**Coding practice Problems 25-11-2024**

**1. Validate if given Tree is BST or not**

#include <iostream>

#include <vector>

#include <set>

using namespace std;

struct Node {

int val;

Node\* left;

Node\* right;

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

};

class Solution {

public:

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

if (!root) return;

inorder\_traverse(root->left, nodes);

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

inorder\_traverse(root->right, nodes);

}

bool isValidBST(Node\* root) {

if (!root) return true;

vector<int> nodes;

inorder\_traverse(root, nodes);

set<int> s(nodes.begin(), nodes.end());

if (s.size() != nodes.size()) return false;

for (int i = 0; i < nodes.size() - 1; i++) {

if (nodes[i] >= nodes[i + 1]) return false;

}

return true;

}

};

int main() {

Node\* root = new Node(2);

root->left = new Node(1);

root->right = new Node(3);

Solution solution;

cout << (solution.isValidBST(root) ? "True" : "False") << endl;

Node\* invalidRoot = new Node(5);

invalidRoot->left = new Node(1);

invalidRoot->right = new Node(4);

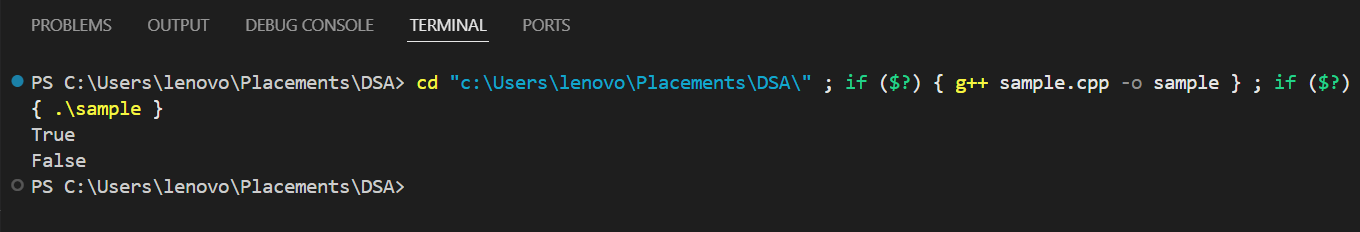
invalidRoot->right->left = new Node(3);

invalidRoot->right->right = new Node(6);

cout << (solution.isValidBST(invalidRoot) ? "True" : "False") << endl;

return 0;

}



TIME COMPLEXITY: O(n log n)  
SPACE COMPLEXITY: O(n)

**2. Convert to BST**

#include <iostream>

#include <vector>

#include <set>

using namespace std;

struct Node {

int val;

Node\* left;

Node\* right;

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

};

class Solution {

public:

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

if (!root) return;

inorder\_traverse(root->left, nodes);

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

inorder\_traverse(root->right, nodes);

}

bool isValidBST(Node\* root) {

if (!root) return true;

vector<int> nodes;

inorder\_traverse(root, nodes);

set<int> s(nodes.begin(), nodes.end());

if (s.size() != nodes.size()) return false;

for (int i = 0; i < nodes.size() - 1; i++) {

if (nodes[i] >= nodes[i + 1]) return false;

}

return true;

}

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

if (start > end) return nullptr;

int mid = (start + end) / 2;

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

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

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

return root;

}

Node\* convertToBST(Node\* root) {

vector<int> nodes;

inorder\_traverse(root, nodes);

sort(nodes.begin(), nodes.end()); // Sorting the values

return sortedArrayToBST(nodes, 0, nodes.size() - 1);

}

};

int main() {

Node\* root = new Node(5);

root->left = new Node(1);

root->right = new Node(4);

root->right->left = new Node(3);

root->right->right = new Node(6);

Solution solution;

