```
public class Striver_TreesPlayList {
   //**********L1.Introduction to Trees | Types of
Trees***********************************************************/
   //**Types of Trees- Make Notes */
   //**********L2.Binary Tree Representation in C++
************************************************************/
   //**********L3.Binary Tree Representation in Java
************************************************************/
   class Solution{
      class TreeNode{
         int val;
         TreeNode left;
         TreeNode right;

         TreeNode(int _val){
            this.val=_val;
         }
      }
   }
   //**********L4.Binary Tree Traversals in Binary Tree|BFS|DFS
************************* **********************/
   class Solution{
      DFS:
      Preorder
      Inorder
      Postorder
      BFS:
      LevelOrder
   }
   //**********L5.Preorder 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);
      }
   }
   //**********L6.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);
    }
}
//**********L7.Postorder 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);
    }
}
//**********L8.Level Order Traversal of Binary
Tree************************************* */
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;
    }
}
//***********L9.Iterative Preorder Traversal*********************************
class Solution {
    public List<Integer> preorderTraversal(TreeNode root) {
     List<Integer> preorder=new ArrayList<>();
     if(root==null)return preorder;
     Stack<TreeNode> st=new Stack<TreeNode>();
     st.push(root);
     while(!st.isEmpty()){
        root=st.pop();
        preorder.add(root.val);
        if(root.right!=null){
            st.push(root.right);
        }
        if(root.left!=null){
            st.push(root.left);
        }
     }
     return preorder;
    }
}
//***********L10.Iterative Inorder Traversal*********************************/
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
    List<Integer> inorder=new ArrayList<>();
    Stack<TreeNode> stack=new Stack<TreeNode>();
    TreeNode node=root;
    while(true){
        if(node!=null){
            stack.push(node);
            node=node.left;
        }
        else{
            if(stack.isEmpty()){
                break;
            }
            node=stack.pop();
            inorder.add(node.val);
            node=node.right;
         }
        }
     return inorder;
}
}
//***********L11.Iterative Postorder Traversal using 2
```

```
Stack*********************************/
   class Solution {
      public List<Integer> postorderTraversal(TreeNode root) {
      List<Integer> postorder=new ArrayList<>();
      Stack<TreeNode> st1=new Stack<TreeNode>();
      Stack<TreeNode> st2=new Stack<TreeNode>();
      if(root==null)return postorder;

      st1.push(root);
      while(!st1.isEmpty()){
         root=st1.pop();
         st2.add(root);
         if(root.left!=null)st1.push(root.left);
         if(root.right!=null)st1.push(root.right);
      }
      while(!st2.isEmpty()){
         postorder.add(st2.pop().val);
      }
      return postorder;
      }
   }
   //**********L12.Iterative Postorder Traversal using 1
Stack*********************************/
   class Solution {
      public List<Integer> postorderTraversal(TreeNode root) {
      List<Integer> postorder=new ArrayList<>();
      Stack<TreeNode> st1=new Stack<TreeNode>();
      TreeNode cur=root;
      while(cur!=null||!st1.isEmpty()){
         if(cur!=null){
            st1.push(cur);
            cur=cur.left;
         }
         else{
            TreeNode temp=st1.peek().right;
            if(temp==null){
               temp=st1.peek();
               st1.pop();
               postorder.add(temp.val);
               while(!st1.isEmpty()&&temp==st1.peek().right){
                  temp=st1.peek();
                  st1.pop();
                  postorder.add(temp.val);
               }
            }
            else{cur=temp;}
         }
      }
            return postorder;
```

```
    }
    }
    //**********L13.Preorder Inorder Postorder Traversals in One
Traversal*******************/

    //**********L14.Maximum Depth in Binary Tree | Height of Binary
Tree*******************/
    class Solution{
        public int heightOfBinaryTree(TreeNode root){
            if(root==null){
                return 0;
            }
            int lh=heightOfBinaryTree(root.left);
            int rh=heightOfBinaryTree(root.right);
