ITI 1521. Introduction à l'informatique II

Arbres binaires de recherche : concept

by

Marcel Turcotte

Version du 25 mars 2020

Préambule

Préambule

Aperçu

Aperçu

Arbres binaires de recherche : concept

Nous débutons par un survol des applications des arbres en informatique afin de représenter des données hiérarchiques, pour la compression des données, et l'accès efficace aux éléments. Nous examinons l'implémentation chaînée des arbres. Nous portons une attention particulière aux arbres binaires de recherche.

Objectif général :

- Cette semaine, vous serez en mesure de concevoir et modifier des programmes informatiques reposant sur le concept d'arbre binaire de recherche.

Préambule

Objectifs d'apprentissage

Objectifs d'apprentissage

- **Nommer** des applications des arbres binaires de recherche.
- **Décrire** les propriétés essentielles des arbres binaires de recherche.

Lectures :

- Pages 257-268 et 282-296 de E. Koffman et P. Wolfgang.

Préambule

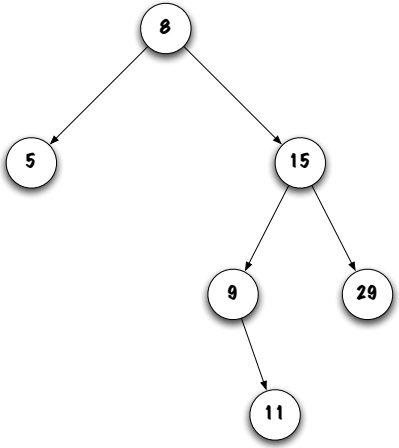
Plan du module

Plan

Théorie

Définition

Un **arbre binaire** est une structure de données **hiérarchique** telle que chaque **noeud** sauvegarde une **valeur** et possède au plus deux fils, que l'on nomme **gauche** («*left*») et **droit** («*right*»).



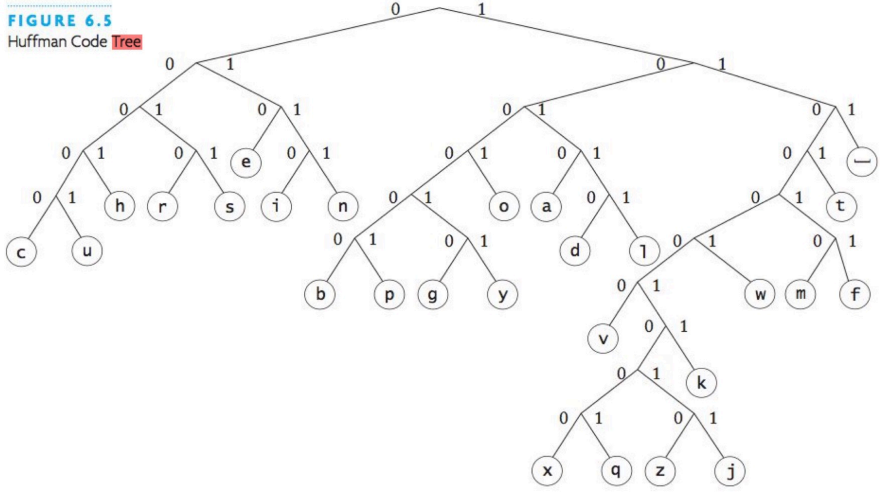
Théorie

Applications

Applications (arbres généraux)

- Représenter des informations hiérarchiques telles 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, les structures associatives et les ensembles.

FIGURE 6.5
Huffman Code Tree



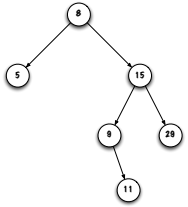
However, to *decode* a file of letters and spaces, you walk down the Huffman tree, starting at the root, until you reach a letter and then append that letter to the output text. Once you have reached a letter, go back to the root. Here is an example. The substrings that represent the individual letters are shown in alternate colors to help you follow the process. The underscore in the second line represents a space character (code is 111).

```
100010100111101010100010101110100011
  g   o   _   e   a   g     l   e   s
```

Théorie

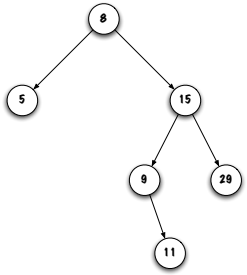
Définitions

Définitions



- 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.

