

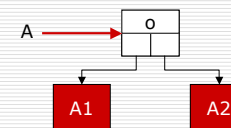
Algorithmes complexes Partie 1

Les structures arborescentes

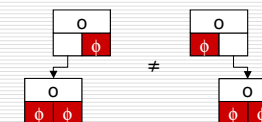
Définition

□ Arbre binaire: structure de donnée qui est soit vide (\emptyset) soit notée $A = \langle o, A1, A2 \rangle$

- o est un nœud de l'arbre appelé racine.
- $A1$ et $A2$ arbres binaires disjoints.
- $A1$ est le sous-arbre gauche et $A2$ est le sous-arbre droit.



□ Il est fondamental de prendre en compte la dissymétrie gauche droite des arbres binaires :
 $\langle o, \langle o, \emptyset, \emptyset \rangle, \emptyset \rangle \neq \langle o, \emptyset, \langle o, \emptyset, \emptyset \rangle \rangle$



Avec \emptyset : arbre vide

Spécification du type

□ Sorte Arbre
 Utilise nœud, élément, booléen

Opérations:

Arbre-vide	: Arbre \rightarrow Booléen
Créer-Arbre	: Nœud \otimes Arbre \otimes Arbre \rightarrow Arbre
Racine	: Arbre \rightarrow Nœud
Sd	: Arbre \rightarrow Arbre
Sg	: Arbre \rightarrow Arbre
Contenu	: Nœud \rightarrow Élément

Spécification du type

□ Avec:
 a : Arbre

Préconditions :

Racine(a) est défini si et seulement si Arbre-vide(a) = faux
 Sd(a) est défini si et seulement si Arbre-vide(a) = faux
 Sg(a) est défini si et seulement si Arbre-vide(a) = faux

Axiomes :

Racine(Créer-arbre($o, a1, a2$)) = o
 Sd(Créer-arbre($o, a1, a2$)) = $a2$
 Sg(Créer-arbre($o, a1, a2$)) = $a1$

Terminologie

- On confond souvent un nœud avec l'élément qu'il contient:
Créer-arbre(o,a1,a2) = Créer-arbre(e,a1,a2)
Où: Contenu(o) = e

On appelle fils droit (respectivement fils gauche) d'un nœud, la racine de son sous-arbre droit (respectivement sous-arbre gauche).

- Si un nœud np a pour fils (droit ou gauche) un nœud nf on dit que np est le père de nf.
Deux nœuds ayant le même père sont dits frères.
Un nœud qui a deux fils est appelé nœud interne ou point double.
Un nœud avec un seul fils est appelé point simple.
Un nœud sans fils est appelé feuille.

Observateurs à valeur entière sur les arbres binaires

- Taille d'un arbre : nombre de ses nœuds.
Taille : Arbre → entier

Avec :
a : Arbre
n : Nœud

```
fonction Taille(a : arbre) : entier {  
  Resultat : entier;  
  SI Arbre-vide(a)  
    ALORS resultat := 0;  
  SINON resultat := 1 + Taille(Sd(a)) + Taille(Sg(a));  
  FINSI  
  retourne resultat;  
}
```

Niveau d'un nœud

- Niveau d'un nœud : profondeur d'un nœud dans un arbre.

Observateur qui indique si un nœud appartient ou non à un arbre :
e : Nœud ⊗ Arbre → Booléen

D'où l'implémentation de niveau:

```
Fonction Niveau(n : Nœud , a : arbre ) : Entier {  
  Resultat : entier;  
  SI n = Racine(a)  
    ALORS resultat := 0;  
  SINON  
    SI n ∈ Sd(a)  
      ALORS resultat := 1 + Niveau(n,Sd(a));  
    SINON resultat := 1 + Niveau(n,Sg(a));  
  FINSI  
  FINSI  
  retourne resultat;  
}
```

Hauteur d'un arbre

- Hauteur d'un arbre : Profondeur de la feuille la plus éloignée de la racine de l'arbre.
Hauteur : Arbre → Entier

