Les arbres binaires de recherche

Arbres binaires de Recherche ABR

- Les ABR sont des structures de données qui peuvent supporter des opérations courantes sur des ensembles dynamiques.
  - ex: rechercher, minimum, maximum, prédécesseur….
- Cette structure de données est maintenue sous forme d'arbre binaire avec racine.
- Un ABR a pour structure logique un arbre binaire.

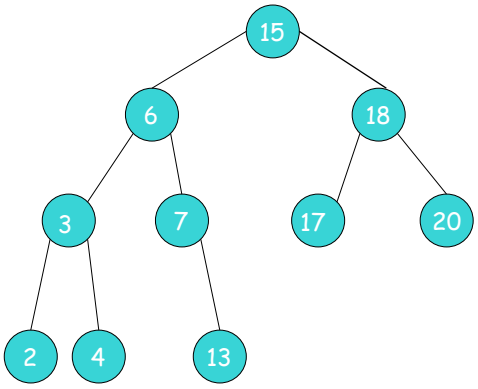
Clés

- Chaque nœud d'un ABR est associé à une clé
  - on supposera que chaque clé est numérique
  - Il faut que 2 clés soient comparables.

Propriétés des ABR

- soit x un nœud d'un ABR
- x a au plus 2 fils
  - fils gauche
  - fils doit
- si y est un nœud du sous arbre gauche de x alors:
  - clé(y)<clé(x)
- si y est un nœud du sous arbre droit de x alors:
  - clé(y)>clé(x)

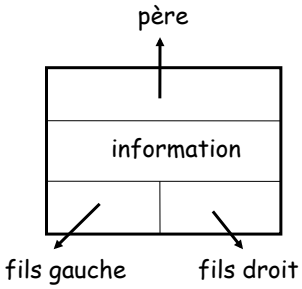
Exemple d'ABR



Description d'un noeud

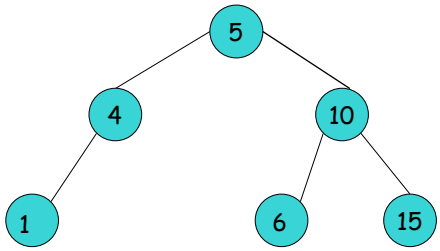
- Un ABR est donc composé de nœuds qui peuvent être créés de manière dynamique
- Si un nœud n'a pas de père, le pointeur est mis à NIL

```
Enregistrement Nœud {
     cle : Entier;
     gauche : ↑Nœud;
     droit : ↑Nœud;
     pere : ↑Nœud;
}
```



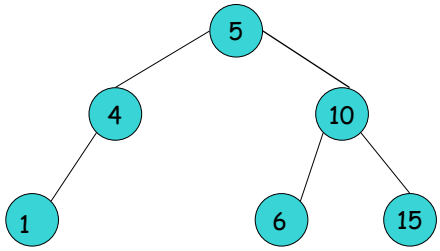
Affichage des nœuds d'un ABR dans l'ordre croissant des clés

- Pour afficher dans l'ordre :
  - Parcours en profondeur d'abord infixe.



Recherche d'une clé minimum ou d'une clé maximum

- On sait que la valeur maximum se trouve dans le sous arbre droit de la racine
- Il faut donc chercher le fils droit du fils droit du fils droit….jusqu'à une feuille.



