```
import java.util.LinkedList;
import java.util.Queue;

class TreeNode {
	int data;
	TreeNode left;
	TreeNode right;

	public TreeNode(int data){
		this.data = data;
		this.left = left;
		this.right = right;
	}

}

class BinarySearchTree {

	static TreeNode DUMMY = new TreeNode(-100);

	public TreeNode find(TreeNode root, int data){
		if(root == null)
			return null;

		while(root != null){

			if(root.data == data)
				return root;
			else if(data < root.data)
				root = root.left;
			else
				root = root.right;
		}

		return root;
	}

	public TreeNode minimum(TreeNode root){
		if(root == null)
			return null;
		if(root.left == null)
			return root;
		while(root.left != null)
			root = root.left;
		return root;
	}

	public TreeNode maximum(TreeNode root ){
		if(root == null)
			return null;
		if(root.right == null)
			return root;
		while(root.right != null)
			root = root.right;
		return root;
```

```
}

public TreeNode insert(TreeNode root, int data){
	if(root == null){
		root = new TreeNode(data);
		return root;
	}
	else{
		if(data < root.data){
			root.left = insert(root.left,data);
		}
		else if(data > root.data){
			root.right = insert(root.right,data);
		}
	}
	return root;
}

public TreeNode deleteNode(TreeNode root, int data){
	TreeNode ptr,parent,parent_replacement,replacement,child_ptr;
	int is_left = 0;
	parent = null;
	ptr = root;
	while(ptr != null){
		if(ptr.data == data)
			break;
		else if(ptr.data > data){
			parent = ptr;
			is_left = 1;
			ptr = ptr.left;
		}
		else {
			parent = ptr;
			is_left = 0;
			ptr = ptr.right;
		}
	}

	if(ptr == null){
		System.out.println("Node with " + data + " not found");
		return root;
	}
	if(ptr.left == null && ptr.right == null)
	{
		if(parent == null){
			if(ptr == root){
				root = null;
			}
		}
		else{
			if(is_left == 1)
				parent.left = null;
			else
				parent.right = null;
		}
```

```
	}
	else if(ptr.left == null || ptr.right == null){
		if(ptr.left != null)
			child_ptr = ptr.left;
		else
			child_ptr = ptr.right;

		if(parent == null){
			if(ptr == root){
				root = child_ptr;
			}
		}
		else {
			if(is_left == 1)
				parent.left = child_ptr;
			else
				parent.right = child_ptr;
		}

	}
	else {
		parent_replacement = ptr;
		replacement = ptr.left;
		is_left = 1;
		while(replacement.right != null){
			parent_replacement = replacement;
			is_left = 0;
			replacement = replacement.right;
		}
		ptr.data = replacement.data;
		if(is_left == 1){
			if(replacement.right == null){
				ptr.left = replacement.left;
			}
		}
		else {
			parent_replacement.right = replacement.left;
		}
	}

	return root;

}

public TreeNode LCA(TreeNode root, TreeNode a, TreeNode b){
	if(root == null)
		return null;
	if(root == a || root == b)
		return root;
	if(Math.max(a.data,b.data) < root.data)
		return LCA(root.left,a,b);
	else if(Math.min(a.data,b.data) > root.data)
		return LCA(root.right,a,b);
	else
		return root;
```

```
	}

	public boolean isBST(TreeNode root){
		return isBSTUtil(root,Integer.MIN_VALUE,Integer.MAX_VALUE);
	}

	private boolean isBSTUtil(TreeNode root, int min, int max)
	{
		if(root == null)
			return true;
		if(root.data < min && root.data > max)
			return false;
		return isBSTUtil(root.left,min,root.data) &&
isBSTUtil(root.right,root.data,max);
	}

	public boolean isBSTUsingInOrder(TreeNode root){
		int prev = Integer.MIN_VALUE;
		return isBSTUsingInOrderUtil(root,prev);
	}

	private boolean isBSTUsingInOrderUtil(TreeNode root, int prev){
		if(root == null)
			return true;
		if(!isBSTUsingInOrderUtil(root.left,prev))
			return false;
		if(root.data < prev)
			return false;
		prev = root.data;
		return isBSTUsingInOrderUtil(root.right,prev);
	}

	public TreeNode floorInBST(TreeNode root, int data){
		TreeNode prev = null;
		return floorInBSTUtil(root,prev,data);
	}

	public TreeNode floorInBSTUtil(TreeNode root, TreeNode prev, int data){
		if(root == null)
			return null;
		if(floorInBSTUtil(root.left,prev,data) == null )
			return null;
		if(root.data == data)
			return root;
		if(root.data > data)
			return prev;
		prev = root;
		return floorInBSTUtil(root.right,prev,data);
	}

	public TreeNode ceilInBST(TreeNode root, int data){
		TreeNode prev = null;
		return ceilInBSTUtil(root,prev,data);
