

Cameras In Binary Tree | Leetcode 968 Binary Tree Cameras

```
public static int MinCamerasInBT(TreeNode root) {  
    if(MinCamerasInBT_(root) == -1) cameras++;  
    return cameras;  
}
```

// MinCamerasInBT_

```
}  
public static int cameras = 0;  
  
public static int MinCamerasInBT_(TreeNode root) {  
    if(root == null) return 1;  
  
    int lchild = MinCamerasInBT_(root.left);  
    int rchild = MinCamerasInBT_(root.right);  
  
    if(lchild == -1 || rchild == -1){  
        cameras++;  
        return 0;  
    }  
  
    if(lchild == 0 || rchild == 0) return 1;  
    return -1;  
}  
  
public static int MinCamerasInBT(TreeNode root) {  
    if(MinCamerasInBT_(root) == -1) cameras++;  
    return cameras;  
}
```

House Robber In Binary Tree | Leetcode 337 House Robber III

```
public static class housePair{
    int withRobbery = 0;
    int withoutRobbery = 0;
}

public static housePair HouseRobber_(TreeNode root) {
    if(root==null) return new housePair();

    housePair left = HouseRobber_(root.left);
    housePair right = HouseRobber_(root.right);

    housePair myAns = new housePair();

    myAns.withRobbery = left.withoutRobbery + root.val + right
    .withoutRobbery;

    myAns.withoutRobbery = Math.max(left.withRobbery, left.withoutRobbery)
+ Math.max(right.withRobbery, right.withoutRobbery);

    return myAns;
}

public static int HouseRobber(TreeNode root) {
    housePair res = HouseRobber_(root);

    return Math.max(res.withRobbery, res.withoutRobbery);
}
```

other way

```
// {with-Robbery Maximum Amount, without-Robbery maximum Amount}
public static int[] HouseRobber_(TreeNode root) {
    if(root==null) return new int[2];

    int[] left = HouseRobber_(root.left);
    int[] right = HouseRobber_(root.right);

    int[] myAns= new int[2];

    myAns[0] = left[1] + root.val +right[1];
    myAns[1] = Math.max(left[0],left[1]) + Math.max(right[0],right[1]);
    return myAns;
}
```

Longest Zig Zag Path In Binary Tree | Leetcode 1372. Longest ZigZag Path in a Binary Tree

```
public static class pair{
    int forwardSlop = -1;
    int backwardSlop = -1;
    int maxLen = 0;
}

public static pair longestZigZagPath_(TreeNode root) {
    if(root == null) return new pair();

    pair left = longestZigZagPath_(root.left);
    pair right = longestZigZagPath_(root.right);

    pair myAns = new pair();
    myAns.maxLen = Math.max(Math.max(left.maxLen, right.maxLen), Math.max(
        left.backwardSlop, right.forwardSlop) + 1);

    myAns.forwardSlop = left.backwardSlop + 1;
    myAns.backwardSlop = right.forwardSlop + 1;

    return myAns;
}
```

```
36
37     static int maxLen = 0;
38
39     // {forwardSlop, backwardSlop}
40     public static int[] longestZigZagPath_02(TreeNode root){
41         if(root == null) return new int[]{-1, -1};
42
43         int[] left = longestZigZagPath_02(root.left);
44         int[] right = longestZigZagPath_02(root.right);
45
46         maxLen = Math.max(maxLen, Math.max(left[1], right[0]) + 1);
47         return new int[]{left[1] + 1, right[0] + 1};
48     }
49
50     public static int longestZigZagPath(TreeNode root) {
51         longestZigZagPath_02(root);
52         return maxLen;
53     }
54 }
```

Validate Bst

```
public static TreeNode prev = null;

public static boolean isValidBST(TreeNode root) {
    if(root == null) return true;

    if(!isValidBST(root.left)) return false;
    ↓ if(prev != null && prev.val > root.val) return false;
    prev = root;
    if(!isValidBST(root.right)) return false;

    return true;
}
```

recover bst

```
public static class TreeNode {
    int val = 0;
    TreeNode left = null;
    TreeNode right = null;

    TreeNode(int val) {
        this.val = val;
    }
}

static TreeNode a = null, b = null, prev = null;

public static boolean recoverTree_(TreeNode root) {

    if (root == null)
        return true;

    if (!recoverTree_(root.left))
        return false;

    if (prev != null && prev.val > root.val) {
        b = root;
        if (a == null)
            a = prev;
        else
            return false;
    }

    prev = root;
    if (!recoverTree_(root.right))
        return false;

    return true;
}

public static void recoverTree(TreeNode root) {
    recoverTree_(root);
    if (a != null) {
        int temp = a.val;
        a.val = b.val;
        b.val = temp;
    }
}
```


