Tree Easy Questions

# **94. Binary Tree Inorder Traversal**

In Order Traversal of Tree :

1. First visits the left sub tree
2. Visit the current node
3. At last visit Right Subtree.

class Solution {

    vector<int> ans;

    void recFunc(TreeNode \*node) {

        if(!node) return;

        recFunc(node->left);

        ans.push\_back(node->val);

        recFunc(node->right);

    }

public:

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

        recFunc(root);

        return ans;

    }

};

# **100. Same Tree**

* Check if both node is null then return true
* If any node is not null, then return false
* If value are not matching return false.
* Check for left and right sub-tree, if both return true then the both trees are same else not.

class Solution {

public:

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

if(!p && !q) return true;

else if(!p || !q) return false;

return

(p->val == q->val)

&& isSameTree(p->left, q->left)

&& isSameTree(p->right, q->right)

;

}

};

# **101. Symmetric Tree**



Consider left subtree and right subtree of root node is identical or not.

class Solution {

bool recFunc(TreeNode \*r1, TreeNode \*r2){

if(!r1 && !r2) return true;

else if((!r1 || !r2) || (r1->val != r2->val)) return false;

return recFunc(r1->left, r2->right) && recFunc(r1->right, r2->left);

}

public:

bool isSymmetric(TreeNode\* root) {

return recFunc(root->left, root->right);

}

};

# **104. Maximum Depth of Binary Tree**

* If tree is empty, then height is 0.
* If it has only one node then height or depth is 1.
* Max depth of tree is 1 + Max(depth of left tree and depth of right tree).

class Solution {

public:

int maxDepth(TreeNode\* root) {

if(!root) return 0;

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

}

};

# **108. Convert Sorted Array to Binary Search Tree**

* Get middle element of index and create root node.
* Create left subtree from the left array, assign it to root->left.
* Create right subtree from the right array and assign that to root’s right pointer.

class Solution {

TreeNode \*recFunc(vector<int> nums, int l, int r){

if(l>r) return NULL;

int ind = l + ((r-l) / 2);

TreeNode \*ptr = new TreeNode(nums[ind]);

ptr->left = recFunc(nums,l,ind-1);

ptr->right = recFunc(nums, ind+1, r);

return ptr;

}

public:

TreeNode\* sortedArrayToBST(vector<int>& nums) {

return recFunc(nums, 0, nums.size()-1);

}

};

# **110. Balanced Binary Tree**

A height-balanced binary tree is a binary tree in which the depth of the two subtrees of every node never differs by more than one.

class Solution {

bool balanced = true;

int getHeight(TreeNode \*node) {

if(!node) return 0;

int lh = getHeight(node->left);

int rh = getHeight(node->right);

if(abs(lh - rh) > 1) balanced = false;

return (1 + max(lh, rh));

}

public:

bool isBalanced(TreeNode\* root) {

getHeight(root);

return balanced;

}

};

# **111. Minimum Depth of Binary Tree**

class Solution {

public:

int minDepth(TreeNode\* root) {

if(!root) return 0;

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

return 1 + (lh == 0 ? rh : rh == 0 ? lh : min(lh,rh));

}

};

# **targetSum**



class Solution {

bool recFunc(TreeNode\* root, int targetSum){

if(!root) return false;

targetSum -= root->val;

if(!root->left && !root->right && targetSum == 0) return true;

return

recFunc(root->left, targetSum) || recFunc(root->right, targetSum);

}

public:

bool hasPathSum(TreeNode\* root, int targetSum) {

return recFunc(root, targetSum);

}

};

# **Binary Tree Preorder Traversal**

class Solution {

vector<int> ans;

void recFunc(TreeNode \*node){

if(!node) return;

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

recFunc(node->left);

recFunc(node->right);

}

public:

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

recFunc(root);

return ans;

}

};

# **Binary Tree Postorder Traversal**

class Solution {

vector<int> ans;

void recFunc(TreeNode \*root) {

if(!root) return ;

recFunc(root->left);

recFunc(root->right);

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

}

public:

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

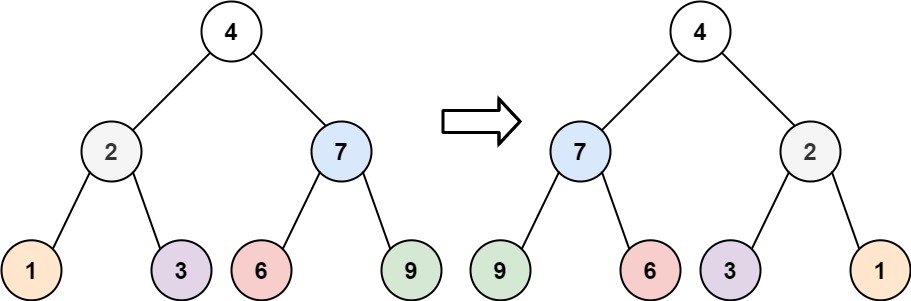
recFunc(root);

return ans;

}

};

# **Invert Binary Tree**



class Solution {

public:

TreeNode\* invertTree(TreeNode\* root) {

if(!root) return NULL;