Maximum d'un ABR: Pseudo code

```
ABR-Max(Nœud x): Entier
Début
   tant que x.droite ≠ NIL
        x ← x.droite;
   fin tant que
   retourner x
Fin
```

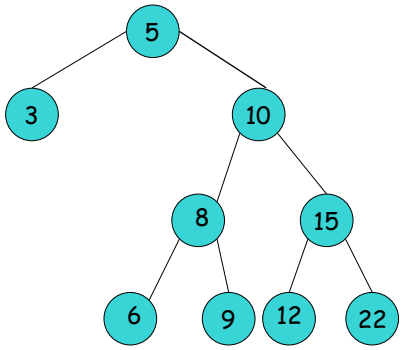
Recherche

- parcours dans l'arbre, depuis la racine jusqu'à l'élément recherché (ou une feuille si l'élément n'existe pas) en branchant à chaque nœud en fonction de la valeur de la clé

```
recherche(o: ↑Nœud, c: clé) : booléen
début
  si o ≠ NIL alors
    si la_clé(o) = c retourne VRAI;
    sinon si la_clé(o) > c retourne recherche(o.filsg, c)
    sinon retourne recherche(o.filsd, c)
    fsi
  sinon retourne FAUX;
fin
```

Successeur et prédécesseur

- Soit x un nœud
- on cherche y tel que
  - clé(y)>clé(x)
- et tel que pour tout nœud z
  - z≠x et z≠y
  - on n'ait pas clé(y)>clé(z)>clé(x)

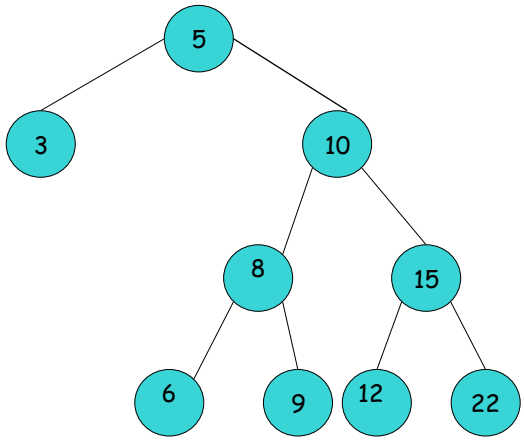


12 est le successeur de 10
22 est le successeur de 15
5 est le successeur de 3

Successeur et prédécesseur

- Le successeur d'un nœud x est le nœud possédant la plus petite clé dans le sous-arbre droit si x en possède un
- sinon c'est le nœud qui est le 1er ancêtre de x dont le fils gauche est aussi un ancêtre de x (ou x lui-même)

Successeur et prédécesseur Exemple



Successeur d'un ABR

```
ABR-Successeur(Nœud x): Noeud
Début
Y : Nœud;
   si x.droite ≠ NIL
      retourner ABR_MIN(x.droite);
   fin si
   y ← x.pere;
   tant que y ≠ NIL et x == y.droite
       x ← y;
       y ← y.pere;
      fin tant que
      retourner y;
Fin
```

Insertion et suppression d'un nœud dans l'ABR

- Les opérations d'insertion et de suppression d'un noeud modifient l'ensemble dynamique représenter par l'ABR
- La structure dynamique doit être modifiée tout en gardant une structure d'ABR

Insertion

- 2 techniques :
  - Insertion à la racine l'ABR
    - Modification de la structure de l'ABR
  - Insertion aux feuilles de l'ABR
    - Modification de la hauteur de l'ABR

Ajout d'une feuille

- recherche de la clé de l'élément que l'on essaye d'insérer
  - **si** élément existant **alors** rien à faire
  - **sinon** la recherche s'est arrêtée sur un arbre vide, qu'il suffira de remplacer par l'élément à insérer
- complexité moyenne de PCE(T)
- complexité au pire de h(T)

