**L1.**

Full Binary Tree: Every node has either 0 ir 2 children

Complete Binary Tree: All levels are completely filled except the last level and last level has all the nodes as left as possible

Perfect Binary Tree: All leaf nodes are at the same level

Balanced Binary Tree: Height of the tree at max log(N) [N=number of nodes]

Degenerate Tree: Skew trees (like linked list)

**L2. Binary Tree representation in C++:**

struct Node{

int data;

struct Node\* left;

struct Node\* right;

Node(int val){

data= val;

left= right= NULL;

}

}

main(){

struct Node\* root= new Node(1);

root->left= new Node(2);

root->right= new Node(3);

root->left->right= new Node(5);

}

**L4. Traversal Techniques: BFS and DFS**

3 types in DFS: Inorder, Preorder and Postorder

1. Inorder (left-root-right)

2. Preorder (root-left-right)

3. Postorder (left-right-root)

BFS: Level by level

**L5. Preorder Traversal:**

void preOrder(Node\* node){

if(node==NULL)

return;

print(node->data);

preOrder(node->left);

preOrder(node->right);

}

**L6. Inorder Traversal:**

void inOrder(Node\* node){

if(node==NULL)

return;

inOrder(node->left);

print(node->data);

inOrder(node->right);

}

**L7. Postorder Traversal:**

void postOrder(Node\* node){

if(node==NULL)

return;

postOrder(node->left);

postOrder(node->right);

print(node->data);

}

**L8. Levelorder Traversal: (Queue: FIFO):**

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;

}

**L9. Iterative Preorder Traversal: (Stack: LIFO):**

vector<int> preOrder(TreeNode\* root){

vector<int> preorder;

if(root==NULL) return preorder;

stack<TreeNode\*> st;

st.push(root);

while(!st.empty()){

root= st.top();

st.pop();

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

if(root->right!=NULL) st.push(root->right); //right is first

if(root->left!=NULL) st.push(root->left);

}

return preorder;

}

**L10. Iterative Inorder Traversal: (Stack: LIFO):**

vector<int> inOrder(TreeNode\* root){

stack<TreeNode\*> st;

TreeNode\* node= root;

vector<int> inorder;

while(true){

if(node!=NULL){

st.push(node);

node= node->left;

}

else{

if(st.empty()==true) break;

node= st.top();

st.pop();

inorder.push\_back(node->val);

node= node->right;

}

}

return inorder;

}

**L11. Iterative Postorder Traversal using 2 stacks:**

Steps:

1. Initial config: Take the root and put it in the 1st stack.

2. Now, take the top from the 1st stack and put it into the 2nd stack

3. After that, if the top in 2nd stack has left → add it in 1st stack. And if the top in 2nd stack has right → add it in the 1st stack.

4. Now again, take the top from the 1st stack and put it into the 2nd stack. Repeat step 2 & 3 untill 1st stack is empty.

5. Pop the element from the 2nd stack and print.

vector<int> postOrder(TreeNode\* root){

vector<int> postorder;

if(root==NULL) return postorder;

stack<TreeNode\*> st1, st2;

st1.push(root);

while(!st1.empty()){

root= st1.top();

st2.push(root);

if(root->left!=NULL) st1.push(root->left);

if(root->right!=NULL) st1.push(root->right);

}

while(!st2.empty()){

postorder.push\_back(st2.top()->val);

st2.pop();

}

return postorder;

}

**L12. Iterative Postorder Traversal using 1 stack:**

while(cur!=NULL || !st.empty()){

if(cur!=NULL){

st.push(cur);

cur= cur->left;

}

else{

temp= st.top->right;

if(temp==NULL){

temp= st.top();

st.pop();

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

while(!st.empty() && temp==st.top()->right){

temp= st.top();

st.pop();

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

}

}

else{

cur= temp;

}

}

}

**L13. Preorder, Inorder and Postorder Traversalsin 1 Traversal:**

vector<int> preInPostTraversal(TreeNode\* root){

stack<pair<TreeNode\*, int>> st;

st.push({root, 1});

vector<int> pre, in, post;

if(root==NULL) return;

while(!st.empty()){

auto it= st.top();

st.pop();

