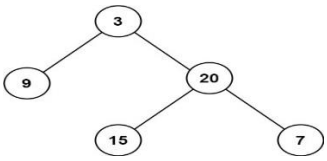
1) **Maximum Depth of Binary Tree**:

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]

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

Output: 3

Example 2:

Input: root = [1,null,2]

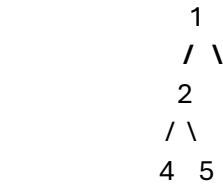
Output: 2

```
int maxDepth(TreeNode* root)
 {
   //Base case:
   if(root==NULL)
     return 0; //return 0 height if root is null
   int ltree = maxDepth(root->left);
   int rtree = maxDepth(root->right);
   return 1+ max(ltree, rtree);
 }
```

- Time complexity: $O(n)$
- Space complexity: $O(n)$

---

2) **Binary Tree Preorder Traversal**: Given the root of a binary tree, return *the preorder traversal of its nodes' values*.



Output: 1 2 4 5 3

Recursive traversal:

```
vector<int> Preorder(TreeNode* root)
{
  vector<int>ans;
  if (root == NULL)
    return ans;

  // Deal with the node
  ans.push_back(root->val);

  // Recur on left subtree
  Preorder(root->left);

  // Recur on right subtree
  Preorder(root->right);
}
```

//Iterative way of traversal

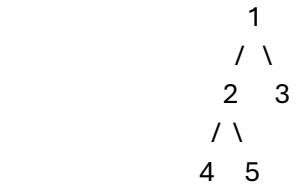
```
//Time complexity:O(n)
//Space complexity:O(n)
vector<int> preorderTraversal(TreeNode* root)
  {
    vector<int>ans;
    // Base Case
  if (root == NULL)
    return ans;

  // Create an empty stack and push root to it
  stack<TreeNode*> nodeStack;
  nodeStack.push(root);

  /* Pop all items one by one. Do following for every popped item
    a) print it
    b) push its right child
    c) push its left child
  Note that right child is pushed first so that left is processed first */
  while (!nodeStack.empty())
```

```
  {
    // Pop the top item from stack and print it
    root = nodeStack.top();
    ans.push_back(root->val);
    nodeStack.pop();

    // Push right and left children of the popped node to stack
    if (root->right)
      nodeStack.push(root->right);
    if (root->left)
      nodeStack.push(root->left);
  }
  return ans;
 }
```

Time complexity: O(n)
Space complexity: O(n)

---

3) **Binary Tree Inorder Traversal**: Given the root of a binary tree, return *the inorder traversal of its nodes' values*.



O/P: 4 2 5 1 3

Recursive Approach:

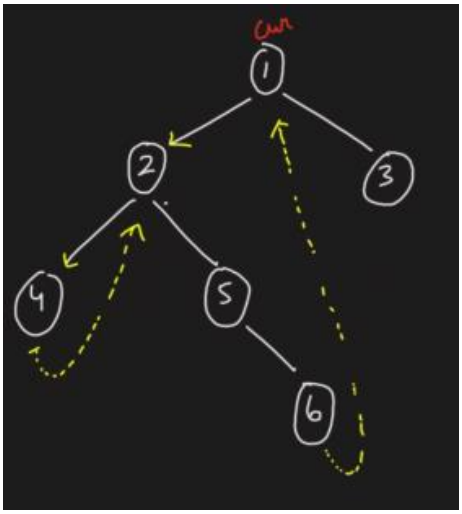
```
vector<int> inOrder(TreeNode* root)
{
   vector<int>ans;
   if (root == NULL)
      return ans;

  inOrderTrav(curr -> left);
  inOrder.push_back(curr -> data);
  inOrderTrav(curr -> right);
}
```

Time complexity: O(n)
Space complexity: O(n)

```
//Iterative function for inorder tree traversal
 vector<int> inorderTraversal(TreeNode* root)
 {
   stack<Node*> s;
    Node* curr = root;

    while (curr != NULL || !s.empty())
{

      // Reach the left most Node of the
      // curr Node
      while (curr != NULL) {

        // Place pointer to a tree node on
        // the stack before traversing
        // the node's left subtree
        s.push(curr);
        curr = curr->left;
      }

      // Current must be NULL at this point
      curr = s.top();
      s.pop();

     ans.push_back(curr->val);

      // we have visited the node and its
      // left subtree.  Now, it's right
      // subtree's turn
      curr = curr->right;

    }
return ans;
}
```

Time complexity: O(n)
Space complexity: O(1)

Morris inorder traversal:

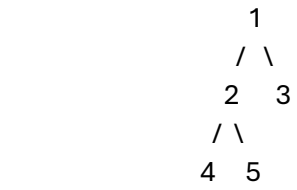



```
 vector<int> MorrisTraversal(TreeNode* root)
{
   vector<int>ans;
   if (root == NULL)
      return;

   TreeNode* current = root;
   while (current != NULL)
  {

      if (current->left == NULL)
     {
         ans.push_back(root->val);
         current = current->right;
      }
      else
     {

         /* Find the inorder predecessor of current */
         pre = current->left;
         while (pre->right != NULL && pre->right != current)
            pre = pre->right;

         /* Make current as the right child of its
           inorder predecessor */
         if (pre->right == NULL) {
            pre->right = current;
            current = current->left;
         }
```

```
        /* Revert the changes made in the 'if' part to
          restore the original tree i.e., fix the right
          child of predecessor */
        else {
          pre->right = NULL;
          ans.push_back(root->val);
          current = current->right;
        } /* End of if condition pre->right == NULL */
      } /* End of if condition current->left == NULL*/
    } /* End of while */
          return ans;
}
```

---

4) **Binary Tree Postorder Traversal**: Given the root of a binary tree, return *the postorder traversal of its nodes' values*.



