1. Identical Binary Trees

int Solution::isSameTree(TreeNode\* A, TreeNode\* B) {

    if(!A && !B) return 1; //important base

    if((!A && B) || (A && !B)) return 0; //case lines

    if(A->val!=B->val) return 0;

    int left=isSameTree(A->left,B->left);

    int right=isSameTree(A->right,B->right);

    if(left && right) return 1;

    else return 0;

}

1. Invert the Binary Tree

TreeNode\* Solution::invertTree(TreeNode\* A) {

if (!root) return root;

root->left = invertTree(root->left);

root->right = invertTree(root->right);

TreeNode \*temp = root->left;

root->left = root->right;

root->right = temp;

return root;

}

1. Maximum Level Sum

int Solution::solve(TreeNode\* root){

if (root == NULL)

return 0;

int result = root->val;

queue<TreeNode\*> q;

q.push(root);

while (!q.empty())

{

int count = q.size() ;

int sum = 0;

while (count--)

{

TreeNode \*temp = q.front();

q.pop();

sum = sum + temp->val;

if (temp->left != NULL)

q.push(temp->left);

if (temp->right != NULL)

q.push(temp->right);

}

result = max(sum, result);

}

return result;

}

1. Symmetric Binary Tree

bool solve(TreeNode\* a,TreeNode\* b)

 {

     if(a==NULL||b==NULL) return (a==b);

     if(a->val!=b->val) return false;

     return solve(a->left,b->right)&&solve(a->right,b->left);

 }

int Solution::isSymmetric(TreeNode\* A) {

    return solve(A->left,A->right);

}

1. Path Sum

int Solution::hasPathSum(TreeNode\* root, int targetSum) {

      if(root == NULL){

            return false;

        }else if(root->left == NULL && root->right == NULL

                 && root->val == targetSum){

            return true;

        }else {

            return (hasPathSum(root->left,targetSum - root->val)||

                hasPathSum(root->right,targetSum - root->val));

    }

}

1. Sorted array to Balanced BST

TreeNode \*res(const vector<int> &vec, int start, int end){

if(start > end) return nullptr;

if(start == end) return new TreeNode(vec[start]);

int mid = start + (end-start)/2;

TreeNode \*node = new TreeNode(vec[mid]);

node->left = res(vec, start, mid-1);

node->right = res(vec, mid+1, end);

return node;

}

TreeNode\* Solution::sortedArrayToBST(const vector<int> &A) {

return res(A, 0, A.size()-1);

}

1. Balanced Binary Tree

bool flag;

int checkit(TreeNode\* A) {

if (!flag) return 0;

if (A == NULL) return 0;

int left, right;

left = checkit(A->left);

right = checkit(A->right);

if (abs(left - right) >= 2) {

flag = false;

return 0;

}

return max(right, left) + 1;

}

int Solution::isBalanced(TreeNode\* A) {

flag = true;

checkit(A);

return flag;

}

1. Merge two Binary Tree

void adding(TreeNode a, TreeNode b){if(a->left and b->left){adding(a->left,b->left);}a->val=a->val+b->val;if(b->left and a->left==nullptr){TreeNodetemp=b->left;a->left=temp;}if(a->right and b->right){adding(a->right,b->right);}if(b->right and a->right==nullptr){TreeNode temps=b->right;  
a->right=temps;  
}

}

TreeNode\* Solution::solve(TreeNode\* A, TreeNode\* B) {  
if(A==nullptr)return B;  
if(B==nullptr)return A;  
adding(A,B);  
return A;

}

1. Postorder Traversal

vector<int> Solution::postorderTraversal(TreeNode\* root) {

    vector<int>v;

    stack<TreeNode \*>s;

    s.push(root);

while(s.empty()==false){

    TreeNode \*curr=s.top();

    v.push\_back(curr->val);

    s.pop();

    if(curr->left!=NULL){

        s.push(curr->left);

    }

    if(curr->right!=NULL){

        s.push(curr->right);

    }

}

reverse(v.begin(),v.end());

return v;

}

1. Inorder Traversal

vector<int> Solution::inorderTraversal(TreeNode\* A) {

stack<TreeNode\*> s;

vector<int> v;

if(A==NULL) return v;

TreeNode\* curr=A->left;

s.push(A);

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

while(curr){

s.push(curr);

curr=curr->left;

