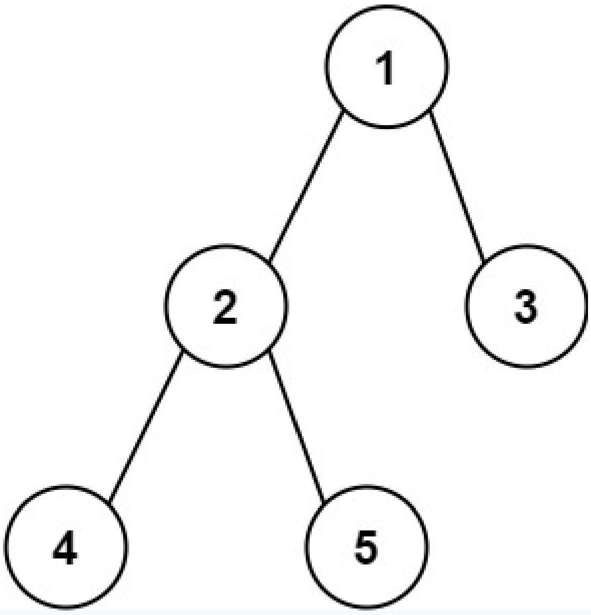
SNo.	Problem Statement
1.	**Easy Level: Diameter of Binary Tree.**

Code:

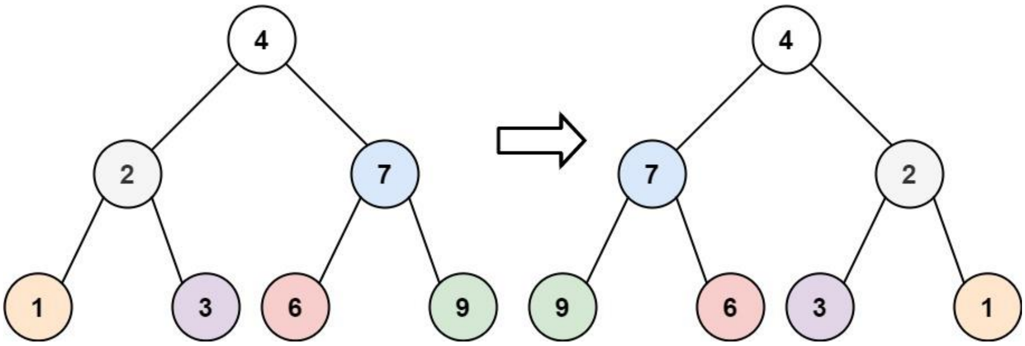


```
Input: root = [1,2,3,4,5]

Output: 3

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

```
int diameterOfBinaryTree(TreeNode* root) {

        int diameter=0;
        height(root,diameter);
        return diameter;
    }

    int height(TreeNode* node,int &diameter)
    {
        if(node==NULL)
            return 0;
        int lh=height(node->left,diameter);
        int rh=height(node->right,diameter);
        diameter=max(diameter,lh+rh);
        return 1+max(lh,rh);
    }
```

2. **Easy Level: Invert Binary Tree.**

Code:

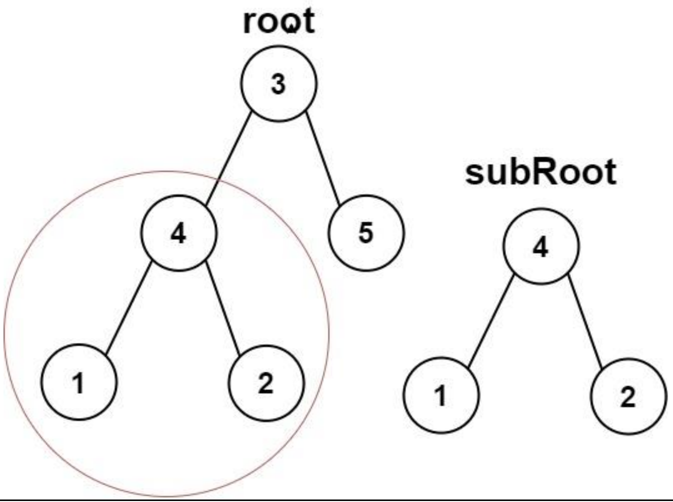


