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 ajouts ou des suppressions, et que cette contrainte ait un coût en $O(\log n)$.

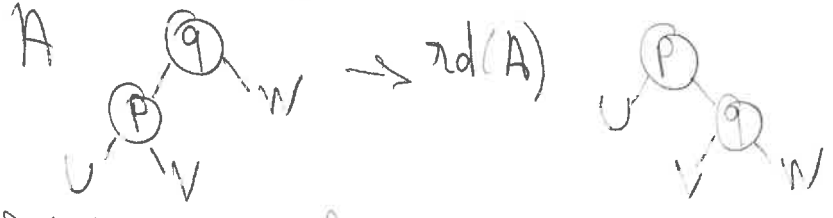
Il y a deux façons d'assurer cela, soit d'avoir des arbres binaires "légèrement déséquilibrés" soit d'avoir des arbres dont les noeuds ont plus de deux fils.

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

5.1 Rotations

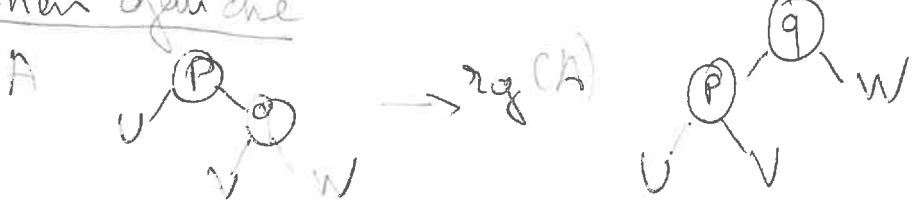
La plupart des algorithmes de rééquilibrage d'arbres utilisent des transformations locales élémentaires appelées des rotations. Il y a quatre types de rotations

Rotation droite

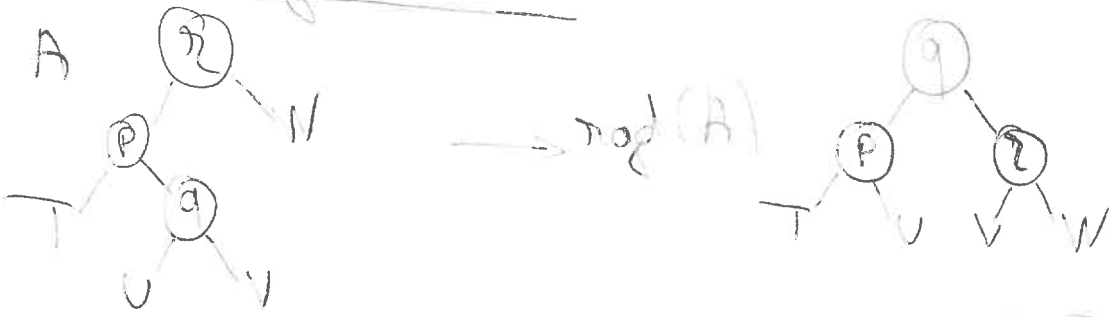


où U, V et W sont des sous-arbres.

Rotation gauche



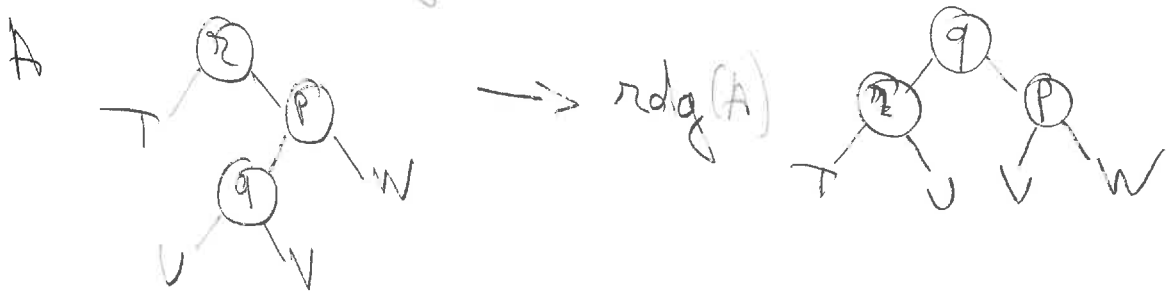
Rotation gauche-droite



où 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 → t_btree
rd : t_btree → t_btree
rgd : t_btree → t_btree

