```
public class TreesStriver {
//**************************Binary Tree
********************************************************************************** */
//**************************InOrder
Traversal*****************************************************************************************
************************** */
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        ArrayList<Integer> result=new ArrayList<>();
        inOrder(root,result);
        return result;
        }
    public void inOrder(TreeNode root,ArrayList<Integer> result){
        if(root==null){
        return ;
        }
        inOrder(root.left,result);
        result.add(root.val);
        inOrder(root.right,result);
        }
    }
//**************************Post Order
Traversal*****************************************************************************************
************************** */
class Solution {
    public List<Integer> postorderTraversal(TreeNode root) {
        ArrayList<Integer> result=new ArrayList<>();
        postOrder(root,result);
        return result;
        }
    public void postOrder(TreeNode root,ArrayList<Integer> result){
        if(root==null){
        return ;
        }
        postOrder(root.left,result);
        postOrder(root.right,result);
        result.add(root.val);
        }
    }
//**************************Pre Order
Traversal*****************************************************************************************
************************** */
class Solution {
    public List<Integer> preorderTraversal(TreeNode root) {
        ArrayList<Integer> result=new ArrayList<>();
        preOrder(root,result);
        return result;
        }
    public void preOrder(TreeNode root,ArrayList<Integer> result){
```

```
        if(root==null){
        return ;
        }
        result.add(root.val);
        preOrder(root.left,result);
        preOrder(root.right,result);
        }
}
//**************************Left View of a Binary
Tree*****************************************************************************************
********** */
class Solution{
    ArrayList<Integer> leftView(Node root)
    {
        ArrayList<Integer> result=new ArrayList<>();
        printLeftView(root,result,0);
        return result;
        }
    public void printLeftView(Node root,ArrayList<Integer> result,int level){
        if(root==null){
        return;
        }
        if(level==result.size()){
        result.add(root.data);
        }
        printLeftView(root.left,result,level+1);
        printLeftView(root.right,result,level+1);
        }
    }
//**************************Right View of a Binary
Tree*****************************************************************************************
********** */
class Solution{
    ArrayList<Integer> RightView(Node root)
    {
        ArrayList<Integer> result=new ArrayList<>();
        printRightView(root,result,0);
        return result;
    }
    public void printRightView(Node root,ArrayList<Integer> result,int level){
        if(root==null){
        return;
        }
        if(level==result.size()){
        result.add(root.data);
        }
        printRightView(root.right,result,level+1);
        printRightView(root.left,result,level+1);
    }
```

```
}
//**************************Top  View of a Binary
Tree*************************************************************************************
********** */
class Solution{
    class Pair{
        Node node;
        int hd;

        Pair(Node temp,int h){
        this.node=temp;
        this.hd=h;
        }
    }
    class Solution
    {
    static ArrayList<Integer> topView(Node root)
    {
        ArrayList<Integer> ans=new ArrayList<>();
            if(root==null)
            return ans;
        Map<Integer,Integer> map=new TreeMap<>();
        Queue<Pair> q=new LinkedList<Pair>();

        q.add(new Pair(root,0));
        while(!q.isEmpty()) {
            Pair it=q.remove();
            int hd =it.hd;
            Node temp=it.node;
            if(map.get(hd)==null)
            map.put(hd, temp.data);
            if(temp.left!=null) {q.add(new Pair(temp.left,hd-1));}
            if(temp.right!=null) {q.add(new Pair(temp.right,hd+1));}
        }
        for (Map.Entry<Integer,Integer> entry : map.entrySet()) {
        ans.add(entry.getValue());
        }
        return ans;
        }
    }
}
//**************************Bottom View of a Binary
Tree*************************************************************************************
********** */
class Solution
