

Arbres I

INF2010

Structures de données et
algorithmes

Arbres I

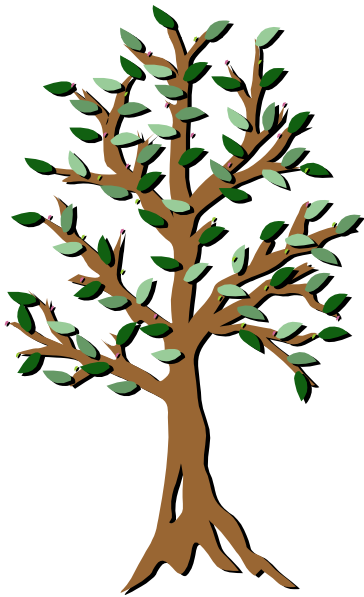
- Définition et généralités sur les arbres
- Arbres binaires
- Méthodes de parcours
- Arbres binaires de recherches

Arbres I

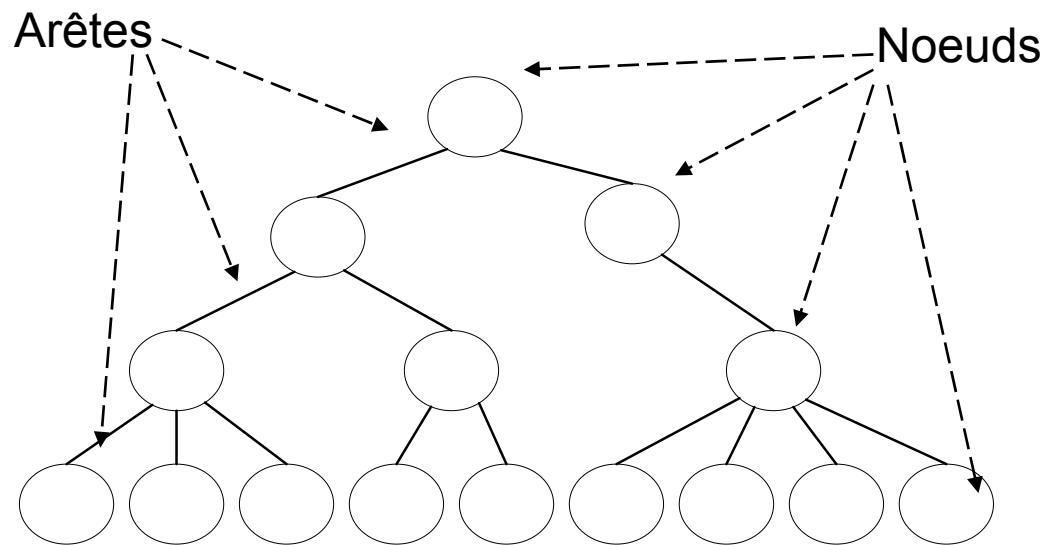
- Définition et généralités sur les arbres
- Arbres binaires
- Méthodes de parcours
- Arbres binaires de recherches

Arbres - Définition non récursive

Un arbre est une structure composée de nœuds et d'arêtes pour laquelle il n'existe qu'un seul chemin pour passer d'un nœud à un autre.



Arbre réel !

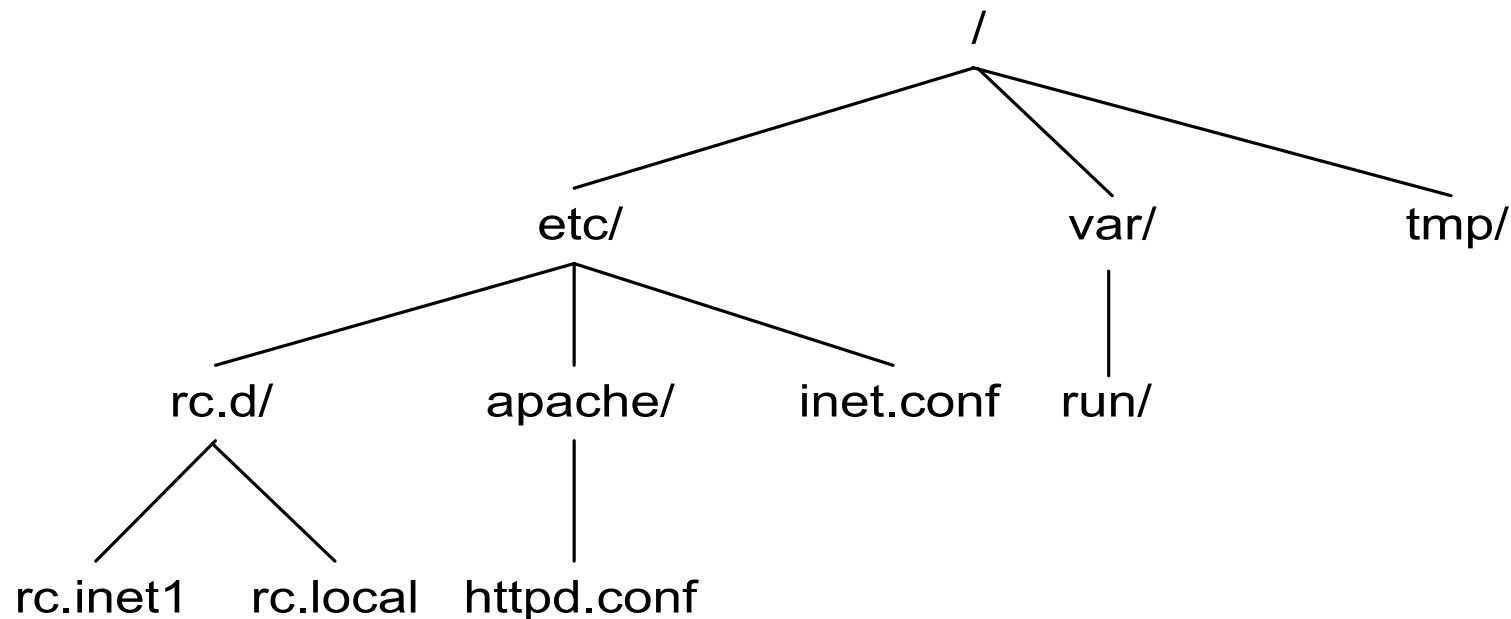


Arbre logiciel !

