A paper with text and images

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

import java.util.Vector;

public class BinaryTree{

    Node root;

    private Node addRecursive(Node current, int value){

        if(current == null){

            return new Node(value);

        }

        if(value < current.value){

            current.left = addRecursive(current.left, value);

        }else if(value > current.value){

            current.right = addRecursive(current.right, value);

        }else{

            return current;

        }

        return current;

    }

    public void add(int value){

        root = addRecursive(root, value);

    }

    class Node{

        int value;

        Node left;

        Node right;

        Node(int value){

            this.value = value;

            right = null;

            left = null;

        }

    }

    private static BinaryTree createBinaryTree(){

        BinaryTree bt = new BinaryTree();

        bt.add(6);

        bt.add(4);

        bt.add(8);

        bt.add(3);

        bt.add(5);

        bt.add(7);

        bt.add(9);

        return bt;

    }

    private boolean containsNodeRecursive(Node current, int value){

        if(current == null){

            return false;

        }

        if(value == current.value){

            return true;

        }

        if(value < current.value){

            return containsNodeRecursive(current.left, value);

        }else{

        return value < current.value ? containsNodeRecursive(current.left, value) : containsNodeRecursive(current.right, value);

        }

    }

    private boolean containsNode(int value){

        return containsNodeRecursive(root, value);

    }

    private Node deleteRecursive(Node current, int value){

        if(current == null){

            return null;

        }

        if(value == current.value){

            if(current.left == null && current.right == null){

                return null;

            }

            if(current.right == null){

                return current.left;

            }

            if(current.left == null){

                return current.right;

            }

            int smallestValue = findSmallestValue(current.right);

            current.value = smallestValue;

            current.right = deleteRecursive(current.right, smallestValue);

            return current;

        }

        if(value < current.value){

            current.left = deleteRecursive(current.left, value);

            return current;

        }

        current.right = deleteRecursive(current.right, value);

        return current;

    }

    private int findSmallestValue(Node root){

        return root.left == null ? root.value : findSmallestValue(root.left);

    }

    public void delete(int value){

        root = deleteRecursive(root, value);

    }

    public static void traverseInOrder(Node node){

        if(node != null){

            traverseInOrder(node.left);

            System.out.print(" " + node.value);

            traverseInOrder(node.right);

        }

    }

    public static void traversePreOrder(Node node){

        if(node != null){

            System.out.print(" " + node.value);

            traversePreOrder(node.left);

            traversePreOrder(node.right);

        }

    }

    public static void traversePostOrder(Node node){

        if(node != null){

            traversePostOrder(node.left);

            traversePostOrder(node.right);

            System.out.print(" " + node.value);

        }

    }

    public static void traverseLevelOrder(Node root){

        if(root == null){

            return;

        }

        Vector<Node> nodes = new Vector<Node>();

        nodes.add(root);

        while(!nodes.isEmpty()){

            Node node = nodes.remove(0);

            System.out.print(" " + node.value);

            if(node.left != null){

                nodes.add(node.left);

            }

            if(node.right != null){

                nodes.add(node.right);

            }

        }

    }

    public static void printTreeBFS(Node root){

        if(root == null){

            return;

        }

        Vector<Node> nodes = new Vector<Node>();

        nodes.add(root);

        while(!nodes.isEmpty()){

            Node node = nodes.remove(0);

            System.out.print(" " + node.value);

            if(node.left != null){

                nodes.add(node.left);

            }

            if(node.right != null){

                nodes.add(node.right);

            }

        }

    }

    public static void main(String[] args) {

        BinaryTree bt = createBinaryTree();

        System.out.println(bt.containsNode(6));

        System.out.println(bt.containsNode(4));

        System.out.println(bt.containsNode(1));

        System.out.println(bt.containsNode(9));

        bt.delete(9);

        System.out.println(bt.containsNode(9));

        traverseInOrder(bt.root);

        System.out.println();

        traversePreOrder(bt.root);

        System.out.println();

        traversePostOrder(bt.root);

        System.out.println();

        traverseLevelOrder(bt.root);

        System.out.println();

        printTreeBFS(bt.root);

    }

}

Output:

A screenshot of a computer

Description automatically generated

Optimization:

class Node:

    def \_\_init\_\_(self, value):

        self.value = value

        self.left = None

        self.right = None

def insert(root, value):

    if root is None:

        return Node(value)

    if value < root.value:

        root.left = insert(root.left, value)

    else:

        root.right = insert(root.right, value)

    return root

def print\_level\_order(root):

    if root is None:

        return

    queue = []

    queue.append(root)

    while queue:

        count = len(queue)

        for i in range(count):

            node = queue.pop(0)

            if node:

                print(node.value, end=" ")

                if node.left:

                    queue.append(node.left)

                else:

                    queue.append(None)

                if node.right:

                    queue.append(node.right)

                else:

                    queue.append(None)

            else:

                print(".", end=" ")

        print()

# Create the tree

values = [6, 4, 8, 3, 5, 7, 9]

root = None

for value in values:

    root = insert(root, value)

# Print tree in level order

print\_level\_order(root)

output:

A screen shot of a computer

Description automatically generated

All code works properly