	}
```

```
public TreeNode ceilInBSTUtil(TreeNode root, TreeNode prev, int data){
      if(root == null)
            return null;
      if(ceilInBSTUtil(root.right,prev,data) == null )
            return null;
      if(root.data == data)
            return root;
      if(root.data < data)
            return prev;
      prev = root;
      return ceilInBSTUtil(root.left,prev,data);
}

public void printNodesInRange(TreeNode root, int K1, int K2){
      Queue<TreeNode> queue = new LinkedList<>();
  queue.add(root);
  queue.add(DUMMY);
  while(!queue.isEmpty()){
      TreeNode node = queue.poll();

      if(node.data != DUMMY.data) {
            if(node.data >= K1 && node.data <= K2){
            System.out.print(node.data + " ");
          }
          if (node.left != null && node.data >= K1) {
              queue.add(node.left);
          }
          if (node.right != null && node.data <= K2) {
              queue.add(node.right);
          }
      }
      else {
          if(!queue.isEmpty()){
              queue.add(DUMMY);
          }
      }
  }
}

public void inOrder(TreeNode root){
      if(root != null){
            inOrder(root.left);
            System.out.print(root.data + " ");
            inOrder(root.right);
      }
}

public static void main(String args[]){
      TreeNode root = new TreeNode(6);
      root.left = new TreeNode(3);
      root.right = new TreeNode(12);
      root.left.left = new TreeNode(1);
      root.left.right = new TreeNode(4);
      root.left.right.right = new TreeNode(5);
      root.right.left = new TreeNode(7);
```

```
            root.right.right = new TreeNode(14);
            root.right.left.right = new TreeNode(9);
            root.right.left.right.left = new TreeNode(8);
            root.right.left.right.right = new TreeNode(10);

            BinarySearchTree bst = new BinarySearchTree();
            bst.inOrder(root);
            System.out.println();
            TreeNode node = bst.find(root,10);
            if(node != null){
                  System.out.println("Node with data 10 found.....");
            }
            node = bst.minimum(root);
            if(node != null){
                  System.out.println("Node with minumum value: " +
node.data);
            }

            node = bst.maximum(root);
            if(node != null){
                  System.out.println("Node with maximum value: " +
node.data);
            }

            node = bst.insert(root,15);
            bst.inOrder(node);
            System.out.println();

            node = bst.deleteNode(root,12);
            bst.inOrder(node);
            System.out.println();

      }

}
```

```
import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;

class TreeNode {
      int data;
      TreeNode left;
      TreeNode right;

      public TreeNode(int data){
            this.data = data;
            this.left = null;
            this.right = null;
      }
}
```

```
class Traversals{

      static TreeNode DUMMY = new TreeNode(-100);

      public void preOrder(TreeNode root){
            if(root != null){
                  System.out.print(root.data + " ");
                  preOrder(root.left);
                  preOrder(root.right);
            }
      }

      public void inOrder(TreeNode root){
            if(root != null){
                  inOrder(root.left);
                  System.out.print(root.data + " ");
                  inOrder(root.right);
            }
      }

      public void postOrder(TreeNode root){
            if(root != null){
                  postOrder(root.left);
                  postOrder(root.right);
                  System.out.print(root.data + " ");
            }
      }

      public void preOrderIterative(TreeNode root){

        Stack<TreeNode> stack = new Stack<TreeNode>();
        if (root == null)
            return;
        stack.push(root);
        while(!stack.empty()){
            TreeNode node = stack.pop();
            System.out.print(node.data + " ");
            if(node.right != null){
                stack.push(node.right);
            }
            if(node.left != null){
                stack.push(node.left);
            }
        }
    }

    public void inOrderIterative(TreeNode root){

