Listes et arbres binaires

Des structures de données dynamiques

- Listes, Listes ordonnées
- Arbres binaires, arbre binaires de recherche

Listes chaînées

Utile si le nombre d'éléments n'est pas connu à l'avance et évolue beaucoup.

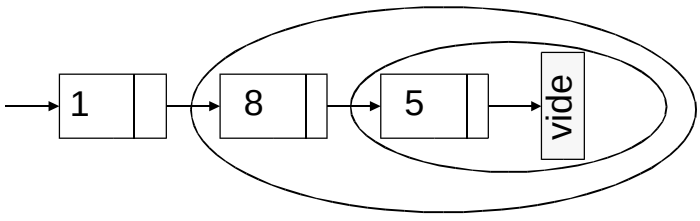
Permet de réaliser les piles, les files, les listes ordonnées, … les structures de données *dynamiques*.

API java : LinkedList

Définition inductive :

Base : liste vide

Règle : si l est une liste, x un élément, alors < x , l > est une liste



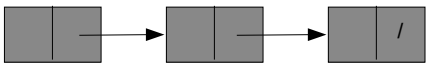
Que fait-on sur les listes ?

- Ajouter un élément : au début, à la fin, à un indice donné
- Chercher un élément : par sa valeur, par son indice
- Supprimer un élément : par sa valeur, par son indice
- Afficher la liste
- Opérations entre deux listes : fusionner deux listes, chercher les éléments communs à deux listes, ...

Comment représenter les listes en java ? *(pas de pointeurs)*

```
/** Une première classe (TROP) simple pour les listes  */
public class Chainon {
   public Object element; // la valeur de la cellule
   public Chainon  reste;   // la suite du chaînage

   /**  Construit un chainon à un élément, l'objet x,  le reste est la liste vide */
   public Chainon(Object x) {
      element = x;
      reste   = null;
   }
//   public void ajouterDebut(Object x) {
//         Chainon premier = new Chainon(x);
//         premier.reste = this;
//         this = premier;   // AIE AIE AIE
//    }
```



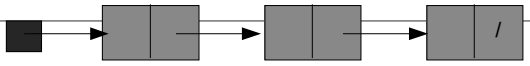
A la compilation :

```
Chainon.java:20: cannot assign a value to final variable this
this = premier; // AIE AIE AIE
       ^
1 error
```

```
  public void ajouterFin(Object x) {
    if (reste==null)
      reste = new Chainon(x);
    else reste.ajouterFin(x);
  }
  public String toString() {
    String result = element.toString();
    if (reste!=null) result += "  " + reste.toString();
    return result;
  }
  public static void main(String[] s) {
     Chainon c = new Chainon("il");
     c.ajouterFin("fait");
     c.ajouterFin("beau");
     System.out.println("la liste c : " + c);    // ----->   la liste c : il  fait  beau
  }
}
```

Problèmes :
- *C'est trop coûteux*
- *Si on veut classer les éléments par ordre croissant ou décroissant il faut pouvoir insérer au début*
- *Si on veut placer un élément à un indice précis il faut pouvoir insérer au début*

Les listes en java : Une solution



Classe Liste

- Contient la tête de liste (objet de type chaînon)
- S'occupe de traiter la liste vide et les ajouts/suppression en tête de liste

```
public class Liste {
    private Chainon tete;
    /** Construit une liste vide */
    public Liste() {
        tete = null;
    }
    /** Teste si la liste est vide. */
    public boolean vide() {
         return tete == null;
    }
    /**  Ajoute un élément en tête de liste */
    public void ajouterDebut(Object x) {
         Chainon c = new Chainon(x);
         c.reste = tete;
         tete = c;
    }
```

Toutes les méthodes non primitives de la classe **Liste** sont de la forme :

if (vide()) *TRAITER LE CAS DE LA LISTE VIDE*

else *APPEL DE LA METHODE SUR TETE*

```
// suite de la classe Liste
/**Retourne une chaîne de caractères représentant la liste */
    public String toString() {
        if (vide()) {
            return " vide ";
        } else return tete.toString();
    }
    /** renvoie la longueur de la liste */
    public int longueur() {
        if (vide()) return 0;
        return tete.longueur();
    }
```


