#include <iostream>

using namespace std;

struct ThreadedNode {

int data;

ThreadedNode\* left;

ThreadedNode\* right;

bool isThreaded;

ThreadedNode(int value) : data(value), left(nullptr), right(nullptr), isThreaded(false) {}

};

class ThreadedBinaryTree {

private:

ThreadedNode\* root;

void threadInOrderUtil(ThreadedNode\* root, ThreadedNode\*& prev) {

if (root) {

threadInOrderUtil(root->left, prev);

if (!root->left) {

root->left = prev;

root->isThreaded = true;

}

if (prev && !prev->right) {

prev->right = root;

prev->isThreaded = true;

}

prev = root;

threadInOrderUtil(root->right, prev);

}

}

void preOrderTraversalUtil(ThreadedNode\* root) {

if (root) {

cout << root->data << " ";

if (!root->isThreaded) {

preOrderTraversalUtil(root->left);

}

preOrderTraversalUtil(root->right);

}

}

public:

ThreadedBinaryTree() : root(nullptr) {}

void threadInOrder() {

ThreadedNode\* prev = nullptr;

threadInOrderUtil(root, prev);

}

void preOrderTraversal() {

preOrderTraversalUtil(root);

cout << endl;

}

void insert(int value) {

root = insertUtil(root, value);

}

ThreadedNode\* insertUtil(ThreadedNode\* root, int value) {

if (!root) {

return new ThreadedNode(value);

}

if (value < root->data) {

root->left = insertUtil(root->left, value);

} else if (value > root->data) {

root->right = insertUtil(root->right, value);

}

return root;

}

};

int main() {

ThreadedBinaryTree threadedTree;

int choice, value;

do {

cout << "Menu:\n";

cout << "1. Insert a node\n";

cout << "2. Thread the tree in-order\n";

cout << "3. Pre-order traversal\n";

cout << "4. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter the value to insert: ";

cin >> value;

threadedTree.insert(value);

break;

case 2:

threadedTree.threadInOrder();

cout << "Tree threaded in-order.\n";

break;

case 3:

cout << "Pre-order traversal: ";

threadedTree.preOrderTraversal();

break;

case 4:

cout << "Exiting...\n";

break;

default:

cout << "Invalid choice. Please try again.\n";

}

} while (choice != 4);

return 0;

}

#include <iostream>

#include <vector>

#include <algorithm>

#include <queue>

using namespace std;

struct DisjointSet {

vector<int> parent, rank;

DisjointSet(int n) : parent(n), rank(n, 0) {

iota(parent.begin(), parent.end(), 0);

}

int find(int u) {

return parent[u] == u ? u : (parent[u] = find(parent[u]));

}

void unionSets(int u, int v) {

int rootU = find(u), rootV = find(v);

if (rootU != rootV) {

if (rank[rootU] < rank[rootV]) swap(rootU, rootV);

parent[rootV] = rootU;

rank[rootU] += (rank[rootU] == rank[rootV]);

}

}

};

class GraphAdjList {

public:

int vertices;

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

GraphAdjList(int v) : vertices(v), adjList(v) {}

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

adjList[u].emplace\_back(v, weight);

adjList[v].emplace\_back(u, weight);

}

};

vector<pair<int, pair<int, int>>> kruskalMST(GraphAdjList& graph) {

vector<pair<int, pair<int, int>>> result;

DisjointSet ds(graph.vertices);

vector<pair<int, pair<int, int>>> edges;

for (int u = 0; u < graph.vertices; ++u)

for (const auto& [v, w] : graph.adjList[u])

edges.push\_back({w, {u, v}});

sort(edges.begin(), edges.end());

for (const auto& [w, uv] : edges) {

int u = uv.first, v = uv.second;

if (ds.find(u) != ds.find(v)) {

result.push\_back({w, {u, v}});

ds.unionSets(u, v);

}

}

return result;

}

vector<pair<int, int>> primMST(GraphAdjList& graph) {

vector<pair<int, int>> result;

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

vector<bool> visited(graph.vertices, false);

pq.push({0, 0});

while (!pq.empty()) {

int u = pq.top().second;

pq.pop();

if (visited[u]) continue;

visited[u] = true;

for (const auto& [v, w] : graph.adjList[u]) {

if (!visited[v]) {

pq.push({w, v});

result.push\_back({u, v});

}

}

}

return result;

}

void printMenu() {

cout << "Menu:\n1. Add Edge to Graph\n2. Find MST (Kruskal)\n3. Find MST (Prim)\n4. Exit\n";

}

int main() {

GraphAdjList campusGraph(5);

int choice;

do {

printMenu();

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1: {

int u, v, weight;

cout << "Enter source vertex (u): ";

cin >> u;

cout << "Enter destination vertex (v): ";

cin >> v;

cout << "Enter weight: ";

cin >> weight;

campusGraph.addEdge(u, v, weight);

break;

}

case 2: {

auto kruskalResult = kruskalMST(campusGraph);

cout << "Minimum Spanning Tree using Kruskal's algorithm:\n";

for (const auto& [w, uv] : kruskalResult)

cout << "Edge: " << uv.first << " - " << uv.second << " Weight: " << w << "\n";

break;

}

case 3: {

GraphAdjList campusGraphPrim(campusGraph.vertices);

auto primResult = primMST(campusGraphPrim);

cout << "Minimum Spanning Tree using Prim's algorithm:\n";

for (const auto& [u, v] : primResult)

cout << "Edge: " << u << " - " << v << "\n";

break;

}

case 4:

cout << "Exiting...\n";

break;

default:

cout << "Invalid choice. Please try again.\n";

}

} while (choice != 4);

return 0;

