

# Cours 7: Arbres de recherche. Tas.

Olivier Bournez

bournez@lix.polytechnique.fr  
LIX, Ecole Polytechnique

# Aujourd'hui

Dictionnaires, Files de priorité

Arbres binaires de recherche

Arbres AVL

Les tas

## Le type abstrait Dictionnaire

- On suppose que l'on a des clés  $k$  qui vivent dans un univers  $U$  **totalemment ordonné**.
- A chaque clé  $k$ , on associe une donnée  $D$ . Les données vivent dans un univers  $V$ .
- Par rapport au cours sur le hachage, on veut ne pas seulement supporter les opérations d'insertion, recherche, et suppression.
- On suppose dans ce cours, pour simplifier, que comme pour le cours sur le hachage, il n'y a pas de clé en "double": une clé identifie une unique donnée.

## Le type abstrait Dictionnaire

On cherche des structures de données, Dictionnaire, qui permettent

- D'insérer un couple (clé, donnée): `insérer(k,D)`.
- Retrouver la donnée<sup>1</sup> associée à une clé: `rechercher(k)`.
- Supprimer la donnée<sup>1</sup> associée à une clé: `supprimer(k)`.

---

<sup>1</sup>Quand elle/il existe

## Le type abstrait `Dictionnaire`

On cherche des structures de données, `Dictionnaire`, qui permettent

- D'insérer un couple (clé, donnée): `insérer(k,D)`.
- Retrouver la donnée<sup>1</sup> associée à une clé: `rechercher(k)`.
- Supprimer la donnée<sup>1</sup> associée à une clé: `supprimer(k)`.

(puisque  $U$  est totalement ordonné)

---

<sup>1</sup>Quand elle/il existe

## Le type abstrait `Dictionnaire`

On cherche des structures de données, `Dictionnaire`, qui permettent

- D'insérer un couple (clé, donnée): `insérer(k,D)`.
- Retrouver la donnée<sup>1</sup> associée à une clé: `rechercher(k)`.
- Supprimer la donnée<sup>1</sup> associée à une clé: `supprimer(k)`.

(puisque  $U$  est totalement ordonné)

- Renvoyer le couple (clé, donnée)<sup>1</sup> associé à la clé la plus petite: `minimum()`.
- Renvoyer le couple (clé, donnée)<sup>1</sup> associé à la clé la plus grande: `maximum()`.

---

<sup>1</sup>Quand elle/il existe

## Application 1: représenter un ensemble

- On peut choisir  $V = \emptyset$ .
- Un Dictionnaire permet alors de représenter un ensemble d'éléments de  $U$ .
- Insérer un élément: `insérer(k)`.

## Application 1: représenter un ensemble

- On peut choisir  $V = \emptyset$ .
- Un Dictionnaire permet alors de représenter un ensemble d'éléments de  $U$ .
- Insérer un élément: `insérer(k)`.
- Tester l'appartenance d'un élément: `rechercher(k)`.

## Application 1: représenter un ensemble

- On peut choisir  $V = \emptyset$ .
- Un Dictionnaire permet alors de représenter un ensemble d'éléments de  $U$ .
- Insérer un élément: `insérer(k)`.
- Tester l'appartenance d'un élément: `rechercher(k)`.
- Supprimer un élément: `supprimer(k)`.

## Application 1: représenter un ensemble

- On peut choisir  $V = \emptyset$ .
- Un Dictionnaire permet alors de représenter un ensemble d'éléments de  $U$ .
- Insérer un élément: `insérer(k)`.
- Tester l'appartenance d'un élément: `rechercher(k)`.
- Supprimer un élément: `supprimer(k)`.
- Renvoyer l'élément minimum: `minimum()`.

## Application 1: représenter un ensemble

- On peut choisir  $V = \emptyset$ .
- Un Dictionnaire permet alors de représenter un ensemble d'éléments de  $U$ .
- Insérer un élément: `insérer(k)`.
- Tester l'appartenance d'un élément: `rechercher(k)`.
- Supprimer un élément: `supprimer(k)`.
- Renvoyer l'élément minimum: `minimum()`.
- Renvoyer l'élément maximum: `maximum()`.

## Application 2: les files de priorité

Structures de données pour représenter un ensemble d'éléments dans un domaine  $V$ , avec une priorité  $k \in U$  associée à chaque élément.

On a besoin:

- D'insérer un couple (priorité, élément): `insérer(k,D)`.
- Trouver l'élément de priorité maximale: `maximum()`.
- Enlever l'élément de priorité maximale:  
`supprimer(clé(maximum()))`.

Exemple d'application:

- gestion des tâches sur un ordinateur partagé:
  - ▶ lorsqu'une tâche se termine, on choisit la tâche suivante parmi celles de priorité maximum.

# Résumé

## ■ Complexité en moyenne:

	rechercher	insérer	supprimer	minimum	maximum
tableau/liste	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
dichotomie	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
hachage	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$
arbre de recherche	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
arbre équilibré	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
tas	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$

## ■ Complexité au pire cas:

tableau/liste	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
dichotomie	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
hachage	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
arbre de recherche	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
arbre équilibré	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
tas	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$

# Aujourd'hui

Dictionnaires, Files de priorité

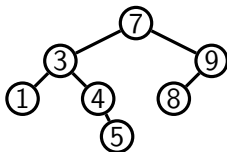
Arbres binaires de recherche

Arbres AVL

Les tas

# Arbre binaire de recherche

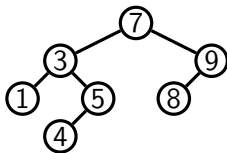
- Un arbre binaire de recherche est un arbre binaire, dans lequel chaque sommet est étiqueté par un couple  $(k, D)$ , avec la propriété suivante:
  - ▶ tous les sommets du sous-arbre
    - gauche ont une clé inférieure à  $k$ .
    - droit ont une clé supérieure à  $k$ .
- Exemple:



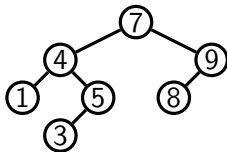
(Seules les clés sont représentées dans les dessins)

# Arbres binaires de recherche

- Autre exemple:



- Contre-exemple:



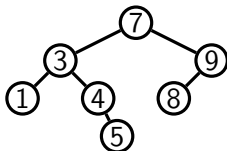
## Coder un arbre binaire de recherche

- Un arbre binaire de recherche est un arbre.

```
class Arbre {  
    int k; int D;  
    Arbre gauche, droite;  
    Arbre (Arbre gauche, int k, int D, Arbre droite) {  
        this.gauche = gauche;  
        this.k = k;  
        this.D = D;  
        this.droite = droite; }}
```

## Première propriété

- Propriété: Un parcours infixe d'un arbre binaire de recherche traite les sommets par clés croissantes.

