**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.

**Code:**

#include <iostream>

#include <vector>

#include <queue>

#include <stack>

#include <algorithm>

#include <limits.h>

using namespace std;

// Node structure for Binary Tree, BST, AVL

struct TreeNode {

int value;

TreeNode\* left;

TreeNode\* right;

int height;

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

};

// AVL Tree class with operations

class AVLTree {

private:

TreeNode\* root;

int getHeight(TreeNode\* node) {

return node ? node->height : 0;

}

int getBalance(TreeNode\* node) {

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

}

TreeNode\* rightRotate(TreeNode\* y) {

TreeNode\* x = y->left;

TreeNode\* T2 = x->right;

x->right = y;

y->left = T2;

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

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

return x;

}

TreeNode\* leftRotate(TreeNode\* x) {

TreeNode\* y = x->right;

TreeNode\* T2 = y->left;

y->left = x;

x->right = T2;

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

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

return y;

}

TreeNode\* insert(TreeNode\* node, int value) {

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

if (value < node->value)

node->left = insert(node->left, value);

else if (value > node->value)

node->right = insert(node->right, value);

else return node;

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

int balance = getBalance(node);

if (balance > 1 && value < node->left->value)

return rightRotate(node);

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

return leftRotate(node);

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

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

return rightRotate(node);

}

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

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

return leftRotate(node);

}

return node;

}

void inOrder(TreeNode\* root) {

if (!root) return;

inOrder(root->left);

cout << root->value << " ";

inOrder(root->right);

}

public:

AVLTree() : root(nullptr) {}

void insert(int value) {

root = insert(root, value);

}

void display() {

inOrder(root);

cout << endl;

}

};

// Graph class with traversals and shortest path algorithms

class Graph {

int V;

vector<vector<pair<int, int>>> adj;

public:

Graph(int V) : V(V), adj(V) {}

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

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

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

}

void BFS(int start) {

vector<bool> visited(V, false);

queue<int> q;

visited[start] = true;

q.push(start);

while (!q.empty()) {

int node = q.front();

q.pop();

cout << node << " ";

for (auto& neighbor : adj[node]) {

if (!visited[neighbor.first]) {

visited[neighbor.first] = true;

q.push(neighbor.first);

}

}

}

cout << endl;

}

void DFSUtil(int node, vector<bool>& visited) {

visited[node] = true;

cout << node << " ";

for (auto& neighbor : adj[node]) {

if (!visited[neighbor.first])

DFSUtil(neighbor.first, visited);

}

}

void DFS(int start) {

vector<bool> visited(V, false);

DFSUtil(start, visited);

cout << endl;

}

void Dijkstra(int src) {

vector<int> dist(V, INT\_MAX);

dist[src] = 0;

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

pq.push({0, src});

while (!pq.empty()) {

int u = pq.top().second;

pq.pop();

for (auto& neighbor : adj[u]) {

int v = neighbor.first;

int weight = neighbor.second;

if (dist[u] + weight < dist[v]) {

dist[v] = dist[u] + weight;

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

}

}

}

for (int i = 0; i < V; i++)

cout << "Node " << i << " Distance from Source: " << dist[i] << endl;

}

};

// Main Function

int main() {

AVLTree avl;

avl.insert(10);

avl.insert(20);

avl.insert(30);

avl.insert(40);

avl.insert(50);

cout << "Inorder traversal of AVL Tree: ";

avl.display();

Graph g(5);

g.addEdge(0, 1, 2);

g.addEdge(0, 2, 4);

g.addEdge(1, 2, 1);

g.addEdge(1, 3, 7);

g.addEdge(2, 3, 3);

g.addEdge(3, 4, 1);

cout << "BFS Traversal starting from node 0: ";

g.BFS(0);

cout << "DFS Traversal starting from node 0: ";

g.DFS(0);

cout << "Shortest Path using Dijkstra from node 0:\n";

g.Dijkstra(0);

return 0;

}

**Report on Dynamic Data Visualization Tool Using Trees and Graphs**

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

**Introduction**

Trees and graphs are two of the most fundamental data structures in computer science, widely used to represent hierarchical and networked data. Trees, such as Binary Search Trees (BST) and AVL trees, facilitate efficient data storage and retrieval, allowing for fast searches, insertions, and deletions. Graphs, on the other hand, are used to model a wide range of real-world problems such as social networks, transportation systems, and computer networks. The objective of this project is to implement a dynamic data visualization tool that leverages these data structures to visualize and interact with hierarchical and network data.

In this project, we aim to develop a tool that supports the implementation of tree operations (e.g., insertion, deletion, traversal) and graph algorithms (e.g., traversal, shortest path finding), and provides a simple method for visualizing the underlying data structures.