avec A, T, U, V, W : t_btree ; p, q, r : Element

préconditions

rg(A) défini_ssi not(isEmpty(A)) et not(isEmpty(rson(A)))
rd(A) défini_ssi not(isEmpty(A)) et not(isEmpty(lson(A)))
rgd(A) défini_ssi not(isEmpty(A)) et not(isEmpty(lson(A)))
et not(isEmpty(rson(lson(A))))

rdg(A) défini ssi not(isEmpty(A)) et not(isEmpty(rsan(A)))
et not(isEmpty(lsan(rsan(A))))

axiomes

rg(rooting(p, U, rooting(q, V, W))) = rooting(q, rooting(p, U, V), W)

rd(rooting(q, rooting(p, U, V), W)) = rooting(p, U, rooting(q, V, W))

rgd(rooting(r, rooting(p, T, rooting(q, U, V)), W)) =
rooting(q, rooting(p, T, U), rooting(r, V, W))

rdg(rooting(r, T, rooting(p, rooting(q, U, V), W))) =
rooting(q, rooting(r, T, U), rooting(p, V, W))

5.2 Arbres H-équilibrés

Définition : On dit qu'un arbre binaire est H-équilibré si en tout noeud 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 → Int

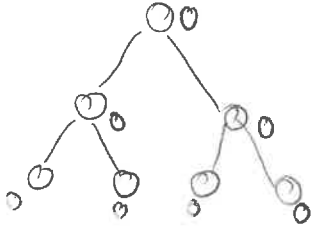
avec x : Element ; g, d : t_btree

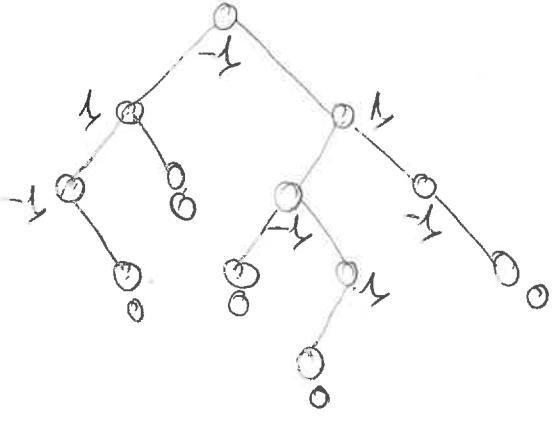
déséquilibre(empty) = 0

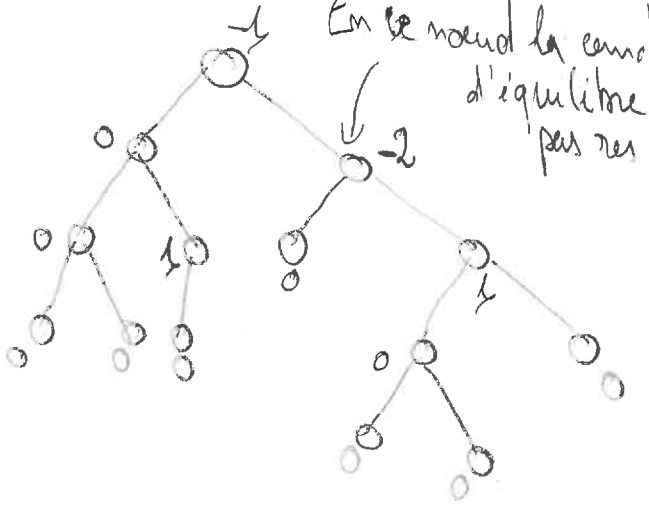
déséquilibre(rooting(x, g, d)) = hauteur(g) - hauteur(d)

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

Exemple : En chaque nœud, on indique la valeur de la fonction de déséquilibre







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

$$\log_2(n+1) \leqslant h+1 \leqslant 1{,}44 \log_2(n+2)$$

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.

