Chapitre 3

Révisions : tableaux et listes

Ce chapitre présente quelques rappels sur les structures linéaires : tableaux et listes. On compare notamment l'intérêt algorithmique de chacune de ses structures.

1 Rappels sur les tableaux

Un tableau est une suite d'éléments consécutifs en mémoire, tous de même type. Un tableau est déterminé par :

- sa taille,
- le type des éléments qu'il peut contenir,
- l'adresse de sa première case.

La figure 3.1 représente un tableau contenant les éléments du multi-ensemble $S = \{\!\{15\,,17\,,42\,,42\,,45\,,84\,,99\}\!\}$ (dans l'ordre où ils sont écrits).

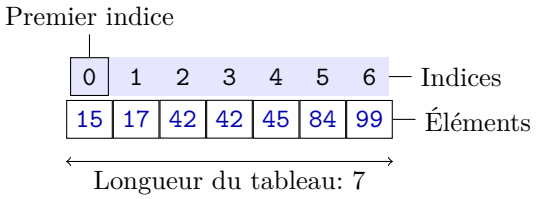

FIG. 3.1 : Un tableau contenant 7 entiers

Remarque. 3.1 Même si on veut seulement représenter un multi-ensemble, un tableau contient en fait plus d'information, car ses éléments sont naturellement ordonnés par leur indice dans le tableau. Un tableau représente donc en fait, mathématiquement, un *vecteur*.

Cela n'empêche pas d'utiliser cette structure pour représenter un multi-ensemble, en ignorant simplement l'ordre des éléments. Autrement dit, si on représente un multi-ensemble par un tableau, permuter les éléments de ce tableau ne change pas le multi-ensemble représenté. Il peut alors être avantageux de maintenir une représentation particulière de ce multi-ensemble (par exemple triée), pour pouvoir effectuer des tâches plus efficacement (par exemple, en utilisant la dichotomie).

Un tableau est implanté en rangeant ses éléments de façon *consécutive en mémoire*, à partir de l'adresse du premier élément du tableau. Ce choix a un avantage et deux inconvénients :

1. **Accès direct**. L'avantage des tableaux est qu'ils permettent l'accès à un élément dont l'indice est connu en temps $O(1)$ (cette propriété est appelée *accès direct*). En effet, pour accéder à l'élément

se trouvant en $i^{\text{ème}}$ case d'un tableau, il suffit d'ajouter à l'adresse de son premier élément i fois la taille de chaque élément. Le calcul de l'adresse de l'élément `p[i]` nécessite donc seulement une addition et une multiplication. L'accès à la valeur `p[i]`, ou `*(p+i)`, nécessite en plus de dé-référencer l'adresse. On a donc un nombre constant d'opérations « élémentaires » à effectuer (une addition, une multiplication et un dé-référencement), ce qui nécessite un temps $O(1)$, c'est-à-dire constant, ***indépendant de l'indice i et de la taille du tableau.***

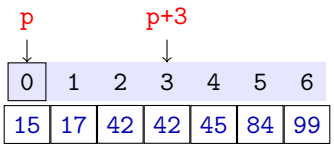

FIG. 3.2 : Accès direct

Remarque. 3.2 En langage C, l'arithmétique des pointeurs effectue la multiplication par la taille des éléments du tableau sans que l'on ait à l'écrire : si on écrit

```
int p[10];
float q[20];
```

alors `p + 5` est l'adresse de la $6^{\text{ème}}$ case de `p` et `q + 15` est l'adresse de la $16^{\text{ème}}$ case de `q`. On n'a pas besoin de (et il ne faut pas, en fait) multiplier par `sizeof(int)` dans le cas de `p`, ou par `sizeof(float)` dans le cas de `q`.

2. **Redimensionnement coûteux.** Dans certains langages, le nombre d'éléments d'un tableau est fixe et décidé au moment de l'allocation mémoire. Dans ce cas, il n'est pas possible d'étendre un tableau dont toutes les cases sont occupées pour y ranger un élément supplémentaire, sans faire soi-même une recopie à la main.

