

Arbres équilibrés

Pour les ABR, les opérations de recherche, d'ajout et de suppression se font en moyenne en $\Theta(\log n)$, mais dans le pire cas la complexité de ces opérations est en $\Theta(n)$ car un ABR peut dégénérer en un peigne ce qui est équivalent à une liste.

Intuitivement, on comprend que pour un arbre "équilibré" (càd dont les feuilles sont sur au plus deux niveaux) il faut $\log n$ comparaisons pour une recherche dans le pire cas.

La question est de savoir si l'on peut maintenir une structure d'arbre équilibré lors des ajout ou des suppressions, et que cette contrainte ait un coût en $O(\log n)$.

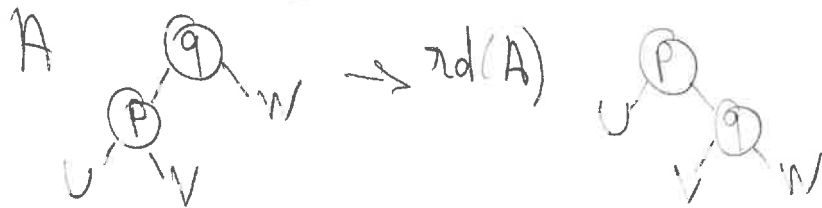
Il y a deux façons d'assurer cela, soit d'avoir des arbres binaires "légèrement déséquilibrés" soit d'avoir des arbres dont les noeuds ont plus de deux fils.

On va étudier une classe d'arbres binaires "légèrement déséquilibrés".

5.1 Rotations

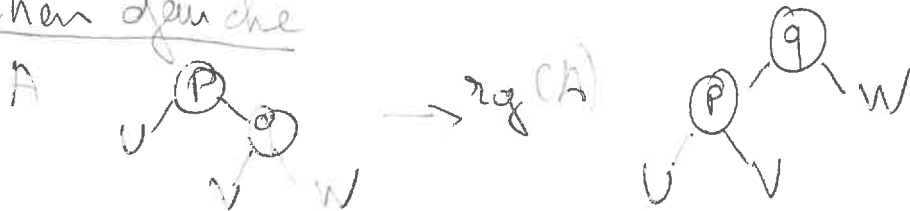
La plupart des algorithmes de rééquilibrage d'arbres utilisent des transformations locales élémentaires appelées des rotations. Il y a quatre types de rotations.

Rotation droite



où U, V et W sont des sous-arbres.

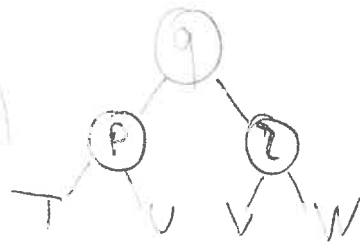
Rotation gauche



Rotation gauche-droite



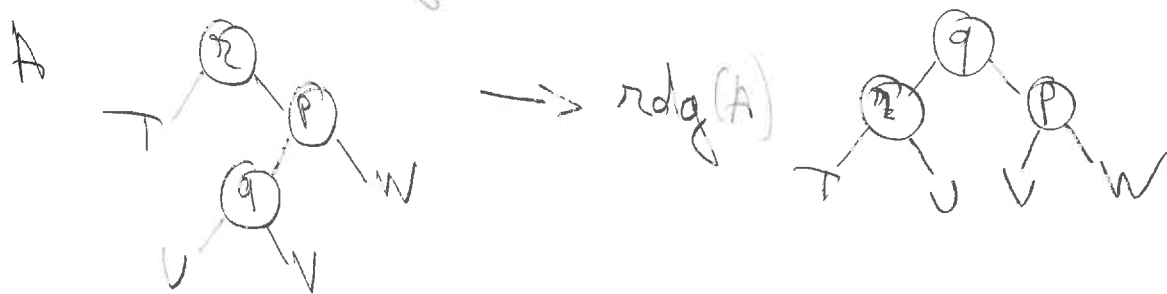
$\rightarrow \text{rot}(A)$



ou T, U, V et W sont des
sous-arbres

On peut voir qu'une rotation gauche-droite
est composée d'une rotation gauche sur le sous-arbre
gauche de A suivie d'une rotation droite sur A

Rotation droite-gauche



On peut voir qu'une rotation droite-gauche est composée d'une rotation droite sur le sous-arbre droit de A suivie d'une rotation gauche sur A.