            return 1+Math.max(lh,rh);
        }
    }
    //**********L15.Check for Balanced Binary Tree*******************/
    class Solution{
        public boolean isBalanced(TreeNode root){
            return dfsHeight(root)!=-1
        }
        public int dfsHeight(TreeNode root){
            if(root==null){
                return 0;
            }
            int lh=dfsHeight(root.left);
            if(lh==-1)return -1;
            int rh=dfsHeight(root.right);
            if(rh==-1)return -1;
            if(Math.abs(lh-rh)>1)return -1;
            return 1+Math.max(lh,rh);
        }
    }
    //**********L16.Diameter of Binary Tree*******************/
    class Solution{
        public int diameterOfBinaryTree(TreeNode root){
            int []diameter=new int[]{0};
            heightOfBinaryTree(root,diameter);
            return diameter[0];
        }
        public int heightOfBinaryTree(TreeNode root,int diameter[]){
            if(root==null){
                return 0;
            }
            int lh=heightOfBinaryTree(root.left,diameter);
            int rh=heightOfBinaryTree(root.right,diameter);
            diameter[0]=Math.max(diameter[0],lh+rh);
            return 1+Math.max(lh,rh);
```

```
    }
  }
  //**************L17.Maximum Path Sum in Binary Tree*******************/
  class Solution{
    public static int maxPathSum(Node root){
      int maxValue[]=new int[1];
      maxValue[0]=Integer.MIN_VALUE;
      maxPathDown(root,maxValue);
      return maxValue[0];
    }

    public static int maxPathDown(Node node,int maxValue[]){
      if (node==null) return 0;
      int left=Math.max(0,maxPathDown(node.left,maxValue));
      int right=Math.max(0,maxPathDown(node.right,maxValue));
      maxValue[0]=Math.max(maxValue[0],left+right+node.val);
      return Math.max(left,right)+node.val;
    }
  }
  //**************L18.Check it two trees are Identical or Not*******************/
  class Solution {
    static boolean isIdentical(Node node1,Node node2){
      if(node1==null&&node2==null)
        return true;
      else if(node1==null||node2==null)
        return false;

      return ((node1.data==node2.data)&&isIdentical(node1.left,node2.left)
&&isIdentical(node1.right,node2.right));
    }
  }
  //**************L19.Zig-Zag or Spiral Traversal in Binary Tree*******************/
  class Solution {
    public static ArrayList<ArrayList<Integer>> zigzagLevelOrder(Node 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;
    }
}
//**************L20.Boundary Traversal in Binary Tree*******************/
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;
        }
    }
    //**************L21.Vertical Order Traversal of Binary Tree********************/
    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;
            }
        }
        public List<List<Integer>> verticalTraversal(TreeNode root){
            List<List<Integer>> list=new ArrayList<>();
            TreeMap<Integer,TreeMap<Integer,PriorityQueue<Integer>>> map=new
TreeMap<>();
            q.offer(new Tuple(root,0,0));
            while(!q.isEmpty()){
                Tuple tup=q.peek();
                TreeNode Node=tup.node;
                int x=tup.row;
                int y=tup.col;

                if(!map.containsKey(x)){
                    map.put(x,new TreeMap<>());
                }
                if(map.get(x).containsKey(y)){
                map.get(x).put(y,new PriorityQueue<Integer>());
                }
                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.left,x+1,y+1));
                }
            }
            for(TreeMap<Integer,PriorityQueue<Integer>> ys: map.values()){
                list.add(new ArrayList<>());
                for(PriorityQueue<Integer>> nodes: ys.values())
                    while(!nodes.isEmpty()){
                        System.out.println(nodes.peek());
                        list.get(list.size()-1).add(nodes.poll());
                    }
            }
```

```
            return list;
        }
    }
    //**************L22.Top View of Binary Tree*******************/
    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;

    }
}
    //**************L23.Bottom View of Binary Tree*******************/
    class Solution{
        static ArrayList<Integer> BottomView(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;
                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;
