{Imp. Topics For Interview}

Lecture 71

Binary Trees

Date

Page

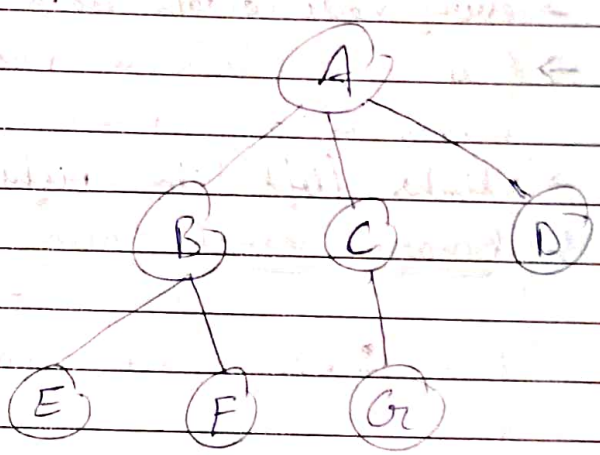
⇒ What is a Tree data Structure:-

→ Non-linear data structures stores hierarchical data.

→ elements are stored at different levels.

→ elements are called nodes ⇒ which are connected / linked together to represent hierarchy.

⇒ Terminology



- Root
- Child Node
- Parent Node
- Sibling Nodes → (E, F)
- External Node / Leaf Node → have no child nodes (E, F, G, D)
- No. of Edges ⇒ link b/w 2 nodes
- Level → no. of edges from root nodes
- Height → max. no. of edges b/w a leaf node and root node.
- size → no. of nodes in a tree
  if n nodes ⇒ n-1 edges
- subtree → tree which is a child of a node

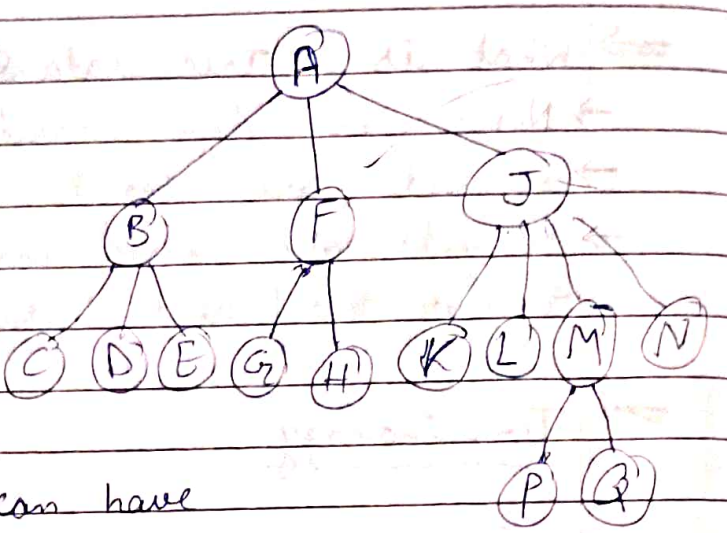
⇒ Binary Tree is said to be extended binary tree (or 2-tree) if each node has either 0 or 2 child. Nodes having 2 children is internal Node and is represented by circle. other external referred by square

Page

(A) Types of trees

1. Generic Trees

A node can have any no. of child nodes.



2. Binary Trees

→ every node contain data

→ tree in which a node can have maximum 2 child nodes.

→ 2 links (left child, Right child)

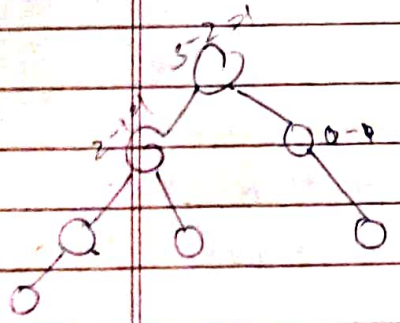
3. Binary Search trees (BST)

→ BT in which all nodes in left subtree will have value less than root node value. &&

All nodes in right subtree will have value more than root node value

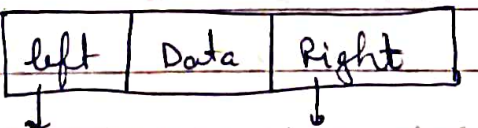
4. AVL Trees → self Balancing tree

→ diff. b/w heights of left Sub tree and right sub tree can be 1, 0, -1.



(A) Implementation of Node Class:-

Node



Stores the address of left & Right child nodes respectively.

