Binary Trees & Binary Search Trees

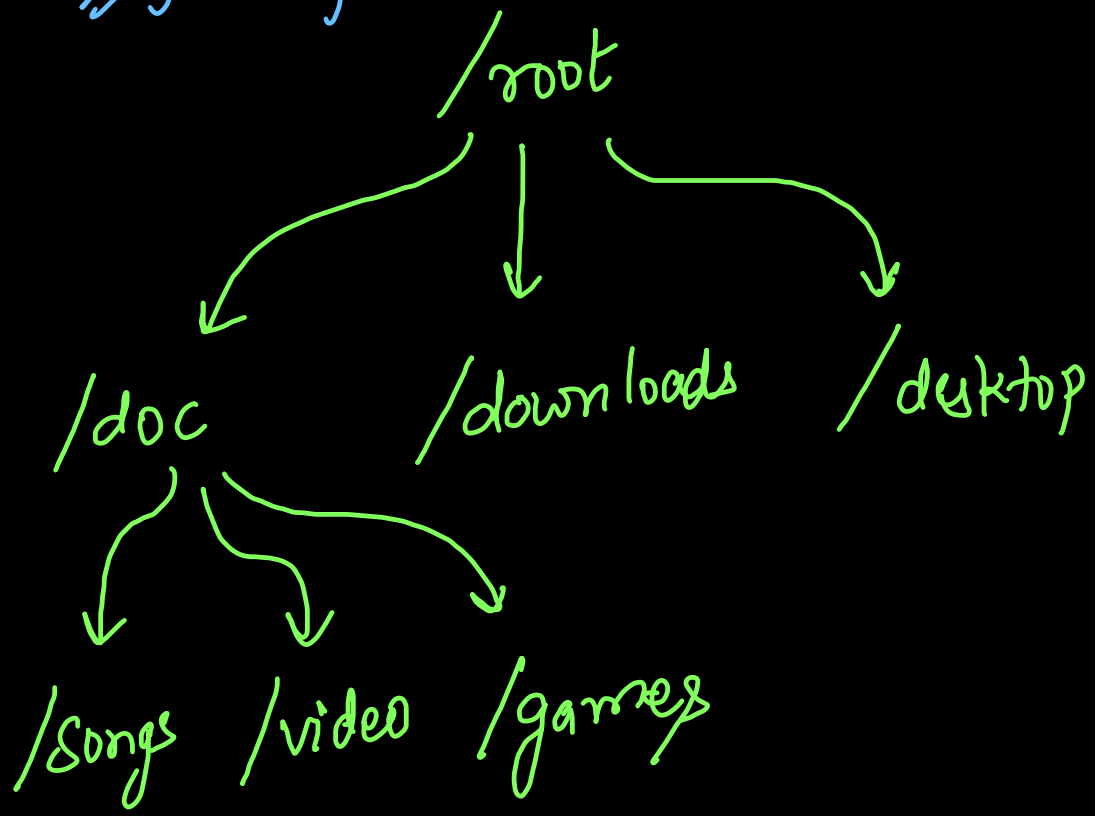
Data Structures

- → Array, String → Static
- → Arraylist, StringBuilder → dynamic
- → LinkedList → dynamic & non-contiguous
- → Stack, Queue, Deque
  - Stack ↓ LIFO
  - Queue ↓ FIFO
  - Deque ↓ addFirst/Last, removeFirst/Last

} Linear Data Structures

Real-life examples of Heirarchial Data Structures

eg file system

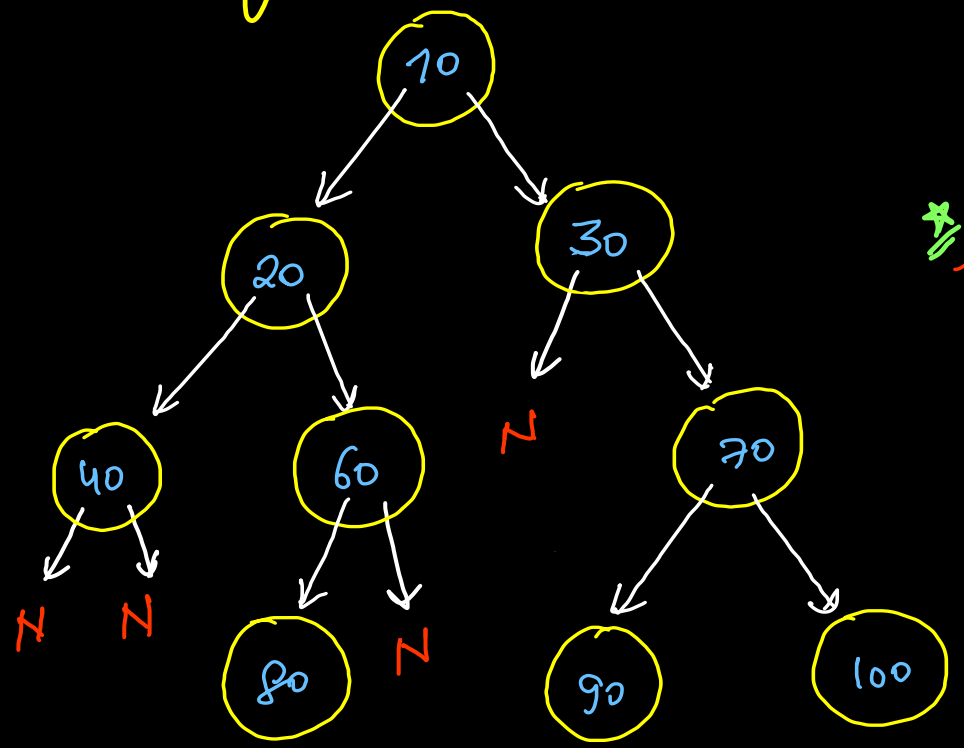


eg github repository management

eg flutter widget tree

eg organizaton hierarchy

Binary Tree



root node : (10)
- no parent of root node
- entire tree can be traversed

left child & right child

parent
- every node (except root) will have exactly one parent.

ancestors
- all nodes from current node till root node

siblings
- nodes with same parent

cousins
- nodes at same level but not siblings

leaves
- nodes having 0 children (40, 80, 90, 100)

Single child nodes → no siblings (80, 70)

Single child parent → either only left child or only right child exist (30, 70)

descendants / subtree
- all nodes that can be traversed from the current node

binary tree node

class TreeNode {

int val;

TreeNode left, right;

TreeNode() { }

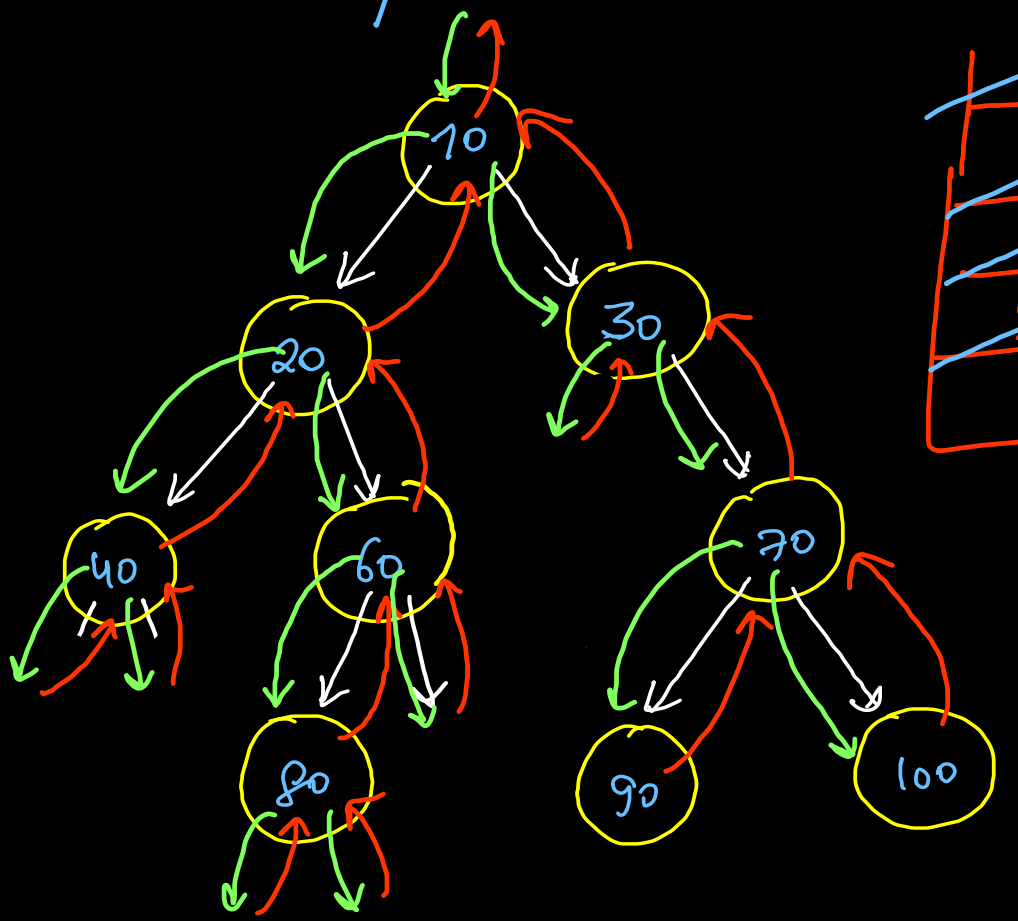
TreeNode(int val) { this.val = val; }

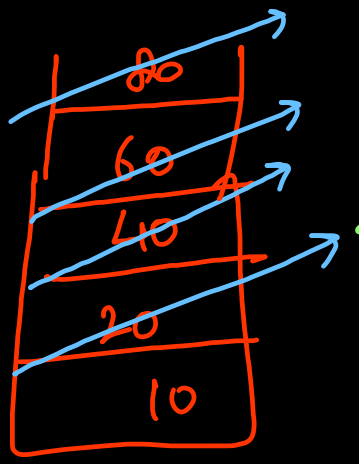
}

objects will be single node

```java
// Definition for a binary tree node.
public class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode() {}
    TreeNode(int val) { this.val = val; }
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}
```

Depth First Search Traversal






```
public void dfs(TreeNode root)
{
0.  if(root == null) return;
1.  preorder.add(root.val);
2.  dfs(root.left);
3.  inorder.add(root.val);
4.  dfs(root.right);
5.  postorder.add(root.val);
}
```

preorder: 10, 20, 40, 60, 80, 30, 70, 90, 100

inorder: 40, 20, 80, 60, 10, 30, 90, 70, 100

postorder: 40, 80, 60, 20, 90, 100, 70, 30, 10,

- 0 → base case
- 0 → meeting expectation (work)
- 1 → expectation
- 2 → faith/recursive call

```java
// Definition for a binary tree node.
public class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode() {}
    TreeNode(int val) { this.val = val; }
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}
```

LC 144

```java
class Solution {
    List<Integer> preorder = new ArrayList<>();

    // Faith: Root -> Left Subtree -> Right Subtree
    public void dfs(TreeNode root){
        if(root == null) return;
        preorder.add(root.val);
        dfs(root.left);
        dfs(root.right);
    }

    public List<Integer> preorderTraversal(TreeNode root) {
        dfs(root);
        return preorder;
    }
}
```

LC 94

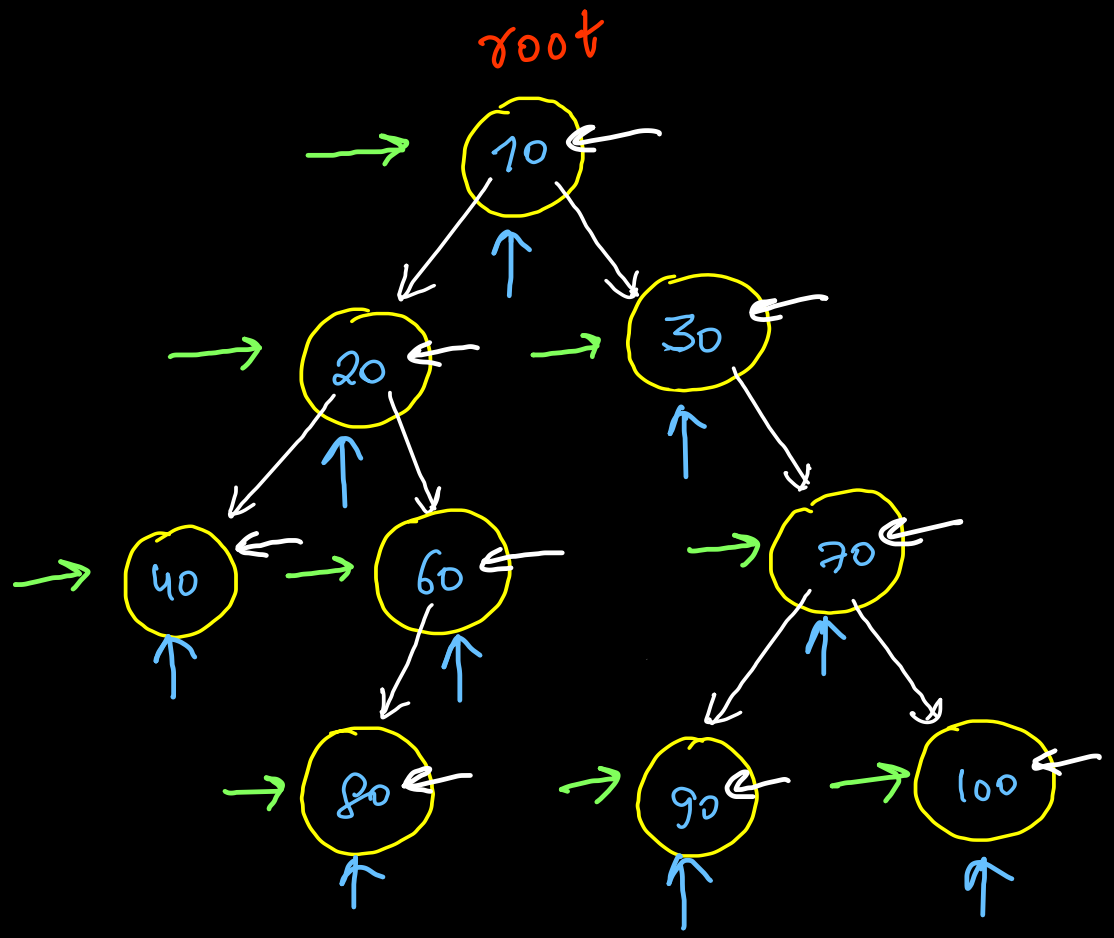
```java
class Solution {
    List<Integer> inorder = new ArrayList<>();

    // Faith: Left Subtree -> Root -> Right Subtree
    public void dfs(TreeNode root){
        if(root == null) return;
        dfs(root.left);
        inorder.add(root.val);
        dfs(root.right);
    }

    public List<Integer> inorderTraversal(TreeNode root) {
        dfs(root);
        return inorder;
    }
}
```

Time complexity ⇒ O(n)
linear

Space Complexity
↳ O(h) or O(d)
recursion call stack

LC 145

```java
class Solution {
    List<Integer> postorder = new ArrayList<>();

    // Faith: Left Subtree -> Right Subtree -> Root
    public void dfs(TreeNode root){
        if(root == null) return;
        dfs(root.left);
        dfs(root.right);
        postorder.add(root.val);
    }

    public List<Integer> postorderTraversal(TreeNode root) {
        dfs(root);
        return postorder;
    }
}
```



Preorder (→)

10, 20, 40, 60, 80, 30, 70, 90, 100

Inorder (↑)

40, 20, 80, 60, 10, 30, 90, 70, 10

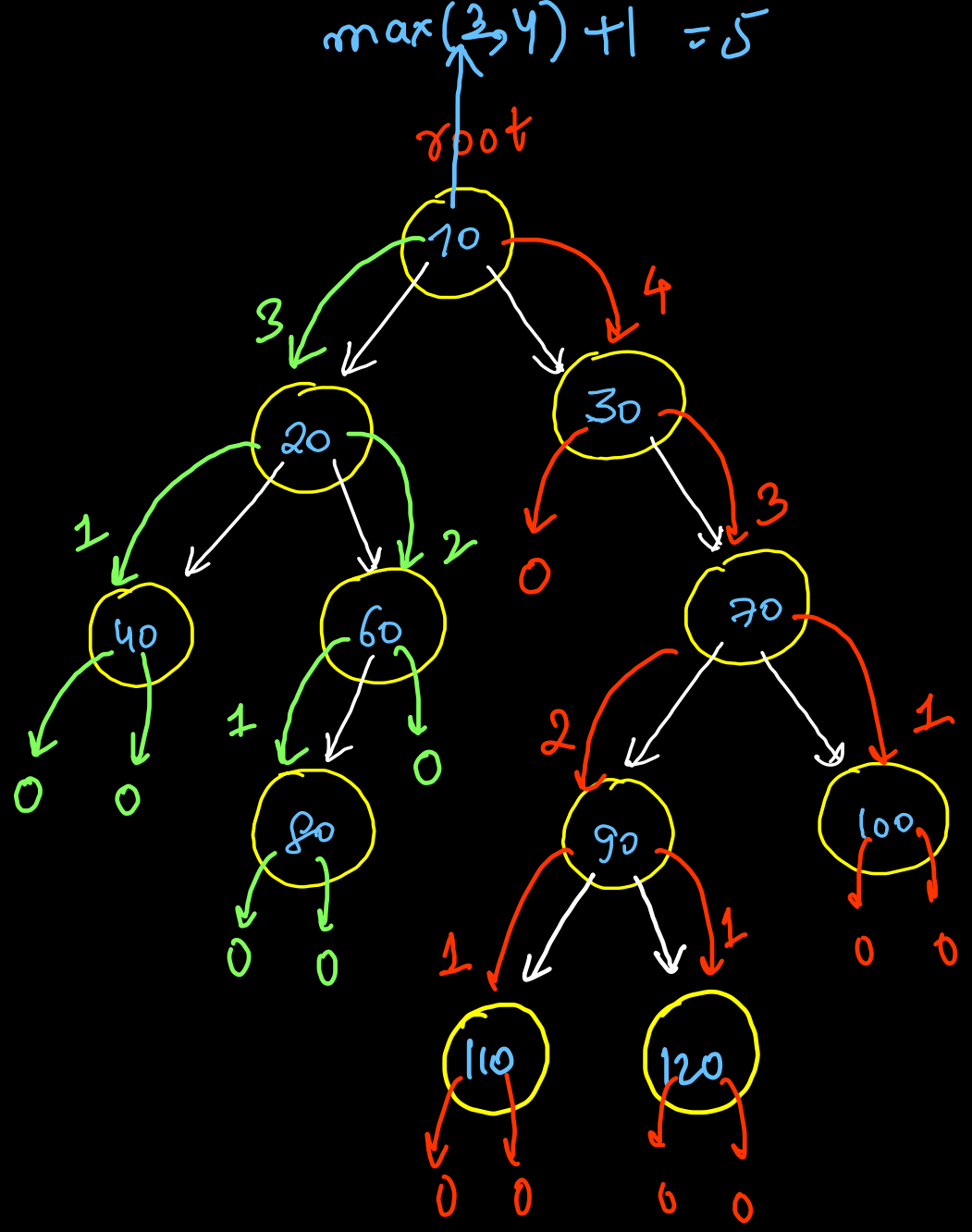
Postorder (←)

40, 80, 60, 20, 90, 100, 70, 30, 10

dfs

left

right



Height / Maxm Depth / Longest path from root to farthest leaf node

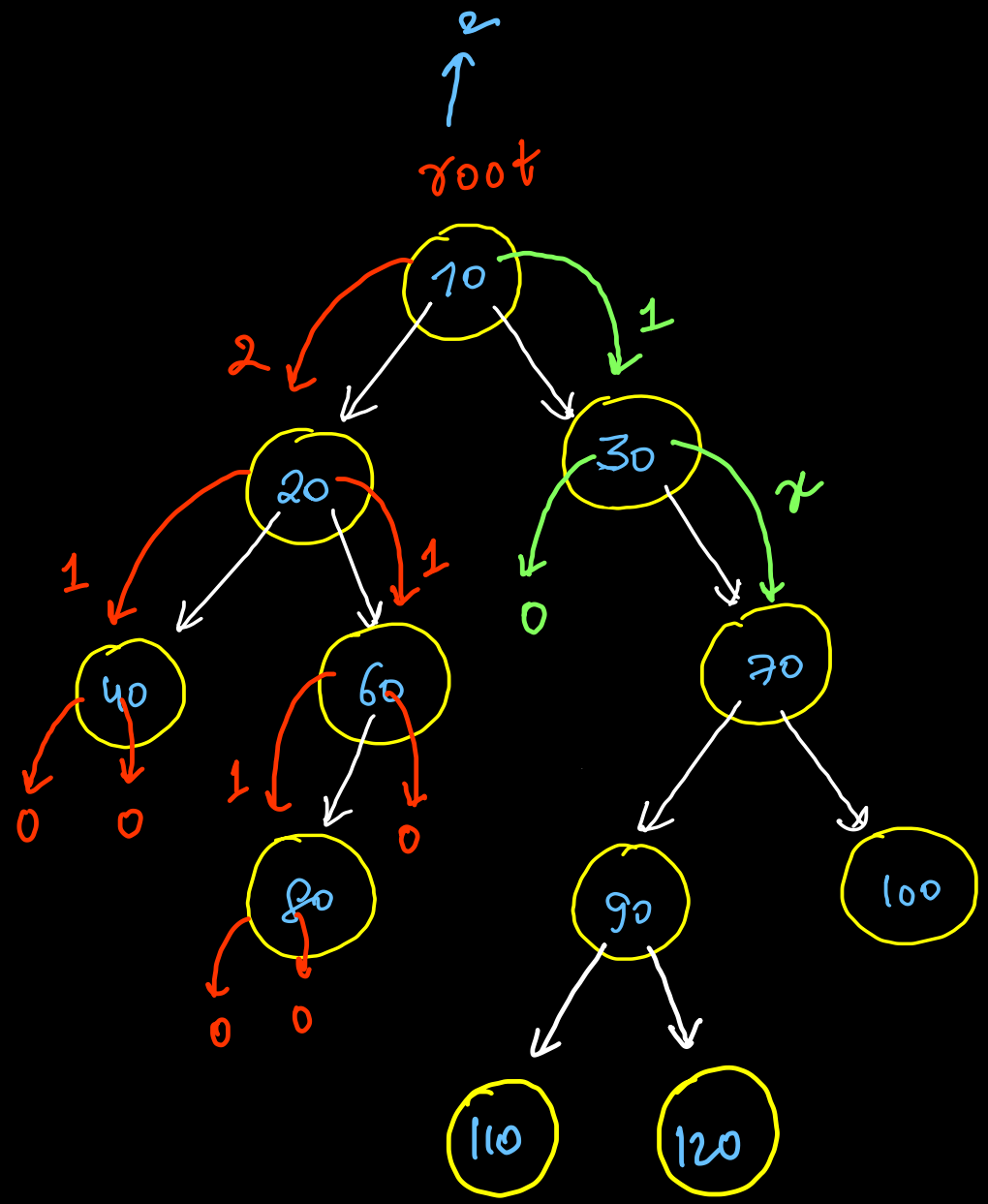
```
int height(TreeNode root) {
    if(root==null) return 0;
    int lh = height(root.left);
    int rh = height(root.right);
    return max(lh,rh)+1;
}
```

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

Time → O(n)
where n = no of nodes

Space → O(h)
where h = avg height of tree

Leetcode 104

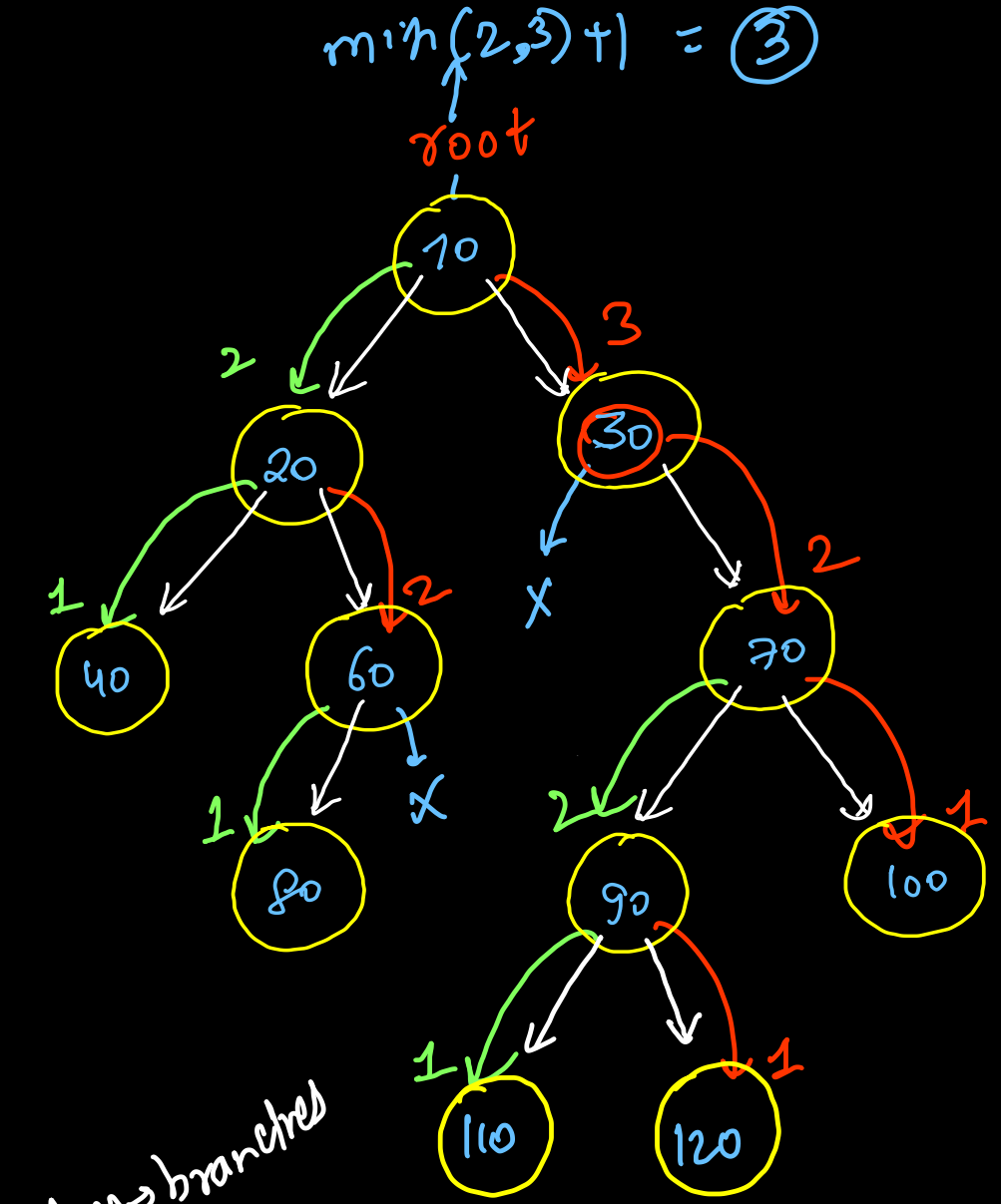


Minm Depth

↳ Dist of Root node to closest leaf node.

```
int height(TreeNode root) {
    if(root==null) return 0;
    int lh = height(root.left);
    int rh = height(root.right);
    return min(lh,rh) + 1;
}
```

↳ wrong logic




```java
public int minDepth(TreeNode root) {
    if(root == null) return 0;
    if(root.right == null)
        return minDepth(root.left) + 1;
    if(root.left == null)
        return minDepth(root.right) + 1;
    int lh = minDepth(root.left);
    int rh = minDepth(root.right);
    return Math.min(lh, rh) + 1;
}
```

(*) single child parent corner cases

Approach 1)

```java
public int minDepth(TreeNode root) {
    if(root == null) return 0;
    if(root.right == null)
        return minDepth(root.left) + 1;
    if(root.left == null)
        return minDepth(root.right) + 1;
    int lh = minDepth(root.left);
    int rh = minDepth(root.right);
    return Math.min(lh, rh) + 1;
}
```

preorder

Approach 2)

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

postorder

Time = O(n)

Space = O(h) → O(logn) avg case

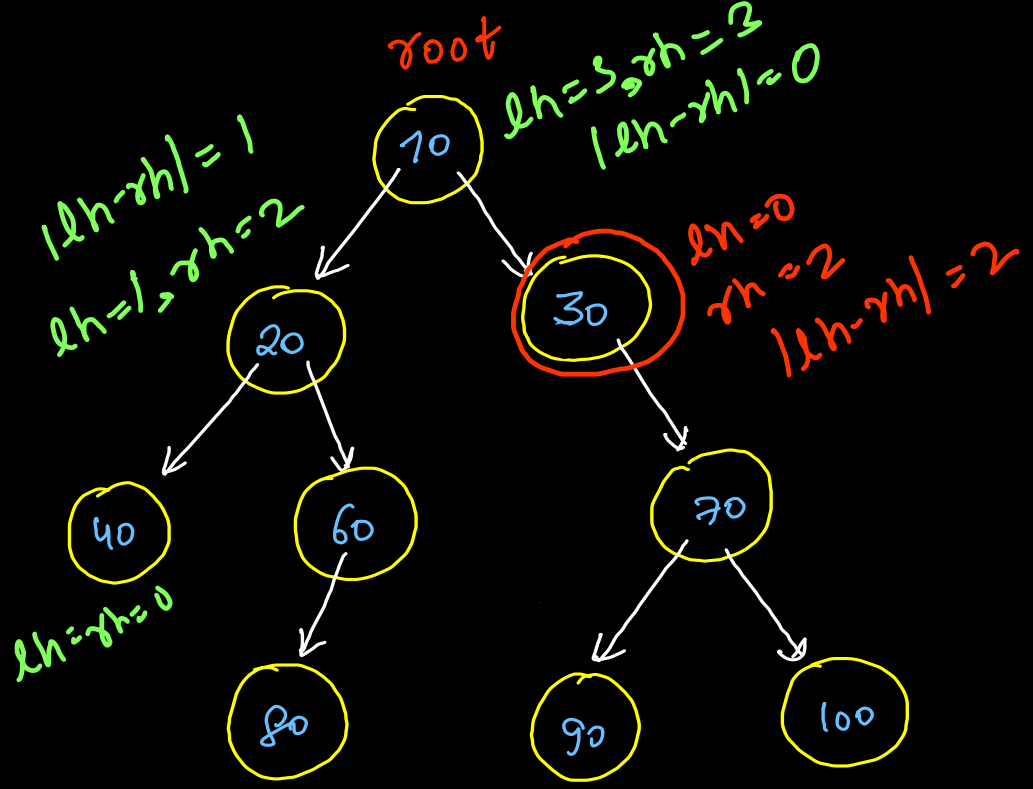
O(n) worst case

DP on Trees
(Rerooting Technique)

Leetcode 110 Balanced Binary Tree

Each node

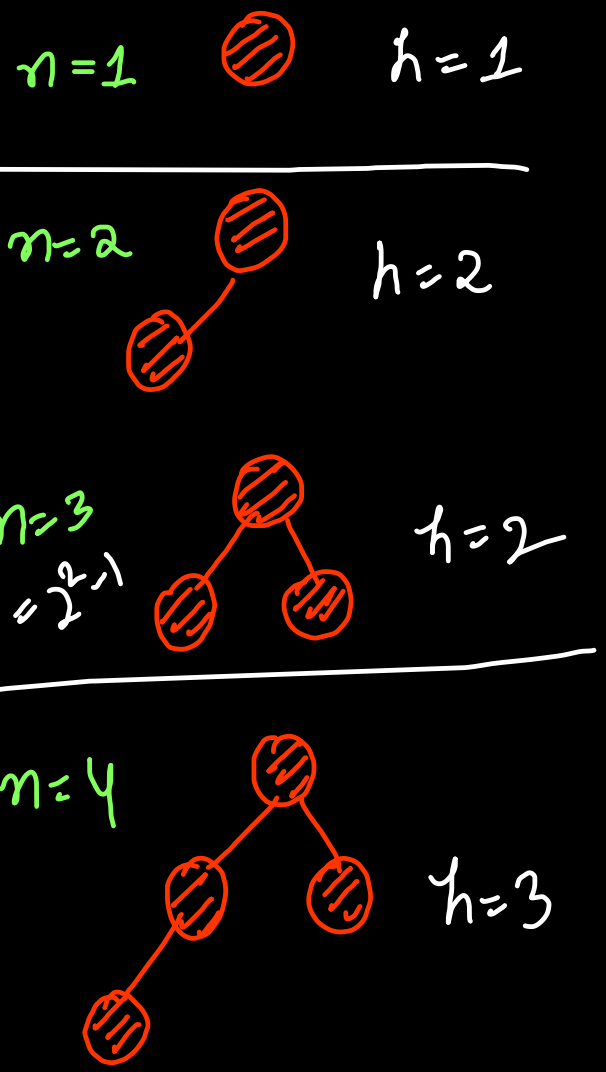
$\Rightarrow$ $|leftheight - rightheight| \leq 1$

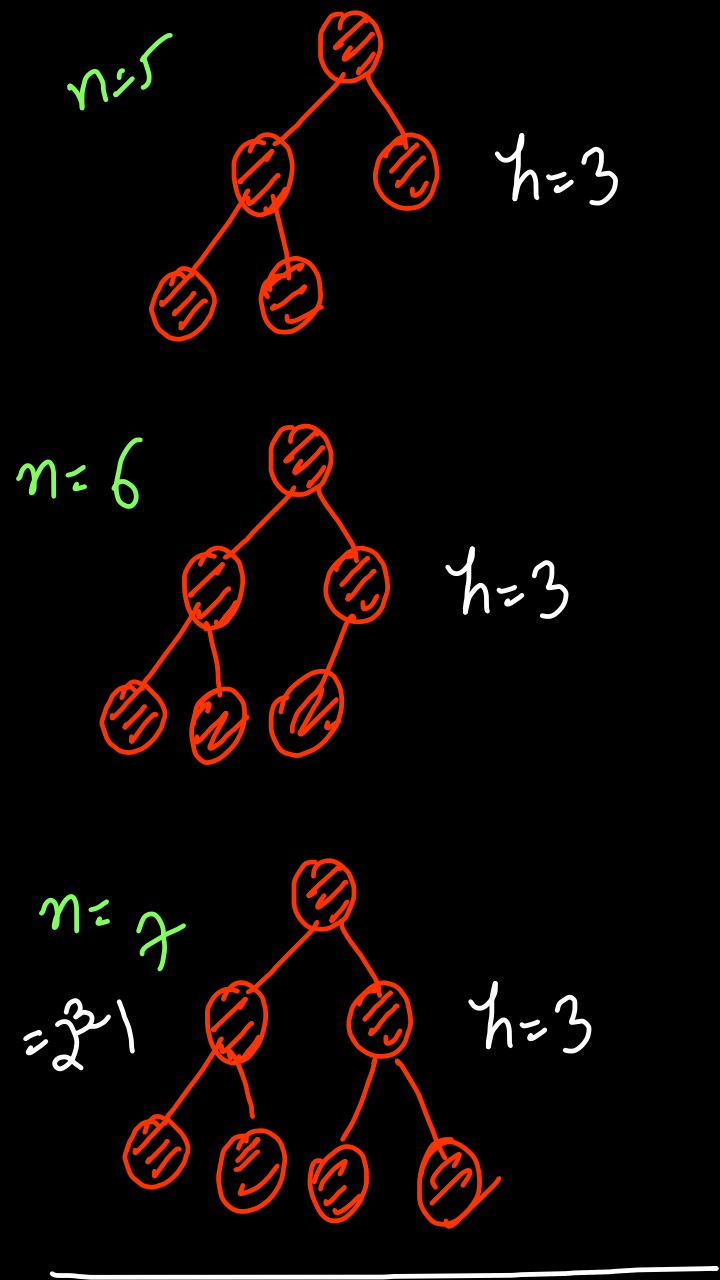


Binary Tree ⟶ BST (height) {balanced}

min height balanced tree

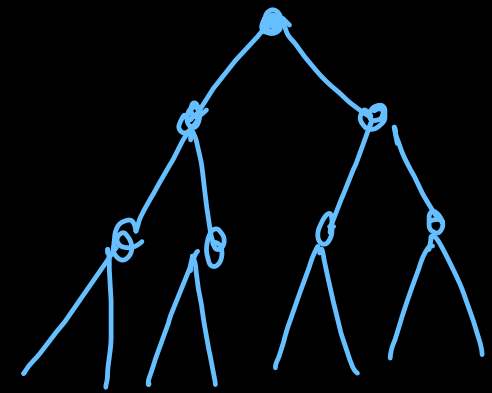
n nodes





n = 8 -----> 15 $2^4 - 1$

h = 4



nodes = n

min height

$= \log_2 n$

best case

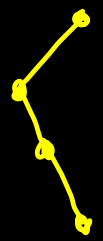
max^m height

Skewed tree

$n=1$  $h=1$

$n=2$  $h=2$

$n=3$  $h=3$

$n=4$  $h=4$

Worstcase

nodes = n

height = n

Approach 1) Brute force

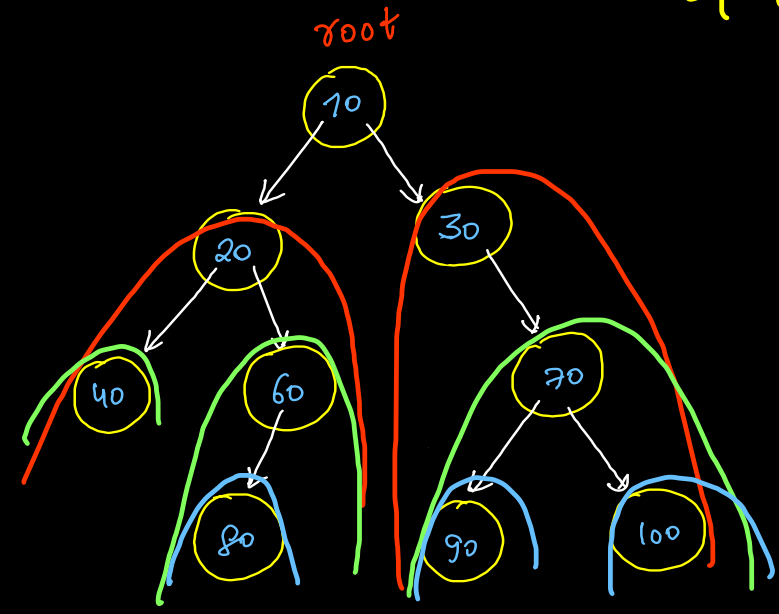
```java
public int height(TreeNode root){
    if(root == null) return 0;

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

    return Math.max(lh, rh) + 1;
}

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

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

    if(Math.abs(lh - rh) > 1) return false;
    return isBalanced(root.left) && isBalanced(root.right);
}
```

dfs $= O(n)$

$O(n^2)$

dfs $= O(n)$

Time $= O(n^2)$, Space $= O(h)$



$1 \cdot 1 + 2 \cdot 2 + 2^2 \cdot 3 + 2^3 \cdot 4 + \cdots 2^{\log n} \cdot \log n$

$= n^2$

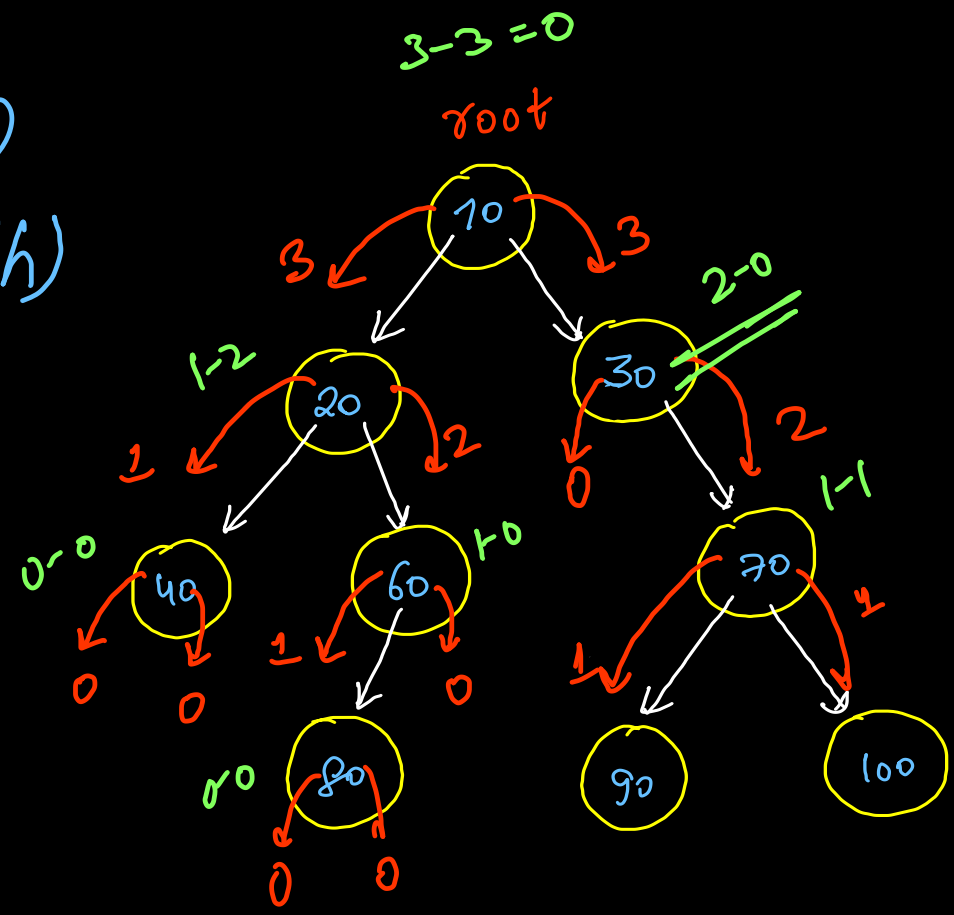
```java
class Solution {
    boolean balanced = true;

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

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

        if(Math.abs(lh - rh) > 1) balanced = false;
        return Math.max(lh, rh) + 1;
    }

    public boolean isBalanced(TreeNode root) {
        height(root);
        return balanced;
    }
}
```

(global)

Time ⇒ O(n)

Space ⇒ O(h)

dfs



Approach 2) Global variable

"global variables are considered evil/bad"

balanced = ~~true~~ false

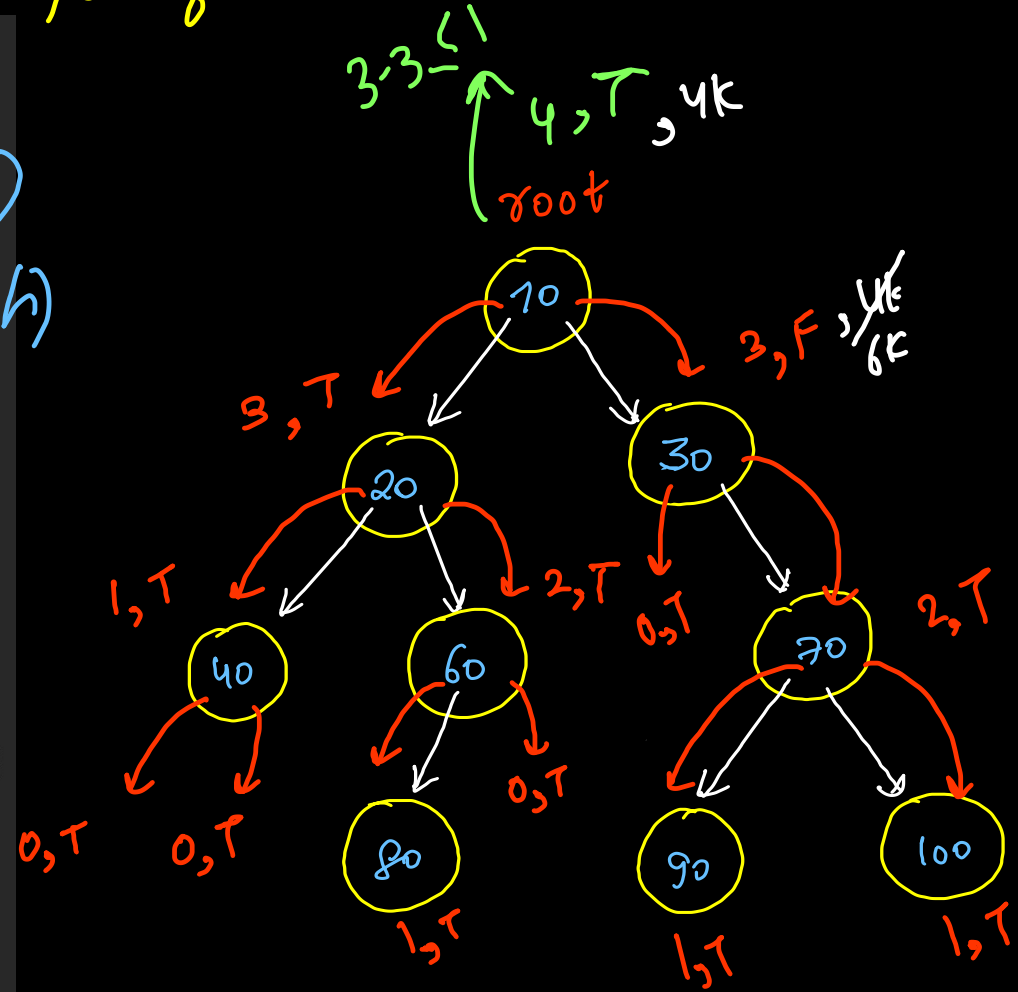
member inner class → object creation w/o object of Sol^n class

```java
class Solution {
    public static class Pair{
        int height = 0;
        boolean balanced = true;
    }

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

        Pair l = dfs(root.left);
        Pair r = dfs(root.right);

        Pair curr = new Pair();
        curr.height = Math.max(l.height, r.height) + 1;
        curr.balanced = l.balanced && r.balanced;
        if(Math.abs(l.height - r.height) > 1) curr.balanced = false;
        return curr;
    }

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

Approach 3)

Time = O(n)

Space = O(h)



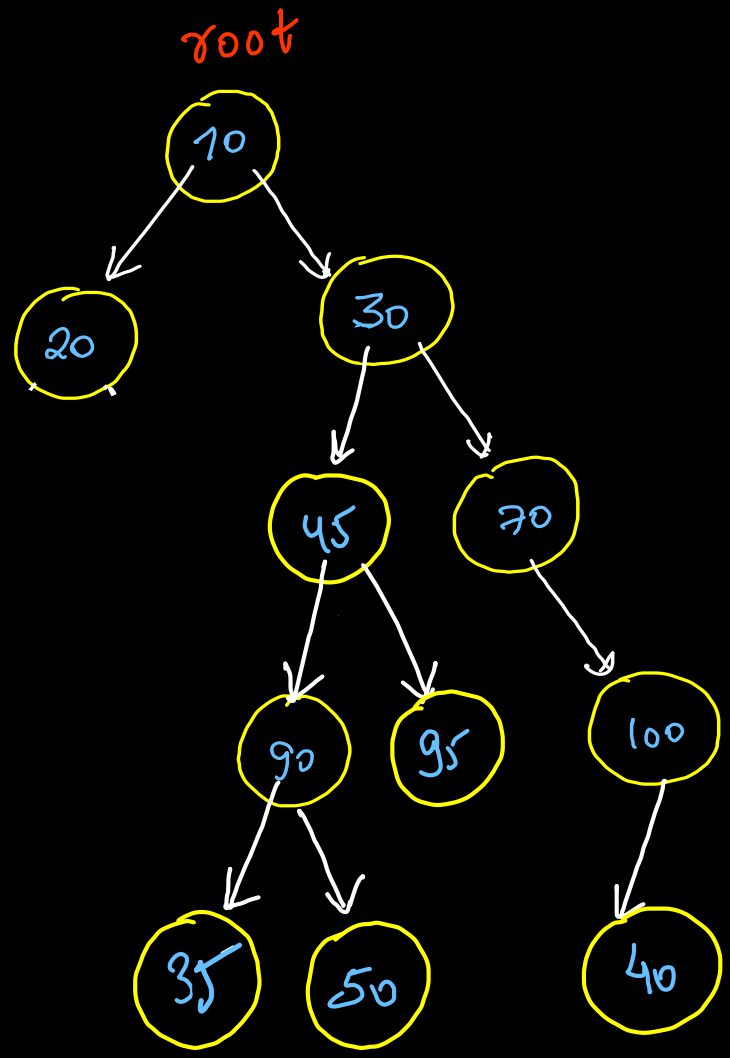
```java
class Solution {
    public int dfs(TreeNode root, boolean[] balanced){
        if(root == null) return 0;
        int lh = dfs(root.left, balanced);
        int rh = dfs(root.right, balanced);

        if(Math.abs(lh - rh) > 1) balanced[0] = false;
        return Math.max(lh, rh) + 1;
    }

    public boolean isBalanced(TreeNode root) {
        boolean[] balanced = {true};
        dfs(root, balanced);
        return balanced[0];
    }
}
```

Approach 4)

Passing balanced variable in array parameter

LC 543 Diameter of Binary Tree

↳ longest node to node distance



diameter = lh + rh + 1

(in terms of nodes)

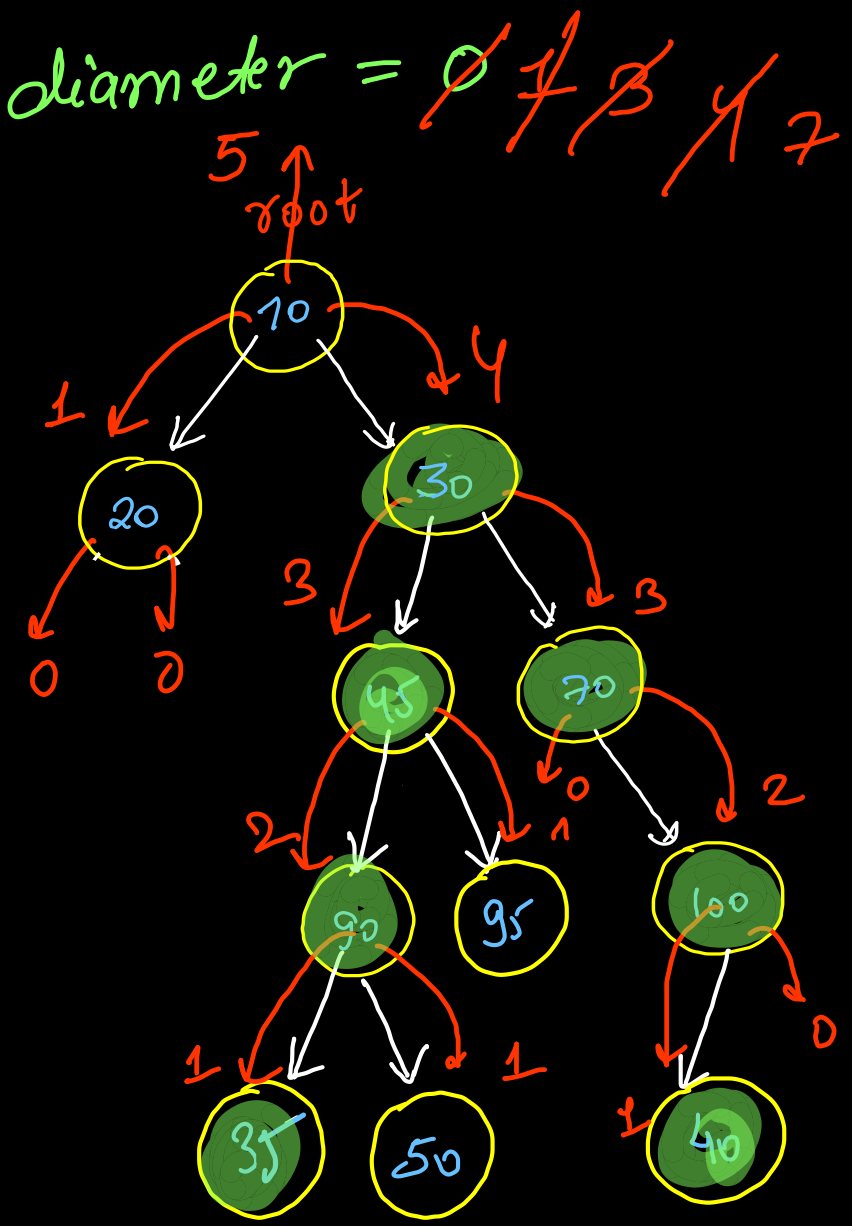
```java
class Solution {
    int diameter = 0;

