Tri par tas

Leçons : 901 ; 903 ; 926

Références : CORMEN–LEISERSON–RIVEST–STEIN, *Introduction à l'algorithmique* (p.139) [?] et LESESVRE–MONTAGNON–LE BARBENCHON–PIERRON, *131 développements pour l'oral* (p. 634) [?]

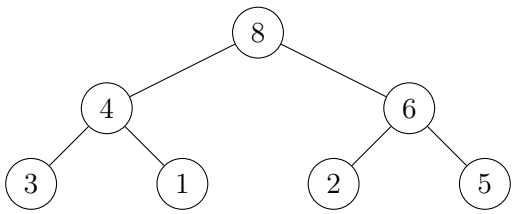
Introduction :

On va présenter ici un algorithme de tri par comparaisons dont la complexité est en $O(n \log n)$, donc c'est un tri optimal pour les tris par comparaisons puisque la complexité minimale est en $O(n \log n)$. Il a la même complexité que les tris fusion et rapide. [1] Le tri par tas n'est pas un tri stable, mais il est en place.

Définition 1 (Tas)**.** Un tas est un arbre binaire où chaque noeud est supérieur ou égal à chacun de ses fils, tous les niveaux de l'arbre doivent être remplis et le dernier niveau est tassé vers la gauche.

On parle en réalité de tas-max, on pourrait définir le tas-min si la relation d'ordre était "inférieur ou égal", mais ici, on ne considèrera que des tas-max.

On implémente le tas dans une structure de tableau indexé en commençant par 1. Ainsi les fils du noeud en case i sont en cases $2i$ pour le fils gauche et $2i+1$ pour le fils droit et le père d'un noeud i est en case $\lfloor i/2 \rfloor$.



Représenté dans le tableau $[\![8;\ 4;\ 6;\ 3;\ 1;\ 2;\ 5]\!]$ avec une taille 7.

Un tas est la donnée de son tableau associé et de sa taille : $T = (tab, taille)$.

On va présenter trois algorithmes : le premier est une fonction auxiliaire, le deuxième construit la structure de tas et le troisième est le tri en lui-même.

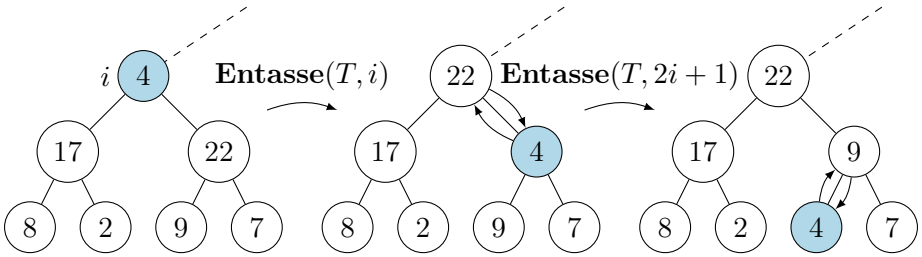
Le premier algorithme est **Entasse**(T, i) qui transforme le sous-arbre de racine i en un tas, en supposant que les deux fils de i sont bien des tas.

1. attention toutefois à quelle complexité on regarde, on parle de temporelle au pire ? en moyenne ?

Algorithme 1 Entasse

```
Procédure Entasse(T, i)
    max ← i ;
    g ← 2i ;        #gauche(i)
    d ← 2i + 1   #droite(i)
    Si (g ⩽ T.taille et T.tab[g] > T.tab[max]) alors
        max ← g
    Fin Si
    Si (d ⩽ T.taille et T.tab[d] > T.tab[max]) alors
        max ← d
    Fin Si
    Si (max ≠ i) alors
        échanger T.tab[i] et T.tab[max] ;
        Entasse(T, max)
    Fin Si

Fin
```



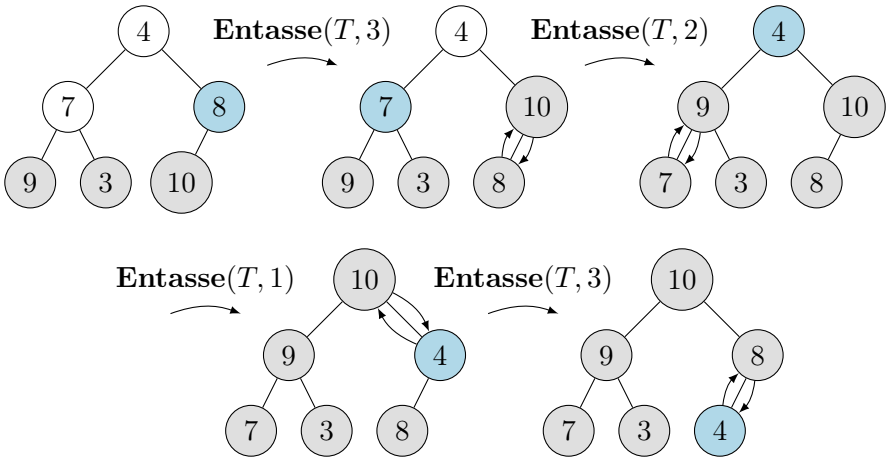
L'algorithme termine car la hauteur est finie. L'algorithme est correct car il place l'élément maximal en position i donc le sous tas inchangé (par exemple $2i$) reste un tas et possède comme père un élément plus grand que tous ses éléments et l'autre sous tas ($2i+1$ dans l'exemple) va avoir l'élément maximal en père et comme ses propres fils sont des tas car $2i+1$ était supposé être un tas, on peut effectuer l'algorithme sur $2i+1$.

