**Binary tree iterative traversal**

**GFG: Boundary Traversal of binary tree**

Given a Binary Tree, find its Boundary Traversal. The traversal should be in the following order:

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

1

/ \

2 3

/ \ / \

4 5 6 7

/ \

8 9

Output: 1 2 4 8 9 6 7 3

**GFG:** **Diagonal Traversal of Binary Tree**

Given a Binary Tree, print the **diagonal traversal** of the binary tree.

Consider lines of slope -1 passing between nodes. Given a Binary Tree, print all diagonal elements in a binary tree belonging to same line.  
If the diagonal element are present in two different subtress then left subtree diagonal element should be taken first and then right subtree.

  8

  / \

  3 10

  / \ \

  1 6 14

  / \ /

  4 7 13

**Output** : 8 10 14 3 6 7 13 1 4

**102. Binary Tree Level Order Traversal**

Given the root of a binary tree, return *the level order traversal of its nodes' values*. (i.e., from left to right, level by level).



**Input:** root = [3,9,20,null,null,15,7]

**Output:** [[3],[9,20],[15,7]]

**103. Binary Tree Zigzag Level Order Traversal**

Given the root of a binary tree, return *the zigzag level order traversal of its nodes' values*. (i.e., from left to right, then right to left for the next level and alternate between).



**Input:** root = [3,9,20,null,null,15,7]

**Output:** [[3],[20,9],[15,7]]

**GFG: Vertical Traversal of Binary Tree**

Given a Binary Tree, find the vertical traversal of it starting from the leftmost level to the rightmost level.  
If there are multiple nodes passing through a vertical line, then they should be printed as they appear in **level order** traversal of the tree.

1

/ \

2 3

/ \ / \

4 5 6 7

\ \

8 9

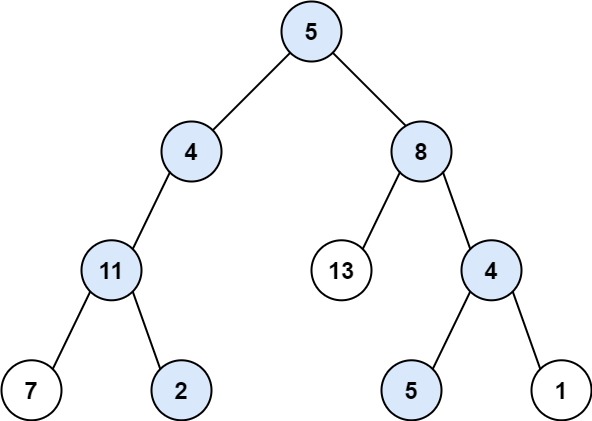
**Output:**

4 2 1 5 6 3 8 7 9

**113. Path Sum II**

Given the root of a binary tree and an integer targetSum, return *all****root-to-leaf****paths where the sum of the node values in the path equals*targetSum*. Each path should be returned as a list of the node****values****, not node references*.

A **root-to-leaf** path is a path starting from the root and ending at any leaf node. A **leaf** is a node with no children.



**Input:** root = [5,4,8,11,null,13,4,7,2,null,null,5,1], targetSum = 22

**Output:** [[5,4,11,2],[5,8,4,5]]

**Explanation:** There are two paths whose sum equals targetSum:

5 + 4 + 11 + 2 = 22

5 + 8 + 4 + 5 = 22

**112. Path Sum**

Given the root of a binary tree and an integer targetSum, return true if the tree has a **root-to-leaf** path such that adding up all the values along the path equals targetSum.

A **leaf** is a node with no children.



**Input:** root = [5,4,8,11,null,13,4,7,2,null,null,null,1], targetSum = 22

**Output:** true

**Explanation:** The root-to-leaf path with the target sum is shown.

**GFG: Max Level Sum in Binary Tree**

Given a Binary Tree having positive and negative nodes. Find the maximum sum of a level in the given Binary Tree.

