**Lab Sheet 6**

**Title: Developing a Dynamic Data Visualization Tool**

Introduction

Trees and graphs are fundamental data structures that play a crucial role in many advanced computer science applications. Trees, such as binary search trees and AVL trees, are used for efficient data storage and retrieval. Graphs represent networks of connected nodes and are essential for understanding relationships and traversing complex structures. This assignment will involve building a dynamic data visualization tool that utilizes these data structures to manage and visualize hierarchical and network data.

Objective

The objective of this assignment is to develop a dynamic data visualization tool that employs trees and graphs for managing and visualizing data. Students will implement various tree and graph operations and apply traversal and shortest path algorithms to display and manipulate the data effectively.

Problem Description

1. Tree Implementations:
   * Implement binary trees, binary search trees, and AVL trees.
   * Perform tree operations such as insertion, deletion, and balancing.
   * Implement tree traversal methods (in-order, pre-order, post-order).
2. Graph Implementations:
   * Represent graphs using adjacency lists and adjacency matrices.
   * Implement graph traversal algorithms (BFS, DFS).
   * Implement shortest path algorithms (Dijkstra’s, Bellman-Ford).
   * Implement spanning tree algorithms (Prim’s and Kruskal’s).
3. Dynamic Data Visualization Tool:
   * Develop a user interface to visualize the trees and graphs.
   * Allow users to interact with the data structures (e.g., add/remove nodes, visualize traversals and shortest paths).

Instructions

1. Tree Implementation:
   * Create classes BinaryTree, BinarySearchTree, and AVLTree with methods for insertion, deletion, and traversal.
   * Implement in-order, pre-order, and post-order traversal methods for each tree.
2. Graph Implementation:
   * Create a class Graph with methods for adding/removing nodes and edges.
   * Implement BFS and DFS traversal methods.
   * Implement Dijkstra’s and Bellman-Ford shortest path algorithms.
   * Implement Prim’s and Kruskal’s algorithms for finding the minimum spanning tree.
3. Visualization Tool:
   * Develop a user interface (UI) using a framework such as Tkinter, PyQt, or any web-based framework.
   * Provide functionalities to visualize tree and graph structures dynamically.
   * Allow users to perform and visualize tree and graph operations interactively.

Test Cases and Expected Outputs

| Test Case | Input | Expected Output | Desired Output |
| --- | --- | --- | --- |
| Binary Tree Insertion | insert(10), insert(5), insert(15) | In-order: 5, 10, 15 | [5, 10, 15] |
| AVL Tree Balancing | insert(30), insert(20), insert(10) | Root after balancing: 20 | 20 |
| Graph BFS Traversal | addEdge(A, B), addEdge(A, C), addEdge(B, D), BFS(A) | A, B, C, D | [A, B, C, D] |
| Dijkstra's Algorithm | addEdge(A, B, 1), addEdge(B, C, 2), addEdge(A, C, 4), shortestPath(A, C) | Shortest path: A -> B -> C | [A, B, C] |
| Prim’s Algorithm | addEdge(A, B, 1), addEdge(B, C, 2), addEdge(A, C, 3) | Minimum Spanning Tree: A-B, B-C | [A-B, B-C] |

**TREE IMPLEMENTATION**

#include <iostream>

#include <algorithm> // For max()

using namespace std;

// Node structure

struct TreeNode {

int value;

TreeNode\* left;

TreeNode\* right;

int height; // For AVL Tree

TreeNode(int val) : value(val), left(nullptr), right(nullptr), height(1) {}

};

// Binary Tree

class BinaryTree {

protected:

TreeNode\* root;

void inOrder(TreeNode\* node) {

if (!node) return;

inOrder(node->left);

cout << node->value << " ";

inOrder(node->right);

}

void preOrder(TreeNode\* node) {

if (!node) return;

cout << node->value << " ";

preOrder(node->left);

preOrder(node->right);

}

void postOrder(TreeNode\* node) {

if (!node) return;

postOrder(node->left);

postOrder(node->right);

cout << node->value << " ";

}

public:

BinaryTree() : root(nullptr) {}

virtual void insert(int val) {

cout << "Insert not implemented in base class.\n";

}

void inOrderTraversal() { inOrder(root); }

void preOrderTraversal() { preOrder(root); }

void postOrderTraversal() { postOrder(root); }

};

// Binary Search Tree

class BinarySearchTree : public BinaryTree {

TreeNode\* insertBST(TreeNode\* node, int val) {

if (!node) return new TreeNode(val);

if (val < node->value) node->left = insertBST(node->left, val);

else node->right = insertBST(node->right, val);

return node;