Classe Chainon

- Contient les éléments de la liste, les éléments sont chainés entre eux
- Un chaînon contient toujours au moins un élément

  BASE : si x est un objet alors < x, null > est un chaînon

  REGLE : si x est un objet et c est un chaînon alors < x, c > est un chaînon
- Applique les fonctions sur les éléments dans la liste
- Toutes les méthodes non primitives de la classe **Chainon** sont de la forme :

  if (dernier()) *TRAITER LE CAS DU DERNIER ELEMENT*

  else *CALCUL ET APPEL RECURSIF SUR RESTE*

Classe Chainon

```
private class Chainon {
     private Object element; // la valeur de la cellule
     private Chainon  reste;   // la suite du chaînage

     /**  Construit un chainon à un élément, l'objet x
     *    le reste est la liste vide */
     public Chainon(Object x) {
         element = x;
         reste   = null;
     }
     /** vrai si this est le dernier chainon */
     public boolean dernier() {
         return reste==null;
     }
```



```
// classe Chainon suite ...
// méthodes non primitives

    public String toString() {
      if (dernier()) return element.toString();
      return element.toString() +  " -> " + reste.toString();
    }

    public int longueur() {
      if (dernier()) return 1;
      return 1 + reste.longueur();
    }
```

A VOUS : recherche d'un objet dans la liste

Dans la classe Liste

```
/** @return : true si o est dans la liste, false sinon
 * Complexité : ..... */
public boolean cherche(Object o) {
        if (vide()) return ..... ;
        return ..... ;
}
```

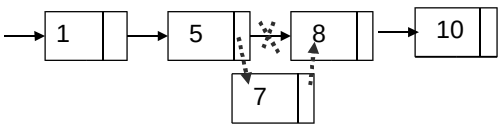
Dans la classe Chainon

```
/**  @return : true si o est dans la liste, false sinon
 * Complexité : ..... */
public boolean cherche(Object o) {
        if (dernier()) return ..... ;
        .....
        return ..... ;
}
```


Listes chaînées ordonnées

- Dans le cas où les éléments peuvent être comparés entre eux
  - les entiers, les chaînes de caractères, …
  - tout objet auquel on associe une clef entière (exemple: humains et n° de sécurité sociale, véhicule et n° de moteur, ...)
- Pour chaque chainon m (qui n'est pas le dernier) :

  m.element.clef <= m.element.reste.clef
- L'insertion se fait en insérant l'élément à son rang



ObjetAClef : une classe pour les objets comparables

```
/** Une classe pour des objets constitués d'une information (un
  Object) et d'une clef (un int).
 * Toutes les méthodes sont en O(1) */
public class ObjetAClef {
    private Object objet; // l'objet
    private int clef;    // la clef de l'objet
    /** Construit un ObjetAClef avec objet et clef. */
    public ObjetAClef(int c, Object o) {
        objet  = o;
        clef = c;
    }
    /** Retourne l'information contenue dans l'ObjetAClef. */
    public Object valeur() {
        return objet;
    }
    /**Retourne la clef de l'ObjetAClef.*/
    public int clef() {
        return clef;
    }
```



```
// classe ObjetAClef suite ...
// méthodes pour comparer les objets à clefs
    public boolean superieurEgal(ObjetAClef o){
             return clef >= o.clef;
    }
    public boolean superieur(ObjetAClef o){
             return clef > o.clef;
    }
    public boolean inferieur(ObjetAClef o){
             return clef < o.clef;
    }
    public boolean egal(ObjetAClef o){
             return clef == o.clef;
    }
    /** Retourne la représentation d'un ObjetAClef */
    public String toString() {
        if ( objet == null )
            return "[" + clef + "] ";
        return "[" + clef + "," + objet + "] ";
    }
}
```

