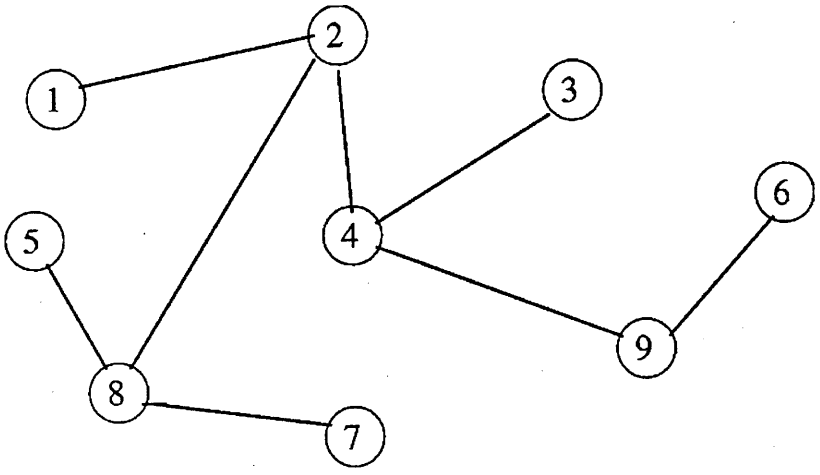
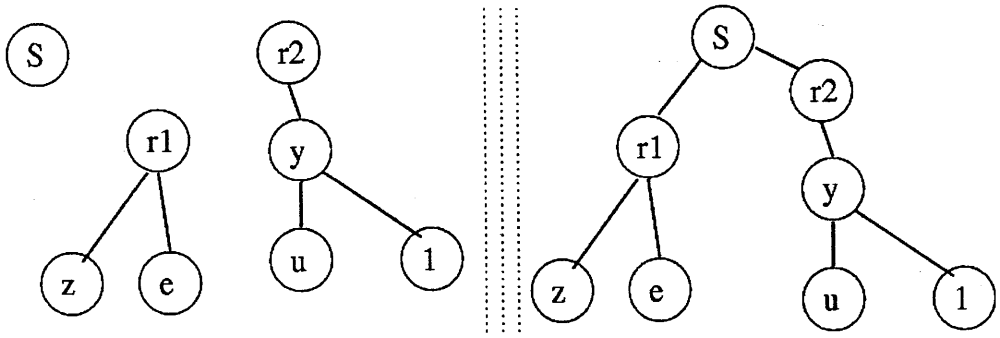
Un arbre est un graphe

- Un arbre est un graphe **acyclique** et **connexe** (avec un noeud particulier, la **racine**)



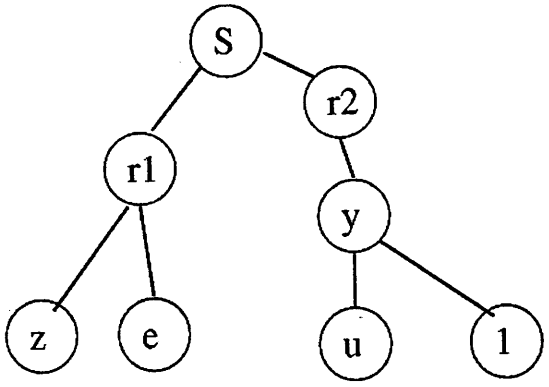
Définition récursive

- Base: Un noeud unique est un arbre (la **racine**)
- Récurrence: Soient $\mathbf{A_1}...\mathbf{A_n}$, **n** arbres de racines $\mathbf{r_1}...\mathbf{r_n}$. Soit **S** un noeud sans arc. Un nouvel arbre est construit avec **S** comme racine en ajoutant les arcs de **S** à $\mathbf{A_1}...\mathbf{A_n}$.



Vocabulaire (les noeuds)

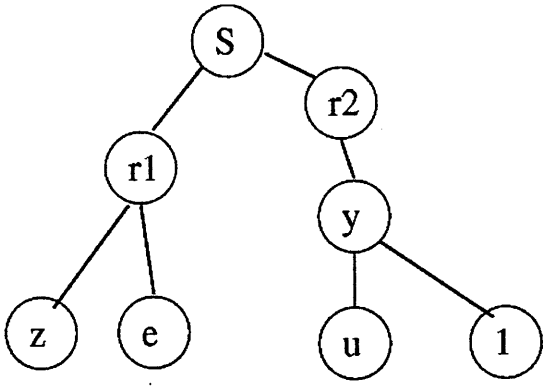
- **S** est la **racine**
- **y** est le **père** de **u**
- **u** est le **fils** de **y**
- **e** est le **frère** de **z** (même père)



- **S, r_2,** et **y** sont les **ancêtres** de **y**
- Un noeud sans fils est une **feuille** (ex: **z,e...**)
- Noeud **intérieur:** noeud qui n'est pas une feuille

Vocabulaire (tailles)

