Trees → Generic Tree, Binary Tree, Binary Search Tree

Level (2) Lecture (1) {Saturday, 12 Feb}

Maximum Height

Minimum Height

Is Tree Balanced?

Diameter of Binary Tree

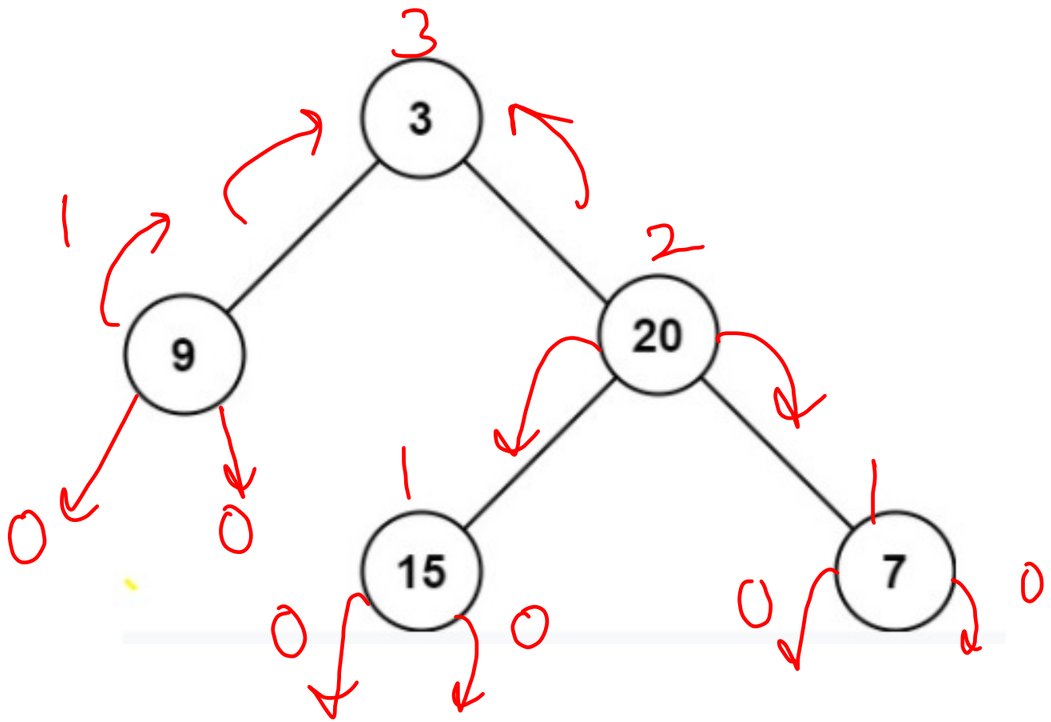
Diameter of NAry Tree

Maximum Depth of Binary Tree

LC 104

```java
class Solution {
    public int maxDepth(TreeNode root) {
        if(root == null) return 0;
        return 1 + Math.max(maxDepth(root.left), maxDepth(root.right));
    }
}
```

root.val



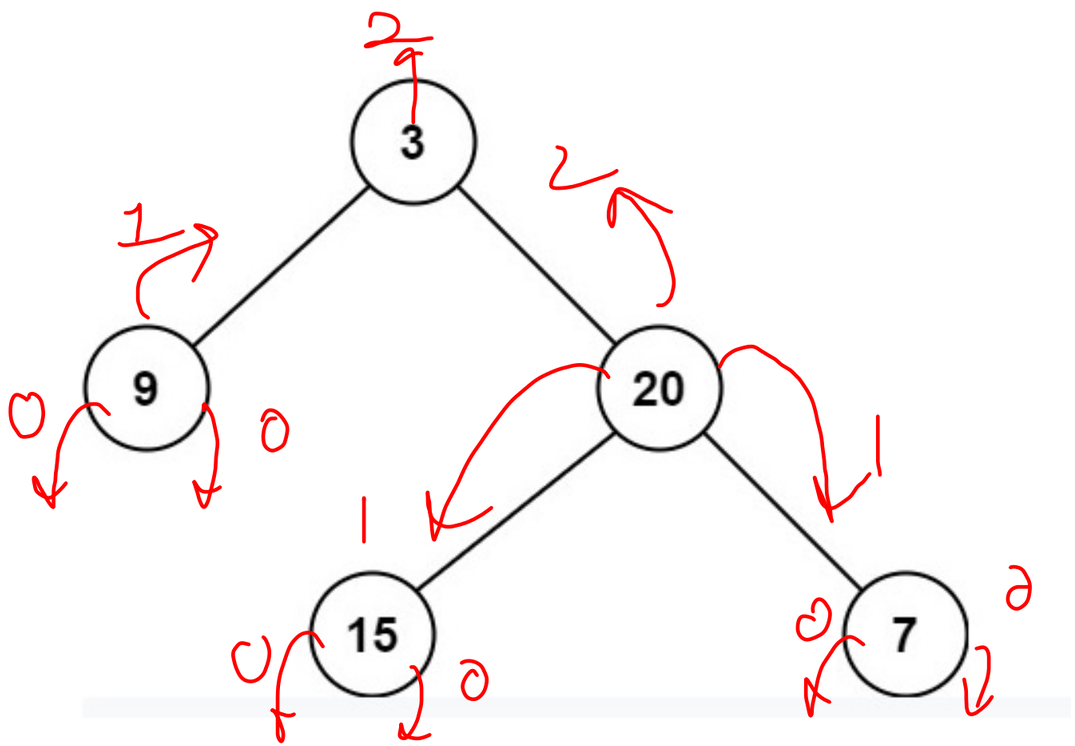
Follow up : Replace 1 with root.val

maximum path sum from root to any leaf node

Minimum Depth (111) LC

Minimum distance from root to any leaf node

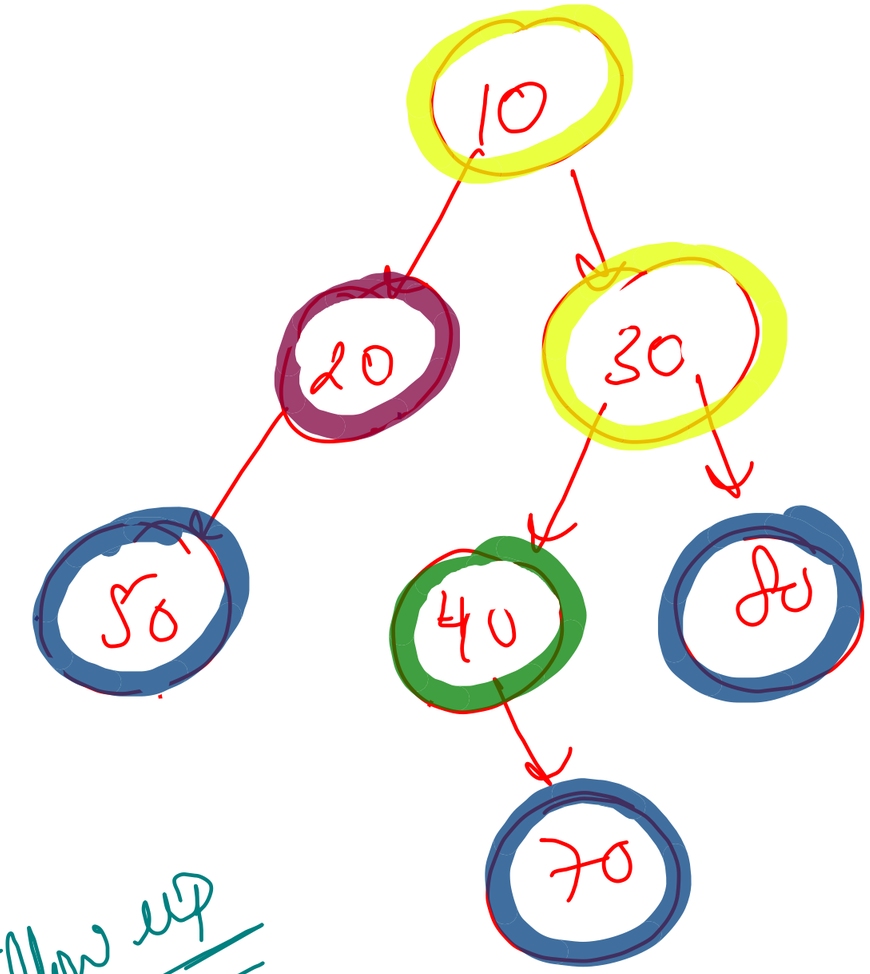
↳ children = 0




```java
class Solution {
    public int maxDepth(TreeNode root) {
        if(root == null) return 0;
        return 1 + Math.max(maxDepth(root.left), maxDepth(root.right));
    }
}
```

min

Code is wrong for single child



Follow up
e.g.

Minimum Path Sum from Root Node to any leaf node

Root == null → 0

Root.left == null && root.right == null

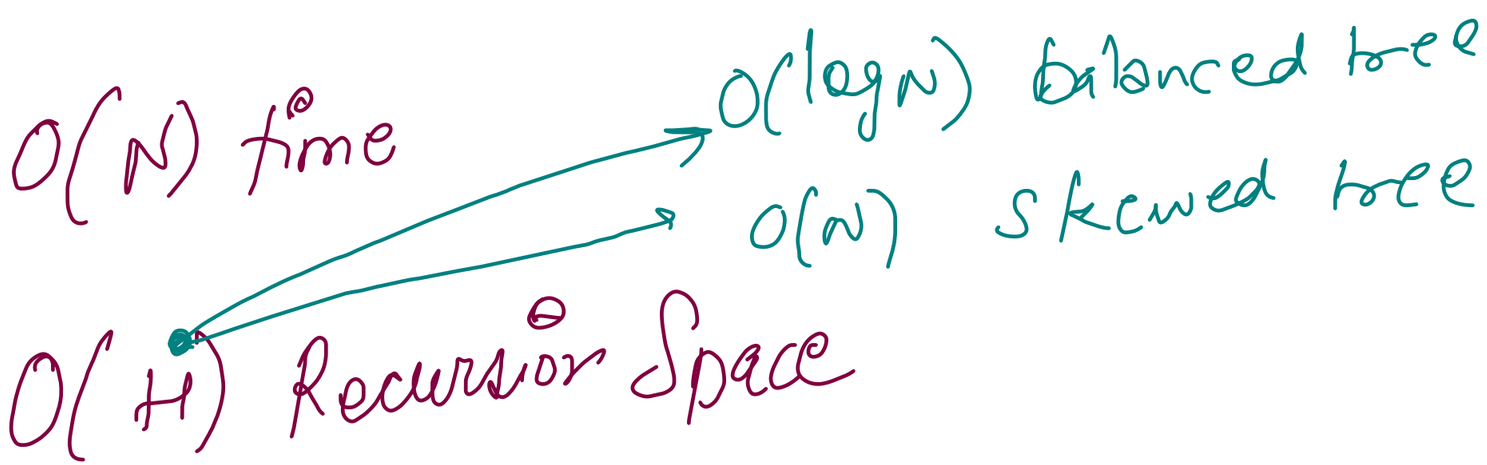
↳ leaf node → 1

Root.left == null → answer from right child +1

Root.right == null → answer from left child +1

2 child node → return (min(lm, rm) + 1)

```java
public int minDepth(TreeNode root) {
    if(root == null) return 0;

    if(root.left == null && root.right == null)
        return 1; // Leaf Node

    if(root.left == null) // Only Right Child
        return 1 + minDepth(root.right);

    if(root.right == null) // Only Left Child
        return 1 + minDepth(root.left);

    // Node with 2 Childs
    return 1 + Math.min(minDepth(root.left), minDepth(root.right));
}
```

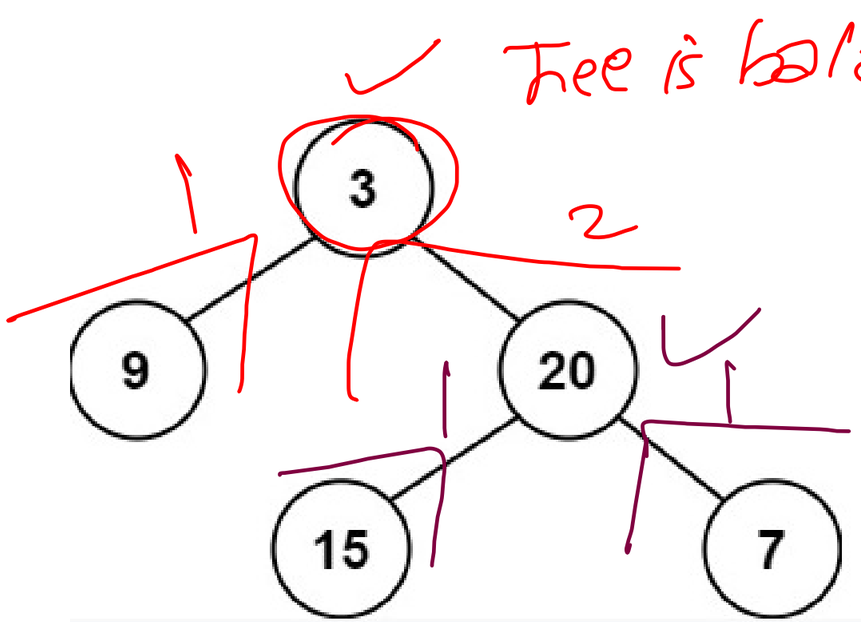


Is Tree Balanced?

110 LC

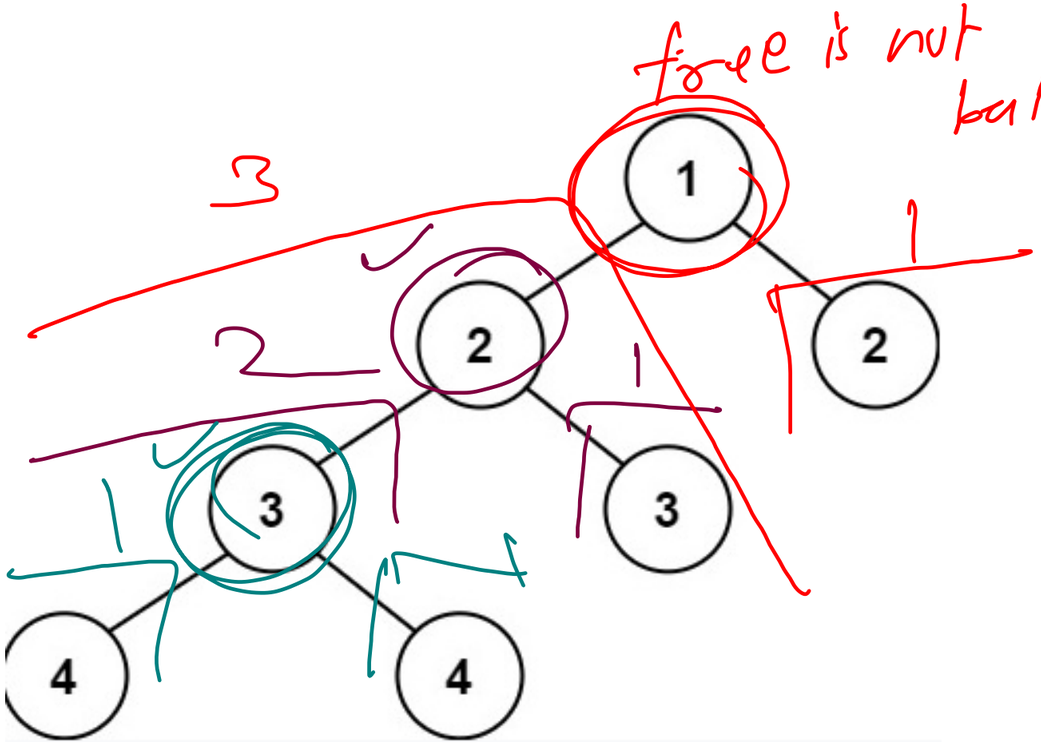
For each node { $|lh - rh| \leq 1$

$lh = rh + 1,\ lh = rh,\ lh = rh - 1$



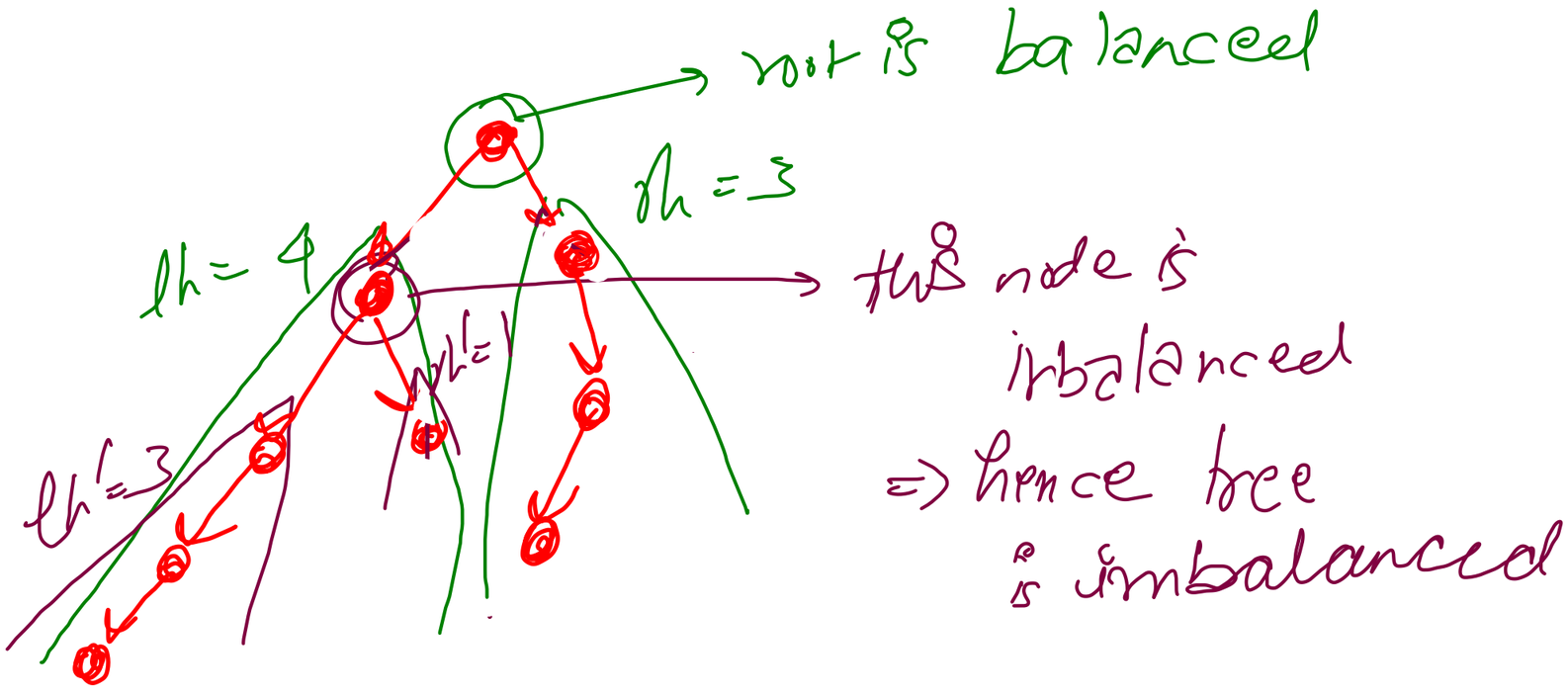
Tree is balanced

True



Tree is not balanced

False

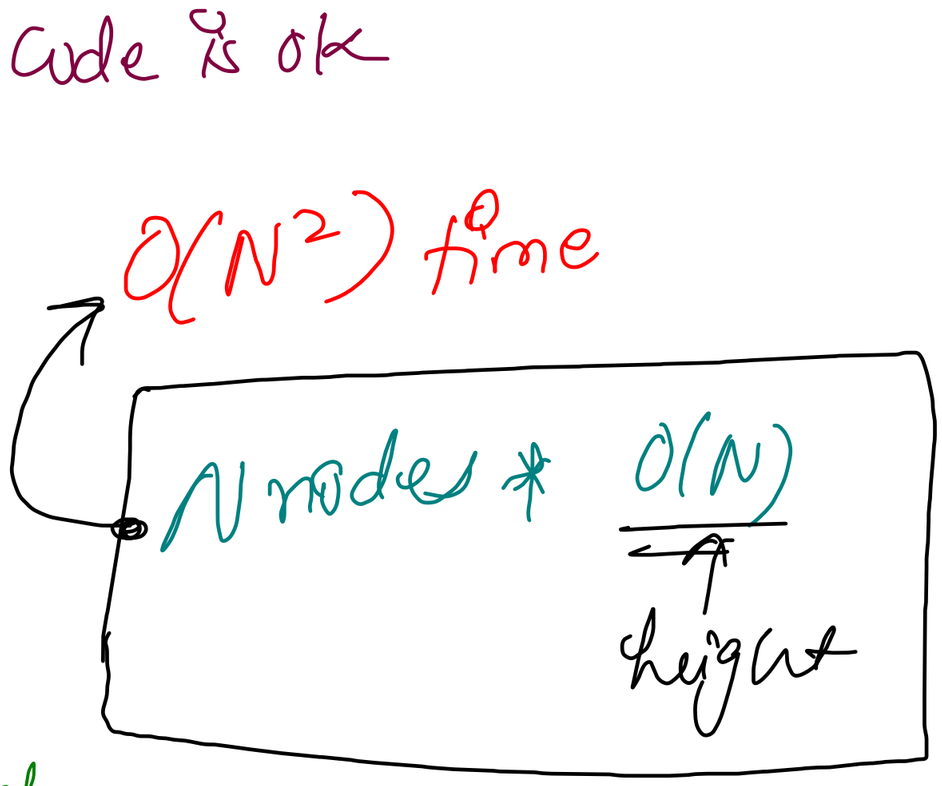


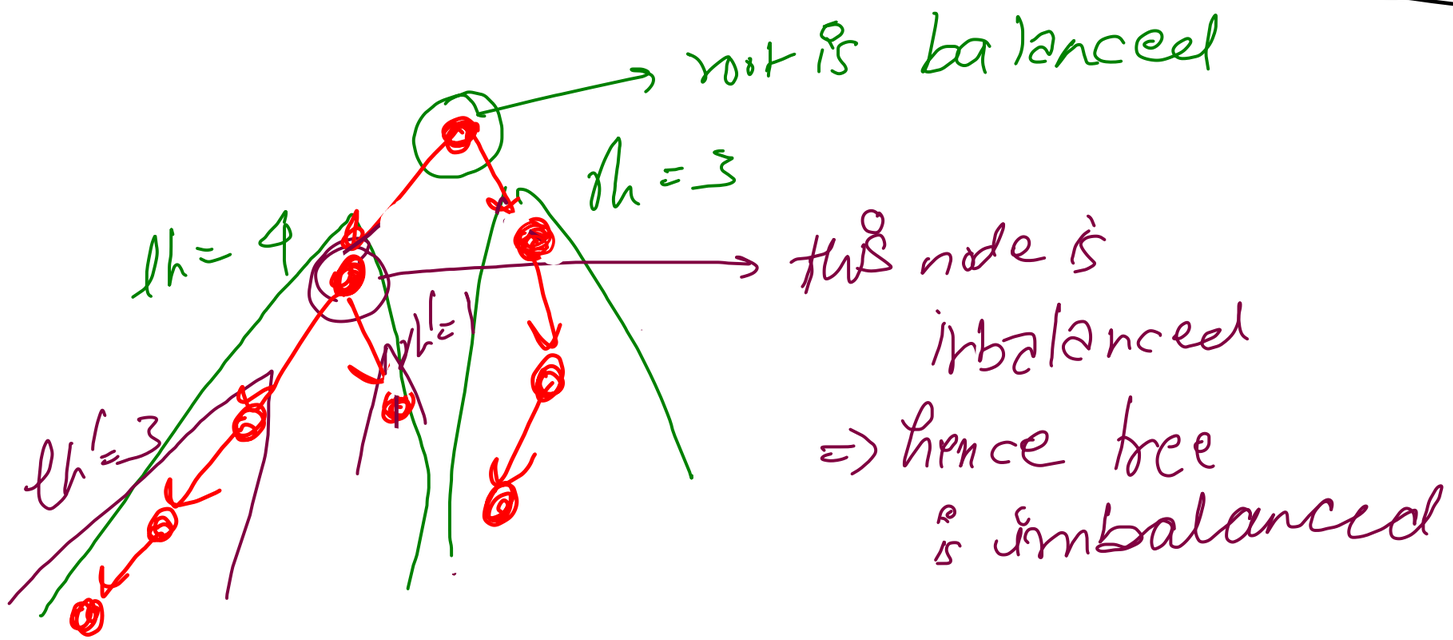

```java
public int height(TreeNode root){
    if(root == null) return 0;
    return 1 + math.max(height(root.left), height(root.right));
}

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

    int lheight = height(root.left);
    int rheight = height(root.right);

    if(lheight - rheight < -1 || lheight - rheight > 1) return false;
    return true;
}
```

code is wrong
↓
You are only checking for root node

```java
public int height(TreeNode root){
    if(root == null) return 0;
    return 1 + Math.max(height(root.left), height(root.right));
}

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

    int lheight = height(root.left);
    int rheight = height(root.right);

    if(lheight - rheight < -1 || lheight - rheight > 1) return false;
    return isBalanced(root.left) && isBalanced(root.right);
}
```

Code is ok





Rerooting Technique (DP on Tree)

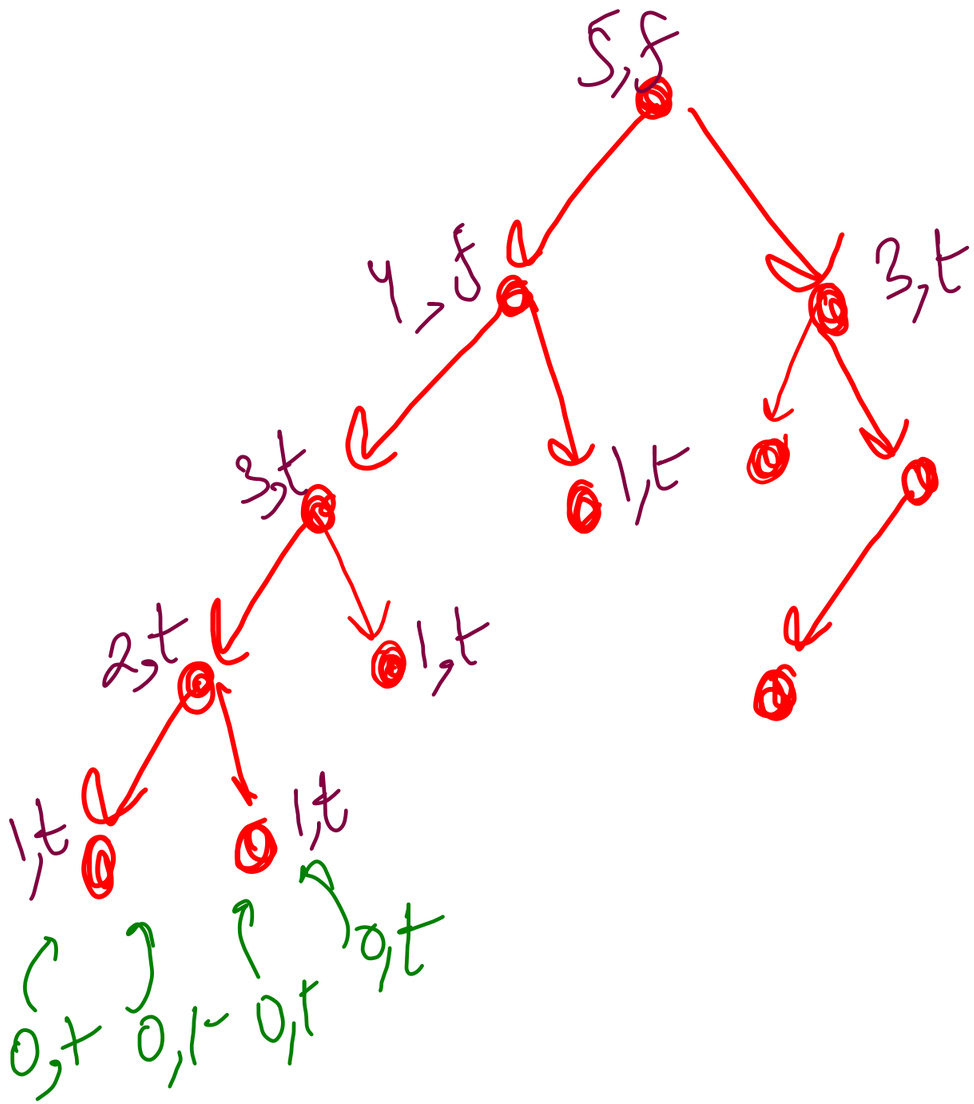
```java
static class Pair{
    boolean isBalanced = true;
    int height = 0;
}

public Pair helper(TreeNode root) {
    if(root == null) return new Pair();

    Pair left = helper(root.left);
    Pair right = helper(root.right);

    Pair curr = new Pair();
    curr.height = Math.max(left.height, right.height) + 1;
    curr.isBalanced = (left.isBalanced && right.isBalanced
        && (Math.abs(left.height - right.height) <= 1));
    return curr;
}

public boolean isBalanced(TreeNode root){
    return helper(root).isBalanced;
}
```

O(N) Time

O(H) Recursion Call Stack Space

O(N) Worst case (Skewed tree)

$O(\log_2 N)$ {balanced binary tree}



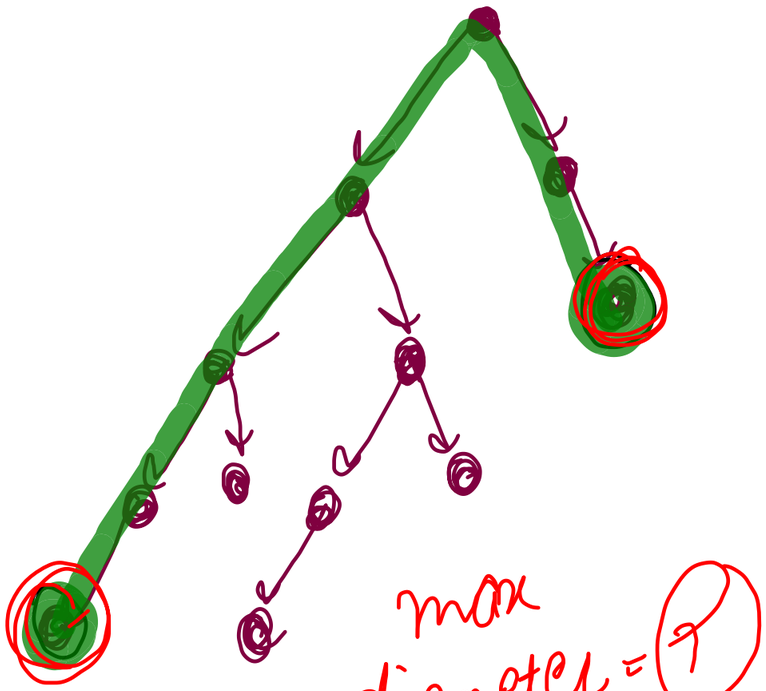

```java
static class Pair{
    boolean isBalanced = true;
    int height = 0;
}

public Pair helper(TreeNode root) {
    if(root == null) return new Pair();

    Pair left = helper(root.left);
    Pair right = helper(root.right);

    Pair curr = new Pair();
    curr.height = Math.max(left.height, right.height) + 1;
    curr.isBalanced = (left.isBalanced && right.isBalanced
        && (Math.abs(left.height - right.height) <= 1));
    return curr;
}

public boolean isBalanced(TreeNode root){
    return helper(root).isBalanced;
}
```

