1 Files de priorité et tas

Nous allons étudier ici une nouvelle structure de données : les *files de priorité*. Nous verrons qu'un certain type d'arbre permettra de les représenter de manière extrêmement efficace.

1.1 Définition

Commençons par définir les files de priorité : intuitivement, il s'agit d'un ensemble d'éléments clients à qui sont attribués un *rang de priorité* avant qu'ils ne rentrent dans la file, et qui en sortiront précisément selon leur rang, à savoir que l'élément de rang le plus élevé sera servi en premier. En d'autres termes, il s'agit d'une structure de données sur laquelle opèrent les trois opérations suivantes :

- une fonction d'insertion d'un élément dans la file (avec son rang de priorité) ;
- une fonction qui retourne l'élément de plus haut rang ;
- une fonction qui supprime l'élément de plus haut rang de la file.

On disposera évidemment de plus d'un objet vide correspondant. On utilise cette structure de données par exemple pour planifier l'ordre d'exécution de tâches de priorités différentes dans un ordinateur.

Il est à noter que les piles et les files peuvent se concevoir comme des files de priorité pour lesquelles les éléments arriveraient tous dans l'ordre croissant ou décroissant, respectivement.

Une première manière naturelle de coder les files de priorité est d'employer justement une file d'attente classique, mais alors si l'insertion se fait en temps constant, la fonction qui retourne ou supprime l'élément de plus haut rang (on dira encore le *maximum* dans la suite) nécessite une recherche préalable de cet élément dans la file, ce qui prend en moyenne un temps en $\Theta(n)$, où n est le nombre d'éléments contenus dans la file.

De même, si l'on choisissait alternativement un tableau trié, la recherche du maximum serait alors en $O(1)$, mais l'insertion prendrait un temps $\Theta(n)$ en moyenne.

On va donc utiliser la structure de *tas* (*heap* en anglais), qui est un arbre vérifiant les deux propriétés suivantes :

1. c'est un arbre binaire *complet*, c'est-à-dire un arbre binaire dont tous les niveaux de profondeur sont complètement remplis, sauf éventuellement le dernier (celui de profondeur la plus élevée) où les nœuds sont néanmoins rangés le plus à gauche possible ;
2. la clé de tout nœud est supérieure ou égale à celles de ses descendants.

La figure 1 nous montre un exemple de tas (les clés sont à l'intérieur des cercles ; on reviendra plus loin sur les nombres situés à l'extérieur).

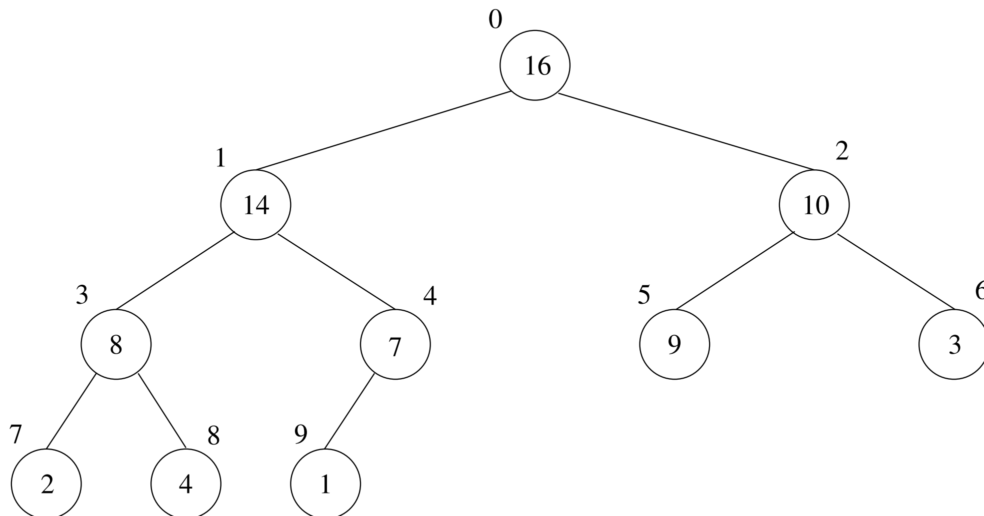

FIGURE 1 – Un exemple de tas

1.2 Opérations sur les tas

Montrons maintenant comment employer un tas pour coder une file de priorité. Tout d'abord, le codage de la fonction qui retourne le maximum est très simple : il suffit de retourner la clé contenue dans la racine (si le tas n'est pas vide). Elle s'exécute donc en temps constant.

