FILES DE PRIORITÉ

File de priorité

Type abstrait d'une **file de priorité** ou **tas** ou **monceau**

(*priority queue* & *heap*)

Objets : ensembles de valeurs naturelles (abstraction de clés comparables)

Opérations :
$\mathsf{insert}(x, H)$: insertion de l'élément x dans H
$\mathsf{deleteMin}(H)$: enlever l'élément de valeur minimale dans H

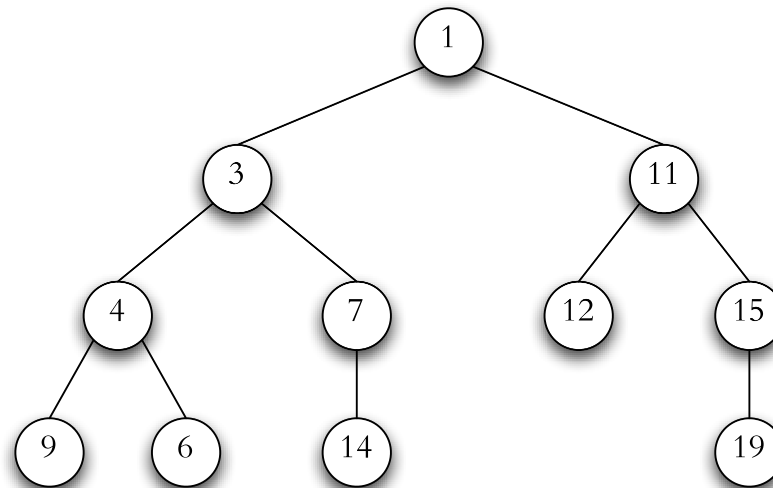
Opérations parfois supportées :
$\mathsf{merge}(H_1, H_2)$: fusionner deux files
$\mathsf{findMin}(H)$: retourne (mais ne supprime pas) l'élément minimal
$\mathsf{delete}(x, H)$: supprimer élément x

Notez qu'un arbre équilibré peut offrir toutes ces fonctionnalités en $O(\log n)$

Monceau

on va implanter la file de priorité par une arborescence dont les nœuds sont dans l'**ordre de monceau** :

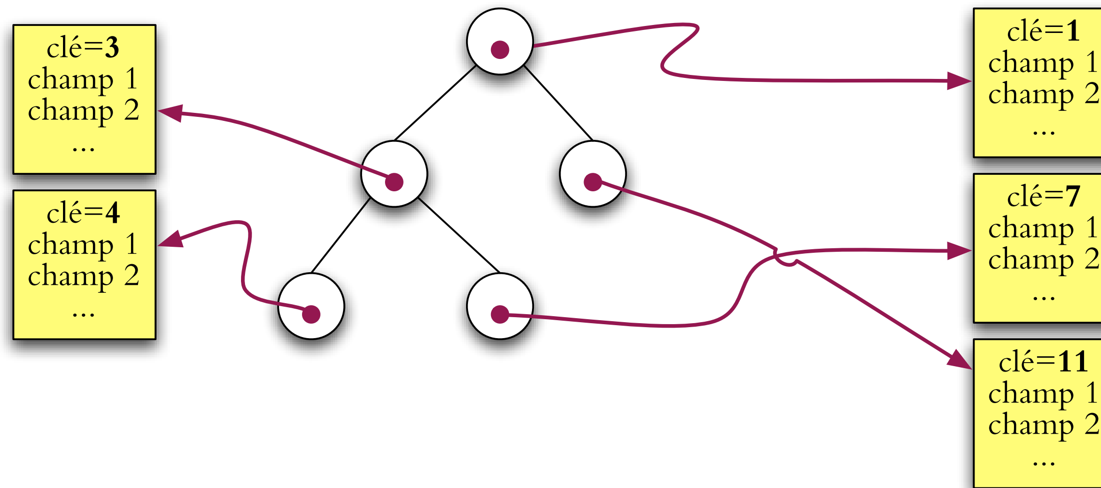
si x n'est pas la racine, alors $\mathsf{val}\Big(\mathsf{parent}(x)\Big) \le \mathsf{val}(x)$.