Avec :
a : Arbre

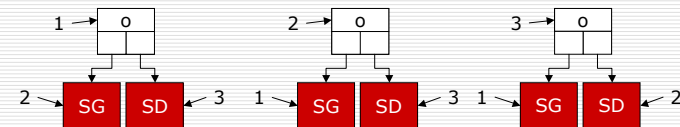
```
Fonction Hauteur(a : arbre) : entier {  
  resultat : entier;  
  SI Arbre-vide(a)  
    ALORS resultat := 0;  
  SINON resultat := Max(Hauteur(Sd(a)),Hauteur(Sg(a)))  
  + 1;  
  FINSI  
  return resultat;  
}
```

Cas particuliers

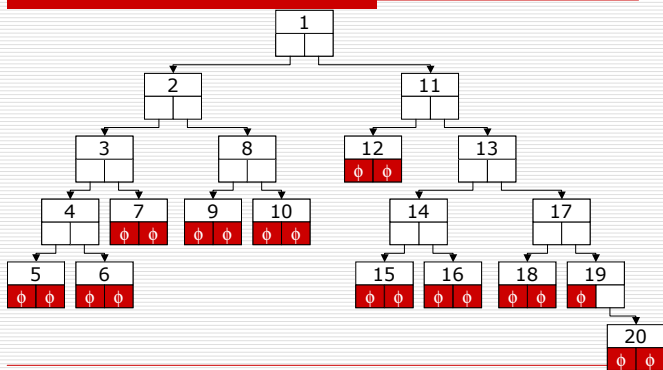
- Arbre binaire dégénéré ou arbre filiforme : l'arbre prend la forme d'une liste.
 - Un arbre binaire complet : chaque niveau est complètement rempli
 → au niveau h : 2^h nœuds
 $1+2+4+\dots+2^h = 2^{(h+1)} - 1$
- arbre binaire parfait : chaque niveau, sauf éventuellement le dernier est complet. Si le dernier n'est pas complet, les nœuds sont groupés le plus à gauche possible.
- Complétion locale d'un arbre binaire a : opération qui consiste à compléter a en un nouvel arbre qui soit localement complet.
 → il s'agit donc de rajouter des feuilles à a de sorte que tous les nœuds de a aient 2 fils.

Parcours d'un arbre

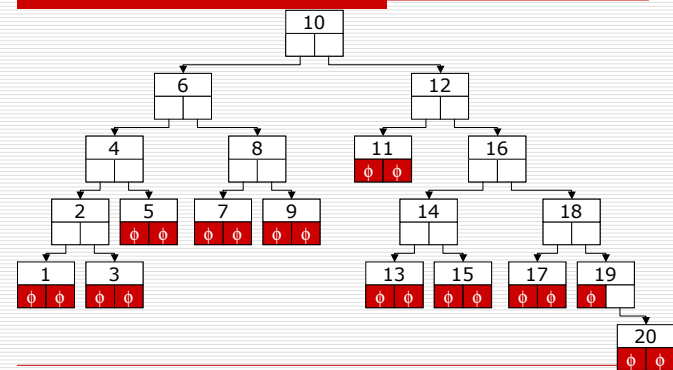
- Examen systématique des nœuds et éventuellement traitement : parcours ou traversée de l'arbre.
- 3 relations d'ordre liées à un parcours d'arbre binaire :
- Ordre préfixe ou préordre
 - Ordre infixe ou symétrique
 - Ordre suffixe ou postfixe



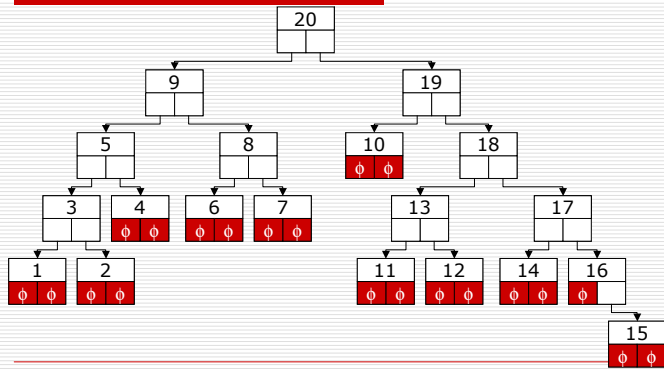
Ordre préfixe



Ordre infixe



Ordre suffixe



Parcours récursif

```
Procédure Parcours_récursif ( a : in_out arbre){  
  SI !(Arbre-vide(a))  
  ALORS  
    Traitementpréfixe;  
    Parcoursrécursif(Sg(a));  
    Traitementinfixe;  
    Parcoursrécursif(Sd(a));  
    ParcoursPostfixe;  
  FINSI;  
}
```

