TREE BEST QUESTIONS

1.) INORDER TRAVERSAL

CODE

```
class Solution {
public:
    void func(TreeNode* root, vector<int> &ans){
        if(root==NULL)
        return;
        func(root->left, ans);
        ans.push_back(root->val);
        func(root->right, ans);
    }

    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> ans;
        func(root, ans);
        return ans;
    }
};
```

2.) PREORDER TRAVERSAL

CODE

```
class Solution {
public:
    void func(TreeNode* root, vector<int> &ans){
        if(root==NULL)
        return;
        ans.push_back(root->val);
        func(root->left, ans);
        func(root->right, ans);
    }

    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> ans;
        func(root, ans);
        return ans;
    }
};
```

TREE BEST QUESTIONS

3.) POSTORDER TRAVERSAL

CODE

```
class Solution {
public:
    void func(TreeNode* root, vector<int> &ans){
        if(root==NULL)
        return;
        func(root->left, ans);
        func(root->right, ans);
        ans.push_back(root->val);
    }

    vector<int> postorderTraversal(TreeNode* root) {
        vector<int> ans;
        func(root, ans);
        return ans;
    }
};
```

4.) N-ARY PREORDER TRAVERSAL

CODE

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

    vector<int> preorder(Node* root) {
        vector<int> ans;
        func(root, ans);
        return ans;
    }
};
```

5.) N-ARY POSTORDER TRAVERSAL

CODE

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

6.) LEVEL ORDER TRAVERSAL

CODE

```
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> ans;

        if(root==NULL)
        return ans;

        queue<TreeNode*> q;
        q.push(root);

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

7.) N-ARY LEVEL ORDER TRAVERSAL

CODE 01

```
class Solution {
public:
    vector<vector<int>> levelOrder(Node* root) {
        vector<vector<int>> ans;
        if(root==NULL)
        return ans;

        queue<Node*> q;
        q.push(root);

        while(!q.empty()){
            int size=q.size();
            vector<int> level;

            for(int i=0;i<size;i++){
                Node* node=q.front();
                q.pop();
                level.push_back(node->val);
                for(auto it:node->children){
                    q.push(it);
                }
            }
            ans.push_back(level);
        }
        return ans;
    }
};
```

CODE 02

```
class Solution {
public:
    vector<vector<int>> levelOrder(Node* root) {
        vector<vector<int>> ans;
        if(root==NULL)
        return ans;

        queue<Node*> q;
        q.push(root);

        while(!q.empty()){
            int size=q.size();
            vector<int> level;
            for(int i=0;i<size;i++){
                Node* node=q.front();
                q.pop();
                for(auto it:node->children){
                    q.push(it);
                }
                level.push_back(node->val);
            }
            ans.push_back(level);
        }
        return ans;
    }
};
```

8.) MAXIMUM DEPTH OF BINARY TREE

CODE

```
class Solution {
public:
    int func(TreeNode* root){
        if(root==NULL)
        return 0;
        int lh=func(root->left);
        int rh=func(root->right);
        return 1+max(lh, rh);
    }

    int maxDepth(TreeNode* root) {
        return func(root);
    }
};
```

9.) MAXIMUM DEPTH OF N-ARY TREE

CODE

```
class Solution {
public:
    int func(Node* root){
        if(root==NULL)
        return 0;

        if(root->children.size()==0)
        return 1;

        vector<int> arr;
        for(int i=0;i<root->children.size();i++){
            arr.push_back(func(root->children[i]));
        }

        int depth=1+*max_element(arr.begin(), arr.end());
        return depth;
    }
    int maxDepth(Node* root) {
        return func(root);
    }
};
```

10.) MINIMUM DEPTH OF BINARY TREE

CODE

```
class Solution {
public:
    int func(TreeNode* root){
        if(root==NULL)
        return 0;

        if(root->left==NULL && root->right!=NULL)
        return 1+func(root->right);

        else if(root->right==NULL && root->left!=NULL)
        return 1+func(root->left);

        int lh=func(root->left);
        int rh=func(root->right);
        return 1+min(lh, rh);
    }
    int minDepth(TreeNode* root) {
        return func(root);
    }
};
```