Définitions



- Un noeud et ses descendants sont 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.

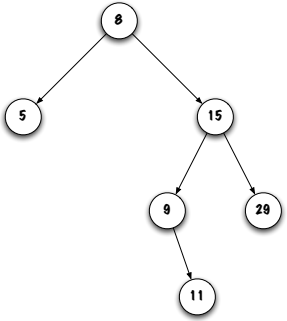
Définitions

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).

Définitions

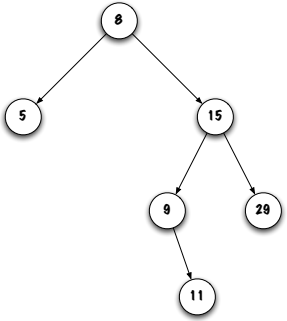
La **profondeur d'un noeud** représente le nombre de liens qu'il faut suivre à partir de la racine afin d'accéder à ce noeud. La racine est le noeud le plus accessible.



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

Définitions

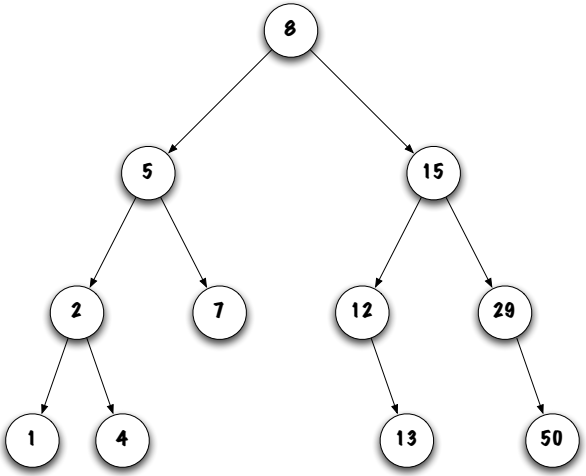
La **profondeur d'un noeud** représente le nombre de liens qu'il faut suivre à partir de la racine afin d'accéder à ce noeud. La racine est le noeud le plus accessible.



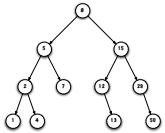
La **profondeur d'un arbre** est la profondeur maximale d'un noeud de l'arbre.

Discussion

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



Définition



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 est 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.

Implémentation

Implémentation

- **Comment** allons nous implémenter cette classe ?
- **Eh 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 de type **Comparable**, **left** et **right** sont de type **Node**.

Implémentation

Une **classe imbriquée «static»** afin de sauvegarder une valeur et créer la structure de l'arbre.