Using Recursion:

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

  // First recur on left subtree
  printPostorder(node->left);

  // Then recur on right subtree
  printPostorder(node->right);

  // Now deal with the node
  cout << node->data << " ";
}
```

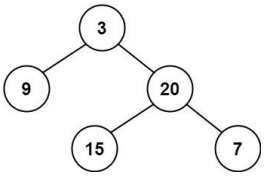
Using Iterative method:

```
vector<int> postorderTraversal(TreeNode* root) {
  vector<int> result;
  if (!root) return result;

  stack<TreeNode*> s;
  TreeNode* lastVisited = nullptr;
```

```
  TreeNode* current = root;

  while (current || !s.empty()) {
    // Go to the leftmost node
    while (current) {
      s.push(current);
      current = current->left;
    }

    // Peek the node on top of the stack
    TreeNode* node = s.top();

    // If the right child exists and hasn't been visited, process the right subtree
    if (node->right && lastVisited != node->right) {
      current = node->right;
    } else {
      // Process the node
      result.push_back(node->val);
      lastVisited = node;
      s.pop();
    }
  }

  return result;
}
```

---

5) **Balanced Binary Tree**: Given a binary tree, determine if it is **height-balanced**



Binary tree would be balanced, when abs(ltree-rtree))<=1

Approach:

1) Base case: if the current node is NULL, return 0 (height of an empty tree)
2) Recursively calculate the height of the left subtree, If the left subtree is unbalanced, propagate the unbalance status(-1)
3) Recursively calculate the height of the right subtree, If the right subtree is unbalanced, propagate the unbalance status(-1)
4) Check if the difference in height between left and right subtrees is greater than 1, If it's greater, the tree is unbalanced,
  return -1 to propagate the unbalance status
5) Return the maximum height of left and right subtrees, adding 1 for the current node

Time Complexity: O(N)
Space Complexity: O(1) Extra Space + O(H) Recursion Stack space (Where “H” is the height of binary tree)

*/

```
 bool isBalanced(Node* root) {
     // Check if the tree's height difference
     // between subtrees is less than 2
     // If not, return false; otherwise, return true
     return dfsHeight(root) != -1;
  }

  // Recursive function to calculate
  // the height of the tree
  int dfsHeight(Node* root) {
     // Base case: if the current node is NULL,
     // return 0 (height of an empty tree)
     if (root == NULL) return 0;

     // Recursively calculate the
     // height of the left subtree
     int leftHeight = dfsHeight(root->left);

     // If the left subtree is unbalanced,
     // propagate the unbalance status
     if (leftHeight == -1)
        return -1;

     // Recursively calculate the
     // height of the right subtree
     int rightHeight = dfsHeight(root->right);

     // If the right subtree is unbalanced,
     // propagate the unbalance status
     if (rightHeight == -1)
        return -1;

     // Check if the difference in height between
     // left and right subtrees is greater than 1
     // If it's greater, the tree is unbalanced,
     // return -1 to propagate the unbalance status
     if (abs(leftHeight - rightHeight) > 1)
        return -1;
```

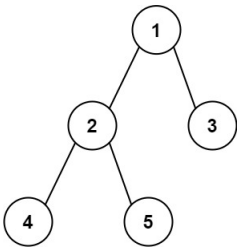
```
    // Return the maximum height of left and
    // right subtrees, adding 1 for the current node
    return max(leftHeight, rightHeight) + 1;
  }
```

---

6) **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.



/*Diameter: 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.

Approach:

1) Since Daimeter is max path between two nodes => (ltree + rtree) at any node => ltree/rthree of nth node would be height of n-1th node

we should calculate and return height of every node, that would become ltree or rtee

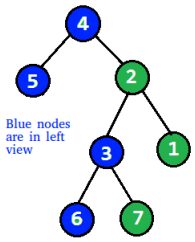
2) Diameter = max(diameter, ltree + rtree)

```
int diameterOfBinaryTree(TreeNode* root) {
    int diameter = 0;
    height(root, diameter);
    return diameter;
  }

  int height(TreeNode* root, int& diameter) {

    if (root==NULL)
    {
      return 0;
    }

    int lh = height(root->left, diameter);
    int rh = height(root->right, diameter);
```

```
    diameter = max(diameter, lh + rh);

    return 1 + max(lh, rh);
  }
```

---

7) **Print Left View of a Binary Tree**



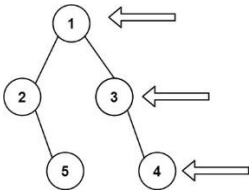
Time Complexity: O(N)
Space Complexity: O(H) (H -> Height of the Tree)

Approach:
We need to follow approach: Rt L R(preorder-recursive), can not follow level order traversal as it will increase time complexity
Used vector to store node value
When vector::size() == level means first node of that level:

```
void checkLeftView(Node* root, int level, vector<int>&ans)
{
        if(root == NULL)
                return;
        if(ans.size()==level)
                ans.push_back(root->data);

 checkLeftView(root->left, level+1, ans);
        checkLeftView(root->right, level+1, ans);

}

vector<int> leftView(Node *root)
{
  vector<int>ans;
  checkLeftView(root, 0, ans);
  return ans;
}
```

---

8) **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*.



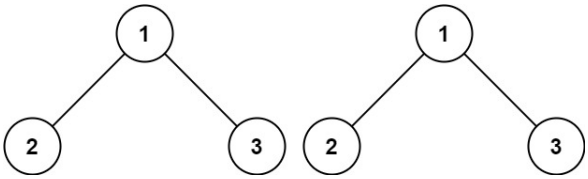
Time Complexity: O(N)
Space Complexity: O(H) (H -> Height of the Tree)

Approach:
We need to follow approach: Rt R L(preorder-recursive), can not follow level order traversal as it will increase time complexity
Used vector to store node value
When vector::size() == level means first node of that level:

```
void checkRightView(Node* root, int level, vector<int>&ans)
{
        if(root == NULL)
                return;
        if(ans.size()==level)
                ans.push_back(root->data);

   checkRightView(root->right, level+1, ans);
   checkRightView(root->left, level+1, ans);

