

## 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<sub>(en)</sub>

#### 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 pour  $i \leftarrow 0, \dots, n-1$  faire  $H.insert(T[i])$ ; si  $i \geq m$  alors  $H.deleteMin()$ 
B3 retourner 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 l'exemples ci-dessous, on utilise une liste chaînée, avec une sentinelle à la tête. L'approche impatiente utilise une liste doublement chaînée, et considère la liste comme structure exogène<sup>1</sup>

Approche **paresseuse** (*lazy*) avec liste chaînée

```

insert( $x$ )
I1 insertFirst( $x$ ) // (insertion à la tête)
deleteMin() // (parcourt la liste)
D1  $N \leftarrow tete; M \leftarrow null; v \leftarrow \infty$ 
D2 tandis que  $N.prochain \neq null$ 
D3  $w \leftarrow N.prochain.info$ 
D4 si  $w < v$  then  $v \leftarrow w; M \leftarrow N$ 
D5  $N \leftarrow N.prochain$ 
D6 delete( $M$ ) // (suppression après nœud  $M$ )
D7 retourner  $v$ 

```

Approche **impatiente** (*eager*) avec liste chaînée

```

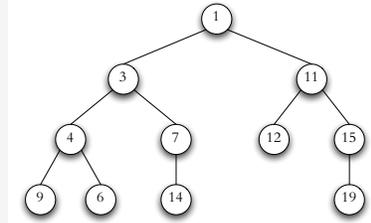
insert( $x$ )
I1 insertLast( $x$ )
I2  $N \leftarrow queue; P \leftarrow N.precedent$ 
I3 tandis que  $P \neq tete$  et  $P.info > x$ 
I4  $N.info \leftarrow P.info$  // (décalage vers la queue)
I5  $N \leftarrow P; P \leftarrow N.precedent$ 
I6  $N.info \leftarrow x$ 
deleteMin()
D1  $v \leftarrow tete.prochain.info$ 
D2 delete( $tete$ ); retourner  $v$  // (suppression à la tête)

```

<sup>1</sup>Alternativement, on pourrait utiliser une solution endogène comme la procédure `INSERT( $k, x$ )` en §1.5.

## 7.2 Ordre de tas

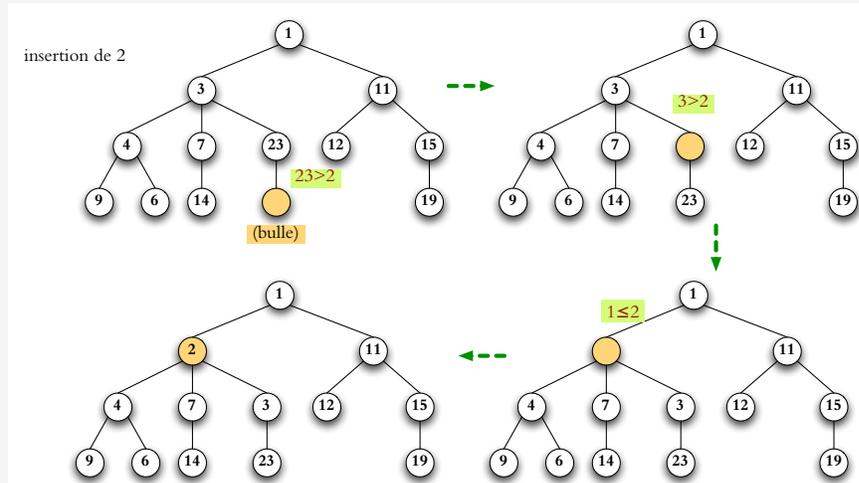
On peut améliorer l'approche impatiente avec des «branchements» dans la liste chaînée (exemple : hiérarchie militaire — le vieux général est remplacé par son meilleur lieutenant-général, qui est remplacé par son major-général, etc.)



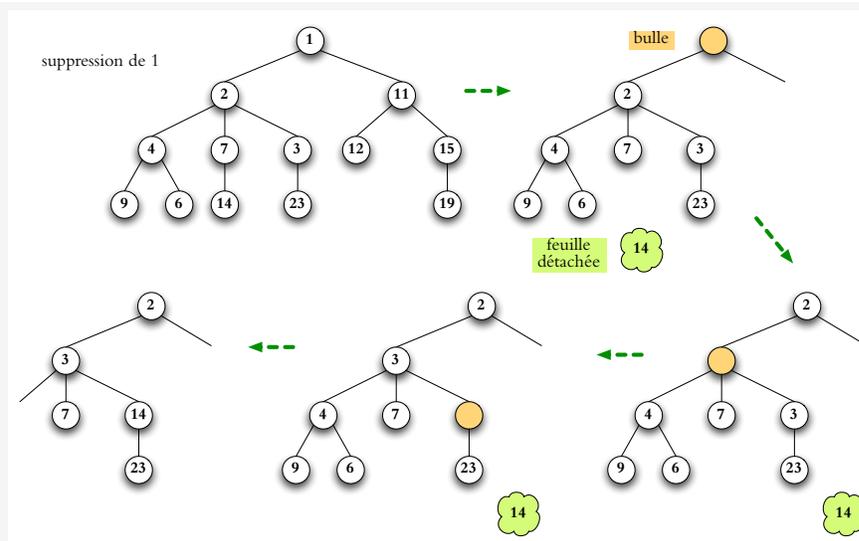
Pour implanter la file de priorité, on se sert d'une arborescence dont les nœuds sont dans l'**ordre de tas** : si  $x$  n'est pas la racine, alors