Opération findMin en $O(1)$: retourner val(racine)

Monceau (cont)

Les valeurs ne sont pas stockées avec les nœuds mais plutôt un pointeur vers les données associées (en Java, il n'y a pas de grande différence : val est un objet `Comparable`)

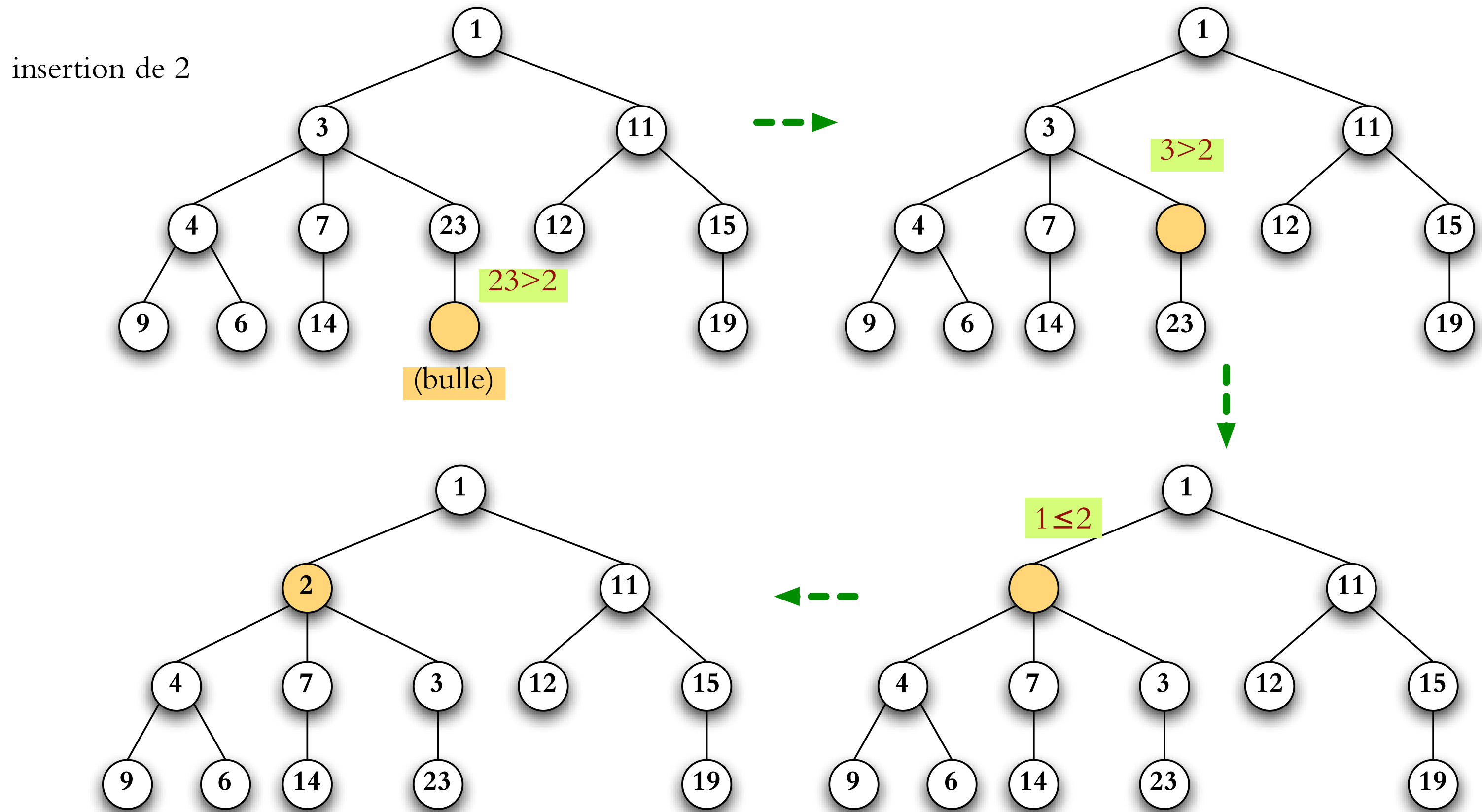


Comment insérer et supprimer ?

