6 Tas binaire

6.1 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 ← 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)                    // 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
```

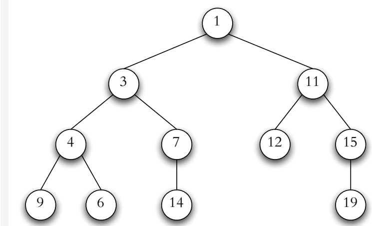
Approche **impatiente** (*eager*) avec liste triée + sentinelle

```
    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
```

6.2 Ordre de tas

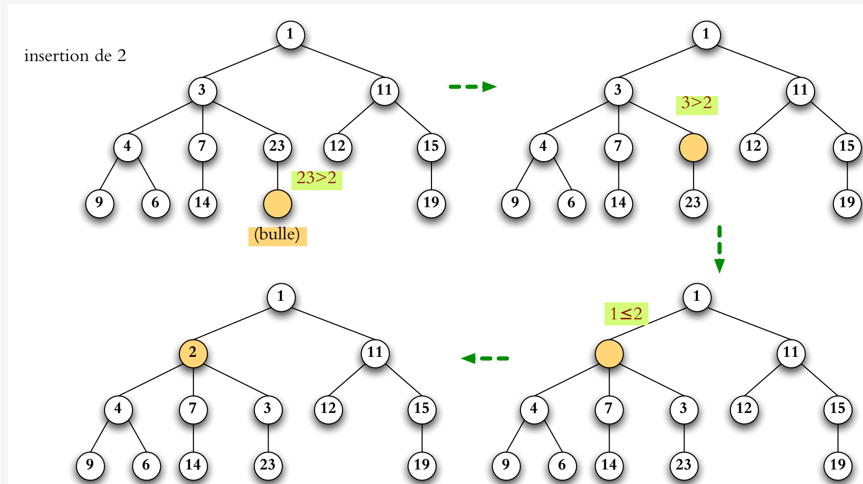
On peut améliorer l'approche impatiente avec des «branchements» dans la liste chaînée (exemple : hiérarchie militaire — le vieux général est remplacé par son meilleur lieutenant-général, qui est remplacé par son major-général, etc.)



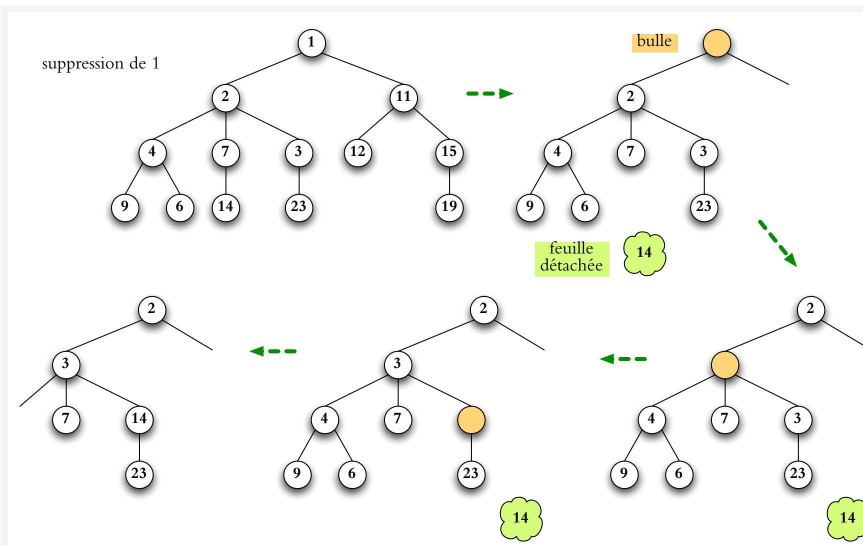
Pour implanter la file de priorité, on se sert d'une arborescence dont les nœuds sont dans l'**ordre de tas** : si x n'est pas la racine, alors

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

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

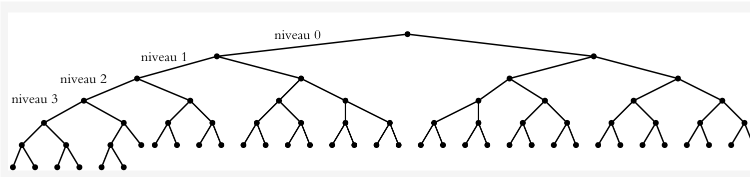


swim (nager) : ajouter une feuille vide («bulle») + monter la bulle vers la racine jusqu'à ce qu'on trouve la place pour la nouvelle valeur
Temps : proportionnel à la profondeur



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

6.3 Arbre binaire complet



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

Théorème 6.1. *Un arbre binaire à n nœuds externes contient $(n-1)$ nœuds internes.*

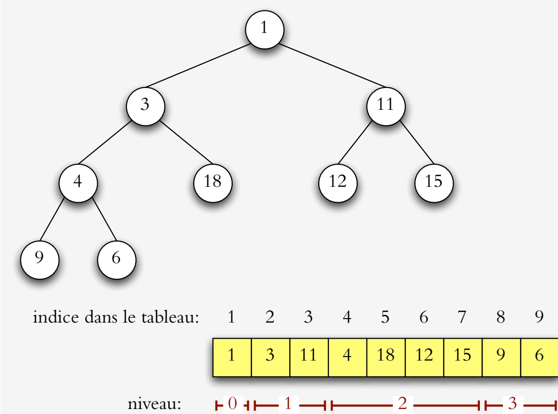
Théorème 6.2. *La hauteur h d'un arbre binaire à n nœuds* externes *est bornée par*

$$\lceil \lg(n) \rceil \leq h \leq n-1. \tag{6.1}$$

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 $k = 0, 1, 2, \ldots, h-1$ (il n'y a pas de nœud interne au niveau h). Par Théorème 6.1, on a $n - 1 = \sum_{k=0}^{h-1} m_k$. Comme $m_k \geq 1$ pour tout $k = 0, \ldots, h-1$, 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. ∎

6.4 Tas binaire

W(fr)

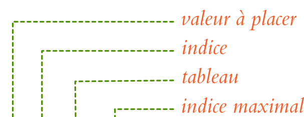


On choisit la structure de l'arbre pour sink/swim : le meilleur choix est un arbre binaire complet (Théorème 6.2).
On utilise un tableau $H[1..n]$ pour l'encodage compact de l'arbre binaire complet. Parent de nœud i est à $\lfloor i/2 \rfloor$, enfant gauche est à $2i$, enfant droit est à $2i+1$.