   Il existe cependant des langages permettant d'étendre un tableau. En C, vous avez vu la fonction de la bibliothèque standard de prototype `void * realloc(void *ptr, size_t size);`. Cette fonction permet de redimensionner un tableau déjà alloué par une des fonctions `malloc` ou `calloc`. Par exemple, l'instruction C :

   ```
   realloc(p, 10*sizeof(int));
   ```

   fait passer de la situation de la Figure 3.3 (a) à celle de la Figure 3.3 (b) (si elle réussit, et en supposant que `p` est un tableau contenant des éléments de type `int`).

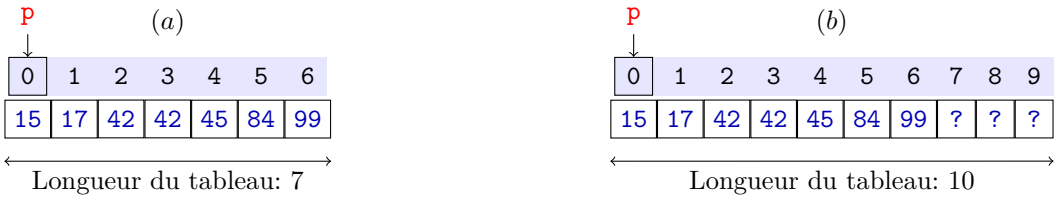

FIG. 3.3 : Redimensionnement d'un tableau

Cependant, même s'il y a assez de mémoire disponible, redimensionner un tableau peut conduire à *recopier* tout le tableau. C'est le cas s'il n'y a pas assez d'espace contigu en mémoire, par exemple si d'autres données ont déjà été allouées « de chaque côté du tableau », comme dans la Figure 3.4.

Remarque. 3.3 Dans l'exemple de la figure 3.4, redimensionner `p2` serait coûteux même s'il y avait de la place disponible « à sa gauche », puisque l'extension doit se faire sur la droite.

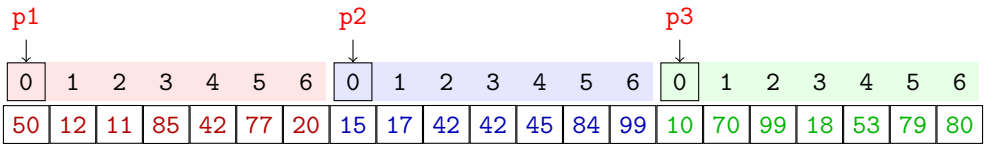

FIG. 3.4 : Une situation où redimensionner `p2` est coûteux

Remarque. 3.4 Au niveau de la programmation, lorsqu'un gestionnaire de mémoire comme celui de la bibliothèque standard C a besoin d'étendre un tableau de taille n, dire que le coût de l'extension est $O(n)$ dans le cas le pire n'est pas tout à fait exact. En effet, pour allouer le nombre voulu de cases, l'algorithme de gestion mémoire doit trouver l'espace ***consécutivement*** en mémoire (car dans un tableau, on range les éléments de façon consécutive). Il est possible qu'il y ait suffisamment d'espace en mémoire, mais pas consécutivement. Si l'algorithme décale certaines zones occupées mémoire pour créer un espace contigu assez grand, le coût du redimensionnement peut être très important (et dépend de la taille de la mémoire disponible, et non plus de celle du tableau).

3. **Insertion et suppression coûteuses**. Enfin, si on veut insérer un élément dans un tableau à un indice i (en supprimant son dernier élément), il faut décaler tous les éléments à un indice $j > i$, ce qui nécessite à nouveau, dans le cas le pire, $O(n)$ affectations. De même, pour supprimer un élément d'un tableau, il faut décaler d'une case vers la gauche tous les éléments d'indice supérieur à l'indice de l'élément à supprimer, ce qui nécessite $O(n)$ affectations dans le cas le pire.