Utilité des arbres

Les arbres sont utiles dans deux principaux cas:

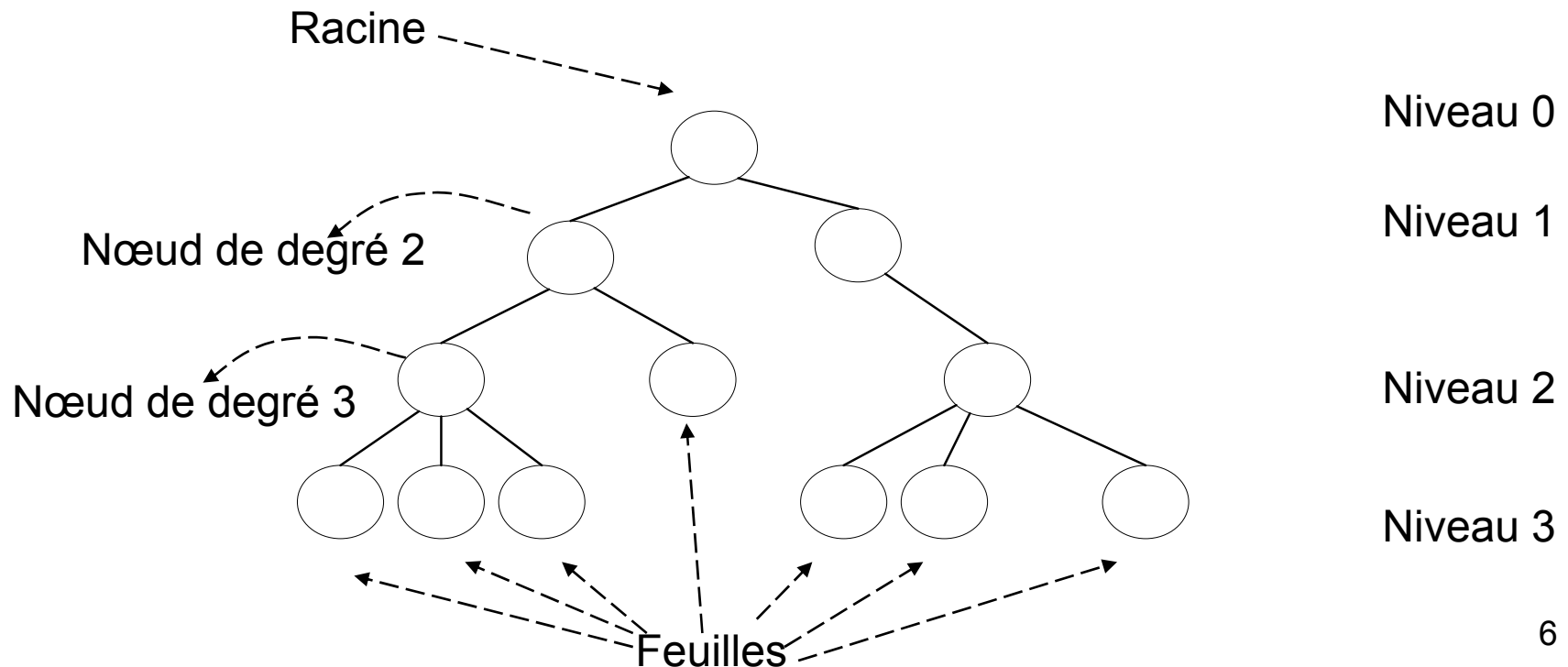
- 1) Recherche de solutions
- 2) Structure de données



Exemple de structure de données - arborescence de fichiers sous Linux

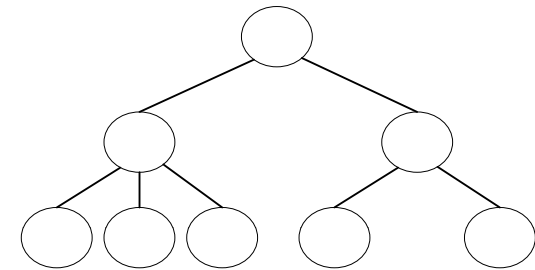
Définition

- **Racine:** Premier nœud d'un arbre (sans père)
- **Feuille:** Nœud sans fils (à la fin d'une branche).
- **Degré d'un nœud:** nombre de fils qu'il possède.
- **Degré d'un arbre:** degré maximal parmi tous les nœuds.
- **Niveau:** Nombre d'arrêtes parcourues depuis la racine pour arriver à un nœud



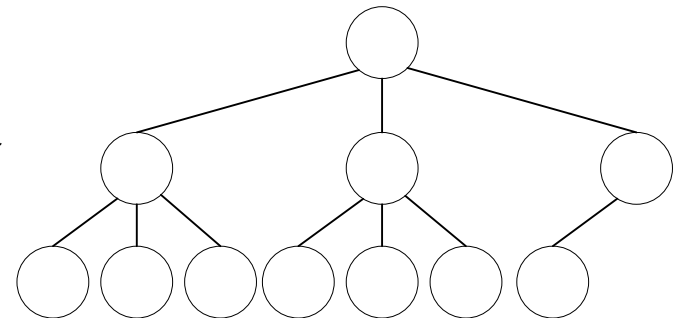
Définition (suite)

- **Arbre complet:** Un arbre de degré n est dit **complet** lorsque tous ses niveaux, à l'exception du dernier, possède un nombre maximal de nœuds. Le dernier niveau, quant à lui, est partiellement rempli de gauche à droite, sans trou.



