CM1602 : Data Structures and Algorithms for AI

7. Trees

Lecture 7 | R. Sivaraman

ROBERT GORDON UNIVERSITY ABERDEEN
TEF Gold
INFORMATICS INSTITUTE OF TECHNOLOGY

MODULE CONTENT

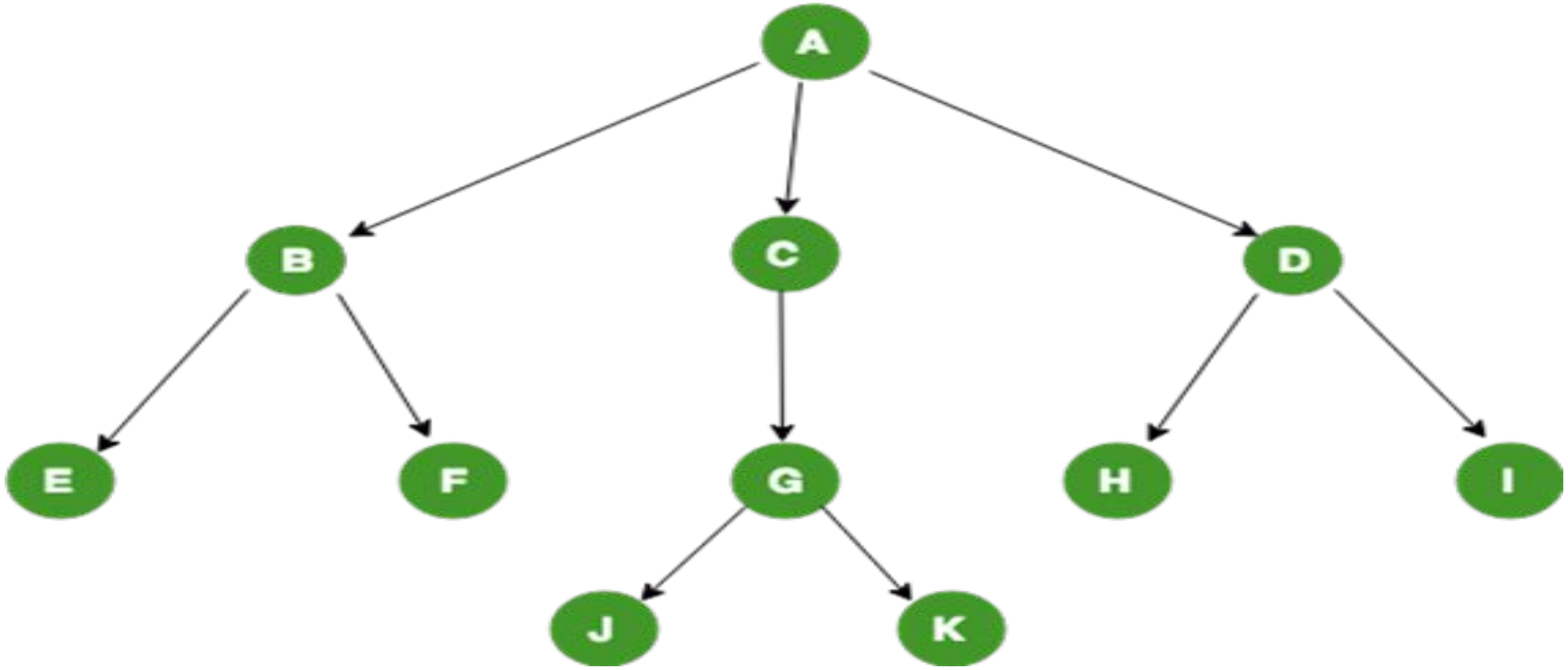
Lecture	Topic
Lecture 01	Introduction to Fundamentals of Algorithms
Lecture 02	Analysis of Algorithms
Lecture 03	Array and Linked Lists
Lecture 04	Stack
Lecture 05	Queue
Lecture 06	Searching algorithms and Sorting algorithms
Lecture 07	Trees
Lecture 08	Maps, Sets, and Lists
Lecture 09	Graph algorithms

Learning Outcomes

- LO1 : Describe the fundamental concepts of algorithms and data structures.
- LO3 : Apply appropriate data structures given a real-world problem to meet requirements of programming language APIs.
- On completion of this lecture, students are expected to be able to:
    - Describe Trees and when Trees are used.
    - Implement Binary Search Trees.