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

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



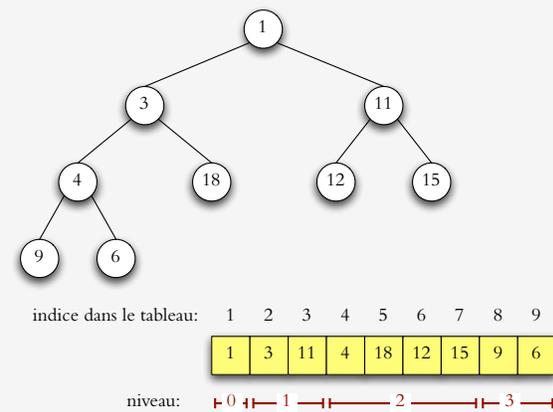
**swim (nager)** : ajouter une feuille vide («bulle») + monter la bulle vers la racine jusqu'à ce qu'on trouve la place pour la nouvelle valeur



**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.

### 7.3 Tas binaire

W<sup>(fr)</sup>



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

Les  $n$  éléments sont placés dans un tableau  $H[1..n]$  qui encode un arbre binaire complet pour  $n$  nœuds. 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( $v, H, n$ ) // tas binaire dans  $H[1..n]$ 
I1 NAGER( $v, n + 1, H$ )
   NAGER( $v, i, H$ ) // rétablit l'ordre de tas dans  $H[1..i]$  en ascendant et place  $v$ 
N1  $p \leftarrow \lfloor i/2 \rfloor$ 
N2 tandis que  $p \neq 0$  et  $H[p] > v$  faire  $H[i] \leftarrow H[p]; i \leftarrow p; p \leftarrow \lfloor i/2 \rfloor$ 
N3  $H[i] \leftarrow v$ 

```

```

DELETEMIN( $H, n$ ) // tas dans  $H[1..n]$ 
D1  $r \leftarrow H[1]$ 
D2  $v \leftarrow H[n]; H[n] \leftarrow \text{null};$  si  $n > 1$  alors COULER( $v, 1, H, n - 1$ )
D3 retourner  $r$ 
   COULER( $v, i, H, n$ ) // rétablit l'ordre de tas dans  $H[i..n]$  en descendant et place  $v$ 
C1  $c \leftarrow \text{MINCHILD}(i, H, n)$ 
C2 tandis que  $c \neq 0$  et  $H[c] < v$  faire  $H[i] \leftarrow H[c]; i \leftarrow c; c \leftarrow \text{MINCHILD}(i, H, n)$ 
C3  $H[i] \leftarrow v$ 
   MINCHILD( $i, H, n$ ) // retourne l'enfant avec clé minimale ou 0 si  $i$  est une feuille
M1  $j \leftarrow 0$ 
M2 si  $2i \leq n$  alors  $j \leftarrow 2i$ 
M3 si  $2i + 1 \leq n$  et  $H[2i + 1] < H[j]$  alors  $j \leftarrow 2i + 1$ 
M4 retourner  $j$ 

```

**Efficacité.** La hauteur  $h$  d'un arbre binaire complet avec  $n$  nœuds est  $h = \lceil \lg n \rceil$ ; les opérations s'exécutent en  $O(h)$  temps.

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

## 7.4 Tas $d$ -aire

Tas  $d$ -aire : on utilise un arbre complet  $d$ -aire avec une arité  $d \geq 2$ . L'implantation utilise un tableau  $A[1..n]$  : Parent de l'indice  $i$  est  $\lceil (i-1)/d \rceil$ , enfants sont à  $d(i-1) + 2..di + 1$ . Ordre de tas : W<sub>(en)</sub>

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

- ★ 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.

## 7.5 Heapisation

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

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

⇒ prend  $O(n \log_d n)$

Meilleure solution :

```
HEAPIFY( $A$ ) // vecteur arbitraire  $A[1..n]$ 
pour  $i \leftarrow n, \dots, 1$  faire COULER( $A[i], i, A, n$ )
```

**Théorème 7.1.** HEAPIFY met les éléments dans l'ordre de tas en temps  $O(n)$  indépendamment de l'arité.

*Démonstration.* Dans un tas  $d$ -aire, si  $i$  est à hauteur  $j$  ( $j$  nœuds au plus jusqu'à une feuille), alors il prend  $O(j \cdot d)$  de faire COULER( $\cdot, i, \cdot$ ). Il y a  $\leq n/d^j$  nœuds à hauteur  $j$ . Donc le temps est de

$$\sum_{j=1}^{\lceil 1+\lg n \rceil} \frac{n}{d^j} O(dj) = O(n) \cdot \sum_{j=0}^{\infty} \frac{j+1}{d^j} = O(n).$$

■

## 7.6 Autres implantations de files de priorité

Il existe d'autres implantations (nécessaires pour un merge efficace) : binomial heap, skew heap, Fibonacci heap. L'opération decreaseKey est important dans quelques algorithmes fondamentaux sur des graphes (plus court chemin, arbre couvrant minimal).

	liste triée	liste non-triée	binaire	$d$ -aire	binomial	skew (amorti)	Fibonacci (amorti)
deleteMin	$O(1)$	$O(n)$	$O(\log n)$	$O(d \log n / \log d)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
insert	$O(n)$	$O(1)$	$O(\log n)$	$O(\log n / \log d)$	$O(\log n)$	$O(1)$	$O(1)$
merge	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$	$O(1)$
decreaseKey	$O(n)$	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$