***Week – 7 (31.05.2021 – 05.06.2021)***

***CODES IN PDF***

1. ***Binary Tree Inorder Traversal:***

class Solution {

public:

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

if (root == NULL) {

return;

}

inorder(root->left, res);

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

inorder(root->right, res);

}

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

vector<int> res;

inorder(root,res);

return res;

}

};

1. ***Maximum Depth of Binary Tree:***

class Solution {

public:

int heigth(TreeNode\* root){

if(root==NULL) return 0;

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

}

int maxDepth(TreeNode\* root) {

if(root == NULL) return 0;

return heigth(root);

}

};

1. ***Binary Tree Preorder Traversal:***

class Solution {

public:

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

if (root == NULL) {

return;

}

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

preorder(root->left, res);

preorder(root->right, res);

}

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

vector<int> res;

preorder(root,res);

return res;

}

};

1. ***Binary Tree Postorder Traversal:***

class Solution {

public:

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

if (root == NULL) {

return;

}

postorder(root->left, res);

postorder(root->right, res);

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

}

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

vector<int> res;

postorder(root,res);

return res;

}

};

1. ***Diameter of Binary Tree:***

class Solution {

public:

int maxdepth(TreeNode\* root){

return root==NULL?0:max(maxdepth(root->left),maxdepth(root->right))+1;

}

void traverse(TreeNode\* root, int& mx){

if(root == NULL) return;

if(root->left && root->right) mx = max(mx,maxdepth(root->left)+maxdepth(root->right));

if(root->left && !root->right) mx = max(mx,maxdepth(root->left));

if(!root->left && root->right) mx = max(mx,maxdepth(root->right));

traverse(root->left, mx);

traverse(root->right, mx);

}

int diameterOfBinaryTree(TreeNode\* root) {

int mx = 0;

traverse(root, mx);

return mx;

}

};

1. ***Same Tree:***

class Solution {

public:

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

if(p==NULL && q==NULL) return true;

if(p==NULL || q==NULL) return false;

return (p->val==q->val && isSameTree(p->left,q->left) && isSameTree(p->right,q->right));

}

};

1. ***Binary Tree Level Order Traversal:***

class Solution {

public:

int height(TreeNode\* root){

return root==NULL?0:max(height(root->left),height(root->right))+1;

}

void currlevel(TreeNode\* root, int level, vector<int> &curr){

if(root == NULL) return;

if(level == 1) curr.push\_back(root->val);

if(level > 1)

{

currlevel(root->left, level-1, curr);

currlevel(root->right, level-1, curr);

}

}

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

vector<int> curr;

vector<vector<int>> res;

int i, h;

h = height(root);

for(i=1; i<=h; i++)

{

currlevel(root, i, curr);

res.push\_back(curr);

curr = {};

}

return res;

}

};

1. ***Binary Tree Level Order Traversal II:***

class Solution {

public:

int height(TreeNode\* root){

return root==NULL?0:max(height(root->left),height(root->right))+1;

}

void currlevel(TreeNode\* root, int level, vector<int> &curr){

if(root == NULL) return;

if(level == 1) curr.push\_back(root->val);

if(level > 1)

{

currlevel(root->left, level-1, curr);

currlevel(root->right, level-1, curr);

}

}

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

vector<int> curr;

vector<vector<int>> res;

int i, h;

h = height(root);

for(i=1; i<=h; i++)

{

currlevel(root, i, curr);

res.push\_back(curr);

curr = {};

}

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

return res;

}

};

1. Kth Smallest Element in a BST:

class Solution {

public:

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

if (root == NULL) {

return;

}

inorder(root->left, res);

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

inorder(root->right, res);

}

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

vector<int> res;

inorder(root,res);

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

return res[k-1];

}

};

1. ***Lowest Common Ancestor of a Binary Search Tree:***

class Solution {

public:

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

if(!root) return nullptr;

if(root->val <= max(p->val,q->val) && root->val >= min(p->val,q->val)) return root;

else if(root->val > max(p->val,q->val)) return lowestCommonAncestor(root->left, p, q);

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

}

};

1. ***Sorted Array To Balanced BST:***

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

if(A.empty())

return NULL;

unsigned int mid = A.size()/2;

vector<int> leftArr(A.begin(), A.begin()+mid), rightArr(A.begin()+mid+1, A.end());

TreeNode\* head = new TreeNode(A[mid]);

head->left = sortedArrayToBST(leftArr);

head->right = sortedArrayToBST(rightArr);

return head;

}

1. ***Merge Two Binary Trees:***

class Solution {

public:

TreeNode\* mergeTrees(TreeNode\* root1, TreeNode\* root2) {

if(!root1) return root2;

if(!root2) return root1;

root1->val = root1->val + root2->val;

root1->left = mergeTrees(root1->left, root2->left);

root1->right = mergeTrees(root1->right, root2->right);

return root1;

}

};

1. ***Binary Tree Zigzag Level Order Traversal:***

class Solution {

public:

int height(TreeNode\* root){

return root==NULL?0:max(height(root->left),height(root->right))+1;

}

void currlevel(TreeNode\* root, int level, vector<int> &curr){

if(root == NULL) return;

if(level == 1) curr.push\_back(root->val);

if(level > 1)

{

currlevel(root->left, level-1, curr);

currlevel(root->right, level-1, curr);

}

}

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

vector<int> curr;

vector<vector<int>> res;

int i, h;

h = height(root);

for(i=1; i<=h; i++)