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

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

        diameter = Math.max(diameter, lh + rh + 1);
        return Math.max(lh, rh) + 1;
    }

    public int diameterOfBinaryTree(TreeNode root) {
        if(root == null) return 0;
        if(root.left == null && root.right == null) return 0;
        height(root);
        return diameter - 1;
    }
}
```

Time = O(n)

Space = O(h)

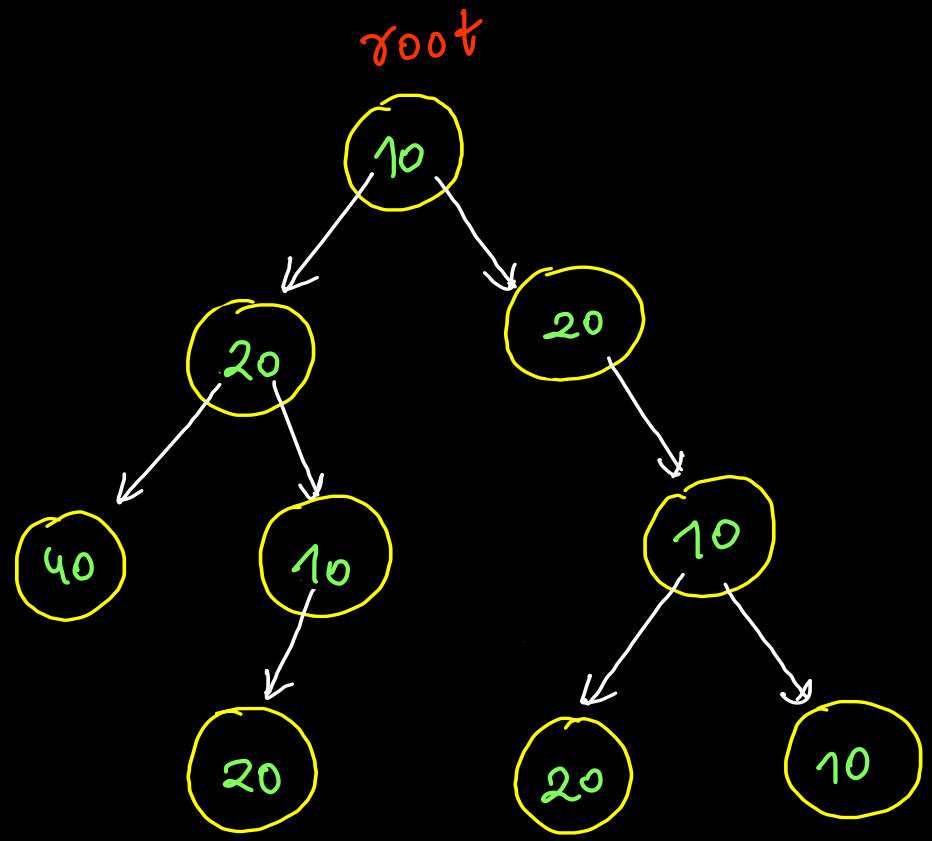
→ empty tree

↳ number of edges



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

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

        diameter[0] = Math.max(diameter[0], lh + rh + 1);
        return Math.max(lh, rh) + 1;
    }

    public int diameterOfBinaryTree(TreeNode root) {
        if(root == null) return 0;
        int[] diameter = {0};
        height(root, diameter);
        return diameter[0] - 1;
    }
}
```

lc 1367) Linked List in Binary Tree



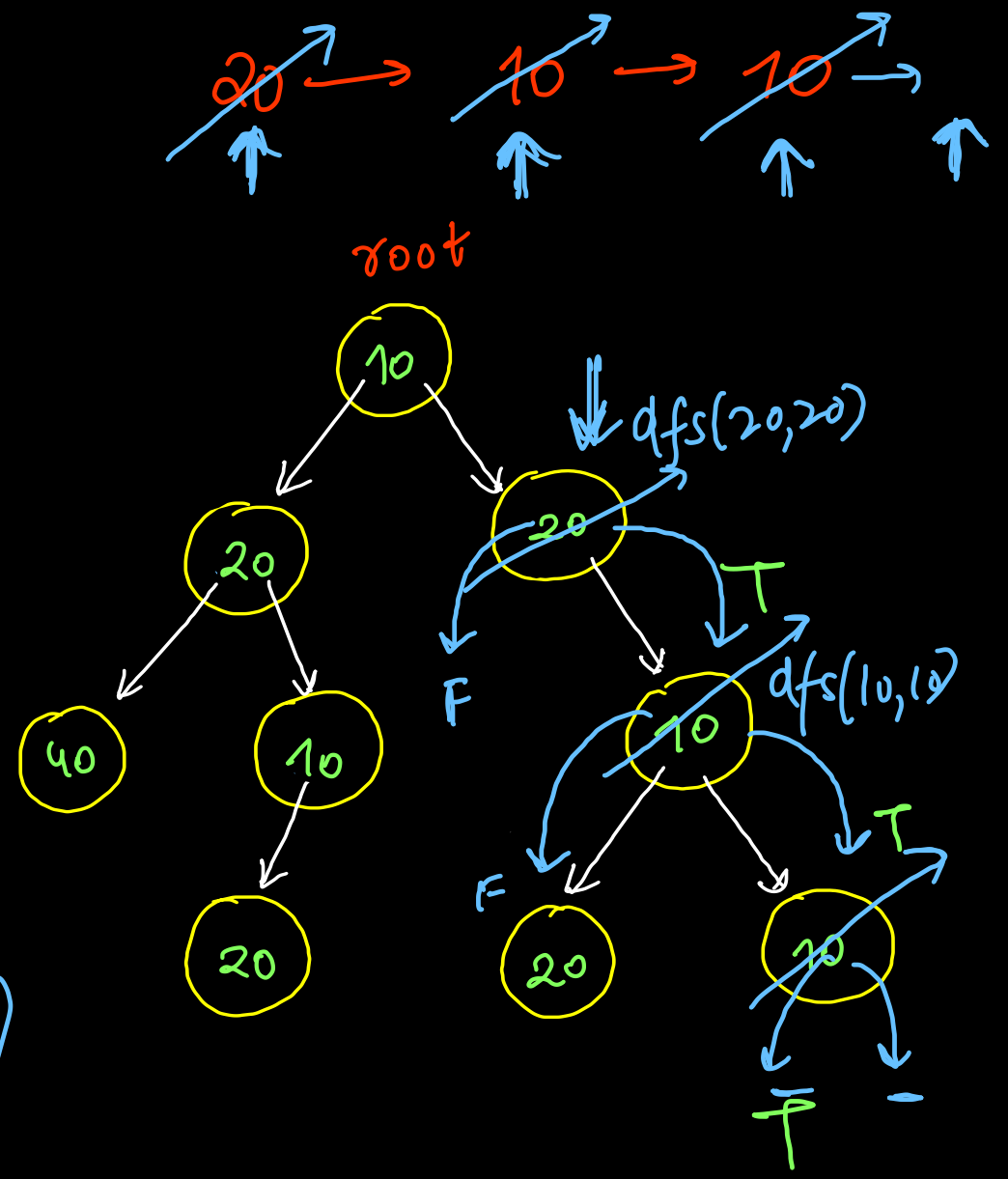
" 20 → 10 → 10 "

⇑
head

```
dfs(TreeNode root, ListNode head)
{
    if(head==null) return true;
    if (root==null) return false;
   if (root.val != head.data)
            return false;
    return  dfs(root.left, head.next)
        || dfs(root.right, head.next);
}
```

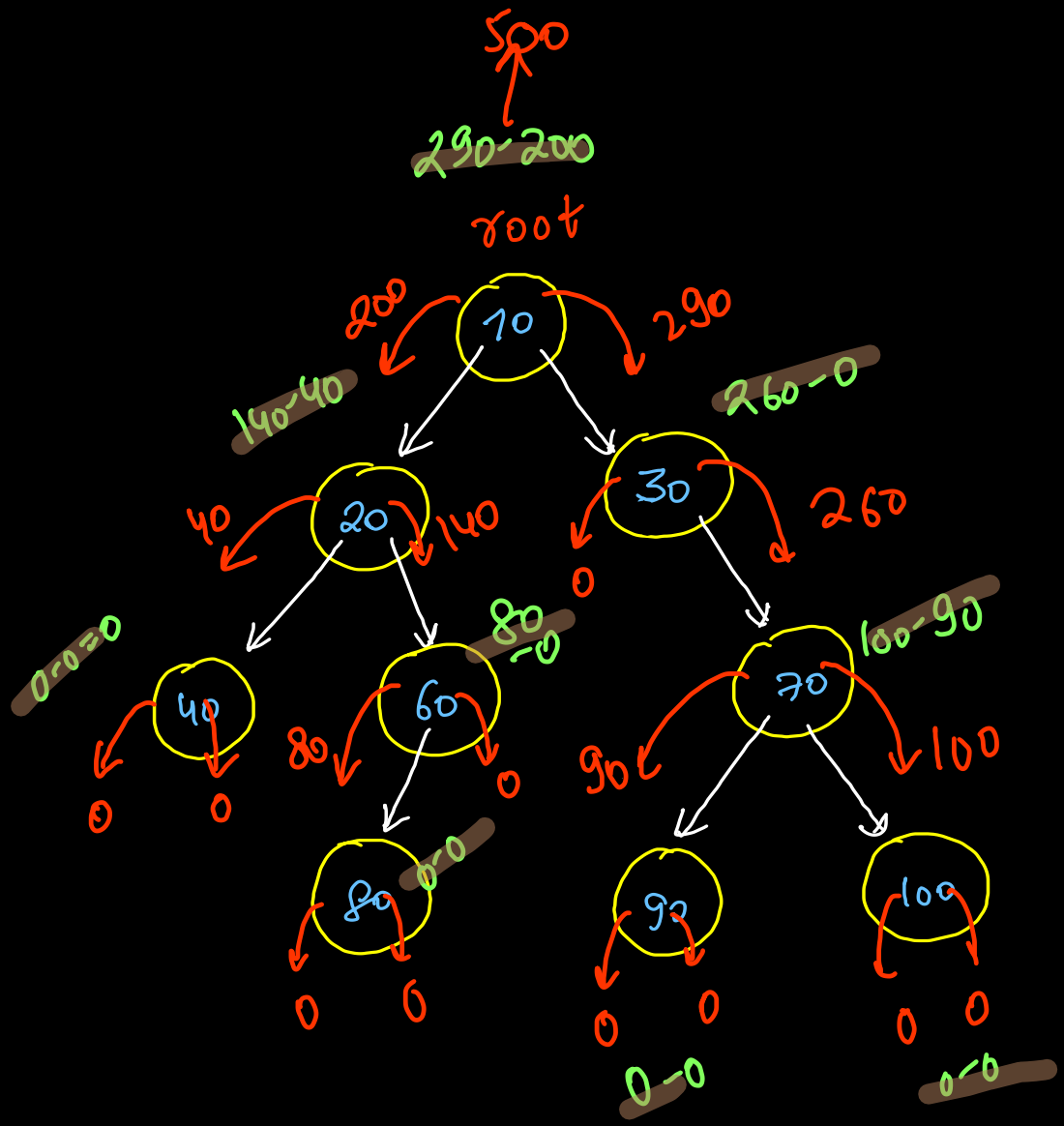
```java
class Solution {
    // fixing head node as the root node
    public boolean dfs(ListNode head, TreeNode root){
        if(head == null) return true;
        if(root == null) return false;
        if(head.val != root.val) return false;
        return dfs(head.next, root.left) || dfs(head.next, root.right);
    }

    // we can start linked list from any node
    public boolean isSubPath(ListNode head, TreeNode root) {
        if(head == null) return true;
        if(root == null) return false;
        return dfs(head, root) || isSubPath(head, root.left)
            || isSubPath(head, root.right);
    }
}
```

O(n) * O(n)



Time ⇒ $O(n^2)$, Space ⇒ $O(n)$

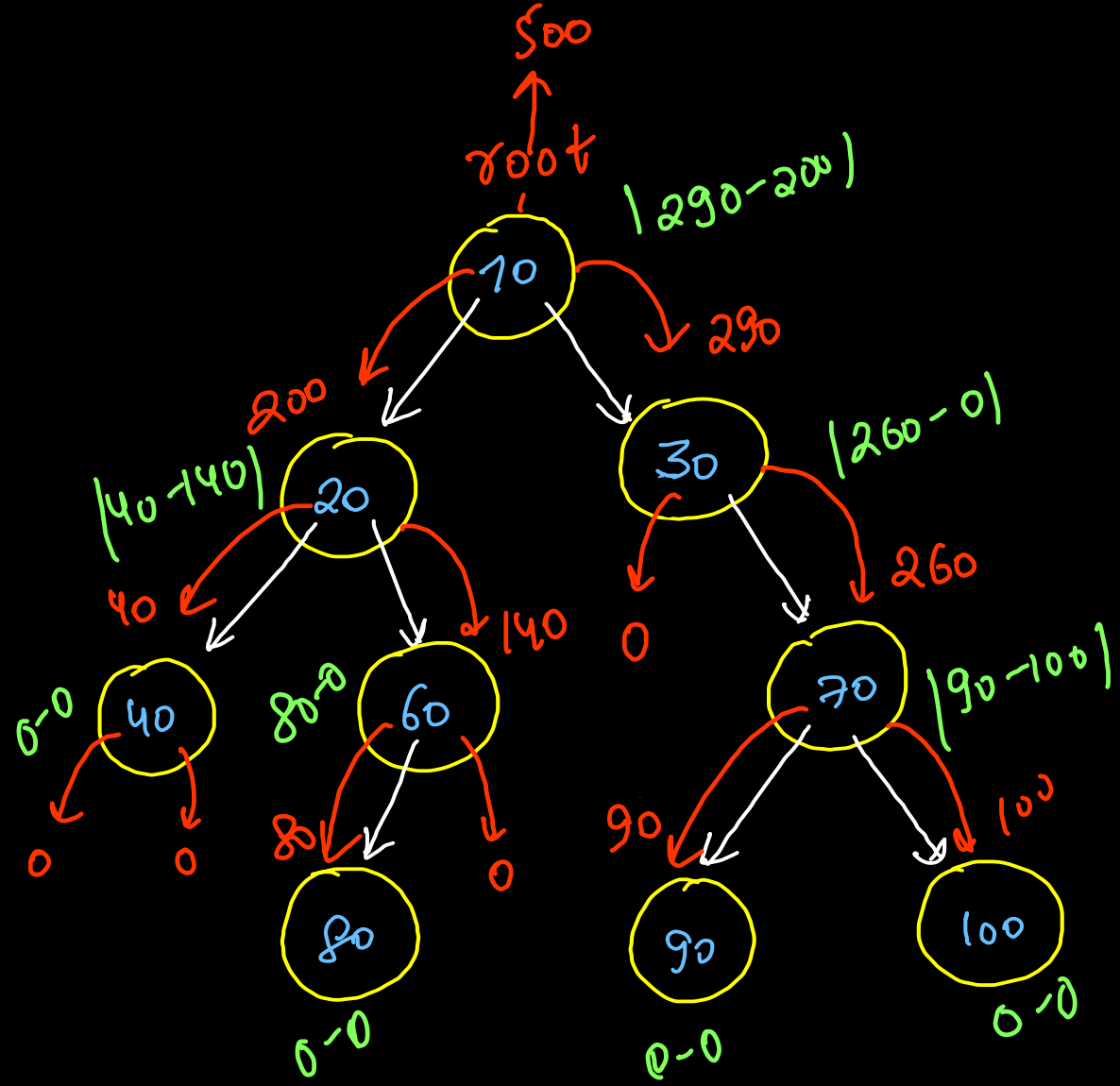
2d dp



Binary Tree Tilts

Sum of all tilts

$\Rightarrow |leftSum - rightSum|$



Subtree sum

⇒ leftsubtree sum

+

rightsubtree sum

+

root.val

tilt

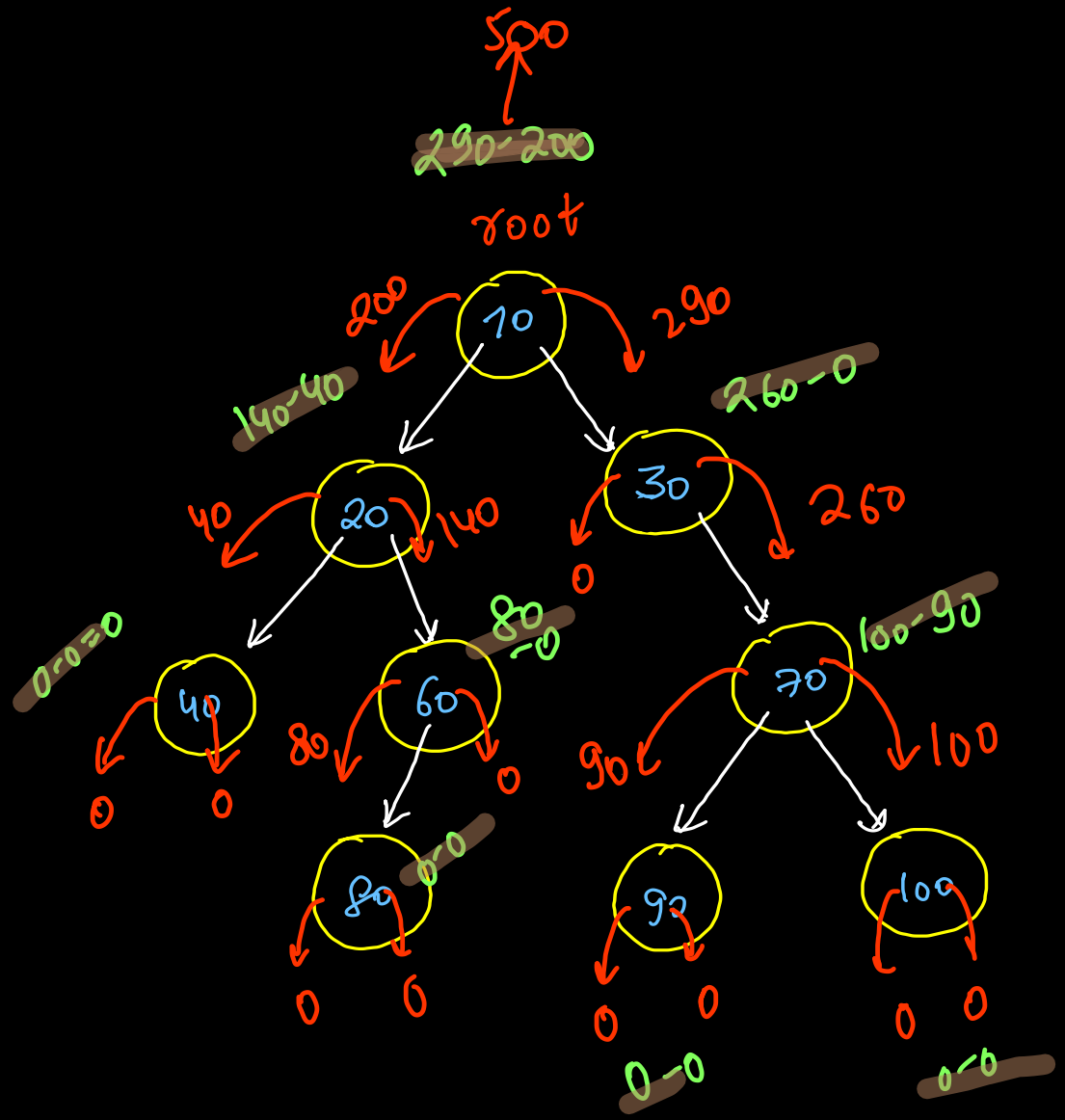
|leftsum

−

rightsum|

Sum of all tilts

tilt = 0



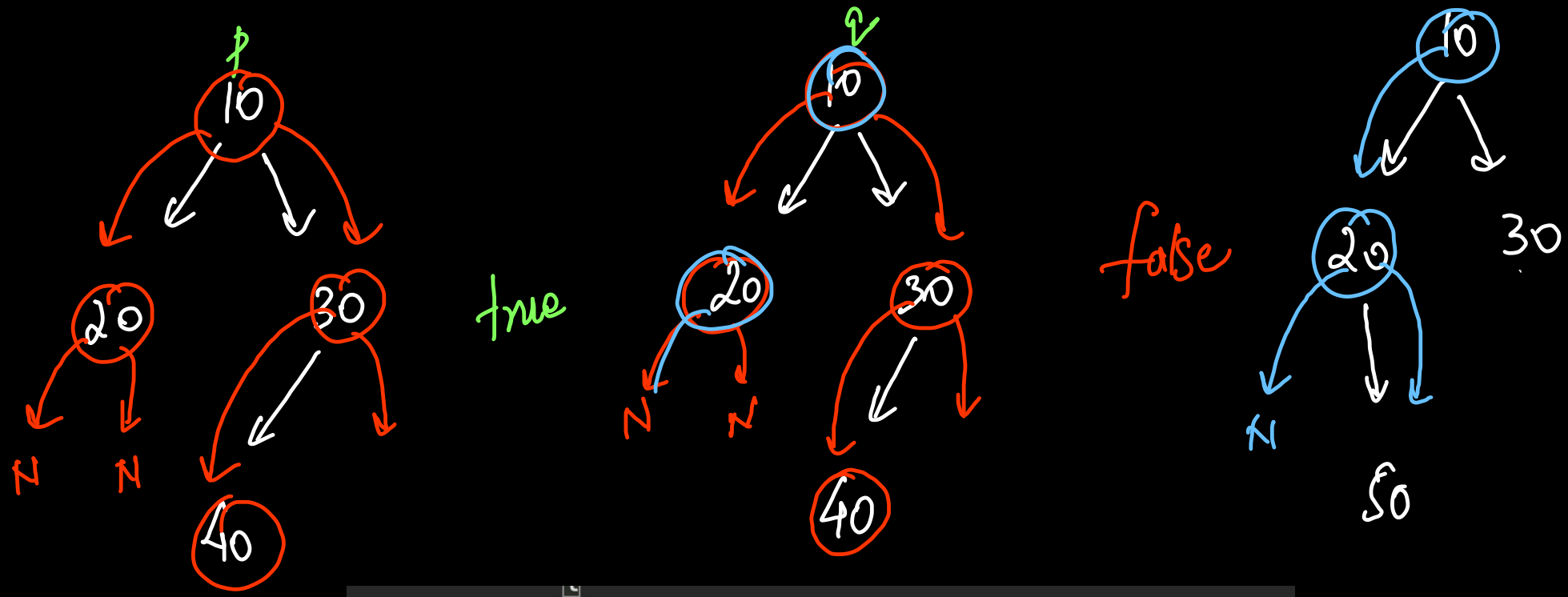
Time = O(n), space = O(n)

```java
class Solution {
    int ans = 0;

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

        int lsum = subtreeSum(root.left);
        int rsum = subtreeSum(root.right);

        int tilt = Math.abs(lsum - rsum);
        ans += tilt;

        return lsum + rsum + root.val;
    }

    public int findTilt(TreeNode root) {
        subtreeSum(root);
        return ans;
    }
}
```

green value

red value

LC 100) Same Tree

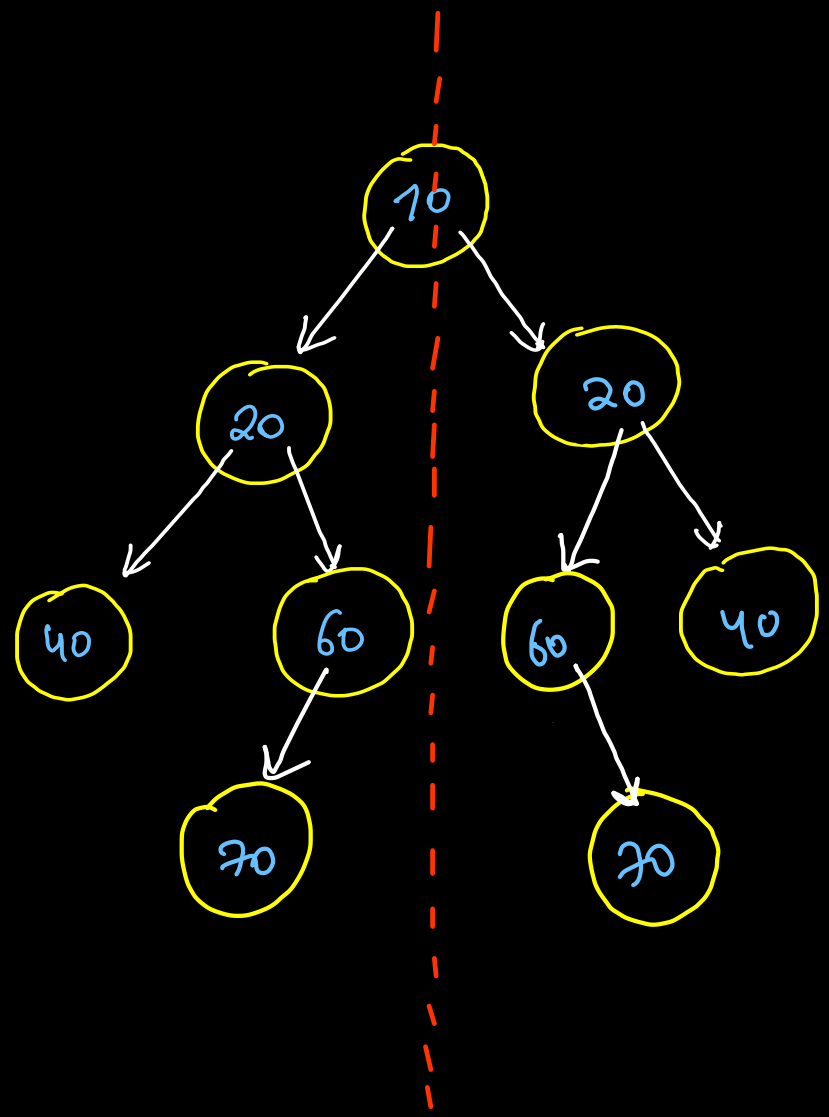



```java
public boolean isSameTree(TreeNode p, TreeNode q) {
    if(p == null && q == null) return true;
    if(p == null || q == null) return false;
    if(p.val != q.val) return false;

    return isSameTree(p.left, q.left)
        && isSameTree(p.right, q.right);
}
```

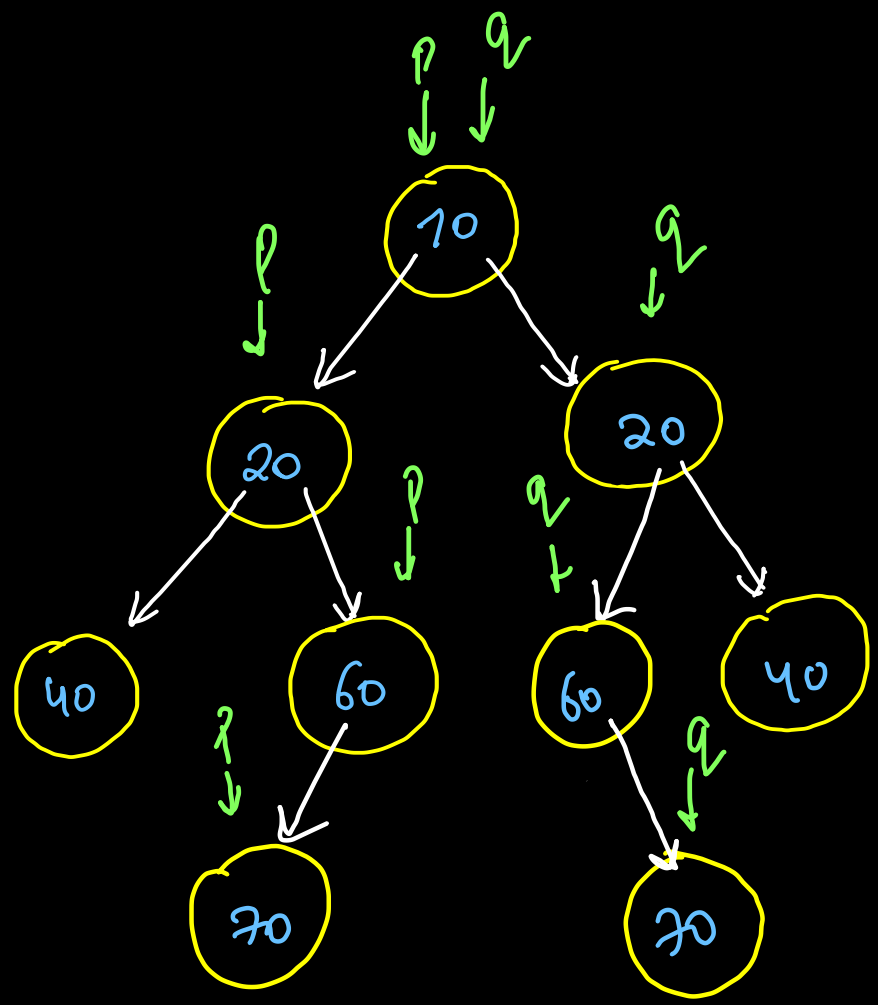
Time ⇒ O(n)

Space ⇒ O(n)

LC 101 Symmetric Tree



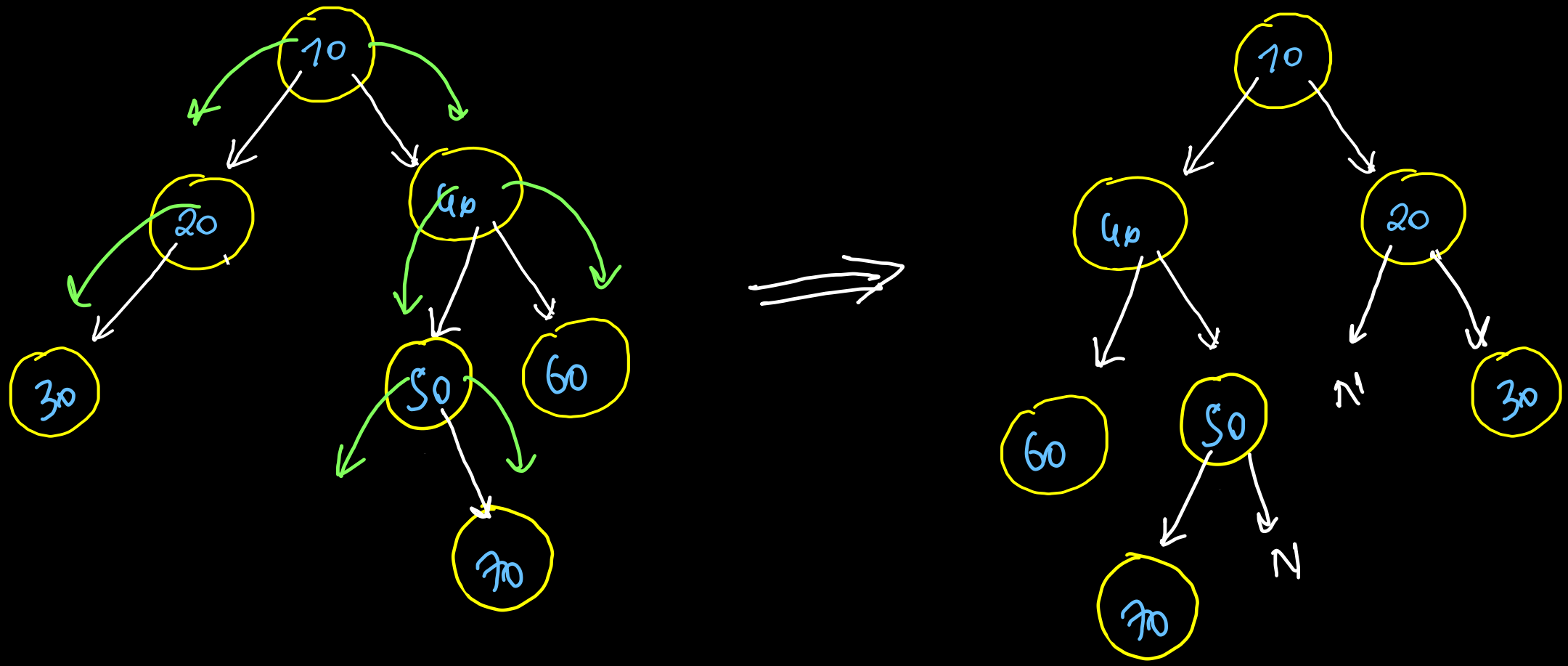

```
isMirror(TreeNode p, TreeNode q)
{
    if(p==null && q==null)
        return true;
    if(p==null || q==null)
        return false;
    if(p.val != q.val) return false;
    return isMirror(p.left, q.right)
        && isMirror(p.right, q.left);
}
```




```java
class Solution {
    public boolean isMirror(TreeNode p, TreeNode q){
        if(p == null && q == null) return true;
        if(p == null || q == null) return false;
        if(p.val != q.val) return false;

        return isMirror(p.left, q.right)
            && isMirror(p.right, q.left);
    }

    public boolean isSymmetric(TreeNode root) {
        return isMirror(root, root);
    }
}
```

↳ Time = O(n), Space = O(n)

Invert Binary Tree



Appl
Postorder

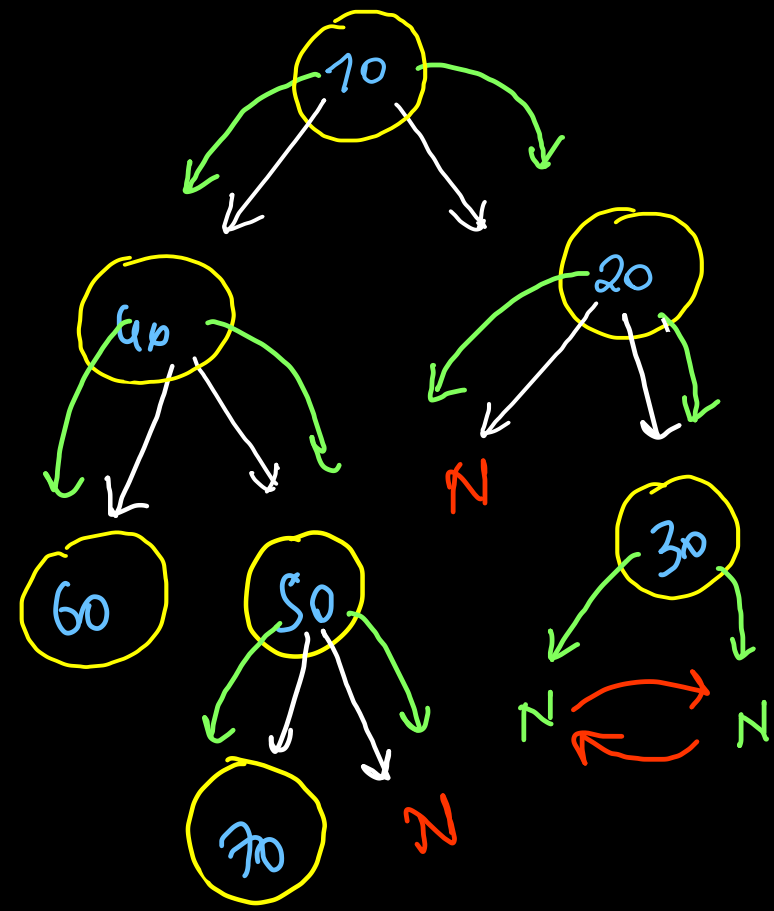
```java
public TreeNode invertTree(TreeNode root) {
    if(root == null) return null;

    TreeNode l = invertTree(root.left);
    TreeNode r = invertTree(root.right);

    root.left = r; root.right = l;
    return root;
}
```

Appz
Preorder

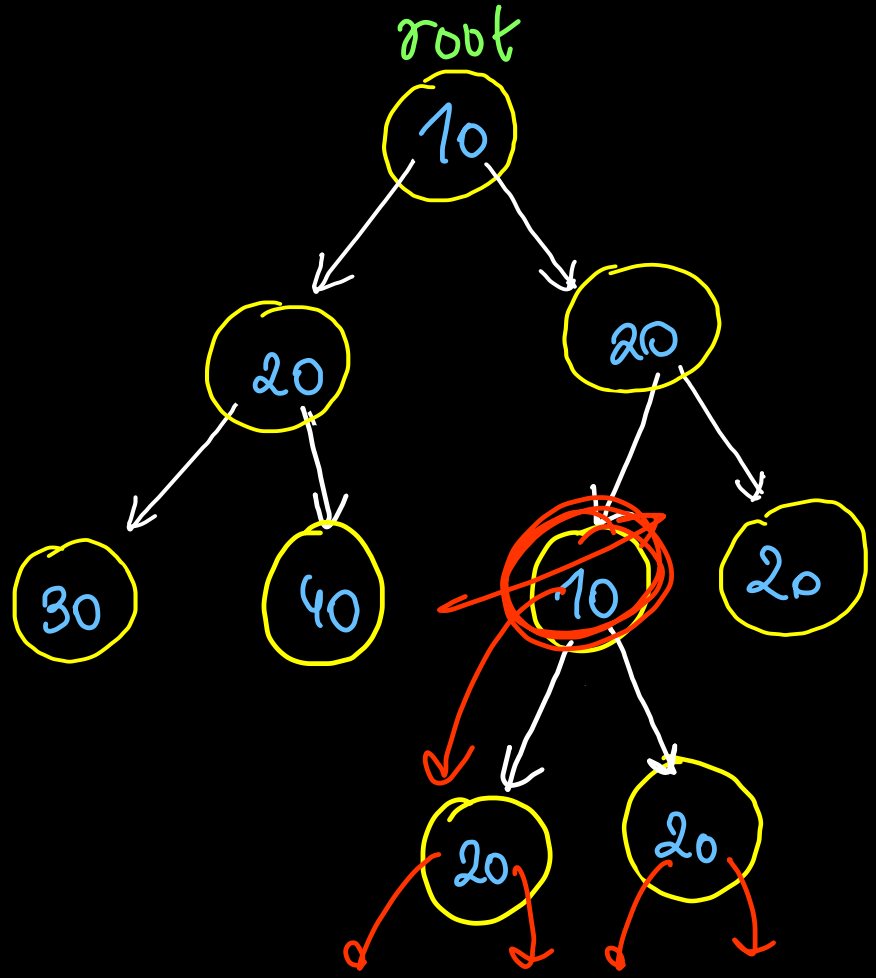
```java
public TreeNode invertTree(TreeNode root) {
    if(root == null) return null;

    TreeNode temp = root.left;
    root.left = root.right; root.right = temp;

    invertTree(root.left);
    invertTree(root.right);

    return root;
}
```

Time ⇒ O(n)

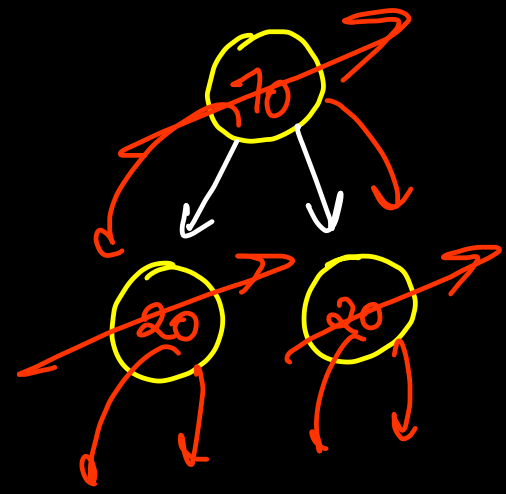
Space ⇒ O(h)



LC572 Subtree of Another Tree



Subtree

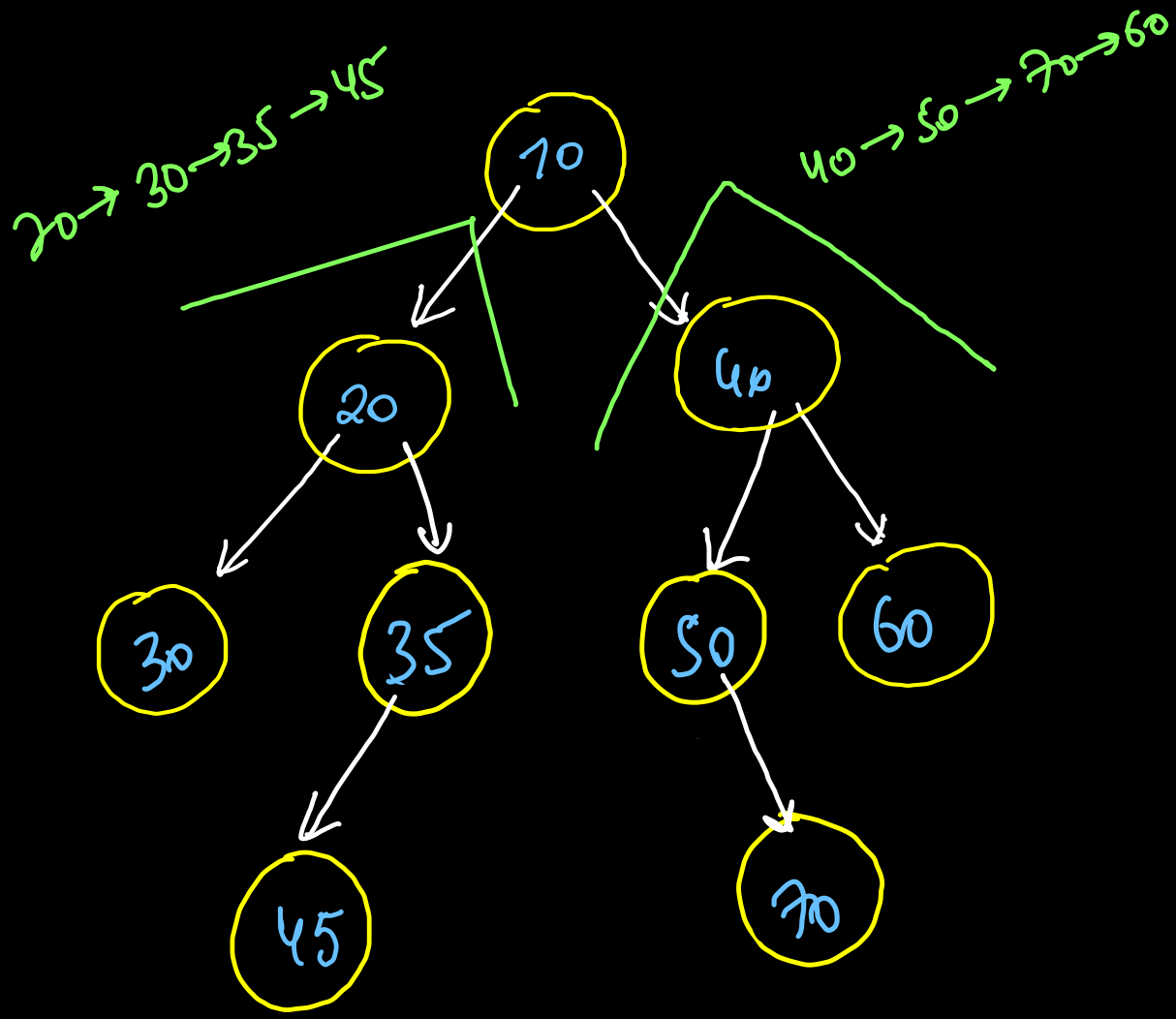


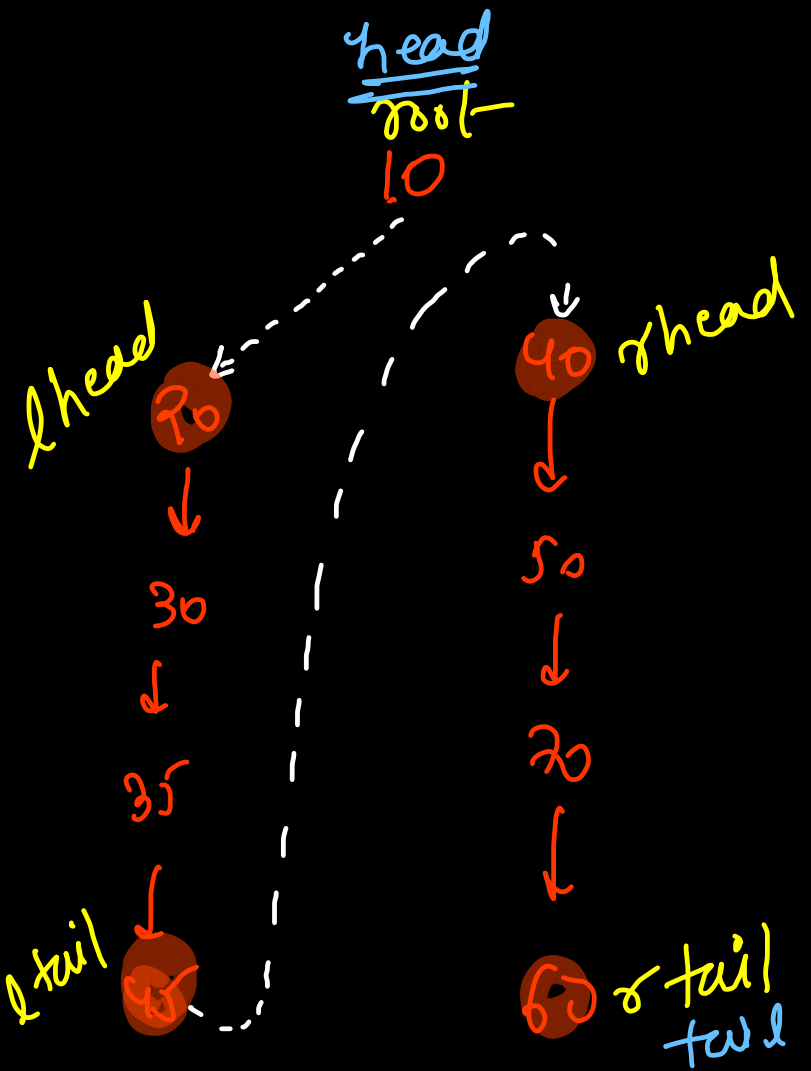
```java
class Solution {
    // fix subtree starting at the root node
    public boolean isSame(TreeNode root, TreeNode subRoot){
        if(root == null && subRoot == null) return true;
        if(root == null || subRoot == null) return false;
        if(root.val != subRoot.val) return false;

        return isSame(root.left, subRoot.left)
            && isSame(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 isSame(root, subRoot) || isSubtree(root.left, subRoot)
            || isSubtree(root.right, subRoot);
    }
}
```

Time
↳ $O(n^2)$

Space
↳ $O(n)$

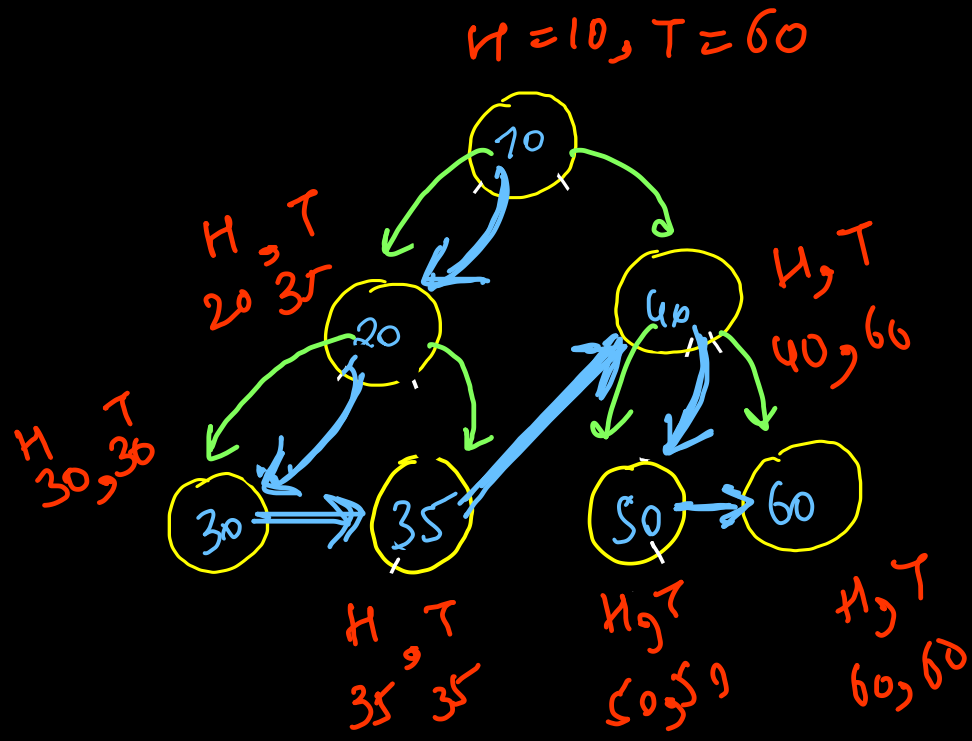
Flatten/Linearize Binary Tree to Linked List





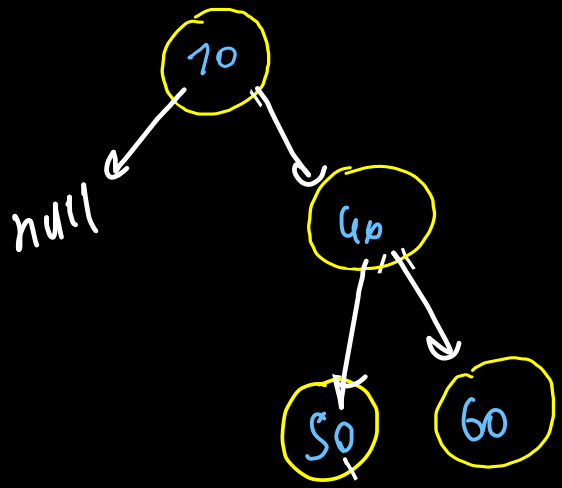
```java
public static class Pair{
    TreeNode head, tail;
    Pair(TreeNode head, TreeNode tail){
        this.head = head;
        this.tail = tail;
    }
}

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

    if(root.left == null && root.right == null)
        return new Pair(root, root);

    Pair l = dfs(root.left);// linearize left subtree
    Pair r = dfs(root.right);// & right subtree

    root.left = null;
    root.right = l.head;
    l.tail.right = r.head;

    return new Pair(root, r.tail);
}
```

leaf node



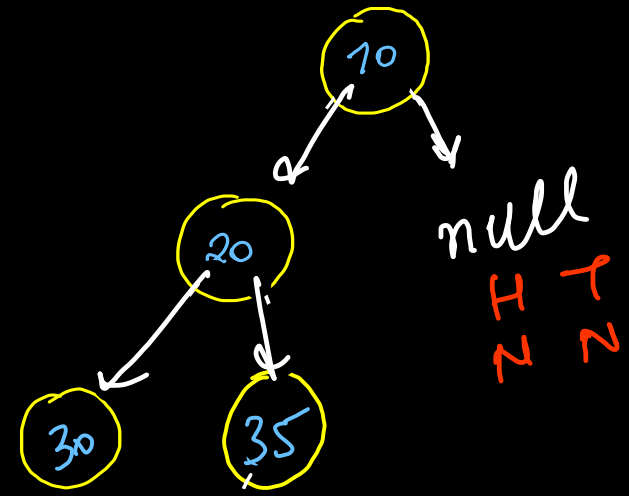
Time ⇒ O(n)

Space ⇒ O(h)



10 → 40 → 50 → 60

Corner cases



10 → 20 → 30 → 35

~~tail~~

```java
public static class Pair{
    TreeNode head, tail;
    Pair(TreeNode head, TreeNode tail){
        this.head = head;
        this.tail = tail;
    }
}
```

```java
public void flatten(TreeNode root) {
    dfs(root);
}
```

Time = O(n)

Space = O(h)

```java
public Pair dfs(TreeNode root){
    if(root == null)
        return new Pair(null, null);

    if(root.left == null && root.right == null)
        return new Pair(root, root);

    if(root.left == null){
        Pair r = dfs(root.right);
        return new Pair(root, r.tail);
    }

    Pair l = dfs(root.left);
    Pair r = dfs(root.right);

    root.left = null;
    root.right = l.head;
    if(r.head == null)
        return new Pair(root, l.tail);

    l.tail.right = r.head;
    return new Pair(root, r.tail);
}
```

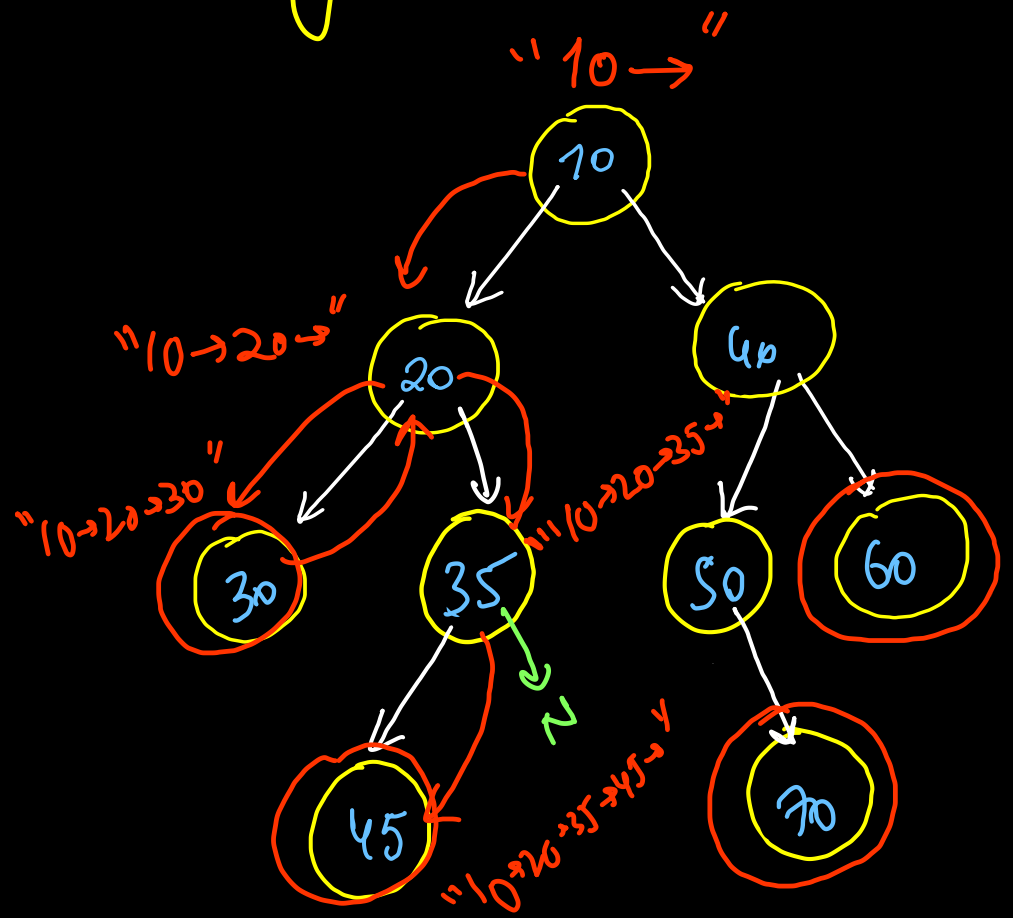
leaf node

only right child

only left child

Path Sum Problems

Binary Tree Paths LC 257 : all root to leaf paths



{ "10→20→30"

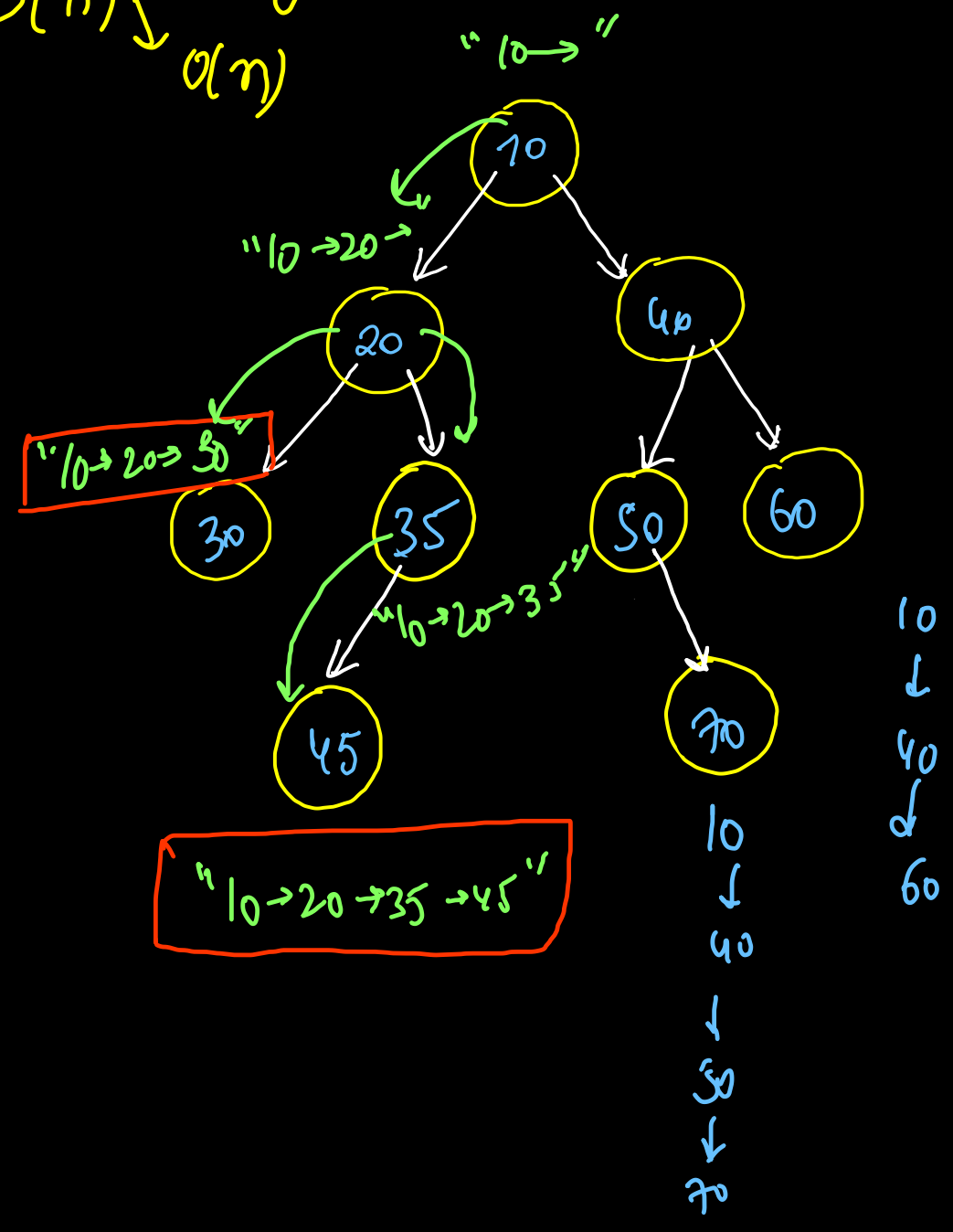
"10→20→35→45"

"10→40→50→70"