```

```
    }
}
    //**************L24.Right View of Binary Tree*******************/
    class Solution{
        public List<Integer> rightSideView(TreeNode root) {
            List<Integer> result=new ArrayList<Integer>();
            rightView(root,result,0);
            return result;
        }

        public void rightView(TreeNode curr,List<Integer> result,int currDepth){
            if(curr==null){return;}
            if(currDepth==result.size()){result.add(curr.val);}
            rightView(curr.right,result,currDepth+1);
            rightView(curr.left,result,currDepth+1);
        }
        public List<Integer> lightSideView(TreeNode root) {
            List<Integer> result = new ArrayList<Integer>();
            leftView(root,result,0);
            return result;
        }

        public void leftView(TreeNode curr,List<Integer> result,int currDepth){
            if(curr==null){return;}
            if(currDepth==result.size()){result.add(curr.val);}
            leftView(curr.left,result,currDepth + 1);
            leftView(curr.right,result,currDepth + 1);
        }
    }
    //**************L25.Check for Symmetrical Binary Tree*******************/
    class Solution{
        public boolean isSymmetric(TreeNode root){
            return root==null||isSymmetricHelp(root.left,root.right);
        }
        private boolean isSymmetricHelp(TreeNode left,TreeNode right){
            if(left==null||right==null)return left==right;
            if(left.val!=right.val)return false;
            return isSymmetricHelp(left.left,right.right)&&isSymmetricHelp(left.right,right.left);
        }
    }
    //**************L26.Print Root to Node Path of Binary Tree*******************/
    class Solution{
        static boolean getPath(Node root, ArrayList < Integer > arr, int x) {
            if(root==null)return false;
            arr.add(root.data);
            if(root.data==x)
                return true;
            if(getPath(root.left,arr,x)||getPath(root.right,arr,x))
                return true;
```

```
            arr.remove(arr.size()-1);
            return false;
        }
    }
    //**************L27.Lowest Common Ancestor of two nodes in Binary
Tree********************/
    class Solution{
        public TreeNode lowestCommonAncestor(TreeNode root,TreeNode p,TreeNode q)
{
            //base case
            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 { //both left and right are not null, we found our result
                return root;
            }
        }
    }
    //**************L28.Maximum Width of Binary Tree********************/
    class Solution{
        class Pair{
            TreeNode node;
            int num;
            Pair(TreeNode _node,int _num) {
                num=_num;
                node=_node;
            }
        }
        public static int widthOfBinaryTree(TreeNode root){
            if(root==null)return 0;
            int ans=0;
            Queue<Pair> q=new LinkedList<>();
            q.offer(new Pair(root,0));
            while(!q.isEmpty()){
                int size=q.size();
                int mmin=q.peek().num;    //to make the id starting from zero
                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;
    }
}
//**************L29.Children Sum Property*******************/
class Solution{
    static void reorder(Node root){
      if (root==null) return;
      int child=0;
      if(root.left!=null){
        child+=root.left.data;
      }
      if(root.right!=null){
        child+=root.right.data;
      }

      if(child<root.data){
        if(root.left!=null) root.left.data=root.data;
        else if(root.right!=null) root.right.data=root.data;
      }

      reorder(root.left);
      reorder(root.right);

      int tot = 0;
      if (root.left!=null) tot+=root.left.data;
      if (root.right!=null) tot+=root.right.data;
      if (root.left!=null || root.right!=null)root.data=tot;
    }}
//**************L30.Print All Nodes at a distance K from a Node*******************/
class Solution{
    private void markParents(TreeNode root, Map<TreeNode,TreeNode>
parent_track,TreeNode target) {
        Queue<TreeNode> queue=new LinkedList<TreeNode>();
        queue.offer(root);
        while(!queue.isEmpty()) {
            TreeNode current=queue.poll();
            if(current.left!=null) {
                parent_track.put(current.left,current);
```

```
                queue.offer(current.left);