* if any child does not exist, pointer will point to NULL.

```
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};
```

```
int main() {
    Node* root = new Node(2);
    root->left = " " (3);
    root->right = " " (4);
    cout << root->value << "\n";
    Cout << root->left->value
}   Cout << root->right->value
}
```

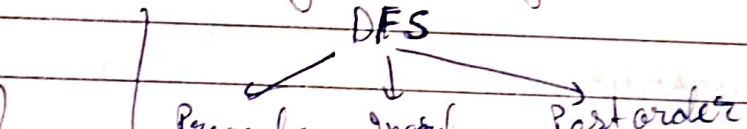
```
Class Node: {
public:
    int value;
    Node* left;
    Node* right;

    Node (int V) {
        value = V;
        left = right = NULL;
    }
};
```

⭐ Traversals

① DFS (Depth First search)
→ travel along the height.

① Preorder

(i) Visit the Root Node
(ii) Left Subtree
(iii) Right Subtree
} recursively.




```
Void preorderTraversal (Node* rootNode) {

    if (rootNode == NULL) {
        return;
    }
    Cout << rootNode->value << " ";

    preorderTraversal (rootNode->left);
    preorderTraversal (rootNode->right);
}
```

FREEMIND

Date

Page

(2) Inorder

(i) Visit left Subtree
(ii) Root node
(iii) Visit right Subtree
} recursively

```
Void InorderTraversal (Node* rootNode) {

    if (rootNode == NULL) {    //base case
        return;
    }
    //recursive call
    InorderTraversal (rootNode -> left);
    cout << rootNode -> value << " ";
    InorderTraversal (rootNode -> right);
}
```

(3) Postorder

(i) Visit left Subtree
(ii) Right subtree
(iii) Root Node
} recursively

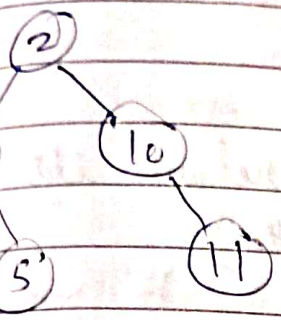
```
Void Postorder (Node* rootNode) {

    if (rootNode == NULL) {
        return;
    }
    Postorder (rootNode -> left);
    Postorder (rootNode -> right);
    cout << rootNode -> value << " ";
}
```

FREEMIND

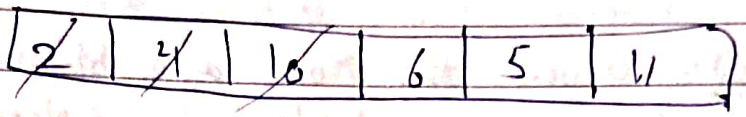
Date

Page

2) Levelorder → Breadth first Traversal



Use Queues data structure



Ek kaata or 2 ghusaye

```
#include <queue>
Class Node {
}
Void levelorderTraversal (Node* rootNode){

    if (rootNode == NULL) {
        return;
    }
    queue <Node*> q;
    q.push(rootNode)

    while (!q.empty()) {

        int nodesAtCurrentLevel = q.size();
        while(nodesAtCurrentLevel--) {
            Node* currNode = q.front();
            q.pop()
            Cout << currNode->value << " ";

            if (currNode->left) {
                q.push(currNode->left);
            }
            if (currNode->right) {
                q.push(currNode->right);
            }
```

```
        }cout << end;
    }
}
```

Q:- Given the root of a binary tree, return its maximum depth. (No. of Nodes Top to bottom)

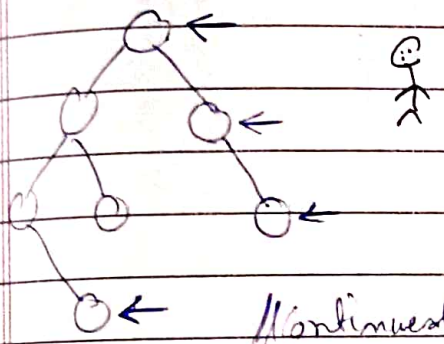
Sol:-
```
int maxDepth(Node* root){

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

    int leftDepth = maxDepth(root->left);
    int rightDepth = maxDepth(root->right);

    return (max(leftDepth, rightDepth) + 1);

}
```

Q:- Given the root of a binary tree, return the no. of leafnodes present in it.

