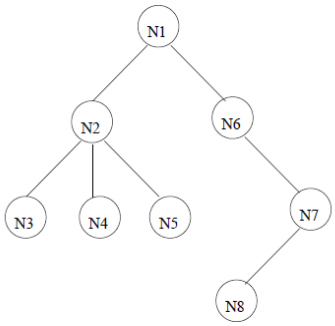
Chapitre VI : Structures hiérarchiques

- Les arbres.
- Les arbres binaires.
- les arbres binaires de recherche.
- les tas et files de priorité.

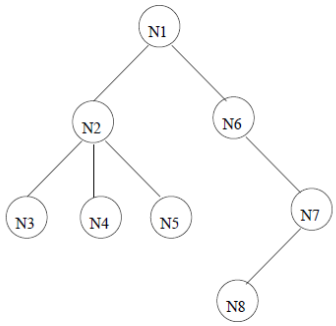
Les arbres

- Un arbre est une structure hiérarchique sur des nœuds à partir d'un nœud particulier, la racine.
- Un arbre peut être défini de manière récursive, c'est:
  - Soit un arbre atomique (ou *nœud terminal ou feuille)*
  - Soit un *nœud ayant pour fils d'autre arbres.*
- En général on étiquète les nœuds avec une valeur.
- Le nœud N1 est la ***racine*** *de l'arbre.*
- Les nœuds N3, N4, N5 et N8 sont des ***feuilles*** *de l'arbre.*
- Les nœuds N2, N6, N7 sont des ***nœuds internes*** *de l'arbre.*
- Le nœud N1 est le ***père*** *des nœuds* N2 et N6.
- N2 et N6 sont des ***nœuds fils*** du nœud N1.
- La racine est un ***ancêtre*** *de tous les* nœuds de l'arbre.
- Un **chemin** est une suite de nœuds consécutifs (**Exemple** : n1,n6,n7 : c'est un chemin)
- Une **branche** est un chemin de la racine à une feuille.



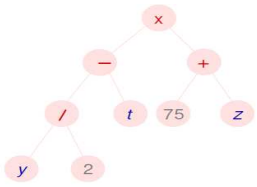
Les arbres

• La **Taille** d'un arbre : nombre de ses nœuds (Taille de notre arbre = 8)
• Le **degré** d'un nœud est le nombre de ses fils:
degré(n1) = 2, degré(n2) = 3, degré(n7)=1, degré(n8)=0
• Le **degré d'un arbre** est le plus grand degré de ses nœuds
Le degré de notre arbre = 3.
• Les nœuds d'un arbre se répartissent par **niveaux** :
Dans notre arbre :
Le niveau 0 contient la racine N1
Le niveau 1 contient N2,N6
• La **hauteur** d'un arbre est le nombre de niveaux de ses nœuds: la hauteur de notre arbre =4.
• la **profondeur** d'un nœud est égale à son niveau:
Profondeur(N2) =1, Profondeur(N1) = 0

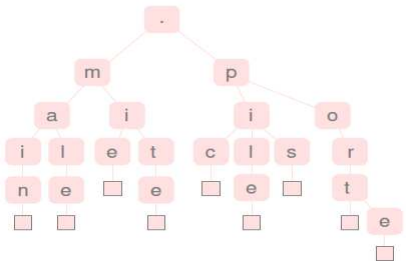


Les arbres

• **Quelques utilisations des arbres :**
• **Expressions arithmétiques.**

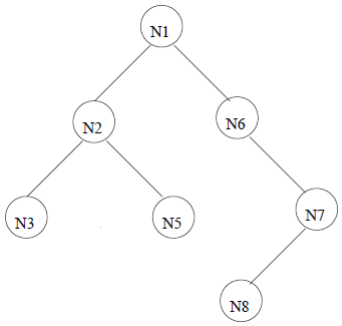


• **Arbre lexicographique.**



Les arbres binaires

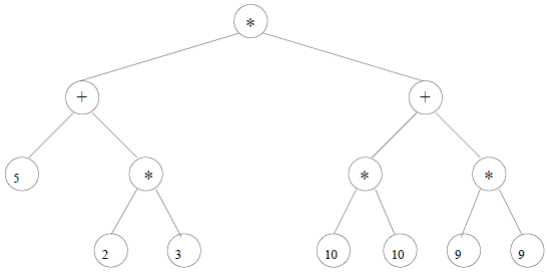
• Un arbre binaire est un arbre dont le degré est égale à 2
• Les algorithmes travaillant sur des arbres sont généralement récursifs.
• En langage algorithmique, on suppose disposer de fonctions d'accès: à partir de chaque nœud N on peut atteindre son père pere(N) et ses fils : filsDroit(N), filsGauche(N).




```
Algorithme parcours(ARBRE N)
   SI filsGauche(N) ≠ NULL ALORS parcours(filsGauche(N)) FINSI
   traiter(N);
   SI filsDroit(N) ≠ NULL ALORS parcours (filsDroit(N)) FINSI
FIN
```


