ALGO1 – File de priorité et tas binaire

François Schwarzentruber

February 18, 2021

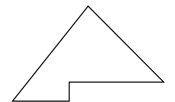
	Tableau	Tableau trié	Tas binaire
`enfiler`	$O(1)$	$O(n)$	$O(\log n)$
`défiler`	$O(n)$	$O(1)$	$O(\log n)$

Avantage des tas :

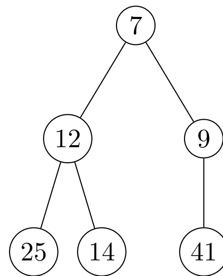
- source d'inspiration pour de bonnes implem (comme tas de Fibonacci) d'une file de priorité
- tri par tas qui est en place et optimal
- structure en place, contrairement aux ABR qui implémentent file de priorité avec la même complexité

1 Arbre binaire presque complet

Définition 1 (arbre binaire presque complet) Un arbre binaire est dit *presque complet* si tous les niveaux sont complets sauf éventuellement le dernier.

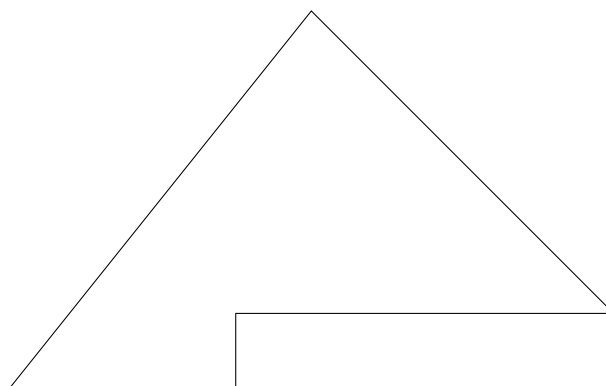


Exemple 2 Voici un arbre binaire presque complet avec 6 nœuds.



Proposition 3 Un arbre binaire presque complet à n nœuds est de hauteur $\leqslant \log_2(n)$.

Démonstration. Soit H la hauteur de l'arbre. Il y a 1 noeuds au niveau 0 (la racine). Il y a 2 noeuds au niveau 1 (les fils de la racine). Il y a au plus 2^ℓ noeuds au niveau ℓ.



$= 1$

$= 2$

$= 2^{H-1}$

≥ 1

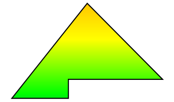
On a $1+2+...2^{H-1}$ sur les niveaux 0 à $H-1$ et au moins un noeud sur le niveau H. Donc : $1+2+...2^{H-1}+1 \leq n$. Autrement dit, $2^H - 1 + 1 = 2^H \leq n$. En passant au log, on obtient le résultat de la proposition.

2 Implémentation d'un arbre binaire presque complet avec un tableau

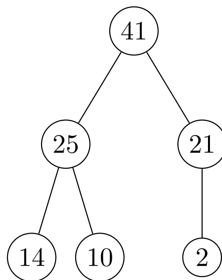
Le tableau contient les éléments d'indice croissant dans l'ordre d'un parcours en largeur de gauche à droite. L'élément en position i a son fils gauche en position $2i$ et son fils droit en position $2i+1$. On note $parent(i) = \lfloor \frac{i}{2} \rfloor$. Soit $T.taille$ le nombre d'éléments dans le tas T. Les feuilles de l'arbre se trouvent entre les indices $\lfloor \frac{T.taille}{2} \rfloor + 1$ et $T.taille$.

3 Définition d'un tas binaire

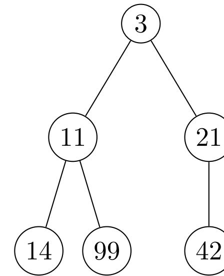
Définition 4 (tas) Un tas est un arbre binaire presque complet où chaque nœud a une valeur plus prioritaire que celle de ses fils.