Trees

- Another data structure commonly implemented using linked nodes.

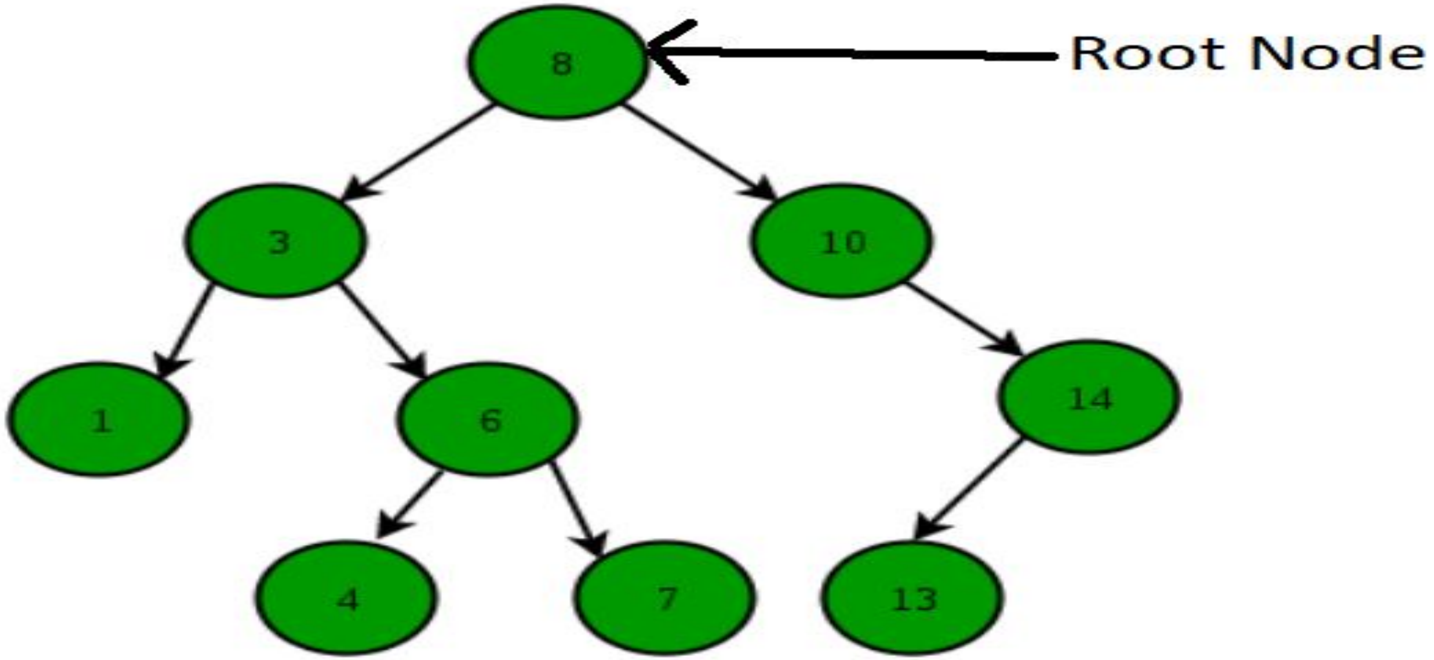


Binary Search Tree

- Binary Search Tree is a tree where a node can have maximum of 2 children.
- **Binary Search Tree** is a node-based binary tree data structure which has the following properties:
  - The left subtree of a node contains only nodes with keys lesser than the node’s key.
  - The right subtree of a node contains only nodes with keys greater than the node’s key.
  - The left and right subtree each must also be a binary search tree.

