Trees

Contents

Trees

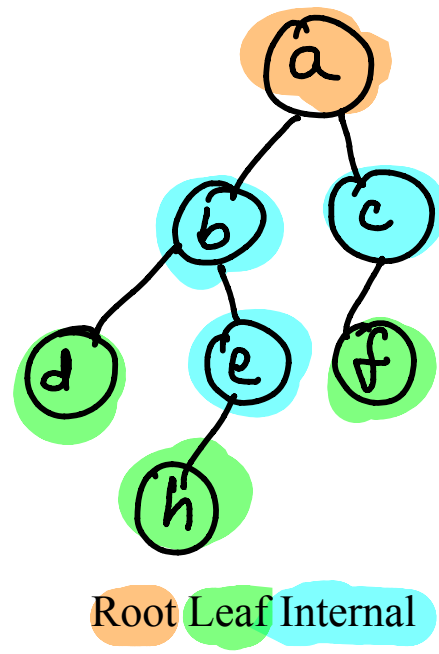
Tree - collection of tree-nodes

why trees?

1. Hiraichy
2. Computer system. (UNIX)

① Class Treenode
- data
- list<Treenode> children

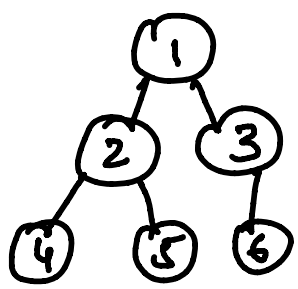
② Binary Tree → atmost 2 children (0,1,2)
- data
- leftchild
- rightchild



③ Types →

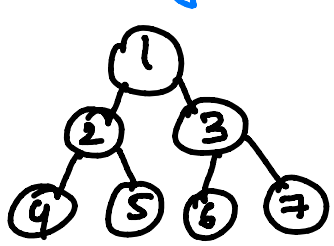
Ⓐ Complete Binary Tree
↳ all levels are completely filled except last one

Eg



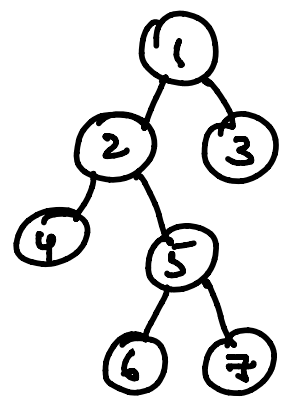
Ⓑ Perfect Binary Tree
↳ every internal node has exactly 2 children

Eg



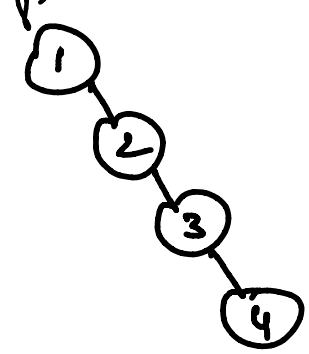
Ⓒ Full Binary tree
↳ if every node has 0 or 2 children

Eg



Ⓓ Skewed Binary Tree
(* used for finding complexity)
↳ all nodes have either one or no child.

Eg

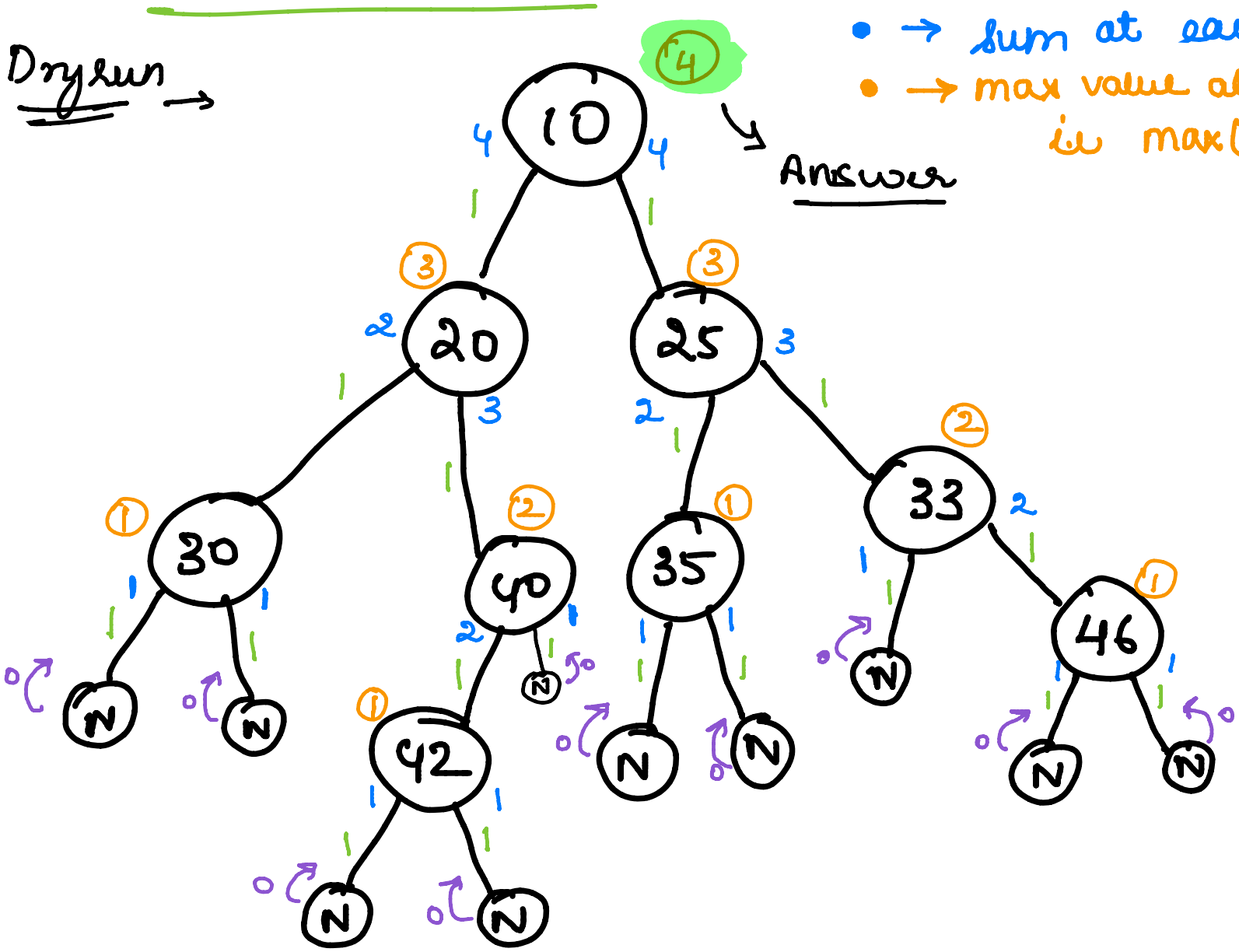


D1

① Depth of a binary tree (Max depth)

- → 1 added while returning.
- → sum at each node.
- → max value at node is max(left, right).

Dryrun →



if null
then ht = 0

- consider max at a node ie either left or right

Tc → O(n)
SC → O(1)
Aux → O(h)
h → height

Code →

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

② Maximum depth of n-ary tree

Idea is same as previous problem, only implementation changes

Code →

```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;
    }
};
*/
class Solution {
    public int maxDepth(Node root) {
        if (root == null) return 0;
        int max = 0;
        for (Node child : root.children)
            max = Math.max(max, maxDepth(child));
        return max +1;
    }
}
```

