Tri par tas binaires

L'idée de cet algorithme de tri consiste à *structurer les données*, c'est-à-dire à les organisée dans une structure non triviale fondée sur les arbres, les tas binaires, et codée de façon très simple dans de banales listes.
L'algorithme obtenu est asymptotiquement optimal dans le cas le pire. En moyenne, l'algorithme se révèle environ deux fois moins efficace que le tri rapide, mais il ne présente pas le défaut d'avoir une complexité dans le cas le pire en $O(n^2)$, ce qui semble avoir été utilisé pour attaquer certains systèmes informatiques (les bloquer en lançant des listes dans lequel l'algorithme de tri rapide explose).

1 Arbres binaires presque complets

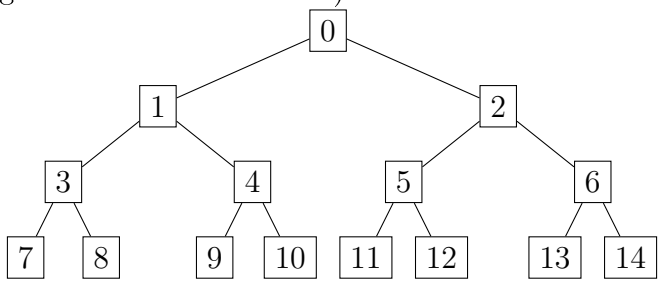
1.1 Vocabulaire des arbres

Un *graphe* est un ensemble (fini) de sommets reliés par des arêtes. Formellement, c'est un couple $\Gamma = (\Gamma_0, \Gamma_1)$ formé par un ensemble fini Γ_0 et une partie Γ_1 de paires de points de Γ_0. On appelle *sommets* ou *nœuds* de Γ les éléments de Γ_0 et *arêtes* les éléments de Γ_1. On dit qu'une arête $\alpha = \{i, j\}$ relie les nœuds i et j.
On appelle *chemin* toute suite finie de sommets reliés à leur successeur par une arête. En symboles, c'est une suite $(i_0, \dots, i_\ell)$ d'éléments de Γ_0 telle que pour un indice k entre 0 et $\ell-1$, la paire $\{i_k, i_{k+1}\}$ est une arête de Γ_1. On dit que le chemin relie i_0 et i_ℓ et que sa longueur est ℓ. Un chemin est un *cycle* si $i_0 = i_\ell$. Un chemin est *élémentaire* si tous les sommets qui le constituent sont tous distincts.
Un graphe est un *arbre libre* s'il est connexe et acyclique, c'est-à-dire si deux nœuds peuvent toujours être reliés par un chemin et s'il n'existe pas de cycle élémentaire de longueur non nulle. Dans un arbre libre, deux points peuvent toujours être reliés par un unique chemin élémentaire.
Un *arbre enraciné*, ou simplement *arbre*, est un arbre libre dans lequel on a choisi un nœud appelé *racine* : formellement, c'est un couple (Γ, r) où $\Gamma = (\Gamma_0, \Gamma_1)$ est un arbre libre et $r \in \Gamma_0$.
La *profondeur* d'un nœud j dans un arbre (Γ, r) est la longueur de l'unique chemin élémentaire qui relie r et j. Par exemple, la profondeur de la racine est 0, la profondeur de ses voisins est 1. La *hauteur* d'un arbre est la profondeur maximale d'un de ses nœuds. La *hauteur* d'un nœud est la différence entre la hauteur de l'arbre et la profondeur du nœud.
Le *père* d'un nœud j qui n'est pas la racine est l'avant-dernier nœud du chemin qui relie la racine à ce nœud : si $(r = i_0, \dots, i_{\ell-1}, i_\ell = j)$ est ce chemin, le père de j est $i_{\ell-1}$. On le notera à l'occasion $\pi(j)$ ou `pere(j)`. Un *fils* d'un nœud j est un nœud dont le père est j –si on veut, c'est un élément de $\pi^{-1}(j)$. Si un nœud j apparaît dans le chemin qui relie la racine à un nœud, on dit que ce nœud est un *descendant* de j. Par exemple, tout fils est un descendant et tout nœud est un descendant de la racine. Si un nœud n'a pas de fils, on dit que c'est une *feuille*, sinon on dit que c'est un *nœud interne*.
Un *arbre orienté* est un arbre dans lequel on choisit un ordre sur l'ensemble des fils de chaque nœud : formellement si un nœud j a f fils, on choisit une bijection $\{1, \dots, f\} \to \pi^{-1}(j)$.