Diameter

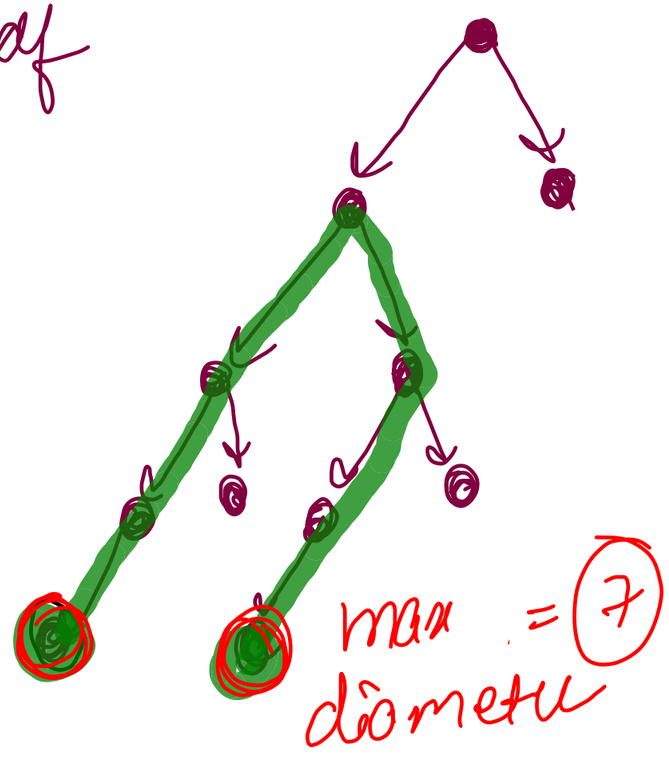
- Binary Tree
- Generic Tree

Longest distance b/w any node to any node

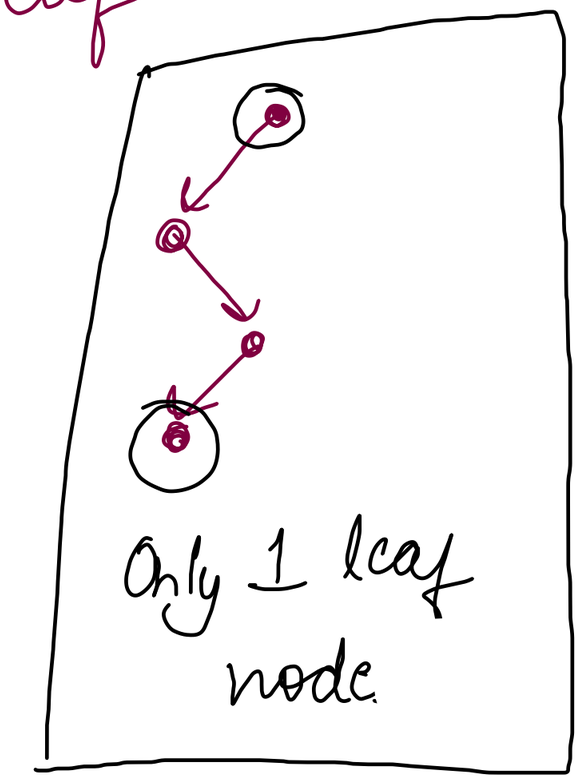
- any node → leaf
- any node → leaf



max diameter = 7



max diameter = 7



only 1 leaf node

```java
public int height(TreeNode root){
    if(root == null) return 0;
    return 1 + Math.max(height(root.left), height(root.right));
}

public int diameterOfBinaryTree(TreeNode root) {
    if(root == null) return 0;

    int lh = height(root.left);
    int rh = height(root.right);

    return 1 + lh + rh;
}
```

Diameter based on Root Node

This code is wrong because diameter may or may not pass through root

```java
public int height(TreeNode root){
    if(root == null) return 0;
    return 1 + Math.max(height(root.left), height(root.right));
}

public int diameter(TreeNode root) {
    if(root == null) return 0;

    int lh = height(root.left);
    int rh = height(root.right);

    int ld = diameter(root.left);
    int rd = diameter(root.right);
    return Math.max(lh + rh + 1, Math.max(ld, rd));
}

public int diameterOfBinaryTree(TreeNode root){
    if(root == null) return 0;
    return diameter(root) - 1;
    // Diameter in Terms of Edges = Diamater in Terms of Nodes - 1
}
```

$O(N^2)$ Time
$O(H)$ Space

$N * N = O(N^2)$

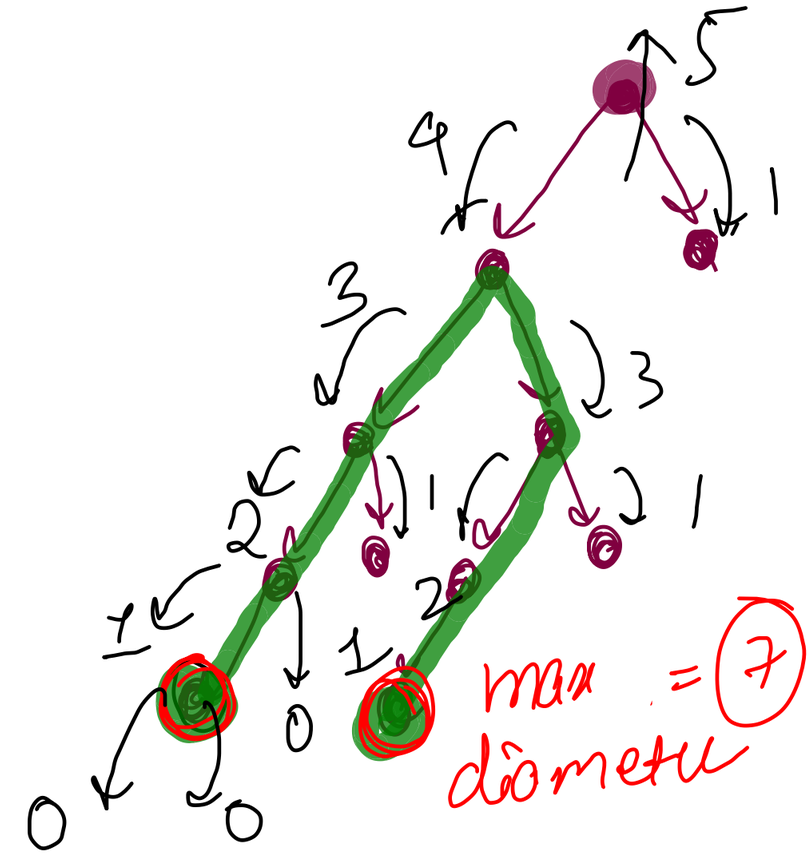
diameter Length.

This correct is but take more time
Correct

```java
public int globalDia = 0;
public int diameter(TreeNode root){
    if(root == null) return 0;

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

    // Global Variable Strategy or Travel & Change Strategy
    globalDia = Math.max(globalDia, lh + rh + 1);
    return Math.max(lh, rh) + 1;
}

public int diameterOfBinaryTree(TreeNode root){
    if(root == null) return 0;
    diameter(root);
    return globalDia - 1;
    // Diameter in Terms of Edges = Diamater in Terms of Nodes - 1
}
```

Postorder

globalDia = 0 1 7 4 7



Travel & Change

O(N) time

O(H) recursion stack

"heavy" → Travel & change

```java
public int diameter(TreeNode root, int[] globalDia){
    if(root == null) return 0;

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

    // Global Variable Strategy or Travel & Change Strategy
    globalDia[0] = Math.max(globalDia[0], lh + rh + 1);
    return Math.max(lh, rh) + 1;
}

public int diameterOfBinaryTree(TreeNode root){
    if(root == null) return 0;
    int[] globalDia = new int[1];

    diameter(root, globalDia);
    return globalDia[0] - 1;
    // Diameter in Terms of Edges = Diamater in Terms of Nodes - 1
}
```

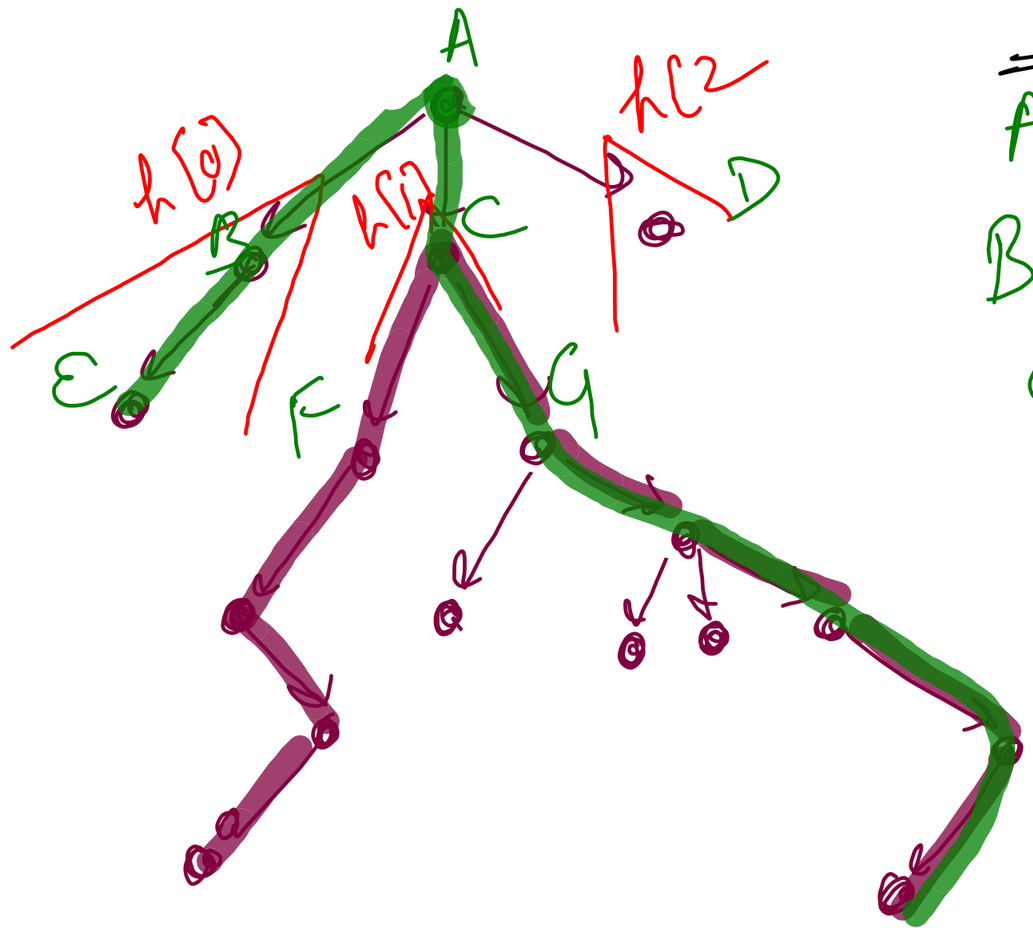
Same approach but without global variable

```java
public static class Pair{
    int height;
    int diameter; // Diameter is of entire subtree
}

public Pair diameter(TreeNode root){
    if(root == null) return new Pair();

    Pair left = diameter(root.left);
    Pair right = diameter(root.right);

    Pair curr = new Pair();
    curr.height = Math.max(left.height, right.height) + 1;
    curr.diameter = left.height + right.height + 1;
    curr.diameter = Math.max(curr.diameter, Math.max(left.diameter, right.diameter));
    return curr;
}

public int diameterOfBinaryTree(TreeNode root){
    if(root == null) return 0;
    // Diameter in Terms of Edges = Diamater in Terms of Nodes - 1
    return diameter(root).diameter - 1;
}
```

O(N) time
O(H) Recursion Call Stack

Diameter of Generic/N-Ary Tree

{ Largest Distance Bw 2 nodes in Tree }



Adjacency List

A → B, C, D

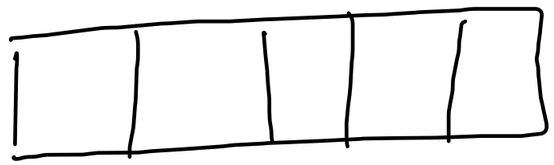
B → E

C → F, G

D → { }

!

!



par [i] → i

lh + rh + 1 (Children's height)

rh → second max

lh → max(height child)

```java
public int globalDia = 0;

public int dfs(ArrayList<Integer>[] adj, int root){
    int maxHeight = 0, secondMaxHeight = 0;
    for(Integer child: adj[root]){
        int height = dfs(adj, child);
        if(height > maxHeight){
            secondMaxHeight = maxHeight;
            maxHeight = height;
        }
        else if(height >= secondMaxHeight)
            secondMaxHeight = height;
    }

    globalDia = Math.max(globalDia, maxHeight + secondMaxHeight + 1);
    return 1 + maxHeight;
}
```

parent's array

```java
public int solve(int[] A) {
    ArrayList<Integer>[] adj = new ArrayList[A.length];
    for(int i=0; i<A.length; i++){
        adj[i] = new ArrayList<>();
    }

    int root = 0;
    for(int i=0; i<A.length; i++){
        if(A[i] == -1) root = i;
        else adj[A[i]].add(i);
    }

    dfs(adj, root);
    return globalDia - 1;
}
```

A[i] = j

if j == -1 ⇒ i is the root node

else j is the parent of i, i is child of j

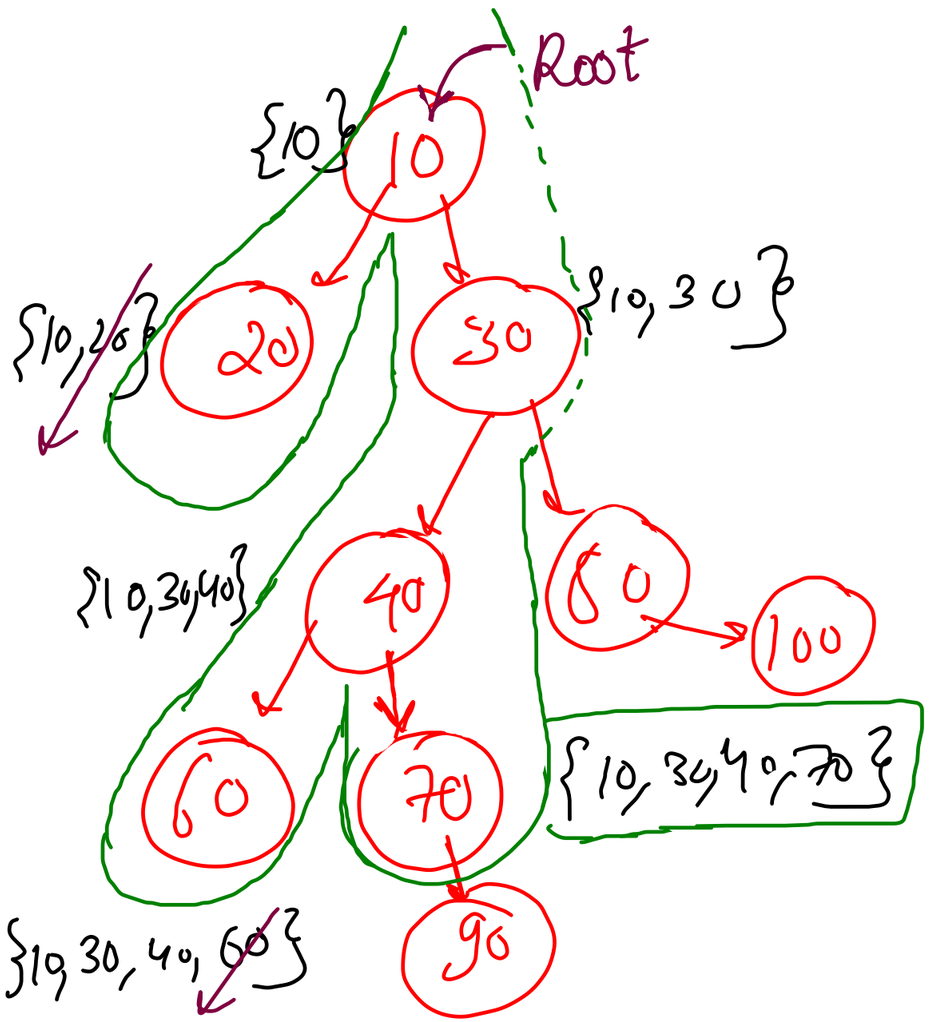
Level (2), Lecture (2) 13 Feb (3PM)

System Design → Static, final, etc.

(1) Node to Root Path
Root to leaf Path
linked List in Tree

(2) Max Sum Subtree
Tilt of Binary Tree

(3) Path Sum
→ Root to Leaf (1&11)
→ Node to Node (downwards)
→ Leaf to Leaf
→ Node to Node (Any)



target ⇒ 70

boolean Find (target, ArrInt> path)

Ancestors Node to Root Path

↳ 70, 40, 30, 10

Root to Node Path

↳ 10, 30, 40, 70

```java
public static boolean Ancestors(Node root, ArrayList<Integer> path, int target){
    if(root == null) return false;
    if(target == root.data) return true;

    path.add(root.data);
    if(Ancestors(root.left, path, target) == true)
        return true;

    if(Ancestors(root.right, path, target) == true)
        return true;

    path.remove(path.size() - 1); // backtrack
    return false;
}
```

Root to Node

O(N) Time

O(H) recursion

```java
public static ArrayList<Integer> Ancestors(Node root, int target)
{
    ArrayList<Integer> ancestors = new ArrayList<>();
    Ancestors(root, ancestors, target);
    Collections.reverse(ancestors);
    return ancestors;
}
```

Node to Root Path

257 Root to leaf Node

```java
public void helper(TreeNode root, String path, List<String> paths){
    if(root == null){
        // if any only if tree is of 0 nodes
        return;
    }

    if(root.left == null && root.right == null){
        // leaf node
        paths.add(path + root.val);
        return;
    }

    helper(root.left, path + root.val + "->", paths);
    helper(root.right, path + root.val + "->", paths);
}

public List<String> binaryTreePaths(TreeNode root) {
    List<String> paths = new ArrayList<>();
    helper(root, "", paths);
    return paths;
}
```

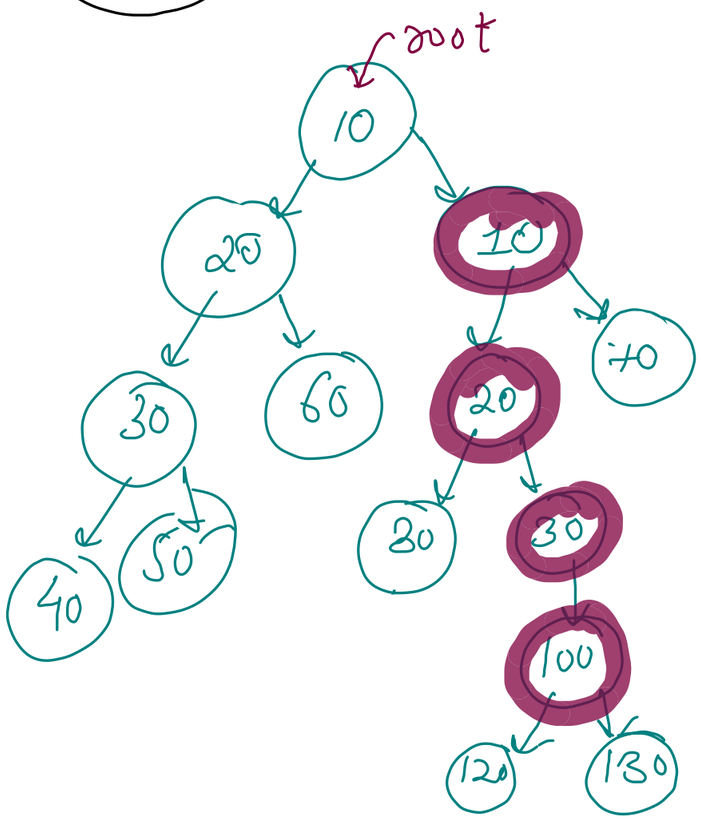
Variations
- # Nodes ⇒ Path to leaf in Range [L, R]
- # All paths dum print

O(N) Time

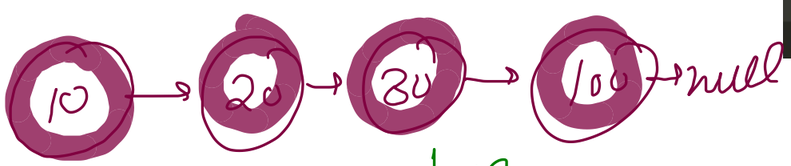
O(H) Recursion call Stack

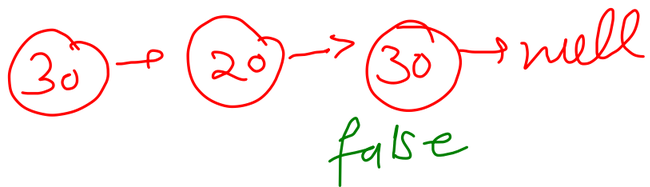
1367 Linked List in Binary Tree

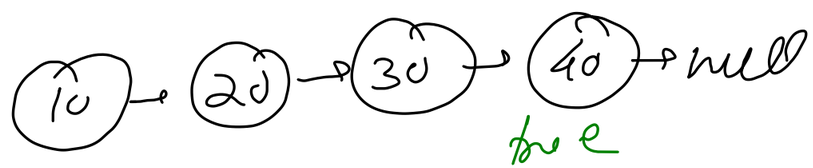
$O(N^2)$ time, $O(H)$ Recursion Space




```java
public boolean isSubPathHelper(ListNode head, TreeNode root){
    if(head == null) return true;
    // Linked List has all nodes present, return true

    if(root == null) return false;
    // Linked List is still remaining, but tree is empty

    if(root.val == head.val){
        if(isSubPathHelper(head.next, root.left) == true)
            return true;

        if(isSubPathHelper(head.next, root.right) == true)
            return true;
    }

    return false;
}

public boolean isSubPath(ListNode head, TreeNode root) {
    if(head == null) return true;
    if(root == null) return false;

    return isSubPathHelper(head, root) || isSubPath(head, root.left)
                                       || isSubPath(head, root.right);
}
```

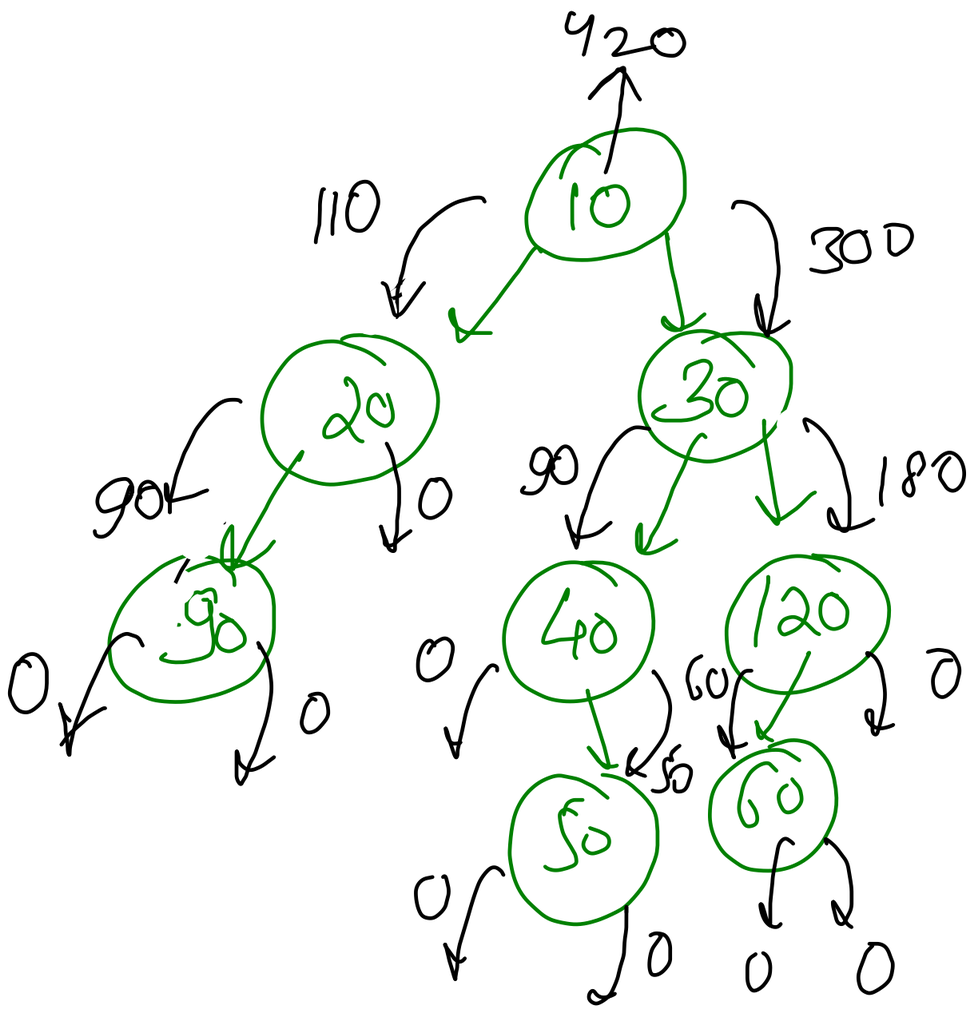






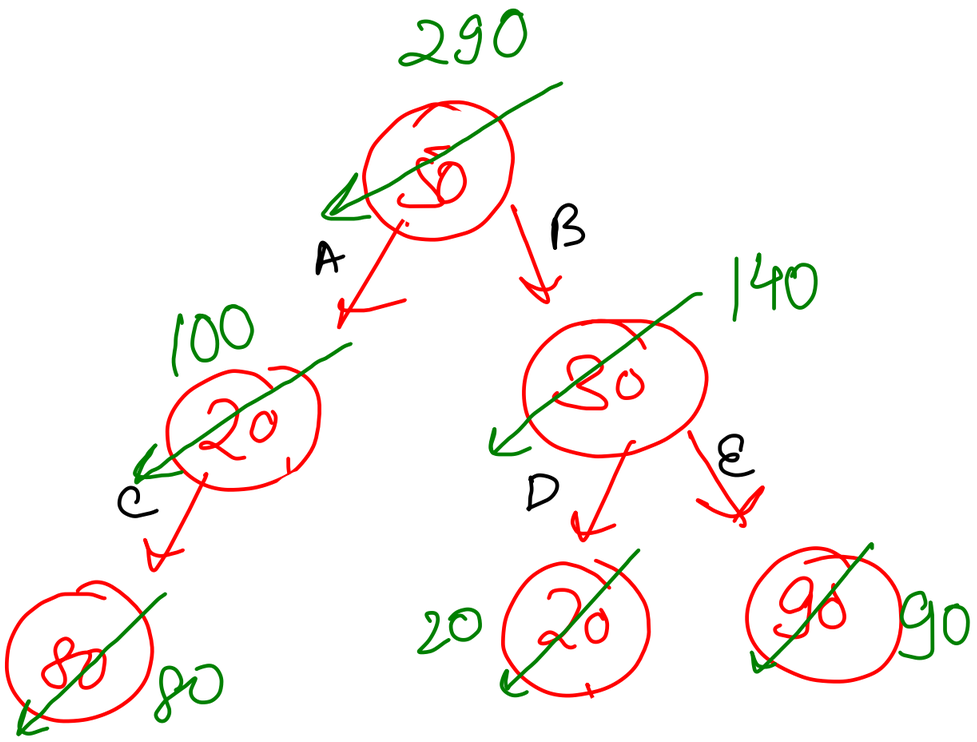
TCLF

SG 3



$$\text{Tilt} = 0+0+90+50+60+90+190$$

1339 Maximum Product Subtree



#1

$S1 = 290 - 100$

$S1 = 50 \to 30 \to 20, 90$

$S2 = 100$

$S2 = 20 \to 80$

B

$= 290 - 140$

$S1 = 50 \to 20 \to 80$

$S2 = 140$

$S2 = 30 \to 20, 90$

C

80 → 50 → 20, 30; 20, 90

S1 * S2

D

$S1 * S2$ → 20

50 → 20, 30; 20, 90; 80

E

50 → 90

50 → 20, 30; 20, 30; 80, 20

$S1 * S2$

$S = 20$

```java
public int sumTree(TreeNode root){
    if(root == null) return 0;
    root.val += sumTree(root.left) + sumTree(root.right);
    return root.val;
}

long maxProduct = 0;
long total = 0;
public void helper(TreeNode root){
    if(root == null) return;

    helper(root.left);
    helper(root.right);

    long leftSubtree = (root.left == null) ? 0l : root.left.val;
    long leftProduct = leftSubtree * (total - leftSubtree);

    long rightSubtree = (root.right == null) ? 0l : root.right.val;
    long rightProduct = rightSubtree * (total - rightSubtree);

    maxProduct = Math.max(maxProduct, Math.max(leftProduct, rightProduct));
}

public int maxProduct(TreeNode root) {
    if(root == null) return 0;
    total = sumTree(root);
    helper(root);
    return (int)(maxProduct % 10000000071);
}
```

converting tree into subtree

left edge removal

right edge removal

O(N) time

O(H) Recursion call stacks

```java
public boolean hasPathSum(TreeNode root, int targetSum) {
    if(root == null) return false;
    if(root.left == null && root.right == null){
        // Root to Leaf Path Sum = targetSum
        return (targetSum == root.val);
    }

    if(hasPathSum(root.left, targetSum - root.val) == true) return true;
    if(hasPathSum(root.right, targetSum - root.val) == true) return true;
    return false;
}
```

Path Sum - (1) [112] LC

Root to leaf Target

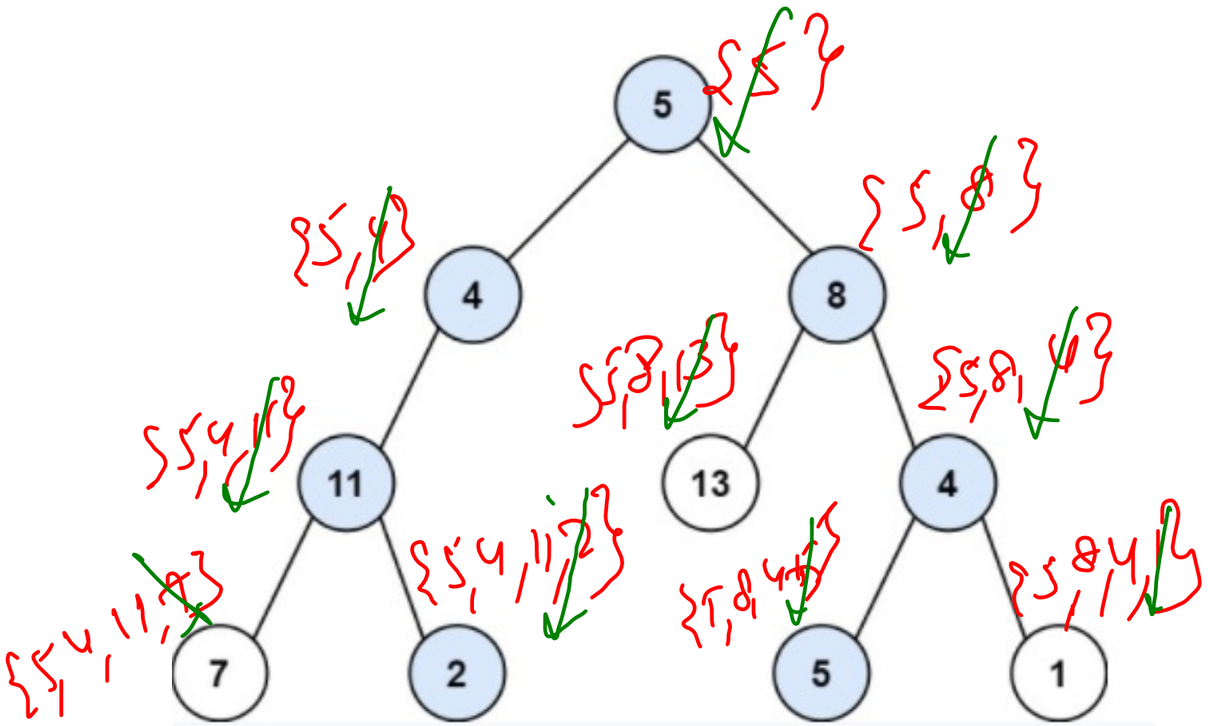
True False

```java
List<List<Integer>> paths;
public void helper(TreeNode root, List<Integer> path, int target){
    if(root == null) return;

    path.add(root.val);
    if(root.left == null && root.right == null){
        if(target == root.val)
            paths.add(new ArrayList<>(path)); // deep copy

        path.remove(path.size() - 1);
        return;
    }

    helper(root.left, path, target - root.val);
    helper(root.right, path, target - root.val);
    path.remove(path.size() - 1); // backtrack
}

public List<List<Integer>> pathSum(TreeNode root, int targetSum) {
    paths = new ArrayList<>();
    helper(root, new ArrayList<>(), targetSum);
    return paths;
}
```

Path Sum (11) [113] LC

Root-to-Leaf All paths

with given Target



target = 22

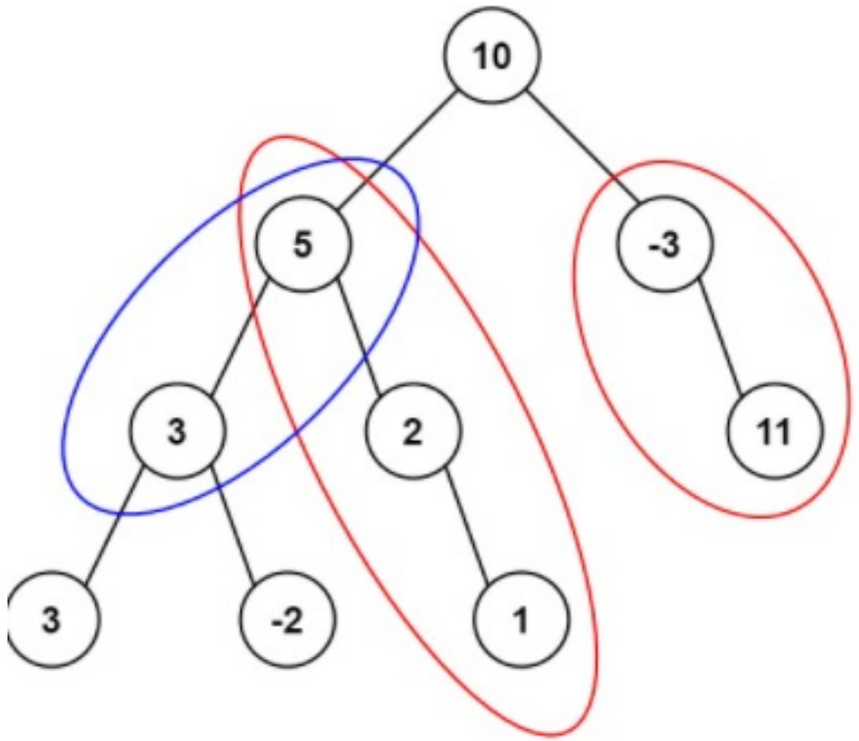
Root to leaf path
with given target-

{ {5, 4, 11, 2} {5, 8, 4, 5} }

base case {deep copy}

2d.add(new ArrayList(ld));

