Tri par tas

Clarence Kineider

Leçons : 901, 903, 926

Référence(s) : Cormen, Leiserson, Rivest, Stein, *Introduction à l'algorithmique.*

Définition : Un tas est un arbre binaire quasi-complet (i.e. tous les niveaux de l'arbre doivent être remplis et le dernier niveau est tassé vers la gauche) où l'étiquette de chaque nœud est supérieure ou égale à celles de ses fils.

Remarque :
On considère ici des tas-max. On peut utiliser des tas-min pour trier des listes par ordre décroissant.

On implante le tas dans une structure de tableau indexé en commençant par 1 (c'est une hérésie, mais ça va simplifier les indices). Ainsi le fils gauche du nœud en case i est en case $2i$ et son fils droit est en case $2i+1$. De plus, le père d'un nœud i est en case $\left\lfloor \frac{i}{2} \right\rfloor$. Un tas est la donnée de son tableau associé et de sa taille (il pourra y avoir des cases non-utilisées à la fin du tableau). On a donc $T = (tab, taille)$. Dans la suite on écrira $T[i]$ à la place de $T.tab[i]$ pour alléger les notations.

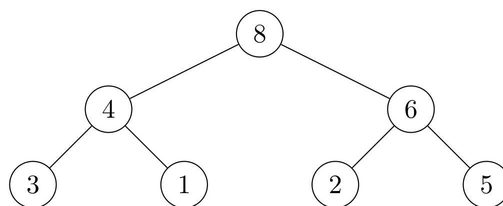
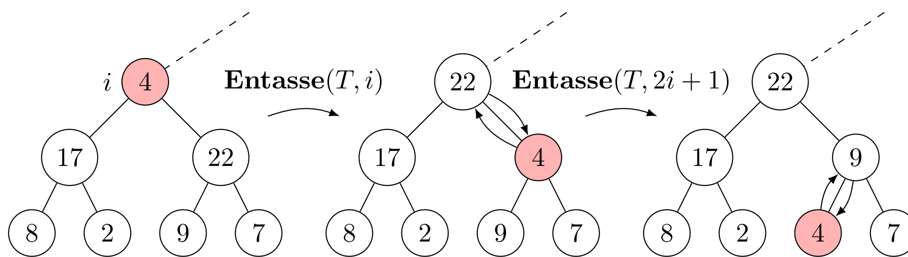


Figure 1 – Ce tas est représenté par le tableau $[\![8;\ 4;\ 6;\ 3;\ 1;\ 2;\ 5]\!]$ avec une taille 7

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 des tas.

Algorithme 1 : Entasse

Entasse(T, i) :
- $max \leftarrow i$
- $g \leftarrow 2i$ #fils gauche de i
- $d \leftarrow 2i+1$ #fils droit de i
- **If** $(g \leq T.taille$ et $T[g] > T[max])$ **then**
  - $max \leftarrow g$
- **If** $(d \leq T.taille$ et $T[d] > T[max])$ **then**
  - $max \leftarrow d$
- **If** $(max \neq i)$ **then**
  - Échanger $T[i]$ et $T[max]$
  - **Entasse**(T, max)



L'appel à **Entasse** termine car la profondeur du nœud en argument croit strictement à chaque appel récursif, et l'arbre T est de profondeur finie.

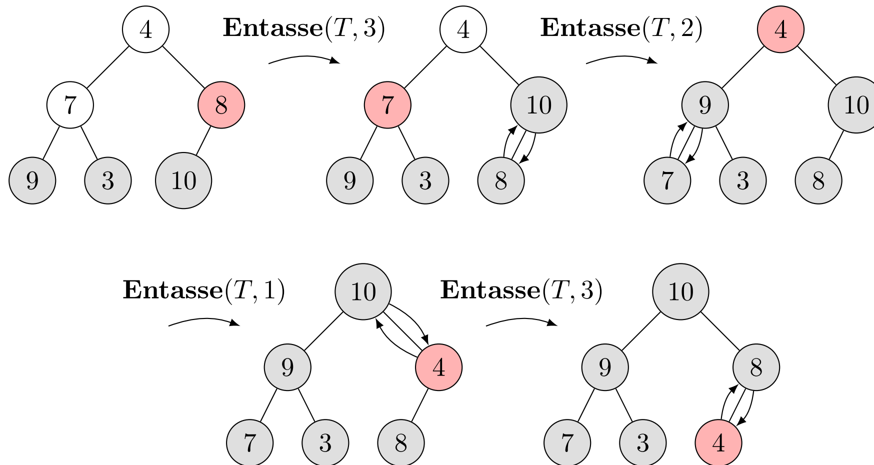
Pour montrer que l'algorithme est correct on raisonne par récurrence sur la hauteur du sous arbre de T enraciné en i, noté T_i. Si $h(T_i) = 1$, i n'a pas de fils donc T_i est un tas après l'appel à **Entasse**(T, i). On suppose que l'algorithme est correct pour des sous arbres de hauteur au plus n. Soit T un arbre et i un nœud de T tel que $h(T_i) = n+1$. Si T_i est un tas, l'algorithme ne modifie pas T donc est correct. Sinon, l'étiquette de i a été échangée avec la plus grande de celles de ses fils. Le fils inchangé reste un tas, le fils k dont l'étiquette à été échangée avec i est un tas après l'appel à **Entasse**(T, k) par hypothèse de récurrence, et l'étiquette de i est plus grande que celle de ses fils. Donc T_i est un tas après l'appel à **Entasse**(T, i).