Serialize and Deserialize of Binary Tree Part 1 | Leetcode 2

```
public static void serialize(TreeNode root, StringBuilder sb){
    if(root == null){
        sb.append("null,");
        return;
    }

    sb.append(root.val+",");
    serialize(root.left, sb);
    serialize(root.right, sb);
}

// Encodes a tree to a single string.
public static String serialize(TreeNode root) {
    StringBuilder sb = new StringBuilder();
    serialize(root,sb);
    return sb.toString();
}
```

```
static int idx = 0;
public static TreeNode deserialize(String[] arr) {
    if(idx >= arr.length || arr[idx].equals("null")){
        idx++;
        return null;
    }

    TreeNode node = new TreeNode(arr[idx++]);
    node.left = deserialize(arr);
    node.right = deserialize(arr);

    return node;
}

// Decodes your encoded data to tree.
public static TreeNode deserialize(String str) {
    String[] arr = str.split(",");
    deserialize(arr);
}
```

left view

```
public static ArrayList<Integer> leftView(TreeNode root) {
    if(root == null) return null;
    ArrayList<Integer> ans = new ArrayList<>();
    LinkedList<TreeNode> que = new LinkedList<>();
    que.addLast(root);
    while(que.size() != 0){
        int size = que.size();
        ans.add(que.getFirst().val);
        while(size-->0){
            TreeNode rn = que.removeFirst(); // rn : remove node
            if(rn.left != null) que.addLast(rn.left);
            if(rn.right != null) que.addLast(rn.right);
        }
    }

    return ans;
}
```

right view

```
public static ArrayList<Integer> rightView(TreeNode root) {
    LinkedList<TreeNode> que = new LinkedList<>();
    que.addLast(root);
    ArrayList<Integer> ans = new ArrayList<>();

    while(que.size() != 0){
        int size = que.size();
        ans.add(que.getFirst().val);
        while(size-- > 0){
            TreeNode rn = que.removeFirst();

            if(rn.right != null) que.addLast(rn.right);
            if(rn.left != null) que.addLast(rn.left);
        }
    }

    return ans;
}
```

Width Of Shadow Of Binary Tree

```
// minimum, maximum, where hl is horizontal level.
public static void width(TreeNode root, int hl, int[] ans) {
    if (root == null) return;

    ans[0] = Math.min(ans[0], hl);
    ans[1] = Math.max(ans[1], hl);

    width(root.left, hl - 1, ans);
    width(root.right, hl + 1, ans);
}

public static int width(TreeNode root) {
    int[] ans = new int[2];
    width(root, 0, ans);
    return ans[1] - ans[0] + 1;
}
```



```

public static class verticalPair {
    TreeNode node = null;
    int hl = 0; // horizontal Level

    verticalPair(TreeNode node, int hl) {
        this.node = node;
        this.hl = hl;
    }
}

public static ArrayList<ArrayList<Integer>> verticalOrderTraversal(TreeNode root) {
    LinkedList<verticalPair> que = new LinkedList<>();
    que.addLast(new verticalPair(root, 0));
    HashMap<Integer, ArrayList<Integer>> map = new HashMap<>();

    int minHL = 0;
    int maxHL = 0;

    while (que.size() != 0) {
        int size = que.size();
        while (size-- > 0) {
            verticalPair rp = que.removeFirst();

            map.putIfAbsent(rp.hl, new ArrayList<>());
            // if (!map.containsKey(rp.hl))
            // map.put(rp.hl, new ArrayList<>());

            map.get(rp.hl).add(rp.node.val);

            minHL = Math.min(minHL, rp.hl);
            maxHL = Math.max(maxHL, rp.hl);

            if (rp.node.left != null)
                que.addLast(new verticalPair(rp.node.left, rp.hl - 1));

            if (rp.node.right != null)
                que.addLast(new verticalPair(rp.node.right, rp.hl + 1));
        }
    }

    ArrayList<ArrayList<Integer>> ans = new ArrayList<>();
    while (minHL <= maxHL) {
        ans.add(map.get(minHL));
        minHL++;
    }

    return ans;
}

