ITI 1521. Introduction à l'informatique II*

Marcel Turcotte
École d'ingénierie et de technologie de l'information

Version du 26 mars 2011

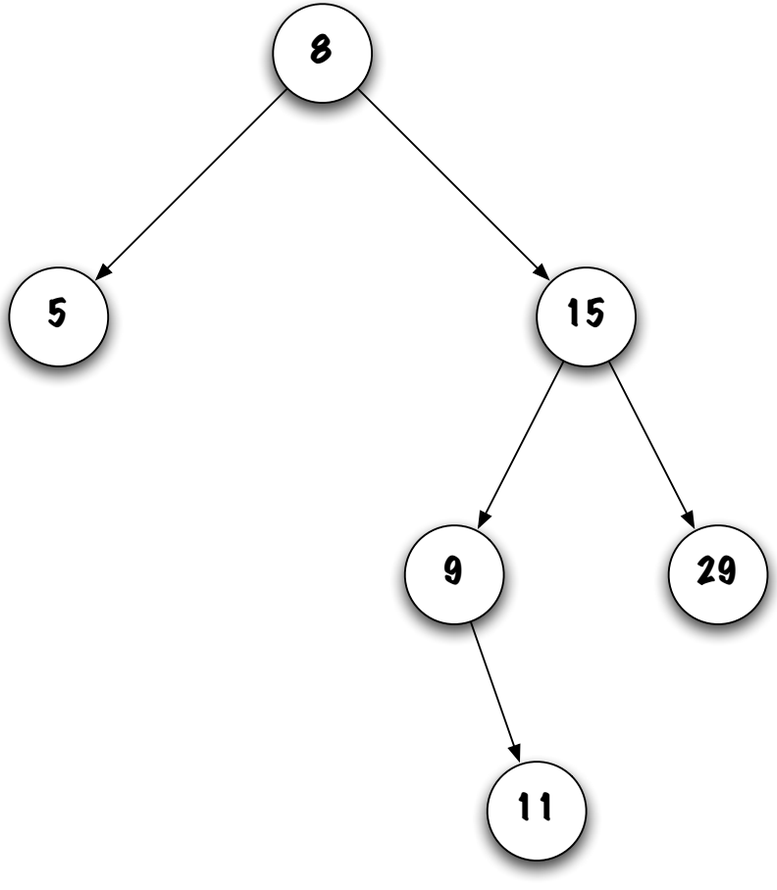
Résumé

– Arbre binaire de recherche (partie 1)

*. Ces notes de cours ont été conçues afin d'être visualiser sur un écran d'ordinateur.

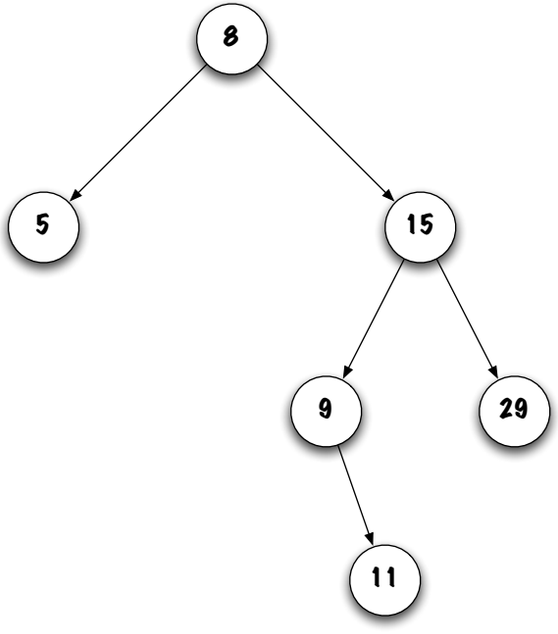
Arbre binaire

Un **arbre binaire** est une structure arborescente (hiérarchique) telle que chaque **noeud** possède une valeur et deux fils (descendants), que l'on nomme **gauche** et **droite**.



Applications (arbres généraux)

- Représenter des informations hiérarchiques tels que les systèmes de fichiers hiérarchiques (HFS) (répertoires et sous-répertoires), programmes (arbre d'analyse syntaxique) ;
- Les arbres de Huffman servent à la compression d'information (fichiers) ;
- L'arbre binaire est une structure de données efficace servant à l'implémentation de types abstraits de données tels que les monceaux (≪heap≫), files de priorité, et les ensembles.



