Plan du cours

1/ Rappels et compléments Java.

2/ Tableaux, boucles et invariants.

3/ Notions élémentaires de complexité.

4/ Récursion.

5/ Structures de données et introduction aux types abstraits de données. **ICI**

6/ Quelques compléments Java.

Structures déjà vues

- Tableau, File, Liste, Arbre, Tas et aujourd'hui `quelques arbres spécifiques` .

Arbres spécifiques

Observation

L'efficacité des algorithmes sur les arbres binaires quelconques peut se dégrader rapidement suivant la forme de cet arbre : une situation `extrême` est celle d'un arbre binaire ordonné dont chaque nœud n'a qu'un seul successeur droit ou gauche. L'arbre se comporte alors comme une `liste` et les algorithmes avec un temps d'exécution normalement `logarithmique` demanderait un temps linéaire.

Arbres spécifiques

Observation

L'efficacité des algorithmes sur les arbres binaires quelconques peut se dégrader rapidement suivant la forme de cet arbre : une situation `extrême` est celle d'un arbre binaire ordonné dont chaque nœud n'a qu'un seul successeur droit ou gauche. L'arbre se comporte alors comme une `liste` et les algorithmes avec un temps d'exécution normalement `logarithmique` demanderait un temps linéaire.

Idée d'Adelson-Velskii et Landis

Pour une meilleure efficacité algorithmique, un arbre doit être équilibré! (cf. "*An algorithm for the organization of information*" dans **Dokl. Akademii Nauk SSSR, 146: 263–266, 1962**.

Observation

L'efficacité des algorithmes sur les arbres binaires quelconques peut se dégrader rapidement suivant la forme de cet arbre : une situation `extrême` est celle d'un arbre binaire ordonné dont chaque nœud n'a qu'un seul successeur droit ou gauche. L'arbre se comporte alors comme une `liste` et les algorithmes avec un temps d'exécution normalement `logarithmique` demanderait un temps linéaire.

Idée d'Adelson-Velskii et Landis

Pour une meilleure efficacité algorithmique, un arbre doit être équilibré! (cf. "*An algorithm for the organization of information*" dans **Dokl. Akademii Nauk SSSR, 146: 263–266, 1962**.

Définition : arbre AVL

Un `arbre binaire équilibré` ou arbre `AVL` (du nom des auteurs) est un arbre binaire tel que pour tout nœud de l'arbre les hauteurs de ses deux sous-arbres diffèrent d'au plus 1.

Une propriété “logarithmique”

Nous allons voir une propriété très importante d’un tel arbre (surtout dans le but de savoir en quoi l’équilibrage récursif est intéressant pour les algorithmes).

Une propriété “logarithmique”

Nous allons voir une propriété très importante d'un tel arbre (surtout dans le but de savoir en quoi l'équilibrage récursif est intéressant pour les algorithmes).

- Le nombre `minimal` f_h de nœuds internes d'un arbre AVL de hauteur h satisfait

$$f_0 = 0, f_1 = 1, f_2 = 2$$

et

$$f_h = 1 + f_{h-1} + f_{h-2}\,, \qquad h \geq 2\,.$$

Une propriété “logarithmique”

Nous allons voir une propriété très importante d'un tel arbre (surtout dans le but de savoir en quoi l'équilibrage récursif est intéressant pour les algorithmes).

- Le nombre `minimal` f_h de nœuds internes d'un arbre AVL de hauteur h satisfait

$$f_0 = 0, f_1 = 1, f_2 = 2$$

et

$$f_h = 1 + f_{h-1} + f_{h-2}\,, \qquad h \geq 2\,.$$

- C'est la même récurrence que Fibonacci (à un décalage près).

$$f_h \text{ croît comme } \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2}\right)^{h+2} \text{ (quand h } \to +\infty)\,.$$

Une propriété “logarithmique”

Nous allons voir une propriété très importante d'un tel arbre (surtout dans le but de savoir en quoi l'équilibrage récursif est intéressant pour les algorithmes).

