figure 3, un exemple de graphe décrivant des tâches à accomplir, avec des contraintes d'ordre sur ces tâches : les sommets représentent les tâches et s'il existe un arc de t_i à t_j, la tâche t_i doit être exécutée avant la tâche t_j.

L'opération qui consiste à trouver un ordre sur les sommets s'appelle un *tri topologique* : il s'agit de construire une liste de tous les sommets d'un graphe dans laquelle aucun sommet n'apparaît avant un de ses prédécesseurs.

Il y a de nombreuses autres situations où l'on est amené à effectuer un tri topologique sur un graphe : par exemple, les sommets du graphe peuvent représenter des définitions de mots dans un dictionnaire, qui s'utilisent les unes les autres et qui ne sont pas circulaires; ou bien des unités d'enseignement telles que certaines unités utilisent des résultats enseignés dans d'autres (prérequis). Dans les systèmes d'exploitation, la gestion de ressources utilise également un tri topologique.

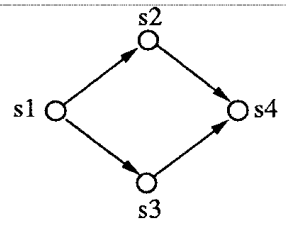
Le tri topologique n'a de solution que si le graphe considéré est sans circuit : il est impossible d'ordonner les sommets d'un circuit de manière à ce que tout sommet apparaisse après son prédécesseur dans le circuit ! Si le graphe considéré est sans circuit, le tri topologique a souvent plusieurs solutions, car on peut avoir à choisir entre plusieurs sommets dont tous les prédécesseurs sont déjà placés (voir figure 1).

Un graphe orienté sans circuit sert souvent de représentation pour la relation d'ordre partiel $\leq$ sur les sommets, fondée sur la relation arc, qui vérifie la propriété suivante :

$$\text{il existe un arc de } x \text{ vers } y \Rightarrow x \leq y$$

Le tri topologique doit son nom au fait qu'il trie les sommets du graphe en respectant cette relation d'ordre. Si l'on obtient plusieurs listes solutions pour un même graphe,

c'est qu'il existe des sommets non comparables pour la relation $\leq$. Par exemple, les sommets $s2$ et $s3$ dans le graphe de la figure 1 sont non comparables.



L1 = <s1, s2, s3, s4>
L2 = <s1, s3, s2, s4>

Figure 1. Tris topologiques $L1$ et $L2$ d'un même graphe.

Il est parfois intéressant d'effectuer un tri topologique inverse : alors, dans la liste résultat tout sommet apparaît avant ses prédécesseurs (et donc tout sommet apparaît après tous ses successeurs) ; cette liste est équivalente au résultat d'un tri topologique sur un graphe dont on a inversé tous les arcs. On présente ici des algorithmes qui effectuent soit un tri topologique, soit un tri topologique inverse.

2. Spécification formelle

On utilise la spécification des graphes orientés qui a été donnée au chapitre 8.

Un tri topologique est une opération qui, étant donné un graphe orienté, rend une liste de sommets. On a donc :

tritopo : Graphe $\rightarrow$ Liste,

où Liste correspond à la spécification vue au chapitre 5, où l'on a remplacé partout Elément par Sommet.

L'opération de tri topologique n'est définie que sur les graphes sans circuit. Quand elle est définie, cette opération est caractérisée par les propriétés suivantes :

tritopo(*graphe-vide*) = *liste-vide*

tritop(G) = *cons*(s, l) $\Rightarrow$

$G \neq$ *graphe-vide* & d^{o-} *de* s *dans* $G = 0$

& l = *tritopo*(*retirer-le-sommet* s *de* G)

Il est important de noter le sens de l'implication dans le deuxième axiome : cet axiome exprime le fait qu'il y a plusieurs solutions au tri topologique d'un graphe. Une liste est une solution acceptable, si le premier sommet est un sommet de demi-degré intérieur nul, et si le reste de la liste est le résultat d'un tri topologique du

graphe sans ce sommet. Rappelons que l'opération *retirer-le-sommet* supprime le sommet en argument et retire tous les arcs dont ce sommet est l'origine.

3. Algorithmes de tri topologique

La spécification récursive qui vient d'être donnée pourrait se programmer directement en construisant, à partir du graphe d'origine, une suite de graphes obtenue par suppressions successives d'un sommet de demi-degré intérieur nul. En fait, on n'a pas besoin de construire complètement ces graphes : on a seulement besoin de connaître, à chaque étape de l'algorithme, les sommets qui restent et leur demi-degré intérieur (qui a évolué en fonction des suppressions de sommets).

On associe donc à ces graphes successifs un tableau, noté d, qui contient pour chaque sommet du graphe d'origine :

-1 si le sommet a déjà été supprimé
et sinon son demi-degré intérieur mis à jour au fur et à mesure des suppressions.

On utilise les opérations des types abstraits Liste et Ensemble. Comme on manipule des sommets, on considère que Elément a été remplacé par Sommet dans ces deux types. On utilise aussi une opération *choisir* qui choisit dans un ensemble un élément. On a l'algorithme général suivant :

```
procedure tritopo(G : Graphe; var L : Liste);
{le graphe G, contenant n sommets, est supposé sans circuit; après
exécution de la procédure, L est la liste résultat du tri topologique sur G}
var i, s, v, w : integer; d : array [1..n] of integer; M : Ensemble;
{s, v et w sont des sommets}
begin
    {initialisations :}
    M := ensemble-vide; L := liste-vide;
    for s := 1 to n do begin
        i := d°⁻ de s dans G;
        if i = 0 then M :=ajouter(s, M);
        d[s] := i
    end;
    {M contient les sommets sans prédécesseurs}
    i := 1;
    {construction de la liste L}
    while M <> ensemble-vide do begin
        v := choisir(M);
        M := supprimer(v, M); d[v] := -1;
        L := insérer(L, i, v);
        i := i + 1;
        {mise à jour du tableau d et de l'ensemble M}
```

```
                for j := 1 to d^o+ de v dans G do begin
                    w := j ème-succ-de v dans G;
                    d[w] := d[w] - 1;
                    if d[w] = 0 then M := ajouter(w, M)
                end
            end
        end tritopo;
```

Cet algorithme se justifie de la façon suivante. A chaque étape, M contient les sommets w tels que $d[w]$ est nul, c'est-à-dire, les sommets non encore placés dans L dont tous les prédécesseurs ont été placés dans L. Montrons par l'absurde que lorsque M est vide, tous les sommets ont été placés dans L. Supposons donc que l'ensemble M soit vide et que la longueur de la liste L soit strictement inférieure à n. Il existe alors un sommet y_0 non placé dans L tel que $d[y_0]$ soit strictement positif. Le sommet y_0 admet donc un prédécesseur y_1 qui n'est pas placé dans L. Comme M est vide, y_1 n'appartient pas à M et donc $d[y_1]$ est strictement positif. On peut continuer ainsi et construire une suite $y_0, y_1, ..., y_i, y_{i+1}, ...$ telle que y_i n'est pas placé dans L et y_{i+1} est un prédécesseur de y_i non placé dans L. Comme le graphe est sans circuit, cette suite est infinie. On obtient une contradiction avec le fait que l'ensemble S des sommets du graphe est fini. En conclusion, si M est vide, la liste L contient les n sommets. De plus, comme tout sommet est placé dans L après ses prédécesseurs, on obtient bien un tri topologique sur G.

Selon la façon dont on représente le graphe G et l'ensemble M, on obtient des algorithmes différents.

3.1. Graphe représenté par une matrice d'adjacence

3.1.1. Algorithme

Dans le cas où le graphe est représenté par une matrice d'adjacence G, on construit un tableau d des demi-degrés intérieurs de chaque sommet, à partir de cette matrice. Puis on choisit n fois un sommet v de demi-degré intérieur nul, en modifiant à chaque fois le tableau d en conséquence. A chaque étape, l'ensemble M est représenté par tous les sommets v tels que $d[v]$ est nul. L'algorithme utilise les déclarations de types suivantes :

```
type GRAPHE = array [1..n, 1..n] of boolean;
     LISTE = array [1..n] of integer;
```

On obtient la procédure suivante :

```
procedure tritopo1(G : GRAPHE; var L : LISTE);
{le graphe G, contenant n sommets, est supposé sans circuit; après
exécution de la procédure, L est la liste résultat du tri topologique sur G}
var d : array [1..n] of integer; i, j, k : integer;
begin
    {construction du tableau des demi-degrés intérieurs}
    for i : = 1 to n do begin
        d[i] := 0;
        for j := 1 to n do if G[j, i] then d[i] := d[i] + 1
    end;
    {construction de la liste résultat}
    for i := 1 to n do begin
        j := 1;
        while d[j] <> 0 do j := j + 1;
        L[i] := j; d[j] := -1;
        for k := 1 to n do if G[j, k] then d[k] := d[k] - 1
    end
end tritopo1;
```

Cet algorithme n'est correct que si le graphe G est sans circuit (sinon il y a débordement d'indice du tableau d). Il peut être modifié de manière à détecter l'existence d'un circuit (voir exercices).

3.1.2. Exemple

Considérons le graphe de la figure 2.

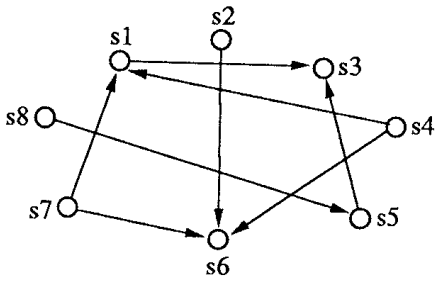


Figure 2. Graphe orienté sans circuit.

Le tableau des demi-degrés intérieurs d est :

d :	2	0	2	0	1	3	0	0
	1	2	3	4	5	6	7	8

Les deux premières étapes de l'exécution de la procédure *tritopo1* donnent :

1) $L[1] = 2$:

d :	2	−1	2	0	1	2	0	0
	1	2	3	4	5	6	7	8

2) $L[2] = 4$:

d :	1	−1	2	−1	1	1	0	0
	1	2	3	4	5	6	7	8

La liste résultat L est :

L :	2	4	7	1	6	8	5	3
	1	2	3	4	5	6	7	8

3.1.3. Analyse de la complexité

Cet algorithme effectue dans tous les cas n^2 tests d'éléments de G pour construire le tableau d des demi-degrés intérieurs et autant pour le mettre à jour. La construction de la liste effectue en tout $\sum_{i=1,\ldots,n} i = n(n+1)/2$ tests du tableau d car chaque sommet j est choisi une fois et une seule, et on a $(n+p)$ affectations dans d, où p est le nombre d'arcs de G. Cet algorithme est donc en $\Theta(n^2)$ opérations.

On va montrer que cette complexité peut être améliorée si on choisit une autre représentation du graphe et si on gère une liste des sommets de demi-degré intérieur nul.

3.2. Graphe représenté par des listes d'adjacence

3.2.1. Algorithme

Rappelons les types Pascal correspondant à la représentation d'un graphe par listes d'adjacence :

```
type adr = ↑ doublet;
     doublet = record no : 1..n; suiv : adr end;
     GRAPHE = array [1..n] of adr;
```

On utilise aussi le type LISTE défini au paragraphe précédent.

Dans l'algorithme suivant, on se sert du tableau d pour chaîner l'ensemble des sommets de demi-degré intérieur nul, c'est-à-dire, les sommets qui appartiennent à l'ensemble M.

```
procedure tritopo2(G : GRAPHE; var L : LISTE);
{le graphe G, contenant n sommets, est supposé sans circuit; après
exécution de la procédure, L est la liste résultat du tri topologique sur G}
var d : array [1..n] of integer; succ : adr; i, nul : integer;
begin
{construction du tableau des demi-degrés intérieurs :}
    for i := 1 to n do d[i] := 0;
    for i : = to n do begin
            succ := G[i];
            while succ <> nil do begin
                d[succ↑.no] := d[succ↑.no] + 1; succ := succ↑.suiv
            end
    end;
{chaînage des indices i tels que d[i] = 0; nul indique le début du chaînage :}
    nul := 0;
    for i := 1 to n do
            if d[i] = 0 then begin d[i] := nul; nul := i end;
{construction de la liste résultat :}
    i := 1;
    while nul <> 0 do begin
            L[i] := nul; nul := d[nul]; d[L[i]] := -1; succ := G[L[i]]; i := i + 1;
    {parcours des successeurs du sommet nul et mise à jour du tableau d :}
            while succ<> nil do begin
                d[succ↑.no] := d[succ↑.no] - 1;
                if d[succ↑.no] = 0 then begin
                        {on insère ce sommet en début de chaînage}
                        d[succ↑.no] := nul; nul := succ↑.no
                end;
                succ := succ↑.suiv
            end
    end
end tritopo2;
```

3.2.2. Exemple

Considérons le graphe de la figure 2. Rappelons que l'on considère que les successeurs d'un sommet sont choisis dans l'ordre des numéros croissants. Initialement on a le tableau d suivant :

d :	2	0	2	0	1	3	0	0
	1	2	3	4	5	6	7	8

Lors de la construction du chaînage, *nul* prend successivement les valeurs 0, 2, 4, 7 et 8, et à la fin de cette construction, le tableau d a pour contenu :

d :	2	**0**	2	**2**	1	3	**4**	**7**
	1	2	3	4	5	6	7	8

et *nul* vaut 8.

La variable *nul* indique le numéro d'un sommet de demi-degré intérieur nul; $d[nul]$ le numéro d'un autre tel sommet et ainsi de suite jusqu'à la valeur 0 qui indique la fin du chaînage.

Les deux premières étapes de l'exécution de la procédure sur le graphe de la figure 2 donnent les valeurs suivantes :

1) $L[1] = 8$:

d :	2	**0**	2	**2**	**7**	3	**4**	-1
	1	2	3	4	5	6	7	8

nul = 5

2) $L[2] = 5$:

d :	2	**0**	1	**2**	-1	3	**4**	-1
	1	2	3	4	5	6	7	8

nul = 7

On obtient la liste résultat suivante :

L :	8	5	7	4	1	3	2	6
	1	2	3	4	5	6	7	8

3.2.3. Analyse de la complexité

Cet algorithme est plus efficace que le précédent. La construction du tableau d demande $2n + 2\sum_{i=1,\ldots,n} d^{o+}(i)$ affectations, soit $2(n+p)$ affectations, où p est le nombre d'arcs de G.

Le tri topologique proprement dit provoque n suppressions du premier élément de la liste chaînée dans d, suivies du traitement des successeurs du sommet sélectionné; pour l'ensemble du tri, on a p tels traitements et donc p mises à jour du tableau d. On a donc $\Theta(n+p)$ opérations en tout, ce qui donne une complexité meilleure que celle de l'algorithme précédent si $p << n^2$.

Remarque : On a obtenu aux §3.2 et 3.3 des solutions différentes pour le tri topologique effectué sur un même graphe. Cela provient de la façon dont agit l'opération *choisir* dans l'ensemble des sommets sans prédécesseurs. Suivant la programmation de cette opération, (elle sélectionne le sommet de plus grand numéro, ou de plus petit numéro, ou le dernier sommet ajouté...), on obtient des listes différentes.

4. Algorithme de tri topologique inverse

4.1. Algorithme

On a vu au chapitre 8, un algorithme de parcours en profondeur d'un graphe. Dans un tel parcours, si on insère dans une liste, chaque sommet après le parcours de tous ses successeurs, on obtient la liste des sommets en ordre topologique inverse : un sommet n'apparaît qu'après tous ses successeurs. L'algorithme de tri topologique inverse revient à construire une liste des sommets de la forêt couvrante du parcours en profondeur en ordre suffixe.

```
procedure tritopo3(G : Graphe; var L : Liste);
{G est un graphe sans circuit contenant n sommets; après exécution de la
procédure, la liste L contient les sommets en ordre suffixe}
var marque : array [1..n] of boolean; i, h : integer;
    procedure parcours(x : integer; G : Graphe;
                       var M : array [1..n] of boolean;
                       var L : Liste; var h : integer);
    {x est un sommet}
    var j, y : integer;
    {y est un sommet}
    begin
        M[x] := true;
        for j := 1 to d^o+ de x dans G do begin
            y := j ème-succ-de x dans G;
            if not M[y] then parcours(y, G, M, L, h)
        end;
        {traitement du sommet x à la dernière rencontre :}
        L := insérer(L, h, x); h := h + 1
    end parcours;
begin
    for i := 1 to n do marque[i] := false;
    h := 1; L := liste-vide;
    {construction de L}
    for i := 1 to n do if not marque[i] then parcours(i, G, marque, L, h)
end tritopo3;
```

La complexité de cet algorithme est exactement celle du parcours en profondeur : on compte le nombre de tests du tableau *marque*, ce test étant l'opération qui est effectuée le plus souvent dans l'algorithme. On a donc une complexité en $\Theta(n^2)$ si le graphe est représenté par une matrice d'adjacence et en $\Theta(\max(n,p))$ si le graphe est représenté par des listes d'adjacence.

4.2. Exemple

Le tri topologique inverse sur le graphe de la figure 3 produit la liste de sommets $L = < s4, s5, s2, s3, s1, s7, s6 >$.

On a pris la convention que les sommets et les successeurs d'un sommet sont choisis par ordre de numéros croissants.

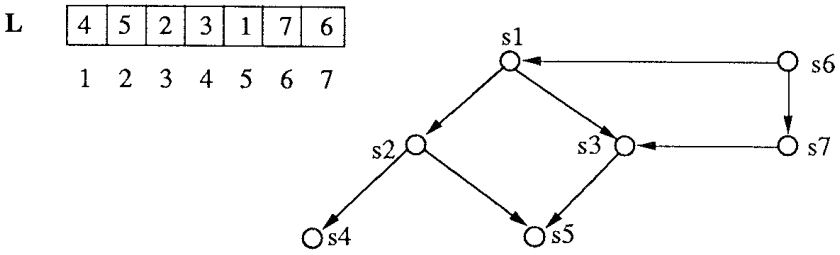


Figure 3. Tri topologique inverse sur un graphe sans circuit.

4.3. Justification

Le parcours en profondeur est toujours possible sur un graphe orienté G qu'il y ait ou non un circuit dans G. On a vu au chapitre 8 que ce parcours détermine une forêt couvrante de G. On va montrer que dans le cas d'un graphe sans circuit, le parcours en ordre suffixe de la forêt couvrante réalise un tri topologique inverse. Pour cela, montrons que si un sommet x apparaît avant un sommet y dans la liste L, alors il n'y a pas de chemin de x vers y.

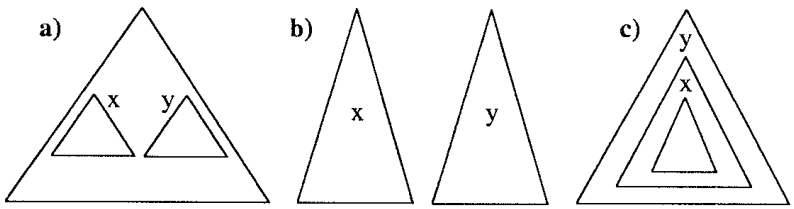


Figure 4. y apparaît après x dans le parcours suffixe de la forêt couvrante de G.

La preuve résulte de l'examen de la forêt couvrante associée. Si x apparaît avant y dans la liste L, c'est que le parcours de x et de tous ses descendants se termine avant le parcours de y et de tous ses descendants. Plusieurs cas sont possibles :

1) x et y sont racines de deux sous-arborescences distinctes d'une même arborescence (cf. figure 4.a) ou sont éléments de deux arborescences de la forêt couvrante (cf. figure 4.b) : il ne peut y avoir de chemin de x vers y, puisque le graphe G est sans circuit. Notons qu'il peut exister un chemin de y vers x car les arcs croisés sont de la droite vers la gauche.

2) x appartient à une sous-arborescence de racine y (cf. figure 4.c) : dans ce cas, il existe un chemin de y vers x par construction de la forêt couvrante. Comme le graphe est sans circuit, il n'y a pas de chemin de x vers y.

Exercices

1. Modifier la procédure *tritopo1* de manière à ce qu'elle détecte l'existence d'un circuit dans le graphe, s'il y en a un, et sinon réalise le tri topologique du graphe.

2. Faire tourner l'algorithme *tritopo2* sur le graphe de la figure 2.

3. a) Pourquoi ne peut-on pas utiliser le parcours en ordre préfixe de la forêt couvrante pour faire un tri topologique?

b) Peut-on utiliser le parcours en largeur (cf. la procédure *larg* au chapitre 8, §5.5) pour effectuer un tri topologique?

Chapitre 19
Connexités

La vérification de la connexité d'un réseau électrique ou d'un réseau téléphonique est un problème pratique majeur : on veut savoir si deux points sont reliés entre eux sans vouloir connaître comment. Exprimé en termes de graphes non orientés, ce problème consiste à déterminer des composantes connexes : tous les sommets d'une même composante sont mutuellement accessibles et deux sommets de deux composantes connexes distinctes ne peuvent être joints par aucune chaîne.

Plus précisément, on a vu au chapitre 8 qu'on appelle *composante connexe* d'un graphe non orienté G, un sous-graphe G' connexe maximal de G, c'est-à-dire, tel qu'aucun autre sous-graphe connexe de G ne contienne strictement G'.

Dans le cas de graphes orientés, on a recours au concept de forte connexité. Rappelons qu'un graphe orienté G est *fortement connexe* si pour tous sommets x, y de G, il existe un chemin de x vers y. On définit alors une *composante fortement connexe* d'un graphe orienté G, comme un sous-graphe fortement connexe maximal de G.

De plus, on peut s'intéresser à des propriétés plus fortes. Dans le cas d'un réseau, on peut vouloir qu'il reste connexe, même si l'un des points est défaillant. En termes de graphes non orientés, cette propriété s'appelle *2-connexité*.

Dans ce chapitre on donne d'abord des algorithmes qui permettent de trouver les composantes connexes d'un graphe non orienté, y compris quand le graphe évolue, puis on se place dans le cadre des graphes orientés et on présente deux algorithmes qui donnent les composantes fortement connexes. On donne ensuite la définition de la 2-connexité et un algorithme qui teste la 2-connexité.

1. Composantes connexes

1.1. Cas statique

On a déjà rencontré au chapitre 8 deux algorithmes qui donnent les composantes connexes d'un graphe non orienté G.

1.1.1. Le parcours en profondeur

L'appel de la procédure *prof*, pour un sommet donné s, est suivi d'appels récursifs de la même procédure qui se terminent lorsque tous les sommets accessibles depuis s sont marqués, et eux seulement. A l'exécution de cette procédure, on associe une arborescence couvrante de racine s. Les composantes connexes sont obtenues en considérant tous les sous-graphes engendrés par les sommets des arborescences de la forêt couvrante associée au parcours en profondeur du graphe tout entier.

1.1.2. Le parcours en largeur

Comme la procédure *prof*, la procédure *larg*, appelée pour un sommet s, marque tous les sommets qui peuvent être atteints depuis s. Les sous-graphes engendrés par les sommets des arborescences de la forêt couvrante associée au parcours en largeur de tout le graphe, donnent les composantes connexes.

1.1.3. La fermeture transitive

On donne un moyen de vérifier si un graphe est connexe, en utilisant sa fermeture transitive.

On appelle *fermeture transitive* d'un graphe non orienté $G = < S, A >$ le graphe $G^* = < S, A^* >$ tel que pour tous sommets x et y de S, il existe une arête entre x et y dans G^* si, et seulement si, il existe une chaîne entre x et y dans G.

Un graphe G est donc connexe si, et seulement si, sa fermeture transitive G^* est un graphe complet. (On rappelle qu'un graphe non orienté est complet si, et seulement si, il existe une arête entre deux sommets quelconques.) De même, deux sommets x et y sont dans la même composante connexe de G si, et seulement si, il existe une arête entre x et y dans la fermeture transitive G^* de G.

Spécification formelle

On peut construire la fermeture transitive par itérations successives. Pour savoir s'il existe une chaîne entre x et y, on regarde d'abord les chaînes entre x et y qui passent par $s1$, puis les chaînes entre x et y qui passent par $s1$ et $s2$, etc., et finalement les chaînes qui passent par tous les sommets du graphe.

Dans le cas où les sommets sont les entiers 1, 2, ..., n, l'opération de fermeture transitive se spécifie à l'aide de l'opération *itérer*. Les profils des opérations sont les suivants :

fermeture-transitive : Graphe $\rightarrow$ Graphe
itérer : Entier $\times$ Graphe $\rightarrow$ Graphe

Ces opérations sont partout définies et vérifient les axiomes suivants :

pour tout graphe G,
$\quad$ *itérer*$(0, G) = G$

pour tout entier $i > 0$, pour tous sommets s et s' et pour tout graphe G,
$\quad$ *{un graphe G a même ensemble de sommets que sa fermeture transitive}*
$\quad$ s *est-un-sommet-de itérer*$(i, G) = s$ *est-un-sommet-de itérer*$(i-1, G)$

$\quad$ $< s, s' >$ *est-une-arête-de itérer*$(i, G) =$
$\qquad\qquad < s, s' >$ *est-une-arête-de itérer*$(i-1, G) \vee$
$\qquad\qquad\qquad (< s, i >$ *est-une-arête-de itérer*$(i-1, G) \wedge$
$\qquad\qquad\qquad < i, s' >$ *est-une-arête-de itérer*$(i-1, G))$

$\quad$ $< s, s' >$ *est-une-arête-de fermeture-transitive*$(G) =$
$\qquad\qquad < s, s' >$ *est-une-arête-de itérer*$($*nb-sommets*$(G), G)$

Algorithme

Soit C la matrice d'adjacence de G (matrice booléenne). On peut calculer la matrice d'adjacence C^* de G^* en suivant la spécification. Supposons que $C_k[i, j]$ représente l'existence d'une chaîne de i à j passant par des sommets inférieurs ou égaux à k. Il existe une chaîne de i à j passant seulement par des sommets inférieurs ou égaux à k si, soit il existe une chaîne de i à j passant seulement par des sommets inférieurs ou égaux à $k-1$, soit il existe une chaîne de i à k passant par des sommets inférieurs ou égaux à $k-1$ ***et*** une chaîne de k à j passant par des sommets inférieurs ou égaux à $k-1$. On a donc :

$$C_k[i, j] = C_{k-1}[i, j] \quad \textbf{or} \quad (C_{k-1}[i, k] \quad \textbf{and} \quad C_{k-1}[k, j]).$$

En fait, on n'a besoin que d'une seule matrice A pour calculer les contenus successifs C_k de C, car $C_{k-1}[i, k] = C_k[i, k]$ et $C_{k-1}[k, j] = C_k[k, j]$. On obtient l'algorithme suivant, dit de ***Warshall*** :

```
procedure Warshall(C : array[1..n, 1..n] of boolean;
                   var A : array[1..n, 1..n] of boolean );
{Cette procédure calcule dans A la fermeture transitive de C}
var i, j, k : integer;
begin
   for i := 1 to n do
       for j := 1 to n do A[i, j] := C[i, j];
   for k := 1 to n do
       for i := 1 to n do
           for j := 1 to n do
               A[i, j] := A[i, j] or (A[i, k] and A[k, j])
end Warshall;
```

Il est clair que la complexité de l'algorithme de Warshall en nombre total d'opérations est en $\Theta(n^3)$.

Pour tester si deux sommets i et j sont dans la même composante connexe, il suffit de regarder si $A[i,j]$ = true, ce qui se fait en temps constant. Cependant, vu le coût de la construction de la matrice A, il est plus avantageux d'utiliser les parcours (en profondeur ou en largeur), pour trouver les composantes connexes. L'algorithme de Warshall est surtout utilisé lorsqu'on veut construire effectivement la fermeture transitive d'un graphe.

Il est important de souligner que l'algorithme de Warshall calcule aussi la fermeture transitive d'un graphe orienté.

1.2. Cas évolutif : algorithmes réunir et trouver

Un graphe non orienté de n sommets peut être connu par la donnée successive de ses arêtes. Les composantes connexes du graphe final peuvent alors être obtenues par regroupements successifs des composantes connexes du graphe en évolution. L'ajout d'une arête x—y au graphe peut, en effet, modifier l'ensemble des composantes connexes en provoquant la réunion de la composante connexe de x avec celle de y, dans le cas où elles étaient distinctes. On est donc amené à *trouver* à quelle composante connexe appartient un sommet donné, et aussi, si deux sommets ne sont pas dans la même composante connexe, à *réunir* leurs composantes connexes en une seule.

Déterminer les composantes connexes d'un graphe G, c'est déterminer l'ensemble des classes pour la relation d'équivalence *chaîne*, où x *chaîne* y est vrai si, et seulement si, il existe une chaîne entre les deux sommets x et y. De façon plus générale, les opérations *trouver* et *réunir* sont utilisées pour construire la partition associée à une relation d'équivalence évolutive, c'est-à-dire une relation dont les instances sont connues progressivement. L'opération *trouver* associe à tout élément de l'ensemble sur lequel est définie la relation d'équivalence un élément privilégié de sa classe d'équivalence, le même pour tous les éléments de cette classe, appelé représentant de la classe. Quant à l'opération réunir, elle réunit deux classes distinctes en une seule.

1.2.1. Spécification formelle

Un ensemble muni d'une relation d'équivalence peut être considéré comme un graphe non orienté qui vérifie certaines propriétés. Les éléments de l'ensemble sont alors représentés par les sommets et la relation d'équivalence par la relation arête. Notons que puisqu'il s'agit de graphes non orientés, la propriété de symétrie résulte des axiomes.

En fait, on ne connaît pas toute la relation d'équivalence, mais on en connaît certaines occurrences, données progressivement. Comme on sait que c'est une relation d'équivalence, l'ajout explicite d'une seule occurrence (c'est-à-dire, l'ajout d'une arête au graphe) s'accompagne de l'ajout implicite d'un certain nombre d'autres occurrences (c'est-à-dire, d'autres arêtes). Cela revient à construire, lors de l'ajout d'une arête, la fermeture réflexive et transitive du nouveau graphe.

Appelons *fermeture* l'opération qui effectue la construction de la fermeture réflexive et transitive d'un graphe. Sa spécification utilise l'opération *fermeture-transitive* spécifiée au paragraphe précédent.

Elle a pour profil :

fermeture : Graphe → Graphe

Elle est partout définie et vérifie les axiomes suivants :

pour tout graphe G et tous sommets s et s' :

{ *un graphe a même ensemble de sommets que sa fermeture :*}
s *est-un-sommet-de fermeture*(G) = s *est-un-sommet-de* G
{ *réflexivité :*}
(s *est-un-sommet-de* G) = vrai
$\Rightarrow$ $< s, s >$ *est-une-arête-de fermeture*(G) = vrai
{*transitivité :*}
$s \neq s' \Rightarrow < s, s' >$ *est-une-arête-de fermeture*(G) =
$< s, s' >$ *est-une-arête-de fermeture-transitive*(G)

Les profils des opérations *trouver* et *réunir* sont les suivants :

trouver : Sommet × Graphe → Sommet
réunir : Sommet × Sommet × Graphe → Graphe

Ces opérations ne sont définies que sur des graphes qui représentent des relations d'équivalence. On a donc les préconditions suivantes :

préconditions
trouver(s, G) **est-défini-ssi**
s *est-un-sommet-de* G = vrai & G = *fermeture*(G)

réunir(s, s', G) **est-défini-ssi**
s *est-un-sommet-de* G = vrai & s' *est-un-sommet-de* G = vrai &
G = *fermeture*(G)

Les opérations vérifient les axiomes suivants :

axiomes
pour tout graphe G et pour tous sommets s et s' :
{*deux éléments sont dans une même classe d'équivalence si, et seulement si, ils ont un même représentant :*}
($< s, s' >$ *est-une-arête-de* G) = vrai $\Leftrightarrow$ *trouver*(s, G) = *trouver*(s', G)
{*un représentant est représenté par lui-même :* }
trouver(*trouver*(s, G), G) = *trouver*(s, G)
{*quand on ajoute une occurrence à la relation, on construit sa fermeture réflexive et transitive :*}
réunir(s, s', G) = *fermeture*(*ajouter-l'arête* $< s, s' >$ *à* G)

1.2.2. Algorithmes élémentaires

Principe

On a déjà vu (cf. $3^{ème}$ partie) que les structures d'arbres sont bien adaptées pour faire des recherches dans des ensembles d'éléments qui évoluent. On choisit donc de représenter les classes d'équivalence par une forêt. En toute rigueur, il s'agit d'une forêt d'arborescences au sens de la théorie des graphes, dans la mesure où l'ordre entre les fils d'un nœud n'est pas pertinent. En fait, on choisit un ordre arbitraire pour représenter ces arborescences par des arbres au sens du chapitre 7.

Chaque arbre de la forêt a pour racine le représentant de la classe qu'il représente. Effectuer l'opération *trouver* pour l'élément x, revient à obtenir la racine de l'arbre qui contient x. Cette opération est facile à effectuer si l'on connaît le père de chaque élément. De plus, pour réunir deux classes d'équivalence, il suffit de raccrocher l'un des arbres à la racine de l'autre.

Algorithme

Pour représenter la forêt d'arbres, on utilise un tableau d'entiers qui indique les pères des nœuds dans la forêt. Un nœud r n'ayant pas de père est une racine : on donne une valeur privilégiée -1 à *père*$[r]$, qui sert de sentinelle lors de la recherche en amont de la racine. Si au départ, toutes les classes d'équivalence sont réduites à des singletons, le tableau *père* est initialisé à -1.

La fonction *trouver* et la procédure *réunir* suivantes utilisent le type *tab*, où n est le cardinal de l'ensemble :

```
type tab = array[1..n] of integer;

function trouver(x : integer; père : tab) : integer;
{trouver(x, père) renvoie la racine de l'arbre contenant x}
var i : integer;
```

```
begin
    i := x;
    while père[i] > 0 do i := père[i];
    trouver := i
end trouver;

procedure réunir(x, y : integer; var père : tab);
{réunir(x, y, père) raccroche la racine de l'arbre contenant y à la racine
de l'arbre contenant x, s'ils sont distincts}
var r1, r2 : integer;
begin
    r1 := trouver(x, père);
    r2 := trouver(y, père);
    if r1 <> r2 then père[r2] := r1
end réunir;
```

Exemple

On examine les mises à jour successives des composantes connexes du graphe évolutif comportant $n = 14$ sommets et dont les arêtes sont connues progressivement dans l'ordre suivant :

3—4, 6—8, 12—14, 1—2, 5—6, 9—10, 4—5, 11—9, 5—7, 6—3, 12—13, 10—14, 13—9, 2—4.

L'ajout de chaque arête x—y provoque l'appel de *réunir*$(x, y,$ *père*$)$. Notons que l'ordre des arguments x et y est important. Au départ, le graphe n'a pas d'arête : les composantes connexes sont réduites à des singletons.

Après ajout des arêtes 3—4, 6—8, 12—14, 1—2 et 5—6, on obtient le graphe G_1. Ses composantes connexes sont représentées par la forêt d'arbres de la figure 1.

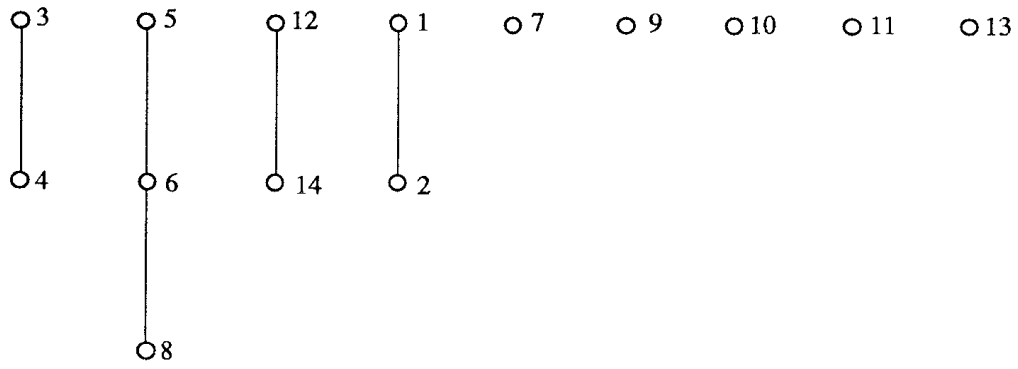


Figure 1. Composantes connexes de G_1.

On continue à ajouter des arêtes à G_1 : après ajout des arêtes 9—10 et 4—5, on obtient le graphe G_2. Ses composantes connexes sont représentées par la forêt de la figure 2.

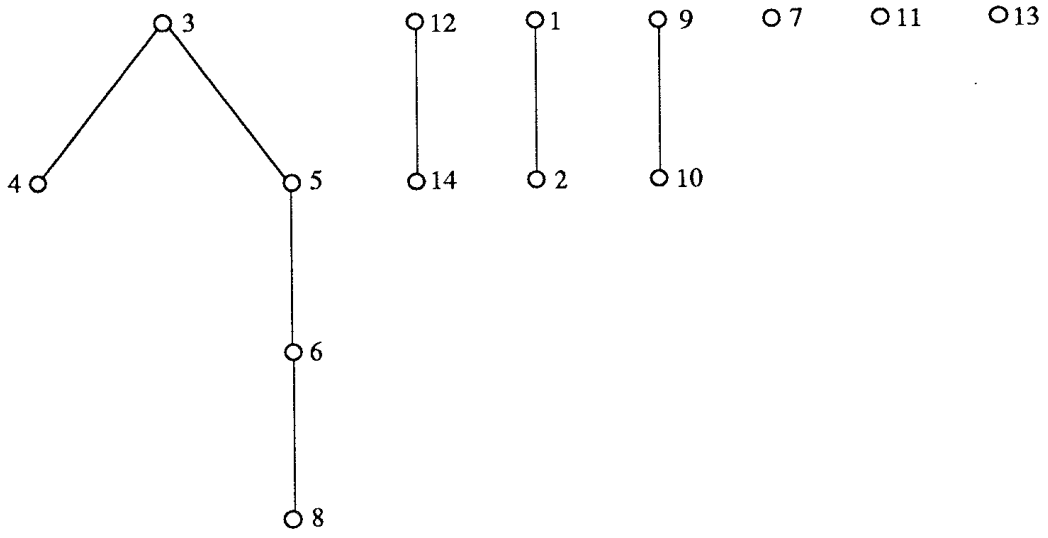


Figure 2. Composantes connexes de G_2.

On continue à ajouter des arêtes à G_2 : après ajout des arêtes 11—9, 5—7, 6—3 et 12—13, on obtient le graphe G_3. Ses composantes connexes sont représentées par la forêt de la figure 3.

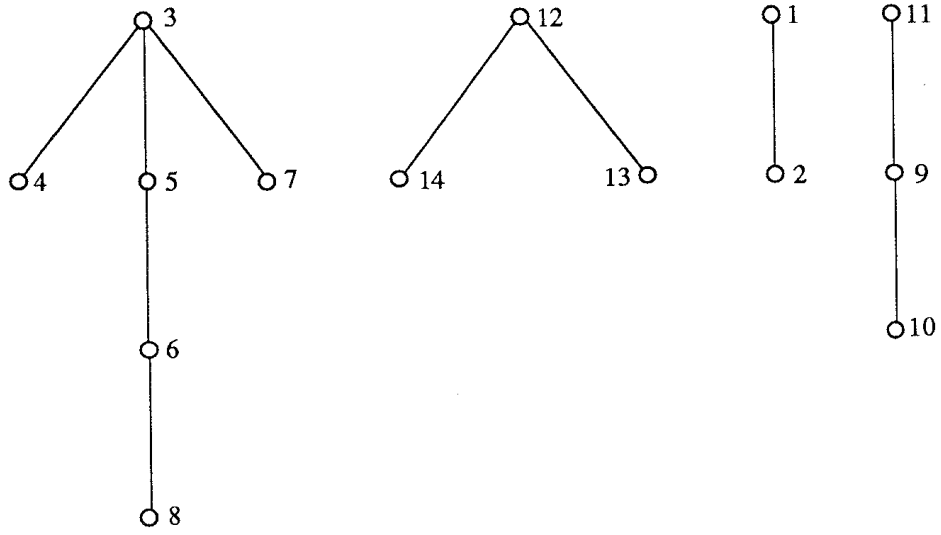


Figure 3. Composantes connexes de G_3.

On ajoute ensuite l'arête 10—14 à G_3. On obtient le graphe G_4 dont les composantes connexes sont représentées à la figure 4.

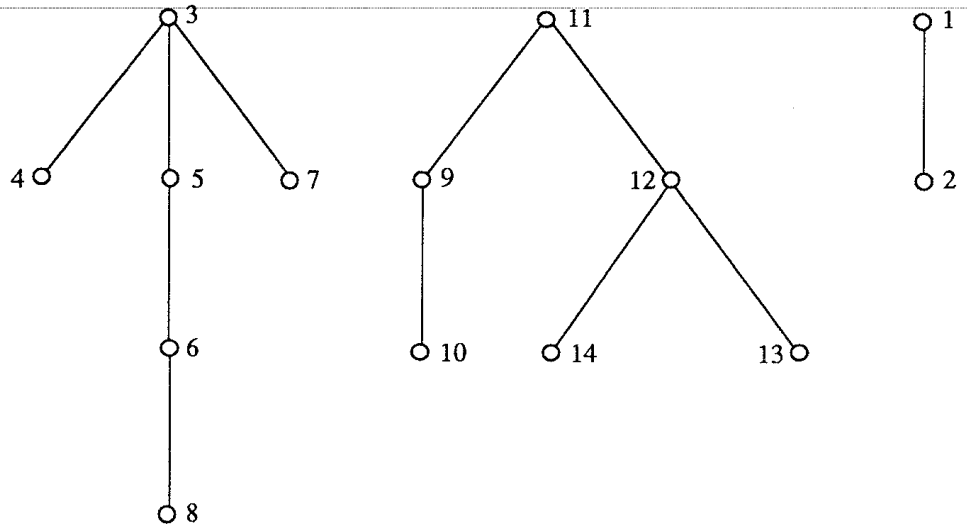


Figure 4. Composantes connexes de G_4.

Finalement on ajoute à G_4 les arêtes 13—9 et 2—4 : on obtient le graphe G_5 dont les deux composantes connexes sont représentées à la figure 5.

Il est important de souligner que, bien que ces arbres correspondent aux composantes connexes du graphe G_5, les liens entre les sommets dans les arbres ne correspondent pas nécessairement aux arêtes de G_5 (on a dessiné le graphe G_5 à la figure 6).

Pour cet exemple, la profondeur maximale d'un arbre de la forêt est 4, alors que le graphe a 14 sommets : la forme obtenue est assez compacte. Malheureusement, ce n'est pas toujours le cas. Considérons, par exemple, le cas d'un graphe de n sommets dont les arêtes (en nombre $n-1$) sont données dans l'ordre suivant :

$$\{2,1\},\{3,1\},...,\{i-1,1\},\{i,1\}...,\{n,1\}.$$

La forêt obtenue par appels successifs de la procédure *réunir* comporte un unique arbre de racine n et de profondeur $n-1$ (cf. figure 7). Pour cet arbre, la complexité en temps dans le pire des cas de la fonction *trouver*, obtenue lorqu'on cherche la racine de l'arbre qui contient le sommet 1, est de l'ordre de $\Theta(n)$. De même, tester si deux éléments sont dans la même classe d'équivalence, nécessite, dans le pire des cas, un temps de l'ordre de $\Theta(n)$.

Pour améliorer la complexité, il est naturel de vouloir éviter les formes dégénérées (filiformes) en essayant d'équilibrer les arbres. On propose deux idées d'améliorations qui, combinées entre elles, conduisent à un algorithme très efficace.

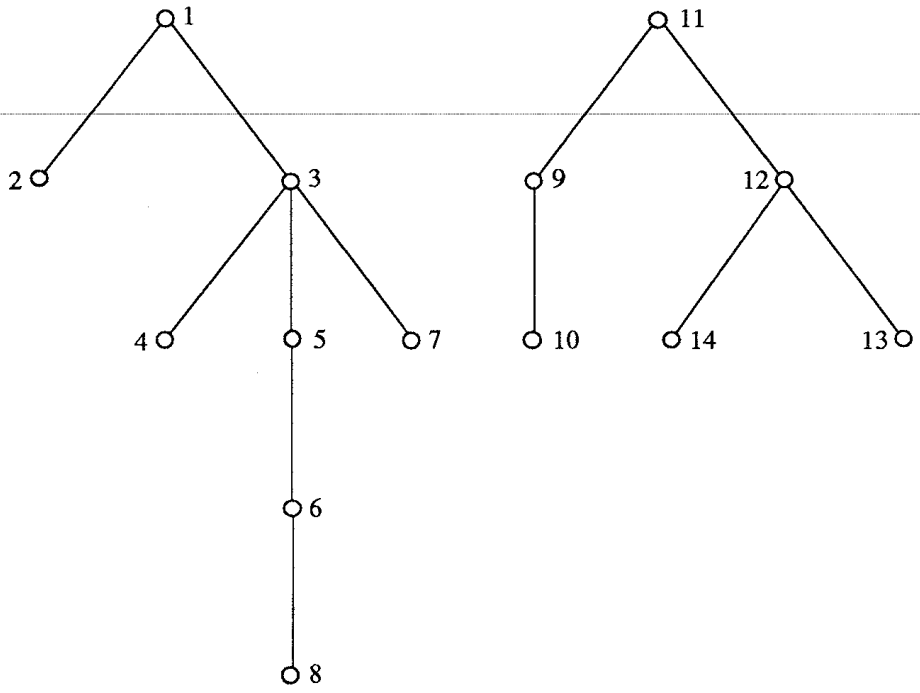


Figure 5. Composantes connexes de G_5.