Les arbres binaires

• **Exemple :** Arbre d'une expression arithmétique

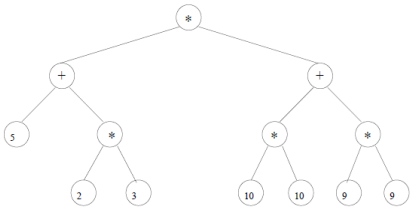


• Si la procédure traiter(N) affiche la valeur d'un nœud , on obtient:

$$5 + 2 * 3 * 10 * 10 + 9 * 9$$

• Est ce l'expression représentée par l'arbre?

Les arbres binaires




```
PRODECURE parcours( ARBRE N)
  SI (valeur(N)=='+') ALORS print('(') FINSI
  SI filsGauche(N) ≠NULL ALORS parcours(filsGauche(N)) FINSI
  PRINT(valeur(N));
  SI filsDroit(N) ≠ NULL ALORS parcours(filsDroit(N)) FINSI
  SI (valeur(N)=='+') ALORS print(')') FINSI
FIN
```

• En rajoutant des parenthèses autour d'une addition:

$$(5 + 2 * 3) * (10 * 10 + 9 * 9)$$

Les arbres binaires

• **Différents parcours d'un arbre binaire :** Suivant l'ordre dans lequel on écrit les instructions de la fonction de parcours, l'ordre de parcours des nœuds est différent.

• **Trois possibilités principales:**

• **Parcours préfixe :**

racine, sous-arbre gauche, sous-arbre droit .

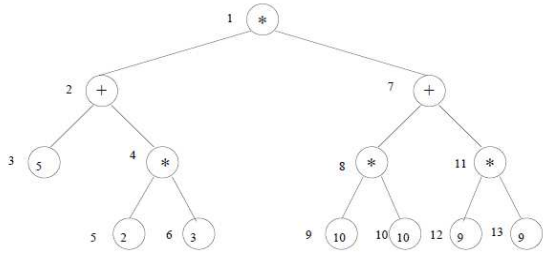
• **Parcours infixe :**

sous-arbre gauche, racine, sous-arbre droit .

• **Parcours postfixe :**

sous-arbre gauche, sous-arbre droit, racine.

Les arbres binaires



PRODECURE parcoursPrefixe (ARBRE N)

traiter(N);

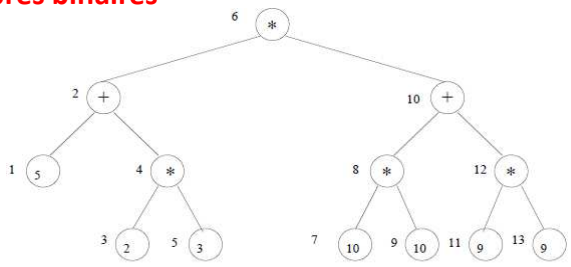
SI filsGauche(N) ≠ NULL ALORS parcours(filsGauche(N)) FINSI

SI filsDroit(N) ≠NULL ALORS parcours (filsDroit(N)) FINSI

FIN

- *** (+ 5 * 2 3) (+ * 10 10 * 9 9)**

Les arbres binaires



PRODECURE parcoursInfixe(ARBRE N)

SI filsGauche(N) ≠ NULL ALORS parcours(filsGauche(N)) FINSI

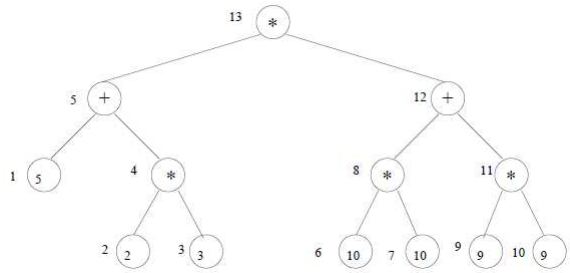
traiter(N);