Idée de base : on ne change pas la structure de l'arbre
- affectation de pointeurs de données seulement

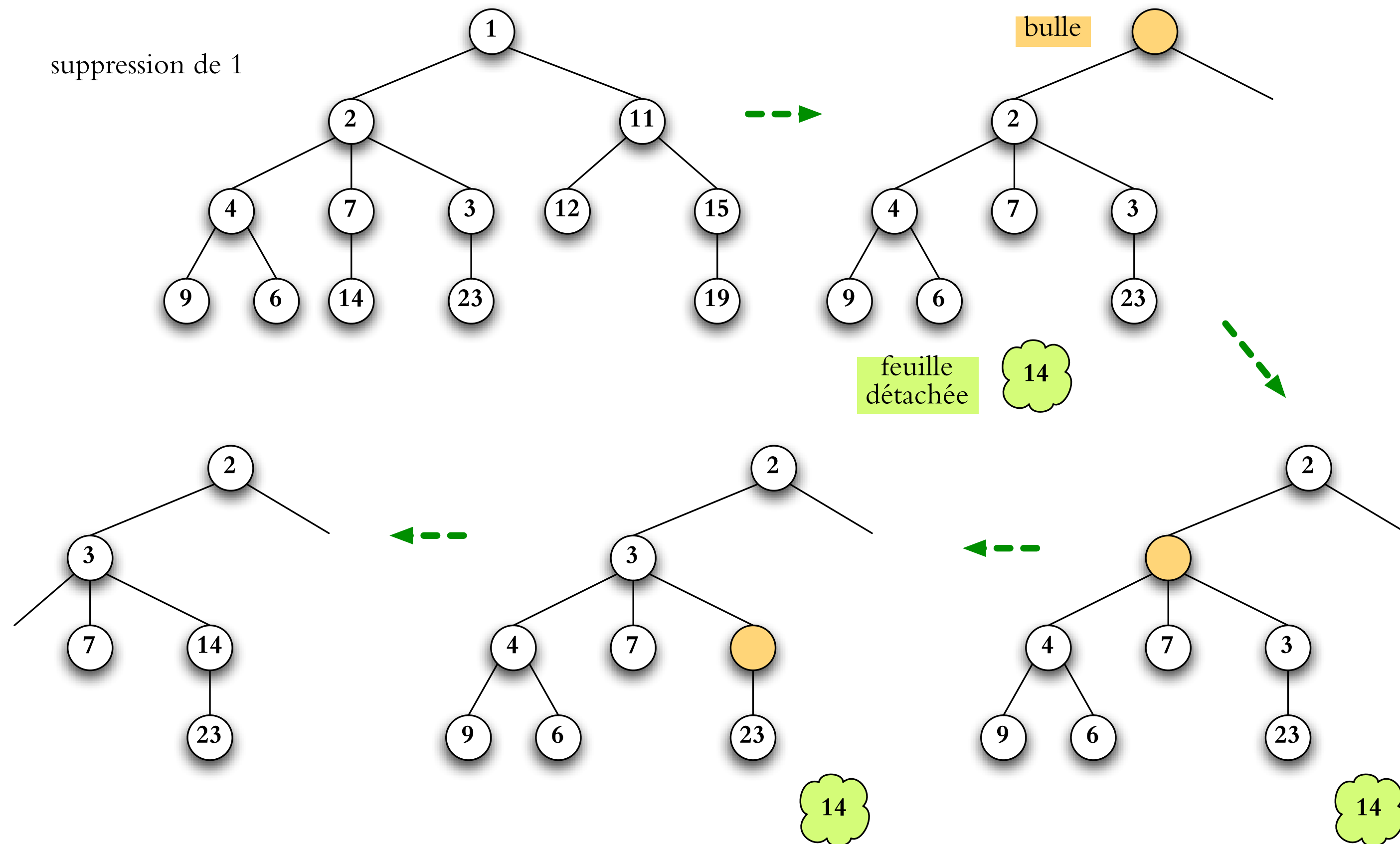
Monceau — insertion

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



Monceau — suppression

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

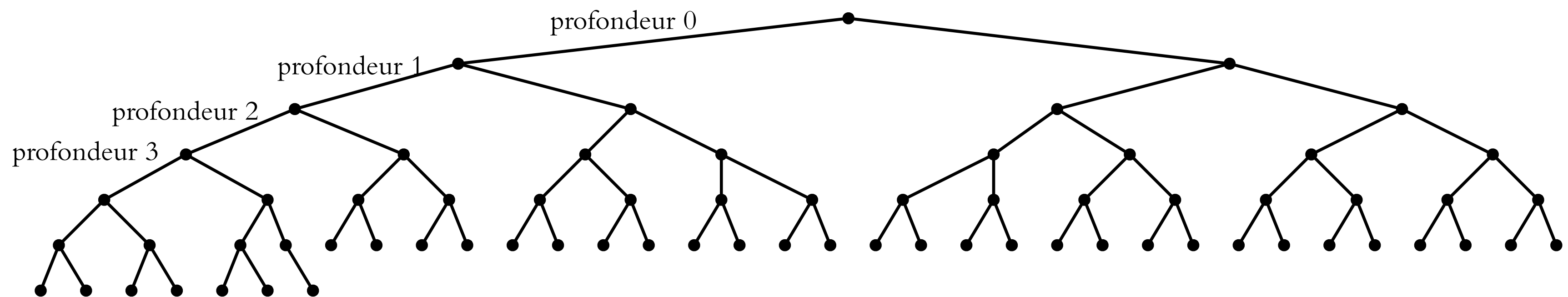


Monceau — efficacité

Temps pour insertion : dépend de la profondeur où on crée la bulle

Temps pour suppression : dépend du nombre des enfants des nœuds échangés avec la bulle

Une solution simple : utiliser un **arbre binaire complet** de hauteur h :
il y a 2^i nœuds de chaque profondeur $i = 0, \ldots, h-1$; les niveaux sont «remplis» de gauche à droit