(437) Path Sum - III { Any node to Any node Downwards }

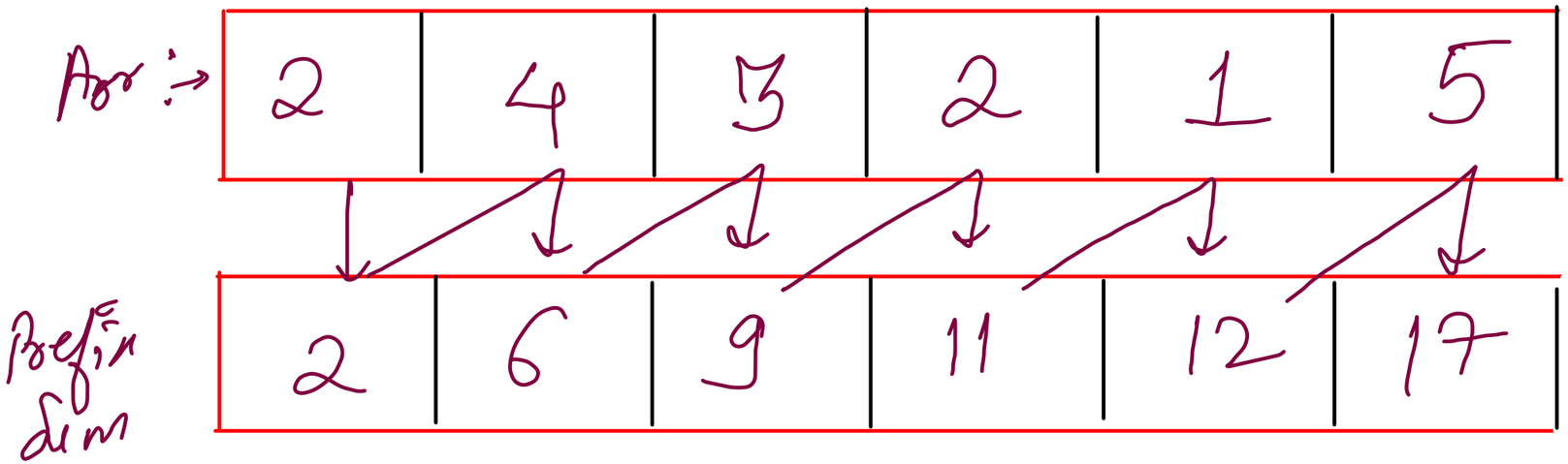


count of targetSum Path

target = 8

{ N * DFS ⇒ N * N ⇒ $O(N^2)$ }

```java
public int dfs(TreeNode root, int targetSum){
    if(root == null) return 0;

    targetSum -= root.val;

    int count = 0;
    if(targetSum == 0)
        count++;

    return count + dfs(root.left, targetSum) + dfs(root.right, targetSum);
}

public int pathSum(TreeNode root, int targetSum) {
    if(root == null) return 0;
    return dfs(root, targetSum) + pathSum(root.left, targetSum) +
        pathSum(root.right, targetSum);
}
```



SDE LC

K sum subarray

Count subarrays with given sum

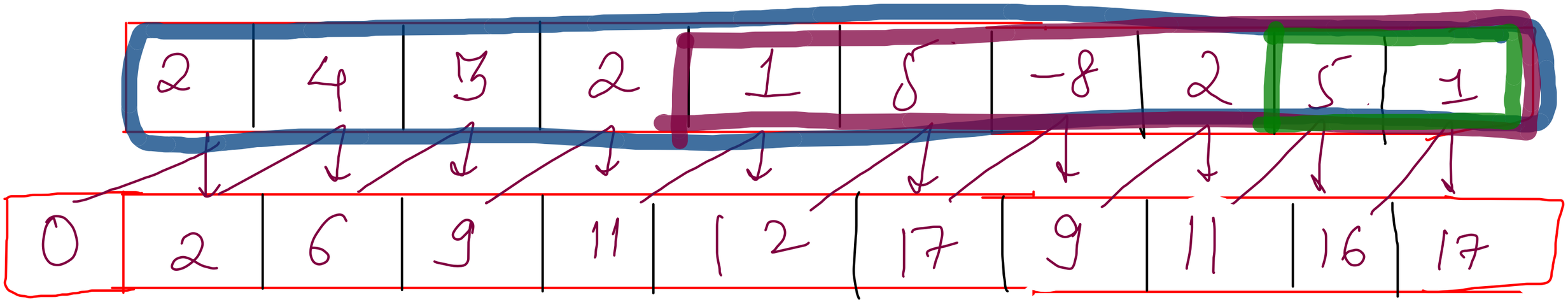
target ≠ 0

{2,4} {3,2,1} {1,5}

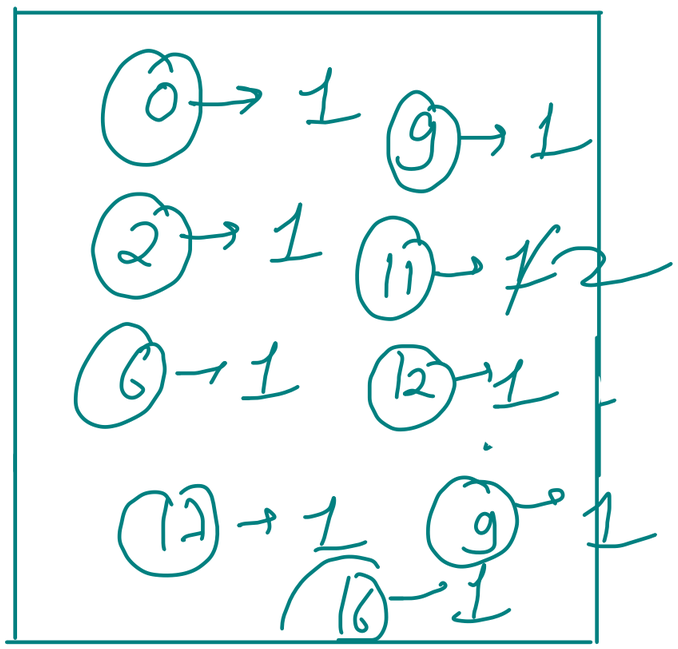
Brute force :→ $O(N^2)$

count of subarrays ending at i with sum target

= freq[prefixSum[i] − target]



target = 6 count = 1+1+1+1+2



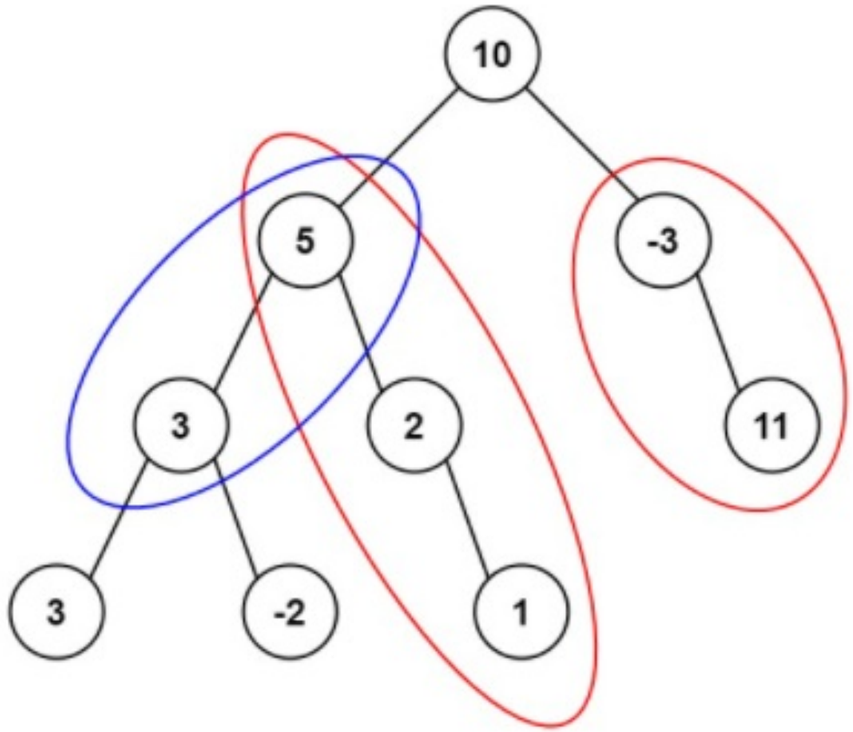
count of subarrays ending at i with sum target

= freq[prefixSum[i] - target]

```java
public int subarraySum(int[] nums, int k) {
    HashMap<Integer, Integer> freq = new HashMap<>();
    int prefSum = 0;
    freq.put(0, 1);

    int res = 0;
    for(int i=0; i<nums.length; i++){
        prefSum += nums[i];
        res += freq.getOrDefault(prefSum - k, 0);
        freq.put(prefSum, freq.getOrDefault(prefSum, 0) + 1);
    }

    return res;
}
```

O(N) time

O(N) hashmap extra space



target ⇒ 8

R2N	sum	Freq
0	10	1
10	15	1
15	18	1
18	21	1
21	16	1
16	17	1
17	18	1
18	7	1
7	18	1

count = 0 1 2 3

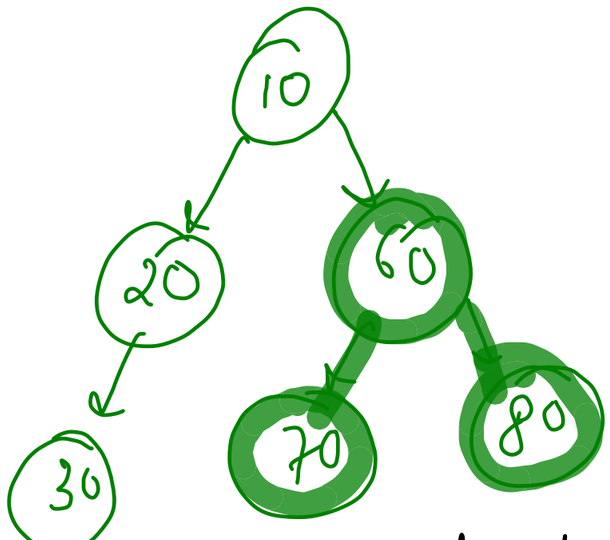
```java
// O(N) Approach
class Solution{
    HashMap<Integer, Integer> freq = new HashMap<>();

    public int helper(TreeNode root, int targetSum, int prefSum){
        if(root == null) return 0;

        prefSum += root.val;
        int count = freq.getOrDefault(prefSum - targetSum, 0);
        freq.put(prefSum, freq.getOrDefault(prefSum, 0) + 1);

        count += helper(root.left, targetSum, prefSum);
        count += helper(root.right, targetSum, prefSum);

        freq.put(prefSum, freq.getOrDefault(prefSum, 0) - 1); // Backtrack

        return count;
    }

    public int pathSum(TreeNode root, int targetSum){
        if(root == null) return 0;
        freq.put(0, 1);
        return helper(root, targetSum, 0);
    }
}
```

O(N) Time

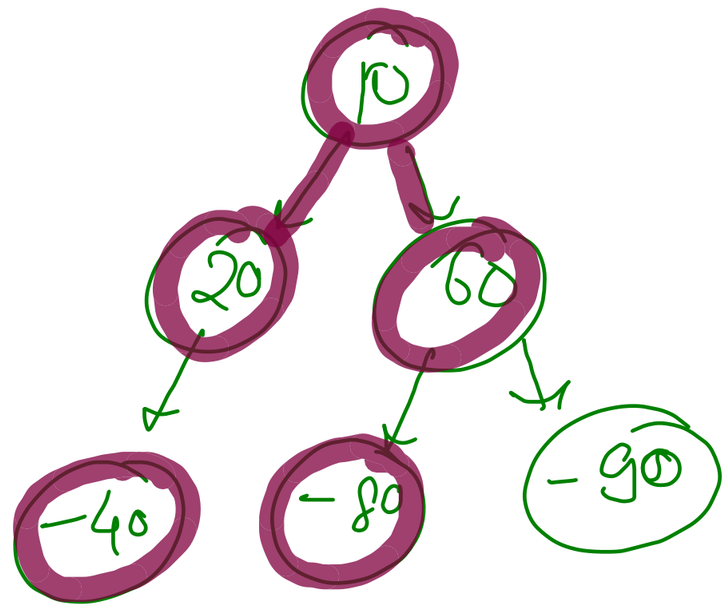
O(H) extra space (Hashmap)

O(H) Recursion call stack

Max Path Sum
b/w 2 leaf nodes



All +ve values



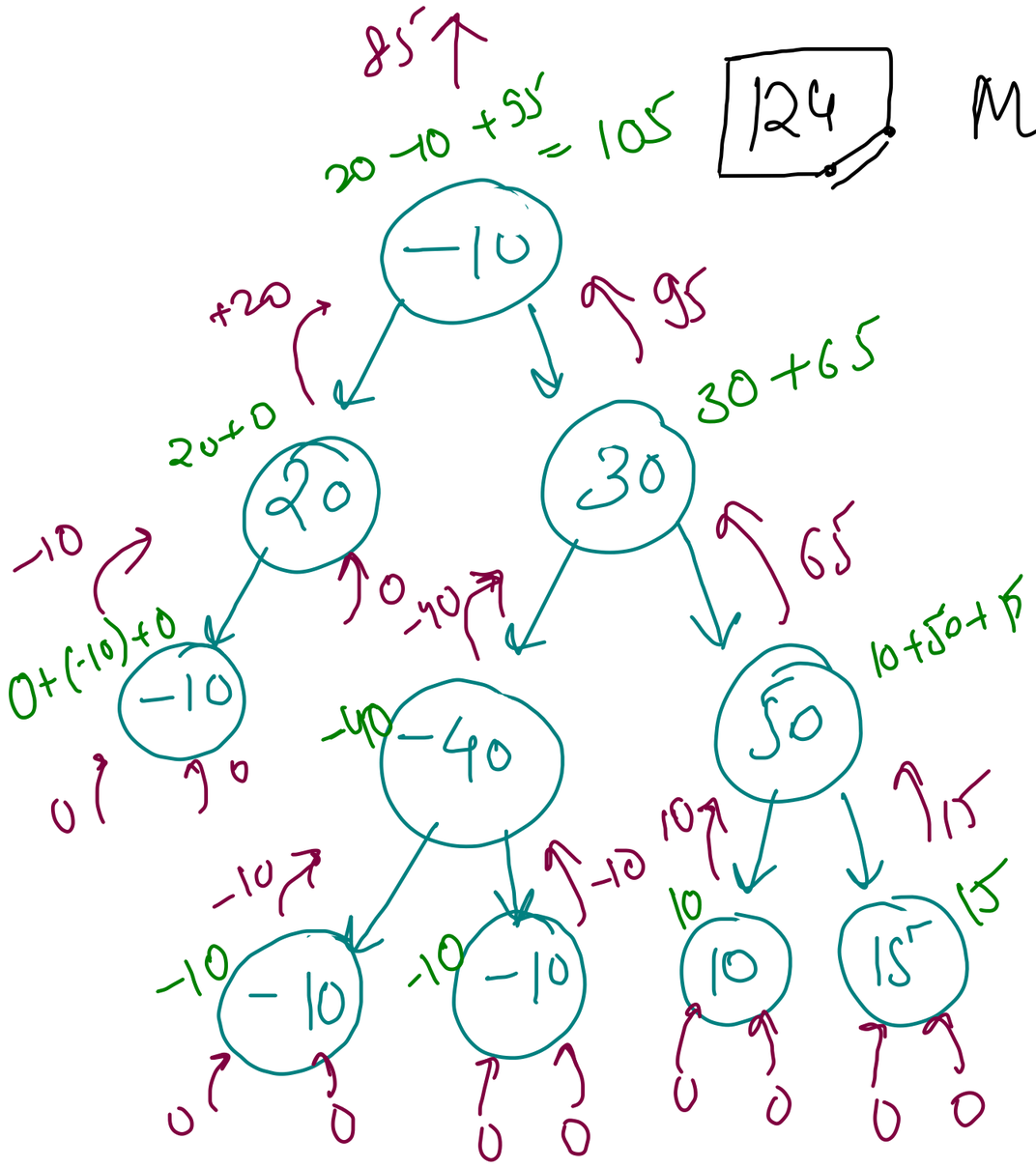
Max Path B/w Any 2 Nodes with all values as positive

```java
static long globalAns = Long.MIN_VALUE;
public long diameter(Node root){
    if(root == null) return 0l;
    if(root.left == null && root.right == null) return 1l * root.data;
    if(root.left == null) return 1l * root.data + diameter(root.right);
    if(root.right == null) return 1l * root.data + diameter(root.left);

    long left = diameter(root.left);
    long right = diameter(root.right);
    long diameter = left + right + 1l * root.data;
    if(diameter > globalAns) globalAns = diameter;
    return root.data * 1l + Math.max(left, right);
}

int maxPathSum(Node root)
{
    globalAns = Long.MIN_VALUE;
    long sum = diameter(root);
    if(root.left == null || root.right == null){
        globalAns = Math.max(globalAns, sum);
    }
    return (int)globalAns;
}
```

O(N) time

O(H) space

{ Diameter } variation



124

Max Path Sum (HARD)

{Any Node to Any Node}

{If All values are positive

⇒ variation of diameter

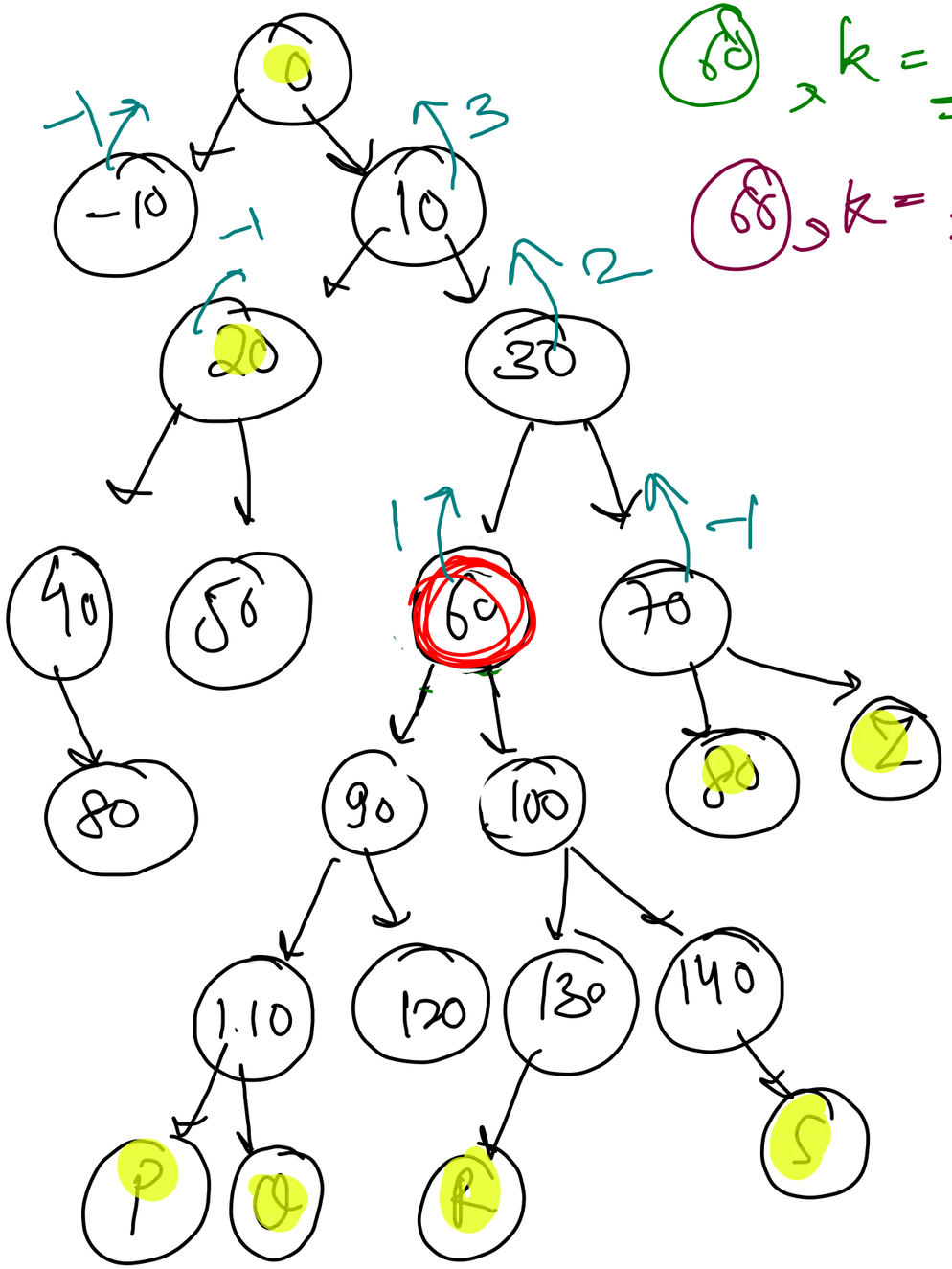
⇒ max Path Sum

Leaf to Leaf}

```java
class Solution {
    int maxPathSum = Integer.MIN_VALUE;

    public int helper(TreeNode root){
        if(root == null) return 0;

        int leftDownPath = helper(root.left);
        int rightDownPath = helper(root.right);

        int currPathSum = root.val + Math.max(0, leftDownPath) + Math.max(0, rightDownPath);
        maxPathSum = Math.max(maxPathSum, currPathSum);

        return Math.max(0, Math.max(leftDownPath, rightDownPath)) + root.val;
    }

    public int maxPathSum(TreeNode root) {
        helper(root);
        return maxPathSum;
    }
```

{ O(N) time
O(H) Recursion call stack }

using Travel & change { Global variable Strategy }



(60), k = 2

(68), k = 3

K Levels Down (GFG)

(2) 110, 120, 130, 140

(3) P, Q, R, S

K Nodes Away (LC-863)

2 → 110, 120, 130, 140

2 → 70

2 → 1

3 → P, Q, R, S

3 → 80, Z

3 → 20

3 → 0

```java
public boolean path(TreeNode root, TreeNode target, ArrayList<TreeNode> r2nPath){
    if(root == null) return false;
    if(root == target){
        r2nPath.add(root);
        return true;
    }

    r2nPath.add(root);
    if(path(root.left, target, r2nPath) == true) return true;
    if(path(root.right, target, r2nPath) == true) return true;
    r2nPath.remove(r2nPath.size() - 1);
    return false;
}
```

R2N Path O(H) Extra Space

```java
public void kLevelDown(TreeNode root, TreeNode blocker, int k, List<Integer> res){
    if(k < 0 || root == null || root == blocker)  return;
    if(k == 0){
        res.add(root.val);
        return;
    }

    kLevelDown(root.left, blocker, k - 1, res);
    kLevelDown(root.right, blocker, k - 1, res);
}
```

DFS from each node on R2N

```java
public List<Integer> distanceK(TreeNode root, TreeNode target, int k) {
    ArrayList<TreeNode> r2nPath = new ArrayList<>();
    if(path(root, target, r2nPath) == false) return new ArrayList<>();

    r2nPath.add(null);
    List<Integer> res = new ArrayList<>();
    for(int i=r2nPath.size()-2; i>=0; i--){
        TreeNode curr = r2nPath.get(i);
        TreeNode blocker = r2nPath.get(i + 1);
        kLevelDown(curr, blocker, k, res);
        k--;
    }
    return res;
}
```

O(n) time

10:11

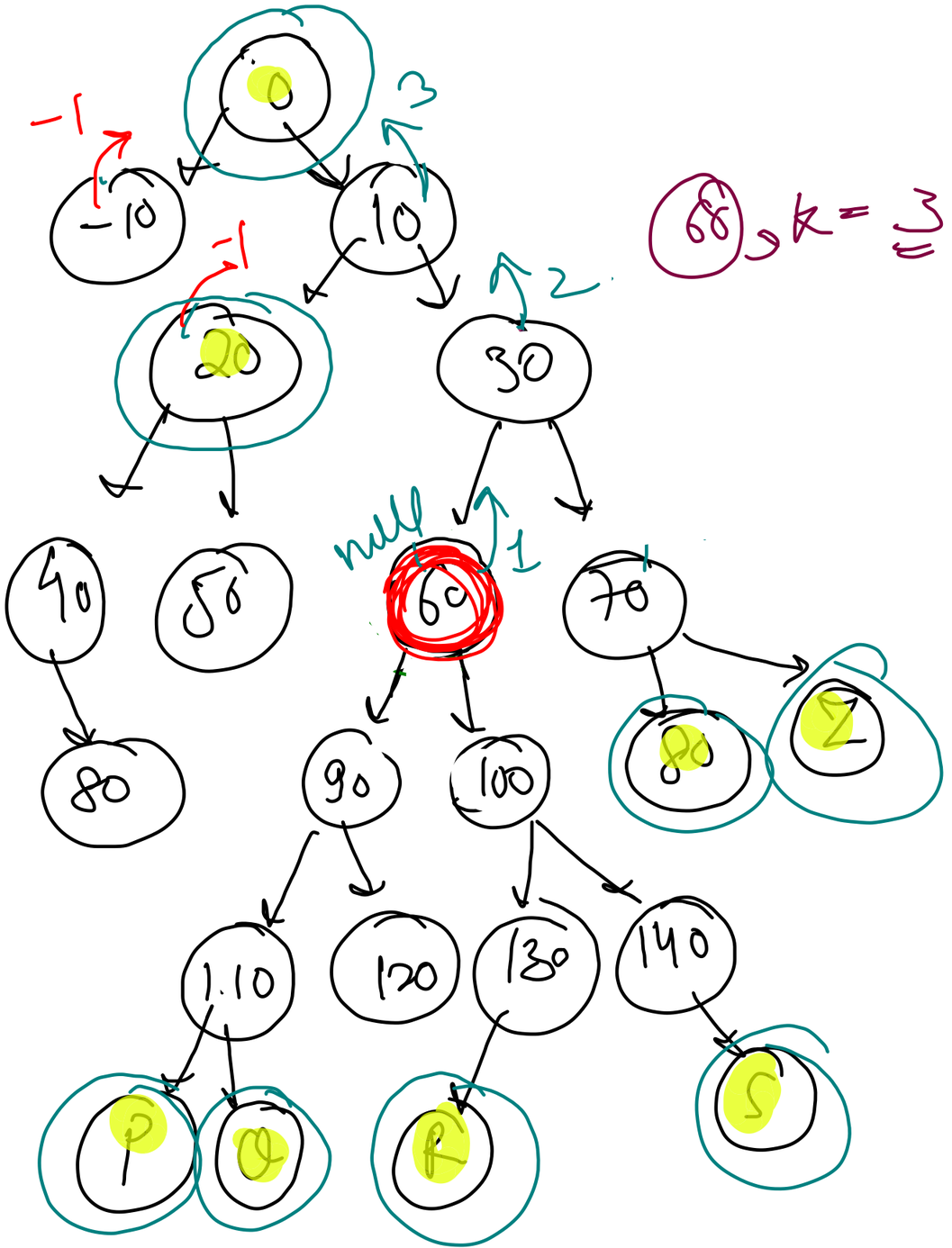
10:20

```java
public int DFS(TreeNode root, TreeNode target, int k, List<Integer> res){
    if(root == null) return -1;
    if(root == target){
        kLevelDown(root, null, k, res);
        return 1;
    }

    int left = DFS(root.left, target, k, res);
    if(left >= 0){
        kLevelDown(root, root.left, k - left, res);
        return left + 1;
    }

    int right = DFS(root.right, target, k, res);
    if(right >= 0){
        kLevelDown(root, root.right, k - right, res);
        return right + 1;
    }

    return -1;
}
```

O(n) time

{ without extra space

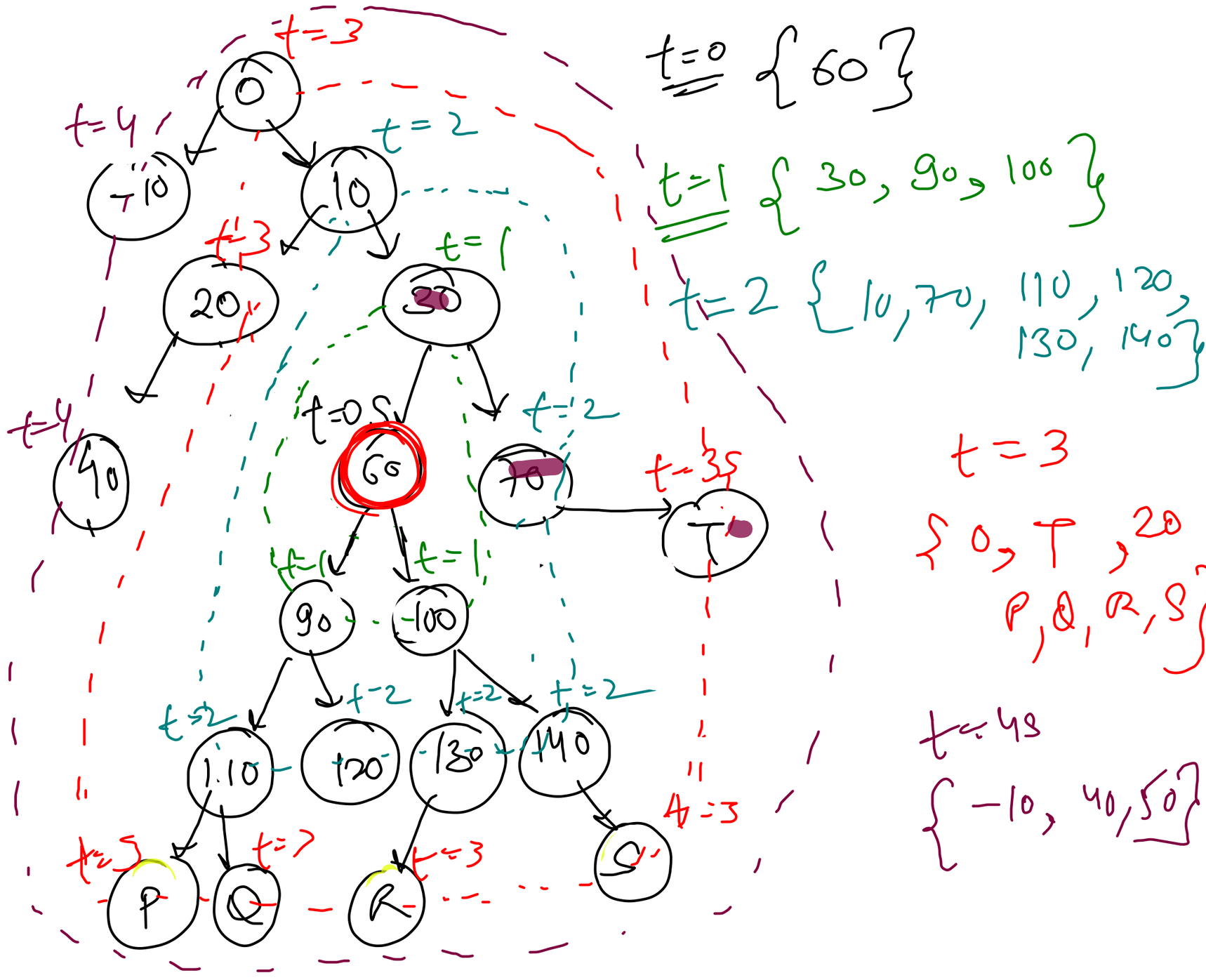
```java
public void kLevelDown(TreeNode root, TreeNode blocker, int k, List<Integer> res){
    if(k < 0 || root == null || root == blocker)  return;
    if(k == 0){
        res.add(root.val);
        return;
    }

    kLevelDown(root.left, blocker, k - 1, res);
    kLevelDown(root.right, blocker, k - 1, res);
}

public List<Integer> distanceK(TreeNode root, TreeNode target, int k) {
    List<Integer> res = new ArrayList<>();
    DFS(root, target, k, res);
    return res;
}
```



Burning Tree

BFS on graph

Infection Spread



```java
static int minTime = 0;
public static int DFS(Node root, int target){
    if(root == null) return -1;

    if(root.data == target){
        height(root, null, 0);
        return 1;
    }

    int left = DFS(root.left, target);
    if(left >= 0){
        height(root, root.left, left);
        return 1 + left;
    }

    int right = DFS(root.right, target);
    if(right >= 0){
        height(root, root.right, right);
        return 1 + right;
    }

    return -1;
}
```

```java
public static void height(Node root, Node blockage, int time){
    if(root == null || root == blockage) return;
    minTime = Math.max(minTime, time);
    height(root.left, blockage, time + 1);
    height(root.right, blockage, time + 1);
}

public static int minTime(Node root, int target)
{
    minTime = 0;
    DFS(root, target);
    return minTime;
}
```

