

9 Tas binaire

9.1 Type abstrait : file de priorité

Type abstrait d'une **file de priorité** (*priority queue*) : objets = ensembles d'objets avec priorités comparables (abstraction : nombres naturels)

$W_{(en)}$

Opérations — min-tas

- ★ `insert(x)` : insertion de l'élément x (avec une priorité)
- ★ `deleteMin()` : suppression de l'élément minimal

Opérations parfois supportées : `merge` (fusion de files), `findMin` (retourne, mais ne supprime pas, l'élément minimal), `delete` (suppression d'un élément), `decreaseKey` (change la priorité d'un élément).

max-tas. définition équivalente avec `deleteMax` et `findMax` — mais non pas max et min en même temps

Clients. simulations d'événements discrets (p.e., collisions), systèmes d'exploitation (interruptions, ordonnancement en temps partagé), algorithmes sur graphes, recherche opérationnelle (plus courts chemins, arbre couvrant), statistiques : maintenir l'ensemble des m meilleurs éléments.

```

MEILLEUR-EMTS( $T[0..n - 1], m$ )
  // (choisit les  $m$  plus grand éléments en utilisant un tas de  $m + 1$  éléments au plus)
B1 initialiser min-tas  $H$ 
B2 for  $i \leftarrow 0, \dots, n - 1$  do  $H.insert(T[i])$ ; if  $i \geq m$  then  $H.deleteMin()$ 
B3 return les éléments de  $H$ 
    
```

Implantations élémentaires. On peut implanter une file de priorité par une liste chaînée ou tableau, soit en une approche paresseuse (insertion n'importe où, suppression parcourt la liste), soit en une approche impatiente (liste ordonnée selon priorités, suppression à la tête). Dans les exemples ci-dessous, on utilise une sentinelle à la tête. L'approche impatiente utilise une liste doublement chaînée, et considère la liste comme structure exogène (c'est info qui est décalé, et non pas le nœud lui-même).

Approche **paresseuse** (*lazy*) avec liste non-triée

```

Opération insert(x) // en  $O(1)$ 
I1 insertFirst(x) // (insertion à la tête)

Opération deleteMin() // en  $\Theta(n)$  au pire
D1  $N \leftarrow head; M \leftarrow null; v \leftarrow \infty$ 
D2 while  $N.next \neq null$ 
D3    $w \leftarrow N.next.info$ 
D4   if  $w < v$ 
D5     then  $v \leftarrow w; M \leftarrow N$  // ( $M$  contient min)
D6    $N \leftarrow N.next$ 
D7 deleteAfter(M) // (suppression après nœud  $M$ )
D8 return  $v$ 
    
```

Approche **impatiente** (*eager*) avec liste triée + sentinelle

```

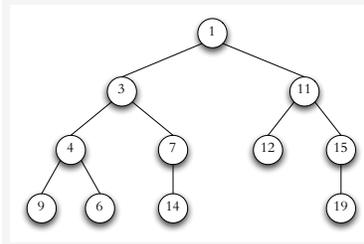
Opération insert(x) // en  $\Theta(n)$  au pire
I1 insertLast(x)
I2  $N \leftarrow queue; P \leftarrow N.precedent$ 
I3 while  $P \neq head$  et  $P.info > x$ 
I4    $N.info \leftarrow P.info$  // (décalage vers la fin)
I5    $N \leftarrow P; P \leftarrow N.precedent$ 
I6  $N.info \leftarrow x$ 

Opération deleteMin() // en  $O(1)$ 
D1  $v \leftarrow head.next.info$ 
D2 deleteAfter(head) // (sentinelle à la tête)
D3 return  $v$ 
    
```

9.2 Ordre de tas

$W_{(fr)}$

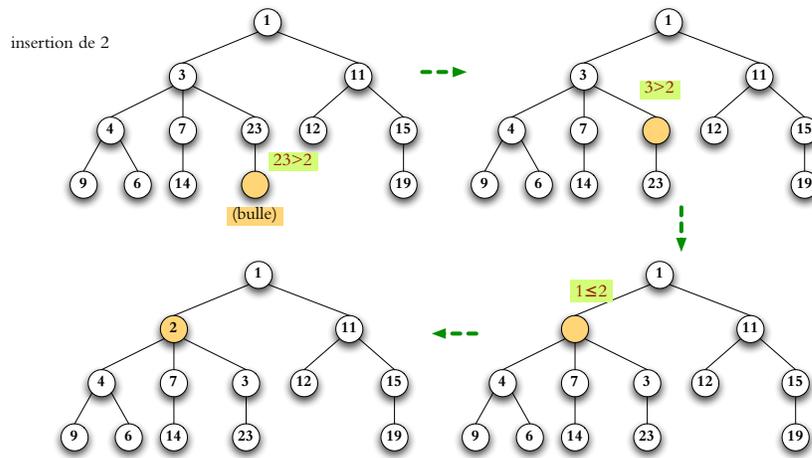
On peut améliorer l'approche impatiente avec des «branchements» dans la liste chaînée (exemple : hiérarchie militaire — le vieux général est remplacé par son meilleur lieutenant-général, qui est remplacé par son major-général, etc.)



