Algorithmique des structures de données arborescentes
Feuille d'exercices 1

1 Révisions : tableaux et listes

Exercice 1.1 1. Rappeler l'algorithme de recherche par dichotomie d'une valeur entière dans un tableau trié d'entiers.

2. Écrire en C une fonction de prototype `int dichotomic_search(int t[], int n, int x);` qui recherche un élément `x` dans un tableau d'entiers `t` de taille `n`, supposé trié. Exploitez le fait que `t` est trié pour utiliser une approche dichotomique. La fonction doit renvoyer une position à laquelle `x` a été trouvé dans `t`, ou `-1` s'il n'est pas trouvé.

3. Quelle est la complexité de la fonction précédente ?

Exercice 1.2 Écrire en OCaml une fonction `append` de type `'a list -> 'a list -> 'a list` qui concatène deux listes (sans utiliser l'opérateur de concaténation `@` de OCaml).
Exemple : `append [1; 2; 3] [4; 5]` s'évalue en `[1; 2; 3; 4; 5]`.
Note. Cette fonction existe dans la bibliothèque standard OCaml, elle peut être utilisée sous le nom `List.append`. Voir https://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html.

Exercice 1.3 Écrire une fonction `mirror` de type `'a list -> 'a list`, qui retourne la liste des éléments dans l'ordre inverse de celui de son argument.
Exemple : `mirror [1; 2; 3; 4; 5]` doit renvoyer la liste `[5; 4; 3; 2; 1]`.
Note. Cette fonction existe déjà dans la bibliothèque standard OCaml, elle s'appelle `List.rev`.

Exercice 1.4 Écrire une fonction `prefix` de type `'a list -> int -> 'a list`, telle que `prefix l n` renvoie la liste des `n` *premiers* éléments de `l`. Si `n` est négatif, renvoyer `[]` et si `l` a moins de `n` éléments, renvoyer `l`.
Exemple : `prefix [1; 2; 3; 4] 2` doit renvoyer `[1; 2]`.

Exercice 1.5 Écrire une fonction `suffix` de type `'a list -> int -> 'a list`, telle que `suffix l n` renvoie la liste des `n` *derniers* éléments de `l`. Si `n` est négatif, renvoyer `[]` et si `l` a moins de `n` éléments, renvoyer `l`.
Exemple : `suffix [1; 2; 3; 4] 2` renvoie `[3; 4]`.

Exercice 1.6 Écrire une fonction `insert` de type `'a list -> 'a -> 'a list` telle que `insert l x` insère dans une liste triée `l` un élément `x`, à sa place dans la liste.
Exemple : `insert [1; 3; 4; 7; 8] 7` doit retourner `[1; 3; 4; 7; 7; 8]`.

Exercice 1.7 Écrire une fonction `insertion_sort` de type `'a list -> 'a list`, qui trie une liste en utilisant la fonction `insert`. Quelle est sa complexité ?

Exercice 1.8 Écrire une fonction `split` de type `'a list -> 'a list * 'a list`, telle que `split l` retourne deux listes d'éléments de `l`, un préfixe `p` et un suffixe `s` telles que `l = p @ s`. On demande aussi que `p` et `s` soient de tailles égales, à un élément près. Si la liste `l` est de longueur impaire, la taille du préfixe retourné sera supérieure (d'une unité) à la taille du suffixe.
Exemple : `split [1; 2; 3; 4; 5]` retourne `([1; 2; 3], [4; 5])`.

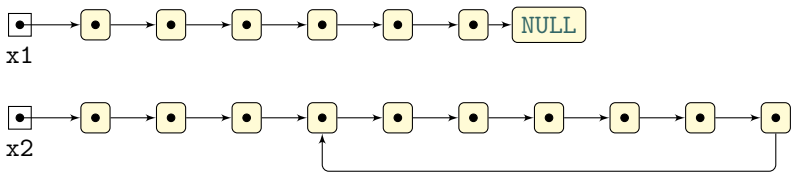
Exercice 1.9 Écrire une fonction `merge` de type `'a list -> 'a list -> 'a list` qui fusionne deux listes triées en une nouvelle liste triée.
Exemple : `merge [1; 4; 4; 7] [1; 2; 6]` retourne `[1; 1; 2; 4; 4; 6; 7]`.

Exercice 1.10 Écrire une fonction `merge_sort` de type `'a list -> 'a list` qui utilise récursivement les deux fonctions `split` et `merge` pour trier une liste. Quelle est sa complexité ?
Exemple : `tri_fusion [8; 2; 4; 4; 5; 6; 19; 5]` doit retourner `[2; 4; 4; 5; 5; 6; 8; 19]`.

Exercice 1.11 — (Plus difficile). On suppose avoir une variable du type suivant, écrit en langage C :

```
struct cell{
  struct cell *next;
}
```

Sur la figure suivante, deux exemples de telles structures sont pointées chacune par une variable, `x1` pour la première et `x2` pour la seconde :



On dit que la variable `x1` représente une *liste*, alors que la variable `x2` représente un *lasso*, car aucune cellule dont l'unique champ est `NULL` n'est accessible depuis `x2`. Enfin, on appelle *longueur* associée à la variable de type `struct cell *` le nombre de cellules accessibles depuis cette variable. Ainsi, la longueur associée à `x1` est 7, et celle associée à `x2` est 10.

1. Écrivez un algorithme de complexité $O(n^2)$ dans le cas le pire, pour déterminer si une variable dont la longueur associée est n représente une liste ou un lasso.
2. Même question avec une complexité $O(n)$. Proposer 5 algorithmes différents, sans changer aucune des valeurs des pointeurs après l'appel. Écrire les algorithmes en C.

2 Propriétés importantes des arbres binaires

Exercice 1.12
1. Trouver des relations de récurrence satisfaites par les fonctions de hauteur, taille, nombre de nœuds internes et nombre de feuilles dans les arbres binaires.
2. En déduire des algorithmes pour calculer la hauteur, la taille, le nombre de nœuds internes et de feuilles d'un arbre binaire donné en argument.
3. Programmer ces algorithmes en OCaml, et optionnellement en C.

Exercice 1.13 On note $h(t)$ la hauteur et $n(t)$ le nombre de nœuds d'un arbre binaire t. Montrer :

a) Qu'on a $h(t) + 1 \leqslant n(t) \leqslant 2^{h(t)+1} - 1$.

b) Que l'arité de tous les nœuds internes de t est 1 si et seulement si $h(t) + 1 = n(t)$.

c) Que l'arbre t est parfait si et seulement si $n(t) = 2^{h(t)+1} - 1$.

Exercice 1.14 Pour un arbre binaire plein t non vide avec $n(t)$ nœuds, $l(t)$ feuilles et $i(t)$ nœuds internes, montrer les deux relations suivantes. Sont-elles vraies pour tous les arbres binaires ?

$$l(t) = 1 + i(t) \quad \text{et} \quad n(t) = 2l(t) - 1 = 2i(t) + 1.$$

Exercice 1.15 Dans un arbre binaire, quel est le nombre maximal de nœuds à profondeur d ? Justifier.

3 Notation O()

Exercice 1.16 Exprimer toutes les relations de la forme $f = O(g)$ pour f et g parmi les fonctions suivantes : $f_1(n) = 42n^7 + 12n^5 - 12345$, $f_2(n) = (\log(n))^{42}$, $f_3(n) = n\log_2(n)$, $f_4(n) = n\log_{10}(n)$, $f_5(n) = n^7$, $f_6(n) = n^{0.001}$, $f_7(n) = 1\ .001^n$ et $f_8(n) = 2^n$.