{
    public ArrayList <Integer> bottomView(Node root)
    // Code here
```

```
    {
        ArrayList<Integer> ans = new ArrayList<>();
        if(root==null)
        return ans;
        Map<Integer,Integer> map=new TreeMap<>();
        Queue<Node> q=new LinkedList<Node>();
        root.hd = 0;
        q.add(root);
        while(!q.isEmpty()) {
            Node temp=q.remove();
            int hd=temp.hd;
            map.put(hd,temp.data);
            if(temp.left!=null){
            temp.left.hd=hd-1;
            q.add(temp.left);
            }
            if(temp.right!=null) {
            temp.right.hd=hd+1;
            q.add(temp.right);
            }
        }

        for(Map.Entry<Integer,Integer> entry: map.entrySet()){
        ans.add(entry.getValue());
        }
        return ans;
    }
}
//**************************Vertical Order
Traversal*****************************************************************************************
*********************************/
class Solution{
    class Tuple{
        TreeNode node;
        int row;
        int col;

        Tuple(TreeNode _node,int _row,int _col){
        this.node=_node;
        this.row=_row;
        this.col=_col;
        }
        }
    class Solution {
    public List<List<Integer>> verticalTraversal(TreeNode root) {
        TreeMap<Integer,TreeMap<Integer,PriorityQueue <Integer>>> map=new
TreeMap<>();
        Queue<Tuple> q=new LinkedList <Tuple>();
        q.offer(new Tuple(root,0,0));
```

```
            while(!q.isEmpty()){
                Tuple tuple=q.poll();
                TreeNode node=tuple.node;
                int x=tuple.row;
                int y=tuple.col;
                if(!map.containsKey(x)){
                    map.put(x,new TreeMap<>());
                }
                if(!map.get(x).containsKey(y)) {
                    map.get(x).put(y, new PriorityQueue<>());
                }
                map.get(x).get(y).offer(node.val);

                if(node.left!=null){
                    q.offer(new Tuple(node.left,x-1,y+1));
                }
                if(node.right!=null){
                    q.offer(new Tuple(node.right,x+1,y+1));
                }
            }
        List<List<Integer>> list=new ArrayList<>();
            for (TreeMap<Integer,PriorityQueue<Integer>> ys: map.values()){
                list.add(new ArrayList<>());
                    for (PriorityQueue<Integer> nodes: ys.values()) {
                        while(!nodes.isEmpty()) {
                            list.get(list.size()-1).add(nodes.poll());
                        }
                }
            }
        return list;
        }
    }
}
//**************************Root  to Node Path in a Binary
Tree*************************************************************************************************
************** */
class Solution{
    public ArrayList<Integer> solve(TreeNode A, int B) {
        ArrayList<Integer> ans=new ArrayList<>();
        if(A==null){
        return ans;
        }
        boolean res=getPath(A,ans,B);
        return res==true?ans:new ArrayList<>();
    }
    public boolean getPath(TreeNode A,ArrayList<Integer> ans,int B){
        if(A==null){
        return false;
        }
```

```
        ans.add(A.val);
        if(A.val==B){
        return true;
        }

        if(getPath(A.left,ans,B)||getPath(A.right,ans,B)){
        return true;
        }
        ans.remove(ans.size()-1);
        return false;
    }
}
//**************************Maximum Width Of a Binary
Tree*************************************************************************************************
******************** */
class Tree{
    class Pair {
        TreeNode node;
        int num;
        Pair(TreeNode _node,int _num){
        node=_node;
        num=_num;
        }
    }
    class Solution {
    public int widthOfBinaryTree(TreeNode root) {
        if(root==null){
            return 0;
        }
        int ans=0;
        Queue<Pair> q=new LinkedList<Pair>();
        q.offer(new Pair(root,0));
        while(!q.isEmpty()){
            int size=q.size();
            int mmin=q.peek().num;
            int first=0,last=0;
            for(int i=0;i<size;i++){
                int cur_id=q.peek().num-mmin;
                TreeNode node=q.peek().node;
                q.poll();
                if(i==0){first=cur_id;}
                if(i==size-1){last=cur_id;}
                if(node.left!=null){
                    q.offer(new Pair(node.left,cur_id*2+1));
                }
