CHAPTER III
TREE & BINARY TREE

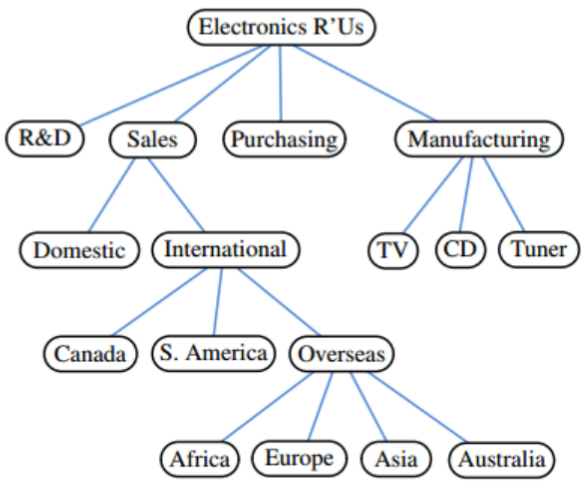
3.1 Learning Objectives

1. Students can understand and are able to implement binary trees
2. Students can understand and are able to implement binary search trees

3.2 Theory

3.2.1 Tree

A *tree* is an abstract data type (ADT) that stores elements hierarchically. With the exception of the top element, each element in a tree has a parent element and zero or more children elements. A tree is usually represented using ovals or rectangles connected by lines (as shown below). We typically call the top element the root of the tree, but it is drawn as the highest element, with the other elements being connected below (just the opposite of a botanical tree).



Formally, we define a tree T as a set of nodes storing elements such that the nodes have a parent-child relationship that satisfies the following properties:

- If T is nonempty, it has a special node, called the root of T, that has no parent.
- Each node v of T different from the root has a unique parent node w; every node with parent w is a child of w.

Note that according to our definition, a tree can be empty, meaning that it does not have any nodes. This convention also allows us to define a tree recursively such that a tree T is either empty or consists of a node r, called the root of T, and a (possibly empty) set of subtrees whose roots are the children of r.

3.2.2 Binary Tree

A binary tree is an ordered tree that satisfies the following rules:

- Each node has at most two children.
- Each child node is labelled as being either a left child or a right child.
- A left child precedes a right child in the order of children of a node.

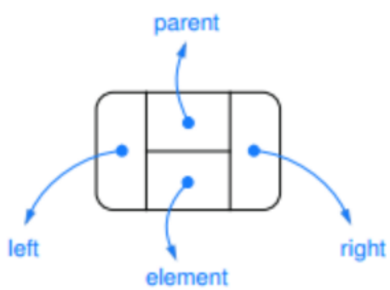
The subtree rooted at left or right child of an internal node v is called a left subtree or right subtree, respectively, of v. A binary tree is proper if each node has either zero or two children. Some people also refer to such trees as being full binary trees. Thus, in a proper binary tree, every internal node has exactly two children. A binary tree that is not proper is improper.

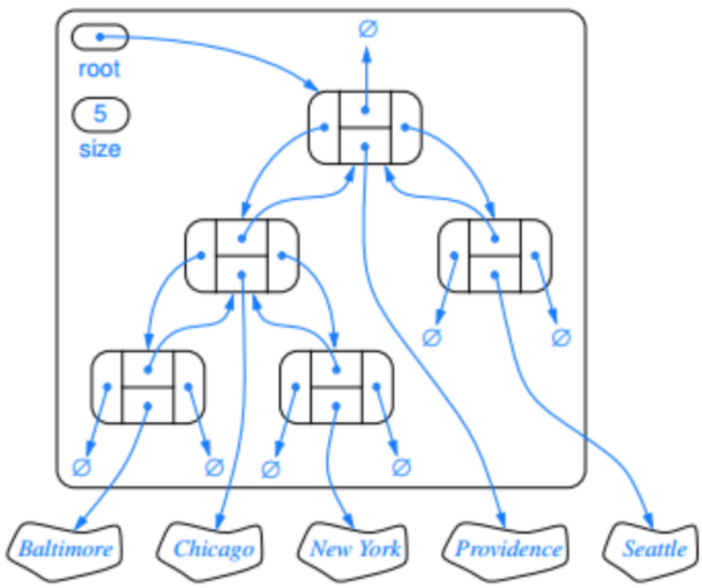
As an ADT, the binary tree is a specialization of a tree that supports the following methods or functions:

- left(p) : returns the position of the left child of p (or null if p has no left child).
- right(p) : returns the position of the right child of p (or null if p has no right child).
- sibling(p): returns the position of the sibling of p (or null if p has no sibling).

One form of binary tree implementation is to use a linked structure, where a node has a memory address to an element stored at position p and has a memory address to another node bound as children or parents of p.

- If p is the root of T, then the parent of p is null.
- If p does not have a left child (or right child), the children connected with p are null
- The tree itself has an instance variable that holds the memory address to the node root and a variable called size which represents the total number of nodes in the tree T.