Parcours itératif

```
Procédure Parcours_itératif_p( a : in out arbre ) {  
  P:=pile;  
  P:= créer-pile;  
  REPETER  
    TANTQUE !(Arbre-vide(a)) FAIRE  
      Traitement-préfixe(Racine(a));  
      P:=Empiler(a);  
      a:=Sg(a);  
    FINTANTQUE;  
    a:=Sommet(P);  
    P:=Dépiler(P);  
    a:=Sd(a);  
  JUSQUE Pile-vide(P);  
}
```

Algorithme de recherche & Arbres triés

- Représentation d'un groupe de n éléments ordonnés par un arbre étiquetés ayant n nœuds. Chaque nœud contient un élément. La structure facilite la recherche, l'addition et la suppression d'un élément.
 - Arbre binaire trié : arbre binaire qui vérifie, pour tout nœud n , les conditions suivante :
 - 1) $\forall n' \in Sg(n) (Contenu(n') \leq Contenu(n))$
 - 2) $\forall n'' \in Sd(n) (Contenu(n'') \geq Contenu(n))$
- Parcours infixe d'un arbre binaire trié : restitue l'ensemble des éléments triés dans l'ordre croissant.

Addition d'un élément dans un arbre binaire trié

- Addition d'un élément tout en conservant un arbre binaire trié

Profil de l'opération d'addition:

Ajout : Élément \otimes Arbre \rightarrow Arbre

Avec:

e,r : Élément

a,G,D : Arbre

Axiomes:

Arbre-vide(a) = vrai \Rightarrow Ajout(e,a) = $\langle e, \phi, \phi \rangle$

$e \leq r \Rightarrow$ Ajout($\langle r, G, D \rangle$) = $\langle r, \text{Ajout}(e, G), D \rangle$

$e > r \Rightarrow$ Ajout($\langle r, G, D \rangle$) = $\langle r, G, \text{Ajout}(e, D) \rangle$

Ajout

Procédure Ajout(In e : Élément, In_Out a : Arbre);

SI Arbre-vide(a)

ALORS

a := Créer-arbre(e, ϕ , ϕ);

SINON

SI $e \leq$ Contenu(Racine(a))

ALORS

Ajout(e, Sg(a));

SINON

Ajout(e, Sd(a));

FINSI

FINSI

Fin

Addition d'un élément à la racine

- Contrainte : insertion d'un élément à la racine d'un nouvel arbre trié à construire.

Séparation des éléments de l'arbre d'origine en fonction de leur valeur par rapport à l'élément à insérer : «coupure»

Ce sont les éléments qui sont sur le chemin de recherche de l'élément ajouté qui déterminent la coupure.

Profil de l'opération "Ajouter-rac"

Ajouter-rac : Élément \otimes Arbre \rightarrow Arbre

Avec:

e, e', r : Élément

G, D : Arbre

Axiomes:

Arbre-vide(a) = vrai \Rightarrow Ajouter-rac(e, a) = $\langle \langle e \rangle, \phi, \phi \rangle$

Contenu(Racine(Ajouter-rac(e, $\langle r, G, D \rangle$))) = e

$e \geq$ Contenu(r) \Rightarrow ($e \in$ Sg(Ajouter-rac(e, $\langle r, G, D \rangle$)))

\Leftrightarrow ($e \in \langle r, G, \text{Sg}(\text{Ajouter-rac}(e, D)) \rangle$)

\wedge

($e \in$ Sd(Ajouter-rac(e, $\langle r, G, D \rangle$)))

\Leftrightarrow ($e \in$ Sd(Ajouter-rac(e, D)))

$e < r \Rightarrow$ ($e \in$ Sd(Ajouter-rac(e, $\langle r, G, D \rangle$)))

\Leftrightarrow ($e \in$ Sg(Ajouter-rac(e, G)))

\wedge

($e \in$ Sd(Ajouter-rac(e, $\langle r, G, D \rangle$)))

\Leftrightarrow ($e \in \langle r, \text{Sd}(\text{Ajouter-rac}(e, G), D) \rangle$)

