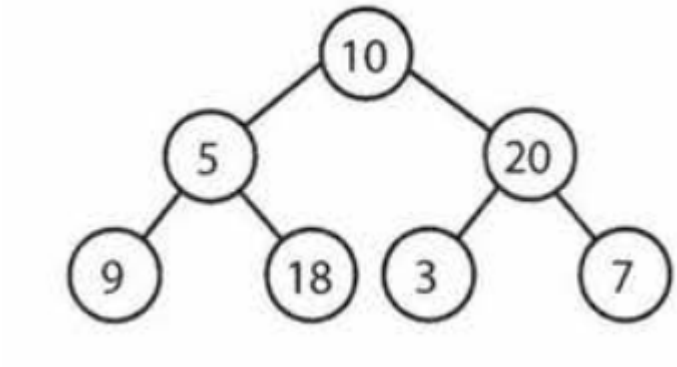
Un árbol es un tipo de grafo que no tiene ciclos, y que contiene un nodo root que puede o no tener nodos hijos, un nodo hoja es aquel que no tiene hijos. Existen distintos tipos de árboles entre ellos los más importantes son :

-Árboles binarios : Los nodos padre no pueden tener más de 2 hijos, pueden tener 0,1,2

-Árbol binario de búsqueda: es un árbol binario donde todos sus nodos del lado izquierdo tienen un valor menor a sus padres inmediatos, y todos los nodos del lado derecho son mayores a su nodo padre inmediato

-Un árbol balanceado: Se considera balanceado si asegura una compeljidad de O(log n) para la inserccion y busqueda.
-Un árbol binario completo: Un árbol es ocmpleto siempre que se llene de izquierda a derecha y cada nivel debe estar lleno por completo, excepto el último nivel, ese puede no estar lleno por completo.

-Un arbol binario perfecto: esta lleno en su totalidad de sus niveles , debe tener (2^k)-1 nodos donde k son los niveles de profundidad que tiene.



Recorridos de un árbol binario:

M: nodo medio o padre
I: nodo hijo izquierdo
D: nodo hijo derecho

INORDER

I**M**D

```
void inOrderTraversal(TreeNode node) {
   if (node != null) {
      inOrderTraversal(node.left);
      visit(node);
      inOrderTraversal(node.right);
   }
}
```

POSTORDER

ID**M**

```
void postOrderTraversal(TreeNode node) {
   if (node != null) {
      postOrderTraversal(node.left);
      postOrderTraversal(node.right);
      visit(node);
   }
}
```

PREORDER

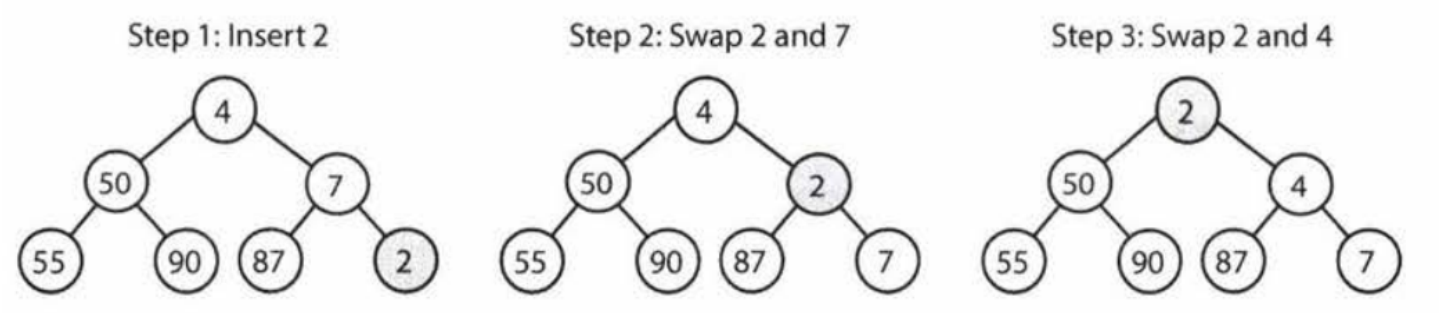
MID

```
void preOrderTraversal(TreeNode node) {
   if (node != null) {
      visit(node);
      preOrderTraversal(node.left);
      preOrderTraversal(node.right);
   }
}
```