}

TreeNode\* temp=s.top();

s.pop();

v.push\_back(temp->val);

curr=temp->right;

}

return v;

}

1. Path to given Node

bool dfs(TreeNode\* A, int B,vector<int>& path){

if(!A) return false;

path.push\_back(A->val);

if(B==A->val) return true;

if(dfs(A->left,B,path)) return true;

if(dfs(A->right,B,path)) return true;

path.pop\_back();

return false;

}

vector<int> Solution::solve(TreeNode\* A, int B) {

vector<int> path;

dfs(A,B,path);

return path;

}

1. Next pointer binary tree

void Solution::connect(TreeLinkNode\* A) {

queue<TreeLinkNode\*> q;

q.push(A);

q.push(NULL);

TreeLinkNode\* t;

while(!q.empty()){

t = q.front();

q.pop();

if(t){

t->next = q.front();

if(t->left) q.push(t->left);

if(t->right) q.push(t->right);

} else if(!q.empty()){

q.push(NULL);

} else

break;

}

}

1. Remove half nodes

//use of post-order traversal

TreeNode\* Solution::solve(TreeNode\* A) {

if(A->left == NULL && A->right == NULL)return A;

if(A->left == NULL)

return solve(A->right);

if(A->right == NULL)

return solve(A->left);

A->left = solve(A->left);

A->right = solve(A->right);

return A;

}

1. XOR between two arrays //trie question
2. Valid binary search tree //keep in mind conditions

int isBST(TreeNode A,int min,int max){if(!A) return 1;if(A->val>max||A->val<min) return 0;return isBST(A->left,min,A->val-1)&&isBST(A->right,A->val+1,max);}int Solution::isValidBST(TreeNode A) {  
return isBST(A,INT\_MIN,INT\_MAX);  
}

1. Max depth of binary tree

int Solution::maxDepth(TreeNode\* A) {

    if(!A) return 0;

    return 1+max(maxDepth(A->left),maxDepth(A->right));

}

1. Consecutive parent child

int Solution::consecutiveNodes(TreeNode\* A) {

    if(!A) return 0;

    return ((A->left && abs(A->left->val-A->val)==1)?1:0) + ((A->right && abs(A->right->val-A->val)==1)?1:0) + consecutiveNodes(A->left) + consecutiveNodes(A->right);

}

1. Preorder Traversal

vector<int> Solution::preorderTraversal(TreeNode\* root) {

        vector<int>v;

    stack<TreeNode \*>s;

    s.push(root);

while(s.empty()==false){

    TreeNode \*curr=s.top();

    v.push\_back(curr->val);

    s.pop();

    if(curr->right!=NULL){

        s.push(curr->right);

    }

    if(curr->left!=NULL){

        s.push(curr->left);

    }

}

return v;

}

1. Min depth of binary tree

int ans;

 void func(TreeNode\* root , int curr\_count)

 {

    /\* if(root==NULL)

     {

         return 0

     }\*/

     // this will be give u wrong ans , as if one of the child is null , it will take

     // null child as value of depth 0

     // so put the condition of leaf nodes

     if(root->left==NULL && root->right==NULL)

     {

         ans = min(ans , curr\_count);

         return ;

     }

     if(root->left)

     {

         func(root->left , curr\_count+1);

     }

     if(root->right)

     {

         func(root->right , curr\_count+1);

     }

 }

int Solution::minDepth(TreeNode\* root) {

  ans=INT\_MAX;

  if(root==NULL)

  {

      return 0;

  }

  func(root , 1);

  return ans;

  // curr\_count is intially bcoz of root node

}

1. Vertical sum of a Binary Tree

map<int, int> mp;

void rec(TreeNode\* root, int HD){

if(root == NULL){

return;

}

mp[HD] += root -> val;

rec(root -> left, HD - 1);

rec(root -> right, HD + 1);

}

vector<int> Solution::verticalSum(TreeNode\* A) {

mp.clear();

rec(A, 0);

vector<int> ans;

for(auto &x : mp){

ans.push\_back(x.second);

}

return ans;

}

1. Reverse Level Order

vector<int> Solution::solve(TreeNode\* root) {

vector<int> res;

    if (root == NULL)

        return res;

    queue<TreeNode\*> q;

    q.push(root);

    while (!q.empty())

    {

        int count = q.size() ;

        int sum = 0;

        while (count--)