Cet algorithme effectue seulement des comparaisons, des affectations ou des échanges (donc en coût constant) puis effectue de nouveau l'algorithme sur un seul de ses fils au pire des cas, donc l'algorithme est en $\mathcal{O}(h - p(i))$ en notant h la hauteur du tas et $p(i)$ la hauteur du noeud i dans le tas. Or $h = \log n$ car le tas est presque complet, donc la complexité de **Entasse** est en au pire $\mathcal{O}(\log n)$.

Le deuxième algorithme **Constuit**(T) sert à construire un tas à partir d'un tableau désordonné. On utilise la programmation dynamique car on va commencer par construire des tas à partir des feuilles et remonter petit à petit jusqu'à la racine en utilisant les tas déjà fabriqués et la fonction **Entasse**.

Algorithme 2 Construit

```
Procédure Construit(T)
    Pour i de ⌊T.taille/2⌋ à 1 faire
        Entasse(T, i)
    Fin Pour
Fin
```



Il suffit de faire une boucle décroissante à partir de $taille/2$ car avant tous les éléments sont des feuilles et donc des tas.

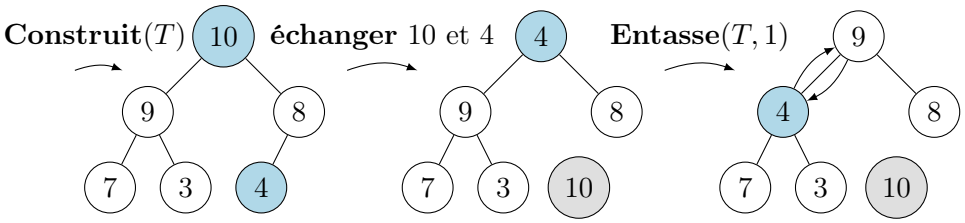
La complexité de **Construit** est naïvement en $\mathcal{O}(n \log n)$ car pour chaque élément on appelle **Entasse**, mais on peut calculer plus finement pour avoir une complexité linéaire.

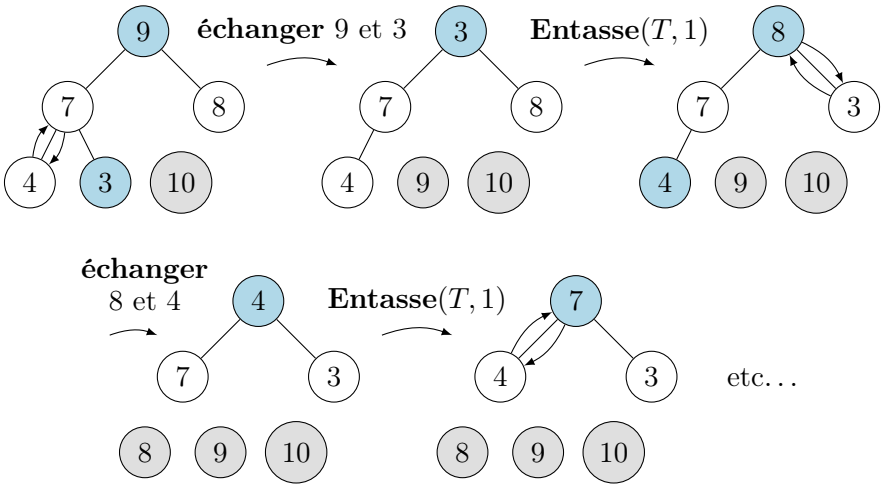
Chaque noeud qui est de profondeur p s'entasse en $\mathcal{O}(h-p)$ et il y a au plus 2^p noeuds de profondeur p dans le tas, donc la complexité de **Construit** est en $\mathcal{O}\big(\sum_{p=0}^{h-1} 2^p(h-p)\big) = \mathcal{O}\big(\sum_{k=1}^{h} 2^{h-k}k\big) = \mathcal{O}\big(n\sum_{k=1}^{h}(1/2)^k k\big) = \mathcal{O}(2n) = \mathcal{O}(n)$ car $2^h = n$.

Le troisième algorithme **Tri_par_tas**(T) est l'algorithme qui permettra de trier notre tableau. Pour cela, on va construire notre tas, puis on va inverser l'élément maximal et le dernier élément du tas, diminuer la taille du tas de 1, puis faire redescendre la racine dans le tas avec la fonction **Entasse**. Ainsi à la fin de l'algorithme les éléments seront triés par ordre croissant.

Algorithme 3 Tri_par_tas

```
Procédure Tri_par_tas(T)
    Construit(T)
    Pour i de T.taille à 2 faire
        échanger T.tab[1] et T.tab[i] ;
        T.taille ← T.taille − 1 ;
        textbfEntasse(T, 1)
    Fin Pour
Fin
```





La complexité du **Tri_par_tas** est en $\mathcal{O}(n) + \mathcal{O}(n \log n) = \mathcal{O}(n \log n)$. Ce tri possède l'avantage d'être un tri en place, il ne demande pas d'allocation mémoire supplémentaire.

Astuces de l'agrégatif :

Le développement peut-être assez long, je choisis de dérouler les algorithmes sur des exemples pour chaque algorithme, cependant il faut bien avoir conscience des preuves de terminaison et correction des algorithmes. On peut faire différemment et prouver la correction de chaque algorithme par exemple.

J'écris l'algorithme **Entasse** en pseudo-code lors du développement.

Questions possibles :

- Quels sont les invariants pour prouver la correction ?
- Connaissez vous d'autres tris en $O(n \log n)$?

  Réponse : Tri fusion, tri rapide
- Est-ce qu'on peut faire mieux que $O(n \log n)$?

  Réponse : pour un tri par comparaison, c'est la borne inférieure, mais avec plus d'hypothèses, on peut avoir des tris linéaires (par dénombrement, par base, par paquets)
- En considérant $([\![5, 7, 4, 4, 6, 2]\!], 6)$, montrer que le tri par tas n'est pas stable.