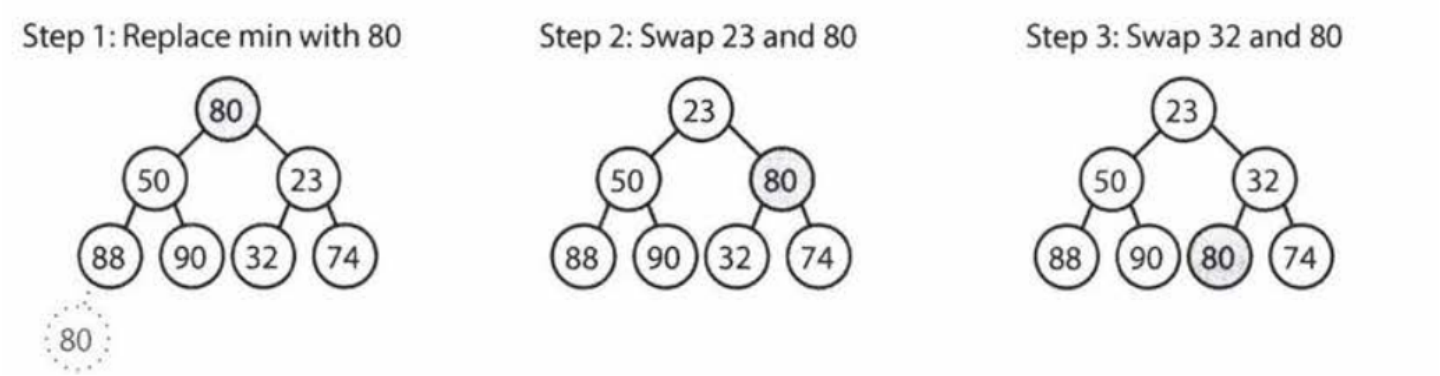
-MONTICULOS BINARIOS

Son un tipo de árbol binario donde los elementos se encuentran ordenados descendientemente en un MAX-heap y ascendente en un MIN-heap. Se tienen 2 operaciones: insertar y extraer.

insertar: se debe de insertar en el nivel mas profundo lo mas a la derecha posible para mantener un arbol completo. Posteriormente dependiendo si es un heap MAX o MIN se debe de swapear con su padre hasta llegar a su lugar adecuado.