Remarque: Si tous les éléments sont distincts, les rotations préservent la propriété d'arbre binaire de recherche

On spécifie ces opérations de la façon suivante :

$rg : t_btree \rightarrow t_btree$

$rd : t_btree \rightarrow t_btree$

$rgd : t_btree \rightarrow t_btree$

avec $A, T, U, V, W : t_btree ; p, q, r : \text{Element}$

préconditions

$rg(A)$ défini-ssi $\text{not}(\text{isEmpty}(A))$ et $\text{not}(\text{isEmpty}(\text{rson}(A)))$

$rd(A)$ défini-ssi $\text{not}(\text{isEmpty}(A))$ et $\text{not}(\text{isEmpty}(\text{lson}(A)))$

$rgd(A)$ défini-ssi $\text{not}(\text{isEmpty}(A))$ et $\text{not}(\text{isEmpty}(\text{lson}(A)))$
et $\text{not}(\text{isEmpty}(\text{rson}(\text{lson}(A))))$

$rdg(A)$ defini se $not(isEmpty(A))$ et $not(isEmpty(rsen(A)))$
 et $not(isEmpty(lsen(rsen(A))))$

axiomes

$$rg(\text{rooting}(p, U, \text{rooting}(q, V, W))) = \text{rooting}(q, \text{rooting}(p, U, V), W)$$

$$rd(\text{rooting}(q, \text{rooting}(p, U, V), W)) = \text{rooting}(p, U, \text{rooting}(q, V, W))$$

$$rg(\text{rooting}(r, \text{rooting}(p, T, \text{rooting}(q, U, V)), W)) = \\ \text{rooting}(q, \text{rooting}(p, T, U), \text{rooting}(r, V, W))$$

$$rdg(\text{rooting}(r, T, \text{rooting}(p, \text{rooting}(q, U, V), W))) = \\ \text{rooting}(q, \text{rooting}(r, T, U), \text{rooting}(p, V, W))$$

5.2 Arbres H-équilibrés

Définition: On dit qu'un arbre binaire est H-équilibré si en tout nœud de l'arbre, les hauteurs des sous-arbres gauche et droit diffèrent d'au plus 1.

On définit la fonction "déséquilibre" de la façon suivante:

déséquilibre: $t_btree \rightarrow \mathbb{Z}$

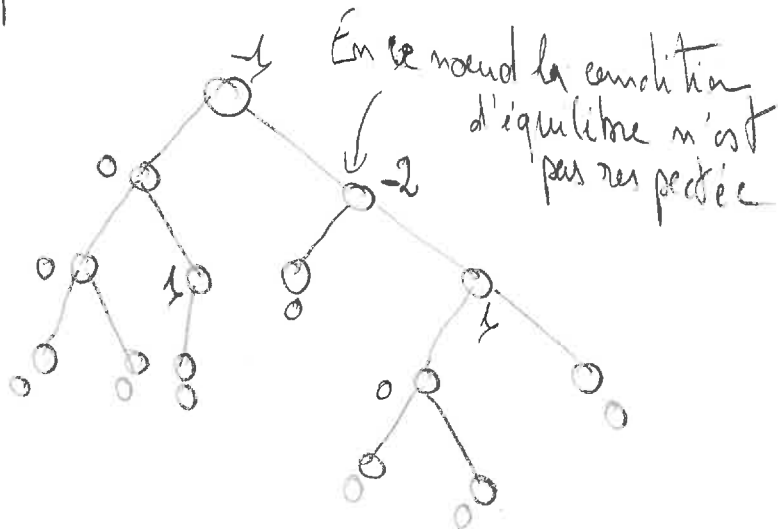
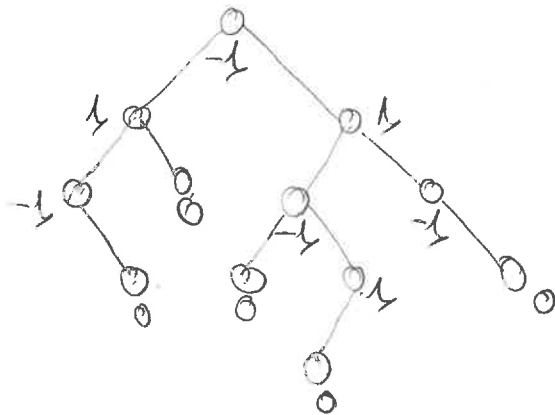
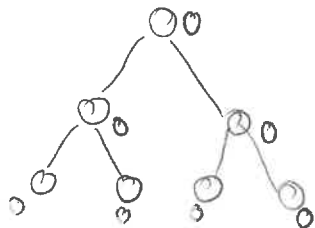
avec $v: \text{Element}$; $g, d: t_btree$

déséquilibre(empty) = 0

déséquilibre(creating(v, g, d))) = hauteur(g) - hauteur(d)