Couper

- Répartition des éléments a en deux sous arbres g(éléments plus petits que e), et d(éléments plus grand que e).

```
Procédure Couper (e : Element, a : Arbre, In_Out g, d : Arbre){
  SI !(Arbre-vide(a))
  ALORS
    SI e < Contenu(Racine(a))
    ALORS
      Ajout(Contenu(Racine(a)),d);
      Parcours-ajout(Sd(a),d);
      Couper(e,Sg(a),g,d);
    SINON
      Ajout(Contenu(Racine(a)),g);
      Parcours-ajout(Sg(a),g);
      Couper(e,Sd(a),g,d);
    FINSI
  FINSI
}
```

Ajouter-rac

- Utilise couper pour créer un nouvel arbre dont la racine est e

```
Procédure Ajouter-rac(e : Elément, In_Out a : Arbre){
  g,d:Arbre;

  g:=φ;d:=φ;
  Couper(e,a,g,d);
  a:=Créer-arbre(Créer-nœud(e),g,d);
}
```

Suppression d'un élément

- Suppression d'un élément dans un arbre binaire de recherche d'abord déterminer sa place puis effectuer la suppression proprement dite :
 - Feuille : suppression immédiate
 - Point simple : remplacement de l'élément par son fils unique, et suppression du fils dans le sous arbre correspondant.
 - Point double : remplacement de l'élément parce lui qui lui est immédiatement inférieur, c'est à dire l'élément maximum de son sous arbre gauche et on supprime cet élément dans le sous arbre gauche.

Élément maximum d'un arbre donné

- Max qui donne l'élément maximum d'un arbre donné.
Profil:
Supprimer : Elément ⊗ Arbre → Arbre
Max : Arbre → Elément
- Avec:
E : Elément
a : Arbre
- Préconditions:
Suppression d'un élément dans lequel il ne se trouve pas est sans effet.
Supprimer(e,a) est toujours défini.
Max(a) est défini si et seulement si Arbre-vide(a) = faux

Axiomes

$\text{Supprimer}(e, \emptyset) = \emptyset$

$(e \in a) = \text{faux} \Rightarrow \text{Supprimer}(e, a) = a$

$\text{Supprimer}(e, \langle \langle e \rangle, \emptyset, \emptyset \rangle) = \emptyset$

$\text{Supprimer}(e, \langle \langle e \rangle, G, \emptyset \rangle) = G$

$\text{Supprimer}(e, \langle \langle e \rangle, \emptyset, D \rangle) = D$

$(G \neq \emptyset) \wedge (D \neq \emptyset)$

$\Rightarrow \text{Supprimer}(e, \langle \langle e \rangle, D, G \rangle) = \langle \text{Max}(G), \text{Supprimer}(\text{Max}(G), G), D \rangle$

$e < r \Rightarrow \text{Supprimer}(e, \langle r, G, D \rangle) = \langle r, \text{Supprimer}(e, G), D \rangle$

$e > r \Rightarrow \text{Supprimer}(e, \langle r, G, D \rangle) = \langle r, G, \text{Supprimer}(e, D) \rangle$

$\text{Max}(\langle r, G, \emptyset \rangle) = r$

$D \neq \emptyset \Rightarrow \text{max}(\langle r, G, D \rangle) = \text{Max}(D)$

Algorithme

```

Procédure Supprimer(e : Elément; In_Out a : arbre){
  SI e = Racine(a)
  ALORS
  ---Feuille
  SI Arbre-vide(Sg(a)) & Arbre-vide(Sd(a))
  ALORS a:=∅;
  FINSI

  ---Pointsimplegauche
  SI !(Arbre-vide(Sg(a)) & Arbre-vide(Sd(a)))
  ALORS a:=Sg(a);
  FINSI

  ---Pointsimpledroit
  SI Arbre-vide(Sg(a)) & !(Arbre-vide(Sd(a)))
  ALORS a:=Sd(a);
  FINSI

  ---Pointdouble
  SI !(Arbre-vide(Sg(a)) & !(Arbre-vide(Sd(a)))
  ALORS Contenu(Racine(a)):=Max(Sg(a));
  Supprimer(Max(Sg(a)),Sg(a));
  FINSI

  SINON
  SI e < Racine(a)
  ALORS Supprimer(e,Sg(a));
  SINON Supprimer(e,Sd(a));
  FINSI
  FINSI
}
  
```

Recherche d'un élément

- Fonction de recherche d'un élément : indique si l'élément se trouve ou non dans l'arbre.