Arbre de degré 3

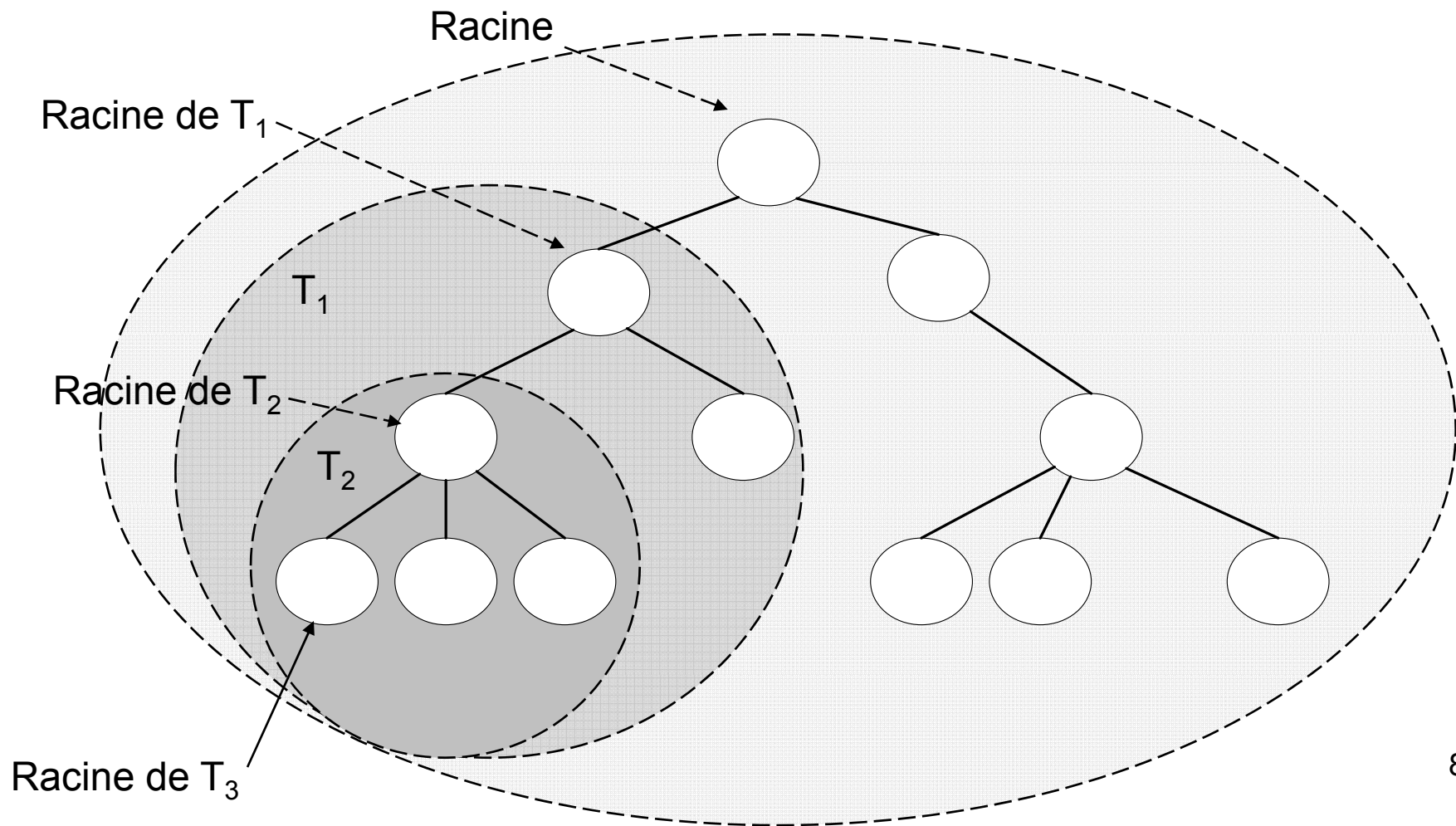
- Ainsi, un arbre complet de degré n doit posséder n^i nœuds au niveau i , à l'exception du dernier niveau.



Arbre de degré 3 complet

Définition récursive

- Un arbre est soit vide ou possède une racine à laquelle est rattaché zéro ou plusieurs sous-arbres non vides.



Arbres I

- Définition et généralités sur les arbres
- Arbres binaires
- Méthodes de parcours
- Arbres binaires de recherches

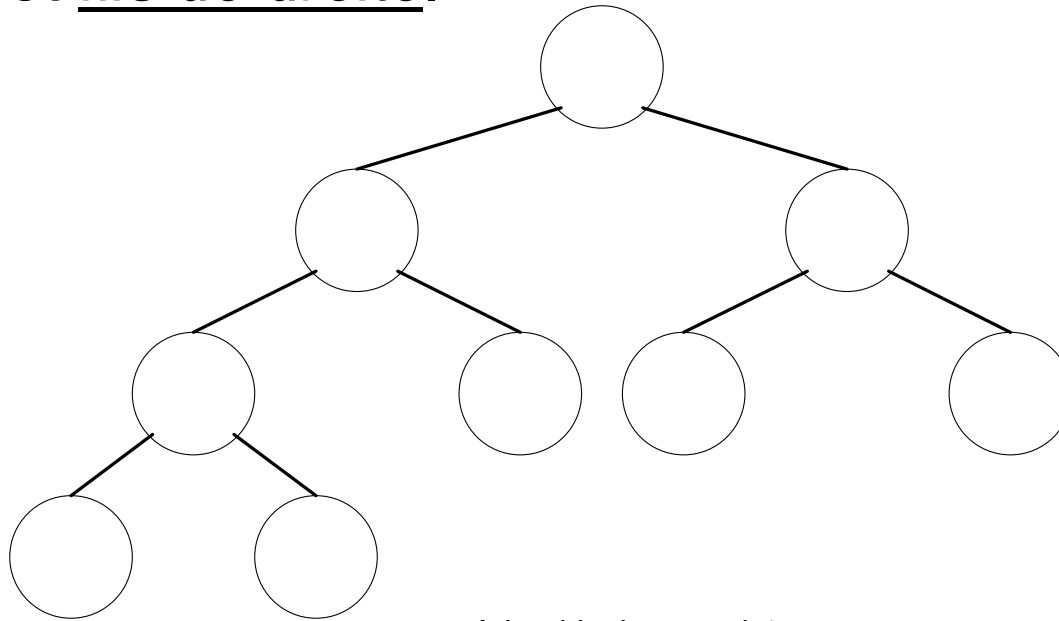
Arbres binaires - Plan

- Définition et généralités sur les arbres
- Arbres binaires
 - Définition
 - Représentation en mémoire
 - Par pointeurs
 - Par tableau séquentiel
 - Arbres arithmétiques
- Méthodes de parcours
- Arbres binaires de recherches

Arbres binaires - Définition

Un arbre binaire est un arbre de degré 2, c'est-à-dire que chaque nœuds a **au plus** deux fils.

On désigne chacun des fils par les appellations *fils de gauche* et *fils de droite*.



Arbre binaire complet

Représentation par pointeurs

- Puisque chaque nœud possède au maximum deux nœuds fils, on maintient une référence sur chacun d'eux.
- Avantages:
 - La taille de l'arbre est dynamique.
 - Facile d'opérer sur des références.
- Inconvénients:
 - On ne peut parcourir l'arbre que de la racine vers les feuilles.

