8 Tas binaire

8.1 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 ← 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, 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 Θ(n) au pire
D1  N ← head; M ← null; v ← ∞
D2  while N.next ≠ null
D3      w ← N.next.info
D4      if w < v
D5      then v ← w; M ← N                        //(M contient min)
D6      N ← N.next
D7  deleteAfter(M)                               //(suppression après nœud M)
D8  return v
```

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

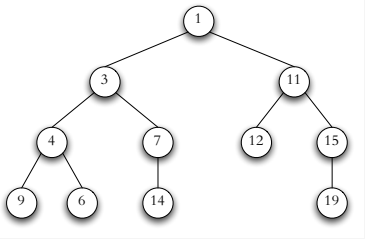
```
    Opération insert(x)                          // en Θ(n) au pire
I1  insertLast(x)
I2  N ← queue; P ← N.precedent
I3  while P ≠ head et P.info > x
I4      N.info ← P.info // (décalage vers la fin)
I5      N ← P; P ← N.precedent
I6  N.info ← x

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

8.2 Ordre de tas

W(fr)

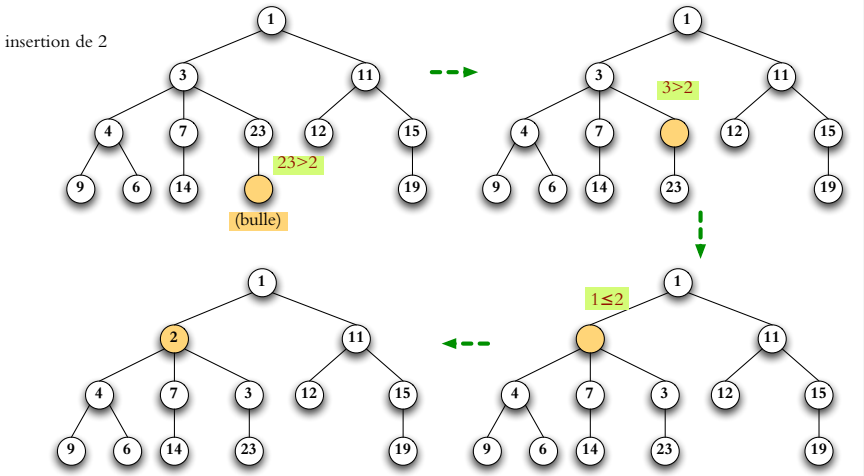
On peut améliorer l'approche impatiente avec des «branchements» dans la liste chaînée (exemple : hiérarchie militaire — le vieux général est remplacé par son meilleur lieutenant-général, qui est remplacé par son major-général, etc.)



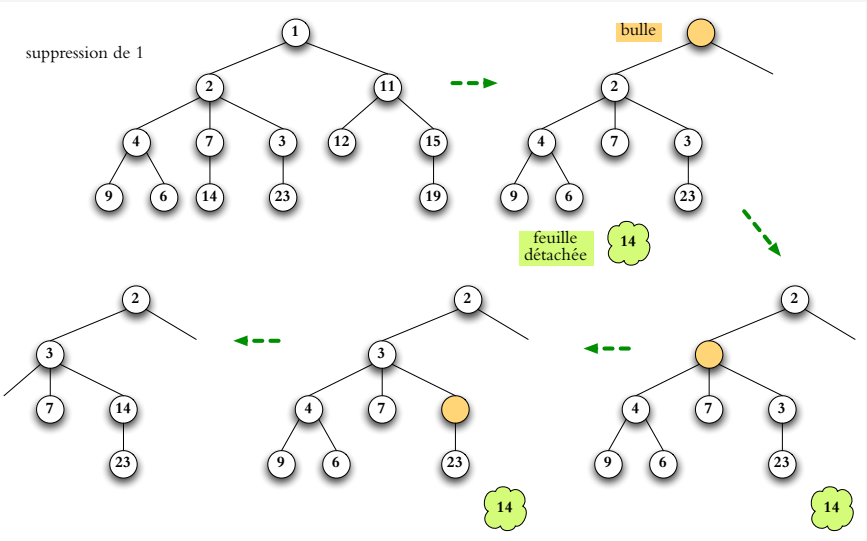
Pour implanter la file de priorité, on se sert d'une arborescence dont les nœuds sont dans l'**ordre de tas** : si x n'est pas la racine, alors

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

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



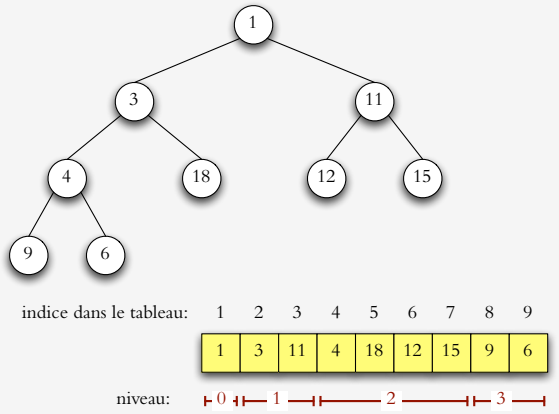
swim (nager) : ajouter une feuille vide («bulle») + monter la bulle vers la racine jusqu'à ce qu'on trouve la place pour la nouvelle valeur
Temps : proportionnel à la profondeur



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

8.3 Tas binaire

W(fr)



On choisit la structure de l'arbre pour sink/swim : le meilleur choix est un arbre binaire complet (Théorème 6.3).
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
```

valeur à placer
indice
tableau
indice maximal

```
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 : $O(1)$