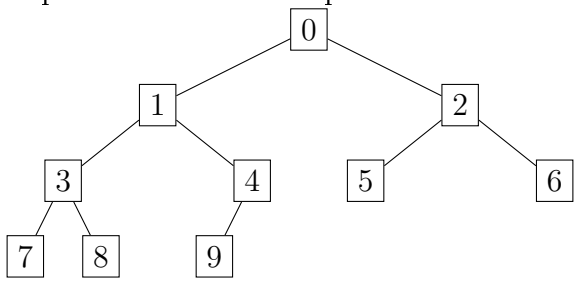
1.2 Arbres binaires (presque) complets

Dans ce qui suit, un *arbre binaire* est un arbre orienté où chaque nœud a au plus deux fils. Si un nœud a deux fils, celui qui correspond à l'indice 1 est appelé le *fils gauche*, celui qui correspond à

l'indice 2 le *fils droit*. S'il n'y a qu'un seul fils, c'est par défaut un fils gauche.
Un arbre binaire est *complet* si, h étant sa hauteur, tous les sommets de profondeur strictement plus petite que h ont exactement deux fils et tous les sommets de profondeur h sont des feuilles.
Il existe une façon standard de numéroter les nœuds d'un arbre binaire complet (parcours en largeur) : on commence par la racine, puis on décrit les sommets par profondeur croissante, de la gauche vers la droite à profondeur donnée (c'est-à-dire qu'on commence par les fils des premiers sommets de la couche précédente, le fils gauche avant le fils droit) :



Cela établit une façon d'indexer les nœuds de l'arbre par $\{0, 1, \ldots, 2^{h+1} - 1\}$.
Un arbre binaire *presque complet* à n nœuds est un arbre que l'on obtient en conservant les nœuds d'indices 0 à n d'un arbre complet ayant plus de n racines. Cela revient à dire que l'arbre est complètement rempli à tous les niveaux, sauf éventuellement le dernier qui est rempli en partant de la gauche et jusqu'à un certain point. Voici un exemple avec 10 sommets :



1. Vérifier que le nombre de nœuds d'un arbre binaire complet de hauteur h est $2^{h+1} - 1$.
2. Dans un arbre binaire complet de hauteur h, quels sont les indices des nœuds de profondeur k, $0 \leq k \leq h$? Quelle est la profondeur du nœud d'indice j, $0 \leq j < 2^{h+1} - 1$?
3. Quelle est la hauteur d'un arbre binaire presque complet à n nœuds ?
4. Si j est l'indice d'un nœud d'un arbre binaire presque complet, montrer que l'indice du fils gauche de j, du fils droit de j et du père de j sont, s'ils existent :

$$FG(j) = 2j + 1, \quad FD(j) = 2j + 2, \quad P(j) = \left\lfloor \frac{j-1}{2} \right\rfloor.$$