- Le nombre `minimal` f_h de nœuds internes d'un arbre AVL de hauteur h satisfait

$$f_0 = 0, f_1 = 1, f_2 = 2$$

et

$$f_h = 1 + f_{h-1} + f_{h-2}\,, \qquad h \geq 2\,.$$

- C'est la même récurrence que Fibonacci (à un décalage près).

$$f_h \text{ croît comme } \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2}\right)^{h+2} \text{ (quand h } \to +\infty)\,.$$

- Réciproquement, la hauteur maximale h d'un arbre AVL peut-être bornée par

$$\frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2}\right)^{h+2} \leq 1$$

donc $\mathbf{h \leq 1.44 \log_2 n + 1}$. **Conséquence**: aller chercher une information dans une branche se fera toujours en temps `logarithmique` .

Algorithme de maintien et d'Ã©quilibrage

Idée sous-jacente

L'idée est **simple** mais `astucieux` . On maintient une sorte de `drapeau`. A chaque nœud de l'arbre, nous devons maintenir l'information indiquant lequel des sous-arbres est le plus haut. Ce drapeau peut prendre 3 valeurs :

- 0 indique que les deux sous-arbres du noeud ont même hauteur.
- -1 indique que le sous-arbre gauche est plus haut que le sous-arbre droit.
- 1 indique que le sous-arbre droit est plus haut que le sous-arbre gauche.

Algorithme de maintien et d'Ã©quilibrage

Idée sous-jacente

L'idée est **simple** mais `astucieux` . On maintient une sorte de `drapeau`. A chaque nœud de l'arbre, nous devons maintenir l'information indiquant lequel des sous-arbres est le plus haut. Ce drapeau peut prendre 3 valeurs :

- 0 indique que les deux sous-arbres du noeud ont même hauteur.
- -1 indique que le sous-arbre gauche est plus haut que le sous-arbre droit.
- 1 indique que le sous-arbre droit est plus haut que le sous-arbre gauche.

Le code

```
class NoeudAVL {
        private NoeudAVL gauche, droit;
        private Objet élément;  /⋆ attention: il faut un ordre sur les éléments ⋆/
        private int equilibre;
        public NoeudAVL() {
                gauche = null;
                droit = null;
                equilibre = 0;
}
        ...
```

Algorithme (suite)

Supposons que le nœud *A* était équilibré. Lors de l'insertion dans le sous-arbre gauche deux cas peuvent se présenter:

- la hauteur du sous-arbre gauche n'augmente pas: l'arbre reste donc **équilibré**.
- la hauteur augmente. Trois sous-cas arrivent:

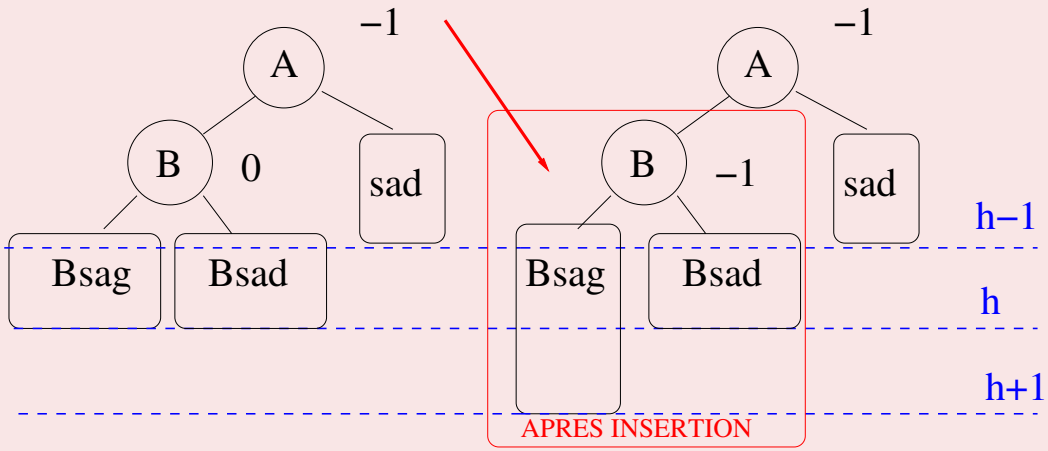
Algorithme (suite)

Supposons que le nœud *A* était équilibré. Lors de l'insertion dans le sous-arbre gauche deux cas peuvent se présenter:

- la hauteur du sous-arbre gauche n'augmente pas: l'arbre reste donc **équilibré**.
- la hauteur augmente. Trois sous-cas arrivent:
  1. la hauteur du s-a-g était égale à la hauteur du s-a-d MOINS UN: le nouvel arbre est **encore mieux équilibré** qu'avant l'insertion;
  2. la hauteur du s-a-g était égale à la hauteur du s-a-d: on a encore l'**équilibre**;
  3. la hauteur du s-a-g était égale à la hauteur du s-a-d PLUS UN: il y a **déséquilibre**!

