IN 101 - TD 12
10 décembre 2010

Énoncé

ENSTA ParisTech

Exercice 1 : Files de priorités et tas

Nous allons étudier ici une nouvelle structure de données : les *files de priorité*, et nous verrons qu'un certain type d'arbres permet de les représenter de manière extrêmement efficace.

Définition. Intuitivement, une file de priorité est un ensemble d'éléments à qui sont attribués un *rang de priorité* avant qu'ils ne rentrent dans la file, et qui en sortiront précisément selon leur rang : 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 renvoie l'élément de rang le plus élevé,
- une fonction qui supprime d'élément de rang le plus élevé de la file.

On disposera de plus d'un objet vide correspondant : la file vide.

Parmi les différentes possibilités de codage des files de priorité, nous utiliserons la structure dite de *tas* ou *maximier* (*heap* en Anglais). Les autres possibilités (file d'attente d'éléments non triés, tableau trié) impliquent ou bien un calcul du maximum en temps linéaire, ou alors le maintien d'un ordre entre tous les éléments qui n'est pas nécessaire.

Une structure de tas est un arbre satisfaisant les propriétés suivantes :

- c'est un arbre binaire *complet*, c'est-à-dire un arbre dont tous les niveaux de profondeurs sont remplis, à l'exception du dernier, celui qui ne comporte que des feuilles, lesquelles sont rangées « le plus à gauche possible »
- la clef de tout nœud est supérieure ou égale à celles de ses descendants.

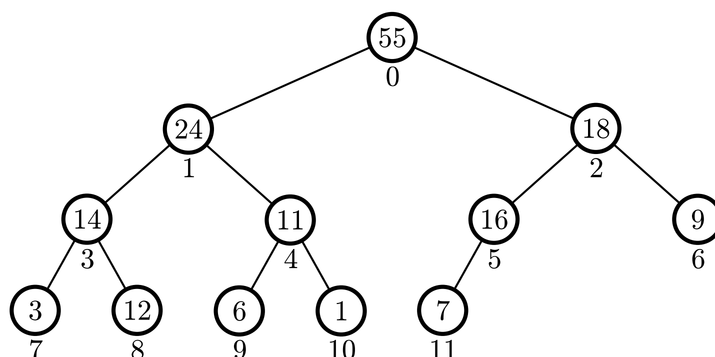

Figure 1 – Un exemple de tas où l'on a numéroté les nœuds à partir de 0.

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 **renvoie le maximum** est très simple : il suffit de renvoyer la clef contenue dans la racine (si le tas n'est pas vide, bien sûr). Cette opération peut donc s'exécuter en temps constant.

En ce qui concerne la fonction **d'insertion d'un élément**, son principe est le suivant :

1. on commence par créer un nouveau nœud contenant la nouvelle clef et on insère ce nœud à la première place disponible de ce tas, c'est-à-dire le plus à gauche possible dans le niveau le plus profond (quitte à créer un nouveau niveau si nécessaire). On obtient alors un arbre qui est certes complet, mais qui peut ne plus être un tas. L'étape suivante vise donc à rétablir l'ordre des nœuds.
2. on compare la nouvelle clef insérée à celle de son père, et on permute ces deux clefs si nécessaire, on réitère ce processus avec la clef du père et celle du grand-père, *etc.* remontant ainsi dans l'arbre jusqu'à ce qu'une permutation ne soit pas nécessaire ou que l'on soit arrivé à la racine. C'est ce qui est illustré sur la figure 2.

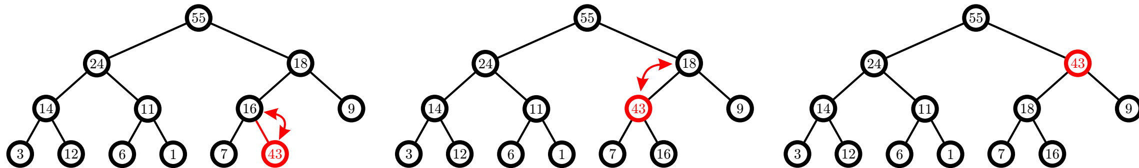

FIGURE 2 – Insertion de l'élément 43 dans un tas.