"10→40→60" }

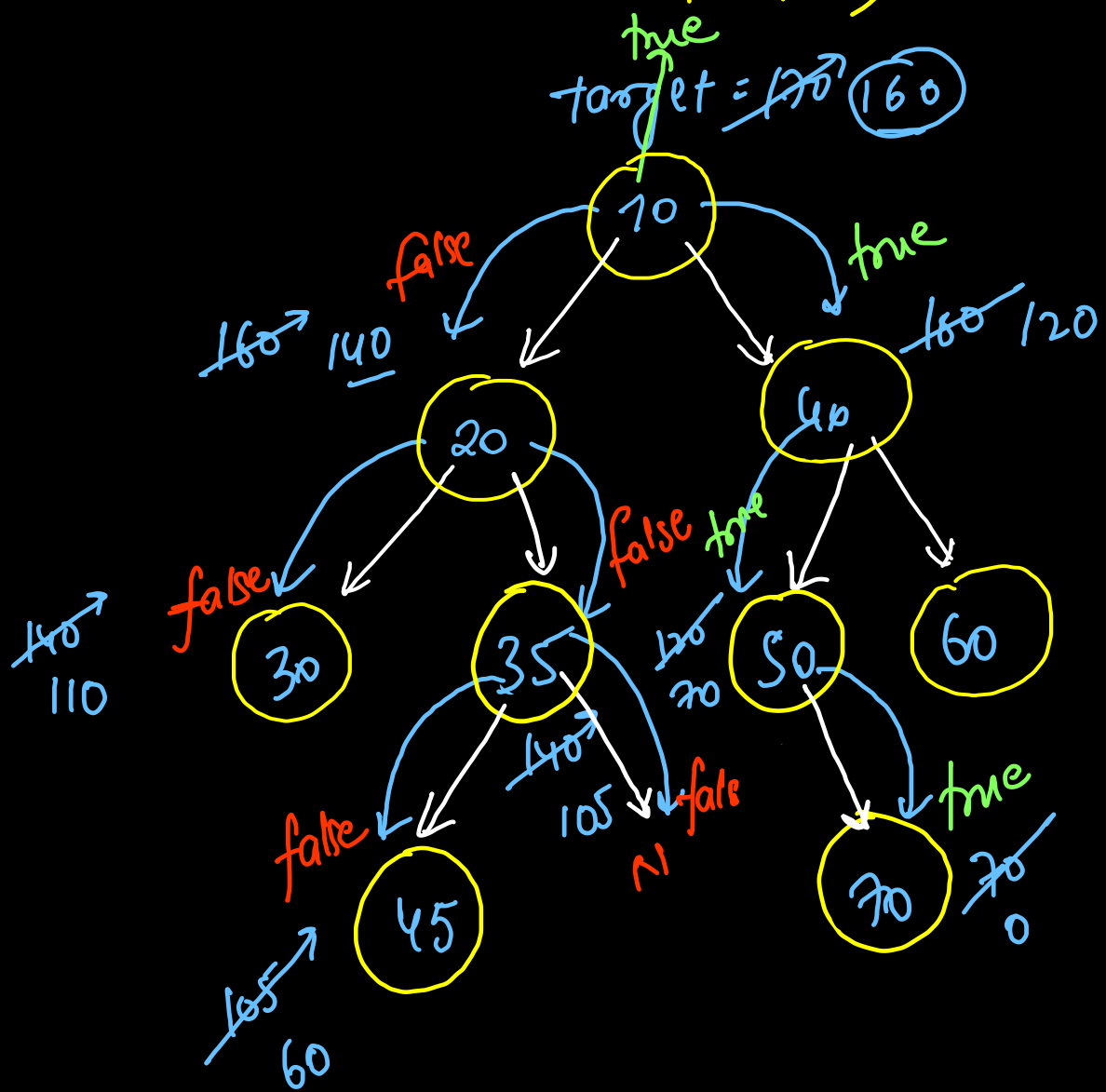
Time = O(n), Space = O(h) ↘ O(logn) / O(n)

```java
List<String> paths = new ArrayList<>();

public void dfs(TreeNode root, String path){
    if(root == null) return;

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

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

public List<String> binaryTreePaths(TreeNode root) {
    dfs(root, "");
    return paths;
}
```



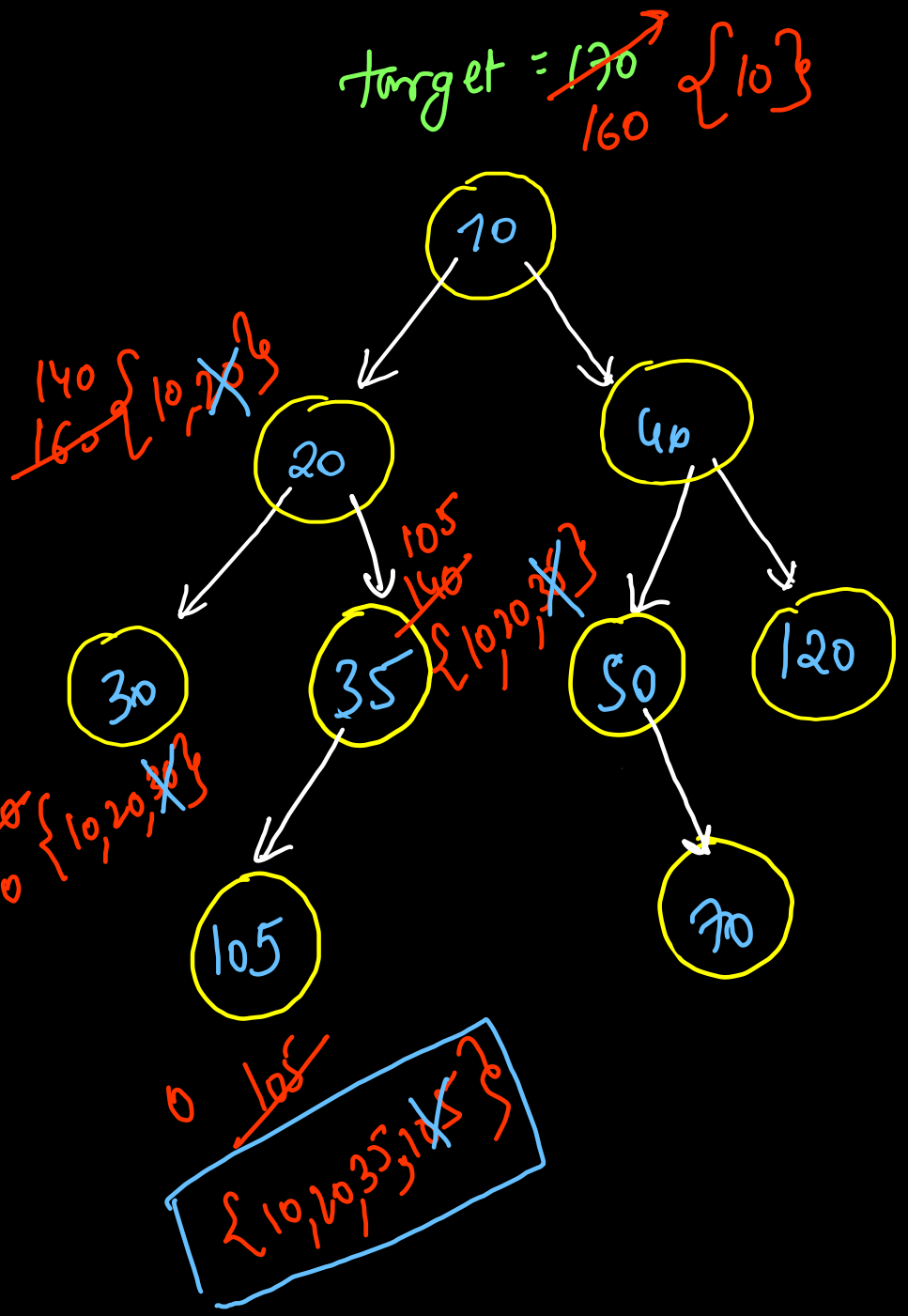
LC 112) Path Sum - I

target = 170




```java
public boolean hasPathSum(TreeNode root, int targetSum) {
    if(root == null) return false;

    targetSum -= root.val;
    if(root.left == null && root.right == null)
        return (targetSum == 0);

    return hasPathSum(root.left, targetSum)
        || hasPathSum(root.right, targetSum);
}
```

Time ⇒ O(n), space ⇒ O(h)



RC 113) Path Sum - II

All targetsum root to leaf paths

{ {10, 20, 35, 105},

{ 10, 40, 50, 70},

{ 10, 40, 120 } }

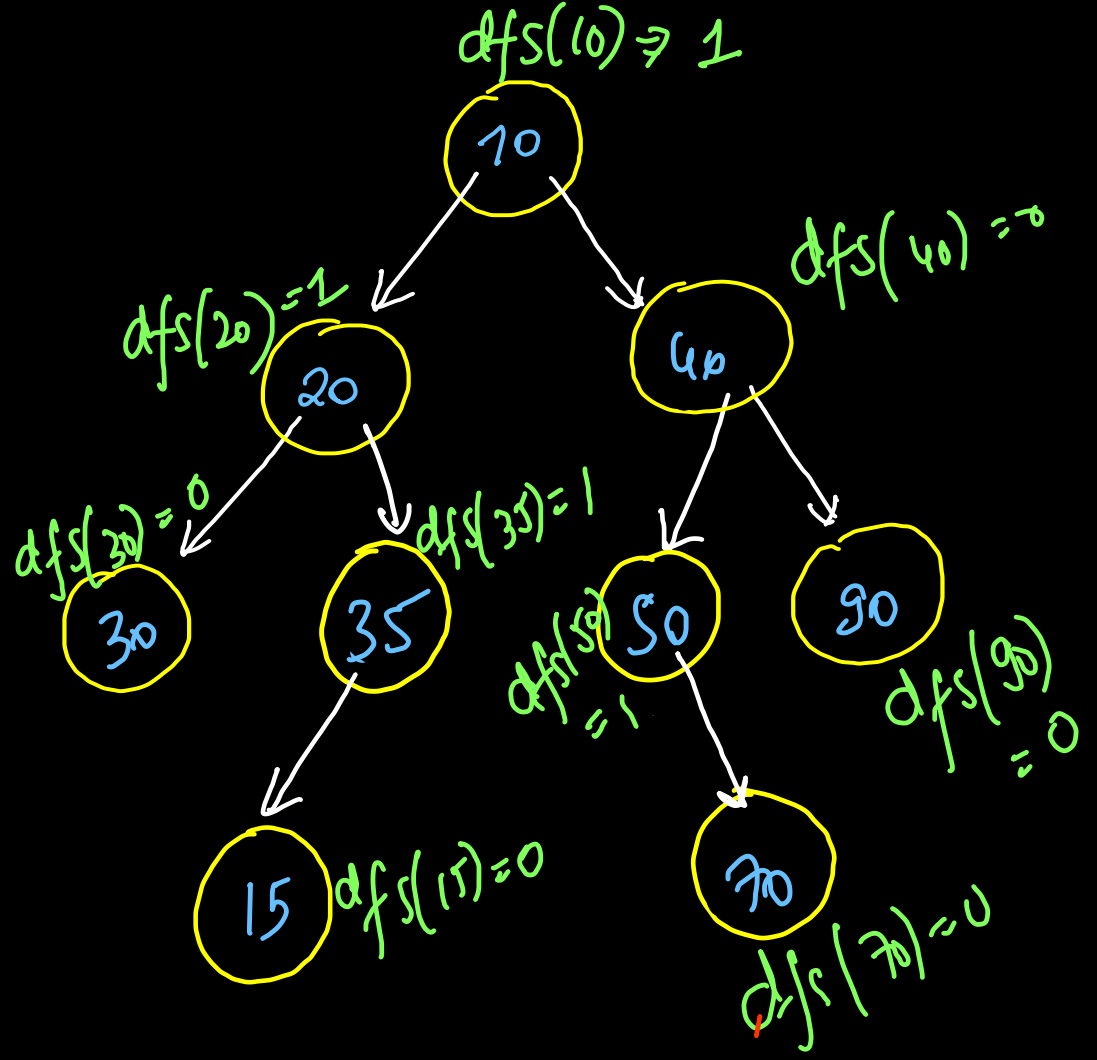
```java
class Solution {
    List<List<Integer>> paths = new ArrayList<>();

    public void dfs(TreeNode root, int targetSum, List<Integer> path) {
        if(root == null) return;

        targetSum -= root.val;
        path.add(root.val);

        if(root.left == null && root.right == null && targetSum == 0){
            paths.add(new ArrayList<>(path)); // deep copy
        }

        dfs(root.left, targetSum, path);
        dfs(root.right, targetSum, path);
        path.remove(path.size() - 1); // backtracking
    }

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

Time ⇒ O(n)

dfs

Space ⇒ O(h)

LC 437
Path Sum - III



targetSum ⇒ 50

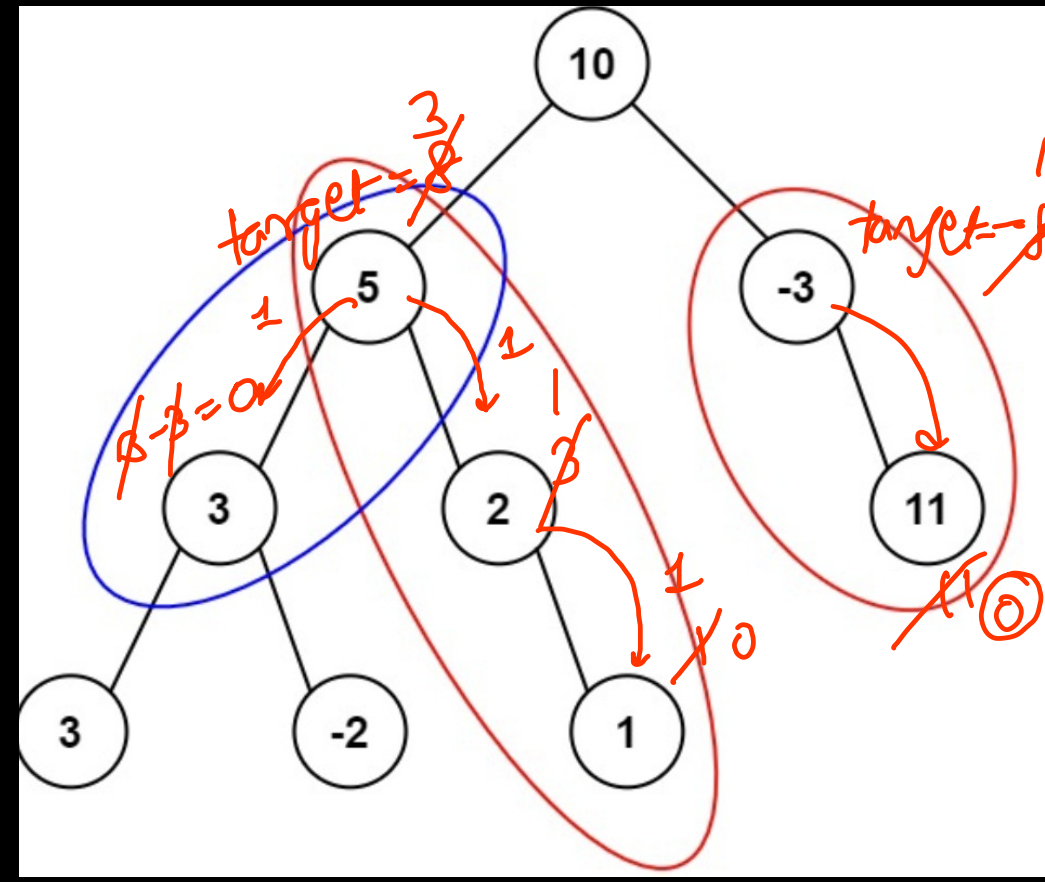
{ 10 → 40, 20 → 30, 35 → 15, 50 }

→ need not start from root

→ need not end at leaf

→ must go downward.

```java
// downward path must start from root
public int dfs(TreeNode root, long targetSum){
    if(root == null) return 0;

    targetSum -= root.val;
    int path = (targetSum == 0) ? 1 : 0;

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

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



targetSum = 8

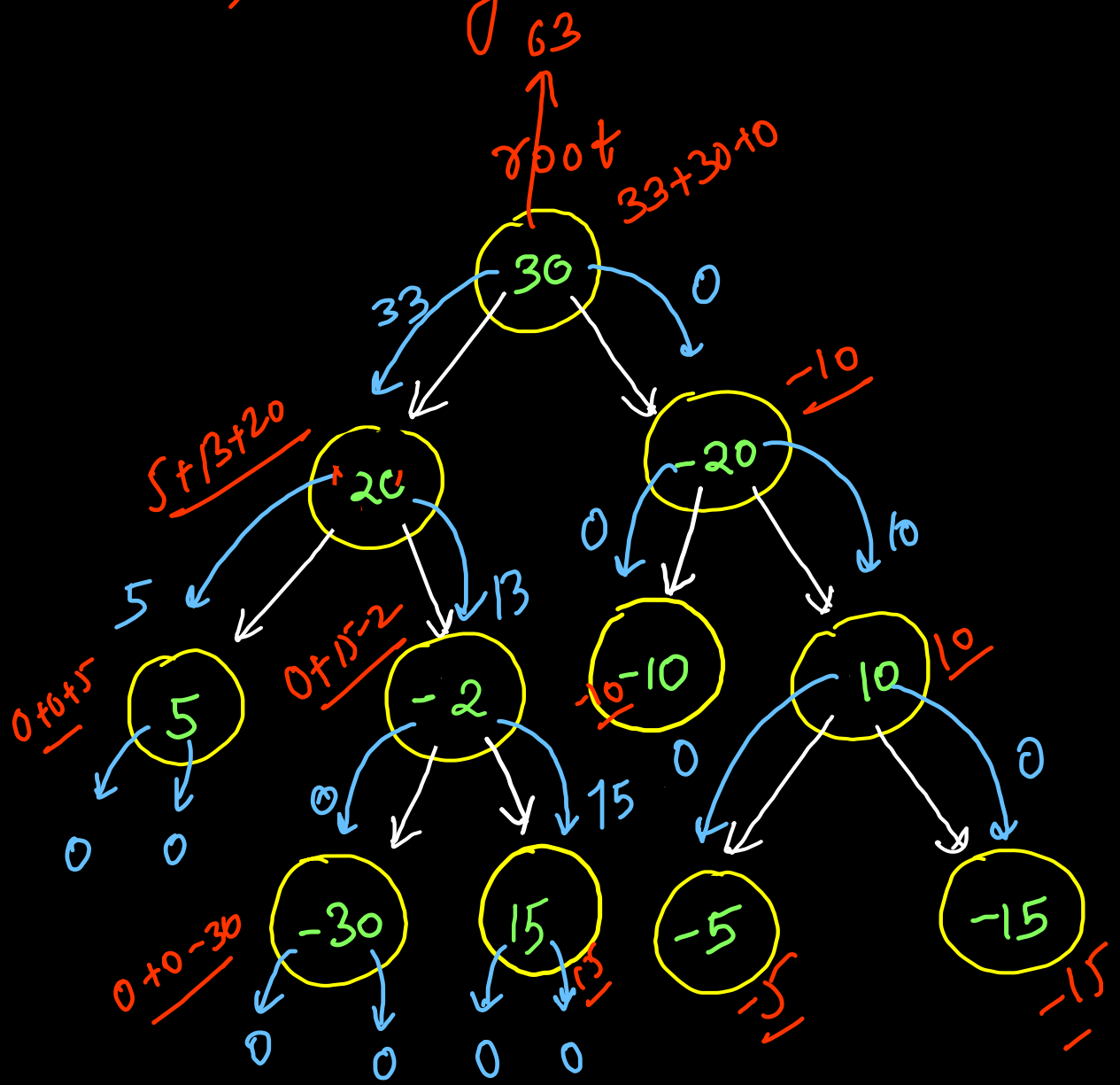
Time ⇒ $O(n^2)$, Space ⇒ $O(h)$

dfs(10) = 0

dfs(5) = 2

dfs(-3) = 1

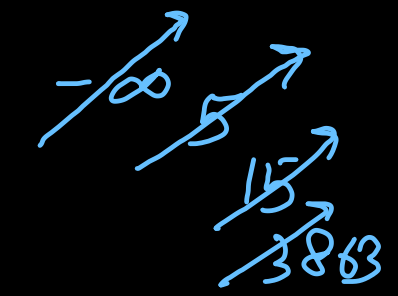
LC 124) Binary Tree Max Path Sum



Path Max^m path Sum

1. need not start from root
2. need not end at leaf
3. need not be downward
4. must be non-empty!

diameter = $-\infty$ (global)



leftsum + rightsum + rootval

height = max(leftsum, rightsum) + rootval;

```java
long diameter = Long.MIN_VALUE;
public long height(TreeNode root){
    if(root == null) return 0;

    long lsum = Math.max(height(root.left), 0);
    long rsum = Math.max(height(root.right), 0);

    diameter = Math.max(diameter, lsum + rsum + root.val);
    return Math.max(lsum, rsum) + root.val;
}

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

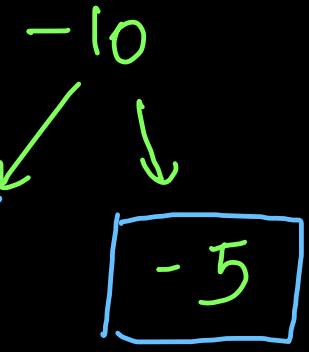
→ non-empty path

→ to discard negative sum (path need not end at leaf)

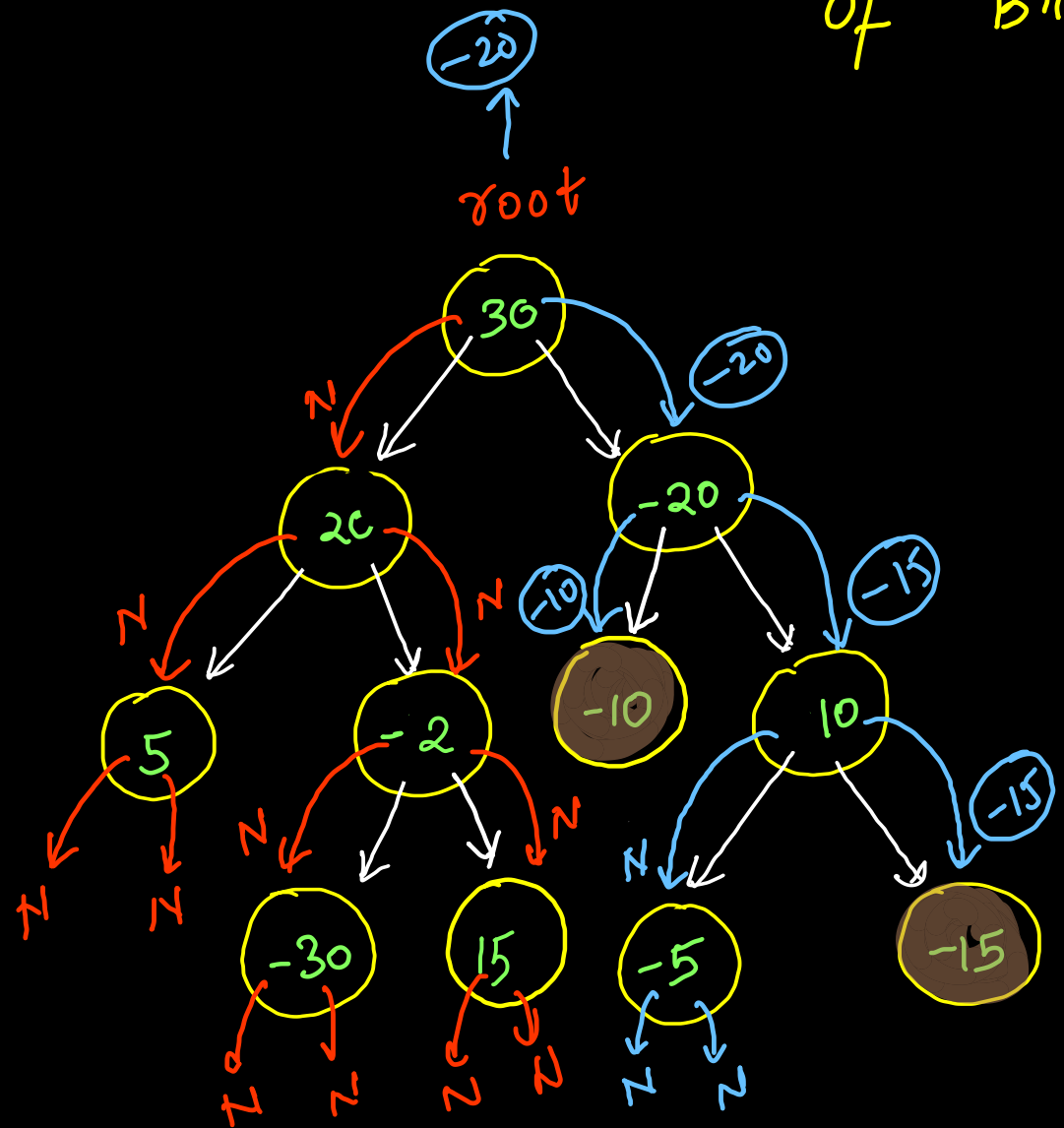
Time = O(n)

Space = O(h)

Corner case: All nodes are negative



Lowest common ancestor of Binary tree



LCA(-2, 5) = 20

-2 ⇒ -2, 20, 30

+5 ⇒ 5, 20, 30

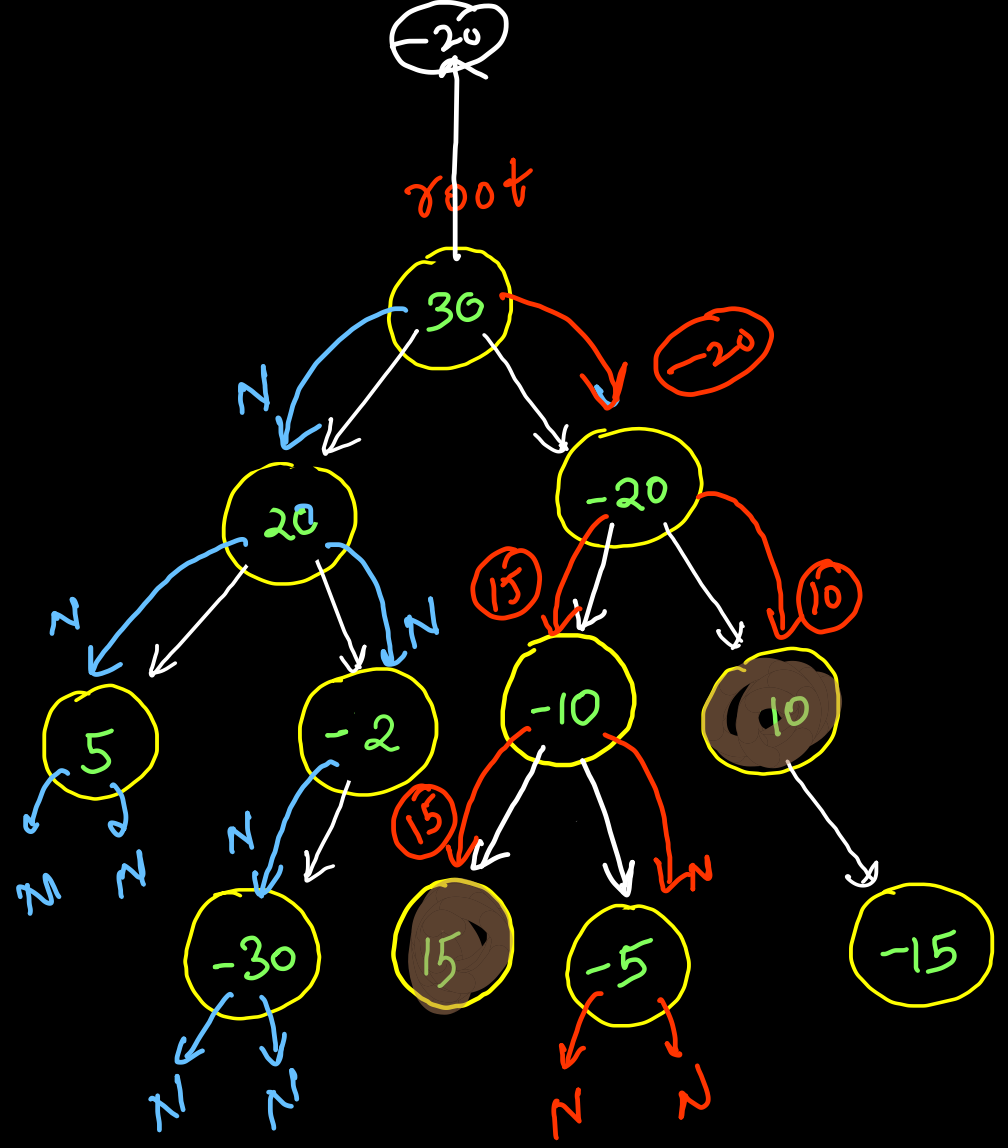
LCA(-10, -15) ⇒ -20

P Q

-10 ⇒ -10, -20, 30

-15 ⇒ -15, 10, -20, 30

LCA(20, -30) ⇒ 20



p = 15 , q = 10

```
if (root == p || root == q || root == N)
    return root;
```

left → N , right → N

⇒ return null;

left != N, right == N

⇒ return left

left == N, right != N

⇒ return right

left != N, right != N

⇒ return root.

```java
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
    if(root == null || root == p || root == q) return root;

    TreeNode lans = lowestCommonAncestor(root.left, p, q);
    TreeNode rans = lowestCommonAncestor(root.right, p, q);

    if(lans != null && rans != null) return root;
    return (lans != null) ? lans : rans;
}
```

Time
↳ O(n)

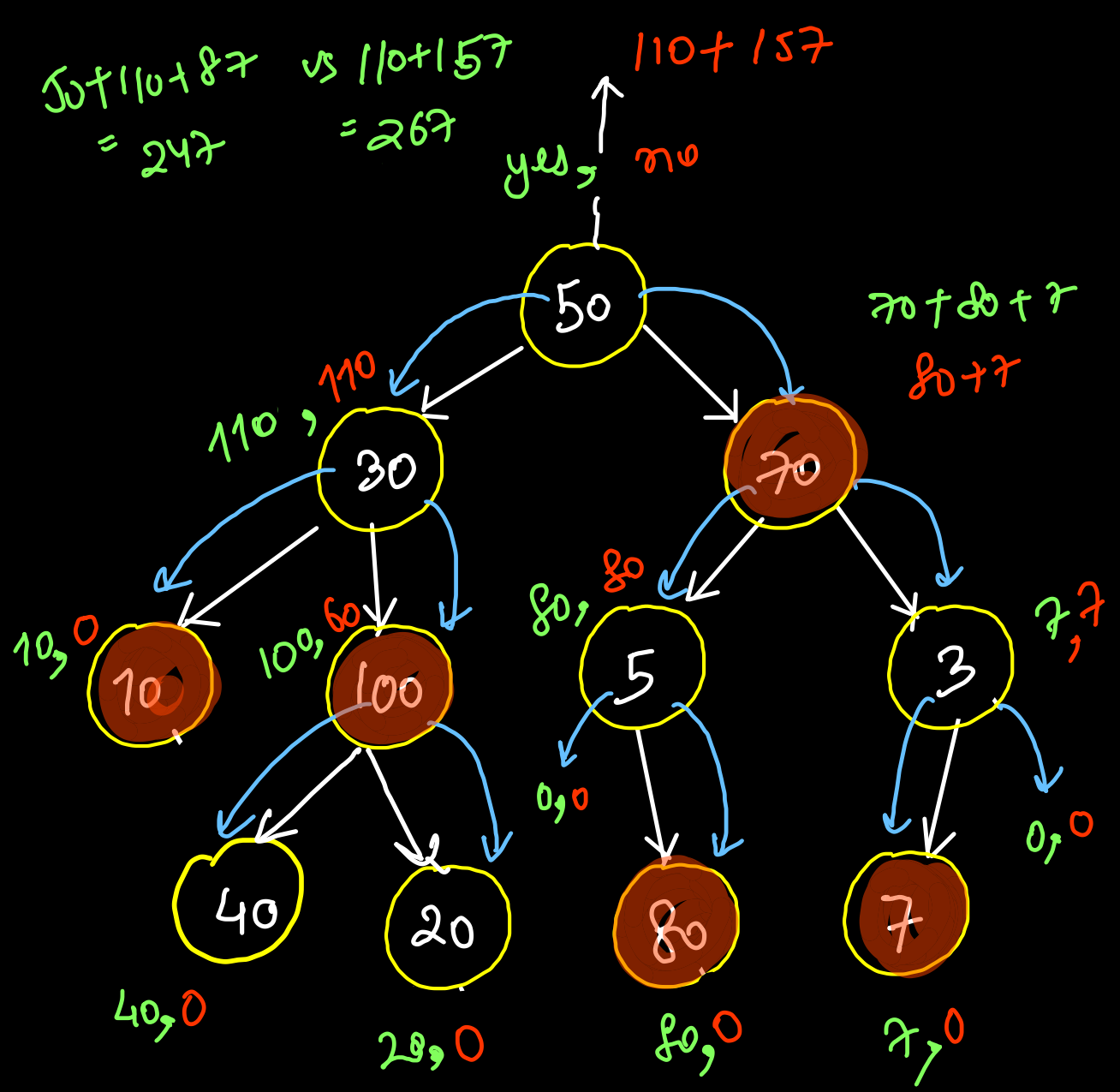
Space
↳ O(h)

This code will only work if both nodes exist or both nodes does not exist.

but it will fail if only one node out of p & q exist.

(DP on tree)

LC 337) House robber - III



max^m amount of money w/o breaking into 2 consecutive houses

root.no = left.yes + right.yes

root.yes = root.no, root.val + left.no + right.no

↳ we can skip 2 contiguous houses.

Approach 1)

```java
public static class Pair{ int yes, no; }

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

    Pair l = dfs(root.left);
    Pair r = dfs(root.right);

    Pair curr = new Pair();
    curr.no = l.yes + r.yes;
    curr.yes = Math.max(curr.no, root.val + l.no + r.no);
    return curr;

}

public int rob(TreeNode root) {
    return dfs(root).yes;
}
```

⇒ yes is not Compulsory
{ root ko le bhi sakte hai
ya chod bhi sakte
hai }

Approach 2)

```java
public static class Pair{ int yes, no; }

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

    Pair l = dfs(root.left);
    Pair r = dfs(root.right);

    Pair curr = new Pair();
    curr.yes = root.val + l.no + r.no;
    curr.no = Math.max(l.yes, l.no) + Math.max(r.yes, r.no);
    return curr;

}

public int rob(TreeNode root) {
    Pair p = dfs(root);
    return Math.max(p.yes, p.no);
}
```

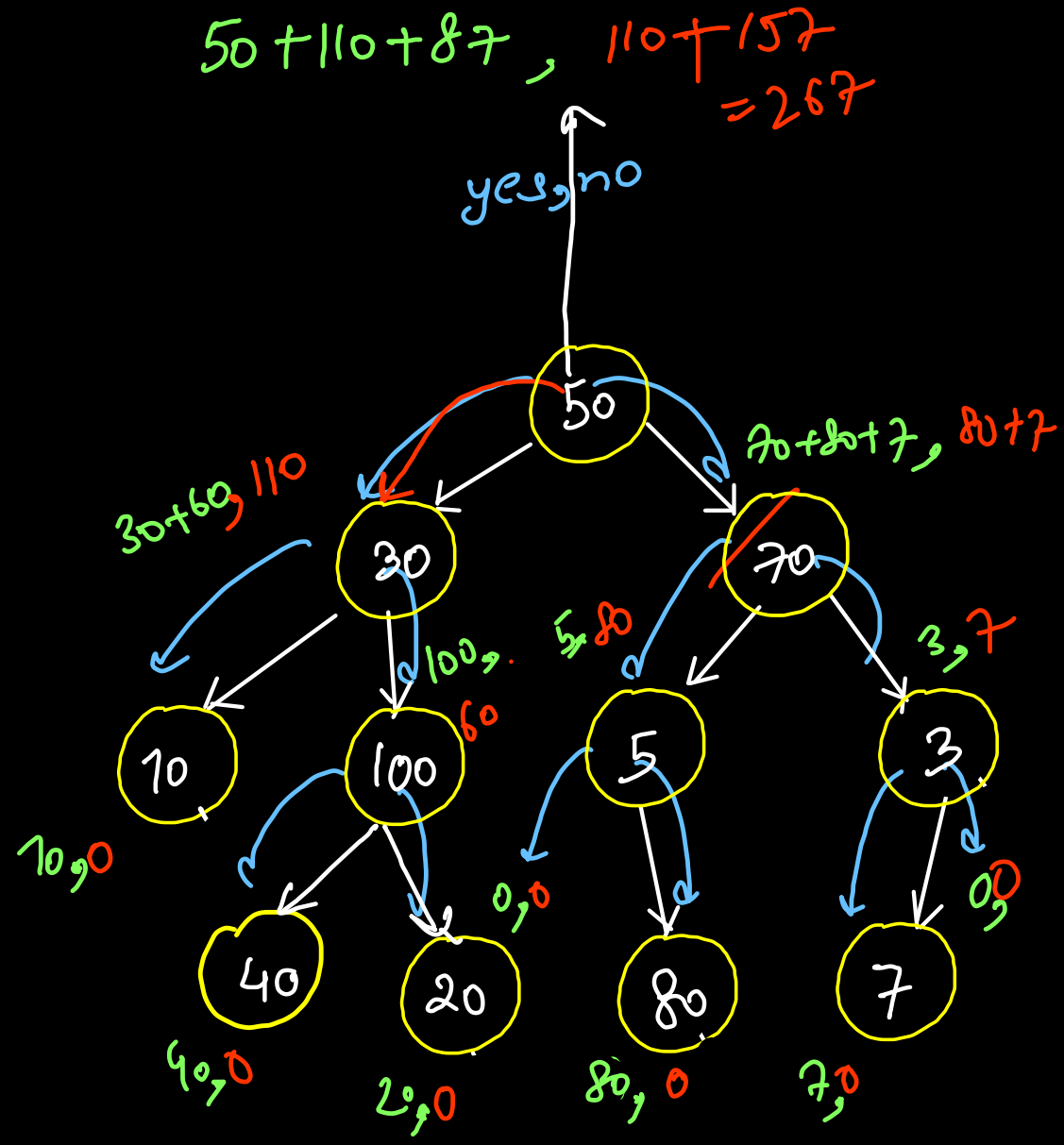
⇒ yes is compulsory
{ root ko lena hi
lena hai }

```java
public static class Pair{ int yes, no; }

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

    Pair l = dfs(root.left);
    Pair r = dfs(root.right);

    Pair curr = new Pair();
    curr.yes = root.val + l.no + r.no;
    curr.no = Math.max(l.yes, l.no) + Math.max(r.yes, r.no);
    return curr;

}

public int rob(TreeNode root) {
    Pair p = dfs(root);
    return Math.max(p.yes, p.no);
}
```

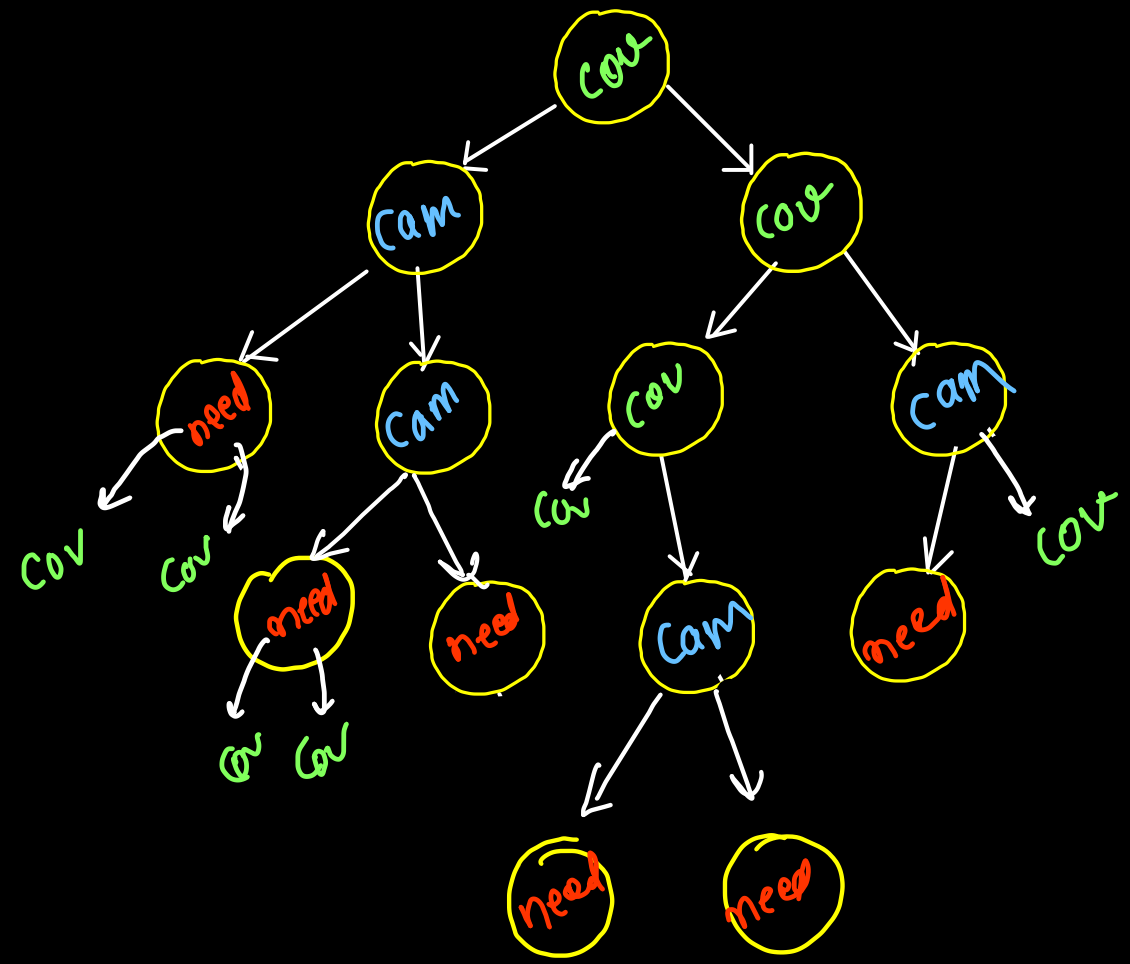
Dfs

Time ⇒ O(n)

Space ⇒ O(h)



LC 968) Binary Tree Cameras



greedy soln

- CAMERA
- COVERED
- NEED { parent }

Camera = 4

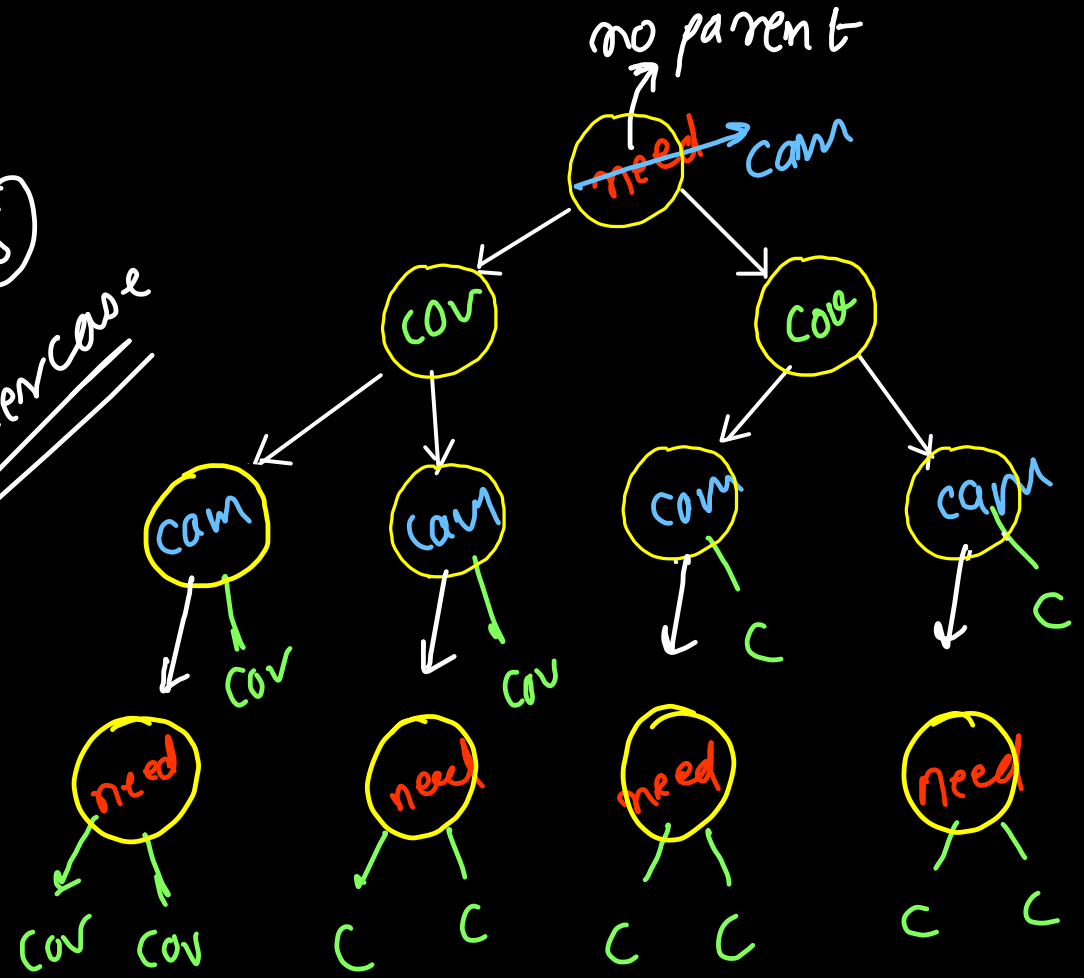
```java
public static enum State{
    COVERED, CAMERA, NEED;
}

int cameras = 0;

public State dfs(TreeNode root){
    if(root == null) return State.COVERED;

    State ls = dfs(root.left);
    State rs = dfs(root.right);

    if(ls == State.NEED || rs == State.NEED){
        cameras++;
        return State.CAMERA;
    }

    if(ls == State.COVERED && rs == State.COVERED){
        return State.NEED;
    }

    return State.COVERED;
}
```

cammy (1) (5)

Cornercase

anyone child have need ⇒ parent will fulfill it

↳ greedily (parent will install camera)

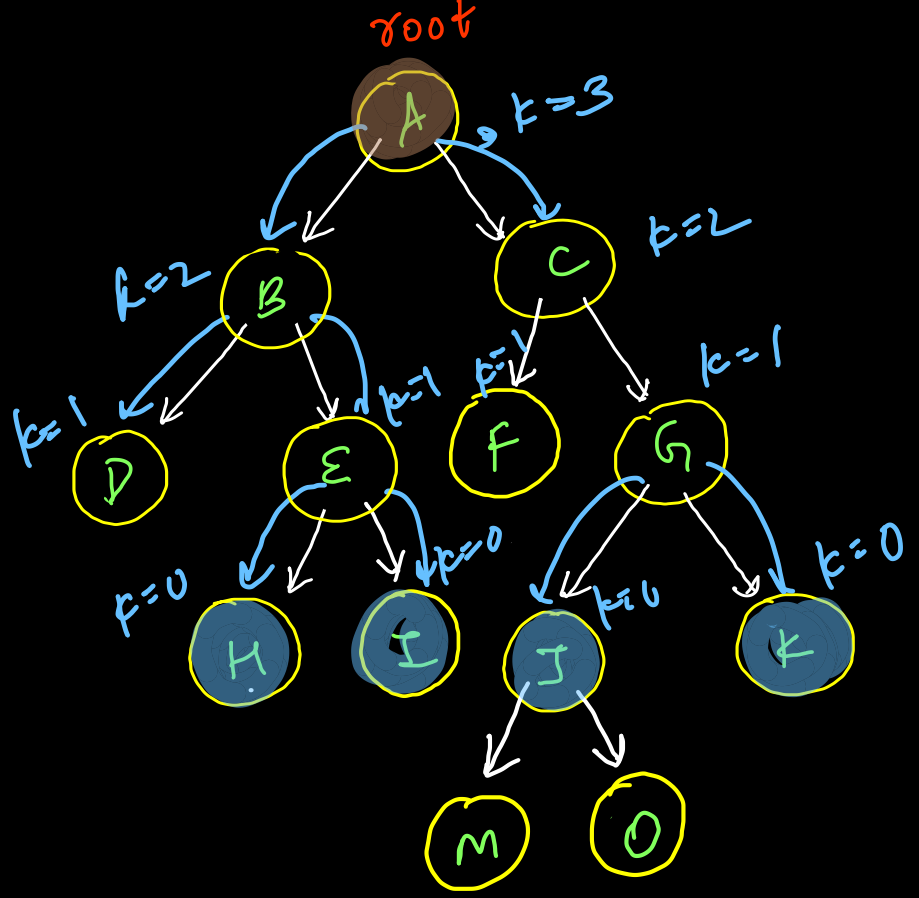



```java
public int minCameraCover(TreeNode root) {
    if(dfs(root) == State.NEED)
        cameras++;
    return cameras;
}
```

↳ if root has need ⇒ it can be fulfilled by root only

All nodes at K distance

target = A , k = 2



target = G

k = 0 :→ G

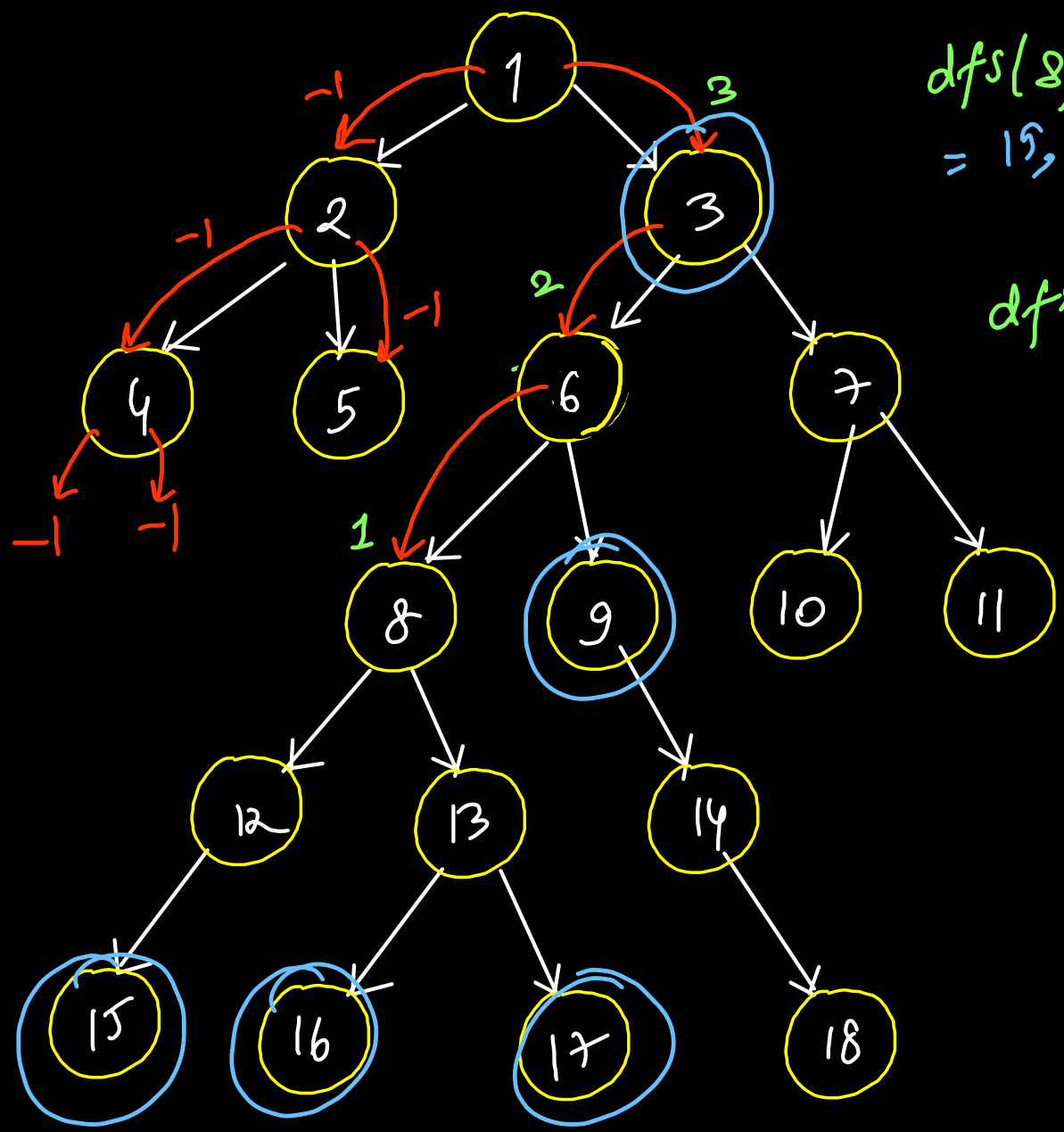
k = 1 :→ C, J, K

k = 2 :→ A, F, M, D

k = 3 :→ B

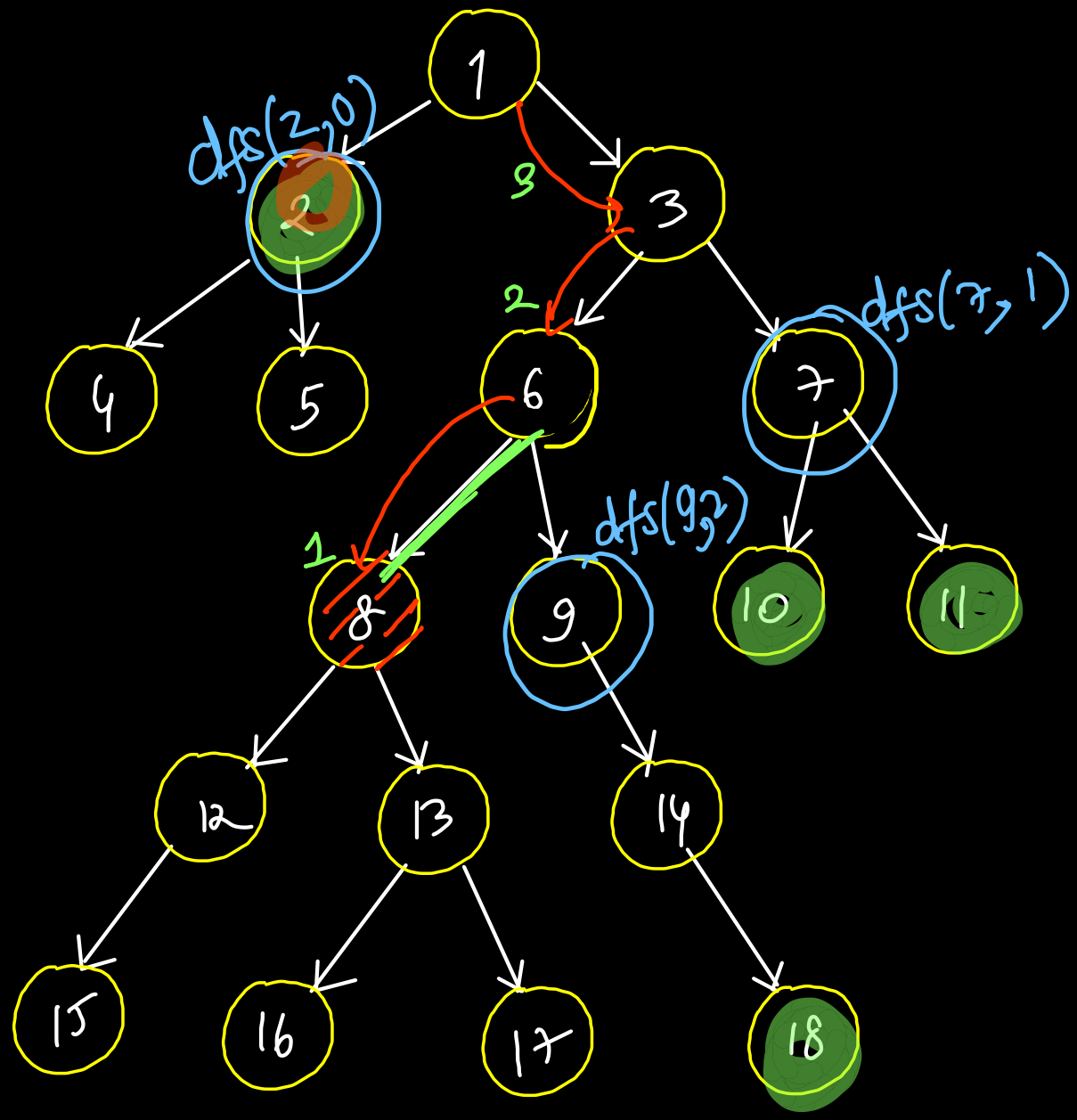
k = 4 :→ D, E

k = 5 :→ H, I