Un arbre A est H-équilibré si pour tout sous-arbre S de A
on a déséquilibre(S) $\in \{-1, 0, 1\}$

Exemple : En chaque nœud, on indique la valeur de la fonction d'équilibre



Propriété: la hauteur de tout arbre binaire
H-équilibré vérifie:

$$\log_2(n+1) \leq h+1 \leq 1,44 \log_2(n+2)$$

Ainsi la hauteur d'un arbre binaire H-équilibré
est en $\Theta(\log n)$ où n est le nombre de nœuds.

On voit donc que la contrainte sur l'équilibre d'un arbre
binaire permet de limiter sa hauteur.

5.3 Arbres AVL

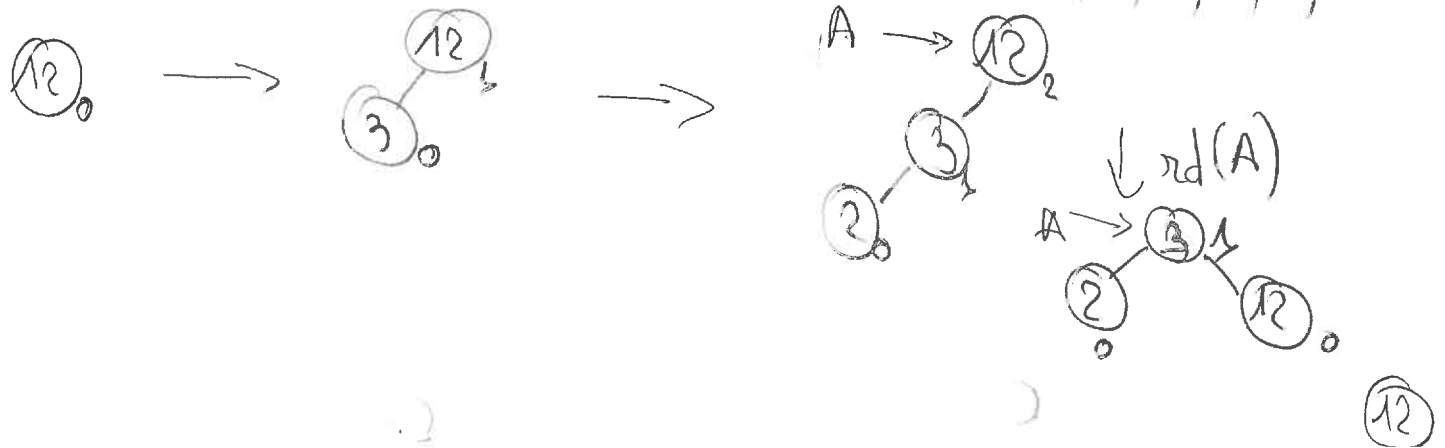
Les arbres AVL sont des arbres H-équilibrés inventés dans les années 1960 par Adelson-Velskii et Landis (d'où AVL). Pour les arbres AVL la recherche, l'ajout et la suppression d'un élément sont en $\Theta(\log n)$ où n est le nombre de nœuds.

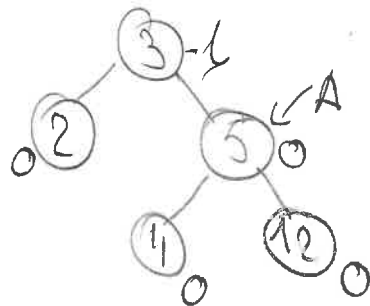
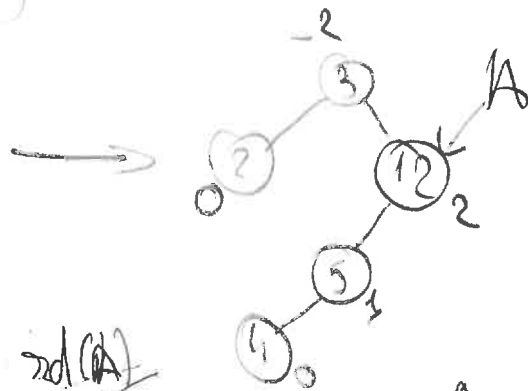
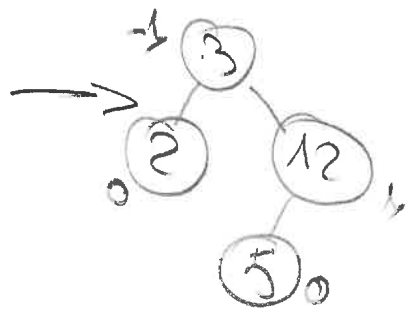
Pour l'ajout et la suppression, il est généralement nécessaire d'effectuer des rotations pour rétablir l'équilibre de l'arbre.