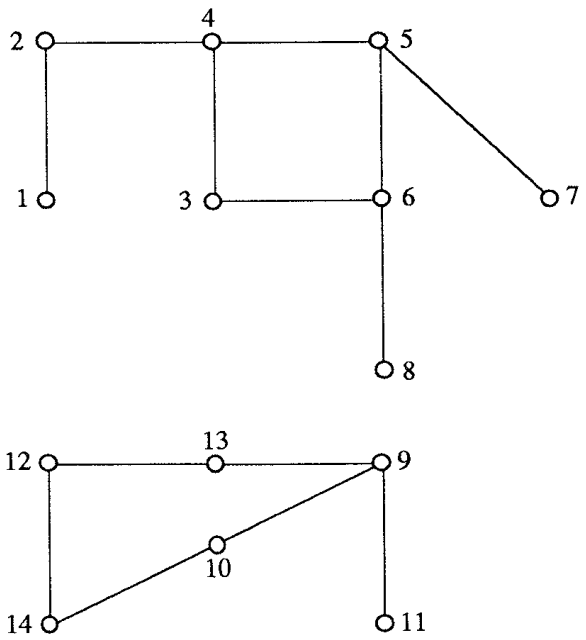


Figure 6. Graphe non orienté G_5.

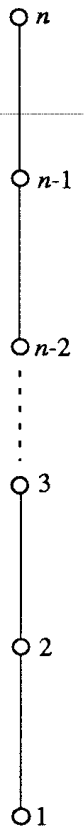


Figure 7. Arbre dégénéré.

1.2.3. Union pondérée

Une première amélioration simple consiste, lors de l'ajout de l'arête x—y à ne pas raccrocher systématiquement l'arbre contenant y, à la racine de l'arbre contenant x, mais à raccrocher celui des deux arbres qui contient le moins d'éléments à la racine de celui qui en contient le plus : on dit qu'on fait l'*union pondérée* des deux arbres.

Algorithme d'union pondérée

La racine de chaque arbre contient un nombre strictement négatif dont la valeur absolue indique le nombre d'éléments de l'arbre. Puisqu'au début de la construction de la partition, les classes d'équivalence sont réduites à des singletons, l'initialisation du tableau *père* à -1 est encore valable. On obtient la procédure *réunir-pondéré* suivante qui a pour arguments les racines r_1 et r_2 des deux arbres contenant respectivement x et y. Ces racines sont obtenues par la fonction *trouver* donnée précédemment : r_1 = *trouver*(x, *père*) et r_2 = *trouver*(y, *père*).

```
procedure réunir-pondéré(r1, r2 : integer; var père : tab);
{réunir-pondéré(r1, r2, père) raccroche l'arbre ayant le moins d'éléments
à la racine de l'arbre qui en a le plus, dans le cas où les deux arbres sont
distincts; r1 et r2 sont les racines des arbres }
begin
    if r1 <> r2 then
        if père[r1] > père[r2] {l'arbre de racine r1 a strictement moins
        d'éléments que l'arbre de racine r2}
            then begin
                père[r2] := père[r2] + père[r1];
                père[r1] := r2
            end
            else begin
                père[r1] := père[r1] + père[r2];
                père[r2] := r1
            end
{S'il y a égalité, on raccroche le deuxième arbre au premier}
end réunir-pondéré;
```

Exemple

Reprenons l'exemple précédent sur les graphes et examinons les différentes forêts successivement obtenues lorsqu'on fait l'union pondérée à chaque fois. Au départ, le graphe n'a pas d'arête : les composantes connexes sont réduites à des singletons.

Après ajout des arêtes 3—4, 6—8, 12—14, 1—2 et 5—6, on obtient la forêt dessinée à la figure 8 qui correspond aux composantes connexes de G_1.

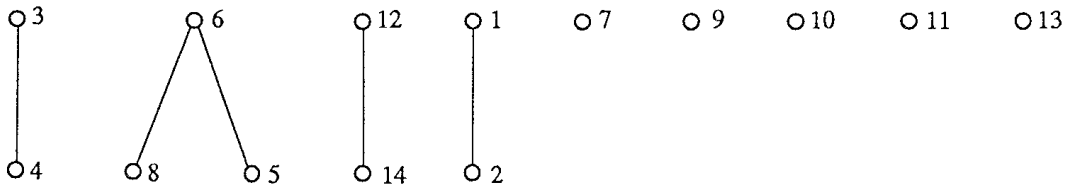


Figure 8. Forêt obtenue pour G_1 avec unions pondérées.

Après ajout à G_1 des arêtes 9—10 et 4—5, on obtient la forêt dessinée à la figure 9 qui correspond aux composantes connexes de G_2.

Après ajout à G_2 des arêtes 11—9, 5—7, 6—3 et 12—13, on obtient la forêt dessinée à la figure 10 qui correspond aux composantes connexes de G_3.

Après ajout à G_3 de l'arête 10—14, on obtient la forêt dessinée à la figure 11 qui correspond aux composantes connexes de G_4. Rappelons que si les deux arbres ont le même nombre d'éléments, on raccroche le deuxième au premier.

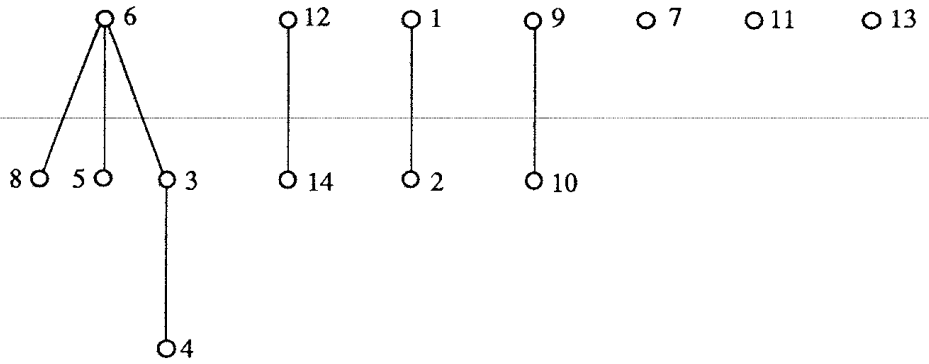


Figure 9. Forêt obtenue pour G_2 avec unions pondérées.

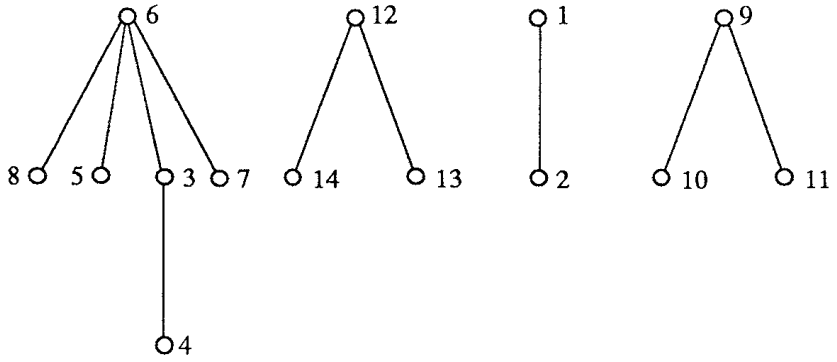


Figure 10. Forêt obtenue pour G_3 avec unions pondérées.

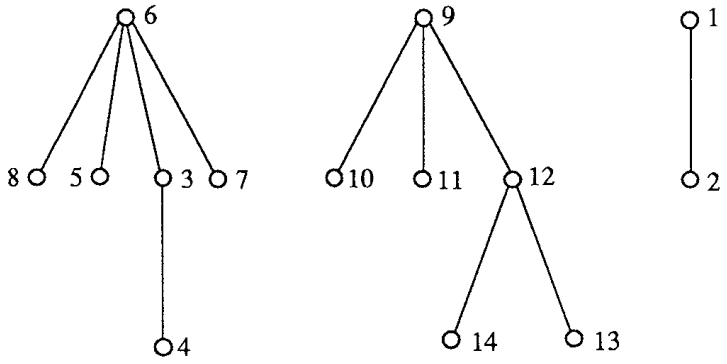


Figure 11. Forêt obtenue pour G_4 avec unions pondérées.

Finalement on ajoute à G_4 les arêtes 13—9 et 2—4 : on obtient le graphe G_5 donné à la figure 6. On obtient une forêt de deux arbres représentée à la figure 12. Notons que la profondeur maximale est de 2 alors que celle de la forêt obtenue sans union pondérée est de 4 (cf. figure 5). On va montrer que cet algorithme est plus intéressant en général.

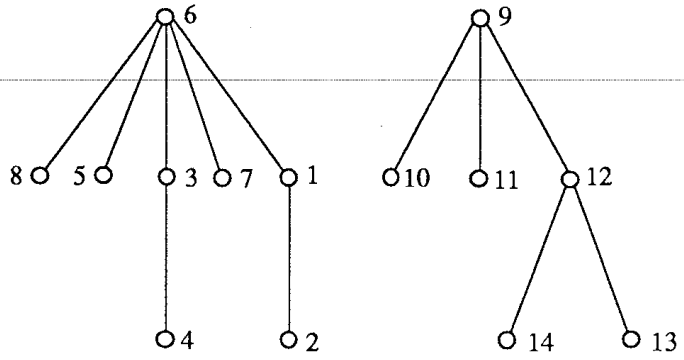


Figure 12. Forêt obtenue pour G_5 avec unions pondérées.

Complexité

En faisant des unions pondérées on évite les formes d'arbres dégénérés. Plus précisément, on a la proposition suivante.

Proposition : Après une suite quelconque d'unions pondérées, tout arbre de m nœuds dans la forêt a une profondeur inférieure ou égale à $\lfloor \log_2 m \rfloor$.

Preuve : On raisonne par récurrence sur m.

$m = 1$: dans ce cas, l'arbre est de profondeur 0; or $\lfloor \log_2 1 \rfloor = 0$.

Hypothèse de récurrence : supposons que pour tout $k < m$, et pour toute forêt, tout arbre qui a k nœuds a une profondeur inférieure ou égale à $\lfloor \log_2 k \rfloor$.

Soit T un arbre ayant m éléments, de profondeur h. L'arbre T résulte de l'union pondérée de deux arbres T_1 et T_2, qui ont respectivement m_1 et m_2 éléments. Supposons par exemple, que $m_1 < m_2$: T a alors été obtenu par raccrochage de T_1 à T_2. Soit h_1 (respectivement h_2) la profondeur de T_1 (respectivement T_2).

$$h = \max(h_1 + 1, h_2).$$

Par hypothèse de récurrence, $h_1 \leq \lfloor \log_2 m_1 \rfloor$ et $h_2 \leq \lfloor \log_2 m_2 \rfloor$. Par suite :

$$h \leq \max(\lfloor \log_2 m_1 \rfloor + 1, \lfloor \log_2 m_2 \rfloor).$$

Or $\lfloor \log_2 m_1 \rfloor + 1 = \lfloor \log_2 2m_1 \rfloor \leq \lfloor \log_2 (m_1 + m_2) \rfloor$ car $m_1 < m_2$.

D'où : $h \leq \lfloor \log_2 (m_1 + m_2) \rfloor = \lfloor \log_2 m \rfloor$.

Le cas où $m_1 = m_2$ se traite de façon analogue. □

Si la forêt est construite par unions pondérées successives et représente les classes d'équivalence d'un ensemble à n éléments, trouver la classe d'équivalence d'un élément, ou tester si deux éléments sont dans la même classe, requièrent une complexité en temps au pire en $\Theta(\log_2 n)$.

1.2.4. Union pondérée avec compression des chemins

Compression des chemins

Afin de rendre encore plus compactes les formes d'arbres obtenues, à chaque appel de la fonction *trouver* pour un sommet x, on raccroche directement à la racine de l'arbre, tous les sommets situés sur le chemin remontant de x vers la racine : on dit qu'on effectue la *compression des chemins.*

Comme on veut obtenir à la fois la racine r_1 de l'arbre contenant x et effectuer la compression du chemin de x vers la racine, en modifiant le tableau *père*, on utilise une procédure plutôt qu'une fonction : c'est la procédure *trouver-rapide* donnée ci-dessous.

```
procedure trouver-rapide(x : integer; var r1 : integer; var père : tab);
{trouver-rapide(x, r1, père) renvoie la racine r1 de l'arbre contenant x
et effectue la compression du chemin de x vers r1}
var i, j : integer;
begin
    i := x;
    while père[i] > 0 do i := père[i];
    r1 := i; {r1 est la racine}
    i := x;
    while père[i] >0 do begin  {compression du chemin}
            j := i; i := père[i]; père[j] := r1
    end
end trouver-rapide;
```

Exemple

On va voir sur un exemple ce qu'apporte l'algorithme d'unions pondérées avec compressions des chemins. Considérons l'exemple sur les graphes : on ajoute toujours les mêmes arêtes, et dans le même ordre. Les forêts obtenues avec *trouver-rapide* et *réunir-pondéré* pour les graphes G_1, G_2, G_3 et G_4 sont les mêmes que celles qu'on obtient avec *réunir-pondéré* seulement. Ce n'est pas surprenant puisque la compression des chemins ne peut être effective que si les arbres ont déjà une profondeur au moins égale à 2.

Après ajout à G_4 des arêtes 13—9 et 2—4, on obtient une forêt différente, donnée à la figure 13. Notons que l'ajout de l'arête 13—9 ne provoque pas de réunion de composantes connexes, mais entraîne une compression de chemin.

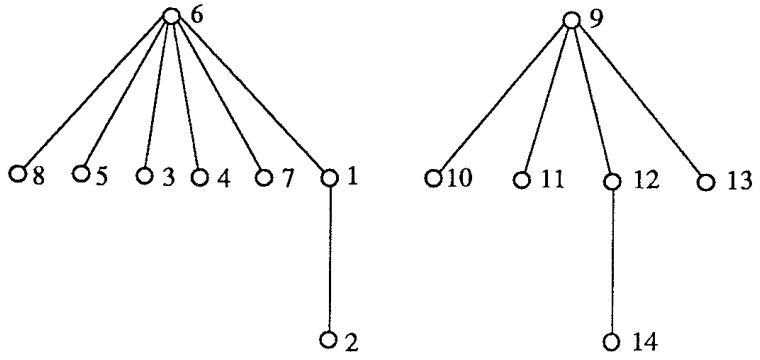


Figure 13. Forêt obtenue pour G_5 avec unions pondérées et compression des chemins.

La forêt obtenue à la figure 13 est différente de la forêt obtenue à la figure 12 (sans compression des chemins), mais elles ont la même profondeur. Pour mieux voir ce qu'apporte la compression des chemins, ajoutons encore l'arête 14—2 au graphe G_5. On obtient alors un graphe G_6 connexe. Avec l'algorithme d'union pondérée seulement, on obtient l'arbre de profondeur 3 dessiné à la figure 14, tandis qu'en effectuant en plus, la compression des chemins, on obtient l'arbre de profondeur 2 dessiné à la figure 15. (Notons qu'avec le premier algorithme élémentaire, on aurait un arbre de profondeur 5.)

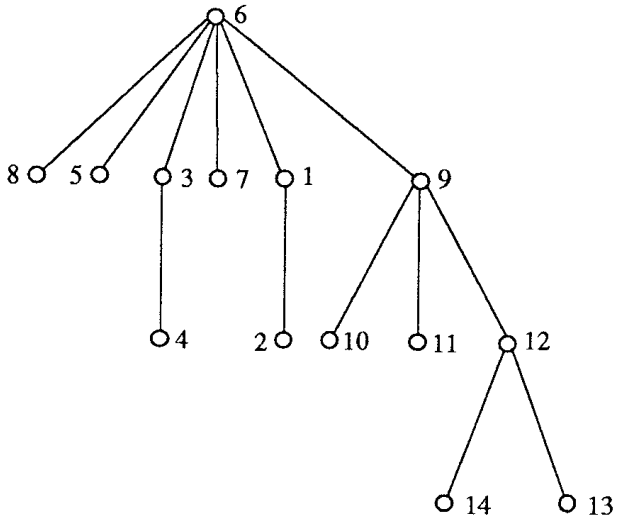


Figure 14. Forêt obtenue pour G_6 avec unions pondérées.

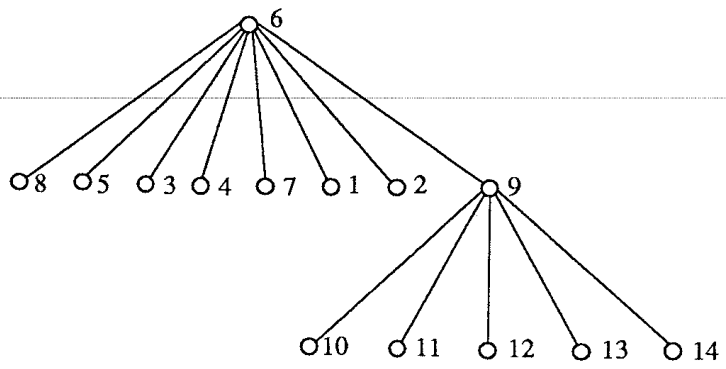


Figure 15. Forêt obtenue pour G_6 avec unions pondérées et compression des chemins.

Complexité

Si on construit les classes d'équivalence d'un ensemble de n éléments par unions pondérées successives et compressions des chemins, trouver la classe d'équivalence d'un élément, ou tester si deux éléments sont dans la même classe, requièrent une complexité en temps au pire quasi-constante. Plus précisément, pour toute suite d'opérations qui sont des unions pondérées et des compressions de chemins, si on a m opérations *trouver-rapide*, la complexité en temps de l'algorithme est dans le pire des cas en $O(n+m\alpha(m+n,n))$, où $\alpha(u,v)$ est une fonction qui croît très lentement, bien plus lentement que la fonction logarithmique (elle est définie comme l'inverse de la fonction d'Ackermann, qui elle, est une fonction qui croît très vite). Dans la pratique, on peut considérer que $\alpha(u,v)$ est une constante inférieure ou égale à 4.

En conclusion, une fois la forêt construite, on peut tester si un graphe non orienté est connexe en un temps au pire quasiment linéaire par rapport au nombre de sommets (on exécute *trouver-rapide* pour chaque sommet). Pour construire la forêt progressivement, il faut un temps au pire quasi-linéaire par rapport au nombre d'arêtes du graphe, si $n \leq p$. C'est donc un peu plus coûteux que le parcours en profondeur, mais cet algorithme ne nécessite pas qu'on stocke les arêtes (algorithme *on-line*).

2. Composantes fortement connexes

On présente maintenant deux algorithmes pour la recherche des composantes fortement connexes. Le premier effectue deux parcours en profondeur simples et une inversion du graphe. Le deuxième effectue un parcours en profondeur complexe et nécessite l'usage d'une pile.

2.1. Parcours en profondeur du graphe et de son inverse

2.1.1. Principe et algorithme

Soit G un graphe orienté. L'algorithme utilise deux fois le parcours en profondeur, la première fois sur G et la deuxième fois sur le graphe G^{-1} obtenu à partir de G en inversant le sens de tous les arcs. Le principe de l'algorithme est le suivant. On effectue un parcours en profondeur de G, en construisant une liste L des sommets en ordre suffixe, à l'aide de la procédure *profsuf*. On effectue ensuite un parcours en profondeur sur G^{-1}, en appelant la procédure *parcours* pour le dernier sommet de la liste L, c'est-à-dire, le sommet ayant le numéro le plus élevé dans la numérotation suffixe de G. Si tous les sommets ne sont pas marqués à la fin de cet appel, on recommence l'exécution de la procédure *parcours* avec le dernier sommet non marqué de L, et ainsi de suite, jusqu'à ce que tous les sommets soient marqués. Les numéros suffixes (ordre pour G) des racines des arborescences de la forêt couvrante de G^{-1} sont donc en ordre décroissant. Les sous-graphes de G engendrés par les sommets des arborescences de la forêt couvrante de G^{-1} ainsi obtenue, constituent les composantes fortement connexes de G.

Dans la procédure qui suit, on associe à chaque sommet le numéro de la composante fortement connexe à laquelle il appartient. On manipule le graphe G^{-1} à l'aide des opérations *$d^{\circ -}$_ de_ dans_* et *_ ème-pred-de_ dans_*, effectuées sur G. On représente la liste L dans un tableau, si bien qu'on peut facilement la parcourir de la fin vers le début.

```
procedure comp-fort-connexe1(G : Graphe; var comp : array [1..n] of integer);
{Après exécution de la procédure comp-fort-connexe1, comp[i] contient
le numéro de la composante fortement connexe du sommet i}
type LISTE = array[1..n] of integer;
var marque : array[1..n] of boolean; i,h,k : integer; L : LISTE;
procedure profsuf (x : integer; G : Graphe; var M : array [1..n] of boolean;
                   var L : LISTE; var h : integer);
{parcours en profondeur du graphe G et construction de la liste L des sommets
de G en ordre suffixe; x est un sommet; h est le nombre de sommets déjà visités}
var j,y : integer;
{y est un sommet}
begin
    M[x] := true;
    for j := 1 to d°+ de x dans G do begin
          y := j ème-succ-de x dans G;
          if not M[y] then profsuf(y,G,M,L,h)
    end;
    {traitement du sommet x à la dernière rencontre :}
    h := h + 1; L[h] := x
end profsuf;
```

```
procedure parcours (x : integer; G : Graphe; var M : array[1..n] of boolean;
                    var cp : array [1..n] of integer; k : integer);
{parcours en profondeur du graphe inverse de G et remplissage du tableau cp
par la valeur k, pour tous les sommets de l'arborescence; x est un sommet}
var y, i : integer;
{y est un sommet}
begin
    M[x] := true;
    cp[x] := k;
    for i := 1 to d°− de x dans G do begin
        y := i ème-pred-de x dans G;
        if not M[y] then parcours(y, G, M, cp, k)
    end
end parcours;

begin
    {initialisations :}
    for i := 1 to n do begin comp[i] := 0; marque[i] := false end;
    k := 0;
    {k est le nombre de composantes fortement connexes détectées jusque-là}
    for h := 1 to n do L[h] := 0;
    h := 0;

    {construction de la liste L des sommets en ordre suffixe de G :}
    for i := 1 to n do if not marque[i] then profsuf(i, G, marque, L, h);
    {parcours de G^-1 :}
    for i := 1 to n do marque[i] := false;
    for i := n downto 1 do if not marque[L[i]] then begin
        k := k + 1;
        parcours(L[i], G, marque, comp, k)
    end
end comp-fort-connexe1;
```

2.1.2. Exemple

Considérons le graphe G de la figure 16.

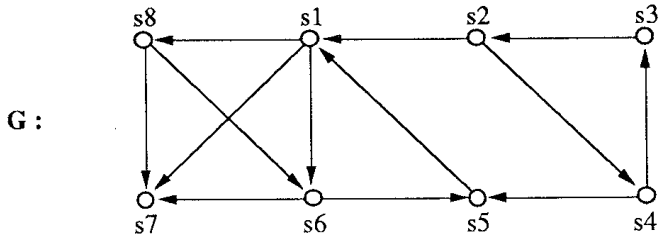


Figure 16. Graphe orienté G.

L'exécution de l'algorithme sur G donne les résultats suivants (on prend toujours la convention que les sommets et les successeurs d'un sommet sont choisis par ordre de numéros croissants).

La forêt couvrante associée au parcours en profondeur de G est dessinée à la figure 17; elle permet d'obtenir facilement la liste L des sommets en ordre suffixe.

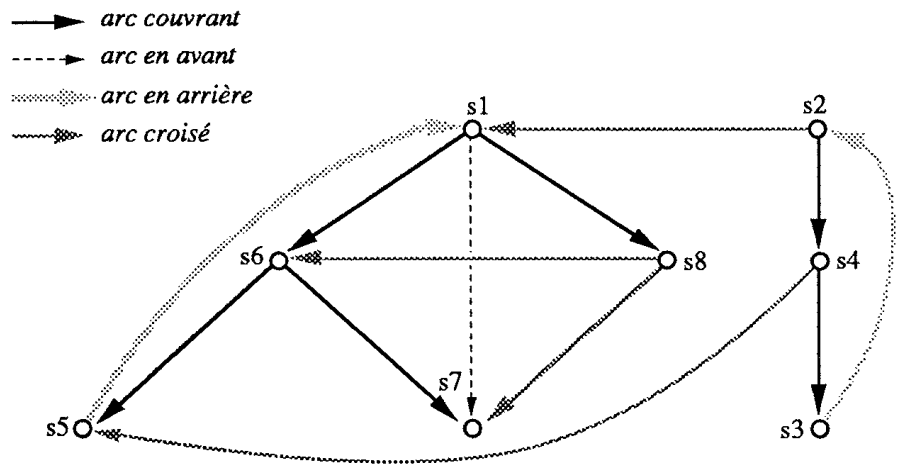


Figure 17. Forêt couvrante de G.

L :	5	7	6	8	1	3	4	2
	1	2	3	4	5	6	7	8

Le graphe G^{-1} est donné à la figure 18.

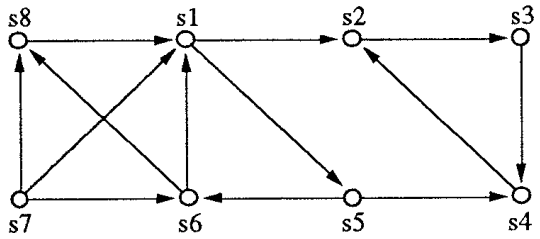


Figure 18. Graphe orienté G^{-1}.

On effectue le parcours en profondeur de G^{-1} en commençant par le sommet $s2$ car c'est le sommet de numéro suffixe le plus élevé dans la liste L. Après avoir obtenu l'arborescence de racine $s2$ (qui contient $s3$ et $s4$), on repart avec $s1$ qui est, parmi les sommets non encore marqués, celui qui a le numéro suffixe le plus élevé.

La forêt couvrante associée au parcours en profondeur de G^{-1}, effectué en lisant la liste L de droite à gauche, est dessinée à la figure 19.

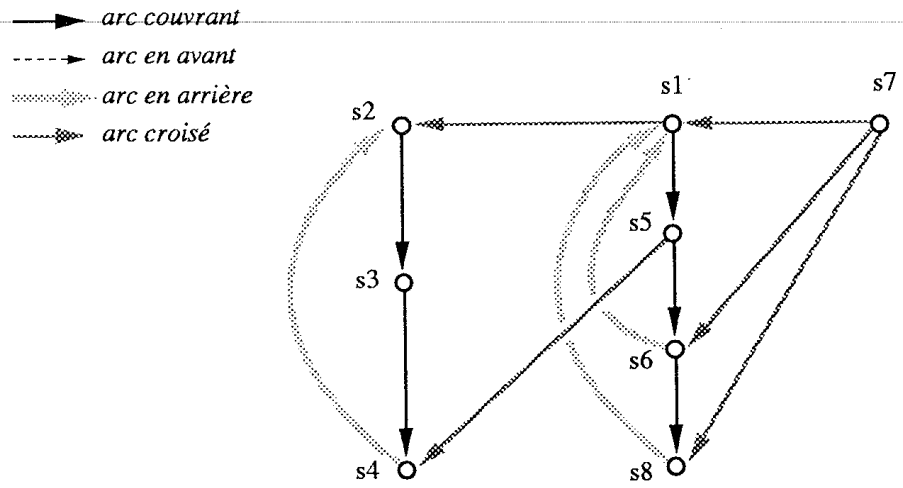


Figure 19. Forêt couvrante de G^{-1}.

Le graphe G a trois composantes fortement connexes.

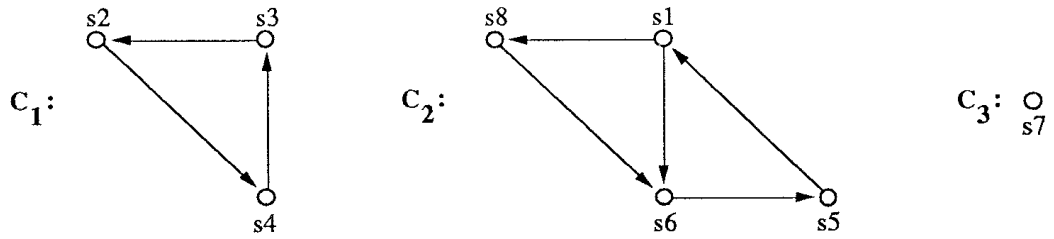


Figure 20. Composantes fortement connexes de G.

On obtient le tableau *comp* suivant :

comp :	2	1	1	1	2	2	3	2
	1	2	3	4	5	6	7	8

2.1.3. Justification

On va montrer que deux sommets x et z appartiennent à la même composante fortement connexe de G si, et seulement si, ils appartiennent à une même arborescence de la forêt couvrante de G^{-1}.

• On suppose d'abord que x et z sont dans la même composante fortement connexe de G; ils sont donc dans la même composante fortement connexe de G^{-1}. Or, deux sommets appartenant à deux arborescences distinctes de la forêt couvrante de G^{-1} ne peuvent être dans la même composante fortement connexe de G^{-1}, car les arcs croisés vont de la droite vers la gauche. Par suite, x et z sont dans la même arborescence de la forêt couvrante de G^{-1}.

• Réciproquement, soit x et z deux sommets d'une même arborescence de la forêt couvrante de G^{-1}. Soit y la racine de cette arborescence. On va d'abord montrer que x et y sont dans une même composante fortement connexe de G.

Comme x appartient à l'arborescence de racine y, il existe un chemin de y vers x dans G^{-1}, et par suite, il existe un chemin de x vers y dans G.

D'autre part, si x appartient à l'arborescence de racine y dans la forêt couvrante de G^{-1}, c'est que le sommet x n'est pas marqué lorsque le parcours en profondeur de G^{-1} marque y; mais alors y apparaît après x dans le parcours suffixe de G. Comme il existe un chemin de x vers y dans G, il est exclu que x et y soient dans des arborescences distinctes de la forêt couvrante de G. Il reste deux possibilités pour x et y dans la forêt couvrante de G, dessinées à la figure 21.

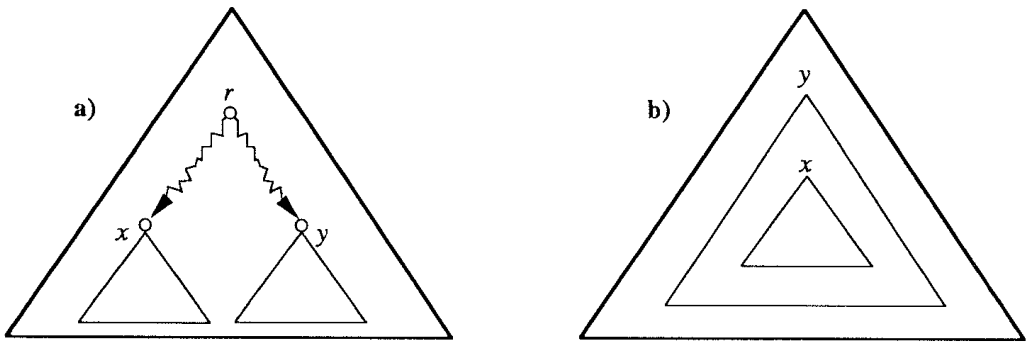


Figure 21. Forêt couvrante de G : y est après x dans l'ordre suffixe de G et il existe un chemin de x vers y dans G.

On va montrer que le premier cas est impossible. Rappelons qu'il existe un chemin de x vers y dans G. Si y n'est pas dans la sous-arborescence de racine x, c'est qu'on a rencontré un sommet du chemin de x vers y, qui était déjà marqué avant le parcours de cette arborescence (cf figure 22). Ce chemin contient donc un sommet v qui est un ancêtre commun à x et y et qui est après x et y dans l'ordre suffixe de la forêt couvrante de G. Si un tel sommet v existe, y ne peut pas être racine de l'arborescence de G^{-1} qui contient x, telle qu'elle est construite par l'algorithme. En effet, si dans cette forêt couvrante de G^{-1}, x est dans l'arborescence de racine y, c'est que x n'est pas accessible dans G^{-1} depuis un sommet v rencontré dans l'ordre suffixe de la forêt couvrante de G, après y, et donc il n'y a pas de chemin de x vers v dans G. Ce premier cas est donc impossible.

Seul reste alors le cas où x appartient à la sous-arborescence de racine y dans la forêt couvrante de G : il existe donc un chemin de y vers x dans G. En conséquence, x et y appartiennent à une même composante fortement connexe de G.

On montre de même que y et z appartiennent à une même composante fortement connexe de G. Il en résulte que x et z sont dans la même composante fortement connexe de G.

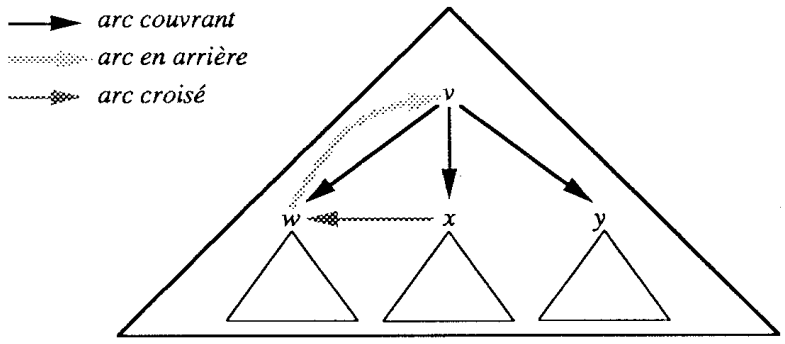


Figure 22. Chemin de x vers y dans G.

2.1.4. Analyse de la complexité

Soit n le nombre de sommets de G (resp. G^{-1}), et soit p le nombre d'arcs de G (resp. G^{-1}). Si on considère une représentation de G et de G^{-1} par des listes d'adjacence, les parcours en profondeur réalisés sur G et sur G^{-1} ont une complexité en $\Theta(\max(n,p))$. Avec cette représentation des graphes, la construction de G^{-1} s'effectue avec la même complexité en temps et demande $\Theta(n+p)$ places. La recherche des sommets des différentes composantes connexes nécessite dans ce cas un temps $\Theta(\max(n,p))$.

Si on choisit pour G (et G^{-1}) une représentation par matrices d'adjacence, la complexité en temps de l'algorithme est en $\Theta(n^2)$, comme celle du parcours en profondeur, et la construction de G^{-1} est inutile, puisqu'on peut utiliser la matrice de G en échangeant les indices. Cette représentation est donc intéressante si on a des graphes avec beaucoup d'arcs.

2.2. Algorithme de Tarjan

2.2.1. Principe

L'algorithme de recherche des composantes fortement connexes présenté maintenant, effectue un seul parcours en profondeur et en a la complexité. Comme l'algorithme précédent, il exploite les propriétés de la forêt couvrante associée au parcours en profondeur, plus précisément ici, les propriétés des arcs en arrière. Cet algorithme, dit de Tarjan, utilise plusieurs propriétés qui sont données ci-dessous.

Propriétés élémentaires

1) Soient x et y deux sommets qui sont dans une même composante fortement connexe. Soit z un sommet qui appartient à un chemin de x vers y, alors z est dans la même composante fortement connexe que x et y.

2) Soient x et y deux sommets tels qu'il y a un chemin de x vers y. Si dans un parcours en profondeur, on marque x alors que y n'est pas encore marqué, y va être dans la même arborescence que x.

Propriété 1 : Soit $G = < S, A >$ un graphe orienté. Soit $C_i =< S_i, A_i >$ une composante fortement connexe de $G, S_i \subseteq S, A_i \subseteq A$. Soit B_i l'ensemble des arcs de A_i qui sont des arcs couvrants de la forêt couvrante associée au parcours en profondeur de G. Alors $G_i =< S_i, B_i >$ est une arborescence.

> *Preuve* : Il est clair que le graphe non orienté sous-jacent à G_i est sans cycle car il provient d'arcs appartenant à une arborescence (cf. la définition d'une arborescence au chapitre 8). Soit r_i le premier sommet de la composante fortement connexe C_i qui est rencontré lors du parcours en profondeur. Tous les sommets de C_i vont être marqués à partir de r_i et sont donc dans l'arborescence A_i de racine r_i. Soit x un sommet de C_i ; tout sommet sur le chemin de r_i vers x dans A_i est dans C_i (car C_i est une composante fortement connexe). Le graphe G_i est donc une arborescence de racine r_i (c'est l'arborescence A_i dans laquelle on a éventuellement supprimé une ou plusieurs sous-arborescences (cf. figure 23)). □

On appelle ***racine de la composante fortement connexe*** C_i, la racine de l'arborescence G_i associée à C_i, comme dans la propriété 1. C'est le premier sommet de la composante fortement connexe rencontré dans l'ordre préfixe.

On numérote en ordre suffixe les racines des m composantes fortement connexes : $r_1, r_2, ..., r_m$. Il est important de remarquer que si i est inférieur à j, alors soit r_i est à gauche de r_j, soit r_i est un descendant de r_j.

Propriété 2 : Pour tout i, les sommets de la composante fortement connexe C_i sont les descendants de r_i dans la forêt couvrante, qui ne sont pas dans $C_1, ..., C_{i-1}$.

> *Preuve* : On a vu que tout sommet de la composante fortement connexe C_i est dans l'arborescence de racine r_i. Considérons l'un des descendants x de la racine r_i dans la forêt couvrante. Si x n'appartient pas à C_i , alors x appartient à un certain C_j dont la racine r_j appartient, elle aussi, à l'arborescence de racine r_i. Comme les racines sont numérotées en ordre suffixe, les sommets,

descendants de r_i, qui ne sont pas dans C_i, sont donc dans C_j, avec j inférieur à i. □

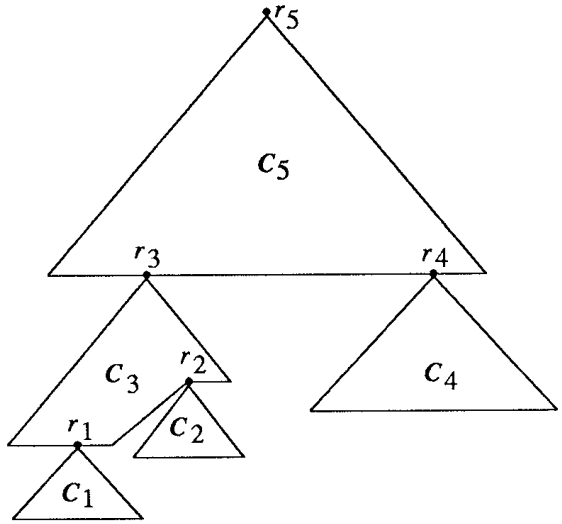


Figure 23. Composantes fortement connexes et leurs racines.

Le principe de l'algorithme de Tarjan est le suivant. Les racines r_i étant considérées en ordre suffixe, on obtient les sommets de la composante fortement connexe C_i en prenant les sommets de l'arborescence de racine r_i qui n'ont pas été pris en compte dans les arborescences précédentes.

Pour décrire l'algorithme, on introduit deux fonctions sur les sommets : la fonction *prefixe* associe à tout sommet sa numérotation dans le parcours préfixe de la forêt couvrante associée à G. La fonction *retour* associe au sommet x le numéro préfixe d'un sommet de la composante fortement connexe de x, rencontré avant x, s'il en existe, ou bien de x lui-même.

Dans quels cas trouve-t-on un sommet z de la composante fortement connexe de x tel que *prefixe*(z) est inférieur à *prefixe*(x)? Un tel sommet doit être accessible depuis x et situé au-dessus de x ou à gauche de x. Pour accéder à ce sommet z, on doit prendre un chemin qui commence par une suite (éventuellement vide) d'arcs couvrants, puis qui continue soit par un arc en arrière, soit par un arc croisé, et qui se poursuit peut-être par d'autres arcs.

Pour savoir s'il existe un sommet z de la composante fortement connexe de x tel que *prefixe*(z) est inférieur à *prefixe*(x), il suffit donc d'examiner si on peut trouver l'un des cas présentés dans la figure 24.

Dans le cas a), il est clair que *prefixe*(z) est inférieur à *prefixe*(x) et que z et x sont dans la même composante fortement connexe.

Dans le cas b), il faut supposer de plus que *prefixe*(z) est inférieur à *prefixe*(x). Si z et x sont dans la même composante fortement connexe, alors x n'est pas le premier sommet de sa composante fortement connexe. Sinon c'est que z appartient à une composante fortement connexe dont la racine r_z a un numéro suffixe plus petit que le numéro suffixe de la racine r_x de la composante fortement connexe de x (car *prefixe*(z) est inférieur à *prefixe*(x)) et donc z a déjà été considéré dans une composante fortement connexe précédente.

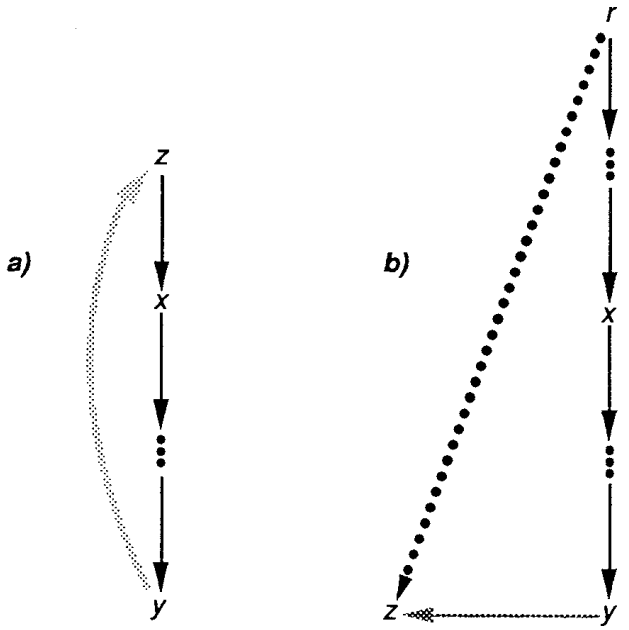


Figure 24. *prefixe*(z) < *prefixe*(x).

On est donc amené à donner la définition suivante de la fonction *retour* :

$$retour(x) = \min\{prefixe(x), prefixe(z), retour(y)\},$$

où le minimum est calculé sur tous les sommets z tels que $x \rightarrow z$ est un arc en arrière, ou un arc croisé (si la racine de la composante fortement connexe de z est un ancêtre de x), et sur tous les sommets y descendants de x dans la forêt couvrante.

Notons que la valeur *prefixe*(x) correspond au cas où il n'y a pas de sommet z tel que *prefixe*(z) est inférieur à *prefixe*(x); la valeur *prefixe*(z) avec $x \rightarrow z$ arc en arrière, ou arc croisé, correspond au cas où le chemin ne comprend pas d'arcs couvrants; et la valeur *retour*(y) correspond au cas où le chemin comprend au moins un arc couvrant $x \rightarrow y$ et où l'on se ramène à la recherche d'un chemin convenable à partir de y.

2.2.2. Algorithme

L'évaluation de *retour*(x) se fait de façon récursive en effectuant un parcours en profondeur du graphe à partir du sommet x. On empile les sommets au fur et à mesure qu'on les rencontre (ils sont donc en ordre préfixe). Lorsqu'on trouve un sommet y dont on peut évaluer la valeur de retour (on a calculé la valeur de retour de tous ses descendants), et tel que *retour*(y) = *prefixe*(y) alors y est la racine r_i d'une composante fortement connexe C_i (voir la justification plus loin). On peut obtenir tous les sommets de C_i en dépilant les sommets empilés jusqu'à r_i, r_i compris. Les sommets dépilés reçoivent alors la valeur $n+1$ en numérotation préfixe, si le graphe a n sommets, afin qu'ils ne soient plus considérés lors de la recherche des autres composantes fortement connexes.