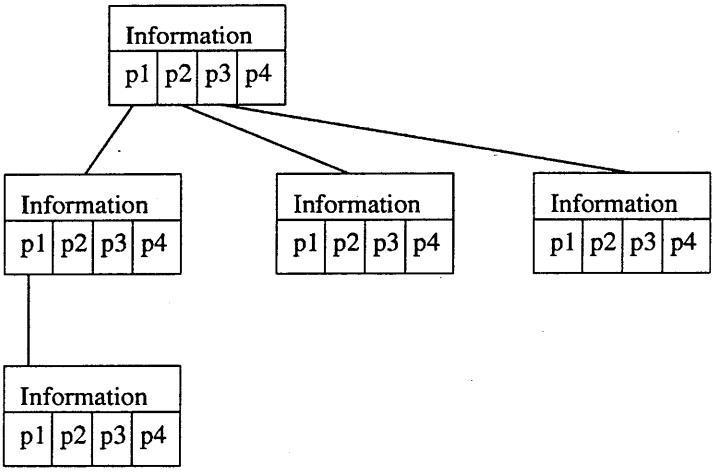
- **Hauteur** d'un noeud: longueur du chemin entre la racine et ce noeud. (**h(y)**=2)
- **Profondeur** d'un arbre: longueur du plus long chemin entre la racine et un noeud (une feuille) (prof=3)



- **Sous–arbre**: Un noeud et tous ses descendants (en vert)

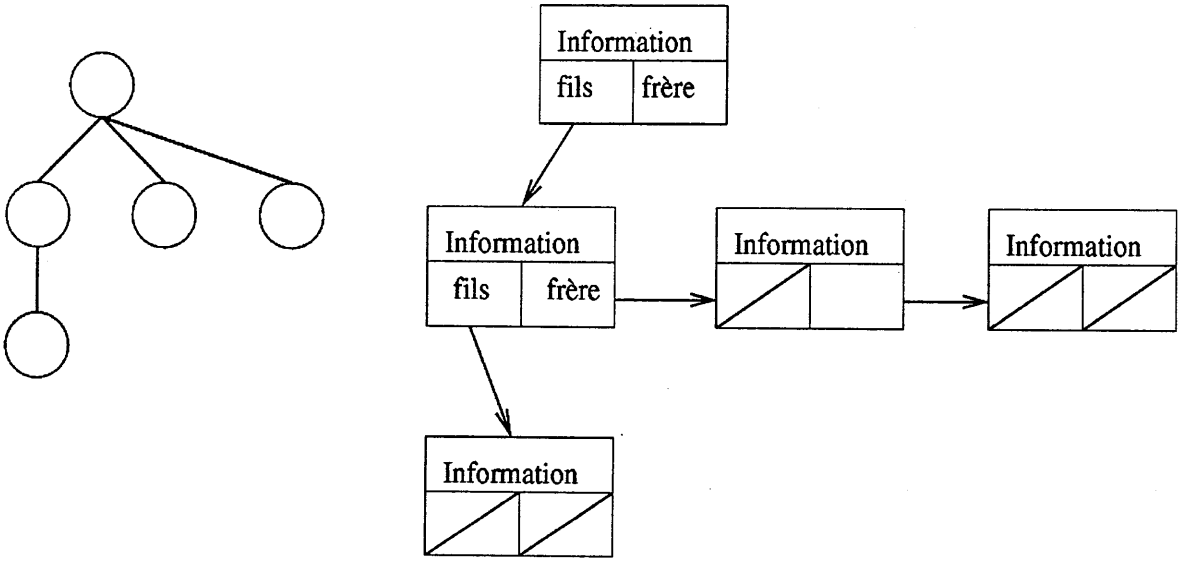
Représentation (tableau de pointeurs)

- Le plus simple: chaque noeud est une structure contenant l'information stockée et un tableau de pointeurs sur ses fils
- Le nombre de fils est limité par la taille du tableau
- Perte d'un peu de place
- Accès très rapide



Représentation (Liste de frères)

- On peut considérer chacun des niveaux de profondeur égale comme une liste.



Comparatif des méthodes

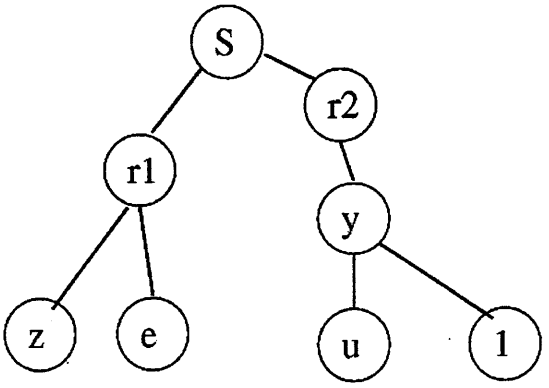
- Représentation par tableau de fils
  - Accès à un fils: O(1)
  - Accès ts les fils: O(n)
- Représentation par liste de frère
  - Accès à un fils: O(n)
  - Accès ts les fils: O(n)
  - Pas de place perdue
  - Pas de limitation de la taille de l'arbre

Parcours en profondeur

- Trivial!
- DFS(Noeud n) {
  - // action **préfixe**
  - Pour tous les fils de n
    - DSF(n)
  - // action **postfixe**
  - }

