**1)**

class Spiral

{

ArrayList<Integer> list = new ArrayList<>();

// Function to perform spiral traversal of the binary tree

public void spiral(Node root) {

if (root == null)

return;

// Create two stacks for alternate levels

Stack<Node> s1 = new Stack<>();

Stack<Node> s2 = new Stack<>();

// Push the root to the first stack

s1.push(root);

// Continue traversal until both stacks are empty

while (!s1.empty() || !s2.empty()) {

// Process nodes of current level from s1 and push children to s2

while (!s1.empty()) {

Node temp = s1.pop();

list.add(temp.data);

// Push right child before left child to ensure left child is processed first

if (temp.right != null)

s2.push(temp.right);

if (temp.left != null)

s2.push(temp.left);

}

// Process nodes of current level from s2 and push children to s1

while (!s2.empty()) {

Node temp = s2.pop();

list.add(temp.data);

// Push left child before right child to ensure right child is processed first

if (temp.left != null)

s1.push(temp.left);

if (temp.right != null)

s1.push(temp.right);

}

}

}

// Wrapper function to initialize traversal from the root

ArrayList<Integer> findSpiral(Node root)

{

spiral(root);

return list;

}

}

**2)**

class Solution {

    public boolean isCompleteTree(TreeNode root) {

        boolean flag = false;

        Queue<TreeNode> queue = new LinkedList<>();

        queue.offer(root);

        while(!queue.isEmpty()){

            TreeNode currentNode = queue.poll();

            if(currentNode == null){

                flag = true;

            }else{

                if(flag){

                    return false;

                }

                queue.offer(currentNode.left);

                queue.offer(currentNode.right);

            }

        }

        return true;

    }

}

**3)**

class Tree

{

Deque<Integer> deque = new LinkedList<>();

ArrayList<Integer> list = new ArrayList<>();

boolean level(Node node,int level){

if(node==null){

return false;

}

//Add elem into deque at level 1

if(level==1){

deque.addFirst(node.data);

return true;

}

boolean right = level(node.right, level-1);

boolean left = level(node.left, level-1);

return left||right;

}

public ArrayList<Integer> reverseLevelOrder(Node node){

int level = 1;

while(level(node , level)){

level++;

}

while(!deque.isEmpty()){

int temp = deque.removeFirst();

list.add(temp);

}

return list;

}

}

**4)**

class Tree

{

//Function to return list containing elements of left view of binary tree.

ArrayList<Integer> list = new ArrayList<>();

void nodeLevel(Node root,int level){

if(root==null){

return;

}if(level>list.size()){

list.add(root.data);

}

nodeLevel(root.left,level+1);

nodeLevel(root.right,level+1);

}

ArrayList<Integer> leftView(Node root)

{

nodeLevel(root,1);

return list;

}

}

**5)**

class Solution {

// Function to convert a binary tree into its mirror tree.

void mirror(Node node) {

// Your code here

if(node==null){

return;

}

Node temp = node.left;

node.left = node.right;

node.right = temp;

mirror(node.left);

mirror(node.right);

}

void inorder(Node node){

if(node==null){

return;

}

inorder(node.left);

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

inorder(node.right);

}

}