```
Input: root = [4,2,7,1,3,6,9]

Output: [4,7,2,9,6,3,1]
```

```
TreeNode* invertTree(TreeNode* root)
 {
    if(root==NULL)
    return 0;
    TreeNode* left=invertTree(root->left);
    TreeNode* right=invertTree(root->right);
    root->left=right;
    root->right=left;
    return root;
 }
```

3. **Easy Level: Subtree of Another Tree.**

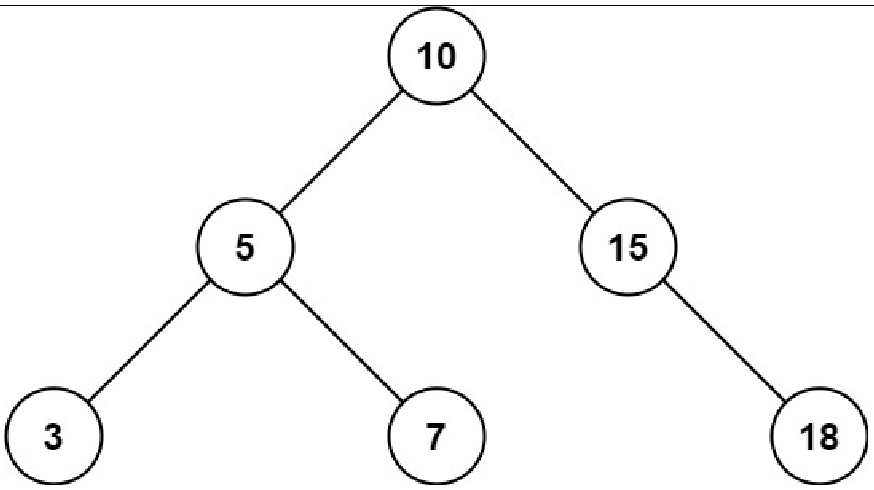
Code:



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

Output: true
```

```
bool dfs(TreeNode* root1,TreeNode* root2)
    {
        if(!root1 and !root2)
            return true;
        if(!root1 || !root2)
            return false;
        if(root1->val!=root2->val)
            return false;
        return dfs(root1->left,root2->left) and dfs(root1->right,root2-
>right);

    }
    bool isSubtree(TreeNode* root, TreeNode* subRoot) {
       if(!root)
           return false;
        if(root->val==subRoot->val)
        {
            if(dfs(root,subRoot))
                return true;
        }
        return isSubtree(root->left,subRoot)||isSubtree(root-
>right,subRoot);
    }
```

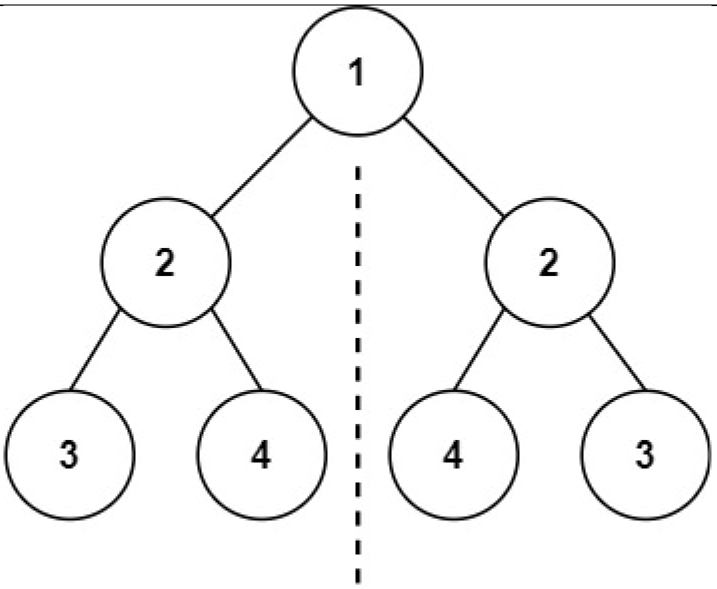
4. **Easy Level: Range Sum of BST.**
Code:



```
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.
```