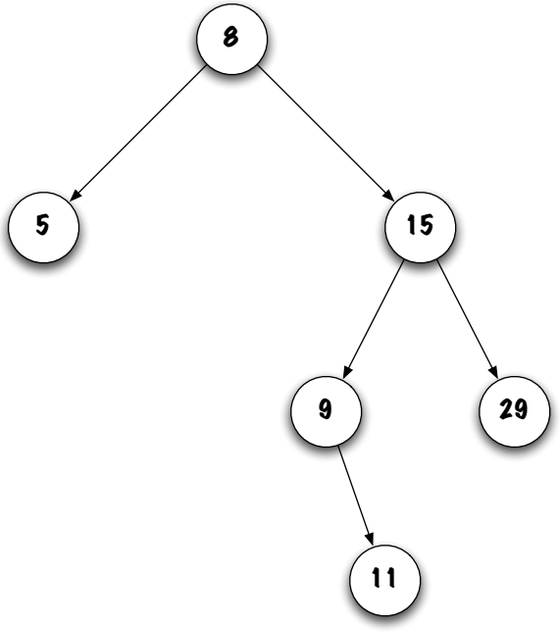
Tous les noeuds ont un seul parent ; à l'exception d'un noeud qui n'a aucun parent et que l'on appelle la **racine** (c'est le noeud tout en haut du diagramme).

Chaque noeud a 0, 1 ou 2 fils.

Les noeuds sans enfant sont les **feuilles** de l'arbre (ou noeuds extérieurs).

Les liens entre les noeuds sont les **branches** de l'arbre.

Arbre binaire



Un noeud et ses descendants est un **sous-arbre**.

La **taille** d'un arbre est le nombre de noeuds de l'arbre. Un arbre **vide** a une taille 0.

Puisque nous ne traiterons que les arbres binaires, j'utiliserai parfois le terme arbre pour désigner un arbre binaire.

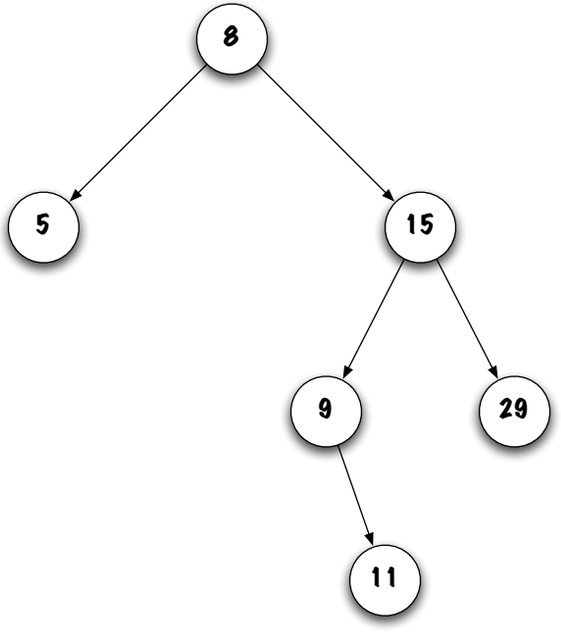
Arbre binaire

On peut donner une définition récursive

– Un arbre binaire est vide, ou ;
– Un arbre binaire est constitué d'une valeur et deux sous-arbres (gauche et droite).

Arbre binaire

La **profondeur d'un noeud** représente le nombre de liens qu'il faut suivre à partir de la racine afin d'accéder à ce noeud. La racine est le noeud le plus accèssible.