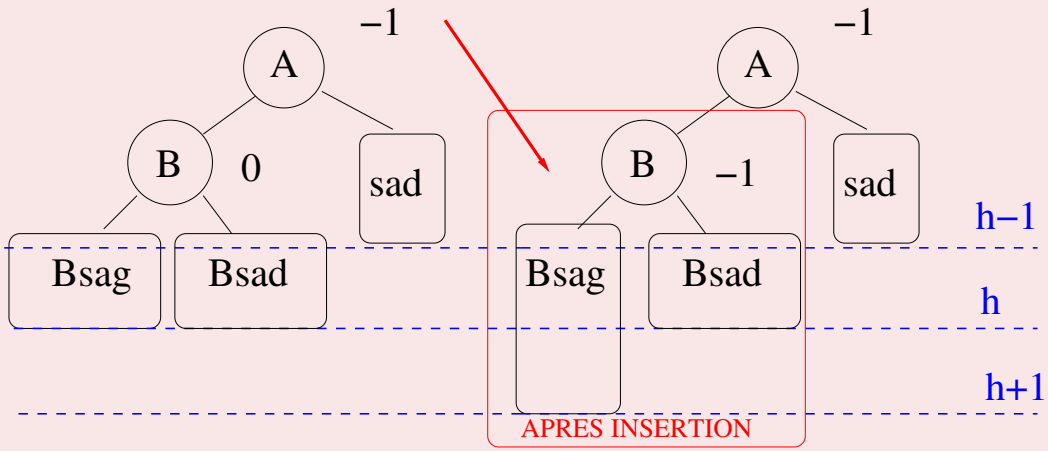
Algorithme (suite ...)

Exemple : le nœud B va pencher à gauche

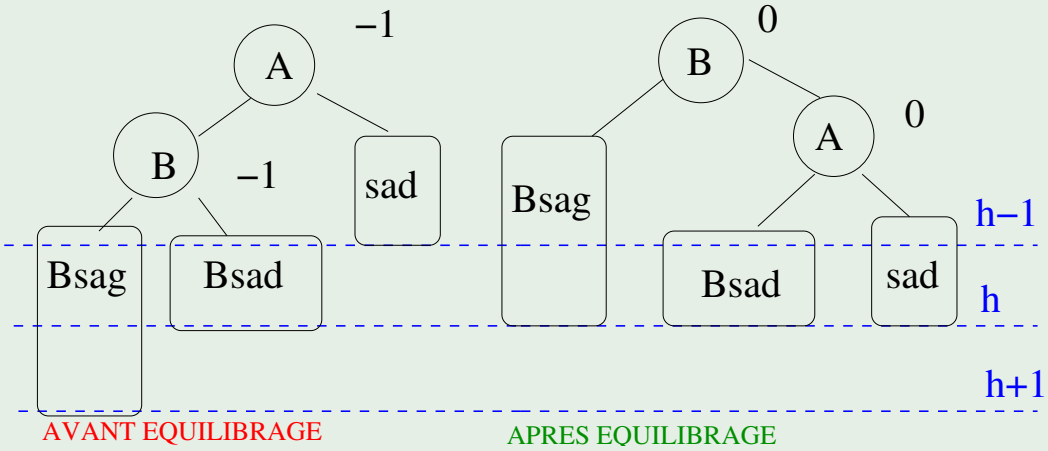


Algorithme (suite ...)