D2

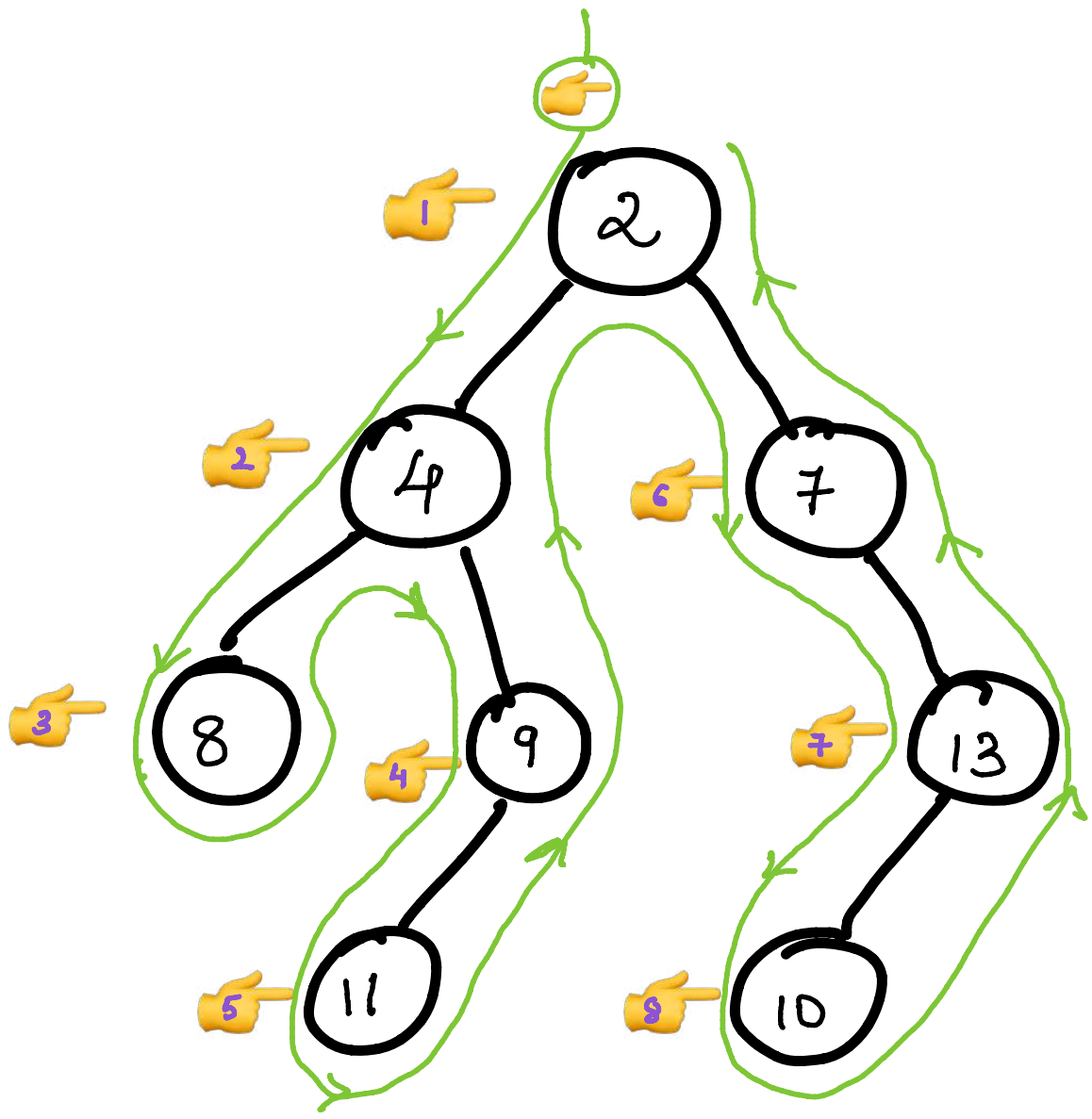
Traversals → DFS → Preorder
Inorder
Postorder

→ BFS → Level order

(A) Preorder →

processing order → node
left child
right child

Eg



* Point fingers as shown and traverse the tree starting from Root

* Order of visiting is the preorder traversal.

[2, 4, 8, 9, 11, 6, 13, 10]

TC → O(n)

SC → O(n) Recursive stack space → O(h) h → height.

③ Pre-order traversal of Binary tree

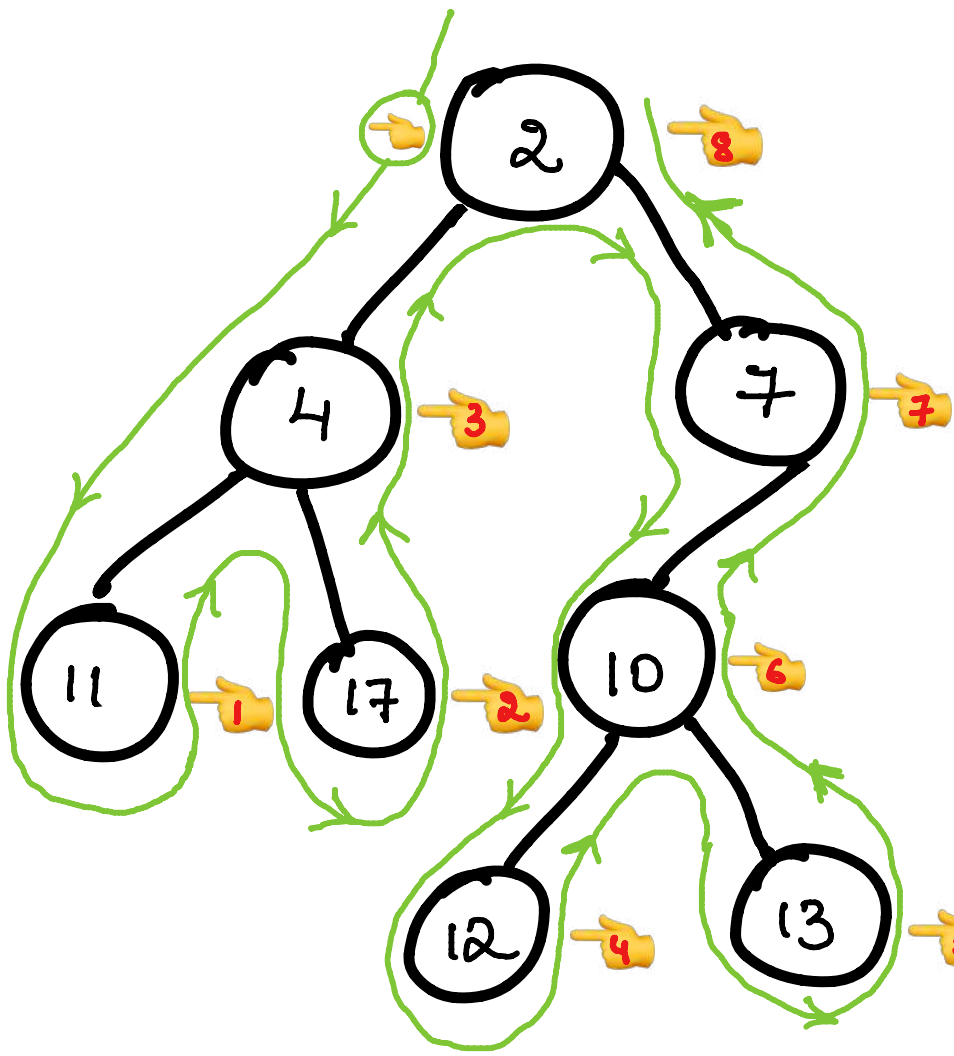
```java
class BinaryTree
{
    //Function to return a list containing the preorder traversal of the tree.
    static ArrayList<Integer> preorder(Node root)
    {
        // Code here
        ArrayList<Integer> list=new ArrayList<>();
        preOrder(root,list);
        return list;

    }

    public static void preOrder(Node root , ArrayList<Integer> list){
        if(root==null)
        return;

        list.add(root.data);
        preOrder(root.left,list);
        preOrder(root.right,list);
    }

}
```

④ Pre-order traversal of n-ary tree

```cpp
class Solution {
public:
    vector<int> preorder(Node* root) {
        vector<int>ans;
        Preorder(root,ans);
        return ans;
    }
    void Preorder(Node* root, vector<int>&ans)
    {
        if(root==NULL) return;
        ans.push_back(root->val);
        for(int i=0;i<root->children.size();i++)
        {
           Preorder(root->children[i],ans);

        }
        return;
    }
};
```

(B) Postorder → processing order → left child, right child, node

Eg



* Point fingers as shown and traverse the tree starting from Root

* Order of visiting is the postorder traversal.

[11, 17, 4, 12, 13, 10, 7, 2]

Tc → O(n)

Sc → O(n) Recursive stack space → O(h) h → height.

⑤ Postorder traversal of Binary tree

```java
class Solution {
    public List<Integer> postorderTraversal(TreeNode root) {
        List<Integer> ans = new ArrayList<>();
        if(root==null)
        return ans;
        ans.addAll(postorderTraversal(root.left));
        ans.addAll(postorderTraversal(root.right));
        ans.add(root.val);
        return ans;
    }
}
```