        {

            TreeNode \*temp = q.front();

            q.pop();

            res.push\_back(temp->val);

            if (temp->right != NULL)

                q.push(temp->right);

            if (temp->left != NULL)

                q.push(temp->left);

        }

    }

reverse(res.begin(),res.end());

return res;

}

1. Kth smallest element in tree

int Solution::kthsmallest(TreeNode\* A, int B) {

    stack<TreeNode\*> s;

    vector<int> v;

    //if(A==NULL) return v;

    TreeNode\* curr=A->left;

    s.push(A);

    while(!s.empty() || curr){

        while(curr){

            s.push(curr);

            curr=curr->left;

        }

        TreeNode\* temp=s.top();

        s.pop();

        v.push\_back(temp->val);

        curr=temp->right;

    }

    return v[B-1];

}

1. Right view of a binary tree

vector<int> Solution::solve(TreeNode\* root) {

if(root==NULL)

return {};

queue<TreeNode\*>q;

vector<int>v;

q.push(root);

// v.push\_back(root->data);

while(!q.empty())

{

int n=q.size();

for(int i=1;i<=n;i++)

{

TreeNode\* temp=q.front();

q.pop();

if(i==n)

{

v.push\_back(temp->val);

}

if(temp->left)

q.push(temp->left);

if(temp->right)

q.push(temp->right);

}

//Node\*x=q.front();

// v.push\_back(x->data);

}

return v;

}

1. Root to leaf paths with sum

* // If you are using C++, make sure to pass the array by reference.

bool f(TreeNode\* root, int currsum, int sum, vector<vector<int>> &ans, vector<int> &tempAns){

    if(root->left == NULL and root->right == NULL){

        currsum += root->val;

        tempAns.push\_back(root->val);

        if(currsum == sum){

            ans.push\_back(tempAns);

            tempAns.pop\_back();

            return true;

        }

        tempAns.pop\_back();

        return false;

    }

    tempAns.push\_back(root->val);

    bool left = 0, right = 0;

    if(root->left != NULL){

        left = f(root->left, currsum + root->val, sum, ans, tempAns);

    }

    if(root->right != NULL){

        right = f(root->right, currsum + root->val, sum, ans, tempAns);

    }

    tempAns.pop\_back();

    return left or right;

}

vector<vector<int> > Solution::pathSum(TreeNode\* root, int sum) {

    vector<vector<int>> ans;

    vector<int> tempAns;

    f(root, 0, sum, ans, tempAns);

    return ans;

}

1. Zigzag level order traversal binary tree

vector<vector<int> > Solution::zigzagLevelOrder(TreeNode\* A) {

deque<TreeNode\*> q;

vector<vector<int>> res;

vector<int> t;

q.push\_back(A);

int level = 0;

while(!q.empty()){

int n = q.size();

t.clear();

for(int i=0;i<n;i++){

if(level%2==0){

TreeNode \*temp = q.front();

q.pop\_front();

t.push\_back(temp->val);

if(temp->left)q.push\_back(temp->left);

if(temp->right)q.push\_back(temp->right);

}else{

TreeNode \*temp = q.back();

q.pop\_back();

t.push\_back(temp->val);

if(temp->right)q.push\_front(temp->right);

if(temp->left)q.push\_front(temp->left);

}

}

res.push\_back(t);

level++;

}

return res;

}

1. Sum root to leaf numbers

int fun(TreeNode\* A,int num){

      num=(num\*10+A->val)%1003;

     if(A->left==NULL&&A->right==NULL){return num;}

     int l1=0,r1=0;

     if(A->left) l1=fun(A->left,num);

     if(A->right) r1=fun(A->right,num);

     return (r1+l1);

 }

int Solution::sumNumbers(TreeNode\* A) {

    return fun(A,0)%1003;

}

1. Construct binary tree from inorder and preorder

TreeNode \*buildTreeUtil(vector<int> &preorder, vector<int> &inorder, int preStart, int preEnd, int inStart, int inEnd){

if (preStart > preEnd || inStart > inEnd) return nullptr;

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

int k = 0;

for (int i = inStart; i <= inEnd; i++){

if (inorder[i] == preorder[preStart]){

k = i;

break;

}

}

root->left = buildTreeUtil(preorder, inorder, preStart + 1, preStart + (k - inStart), inStart, k-1);

root->right = buildTreeUtil(preorder, inorder, preStart + (k-inStart+1), preEnd, k+1, inEnd);

return root;