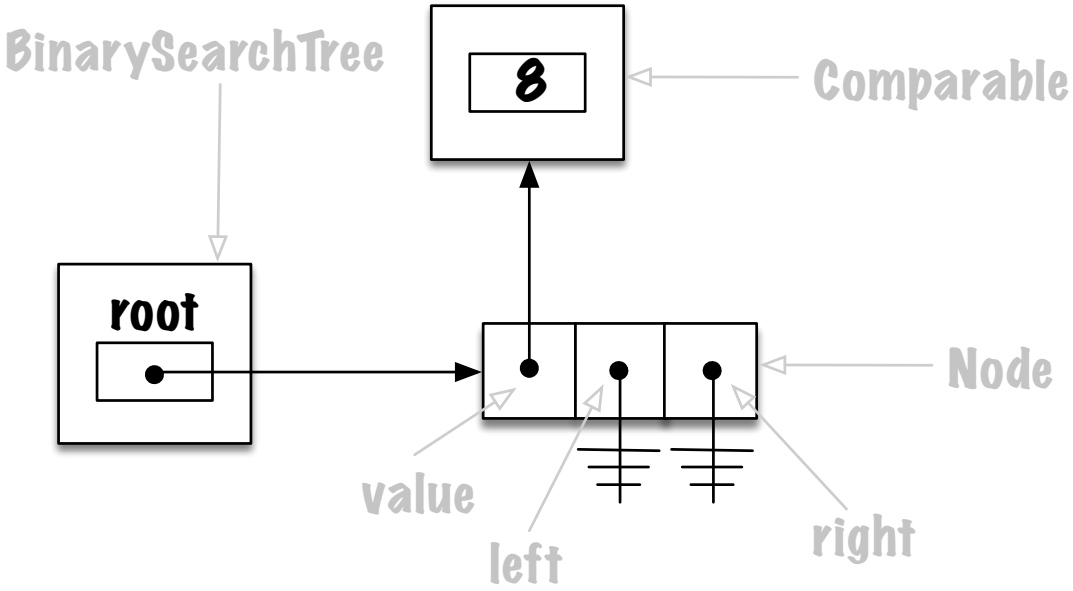
```
public class BinarySearchTree<E extends Comparable<E>> {

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

}
```

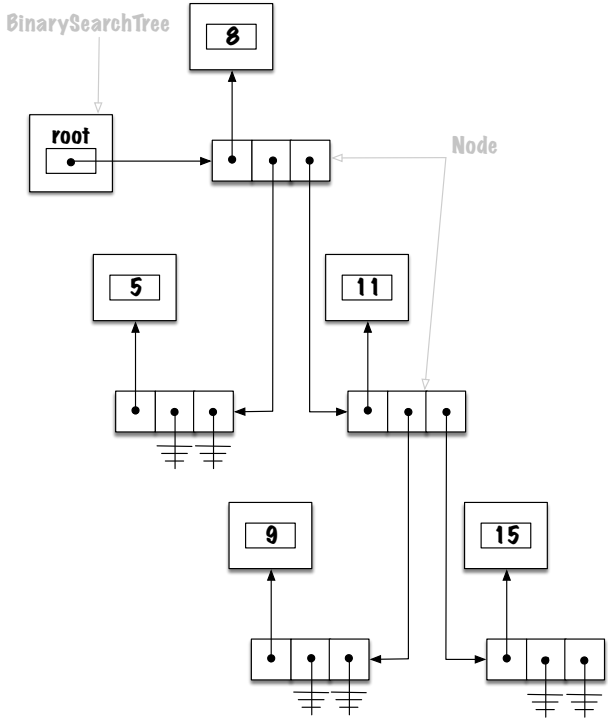
Implémentation

Quelles sont les **variables d'instance** de **BinarySearchTree** ?

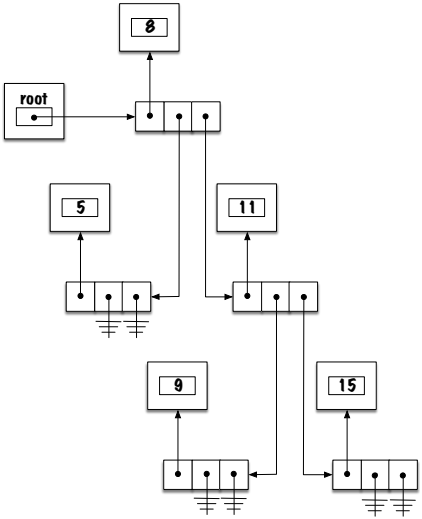
```
public class BinarySearchTree<E extends Comparable<E>> {

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

    private Node<E> root;
}
```

