9. File de priorité

LA **FILE DE PRIORITÉ**[1] (*priority queue*) est un type abstrait qui formalise la notion d'ordre entre les éléments d'une collection à l'aide de l'opération principale d'accès appellée deleteMin. L'opération enlève le plus petit élément dans une collection où «le plus petit» est défini mathématiquement par un ordre total[2], et en Java par des objets comparables[3].

[1] W(en):*priority queue*

[2] W(fr):relation d'ordre total

[3] Tutoriel : (*Object ordering*)

9.1 Type abstrait : file de priorité

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 (Fig. 1).

```
Entrée: fichier/tableau de valeurs
        x_0, x_1, x_2, x_{n-1} ; nombre de
        meilleurs m < n
initialiser min-tas PQ
for i ← 0, ..., n − 1 do
    PQ.insert(x_i)
    if |PQ| > m then
        PQ.deleteMin()
    // PQ contient les plus grands
       parmi x_0, x_1, ..., x_i
end
return les éléments de PQ
```

FIG. 1: Choisir les m plus grands éléments parmi n. L'algorithme maintient les m plus grands éléments dans une file de priorité de taille $\leq (m+1)$ à tout temps. Il s'agit donc d'un algorithme «en ligne« (*online algorithm*) qu'on peut facilement adapter au traitement de n'importe quel flux de données (*data stream*) même si la taille n est inconnue au début.

Implémentations élémentaires. Si on veut implanter une file de priorité par une liste chaînée ou par un tableau, on peut choisir entre

- ⋆ l'**approche paresseuse** avec liste non-ordonnée (Figure 2), ou
- ⋆ l'**approche impatiente** : avec liste ordonnée (Figure 3)

En tout cas, cela mène à un temps de $\Theta(n)$ pour une opération principale et est donc utile seulement quand n est très petit.

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

```
   insert(x)                                        // en O(1)
I1 N ← new nœud; N.info ← x; head.insertNext(N)

   deleteMin()                                      // parcours en Θ(n)
D1 N ← head; M ← null; v ← ∞
D2 while N.next ≠ null
D3     w ← N.next.info ; if w < v then v ← w; M ← N
D4     N ← N.next
D5 M.deleteNext() //nœud M.next contient l'élément minimal v
D6 return v
```

FIG. 2: Implémentation de file de priorité basée sur une liste chaînée avec sentinelle à la tête.

Approche impatiente (*eager*) : liste triée

Opération insert(x) // *en* $\Theta(n)$ *au pire*

I1 $P \leftarrow$ head.precedent; $N \leftarrow$ new nœud; P.insertNext(N)

I2 **while** $P \neq$ head et P.info $> x$ **do**

I3 N.info $\leftarrow P$.info; $N \leftarrow P$; $P \leftarrow N$.precedent

I4 N.info $\leftarrow x$

Opération deleteMin$()$ // *en* $O(1)$

D1 $v \leftarrow$ head.next.info; head.deleteNext$()$

D2 **return** v

FIG. 3: Implémentation de file de priorité basée sur une liste doublement chaînée avec sentinelle à la tête. Ici, on considère la liste comme structure exogène : c'est info qui est décalé, et non pas le nœud lui-même. Dans l'approche paresseuse on manipule plutôt les nœuds sur la liste.

9.2 *Ordre de tas*

Afin d'améliorer l'approche impatiente, on se sert d'une arborescence dont les nœuds sont dans l'**ordre de tas**[4] : si x n'est pas la racine, alors

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

L'opération findMin ainsi prend un temps constant car le minimum est toujours à la racine.

[4] W(fr):ordre de tas

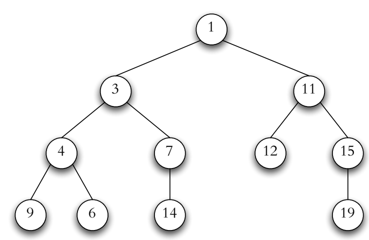

FIG. 4: Ordre de tas dans un arbre : la clé du parent est inférieure ou égàle à celle de l'enfant.

Insertion L'insertion se fait par la technique algortihmique de **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 (où le parent a une clé inférieure). Temps : proportionnel à la profondeur.