Enfin, pour la **suppression du maximum** d'un tas non vide, on commence par extraire la clef de la racine (le maximum), puis on la remplace par la clef du nœud le plus à droite du niveau de profondeur maximale, et on supprime ce nœud.

Comme dans la cas de l'insertion, on obtient un arbre binaire complet, mais qui peut ne plus être un tas. Il faut donc rétablir la propriété de tas de l'arbre, en partant de la racine : on compare sa clef à celles de ses deux fils, et la permute avec le maximum de ces deux clefs. On itère ce processus sur le nœud où on a placé la clef, la faisant ainsi descendre dans l'arbre jusqu'à ne plus avoir à permuter, ou jusqu'à arriver à une feuille. C'est ce qui est illustré sur la figure 3.

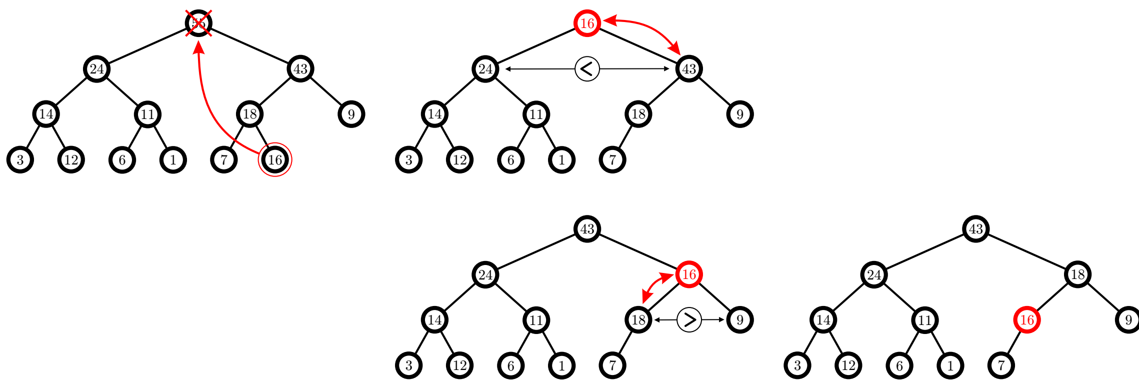

FIGURE 3 – Suppression de la racine d'un tas.

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

1.b] Exprimer h en fonction du nombre d'éléments n contenus dans le tas. En déduire la complexité des trois opérations en fonction de n.

Implémentation. Les tas peuvent bien sûr se coder assez naturellement comme des arbres, mais il faudrait ajouter au codage classique des arbres le moyen d'accéder efficacement au père de chaque nœud, ainsi qu'au nœud le plus à droite du dernier niveau.

Il se trouve qu'il est à la fois plus simple et plus efficace de coder un arbre binaire complet, et donc une structure de tas, dans un tableau où les éléments sont rangés dans l'ordre de leur numérotation (voir figure 4).

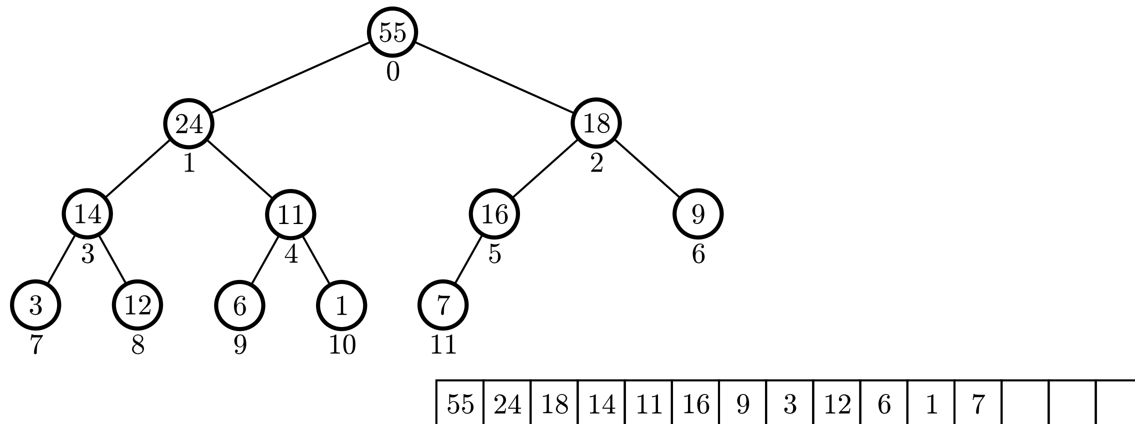

FIGURE 4 – Représentation d'un tas par un tableau.