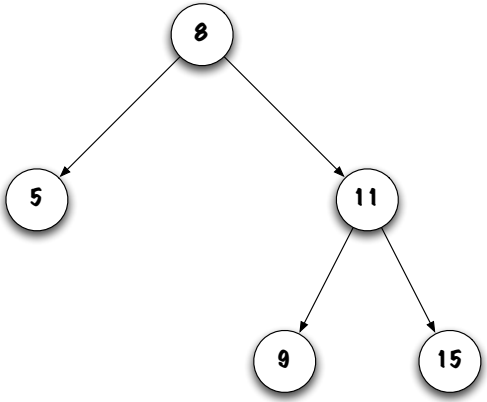
Diagramme de mémoire




BinarySearchTree
8
root
Node
5
11
9
15

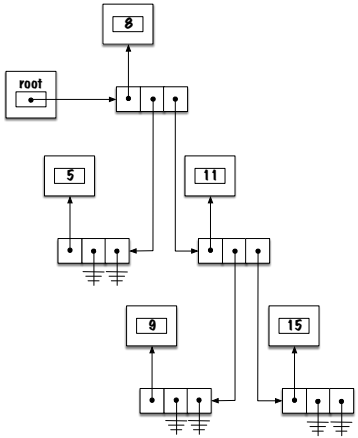
Représentations

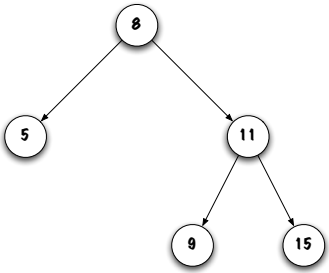




Observations

- 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 souvent la représentation plus abstraite de droite.



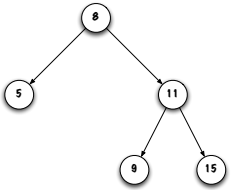


Méthodes

Méthodes

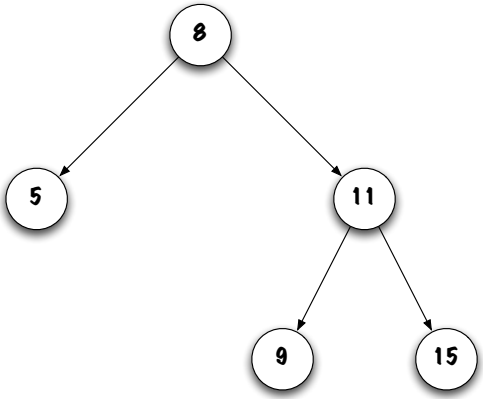
contains

boolean contains(E element)



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

boolean contains(E element)



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

public boolean contains(E element)

- La présentation suggère un **algorithme récursif**.
- Quelle sera la signature de la méthode ?

```
public boolean contains(E element) {

    if (element == null) {
        throw new NullPointerException();
    }
    return contains(root, element);
}
```

- 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 element)

Cas de base :

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

mais aussi

```
if (element.equals(current.value)) {
  result = true;
}
```

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

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

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

```java
private boolean contains(Node<E> current, E element) {

    boolean result;

    if (current == null) {
        result = false;
    } else {
        int test = element.compareTo(current.value);
        if (test == 0) {
            result = true;
        } else if (test < 0) {
            result = contains(current.left, element);
        } else {
            result = contains(current.right, element);
        }
    }

    return result;
}
```

public boolean contains(E element) (prise 2)

- Est-ce que la méthode **boolean contains(E element)** 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 element) (prise 2)

```java
public boolean contains2(E element) {

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

Traverser un arbre

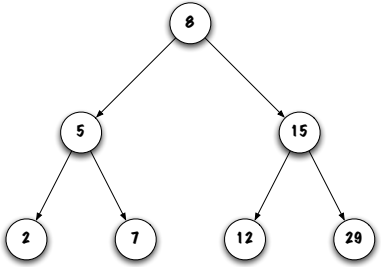
Traverser un arbre

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** ?

Traverser l'arbre

Pré-ordre : racine, gauche, droit ;

Symmétrique : gauche, racine, droit ;

Post-ordre : gauche, droit, racine.

Traverser l'arbre

