**Lab Sheet 7**

**Title: Developing a Disaster Response Management System**

**Introduction**

In times of natural disasters, efficient management and coordination of resources and information are crucial for effective response and recovery. This project-based assignment focuses on building a disaster response management system using trees and graphs. The system will help manage information about affected areas, available resources, and rescue operations, ensuring that help reaches those in need promptly.

**Objective**

The objective of this assignment is to develop a disaster response management system that utilizes trees and graphs to manage and coordinate disaster response efforts. Students will implement various tree and graph operations to handle information about affected areas, resources, and rescue operations.

**Introduction to Data Structures**

1. Trees:
   * Terminologies: Trees are hierarchical data structures consisting of nodes connected by edges. Key terms include root (the top node), leaf (a node with no children), and height (the length of the longest path from the root to a leaf).
   * Binary Trees: A type of tree where each node has at most two children, referred to as the left and right child.
   * Binary Search Trees (BST): A binary tree where the left child of a node contains only nodes with values less than the parent node, and the right child contains only nodes with values greater than the parent node.
   * AVL Trees: A self-balancing binary search tree where the difference in heights between the left and right subtrees of any node is at most one.
   * Tree Traversals: Methods to visit all nodes in a tree, including in-order (left, root, right), pre-order (root, left, right), and post-order (left, right, root) traversals.
2. Graphs:
   * Definitions: Graphs consist of nodes (or vertices) and edges connecting pairs of nodes. They can be directed or undirected.
   * Representations: Graphs can be represented using adjacency lists (a list of lists where each sublist represents a node and its adjacent nodes) or adjacency matrices (a 2D array where the cell [i][j] indicates the presence of an edge between nodes i and j).
   * Graph Traversal Algorithms: Techniques to visit all nodes in a graph, such as Breadth-First Search (BFS) and Depth-First Search (DFS).
   * Shortest Path Algorithms: Methods to find the shortest path between nodes, including Dijkstra’s and Bellman-Ford algorithms.
   * Spanning Trees: A subgraph that includes all the nodes of the original graph, connected with the minimum number of edges. Prim’s and Kruskal’s algorithms are used to find minimum spanning trees.

**Problem Description**

1. Tree Implementations:
   * Implement AVL trees to manage hierarchical data of affected regions.
   * Perform tree operations such as insertion, deletion, and balancing.
   * Implement tree traversal methods (in-order, pre-order, post-order) to retrieve information about affected areas.
2. Graph Implementations:
   * Represent the disaster-affected regions and resource centers using graphs (adjacency lists and adjacency matrices).
   * Implement graph traversal algorithms (BFS, DFS) to navigate through the network of affected areas and resource centers.
   * Implement shortest path algorithms (Dijkstra’s, Bellman-Ford) to determine the quickest route for resource delivery.
   * Implement spanning tree algorithms (Prim’s and Kruskal’s) to establish efficient communication networks.
3. Disaster Response Management System:
   * Develop a user interface to manage and visualize the affected areas, resource centers, and rescue operations.
   * Allow users to interact with the system (e.g., add/remove affected areas, visualize resource distribution routes, manage rescue operations).

**Instructions**

1. Tree Implementation:
   * Create a class AVLTree with methods for insertion, deletion, and traversal.
   * Implement in-order, pre-order, and post-order traversal methods to retrieve hierarchical data.
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. Disaster Response Management System:
   * Develop a user interface (UI) using a framework such as Tkinter, PyQt, or any web-based framework.
   * Provide functionalities to manage and visualize affected areas and resource centers.
   * Allow users to perform and visualize tree and graph operations interactively.

