

FILES À PRIORITÉS

File à priorités

Type abstrait d'une **file à priorités** (*priority queue*)

Objets : ensembles d'objets avec clés comparables (abstraction : nombres naturels)

Opérations :

$\text{insert}(x, H)$: insertion de l'élément x dans H

$\text{deleteMin}(H)$: enlever l'élément de valeur minimale dans H

Opérations parfois supportées :

$\text{merge}(H_1, H_2)$: fusionner deux files

$\text{findMin}(H)$: retourne (mais ne supprime pas) l'élément minimal

$\text{delete}(x, H)$: supprimer élément x

(ou définition équivalente avec deleteMax et findMax — mais pas max et min en même temps)

Applications

simulations d'événements discrets (p.e., collisions)

systèmes d'exploitation (interruptions)

algorithmes sur graphes, recherche opérationnelle (plus courts chemins, arbre couvrant)

statistiques : maintenir l'ensemble des m meilleurs éléments

Implantations inefficaces

- tableau / liste chaînée avec des éléments non-ordonnés — approche **paresseuse**
 - insert en $O(1)$
 - deleteMin en $O(n)$
- tableau / liste chaînée avec des éléments ordonnés — approche **impatiente**
 - insert en $O(n)$
 - deleteMin en $O(1)$

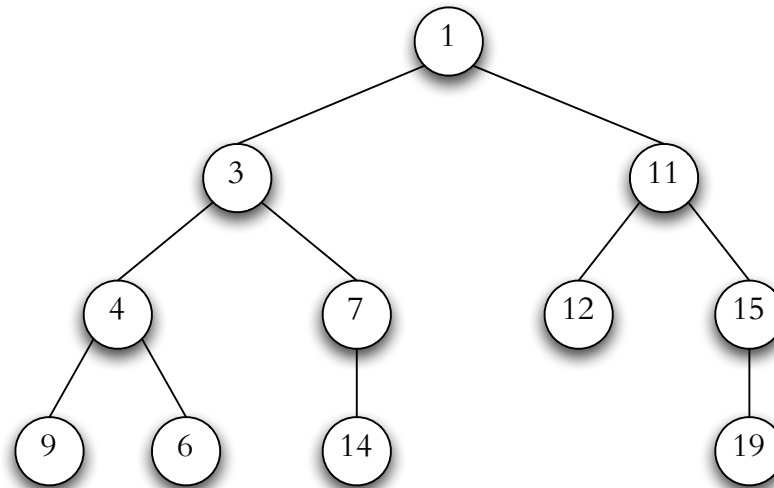
On veut une meilleure solution. . .

Tas

on va implanter la file de priorité à l'aide d'une arborescence dont les nœuds sont dans

l'ordre de tas :

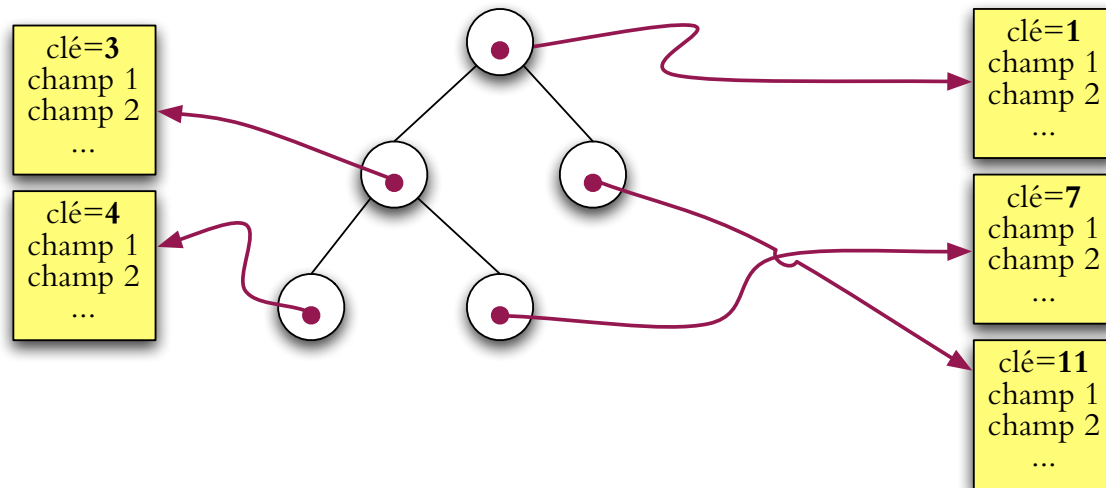
si x n'est pas la racine, alors $\text{val}(\text{parent}(x)) \leq \text{val}(x)$.