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

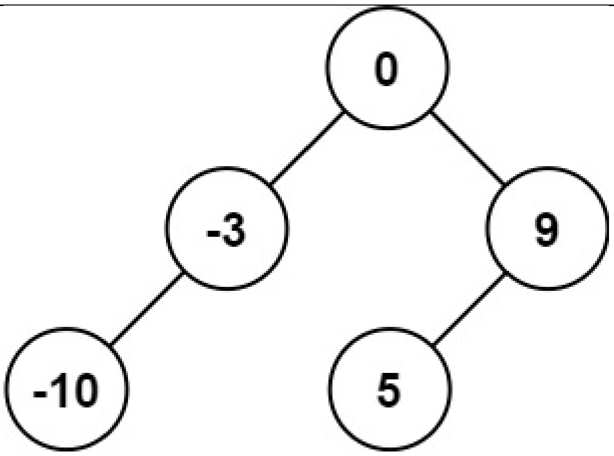
5. **Easy Level: Symmetric Tree.**
Code:



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

Output: true
```

```
bool dfs(TreeNode* root1, TreeNode* root2)
{
   if(root1==NULL and root2==NULL)
   {
      return true;
   }
   if(root1==NULL or root2==NULL)
   {
      return false;
   }
   return ((root1->val==root2->val) and dfs(root1->left,root2->right)
and dfs(root1->right,root2->left));
}
bool isSymmetric(TreeNode* root) {

   return dfs(root->left,root->right);
}
```

6. **Easy Level: Convert Sorted Array to Binary Search Tree.**
Code:

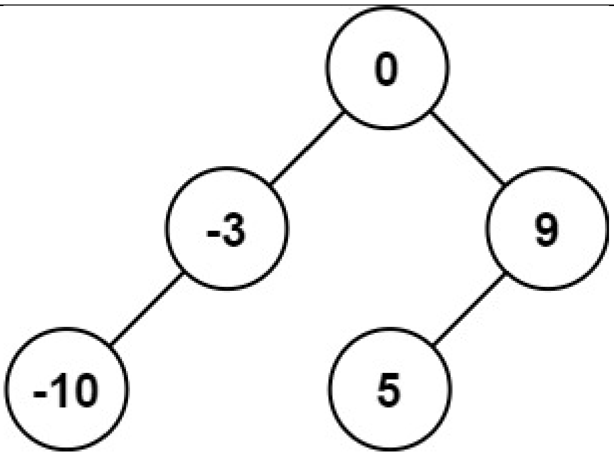


```
Input: nums = [-10,-3,0,5,9]

Output: [0,-3,9,-10,null,5]

Explanation: [0,-10,5,null,-3,null,9] is also accepted:
```

```
TreeNode* binaryST(int s,int e,vector<int>nums)
    {
         if(s>e)
             return NULL;
        if(s==e)
        {
            return new TreeNode(nums[e]);
        }
         int mid=(e+s)/2;
        TreeNode* root=new TreeNode(nums[mid]);
        root->left=binaryST(s,mid-1,nums);
        root->right=binaryST(mid+1,e,nums);
        return root;
    }
    TreeNode* sortedArrayToBST(vector<int>& nums) {

       return binaryST(0,nums.size()-1,nums);
    }
```

7. **Easy Level: Merge Two Binary Trees.**

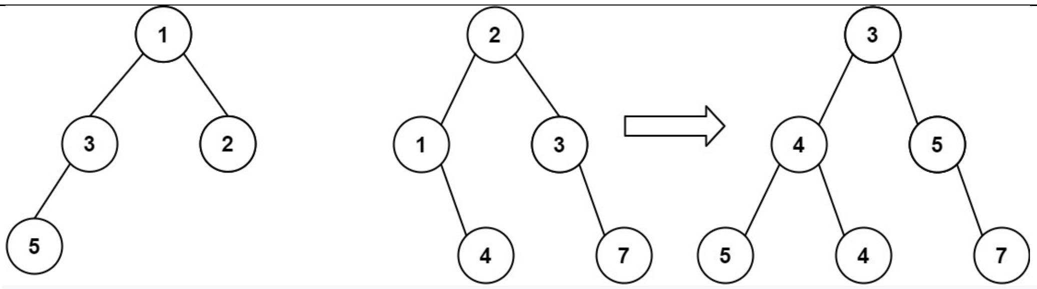
Code:



```
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* mergeTrees(TreeNode* root1, TreeNode* root2) {

      if(root1==NULL )
         return root2;
      if(root2==NULL)
         return root1;
      root1->val=root1->val+root2->val;
      root1->left=mergeTrees(root1->left,root2->left);
      root1->right=mergeTrees(root1->right,root2->right);
      return root1;



   }
```

8. **Easy Level: Maximum Depth of Binary Tree.**
Code:



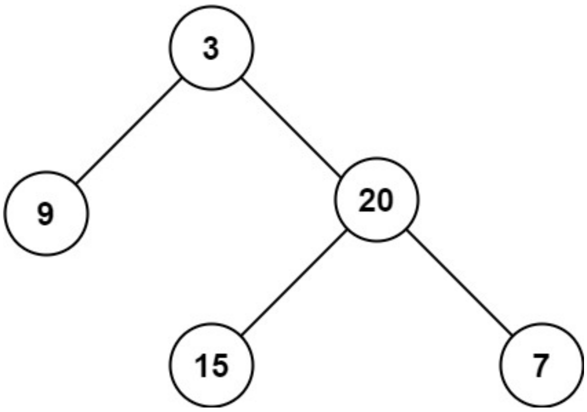
```
Input: root1 = [1,3,2,5], root2 = [2,1,3,null,4,null,7]

Output: [3,4,5,5,4,null,7]
```

```
int maxDepth(TreeNode* root) {

        if(root==NULL)
            return 0;
        return max(maxDepth(root->left),maxDepth(root->right))+1;
    }
```

9.

Easy Level: Same Tree.

Code:



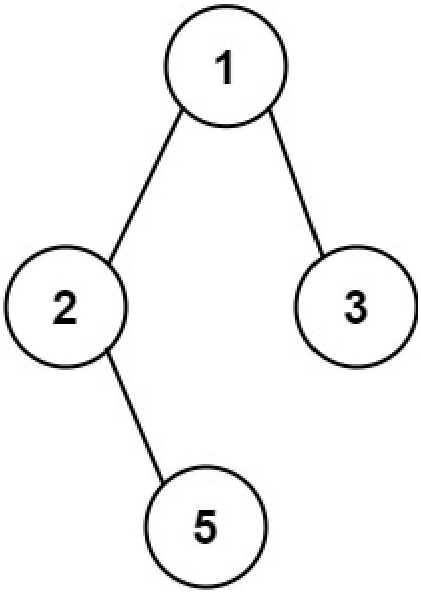
```
Input: root = [3,9,20,null,null,15,7]

Output: 3
```

```
bool isSameTree(TreeNode* p, TreeNode* q) {

        if(p==NULL && q==NULL)
            return true;
        if(q==NULL || p==NULL)
```

```
             return false;
         if(p->val!=q->val)
             return false;
         return isSameTree(p->right,q->right) and isSameTree(p->left,q->left);


     }
```

10. **Easy Level: Lowest Common Ancestor of a Binary Search Tree.**

Code:

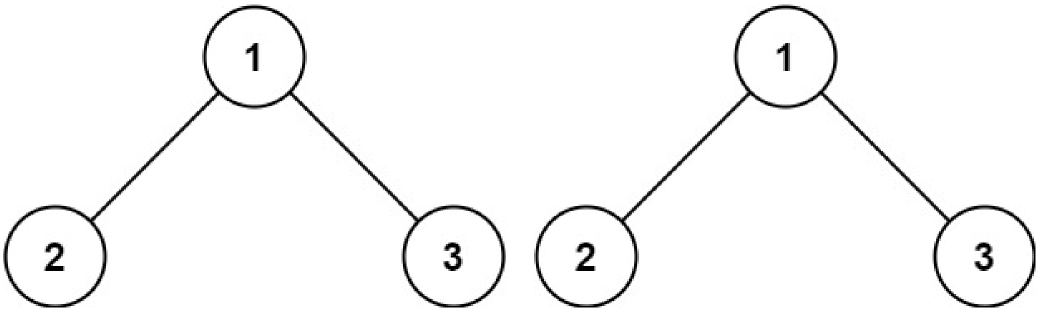