Sol.→
```
int LeafNodes(Node* root){

    if(root == NULL){
        return 0;
    }
    if(root->left == NULL && root->right == NULL){
        return 1;
    }
```

```
int leftsubtreeleafnodes = LeafNodes(root->left);
int rightsubtreeleafnodes = LeafNodes(root->right);

return leftsubtreeLeafnodes + rightsubtreeLeafnodes;
}
```

Q5 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 from top to bottom.



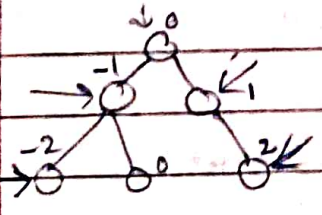
//continued

Sol

```
class Node{
};
Vector<int> rightViewBinaryTree(Node*root){

    Vector<int> ans;
    if(root == NULL){
        return ans;
    }
    queue<Node*> q;
    q.push(root);

    while(!queue.empty()){

        int nodesatcurrlevel = q.size();
        while(nodesatcurrlevel){

            Node* currNode = q.front();
            q.pop();
```

```
        if (nodesatCurrLevel == 1) {
            ans.pushback(currnode->value);
        }
        if (currnode->left) {
            q.push(currnode->left);
        }
        if (currnode->right) {
            q.push(currnode->right);
        }

        nodesatCurrLevel--;

    }
}
return ans;
}
int main() {
    Node* rootNode = new Node(2)
    ;
    ;

Vector<int> ans = rightViewBinaryTree(rootNode);
for (auto i: ans) {
    cout << i << " ";
} cout << endl;

return 0;
}
```

Q3→ Print the top view of a binary tree.



queue - Pair(Node, column)

Map - Column → Value.

Col. of left child → Col of Parent - 1

Col " Right " → Col " " + 1

FREEMIND

Date

Page

sol:-

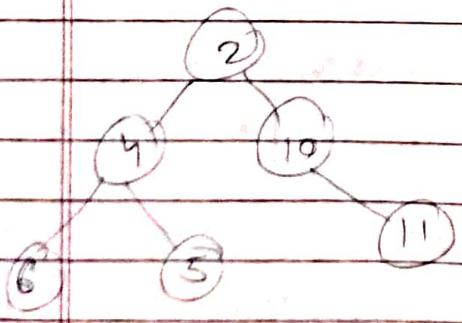
```
#include <vector>
#include <queue>
#include <map>
using namespace std;

Class Node {
};
Vector <int> TopViewBinaryTree(Node* root) {

    Vector <int> ans;
    if (root == NULL) {
        return ans;
    }
    queue < pair <Node*, int> > q;
    q.push (make_pair (root,0));
    map<int,int> m;
    while (!q.empty()) {
        int nodesatcurrlevel = q.size();
        while (nodesatcurrlevel--) {

            pair <Node*, int> p = q.front();
            Node* currNode = p.first;
            int currcolumn = p.second;
            q.pop();

            if (m.find(currNode) == m.end()) {
                m[currcolumn] = currNode->value;
            }
            if (currNode->left) {
            q.push(make_pair(currNode->left,
                             currcolumn-1));
            }
            if(currNode->right) {
            q.push(make_pair(currNode->right,
                             currcolumn+1)
```

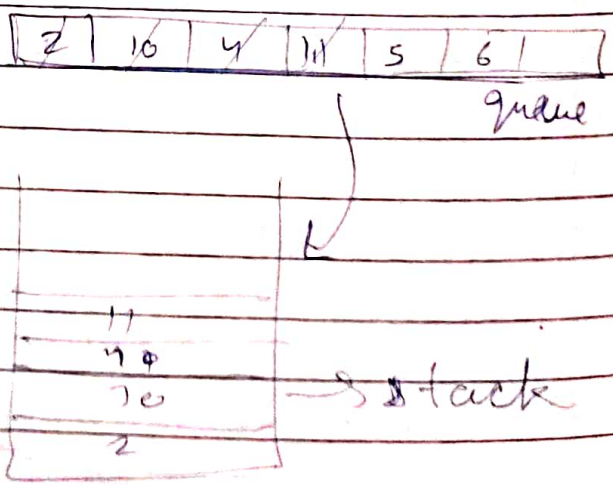
```
        }
    }
    for (auto it : m) {
        ans.push_back(it.second);
    }
}
int main() {
    Node* rootNode = new Node(2)
    ;
    Vector<int> ans = TopView BinaryTree (rootNode);
    for(auto i:ans) {
        cout<<i<<" ";
    } cout<<endl;
    return 0;
}
```

(N.W)

Q4) Given a binary tree, print the bottom view from left to right. A node is included in the bottom view if it can be seen when we look at the tree from the bottom.