            }
            if(current.right!=null) {
                parent_track.put(current.right,current);
                queue.offer(current.right);
            }
        }
    }
    public List<Integer> distanceK(TreeNode root,TreeNode target,int k) {
        Map<TreeNode,TreeNode> parent_track=new HashMap<>();
        markParents(root,parent_track,root);
        Map<TreeNode,Boolean> visited=new HashMap<>();
        Queue<TreeNode> queue=new LinkedList<TreeNode>();
        queue.offer(target);
        visited.put(target,true);
        int curr_level=0;
        while(!queue.isEmpty()){ /*Second BFS to go upto K level from target node and
using our hashtable info*/
            int size=queue.size();
            if(curr_level == k) break;
            curr_level++;
            for(int i=0;i<size;i++) {
                TreeNode current=queue.poll();
                if(current.left!=null&&visited.get(current.left)==null) {
                    queue.offer(current.left);
                    visited.put(current.left,true);
                }
                if(current.right!=null&&visited.get(current.right)==null ) {
                    queue.offer(current.right);
                    visited.put(current.right,true);
                }
                if(parent_track.get(current)!=null&&visited.get(parent_track.get(current))
==null) {
                    queue.offer(parent_track.get(current));
                    visited.put(parent_track.get(current),true);
                }
            }
        }
        List<Integer> result=new ArrayList<>();
        while(!queue.isEmpty()) {
            TreeNode current=queue.poll();
            result.add(current.val);
        }
        return result;
    }
}
//**************L31.Minimum time taken to burn a Tree*******************/
class Solution{
    private static BinaryTreeNode<Integer>
```

```
bfsToMapParents(BinaryTreeNode<Integer> root,HashMap<BinaryTreeNode<Integer>,
 BinaryTreeNode<Integer>> mpp, int start){
        Queue<BinaryTreeNode<Integer>> q=new LinkedList<>();
        q.offer(root);
        BinaryTreeNode<Integer> res=new BinaryTreeNode<>(-1);
        while(!q.isEmpty()){
            BinaryTreeNode<Integer> node=q.poll();
            if(node.data==start)res=node;
            if(node.left!=null){
                mpp.put(node.left,node);
                q.offer(node.left);
            }
            if(node.right!=null){
                mpp.put(node.right,node);
                q.offer(node.right);
            }
        }
        return res;
    }
    private static int findMaxDistance(HashMap<BinaryTreeNode<Integer>,
BinaryTreeNode<Integer>> mpp,BinaryTreeNode<Integer> target){
        Queue<BinaryTreeNode<Integer>> q=new LinkedList<>();
        q.offer(target);
        HashMap<BinaryTreeNode<Integer>,Integer> vis=new HashMap<>();
        vis.put(target,1);
        int maxi=0;

        while(!q.isEmpty()){
            int sz=q.size();
            int fl=0;

            for(int i=0;i<sz;i++){
                BinaryTreeNode<Integer> node=q.poll();
                if(node.left!=null&&vis.get(node.left)==null){
                    fl=1;
                    vis.put(node.left,1);
                    q.offer(node.left);
                }
                if(node.right!=null&&vis.get(node.right)==null){
                    fl=1;
                    vis.put(node.right,1);
                    q.offer(node.right);
                }
                if(mpp.get(node)!=null&&vis.get(mpp.get(node))==null){
                    fl=1;
                    vis.put(mpp.get(node),1);
                    q.offer(mpp.get(node));
                }
            }
```

```
            if(fl==1)maxi++;
        }
        return maxi;
    }
    public static int timeToBurnTree(BinaryTreeNode<Integer> root, int start)
    {
        HashMap<BinaryTreeNode<Integer>, BinaryTreeNode<Integer>> mpp=new
HashMap<>();
        BinaryTreeNode<Integer> target=bfsToMapParents(root, mpp, start);
        int maxi=findMaxDistance(mpp, target);
        return maxi;
    }
}
    //**************L32.Count Total Nodes in a Binary Tree*******************/
    class Solution{
        public int countNodes(TreeNode root) {
            if(root==null)return 0;
            int left=getHeightLeft(root);
            int right=getHeightRight(root);
            //If left and right are equal it means that the tree is complete and hence go for
2^h -1.