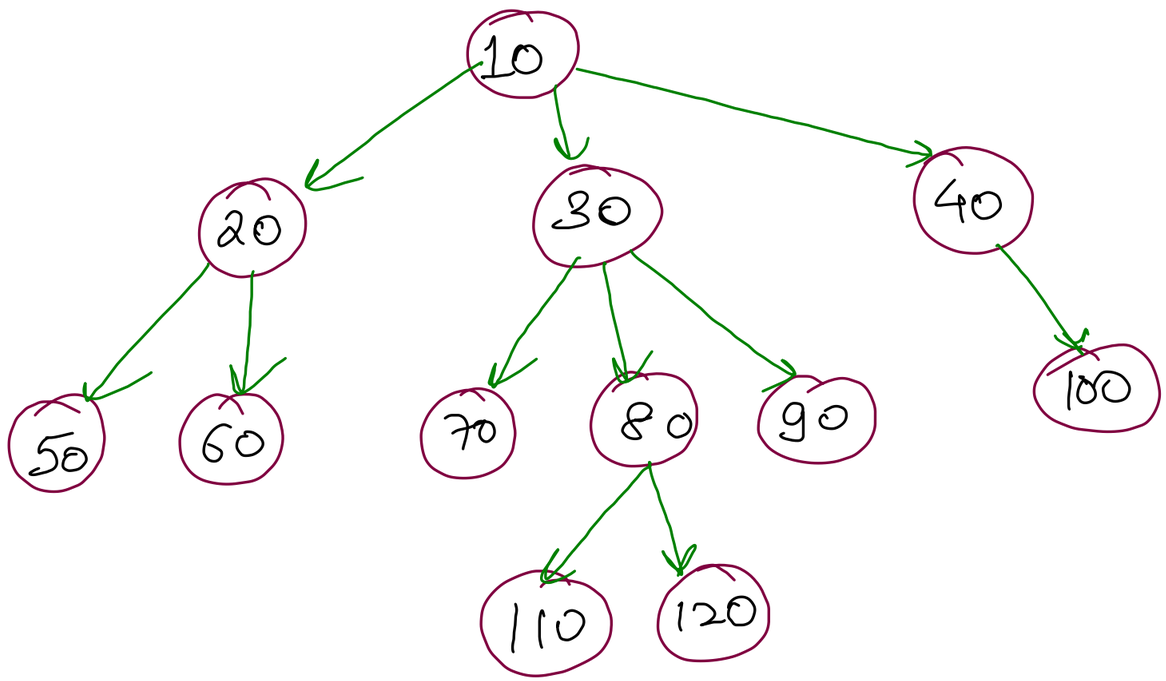
O(N) Time

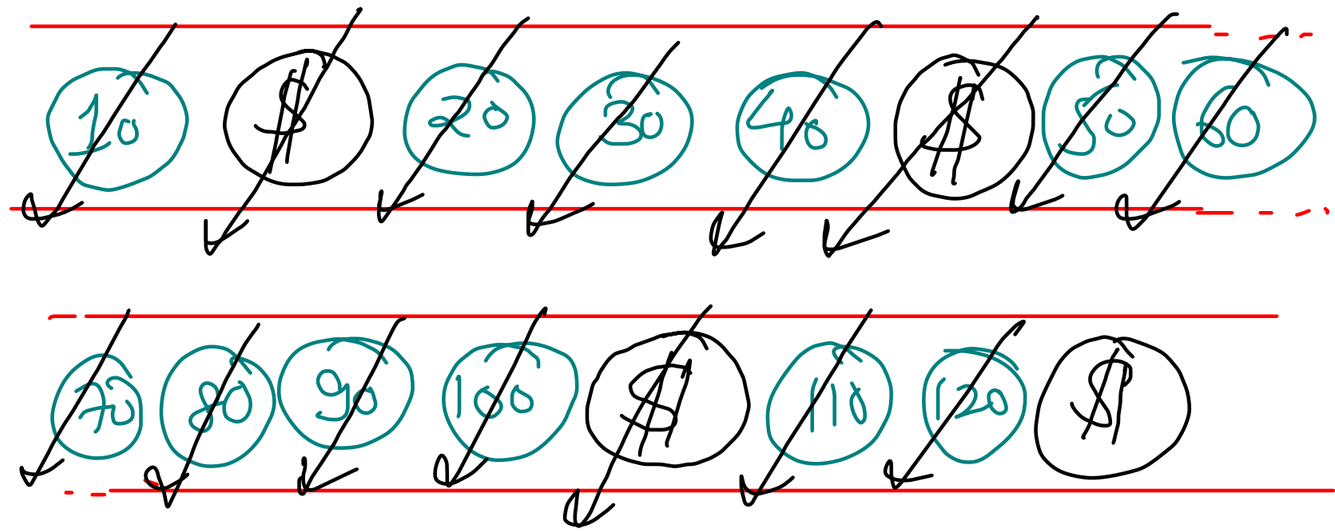
O(H) Recursion call stack

BFS

Level order Traversal & Variations

429





{ 10 }, { 20, 30, 40 }

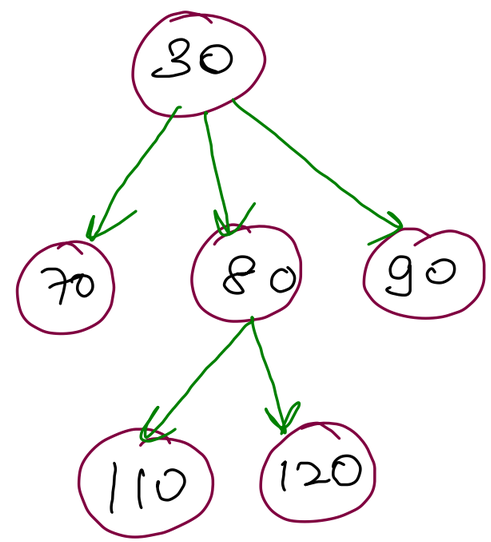
{ 50, 60, 70, 80, 90, 100 }

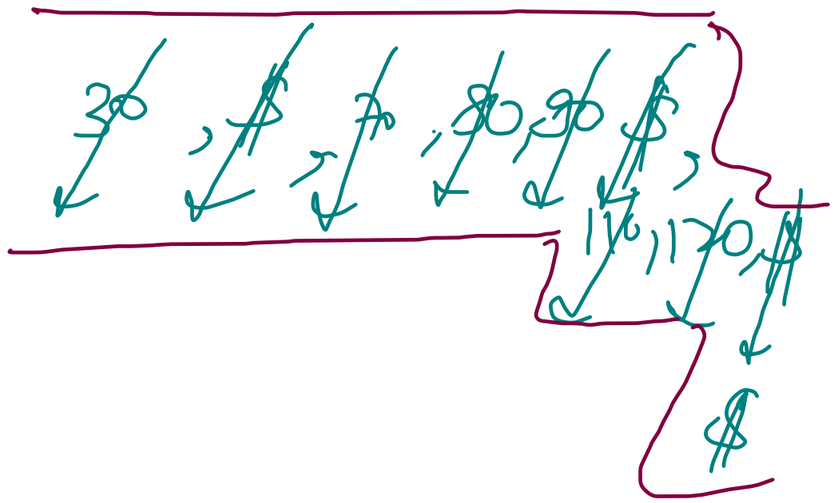
{ 110, 120 }

using marker node method

```java
public List<List<Integer>> levelOrder(Node root) {
    List<List<Integer>> res = new ArrayList<>();
    if(root == null) return res;

    Queue<Node> q = new ArrayDeque<>();
    q.add(root);
    Node marker = new Node(-1);
    q.add(marker);

    List<Integer> level = new ArrayList<>();
    while(q.size() > 1){
        Node curr = q.remove();
        if(curr == marker){
            res.add(level);
            level = new ArrayList<>();
            q.add(marker);
        }
        else {
            level.add(curr.val);
            for(Node child: curr.children){
                q.add(child);
            }
        }
    }
    if(level.size() > 0) res.add(level);
    return res;
}
```

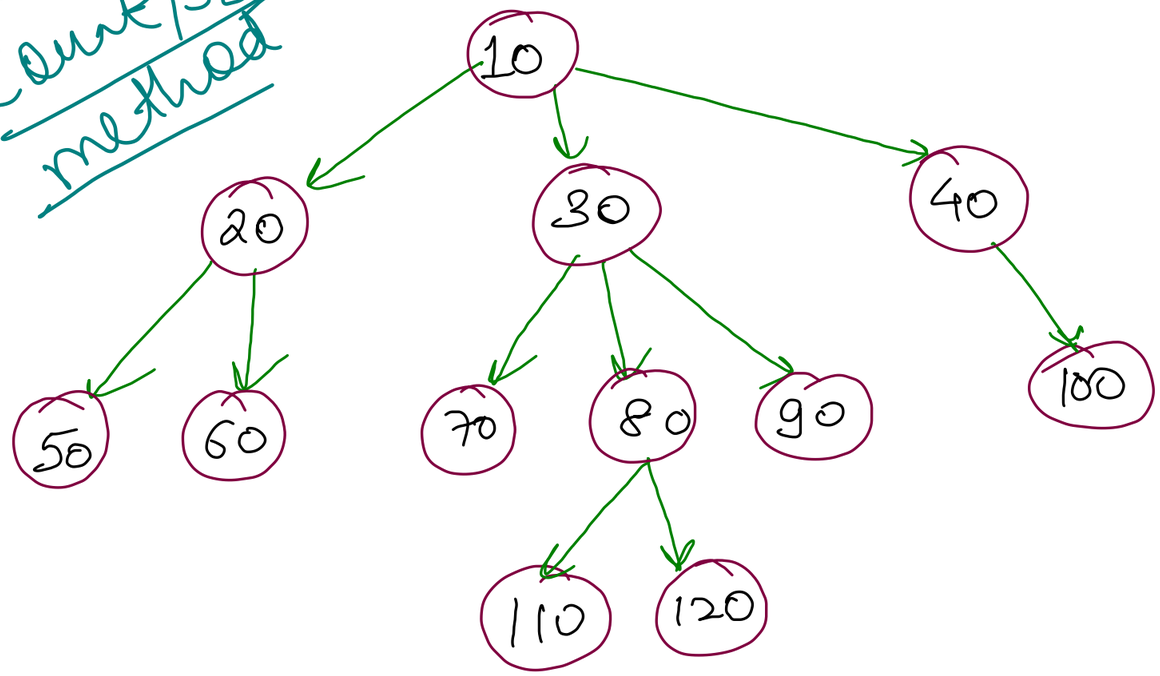
using marker node

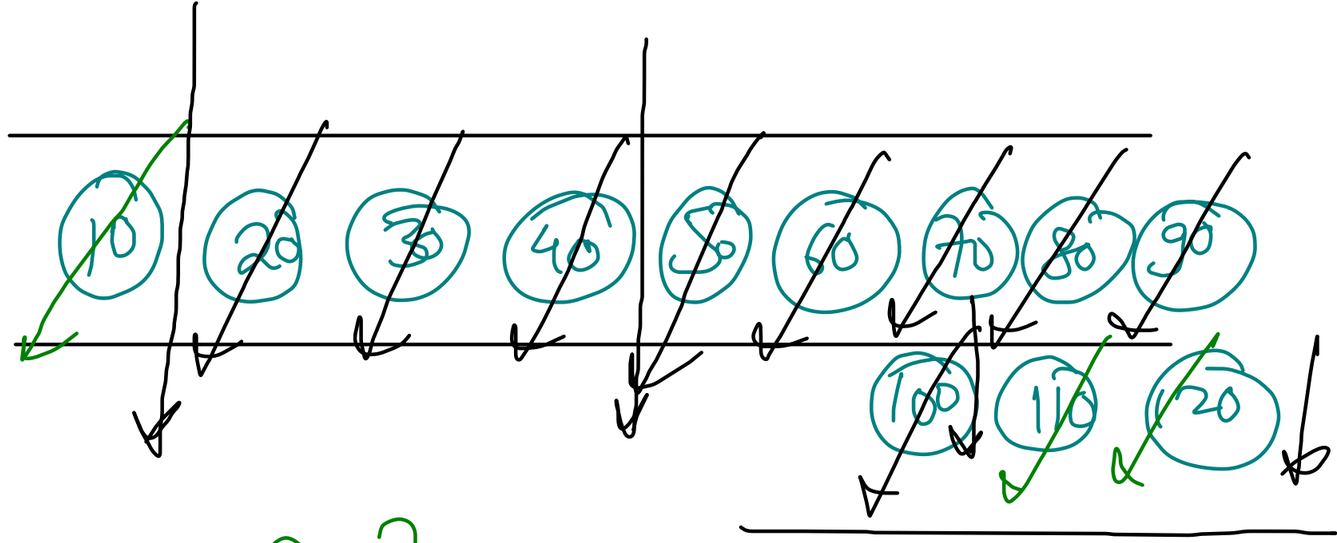




{{30} {70,80,90} {110,120}}

using count/size method





{10}

{20, 30, 40}

{50, 60, 70, 80, 90, 100}

{110, 120}

```
int count = q.size();
for(int i=0; i<count ; i++)
```

```java
public List<List<Integer>> levelOrder(Node root) {
    List<List<Integer>> res = new ArrayList<>();
    if(root == null) return res;

    Queue<Node> q = new ArrayDeque<>();
    q.add(root);

    while(q.size() > 0){
        int count = q.size();
        List<Integer> level = new ArrayList<>();
        while(count-- > 0){
            Node curr = q.remove();
            level.add(curr.val);
            for(Node child: curr.children){
                q.add(child);
            }
        }
        res.add(level);
    }

    return res;
}
```

using count method

O(N) time

O(breadth) extra space complexity

best case

worst case

O(N)

O(N)

{N/2}

even if tree is balanced

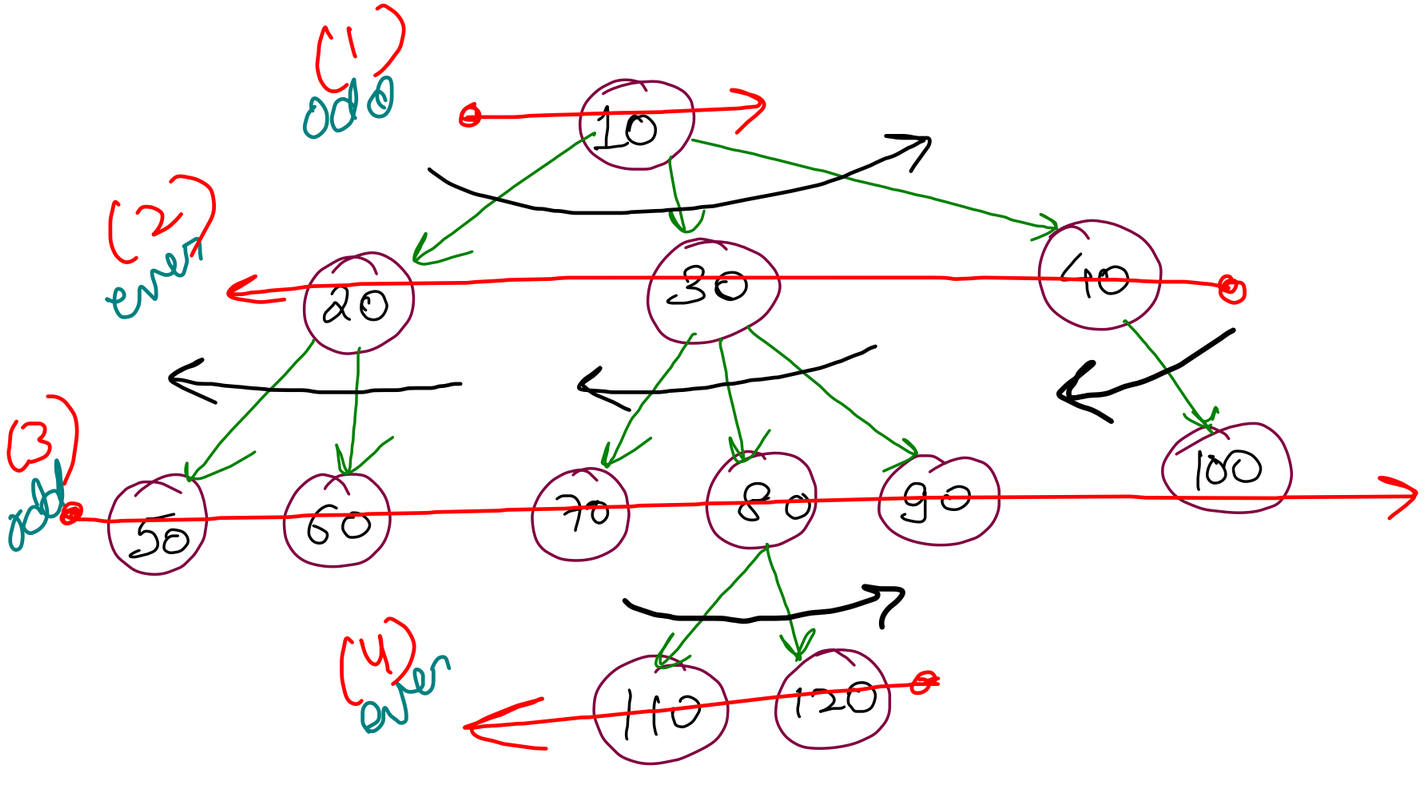
```java
public List<List<Integer>> levelOrder(TreeNode root) {
    List<List<Integer>> res = new ArrayList<>();
    if(root == null) return res;

    Queue<TreeNode> q = new ArrayDeque<>();
    q.add(root);

    while(q.size() > 0){
        int count = q.size();
        List<Integer> level = new ArrayList<>();
        while(count-- > 0){
            TreeNode curr = q.remove();
            level.add(curr.val);
            if(curr.left != null) q.add(curr.left);
            if(curr.right != null) q.add(curr.right);
        }
        res.add(level);
    }

    return res;
}
```

Binary Tree level order

102

O(N) time

O(N) Queue extra space



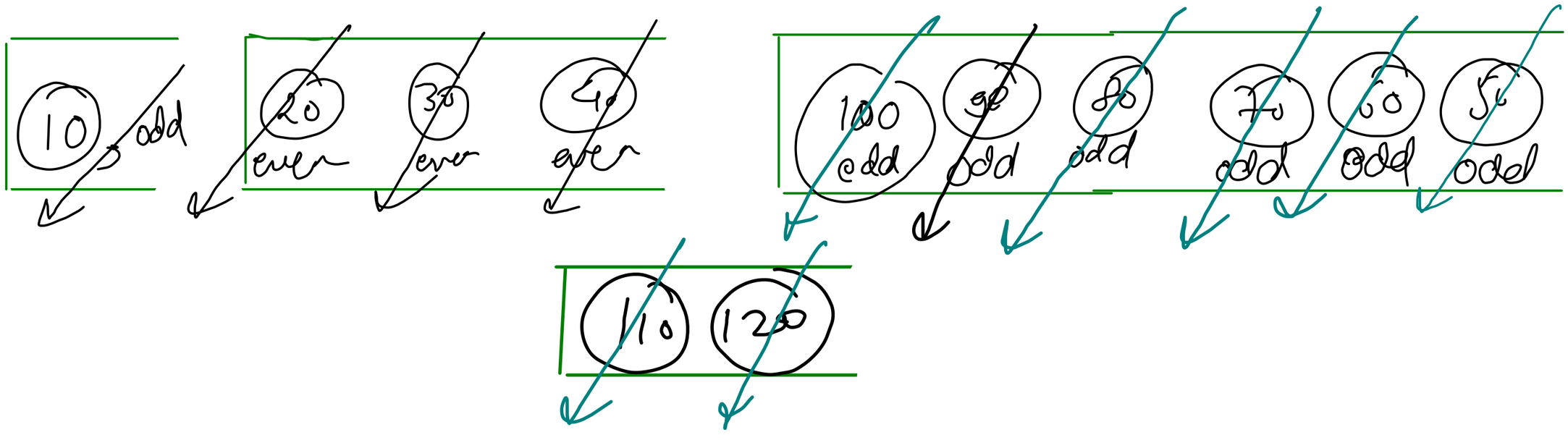
{10}

{40, 30, 20}

{50, 60, 70, 80, 90, 100}

{120, 110}

Zig Zag level order



```java
public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
    List<List<Integer>> res = new ArrayList<>();
    if(root == null) return res;

    Stack<TreeNode> odd = new Stack<>();
    Stack<TreeNode> even = new Stack<>();
    odd.push(root);
    int level = 1;

    while(odd.size() > 0 || even.size() > 0){
        List<Integer> oned = new ArrayList<>();
        if(level % 2 == 1){
            while(odd.size() > 0){
                TreeNode curr = odd.pop();
                oned.add(curr.val);

                if(curr.left != null) even.push(curr.left);
                if(curr.right != null) even.push(curr.right);
            }
        }
        else {
            while(even.size() > 0){
                TreeNode curr = even.pop();
                oned.add(curr.val);

                if(curr.right != null) odd.push(curr.right);
                if(curr.left != null) odd.push(curr.left);
            }
        }

        level++;
        res.add(oned);
    }
    return res;
}
```

O(N) time

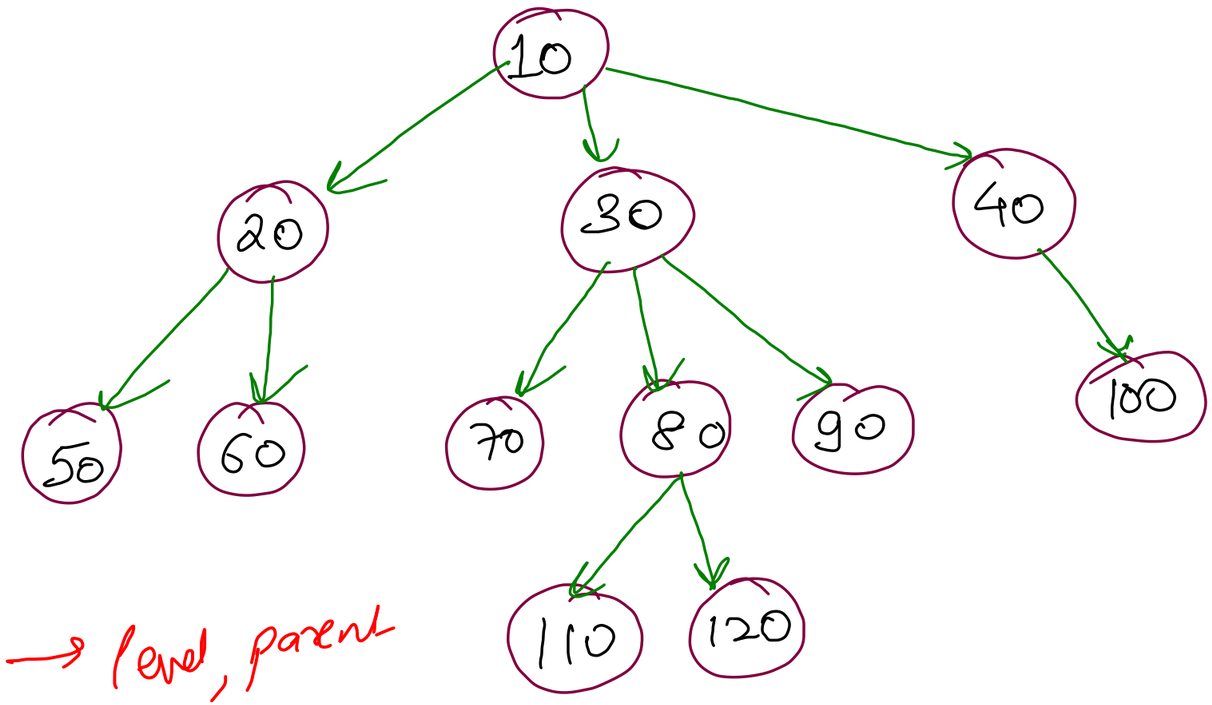
O(N) space

```java
public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
    List<List<Integer>> res = new ArrayList<>();
    if(root == null) return res;

    Stack<TreeNode> odd = new Stack<>();
    Stack<TreeNode> even = new Stack<>();
    odd.push(root);
    int level = 1;

    while(odd.size() > 0 || even.size() > 0){
        List<Integer> oned = new ArrayList<>();
        if(level % 2 == 1){
            while(odd.size() > 0){
                TreeNode curr = odd.pop();
                oned.add(curr.val);

                if(curr.left != null) even.push(curr.left);
                if(curr.right != null) even.push(curr.right);
            }
        }
        else {
            while(even.size() > 0){
                TreeNode curr = even.pop();
                oned.add(curr.val);

                if(curr.right != null) odd.push(curr.right);
                if(curr.left != null) odd.push(curr.left);
            }
        }

        level++;
        res.add(oned);
    }
    return res;
}
```

O(N) time

O(N) space

Cousins in Binary Tree



A → level, parent

B → level, parent

```
// If X.level != Y.level return false (Neither Sibling Nor Cousin)
// Else If X.Parent == Y.Parent return false (Sibling but not Cousin)
// Else Return true (Same Level & Different Parent -> Cousin)
```

nodes with same parent

sibling → 70, 90

80

cousins → 50, 60, 100

same level, should not be sibling

cousin (10, 40) → false

cousin (30, 30) → false

cousin (20, 30) → false

cousin (70, 80) → false

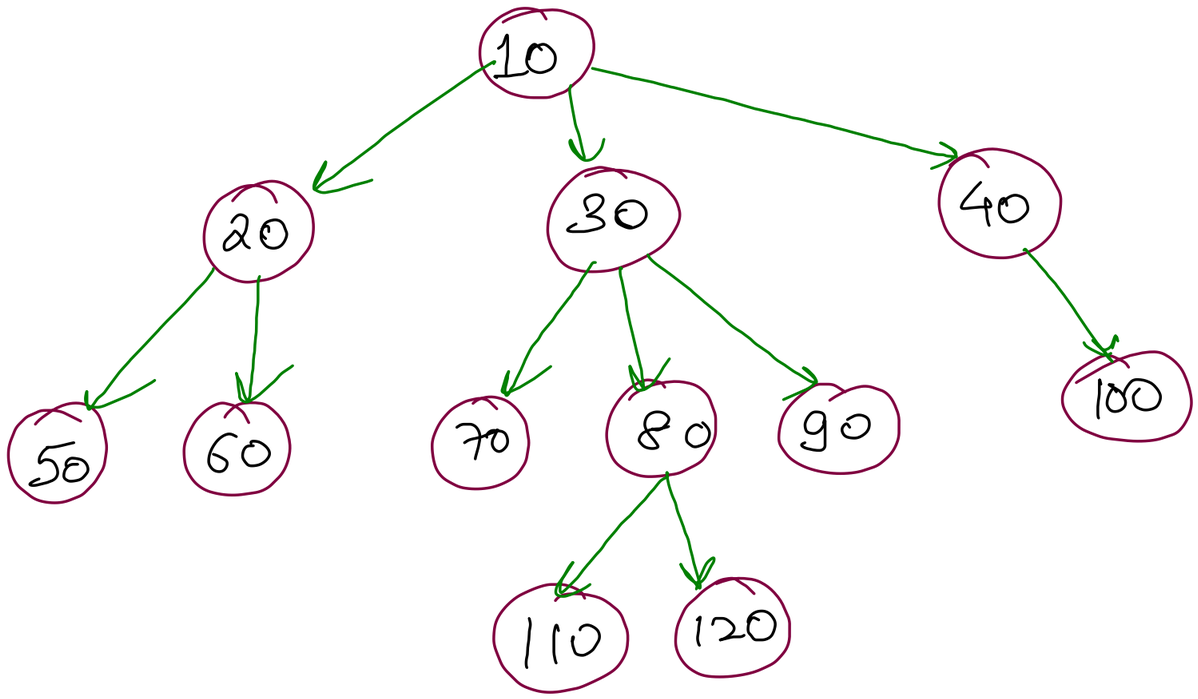
cousin (50, 80) → true

```
int xLevel = 0, yLevel = 0, xParent = -1, yParent = -1;

public void DFS(TreeNode root, int x, int y, int level){
    if(root == null) return;

    if(root.val == x) xLevel = level;
    if(root.val == y) yLevel = level;

    if(root.left != null){
        if(root.left.val == x) xParent = root.val;
        if(root.left.val == y) yParent = root.val;
    }

    if(root.right != null){
        if(root.right.val == x) xParent = root.val;
        if(root.right.val == y) yParent = root.val;
    }

    DFS(root.left, x, y, level + 1);
    DFS(root.right, x, y, level + 1);
}

public boolean isCousins(TreeNode root, int x, int y) {
    DFS(root, x, y, 0);

    if(x == y) return false;
    if(xLevel != yLevel) return false;
    if(xParent == yParent) return false;
    return true;
}
```

Approach 1

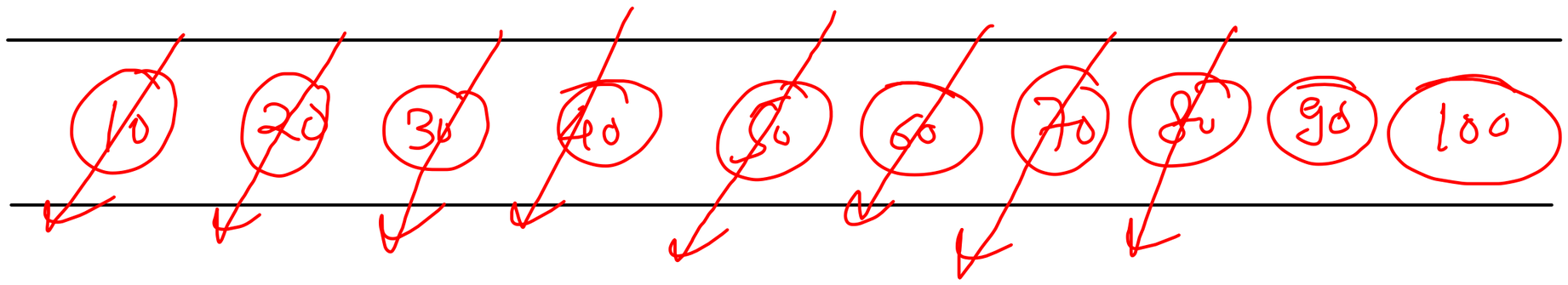
```
class Solution {
    int xLevel = 0, yLevel = 0, xParent = -1, yParent = -1;

    public void DFS(TreeNode root, int x, int y, int level, int parent){
        if(root == null) return;

        if(root.val == x) {
            xLevel = level;
            xParent = parent;
        }
        if(root.val == y){
            yLevel = level;
            yParent = parent;
        }

        DFS(root.left, x, y, level + 1, root.val);
        DFS(root.right, x, y, level + 1, root.val);
    }

    public boolean isCousins(TreeNode root, int x, int y) {
        DFS(root, x, y, 0, -1);

        if(x == y) return false;
        if(xLevel != yLevel) return false;
        if(xParent == yParent) return false;
        return true;
    }
}
```

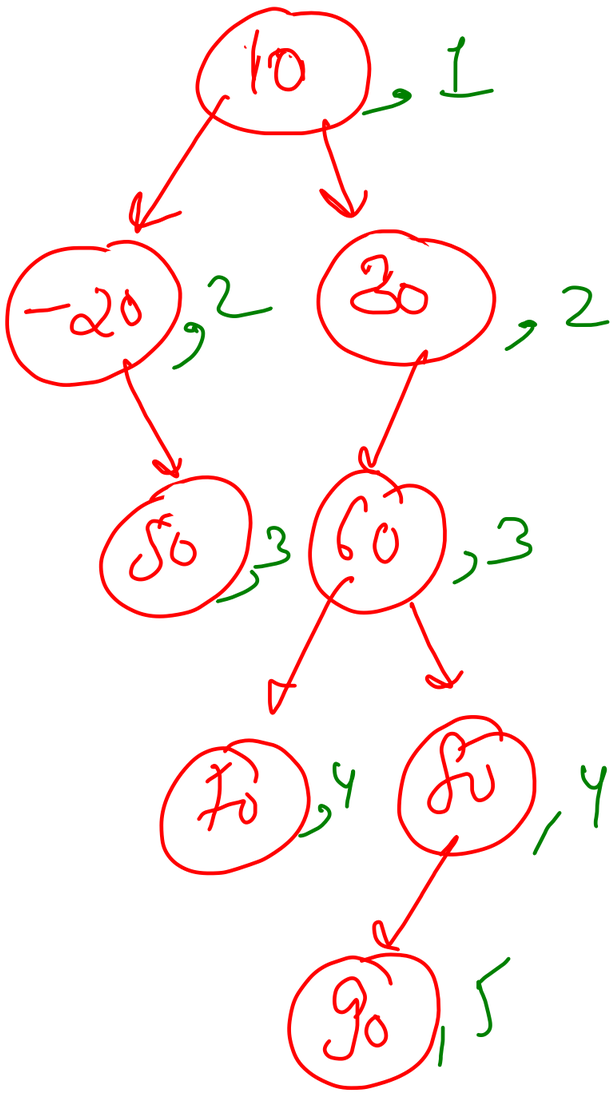
Approach (2)