Insertion d'un nœud aux feuilles

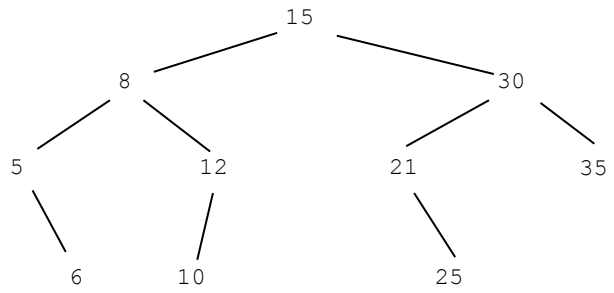
```
ABR-Inserer(T : Noeud, z : Noeud)
Entrées : T la racine de l'ABR, Z le nouveau nœud à inserér
Sorties : T l'arbre dans lequel on a inséré z.
Début
   x,y : Nœud;

   y ← NIL;
   x ← T;
   tant que x ≠ NIL
   y ← x;
   si z.clé < x.clé
        x ← x.gauche;
   sinon
        x ← x.droit;
   fin si
   fin tant que
        z.pere ← y
        si y == NIL
           T ← z
        sinon
           si z.clé < y.clé
              y.gauche ← z;
           sinon
              y.droit ← z;
           fin si
        fin si
     Fin
```

Insertion d'un nœud aux feuilles

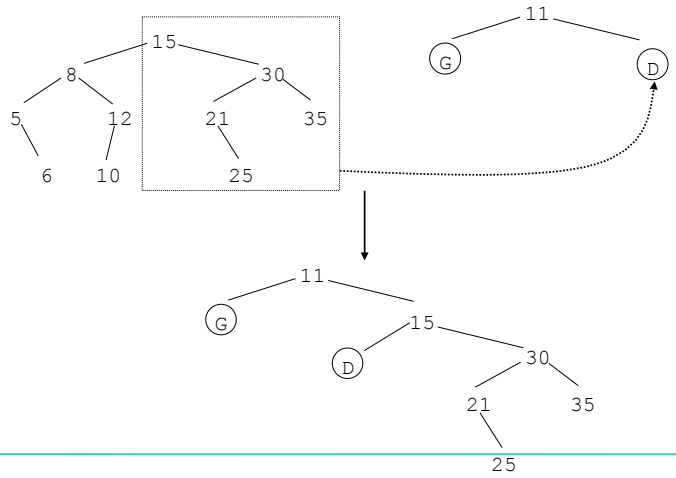
```
ABR-Inserer(T : Noeud, z : Noeud) : Noeud
Entrées : T la racine de l'ABR, Z le nouveau nœud à inserér
Sorties : T l'arbre dans lequel on a inséré z.
Début
   si (T == NIL)
      retourner z;
   sinon
   si (z.clé ≤ T.clé)
         T.gauche ← ABR-Inserer(T.gauche,z);
         retourner T;
      sinon
         T.droit ← ABR-Inserer(T.droit,z);
         retourner T;
      fin si
   fin si
Fin
```

Ajout à la racine : exemple (1)

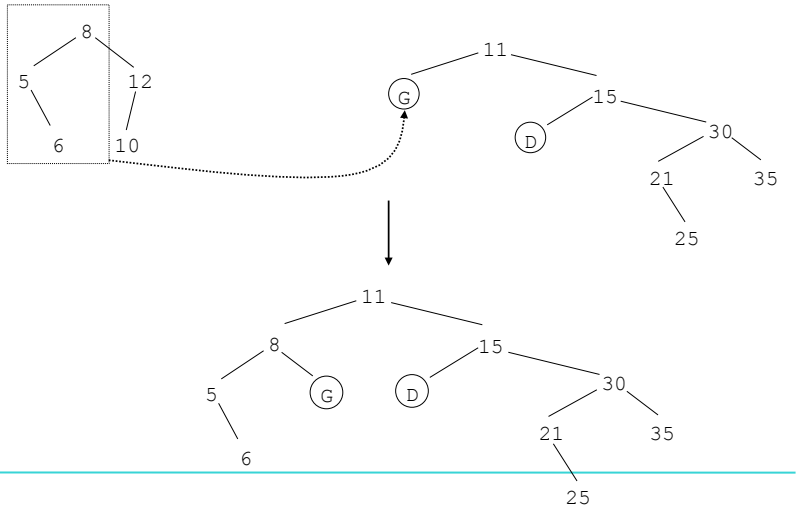
- Ajout de 11 à l'arbre suivant :