            if(left==right)return ((2<<(left)) -1);
            //else recursively calculate the number of nodes in left and right and add 1 for
root.
            else return countNodes(root.left)+countNodes(root.right)+1;
        }
        public int getHeightLeft(TreeNode root){
            int count=0;
            while(root.left!=null){
                count++;
                root=root.left;
            }
            return count;
        }
        public int getHeightRight(TreeNode root){
            int count=0;
            while(root.right!=null){
                count++;
                root=root.right;
            }
            return count;
        }
    }
    //**************L33.Requirements to construct a unique Binary Tree*******************/
    class Solution{
        //You cannot construct a unique tree from pre and post order
    }
    //**************L34.Construct a Binary Tree from PreOrder and
InOrder*******************/
```

```
   class Solution{
      static TreeNode buildTree(int[] preorder,int[] inorder) {
        Map<Integer,Integer> inMap=new HashMap<Integer,Integer>();
        for (int i=0;i<inorder.length;i++) {
          inMap.put(inorder[i],i);
        }
        TreeNode root=buildTree(preorder,0,preorder.length - 1,inorder,0,inorder.length-
1,inMap);
        return root;
      }
      static TreeNode buildTree(int[] preorder,int preStart,int preEnd,int[]inorder,int
inStart,int inEnd,Map < 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=buildTree(preorder,preStart+1,preStart+numsLeft,inorder,inStart,inRoot-
1,inMap);
        root.right=buildTree(preorder,preStart+numsLeft+1,preEnd,inorder,inRoot+1,
inEnd,inMap);
        return root;
      }
   }
   //*************L35.Construct a Binary Tree from PostOrder and
InOrder********************/
   class Solution{
      public TreeNode buildTree(int[] inorder,int[] postorder){
         if (inorder==null||postorder==null||inorder.length!=postorder.length)
            return null;
         HashMap<Integer,Integer> hm=new HashMap<Integer,Integer>();
         for (int i=0;i<inorder.length;++i)
            hm.put(inorder[i],i);
         return buildTreePostIn(inorder,0,inorder.length-1,postorder,0,postorder.length-1,
hm);
      }

      private TreeNode buildTreePostIn(int[] inorder,int is,int ie,int[] postorder,int ps,int
pe,HashMap<Integer,Integer> hm){
         if(ps>pe||is>ie) return null;
         TreeNode root=new TreeNode(postorder[pe]);
         int inroot=hm.get(postorder[pe]);
         int numsLeft=inroot-is;
         root.left=buildTreePostIn(inorder,is,inroot-1,postorder,ps,ps+numsLeft-1,hm);
         root.right=buildTreePostIn(inorder,inroot+1,ie,postorder,ps+numsLeft,pe-1,hm);
         return root;
      }
   }
   //**************L36.Serialize and DeSerialize 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 node=q.poll();
            if(node==null) {
                res.append("n ");
                continue;
            }
            res.append(node.val+" ");
            q.add(node.left);
            q.add(node.right);
        }
        return res.toString();
    }

    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;
    }
}
//**************L37.Morris Traversal********************/
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> inorder=new ArrayList<Integer>();
        TreeNode cur=root;
        while(cur!=null){
            if(cur.left==null){
                inorder.add(cur.val);
                cur=cur.right;
```

```
            }
            else{
                TreeNode prev=cur.left;
                while(prev.right!=null&&prev.right!=cur){
                    prev=prev.right;
                }
                if(prev.right==null) {
                    prev.right=cur;
                    cur=cur.left;
                }
                else{
                    prev.right=null;
                    inorder.add(cur.val);
                    cur=cur.right;
                }
            }
        }
        return inorder;
    }
    static ArrayList<Integer> preorderTraversal(Node root){
        ArrayList<Integer> preorder=new ArrayList<>();
        Node cur=root;
        while(cur!=null){
            if(cur.left==null){
                preorder.add(cur.data);