                if(node.right!=null){
                    q.offer(new Pair(node.right,cur_id*2+2));
                }
            }
```

```
            ans=Math.max(ans,last-first+1);
            }
        return ans;
        }
    }
}
//**************************Pre/In/PostOrder
Traversal*********************************************************************************************
****************************** */
class Tree{
    class Pair {
        TreeNode node;
        int num;
        Pair(TreeNode _node, int _num) {
        num = _num;
        node = _node;
        }
    }
public class TUF {
    public static void allTraversal(TreeNode root,List<Integer> pre,List<Integer> in,
List<Integer> post){
        Stack<Pair> st=new Stack<Pair>();
        st.push(new Pair(root,1));

        if (root==null)
        return;

        while (!st.isEmpty()) {
        Pair it = st.pop();
        if(it.num == 1) {
            pre.add(it.node.val);
            it.num++;
            st.push(it);
            if(it.node.left!=null){
                st.push(new Pair(it.node.left,1));
            }
        }
        else if(it.num == 2){
            in.add(it.node.val);
            it.num++;
            st.push(it);

            if(it.node.right!=null){
                st.push(new Pair(it.node.right,1));
            }
        }
        else {
            post.add(it.node.val);
        }
```

```
        }
        }
    }
}

//**************************Binary Tree
I********************************************************************* */
//**************************Level Order
Traversal*****************************************************************************************
********************************** */
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        Queue<TreeNode> q=new LinkedList<TreeNode>();
        List<List<Integer>> wraplist=new LinkedList<List<Integer>>();
        if(root==null){
        return wraplist;
        }
        q.offer(root);
        while(!q.isEmpty()){
            int size=q.size();
            List<Integer> sublist=new LinkedList<Integer>();
            for(int i=0;i<size;i++){
                if(q.peek().left!=null){q.offer(q.peek().left);}
                if(q.peek().right!=null){q.offer(q.peek().right);}
                sublist.add(q.poll().val);
            }
            wraplist.add(sublist);
        }
        return wraplist;
    }
}
//**************************Height of a Binary
Tree*********************************************************************************************
******************************/
class Solution {
    public int maxDepth(TreeNode root) {
        if(root==null)
        return 0;
        int lh=maxDepth(root.left);
        int rh=maxDepth(root.right);

        return 1+Math.max(lh,rh);
        }
}
//**************************Diameter of a Binary
Tree*********************************************************************************************
****************************/
class Solution{
    public int diameterOfBinaryTree(TreeNode root){
```

```
        int[] diameter=new int[1];
        height(root,diameter);
        return diameter[0];
    }

    private int height(TreeNode node,int[] diameter){
        if(node==null){
        return 0;
        }
        int lh=height(node.left,diameter);
        int rh=height(node.right,diameter);
        diameter[0]=Math.max(diameter[0],lh+rh);
        return 1+Math.max(lh,rh);
    }
}
//*************************Check if a Binary Tree is Height
balanced********************************************** */
class Solution {
    public boolean isBalanced(TreeNode root) {
        return height(root)!=-1;
    }
    public int height(TreeNode root){
        if(root==null){
            return 0;
        }
        int lh=height(root.left);
        if(lh==-1){return -1;}

        int rh=height(root.right);
        if(rh==-1){return -1;}

        if(Math.abs(rh-lh)>1){
            return -1;
        }

        return 1+Math.max(height(root.left),height(root.right));

    }
}
//*************************LCA of Two Nodes(Lowest Common Ancestor)
*********************************************************************************************
*/
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if(root==null||root==p||root==q){
        return root;
        }
        TreeNode left=lowestCommonAncestor(root.left,p,q);
        TreeNode right=lowestCommonAncestor(root.right,p,q);
```

```
        if(left==null){
            return right;
        }
        else if(right==null){