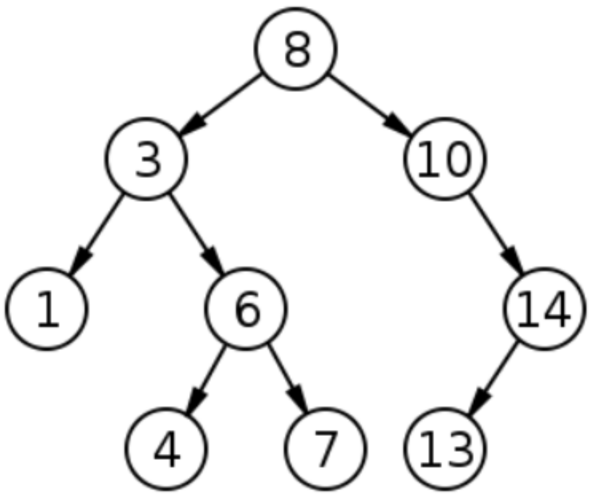


In order to update binary tree elements, the following functions must be supported:

- addNode(node): Add a node to a binary tree. If the tree is empty, then the node become root.
- insertNode() : Add/insert a new node to the existing binary tree.
- deleteNode(node): Remove a node from a binary tree.

There are three methods for printing all elements in a binary tree. The printing order of the elements is based on the order in which the nodes are processed.

- preOrder: The root node will be printed first, then the left branch tree, and finally the right branch tree
- postOder: The left branch tree will be printed first, then the right branch tree and finally the root node
- inOrder: Left branch tree will print first, then node root and finally the right branch tree



There is a binary tree example above, the following is the result of printing the binary tree elements.

- PREODER (Root – Left – Right) →8 3 1 6 4 7 10 14 13
- POSTODER (Left – Right – Root) →1 4 7 6 3 13 14 10 8
- INODER (Left – Root – Right) →1 3 4 6 7 8 10 13 14

3.2.3 Binary Search Tree

A *binary search tree (BST)* or commonly known as a sorted binary tree is a data structure that stores items (such as numbers, names, etc.) in memory. BST speeds up the process of searching, adding, and deleting data. In order to be able to update elements binary search tree, the following functions must be supported:

- addNode(node): Add a node to a binary search tree. If the tree is empty, then the node become root.
- insertNode(): Add/insert a new node to the existing binary search tree .
- deleteNode(node): Remove a node from a binary search tree.
- searchValue(root, value): A static method to check if the given value exists in an existing binary search tree.

The following is an implementation of a binary search tree. First, create a Node class and save it as Node.java

```java
package binarysearchtree;

public class Node{
    public int value;         //element data
    public Node leftChild;    //pointer to leftChild node
    public Node rightChild;   //pointer to rightChild node

    Node(int value){          //constructor
        this.value = value;   //initialize element data of node
    }

    public int getValue(){    //function getValue
        return value;         //to get the value of element data
    }
}