```
Input: root = [1,2,3,null,5]

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

```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {

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

```
        if(left==NULL)
            return right;
        if(right==NULL)
            return left;
        else
            return root;
    }
```

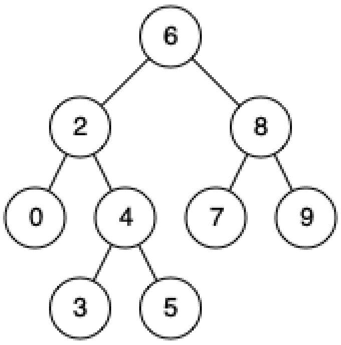
11. **Easy Level: Path Sum.**
Code:



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

Output: true
```

```
bool hasPathSum(TreeNode* root, int targetSum) {
        if(root==NULL)
            return false;
        if(root->left==NULL and root->right==NULL)
        {
            return (targetSum - root->val==0);
        }
        return (hasPathSum(root->right, targetSum - root-
>val)||hasPathSum(root->left, targetSum - root->val));
    }
```

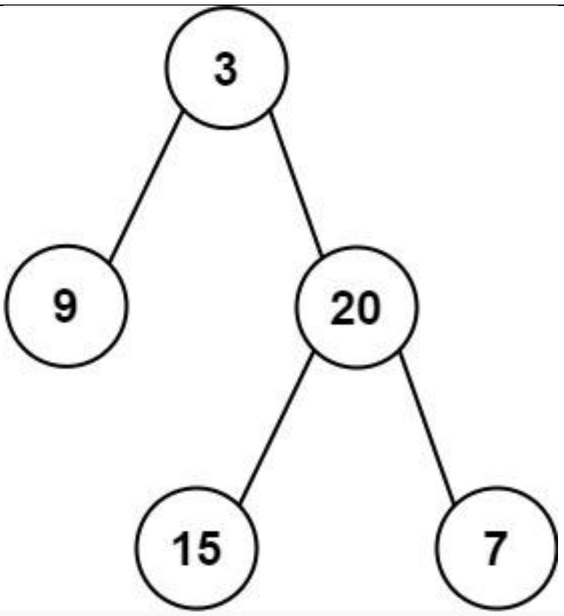
12. **Easy Level: Minimum Absolute Difference in BST.**
Code:



```
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.
```

```
int diff = INT_MAX;
     TreeNode* prev = NULL;
    void dfs(TreeNode* root)
    {
        if(root==NULL)
            return;
        dfs(root->left);
        if(prev)
            diff = min(diff, abs(prev->val-root->val));
            prev = root;
        dfs(root->right);

    }
    int getMinimumDifference(TreeNode* root) {
        if(root==NULL)
            return 0;
        dfs(root);
        return diff;
    }
```

13. **Easy Level: Sum of Left Leaves.**
Code:



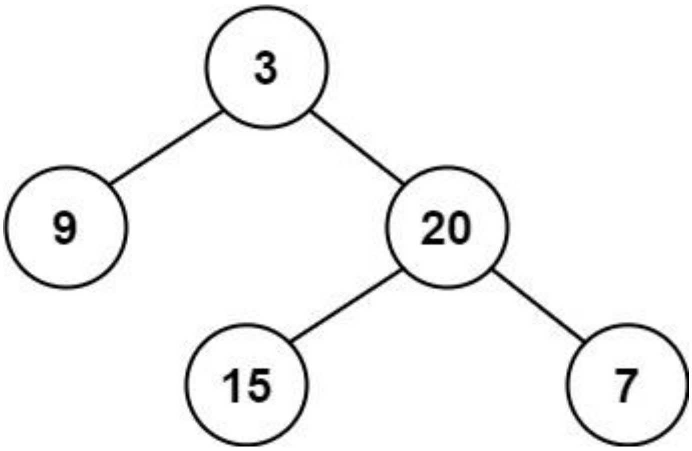
```
Input: root = [3,9,20,null,null,15,7]

Output: 24

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