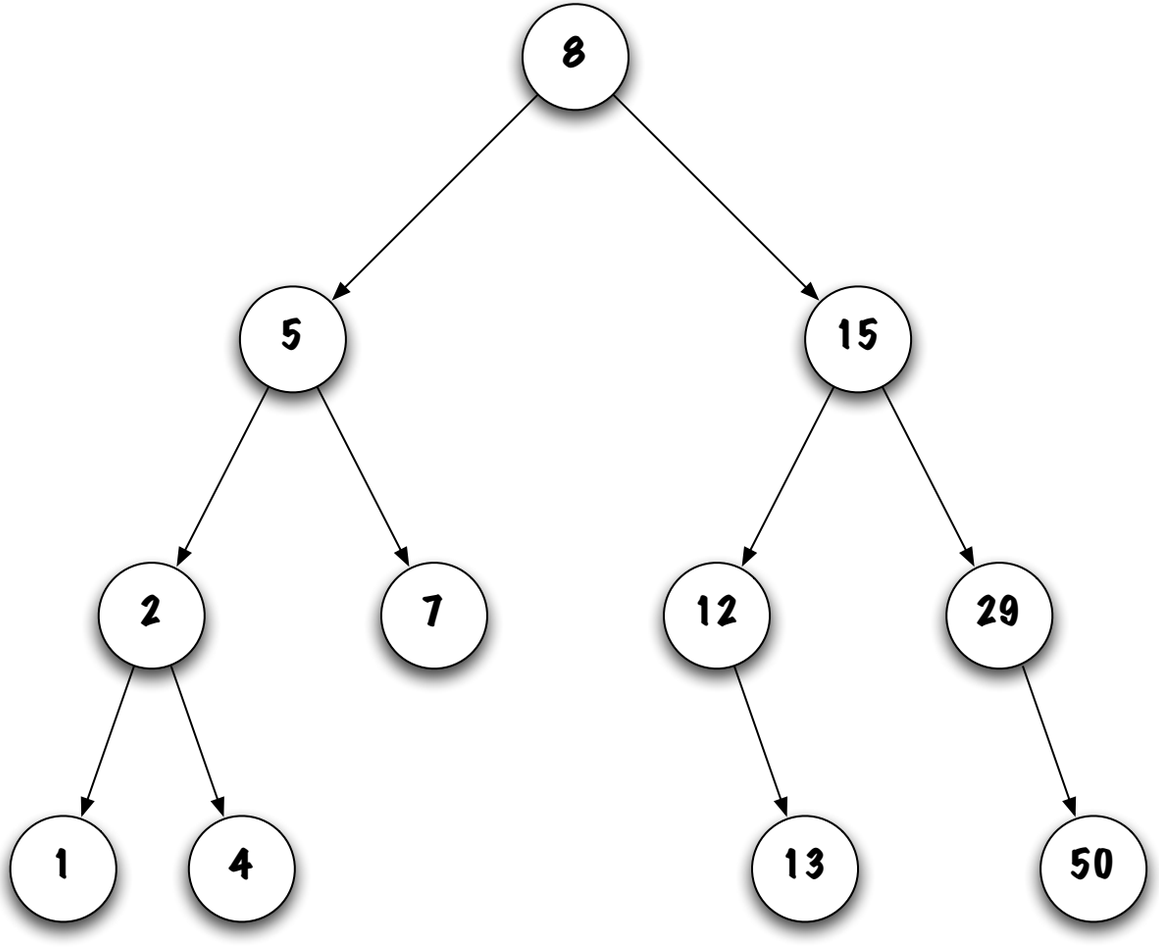


Quelle la profondeur de la racine ? La racine est toujours à profondeur 0.

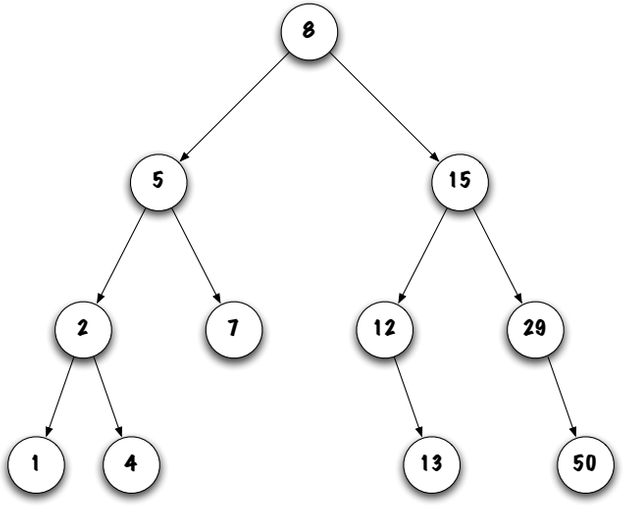
La **profondeur d'un arbre** est la profondeur maximale d'un noeud de l'arbre.

Arbre binaire

Tous les arbres présentés ont une propriété en commun, quelle est-elle ?



Arbre binaire de recherche



Un **arbre binaire de recherche** est un arbre binaire dont les noeuds vérifient les propriétés suivantes :

- tous les noeuds de son sous-arbre gauche ont des valeurs plus petites que celle de ce noeud (ou encore le sous-arbre gauche vide) ;
- tous les noeuds de son sous-arbre droit ont des valeurs plus grandes que celle de ce noeud (ou ce sous arbre est vide).

Corollaire : les valeurs sont uniques.

Arbre binaire de recherche

Implémentation d'un arbre binaire de recherche, que faut-il ?

Et bien oui ! Nous utiliserons une classe **Node** imbriquée et ≪static≫. Quelles sont ses variables d'instance ?

Les variables d'instance sont : **value**, **left** et **right**.