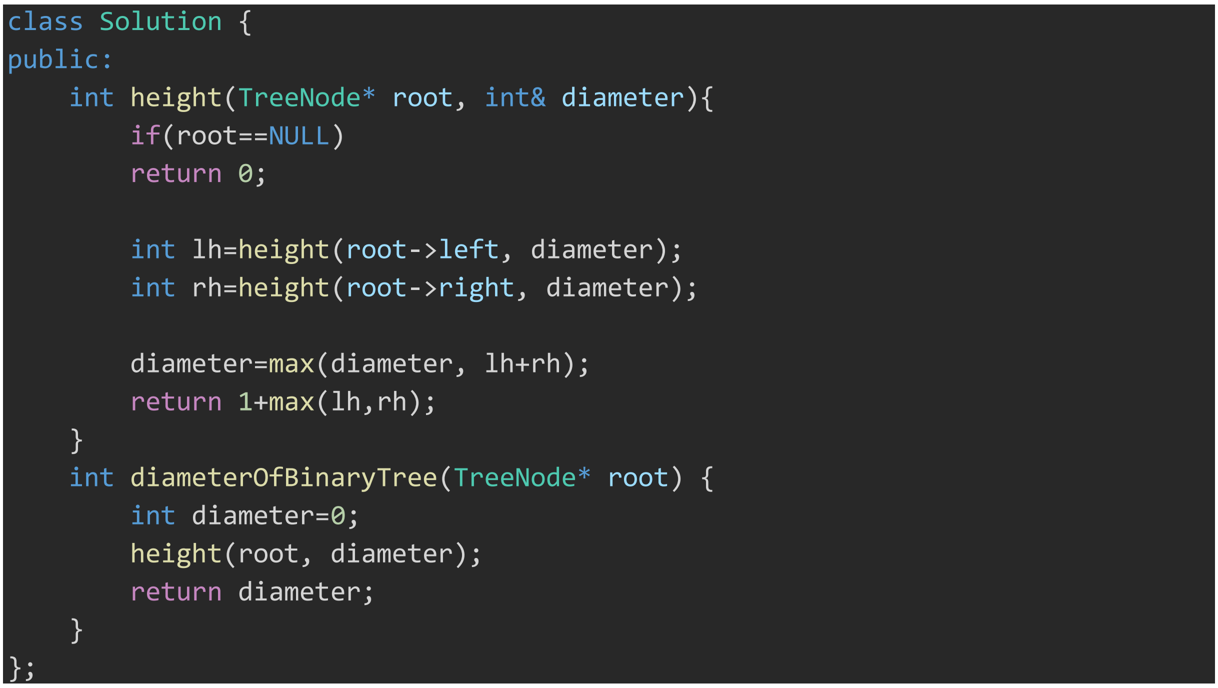
11.) CHECK FOR BALANCED BINARY TREE

CODE

```
class Solution {
public:
    int height(TreeNode* root){
        if(root==NULL)
        return 0;

        int lh=height(root->left);
        if(lh==-1)
        return -1;

        int rh=height(root->right);
        if(rh==-1)
        return -1;

        if(abs(lh-rh)>1)
        return -1;

        return 1+max(lh, rh);
    }
    bool isBalanced(TreeNode* root) {
        if(height(root)==-1)
        return false;
        else
        return true;
    }
};
```

12.) DIAMETER OF BINARY TREE

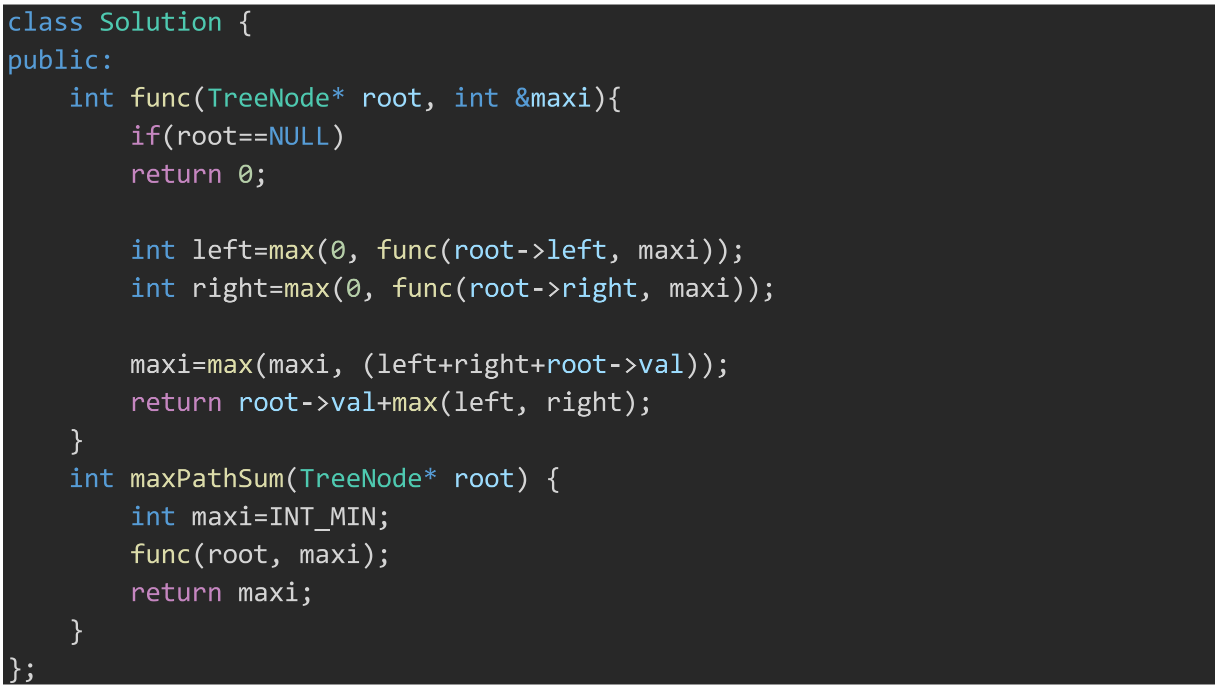
CODE



```
class Solution {
public:
    int height(TreeNode* root, int& diameter){
        if(root==NULL)
        return 0;

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

        diameter=max(diameter, lh+rh);
        return 1+max(lh,rh);
    }
    int diameterOfBinaryTree(TreeNode* root) {
        int diameter=0;
        height(root, diameter);
        return diameter;
    }
};
```

13.) MAXIMUM PATH SUM

CODE



```
class Solution {
public:
    int func(TreeNode* root, int &maxi){
        if(root==NULL)
        return 0;

        int left=max(0, func(root->left, maxi));
        int right=max(0, func(root->right, maxi));

        maxi=max(maxi, (left+right+root->val));
        return root->val+max(left, right);
    }
    int maxPathSum(TreeNode* root) {
        int maxi=INT_MIN;
        func(root, maxi);
        return maxi;
    }
};
```

14.) SAME TREE

CODE

```
class Solution {
public:
    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);
    }
};
```