Types de parcours DFS

- Essayez un parcours DFS avec
  - un affichage préfixe,
  - un affichage postfixe

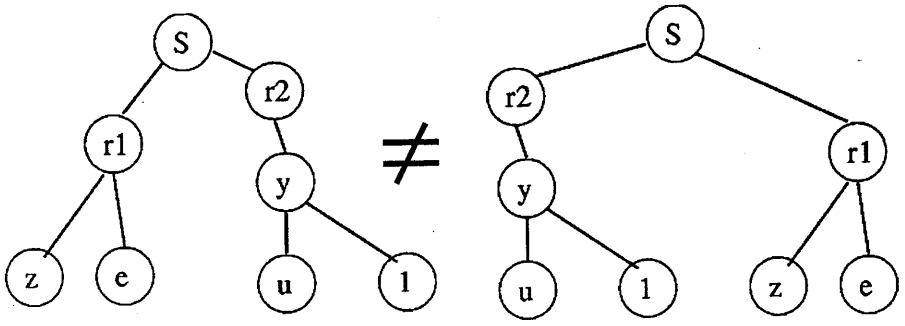


Petits exercices ludiques

- Vous disposez d'un arbre implémenté sous forme de tableau de fils, donnez des algorithmes pour
  - Lister toutes les feuilles de l'arbre
  - Calculer la profondeur d'un arbre

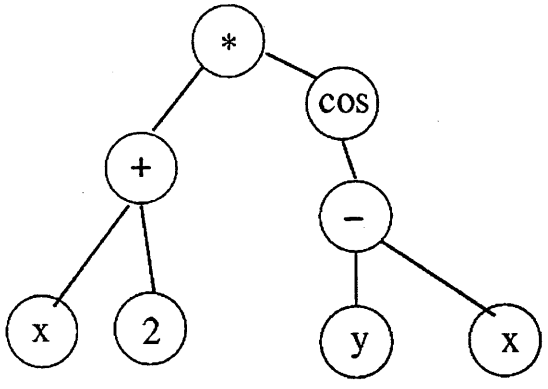
Arbres ordonnés

- Arbres ou l'ordre des fils a une importance
- On parle alors de frères **aînés** ou **cadets**...
- et de noeud plus à **gauche** ou plus à **droite**



Arbres d'expressions

- Comment représenter une expression arithmétique?




```
(x+2)*cos(y-x)
```

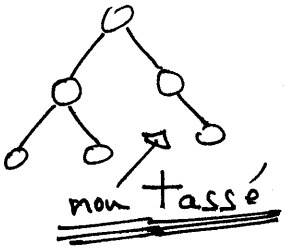
Arbres binaires: définition

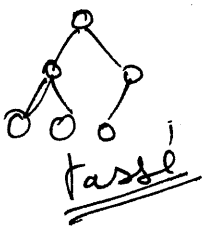
- Un arbre binaire est un arbre dont chaque noeud a au plus deux fils (fils droit, fils gauche)
- Les arbres binaires sont des arbres **ordonnés**
- Le fils droits et fils gauches ne sont donc pas échangeables!
- Si un arbre classique a au moins un noeud, l'arbre vide est un arbre binaire!

Implémentation des arbres binaire

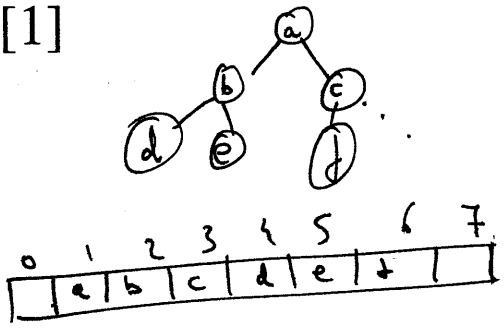
- Méthode chaînée
- Noeud: Tableau de fils, avec 2 pointeurs, fils droit et fils gauche

Stockage des arbres binaires compacts





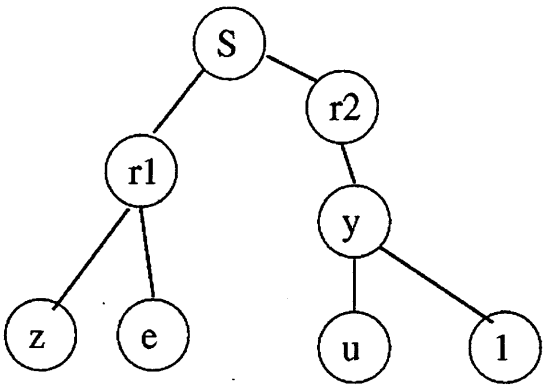
- Sous forme d'un tableau unique t[2^(depth+1)]
- la racine est stockée en t[1]
- Soit le noeud d'indice n
  - Fils gauche: 2*n
  - Fils droit: 2*n+1



