Algorithmique avancée - Feuille de TD n^o 2

Structures de données

I Tas Binaires : gestion de priorité

1) Construire le tas binaire maximal correspondant au tableau suivant :

10	1	6	12	25	8	14	29	18	11	17	38	27

2) Donner les nombres minimum et maximum d'éléments dans un tas binaire de hauteur h.

3) Montrer que, dans tout sous-arbre d'un tas maximal, la racine du sous-arbre à une clé supérieure à tous les éléments du sous arbre.

4) En supposant que tous les éléments d'un tas maximal sont différents, à quelle position se trouve la valeur minimale dans un tas maximal.

5) Montrer que la taille maximale d'un sous arbre d'un arbre presque plein de taille n est inférieure à $2n/3$.

6) Il est facile de constater que la fonction BuildHeap a une complexité en pire des cas en $O(n \log n)$ pour un tableau de taille n. On désire montrer plus précisément que BuildHeap a une complexité en pire des cas en $\Theta(n)$. Pour cela, répondez aux questions suivantes :
   - Combien y a-t'il de noeuds à une profondeur p dans un tas binaire de hauteur h avec $p < h$?
   - Combien d'opérations doit faire la fonction BuildHeap sur chaque noeud de hauteur p dans le pire des cas ?
   - En déduire que $T_{\text{BuildHeap}}(n) \leq 2^h \sum_{i=1}^{h} \frac{i}{2^i}$.
   - Soit $S = \sum_{i=1}^{h} \frac{i}{2^i}$, calculez $S - \frac{1}{2}S$. En déduire que $S \leq 2$.
   - Conclure sur $T_{\text{BuildHeap}}(n)$.

7) (Travail personnel) Montrer qu'un tas binaire de n éléments à une hauteur de $\lfloor \log_2 n \rfloor$

8) (Travail personnel) Un tableau trié est-il un tas minimal ?

9) (Travail personnel) Montrer que dans un tableau représentant un tas binaire avec n éléments, les éléments d'indices $(\lfloor n/2 \rfloor + 1)..n$ sont des feuilles.

L'algorithme de Tri par tas HeapSort(T) prend un tableau T en entrée et effectue un BuildHeap(T) pour obtenir un tas binaire **maximum**, ensuite il effectue n RemoveBis successifs sur T, où RemoveBis est l'opération particulière qui échange la racine avec le dernier élément (au lieu de simplement remplacer la racine par ce dernier élément) puis effectue un PercolateDown de la racine. Après une suppression le plus grand élément est à la dernière place du tableau ($T[n]$) mais la taille du tas T.Size est décrémentée (T.Size$= n-1$), la capacité du tas restant la même (T.Length$= n$).

11. Effectuez un HeapSort sur le tableau suivant : | 10 | 1 | 6 | 12 | 25 | 8 | 14 |.

12. Exprimez la complexité spatiale du HeapSort en pire cas et en notation Θ.

13. En considérant les différentes hauteurs successives de percolation effectuées en pire cas par l'algorithme, montrez que la complexité temporelle est en $\Theta(n \log n)$.

14. Expliquez intuitivement d'où vient la différence entre les deux complexités temporelles en pire cas du BuildHeap (en $\Theta(n)$) et du HeapSort (en $\Theta(n \log n)$).

II Tas binomiaux : gestion de priorité et union

1. Le tas binomial de la Figure **??** correspond-il aux ajouts successifs par des Add des éléments du tableau de la question 1 ? sinon quel tas obtiendrait-on ?

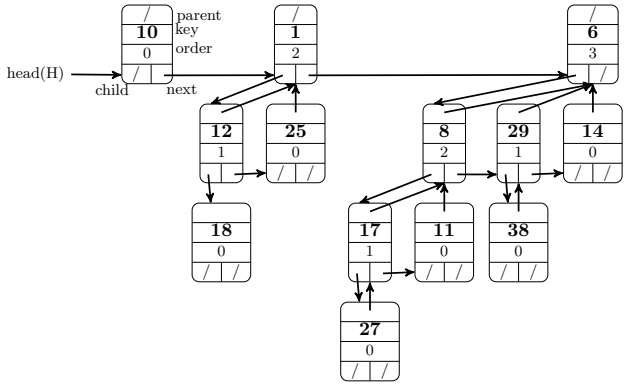


Figure 1 – Représentation d'un tas binomial.

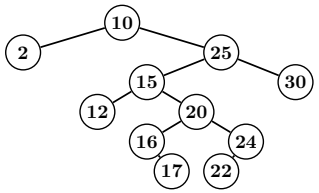
2. Dessiner le tas binomial résultant de la suppression de l'élément minimum dans le tas binomial de la Figure **??**.

3. Dessiner le tas binomial résultant de la suppression de l'élément 17 dans le tas binomial de la Figure **??**.

4. Coder en binaire le nombre n d'éléments du tas binomial obtenu à la question précédente. Donner l'opération sur des nombres binaires qui a permis d'obtenir ce nombre binaire en vous servant de l'union de tas que vous avez réalisée lors de la question précédente.