        if (root == null)
            return;
        Stack<TreeNode> stack = new Stack<TreeNode>();
        boolean done = false;
        TreeNode node = root;
        while(!done){
```

```
            if(node!= null){
                stack.push(node);
                node = node.left;
            }
            else{
                if(stack.isEmpty()){
                    done = true;
                }
                else {
                    node = stack.pop();
                    System.out.print(node.data + " ");
                    node = node.right;
                }
            }
        }
    }

    public void levelOrder(TreeNode root){
        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root);
        queue.add(DUMMY);
        while(!queue.isEmpty()){
            TreeNode node = queue.poll();

            if(node.data != DUMMY.data) {
                System.out.print(node.data + " ");
                if (node.left != null) {
                    queue.add(node.left);
                }
                if (node.right != null) {
                    queue.add(node.right);
                }
            }
            else {
                if(!queue.isEmpty()){
                    queue.add(DUMMY);
                }
            }
        }

    }

    public void postOrderIterative(TreeNode root){
        Stack<TreeNode> stack = new Stack<>();
        stack.push(root);
        TreeNode prev = null;
        while(!stack.isEmpty()){
            TreeNode current = stack.peek();
            if(prev == null || prev.left == current || prev.right ==
current){
                if(current.left != null){
                    stack.push(current.left);
                }
                else if(current.right != null){
                    stack.push(current.right);
```

```
                }
            }
            else if(current.left == prev){
                if(current.right != null){
                    stack.push(current.right);
                }
            }
            else{
                System.out.print(current.data + " ");
                stack.pop();
            }
            prev = current;
        }
    }

    public static void main(String args[]){
      TreeNode root = new TreeNode(1);
      root.left = new TreeNode(2);
      root.right = new TreeNode(3);
      root.left.left = new TreeNode(4);
      root.left.right = new TreeNode(5);
      root.right.left = new TreeNode(6);
      root.right.right = new TreeNode(7);

      Traversals traversals = new Traversals();
      System.out.println("PreOrder Recursive....");
      traversals.preOrder(root);
      System.out.println();
      System.out.println("PreOrder Iterative....");
      traversals.preOrderIterative(root);
      System.out.println();
      System.out.println("InOrder Recursive....");
      traversals.inOrder(root);
      System.out.println();
      System.out.println("InOrder Iterative....");
      traversals.inOrderIterative(root);
      System.out.println();
      System.out.println("PostOrder Recursive....");
      traversals.postOrder(root);
      System.out.println();
      System.out.println("PostOrder Iterative....");
      traversals.postOrderIterative(root);
      System.out.println();
      System.out.println("Level Order Traversal.....");
      traversals.levelOrder(root);
      System.out.println();

    }
}
```

```
import java.util.LinkedList;
import java.util.Queue;

class SiblingTreeNode {
	int data;
	SiblingTreeNode left;
	SiblingTreeNode right;
	SiblingTreeNode nextSibling;

	public SiblingTreeNode(int data){
		this.data = data;
		this.left = null;
		this.right = null;
		this.nextSibling = null;
	}
}

class SiblingTree {

	static SiblingTreeNode DUMMY = new SiblingTreeNode(-100);

	public void fillSibling(SiblingTreeNode root){
		if(root == null)
			return;
		if(root.left != null)
			root.left.nextSibling = root.right;
		if(root.right != null){
			if(root.nextSibling != null)
				root.right.nextSibling = root.nextSibling.left;
			else
				root.right.nextSibling = null;
		}
		fillSibling(root.left);
		fillSibling(root.right);
	}

	public void printSibling(SiblingTreeNode root){
		Queue<SiblingTreeNode> queue = new LinkedList<>();
	  queue.add(root);
	  queue.add(DUMMY);