SI filsDroit(N) ≠ NULL ALORS parcours(filsDroit(N)) FINSI

FIN

- **(5 + 2 * 3) * (10 * 10 + 9 * 9)**

Les arbres binaires

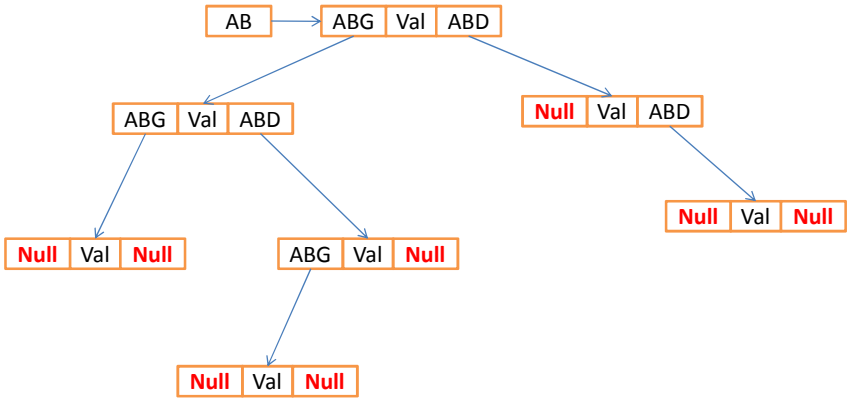



```
PRODECURE parcoursPostfixe( ARBRE N)
SI filsGauche(N) ≠ NULL ALORS parcours(filsGauche(N)) FINSI
SI filsDroit(N) ≠ NULL ALORS parcours(filsDroit(N)) FINSI
traiter(N);
FIN
```

- (5 2 3 * +) (10 10 * 9 9 * +) *

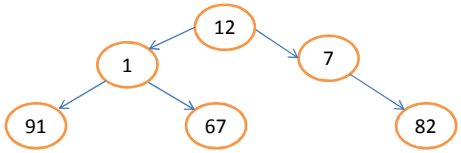
Les arbres binaires

•La représentation des arbres en utilisant les listes chainées.

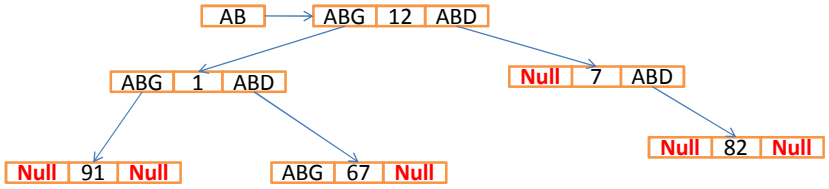


Les arbres binaires

•Exemple : représentation d'un arbre



• Sera représenté en mémoire par :

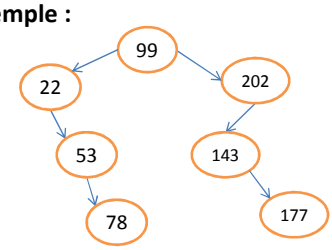


Les arbres binaires de recherche

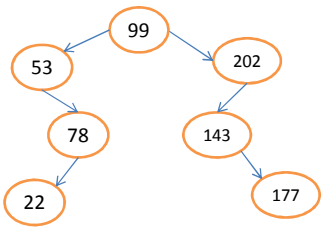
• Un Arbre Binaire de Recherche (ABR) est un arbre binaire étiqueté par des éléments d'un ensemble A totalement ordonne et tel que si ***a*** est l'étiquette d'un nœud *n* :

- Tous les nœuds du sous-arbre gauche de *n* sont étiquetés par des éléments *b* tels que $b < a$, ET
- Tous les nœuds du sous-arbre droit de *n* sont étiquetés par des éléments *b* tels que $b > a$.

•Exemple :



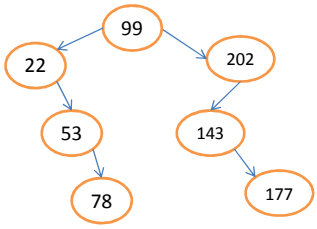
Un Arbre binaire de recherche



Un Arbre binaire

Les arbres binaires de recherche

- Le parcours infixé d'un arbre binaire de recherche donne la suite des étiquettes en ordre croissant.
- **Le parcours infixé de l'arbre suivant :**