```
private void visit(Node<E> current) {
    System.out.print(current.value);
}
```

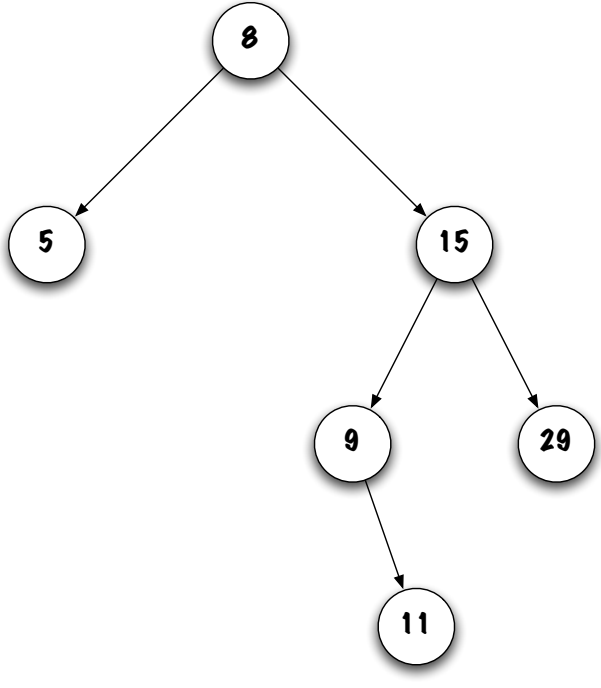
```
public void preOrder() {
    preOrder(root);
}

public void inOrder() {
    inOrder(root);
}

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

Traverser un arbre

Préfixe

Préfixe

```
private void preOrder(Node<E> current) {

    if (current != null) {

        visit(current);
        preOrder(current.left);
        preOrder(current.right);

    }

}
```

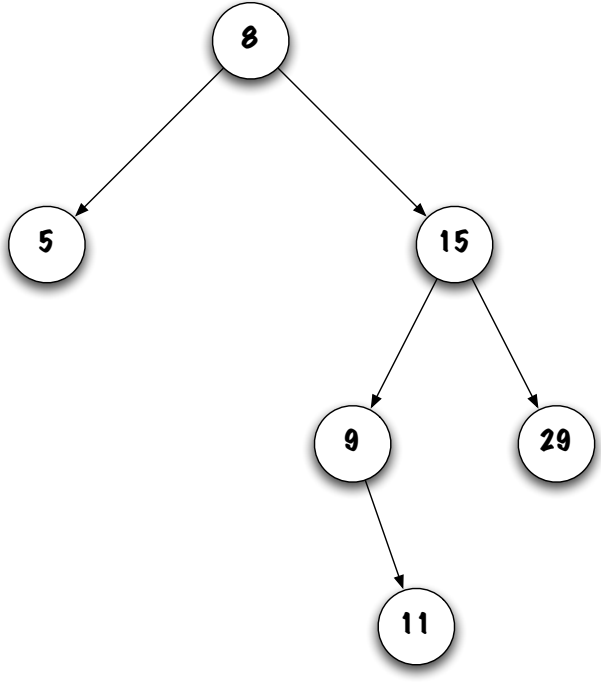

8
5
15
9
29
11

Traverser un arbre

Infixe

Infixe

```
private void inOrder(Node<E> current) {

    if (current != null) {

        inOrder(current.left);
        visit(current);
        inOrder(current.right);

    }

}
```