                cur=cur.right;
            } else {
                Node prev=cur.left;
                while(prev.right!=null&&prev.right!=cur) {
                    prev = prev.right;
                }

                if(prev.right==null){
                    prev.right=cur;
                    preorder.add(cur.data);
                    cur=cur.left;
                }else{
                    prev.right=null;
                    cur=cur.right;
                }
            }
        }
        return preorder;
    }
}
//**************L38.Flatten a Binary Tree to a Linked List*******************/
class Solution {
    static Node prev=null;
    static void flatten(Node root){
```

```
    if(root==null)return;
    flatten(root.right);
    flatten(root.left);
    root.right=prev;
    root.left=null;
    prev=root;
  }
static Node prev=null;
static void flatten(Node root){
    if(root==null)return;
    Stack<Node > st=new Stack<>();
    st.push(root);
    while(!st.isEmpty()){
        Node  cur=st.peek();
        st.pop();

        if(cur.right!=null){
        st.push(cur.right);
        }
        if(cur.left!=null){
        st.push(cur.left);
        }
        if(!st.isEmpty()){
        cur.right=st.peek();
        }
        cur.left=null;
    }
}
static ArrayList<Integer> preorderTraversal(Node root){
    ArrayList<Integer> preorder=new ArrayList<>();
    Node cur=root;
    while(cur!=null){
        if(cur.left==null){
            preorder.add(cur.data);
            cur=cur.right;
        }else{
            Node prev=cur.left;
            while(prev.right!=null&&prev.right!=cur){
                prev=prev.right;
            }

            if (prev.right==null){
                prev.right=cur;
                preorder.add(cur.data);
                cur=cur.left;
            }else{
                prev.right=null;
                cur=cur.right;
            }
```

```
            }
        }
        return preorder;
    }
}
//**************L39.Introduction to BST*****************************/
class Solution{
    Left < Node < Right;
    Left <= Node < Right;//Duplicates;
    Height =Log N
    Left Subtree= BST;
    Right Subtree= BST;
}
//**************L40.Search in a BST*********************************/
class Solution{
    //O(Log N)
    public TreeNode searchBST(TreeNode root, int val) {
        while(root!=null&&root.val!=val){
            root=val<root.val?root.left:root.right;
        }
        return root;
    }
}
//**************L41.Ceil in a BST***********************************/
class Solution {
    public static int findCeil(TreeNode<Integer> root,int key) {
        int ceil=-1;
        while(root!=null){
            if(root.data==key){
                ceil=root.data;
                return ceil;
            }

            if(key>root.data){
                root=root.right;
            }
            else{
                ceil=root.data;
                root=root.left;
            }
        }
        return ceil;
    }
}
//**************L42.Floor in a BST**********************************/
class Solution {
    public static int floorInBST(TreeNode<Integer> root,int key) {
        int floor=-1;
        while(root!=null){
```

```
            if(root.data==key){
                floor=root.data;
                return floor;
            }
            if(key>root.data) {
                floor=root.data;
                root=root.right;
            }
            else {
                root=root.left;
            }
        }
        return floor;
    }
}
//**************L43.Insert a given node in a BST*******************/
class Solution {
    public TreeNode insertIntoBST(TreeNode root,int val) {
        if(root==null) return new TreeNode(val);
        TreeNode curr=root;
        while(true){
            if(curr.val<=val){
                if(curr.right!=null)
                curr=cur.right;
                else{
                    curr.right=new TreeNode(val);
                    break;
                }
            }
            else{
                if(curr.left!=null)
                 curr=curr.left;
                else{
                    curr.left=new TreeNode(val);
                    break;
                }
            }
        }
        return root;
    }
}
