POLYTECH MARSEILLE Aix*Marseille Université **HUGO3**

Algorithmique

TD n°8 : les arbres équilibrés et les tas

Exercice 1 – Arbres AVL

On définit la structure suivante pour représenter les nœuds d'un arbre AVL :

```
typedef struct AVL_t {
  int label;
  int hauteur;
  struct AVL_t* fils_gauche;
  struct AVL_t* fils_droit;
  struct AVL_t* parent;
} AVL_t;
```

Q 1.1 Écrivez une fonction `rotation_droite` qui prend en argument un pointeur sur un arbre de type `AVL_t` et qui effectue une rotation droite de cet arbre. Si le pointeur est NULL, la fonction doit lever une exception. Ici, vous supposerez que la racine de votre arbre peut avoir elle-même un parent (dans ce cas, c'est un sous-arbre). Votre fonction doit renvoyer la nouvelle racine de votre arbre obtenu après rotation.

Q 1.2 Écrivez une fonction `rotation_gauche` qui réalise les mêmes opérations que la fonction précédente mais avec une rotation gauche.

Q 1.3 Écrivez une fonction `double_rotation_droite` qui réalise les mêmes opérations que la fonction précédente mais avec une double rotation droite.

Q 1.4 Écrivez une fonction `double_rotation_gauche` qui réalise les mêmes opérations que la fonction précédente mais avec une double rotation gauche.

Q 1.5 Écrivez une fonction `create_node` qui prend en argument un entier `n`, qui alloue un nouveau nœud de type `AVL_t` sans parent ni enfants dont le `label` est `n`, et qui renvoie ce nœud.

Q 1.6 Écrivez une fonction `insert_ABR` qui prend en argument un pointeur sur un arbre AVL de type `AVL_t` ainsi qu'un entier `n`. La fonction crée le nouveau nœud contenant `n`, l'insère dans l'arbre comme si c'était un arbre binaire de recherche simple (non AVL, donc sans rotation). La fonction renvoie un pointeur sur le nœud ainsi créé.

Q 1.7 Écrivez une fonction `insert` qui prend en argument un pointeur sur un arbre AVL de type `AVL_t` ainsi qu'un entier `n`. La fonction crée le nouveau nœud contenant `n` et l'insère dans l'arbre existant de telle sorte que celui-ci soit à nouveau AVL.

Exercice 2 – Tas et heapsort

Dans cet exercice, un tas (*heap* en anglais) est une arborescence binaire, construite sur un ensemble E muni d'une relation d'ordre R, ayant les deux propriétés suivantes :

1. tous les niveaux sont entièrement remplis à l'exception, peut-être, du dernier niveau, et ce dernier niveau est rempli « à gauche ».
2. pour tout nœud X et tout fils Y de X, on a $X\,R\,Y$.

La propriété numéro 1, qui n'est pas requise habituellement, est imposée ici afin de pouvoir représenter les tas par des tableaux.

Par exemple, la figure 1 est un tas pour la relation $\geq$. Notez la différence entre un tas et une arborescence binaire de recherche : dans un tas, on a $X\,R\,Y$ pour tout descendant Y de X, alors que dans une arborescence binaire de recherche on a $X\,R\,Y$ pour tout Y dans un sous-arbre droit de X et $Y\,R\,X$ pour tout Y dans un sous-arbre gauche de X.

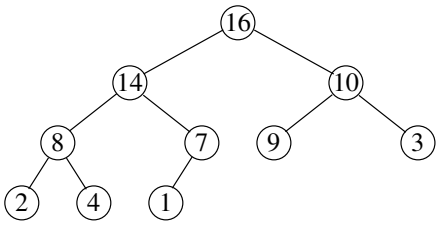


FIGURE 1 – Un tas pour la relation $\geq$.

En principe, on stocke les tas sous forme de tableaux, par exemple celui de la figure 1 peut être représenté par :

16	14	10	8	7	9	3	2	4	1

Le tableau est rempli en balayant l'arborescence du haut vers le bas et de gauche à droite. Si la racine de l'arborescence est stockée à l'index 0 du tableau, alors tout nœud d'index i du tableau a pour fils gauche le nœud d'index $2 \times i + 1$ et pour fils droit le nœud d'index $2 \times i + 2$. De même, tout nœud d'index i a pour parent le nœud d'index $\lfloor \frac{i-1}{2} \rfloor$ (si cette quantité est positive ou nulle bien entendu).

Dans cet exercice, on se propose de manipuler des tas d'entiers munis de la relation $\geq$ en utilisant le type suivant :

```
typedef struct {
  int elts[100]; // les noeuds du tas
  int nb_elts;    // le nombre d'éléments du tas
} heap_t;
```

Q 2.1 Définissez les fonctions :

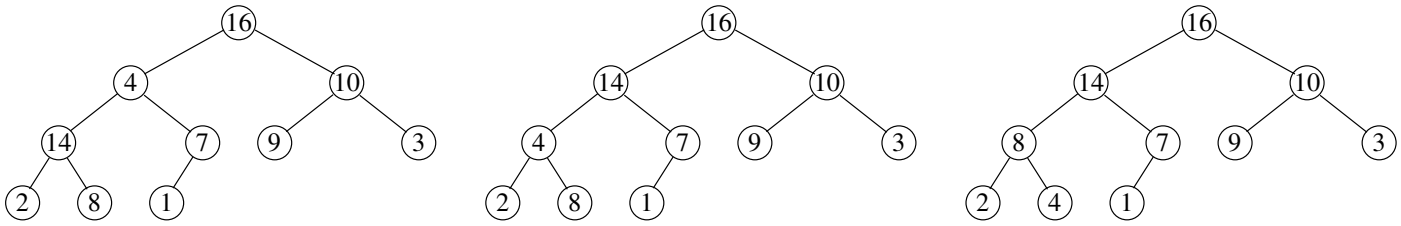
- `left` qui, étant donné un tas et un indice i dans le tableau du tas (on rappelle que la racine a pour indice 0), renvoie l'indice du fils gauche du nœud d'indice i si celui-ci existe et lève une exception sinon.
- `right` qui effectue la même opération mais avec le fils droit.
- `parent` qui est similaire mais renvoie l'indice du parent si celui-ci existe.