Implémentation - Noeud

```
private static class BinaryNode<AnyType>
{
    // Constructors
    BinaryNode( AnyType theElement )
    {
        this( theElement, null, null );
    }

    BinaryNode(AnyType theElement,
               BinaryNode<AnyType> lt, BinaryNode<AnyType> rt )
    {
        element = theElement;
        left = lt;
        right = rt;
    }

    AnyType element;           // The data in the node
    BinaryNode<AnyType> left;   // Left child
    BinaryNode<AnyType> right;  // Right child
}
```

Implémentation – Arbre binaire

```
public class BinarySearchTree<AnyType extends Comparable<? super AnyType>>
{
    private static class BinaryNode<AnyType>
    { /* voir précédent */ }

    private BinaryNode<AnyType> root; ←

    public BinarySearchTree( )
    { root = null; }

    public void makeEmpty( )
    { root = null; }

    public boolean isEmpty( )
    { return root == null; }

    public void insert( AnyType x )
    { root = insert( x, root ); }

    private BinaryNode<AnyType> insert( AnyType x, BinaryNode<AnyType> t )
    { /* voir plus loin */ }

    public void remove( AnyType x )
    { root = remove( x, root ); }

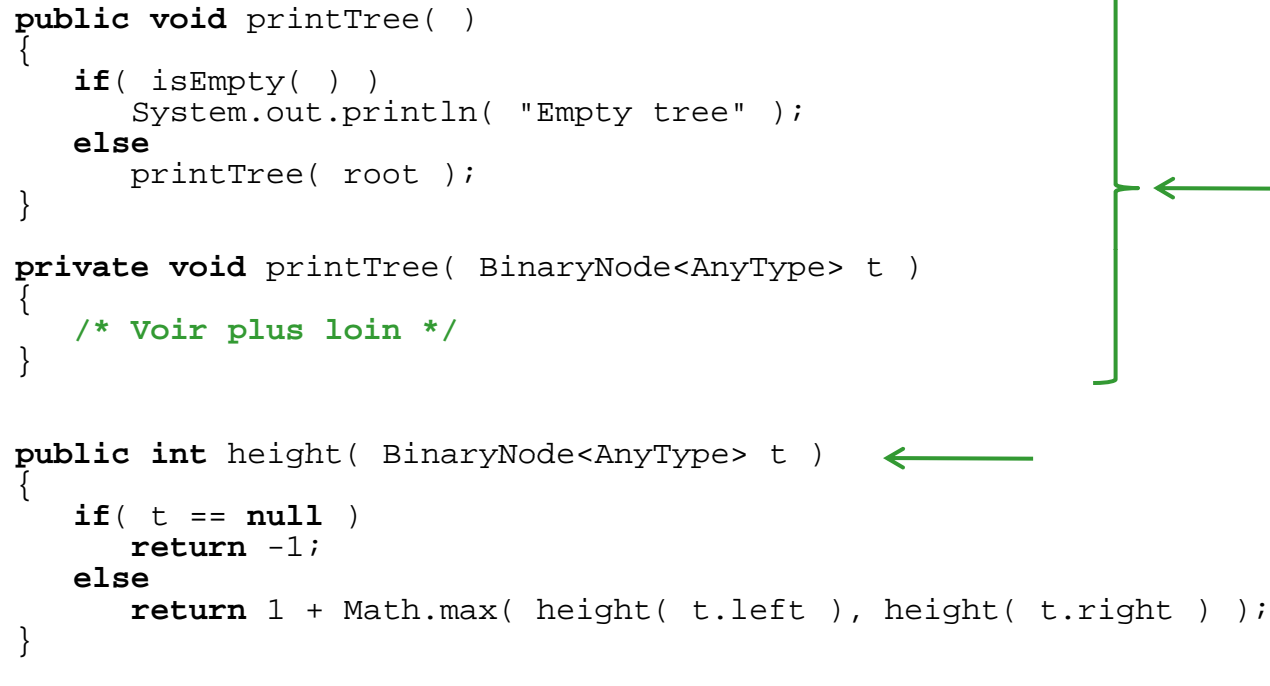
    private BinaryNode<AnyType> remove( AnyType x, BinaryNode<AnyType> t )
    { /* voir plus loin */ }

    public boolean contains( AnyType x )
    { return contains( x, root ); }

    private boolean contains( AnyType x, BinaryNode<AnyType> t )
    { /* voir plus loin */ }
```

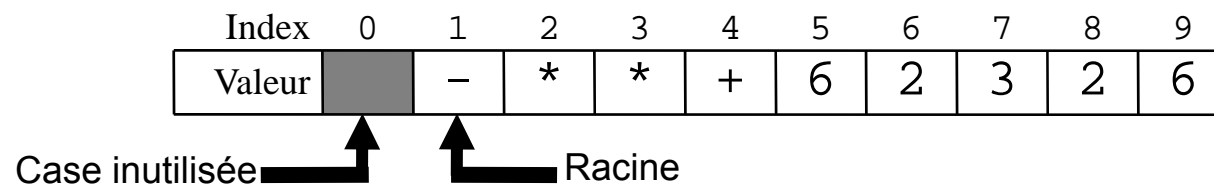
Implémentation – Arbre (suite)

```
public void printTree( )  
{  
    if( isEmpty( ) )  
        System.out.println( "Empty tree" );  
    else  
        printTree( root );  
}  
  
private void printTree( BinaryNode<AnyType> t )  
{  
    /* Voir plus loin */  
}  
  
public int height( BinaryNode<AnyType> t )  
{  
    if( t == null )  
        return -1;  
    else  
        return 1 + Math.max( height( t.left ), height( t.right ) );  
}
```



Représentation par tableau

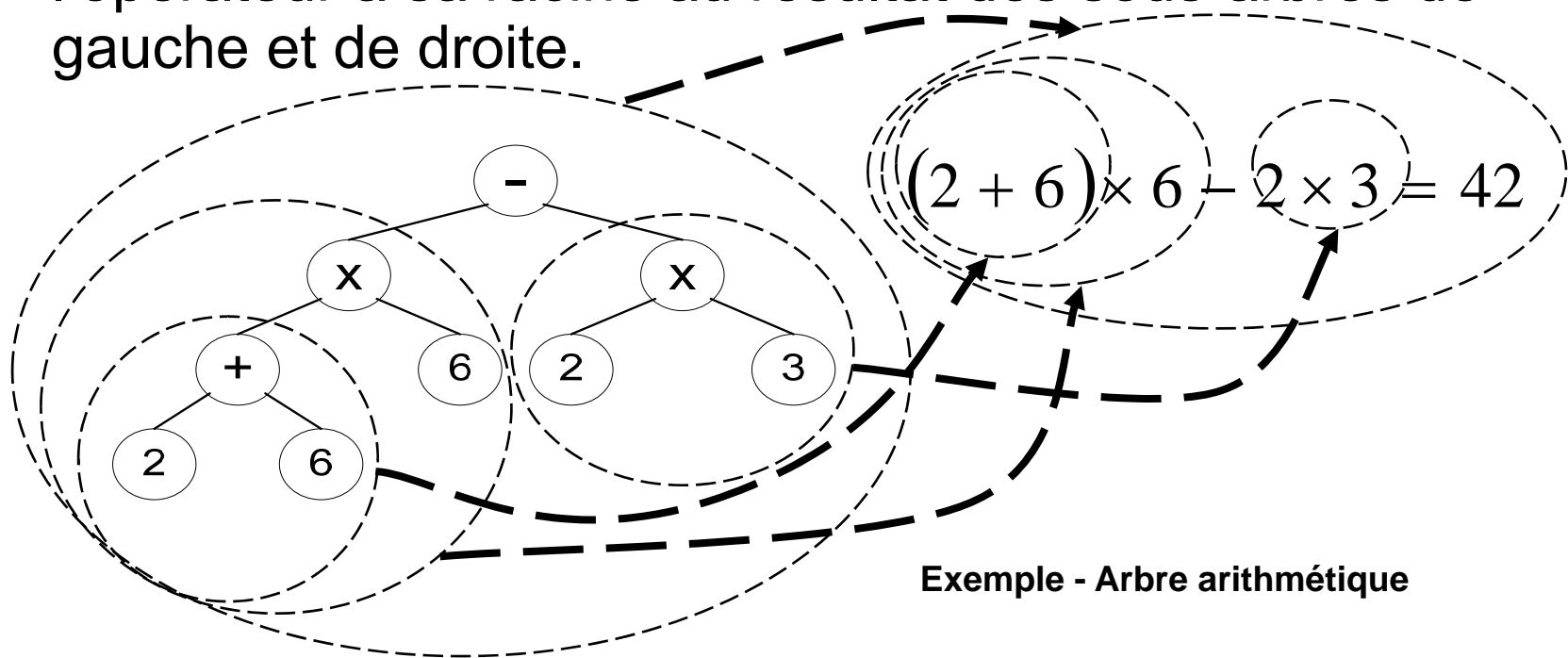
- Parce qu'on connaît le degré d'un arbre binaire, on peut réserver d'avance des cases d'un tableau séquentiel pour chacun des éléments.



