ITI 1121. Introduction to Computing II

Binary search tree: concept

by

Marcel Turcotte

Version March 27, 2020

Preamble

Preamble

Overview

Overview

Binary search tree: concept

We begin with an overview of the applications of trees in computing: to represent hierarchical data, for compression, and efficient access to elements. We examine the linked implementation of trees. We pay particular attention to binary search trees.

General objective:

- This week you will be able to design and modify computer programs based on the concept of a binary search tree.

Preamble

Learning objectives

Learning objectives

- **Name** applications of binary search trees.
- **Describe** the essential properties of binary search trees.

Readings:

- Pages 257-268 and 282-296 of E. Koffman and P. Wolfgang.

Preamble

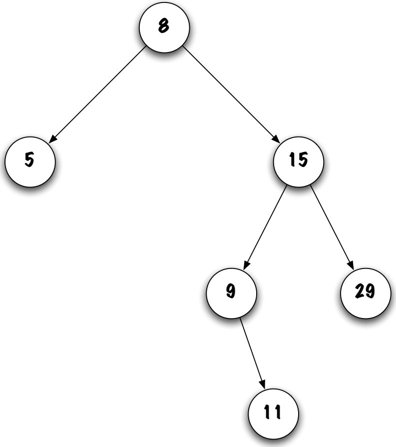
Plan

Plan

Theory

Definition

A **binary tree** is a **hierarchical** data structure such that each **node** stores one **value** and has at most two children, called **left** and **right**.



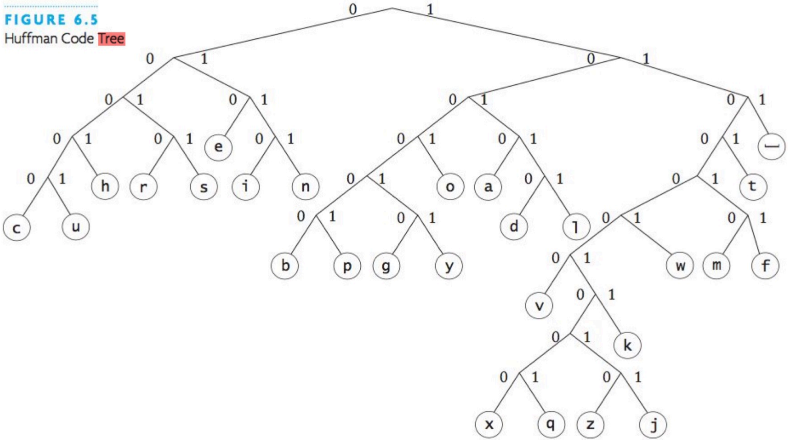
Theory

Applications

Applications (general trees)

- Represent hierarchical information such as **hierarchical file systems** (HFS) (directories and subdirectories), programs (**parse tree**);
- **Huffman trees** are used to compress information (files);
- The binary tree is an efficient data structure for implementing **abstract data types** such as heaps, priority queues, associative structures and sets.

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).

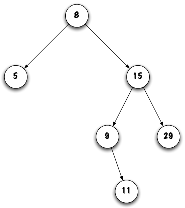
```
10001010011111010101010001010111010100011
  g    o   _  e   a    g   l   e    s
```

Source: [1] Figure 6.5.

Theory

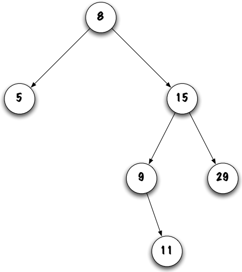
Definitions

Definitions



- All nodes have **only one parent**, except for one node that has no parent, and is called the **root** (this is the node at the very top of the diagram);
- Each node has either **0**, **1** or **2** children;
- The childless nodes are the **leaves** of the tree (or outer nodes);
- The links between the nodes are the **branches** of the tree.