15.) ZIG ZAG TRAVERSAL OR SPIRAL TRAVERSAL

CODE

```
class Solution {
public:
    vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
        vector<vector<int>> ans;
        if(root==NULL)
        return ans;

        queue<TreeNode*> q;
        q.push(root);

        bool flag=true;

        while(!q.empty()){
            int size=q.size();
            vector<int> row(size);

            for(int i=0;i<size;i++){
                TreeNode* node=q.front();
                q.pop();

                int index=0;
                if(flag==true)
                index=i;
                else
                index=size-1-i;

                row[index]=node->val;

                if(node->left!=NULL)
                q.push(node->left);
                if(node->right!=NULL)
                q.push(node->right);
            }

            flag=!flag;
            ans.push_back(row);
        }
        return ans;
    }
};
```

16.) BOUNDARY TRAVERSAL

CODE

```
class Solution {
public:
    bool isLeaf(Node* root) {
        return !root->left && !root->right;
    }

    void addLeftBoundary(Node* root, vector<int>& res) {
        Node* curr = root->left;

        while (curr) {
            if (!isLeaf(curr)) {
                res.push_back(curr->data);
            }
            if (curr->left) {
                curr = curr->left;
            } else {
                curr = curr->right;
            }
        }
    }

    void addRightBoundary(Node* root, vector<int>& res) {
        Node* curr = root->right;
        vector<int> temp;

        while (curr) {
            if (!isLeaf(curr)) {
                temp.push_back(curr->data);
            }
            if (curr->right) {
                curr = curr->right;
            } else {
                curr = curr->left;
            }
        }

        for (int i = temp.size() - 1; i >= 0; --i) {
            res.push_back(temp[i]);
        }
    }

    void addLeaves(Node* root, vector<int>& res) {
        if (isLeaf(root)) {
            res.push_back(root->data);
            return;
        }
        if (root->left) {
            addLeaves(root->left, res);
        }
        if (root->right) {
            addLeaves(root->right, res);
        }
    }

    vector<int> printBoundary(Node* root) {

        vector<int> res;
        if (!root) {
            return res;
        }

        if (!isLeaf(root)) {
            res.push_back(root->data);
        }

        addLeftBoundary(root, res);
        addLeaves(root, res);
        addRightBoundary(root, res);
        return res;
    }
};
```

17.) VERTICAL ORDER TRAVERSAL

CODE

```
class Solution {
public:
    vector<vector<int>> verticalTraversal(TreeNode* root) {
        map<int, map<int, multiset<int>>> nodes;

        queue<pair<TreeNode*, pair<int, int>>> todo;
        todo.push({root, {0, 0}});

        while(!todo.empty()){
            auto p=todo.front();
            todo.pop();

            TreeNode* temp=p.first;
            int x=p.second.first;
            int y=p.second.second;

            nodes[x][y].insert(temp->val);

            if(temp->left){
                todo.push({temp->left, {x-1, y+1}});
            }

            if(temp->right){
                todo.push({temp->right, {x+1, y+1}});
            }
        }

        vector<vector<int>> ans;
        for(auto p:nodes){
            vector<int> col;
            for(auto q:p.second){
                col.insert(col.end(), q.second.begin(), q.second.end());
            }
            ans.push_back(col);
        }
        return ans;
    }
};
```

18.) RIGHT SIDE VIEW OF BINARY TREE

CODE

```
class Solution {
public:
    void func(TreeNode* root, int level, vector<int> &ans){
        if(root==NULL)
        return;

        if(ans.size()==level){
            ans.push_back(root->val);
        }
        func(root->right, level+1, ans);
        func(root->left, level+1, ans);
    }
    vector<int> rightSideView(TreeNode* root) {
        vector<int> ans;
        func(root, 0, ans);
        return ans;
    }
};
```

19.) LEFT SIDE VIEW OF BINARY TREE

CODE

```
class Solution {
public:
    void func(TreeNode* root, int level, vector<int> &ans){
        if(root==NULL)
        return;

        if(ans.size()==level){
            ans.push_back(root->val);
        }
        func(root->left, level+1, ans);
        func(root->right, level+1, ans);
    }
    vector<int> leftSideView(TreeNode* root) {
        vector<int> ans;
        func(root, 0, ans);
        return ans;
    }
};
```