Dans l'algorithme qui suit (voir page suivante), on utilise les opérations sur les piles, et les opérations sur les graphes spécifiées au chapitre 8. On stocke dans le tableau d'entiers *comp*, à l'indice i, le numéro de la composante fortement connexe du sommet i. On stocke dans *pref*[i] le numéro de i dans la numérotation préfixe, et dans *ret*[i] le sommet *retour*(i), où *pref* et *ret* sont des tableaux d'entiers de taille n.

2.2.3. Exemple

On reprend le graphe G de la figure 16, dont la forêt couvrante est dessinée à la figure 17. A côté de chaque sommet x, on a indiqué sur la figure 25 la valeur de *ret*[x] en éclairé, et la valeur de *pref*[x] en italique. (On n'a pas tenu compte des instructions *pref*[x] := $n+1$.)

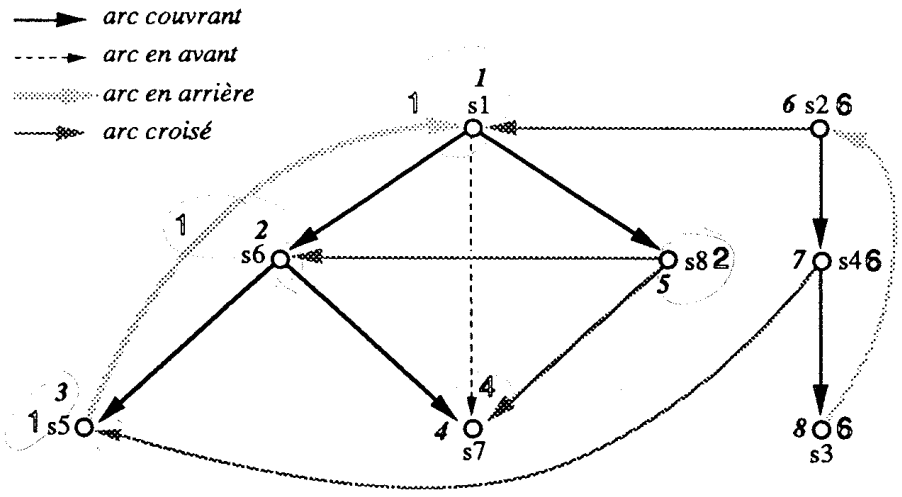


Figure 25. Forêt couvrante de G avec numérotations préfixe et de retour.

Programme principal :
{Le graphe G a n sommets ; cpt, m et i sont des variables de type integer ;
P est une variable de type Pile}
cpt := 0 ; *P* := *pile-vide* ; *m* := 0 ;
{m désigne le nombre de composantes fortement connexes déjà trouvées}
for *i* := 1 **to** *n* **do** *pref*[*i*] := 0 ;
for *i* := 1 **to** *n* **do**
  if *pref*[*i*] = 0 **then** *comp-fort-connexe2*(*i*, *G*, *cpt*, *m*, *P*, *pref*, *ret*, *comp*) ;

procedure *comp-fort-connexe2*(*x* : integer ; *G* : Graphe ; **var** *cpt*, *m* : integer ;
var *P* : Pile ; **var** *pref, ret, comp* : **array**[1..*n*] **of** integer) ;
{Après exécution de la procédure comp-fort-connexe1, comp[*i*] *contient le*
numéro de la composante fortement connexe du sommet i ; x est un sommet}
var *y*, *j*, *min* : integer ;
{y est un sommet et min est la valeur provisoire de retour(x)}

begin
  {numérotation préfixe de x :}
  cpt := *cpt* + 1 ; *pref*[*x*] := *cpt* ;
  min := *cpt* ; *{ret*[*x*] *est inférieur ou égal à pref*[*x*]*}*
  P := *empiler*(*P*, *x*) ;
  for *j* :=1 **to** d^{o+} *de x dans G* **do begin**
    y := *j ème-succ-de x dans G* ;
    if *pref*[*y*] = 0 **then begin** *{première rencontre de y}*
      comp-fort-connexe2(*y*, *G*, *cpt*, *m*, *P*, *pref*, *ret*, *comp*) ;
      if *ret*[*y*] < *min* **then** *min* := *ret*[*y*]
    end
    else *{y a déjà été rencontré : x → y est un arc en arrière*
      ou un arc croisé}
      if *pref*[*y*] < *min* **then** *min* := *pref*[*y*]
        {si pref[*y*] = *n* + 1, *c'est que le sommet y a déjà été*
        mis dans une autre composante fortement connexe}
  end ;
  ret[*x*] := *min* ;
  if *ret*[*x*] = *pref*[*x*] **then begin**
    {on a obtenu une composante fortement connexe}
    m := *m* + 1 ;
    while *sommet*(*P*) <> *x* **do begin**
      y := *sommet*(*P*) ; *comp*[*y*] := *m* ;
      pref[*y*] := *n* + 1 ; *P* := *dépiler* (*P*)
    end ;
    pref[*x*] := *n* + 1 ; *comp*[*x*] := *m* ; *P* := *dépiler*(*P*)
  end
end *comp-fort-connexe2* ;

Dans le parcours en profondeur de $G, s7$ est le premier sommet rencontré dont on connaisse la valeur de retour et qui soit tel que cette valeur de retour soit égale à sa numérotation préfixe. Comme $s7$ n'a pas de descendant, il est l'unique sommet d'une composante fortement connexe dont il est la racine.

Ensuite, on rencontre le sommet $s1$ qui est tel que $ret[1] = pref[1]$. Le sommet $s1$ et ceux de ses descendants dans la forêt qui n'ont pas été déjà mis dans une composante fortement connexe, constituent une autre composante fortement connexe de racine $s1$: $s1, s6, s5$ et $s8$ sont dans la même composante.

Enfin, on rencontre le sommet $s2$ tel que $ret[2] = pref[2]$. Les sommets $s2, s4$ et $s3$ engendrent la troisième composante fortement connexe de racine $s2$. On retrouve les composantes fortement connexes données à la figure 20.

2.2.4. Justification de l'algorithme

On va montrer qu'un sommet x est racine d'une composante fortement connexe si, et seulement si, $retour(x) = prefixe(x)$.

1) On suppose que x est racine d'une composante fortement connexe C_i. Par définition de la fonction *retour*, $retour(x) \leq prefixe(x)$. Raisonnons par l'absurde, en supposant que l'inégalité est stricte. Il existe alors des sommets y, z et r tels que :

a) y est descendant de x,
b) z est atteint à partir de y par un arc croisé ou un arc en arrière,
c) r est la racine de la composante fortement connexe de z,
d) r est un ancêtre de x, racine de C_i,
e) $prefixe(z) < prefixe(x)$.

On en déduit que :

$prefixe(r) \leq prefixe(z)$ d'après c).

D'où : $prefixe(r) < prefixe(x)$ d'après e).

Par suite, r est un ancêtre de x, distinct de x d'après d).

Il existe donc un chemin de r vers x. Mais il y a aussi un chemin de x vers z. Comme r et z sont dans la même composante fortement connexe, il y a aussi un chemin de z vers r. Donc x et r sont tous les deux dans C_i ; or, $prefixe(r)$ étant strictement inférieur à $prefixe(x), x$ ne peut pas être racine de C_i, d'où la contradiction.

2) On suppose que $retour(x) = prefixe(x)$. Raisonnons par l'absurde en supposant que x n'est pas la racine de la composante fortement connexe C_i.

Soit r_i la racine de C_i : r_i est un ancêtre de x, distinct de x. Comme x et r_i sont tous deux dans C_i, il existe un chemin CH de x vers r_i. Considérons le premier arc de CH qui aille d'un sommet y descendant de x, vers un sommet z qui ne soit pas un descendant de x : cet arc est soit un arc croisé, soit un arc en arrière. Dans

tous les cas, $prefixe(z) < prefixe(x)$. Comme r_i est un ancêtre de x, il y a un chemin de r_i vers x; le chemin CH va de x vers r_i, en passant par z. Donc z et r_i sont dans la même composante fortement connexe et z doit être considéré pour calculer $retour(x)$: $retour(x) \leq prefixe(z) < prefixe(x)$.

D'où la contradiction.

La vérification du fait que la procédure *comp-fort-connexe2* calcule correctement la fonction *retour*, est laissée en exercice.

2.3. Comparaison

On a présenté dans ce paragraphe deux algorithmes pour la recherche des composantes fortement connexes qui sont du même ordre de complexité, puisqu'ils effectuent des parcours en profondeur. Le premier effectue deux parcours en profondeur simples et une inversion du graphe. Le deuxième effectue un parcours en profondeur complexe et implique l'usage d'une pile supplémentaire de taille n au pire. L'inversion du graphe n'est significative pour la complexité que si le graphe est représenté par des listes d'adjacence. Dans le cas où le graphe peut être représenté par une matrice d'adjacence, c'est le premier algorithme qui peut être recommandé.

Ces algorithmes sur la forte connexité montrent bien la difficulté de choisir une représentation bien adaptée à un graphe et à un algorithme donnés.

3. 2-connexité

Dans un réseau de communication, il est intéressant lorsqu'un nœud tombe en panne, que les autres nœuds continuent à communiquer. Cette propriété vérifiée par les graphes non orientés qui restent connexes, même quand on leur enlève un sommet, s'appelle la 2-connexité. Puisque cette propriété est plus forte que la connexité, dans tout ce paragraphe, les graphes sont supposés non orientés connexes.

La propriété de 2-connexité est utilisée dans les problèmes de vulnérabilité dans les réseaux. Il arrive souvent aussi, par exemple dans le domaine de la reconnaissance des formes, que l'on décompose un problème sur un graphe en sous-problèmes sur ses composantes 2-connexes.

3.1. Définitions et propriétés

• Soit G un graphe (connexe non orienté). On dit que G est ***2-connexe*** s'il contient au moins trois sommets et si pour tout sommet s, le graphe obtenu en retirant le sommet s est encore connexe.

• Soit s un sommet de G. On dit que s est un ***point d'articulation*** de G si le graphe obtenu en retirant le sommet s n'est plus connexe.

Cela revient à dire qu'il existe deux sommets x et y, tels que toute chaîne entre x et y, passe par s.

Il est clair qu'un graphe 2-connexe n'a pas de point d'articulation.

Le graphe de la figure 26 est 2-connexe, celui de la figure 27 ne l'est pas car $s6$ est point d'articulation.

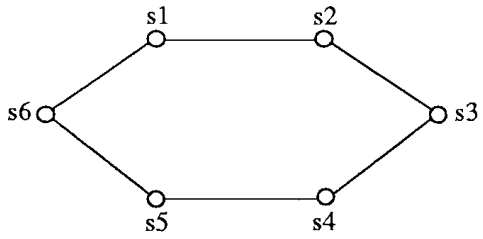


Figure 26. Graphe 2-connexe.

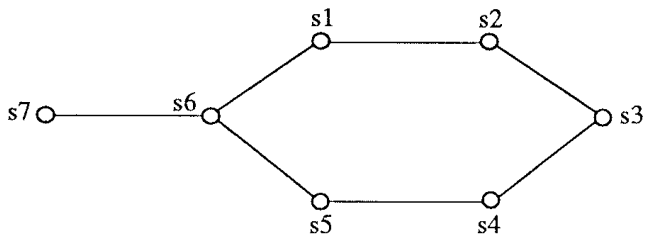


Figure 27. $s6$ est point d'articulation.

Pour définir les composantes 2-connexes, on introduit la notion de bloc.

• Un ***bloc*** est soit une arête, soit un graphe 2-connexe.

• On appelle ***composante 2-connexe*** d'un graphe G tout bloc maximal de G, c'est-à-dire, tout bloc qui n'est pas strictement inclus dans un autre bloc de G.

Exemple : Les points d'articulation du graphe de la figure 28 sont encerclés et ses composantes 2-connexes sont données dans la figure 29.

La construction des composantes 2-connexes d'un graphe repose sur les propriétés suivantes dont la preuve est laissée en exercice :

1) Les composantes 2-connexes forment une partition de l'ensemble des arêtes.

2) Deux composantes 2-connexes sont disjointes ou ont en commun un point d'articulation.

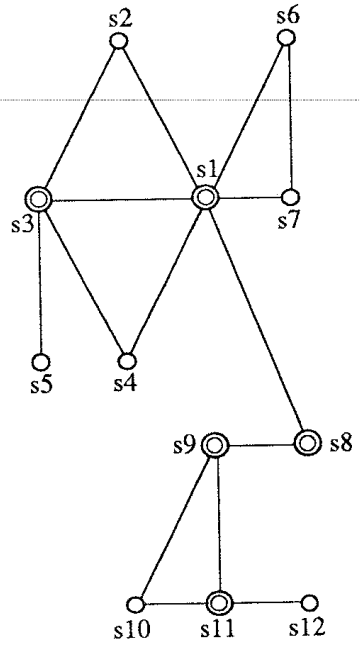


Figure 28. Points d'articulation d'un graphe connexe non orienté.

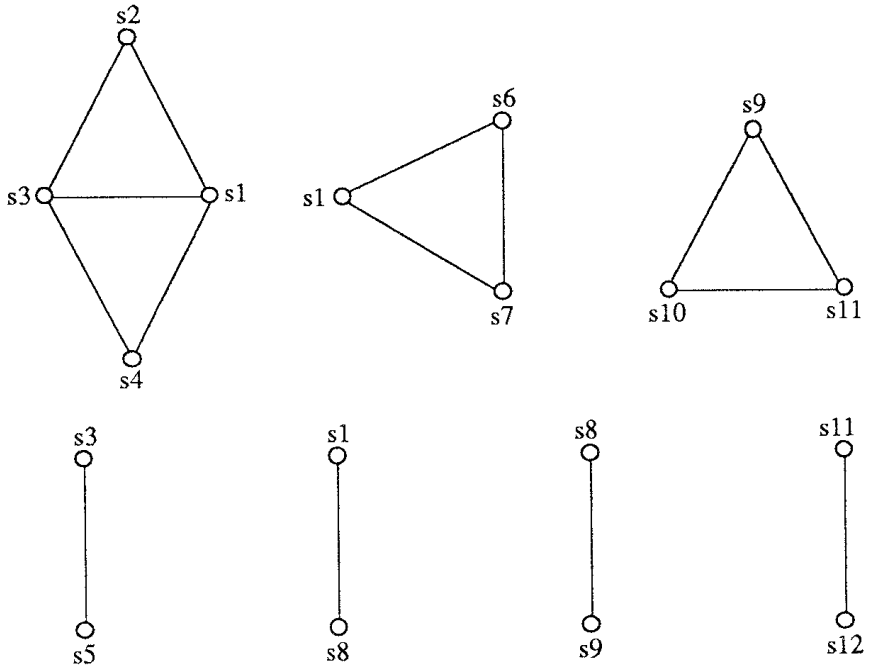


Figure 29. Composantes 2-connexes du graphe de la figure 28.

Dans la suite, on présente un algorithme de recherche des points d'articulation d'un graphe. La modification de cet algorithme pour construire les composantes 2-connexes d'un graphe est reportée en exercice.

3.2. Principe de la recherche des points d'articulation

La détection des points d'articulation se fait en examinant certaines propriétés des sommets dans la forêt couvrante associée au parcours en profondeur du graphe. Rappelons que cette forêt est une arborescence, puisque le graphe est connexe et qu'elle ne comporte pas d'arcs croisés (ni d'arcs en avant), vu que le graphe est non orienté. La figure 30 présente la forêt couvrante correspondant au graphe de la figure 28.

La racine $s1$ de l'arborescence couvrante a au moins deux fils dans l'arborescence : $s2$ et $s6$, par exemple; pour aller de $s2$ à $s6$, il faut passer par $s1$, donc $s1$ est un point d'articulation.

Le sommet $s2$ n'est pas un point d'articulation : sa suppression ne coupe pas le sommet $s3$ du sommet $s1$, en raison de la présence de l'arc en arrière de $s3$ vers $s1$. En revanche, le sommet $s3$ est point d'articulation car sa suppression coupe le sommet $s5$ des sommets $s1$ et $s2$: il n'y a pas de possibilité d'atteindre des sommets ancêtres de $s5$ par l'intermédiaire d'arcs en arrière.

Lorsqu'on supprime un sommet x, on supprime une chaîne entre les ancêtres de x et ses descendants dans l'arborescence. A-t-on perdu la connexité ? Le problème se ramène à savoir si on peut trouver pour chacun des fils y de x une chaîne entre y et l'un des ancêtres de x, dans $G - \{x\}$. Cette chaîne sera alors formée d'arcs couvrants situés en dessous de x et d'un arc en arrière.

Une telle possibilité va être testée à l'aide de la fonction *plushaut* qui indique, pour chaque sommet x le numéro en ordre préfixe du sommet le plus haut de l'arborescence qui peut être atteint à partir de x, en suivant un chemin, (éventuellement de longueur nulle), formé d'arcs couvrants, suivi (éventuellement) d'un arc en arrière. La numérotation en ordre préfixe est donnée par la fonction *prefixe*, comme au paragraphe précédent.

On obtient la définition récursive suivante de la fonction *plushaut* :

$$\mathit{plushaut}(x) = \min \{ \mathit{prefixe}(x), \mathit{prefixe}(y), \mathit{plushaut}(z) \},$$

où le minimum est calculé sur tous les sommets y tels que $x \rightarrow y$ est un arc en arrière et les sommets z tels que $x \rightarrow z$ est un arc couvrant.

Notons que, dans la définition précédente, la valeur $\mathit{prefixe}(x)$ correspond à un chemin de longueur nulle; la valeur $\mathit{prefixe}(y)$ correspond à un chemin ne comprenant aucun arc couvrant mais un arc en arrière; et la valeur $\mathit{plushaut}(z)$ correspond au cas où le chemin comprend au moins un arc couvrant $x \rightarrow z$ et où l'on se ramène

à la recherche d'un chemin convenable à partir de z.

Cette définition rappelle celle de la fonction *retour* du paragraphe précédent; elle en diffère essentiellement par l'absence des arcs croisés.

Le calcul des fonctions *prefixe* et *plushaut* se fait lors du parcours en profondeur du graphe dans la procédure *parcours*. Les valeurs de *prefixe* sont obtenues lors de la première rencontre du sommet. Comme *plushaut*(x) est calculée en fonction des valeurs de *plushaut* et de *prefixe* pour ses successeurs dans le graphe, les valeurs de *plushaut* ne peuvent être calculées que lors de la dernière rencontre du sommet (ordre suffixe). Les valeurs de *prefixe* et *plushaut* sont indiquées à côté des sommets, sur la figure 30.

A l'aide de ces deux fonctions, on peut reconnaître si un sommet est un point d'articulation (voir plus loin la justification).

• Soit r la racine de l'arborescence; si elle a au moins deux fils dans l'arborescence, alors r est un point d'articulation.

La numérotation préfixe est indiquée en italique.
Les valeurs de *plushaut* sont indiquées en éclairé.
Les points d'articulation sont encerclés.

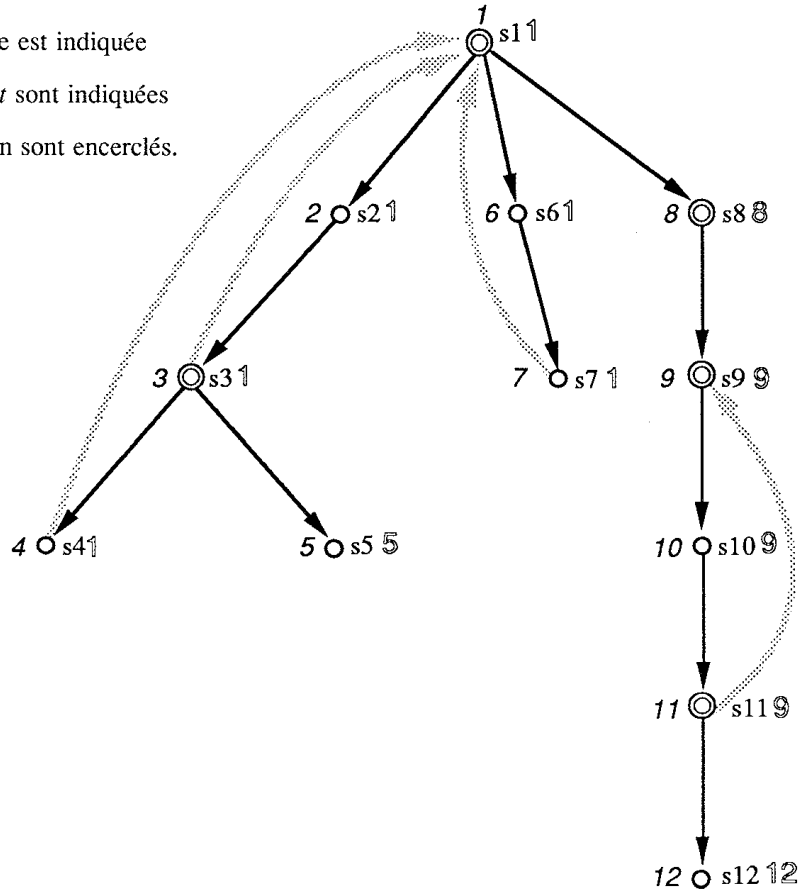


Figure 30. Forêt couvrante associée au graphe de la figure 28.

• Soit x un sommet qui n'est pas la racine de l'arborescence; s'il existe un sommet y tel que $x \rightarrow y$ est un arc couvrant et que *plushaut*(y) est supérieur ou égal à *prefixe*(x), alors x est un point d'articulation (on ne peut atteindre de sommet ancêtre de x depuis y).

Sur la figure 30, les points d'articulation sont encerclés.

3.3. Algorithme

Dans la procédure de parcours qui suit, on utilise les types abstraits Graphe et Liste ainsi que le type tab suivant :

type tab = **array** [1..n] **of** integer;

Les valeurs des fonctions *prefixe* et *plushaut* sont stockées dans les tableaux *pref* et *plushaut* de type tab. On n'a pas de tableau booléen de marques, on utilise le tableau *pref*; *pref*[i] = 0 si, et seulement si, le sommet i n'est pas encore visité.

```
procedure point-d-articulation(G : Graphe; var L : Liste);
{la procédure point-d-articulation construit la liste L des points d'articulation
du graphe non orienté G de n sommets (n ≥ 3), supposé connexe}
var r, x, appel, cpt, i, k : integer; pref, plushaut, père : tab;
{r et x sont des sommets; appel sert pour compter les fils de la racine dans
l'arborescence}

procedure parcours(x : integer; G : Graphe; var pref : tab; var plushaut : tab;
                   var père : tab; var cpt : integer; var L : Liste; var i : integer);
{cpt sert pour la numérotation préfixe des sommets; i est le nombre de points
d'articulation déjà détectés; le tableau père indique les pères des sommets
dans l'arborescence couvrante}
var y, j, ph : integer; detecte : boolean;
{x et y sont des sommets; ph donne la valeur provisoire de plushaut[x]; detecte
indique si on a déjà reconnu x comme point d'articulation}
begin
    {initialisations}
    pref[x] := cpt; ph := cpt; cpt := cpt + 1; detecte := false;
    for j := 1 to d° de x dans G do begin
            {on regarde les successeurs de x}
            y := j ème-succ-de x dans G;
            if pref[y] = 0 then begin
                    {première rencontre de y : x → y est un arc couvrant}
                    père[y] := x;
                    parcours(y, G, pref, plushaut, père, cpt, L, i);
                    if plushaut[y] < ph then ph := plushaut[y];
```

```
                if (not detecte) and (plushaut[y] ≥ pref[x]) then begin
                        detecte := true;
                        i := i + 1; {x est un point d'articulation}
                        L := insérer(L, i, x)
                    end
            end
            else {y a déjà été rencontré ; si ce n'est pas le père de x, alors
                x → y est un arc en arrière}
            if (y <> père[x]) and (pref[y] < ph) then ph := pref[y]
    end;
    plushaut[x] := ph
end parcours;
begin
    {initialisations}
    for k := 1 to n do begin
            pref[k] := 0; père[k] := 0; plushaut[k] := 0
    end;
    cpt := 1; L := liste-vide; i := 0; r := 1; appel := 0;
    pref[r] := cpt; cpt := cpt + 1;
    plushaut[r] := pref[r];
    for k := 1 to d° de r dans G do begin
            x := k ème-succ-de r dans G;
            if pref[x] = 0 then begin
                    père[x] := r;
                    parcours(r, G, pref, plushaut, père, cpt, L, i);
                    appel := appel + 1
                end
     end;
     if appel ≥ 2 then begin {la racine est point d'articulation}
            i := i + 1;
            L := insérer(L, i, r)
     end
end point-d-articulation;
```

Notons que, si après exécution de la procédure la liste L est vide, c'est que le graphe G est 2-connexe.

Cette procédure peut être modifiée, de manière à obtenir les composantes 2-connexes (cf. exercices).

La complexité de cet algorithme est exactement celle du parcours en profondeur. Pour un graphe ayant n sommets et p arcs, on a une complexité en $\Theta(n^2)$ si le graphe est représenté par une matrice d'adjacence, et en $\Theta(\max(n,p))$ si le graphe est représenté par des listes d'adjacence.

3.4. Justification

On prouve la caractérisation des points d'articulation qui a été utilisée dans l'algorithme.

Propriété : Soit G un graphe non orienté connexe. Soit T l'arborescence couvrante associée au parcours en profondeur et soit x un sommet de G.

• Si x est la racine r de T, x est point d'articulation si, et seulement si, x a au moins deux fils dans l'arborescence.

• Si x n'est pas racine de T, x est point d'articulation si, et seulement si, il existe un sommet y tel que $x \rightarrow y$ est un arc couvrant et *plushaut*$(y) \geq$ *prefixe*(x).

Preuve

Premier cas : x est la racine r de l'arborescence couvrante T.

• Supposons que r a au moins deux fils y_1 et y_2 dans T. Comme y_2 n'appartient pas à la sous-arborescence de racine y_1, les deux sous-arborescences de racines y_1 et y_2 sont disjointes (cf. figure 31) et comme il n'y a pas d'arc croisé (graphe non orienté), la seule chaîne possible entre y_1 et y_2 passe par r.

• Réciproquement, supposons que x est point d'articulation. Si x avait un seul fils, on pourrait supprimer x, sans déconnecter le graphe ; donc G a nécessairement au moins deux fils.

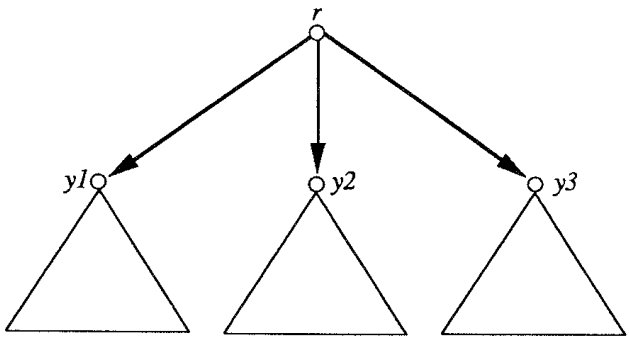


Figure 31. La racine r est point d'articulation.

Deuxième cas : x n'est pas la racine r de l'arborescence couvrante T.

• Supposons que x est point d'articulation.

$G - \{x\}$ n'est pas connexe : soit $C_1, C_2, ..., C_k$ ses composantes connexes (cf. figure 32). Supposons que r appartient à C_1, par exemple. Il existe y appartenant à C_2 tel que $x \rightarrow y$ est un arc couvrant de T et donc *prefixe*$(y) \geq$ *prefixe*(x). Montrons que *plushaut*(y) est supérieur ou égal à *prefixe*(x). Tout chemin issu de y et comportant des arcs couvrants de T, éventuellement suivis d'un arc en arrière de T, correspond

à une chaîne dans G issue de y : ce chemin ne peut aboutir qu'à un sommet u de la même composante connexe que y, (c'est-à-dire, u appartient à C_2), ou au sommet x. On a donc soit *plushaut*(y) = *prefixe*(u), soit *plushaut*(y) = *prefixe*(x). Comme tous les sommets de C_2 sont visités après x, on a bien *prefixe*$(u) \geq$ *prefixe*(x).

Réciproquement, supposons qu'il existe un sommet y tel que $x \rightarrow y$ est un arc couvrant et *plushaut*$(y) \geq$ *prefixe*(x).

Comme x n'est pas la racine r, il existe un sommet z visité avant x (éventuellement $z = r$), tel que *prefixe*$(z) <$ *prefixe*(x) et $z \rightarrow x$ est un arc couvrant. Pour montrer que x est un point d'articulation, il suffit de montrer qu'il n'y a pas de chaîne entre y et z dans $G - \{x\}$. Pour cela, on va montrer que toute chaîne issue de y dans $G - \{x\}$ aboutit à un sommet t tel que *prefixe*$(t) \geq$ *prefixe*(y). Comme *prefixe*$(y) >$ *prefixe*$(x) >$ *prefixe*(z), le sommet t ne peut être z. Considérons donc une chaîne de $G - \{x\}$ issue de y allant vers t. Il lui correspond dans T un chemin Γ comprenant des arcs couvrants et des arcs en arrière. Deux cas se présentent.

(i) Ce chemin Γ est formé uniquement d'arcs couvrants; dans ce cas, t est nécessairement dans la sous-arborescence de racine y et *prefixe*$(t) \geq$ *prefixe*(y).

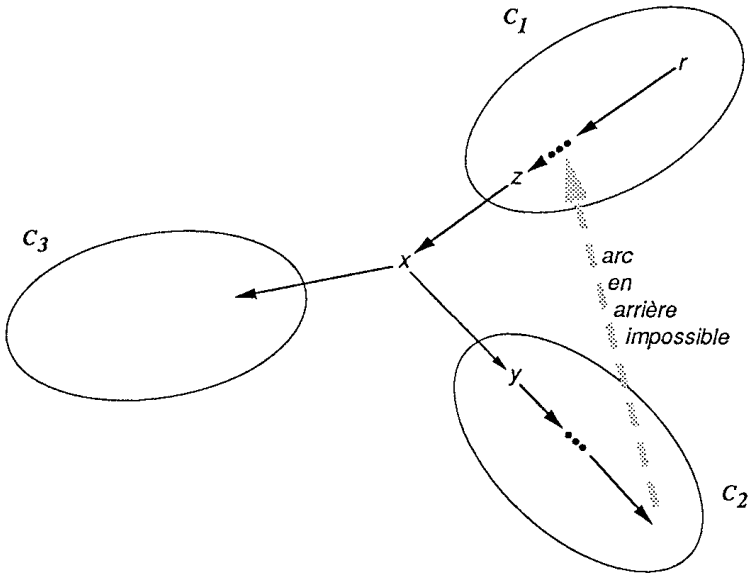


Figure 32. Composantes connexes de G - $\{x\}$: C_1, C_2 et C_3.

(ii) Ce chemin Γ comprend au moins un arc en arrière. Soit $u \rightarrow v$ le premier arc en arrière de ce chemin (figure 33). (Notons que $u = y$ est possible). On a : *prefixe*$(v) <$ *prefixe*(u) et *prefixe*$(v) \geq$ *plushaut*(y), par définition de la fonction *plushaut* pour y. Par ailleurs, on ne peut avoir *prefixe*$(v) <$ *prefixe*(x) car *plushaut*$(y) \geq$ *prefixe*(x) ; de plus, v est distinct de x car on considère une chaîne dans $G - \{x\}$. Par suite, *prefixe*$(v) >$ *prefixe*(x). Quel que soit u descendant de y dans l'arborescence, un arc en arrière $u \rightarrow v$ ne peut atteindre que des sommets v tels que *prefixe*$(v) \geq$

prefixe(y); s'il y a d'autres arcs couvrants dans le chemin Γ, la valeur de *prefixe*(t) ne peut qu'augmenter, et un autre arc en arrière ne peut abaisser cette valeur en dessous de *prefixe*(y). En conclusion, toute chaîne aboutit à un sommet t tel que *prefixe*$(t) \geq$ *prefixe*(y). Donc z ne peut être atteint à partir de y, et x est un point d'articulation. □

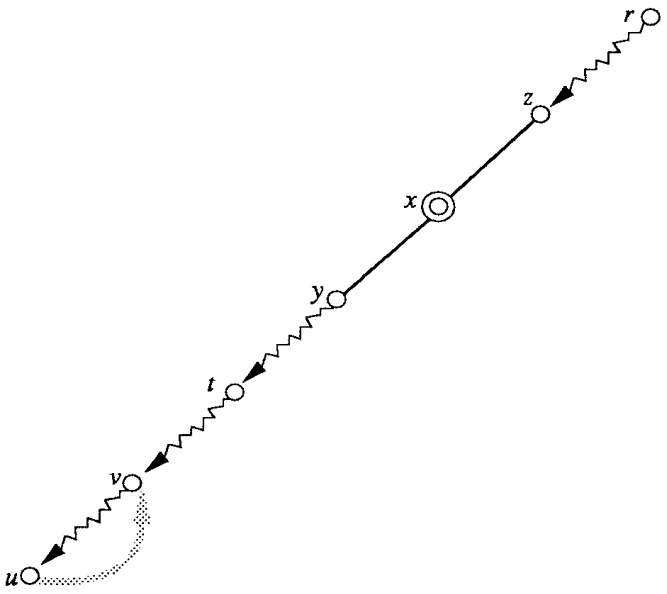


Figure 33. x est un point d'articulation et n'est pas la racine de T.

Exercices

1. Soit M une matrice carrée booléenne d'ordre n. On note M^* la matrice obtenue par application de la procédure *Warshall* sur M.

a) Donner un exemple de matrice M dans laquelle n éléments seulement sont égaux à 1 mais telle que tout élément de M^* est égal à 1.

b) Donner un exemple de matrice M dans laquelle tous les éléments, sauf $(n-1)$, sont égaux à 1, mais tel qu'il existe au moins un élément de M^* différent de 1.

c) En utilisant les idées sous-jacentes à la résolution de a) et b), donner deux algorithmes qui, appliqués à la matrice d'adjacence M d'un graphe orienté G égal à sa fermeture transitive, déterminent en un temps $\Theta(n)$ si G est fortement connexe ou non.

2. Donner les forêts obtenues par unions successives pour la suite d'arêtes :

2—1, 4—3, 6—5, 8—7, 10—9, 12—11, 2—11, 4—9, 6—7, 11—7, 1— 5, 8—9,

dans chacun des cas suivants :

a) en utilisant la fonction *trouver* et la procédure *réunir*,

b) en utilisant la fonction *trouver* et la procédure *réunir-pondéré*,

c) en utilisant les procédures *trouver-rapide* et *réunir-pondéré*.

3. Montrer qu'on peut construire par unions pondérées successives pour tout m, un arbre de m nœuds ayant une profondeur égale à $\lfloor \log_2 m \rfloor$. Dans le cas où il s'agit d'un graphe G ayant $m = 16$ sommets, donner une suite d'arêtes possibles, qui réalise cette construction.

4. Examiner chacune des assertions suivantes : la prouver si elle est exacte et expliquer pourquoi elle est fausse sinon.

a) La réunion des arêtes des composantes connexes d'un graphe non orienté $G_1 = < S_1, A_1 >$ est égale à A_1.

b) La réunion des arcs des composantes fortement connexes d'un graphe orienté $G_2 = < S_2, A_2 >$ est égale à A_2.

5. Combien y-a-t-il de composantes fortement connexes dans un graphe orienté, sans circuit, de n sommets ? Même question avec un graphe orienté de n sommets ayant un circuit élémentaire de longueur n.

6. On appelle ***graphe réduit*** d'un graphe orienté G ayant m composantes fortement connexes $C_1, ..., C_m$, le graphe G' qui a m sommets $x_1, ..., x_m$ tels qu'il existe un arc de x_i vers x_j dans G', si, et seulement si, il existe un sommet x dans C_i, un sommet y dans C_j et un arc $x \rightarrow y$ dans G.

Montrer que le graphe réduit G' d'un graphe orienté G est sans circuit.

7. On appelle ***forêt maximale*** d'un graphe non orienté G, un graphe partiel de G, sans cycle, maximal pour l'inclusion. Pour construire une forêt maximale de G, on propose l'algorithme de coloriage suivant.

Initialement, toutes les arêtes du graphe sont incolores. On examine successivement toutes les arêtes du graphe, en les coloriant soit en rouge, soit en vert. Soit u une arête incolore, non encore examinée.

i) S'il passe par u un cycle élémentaire dont toutes les arêtes (sauf u) sont rouges, on colore u en vert.

ii) S'il n'existe pas de tel cycle, on colore u en rouge.

a) Montrer que le graphe partiel engendré par les arêtes rouges est une forêt maximale de G.

b) Montrer que si G est un graphe non orienté de n sommets et m composantes connexes, une forêt maximale de G comporte exactement $n - m$ arêtes.

8. Trouver les composantes fortement connexes du graphe donné à la figure 34 :

a) en utilisant le parcours en profondeur du graphe et de son inverse (*comp-fort-connexe1*)

b) en utilisant l'algorithme de Tarjan (*comp-fort-connexe2*).

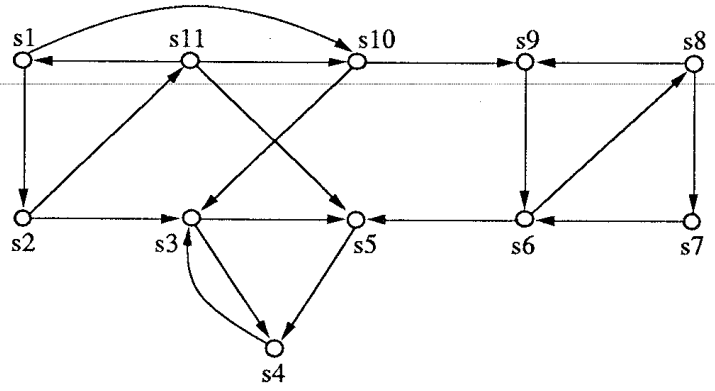


Figure 34

9. Implémenter l'algorithme de Tarjan (*comp-fort-connexe2*) en représentant le graphe par des listes d'adjacence. Evaluer la complexité en temps de ce programme.

10. Prouver que la procédure *comp-fort-connexe2* calcule correctement la fonction *retour*.

11. Prouver la propriété suivante (théorème de Menger).

Soit G un graphe non orienté ayant au moins trois sommets. G est 2-connexe si, et seulement si, pour tous sommets x et y de G, il existe 2 chaînes reliant x et y qui n'ont pas d'autre sommet en commun que x et y.

12. Soit G un graphe non orienté connexe, ayant au moins trois sommets. Montrer que les propriétés suivantes sont équivalentes :

(1) G est 2-connexe.

(2) Pour tous sommets x et y de G, il existe un cycle passant par x et par y.

(3) Pour tout sommet x de G et pour toute arête u de G, il existe un cycle contenant x et u.

(4) Pour toutes arêtes $u1$ et $u2$ de G, il existe un cycle contenant $u1$ et $u2$.

13. Le but de l'exercice est d'écrire un algorithme qui construise les composantes 2-connexes d'un graphe G non orienté connexe.

a) Prouver les propriétés suivantes (on pourra utiliser l'exercice 2) :

(1) Les composantes 2-connexes de G forment une partition de l'ensemble des arêtes de G.

(2) Deux composantes 2-connexes de G distinctes sont disjointes ou ont en commun un point d'articulation.

b) Modifier la procédure *point-d-articulation* de façon à obtenir les arêtes des composantes 2-connexes, en utilisant les propriétés ci-dessus. Pour cela, on empile les arêtes au fur et à mesure qu'on les rencontre dans un parcours en profondeur du graphe. Chaque fois qu'on rencontre une arête x—y (qui correspond à un arc couvrant $x \to y$) telle que $\mathit{retour}(y) \geq \mathit{prefixe}(x)$, on dépile toutes les arêtes restant sur le dessus de la pile, jusqu'à l'arête x—y incluse : elles forment une composante 2-connexe.

14. Une propriété voisine de la 2-connexité est la ***2-arête-connexité***.

Soit G un graphe non orienté ayant au moins une arête. On dit que G est 2-arête-connexe si quelle que soit l'arête u qu'on enlève à $G, G - \{u\}$ est encore connexe. Montrer les propriétés suivantes :

(1) Si G est 2-connexe alors G est 2-arête-connexe.
(2) Si G est 2-arête-connexe, alors tout sommet est de degré supérieur ou égal à 2.

15. Soit G un graphe non orienté connexe. On appelle ***isthme*** de G une arête u telle que $G - \{u\}$ est non connexe.

a) Soit T l'arborescence couvrante associée au parcours en profondeur de G. Montrer que x—y est un isthme de G si, et seulement si, $x \to y$ est un arc couvrant de T, avec $\mathit{retour}(y) \geq \mathit{prefixe}(y)$ ou si $y \to x$ est un arc couvrant de T avec $\mathit{retour}(x) \geq \mathit{prefixe}(x)$.

b) En déduire un algorithme de détection des isthmes.

Chapitre 20
Plus courts chemins

On présente dans ce chapitre un problème typique de cheminement dans les graphes : la recherche d'un plus court chemin entre deux sommets. Ce problème a de nombreuses applications. On en a déjà vu un exemple au chapitre 8 : trouver le moyen le plus économique pour aller de Brest à Lyon, connaissant pour chaque ligne aérienne le prix du billet d'avion. Citons quelques autres applications : les problèmes d'optimisation de réseaux (réseaux routiers ou réseaux de télécommunications), les problèmes d'ordonnancement et les problèmes d'intelligence artificielle tels que la circulation dans un labyrinthe.

1. Définition

1.1. Spécification

On s'intéresse ici aux chemins de longueur non nulle. Formellement, la notion de chemin doit être spécifiée, comme au chapitre 19 à l'aide de l'opération de *fermeture transitive*. Plus précisément, il existe un chemin de x vers y dans le graphe orienté G si, et seulement si, il existe un arc de x vers y dans *fermeture-transitive*(G).

La notion de plus court chemin suppose qu'on travaille sur un graphe valué orienté G. On a donc une fonction *coût* définie sur les couples de sommets (x, y) tels que $x \rightarrow y$ est un arc de G, et à valeurs dans $\mathbb{R}$.

On appelle ***coût cumulé d'un chemin*** μ (ou plus simplement coût ou poids de μ), noté *coût*(μ), la somme des coûts des arcs qui le composent. On appelle ***plus court chemin de x vers y*** un chemin allant de x à y de coût minimum. Un tel chemin n'existe pas toujours. On appelle ***plus petite distance de x à y***, le coût de ce chemin s'il existe.

Afin de pouvoir exprimer les algorithmes de construction des plus courts chemins, on ajoute à la spécification d'un graphe orienté valué, donnée au chapitre 8, les opérations suivantes :

ppdistance : Sommet × Sommet × Graphe → Réel
père : Sommet × Sommet × Graphe → Sommet

père(x, y, G) désigne le sommet de G qui précède y dans un plus court chemin de x à y dans G, et *ppdistance*(x, y, G) est le coût de ce chemin.

Ces deux opérations ne sont pas définies partout, mais seulement pour les couples de sommets (x, y) pour lesquels il existe un plus court chemin de x vers y. Quand elles sont définies, elles satisfont les propriétés suivantes :

(1) pour tout chemin $x_0, x_1, \ldots, x_m$ de G,
$$\sum_{i=0,\ldots,m-1} \textit{coût}(x_i, x_{i+1}, G) \geq \textit{ppdistance}(x_0, x_m, G)$$

(2) pour tout chemin $x_0, x_1, \ldots, x_m$ de G
$(\forall i$ tel que $0 \leq i < m-1$, $x_i = \textit{père}(x_0, x_{i+1}, G)) \Rightarrow$
$$\sum_{i=0,\ldots,m-1} \textit{coût}(x_i, x_{i+1}, G) = \textit{ppdistance}(x_0, x_m, G)$$

Dans ce chapitre, on recherche des plus courts chemins dans un graphe G fixé. Pour simplifier les notations, on omettra, s'il n'y a pas d'ambiguïté, de mentionner G en argument des opérations sur les sommets, les arcs et les chemins.

1.2. Conditions d'existence

A quelles conditions le problème de la recherche d'un plus court chemin admet-il une solution ?

Considérons un chemin μ allant de x à y ; supposons que ce chemin contienne un circuit Γ. Soit μ' le chemin obtenu en supprimant ce circuit de μ. On a :

$$\textit{coût}(\mu) = \textit{coût}(\mu') + \textit{coût}(\Gamma)$$

Si *coût*(Γ) est négatif (strictement inférieur à zéro), il n'existe pas de plus court chemin de x à y, car étant donné un chemin λ de x à y, on peut toujours construire un chemin λ' de coût strictement inférieur à λ, en passant une fois de plus dans Γ.

Si *coût*(Γ) est positif ou nul, μ' est de coût inférieur ou égal à μ ; c'est un meilleur candidat que μ pour être solution du problème si coût(Γ) est strictement positif.