	  while(!queue.isEmpty()){
		SiblingTreeNode node = queue.poll();

		if(node.data != DUMMY.data) {
			if(node.nextSibling != null){
			System.out.println("Node: " + node.data + " sibling: " +
node.nextSibling.data);
			}
		    if (node.left != null) {
		        queue.add(node.left);
		    }
		    if (node.right != null) {
		        queue.add(node.right);
		    }
		}
```

```
            else {
                if(!queue.isEmpty()){
                    queue.add(DUMMY);
                }
            }
        }
      }

      public static void main(String args[]){
            SiblingTreeNode root = new SiblingTreeNode(1);
      root.left = new SiblingTreeNode(2);
      root.right = new SiblingTreeNode(3);
      root.left.left = new SiblingTreeNode(4);
      root.left.right = new SiblingTreeNode(5);
      root.right.left = new SiblingTreeNode(6);
      root.right.right = new SiblingTreeNode(7);

      SiblingTree obj = new SiblingTree();
      obj.fillSibling(root);
      obj.printSibling(root);
      }
}




import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;

class TreeNode {
      int data;
      TreeNode left;
      TreeNode right;

      public TreeNode(int data){
            this.data = data;
            this.left = null;
            this.right = null;
      }
}

class Height
{
    int h;
}

class TreeOperations {

      static TreeNode DUMMY = new TreeNode(-100);

      public int maxElement(TreeNode root){
            int max = Integer.MIN_VALUE;
            if(root != null){
```

```
            int left = maxElement(root.left);
            int right = maxElement(root.right);

            if(left > right)
                  max = left;
            else
                  max = right;

            if(max < root.data)
                  max = root.data;
      }

      return max;
  }

  public boolean search(TreeNode root, int data){
      if(root == null)
            return false;
      if(root.data == data)
            return true;
      return search(root.left,data) || search(root.right,data);
  }

  public TreeNode searchTreeNode(TreeNode root, int data){
    if(root == null)
        return null;
    Queue<TreeNode> queue = new LinkedList<>();
    queue.add(root);
    TreeNode temp = null;
    while(!queue.isEmpty()){
        TreeNode node = queue.poll();
        //System.out.println(node.data + " ");
        if(node.data == data) {
            temp = node;
            break;
        }
        else {
            if (node.left != null) {
                queue.add(node.left);
            }
            if (node.right != null) {
                queue.add(node.right);
            }
        }
    }
    return temp;
}


  public void insert(TreeNode root, int data){
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    while(!queue.isEmpty()){
        TreeNode node = queue.poll();
        if(node != null){
```

```
            if(node.left != null)
                queue.offer(node.left);
            else {
                node.left = new TreeNode(data);
                return;
            }
            if(node.right != null)
                queue.offer(node.right);
            else {
                node.right = new TreeNode(data);
                return;
            }
        }

    }
}

public int sizeOfTree(TreeNode root){
  if(root == null)
        return 0;
  int count = 0;
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    while(!queue.isEmpty()){
        TreeNode node = queue.poll();
        if(node != null){
            count++;
        }
        if(node.left != null)
            queue.offer(node.left);
        if(node.right != null)
              queue.offer(node.right);
    }
    return count;
}

public void levelOrderReverse(TreeNode root){
  Queue<TreeNode> queue = new LinkedList<>();
  Stack<TreeNode> stack = new Stack<>();
    queue.add(root);
    queue.add(DUMMY);
    while(!queue.isEmpty()){
        TreeNode node = queue.poll();

        if(node.data != DUMMY.data) {
            //System.out.print(node.data + " ");

            if (node.right != null) {
                queue.add(node.right);
            }
            if (node.left != null) {
                queue.add(node.left);
            }
            stack.push(node);
