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 W(en) (abstraction : nombres naturels)

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 ← 0, . . . , n − 1 do H.insert(T[i]) ; if i ≥ 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

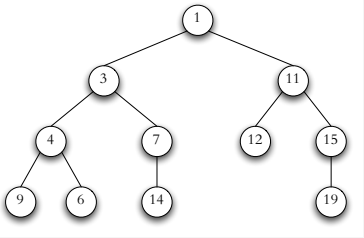
```
    Opération insert(x)
I1  insertFirst(x)                    // (insertion à la tête)
    Opération deleteMin()
D1  N ← head; M ← null; v ← ∞
D2  while N.next ≠ null
D3      w ← N.next.info
D4      if w < v then v ← w; M ← N
D5      N ← N.next
D6  deleteAfter(M)          //(suppression après nœud M)
D7  return v
```

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

```
    Opération insert(x)
I1  insertLast(x)
I2  N ← queue; P ← N.precedent
I3  while P ≠ head et P.info > x
I4      N.info ← P.info // (décalage vers la fin)
I5      N ← P; P ← N.precedent
I6  N.info ← x
    Opération deleteMin()     // (suppression à la tête)
D1  v ← head.next.info
D2  deleteAfter(head) ; return v
```

7.2 Ordre de tas

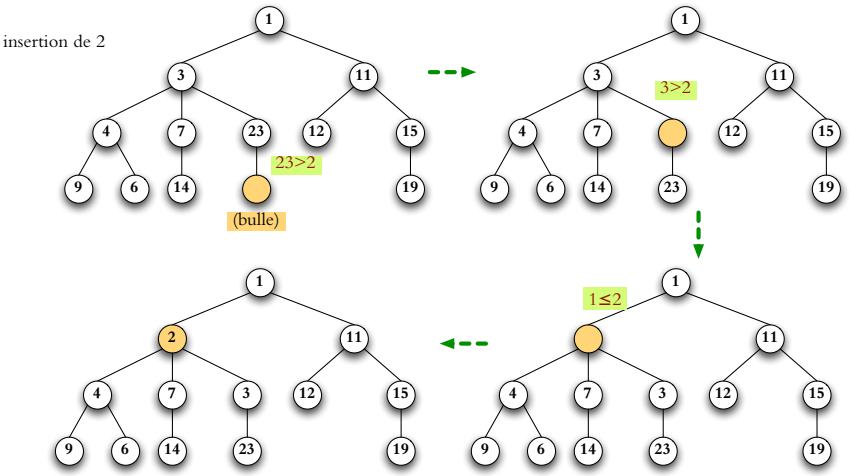
On peut améliorer l'approche impatiente avec des «branchements» dans la liste chaînée (exemple : hiérarchie militaire — le vieux général est remplacé par son meilleur lieutenant-général, qui est remplacé par son major-général, etc.)



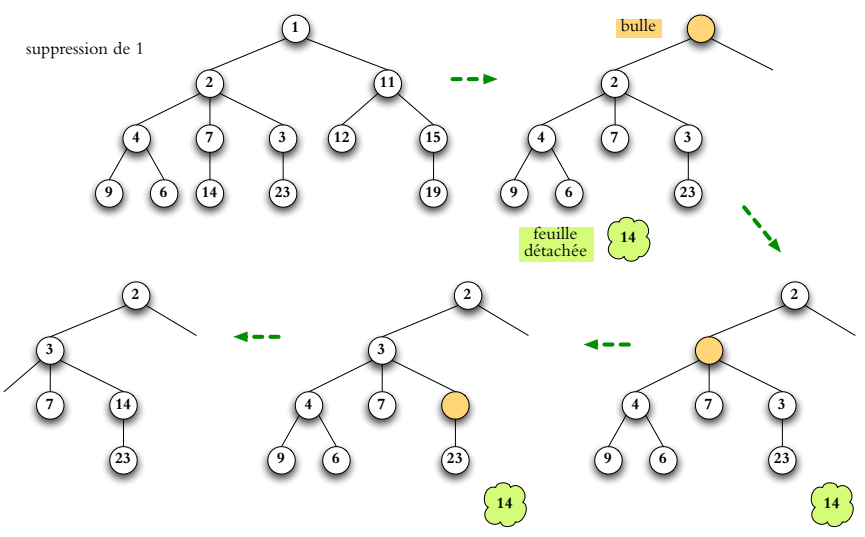
Pour implanter la file de priorité, on se sert d'une arborescence dont les nœuds sont dans l'**ordre de tas** : si x n'est pas la racine, alors

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

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



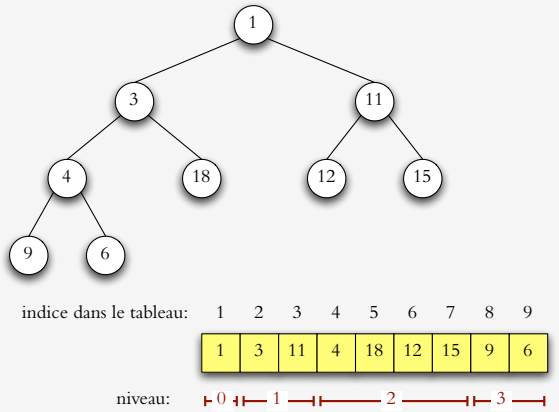
swim (nager) : ajouter une feuille vide («bulle») + monter la bulle vers la racine jusqu'à ce qu'on trouve la place pour la nouvelle valeur



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(fr)



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  SWIM(v, n+1, H)

    SWIM(v, i, H) // rétablit l'ordre de tas dans H[1..i] en ascendant et trouve un placement pour v
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) // tas dans H[1..n]
D1  r ← H[1]
D2  v ← H[n] ; H[n] ← null ; if n > 1 then SINK(v, 1, H, n − 1)
D3  retourner r

    SINK(v, i, H, n) // rétablit l'ordre de tas dans H[i..n] en descendant et trouve un placement pour v
C1  c ← MINCHILD(i, H, n)
C2  while c ≠ 0 et H[c] < v do H[i] ← H[c] ; i ← c ; c ← MINCHILD(i, H, n)
C3  H[i] ← v

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

Efficacité. La hauteur h d'un arbre binaire complet avec n nœuds est $h = \lfloor \lg n \rfloor$; 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(en)

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

- ⋆ 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 $O(d \log_d n)$

⇒ 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 ?

$$H \leftarrow \emptyset \,;\, \textbf{for } i \leftarrow 1, \ldots, n \textbf{ do } \textsc{insert}(A[i], H, i) \,;\, A \leftarrow H$$

⇒ prend $O(n \log_d n)$

Meilleure solution :

```
HEAPIFY(A) // vecteur arbitraire A[1..n]
for i ← n, ..., 1 do SINK(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)