```

```

public static class verticalPair {
    TreeNode node = null;
    int hl = 0; // horizontal Level

    verticalPair(TreeNode node, int hl) {
        this.node = node;
        this.hl = hl;
    }
}

```

```

// ans = {minHL,maxHL}
public static void width(TreeNode root, int hl, int[] ans) {
    if (root == null)
        return;

    ans[0] = Math.min(hl, ans[0]);
    ans[1] = Math.max(hl, ans[1]);

    width(root.left, hl - 1, ans);
    width(root.right, hl + 1, ans);
}

```

```

public static ArrayList<ArrayList<Integer>> verticalOrderTraversal(TreeNode root) {
    PriorityQueue<verticalPair> que = new PriorityQueue<>((a, b) -> {
        return a.node.val - b.node.val; // this - other for default behaviour
    });
    PriorityQueue<verticalPair> childQue = new PriorityQueue<>((a, b) -> {
        return a.node.val - b.node.val;
    });

    int[] minMax = new int[2];
    width(root, 0, minMax);
    int length = minMax[1] - minMax[0] + 1;
    ArrayList<ArrayList<Integer>> ans = new ArrayList<>();
    for (int i = 0; i < length; i++)
        ans.add(new ArrayList<>());

    que.add(new verticalPair(root, -minMax[0]));

    while (que.size() != 0) {
        verticalPair rp = que.remove();

        ans.get(rp.hl).add(rp.node.val);

        if (rp.node.left != null)
            childQue.add(new verticalPair(rp.node.left, rp.hl - 1));

        if (rp.node.right != null)
            childQue.add(new verticalPair(rp.node.right, rp.hl + 1));

        if (que.size() == 0) {
            PriorityQueue<verticalPair> temp = que;
            que = childQue;
            childQue = temp;
        }
    }

    return ans;
}

```

```

// you, 21 hours ago | 1 author (you)
class Pair implements Comparable<Pair> {
    TreeNode node;
    int key;
    int level = 0;

    public Pair() {

    }

    public Pair(TreeNode node, int key, int level) {
        this.node = node;
        this.key = key;
        this.level = level;
    }

    // make this note
    @Override
    public int compareTo(Pair obj) {
        if (this.level == obj.level)
            return this.node.val - obj.node.val;

        return this.level - obj.level;
    }
}

```

```

int min = 0;
int max = 0;

public void getWidth(TreeNode root, int level) {
    if (root == null) {
        return;
    }

    min = Math.min(min, level);
    max = Math.max(max, level);
    getWidth(root.left, level - 1);
    getWidth(root.right, level + 1);
}

```

```

public List<List<Integer>> verticalTraversal(TreeNode root) {

    // getwidth
    getWidth(root, level: 0);
    int width = max - min + 1;
    HashMap<Integer, ArrayList<Integer>> map = new HashMap<>();

    for (int i = 0; i < width; i++) {
        map.put(i, new ArrayList<>());
    }

    PriorityQueue<Pair> q = new PriorityQueue<>();
    q.add(new Pair(root, Math.abs(min), level: 0));
    while (q.size() > 0) {
        int size = q.size();
        while (size-- > 0) {
            Pair rem = q.remove();
            map.get(rem.key).add(rem.node.val);
            if (rem.node.left != null) {
                q.add(new Pair(rem.node.left, rem.key - 1, rem.level + 1));
            }
            if (rem.node.right != null) {
                q.add(new Pair(rem.node.right, rem.key + 1, rem.level + 1));
            }
        }
    }

    // make ans
    List<List<Integer>> ans = new ArrayList<>();

    for (int i = 0; i < width; i++) {
        ans.add(map.get(i));
    }

    return ans;
}