Tableau en ordre de tas : $H[i] \le H[2i], H[2i+1]$.

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 $v \leftarrow H[n]$; $H[n] \leftarrow$ 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 $c \leftarrow$ MINCHILD(i, H, n)
C3 **while** $c \neq 0$ et $H[c] < v$ **do** $H[i] \leftarrow H[c]$; $i \leftarrow c$; $c \leftarrow$ 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** $2i \le n$ **then** $j \leftarrow 2i$
M3 **if** $(2i+1 \le n)$ && $(H[2i+1] < H[j])$ **then** $j \leftarrow 2i+1$
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 $O(h)$ temps.

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

6.5 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)

Tas d-aire. Au lieu d'un arbre binaire, on peut baser le tas sur un arbre complet avec arité arbitraire $d \geq 2$. 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 : 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.

6.6 Heapisation

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

$H \leftarrow \emptyset$; **for** $i \leftarrow 1, \ldots, n$ **do** INSERT$(A[i], H, i)$; $A \leftarrow H$

⇒ prend $\Theta(n \log n)$ au pire

Meilleure solution :

```
HEAPIFY(A) // tableau arbitraire A[1..n]
for i ← ⌈n/2⌉, ..., 1 do SINK(A[i], i, A, n)
```

Théorème 6.3. 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 $T(n) \leq \sum_{h=1}^{1+\lfloor \lg n \rfloor} \lceil \frac{n}{2^h} \rceil \times O(h) = O(n) \cdot \left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots \right) = O(n)$. ■

6.7 Tri par tas

W(fr)

On peut utiliser une file de priorité pour tri : mettre tous éléments dans la file (heapify), et retirer-les un après l'autre dans l'ordre croissant. Avec un tas binaire, on peut faire le tri en place. 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. (On finira avec l'ordre décroissant — pour l'ordre croissant, utiliser un max-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)     // rétablissement de l'ordre de tas en A[1..i − 1]
```

- ⋆ **heapsort** : temps $O(n \log n)$ dans le pire des cas, sans espace additionnelle !
- ⋆ **quicksort** : $O(n^2)$ dans le pire des cas
- ⋆ **mergesort** : $O(n \log n)$ dans le pire des cas mais utilise un espace auxiliaire de taille n.