Les circuits de coût négatif sont appelés ***circuits absorbants***. Dans toute la suite de ce paragraphe, on suppose qu'il n'existe pas de tels circuits. S'il existe des circuits, ils sont donc de coût positif ou nul. Or, on vient de voir qu'un plus court chemin ne peut contenir un circuit de coût strictement positif. S'il y a des circuits de coût nul, il y a plusieurs solutions au problème de la recherche d'un plus court chemin.

Comme le nombre de chemins élémentaires de x vers y (rappelons qu'il s'agit de chemins qui ne contiennent pas plusieurs fois un même sommet) est fini, il existe

au moins un plus court chemin élémentaire de x vers y. (Notons qu'il peut exister plusieurs plus courts chemins élémentaires.) Un plus court chemin élémentaire est un plus court chemin de x vers y, s'il n'y a pas de circuit absorbant.

Les algorithmes proposés ici, dans le cas où il n'y a pas de circuit absorbant, donnent des solutions qui sont des ***chemins élémentaires.***

En conclusion, le problème de la recherche d'un plus court chemin de x vers y, admet une solution dès qu'il existe un chemin de x vers y, et qu'il n'y a pas de circuit absorbant.

1.3. Variantes du problème

Le problème de la recherche d'un plus court chemin se rencontre sous l'une des trois formes suivantes :

(i) on recherche un plus court chemin entre deux sommets donnés,

(ii) on recherche les plus courts chemins entre un sommet donné, appelé *source* et tous les autres sommets,

(iii) on recherche les plus courts chemins entre tous les sommets pris deux à deux.

On ne connaît pas de solution meilleure pour le problème (i) que celle qui consiste à passer par les solutions de (ii). On s'intéresse dans les paragraphes suivants aux problèmes (ii) et (iii). Les deux premiers algorithmes présentés résolvent le problème (ii) et le troisième le problème (iii).

Il existe différents algorithmes qui sont plus ou moins bien adaptés selon les propriétés du graphe et la forme du problème étudié ((ii) ou (iii)), mais il n'y a pas d'algorithme général qui soit efficace et intéressant dans tous les cas.

2. Algorithme de Dijkstra (coûts ≥ 0)

Cet algorithme n'est utilisable que dans le cas, très fréquent, où les coûts des arcs sont tous positifs ou nuls. Il calcule un plus court chemin entre une source s et tous les sommets accessibles depuis s : on obtient alors une *arborescence de racine* s formée par ces plus courts chemins.

2.1. Principe de l'algorithme

Soit $G = < S,\ A,\ C >$ un graphe valué orienté de n sommets. Soit s un sommet de G. On construit progressivement un ensemble CC de sommets x pour lesquels on connaît un plus court chemin de s vers x.

Au départ CC ne contient que s. A chaque étape, on ajoute un sommet x à CC pour lequel *ppdistance*(s, x) et *père*(s, x) sont connus. On s'arrête lorsque CC contient tous les sommets accessibles depuis s.

Soit $M = S - CC$. Pour tout sommet y de M on appelle ***chemin de s à y dans CC*** tout chemin de s à y ne comportant que des éléments de CC, sauf y. On note $distance_{s,CC}(y)$ le coût d'un plus court chemin de s à y dans CC et $pred_{s,CC}(y)$ le prédécesseur de y dans ce plus court chemin. A chaque étape, on choisit un sommet m de M tel que :

$$distance_{s,CC}(m) = \inf_{y \in M}\{distance_{s,CC}(y)\}$$

En effet, pour un tel sommet, on montre (voir plus loin la justification de l'algorithme) que :

$distance_{s,CC}(m) = ppdistance(s, m)$
$pred_{s,CC}(m) = père(s, m)$

On supprime m de M et on l'ajoute à CC. Comme m est maintenant élément de CC, pour un sommet y de M, il peut exister de nouveaux chemins de s à y dans CC (ce sont des chemins passant par m tels que m est prédécesseur de y). Pour tout sommet y restant dans M, il faut modifier les valeurs de $distance_{s,CC}(y)$ et de $pred_{s,CC}(y)$ si l'ancienne valeur de $distance_{s,CC}(y)$ est strictement supérieure à $distance_{s,CC}(m)$ + *coût*(m, y).

2.2. Algorithme

On stocke $distance_{s,CC}(i)$ dans la case d'indice i du tableau d, et $pred_{s,CC}(i)$ dans la case d'indice i du tableau pr. Pour tout sommet i, on initialise $pr[i]$ à s et $d[i]$ à *coût*(s, i).

Pour éviter des tests, on prolonge l'opération *coût* pour qu'elle soit définie pour tout couple de sommets (x, y) avec :

coût$(x, x) = 0$ (ce qui est possible car on a supposé les graphes sans boucle)

x *arc* y = faux $\Rightarrow$ *coût*$(x, y) = \infty$, où ∞ désigne une valeur strictement plus grande que tous les coûts du graphe.

On se donne une opération *choisir-min*(M, d) qui a pour résultat un élément m de M tel que $d[m]$ est minimum.

Dans la procédure qui suit, on utilise également les opérations *ensemble-vide*, *supprimer*, *ajouter* et $\in$ sur les ensembles, qui ont été spécifiées au chapitre 6 et les opérations sur les graphes spécifiées au chapitre 8.

```
procedure Dijkstra(s : integer; G : Graphe; var d : array[1..n] of real;
                                  var pr : array[1..n] of integer);
{Le graphe G a n sommets : 1, 2,..., n ; s est un sommet de G ; on recherche un
plus court chemin de s à chacun des autres sommets ; les coûts doivent être positifs
ou nuls ; après exécution, si d[y] < ∞, d[y] est le coût d'un plus court chemin de s
vers y et pr[y] est le prédécesseur de y sur ce chemin }
var i, y, m : integer; v : real; M : Ensemble;
{y et m sont des sommets}


begin
    {initialisations}
    M := ensemble-vide;
    for i := 1 to n do begin
        d[i] := coût(s, i, G);
        pr[i] := s;
        M := ajouter(i, M)
    end;
    M := supprimer(s, M); {CC = {s}}
    {enrichissements successifs de CC :}
    while M <> ensemble-vide do begin
        m := choisir-min(M, d);
        if d[m] = ∞ then return; {car les sommets qui restent dans M sont
        inaccessibles depuis s}
        M := supprimer(m, M); {on ajoute m à CC}
        for i:= 1 to d°+ de m dans G do begin
            {on réajuste les valeurs de distance_{s,CC}(y) et pred_{s,CC}(y) :}
            y := i ème-succ-de m dans G;
            if y ∈ M then begin
                        v := d[m] + coût(m, y, G);
                        if v < d[y] then begin d[y] := v; pr[y] := m end
            end
        end
    end
end Dijkstra;
```

2.3. Exemple

On considère le graphe G orienté valué par des coûts positifs ou nuls de la figure 1. Les nombres au-dessus des arcs indiquent leur coût. On cherche un plus court chemin de $s1$ vers chaque sommet accessible depuis $s1$, ainsi que son coût.

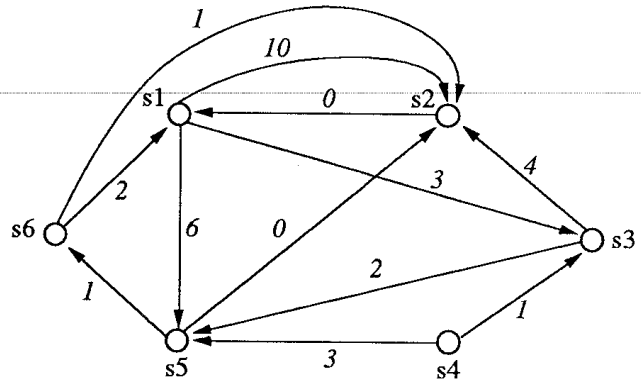


Figure 1. Graphe orienté valué par des coûts positifs ou nuls.

L'exécution de la procédure sur G se fait selon les étapes suivantes :

1) Initialisation : $M = \{s2, s3, s4, s5, s6\}$

d :	0	10	3	∞	6	∞
	1	2	3	4	5	6

pr :	1	1	1	1	1	1
	1	2	3	4	5	6

2) $M = \{s2, s4, s5, s6\}$

d :	0	7	3	∞	5	∞
	1	2	3	4	5	6

pr :	1	3	1	1	3	1
	1	2	3	4	5	6

3) $M = \{s2, s4, s6\}$

d :	0	5	3	∞	5	6
	1	2	3	4	5	6

pr :	1	5	1	1	3	5
	1	2	3	4	5	6

4) $M = \{s4, s6\}$

d :	0	5	3	∞	5	6
	1	2	3	4	5	6

pr :	1	5	1	1	3	5
	1	2	3	4	5	6

5) $M = \{s4\}$

d :	0	5	3	∞	5	6
	1	2	3	4	5	6

pr :	1	5	1	1	3	5
	1	2	3	4	5	6

La procédure s'arrête car *choisir-min*(M, d) $= s4$ et $d[4] = \infty$.

A l'issue de la procédure, on obtient une arborescence de racine s des plus courts chemins de s vers les autres sommets accessibles, «en remontant» dans le tableau pr (voir exercices). Par exemple, un plus court chemin de $s1$ vers $s6$ est :

$$s1 \rightarrow s3 \rightarrow s5 \rightarrow s6 \text{ car } pr[6] = 5,\ pr[5] = 3 \text{ et } pr[3] = 1.$$

On n'a pas trouvé de plus court chemin de $s1$ vers $s4$, ($d[4] = \infty$, à la fin), car $s4$ est inaccessible depuis $s1$.

Remarque : L'hypothèse «les coûts sont tous positifs ou nuls» est fondamentale. L'utilisation de l'algorithme pour le graphe de la figure 2, où les poids ne sont pas tous positifs ou nuls, conduit à un résultat incorrect si on choisit comme source le sommet $s1$. En effet, on ôte d'abord le sommet $s2$ de M, et on n'examine plus $distance_{s1,CC}(s2)$ alors que le chemin de $s1$ vers $s2$ passant par $s3$ est plus court.

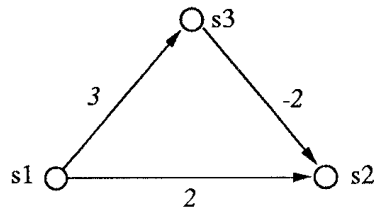


Figure 2. Graphe orienté valué par des coûts quelconques.

2.4. Justification de l'algorithme

Si à la fin de la procédure, pour un sommet i, $d[i] = \infty$, c'est qu'il n'existe pas de chemin s vers i et par suite, pas de plus court chemin de s vers i. Ce cas ne se produit pas si s est une racine du graphe.

On va maintenant montrer que si à la fin de la procédure $d[x] < \infty$, alors $d[x] = ppdistance(s,\ x)$ et $pr[x] = père(s,\ x)$.

Considérons l'ensemble M à une étape donnée. Soit m un sommet de M tel que $distance_{s,CC}(m) = \inf_{y \in M} \{distance_{s,CC}(y)\}$.

$distance_{s,CC}(m)$ est le coût d'un plus court chemin λ de s à m dans CC. Pour montrer que $distance_{s,CC}(m) = ppdistance(s, m)$, il faut montrer qu'il n'existe pas de chemin, qui ne soit pas dans CC, plus court que λ.

Soit μ un chemin de s à m qui n'est pas dans CC ; soit w le premier sommet de μ qui est dans M. Le chemin μ se décompose en deux chemins μ_1 et μ_2 ; μ_1 allant de s à w et μ_2 de w à m (cf. figure 3) ; μ_1 est un chemin de s à w dans CC. Comme m réalise le minimum de $\{distance_{s,CC}(y)\}$ pour y élément de M, $coût(\mu_1) \geq coût(\lambda)$. Or $coût(\mu_2) \geq 0$, puisque ***les coûts sont positifs ou nuls*** (c'est ici que cette

hypothèse intervient). Par conséquent :

$$coût(\mu) = coût(\mu_1) + coût(\mu_2) \geq coût(\lambda).$$

λ est bien un plus court chemin de s vers m :

$$distance_{s,CC}(m) = ppdistance(s,\ m).$$

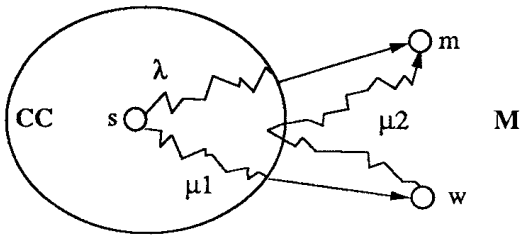


Figure 3. Chemins dans CC.

De plus, $pred_{s,CC}(m)$ qui est le prédécesseur de m dans λ, est bien le prédécesseur de m dans un plus court chemin de s à m :

$$pred_{s,CC}(m) = père(s,\ m).$$

On justifie maintenant les réajustements des tableaux d et pr après chaque modification de CC. Soit M' l'ensemble M privé de m et soit CC' l'ensemble CC auquel on a ajouté m. En ajoutant m à CC on crée la possibilité, pour tout sommet y de M', d'un chemin μ dans CC' plus court que le plus court chemin dans CC. On va montrer qu'un tel chemin μ doit être un chemin dans CC de s vers m suivi de l'arc $m \rightarrow y$.

Considérons un chemin μ dans CC' de s vers y passant par m ; il se décompose en un chemin μ_1, chemin de s à m dans CC, suivi d'un arc de m vers un sommet x de CC, puis d'un chemin μ_2 dans CC vers un sommet z, suivi de l'arc $z \rightarrow y$ (cf. figure 4). Montrons que μ n'est pas plus court que le plus court chemin de s à y dans CC.

Puisque z a été ôté de M avant m, il existe un plus court chemin μ_3 de s à z, qui est dans CC et tel que :

$coût(\mu_3) \leq coût(\mu_1) + coût(m,\ x) + coût(\mu_2)$.

Or $coût(\mu) = coût(\mu_1) + coût(m,\ x) + coût(\mu_2) + coût(z,\ y)$.

Donc $coût(\mu) \geq coût(\mu_3) + coût(z,\ y)$.

Le chemin μ est donc de coût supérieur ou égal au coût d'un plus court chemin de s à y dans CC.

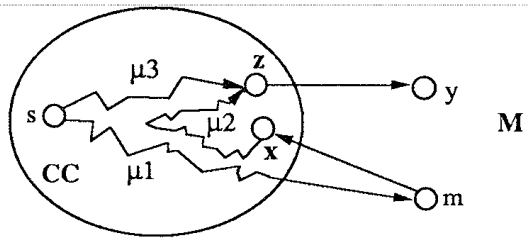


Figure 4. Plus court chemin dans $CC \cup \{m\}$ impossible.

Il en résulte que les seuls chemins de s à y dans CC', qui peuvent être strictement plus courts que le plus court chemin de s à y dans CC, se décomposent en un chemin de s à m dans CC, suivi de l'arc $m \rightarrow y$.

Le plus court de ces chemins a pour coût : $distance_{s,CC}(m)$ + $coût(m, y)$. Si ce coût est strictement inférieur à $distance_{s,CC}(y)$, on a trouvé un chemin de s à y, dans CC', plus court que les chemins de s à y dans CC, et le prédécesseur de y dans ce plus court chemin est m : $pred_{s,CC}(y) = m$.

2.5. Analyse de la complexité

On évalue la complexité au pire de l'algorithme en nombre total d'opérations, en fonction du nombre de sommets n et du nombre d'arcs p du graphe.

• La première boucle, qui correspond à l'initialisation des tableaux, comporte $\Theta(n)$ affectations. L'initialisation de M est également en $\Theta(n)$.

• Il y a au plus $n - 1$ itérations dans la boucle **while**. Le nombre $n - 1$ correspond au cas où s est racine du graphe.

Examinons le coût de l'instruction $m :=$ *choisir-min*(M, d). Son coût au pire est du même ordre que le nombre de comparaisons effectuées.

(i) Si M est représenté par une liste chaînée de longueur $|M|$, la recherche de *choisir-min*(M, d) nécessite $|M| - 1$ comparaisons. Comme $|M|$ diminue de 1 à chaque itération de la boucle **while**, et cela au plus $(n - 1)$ fois, on a au plus $\sum_{i=1}^{n-1}(i - 1)$ comparaisons en tout, soit $\Theta(n^2)$ opérations au pire.

(ii) Si M est représenté par un tableau de n booléens, l'opération *choisir-min* nécessite $n - 1$ comparaisons, soit pour la boucle **while** au plus $(n - 1) \times (n - 1)$ en tout, c'est-à-dire, encore $\Theta(n^2)$ opérations au pire.

Dans le premier cas, la suppression de m revient à supprimer un pointeur, et dans le second à changer un booléen. Il y a au pire $\Theta(n)$ opérations de ce genre.

• Evaluons globalement le nombre de comparaisons, affectations, additions et mises à jour des tableaux d et pr, dues au réajustement des valeurs de d et pr, pour une exécution de l'algorithme. Pour chaque sommet m de M, chacun de ces nombres est majoré par $d^{o+}(m)$. Le nombre total de chacune de ces opérations est donc majoré par le nombre d'arcs p, dans le cas d'une représentation du graphe par listes d'adjacence. Si le graphe est représenté par matrice d'adjacence, cette complexité est en $\Theta(n^2)$.

Comme $p \leq n^2$, la complexité totale de la procédure, dans le pire des cas, est en $\Theta(n^2)$.

Dans le cas de *graphes denses*, où p est assez voisin de n^2, cette complexité est raisonnable. Mais si n^2 est beaucoup plus grand que p, il est légitime de se demander si on peut faire mieux. En fait, on peut dans ce cas améliorer la complexité de l'algorithme en représentant l'ensemble des sommets i de M par un tas ordonné par les valeurs de $d[i]$ (cf. chapitre 15) et le graphe par des listes d'adjacence.

Avec cette représentation, l'opération *choisir-min* est en $\Theta(1)$ et la suppression de m est en $\Theta(\log|M|)$ pour le nombre de comparaisons. Pour toute la boucle **while**, ces opérations impliquent alors au pire $\Theta(n.\log n)$ comparaisons.

Comme on modifie à la fois le cardinal de M et les valeurs associées aux éléments de M, il faut maintenir la structure de tas lorsqu'on réajuste ces valeurs (c'est-à-dire les tableaux d et pr de la procédure Dijkstra). On a vu qu'il y a au pire p réajustements pour une exécution de l'algorithme, et chaque réajustement consiste à supprimer du tas un élément y et son ancienne valeur $d[y]$ et à ajouter au tas cet élément, avec sa nouvelle valeur. Une telle suppression dans un tas de taille i nécessite au pire $\Theta(\log i)$ comparaisons. Il en est de même de l'adjonction. Le nombre total de comparaisons nécessaires, si on utilise un tas, est donc au pire en $O(p.\log n)$. (Il s'agit de la complexité en temps; il faudrait aussi évaluer la complexité en espace dans ce cas.) Par ailleurs, on a toujours $\Theta(p)$ opérations d'additions, de mises à jour des tableaux d et pr, etc.

Avec ces améliorations, si le graphe est représenté par des listes d'adjacence de successeurs, *la complexité au pire de l'algorithme est en* $O(\max(p,n).(\log n))$. Lorsque $p \geq n$ on a une complexité en $O(p.\log n)$ qui est préférable à $\Theta(n^2)$ si $n^2 >> p$.

3. Algorithme de Bellman (graphe sans circuit, coûts quelconques)

Cet algorithme est valable pour des graphes sans circuit, valués par des coûts quelconques. Il donne un plus court chemin d'un sommet r à tous les autres sommets

ainsi que son coût, ***dans le cas où r est racine du graphe*** (une extension au cas où r n'est pas racine est proposée en exercice).

3.1. Algorithme

On note M l'ensemble des sommets x pour lesquels *ppdistance*(r, x) n'est pas encore connue. On ne calcule *ppdistance*(r, x) que lorsque tous les prédécesseurs de x sont dans $CC = S - M$. Il est donc nécessaire de connaître, pour tout sommet x, son demi-degré intérieur et son $i^{ième}$ prédécesseur.

L'algorithme repose sur l'idée suivante : un plus court chemin de r à x doit passer par l'un des prédécesseurs y de x, celui pour lequel *ppdistance*(r, y) + *coût*(y, x) est minimum. L'algorithme choisit donc dans M un sommet x tel que tous ses prédécesseurs sont dans CC. Le choix du sommet x est réalisé par l'opération *choisir-suivant* de profil :

choisir-suivant : Ensemble × Graphe → Sommet

Cette opération n'est pas définie pour l'ensemble vide.

Pour tout ensemble M non vide, pour tout graphe G et pour tout i tel que

$1 \leq i \leq d^{o-}$ *de* (*choisir-suivant*(M, G)) *dans* G, on a :

(*i ème-pred-de choisir-suivant*(M, G) *dans* G) $\in M$ = faux
choisir-suivant$(M, G) \in M$ = vrai

La programmation de cette opération varie selon la représentation choisie pour le graphe. Dans tous les cas, elle nécessite le parcours de l'ensemble des prédécesseurs des sommets.

En utilisant cette opération, l'algorithme s'exprime de la manière suivante :

procedure *Bellman*(r : integer; G : Graphe; **var** d : **array** [1..n] **of** real;
var pr : **array**[1..n] **of** integer);
{*Le graphe G a n sommets :* $1, 2, \ldots, n$ *et est valué par des coûts quelconques; r est racine du graphe G; on recherche un plus court chemin de r à chacun des autres sommets; G est supposé sans circuit*}
var i, x, y, z : integer; *min*, *aux* : real; M : Ensemble;
{ x, y *et* z *sont des sommets*}

begin
{*initialisations*}
M := *ensemble-vide*;
for $i := 1$ **to** n **do** M := *ajouter*(i, M);
M := *supprimer*(r, M);
$d[r] := 0$; $pr[r] := r$;

```
    {choix successifs d'un élément dans M}
    while M <> ensemble-vide do begin
        x := choisir-suivant(M, G);
        M := supprimer(x, M);
        {on cherche y, prédécesseur de x, tel que d[y] + coût(y, x, G) soit
        minimum :}
        y := 1 ème-pred-de x dans G;
        min := d[y] + coût(y, x, G);
        for i := 2 to d°− de x dans G do begin
            z := i ème-pred-de x dans G;
            aux := d[z] + coût(z, x, G);
            if aux < min then begin min := aux; y := z end
        end;
        d[x] := min; {d[x] = ppdistance(r, x)}
        pr[x] := y   {pr[x] = père(r, x)}
    end
end Bellman;
```

3.2. Exemple

On considère le graphe G sans circuit de la figure 5, qui a pour racine le sommet $s1$.

L'exécution de la procédure sur G se fait selon les étapes suivantes :

1) Initialisation : $M = \{s2, s3, s4, s5, s6\}$

d :	0					
	1	2	3	4	5	6

pr :	1					
	1	2	3	4	5	6

2) $M = \{s2, s4, s5, s6\}$

d :	0		−2			
	1	2	3	4	5	6

pr :	1		1			
	1	2	3	4	5	6

3) $M = \{s2, s4, s6\}$. A cette étape, on aurait pu choisir de supprimer le sommet $s2$, au lieu du sommet $s5$.

d :	0		−2		2	
	1	2	3	4	5	6

pr :	1		1		3	
	1	2	3	4	5	6

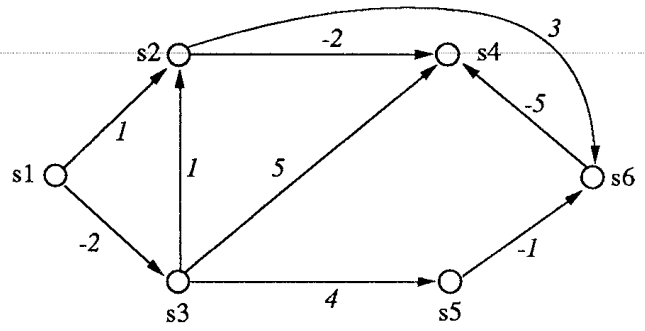


Figure 5. Graphe valué sans circuit de racine $s1$.

4) $M = \{s4, s6\}$

d :	0	−1	−2		2		pr :	1	3	1		3	
	1	2	3	4	5	6		1	2	3	4	5	6

5) $M = \{s4\}$

d :	0	−1	−2		2	1	pr :	1	3	1		3	5
	1	2	3	4	5	6		1	2	3	4	5	6

6) $M = \emptyset$

d :	0	−1	−2	−4	2	1	pr :	1	3	1	6	3	5
	1	2	3	4	5	6		1	2	3	4	5	6

3.3 Justification de l'algorithme

Pour tout sommet intermédiaire z sur un plus court chemin de r à y, la portion de ce chemin comprise entre r et z est un plus court chemin de r à z. Par suite, on obtient bien un plus court chemin de r à tout sommet de CC, ainsi que son coût. Comme à la fin de la procédure, M est vide, on aura montré que l'algorithme est correct, si on montre qu'à chaque itération de la boucle **while**, l'opération *choisir-suivant* est bien définie.

Supposons que M soit non vide et qu'on ne puisse pas trouver un sommet x dans M dont tous les prédécesseurs soient hors de M. Puisque r est racine du graphe, tout sommet autre que r admet un prédécesseur. Soit y_0 un sommet de M, y_0 est différent de r car M est initialisé à $S - \{r\}$; y_0 admet au moins un

prédécesseur y_1 dans M. On peut recommencer avec ce sommet y_1 et construire une suite, $y_0, y_1, \ldots, y_i, y_{i+1}, \ldots$ telle que y_i est élément de M et y_{i+1} est un prédécesseur de y_i qui est dans M. Comme le graphe est sans circuit, cette suite est infinie. On obtient une contradiction avec le fait que M est nécessairement fini. L'opération *choisir-suivant* est donc toujours possible.

3.4. Analyse de la complexité

On évalue la complexité de l'algorithme en nombre total d'opérations, en fonction du nombre n de sommets et du nombre p d'arcs du graphe.

Cette complexité dépend des représentations du graphe, de l'ensemble M et de l'opération *choisir-suivant.*

Plaçons-nous dans le cas, favorable pour la complexité en temps de cet algorithme, où le graphe est représenté par listes de successeurs et par listes de prédécesseurs. La représentation de l'ensemble M doit être choisie en tenant compte de la complexité de l'opération *choisir-suivant* et de celle des mises à jour. L'opération *choisir-suivant* nécessite de connaître, pour chaque sommet de M, le nombre de ses prédécesseurs qui sont dans M. Pour éviter de recalculer ce nombre de prédécesseurs pour chaque nouvelle valeur de l'ensemble M, on peut gérer un tableau des nombres de prédécesseurs, tableau qui doit être mis à jour après chaque suppression d'un élément de M. Dans ce cas, effectuer *choisir-suivant* revient à choisir dans ce tableau un indice de valeur nulle. Pour ne pas avoir à parcourir le tableau, on chaîne dans le tableau les indices de valeurs nulles, comme dans le tri topologique (cf. la procédure *tritopo2* au chapitre 18). L'opération *choisir-suivant* est alors en $\Theta(1)$.

On calcule la complexité de l'algorithme pour ces représentations, en supposant que le graphe est donné par listes de successeurs et listes de prédécesseurs.

L'initialisation de M est en $\Theta(n)$. Lors des $n-1$ exécutions de la boucle **while**, on fait en tout $\sum_{x=1,\ldots,n} d^{o+}(x) = p$ mises à jour du tableau des nombres de prédécesseurs, et pour la recherche de y et de *min*, de l'ordre de $\sum_{x=1,\ldots,n} d^{o-}(x) = p$ opérations.

De plus, à chaque itération, l'exécution de *choisir-suivant* et *supprimer* est en $\Theta(1)$. Si on connaît pour chaque sommet la liste de ses prédécesseurs et la liste de ses successeurs, alors on a une complexité totale de $\Theta(n+p)$ opérations. Comme $n-1 \leq p$ (G admet une racine r), la complexité totale est en $\Theta(p)$.

Si la représentation du graphe rend inefficace le parcours de la liste des successeurs ou de la liste des prédécesseurs d'un sommet, la complexité peut augmenter.

4. Algorithme de Floyd (coûts quelconques)

On s'intéresse maintenant au problème de la recherche d'un plus court chemin pour tout couple de sommets (x, y). On pourrait utiliser les algorithmes précédents en faisant varier la source s. En fait, il existe des algorithmes simples qui calculent directement tous les plus courts chemins en utilisant une représentation des graphes par matrice d'adjacence.

On donne ici une méthode matricielle, l'algorithme de Floyd, valable pour des graphes valués par des coûts quelconques (*dans le cas où il n'y a pas de circuit absorbant*). Une autre méthode matricielle (l'algorithme de Dantzig) est proposée en exercice.

4.1. Principe et algorithme

Le principe de l'algorithme est analogue à celui de l'algorithme de Warshall vu au chapitre 19.

Soit $S = \{1, 2, \ldots, n\}$ l'ensemble des sommets du graphe.

On considère successivement les chemins de i vers j qui ne passent d'abord par aucun autre sommet, puis qui passent éventuellement par le sommet 1, puis par les sommets 1 et 2, etc. A l'étape k, on calcule le coût $d_k(i, j)$ d'un plus court chemin de i à j passant par des sommets inférieurs ou égaux à k. On note $pred_k(i, j)$ le prédécesseur de j dans ce plus court chemin.

On utilise une matrice carrée A d'ordre n, initialisée aux valeurs de l'opération *coût* (on prolonge cette opération comme dans l'algorithme de Dijkstra, de manière à ce qu'elle soit définie pour tout couple de sommets). A chaque étape, cette matrice contient les coûts d_k des plus courts chemins calculés. On utilise également une matrice carrée P d'ordre n telle que $P[i, j] = père(i, j)$, s'il existe un chemin de i vers j. Au départ, $P[i, j] = i$. L'algorithme s'écrit de la manière suivante (en ignorant les problèmes d'additions avec l'infini) :

```
procedure Floyd(var A : array[1..n, 1..n] of real; G : Graphe;
                var P : array[1..n, 1..n] of integer);
{Le graphe G est valué par des coûts quelconques ; on suppose qu'il n'y a pas de circuit absorbant ; on cherche un plus court chemin entre deux sommets quelconques}
var i, j, k : integer;
begin
    {initialisation}
    for i := 1 to n do
        for j := 1 to n do begin
            A[i, j] := coût(i, j, G);
            P[i, j] := i
        end;
```

```
{calcul des plus courts chemins}
for k := 1 to n do
    for i := 1 to n do
        for j := 1 to n do
            if A[i,k] + A[k,j] < A[i,j] then begin
                A[i,j] := A[i,k] + A[k,j];
                P[i,j] := P[k,j]
            end
{si A[i,j] < ∞, A[i,j] = ppdistance(i, j) et P[i,j] =père(i, j)}
end Floyd;
```

4.2. Exemple

L'exécution de la procédure sur le graphe de la figure 6 donne les matrices successives suivantes.

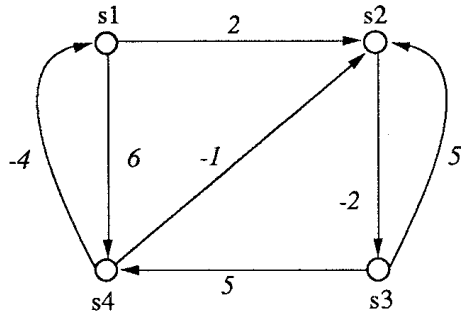


Figure 6. Graphe valué par des coûts quelconques, sans circuit absorbant.

1) Initialisation

$$\boldsymbol{A} = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{bmatrix} 0 & 2 & \infty & 6 \\ \infty & 0 & -2 & \infty \\ \infty & 5 & 0 & 5 \\ -4 & -1 & \infty & 0 \end{bmatrix} \end{array} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \qquad \boldsymbol{P} = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 \end{bmatrix} \end{array} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix}$$

$k = 1$

$$\boldsymbol{A} = \begin{bmatrix} 0 & 2 & \infty & 6 \\ \infty & 0 & -2 & \infty \\ \infty & 5 & 0 & 5 \\ -4 & -2 & \infty & 0 \end{bmatrix} \qquad \boldsymbol{P} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 4 & 1 & 4 & 4 \end{bmatrix}$$

$k = 2$

$$A = \begin{bmatrix} 0 & 2 & 0 & 6 \\ \infty & 0 & -2 & \infty \\ \infty & 5 & 0 & 5 \\ -4 & -2 & -4 & 0 \end{bmatrix} \qquad P = \begin{bmatrix} 1 & 1 & 2 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 4 & 1 & 2 & 4 \end{bmatrix}$$

$k = 3$

$$A = \begin{bmatrix} 0 & 2 & 0 & 5 \\ \infty & 0 & -2 & 3 \\ \infty & 5 & 0 & 5 \\ -4 & -2 & -4 & 0 \end{bmatrix} \qquad P = \begin{bmatrix} 1 & 1 & 2 & 3 \\ 2 & 2 & 2 & 3 \\ 3 & 3 & 3 & 3 \\ 4 & 1 & 2 & 4 \end{bmatrix}$$

$k = 4$

$$A = \begin{bmatrix} 0 & 2 & 0 & 5 \\ -1 & 0 & -2 & 3 \\ 1 & 3 & 0 & 5 \\ -4 & -2 & -4 & 0 \end{bmatrix} \qquad P = \begin{bmatrix} 1 & 1 & 2 & 3 \\ 4 & 2 & 2 & 3 \\ 4 & 1 & 3 & 3 \\ 4 & 1 & 2 & 4 \end{bmatrix}$$

4.3. Justification de l'algorithme

A l'étape k, on considère les chemins de i à j qui passent par des sommets inférieurs ou égaux à k. On peut éventuellement obtenir un chemin μ de i à j, plus court que le chemin λ obtenu à l'étape $k-1$, en prenant un plus court chemin de i à k, et un plus court chemin de k à j passant tous deux par des sommets inférieurs ou égaux à $k-1$ (voir figure 7). La valeur $d_k(i,j)$ est donc donnée par :

$$d_k(i,j) = \min(d_{k-1}(i,j), d_{k-1}(i,k) + d_{k-1}(k,j))$$

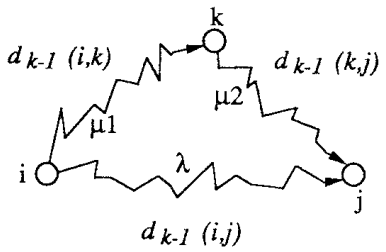


Figure 7. Plus courts chemins passant par des sommets inférieurs ou égaux à k.

On n'obtient un chemin μ de i vers j plus court que λ que si $d_{k-1}(i,k) + d_{k-1}(k,j) < d_{k-1}(i,j)$. Dans ce cas, ce chemin se termine par un plus court chemin $\mu 2$ de k à j calculé à l'étape $k-1$. Le prédécesseur de j dans ce nouveau chemin est donc le prédécesseur de j dans $\mu 2$: $pred_k(i,j) = pred_{k-1}(k,j)$.

Au bout de n étapes, $d_n(i,j) = ppdistance(i,j)$ et $pred_n(i,j) = père(i,j)$.

Comme pour l'algorithme de Warshall, on peut n'utiliser qu'une seule matrice A pour le calcul des $d_k(i,j)$ car entre l'étape $k-1$ et l'étape k les valeurs de $d_{k-1}(i,k)$ et de $d_{k-1}(k,j)$ qui interviennent dans le calcul des $d_k(i,j)$, ne changent pas.

Obtention des plus courts chemins

Pour obtenir un plus court chemin de i à j, il suffit d'utiliser la ligne numéro i de la matrice P. Par exemple, si on veut obtenir le plus court chemin μ de $s4$ à $s3$, dans le graphe de la figure 6, on consulte la matrice P ainsi : $P[4,3] = 2 : s2$ est donc le prédécesseur de $s3$ dans μ; $P[4,2] = 1 : s1$ est le prédécesseur de $s2$ dans μ; $P[4,1] = 4 : s4$ est le prédécesseur de $s1$ dans μ. Finalement, le chemin μ s'écrit : $s4 \rightarrow s1 \rightarrow s2 \rightarrow s3$.

On peut obtenir les plus courts chemins d'une autre façon. A la $k^{ième}$ itération, si $A[i,k] + A[k,j] < A[i,j]$, on indique que le sommet k figure sur un plus court chemin de i à j passant par des sommets inférieurs ou égaux à k. Dans ce cas, on initialise la matrice P partout à 0 et on remplace l'instruction $P[i,j] := P[k,j]$ par l'instruction $P[i,j] := k$. Avec ces deux méthodes, l'impression d'un plus court chemin peut se faire de façon récursive (cf. exercices).

4.4. Analyse de la complexité

Il est clair que le nombre d'opérations (additions, comparaisons, affectations) effectuées par l'algorithme de Floyd est en $\Theta(n^3)$.

Dans le cas où les coûts sont tous positifs ou nuls, on aurait pu aussi utiliser l'algorithme de Dijkstra n fois. Avec une représentation par matrice d'adjacence, et sans utiliser de tas, l'algorithme de Dijkstra est en $\Theta(n^2)$ dans le pire des cas : on obtient ainsi une méthode qui est encore en $\Theta(n^3)$ pour la recherche des plus courts chemins pour tous les couples de sommets.

Si le graphe est peu dense et s'il est représenté par des listes de successeurs, l'utilisation d'un tas permet d'avoir une complexité en $O(p.\log n)$ pour l'algorithme de Dijkstra. Sa répétition n fois est alors en $O(n.p.\log n)$, ce qui est plus intéressant que l'algorithme de Floyd quand $p << n^2$.

Exercices

1. Examiner chacune des assertions suivantes ; la prouver si elle est exacte et fournir un contre-exemple, sinon.

a) Si les coûts des arcs d'un graphe orienté valué sont tous distincts, alors pour tous sommets x et s distincts, il existe au plus un plus court chemin de s à x.

b) On peut avoir deux arborescences de plus courts chemins distinctes de même racine.

c) Pour trouver les plus longs chemins dans un graphe orienté sans circuit $G = < S,\ A,\ C >$, où C est une fonction de coût à valeurs positives, on peut se ramener à la recherche des plus courts chemins dans le graphe $G' = < S,\ A,\ C' >$, avec $C'(x, y) = K - C(x, y)$, pour tout K suffisamment grand.

2. Dans le cas où tous les arcs d'un graphe orienté sont de coût identique (égal à 1 par exemple), montrer que l'algorithme de parcours en largeur (cf. chapitre 8, §5.5), effectué à partir d'un sommet s, permet d'obtenir un plus court chemin de s vers chaque autre sommet accessible depuis s.

3. a) Faire tourner l'algorithme de Dijkstra sur le graphe de la figure 8, en choisissant le sommet $s1$ comme sommet source .

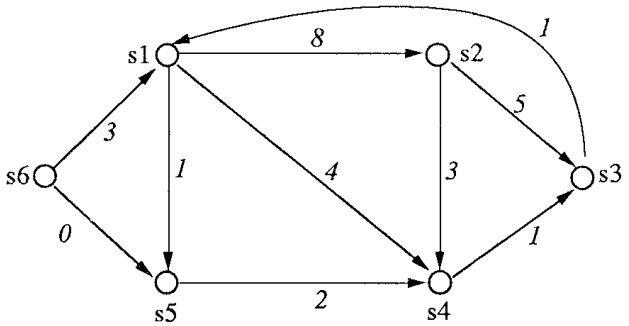


Figure 8

b) Ecrire une procédure pour l'algorithme de Dijkstra, avec une représentation du graphe par matrice d'adjacence, sans utiliser de tas. Evaluer sa complexité.

c) Ecrire une procédure pour l'algorithme de Dijkstra, en représentant le graphe par listes de successeurs et en utilisant un tas. Evaluer sa complexité en temps et en espace.

d) Donner une procédure récursive qui imprime un plus court chemin d'un sommet source à un sommet accessible depuis cette source. Montrer que cette procédure ne boucle pas.

4. Trouver une configuration de graphe ayant n sommets et p arcs telle que le nombre de réajustements des valeurs de $d[k]$ et $pr[k]$, dans l'algorithme de Dijkstra soit en $\Theta(n^2)$. (On choisira p tel que $p = \Theta(n^2)$).

5. a) Montrer qu'un sommet r d'un graphe orienté sans circuit est une racine si et seulement si c'est l'unique sommet y tel que $d^{o-}(y) = 0$.

b) Ecrire une procédure pour l'algorithme de Bellman avec une représentation du graphe par matrice d'adjacence. Evaluer sa complexité.

c) Modifier la procédure *Bellman* de sorte qu'elle donne un plus court chemin d'un sommet x quelconque vers chaque sommet accessible depuis x, dans un graphe orienté sans circuit, valué par des coûts quelconques.

6. Dans cet exercice, on propose un algorithme général attribuable sans doute à Ford, valable pour tout graphe G orienté valué, qui indique s'il existe un circuit absorbant, et s'il n'y en a pas, renvoie une arborescence de plus courts chemins issus d'un sommet source s. Le principe de l'algorithme consiste à modifier une arborescence de racine s, obtenue auparavant, (par exemple, par application de l'algorithme de Dijkstra), jusqu'à ce qu'elle devienne une arborescence des plus courts chemins de s vers les sommets accessibles depuis s. Cette méthode est donc très intéressante lorsqu'il s'agit de mettre à jour une arborescence des plus courts chemins après modification du coût de certains arcs. L'algorithme est le suivant :

(1) Appliquer la procédure Dijkstra au sommet s. Soit d et pr les tableaux résultats; pr décrit une arborescence AB de racine s (sous-graphe de G).

(2) Chercher un arc $x \to y$ qui n'est pas dans AB (c'est-à-dire, $x \neq pr[y]$) tel que $d[y] - d[x] > coût(x, y)$.

(i) **si** un tel arc n'existe pas **alors** FIN; AB est une arborescence des plus courts chemins.

(ii) **sinon si** $AB \cup \{x \to y\}$ admet un circuit alors FIN; on a trouvé un circuit absorbant et le problème de la recherche des plus courts chemins à partir de s n'a pas de solution.

(iii) **sinon aller en** (3).

(3) Enlever de l'arborescence AB, l'arc $pr[y] \to y$ et ajouter l'arc $x \to y$. On a encore une arborescence de racine s. **Pour tout** sommet z tel que : $(z = y)$ ou (z est un descendant de y), **faire** :

$$d[z] := d[z] - (d[y] - d[x] - coût(x, y))$$

Aller en (2).

a) Faire tourner l'algorithme sur le graphe de la figure 5, en prenant $s1$ comme sommet s.

b) Prouver que l'algorithme est correct. On montrera que l'algorithme s'arrête en temps fini, soit en donnant un circuit absorbant s'il y en a un, et sinon en donnant une arborescence des plus courts chemins. Pour cela, on remarquera que la quantité $\sum_{z \in S} d[z]$ est bornée inférieurement et qu'à chaque étape, elle décroît strictement.

c) Ecrire une procédure qui réalise cet algorithme et calculer sa complexité.

7. a) Ecrire un programme qui imprime un plus court chemin entre deux sommets i et j, s'il existe, en utilisant la matrice P calculée dans la procédure *Floyd.*

b) On a vu qu'on peut obtenir les plus courts chemins pour tout couple de sommets, en remplaçant dans la procédure *Floyd*, l'instruction $P[i, j] := P[k, j]$ par l'instruction $P[i, j] := k$, où $P[i, j]$ indique alors un sommet figurant dans le plus court chemin de i à j.

Ecrire une procédure récursive qui imprime le plus court chemin de i à j, s'il existe. On vérifiera que cette procédure ne boucle pas.

c) Montrer comment on peut modifier la procédure *Floyd*, de sorte qu'elle s'arrête dès qu'elle détecte un circuit absorbant, s'il y en a un.

8. Dans certains problèmes de communication, on a besoin de connaître le point (ou un des points) le moins éloigné de tous les autres points d'un réseau. On appelle ***excentricité*** d'un sommet x, dans un graphe orienté et valué par des coûts positifs, $G =< S, A, C >$, la quantité :

$$\text{exc}(x) = \max_{y \in S} \{ppdistance(x, y)\}$$

C'est la distance maximum nécessaire pour atteindre x depuis chaque autre sommet. On appelle ***centre*** du graphe G, un sommet d'excentricité minimum.

a) Montrer comment on peut trouver le centre d'un graphe en utilisant la procédure *Floyd.*

b) Trouver le centre du graphe donné à la figure 8 (exercice n° 3).

9. L'algorithme de Dantzig calcule les coûts des plus courts chemins pour tout couple de sommets d'un graphe orienté de n sommets, valué par des coûts quelconques, sans circuit absorbant. C'est une méthode matricielle qui procède par extensions successives du graphe et qui est intéressante lorsqu'on est amené à ajouter des sommets à un graphe pour lequel on connaît déjà la matrice des plus courts chemins.