}

vector<int> rightView(Node *root)
{
  vector<int>ans;
  checkRightView(root, 0, ans);
  return ans;
}
```

---

9) **Same Tree**/Identical 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.



Approach:

1) We will call method isSametree with two args p and q

Will check if p->val == q->val && then will call isSameTree(p->left, q->left) && isSameTree(p->right, q->right) to ensure all returns true.

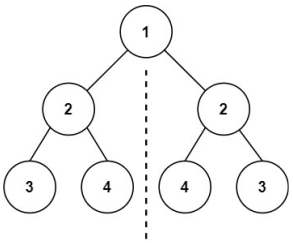
2)

3) Check if left or right null, if anyone is null then return lt==rt

```
bool isSameTree(TreeNode* p, TreeNode* q)
  {
    if(p==NULL || q==NULL )
    {
      return p==q;
    }

    return (p->val == q->val) && isSameTree(p->left, q->left) && isSameTree(p->right, q->right);

  }
```

---

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



Approach:

1) We will call method isSymmetricBT with two args root->left and root->right

2) Check if left or right null, if anyone is null then return lt==rt

3) Check lt->val==rt->val && recursively check below condition:

(lt->left, rt->right) &&(lt->right, rt->left)

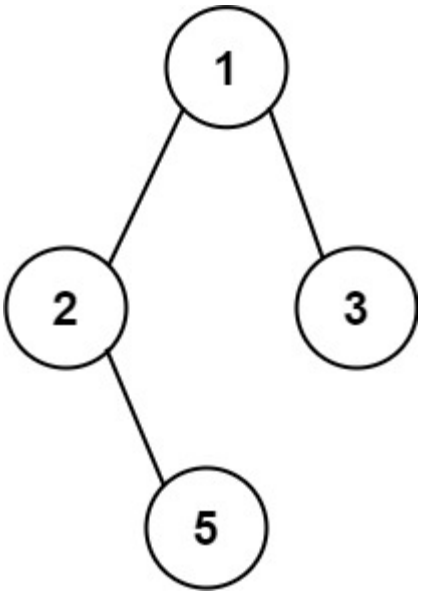
```
bool isSymmetric(TreeNode* root)
  {
    if(root==NULL)
      return true;
    return isSymmetricBT(root->left, root->right);
  }
private:
  bool isSymmetricBT(TreeNode* lt, TreeNode* rt)
  {
    if(lt==NULL || rt==NULL)
      return lt==rt;
```

```
    return (lt->val==rt->val) && isSymmetricBT(lt->right, rt->left) && isSymmetricBT(lt->left, rt->right);
  }
```

---

11) **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.

Example 1:



Input: root = [1,2,3,null,5]
Output: ["1->2->5","1->3"]
Example 2:
Input: root = [1]
Output: ["1"]

Approach:

- will call binaryTreePaths() and inside it will call paths(TreeNode* root, string str, vector<string> &ans) => will pass default string "" to identify empty/first time called to append only value to empty string.
- If str is not empty then we should append -> and root->val
- Check if root is leaf: if(root->left == NULL && root->right ==NULL) , means we found first path which will be pushed back to ans.
- Otherwise recursive call on left subtree and right subtree until we get root == null.

```
void paths(TreeNode* root, string str, vector<string> &ans) {
    if(root == NULL) return;

   //It means, it's first time and we should append root->val to str
    if(str == "")
```

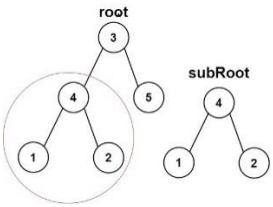
```
            str+=  to_string(root->val);

        //It means, it's not first time and we should append -> to str
        else
            str+= "->"+to_string(root->val);

        //We found leaf node and now we got result string and can be added to ans.
        if(root->left == NULL && root->right ==NULL)
          ans.push_back(str);
        else {
          paths(root->left, str, ans);
          paths(root->right, str, ans);
        }
    }
    vector<string> binaryTreePaths(TreeNode* root) {
        vector<string> ans;
        paths(root, "",ans);
        return ans;
    }
```

---

12) **Subtree of Another Tree**: Given the roots of two binary trees root and subRoot, return true if there is a subtree of root with the same structure and node values of subRoot and false otherwise.
A subtree of a binary tree tree is a tree that consists of a node in tree and all of this node's descendants. The tree tree could also be considered as a subtree of itself.

Approach:

- First, check if subroot(root of smaller tree) is null, if null then return true, because tree with null root will be subroot of any tree.
- Now check if root of main tree is null, if so, then return false
- Now check if both tree are identical, if yes then return true.
- Finally, recursively call issubtree function for left subtree and subroot and right subtree with subroot, either should be true to return true.




```
bool isSubtree(TreeNode* root, TreeNode* subRoot) {
    if(subRoot==NULL)
      return true;
    if(root==NULL)
      return false;
```

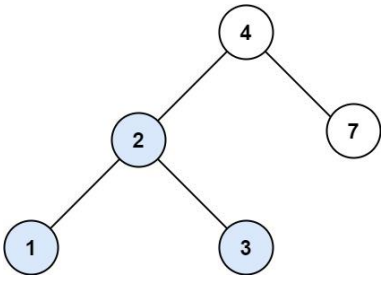
```
    if(isidentical(root,subRoot))
      return true;
    return isSubtree(root->left,subRoot) || isSubtree(root->right,subRoot);
  }

  bool isidentical(TreeNode* a, TreeNode* b){
    if(a==NULL || b==NULL)
      return a==b;
    return (a->val == b->val) && isidentical(a->left,b->left) && isidentical(a->right,b->right);
  }
```

---

12) **Search in a Binary Search Tree**:

You are given the root of a binary search tree (BST) and an integer val.

Find the node in the BST that the node's value equals val and return the subtree rooted with that node. If such a node does not exist, return null.



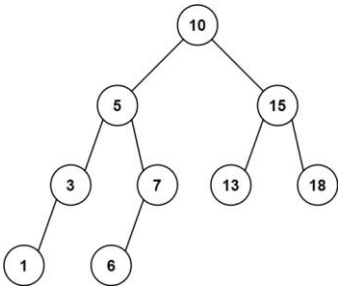
Input: root = [4,2,7,1,3], val = 2

Output: [2,1,3]