        }
```

```
        else {
            if(!queue.isEmpty()){
                queue.add(DUMMY);
            }
        }
    }
    while(!stack.isEmpty()){
        System.out.print(stack.pop().data + " ");
    }
    System.out.println();
}

public static int height(TreeNode root){
  Queue<TreeNode> queue = new LinkedList<>();
    queue.add(root);
    queue.add(DUMMY);
    int count = 1;
    while(!queue.isEmpty()){
        TreeNode node = queue.poll();

        if(node.data != DUMMY.data) {
            //System.out.print(node.data + " ");
            if(node.left == null && node.right == null)
              return count;

            if (node.left != null) {
                queue.add(node.left);
            }

            if (node.right != null) {
                queue.add(node.right);
            }
        }
        else {
            if(!queue.isEmpty()){
              count++;
                queue.add(DUMMY);
            }
        }
    }
    return count;
}

public static TreeNode deepestNode(TreeNode root){
  Queue<TreeNode> queue = new LinkedList<>();
    queue.add(root);
    //queue.add(DUMMY);
    TreeNode node = null;
    while(!queue.isEmpty()){
        node = queue.poll();

        if(node.data != DUMMY.data) {
            //System.out.print(node.data + " ");
            if (node.left != null) {
                queue.add(node.left);
```

```
            }
            if (node.right != null) {
                queue.add(node.right);
            }
        }
        else {
            if(!queue.isEmpty()){
                queue.add(DUMMY);
            }
        }
    }
    return node;
}

public void deleteDeepestNode(TreeNode root, TreeNode deepestNode){
  Queue<TreeNode> queue = new LinkedList<>();
    queue.add(root);
    //queue.add(DUMMY);
    while(!queue.isEmpty()){
        TreeNode node = queue.poll();
        if(node.left != null){
              if(node.left == deepestNode){
                    node.left = null;
                    return;
              }
              else{
                    queue.offer(node.left);
              }
        }
        if(node.right != null){
              if(node.right == deepestNode){
                    node.right = null;
                    return;
              }
              else{
                    queue.offer(node.right);
              }
        }
    }
}

public void deleteTreeNode(TreeNode root, int data){
  TreeNode node = searchTreeNode(root,data);
  TreeNode deepest = deepestNode(root);
  node.data = deepest.data;
  deepest.data = data;
  deleteDeepestNode(root,deepest);
}

public void levelOrder(TreeNode root){
    Queue<TreeNode> queue = new LinkedList<>();
    queue.add(root);
    queue.add(DUMMY);
    while(!queue.isEmpty()){
        TreeNode node = queue.poll();
```

```
            if(node.data != DUMMY.data) {
                System.out.print(node.data + " ");
                if (node.left != null) {
                    queue.add(node.left);
                }
                if (node.right != null) {
                    queue.add(node.right);
                }
            }
            else {
                if(!queue.isEmpty()){
                    queue.add(DUMMY);
                }
            }
        }

    }

    public boolean identicalTrees(TreeNode root1, TreeNode root2){
      if(root1 == null && root2 == null)
            return true;
      return (root1.data == root2.data &&
identicalTrees(root1.left,root2.left) &&
identicalTrees(root1.right,root2.right));
    }

    public int diameter(TreeNode root, Height height){
      if(root == null){
            height.h = 0;
            return 0;
      }
      Height lh = new Height();
      Height rh = new Height();

      int ldiameter = diameter(root.left,lh);
      int rdiameter = diameter(root.right,rh);

      height.h = Math.max(lh.h,rh.h) + 1;

      return Math.max(lh.h+rh.h+1, Math.max(ldiameter,rdiameter));
    }

    public int widthOfTree(TreeNode root){
      Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root);
        int maxWidth = 0;
        while(!queue.isEmpty()){
            int count = queue.size();
            maxWidth = Math.max(count,maxWidth);

            while(count-- > 0){