5.3.1 Exemple d'ajout dans un AVL

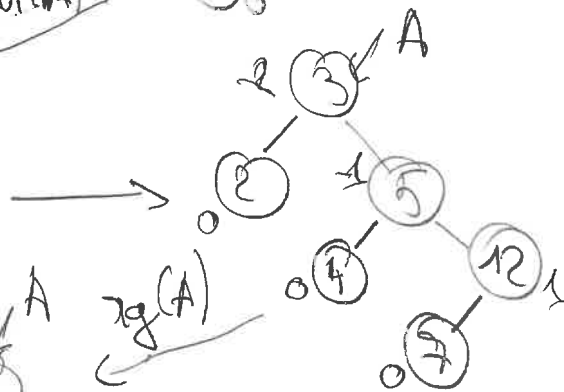
L'ajout d'un élément dans un AVL demande de faire des rotations à certains endroits de l'arbre pour le garder équilibré.

Exemple: On construit l'AVL associé à l'ajout des feuilles de la suite d'éléments 12, 3, 2, 5, 4, 7, 9, 11, 14, 10

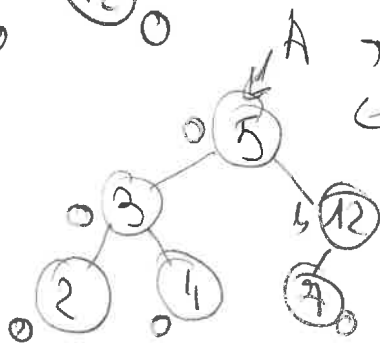


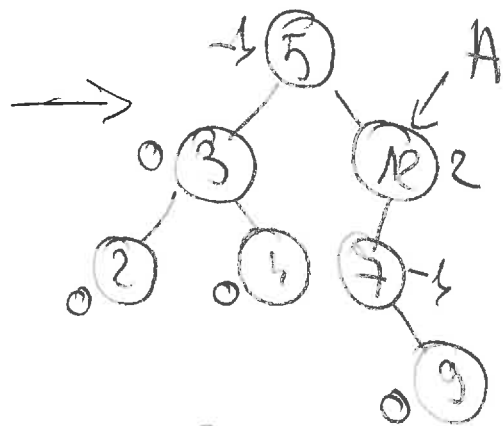


~~20(A)~~

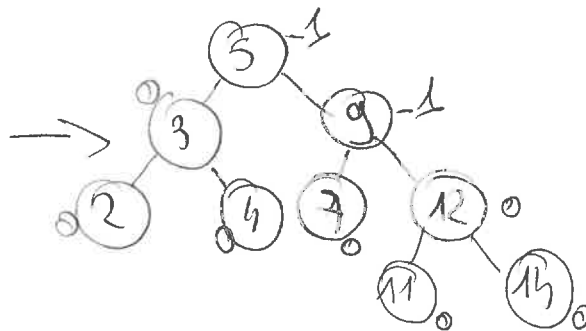
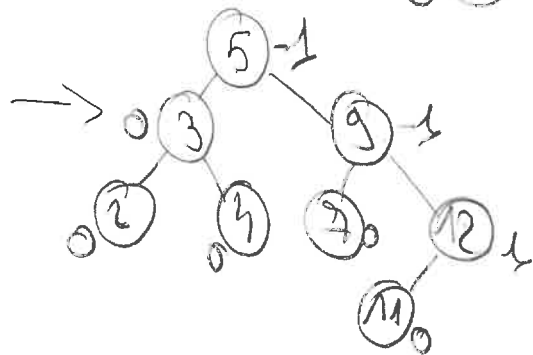
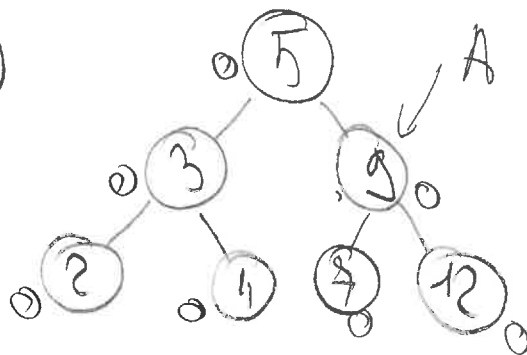