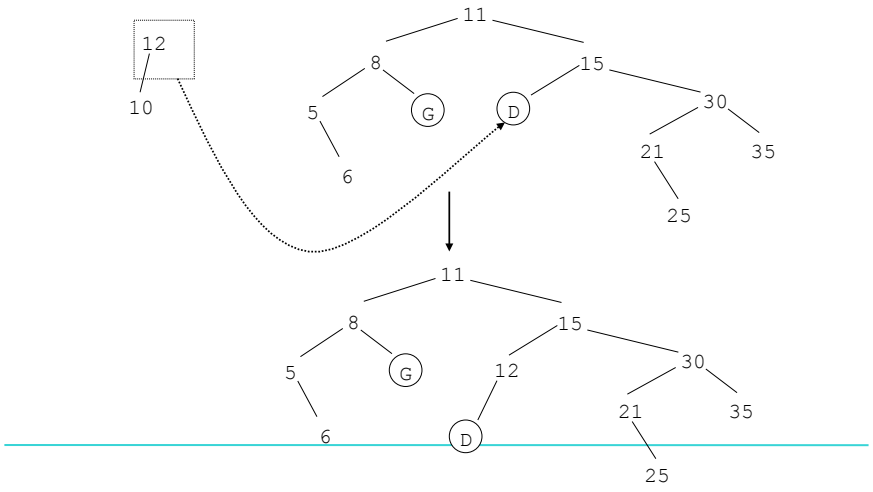
Ajout à la racine : exemple (2)



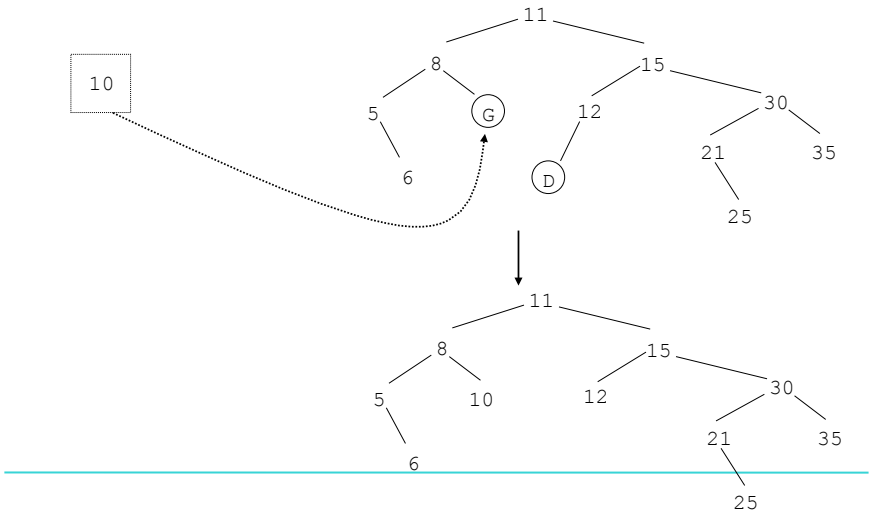
Ajout à la racine : exemple (3)



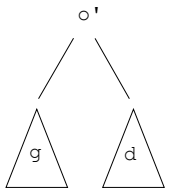
Ajout à la racine : exemple (4)



Ajout à la racine : exemple (5)



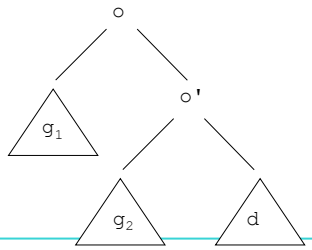
Généralisation (1)



- Soit un arbre `a = <o', g, d>`
- Ajouter le nœud `o` à `a`, c'est construire l'arbre `<o, a1, a2>` tel que :
  - `a1` contienne tous les nœuds dont la clé est inférieure à celle de `o`
  - `a2` contienne tous les nœuds dont la clé est supérieure à celle de `o`