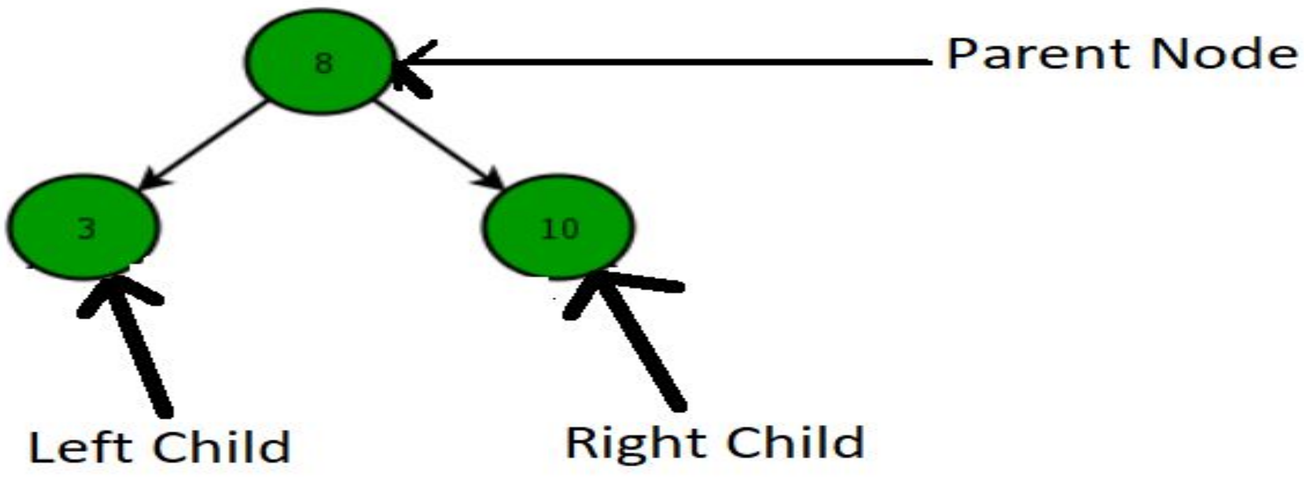
Root Node

- A tree will have only one root Node.
- The root of the whole tree will be the only node which does not have a parent



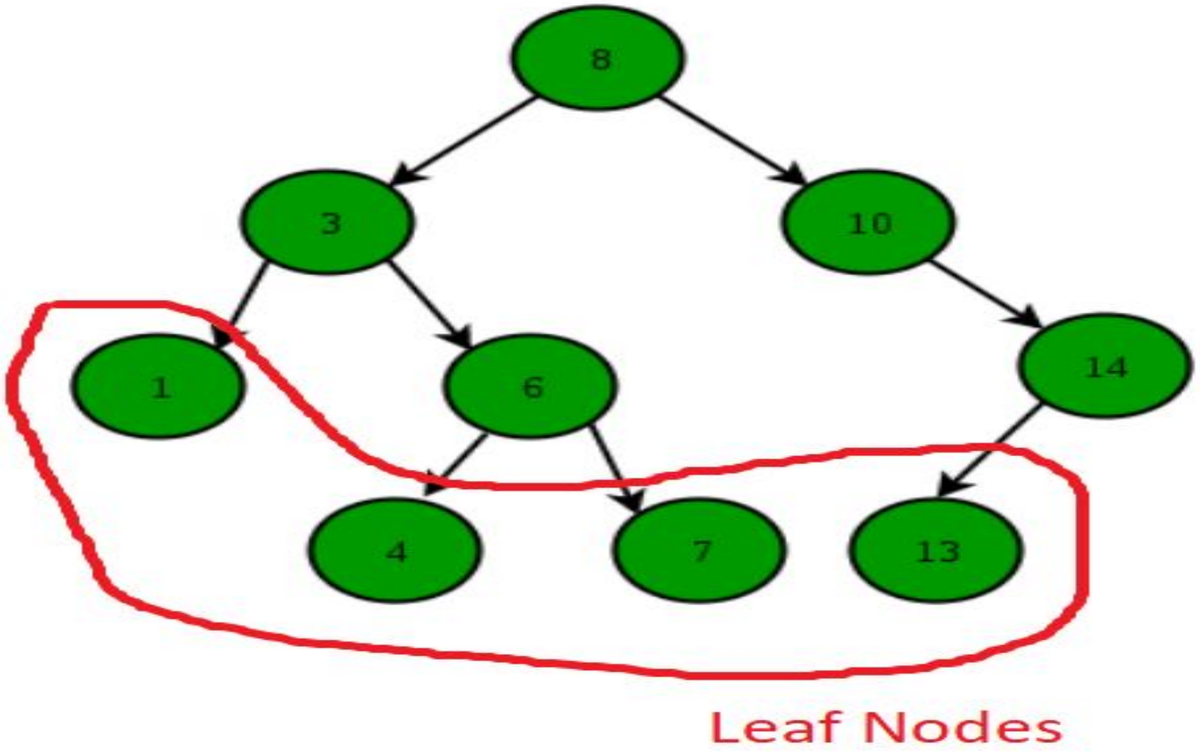
Parent and Child Nodes

- A node can have maximum of 2 children.
- Left child is smaller than the parent.
- Right child is larger than the parent.