- Arbitrairement, on décide de ne pas utiliser la case à l'index 0. Ainsi, pour chaque nœud à l'index i , on a que:
 - Nœud à l'index $2i$ est son fils de gauche.
 - Nœud à l'index $2i+1$ est son fils de droite.
 - Nœud à l'index $\lfloor i/2 \rfloor$ est son parent.

Arbres arithmétiques

- Les arbres binaires peuvent être utiles, par exemple, pour l'évaluation d'expressions arithmétiques.
- Les opérandes sont placées dans les nœuds feuilles, les opérateurs binaires dans les autres nœuds.
- L'évaluation d'un arbre arithmétique se fait en appliquant l'opérateur à sa racine au résultat des sous-arbres de gauche et de droite.



Exemple - Arbre arithmétique

Arbres I

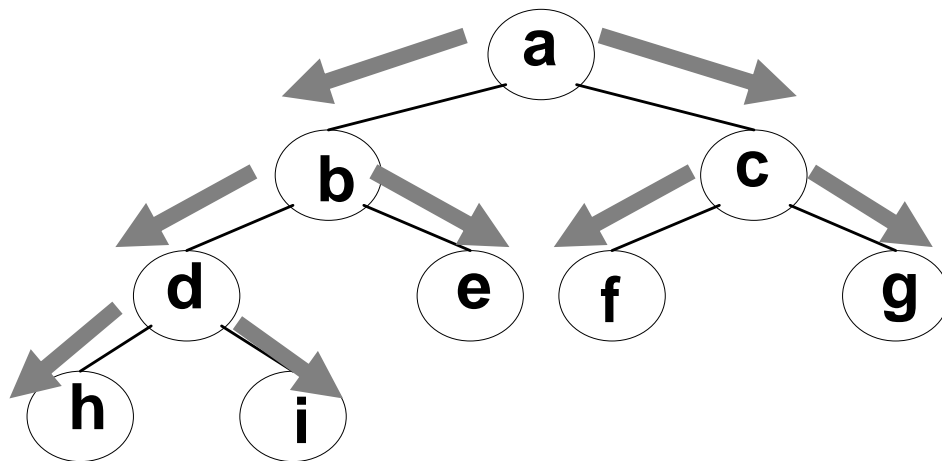
- Définition et généralités sur les arbres
- Arbres binaires
- Méthodes de parcours
- Arbres binaires de recherches

Méthode de parcours d'arbres

- La façon de traiter l'information contenue dans un arbre binaire dépend de la manière dont ses nœuds sont visités.
- Quatre méthodes de parcours sont couramment utilisées:
 - Parcours postordre
 - Parcours préordre
 - Parcours en ordre
 - Parcours par niveau
- Ces quatre méthodes font appel à la nature récursive des arbres.

Postordre

- Dans le parcours postordre, les descendants d'un nœud sont traités avant lui:
 1. Fils de gauche
 2. Fils de droite
 3. Noeud



Résultat de parcours:
h i d e b f g c a

Postordre

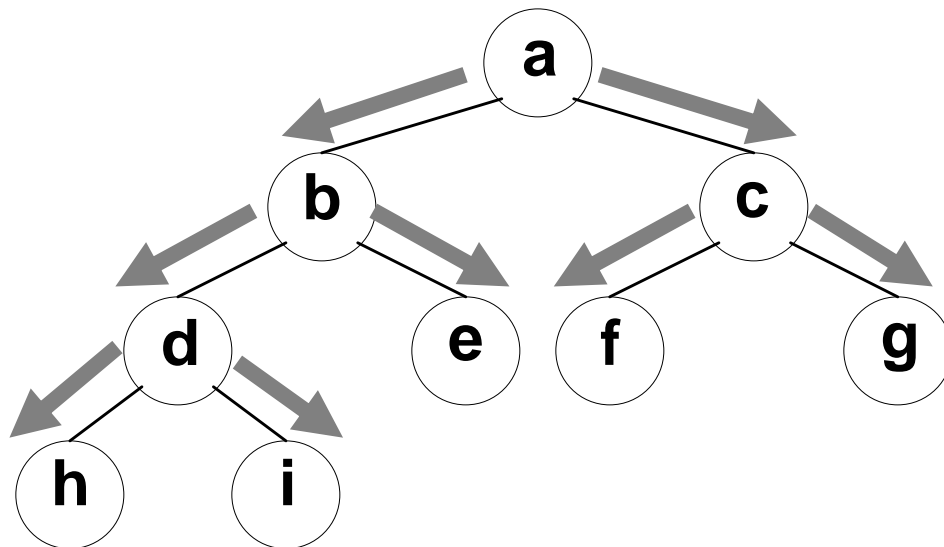
- Dans le parcours postordre, les descendants d'un nœud sont traités avant lui:
 1. Fils de gauche
 2. Fils de droite
 3. Noeud

```
// implémentation post-ordre
private void printTree( BinaryNode<AnyType> t )
{
    if( t != null )
    {
        printTree( t.left );
        printTree( t.right );
        System.out.println( t.element );
    }
}
```