```
TreeNode* searchBST(TreeNode* root, int val) {
    if(root == NULL || root->val == val) return root;
    if(val < root->val) return searchBST(root->left, val);
    else return searchBST(root->right, val);
  }
```

---

13) **Range Sum of BST**: Given the root node of a binary search tree and two integers low and high, return *the sum of values of all nodes with a value in the* ***inclusive*** *range* [low, high].



Input: root = [10,5,15,3,7,null,18], low = 7, high = 15

Output: 32

Explanation: Nodes 7, 10, and 15 are in the range [7, 15]. 7 + 10 + 15 = 32.

Approach:

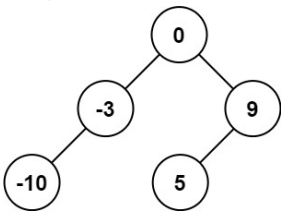
The function first checks if the root is NULL. If it is, then it returns 0. If the root is not NULL, the function recursively calls itself for the left and right subtrees, and stores the sum of values in 'left' and 'right' variables respectively.
Then, the function checks if the value of the root node falls within the given range. If it does, then it returns the sum of the root node's value, 'left' and 'right'. If it doesn't, then it returns 'left' and 'right' sum.

```
int rangeSumBST(TreeNode* root, int low, int high) {
    if(root == NULL) return 0;
    int left = rangeSumBST(root->left, low, high);
    int right = rangeSumBST(root->right, low, high);
    if(root->val >= low && root->val <= high){
      return root->val + left+right;
    }
    return left+right;
  }
```

- Time complexity: $O(n)$
- Space complexity: $O(h)$ where h is the height of the binary search tree

---

14) **Convert Sorted Array to Binary Search Tree**: Given an integer array nums where the elements are sorted in **ascending order**, convert *it to a* ***height-balanced*** *binary search tree*.

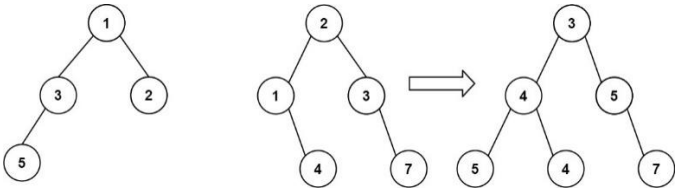


Input: nums = [-10,-3,0,5,9]
Output: [0,-3,9,-10,null,5]
Explanation: [0,-10,5,null,-3,null,9] is also accepted:

```
TreeNode*helper(vector<int>&arr,int low,int high) {
    if(low > high) return NULL;
    int mid = low+(high-low)/2;
    TreeNode*root=new TreeNode(arr[mid]);
    root->left=helper(arr,low,mid-1);
    root->right=helper(arr,mid+1,high);
    return root;

  }
  TreeNode* sortedArrayToBST(vector<int>& arr) {
    int n=arr.size();
    return helper(arr,0,n-1);
  }
```

---

15) **Merge Two Binary Trees**: You are given two binary trees root1 and root2.
Imagine that when you put one of them to cover the other, some nodes of the two trees are overlapped while the others are not. You need to merge the two trees into a new binary tree. The merge rule is that if two nodes overlap, then sum node values up as the new value of the merged node. Otherwise, the NOT null node will be used as the node of the new tree.
Return *the merged tree*.

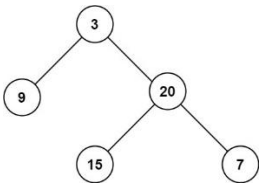


Input: root1 = [1,3,2,5], root2 = [2,1,3,null,4,null,7]
Output: [3,4,5,5,4,null,7]

```
TreeNode* mergeTrees(TreeNode* root1, TreeNode* root2) {
    if (root1==NULL) return root2; // If root1 is null, return root2
    if (root2==NULL) return root1; // If root2 is null, return root1
    // Merge the values of root1 and root2 and create a new node
    TreeNode* merged = new TreeNode(root1->val + root2->val);
    // Recursively merge the left children
    merged->left = mergeTrees(root1->left, root2->left);
    // Recursively merge the right children
    merged->right = mergeTrees(root1->right, root2->right);

    return merged;
  }
```

---

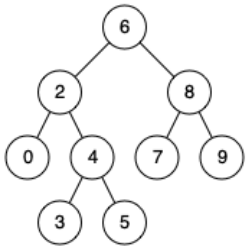
16) **Maximum Depth of Binary Tree**: Given the root of a binary tree, return *its maximum depth*. A binary tree's **maximum depth** is the number of nodes along the longest path from the root node down to the farthest leaf node.



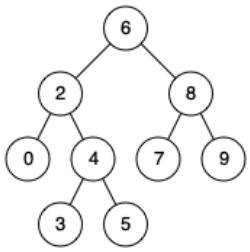

```
int maxDepth(TreeNode* root) {
    if (root == NULL)
      return 0;
    int l = maxDepth(root->left);
    int r = maxDepth(root->right);
    return max(l, r) + 1;
```

---

17) **Lowest Common Ancestor of a Binary Search Tree**: Given a binary search tree (BST), find the lowest common ancestor (LCA) node of two given nodes in the BST.
According to the definition of LCA on Wikipedia: "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 = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
Output: 6
Explanation: The LCA of nodes 2 and 8 is 6.



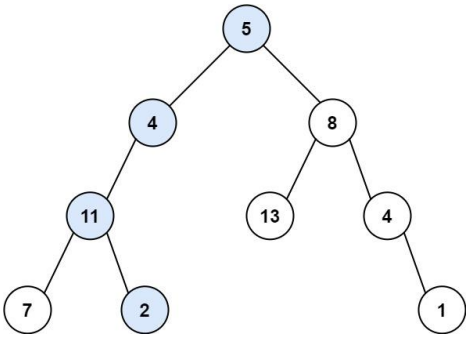
Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4
Output: 2
Explanation: The LCA of nodes 2 and 4 is 2, since a node can be a descendant of itself according to the LCA definition.