- Très rapide!
- Utile uniquement si arbre binaire **tassé**

Parcours DFS pour arbres binaire

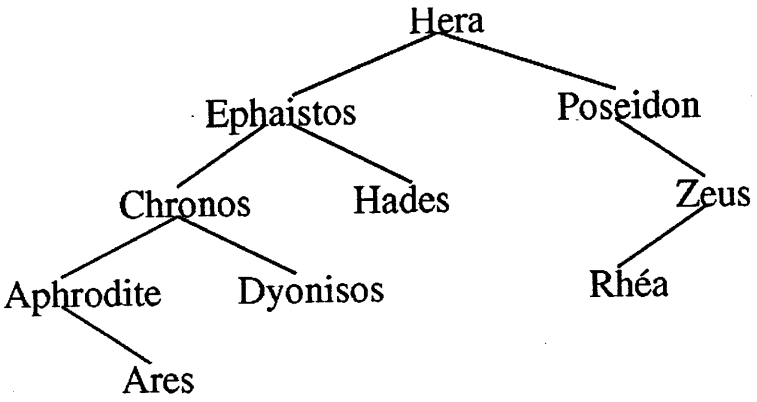
- DFS(Noeud n) {
  - // action **préfixe**
  - DSF(n.fils_gauche)
  - // Action **infixe**
  - DFS(n.fils_droit)
  - // action **postfixe**
  - }
- ordre infixe en plus



Arbres binaires de recherche

- Idée: maintenir la propriété suivante pour chacun des noeuds d'un arbre
  - Tout ce qui est plus petit que le noeud est à gauche
  - Tout ce qui est plus grand que le noeud est à droite

Placer Gaia, Apollon, Thanatos, Arthémis, Athéna



Algo d'insertion

- Ajouter(Noeud n, Élément e)
  - Si (e<n.elt) // Insertion gauche
    - Si (n.fg==null)
      - n.fg= new Noeud(e) // Il y a de la place
    - Sinon
      - Ajouter(n.fg,e)
  - Sinon
    - Si (n.fd==null)
      - n.fd= new Noeud(e) // Il y a de la place
    - Sinon
      - Ajouter(n.fd,e)

Algo de recherche

- Élément rechercher(Noeud n, Élément e)
  - Si (e==n.elt)
    - return e;
  - Si (e<n.elt)
    - Si (n.fg==null)
      - throw new NotFoundException
    - Sinon
      - return rechercher(n.fg,e)
  - Sinon
    - Si (n.fd==null)
      - throw new NotFoundException
    - Sinon
      - return rechercher(n.fd,e)

Algo de suppression

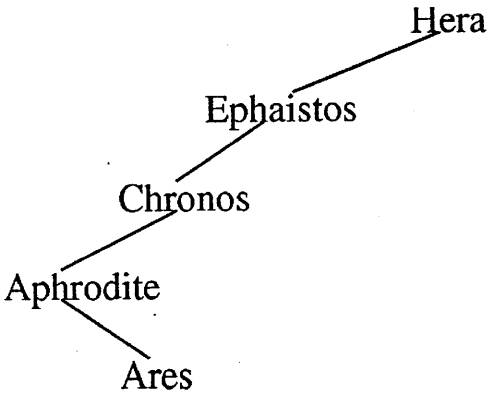
- extraire(Noeud n, Élément e)
  - Si (e==n.elt)
    - supprimer_noeud(n)
    - return n.elt
  - Si (e<n.elt)
    - Si (n.fg==null)
      - throw new NotFoundException
    - Sinon
      - return extraire(n.fg,e)
  - Sinon
    - Si (n.fd==null)
      - throw new NotFoundException
    - Sinon
      - return extraire(n.fd,e)

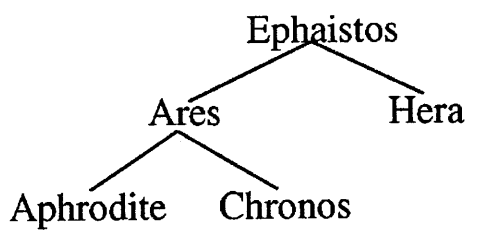
Faire remonter les fils

- supprimer_noeud(noeud n)
- Fait remonter le fils droit ou le fils gauche, suivant si ils existent (peu importe)

Equilibrage des arbres

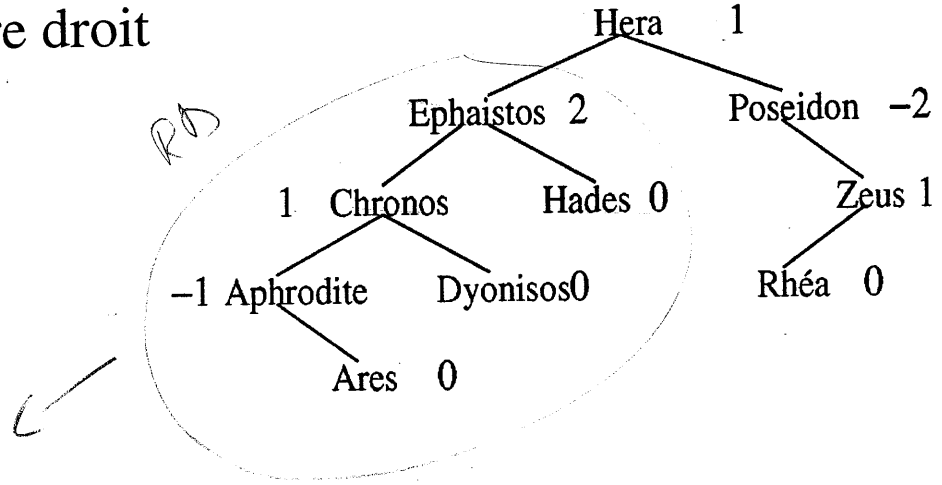
- Cout des opérations de recherche et d'ajout:
  - O(profondeur de l'arbre)
- Idée: maintenir la profondeur de l'arbre minimale