A la $k^{ième}$ itération, on a une matrice carrée d'ordre k, M_k, qui indique les coûts des plus courts chemins du sous-graphe engendré par les sommets $1, 2, \ldots, k (k < n)$. On obtient M_{k+1} à partir de M_k comme suit :

– on calcule d'abord pour i tel que $1 \leq i \leq k$:

$$M_{k+1}[i, k+1] = \min_{1 \leq j \leq k} \{M_k[i, j] + coût(j,\ k+1)\}$$

$$M_{k+1}[k+1, i] = \min_{1 \leq j \leq k} \{coût(k+1,\ j) + M_k[j, i]\}$$

– puis on calcule pour i et j tels que $1 \leq i, j \leq k$:

$$M_{k+1}[i, j] = \min\{M_k[i, j],\ M_{k+1}[i, k+1] + M_{k+1}[k+1, j]\}$$

a) Prouver que la matrice M_n donne les coûts des plus courts chemins pour un graphe sans circuit absorbant.

b) Ecrire une procédure pour cet algorithme (on s'assurera qu'on n'a besoin de garder en mémoire qu'une seule matrice). Evaluer la complexité de cette procédure.

c) Modifier l'algorithme de manière à ce qu'il puisse détecter l'existence d'un circuit absorbant.

10. Les algorithmes de Dijkstra, Bellman et Floyd sont-ils valables pour des graphes non orientés ? Quelles adaptations faut-il éventuellement faire ?

11. Beaucoup de réalisations techniques ont un objectif à atteindre qui suppose l'exécution de multiples tâches t_i, soumises à des contraintes de successions : la tâche t_i de durée d_i doit être achevée pour que la tâche t_j commence. C'est un exemple de problème *d'ordonnancement.* On représente un tel projet par un graphe orienté dont les sommets représentent les tâches (graphe potentiel-tâches). On définit un arc de coût d_i entre t_i et t_j, si la tâche t_i doit immédiatement précéder la tâche t_j et qu'elle dure d_i. Ce graphe est sans circuit et admet des sommets sans prédécesseur et des sommets sans successeur. On ajoute au graphe deux sommets α et ω correspondant à des tâches fictives telles que α est la tâche de début de

projet, de durée nulle, qui doit être antérieure à toutes les autres tâches (on relie α aux sommets sans prédécesseur par des arcs de coût nul) et ω est la tâche de fin de projet, qui doit être postérieure à toutes les autres tâches (on relie les sommets sans successeur à ω par des arcs de coût nul). Le projet commence à la date 0 et on cherche une exécution des tâches qui minimise la durée totale du projet. On note τ_i la date au plus tôt à laquelle la tâche t_i peut être commencée.

a) Montrer que τ_i est égale au coût d'un plus long chemin de α vers t_i. Quelle est la durée minimum du projet ?

b) Montrer comment on peut calculer τ_i en modifiant

(i) la procédure *Bellman*,

(ii) la procédure *Floyd*.

Chapitre 21
Arbres de recouvrement minimums

1. Spécification informelle

On a défini au chapitre 8 une structure de graphe particulière, la structure d'arbre, qui joue un rôle important en théorie des graphes. Un arbre est un graphe non orienté connexe et sans cycle. On présente maintenant la notion d'arbre de recouvrement d'un graphe.

On appelle ***arbre de recouvrement d'un graphe G non orienté***, un graphe partiel de G qui est un arbre. (On rappelle qu'un graphe partiel G' de G est un graphe qui a les mêmes sommets que G mais dont l'ensemble des arêtes est inclus dans celui de G). Dans le cas où G est un graphe valué, on appelle ***coût d'un graphe partiel G'*** de G et on note $coût(G')$, la somme des coûts (ou poids) des arêtes de G'. Dans ce chapitre, on étudie deux algorithmes qui permettent de trouver, s'il existe, ***un arbre de recouvrement minimum*** pour un graphe G non orienté valué, c'est-à-dire, un arbre de recouvrement de G de coût minimum.

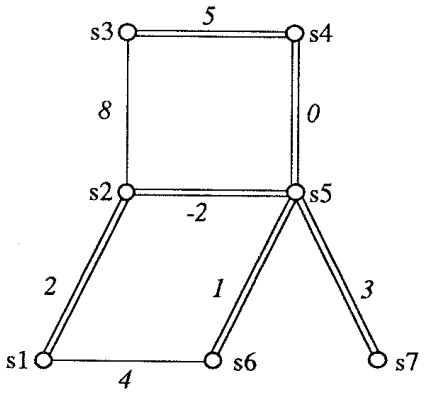


Figure 1. Arbre de recouvrement minimum.

Ce problème a des applications importantes. On le rencontre chaque fois que, connaissant le coût des liaisons possibles entre les nœuds d'un réseau, on veut connecter ces nœuds de la façon la plus économique possible (réseau routier, câblage de circuits électroniques, etc).

Dans toute la suite de ce chapitre, on considère des graphes non orientés, valués. A la figure 1, on a donné un tel graphe et on a dessiné en traits doubles un arbre de recouvrement minimum pour ce graphe.

Soit $u = \{x, y\}$ une arête d'un tel graphe; on note indifféremment $coût(x, y)$ ou $coût(u)$.

1.1. Conditions d'existence

Les conditions d'existence d'un arbre de recouvrement minimum pour un graphe non orienté valué, sont liées aux propriétés caractéristiques des arbres qu'on rappelle ici.

Proposition 1 : Soit $G = < S, A >$ un graphe non orienté de n sommets. Les propriétés suivantes sont équivalentes.

(1) G est connexe et sans cycle.
(2) G est connexe et si on supprime une arête, G n'est plus connexe.
(3) G est connexe avec $n - 1$ arêtes.
(4) G est sans cycle et si on ajoute une arête, on crée un cycle.
(5) G est sans cycle avec $n - 1$ arêtes.
(6) Tout couple de sommets de S est relié par une unique chaîne de G.

• Soit G un graphe connexe; on peut toujours trouver un arbre de recouvrement de G, en supprimant de G les arêtes qui forment des cycles. Le graphe obtenu alors est encore connexe. Du fait qu'il y a un nombre fini d'arbres de recouvrement pour un graphe non orienté, l'existence d'un arbre de recouvrement de coût minimum pour un graphe non orienté connexe est assurée.

• Si le graphe G n'est pas connexe, en supprimant de G les arêtes qui forment des cycles, on construit un graphe partiel de G sans cycle non connexe : on obtient une *forêt de recouvrement* composée d'autant d'arbres qu'il y a de composantes connexes dans G. Dans ce cas, on cherche à obtenir une forêt de recouvrement de coût minimum, où le coût d'une forêt est égal à la somme des coûts des arbres qui la constituent.

Remarque : Rechercher un arbre de recouvrement minimum d'un graphe G valué par des coûts strictement positifs, revient à rechercher un graphe partiel G' de G, connexe, de coût minimum. En effet, un tel graphe G' ne peut avoir de cycle, car sinon la suppression d'une arête d'un cycle donnerait un autre graphe partiel G''

connexe, de coût strictement inférieur à celui de G'. Comme G' est connexe, sans cycle et a n sommets, c'est un arbre de recouvrement de G.

1.2. Conditions d'unicité

En général, il peut y avoir plusieurs arbres de recouvrement minimums pour un graphe non orienté, valué et connexe. On en donne un exemple à la figure 2.

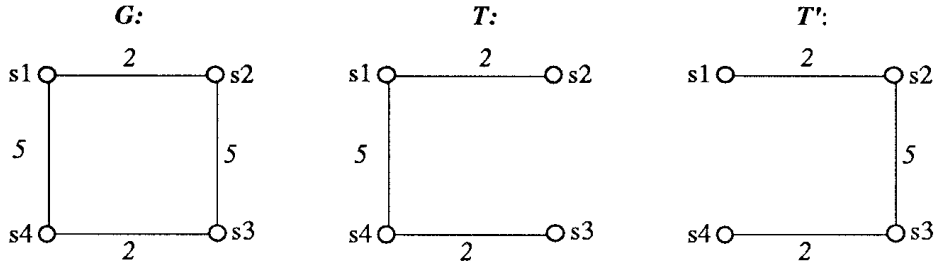


Figure 2. Arbres de recouvrement minimums T et T' pour le graphe G.

Si les coûts des arêtes de G satisfont une condition supplémentaire, on peut montrer qu'il y a unicité.

Proposition 2 : Soit G un graphe non orienté, connexe, valué, tel que les coûts des arêtes sont deux à deux distincts. Alors G admet un unique arbre de recouvrement minimum.

Preuve : L'existence a déjà été établie. Montrons l'unicité en raisonnant par l'absurde. Supposons qu'il existe deux arbres de recouvrement minimums $T1$ et $T2$ distincts. Ordonnons par ordre des coûts croissants la suite des arêtes de chacun de ces deux arbres. Soit $e_1, e_2, ..., e_{i-1}, u_i, ..., u_{n-1}$ la suite des arêtes de $T1$ et soit $e_1, e_2, ..., e_{i-1}, v_i, ..., v_{n-1}$ celle des arêtes de $T2$: i $(1 \leq i < n-1)$ est le premier indice pour lequel les deux suites diffèrent. Comme les coûts des arêtes sont tous distincts, on a $coût(u_i) \neq coût(v_i)$; soit par exemple, $coût(u_i) < coût(v_i)$.

Puisque u_i n'appartient pas à $T2$, l'adjonction de u_i à $T2$ crée un cycle (cf. proposition 1 (4)); $T1$ étant un arbre, ce cycle n'est pas inclus dans $T1$. Il existe donc une arête e' de ce cycle qui n'est pas dans $T1$; comme c'est une arête de $T2 \cup \{u_i\} - T1, e'$ est l'une des arêtes $v_i, v_{i+1}, ..., v_{n-1}$. Soit $j, i \leq j \leq n-1$ tel que $e' = v_j$. Considérons $T3 = T2 \cup \{u_i\} - \{v_j\}$. $T3$ est encore un arbre de recouvrement de G car c'est un graphe de n sommets, sans cycle, avec $n-1$ arêtes (cf. proposition 1 (5)).

$$coût(T3) = coût(T2) + coût(u_i) - coût(v_j)$$

Or $coût(v_j) \geq coût(v_i) > coût(u_i)$; d'où $coût(T3) < coût(T2)$.

On a donc construit un arbre de recouvrement $T3$ de coût strictement inférieur à celui de $T2$. Il y a contradiction avec le fait que $T2$ est un arbre de recouvrement minimum. L'unicité en résulte. □

2. Spécification formelle

La construction d'un arbre de recouvrement minimum d'un graphe est une opération de profil :

arb-rec-min : Graphe → Graphe

Sa spécification utilise celle des graphes non orientés vue au chapitre 8.

• Soit T un graphe non orienté. D'après la proposition 1 (6), T est un arbre s'il vérifie la propriété suivante : pour tout couple de sommets x, y de T, il existe une unique chaîne dans T reliant x et y.

Formellement, la notion de chaîne se définit à l'aide de l'opération *fermeture-transitive*, vue au chapitre 19. Plus précisément, il existe une chaîne entre x et y dans T si, et seulement si, il existe une arête dans *fermeture-transitive*(T).

• L'opération *arb-rec-min* est définie pour tout graphe G connexe.

Soit G un graphe non orienté, valué et soit T = *arb-rec-min*(G). Alors :

T est un *arbre de recouvrement* de G, c'est-à-dire, T est un arbre qui vérifie :

pour tous sommets x et y :

x *est-un-sommet-de* $T = x$ *est-un-sommet-de* G

$< x, y >$ *est-une-arête-de* T = vrai $\Rightarrow < x, y >$ *est-une-arête-de* G = vrai

T est un arbre de recouvrement *minimum* de G, c'est-à-dire,

pour tout arbre de recouvrement $T1$ de G :

$$\sum_{\{x,y\} \text{ dans } T} \textit{coût}(x, y, G) \quad \leq \quad \sum_{\{v,w\} \text{ dans } T1} \textit{coût}(v, w, G).$$

Dans ce chapitre, on recherche des arbres de recouvrement minimums pour un graphe G fixé. Pour simplifier les notations, on omettra, s'il n'y a pas d'ambiguïté, de mentionner G en argument des opérations sur les sommets et les arêtes.

Les deux algorithmes que l'on présente maintenant utilisent les propriétés caractéristiques des arbres. Dans les deux cas, on construit progressivement un arbre de recouvrement minimum par adjonctions successives d'arêtes à un graphe initialement vide. Cependant, dans l'algorithme de Prim, on maintient la connexité de l'ensemble

des arêtes du graphe en construction, alors que dans l'algorithme de Kruskal, on maintient l'absence de cycle.

3. Algorithme de Prim

3.1. Principe

Le principe de cet algorithme est voisin de celui de l'algorithme de Dijkstra : un minimum local est choisi à chaque étape selon des critères qui assurent qu'il fait partie de la solution globale.

Soit $G = < S, A, C >$ un graphe non orienté, valué et connexe. On va construire progressivement un sous-graphe T de G. Au départ, T est le graphe vide. On se donne un sommet s de S. On appelle CC l'ensemble des sommets reliés à s dans T, et M son complémentaire. (Au départ CC est réduit au sommet s). On modifie le graphe T, par l'adjonction, à chaque étape, d'une arête. L'arête ajoutée est choisie de sorte qu'elle soit de coût minimum parmi toutes les arêtes ayant une extrémité x dans M et l'autre y dans CC. On ne crée donc pas de cycle dans T : T est bien un arbre. On ajoute alors le sommet x à CC et on recommence. On s'arrête lorsque CC est l'ensemble S : T est alors un arbre de recouvrement de G.

Pour tout sommet y de M, on note $plusproche_{CC}(y)$ le sommet de l'arbre T en cours de construction, qui est le plus proche de y :

$$\forall z \in CC, coût(y, plusproche_{CC}(y)) \leq coût(y, z).$$

Soit $distance_{CC}(y) = coût(y, plusproche_{CC}(y))$. Le coût minimum des arêtes ayant une extrémité dans M et l'autre dans CC vaut alors $\inf_{y \in M} \{distance_{CC}(y)\}$.

Soit $m \in M$ un sommet tel que $distance_{CC}(m)$ réalise le minimum. On choisit d'ôter m de M, et d'ajouter à l'arbre T l'arête $\{m, plusproche_{CC}(m)\}$. Comme on a ajouté à CC le sommet m, on a éventuellement diminué la distance au nouvel arbre des sommets y restant dans M, dans le cas où il existe une arête $\{m, y\}$ de coût strictement inférieur à la distance de y à l'ancien arbre. Il faut donc réajuster $distance_{CC}(y)$ pour tous les sommets y adjacents à m.

3.2. Algorithme

On range le sommet $plusproche_{CC}(y)$ dans la case d'indice y du tableau d'entiers *plusproche*; ce tableau est initialisé à s.

La valeur $distance_{CC}(y)$ est stockée dans *dist*[y], qui est initialisée à $coût(s, y, G)$, où *dist* est un tableau de réels de taille n. Pour cela, comme au chapitre 20, on prolonge l'opération *coût*, de sorte qu'elle soit définie pour tout couple de sommets :

$coût(x,x) = 0$ et

$coût(x,y) = \infty$, s'il n'y a pas d'arête entre x et y.

Dans la procédure qui suit, le choix de l'élément m dans M tel que $distance_{CC}(m) = \inf_{y \in M} \{distance_{CC}(y)\}$ est réalisé par l'opération *choisir-min*(M, *dist*).

```
procedure Prim(s : integer; G : Graphe; var T : Graphe);
{Le graphe G a n sommets 1, 2, ..., n et est supposé connexe; s est un
sommet de G. Après exécution de Prim, T est un arbre de recouvrement
minimum de G}
var i, m, y, z : integer; M : Ensemble; v : real;
    plusproche : array[1..n] of integer; dist : array[1..n] of real;
{y, z et m sont des sommets}
begin
{initialisation de T comme graphe vide :}
T := graphe-vide;
{initialisation des tableaux dist, plusproche et M :}
M := ensemble-vide;
for i := 1 to n do begin
      dist[i] := coût(s, i, G);
      plusproche[i] := s;
      M := ajouter(i, M)
end;
M := supprimer(s, M); {CC = {s}}
{enrichissements successifs de T :}
while M <> ensemble-vide do begin
      m := choisir-min(M, dist);
      M := supprimer(m, M);
      z := plusproche[m]; {z = plusproche_CC(m)}
      v := coût(m, z, G);
      T := ajouter-l'arête < m, z > de-coût v à T;
      {mises à jour de dist et de plusproche :}
      for i := 1 to d° de m dans G do begin
            y := i ème-succ-de m dans G;
            if y ∈ M and (coût(m, y, G) < dist[y]) then begin
                        dist[y] := coût(m, y, G); {dist[y] = distance_CC(y)}
                        plusproche[y] := m
            end
      end
end
end Prim;
```

3.3. Exemple

On considère le graphe G non orienté, connexe et valué de la figure 3.

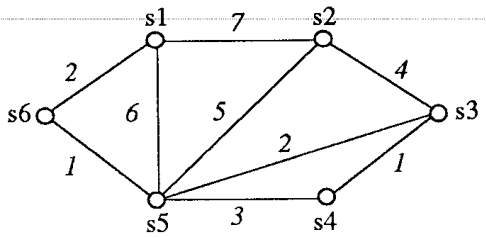


Figure 3. Graphe valué non orienté connexe.

Choisissons comme sommet s, le sommet $s1$. L'exécution de l'algorithme sur G donne successivement les graphes T suivants.

Initialisation : $M = \{s2, s3, s4, s5, s6\}$

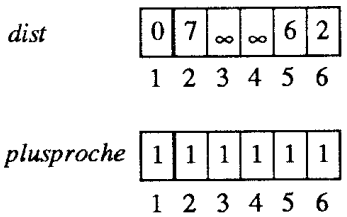


Figure 4

1) $M = \{s2, s3, s4, s5\}$

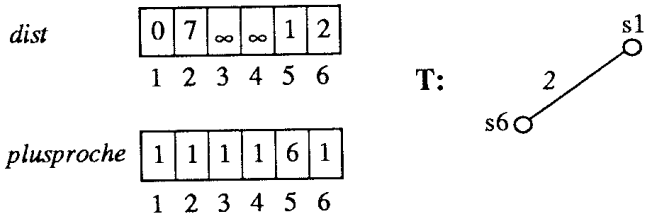


Figure 5

2) $M = \{s2, s3, s4\}$

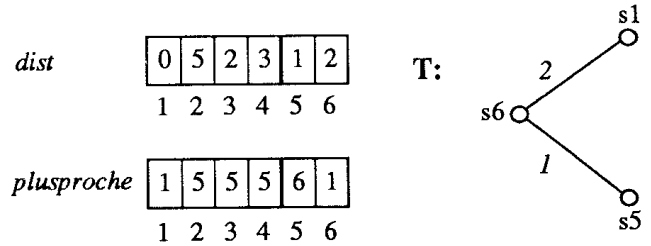


Figure 6

3) $M = \{s2, s4\}$

	1	2	3	4	5	6
dist	0	4	2	1	1	2
plusproche	1	3	5	3	6	1

T:

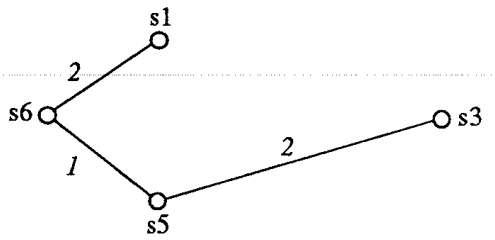


Figure 7

4) $M = \{s2\}$

	1	2	3	4	5	6
dist	0	4	2	1	1	2
plusproche	1	3	5	3	6	1

T:

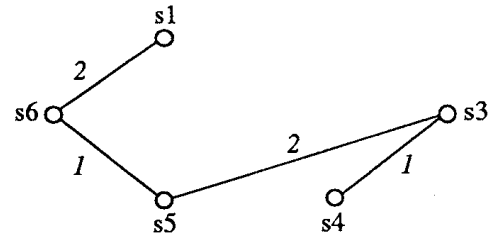


Figure 8

5) $M = \emptyset$

	1	2	3	4	5	6
dist	0	4	2	1	1	2
plusproche	1	3	5	3	6	1

T:

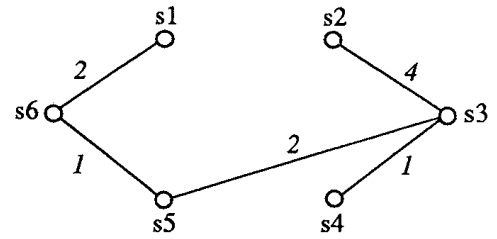


Figure 9

On obtient à cette étape un arbre de recouvrement minimum de G.

3.4. Justification de l'algorithme

L'algorithme de Prim repose sur la propriété fondamentale des arbres de recouvrement minimums suivante.

Proposition 3 : Soit $G = < S, A, C >$ un graphe valué non orienté connexe. Soit $A_1 \subseteq A$ un sous-ensemble d'arêtes qui forment une partie de l'ensemble des arêtes d'un arbre de recouvrement minimum $T1$ de G. Soit CC l'ensemble des sommets incidents aux arêtes de A_1. Soit $u = \{x, y\}$ une arête de coût minimum telle que

$x \in CC$ et $y \notin CC$. Alors il existe un arbre de recouvrement minimum qui contient $A_1 \cup \{u\}$.

Preuve : Comme G est connexe, il existe un arbre de recouvrement minimum $T1$ de G.

• Si $\{x,y\}$ est une arête de $T1$, $T1$ est l'arbre de recouvrement minimum cherché.

• Sinon, $\{x,y\}$ n'est pas une arête de $T1$. Comme $T1$ est un arbre de recouvrement, il existe une chaîne dans $T1$ reliant x et y (cf. proposition 1 (6)). Or $x \in CC$ et $y \notin CC$; il existe donc deux sommets adjacents x' et y' de cette chaîne tels que $x' \in CC$ et $y' \notin CC$ (figure 10), avec peut-être $x = x'$ ou $y = y'$.

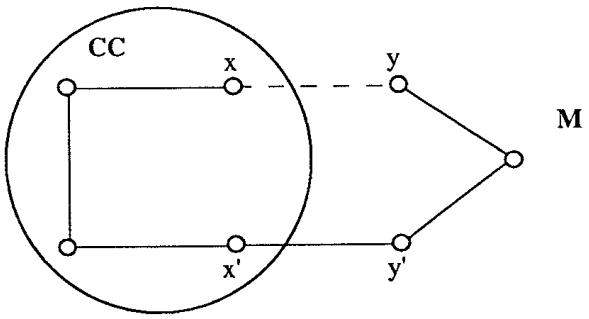


Figure 10. Chaîne dans $T1$ reliant $x \in CC$ et $y \notin CC$.

Soit $T2 = T1 \cup \{\{x,y\}\} - \{\{x',y'\}\}$.

$T2$ est encore un graphe partiel, connexe, sans cycle : c'est un arbre de recouvrement de G. Evaluons son coût :

$$coût(T2) = coût(T1) + coût(\{x,y\}) - coût(\{x',y'\})$$

Or, $coût(\{x',y'\}) \geq coût(\{x,y\})$, par choix de l'arête $\{x,y\}$ qui est de coût minimum.

D'où : $coût(T2) \leq coût(T1)$.

$T2$ est un arbre de recouvrement de coût minimum qui contient l'arête $\{x,y\}$. □

On a déjà vu que le graphe T obtenu à la fin de l'algorithme de Prim est un arbre de recouvrement. Montrons qu'il est minimum en raisonnant par récurrence sur le nombre k d'arêtes de T.

La preuve du cas de base $k = 1$ est analogue à celle de la proposition 3 : on reprend le même raisonnement, avec $A_1 = \emptyset, T1 = B$ et $CC = \{s\}$.

L'étape de récurrence résulte directement de la proposition 3.

Pour $k = n - 1$, on sait alors qu'il existe un arbre de recouvrement minimum de G qui contient T. Puisque G a n sommets et que T a $n - 1$ arêtes, il s'agit de T lui-même.

3.5. Analyse de la complexité

On évalue la complexité au pire de l'algorithme en nombre total d'opérations, en fonction du nombre de sommets n et du nombre d'arêtes p du graphe. Le calcul est analogue à celui qui a été fait pour l'analyse de l'algorithme de Dijkstra.

• L'initialisation du graphe T est en $\Theta(n^2)$ si T est représenté par une matrice d'adjacence et en $\Theta(n)$ si T est représenté par des listes d'adjacence.

• La deuxième boucle qui correspond à l'initialisation des tableaux *dist* et *plus-proche*, et de M est en $\Theta(n)$.

• Il y a $n - 1$ itérations dans la boucle **while**. Comme pour l'opération *choisir-min*(M, d) de l'algorithme de Dijkstra, que M soit représenté par une liste chaînée ou par un tableau de marques, la recherche de *choisir-min*$(M, dist)$ nécessite en tout $\Theta(n^2)$ comparaisons au pire. Dans les deux cas, la suppression de m est en $\Theta(1)$, d'où un coût en $\Theta(n)$ pour l'ensemble des suppressions dans la boucle **while**.

• Evaluons globalement le nombre de comparaisons et de mises à jour des tableaux *dist* et *plusproche*, lors du réajustement des valeurs de *dist* et de *plusproche*, pour une exécution de l'algorithme. Pour chaque sommet m de M, chacun de ces nombres est majoré par le degré de m. Dans le cas où le graphe est représenté par des listes d'adjacence, le nombre total de chacune de ces opérations est majoré par le nombre d'arêtes p. Si le graphe est représenté par une matrice d'adjacence, la complexité de ces opérations est en $\Theta(n^2)$.

La complexité totale de la procédure, dans le pire des cas, est donc en $\Theta(n^2)$.

Comme pour l'algorithme de Dijkstra, dans le cas de graphes creux ($n^2 >> p$), on peut abaisser cette complexité à $O(p.\log n)$, dans le pire des cas, en représentant judicieusement l'ensemble des sommets i de M par un tas ordonné selon les valeurs de *dist*[i] , et le graphe par des listes d'adjacence.

4. Algorithme de Kruskal

4.1. Principe

Soit $G = < S, A, C >$ un graphe non orienté valué et connexe, de n sommets et p arêtes. Comme précédemment, on part d'un graphe T vide. On ajoute à T des arêtes

de G, une par une, en choisissant à chaque étape, parmi les arêtes qui ne sont pas dans T, une arête de coût minimum, qui ne forme pas de cycle avec les arêtes qui sont déjà dans T. Lorsqu'on a ajouté ainsi $n-1$ arêtes, sans créer de cycle, on a obtenu un arbre recouvrement minimum (voir plus loin la justification).

Pour savoir si l'ajout d'une arête $\{x,y\}$ à T crée un cycle, il faut savoir s'il existe déjà dans T une chaîne entre x et y, autrement dit, si x et y sont dans la même composante connexe de T. Si ce n'est pas le cas, l'ajout de $\{x,y\}$ à T réunit les deux composantes connexes en une seule.

4.2. Algorithme

On a déjà vu au chapitre 19 des algorithmes *réunir* et *trouver* qui permettent de construire les composantes connexes d'un graphe évolutif, en les représentant par des arbres. La structure de ces arbres est donnée par un tableau d'entiers *père* initialisé à -1. On va reprendre ici les procédures *trouver-rapide* et *réunir-pondéré*. Rappelons que la procédure *trouver*(x, r_1, *père*) renvoie la racine r_1 de l'arbre qui contient x et effectue la compression du chemin de x vers r_1. Quant à la procédure *réunir-pondéré*(r_1, r_2, *père*), elle réunit les arbres de racines r_1 et r_2, en raccrochant l'arbre ayant le moins d'éléments à la racine de l'arbre qui en a le plus, dans le cas où ces arbres sont distincts.

Soit U l'ensemble des arêtes valuées du graphe qui n'ont pas encore été examinées. Au départ, U est initialisé à l'ensemble des arêtes valuées du graphe. Le choix dans U d'une arête de coût minimum est réalisé par l'opération *min*(U). On supprime alors cette arête de U, qu'elle forme ou non un cycle avec les arêtes qui sont déjà dans T, pour être sûr de ne pas la considérer à nouveau.

```
procedure Kruskal(G : Graphe; var T : Graphe);
{G est un graphe non orienté connexe de n sommets : 1, 2, .., n ; après
exécution de la procédure, T est un arbre de recouvrement minimum de G ;
on utilise les procédures trouver-rapide et réunir-pondéré }
var père : array[1..n] of integer;
    U : Ensemble; {U est un ensemble d'arêtes valuées}
    i, j, x, y, r1, r2 : integer; {x, y, r1 et r2 sont des sommets}
    v : real;
begin
    {initialisations}
    T := graphe-vide;
    U := ensemble-vide;
    for x := 1 to n do
        for j := 1 to d° de x dans G do begin
            y := j ème-succ-de x dans G;
            U := ajouter({x, y, coût(x, y, G)}, U)
        end;
```

```
        {initialisation des composantes connexes à des singletons :}
        for i := 1 to n do père[i] := -1;
        i := 1;
        {enrichissements successifs de T :}
        while i < n do begin
                {x, y, v} := min(U);
                U := supprimer({x, y, v}, U);
                trouver-rapide(x, r1, père);
                trouver-rapide(y, r2, père);
                if r1 <> r2 then begin
                        réunir-pondéré(r1, r2, père);
                        {on augmente T :}
                        T := ajouter-l'arête < x, y > de-coût v à T;
                        i := i + 1
                end
        end
end Kruskal;
```

4.3. Exemple

On considère à nouveau le graphe G de la figure 3. L'exécution de l'algorithme donne successivement les graphes T suivants.

Initialisation

$U = \{\{s1, s2, 7\}, \{s1, s5, 6\}, \{s1, s6, 2\}, \{s2, s3, 4\}, \{s2, s5, 5\}, \{s3, s4, 1\}, \{s3, s5, 2\}, \{s4, s5, 3\}, \{s5, s6, 1\}\}$

1) $min(U) = \{s3, s4, 1\}; U := U - \{s3, s4, 1\}; i = 1$

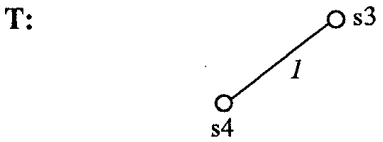


Figure 11

2) $min(U) = \{s5, s6, 1\}; U := U - \{s5, s6, 1\}; i = 2$

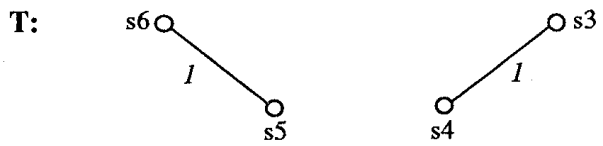


Figure 12

3) $min(U) = \{s1, s6, 2\}; U := U - \{s1, s6, 2\}; i = 3$

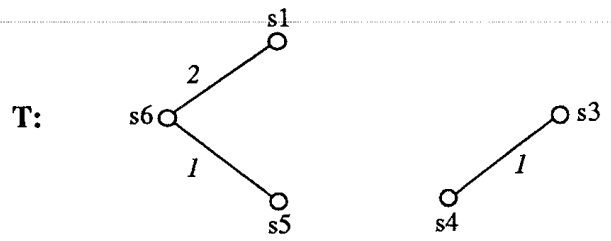


Figure 13

4) $min(U) = \{s3, s5, 2\}; U := U - \{s3, s5, 2\}; i = 4$

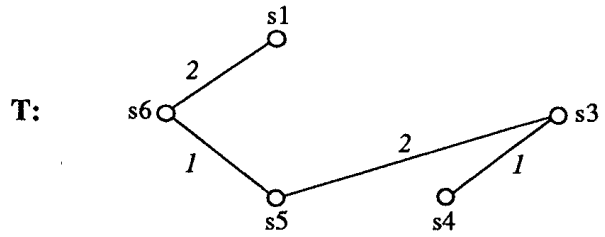


Figure 14

5) $min(U) = \{s4, s5, 3\}; U := U - \{s4, s5, 3\}; i = 4$

L'arête $\{s4, s5\}$ forme un cycle dans T : elle est rejetée.

6) $min(U) = \{s2, s3, 4\}; U := U - \{s2, s3, 4\}; i = 5$

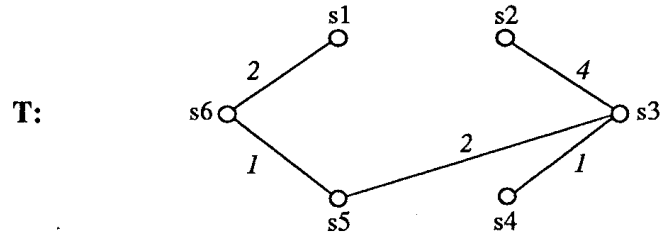


Figure 15

Lorsque $i = n - 1 = 5$, on obtient un arbre de recouvrement minimum; il se trouve que sur cet exemple il est identique à celui qu'on obtient par l'algorithme de Prim.

4.4. Justification de l'algorithme

Soit $G = < S, A, C >$ un graphe connexe non orienté valué de n sommets. Le sous-graphe T de G est initialement vide. Comme on lui ajoute $n-1$ arêtes qui ne forment pas de cycle, d'après la proposition 1 (5), on obtient à la fin de l'algorithme un arbre de recouvrement de G. Notons que puisqu'on part de n composantes connexes réduites à des singletons et que le graphe est connexe, il faut faire $n-1$ réunions de composantes connexes. On est alors sûr d'obtenir une seule composante connexe, après $n-1$ itérations au moins et p au plus, (puisqu'on n'envisage jamais deux fois la même arête).

Prouvons maintenant que l'arbre T obtenu est de coût minimum, par un raisonnement analogue à celui qu'on a fait pour l'algorithme de Prim. Montrons par récurrence sur le nombre d'arêtes ajoutées à T, que si E désigne l'ensemble des arêtes déjà choisies selon l'algorithme de Kruskal à une étape donnée, $(E \subseteq A)$, alors il existe un arbre recouvrement minimum contenant E.

A l'étape d'initialisation, $|E| = 1$; on choisit une arête $\{x, y\}$ de coût minimum. Cette arête est de coût minimum parmi toutes les arêtes d'extrémité x. D'après la proposition 3, il existe alors un arbre de recouvrement minimum contenant $\{x, y\}$.

A l'étape de récurrence, on suppose que si $|E| \geq 1$, il existe un arbre de recouvrement minimum $T1$ dont l'ensemble d'arêtes contient E. On choisit alors dans $A - E$ une arête u de coût minimum telle que $E \cup \{u\}$ est sans cycle.

• Si u est une arête de $T1, T1$ est un arbre de recouvrement minimum contenant $E \cup \{u\}$.

• Sinon u n'est pas une arête de $T1$; d'après la proposition 1 (4), l'adjonction de u à $T1$ crée un cycle. Comme $E \cup \{u\}$ est sans cycle, il existe une arête u' du cycle distincte de u, telle que u' appartient à $T1 \cup \{u\} - E$. Sur la figure 16, les arêtes de E sont dessinées avec des traits doubles.

Considérons $T2 = T1 \cup \{u\} - \{u'\}$. $T2$ est encore un arbre de recouvrement de G.

$$coût(T2) = coût(T1) + coût(u) - coût(u')$$

Comme par choix de u, $coût(u) \leq coût(u')$, on a $coût(T2) \leq coût(T1)$.

$T2$ est aussi un arbre de recouvrement minimum. De plus, il contient $E \cup \{u\}$.

La récurrence est établie.

Pour $|E| = n - 1$, l'arbre de recouvrement minimum contenant E ne peut être que T lui-même.

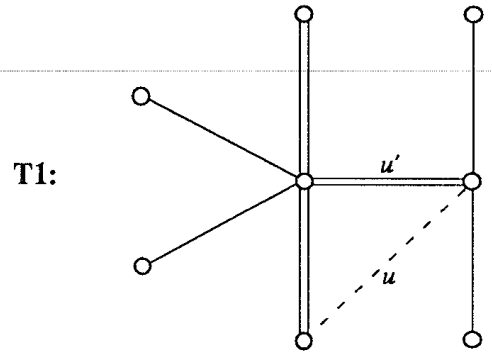


Figure 16

4.5. Analyse de la complexité

Comme pour l'algorithme de Prim, on évalue la complexité au pire de l'algorithme de Kruskal en nombre total d'opérations, en fonction du nombre de sommets n et du nombre d'arêtes p.

• L'initialisation du graphe T est en $\Theta(n^2)$ si T est représenté par une matrice d'adjacence et en $\Theta(n)$ s'il est représenté par des listes d'adjacence.

• On choisit de représenter U par un tableau d'enregistrements à trois champs de taille p : un champ pour chacun des deux sommets extrémités de l'arête et un champ pour le coût de l'arête. L'initialisation de U est en $\Theta(p)$ si le graphe G est représenté par listes d'adjacence et en $\Theta(n^2)$ si G est représenté par matrice d'adjacence.

• La deuxième boucle qui correspond à l'initialisation des composantes connexes est en $\Theta(n)$.

• Les opérations *min*(U) et *supprimer*$(\{x,y,v\},U)$ sont effectuées au pire p fois. Si U est une liste d'arêtes triées en ordre croissant, *min*(U) est en $\Theta(1)$; *supprimer*$(\{x,y,v\},U)$ est en $\Theta(1)$. Le tri de la liste d'arêtes peut se faire en $\Theta(p \log p)$. Notons que l'organisation de U en tas n'améliore pas la complexité : la réorganisation de U en tas après la destruction du minimum, est en $\Theta(\log(|U|))$ et peut avoir lieu $p-1$ fois pour $|U| = p-1, p-2, ..., 2$. L'utilisation d'un tas est donc également en $\Theta(p \log p)$.

• Au pire, le nombre de comparaisons $r_1 <> r_2$ est proportionnel au nombre d'arêtes p.

• On a vu au chapitre 19 que la complexité totale des algorithmes *trouver-rapide* et *réunir-pondéré* exécutés pour une suite de p arêtes d'un graphe de n sommets est quasi-linéaire.

• L'opération *ajouter-l'arête* $< x, y >$ *de-coût* v *à* T est en $\Theta(1)$ et est effectuée $n-1$ fois, d'où la complexité en $\Theta(n)$ pour cette opération dans l'ensemble de la procédure.

L'algorithme de Kruskal a donc une complexité au pire totale en $\Theta(p \log p)$ si le graphe est représenté par des listes d'adjacence et en $\Theta(\max(n^2, p \log p))$ si G est représenté par matrice d'adjacence.

En conclusion, on constate que l'algorithme de Prim est meilleur en complexité que l'algorithme de Kruskal, mais qu'il exige des concepts plus élaborés et des structures de données plus compliquées.

Exercices

1. Dans le problème de la recherche d'un arbre de recouvrement minimum, on a considéré un graphe connexe non orienté valué par des coûts quelconques. Montrer qu'on peut toujours se ramener au cas où le graphe admet uniquement des arêtes de coûts positifs.

2. Dans cet exercice, on propose une variante de l'algorithme de Kruskal.

Soit $u_1, u_2, ..., u_p$ la suite des p arêtes d'un graphe connexe non orienté valué G, rangées dans un ordre quelconque. Soit T le sous-graphe de G qui n'a qu'une arête, u_1. On modifie progressivement T comme suit : à la $k^{ième}$ étape $(2 \leq k \leq p)$, on ajoute l'arête u_k à T. Si T ainsi modifié ne contient pas de cycle, on continue; sinon, on choisit une arête v du cycle, de coût maximum, qu'on ôte à T et on continue.

Montrer qu'au bout de p étapes, T est un arbre de recouvrement minimum de G.

3. Peut-on modifier simplement les algorithmes de Prim et de Kruskal pour qu'ils construisent un arbre de recouvrement de coût maximum? Effectuer ces transformations.

4. Etablir le résultat d'optimalité suivant : dans le pire des cas, la recherche d'un arbre de recouvrement minimum d'un graphe de p arêtes est en $\Theta(p)$ opérations.

5. Evaluer la complexité en place de l'algorithme de Prim, lorsqu'on représente judicieusement l'ensemble des sommets i de M par un tas ordonné selon les valeurs de *dist*[i], et le graphe par des listes d'adjacence.

6. Examiner si l'arbre T obtenu dans la construction suivante est un arbre de recouvrement minimum de G; si oui, expliquer pourquoi, sinon construire un contre-exemple.

Soit $G = < S, A, C >$ un graphe non orienté connexe valué. On partage G en deux sous-graphes de G, $G_1 = < S_1, A_1, C >$ et $G_2 = < S_2, A_2, C >$ avec $S = S_1 \cup S_2$. Soit

T_1 (respectivement T_2) un arbre de recouvrement minimum de G_1 (respectivement G_2). On choisit une arête v de coût minimum parmi les arêtes ayant une extrémité dans S_1 et l'autre dans S_2. T est alors le graphe partiel de G résultant de la réunion des arêtes de T_1, de T_2 et de v.

7. Examiner si l'arbre T obtenu dans la construction suivante est un arbre de recouvrement minimum de G ; si oui, expliquer pourquoi, sinon construire un contre-exemple.

Soit $G = < S, A, C >$ un graphe non orienté connexe valué. Initialement T est vide. Soit s un sommet quelconque. On choisit une arête v incidente à s, de coût minimum, qu'on ajoute à T. Soit x l'autre extrémité de l'arête. On recommence avec ce sommet x : on cherche une arête incidente à x, de coût minimum qui n'est pas dans T et on l'ajoute à T, et ainsi de suite. On s'arrête lorsque T a $n-1$ arêtes.

8. Soit $G = < S, A, C >$ un graphe non orienté connexe valué. Les algorithmes de recherche d'arbres de recouvrement minimum qu'on a proposés peuvent se décrire dans le cadre plus général de l'*algorithme de coloriage* suivant.

Initialement, toutes les arêtes sont non colorées. On colore les arêtes successivement, soit en bleu (acceptée), soit en rouge (rejetée). Le coloriage se fait en respectant la propriété suivante : il existe un arbre de recouvrement minimum de G qui contient toutes les arêtes bleues et aucune arête rouge (P).

Le coloriage suit les règles suivantes :

(i) choisir un cycle élémentaire ne contenant pas d'arête rouge. Parmi les arêtes non colorées du cycle, choisir une arête de coût maximum et la colorer en rouge.

(ii) choisir un sous-ensemble U de S tel qu'il n'y a pas d'arête bleue entre U et $S-U$; choisir une arête non colorée de coût minimum et la colorer en bleu.

a) Montrer que les règles de coloriage maintiennent la propriété (P) et qu'on obtient bien un arbre de recouvrement minimum, en prenant toutes les arêtes bleues, lorsqu'il ne reste plus d'arête incolore.

b) Montrer que les algorithmes de Prim et de Kruskal sont des cas particuliers de cet algorithme.

Annexe
Outils mathématiques

Annexe
Outils mathématiques

1. Asymptotique

On a présenté au chapitre 2 la notion de complexité d'un algorithme, et on a montré qu'il suffit en général de déterminer l'*ordre de grandeur* de la complexité des algorithmes sur *les données de grande taille* (complexité asymptotique), pour pouvoir, d'une part comparer entre eux différents algorithmes traitant le même problème, et d'autre part évaluer la taille maximum du problème que peut traiter un algorithme.

Ce paragraphe présente les outils mathématiques nécessaires pour l'étude du comportement asymptotique de la complexité des algorithmes : définitions et règles de calcul sur les relations de comparaison entre fonctions; développement asymptotique d'une fonction selon une échelle de comparaison, approximation asymptotique de sommes partielles, telle la formule de Stirling.

1.1. Relations de comparaison

Dans toute la suite du paragraphe 1, les fonctions considérées sont à valeurs positives, ce qui est le cas aussi bien des fonctions de coût que des fonctions de dénombrement. On note $\mathbb{R}^+$ l'ensemble des réels positifs ou nuls et $\mathbb{R}^{+*}$ l'ensemble des réels strictement positifs.

Rappelons d'abord les définitions introduites au chapitre 2.

Définition : Soit f et g deux fonctions de $\mathbb{N}$ dans $\mathbb{R}^+$:

- $f = O(g) \Leftrightarrow \exists c \in \mathbb{R}^{+*}, \exists n_0 \in \mathbb{N}$ tels que
  $\forall n > n_0,\ f(n) \leq c \cdot g(n)$

on dit alors que l'*ordre de grandeur asymptotique de f est inférieur ou égal* à celui de g, ou encore que f est ***dominée asymptotiquement*** par g.