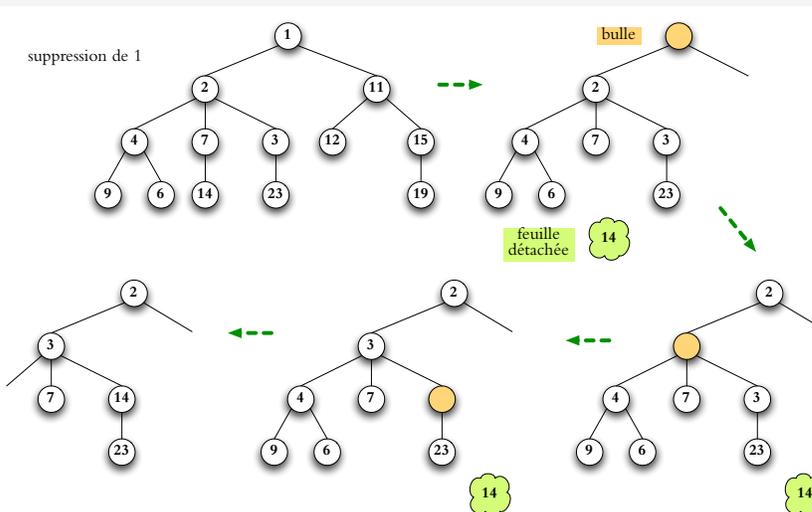
Pour implanter la file de priorité, on se sert d'une arborescence dont les nœuds sont dans l'**ordre de tas** : si x n'est pas la racine, alors

$$x.\text{parent.priorite} \leq x.\text{priorite}.$$

Opération findMin en $O(1)$: c'est à la racine.



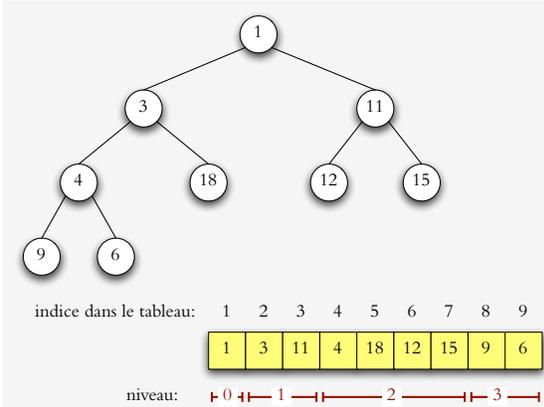
swim (nager) : ajouter une feuille vide («bulle») + monter la bulle vers la racine jusqu'à ce qu'on trouve la place pour la nouvelle valeur
Temps : proportionnel à la profondeur



sink (couler) : remplacer le nœud par une «bulle», enlever une feuille et pousser la bulle vers les feuilles jusqu'à ce qu'on trouve la place pour la nouvelle valeur.
Temps : proportionnel à la hauteur

9.3 Tas binaire

W_(fr)



On choisit la structure de l'arbre pour sink/swim : le meilleur choix est un arbre binaire complet (Théorème 8.3).

On utilise un tableau $H[1..n]$ pour l'encodage compact de l'arbre binaire complet. Parent de nœud i est à $\lfloor i/2 \rfloor$, enfant gauche est à $2i$, enfant droit est à $2i + 1$.