Exemple 5 (tas-max)



Exemple 6 (tas-min)

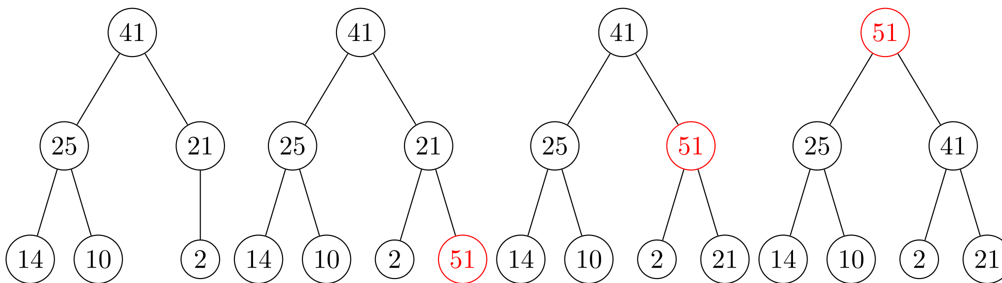


4 Opérations

pre : T est un tas
post : T est un tas contenant les même éléments que $T_{initial}$ avec e en plus
procédure enfiler(T, e)
  $T.taille := T.taille + 1$
  $T[T.taille] := e$
  remonter$(T, T.taille)$

pre: T est un tas sauf en $T[i]$ qui est éventuellement trop grand
post : T est un tas contenant les même élements que $T_{initial}$
procédure remonter(T, i)
  si $i > 1$ **alors**
    $j := parent(i)$
    si $T[i]$ strictement plus prioritaire que $T[j]$ **alors**
      échanger $T[i]$ et $T[j]$
      remonter(T, j)

Exemple 7 Enfiler 51 :



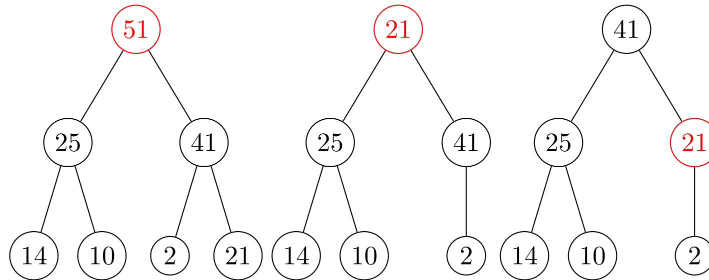
La complexité de remonter est en $O(h) = O(\ln(n))$ et celle de enfiler est donc en $O(\ln(n))$.

Remarque 8 La fonction remonter est récursive terminale. On peut en écrire une version sans avoir à simuler une pile d'appel.

pre :Un tas T non vide
post : l'élément max de T
effet de bord : T contient les même éléments sauf l'élément le plus prioritaire
fonction défiler(T)
    max := $T[1]$
    $T[1] := T$[T.taille]
    T.taille:=T.taille-1
    tasser($T, 1$)
    retourner max

pre : Un tableau T et un indice i où T est presque un tas sauf en $T[i]$
effet de bord : T contient les même éléments mais est un tas
procédure tasser(T, i)
    i_{prio}:= indice $j \in \{i, 2i, 2i+1\} \cap \{1, \ldots, T.taille\}$ avec $T[j]$ le plus prioritaire
    si $i_{\text{prio}} \neq i$ **alors**
        échanger $T[i]$ et $T[i_{\text{prio}}]$
        tasser(T, i_{prio})