**Code:**

#include <iostream>

#include <vector>

#include <queue>

#include <climits>

#include <stack>

#include <map>

#include <set>

#include <algorithm>

using namespace std;

// AVL Tree Implementation for Managing Affected Areas

class AVLTree {

private:

struct Node {

string data;

Node\* left;

Node\* right;

int height;

Node(string data) : data(data), left(nullptr), right(nullptr), height(1) {}

};

Node\* root;

int height(Node\* node) {

return node ? node->height : 0;

}

int balanceFactor(Node\* node) {

return height(node->left) - height(node->right);

}

Node\* rotateRight(Node\* y) {

Node\* x = y->left;

Node\* 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;

}

Node\* rotateLeft(Node\* x) {

Node\* y = x->right;

Node\* 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;

}

Node\* balance(Node\* node) {

int balance = balanceFactor(node);

if (balance > 1) {

if (balanceFactor(node->left) < 0) {

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

}

return rotateRight(node);

}

if (balance < -1) {

if (balanceFactor(node->right) > 0) {

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

}

return rotateLeft(node);

}

return node;

}

Node\* insert(Node\* node, const string& data) {

if (!node) return new Node(data);

if (data < node->data)

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

else if (data > node->data)

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

else

return node; // duplicate data not allowed

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

return balance(node);

}

void inOrder(Node\* node, vector<string>& result) {

if (node) {

inOrder(node->left, result);

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

inOrder(node->right, result);

}

}

public:

AVLTree() : root(nullptr) {}

void insert(const string& data) {

root = insert(root, data);

}

vector<string> inOrderTraversal() {

vector<string> result;

inOrder(root, result);

return result;

}

};

// Graph Implementation for Disaster Response Management

class Graph {

private:

map<string, vector<pair<string, int>>> adjList;

public:

void addEdge(const string& u, const string& v, int weight) {

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

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

}

void BFS(const string& start) {

set<string> visited;

queue<string> q;

visited.insert(start);

q.push(start);

while (!q.empty()) {

string node = q.front();

q.pop();

cout << node << " ";

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

if (visited.find(neighbor.first) == visited.end()) {

visited.insert(neighbor.first);

q.push(neighbor.first);

}

}

}

cout << endl;

}

void DFS(const string& start) {

set<string> visited;

stack<string> s;

s.push(start);

while (!s.empty()) {

string node = s.top();

s.pop();

if (visited.find(node) == visited.end()) {

visited.insert(node);

cout << node << " ";

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

if (visited.find(neighbor.first) == visited.end()) {

s.push(neighbor.first);

}

}

}

}

cout << endl;

}

vector<string> dijkstra(const string& source, const string& target) {

map<string, int> dist;

map<string, string> parent;

set<string> visited;

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

for (const auto& node : adjList) {

dist[node.first] = INT\_MAX;

}

dist[source] = 0;

pq.push({0, source});

while (!pq.empty()) {

string node = pq.top().second;

pq.pop();

if (visited.find(node) != visited.end()) continue;

visited.insert(node);

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

string neighborNode = neighbor.first;

int weight = neighbor.second;

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

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

parent[neighborNode] = node;

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

}

}

}

vector<string> path;

string node = target;

while (parent.find(node) != parent.end()) {

path.push\_back(node);

node = parent[node];

}

path.push\_back(source);

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

return path;

}

vector<string> primMST() {

set<string> visited;

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

vector<string> mst;

for (auto& node : adjList) {

if (visited.find(node.first) == visited.end()) {

visited.insert(node.first);

for (auto& neighbor : node.second) {

pq.push({neighbor.second, {node.first, neighbor.first}});

}

while (!pq.empty()) {

auto edge = pq.top();

pq.pop();

string u = edge.second.first;

string v = edge.second.second;

int weight = edge.first;

if (visited.find(v) == visited.end()) {

visited.insert(v);

mst.push\_back(u + "-" + v);

for (auto& neighbor : adjList[v]) {

if (visited.find(neighbor.first) == visited.end()) {

pq.push({neighbor.second, {v, neighbor.first}});

}

}

}

}

}

}

return mst;

}

};

// Main Function to Simulate the System

int main() {

AVLTree tree;

Graph graph;