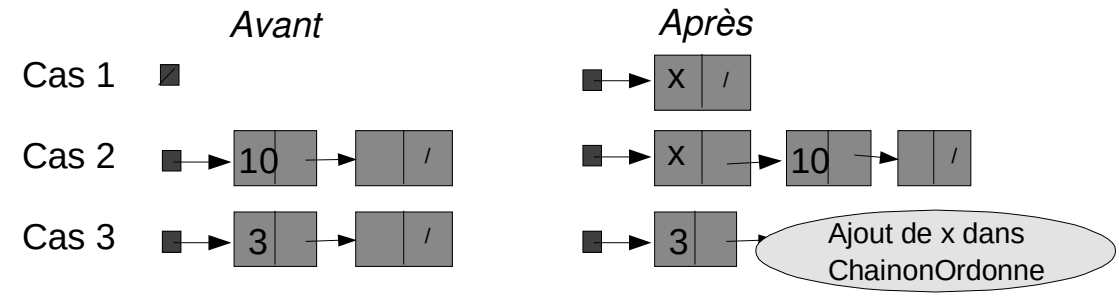
Classe ListeOrdonnee

```
public class ListeOrdonnee {
   private ChainonOrdonne tete; // le lien vers un ChainonOrdonne
   /**  Construit une liste vide. */
   public ListeOrdonnee() {
      tete = null;
   }
  /** Teste si la liste est vide */
   public boolean vide() {
      return tete == null;
   }
```



```
/**  Accroche un nouveau Maillon contenant l'objet x
    * antécédent : la liste est triée par ordre croissant,
    * conséquent : la liste contient x et est triée par ordre croissant    */
```

Ajout de l'objet X de clef 5




```
public void ajouter(ObjetAClef x) {
       if (vide()) tete = new ChainonOrdonne(x);      // Cas 1
       else if (tete.element.superieur(x))            // Cas 2
         tete = new ChainonOrdonne(x,tete);
       else tete.ajouter(x);                          // Cas 3
}
```

```
private class ChainonOrdonne {
   public ObjetAClef element; // la valeur de la cellule
   public ChainonOrdonne  reste;   // la suite du chaînage
  /**  Construit un chainon à un élément, l'objet x,  le reste est la liste vide */
   public ChainonOrdonne(ObjetAClef x) {
       element = x;
       reste   = null;
   }
   /** vrai si this est le dernier chainon */
   public boolean dernier() {
       return reste==null;
   }
   /** insère l'objet x à sa place */
   public void ajouter(ObjetAClef x) {
       if (dernier())
          reste = new ChainonOrdonne(x);
       else if (reste.element.superieur(x))
          reste = new ChainonOrdonne(x,reste);
       else reste.ajouter(x);
   }
```