- **Donne : (22 53 78 99 143 177 202)**

Les arbres binaires de recherche en java

- nous manipulons cela en utilisant la classe suivante :

```
class ArbreBinaireRecherche<Type extends Comparable<Type>>
 {
 class Noeud<Type> {…}  /*Classe interne pour créer les noeuds  */
 private Noeud<Type> root;    /* La racine de l'arbre */

 public ArbreBinaireRecherche() {…}    /*Constructeur de l'arbre*/
 public boolean vide() {…}             /*Teste si l'arbre est vide*/
 public boolean trouver(Type x) {…}  /*Trouver un élément x */
 public Type min() throws Exception {…} /*Trouver l'élément minimum*/
 public Type max() throws Exception {…} /*Trouver l'élément Maximum*/
 public void ajouter( Type x ) {…} /* Insérer un élément*/
 public void supprimer( Type x ) {…} /*Supprimer un élément*/
 public void afficherArbre() {…} /*Afficher l'arbre*/

 private boolean trouver(Type x, Noeud<Type> t) {…} /*Appelée par méthodes public*/
 private Noeud<Type> min(Noeud<Type> t) {…}   /*Appelée par méthodes public*/
 private Noeud<Type> max(Noeud<Type> t) {…} /*Appelée par méthodes public*/

 private Noeud<Type> ajouter( Type x, Noeud<Type> t ){…} /*Appelée par méthodes public*/
 private Noeud<Type> supprimer( Type x, Noeud<Type> t ){…} /*Appelée par méthodes public*/
 private void afficherArbre( Noeud<Type> t ) {…} /*Appelée par méthodes public*/
 }
```

Les arbres binaires de recherche en java

- La classe Interne Nœud : permettant la création des nœuds

```
class Noeud<Type> {
Noeud( Type elem )
 { this( elem, null, null ); }

 Noeud( Type elem, Noeud<Type> gche, Noeud<Type> rte )
 {
     element = elem; gauche = gche; droite = rte;
 }

  Type element;
  Noeud<Type> gauche;
  Noeud<Type> droite;
}
```


Les arbres binaires de recherche en java

- **Le constructeur de l'arbre :**

```
public ArbreBinaireRecherche() {
root = null;
}
```

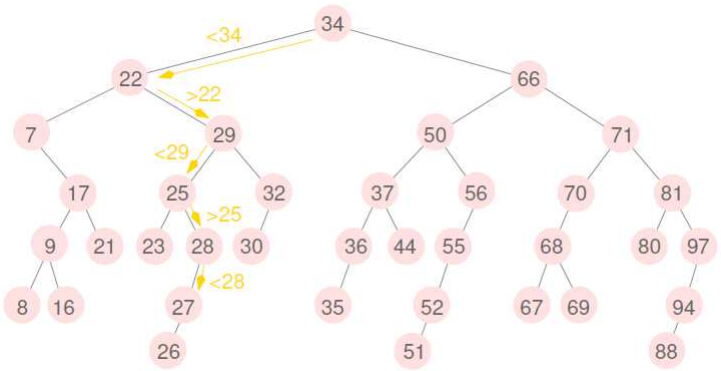
- **tester si l'arbre est vide :**

```
public boolean vide()
{
   return root == null;
}
```

Les arbres binaires de recherche en java

La recherche d'un élément: La recherche d'une valeur dans un ABR consiste à parcourir une branche en partant de la racine, en descendant chaque fois sur le fils gauche ou sur le fils droit suivant que la valeur portée par le nœud est plus grande ou plus petite que la valeur cherchée. La recherche s'arrête dés que la valeur est rencontrée ou que l'on a atteint l'extrémité d'une branche (le fils sur lequel il aurait fallu descendre n'existe pas).

Exemple : recherche de la valeur 27



Les arbres binaires de recherche en java

- **La recherche d'un élément:**

```
public boolean trouver(Type x)
 {
    return trouver(x,root);
 }
```

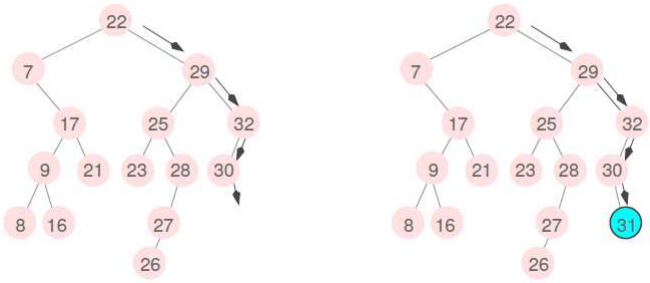
- **La méthode trouver (x,root) :**

```
 private boolean trouver(Type x, Noeud<Type> t)
  { if( t == null )
            return false;
   int compareResult = x.compareTo( t.element );

   if( compareResult < 0 )
           return trouver( x, t.gauche );
   else if( compareResult > 0 )
           return trouver( x, t.droite );
   else
        return true;
  }
```

Les arbres binaires de recherche en java

L'insertion d'un élément: Le principe est le même que pour la recherche. Un nouveau nœud est créé avec la nouvelle valeur et inséré à l'endroit où la recherche s'est arrêtée.