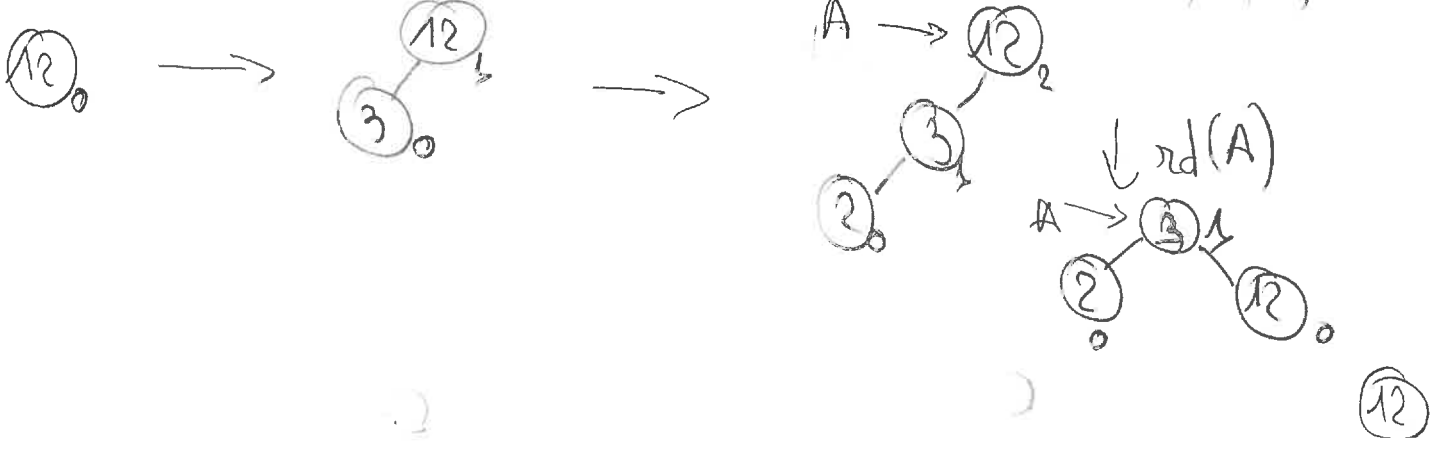
5.3 Arbres AVL

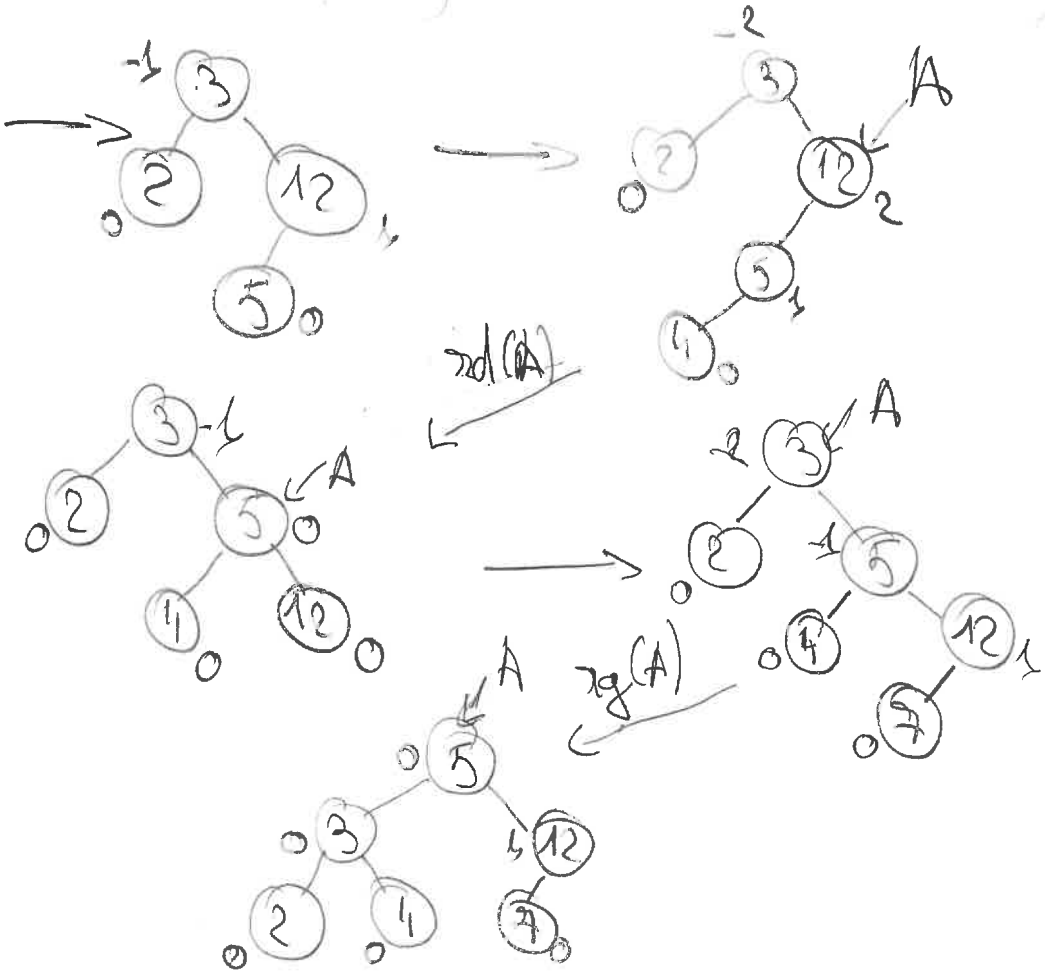
Les arbres AVL sont des arbres H-équilibrés inventés dans les années 1960 par Adelson-Velskii et Landis (d'où AVL)
Pour les arbres AVL la recherche, l'ajout et la suppression d'un élément sont en $\Theta(\log n)$ où n est le nombre de noeuds.