Definitions



- A node and its descendants are a **sub-tree**;
- The **size** of a tree is the number of nodes in the tree. An **empty tree** has a size 0;
- **Since** we will only deal with binary trees, I will sometimes use the term tree to refer to a binary tree.

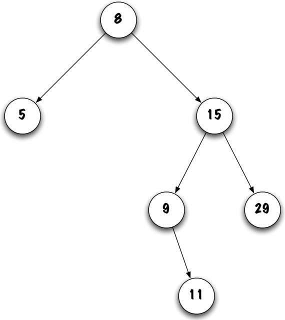
Definitions

We can give a **recursive** definition:

- A binary tree is **empty**, or;
- A binary tree consists of a **value** and **two subtrees** (left and right).

Definitions

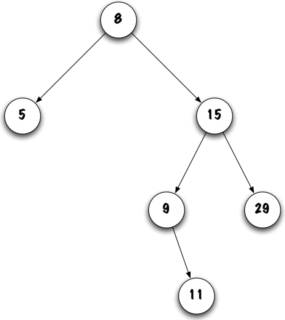
The **node depth** represents the number of links you have to follow from the root in order to access that node. The root is the most accessible node.



What's the depth of the root? The root is always at depth 0.

Definitions

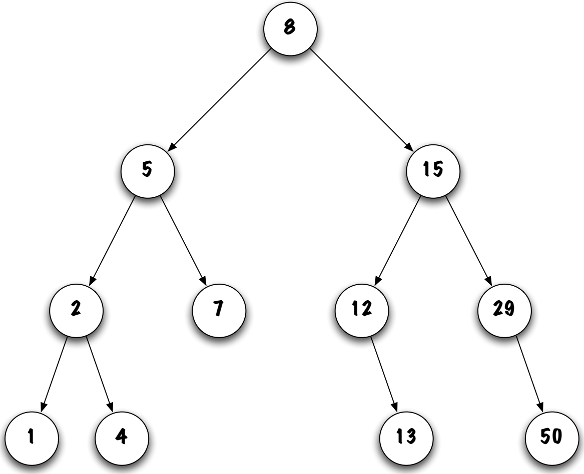
The **node depth** represents the number of links you have to follow from the root in order to access that node. The root is the most accessible node.



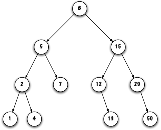
The **depth of a tree** is the maximum depth of a node in the tree.

Discussion

- All the trees shown here have **a property in common**. What is it?



Definition



A **binary search tree** is a binary tree whose nodes satisfy the following properties:

- all the nodes of its **left subtree** have **smaller** values than this node (or the left subtree is empty);
- all the nodes of its **right subtree** have **greater** values than this node (or this subtree is empty).

Corollary: the values are unique.

Implementation

Implementation

- **How** are we going to implement this class?
- **Indeed**, we'll use a "nested" and "static" class, **Node**.
- What are its **instance variables**?
  - The instance variables are: **value**, **left** and **right**;
- What is the type of these variables?
  - **value** is of type **Comparable**, **left** and **right** are of type **Node**.

Implementation

A **static nested class** to save a value and create the structure of the tree.