Si e est l'élément contenu par le nœud n, on confond $\langle n, G, D \rangle$ et $\langle e, G, D \rangle$

Profil de la fonction de recherche :

Recherche : Elément @ Arbre → Booléen

Avec:

e,r : Elément

a,G,D : Arbre

Axiomes:

$\text{Arbre-vide}(a) \Rightarrow \text{Recherche}(e, a) \Rightarrow \text{faux}$

$e=r \Rightarrow \text{Recherche}(e, \langle r, G, D \rangle) = \text{vrai}$

$e < r \Rightarrow \text{Recherche}(e, \langle r, G, D \rangle) = \text{Recherche}(e, G)$

$e > r \Rightarrow \text{Recherche}(e, \langle r, G, D \rangle) = \text{Recherche}(e, D)$

Recherche

```

Fonction Recherche(e : élément, a : arbre) : booléen {
  Resultat : booléen;
  SI Arbre-vide(a)
  ALORS Resultat := faux
  SINON
  SI (e = racine(a))
  ALORS
  Resultat := vrai
  SINON
  SI ( e < Racine(a) )
  ALORS Resultat :=
  Recherche(e,Sg(a))
  SINON Resultat :=
  Recherche(e,Sd(a))
  FINSI
  FINSI
  Retourne resultat;
}
  
```

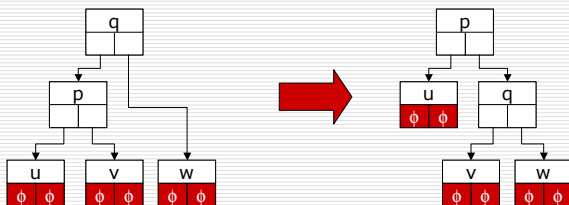
Les arbres équilibrés

- ❑ Opérations de recherche, d'adjonction et de suppression d'un élément dans un arbre binaire : complexité proportionnelle à la profondeur de l'arbre ($\theta(\log 2n)$).
 - ❑ Quand un arbre binaire trié dégénère en une liste : complexité des opérations précédentes en $\theta(n)$.
Classe d'arbres dits "équilibrés" : profondeur toujours une fonction logarithmique du nombre de nœuds.
 - ❑ En cas de déséquilibre : on définit pour ces arbres des opérations de "rééquilibrage" dont la complexité est logarithmique.
-

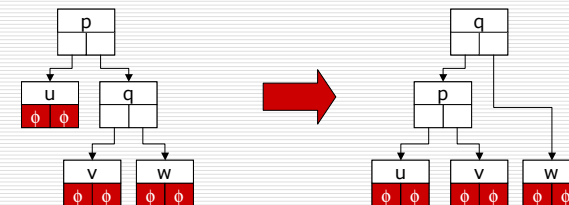
La notion de rotation

- ❑ Les algorithmes de rééquilibrage utilisent des transformations locales appelées rotation.
Il existe des rotations simples et des rotations doubles.
 - ❑ Il existe quatre fonction de rotation : la rotation gauche, droite, gauche-droite et droite-gauche
-

Rotation droite

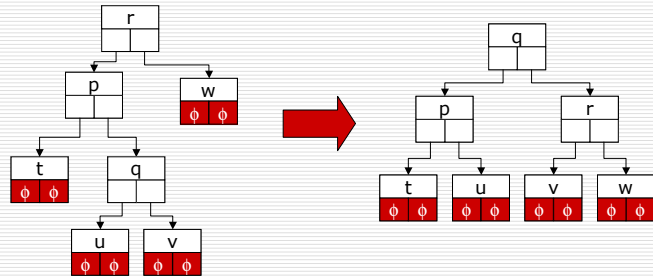


Rotation gauche



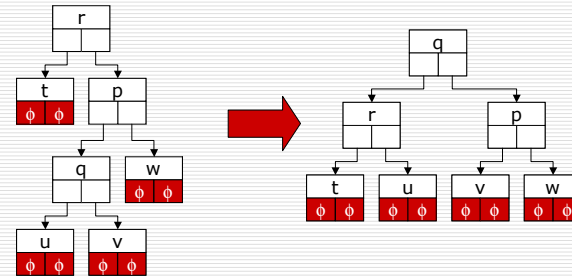
Rotation gauche droite

- Rotation gauche sur le sous arbre gauche suivie d'une rotation droite sur l'arbre complet.