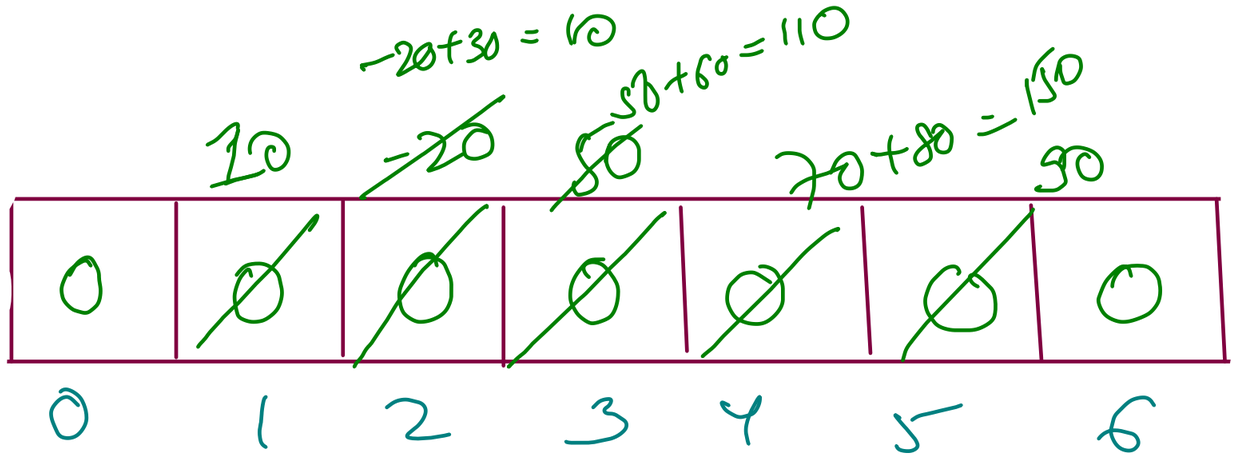


① $x = 50,\ y = 80 \rightarrow$ true

② $x = 70,\ y = 80 \Rightarrow$ false







O(N) time

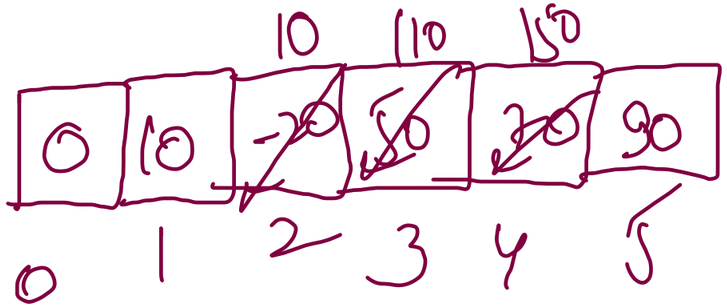
O(H) Extra Space (for ArrayList)

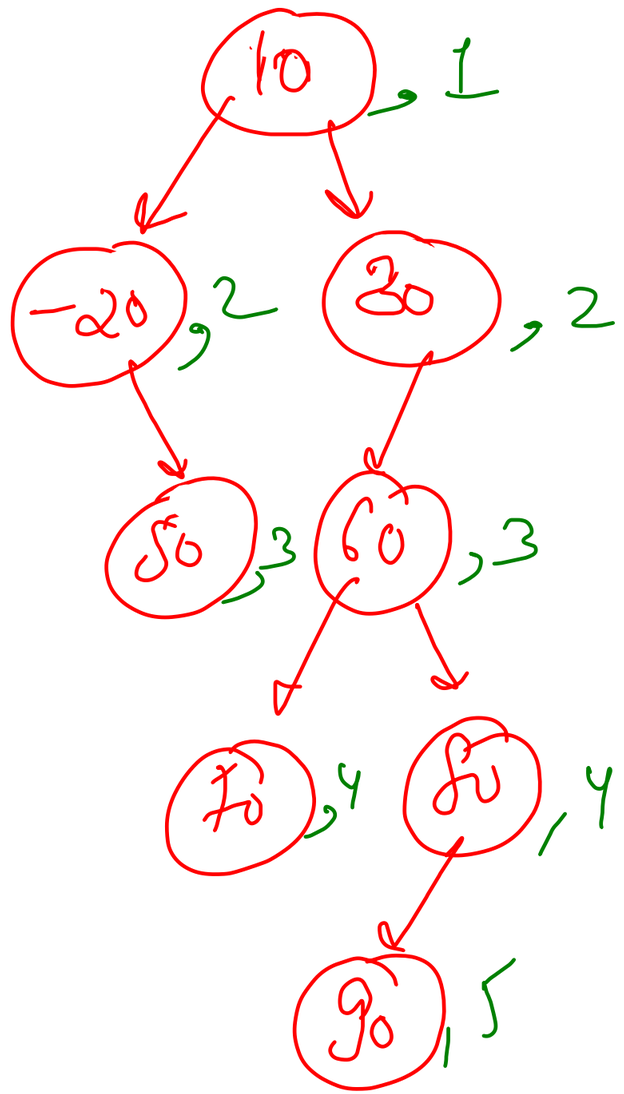
O(H) Recursion call stack

```java
public void DFS(TreeNode root, ArrayList<Integer> levelSum, int level){
    if(root == null) return;

    if(levelSum.size() > level)
        levelSum.set(level, levelSum.get(level) + root.val);
    else levelSum.add(root.val);

    DFS(root.left, levelSum, level + 1);
    DFS(root.right, levelSum, level + 1);
}
public int maxLevelSum(TreeNode root) {
    ArrayList<Integer> levelSum = new ArrayList<>();
    levelSum.add(0);
    DFS(root, levelSum, 1);

    int maxSumLevel = 1;
    for(int i=1; i<levelSum.size(); i++){
        if(levelSum.get(i) > levelSum.get(maxSumLevel)){
            maxSumLevel = i;
        }
    }
    return maxSumLevel;
}
```

O(H) height

smallest level with max^m level





Binary Tree - level (2) lecture (6)

→ Next Right Pointer – I, II

→ Diagonal Order – I, II

→ Vertical order – I, II

→ left View, Right View

→ Top View, Bottom View

→ Boundary Traversal

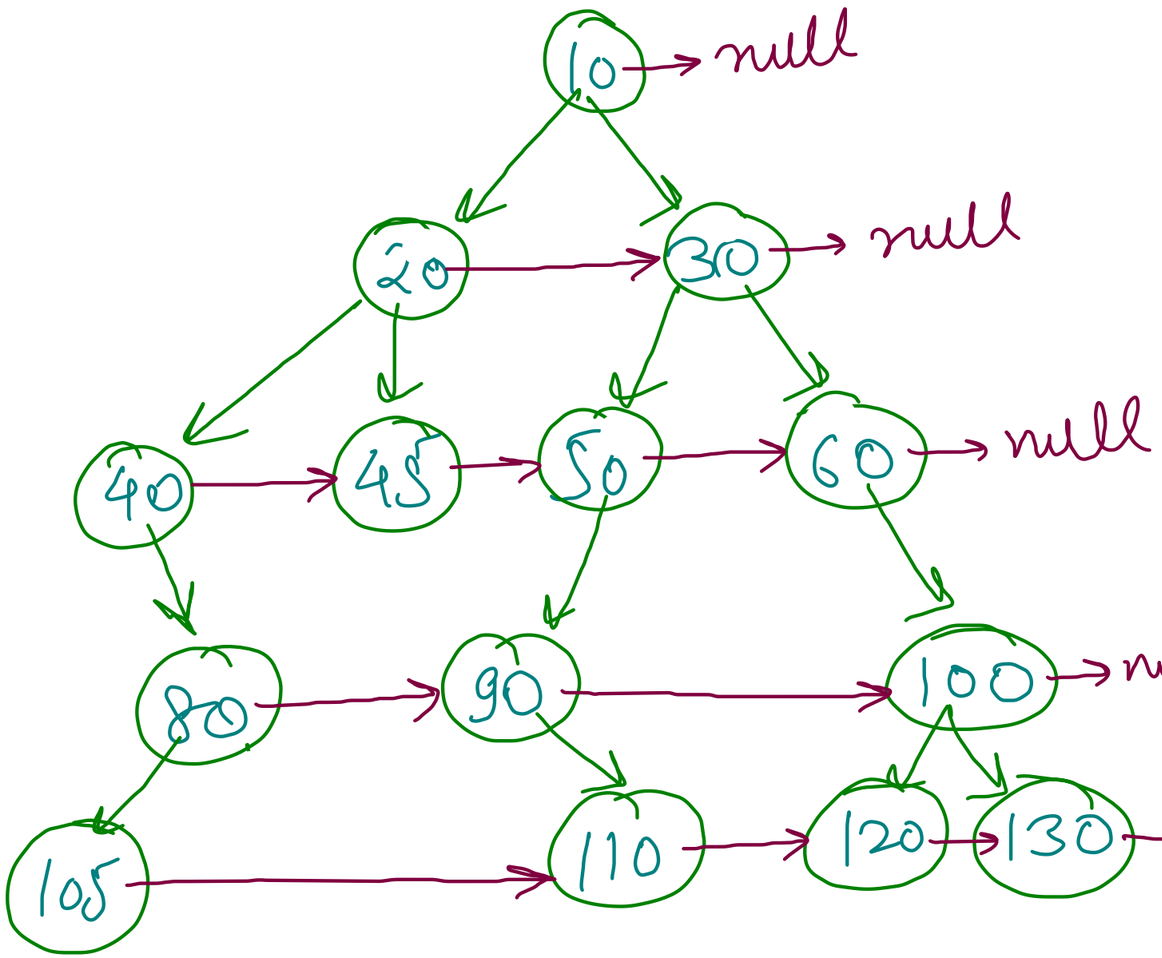
Pre-requisite

→ DFS (Preorder)

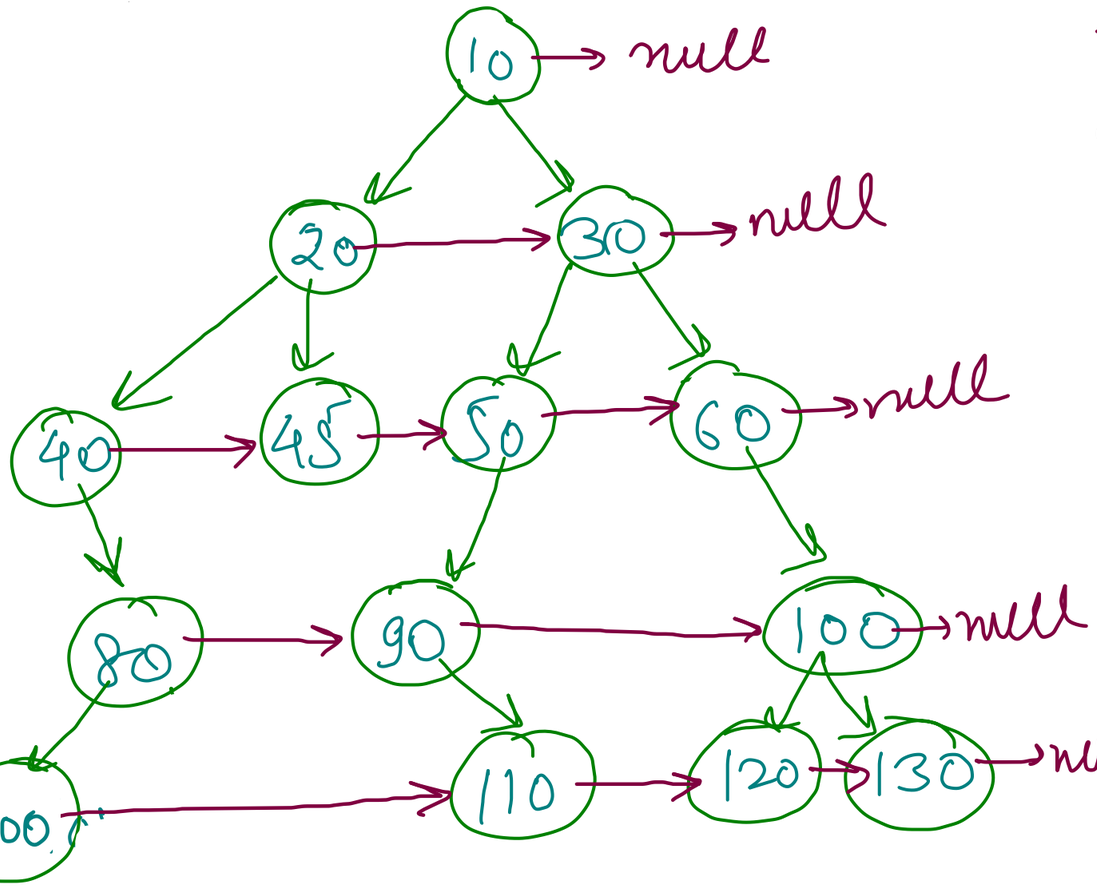
→ BFS (level Order)

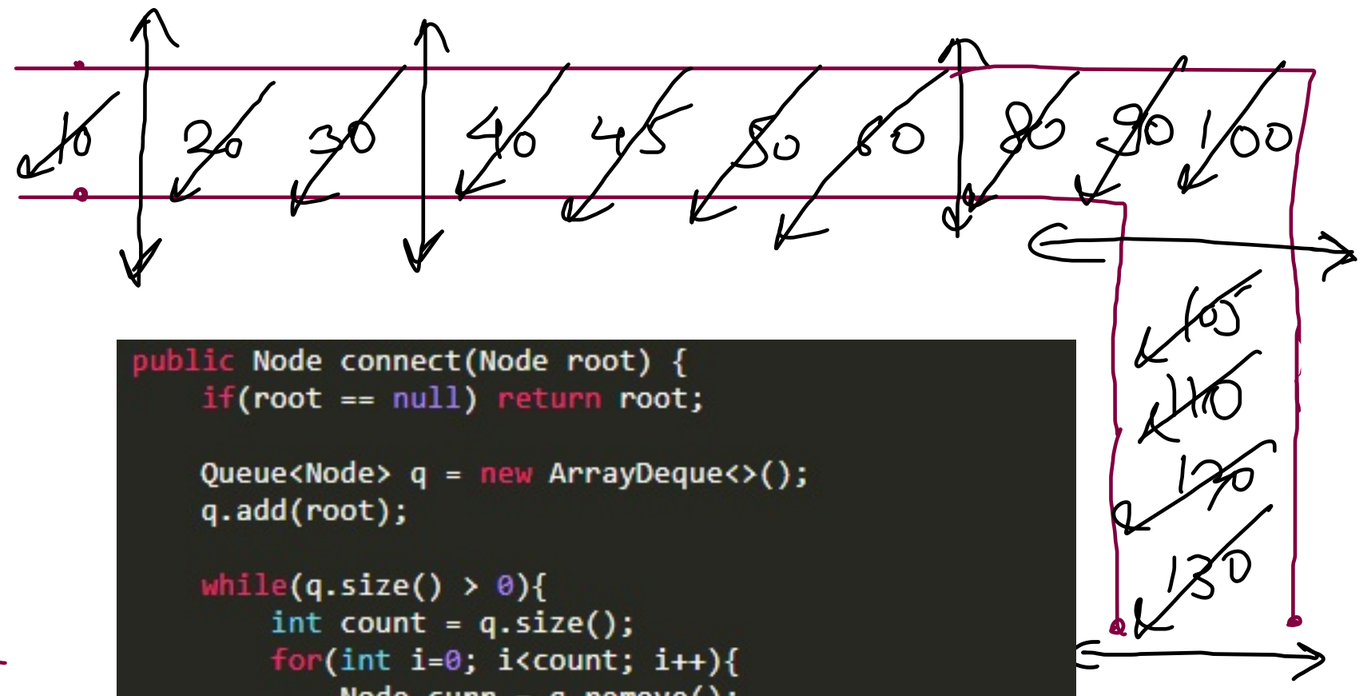
Populating Next Right Pointer

{ 116 and 117 }

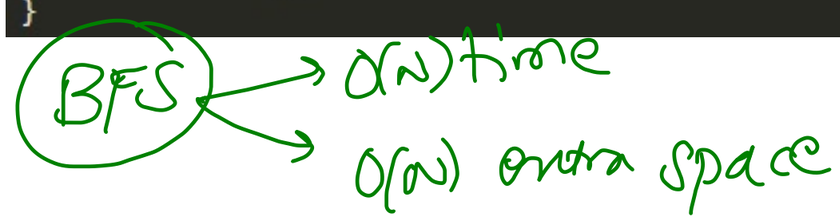


- data
- left
- right
- next



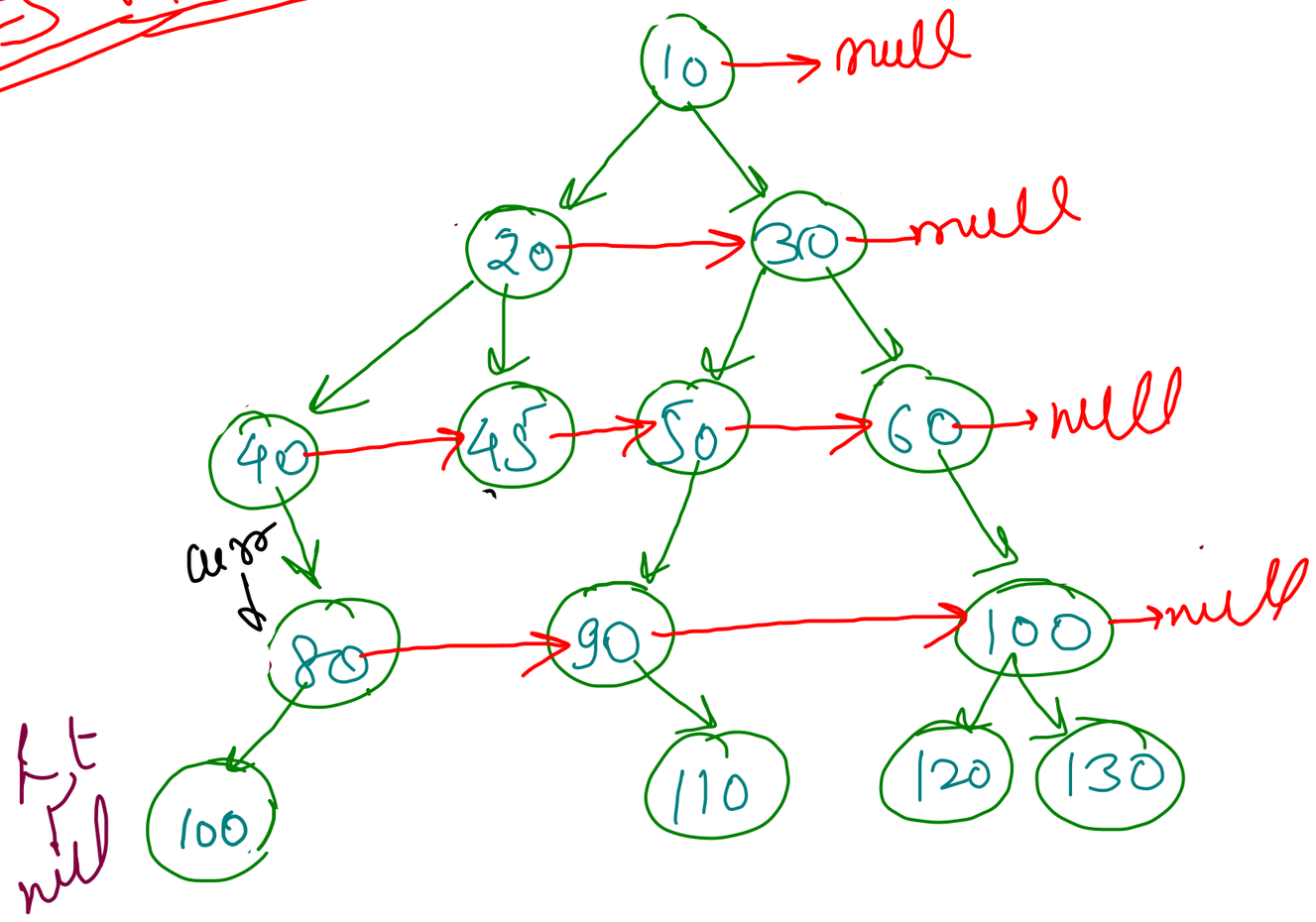


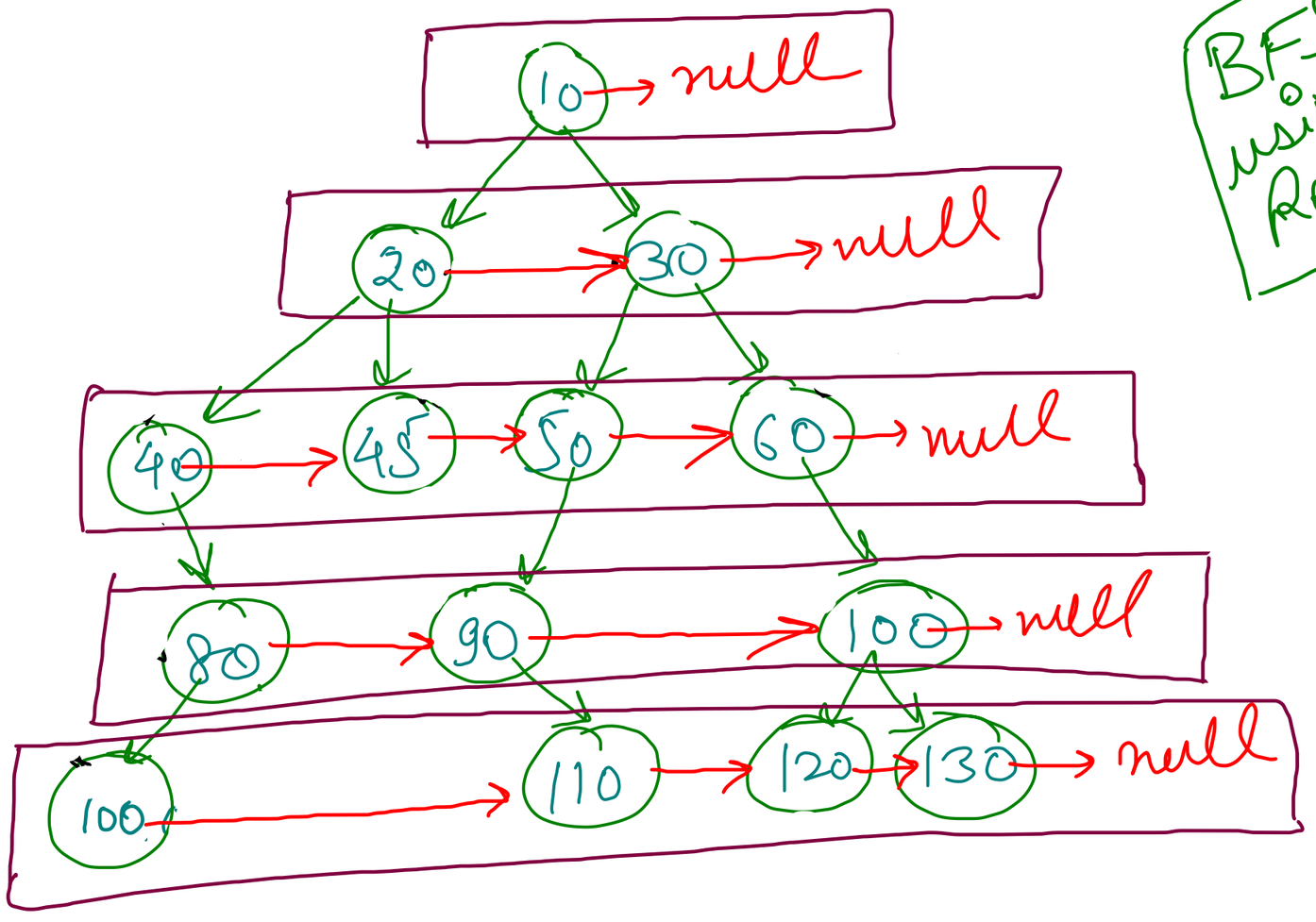

```java
public Node connect(Node root) {
    if(root == null) return root;

    Queue<Node> q = new ArrayDeque<>();
    q.add(root);

    while(q.size() > 0){
        int count = q.size();
        for(int i=0; i<count; i++){
            Node curr = q.remove();

            // Populate Next Right Pointer of Curr
            if(i < count - 1) curr.next = q.peek();
            if(curr.left != null) q.add(curr.left);
            if(curr.right != null) q.add(curr.right);
        }
    }

    return root;
}
```



BFS W/O Queue

```java
public Node connect(Node root){
    Node curr = root;
    while(curr != null){
        Node head = null, tail = null;

        // Travel on Current Level and Construct Next Level
        while(curr != null){
            if(curr.left != null){
                if(head == null)
                    head = tail = curr.left;
                else {
                    tail.next = curr.left;
                    tail = tail.next;
                }
            }
            if(curr.right != null){
                if(head == null)
                    head = tail = curr.right;
                else {
                    tail.next = curr.right;
                    tail = tail.next;
                }
            }

            curr = curr.next;
        }

        curr = head; // Update current as Head of Next Level
    }

    return root;
}
```





BFS using Recursion

linked list type + level Order

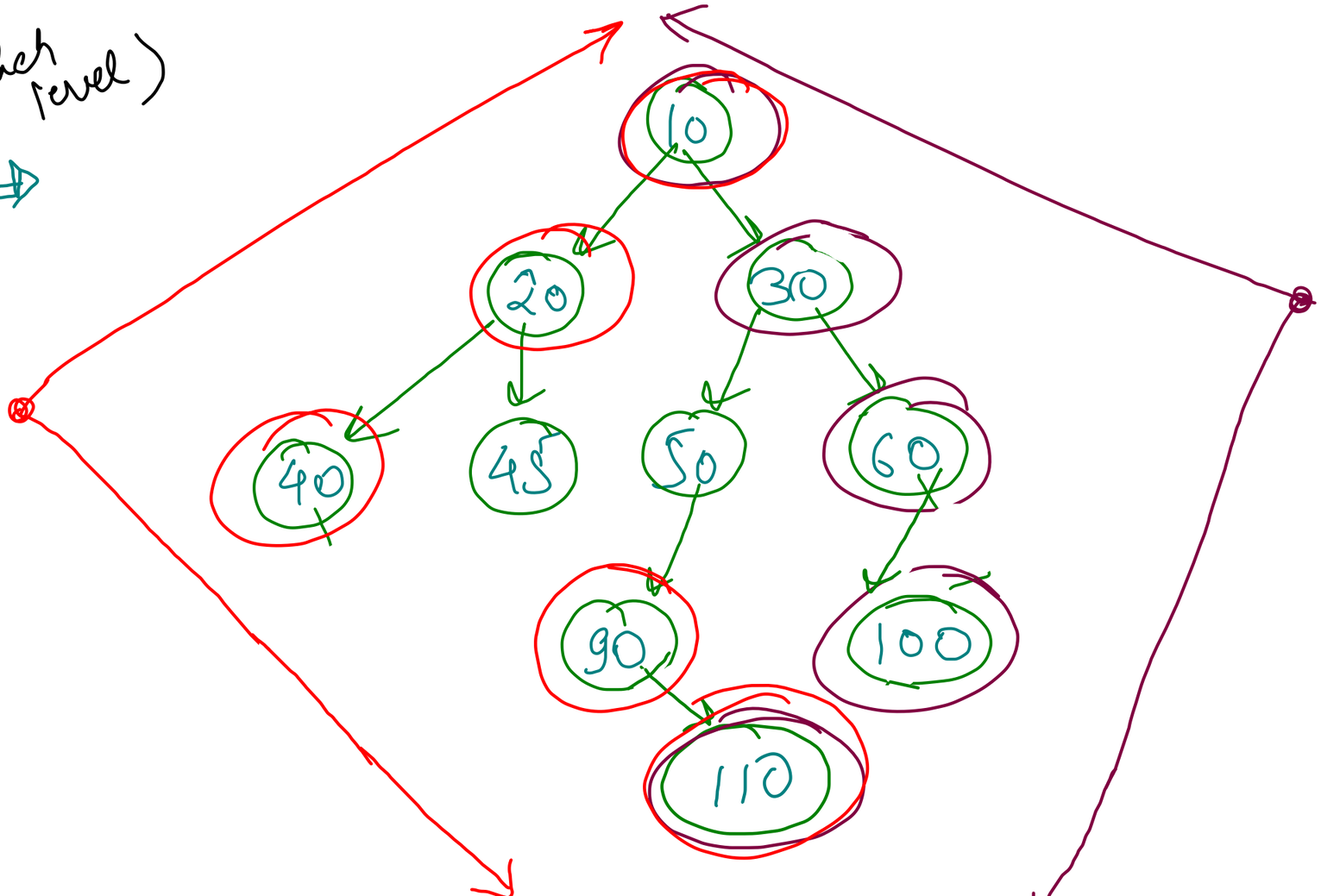
{ on traversing ith level populate next pointers of $(i+1)^{th}$ level using head & tail strategy }

Left View, Right View

(first node of each level)

left view →

10
20
40
90
100



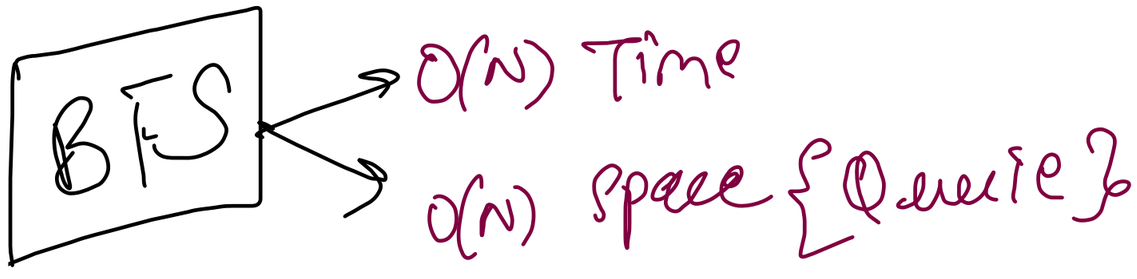
(last node of each level)

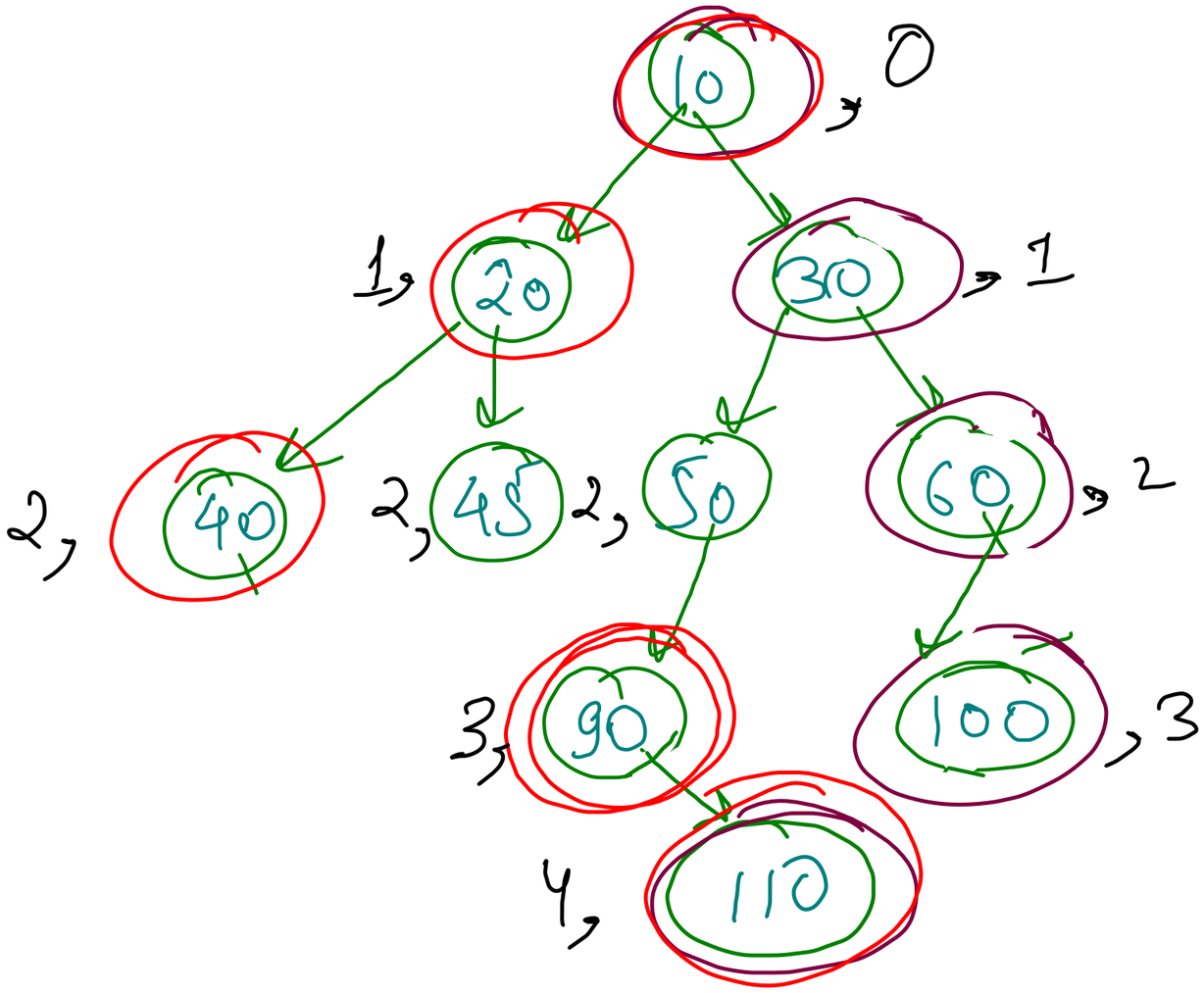
← right side

10
30
60
100
110

```java
ArrayList<Integer> leftView(Node root)
{
    ArrayList<Integer> leftView = new ArrayList<>();
    if(root == null) return leftView;

    Queue<Node> q = new ArrayDeque<>();
    q.add(root);

    while(q.size() > 0){
        int count = q.size();
        for(int i=0; i<count; i++){
            Node curr = q.remove();

            if(i == 0) leftView.add(curr.data);
            if(curr.left != null) q.add(curr.left);
            if(curr.right != null) q.add(curr.right);
        }
    }

    return leftView;
}
```

```java
public List<Integer> rightSideView(TreeNode root) {
    List<Integer> rightView = new ArrayList<>();
    if(root == null) return rightView;

    Queue<TreeNode> q = new ArrayDeque<>();
    q.add(root);

    while(q.size() > 0){
        int count = q.size();
        for(int i=0; i<count; i++){
            TreeNode curr = q.remove();

            if(i == count - 1) rightView.add(curr.val);
            if(curr.left != null) q.add(curr.left);
            if(curr.right != null) q.add(curr.right);
        }
    }

    return rightView;
}
```