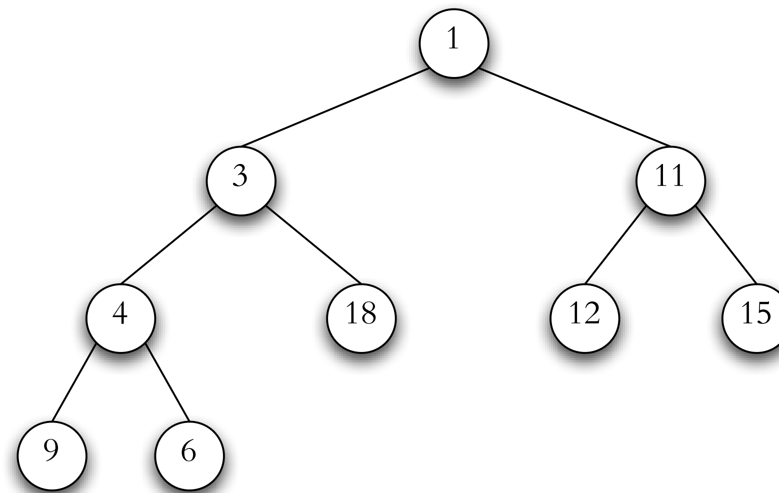


Tas binaire

Arbre binaire complet $\rightarrow$ pas de pointeurs parent, left, gauche !

Les n clés sont stockés dans un tableau $H[1..n]$.

Parent de nœud i est à $\lceil (i-1)/2 \rceil$, enfant gauche est à $2i$, enfant droit est à $2i+1$.



indice dans le tableau:	1	2	3	4	5	6	7	8	9
	1	3	11	4	18	12	15	9	6
profondeur:	0	1		2				3	

Tas binaire — insertion

```
    INSERT(v, H) // tas binaire dans H[1..|H|]
I1  NAGER(v, |H| + 1, H)
```

```
    NAGER(v, i, H) // tas binaire dans H[1..|H|]
N1  p ← ⌈(i − 1)/2⌉
N2  while p ≠ 0 et H[p] > v do
N3      H[i] ← H[p]
N4      i ← p
N5      p ← ⌈(i − 1)/2⌉
N6  H[i] ← v
```