Exemple 9



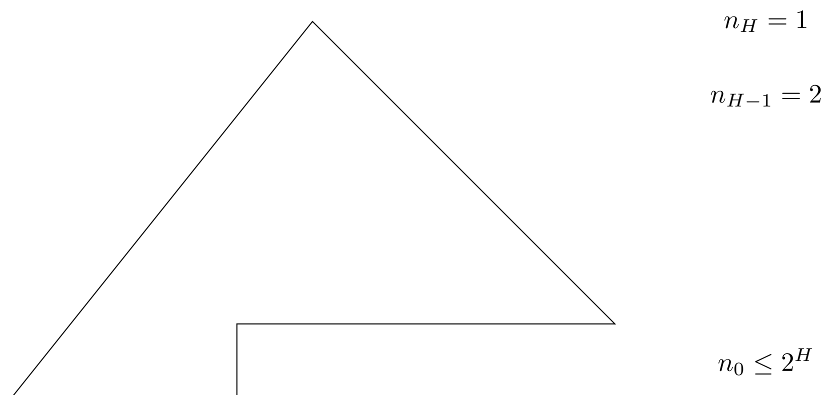
5 Construire un tas

pre : Un tableau T
effet de bord : T contient les même éléments et est un tas
procédure construireTas(T)
    T.taille:=$|T|$
    pour $i = \lfloor \frac{|T|}{2} \rfloor$ à 1 **faire**
        tasser(T, i)

La complexité de `construire_tas` est en $O(n \ln(n))$. Mais, on peut démontrer mieux :

Théorème 10 La complexité de `construire_tas` est en $O(n)$.

Démonstration. L'entier n désigne le nombre d'éléments dans le tas. La hauteur H du tas est $\lfloor \log n \rfloor$. Notons n_h le nombre de noeuds de hauteur h.



La complexité est $C(n) = \sum_{h:=0}^{\lfloor \log n \rfloor} n_h O(h)$.

Lemme 11 On a $n_h \leq \frac{n}{2^h}$.

Démonstration. $n_h \leq 2^{H-h} \leq \frac{n}{2^h}$. ■
On a donc $C(n) = O(n \sum_{h:=0}^{\lfloor \log n \rfloor} \frac{h}{2^h})$. Le lemme suivant permet de conclure que $C(n) = O(n)$.

Lemme 12 $\sum_{h:=0}^{\infty} \frac{h}{2^h} = 2$.

Démonstration. Pour tout $x < 1$, on a

$$\sum_{h=0}^{\infty} x^h = \frac{1}{1-x}.$$

En dérivant en x, on obtient :

$$\sum_{h=1}^{\infty} hx^{h-1} = -\frac{-1}{(1-x)^2} = \frac{1}{(1-x)^2}.$$

En multipliant par x on a :

$$\sum_{h=1}^{\infty} hx^h = \frac{x}{(1-x)^2}.$$

Ainsi, en posant $x = \frac{1}{2}$, on obtient :

$$\sum_{h:=0}^{\infty} \frac{h}{2^h} = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2.$$

■ ■

6 Application : tri par tas

Définition 13 (problème du tri par comparaison)

- Entrée : un tableau T contenant des éléments comparables ;
- Sortie : une permutation triée des éléments de T.

```
procédure  pre : un tableau T
post : T est une permutation triée du tableau initial
triParTas(T)
    construireTas(T)      (tas-max)
    toutDéfiler (T)
```

```
pre : Un tas T
effet de bord : T est une permutation triée du tableau initial
procédure  toutDéfiler(T)
    pour i = T.taille à 2 faire
        invariant: T.taille = i
        T[i] :=défiler(T)
```

Théorème 14 Le tri par tas utilise $O(1)$ de mémoire auxiliaire.

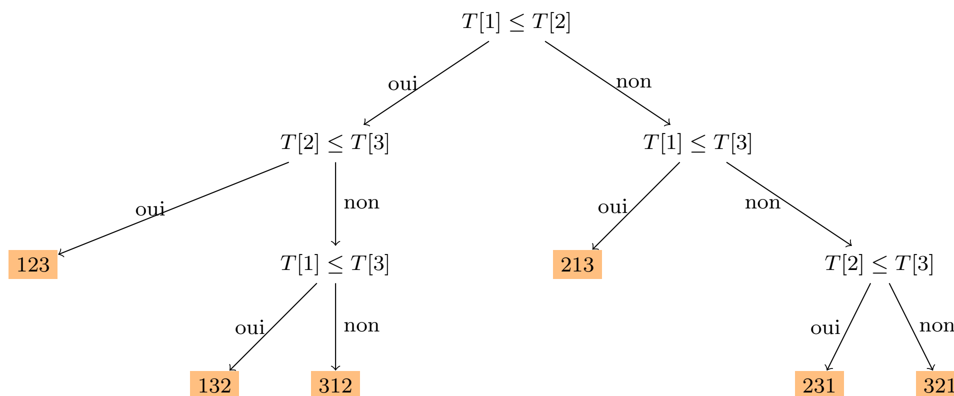
Théorème 15 Le tri par tas est en $O(n \log n)$.