// this is part of pre

// increment 1 to 2

// push the left side of the tree

if(it.second==1){

pre.push\_back(it.first->val);

it.second++;

st.push(it);

if(it.first->left!=NULL)

st.push({it.first->left, 1});

}

// this is part of in

// increment 2 to 3

// push right

else if(it.second==2){

in.push\_back(it.first->val);

it.second++;

st.push(it);

if(it.first->right!=NULL)

st..push({it.first->right, 1});

}

// don't push it back again

else{

post.push\_back(it.first->val);

}

}

}

**L14. Maximum Depth (Height) in Binary Tree:**

int maxDepth(TreeNode\* root){

if(root==NULL) return 0;

int lh= maxDepth(root->left);

int rh= maxDepth(root->right);

return 1+ max(lh, rh);

}

Levelorder Traversal Approach for Maximum Depth:

int maxDepth(TreeNode\* root){

if(!root) return 0;

queue<TreeNode\*> q;

q.push(root);

int depth= 0;

while(!q.empty()){

int size= q.size();

depth++;

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

TreeNode\* temp= q.front();

q.pop();

if(temp->left!=NULL) q.push(temp->left);

if(temp->right!=NULL) q.push(temp->right);

}

}

return depth;

}

**L15. Check for Balanced Binary Tree:**

Balanced BT: For every node, height(left)-height(right)<=1

Yes=>height of tree, No=>-1

bool isBalanced(TreeNode\* root){

return (check(root)!=-1);

}

int check(TreeNode\* root){

if(root==NULL) return 0;

int lh= check(root->left);

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

int rh= check(root->right);

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

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

return 1+ max(lh, rh);

}

**L16. Diameter of Binary Tree:**

Diameter: Longest path between 2 nodes, and this path does not need to pass via root

int diameterOfBinaryTree(TreeNode\* root){

int diameter= 0;

height(root, diameter);

return diameter;

}

int height(TreeNode\* node, int& diameter){

if(!node) return 0;

int lh= height(node->left, diameter);

int rh= height(node->right, diameter);

diameter= max(diameter, lh+rh);

return 1+ max(lh, rh);

}

// TC= O(n), SC= O(n)s

**L17. Max Path Sum in Binary Tree:**

// On every node, which is the longest path, is what we return

// and use maxi variable to compute the sum of longest path

int maxPathSum(TreeNode\* root){

int maxi= INT\_MIN;

maxPathDown(root, maxi);

return maxi;

}

int maxPathDown(TreeNode\* node, int& maxi){

if(node==NULL) return 0;

int left= max(0, maxPathDown(node->left, maxi));

int right= max(0, maxPathDown(node->right, maxi));

maxi= max(maxi, left+ right+ node->val);

return max(left, right)+ node->val;

}

**L18. Check if 2 trees are identical:**

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);

}

**L19. Zig-Zag/Spiral Traversal:**

vector<vector<int>> zigzagLevelOrder(

vector<vector<int>> result;

if(root==NULL) return result;

queue<TreeNode\*> nodesQueue;

nodesQueue.push(root);

bool leftToRight= true;

while(!nodesQueue.empty()){

int size= nodeQueue.size();

vector<int> row(size);

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

TreeNode\* node= nodesQueue.front();

nodesQueue.pop();

// find position to fill node's value

// true means L->R, and false means R->L

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;

}

**L20. Boundary Traversal in Binary Tree:**

1. Left boundary excluding the leaf

2. Leaf nodes-> Inorder Traversal

3. Right boundary in the reverse, excluding the leaf

bool isLeaf(Node\* root){

if(!root->left && !root->right)

return true;

return false;

}

void addLeftBoundary(Node\* root, vector<int> &res){

Node\* cur= root->left;

while(cur){

if(!isLeaf(cur)) res.push\_back(cur->data);

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

else cur= cur->right;

}

}

void addRightBoundary(Node\* root, vector<int> &res){

Node\* cur= root->right;

vector<int> temp;

while(cur){

if(!isLeaf(cur)) temp.push\_back(cur->data);

if(cur->right) cur= cur->right;

else cur= cur->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;

}

**L21. Vertical Order Traversal of Binary Tree:**

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\* node= p.first;

int x= p.second.first, y= p.second.second;

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

if(node->left)

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

if(node->right)

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

}

vector<vector<int>> ans;

for(auto p: nodes){

vector<int> col;

for(auto a: p.second)

col.insert(col.end(), q.second.begin(), q.second.end());

ans.push\_back(col);