4

/ \

2 -5

/ \ / \

-1 3 -2 6

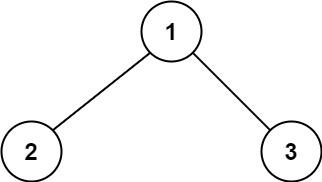
**Output:** 6

**124. Binary Tree Maximum Path Sum**

A **path** in a binary tree is a sequence of nodes where each pair of adjacent nodes in the sequence has an edge connecting them. A node can only appear in the sequence **at most once**. Note that the path does not need to pass through the root.

The **path sum** of a path is the sum of the node's values in the path.

Given the root of a binary tree, return *the maximum****path sum****of any****non-empty****path*.



**Input:** root = [1,2,3]

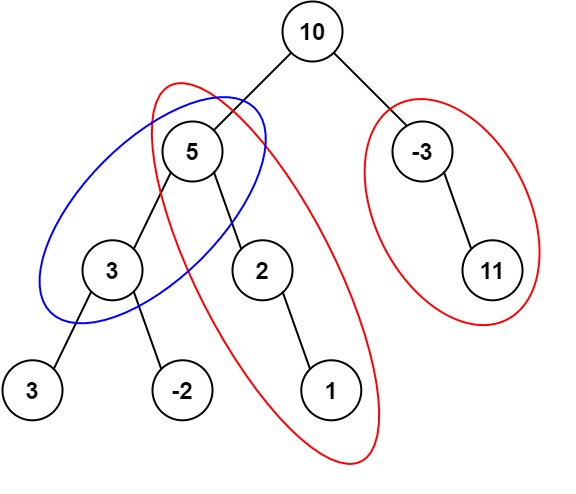
**Output:** 6

**Explanation:** The optimal path is 2 -> 1 -> 3 with a path sum of 2 + 1 + 3 = 6.

**437. Path Sum III**

Given the root of a binary tree and an integer targetSum, return the number of paths where the sum of the values along the path equals targetSum.

The path does not need to start or end at the root or a leaf, but it must go downwards (i.e., traveling only from parent nodes to child nodes).



**Input:** root = [10,5,-3,3,2,null,11,3,-2,null,1], targetSum = 8

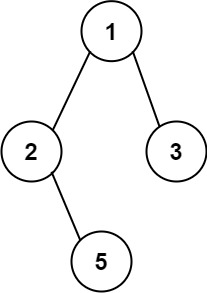
**Output:** 3

**Explanation:** The paths that sum to 8 are shown.

**257. Binary Tree Paths**

Given the root of a binary tree, return all root-to-leaf paths in ***any order***.

A **leaf** is a node with no children.



**Input:** root = [1,2,3,null,5]

**Output:** ["1->2->5","1->3"]

**404. Sum of Left Leaves**

Given the root of a binary tree, return the sum of all left leaves.

A **leaf** is a node with no children. A **left leaf** is a leaf that is the left child of another node.



**Input:** root = [3,9,20,null,null,15,7]

**Output:** 24

**Explanation:** There are two left leaves in the binary tree, with values 9 and 15 respectively.

**Misc**

**GFG: Bottom View of a Binary Tree**

Given a Binary Tree, The task is to print the **bottom view** from left to right. A node **x**is there in output if x is the bottommost node at its horizontal distance. The horizontal distance of the left child of a node x is equal to a horizontal distance of x minus 1, and that of a right child is the horizontal distance of x plus 1.

***Input:****20  
                    /     \  
                8         22  
             /      \         \  
          5         3        25  
                   /    \        
              10       14*

***Output:****5, 10, 3, 14, 25.*

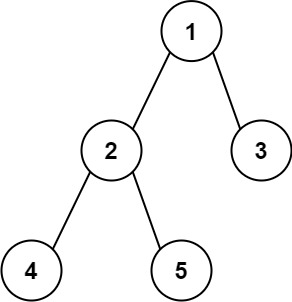
**543. Diameter of Binary Tree**

Given the root of a binary tree, return *the length of the****diameter****of the tree*.

The **diameter** of a binary tree is the **length** of the longest path between any two nodes in a tree. This path may or may not pass through the root.

The **length** of a path between two nodes is represented by the number of edges between them.