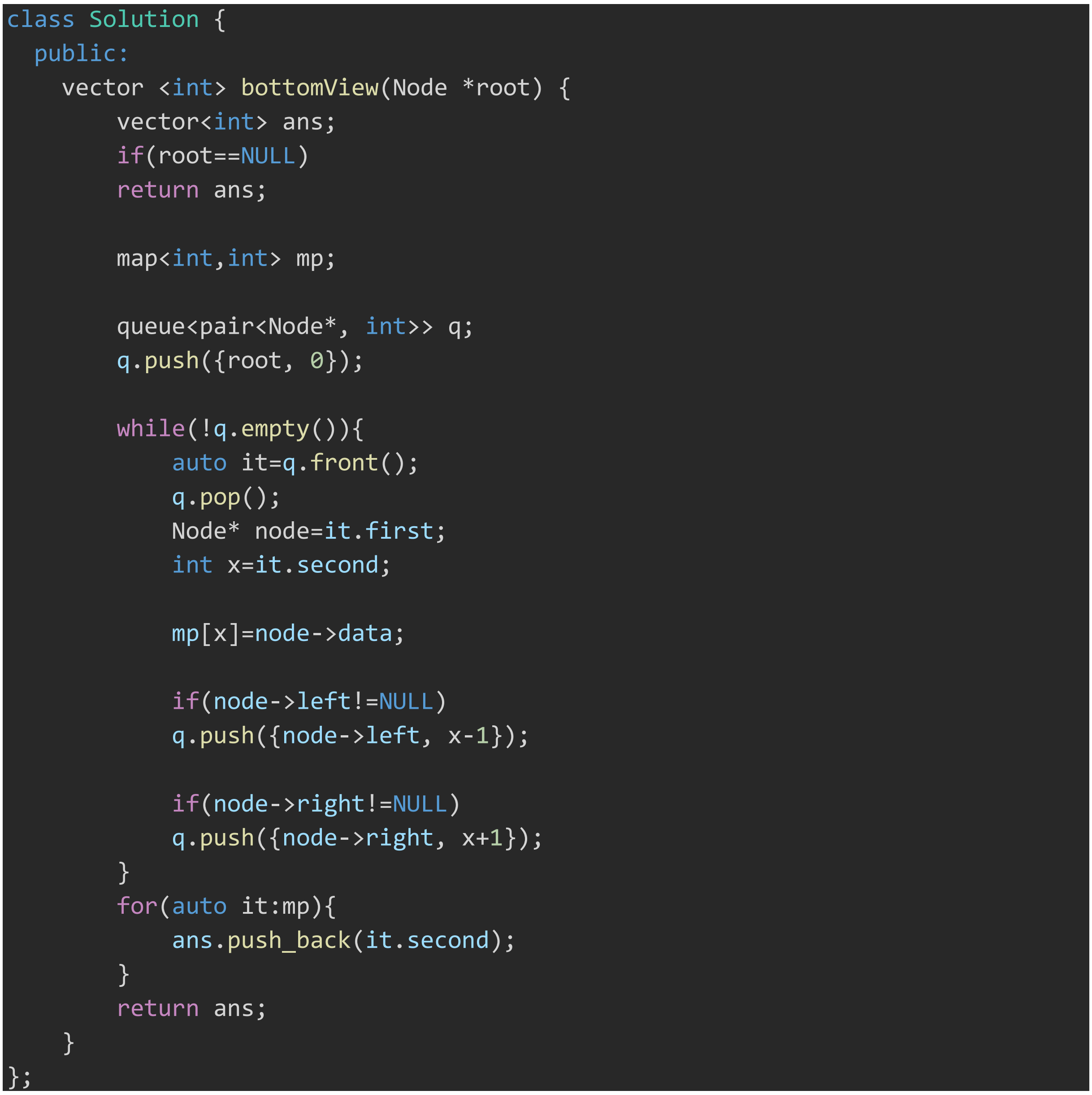
20.) TOP VIEW OF BINARY TREE

CODE

```
class Solution
{
    public:
    //Function to return a list of nodes visible from the top view
    //from left to right in Binary Tree.
    vector<int> topView(Node *root)
    {
        vector<int> ans;
        if(root==NULL)
        return ans;

        map<int, int> mp;

        queue<pair<Node*, int>> q;
        q.push({root, 0});

        while(!q.empty()){
            auto it=q.front();
            q.pop();
            Node* node=it.first;
            int x=it.second;

            if(mp.find(x)==mp.end()){
                mp[x]=node->data;
            }

            if(node->left!=NULL){
                q.push({node->left, x-1});
            }
            if(node->right!=NULL){
                q.push({node->right, x+1});
            }
        }
        for(auto it:mp){
            ans.push_back(it.second);
        }
        return ans;
    }

};
```

21.) BOTTOM VIEW OF BINARY TREE

CODE



```cpp
class Solution {
  public:
    vector <int> bottomView(Node *root) {
        vector<int> ans;
        if(root==NULL)
        return ans;

        map<int,int> mp;

        queue<pair<Node*, int>> q;
        q.push({root, 0});

        while(!q.empty()){
            auto it=q.front();
            q.pop();
            Node* node=it.first;
            int x=it.second;

            mp[x]=node->data;

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

            if(node->right!=NULL)
            q.push({node->right, x+1});
        }
        for(auto it:mp){
            ans.push_back(it.second);
        }
        return ans;
    }
};
```