}

return ans;

}

Vertical Traversal using Inorder Traversal:

void inOrder(TreeNode\* root, int x, int level, map<int, map<int, multiset<int>>> &map){

if(root==NULL) return;

inOrder(root->left, x-1, level+1, map);

map[x][level].insert(root->val);

inOrder(root->right, x+1, level+1, map);

}

vector<vector<int>> verticalTraversal(TreeNode\* root){

if(root==NULL) return {};

map<int, map<int, multiset<int>>> map;

inOrder(root, 0, 0, map);

vector<vector<int>> ans;

for(auto it: map){

vector<int> col;

for(auto p: it.second)

col.insert(col.end(), p.second.begin(), p.second.end());

ans.push\_back(col);

}

return ans;

}

// TC= O(n), SC= O(n)

**L22. Top View of Binary Tree:**

vector<int> topView(Node\* root){

vector<int> ans;

if(root==NULL) return ans;

map<int, int> map;

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();

if(map.find(line)==map.end()) map[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: map)

ans.push\_back(it.second);

return ans;

}

// TC= O(n), SC= O(n)

**L23. Bottom View of Binary Tree:**

vector<int> bottomView(Node\* root){

vector<int> ans;

if(root==NULL) return ans;

map<int, int> map;

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;

map[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: map)

ans.push\_back(it.second);

return ans;

}

// TC= O(n), SC= O(n)

**L24. Right/Left View of Binary Tree:**

vector<int> rightSideView(TreeNode\* root){

vector<int> res;

recursion(root, 0, res);

return res;

}

void recursion(TreeNode\* root, int level, vector<int> &res){

if(root==NULL) return;

if(res.size()==level) res.push\_back(root->val);

recursion(root->right, level+1, res);

recursion(root->left, level+1, res);

}

Using Level order Traversal:

vector<int> rightSideView(TreeNode\* root){

vector<int> ans;

if(root==NULL) return ans;

queue<pair<TreeNode\*, int>> q;

map<int, int> map;

q.push({root, 0});

while(!q.empty()){

int size= q.size();

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

TreeNode\* node= q.front();

q.pop();

if(i==size-1) // only insert last element of level in the ans

ans.push\_back(node->val);

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

if(node->right) q.push({node->right, level+1});

}

}

return ans;

}

OR

vector<int> rightSideView(TreeNode\* root){

vector<int> ans;

if(root==NULL) return ans;

queue<pair<TreeNode\*, int>> q;

map<int, int> mpp;

q.push({root, 0});

while(!q.empty()){

auto it= q.front();

q.pop();

TreeNode\* node= it.first;

int level= it.second;

if(mpp.find(level)==mpp.end())

mpp[level]= node->val;

if(node->right) q.push(node->right, level+1);

if(node->left\_ q.push(node->left, level+1);

}

for(auto temp: mpp)

ans.push\_back(temp.second);

return ans;

}

**L25. Check for Symmetrical Binary Trees:**

bool isSymmetric(TreeNode\* root){

return root==NULL || isSymmetricHelp(root->left, root->right);

}

bool isSymmetricHelp(TreeNode\* left, TreeNode\* right){

if(left==NULL || right==NULL)

return left==right;

if(left->val!=right->val) return false;

return isSymmetricHelp(left->left, right->right) && isSymmetricHelp(left->right, right->left);

}

**L26. Print Root to Node Path in Binary Tree:**

bool getPath(TreeNode\* root, vector<int> &arr, int x){

if(!root) return false;

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

if(root->val==x) return true;

if(getPath(root->left, arr, x) || getPath(root->right, arr, x))

return true;

arr.pop\_back();

return false;

}

vector<int> path(TreeNode\* root, int x){

vector<int> arr;

if(root==NULL) return arr;

getPath(root, arr, x);

return arr;

}

**L27. Lowest Common Ancestor in Binary Tree:**

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;

// both left and right are not null, we found our result

else return root;