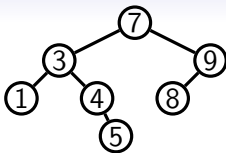


- Exemple:

```
static void affiche(Arbre a) {  
    if (a==null) return;  
    affiche(a.gauche);  
    System.out.print(a.k+" ");  
    affiche(a.droite);} 
```

Affiche: 1 3 4 5 7 8 9

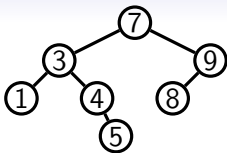
## Rechercher une clé $k$ dans un arbre $A$



- Solution récursive simple:

```
static Arbre rechercher(int k, Arbre a) {  
    if (a==null) return null;  
    else if (a.k==k) return a;  
    else if (k<a.k) return rechercher(k,a.gauche);  
    else return rechercher(k,a.droite);  
}
```

## Rechercher une clé $k$ dans un arbre $A$



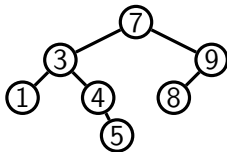
- Solution récursive simple:

```
static Arbre rechercher(int k, Arbre a) {  
    if (a==null) return null;  
    else if (a.k==k) return a;  
    else if (k<a.k) return rechercher(k,a.gauche);  
    else return rechercher(k,a.droite);  
}
```

- `rechercher(k, a)` renvoie
  - ▶ le sous-arbre de racine le sommet avec la clé  $k$ , si  $k$  est présente dans l'arbre,
  - ▶ et `null` si  $k$  n'y est pas.

## Rechercher une clé $k$ dans un arbre $A$

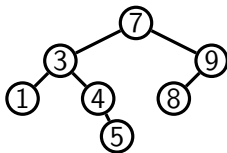
- Version itérative: pas vraiment plus compliqué. On suit un chemin dans l'arbre.



```
static Arbre rechercher(int k, Arbre a) {  
    while (a != null) {  
        if (a.k == k) return a;  
        else if (k < a.k) a = a.gauche;  
        else if (k > a.k) a = a.droite;  
    }  
    return null;  
}
```

## Insérer une nouvelle clé $k$

- Solution récursive (persistante):

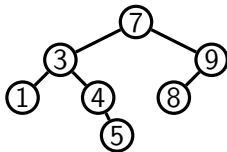


```
static Arbre insérer(int k, int D, Arbre a) {  
    if (a==null)  
        return new Arbre(null,k,D,null);  
    else if (k<a.k) // On insère à gauche  
        return new Arbre(insérer(k,D,a.gauche),a.k,a.D,a.droite);  
    else if (k>a.k) // On insère à droite  
        return new Arbre(a.gauche,a.k,a.D,insérer(k,D,a.droite));  
    else // k==a.k: clé déjà présente, ne rien faire.  
        return a;}
```

- Solution itérative plus compliquée.

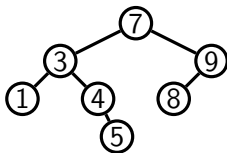
## Insérer une nouvelle clé $k$

- Solution récursive (non-persistante):



```
static Arbre insérer2(int k, int D, Arbre a) {  
    if (a==null)  
        return new Arbre(null,k,D,null);  
    else if (k<a.k) // On insère à gauche  
        a.gauche = insérer(k,D,a.gauche);  
    else if (k>a.k) // On insère à droite  
        a.droite= insérer(k,D,a.droite);  
    return a;}
```

## Renvoyer l'élément de clé maximum

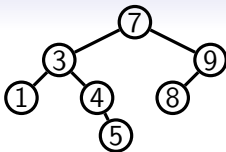


- L'élément de clé la plus grande s'obtient en allant systématiquement à droite.

```
static Arbre maximum(Arbre a) {  
    if (a==null) return null;  
    while (a.droite!=null) a=a.droite;  
    return a;  
}
```

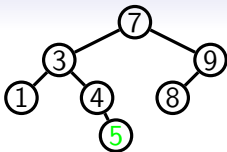
- Symétriquement pour minimum.

## Supprimer une clé



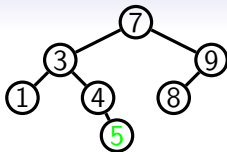
- 3 niveaux de difficulté:
  - ▶ Supprimer une feuille
  - ▶ Supprimer un sommet qui n'a qu'un fils
  - ▶ Supprimer un sommet qui a deux fils

## Supprimer une clé



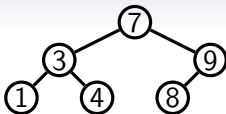
- 3 niveaux de difficulté:
  - ▶ Supprimer une feuille (exemple:  $k = 5$ ):
  - ▶ Supprimer un sommet qui n'a qu'un fils
  - ▶ Supprimer un sommet qui a deux fils

## Supprimer une clé



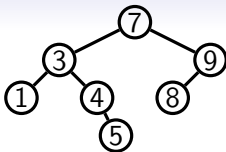
- 3 niveaux de difficulté:
  - ▶ Supprimer une feuille (exemple:  $k = 5$ ):
    - Trivial. Il suffit de l'enlever.
  - ▶ Supprimer un sommet qui n'a qu'un fils
  - ▶ Supprimer un sommet qui a deux fils

## Supprimer une clé



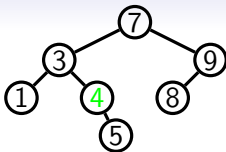
- 3 niveaux de difficulté:
  - ▶ Supprimer une feuille (exemple:  $k = 5$ ):
    - Trivial. Il suffit de l'enlever.
  - ▶ Supprimer un sommet qui n'a qu'un fils
  - ▶ Supprimer un sommet qui a deux fils

## Supprimer une clé



- 3 niveaux de difficulté:
  - ▶ Supprimer une feuille (exemple:  $k = 5$ ):
    - Trivial. Il suffit de l'enlever.
  - ▶ Supprimer un sommet qui n'a qu'un fils
  
  - ▶ Supprimer un sommet qui a deux fils

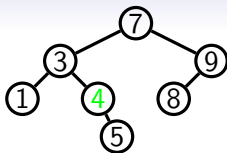
## Supprimer une clé



### ■ 3 niveaux de difficulté:

- ▶ Supprimer une feuille (exemple:  $k = 5$ ):
  - Trivial. Il suffit de l'enlever.
- ▶ Supprimer un sommet qui n'a qu'un fils (exemple:  $k = 4$ ).
- ▶ Supprimer un sommet qui a deux fils

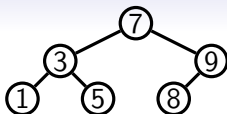
## Supprimer une clé



### ■ 3 niveaux de difficulté:

- ▶ Supprimer une feuille (exemple:  $k = 5$ ):
  - Trivial. Il suffit de l'enlever.
- ▶ Supprimer un sommet qui n'a qu'un fils (exemple:  $k = 4$ ).
  - Facile. Il suffit de l'enlever, et de faire de l'unique fils un fils du père du sommet supprimé = remonter l'unique fils d'un niveau.
- ▶ Supprimer un sommet qui a deux fils