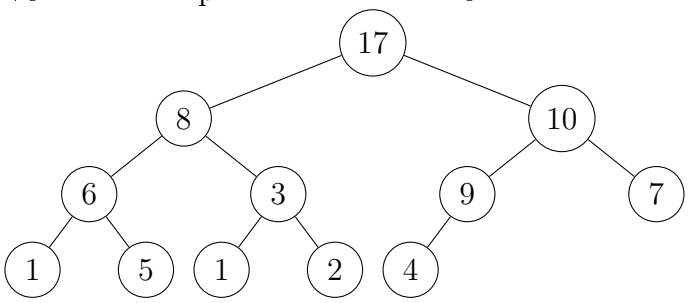
   Implémenter ces fonctions avec Xcas.
5. Dans un arbre binaire presque complet ayant n nœuds, montrer que les indices des feuilles sont $\lfloor (n+1)/2 \rfloor, \ldots, n-1, n$.
6. Dans un arbre binaire presque complet ayant n sommets, montrer que le nombre maximal de descendants d'un fils de la racine est $2n/3$. On pourra commencer par le cas où « la dernière ligne est remplie à moitié » (c'est-à-dire qu'il y a autant de feuilles de profondeur maximale et de profondeur plus petite), et montrer que c'est le cas le pire.
7. Dans un arbre binaire de hauteur h, il y a au plus 2^{h-k} nœuds de hauteur k.

2 Tas binaires

Un *arbre binaire étiqueté* est la donnée d'un arbre binaire et d'une application définie sur l'ensemble des nœuds. Les images des nœuds seront appelés les étiquettes : ce sont des réels, des entiers, des chaînes de caractères... On supposera pouvoir comparer les étiquettes, c'est-à-dire disposer d'une relation d'ordre sur l'ensemble image.
Un *tas binaire* est un arbre binaire étiqueté auquel on associe un entier inférieur au nombre de sommets t qu'on appelle sa *taille*, et qui satisfait à la *propriété du tas* : l'étiquette d'un nœud j est supérieure à l'étiquette de ses fils dont l'indice n'est pas plus grand que la taille :

$$A[j] \geq A[FG(j)] \text{ et } A[j] \geq A[FD(j)].$$

Si un des fils de j est manquant ou s'il a un indice plus grand que la taille, on oublie la condition correspondante. En d'autres termes, pour la condition de tas, on ne considère que le sous-arbre des t nœuds d'indices les plus petits.
Voici un exemple de tas binaire dont la taille est le nombre de nœuds :



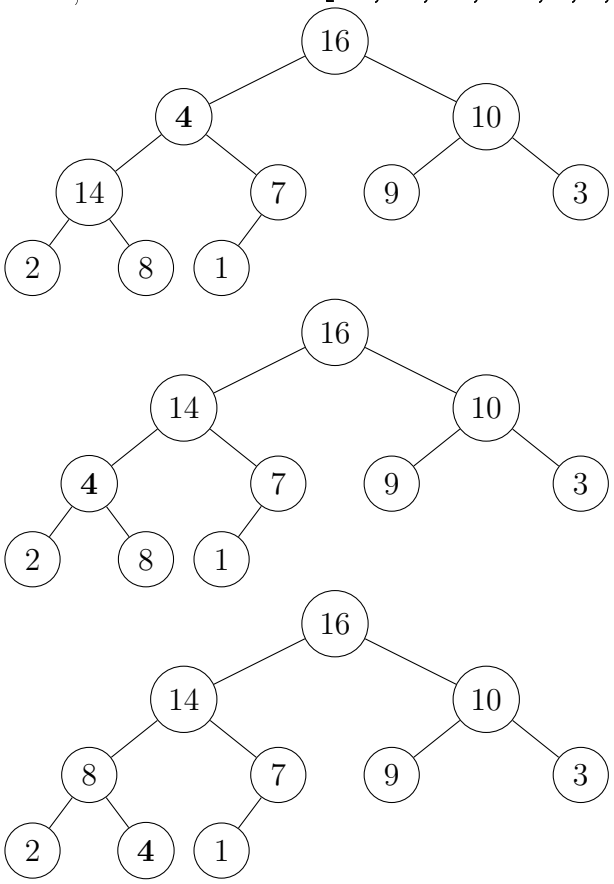
2.1 Tas binaire : entasser

La première procédure utile est donnée ainsi :