```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
    //base case
    if (root == NULL || root == p || root == q) {
      return root;
    }
    TreeNode* left = lowestCommonAncestor(root->left, p, q);
    TreeNode* right = lowestCommonAncestor(root->right, p, q);

    //result
    if(left == NULL) {
      return right;
    }
    else if(right == NULL) {
      return left;
    }
```

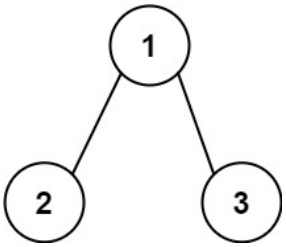
```
    else { //both left and right are not null, we found our result
      return root;
    }
  }
```

---

18) **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.



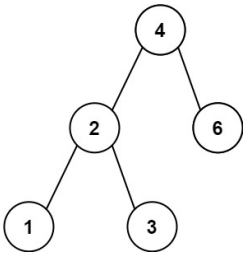
Input: root = [1,2,3], targetSum = 5
Output: false
Explanation: There two root-to-leaf paths in the tree:
(1 --> 2): The sum is 3.
(1 --> 3): The sum is 4.
There is no root-to-leaf path with sum = 5.

```
bool solve(TreeNode* &root, int &targetSum, int sum) {
    if (root == NULL)
      return false;
    sum = sum + root->val;
    if (root->left == NULL && root->right == NULL) {
      if (sum == targetSum)
        return true;
      else
        return false;
```

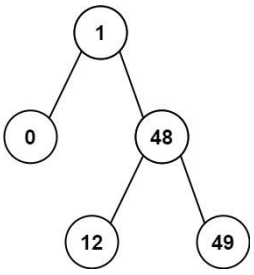
```
    }
    bool leftAns = solve(root->left, targetSum, sum);
    bool rightAns = solve(root->right, targetSum, sum);
    return leftAns || rightAns;
  }
  bool hasPathSum(TreeNode* root, int targetSum) {
    if (root == NULL)
      return false;
    int sum = 0;
    bool ans = solve(root, targetSum, sum);
    return ans;
  }
```

---

19) **Minimum Absolute Difference in BST**: Given the root of a Binary Search Tree (BST), return *the minimum absolute difference between the values of any two different nodes in the tree*.



Input: root = [4,2,6,1,3]
Output: 1



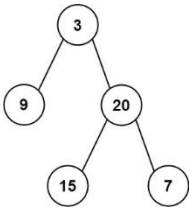
Input: root = [1,0,48,null,null,12,49]
Output: 1

```
void inorder(TreeNode* root){
    if(root){
      inorder(root->left);
      a.push_back(root->val);
      inorder(root->right);
    }
  }

  int getMinimumDifference(TreeNode* root) {
    int m=INT_MAX;
```

```
    inorder(root);


    for(int i=1;i<a.size();i++){
      m=min(m,a[i]-a[i-1]);

    }
    return m;

  }
```

---

20) **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.

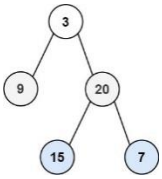


Input: root = [3,9,20,null,null,15,7]
Output: 24
Explanation: There are two left leaves in the binary tree, with values 9 and 15 respectively.

```
void solve(TreeNode* &root,int &ans){
  if(root==NULL){
    return;
  }
    if(root->left!=NULL){
      if(root->left->left==NULL && root->left->right==NULL){
        ans=ans+root->left->val;
      }

    }
    solve(root->left,ans);
    solve(root->right,ans);

}
  int sumOfLeftLeaves(TreeNode* root) {
    int ans=0;
      solve(root,ans);
      return ans;
  }
```

---

21) **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).



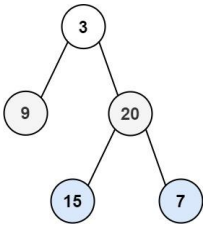
Example 1:
Input: root = [3,9,20,null,null,15,7]
Output: [[3],[9,20],[15,7]]
Example 2:
Input: root = [1]
Output: [[1]]
Example 3:
Input: root = []
Output: []

```
vector<vector<int>> levelOrder(TreeNode* root)
{
    vector<vector<int>>ans;
    //Base case
    if(root==NULL)
        return ans;
    queue<TreeNode*>q;
    q.push(root);

    while(!q.empty()){
      int s=q.size();
      vector<int>v;
      for(int i=0;i<s;i++){
        TreeNode *node=q.front();
        q.pop();
        if(node->left!=NULL)q.push(node->left);
        if(node->right!=NULL)q.push(node->right);
        v.push_back(node->val);
      }
      ans.push_back(v);
    }
    return ans;
  }
```

Time complexity: o(n)
Space complexity: O(n)

---

22) **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).




```
vector < vector < int >> zigzagLevelOrder(TreeNode * root)
 {
  vector < vector < int >> result;
  //Base case
  if (root == NULL)
  {
    return result;
  }

  queue < TreeNode * > nodesQueue;
  nodesQueue.push(root);
  bool leftToRight = true;

  while (!nodesQueue.empty())
  {
    int size = nodesQueue.size();
    //Take vector of size, will store element by index to print in left to right or     ///reverse.
    vector < int > row(size);
    for (int i = 0; i < size; i++)
    {
      TreeNode * node = nodesQueue.front();
      nodesQueue.pop();

      // find position to fill node's value
      int index = (leftToRight) ? i : (size - 1 - i);

      row[index] = node -> val;
      if (node -> left)
      {
        nodesQueue.push(node -> left);
      }
      if (node -> right)
      {
        nodesQueue.push(node -> right);
      }
    }
    // after this level
    leftToRight = !leftToRight;
    result.push_back(row);
```

```
  }
  return result;
 }
```

---

23) **Top View of Binary Tree:** Given below is a binary tree. The task is to print the top view of binary tree. Top view of a binary tree is the set of nodes visible when the tree is viewed from the top. For the given below tree