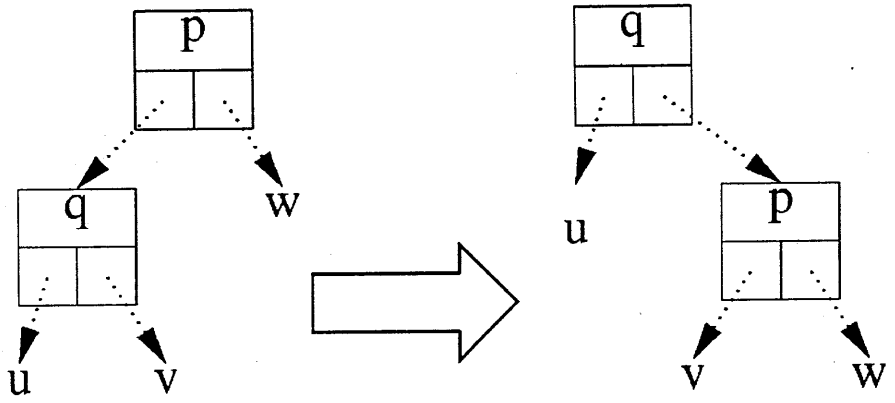


Outils pour l'équilibrage

- Modifications de l'arbre
- Principe: rotations (échange de cellules)
- Outils de décision: Le **déséquilibre**, différence entre la profondeur du sous arbre gauche et du sous–arbre droit

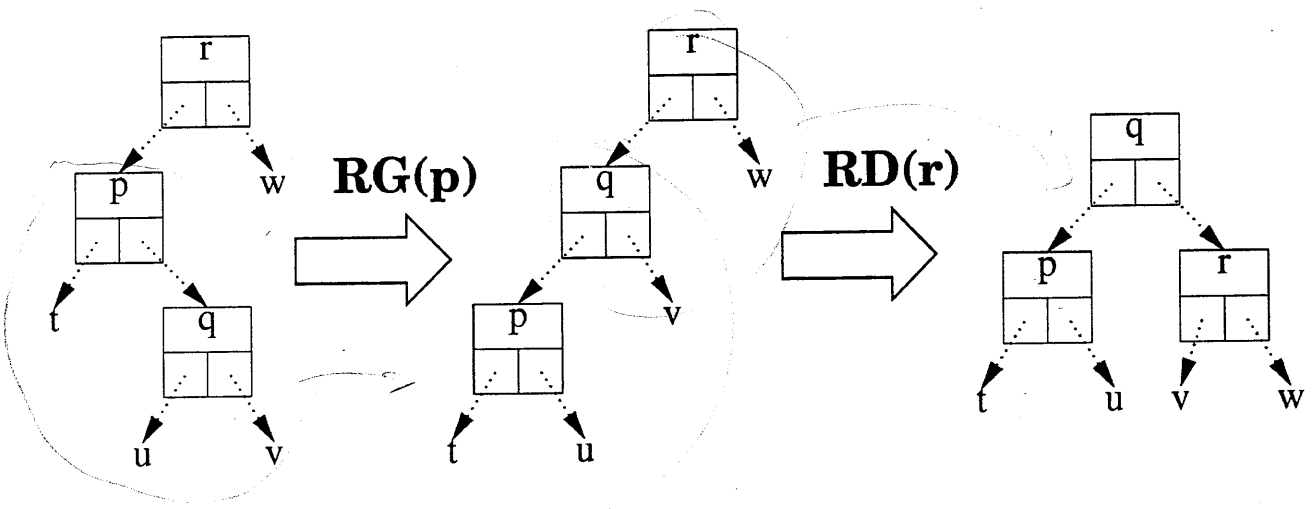


Rotation à droite (RD)



•La rotation à gauche (RG) est purement symétrique à la rotation à droite

Rotation gauche/droite (RGD)