```java
// if root is the target node
public void dfs(TreeNode root, int k){
    if(k < 0 || root == null) return;
    if(k == 0) ans.add(root.val);
    dfs(root.left, k - 1);
    dfs(root.right, k - 1);
}
```

```java
public int find(TreeNode root, TreeNode target, int k){
    if(root == null) return -1;
    if(root == target){
        dfs(root, k);
        return 1;
    }

    int left = find(root, target, k);
    int right = find(root, target, k);
```

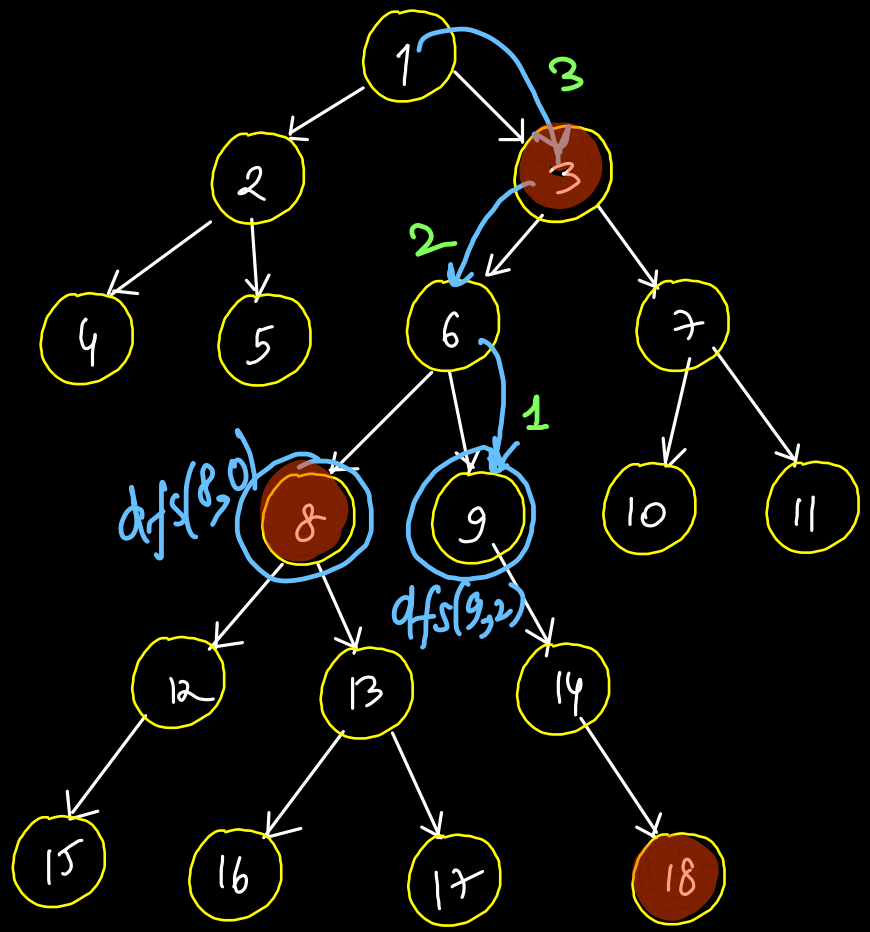

target = 8, k = 4
1
dfs(2,0)
2
3
9
2
4
5
6
dfs(7,1)
7
1
dfs(9,2)
8
9
10
11
12
13
14
15
16
17
18

```java
public int find(TreeNode root, TreeNode target, int k){
    if(root == null) return -1;
    if(root == target){
        dfs(root, k);
        return 1; //dist of target to parent
    }

    int left = find(root.left, target, k);
    if(left > 0) {
        dfs(root.right, k - left - 1);
        if(left == k) ans.add(root.val);
        return 1 + left;
    }

    int right = find(root.right, target, k);
    if(right > 0) {
        dfs(root.left, k - right - 1);
        if(right == k) ans.add(root.val);
        return 1 + right;
    }

    return -1;
}
```

9 k=2

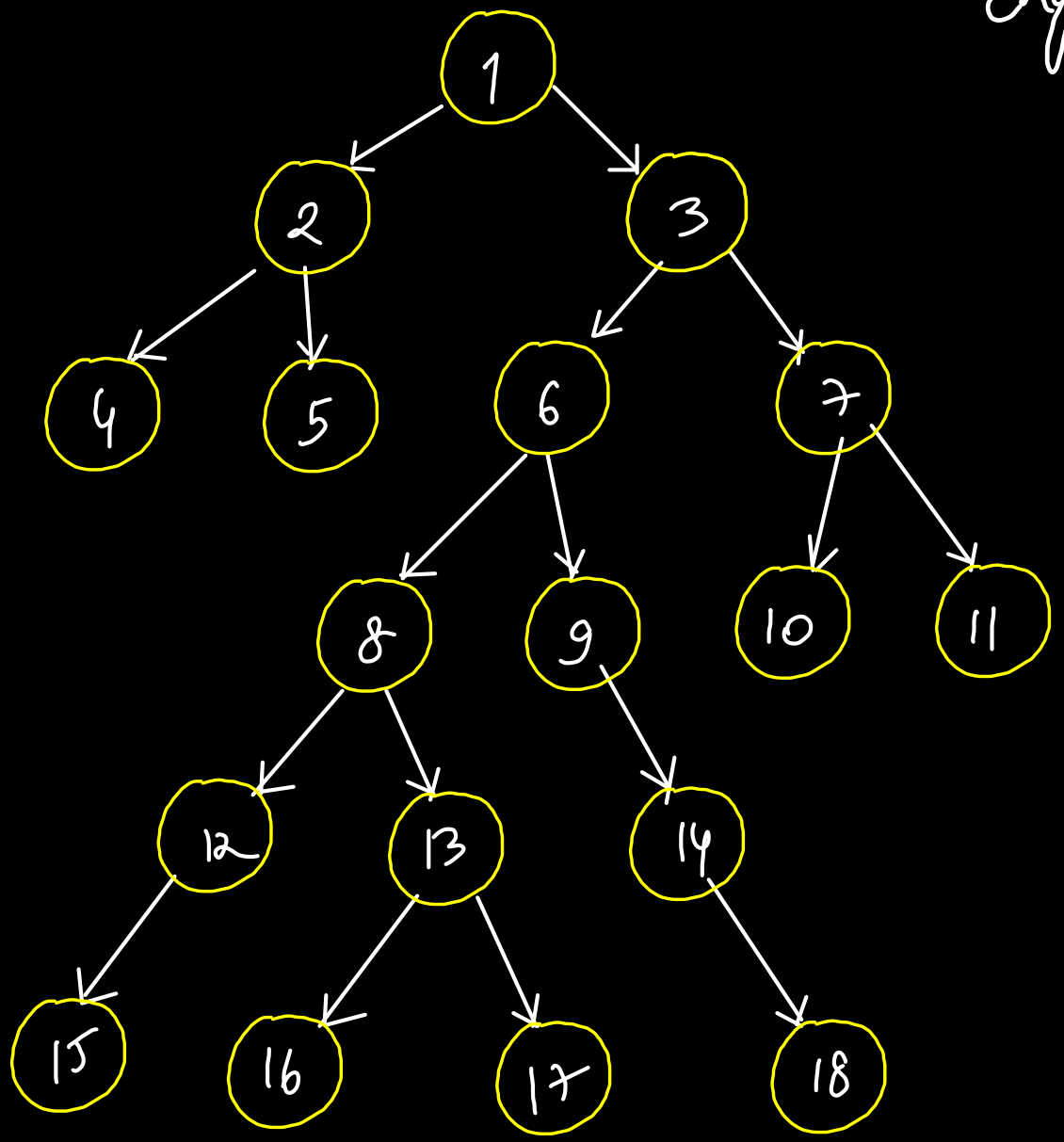
Time ⇒ linear

Space ⇒ linear




```java
public List<Integer> distanceK(TreeNode root, TreeNode target, int k) {
    find(root, target, k);
    return ans;
}
```

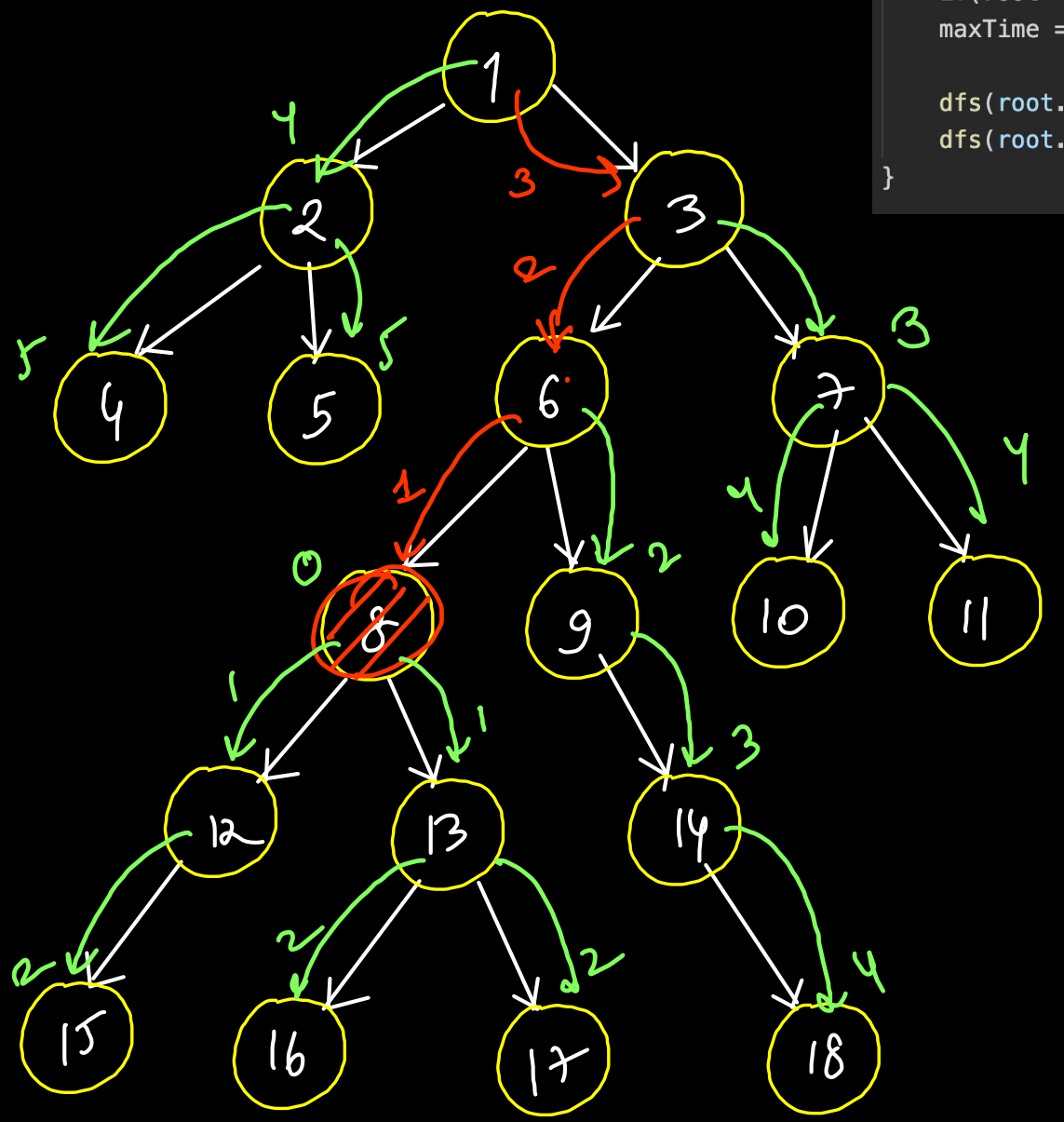
```java
List<Integer> ans = new ArrayList<>();

// if root is the target node
public void dfs(TreeNode root, int k){
    if(k < 0 || root == null) return;
    if(k == 0) ans.add(root.val);
    dfs(root.left, k - 1);
    dfs(root.right, k - 1);
}
```



Engifest → 17 → Sunidhi Chaudhary
18 → Amit Trivedi
19 → AP dhillon

Accepted Apr 20, 2020	C++
Wrong Answer Apr 20, 2020	C++
Wrong Answer Apr 20, 2020	C++
Wrong Answer Apr 20, 2020	C++
Wrong Answer Apr 20, 2020	C++
Wrong Answer Apr 20, 2020	C++
Wrong Answer Apr 20, 2020	C++
Wrong Answer Apr 20, 2020	C++

maxtime = 0 1 2 3 4 5




```
int maxTime = 0;
                    //subtree to get infected
public void dfs(TreeNode root, int time){
    if(root == null) return;
    maxTime = Math.max(maxTime, time);

    dfs(root.left, time + 1);
    dfs(root.right, time + 1);
}
```

Time → linear
Space → linear

Burning Tree

```
public int find(TreeNode root, int target){
    if(root == null) return -1;
    if(root.val == target){
        dfs(root, 0);
        return 1;
    }

    int left = find(root.left, target);
    if(left > 0) {
        maxTime = Math.max(maxTime, left);
        dfs(root.right, left + 1);
        return 1 + left;
    }

    int right = find(root.right, target);
    if(right > 0) {
        maxTime = Math.max(maxTime, right);
        dfs(root.left, right + 1);
        return 1 + right;
    }

    return -1;
}
```

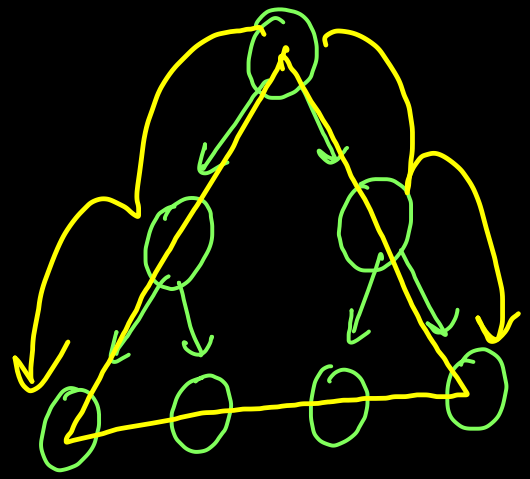
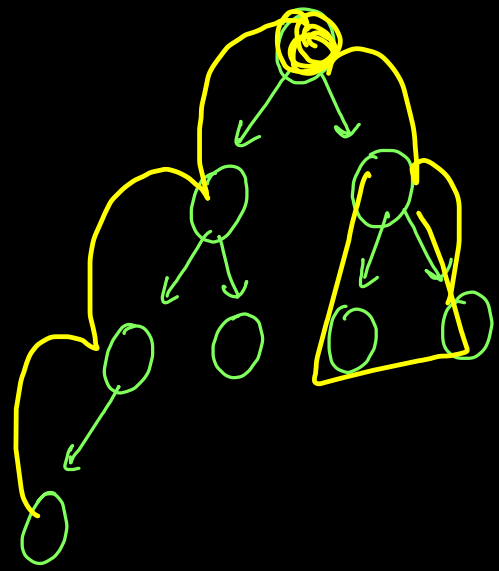
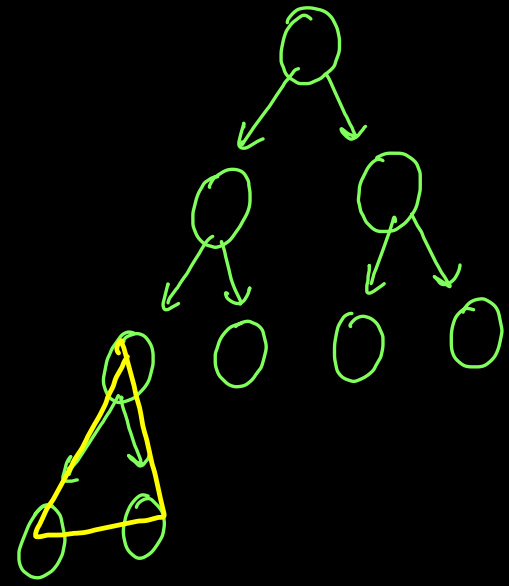
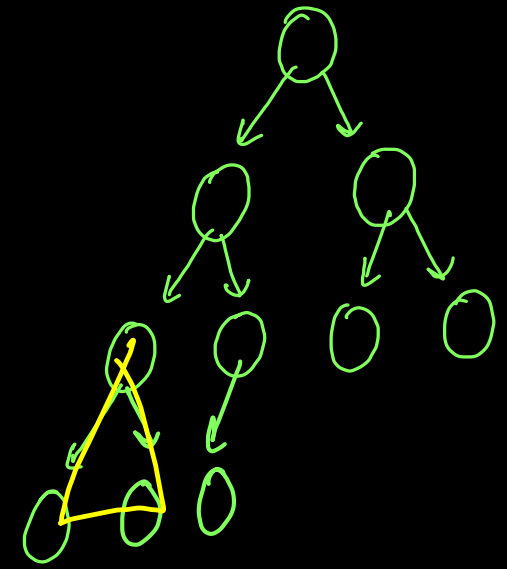
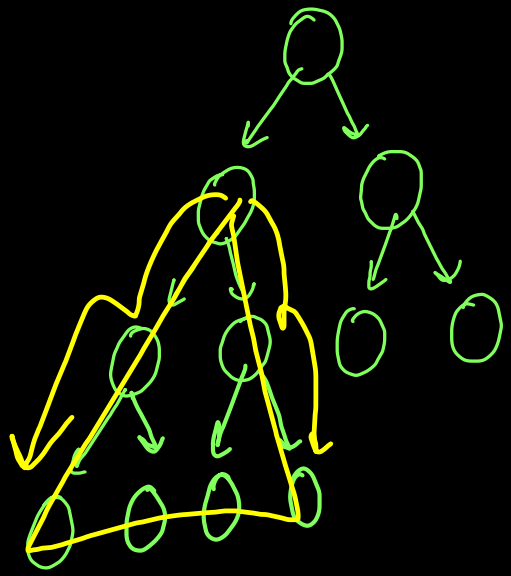
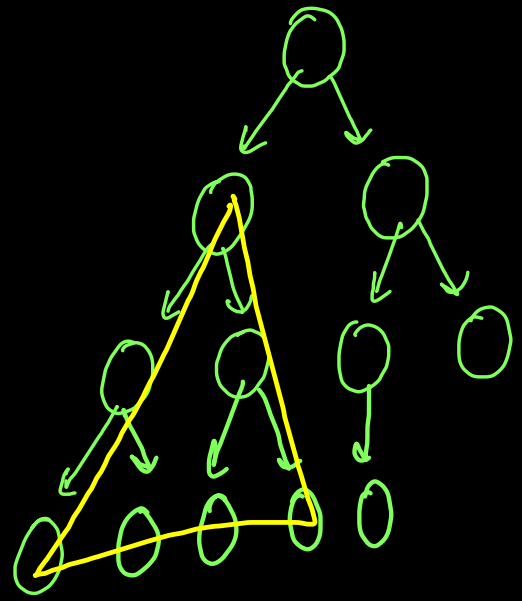
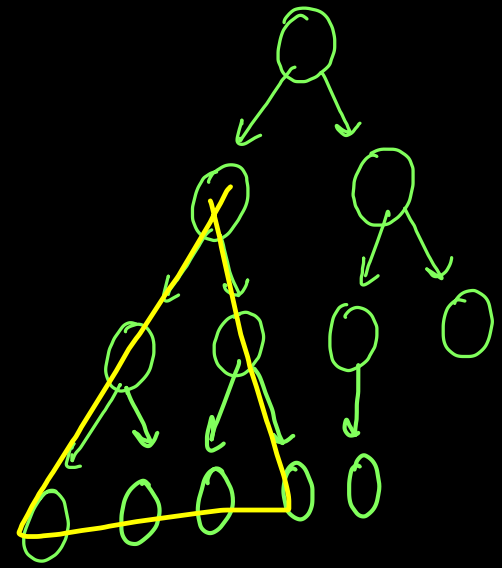
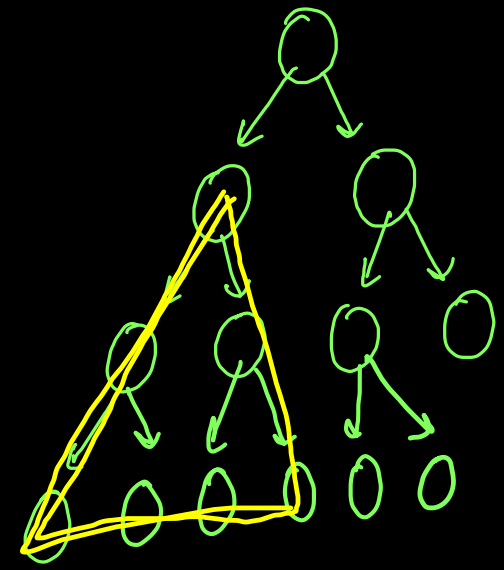
```
public int amountOfTime(TreeNode root, int target) {
    find(root, target);
    return maxTime;
}
```

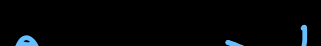
Complete Binary Tree → ordering insertion will be top to bottom left to right

↳ all levels (except the last) are complete

& last level nodes are as left as possible

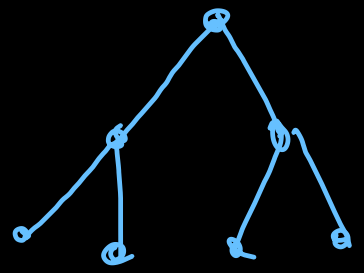

```java
// approach 1: dfs -> O(n) time, O(log n) space
public int size(TreeNode root){
    if(root == null) return 0;
    return 1 + size(root.left) + size(root.right);
}
```

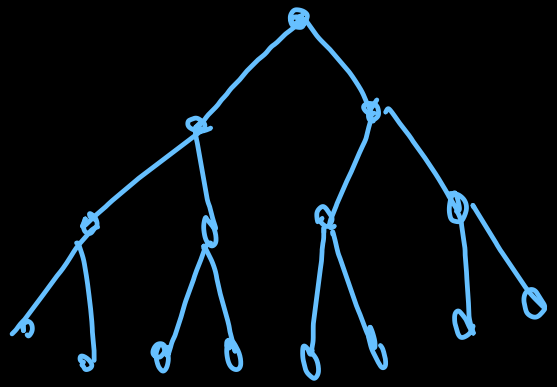


$\rightarrow 1$

$\rightarrow 1+2 = 3$



$\rightarrow 1+2+2^2 = 7$



$\rightarrow 1+2+2^2+2^3$

$= 15$

height = h { Perfect binary tree }

$= 2^0 + 2^1 + 2^2 + \cdots 2^{h-1}$

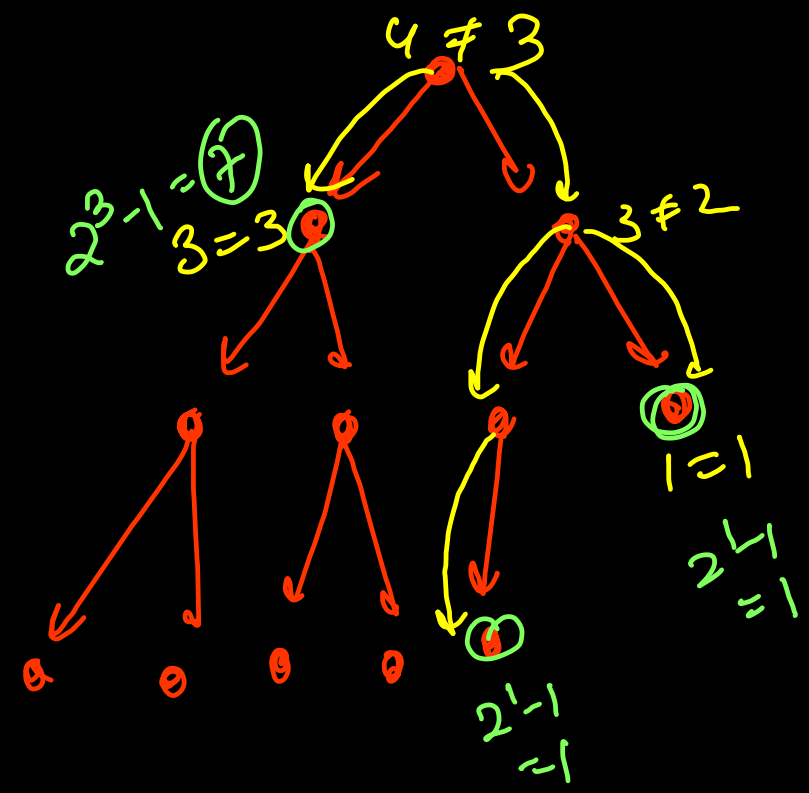
$= 2^h - 1$ { nodes }

Complete binary tree is always balanced $\Rightarrow h = \log_2 n$

```java
public int leftMost(TreeNode root){
    if(root == null) return 0;
    return 1 + leftMost(root.left);
}

public int rightMost(TreeNode root){
    if(root == null) return 0;
    return 1 + rightMost(root.right);
}

public int countNodes(TreeNode root) {
    if(root == null) return 0;
    int lh = leftMost(root);
    int rh = rightMost(root);

    if(lh == rh) return (1 << lh) - 1;
    // perfect binary tree

    return countNodes(root.left) + countNodes(root.right) + 1;
}
```

O(h)

O(h)



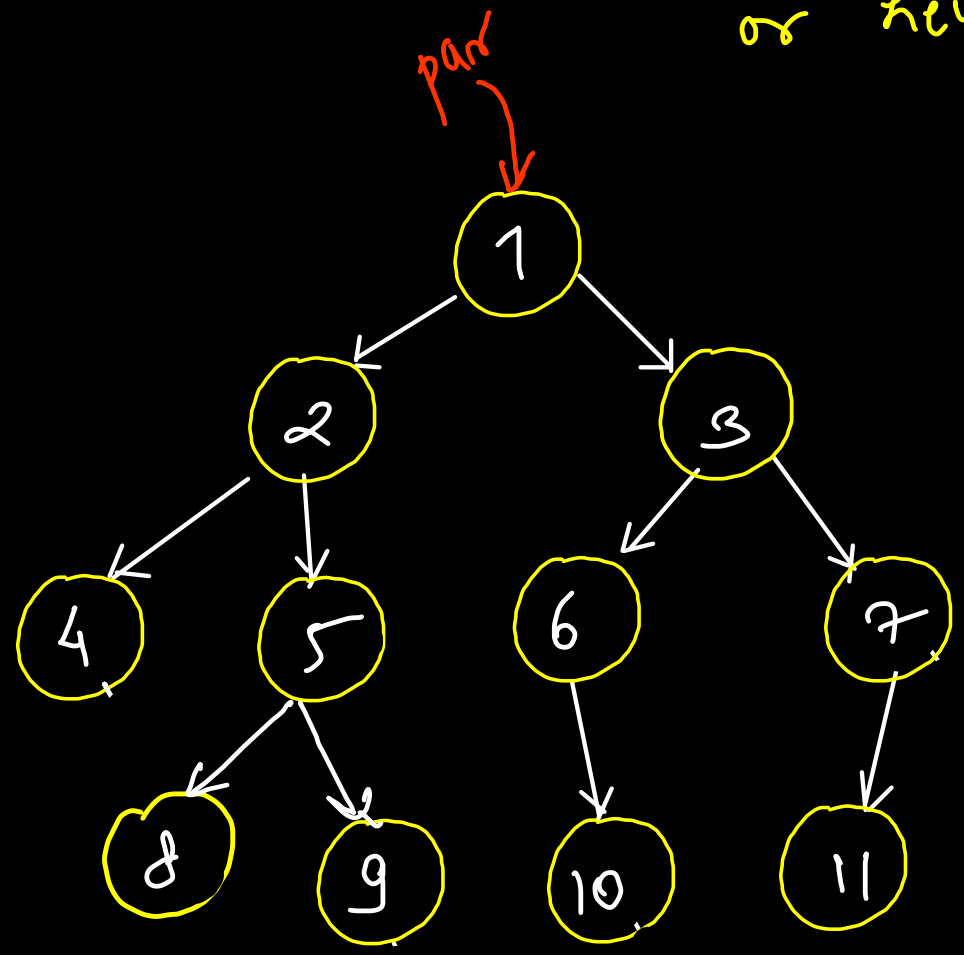
Total time $\Rightarrow O(h^2)$

$\Rightarrow O((\log n)^2)$

$\Rightarrow$

Breadth First Traversal (BFS)

or level order traversal



GFG → top to bottom, left to right

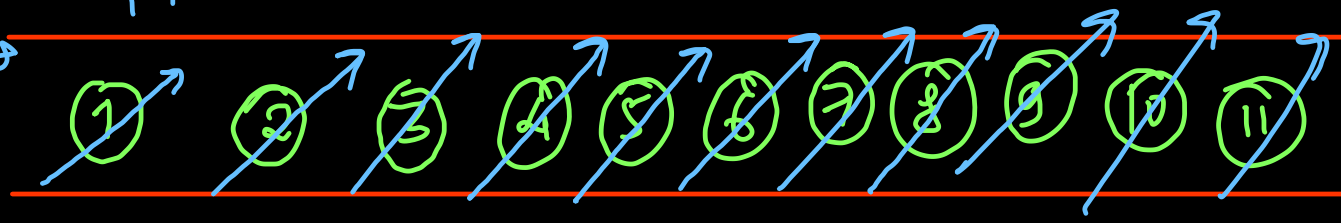
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}

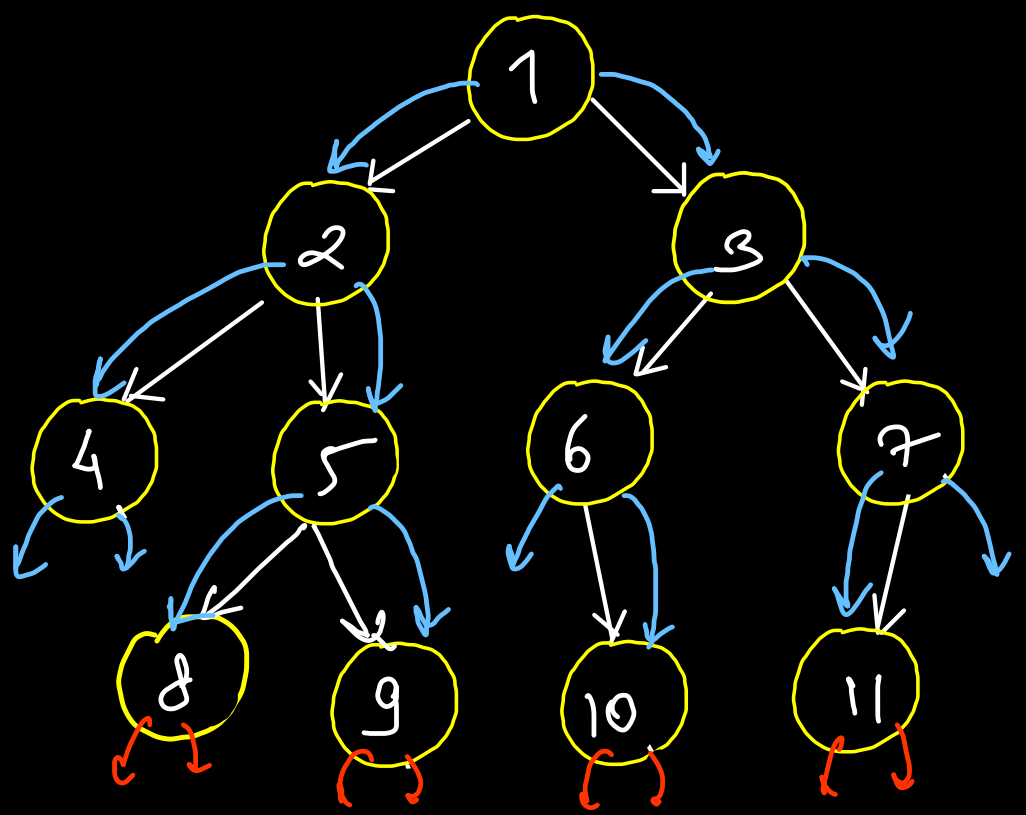
dfs → recursion → stack

bfs → fifo → queue

front (pop)

(push) rear





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

0-1 bfs → queue will atmost contain 2 level

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

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

    while(q.size() > 0) {
        root = q.remove();
        bfs.add(root.data);

        if(root.left != null) q.add(root.left);
        if(root.right != null) q.add(root.right);
    }

    return bfs;
}
```

Time Complexity

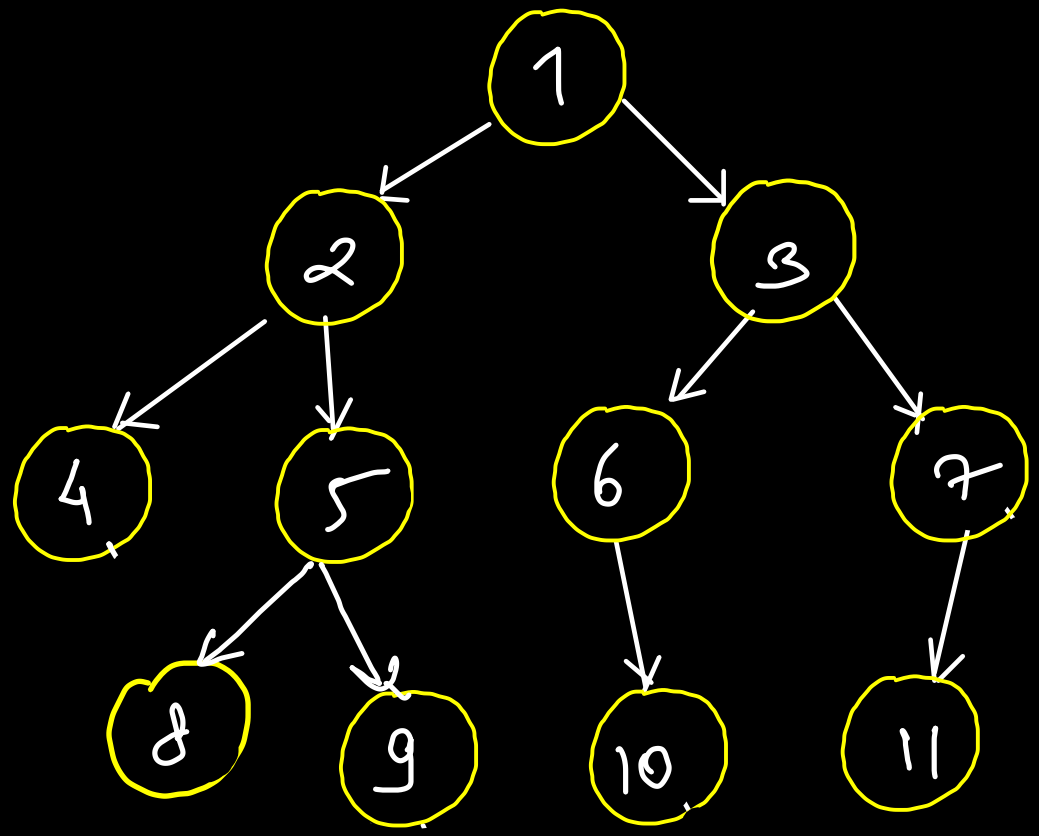
↳ O(n)

Space Complexity

↳ Queue Auxillary Data Structure

↳ O(n)

Leetcode 102 Level order level by level



top to down, left to right, level by level

{{ 1 }

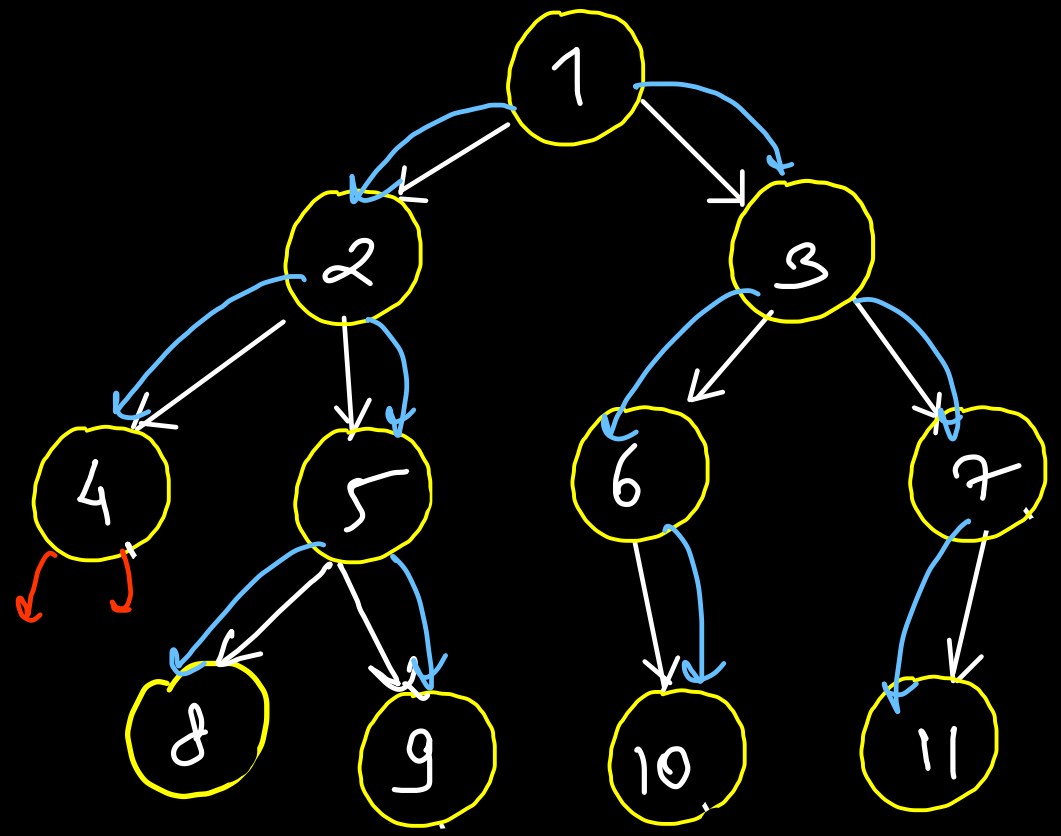
{ 2, 3 }

{ 4, 5, 6, 7 }

{ 8, 9, 10, 11 } }

Approach 1) Queue → node
height/level/depth → Pair

1,0 | 2,1 | 3,1 | 4,2 | 5,2 | 6,2 | 7,2 | 8,3 | 9,3 | 10,3 | 11,3



4k 6k

{ {1}, {2,3},

{4,5,6,7} {8,9,10,11} }

8k 10k

{ 4k, 6k, 8k, 10k }

```java
public static class Pair{
    TreeNode root;
    int level;
    Pair(TreeNode root, int level){
        this.root = root;
        this.level = level;
    }
}
```

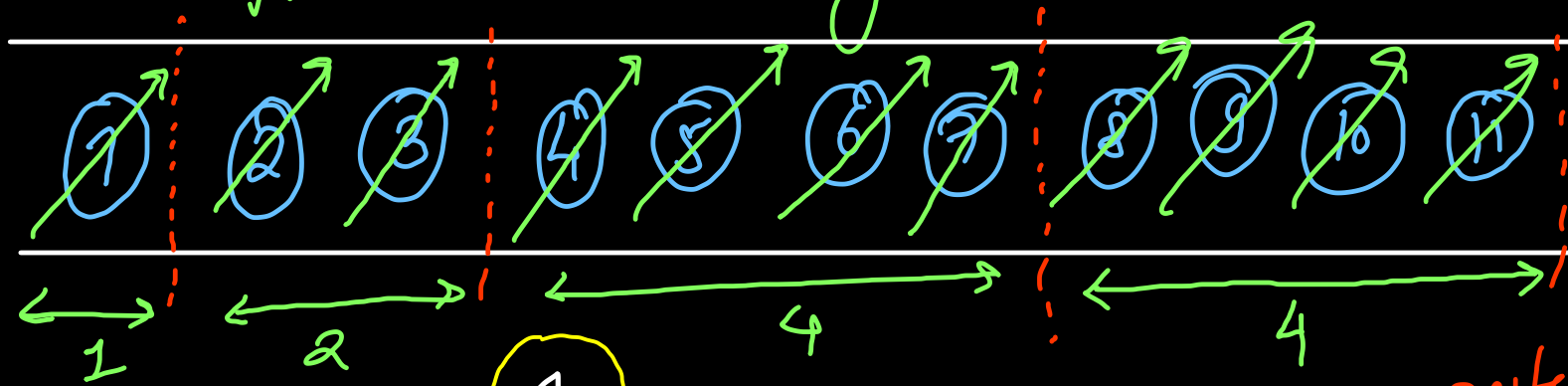
Time ⇒ O(n) } bfs

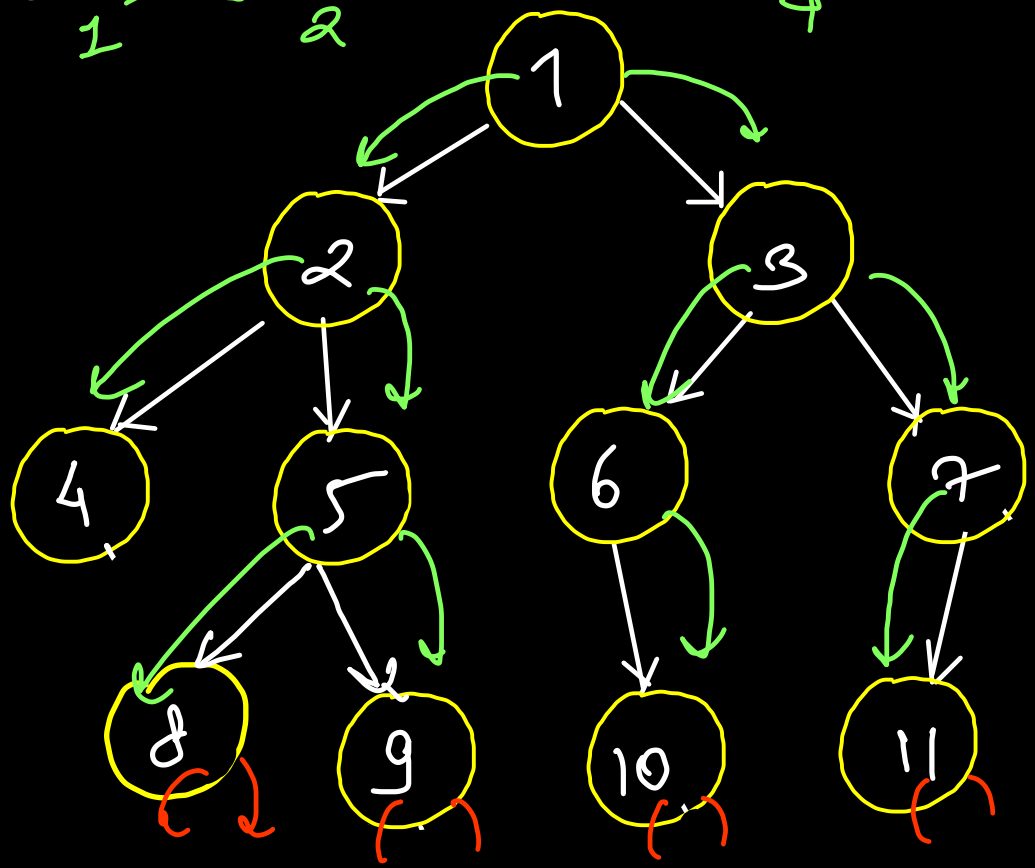
Space ⇒ O(n)

Approach 1) Using pair

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

    Queue<Pair> q = new ArrayDeque<>();
    q.add(new Pair(root, 0));
    int level = -1;

    while(q.size() > 0){
        Pair p = q.remove();

        if(p.level > level){
            levels.add(new ArrayList<>());
            level = p.level;
        }
        levels.get(levels.size() - 1).add(p.root.val);

        if(p.root.left != null)
            q.add(new Pair(p.root.left, p.level + 1));
        if(p.root.right != null)
            q.add(new Pair(p.root.right, p.level + 1));
    }

    return levels;
```

Approach 2) Using count/nested loops





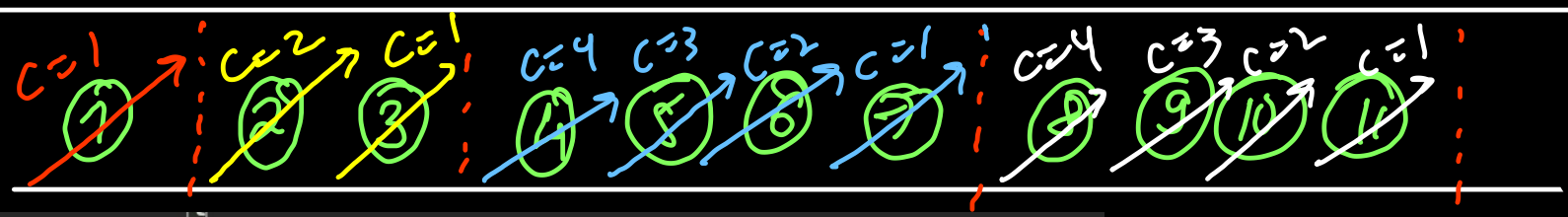
outer loop → all levels

inner loop → current level

{ {1}, {2,3}, {4,5,6,7} }

{8, 9, 10, 11}

for (int c=0; c<size; c++)



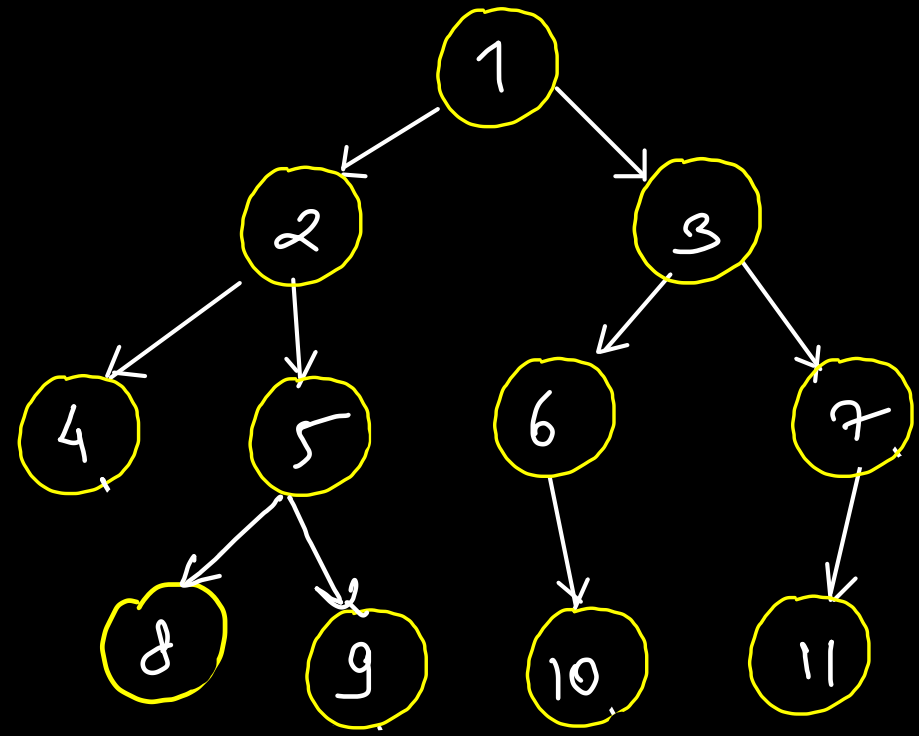
Time ⇒ O(n) linear

Space ⇒ O(n) queue

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

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

    while(q.size() > 0) {
        List<Integer> level = new ArrayList<>();
        for(int c = q.size(); c > 0; c--){
            root = q.remove();
            level.add(root.val);

            if(root.left != null) q.add(root.left);
            if(root.right != null) q.add(root.right);
        }
        levels.add(level);
    }

    return levels;
}
```

→ level by level

↳ current level



{ {1} {2,3} {4,5,6,7} {8,9,10,11} }

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

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

        List<Integer> level = new ArrayList<>();
        while(q.size() > 1 || q.peek() != marker){
            root = q.remove();

            if(root != marker){
                level.add(root.val);
                if(root.left != null) q.add(root.left);
                if(root.right != null) q.add(root.right);
            }
            else {
                q.add(marker);
                levels.add(level);
                level = new ArrayList<>();
            }
        }

        levels.add(level);
        return levels;
    }
}
```

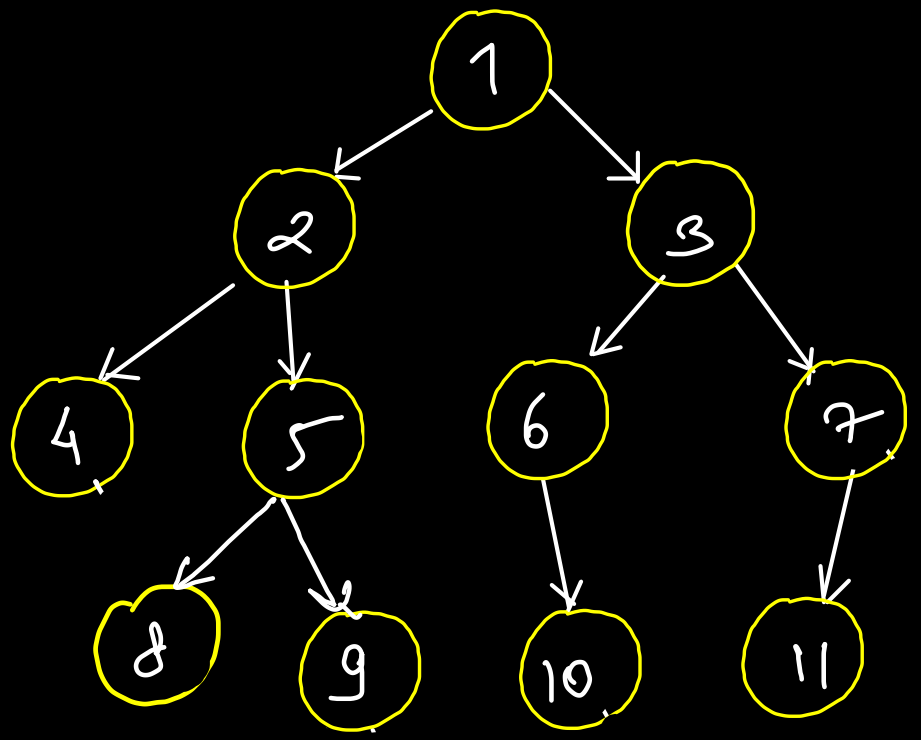
Approach3) Using

marker / dummy / delimiter

Time ⇒ O(n)

Space ⇒ O(n)

LC 107) Level order (Bottom to top)



{ {1}, {2,3}, {4,5,6,7}, {8,9,10,11} }
0 1 2 3

O(h) ↓ Collections.reverse(levels)

{ {8,9,10,11}, {4,5,6,7}, {2,3}, {1} }

Time ⇒ O(n)

Space ⇒ O(n)

LC 103

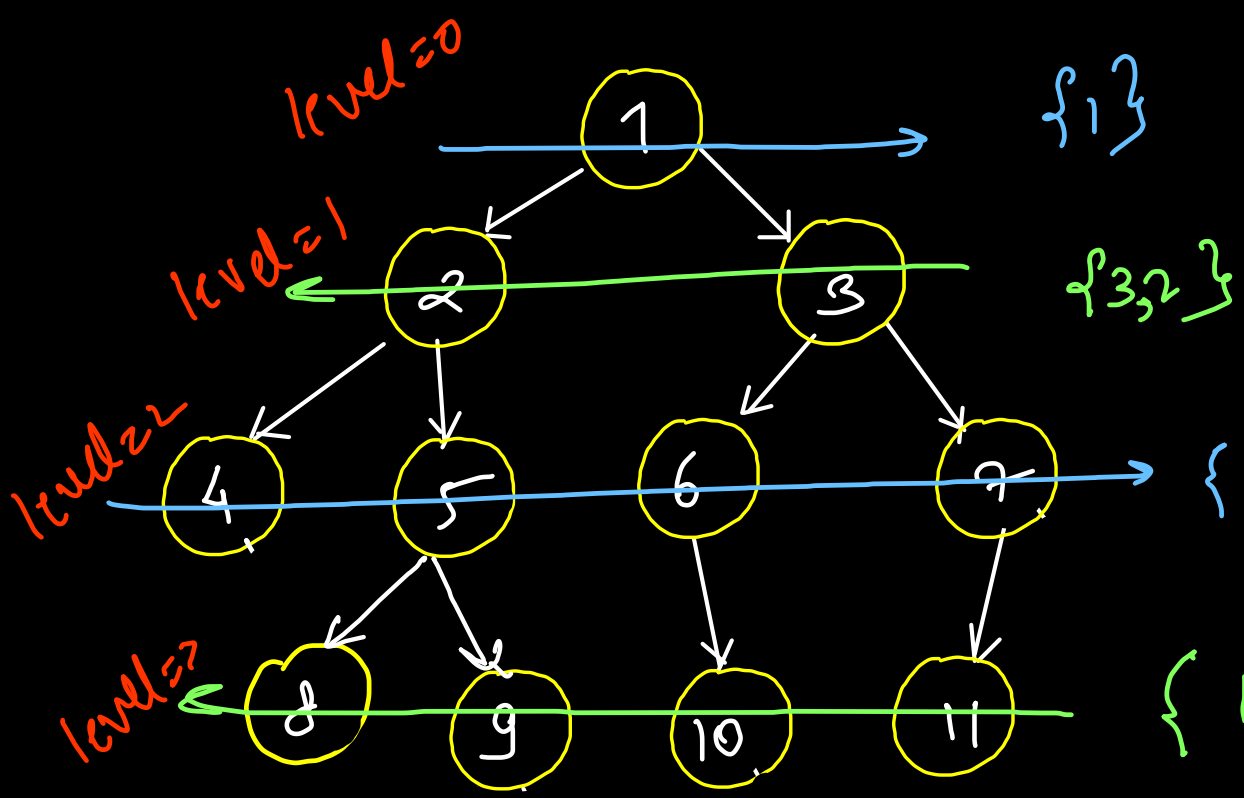
Zig zag

Level order

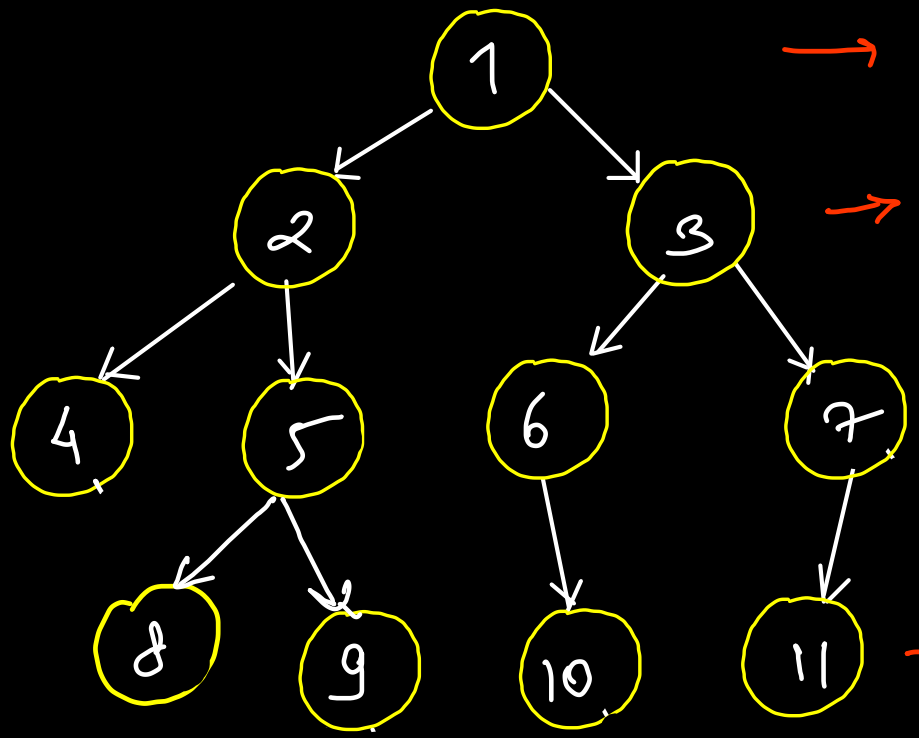
```java
public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
    List<List<Integer>> levels = new ArrayList<>();
    if(root == null) return levels;