            return left;
        }
        else{
            return root;
        }
    }
}
//**************************Check If Two Binary Trees Are
Equal*****************************************************************************************
********** */
class Solution {
    public boolean isSameTree(TreeNode p, TreeNode q) {
        if(p==null&&q==null){
            return true;
        }
        else if((p==null&&q!=null)||(p!=null&&q==null))
        {   return false;}
        return p.val==q.val&&isSameTree(p.left,q.left)&&isSameTree(p.right,q.right);
    }
}
//**************************Zig Zag
Traversal*************************************************************************************
******************************** */
class Solution {
    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        Queue<Node> queue=new LinkedList<Node>();
        ArrayList<ArrayList<Integer>> wrapList=new ArrayList<>();
        if (root == null) return wrapList;
        queue.offer(root);
        boolean flag = true;
        while (!queue.isEmpty()) {
            int levelNum = queue.size();
            ArrayList<Integer> subList=new ArrayList<Integer>(levelNum);
            for(int i=0;i<levelNum;i++){
                int index = i;
                if(queue.peek().left != null) queue.offer(queue.peek().left);
                if(queue.peek().right != null) queue.offer(queue.peek().right);
                if(flag == true) subList.add(queue.poll().val);
                else subList.add(0, queue.poll().val);
                }
        flag = !flag;
        wrapList.add(subList);
        }
        return wrapList;
```

```
    }
}
//*************************Boundary
Traversal*************************************************************************************
************************************ */
class Solution{
    static Boolean isLeaf(Node root) {
        return(root.left==null)&&(root.right==null);
    }
    static void addLeftBoundary(Node root,ArrayList<Integer> res){
        Node cur=root.left;
        while(cur!=null){
            if(isLeaf(cur)==false)
            res.add(cur.data);
            if(cur.left!=null)
                cur=cur.left;
            else
                cur=cur.right;
        }
    }
    static void addRightBoundary(Node root,ArrayList<Integer> res){
        Node cur=root.right;
        ArrayList<Integer> tmp=new ArrayList<Integer>();
        while(cur!=null){
            if(isLeaf(cur)==false)
             tmp.add(cur.data);
            if(cur.right!=null)
                cur=cur.right;
            else
                cur=cur.left;
        }
        int i;
        for(i=tmp.size()-1;i>=0;--i){
            res.add(tmp.get(i));
        }
    }
    static void addLeaves(Node root,ArrayList<Integer> res){
        if(isLeaf(root)){
            res.add(root.data);
            return;
        }
        if(root.left!=null)
            addLeaves(root.left,res);
        if(root.right!=null)
            addLeaves(root.right,res);
        }
    static ArrayList<Integer> printBoundary(Node node){
        ArrayList<Integer> ans=new ArrayList<Integer>();
        if(isLeaf(node)==false)
```

```
            ans.add(node.data);
            addLeftBoundary(node,ans);
            addLeaves(node,ans);
            addRightBoundary(node,ans);
            return ans;
        }
    }

//*************************Binary Tree-
III************************************************************ */
//*************************Maximum Path
Sum********************************************************************** */
class Solution{
    public int maxPathSum(TreeNode root) {
        int maxPS[]=new int[1];
        maxPS[0]=Integer.MIN_VALUE;
        maxPathSumNode(root,maxPS);
        return maxPS[0];
    }
    public int maxPathSumNode(TreeNode root,int maxPS[]){
        if(root==null){
            return 0;
        }
        int left=Math.max(0,maxPathSumNode(root.left,maxPS));
        int right=Math.max(0,maxPathSumNode(root.right,maxPS));
        maxPS[0]=Math.max(maxPS[0],left+right+root.val);
        return Math.max(left,right)+root.val;
    }
}
//*************************Build Tree From PreOrder and
InOrder************************************************** */
class Solution{
    public TreeNode buildTree(int[] preorder, int[] inorder) {
        HashMap<Integer,Integer> inMap=new HashMap<>();
        for(int i=0;i<inorder.length;i++){
            inMap.put(inorder[i],i);
        }
        TreeNode root=buildTreeRecur(preorder,0,preorder.length-1,inorder,0,inorder.