Pour l'ajout et la suppression, il est généralement nécessaire d'effectuer des rotations pour rétablir l'équilibre de l'arbre.

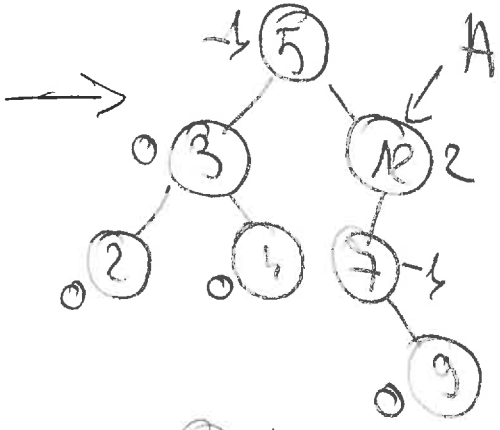
5.3.1 Exemple d'ajout dans un AVL

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

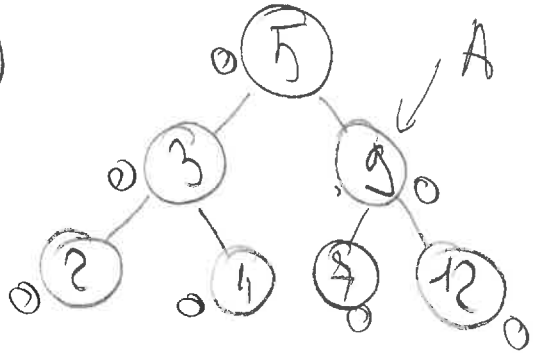
Exemple : On construit l'AVL associé à l'ajout successif de la suite d'éléments 12, 3, 2, 5, 4, 7, 9, 11, 14, 10