$\bullet f = \Theta(g) \Leftrightarrow \exists c,\ d \in \mathbb{R}^{+*},\ \exists n_0 \in \mathbb{N}$ tels que

$$\forall n > n_0,\ d \cdot g(n) \leq f(n) \leq c \cdot g(n)$$

$$\Leftrightarrow f = O(g) \text{ et } g = O(f)$$

on dit alors que f et g ont un ***même ordre de grandeur asymptotique.***

Remarque : On utilisera abusivement les notations $f(n) = O(g(n))$ et $f(n) = \Theta(g(n))$.

Exemple 1 : Montrons que pour tout entier positif p, la fonction n^p est dominée asymptotiquement de façon stricte par la fonction n^{p+1}, c'est-à-dire :

$$n^p = O(n^{p+1}) \text{ mais } n^p \text{ n'est pas en } \Theta(n^{p+1}).$$

Revenons à la définition. Il est clair que n^p est dominée asymptotiquement par n^{p+1}, puisque n^p est inférieure à n^{p+1} pour tout $n > 1$; d'autre part si n^{p+1} était dominée par n^p cela voudrait dire qu'il existe une constante positive c, et un entier n_0 tel que pour tout $n > n_0$ on ait $n < c$, ce qui est évidemment impossible.

Remarques

a) Il est clair que si $f = \Theta(g)$ alors $f = O(g)$ et $g = \Theta(f)$.

b) O et Θ peuvent être considérées comme des relations entre fonctions : O est la relation «être dominée asymptotiquement par», et Θ la relation «avoir même ordre de grandeur que». On montre facilement que la relation O est réflexive et transitive (relation de préordre); et que la relation Θ est réflexive, symétrique et transitive (relation d'équivalence induite par le préordre O).

c) Il faut être conscient de ce que dans la notation $f = O(g)$, l'utilisation du signe $=$ est abusive : les termes à gauche et à droite de ce signe *ne sont pas de même nature* et «l'égalité» est fortement *dissymétrique.*

On doit donc se garder de calculer mécaniquement avec les notations O et Θ comme s'il s'agissait de nombres ou de fonctions : les égalités $n^2 + n = O(n^2)$ et $2n^2 = O(n^2)$, n'impliquent évidemment pas que $n^2 + n = 2n^2$.

Cependant il existe un certain nombre de règles de calcul simples (cf. paragraphe suivant) qui permettent de déduire l'ordre de grandeur d'une expression fonctionnelle à partir de l'ordre de grandeur de ses composants.

d) Il arrive que la complexité d'un algorithme dépende de plusieurs paramètres, par exemple pour certains algorithmes sur les graphes, la complexité dépend à la fois du nombre de sommets et du nombre d'arêtes; dans ce cas, on définit l'ordre de grandeur d'un algorithme en fonction de plusieurs paramètres, et les notations asymptotiques se généralisent facilement (mais leur manipulation est beaucoup plus

délicate). Donnons, par exemple, la définition de la relation O pour des fonctions à deux arguments :

$$f(n,p) = O(g(n,p)) \Leftrightarrow \exists c \in \mathbb{R}^{+*},\ \exists n_0 \in \mathbb{N},\ \exists p_0 \in \mathbb{N}$$
$$\text{tels que } \forall n > n_0 \text{ et } \forall p > p_0 \quad f(n,p) \leq c.g(n,p)$$

1.1.1. Calcul sur les relations de comparaison

Les règles de calcul suivantes sont des conséquences évidentes des définitions :

R$_0$ a) $g = O(g)$
b) $g = \Theta(g)$

R$_1$ $f = \Theta(g) \Rightarrow g = \Theta(f)$

R$_2$ a) $f = O(g)$ et $g = O(h) \Rightarrow f = O(h)$
b) $f = \Theta(g)$ et $g = \Theta(h) \Rightarrow f = \Theta(h)$

N.B. Les règles **R**$_0$ et **R**$_2$ traduisent la réflexivité et la transitivité des relations O et Θ ; la règle **R**$_1$ traduit la symétrie de la relation Θ.

R$_3$ a) $f = O(g) \Rightarrow \lambda f = O(g)$ où $\lambda \in \mathbb{R}^{+*}$
b) $f = \Theta(g) \Rightarrow \lambda f = \Theta(g)$ où $\lambda \in \mathbb{R}^{+*}$

R$_4$ a) $\left.\begin{array}{l} f_1 = O(g_1) \\ f_2 = O(g_2) \end{array}\right\} \Rightarrow f_1 + f_2 = O(\max(g_1, g_2))$

b) Soient f_1 et f_2 telles que $f_1 - f_2 \geq 0$
$f_1 = O(g_1) \Rightarrow f_1 - f_2 = O(g_1)$

c) $\left.\begin{array}{l} f_1 = \Theta(g_1) \\ f_2 = \Theta(g_2) \end{array}\right\} \Rightarrow f_1 + f_2 = \Theta(\max(g_1, g_2))$

d) Soient f_1 et f_2 telles que $f_1 - f_2 \geq 0$

$$\left.\begin{array}{l} f_1 = \Theta(g_1) \text{ et } f_2 = \Theta(g_2) \\ \lim\limits_{n\to\infty} \dfrac{g_2(n)}{g_1(n)} = 0 \end{array}\right\} \Rightarrow f_1 - f_2 = \Theta(g_1)$$

R$_5$ a) $\left.\begin{array}{l} f_1 = O(g_1) \\ f_2 = O(g_2) \end{array}\right\} \Rightarrow f_1 \cdot f_2 = O(g_1 \cdot g_2)$

b) $\left.\begin{array}{l} f_1 = \Theta(g_1) \\ f_2 = \Theta(g_2) \end{array}\right\} \Rightarrow f_1 \cdot f_2 = \Theta(g_1 \cdot g_2)$

Exemple 2 : En application de ces règles, montrons que pour tout couple d'entiers $(k,\ n)$, on a :

$$\sum_{i=1}^{n} i^k = \Theta\left(n^{k+1}\right) \tag{1}$$

La démonstration se fait par récurrence sur k.

• Pour $k = 1$, on sait que la somme des n premiers entiers vaut :

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$$

or $\quad \dfrac{n^2}{2} = \Theta(n^2),$ et $\dfrac{n}{2} = \Theta(n)$ d'après $\mathbf{R}_3$ b)

donc $\quad \dfrac{n^2}{2} + \dfrac{n}{2} = \Theta(n^2),$ d'après $\mathbf{R}_4$ c)

puisqu'on a vu dans l'exemple 1 que n_2 domine asymptotiquement n de façon stricte.

• Supposons la relation (1) vraie pour tous les j strictement inférieurs à k :

$$\sum_{i=1}^{n} i^j = \Theta\left(n^{j+1}\right) \ \forall j = 1 \ldots k-1$$

Pour montrer qu'elle est vraie à l'ordre k, on développe la somme des i^{k+1} en récrivant chaque terme à l'aide de la formule du binôme :

$$i^{k+1} = \Big((i-1)+1\Big)^{k+1} = \sum_{p=0}^{k+1} \binom{k+1}{p} (i-1)^p$$

Cela donne :

$$\begin{aligned}
n^{k+1} &= (n-1)^{k+1} + (k+1)(n-1)^k + \binom{k+1}{2}(n-1)^{k-1} + \ldots + 1 \\
(n-1)^{k+1} &= (n-2)^{k+1} + (k+1)(n-2)^k + \binom{k+1}{2}(n-2)^{k-1} + \ldots + 1 \\
&\ldots \qquad\qquad\qquad\qquad \ldots \\
2^{k+1} &= 1^{k+1} + (k+1)1^k + \binom{k+1}{2}1^{k-1} + \ldots + 1 \\
1^{k+1} &= 0^{k+1} + (k+1)0^k + \binom{k+1}{2}0^{k-1} + \ldots + 1
\end{aligned}$$

En sommant à gauche et à droite des égalités, on obtient après simplification :

$$n^{k+1} = (k+1)\sum_{i=1}^{n-1} i^k + \binom{k+1}{2}\sum_{i=1}^{n-1} i^{k-1} + \ldots + n$$

c'est-à-dire, en changeant n en $(n+1)$:

$$\sum_{i=1}^{n} i^k = \frac{1}{k+1}(n+1)^{k+1} - \frac{1}{k+1}\binom{k+1}{2}\sum_{i=1}^{n} i^{k-1} - \ldots - \frac{(n+1)}{k+1}$$

$$\text{or } (n+1)^{k+1} = \sum_{p=0}^{k+1}\binom{k+1}{p} n^p = \Theta(n^{k+1}), \text{ d'après } \mathbf{R}_3 \text{ b) et } \mathbf{R}_4 \text{ c)}$$

De plus, d'après l'hypothèse de récurrence,

$$\sum_{i=1}^{n} i^j = \Theta(n^{j+1}) \ \forall j = 1 \ldots k-1$$

et donc, en appliquant plusieurs fois la règle $\mathbf{R}_4$ c) et le résultat de l'exemple 1, on a :

$$\sum_{i=1}^{n} i^k = \Theta(n^{k+1})$$

1.1.2. Critères pour classer les fonctions

On peut souvent comparer deux fonctions en étudiant le comportement de leur quotient quand n devient grand.

Lemme 1 : Soit f et g deux fonctions de $\mathbb{N}$ dans $\mathbb{R}^+$.

$\alpha)$ $\lim_{n\to\infty} \dfrac{f(n)}{g(n)} = a \neq 0 \quad \Rightarrow \ f = \Theta(g)$

$\beta)$ $\lim_{n\to\infty} \dfrac{f(n)}{g(n)} = 0 \quad \Rightarrow \ f = O(g)$ et f n'est pas en $\Theta(g)$

$\gamma)$ $\lim_{n\to\infty} \dfrac{f(n)}{g(n)} = \infty \quad \Rightarrow \ g = O(f)$ et g n'est pas en $\Theta(f)$

Preuve : Il suffit d'appliquer la définition d'une limite ; faisons-le, par exemple, pour le cas $\alpha)$. Si le rapport $\dfrac{f(n)}{g(n)}$ tend vers une limite non nulle a (nécessairement $a > 0$ puisque f et g sont à valeurs positives), par définition d'une limite on a :

$$\forall \varepsilon > 0,\ \exists n_0 \in \mathbb{N} \text{ tel que } \forall n > n_0,\ \left| \frac{f(n)}{g(n)} - a \right| < \varepsilon$$

c'est-à-dire :

$$\forall \varepsilon > 0,\ \exists n_0 \in \mathbb{N} \text{ tel que } \forall n > n_0,\ (a - \varepsilon) \cdot g(n) < f(n) < (a + \varepsilon) \cdot g(n)$$

Il suffit donc de choisir ε assez petit pour que $a - \varepsilon$ et $a + \varepsilon$ soient des réels positifs, pour pouvoir conclure que $f = \Theta(g)$. □

Remarque 1 :

a) Lorsque $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 1$, on dit en analyse mathématique que les fonctions f et g sont ***asymptotiquement équivalentes***, et l'on note $f \sim g$; on peut par conséquent récrire $\alpha)$ sous la forme :

$$f \sim ag\ (a > 0) \Rightarrow f = \Theta(g).$$

b) Lorsque $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$, on dit en analyse mathématique que la fonction f est ***négligeable*** devant la fonction g, et l'on note $f = o(g)$; on peut par conséquent récrire $\beta)$ sous la forme :

$$f = o(g) \Rightarrow f = O(g) \text{ et } f \text{ n'est pas en } \Theta(g).$$

Remarque 2 : Le lemme 1 est un ***critère suffisant*** pour classer les fonctions : les réciproques des propriétés $\alpha)$ $\beta)$ $\gamma)$ sont fausses; dans le cas où le rapport $f(n)/g(n)$ n'a pas de limite quand n devient grand, il faut revenir aux définitions de O et de Θ pour pouvoir comparer f et g.

Par exemple :

soit $f(n) = n$ si n impair
$\quad\quad\quad\ \ 2n$ si n pair
et $g(n) = 2n$ pour tout entier n

alors $\frac{f(n)}{g(n)}$ n'a pas de limite, mais $f = \Theta(g)$, puisque pour tout entier n, on a :

$$\frac{1}{2}\, g(n) \leq f(n) \leq g(n).$$

Le lemme 1 ne s'applique pas lorsque le quotient $\frac{f(n)}{g(n)}$ a une limite indéterminée $\left(\frac{0}{0}$ ou $\frac{\infty}{\infty}\right)$; dans certains cas on peut lever l'indétermination grâce à la *règle de l'Hospital*, et appliquer ensuite le lemme 1.

Lemme 2 (Règle de l'Hospital) : Soit f et g des fonctions de $\mathbb{N}$ dans $\mathbb{R}^+$ telles que : $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = \infty$; alors si $\lim_{n\to\infty} \frac{f'(n)}{g'(n)}$ existe, on a :

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}.$$

Par exemple, en utilisant deux fois la règle de l'Hospital, on peut montrer que n^2 est négligeable devant e^n :

$$\lim_{n\to\infty} \frac{n^2}{e^n} = \lim_{n\to\infty} \frac{2n}{e^n} = \lim_{n\to\infty} \frac{2}{e^n} = 0.$$

1.2. Développement asymptotique selon une échelle de comparaison

Pour avoir une bonne approximation d'une fonction au voisinage de l'infini, on essaie de la situer par rapport à un ensemble de fonctions de référence.

1.2.1. Echelle de comparaison

Définition : on appelle ***échelle de comparaison*** un ensemble E de fonctions de $\mathbb{N}$ dans $\mathbb{R}^+$ telles que :

1. $\forall g \in E$: $g(n) > 0$ quand n devient grand
2. $\forall g \in E$: si $g \neq 1$ alors $\lim_{n\to\infty} g(n) = 0$ ou ∞
3. $\forall g,\ h \in E$: si $g \neq h$ alors ou bien $g = o(h)$ (*i.e.* g est négligeable devant h)
   ou bien $h = o(g)$

Remarquons à partir de cette définition que dans une échelle de comparaison on ne peut avoir qu'un seul élément par classe d'équivalence modulo la relation Θ.

Exemples d'échelles de comparaison

- l'ensemble E_1 des fonctions n^γ, $\gamma \in \mathbb{R}$
- l'ensemble E_2 des fonctions $n^\gamma(\mathrm{Log}\, n)^\delta$, $\gamma,\ \delta \in \mathbb{R}$
- l'ensemble E_3 des fonctions $\exp(\alpha n^\beta) \cdot n^\gamma \cdot (\mathrm{Log}\, n)^\delta$, $\alpha,\ \gamma,\ \delta \in \mathbb{R}$ et $\beta > 0$

(On note indifféremment $\exp(x)$ ou e^x.)

Pour comparer deux fonctions

$$f_1(n) = \exp(\alpha_1 n^{\beta_1}) \cdot n^{\gamma_1} \cdot (\mathrm{Log}n)^{\delta_1} \text{ et } f_2(n) = \exp(\alpha_2 n^{\beta_2}) \cdot n^{\gamma_2} \cdot (\mathrm{Log}n)^{\delta_2},$$

telles que $\beta_1 \geq \beta_2 > 0$,

on procède de la façon suivante.

• Si $\beta_1 > \beta_2$, alors si $\alpha_1 > 0$: $f_2 = o(f_1)$
et si $\alpha_1 < 0$: $f_1 = o(f_2)$

• Si $\beta_1 = \beta_2$, alors on compare les mots $\alpha_1\gamma_1\delta_1$ et $\alpha_2\gamma_2\delta_2$ selon l'ordre lexicographique (c'est-à-dire, l'ordre des mots d'un dictionnaire) :

si $\alpha_1\gamma_1\delta_1 > \alpha_2\gamma_2\delta_2$ alors $f_2 = o(f_1)$
si $\alpha_1\gamma_1\delta_1 < \alpha_2\gamma_2\delta_2$ alors $f_1 = o(f_2)$

• On peut encore élargir l'échelle de comparaison précédente en y incluant des logarithmes imbriqués (Log...(Log n)) et des exponentielles de fonctions de n exp $(P(n))$ avec $P(n)$ polynôme en n de degré > 1, ou exp (... exp (n))...

1.2.2. Développement asymptotique

Etant donnée une échelle de comparaison E , il s'agit maintenant de comparer aux fonctions de E une fonction donnée f.

a) On cherche donc s'il existe une fonction g_1 appartenant à E telle que $f \sim a_1 g_1$ (a_1 est une constante réelle non nulle). Si une telle fonction g_1 existe, elle est unique, puisque les fonctions d'une échelle de comparaison ont des ordres de grandeur tous différents; on dit alors que f a pour *partie principale* $a_1 g_1$ par rapport à l'échelle E et l'on peut écrire $f = a_1 g_1 + o(g_1)$.

Une fonction f n'admet pas toujours de partie principale par rapport à une échelle

- soit pour des raisons intrinsèques à la fonction f :

  (i) f s'annule pour des valeurs de n arbitrairement grandes. C'est le cas, par exemple, de la fonction $n\sin(\pi\frac{n}{2})$; alors elle n'admet pas de partie principale (d'après la première partie de la définition d'une échelle de comparaison);

  (ii) f n'a pas de limite quand n devient grand. C'est le cas par exemple de la fonction $1 + n\sin|(\pi\frac{n}{2})|$; alors elle n'admet pas de partie principale (d'après la seconde partie de la définition d'une échelle de comparaison).

- soit parce que l'échelle de comparaison considérée n'est pas assez complète : par exemple, les fonctions exp(exp n) et n^n n'admettent pas de partie principale par rapport à l'échelle E_3 du paragraphe précédent.

b) Dans le cas où f admet pour partie principale $a_1 g_1$, on peut essayer d'approcher f de façon plus précise en comparant $(f - a_1 g_1)$ aux fonctions de E; si $f - a_1 g_1$

admet une partie principale $a_2 g_2$ alors on a nécessairement $g_2 = o(g_1)$, et l'on peut écrire $f = a_1 g_1 + a_2 g_2 + o(g_2)$.

D'une façon générale, on appelle ***développement asymptotique à k termes***, de la fonction f ***par rapport à une échelle*** E, une expression de la forme :

$$f = a_1 g_1 + a_2 g_2 + \ldots + a_k g_k + o(g_k)$$

où les a_i sont des constantes réelles non nulles, et les g_i des fonctions de E telles que $g_{i+1} = o(g_i)$ pour $i = 1 \ldots k-1$. Le développement à k termes, s'il existe, est unique par construction.

Par exemple, la fonction $f(n) = 3n^2 - n\log n + n\sin\left(\frac{\pi n}{2}\right)$ admet, par rapport à l'échelle E_3 un développement asymptotique à deux termes :

$$f(n) = 3n^2 - n\log n + o(n\log n)$$

mais $f(n)$ n'a pas de développement asymptotique à trois termes.

Il existe aussi des fonctions pour lesquelles on peut trouver des développements asymptotiques à tout ordre. C'est par exemple le cas pour les fonctions $\exp(\frac{1}{n})$ ou $\left(1+\frac{1}{n}\right)^{\alpha}$, dont le développement de Taylor peut être poussé arbitrairement loin.

c) Développement de Taylor.
On peut aussi considérer le développement asymptotique d'une fonction $f(x)$ au voisinage d'un point x_0 selon l'échelle des puissances de x : c'est le *développement de Taylor* de $f(x)$. Le développement de Taylor de $f(x)$ à l'ordre k, qui est valable pour toute fonction k fois continûment dérivable dans un voisinage de x_0 , est donné par la formule :

$$f(x) = f(x_0) + f'(x_0)(x-x_0) + \ldots + f^{(k)}(x_0)\frac{(x-x_0)^k}{k!} + o(x-x_0)^k$$

Rappelons les développements de Taylor *au voisinage de 0* les plus utilisés, qui sont valables pour tout $k \geq 0$:

$$(1+x)^{\alpha} = 1 + \alpha x + \alpha(\alpha-1)\frac{x^2}{2!} + \ldots + \alpha(\alpha-1)\ldots(\alpha-k+1)\frac{x^k}{k!} + o(x^k), \alpha \in \mathbb{R}^*$$

$$\exp(x) = 1 + x + \frac{x^2}{2!} + \ldots + \frac{x^k}{k!} + o(x^k)$$

$$\mathrm{Log}\,(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} + \ldots + (-1)^{k+1}\frac{x^k}{k} + o(x^k)$$

Remarque : Calculs sur les développements asymptotiques

Connaissant les développements asymptotiques de deux fonctions f_1 et f_2, on obtient facilement les développements de $f_1 + f_2$ et $f_1 \cdot f_2$ en appliquant des règles de calcul

du type de celles qui ont été données au §1.1; il en est de même pour f_1^p avec p entier; pour f_1^α avec α réel ($f_1 > 0$), ou pour Log f_1, ou pour exp(f_1), la méthode peut aussi s'appliquer mais elle est plus compliquée.

1.3. Approximations asymptotiques de sommes partielles

On donne dans ce paragraphe deux exemples bien connus d'estimation asymptotique reposant sur l'approximation d'une somme partielle par une intégrale :

- la formule de Stirling : $n! \sim \sqrt{2\pi n} \cdot \frac{n^n}{e^n}$;
- l'équivalent asymptotique du $n^{ième}$ nombre harmonique H_n : $H_n \sim \text{Log}n$, où $H_n = 1 + \frac{1}{2} + \ldots + \frac{1}{n}$.

Lorsqu'une fonction est monotone, on peut toujours encadrer sa somme par son intégrale; si de plus la fonction varie lentement, alors la somme et l'intégrale ont le même ordre de grandeur asymptotique.

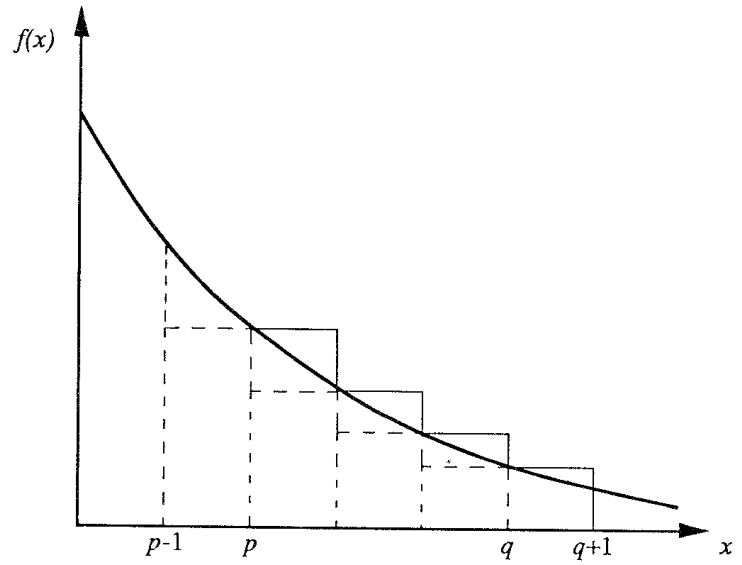


Figure 1. Approximations d'une fonction décroissante.

a) Pour toute fonction monotone décroissante, on a clairement (cf. figure 1) :

$$\int_p^{q+1} f(x)\,dx \leq \sum_{i=p}^{q} f(i) \leq \int_{p-1}^{q} f(x)\,dx \qquad (2)$$

Exemple : Considérons la fonction monotone décroissante $f(x) = \frac{1}{x}$. D'après (2) :

$$\int_2^{n+1} \frac{1}{x}\, dx \leq \sum_{i=2}^{n} \frac{1}{i} \leq \int_1^n \frac{1}{x}\, dx$$

d'où l'on tire l'encadrement suivant pour le $n^{ième}$ nombre harmonique H_n :

$$\mathrm{Log}(n+1) + (1 - \mathrm{Log}\, 2) \leq H_n \leq \mathrm{Log}\, n + 1$$

On a donc montré que $H_n = \Theta(\mathrm{Log}\, n)$; une étude plus compliquée permet de trouver un développement asymptotique d'ordre 3 de H_n. On en donne simplement le résultat :

$$H_n = \mathrm{Log}\, n + \gamma + \frac{1}{2n} + o\left(\frac{1}{n}\right)$$

où γ est la constante d'Euler : $\gamma = 0,57721\ldots$

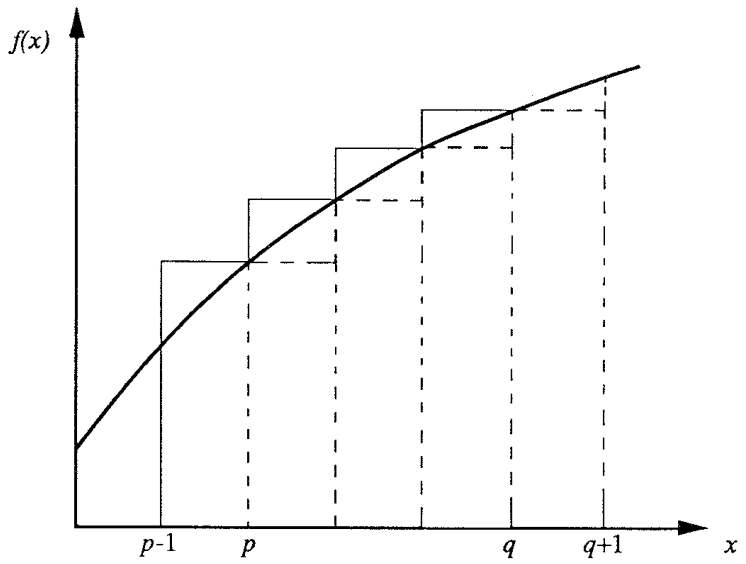


Figure 2. Approximations d'une fonction croissante.

b) Pour toute fonction monotone croissante, on a clairement (cf. figure 2) :

$$\int_{p-1}^{q} f(x)\, dx \leq \sum_{i=p}^{q} f(i) \leq \int_p^{q+1} f(x)\, dx \qquad (3)$$

Exemple : Encadrement de $n!$. On encadre en fait $\mathrm{Log}(n!) = \sum_{i=1}^{n} \mathrm{Log}\, i$.

D'après (3) on a :

$$\int_1^n \mathrm{Log}\, x\ dx \leq \sum_{i=1}^{n} \mathrm{Log}\, i \leq \int_1^{n+1} \mathrm{Log}\, x\ dx$$

or, en intégrant par parties, on obtient :

$$\int_1^n \mathrm{Log}\, x\ dx = [x\mathrm{Log}\, x]_1^n - \int_1^n dx$$

d'où l'on déduit l'encadrement suivant pour $\mathrm{Log}(n!)$:

$$n\mathrm{Log}\, n - n + 1 \leq \mathrm{Log}(n!) \leq (n+1)\mathrm{Log}(n+1) - n$$

La fonction exponentielle étant croissante, on conserve les inégalités en prenant l'exponentielle des deux membres, et l'on obtient :

$$e^{1-n}n^n \leq n! \leq e^{-n}(n+1)^{n+1}$$

D'après la borne supérieure de l'encadrement, on a donc $n! = O(n^{n+1})$; mais cet encadrement n'est pas suffisant pour déduire l'ordre de grandeur exact de $n!$ (qui est $n^{n+1/2}$), car la fonction exponentielle est à croissance trop rapide.

La démonstration de la formule de Stirling, qui est un développement asymptotique de $n!$, nécessite des techniques beaucoup plus complexes ; on se contente de présenter le résultat :

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n} + o\left(\frac{1}{n}\right)\right)$$

2. Séries génératrices

L'utilisation de séries génératrices permet de représenter une suite infinie de nombres par une fonction, qui est en général plus facile à manipuler que la suite infinie. En particulier, si la suite de nombres est définie par une relation de récurrence, cette relation se traduit par une équation fonctionnelle sur la série génératrice, et l'on peut souvent résoudre cette équation par des techniques classiques d'algèbre ou d'analyse.

On présente ici les bases de la manipulation des séries génératrices, et leur application en analyse de la complexité des algorithmes.

2.1. Séries génératrices ordinaires

Définition : Soit $(a_n)_{n\geq 0}$ une suite de nombres, on appelle ***série génératrice (ordinaire)*** de coefficient a_n la série $a(z) = \sum_{n\geq 0} a_n z^n$.

N.B. z est une variable quelconque, dont les puissances servent à différencier les éléments de la suite.

Indépendamment de toute considération de convergence de la série, on peut définir formellement un certain nombre d'opérations sur les séries génératrices, et utiliser des méthodes de calcul algébrique; c'est ce qui est fait dans le paragraphe suivant.

Remarque : Lorsque, de plus, la série génératrice converge pour certaines valeurs de la variable z, l'utilisation d'outils analytiques permet d'obtenir la valeur exacte, ou une estimation asymptotique des coefficients de la série.

Notons aussi que toute fonction développable en série entière détermine une suite de nombres dont elle est la série génératrice, puisque le développement en série est unique. Le lecteur trouvera dans le formulaire une liste des développements en série des fonctions usuelles.

Soit $(a_n)_{n\geq 0}$, $(b_n)_{n\geq 0}$, $(c_n)_{n\geq 0}$ des suites de nombres, et soient $a(z)$, $b(z)$ et $c(z)$ leurs séries génératrices associées, on étudie la correspondance entre opérations sur les suites de nombres et opérations sur les séries génératrices ordinaires.

2.1.1. Somme de deux séries

Définition : Etant donnés deux réels α et β, alors $c(z) = \alpha a(z) + \beta b(z)$ si, et seulement si, $c_n = \alpha a_n + \beta b_n$, pour tout $n \geq 0$.

Exemple : Soit $(a_n)_{n\geq 0}$ la suite constante $a_n = 1$ pour tout $n \geq 0$, et soit $(b_n)_{n\geq 0}$ la suite telle que $b_0 = 0$ et $b_n = 1$ pour tout $n \geq 1$;

alors $a(z) = \sum_{n\geq 0} a_n z^n = \sum_{n\geq 0} z^n$ et $b(z) = \sum_{n\geq 0} b_n z^n = \sum_{n\geq 1} z^n = z \cdot a(z)$

La différence $c(z) = a(z) - b(z)$ a tous ses coefficients nuls, sauf le coefficient c_0 qui est égal à 1; et donc $c(z) = 1$.

D'où $a(z)(1 - z) = 1$, et l'on retrouve le développement bien connu :

$$\frac{1}{1-z} = 1 + z + z^2 + \ldots + z^n + \ldots \qquad (1)$$

et donc par changement de variable, pour tout réel λ :

$$\frac{1}{1-\lambda z} = \sum_{n\geq 0} \lambda^n z^n. \tag{2}$$

2.1.2. *Produit de Cauchy*

Définition : $c(z)$ est le ***produit de Cauchy*** (ou produit de convolution) des séries $a(z)$ et $b(z)$, ce que l'on note $c(z) = a(z) \cdot b(z)$, si, et seulement si,

$$\text{pour tout } n \geq 0 \quad c_n = \sum_{p=0}^{n} a_p b_{n-p}.$$

Cas particuliers

• Si la série $b(z)$ se réduit à un seul terme, le produit est un décalage :

$c(z) = a(z) \cdot z^k$ si et seulement si $c_n = a_{n-k}$, pour $n \geq k$, et 0 sinon.

Par exemple, on peut montrer ainsi que $\displaystyle\sum_{n=0}^{p} z^n = \frac{1-z^{p+1}}{1-z}$. (3)

en effet : $\displaystyle\sum_{n=0}^{p} z^n = \sum_{n\geq 0} z^n - \sum_{n\geq p+1} z^n = \frac{1}{1-z} - \frac{z^{p+1}}{1-z}$.

• $c(z) = a(z) \cdot (1+z)$ si, et seulement si, $c_n = a_n + a_{n-1}$.

Comme application, le lecteur est invité à retrouver par récurrence la formule du binôme pour tout entier p :

$$(1+z)^p = \sum_{n\geq 0} \binom{p}{n} z^n. \tag{4}$$

• $c(z) = a(z) \cdot \dfrac{1}{1-z}$ si, et seulement si, $c_p = \displaystyle\sum_{n=0}^{p} a_n$.

Par exemple $\dfrac{1}{(1-z)^2} = \sum_{n\geq 0}(n+1)z^n$, et plus généralement on peut montrer que

pour tout $k \in \mathbb{N}$, $\dfrac{1}{(1-z)^k} = \displaystyle\sum_{n\geq 0} \binom{n+k-1}{k-1} z^n$. (5)

2.1.3. *Dérivation et intégration*

• La série dérivée est obtenue en dérivant terme à terme la série initiale :

$c(z) = a'(z)$ si, et seulement si, $c_n = (n+1)a_{n+1}$, pour $n \geq 0$.

Par exemple, la dérivée de $\frac{1}{1-z}$ est $\frac{1}{(1-z)^2}$ et donc $\frac{1}{(1-z)^2} = \sum_{n\geq 0}(n+1)z^n$ et l'on peut retrouver la formule (5) par dérivations successives.

• On peut de même intégrer terme à terme une série :

$$c(z) = \int_0^z a(t)\,dt \text{ si et seulement si } c_n = \frac{1}{n}a_{n-1}, \text{ pour tout } n \geq 1, \text{ et } c_0 = 0.$$

Par exemple, la primitive de $\frac{1}{1-z}$ est $\mathrm{Log}\left(\frac{1}{1-z}\right)$, et donc $\mathrm{Log}\left(\frac{1}{1-z}\right) = \sum_{n\geq 1}\frac{z^n}{n}$

Comme exemple d'application de cette dernière série, on peut obtenir la série génératrice des nombres harmoniques en utilisant la règle du produit :

$$\frac{1}{1-z}\mathrm{Log}\left(\frac{1}{1-z}\right) = \sum_{n\geq 1} H_n z^n \tag{6}$$

et en dérivant chacun des termes de l'équation (6), on montre une formule classique sur les nombres harmoniques :

$$\sum_{p=1}^{n} H_p = (n+1)H_n - n.$$

$c_n = \alpha a_n + \beta b_n$	$\Leftrightarrow$	$c(z) = \alpha \cdot a(z) + \beta \cdot b(z)$
$c_n = \sum_{p=0}^{n} a_p b_{n-p}$	$\Leftrightarrow$	$c(z) = a(z) \cdot b(z)$
$c_n = a_{n-k}$	$\Leftrightarrow$	$c(z) = z^k \cdot a(z)$
$c_n = (n+1)a_{n+1}$	$\Leftrightarrow$	$c(z) = a'(z)$
$c_n = \frac{1}{n}\,a_{n-1}$	$\Leftrightarrow$	$c(z) = \int_0^z a(t)dt$

Figure 3. Correspondance entre suites et séries génératrices ordinaires.

2.2. Séries génératrices exponentielles

A toute suite $(a_n)_{n\geq 0}$, on peut également associer sa ***série génératrice exponentielle*** $\hat{a}(z) = \sum_{n\geq 0} a_n \frac{z^n}{n!}$.

Ainsi la série génératrice exponentielle de la suite $(a_n)_{n\geq 0}$ est la série génératrice ordinaire de la suite $\left(\frac{a_n}{n!}\right)_{n\geq 0}$.

Soit $\hat{a}(z)$, $\hat{b}(z)$ et $\hat{c}(z)$ les séries génératrices exponentielles associées aux suites $(a_n)_{n\geq 0}$, $(b_n)_{n\geq 0}$, $(c_n)_{n\geq 0}$; on peut établir une correspondance entre les opérations sur les suites et les opérations sur les séries génératrices exponentielles (voir figure 4).

Somme de deux séries exponentielles

$\hat{c}(z) = \hat{a}(z) + \hat{b}(z)$ si, et seulement si, $c_n = a_n + b_n$, pour tout $n \geq 0$

Produit de deux séries exponentielles

$\hat{c}(z) = \hat{a}(z) \cdot \hat{b}(z)$ si, et seulement si $c_n = \sum_{p=0}^{n} \binom{n}{p} a_p b_{n-p}$, pour tout $n \geq 0$.

Montrons cette formule de produit en utilisant le produit de Cauchy des séries ordinaires :

$$\hat{c}(z) = \sum_{n\geq 0} \frac{1}{n!} a_n z^n \cdot \sum_{n\geq 0} \frac{1}{n!} b_n z^n = \sum_{n\geq 0} \left(\sum_{p=0}^{n} \frac{1}{p!(n-p)!}\, a_p \cdot b_{n-p} \right) z^n,$$

et l'on obtient le résultat, en multipliant le terme de droite par $n!$ en haut et en bas.

Dérivation

$\hat{c}(z) = \hat{a}'(z)$ si, et seulement si, $c_n = a_{n+1}$, pour tout $n \geq 0$:

en effet $\hat{a}'(z) = \sum_{n\geq 0} n a_n \frac{z^{n-1}}{n!} = \sum_{n\geq 1} a_n \cdot \frac{z^{n-1}}{(n-1)!}$

Intégration

$\hat{c}(z) = \int_0^z \hat{a}(t)\, dt$ si, et seulement si $c_n = a_{n-1}$, pour tout $n \geq 1$, et $c_0 = 0$:

en effet $\int_0^z \hat{a}(t)\, dt = \sum_{n\geq 0} \int_0^z a_n \frac{t^n}{n!}\, dt = \sum_{n\geq 0} a_n \frac{z^{n+1}}{(n+1)!}$

Un exemple de série génératrice exponentielle est donné par le développement en série de la fonction exponentielle, notée $\exp(z)$: $\exp(z) = \sum_{n\geq 0} \frac{z_n}{n!}$; c'est la série génératrice exponentielle de la suite dont tous les termes sont égaux à 1.

$c_n = \alpha a_n + \beta b_n$	$\Leftrightarrow$	$\hat{c}(z) = \alpha \cdot \hat{a}(z) + \beta \cdot \hat{b}(z)$
$c_n = \sum_{p=0}^{n} \binom{n}{p} a_p b_{n-p}$	$\Leftrightarrow$	$\hat{c}(z) = \hat{a}(z) \cdot \hat{b}(z)$
$c_n = a_{n+1}$	$\Leftrightarrow$	$\hat{c}(z) = \hat{a}'(z)$
$c_n = a_{n-1}$	$\Leftrightarrow$	$\hat{c}(z) = \int_0^z \hat{a}(t)dt$

Figure 4. Correspondance entre suites et séries génératrices exponentielles.

2.3. Exemples d'utilisation

2.3.1. Résolution de récurrences

Les séries génératrices sont un outil puissant pour résoudre des équations de récurrence (cf. paragraphe 3). On donne ici un exemple de résolution d'une récurrence non linéaire : celle du nombre d'arbres binaires de taille n.

On rappelle la définition de l'ensemble des arbres binaires donnée au chapitre 7 :

$$\mathbb{B} = \emptyset + < o,\ \mathbb{B},\ \mathbb{B} > \tag{7}$$

Soit b_n le nombre d'arbres binaires ayant n nœuds, et soit $b(z) = \sum_{n \geq 0} b_n z^n$ la série génératrice associée à la suite (b_n).

On dit que $b(z)$ est la *série génératrice de dénombrement* des arbres binaires.

D'après l'équation (7) il y a un arbre vide, et tout arbre non vide de n nœuds est formé d'une racine, et de deux sous-arbres, gauche et droit, de taille quelconque inférieure ou égale à $n-1$, telle que la somme de ces tailles vaut $n-1$. Cela se traduit par la récurrence suivante : $b_0 = 1$, et pour $n \geq 1$,

$$b_n = \sum_{k=0}^{n-1} b_k \cdot b_{n-1-k}.$$

En utilisant cette définition récurrente de la suite (b_n) dans la série génératrice $b(z)$, on obtient :

$$b(z) = 1 + \sum_{n \geq 1} \Big(\sum_{k=0}^{n-1} b_k \cdot b_{n-k-1} \Big) z^n,$$

soit en posant $m = n - 1$, et d'après la définition du produit de Cauchy :

$$b(z) = 1 + z \cdot (b(z))^2 \tag{8}$$

La résolution de cette équation du deuxième degré en $b(z)$ donne :

$$b(z) = \frac{1 - \sqrt{1 - 4z}}{2z} \tag{9}$$

(On élimine la racine $b(z) = \frac{1 + \sqrt{1 - 4z}}{2z}$ pour des raisons de cohérence avec la valeur de $b(z)$ pour $z = 0$.)

En utilisant le développement en série entière de $(1+x)^\alpha$, valable pour toute valeur réelle de α :

$$(1 + x)^\alpha = \sum_{k \geq 0} \frac{\alpha(\alpha - 1) \ldots (\alpha - k + 1)}{k!} x^k,$$

on obtient :

$$\sqrt{1 - 4z} = 1 + \sum_{n \geq 1} \frac{(-1)^{n-1}}{2^n} (-4)^n \cdot 1 \cdot 3 \cdot 5 \ldots (2n - 3) \cdot \frac{z^n}{n!}$$

d'où, en multipliant le numérateur et le dénominateur de chaque terme de la somme par $2 \cdot 4 \cdot 6 \ldots (2n - 2)$, il vient :

$$\sqrt{1 - 4z} = 1 + \sum_{n \geq 1} (-4)^n \cdot (-1)^{n-1} \cdot \frac{2}{4^n} \cdot \frac{(2n - 2)!}{(n - 1)! n!} \cdot z^n, \text{ et donc}$$

$$b(z) = \sum_{n \geq 0} \frac{1}{n + 1} \binom{2n}{n} z^n. \tag{10}$$

Ainsi le coefficient de z^n dans la série $b(z)$, que l'on note $[z^n]b(z)$, est égal au $n^{ième}$ nombre de Catalan $c_n = \frac{1}{n+1}\binom{2n}{n}$. Les nombres de Catalan apparaissent dans le dénombrement de beaucoup d'objets combinatoires (cf. exercices). Leurs premières valeurs sont données par le tableau suivant

n	0	1	2	3	4	5	6	7	8
c_n	1	1	2	5	14	42	132	429	1430

Pour les grandes valeurs de n, l'application de la formule de Stirling (cf. paragraphe 1) donne un équivalent asymptotique de c_n :

$$c_n \sim \frac{1}{\sqrt{\pi}} \cdot 4^n \cdot n^{-\frac{3}{2}} \tag{11}$$

Remarque : Dans beaucoup de problèmes combinatoires, une manipulation un peu différente des séries génératrices de dénombrement permet de trouver les équations fonctionnelles vérifiées par ces séries directement à partir de la définition de construction des objets, sans passer par des relations de récurrence.

Reprenons avec cette méthode le dénombrement des arbres binaires : dans la définition de la série génératrice $b(z)$, chaque puissance $n^{ième}$ de z apparaît autant de fois qu'il y a d'arbres binaires de taille n, on peut donc écrire aussi :

$b(z) = \sum_{A \in \mathbb{B}} z^{|A|}$, où $| A |$ est le nombre de nœuds de l'arbre binaire A.

Alors, on obtient à partir de la définition (7) de l'ensemble $\mathbb{B}$ des arbres binaires :

$$b(z) = z^0 + \sum_{A \in \mathbb{B} - \emptyset} z^{|A|} = 1 + \sum_{A = <o, A_1, A_2>} z^{1+|A_1|+|A_2|}$$
$$= 1 + z \cdot \sum_{A_1, A_2 \in \mathbb{B}} z^{|A_1|+|A_2|} = 1 + z \cdot \sum_{A_1 \in \mathbb{B}} z^{|A_1|} \cdot \sum_{A_2 \in \mathbb{B}} z^{|A_2|}$$

Ainsi, on a trouvé directement à partir de la définition de $\mathbb{B}$, l'équation vérifiée par la série génératrice : $b(z) = 1 + z(b(z))^2$, à partir de laquelle on peut obtenir les coefficients b_n comme précédemment.

2.3.2. Dénombrements combinatoires

L'utilisation de séries génératrices permet d'unifier et de généraliser les dénombrements classiques. Rappelons tout d'abord les définitions de base. Etant donné une collection d'éléments *distincts*,

- une ***combinaison*** de p éléments parmi n est le résultat du choix de p éléments, l'ordre de sélection n'intervenant pas;

- un ***arrangement*** de p éléments parmi n est le résultat du choix *ordonné* de p éléments; deux arrangements sont considérés comme différents si l'ordre de sélection diffère, même si *globalement* les éléments choisis sont identiques.

Les résultats suivants sont bien connus :

- le nombre de combinaisons de p éléments parmi n est noté $\binom{n}{p}$. On a :

$$\binom{n}{p} = \frac{n!}{p!(n-p)!} \quad \text{pour } 0 \leq p \leq n \text{ et}$$
$$= 0 \text{ pour } p > n.$$

On le note encore C_n^p.

• le nombre d'arrangements de p éléments parmi n est $p!\binom{n}{p}$. On a :

$$p!\binom{n}{p} = \frac{n!}{(n-p)!} \quad \text{pour } 0 \leq p \leq n \text{ et}$$
$$= 0 \text{ pour } p > n.$$

On le note encore A_n^p.

L'introduction de séries génératrices permet de traiter de manière semblable un grand nombre de dénombrements combinatoires.

