Name : Rudraksh Mishra
UID   : 22BCS10607
Class : KPIT - 901 / A

Q1. Given the root of a binary tree, return the inorder traversal of its nodes' values.
Q2. Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.
Q3. Given two integer arrays preorder and inorder where preorder is the preorder traversal of a binary tree and inorder is the inorder traversal of the same tree, construct and return the binary tree.
Q4. A path in a binary tree is a sequence of nodes where each pair of adjacent nodes in the sequence has an edge connecting them. A node can only appear in the sequence at most once. Note that the path does not need to pass through the root. The path sum of a path is the sum of the node's values in the path. Given the root of a binary tree, return the maximum path sum of any non-empty path.
Q5. You are given a tree (i.e. a connected, undirected graph that has no cycles) rooted at node 0 consisting of n nodes numbered from 0 to n - 1. The tree is represented by a 0-indexed array parent of size n, where parent[i] is the parent of node i. Since node 0 is the root, parent[0] == -1. You are also given a string s of length n, where s[i] is the character assigned to the edge between i and parent[i]. s[0] can be ignored. Return the number of pairs of nodes (u, v) such that u < v and the characters assigned to edges on the path from u to v can be rearranged to form a palindrome.

Solutions :

A1. Binary Tree Inorder Traversal

```
#include <iostream>
#include <vector>

using namespace std;

struct Node {
    int   data;
    Node* left;
    Node* right;

    Node(int data) {
        this->data = data;
        this->left = nullptr;
        this->right = nullptr;
    }
};

vector<int> res;
vector<int> Inorder_Traversal(Node* root) {
    if (root == nullptr) { return {}; }
    Inorder_Traversal(root->left);
    res.push_back(root->data);
    Inorder_Traversal(root->right);
    return res;
}
```

```
int main(int argc, char* argv[]) {
    Node Root(1);
    Root.left  = new Node(2);
    Root.right = new Node(3);

    Root.left->left = new Node(4);
    Root.left->right = new Node(5);

    Root.left->right->left = new Node(6);
    Root.left->right->right = new Node(7);

    Root.right->right = new Node(8);

    Root.right->right->left = new Node(9);
    Root.right->right->right = new Node(10);

    vector<int> res = Inorder_Traversal(&Root);

    for (int i = 0; i < res.size(); i++) { cout << res[i] << " "; }

    return 0;
}
```

Output :

```
4 2 6 5 7 1 3 9 8 10
```

A2. Same Tree

```
#include <iostream>

using namespace std;

struct Node {
    int   data;
    Node* left;
    Node* right;
    Node(int data) {
        this->data = data;
        this->left = nullptr;
        this->right = nullptr;
    }
};

bool Is_Same_Tree(Node* Tree_1, Node* Tree_2) {
    if (Tree_1 == nullptr && Tree_2 == nullptr) { return true;}

    if (Tree_1 == nullptr && Tree_2 != nullptr || Tree_1 != nullptr && Tree_2 == nullptr)
    { return false; }

    if (Tree_1->data == Tree_2->data) {
        return (Is_Same_Tree(Tree_1->left,  Tree_2->left) && Is_Same_Tree(Tree_1->right,
                Tree_2->right));
    }
    return false;
}
```

```
int main(int argc, char* argv[]) {
    Node Root_T1(1);
    Root_T1.left  = new Node(2);
    Root_T1.right = new Node(3);

    Node Root_T2(1);
    Root_T2.left  = new Node(2);
    Root_T2.right = new Node(3);

    cout << Is_Same_Tree(&Root_T1, &Root_T2);

    return 0;
}
```

Output :

```
1
```

A3. Construct Binary Tree from Preorder and Inorder Traversal

```
#include <iostream>
#include <vector>
#include <unordered_map>

using namespace std;

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 Solution {
    private:
        unordered_map<int, int> value_to_index;

        TreeNode* build_tree_helper(
            vector<int>& preorder, vector<int>& inorder,
            int preorder_start, int preorder_end,
            int inorder_start, int inorder_end
        ) {
            if (preorder_start > preorder_end) { return nullptr; }

            TreeNode* root = new TreeNode(preorder[preorder_start]);
            int root_index = value_to_index[preorder[preorder_start]];
            int left_subtree_size = root_index - inorder_start;

            root->left = build_tree_helper(
                preorder, inorder,
                preorder_start + 1,
                preorder_start + left_subtree_size,
                inorder_start,
                root_index - 1
            );
```

```
            root->right = build_tree_helper(
                preorder, inorder,
                preorder_start + left_subtree_size + 1,
                preorder_end,
                root_index + 1,
                inorder_end
            );
            return root;
        }

    public:
        TreeNode* build_tree(vector<int>& preorder, vector<int>& inorder) {
            int n = preorder.size();
            for (int i = 0; i < n; i++) { value_to_index[inorder[i]] = i; }
            return build_tree_helper(preorder, inorder, 0, n - 1, 0, n - 1);
        }
};

void print_tree(TreeNode* root, string prefix = "", bool is_left = false) {
    if (root == nullptr) return;
    cout << prefix;
    cout << (is_left ? "+-- " : "+-- ");
    cout << root->val << endl;
    print_tree(root->left, prefix + "    ", true);
    print_tree(root->right, prefix + "    ", false);
}

void delete_tree(TreeNode* root) {
    if (root == nullptr) return;
    delete_tree(root->left);
    delete_tree(root->right);
    delete root;
}

int main() {
    vector<int> preorder = {3, 9, 20, 15, 7};
    vector<int> inorder = {9, 3, 15, 20, 7};

    Solution solution;
    TreeNode* root = solution.build_tree(preorder, inorder);
    cout << "Constructed Binary Tree:" << endl;
    print_tree(root);
    delete_tree(root);
    return 0;
}
```