}

TreeNode\* Solution::buildTree(vector<int> &preorder, vector<int> &inorder) {

if (preorder.size() <= 0) return nullptr;

return buildTreeUtil(preorder, inorder, 0, preorder.size()-1, 0, inorder.size()-1);

}

1. Diagonal Traversal

map<int, vector<int>> m;

void preOrder(TreeNode\* a, int dia) {

    if(!a) return;

    m[dia].push\_back(a->val); //hashing

    preOrder(a->left,dia+1); //giving diagonal elements same value to identify them

    preOrder(a->right,dia);

}

vector<int> Solution::solve(TreeNode\* a) {

    m = map<int,vector<int>> ();

    preOrder(a,0);

    vector<int> ans;

    for(auto it:m) {

        for(auto it1:it.second) {

            ans.push\_back(it1);

        }

}

return ans;

}

1. Binary tree from inorder and postorder

TreeNode \*buildTreeUtil(vector<int> &postorder, vector<int> &inorder, int postStart, int postEnd, int inStart, int inEnd){

    if (postStart > postEnd || inStart > inEnd) return nullptr;

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

    int k = 0;

    for (int i = inStart; i <= inEnd; i++){

        if (inorder[i] == postorder[postEnd]){

            k = i;

            break;

        }

    }

    root->left = buildTreeUtil(postorder, inorder, postStart,postStart+ k-inStart-1, inStart, k-1);

    root->right = buildTreeUtil(postorder, inorder, postEnd-inEnd+k, postEnd-1, k+1, inEnd);

    return root;

}

TreeNode\* Solution::buildTree(vector<int> &inorder, vector<int> &postorder) {

    if (postorder.size() <= 0) return nullptr;

    return buildTreeUtil(postorder, inorder, 0, postorder.size()-1, 0, inorder.size()-1);

}

1. Inorder Traversal of a cartesian tree //max-heap binary tree

TreeNode\* Solution::buildTree(vector<int> &A) {

    int maxpos=-1;

    if(A.empty())

    return NULL;

    for(int i=0;i<A.size();i++)

    {

        if(A[maxpos]<A[i])

        maxpos=i;

    }

    vector<int> left(A.begin(),A.begin()+maxpos);

    vector<int> right(A.begin()+maxpos+1,A.end());

    TreeNode\* root=new TreeNode(A[maxpos]);

    root->left=buildTree(left);

    root->right=buildTree(right);

    return root;

}

1. Maximum Edge Removal

const int maxN = 1e5 + 5;

vector<int> g[maxN];

int res = 0;

// Utility method to do DFS of the graph and count edge removal

int dfs(int u, int par) {

int currComponentNode = 0;

// iterate over all neighbor of node u

for (auto v: g[u]) {

if(v == par){

continue;

}

// Count the number of nodes in a subtree

int subtreeNodeCount = dfs(v, u);

// if returned node count is even, disconnect the subtree and increase result by one.

if (subtreeNodeCount % 2 == 0)

res++;

// else add subtree nodes in current component

else

currComponentNode += subtreeNodeCount;

}

// number of nodes in current component and one for current node

return (currComponentNode + 1);

}

int Solution::solve(int A, vector<vector<int> > &B) {

assert(A%2 == 0 && A <= 1e5);

for(int i = 1; i <= A; i++){

g[i].clear();

}

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

g[B[i][0]].push\_back(B[i][1]);

g[B[i][1]].push\_back(B[i][0]);

}

res = 0;

// calling the dfs from node-1 and making its parent as -1

dfs(1, 0);

return res;

}

1. Least Common Ancestor //cnc code is complex, this is better

bool helper(TreeNode\* root,int B,int C,TreeNode\* &ans){

if(!root) return false;

bool flag = root->val==B || root->val==C;

bool l = helper(root->left,B,C,ans);

bool r = helper(root->right,B,C,ans);

if(flag && B==C)

ans = root;

if(((flag && l) || (flag && r) || (l && r)) && !ans)

ans = root;

return flag | l | r;

}

int Solution::lca(TreeNode\* A, int B, int C) {

TreeNode\* ans = NULL;

helper(A,B,C,ans);

return ans ? ans->val : -1;

