

Correction TD4

Algorithm I



Matthieu Jimenez

Hiver 2015

Correction TD4

Algorithm I

Exercice I

Dans cette exercice, nous allons revoir brièvement les 3 tris simples vu en cours:

(a) Tri par sélection

Une méthode de tri simple est le tri par sélection, il s'agit de chercher le plus petit élément du tableau T[1..N] et on l'échange avec T[1]. Puis on recommence en cherchant le plus petit élément de T [2..n] qu'on échange avec T [2]. Et ainsi de suite...

```
for (int i=1; i<=n-1; ++i) {  
    trouver minimum A[j] de A[i] ..A[n] Echanger A[i] et A[j] }
```

(b) Tri par Bulle

Le tri par bulle n'est pas le tri, le plus efficace mais permet néanmoins de trier une liste avec une complexité O(n²).

```
for (int i=1; i<=n-1; ++i)  
    for (int j=n; j>=i+1; --j)  
        if (A[j]<A[j-1])  
            Echanger A[j] et A[j-1];
```

(c) Tri par insertion

L'idée de cet algorithme est de trier les éléments du tableau un par un, en insérant chaque élément à la bonne place dans la partie du tableau déjà triée. Pensez à la façon dont vous triez un jeu de carte.

```
for (int i=2; i<=n; ++i)  
    insérer A[i] a la bonne position dans A[1] ,..., A[i-1]
```

Maintenant, prenons les listes suivantes:

- (a) [4,3,2,1]
- (b) [2,4,1,3]

Pour chacune de ces listes présenter le résultat de chaque itération de boucle for principale pour chacun des algorithmes

Correction:

		[4,3,2,1]	[2,4,1,3]
Sélection	i=1	[1,3,2,4]	[1,4,2,3]
	i=2	[1,2,3,4]	[1,2,4,3]
	i=3	[1,2,3,4]	[1,2,3,4]
Bulle	i=1	[1,4,3,2]	[1,2,4,3]
	i=2	[1,2,4,3]	[1,2,3,4]
	i=3	[1,2,3,4]	[1,2,3,4]
Insertion	i=2	[3,4,2,1]	[2,4,1,3]
	i=3	[2,3,4,1]	[1,2,4,3]
	i=4	[1,2,3,4]	[1,2,3,4]

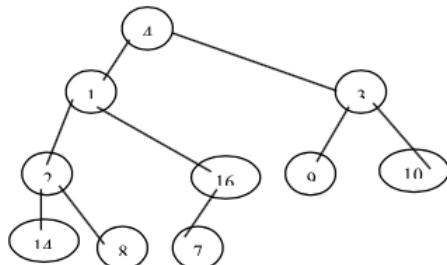
Exercice II

Enoncé:

Illustrer l'opération Buildheap en la simulant sur le heap 4,1,3,2,16,9,10,14,8,7.

1. Dessiner l'arbre correspondant
2. Indiquer la suite des appels Heapify qui vont changer l'arbre et dessiner l'arbre résultant de chaque tel appel.

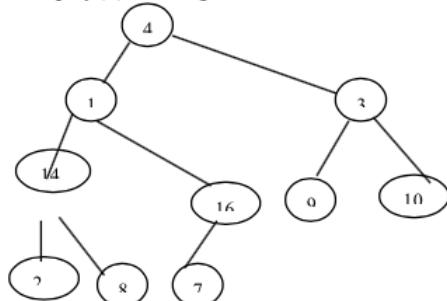
Correction:



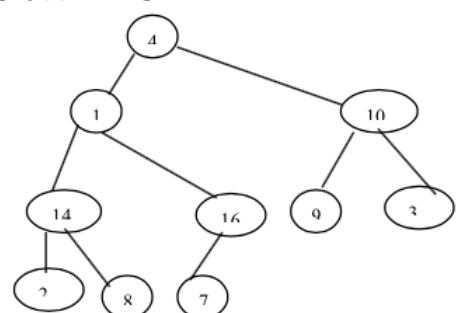
(b) Indiquer la suite des appels Heapify qui vont changer l'arbre et dessiner l'arbre résultant de chaque tel appel.
Suite des appels:

Heapify(7), Heapify(8), Heapify(14), Heapify(10), Heapify(9), Heapify(16): n'ont pas d'effet

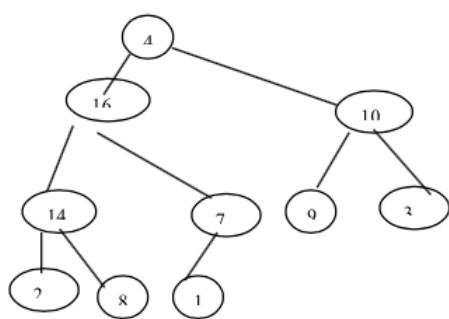
Heapify(2): échange de 2 et 14



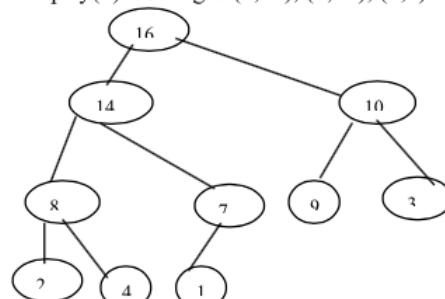
Heapify(3): échange de 3 et 10



Heapify(1): échange (16,1), puis (1,7)



Heapify(4): échanges (4,16), (4,14), (4,8)



Exercice III

Enoncé:

Une "priority queue" (file à priorité) est une structure de données qui maintient un ensemble de données ayant une clé qui est un entier et supportant les opérations suivantes:

- Insert(S, x): insère l'élément x dans l'ensemble S
- Maximum(S) : retourne l'élément avec la plus grande clé
- ExtractMax(S) : retourne et supprime l'élément de S avec la plus grande clé.

(Les priority queues sont utilisées par exemple dans les systèmes d'exploitation pour ranger les tâches par priorité.) Montrez qu'on peut implémenter une priority queue avec un heap de façon à ce que l'opération Maximum(S) prend un temps $O(1)$ et les autres opérations s'exécutent en temps $O(\log n)$. Donner le pseudo-code pour chaque opération et faire une analyse.

Correction:

Maximum: On maintient les clés (avec un pointeur sur les données) dans un heap. La clé maximale se trouve dans la racine et peut donc être récupérée en temps $O(1)$.

Insert: on ajoute la nouvelle clé à la fin du tableau représentant le heap (c'est- à-dire, on rajoute la clé comme nouvelle feuille).

on "pousse" ensuite cette clé vers le haut jusqu'à ce que le parent ait une clé plus grande.
Voici le pseudo- code.

```
ceNoeud = x;  
// x est la nouvelle feuille  
while (ceNoeud.clé > ceNoeud.parent.clé){  
    échanger (ceNoeud.clé , ceNoeud.parent.clé);  
    ceNoeud = ceNoeud.parent;  
}
```

Cette procédure prend temps $O(h)$ où h est la longueur du chemin de la feuille vers la racine.
Dans un heap on a $h = O(\log n)$. L'opération Insert(S, x) se fait donc en temps $O(\log n)$.

Extract Maximum:

on retourne l'élément à la racine, puis on échange le dernier élément (dans le tableau) avec le premier, on supprime le dernier élément et on appelle HEAPIFY(racine).

En fait, on applique la même méthode que dans HEAPSORT.

Elle prend temps $O(\log n)$ (proportionnelle à la hauteur du heap).