**Working with Binary search tree**

#include <iostream>

#include <vector>

#include <climits>

#include <algorithm>

#include <queue> // To use the queue for level-order traversal

using namespace std;

// Structure to represent a Node in the tree

struct Node {

int data;

Node\* left;

Node\* right;

Node(int value) {

data = value;

left = nullptr;

right = nullptr;

}

};

// Function to check if the tree is a valid Binary Search Tree

bool isBST(Node\* root, int minVal = INT\_MIN, int maxVal = INT\_MAX) {

if (root == nullptr) return true;

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

return isBST(root->left, minVal, root->data) && isBST(root->right, root->data, maxVal);

}

// Function to do an in-order traversal and store the elements

void inorderTraversal(Node\* root, vector<int>& nodes) {

if (root == nullptr) return;

inorderTraversal(root->left, nodes);

nodes.push\_back(root->data);

inorderTraversal(root->right, nodes);

}

// Helper function to build a balanced BST from sorted nodes

Node\* sortedArrayToBST(vector<int>& nodes, int start, int end) {

if (start > end) return nullptr;

int mid = (start + end) / 2;

Node\* node = new Node(nodes[mid]);

node->left = sortedArrayToBST(nodes, start, mid - 1);

node->right = sortedArrayToBST(nodes, mid + 1, end);

return node;

}

// Function to fix the tree if it's not a BST by making it a valid BST

Node\* makeBST(Node\* root) {

vector<int> nodes;

inorderTraversal(root, nodes); // Store nodes in inorder

sort(nodes.begin(), nodes.end()); // Sort the nodes to satisfy BST property

return sortedArrayToBST(nodes, 0, nodes.size() - 1); // Rebuild the tree

}

// Function to insert nodes in the binary tree (level-order insertion)

Node\* insertNode(Node\* root, string data) {

if (data == "null") {

return nullptr;

}

int value = stoi(data); // Convert string input to integer

if (root == nullptr) {

return new Node(value); // If the tree is empty, create the root node

}

// Use a queue for level-order traversal

queue<Node\*> q;

q.push(root); // Start with the root node

while (!q.empty()) {

Node\* current = q.front(); // Get the node at the front of the queue

q.pop();

// If left child is empty, insert the new node there

if (current->left == nullptr) {

current->left = new Node(value);

return root;

} else {

q.push(current->left); // Else, add left child to the queue

}

// If right child is empty, insert the new node there

if (current->right == nullptr) {

current->right = new Node(value);

return root;

} else {

q.push(current->right); // Else, add right child to the queue

}

}

return root; // This return statement is just for safety; the function will always return inside the loop.

}

// Function to print the tree in pre-order

void printTree(Node\* root) {

if (root == nullptr) {

cout << "null ";

return;

}

cout << root->data << " ";

printTree(root->left);

printTree(root->right);

}

int main() {

Node\* root = nullptr;

int n;

string data;

// Input number of nodes

cout << "Enter the number of nodes in the tree: ";

cin >> n;

// Input tree nodes

cout << "Enter the nodes of the tree (use 'null' for no node): ";

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

cin >> data;

root = insertNode(root, data); // Insert node in the tree

}

// Check if it's a valid BST

if (isBST(root)) {

cout << "Yes, it is a valid Binary Search Tree!" << endl;

} else {

cout << "No, it's not a valid Binary Search Tree!" << endl;

root = makeBST(root); // Fix the tree to make it a BST

cout << "Tree has been fixed into a valid Binary Search Tree!" << endl;

}

// Print the values of the tree (pre-order traversal)

cout << "Values of the tree: ";

printTree(root);

cout << endl;

return 0;

}  
  
  
  
**Views in Tree**

**Top View:**

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

vector <int> ans;

If (root==NULL) return ans;

map<int,int> mpp;// map has line,node  
queue <pair<Node\* , int> > q; //q has node,line

q.push({root,0});

while (!q.empty()){

f=q.front(); //node,line[0,1]

q.pop();

Node\* node=f.first;//0

Int line =f.second;//1

If mpp.find(line)==map.end())

Mpp[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 i:mpp){

ans.push\_back(i.second);

}

Return ans;

}

**Bottom view**

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

vector <int> ans;

If (root==NULL) return ans;

map<int,int> mpp;// map has line,node

queue <pair<Node\* , int> > q; //q has node,line

q.push({root,0});

while (!q.empty()){

f=q.front(); //node,line[0,1]

q.pop();

Node\* node=f.first;//0

Int line =f.second;//1

~~(If mpp.find(line)==map.end())~~

~~Mpp[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 i:mpp){

ans.push\_back(i.second);

}

Return ans;

}

**Right View(recursive approach is simpler than iterative) reverse preorder traversal(root->right->left)**

Class solution {

Vector <int> rv(TreeNode\* root){

Vector <int> res;

Recur(root,0,res);

Return res;

}

Public:

Void recur(TreeNode\* root, int level, vector<int> res){

If (root==NULL) return ;

If (res.size()==level); //if the array size equals the level I.e, the elt occurs first time  
res.push\_back(root->val);

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

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

}};

**Left View(recursive approach is simpler than iterative) preorder traversal(root->left->right)**

Class solution {

Vector <int> lv(TreeNode\* root){

Vector <int> res;

Recur(root,0,res);

Return res;

}

Public:

Void recur(TreeNode\* root, int level, vector<int> res){

If (root==NULL) return ;

If (res.size()==level); //if the array size equals the level I.e, the elt occurs first time

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

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

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

}};