Exemple : Insertion de l'élément 31



Les arbres binaires de recherche en java

- **L'insertion d'un élément:**

```
public void ajouter( Type x )
 {
    root = ajouter(x,root);
 }
```

- **La méthode ajouter (x,root) :**

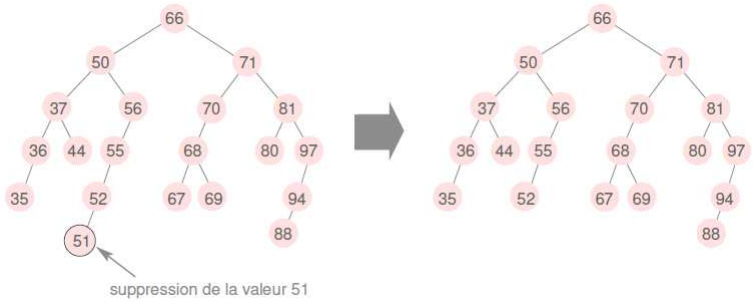
```
private Noeud<Type> ajouter( Type x, Noeud<Type> t )
 {
    if( t == null )
          return new Noeud<>( x, null, null );

    int compareResult = x.compareTo( t.element );

    if(compareResult < 0)
          t.gauche = ajouter(x, t.gauche);
    else if( compareResult > 0 )
         t.droite = ajouter( x, t.droite );

return t;
 }
```

Les arbres binaires de recherche en java

La suppression d'un nœud : L'opération dépend du nombre de fils du nœud à supprimer. On distingue trois cas :

Cas 1 : le nœud à supprimer n'a pas de fils, c'est une feuille. Il suffit de décrocher le nœud de l'arbre, c'est-`a-dire de l'enlever en modifiant le lien du père, si il existe, vers ce fils. Si le père n'existe pas l'arbre devient l'arbre vide.

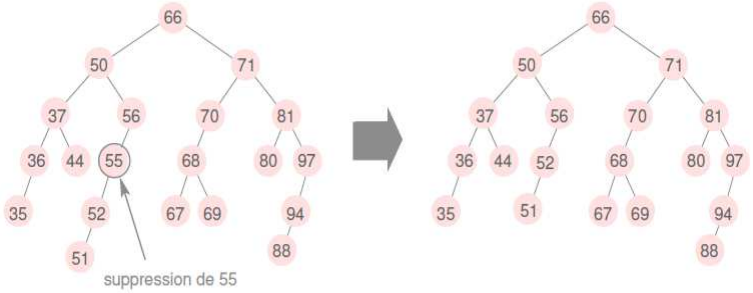
Exemple : suppression de nœud 51



Les arbres binaires de recherche en java

Cas 2 : le nœud à supprimer a un fils et un seul. Le nœud est décroché de l'arbre comme dans le cas 1. Il est remplacé par son fils unique dans le nœud père, si ce père existe. Sinon l'arbre est réduit au fils unique du nœud supprimé.

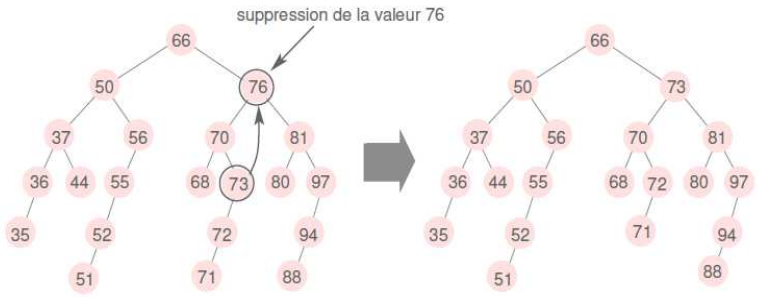
Exemple : Suppression de nœud 55



Les arbres binaires de recherche en java

Cas 3 : le nœud à supprimer *p* a deux fils. Soit q le nœud de son sous arbre gauche qui a la valeur la plus grande (on peut prendre indifféremment le nœud de son sous-arbre droit de valeur la plus petite). Il suffit de recopier la valeur de *q* dans le nœud *p* et de décrocher le nœud *q*. Puisque le nœud *q* a la valeur la plus grande dans le fils gauche, il n'a donc pas de fils droit, et peut être décroché comme on l'a fait dans les cas 1 et 2.