Etant donnés n éléments distincts $e_1, \ldots, e_n$, on peut coder l'ensemble (resp. le nombre) des configurations d'éléments vérifiant une propriété P, par une ***série génératrice d'énumération*** $SE(t_1, \ldots, t_n)$ (resp. par une ***série génératrice de dénombrement*** $SD(t)$).

Le codage se fait de la façon suivante : soit $c = (e_{i_1}, \ldots, e_{i_k})$, une configuration de taille k et soit d_j le nombre d'occurrences de l'élément e_j dans c; on fait correspondre à c le terme $t_1^{d_1} \cdot \ldots \cdot t_n^{d_n}$, $(d_1 + \ldots + d_n = k)$ dans la série d'énumération $SE(t_1, \ldots, t_n)$ et le terme $t_1^{d_1+\ldots+d_n} = t^k$ dans la série de dénombrement $SD(t)$. Par exemple si $n = 4$, on fait correspondre à la configuration (e_1, e_2, e_1, e_4) le terme $t_1^2 \cdot t_2 \cdot t_4$ dans la série $SE(t_1, t_2, t_3, t_4)$, et le terme t^4 dans la série $SD(t)$.

On a donc :

$$SE(t_1, \ldots, t_n) = \sum_{k \geq 0} \sum_{d_1+\ldots+d_n=k} nb_{d_1,\ldots,d_n} \cdot t_1^{d_1} t_2^{d_2} \ldots t_n^{d_n},$$

où $nb_{d_1,\ldots,d_n}$ est le nombre de configurations vérifiant la propriété P, avec d_1 fois l'élément $e_1, \ldots, d_n$ fois l'élément e_n, et

$$SD(t) = SE(t, \ldots, t).$$

Le nombre de configurations de taille k vérifiant P est le coefficient de t^k dans $SD(t)$.

On va présenter les séries génératrices d'énumération et de dénombrement pour des types de configurations classiques. On retrouve ainsi des résultats bien connus.

Exemple 1 : Combinaisons

Soit $CE(t_1, \ldots, t_n)$ la série d'énumération des combinaisons de k éléments parmi n. Le nombre $nb_{d_1,\ldots,d_n}$ est nul si l'un des d_i est différent de 0 ou de 1, et est égal à 1 sinon.

D'où :

$$CE(t_1, \ldots, t_n) = \sum_{k \geq 0} \sum_{\substack{d_1+\ldots+d_n=k \\ d_1,\ldots,d_n \in \{0,1\}}} t_1^{d_1} t_2^{d_2} \ldots t_n^{d_n}.$$

En développant, on obtient :

$$CE(t_1, \ldots, t_n) = 1 + (t_1 + \ldots + t_n) + (\sum_{i<j} t_i \cdot t_j) + (\sum_{i<j<k} t_i \cdot t_j \cdot t_k) + \ldots + (t_1 \cdot \ldots \cdot t_n)$$

D'où :

$$CE(t_1, \ldots, t_n) = (1 + t_1) \cdot (1 + t_2) \cdot \ldots \cdot (1 + t_n).$$

La série de dénombrement est donc

$$CD(t) = (1 + t)^n$$

Le coefficient de t^k dans $(1 + t)^n$ est le nombre de configurations où l'on choisit k éléments, c'est-à-dire, le nombre de combinaisons de k éléments parmi n. On retrouve donc le résultat bien connu (formule du binôme) :

$$(1 + t)^n = \sum_{k=0}^{n} \binom{n}{k} t^k.$$

Remarque : On a vu que $CE(t_1, \ldots, t_n) = (1 + t_1) \cdot (1 + t_2) \cdot \ldots \cdot (1 + t_n)$. Ce résultat aurait pu s'obtenir directement par le raisonnement suivant.

Dans une combinaison d'éléments pris parmi n, chacun des n éléments peut être choisi ou non choisi : pour un élément e_i il n'y a que deux configurations possibles et la série d'énumération pour un élément e_i est donc réduite à $(t_i + 1)$. De plus le choix d'un élément est totalement indépendant du choix des autres éléments, et ceci se traduit au niveau des séries génératrices par un produit. L'ensemble des configurations possibles des combinaisons prises parmi n éléments est donc codé par la série :

$$CE(t_1, \ldots, t_n) = (1 + t_1) \cdot (1 + t_2) \cdot \ldots \cdot (1 + t_n).$$

Exemple 2 : Combinaisons avec répétitions

Soit $CRE(t_1, \ldots, t_n)$ la série d'énumération des combinaisons avec répétitions de k éléments parmi n éléments. Le nombre $nb_{d_1,\ldots,d_n}$ est égal à 1 pour tout $d_i \geq 0$.

D'où :

$$CRE(t_1, \ldots, t_n) = \sum_{k \geq 0} \sum_{\substack{d_1+\ldots+d_n=k \\ d_1 \geq 0, \ldots, d_n \geq 0}} t_1^{d_1} t_2^{d_2} \ldots t_n^{d_n}$$

$$= \sum_{d_1 \geq 0, \ldots, d_n \geq 0} t_1^{d_1} t_2^{d_2} \ldots t_n^{d_n}$$

$$= \Big(\sum_{d_1 \geq 0} t_1^{d_1} \Big) \Big(\sum_{d_2 \geq 0} t_2^{d_2} \Big) \ldots \Big(\sum_{d_n \geq 0} t_n^{d_n} \Big)$$

D'où :

$$CRE(t_1, \ldots, t_n) = \frac{1}{1-t_1} \cdot \frac{1}{1-t_2} \cdot \ldots \cdot \frac{1}{1-t_n}$$

On a donc : $CRD(t) = \dfrac{1}{(1-t)^n}$.

Or, on a vu précédemment que $\dfrac{1}{(1-t)^n} = \sum_{k \geq 0} \binom{n+k-1}{k} \cdot t^k$.

Le nombre de combinaisons avec répétitions de k éléments parmi n est le coefficient de t^k dans cette série. C'est donc le nombre binômial $\binom{n+k-1}{k}$.

Remarque : Là encore, on aurait pu trouver $CRE(t_1, \ldots, t_n)$ autrement.

Pour les combinaisons avec répétitions, on considère que l'on a n éléments $e_1, e_2, \ldots, e_n$, et que l'on peut choisir plusieurs fois le même élément. Puisque chaque élément peut être choisi 0 fois, ou 1 fois, ou 2 fois ... , la série d'énumération pour les combinaisons avec répétitions est donc :

$$(1 + t_i + t_i^2 + t_i^3 + \ldots) = \frac{1}{1-t_i}$$

De plus le choix pour un élément est totalement indépendant du choix pour les autres éléments, et ceci se traduit au niveau des séries génératrices par un produit. L'ensemble de toutes les combinaisons possibles avec répétitions de n éléments est donc représenté par la série :

$$CRE(t_1, \ldots, t_n) = \frac{1}{1-t_1} \cdot \frac{1}{1-t_2} \cdot \ldots \cdot \frac{1}{1-t_n}$$

Exemple 3 : Arrangements

Soit $AE(t_1, \ldots, t_n)$ la série d'énumération des arrangements de k éléments parmi n éléments. Le nombre $nb_{d_1, \ldots, d_n}$ est nul si l'un des d_i est différent de 0 ou de 1, et est

égal à $k!$ sinon. En effet, au monôme $t_1^{d_1} t_2^{d_2} \ldots t_n^{d_n}$ correspondent $k!$ configurations, car l'ordre est ici important.

D'où :

$$AE(t_1, \ldots, t_n) = \sum_{k \geq 0} \sum_{\substack{d_1 + \ldots + d_n = k \\ d_1, \ldots, d_n \in \{0,1\}}} k! \cdot t_1^{d_1} t_2^{d_2} \ldots t_n^{d_n},$$

D'où :

$$AD(t) = \sum_{k \geq 0} \sum_{\substack{d_1 + \ldots + d_n = k \\ d_1, \ldots, d_n \in \{0,1\}}} k! \cdot t^k$$

Donc :

$$AD(t) = \sum_{k \geq 0} k! \cdot t^k \cdot \sum_{\substack{d_1 + \ldots + d_n = k \\ d_1, \ldots, d_n \in \{0,1\}}} 1$$

Or, on a vu dans l'exemple 1 que $\displaystyle\sum_{\substack{d_1 + \ldots + d_n = k \\ d_1, \ldots, d_n \in \{0,1\}}} 1 = \binom{n}{k}$. D'où :

$$AD(t) = \sum_{k \geq 0} k! \cdot \binom{n}{k} \cdot t^k.$$

Le nombre d'arrangements de k éléments parmi n est le coefficient de t^k dans $AD(t)$. On retrouve bien :

$$\mathrm{A}_n^k = \frac{n!}{(n-k)!}$$

Exemple 4 : Arrangements avec répétitions

Soit $ARE(t_1, \ldots, t_n)$ la série d'énumération des arrangements avec répétitions de k éléments parmi n éléments. Le nombre $nb_{d_1, \ldots, d_n}$ est le nombre de k-uplets distincts avec d_1 fois l'élément $e_1, \ldots, d_n$ fois l'élément e_n.

Ce nombre vaut : $\dfrac{k!}{d_1! \cdot \ldots \cdot d_n!}$ (c'est le coefficient multinômial introduit en exercice).

On a donc :

$$ARE(t_1, \ldots, t_n) = \sum_{k \geq 0} \sum_{\substack{d_1 + \ldots + d_n = k \\ d_1 \geq 0, \ldots, d_n \geq 0}} \frac{k!}{d_1! \cdot \ldots \cdot d_n!} t_1^{d_1} t_2^{d_2} \ldots t_n^{d_n},$$

d'où :

$$ARE(t_1, \ldots, t_n) = \sum_{k \geq 0} (t_1 + \ldots + t_n)^k.$$

Par suite :

$$ARD(t) = \sum_{k \geq 0} (nt)^k.$$

Le nombre d'arrangements avec répétitions de k éléments parmi n éléments est donc le coefficient de t^k dans $ARD(t)$, c'est-à-dire, n^k.

2.3.3. Partages d'entiers

Un autre problème important en combinatoire est le suivant : étant donnés les entiers $a_1, \ldots, a_p$ et n, trouver le nombre de p-uplets d'entiers $(r_1, \ldots, r_p)$ solutions de l'équation $a_1 r_1 + a_2 r_2 + \ldots + a_p r_p = n$.

Ce problème peut être généralisé en imposant plus ou moins de restrictions aux éléments des solutions (nombre, taille, répétition) ; par exemple, dans le cas où l'on ne précise pas la taille du p-uplet solution et où tous les a_i sont égaux à 1 , cela revient à chercher le nombre de partages de l'entier n en somme d'entiers; il y a, par exemple, cinq partages de l'entier 4 : 4, $3+1$, $2+2$, $2+1+1$ et $1+1+1+1$.

On étudie ici un cas particulier simple : il s'agit de calculer le nombre Q de façons de faire la monnaie de 100 F avec des pièces de 2 F et 3 F. Q est le nombre de couples d'entiers (r, s) solutions de l'équation $2r + 3s = 100$.

Soit $g(x) = \sum_{r, s \in \mathbb{N}} x^{2r+3s}$; dans $g(x)$, chaque puissance $n^{ième}$ de x apparaît autant de fois qu'il existe de couples (r, s) tels que $2r + 3s = n$; Q est donc le coefficient de x^{100} dans $g(x)$.

Or $g(x) = \sum_{r, s \in \mathbb{N}} x^{2r} \cdot x^{3s} = \sum_{r \in \mathbb{N}} x^{2r} \cdot \sum_{s \in \mathbb{N}} x^{3s}$. La série $g(x)$ est donc le produit des séries $a(x) = \sum_{r \in \mathbb{N}} x^{2r} = \sum_{n \geq 0} a_n x^n$ et $b(x) = \sum_{s \in \mathbb{N}} x^{3s} = \sum_{n \geq 0} b_n x^n$. Le coefficient de x^n dans $g(x)$ est donc $g_n = \sum_{p=0}^{n} a_p b_{n-p}$, avec :

$a_n = 1$ si n est pair
$= 0$ sinon
et $b_n = 1$ si n est un multiple de 3
$= 0$ sinon

Ainsi, $g_{100} = \sum_{p=0}^{100} a_p b_{100-p} = \sum_{i=0}^{50} b_{100-2i}$, et $b_{100-2i} = 1$ si, et seulement si, $100 - 2i$ est un multiple de 3.

Or il y a 17 multiples de 6 compris entre 0 et 100, d'où : $g_{100} = 17$.

Il y a donc 17 façons de faire la monnaie de 100 F avec des pièces de 2 F et de 3 F.

2.4. Application à l'analyse d'algorithmes

Les séries génératrices sont aussi très utiles en analyse d'algorithmes, en particulier pour déterminer la complexité en moyenne telle qu'elle a été définie au chapitre 2.

Considérons un algorithme A qui opère sur des données d'un ensemble D ; notons D_n l'ensemble des données de taille n, et supposons que toutes les données de taille n sont équiprobables.

Notons $coût_A(d)$ (ou plus brièvement $c(d)$) la complexité en temps de l'algorithme A sur la donnée d. Alors la complexité en moyenne de A sur les données de taille n est :

$$\text{Moy}_A(n) = \frac{1}{||D_n||} \cdot \sum_{d \in D_n} c(d) \tag{12}$$

2.4.1. Coût moyen

Soit $c(z) = \sum_{d \in D} c(d) z^{|d|}$ la série génératrice du coût de l'algorithme A sur les données de D ; alors $c(z) = \sum_{n \geq 0} \gamma(n) z^n$, où $\gamma(n) = \sum_{d \in D_n} c(d)$.

(γ_n est la somme des coûts de l'algorithme A sur toutes les données de taille n.)

Si d'autre part on introduit la série génératrice de dénombrement de D :

$$d(z) = \sum_{n \geq 0} ||D_n||\ z^n,$$

alors d'après (12), la complexité moyenne est :

$$\text{Moy}_A(n) = \frac{[z^n]c(z)}{[z^n]d(z)} \tag{12bis}$$

Prenons pour exemple de cette approche le calcul de la *hauteur moyenne de pile* lors du parcours itératif d'un arbre binaire ayant n nœuds. Dans le parcours d'un

arbre binaire donné B de taille n, la hauteur moyenne de la pile correspond à la profondeur moyenne des nœuds de B, c'est-à-dire, à $\frac{LC(B)}{n}$, (où $LC(B)$ est la longueur de cheminement de B définie au chapitre 7).

Pour un arbre binaire quelconque de taille n , en supposant que les b_n arbres binaires de taille n sont équiprobables, la hauteur moyenne de pile est donc :

$$hp_n = \frac{1}{n} \cdot \frac{1}{b_n} \cdot \sum_{|B|=n} LC(B) \tag{13}$$

Posons $lc(z) = \sum_{B \in \mathcal{B}} LC(B) \cdot z^{|B|}$;

on a $lc(z) = \sum_{n \geq 0} lc_n \cdot z^n$, avec $lc_n = \sum_{|B|=n} LC(B)$.

Or, la longueur de cheminement des arbres binaires est définie récursivement par : $LC(\emptyset) = 0$, et pour $B =< o, B1, B2 >$: $LC(B) = LC(B1) + LC(B2) + \mid B \mid -1$.

D'où :

$$\begin{aligned} lc(z) &= 0 \cdot z^o + \sum_{B_1, B_2 \in \mathcal{B}} \Big(LC(B1) + LC(B2) + \mid B1 \mid + \mid B2 \mid \Big) \cdot z^{1+|B1|+|B2|} \\ &= \sum_{B_1, B_2 \in \mathcal{B}} LC(B1) \cdot z^{1+|B1|+|B2|} + \sum_{B_1, B_2 \in \mathcal{B}} LC(B2) \cdot z^{1+|B1|+|B2|} \\ &\qquad + \sum_{B \in \mathcal{B} - \emptyset} (\mid B \mid -1) \cdot z^{|B|} \\ &= z \cdot \sum_{B1 \in \mathcal{B}} LC(B1) \cdot z^{|B1|} \cdot \sum_{B2 \in \mathcal{B}} z^{|B2|} + z \cdot \sum_{B_2 \in \mathcal{B}} LC(B2) \cdot z^{|B2|} \cdot \sum_{B1 \in \mathcal{B}} z^{|B1|} \\ &\qquad + \sum_{B \in \mathcal{B} - \emptyset} \mid B \mid \cdot z^{|B|} - \sum_{B \in \mathcal{B} - \emptyset} z^{|B|} \end{aligned}$$

Ainsi, $lc(z) = 2z \cdot lc(z) \cdot b(z) + z \cdot b'(z) - (b(z) - 1)$, où $b(z)$ est la série génératrice de dénombrement des arbres binaires, et donc

$$lc(z) = \frac{zb'(z) - b(z) + 1}{1 - 2z \cdot b(z)}$$

or $b(z) = \frac{1 - \sqrt{1 - 4z}}{2z}$ (voir paragraphe précédent), d'où

$$b'(z) = \frac{1}{4z^2} \Big(\frac{4z}{\sqrt{1 - 4z}} + 2\sqrt{1 - 4z} - 2 \Big)$$

et donc :

$$lc(z) = \frac{1}{1-4z} + \frac{1}{z} - \frac{1}{z\sqrt{1-4z}} + \frac{1}{\sqrt{1-4z}}$$

de plus le développement en séries de $\frac{1}{\sqrt{1-4z}}$ est :

$$\frac{1}{\sqrt{1-4z}} = \sum_{n\geq 0} \binom{2n}{n} z^n,$$

d'où l'on déduit que la somme des longueurs de cheminement de tous les arbres binaires de taille n est :

$$lc_n = [z^n]lc(z) = 4^n - \frac{1+3n}{1+n} \cdot \binom{2n}{n} = 4^n - (1+3n) \cdot b_n \qquad (14)$$

D'après l'expression asymptotique des nombres de Catalan (équation (11) du paragraphe précédent), on obtient à partir de (14) un équivalent asymptotique de la profondeur moyenne d'un nœud dans un arbre binaire quelconque de taille n :

$$hp_n = \frac{1}{n \cdot b_n} \cdot lc_n \sim \sqrt{\pi n}$$

2.4.2. Distribution du coût

La méthode développée au §2.4.1 permet de trouver la valeur moyenne $\text{Moy}_A(n)$ du coût de l'algorithme A sur les données de taille n, mais elle est insuffisante pour une étude plus fine de la distribution du coût, par exemple, pour la détermination de la variance $\text{Var}_A(n)$, qui représente le carré de l'écart moyen de la valeur du coût autour de sa valeur moyenne (voir formulaire) :

$$\begin{aligned}\text{Var}_A(n) &= \frac{1}{||D_n||} \cdot \sum_{d\in D_n} \Big(c(d) - \text{Moy}_A(n)\Big)^2 \\ &= \Big(\frac{1}{||D_n||} \cdot \sum_{d\in D_n} (c(d))^2\Big) - (\text{Moy}_A(n))^2\end{aligned}$$

Pour pouvoir calculer la variance du coût de l'algorithme A sur les données de taille n, on introduit les séries génératrices de coût pour une taille de données fixée :

$$c_n(x) = \sum_{k\geq 0} c_{n,k} \cdot x^k$$

où $c_{n,k}$ est le nombre de données d de taille n telles que $c(d) = k$.

Alors la moyenne définie par la formule (12) peut aussi se mettre sous la forme :

$$\text{Moy}_A(n) = \frac{1}{||D_n||} \sum_{k \geq 0} k \cdot c_{n,k}.$$

D'où $\text{Moy}_A(n) = \sum_{k \geq 0} k \cdot p_{n,k}$, où $p_{n,k}$ est la probabilité pour que le coût de A sur une donnée de taille n soit k : $p_{n,k} = \frac{1}{||D_n||} c_{n,k}$.

Considérons donc la série génératrice des $c_{n,k}$ pour n fixé :

$$c_n(x) = \sum_{k \geq 0} c_{n,k} \cdot x^k,$$

alors $c'_n(x) = \sum_{k \geq 0} k \cdot c_{n,k} \cdot x^{k-1}$, donc $c'_n(1) = \sum_{k \geq 0} k \cdot c_{n,k}$.
De plus $c_n(1) = \sum_{k \geq 0} c_{n,k} = ||D_n||$.

On en déduit donc que l'espérance aléatoire du paramètre *coût* est :

$$\text{Moy}_A(n) = \frac{c'_n(1)}{c_n(1)} \tag{15}$$

On peut aussi calculer la variance du paramètre *coût* pour les données de taille n. D'après la définition de la variance on a :

$$\text{Var}_A(n) = \Big(\sum_{k \geq 0} k^2 \cdot p_{n,k} \Big) - (\text{Moy}_A(n))^2$$

Or $c''_n(x) = \sum_{k \geq 0} k \cdot (k-1) \cdot c_{n,k} \cdot x^{k-2}$.

Donc, $c''_n(1) + c'_n(1) = \sum_{k \geq 0} k^2 \cdot c_{n,k}$ et l'on a :

$$\text{Var}_A(n) = \frac{c''_n(1)}{c_n(1)} + \frac{c'_n(1)}{c_n(1)} - \Big(\frac{c'_n(1)}{c_n(1)} \Big)^2 \tag{16}$$

Dans l'exemple du chapitre 3, c'est de la façon qui vient d'être présentée que l'on calcule le nombre moyen d'affectations du maximum dans l'algorithme de recherche du plus grand élément d'un tableau. Le lecteur pourra de plus, en application de la formule (16), calculer la variance de ce nombre d'affectations.

3. Equations de récurrence

On a défini au chapitre 2 différentes mesures de la complexité des algorithmes, faisant intervenir des quantités (nombre d'opérations en moyenne, au pire ...) qui dépendent de la taille n des données. Le nombre de données possibles de taille n intervient souvent dans le calcul de ces mesures. Dans certains cas, ces quantités s'expriment à l'aide de fonctions usuelles de n et peuvent donc être évaluées directement; c'est le cas, par exemple, pour le produit de matrices vu au chapitre 2. Mais le plus souvent, pour évaluer une quantité $T(n)$, relative à un problème sur des données de taille n, on décompose le problème en sous-problèmes sur des données de tailles plus petites, et on exprime $T(n)$ en fonction de divers $T(p)$ avec $p < n$:

$$T(n) = f(\{T(p);\ p < n\})$$

On obtient alors une équation de récurrence, et pour que cette équation détermine une fonction, il faut encore évaluer directement T pour certaines valeurs de n petites.

Selon la manière dont on décompose le problème en sous-problèmes, on obtient diverses classes d'équations de récurrence; on peut faire *deux classifications orthogonales* des équations de récurrence :

a) Suivant le type de la fonction f : f peut être une *combinaison linéaire* des $T(p)$, à coefficients *constants* ou *variables*; ou encore un polynôme en $T(p)$, etc.

b) Suivant l'ensemble des p qui interviennent pour calculer $T(n)$:

• la valeur de T en n ne dépend que de sa valeur en un seul p inférieur à n; si $T(n)$ ne dépend que de $T(n-1)$, on dit alors que *l'équation est d'ordre 1*.

• il existe un nombre k fixé tel que la valeur de T en n ne dépend que de ses valeurs en exactement k rangs inférieurs à n; en particulier, si $T(n)$ ne dépend que de $T(n-1)\ldots T(n-k)$, on dit que *l'équation est d'ordre k*.

• on ne peut pas fixer, indépendamment de n, le nombre de $T(p)$ nécessaires pour déterminer $T(n)$; c'est en particulier le cas si la valeur de T en n dépend des valeurs de T pour *tous* les p inférieurs à n, on dit alors que l'*équation est complète.*

Pour l'analyse des algorithmes présentés dans ce livre, on rencontre essentiellement trois types d'équations de récurrence.

Type 1 : les récurrences linéaires d'ordre k

Ce sont des équations du type :

$$T(n) = f(n,\ T(n-1), \ldots,\ T(n-k)) + g(n),$$

où $k \geq 1$ est un entier fixé, f est une combinaison linéaire des $T(i)$ pour i tel que $n-k \leq i \leq n-1$, et g est une fonction quelconque de n.

Exemple 1

$$a_n = 2 \cdot a_{n-1} + 1 \qquad (n \geq 1)$$
$$a_0 = 1$$

a_n désigne le nombre de noeuds d'un arbre binaire complet (voir chapitre 7) de hauteur n. On a considéré ici qu'un arbre binaire complet de hauteur n se compose d'une racine et de deux sous-arbres binaires complets de hauteur $n-1$. On a ramené le problème sur des données de taille n à un problème sur des données de taille immédiatement inférieure (ici $k = 1$).

Type 2 : les récurrences de partitions

Ce sont des équations du type :

$$T(n) = a \cdot T(\frac{n}{b}) + d(n),$$

où a et b sont des constantes entières, et $d(n)$ une fonction quelconque de n. Notons que l'équation précédente n'est une équation de récurrence que si n est une puissance de b.

On résoud les équations de ce type en les transformant pour obtenir des récurrences linéaires d'ordre 1. On obtient souvent des équations de ce type dans l'analyse d'algorithmes récursifs travaillant par partitions.

Exemple 2

$$T(n) = 2 \cdot T(\frac{n}{2}) + n - 1 \qquad (n \geq 2)$$
$$T(1) = 0$$

$T(n)$ désigne le nombre de comparaisons effectuées par l'algorithme de tri fusion. Cet algorithme de tri récursif consiste à séparer une liste de taille n en deux sous-listes de taille $n/2$, à trier chaque moitié et à fusionner, en $n-1$ comparaisons, les deux sous-listes obtenues. On a décomposé ici le problème de taille n en deux sous-problèmes de même taille $n/2$.

Type 3 : les récurrences complètes, linéaires ou polynômiales

Ce sont des équations du type :

$$T(n) = f(n,\ T(n-1),\ T(n-2), \ldots,\ T(0)) + g(n),$$

où f est une fonction linéaire, ou polynômiale des $T(i)$, et $g(n)$ une fonction quelconque de n.

Exemple 3 : Récurrence complète polynômiale

$$b_n = \sum_{k=0}^{n-1} b_k \cdot b_{n-1-k} \qquad (n \geq 1)$$
$$b_0 = 1$$

b_n désigne le nombre d'arbres binaires ayant n noeuds (cf. chapitre 7). L'évaluation de b_n nécessite celle de tous les b_i pour i tel que $0 \leq i \leq n-1$.

Exemple 4 : Récurrence complète linéaire

$$c_n = n + 1 + \frac{2}{n} \cdot \sum_{k=1}^{n-1} c_k \qquad (n \geq 2)$$
$$c_0 = c_1 = 0$$

Cette équation définit la complexité moyenne de l'algorithme de tri rapide (cf. chapitre 15).

La suite de ce paragraphe est consacrée à l'exposé de méthodes de résolution pour ces divers types d'équations de récurrence. On traite d'abord les récurrences linéaires d'ordre k (du type 1); puis les équations se ramenant à des récurrences linéaires et en particulier les récurrences du type 2; et enfin, les récurrences complètes du type 3.

3.1. Equations linéaires

3.1.1. Cas simples

Certaines équations de récurrence linéaires peuvent être résolues simplement par une méthode directe. Donnons quelques exemples.

Exemple 5 :

$$a_n = a_{n-1} + 2^n \qquad (n \geq 1)$$
$$a_0 = 1,$$

où a_n désigne, comme dans l'exemple 1, le nombre de nœuds d'un arbre binaire complet de hauteur n. On a considéré ici que a_n est la somme du nombre de nœuds internes (nombre de nœuds d'un arbre binaire complet de hauteur $n-1$) et du nombre de feuilles d'un arbre binaire complet de hauteur n.

Cette équation se résoud en récrivant la relation donnée à l'ordre n, à l'ordre $n-1$, $n-2, \ldots,$ 1, en sommant les égalités obtenues et en simplifiant les deux membres :

$$\begin{aligned} a_n &= a_{n-1} + 2^n \\ a_{n-1} &= a_{n-2} + 2^{n-1} \\ a_{n-2} &= a_{n-3} + 2^{n-2} \\ &\dots \\ a_2 &= a_1 + 2^2 \\ a_1 &= a_0 + 2^1 \end{aligned}$$

On obtient :

$$a_n = a_0 + \sum_{i=1}^{n} 2^i \qquad (n \geq 1),$$

d'où $$a_n = 1 + \frac{2 - 2^{n+1}}{1 - 2}.$$

Ce qui donne $a_n = 2^{n+1} - 1$ pour $n \geq 0$, car la relation est vraie pour $n = 0$.

Exemple 6 : Reprenons l'équation de l'exemple 1

$$\begin{aligned} a_n &= 2 \cdot a_{n-1} + 1 \qquad (n \geq 1) \\ a_0 &= 1 \end{aligned}$$

On récrit cette relation à l'ordre n, $n-1$, $n-2$, ..., 1 :

$$\begin{aligned} a_n &= 2 \cdot a_{n-1} + 1 && (2^0) \\ a_{n-1} &= 2 \cdot a_{n-2} + 1 && (2^1) \\ a_{n-2} &= 2 \cdot a_{n-3} + 1 && (2^2) \\ &\dots \\ a_2 &= 2 \cdot a_1 + 1 && (2^{n-2}) \\ a_1 &= 2 \cdot a_0 + 1 && (2^{n-1}) \end{aligned}$$

Si l'on veut pouvoir sommer et simplifier les deux membres, il faut auparavant multiplier chaque égalité par un facteur différent, comme c'est indiqué entre parenthèses sur chaque ligne. En sommant et simplifiant, on obtient alors :

$$a_n = 2^n \cdot a_0 + 1 + 2 + 2^2 + \dots + 2^{n-2} + 2^{n-1}.$$

Or, $$\sum_{i=0}^{n-1} 2^i = \frac{1 - 2^n}{1 - 2} = 2^n - 1, \qquad \text{et } a_0 = 1.$$

On retrouve le résultat précédent : $a_n = 2^{n+1} - 1$ pour $n \geq 0$.

La méthode qui vient d'être présentée sur l'exemple 6 est appelée ***méthode des facteurs sommants.*** Elle se généralise aux équations de la forme :

$$a(n) \cdot u_n = b(n) \cdot u_{n-1} + c(n), \tag{1}$$

où $a(n)$, $b(n)$, $c(n)$ sont des fonctions de n, et u_0 est une constante.

Introduisons la fonction :

$$f(n) = \frac{\prod_{i=1}^{n-1} a(i)}{\prod_{i=1}^{n} b(i)}$$

et multiplions les deux membres de l'équation (1) par $f(n)$. On obtient :

$$\frac{\prod_{i=1}^{n} a(i)}{\prod_{i=1}^{n} b(i)} u_n = \frac{\prod_{i=1}^{n-1} a(i)}{\prod_{i=1}^{n-1} b(i)} u_{n-1} + f(n)c(n)$$

Posons alors $v_n = a(n) \cdot f(n) \cdot u_n$.

On obtient une équation pour v_n : $v_n = v_{n-1} + f(n) \cdot c(n)$, qui se résoud directement comme on a vu précédemment. La solution de l'équation (1) s'écrit finalement :

$$u_n = \frac{1}{a(n) \cdot f(n)} \left(a(0) \cdot f(0) \cdot u_0 + \sum_{i=1}^{n} f(i) \cdot c(i) \right)$$

Remarque : Une variante de la méthode des facteurs sommants est la ***méthode par induction.*** Considérons à nouveau l'exemple 6 :

$$a_n = 2 \cdot a_{n-1} + 1$$

Récrivons cette équation à l'ordre $n-1$ et remplaçons la valeur de a_{n-1} dans la 1^e relation; on obtient :

$$a_n = 2 \cdot (2 \cdot a_{n-2} + 1) + 1,$$

soit $a_n = 2^2 \cdot a_{n-2} + (2^0 + 2^1)$.

Recommençons pour a_{n-2}, il en résulte :

$$a_n = 2^2 \cdot (2 \cdot a_{n-3} + 1) + (2^0 + 2^1),$$

soit $a_n = 2^3 \cdot a_{n-3} + (2^0 + 2^1 + 2^2)$.

A partir de là, on peut penser que la forme générale de a_n est $a_n = 2^i a_{n-i} + \sum_{j=0}^{i-1} 2^j$, et le vérifier par induction; a_n est alors obtenu pour $i = n$.

3.1.2. Equations linéaires à coefficients constants

A. Méthode de l'équation caractéristique

Une équation linéaire d'ordre k à coefficients constants est une équation de la forme :

$$u_n + a_1 \cdot u_{n-1} + a_2 \cdot u_{n-2} + \ldots + a_k \cdot u_{n-k} = b(n), \tag{2}$$

où $a_1, \ldots, a_k$ sont des constantes.

La résolution d'une telle équation passe par celle de son ***équation homogène associée*** (on dit aussi *sans second membre*) :

$$u_n + a_1 \cdot u_{n-1} + a_2 \cdot u_{n-2} + \ldots + a_k \cdot u_{n-k} = 0 \tag{3}$$

1) *Résolution de l'équation sans second membre*

Le principe de la méthode de résolution repose sur le fait que l'on sait (cf. cours de premier cycle) que l'ensemble des solutions de l'équation (3) forme un espace vectoriel de dimension inférieure ou égale à k, et que certaines solutions sont de la forme $u_n = r^n$. Remplaçons u^n par r^n, on obtient l'***équation caractéristique*** de (3) :

$$r^k + a_1 \cdot r^{k-1} + \ldots + a_k = 0 \tag{4}$$

• Si l'équation caractéristique a k racines distinctes $r_1, \ldots, r_k$, alors la solution générale de (3) est :

$$u_n = \lambda_1 r_1^n + \lambda_2 r_2^n + \ldots + \lambda_k r_k^n,$$

où $\lambda_1, \ldots, \lambda_k$ sont des constantes déterminées par les valeurs initiales $u_0, u_1, \ldots, u_{k-1}$.

• Si les racines ne sont pas toutes distinctes, soit $r_1, \ldots, r_s$ ces racines et soit $\omega_1, \ldots, \omega_s$ leurs ordres respectifs ($\omega_1 + \ldots + \omega_s = k$), alors la solution de (3) est :

$$u_n = \sum_{i=1}^{s} p_i(n) \cdot r_i^n,$$

où p_i est un polynôme en n de degré inférieur ou égal à $\omega_i - 1$.

2) *Résolution de l'équation avec second membre*

La solution de l'équation initiale (2) s'obtient en ajoutant à la solution de (3) une solution particulière de (2). La détermination de la solution particulière est en général difficile, sauf pour certaines formes du second membre $b(n)$.

• Si $b(n) = P(n)$, polynôme en n, il faut chercher une solution particulière sous la forme d'un polynôme $Q(n)$ du même degré que P (sauf si 1 est une racine d'ordre p de l'équation caractéristique, auquel cas on cherche une solution particulière de la forme $n^p \cdot Q(n)$).

• Si $b(n) = \alpha^n$, il faut chercher une solution particulière de la forme $c \cdot \alpha^n$, où c est une constante (sauf si α est une racine d'ordre p de l'équation caractéristique, auquel cas on cherche une solution particulière de la forme $c \cdot n^p \cdot \alpha^n$).

• Plus généralement si $b(n)$ est une somme de termes de la forme $\alpha^n \cdot P(n)$, on cherche une solution particulière sous la forme d'une somme de termes $\alpha^n \cdot Q(n)$.

La recherche de la solution particulière se fait par la ***méthode des coefficients indéterminés*** : dans l'équation (3), on remplace u_n par une solution particulière contenant des coefficients inconnus, et l'on détermine ces coefficients par identification. Par exemple, pour trouver une solution particulière de l'équation :

$$u_{n+3} - u_{n+2} + u_{n+1} - u_n = 2^n \cdot n^2 \tag{5}$$

on cherche une solution de la forme $u_n = 2^n(A \cdot n^2 + B \cdot n + C)$ car 2 n'est pas racine de l'équation caractéristique; en remplaçant u_n par cette valeur dans l'équation (5), on obtient :

$$5An^2 + (36A + 5B)n + (58A + 18B + 5C) = n^2 \tag{6}$$

Par identification des coefficients des deux termes de l'équation (6), on obtient $A = \frac{1}{5}$, $B = -\frac{36}{25}$, $C = \frac{328}{125}$; donc une solution particulière de (5) est :

$$u_n = \frac{2^n(25n^2 - 180n + 358)}{125}.$$

Donnons quelques exemples de résolution d'équations linéaires par la méthode de l'équation caractéristique.

Exemple 7

$$F_n = F_{n-1} + F_{n-2} \qquad (n \geq 2)$$
$$F_0 = 0,\ F_1 = 1$$

Cette équation définit les nombres de Fibonacci qui interviennent, entre autres, dans l'analyse du tri polyphasé (cf. chapitre 17). L'équation caractéristique s'écrit :

$$r^2 - r - 1 = 0$$

Elle admet deux racines simples :

$$\phi = \frac{1+\sqrt{5}}{2} \quad \text{et} \quad \bar{\phi} = \frac{1-\sqrt{5}}{2}$$

La solution est donc de la forme : $F_n = \lambda_1 \phi^n + \lambda_2 \bar{\phi}^n$. Les conditions initiales déterminent λ_1 et λ_2 :

$$F_0 = 0 \Rightarrow \lambda_1 + \lambda_2 = 0 \quad \text{et} \quad F_1 = 1 \Rightarrow \lambda_1 \phi + \lambda_2 \bar{\phi} = 1$$

d'où $\lambda_1 = -\lambda_2 = -\frac{1}{\sqrt{5}}$, et $F_n = \frac{1}{\sqrt{5}} (\phi^n - \bar{\phi}^n) \quad (n \geq 0)$.

Exemple 8

$$u_n - 4 \cdot u_{n-1} + 4 \cdot u_{n-2} = 2n \quad (n \geq 2)$$
$$u_0 = 0 \quad u_1 = 1$$

L'équation homogène associée a pour équation caractéristique $r^2 - 4r + 4 = 0$. Elle admet une racine double $r_1 = 2$. La solution de l'équation homogène est donc :

$$u_n = p_1(n) \cdot r_1^n, \text{ où } p_1(n) = a \cdot n + b,$$

c'est-à-dire, $u_n = 2^n \cdot (a \cdot n + b)$.

Une solution particulière est à chercher sous la forme $\alpha n + \beta$, étant donné la forme du second membre.

En remplaçant u_n, u_{n-1} et u_{n-2} par leurs valeurs dans l'équation initiale, on obtient $\alpha = 2$ et $\beta = 8$, par identification des coefficients des deux polynômes.

La solution générale de l'équation est alors : $u_n = (a \cdot n + b) \cdot 2^n + 2n + 8$ où a et b sont déterminés par les valeurs de u_0 et de u_1.

Finalement on obtient $u_n = \left(\frac{7}{2} \cdot n - 8\right) \cdot 2^n + 2n + 8$.

B. Utilisation de séries génératrices

Une autre méthode de résolution des équations linéaires à coefficients constants consiste à traduire une équation de récurrence par une équation fonctionnelle sur la série génératrice associée.

En effet, si la suite u_n vérifie :

$$u_n = a_1 \cdot u_{n-1} + a_2 \cdot u_{n-2} + \ldots + a_k \cdot u_{n-k},$$

alors la série génératrice $u(z) = \sum_{n \geq 0} u_n \cdot z^n$ vérifie :

$$u(z) = p(z) + u(z)(a_1 \cdot z + a_2 \cdot z^2 + \ldots + a_k \cdot z^k),$$

où $p(z)$ est un polynôme en z de degré $k-1$ dont les coefficients dépendent des a_i et des valeurs initiales $u_0,\ u_1, \ldots,\ u_{k-1}$. D'où :

$$u(z) = \frac{p(z)}{1 - a_1 \cdot z - a_2 \cdot z^2 - \ldots - a_k \cdot z^k}.$$

Reprenons l'exemple 7 :

$$F_n = F_{n-1} + F_{n-2} \qquad (n \geq 2)$$
$$F_0 = 0,\ F_1 = 1$$

Soit $F(z) = \sum_{n \geq 0} F_n \cdot z^n$. Pour $n \geq 2$ on a : $z^n \cdot F_n = z^n \cdot F_{n-1} + z^n \cdot F_{n-2}$.

On somme pour $n \geq 2$ les deux membres de l'égalité, d'où :

$$F(z) - (F_0 + F_1 \cdot z) = z \cdot (F(z) - F_0) + z^2 \cdot F(z),$$

et donc $F(z) = \dfrac{z}{1 - z - z^2}$.

Une décomposition en éléments simples donne :

$$F(z) = \frac{1}{\sqrt{5}}\left(\frac{1}{1 - \phi z} - \frac{1}{1 - \bar{\phi} z}\right)$$

Par identification des coefficients de la série génératrice, on retrouve donc :

$$F_n = \frac{1}{\sqrt{5}}\left(\phi^n - \bar{\phi}^n\right), \text{ pour } n \geq 0.$$

3.1.3. Equations linéaires à coefficients variables

Lorsque les coefficients sont variables, il n'est pas garanti de trouver une solution pour les équations linéaires. Cependant, on propose deux méthodes qui peuvent convenir dans certains cas.

A. Facteurs sommants

C'est la méthode décrite en 1) pour les équations de la forme :

$$a_n \cdot u_n = b_n \cdot u_{n-1} + c_n$$

Elle s'applique aux équations linéaires à coefficients variables du premier ordre.

B. Séries génératrices

Comme dans le cas où les coefficients sont constants, on peut recourir aux séries génératrices, si les coefficients sont polynômiaux. On obtiendra alors une équation

différentielle faisant intervenir les dérivées d'ordre 1, 2 ou plus, de la série génératrice. De telles équations sont souvent difficiles à résoudre. On donne un exemple où l'équation différentielle est simple.

Exemple 9

$$n \cdot u_n + (n-2) \cdot u_{n-1} - u_{n-2} = 0$$
$$u_0 = 1,\ u_1 = 1$$

Multiplions la relation de récurrence par z^n et sommons pour $n \geq 2$; on obtient :

$$\sum_{n\geq 2} n \cdot u_n \cdot z^n + \sum_{n\geq 2}((n-1)-1) \cdot u_{n-1} \cdot z^n - \sum_{n\geq 2} u_{n-2} \cdot z^n = 0$$

Posons $G(z) = \sum_{n\geq 0} u_n \cdot z^n$; donc $G'(z) = \sum_{n\geq 1} n \cdot u_n \cdot z^{n-1}$.

La relation s'écrit alors :

$$(z \cdot G'(z) - z \cdot u_1) + z^2 \cdot G'(z) - z \cdot G(z) + z \cdot u_0 - z^2 \cdot G(z) = 0$$

Remplaçons u_0 par 1, u_1 par 1 et divisons par z ; on obtient l'équation différentielle : $G'(z) \cdot (1+z) = G(z) \cdot (1+z)$, soit $G'(z) = G(z)$. Cette équation est classique et sa solution est :

$$G(z) = \lambda \cdot e^z.$$

Or $G(0) = u_0 = 1 \Rightarrow \lambda = 1$. D'où

$$G(z) = \sum_{n\geq 0} \frac{1}{n!} \cdot z^n \text{ et } u_n = \frac{1}{n!} \quad (n \geq 0)$$

3.2. Equations se ramenant à des récurrences linéaires

3.2.1. Changement de variable

Jusqu'à présent on n'a manipulé que des équations du type 1. On s'intéresse maintenant aux équations du type 2, c'est-à-dire, de la forme :

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + d(n) \qquad (n \geq 2),\ \text{avec } a \text{ et } b \text{ constantes}$$
$$T(1) = 1$$

$T(n)$ représente le coût pour obtenir la solution d'un problème de taille n : on a décomposé le problème en a sous-problèmes de taille n/b; la création des sous-problèmes et l'obtention de la solution du problème de taille n à partir des solutions des sous-problèmes coûtent $d(n)$.

La méthode qui suit repose sur l'hypothèse que n *est une puissance de* b : $n = b^k$. Remplaçons n par b^k dans l'équation initiale, on obtient : $T(b^k) = a \cdot T(b^{k-1}) + d(b^k)$.

Posons $t_k = T(b^k)$. On obtient alors une équation linéaire $t_k = a.\ t_{k-1} + d(b^k)$, avec $t_0 = 1$, qui peut se résoudre comme précédemment :

$$t_k = a^k + \sum_{j=0}^{k-1} a^j \cdot d\left(b^{k-j}\right).$$

Or, $k = \log_b n$; comme $a^{\log_b n} = n^{\log_b a}$, la solution s'écrit donc pour n puissance de b :

$$T(n) = n^{\log_b a} + \sum_{j=0}^{(\log_b n)-1} a^j \cdot d\left(\frac{n}{b^j}\right) \qquad (7)$$

Reprenons l'exemple 2 :

$$T(n) = 2 \cdot T(\frac{n}{2}) + n - 1 \qquad (n \geq 2)$$
$$T(1) = 1$$

Ici $a = b = 2$ et $d(n) = n - 1$, donc d'après l'équation (7) :

$$T(n) = n + \sum_{j=0}^{k-1} 2^j \left(\frac{n}{2^j} - 1\right)$$
$$= n + kn - (2^k - 1)$$

d'où, comme $k = \log_2 n$, on a pour n puissance de 2 :

$$T(n) = n\log_2 n + 1.$$

Dans l'analyse de la complexité des algorithmes, c'est souvent l'*ordre de grandeur* de la fonction de coût qui nous intéresse.