- RDG Idem RGD en symétrique

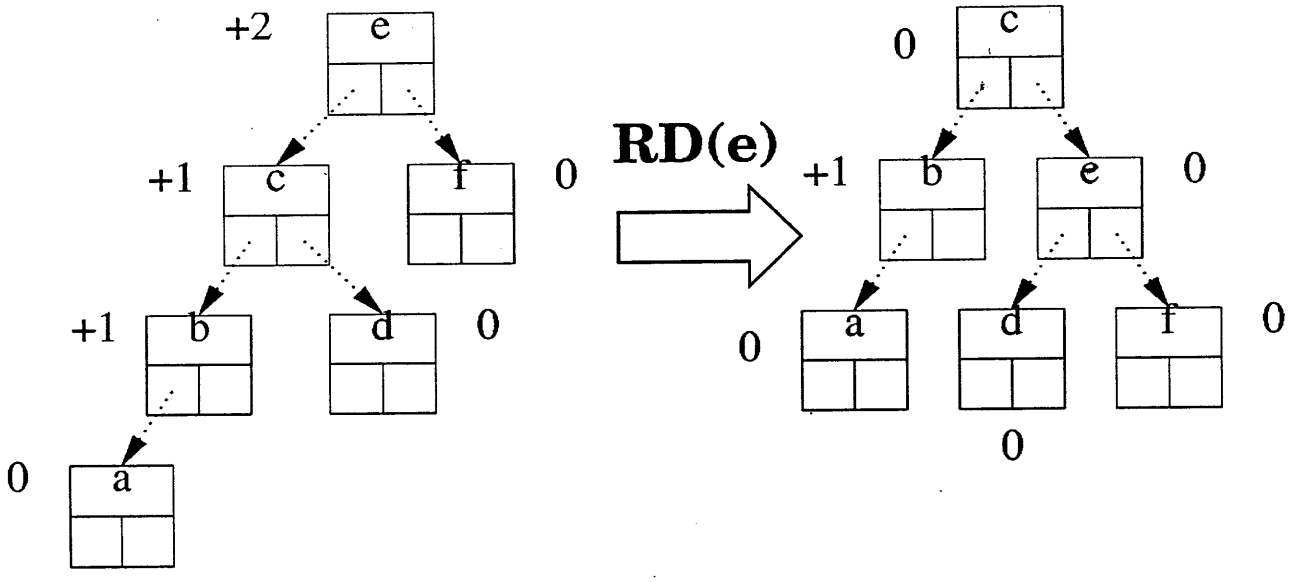
Maintenir les arbres équilibrés

- A tout moment, la valeur abs. du déséquilibre de chacun des sous arbre ne dépasse pas 1
- Sinon, rééquilibrage! ssi déséquilibre = -2 ou +2.
- Appelé lors des insertions et suppressions

Quel rotation appliquer?

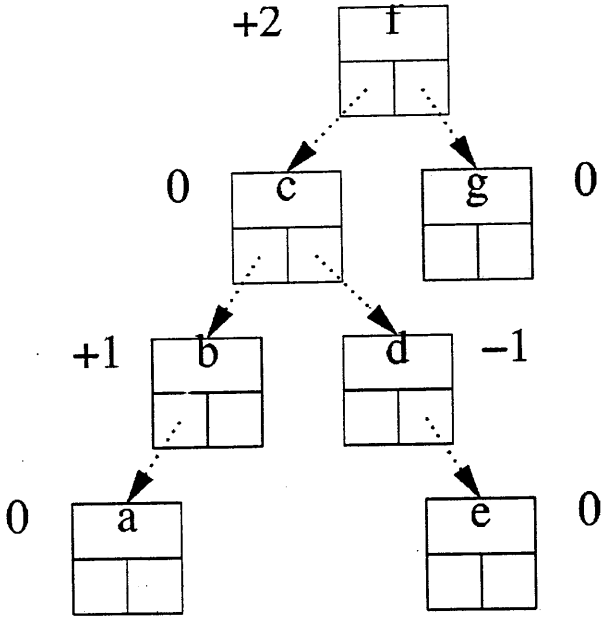
- Rééquilibrer(Noeud n)
  - d=n.déséquilibre()
  - Si (d==2)
    - Si (fg.déséquilibre()==−1)
      - n.rotationGD()
    - else
      - n.rotationD()
  - Si (d==−2)
    - Si (fd.déséquilibre()==1)
      - n.rotationDG()
    - else
      - n.rotationG()

Rééquilibrage: Cas 1 (avec +2)