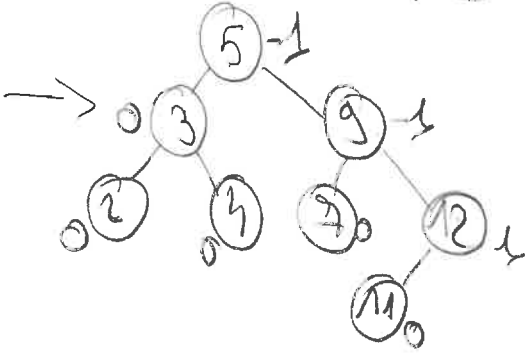


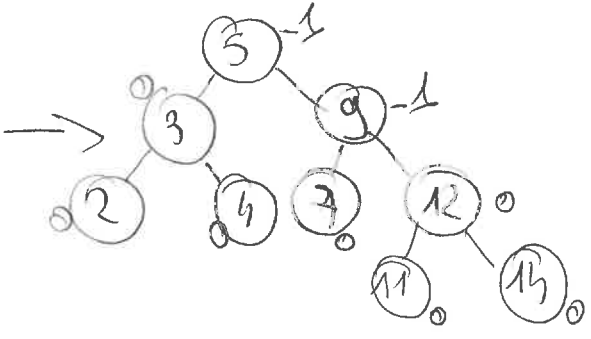

-1
3
2
12
5
-2
A
rd(A)
rg(A)

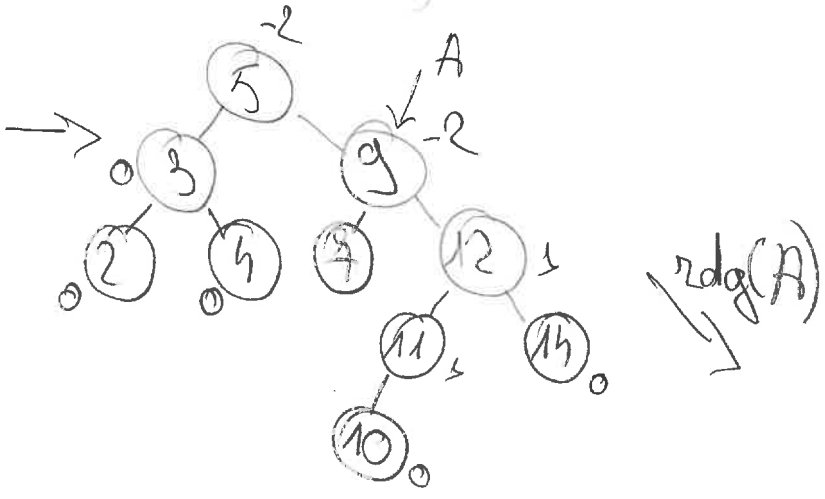

-1
5
A
3
12
2
2
4
7
-1
9

rotd(A)

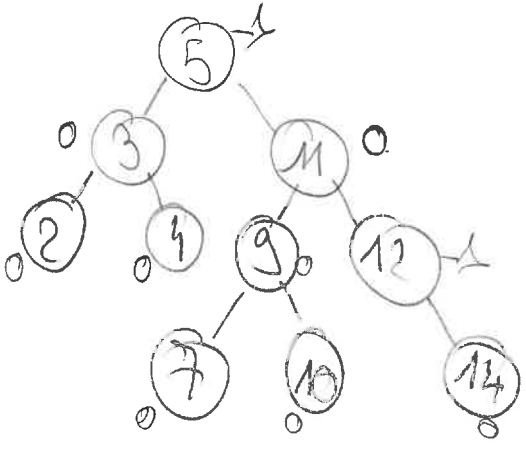

0
5
A
3
9
0
2
4
7
12


5
-1
3
9
1
2
4
7
12
1
11


5
-1
3
9
-1
2
4
7
12
11
13



rdg(A)