10	20	40	90	110

Left view

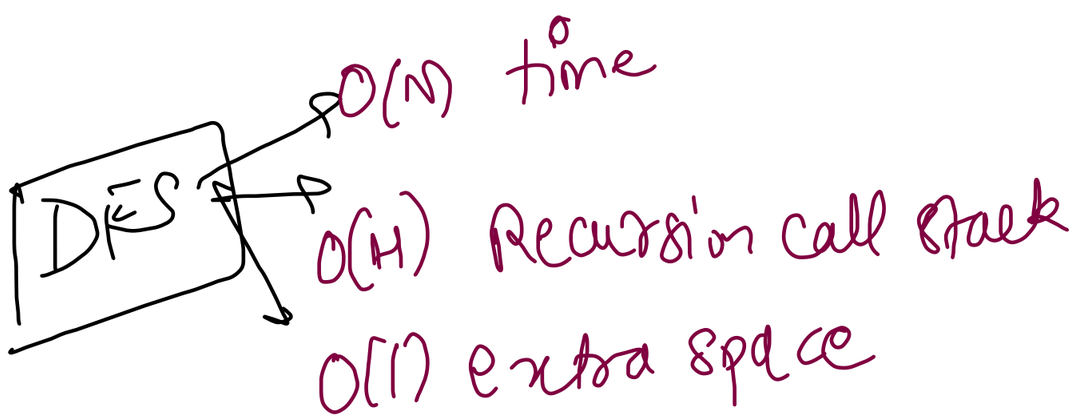
60
~~50~~
~~45~~
100

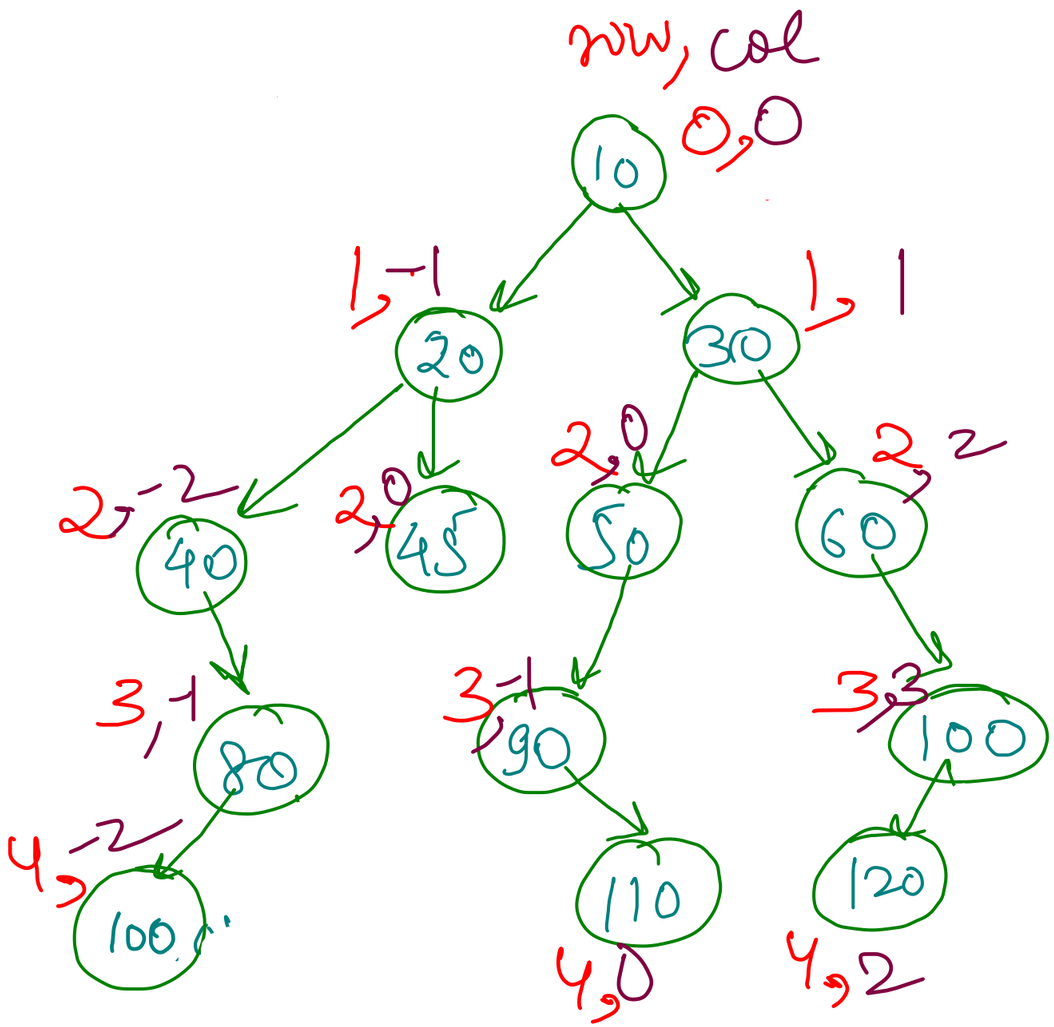
10	~~20~~ 30	~~40~~	~~90~~	110

right view

```java
public void DFS(TreeNode root, List<Integer> rightView, int level){
    if(root == null) return;

    if(level >= rightView.size())
        rightView.add(root.val);
    else rightView.set(level, root.val);

    DFS(root.left, rightView, level + 1);
    DFS(root.right, rightView, level + 1);
}

public List<Integer> rightSideView(TreeNode root) {
    List<Integer> rightView = new ArrayList<>();
    DFS(root, rightView, 0);
    return rightView;
}
```

```java
public void DFS(Node root, ArrayList<Integer> leftView, int level){
    if(root == null) return;

    if(level >= leftView.size())
        leftView.add(root.data);

    DFS(root.left, leftView, level + 1);
    DFS(root.right, leftView, level + 1);
}

ArrayList<Integer> leftView(Node root)
{
    ArrayList<Integer> leftView = new ArrayList<>();
    DFS(root, leftView, 0);
    return leftView;
}
```





level 0

level 1

level 2

level 3

level 4

col → vertical order

-2 ⇒ { 40, 100 }

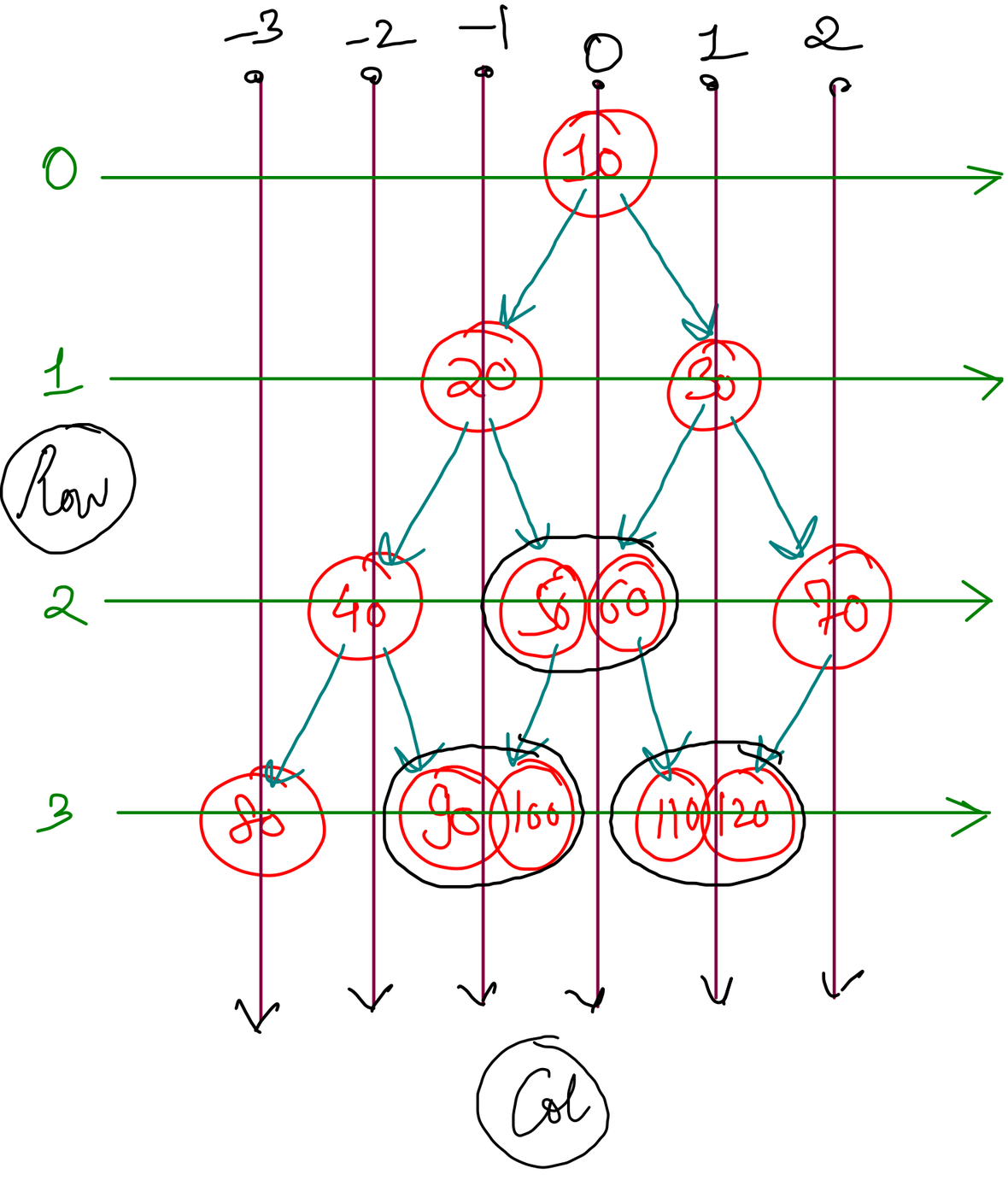
-1 ⇒ { 20, 80, 90 }

0 ⇒ { 10, 45, 50, 110 }

1 ⇒ { 30 }

2 ⇒ { 60, 120 }

3 ⇒ { 100 }



Key (Integer)

Value

{ ArrayList <Integer> }

-3 → { 80 }

-2 → { 40 }

-1 → { 20, 90, 100 }

0 → { 10, 50, 60 }

1 → { 30, 110, 120 }

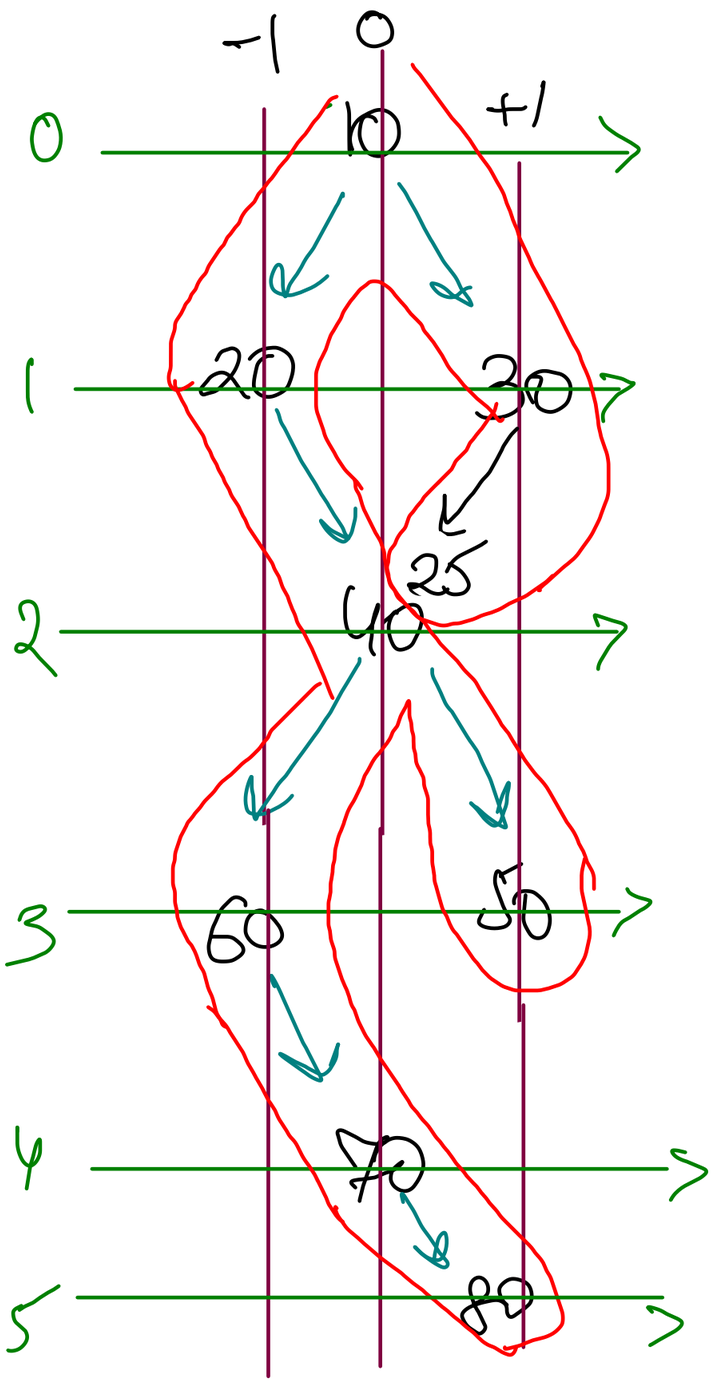
2 → { 70 }

Same Vertical order

↓

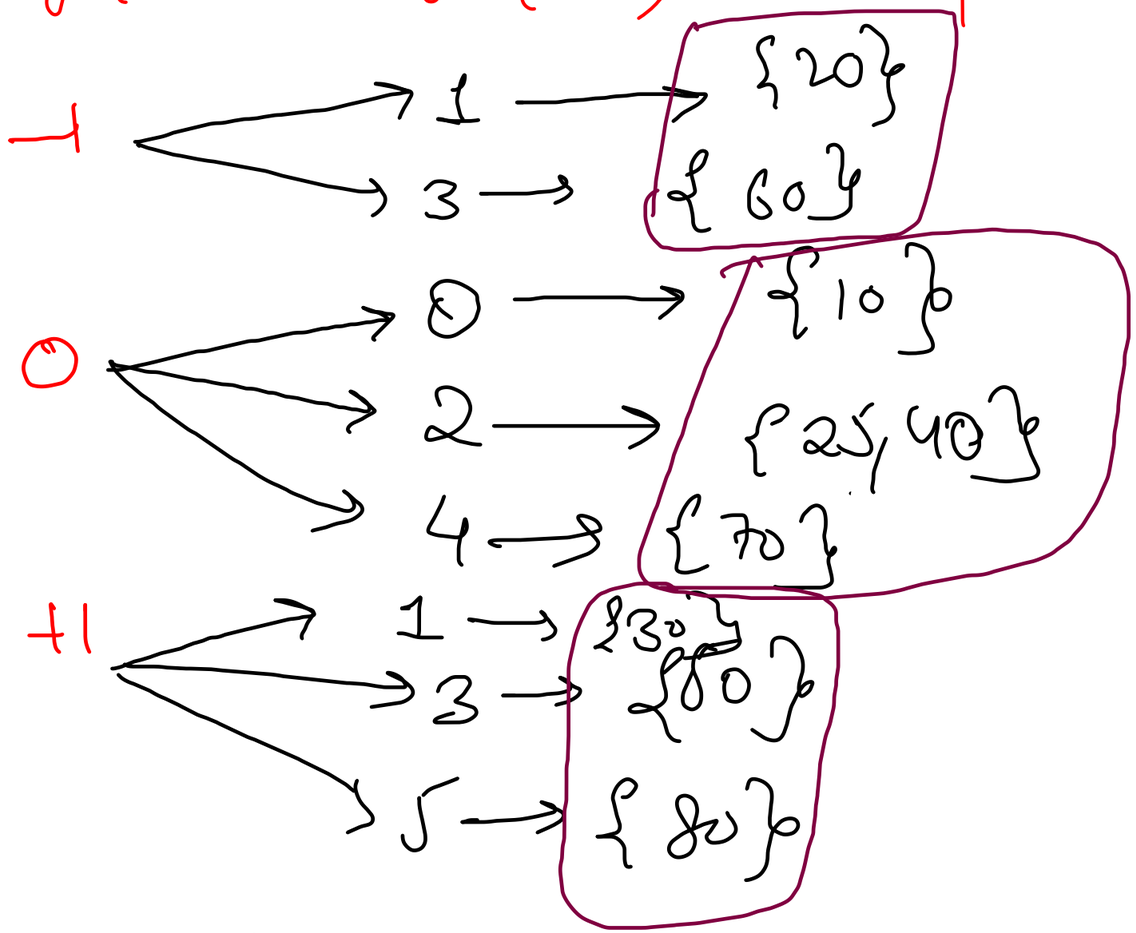
Elements in the level order

If h_0, v_0 is same, increasing set



2D (Nested) hashmap

key1(col) → key2(row) → value{TreeSet}



"Level order cannot be maintained in DFS"

```java
TreeMap<Integer, TreeMap<Integer, ArrayList<Integer>>> vertical;

public void DFS(TreeNode root, int row, int col){
    if(root == null) return;

    if(vertical.containsKey(col) == false)
        vertical.put(col, new TreeMap<>());

    if(vertical.get(col).containsKey(row) == false)
        vertical.get(col).put(row, new ArrayList<>());

    vertical.get(col).get(row).add(root.val);

    DFS(root.left, row + 1, col - 1);
    DFS(root.right, row + 1, col + 1);
}
```

```java
public List<List<Integer>> verticalTraversal(TreeNode root) {
    vertical = new TreeMap<>();
    DFS(root, 0, 0);

    List<List<Integer>> res = new ArrayList<>();
    for(Integer col: vertical.keySet()){
        TreeMap<Integer, ArrayList<Integer>> curr = vertical.get(col);
        List<Integer> res1d = new ArrayList<>();
        for(Integer row: curr.keySet()){
            ArrayList<Integer> oned = curr.get(row);
            Collections.sort(oned);
            for(Integer val: oned){
                res1d.add(val);
            }
        }
        res.add(res1d);
    }
    return res;
}
```

```cpp
class Solution {
public:
    map<int,map<int,set<int>>> m;
    void solve(TreeNode* root,int row,int col)
    {
        if(root == NULL)
            return;
        m[col][row].insert(root->val);
        solve(root->left,row + 1, col - 1);
        solve(root->right,row + 1, col + 1);
    }
    vector<vector<int>> verticalTraversal(TreeNode* root) {
        vector<vector<int>> res;
        if(root == NULL)
            return res;
        solve(root,0,0);
        for(auto i:m)
        {
            vector<int> arr;
            for(auto j: i.second)
            {
                for(auto k: j.second)
                    arr.push_back(k);
            }
            res.push_back(arr);
        }
        return res;
    }
};
```

→ Next Right Pointer – I, II

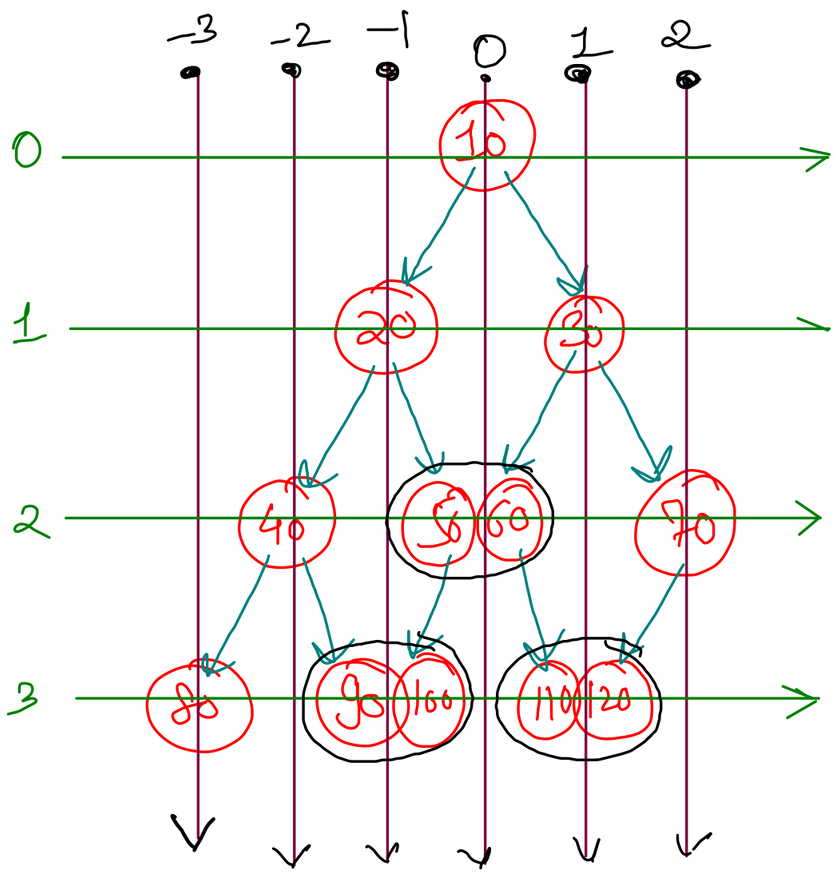
→ Vertical order – I, II

→ left View, Right View

→ Top View, Bottom View

→ Diagonal Order – I, II

→ Boundary Traversal



hashmap for Top View

-3 → 3 → {40}
-2 → 2 → {40}
-1 → 1 → {20}
0 → 0 → {10}
1 → 1 → {30}
2 → 2 → {70}

hashmap for bottom view

-3 → 3 → {80}
-2 → 2 → {40}
-1 → 3 → {90, 100}
0 → 2 → {50, 60}
1 → 3 → {110, 120}
2 → 2 → {70}

TreeMap<Integer, Pair<Integer, Array(sk(Int))>>
col row Nodes

(#) TOP VIEW {Min row nodes of each col}

-3 ⇒ 80
-2 ⇒ 40
-1 ⇒ 20
0 ⇒ 10
1 ⇒ 30
2 ⇒ 70

(#) BOTTOM VIEW {Max row node of each col}

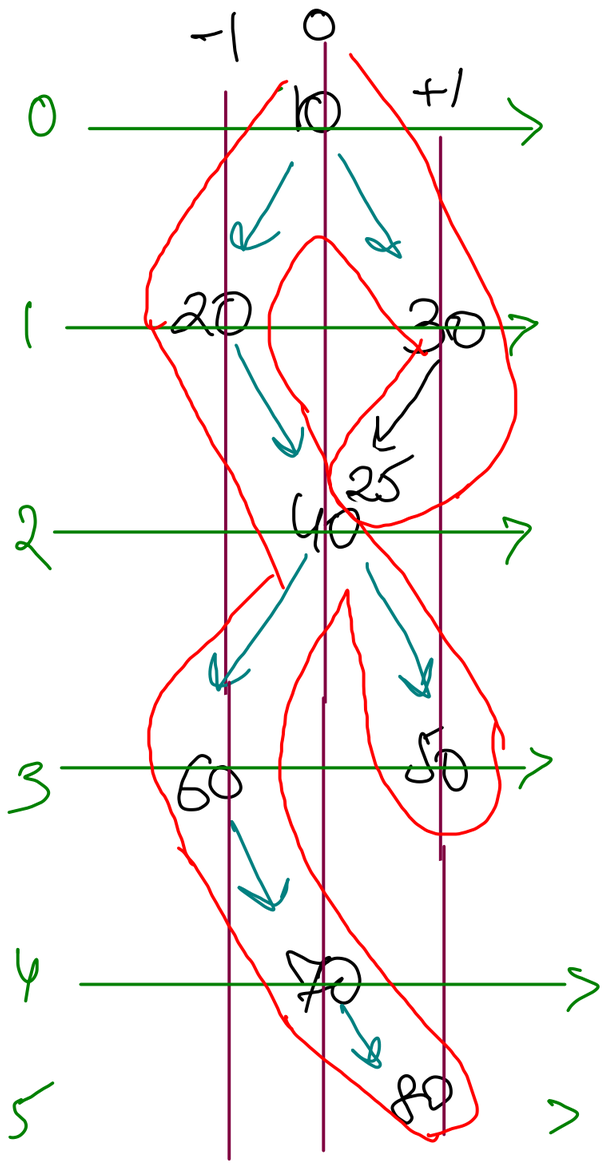
-3 ⇒ 80
-2 ⇒ 40
-1 ⇒ 90, 100
0 ⇒ 50, 60
1 ⇒ 110, 120
2 ⇒ 70

(x, y)
(x+1, y-1) (x+1, y+1)

```java
public static class Pair{
    int data;
    int row;
    Pair(int data, int row){
        this.data = data;
        this.row = row;
    }
}
// Column (Vertical Level) as Key vs Row & Its Elements in Pair
static TreeMap<Integer, Pair> vertical;

static void DFS(Node root, int row, int col){
    if(root == null) return;
    if(vertical.containsKey(col) == false){
        vertical.put(col, new Pair(root.data, row));
    } else if(vertical.get(col).row > row) {
        vertical.get(col).row = row;
        vertical.get(col).data = root.data;
    }

    DFS(root.left, row + 1, col - 1);
    DFS(root.right, row + 1, col + 1);
}

static ArrayList<Integer> topView(Node root)
{
    vertical = new TreeMap<>();
    DFS(root, 0, 0);
    ArrayList<Integer> topView = new ArrayList<>();
    for(Integer key: vertical.keySet()){
        topView.add(vertical.get(key).data);
    }
    return topView;
}
```

TopView (DFS)

Java



O(N + C log C)

hashmap
Integer, Pair

-1 → (1, 20)

0 → (0, 10)

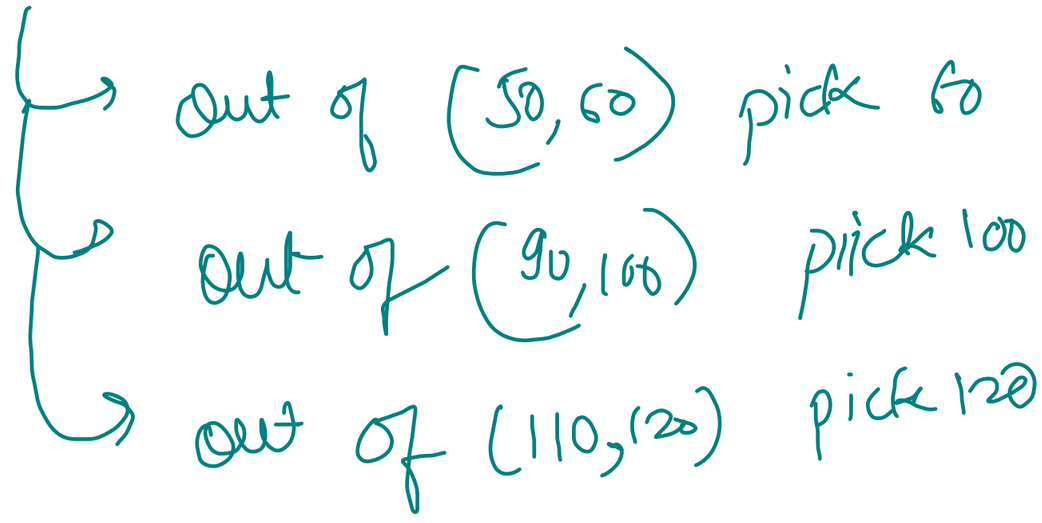
+1 → (1, 30)

```java
public static class Pair{
    int data;
    int row;
    Pair(int data, int row){
        this.data = data;
        this.row = row;
    }
}
// Column (Vertical Level) as Key vs Row & Its Elements in Pair
TreeMap<Integer, Pair> vertical;

void DFS(Node root, int row, int col){
    if(root == null) return;
    if(vertical.containsKey(col) == false){
        vertical.put(col, new Pair(root.data, row));
    } else if(vertical.get(col).row <= row) {
        vertical.get(col).row = row;
        vertical.get(col).data = root.data;
    }

    DFS(root.left, row + 1, col - 1);
    DFS(root.right, row + 1, col + 1);
}

public ArrayList <Integer> bottomView(Node root)
{
    vertical = new TreeMap<>();
    DFS(root, 0, 0);
    ArrayList<Integer> bottomView = new ArrayList<>();
    for(Integer key: vertical.keySet()){
        bottomView.add(vertical.get(key).data);
    }
    return bottomView;
}
```

O(N + ClogC)

Bottom View Java Using DFS

If there are **multiple** bottom-most nodes for a horizontal distance from root, then print the later one in level traversal. For example, in the below diagram, 3 and 4 are both the bottommost nodes at horizontal distance 0, we need to print 4.