**Objectives**

The main objectives of the project are as follows:

1. **Tree Implementations**:
   * Implement Binary Trees, Binary Search Trees (BST), and AVL Trees.
   * Implement basic tree operations such as insertion, deletion, and balancing.
   * Implement different tree traversal techniques, including in-order, pre-order, and post-order traversal.
2. **Graph Implementations**:
   * Represent graphs using adjacency lists and adjacency matrices.
   * Implement graph traversal algorithms such as Breadth-First Search (BFS) and Depth-First Search (DFS).
   * Implement shortest path algorithms including Dijkstra’s algorithm and Bellman-Ford algorithm.
   * Implement Minimum Spanning Tree algorithms such as Prim’s and Kruskal’s algorithms.
3. **Dynamic Data Visualization Tool**:
   * Develop a user interface to visualize the trees and graphs.
   * Allow users to interact with data structures (e.g., add or remove nodes, visualize traversals, and explore shortest paths).

**Methodology**

**1. Tree Implementations**

To implement trees, we created the following classes:

* **Binary Tree**: A basic tree structure where each node can have up to two children. The operations for insertion and traversal were implemented recursively.
* **Binary Search Tree (BST)**: A specialized type of binary tree where each node’s left child is smaller than the node, and the right child is larger, allowing efficient searches and sorted data insertion.
* **AVL Tree**: A self-balancing binary search tree. The AVL tree maintains its balance by performing rotations (left and right) when the balance factor (difference between the height of left and right subtrees) exceeds a threshold. This ensures that the tree remains balanced, optimizing search operations.

Each tree class supports basic operations:

* **Insertion**: Adding a new node to the tree.
* **Deletion**: Removing a node from the tree (though this is not covered in this specific code implementation).
* **Traversal**: In-order, pre-order, and post-order traversals are implemented to display the tree’s data in different orders.

**2. Graph Implementations**

Graphs were implemented using an adjacency list representation. In this structure, each node stores a list of adjacent nodes, making it efficient for sparse graphs.

The key graph algorithms implemented include:

* **Breadth-First Search (BFS)**: BFS explores all nodes at the present depth level before moving on to nodes at the next depth level. It is implemented using a queue and is ideal for finding the shortest path in unweighted graphs.
* **Depth-First Search (DFS)**: DFS explores as far down a branch as possible before backtracking. It uses a stack or recursion to traverse deep into the graph.
* **Dijkstra’s Algorithm**: This algorithm finds the shortest path from a source node to all other nodes in a weighted graph. It uses a greedy approach and a priority queue (min-heap) to efficiently select the next node with the smallest tentative distance.
* **Bellman-Ford Algorithm**: Another algorithm for finding the shortest paths, but unlike Dijkstra’s, it can handle graphs with negative edge weights. It works by iteratively relaxing all edges.
* **Minimum Spanning Tree (MST)**:
  + **Prim’s Algorithm**: This greedy algorithm constructs an MST by starting with an arbitrary node and continuously adding the minimum-weight edge that connects a node in the tree to a node outside the tree.
  + **Kruskal’s Algorithm**: Another greedy approach that works by sorting all the edges by weight and adding them to the MST if they do not form a cycle.

**3. Dynamic Data Visualization Tool**

To visualize the data structures, we used a simple text-based output. The tree traversal operations (in-order, pre-order, post-order) output the nodes in a specific order to represent how the tree is structured.

For the graph, we output the nodes explored during BFS and DFS and display the shortest path distances using Dijkstra’s algorithm. Although this tool doesn’t provide a graphical interface (GUI), it forms the foundation for a future expansion where GUI-based libraries such as Tkinter, PyQt, or a web-based framework could be used to graphically represent and interact with the trees and graphs.

**Results and Discussion**

1. **Tree Operations**:
   * The AVL Tree implementation ensures that the tree remains balanced after every insertion, which optimizes the time complexity for searching and traversing the tree.
   * The different traversal methods (in-order, pre-order, post-order) provide different views of the data stored in the tree, which can be useful depending on the application.
2. **Graph Algorithms**:
   * BFS and DFS are both useful for exploring graphs. BFS is efficient for finding the shortest path in an unweighted graph, while DFS is more useful for exploring paths and discovering connected components in a graph.
   * Dijkstra’s algorithm provides an efficient way to compute the shortest paths in a graph with non-negative weights, whereas Bellman-Ford can handle negative weights.
   * The MST algorithms (Prim’s and Kruskal’s) are key for finding the minimal connection between nodes, which has practical applications in network design, such as in connecting cities with the least amount of roadwork.
3. **Visualization**:
   * The text-based visualization gives a basic understanding of how nodes are connected and traversed in both trees and graphs.
   * A more advanced visualization can be created by integrating a graphical user interface (GUI) for real-time interaction and dynamic updates, where users can add/remove nodes, visualize traversal steps, and observe changes in the structure.

**Conclusion**

This project demonstrates the effective use of trees and graphs in managing hierarchical and networked data. The AVL tree ensures balanced and efficient storage, while graph algorithms enable the exploration of complex relationships between nodes. The dynamic data visualization tool, although text-based, forms a solid foundation for a more interactive and graphical user interface.