{

currlevel(root, i, curr);

if(i%2 == 0) reverse(curr.begin(), curr.end());

res.push\_back(curr);

curr = {};

}

return res;

}

};

1. ***Flatten Binary Tree to Linked List:***

class Solution {

public:

TreeNode\* head;

void flatten(TreeNode\* root) {

if(!root) return;

if(root->right) flatten(root->right);

if(root->left) flatten(root->left);

root->right = head;

root->left = nullptr;

head = root;

}

};

1. ***Construct Binary Tree from Preorder and Inorder Traversal:***

class Solution {

public:

int preind = 0;

map<int,int> m;

TreeNode\* createTree(vector<int>& preorder, int l, int r)

{

if(l > r) return NULL;

int rootval = preorder[preind++];

TreeNode\* root = new TreeNode(rootval);

root -> left = createTree(preorder, l, m[rootval] - 1);

root -> right = createTree(preorder, m[rootval] + 1, r);

return root;

}

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

int i;

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

m[inorder[i]] = i;

return createTree(preorder, 0 , preorder.size() - 1);

}

};

1. ***Construct Binary Tree from Preorder and Postorder Traversal:***

class Solution {

public:

TreeNode\* construct(vector<int>& pre, vector<int>& post, int i, int j, int m, int n)

{

if(i>=j) return nullptr;

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

int k = 0;

while(i+1+k < j)

{

if(pre[i + 1] == post[m + k])

{

k = k + 1;

break;

}

k++;

}

root->left = construct(pre, post, i + 1, i + 1 + k, m, m + k);

root->right = construct(pre, post, i + 1 + k, j, m + k, n - 1);

return root;

}

TreeNode\* constructFromPrePost(vector<int>& pre, vector<int>& post) {

return construct(pre, post, 0, pre.size(), 0, post.size());

}

};

1. ***Binary Tree Paths:***

class Solution {

public:

vector<string> res;

void build(TreeNode\* curr, string path)

{

if(!curr) return;

if(curr->left == curr->right)

{

path += to\_string(curr->val);

res.push\_back(path);

}

else

{

path += to\_string(curr->val)+"->";

build(curr->left, path);

build(curr->right, path);

}

}

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

string path;

build(root, path);

return res;

}

};

1. ***Right view of Binary tree:***

int height(TreeNode\* root){

return root==NULL?0:max(height(root->left),height(root->right))+1;

}

void currlevel(TreeNode\* root, int level, vector<int> &curr){

if(root == NULL) return;

if(level == 1) curr.push\_back(root->val);

if(level > 1)

{

currlevel(root->left, level-1, curr);

currlevel(root->right, level-1, curr);

}

}

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

vector<int> curr;

vector<int> res;

int i, h;

h = height(A);

for(i=1; i<=h; i++)

{

currlevel(A, i, curr);

res.push\_back(curr[curr.size()-1]);

curr = {};

}

return res;

}

1. ***Second Minimum Node In a Binary Tree:***

class Solution {

public:

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

if (root == NULL) {

return;

}

inorder(root->left, res);

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

inorder(root->right, res);

}

int findSecondMinimumValue(TreeNode\* root) {

vector<int> res;

inorder(root,res);

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

res.erase(unique(res.begin(), res.end()), res.end());

if(res.size() < 2) return -1;

else return res[1];

}

};

1. ***Validate Binary Search Tree:***

class Solution {

public:

bool BST(TreeNode\* root, TreeNode\* minNode, TreeNode\* maxNode) {

if(!root) return true;

if(minNode && root->val <= minNode->val || maxNode && root->val >= maxNode->val) return false;

return BST(root->left, minNode, root) && BST(root->right, root, maxNode);

}

bool isValidBST(TreeNode\* root){

return BST(root, NULL, NULL);

}

};

1. ***Balanced Binary Tree:***

class Solution {

public:

int calcHeight(TreeNode\* root) {

if(!root) return 0;

int lheight = calcHeight(root->left);

int rheight = calcHeight(root->right);

if (lheight == -1 || rheight == -1 || abs(lheight - rheight) > 1) return -1;

return max(lheight, rheight) + 1;

}

bool isBalanced(TreeNode\* root) {

if(!root) return true;

return calcHeight(root) != -1;

}

};

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

class Solution {

public:

unsigned long long int res;

unordered\_map<int, unsigned long long int> mapp;

void func(TreeNode \*node, unsigned long long int idx, int level){

if(!node) return;

func(node->left, 2\*idx, level+1);

func(node->right, 2\*idx+1, level+1);

if(mapp[level] == 0) mapp[level] = idx;

else mapp[level] = min(mapp[level], idx);

res = max(res, idx-mapp[level]+1);

}

int widthOfBinaryTree(TreeNode\* root) {

res=0;

func(root, 1, 0);

return res;

}

};

1. ***Check Completeness of a Binary Tree:***

class Solution {

public:

bool isCompleteTree(TreeNode\* root) {

if(root==NULL) return true;

queue<TreeNode\*>q;

q.push(root);

bool flag=false;

while(!q.empty())

{

TreeNode\* temp=q.front();

q.pop();

if(temp->left)

{

if(flag==true) return false;

q.push(temp->left);

}

else flag=true;

if(temp->right)

{

if(flag==true) return false;

q.push(temp->right);

}

else flag=true;

}

return true;

}

};