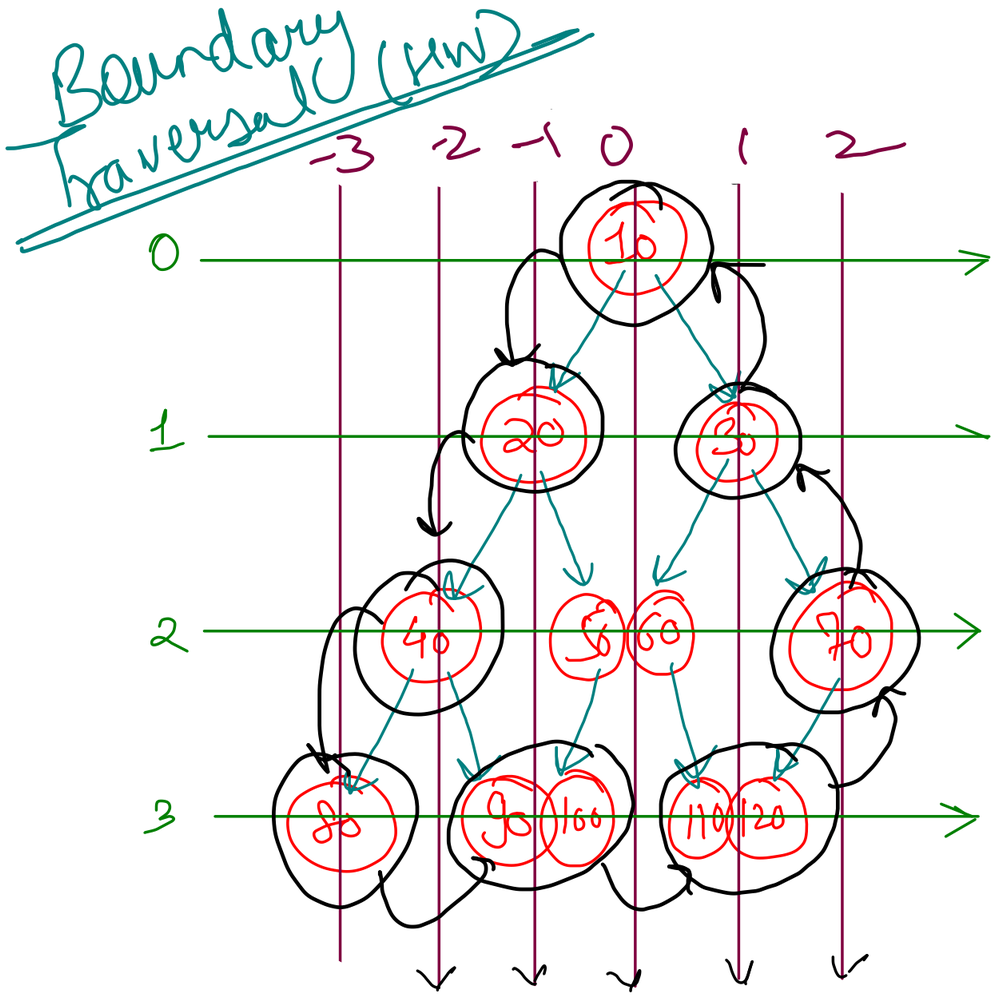
```cpp
void topView(struct Node *root)
{
    if(root == NULL) return;

    map<int,int> m;
    queue<pair<struct Node*,int> > que;
    que.push(make_pair(root,0));
    while(!que.empty())
    {
        pair<struct Node*,int> cur=que.front();
        que.pop();
        if(m.find(cur.second)==m.end())
         m[cur.second]=cur.first->data;
        if(cur.first->left)
        que.push(make_pair(cur.first->left,cur.second-1));
        if(cur.first->right)
        que.push(make_pair(cur.first->right,cur.second+1));

    }

    map<int,int>:: iterator itr;
    for(itr = m.begin();itr!=m.end();itr++)
     cout<<itr->second<<" ";
}
```

Top View (C++)
using level order (BFS)

```cpp
if(root == NULL) return;

map<int,int> m;
queue<pair<struct Node*,int> > que;
que.push(make_pair(root,0));
while(!que.empty())
{
    pair<struct Node*,int> cur=que.front();
    que.pop();
    m[cur.second]=cur.first->data;
    if(cur.first->left)
    que.push(make_pair(cur.first->left,cur.second-1));
    if(cur.first->right)
    que.push(make_pair(cur.first->right,cur.second+1));

}

map<int,int>:: iterator itr;
for(itr = m.begin();itr!=m.end();itr++)
 cout<<itr->second<<" ";
```

Bottom View (C++)
using level order (BFS)

Boundary Traversal (HW)



Left view →

10, 20, 40, 80

Right View →

10, 30, 70, 120

1. **Left boundary nodes:** defined as the path from the root to the left-most node ie- the leaf node you could reach when you always travel preferring the left subtree over the right subtree.
2. **Leaf nodes:** All the leaf nodes except for the ones that are part of left or right boundary.
3. **Reverse right boundary nodes:** defined as the path from the right-most node to the root. The right-most node is the leaf node you could reach when you always travel preferring the right subtree over the left subtree. Exclude the root from this as it was already included in the traversal of left boundary nodes.

Exclude Repetition

Boundary Traversal :→

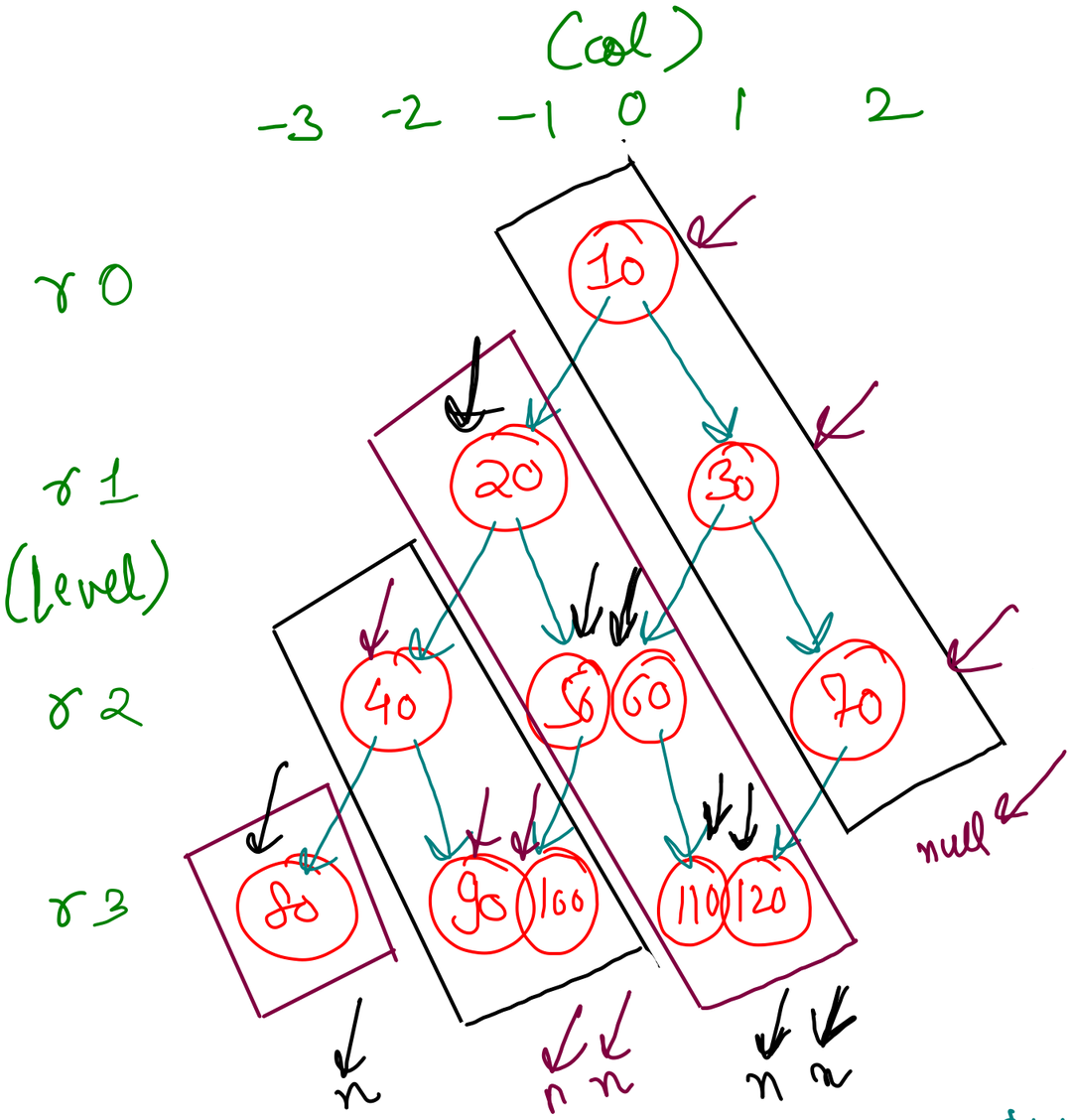
10, 20, 40, 80, 90, 100,

110, 120, 70, 30, 10

left view

leaf nodes

Right View (in Reverse order)



push left child & go on right child

if right child does not exist, then pop from Queue

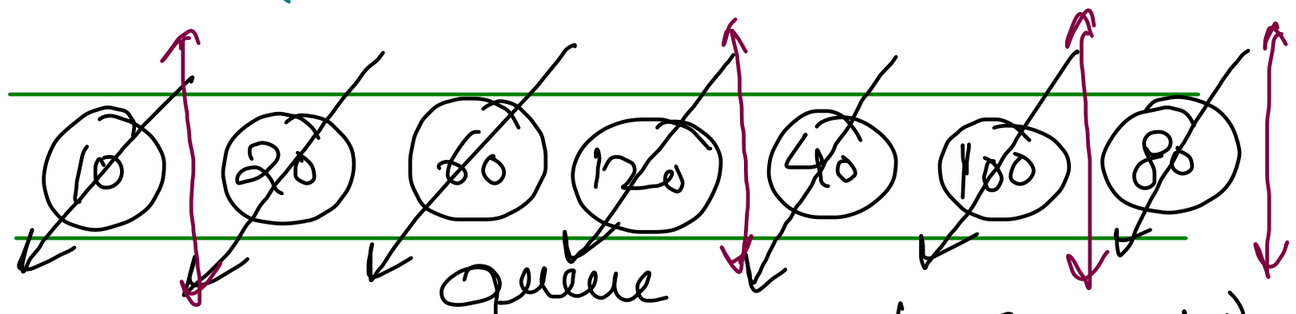
Diagonal traversal

d1 → 10, 30, 70

d2 → 20, 50, 60, 110, 120

d3 → 40, 90, 100

d4 → 80



" Traverse on current diagonal & store the next diagonal "

```java
public ArrayList<Integer> diagonal(Node root)
 {
    Queue<Node> q = new ArrayDeque<>();
    q.add(root);
    ArrayList<Integer> res = new ArrayList<>();

    Node curr = null;
    while(curr != null || q.size() > 0){
        if(curr == null) {
            curr = q.remove();
            continue;
        }

        res.add(curr.data);
        if(curr.left != null) q.add(curr.left);
        curr = curr.right;
    }

    return res;
}
```

O(N) Time

O(N) Extra Space

222 # Count Complete Tree Nodes

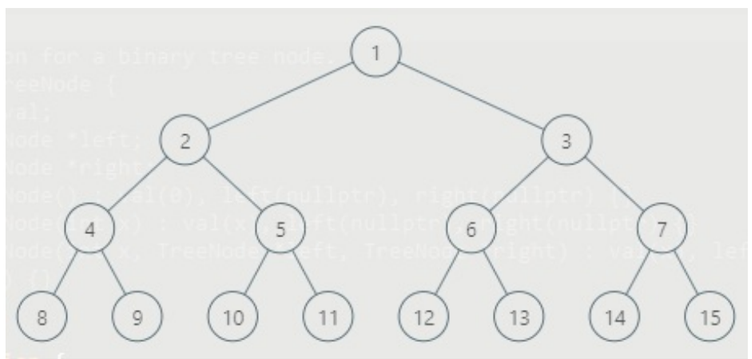
Tree with (h-1) levels are complete, and the last level may or may not have all nodes. But they will be as left as possible.

A tree which is complete will always be balanced.

N nodes ⇒ $\log_2 N$ height.

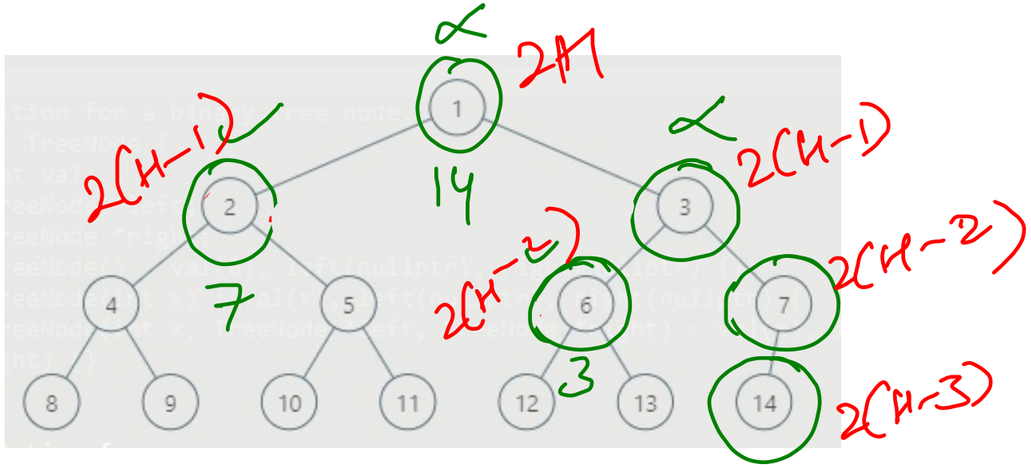
If leftmost node depth = rightmost node depth = H
⇒ complete tree will be perfect.

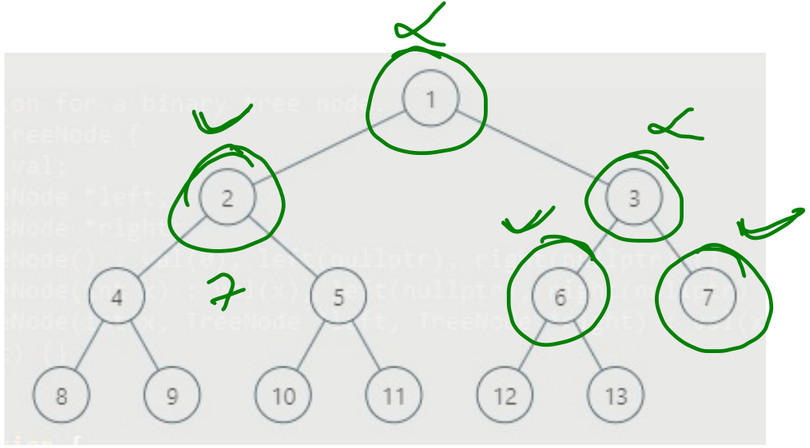
⇒ No of node = $2^H - 1$

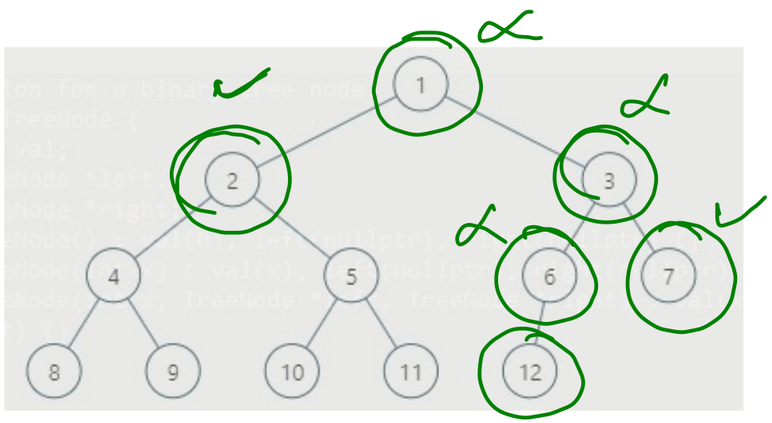


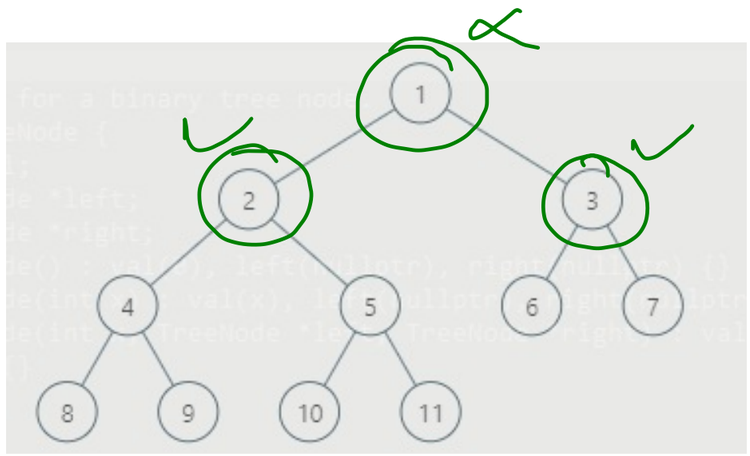
perfect tree height(H) ⇒ Node ⇒ $2^H - 1$

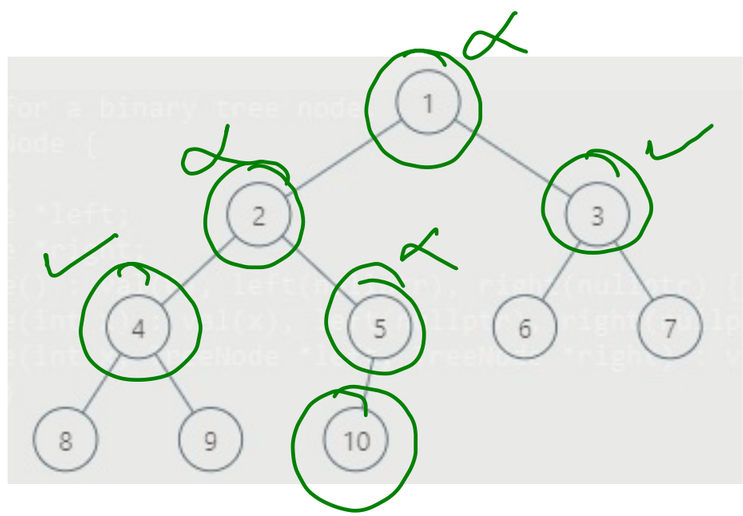
If root is not perfect, then either
left child is perfect or
right child is perfect.

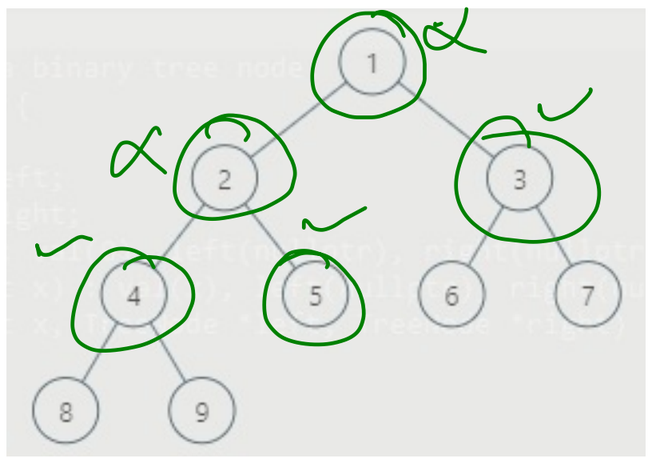

1
2
3
4
5
6
7
8
9
10
11
12
13
14

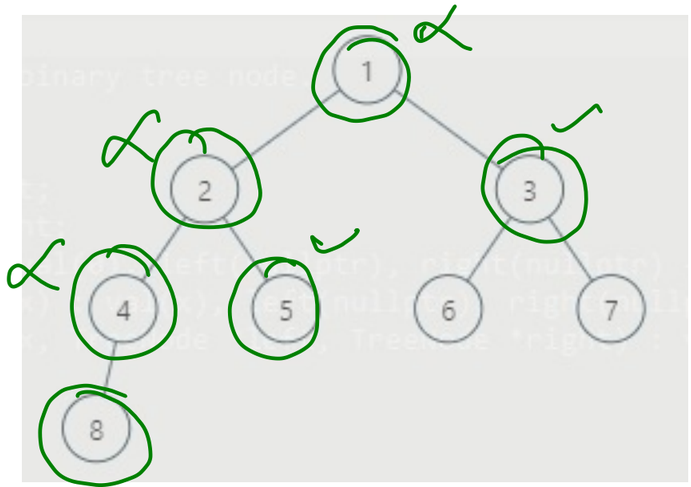

1
2
3
4
5
6
7
8
9
10
11
12
13

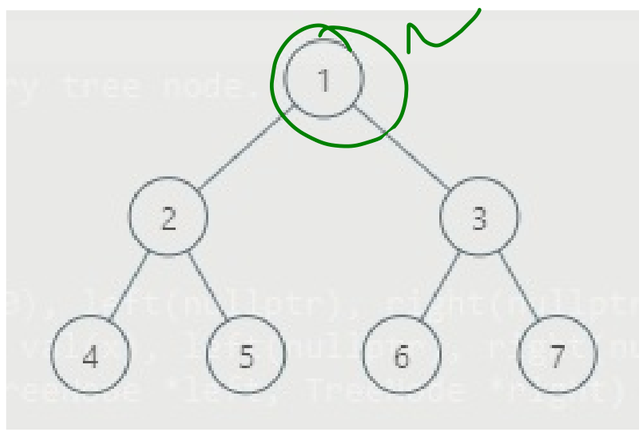

1
2
3
4
5
6
7
8
9
10
11
12


1
2
3
4
5
6
7
8
9
10
11


1
2
3
4
5
6
7
8
9
10


1
2
3
4
5
6
7
8
9


1
2
3
4
5
6
7
8


1
2
3
4
5
6
7

```java
public int leftMostdepth(TreeNode root){
    int ans = 0;
    while(root != null){
        ans++;
        root = root.left;
    }
    return ans;
}
public int rightMostdepth(TreeNode root){
    int ans = 0;
    while(root != null){
        ans++;
        root = root.right;
    }
    return ans;
}

public int countNodes(TreeNode root) {
    if(root == null) return 0;

    // Is Tree Perfect
    int left = leftMostdepth(root); // O(H)
    int right = rightMostdepth(root); // O(H)

    // No of nodes in perfect tree of height h is 2 ^ h - 1
    if(left == right) return ((1 << left) - 1);
    return 1 + countNodes(root.left) + countNodes(root.right);
}
```

O(H)

O(H)

Total Time Complexity in worst case

$= 2H + 4*(H-1) + 4*(H-2) + \cdots + 4*1$

$= k\{ H + (H-1) + (H-2) + \cdots + 1 \}$

$= k * \frac{H*(H+1)}{2} = k * \frac{\log n * (\log n + 1)}{2}$

$\Rightarrow \boxed{O(\log^2 n)}$

Construct Tree from Traversals

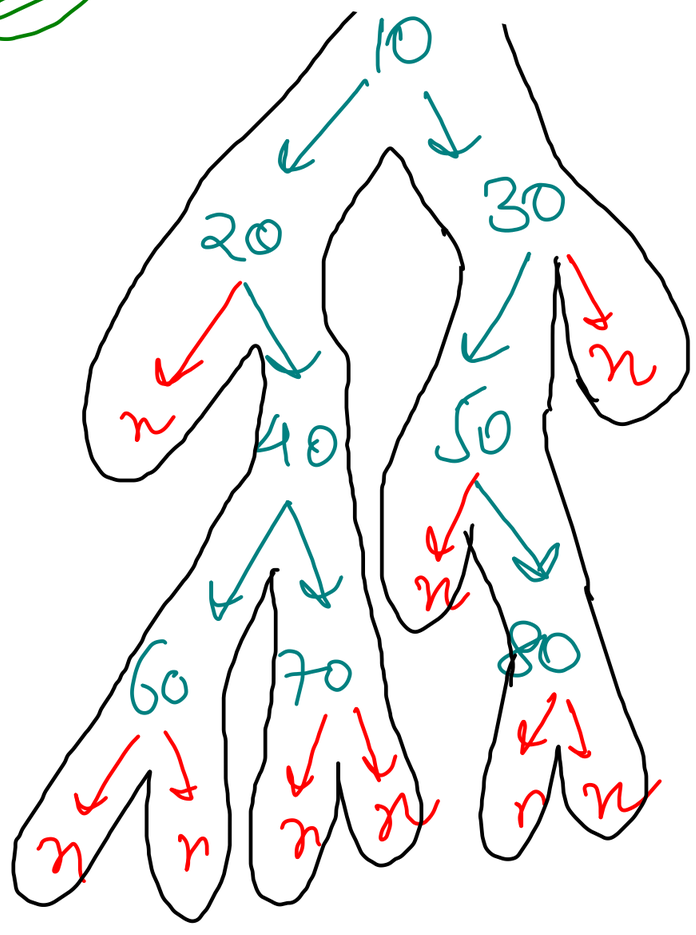
Starting in 10 mins (3:15)

- ~~→~~ from Inorder & → Preorder (105) / Postorder (106)
- → from Preorder & Postorder (889)
- → from Inorder & levelorder (GFG)
- ~~✓~~ Serialize & Deserialize → Binary Tree (297) / Generic Tree (NADOS)
- → Construct String from Tree (606)
- Construct Tree from String (Lintcode 880)
- → Verify Preorder Serialization (331)

297 Serialize & De-serialize Binary tree

Construction of Binary tree

Preorder serialization



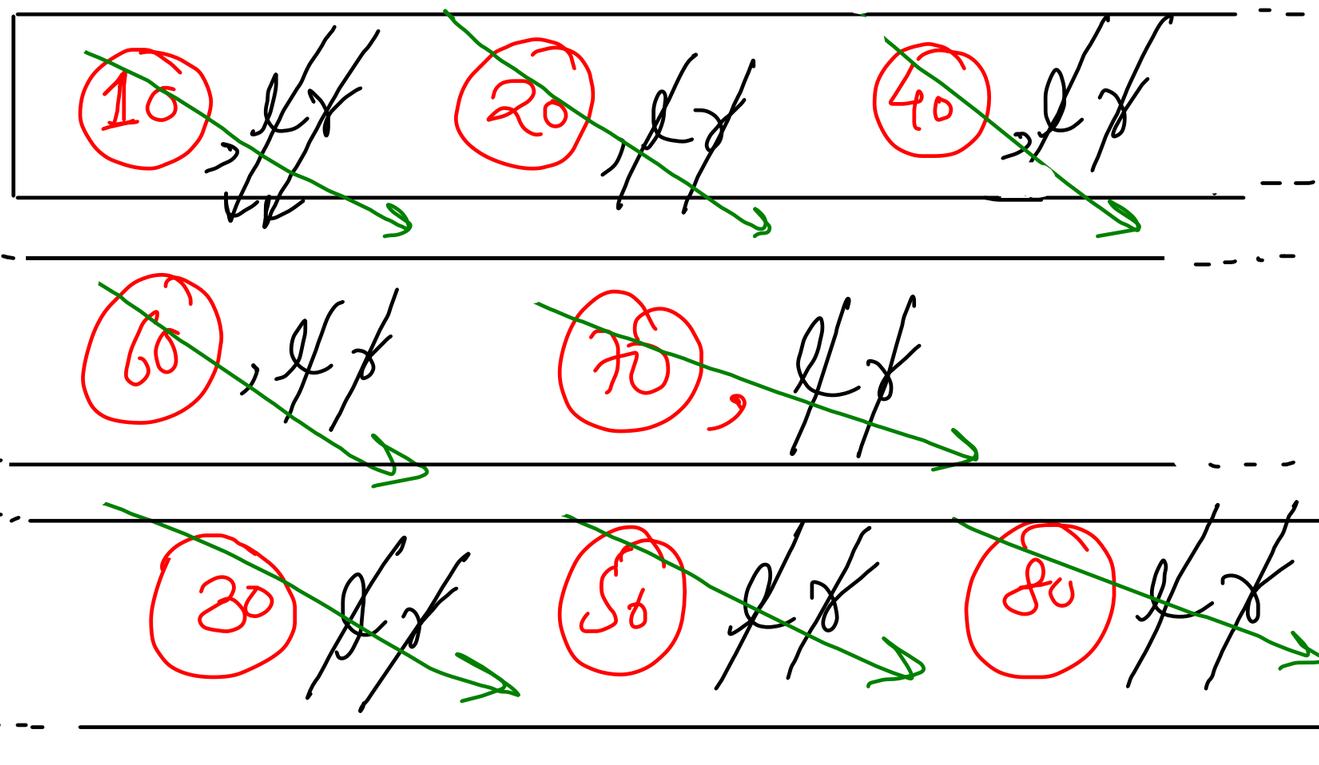
"10, 20, null, 40, 60, null, null,
70, null, null, 30, 50, null,
80, null, null, null, "

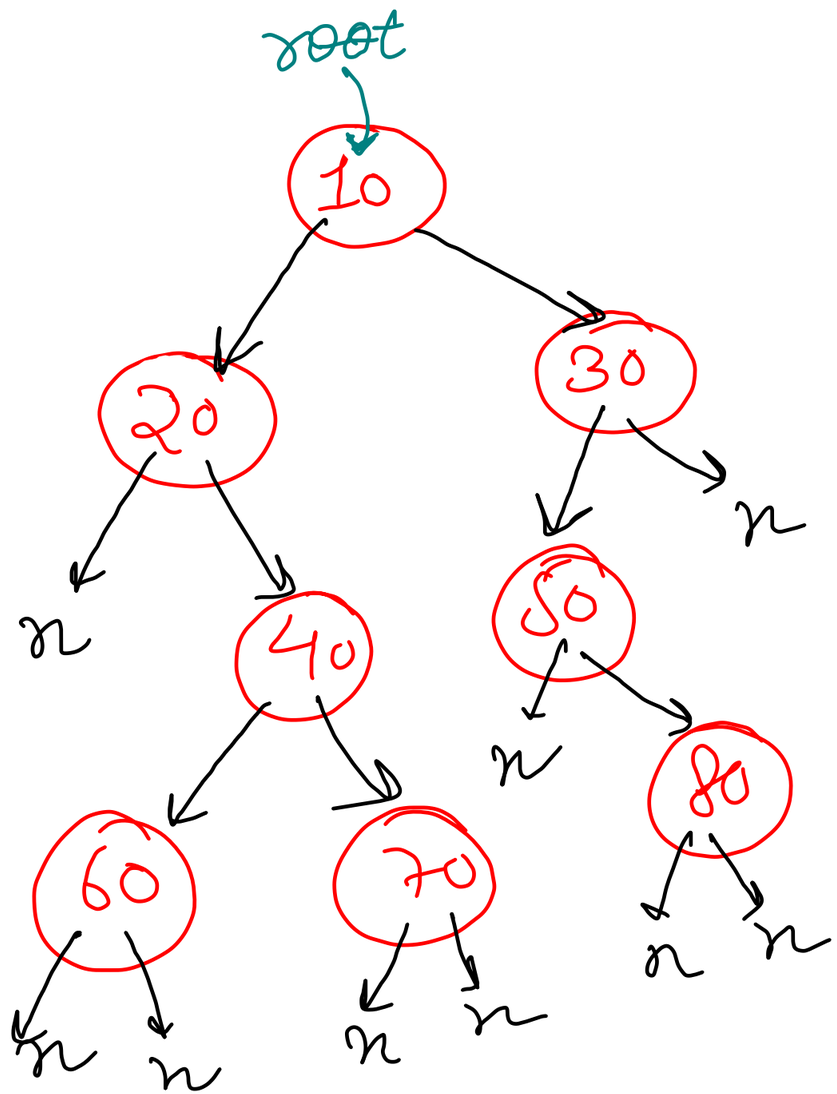
```
// Encodes a tree to a single string.
public String serialize(TreeNode root) {
    if(root == null) return "null,";
    return root.val + "," + serialize(root.left) + serialize(root.right);
}
```

leetcode serializes binary tree using level order

String[] tokens = Str.split(",");

"10, 20, null, 40, 60, null, null,
70, null, null, 30, 50, null,
80, null, null, null, "



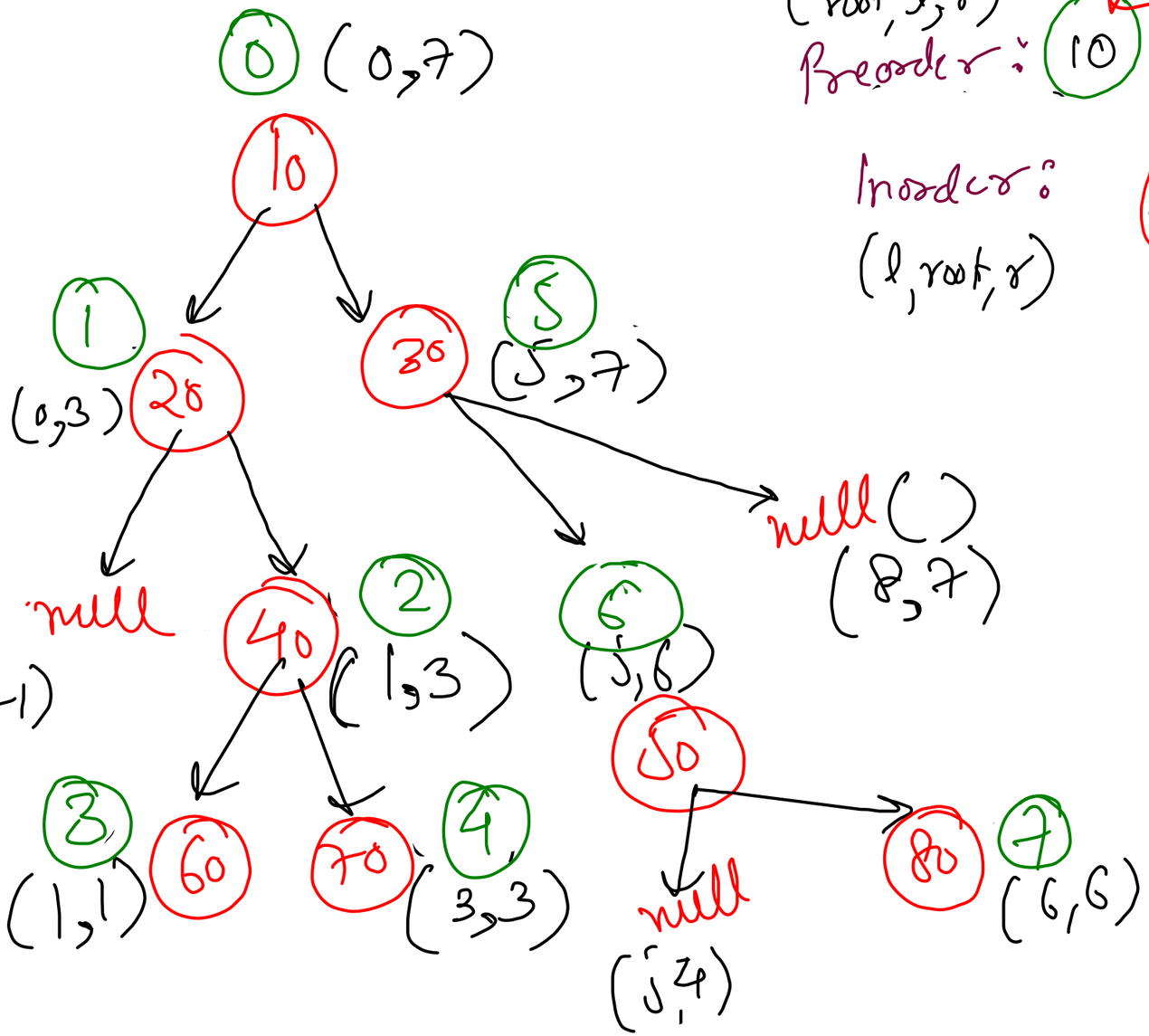


```java
public TreeNode deserialize(String data) {
    if(data.equals("null,") == true) return null;

    Stack<Pair> stk = new Stack<>();
    String[] tokens = data.split(",");
    TreeNode root = null;

    for(String token: tokens){
        if(token.equals("null") == true){
            if(stk.peek().state == 'l')
                stk.peek().state = 'r';
            else stk.pop();

        } else {
            TreeNode child = new TreeNode(Integer.parseInt(token));
            if(stk.empty()){
                root = child;
            } else {
                if(stk.peek().state == 'l'){
                    stk.peek().node.left = child;
                    stk.peek().state = 'r';
                } else {
                    stk.peek().node.right = child;
                    stk.pop();
                }
            }
            stk.push(new Pair(child));
        }
    }

    return root;
}
```