Préordre

- Dans le parcours préordre, les descendants d'un nœud sont traités après lui:
 1. Nœud
 2. Fils de gauche
 3. Fils de droite

Résultat de parcours:
a b d h i e c f g



Préordre

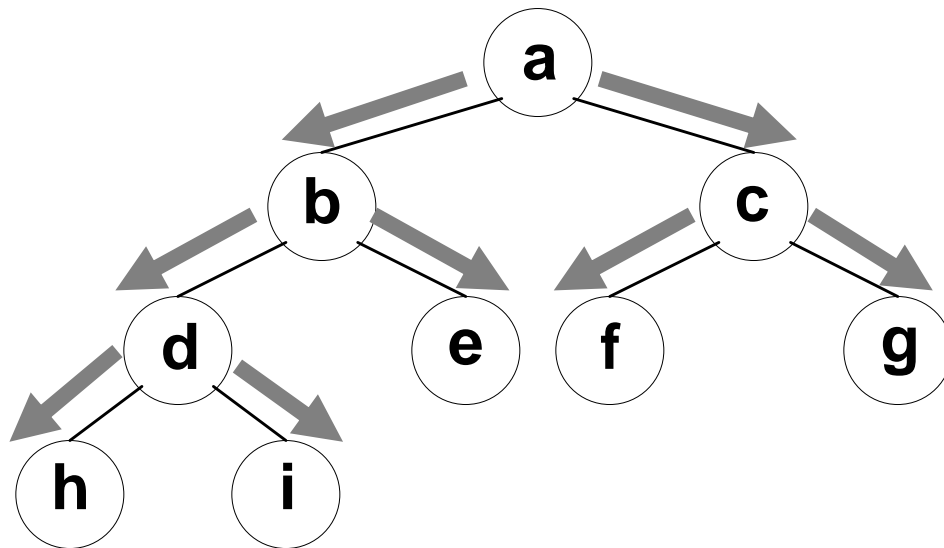
- Dans le parcours préordre, les descendants d'un nœud sont traités après lui:
 1. Nœud
 2. Fils de gauche
 3. Fils de droite

```
// implémentation pré-ordre
private void printTree( BinaryNode<AnyType> t )
{
    if( t != null )
    {
        System.out.println( t.element );
        printTree( t.left );
        printTree( t.right );
    }
}
```

En ordre

- Dans le parcours en ordre, un descendant est traité avant le nœud, l'autre est traité après lui:
 1. Fils de gauche
 2. Noeud
 3. Fils de droite

Résultat de parcours:
hdibeafcg



En ordre

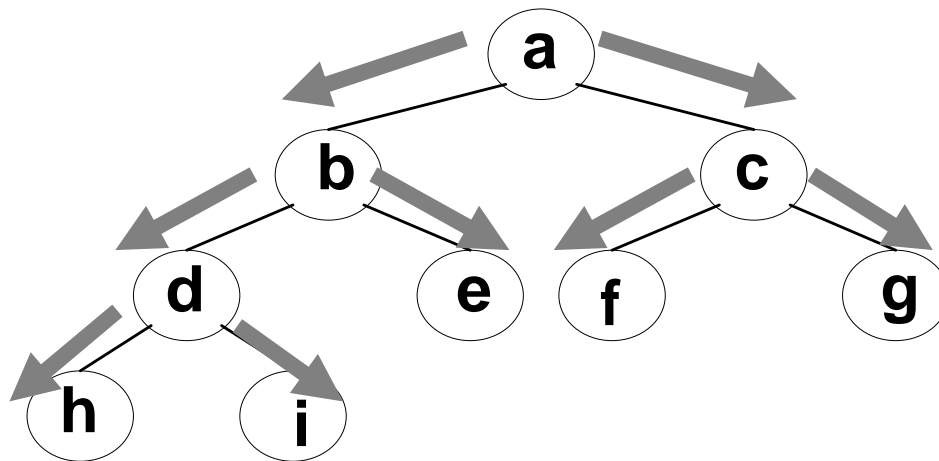
- Dans le parcours en ordre, un descendant est traité avant le nœud, l'autre est traité après lui:
 1. Fils de gauche
 2. Noeud
 3. Fils de droite

```
// implémentation en ordre
private void printTree( BinaryNode<AnyType> t )
{
    if( t != null )
    {
        printTree( t.left );
        System.out.println( t.element );
        printTree( t.right );
    }
}
```

Par niveau

- Dans le parcours par niveau, tous les nœuds d'un même niveau sont traités avant de descendre au niveau suivant.
- L'implémentation n'est pas récursive! Elle peut être réalisée en employant une file.

Résultat de parcours:
a b c d e f g h i



Arbres binaires de recherche - Plan

- Définition et généralités sur les arbres
- Arbres binaires
- Méthodes de parcours
- Arbres binaires de recherches
 - Définition
 - Exemple de contexte d'utilisation
 - Opérations
 - Recherche d'un élément
 - Insertion
 - Retrait
 - Problème de dégénération de l'arbre.

Arbres binaires de recherche

- Les arbres binaires de recherche sont des arbres binaires structurés de façon à respecter la propriété suivante:
Pour chaque nœud i :
 - Tous les éléments de son **sous-arbre de gauche** ont une valeur de clé **inférieure** à celle du nœud i .
 - Tous les éléments de son **sous-arbre de droite** ont une valeur de clé **supérieure** à celle du nœud i .
- Cette structure permet un temps moyen de $O(\log \mathcal{N})$ pour la plupart des opérations.

Exemple – dossiers d'étudiants

- Chaque étudiant possède un identificateur unique (un matricule). On peut utiliser cet identificateur (ou clé) pour bâtir un arbre binaire de recherche permettant une recherche plus rapide qu'avec une simple liste.
- Soit l'ensemble de matricules suivant:
101, 42, 28, 12, 54, 65, 103, 115, 77
- Si l'on cherche le dossier associé au matricule 77, combien de nœuds seront visités?

Exemple (suite)

Algorithme de recherche:

Initialiser le premier nœud à visiter à la racine.