extraer el minimo o maximo: Se debe swapear el nodo de la cima del arbol es decir de root contra el nodo mas abajo mas a la deracha. Finalmente se swapea hasta bajar y dejarlo en su lugar indicado.



```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Tree<T>{

    private T data = null;
    private List<Tree> children = new ArrayList<>();
    private Tree parent = null;

    public Tree(T data) {
        this.data = data;
    }

    public void addChild(Tree child) {
            System.out.println("Añadiendo hijo^^^^^^^^^^^^^^^");
            System.out.printf("CHILD: %s, %s\n", child,child.data);
            System.out.printf("PARENT2 : %s, %s\n", this, this.data);
            child.setParent(this);
        this.children.add(child);
    }

    public void addChild(T data) {
            System.out.println("Añadiendo hijo**************");
        Tree<T> newChild = new Tree<>(data);
        System.out.printf("CHILD: %s, %s\n", newChild,newChild.data);
        System.out.printf("PARENT1 : %s, %s\n", this,this.data);
        newChild.setParent(this);//this es el nodo que llamo a addChild
        children.add(newChild);
    }

    public void addChildren(List<Tree> children) {
            System.out.println("Añadiendo hijos----------");
        for(Tree t : children) {
            t.setParent(this);
        }
        this.children.addAll(children);
    }

    public List<Tree> getChildren() {
        return children;
    }

    public T getData() {
        return data;
    }

    public void setData(T data) {
        this.data = data;
    }
```

```
    private void setParent(Tree parent) {
        this.parent = parent;//parent es el nodo que llamo a addChild osea el padre, this.parent es el nodo recien creado newChild
    }

    public Tree getParent() {
        return parent;
    }

    public static void main(String[] args){

        Tree<String> root = new Tree<>("Root");

        /*primer nivel*/
        Tree<String> ch1 = new Tree<>("CH1");
        Tree<String> ch2 = new Tree<>("CH2");
        Tree<String> ch3 = new Tree<>("CH3");
        /*segundo nivel*/
        Tree<String> g1 = new Tree<>("G1");
        Tree<String> g2 = new Tree<>("G2");
        Tree<String> g3 = new Tree<>("G3");

        ch1.addChild(g1); ch1.addChild(g2); ch2.addChild(g3);

        /*Tree<String> child1 = new Tree<>("Child1");
        child1.addChild("Grandchild1");//creamos y agregamos a "grandchild1" a child1 como su hijo
        child1.addChild("Grandchild2");

        Tree<String> child2 = new Tree<>("Child2");
        child2.addChild("Grandchild3");

        root.addChild(child1);//agregamos un nodo a root
        root.addChild(child2);
        root.addChild("Child3");

        root.addChildren(Arrays.asList(
            new Tree<>("Child4"),
            new Tree<>("Child5"),
            new Tree<>("Child6")
        ));*/

        for(Tree node : root.getChildren()) {
            System.out.println(node.getData());
        }
    }
}
```

```
public class BinaryTree {

    Node root;

    public void addNode(int key, String name) {

        // Create a new Node and initialize it

        Node newNode = new Node(key, name);

        // If there is no root this becomes root

        if (root == null) {

            root = newNode;

        } else {

            // Set root as the Node we will start
            // with as we traverse the tree

            Node focusNode = root;

            // Future parent for our new Node

            Node parent;

            while (true) {

                // root is the top parent so we start
                // there

                parent = focusNode;

                // Check if the new node should go on
                // the left side of the parent node

                if (key < focusNode.key) {

                    // Switch focus to the left child

                    focusNode = focusNode.leftChild;

                    // If the left child has no children

                    if (focusNode == null) {

                        // then place the new node on the left of it

                        parent.leftChild = newNode;
                        return; // All Done
```

```
                    }

                } else { // If we get here put the node on the right

                    focusNode = focusNode.rightChild;

                    // If the right child has no children

                    if (focusNode == null) {

                        // then place the new node on the right of it

                        parent.rightChild = newNode;
                        return; // All Done

                    }

                }

            }
        }

    }

    // All nodes are visited in ascending order
    // Recursion is used to go to one node and
    // then go to its child nodes and so forth

    public void inOrderTraverseTree(Node focusNode) {

        if (focusNode != null) {

            // Traverse the left node

            inOrderTraverseTree(focusNode.leftChild);

            // Visit the currently focused on node

            System.out.println(focusNode);

            // Traverse the right node

            inOrderTraverseTree(focusNode.rightChild);

        }

    }
```

```
public void preorderTraverseTree(Node focusNode) {

        if (focusNode != null) {

                System.out.println(focusNode);

                preorderTraverseTree(focusNode.leftChild);
                preorderTraverseTree(focusNode.rightChild);

        }

}

public void postOrderTraverseTree(Node focusNode) {

        if (focusNode != null) {

                postOrderTraverseTree(focusNode.leftChild);
                postOrderTraverseTree(focusNode.rightChild);

                System.out.println(focusNode);

        }

}

public Node findNode(int key) {

        // Start at the top of the tree

        Node focusNode = root;

        // While we haven't found the Node
        // keep looking

        while (focusNode.key != key) {

                // If we should search to the left

                if (key < focusNode.key) {

                        // Shift the focus Node to the left child

                        focusNode = focusNode.leftChild;

                } else {

                        // Shift the focus Node to the right child

                        focusNode = focusNode.rightChild;

                }
```

```
                    // The node wasn't found

                    if (focusNode == null)
                        return null;

                }

                return focusNode;

        }

        public static void main(String[] args) {

                BinaryTree theTree = new BinaryTree();

                theTree.addNode(50, "Boss");

                theTree.addNode(25, "Vice President");

                theTree.addNode(15, "Office Manager");

                theTree.addNode(30, "Secretary");

                theTree.addNode(75, "Sales Manager");

                theTree.addNode(85, "Salesman 1");

                // Different ways to traverse binary trees

                // theTree.inOrderTraverseTree(theTree.root);

                 //theTree.preorderTraverseTree(theTree.root);

                theTree.postOrderTraverseTree(theTree.root);

                // Find the node with key 75

                //System.out.println("\nNode with the key 25");

                //System.out.println(theTree.findNode(25));

        }
}

class Node {

        int key;
        String name;

        Node leftChild;
        Node rightChild;
```

```
    Node(int key, String name) {

        this.key = key;
        this.name = name;

    }

    public String toString() {

        return name + " has the key " + key;

        /*
         * return name + " has the key " + key + "\nLeft Child: " + leftChild +
         * "\nRight Child: " + rightChild + "\n";
         */

    }

}
```