Exemple : le nœud B va pencher à gauche

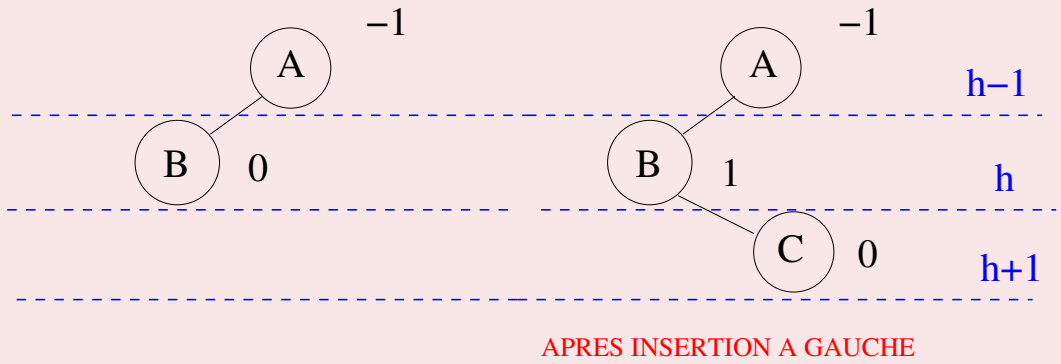


Re-équilibrage

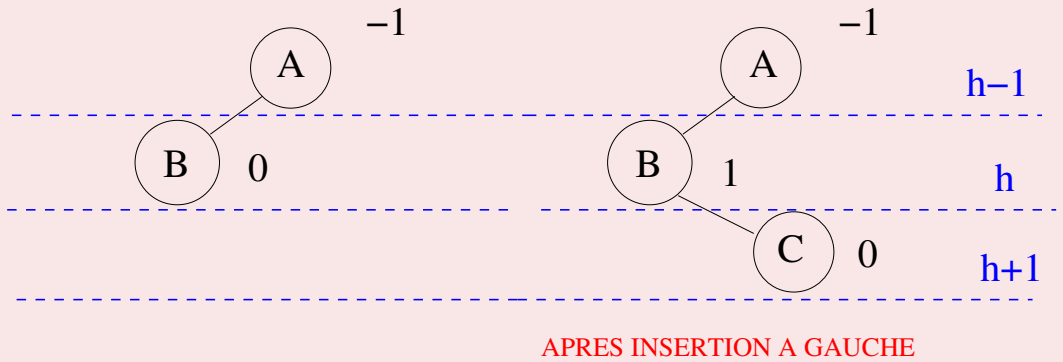


Algorithme (suite ...)

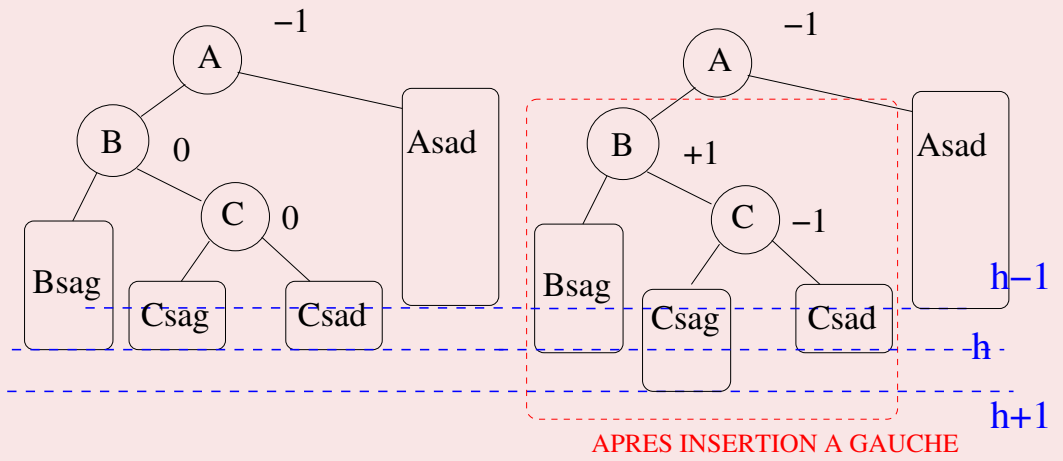
Exemple : le nœud B va pencher à droite (1)



Exemple : le nœud B va pencher à droite (1)