**Example 1:**



**Input:** root = [1,2,3,4,5]

**Output:** 3

**Explanation:** 3 is the length of the path [4,2,1,3] or [5,2,1,3].

Sol:

Brute force approach is find the height of left subtree and right subtree of each node and the max of left and right subtree height + 1 will be the diameter of the BT. Its TC = n2

Optimized:

While finding the height of the tree itself just pass one paratmeter max and after the left and right recursive call just find the max = Math.max(max, lh + rh);

**236. Lowest Common Ancestor of a Binary Tree**

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

According to the [definition of LCA on Wikipedia](https://en.wikipedia.org/wiki/Lowest_common_ancestor): “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow **a node to be a descendant of itself**).”



**Input:** root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1

**Output:** 3

**Explanation:** The LCA of nodes 5 and 1 is 3.

Sol:

Brute force approach is: find the path from root to node 5 which is 3, 5 and root to node path from root to node 1 is 3, 1

Now traverse the both array and find the last matching element which is 3 is the LCA of BT. Here we are taking extra space to store the path for two nodes.

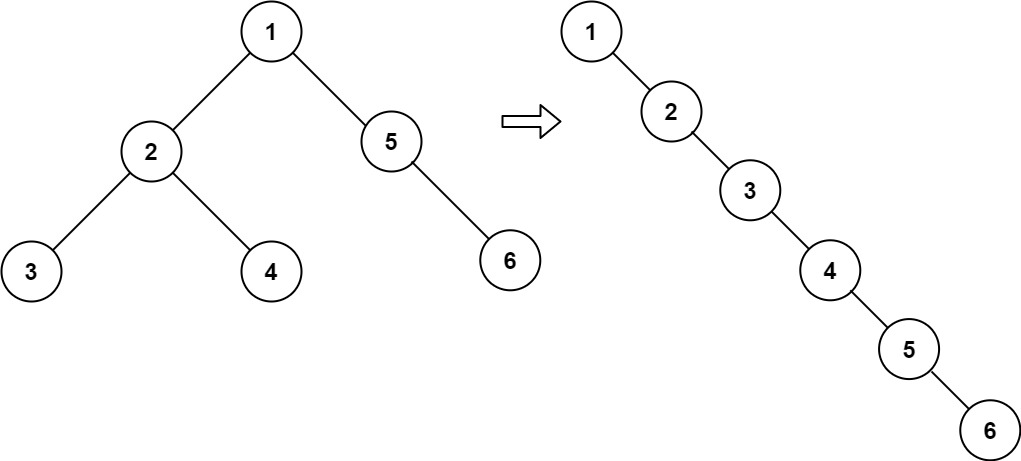
Optimized:

While doing traversal itself we will check if left == null then take right one….if right == null then take left one. If both left and right is not null then this is the LCA.

**114. Flatten Binary Tree to Linked List**

Given the root of a binary tree, flatten the tree into a "linked list":

* The "linked list" should use the same TreeNode class where the right child pointer points to the next node in the list and the left child pointer is always null.
* The "linked list" should be in the same order as a [**pre-order traversal**](https://en.wikipedia.org/wiki/Tree_traversal#Pre-order,_NLR) of the binary tree.



**Input:** root = [1,2,5,3,4,null,6]

**Output:** [1,null,2,null,3,null,4,null,5,null,6]

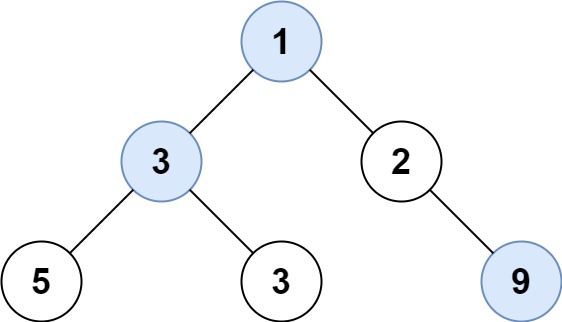
Sol:

Take a global variable Node pre = null;

Do a reverse post order traversal and in last do the following



**515. Find Largest Value in Each Tree Row**

Given the root of a binary tree, return *an array of the largest value in each row* of the tree **(0-indexed)**. 