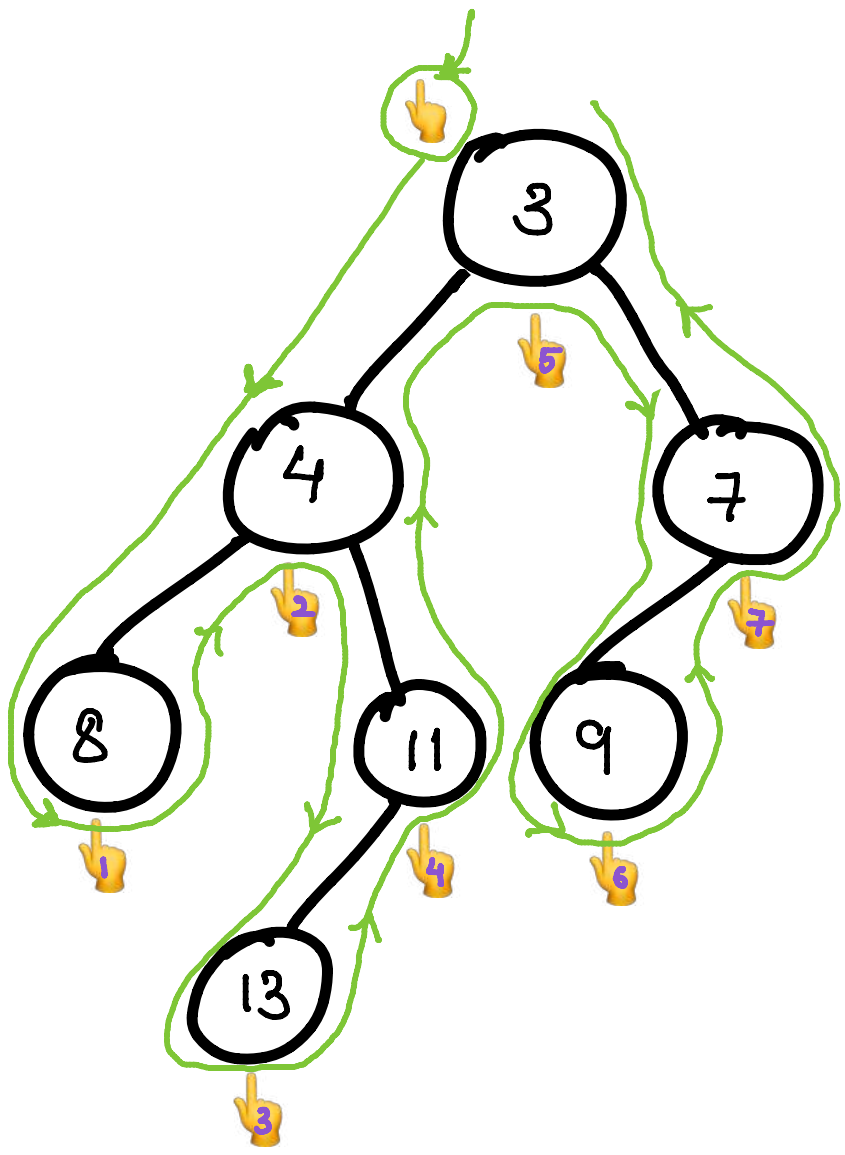
⑥ Postorder traversal of nary tree

```cpp
class Solution {
public:
    vector<int> postorder(Node* root) {
        vector<int>ans;
        Postorder(root,ans);
        return ans;
    }
    void Postorder(Node* root, vector<int>&ans)
    {
        if(root == NULL) return;
        for(int i=0;i<root->children.size();i++)
        {
            Postorder(root->children[i],ans);
        }
        ans.push_back(root->val);
        return;
    }
};
```

© Inorder →

processing order → left child
node
right child

Eg



* Point fingers as shown and traverse the tree starting from Root

* Order of visiting is the Inorder traversal.

[8, 4, 13, 11, 3, 9, 7]

Tc → O(n)

Sc → O(n) Recursive Stack space → O(h) h → height.

⑦ In-order traversal of Binary tree

```java
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> ans = new ArrayList<>();
        if(root == null)
        return ans;
        ans.addAll(inorderTraversal(root.left));
        ans.add(root.val);
        ans.addAll(inorderTraversal(root.right));
        return ans;

    }
}
```

Inorder traversal of n-ary tree

Approach:

The inorder traversal of an N-ary tree is defined as visiting all the children except the last then the root and finally the last child recursively.

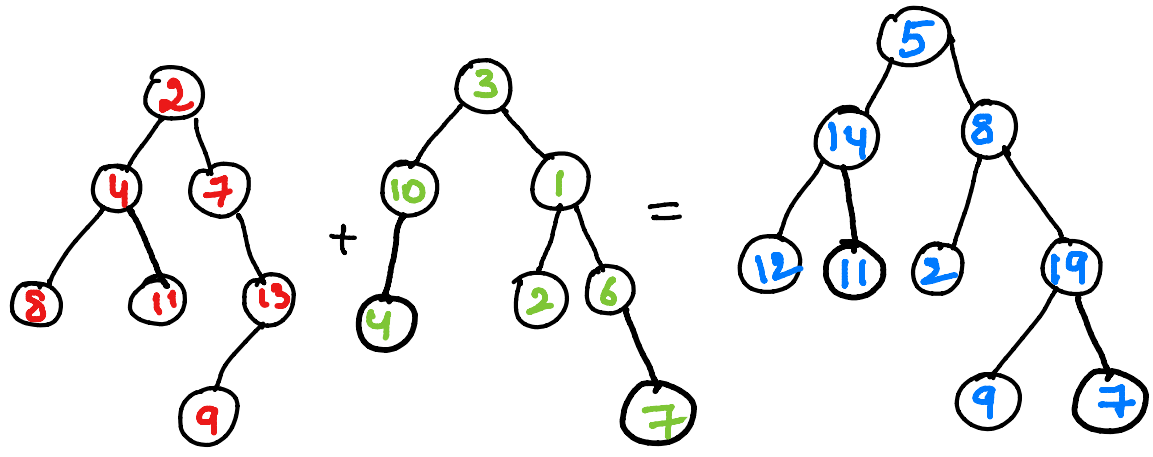
- Recursively visit the first child.
- Recursively visit the second child.
- .....
- Recursively visit the second last child.
- Print the data in the node.
- Recursively visit the last child.
- Repeat the above steps till all the nodes are visited.

```
void inorder(Node *node)
{
    if (node == NULL)
        return;

    // Total children count
    int total = node->length;

    // All the children except the last
    for (int i = 0; i < total - 1; i++)
        inorder(node->children[i]);

    // Print the current node's data
    cout<< node->data << " ";

    // Last child
    inorder(node->children[total - 1]);
}
```

D3 (8) Merge two Binary trees →

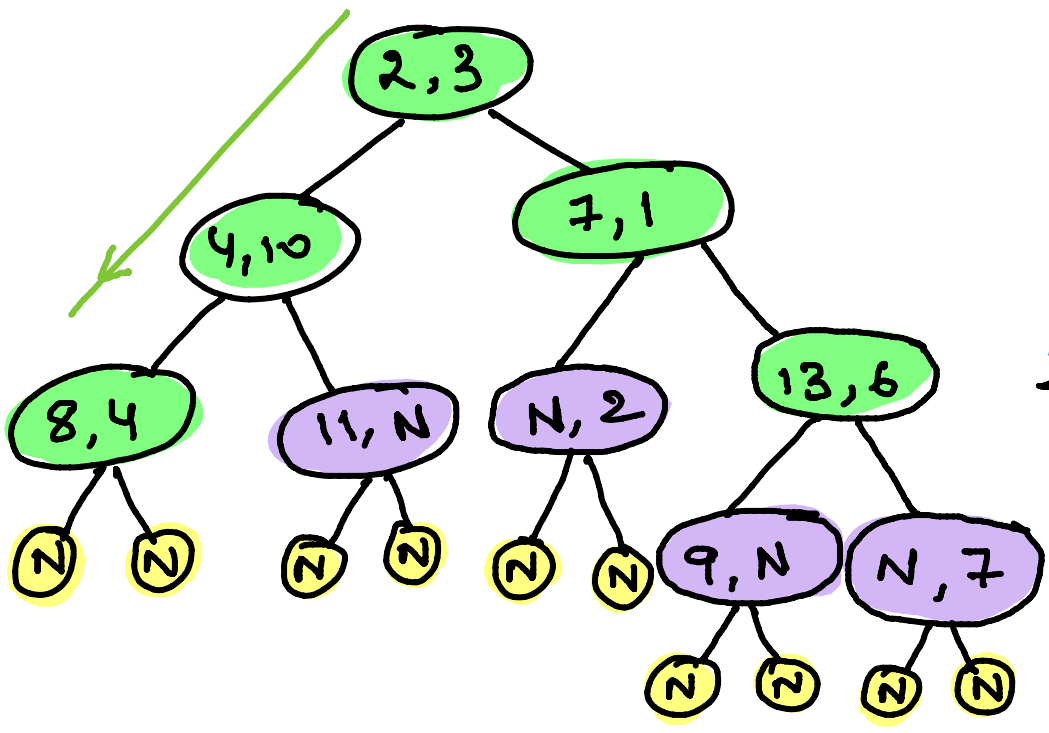
Given root nodes of 2 binary trees, return root of the sum tree

Eg

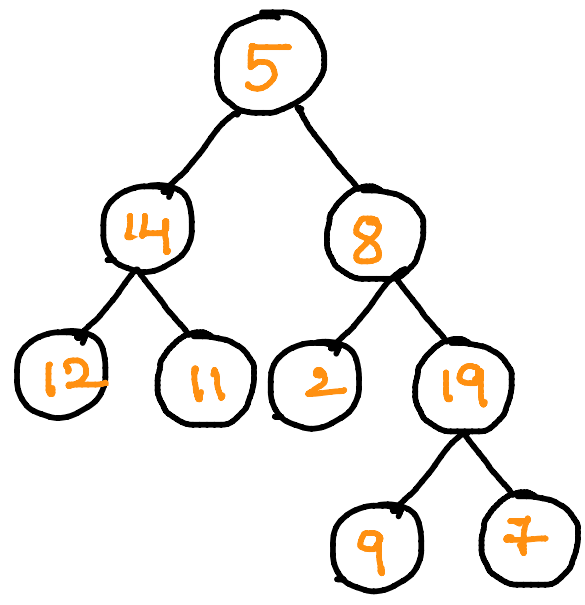


we will perform preorder traversal on the binary tree because the node/root need to be processed first.