Opération findMin en $O(1)$: retourner $\text{val}(\text{racine})$

Tas (cont)

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



```
class NoeudTas
{
    NoeudTas enfants[], parent;
    Comparable val;
}
```

Tas (cont)

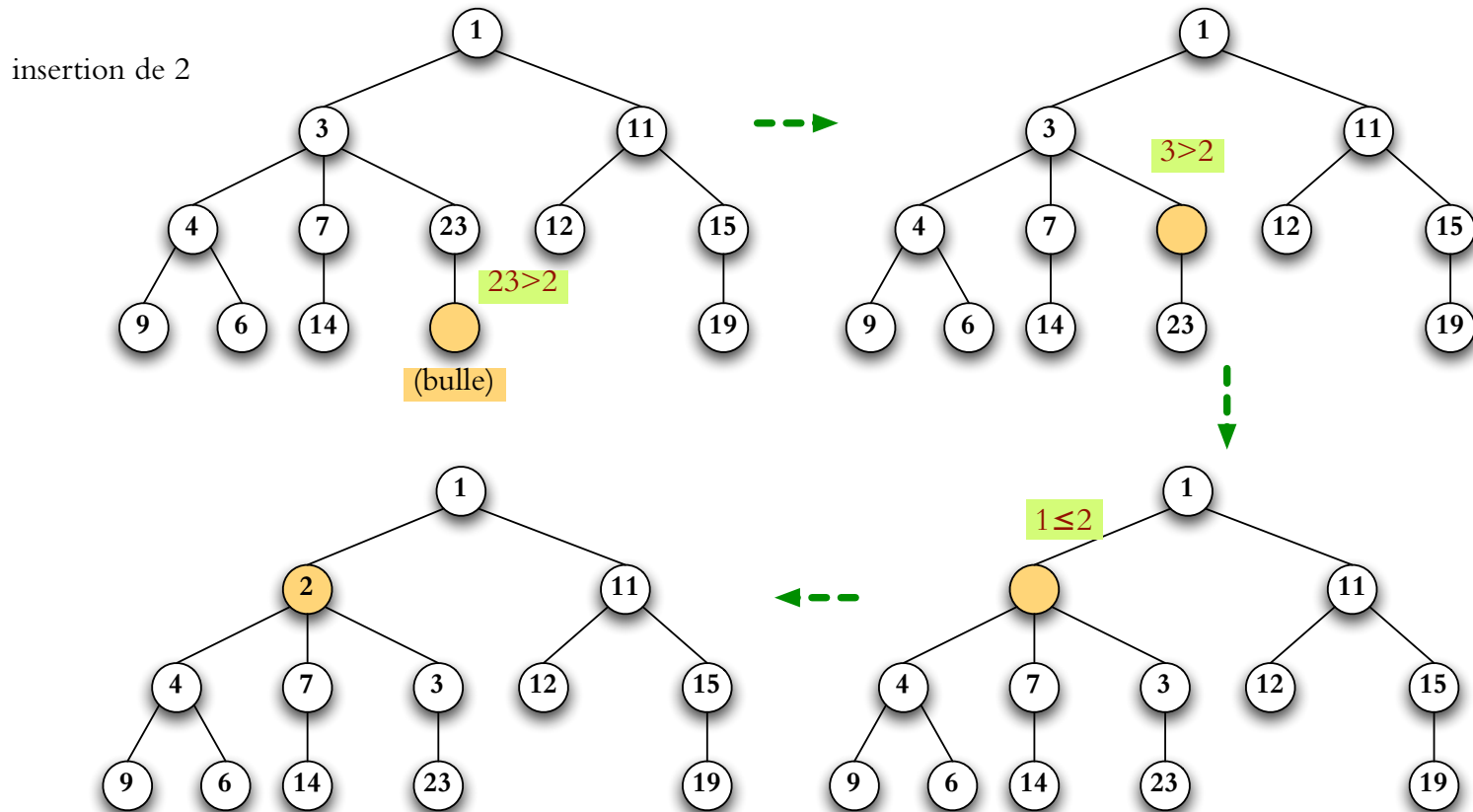
Comment insérer et supprimer ?