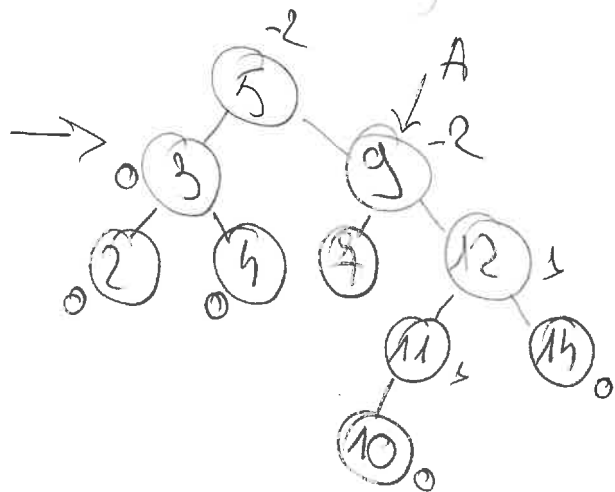
~~28(A)~~



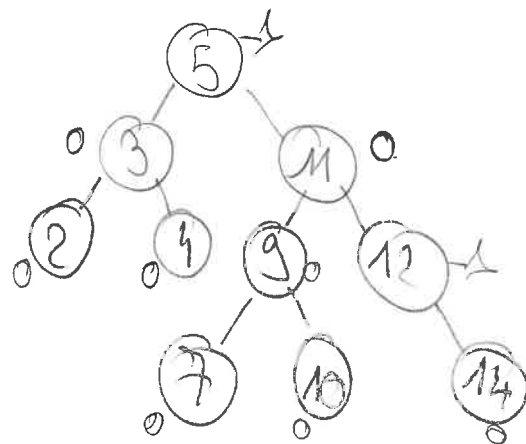


$rot(A)$





$\swarrow \text{rdg}(A)$



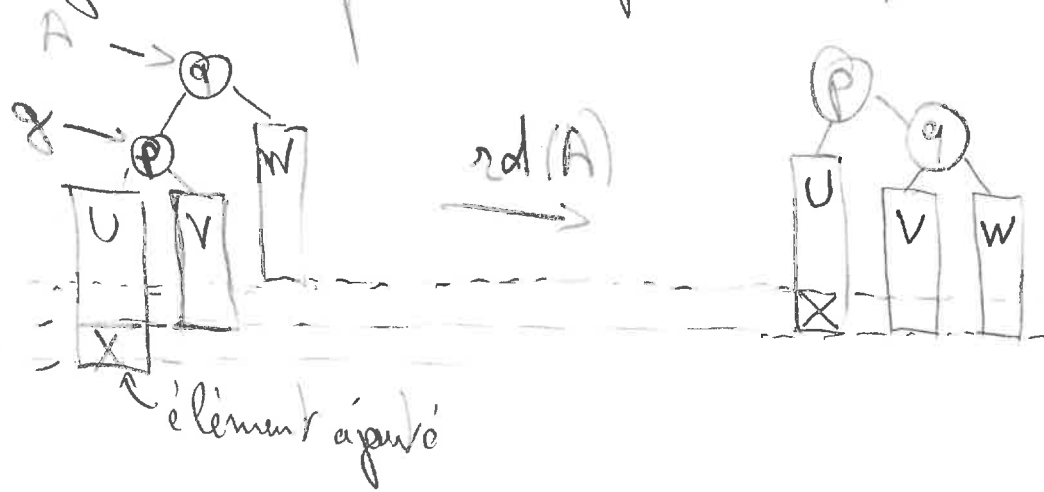
5.3.2 Rééquilibrage d'un AVL

Différents cas possibles

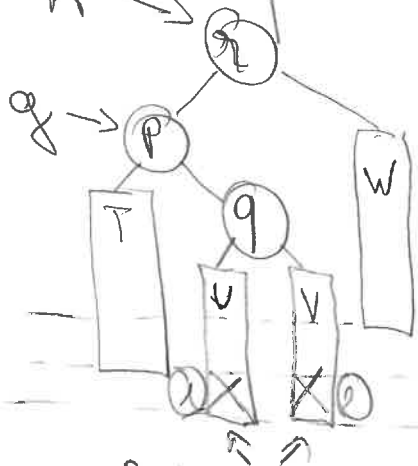
Considérons $A = (r, g, d)$ un AVL et supposons que l'ajout se fait sous le sous-arbre g , que l'ajout augmente la hauteur de g et que g reste un AVL (ad le déséquilibre de g vaut 0)

- 1) si $\text{déséquilibre}(A) = 0$ avant l'ajout alors $\text{déséquilibre}(A) = 1$ après l'ajout, A reste un AVL
- 2) si $\text{déséquilibre}(A) = -1$ avant l'ajout alors $\text{déséquilibre}(A) = 0$ après l'ajout, A reste un AVL

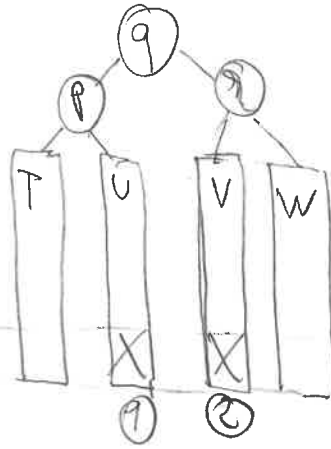
3) si déséquilibre $(A) = 1$ avant l'ajout et déséquilibre $(A) = 2$ après l'ajout, A n'est plus un AVL. Il faut rééquilibrer A. Il y a 2 cas possibles. Si l'ajout se fait dans le fils gauche de g, on rééquilibre A par une rotation droite



Si l'ajout se fait dans le fils droit de q alors $\text{déséquilibre}(q) = 1$
 On rééquilibre A par une rotation gauche-droite



$\text{rot}(A) \rightarrow$



Soit l'élément est
 ajouté dans fils gauche
 de q (cas 1), soit l'élément
 est ajouté dans le fils droit de q
 (cas 2)

Remarque que dans les deux
 cas la hauteur n'est pas modifiée
 après l'ajout et la rotation.

Dans les 2 cas l'arbre obtenu après rééquilibrage est un AVL et il retrouve la hauteur qu'il avait avant l'ajout.

Les autres cas quand déséquilibre $(A) = -2$ après l'ajout et que déséquilibre $(d) = 1$ ou -1 s'analysent de façon symétriquement semblable.