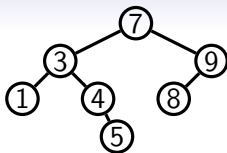
## Supprimer une clé



### ■ 3 niveaux de difficulté:

- ▶ Supprimer une feuille (exemple:  $k = 5$ ):
  - Trivial. Il suffit de l'enlever.
- ▶ Supprimer un sommet qui n'a qu'un fils (exemple:  $k = 4$ ).
  - Facile. Il suffit de l'enlever, et de faire de l'unique fils un fils du père du sommet supprimé = remonter l'unique fils d'un niveau.
- ▶ Supprimer un sommet qui a deux fils

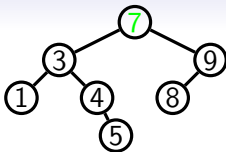
## Supprimer une clé



### ■ 3 niveaux de difficulté:

- ▶ Supprimer une feuille (exemple:  $k = 5$ ):
  - Trivial. Il suffit de l'enlever.
- ▶ Supprimer un sommet qui n'a qu'un fils (exemple:  $k = 4$ ).
  - Facile. Il suffit de l'enlever, et de faire de l'unique fils un fils du père du sommet supprimé = remonter l'unique fils d'un niveau.
- ▶ Supprimer un sommet qui a deux fils

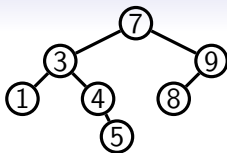
## Supprimer une clé



### ■ 3 niveaux de difficulté:

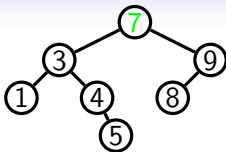
- ▶ Supprimer une feuille (exemple:  $k = 5$ ):
  - Trivial. Il suffit de l'enlever.
- ▶ Supprimer un sommet qui n'a qu'un fils (exemple:  $k = 4$ ).
  - Facile. Il suffit de l'enlever, et de faire de l'unique fils un fils du père du sommet supprimé = remonter l'unique fils d'un niveau.
- ▶ Supprimer un sommet qui a deux fils (exemple:  $k = 7$ ).
  - Moins facile.

## Le cas moins facile: supprimer un sommet $s$ d'arité 2



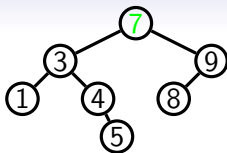
- Supprimer un sommet  $s$  qui a deux fils

## Le cas moins facile: supprimer un sommet $s$ d'arité 2



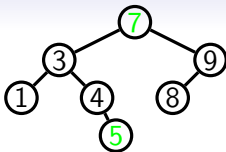
- Supprimer un sommet  $s$  qui a deux fils (exemple:  $k = 7$ ).

## Le cas moins facile: supprimer un sommet $s$ d'arité 2



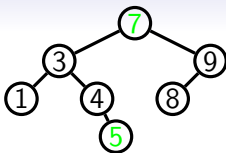
- Supprimer un sommet  $s$  qui a deux fils (exemple:  $k = 7$ ).
  - ▶ On cherche l'élément  $g$  de clé maximum dans son sous-arbre gauche

## Le cas moins facile: supprimer un sommet $s$ d'arité 2



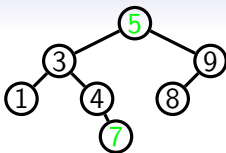
- Supprimer un sommet  $s$  qui a deux fils (exemple:  $k = 7$ ).
  - ▶ On cherche l'élément  $g$  de clé maximum dans son sous-arbre gauche (sur l'exemple:  $g$  est l'élément de clé 5).

## Le cas moins facile: supprimer un sommet $s$ d'arité 2



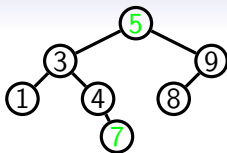
- Supprimer un sommet  $s$  qui a deux fils (exemple:  $k = 7$ ).
  - ▶ On cherche l'élément  $g$  de clé maximum dans son sous-arbre gauche (sur l'exemple:  $g$  est l'élément de clé 5).
  - ▶ On échange le contenu du sommet  $s$  par le contenu du sommet  $g$ .

## Le cas moins facile: supprimer un sommet $s$ d'arité 2



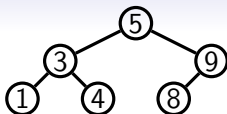
- Supprimer un sommet  $s$  qui a deux fils (exemple:  $k = 7$ ).
  - ▶ On cherche l'élément  $g$  de clé maximum dans son sous-arbre gauche (sur l'exemple:  $g$  est l'élément de clé 5).
  - ▶ On échange le contenu du sommet  $s$  par le contenu du sommet  $g$ .

## Le cas moins facile: supprimer un sommet $s$ d'arité 2



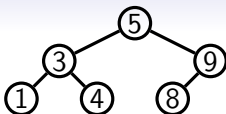
- Supprimer un sommet  $s$  qui a deux fils (exemple:  $k = 7$ ).
  - ▶ On cherche l'élément  $g$  de clé maximum dans son sous-arbre gauche (sur l'exemple:  $g$  est l'élément de clé 5).
  - ▶ On échange le contenu du sommet  $s$  par le contenu du sommet  $g$ .
  - ▶ On supprime  $g$ .

## Le cas moins facile: supprimer un sommet $s$ d'arité 2



- Supprimer un sommet  $s$  qui a deux fils (exemple:  $k = 7$ ).
  - ▶ On cherche l'élément  $g$  de clé maximum dans son sous-arbre gauche (sur l'exemple:  $g$  est l'élément de clé 5).
  - ▶ On échange le contenu du sommet  $s$  par le contenu du sommet  $g$ .
  - ▶ On supprime  $g$ .

## Le cas moins facile: supprimer un sommet $s$ d'arité 2



- Supprimer un sommet  $s$  qui a deux fils (exemple:  $k = 7$ ).
  - ▶ On cherche l'élément  $g$  de clé maximum dans son sous-arbre gauche (sur l'exemple:  $g$  est l'élément de clé 5).
  - ▶ On échange le contenu du sommet  $s$  par le contenu du sommet  $g$ .
  - ▶ On supprime  $g$ .
- Remarques:
  - ▶  $g$  n'a pas de fils droit (sinon, il ne serait pas maximum).
  - ▶  $g$  est donc dans le cas trivial, ou facile, si on veut le supprimer.

# Code complet en JAVA

- Solution récursive (persistante).