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

(Donc il s'agit d'une structure exogène. . .)

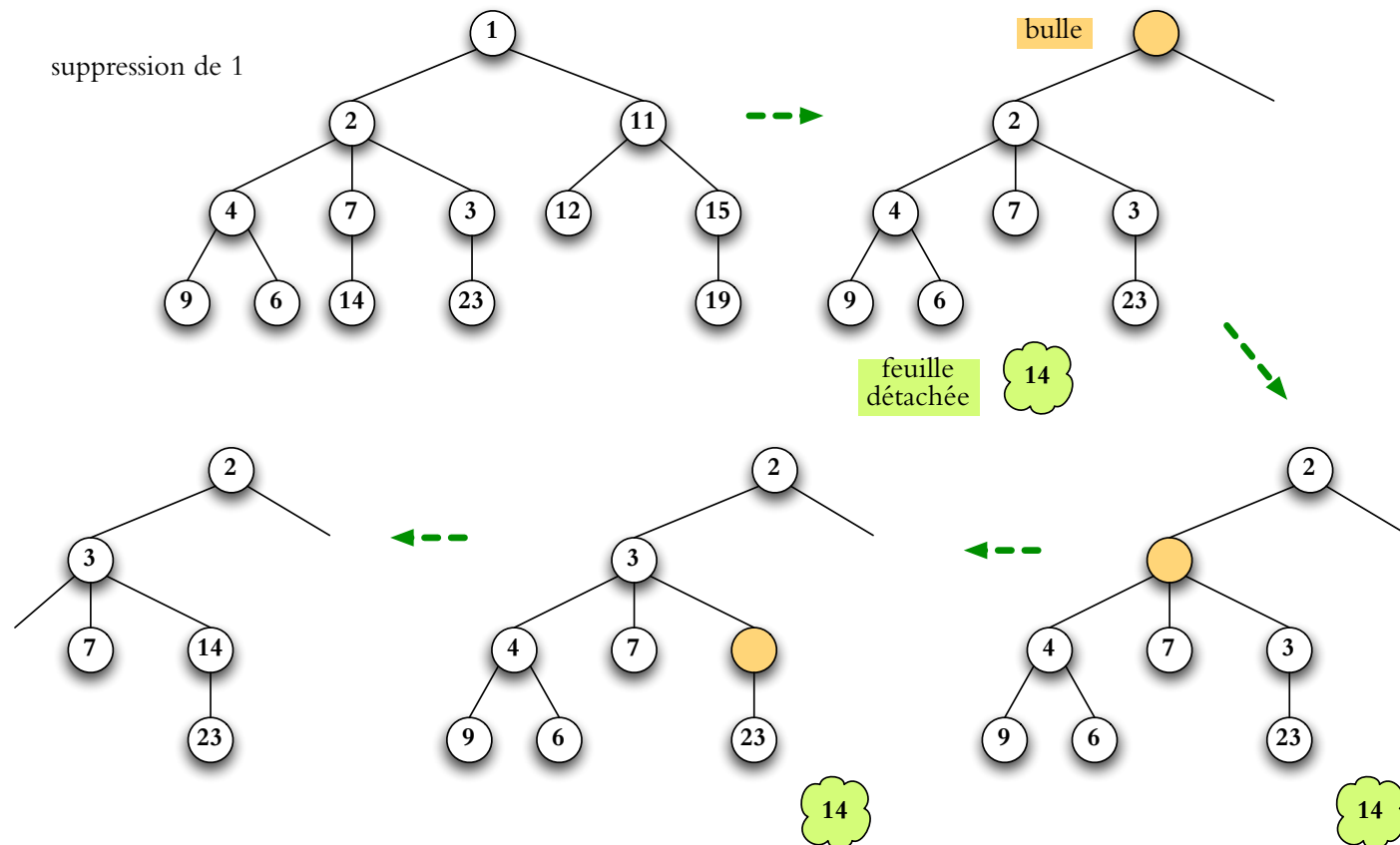
Tas — insertion

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



Tas — 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 (**sombrer**)