Rotation droite gauche

- Rotation droite sur le sous arbre droit suivie d'une rotation gauche sur l'arbre complet.



Profil des quatre rotations

- Rg : Arbre → Arbre
- Rd : Arbre → Arbre
- Rgd : Arbre → Arbre
- Rdg : Arbre → Arbre

Avec:
 A, T, U, V, W : Arbre
 p, q, r : Nœud

Spécifications

- Préconditions :
 Rd(A) est défini si et seulement si !(Arbre-vide(Sg(A)))
 Rg(A) est défini si et seulement si !(Arbre-vide(Sd(A)))
 Rgd(A) est défini si et seulement si !(Arbre-vide(Sd((Sg(A)))))
 Rdg(A) est défini si et seulement si !(Arbre-vide(Sg((Sd(A)))))

Axiomes:
 $Rd(\langle q, \langle p, U, V \rangle, W \rangle) = \langle p, U, \langle q, V, W \rangle \rangle$
 $Rg(\langle p, U, \langle q, V, W \rangle \rangle) = \langle q, \langle p, U, V \rangle, W \rangle$
 $Rgd(\langle r, \langle p, T, \langle q, U, V \rangle \rangle, W \rangle) = \langle q, \langle p, T, U \rangle, \langle r, V, W \rangle \rangle$
 $Rdg(\langle r, T, \langle p, \langle q, U, V \rangle, W \rangle \rangle) = \langle q, \langle r, T, U \rangle, \langle p, V, W \rangle \rangle$

Les arbres AVL (Adelson-Velskii et Landis)

- Arbre binaire trié AVL : en tout nœud $\langle o, g, d \rangle$ de l'arbre, les hauteurs des sous arbres gauche et droit diffèrent au plus de 1.
Fonction déséquilibre : mesure la différence des hauteurs des sous arbres gauche et droit.

Profil:
Déséquilibre : Arbre \rightarrow Entier

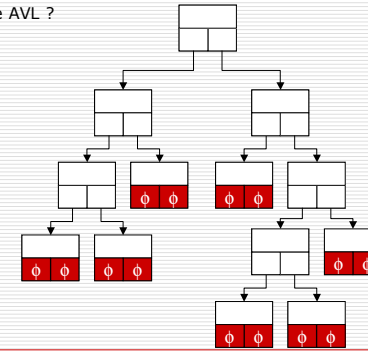
Avec:
 a, g, d, t : Arbre

Axiomes:
Arbre-vide(a) \Rightarrow Déséquilibre(a) = 0
Déséquilibre($\langle o, g, d \rangle$) = hauteur(g) - hauteur(d)

Définition d'un arbre AVL
AVL(t) $\Leftrightarrow \forall s \in \{\text{SousArbre}(t)\}, \text{Déséquilibre}(s) \in \{-1, 0, 1\}$

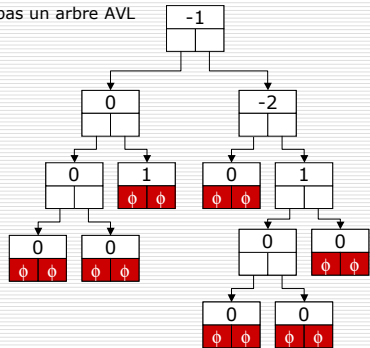
Exemple

- arbre AVL ?