On étudie ici l'ordre de grandeur de $T(n)$ dans le cas où la fonction $d(n)$ vérifie la propriété : $d(x \cdot y) = d(x) \cdot d(y)$; on dit alors que $d(n)$ est une fonction ***multiplicative***. Par exemple, la fonction $d(n) = n^2$ est multiplicative, mais la fonction $d(n) = n - 1$ ne l'est pas.

Dans le cas où d est multiplicative, on va montrer que selon la valeur de a par rapport à celle de $d(b)$, c'est le premier ou le deuxième terme de $T(n)$ qui est prépondérant.

Si l'on suppose d multiplicative, on a :

$$\sum_{j=0}^{k-1} a^j d(b^{k-j}) = d(b)^k \sum_{j=0}^{k-1} \left(\frac{a}{d(b)}\right)^j$$

Si $a = d(b)$, le second terme de $T(n)$ vaut :

$$k \cdot d(b)^k = \log_b n \cdot n^{\log_b d(b)}$$

Si $a \neq d(b)$, le second terme de $T(n)$ s'écrit :

$$d(b)^k \cdot \frac{1 - \left(\frac{a}{d(b)}\right)^k}{1 - \frac{a}{d(b)}} = \frac{n^{\log_b a} - n^{\log_b d(b)}}{\frac{a}{d(b)} - 1}$$

En fait, en supposant T croissante au sens large à partir d'un certain rang, on peut évaluer l'ordre de grandeur de T, à partir des valeurs de $T(n)$ pour n puissance de b (voir exercice n° 58).

On obtient alors pour $T(n)$ les ordres de grandeur suivants :

- si $a > d(b)$ alors $T(n) = \Theta(n^{\log_b a})$;
- si $a < d(b)$ alors $T(n) = \Theta(n^{\log_b d(b)})$,
  et dans le cas où $d(n) = n^\alpha$, on a $T(n) = \Theta(n^\alpha) = \Theta(d(n))$;
- si $a = d(b)$ alors $T(n) = \Theta(n^{\log_b a} \cdot \log_b n)$,
  et dans le cas où $d(n) = n^\alpha$, on a $T(n) = \Theta(n^\alpha \cdot \log_b n)$.

Remarque : Dans la solution

$$T(n) = n^{\log_b a} + \sum_{j=0}^{k-1} a^j d\left(\frac{n}{b^j}\right),$$

qui représente le coût d'un algorithme sur un problème de taille n, le premier terme correspond au coût pour résoudre les sous-problèmes, et le deuxième au coût pour créer les sous-problèmes et les ajuster.

Ainsi, si c'est le premier terme qui l'emporte, pour améliorer l'algorithme, il faut jouer sur a et b : diminuer le nombre a de sous-problèmes et donner à chaque sous-problème une taille plus petite (c'est-à-dire, augmenter b). Si c'est le second terme qui l'emporte, pour améliorer l'algorithme, il faut aussi modifier $d(n)$, c'est-à-dire, optimiser la façon de combiner les sous-problèmes.

3.2.2. Transformations du domaine

Certaines équations de récurrence non linéaires se ramènent à des équations linéaires par des transformations simples. On en donne quelques exemples.

Exemple 10

$$a_n = 3 \cdot a_{n-1}^2 \quad (n \geq 1)$$
$$a_0 = 1$$

Posons $b_n = \log_2 a_n$ et prenons le logarithme des deux membres de l'égalité. L'équation devient :

$$b_n = 2 \cdot b_{n-1} + \log_2 3$$
$$b_0 = 0$$

C'est une équation linéaire à coefficients constants, que l'on peut résoudre comme il a été vu précédemment; on obtient :

$$b_n = (2^n - 1)\log_2 3, \quad \text{et donc } a_n = 2^{(2^n-1)\log_2 3} = 3^{2^n-1}$$

Exemple 11

$$u_n = u_{n-1} - u_n \cdot u_{n-1} \quad (n \geq 1)$$
$$u_0 = 1$$

Supposons $u_n \neq 0$ et divisons par $(u_n \cdot u_{n-1})$; l'équation se récrit :

$$\frac{1}{u_{n-1}} = \frac{1}{u_n} - 1$$

Posons alors $v_n = \frac{1}{u_n}$; on obtient une équation simple :

$$v_n = v_{n-1} + 1$$
$$v_0 = 1$$

qui se résoud immédiatement en $v_n = n + 1$, et donc $u_n = \frac{1}{n+1}$.

Exemple 12

$$G_{h+1}(z) = \frac{z}{1 - G_h(z)} \quad (h \geq 0)$$
$$G_0(z) = z$$

$G_h(z) = \sum_{n \geq 0} A_{n,h} \cdot z^n$, où $A_{n,h}$ désigne le nombre d'arbres généraux ayant n nœuds et une hauteur inférieure ou égale à h (cf. exercices).

Cette forme de récurrence se résoud par la transformation suivante : on pose

$$G_h(z) = \frac{z \cdot P_h(z)}{P_{h+1}(z)}$$

où $P_h(z)$ et $P_{h+1}(z)$ sont des polynômes premiers entre eux, ce qui donne l'équation de récurrence linéaire :

$$P_{h+1}(z) = P_h(z) - z \cdot P_{h-1}(z), \quad h \geq 1.$$

Les conditions initiales s'obtiennent en identifiant $(z \cdot \frac{P_0}{P_1})$ à la valeur initiale de G_0 ; sachant que les polynômes P_0 et P_1 sont premiers entre eux, on a donc

$$P_0(z) = P_1(z) = 1.$$

Cette dernière équation se résoud de façon classique, pour obtenir

$$P_h(z) = \frac{1}{\sqrt{1-4z}} \cdot \left[\left(\frac{1+\sqrt{1-4z}}{2} \right)^{h+1} - \left(\frac{1-\sqrt{1-4z}}{2} \right)^{h+1} \right].$$

D'où :

$$G_h(z) = z \cdot \frac{\left(\frac{1+\sqrt{1-4z}}{2} \right)^{h+1} - \left(\frac{1-\sqrt{1-4z}}{2} \right)^{h+1}}{\left(\frac{1+\sqrt{1-4z}}{2} \right)^{h+2} - \left(\frac{1-\sqrt{1-4z}}{2} \right)^{h+2}}$$

Enfin, certaines équations linéaires avec second membre peuvent être ramenées simplement à des équations linéaires homogènes. Donnons un exemple.

Exemple 13

$$u_n = u_{n-1} + u_{n-2} + 1, \ n \geq 2$$
$$u_0 = 1, \ u_1 = 2$$

u_n désigne le nombre de nœuds d'un arbre de Fibonacci de hauteur n (cf. chapitre 11).

L'équation initiale se récrit : $u_n + 1 = (u_{n-1} + 1) + (u_{n-2} + 1)$. Posons $v_n = u_n + 1$; on obtient l'équation linéaire homogène $v_n = v_{n-1} + v_{n-2}$, et de plus $v_0 = 2$, $v_1 = 3$. Le lecteur aura reconnu l'équation de l'exemple 7 : v_n est un nombre de Fibonacci F_p où p est déterminé à l'aide des valeurs initiales.

3.3. Récurrences complètes

On s'intéresse dans ce paragraphe aux équations de récurrence complètes du type 3 : linéaires ou polynômiales.

3.3.1. Utilisation de séries génératrices

On a déjà utilisé des séries génératrices, au paragraphe 2.3., pour calculer le nombre b_n d'arbres binaires ayant n sommets (exemple 3). On traite ici l'exemple 4, qui intervient dans le calcul de la complexité en moyenne du tri rapide (cf. chapitre 15).

$$c_n = n + 1 + \frac{2}{n} \cdot \sum_{k=1}^{n-1} c_k \quad (n \geq 2)$$
$$c_0 = c_1 = 0$$

Soit $c(z) = \sum_{n\geq 0} c_n z^n$; d'après la récurrence on a :

$$\sum_{n\geq 2} c_n z^n = \sum_{n\geq 2} (n+1) z^n + 2 \sum_{n\geq 2} \left(\sum_{k=0}^{n-1} c_k \right) \frac{z^n}{n}$$

or :

$$\sum_{n\geq 2} (n+1) z^n = \frac{1}{(1-z)^2} - (1+2z)$$

et :

$$\sum_{n\geq 2} \left(\sum_{k=0}^{n-1} c_k \right) \cdot \frac{z^n}{n} = \sum_{n\geq 1} \left(\sum_{k=0}^{n} c_k \right) \cdot \frac{z^{n+1}}{n+1} = \int_0^z \sum_{n\geq 0} \left(\sum_{k=0}^{n} c_k \right) \cdot s^n ds.$$

en utilisant le fait que $c_0 = 0$; on reconnaît (cf. §2.1), sous le signe intégral, le produit de Cauchy de la série $c(s)$ par la série $\frac{1}{1-s}$; on obtient donc :

$$c(z) = \frac{1}{(1-z)^2} - (1+2z) + 2 \int_0^z c(s) \, \frac{1}{1-s} \, ds,$$

d'où, en dérivant :

$$c'(z) = \frac{2}{(1-z)^3} - 2 + 2 \, \frac{c(z)}{1-z}. \tag{8}$$

L'équation différentielle (8) se résoud selon la technique habituelle

• Résolution de l'équation sans second membre :

$$\frac{c'(z)}{c(z)} = \frac{2}{1-z}, \text{ d'où } c(z) = \frac{\lambda}{(1-z)^2}$$

• Variation de la constante :

en remplaçant λ par $\lambda(z)$ et, dans l'équation (8), $c(z)$ par la valeur obtenue à l'étape précédente, on obtient :

$$\frac{\lambda'(z)}{(1-z)^2} = \frac{2}{(1-z)^3} - 2,$$

d'où en intégrant :

$$\lambda(z) = 2\text{Log}\frac{1}{1-z} + \frac{2}{3}(1-z)^3 + \text{cste.}$$

Or $\lambda(0) = c(0) = 0$, d'où cste$= -2/3$. Ainsi :

$$c(z) = \frac{2}{(1-z)^2}\text{Log}\Big(\frac{1}{1-z}\Big) + \frac{2}{3}(1-z) - \frac{2}{3}\,\frac{1}{(1-z)^2}$$

or,

$$\frac{1}{(1-z)^2}\text{Log}\Big(\frac{1}{1-z}\Big) = \sum_{n\geq 1}\Big((n+1)H_n - n\Big)z^n,$$

où H_n est le $n^{ième}$ nombre harmonique (voir exercices). Donc le coefficient de z^n dans la série $c(z)$ est $2(n+1)H_n - 2n - \frac{2}{3}(n+1)$. D'où :

$$c_n = 2(n+1)\Big(H_{n+1} - \frac{4}{3}\Big) \quad (n \geq 2).$$

3.3.2. Résolution par soustraction

La stratégie consiste à soustraire des combinaisons convenables de relations de récurrence écrites à des ordres voisins.

On l'illustre en reprenant l'équation de récurrence précédente :

$$c_n = n + 1 + \frac{2}{n}\cdot\sum_{k=1}^{n-1} c_k \qquad (n \geq 2)$$

$$c_0 = c_1 = 0$$

Isolons la sommation finie dans un membre, on obtient :

$$n\big(c_n - (n+1)\big) = 2\sum_{k=1}^{n-1} c_k \quad (n \geq 2).$$

Récrivons cette dernière égalité à l'ordre $n-1$:

$$(n-1)(c_{n-1} - n) = 2\sum_{k=1}^{n-2} c_k \quad (n \geq 3).$$

Retranchons la dernière égalité à l'avant-dernière :

$$n \cdot c_n - (n-1) \cdot c_{n-1} - 2n = 2c_{n-1}, \text{ pour } n \geq 3,$$

soit $n \cdot c_n = (n+1) \cdot c_{n-1} + 2n$.

Divisons les deux membres par $n(n+1)$; on obtient :

$$\frac{1}{n+1} \cdot c_n = \frac{1}{n} \cdot c_{n-1} + \frac{2}{n+1}, \quad \text{pour } n \geq 3$$

et $c_0 = c_1 = 0$, et $c_2 = 3$.

Cette dernière relation de récurrence se résoud simplement par la méthode des facteurs sommants :

$$\begin{aligned}
\frac{1}{n+1} \cdot c_n &= \frac{1}{n} \cdot c_{n-1} + \frac{2}{n+1} \\
\frac{1}{n} \cdot c_{n-1} &= \frac{1}{n-1} \cdot c_{n-2} + \frac{2}{n} \\
&\vdots \qquad\qquad \vdots \\
\frac{1}{4} \cdot c_3 &= \frac{1}{3} \cdot c_2 + \frac{2}{4}
\end{aligned}$$

D'où, en sommant :

$$\frac{1}{n+1} \cdot c_n = \frac{1}{3} \cdot c_2 + 2\sum_{i=4}^{n+1} \frac{1}{i}.$$

Donc :

$$\begin{aligned}
c_n &= 2(n+1)\Big(\sum_{i=4}^{n+1} \frac{1}{i}\Big) + n + 1 \\
&= 2(n+1)\Big(H_{n+1} - 1 - \frac{1}{2} - \frac{1}{3}\Big) + n + 1.
\end{aligned}$$

Et l'on retrouve :

$$c_n = 2(n+1)\Big(H_{n+1} - \frac{4}{3}\Big).$$

Remarque : On s'est intéressé dans ce paragraphe aux équations de récurrence que l'on rencontre en analyse d'algorithmes : on en a donné une typologie, et présenté diverses méthodes de résolution. Cependant, il arrive que la fonction de coût $T(n)$ ne puisse être exprimée que par une *inéquation de récurrence*. On essaie alors de la résoudre à l'aide de techniques analogues à celles qui ont été présentées. On n'obtient pas de valeur exacte pour $T(n)$, mais on peut estimer son ordre de grandeur – en général sous la forme $O(f(n))$ (cf. exercices); si l'on peut borner $T(n)$ dans les deux sens par la même fonction (à un coefficient près), alors on peut estimer son ordre de grandeur sous la forme $\Theta(f(n))$.

Exercices

Asymptotique

1. Etablir les règles $\mathbf{R}_3$, $\mathbf{R}_4$ et $\mathbf{R}_5$ du §1.1.1.

2. Soit f_1 et f_2 deux fonctions telles que $f_1 = \Theta(g)$, $f_2 = \Theta(g)$ et $f_1 \geq f_2$.
A-t-on $f_1 - f_2 = \Theta(g)$?

3. Pour $n > 2$, laquelle des deux quantités n^{2n} ou $(2n)^n$ est la plus grande ?

4. Expliquer pourquoi le raisonnement suivant est faux : pour trouver l'ordre de grandeur asymptotique de $\sigma = \sum_{i=1}^{n} i^2 = 1 + 2^2 + \ldots + n^2$, il suffit d'appliquer «plusieurs fois» la règle $\mathbf{R}_4$ c). Or, max $(1, 2^2, \ldots, n^2) = n^2$, donc $\sigma = \Theta(n^2)$.

5. Comparer deux à deux les comportements asymptotiques des fonctions suivantes :

a) $f_1(n) = n^2$

b) $f_2(n) = n^2 + 100n$

c) $f_3(n) = n$ si n est impair
n^3 si n est pair

d) $f_4(n) = n$ si $n \leq 100$
n^3 si $n > 100$

6. Regrouper en classes d'équivalence pour la relation Θ les 12 fonctions suivantes :

1) $f_1(n) = n$

2) $f_2(n) = 2^n$

3) $f_3(n) = n\log_2 n$

4) $f_4(n) = n - n^3 + 7.n^5$

5) $f_5(n) = n^2 + \log_2 n$

6) $f_6(n) = n^2$

7) $f_7(n) = \log_2 n$

8) $f_8(n) = n^3$

9) $f_9(n) = \sqrt{n} + \log_2 n$

10) $f_{10}(n) = (\log_2 n)^2$

11) $f_{11}(n) = n!$

12) $f_{12}(n) = \mathrm{Log}\, n$

Ordonner les classes d'équivalence obtenues selon la relation O.

7. Calculer la complexité $f(n)$ de la procédure suivante :

```
procedure mystère;
var i, j, k : integer;
begin
    for i := 1 to n − 1 do
        for j := i + 1 to n do
            for k := 1 to j do {instruction}
end mystère;
```

On supposera que *instruction* prend un temps constant c. Préciser la mesure de complexité choisie. En déduire l'ordre de grandeur asymptotique de $f(n)$.

8. Vérifier que les ensembles E_1, E_2, E_3 du §1.2 sont bien des échelles de comparaison.

9. Donner un développement asymptotique d'ordre 3 de $n.\left(a^{\frac{1}{n}} - 1\right)$, pour $a > 0$, selon l'échelle E_3.

10. Prouver l'égalité suivante :

$$(n+a)^{n+b} = n^{n+b}.e^a.\left(1 + a.\left(b - \frac{a}{2}\right).\frac{1}{n} + o\left(\frac{1}{n}\right)\right)$$

11. On a vu au chapitre 7 l'égalité suivante :

$$\sum_{i=1}^{n} \lfloor \log_2 i \rfloor = 2 + (n+1)\lfloor \log_2 n \rfloor - 2^{\lfloor \log_2 n \rfloor + 1}$$

Utiliser ce résultat pour montrer que :

$$n\log_2 n - 3n + \log_2 n + 1 < \sum_{i=1}^{n} \log_2 i < n\log_2 n + \log_2 n + 2$$

En déduire l'encadrement suivant : $16(\frac{n}{8})^{n+1} < n! < 4.n^{n+1}$

On obtient ainsi un meilleur encadrement que celui qui est proposé au §1.3, mais il ne donne pas encore l'ordre de grandeur exact de $n!$: les bornes supérieure et inférieure diffèrent par un facteur de plus de 8^n.

12. En procédant comme au §1.3, montrer qu'on a l'encadrement suivant :

$$\frac{2}{3}.n^{\frac{3}{2}} - n^{\frac{1}{2}} + \frac{1}{3} \leq \sum_{i=1}^{n-1} i^{\frac{1}{2}} < \frac{2}{3}.n^{\frac{3}{2}} - \frac{2}{3}$$

Combinatoire et séries génératrices

13. De combien de façons différentes n enfants peuvent-ils se placer pour former une ronde ?

14. Dans une boutique, il y a k sortes de cartes postales (et autant de cartes postales que l'on veut de chaque sorte). On veut envoyer des cartes postales à n amis.

Quel est le nombre de manières de faire lorsque :

a) chaque ami reçoit une carte quelconque,

b) des amis distincts reçoivent des cartes postales distinctes,

c) chaque ami reçoit deux cartes différentes (mais des amis distincts peuvent très bien recevoir une même carte, voire deux mêmes cartes).

15. Déterminer le nombre d'applications strictement croissantes de $\{1, \ldots, n\}$ dans $\{1, \ldots, m\}$.

16. On dispose de n lettres non toutes distinctes : q_1 lettres a_1, q_2 lettres a_2, ..., q_p lettres a_p, telles que $\sum_{1 \leq i \leq p} q_i = n$. Montrer que le nombre de mots différents de longueur n que l'on peut former avec ces n lettres est $\frac{n!}{q_1!.\ldots.q_p!}$ (*coefficient multinômial*).

30. Prouver par récurrence que $F_{n+1}.F_{n-1} - F_n^2 = (-1)^n$.

31. Montrer que $\sum_{k=0}^{n} F_k.F_{n-k} = \frac{n-1}{5}.F_n + \frac{2n}{5}.F_{n-1}$ $(F_0 = 0,\ F_1 = 1)$.

32. Reprendre l'exemple traité au §2.3.3 et calculer le coefficient de x^{100} dans $g(x)$ en décomposant la série entière en éléments simples.

33. On rappelle (cf. chapitre 7) que dans un arbre binaire localement complet, tout nœud a soit deux fils (nœud interne), soit zéro fils (nœud externe), et que si on note I le nombre de nœuds internes et E le nombre de nœuds externes, on a la relation : $E = I + 1$. On note i_k (resp. e_k) le nombre de nœuds internes (resp. externes) à profondeur k d'un arbre binaire localement complet T. A tout arbre binaire localement complet T, on associe deux séries génératrices

$$I_T(x) = \sum_{k\geq 0} i_k.x^k \text{ et } E_T(x) = \sum_{k\geq 0} e_k.x^k.$$

a) Montrer que $I_T(x).(1-2x) = 1 - E_T(x)$.

b) On note $LCI(T)$ (resp. $LCE(T)$) la longueur de cheminement interne (resp. externe) de T. Exprimer $LCE(T)$ et $LCI(T)$ en fonction des séries génératrices $E_T(x)$ et $I_T(x)$. Retrouver le fait que pour un arbre binaire localement complet tel que $I = n$, on a la relation :

$$LCE(T) = LCI(T) + 2n.$$

34. Soit n et m deux entiers. On note $P_{n,m}$ le nombre de partitions d'un ensemble à n éléments en exactement m classes (non vides).

a) Calculer $P_{4,2}$. Donner tous les couples $(n,\ m)$ pour lesquels $P_{n,m} = 0$. Quelles sont les valeurs de $P_{n,1}$ et $P_{n,n} (n \geq 1)$?

b) Etablir la relation de récurrence : $P_{n,m} = P_{n-1,\ m-1} + m.P_{n-1,\ m}$. Préciser pour quelles valeurs de n et de m elle est valide.

c) Montrer par récurrence sur n que :

$$P_{n,m} = \frac{1}{m!} \sum_{j\leq 0\leq m} (-1)^j \binom{m}{j} (m-j)^n$$

d) Pour $m \geq 1$, on note $B_m(x) = \sum_{n\geq 1} P_{n,m}.x^n$. Calculer $B_1(x)$. Etablir une relation de récurrence entre $B_m(x)$ et $B_{m-1}(x)$ pour $m \geq 2$ et la résoudre.

e) Soit $F_m(x) = \prod_{1 \leq j \leq m} (1 - jx)^{-1}$

Montrer que : $F_m(x) = \sum_{n \geq m} P_{n,m} x^{n-m}$.

f) Etablir l'encadrement suivant : $m^{n-m} \leq P_{n,m} \leq \binom{n-1}{m-1} m^{n-m}$

g) En utilisant la question e) montrer que $P_{n,m} = \sum_{j=m}^{n} P_{j-1,m-1} . m^{n-j}$.

h) En déduire que $P_{n,m} \underset{n \to \infty}{\sim} \frac{1}{m!} m^n$.

35. Etant donnés n facteurs $a_1 \ldots a_n$, on note c_n le nombre de parenthésages possibles dans l'écriture de leur produit. Ainsi $c_3 = 2$ correspond aux deux écritures $((a_1 a_2) a_3)$ et $(a_1 (a_2 a_3))$. On pose $c_1 = 1$.

a) Calculer c_4.

b) En considérant le dernier produit que l'on effectue dans le calcul du produit total, établir la relation de récurrence :

$$c_n = \sum_{i=1}^{n-1} c_i . c_{n-i} \quad \text{pour } n \geq 2$$

(c_n est un ***nombre de Catalan***).

36. Etant donné un polygone convexe à $n+1$ sommets P_n, on cherche à déterminer le nombre de *triangulations* de P_n. (On appelle triangulation d'un polygone convexe, toute division de son intérieur obtenue en ajoutant des diagonales qui ne se coupent pas deux à deux.) On note d_n le nombre de triangulations d'un polygone convexe à $n+1$ sommets. Etablir la relation de récurrence suivante :

$$d_n = \sum_{i=1}^{n-1} d_i . d_{n-i} \quad n \geq 2.$$

Préciser pour quelles valeurs elle est valable et donner les valeurs initiales.

(d_n est égal au nombre c_n de l'exercice précédent, le nombre de parenthésages possibles d'un produit de n facteurs – nombre de Catalan –.)

37. Soit $a_n = \sum_{k=0}^{n} \binom{n+k}{2k}$, et $b_n = \sum_{k=0}^{n-1} \binom{n+k}{2k+1}$, pour $n \geq 1$ et $a_0 = 1$ et $b_0 = 0$.

a) Montrer que a_n et b_n satisfont les relations :

$$a_{n+1} = a_n + b_{n+1}$$
$$b_{n+1} = a_n + b_n$$

b) Trouver les séries génératrices des a_n et b_n.

c) Exprimer a_n et b_n en fonction des nombres de Fibonacci.

38. Les nombres de Bernouilli sont définis par :

$$b_n = \sum_{k=0}^{n} \binom{n}{k} b_{n-k}, \quad \text{pour } n \geq 2 \text{ et } b_0 = 1.$$

a) Calculer b_n pour $0 \leq n \leq 5$.

b) Trouver la série génératrice exponentielle des b_n.

39. a) On note $G_{n,h}$ le nombre d'arbres généraux ayant n nœuds et de hauteur inférieure ou égale à h. Evaluer $G_{0,h}$ et montrer la relation de récurrence suivante :

$$G_{n,h+1} = \sum_{k \geq 1} \sum_{p_1 + \ldots + p_k = n-1} G_{p_1,h} \cdot \ldots \cdot G_{p_k,h}$$

b) Soit $G_h(z)$ la série génératrice des $G_{n,h}$. Etablir la relation de récurrence suivante (elle est résolue dans l'exemple 12 du §3) : $G_{h+1}(z) = \dfrac{z}{1 - G_h(z)}$. Quelle est la valeur de $G_0(z)$?

40. On note f_n le nombre de forêts ayant n nœuds (cf. chapitre 7) et $f(x)$ la série génératrice associée aux f_n. Soit g_n le nombre d'arbres planaires généraux ayant n nœuds et soit $g(x)$ la série génératrice associée aux g_n. En utilisant les définitions des arbres et des forêts, montrer que :

$$f(x) = \frac{1}{1 - xf(x)}.$$

En déduire la valeur des f_n. De l'expression de $g(x)$ en fonction de $f(x)$, déduire la valeur des g_n.

41. On appelle *palindrome* sur un alphabet A de a lettres, un mot dont la lecture de gauche à droite et la lecture de droite à gauche coïncident. Par exemple, $a_1a_2a_1$ et $a_1a_2a_3a_3a_2a_1$ sont des palindromes sur $A = \{a_1, a_2, a_3\}$. On note e_n le nombre de palindromes de longueur n sur A.

a) Calculer la série génératrice exponentielle de e_n.

b) On se donne un deuxième alphabet B de b lettres, disjoint de A et on note e'_n le nombre de palindromes de longueur n sur B. Soit i_n le nombre d'interclassements de longueur n de palindromes de A et B, c'est-à-dire, de mots de longueur n sur $A \cup B$, tels qu'en effaçant les lettres de B (resp. A) on obtienne un palindrome de A (resp. B). Calculer la série génératrice exponentielle des i_n.

On pourra utiliser les développements en série entière de $\mathrm{ch}(z)$ et $\mathrm{sh}(z)$ donnés dans le formulaire.

42. a) Etant donné un ensemble E d'éléments dont e_n sont de taille n, (on exige que la taille d'un élément soit strictement positive), on considère la série génératrice :

$$s(z) = \sum_{n \geq 0} e_n \cdot z^n.$$

On considère l'ensemble $E^* = \bigcup_{p \geq 0} E^p$, où E^p est l'ensemble des suites de p éléments de E. La taille d'une telle suite est la somme des tailles des éléments qui y figurent.

Montrer que la série génératrice de dénombrement de E^*, c'est-à-dire la série génératrice des t_k, où t_k est le nombre d'éléments de taille k, est $t(z) = \frac{1}{1 - s(z)}$.

b) Pour écrire dans un fichier (opération représentée par la lettre w) éventuellement plusieurs fois de suite, il faut d'abord l'ouvrir (opération représentée par la lettre o); après quoi, on le ferme (opération représentée par la lettre c). Une «transaction» élémentaire peut être, par exemple, oc, ou $owwwc$. Quelle est la série génératrice associée à l'ensemble des transactions élémentaires quand on prend pour taille d'une transaction le nombre d'opérations?

Une transaction quelconque est une suite de transactions élémentaires. Quelle est la série génératrice associée à l'ensemble des transactions quelconques? En déduire le nombre de transactions quelconques de longueur n.

c) On se donne deux ensembles de mots E et F, écrits respectivement sur des alphabets disjoints A et B. Ils contiennent respectivement e_n et f_n mots de longueur n. On dit qu'un mot est un *interclassement* d'un mot de E et d'un mot de F si en effaçant toutes les lettres de A (resp. B), on obtient un mot de F (resp. E).

Donner le nombre d'interclassements de longueur n, en fonction des e_i et f_j pour $i, j \leq n$. En déduire la série génératrice exponentielle $\widehat{w}(z)$ associée à l'ensemble des interclassements, en fonction des séries génératrices exponentielles $\widehat{u}(z)$ et $\widehat{v}(z)$ associées aux ensembles E et F.

d) Quelle est la série génératrice exponentielle associée à l'ensemble des transactions quelconques concernant deux fichiers? Généraliser à k fichiers.

43. Soit R_n le nombre de partitions d'un ensemble à n éléments en classes ayant chacune un nombre pair d'éléments.

a) Montrer que pour $n \geq 1$: $R_n = \sum_{k=1}^{\lfloor n/2 \rfloor} \binom{n-1}{2k-1} R_{n-2k}$

b) Soit $\widehat{R}(z)$ la série génératrice exponentielle des R_n : $\widehat{R}(z) = \sum_{n \geq 0} \frac{1}{n!} R_n \cdot z^n$.

Calculer la série dérivée $\widehat{R}'(z)$ de $\widehat{R}(z)$ et en déduire $\widehat{R}(z)$. (On pourra utiliser l'expression et le développement en série entière de $\mathrm{sh}(z)$ donné dans le formulaire.)

Equations de récurrence

44. Prouver par récurrence que $2^n < n!$ pour $n \geq 4$.

45. Trouver l'erreur dans le raisonnement par récurrence suivant.

Soit $P(n)$ l'assertion : «Tous les individus d'un groupe de n personnes ont les cheveux de la même couleur.»

Etape de base : $P(1)$ est trivialement vraie.

Etape d'induction : supposons $P(n)$ vraie. Soit un groupe G de $n+1$ personnes. Formons une liste de ces $n+1$ personnes. Considérons le groupe G_1 des n premières personnes. $P(n)$ est vraie pour ce groupe : toutes les personnes de G_1 ont donc les cheveux de la même couleur. Considérons ensuite le groupe G_2 des n dernières personnes. $P(n)$ est également vraie pour ce groupe : toutes les personnes de G_2 ont donc aussi les cheveux de la même couleur. Or, la $n^{ième}$ personne appartient à la fois à G_1 et à G_2. Donc, toutes les personnes de G ont les cheveux de la même couleur.

Conclusion : $P(n+1)$ est vraie.

46. On trace dans le plan n cercles de taille quelconque, se coupant deux à deux. Ces cercles ne sont pas tangents et trois cercles quelconques n'ont pas de point d'intersection commun. Soit r_n le nombre de régions du plan délimitées par ces cercles (on compte aussi la région infinie). Etablir une équation de récurrence pour r_n et la résoudre.

47. Un enfant possède n francs; calculer le nombre de façons A_n qu'il a de dépenser son argent, s'il achète chaque jour ou bien un bonbon à 1F, ou bien une glace à 2F ou bien un chocolat à 2F. Etablir une relation de récurrence pour A_n. La résoudre :

a) par la méthode de l'équation caractéristique,

b) par la méthode des séries génératrices.

48. Trouver le nombre a_n de mots binaires de longueur n n'ayant pas deux 0 consécutifs. (Indication : a_n est un nombre de Fibonacci.)

49. Déterminer le nombre de mots sur l'alphabet $\{1, 2, 3\}$ de longueur n ayant un nombre pair de 1. On établira une relation de récurrence que l'on résoudra directement.

50. Donner la solution générale de l'équation de récurrence suivante :

$$u_{n+3} - 7.u_{n+2} + 15.u_{n+1} - 9.u_n = 3^n (n \geq 3).$$

51. Donner la solution générale de l'équation du second ordre suivante :

$$T_n = a_n.T_{n-2} + b_n.$$

(Indication : considérer les suites $U_n = T_{2n}$ et $V_n = T_{2n+1}$.)

52. On emprunte une somme S au taux α sur N années. Calculer :

a) La valeur de l'annuité A de remboursement constante (c'est-à-dire, on rembourse A chaque année).

b) La valeur des annuités de remboursement progressives au taux β (c'est-à-dire, chaque année l'annuité A devient $A + \beta.A$).

53. a) Montrer que le nombre $S_{n,m}$ de surjections d'un ensemble à n éléments sur un ensemble à m éléments vérifie la relation de récurrence suivante :

$$S_{n,m} = m.S_{n-1,\ m-1} + m.S_{n-1,\ m}.$$

Préciser pour quelles valeurs de n et m elle est valide et donner les valeurs initiales.

b) Montrer par récurrence sur n que :

$$S_{n,m} = \sum_{0 \leq j \leq m} (-1)^j \binom{m}{j} (m-j)^n$$

c) En déduire le nombre de partitions $P_{n,m}$ d'un ensemble de n éléments en m classes.

d) Application : on achète n cartes postales distinctes qu'on veut toutes envoyer à m amis (un ami peut recevoir un nombre quelconque de cartes, ou même ne rien recevoir du tout). De combien de manières cela peut-il être fait ? Que se passe-t-il si l'on veut que chaque ami reçoive au moins une carte ?

54. On appelle *partage* d'un entier n en m parts la donnée de m entiers $n_1, n_2, \ldots, n_m$ tels que $n_1 + n_2 + \ldots + n_m = n$ $(1 \leq n_1 \leq n_2 \leq \ldots \leq n_m \leq n)$. Par exemple, 1, 2, 4 et 1, 3, 3 sont deux partages de 7 en 3 parts. On note $T(n, m)$ le nombre de partages d'un entier n en m parts. Etablir la relation de récurrence suivante :

$$T(n, m) = \sum_{i=1}^{m} T(n - m, i).$$

Préciser pour quelles valeurs de m et n elle est valable et donner les valeurs initiales.

55. Peut-on résoudre l'équation de récurrence $t_n = n + \sum_{i=1}^{n-1} t_i$, si on ne connaît que $t_0 = 1$? Que faut-il connaître de plus ?

56. Etant donné un arbre binaire B, on note $p(B)$, le nombre de nœuds de B qui ont un sous-arbre gauche non-vide. Soit $p_n = \sum_{|B|=n} p(B)$ le nombre total de tels nœuds dans tous les arbres binaires de taille n. Montrer que :

$$p_n = \sum_{i=1}^{n-1} 2.p_i.b_{n-i-1} + b_n - b_{n-1}, \quad \text{avec } p_0 = 0.$$

(b_n est le nombre d'arbres binaires ayant n nœuds.)

57. Calculer la complexité en nombre d'appels récursifs de la fonction suivante :

```
function rec(n : integer) : integer;
begin
    if n ≤ 1 then rec := 1
    else rec := rec(n − 1) + rec(n − 1)
end rec;
```

Que devient cette complexité si on remplace la dernière ligne de la fonction par : $rec := 2 * rec(n - 1)$?

58. On va montrer que si on connaît l'ordre de grandeur d'une fonction T, pour certaines valeurs de n, alors, sous certaines conditions, on peut en déduire l'ordre de grandeur de T en général.

Soit $b \geq 2$. On se donne deux fonctions T et f qui vérifient les propriétés suivantes :

(i) T est une fonction croissante au sens large à partir d'un certain rang.

(ii) f est une fonction croissante au sens large à partir d'un certain rang, telle que $f(nb) = \Theta(f(n))$.

(iii) $\exists c,\ d \in \mathbb{R}^{+*},\ \exists n_o \in \mathbb{N}$ tels que $\forall n > n_o$, si n est une puissance de b alors

$d.f(n) \leq T(n) \leq c.f(n)$.

a) Montrer que $T = \Theta(f)$.

b) En déduire que si $T(n)$ est donnée par l'équation de récurrence :

$T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + n - 1$, pour $n \geq 2$ et $T(1) = 0$, alors $T(n) = \Theta(n \log n)$.

(On obtient cette équation de récurrence dans l'analyse du tri fusion.)

59. a) On considère un problème qu'on sait résoudre en un temps n^2. Supposons qu'on sache diviser ce problème, en un temps n^α, en deux sous-problèmes de taille $n/2$. Pour quelles valeurs de α est-il plus intéressant de diviser le problème en sous-problèmes, sur des données de grande taille, plutôt que de le résoudre directement ?

b) On considère maintenant un problème qu'on sait résoudre en un temps n^3. Supposons qu'on sache diviser ce problème en un temps 1, en k sous-problèmes de taille $n/2$. Pour quelles valeurs de k est-il plus intéressant de diviser le problème en sous-problèmes, sur des données de grande taille, plutôt que de le résoudre directement ?

60. Donner les ordres de grandeur asymptotiques de chacune des fonctions suivantes définies par une équation de récurrence et telles que $T_i(1) = 1$, $(1 \leq i \leq 5)$.

1) $T_1(n) = 3.T_1\left(\frac{n}{2}\right) + n$

2) $T_2(n) = 3.T_2\left(\frac{n}{2}\right) + n^2$

3) $T_3(n) = 8.T_3\left(\frac{n}{2}\right) + n^3$

4) $T_4(n) = 3.T_4\left(\frac{n}{2}\right) + n^3$

5) $T_5(n) = T_5\left(\frac{n}{2}\right) + c$

Traiter les cas où $c = 1$ et où $c \neq 1$.

61. En utilisant les séries génératrices, établir l'égalité suivante utilisée au §3.3 :

$$\frac{1}{(1-z)^2}.\mathrm{Log}\frac{1}{1-z} = \sum_{n\geq 1}((n+1)H_n - n)\cdot z^n.$$

Lectures conseillées pour l'annexe

Greene & Knuth, *Mathematics for the analysis of Algorithms*, Birkhaüser Boston, 1982.

Flajolet, «Mathematical Methods in the analysis of Algorithms and Data structures», in : Egon Börger (ed.), *Trends in Theoretical Computer Science*, Computer Science Press, 1988.

Knuth, *The art of Computer Programming*, vol. 1 : *Fundamental Algorithms*, Addison-Wesley, 1968.

Purdom & Brown, *The Analysis of Algorithms*, Holt, Rinehart & Winston, 1985.

Riordan, *An Introduction to Combinatorial Analysis*, John Wiley, 1967.

Formulaire

Ensemble des réels : $\mathbb{R}$

$\mathbb{R}^+$: ensemble des réels positifs ou nuls

$\mathbb{R}^*$: ensemble des réels non nuls

$\mathbb{R}^{+*}$: ensemble des réels strictement positifs

Logarithmes : (x, y, z, a, b sont des réels)

$\text{Log } e = 1$

$\text{Log}(x.y) = \text{Log } x + \text{Log } y$

$\log_b x = \dfrac{\text{Log } x}{\text{Log } b}$

$x^y = \exp(y \text{ Log } x)$

$x^{y+z} = x^y . x^z$

$a^{\log_b x} = x^{\log_b a}$

$2^{\log_2 x} = x$

Parties entières : (x est un réel, n et k sont des entiers)

Partie entière inférieure de x : $\lfloor x \rfloor$

Partie entière supérieure de x : $\lceil x \rceil$

$\lfloor x \rfloor = n$ si, et seulement si, $n \leq x < n + 1$

$\lceil x \rceil = n + 1$ si, et seulement si, $n < x \leq n + 1$

$\lceil n/2 \rceil + \lfloor n/2 \rfloor = n$

Si $2^k \leq n < 2^{k+1}$ alors $k = \lfloor \log_2 n \rfloor$

Si $2^{k-1} < n \leq 2^k$ alors $k = \lceil \log_2 n \rceil$

$\lceil \log_2(n+1) \rceil = \lfloor \log_2 n \rfloor + 1$

Combinatoire :

Nombre de *combinaisons* de p éléments parmi n : $\binom{n}{p}$ ou C_n^p

$$\binom{n}{p} = \frac{n!}{p!(n-p)!} \text{ pour } 0 \leq p \leq n, \text{ et}$$
$$= 0 \text{ pour } p > n$$

Nombre d'*arrangements* de p éléments parmi n : A_n^p

$$A_n^p = p!\binom{n}{p} = \frac{n!}{(n-p)!} \text{ pour } 0 \leq p \leq n, \text{ et}$$
$$= 0 \text{ pour } p > n$$

Formule du binôme :

$$(1+z)^p = \sum_{n \geq 0} \binom{p}{n} z^n, \text{ pour tout entier } p$$

Développements en séries entières :

$$(1+z)^\alpha = \sum_{n \geq 0} \alpha(\alpha-1)...(\alpha-n+1)\frac{z^n}{n!}, \text{ pour tout réel } \alpha \text{ non nul}$$

$$\frac{1}{1-\lambda z} = \sum_{n \geq 0} \lambda^n z^n, \text{ pour tout réel } \lambda$$

$$\frac{1}{(1-z)^p} = \sum_{n \geq 0} \binom{n+p-1}{p-1} z^n, \text{ pour tout entier } p \text{ non nul}$$

$$\text{Log } (1+z) = \sum_{n \geq 1} (-1)^{n+1} \quad \frac{z^n}{n}$$

$$\text{Log } \left(\frac{1}{1-z}\right) = \sum_{n \geq 1} \frac{z^n}{n}$$

$$\exp(z) = \sum_{n \geq 0} \frac{z^n}{n!}$$

$$\mathrm{ch}(z) = \frac{\exp(z) + \exp(-z)}{2} = \sum_{n \geq 0} \frac{z^{2n}}{(2n)!}$$

$$\mathrm{sh}(z) = \frac{\exp(z) - \exp(-z)}{2} = \sum_{n \geq 0} \frac{z^{2n+1}}{(2n+1)!}$$

Nombre harmonique d'ordre n :

$H_n = 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n}$

$H_n = \mathrm{Log}n + \gamma + \frac{1}{2n} + o(\frac{1}{n})$, où γ est la constante d'Euler : $\gamma = 0{,}57721...$

Formule de Stirling :

$$n! = \sqrt{2\pi n}\ \frac{n^n}{e^n}\quad (1 + \frac{1}{12n} + o(\frac{1}{n}))$$

Probabilités :

X est une variable aléatoire discrète (à valeurs positives), et P est la mesure de probabilité associée.

Espérance aléatoire de X : E(X)

$$E(X) = \sum_{k \geq 0} k.P(X = k)$$

Variance de X : Var*(X)*

$$\mathrm{Var}(X) = E((X - E(X))^2) = E(X^2) - (E(X))^2$$

Index

Photocomposition et impression
IMPRIMERIE LOUIS-JEAN
BP 87 — 05003 GAP Cedex
Tél. : 92.51.35.23
Dépôt légal : 9 — Janvier 1990
Imprimé en France

COLLECTION INFORMATIQUE

dirigée par Claude Puech et Jean-Marie Rifflet

La Collection Informatique *est dirigée par Claude Puech, Directeur du Laboratoire d'informatique de l'Ecole Normale Supérieure, et Jean-Marie Rifflet, Professeur à l'université Paris VII. Elle a pour objectif de proposer des ouvrages d'excellente qualité, tant d'un point de vue scientifique que pédagogique, destinés plus particulièrement aux étudiants des seconds cycles universitaires, élèves-ingénieurs et ingénieurs confirmés désirant compléter leur formation.*

L'étude des types de données et des algorithmes fondamentaux de l'informatique constitue l'un des enseignements de base en informatique. Ce livre résulte de plusieurs années d'expérience didactique sur le sujet. Il présente les types de données et les algorithmes usuels, dont la connaissance est indispensable à tout informaticien, en développant de façon accessible les résultats récents dans ce domaine. Le langage utilisé pour écrire les algorithmes est Pascal.

*Ce livre introduit les fondements de l'**analyse de la complexité des algorithmes** et la notion de **type abstrait.** Il présente divers types de données et montre comment le choix de tel ou tel type influe sur la réalisation d'un algorithme et ses performances. Il expose en détail les principaux algorithmes pour trois grandes classes de problèmes : recherche, tri, graphes. Les algorithmes présentés sont accompagnés de leur spécification. De plus, leur complexité en place mémoire et en temps d'exécution est évaluée, ce qui en permet une étude comparative.*

Les différents points traités sont amplement illustrés par des exemples et accompagnés de nombreuses figures et par plus de 300 exercices. Une annexe importante est consacrée à la description des outils mathématiques requis.

9 782704 212170

ISBN : 2-7042-1217-1
ISSN : 0989-392 X