Q5) Print the level order in reverse manner i.e. print the last row first and then the rows below.





FREEMIND

Date

Page

```
#include<vector>
#include<queue>
#include<stack>
using
```

```
Vector<int> reverseLevelOrder(Node* root){

    Vector<int> ans;
    if(root==NULL){
        return ans;
    }

    queue<Node*> q;
    stack<int> s;
    q.push(root);
    while(!q.empty()){

        int NodesatCurrlevel = q.size();
        while(NodesatCurrlevel--){
            Node* currNode = q.front();
            q.pop();
            s.push(currNode);
            if(currNode->right){
                q.push(currNode->right);
            }
            if(currNode->left){
                q.push(currNode->left);
            }
        }
    }
    while(!s.empty()){
        ans.push_back(s.top()->value);
        s.pop();
    }
```

FREEMIND

Date

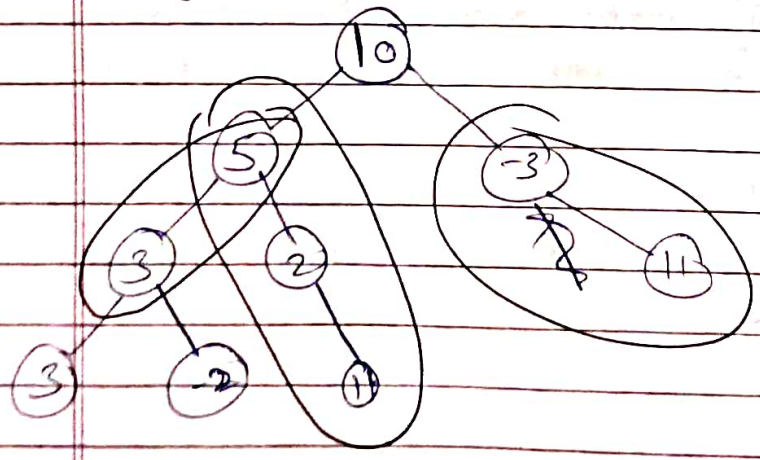
Page

```
    return ans;
}
int main() {

    Vector<int> ans = reverseLevelOrder(rootNode);
    for(auto i: ans)
        cout << ans << " ";
    } cout << endl;
    return 0;
}
```

Lecture 72

Interview Problems on Binary Trees.

Q) Given the root of a binary ~~tree~~ true and an integer targetSum, return the no. of paths where the sum of the values along the path equals targetSum. The path does not need to start or end at the root or a leaf, but it must go downwards (i.e. traveling only from parent nodes to child nodes).



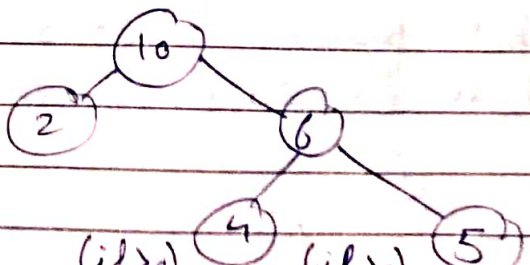
Date
Page

Soln

```
Class Solution {
public:
    int pathSumHelper(TreeNode* root, int targetSum,
                      int currSum, unordered_map
                      <long,int, int> pathCount){

        if (root == NULL){
            return 0;
        }
        currSum += root->value;
        int ansCount = pathCount[currSum - targetSum];
        pathCount[currSum]++;
        ansCount += pathSumHelper(root->left, targetSum,
                                  currSum, pathCount)
                  + pathSumHelper(root->right, targetSum,
                                  currSum, pathCount);
        pathCount[currSum]--; //while Backtracking.
        return ansCount;
    }
    int pathSum(TreeNode* root, int targetSum) {
    unordered_map<long int, int> pathCount;
    pathCount[0] = 1;
    return pathSumHelper(root, targetSum, 0, pathCount);
    }
};
```

Q2) Given the root of a binary tree, return the maximum path sum of any non-empty path.