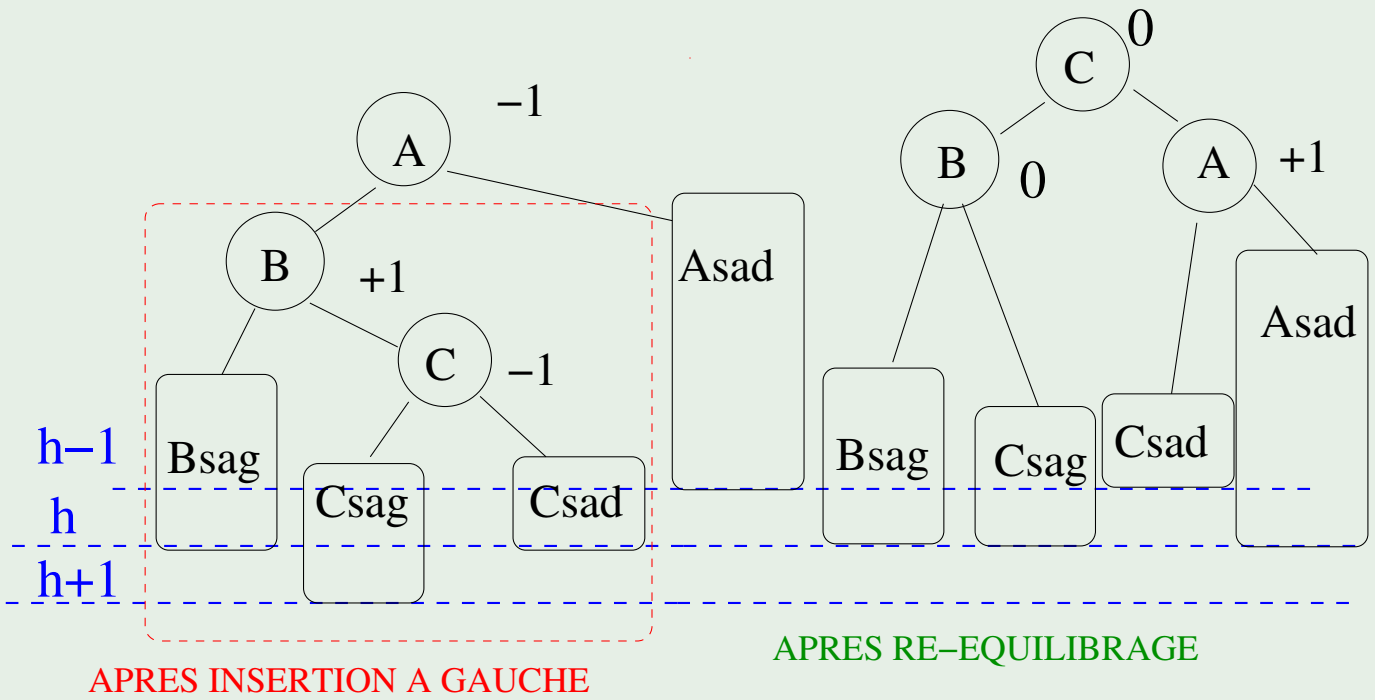


Exemple : le nœud B va pencher à droite (2)



Algorithme (suite ...)

Exemple : le nœud B va pencher à droite (2)