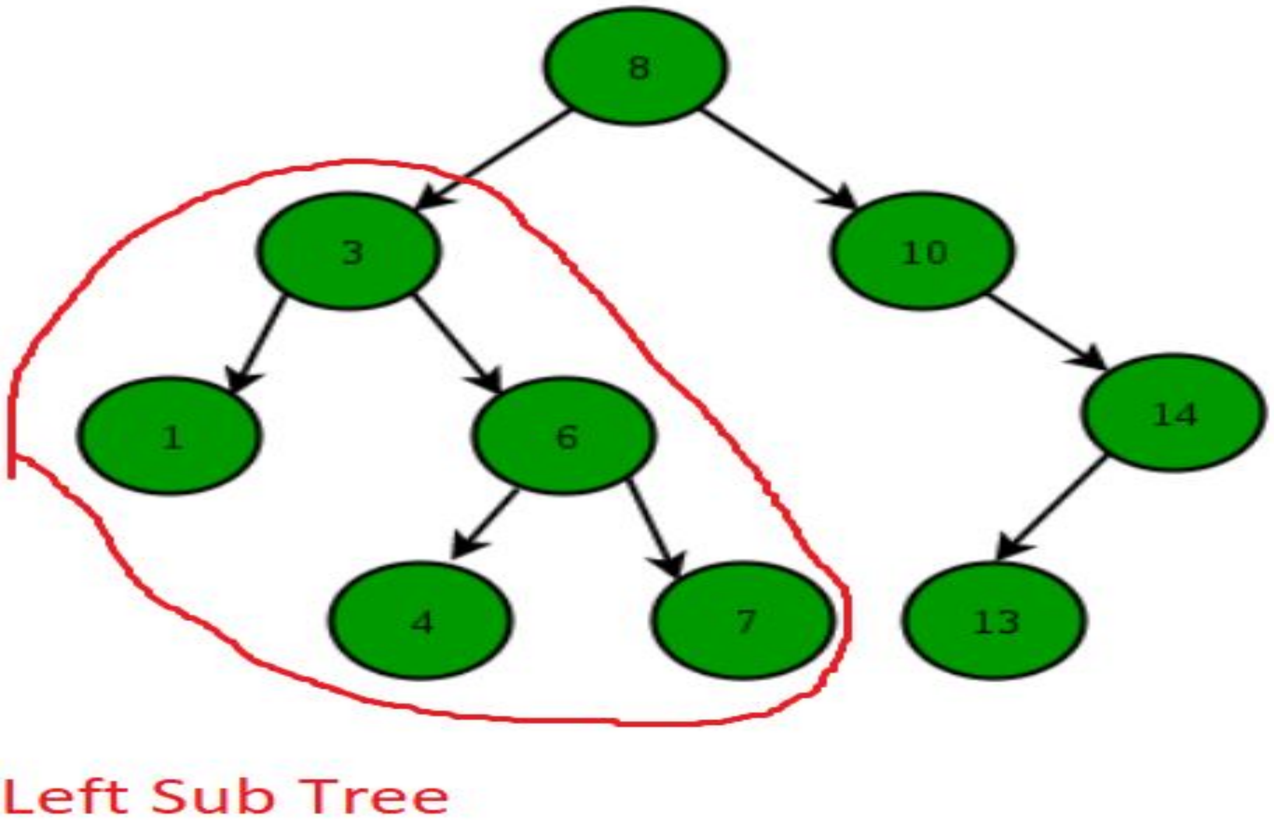


Leaf Nodes