```

```

public static class verticalPair {
    TreeNode node = null;
    int hl = 0; // horizontal Level

    verticalPair(TreeNode node, int hl) {
        this.node = node;
        this.hl = hl;
    }
}

// ans = {minHL, maxHL}
public static void width(TreeNode root, int hl, int[] ans) {
    if (root == null)
        return;

    ans[0] = Math.min(hl, ans[0]);
    ans[1] = Math.max(hl, ans[1]);

    width(root.left, hl - 1, ans);
    width(root.right, hl + 1, ans);
}

public static ArrayList<Integer> BottomView(TreeNode root) {
    LinkedList<verticalPair> que = new LinkedList<>();

    int[] minMax = new int[2];
    width(root, 0, minMax);
    int length = minMax[1] - minMax[0] + 1;
    ArrayList<Integer> ans = new ArrayList<>();
    for (int i = 0; i < length; i++)
        ans.add(0);

    que.addLast(new verticalPair(root, -minMax[0]));

    while (que.size() != 0) {
        int size = que.size();
        while (size-- > 0) {
            verticalPair rp = que.removeFirst();

            ans.set(rp.hl, rp.node.val);

            if (rp.node.left != null)
                que.addLast(new verticalPair(rp.node.left, rp.hl - 1));

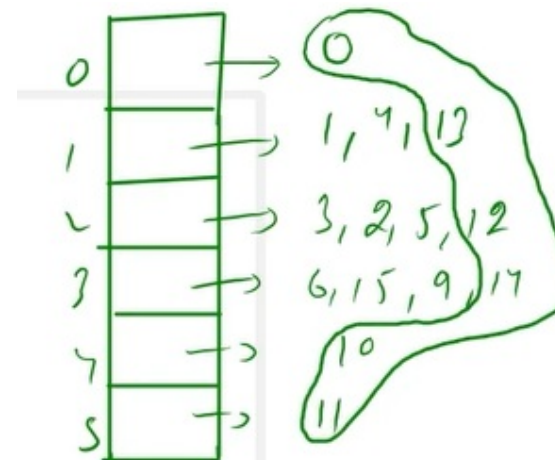
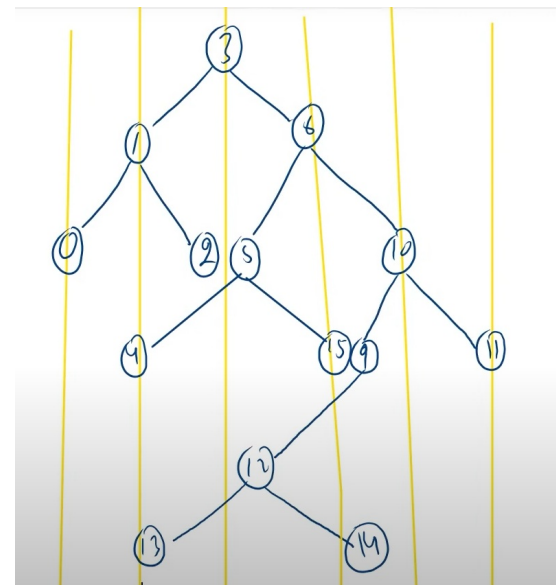
            if (rp.node.right != null)
                que.addLast(new verticalPair(rp.node.right, rp.hl + 1));
        }
    }

    return ans;
}

// Input section

```

only change this
just override



0, 13, 12, 14, 10, 11


```

public static class verticalPair {
    TreeNode node = null;
    int hl = 0; // horizontal Level

    verticalPair(TreeNode node, int hl) {
        this.node = node;
        this.hl = hl;
    }
}

// ans = {minHL,maxHL}
public static void width(TreeNode root, int hl, int[] ans) {
    if (root == null)
        return;

    ans[0] = Math.min(hl, ans[0]);
    ans[1] = Math.max(hl, ans[1]);

    width(root.left, hl - 1, ans);
    width(root.right, hl + 1, ans);
}

public static ArrayList<Integer> TopView(TreeNode root) {
    LinkedList<verticalPair> que = new LinkedList<>();

    int[] minMax = new int[2];
    width(root, 0, minMax);
    int length = minMax[1] - minMax[0] + 1;
    ArrayList<Integer> ans = new ArrayList<>();
    for (int i = 0; i < length; i++)
        ans.add((int) 1e9);

    que.addLast(new verticalPair(root, -minMax[0]));

    while (que.size() != 0) {
        int size = que.size();
        while (size-- > 0) {
            verticalPair rp = que.removeFirst();

            if (ans.get(rp.hl) == (int) 1e9)
                ans.set(rp.hl, rp.node.val);

            if (rp.node.left != null)
                que.addLast(new verticalPair(rp.node.left, rp.hl - 1));

            if (rp.node.right != null)
                que.addLast(new verticalPair(rp.node.right, rp.hl + 1));
        }
    }

    return ans;
}