    Queue<TreeNode> q = new ArrayDeque<>();
    q.add(root);
    while(q.size() > 0)
    {
        List<Integer> level = new ArrayList<>();
        for(int c = q.size(); c > 0; c--)
        {
            root = q.remove();
            level.add(root.val);

            if(root.left != null) q.add(root.left);
            if(root.right != null) q.add(root.right);
        }
        if(levels.size() % 2 == 1) Collections.reverse(level);
        levels.add(level);
    }

    return levels;
}
```

height of current level

Right to left



LC 637) Average of levels



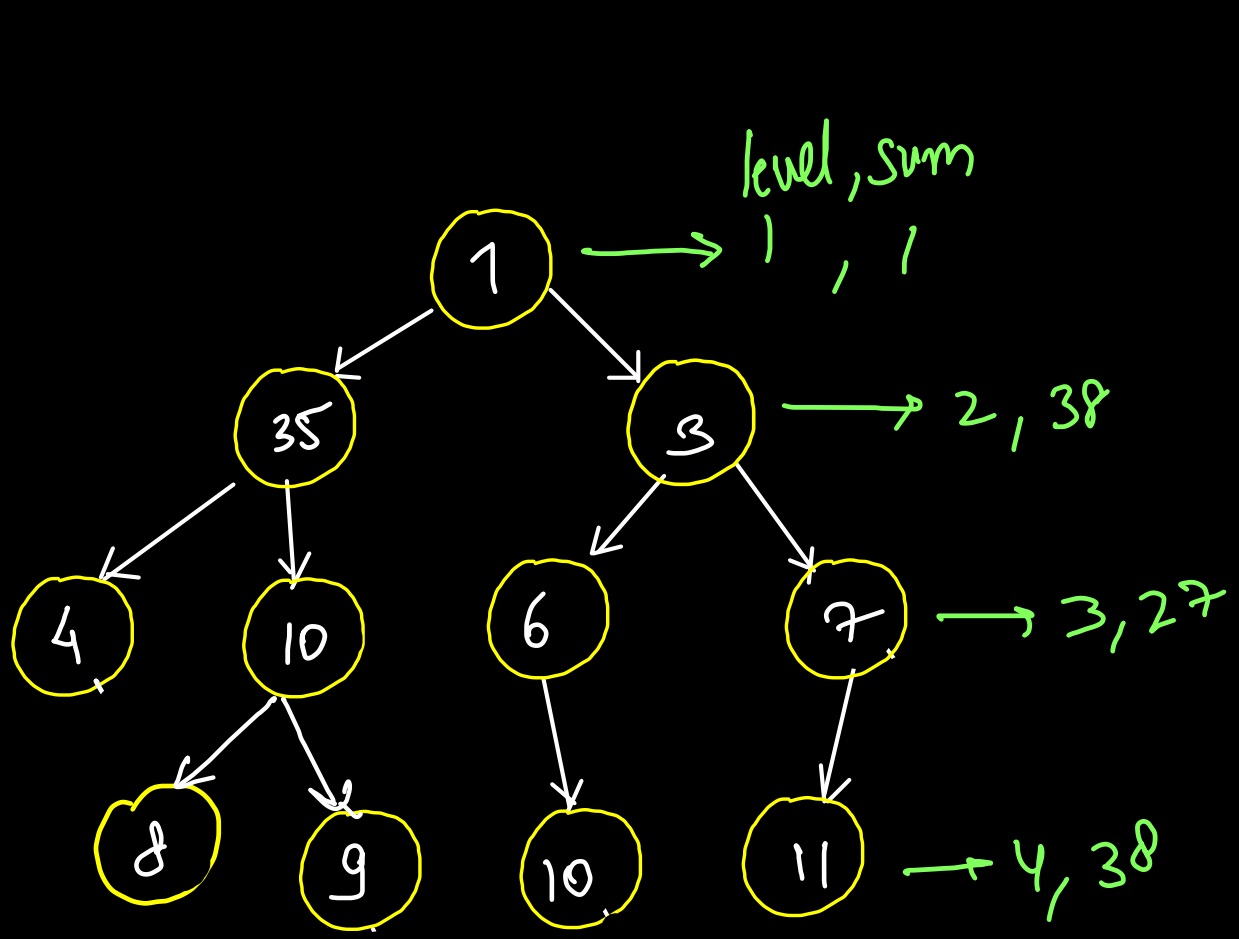
→ 1/1.0 = 1.0

→ (2+3)/2.0 = 2.5

→ (4+5+6+7)/4.0 = 5.5

→ (8+9+10+11)/4.0 = 9.5

LC 1161) Max^m level Sum



maxSum = ~~0~~ / ~~1~~ 38

maxlevel = ~~0~~ / ~~1~~ 2

time = O(n), space = O(n)

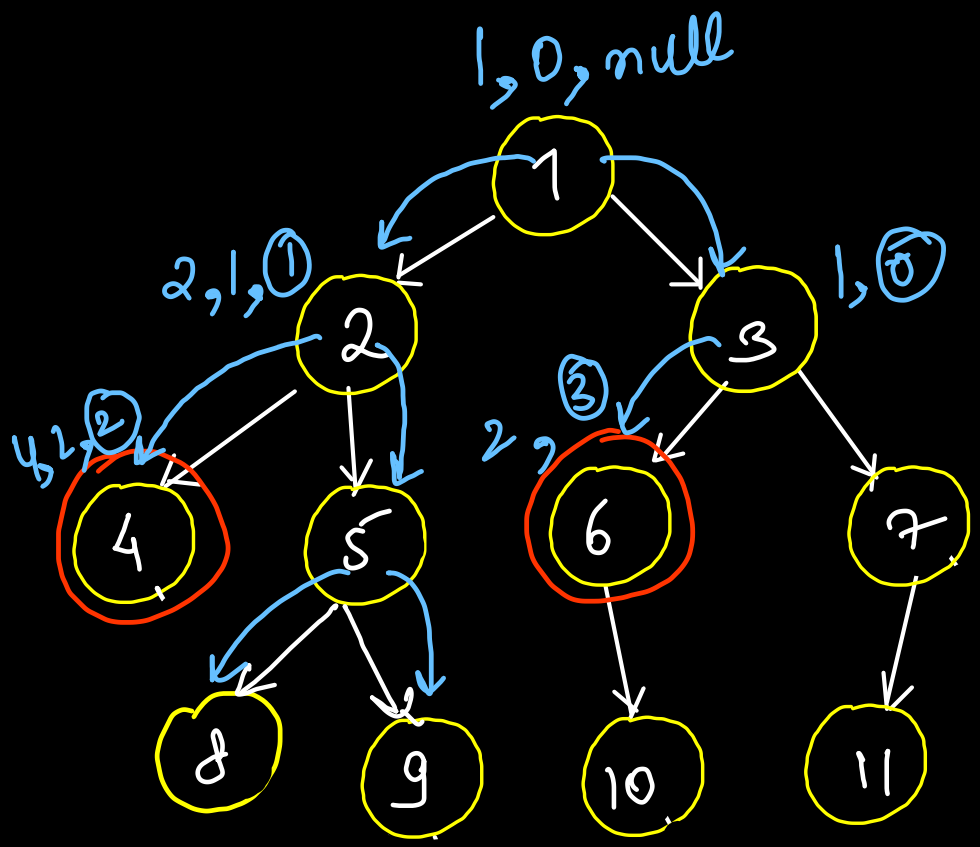
```java
public List<Double> averageOfLevels(TreeNode root) {
    List<Double> levels = new ArrayList<>();
    if(root == null) return levels;

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

    while(q.size() > 0) {
        double sum = 0, count = q.size();
        for(int c = q.size(); c > 0; c--) {
            root = q.remove();
            sum += root.val;
            if(root.left != null) q.add(root.left);
            if(root.right != null) q.add(root.right);
        }
        levels.add(sum / count);
    }

    return levels;
}
```

LC 637)

time = O(n), space = O(n)

```java
public int maxLevelSum(TreeNode root) {
    Queue<TreeNode> q = new ArrayDeque<>();
    q.add(root);

    int maxsum = Integer.MIN_VALUE, maxlevel = 0;
    int currsum = 0, currlevel = 0;
    while(q.size() > 0) {
        currsum = 0; currlevel++;

        for(int c = q.size(); c > 0; c--) {
            root = q.remove();
            currsum += root.val;
            if(root.left != null) q.add(root.left);
            if(root.right != null) q.add(root.right);
        }

        if(currsum > maxsum){
            maxsum = currsum;
            maxlevel = currlevel;
        }
    }

    return maxlevel;
}
```

LC 1161)



2, 3 → sibling false

x, y
4, 6 → cousins true

5, 7 → cousins true

7, 10 → neither cousin nor siblings. false

x.parent = 2 y.parent = 3
x.level = 2 y.level = 2

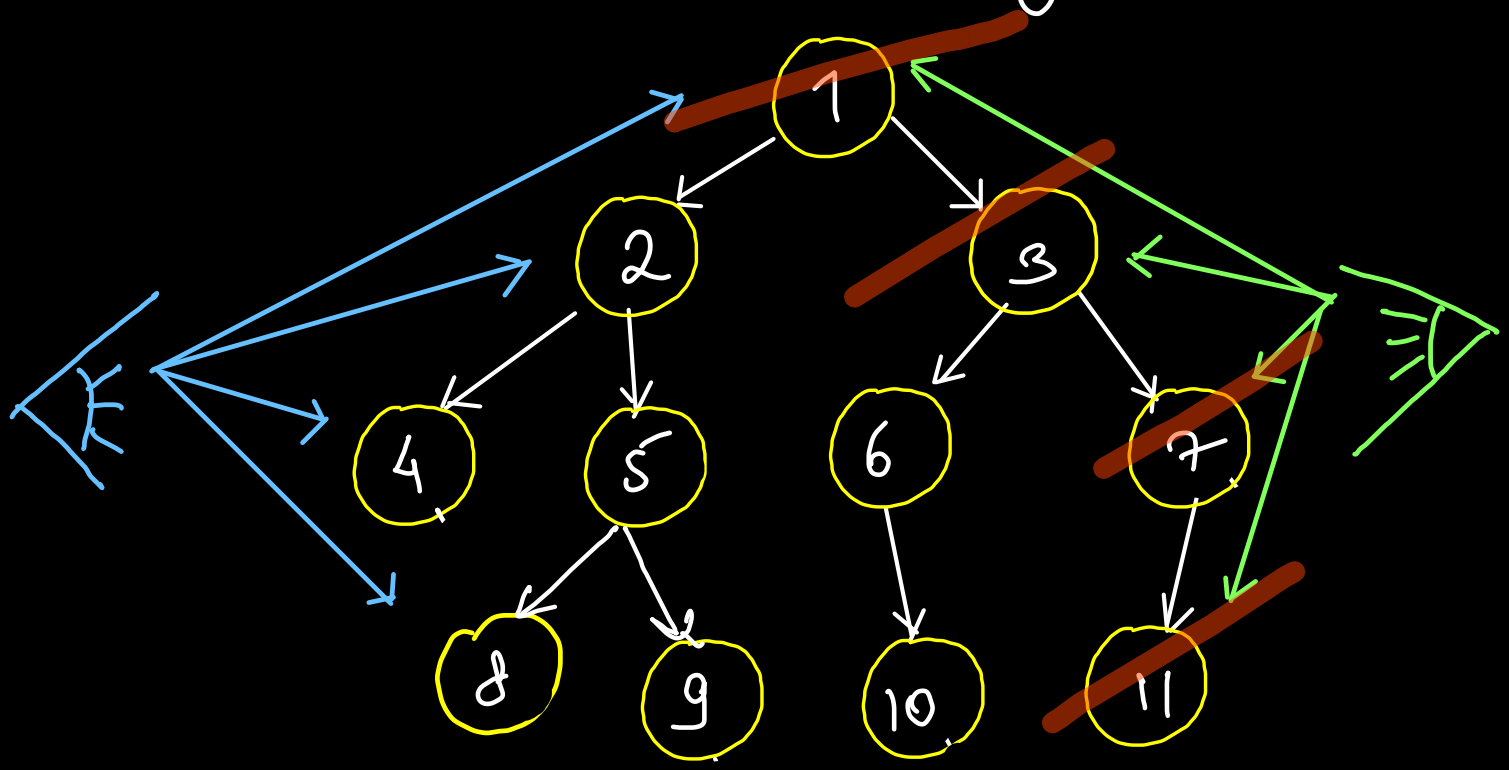
x.parent != y.parent &&
x.level == y.level

```java
class Solution {
    int xparent = -1, yparent = -1, xlevel = -1, ylevel = -1;

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

        if(root.val == x) {
            xparent = parent;
            xlevel = level;
        }

        if(root.val == y){
            yparent = parent;
            ylevel = level;
        }

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

    public boolean isCousins(TreeNode root, int x, int y) {
        dfs(root, 0, -1, x, y);
        return (xparent != yparent) && (xlevel == ylevel);
    }
}
```

Time = O(n)

Space = O(h)

Left View & Right View



left view :→ 1, 2, 4, 8
(leftmost/first node) of each level

right view → 1, 3, 7, 11
(rightmost/last node) of each level

Left View

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

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

    while(q.size() > 0){
        left.add(q.peek().data);
        for(int c = q.size(); c > 0; c--) {
            root = q.remove();
            if(root.left != null) q.add(root.left);
            if(root.right != null) q.add(root.right);
        }
    }

    return left;
}
```

Right View

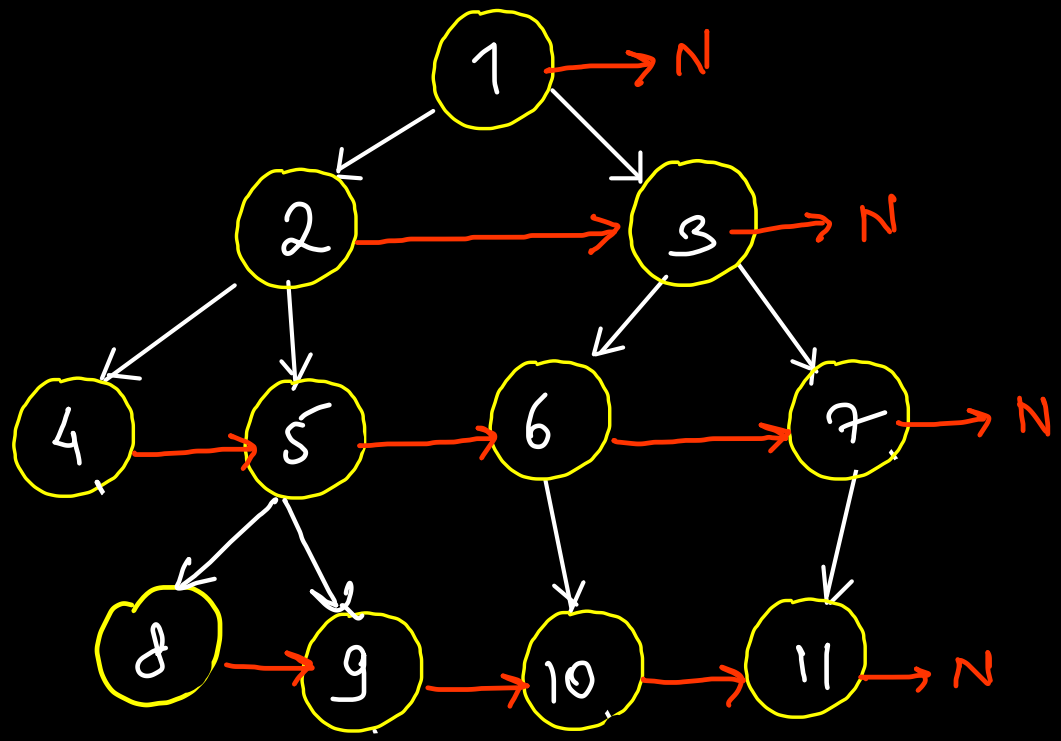
```
public List<Integer> rightSideView(TreeNode root) {
    List<Integer> right = new ArrayList<>();
    if(root == null) return right;

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

    while(q.size() > 0){
        for(int c = q.size(); c > 0; c--) {
            root = q.remove();
            if(root.left != null) q.add(root.left);
            if(root.right != null) q.add(root.right);
        }
        right.add(root.val);
    }

    return right;
}
```

Time = O(n)

Space = O(n)

Populating next right pointer

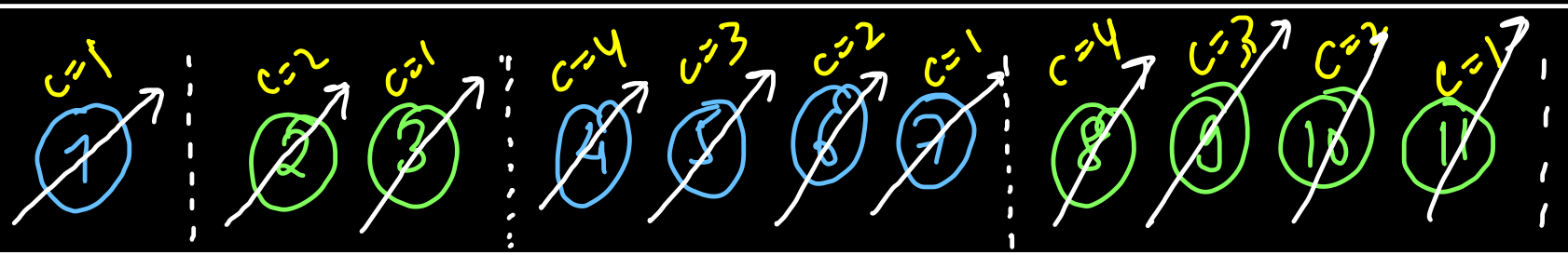
LC 116 & 117

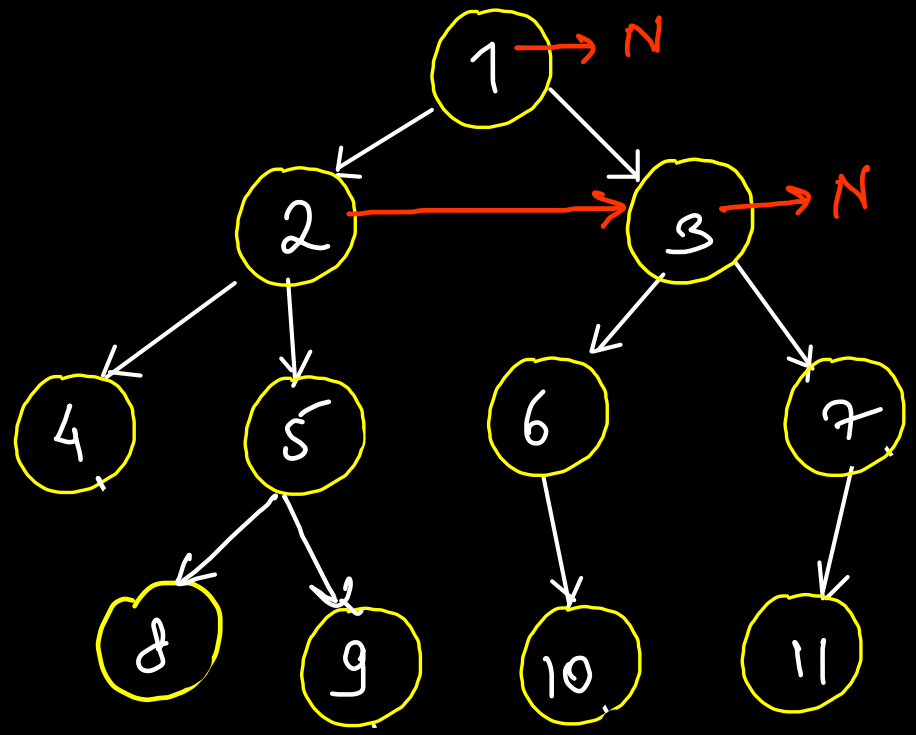


Approach 1)

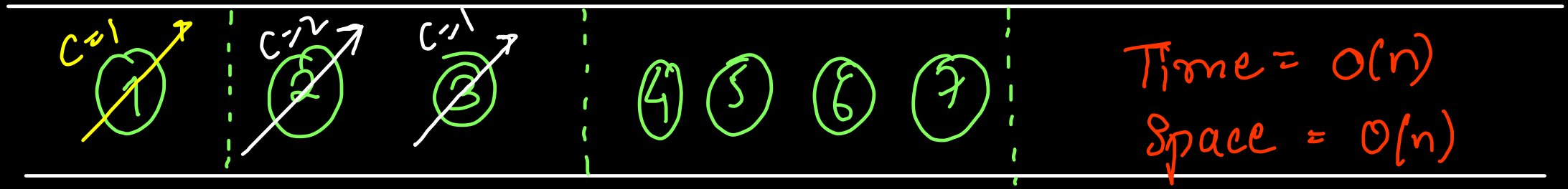
Queue → Time = O(n)
Space = O(n)

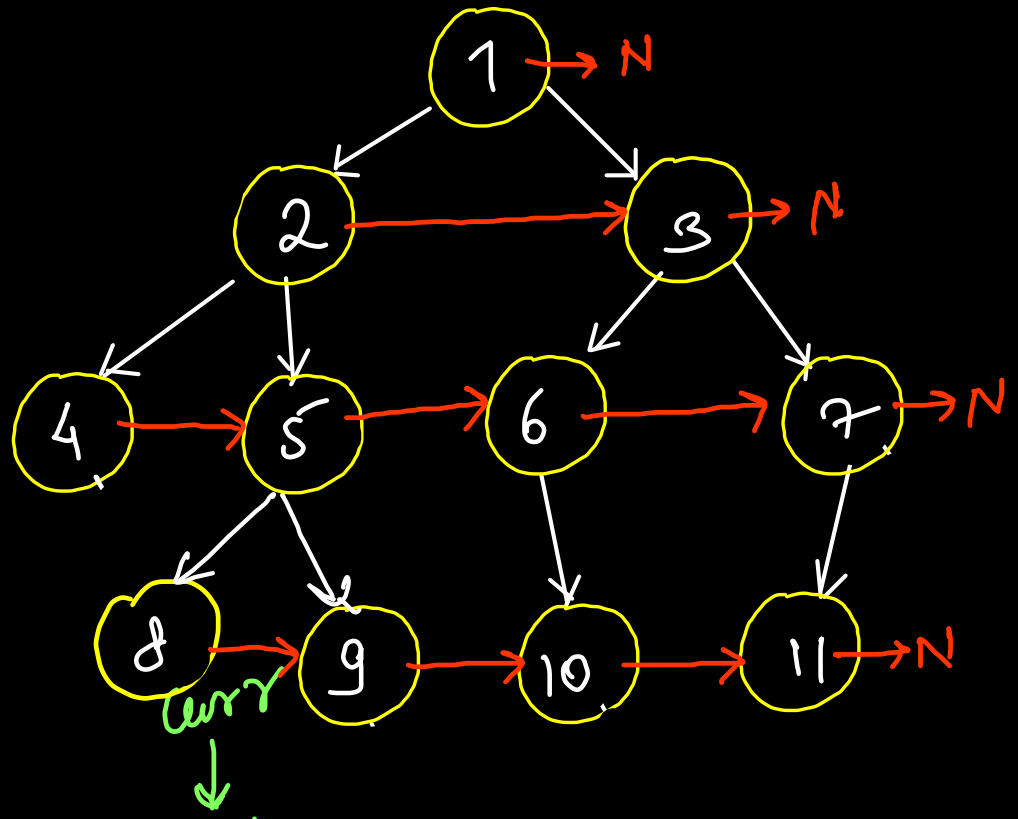
bfs level order






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

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

    while(q.size() > 0){
        for(int c = q.size(); c > 0; c--){
            Node curr = q.remove();
            if(c > 1) curr.next = q.peek();

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

    return root;
}
```





Queue { level order

↓

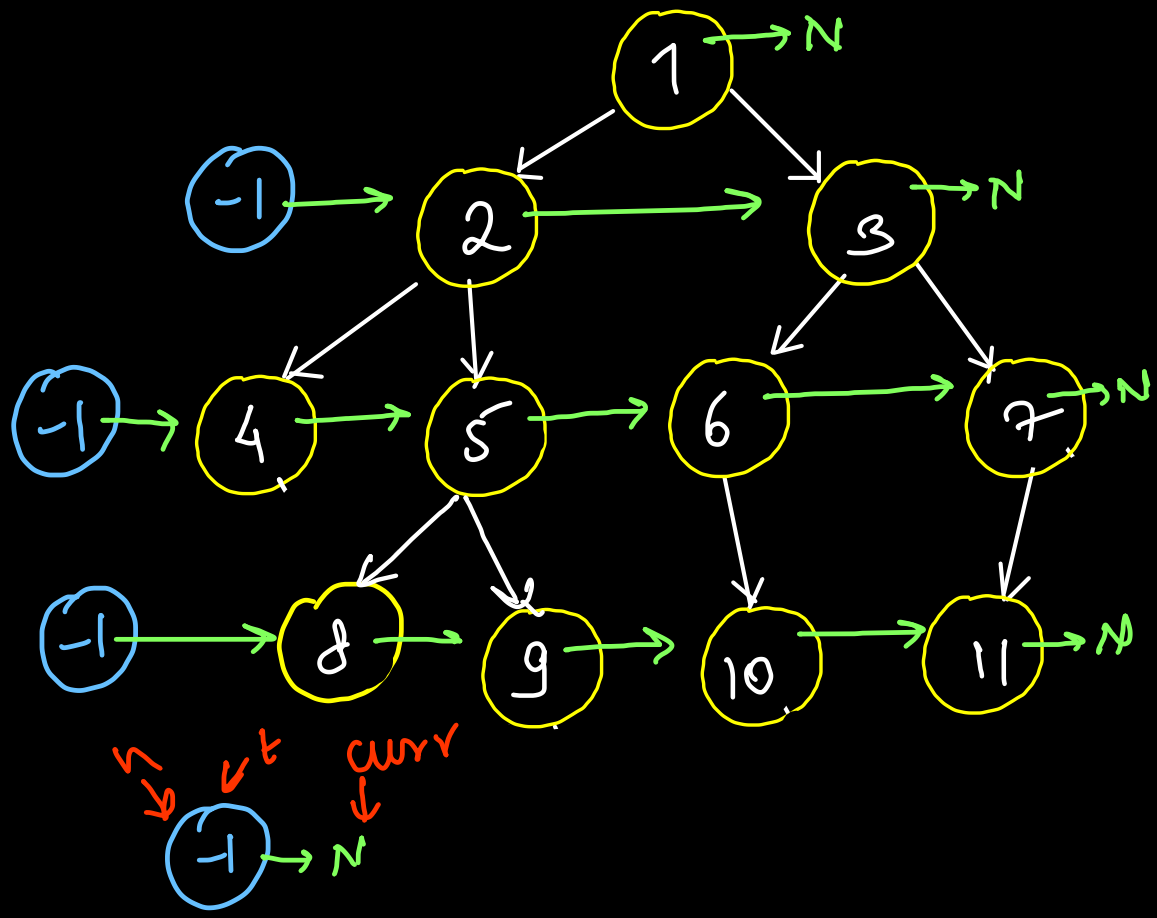
2 levels → curr level pop

→ next level add

Next Pointer (LL) { Approach 2

2 levels → curr level (next)

→ next level Construct




```java
public Node connect(Node root) {
    Node curr = root;
    while(curr != null)
    {
        Node head = new Node(-1);
        Node tail = head;
        for( ; curr != null; curr = curr.next){
            if(curr.left != null){
                tail.next = curr.left;
                tail = tail.next;
            }
            if(curr.right != null) {
                tail.next = curr.right;
                tail = tail.next;
            }
        }

        curr = head.next;
    }
    return root;
}
```

Time = O(n)

Space = O(1)

```java
public Node connect(Node root) {
    Node curr = root;
    while(curr != null)
    {
        Node head = new Node(-1);
        Node tail = head;
        for( ; curr != null; curr = curr.next){
            if(curr.left != null)
                tail = tail.next = curr.left;
            if(curr.right != null)
                tail = tail.next = curr.right;
        }

        curr = head.next;
    }
    return root;
}
```

Smaller version

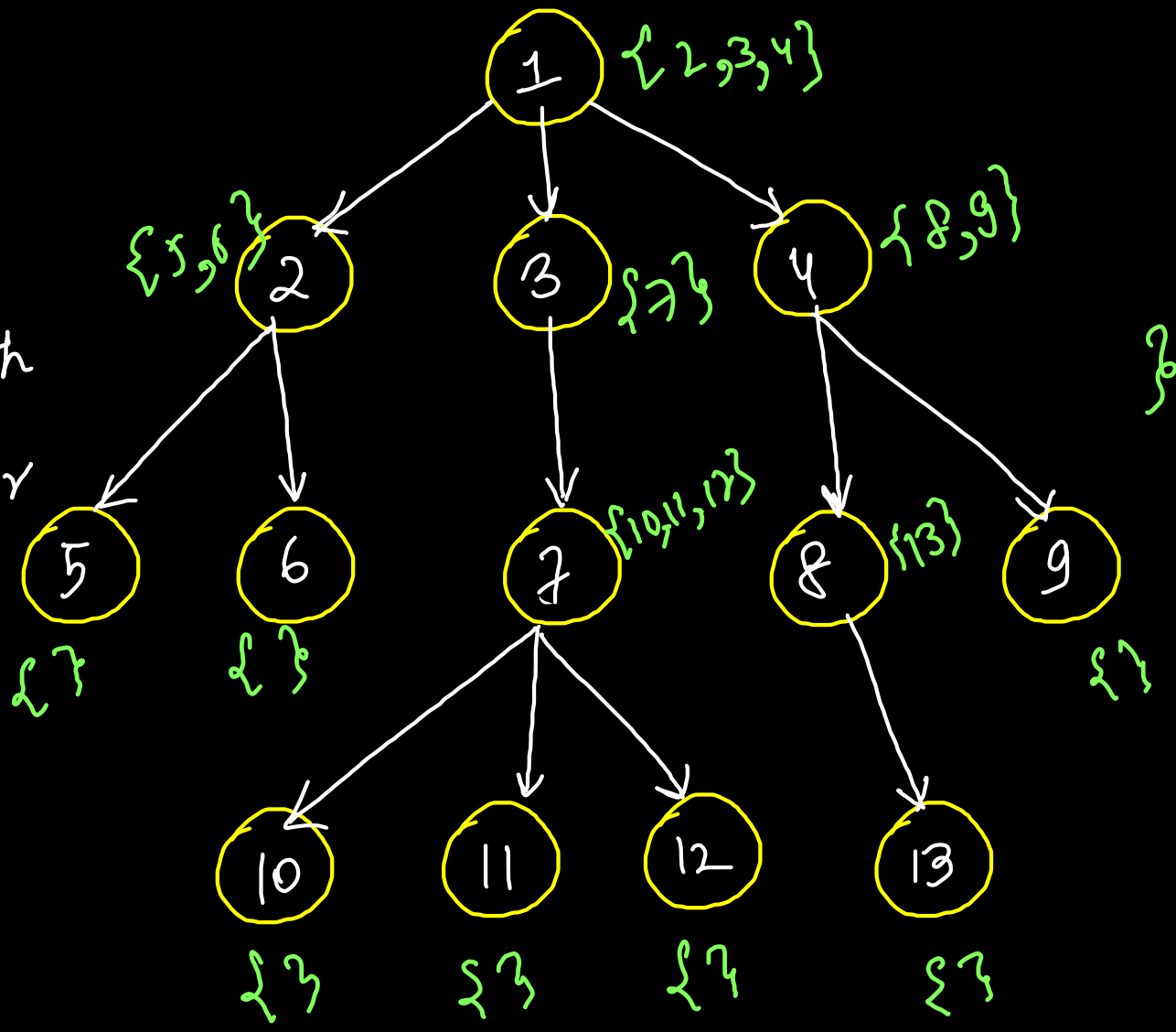
Nary or Generic Tree

Lc 589) Preorder

Lc 590) Postorder

Lc 559) Max^m depth

Lc 429) Level order




```
Node {
int val;
List<Node>
children;
}
```

```java
// Definition for a Node.
class Node {
    public int val;
    public List<Node> children;

    public Node() {}

    public Node(int _val) {
        val = _val;
    }

    public Node(int _val, List<Node> _children) {
        val = _val;
        children = _children;
    }
};
```

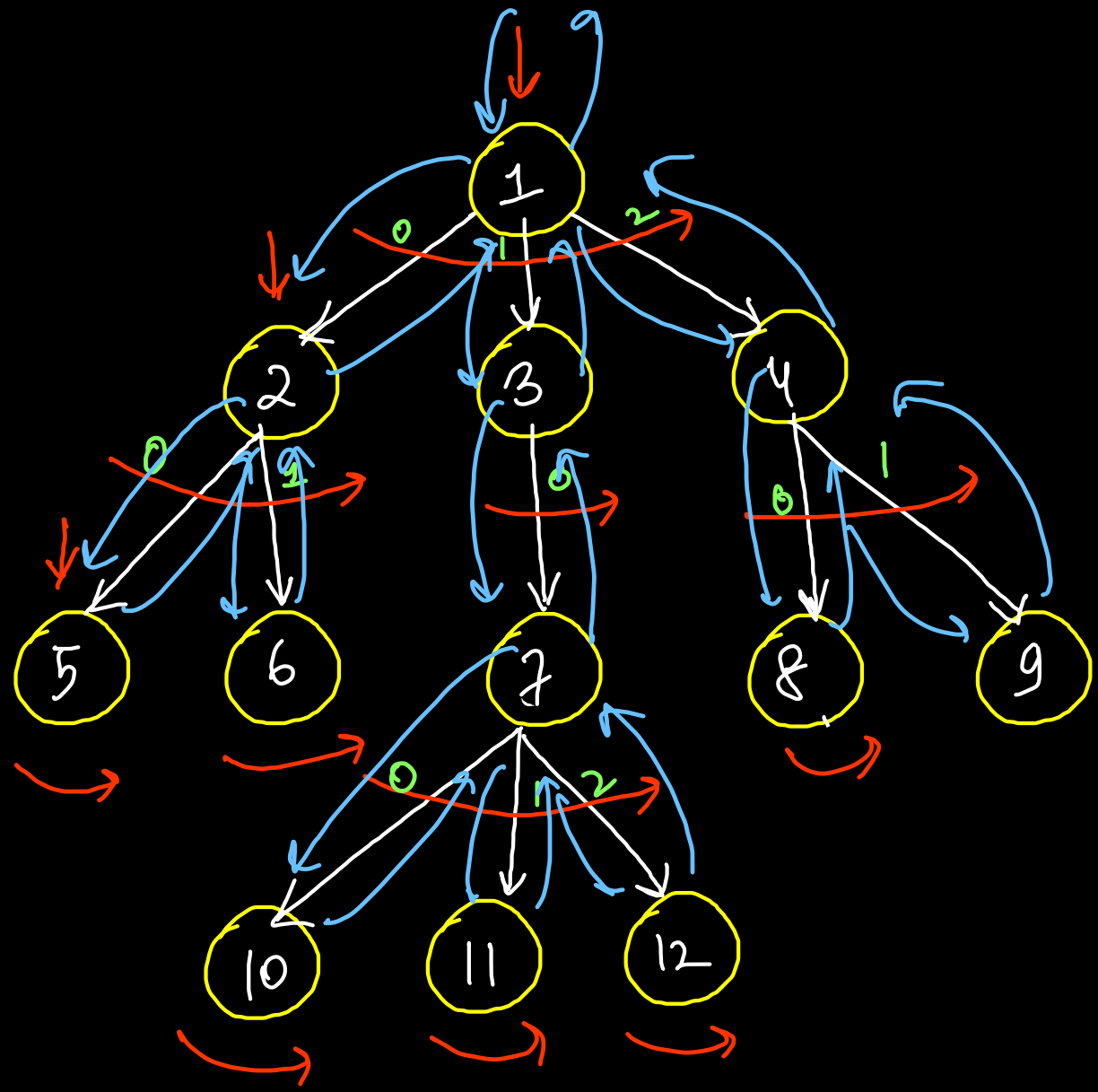
Input Space Complexity

: O(nodes + edges)

= O(N + N-1)

= O(2N-1)

Linear



dfs

- Preorder
  - root → children
- Postorder
  - children → root (l to r)

Preorder :→

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

Postorder :→

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

Preorder (Lc 589)

```java
class Solution {
    List<Integer> ans = new ArrayList<>();
    public void dfs(Node root){
        ans.add(root.val);
        for(Node child: root.children)
            dfs(child);
    }

    public List<Integer> preorder(Node root) {
        if(root != null) dfs(root);
        return ans;
    }
}
```

Postorder (Lc 590)

```java
class Solution {
    List<Integer> ans = new ArrayList<>();
    public void dfs(Node root){
        for(Node child: root.children)
            dfs(child);
        ans.add(root.val);
    }

    public List<Integer> postorder(Node root) {
        if(root != null) dfs(root);
        return ans;
    }
}
```

Time ⇒ Recursion + Loop (independent)

↓ O(n) + ↓ O(e) = O(n-1) total

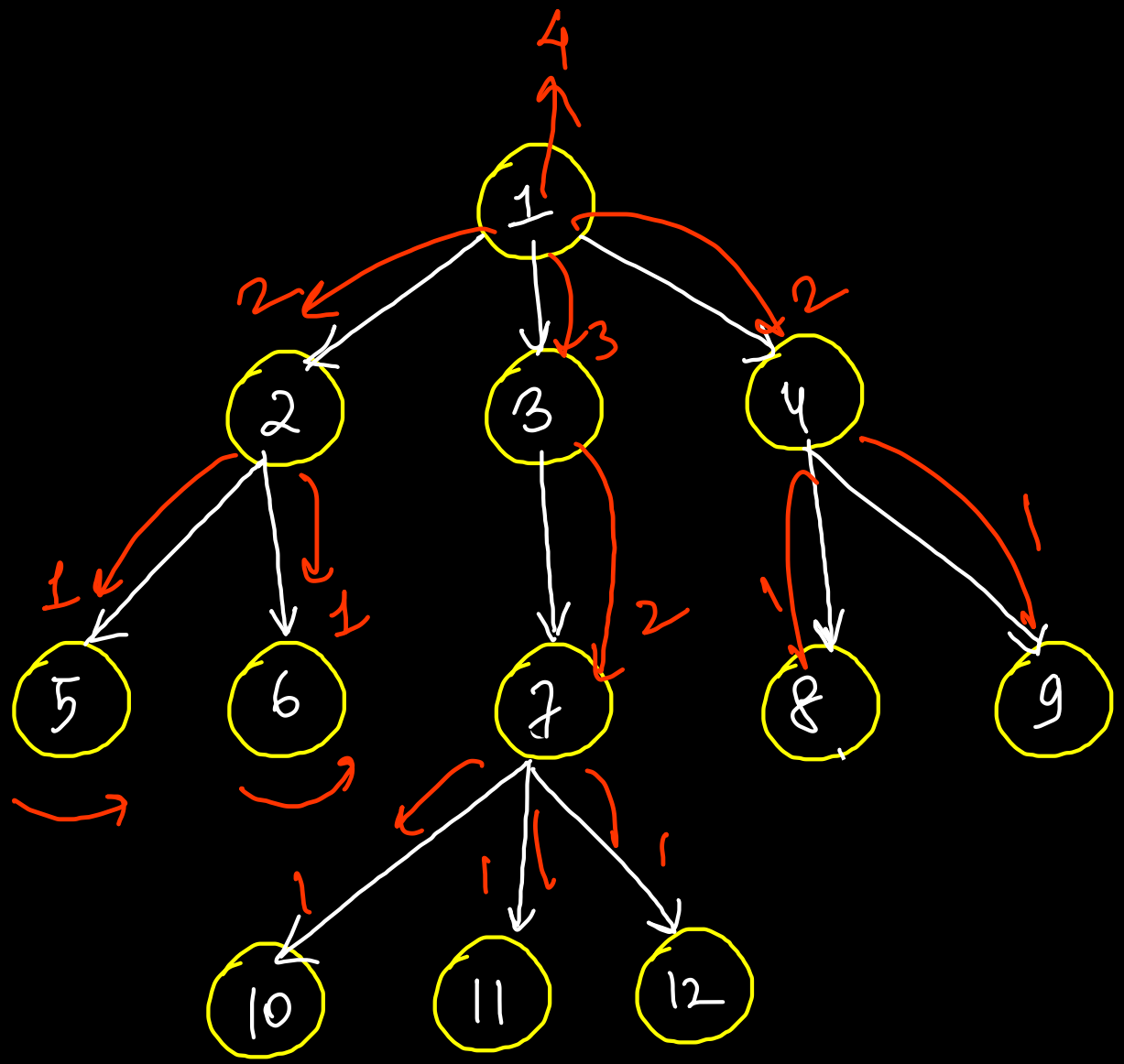
O(2n-1) linear

Space ⇒ recursion call stack

O(height)

↓ avg O($\log_D$ H)

↓ worst O(n)



Max Depth of Nary Tree

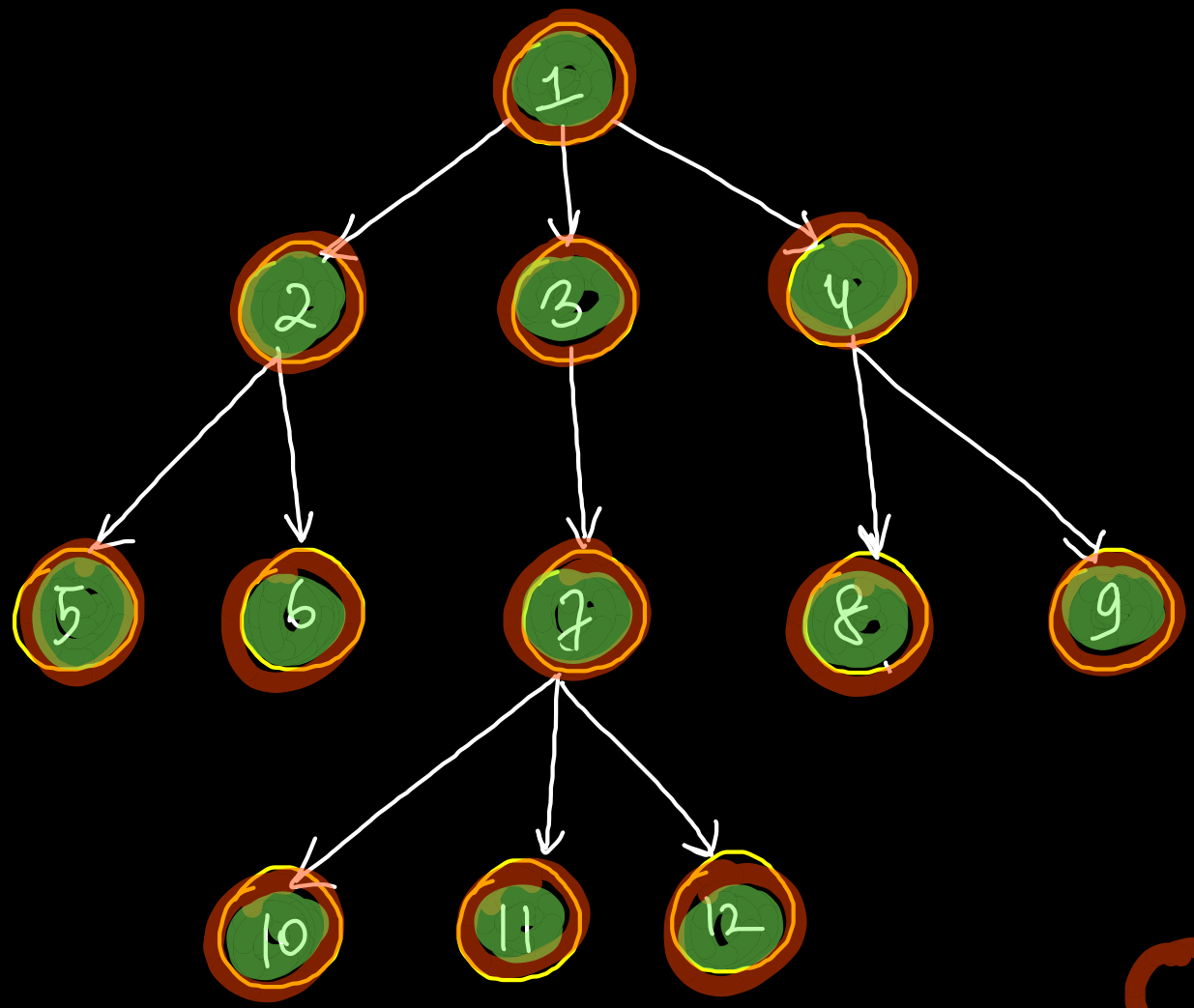
LC 559

```java
public int maxDepth(Node root) {
    if(root == null) return 0;

    int height = 0;
    for(Node child: root.children){
        height = Math.max(height, maxDepth(child));
    }
    return 1 + height;
}
```

Time ⇒ O(n) linear

Space ⇒ O(h) dfs

429 Level order



{{1}

{2,3,4}

{5,6,7,&9}

{10,11,12}}

→ push

→ pop

```java
public List<List<Integer>> levelOrder(Node root) {
    List<List<Integer>> levels = new ArrayList<>();
    if(root == null) return levels;
    Queue<Node> q = new ArrayDeque<>();
    q.add(root);

    while(q.size() > 0){
        List<Integer> level = new ArrayList<>();
        for(int c = q.size(); c > 0; c--){
            root = q.remove();
            level.add(root.val);
            for(Node child: root.children)
                q.add(child);
        }
        levels.add(level);
    }
    return levels;
}
```

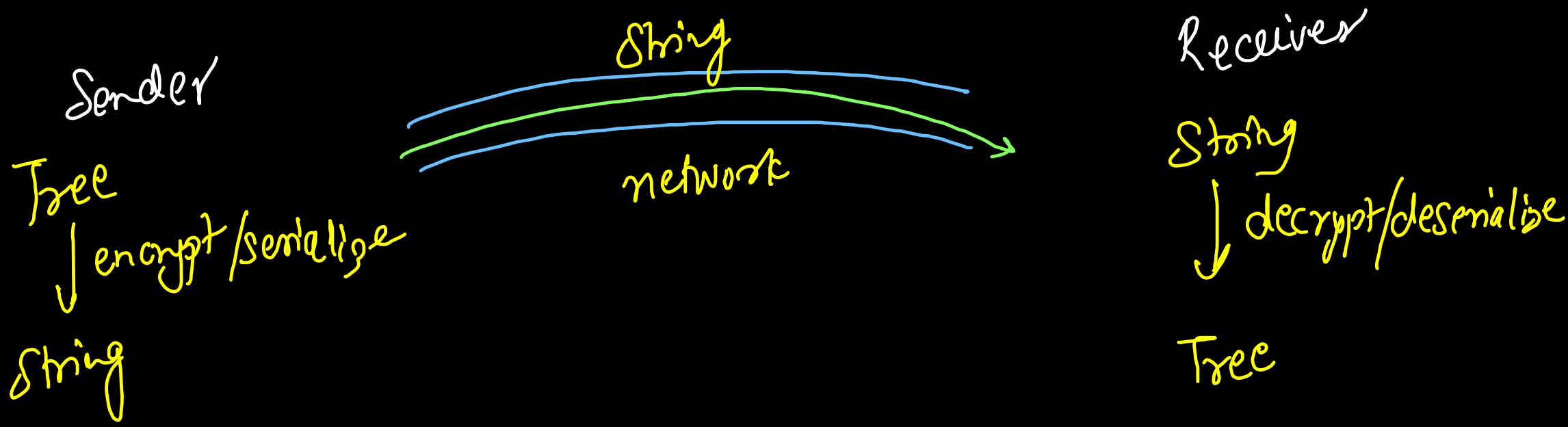
LC 429

Time = O(nte) = O(2n)
linear

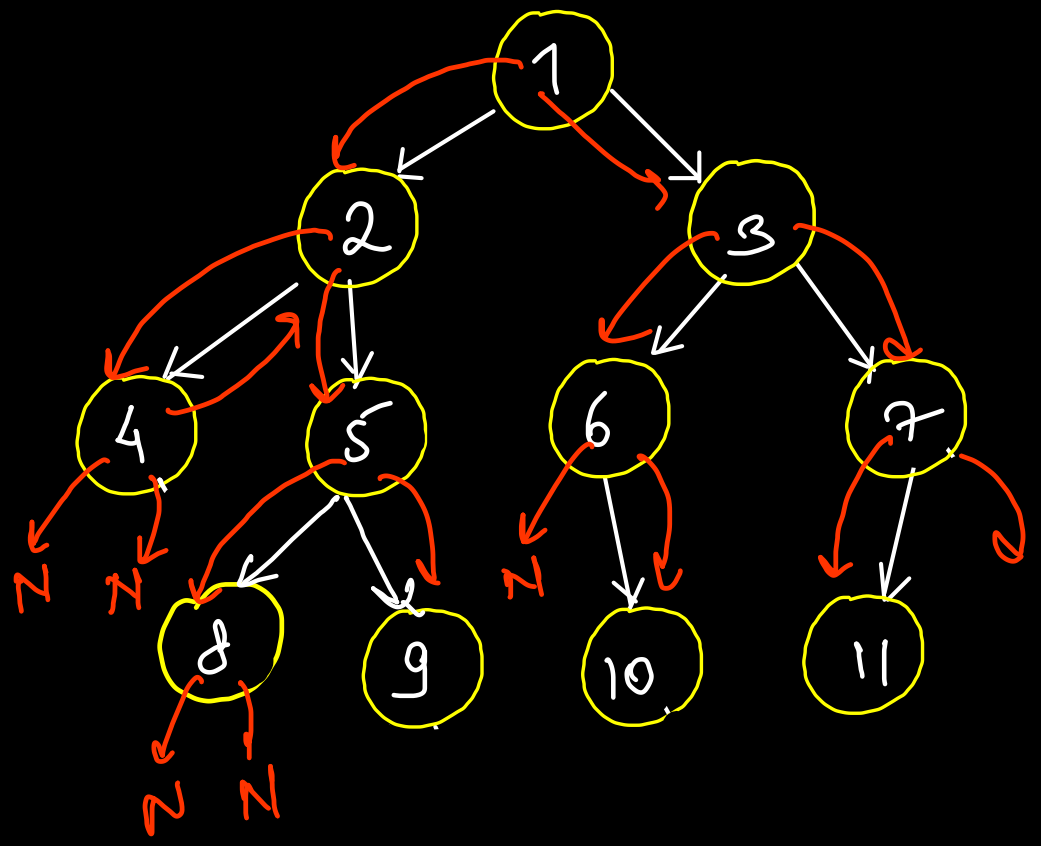
Space = O(n)

Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be reconstructed later in the same or another computer environment.

Design an algorithm to serialize and deserialize a binary tree. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary tree can be serialized to a string and this string can be deserialized to the original tree structure.



Approach 1) dfs

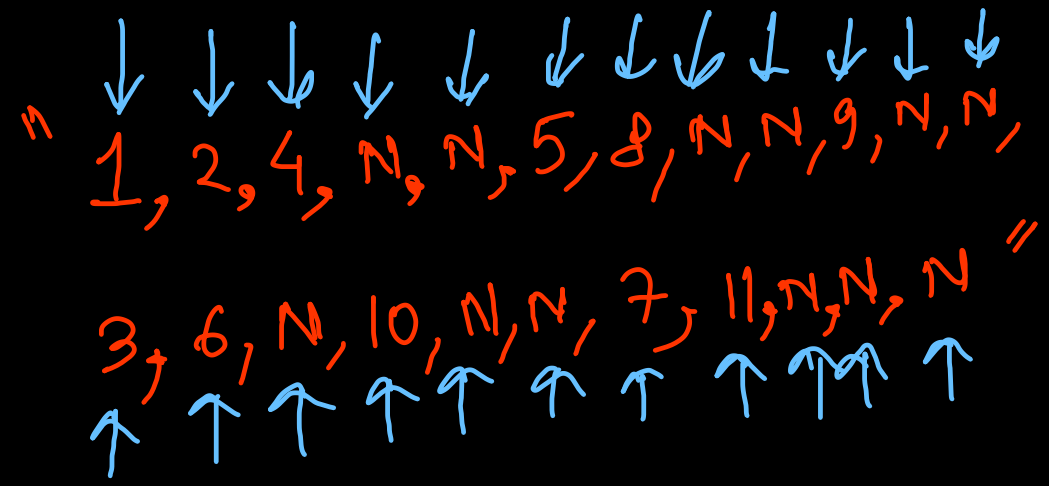


Preorder Serialization

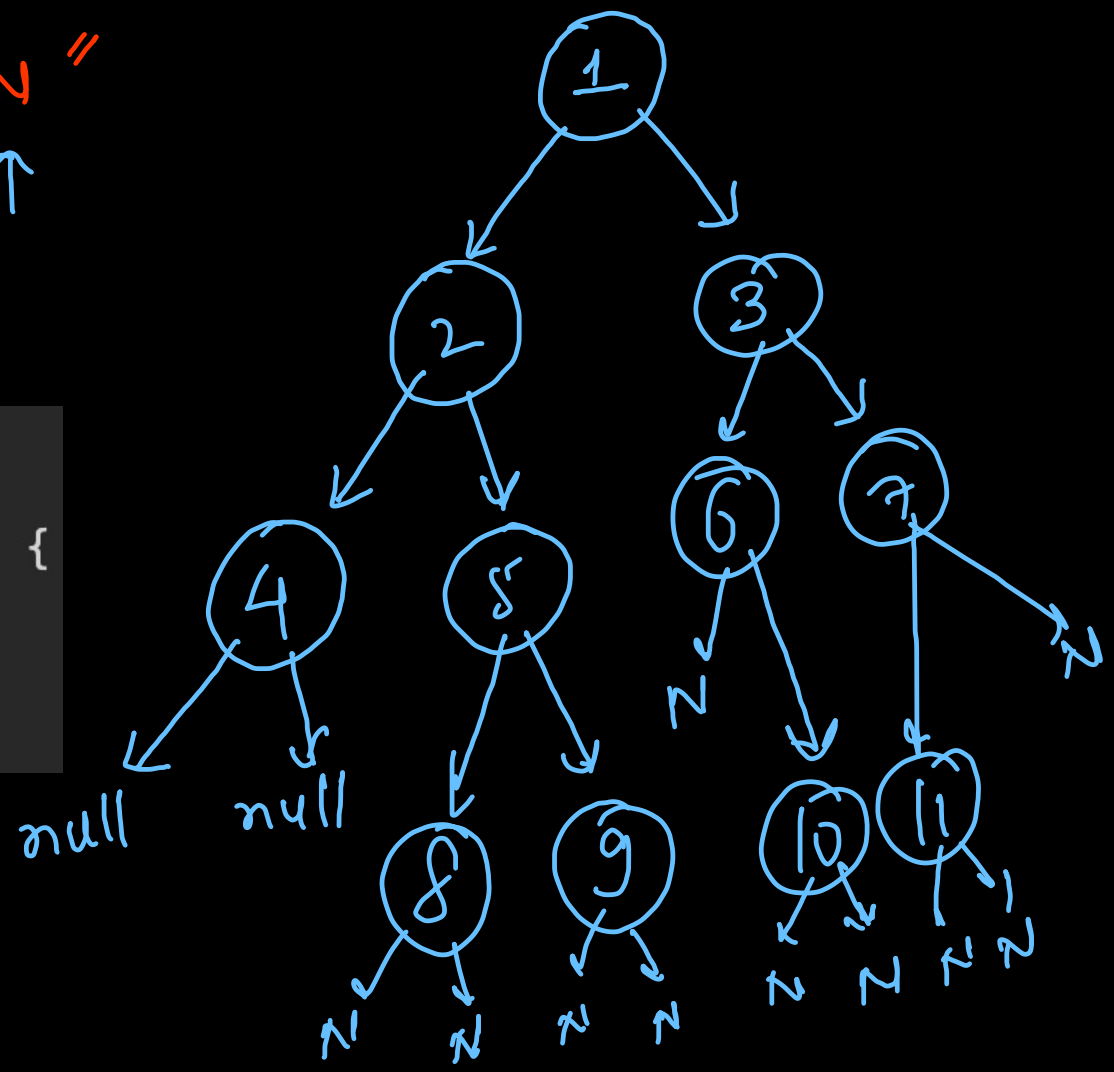
String :

" 1, 2, 4, N, N, 5, 8, N, N, 9, N, N,

3, 6, N, 10, N, N, 7, 11, N, N, N "



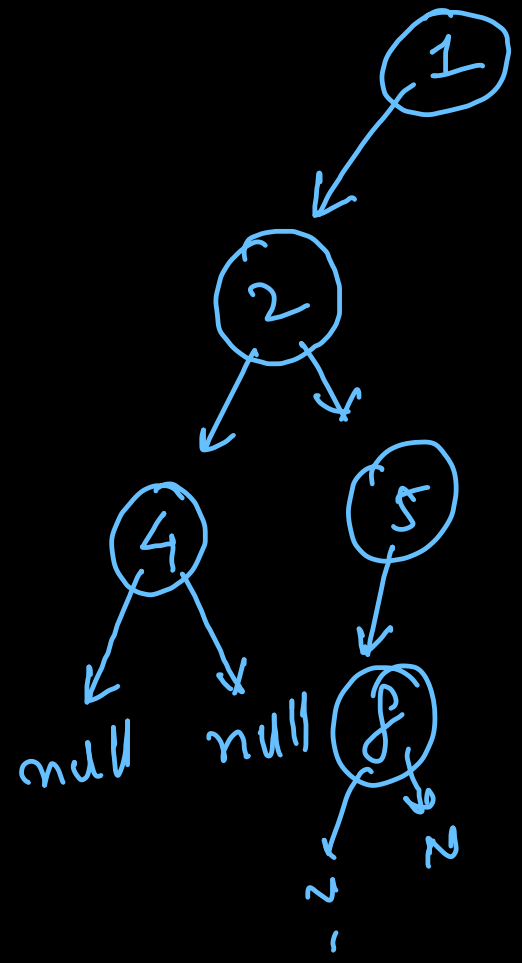
root → root.left → root.right