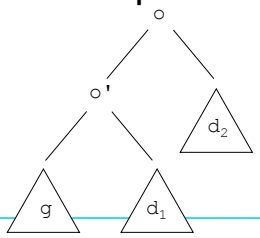
Généralisation (2)

- si la_clé(`o`) < la_clé(`o'`)
  - `a1 = g1` et `a2 = <o', g2, d>`
- `g1` = nœuds de `g` dont la clé est inférieure à la clé de `o`
- `g2` = nœuds de `g` dont la clé est supérieure à la clé de `o`



Généralisation (3)

- si la_clé(o) > la_clé(o')
  - a1 = <o', g, d1> et a2 = d2
- d1 = nœuds de d dont la clé est inférieure à la clé de o
- d2 = nœuds de d dont la clé est supérieure à la clé de o

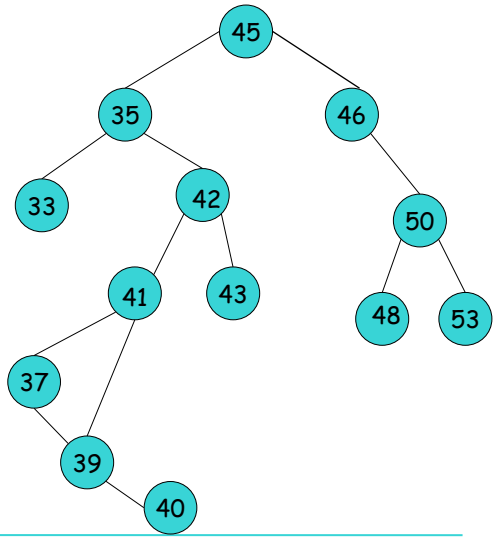


Suppression

- Recherche du nœud
  - si feuille, suppression simple
  - si nœud interne au sens large, suppression du nœud et raccordement du sous-arbre
  - si nœud interne, suppression du nœud et remplacement soit par
    - le nœud du sous-arbre gauche dont la clé est la plus grande
    - le nœud du sous-arbre droit dont la clé est la plus petite
- Complexité
  - Complexité moyenne de PC(a)
  - complexité au pire de h(a)

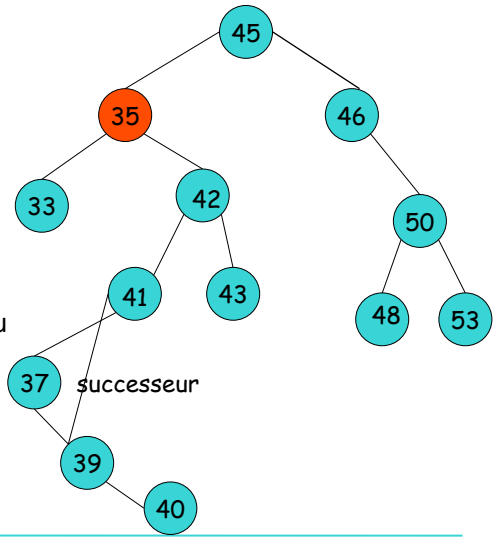
Suppression d'un nœud

1. Suppression d'une feuille
2. Suppression d'un nœud avec 1 enfant
3. Suppression d'un nœud avec 2 enfants



Suppression d'un nœud

- Suppression d'un nœud avec 2 enfants
  - écraser le nœud avec son successeur
  - le fils droit du successeur
    - devient le fils gauche du père du successeur