- ▶ On recherche le sommet correspondant à une clé:

```
static Arbre supprimer(int k, Arbre a) {  
    if (a==null) return a;  
    if (k==a.k) return supprimerRacine(a);  
    else if (k<a.k) return  
        new Arbre (supprimer(k,a.gauche),a.k,a.D,a.droite);  
    else return  
        new Arbre(a.gauche,a.k,a.D,supprimer(k,a.droite));}
```

- ▶ On supprime la racine du sous-arbre correspondant:

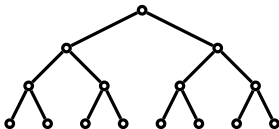
```
static Arbre supprimerRacine(Arbre a) {  
    // Cas trivial & Facile  
    if (a.gauche==null) return a.droite;  
    if (a.droite==null) return a.gauche;  
    // Cas moins facile  
    Arbre g= maximum(a.gauche);  
    return  
        new Arbre(supprimer(g.k,a.gauche),g.k,g.D,a.droite);}
```

# Complexité des algorithmes

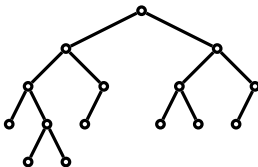
- Si  $h$  est la hauteur d'un arbre binaire de recherche, et  $n$  le nombre de sommets
  - ▶ insérer, rechercher, supprimer, maximum, minimum sont en  $O(h)$ .

## Arbres avec $h$ de l'ordre de $\log n$

- Un arbre (très bien) équilibré:  $h = \log n$ .

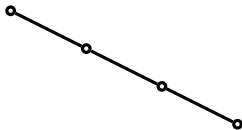


- Un arbre à peu près équilibré:  $h \leq 2 \log n$ .

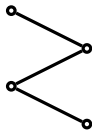


## Arbres avec $h$ de l'ordre de $n$

- Un arbre très mal équilibré:  $h = n - 1$ .

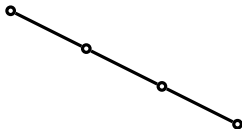


- Un autre arbre très mal équilibré:  $h = n - 1$ .



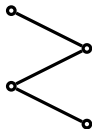
## Arbres avec $h$ de l'ordre de $n$

- Un arbre très mal équilibré:  $h = n - 1$ .



- ▶ On obtient un arbre comme cela si l'on insère  $1, 2, 3, \dots, n$  dans cet ordre.

- Un autre arbre très mal équilibré:  $h = n - 1$ .



# Complexité des algorithmes

- Si  $h$  est la hauteur d'un arbre binaire de recherche, et  $n$  le nombre de sommets
  - ▶ insérer, rechercher, supprimer, maximum, minimum sont en  $O(h)$ .
  - ▶ Donc au pire cas en  $O(n)$ .
- Pour garantir un pire cas en  $O(\log n)$ , on utilise des arbres *équilibrés*: arbres AVL, arbres a-b, arbres rouge et noir, ...

## En fait:

- La hauteur moyenne d'un arbre à  $n$  sommets est  $O(\sqrt{n})$ , si on suppose tous les arbres binaires équiprobables.
- Mais plusieurs suites d'insertions peuvent produire les mêmes arbres: par exemple, 2, 1, 3 et 2, 3, 1 produisent l'arbre de hauteur 1 de racine 2.
- La hauteur moyenne d'un arbre à  $n$  sommets obtenu par insertion de  $1, 2, \dots, n$ , dans tous les ordres possibles, ces ordres étant équiprobables est  $O(\log n)$ .
- On peut donc considérer que les algorithmes précédents se comportent en  $O(\log n)$ , c'est-à-dire bien, en moyenne. . .

... mais parfois/avec un pire cas en  $O(n)$ .

- Les arbres équilibrés permettent de garantir un pire cas et une moyenne en  $O(\log n)$ .

# Résumé

## ■ Complexité en moyenne:

	rechercher	insérer	supprimer	minimum	maximum
tableau/liste	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
dichotomie	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
hachage	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$
arbre de recherche	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
arbre équilibré	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
tas	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$

## ■ Complexité au pire cas:

tableau/liste	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
dichotomie	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
hachage	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
arbre de recherche	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
arbre équilibré	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
tas	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$

# Aujourd'hui

Dictionnaires, Files de priorité

Arbres binaires de recherche

**Arbres AVL**

Les tas

# Principe des arbres AVL

- Un arbre AVL (Adelson-Velskii et Landis) est
  1. un arbre binaire de recherche
  2. tel qu'en tout sommet, la hauteur du fils droit et du fils gauche diffèrent *au plus* d'un.
  
- Intérêt majeur:
  - ▶ Cette propriété garantie que la hauteur d'un arbre AVL est en  $O(\log n)$ .

## Pourquoi?

- Plus précisément, cela découle de

$$\log_2(1 + n) \leq 1 + h \leq 1.44 \log_2(2 + n).$$

## Pourquoi?

- Plus précisément, cela découle de

$$\log_2(1 + n) \leq 1 + h \leq 1.44 \log_2(2 + n).$$

- La borne  $\log_2(1 + n) \leq 1 + h$  est vraie pour tout arbre binaire (cf cours 2): un arbre binaire de hauteur  $h$  a au plus  $2^{h+1} - 1$  sommets.

# Pourquoi?

- Plus précisément, cela découle de

$$\log_2(1 + n) \leq 1 + h \leq 1.44 \log_2(2 + n).$$

- La borne  $\log_2(1 + n) \leq 1 + h$  est vraie pour tout arbre binaire (cf cours 2): un arbre binaire de hauteur  $h$  a au plus  $2^{h+1} - 1$  sommets.
- Pour l'autre borne, on considère le nombre minimal  $N(h)$  de sommets dans un arbre AVL de hauteur  $h$ .
  - ▶ Hauteur 0:  $N(0) = 1$ .
  - ▶ Hauteur 1:  $N(1) = 2$ .
  - ▶ Hauteur  $h$ ,  $h \geq 2$ : une racine + un sous-arbre de hauteur  $h - 1$  + un sous-arbre de hauteur  $h - 2$ .

$$N(h) = 1 + N(h - 1) + N(h - 2).$$



■ Posons  $F(h) = 1 + N(h)$ ,

- ▶ On a  $F(0) = 2$ ,  $F(1) = 3$ , et
- ▶  $F(h) = F(h-1) + F(h-2)$ , pour  $h \geq 2$ .

■ C'est la suite de Fibonacci "décalée".

■ On trouve

$$F(h) = \frac{1}{\sqrt{5}}(\phi^{h+3} - \phi^{-(h+3)}),$$

où  $\phi = \frac{1+\sqrt{5}}{2}$  est le nombre d'or.