The recursive tree structure would be like :



condensed to



Resultant Tree

- NULL & NULL
- Node & NULL
- Node & Node

TC → O(n+m)

SC → O(max(n,m))

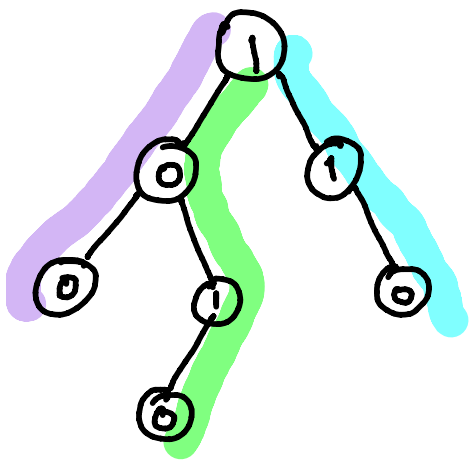
Recursive stack → O(max(h1,h2))

code →

```java
class Solution
{
    public TreeNode mergeTrees(TreeNode root1, TreeNode root2)
    {
        if(root1 == null)
            return root2;
        if(root2 == null)
            return root1;
        TreeNode left= mergeTrees(root1.left, root2.left);
        TreeNode right= mergeTrees(root1.right, root2.right);
        TreeNode node= new TreeNode(root1.val+root2.val, left, right);
        return node;
    }
}
```

(9) Sum of root to leaf paths →

Eg

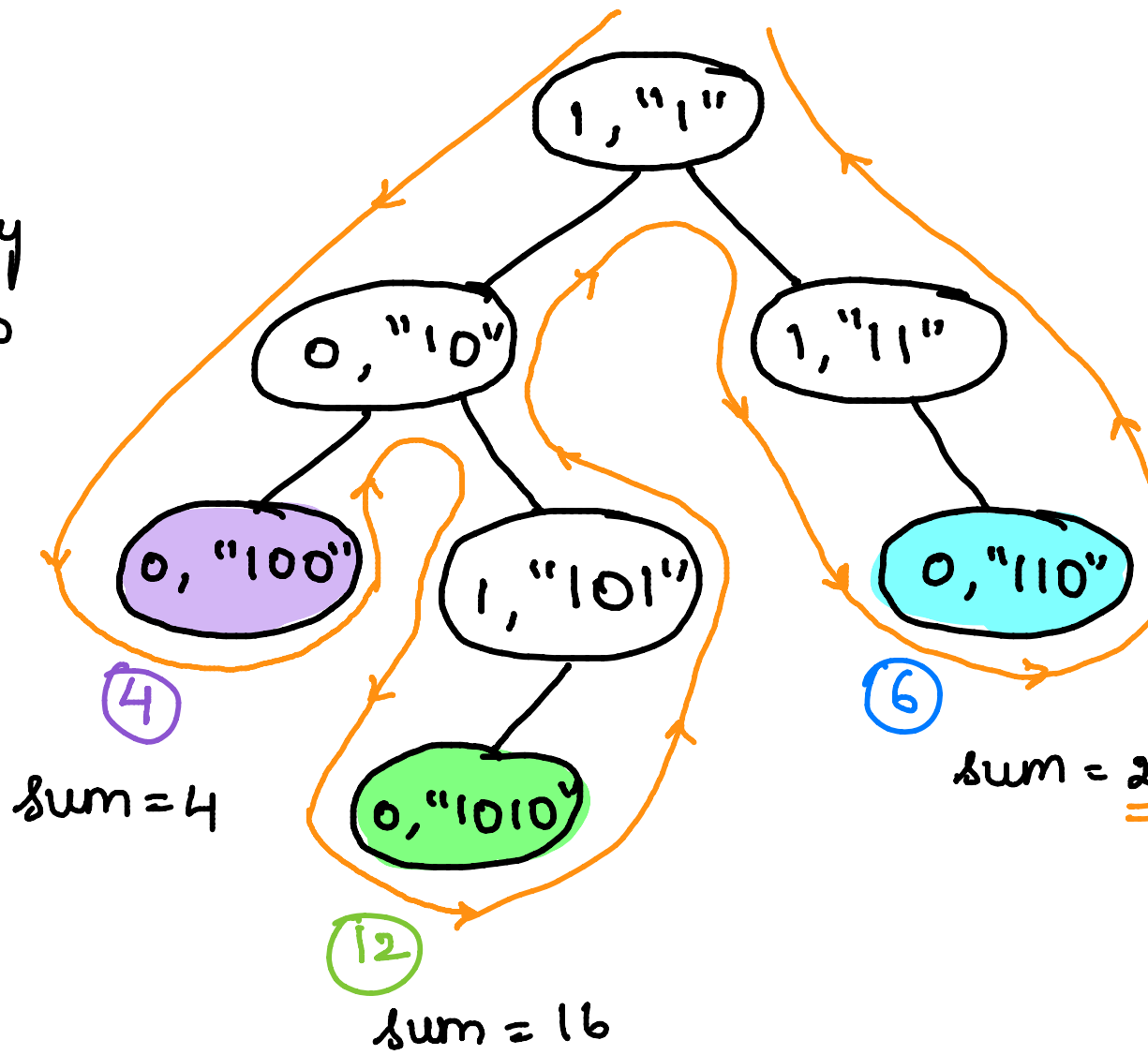


→ $(100)_2 + (1010)_2 + (110)_2$

$4 + 12 + 6$

$= 22$.

=



Initially

sum = 0

* If root becomes null convert string to integer & add to sum.

Time → O(n)

Space → O(n)

Recursive stack → O(h)

Code

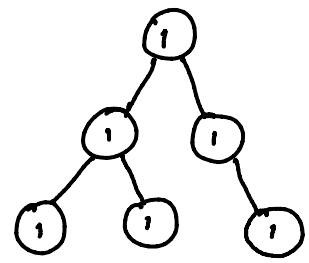
```java
public class Solution {
    int total;
    public int sumNumbers(TreeNode root) {
        total = 0;
        helper(root, 0);
        return total;
    }
    void helper(TreeNode root, int sum) {
        if (root == null) return;
        sum = sum * 10 + root.val;
        if (root.left == null && root.right == null) {
            total += sum;
            return;
        }
        helper(root.left, sum);
        helper(root.right, sum);
    }
}
```

Note →

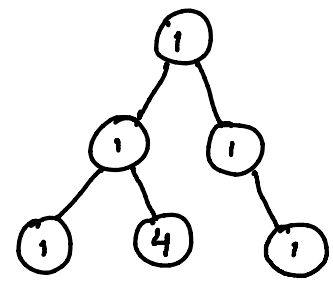
stoi() can take upto three parameters, the second parameter is for starting index and third parameter is for base of input number.

[to convert from binary to decimal we give it as 2]