}

**L28. Maximum Width of Binary Tree:**

Width: Number of nodes in a level, between any 2 nodes

For 0-based indexing, for ith node, left child= 2\*i+1, right child= 2\*i+2

For 1-based indexing, for ith node, left child= 2\*i, right child= 2\*i+1

int widthOfBinaryTree(TreeNode\* root){

if(!root) return 0;

int ans= 0;

queue<pair<TreeNode\*, int>> q;

q.push({root, 0});

while(!q.empty()){

int size= q.size();

int mmin= q.front().second;

int first, last;

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

int cur\_id= q.front().second-mmin;

TreeNode\* node= q.front().first;

q.pop();

if(i==0) first= cur\_id;

if(i==size-1) last= cur\_id;

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

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

}

ans= max(ans, last-first+1);

}

return ans;

}

// TC= O(n), SC= O(n)

**L29. Children Sum Property in Binary Tree:**

Children Sum Property: Any nodes value= left child value+right child value

Self Notes:

If both children sum is less than parent, make children's value to parent's value

If both children values sum is greater than or equal to parent, make parent's value to children's sum

Recursively go left and right. Traversal type: DFS

When coming back up the tree, take children sum and replace it in parent

At any point we reach null, just return (base case)

Intuition: while going down, increase the children values so we make sure to never fall short, then all we have to do is sum both children and replace it in parent

void changeTree(BinaryTreeNode<int>\* root){

if(root==NULL) return;

int child= 0;

if(child->left)

child+= root->left->data;

if(root->right)

child+= root->right->data;

if(child>=root->data)

root->data= child;

else{

if(root->left) root->left->data= root->data;

if(root->right) root->right->data= root->data;

}

reorder(root->left);

reorder(root->right);

int tot= 0;

if(root->left) tot+= root->left->data;

if(root->right) tot+= root->right->data;

if(root->left || root->right) root->data= tot;

}

// TC= O(n), SC= O(h)~O(n)

**L30. Print all the Nodes at a distance of K from any given node in Binary Tree:**

Self Notes:

🍋 Mark each node to its parent to traverse upwards

🍋 We will do a BFS traversal starting from the target node

🍋 As long as we have not seen our node previously, Traverse up, left, right until reached Kth distance

🍋 when reached Kth distance, break out of BFS loop and remaining node's values in our queue is our result

void markParents(TreeNode\* root, unordered\_map<TreeNode\*, TreeNode\*> &parent\_track, TreeNode\* target){

queue<TreeNode\*> queue;

queue.push(root);

while(!queue.empty()){

TreeNode\* current= queue.front();

queue.pop();

if(current->left){

parent\_track[current->left]= current;

queue.push(current->left);

}

if(current->right){

parent\_track[current->right]= current;

queue.push(current->right);

}

}

}

vector<int> distanceK(TreeNode\* root, TreeNode\* target, int k){

unordered\_map<TreeNode\*, TreeNode\*> parent\_track; // node->parent

markParents(root, parent\_track, target);

unordered\_map<TreeNode\*, bool> visited;

queue<TreeNode\*> queue;

queue.push(target);

visited[target]= true;

int cur\_level= 0;

while(!q.empty()){ // Second BFS to go upto K level from target node and using our hashtable info

int size= q.size();

if(cur\_level++ == k) break;

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

TreeNode\* current= queue.front();

queue.pop();

if(current->left && !visited[current->left]){

queue.push(current->left);

visited[current->left]= true;

}

if(current->right && !visited[current->right]){

queue.push(current->right);

visited[current->right)= true;

}

if(parent\_track[current] && !visited[parent\_track[current]]){

queue.push(parent\_track[current]);

visited[parent\_track[current]]= true;

}

}

}

vector<int> result;

while(!queue.empty()){

TreeNode\* current= queue.front();

queue.pop();

result.push\_back(current->val);

}

return result;

}

// TC= O(n), SC= O(n)+O(n)+O(n)= O(n)

**L31. Minimum time taken to BURN the Binary Tree from a Node:**

Self Notes:

🍊 Mark each node to its parent to traverse upwards in a binary tree

🍊 We will do a BFS traversal from our starting node

🍊 Traverse up, left, right until 1 radial level (adjacent nodes) are burned and increment our timer

int findMaxDistance(map<BinaryTreeNode<int>\*, BinaryTreeNode<int>\*> &mpp, BinaryTreeNode<int>\* target){

queue<BinaryTreeNode<int>\*> q;

q.push(target);

map<BinaryTreeNode<int>\*, int> vis;

vis[target]= 1;

int maxi= 0;

while(!q.empty()){

int sz= q.size();

int flag= 0;