III Arbres Binaires de Recherche : recherche d'information

1. Effectuez un parcours infixe (InOrderTreeWalk) de l'arbre binaire de recherche suivant :



L'algorithme TreeDelete(T,z) rappelé ci-dessous supprime le nœud z dans l'arbre T, où Transplant(T, x, y) place le sous-arbre de racine y à la place de x dans T

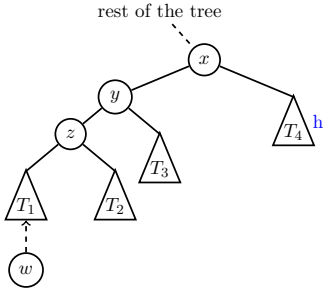
et TreeMinimum(x) renvoie le noeud contenant le minimum du sous-arbre de racine x :

Procedure TreeDelete(T, z)

```
if left(z) = NIL then Transplant(T, z, right(z))
else if right(z) = NIL then Transplant(T, z, left(z))
else
    y ← TreeMinimum(right(z))
    if parent(y) ≠ z then
        Transplant(T, y, right(y))
        right(y) ← right(z)
        parent(right(y)) ← y
    Transplant(T, z, y)
    left(y) ← left(z)
    parent(left(y)) ← y
```

3. Mettre en oeuvre TreeDelete(24), puis TreeDelete(15).
4. (Travail personnel) Est-ce que l'opération TreeDelete est commutative dans le sens où supprimer les nœuds x puis y donne le même arbre que supprimer y puis x.
5. Quand on supprime un nœud z possédant 2 fils, on peut prendre comme nœud de remplacement y le prédécesseur de z au lieu de son successeur. Quels sont les changements à effectuer dans la fonction TreeDelete pour implanter cette stratégie ? On rappelle que les fonctions disponibles pour un noeud dans un arbre binaire de recherche sont *key*, *right*, *left*, *parent* qui renvoient respectivement la clé, le fils droit, le fils gauche et le père du noeud.
6. Comparer l'application de votre fonction de suppression sur 25, puis 20 puis 10 à la suppression avec le TreeDelete classique.

   On remarque qu'on obtient une meilleure performance pratique en choisissant aléatoirement à chaque suppression, soit la procédure qui remplace le nœud supprimé par son successeur soit celle qui le remplace par son prédécesseur. Car cette opération non commutative, risque de déséquilibrer l'arbre ce qui rend moins efficace les opérations de recherche.
7. Arbres AVL : Les arbres AVL [1] sont des arbres équilibrés de telle sorte que, pour chaque nœud x, la hauteur des sous-arbres gauche et droite ne diffère au plus que de 1. A partir de l'implantation des arbres binaire de recherche classique, les arbres AVL sont implantés en ajoutant à chaque nœud un attribut supplémentaire $height(x)$ représentant la hauteur du nœud x.
   (a) L'arbre de la question 1 est-il un AVL ? sinon le modifier par des rotations simples.
   (b) L'insertion dans un arbre AVL commence par une insertion classique dans un arbre binaire de Recherche. Après une insertion, l'arbre peut ne plus être équilibré. Proposez une valeur à insérer dans l'AVL trouvé à la question précédente qui fait perdre à l'arbre la propriété des AVL.
   (c) Après une insertion dans un AVL, on doit remonter en direction de la racine jusqu'au premier nœud s'il existe (sinon l'arbre reste un AVL et on a fini) dont les hauteurs des sous arbre gauche et droit diffèrent de 2. Considérons le cas d'une insertion d'un noeud w dans le sous-arbre T_1 de la figure ci-dessous telle que le premier noeud qui viole la propriété d'AVL est x, avec T_4 de hauteur h. En utilisant le fait que l'arbre était un AVL avant l'insertion, déduire les hauteurs de y, x, z, T_1, T_2 et T_3 avant et après.



   (d) Quelle opération permet de retrouver la propriété d'AVL pour x, y et z ? Dessinez l'arbre obtenu. Quel noeud est racine du nouveau sous-arbre obtenu et quel est sa hauteur, que peut-on en conclure pour le reste de l'arbre ?
   (e) La configuration précédente s'appelle une "Left-Left configuration". Même questions que c) et d) sur le sous-arbre de racine x ayant pour fils gauche y qui a pour fils droit z avec une insertion de w dans le sous-arbre gauche de z. La configuration s'appelle alors une "Left-Right configuration".
   (f) Pour tous les autres cas possibles de rupture de la propriété AVL par insertion, en notant x le premier noeud déséquilibré, y et z ses fils et petits-fils sur la branche concernée par l'insertion, décrivez les opérations permettant de la rétablir.
   (g) (Travail personnel) Démontrer que, dans un arbre AVL possédant n nœuds, la hauteur de l'arbre est $O(\log(n))$. [2]
   (h) (Travail personnel) Montrer que AVLTreeInsert a une complexité temporelle en $O(\log n)$ et réalise $O(1)$ rotations.

IV B-arbres : structure compacte pour la recherche d'information

1. Donnez le B-Tree de degré minimum $t = 3$ résultant de l'insertion successive des lettres A,L,G,O,R,I,T,H,M,S,U,P. Vous détaillerez les passages d'éclatement des noeuds.
2. Donnez, en fonction du degré minimum t, le nombre maximum et minimum de clés qui peuvent être stockées dans un B-Tree de hauteur h. En déduire une expression asymptotique en Θ de h.
3. Dans un B-Tree, en supposant que l'on implémente la recherche de la clé par une recherche dichotomique plutôt qu'une recherche linéaire (l'ordre des clés dans un nœud le permet), montrer que ce changement fait que la fonction BTreeSearch a alors une complexité en $O(\log(n))$, indépendante de t.
4. Expliquer comment trouver la clé de valeur minimum d'un B-Tree et comment trouver le prédécesseur d'une clé dans un B-Tree.
5. (Travail personnel) Écrire le pseudo-code de la fonction BTreeDelete

1. Ce nom provient des initiales des inventeurs Adelson-Velsky et Landis.

2. Indication : prouver qu'un arbre AVL de hauteur h possède au moins F_h nœuds, avec F_h le $h^{\text{ème}}$ nombre de Fibonnacci.