Pseudo-code

```
def entasser_max(A,t,j):
    /* A un tableau, t sa taille, j le noeud où on entasse */
    /* calcul des fils du noeud j */
    l := gauche(j)
    r := droit(j)
    /* trouver l'indice m dont l'étiquette est maximale parmi j,r,l */
    si l <= t et A[r] > A[m]
       alors m := l
       sinon m := j
    si r <= t et A[r] > A[m]
       alors m := r
    /* l'indice d'étiquette maximale est m */
    /* on met l'étiquette maximale à la bonne place
       et on rétablit la propriété de tas */
    si m <> j
       alors echanger A[j] et A[m]
             entasser_max(A,t,m)
```

Voici, sur le tableau `A=[16, 4,10, 14,7,9,3, 2,8,1]`, l'effet de `entasser_max(A,11,1)`.



1. Examiner l'effet de la procédure `entasser_max(A,14,2)` sur le tableau
   `A=[27, 17,3, 16,13,10,1, 5,9,12,4,8,9,0]`.
   (Attention, le tableau est indexé de 0 à 13).
2. Pourquoi la procédure s'arrête-t-elle ?
3. On suppose que dans l'arbre d'entrée, les arbres des descendants des fils r et ℓ de j ont la propriété du tas. Montrer qu'en sortie de fonction, l'arbre des descendants de j a la propriété du tas.
4. On note c_n le nombre maximal de comparaisons effectuées pendant le calcul de la fonction `entasser_max` à partir d'un arbre à n nœuds. Démontrer que l'on a pour tout n :
   $$c_n \leq 2 + c_{2n/3},$$
   où $c_{2n/3}$ est un raccourci pour $c_{\lfloor 2n/3 \rfloor}$.
5. Montrer qu'il existe deux constantes positives K et L telles que pour tout entier n, on ait :
   $$c_n \leq K \log n + L.$$
6. Traduire le pseudo-code en Xcas.

2.2 Construction de tas binaires

Partant d'un tableau quelconque, représentant un arbre binaire presque complet étiqueté, on veut le transformer en un tableau représentant un tas binaire en permutant les étiquettes.
On a vu que dans un arbre binaire presque complet à n nœuds, les feuilles portent les indices

$$\left\lfloor \frac{n+1}{2} \right\rfloor, \ldots, n-1, n.$$

On introduit la procédure suivante :

Pseudo-code

```
def construire_tas(A):
    n := size(A)
    pour j = floor((n-1)/2) jusque 1 pas -1
        faire entasser_max(A,n,j)
```

On va commencer par montrer que l'arbre binaire étiqueté obtenu en sortie de boucle a la propriété de tas, puis évaluer la complexité.

1. Montrer que l'assertion suivante est un invariant de boucle : le sous-tableau `A[j..n]`, correspondant aux indices j à n, possède la propriété de tas.
2. Montrer facilement que la complexité de cette procédure est en $O(n \log n)$.
3. Montrer (plus délicat) que la complexité est en $O(n)$. (L'idée est qu'il y a de nombreux sous-arbres de faible hauteur.)
4. Traduire le pseudo-code en Xcas.

2.3 Tri d'un tas

Voici enfin l'algorithme de tri par tas :

Pseudo-code

```
def trie_par_tas(A,t):
    construire_tas(A)
    pour t := size(A) jusque 2 pas -1
        faire echange A[0] et A[t-1]
              entasse_tas(A,t,0)
```

1. Montrer que l'assertion suivante est un invariant de la boucle `pour` : le sous-tableau `A[0..t-1]` correspondant aux indices de 0 à t a la propriété de tas et $A[0] \leq A[t] \leq A[t+1] \leq \cdots \leq A[n]$.
2. Montrer que la complexité de cet algorithme est, dans le cas le pire, $O(n \log n)$.
3. Traduire le pseudo-code en Xcas.