(NAGER est «percolate up» dans le livre)

en N1 et N5, on peut juste faire un décalage binaire (`p=i>>1` en Java) — très rapide

Tas binaire — suppression

```
    DELETEMIN(H) // tas dans H[1..|H|]
D1  r ← H[1]
D2  if |H| > 1
D3  then v ← H[|H|] ; H[|H|] ← null ; COULER(v, 1, H)
D4  return r
```

```
    COULER(v, i, H) // tas dans H[1..|H|]
C1  c ← MINCHILD(i, H)
C2  while c ≠ 0 et H[c] < v do
C3      H[i] ← H[c]
C4      i ← c
C5      c ← MINCHILD(i, H)
C6  H[i] ← v
```

(COULER est implanté par `percolateDown` dans le livre)

```
    MINCHILD(i, H)
    // retourne l'enfant avec clé minimale ou 0 si i est une feuille
```

Tas binaire — efficacité

Hauteur de l'arbre est toujours $\lfloor \lg n \rfloor$

deleteMin : $O(\lg n)$

insert : $O(\lg n)$

findMin : $O(1)$

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 :

parent de l'indice i est $\lceil (i-1)/d \rceil$, enfants sont à $d(i-1)+2..di+1$

ordre de monceau :

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

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 : $O(1)$
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

Tas d-aire — construction

Opération heapify (A) met les éléments de la vecteur $A[1..n]$ dans l'ordre de monceau.

Triviale ?

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

$\Rightarrow$ prend $O(n \log_d n)$

Meilleure solution :

	$\textsc{Heapify}(A)$ // *vecteur arbitraire* $A[1..n]$
H1	**for** $i \leftarrow n, \ldots, 1$ **do** $\textsc{Couler}(A[i], i, A)$

$\Rightarrow$ prend $O(n)$

Tas d-aire — construction (cont)

Preuve du temps de calcul : si i est à la hauteur j, alors il prend $O(j)$ de faire COULER$(\cdot, i, \cdot)$. Il y a $\leq n/d^j$ nœuds à la hauteur j. Donc temps est de

$$\sum_j \frac{n}{d^j} O(j) = O\Bigl(n \sum_j \frac{j}{d^j}\Bigr) = O(n).$$

$\square$

«Évidemment», $O(n)$ est optimal pour construire le tas.

Preuve formelle :

- Trouver le minimum des élements dans une vecteur de taille n prend $n-1$ comparaisons, donc un temps de $\Omega(n)$ est nécessaire pour trouver le minimum.
- Avec n'importe quelle implantation de heapify, on peut appeler findMin après pour retrouver le minimum en $O(1)$.
- Donc le temps de heapify doit être $\Omega(n)$, sinon on pourrait trouver le minimum en utilisant heapify+findMin en un temps $o(n) + O(1) = o(n)$. $\square$

Tri par tas

```
    HEAPSORT(A) // vecteur non-trié A[1..n]
H1  heapify(A)
H2  for i ← |A|, ... 2 do
H3      échanger A[1] ↔ A[i]
H4      COULER(A[1], 1, A[1..i − 1])
```

$A[1..n]$ est dans l'ordre décroissant à la fin

(pour l'ordre croissant, utiliser un max-monceau)

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

Files de priorité

Autres implantations existent (nécessaires pour un merge efficace) :
binomial heap, skew heap, Fibonacci heap

	binaire (pire)	**binomial** (pire)	**skew** (amorti)	**Fibonacci** (amorti)
deleteMin	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
insert	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$
merge	$O(n)$	$O(\log n)$	$O(1)$	$O(1)$
decreaseKey	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$

opération decreaseKey : change la priorité d'un élément — dans un tas binaire on peut le faire à l'aide de NAGER

decreaseKey est important dans quelques algorithmes fondamentaux sur des graphes (plus court chemin, arbre couvrant minimal)

Applications

Simulations d'événements discrets

Algorithme A* : arriver à sa destination malgré des obstacles (labyrinthe)