En ce qui concerne la **fonction d'insertion d'un élément**, voici brièvement le principe : on commence par créer un nouveau nœud contenant la nouvelle clé qu'on insère en bonne place dans le tas, c'est-à-dire le plus à gauche possible sur le niveau de profondeur le plus élevée (ou s'il est plein, à l'extrême gauche d'un nouveau niveau). On a donc bien toujours une structure d'arbre binaire complet, mais plus nécessairement un tas : il faut alors comparer la nouvelle clé avec celle de son père, les permuter si nécessaire, et recommencer le processus jusqu'à ce qu'il n'y ait plus d'échange à réaliser.

Sur notre exemple (voir figure 2), supposons vouloir insérer la nouvelle clé 15 : on crée un nœud la contenant qu'on insère comme fils droit de 7, puis l'on permute 7 et 15. On compare 15 et son nouveau père 14 ; il faut encore les échanger, puis comparant 15 à 16. On s'aperçoit alors que la clé est en bonne place et on a bien reconstitué un tas.

La **suppression d'un élément** d'un tas non vide commence par l'extraction de la clé contenue dans la racine, qu'on remplace par la clé du nœud situé le plus à droite du niveau de profondeur maximal du tas, nœud que l'on supprime alors. Si l'on obtient toujours un arbre binaire complet, la propriété de tas n'est plus garantie ; on va la rétablir en effectuant cette fois-ci des permutations le long d'un chemin descendant. Pour ce faire, on compare la clé située à la racine avec ses fils, et on la permute avec le plus grand des

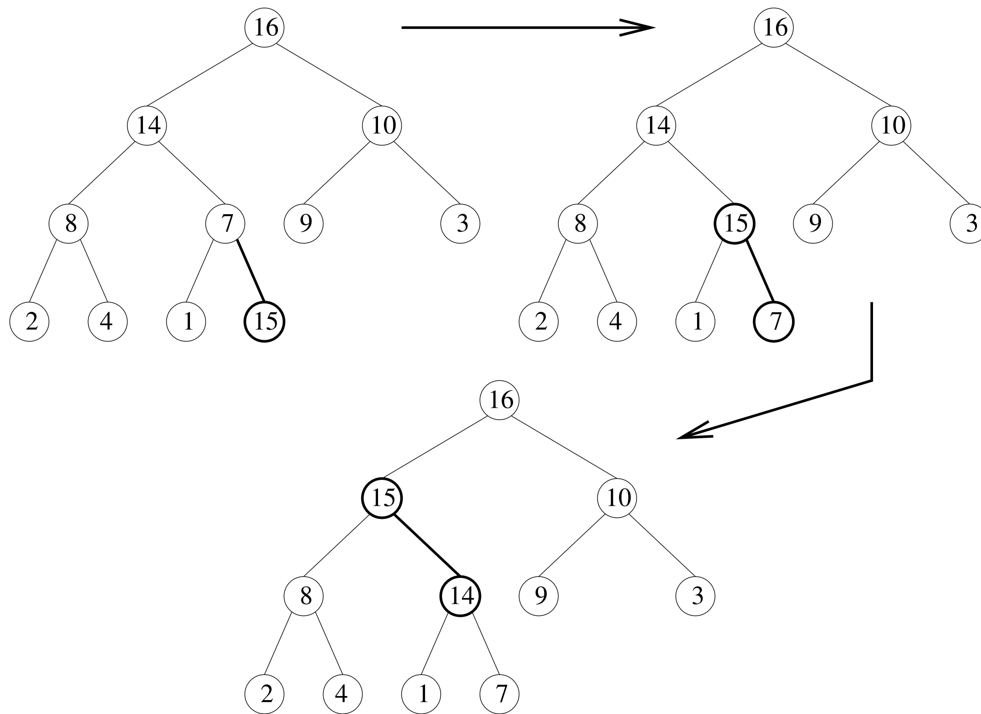

FIGURE 2 – Insertion d'un élément dans un tas

trois (ou des deux) : s'il n'y a pas d'échange à faire, c'est terminé, sinon on recommence plus bas, jusqu'à ce qu'on n'ait pas de permutation à faire ou qu'on arrive à une feuille.

Sur l'exemple précédent (voir figure 3), on supprime donc la clé 16 qu'on remplace par la clé 1, et on supprime ce nœud. On compare alors 1 et ses fils : 14 et 10, on permute donc 1 et 14. Ensuite on compare 1 et ses nouveaux fils, 8 et 7, et l'on permute 1 et 8. On recommence la comparaison avec 2 et 4 ; 4 l'emporte et on l'échange avec 1, qui alors n'a plus de fils, et c'est donc terminé.