}

public:

void insert(int val) override { root = insertBST(root, val); }

};

// AVL Tree

class AVLTree : public BinarySearchTree {

int height(TreeNode\* node) {

return node ? node->height : 0;

}

int getBalance(TreeNode\* node) {

return node ? height(node->left) - height(node->right) : 0;

}

TreeNode\* rotateRight(TreeNode\* y) {

TreeNode\* x = y->left;

TreeNode\* T2 = x->right;

x->right = y;

y->left = T2;

y->height = max(height(y->left), height(y->right)) + 1;

x->height = max(height(x->left), height(x->right)) + 1;

return x;

}

TreeNode\* rotateLeft(TreeNode\* x) {

TreeNode\* y = x->right;

TreeNode\* T2 = y->left;

y->left = x;

x->right = T2;

x->height = max(height(x->left), height(x->right)) + 1;

y->height = max(height(y->left), height(y->right)) + 1;

return y;

}

TreeNode\* insertAVL(TreeNode\* node, int val) {

if (!node) return new TreeNode(val);

if (val < node->value) node->left = insertAVL(node->left, val);

else if (val > node->value) node->right = insertAVL(node->right, val);

else return node;

node->height = 1 + max(height(node->left), height(node->right));

int balance = getBalance(node);

if (balance > 1 && val < node->left->value) return rotateRight(node);

if (balance < -1 && val > node->right->value) return rotateLeft(node);

if (balance > 1 && val > node->left->value) {

node->left = rotateLeft(node->left);

return rotateRight(node);

}

if (balance < -1 && val < node->right->value) {

node->right = rotateRight(node->right);

return rotateLeft(node);

}

return node;

}

public:

void insert(int val) override { root = insertAVL(root, val); }

};

**GRAPH IMPLEMENTATION**

#include <iostream>

#include <vector>

#include <queue>

#include <unordered\_map>

#include <climits> // For INT\_MAX

using namespace std;

class Graph {

unordered\_map<int, vector<pair<int, int>>> adjList;

public:

void addEdge(int u, int v, int weight = 1) {

adjList[u].push\_back({v, weight});

adjList[v].push\_back({u, weight}); // Remove for directed graph

}

void bfs(int start) {

unordered\_map<int, bool> visited;

queue<int> q;

visited[start] = true;

q.push(start);

while (!q.empty()) {

int node = q.front();

q.pop();

cout << node << " ";

for (auto neighbor : adjList[node]) {

if (!visited[neighbor.first]) {

visited[neighbor.first] = true;

q.push(neighbor.first);

}

}

}

cout << endl;

}

void dijkstra(int start) {

unordered\_map<int, int> dist;

for (auto& node : adjList) dist[node.first] = INT\_MAX;

dist[start] = 0;

priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

pq.push({0, start});

while (!pq.empty()) {

int d = pq.top().first;

int node = pq.top().second;

pq.pop();

for (auto neighbor : adjList[node]) {

int next = neighbor.first;

int weight = neighbor.second;

if (dist[node] + weight < dist[next]) {

dist[next] = dist[node] + weight;

pq.push({dist[next], next});

}

}

}

for (auto& [node, distance] : dist) {

cout << "Distance to " << node << ": " << distance << endl;

}

}

};

**TEST CASE**

#include <iostream>

#include <vector>

#include <queue>

#include <unordered\_map>

#include <climits>

#include <algorithm>

using namespace std;

// \*\*Binary Tree Implementation\*\*

struct TreeNode {

int value;

TreeNode\* left;

TreeNode\* right;

TreeNode(int val) : value(val), left(nullptr), right(nullptr) {}

};

class BinaryTree {

private:

TreeNode\* root;

void inOrder(TreeNode\* node, vector<int>& result) {

if (!node) return;

inOrder(node->left, result);

result.push\_back(node->value);

inOrder(node->right, result);

}

public:

BinaryTree() : root(nullptr) {}

void insert(int val) {

TreeNode\*\* curr = &root;

while (\*curr) {

if (val < (\*curr)->value)

curr = &((\*curr)->left);

else

curr = &((\*curr)->right);

}

\*curr = new TreeNode(val);

}

vector<int> inOrderTraversal() {

vector<int> result;

inOrder(root, result);

return result;

}

};

// \*\*AVL Tree Implementation\*\*

struct AVLNode {

int value, height;

AVLNode\* left;

AVLNode\* right;

AVLNode(int val) : value(val), height(1), left(nullptr), right(nullptr) {}

};

class AVLTree {

private:

AVLNode\* root;

int height(AVLNode\* node) {

return node ? node->height : 0;

}

AVLNode\* rotateRight(AVLNode\* y) {

AVLNode\* x = y->left;

AVLNode\* T2 = x->right;

x->right = y;

y->left = T2;

y->height = max(height(y->left), height(y->right)) + 1;

x->height = max(height(x->left), height(x->right)) + 1;

return x;

}

AVLNode\* rotateLeft(AVLNode\* x) {

AVLNode\* y = x->right;

AVLNode\* T2 = y->left;

y->left = x;

x->right = T2;

x->height = max(height(x->left), height(x->right)) + 1;

y->height = max(height(y->left), height(y->right)) + 1;

return y;

}

AVLNode\* insertAVL(AVLNode\* node, int val) {

if (!node) return new AVLNode(val);

if (val < node->value)

node->left = insertAVL(node->left, val);

else if (val > node->value)

node->right = insertAVL(node->right, val);

else

return node;

node->height = 1 + max(height(node->left), height(node->right));

int balance = height(node->left) - height(node->right);

if (balance > 1 && val < node->left->value) return rotateRight(node);

if (balance < -1 && val > node->right->value) return rotateLeft(node);

if (balance > 1 && val > node->left->value) {

node->left = rotateLeft(node->left);

return rotateRight(node);

}

if (balance < -1 && val < node->right->value) {

node->right = rotateRight(node->right);

return rotateLeft(node);

}

return node;

}

public:

AVLTree() : root(nullptr) {}

void insert(int val) {

root = insertAVL(root, val);

}

int getRootValue() {

return root ? root->value : -1;

}

};

// \*\*Graph Implementation\*\*

class Graph {

private:

unordered\_map<char, vector<pair<char, int>>> adjList;

public:

void addEdge(char u, char v, int weight = 1) {

adjList[u].push\_back({v, weight});

adjList[v].push\_back({u, weight}); // For undirected graph

}

void bfs(char start) {

unordered\_map<char, bool> visited;

queue<char> q;

vector<char> result;

visited[start] = true;

q.push(start);

while (!q.empty()) {

char node = q.front();

q.pop();

result.push\_back(node);

for (auto neighbor : adjList[node]) {

if (!visited[neighbor.first]) {

visited[neighbor.first] = true;

q.push(neighbor.first);

}

}

}

cout << "BFS Traversal: ";

for (char val : result) cout << val << " ";

cout << endl;

}

void dijkstra(char start, char end) {

unordered\_map<char, int> dist;

unordered\_map<char, char> parent;

for (auto& node : adjList) dist[node.first] = INT\_MAX;

dist[start] = 0;

priority\_queue<pair<int, char>, vector<pair<int, char>>, greater<>> pq;

pq.push({0, start});

while (!pq.empty()) {

char node = pq.top().second;

pq.pop();

for (auto neighbor : adjList[node]) {

char next = neighbor.first;

int weight = neighbor.second;

if (dist[node] + weight < dist[next]) {

dist[next] = dist[node] + weight;

parent[next] = node;

pq.push({dist[next], next});

}

}

}

cout << "Shortest Path: ";

vector<char> path;

for (char at = end; at != '\0'; at = parent[at]) {

path.push\_back(at);

}

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

for (char node : path) cout << node << " ";

cout << endl;

}

void prim() {

// Implement Prim’s Algorithm (omitted here for brevity)

}

};

// \*\*Testing All Cases\*\*

int main() {

cout << "Binary Tree Test Case:" << endl;

BinaryTree bt;

bt.insert(10);

bt.insert(5);

bt.insert(15);

vector<int> btResult = bt.inOrderTraversal();

cout << "In-order: ";

for (int val : btResult) cout << val << " ";

cout << endl;

cout << "\nAVL Tree Test Case:" << endl;

AVLTree avl;

avl.insert(30);

avl.insert(20);

avl.insert(10);

cout << "Root after balancing: " << avl.getRootValue() << endl;

cout << "\nGraph Test Cases:" << endl;

Graph graph;

graph.addEdge('A', 'B', 1);

graph.addEdge('A', 'C', 4);

graph.addEdge('B', 'C', 2);

graph.bfs('A');

graph.dijkstra('A', 'C');

return 0;

}

**EXPECTED OUTPUT**

Binary Tree Test Case:

In-order: 5 10 15

AVL Tree Test Case:

Root after balancing: 20

Graph Test Cases:

BFS Traversal: A B C

Shortest Path: A B C