```java
// Decodes your encoded data to tree.
public TreeNode deserialize(String data) {
    String[] nodes = data.split(",");
    return dfs(nodes);
}
```

```java
int idx = 0;
public TreeNode dfs(String[] nodes){
    if(idx == nodes.length) return null;
    if(nodes[idx].equals("N")) {
        idx++;
        return null;
    }

    int val = Integer.parseInt(nodes[idx++]);
    TreeNode root = new TreeNode(val);
    root.left = dfs(nodes);
    root.right = dfs(nodes);
    return root;
}
```

O(n)



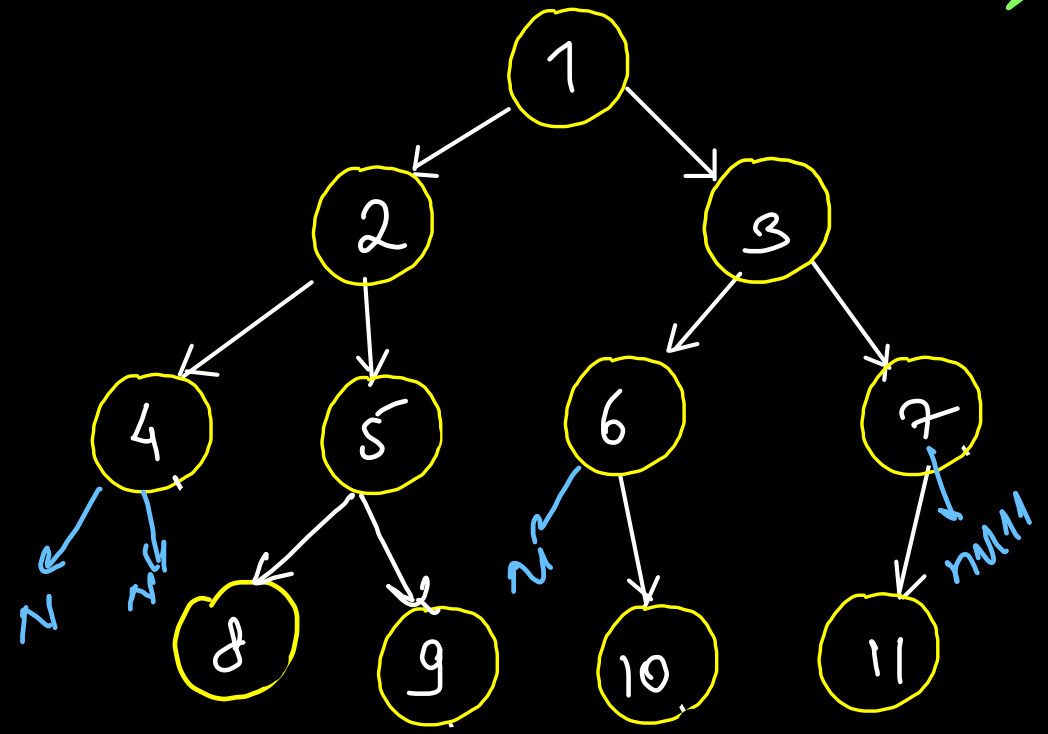
"1, 2, 4, N, N, 5, 8, N, N, 9, N, N,
3, 6, N, 10, N, N, 7, 11, N, N, N"

Approach 1

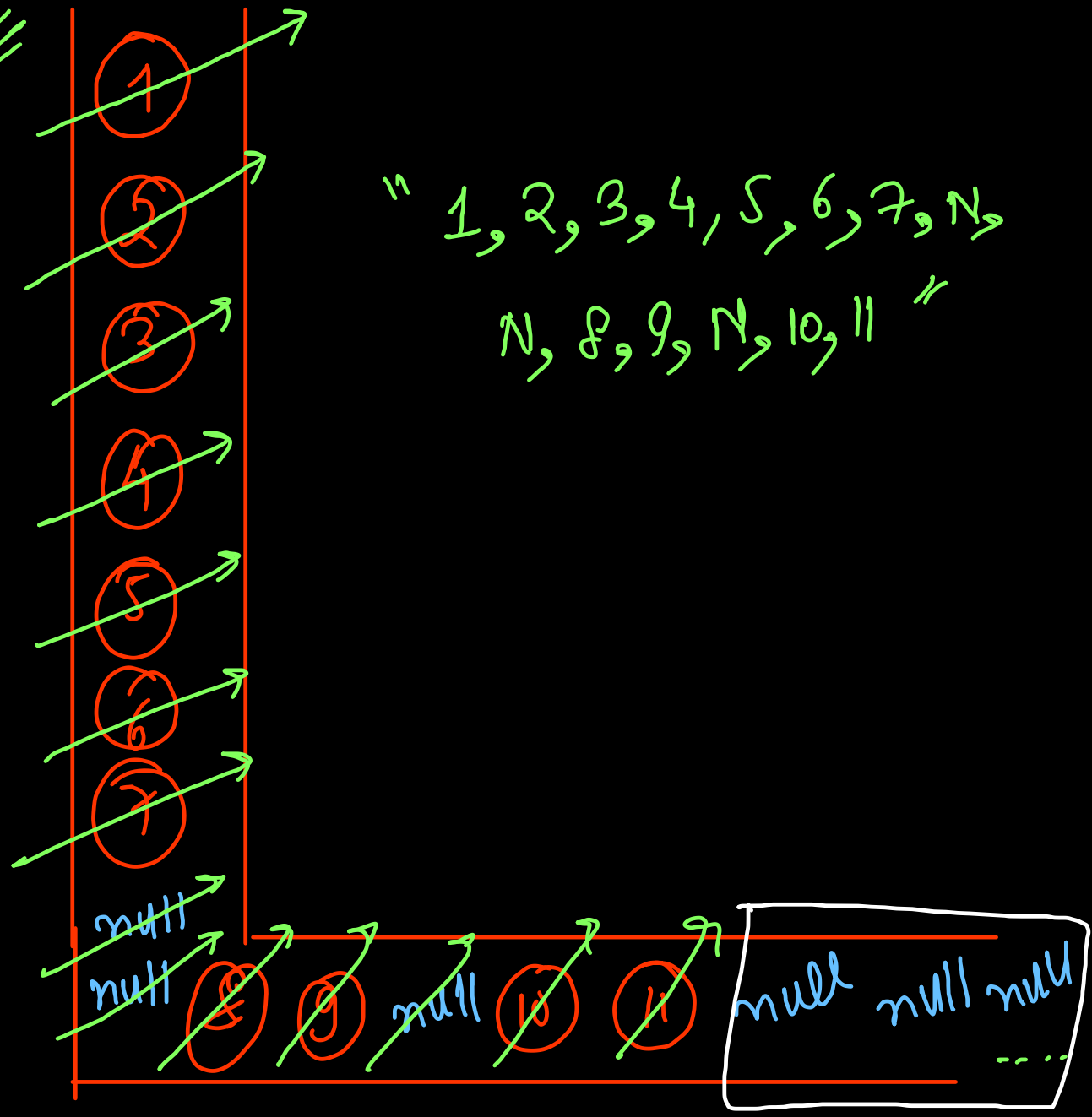
```java
// Encodes a tree to a single string.
public String serialize(TreeNode root) {
    if(root == null) return "N,";
    String ans = root.val + ",";
    ans += serialize(root.left);
    ans += serialize(root.right);
    return ans;
}

int idx = 0;
public TreeNode dfs(String[] nodes){
    if(idx == nodes.length) return null;
    if(nodes[idx].equals("N")) {
        idx++;
        return null;
    }

    int val = Integer.parseInt(nodes[idx++]);
    TreeNode root = new TreeNode(val);
    root.left = dfs(nodes);
    root.right = dfs(nodes);
    return root;
}

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

O(n)

O(n)



LL



" 1, 2, 3, 4, 5, 6, 7, N, N, 8, 9, N, 10, 11 "

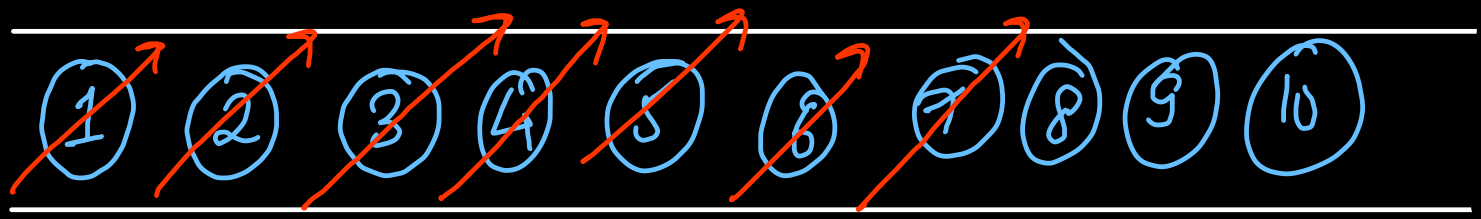
Approach 2) Level order

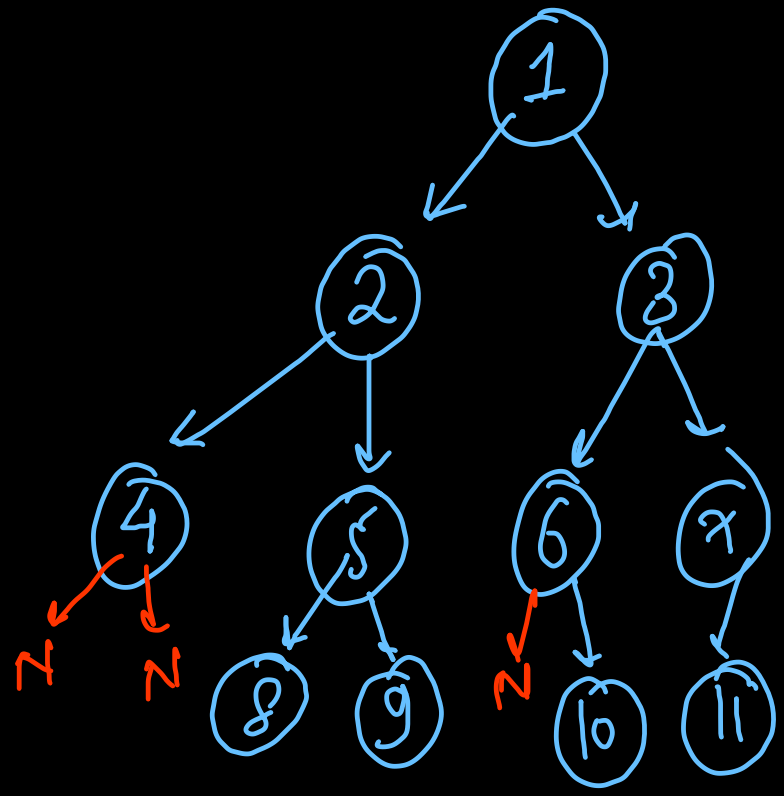
```java
public String serialize(TreeNode root) {
    LinkedList<TreeNode> q = new LinkedList<>();
    q.addLast(root);

    StringBuilder res = new StringBuilder();
    while(q.size() > 0){
        root = q.removeFirst();

        if(root == null) {
            res.append("N,");
            continue;
        }

        res.append(root.val + ",");
        q.addLast(root.left);
        q.addLast(root.right);
    }

    return res.toString();
}
```

O(n) time

O(n) space

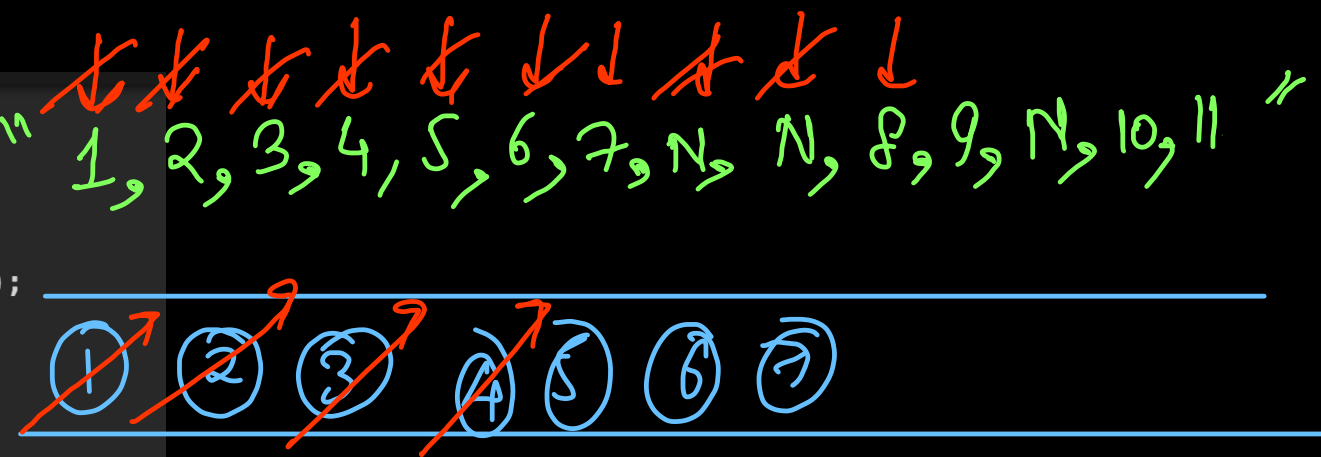
" 1, 2, 3, 4, 5, 6, 7, N, N, 8, 9, N, 10, 11 "

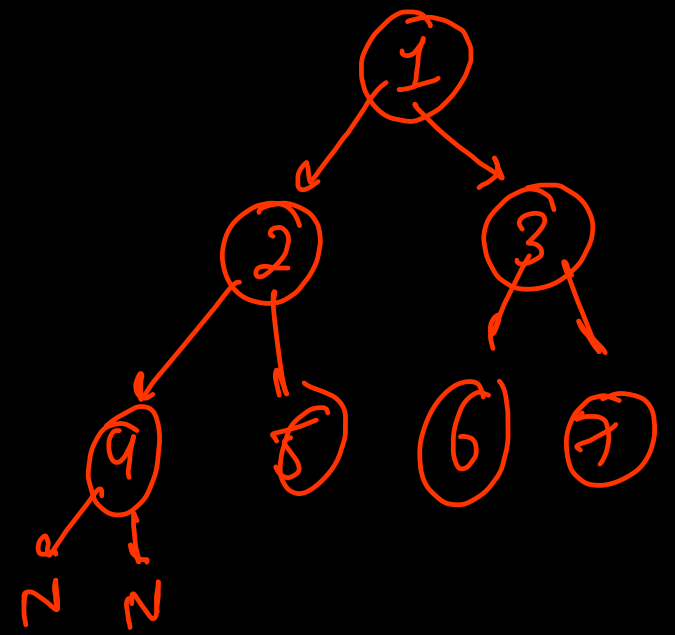




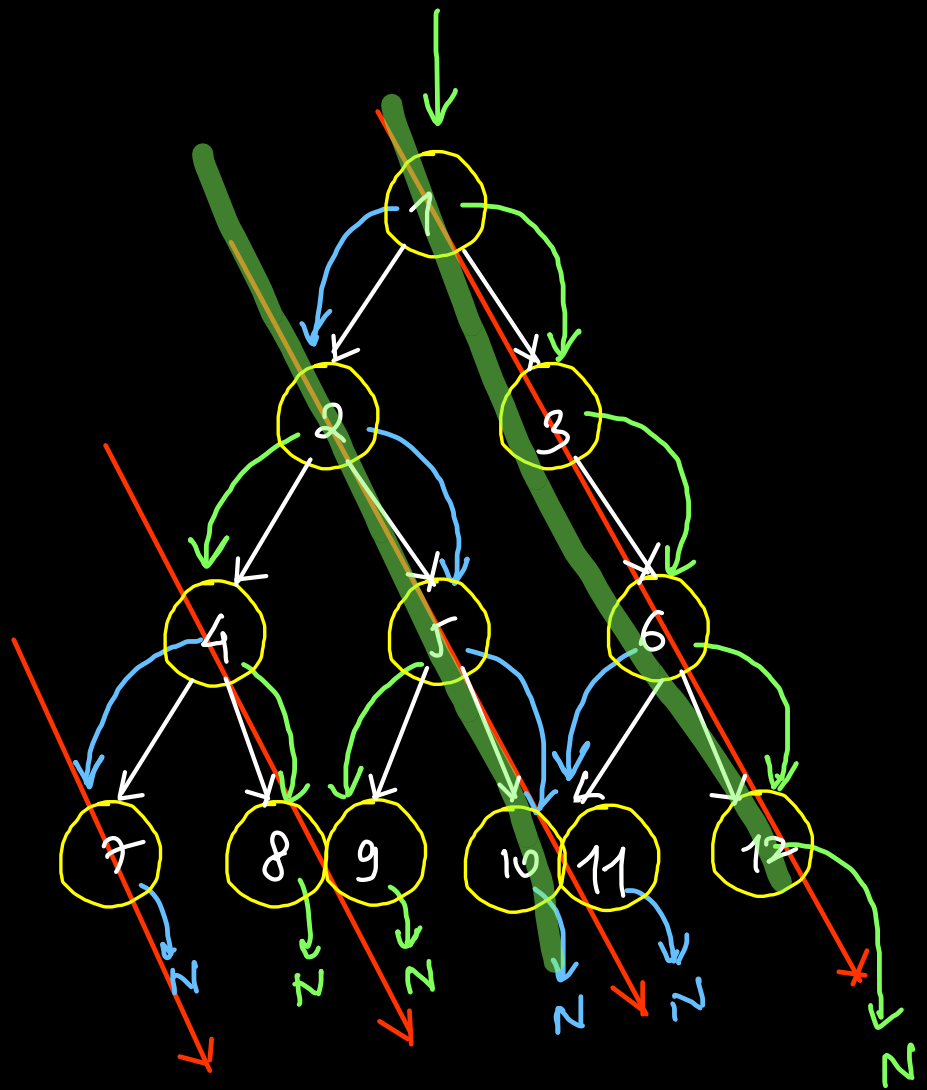
O(n)

```java
public TreeNode deserialize(String data) {
    String[] nodes = data.split(",");
    if(nodes[0].equals("N") == true) return null;

    TreeNode root = new TreeNode(Integer.parseInt(nodes[0]));
    LinkedList<TreeNode> q = new LinkedList<>();
    q.addLast(root);
    int idx = 1;

    while(q.size() > 0){
        TreeNode curr = q.removeFirst();

        if(nodes[idx].equals("N") == false) {
            curr.left = new TreeNode(Integer.parseInt(nodes[idx]));
            q.addLast(curr.left);
        }
        idx++;

        if(nodes[idx].equals("N") == false) {
            curr.right = new TreeNode(Integer.parseInt(nodes[idx]));
            q.addLast(curr.right);
        }
        idx++;
    }

    return root;
}
```





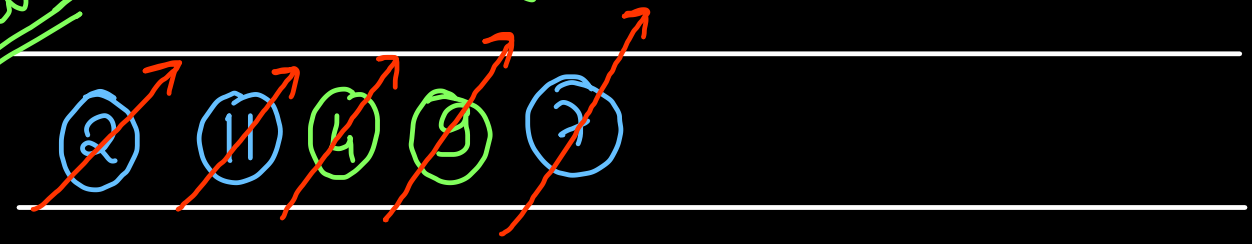
Diagonal Traversal of Binary Tree



Same diagonal → right child

next diagonal → left child

queue



res { 1, 3, 6, 12, 2, 5, 10, 11, 4, 8, 9, 7 }

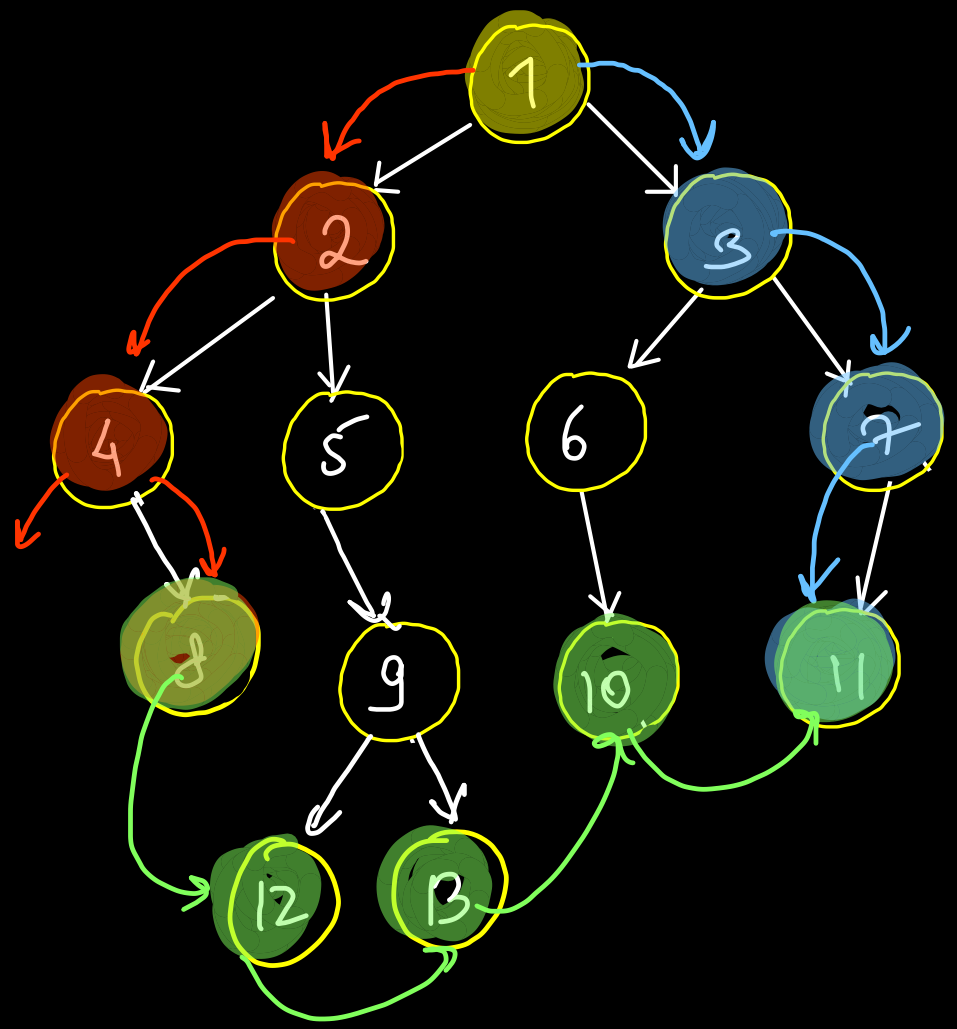
```java
public ArrayList<Integer> diagonal(Node root)
{
    ArrayList<Integer> res = new ArrayList<>();
    Queue<Node> q = new ArrayDeque<>();

    while(root != null || q.size() > 0){
        if(root == null){
            root = q.remove();
        } else {
            res.add(root.data);
            if(root.left != null) q.add(root.left);
            root = root.right;
        }
    }

    return res;
}
```

same

while(!(root == null && q.size() == 0))

Space ⇒ O(diagonal) (Queue) or O(h)

Time ⇒ O(n)



preorder

Left Boundary (Top to Bottom)

≠ Left view

1, 2, 4, 8

Leaf node (left to right)

8, 12, 13, 10, 11

postorder

Right Boundary (Bottom to top)

11, 7, 3, 1 ≠ Right view

```java
ArrayList<Integer> res;
void leftBoundary(Node root){
    if(root == null) return;
    if(root.left == null && root.right == null)
        return; // leaf node is ignored

    res.add(root.data); // preorder
    if(root.left != null) leftBoundary(root.left);
    else leftBoundary(root.right);
}

void rightBoundary(Node root){
    if(root == null) return;
    if(root.left == null && root.right == null)
        return; // leaf node is ignored

    if(root.right != null) rightBoundary(root.right);
    else rightBoundary(root.left);
    res.add(root.data); // postorder
}
```

O(h)

O(h)

```java
void dfs(Node root){
    if(root == null) return;
    if(root.left == null && root.right == null)
        res.add(root.data);
    dfs(root.left);
    dfs(root.right);
}

ArrayList <Integer> boundary(Node root)
{
    res = new ArrayList<>();
    res.add(root.data);
    leftBoundary(root.left);
    dfs(root.left);
    dfs(root.right);
    rightBoundary(root.right);
    return res;
}
```

O(n)

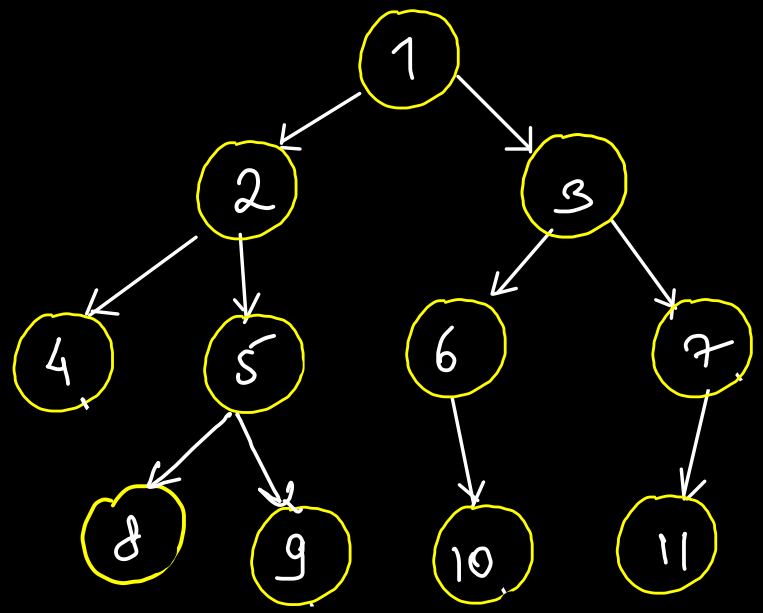
O(h)

O(n)

O(h)

time = O(H + N + H)

Space = O(H) recursion.



root → rootleft → rootright

rootidx

0 1 2 3 4 5 6 7 8 9 10

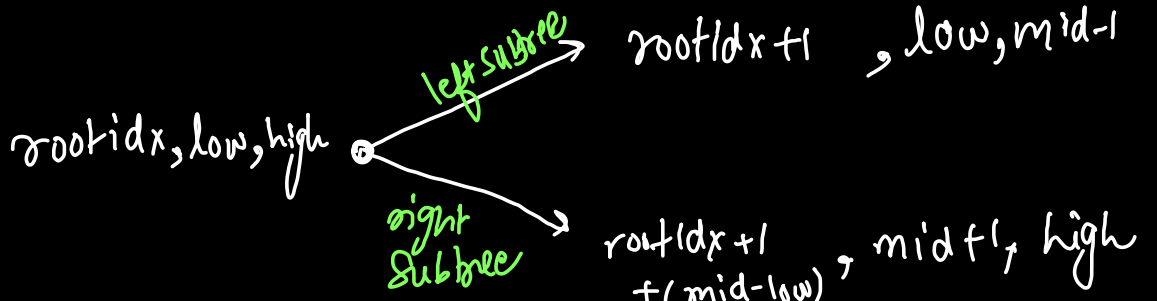
preorder : { 1, 2, 4, 5, 8, 9, 3, 6, 10, 7, 11 }

inorder : { 4, 2, 8, 5, 9, 1, 6, 10, 8, 11, 7 }

0 1 2 3 4 5 6 7 8 9 10

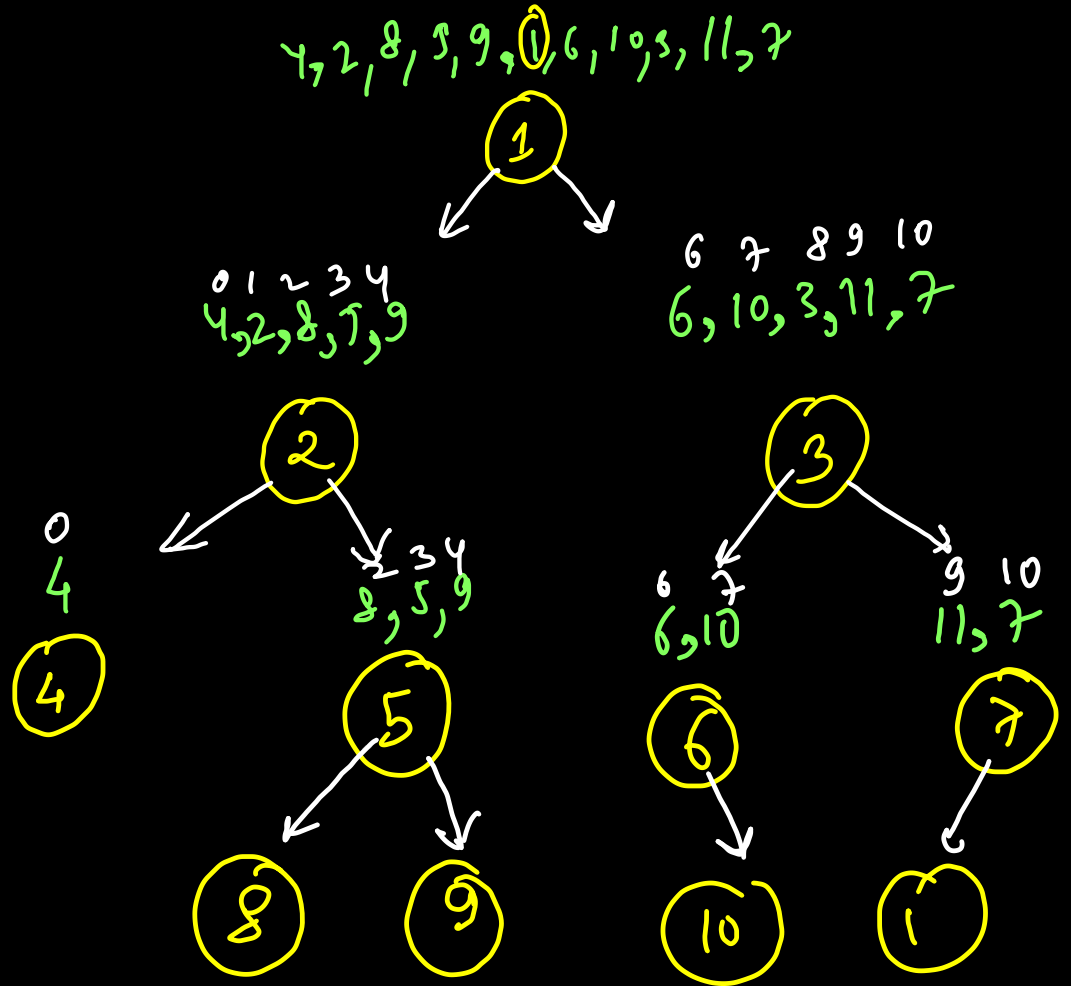
low mid high

left → root → right



preorder : { 1, 2, 4, 5, 8, 9, 3, 6, 10, 7, 11 }
0 1 2 3 4 5 6 7 8 9 10

inorder : { 4, 2, 8, 5, 9, 1, 6, 10, 3, 11, 7 }
0 1 2 3 4 5 6 7 8 9 10



```java
class Solution {
    int[] preorder, inorder;

    public int search(int val){
        for(int i = 0; i < inorder.length; i++){
            if(inorder[i] == val) return i;
        }
        return -1;
    }
    public TreeNode construct(int rootidx, int low, int high){
        if(low > high) return null;
        if(low == high) return new TreeNode(preorder[rootidx]);

        TreeNode root = new TreeNode(preorder[rootidx]);
        int mid = search(preorder[rootidx]);
        root.left = construct(rootidx + 1, low, mid - 1);
        root.right = construct(rootidx + 1 + (mid - low), mid + 1, high);
        return root;
    }

    public TreeNode buildTree(int[] preorder, int[] inorder) {
        this.preorder = preorder;
        this.inorder = inorder;
        return construct(0, 0, inorder.length - 1);
    }
}
```

→ assuming search is constant-

→ no of nodes in left subtree

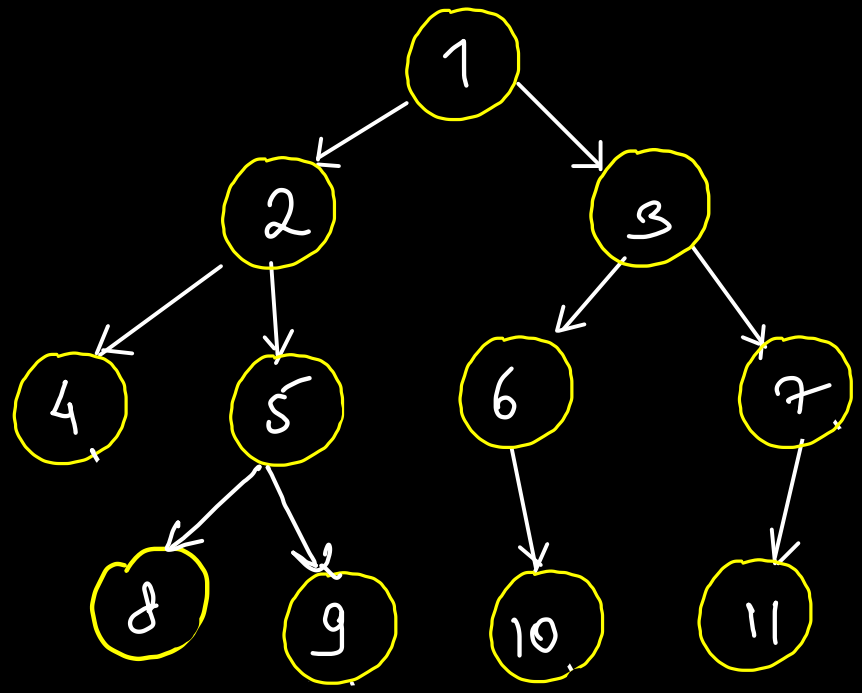
if balanced binary tree

: O(n log n) avg

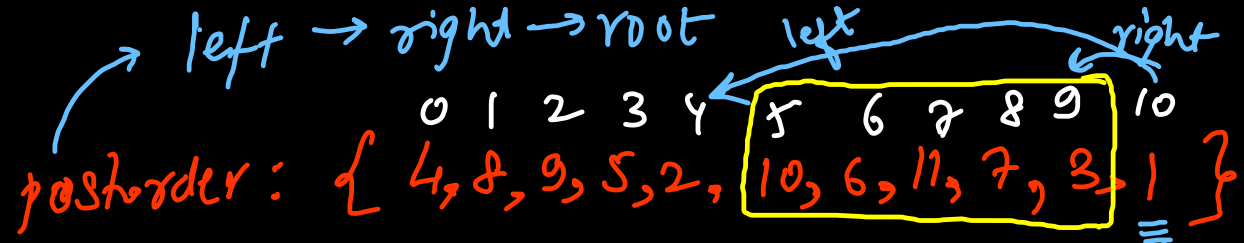
else : $O(n^2)$
worst case

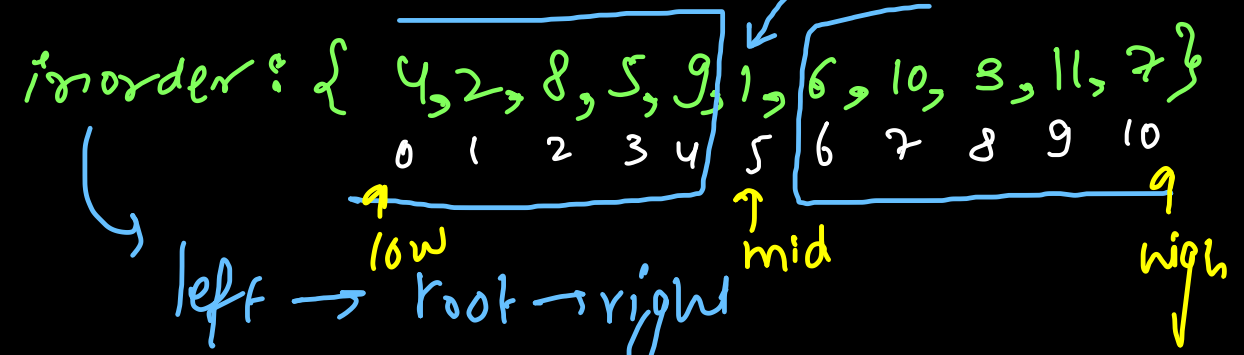
Similar to Divide & conquer !

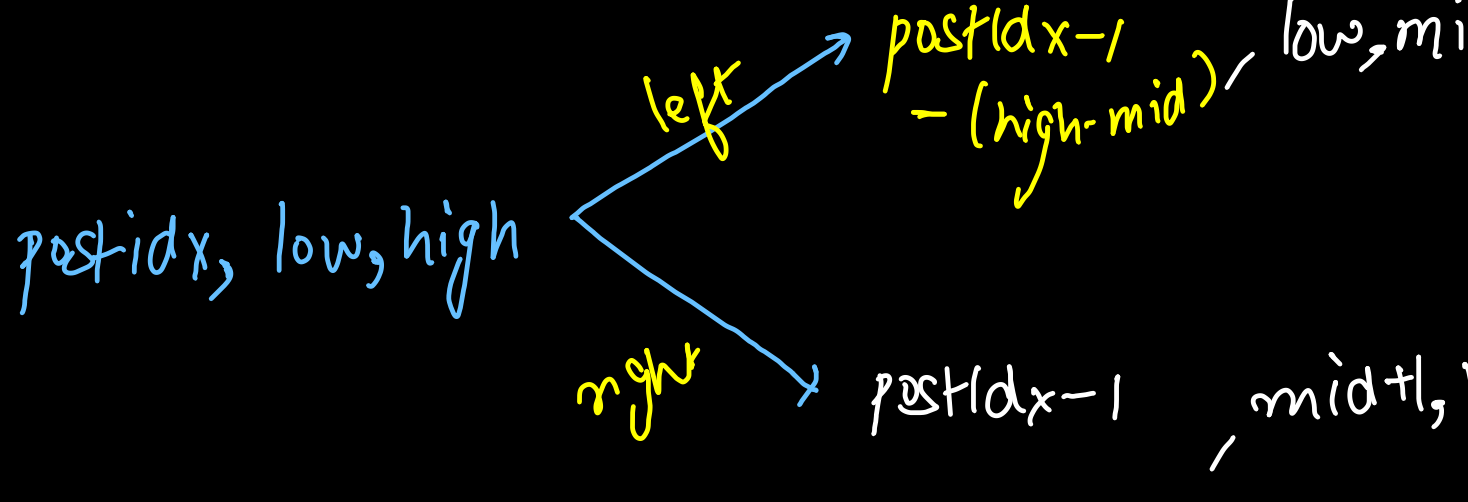
LC 106)



left → right → root

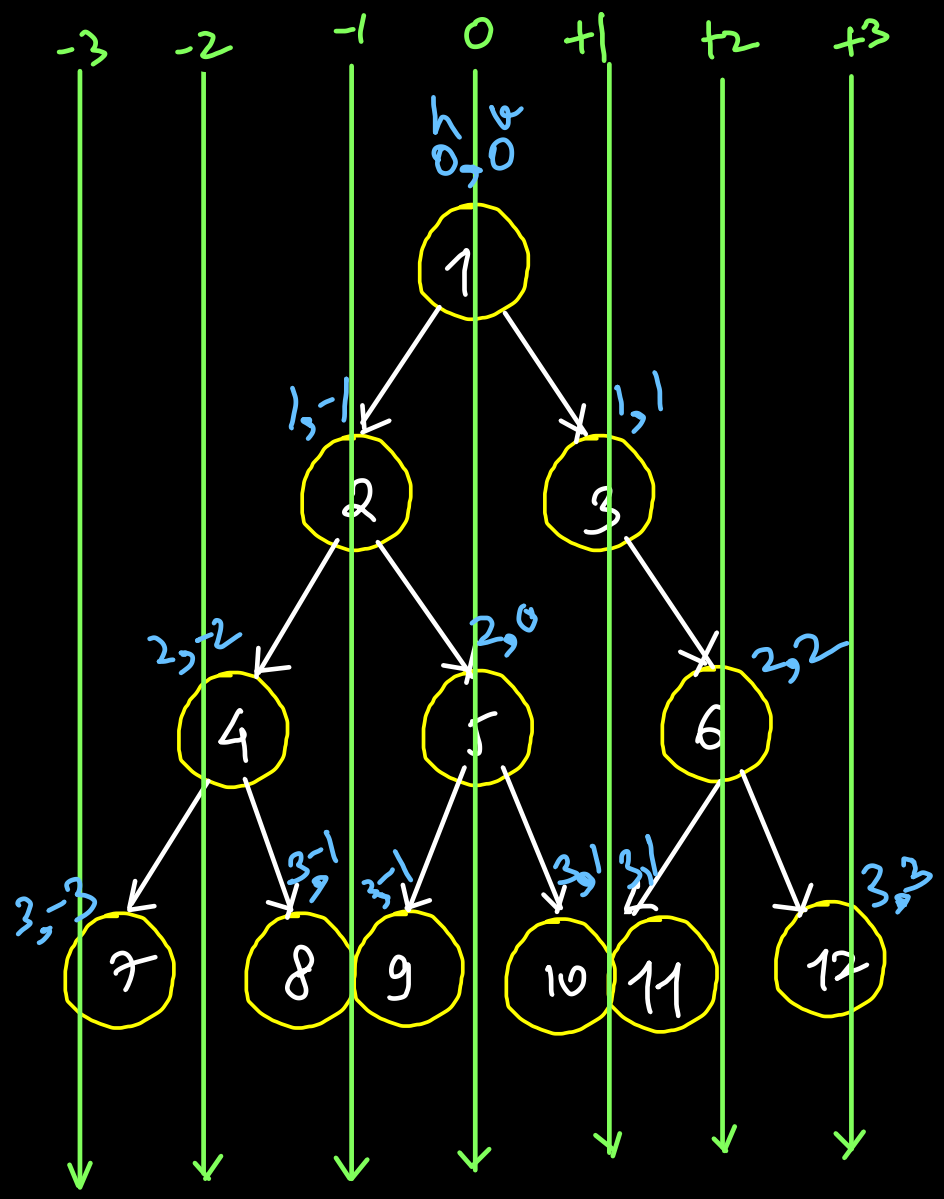


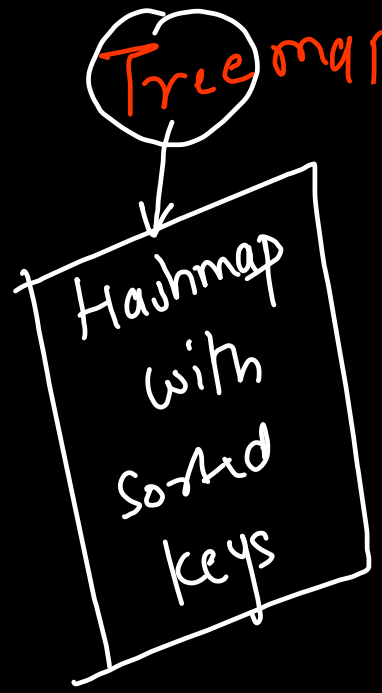




```java
class Solution {
    HashMap<Integer, Integer> in = new HashMap<>();

    public TreeNode construct(int[] postorder, int i, int l, int r){
        if(l > r) return null;
        TreeNode root = new TreeNode(postorder[i]);

        int j = in.get(postorder[i]);
        root.left = construct(postorder, i - (r - j) - 1, l, j - 1);
        root.right = construct(postorder, i - 1, j + 1, r);

        return root;
    }

    public TreeNode buildTree(int[] inorder, int[] postorder) {
        for(int i = 0; i < inorder.length; i++)
            in.put(inorder[i], i);
        return construct(postorder, postorder.length - 1, 0, inorder.length - 1);
    }
}
```

Vertical Traversal of Binary Tree





ArrayList<Integer>>

0 → { 1, 5 }

-1 → { 2, 8, 9 }

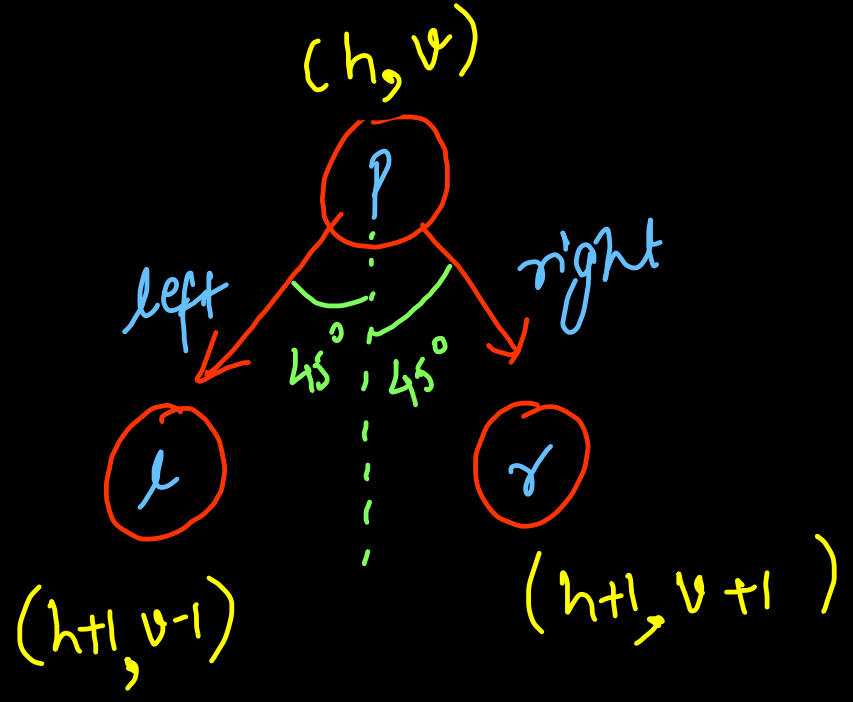
-2 → { 4 }

-3 → { 7 }

1 → { 3, 10, 11 }

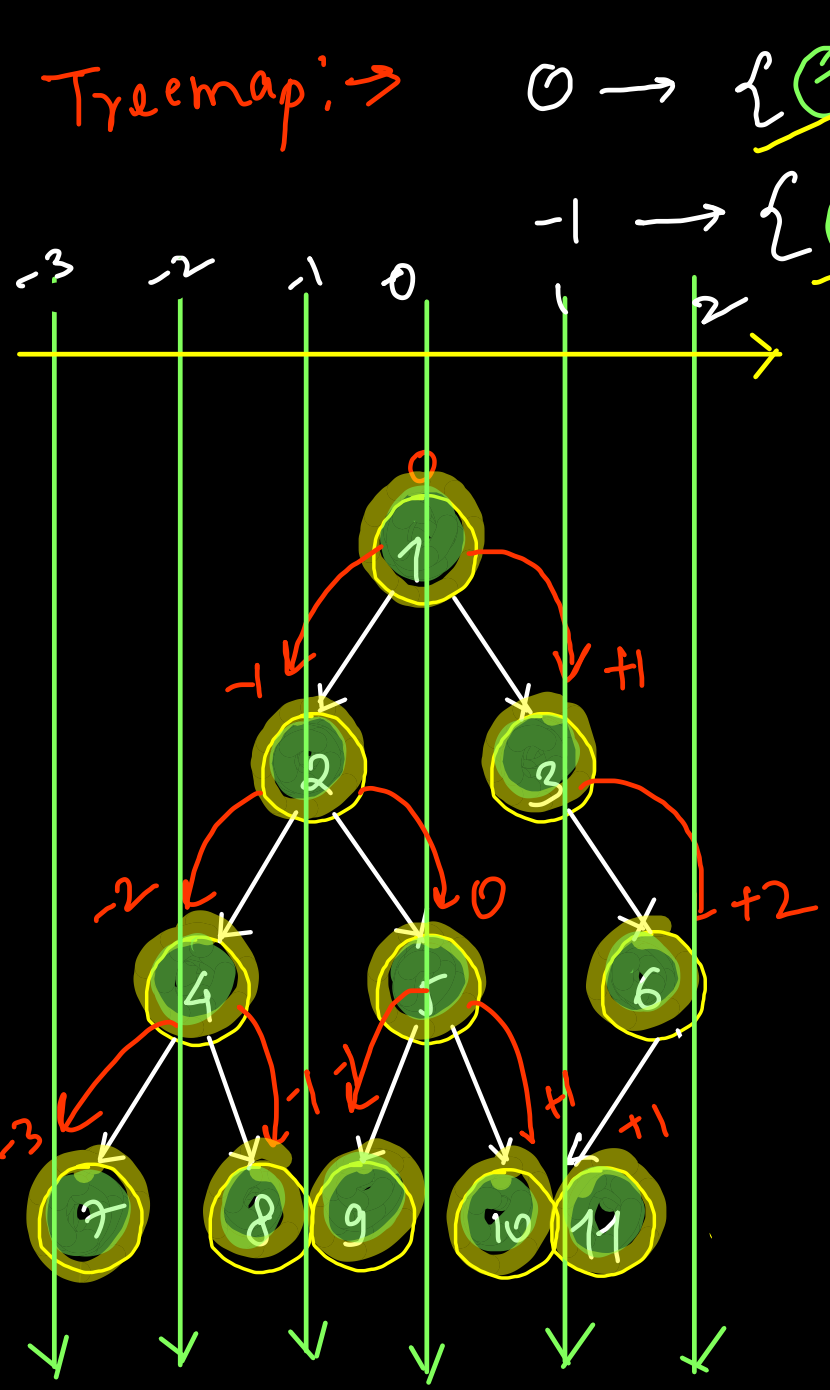
2 → { 6 }

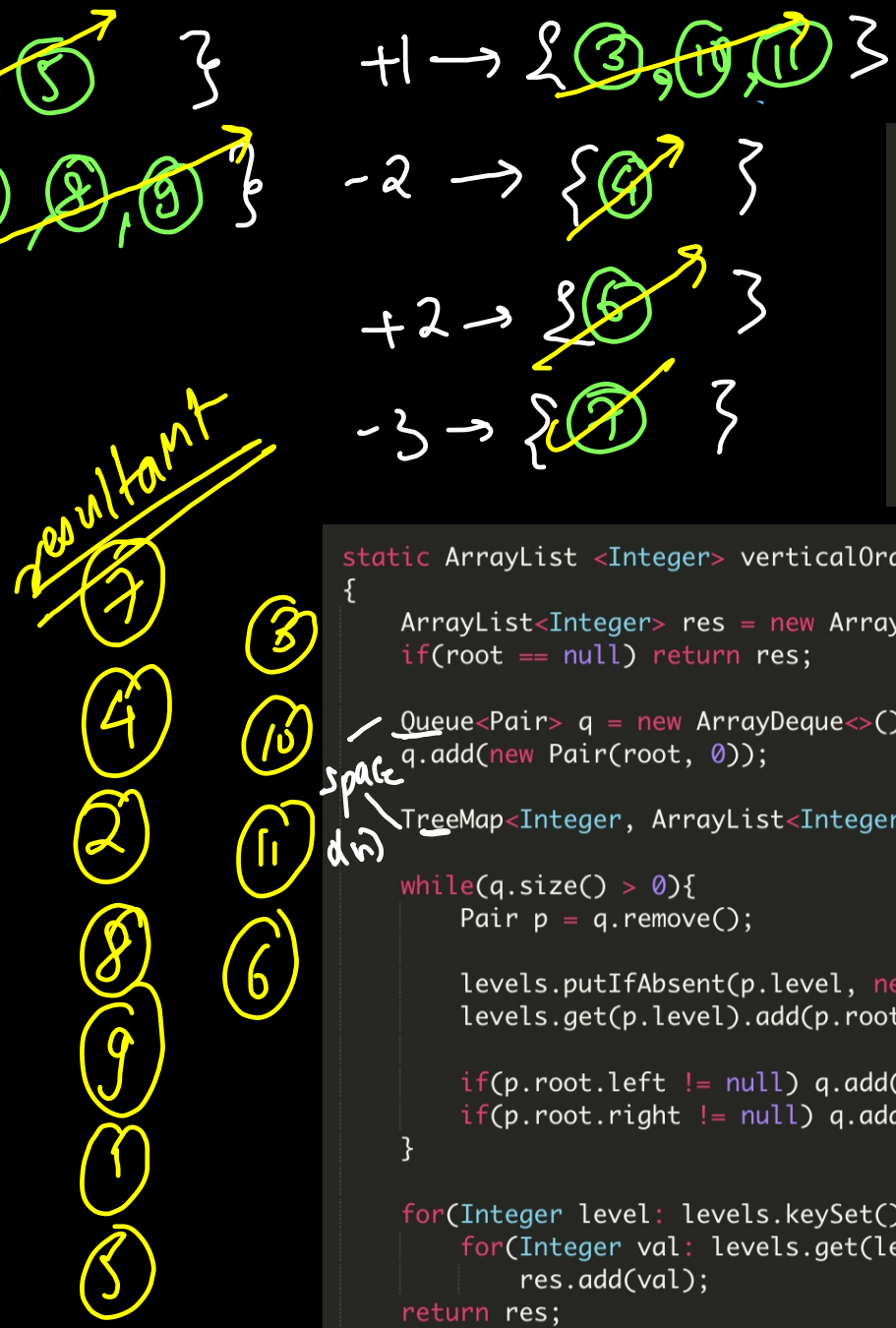
3 → { 12 }



p.horizontal = h

p.level = v






```java
static class Pair{
    int level;
    Node root;

    Pair(Node root, int level){
        this.root = root;
        this.level = level;
    }
}
```