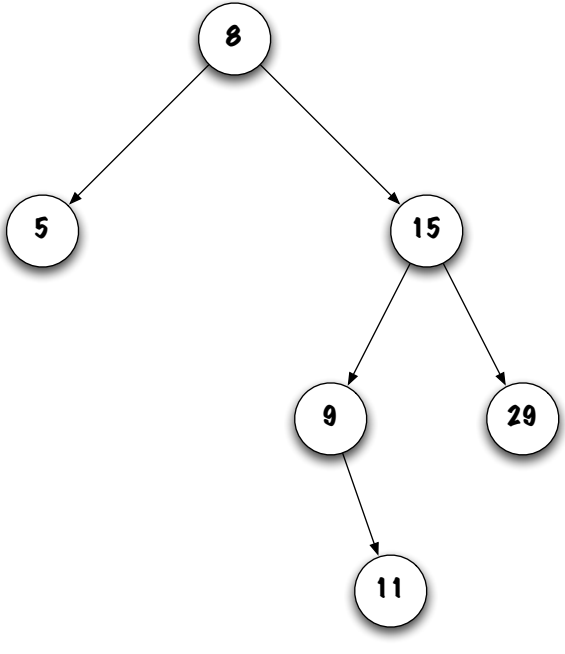

8
5
15
9
29
11

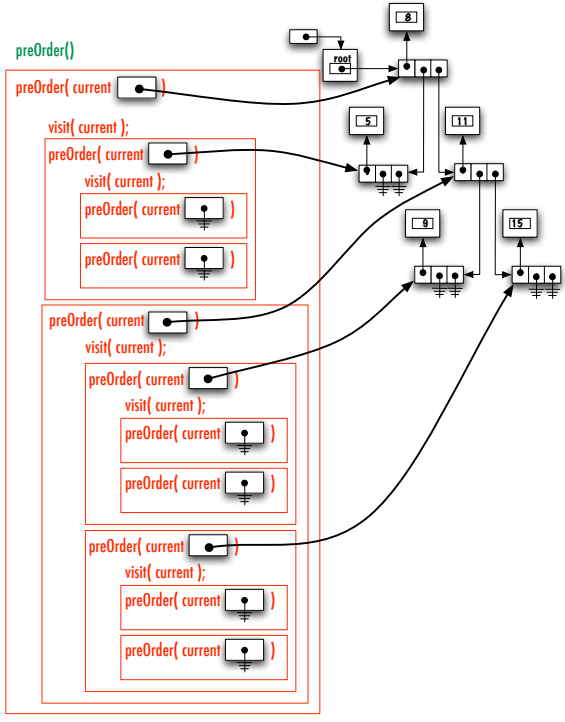
Traverser un arbre

Postfixe

Postfixe

```
private void postOrder(Node<E> current) {

  if (current != null) {

    postOrder(current.left);
    postOrder(current.right);
    visit(current);

  }

}
```


8
5
15
9
29
11


preOrder()
preOrder(current)
visit(current);
preOrder(current)
visit(current);
preOrder(current)
preOrder(current)
preOrder(current)
visit(current);
preOrder(current)
visit(current);
preOrder(current)
preOrder(current)
preOrder(current)
visit(current);
preOrder(current)
preOrder(current)
root
8
5
11
9
15

Observations

- Les méthodes qui parcourent **un seul chemin**, de la racine à une feuille par exemple, sont faciles à implémenter **sans appel récursif**, voir **contains** ;
- Les méthodes visitant **plusieurs sous-arbres** sont souvent plus facilement implémenter à l'aide de la **récursivité**.

Prologue

Résumé

- Un **arbre binaire de recherche** est un arbre binaire dont chaque noeud vérifie les deux propriétés suivantes :
  - Tous les noeuds de son sous-arbre **gauche** ont des valeurs **plus petites** que celle de ce noeud ou son sous-arbre gauche est vide ;
  - Tous les noeuds de son sous-arbre **droit** ont des valeurs **plus grandes** que celle de ce noeud ou son sous-arbre droit est vide.
- Implémentations à l'aide d'**éléments chaînés**.

Prochain module

- **Arbres binaires de recherche** : retrait d'un élément.

References I


E. B. Koffman and Wolfgang P. A. T.
Data Structures : Abstraction and Design Using Java.
John Wiley & Sons, 3e edition, 2016.

Marcel **Turcotte**

Marcel.Turcotte@uOttawa.ca

École de **science informatique** et de génie électrique (SIGE)

Université d'Ottawa