A VOUS : recherche d'un élément de clef c

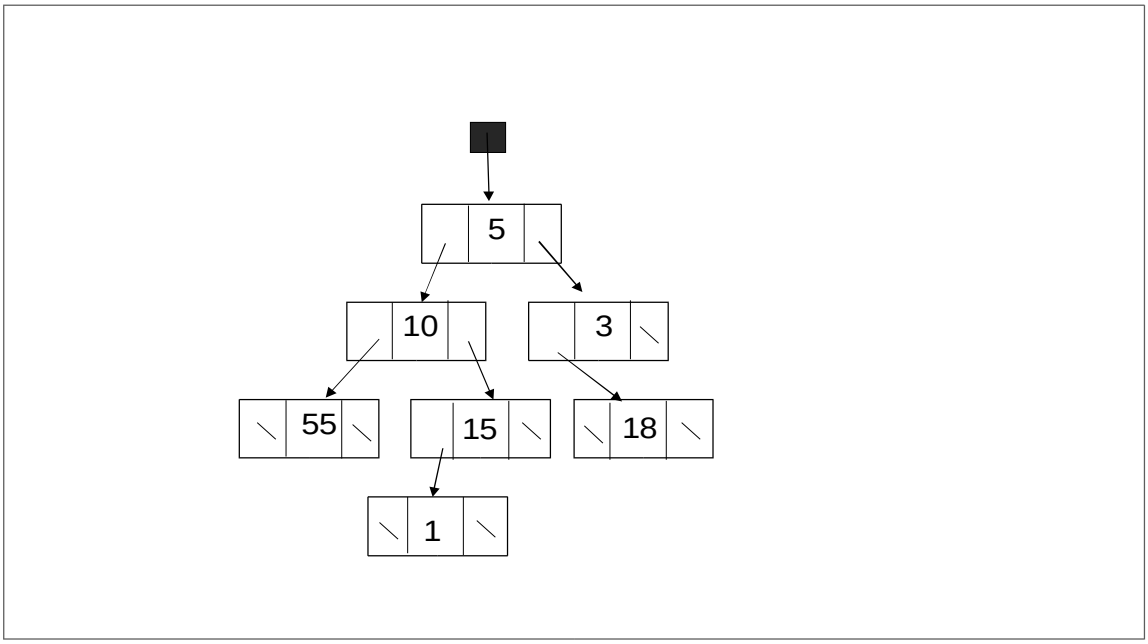
Dans la classe ListeOrdonnee

```
   /** @return : null s'il n'y a pas d'objet de clef c dans la liste, sinon le premier objet de clef c
    * Complexité : ..... */
   public ObjetAClef cherche(int c) {
            if (vide()) return ..... ;
            return ..... ;
   }
```

Dans la classe ChainonOrdonne

```
   /**  @return : null s'il n'y a pas d'objet de clef c dans la liste, sinon le premier objet de clef c
    * Complexité : ..... */
   public ObjetAClef cherche(int c) {
            if (dernier()) ..... ;
            .....
            .....
            return ..... ;
   }
```

Arbres binaires



Définition inductive :

Base : l'arbre vide est un arbre binaire

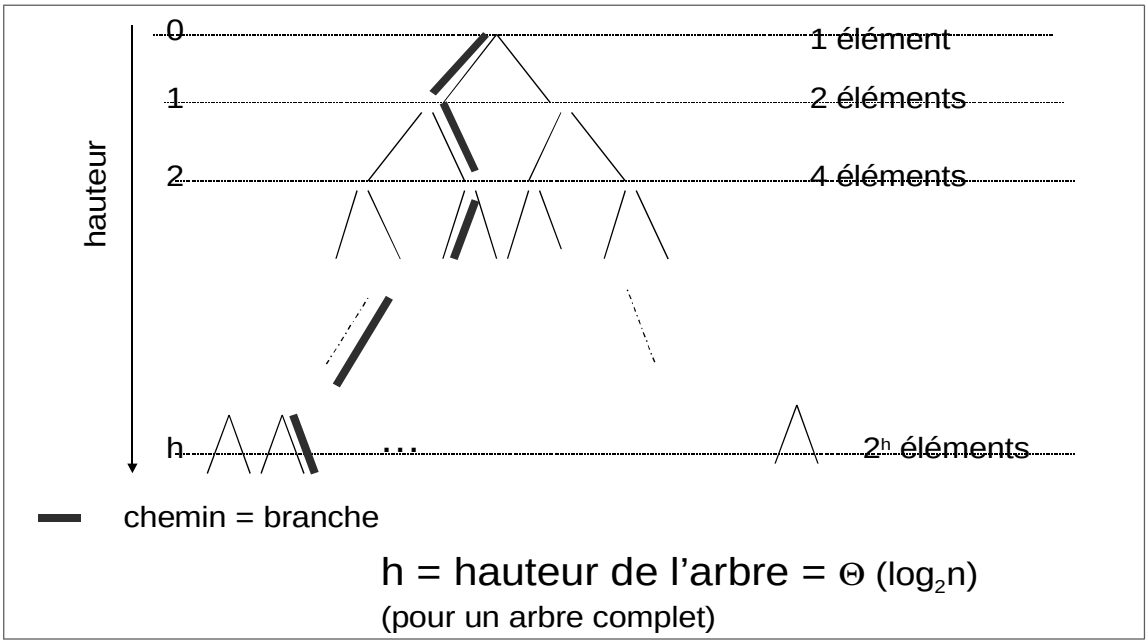
Règle : si g est un arbre binaire, si d est un arbre binaire, si x est un élément, alors