}

#include <iostream>

#include <vector>

#include <limits>

#include <queue>

#include <algorithm>

using namespace std;

const int INF = numeric\_limits<int>::max();

class CityGraph {

public:

int landmarks;

vector<vector<int>> adjacencyMatrix;

vector<vector<pair<int, int>>> adjacencyList;

CityGraph(int n) : landmarks(n), adjacencyMatrix(n, vector<int>(n, INF)), adjacencyList(n) {}

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

adjacencyMatrix[u][v] = distance;

adjacencyMatrix[v][u] = distance;

adjacencyList[u].emplace\_back(v, distance);

adjacencyList[v].emplace\_back(u, distance);

}

vector<int> dijkstraShortestPath(int source) {

vector<int> distance(landmarks, INF);

distance[source] = 0;

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

pq.push({0, source});

while (!pq.empty()) {

int u = pq.top().second;

int distU = pq.top().first;

pq.pop();

if (distU > distance[u]) continue;

for (const auto& neighbor : adjacencyList[u]) {

int v = neighbor.first;

int weight = neighbor.second;

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

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

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

}

}

}

return distance;

}

};

void printMenu() {

cout << "Menu:\n1. Add Edge\n2. Find Shortest Paths\n3. Exit\n";

}

int main() {

CityGraph city(6);

int choice;

do {

printMenu();

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1: {

int u, v, distance;

cout << "Enter source landmark (u): ";

cin >> u;

cout << "Enter destination landmark (v): ";

cin >> v;

cout << "Enter distance: ";

cin >> distance;

city.addEdge(u, v, distance);

break;

}

case 2: {

int source;

cout << "Enter the source landmark (0 to " << city.landmarks - 1 << "): ";

cin >> source;

vector<int> shortestPaths = city.dijkstraShortestPath(source);

cout << "Shortest paths from landmark " << source << ":\n";

for (int i = 0; i < city.landmarks; ++i)

cout << "To landmark " << i << ": " << shortestPaths[i] << " units\n";

break;

}

case 3:

cout << "Exiting...\n";

break;

default:

cout << "Invalid choice. Please try again.\n";

}

} while (choice != 3);

return 0;

}

#include <iostream>

#include <vector>

using namespace std;

void heapify(vector<int>& arr, int n, int i) {

int largest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

if (left < n && arr[left] > arr[largest])

largest = left;

if (right < n && arr[right] > arr[largest])

largest = right;

if (largest != i) {

swap(arr[i], arr[largest]);

heapify(arr, n, largest);

}

}

void heapSort(vector<int>& arr) {

int n = arr.size();

for (int i = n / 2 - 1; i >= 0; --i)

heapify(arr, n, i);

for (int i = n - 1; i > 0; --i) {

swap(arr[0], arr[i]);

heapify(arr, i, 0);

}

}

int main() {

vector<int> values = {12, 11, 13, 5, 6, 7};

int n = values.size();

cout << "Original array: ";

for (int i : values)

cout << i << " ";

cout << "\n";

heapSort(values);

cout << "Sorted array: ";

for (int i : values)

cout << i << " ";

cout << "\n";

return 0;

}

#include <fstream>

#include <iostream>

#include <iomanip>

using namespace std;

struct Student {

int rollNumber;

string name;

char division;

string address;

};

void writeStudentData(const Student& student) {

ofstream outFile("student\_database.txt", ios::app);

outFile << student.rollNumber << " " << student.name << " " << student.division << " " << student.address << "\n";

outFile.close();

}

void displayStudentData(int rollNumber) {

ifstream inFile("student\_database.txt");

Student student;

bool found = false;

while (inFile >> student.rollNumber >> student.name >> student.division >> ws && getline(inFile, student.address)) {

if (student.rollNumber == rollNumber) {

found = true;

cout << "Roll Number: " << student.rollNumber << "\n";

cout << "Name: " << student.name << "\n";

cout << "Division: " << student.division << "\n";

cout << "Address: " << student.address << "\n";

break;

}

}

if (!found) {

cout << "Student with Roll Number " << rollNumber << " not found.\n";

}

inFile.close();

}

void deleteStudentData(int rollNumber) {

ifstream inFile("student\_database.txt");

ofstream tempFile("temp.txt");

Student student;

bool found = false;

while (inFile >> student.rollNumber >> student.name >> student.division >> ws && getline(inFile, student.address)) {

if (student.rollNumber == rollNumber) {

found = true;

cout << "Student with Roll Number " << rollNumber << " has been deleted.\n";

} else {

tempFile << student.rollNumber << " " << student.name << " " << student.division << " " << student.address << "\n";

}

}

if (!found) {

cout << "Student with Roll Number " << rollNumber << " not found.\n";

}

inFile.close();

tempFile.close();

remove("student\_database.txt");

rename("temp.txt", "student\_database.txt");

}

int main() {

int choice;

do {

cout << "Menu:\n";

cout << "1. Add Student\n";

cout << "2. Display Student\n";

cout << "3. Delete Student\n";

cout << "4. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1: {

Student student;

cout << "Enter Roll Number: ";

cin >> student.rollNumber;

cout << "Enter Name: ";

cin.ignore(); // consume newline left in the buffer

getline(cin, student.name);

cout << "Enter Division: ";

cin >> student.division;

cout << "Enter Address: ";

cin.ignore(); // consume newline left in the buffer

getline(cin, student.address);

writeStudentData(student);

break;

}

case 2: {

int rollNumber;

cout << "Enter Roll Number to display: ";

cin >> rollNumber;

displayStudentData(rollNumber);

break;

}

case 3: {

int rollNumber;

cout << "Enter Roll Number to delete: ";

cin >> rollNumber;

deleteStudentData(rollNumber);

break;

}

case 4:

cout << "Exiting...\n";

break;

default:

cout << "Invalid choice. Please try again.\n";

}

} while (choice != 4);

return 0;

}