// Example 1: Insert regions into AVL Tree

tree.insert("Region1");

tree.insert("Region2");

tree.insert("Region3");

vector<string> treeResult = tree.inOrderTraversal();

cout << "AVL Tree In-Order Traversal: ";

for (const string& region : treeResult) {

cout << region << " ";

}

cout << endl;

// Example 2: Graph BFS Traversal

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

graph.addEdge("A", "C", 2);

graph.addEdge("B", "D", 1);

cout << "Graph BFS Traversal from A: ";

graph.BFS("A");

// Example 3: Graph Dijkstra's Algorithm

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

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

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

vector<string> dijkstraResult = graph.dijkstra("A", "C");

cout << "Dijkstra's Shortest Path from A to C: ";

for (const string& node : dijkstraResult) {

cout << node << " ";

}

cout << endl;

// Example 4: Graph Prim's Algorithm

vector<string> mst = graph.primMST();

cout << "Prim's Minimum Spanning Tree: ";

for (const string& edge : mst) {

cout << edge << " ";

}

cout << endl;

return 0;

}

**Report: Developing a Disaster Response Management System**

**Introduction**

Natural disasters require a well-coordinated response to effectively manage resources and provide timely aid to affected areas. In such emergencies, a robust system that organizes information about affected regions, available resources, and ongoing rescue operations is critical for ensuring swift and efficient disaster relief efforts. This project focuses on creating a disaster response management system that leverages trees and graphs as data structures to manage and visualize crucial information, facilitating better decision-making in times of crisis.

**Objective**

The primary objective of this project is to develop a disaster response management system that employs trees and graphs to coordinate resources and rescue operations across affected areas. By implementing various tree and graph operations, the system aims to efficiently handle data on disaster regions, resource centers, and operational routes, ultimately enhancing the delivery of emergency services.

**Introduction to Data Structures**

1. **Trees**
   * **Terminology**: Trees are hierarchical structures comprising nodes connected by edges, with the topmost node known as the root. Each node may have children, with nodes that lack children called leaves. The height of a tree is the longest path from the root to a leaf.
   * **Binary Trees**: In a binary tree, each node has up to two children, identified as left and right children.
   * **Binary Search Trees (BST)**: In BSTs, nodes follow an order where the left child has values lower than the parent, while the right child has values higher than the parent.
   * **AVL Trees**: AVL trees are self-balancing binary search trees that ensure the difference in height between left and right subtrees of any node is no more than one.
   * **Tree Traversals**: Tree traversal methods are crucial for accessing and retrieving data from a tree structure. In-order traversal (left-root-right), pre-order traversal (root-left-right), and post-order traversal (left-right-root) are among the common traversal techniques used.
2. **Graphs**
   * **Definitions**: A graph is a collection of nodes (or vertices) and edges, which can be directed (where edges have a specific direction) or undirected.
   * **Representations**: Graphs can be represented using adjacency lists (each node has a list of adjacent nodes) or adjacency matrices (a 2D array where each cell indicates an edge between nodes).
   * **Graph Traversal Algorithms**: Algorithms like Breadth-First Search (BFS) and Depth-First Search (DFS) are used to visit nodes within a graph.
   * **Shortest Path Algorithms**: Algorithms such as Dijkstra’s and Bellman-Ford help find the shortest path between nodes, which is vital for efficiently routing resources in a disaster response system.
   * **Spanning Trees**: A spanning tree includes all nodes of the original graph connected with the minimum number of edges. Prim’s and Kruskal’s algorithms are commonly used to construct minimum spanning trees, establishing efficient communication networks.