5.3.2 Rééquilibrage d'un AVL

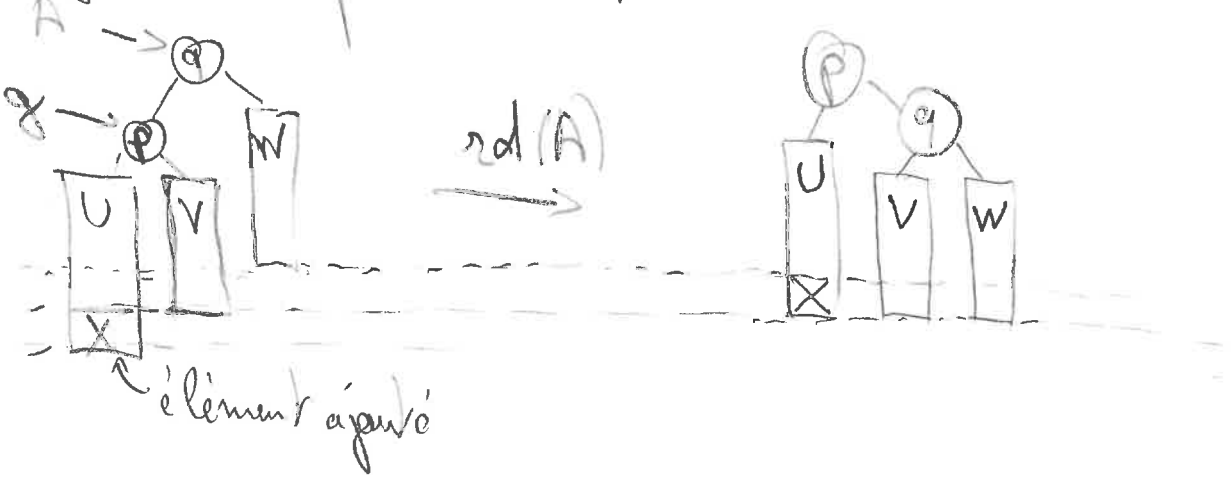
Différents cas possibles

Considérons $A=(v, g, d)$ un AVL et supposons que l'ajout se fait dans le sous-arbre g, que l'ajout augmente la hauteur de g et que g reste un AVL (c'est-à-dire le déséquilibre de g vaut 0)

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

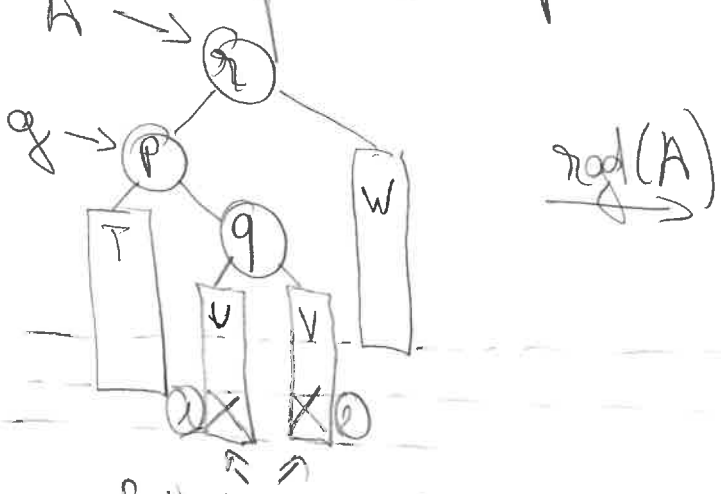
3) si déséquilibre(A)=1 avant l'ajout alors déséquilibre(A)=2 après l'ajout, A n'est plus un AVL. Il faut rééquilibrer A

Il y a 2 cas possibles. Si l'ajout se fait dans le fils gauche de g, on rééquilibre A par une rotation droite