■ On a donc  $1 + n \geq F(h) > \frac{1}{\sqrt{5}}(\phi^{h+3} - 1)$  ce qui donne

$$h + 3 < \log_{\phi}(\sqrt{5}(2 + n)) < \frac{\log_2(2 + n)}{\log_2(\phi)} + 2.$$

■ On observe que  $\frac{1}{\log_2(\phi)} \leq 1.44$ .

## Garder de la hauteur

```
class Arbre {
  int k; int D;
  Arbre gauche, droite;
  int hauteur;

  static int hauteur(Arbre a) {
    return a==null? -1:a.hauteur;
  }

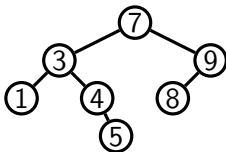
  Arbre (Arbre gauche, int k, int D, Arbre droite) {
    this.gauche = gauche;
    this.k = k; this.D = D;
    this.droite = droite;

    hauteur=1+ Math.max(hauteur(gauche),hauteur(droite));}}}
```

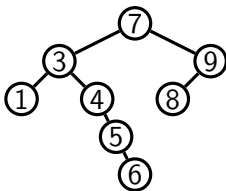
- Petite astuce JAVA: la fonction **static** hauteur renvoie soit la hauteur, soit  $-1$  si l'arbre est vide.

# Comment garder l'équilibre?

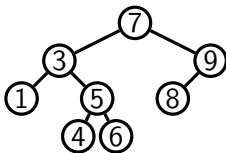
- Partant de:



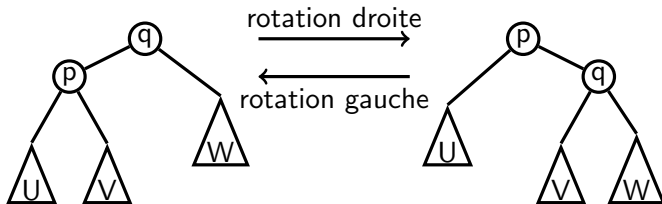
- Après avoir inséré un élément de clé 6 ...



- ..., on rétablit l'équilibre, par une *rotation*.



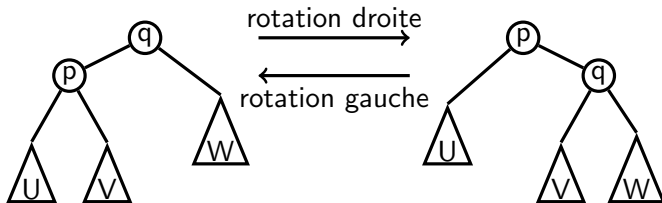
## Le concept de rotation



### ■ Formellement:

- ▶ Rotation droite: A partir de  $((U, p, V), q, W)$ , on construit  $(U, p, (V, q, W))$ .
  - ▶ Rotation gauche: A partir de  $(U, p, (V, q, W))$ , on construit  $((U, p, V), q, W)$ .
- $U, V, W$  sont des sous-arbres, qui peuvent être vides.  
Lorsqu'on fait une rotation droite (respectivement gauche), on suppose que l'arbre possède un fils gauche (resp. droit).

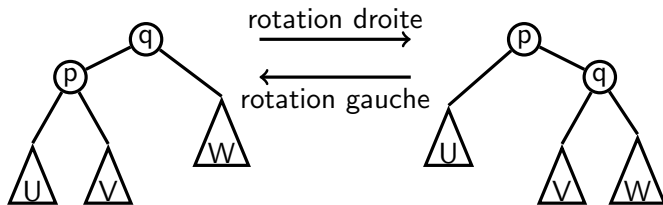
## Rotation en JAVA



```
static Arbre rotationdroite(Arbre a) {  
    return new Arbre(a.gauche.gauche,a.gauche.k,a.gauche.D,  
        new Arbre(a.gauche.droite,a.k,a.D,a.droite));  
}
```

```
static Arbre rotationgauche(Arbre a) {  
    return new Arbre(new Arbre(a.gauche,a.k,a.D, a.droite.gauche),  
        a.droite.k,a.droite.D,a.droite.droite);  
}
```

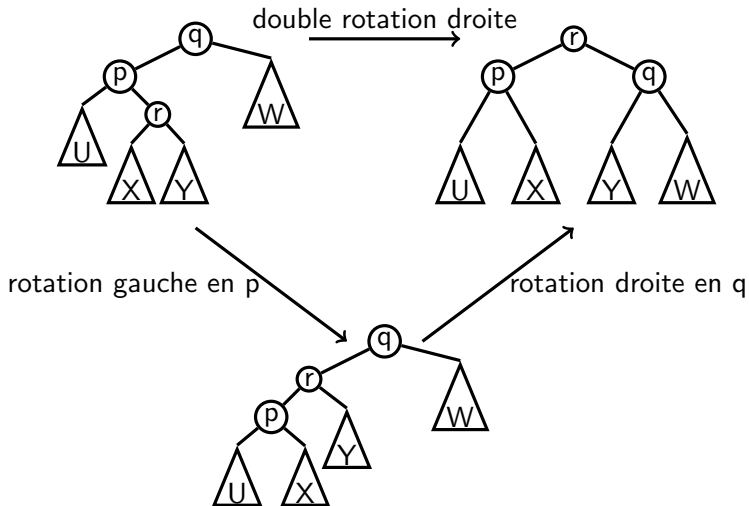
# Propriétés des rotations



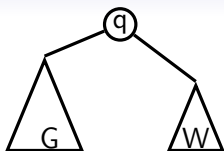
- Propriété 1: les rotations préservent l'ordre infixe, et donc la propriété d'arbre binaire de recherche.
- Par contre, elles modifient la hauteur de certains sous-arbres de 1 ou  $-1$ .

## Double rotation droite

- On définit une double rotation droite comme une rotation gauche en le fils gauche, puis une rotation droite.

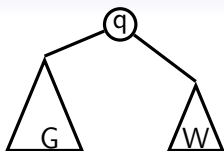


## Petit raisonnement



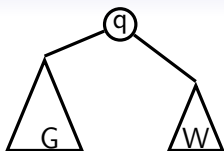
- Une insertion ou une suppression perturbe d'au plus 1 la hauteur du fils droit ou du fils gauche d'un sommet  $q$ .

## Petit raisonnement



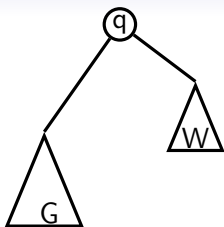
- Une insertion ou une suppression perturbe d'au plus 1 la hauteur du fils droit ou du fils gauche d'un sommet  $q$ .
- Si  $q$  n'est plus équilibré, c'est donc que  $|hauteur(G) - hauteur(W)| = 2$ .