22.) SYMMETRIC TREE

CODE

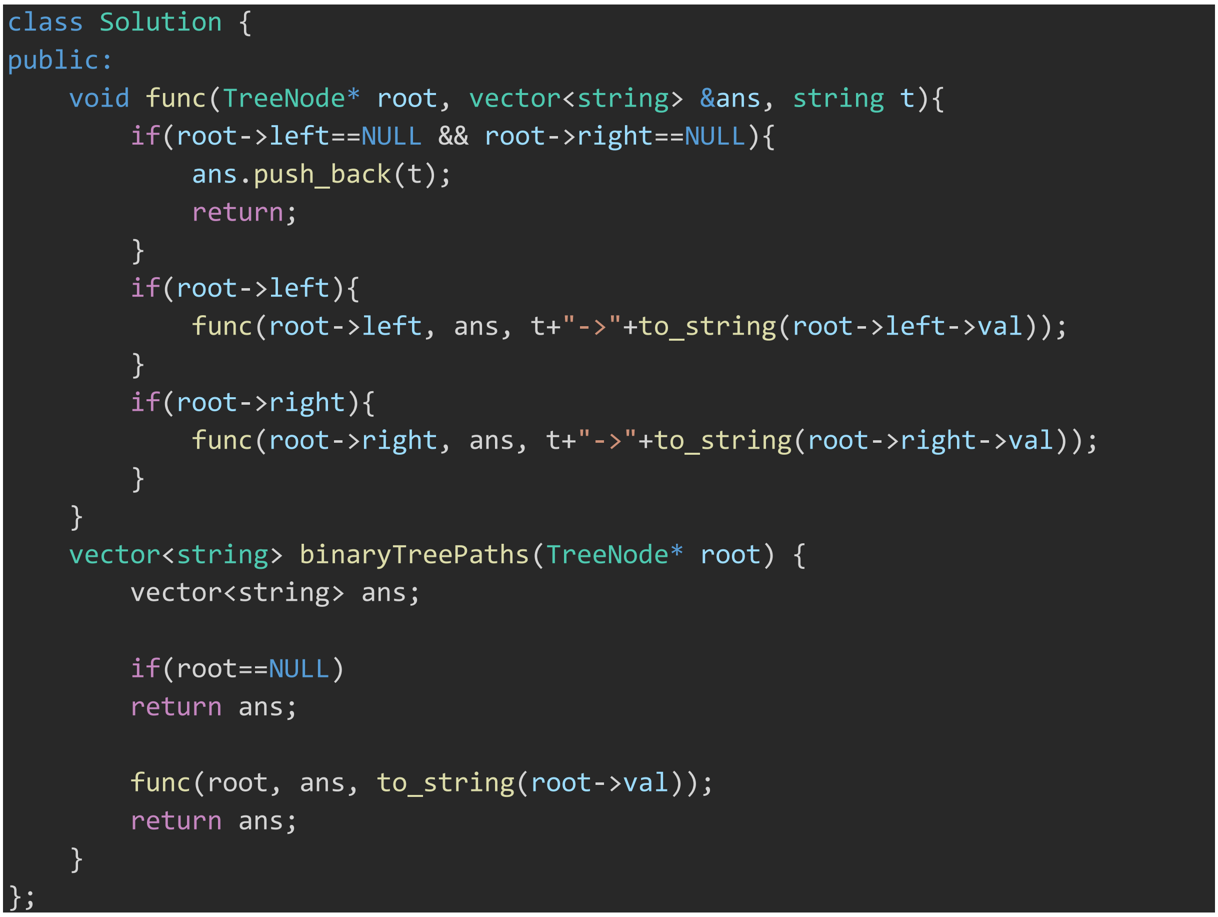
```
class Solution {
public:
    bool func(TreeNode* l, TreeNode* r){
        if(l==NULL && r==NULL)
        return true;

        if(l==NULL && r!=NULL || l!=NULL && r==NULL)
        return false;

        if(l->val!=r->val)
        return false;

        return func(l->left, r->right) & func(l->right, r->left);
    }
    bool isSymmetric(TreeNode* root) {
        return func(root->left, root->right);
    }
};
```

23.) ROOT TO LEAF ALL PATHS

CODE 01

```
class Solution {
public:
    void func(Node* root, vector<vector<int>> &ans, vector<int> &temp) {
        if (root == NULL)
            return;
        if (root->left == NULL && root->right == NULL) {
            temp.push_back(root->data);
            ans.push_back(temp);
            temp.pop_back();
            return;
        }
        temp.push_back(root->data);
        func(root->left, ans, temp);
        func(root->right, ans, temp);
        temp.pop_back();
    }

    vector<vector<int>> Paths(Node* root) {
        vector<vector<int>> ans;
        vector<int> temp;
        func(root, ans, temp);
        return ans;
    }
};
```

CODE 02



```
class Solution {
public:
    void func(TreeNode* root, vector<string> &ans, string t){
        if(root->left==NULL && root->right==NULL){
            ans.push_back(t);
            return;
        }
        if(root->left){
            func(root->left, ans, t+"->"+to_string(root->left->val));
        }
        if(root->right){
            func(root->right, ans, t+"->"+to_string(root->right->val));
        }
    }
    vector<string> binaryTreePaths(TreeNode* root) {
        vector<string> ans;

        if(root==NULL)
        return ans;

        func(root, ans, to_string(root->val));
        return ans;
    }
};
```

24.) LOWEST COMMON ANCESTOR

CODE

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if(root==NULL || root==p || root==q)
        return root;

        TreeNode* left=lowestCommonAncestor(root->left, p, q);
        TreeNode* right=lowestCommonAncestor(root->right, p, q);