Exemple

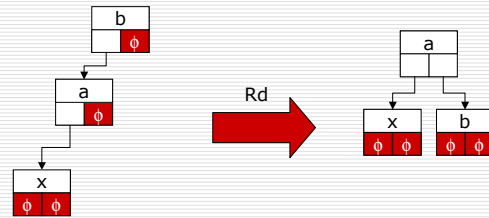
- \rightarrow ceci n'est pas un arbre AVL



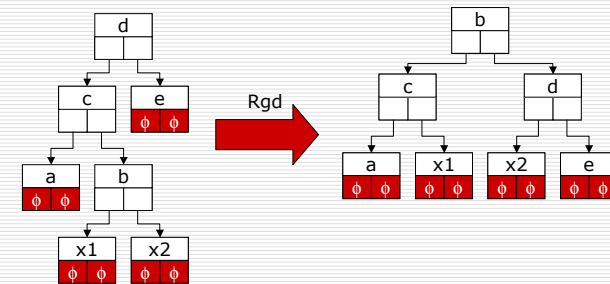
Principes généraux de rééquilibrage

- $T = \langle r, G, D \rangle$ arbre AVL
Adjonction de l'élément x sur une feuille de G
- Déséquilibre de $T = 0$ avant l'adjonction, il vaut 1 après : T reste un AVL et sa hauteur a augmenté de 1.
- Déséquilibre de $T = -1$ avant l'adjonction, il vaut 0 après : T reste un AVL et sa hauteur n'est pas modifiée.
- Déséquilibre de $T = 1$ avant l'adjonction, il vaut 2 après : T n'est plus équilibré, il faut le restructurer.

1er cas



2ème cas



Adjonction dans un arbre AVL

- Arbre AVL, étant des arbres binaires triés : utilisation des méthodes précédentes pour rechercher, ajouter ou supprimer un élément. Un ajout (aux feuilles) ou une suppression dans un arbre AVL peuvent le déséquilibrer. On procède à son rééquilibrage.

Principe de l'adjonction

- Nouvelles opérations : Ajouter-avl et Rééquilibrer

Profils:

Ajouter-avl : Elément \otimes Arbre \rightarrow Arbre
Rééquilibrer : Arbre \rightarrow Arbre

Avec:

t, g, d : Arbre
x, r : Elément

Précondition :

Rééquilibrer(t) est défini si et seulement si Déséquilibrer(t) \in $\{-2, -1, 0, 1, 2\}$

Axiomes

- Déséquilibre(t) = 0, 1 ou -1 \Rightarrow Rééquilibrer(t) = t
Déséquilibre(t) = 2 \wedge Déséquilibre(g(t)) = 1 \Rightarrow Rééquilibrer(t) = Rd(t)
Déséquilibre(t) = -2 \wedge Déséquilibre(d(t)) = -1 \Rightarrow Rééquilibrer(t) = Rg(t)
Déséquilibre(t) = 2 \wedge Déséquilibre(g(t)) \in {0, -1} \Rightarrow Rééquilibrer(t) = Rgd(t)
Déséquilibre(t) = -2 \wedge Déséquilibre(d(t)) \in {0, 1} \Rightarrow Rééquilibrer(t) = Rdg(t)

Algorithme de rééquilibrage et d'adjonction

```
Procédure Rééquilibrer(In_Out a : Arbre){
SI (!Arbrevide(arbre))
ALORS
  SI Hauteur(Sg(a)) - Hauteur(Sd(a)) = 2
  ALORS
    SI Hauteur(Sg(Sg(a)) - Hauteur(Sd(Sg(a))) = 1
    ALORS a := Rd(a); ---1er cas ci-dessus
    SINON a := Rgd(a); ---2ème cas ci-dessus
  FINSI
  SINON
    SI Hauteur(Sd(a)) - Hauteur(Sg(a)) = 2
    ALORS
      SI Hauteur(Sd(Sd(a))) - Hauteur(Sg(Sd(a))) = 1
      ALORS a := Rg(a);
      SINON a := Rdg(a);
    FINSI
  FINSI
  FINSI
}
}
```

Ajout & Suppression dans un arbre AVL

- Ajout d'éléments dans un arbre AVL : comme dans un arbre binaire trié ordinaire.
Après ajout, l'arbre peut ne plus être un arbre AVL : il faut le rééquilibrer

```
Procédure Ajouter-avl(e : Elément, In_Out a : Arbre){
  Ajouter(e,a);
  Rééquilibrer(a);
}
```

- Suppression d'éléments dans un arbre AVL : comme dans un arbre binaire trié ordinaire.
Après suppression, l'arbre peut ne plus être un arbre AVL : il faut le rééquilibrer

```
Procédure Supprimer-avl(e : Elément, In_Out a : Arbre){
  Supprimer(e,a);
  Rééquilibrer(a);
}
```