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

auto node= q.front();

q.pop();

if(node->left && !vis[node->left]){

flag= 1;

vis[node->left]= 1;

q.push(node->left);

}

if(node->right && !vis[node->right]){

flag= 1;

vis[node->right]= 1;

q.push(node->right);

}

if(mpp[node] && !vis[mpp[node]]){

flag= 1;

vis[mpp[node]]= 1;

q.push(mpp[node]);

}

}

if(flag) maxi++;

}

return maxi;

}

BinaryTreeNode<int>\* bfsToMapParents(BinaryTreeNode<int>\* root, map<BinaryTreeNode<int>\*, BinaryTreeNode<int>\*> &mpp, int start){

queue<BinaryTreeNode<int>\*> q;

q.push(root);

BinaryTreeNode<int>\* res;

while(!q.empty()){

BinaryTreeNode<int>\* node= q.front();

if(node->data==start) res= node;

q.pop();

if(node->left){

mpp[node->left]= node;

q.push(node->left);

}

if(node->right){

mpp[node->right]= node;

q.push(node->right);

}

}

return res;

}

int timeToBurnTree(BinaryTreeNode<int>\* root, int start){

map<BinaryTreeNode<int>\*, BinaryTreeNode<int>\*> mpp;

BinaryTreeNode<int>\* target= bfsToMapParents(root, mpp, start);

int maxi= findMaxDistance(mpp, target);

return maxi;

}

// TC= O(n)+O(n)= O(n), SC= O(n)

**L32. Count total Nodes in a COMPLETE Binary Tree:**

Complete Binary Tree: Every level, except possibly the last, is completely filled in a complete binary tree, and all nodes in the last level are as left as possible. It can have between 1 and 2^h nodes inclusive at the last level h

int countNodes(TreeNode\* root){

if(root==NULL) return 0;

int lh= findHeightLeft(root);

int rh= findHeightRight(root);

if(lh==rh) return (1<<lh)-1; // (2^lh)-1

return 1+countNodes(root->left)+countNodes(root->right);

}

int findHeightLeft(TreeNode\* root){

int height= 0;

while(node){

height++;

node= node->left;

}

return height;

}

int findHeightRight(TreeNode\* root){

int height= 0;

while(node){

height++;

node= node->right;

}

return height;

}

// TC= O(log(n)\*log(n)), SC= log(n)

**L33. Requirements needed to construct a Unique Binary Tree:**

Can you construct a unique binary tree with the following?

Preorder and Postorder- NO

Inorder and Preorder- YES

Inorder and Postorder- YES

**L34. Construct a Binary Tree from Preorder and Inorder Traversal:**

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

map<int, int> inMap;

for(int i=0;i<inorder.size():i++)

inMap[inorder[i]]= i;

TreeNode\* root= buildTree(preorder, 0, preorder.size()-1, inorder, 0, inorder.size()-1, inMap);

return root;

}

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

if(preStart>preEnd || inStart>inEnd)

return NULL;

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

int inRoot= inMap[root->val];

int numsLeft= inRoot-inStart;

root->left= buildTree(preorder, preStart+1, preStart+numsLeft, inorder, inStart, inRoot-1, inMap);

root->right= buildTree(preorder, preStart+numsLeft+1, preEnd, inorder, inRoot+1, inEnd, inMap);

return root;

}

// TC= O(n), SC= O(n)

**L35. Construct a Binary Tree from Postorder and Inorder Traversal:**

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

if(inorder.size()!=postorder.size())

return NULL;

map<int, int> hm;

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

hm[inorder[i]]= i;

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

}

TreeNode\* buildTreePostIn(vector<int> &inorder, int is, int ie, vector<int> &postorder, int ps, int pe, map<int, int> &hm){

if(ps>pe || is>ie)

return NULL;

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

int inRoot= hm[postorder[pe]];

int numsLeft= inRoot-is;

root->left= buildTreePostIn(inorder, is, inRoot-1, postorder, ps, ps+numsLeft-1, hm);

root->right= buildTreePostIn(inorder, inRoot+1, ie, postorder, ps+numsLeft, pe-1, hm);

return root;

}

// TC= O(n), SC= O(n)