Quel est le type de ces variables ? **value** est un **Comparable**, **left** et **right** sont de type **Node**.

Arbre binaire de recherche

Une classe imbriquée ≪static≫ afin de sauvegarder une valeur et créer la structure de l'arbre.

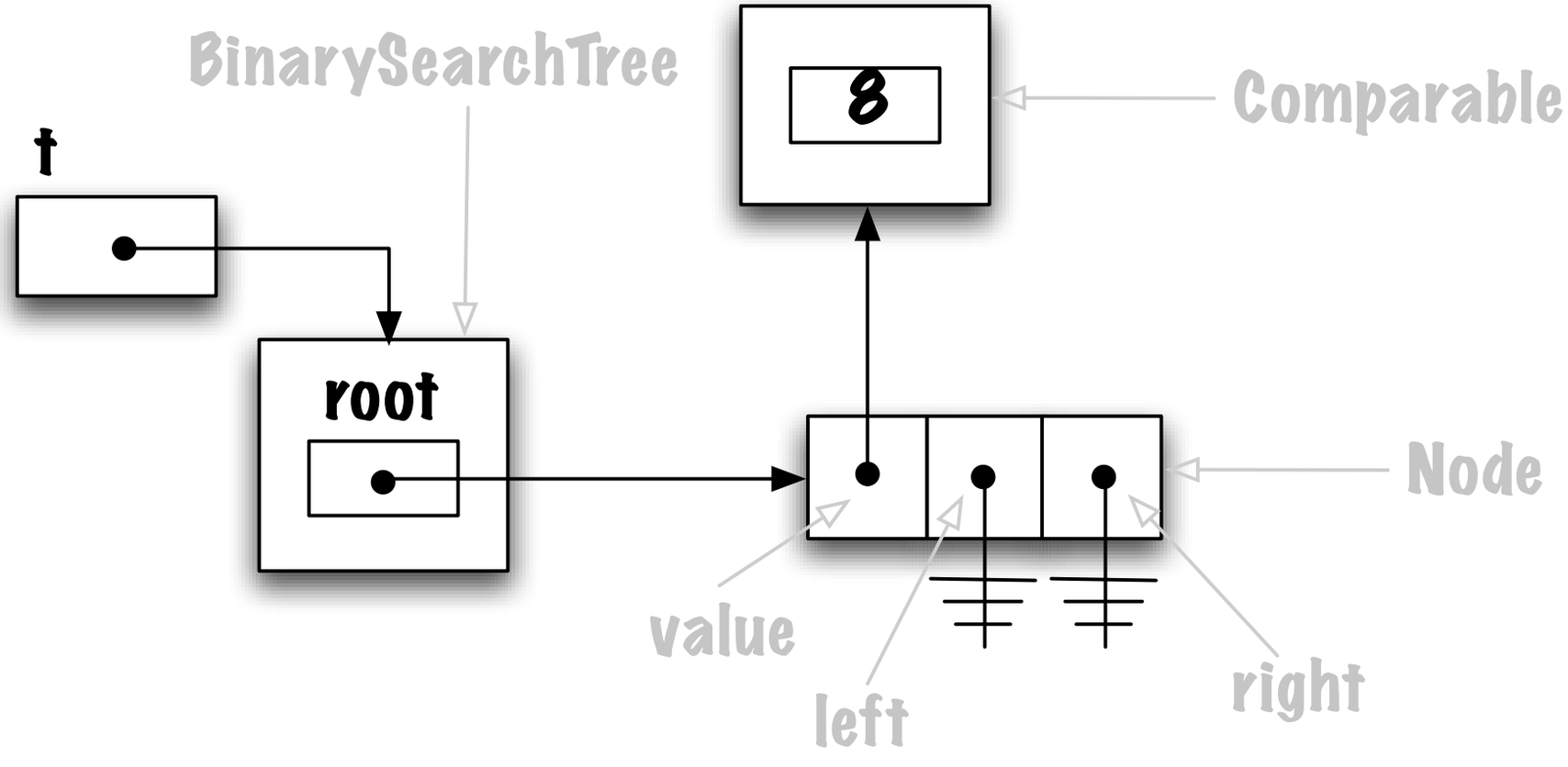
```
public class BinarySearchTree< E extends Comparable<E> > {

    private static class Node<E> {
        private E value;
        private Node<E> left;
        private Node<E> right;
    }
```