## Petit raisonnement



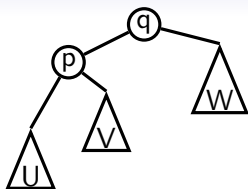
- Une insertion ou une suppression perturbe d'au plus 1 la hauteur du fils droit ou du fils gauche d'un sommet  $q$ .
- Si  $q$  n'est plus équilibré, c'est donc que  $|hauteur(G) - hauteur(W)| = 2$ .
- Supposons que  $hauteur(G) = hauteur(W) + 2$  (l'autre cas s'obtient par symétrie).

## Petit raisonnement



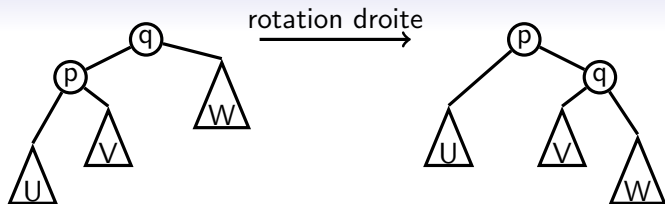
- Une insertion ou une suppression perturbe d'au plus 1 la hauteur du fils droit ou du fils gauche d'un sommet  $q$ .
- Si  $q$  n'est plus équilibré, c'est donc que  $|hauteur(G) - hauteur(W)| = 2$ .
- Supposons que  $hauteur(G) = hauteur(W) + 2$  (l'autre cas s'obtient par symétrie).

## Petit raisonnement



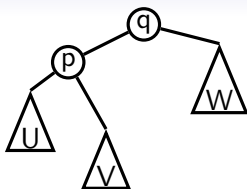
- Une insertion ou une suppression perturbe d'au plus 1 la hauteur du fils droit ou du fils gauche d'un sommet  $q$ .
- Si  $q$  n'est plus équilibré, c'est donc que  $|hauteur(G) - hauteur(W)| = 2$ .
- Supposons que  $hauteur(G) = hauteur(W) + 2$  (l'autre cas s'obtient par symétrie).
- Si  $hauteur(U) > hauteur(V)$ , il suffit de faire un rotation droite en  $q$ .

## Petit raisonnement



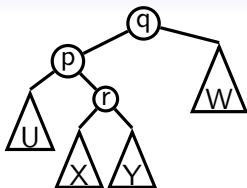
- Une insertion ou une suppression perturbe d'au plus 1 la hauteur du fils droit ou du fils gauche d'un sommet  $q$ .
- Si  $q$  n'est plus équilibré, c'est donc que  $|hauteur(G) - hauteur(W)| = 2$ .
- Supposons que  $hauteur(G) = hauteur(W) + 2$  (l'autre cas s'obtient par symétrie).
- Si  $hauteur(U) > hauteur(V)$ , il suffit de faire un rotation droite en  $q$ .

## Petit raisonnement



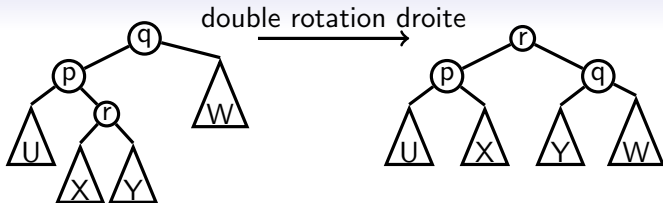
- Une insertion ou une suppression perturbe d'au plus 1 la hauteur du fils droit ou du fils gauche d'un sommet  $q$ .
- Si  $q$  n'est plus équilibré, c'est donc que  $|hauteur(G) - hauteur(W)| = 2$ .
- Supposons que  $hauteur(G) = hauteur(W) + 2$  (l'autre cas s'obtient par symétrie).
- Si  $hauteur(U) > hauteur(V)$ , il suffit de faire un rotation droite en  $q$ .
- Si  $hauteur(U) \leq hauteur(V)$ , il suffit de faire une double rotation droite en  $q$  (= une rotation gauche en  $p$ , puis droite en  $q$ ).

## Petit raisonnement



- Une insertion ou une suppression perturbe d'au plus 1 la hauteur du fils droit ou du fils gauche d'un sommet  $q$ .
- Si  $q$  n'est plus équilibré, c'est donc que  $|hauteur(G) - hauteur(W)| = 2$ .
- Supposons que  $hauteur(G) = hauteur(W) + 2$  (l'autre cas s'obtient par symétrie).
- Si  $hauteur(U) > hauteur(V)$ , il suffit de faire un rotation droite en  $q$ .
- Si  $hauteur(U) \leq hauteur(V)$ , il suffit de faire une double rotation droite en  $q$  (= une rotation gauche en  $p$ , puis droite en  $q$ ).

## Petit raisonnement



- Une insertion ou une suppression perturbe d'au plus 1 la hauteur du fils droit ou du fils gauche d'un sommet  $q$ .
- Si  $q$  n'est plus équilibré, c'est donc que  $|hauteur(G) - hauteur(W)| = 2$ .
- Supposons que  $hauteur(G) = hauteur(W) + 2$  (l'autre cas s'obtient par symétrie).
- Si  $hauteur(U) > hauteur(V)$ , il suffit de faire un rotation droite en  $q$ .
- Si  $hauteur(U) \leq hauteur(V)$ , il suffit de faire une double rotation droite en  $q$  (= une rotation gauche en  $p$ , puis droite en  $q$ ).

# En JAVA

```
static Arbre équilibrer(Arbre a) {  
    a.hauteur = 1 + Math.max(hauteur(a.gauche), hauteur(a.droite));  
  
    if (hauteur(a.gauche) - hauteur(a.droite) == 2) {  
        if (hauteur(a.gauche.gauche) ≤ hauteur(a.gauche.droite))  
            a.gauche = rotationgauche(a.gauche);  
        return rotationdroite(a);}  
  
    if (hauteur(a.gauche) - hauteur(a.droite) == -2) {  
        if (hauteur(a.droite.droite) ≤ hauteur(a.droite.gauche))  
            a.droite = rotationdroite(a.droite);  
        return rotationgauche(a);}  
    return a;}  
}
```

# Insérer

```
static Arbre insérer(int k, int D, Arbre a) {  
    if (a==null)  
        return new Arbre(null,k,D,null);  
    else if (k<a.k) // On insère à gauche  
        a=new Arbre(insérer(k,D,a.gauche),a.k,a.D,a.droite);  
    else if (k>a.k) // On insère à droite  
        a= new Arbre(a.gauche,a.k,a.D,insérer(k,D,a.droite));  
    return équilibrer(a);  
}
```

- On a simplement ajouté un appel à la méthode équilibrer avant de renvoyer.