Tant que l'élément recherché n'est pas trouvé et qu'il reste des nœuds à visiter:

Si l'élément courant == l'élément recherché

L'élément est trouvé, retourner l'élément

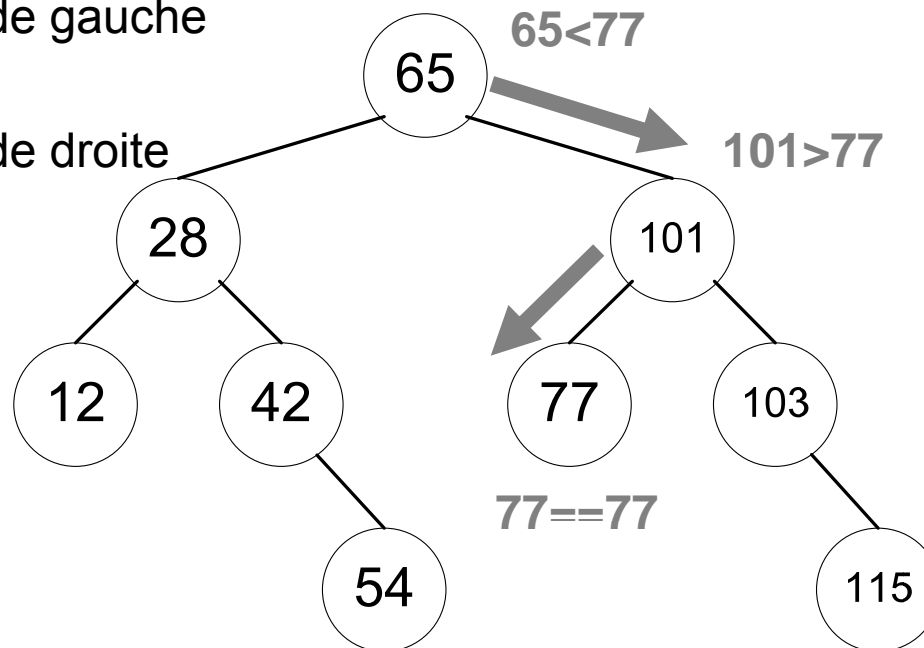
sinon

si l'élément recherché < l'élément courant

suivre le fils de gauche

sinon

suivre le fils de droite



3 nœuds visités

Opérations - Insertion

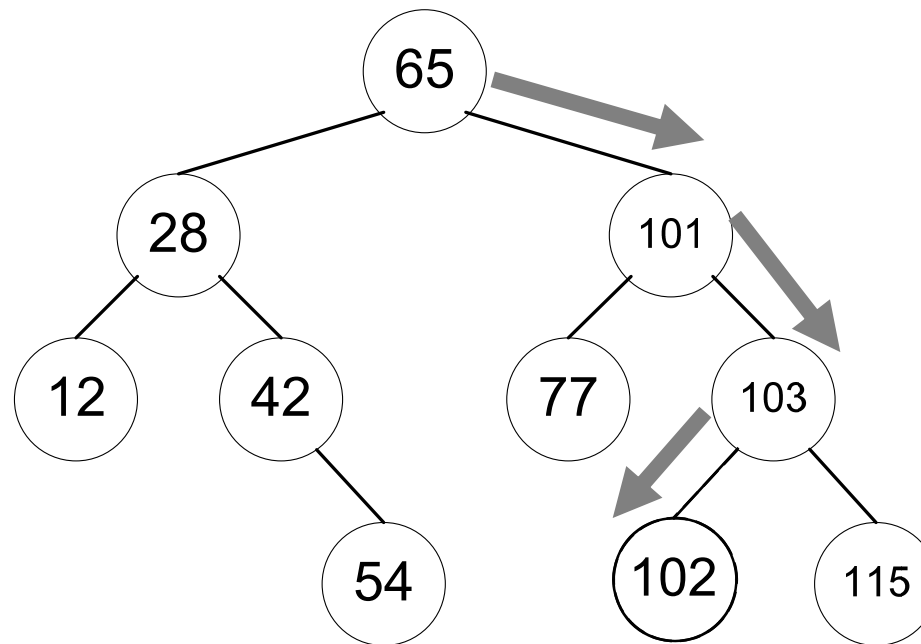
L'insertion d'un nouveau nœud se fait toujours à la fin d'une branche existante de l'arbre.

Algorithme d'insertion:

Rechercher la position d'insertion

Raccorder le nouveau nœud à son parent

Exemple: ajout du matricule 102
à l'arbre existant



Insertion - code

```
private BinaryNode<AnyType>
insert(AnyType x, BinaryNode<AnyType> t)
{
    // Insérer si position courante est une feuille
    if( t == null )
        return new BinaryNode<AnyType>( x, null, null );

    // Sinon trouver position
    int compareResult = x.compareTo( t.element )

    if(compareResult < 0 )
        t.left = insert( x, t.left );
    else if( compareResult > 0 )
        t.right = insert( x, t.right );
    else
        ; // Rejeter en cas de doublon

    return t;
}
```


Opérations - Retrait

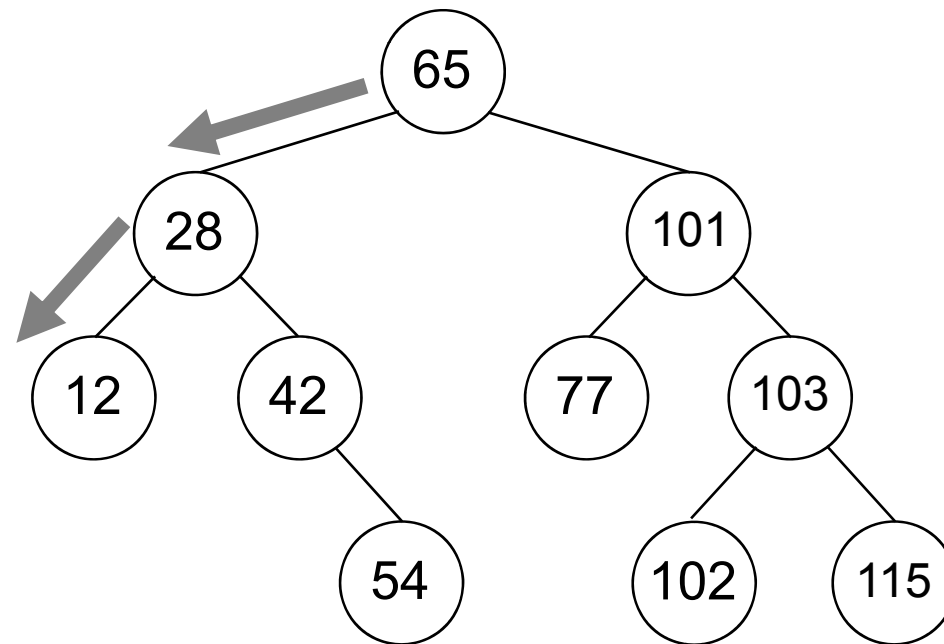
Le retrait d'un élément dans un arbre binaire de recherche diffère selon le nombre de descendants que le nœud à retirer possède.

1. Si le nœud ne possède aucun descendant:
Cas trivial, le nœud est simplement retiré.
1. Si le nœud possède un seul descendant:
L'unique descendant prend la place du nœud retiré.
1. Si le nœud possède deux descendants:
Le plus petit nœud de la branche de droite prend la place du nœud retiré.