**L36. Serialize and De-serialize Binary Tree:**

// Encodes a tree to a single string

string serialize(TreeNode\* root){

if(!root) return "";

string s= "";

queue<TreeNode\*> q;

q.push(root);

while(!q.empty()){

TreeNode\* curNode= q.front();

q.pop();

if(curNode==NULL) s.append("#,");

else s.append(to\_string(curNode->val)+',');

if(curNode!=NULL){

q.push(curNode->left);

q.push(curNode->right);

}

}

cout<<s<<endl;

return s;

}

// TC= O(n), SC= O(n)

// Decodes your encoded data to tree

TreeNode\* deserialize(string data){

if(data.size()==0) return NULL;

stringstream s(data);

string str;

getline(s, str, ',');

TreeNode\* root= new TreeNode(stoi(str));

queue<TreeNode\*> q;

q.push(root);

while(!q.empty()){

TreeNode\* node= q.front();

q.pop();

getline(s, str, ',');

if(str=="#"){

node->left= NULL;

}

else{

TreeNode\* leftNode= new TreeNode(stoi(str));

node->left= leftNode;

q.push(leftNode);

}

getline(s, str, ',');

if(str=="#"){

node->left= NULL;

}

else{

TreeNode\* rightNode= new TreeNode(stoi(str));

node->right= rightNode;

q.push(rightNode);

}

}

return root;

}

// TC= O(n), SC= (n)

**L37. Morris Traversal:**

Self notes:

Inorder Morris Traversal:

🌳 1st case: If left is null, print current node and go right

🌳 2nd case: Before going left, make right most node on left subtree connected to current node, then go left

🌳 3rd case: If thread is already pointed to current node, then remove the thread

vector<int> getInorder(TreeNode\* root){

vector<int> inorder;

TreeNode\* cur= root;

while(cur!=NULL){

if(cur->left==NULL){

inorder.push\_back(cur->val);

cur= cur->right;

}

else{

TreeNode\* prev= cur->left;

while(prev->right && prev->right!=cur){

prev= prev->right;

}

if(prev->right==NULL){

prev->right= cur;

cur= cur->left;

}

else{

prev->right= NULL;

inorder.push\_back(cur->val);

cur= cur->right;

}

}

}

return inorder;

}

// TC~O(n), SC= O(1)

**L38. Flatten a Binary Tree to Linked List (3 Approaches):**

Approach 1:

prev= NULL;

flatten(node){

if(node==NULL)

return;

flatten(node->right);

flatten(node->left);

node->right= prev;

node->left= NULL;

prev= node;

}

// TC= O(n), SC= O(n)

Approach 2:

st.push(root);

while(!st.empty()){

cur= st.top();

st.pop();

if(cur->right)

st.push(cur->right);

if(cur->left)

st.push(cur->left);

if(!st.empty())

cur->right= st.top();

cur->left= NULL;

}

// TC= O(n), SC= O(n)

Approach 3:

cur= root;

while(cur!=NULL){

if(cur->left!=NULL){

prev= cur->left;

while(prev->right)

prev= prev->right;

prev->right= cur->right;

cur->right= cur->left;

cur->left= NULL;

}

cur= cur->right;

}

// TC= O(n), SC= O(1)

**L39. Introduction to Binary Search Tree:**

N

/ \

L R

L<N<R (L<=N<R in case of duplicates)

Left Sub-Tree- BST and Right Sub-Tree- BST

**L40. Search in a Binary Search Tree:**

TreeNode\* searchBST(TreeNode\* root, int val){

while(root!=NULL && root->val!=val){

root= val<root? root->left: root->right;

}

return root;

}

// TC= O(log2(n))

**L41. Ceil in a Binary Search Tree:**

int findCeil(BinaryTreeNode<int>\* root, int key){

int ceil= -1;

while(root){

if(root->data==key){

ceil= root->data;

return ceil;

}

if(key>root->data){

root= root->right;

}

else{

ceil= root->data;

root= root->left;

}

}

return ceil;

}

// TC= O(log2(n))

**L42. Floor in a Binary Search Tree:**

int floorBST(TreeNode<int>\* root, int key){

int floor= -1;

while(root){

if(root->val==key){

floor= root->val;

return floor;

}

if(key>root->val){

floor= root->val;

root= root->right;