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

$$\forall i \in [1, n-1],\ A[i] \leq A\left[\lfloor \tfrac{i-1}{2} \rfloor\right]$$

Précisément, en C, un tas est (un pointeur vers) une `struct` contenant un tableau `tab`, un entier `max` représentant le nombre maximum d'éléments et un entier `n` représentant le nombre d'éléments effectivement stockés :

```
1 typedef struct {
2   int max;
3   int n;
4   int* tab;
5 } heap;
```

1.c] Implémentez les tas au moyen de tels tableaux, et programmez les opérations associées de création, d'impression, de recherche du maximum et d'insertion. Voici les signatures des fonctions que vous devez implémenter.

```
1 /* Création d'un tas vide de taille m (et allocation de la mémoire) */
2 heap* init_heap (int m) {...}
3 /* Impression (en largeur) du contenu d'un tas */
4 void print_heap (heap* H) {...}
5 /* maximum d'un tas */
6 int heap_maximum (heap* H) {...}
7 /* insertion de la clef k dans un tas */
8 void heap_insert (int k, heap* H) {...}
```

Programmez une fonction `main` qui initialise un tas pour contenir jusqu'à 31 éléments et y insère les éléments de l'arbre de la figure 4, dans l'ordre. Insérez y ensuite l'élément 43 et vérifiez que l'ordre des nœuds est bien celui de la figure 2.

1.d] Ajoutez à cela la fonction de suppression (avec retour du maximum) ayant comme signature `int heap_delete (heap* H)`. Dans le `main`, supprimez la racine de votre tas, vérifiez que cela vous retourne bien 55 et qu'ensuite, les nœuds du tas sont bien dans l'ordre de la figure 3.

Exercice 2 : Tri par tas

Un algorithme de tri efficace, appelé *tri par tas* (*heapsort* en Anglais) peut être déduit de la construction précédente : il consiste à ajouter chaque élément du tableau à trier dans un tas, puis à retirer un à un les éléments du tas.

2.a] Quelle est la complexité du tri par tas ?

2.b] Programmez ce tri. Il devra modifier le tableau `tab` passé en argument pour en réordonner les éléments. La signature de la fonction est `void heapsort (int* tab, int n)`. Attention, ce tri doit être fait « en place, » en allouant une taille mémoire supplémentaire indépendante de n. Pour cela, le même tableau contient les éléments du tas et ceux non encore insérés dans le tas. Une fois le tas remplis, il faut en extraire les éléments et les placer à la fin du tableau `tab`.

Exercice 3 : Implémentation d'un arbre AVL

Nous allons implémenter une structure d'arbre AVL (sauf la suppression). Pour cela, nous avons besoin d'une structure d'arbre binaire de recherche à laquelle nous ajoutons dans chaque nœud la hauteur du sous-arbre dont ce nœud est la racine (0 si il n'a pas de fils) et le coefficient d'équilibrage correspondant à la différence de hauteur entre le sous-arbre droit et le sous-arbre gauche de ce nœud (un sous-arbre vide a une hauteur de -1).

3.a] Créez une structure adaptée pour représenter un nœud d'un arbre AVL.

3.b] Implémentez une fonction `void insert(node** A, int v)` qui insère la valeur `v` dans l'arbre ayant pour racine `*A`. Pour l'instant cette fonction devra simplement insérer le nouveau nœud et mettre à jour les hauteurs d'arbres et les coefficients d'équilibrage.

3.c] Implémentez les fonctions de rotation `void rot_G(node** A)` qui effectue une rotation à gauche en partant du nœud `*A` et `void rot_D(node** A)` qui effectuent une rotation à droite. Attention, ces deux fonctions doivent aussi mettre à jour les hauteurs et les coefficients d'équilibrage des deux nœuds modifiés.

3.d] Modifiez votre fonction d'insertion pour qu'elle effectue une rotation à gauche, une rotation à droite ou une double rotation pour rétablir la propriété d'arbre AVL quand le coefficient d'équilibrage passe à 2 ou -2.