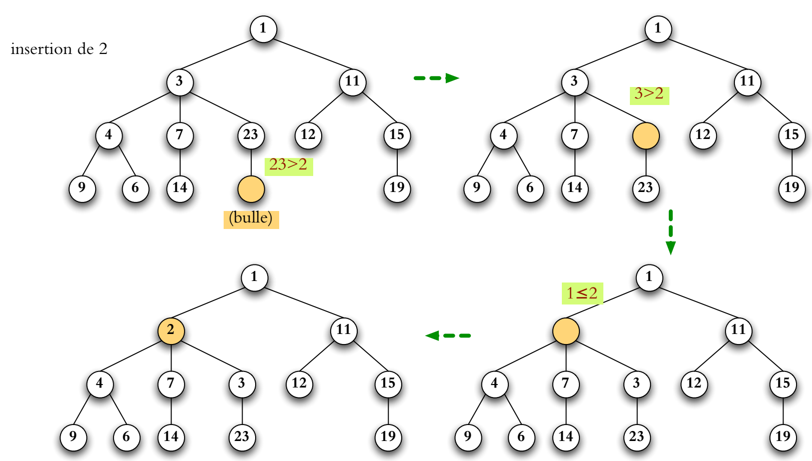

FIG. 5: Insertion d'une nouvelle feuille dans un arbre ordonné en tas.

Suppression La suppression du minimum s'implémente par la technique de **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.

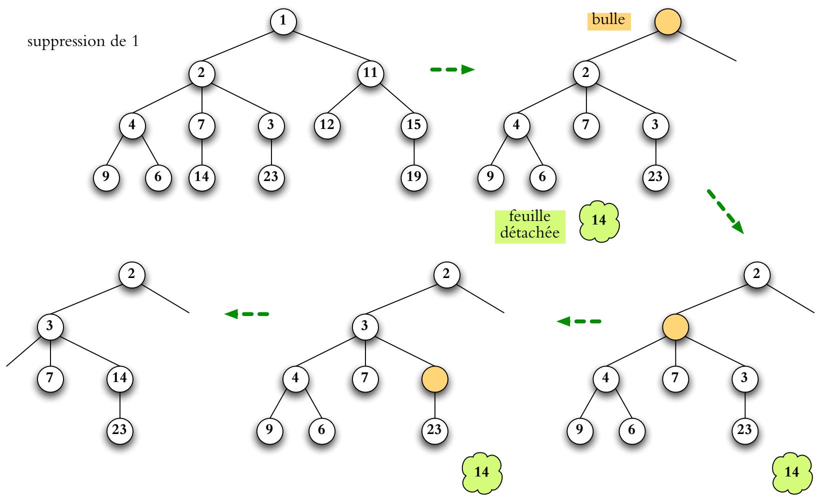

FIG. 6: Suppression de feuille dans un arbre ordonné en tas.

9.3 *Tas binaire*

On a la liberté de choisir la structure de l'arbre pour sink/swim ! Hauteur minimale pour n éléments atteinte par **arbre binaire complet** : il y a 2^i nœuds à chaque niveau $i = 0, \ldots, h-1$; les niveaux «remplis» de gauche à droit. Il n'est pas nécessaire alors de stocker des pointeurs pointeurs parent, left, right, mais on se sert plutôt l'indexage des nœuds défini par le parcours par largeur. Parent de nœud i est à $\lfloor i/2 \rfloor$, enfant gauche est à $2i$, enfant droit est à $2i+1$ (v. Figure 7). Tableau en ordre de tas $H[1..n]$:

$$
\begin{aligned}
H[i] &\le H[2i] & \{0 < i \le n/2\} \\
H[i] &\le H[2i+1] & \{0 < i \le (n-1)/2\}
\end{aligned}
$$

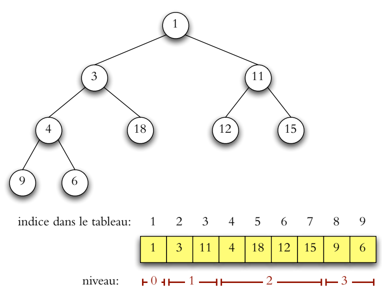

FIG. 7: Tas binaire : arbre binaire complet encodé dans un tableau.