Exercice 1 *En notant h la hauteur du tas, exprimer la complexité des opérations de recherche de maximum, d'insertion et de suppression en fonction de h.*

Exercice 2 *Exprimer h en fonction du nombre d'éléments n contenu dans le tas. En déduire la complexité des trois opérations en fonction de n. En déduire que les tas constituent une bonne structure de données pour coder les files de priorité.*

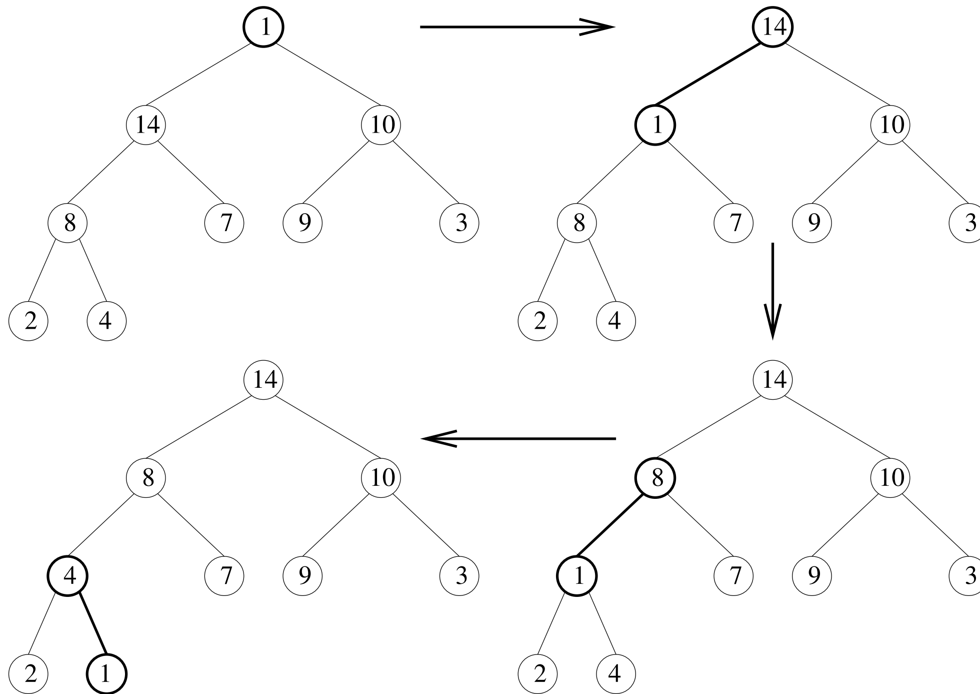

FIGURE 3 – Suppression d'un élément dans un tas

i	0	1	2	3	4	5	6	7	8	9
$A[i]$	16	14	10	8	7	9	3	2	4	1

FIGURE 4 – Représentation d'un tas à l'aide d'un tableau

1.3 Implémentation

Les tas se codent bien sûr tout naturellement comme des arbres mais il se pose alors le problème d'accéder efficacement au père d'un nœud ainsi que le problème de trouver le noeud le plus à droite du dernier niveau. On préfère donc utiliser un codage, particulièrement efficace, à base de tableaux. En effet, on peut coder un arbre binaire complet par un tableau et un index qui en indique la taille (le nombre de nœuds de l'arbre), en numérotant tout simplement les nœuds par un parcours en largeur d'abord : sur l'exemple de la figure 1, les nombres extérieurs aux nœuds sont les indices du tableau correspondant, représenté sur la figure 4.

Avec cette représentation, le calcul des fils et du père d'un nœud est particulièrement simple : le fils gauche du nœud d'indice i, s'il existe, est situé à l'indice $2i + 1$, le fils droit à l'indice $2i + 2$, et le père à l'indice $\lfloor (i-1)/2 \rfloor$. La propriété de tas s'énonce donc alors, pour un tas de taille n représenté par un tableau A :

$$\forall i \in [1, n-1], A[i] \leq A[\lfloor (i-1)/2 \rfloor]$$

Les opérations précédentes s'exécutent sur machine de manière extrêmement efficace, et c'est ainsi qu'on code les tas en pratique.

Plus précisément, un tas est une structure contenant un tableau (`int *t`), un entier indiquant le nombre maximal d'éléments (`int max`) et un entier contenant le nombre d'éléments réellement stockés (`int n`).

```
struct tas
{
  int n;
  int max;
  int *t;
};

typedef struct tas Tas;
```

Exercice 3 *Implémenter les tas et les opérations associées d'initialisation, d'impression, de recherche de maximum et d'insertion au moyen de tableaux.*

Exercice 4 *Ajouter la fonction, plus difficile à implémenter que les précédentes, de suppression de l'élément maximal.*

2 Tri par tas

Un algorithme de tri efficace, appelé *tri par tas* ou *heapsort* peut être déduit de la construction précédente. Il consiste a ajouter chaque élément du tableau à trier dans un tas puis à retirer un à un les éléments du tas.

Exercice 5 *Quelle est la complexité du tri par tas ?*

Exercice 6 *Programmer ce tri.*