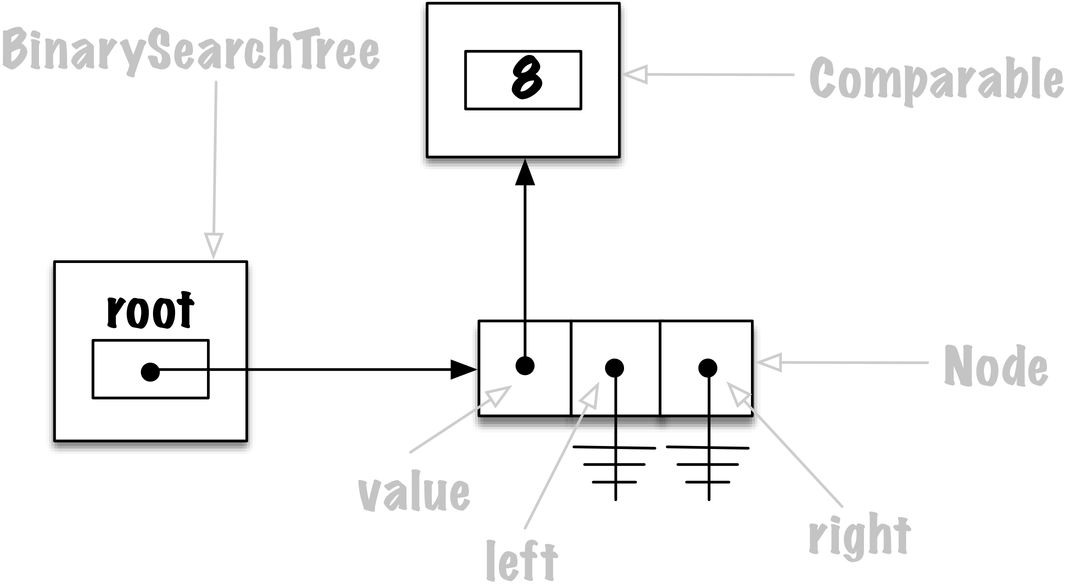
```
public class BinarySearchTree<E extends Comparable<E>> {

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

}
```

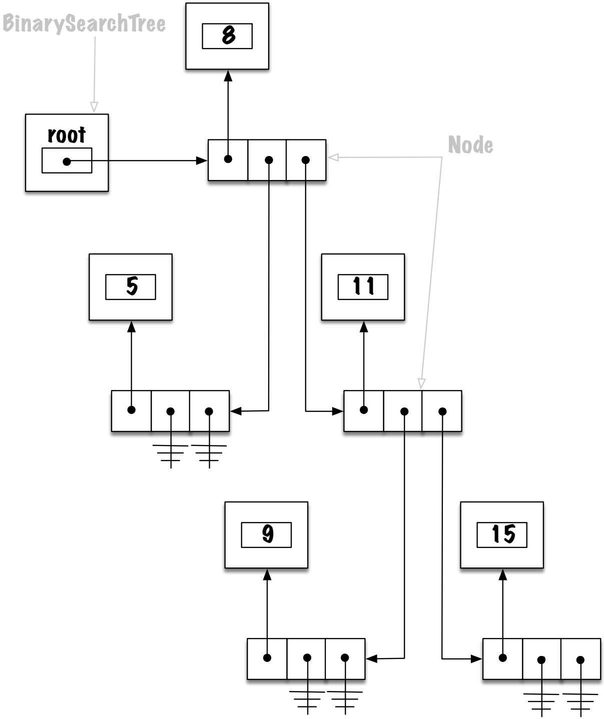
Implementation

What are the **instance variables** of the **BinarySearchTree**?

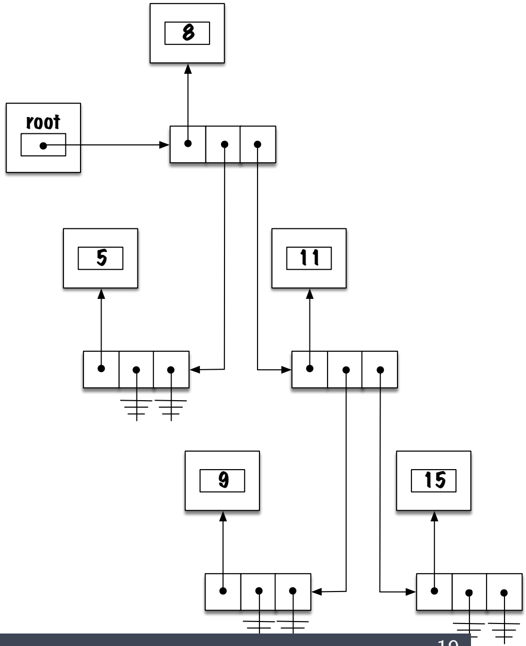
```
public class BinarySearchTree<E extends Comparable<E>> {

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

    private Node<E> root;
}
```

