

Tas binaire — suppression

```
    DELETEMIN( $H$ ) // tas dans  $H[1..|H|]$   
D1  $r \leftarrow H[1]$   
D2 if  $|H| > 1$   
D3 then  $v \leftarrow H[|H|]$ ;  $H[|H|] \leftarrow \text{null}$ ; COULER( $v, 1, H$ )  
D4 return  $r$ 
```

```
    COULER( $v, i, H$ ) // tas dans  $H[1..|H|]$   
C1  $c \leftarrow \text{MINCHILD}(i, H)$   
C2 while  $c \neq 0$  et  $H[c] < v$  do  
C3      $H[i] \leftarrow H[c]$   
C4      $i \leftarrow c$   
C5      $c \leftarrow \text{MINCHILD}(i, H)$   
C6  $H[i] \leftarrow v$ 
```

(COULER est implanté par `percolateDown` dans le livre)

```
    MINCHILD( $i, H$ )  
    // retourne l'enfant avec clé minimale ou 0 si  $i$  est une feuille
```

Tas binaire — efficacité

Hauteur de l'arbre est toujours $\lfloor \lg n \rfloor$

deleteMin : $O(\lg n)$

insert : $O(\lg n)$

findMin : $O(1)$

Tas d -aire

Tas d -aire : on utilise un arbre complet d -aire avec une arité $d \geq 2$.

L'implantation utilise un tableau A :

parent de l'indice i est $\lceil (i - 1)/d \rceil$, enfants sont à $d(i - 1) + 2..di + 1$

ordre de monceau :

$$A[i] \geq A\left[\left\lceil \frac{i - 1}{d} \right\rceil\right] \quad \text{pour tout } i > 1$$

deleteMin : $O(d \log_d n)$ dans un tas d -aire sur n éléments

insert : $O(\log_d n)$ dans un tas d -aire sur n éléments

findMin : $O(1)$

NAGER et COULER : $O(\log_d n)$ et $O(d \log_d n)$

Permet de balancer le coût de l'insertion et de la suppression si on a une bonne idée de leur fréquence

Tas d -aire — construction

Opération **heapify** (A) met les éléments de la vecteur $A[1..n]$ dans l'ordre de monceau.

Triviale ?

$H \leftarrow \emptyset$; **for** $i \leftarrow 1, \dots, n$ **do** INSERT($A[i], H$); $A \leftarrow H$

$\Rightarrow$ prend $O(n \log_d n)$

Meilleure solution :

HEAPIFY(A) // vecteur arbitraire $A[1..n]$
H1 **for** $i \leftarrow n, \dots, 1$ **do** COULER($A[i], i, A$)

$\Rightarrow$ prend $O(n)$

Tas d -aire — construction (cont)

Preuve du temps de calcul : si i est à la hauteur j , alors il prend $O(j)$ de faire $\text{COULER}(\cdot, i, \cdot)$. Il y a $\leq n/d^j$ nœuds à la hauteur j . Donc temps est de

$$\sum_j \frac{n}{d^j} O(j) = O\left(n \sum_j \frac{j}{d^j}\right) = O(n).$$

□

«Évidemment», $O(n)$ est optimal pour construire le tas.

Preuve formelle :

- Trouver le minimum des éléments dans un vecteur de taille n prend $n - 1$ comparaisons, donc un temps de $\Omega(n)$ est nécessaire pour trouver le minimum.
- Avec n'importe quelle implantation de **heapify**, on peut appeler **findMin** après pour retrouver le minimum en $O(1)$.
- Donc le temps de **heapify** doit être $\Omega(n)$, sinon on pourrait trouver le minimum en utilisant **heapify**+**findMin** en un temps $o(n) + O(1) = o(n)$. □

Tri par tas

```
    HEAPSORT(A) // vecteur non-trié A[1..n]  
H1 heapify(A)  
H2 for i ← |A|, ... 2 do  
H3     échanger A[1] ↔ A[i]  
H4     COULER(A[1], 1, A[1..i - 1])
```

$A[1..n]$ est dans l'ordre décroissant à la fin

(pour l'ordre croissant, utiliser un **max**-monceau)

Temps $O(n \log n)$ dans le pire des cas, sans espace additionnelle !

quicksort : $O(n^2)$ dans le pire des cas

mergesort : $O(n \log n)$ dans le pire des cas mais utilise un espace auxiliaire de taille n

Files de priorité

Autres implantations existent (nécessaires pour un **merge** efficace) :
binomial heap, skew heap, Fibonacci heap

	binaire (pire)	binomial (pire)	skew (amorti)	Fibonacci (amorti)
deleteMin	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
insert	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$
merge	$O(n)$	$O(\log n)$	$O(1)$	$O(1)$
decreaseKey	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$

opération **decreaseKey** : change la priorité d'un élément — dans un tas binaire on peut le faire à l'aide de **NAGER**

decreaseKey est important dans quelques algorithmes fondamentaux sur des graphes (plus court chemin, arbre couvrant minimal)

Applications

Simulations d'événements discrets

Algorithme A^* : arriver à sa destination malgré des obstacles (labyrinthe)