TreeNode \*l = root->left, \*r = root->right;

invertTree(l);

invertTree(r);

root->left = r;

root->right = l;

return root;

}

};

# **Binary Tree Paths**

class Solution {

vector<string> ans;

void recFunc(TreeNode \*node, string v){

if(!node) return;

v = v+to\_string(node->val);

if(!node->left && !node->right) {

ans.push\_back(v);

return;

}

recFunc(node->left , v + "->");

recFunc(node->right, v + "->");

}

public:

vector<string> binaryTreePaths(TreeNode\* root) {

string v = "";

recFunc(root, v);

return ans;

}

};

# 404. Sum of Left Leaves

class Solution {

int recFunc(TreeNode \*node, bool isLeftTree) {

if(!node) return 0;

else if(!node->left && !node->right)

return isLeftTree ? node->val : 0;

return recFunc(node->left, true) + recFunc(node->right, false);

}

public:

int sumOfLeftLeaves(TreeNode\* root) {

return recFunc(root, false);

}

};

# [501. Find Mode in Binary Search Tree](https://leetcode.com/problems/find-mode-in-binary-search-tree/description/)

class Solution {

void recFunc(TreeNode \*root, map<int,int> &mp, int &m) {

if(!root) return;

mp[root->val] += 1;

m = max(m, mp[root->val]);

recFunc(root->left, mp, m);

recFunc(root->right, mp, m);

}

public:

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

map<int,int> mp;

int m = 0;

recFunc(root, mp, m);

vector<int> ans;

for(auto it : mp) {

if(it.second == m) ans.push\_back(it.first);

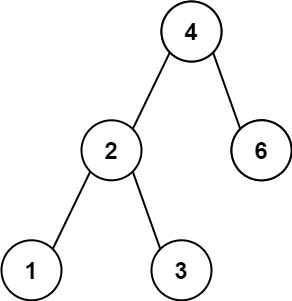
}

return ans;

}

};

# 530. Minimum Absolute Difference in BST



Approach

* Append each element in array then sort and get the diff of neighbour elements.
* Preorder will always return sorted array.
  + So don’t need to sort using O(n logn) time.
  + Extra space will be O(n) to store preorder traversal, but we can avoid using prevVal variable.

class Solution {

public:

int getMinimumDifference(TreeNode\* root) {

stack<TreeNode \*> st;

TreeNode \*curr = root;

int prevNode = -1, ans = INT\_MAX;

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

if(curr) {

st.push(curr);

curr = curr->left;

continue;

}

curr = st.top();

st.pop();

if(prevNode == -1) prevNode = curr->val;

else {

ans = min(ans, curr->val - prevNode);

prevNode = curr->val;

}

curr = curr->right;

}

return ans;

}

};

# **543. Diameter of Binary Tree**

class Solution {

int ans = 0;

public:

int recFunc(TreeNode\* root) {

if(!root) return 0;

int lh = recFunc(root->left);

int rh = recFunc(root->right);

if(lh+rh > ans) ans = lh + rh;

return 1 + max(lh, rh);

}

int diameterOfBinaryTree(TreeNode\* root) {

recFunc(root);

return ans;

}

};

# Evaluate Boolean Binary Tree

A full binary tree is a binary tree where each node has either 0 or 2 children.

A leaf node is a node that has zero children.

class Solution {

public:

bool evaluateTree(TreeNode\* root) {

if(!root->left && !root->right) return root->val;

int lv = evaluateTree(root->left);

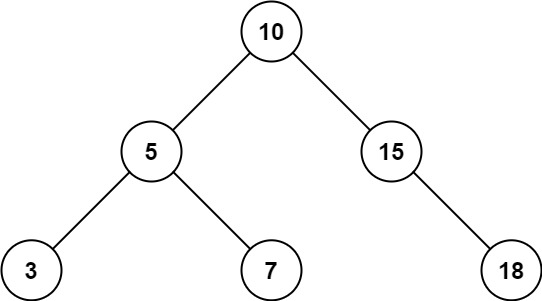
int rv = evaluateTree(root->right);

return root->val == 2 ? lv | rv : lv & rv;

}

};

# Range Sum of BST



class Solution {

public:

int rangeSumBST(TreeNode\* root, int low, int high) {

if(!root) return 0;

int ans = 0;

if(root->val >= low && root->val <= high) ans += root->val;

if(root->val > low) ans += rangeSumBST(root->left, low, high);

if(root->val < high) ans += rangeSumBST(root->right, low, high);

return ans;

}

};

# Search in a Binary Search Tree

class Solution {

public:

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

if(!root) return root;

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

if(root->val > val) return searchBST(root->left, val);

return searchBST(root->right, val);

}

};

# Find a Corresponding Node of a Binary Tree in a Clone of That Tree

class Solution {

public:

TreeNode\* getTargetCopy(TreeNode\* original, TreeNode\* cloned, TreeNode\* target) {

if(!original || !cloned) return NULL;

if(cloned->val == target->val) return cloned;

TreeNode \*ln = getTargetCopy(original->left, cloned->left, target);

if(ln) return ln;

return getTargetCopy(original->right, cloned->right, target);

}

};