# Supprimer

```
static Arbre supprimer(int k, Arbre a) {  
    if (a==null) return a;  
    if (k==a.k) return supprimerRacine(a);  
    else if (k<a.k) a=  
        new Arbre (supprimer(k,a.gauche),a.k,a.D,a.droite);  
    else a=  
        new Arbre(a.gauche,a.k,a.D,supprimer(k,a.droite));  
    return équilibrer(a);}
```

```
static Arbre supprimerRacine(Arbre a) {  
    // Cas trivial & Facile  
    if (a.gauche==null) return a.droite;  
    if (a.droite==null) return a.gauche;  
    // Cas moins facile  
    Arbre g= maximum(a.gauche);  
    a= new Arbre(supprimer(g.k,a.gauche),g.k,g.D,a.droite);  
    return équilibrer(a);}
```

- Même remarque.

# Complexité

- La complexité de chacune des procédures est majorée par la hauteur de l'arbre.
- Remarque: les rotations se font en temps constant.
- Bilan:
  - ▶ insérer, rechercher, supprimer, maximum, minimum s'effectuent en temps  $O(\log n)$ .
- Remarque: rééquilibrer un arbre
  - ▶ après une insertion, en fait une seule rotation, ou double rotation suffit.
  - ▶ après une suppression, il peut y avoir jusqu'à  $h$  rotations ou double rotations.

# Résumé

## ■ Complexité en moyenne:

	rechercher	insérer	supprimer	minimum	maximum
tableau/liste	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
dichotomie	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
hachage	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$
arbre de recherche	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
arbre équilibré	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
tas	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$

## ■ Complexité au pire cas:

tableau/liste	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
dichotomie	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
hachage	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
arbre de recherche	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
arbre équilibré	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
tas	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$

# Aujourd'hui

Dictionnaires, Files de priorité

Arbres binaires de recherche

Arbres AVL

**Les tas**

# Les tas

- Les *tas* constituent une structure de données optimisée pour la gestion des files de priorité.
- Opérations supportées efficacement:
  - ▶ `insérer(k,D)`:  $O(\log n)$ .
  - ▶ `maximum()`:  $O(1)$ .
  - ▶ `supprimer(clé(maximum()))`:  $O(\log n)$ .
- $n$  est le nombre d'éléments.
- En fait, les tas sont souvent utilisés comme briques élémentaires dans d'autres algorithmes plus complexes (ex: compression de Huffman & poly).

# Tri par tas

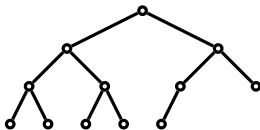
```
static void TriParTas (int tab[]) {  
    Tas t= new Tas(tab.length);  
  
    for (int i=0; i<tab.length;i++)  
        t.insérer(tab[i]);  
  
    for (int i=tab.length-1; i>=0;i--) {  
        tab[i]=t.maximum();  
        t.supprimerMaximum();  
    }  
}}
```

- Le tri par tas est parmi les tris optimaux en  $O(n \log n)$  comparaisons.

# Définition d'un arbre tassé

- Un arbre binaire est dit *tassé* si
  - ▶ tous les niveaux sauf peut-être le dernier sont complets.
  - ▶ le dernier est "rempli à gauche".

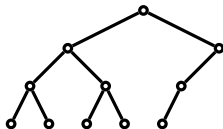
- Exemple:



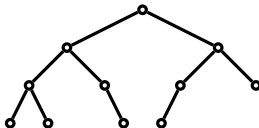
- Un arbre tassé à  $n$  sommets est de hauteur  $\lfloor \log_2 n \rfloor$ .

# Contre-exemples

- Le niveau 2 n'est pas plein.

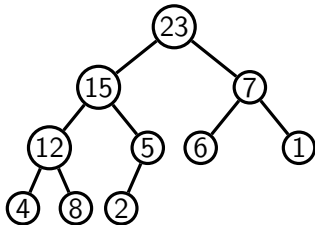


- Le niveau 3 n'est pas "rempli à gauche"



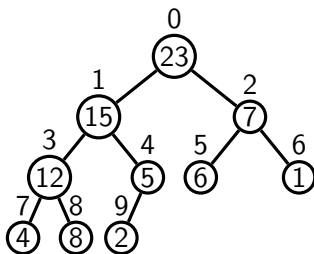
# Un tas

- Un *tas* est un arbre binaire tassé,
  - ▶ dont les sommets sont étiquetés par des clés;
  - ▶ tel que tout sommet possède un clé supérieure ou égale aux clés de ses fils.



## Tas et tableaux

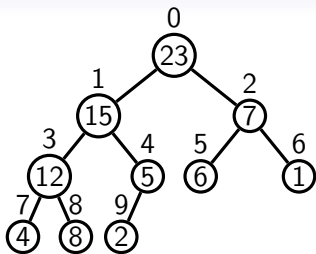
- Un tas se représente naturellement par un tableau.



- On numérote les sommets par un parcours en largeur, de gauche à droite.
- Le sommet  $i$  est rangé dans la case d'indice  $i$  du tableau.

	0	1	2	3	4	5	6	7	8	9
tab	23	15	7	12	5	6	1	4	8	2

## Relations de filiation

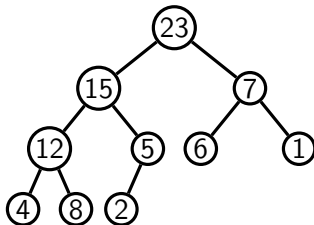


	0	1	2	3	4	5	6	7	8	9
tab	23	15	7	12	5	6	1	4	8	2

racine	:	sommet 0
parent du sommet $i$	:	sommet $\lfloor (i - 1)/2 \rfloor$
fils gauche du sommet $i$	:	sommet $2i + 1$
fils droit du sommet $i$	:	sommet $2i + 2$
sommet $i$ est une feuille	:	$2i + 1 \geq n$
sommet $i$ a un fils droit	:	$2i + 2 < n$

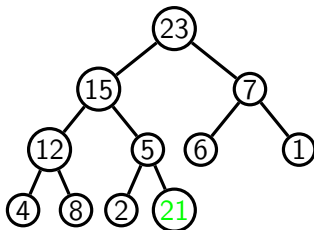
# Insérer dans un tas

- Pour insérer la clé  $v$  dans un tas.
  - ▶ On insère la clé  $v$  à la fin du dernier niveau de l'arbre (à la fin du tableau).
  - ▶ Tant que la clé du père de  $v$  est plus petite que la clé  $v$ :
    - on échange la clé du père de  $v$  avec la clé  $v$ .
- Exemple: insertion de 21.



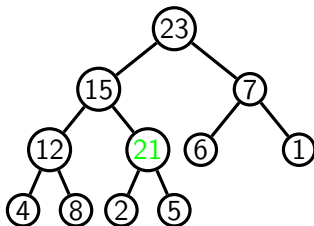
## Insérer dans un tas

- Pour insérer la clé  $v$  dans un tas.
  - ▶ On insère la clé  $v$  à la fin du dernier niveau de l'arbre (à la fin du tableau).
  - ▶ Tant que la clé du père de  $v$  est plus petite que la clé  $v$ :
    - on échange la clé du père de  $v$  avec la clé  $v$ .
- Exemple: insertion de 21.