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

    Queue<Pair> q = new ArrayDeque<>();
    q.add(new Pair(root, 0));

    TreeMap<Integer, ArrayList<Integer>> levels = new TreeMap<>();

    while(q.size() > 0){
        Pair p = q.remove();

        levels.putIfAbsent(p.level, new ArrayList<>());
        levels.get(p.level).add(p.root.data);

        if(p.root.left != null) q.add(new Pair(p.root.left, p.level - 1));
        if(p.root.right != null) q.add(new Pair(p.root.right, p.level + 1));
    }

    for(Integer level: levels.keySet())
        for(Integer val: levels.get(level))
            res.add(val);
    return res;
}
```

DSA based Projects

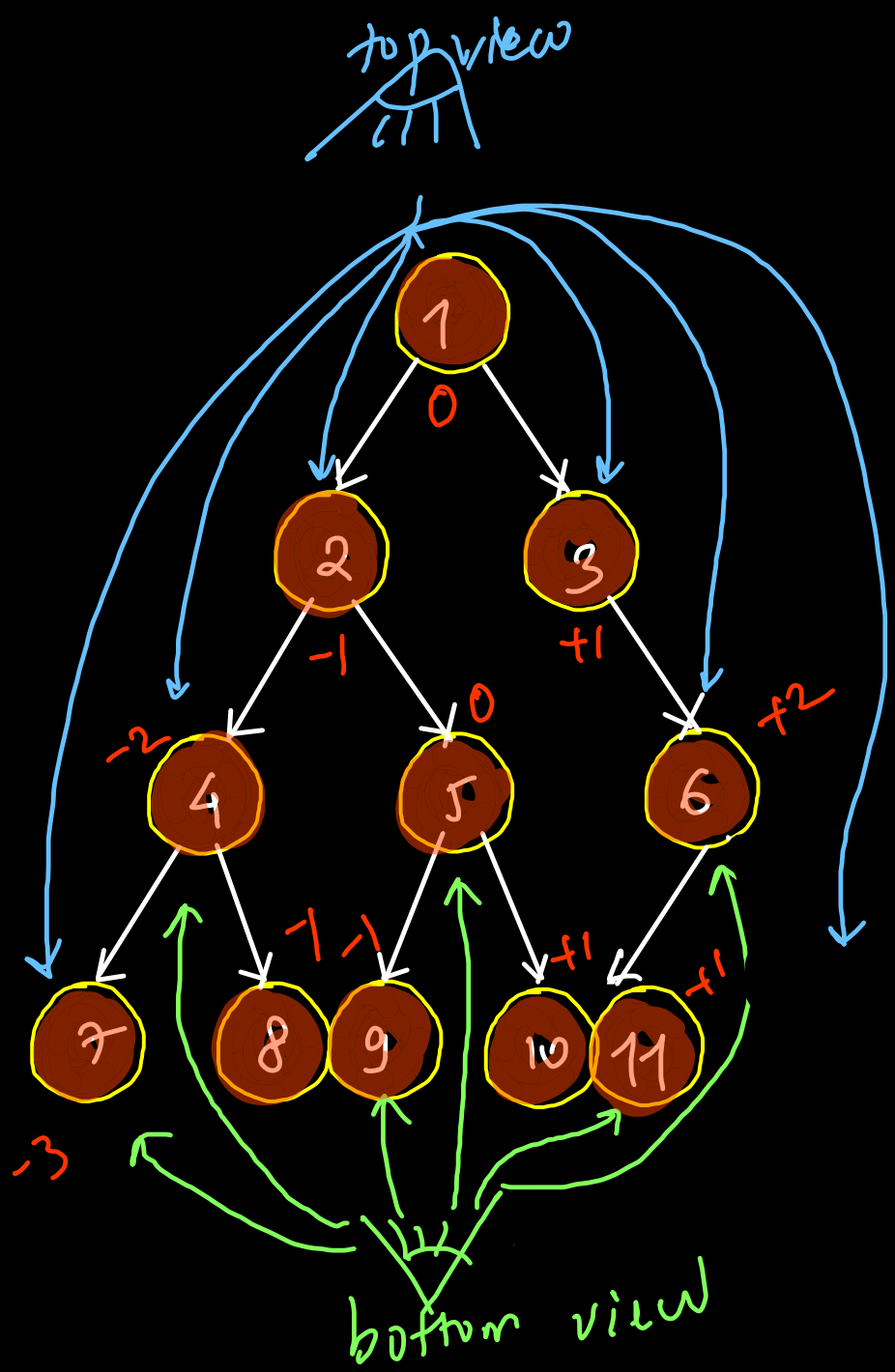
① Library → Trees & Binary Search Tree
- Tries
- Grap

② Huffman Encoding & Decoding → Zipper/compress

③ Games Website
- crossword
- snakes & ladder
- Sudoku
- tic-tac-toe

④ LRU Cache
↓
Chat Applicam

⑤ Splitwise



putIfAbsent

left to right

7, 4, 2, 1, 3, 6, 12

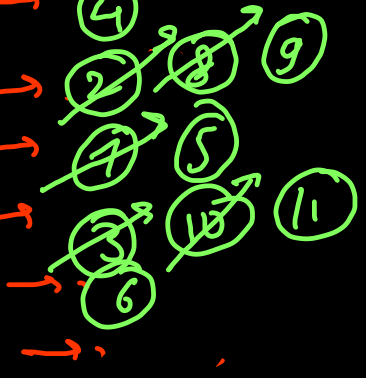
-3 → 7
-2 → 4
-1 → 2
0 → 1
+1 → 3
+2 → 6
+3 →

left to right

7, 4, 9, 5, 11, 6

put

-3 → 7
-2 → 4
-1 → 2
0 → 1
+1 → 3
+2 → 6
+3 →



Top view

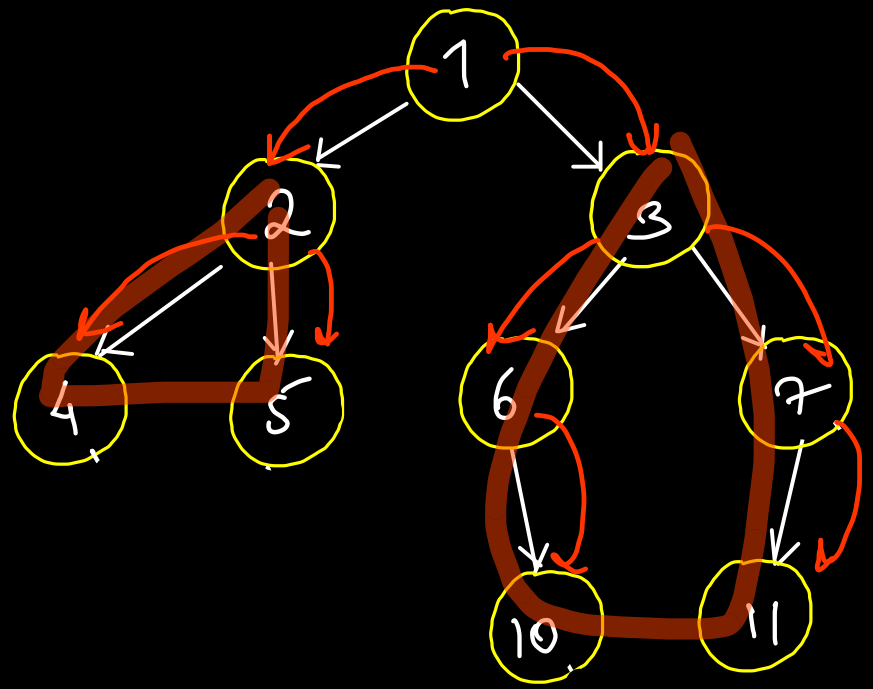
```
static class Pair{
    int level;
    Node root;

    Pair(Node root, int level){
        this.root = root;
        this.level = level;
    }
}

static ArrayList<Integer> topView(Node root)
{
    ArrayList<Integer> res = new ArrayList<>();
    if(root == null) return res;

    Queue<Pair> q = new ArrayDeque<>();
    q.add(new Pair(root, 0));

    TreeMap<Integer, Integer> levels = new TreeMap<>();

    while(q.size() > 0){
        Pair p = q.remove();

        levels.putIfAbsent(p.level, p.root.data);
        if(p.root.left != null) q.add(new Pair(p.root.left, p.level - 1));
        if(p.root.right != null) q.add(new Pair(p.root.right, p.level + 1));
    }

    for(Integer key: levels.keySet())
            res.add(levels.get(key));
    return res;

}
```

Bottom view

```
static class Pair{
    int level;
    Node root;

    Pair(Node root, int level){
        this.root = root;
        this.level = level;
    }
}

static ArrayList<Integer> bottomView(Node root)
{
    ArrayList<Integer> res = new ArrayList<>();
    if(root == null) return res;

    Queue<Pair> q = new ArrayDeque<>();
    q.add(new Pair(root, 0));

    TreeMap<Integer, Integer> levels = new TreeMap<>();

    while(q.size() > 0){
        Pair p = q.remove();

        levels.put(p.level, p.root.data);
        if(p.root.left != null) q.add(new Pair(p.root.left, p.level - 1));
        if(p.root.right != null) q.add(new Pair(p.root.right, p.level + 1));
    }

    for(Integer key: levels.keySet())
            res.add(levels.get(key));
    return res;

}
```

O(n) time, O(n) space

#606 Construct String from Tree



1 (2 (4) (5)) (3 (6) (10)) (7(11))

leaf node

only right child

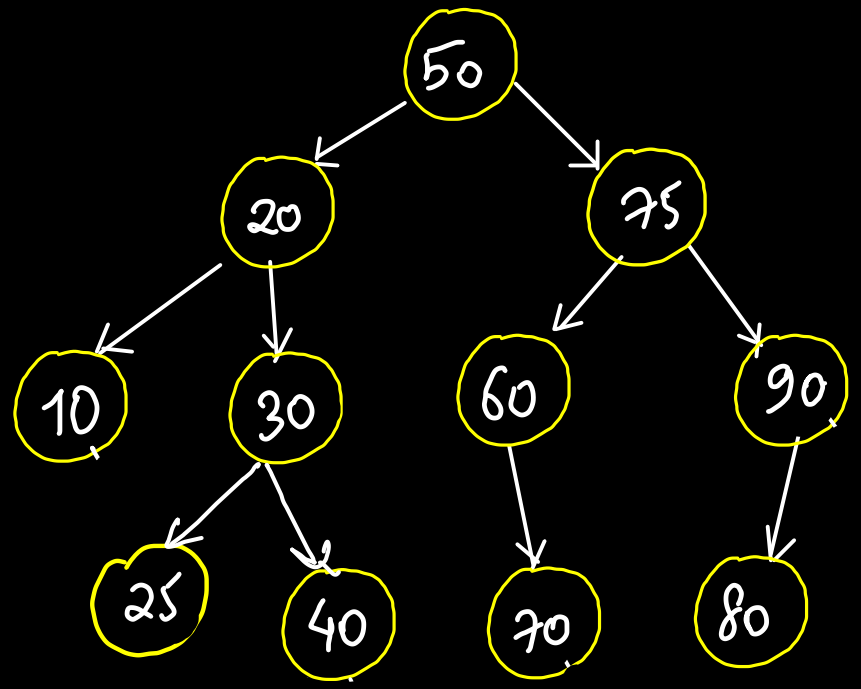
only left child

```java
public String tree2str(TreeNode root) {
    if(root == null) return "";
    if(root.left == null && root.right == null)
        return Integer.toString(root.val); // leaft
    if(root.right == null){
        String left = tree2str(root.left);
        return root.val + "(" + left + ")";
    }
    if(root.left == null){
        String right = tree2str(root.right);
        return root.val + "()" + "(" + right + ")";
    }

    String left = tree2str(root.left);
    String right = tree2str(root.right);
    return root.val + "(" + left + ")" + "(" + right + ")";
}
```

assuming string operations are O(1)

Time = O(n)
space = O(h)

Binary Search Tree



⇒ Binary Tree + Searching

⇒ Binary Search + Tree
(Divide & Conquer)

① For every node

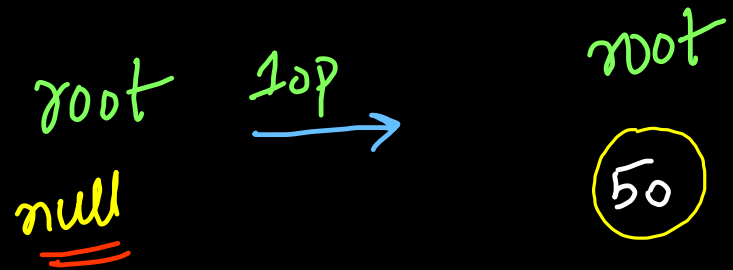
left subtree < root < right node

② Inorder Traversal ⇒ Increasing order

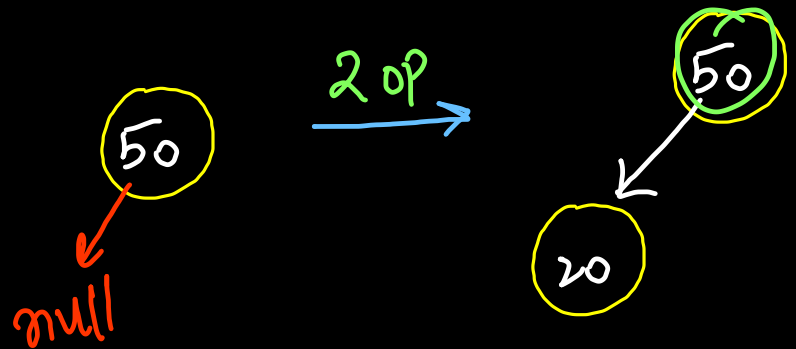
10, 20, 25, 30, 40, 50, 60, 70, 75, 80, 90

Insertion into BST { Will be always at leaf }

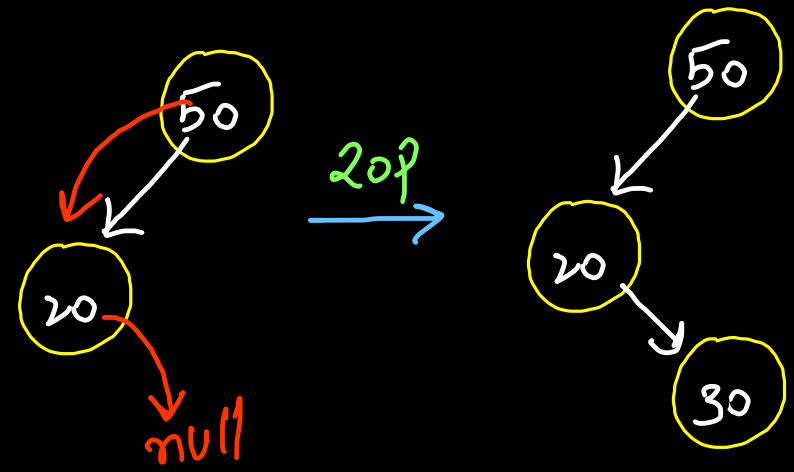
1) insert 50



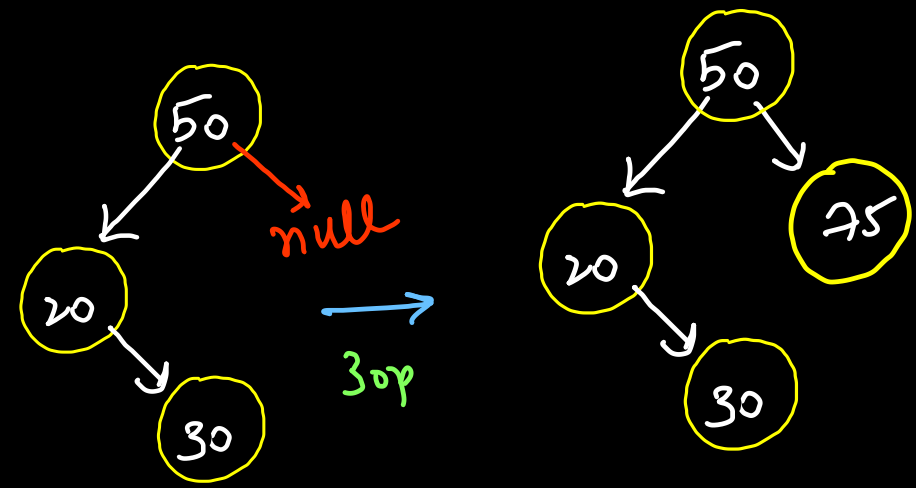
2) insert 20



3) insert 30



4) insert 75



5) insert 70

Total insertion

$\Rightarrow 1\times1 + 2\times2 + 3\times4 + 4\times8$

$\Rightarrow \sum h * 2^{h-1}$

$\Rightarrow n\log n$ for n nodes

Avg case $\Rightarrow O(\log n)$ per insert

balanced bst

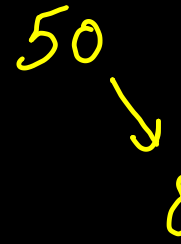
```
public TreeNode insertIntoBST(TreeNode root, int val) {
    if(root == null) return new TreeNode(val);
    if(val < root.val) root.left = insertIntoBST(root.left, val);
    else root.right = insertIntoBST(root.right, val);
    return root;
}
```

Worst case analysis (skewed binary tree)

1) insert 50

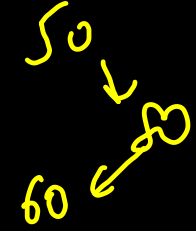
1 op

2) insert 80



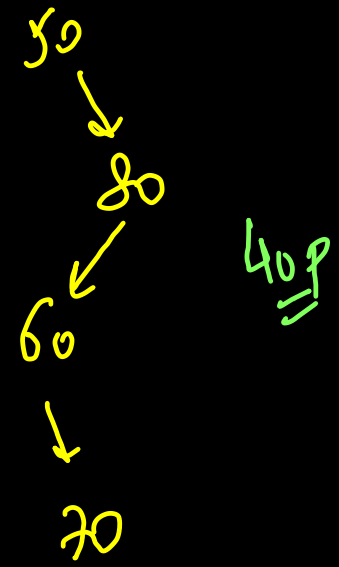
2 op

3) insert 60



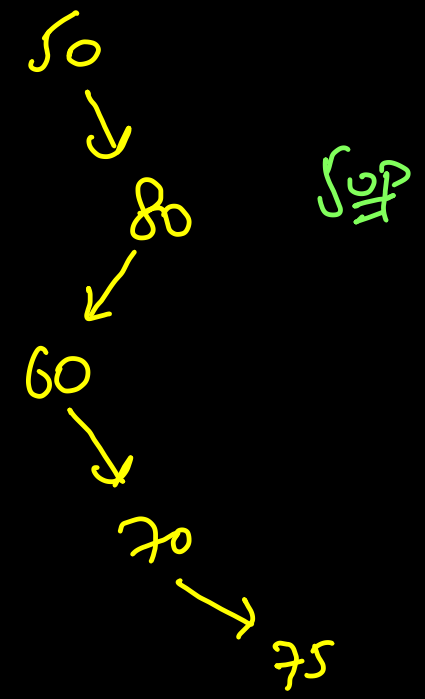
3 op

4) insert 70



4 op

5) insert 75



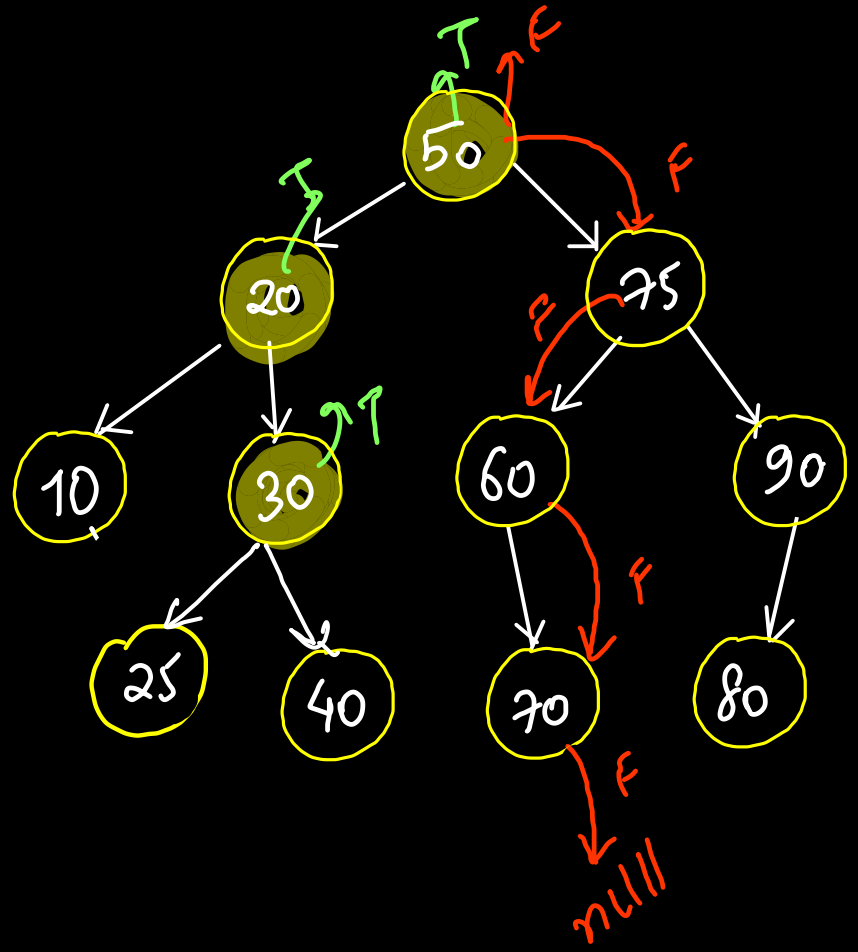
5 op

Worst case total = $1+2+3+4+\cdots n$

$= \frac{n \times (n+1)}{2} = O(n^2)$

single insertn $\Rightarrow O(n^2/n) = O(n)$

Search in BST LC 700



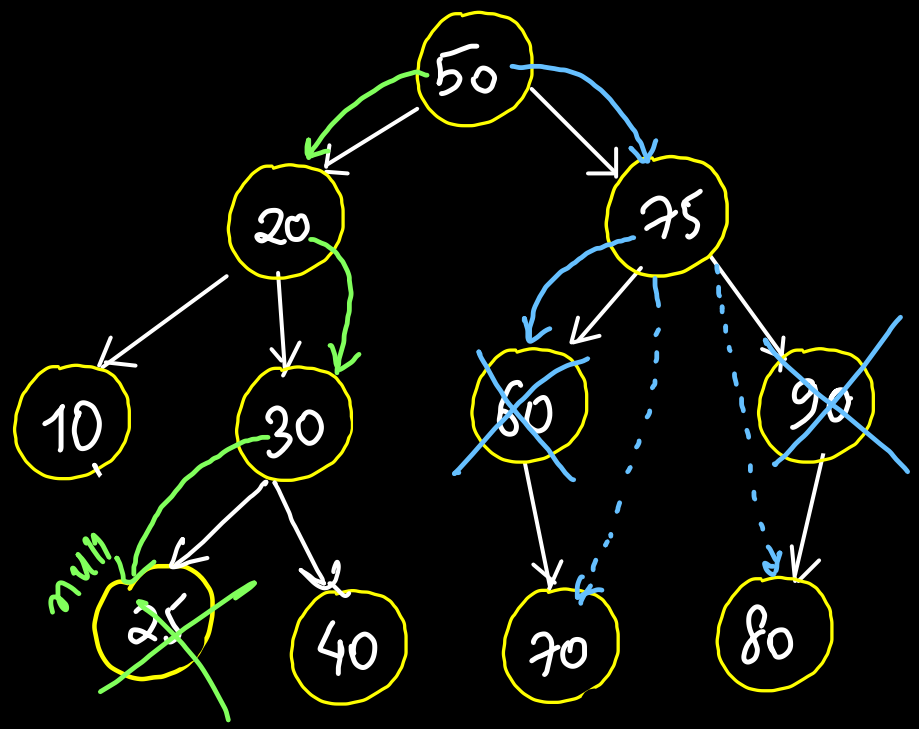
Search (30) ⇒ true

search(72) ⇒ false (null)

```
public TreeNode searchBST(TreeNode root, int target) {
    if(root == null || root.val == target) return root;
    if(target < root.val) return searchBST(root.left, target);
    return searchBST(root.right, target);
}
```

time O(h) → O(logn) balanced / O(n) skewed

space

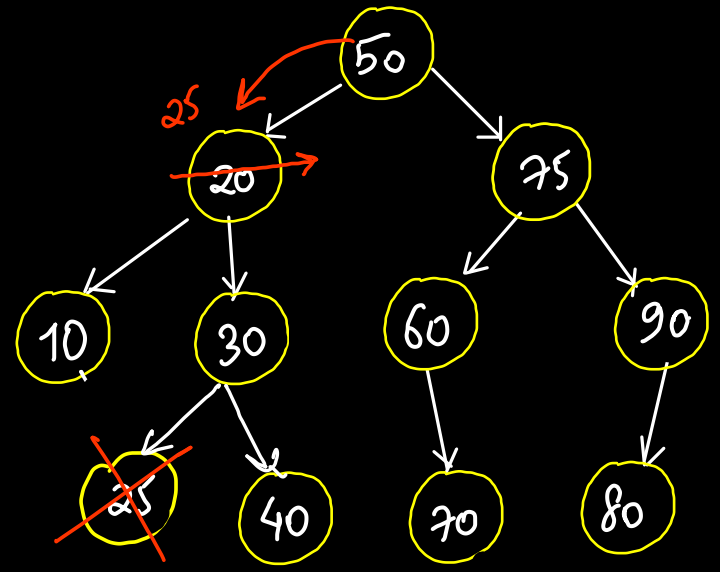


LC (450) Deletn in BST

Node not present
Case1 delete 100
⇒ nothing happens (node not found)

Leaf node
Case2 delete 25
⇒ directly delete it (replace with null)

only one child present
Case3 delete 60
delete 90
⇒ delete node (replace with the child)



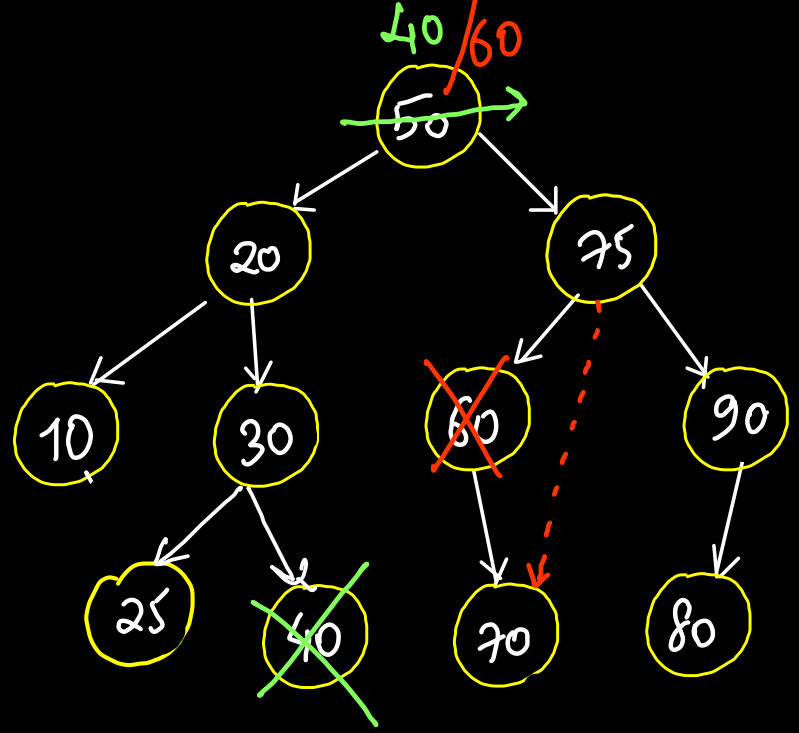
both child present

delete 20

= replace (x)
+ delete(x)

x = just smaller (floor) = 10

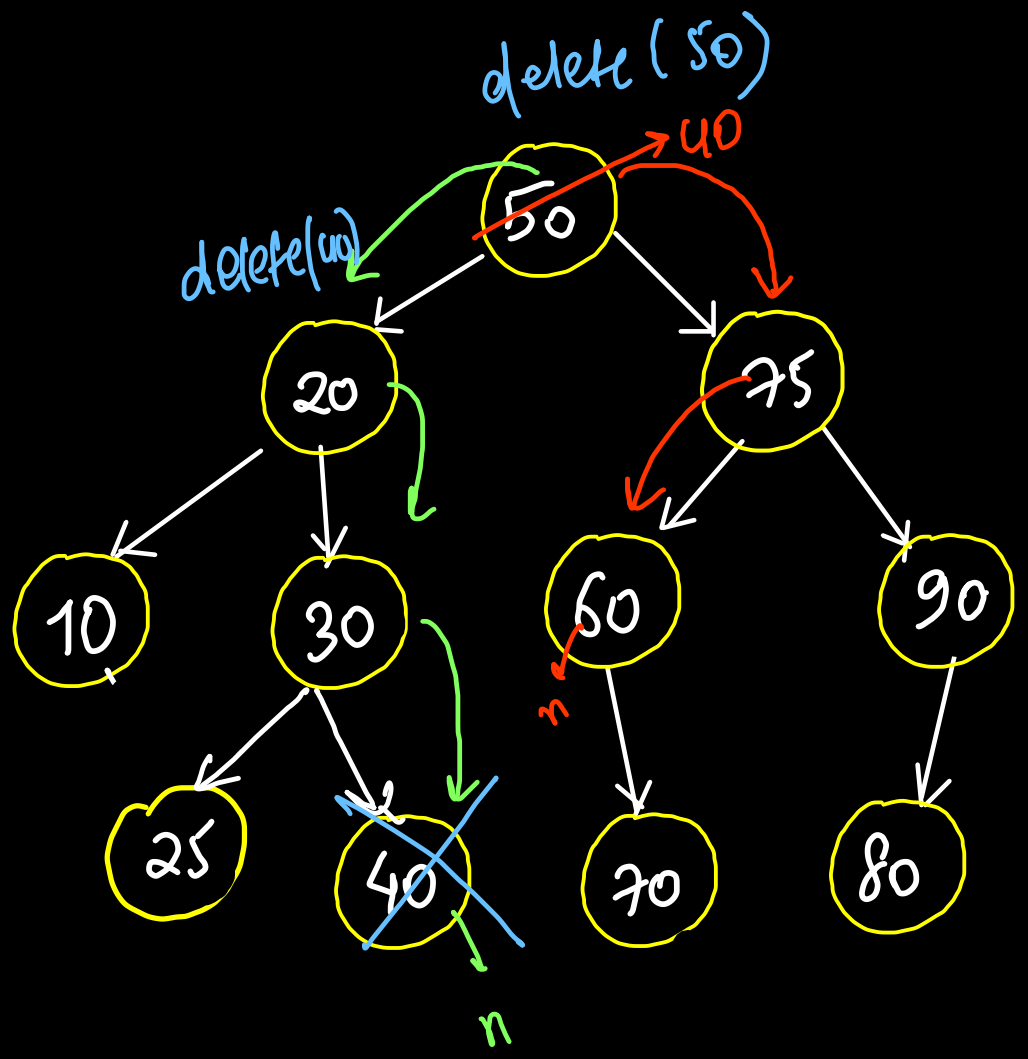
or just larger (ceil) = 25



delete 50

= replace (x) + delete (x)

x → floor = 40
or
ceil = 60



Sorted inorder predecessor

↑

floor = maxm node in root.left

or

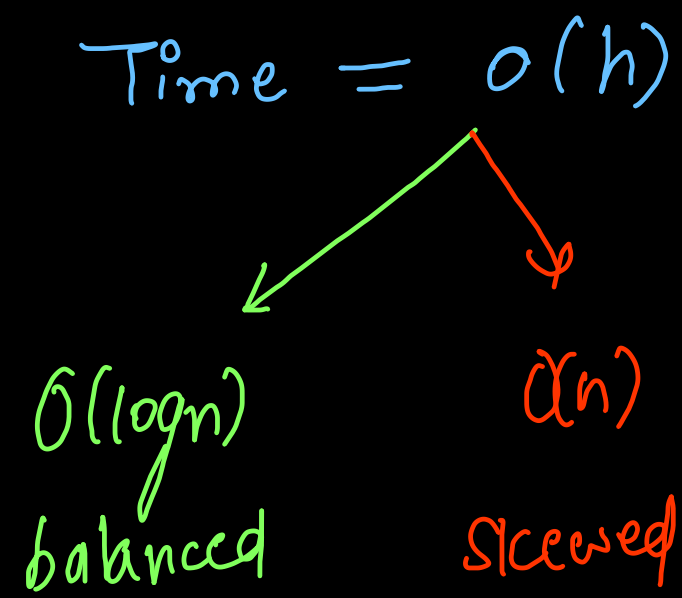
=

ceil = minm node in root.right

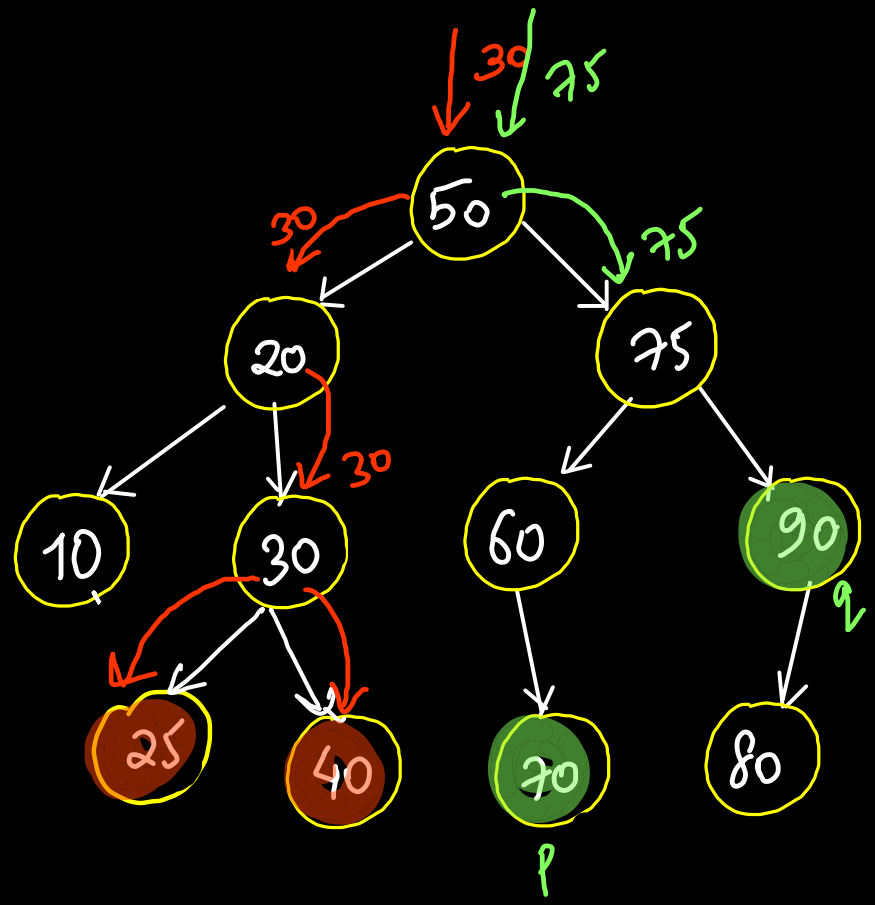
↓

Sorted inorder successor

```java
class Solution {
    public int floor(TreeNode root){
        root = root.left;
        while(root.right != null){
            root = root.right;
        }
        return root.val;
    }
    public TreeNode deleteNode(TreeNode root, int key) {
        if(root == null) return null;
        if(key < root.val) root.left = deleteNode(root.left, key);
        if(key > root.val) root.right = deleteNode(root.right, key);
        if(key == root.val){
            if(root.left == null) return root.right;
            if(root.right == null) return root.left;
            else {
                root.val = floor(root);
                root.left = deleteNode(root.left, root.val);
            }
        }
        return root;
    }
}
```



LC 235 Lowest Common Ancestor



LCA(30, 10) = 20

LCA(70, 80) = 75

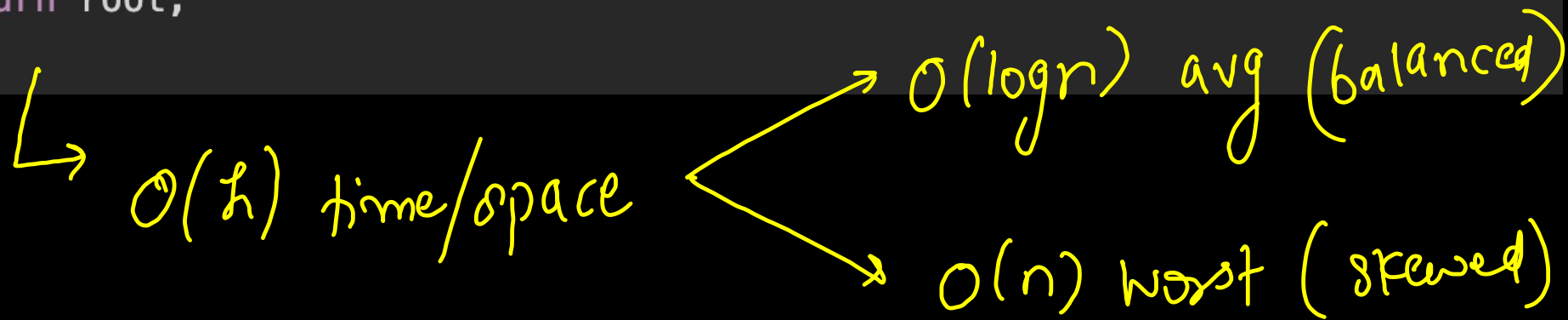
LCA(25, 60) = 50

LCA(30, 40) = 30

App1) Brute force :→ dfs O(n)

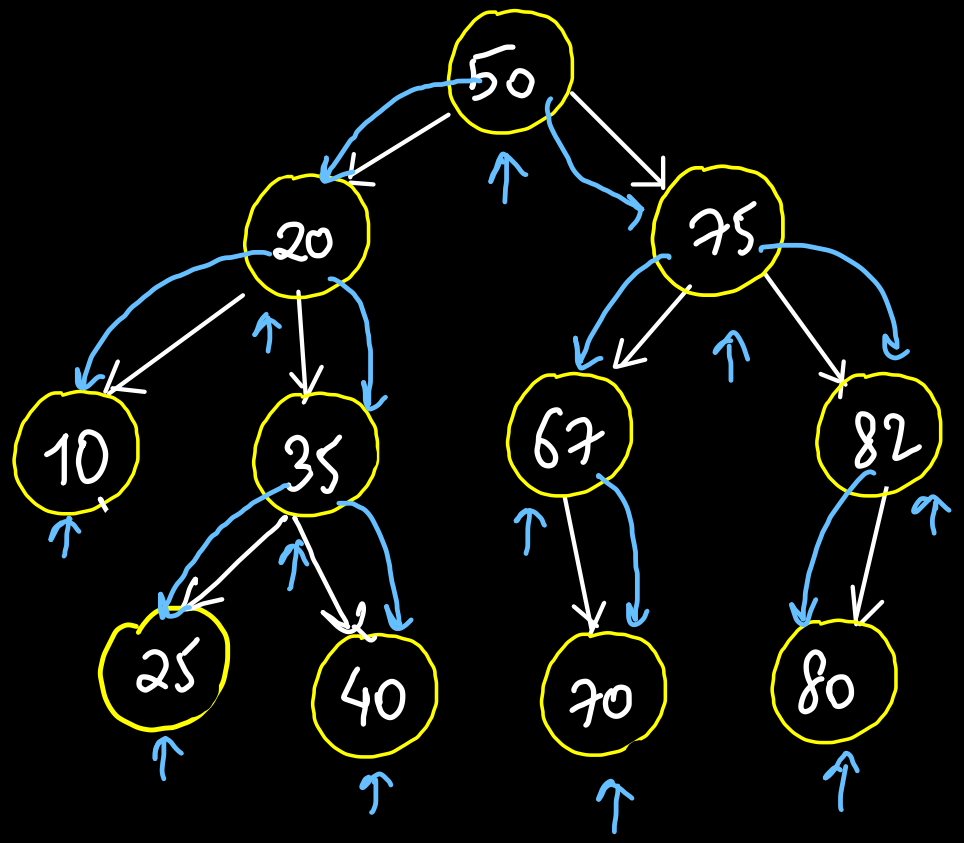
App2)

```java
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
    if(p.val < root.val && q.val < root.val)
        return lowestCommonAncestor(root.left, p, q);
    if(p.val > root.val && q.val > root.val)
        return lowestCommonAncestor(root.right, p, q);
    return root;
}
```



LC 530 Min Absolute Difference in BST

O(n) inorder → sorted increasing



prev = ~~-∞~~ ~~10~~ ~~20~~ ~~25~~ ~~35~~ ~~40~~ ~~50~~ ~~67~~ ~~70~~ ~~75~~ ~~80~~ ~~82~~

minDiff = ~~+∞~~ (20-10) (25-20) (70-67) (82-80)

```java
int prev = (int)-1e8, diff = (int)1e8;

public void inorder(TreeNode root){
    if(root == null) return;
    inorder(root.left);
    diff = Math.min(diff, root.val - prev);
    prev = root.val;
    inorder(root.right);
}

public int getMinimumDifference(TreeNode root) {
    inorder(root);
    return diff;
}
```

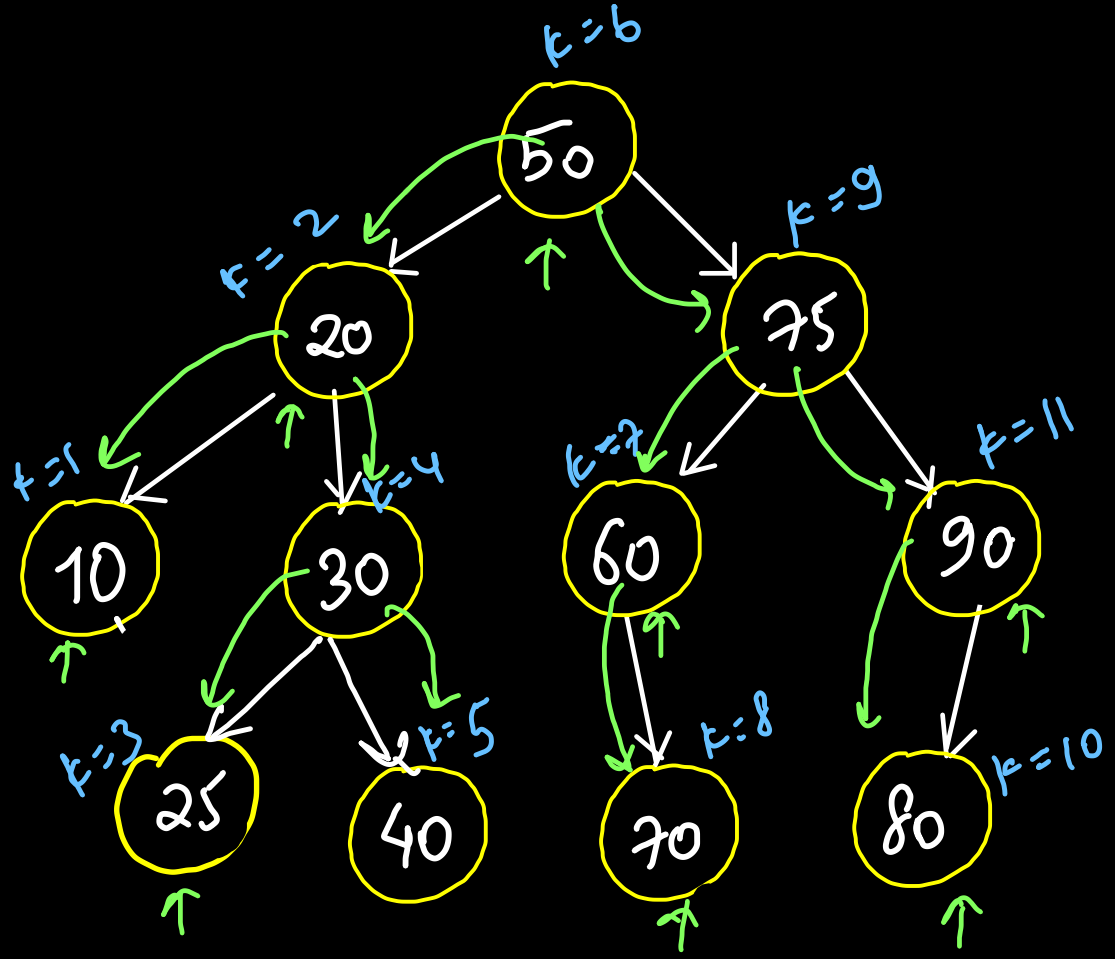
Time = O(n)
dfs

Space = O(h)
recursion call

O(logn)
avg

O(n)
worst

kth smallest Element



k=8 ?

count = ~~0 1 2 3 4 5 6 7 8~~ 9 10 11

ans = ~~7~~ 70

```java
class Solution {
    int count, ans;

    public void inorder(TreeNode root){
        if(root == null) return;
        inorder(root.left);
        count--;
        if(count == 0) ans = root.val;
        inorder(root.right);
    }

    public int kthSmallest(TreeNode root, int k) {
        this.count = k;
        inorder(root);
        return ans;
    }
}
```

Time = O(n)
dfs

Space = O(h)
recursion call

O(logn)
avg

O(n)
worst

LC 1038

BST to GST

(decreasing order)

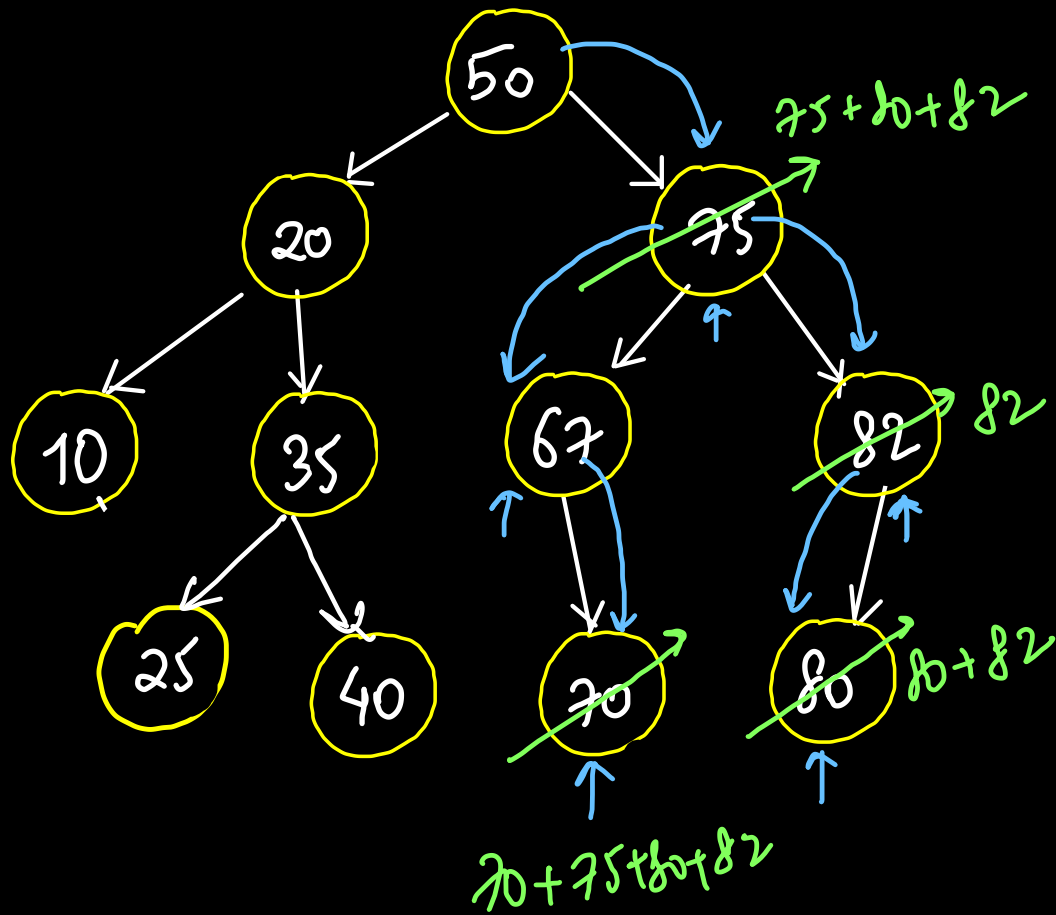
reverse inorder

↳ dfs(root.right)

sum += root.val

root.val = sum

dfs(root.left)



Sum = 0 + 82 + 80 + 75 + 70 + 67

```java
class Solution {
    int sum = 0;

    public void dfs(TreeNode root){
        if(root == null) return;
        dfs(root.right);
        sum += root.val;
        root.val = sum;
        dfs(root.left);
    }

    public TreeNode bstToGst(TreeNode root) {
        dfs(root);
        return root;
    }
}
```

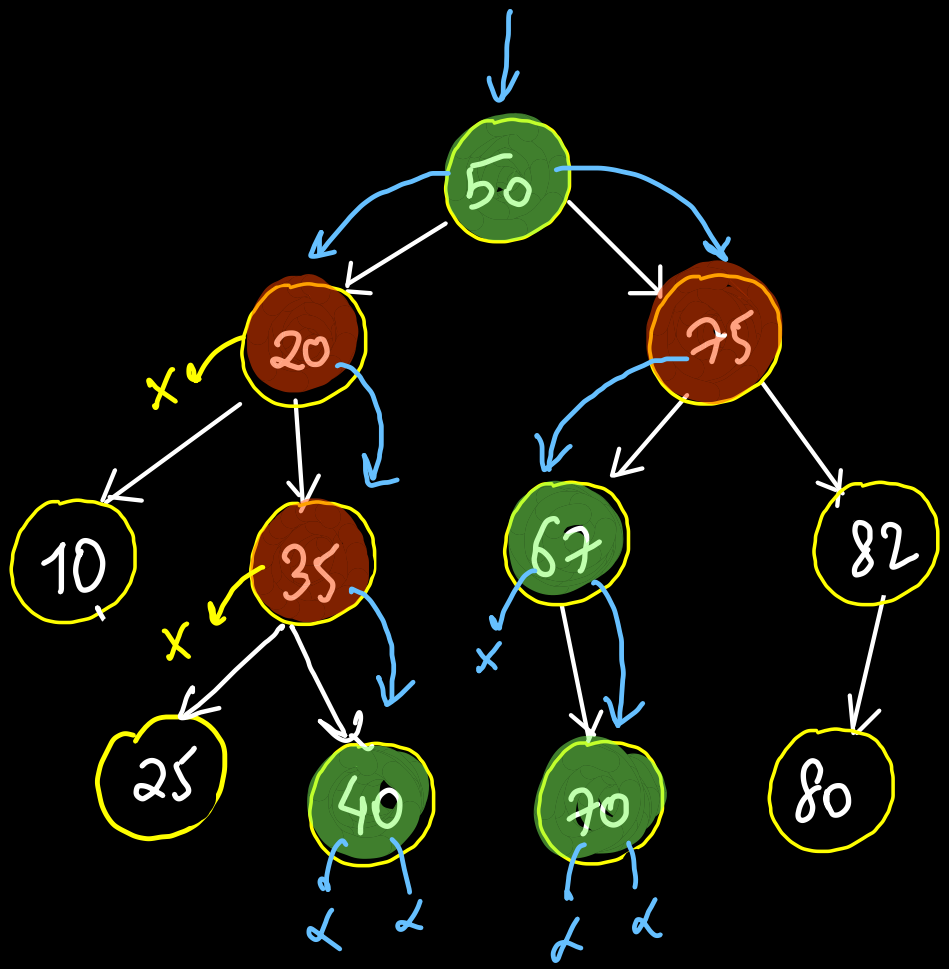
Time = O(n)
dfs

Space = O(h)
recursion call

O(logn)
avg

O(n)
worst

LC 938 Range Sum of BST



Approach1) Normal dfs

↳ O(n) time

Approach2) dfs + prunings

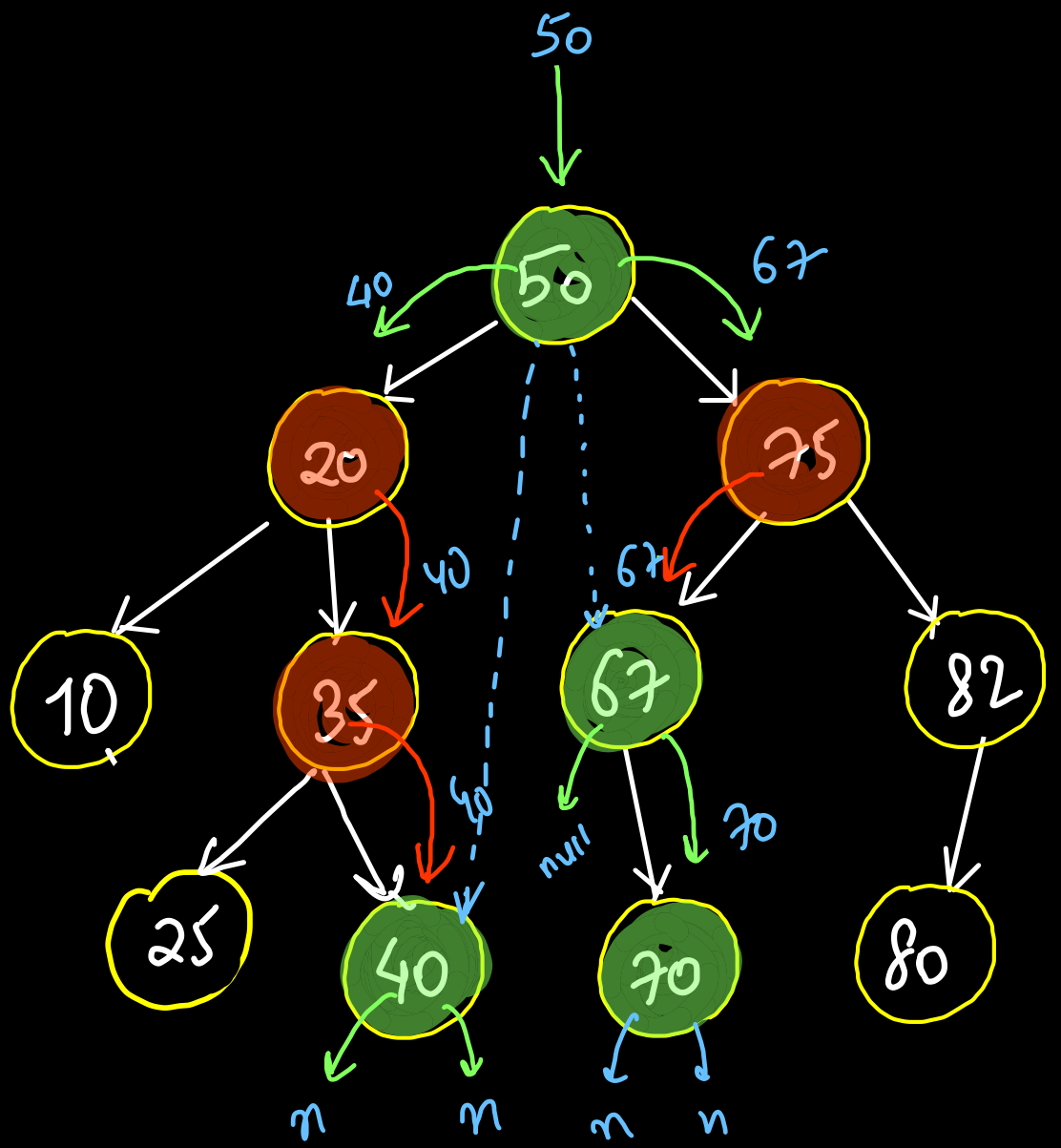
[37, 72]

sum = 0 + 50 + 40 + 67 + 70