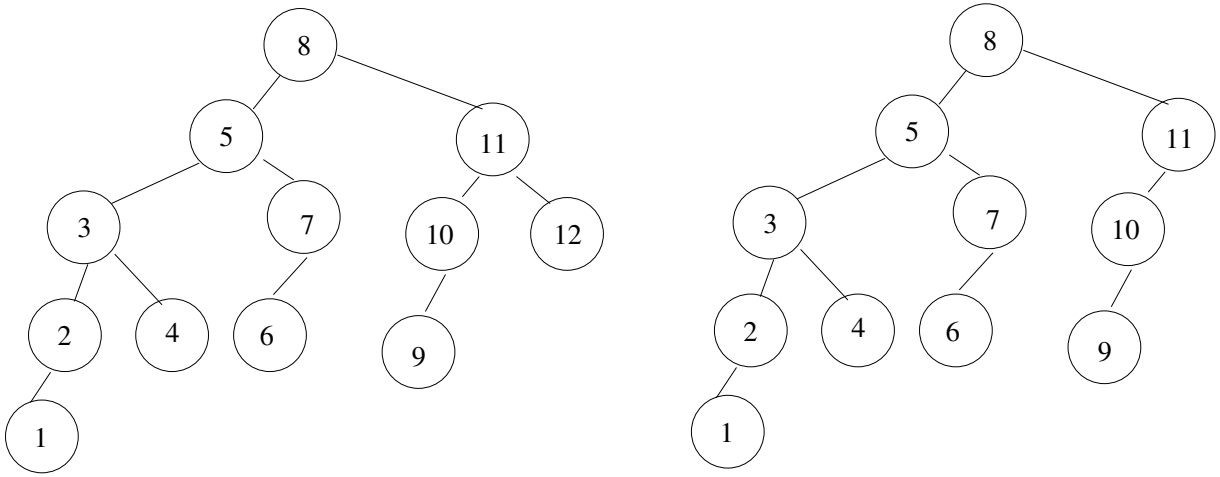


Code java rééquilibrage

cf. avl1.java

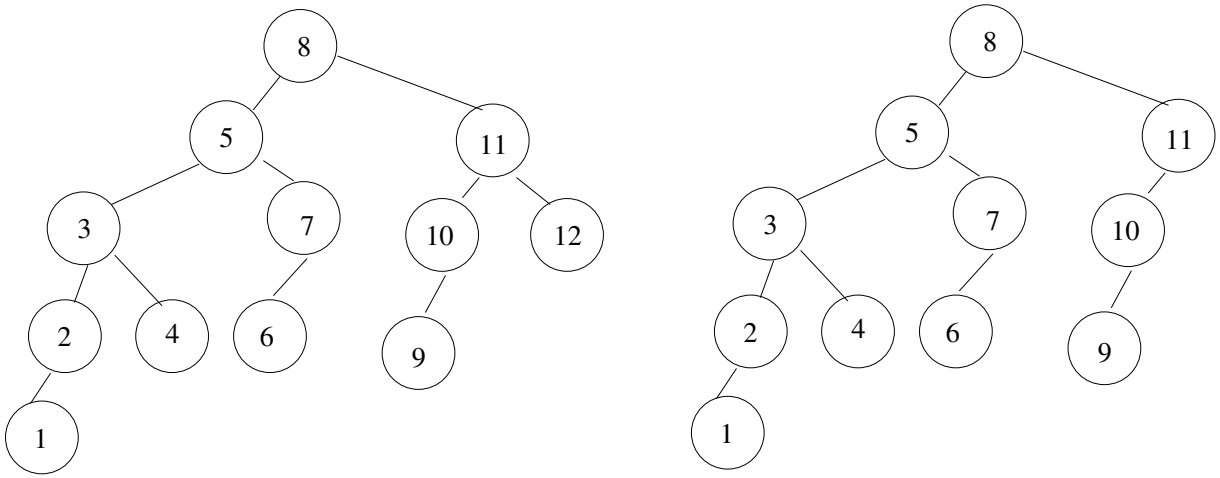
Fonction de suppression

De même une suppression peut entraîner plusieurs rotations et rééquilibrages:

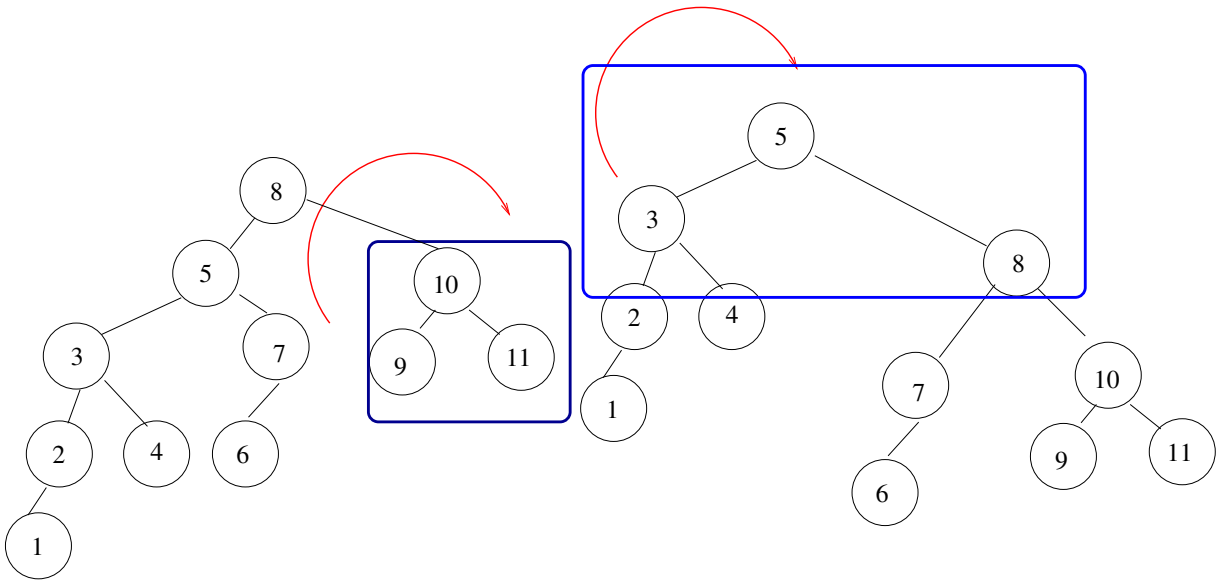


Fonction de suppression

De même une suppression peut entraîner plusieurs rotations et rééquilibrages:



La suppression de 12 exige `deux rotations` pour le rééquilibrage:



Fonctions de rééquilibrage

cf. avl2.java

Définition

Les arbres `rouge-noirs` sont des arbres binaires ordonnés pour lesquels on a:

- chaque nœud est soit noir, soit rouge.
- chaque feuille (nœud sentinelle) est noire.
- la racine est toujours noire.
- si un nœud est rouge, alors ses deux enfants seront noirs.
- pour tout nœud de l'arbre, les chemins de ce nœud vers les feuilles (nœud sentinelle) qui en dépendent ont le même nombre de nœuds noirs.

Propriété fondamentale

La hauteur noire d'un nœud n notée $hn(n)$ est le nombre de nœuds noirs dans un chemin du nœud à une de ses feuilles (le nœud n n'est pas compris). La hauteur noire d'un arbre rouge-noir est la hauteur noire de sa racine.
Un arbre rouge-noir contenant n nœuds internes a une hauteur inférieure ou égale à $2\log_2(n+1)$.

Propriété fondamentale

La hauteur noire d'un nœud n notée $hn(n)$ est le nombre de nœuds noirs dans un chemin du nœud à une de ses feuilles (le nœud n n'est pas compris). La hauteur noire d'un arbre rouge-noir est la hauteur noire de sa racine.
Un arbre rouge-noir contenant n nœuds internes a une hauteur inférieure ou égale à $2\log_2(n+1)$.

Montrons d'abord par récurrence sur la hauteur noire d'un nœud, que le nombre de nœuds internes du sous-arbre enraciné en x est au moins égal à $2hn(x) - 1$.

- Si la hauteur noire de x est 0, alors c'est le nœud sentinelle, et le sous arbre enraciné en n contient $2hn(x) - 1 = 0$ nœuds internes.

Propriété fondamentale

La hauteur noire d'un nœud n notée $hn(n)$ est le nombre de nœuds noirs dans un chemin du nœud à une de ses feuilles (le nœud n n'est pas compris). La hauteur noire d'un arbre rouge-noir est la hauteur noire de sa racine.
Un arbre rouge-noir contenant n nœuds internes a une hauteur inférieure ou égale à $2\log_2(n+1)$.

Montrons d'abord par récurrence sur la hauteur noire d'un nœud, que le nombre de nœuds internes du sous-arbre enraciné en x est au moins égal à $2hn(x) - 1$.

- Si la hauteur noire de x est 0, alors c'est le nœud sentinelle, et le sous arbre enraciné en n contient $2hn(x) - 1 = 0$ nœuds internes.
- Si la hauteur noire de x est > 0, alors chacun de ses fils a une hauteur noire égale soit à *hn*(x) s'il est rouge, soit à *hn*(x)-1 s'il est noir.
  Donc, en appliquant l'hypothèse de récurrence aux deux sous arbres de x, le sous arbre enraciné en x contient au moins: $2 \times (2hn(x) - 1 - 1) + 2 = 2hn(x)$ nœuds internes.

Borne sur la hauteur

Borne supérieure de la hauteur

Soit h la hauteur d'un arbre rouge-noir, la moitié au moins des nœuds vers une feuille doivent être noirs.

Borne supérieure de la hauteur

Soit h la hauteur d'un arbre rouge-noir, la moitié au moins des nœuds vers une feuille doivent être noirs. Donc la hauteur noire d'un arbre rouge-noir est au moins $h/2$. On a:

$$n \geq 2^{hn(\text{racine})} - 1, \log_2(n+1) \geq hn(\text{racine}) \geq \frac{h}{2} \text{ et } 2\log_2(n+1) \geq h.$$

Code java

cf. rouge-noir.java