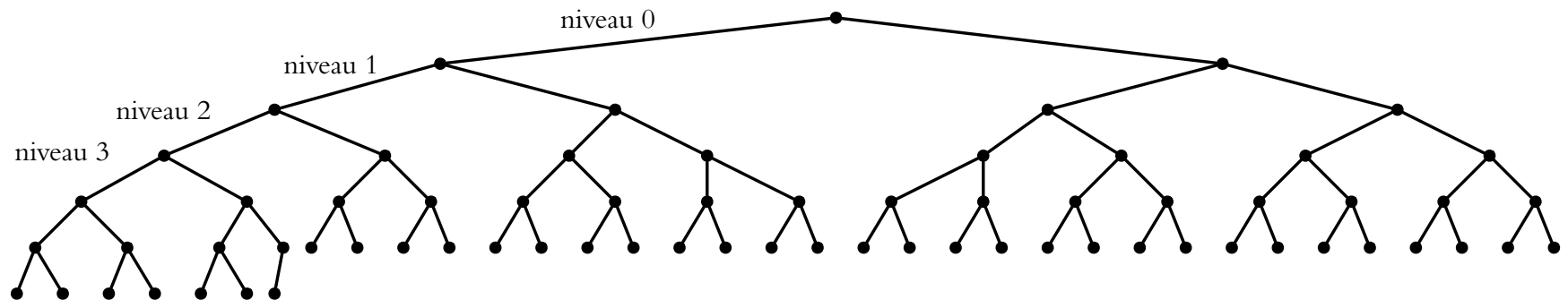


Tas — efficacité

Temps pour insertion : dépend du niveau 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 à chaque niveau $i = 0, \dots, h - 1$; les niveaux sont «remplis» de gauche à droit