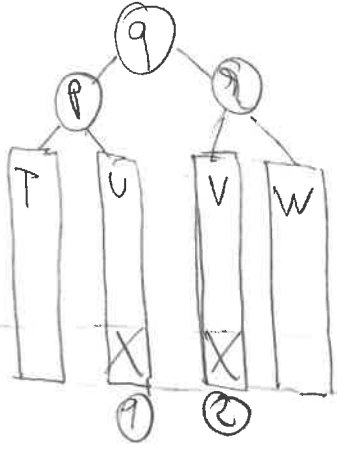


Si l'ajout se fait dans le fils droit de q alors déséquilibre(q)=1

On rééquilibre A par une rotation gauche-droite



rotd(A)



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

Remarquez 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 AV2 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'enchaînent 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

Idée de preuve : On cherche le nœud le plus bas où le déséquilibre est non nul sur le chemin de l'ajout de l'élément

On appelle A l'arbre enraciné à ce nœud 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 en plus une rotation pour rééquilibrer A.

On peut maintenant donner la spécification de l'opération rééquilibrer

rééquilibrer : t_avltree → t_avltree
déséquilibre : t_avltree → Int

avec A : t_avltree

précondition rééquilibre(A) est défini ssi déséquilibre(A) ∈ {-2,-1,0,1,2}

anciennes

rééquilibrer(A) = A si déséquilibre(A) = 0 ou 1 ou -1

rééquilibrer(A) = rd(A) si déséquilibre(A) = 2 et
déséquilibre(lson(A)) = 1

rééquilibrer(A) = rgd(A) si déséquilibre(A) = 2 et
déséquilibre(lson(A)) = -1

rééquilibrer(A) = rg(A) si déséquilibre(A) = -2 et
déséquilibre(rson(A)) = -1

rééquilibrer(A) = rdg(A) si déséquilibre(A) = -2 et
déséquilibre(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

ajt_avl : Element * t_avltree -> t_avltree

avec g, d : t_avltree ; v, e : Element

axiomes

ajt_avl(e, empty) = (e, empty, empty)

ajt_avl(e, (v, g, d)) = rééquilibrer((v, ajt_avl(e, g), d)) si e < v

ajt_avl(e, (v, g, d)) = rééquilibrer((v, g, ajt_avl(e, d))) si e > v

ajt_avl(e, (v, g, d)) = (v, g, d) si e = v

(je n'ai pas écrit les "accolades")

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 des opérations max qui renvoie l'élément maximal d'un AVL et dmax 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 → Element

dmaxe : t_avltree → t_avltree

avec t, g, d : t_avltree ; v : Element

préconditions maxe(t) définie ssi not(isEmpty(t))

dmaxe(t) définie ssi not(isEmpty(t))

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

maxe(rooting(v, g, d)) = maxe(d) si not(isEmpty(d))

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

dmaxe(rooting(v, g, d)) = rééquilibrer(rooting(v, g, dmaxe(d))) si not(isEmpty(d))

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

suppr_avl : Element * t_avltree -> t_avltree

avec e, v : Element ; g, d : t_avltree

(je n'ai pas écrit les "matching")

suppr_avl (e, empty) = empty

suppr_avl (e, (v, g, d)) = rééquilibrer((v, suppr_avl(e, g), d))
si e < v

suppr_avl(e, (v, g, d)) = rééquilibrer((v, g, suppr_avl(e, d)))
si e > v

suppr_avl(e, (v, g, empty)) = g si e = v

suppr_avl(e, (v, empty, d)) = d si e = v et not(isEmpty(d))

suppr_avl(e, (v, g, d)) = rééquilibrer((max(g), dmax(g), d)
si e = v et not(isEmpty(g)) et
not(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 Θ-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 annexe, 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 $1{,}44 \log n$). Comme les rotations se font en $\Theta(1)$. la complexité de la suppression est en $\Theta(\log n)$ dans le pire des cas.