```
  1
 /   \
 2    3
/ \  /  \
4  5 6  7
```

Top view will be: 4 2 1 3 7

Approach:

we will store Node and line(level number) to queue.
we will travere in level order traversal
we will store node->data into map[line]
*/

```
vector<int> topView(Node *root)
  {
      vector<int> ans;
     if(root == NULL) return ans;
     map<int,int> mpp;
     queue<pair<Node*, int>> q;
     q.push({root, 0});
     while(!q.empty()) {
        auto it = q.front();
        q.pop();
        Node* node = it.first;
        int line = it.second;

                                //Check if level/line is not already entered then only add it in map
                                //otherwise top view data will be overwritten by bottom view nodes.
                                if(mpp.find(line) == mpp.end())
                                {
                                        mpp[line] = node->data;
                                }

        if(node->left != NULL) {
           q.push({node->left, line-1});
        }
        if(node->right != NULL) {
           q.push({node->right, line + 1});
```

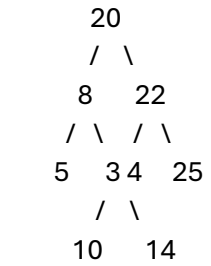
```
        }

      }

      for(auto it : mpp) {
        ans.push_back(it.second);
      }
      return ans;

    }
    Time Complexity: O(N)
    Space Complexity: O(N)
```

---

24) **Bottom View of Binary Tree:**

Given a binary tree, return an array where elements represent the bottom view of the binary tree from left to right.

Note: If there are multiple bottom-most nodes for a horizontal distance from the root, then the latter one in the level traversal is considered. For example, in the below diagram, 3 and 4 are both the bottommost nodes at a horizontal distance of 0, here 4 will be considered.



Approach:

we will store Node and line(level number) to queue.
we will travere in level order traversal
we will store node->data into map[line]
*/

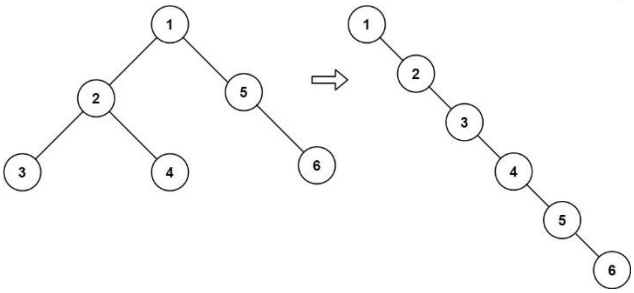
```
vector <int> bottomView(Node *root)
{
      vector<int> ans;
      if(root == NULL) return ans;
      map<int,int> mpp;
      queue<pair<Node*, int>> q;
      q.push({root, 0});
      while(!q.empty()) {
        auto it = q.front();
        q.pop();
        Node* node = it.first;
        int line = it.second;
```

```
        //Add data into map, no need to check if exist in map or not, anyway it's bottom view and will be overwritten for bottom view nodes
        mpp[line] = node->data;

        if(node->left != NULL) {
          q.push({node->left, line-1});
        }
        if(node->right != NULL) {
          q.push({node->right, line + 1});
        }

      }

      for(auto it : mpp) {
        ans.push_back(it.second);
      }
      return ans;
    }
    Time Complexity: O(N)
    Space Complexity: O(N)
```

---

25) **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** of the binary tree.



Example 1:
Input: root = [1,2,5,3,4,null,6]
Output: [1,null,2,null,3,null,4,null,5,null,6]
Example 2:
Input: root = []
Output: []
Example 3:
Input: root = [0]
Output: [0]

Morris traversal approach:

1) set current to root
2) Traverse until current is not null
3) check if current->left is not null then set prev to current->left
4) Check right of prev until it is not null, when it's null set prev->right = current->right
5) set current->right = current->left and remove current->left
6) Now set current=current->right(else part of if(current->left == NULL)

Time complecity:O(n)
Space complexity: O(1)
*/

```
void flatten(TreeNode* root)
  {
    TreeNode* current = root;
    while(current != NULL)
    {
       if(current->left != NULL)
       {
          TreeNode* prev= current->left;
          while(prev->right != NULL)
          {
             prev = prev->right;
          }
          prev->right = current->right;
          current->right= current->left;
          current->left = NULL;
       }
       current = current->right;
    }
  }
```

---

26) **Add node to leaf in binary tree:**

```
void addNodeToLeafNode(treeNode* root)
{
   // if node is null, return
   if (!root)
      return;

   // if node is leaf node, print its data
   if (!root->left && !root->right)
   {
      root->left = createNode(root->data);
      root->right = createNode(root->data);
      return;
```

```
    }

    // if left child exists, check for leaf
    // recursively
    if (root->left)
        addNodeToLeafNode(root->left);

    // if right child exists, check for leaf
    // recursively
    if (root->right)
        addNodeToLeafNode(root->right);
}

void traversePreOrder(treeNode* temp)
{
    if (temp != NULL)
    {
        cout << " " << temp->data;
        traversePreOrder(temp->left);
        traversePreOrder(temp->right);
    }
}

int main() {
    treeNode* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->right->left = createNode(5);

    //addNodeToLeafNode(root);
    traversePreOrder(root);

}
```

---

27) **Predecessor and Successor:** There is BST given with the root node with the key part as an integer only. You need to find the in-order successor and predecessor of a given key. If either predecessor or successor is not found, then set it to NULL.

Note:- In an inorder traversal the number just smaller than the target is the predecessor and the number just greater than the target is the successor**.**

Input:

```
    8
   /  \
```

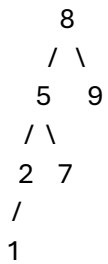
1 9
**\ **
4 10
/
3
key = 8
Output: 4 9
Explanation:
In the given BST the inorder predecessor of 8 is 4 and inorder successor of 8 is 9.
Example 2:
Input:
10
**/ **
2 11
**/ **
1 5
**/ **
3 6
****
4
key = 11
Output: 10 -1
Explanation:In given BST, the inorder predecessor of 11 is 10 whereas it does not have any inorder successor.

---

28) Check whether BST contains Dead End

Given a Binary Search Tree that contains unique positive integer values greater than 0. The task is to complete the function isDeadEnd which returns true if the BST contains a dead end else returns false. Here Dead End means a leaf node, at which no other node can be inserted.

Input :



Output :
Yes

Explanation :
Node 1 is a Dead End in the given BST.
Example 2:
Input :

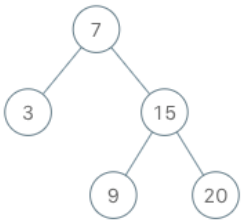
```
        8
       /  \
      7    10
     /    /  \
    2    9    13
```

Output :
Yes
Explanation :
Node 9 is a Dead End in the given BST.

---

29) **Binary Search Tree Iterator**: Implement the BSTIterator class that represents an iterator over the **in-order traversal** of a binary search tree (BST):

- BSTIterator(TreeNode root) Initializes an object of the BSTIterator class. The root of the BST is given as part of the constructor. The pointer should be initialized to a non-existent number smaller than any element in the BST.
- boolean hasNext() Returns true if there exists a number in the traversal to the right of the pointer, otherwise returns false.
- int next() Moves the pointer to the right, then returns the number at the pointer.
  Notice that by initializing the pointer to a non-existent smallest number, the first call to next() will return the smallest element in the BST.
  You may assume that next() calls will always be valid. That is, there will be at least a next number in the in-order traversal when next() is called.



Input
["BSTIterator", "next", "next", "hasNext", "next", "hasNext", "next", "hasNext", "next", "hasNext"]
[[[7, 3, 15, null, null, 9, 20]], [], [], [], [], [], [], [], [], []]
Output
[null, 3, 7, true, 9, true, 15, true, 20, false]

Explanation

```
BSTIterator bSTIterator = new BSTIterator([7, 3, 15, null, null, 9, 20]);
bSTIterator.next();    // return 3
bSTIterator.next();    // return 7
bSTIterator.hasNext(); // return True
bSTIterator.next();    // return 9
bSTIterator.hasNext(); // return True
bSTIterator.next();    // return 15
bSTIterator.hasNext(); // return True
bSTIterator.next();    // return 20
bSTIterator.hasNext(); // return False

class BSTIterator {
public:
  stack<TreeNode*> s;
  BSTIterator(TreeNode* root) {
    partialInorder(root);
  }

  void partialInorder(TreeNode* root){
    while(root != NULL){
      s.push(root);
      root = root->left;
    }
  }

  int next() {
    TreeNode* top = s.top();
    s.pop();
    partialInorder(top->right);
    return top->val;
  }

  bool hasNext() {
    return !s.empty();
  }
};
```

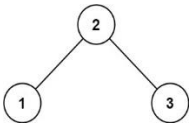
Time Complexity : O(h)

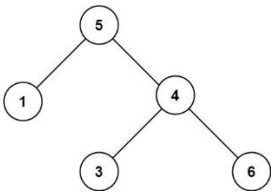
---

30) **Validate Binary Search Tree**: Given the root of a binary tree, *determine if it is a valid binary search tree (BST)*.

A **valid BST** is defined as follows:

- The left subtree of a node contains only nodes with keys **less than** the node's key.

- The right subtree of a node contains only nodes with keys **greater than** the node's key.
- Both the left and right subtrees must also be binary search trees.
- 

- **Input:** root = [2,1,3]
- **Output:** true
- 

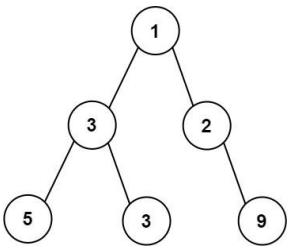
- **Input:** root = [5,1,4,null,null,3,6]
- **Output:** false
- **Explanation:** The root node's value is 5 but its right child's value is 4.

```
bool check(TreeNode* root, long long int lb, long long int up){
    if (root==NULL)return true;
    if (root->val <=lb || root->val >=up)return false;
    return (check(root->left,lb,root->val) && check(root->right,root->val,up));
  }
  bool isValidBST(TreeNode* root) {
    long long int lb= LLONG_MIN; long long int up= LLONG_MAX;
    return check(root,lb,up);
  }
```

- Time complexity: $O(n)$
- Space complexity: $O(1)$

---

31) 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.



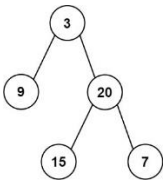
Input: root = [1,3,2,5,3,null,9]
Output: 4
Explanation: The maximum width exists in the third level with length 4 (5,3,null,9).

```
int widthOfBinaryTree(TreeNode* root) {

    queue<pair<TreeNode*, int>>q;
    q.push({root, 0});
    int maxWidth = 1;

    while(!q.empty()){
      int start = q.front().second;
      int end = q.back().second;

      maxWidth = max(maxWidth, end - start + 1);

      int count = q.size();

      while(count--){
        int idx = q.front().second - end;
        TreeNode* node = q.front().first;
        q.pop();

        if(node->left) {
          q.push({node->left, 2 * idx + 1});
        }

        if(node->right) {
          q.push({node->right, 2 * idx + 2});
        }
      }
    }