}

1. Cousins in Binary Tree

//level-order, but don’t push the node which is sibling of B

vector<int> Solution::solve(TreeNode\* A, int B) {

     vector<int> ret;

    queue<TreeNode\*>q;

    q.push(A);

    bool found = false;

    while(!q.empty() && !found){

        int size = q.size();

        while(size--){

            TreeNode \*t = q.front();

            q.pop();

            if(t->left && t->left->val == B){

                found = true;

            }

            else if(t->right && t->right->val == B){

                found = true;

            }

            else{

                if(t->left) q.push(t->left);

                if(t->right) q.push(t->right);

            }

        }

        if(found){

            while(!q.empty()){

                TreeNode \*t = q.front();

                q.pop();

                ret.push\_back(t->val);

            }

        }

    }

    return ret;

}

1. Burn a tree

we want that our diameter to always picks up the given leaf node with value B.

So when the recursive call, comes to the leaf, I return a very high value, and it will ensure that my leaf node always gets picked up.

int fun(TreeNode\* root, int &res, int B){  
if(!root) return 0;  
if(root->left==NULL && root->right==NULL && root->val==B) return (10000000);  
int l=fun(root->left, res, B);  
int r=fun(root->right, res, B);  
int temp=1+ max(l, r);  
int ans=max(temp, l+r+1);  
res= max(ans, res);  
return temp;  
}  
int Solution::solve(TreeNode\* A, int B) {  
int res=0;  
fun(A, res, B);  
return res-10000000;  
}

1. Valid BST from preorder

Here we find the next greater element and after finding next greater, if we find a smaller element, then return false.

1. Create an empty stack.
2. Initialize root as INT\_MIN.
3. Do following for every element pre[i]
   1. If pre[i] is smaller than current root, return false.
   2. Keep removing elements from stack while pre[i] is greater  
      then stack top. Make the last removed item as new root (to  
      be compared next).  
      At this point, pre[i] is greater than the removed root  
      (That is why if we see a smaller element in step a), we  
      return false)
   3. push pre[i] to stack (All elements in stack are in decreasing  
      order)

int Solution::solve(vector<int> &A) {

    stack<int> st;

    int root=-1e9;

    for(auto curr : A){

        while(st.size()>0 and curr >=st.top()){

            root=st.top();

            st.pop();

        }

        if(root>=curr) return false;

        st.push(curr);

    }

    return true;

}

1. Shortest Unique Prefix

struct Trie{

char c;

bool eow;

int fr;

Trie \*children[26];

Trie(char x){

for(int i=0;i<26;i++)children[i]=nullptr;

c=x;

fr=0;

eow=0;

}

};

vector<string> Solution::prefix(vector<string> &vec) {

Trie\* root=new Trie('/');

vector<string>res;

for(string s:vec){

int n=s.size();

Trie\* curr=root;

for(int i=0;i<n;i++){

if(!curr->children[s[i]-'a']){

curr->children[s[i]-'a']=new Trie(s[i]-'a');

}

curr->children[s[i]-'a']->fr++;

curr=curr->children[s[i]-'a'];

}

curr->eow=1;

}

for(string s:vec){

Trie\* curr=root;

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

curr=curr->children[s[i]-'a'];

if(curr->fr==1){

string sub=s.substr(0,i+1);

res.push\_back(sub);

break;

}

}

}

return res;

}

1. Vertical order traversal of binary tree

//don’t use dfs coz of condition of level order, use bfs

map<int,vector<int>>mp;

 void dfs(TreeNode\*A, int pos){

     if(A==NULL) return;

     queue<pair<TreeNode\*,int>>q;

     q.push({A,pos});

     while(!q.empty()){

        pair<TreeNode\*,int>front=q.front();

        q.pop();

        mp[front.second].push\_back(front.first->val);

        if(front.first->left!=NULL) q.push({front.first->left,front.second-1});

        if(front.first->right!=NULL)  q.push({front.first->right,front.second+1});

     }

 }

vector<vector<int> > Solution::verticalOrderTraversal(TreeNode\* A) {

    mp.clear();

     vector<vector<int>>ans;

    dfs(A,0);

    for(auto i:mp){

        ans.push\_back(i.second);

    }

    return ans;

}

1. Hotel reviews //trie-based
2. Last Node in a complete binary tree

int Solution::lastNode(TreeNode\* root) {

queue<TreeNode\*> q;

q.push(root);

int ans;

while(!q.empty())

{

TreeNode\* temp = q.front();

ans = temp->val;

q.pop();

if(temp->left)

{

q.push(temp->left);