```java
public int rangeSumBST(TreeNode root, int low, int high) {
    if(root == null) return 0;
    if(root.val < low) return rangeSumBST(root.right, low, high);
    if(root.val > high) return rangeSumBST(root.left, low, high);

    int sum = root.val;
    sum += rangeSumBST(root.left, low, high);
    sum += rangeSumBST(root.right, low, high);
    return sum;
}
```

O(n) worst case

O(range) avg



Trim a BST LC 669

[37, 72]

```
public TreeNode trimBST(TreeNode root, int low, int high) {
    if(root == null) return null;
C1  if(root.val < low) return trimBST(root.right, low, high);
C2  if(root.val > high) return trimBST(root.left, low, high);

C3  root.left = trimBST(root.left, low, high);
    root.right = trimBST(root.right, low, high);
    return root;
}
```

root is in range if should not be deleted

case1: delete root + left subtree
eg 20

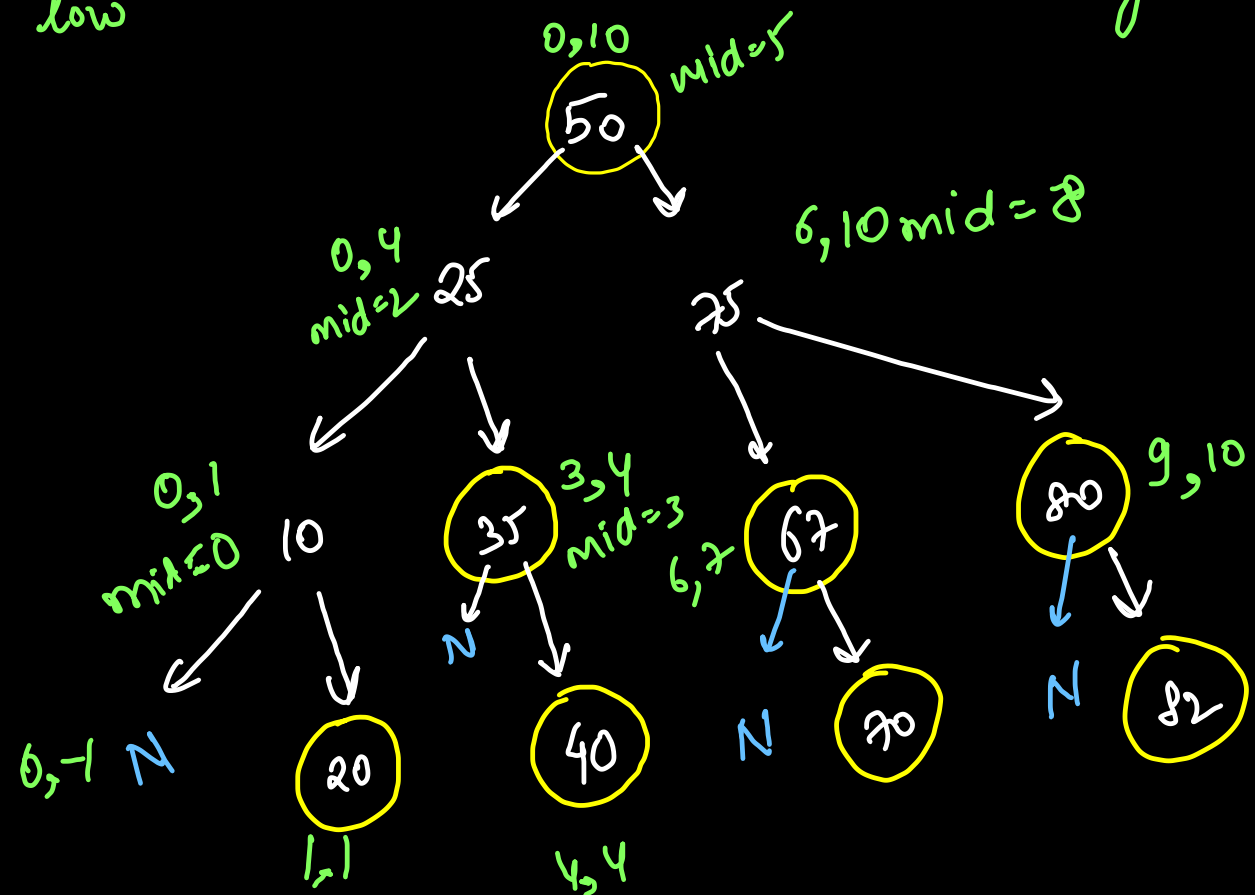
case2: delete root + right subtree
eg 75

Sorted Array to BST → height-balanced (lh == rh ± 1)

#c108

{ 10, 20, 25, 35, 40, 50, 67, 70, 75, 80, 82 }
indices: 0 1 2 3 4 5 6 7 8 9 10

low = 0, mid = 5, high = 10



$T(n) = 2T(n/2) + O(1)$

↓

$T(n) = 2^h \cdot h + 1$

$= 2^{\log_2 n} \cdot \log n$

$= n \log n$

```java
public TreeNode dfs(int[] nums, int low, int high){
    if(low > high) return null;

    int mid = low + (high - low) / 2;
    TreeNode root = new TreeNode(nums[mid]);

    root.left = dfs(nums, low, mid - 1);
    root.right = dfs(nums, mid + 1, high);
    return root;
}
public TreeNode sortedArrayToBST(int[] nums) {
    return dfs(nums, 0, nums.length - 1);
}
```

Time ⇒ O(n)

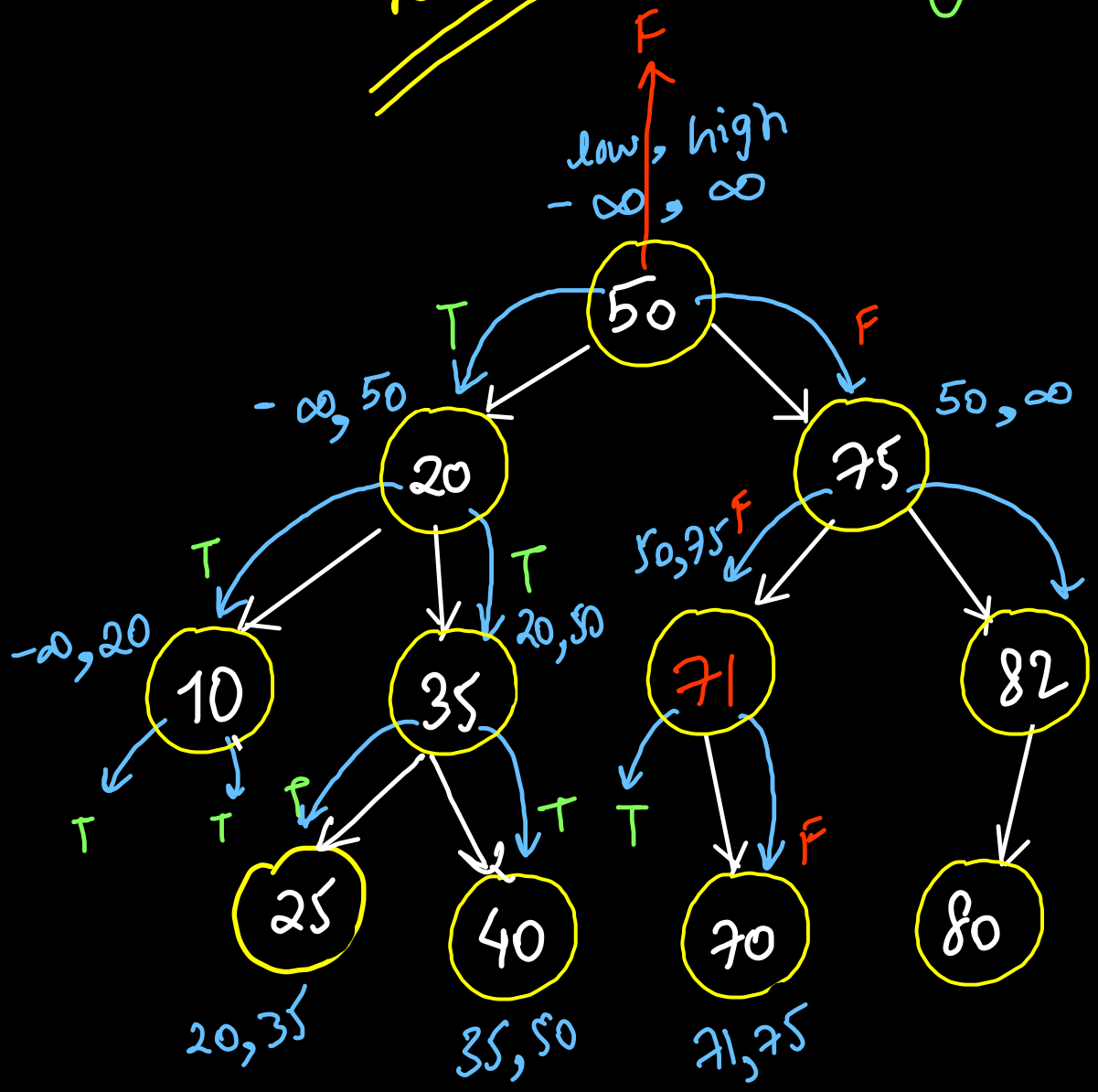
Space ⇒ O(logn)
recursion

Recursion
calls^height + Work x height

$2^{\log_2 n} + O(1) * \log_2 n$

$n + \log_2 n$

LC 98 is binary search tree valid



for each node

left < root < right

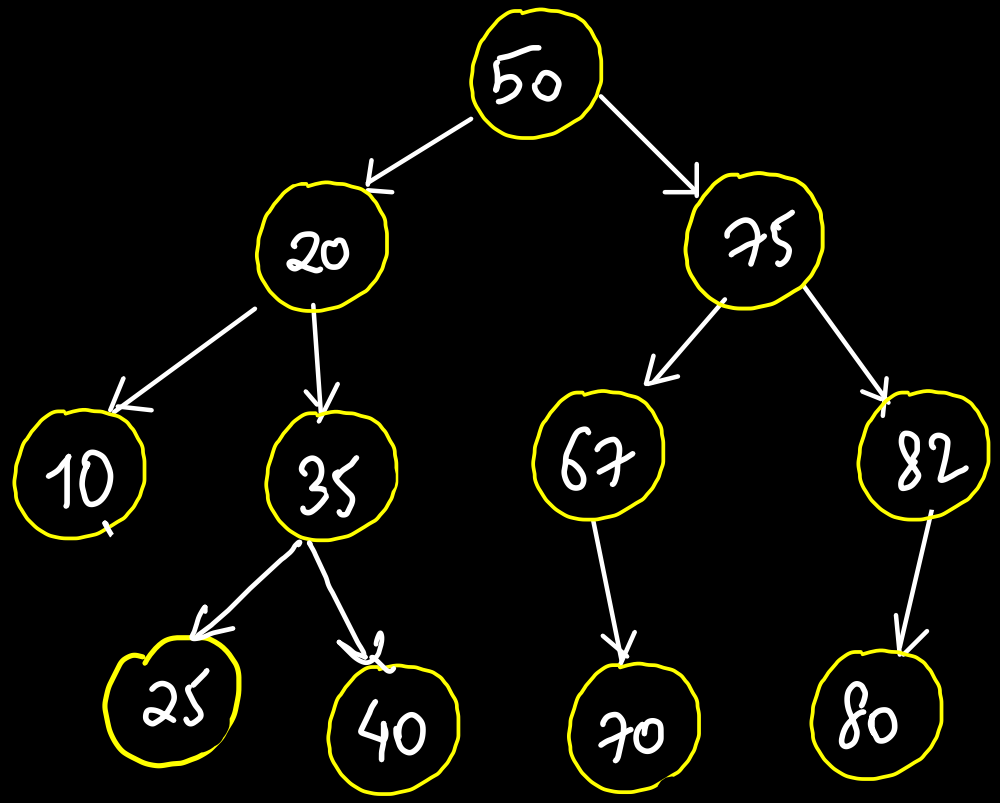
```
public boolean isValidBST(TreeNode root, long low, long high){
    if(root == null) return true;
    if(root.val <= low || root.val >= high) return false;

    boolean isLeftBST = isValidBST(root.left, low, root.val);
    boolean isRightBST = isValidBST(root.right, root.val, high);
    return isLeftBST && isRightBST;
}

public boolean isValidBST(TreeNode root) {
    return isValidBST(root, (long)-1e15, (long)1e15);
}
```

Time = O(n)

Space = O(h)

Iterative DFS Traversal of Binary Tree

→ Preorder/Inorder/Postorder



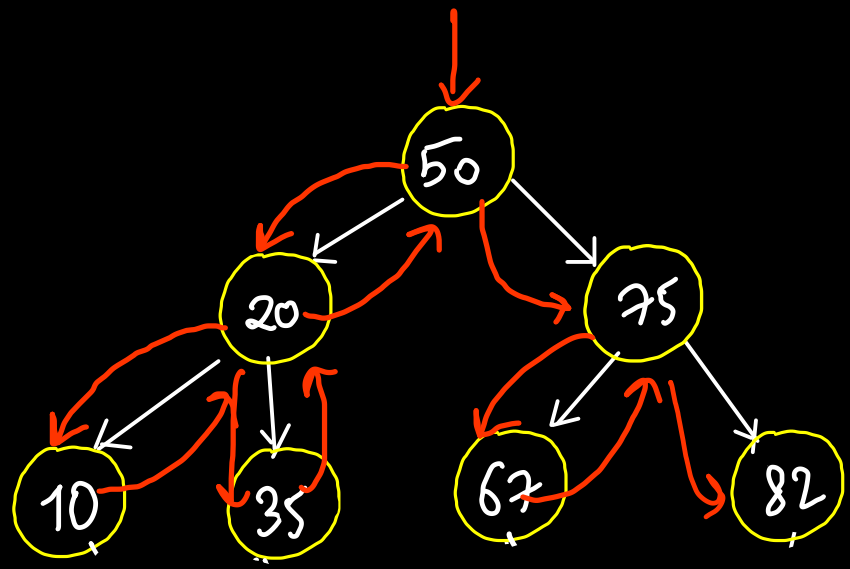
Stack <Pair>

node

Integer

1, 2, 3

pre in postorder

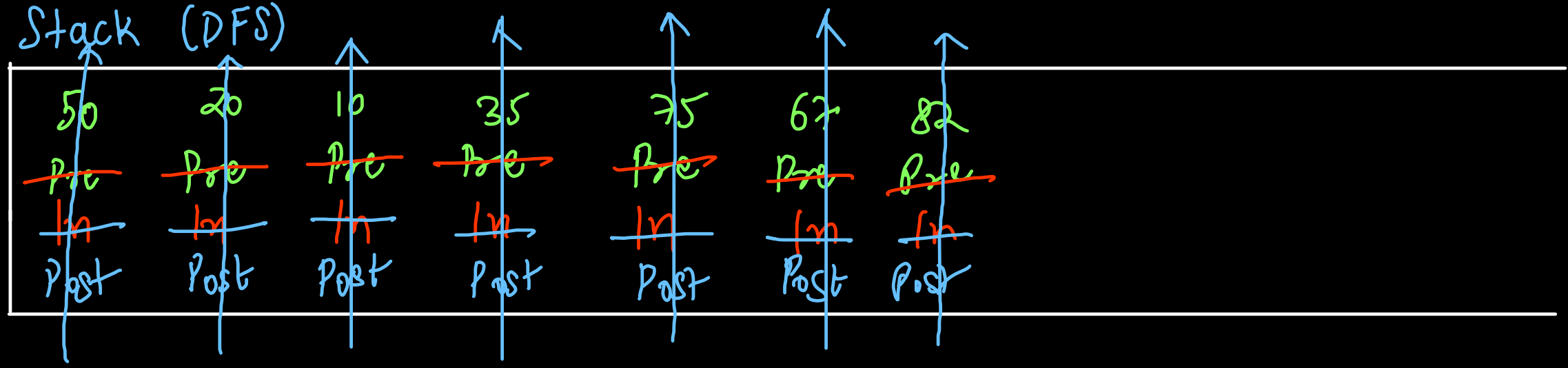


Preorder : 50 20 10 35 75 67 82

Inorder : 10 20 35 50 67 75 82

Postorder : 10 35 20 67 82 75 50

Stack (DFS)



```java
static class Pair{
    TreeNode root;
    int state = 1; // 1 -> preorder, 2 -> inorder, 3 -> postorder
    Pair(TreeNode root) { this.root = root; }
}
```

Time = O(n) dfs

Space = O(h) stackspace

↙ O(log n) avg

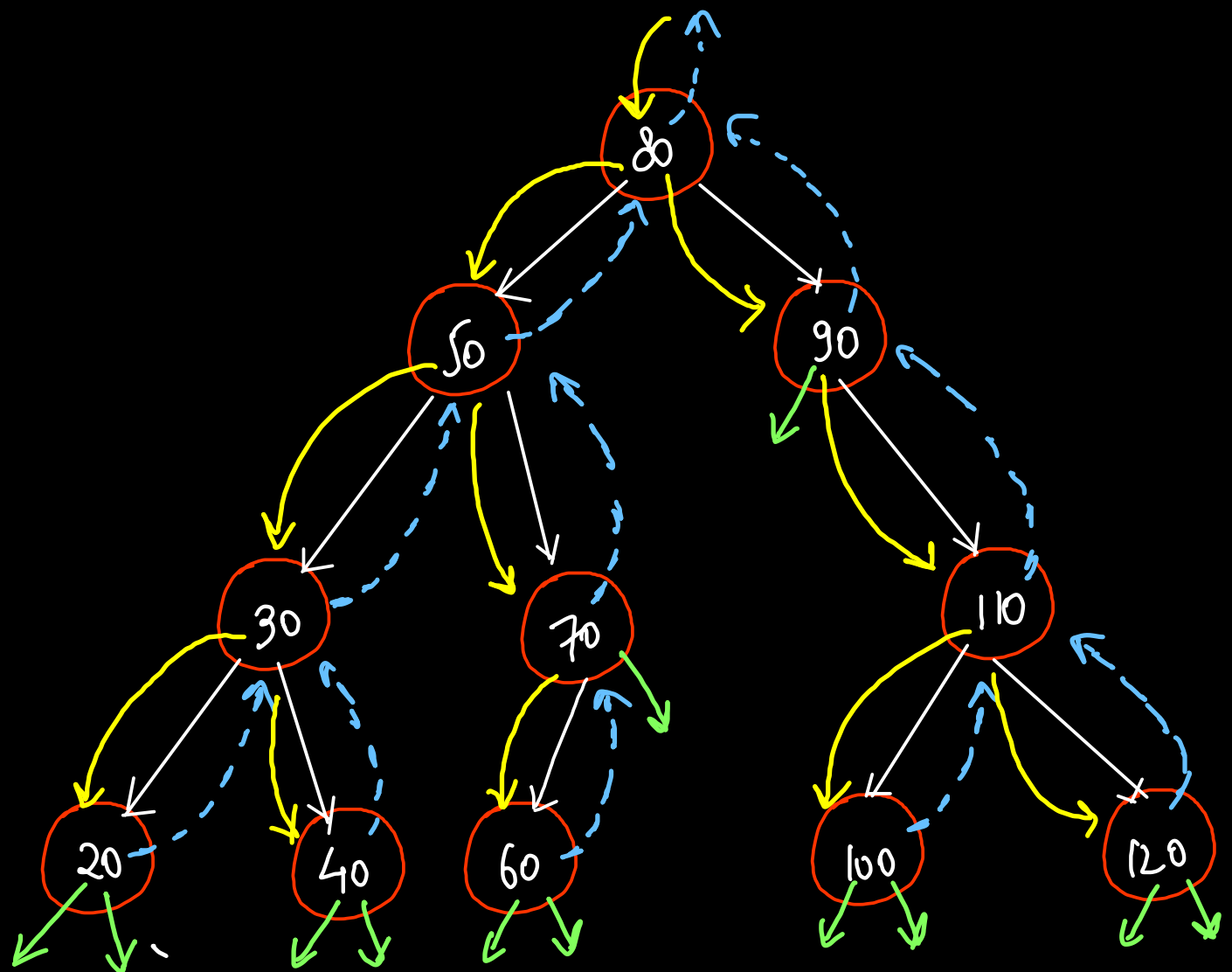
↘ O(n) worst-

No Stack Overflow Exceptions

```java
List<Integer> preorder = new ArrayList<>();
List<Integer> inorder = new ArrayList<>();
List<Integer> postorder = new ArrayList<>();

Stack<Pair> stk = new Stack<>();
if(root != null) stk.push(new Pair(root));

while(stk.size() > 0){
    root = stk.peek().root;
    int state = stk.peek().state++;

    if(state == 1){
        preorder.add(root.val);
        if(root.left != null)
            stk.push(new Pair(root.left));
    }
    else if(state == 2){
        inorder.add(root.val);
        if(root.right != null)
            stk.push(new Pair(root.right));
    }
    else {
        postorder.add(root.val);
        stk.pop();
    }
}
```

Tree Traversals

Recursive DFS	Iterative DFS	BFS or Level order	Morris Traversal DFS (Pre/In/Postorder)
DS: Recursion call stack (at max 20000)	DS: Stack<Pair> (inside heap) (at max 10^5-10^6)	DS: Queue or Linked List (at max 10^5-10^6)	DS: Threaded Binary Tree (same input tree)
Time = O(n)	Time = O(n)	Time = O(n)	Time = O(n) ≈ linear
Space = O(h) → O(logn) / O(n)	Space = O(h) → O(logn) / O(n)	Space = O(2×breadth) = O(n) balanced	Space = O(1) no extra/constant Space

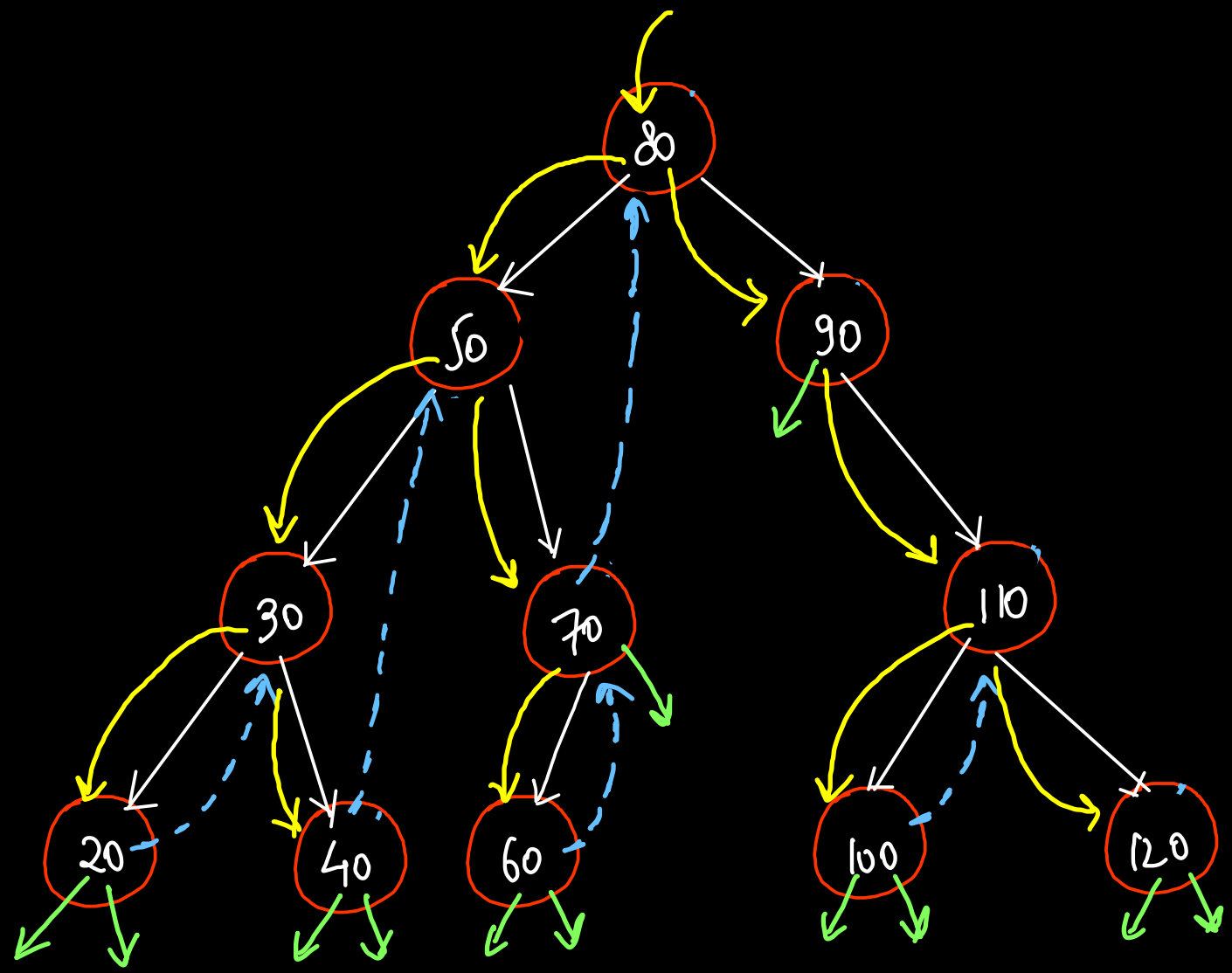


1. dfs(root.left)
2. dfs(root.right)

white → tree edges

yellow → forward edges

blue → backtracking / backward edge



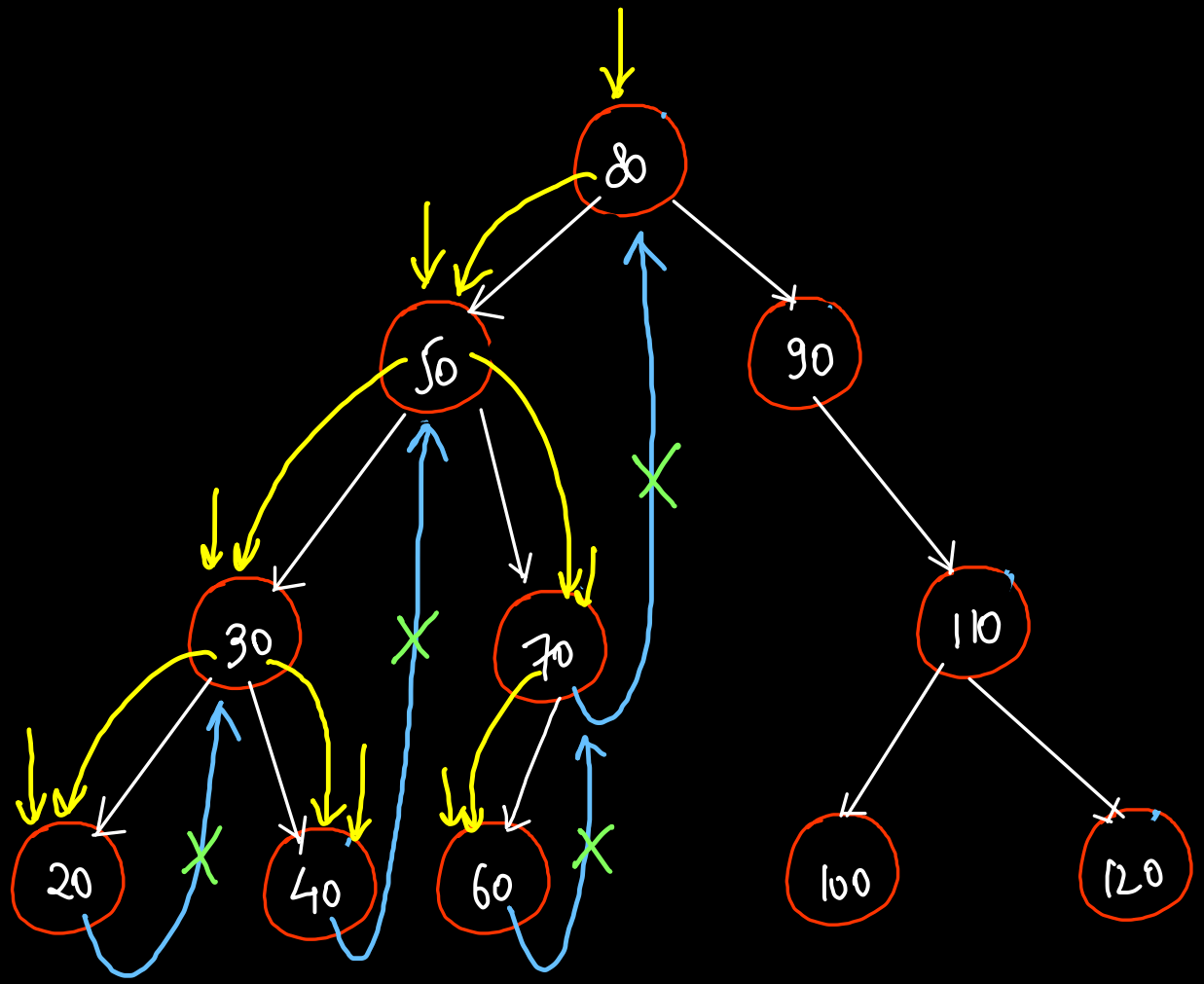
(1) Threads → create

⇒ loops/cycle

if threads are created initialy

20 → 30
40 → 50
60 → 70
70 → 80
100 → 110

threads
① floor
② inorder predecessor



Intuition

preorder
1. thread → create
2. left subtree

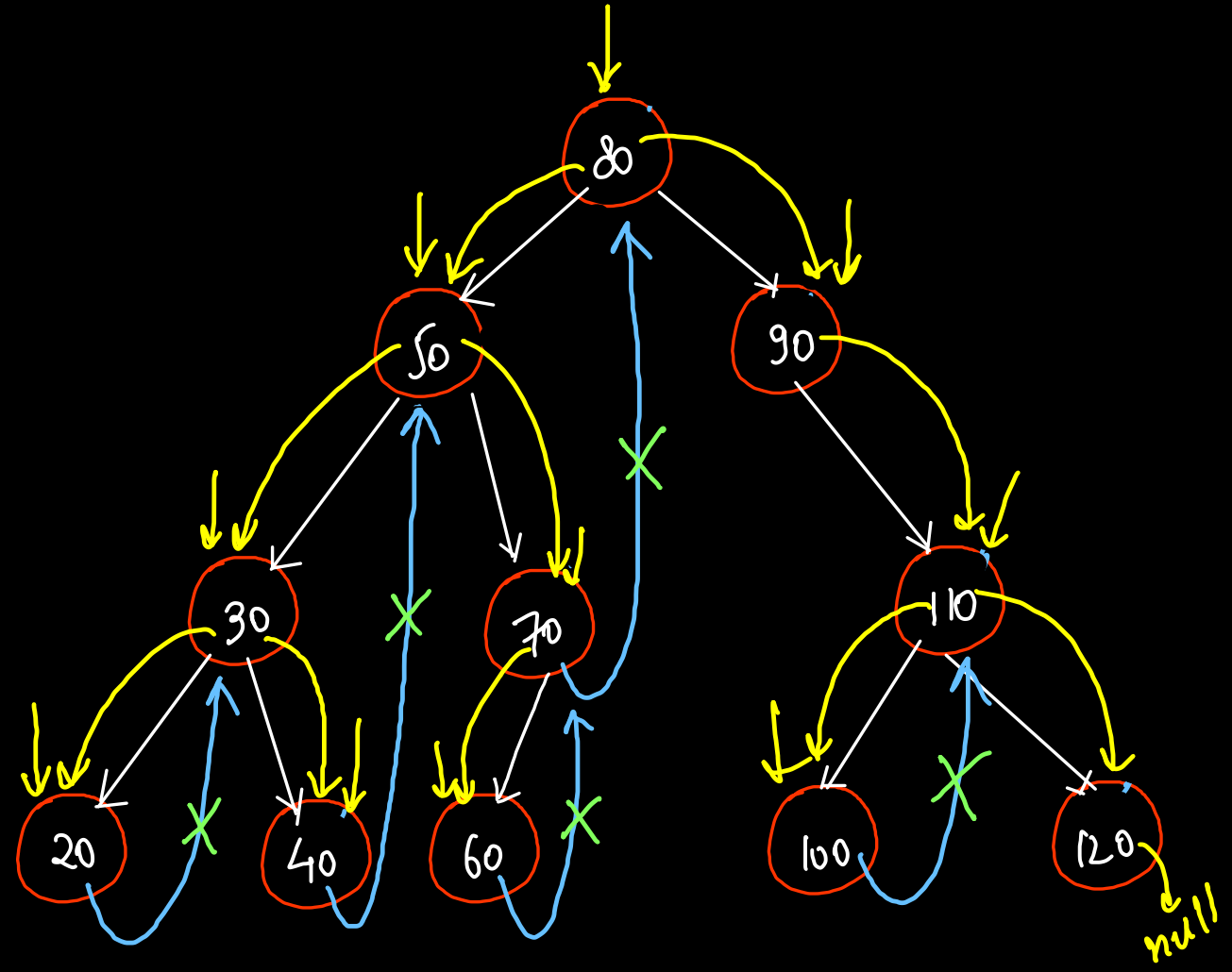
inorder
3. thread → delete
4. right subtree

preorder:

80 50 30 20 40 70 60

inorder:

20 30 40 50 60 70 80



preorder:

80 50 30 20 40 70 60 90 110 100 120

inorder:

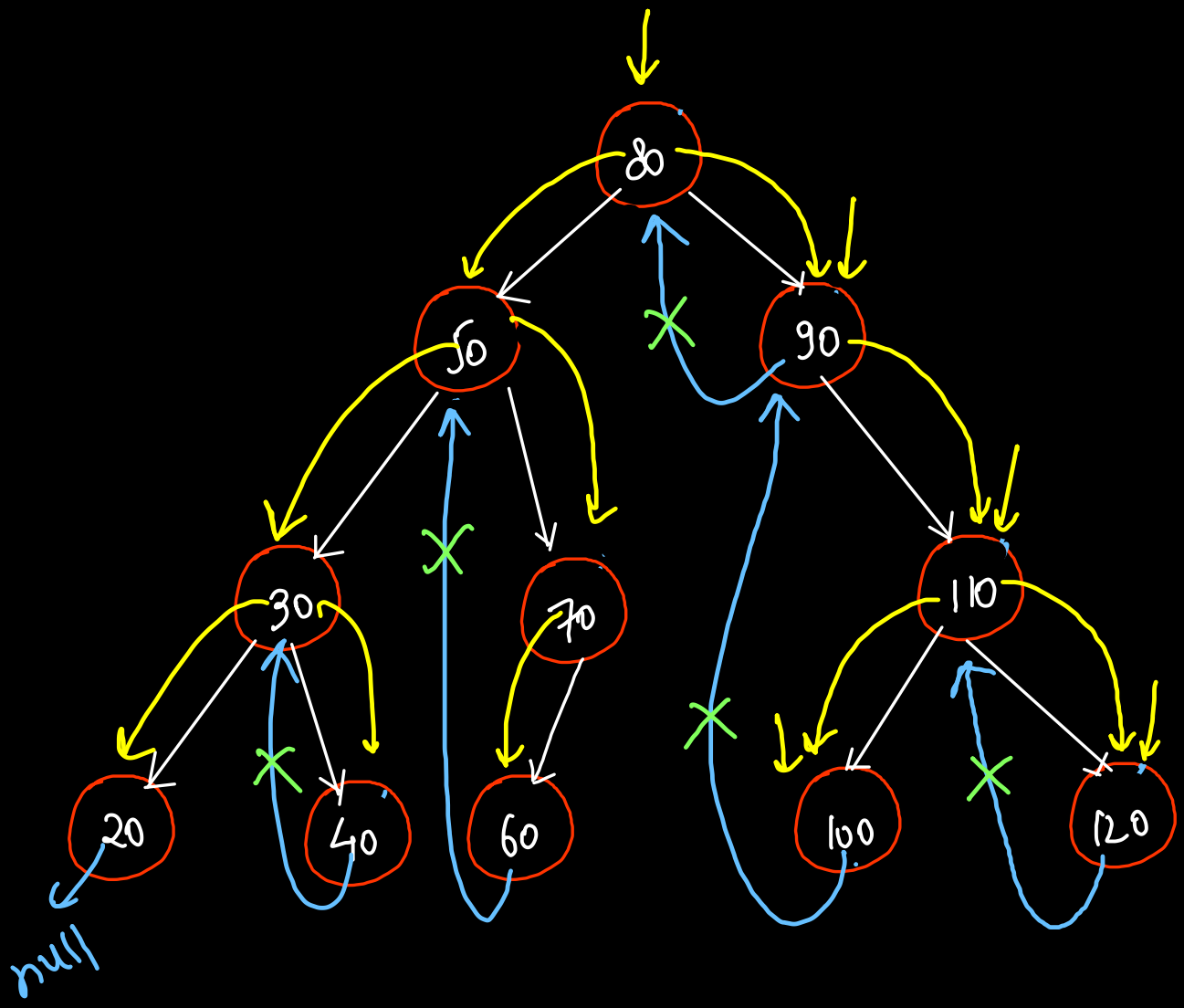
20 30 40 50 60 70 80 90 100 110 120

```java
public TreeNode floor(TreeNode root){
    TreeNode ans = root.left;
    while(ans.right != null && ans.right != root)
        ans = ans.right;

    return ans;
}
```

```java
List<Integer> preorder = new ArrayList<>();
List<Integer> inorder = new ArrayList<>();

while(root != null){
    if(root.left == null){
        preorder.add(root.val);
        inorder.add(root.val);
        root = root.right;
        continue;
    }

    TreeNode floor = floor(root);
    if(floor.right == null){
        floor.right = root; // thread creation
        preorder.add(root.val);
        root = root.left;
    } else {
        floor.right = null; // thread deletion
        inorder.add(root.val);
        root = root.right;
    }
}
```

Time = O(n)

Space = O(1) constant



Morris Postorder

left → right → root

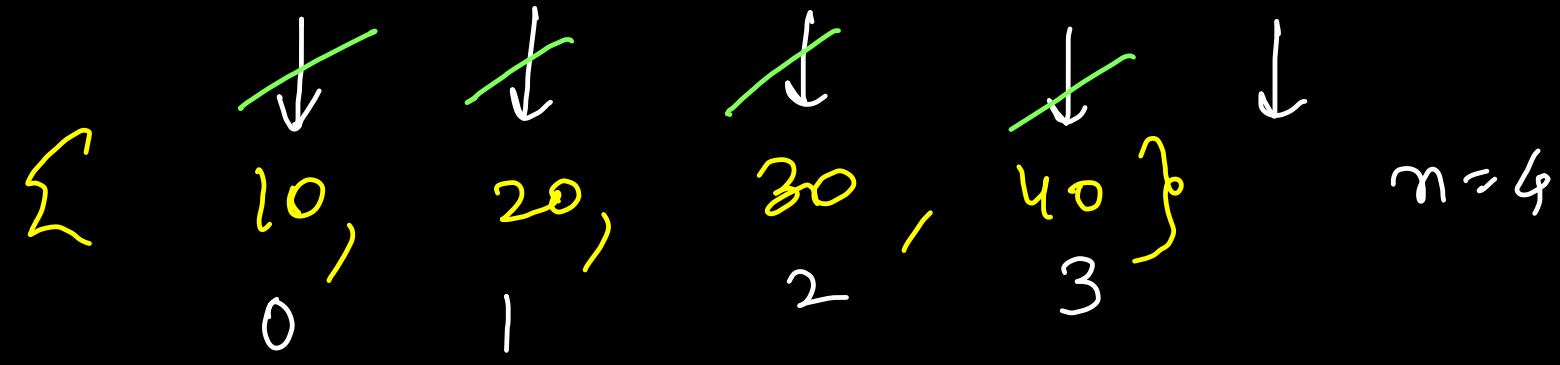
↓ reverse

root → right → left

80 90 110 120 100

50 70 60 30 40 20

Hint : Use left threaded binary tree



hasNext()

boolean

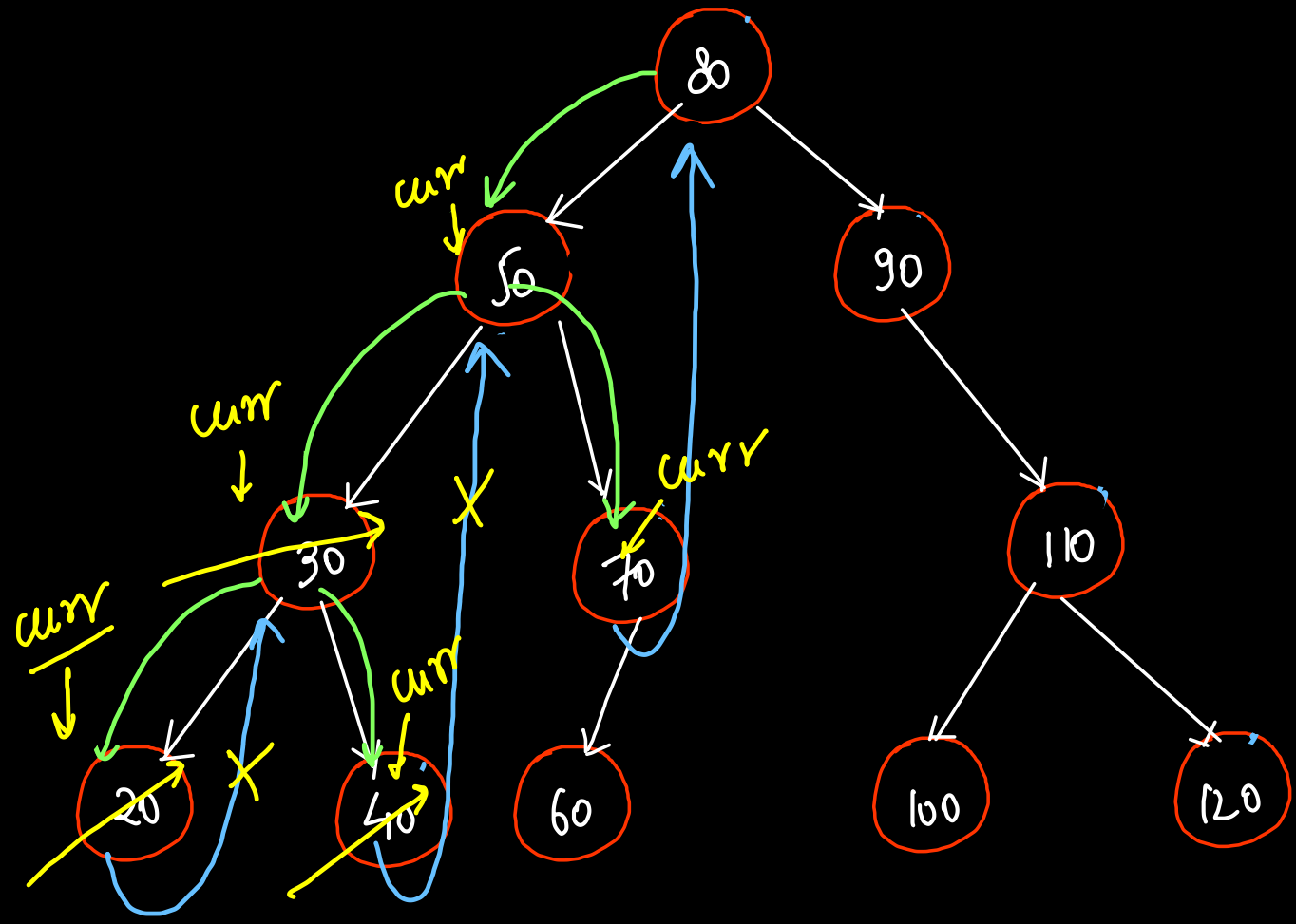
↓

```
if(ptr == n)
    return false;
else return true;
```

next()

returns curr value

& goes to next value



Constructor

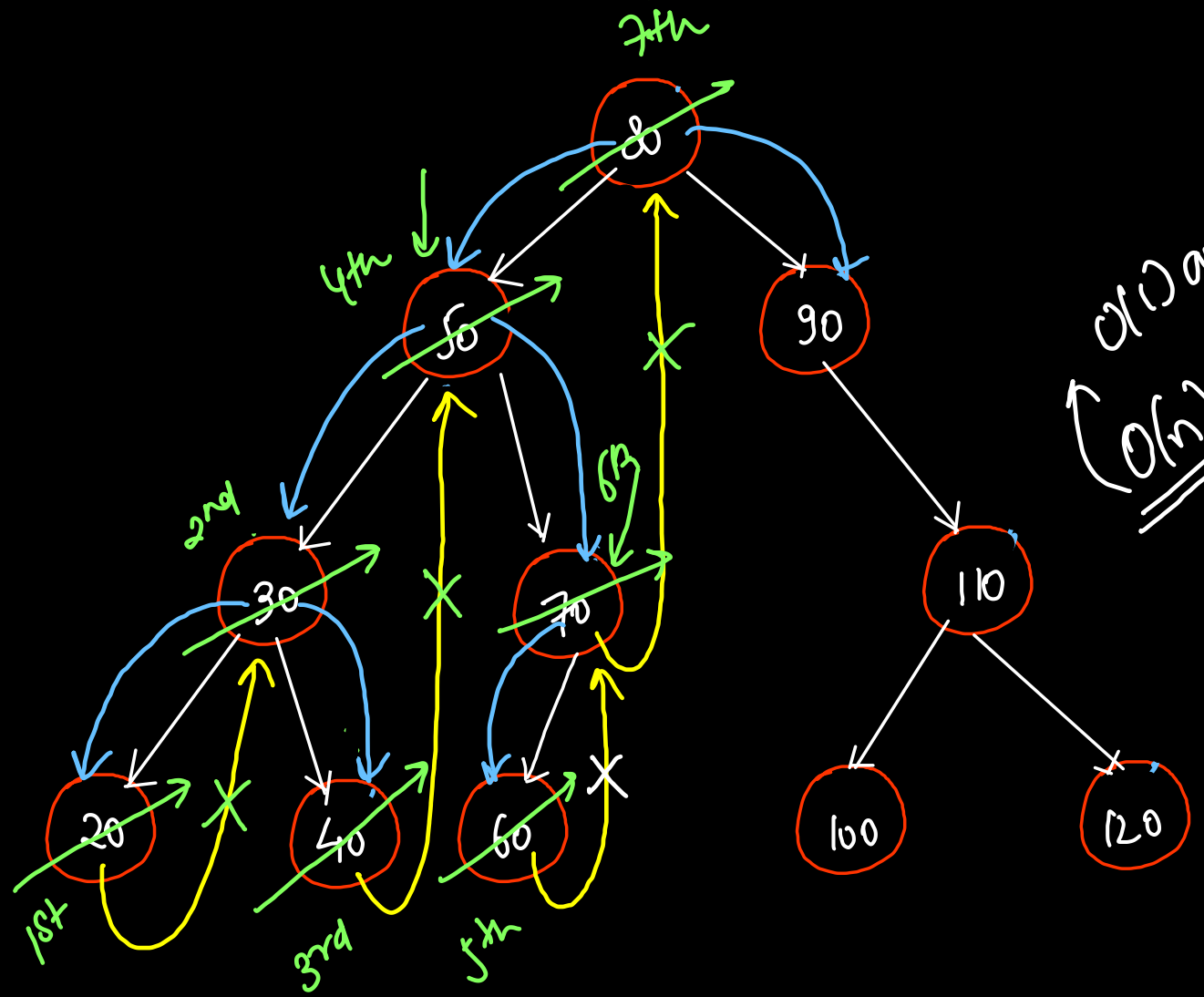
↳ root == smallest node

next() ⟶ 20

next() ⟶ 30

next() ⟶ 40

next() ⟶ 50



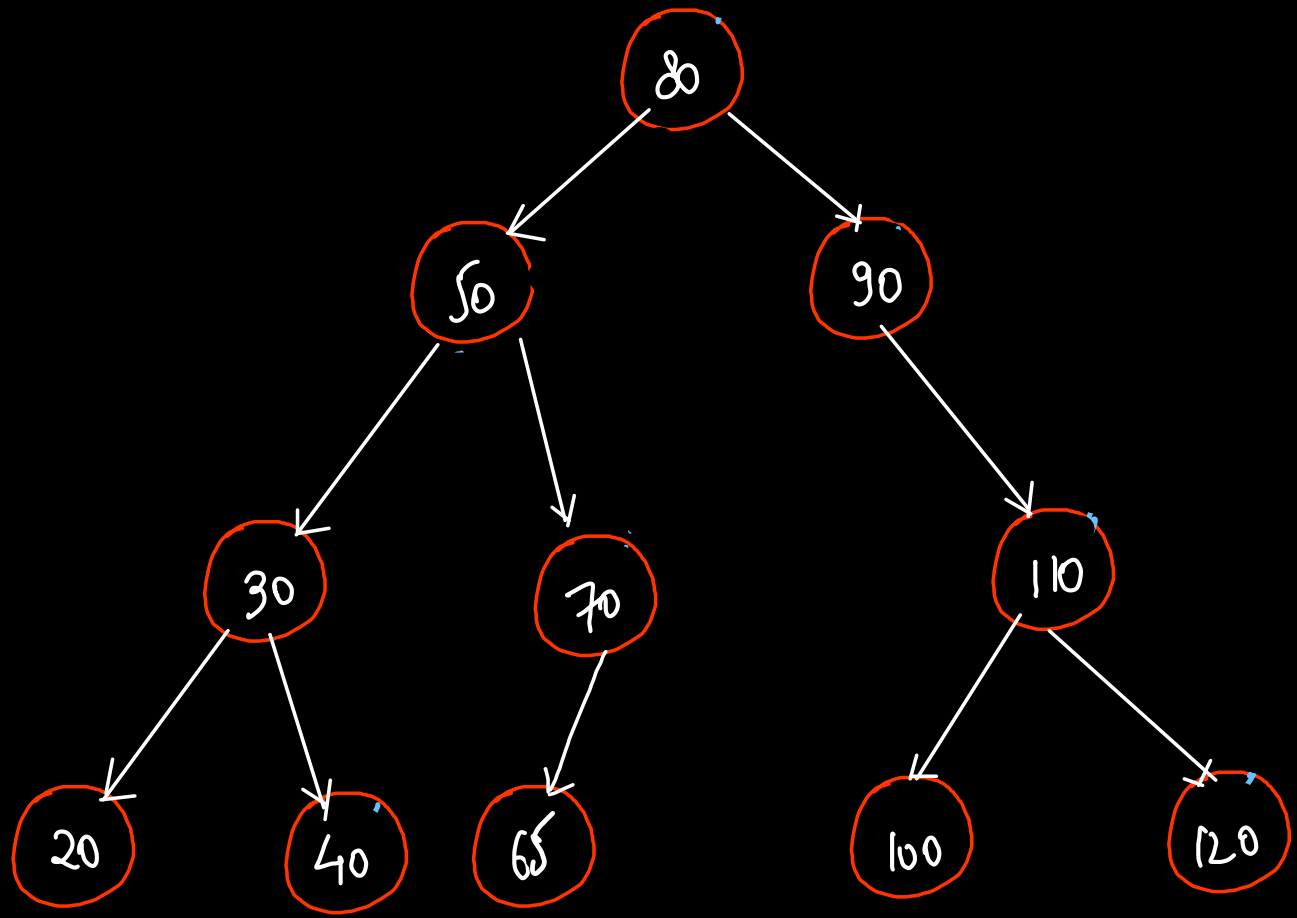

```
TreeNode root;

public BSTIterator(TreeNode curr) {
    root = curr;
}

public TreeNode floor(TreeNode root) {
    TreeNode ans = root.left;
    while (ans.right != null && ans.right != root)
        ans = ans.right;

    return ans;
}

public int next() {
    int ans = -1;
    while(root != null){
        if(root.left == null){
            ans = root.val;
            root = root.right;
            return ans;
        }
        TreeNode floor = floor(root);
        if(floor.right == null){
            floor.right = root; // thread creation
            root = root.left;
        } else {
            floor.right = null; // thread deletion
            ans = root.val;
            root = root.right;
            break;
        }
    }
    return ans;
}

public boolean hasNext() {
    if(root == null) return false;
    return true;
}
```

O(1)

O(1) avg per call

O(h) for n calls

O(1)



Target Sum Pair

pair target = 135

Approach 1)

Inorder

Time

Sorted

{ 10, 11, 15, 20, 23, 30 }

target = 35

10+30 = 40 ⇒ r--

10+23 = 33 ⇒ l++

11+23 = 34 ⇒ l++

15+23 = 38 ⇒ r--

```java
static class ForwardIterator{
    Stack<TreeNode> stk;

    public ForwardIterator(TreeNode root) {
        stk = new Stack<>();
        inorderSucc(root);
    }

    public void inorderSucc(TreeNode curr){
        while(curr != null){
            stk.push(curr);
            curr = curr.left;
        }
    }

    public int peek(){
        if(hasNext() == false) return 0;
        return stk.peek().val;
    }

    public int next() {
        if(hasNext() == false) return 0;
        TreeNode curr = stk.pop();
        inorderSucc(curr.right);
        return curr.val;
    }

    public boolean hasNext() {
        return (stk.size() > 0);
    }
}
```

```java
static class BackWardIterator{
    Stack<TreeNode> stk;

    public BackWardIterator(TreeNode root) {
        stk = new Stack<>();
        inorderPred(root);
    }

    public void inorderPred(TreeNode curr){
        while(curr != null){
            stk.push(curr);
            curr = curr.right;
        }
    }

    public int peek(){
        if(hasPrev() == false) return 0;
        return stk.peek().val;
    }

    public int prev() {
        if(hasPrev() == false) return 0;
        TreeNode curr = stk.pop();
        inorderPred(curr.left);
        return curr.val;
    }

    public boolean hasPrev() {
        return (stk.size() > 0);
    }
}
```

Approach2)

LC 653)

Target Sum Pair in BST

```java
public boolean findTarget(TreeNode root, int target) {
    if(root == null || (root.left == null && root.right == null)) return false;

    ForwardIterator left = new ForwardIterator(root);
    BackWardIterator right = new BackWardIterator(root);

    while(left.hasNext() == true && right.hasPrev() == true && left.peek() < right.peek()){
        if(left.peek() + right.peek() == target) return true;
        if(left.peek() + right.peek() < target) left.next();
        else right.prev();
    }

    return false;
}
```