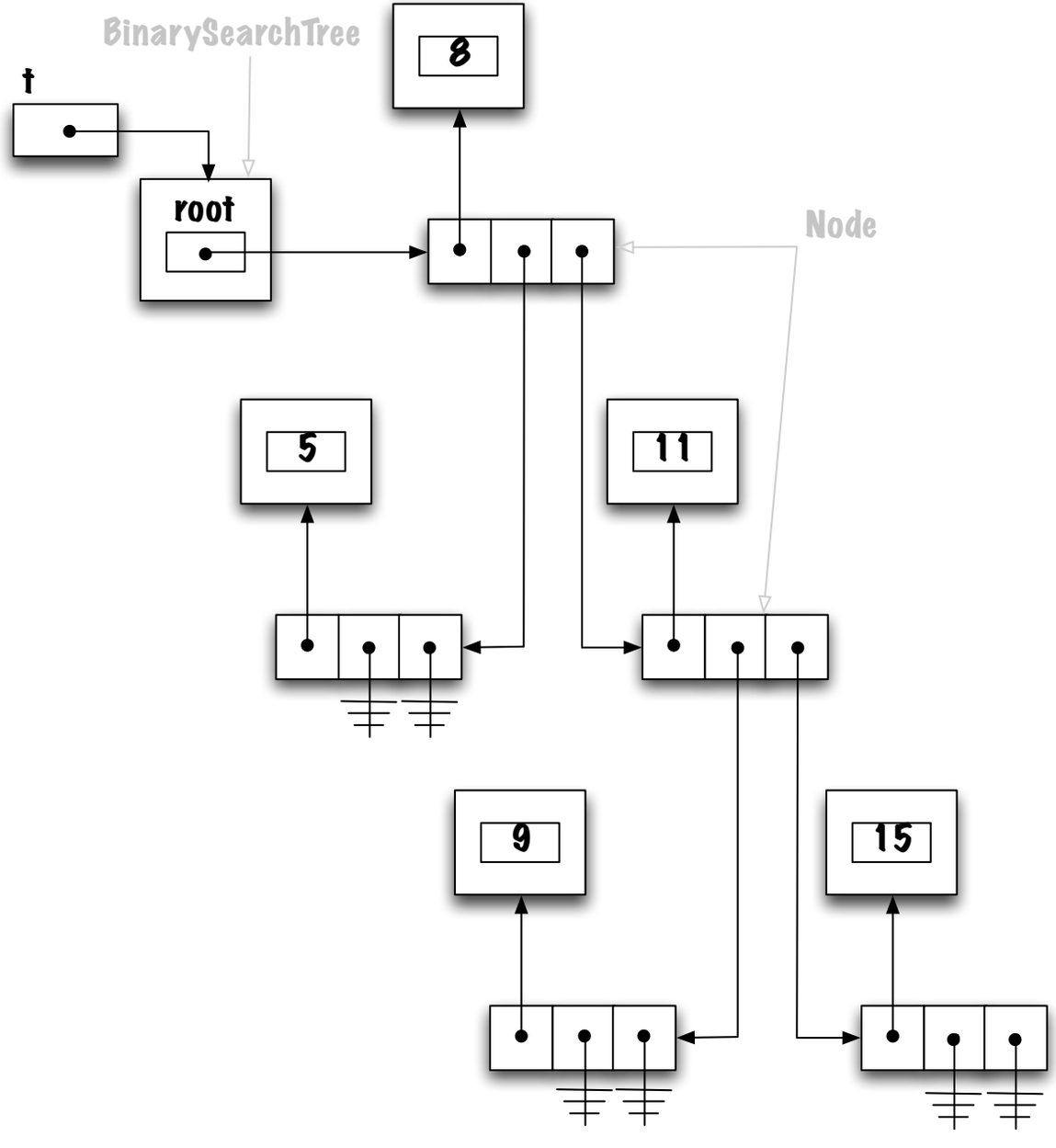
Arbre binaire de recherche

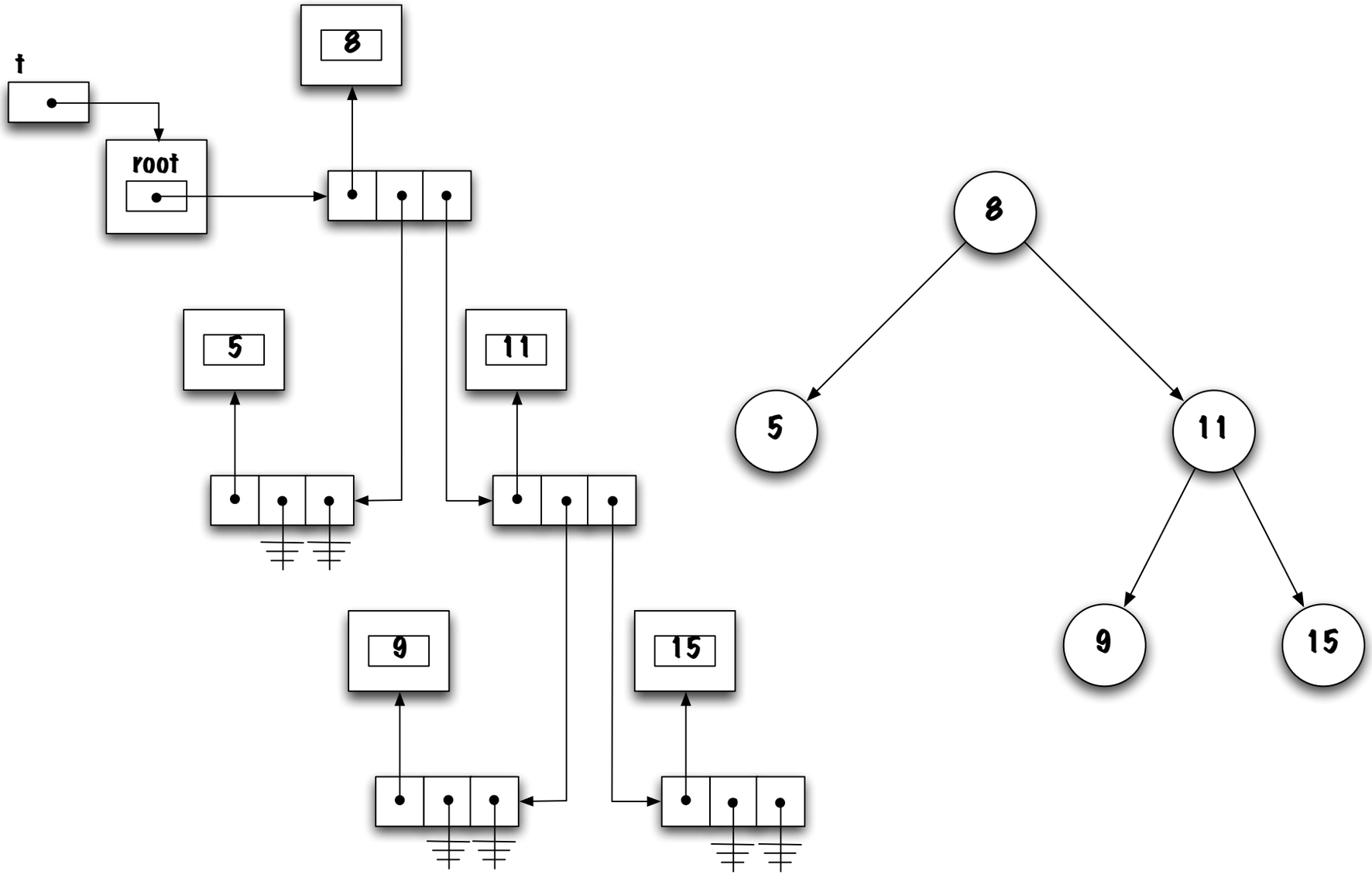
Variable(s) d'instance de la classe **BinarySearchTree** ?

```
public class BinarySearchTree< E extends Comparable<E> > {

    private static class Node<E> {
        private E value;
        private Node<E> left;
        private Node<E> right;
    }

    private Node<E> root;
```