                  TreeNode node = queue.poll();
                  if(node.left != null)
                        queue.push(node.left);
```

```
                if(node.right != null)
                      queue.push(node.right);
          }

      }
      return maxWidth;
}

public int maxSumLevel(TreeNode root){
  Queue<TreeNode> queue = new LinkedList<>();
    queue.add(root);
    int maxSum = 0;
    while(!queue.isEmpty()){
          int count = queue.size();
          int sum = 0;

          while(count-- > 0){
                TreeNode node = queue.poll();
                sum = sum + node.data;
                if(node.left != null)
                      queue.push(node.left);
                if(node.right != null)
                      queue.push(node.right);
          }

          maxSum = Math.max(sum,maxSum);
    }
    return maxSum;
}

public void printPaths(TreeNode root){
  int paths[] = new int[256];
  printAllPaths(root,paths,0);
}

private void printAllPaths(TreeNode root, int paths[], int len){
  if(root == null)
        return;
  paths[len] = root.data;
  len++;
  if(root.left == null && root.right == null)
        printPath(paths,len);
  else{
        printAllPaths(root.left,paths,len);
        printAllPaths(root.right,paths,len);
  }
}

private void printPath(int paths[], int len){
  for(int i=0;i<len;i++){
        System.out.print(paths[i] + " ");
  }
  System.out.println();
}
```

```
    public boolean hasPathSum(TreeNode root, int sum){
      if(root == null)
            return false;
      if(root.left == null && root.right == null && root.data == sum)
            return true;
      else
            return hasPathSum(root.left,sum-root.data) ||
hasPathSum(root.right,sum-root.data);
    }

    public TreeNode mirrorOfTree(TreeNode root){
      TreeNode temp;
      if(root != null){
            mirrorOfTree(root.left);
            mirrorOfTree(root.right);
            temp = root.left;
            root.left = root.right;
            root.right = temp;
      }
      return root;
    }

    public boolean areMirrors(TreeNode root1, TreeNode root2){
      if(root1 == null && root2 == null)
            return true;
      if(root1 == null || root2 == null)
            return false;
      if(root1.data != root2.data)
            return false;
      else
            return areMirrors(root1.left,root2.right) &&
areMirrors(root1.right,root2.left);
    }

    public TreeNode buildTreeUsingInAndPreOrder(int preorder[], int
inorder[]){
      return buildTree(preorder, 0, preorder.length-1, inorder, 0,
inorder.length-1);
    }

    private TreeNode buildTree(int preorder[], int preStart, int preEnd, int
inorder[], int inStart, int inEnd){
      if(preStart > preEnd || inStart > inEnd || preorder.length !=
inorder.length)
            return null;
      TreeNode current = new TreeNode(preorder[preStart]);
      int offset = inStart;
      for(;offset<inEnd;offset++){
            if(inorder[offset] == current.data)
                  break;
      }
      current.left = buildTree(preorder,preStart+1,preStart+offset-
inStart,inorder,inStart,offset-1);
      current.right = buildTree(preorder,preStart+offset-
inStart+1,preEnd,inorder,offset+1,inEnd);
```

```
      return current;
    }

    public TreeNode buildTreeUsingInAndPreOrder(int postorder[], int
inorder[]){
      return buildTreePost(postorder, 0, postorder.length-1, inorder, 0,
inorder.length-1);
    }

    private TreeNode buildTreePost(int postorder[], int postStart, int
postEnd, int inorder[], int inStart, int inEnd){
      if(postStart > postEnd || inStart > inEnd || postorder.length !=
inorder.length)
            return null;
      TreeNode current = new TreeNode(postorder[postEnd]);
      int offset = inStart;
      for(;offset<inEnd;offset++){
            if(inorder[offset] == current.data)
                  break;
      }
      current.left = buildTreePost(postorder,postStart,postStart+offset-
inStart-1,inorder,inStart,offset-1);
      current.right = buildTreePost(postorder,postStart+offset-
inStart,postEnd-1,inorder,offset+1,inEnd);
      return current;
    }

    public boolean printAllAncestors(TreeNode root, TreeNode node){
      if(root == null)
            return false;