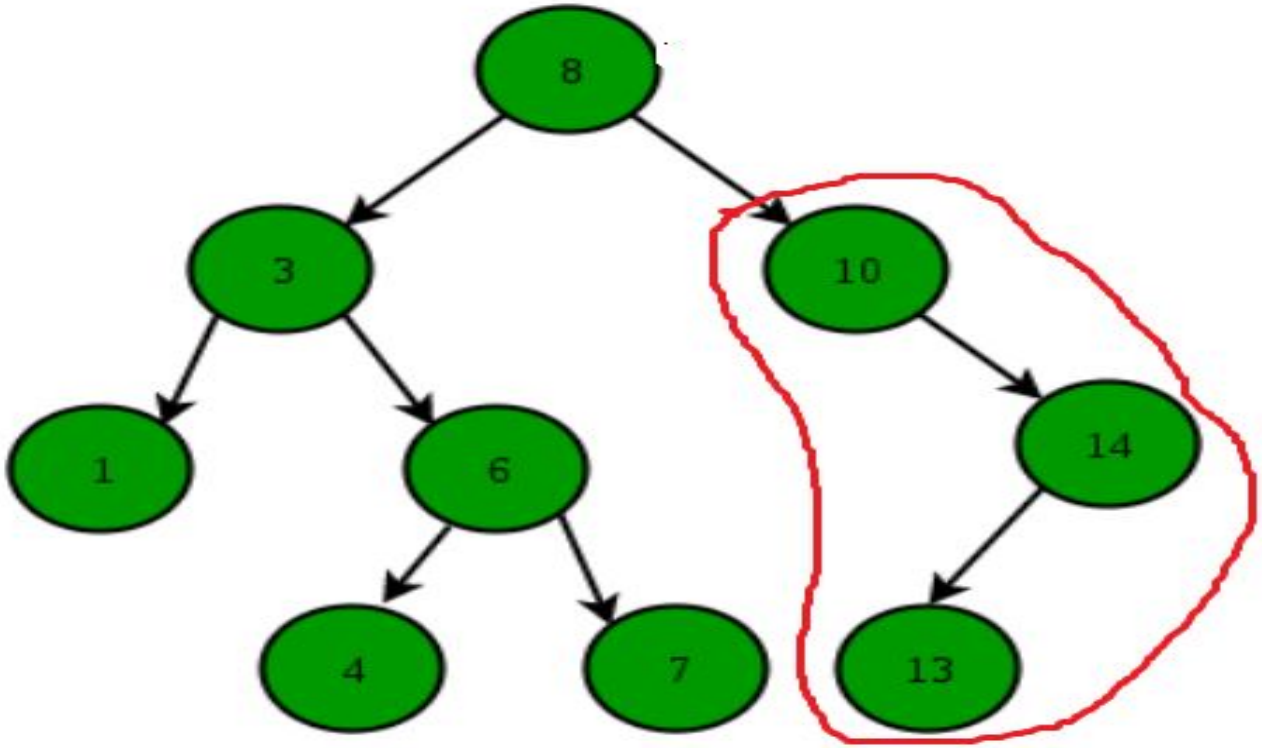
- Leaf Nodes do not have any children.