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

25.) MAX WIDTH OF BINARY TREE

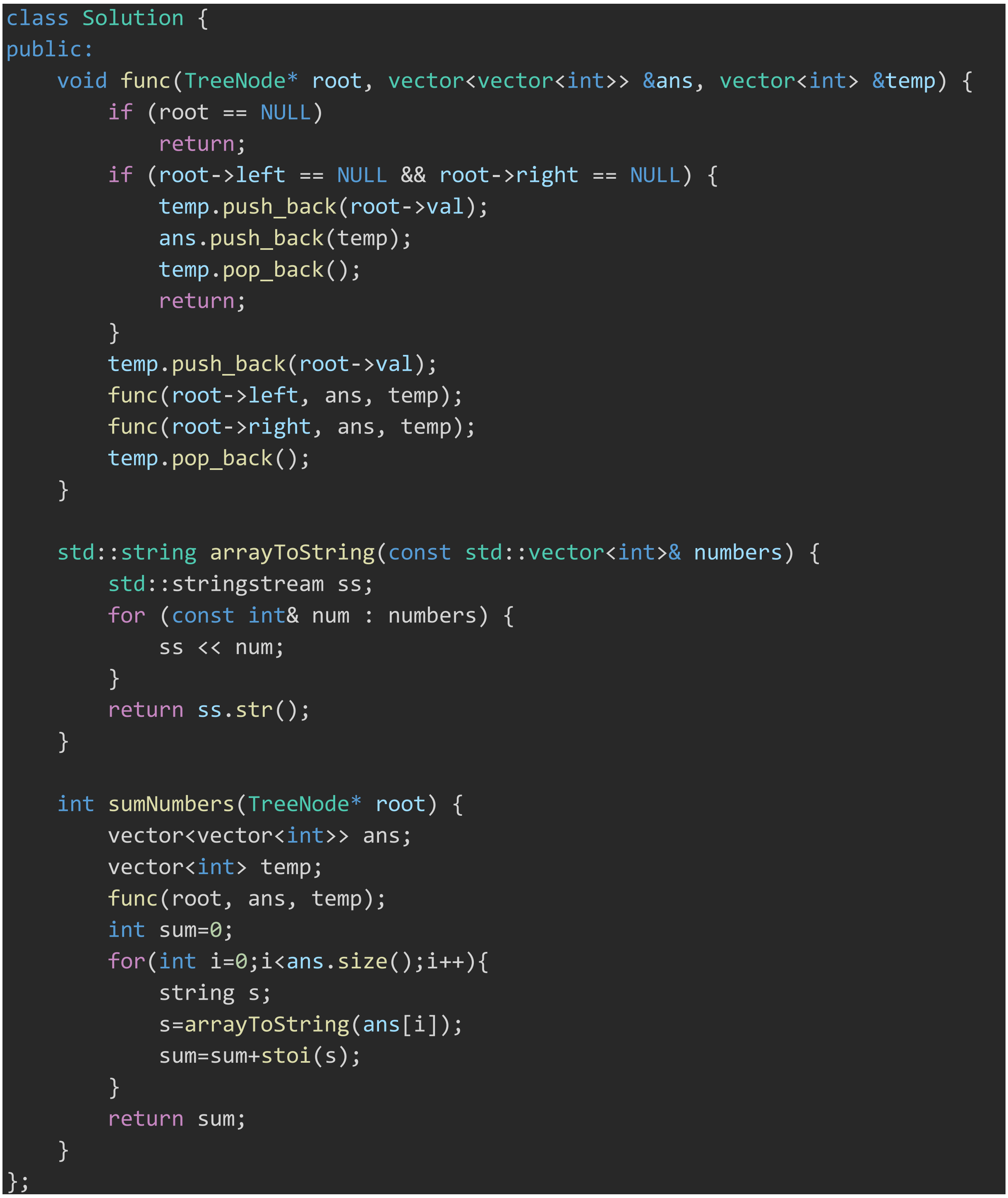
CODE

```
class Solution {
public:
    int widthOfBinaryTree(TreeNode* root) {
        if(root==NULL)
        return 0;

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

        while(!q.empty()){
            int size=q.size();
            int mini=q.front().second;
            int first, last;

            for(int i=0;i<size;i++){
                long long int curr=q.front().second-mini;
                TreeNode* node=q.front().first;
                q.pop();

                if(i==0)
                first=curr;
                if(i==size-1)
                last=curr;

                if(node->left)
                q.push({node->left, curr*2+1});
                if(node->right)
                q.push({node->right, curr*2+2});
            }
            ans=max(ans, last-first+1);
        }
        return ans;
    }
};
```

26.) SUM ROOT TO LEAF NODES

CODE

SAME AS ROOT TO LEAF NODE PATH WALA QUESTION



```
class Solution {
public:
    void func(TreeNode* root, vector<vector<int>> &ans, vector<int> &temp) {
        if (root == NULL)
            return;
        if (root->left == NULL && root->right == NULL) {
            temp.push_back(root->val);
            ans.push_back(temp);
            temp.pop_back();
            return;
        }
        temp.push_back(root->val);
        func(root->left, ans, temp);
        func(root->right, ans, temp);
        temp.pop_back();
    }

    std::string arrayToString(const std::vector<int>& numbers) {
        std::stringstream ss;
        for (const int& num : numbers) {
            ss << num;
        }
        return ss.str();
    }

    int sumNumbers(TreeNode* root) {
        vector<vector<int>> ans;
        vector<int> temp;
        func(root, ans, temp);
        int sum=0;
        for(int i=0;i<ans.size();i++){
            string s;
            s=arrayToString(ans[i]);
            sum=sum+stoi(s);
        }
        return sum;
    }
};
```