//**************L44.Delete a given node in a BST*******************/
class Solution {
    public TreeNode deleteNode(TreeNode root,int key) {
        if(root==null) {
            return null;
        }
        if(root.val==key){
            return helper(root);
```

```
        }
        TreeNode dummy=root;
        while(root!=null) {
            if(root.val>key) {
                if (root.left!=null&&root.left.val==key){
                    root.left=helper(root.left);
                    break;
                }
                else{
                    root=root.left;
                }
            }
            else{
                if(root.right!=null&&root.right.val==key) {
                    root.right=helper(root.right);
                    break;
                }
                else{
                    root=root.right;
                }
            }
        }
        return dummy;
    }
    public TreeNode helper(TreeNode root) {
            if(root.left==null) {
                return root.right;
            }
            else if(root.right==null){
                return root.left;
            }
            else {
                TreeNode rightChild=root.right;
                TreeNode lastRight=findLastRight(root.left);
                lastRight.right=rightChild;
                return root.left;
            }
    }
    public TreeNode findLastRight(TreeNode root) {
        if(root.right==null){
            return root;
        }
        return findLastRight(root.right);
    }
}
//**************L45.Kth Smallest/Largest Element in BST*******************/
class Solution{
static Node kthlargest(Node root,int k[])
{
```

```
      if(root==null)return null;
      Node right=kthlargest(root.right,k);
      if(right!=null){return right;}
      k[0]--;
      if(k[0]==0){return root;}
      return kthlargest(root.left,k);
    }

    static Node kthsmallest(Node root,int k[])
    {
        if(root==null){return null;}
        Node left=kthsmallest(root.left,k);
        if(left!=null){return left;}
        k[0]--;
        if(k[0]==0){return root;}
        return kthsmallest(root.right,k);
    }
}
    //**************L46.Check if a tree is BST or BT********************/
    class Solution {
        private boolean checkBST(TreeNode node,long min,long max) {
            if(node==null)return true;
            if(node.val<=min||node.val>=max)return false;

            if(checkBST(node.left,min,node.val)&&checkBST(node.right,node.val,max)){
                return true;
            }
            return false;
        }
        public boolean isValidBST(TreeNode root) {
            return checkBST(root, Long.MIN_VALUE, Long.MAX_VALUE);
        }
    }
    //**************L47.LCA in a BST********************/
    class Solution {
        public TreeNode lowestCommonAncestor(TreeNode root,TreeNode p,TreeNode q)
{
            if(root==null)return null;
            int curr=root.val;
            if(curr<p.val&&curr<q.val) {
                return lowestCommonAncestor(root.right,p,q);
            }
            if(curr>p.val&&curr>q.val) {
                return lowestCommonAncestor(root.left,p,q);
            }
            return root;
        }
    }
    //**************L48.Construct BST from Preorder********************/
```

```
class Solution {
    public TreeNode bstFromPreorder(int[] A) {
        return bstFromPreorder(A,Integer.MAX_VALUE,new int[]{0});
    }
    public TreeNode bstFromPreorder(int[] A,int bound,int[] i){
        if(i[0]==A.length||A[i[0]]>bound) return null;
        TreeNode root=new TreeNode(A[i[0]++]);
        root.left=bstFromPreorder(A,root.val,i);
        root.right=bstFromPreorder(A,bound,i);
        return root;
    }
}
//**************L49.Inorder Successor/Predecessor*******************/
class Solution {
    public TreeNode inorderSuccessor(TreeNode root,TreeNode p){
        TreeNode successor = null;
        while (root != null) {
            if (p.val>=root.val) {
                root=root.right;
            }else{
                successor=root;
                root=root.left;
            }
        }
        return successor;
    }
    public TreeNode inorderPredecessor(TreeNode root,TreeNode p){
        TreeNode predecessor=null;
        while(root!=null) {
            if (p.val<=root.val) {