}

else{

root= root->left;

}

}

return floor;

}

// TC= O(log2(n))

**L43. Insert a given Node in Binary Search Tree:**

TreeNode\* insertIntoBST(TreeNode\* root, int val){

if(root==NULL)

return new TreeNode(val);

TreeNode\* cur= root;

while(true){

if(cur->val<=val){

if(cur->right!=NULL){

cur= cur->right;

}

else{

cur->right= new TreeNode(val)

break;

}

}

else{

if(cur->left!=NULL){

cur= cur->left;

}

else{

cur->left= new TreeNode(val);

break;

}

}

}

return root;

}

// TC= O(log2(n))

**L44. Delete a Node in Binary Search Tree:**

TreeNode\* deleteNode(TreeNode\* root, int key){

if(root==NULL) return NULL;

if(root->val==key)

return helper(root);

TreeNode\* dummy= root;

while(root!=NULL){

if(root->val>key){

if(root->left!=NULL && root->left->val==key){

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

break;

}

else{

root= root->left;

}

}

else{

if(root->right!=NULL && root->right->val==key){

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

break;

}

else{

root= root->right;

}

}

}

return dummy;

}

TreeNode\* helper(TreeNode\* root){

if(root->left==NULL)

return root->right;

else if(root->right==NULL)

return root->left;

TreeNode\* rightChild= root->right;

TreeNode\* lastRight= findLastRight(root->left);

lastRight->right= rightChild;

return root->left;

}

TreeNode\* findLastRight(TreeNode\* root){

if(root->right==NULL)

return root;

return findLastRight(root->right);

}

// TC= O(log2(n))

**L45. Kth Smallest/Largest Element in BST:**

Kth Largest:

TreeNode\* kthLargest(TreeNode\* root, int &k){

if(root==NULL)

return NULL;

TreeNode\* right= kthLargest(root->right, k);

if(right!=NULL)

return right;

k--;

if(k==0)

return root;

return kthLargest(root->left, k);

}

Kth Smallest:

TreeNode\* kthSmallest(TreeNode\* root, int &k){

if(root==NULL)

return NULL;

TreeNode\* left= kthSmallest(root->left, k);

if(left!=NULL)

return left;

k--;

if(k==0)

return root;

return kthSmallest(root->right, k);

}

//TC= O(min(k, n)), SC= O(min(k, n))

**L46. Check if a Tree is a BST or BT:**

bool isValidBST(TreeNode\* root){

return isValidBST(root, INT\_MIN, INT\_MAX);

}

bool isValidBST(TreeNode\* root, int minVal, int maxVal){

if(root==NULL) return true;

if(root->val>=maxVal || root->val<=minVal) return false;

return isValidBST(root->left, minVal, root->val) && isValidBST(root->right, root->val, maxVal);

}

//TC= O(n), SC= (1)

**L47. LCA in Binary Search Tree:**

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

if(root==NULL) return NULL;

int cur= root->val;

if(cur<p->val && cur<q->val)

return lowestCommonAncestor(root->right, p, q);

if(cur>p->val && cur>q->val)

return lowestCommonAncestor(root->left, p, q);

return root;

}

// TC= O(log2(n)) (O(h)), SC= O(1) (Auxillary Space= O(n))

**L48. Construct a BST from a Preorder Traversal:**

TreeNode\* bstFromPreorder(vector<int> &A){

int i=0;

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

}

TreeNode\* build(vector<int> &A, int &i, int bound){ // bound is the upper 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;

}

// TC= O(n), SC= O(n)

**L49. Inorder Successor/Predecessor in BST:**

TreeNode\* inorderSuccessor(TreeNode\* root, TreeNode\* p){

TreeNode\* successor= NULL;

while(root!=NULL){

if(p->val>=root->val){

root= root->right;

}

else{

successor= root;

root= root->left;

}

}

return successor;

}

TreeNode\* inorderPredecessor (TreeNode\* root, TreeNode\* p){

TreeNode\* predecessor= NULL;

while(root!=NULL){

if(root->val>=p->val){

root= root->left;

}

else{

predecessor= root;

root= root->right;

}

}

return predecessor;

}

// TC= O(log2(n)) (O(h)), SC= O(1)

**L50. Binary Search Tree Iterator:**

// Definition for a binary tree node

struct TreeNode{

int val;