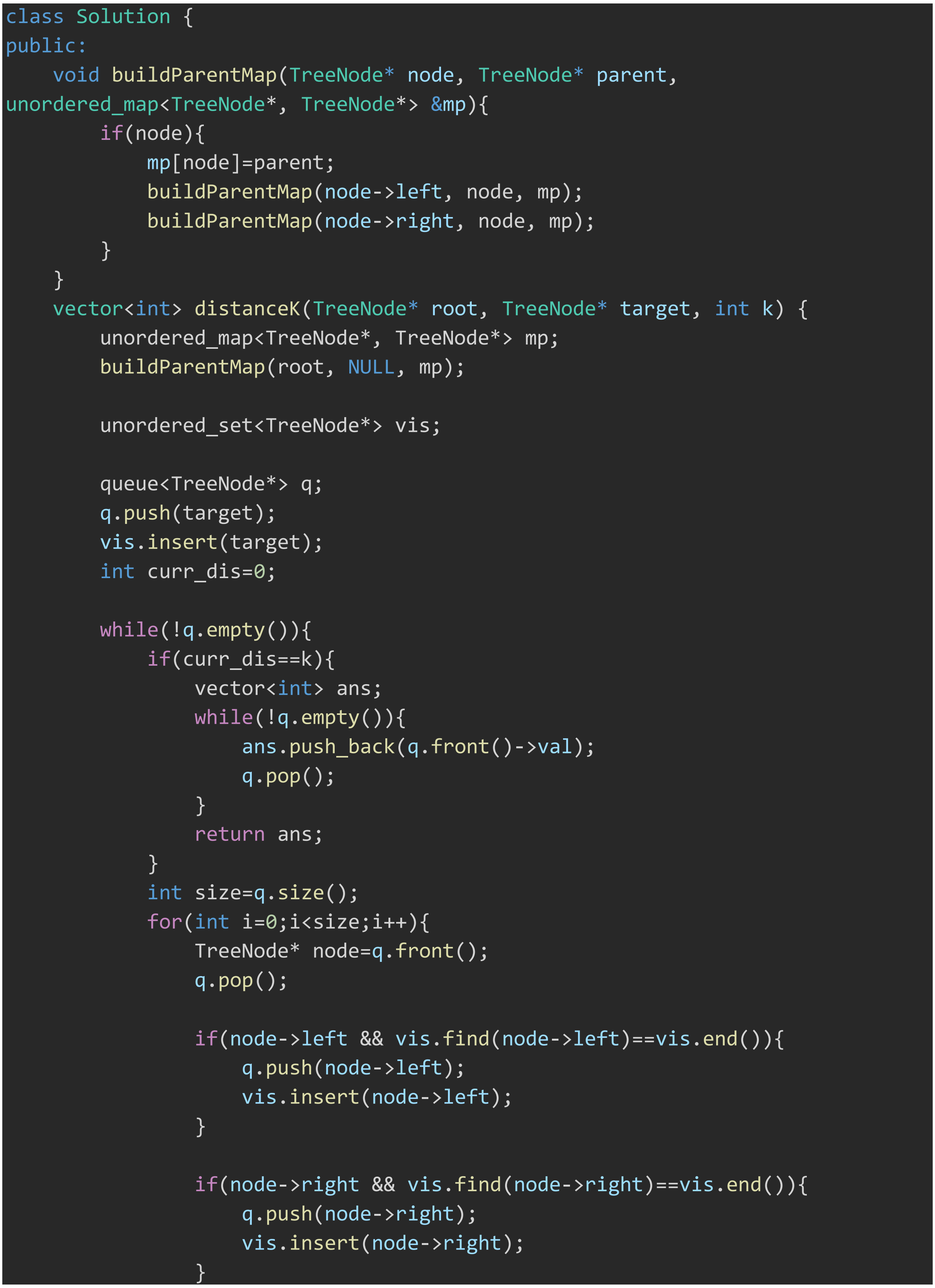
27.) CHECK FOR CHILDREN SUM PROPERTY

CODE

```
class Solution{
    public:
    //Function to check whether all nodes of a tree have the value
    //equal to the sum of their child nodes.
    int isSumProperty(Node *node){
        if(node==NULL)
        return 1;
        int sum=0;

        if(node->left==NULL && node->right==NULL)
        return 1;
        else{
            if(node->left!=NULL)
            sum=sum+node->left->data;
            if(node->right!=NULL)
            sum=sum+node->right->data;

            return ((node->data==sum)
                     && isSumProperty(node->left)
                     && isSumProperty(node->right));
        }
    }
};
```

28.) ALL NODES DISTANCE K IN BINARY TREE

CODE



```
class Solution {
public:
    void buildParentMap(TreeNode* node, TreeNode* parent,
unordered_map<TreeNode*, TreeNode*> &mp){
        if(node){
            mp[node]=parent;
            buildParentMap(node->left, node, mp);
            buildParentMap(node->right, node, mp);
        }
    }
    vector<int> distanceK(TreeNode* root, TreeNode* target, int k) {
        unordered_map<TreeNode*, TreeNode*> mp;
        buildParentMap(root, NULL, mp);

        unordered_set<TreeNode*> vis;

        queue<TreeNode*> q;
        q.push(target);
        vis.insert(target);
        int curr_dis=0;

        while(!q.empty()){
            if(curr_dis==k){
                vector<int> ans;
                while(!q.empty()){
                    ans.push_back(q.front()->val);
                    q.pop();
                }
                return ans;
            }
            int size=q.size();
            for(int i=0;i<size;i++){
                TreeNode* node=q.front();
                q.pop();

                if(node->left && vis.find(node->left)==vis.end()){
                    q.push(node->left);
                    vis.insert(node->left);
                }

                if(node->right && vis.find(node->right)==vis.end()){
                    q.push(node->right);
                    vis.insert(node->right);
                }
                if(mp[node] && vis.find(mp[node])==vis.end()){
                    q.push(mp[node]);
                    vis.insert(mp[node]);
                }
            }
            curr_dis++;
        }
        return {};
    }
};
```

29.) AMOUNT OF TIME FOR BINARY TREE TO BE INFECTED

CODE

```
class Solution {
public:
    TreeNode* findStartNode(TreeNode* node, int start,
unordered_map<TreeNode*, TreeNode*>& pMap) {
        if (!node) return nullptr;
        queue<TreeNode*> q;
        q.push(node);
        TreeNode* sNode = nullptr;

        while (!q.empty()) {
            TreeNode* curr = q.front();
            q.pop();

            if (curr->val == start) {
                sNode = curr;
            }

            if (curr->left) {
                pMap[curr->left] = curr;
                q.push(curr->left);
            }
            if (curr->right) {
                pMap[curr->right] = curr;
                q.push(curr->right);
            }
        }

        return sNode;
    }

    int bfs(TreeNode* sNode, unordered_map<TreeNode*, TreeNode*>& pMap) {
        unordered_set<TreeNode*> visited;
        queue<TreeNode*> q;
        q.push(sNode);
        visited.insert(sNode);
        int time = 0;

        while (!q.empty()) {
            int size = q.size();
            for (int i = 0; i < size; ++i) {
                TreeNode* node = q.front();
                q.pop();

                if (node->left && visited.find(node->left) == visited.end()) {
                    visited.insert(node->left);
                    q.push(node->left);
                }
                if (node->right && visited.find(node->right) == visited.end())
{
                    visited.insert(node->right);
                    q.push(node->right);
                }
                if (pMap[node] && visited.find(pMap[node]) == visited.end()) {
                    visited.insert(pMap[node]);
                    q.push(pMap[node]);
                }
            }
            if (!q.empty()) {
                ++time;
            }
        }

        return time;
    }
    int amountOfTime(TreeNode* root, int start) {
        unordered_map<TreeNode*, TreeNode*> parentMap;
        TreeNode* startNode = findStartNode(root, start, parentMap);
        return bfs(startNode, parentMap);
    }
};
```