                root=root.left;
            }else{
                predecessor=root;
                root=root.right;
            }
        }
        return predecessor;
    }
}
//**************L50.BST Iterator*******************/
class Solution{
    public class BSTIterator {
        private Stack<TreeNode> stack = new Stack<TreeNode>();
        public BSTIterator(TreeNode root){
            pushAll(root);
        }
        /** @return whether we have a next smallest number */
        public boolean hasNext(){
            return !stack.isEmpty();
```

```
        }
        /** @return the next smallest number */
        public int next(){
            TreeNode tmpNode=stack.pop();
            pushAll(tmpNode.right);
            return tmpNode.val;
        }

        private void pushAll(TreeNode node) {
            for (;node!=null;stack.push(node),node=node.left);
        }
    }
}
//**************L51.Two Sum in a BST********************/
class Solution{
class BSTIterator {
    private Stack<TreeNode> stack=new Stack<TreeNode>();
    boolean reverse=true;

    public BSTIterator(TreeNode root,boolean isReverse){
        reverse=isReverse;
        pushAll(root);
    }
    /** @return whether we have a next smallest number */
    public boolean hasNext() {
        return !stack.isEmpty();
    }

    /** @return the next smallest number */
    public int next() {
        TreeNode tmpNode=stack.pop();
        if(reverse==false) pushAll(tmpNode.right);
        else pushAll(tmpNode.left);
        return tmpNode.val;
    }

    private void pushAll(TreeNode node){
        while(node!=null){
            stack.push(node);
            if(reverse==true){
                node=node.right;
            }else{
                node=node.left;
            }
        }
    }
}
class Solution {
    public boolean findTarget(TreeNode root,int k){
```

```
        if(root==null) return false;
        BSTIterator l=new BSTIterator(root,false);
        BSTIterator r=new BSTIterator(root,true);

        int i=l.next();
        int j=r.next();
        while(i<j){
            if(i+j==k) return true;
            else if(i+j<k) i=l.next();
            else j=r.next();
        }
        return false;
    }
  }
}
  //**************L52.Recover BST/Correct BST*******************/
  class Solution {
    private TreeNode first;
    private TreeNode prev;
    private TreeNode middle;
    private TreeNode last;
    private void inorder(TreeNode root){
        if(root==null) return;
        inorder(root.left);
        if (prev!=null&&(root.val<prev.val))
        {
            if(first==null)
            {
                first = prev;
                middle = root;
            }
            else
                last = root;
        }
        prev=root;
        inorder(root.right);
    }
    public void recoverTree(TreeNode root) {
        first=middle=last=null;
        prev=new TreeNode(Integer.MIN_VALUE);
        inorder(root);
        if(first!=null&&last!=null){
            int t=first.val;
            first.val=last.val;
            last.val=t;
        }
        else if(first!=null&&middle!=null){
            int t=first.val;
            first.val=middle.val;
```

```
            middle.val=t;
        }
    }

}
//**************L53.Largest BST in a BT********************/
class Solution {
    class NodeValue {
        public int maxNode,
        public int minNode,
        public int maxSize;
        NodeValue(int maxNode,int minNode,int maxSize){
            this.maxNode=maxNode;
            this.minNode=minNode;
            this.maxSize=maxSize;
        }
    }
    private NodeValue largestBSTSubtreeHelper(TreeNode root){
        if(root==null) {
            return new NodeValue(Integer.MIN_VALUE,Integer.MAX_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.max(root.val,right.maxNode), Math.min(root.val,
left.minNode),left.maxSize+right.maxSize+1);
        }
        return new NodeValue(Integer.MAX_VALUE,Integer.MIN_VALUE,Math.max(left.
maxSize, right.maxSize));
    }
    public int largestBSTSubtree(TreeNode root) {
        return largestBSTSubtreeHelper(root).maxSize;
    }
}

//**************************TAKE U FORWARD YOUTUBE TREES
PLAYLIST********************************************** */




}
```