7 Optimalité du tri par tas

Définition 16 Un algorithme est *optimal* si tout algo de même spécification a une complexité moins bonne (ou =).

Théorème 17 Un algorithme de tri par comparaisons requiert $\Omega(n \ln(n))$ comparaisons dans le pire des cas.

Démonstration. Considérons un algorithme quelconque de tri. Le problème ici est de trier des éléments qui sont des *boîtes noires* : les seules opérations dessus sont des comparaisons ($=, <, >, \geq, \leq$). On modélise l'algorithme de tri par une suite d'*arbres de décision* (un arbre pour chaque taille de tableau). Par exemple, l'algorithme 'triInsertion' avec $n = 3$ est modélisé par l'arbre de décision suivant :



algorithme pour une taille de tableau fixé n	modélisé par un arbre de décision
décisions binaires (comparaison)	nœuds internes
fins de l'algorithme	feuilles
exécution	branche
temps d'exécution	longueur d'une branche
temps d'exécution dans le pire cas	hauteur de l'arbre

Le nombre d'opérations dans le pire des cas est la hauteur h de l'arbre.

Lemme 18 $\log_2(n!) \leqslant h$.

DÉMONSTRATION. D'une part $n! \leqslant f$ où f est le nombre de feuilles. En effet, si ce n'était pas le cas, on trouverait une permutation de départ que l'algorithme de tri ne pourrait pas trier.

D'autre part, on a $f \leqslant 2^h$ dans tout arbre binaire. Donc $n! \leq 2^h$. En passant au logarithme, on a $\log_2(n!) \leqslant h$. ■

Lemme 19 $\log(n!) = \Omega(n \ln n)$.

DÉMONSTRATION. Minorons à présent $\log_2(n!)$ par comparaison à l'intégrale de la fonction logarithme.

$$\log(n!) = \sum_{i=1}^{n} \log i \geqslant \sum_{i=\lfloor n/2 \rfloor}^{n} \log i \geqslant \sum_{i=\lfloor n/2 \rfloor}^{n} \log \lfloor n/2 \rfloor \geqslant n/2 \log \lfloor n/2 \rfloor = \Omega(n \ln n).$$

■

Donc $\Omega(n \ln(n)) \leqslant h$. ■

Corollaire 20 Le tri par tas est optimal.

Remarque 1 Comme corollaire, pour tout implémentation d'une file de priorité, l'une des opérations prend $\Omega(\log n)$.

Remarque 2 En Python :

En C++ :

Le tri par tas est utilisé ici.

Timsort : mélange de tri fusion et tri insertion utilisé dans Python

8 Notes bibliographiques

La notion de tas a été inventé par Williams [Wil64] pour présenter le tri par tas. Le tri par tas est présenté dans le livre [CLRS09]. Les tas ont inspiré des structures de données plus riches. Par exemple, les tas binomiaux [Vui78] permettent l'union en $O(\log n)$. Les tas de Fibonacci [FT87] ont de meilleures complexités amorties, qui en font de bonnes implémentations pour les files de priorités dans des algorithmes best-first search, comme l'algorithme de Dijkstra. Les tas de Fibonacci ont été amélioré [BLT12] avec les même complexités mais non amorties.

References

[BLT12] Gerth Stølting Brodal, George Lagogiannis, and Robert Endre Tarjan. Strict fibonacci heaps. In Howard J. Karloff and Toniann Pitassi, editors, *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012, New York, NY, USA, May 19 - 22, 2012*, pages 1177–1184. ACM, 2012.

[CLRS09] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[FT87] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

[Vui78] Jean Vuillemin. A data structure for manipulating priority queues. *Commun. ACM*, 21(4):309–315, 1978.

[Wil64] John William Joseph Williams. Algorithm 232: heapsort. *Commun. ACM*, 7:347–348, 1964.