Cet algorithme effectue seulement des comparaisons, des affectations ou des échanges (donc en coût constant) puis effectue un appel récursif sur un seul de ses fils au pire des cas, donc l'algorithme est en $O(h(T_i))$. Or $h(i) \leq h(T) = \log n$ car le tas est presque complet, donc la complexité de **Entasse** est en au pire $O(\log n)$.

Le deuxième algorithme **Constuit**(T) construit un tas à partir d'un tableau. 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

Construit(T) :
  For i **from** $\lfloor \frac{T.taille}{2} \rfloor$ **to** 1 **do**
    Entasse(T, i)



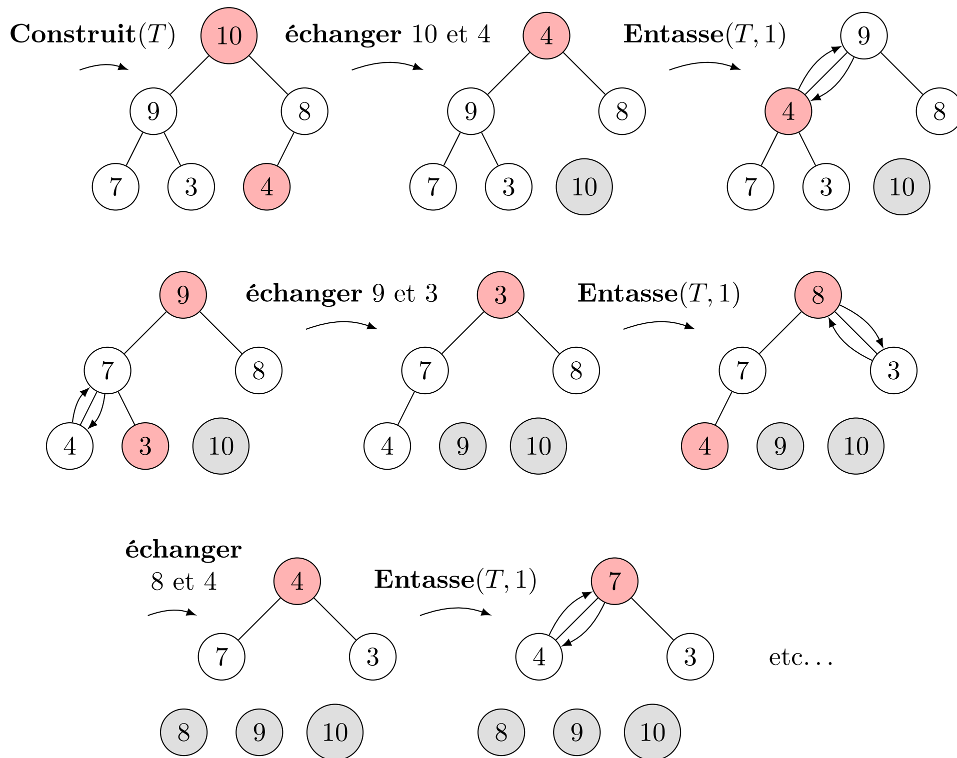
Il suffit de faire une boucle décroissante à partir de $\lfloor \frac{T.taille}{2} \rfloor$ car tous les éléments d'indice supérieur à $\lfloor \frac{T.taille}{2} \rfloor$ sont des feuilles et donc des tas.

La complexité de **Construit** est naïvement en $O(n \log n)$ car pour chaque élément on appelle **Entasse**, mais on peut calculer plus finement pour avoir une complexité linéaire. On note h la hauteur de T. Chaque nœud qui est de profondeur p s'entasse en temps $O(h - p)$ et il y a au plus 2^p nœuds de profondeur p dans le tas, donc la complexité de **Construit** est en $O\left(\sum_{p=0}^{h-1} 2^p(h-p)\right) = O\left(\sum_{k=1}^{h} 2^{h-k}k\right) = O\left(n \sum_{k=1}^{h} (1/2)^k k\right) = O(n)$ car $2^h = n$ et la série de terme général $\dfrac{k}{2^k}$ est convergente.

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

Tri_par_tas(T) :
  Construit(T)
  For i **from** $T.taille$ **to** 2 **do**
    échanger $T[1]$ et $T[i]$;
    $T.taille \leftarrow T.taille - 1$;
    Entasser$(T, 1)$



La correction de l'algorithme repose sur un invariant de boucle dont la démonstration est immédiate :

$$\begin{cases} T[1;...;T.taille] \text{ est un tas} \\ T[T.taille+1;...] \text{ est triée} \\ \forall i \leq T.taille, \forall j > T.taille, T[i] \leq T[j] \end{cases}$$

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

Remarque :
Ce développement peut sembler long, mais il y a beaucoup de chose que l'on dit à l'oral sans écrire, et beaucoup de dessins qui ne sont pas indispensables. L'explication de l'implantation d'un tas peut se faire avec un dessin pour gagner du temps et de la place. La démonstration de la correction de **Entasse** n'a pas besoin d'être autant développée.

Merci à Pierre Le Barbenchon *pour ce développement* ♡