< x , g , d > est un arbre binaire

Utilisé pour des structures de données dynamiques

- arbres syntaxiques (expressions, XML, …)
- arbres binaires de recherche

Intérêt par rapport aux listes : pour n éléments, si une opération nécessite de parcourir une branche de l'arbre, cette opération est en $\Theta(\log_2 n)$



Arbre en java : similaire aux listes

- Classe Arbre

```
Public class Arbre {
  Private ChainonArbre racine;
```

  - Gère l'arbre vide
  - Fait appel aux méthodes de la classe ChainonArbre sur la racine

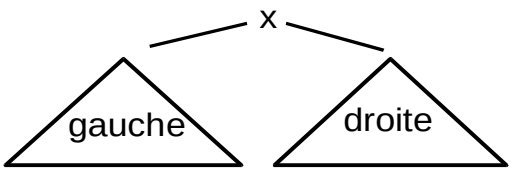
- Classe ChainonArbre interne

```
Public class ChainonArbre {
  Private Object element;
  Private ChainonArbre gauche;
  Private ChainonArbre droite;
```

  - BASE : si x est un objet alors < x, null, null> est un arbre
  - REGLE : si x est un objet, si g et d sont des arbres alors < x, g, d> est un arbre
    - Constructeur pour un arbre avec un élément et gauche = droite = null
    - Pimitives gaucheVide(), droiteVide() et dernier()

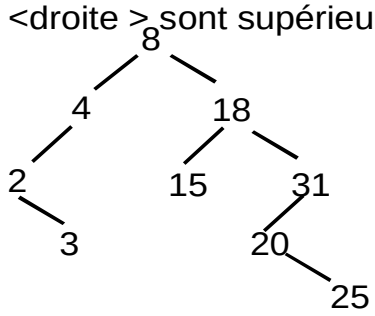
Arbres binaires de recherche

Pour stocker des éléments que l'on peut ordonner.



tous les éléments de <gauche> sont inférieurs à x

tous les éléments de <droite > sont supérieurs à x



Arbre de recherche en java

- Classe ArbreRecherche

```
Public class ArbreRecherche {
  Private ChainonArbreRecherche racine;
```

  - Gère l'arbre vide
  - Fait appel aux méthodes de la classe ChainonArbreRecherche sur la racine

- Classe ChainonArbreRecherche interne

```
Public class ChainonArbreRecherche {
  Private ObjetAClef element;
  Private ChainonArbreRecherche gauche;
  Private ChainonArbreRecherche droite;
```

  - BASE : si x est un objet à clef alors < x, null, null> est un arbre
  - REGLE : si x est un objet à clef, si g et d sont des arbres de recherche et que les éléments de g sont supérieurs à x et inférieurs aux éléments de d alors < x, g, d> est un arbre de recherche

Complexité des méthodes sur les arbres binaires de recherche :

- **recherche ou insertion d'un élément :**

  on rappelle récursivement à droite *ou* à gauche; si l'arbre est bien équilibré on a :

  $$C(n) = C(n/2) + \Theta(1)$$

  donc $C(n) = \Theta(\log_2(n))$

  Par contre, si l'arbre est une liste il a un seul fils et on a :

  $$C(n) = C(n-1) + \Theta(1)$$

  et donc $C(n) = \Theta(n)$

  Remarque : pour que la structure d'arbre soit intéressante il faut que l'arbre soit bien équilibré (la hauteur du sous-arbre gauche est presque égale à celle du sous-arbre droit).