TreeNode\* left;

TreeNode\* right;

TreeNode(): val(0), left(nullptr), right(nullptr) {}

TreeNode(int x): val(x), left(nullptr), right(nullptr) {}

TreeNode(int x, TreeNode\* left, TreeNode\* right): val(x), left(left), right(right) {}

};

class BSTIterator{

private:

stack<TreeNode\*> myStack;

public:

BSTIterator(TreeNode\* root){

pushAll(root);

}

// return whether we have a next smallest number

bool hasNext(){

return !myStack.empty();

}

// return the next smallest number

int next(){

TreeNode\* tmpNode= myStack.top();

myStack.pop();

pushAll(tempNode->right);

return tempNode->val;

}

private:

void pushAll(TreeNode\* node){

for( ; node!=NULL; myStach.push(node), node= node-<left);

}

};

// TC= O(1), SC= O(log2(n)) (O(h))

**L51. Two Sum in BST (Check if there exists a pair with sum K):**

class BSTIterator{

stack<TreeNode\*> myStack;

// reverse= true=> before

// reverse= false=> next

bool reverse= true;

public:

BSTIterator(TreeNode\* root, bool isReverse){

reverse= isReverse;

pushAll(root);

}

// return whether we have a next smallest number

bool hasNext(){

return !myStack.empty();

}

// return the next smallest number

int next(){

TreeNode\* tmpNode= myStack.top();

myStack.pop();

if(!reverse) pushAll(tmpNode->right);

else pushAll(tmpNode->left);

return tmpNode->val;

}

private:

void pushAll(TreeNode\* node){

for( ; node!=NULL; ){

myStack.push(node);

if(reverse==true)

node= node->right;

else

node= node->left;

}

}

};

bool findTarget(TreeNode\* root, int k){

if(!root) return false;

// for next

BSTIterator l(root, false);

// for before

BSTIterator r(root, true);

int i= l.next();

int j= r.next();

while(i<j){

if(i+j==k) return true;

else if(i+j<k) i= l.next();

else j= r.next();

}

return false;

}

// TC= O(n), SC= O(log2(n)) (O(h))

**L52. Recover BST (Correct BST with two nodes swapped):**

class Solution{

private:

TreeNode\* first;

TreeNode\* prev;

TreeNode\* middle;

TreeNode\* last;

private:

void inorder(TreeNode\* root){

if(root==NULL) return;

inorder(root->left);

if(prev!=NULL && (root->val<prev->val)){

// If this is the 1st violation, mark these two nodes as 'first' and 'middle'

if(first==NULL){

first= prev;

middle= root;

}

// If this is the 2nd violation, mark this node as last

else

last= root;

}

// Mark this node as previous

prev= root;

inorder(root->right);

}

public:

void recoverTree(TreeNode\* root){

first= middle= last= NULL;

prev= new TreeNode(INT\_MIN);

inorder(root);

if(first && last) swap(first->val, last->val);

else if(first && middle) swap(first->val, middle->val);

}

};

// TC= O(n), SC= O(1)

**L53. Largest BST in Binary Tree:**

class NodeValue{

public:

int maxNode, minNode, maxSize;

NodeValue(int minNode, int maxNode, int maxSize){

this->maxNode= maxNode;

this->minNode= minNode;

this->maxSize= maxSize;

}

};

NodeValue largestBSTSubtreeHelper(TreeNode\* root){

// An empty tree is a BST of size 0

if(!root)

return NodeValue(INT\_MAX, INT\_MIN, 0);

// Get values from left and right subtree of current tree

auto left= largestBSTSubtreeHelper(root->left);

auto right- largestBSTSubtreeHelper(root->right);

// Current node is greater than max in left and smaller than min in right

if(left.maxNode<root->val && root->val<right.minNode){

// It is a BST

return NodeValue(min(root->val, left.minNode), max(root->val, right.maxNode, left.maxSize+right.maxSize+1);

}

// Otherwise, return [-inf, inf] so that parent can't be in valid BST

return NodeValue(INT\_MIN, INT\_MAX, max(left.maxSize, right.maxSize));

}

int largestBSTSubtree(TreeNode\* root){

return largestBSTSubtreeHelper(root).maxSize;

}

// TC= O(n), SC= O(1)