```
int sumOfLeftLeaves(TreeNode *root)
{
    int sum=0;
    std::queue<TreeNode* >q ;
    while(!q.empty())
    {
        q.push(root);
        TreeNode* node=q.front();
        TreeNode* p;
        if(node->left)
        {
            p=node->left;
            if(p->left==NULL and p->right==NULL)
            {
                sum+=node->left->val;
            }
        }
        q.pop();
        if(node->left)
```

```
          q.push(node->left);
          if(node->right)
          q.push(node->right);
      }
      return sum;
  }
```

14. **Easy Level: Balanced Binary Tree.**

Code:



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

Output: true
```

```
bool isBalanced(TreeNode* root) {

          return height(root)!=-1;

      }

      int height(TreeNode* root)
      {
          if(root==NULL)
              return 0;
          int leftHeight=height(root->left);
          if(leftHeight==-1)
              return -1;
          int rightHeight=height(root->right);
          if(rightHeight==-1)
              return-1;
```

```
        if(abs(leftHeight - rightHeight)>1)
            return -1;
        return max(leftHeight,rightHeight)+1;
    }
```

15. **Easy Level: Predecessor and Successor.**

Code:

Input:

```
2
6
50 30 L 30 20 L 30 40 R 50 70 R 70 60 L 70 80 R
65
6
50 30 L 30 20 L 30 40 R 50 70 R 70 60 L 70 80 R
100
```

Output:

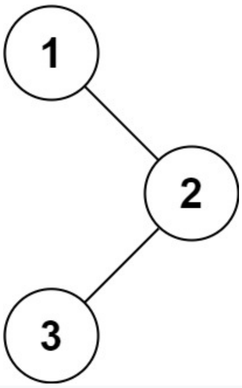
```
60 70
80 -1
```

```
void inorder_successor(Node* root, Node* &succ, int key)
{
    while(root!=NULL)
    {
        if(root->key<=key)
        {
            root=root->right;
        }
        else if(root->key>key)
        {
            succ=root;
            root=root->left;
        }
    }
}

void inorder_predecessor(Node* root, Node* &pred, int key)
{
    while(root!=NULL)
    {
        if(root->key>=key)
        {
           root=root->left;
        }
        else if(root->key<key)
        {
           pred=root;
           root=root->right;
        }
    }
}

void findPreSuc(Node* root, Node*& pre, Node*& suc, int key)
{

// Your code goes here

    inorder_successor(root,suc,key);
    inorder_predecessor(root,pre,key);

}
```

16. **Easy Level: Binary Tree Inorder Traversal.**

Code:

```
Input: root = [1,null,2,3]

Output: [1,3,2]
```



```
Input: root = [1,null,2,3]
```

```
Output: [1,3,2]
```

```
void tree(TreeNode* root,vector<int>&v)
    {
        if(root==NULL)
            return;
        tree(root->left,v);
        v.push_back(root->val);
        tree(root->right,v);

    }
    vector<int> inorderTraversal(TreeNode* root) {

        vector<int>v;
        tree(root,v);
        return v;
    }
```

17. **Easy Level: Check whether BST contains Dead End.**

Code:

```
int c=0;
bool fun(Node* root,int lb,int ub)
{
    if(root==0&& abs(lb-ub)==1)
    return 1;
    if(root==0)
    return 0;

    bool l=fun(root->left,lb,root->data);
    bool r=fun(root->right,root->data,ub);

    if(l&&r)
    c=1;
    return 0;
}
bool isDeadEnd(Node *root)
{
    //Your code here
    c=0;
    fun(root,0,INT_MAX);
    return c;
}
```