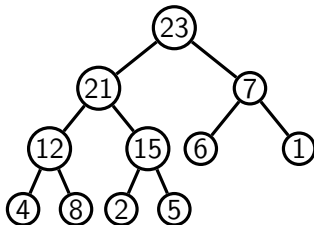
# Insérer dans un tas

- Pour insérer la clé  $v$  dans un tas.
  - ▶ On insère la clé  $v$  à la fin du dernier niveau de l'arbre (à la fin du tableau).
  - ▶ Tant que la clé du père de  $v$  est plus petite que la clé  $v$ :
    - on échange la clé du père de  $v$  avec la clé  $v$ .
- Exemple: insertion de 21.



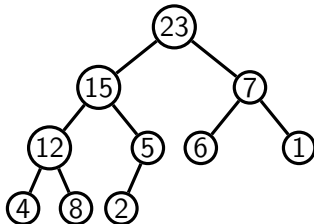
# Insérer dans un tas

- Pour insérer la clé  $v$  dans un tas.
  - ▶ On insère la clé  $v$  à la fin du dernier niveau de l'arbre (à la fin du tableau).
  - ▶ Tant que la clé du père de  $v$  est plus petite que la clé  $v$ :
    - on échange la clé du père de  $v$  avec la clé  $v$ .
- Exemple: insertion de 21.



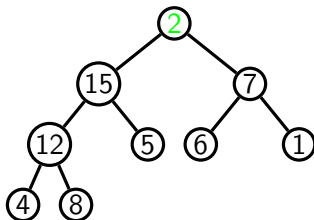
# Supprimer le maximum

- La clé maximum se lit à la racine.
- Pour enlever la racine:
  - ▶ On remplace la racine par le dernier élément  $v$  (à la fin du tableau).
  - ▶ Tant que la clé de  $v$  est inférieure à celle de l'un de ses fils:
    - on échange la clé de  $v$  avec celle du plus grand de ses fils.
- Exemple: suppression du maximum.



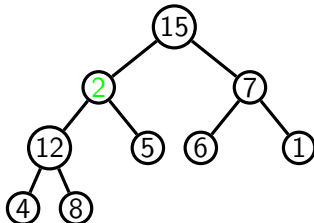
# Supprimer le maximum

- La clé maximum se lit à la racine.
- Pour enlever la racine:
  - ▶ On remplace la racine par le dernier élément  $v$  (à la fin du tableau).
  - ▶ Tant que la clé de  $v$  est inférieure à celle de l'un de ses fils:
    - on échange la clé de  $v$  avec celle du plus grand de ses fils.
- Exemple: suppression du maximum.



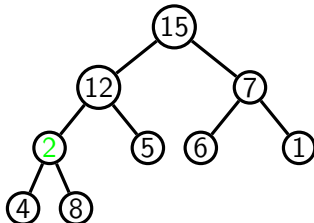
# Supprimer le maximum

- La clé maximum se lit à la racine.
- Pour enlever la racine:
  - ▶ On remplace la racine par le dernier élément  $v$  (à la fin du tableau).
  - ▶ Tant que la clé de  $v$  est inférieure à celle de l'un de ses fils:
    - on échange la clé de  $v$  avec celle du plus grand de ses fils.
- Exemple: suppression du maximum.



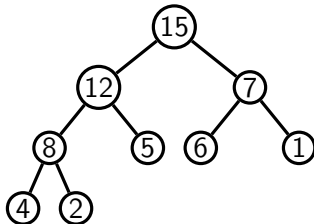
# Supprimer le maximum

- La clé maximum se lit à la racine.
- Pour enlever la racine:
  - ▶ On remplace la racine par le dernier élément  $v$  (à la fin du tableau).
  - ▶ Tant que la clé de  $v$  est inférieure à celle de l'un de ses fils:
    - on échange la clé de  $v$  avec celle du plus grand de ses fils.
- Exemple: suppression du maximum.



# Supprimer le maximum

- La clé maximum se lit à la racine.
- Pour enlever la racine:
  - ▶ On remplace la racine par le dernier élément  $v$  (à la fin du tableau).
  - ▶ Tant que la clé de  $v$  est inférieure à celle de l'un de ses fils:
    - on échange la clé de  $v$  avec celle du plus grand de ses fils.
- Exemple: suppression du maximum.



# Complexité des opérations

- La complexité des opérations d'insertion et de suppression du maximum est majorée par la hauteur de l'arbre.
- Complexité:
  - ▶ `insérer(k,D)`:  $O(\log n)$ .
  - ▶ `maximum()`:  $O(1)$ .
  - ▶ `supprimer(clé(maximum()))`:  $O(\log n)$ .
- $n$  est le nombre d'éléments.

# Un tas en JAVA

```
class Tas {  
  
    private int [] t ;  
    // Tableau des sommets  
    private int n ;  
    // Nombre de sommets  
  
    Tas(int sz) {  
        t = new int [sz+1] ;  
        n = 0 ;}  
}
```

```
int maximum() {  
    return t[0];  
}
```

```
int père(int i) {  
    return (i-1)/2;}  
  
int gauche(int i) {  
    return 2 * i + 1;}  
  
int droite(int i) {  
    return 2 * i + 2;}  
  
boolean estUneFeuille(int i) {  
    return (2 * i + 1  $\geq$  n);}  
  
boolean aUnFilsDroit(int i) {  
    return (2 * i + 2 < n);}  
  
...}
```

## Insérer en JAVA

```
void insérer(int v) {  
    int i = n;  
    n++;  
  
    while (i > 1 && t[père(i)] ≤ v) {  
        t[i] = t[père(i)] ;  
        i = père(i);  
    }  
  
    t[i] = v;}
```

- En fait, on n'échange pas le père de  $v$  avec  $v$ , mais on "descend" le père de  $v$ , et  $v$  est mis directement à sa place finale.

## Supprimer le maximum en JAVA

```
void supprimerMaximum() {
    n--;
    int v = t[n];
    int i = 0;
    while (!estUneFeuille(i)) {
        int j = gauche(i);
        if (aUnFilsDroit(i) && t[droite(i)] > t[gauche(i)])
            j = droite(i);
        if (v >= t[j])
            break;
        t[i] = t[j];
        i = j;
    }
    t[i] = v;}
}
```

- Même astuce.

# Résumé

## ■ Complexité en moyenne:

	rechercher	insérer	supprimer	minimum	maximum
tableau/liste	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
dichotomie	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
hachage	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$
arbre de recherche	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
arbre équilibré	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
tas	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$

## ■ Complexité au pire cas:

tableau/liste	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
dichotomie	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
hachage	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
arbre de recherche	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
arbre équilibré	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
tas	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$