length-1,inMap);
        return root;
    }
    public TreeNode buildTreeRecur(int[] preorder,int preStart,int preEnd,int[] inorder,int
inStart,int inEnd,HashMap<Integer,Integer> inMap){
        if(preStart>preEnd||inStart>inEnd){
            return null;
        }
        TreeNode root=new TreeNode(preorder[preStart]);
        int inroot=inMap.get(root.val);
        int numsLeft=inroot-inStart;
```

```
        root.left=buildTreeRecur(preorder,preStart+1,preStart+numsLeft,inorder,inStart,
inroot-1,inMap);
        root.right=buildTreeRecur(preorder,preStart+numsLeft+1,preEnd,inorder,inroot+1,
inEnd,inMap);
        return root;
        }
    }
//*************************Build Tree from PostOrder and
InOrder************************************************************** */
class Solution{
public TreeNode buildTree(int[] inorder, int[] postorder) {
HashMap<Integer,Integer> inMap=new HashMap<>();
        for(int i=0;i<inorder.length;i++){
            inMap.put(inorder[i],i);
        }
        TreeNode root=buildTreeRecur(postorder,0,postorder.length-1,inorder,0,inorder.
length-1,inMap);
        return root;
    }
    public TreeNode buildTreeRecur(int[] postorder,int postStart,int postEnd,int[] inorder,
int inStart,int inEnd,HashMap<Integer,Integer> inMap){
        if(postStart>postEnd||inStart>inEnd){
            return null;
        }
        TreeNode root=new TreeNode(postorder[postEnd]);
        int inroot=inMap.get(root.val);
        int numsLeft=inroot-inStart;

        root.left=buildTreeRecur(postorder,postStart,postStart+numsLeft-1,inorder,inStart,
inroot-1,inMap);
        root.right=buildTreeRecur(postorder,postStart+numsLeft,postEnd-1,inorder,
inroot+1,inEnd,inMap);
        return root;
    }
}
//*************************Check if a Binary Tree is
Symmetric***************************************** */
class Solution {
    public boolean isSymmetric(TreeNode root) {
        return root==null||isSymmetricHelp(root.left,root.right);
    }
    public boolean isSymmetricHelp(TreeNode L,TreeNode R){
        if(L==null||R==null){
        return L==R;
        }
        if(L.val!=R.val){
        return false;
        }
```

```
        return isSymmetricHelp(L.left,R.right)&&isSymmetricHelp(L.right,R.left);
        }
    }
//*************************Mirror
Tree********************************************************************* */
class Solution {
// Function to convert a binary tree into its mirror tree.