Exemple : Suppression de nœud 76



Les arbres binaires de recherche en java

- **La suppression d'un nœud :**

```
public void supprimer( Type x )
 {
   root = supprimer(x, root); }
```

- **La méthode** supprimer(x, root)**:**

```
private Noeud<Type> supprimer( Type x, Noeud<Type> t )
 {
    if( t == null )  return t;
 int compareResult = x.compareTo( t.element );
 if( compareResult < 0 )
         t.gauche = supprimer( x, t.gauche );
 else if( compareResult > 0 )
         t.droite = supprimer( x, t.droite );
 else if( t.gauche != null && t.droite != null )
 {
     t.element = min( t.droite ).element;
     t.droite = supprimer( t.element, t.droite );
 }
 else
     t = ( t.gauche != null ) ? t.gauche : t.droite;
 return t;
 }
```

Les arbres binaires de recherche en java

- **L'affichage d'un arbre :**

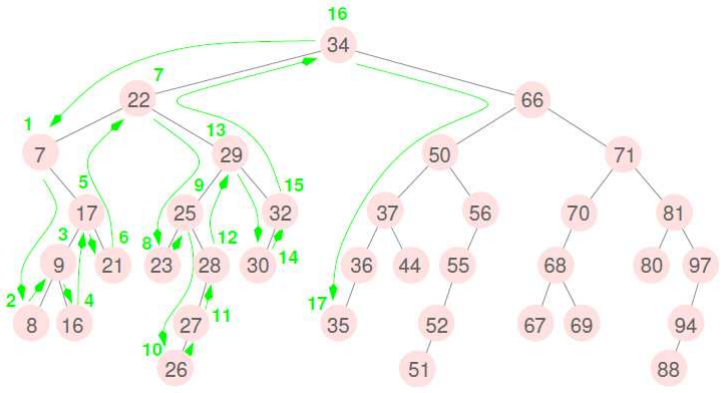
```
public void afficherArbre()
{
    afficherArbre(root);
}
```

•**La méthode afficheArbre(root):** cette méthode est basée sur le parcours infixé

```
private void afficherArbre( Noeud<Type> t )
{
  if( t != null )
    {
      afficherArbre( t.gauche );
      System.out.println( t.element );
      afficherArbre( t.droite );
    }
}
```


Les arbres binaires de recherche en java

- **Exemple :** Affichage de l'arbre suivant



•La méthode affiche :

7 8 9 16 17 21 22 23 25 26 27 28 29 30 32 34 35 36 37 ...

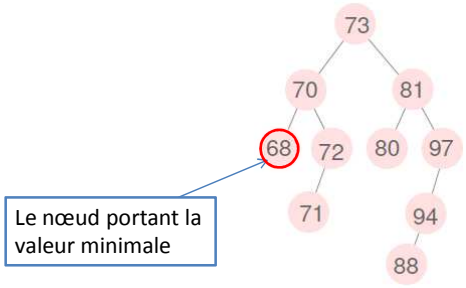
Les arbres binaires de recherche en java

La valeur minimale d'un arbre: Le nœud de valeur minimale est celui qui est "le plus à gauche" = dernier fils gauche du fils gauche du .... du fils gauche de la racine

```
public Type min() throws Exception
 {
    if (vide()) throw new Exception();
    return min(root).element;
 }
```

- **La méthode min(root):**

```
private Noeud<Type> min(Noeud<Type> t)
{
   if( t == null )
         return null;
   else if( t.gauche == null )
         return t;
   return min( t.gauche );
}
```



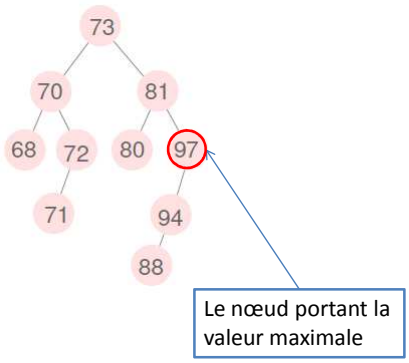
Les arbres binaires de recherche en java

La valeur maximale d'un arbre: Le nœud de valeur maximale est celui qui est "le plus à droit" = dernier fils droit du fils droit du .... du fils droit de la racine

```
public Type max() throws Exception
 {
    if(vide()) throw new Exception();
    return max(root).element;
 }
```

- **La méthode max(root):**

```
private Noeud<Type> max(Noeud<Type> t)
{
   if( t != null )
     while( t.droite != null )
          t = t.droite;
   return t;
}
```



Les tas et files de priorité

• Une **file de priorité** est un type abstrait de données opérant sur un ensemble ordonné, et muni des opérations suivantes :

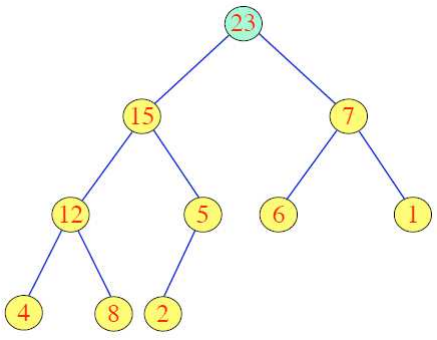
- trouver le **minimum**,
- **insérer** un élément,
- **retirer** le minimum.