Diagramme de mémoire




BinarySearchTree
t
root
8
Node
5
11
9
15

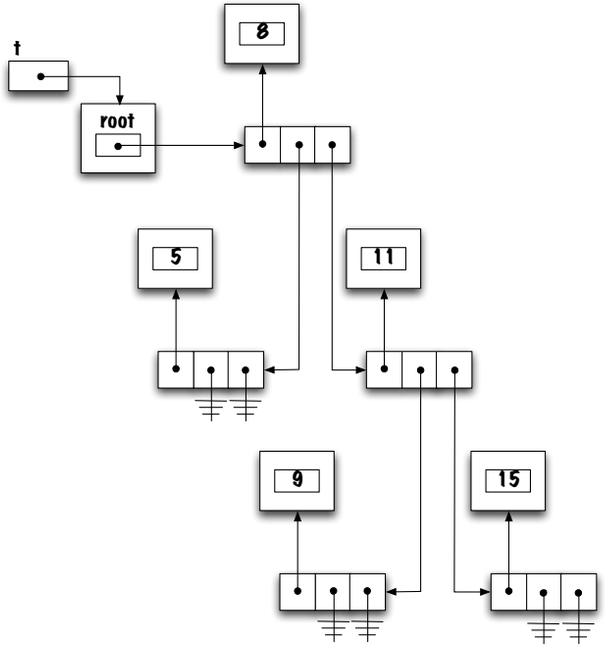

t
root
8
5
11
9
15
8
5
11
9
15

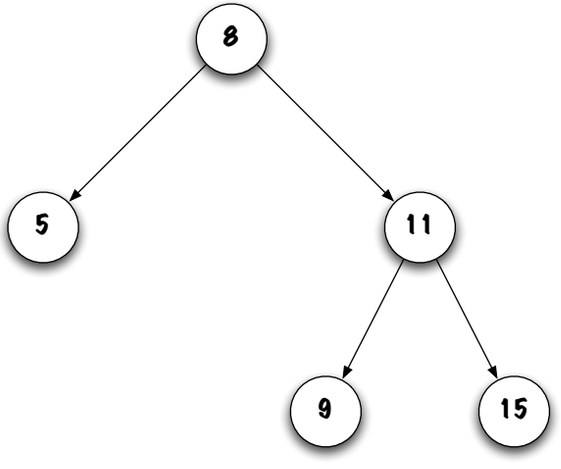
Observations

Une feuille est un noeud tel que ses deux descendants sont **null**.

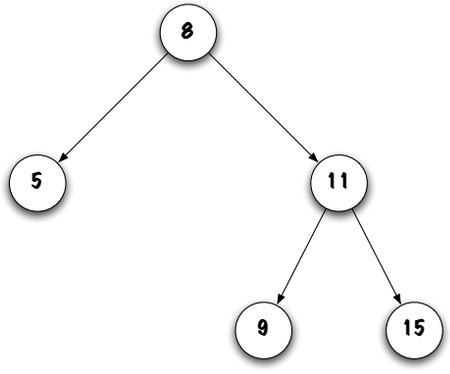
La variable **root** peut être **null**, alors l'arbre est vide et de taille 0.

Par souci d'économie, j'utiliserai parfois la représentation plus abstraite de droite.



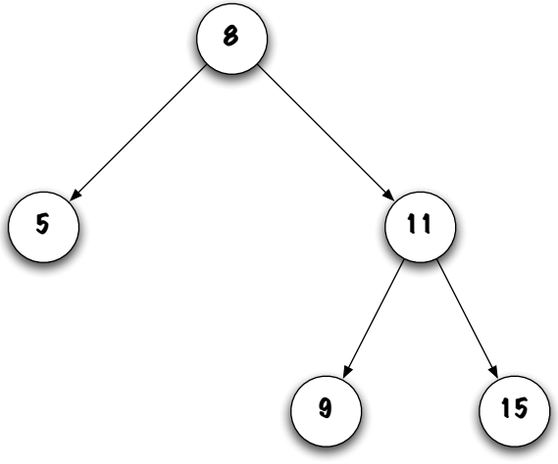


boolean contains(E obj)



1. Arbre vide ? **obj** est absent ;
2. La racine locale contient **obj** ? **obj** est présent ; Sinon ? Où cherche-t-on ?
3. Si **obj** est plus petit que la valeur sauvegardé dans le noeud courant ? Chercher **obj** dans le sous-arbre gauche ;
4. Sinon (**obj** est forcément plus grand que la valeur du noeud courant) ? Chercher **obj** dans le sous-arbre droit.

boolean contains(E obj)



Exercices : appliquez l'algorithme afin de trouver les valeurs 8, 9 et 7 dans l'arbre ci-haut.

public boolean contains(E obj)

La présentation suggère un algorithme récursif. Que sera la signature de la méthode ?

```
public boolean contains( E obj ) {
    // précondition:
    if ( obj == null ) {
        throw new IllegalArgumentException( "null" );
    }
    return contains( root, obj );
}
```

Tout comme le traitement récursif des listes chaînées, nos méthodes auront deux parties, une partie publique, et une partie privée dont la signature comporte un paramètre de type **Node**.

boolean contains(Node<E> current, E obj)

Cas de base :

```
if ( current == null ) {
    result = false;
}
```

mais aussi

```
if ( obj.compareTo( current.value ) == 0 ) {
    result = true;
}
```

boolean contains(Node<E> current, E obj)

Cas général : Chercher à gauche ou à droite (récursivement).

```
if ( obj.compareTo( current.value ) < 0 ) {
     result = contains( obj, current.left );
} else {
     result = contains( obj, current.right );
}
```

boolean contains(Node<E> current, E obj)

```
private boolean contains( Node<E> current, E obj ) {
    boolean result;
    if ( current == null ) {
        result = false;
    } else {
        int test = obj.compareTo( current.value );
        if ( test == 0 ) {
            result = true;
        } else if ( test < 0 ) {
            result = contains( obj, current.left );
        } else {
            result = contains( obj, current.right );
        }
    }
    return result;
}
```

public boolean contains(E obj) (prise 2)

Est-ce que la méthode **boolean contains(E o)** est forcément récursive ? Non.