```

backward diagonal

```
public static ArrayList<ArrayList<Integer>> diagonalOrder
(TreeNode root) {
    LinkedList<TreeNode> que = new LinkedList<>();
    ArrayList<ArrayList<Integer>> ans = new ArrayList<>();

    que.addLast(root);
    while(que.size() != 0){ // diagonal

        int size = que.size();
        ArrayList<Integer> smallAns = new ArrayList<>();

        while(size-->0){ // help to traverse each Component of
that particular diagonal.
            TreeNode rn = que.removeFirst();
            while(rn != null){ // traverse a component.
                smallAns.add(rn.val);
                if(rn.left != null) que.addLast(rn.left);

                rn = rn.right;
            }
        }

        ans.add(smallAns);
    }

    return ans;
}
```


forward diagonal

```
public static ArrayList<ArrayList<Integer>> diagonalOrder
(TreeNode root) {
    LinkedList<TreeNode> que = new LinkedList<>();
    ArrayList<ArrayList<Integer>> ans = new ArrayList<>();

    que.addLast(root);
    while(que.size() != 0){ // help to traverse each diagonal
        int size = que.size();
        ArrayList<Integer> smallAns = new ArrayList<>();

        while(size-->0){ // help to traverse all components.

            TreeNode rn = que.removeFirst();
            while(rn != null){ // help to traverse each component.
                smallAns.add(rn.val);
                if(rn.right != null) que.addLast(rn.right);
                rn = rn.left;
            }
        }

        ans.add(smallAns);
    }
    return ans;
}
```

```

public static class verticalPair {
    TreeNode node = null;
    int hl = 0; // horizontal Level

    verticalPair(TreeNode node, int hl) {
        this.node = node;
        this.hl = hl;
    }
}

// ans = {minHL,maxHL}
public static void width(TreeNode root, int hl, int[] ans) {
    if (root == null)
        return;

    ans[0] = Math.min(hl, ans[0]);
    ans[1] = Math.max(hl, ans[1]);

    width(root.left, hl - 1, ans);
    width(root.right, hl + 1, ans);
}

public static ArrayList<Integer> verticalOrderSum(TreeNode root) {
    LinkedList<verticalPair> que = new LinkedList<>();

    int[] minMax = new int[2];
    width(root, 0, minMax);
    int length = minMax[1] - minMax[0] + 1;
    ArrayList<Integer> ans = new ArrayList<>();
    for (int i = 0; i < length; i++)
        ans.add(0);

    que.addLast(new verticalPair(root, -minMax[0]));

    while (que.size() != 0) {
        int size = que.size();
        while (size-- > 0) {
            verticalPair rp = que.removeFirst();

            ans.set(rp.hl, ans.get(rp.hl) + rp.node.val);

            if (rp.node.left != null)
                que.addLast(new verticalPair(rp.node.left, rp.hl - 1));

            if (rp.node.right != null)
                que.addLast(new verticalPair(rp.node.right, rp.hl + 1));
        }
    }

    return ans;
}

```

```

public static void dfs(TreeNode root, int hl, ArrayList<Integer>
ans){
    if(root == null) return;

    ans.set(hl, ans.get(hl) + root.val);

    dfs(root.left, hl - 1, ans);
    dfs(root.right, hl + 1, ans);
}

public static ArrayList<Integer> verticalOrderSum(TreeNode root){

    int[] minMax = new int[2];
    width(root, 0, minMax);

    int len = minMax[1] - minMax[0] + 1;

    ArrayList<Integer> ans = new ArrayList<>();
    for(int i = 0; i < len; i++) ans.add(0);

    dfs(root, Math.abs(minMax[0]), ans);
    return ans;
}

```

```

public static ArrayList<Integer> diagonalOrderSum(TreeNode root) {
    LinkedList<TreeNode> que = new LinkedList<>();
    ArrayList<Integer> ans = new ArrayList<>();
    que.addLast(root);
    while(que.size() != 0){
        int size = que.size();
        int sum = 0;
        while(size-->0){
            TreeNode rn = que.removeFirst();
            while(rn != null){
                sum += rn.val;
                if(rn.left != null) que.addLast(rn.left);

                rn = rn.right;
            }
        }

        ans.add(sum);
    }
    return ans;
}

```

```

14     }
15
16     public static void dfs(TreeNode node,int diagNo,ArrayList<Integer>
17     ans){
18         if(node == null) return;
19
20         if(diagNo == ans.size()) ans.add(0);
21         ans.set(diagNo, ans.get(diagNo) + node.val);
22
23         dfs(node.left, diagNo + 1, ans);
24         dfs(node.right, diagNo, ans);
25     }
26
27     public static ArrayList<Integer> diagonalOrderSum(TreeNode root) {
28         ArrayList<Integer> ans = new ArrayList<Integer>();
29         dfs(root, 0, ans);
30         return ans;
31     }

```