• En inversant l'ordre, on obtient un type abstrait permettant de trouver le **maximum** et de le **retirer**.

• Une file de priorité peut être représentée par un **Tas.**

Les tas et files de priorité

• Un **tas** est un **arbre binaire tassé** tel que le contenu de chaque nœud soit supérieur ou égal (ou bien inférieur ou égal selon le cas) à celui de ses fils.

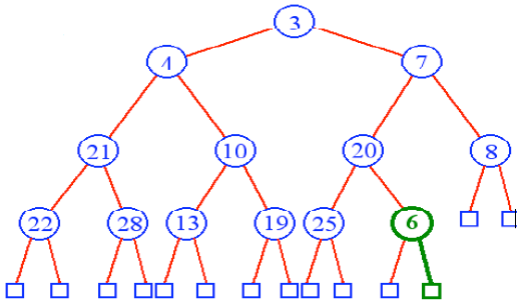


Les tas et files de priorité

- **Insertion d'un nœud :**
- Ajoutez le nœud à la prochaine position disponible dans le tas:
  - Si le dernier niveau est plein, ajouter l'élément comme fils gauche de l'élément le plus à gauche du dernier niveau.
  - Sinon ajouter l'élément après l'élément le plus à droite du dernier niveau.
- Réarranger le tas pour garder ses propriétés, en échangeant les nœuds père-fils non ordonnés.

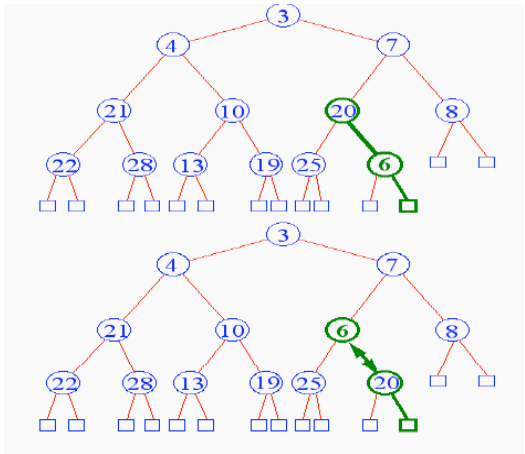
Exemple : Insertion de la valeur 6

Etape 1 :



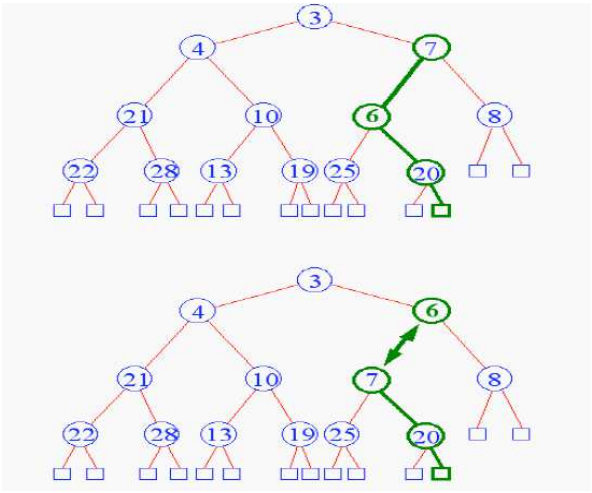
Les tas et files de priorité

- **Etape 2 : Echanger père-fils**



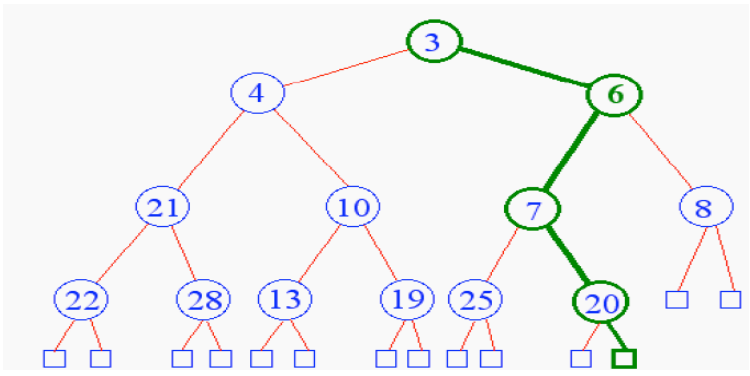
Les tas et files de priorité

• **Etape 2(suite) : Echanger père-fils**



Les tas et files de priorité

• La permutation se termine quand la valeur du nouveau nœud inséré est plus grande que la valeur de son parent ou quand la racine de tas est atteint



Les tas et files de priorité

• Suppression :