Élaborez une stratégie.

1. Utilise une variable **current** de type **Node** ;
2. Initialise cette variable afin de désigner la racine de l'arbre ;
3. Si **current** est **null** alors la valeur est absente, fin ;
4. Si **current.value** est la valeur recherchée, fin ;
5. Si la valeur recherchée est plus petite alors **current = current.left**, allez à 3 ;
6. Sinon **current = current.right**, allez à 3.

public boolean contains(E obj) (prise 2)

```
public boolean contains2( E obj ) {

    boolean found = false;
    Node<E> current = root;
    while ( ! found && current != null ) {
        int test = obj.compareTo( current.value );
        if ( test == 0 ) {
            found = true;
        } else if ( test < 0 ) {
            current = current.left;
        } else {
            current = current.right;
        }
    }
    return found;
}
```

Traverser un arbre

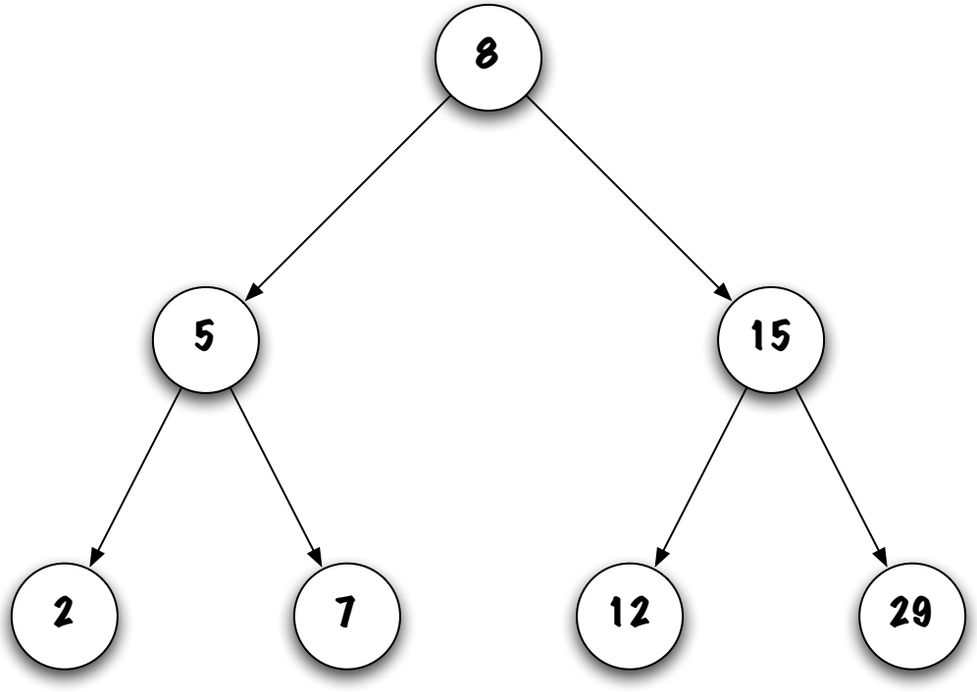
Il faut parfois **traverser** l'arbre afin de **visiter** tous ses noeuds.

Lorsqu'on **visite** un noeud, on exécute certaines opérations sur le noeud.

- Parcours **préfixe** ou **pré-ordre** (pre-order) : visiter la racine, traverser le sous-arbre gauche, traverser le sous-arbre droit ;
- Parcours **infixe** ou **symétrique** (in-order) : traverser le sous arbre gauche, visiter la racine, traverser le sous-arbre droit ;
- Parcours **suffixe** ou **post-odre** (post-order) : traverser le sous-arbre gauche, traverser le sous-arbre droit, visiter la racine.

Exercices

L'opération la plus simple consiste à afficher la valeur sauvegardée dans le noeud.



Donnez le résultat affiché pour chaque stratégie, **pré-ordre**, **symétrique** et **post-ordre**.

Quelle stratégie affiche les données en ordre croissant ?