**Input:** root = [1,3,2,5,3,null,9]

**Output:** [1,3,9]

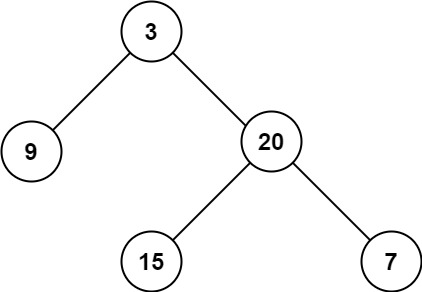
Sol:

Do the level order

**104. Maximum Depth of Binary Tree**

Given the root of a binary tree, return its maximum depth.

A binary tree's **maximum depth** is the number of nodes along the longest path from the root node down to the farthest leaf node.



**Input:** root = [3,9,20,null,null,15,7]

**Output:** 3

Sol:

Height of the BT is nothing but maximum depth of the BT

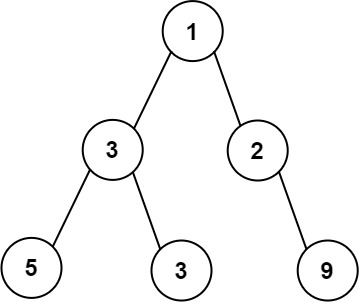
**662. Maximum Width of Binary Tree**

Given the root of a binary tree, return the ***maximum width*** of the given tree.

The **maximum width** of a tree is the maximum **width** among all levels.

The **width** of one level is defined as the length between the end-nodes (the leftmost and rightmost non-null nodes), where the null nodes between the end-nodes that would be present in a complete binary tree extending down to that level are also counted into the length calculation.

It is **guaranteed** that the answer will in the range of a **32-bit** signed integer.



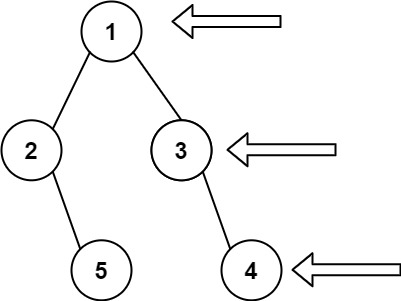
**Input:** root = [1,3,2,5,3,null,9]

**Output:** 4

**Explanation:** The maximum width exists in the third level with length 4 (5,3,null,9).

**199. Binary Tree Right Side View**

Given the root of a binary tree, imagine yourself standing on the **right side** of it, return the values of the nodes you can see ordered from top to bottom.



**Input:** root = [1,2,3,null,5,null,4]

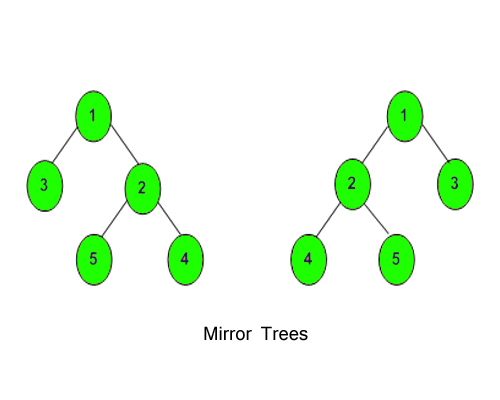
**Output:** [1,3,4]

Sol:

Do the level order traversal in reverse order... I mean first take right node then take left node. And then take 1st node of each level.

**Construction:**

**GFG : Mirror Tree**

Given a Binary Tree, convert it into its mirror.  


**Input:**

1

  / \

  2 3

**Output:** 3 1 2

**Explanation:** The tree is

   1   (mirror) 1

/  \   =>      /  \

2    3          3   2

The inorder of mirror is 3 1 2

**GFG: Transform to Sum Tree**

Given a Binary Tree of size N , where each node can have positive or negative values. Convert this to a tree where each node contains the sum of the left and right sub trees of the original tree. The values of leaf nodes are changed to 0

**Input:**

10

/ \

-2 6

/ \ / \

8 -4 7 5

**Output:**

20

/ \

4 12

/ \ / \

0 0 0 0

**Explanation:**

(4-2+12+6)

/ \

(8-4) (7+5)

/ \ / \

0 0 0 0

**Check and print:**

**GFG:** **Sum Tree**

Given a Binary Tree. Return **true** if, for every node **X** in the tree other than the leaves, its value is equal to the sum of its left subtree's value and its right subtree's value. Else return **false**.