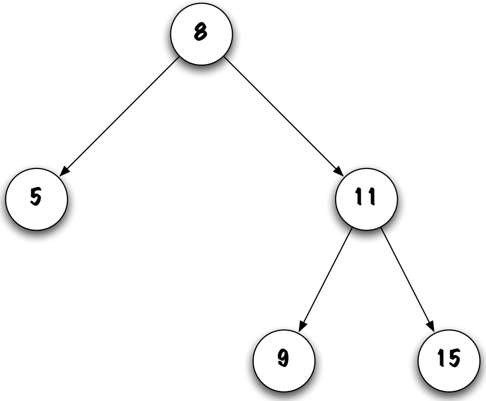
Memory diagram




BinarySearchTree
8
root
Node
5
11
9
15

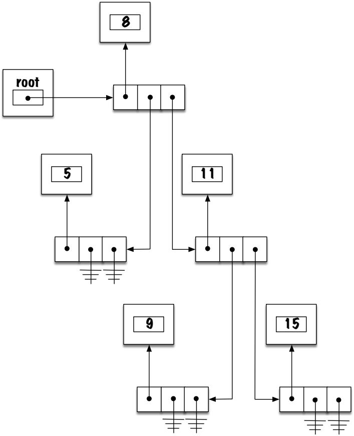
Representations

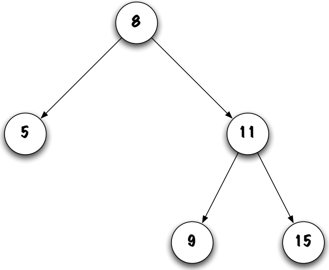




Observations

- A **leaf** is a node such that its two descendants are both **null**.
- The variable **root** can be **null**, so the tree is empty and of size 0.
- For the sake of simplicity, I will often use the more abstract representation on the right.



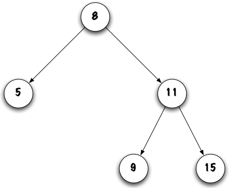


Methods

Methods

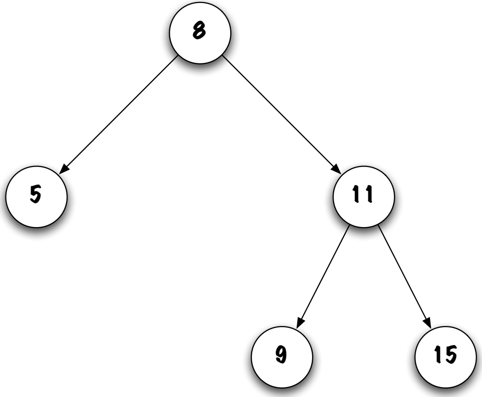
contains

boolean contains(E element)



1. Empty tree? **element** is not found;
2. The local root contains **element**? **element** is found; Otherwise? Where are we looking for?
3. If **element** is smaller than the value stored in the current node? Look for **element** in the left subtree;
4. Otherwise (**element** is necessarily greater than the value in the current node.)? Look for **element** in the right subtree.

boolean contains(E element)



Exercises: apply the algorithm to find the values **8**, **9** and **7** in the tree above.

public boolean contains(E element)

- The presentation suggests a **recursive algorithm**.
- What will be the signature of the method?

```
public boolean contains(E element) {

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

- Like the recursive processing of linked lists, our methods will have **two parts**, a **public** part, and a **private** part whose signature has a parameter of type **Node**.

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

Base case:

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

but also

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

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

General case: . Search left or right (recursively).

```
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) (take 2)

- Is the method **boolean contains(E element)** necessarily recursive?
  - No.

Develop a strategy.