    void mirror(Node node) {
        mirrorutil(node);
    }       // Your code here
    Node mirrorutil(Node node){
        if(node==null){
        return node;
        }
        Node L=mirrorutil(node.left);
        Node R=mirrorutil(node.right);

        node.left=R;
        node.right=L;

        return node;
        }
    }

//*************************Binary Search
Tree********************************************************************* */
//*************************Populate Next Pointer of a
Tree************************************************** */
class Solution {
    public Node connect(Node root) {
        Queue<Node> q = new LinkedList<>();
        q.add(root); // adding nodes to the queue
        Node temp=null; // initializing prev to null
        while(!q.isEmpty()){
            int n=q.size();
            for(int i=0;i<n;i++){
                Node prev=temp;
                temp=q.poll();
                if(i>0)
                    prev.next = temp;
                if(temp.left!=null)
                    q.add(temp.left);
                if(temp.right!=null)
                    q.add(temp.right);
            }
            temp.next=null;
        }
        return root;
    }
```

```
}
//*************************Search in a Binary Search
Tree************************************************** */
class Solution {
    public TreeNode searchBST(TreeNode root, int val) {
        TreeNode temp=root;
        while(temp!=null &&temp.val!=val){
            temp=temp.val>val?temp.left:temp.right;
        }
        return temp;
    }
}
//*************************Construct A BST From Preorder
traversal********************************************** */
class Solution {
    public TreeNode bstFromPreorder(int[] preorder) {
        return bstFromPreOrderUtil(preorder,Integer.MAX_VALUE,new int[]{0});
    }
    public TreeNode bstFromPreOrderUtil(int A[],int bound,int k[]){
        if(k[0]==A.length||A[k[0]]>bound)
            return null;
        TreeNode root=new TreeNode(A[k[0]++]);
        root.left=bstFromPreOrderUtil(A,root.val,k);
        root.right=bstFromPreOrderUtil(A,bound,k);
        return root;
    }
}
//*************************Check if a Binary Tree is a BST or
not****************************************************** */
class Solution {
    public boolean isValidBST(TreeNode root) {
        return isValidBSTUtil(root,Integer.MIN_VALUE,Integer.MAX_VALUE);
    }
    public boolean isValidBSTUtil(TreeNode root,int min,int max){
        if(root==null){
            return true;
        }
        if(root.val>max||root.val<min){return false;}
        return isValidBSTUtil(root.left,min,root.val)&&isValidBSTUtil(root.right,root.val,max)
;
    }
}
//*************************LCA In a Binary Search
Tree************************************************************** */
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if(root==null){
            return null;
        }
```

```
        int cur=root.val;
        if(cur>p.val&&cur>q.val){
            return lowestCommonAncestor(root.left,p,q);
        }
        else if(cur<p.val&&cur<q.val){
            return lowestCommonAncestor(root.right,p,q);
        }
        return root;
    }
}
//*************************Convert Sorted Array to
BST************************************************** */
class Solution {
    public TreeNode sortedArrayToBST(int[] nums) {
        TreeNode root=createArraytoBST(nums,0,nums.length-1);
        return root;
    }
    public TreeNode createArraytoBST(int []nums,int start,int end){
        if(start>end){
            return null;
        }
        int mid=(start+end)/2;
        TreeNode root=new TreeNode(nums[mid]);
        root.left=createArraytoBST(nums,start,mid-1);
        root.right=createArraytoBST(nums,mid+1,end);
        return root;
    }
}

//*************************Binary Search Tree -
II************************************************************************ */
//*************************Floor in a
BST********************************************************************************************** */
public class Solution {

    public static int floorInBST(TreeNode<Integer> root, int X) {
    //    Write your code here.
        int floor=-1;
        TreeNode curr=root;
        while(curr!=null){
            if(curr.data==X){
                floor=X;
                return floor;
            }
            else if(X>curr.data){
                floor=curr.data;
                curr=curr.right;
            }
            else{
```

```
            curr=curr.left;
          }
       }
       return floor;
    }
}
//*************************Ceil in a BST********************************************************* */
public class Solution {
    public  static int findCeil(TreeNode<Integer> node, int x) {
    int ceil=-1;
    TreeNode<Integer> curr=node;
    while(curr!=null){