Iterative DFS

↳ Construction from String

105 Construct Binary tree from Preorder & Inorder



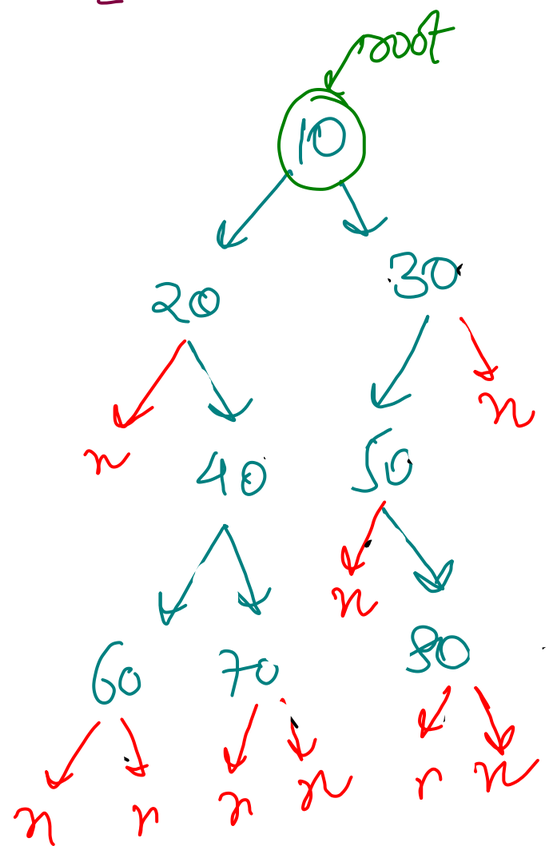
(root, l, r)

Preorder: 10 20 40 60 70 30 50 80

Inorder: (l, root, r)

20 60 40 70 10 50 80 30

0 1 2 3 4 5 6 7



```java
public TreeNode buildTree(int[] preorder, HashMap<Integer, Integer> inorder, int inL, int inR, int preIdx){
    if(inL > inR) return null;

    TreeNode curr = new TreeNode(preorder[preIdx]);
    int inIdx = inorder.get(preorder[preIdx]);

    curr.left = buildTree(preorder, inorder, inL, inIdx - 1, preIdx + 1);
    curr.right = buildTree(preorder, inorder, inIdx + 1, inR, preIdx + 1 + (inIdx - inL));
    return curr;
}

public TreeNode buildTree(int[] preorder, int[] inorder) {
    HashMap<Integer, Integer> hm = new HashMap<>();
    for(int i=0; i<inorder.length; i++) hm.put(inorder[i], i);

    return buildTree(preorder, hm, 0, inorder.length - 1, 0);
}
```

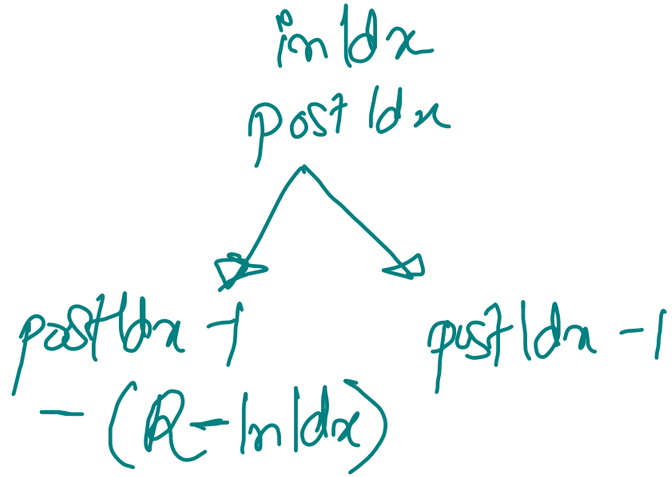
O(N) extra space

O(N) DFS

Initial preprocessing

106

```java
class Solution {
    public TreeNode buildTree(int[] postorder, HashMap<Integer, Integer> inorder, int inL, int inR, int postIdx){
        if(inL > inR) return null;

        TreeNode curr = new TreeNode(postorder[postIdx]);
        int inIdx = inorder.get(postorder[postIdx]);

        curr.left = buildTree(postorder, inorder, inL, inIdx - 1, postIdx - 1 - (inR - inIdx));
        curr.right = buildTree(postorder, inorder, inIdx + 1, inR, postIdx - 1);
        return curr;
    }

    public TreeNode buildTree(int[] inorder, int[] postorder) {
        HashMap<Integer, Integer> hm = new HashMap<>();
        for(int i=0; i<inorder.length; i++) hm.put(inorder[i], i);

        return buildTree(postorder, hm, 0, inorder.length - 1, postorder.length - 1);
    }
}
```



889 Construct Tree from Preorder & Postorder

struct
↳ 0/2 childs

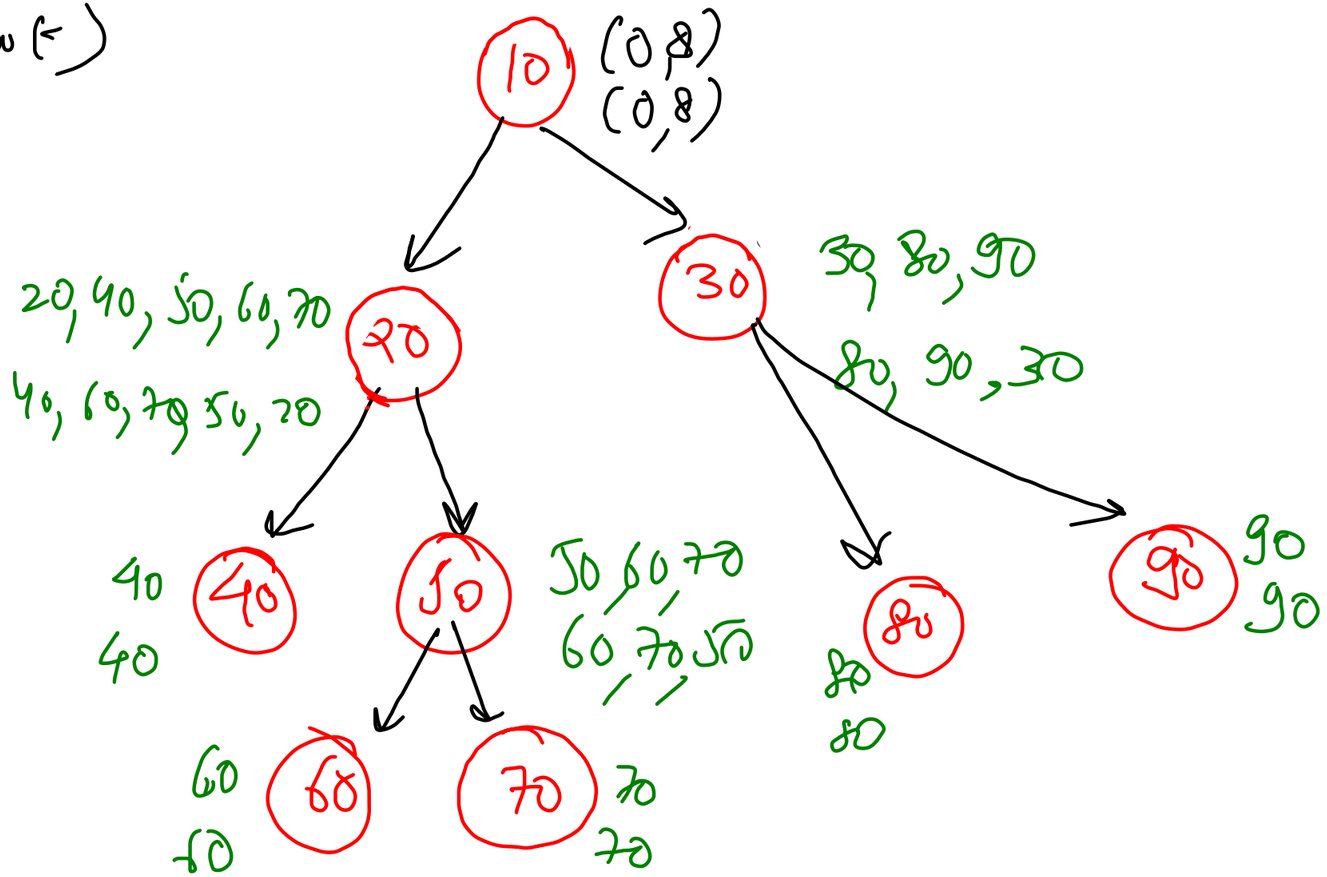
(root, l, r)

Preorder : 10, 20, 40, 50, 60, 70, 30, 80, 90

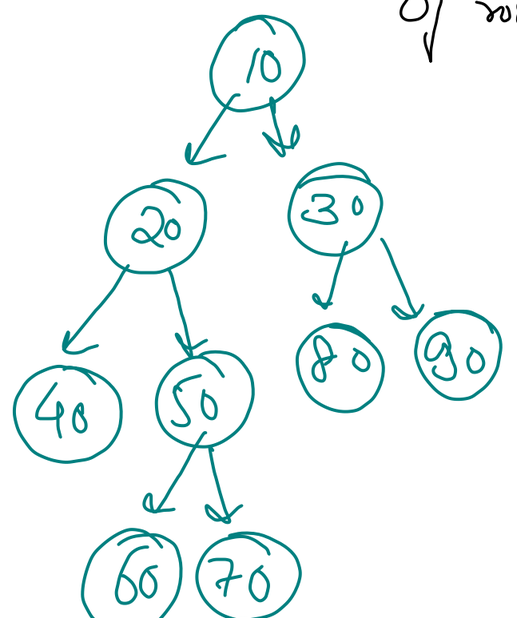
Postorder : 40, 60, 70, 50, 20, 80, 90, 30, 10

(l, r, root)

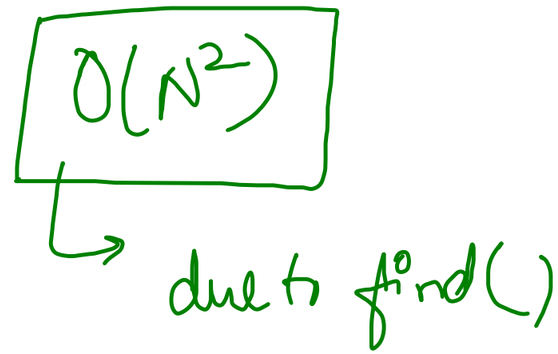
Preorder 1st node
= Postorder last node
= root node



left subtree
↳ root node → next node
↳ root node



```java
public int find(int val, int[] postorder){
    for(int i=0; i<postorder.length; i++){
        if(postorder[i] == val) return i;
    }
    return -1;
}

public TreeNode construct(int[] preorder, int[] postorder, int prel, int prer, int postl, int postr){
    if(prel > prer) return null;
    if(prel == prer) return new TreeNode(preorder[prel]);

    TreeNode root = new TreeNode(preorder[prel]);

    if(prel + 1 >= preorder.length) return root;

    int leftChild = preorder[prel + 1];
    int postIdx = find(leftChild, postorder);
    int leftCount = postIdx - postl + 1;
    root.left = construct(preorder, postorder, prel + 1, prel + leftCount, postl, postIdx);
    root.right = construct(preorder, postorder,  prel + leftCount + 1, prer, postIdx + 1, postr - 1);

    return root;
}

public TreeNode constructFromPrePost(int[] preorder, int[] postorder) {
    return construct(preorder, postorder, 0, preorder.length - 1, 0, postorder.length - 1);
}
```



(root, l, r)

0 1 2 3 4 5 6 7 8

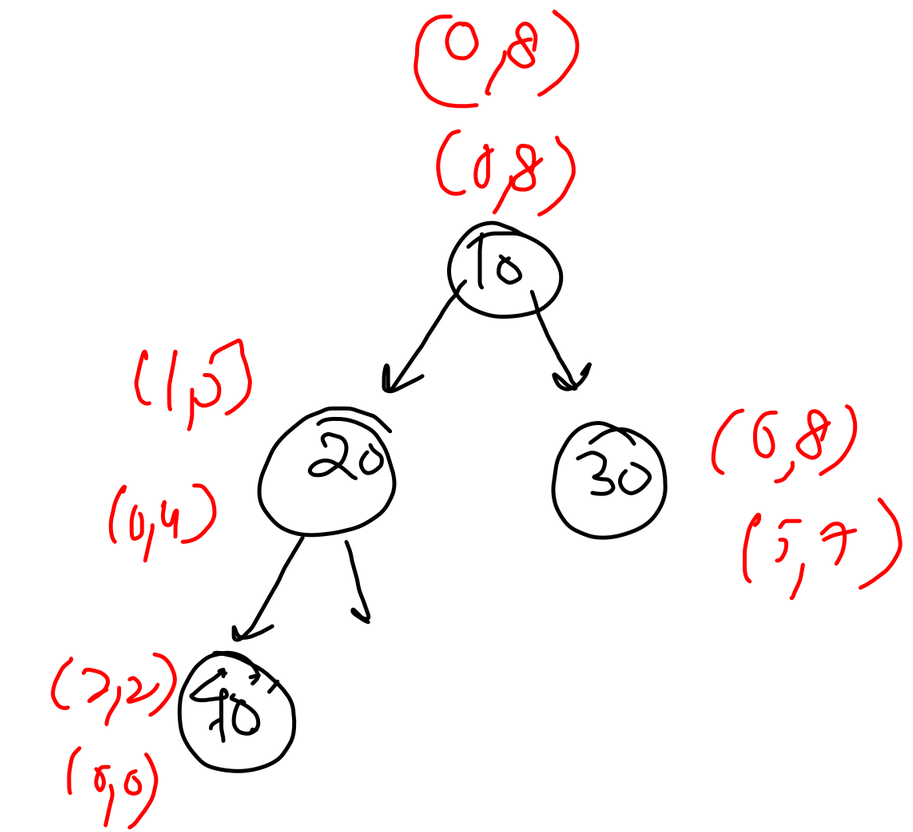
Preorder: 10, 20, 40, 50, 60, 70, 30, 80, 90

Postorder: 40, 60, 70, 50, 20, 80, 90, 30, 10

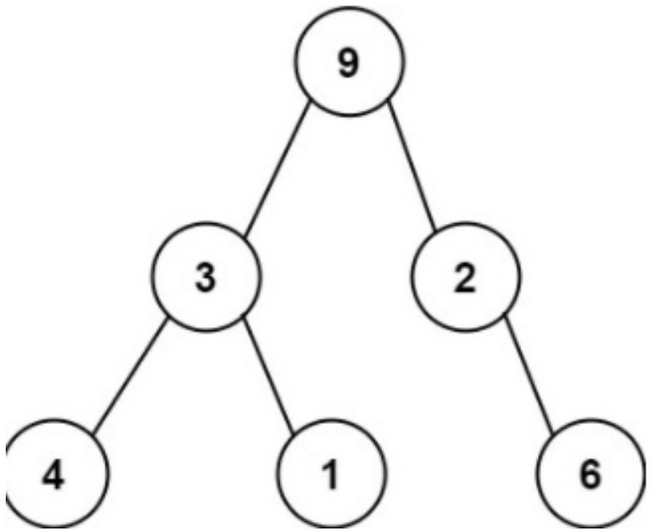
0 1 2 3 4 5 6 7 8

(l, r, root)

```java
public int find(int val, int[] postorder){
    for(int i=0; i<postorder.length; i++){
        if(postorder[i] == val) return i;
    }
    return -1;
}

public TreeNode construct(int[] preorder, int[] postorder, int prel, int prer, int postl, int postr){
    if(prel > prer) return null;
    if(prel == prer) return new TreeNode(preorder[prel]);

    TreeNode root = new TreeNode(preorder[prel]);

    if(prel + 1 >= preorder.length) return root;

    int leftChild = preorder[prel + 1];
    int postIdx = find(leftChild, postorder);
    int leftCount = postIdx - postl + 1;
    root.left = construct(preorder, postorder, prel + 1, prel + leftCount, postl, postIdx);
    root.right = construct(preorder, postorder,  prel + leftCount + 1, prer, postIdx + 1, postr - 1);

    return root;
}

public TreeNode constructFromPrePost(int[] preorder, int[] postorder) {
    return construct(preorder, postorder, 0, preorder.length - 1, 0, postorder.length - 1);
}
```



Verify Preorder serialization (331)



Preorder ←

eg 1

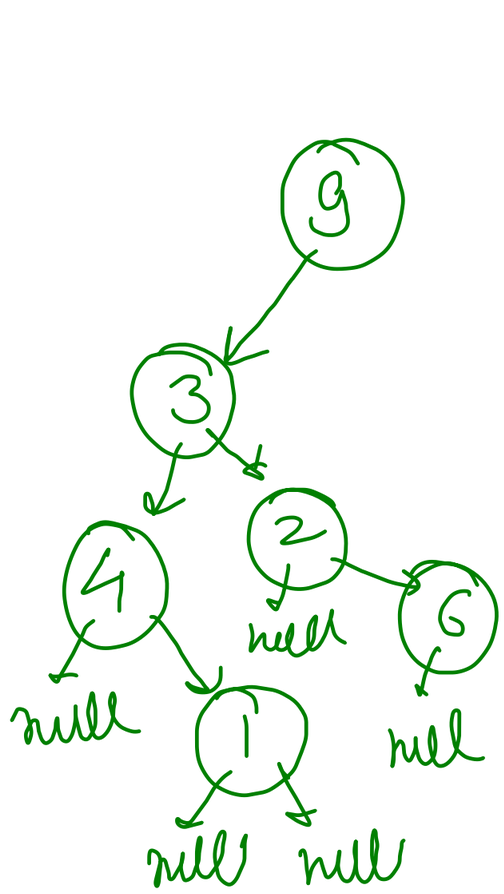
9, 3, 4, #, #, 1, #, #, 2, #, 6, #, #

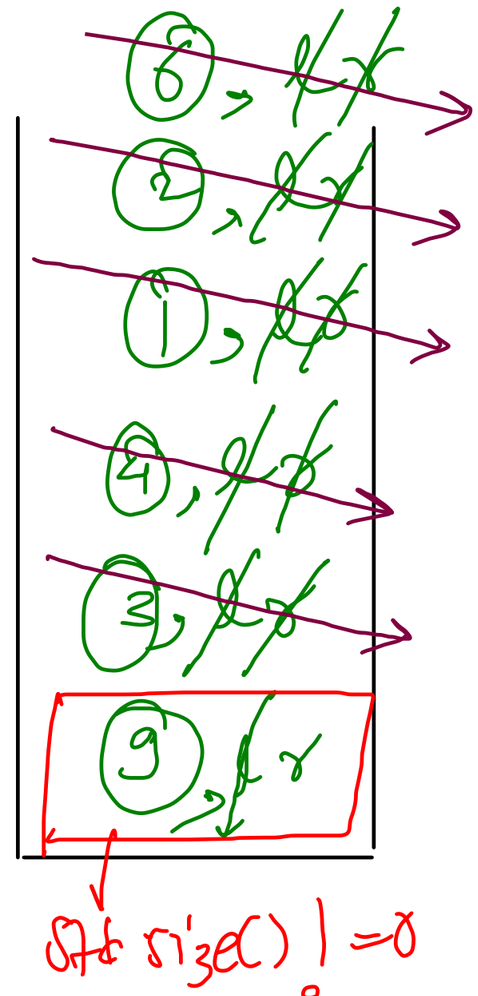
true

eg 2

9, 3, 4, #, #, 1, #, #, 2, #, 6, #, #

false

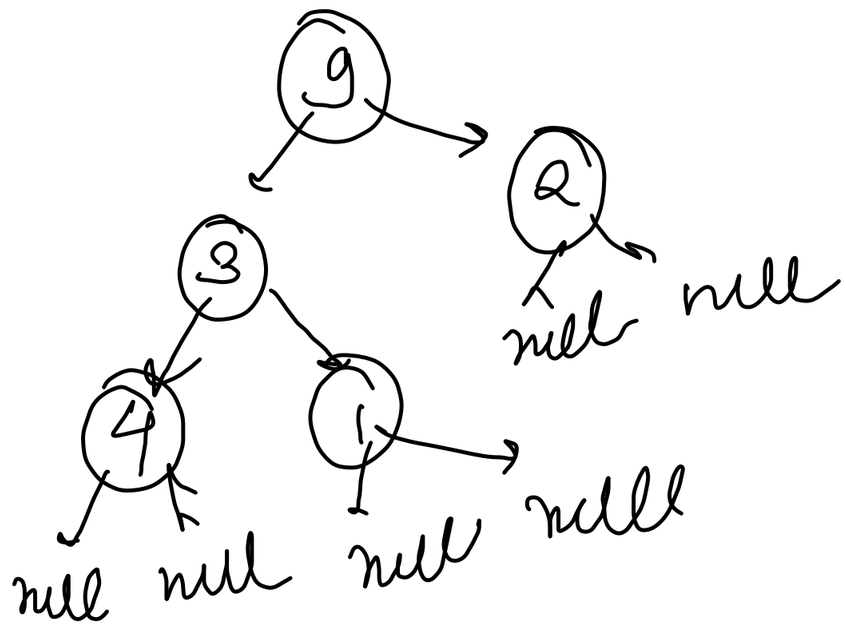




eg: 9, 3, 4, #, #, 1, #, #, 2, #, #, 6, #, #

Stk.empty()

⇒ it is not root node



⇒ Preorder Serialization is wrong

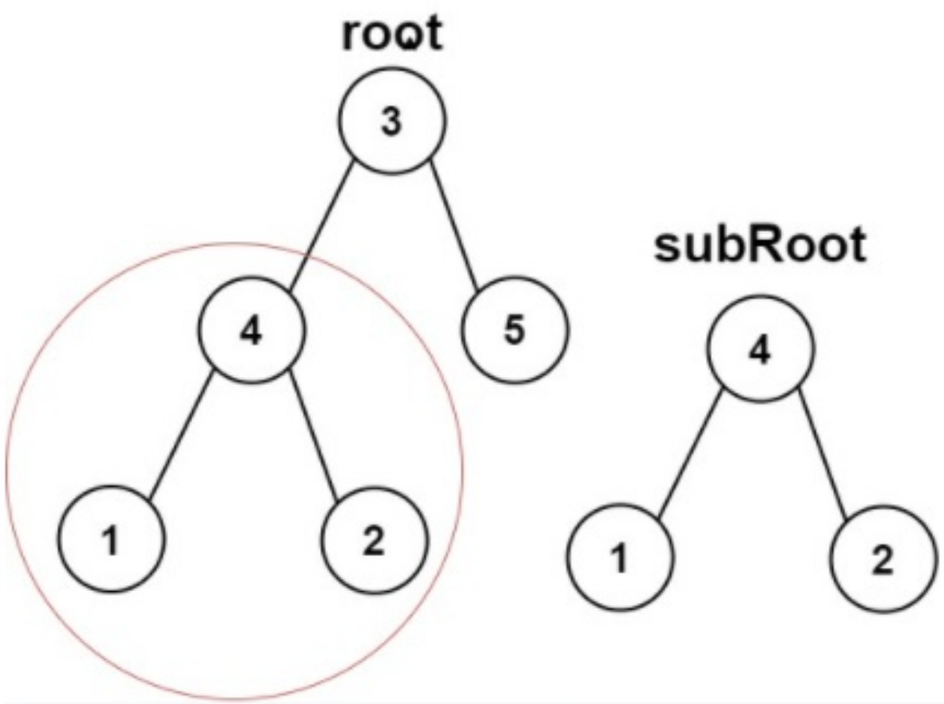
```java
if(preorder.equals("#") == true)
    return true;

Stack<Character> stk = new Stack<>();
String[] tokens = preorder.split(",");
boolean root = false;
```

O(N) time

O(N) space (stack)

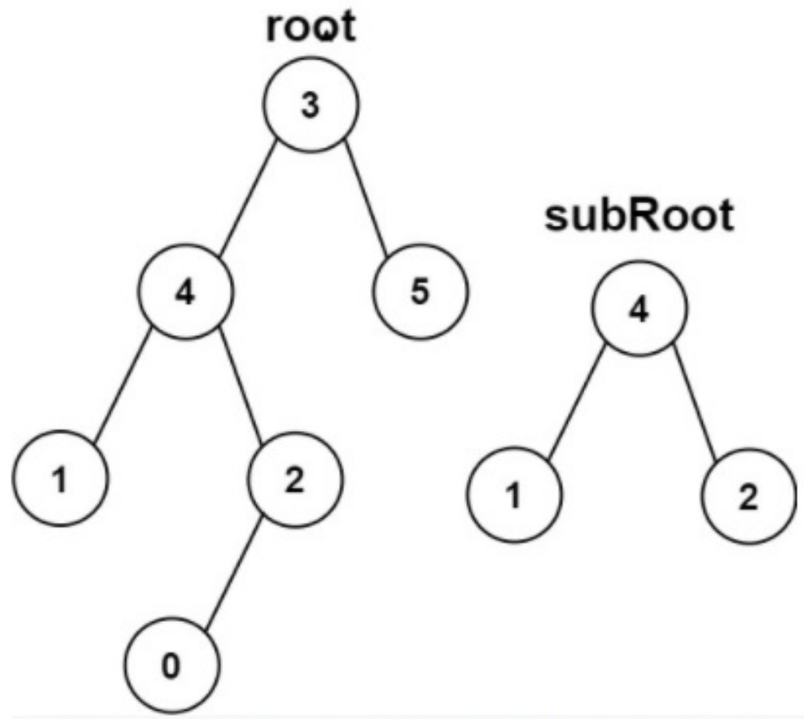
```java
for(String token: tokens){
    if(token.equals("#") == true){
        if(stk.size() == 0) return false;
        else if(stk.peek() == 'l') {
            stk.pop();
            stk.push('r');
        } else {
            stk.pop();
        }

    } else {
        if(stk.size() == 0){
            if(root == false){
                root = true;
            } else return false;
        }
        else {
            if(stk.peek() == 'l'){
                stk.pop();
                stk.push('r');
            } else {
                stk.pop();
            }
        }

        stk.push('l');
    }
}

if(stk.size() == 0) return true;
return false;
```

Example 1:




```
Input: root = [3,4,5,1,2], subRoot = [4,1,2]
Output: true
```

Example 2:




```
Input: root = [3,4,5,1,2,null,null,null,null,0], subRoot = [4,1,2]
Output: false
```

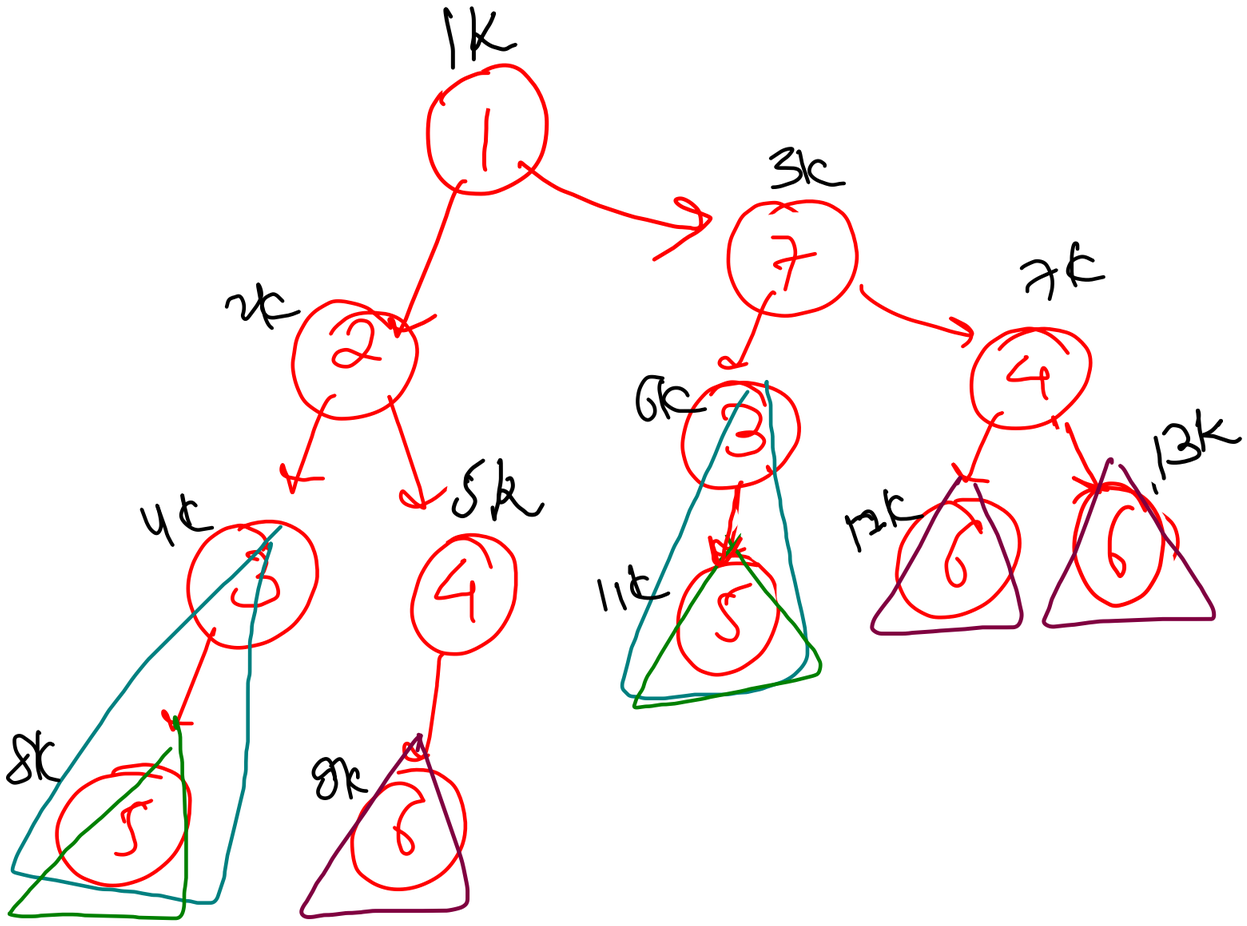
$O(N^2)$

Brute force

Subtree of Another Tree → 572

```java
public boolean equals(TreeNode root, TreeNode subRoot){
    if(root == null && subRoot == null) return true;
    if(root == null || subRoot == null) return false;
    if(root.val != subRoot.val) return false;
    return equals(root.left, subRoot.left) && equals(root.right, subRoot.right);
}

public boolean isSubtree(TreeNode root, TreeNode subRoot) {
    if(root == null && subRoot == null) return true;
    if(root == null || subRoot == null) return false;
    return equals(root, subRoot) || isSubtree(root.left, subRoot) || isSubtree(root.right, subRoot);
}
```

Duplicate Subtrees 652



String vs Int ?

5 # # → 1/2

3 5 # # # → 1/2

6 # # → 1/2/3

4 6 # # # → 1

2 3 5 # # # 4 6 # # # → 1

4 6 # # 6 # # → 1

Ans:→ {11K, 6K, 12K}

```java
List<TreeNode> res;
HashMap<String, Integer> duplicates;

public String DFS(TreeNode root){
    if(root == null) return "#";

    String hash = root.val + "," + DFS(root.left) + "," + DFS(root.right);
    if(duplicates.containsKey(hash) == true && duplicates.get(hash) == 1){
        res.add(root);
    }

    duplicates.put(hash, duplicates.getOrDefault(hash, 0) + 1);
    return hash;
}

public List<TreeNode> findDuplicateSubtrees(TreeNode root) {
    res = new ArrayList<>();
    duplicates = new HashMap<>();
    DFS(root);
    return res;
}
```

$O(N^2)$

for each node,

we are finding the preorder serialization