}

if(temp->right)

{

q.push(temp->right);

}

}

return ans;

}

1. Covered uncovered nodes

long Solution::coveredNodes(TreeNode \* A) {

long cover = 0, uncover = 0;

queue < TreeNode \* > q;

q.push(A);

int level = 0;

while (!q.empty()) {

int n = q.size();

for (int i = 0; i < n; i++) {

TreeNode \* node = q.front();

q.pop();

if (i == 0 || i == n - 1) {

uncover += node -> val;

} else {

cover += node -> val;

}

if (node -> left != NULL)

q.push(node -> left);

if (node -> right != NULL)

q.push(node -> right);

}

}

return abs(cover - uncover);

}

1. Construct bst from preorder

TreeNode\* build(vector <int>&A, int &i, int bound){

if(i == A.size() || A[i] > bound) return NULL;

TreeNode\* root = new TreeNode (A[i++]);

root->left = build(A, i, root->val);

root->right = build(A, i, bound);

return root;

}

TreeNode\* Solution::constructBST(vector<int> &A) {

int i = 0;

return build(A, i, INT\_MAX);

}

1. BST Iterator

//Lets look at the version of this problem when the trees have a back pointer. Can you solve the problem without using additional space ? When you are on node N and are asked for next element, you obviously won’t go to the left subtree as all the elements there are smaller than N. We would go to the smallest number in the right subtree if the right subtree is not null. If the right subtree is null, that means that we need to move up, and keep moving up till we are coming from the right subtree.  
Now we don’t have the back pointer in this case. So, we need something to keep track of the path from root to the current node, so we can move to the parent when needed. Do note that storing the path from root to the current node only requires memory equivalent to the length of the path which is the depth of the tree. Also, we can track the path using stack.

stack<TreeNode\*> S;

BSTIterator::BSTIterator(TreeNode \*root) {

//S.clear();

while (root) {

S.push(root);

root = root->left;

}

}

/\*\* @return whether we have a next smallest number \*/

bool BSTIterator::hasNext() {

return !S.empty();

}

/\*\* @return the next smallest number \*/

int BSTIterator::next() {

TreeNode \*next = S.top();

S.pop();

int ret = next->val;

if (next->right) {

next = next->right;

while (next) {

S.push(next);

next = next->left;

}

}

return ret;

}

1. Nodes at distance k

//didn’t get it

1. Recover Binary search tree

//easy approach using O(N) space

void inorder(TreeNode\* A,vector < int> &v){  
if(A==NULL) {  
return ;  
}  
inorder(A->left,v);  
v.push\_back(A->val);  
inorder(A->right,v);  
}  
vector Solution::recoverTree(TreeNode\* A) {  
vector< int> v,c,d;  
inorder(A,v);  
int n=v.size();  
// cout<<n;  
for(int i=0;i<n;i++){  
c.push\_back(v[i]);  
}  
sort(c.begin(),c.end());  
for(int i=0;i<n;i++){  
if(c[i]!=v[i]){  
d.push\_back(c[i]);  
d.push\_back(v[i]);  
break;  
}  
}  
return d;  
}

//Morris Traversal – Better Approach

1. Flatten binary tree to linked list

TreeNode\* Solution::flatten(TreeNode\* A) {

TreeNode \*ret = A;

while (A) {

if (A->left) {

TreeNode \*temp = A->left;

while (temp->right) {

temp = temp->right;

}

temp->right = A->right;

A->right = A->left;

A->left = NULL;

}

A = A->right;

}

return ret;

}

1. Populate next right pointers //didn’t get

TreeLinkNode \*getptr(TreeLinkNode \*node) {

node = node->next;

while(node) {

if(node->left) return node->left;

else if(node->right) return node->right;

else node = node->next;

}

return nullptr;

}

void helper(TreeLinkNode \*root) {

if(!root) return;

root->next = nullptr;

TreeLinkNode \*curr;

TreeLinkNode \*prev = root;

while(prev) {

curr = prev;

while(curr) {

if(curr->left) {

curr->left->next = curr->right ? curr->right : getptr(curr);

}

if(curr->right) curr->right->next = getptr(curr);

curr = curr->next;

}

if(prev->left) prev = prev->left;

else if(prev->right) prev = prev->right;

else prev = getptr(prev);

}

}

void Solution::connect(TreeLinkNode\* root) {

return helper(root);

}