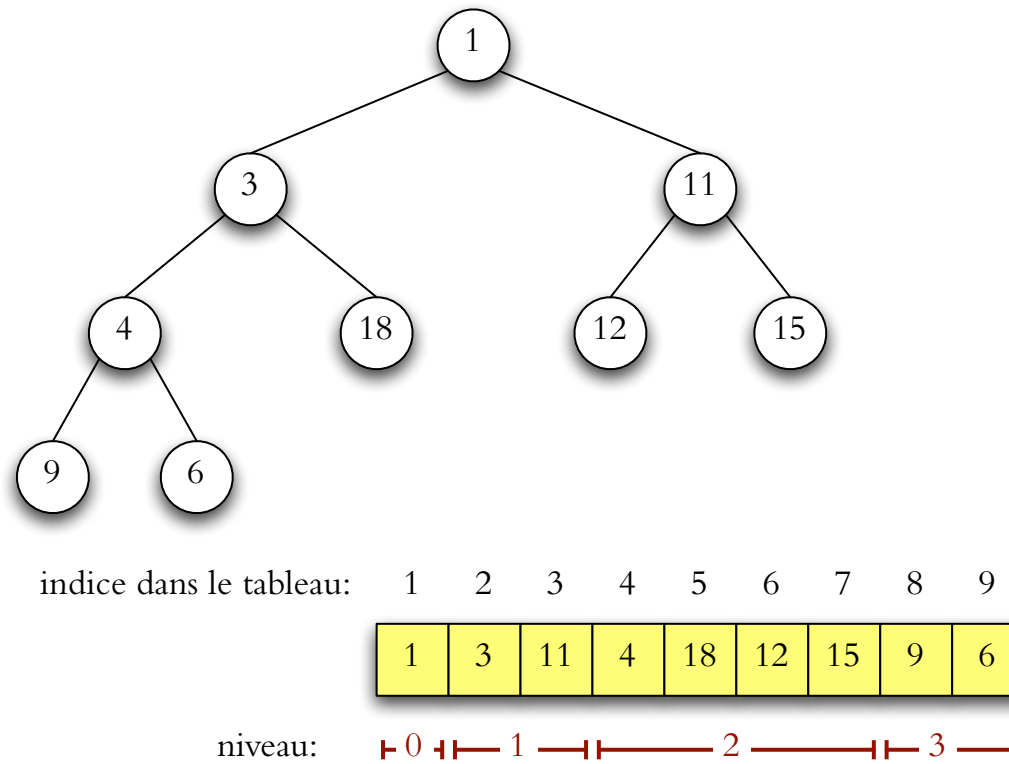


Tas binaire

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

Les n clés sont placées 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 $2i + 1$.



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 $p \leftarrow \lceil (i - 1) / 2 \rceil$

N2 **tandis que** $p \neq 0$ et $H[p] > v$ **faire**

N3 $H[i] \leftarrow H[p]$

N4 $i \leftarrow p$

N5 $p \leftarrow \lceil (i - 1) / 2 \rceil$

N6 $H[i] \leftarrow v$

(NAGER est swim 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  $r \leftarrow H[1]$   
D2 si  $|H| > 1$   
D3 alors  $v \leftarrow H[|H|]$  ;  $H[|H|] \leftarrow \text{null}$  ; SOMBRER( $v, 1, H$ )  
D4 retourner  $r$ 
```

```
SOMBRER( $v, i, H$ ) // tas dans  $H[1..|H|]$   
C1  $c \leftarrow \text{MINCHILD}(i, H)$   
C2 tandis que  $c \neq 0$  et  $H[c] < v$  faire  
C3      $H[i] \leftarrow H[c]$   
C4      $i \leftarrow c$   
C5      $c \leftarrow \text{MINCHILD}(i, H)$   
C6  $H[i] \leftarrow v$ 
```

(COULER est implanté par sink 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 $\lceil \lg n \rceil$

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 tas :

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

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 SOMBRER : $O(\log_d n)$ 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

Files à priorité

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

	liste triée	liste non-triée	binaire	binomial
deleteMin	$O(1)$	$O(n)$	$O(\log n)$	$O(\log n)$
insert	$O(n)$	$O(1)$	$O(\log n)$	$O(\log n)$
merge	$O(n)$	$O(1)$	$O(n)$	$O(\log n)$
decreaseKey	$O(n)$	$O(1)$	$O(\log n)$	$O(\log n)$

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)

Tas d -aire — construction

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

Triviale ?

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

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

Meilleure solution :

HEAPIFY(A) // vecteur arbitraire $A[1..n]$
H1 **pour** $i \leftarrow n, \dots, 1$ **faire** SOMBRER($A[i], i, A$)

$\Rightarrow$ 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 $\text{SOMBRER}(\cdot, i, \cdot)$. Il y a $\leq n/d^j$ nœuds à la hauteur j . Donc temps est de

$$\sum_j \frac{n}{d^j} O(j) = O\left(n \sum_j \frac{j}{d^j}\right) = O(n).$$

□

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

Preuve formelle :

- Trouver le minimum des éléments dans un 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 $o(n) + O(1) = o(n)$. □