(10) Univalued Binary Tree →

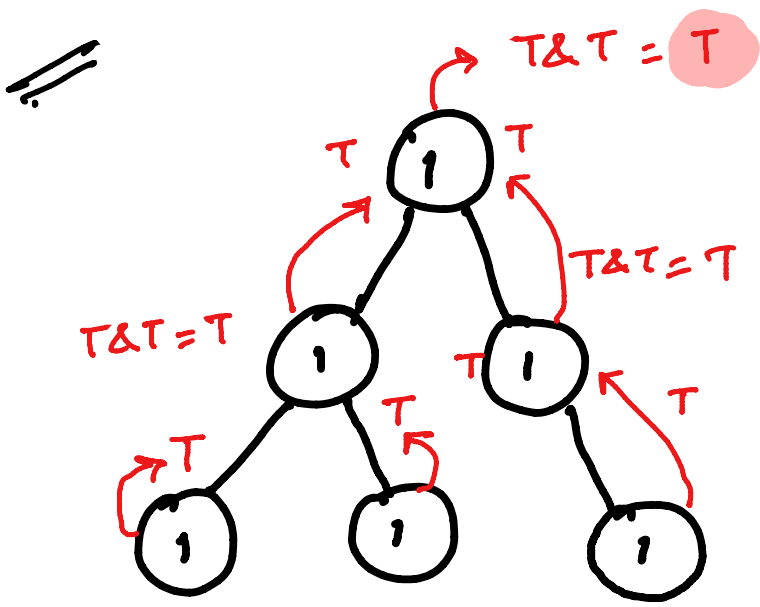


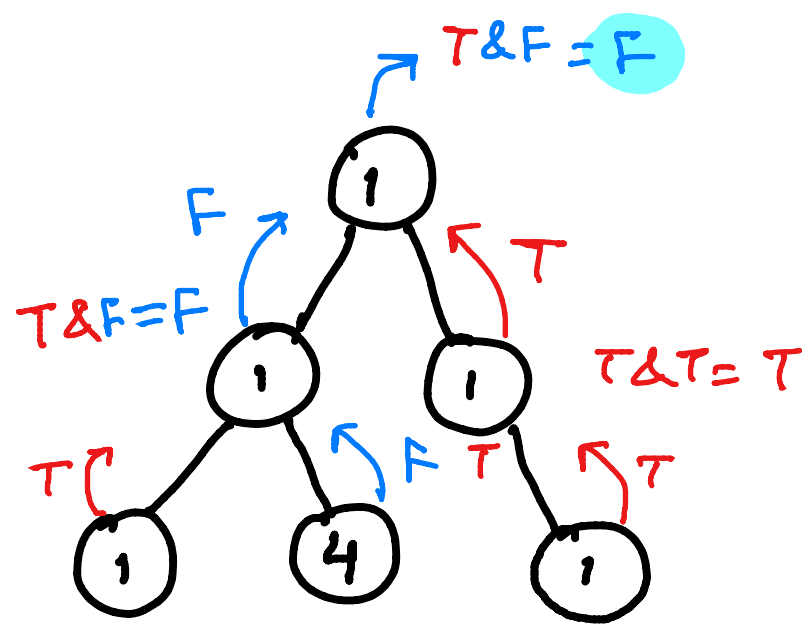
Is returns true



Is returns false

Eg





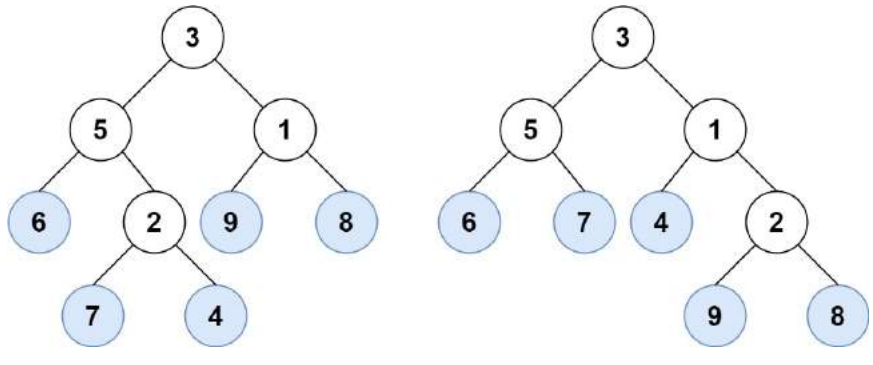
Code

```java
class Solution {
    public boolean isUnivalTree(TreeNode root) {
    if (root == null) return true;

    if (root.left != null && root.left.val != root.val)
        return false;
    if (root.right != null && root.right.val != root.val)
        return false;
    return isUnivalTree(root.left) && isUnivalTree(root.right);
    }
}
```

(11) Leaf Similar trees

↳ Return true if all leaves are in same order for both trees.

Eg



V_1 = 6, 7, 4, 9, 8 ⇒ $V_1 == V_2$

V_2 = 6, 7, 4, 9, 8 ↳ return true else false.

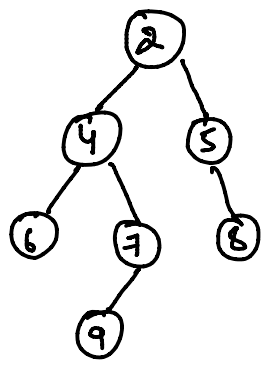
Code →

```java
class Solution {
        ArrayList<Integer> leaf1 = new ArrayList<>();
        ArrayList<Integer> leaf2 = new ArrayList<>();
    public boolean leafSimilar(TreeNode root1, TreeNode root2) {
        leaf1 = findLeafNodes(root1,leaf1);
        leaf2 = findLeafNodes(root2,leaf2);
        return leaf1.equals(leaf2);
    }
    public static ArrayList<Integer> findLeafNodes(TreeNode root,ArrayList<Integer> leaf){
        if(root == null){
            return null;
        }
        if(root.left == null && root.right == null){
            leaf.add(root.val);
        }
        findLeafNodes(root.left,leaf);
        findLeafNodes(root.right,leaf);
        return  leaf;
    }
}
```

DS (12) Binary Tree paths

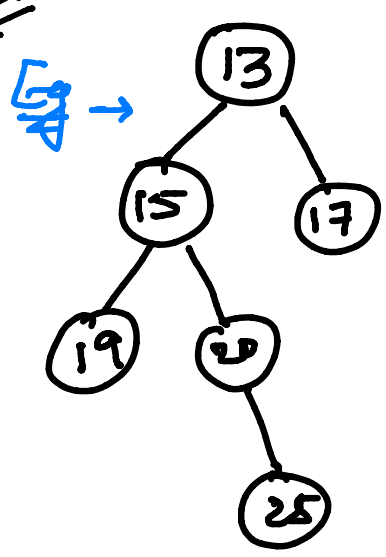
↳ given root print all the paths from root to leaf

Eg



⇒ ["2→4→6", "2→4→7→9", "2→5→8"]

Eg →



Result =

["13→15→19", "13→15→20→25", "13→17"]

Time complexity = O(n)

Space complexity = O(x) + O(h) → recursive stack.

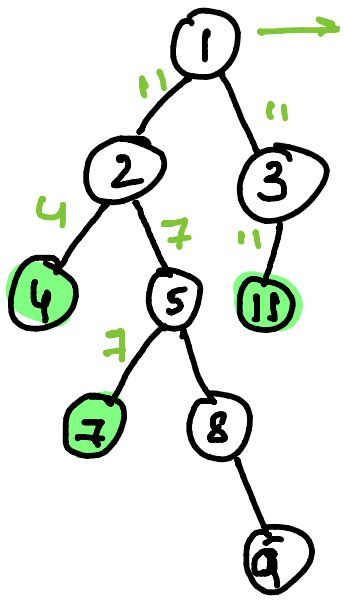
↳ Answer array

Code →

```java
class Solution {
    public List<String> binaryTreePaths(TreeNode root) {
        List<String> answer = new ArrayList<String>();
        if (root != null) searchBT(root, "", answer);
        return answer;
    }
    private void searchBT(TreeNode root, String path, List<String> answer) {
        if (root.left == null && root.right == null) answer.add(path + root.val);
        if (root.left != null) searchBT(root.left, path + root.val + "->", answer);
        if (root.right != null) searchBT(root.right, path + root.val + "->", answer);
    }
}
```

(13) Sum of left leaves →

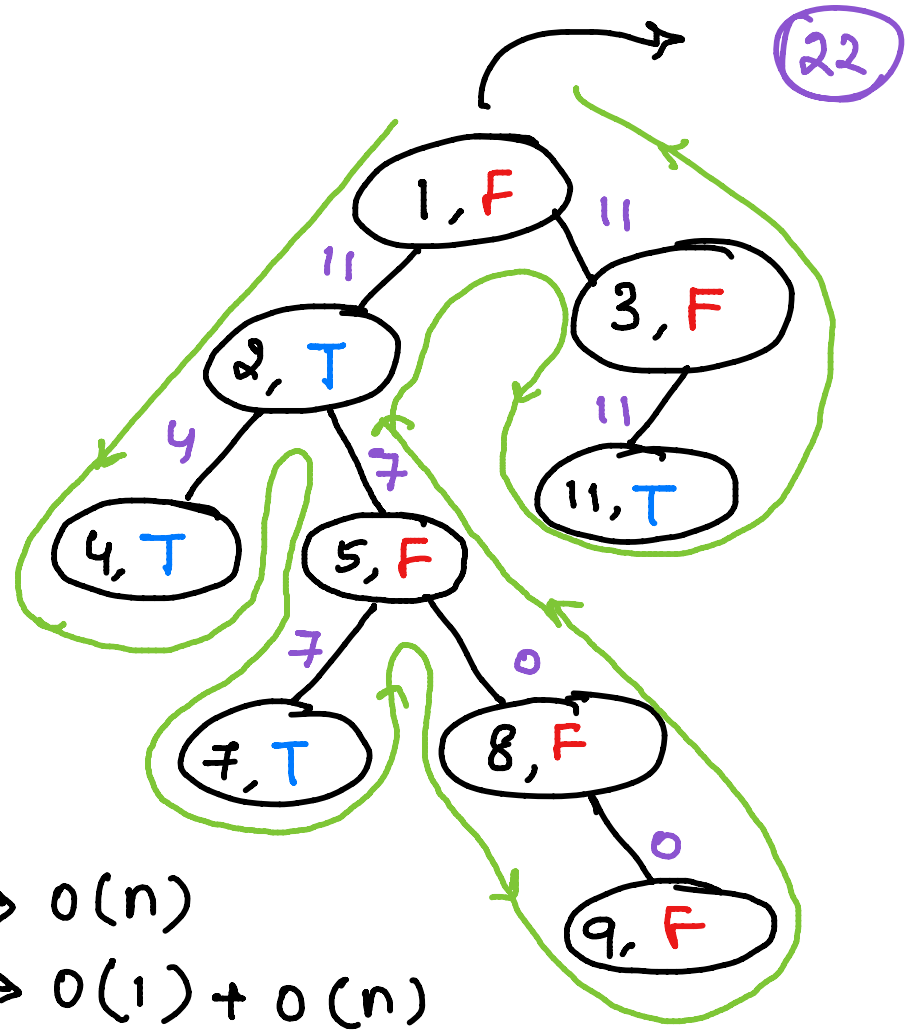
Eg



→ 22.

Result = 4 + 7 + 11

= 22.