Q 2.2 (Propriété du tas)

Dans cette question, on considère une arborescence binaire dont on sait que les sous-arborescences droite et gauche de la racine sont des tas. On veut la transformer de manière à ce qu'elle soit elle-même

un tas. Si la racine, appelons la X, est supérieure à ses enfants, alors on a bien un tas. Sinon, soit Y le plus grand de ses enfants. Échangeons X et Y. Dans ce cas, si la sous-arborescence de racine X est un tas, toute l'arborescence en est aussi un. Il suffit donc de recommencer l'opération avec la sous-arborescence de racine X, et ainsi de suite jusqu'à ce que l'on ait un tas ou que l'on atteigne une feuille (une feuille étant bien évidemment un tas).

Par exemple, si l'on applique cet algorithme sur la sous-arborescence de racine 4 de la figure 2(a), on s'aperçoit que cette dernière n'est pas un tas puisque 4 est inférieur à ses enfants. On échange donc 4 avec le plus grand de ses enfants (ici 14). On n'a toujours pas un tas puisque, maintenant, 4 a pour enfant 8. On échange donc 4 et 8. 4 est maintenant une feuille, on a bien un tas.



(a) échange de 4 et 14 (b) échange de 4 et 8 (c) feuille : plus d'échange

FIGURE 2 – Restauration de la propriété de tas.

Écrivez une fonction `heapify` qui, étant donné un tas et l'index d'un nœud X, réalise l'algorithme ci-dessus sur la sous-arborescence de racine X. *Important pour la suite :* utilisez les fonctions `left` et `right` que vous avez définis précédemment, ainsi que le champ `nb_elts` du type `heap_t`.

Q 2.3 (création du tas)

Écrivez une fonction `build_heap` qui, étant donné un tableau d'entiers dont les éléments sont dans n'importe quel ordre, renvoie un tas constitué de ces éléments. Pour cela, on remarquera que, si ce tableau représentait une arborescence binaire alors les feuilles seraient des tas. Pour transformer tout le tableau en tas, il suffit donc de remonter des feuilles vers la racine et d'appeler la fonction `heapify` sur tous les nœuds rencontrés (en fait, on peut remarquer qu'il suffit d'appliquer `heapify` sur tous les éléments d'index $\lfloor \frac{n-1}{2} \rfloor$ à 0, où n est le nombre d'éléments du tableau).

Q 2.4 (heapsort)

Le heapsort est un algorithme de tri classique. Imaginons que nous voulions trier un tableau par ordre croissant. Si l'on a constitué un tas à partir de ce tableau, alors la racine du tas est le dernier élément du tableau trié. Appelons A le tableau du tas et n le nombre d'éléments dans le tas. Échangeons le dernier élément de A avec la racine X du tas et diminuons la valeur de n d'une unité. Dans ce cas, les n (nouvelle valeur) premiers éléments de A forment un nouvel arbre binaire qui ne contient plus X. Comme celui-ci n'est plus forcément un tas, on applique `heapify` sur cet arbre de manière à obtenir à nouveau un tas et on peut recommencer l'opération. Lorsqu'il ne reste plus qu'un seul élément dans le tas, le tableau A contient l'ensemble des éléments du tableau d'origine triés par ordre croissant.

Définissez la fonction `heapsort` qui implante l'algorithme ci-dessus.

Exercice 3 – Files avec priorité

Dans cet exercice, on reprend la structure de tas vue précédemment. On veut gérer une file avec priorité (permettant par exemple de gérer différents process travaillant en concurrence). Dans une telle file, on veut pouvoir ajouter des éléments à la file et extraire celui ayant la plus grande priorité.

Q 3.1 Définissez la fonction `extract_max` qui, étant donné un tas, renvoie l'élément maximal de ce

tas et le supprime du tas. Idée : échanger la racine du tas avec le dernier élément du tas. Si le tas est vide, la fonction doit lever une exception.

Q 3.2 Définissez une fonction `empty` qui renvoie un tas vide, puis une fonction `heap_insert` qui prend en argument un tas A et un entier X, et qui rajoute ce dernier au tas. Pour cela, il suffit de rajouter l'élément juste après le dernier élément du tas, c'est-à-dire que X devient maintenant une feuille de A. Si le parent de X est supérieur à X, on a bien à nouveau un tas. Sinon, on échange X avec son parent et on recommence les échanges jusqu'à ce que l'on soit remonté à la racine ou bien que X ait un parent supérieur à lui. Si le tableau contenant A est déjà plein — appelons n sa taille — on crée un nouveau tableau de taille $2 \times n + 1$, dont on copie dans les n premières cases le tableau A et auquel on rajoute X.