Tableaux : résumé

- L'avantage des tableaux est l'accès ***en temps constant*** à n'importe quel élément dont on connaît la position : le nombre d'opérations pour accéder à un élément ne dépend pas de la taille du tableau.
- Un inconvénient des tableaux est qu'on ne peut pas toujours les redimensionner, et même si on le peut, un redimensionnement peut induire une ***recopie complète*** de tout le tableau. Le nombre d'affectations nécessaires est, dans ce cas, proportionnel au nombre d'éléments du tableau.
- Un second inconvénient est que l'insertion dans un tableau nécessite de décaler tous les éléments à la droite de celui qu'on veut insérer, ce qui demande à nouveau une ***recopie***. Dans le cas le pire, si on insère un élément en première place, ce décalage demande un nombre d'affectations proportionnel au nombre d'éléments du tableau.

2 La structure de donnée de liste

Une autre structure de données dite « linéaire » qui permet de représenter en mémoire des multi-ensembles est la *liste*. Au lieu de représenter un multi-ensemble sous la forme d'un tableau, c'est-à-dire par une suite de cases *consécutives* en mémoire, nous utilisons l'idée suivante : chaque élément est placé dans une structure de deux champs, le premier contenant l'élément à mémoriser, et le second contenant l'adresse de la structure suivante. Le dernier élément de la liste sera associé à une valeur spéciale, notée traditionnellement `NULL`. La liste elle-même peut être représentée par un pointeur sur le premier élément de la liste, appelé tête de la liste, ou `head` (si elle est vide, la liste sera représentée par la valeur `NULL`). Du point de vue de la programmation, en langage C par exemple, la valeur `NULL` est une adresse invalide, ce qui permet de la différencier des pointeurs de la liste, et ainsi de détecter la fin de la liste. La Figure 3.5 représente le même multi-ensemble que précédemment, $S = \{\!\{15, 17, 42, 42, 45, 84, 99\}\!\}$, les éléments apparaissant dans cet ordre.

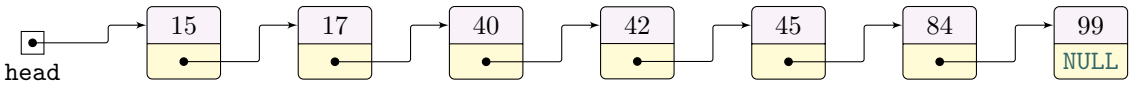

FIG. 3.5 : Une liste simplement chaînée d'entiers

Remarque. 3.5 Comme dans le cas des tableaux, une liste fournit implicitement un ordre sur les éléments (voir Remarque 1).

À nouveau, ce choix d'implantation a des avantages et des inconvénients.

1. Un inconvénient est que contrairement au cas des tableaux, l'accès à un élément n'est pas direct : accéder à un élément nécessite de « suivre » la chaîne de pointeurs, ce qui, à partir du début de la liste, a un coût $O(n)$ dans le cas le pire (où n est la taille de la liste).

2. Au contraire des tableaux, les opérations d'insertion en tête et de suppression peuvent être implantées en $O(1)$. En particulier, le coût de l'ajout de k éléments en tête d'une liste de taille n a un coût $O(k)$, qui ne dépend pas de n.

La Figure 3.6 montre les $O(1)$ étapes de suppression d'un élément de la liste, en supposant qu'il a déjà été recherché (rappel : la recherche elle-même a une complexité $O(n)$, où n est la taille de la liste). On suppose donc l'élément à supprimer pointé par `x`, et on appelle `next` le champ de la structure pointant sur la cellule suivante de la liste.