30.) COUNT COMPLETE TREE NODES

CODE 01

```
class Solution {
public:
    void func(TreeNode* root, int &ans){
        if(root==NULL)
        return;
        ans=ans+1;
        func(root->left, ans);
        func(root->right, ans);
    }
    int countNodes(TreeNode* root) {
        int ans=0;
        func(root, ans);
        return ans;
    }
};
```

CODE 02

```
class Solution {
public:
    int countNodes(TreeNode* root) {
        if(root==NULL)
        return 0;
        int lh=0;
        int rh=0;
        TreeNode* l=root;
        TreeNode* r=root;

        while(l!=NULL){
            lh++;
            l=l->left;
        }
        while(r!=NULL){
            rh++;
            r=r->right;
        }

        if(lh==rh)
        return (1<<lh)-1;

        int left=countNodes(root->left);
        int right=countNodes(root->right);
        return 1+left+right;
    }
};
```

31.) MORRIS TRAVERSAL (INORDER)

CODE

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> ans;
        TreeNode* curr=root;

        while(curr!=NULL){
            if(curr->left==NULL){
                ans.push_back(curr->val);
                curr=curr->right;
            }
            else{
                TreeNode* prev=curr->left;
                while(prev->right!=NULL && prev->right!=curr){
                    prev=prev->right;
                }
                if(prev->right==NULL){
                    prev->right=curr;
                    curr=curr->left;
                }
                else{
                    prev->right=NULL;
                    ans.push_back(curr->val);
                    curr=curr->right;
                }
            }
        }
        return ans;
    }
};
```

32.) COVERT A TREE INTO ITS MIRROR

CODE

```
class Solution {
  public:
    // Function to convert a binary tree into its mirror tree.
    void mirror(Node* node) {
        if(node==NULL)
        return;

        Node* temp;
        temp=node->left;
        node->left=node->right;
        node->right=temp;

        mirror(node->left);
        mirror(node->right);
    }
};
```

33.) CONSTRUCT BINARY TREE FROM PREORDER AND INORDER TRAVERSAL

```
class Solution {
public:
    TreeNode* buildTreeHelper(vector<int>& preorder, int preStart, int preEnd,
                              vector<int>& inorder, int inStart, int inEnd,
                              unordered_map<int, int>& inorderMap) {
        if (preStart > preEnd || inStart > inEnd) {
            return nullptr;
        }

        int rootVal = preorder[preStart];
        TreeNode* root = new TreeNode(rootVal);

        int rootIdxInorder = inorderMap[rootVal];
        int leftSubtreeSize = rootIdxInorder - inStart;

        root->left = buildTreeHelper(preorder, preStart + 1,
                                     preStart + leftSubtreeSize, inorder,
                                     inStart, rootIdxInorder - 1, inorderMap);
        root->right = buildTreeHelper(preorder,
                                      preStart + leftSubtreeSize + 1,
                                      preEnd, inorder, rootIdxInorder + 1,
                                      inEnd, inorderMap);

        return root;
    }

    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        unordered_map<int, int> inorderMap;
        for (int i = 0; i < inorder.size(); ++i) {
            inorderMap[inorder[i]] = i;
        }
        return buildTreeHelper(preorder, 0, preorder.size() - 1, inorder, 0,
                               inorder.size() - 1, inorderMap);
    }
};
```

34.) CONSTRUCT BINARY TREE FROM INORDER AND POSTORDER TRAVERSAL

```
class Solution {
public:
    TreeNode* buildTreeHelper(vector<int>& inorder, int inStart, int inEnd,
                              vector<int>& postorder, int postStart,
                              int postEnd,
                              unordered_map<int, int>& inorderMap) {
        if (inStart > inEnd || postStart > postEnd) {
            return nullptr;
        }

        int rootVal = postorder[postEnd];
        TreeNode* root = new TreeNode(rootVal);

        int rootIdxInorder = inorderMap[rootVal];
        int leftSubtreeSize = rootIdxInorder - inStart;

        root->left = buildTreeHelper(inorder, inStart, rootIdxInorder - 1,
                                     postorder, postStart,
                                     postStart + leftSubtreeSize - 1,
                                     inorderMap);
        root->right = buildTreeHelper(inorder, rootIdxInorder + 1, inEnd,
                                      postorder, postStart + leftSubtreeSize,
                                      postEnd - 1, inorderMap);

        return root;
    }

    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        unordered_map<int, int> inorderMap;
        for (int i = 0; i < inorder.size(); ++i) {
            inorderMap[inorder[i]] = i;
        }
        return buildTreeHelper(inorder, 0, inorder.size() - 1, postorder, 0,
                               postorder.size() - 1, inorderMap);
    }
};
```

THANK YOU !