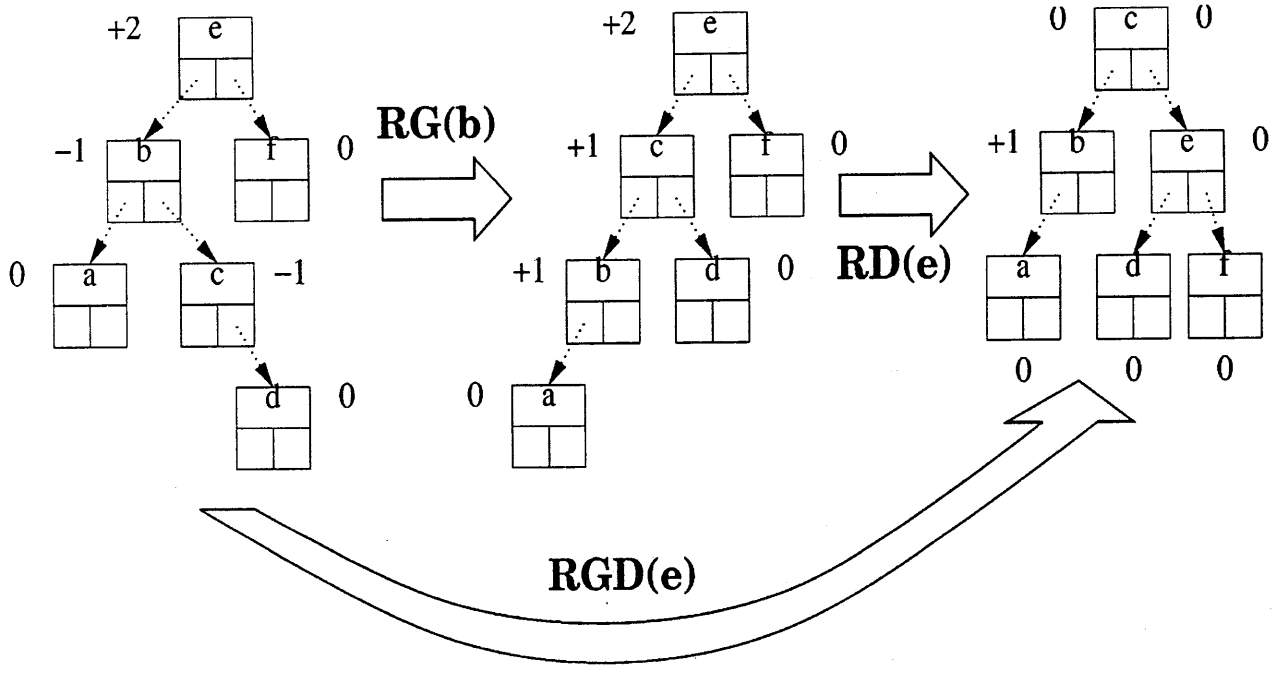


Rééquilibrage: Cas 2 (avec +2)

- N'arrive jamais si arbre maintenu équilibré!