An empty tree is also a Sum Tree as the sum of an empty tree can be considered to be 0. A leaf node is also considered a Sum Tree.

**Input:**

3

/ \

1 2

**Output:** 1

**Explanation:**

The sum of left subtree and right subtree is

1 + 2 = 3, which is the value of the root node.

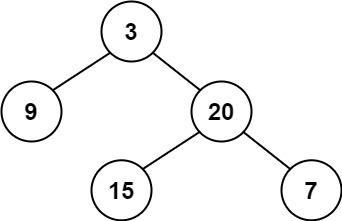
Therefore, the given binary tree is a **sum tree**.

**110. Balanced Binary Tree**

Given a binary tree, determine if it is height-balanced.

For this problem, a height-balanced binary tree is defined as:

a binary tree in which the left and right subtrees of *every* node differ in height by no more than 1.



**Input:** root = [3,9,20,null,null,15,7]

**Output:** true

Sol:

Bruteforce approach is find the height of left and right subtree of each node and then check its diff. if its > 1 then its not a balanced tree

Optimized:

While finding the height of the BT we will check if Math.abs(left – right) > 1 then return -1 else return height of BT.

**GFG : Perfect Binary Tree**

Given a Binary Tree, write a function to check whether the given Binary Tree is a prefect Binary Tree or not. A Binary tree is Perfect Binary Tree in which all internal nodes have two children and all leaves are at same level.

**Input:**

7

  / \

  4 9

**Output:** YES

**Explanation:**

As the root node 7 has two children and

two leaf nodes 4 and 9 are at same level

so it is a perfect binary tree.

**Sol:**

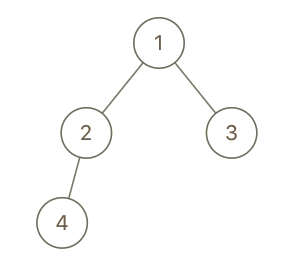
Find the depth of any leaf node. Once we get the depth of any node then we will check all the leaf node its level is same as depth. If any node having only one child, then return false

**993. Cousins in Binary Tree**

Given the root of a binary tree with unique values and the values of two different nodes of the tree x and y, return true if the nodes corresponding to the values x and y in the tree are ***cousins***, or false otherwise.

Two nodes of a binary tree are **cousins** if they have the same depth with different parents.

Note that in a binary tree, the root node is at the depth 0, and children of each depth k node are at the depth k + 1.



**Input:** root = [1,2,3,4], x = 4, y = 3

**Output:** false

Sol:

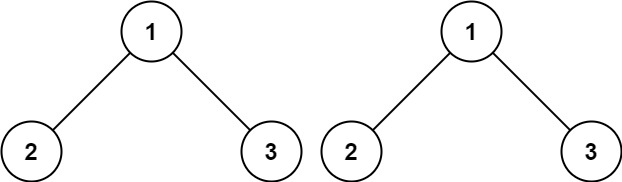
Check the level of both node. It should be same.

Then both node should not be sibling. It means its parent should not be the same node.

**100. Same Tree**

Given the roots of two binary trees p and q, write a function to check if they are the same or not.

Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.



**Input:** p = [1,2,3], q = [1,2,3]

**Output:** true

**101. Symmetric Tree**

Given the root of a binary tree, *check whether it is a mirror of itself* (i.e., symmetric around its center).



**Input:** root = [1,2,2,3,4,4,3]

**Output:** true

**Print nodes at k distance from root**

Given a Binary Tree of size **N** and an integer **K**. Print all nodes that are at distance k from root (root is considered at distance 0 from itself). Nodes should be printed from left to right. If k is more that height of tree, nothing should be printed

For example, if below is given tree and k is 2. Output should be 4 5 6.

          1  
       /     \  
     2        3  
   /         /   \  
  4        5    6   
     \  
      8