      if(root.left == node || root.right == node ||
printAllAncestors(root.left,node) || printAllAncestors(root.right,node)){
            System.out.println(root.data);
            return true;
      }
      return false;
    }

    public TreeNode LCA(TreeNode root, TreeNode a, TreeNode b){
      if(root == null)
            return null;
      if(root == a || root == b)
            return root;
      TreeNode left = LCA(root.left,a,b);
      TreeNode right = LCA(root.right,a,b);

      if(left != null && right!= null)
            return root;
      else
            return (left != null ? left : right);
    }

    public void verticalSum(TreeNode root){
      HashMap<Integer,Integer> hashMap = new HashMap<>();
      vSum(root,hashMap,0);
```

```
      for(int k: hashMap.keySet()){
            System.out.println("Key: " + k + " with Sum: " + hashMap.get(k));
      }
    }

    public void vSum(TreeNode root, HashMap<Integer,Integer> hashMap, int
column){
      if(root == null)
            return;
      if(root.left != null){
            vSum(root.left,hashMap,column-1);
      }
      if(root.right != null){
            vSum(root.right,hashMap,column+1);
      }
      int data = 0;
      if(hashMap.containsKey(column)){
            data = hashMap.get(column);
      }
      hashMap.put(column,root.data+data);

    }

    public ArrayList<Integer> zigZagOrder(TreeNode root){
        ArrayList<Integer> result = new ArrayList<Integer>();
        Queue<TreeNode> queue = new LinkedList<>();

        queue.add(root);
        queue.add(DUMMY);
        boolean leftToRight = true;
        ArrayList<Integer> current = new ArrayList<Integer>();
        while(!queue.isEmpty()){
            TreeNode node = queue.poll();

            if(node.data != DUMMY.data) {
                current.add(node.data);
                if (node.left != null) {
                    queue.add(node.left);
                }
                if (node.right != null) {
                    queue.add(node.right);
                }

            }
            else {
                if(leftToRight){
                    result.addAll(current);
                    current.clear();
                }
                else{
                    Stack<Integer> stack = new Stack<Integer>();
                    stack.addAll(current);
                    while(!stack.isEmpty()){
                        result.add(stack.pop());
                    }
```

```
                current.clear();
            }
            if(!queue.isEmpty()){
                queue.add(DUMMY);
                leftToRight = !leftToRight;
            }
        }
    }
    return result;

}

public ArrayList<TreeNode> generateTrees(int n){
  if(n==0)
        return generate(1,0);
  return generate(1,n);
}

public ArrayList<TreeNode> generate(int start,int end){
  ArrayList<TreeNode> subtrees = new ArrayList<TreeNode>();
  if(start > end){
        subtrees.add(null);
        return subtrees;
  }
  for(int i=start;i<=end;i++){
        for(TreeNode left : generate(start,i-1)){
              for(TreeNode right : generate(i+1,end)){
                    TreeNode node = new TreeNode(i);
                    node.left = left;
                    node.right = right;
                    subtrees.add(node);
              }
        }
  }
  return subtrees;
}

public static void main(String args[]){
  TreeNode root = new TreeNode(1);
  root.left = new TreeNode(2);
  root.right = new TreeNode(3);
  root.left.left = new TreeNode(4);
  root.left.right = new TreeNode(5);
  root.right.left = new TreeNode(6);
  root.right.right = new TreeNode(7);

  TreeOperations obj = new TreeOperations();

  System.out.println("Max Element: " + obj.maxElement(root));
  System.out.println("Search Element Found: " + obj.search(root,12));
  obj.insert(root,8);
  obj.insert(root,9);
  System.out.println("Size of the Tree: " + obj.sizeOfTree(root));
  obj.levelOrderReverse(root);
  System.out.println("Height of Tree: " + height(root));
```

```
      System.out.println("Deepest Node: " + deepestNode(root).data);
      obj.deleteTreeNode(root,3);
      obj.levelOrder(root);
      System.out.println();
      System.out.println("Diameter of the Tree: " + obj.diameter(root,new
Height()));

    }


}
```