Rééquilibrage: Cas 3 (+2)



```
import java.io.*;
import java.util.*;

/** class Arbre
    <p> Implémente un arbre de recherche. Il est
    <li>
    <ul> ordonné
    <ul> binaire
    </li>
*/
public class Arbre {
    protected class Noeud {
        protected Info info;
        protected Noeud fg;
        protected Noeud fd;

        public Noeud(Info info)
        {
            this.fg = null;
            this.fd = null;
            this.info = info;
        }
        public Noeud(Noeud n)
        {
            fg = n.fg;
            fd = n.fd;
            info = n.info;
        }
        public void ajoute(Info i)
        {
            if(info.compareTo(i)>0)
                if(fg≡null)
                    fg = new Noeud(i);
                else
                    fg.ajoute(i);
            else
                if(fd≡null)
                    fd = new Noeud(i);
                else
                    fd.ajoute(i);

        }
        public boolean estFeuille()
        {
            return (fg≡null ∧ fd≡null);
        }

        public int profondeur()
        {
            if(estFeuille())
                return 0;
            if(fg≡null)
                return 1+fd.profondeur();
            if(fd≡null)
                return 1+fg.profondeur();
            return 1+Math.max(fg.profondeur(),fd.profondeur());
        }

        public Info recherche(Info info)
        {
            if(this.info.compareTo(info)≡0)
                return info;
            Noeud cible;

            if(this.info.compareTo(info)>0)
                cible = fg;
            else
                cible = fd;

            if(cible≡null)
                return null;
            else
                return cible.recherche(info);
        }
    }

    public Arbre(Info info)
    {
        // Tout arbre de recherche a au moins un noeud!
        racine = new Noeud(info);
    }
    protected String linearise(Noeud n)
    {
        if(n≡null)
            return "/";
        return n.info.toString()+"("+linearise(n.fg)+","+linearise(n.fd)+")";
    }

    public String toString()
    {
        return linearise(racine);
    }

    public void ajoute(Info i)
    {
        racine.ajoute(i);
        nbElements++;
    }

    public int profondeur()
    {
        return racine.profondeur();
    }
    public Info recherche(Info info)
    {
        return racine.recherche(info);
    }

    protected Noeud racine;
    protected int nbElements;

}
```

```
import java.io.*;
import java.util.*;

/** class ArbreE
    <p> Implémente un arbre de recherche toujours equilibré. Il est
    <li>
    <ul> ordonné
    <ul> binaire
    </li>
*/
public class ArbreE extends Arbre {
    protected class NoeudE extends Arbre.Noeud{
        protected int hauteur;

        public NoeudE(Info info, int hauteur)
        {
            super(info);
            this.hauteur = hauteur;
        }

        public NoeudE(NoeudE n)
        {
            super(n);
            hauteur = n.hauteur;
        }

        public void reequilibrer()
        {
            NoeudE fg = (NoeudE)this.fg;
            NoeudE fd = (NoeudE)this.fd;

            int d=desequilibre();
            if(d==2)
                {
                if(fg.desequilibre()==-1)
                    rotationGD();
                else
                    rotationD();
                }
            if(d==-2)
                if(fd.desequilibre()==1)
                    rotationDG();
                else
                    rotationG();
        }

        public void ajoute(Info i)
        {
            if(info.compareTo(i)>0)
                if(fg==null)
                    fg = new NoeudE(i,1);
                else
                    fg.ajoute(i);
            else
                if(fd==null)
                    fd = new NoeudE(i,1);
                else
                    fd.ajoute(i);

            majHauteur();
            reequilibrer();
        }
```



```
        public void majHauteur()
        {
            int hg, hd; // Hauteurs respectives
                        // des fils droits et gauches
            if(fg==null)
                hg = 0;
            else
                hg = ((NoeudE)fg).hauteur;

            if(fd==null)
                hd = 0;
            else
                hd = ((NoeudE)fd).hauteur;

            hauteur = Math.max(hd, hg)+1;
        }

        public int desequilibre()
        {
            if(estFeuille())
                return 0;

            int hg, hd; // Hauteurs respectives
                        // des fils droits et gauches
            if(fg==null)
                hg = 0;
            else
                hg = ((NoeudE)fg).hauteur;

            if(fd==null)
                hd = 0;
            else
                hd = ((NoeudE)fd).hauteur;

            return hg - hd;
        }

        public void rotationD()
        {
            NoeudE copieThis = new NoeudE(this);

            info = fg.info;
            hauteur = ((NoeudE)fg).hauteur;
            copieThis.fg = fg.fd;

            fg = fg.fg;
            fd = copieThis;

            majHauteur();
            ((NoeudE)fd).majHauteur();
        }

        public void rotationG()
        {
            NoeudE copieThis = new NoeudE(this);

            info = fd.info;
            hauteur = ((NoeudE)fd).hauteur;
            copieThis.fd = fd.fg;

            fd = fd.fd;
```

```
            fg = copieThis;

            majHauteur();
            ((NoeudE)fg).majHauteur();
        }
        public void rotationGD()
        {
            NoeudE fg = (NoeudE)this.fg;
            NoeudE fd = (NoeudE)this.fd;
            fg.rotationG();
            rotationD();
        }
        public void rotationDG()
        {
            NoeudE fg = (NoeudE)this.fg;
            NoeudE fd = (NoeudE)this.fd;
            fd.rotationD();
            rotationG();
        }


    }
    public ArbreE(Info info)
    {
        super(info);
        // Tout arbre de recherche a au moins un noeud!
        racine =  new NoeudE(info,1);
    }

    public void ajoute(Info i)
    {
        (racine).ajoute(i);
        nbElements++;
    }

}
```

```
import java.util.*;

public class Info {
    public Info(String s)
    {
        this.s = s;
    }
    private String s;
    public String toString()
    {
        return s;
    }
    public int compareTo(Info i)
    {
        return s.compareTo(i.s);
    }
}
```

```
import java.io.*;
import java.util.*;

public class Test {
    public void ajouterNoeuds(Arbre a)
    {
        a.ajoute(new Info("Aphrodite"));
        a.ajoute(new Info("Apollon"));
        a.ajoute(new Info("Ares"));
        a.ajoute(new Info("Arthémis"));
        a.ajoute(new Info("Athena"));
        a.ajoute(new Info("Chronos"));
        a.ajoute(new Info("Dyonisos"));
        a.ajoute(new Info("Ephaistos"));
        a.ajoute(new Info("Eros"));
        a.ajoute(new Info("Gaia"));
        a.ajoute(new Info("Hades"));
        a.ajoute(new Info("Hera"));
        a.ajoute(new Info("Herakles"));
        a.ajoute(new Info("Poseidon"));
        a.ajoute(new Info("Rhéa"));
        a.ajoute(new Info("Thanatos"));
    }
    public void testerRecherche(Arbre a, String s)
    {
        System.out.print("Recherche de "+s+"... ");
        if(a.recherche(new Info(s))≠null)
            System.out.println("Trouvé!");
        else
            System.out.println("Absent");

    }

    public void testerArbre(Arbre a)
    {
        ajouterNoeuds(a);
        System.out.println(a);
        System.out.println("Profondeur: "+a.profondeur());


        testerRecherche(a,"Hades");
        testerRecherche(a,"Zeus");
        testerRecherche(a,"Neptune");
        testerRecherche(a,"Apollon");
    }
    public Test()
    {
    }
    public static void main(String argv[])
    {
        Test test = new Test();

        System.out.println("Test de l'arbre de recherche");
        Arbre a = new Arbre(new Info("Zeus"));
        test.testerArbre(a);

        System.out.println("-------------------------");
        System.out.println("Test de l'arbre de recherche Version ordonnée");
        a = new ArbreE(new Info("Zeus"));
        test.testerArbre(a);
    }
}
```

```
allClasses = Arbre.class Test.class Info.class ArbreE.class

%.class: %.java
        javac $?

exe: $(allClasses)
        java Test

comp: $(allClasses)
```