Ans) 1. left child
2. Right child
3. both left & right child
4. only Root node



maxSum = (leftmaxSum + rightmaxSum + root->Val)
max(maxSum, ↑)

return root->val + max(leftmaxSum, rightmaxSum);

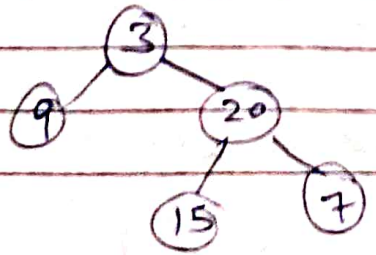
Sol.→
```
class Solution {
public:
    int maxSumPathHelper(TreeNode* root, int& maxSum) {
        if (root == NULL) {
            return 0;
        }
        int leftmaxSum = max(0, maxSumPathHelper(root->left, maxSum));
        int rightmaxSum = max(0, maxSumPathHelper(root->right, maxSum));
        maxSum = max(maxSum, root->val + leftmaxSum + rightmaxSum);
        return (root->val + max(leftmaxSum, rightmaxSum));
    }
    int maxPathSum(TreeNode* root) {
        int maxSum = INT_MIN;
        maxSumPathHelper(root, maxSum);
        return maxSum;
    }
};
```

T.C, S.C
O(N)

Q 3→ 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]



Sol:-

```
Class Solution {
public:
    TreeNode* buildTreeHelper(vector<int>& preorder, int prestart,
                              int preend, vector<int>& inorder,
                              int instart, int inend,
                              unordered_map<int, int>& inmap) {
        if (prestart > preend || instart > inend) {
            return NULL; //handling the leaf nodes.
        }
        TreeNode* rootNode = new TreeNode(preorder[prestart]);
        int rootInorderIndex = inmap[rootNode->val];
        int leftSubtreesize = rootInorderIndex - Instart;

        rootNode->left = buildTreeHelper(preorder, prestart+1,
                         prestart+leftSubtreesize, inorder, instart,
                         root inorder rootInorderIndex-1, inmap);
        rootNode->right = buildTreeHelper(preorder, prestart+
                          leftSubtreesize+1, preend, inorder,
                          rootInorderIndex+1, inend, inmap);
        return rootNode;
    }
    TreeNode* buildtree(vector<int>& preorder, vector<int>& inorder){
        unordered_map<int, int> inmap;
        for (int i=0; i<inorder.size(); i++) {
            inmap[inorder[i]] = i;
        }
        return buildTreeHelper(preorder, 0, preorder.size()-1,
                               inorder, 0, inorder.size()-1, inmap);
    }
};
```

T.C = O(N)
S.S = O(N)

FREEMIND

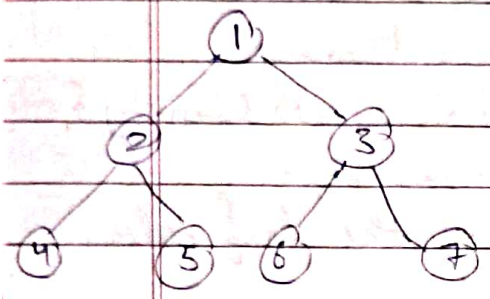
Date

Page

⭐ Construct BT From Preorder and Postorder

input :- preorder = [1, 2, 4, 5, 3, 6, 7], postorder = [4, 5, 2, 6, 7, 3, 1]

output :-



	Left Subtree	Right Subtree
Postree :	Poststart, Left Child index	left child index + 1, Postend - 1
Preorder :	Prestart + 1	Prestart + leftSubTree +1, Preend

Prestart + left Subtree Elements

leftchildIndex - poststart + 1

Sol :-

```
Class Solution {
public:
    TreeNode* ConstructFromPrePostHelper(Vector<int>& preorder,
        int prestart, int preend, Vector<int>& postorder,
        int poststart, int postend, Unordered_map
                                    <int,int>& postmap) {
        if (prestart > preend || poststart > postend) {
            return NULL;
        }
        TreeNode* rootNode = new TreeNode(preorder[prestart]);
        if (prestart == preend) {
            return rootNode;
        }
        int leftchildVal = preorder[prestart+1];
        int leftchildIndex = postmap[leftchildval];
        int leftSubtreeSize = leftchildIndex - poststart +1;

        rootNode->left = ConstructFromPrePostHelper(
(preorder, prestart+1, prestart+leftSubtreesize, postorder, poststart,
    leftchildIndex, postmap);
```

rootNode->right = ConstructFromPrePostHelper(preorder, preStart+leftSubtreeSize+1, preend, postorder, leftchildIndex+1, postend-1, postmap);

return rootNode;

}

TreeNode* constructFromPrePost(Vector<int>& preorder, Vector<int>& postorder) {

unordered_map<int, int> postmap;

for(int i=0; i<postorder.size(); i++) {

postmap[postorder[i]] = i;