Avec l'encodage du tas binaire, les procédures de nager et couler se font par des boucles simples sur les indices : v. Figure 8.

INSERT(v, H, n) // en $O(\lg n)$
I1 swim$(v, n+1, H)$ // // tas binaire dans $H[1..n]$

DELETEMIN(H, n) // en $O(\lg n)$
D1 $r \leftarrow H[1]$ // tas dans $H[1..n]$
D2 $v \leftarrow H[n]$; $H[n] \leftarrow$ null ; **if** $n > 1$ **then** sink$(v, 1, H, n-1)$
D3 **return** r

swim

Entrée: tas $H[1..]$, indice i, valeur v — placement de v dans $H[1..i]$
$p \leftarrow \lfloor i/2 \rfloor$
while $p \neq 0$ && $H[p] > v$ **do**
  $H[i] \leftarrow H[p]$; $i \leftarrow p$; $p \leftarrow \lfloor i/2 \rfloor$
end
$H[i] \leftarrow v$

sink

Entrée: tas $H[1..]$, indice i, valeur v, taille n — placement de v dans $H[i..n]$
$c \leftarrow \textsc{minChild}(i, H, n)$
while $c \neq 0$ && $H[c] < v$ **do**
  $H[i] \leftarrow H[c]$; $i \leftarrow c$; $c \leftarrow \textsc{minChild}(i, H, n)$
end
$H[i] \leftarrow v$

MINCHILD (indice de l'enfant minimal)

Entrée: tas $H[1..]$, indice i
if $2i > n$ **then** $j \leftarrow 0$
else if $(2i + 1 \leq n)$ && $(H[2i+1] < H[2i])$ **then** $j \leftarrow 2i + 1$
else $j \leftarrow 2i$
return j *// retourne $2i$ ou $2i + 1$ ou 0*

FIG. 8: Procédures de nager et couler. swim(i, v) place la valeur v sur le chemin de l'indice i à la racine. sink(i, v) place v sur un chemine de l'indice i vers les feuilles.

Tas d-aire

Au lieu d'un arbre binaire, on peut baser le tas sur un arbre complet avec arité arbitraire $d \geq 2$ et ainsi définir le tas d-aire[5]. 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 :

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

- ⋆ swim contre sink : 1 contre d comparaisons par niveau
- ⋆ deleteMin : $\sim (d \log_d n)$ comparaisons au pire
- ⋆ insert : $\sim (\log_d n)$ comparaisons au pire
- ⋆ findMin : $O(1)$

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

[5] W(en): *d-ary heap*

9.4 Tri par tas

Heapisation. On veut une procédure heapify (A) qui met les éléments du tableau $A[1..n]$ dans l'ordre de tas. Triviale ? **for** $i \leftarrow 1, \ldots, n$ **do** swim$(A[i], i, A)$ prend $\Theta(n \log n)$ au pire. Une meilleure solution : **for** $i \leftarrow \lfloor n/2 \rfloor, \ldots, 1$ **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

$$T(n) \leq \sum_{h=1}^{1+\lfloor \lg n \rfloor} \lceil \frac{n}{2^h} \rceil \times O(h) \leq O(n) \cdot \sum_{h=1}^{\infty} h \cdot (1/2)^h = O(n).$$

■

Tri par tas. La file de priorité permet donc de trier une collection d'éléments : insertion de n éléments + appeler n fois deleteMin. Avec un tas binaire, on peut faire le tri en place : il s'agit de l'algorithme du **tri par tas**[6] (*heapsort*). Après heapify, on maintient l'ordre de tas dans le préfixe $A[1..i]$ en une boucle $i \leftarrow n, n-1, \ldots 2$. 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.

[6] W(fr):tri par tas

```
   HEAPSORT(A[1..n])                              // tableau non-trié
H1 heapify(A)
H2 for i ← n,...2 do
H3     v ← A[i] ; A[i] ← A[1]                     // échange A[i] ↔ A[1]
H4     sink(v, 1, A, i − 1)                       // tas raccourci à 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).