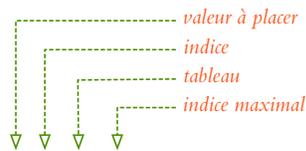
Tableau en ordre de tas : $H[i] \leq H[2i], H[2i + 1]$.

```

INSERT(v, H, n) // en O(lg n)
I1 SWIM(v, n + 1, H) // // tas binaire dans H[1..n]
    SWIM(v, i, H) // placement de v en H[1..i]
N1 p ← ⌊i/2⌋
N2 while p ≠ 0 et H[p] > v do H[i] ← H[p]; i ← p; p ← ⌊i/2⌋
N3 H[i] ← v
    
```

```

DELETEMIN(H, n) // en O(lg n)
D1 r ← H[1] // tas dans H[1..n]
D2 v ← H[n]; H[n] ← null; if n > 1 then SINK(v, 1, H, n - 1)
D3 retourner r
    
```



```

C1 SINK(v, i, H, n) // placement de v en H[i..n]
C2 c ← MINCHILD(i, H, n)
C3 while c ≠ 0 et H[c] < v do H[i] ← H[c]; i ← c; c ← MINCHILD(i, H, n)
C4 H[i] ← v

    MINCHILD(i, H, n) // retourne l'enfant avec H minimale ou 0 si i est une feuille
M1 j ← 0
M2 if 2i ≤ n then j ← 2i
M3 if (2i + 1 ≤ n) && (H[2i + 1] < H[j]) then j ← 2i + 1
M4 return j
    
```

Efficacité. La hauteur h d'un arbre binaire complet avec n nœuds internes est $h = 1 + \lfloor \lg n \rfloor$; les opérations s'exécutent en h itérations au plus.

- ★ deleteMin : $O(\lg n)$
- ★ insert : $O(\lg n)$
- ★ findMin : $O(1)$

9.4 Heapisation

Opération **heapify**(A) met les éléments du tableau $A[1..n]$ dans l'ordre de tas. Triviale ?

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

\Rightarrow prend $\Theta(n \log n)$ au pire

Meilleure solution :

```
HEAPIFY( $A$ ) // tableau arbitraire  $A[1..n]$ 
for  $i \leftarrow \lceil n/2 \rceil, \dots, 1$  do SINK( $A[i], i, A, n$ )
```

Théorème 9.1. HEAPIFY met les éléments dans l'ordre de tas en temps $O(n)$.

Démonstration. SINK prend $O(h)$ temps où h est la hauteur du nœud qui correspond à l'indice i dans la représentation arborescente du tas. Il y a $\leq \lceil n/2^h \rceil$ nœuds internes avec hauteur h . Donc le temps de calcul est borné par $T(n) \leq \sum_{h=1}^{1+\lceil \lg n \rceil} \lceil \frac{n}{2^h} \rceil \times O(h) \leq O(n) \cdot \sum_{h=1}^{\infty} h \cdot (1/2)^h = O(n)$. ■

9.5 Tri par tas

On peut utiliser une file de priorité pour tri : mettre tous éléments dans la file (heapify), et retirer l'un après l'autre dans l'ordre croissant. Avec un tas binaire, on peut faire le tri en place. Après heapify, on maintient l'ordre de tas dans le préfixe $A[1..i]$ en une boucle $i \leftarrow n, n-1, \dots, 2$. Le suffixe $A[i..n]$ est toujours trié en ordre décroissant. À chaque itération, après avoir échangé $A[1] \leftrightarrow A[i]$, on rétablit l'ordre de tas en $A[1..i-1]$ pour la prochaine itération.

$W_{(fr)}$

```
HEAPSORT( $A[1..n]$ ) // tableau non-trié
H1 heapify( $A$ )
H2 for  $i \leftarrow n, \dots, 2$  do
H3    $v \leftarrow A[i]; A[i] \leftarrow A[1]$  // échange  $A[i] \leftrightarrow A[1]$ 
H4   SINK( $v, 1, A, i-1$ ) // rétablissement de l'ordre de tas en  $A[1..i-1]$ 
```

Ordre des éléments. On finira avec l'ordre décroissant — pour l'ordre croissant :

- ★ renverser le résultat en place : $i \leftarrow 1; j \leftarrow n$; **while** ($i < j$) { $A[i] \leftrightarrow A[j]; i \leftarrow i+1; j \leftarrow j-1$ }
- ★ ou implanter l'ordre de max-tas dans le code

Temps de calcul et mémoire. Tri par tas finit en $O(n \log n)$ temps. C'est un **tri en place** parce qu'il utilise seulement $O(1)$ espace additionnelle (quelques variables à part du tableau à trier).

9.6 Tas d -aire

Au lieu d'un arbre binaire, on peut baser le tas sur un arbre complet avec arité arbitraire $d \geq 2$. On encode l'arbre dans le tableau $A[1..n]$. Parent de l'indice i est $\lceil (i-1)/d \rceil$; les enfants sont à $d(i-1) + 2..di + 1$. Ordre de tas :

$W_{(en)}$

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

- ★ 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 : $O(1)$
- ★ SWIM et SINK : $O(\log_d n)$ et $O(d \log_d n)$

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