}

return constructFromPrePostHelper(preorder, 0, preorder.size()-1, postorder, 0, postorder.size()-1, postmap);

}

};

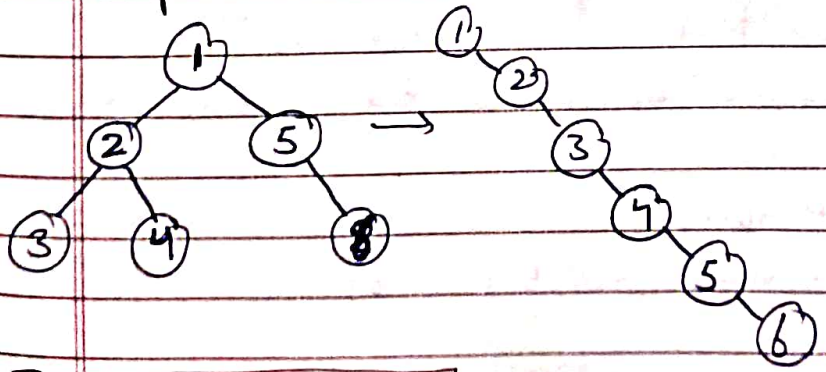
(A) FLatten Binary Tree to linked list:

⇒ ① The ll 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.

② The "linked-list" should be in the same order as a pre-order traversal of the binary tree.

input:- root = [1,2,5,3,4,null,6]

output:- 1→2→3→4→5→6




```
{ if(root==NULL) return; //base case
  flatten(root->right)
  flatten(root->left) //recursive call
  root->right = last;
  root->left = NULL;
  last = root; // self work
  return;
}
```

T.C, S.C → O(N)

FREEMIND

Date

Page

Ans:

```
Class Solution {
public:
    TreeNode* lastNode = NULL;
    Void flatten (TreeNode* root) {

        if (root == NULL) {
            return;
        }

        flatten (root → right);
        flatten (root → left);

        root → right = lastNode;
        root → left = NULL;

        lastNode = root;
        return;
    }
};
```

(A) Amount of time for binary tree to be infected.

Q) You are given the root of a binary tree with unique values, and an integer start. At minute 0, an infection starts from the node with value start. Each minute a node becomes infected if:

- The node is currently uninfected.
- The node is adjacent to an infected node.

Return the no. of minutes needed for the entire tree to be infected.

input:- root = [1,2,5,3,4,null,6]

output:- [1,null,2,null,3,null,4,null,5,null,6]

FREEMIND
Date
Page

Sol:-

```
class Solution {
public:
    int calculateTime(TreeNode* startNode, unordered_map<
                      TreeNode*, TreeNode*>& parent){
        unordered_set<TreeNode*> infected;
        queue<TreeNode*> q;
        q.push(starNode);
        infected.insert(starNode);
        int time = 0;

        while(!q.empty()){
            int currLevelNodes = q.size();
            bool infectFlag = false;
            while(currLevelNodes--){

                TreeNode* currNode = q.front();
                q.pop();
                if(currNode->left && !infected.count(
                                    currNode->left)){
                    infectFlag = true;
                    infected.insert(currNode->left);
                    q.push(currNode->left);
                }
                if(currNode->right && !infected.count(currNode->right
                                                      )){

                    infectFlag = true;
                    infected.insert(currNode->right);
                    q.push(currNode->right);
                }
            }
        }
```

FREEMIND
Date
Page

```
        if (parent[currNode] && !infected.count(parent[currNode])) {
            infectFlag = true;
            infected.insert(parent[currNode]);
            q.push(parent[currNode]);
        }
    }
    if (infectFlag) time++;
}
return time;
}
TreeNode* makeParent(TreeNode* root, unordered_map<TreeNode*, TreeNode*>
                                        &parent, int start) {
    queue<TreeNode*> q;
    q.push(root);
    TreeNode* startNode;
    while(!q.empty()){
        TreeNode* currNode = q.front();
        q.pop();
        if(currNode->val == start){
            startNode = currNode;
        }
        if(currNode->left){ parent[currNode->left] = currNode;
                            q.push[currNode->left];
        } if(currNode->right){ parent[currNode->right] = currNode;
                               q.push[currNode->right];
        }
    } return startNode;
}
int amountOfTime(TreeNode* root, int start) {
    unordered_map<TreeNode*, TreeNode*> parent;
    TreeNode* startNode = makeParent(root, parent, start);
    return calculateTime(startNode, parent);
}
```