

7 Tas binaire

7.1 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 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, 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

```

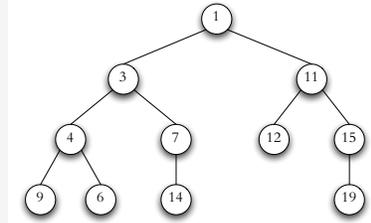
Opération insert(x) // en  $\Theta(n)$  au pire
I1 insertLast(x)
I2  $N \leftarrow queue; P \leftarrow N.precedent$ 
I3 while  $P \neq head$  et  $P.info > x$ 
I4  $N.info \leftarrow P.info$  // (décalage vers la fin)
I5  $N \leftarrow P; P \leftarrow N.precedent$ 
I6  $N.info \leftarrow x$ 

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

```

7.2 Ordre de tas

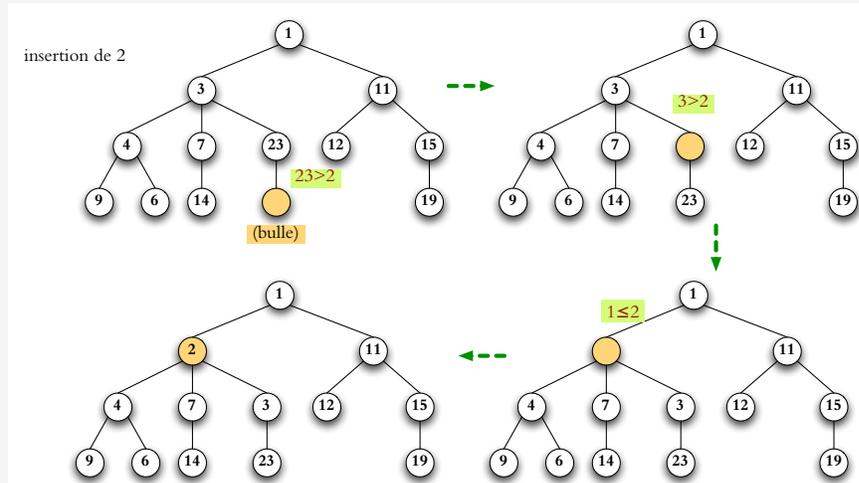
On peut améliorer l'approche impatiente avec des «branchements» dans la liste chaînée (exemple : hiérarchie militaire — le vieux général est remplacé par son meilleur lieutenant-général, qui est remplacé par son major-général, etc.)



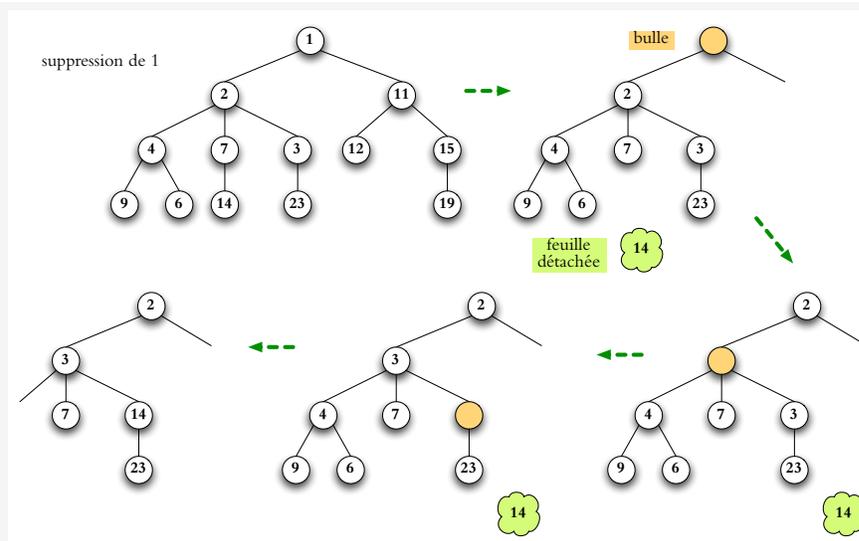
Pour implanter la file de priorité, on se sert d'une arborescence dont les nœuds sont dans l'**ordre de tas** : si x n'est pas la racine, alors

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

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

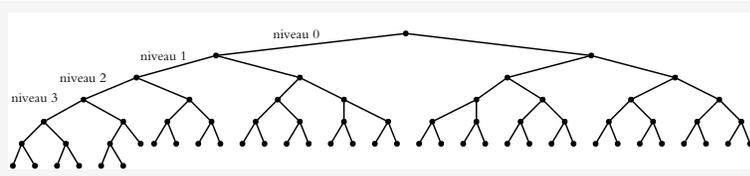


swim (nager) : ajouter une feuille vide («bulle») + monter la bulle vers la racine jusqu'à ce qu'on trouve la place pour la nouvelle valeur
Temps : proportionnel à la profondeur



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

7.3 Arbre binaire complet



Un **arbre binaire complet** (*complete binary tree*) de hauteur h : il y a 2^i nœuds internes à chaque niveau $i = 0, \dots, h - 1$. On «remplit» les niveaux de gauche à droit.