Left sub Tree



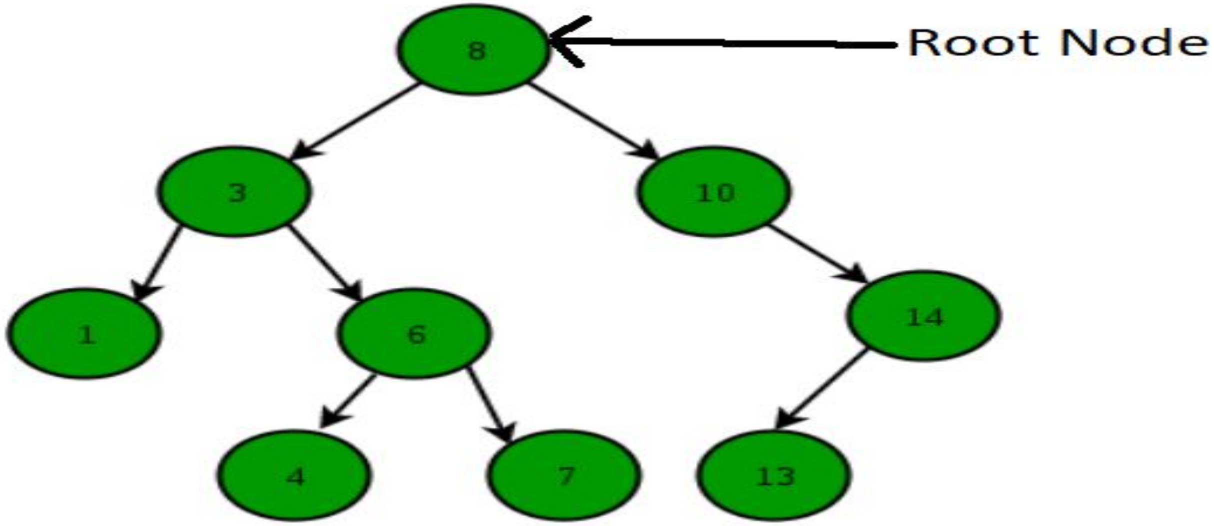
Right Sub Tree



Right Sub Tree

Depth of Tree

- Maximum number of edges/arc from the root node to the leaf node of the **tree** is called as the **Depth.**
- What is the depth of the tree given below? - 3



Binary Search Tree

- There are a number of other operations we often want to do with data structures
- Output (in increasing order – In Order Traversal)
- Insertion
- Deletion

Binary Search Tree - Node

```java
public class BinarySearchTree {

    public class TreeNode{
        public int data;
        public TreeNode leftChild, rightChild, parent;

        public TreeNode(int d){
            data = d;
            leftChild = null;
            rightChild = null;
            parent = null;
        }


    }
```

Binary Search Tree

- Binary Search Tree Class have to store the root node.
- Root node is an object of Tree Node Class

```
private TreeNode root;
```

Search Method

- Start from the root node
- If the value is smaller than current node, move to the left.
- If the value is larger than current node, move to the right.
- If the value is equal to the current node, value is found.

Search Method

```java
public TreeNode find(int findMe){
    TreeNode n = root;
    while(n != null){
        if(n.data == findMe)
            return n;
        if(n.data < findMe)
            n = n.rightChild;
        else
            n = n.leftChild;
    }
    return null;
}
```

Output

- To Print in order:
  - Left.
  - Center – current.
  - Right.

Output

```java
public void output(){
    TreeNode n = root;
    inorderRec(n);
}

private void inorderRec(TreeNode node)
{
    if (node != null) {
        inorderRec(node.leftChild);
        System.out.println(node.data);
        inorderRec(node.rightChild);
    }
}
```

Insert Method

- If the tree is empty, the first value inserted will be the Root Node.
- If the tree is not empty, the parent node have to be found (Similar to Search method).
- If the value is smaller than parent, it will be the left child.
- If the value is larger than parent, it will be the right child.

Insert Method

```java
    public void insert(int value){
        root = insertRec(root, value);
    }
    private TreeNode insertRec(TreeNode node, int value)
    {
        /* If the tree is empty,
           return a new node */
        if (node == null)
        {
            node = new TreeNode(value);
            return node;
        }
        /* Otherwise, recur down the tree */
        if (value < node.data)
            node.leftChild = insertRec(node.leftChild, value);
        else if (value > node.data)
            node.rightChild = insertRec(node.rightChild, value);
        /* return the (unchanged) node pointer */
        return node;
    }
```

Delete Method

Fide the Element to be deleted (Similar to search method).

If:

1. Leaf Node – Delete the node.
2. Node have 1 child – Delete the node and then connect the child node (of the deleted node) with parent node (of the deleted node).
3. Nod have 2 children - Delete the node and the replace it with the Right most node of the Left Sub Tree

Delete Method

```java
public void remove(int value){
    root = deleteRec(root, value);
}

private TreeNode deleteRec(TreeNode node, int value)
{
    /* Base Case: If the tree is empty */
    if (node == null)
        return node;

    /* Otherwise, recur down the tree */
    if (value < node.data)
        node.leftChild = deleteRec(node.leftChild, value);
    else if (value > node.data)
        node.rightChild = deleteRec(node.rightChild, value);

```

Delete Method

```
        // if key is same as root's
        // key, then This is the
        // node to be deleted
        else {
            // node with only one child or no child
            if (node.leftChild == null)
                return node.rightChild;
            else if (node.rightChild == null)
                return node.leftChild;

            // node with two children: Get the inorder
            // successor (smallest in the right subtree)
            node.data = minValue(node.rightChild);

            // Delete the inorder successor
            node.rightChild = deleteRec(node.rightChild, node.data);
        }

        return node;
    }
```

Delete Method

```
int minValue(TreeNode node)
{
    int minv = node.data;
    while (node.leftChild != null)
    {
        minv = node.leftChild.data;
        node = node.leftChild;
    }
    return minv;
}
```