Tc → O(n)

Sc → O(1) + O(n)

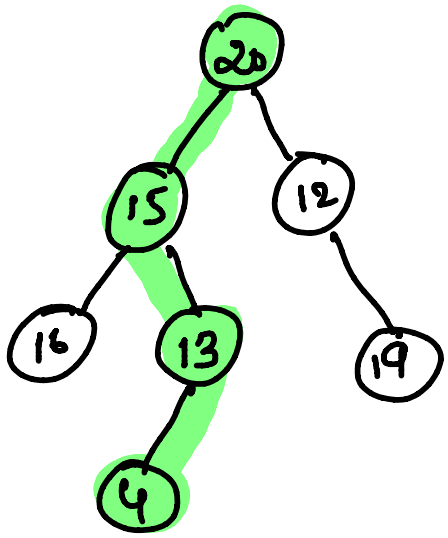
↳ stack

Code →

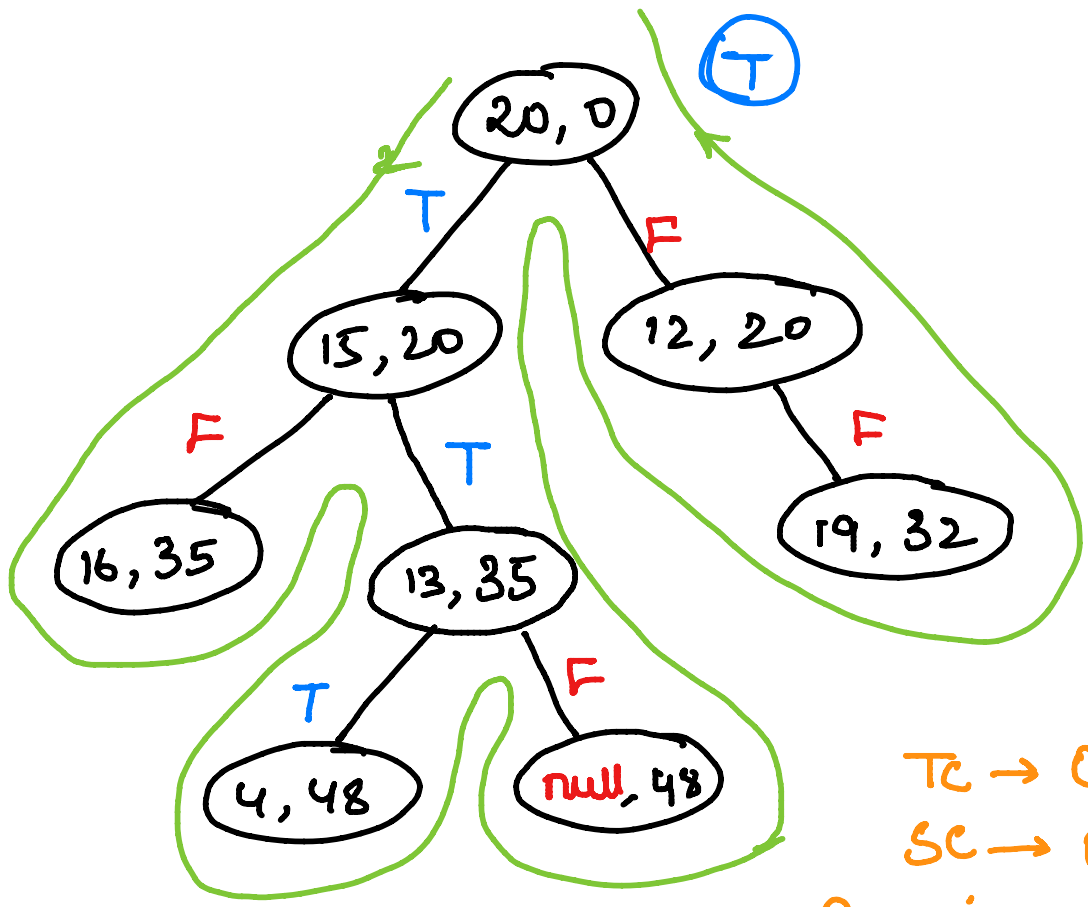
```java
class Solution {
    public int sumOfLeftLeaves(TreeNode root) {
        if (root == null) {
            return 0;
        }
        if (root.left != null && root.left.left == null && root.left.right == null) {
            return root.left.val + sumOfLeftLeaves(root.right);
        }
        return sumOfLeftLeaves(root.left) + sumOfLeftLeaves(root.right);
    }
}
```

(14) Path sum → Sum of all nodes from root to leaf is equal to target sum → then T else F.

Eg



target = 52



Tc → O(n)

Sc → O(1)

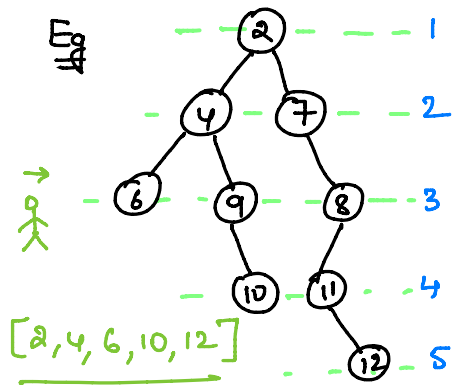
Recursive Stack → O(h)

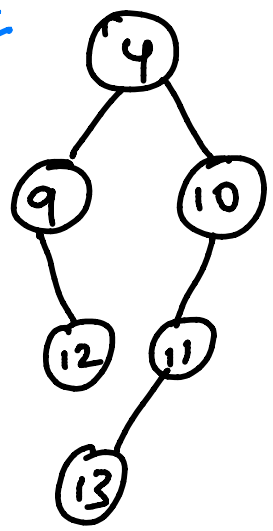
Code

```
16  class Solution {
17      public boolean hasPathSum(TreeNode root, int targetSum) {
18          Stack<TreeNode> stack = new Stack<TreeNode>();
19          Stack<Integer> sumss = new Stack<Integer>();
20          stack.push(root);
21          sumss.push(targetSum);
22          while(!stack.isEmpty()&&(root!=null)){
23              int value = sumss.pop();
24              TreeNode top = stack.pop();
25              if(top.left==null&&top.right==null&&top.val==value){
26                  return true;
27              }
28              if(top.right!=null){
29                  stack.push(top.right);
30                  sumss.push(value-top.val);
31              }
32              if(top.left!=null){
33                  stack.push(top.left);
34                  sumss.push(value-top.val);
35              }
36          }
37          return false;
38      }
39  }
```

D6 (15) Left view of a Binary Tree

Eg





Ans = [4, 9, 12, 13]

→ For every level traversed,
check if it already exist in the set,

if already exist then continue,
else add the node's value
to array & into the set

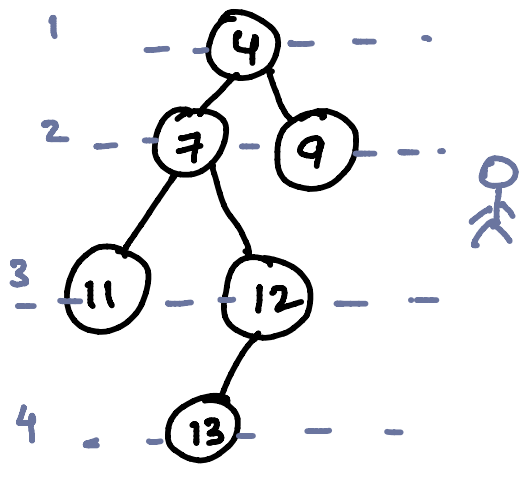
Tc → O(n)

Sc → O(n) + O(n) + O(h)
result

Code →

```java
class Solution {
    public List<Integer> leftSideView(TreeNode root) {
        List<Integer> ans = new ArrayList<>();
        if(root == null)
        return ans;
        left(root,ans,0);
        return ans;
    }
    public void left(TreeNode node , List<Integer> ans , int level)
    {
        if(node == null)
        return;

        if(level == ans.size())
        ans.add(node.val);

        left(node.left , ans , level + 1);
        left(node.right , ans , level + 1);
    }
}
```

(16) Right view of Binary Tree →



Result = [4, 9, 12, 13].

→ The entire approach to solve the problem is same as the left view of binary tree. Even the time complexities.

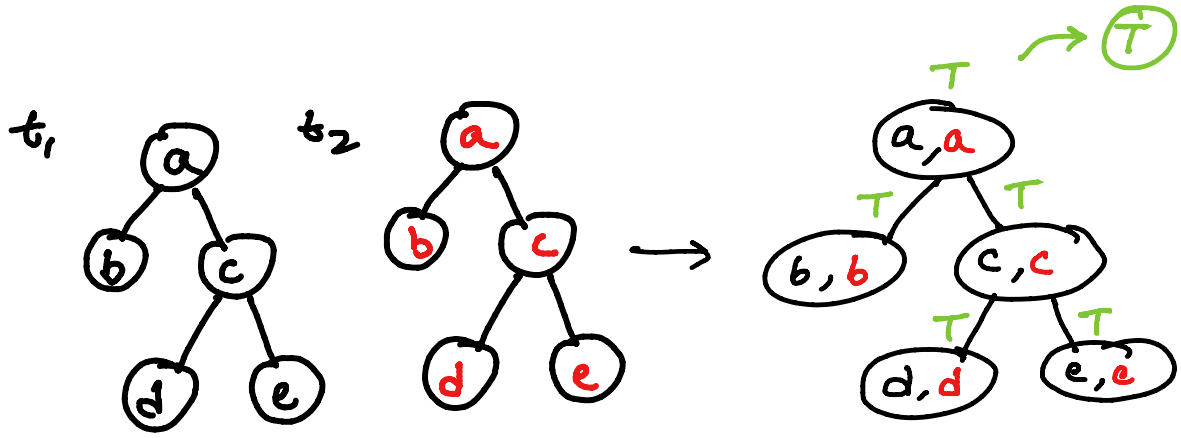
→ Only order of calling the branches change.