**Problem Description**

1. **Tree Implementations**
   * **AVL Trees for Affected Regions**: AVL trees are used to represent hierarchical data about disaster-affected areas. These trees enable rapid data access and maintain balance through rotation operations, allowing for efficient insertion, deletion, and traversal (in-order, pre-order, post-order).
2. **Graph Implementations**
   * **Graphs for Resource and Region Management**: Graphs are employed to represent both affected regions and resource centers. Using adjacency lists or matrices, these graphs facilitate the management of nodes (regions) and edges (resource routes).
   * **Graph Traversal**: BFS and DFS algorithms are implemented to enable navigation through networks of affected areas and resource centers, ensuring comprehensive access to all relevant nodes.
   * **Shortest Path Algorithms**: Dijkstra’s and Bellman-Ford algorithms are applied to identify the quickest route for delivering resources, ensuring that aid reaches affected areas as swiftly as possible.
   * **Spanning Tree Algorithms**: Prim’s and Kruskal’s algorithms create efficient communication networks by minimizing the number of edges needed to connect all nodes in the graph.
3. **Disaster Response Management System**
   * **User Interface (UI)**: The system includes a user-friendly interface for managing and visualizing data on affected regions, resource centers, and operational routes. It allows users to interact with the system by adding or removing affected areas, visualizing resource routes, and managing rescue efforts.
   * **Interactive Operations**: Users can perform and visualize tree and graph operations, facilitating informed decision-making through accessible data representation.

**Instructions for System Development**

1. **Tree Implementation**
   * **AVLTree Class**: An AVLTree class is designed to handle insertion, deletion, and traversal operations, maintaining balanced hierarchical data on affected regions. Traversal methods (in-order, pre-order, post-order) allow efficient retrieval of data.
2. **Graph Implementation**
   * **Graph Class**: A Graph class manages nodes and edges, supporting node addition and removal as well as BFS and DFS traversal. Dijkstra’s and Bellman-Ford algorithms for shortest paths and Prim’s and Kruskal’s algorithms for spanning trees are incorporated to facilitate optimal resource distribution.
3. **Disaster Response Management System UI**
   * **UI Framework**: The system’s UI, created using frameworks like Tkinter or PyQt, allows users to manage and visualize affected areas, resource centers, and rescue operations. Interactive functionality supports tree and graph operations, improving usability.

**Test Cases and Expected Outputs**

1. **AVL Tree Insertion and Balancing**
   * **Test Case**: Insert regions in various orders, e.g., insert("Region1"), insert("Region2"), insert("Region3").
   * **Expected Output**: After insertion and in-order traversal, the output should be a sorted sequence, such as ["Region1", "Region2", "Region3"].
   * **Tree Balancing**: If regions are inserted in reverse order, such as insert("Region3"), insert("Region2"), insert("Region1"), the AVL tree should balance itself, resulting in Region2 as the root node.
2. **Graph Traversal and Shortest Path**
   * **BFS Traversal**: For example, if edges are added between regions as addEdge(A, B), addEdge(A, C), addEdge(B, D), BFS traversal starting from A should visit nodes in the sequence ["A", "B", "C", "D"].
   * **Dijkstra’s Algorithm**: Adding weighted edges as addEdge(A, B, 1), addEdge(B, C, 2), addEdge(A, C, 4), Dijkstra’s algorithm should output the shortest path from A to C as ["A", "B", "C"].
3. **Minimum Spanning Tree (Prim’s Algorithm)**
   * **Example Test**: Given edges addEdge(A, B, 1), addEdge(B, C, 2), addEdge(A, C, 3), the minimum spanning tree should include edges ["A-B", "B-C"] to cover all nodes with minimal edges.

**Expected Learnings**

1. **AVL Tree Proficiency**: Students will learn to implement and use AVL trees for hierarchical data management.
2. **Graph Representation and Algorithms**: Skills in graph representation (adjacency lists/matrices), traversal (BFS/DFS), shortest path (Dijkstra’s/Bellman-Ford), and spanning tree algorithms (Prim’s/Kruskal’s) will be acquired.
3. **Interactive Systems with Data Structures**: By developing an interactive UI, students will gain experience in combining data structures with visual elements, enhancing both technical and user-centric skills.

**Conclusion**

This project-based approach equips students with the skills to use trees and graphs to design an essential management system for disaster response. These foundational data structures and algorithms are powerful tools in handling complex, dynamic information—crucial for efficient emergency management.