Suppression d'un nœud

```
Supprimer_ABR(Nœud T, Nœud z)
Entrées : Un ABR T et un nœud Z
Sortie : Un ABR T dans lequel le nœud z a été supprimé
Début
      si T ≠ NIL
           si (z.clé < T.clé)
                retourner (Supprimer_ABR(T.gauche, z));
          sinon si (z.clé > T.clé)
                    retourner(Supprimer_ABR(T.droit,z));
               sinon si (T.gauche == NIL)
                         retourner (T.droit);
                    sinon si (T.droit == NIL)
                              retourner (T.gauche)
                         sinon
                  T.clé ←successeur(T.gauche,z);
             retourner (Supprimer_ABR(T.gauche,T.clé));
                   fin si
                    fin si
               fin si
      fin si
      retourner T;
fin
```

Notes sur la complexité

- Les différentes opérations ont une complexité au pire de h(a)
  - $\lfloor \log_2 n \rfloor \leq h(a) \leq n-1$
  - Pour les arbres complets, complexité en $O(\log_2 n)$
  - Pour les arbres dégénérés, complexité en O(n)
- La complexité dépend de la forme de l'arbre, qui dépend des opérations d'ajout et de suppression
  - ajout d'éléments par clés croissantes → arbre dégénéré
  - en moyenne, la profondeur est de $2 \log_2 n$
  - but = équilibrer les arbres en hauteur

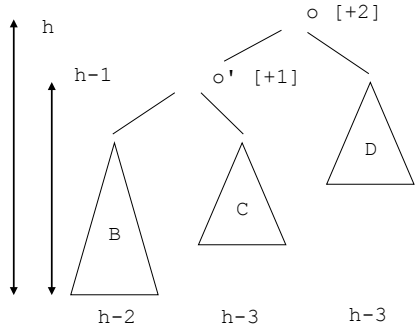
Arbres H-équilibrés / arbres AVL

- Définitions
  - déséquilibre(a) = h(g(a)) – h(d(a))
  - un arbre a est **H-équilibré** si pour tous ses sous-arbres b, on a :
    - déséquilibre(b) ∈ {-1, 0, 1}
  - un **arbre AVL** est un arbre de recherche qui est H-équilibré
    - les propriétés et les opérations définies sur les arbres de recherche peuvent s'appliquer aux arbres AVL

Opérations de rotation

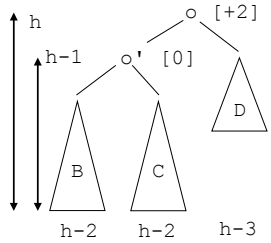
- Le problème est d'essayer de rééquilibrer un arbre déséquilibré afin de le ramener à un arbre H-équilibré.
- Cas d'un déséquilibre +2
  - on suppose que les sous-arbres droit et gauche sont H-équilibrés
  - hauteur du sous-arbre gauche supérieure de 2 à la hauteur du sous-arbre droit
  - opération à pratiquer dépend du déséquilibre du sous-arbre gauche qui peut être +1, 0, -1

Déséquilibre +1 sur le fils gauche



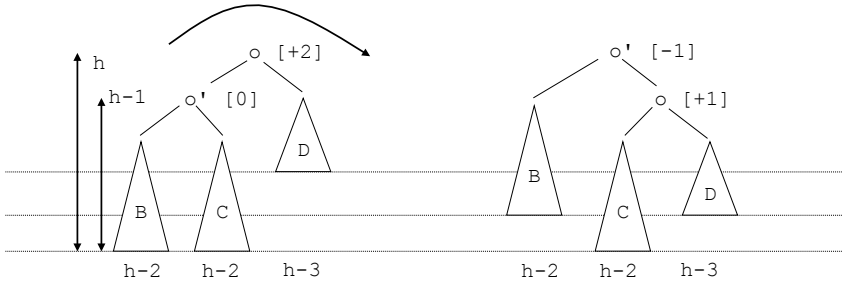
- rotation à droite :
  - o' devient la racine de l'arbre
  - Le fils droit de o' devient le fils gauche de o
  - o devient le fils droit de o'