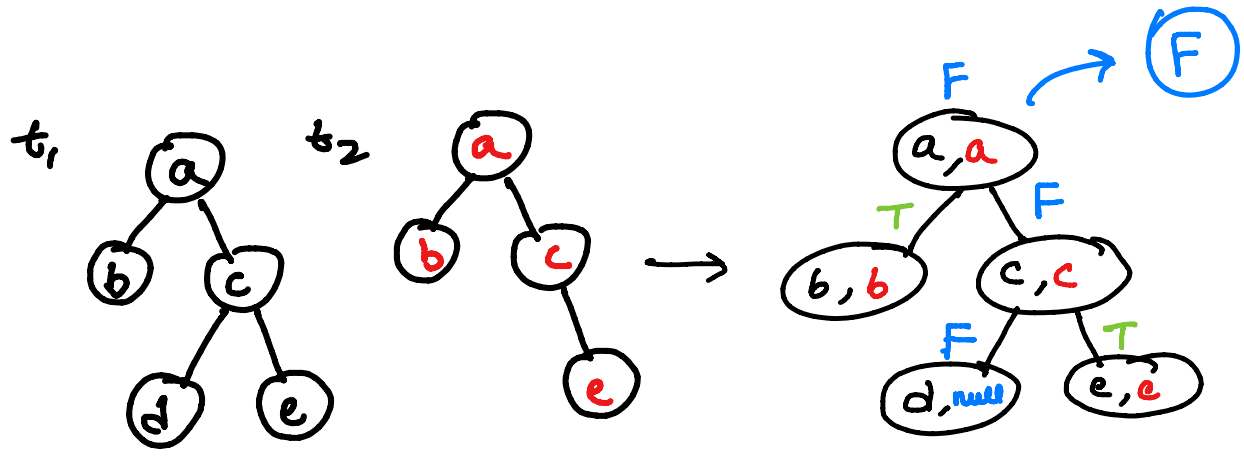
① right
② left

Code

```java
class Solution {
    public List<Integer> rightSideView(TreeNode root) {
        List<Integer> ans = new ArrayList<>();
        if(root == null)
        return ans;
        right(root,ans,0);
        return ans;
    }
    public void right(TreeNode node , List<Integer> ans , int level)
    {
        if(node == null)
        return;

        if(level == ans.size())
        ans.add(node.val);

        right(node.right , ans , level + 1);
        right(node.left , ans , level + 1);
    }
}
```

(17) Same tree → return true if both tree are same else false





Tc → O(min(m, n))

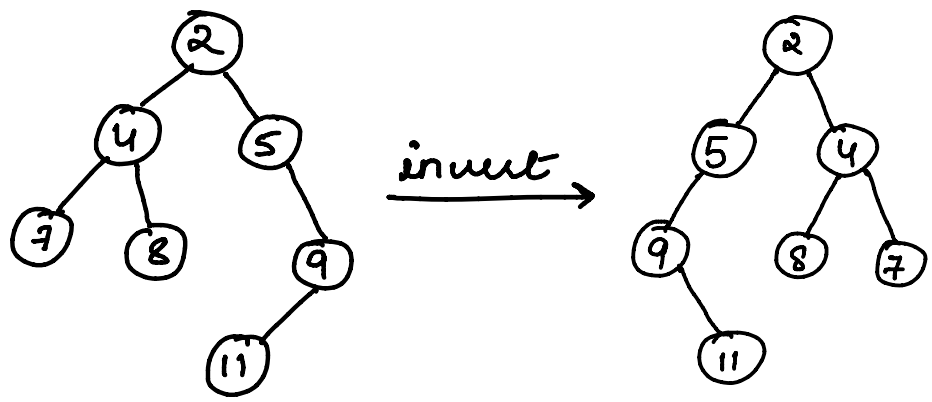
Sc → O(1) + O(min(h1, h2))

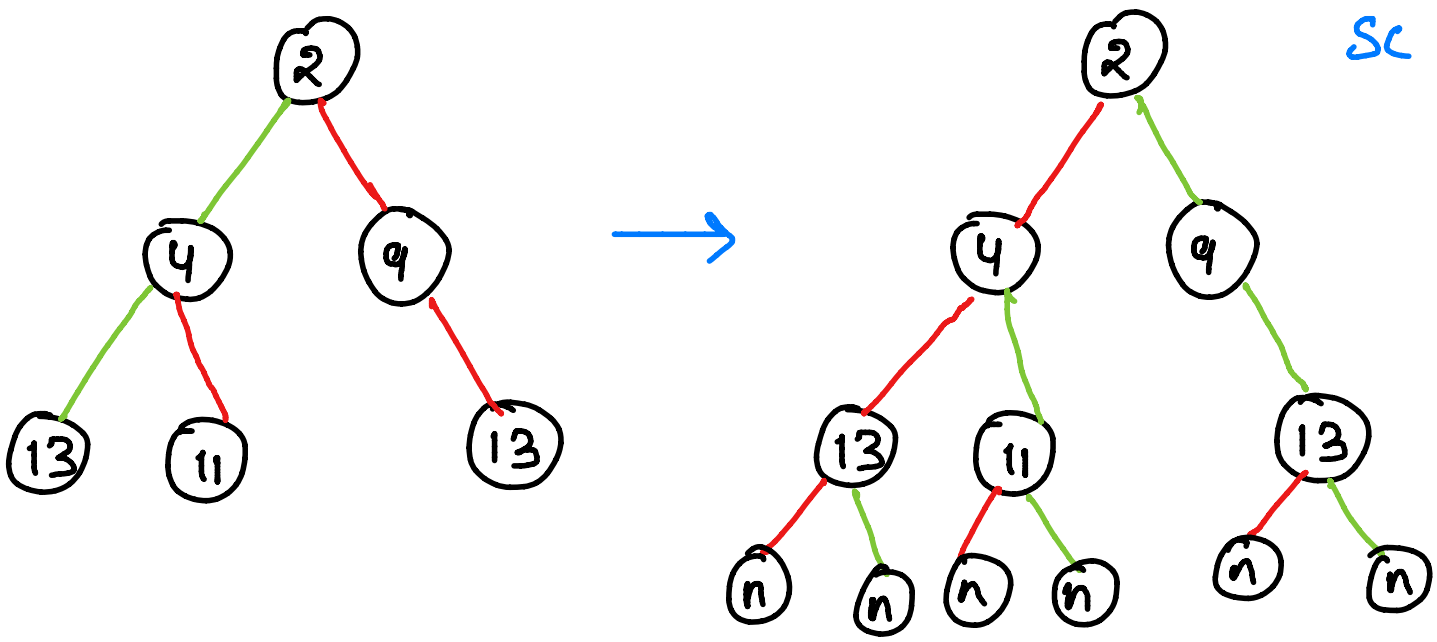
code →

```java
class Solution {
    public boolean isSameTree(TreeNode p, TreeNode q) {
        if(p == null && q == null)
        return true;
        else if(p == null || q == null)
        return false;
        return ((p.val==q.val) && (isSameTree(p.left,q.left)) && (isSameTree(p.right,q.right)));
    }
}
```

(8) Invert Binary Tree → given the root of BT, find its mirror img.