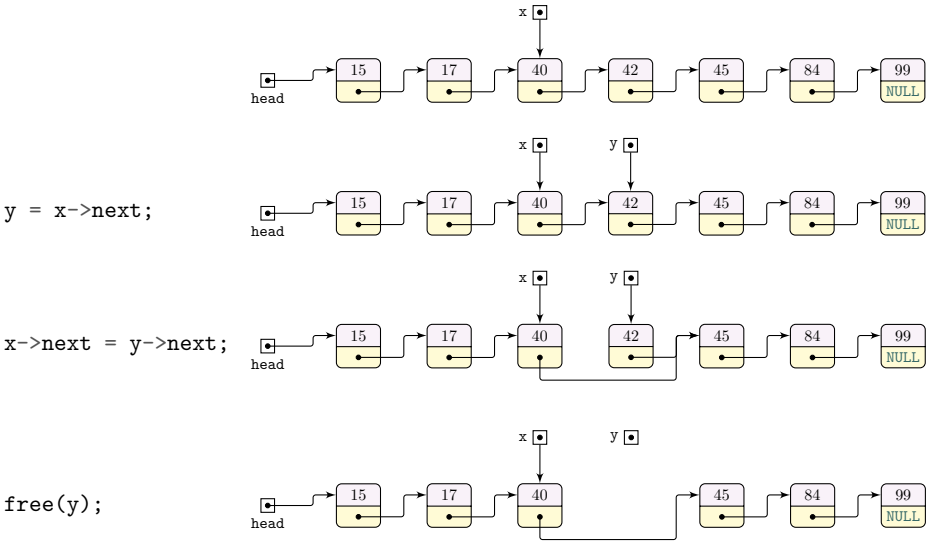

FIG. 3.6 : Suppression dans une liste en temps constant

Listes : résumé

- L'inconvénient principal des listes est l'accès d'un élément en temps $O(n)$ dans le cas le pire, si la liste est de taille n, même si on sait que l'élément est le $i^{\text{ème}}$ dans la liste.
- Un avantage est qu'on peut ***ajouter/supprimer*** un élément en tête, en temps $O(1)$.
- De façon plus générale, si on dispose d'un pointeur sur un élément de la liste, on peut ***insérer*** ou ***supprimer*** un élément en temps $O(1)$, sans ***sans recopie*** des éléments présents.

2.1 Les listes en OCaml

Tout comme dans la plupart des langages, on pourrait définir un type liste en langage OCaml. La définition peut se faire de la façon suivante, si on veut définir les listes d'entiers.

```
type int_list =
  | Empty
  | Cell of int * int_list
```

C'est une définition de type récursive. Elle exprime qu'une liste est :

- soit la liste vide, qu'on choisit de représenter par la valeur `Empty`.
- soit composée d'un couple (entier, liste).

Par exemple, la liste de la figure 3.5 s'écrirait

```
Cell (15,
      Cell (17,
            Cell (40,
                  Cell (42,
                        Cell(45,
                             (Cell(84,
                                   Cell(99, Empty)))))))
```

Remarque. 3.6 Les identificateurs permettant de construire les types, ici `Empty` et `Cell`, doivent commencer par une majuscule (au contraire du nom du type, `int_list`, qui doit commencer par une minuscule).

Une remarque importante est que définir une liste de flottants, ou une liste de chaînes de caractères, se ferait de façon analogue. Comme de nombreux algorithmes sur les listes ne dépendent pas du type des éléments de la liste, il serait intéressant d'avoir une notion de liste dont le type des éléments est un paramètre. Il est possible d'écrire une telle définition en OCaml. Dans ce cas, il faut utiliser un paramètre de type, commençant par une apostrophe, par exemple `'a`, ou `'foo`. Par exemple :

```
type 'foo list =
  | Empty
  | Cell of 'foo * 'foo list
```

OCaml déterminera le type d'une liste en fonction de celui de ces éléments. Par exemple :

```
# Empty;;
- : 'a list = Empty
# Cell(1, Empty);;
- : int list = Cell (1, Empty)
# Cell(1.5, Empty);;
- : float list = Cell (1.5, Empty)
```

Enfin, il est en fait inutile d'écrire notre propre type, car Ocaml fournit déjà un type `'a list`, ainsi que des fonctions et opérateurs. De ce fait,

- Plutôt que la notation lourde ci-dessus pour écrire la liste de la Figure 3.5, on écrit simplement `[15;17;40;42;45;84;99]`. Attention, les éléments sont séparés par des « ; » et non des « , ».
- La liste vide se note donc `[]`.
- On peut ajouter un élément en tête de liste avec l'opérateur `::`. Par exemple, `1::[2;3]` est la liste `[1;2;3]`, qu'on peut aussi obtenir par `1::2::3::[]`.

- On peut concaténer deux listes grâce à l'opérateur `@`. Par exemple, `[1;2;3] @ [4;5]` est la liste `[1;2;3;4;5]`.
- Plusieurs fonctions utiles sont disponibles dans la bibliothèque : voir https ://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html. Il est possible de visualiser leur code à l'aide de l'utilitaire `ocamlbrowser`.