Théorème 7.1. Un arbre binaire à n nœuds externes contient $(n - 1)$ nœuds internes.

Théorème 7.2. La hauteur h d'un arbre binaire à n nœuds externes est bornée par

$$\lceil \lg(n) \rceil \leq h \leq n - 1. \quad (7.1)$$

Démonstration. Un arbre de hauteur $h = 0$ ne contient qu'un seul nœud externe, et les bornes sont correctes. Pour $h > 0$, on définit m_k comme le nombre de nœuds internes au niveau $k = 0, 1, 2, \dots, h - 1$ (il n'y a pas de nœud interne au niveau h). Par Théorème 7.1, on a $n - 1 = \sum_{k=0}^{h-1} m_k$. Comme $m_k \geq 1$ pour tout $k = 0, \dots, h - 1$, on a que $n - 1 \geq \sum_{k=0}^{h-1} 1 = h$. Pour une borne supérieure, on utilise que $m_0 = 1$, et que $m_k \leq 2m_{k-1}$ pour tout $k > 0$. En conséquence, $n - 1 \leq \sum_{k=0}^{h-1} 2^k = 2^h - 1$, d'où $h \geq \lg n$. La preuve montre aussi les arbres extrêmes : une chaîne de nœuds pour $h = n - 1$, et un arbre binaire complet. ■

Exercice 7.1. Soit n_k le nombre de nœuds avec k enfants dans un arbre ordonné ($k = 0, 1, 2, \dots$). Démontrer que

$$n_0 = 1 + n_2 + 2 \cdot n_3 + 3 \cdot n_4 + \dots + (d - 1) \cdot n_d,$$

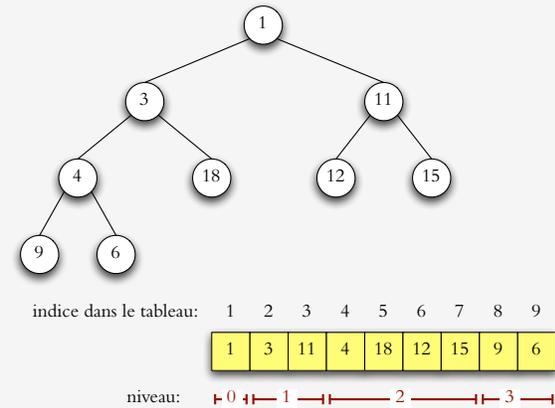
où d est le degré maximal dans l'arbre. Notez que Théorème 7.1 est un cas spécial de cette égalité avec $d = 2$. **Indice** : compter les nœuds deux fois — une fois selon le nombre d'enfants, et une fois selon le parent.

Exercice 7.2. Démontrer que pour la hauteur h d'un arbre binaire à n nœuds internes,

$$1 + \lceil \lg(n) \rceil \leq h \leq n. \quad (7.2)$$

7.4 Tas binaire

W^(fr)



On choisit la structure de l'arbre pour sink/swim : le meilleur choix est un arbre binaire complet (Théorème 7.2).

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  $p \leftarrow \lfloor i/2 \rfloor$ 
N2 while  $p \neq 0$  et  $H[p] > v$  do  $H[i] \leftarrow H[p]; i \leftarrow p; p \leftarrow \lfloor i/2 \rfloor$ 
N3  $H[i] \leftarrow v$ 
    
```

```

DELETEMIN( $H, n$ ) // en  $O(\lg n)$ 
D1  $r \leftarrow H[1]$  // tas dans  $H[1..n]$ 
D2  $v \leftarrow H[n]; H[n] \leftarrow \text{null};$  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  $c \leftarrow \text{MINCHILD}(i, H, n)$ 
C3 while  $c \neq 0$  et  $H[c] < v$  do  $H[i] \leftarrow H[c]; i \leftarrow c; c \leftarrow \text{MINCHILD}(i, H, n)$ 
C4  $H[i] \leftarrow v$ 

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