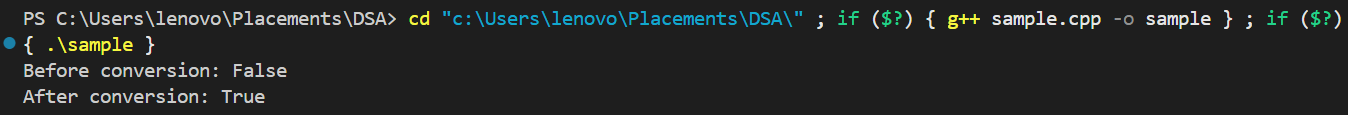
cout << "Before conversion: " << (solution.isValidBST(root) ? "True" : "False") << endl;

Node\* validBST = solution.convertToBST(root);

cout << "After conversion: " << (solution.isValidBST(validBST) ? "True" : "False") << endl;

return 0;

}



TIME COMPLEXITY: O(n log n)  
SPACE COMPLEXITY: O(n)

**3. Top View, Bottom View, Right View, Left View of a BST**

#include <iostream>

#include <queue>

#include <map>

#include <vector>

#include <climits>

using namespace std;

struct Node {

int val;

Node\* left;

Node\* right;

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

};

class Solution {

public:

// Top View

void topView(Node\* root) {

if (!root) return;

map<int, int> topNodes;

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

q.push({root, 0});

while (!q.empty()) {

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

int hd = q.front().second;

q.pop();

if (topNodes.find(hd) == topNodes.end()) {

topNodes[hd] = node->val;

}

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

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

}

for (auto& node : topNodes) {

cout << node.second << " ";

}

cout << endl;

}

// Bottom View

void bottomView(Node\* root) {

if (!root) return;

map<int, int> bottomNodes;

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

q.push({root, 0});

while (!q.empty()) {

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

int hd = q.front().second;

q.pop();

bottomNodes[hd] = node->val;

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

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

}

for (auto& node : bottomNodes) {

cout << node.second << " ";

}

cout << endl;

}

// Right View

void rightView(Node\* root) {

if (!root) return;

queue<Node\*> q;

q.push(root);

while (!q.empty()) {

int n = q.size();

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

Node\* node = q.front();

q.pop();

if (i == n - 1) {

cout << node->val << " ";

}

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

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

}

}

cout << endl;

}

// Left View

void leftView(Node\* root) {

if (!root) return;

queue<Node\*> q;

q.push(root);

while (!q.empty()) {

int n = q.size();

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

Node\* node = q.front();

q.pop();

if (i == 0) {

cout << node->val << " ";

}

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

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

}

}

cout << endl;

}

};

int main() {

Node\* root = new Node(1);

root->left = new Node(2);

root->right = new Node(3);

root->left->left = new Node(4);

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

root->right->left = new Node(6);

root->right->right = new Node(7);

root->left->left->left = new Node(8);

root->left->left->right = new Node(9);

Solution solution;

cout << "Top View: ";

solution.topView(root);

cout << "Bottom View: ";

solution.bottomView(root);

cout << "Right View: ";

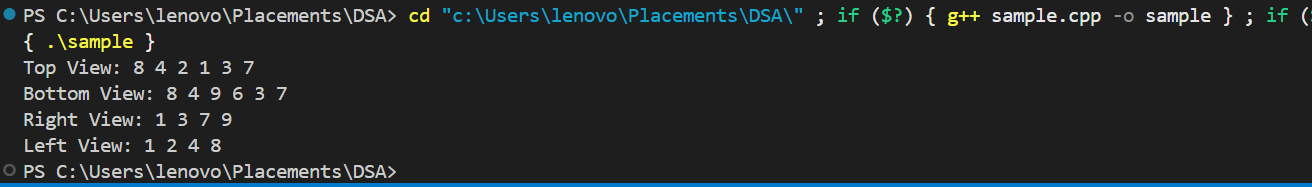
solution.rightView(root);

cout << "Left View: ";

solution.leftView(root);

return 0;

}

****

TIME COMPLEXITY: O(n)  
SPACE COMPLEXITY: O(n)