De plus, on a la propriété suivante :

Propriété:

Tout ajout dans un AVL
demande au plus une
rotation pour le rééquilibrer.

Id de preuve : On cherche le
noeud le plus bas où le déséquilibre
est non nul sur le chemin de
l'ajout de l'élément.

On appelle A l'arbre équilibré à ce moment le plus bas. On a vu que l'ajout ne modifie pas la hauteur de l'arbre A . Ainsi, les déséquilibres des sous-arbres enracinés au dessus de A ne sont pas modifiés et l'on a fait au plus une rotation pour rééquilibrer A .

On peut maintenant lancer la spécification de l'opération rééquilibrer

rééquilibrer : $t_avl_tree \rightarrow t_avl_tree$

déséquilibre : $t_avl_tree \rightarrow \mathbb{N}^+$

avec

$A : t_avl_tree$

précondition $\text{rééquilibrer}(A)$ si $\text{déséquilibre}(A) \in \{2, 3, 0, 1, 2\}$

axiomes

$\text{rééquilibrer}(A) = A$ si $\text{déséquilibre}(A) = 0$ ou 1 ou -1

$\text{rééquilibrer}(A) = \text{rd}(A)$ si $\text{déséquilibre}(A) = 2$ et

$\text{déséquilibre}(\text{lson}(A)) = 1$

$\text{rééquilibrer}(A) = \text{rgd}(A)$ si $\text{déséquilibre}(A) = 2$ et

$\text{déséquilibre}(\text{lson}(A)) = -1$

$\text{rééquilibrer}(A) = \text{ag}(A)$ si $\text{déséquilibre}(A) = -2$ et

$\text{déséquilibre}(\text{rson}(A)) = -1$

$\text{rééquilibrer}(A) = \text{rdg}(A)$ si $\text{déséquilibre}(A) = -2$ et

$\text{déséquilibre}(\text{rson}(A)) = 1$

5.3.3 Ajout dans un AVL

On spécifie l'ajout dans un AVL comme l'ajout dans un ABR en rééquilibrant l'arbre après chaque ajout ajkt_avl : $\text{Element} * t_avltree \rightarrow t_avltree$