Opérations – Retrait (suite)

Exemple: retrait du matricule 12

Le nœud à retirer n'a pas de descendant

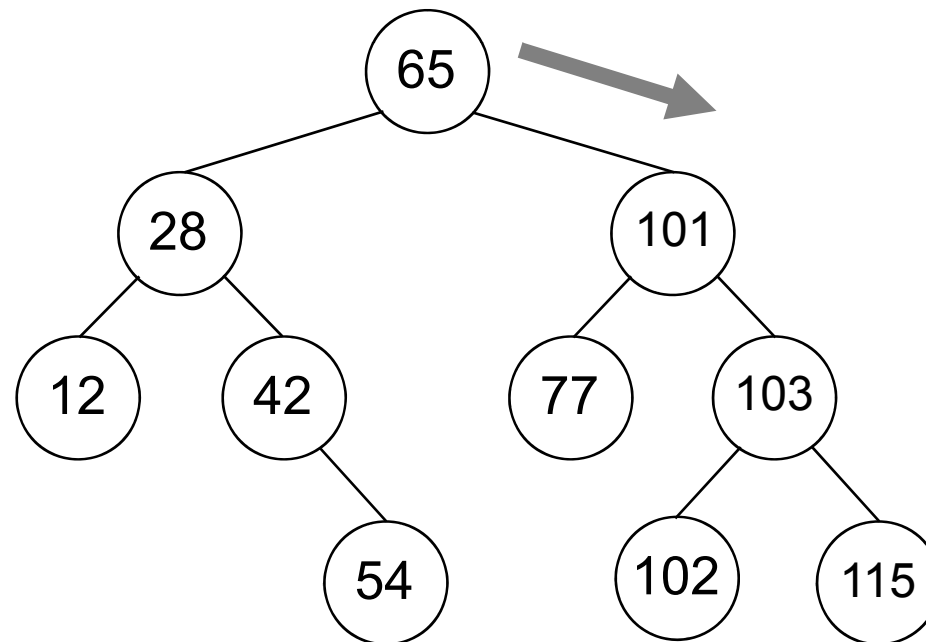


Opérations – Retrait (suite)

Exemple: retrait du matricule 101

Le nœud à retirer a deux fils.

Remplacer par le plus petit nœud de la branche de droite



Retrait - Implémentation

Le retrait d'un nœud avec deux descendants peut être implémenté de deux façons différentes:

1. Écrasement de l'élément à retirer par l'élément minimal du sous-arbre de droite et effacement du nœud minimal.
1. Ajustement des liens du nœud minimal de sous-arbre de droite et effacement du nœud contenant l'élément à retirer.

L'efficacité d'une méthode par rapport à l'autre dépend de la nature des éléments de l'arbre.

Retrait - code

```
private BinaryNode<AnyType> remove(AnyType x, BinaryNode<AnyType> t)
{
    // Si x absent, ressortir
    if( t == null )
        return t;

    // Rechercher position de x
    int compareResult = x.compareTo( t.element );

    if( compareResult < 0 )
        t.left = remove( x, t.left );
    else if( compareResult > 0 )
        t.right = remove( x, t.right );

    // Une fois x trouvé
    else if( t.left != null && t.right != null ) // Cas deux enfants
    {
        t.element = findMin( t.right ).element;
        t.right = remove( t.element, t.right );
    }
    else // Cas trivial
        t = ( t.left != null ) ? t.left : t.right;

    return t;
}
```

Recherche du min

```
public AnyType findMin( )
{
    if( isEmpty( ) )
        throw new UnderflowException( );

    return findMin( root ).element;
}

private BinaryNode<AnyType> findMin( BinaryNode<AnyType> t )
{
    if( t == null )
        return null;
    else if( t.left == null )
        return t;

    return findMin( t.left );
}
```

Recherche du max

```
public AnyType findMax( )
{
    if( isEmpty( ) )
        throw new UnderflowException( );

    return findMax( root ).element;
}

private BinaryNode<AnyType> findMax( BinaryNode<AnyType> t )
{
    if( t != null )
        while( t.right != null )
            t = t.right;

    return t;
}
```

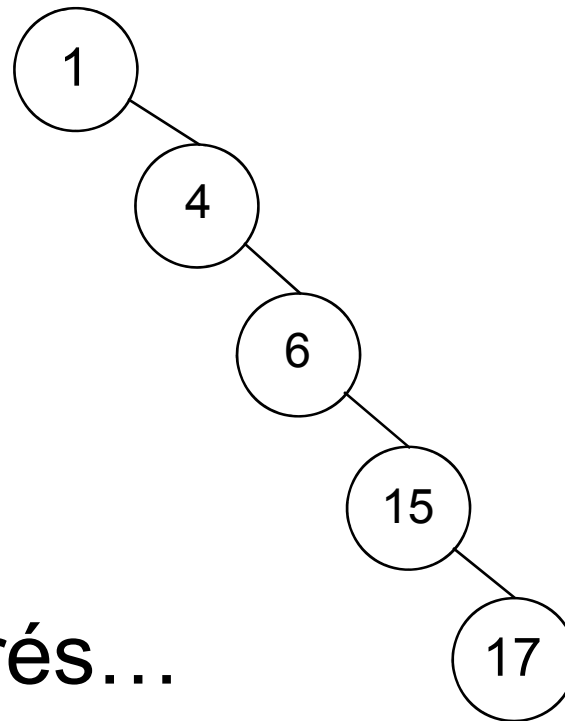
Problème lié à l'insertion

- L'ordre d'insertion des éléments dans un arbre binaire de recherche est important.
- Un arbre peut devenir **dégénéré** et perdre ainsi ses propriétés de recherche en temps $O(\lg(n))$.

Exemple:
insérer en ordre les
éléments 1,4,6,15,17

On tombe sur une simple liste.
Temps de recherche: $O(\mathcal{N})$

Solution: arbres équilibrés...



Arbres binaires - analyse

- Le prix d'une opération (recherche, insertion, retrait) est proportionnel au nombre de noeuds visités
- Donc, coût proportionnel à $1 + \text{profondeur}$
- Meilleur cas: arbre équilibré (les feuilles à peu près toutes à la même profondeur)
 - insertion et retrait aléatoire tendent à créer un arbre équilibré
 - profondeur = $\lfloor \lg n \rfloor$
- Pire cas: liste chaînée
 - par exemple lors de l'insertion d'éléments ordonnés
 - profondeur = n
- Donc, le coût est $O(\lg n)$ dans le meilleur cas et $O(n)$ dans le pire cas