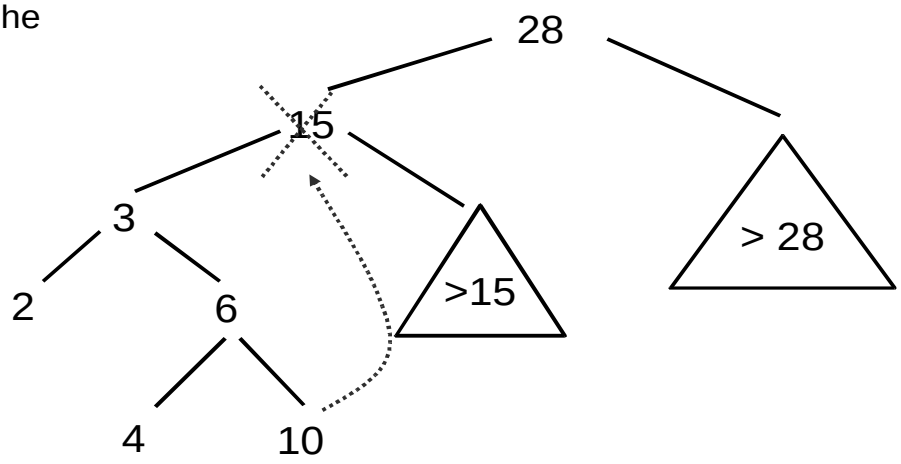
- **affichage des éléments :**

  on rappelle récursivement à droite *et* à gauche; si l'arbre est bien équilibré on a :

  $$C(n) = C(n/2) + C(n/2) + \Theta(1)$$

  donc $C(n) = \Theta(n)$

- Suppression d'un élément : on le remplace par le max du sous-arbre gauche

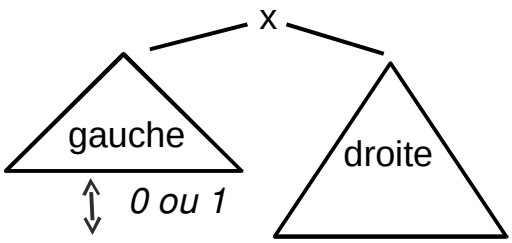


éléments de gauche < max(gauche) < éléments de droite

$C(n) = \Theta(\log_2(n))$

AVL

- Arbres binaires « bien équilibrés »