On supprime la racine

• La suppression de la clé racine laisse un trou

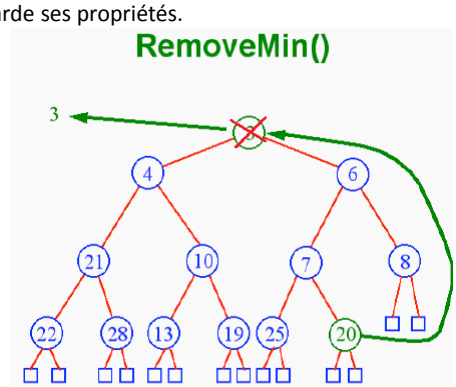
• Nous devons réparer le tas

• Remplacer le trou par le dernier nœud du tas (l'élément du dernier niveau le plus à droite)

• Ensuite, réarranger le tas pour qu'il garde ses propriétés.

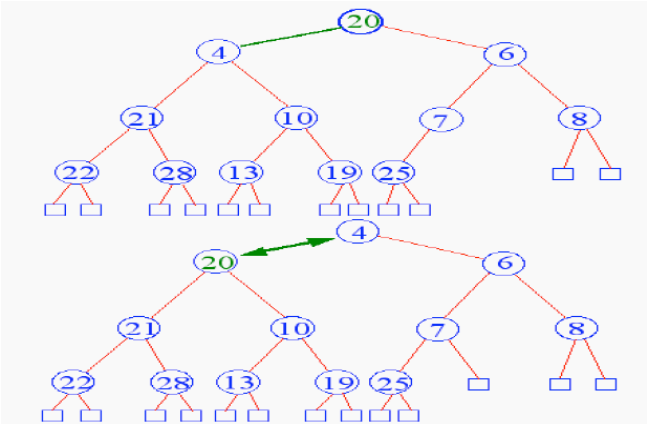
Exemple : suppression

Etape 1 : Suppression de la racine et réparation du trou



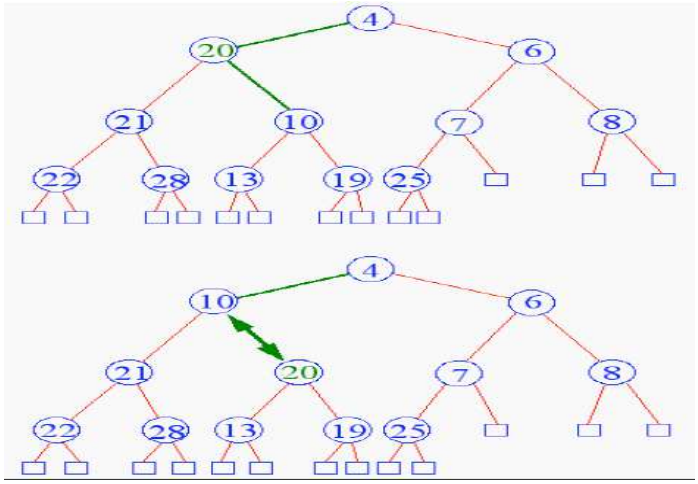
Les tas et files de priorité

Etape 2: réarrangement du tas, on compare le parent avec son fils le plus petit. Si cet enfant est plus petit que le parent, alors on les échange



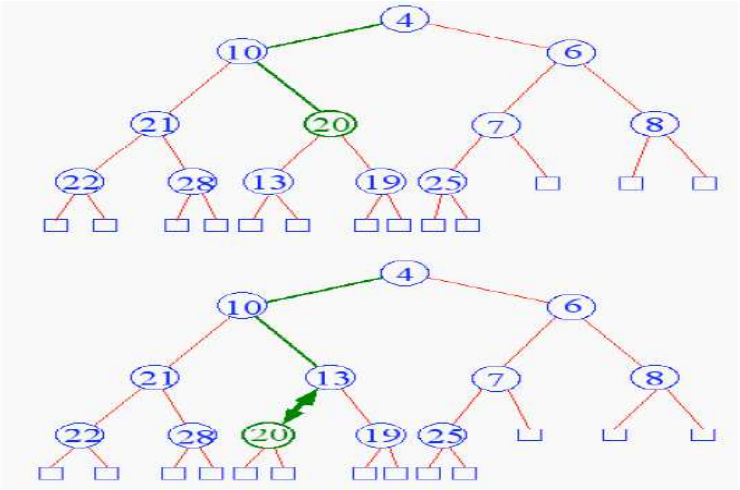
Les tas et files de priorité

Etape 2(suite) :



Les tas et files de priorité

Etape 2(suite) :



Les tas et files de priorité

• On se termine quand la valeur du nœud est plus petite que les valeurs de ses deux enfants ou quand on atteint le dernier niveau.