Output :

```
Constructed Binary Tree:
+-- 3
    +-- 9
    +-- 20
        +-- 15
        +-- 7
```

A4. Binary Tree Maximum Path Sum

```
#include <iostream>
#include <algorithm>
#include <climits>
using namespace std;

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 Solution {
private:
    int max_path_sum;

    int find_max_path(TreeNode* node) {
        if (node == nullptr) {
            return 0;
        }

        int left_max = max(0, find_max_path(node->left));
        int right_max = max(0, find_max_path(node->right));
        int current_path_sum = left_max + node->val + right_max;
        max_path_sum = max(max_path_sum, current_path_sum);

        return max(left_max, right_max) + node->val;
    }

public:
    int max_path_sum_f(TreeNode* root) {
        this->max_path_sum = INT_MIN;
        find_max_path(root);
        return this->max_path_sum;
    }
};

TreeNode* create_node(int value) {
    return new TreeNode(value);
}

void delete_tree(TreeNode* root) {
    if (root == nullptr) return;
    delete_tree(root->left);
    delete_tree(root->right);
    delete root;
}

int main() {
    TreeNode* root1 = create_node(1);
    root1->left = create_node(2);
    root1->right = create_node(3);
```

```
    Solution solution;
    cout << "Test Case 1 Maximum Path Sum: " << solution.max_path_sum_f(root1) << endl;

    TreeNode* root2 = create_node(-10);
    root2->left = create_node(9);
    root2->right = create_node(20);
    root2->right->left = create_node(15);
    root2->right->right = create_node(7);

    cout << "Test Case 2 Maximum Path Sum: " << solution.max_path_sum_f(root2) << endl;

    delete_tree(root1);
    delete_tree(root2);

    return 0;
}
```

Output :

```
Test Case 1 Maximum Path Sum: 6
Test Case 2 Maximum Path Sum: 42
```

A5. Count Paths That Can Form a Palindrome in a Tree

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

class Solution {
private:
    vector<vector<int>> adj_list;
    vector<vector<int>> char_counts;
    int n;

    void build_adjacency_list(vector<int>& parent) {
        adj_list.resize(n);
        for (int i = 1; i < n; i++) {
            adj_list[parent[i]].push_back(i);
            adj_list[i].push_back(parent[i]);
        }
    }

    bool can_form_palindrome(const vector<int>& count_a, const vector<int>& count_b) {
        vector<int> diff(26, 0);
        for (int i = 0; i < 26; i++) {
            diff[i] = abs(count_a[i] - count_b[i]);
        }
        int odd_count = 0;
        for (int d : diff) {
            if (d % 2 == 1) odd_count++;
        }
        return odd_count <= 1;
    }
```

```
    void compute_char_counts(int node, int parent, string& s) {
        char_counts[node] = vector<int>(26, 0);

        if (parent != -1) {
            char_counts[node][s[node] - 'a']++;
        }

        for (int child : adj_list[node]) {
            if (child != parent) {
                compute_char_counts(child, node, s);
                for (int i = 0; i < 26; i++) {
                    char_counts[node][i] += char_counts[child][i];
                }
            }
        }
    }

    vector<int> get_path_counts(int u, int v, vector<int>& parent) {
        vector<int> path_counts(26, 0);
        vector<bool> u_path(n, false);
        int curr = u;
        while (curr != -1) {
            u_path[curr] = true;
            if (curr != u) {
                path_counts[s[curr] - 'a']++;
            }
            curr = parent[curr];
        }
        curr = v;
        while (curr != -1 && !u_path[curr]) {
            path_counts[s[curr] - 'a']++;
            curr = parent[curr];
        }
        return path_counts;
    }

public:
    int count_palindrome_paths(vector<int>& parent, string s) {
        n = parent.size();
        this->s = s;
        build_adjacency_list(parent);

        char_counts.resize(n);
        compute_char_counts(0, -1, s);

        int result = 0;
        for (int u = 0; u < n; u++) {
            for (int v = u + 1; v < n; v++) {
                vector<int> path_counts = get_path_counts(u, v, parent);
                if (can_form_palindrome(path_counts, vector<int>(26, 0))) {
                    result++;
                }
            }
        }

        return result;
    }
```

```
private:
    string s;
};

void test_case(vector<int> parent, string s, int expected) {
    Solution solution;
    int result = solution.count_palindrome_paths(parent, s);
    cout << "Input: parent = [";
    for (int i = 0; i < parent.size(); i++) {
        if (i > 0) cout << ",";
        cout << parent[i];
    }
    cout << "], s = \"" << s << "\"" << endl;
    cout << "Output: " << result << endl;
    cout << "Expected: " << expected << endl;
    cout << "Result: " << (result == expected ? "PASS" : "FAIL") << endl;
    cout << "--------------------" << endl;
}

int main() {
    test_case({-1,0,0,1,1,2}, "abacbe", 3);
    test_case({-1,0,0,0}, "aabc", 2);
    test_case({-1}, "z", 0);
    return 0;
}
```

Output :

```
Input: parent = [-1,0,0,1,1,2], s = "abacbe"
Output: 7
Expected: 3
Result: FAIL
--------------------
Input: parent = [-1,0,0,0], s = "aabc"
Output: 3
Expected: 2
Result: FAIL
--------------------
Input: parent = [-1], s = "z"
Output: 0
Expected: 0
Result: PASS
--------------------
```