Déséquilibre 0 sur le fils gauche



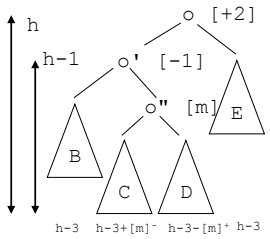
- rotation à droite :
  - o' devient la racine de l'arbre
  - Le fils droit de o' devient le fils gauche de o
  - o devient le fils droit de o'

Déséquilibre 0 sur le fils gauche



- Si l'arbre A est un arbre binaire de recherche, le résultat est un arbre binaire de recherche
  - le déséquilibre de l'arbre résultant est de -1
  - arbre H-équilibré dont la hauteur est identique

Déséquilibre -1 sur le fils gauche

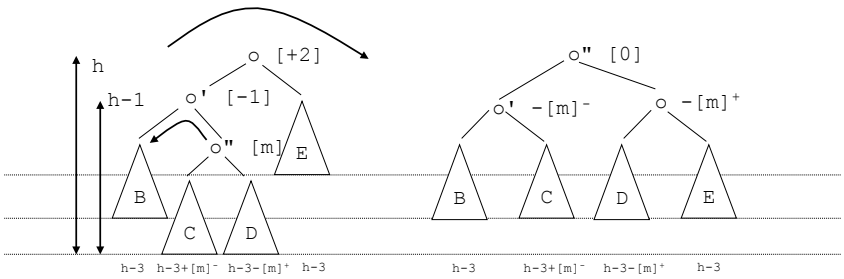


Avec m le déséquilibre de o"

- $[m]^+ = \max(0, m)$
- $[m]^- = \min(0, m)$

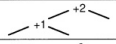
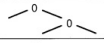
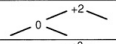
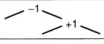
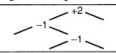
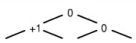
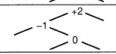
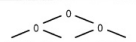
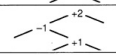
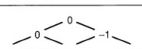
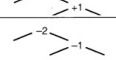
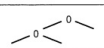
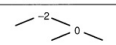
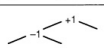
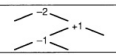
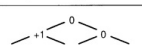
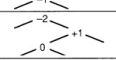
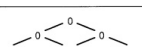
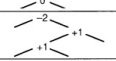
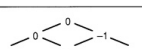
- rotation gauche-droite
  - Rotation à gauche sur le fils gauche
  - Rotation à droite sur la racine

Déséquilibre -1 sur le fils gauche



- Si l'arbre A est un arbre binaire de recherche, le résultat est un arbre binaire de recherche
  - le déséquilibre de l'arbre résultant est de 0
  - arbre H-équilibré dont la hauteur est diminuée d'1

Opérations de rééquilibrage

arbre origine	opération	résultat	hauteur
	rotation droite		diminution
	rotation droite		identique
	rotation gauche droite		diminution
	rotation gauche droite		diminution
	rotation gauche droite		diminution
	rotation gauche		diminution
	rotation gauche		identique
	rotation droite gauche		diminution
	rotation droite gauche		diminution
	rotation droite gauche		diminution

Opération d'ajout

- Principe :
  - ajout du nœud par l'opération ajouter-f
  - rééquilibrage de l'arbre en partant de la feuille et en remontant vers la racine
- Complexité :
  - complexité au pire en $O(\log_2 n)$ en nb de comparaisons
  - au plus une rotation
  - expérimentalement : en moyenne une rotation pour 2 ajouts

Opération de suppression

- Principe :
  - suppression du nœud par l'opération sur les arbres de recherche
  - rééquilibrage de l'arbre en partant du nœud supprimé et en remontant vers la racine
- Complexité :
  - complexité au pire en $O(\log_2 n)$ en nb de comparaisons et en nb de rotations
  - il peut y avoir plus d'une rotation
  - expérimentalement : en moyenne une rotation pour 5 suppressions