1. Use a local variable **current** of type **Node**;
2. Initialize the variable to designate the root node of the tree;
3. If **current** is **null** then the value is not found, stop;
4. If **current.value** is the value sought, stop;
5. If the sought value is smaller than **current = current.left**, goto 3;
6. Else **current = current.right**, goto 3.

public boolean contains(E element) (take 2)

```
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;
}
```

Traversing a tree

Traversing a tree

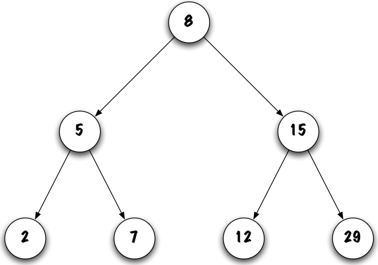
Sometimes one has to **traverse** the tree in order to **visit** all of its nodes.

When **visiting** a node, you perform certain operations on the node.

- **Pre-order** traversal: **visit the root**, traverse the left subtree, traverse the right subtree;
- **In-order** (infix, symmetrical) traversal: traverse the left subtree, **visit the root**, traverse the right subtree;
- **Post-order** (suffix) traversal: traverse the left subtree, traverse the right subtree, **visit the root**;

Exercises

The simplest operation is to display the value stored in the node.



- Give the result displayed for each strategy, **pre-order**, **in-order** and **post-order**.
- What strategy is displaying the data in **ascending order**?

Traversing a tree

Pre-order: **root**, left, right;

In-order: left, **root**, right;

Post-order: left, right, **root**.

Traversing a tree

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

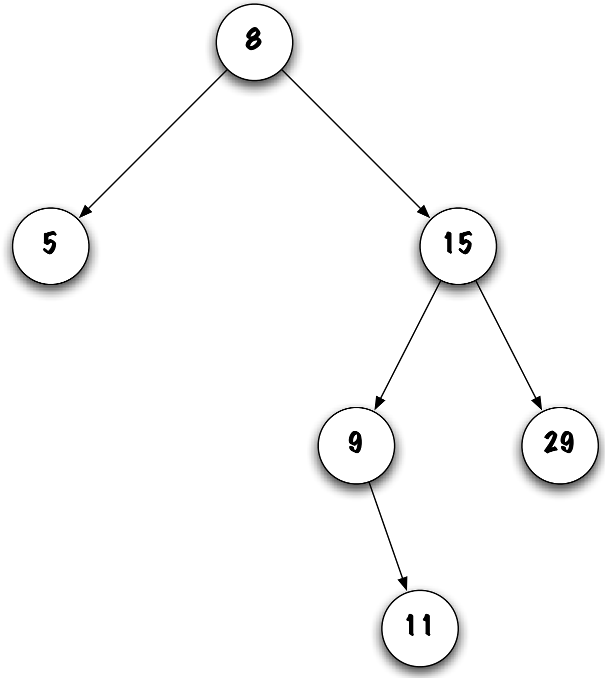
```java
public void preOrder() {
    preOrder(root);
}

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

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

Traversing a tree

Pre-order

Pre-order

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

    if (current != null) {

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

    }

}
```