Eg





TC → O(n)

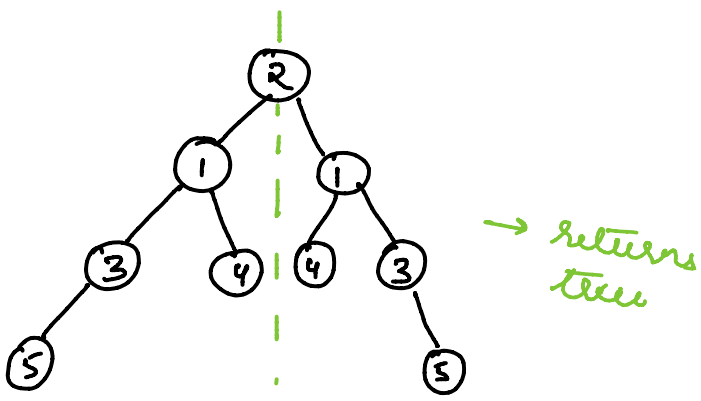
SC → O(n)+O(h)

Code →

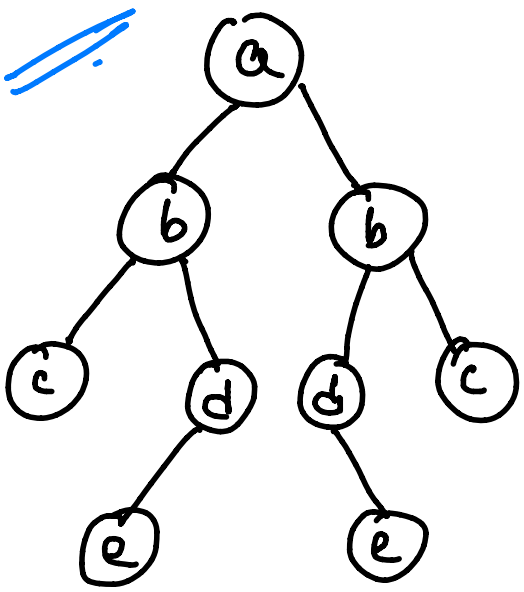
```java
class Solution {
    public TreeNode invertTree(TreeNode root) {
        if(root == null)
        return root;
        invertTree(root.left);
        invertTree(root.right);
        TreeNode a = root.left;
        root.left=root.right;
        root.right=a;
        return root;
    }
}
```

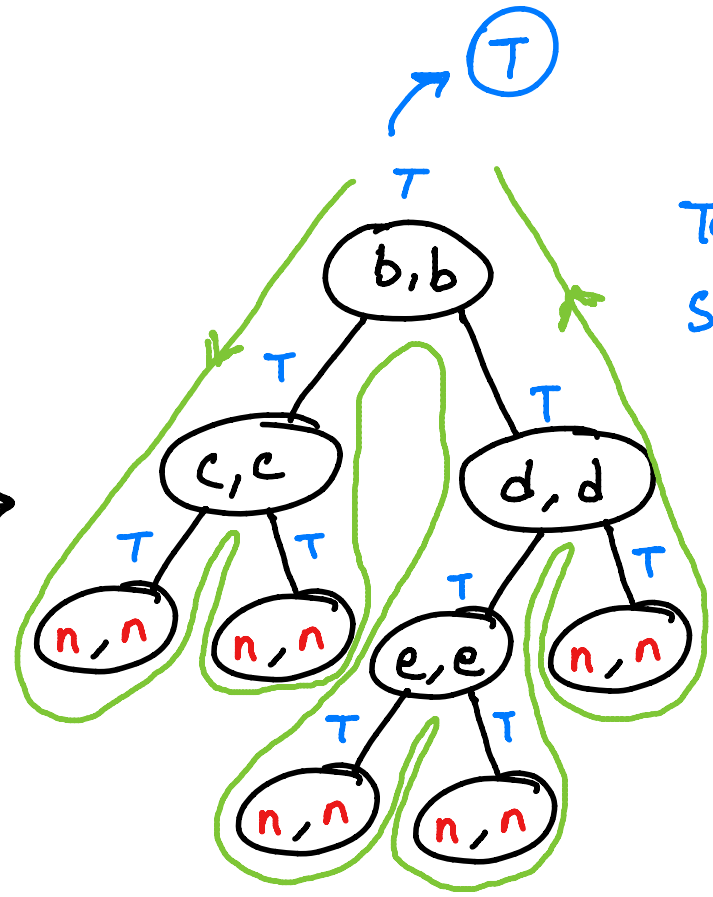
D7 (19) Symmetric Tree →

return true if left subtree is equal to right subtree, else return false



→ returns true





Tc → O(n)

Sc → O(1) + O(h)

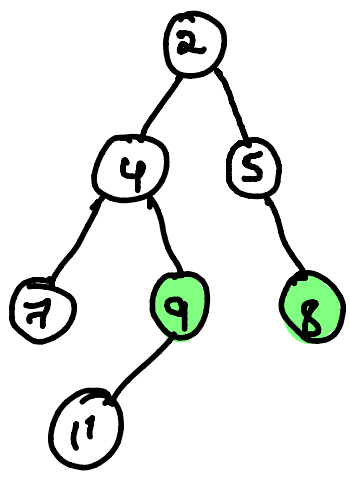
Code →

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

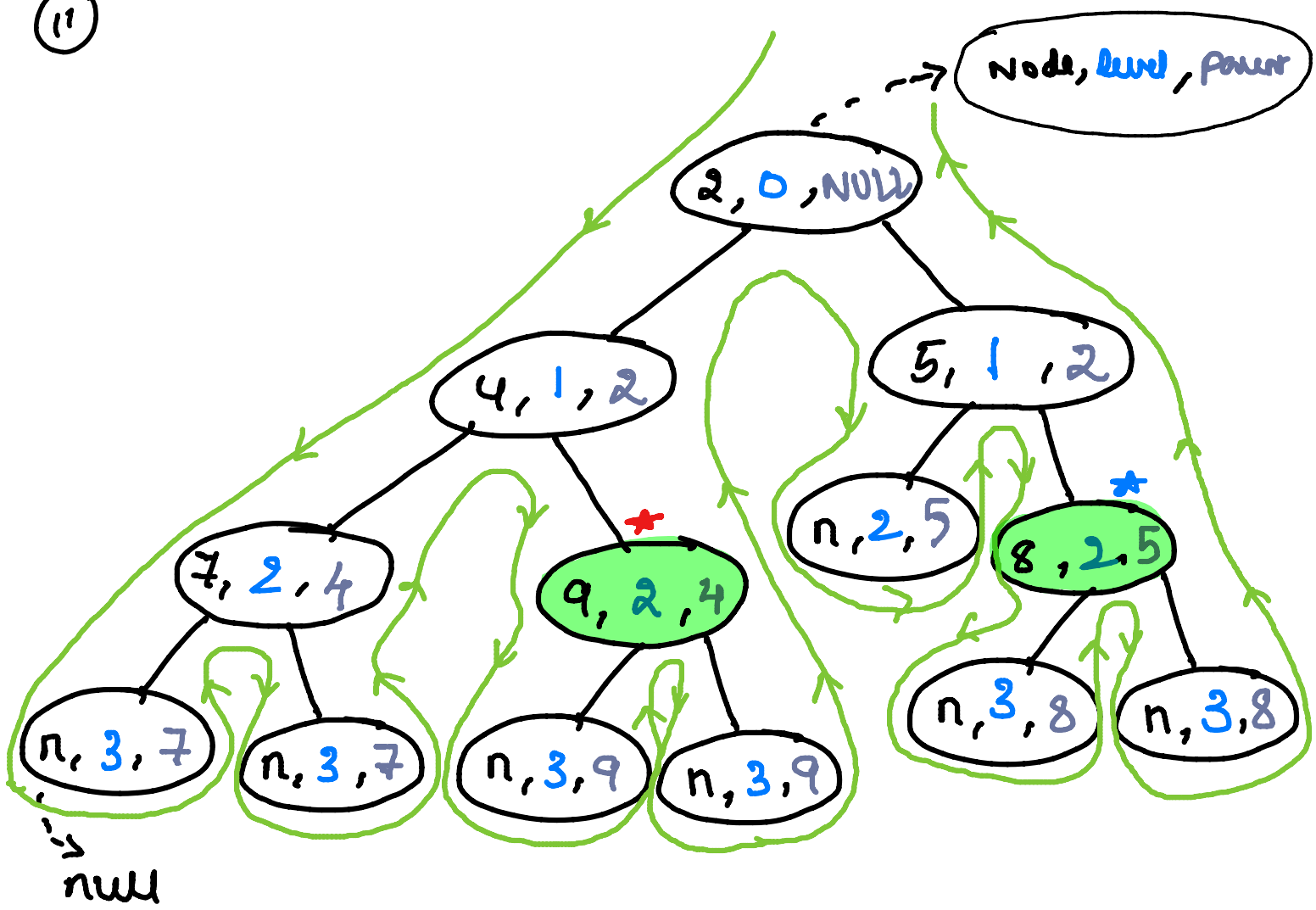
(20) Cousins of a Binary Tree → given two nodes, find if they are cousins of each other.

↳ same level but diff parents.

Eg



x = 9
y = 8
} cousins



* at this step as value = 9 is
x is found store it's parent & level in separate variables
* later compare its value with other ocurance in Y
such that

1) x.parent != y.parent
2) x.level == y.level.

TC → O(n)

SC → O(1)

Recursive stack → O(n)

Code

```java
class Solution {
    public boolean isCousins(TreeNode root, int x, int y) {
        if(root == null)
            return false;
        Queue<TreeNode> q = new LinkedList();
        q.add(root);
        while(q.size()>0){
            List<Integer> level = new ArrayList();
            int size = q.size();
            for(int i = 0;i<size;i++){
                TreeNode temp = q.remove();
                if(temp.left != null && temp.right != null){
                    if(temp.left.val == x && temp.right.val == y)
                        return false;
                    if(temp.right.val == x && temp.left.val == y)
                        return false;
                }
                level.add(temp.val);

                if(temp.left != null)
                    q.add(temp.left);
                if(temp.right != null)
                    q.add(temp.right);
            }
            if(level.contains(x) && level.contains(y))
                return true;
        }
            return false;
    }
}
```