    return maxWidth;
  }
```

- Time complexity: $O(N)$
- Space complexity: $O(N)$

---

32) **Construct Binary Tree from Preorder and Inorder Traversal**:
Given two integer arrays preorder and inorder where preorder is the preorder traversal of a binary tree and inorder is the inorder traversal of the same tree, construct and return *the binary tree*.



Input: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]
Output: [3,9,20,null,null,15,7]

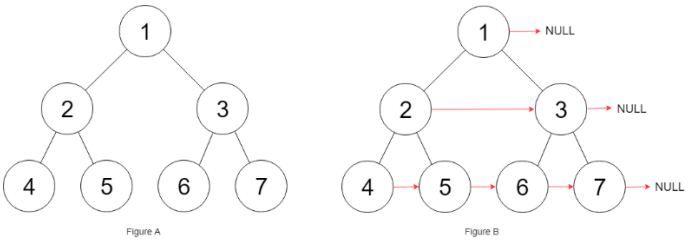
```
class Solution {
public:
  TreeNode* buildTree(vector<int>& preOrder, vector<int>& inOrder) {
    int pIndex = 0;
    unordered_map<int, int> mapping;
    for (int i = 0; i < inOrder.size(); i++)
      mapping[inOrder[i]] = i;
    return BuildTreeHelper(preOrder, inOrder, 0, preOrder.size() - 1, mapping, pIndex);
  }
  static TreeNode *BuildTreeHelper(vector<int> &preOrder, vector<int> &inOrder, int left, int
right, unordered_map<int, int> &mapping, int &pIndex)
  {
    if (left > right)
      return nullptr;
    int curr = preOrder[pIndex++];
    TreeNode *root = new TreeNode(curr);
    if (left == right)
      return root;
    int inIndex = mapping[curr];
    root->left = BuildTreeHelper(preOrder, inOrder, left, inIndex - 1, mapping, pIndex);
    root->right = BuildTreeHelper(preOrder, inOrder, inIndex + 1, right, mapping, pIndex);
    return root;
  }
```

---

33) **Populating Next Right Pointers in Each Node**: You are given a **perfect binary tree** where all leaves are on the same level, and every parent has two children. The binary tree has the following definition:

```
struct Node {
 int val;
 Node *left;
 Node *right;
 Node *next;
}
```

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.
Initially, all next pointers are set to NULL.

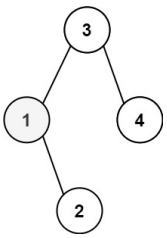


Input: root = [1,2,3,4,5,6,7]
Output: [1,#,2,3,#,4,5,6,7,#]
Explanation: Given the above perfect binary tree (Figure A), your function should populate each next pointer to point to its next right node, just like in Figure B. The serialized output is in level order as connected by the next pointers, with '#' signifying the end of each level.
Example 2:
Input: root = []
Output: []

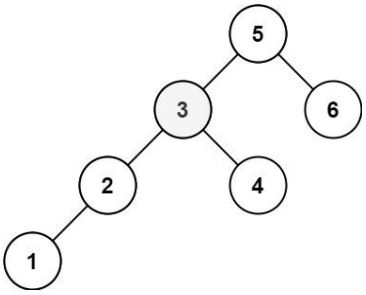
```
Node* connect(Node* root) {
  queue<Node*> q;
  if (root) q.push(root);
  while (q.size()) {
    int len = q.size();
    Node* curr;
    while (len--) {
      curr = q.front(), q.pop();
      curr->next = len ? q.front():NULL;
      if (curr->left) q.push(curr->left);
      if (curr->right) q.push(curr->right);
    }
  }
  return root;
}
```

---

35) **Kth Smallest Element in a BST**: Given the root of a binary search tree, and an integer k, return *the* kth *smallest value (**1-indexed**) of all the values of the nodes in the tree*.



Input: root = [3,1,4,null,2], k = 1
Output: 1



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

```
void InOrder(TreeNode* root,vector<int>&ans,int k){
    if(root==NULL) return;
    InOrder(root->left,ans,k);
    ans.push_back(root->val);
    InOrder(root->right,ans,k);
  }
  int kthSmallest(TreeNode* root, int k) {
    vector<int>ans;
    InOrder(root,ans,k);
    // 1 based indexing
    return ans[k-1];
  }
```

---

36) **K'th Largest element in BST**

Given a binary search tree, task is to find Kth largest element in the binary search tree.

Input : k = 3

```
     Root of following BST
       10
      /   \
     4     20
    /     /  \
   2    15    40
```

Output : 15

```
Node* KthLargestUsingMorrisTraversal(Node* root, int k)
{
  Node* curr = root;
  Node* Klargest = NULL;

  // count variable to keep count of visited Nodes
  int count = 0;

  while (curr != NULL) {
    // if right child is NULL
    if (curr->right == NULL) {
```

```
        // first increment count and check if count = k
        if (++count == k)
            Klargest = curr;

        // otherwise move to the left child
        curr = curr->left;
      }

      else {

        // find inorder successor of current Node
        Node* succ = curr->right;

        while (succ->left != NULL && succ->left != curr)
            succ = succ->left;

        if (succ->left == NULL) {

            // set left child of successor to the
            // current Node
            succ->left = curr;

            // move current to its right
            curr = curr->right;
        }

        // restoring the tree back to original binary
        //  search tree removing threaded links
        else {

            succ->left = NULL;

            if (++count == k)
                Klargest = curr;

            // move current to its left child
            curr = curr->left;
        }
      }
    }

    return Klargest;
}
```

----------Hard----------

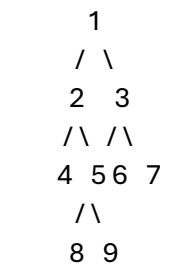
-32) **Boundary Traversal of a Binary Tree:**

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.

Note: If the root doesn't have a left subtree or right subtree, then the root itself is the left or right boundary.

Input:



Output: 1 2 4 8 9 6 7 3

[Vertical Order Traversal of a Binary Tree]:

Count BST nodes that lie in a given range

Given a Binary Search Tree (BST) and a range **l-h(inclusive)**, count the number of nodes in the BST that lie in the given range.

- The values smaller than root go to the left side
- The values greater and equal to the root go to the right side

- **Input:**
- 10

- / \
- 5 50
- / / \
- 1 40 100
- l = 5, h = 45
- **Output:** 3
- **Explanation:** 5 10 40 are the node in the
- range
- **Example 2:**
- **Input:**
- 5
- / \
- 4 6
- / \
- 3 7
- l = 2, h = 8
- **Output:** 5
- **Explanation:** All the nodes are in the
- given range.