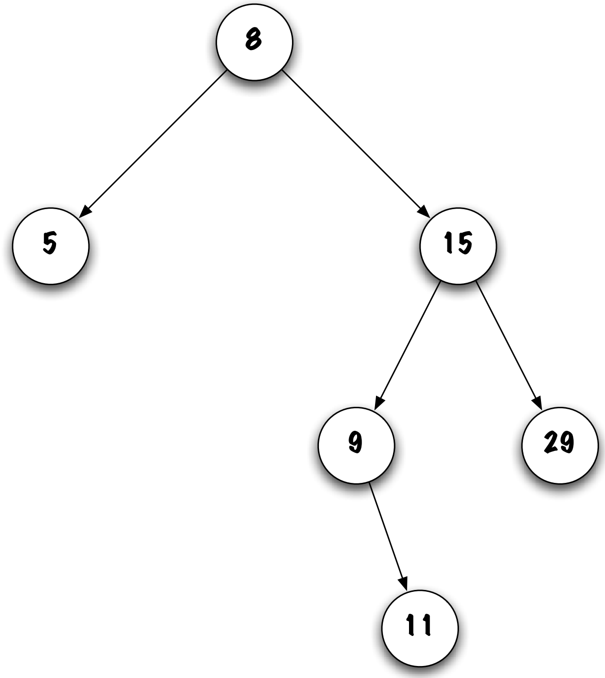

8
5
15
9
29
11

Traversing a tree

In-order

Infixe

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

    if (current != null) {

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

    }

}
```

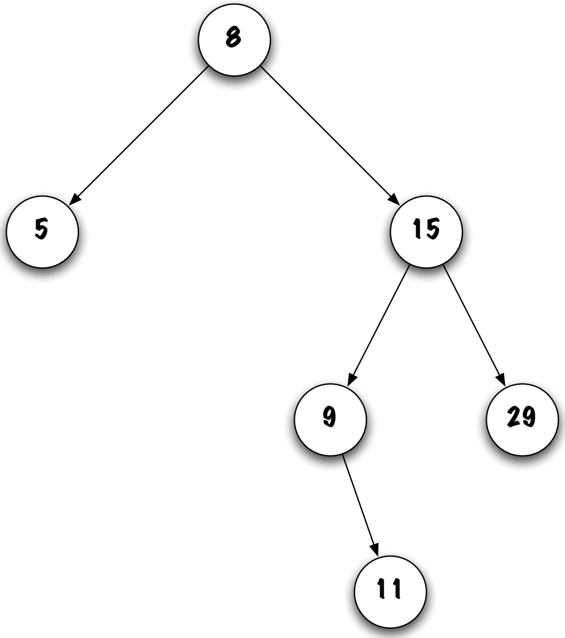

8
5
15
9
29
11

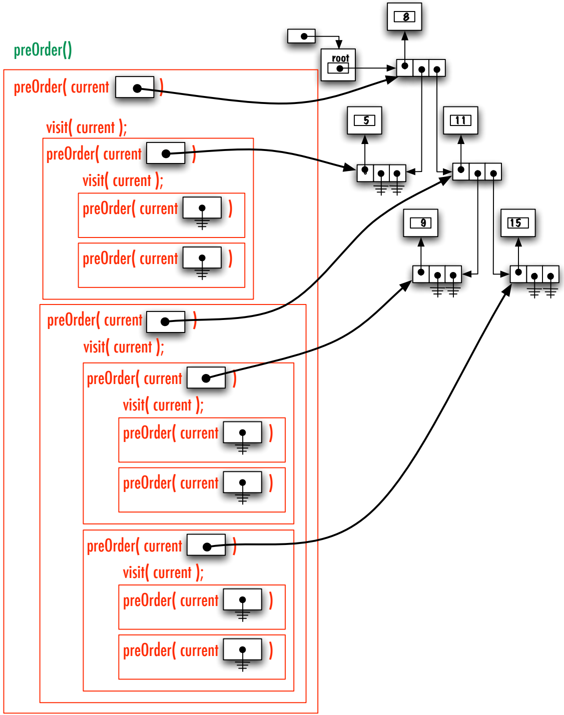
Traversing a tree

Post-order

Post-order

```java
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

- Methods that follow **only one path**, from the root to a leaf, for example, are easy to implement **without recursive calls**, see **contains**;
- Methods that visit **several subtrees** are often more easily implemented using **recursivity**.

Prologue

Summary

- A **binary search tree** is a binary tree where each node satisfies the following two properties:
  - All the nodes in its **left** subtree have **smaller** values than this node's or its left subtree is empty;
  - All the nodes of its **right** subtree have **greater** values than this node or its right subtree is empty.
- Implemented using **linked elements**.

Next module

- **Binary search trees** : removal of an element.

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

School of Electrical Engineering and **Computer Science** (EECS)
University of Ottawa