avec $g, d : t_avltree ; v, e : \text{Element}$

associames

(je m'en sers ici car les
"routines")

$$\text{ajkt_avl}(e, \text{empty}) = (e, \text{empty}, \text{empty})$$

$$\text{ajkt_avl}(e, (v, g, d)) = \text{rééquilibrer}((v, \text{ajkt_avl}(e, g), d))$$

$$\text{ajkt_avl}(e, (v, g, d)) \stackrel{\text{si } e \leq v}{=} \text{rééquilibrer}((v, g, \text{ajkt_avl}(e, d)))$$

$$\text{ajkt_avl}(e, (v, g, d)) \stackrel{\text{si } e > v}{=} (v, g, d) \text{ si } e = v$$

5.3.4 Suppression dans un AVL

L'algorithme de suppression dans un AVL est semblable à celui spécifié pour un ABR, mais en tenant compte des rééquilibrages.

On utilise de même les opérations maxse qui renvoie l'élément maximal d'un AVL et minse qui renvoie un AVL privé de son élément maximal auquel cas il faut le rééquilibrer.

Voici la spécification de ces deux fonctions

maxe: t-avlTree \rightarrow Element

dmase: t-avlTree \rightarrow t-avlTree

avec t, g, d : t-avlTree ; v : Element

préconditions

maxe(t) défini si $\text{not}(\text{isEmpty}(t))$

dmase(t) défini si $\text{not}(\text{isEmpty}(t))$

associations

maxe(rooting(v, g, empty)) = v

maxe(rooting(v, g, d)) = maxe(d) si $\text{not}(\text{isEmpty}(d))$

dmase(rooting(v, g, empty)) = g

dmase(rooting(v, g, d)) = rééquilibrer(rooting($v, g, \text{dmase}(d)$))
si $\text{not}(\text{isEmpty}(d))$ (24)

On peut maintenant écrire la spécification de l'opération de suppression

$\text{suppr_avl} : \text{Element} * \text{t_avl_tree} \rightarrow \text{t_avl_tree}$

avec $e, v : \text{Element}; g, d : \text{t_avl_tree}$

(je n'ai pas écrit les "rooting")

$\text{suppr_avl}(e, \text{empty}) = \text{empty}$

$\text{suppr_avl}(e, (v, g, d)) = \text{reequilibrer}((v, \text{suppr_avl}(e, g), d))$
si $e \leq v$

$\text{suppr_avl}(e, (v, g, d)) = \text{reequilibrer}((v, g, \text{suppr_avl}(e, d)))$
si $e > v$

$\text{suppr_avl}(e, (v, g, \text{empty})) = g$ si $e = v$

$\text{suppr_avl}(e, (v, \text{empty}, d)) = d$ si $e = v$ et $\text{not}(\text{isEmpty}(d))$

$\text{suppr_avl}(e, (v, g, d)) = \text{rééquilibrer}(\text{max}(g), \text{dmax}(g), d)$

si $e = v$ et $\text{not}(\text{isEmpty}(g))$ et

$\text{not}(\text{isEmpty}(d))$

5.3.5 Complexité des algorithmes d'ajout et de suppression

On considère uniquement la complexité dans le cas le pire.

Pour l'ajout, comme un AVL est un arbre équilibré sa hauteur est en $\Theta(\log n)$ où n est la taille de l'arbre. Comme, il y a au plus une rotation et que les rotations sont en $\Theta(1)$ (temps constant), la complexité de l'ajout est en $\Theta(\log n)$ dans le pire des cas.

Pour la suppression, à cause de l'opération d'insertion, il peut y avoir des rotations en cascade jusqu'à la racine de l'arbre (c'est-à-dire que le nombre de rotations est au plus $1.44 \log_2 n$). Comme les rotations se font en $\Theta(1)$, la complexité de la suppression est en $\Theta(\log n)$ dans le pire des cas.