       if(curr.data==x){
          ceil=x;
          return x;
       }
       else if(x<curr.data){
          ceil=curr.data;
          curr=curr.left;
       }
       else{
          curr=curr.right;
       }
    }
    return ceil;
    }
}
//*************************Get Kth Largest Smallest in a BST
*******************************************/
class Solution {
    public int kthSmallest(TreeNode root, int k) {
       int res[]=new int[]{0};
       int count[]=new int[]{0};
       getKthSmallest(root,res,count,k);
       return res[0];
    }
    public void getKthSmallest(TreeNode root,int res[],int count[],int k){
       if(root==null){
       return ;
       }
       getKthSmallest(root.left,res,count,k);
       count[0]++;
       if(count[0]==k){
       res[0]=root.val;
       return;
       }
       getKthSmallest(root.right,res,count,k);
       }
}
```

```
//************************Get Kth Largest Element in a
BST************************************************** */
class Solution
{
// return the Kth largest element in the given BST rooted at 'root'
    public int kthLargest(Node root,int K)
    {
    //Your code here
        int count[]=new int[]{0};
        int res[]=new int[]{0};
        int S=getSizeUtil(root);
        getKthLargestElement(root,count,S-K+1,res);
        return res[0];

    }
    public void getKthLargestElement(Node root,int count[],int k,int res[]){
        if(root==null){
        return ;
        }
        getKthLargestElement(root.left,count,k,res);
        count[0]++;
        if(count[0]==k){
        res[0]=root.data;
        return;
        }
        getKthLargestElement(root.right,count,k,res);

    }
    public int getSizeUtil(Node root){
        if(root==null){
        return 0;
        }
        return getSizeUtil(root.left)+getSizeUtil(root.right)+1;
    }
}
//************************BST Iterator************************************** */
class BSTIterator {
    public Stack<TreeNode> stack=new Stack<>();
    public BSTIterator(TreeNode root) {
        pushAll(root);
    }

    public int next() {
        TreeNode tmpNode=stack.pop();
        pushAll(tmpNode.right);
        return tmpNode.val;
    }

    public boolean hasNext() {
```

```
        return !stack.isEmpty();
    }
    public void pushAll(TreeNode root){
        for(;root!=null;stack.push(root),root=root.left);
    }
}
//************************Size of Maximum BST in a BT*********************************** */
class BST{
    class NodeValue {
    public int maxNode, minNode, maxSize;

    NodeValue(int minNode, int maxNode, int maxSize) {
    this.maxNode = maxNode;
    this.minNode = minNode;
    this.maxSize = maxSize;
    }
    };
class Solution {
    private NodeValue largestBSTSubtreeHelper(TreeNode root) {
        if(root == null){
        return new NodeValue(Integer.MAX_VALUE,Integer.MIN_VALUE,0);
        }
        NodeValue left=largestBSTSubtreeHelper(root.left);
        NodeValue right=largestBSTSubtreeHelper(root.right);
        if(left.maxNode < root.val && root.val < right.minNode){
        return new NodeValue(Math.min(root.val,left.minNode),Math.max(root.val,right.
maxNode),left.maxSize+right.maxSize+1);
        }

        return new NodeValue(Integer.MIN_VALUE,Integer.MAX_VALUE,Math.max(left.
maxSize,right.maxSize));
    }
    public int largestBSTSubtree(TreeNode root) {
        return largestBSTSubtreeHelper(root).maxSize;
    }
    }
}
//************************Serialize and Deserealize a Binary
tree********************************************** */
class Solution{
    public String serialize(TreeNode root) {
        if(root==null){
        return "";
        }
        Queue<TreeNode> q=new LinkedList<>();
        StringBuilder res=new StringBuilder();
        q.add(root);
        while(!q.isEmpty()){
        TreeNode temp=q.poll();
```

```
        if(temp==null){
        res.append("n ");
        continue;
        }
        res.append(temp.val+" ");
        q.add(temp.left);
        q.add(temp.right);
        }
        return res.toString();
    }

// Decodes your encoded data to tree.
    public TreeNode deserialize(String data) {
        if(data=="")
        return null;
        Queue<TreeNode> q=new LinkedList<>();
        String[] values=data.split(" ");
        TreeNode root=new TreeNode(Integer.parseInt(values[0]));
        q.add(root);
            for (int i=1;i<values.length;i++){
                TreeNode parent=q.poll();
                if(!values[i].equals("n")){
                TreeNode left=new TreeNode(Integer.parseInt(values[i]));
                parent.left=left;
                q.add(left);
                }
                if(!values[++i].equals("n")){
                TreeNode right=new TreeNode(Integer.parseInt(values[i]));
                parent.right=right;
                q.add(right);
        }
        }
    return root;
    }
}
```