```

Then, create a BinarySearchTree class that defines all the methods in the binary search tree and save it as BinarySearchTree.java.

```java
package binarysearchtree;

public class BinarySearchTree {
    public Node root;                       //ref to root node on tree

    public void addNode(Node node){ //method to add a node on the tree
        if(root == null){           //if the root is empty
            root = node;            //set node as the root of the tree
        }
        else{                       //if the root is not empty
            insertNode(root, node); //insert a node on the tree
        }                           //using function insertNode
    }

    public void insertNode(Node parent, Node node){ //method to insert a node on the tree
        if(parent.getValue() > node.getValue()){    //if the value of parent > node
            if(parent.leftChild == null){           //if the leftChild of parent is null
                parent.leftChild = node;            //set node as the leftChild of parent node
            }
            else{                                   //if the leftChild of parent is not null
                insertNode(parent.leftChild, node); //call function insertNode to
            }                                       //insert a node as a child of the leftChild of parent node
        }
        else{                                       //if the value of parent <= node
            if(parent.rightChild == null){          //if the rightChild of parent is null
                parent.rightChild = node;           //set node as the rightChild of parent node
            }
            else{                                   //if the rightChild of parent is not null
                insertNode(parent.rightChild, node);//call function insertNode to
            }                                       //insert a node as a child of the rightChild of parent node
        }
    }

    int minValue(Node root){                        //method to get the minimum value of tree
        int minv = root.value;                      //set the value of root of the tree as minv
        while (root.leftChild != null){             //repeat if the leftChild of root is not null
            minv = root.leftChild.value;            //update minv as the value of the leftChild of root
            root = root.leftChild;                  //update root as the leftChild of root
        }
        return minv;                                //return the minimum value of the tree
    }

    public void deleteNode(int value) {             //method to delete node of the tree based on value
        root = deleteFunc(root, value);             //using deleteFunc function
    }

    public Node deleteFunc(Node root, int value){   //function to delete node of the tree based on value
        if (root == null){                          //if the tree is empty
            return root;                            //return root
        }

        // Otherwise, recur down the tree
        if (value < root.value)                     //if the value < the value of root
            root.leftChild = deleteFunc(root.leftChild, value);
        else if (value > root.value)                //id the value > the value of root
            root.rightChild = deleteFunc(root.rightChild, value);

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

            // node with two children:
            // Get the inorder successor (smallest in the right subtree)
            root.value = minValue(root.rightChild); //set the min value of right subtree as the value of root

            // Delete the inorder successor
            root.rightChild = deleteFunc(root.rightChild, root.value);
        }
        return root;
    }
```

```java

    public static void preorderPrint(Node root) {
    //method to print all elements in a binary tree in preorder process
    //The the root node is printed first, then left subtree, and the last is right subtree
        if ( root != null ) { //if root is not null
            System.out.print( root.value + " " );   //Print the value root node
            preorderPrint( root.leftChild );        //Print the item in left subtree recursively
            preorderPrint( root.rightChild );       //Print the item in right subtree recursively
        }
    }

    public static void postorderPrint(Node root) {
    //method to print all elements in a binary tree in postorder process
    //The left subtree is printed first, then right subtree and the last is the root node
        if ( root != null ) { //if root is not null
            postorderPrint( root.leftChild );       //Print the item in left subtree recursively
            postorderPrint( root.rightChild );      //Print the item in right subtree recursively
            System.out.print( root.value + " " );   //Print the value root node
        }
    }

    public static void inorderPrint(Node root) {
    //method to print all elements in a binary tree in postorder process
    //The left subtree is printed first, then the root node, and the last is right subtree
        if ( root != null ) { //if root is not null
           inorderPrint( root.leftChild );          //Print the item in left subtree recursively
           System.out.print( root.value + " " );    //Print the item in right subtree recursively
           inorderPrint( root.rightChild );         // Print items di pohon cabang kanan
        }
    }

    public static boolean searchValue(Node root, int value){
    //method to searchValue in binary tree
        if(root == null){                          //if the tree is empty
            return false;                          //return false
        }
        else{ //if the tree is not empty
            if(root.getValue() == value){          //if the value of root = the value that we search
                return true;                       //return true
            }
            else if(root.getValue() > value){      //if the value of root > the value that we search
                return searchValue(root.leftChild, value); //search the value in the leftChild of root
            }
            else{                                  //if the value of root < the value that we search
                return searchValue(root.rightChild, value); //search the value in the rightChild of root
            }
        }
    }
}

```

Then, create a main method and save it to Main.java.

```java
package binarysearchtree;

public class Main {
    public static void main(String[] args) {
        //Create a binary tree
        BinarySearchTree bt = new BinarySearchTree();
        bt.addNode(new Node(8));
        bt.addNode(new Node(3));
        bt.addNode(new Node(1));
        bt.addNode(new Node(6));
        bt.addNode(new Node(4));
        bt.addNode(new Node(7));
        bt.addNode(new Node(10));
        bt.addNode(new Node(14));
        bt.addNode(new Node(13));

        //Print a binary tree
        System.out.println("\nPREODER: Root - Left - Right");
        BinarySearchTree.preorderPrint(bt.root);
        System.out.println("\nPOSTODER: Left - Right - Root");
        BinarySearchTree.postorderPrint(bt.root);
        System.out.println("\nINODER: Left - Root - Right");
        BinarySearchTree.inorderPrint(bt.root);
        System.out.println("");

        //Search an element in a binary tree
        System.out.println("\nSearch");
        System.out.println(BinarySearchTree.searchValue(bt.root, 3));

        //Delete a node
        System.out.println("\nDelete 4");
        bt.deleteNode(4);

        //Print a binary tree
        System.out.println("\nPREODER: Root - Left - Right");
        BinarySearchTree.preorderPrint(bt.root);

        System.out.println("\nPOSTODER: Left - Right - Root");
        BinarySearchTree.postorderPrint(bt.root);

        System.out.println("\nINODER: Left - Root - Right");
        BinarySearchTree.inorderPrint(bt.root);

        System.out.println("");
    }
}
```

The following is the output when the program is run.

```
Output - BinarySearchTree (run)

run:

PREODER: Root - Left - Right
8 3 1 6 4 7 10 14 13
POSTODER: Left - Right - Root
1 4 7 6 3 13 14 10 8
INODER: Left - Root - Right
1 3 4 6 7 8 10 13 14

Search
true

Delete 4

PREODER: Root - Left - Right
8 3 1 6 7 10 14 13
POSTODER: Left - Right - Root
1 7 6 3 13 14 10 8
INODER: Left - Root - Right
1 3 6 7 8 10 13 14
BUILD SUCCESSFUL (total time: 0 seconds)
```

3.3 Task

1. Implement a binary search tree using a linked structure with all the methods described.
2. Implement the delete function in the binary search tree.