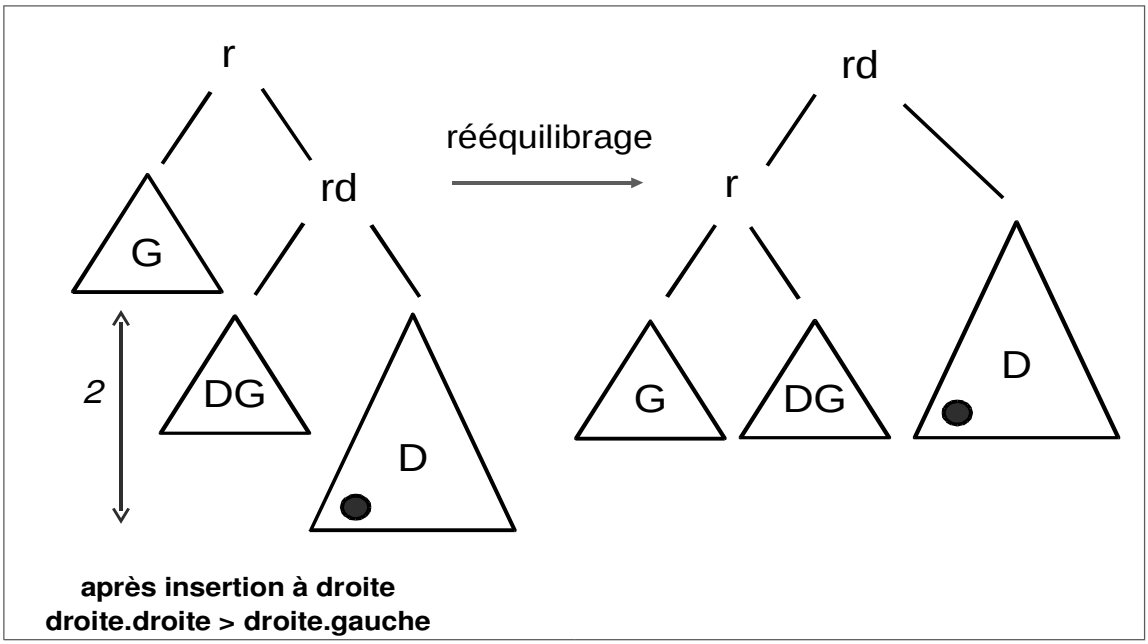


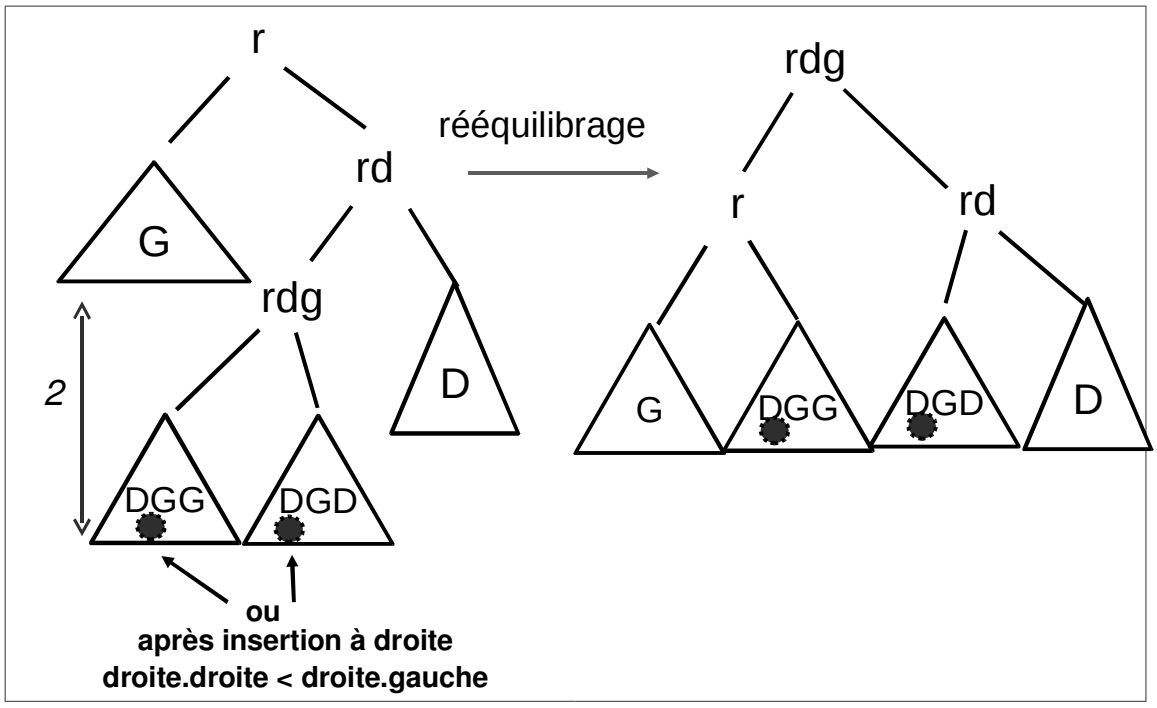
Les sous-arbres sont des AVL

La hauteur des deux sous-arbres diffère au plus de 1

Insertion et suppresion dans l'arbre suivies d'un ré-équilibrage si nécessaire (rotation des sous-arbres)

Les opérations sont toujours en $\Theta(\log_2 n)$


r
rd
G
DG
D
2
rééquilibrage
rd
r
G
DG
D
après insertion à droite
droite.droite > droite.gauche


r
rd
G
rdg
D
2
DGG
DGD
ou
après insertion à droite
droite.droite < droite.gauche
rééquilibrage
rdg
r
rd
G
DGG
DGD
D