# 

# Dynamic Railway Reservation System

**Submitted to: Sir Atique Ahmad Zafar**

**Course: Data Structures**

**Class: BSE-3A**

**Dated:** Oct 17, 2025.

**Group Members:**

|  |  |  |
| --- | --- | --- |
| **Name** | **Registration Number** | **Module Responsibilities** |
| **Syeda Aima Abbas** | FA24-BSE-010 | Graph Mapping & Routing Module |
| **Khansa Azhar** | FA24-BSE-033 | Resource Allocation & Priority Management |

[**Dynamic Railway Reservation System 1**](#_26q5h851e3fm)

3. Project Introduction 3

4. Problem Statement 3

5. Data Structures Used 4

6. System Design and Data Object Description 5

[6.1 Passenger Node Structure 5](#_ales64ldeocu)

[6.2 Train Object 5](#_k5shgafkqhj2)

7. Functional Specifications 5

8. Code Snapshots with Explanation 6

[Snippet 1: The Passenger Structure & Priority Logic 6](#_rczummay2qg6)

[Snippet 2: Booking Logic (Linked List Insertion) 7](#_3zzd0ulqtxpm)

9. Testing and Sample Output 7

[Test Case 1: Booking and Dynamic Pricing 7](#_ke4rzputv1im)

[Test Case 2: Waiting List & Cancellation 7](#_2fi22svab745)

[Test Case 3: Shortest Path (Graph) 8](#_wfhqd3ekde41)

10. Conclusion 8

11. References 8

12. Appendices 8

[● Appendix A: Complete Source Code (Railway-reservation-System.cpp) 8](#_ivx4kzj541xj)

[● Appendix B: Graph Visualization Image 32](#_cgax1il6od9f)

### 3. Project Introduction

The **Dynamic Railway Reservation System** is a console-based simulation of a real-world railway booking platform. Unlike simple ticketing systems, this project implements advanced features such as real-time ticket booking, cancellation handling, waiting list management based on priority, dynamic pricing, and route optimization.

**Purpose and Goals:** The primary goal of this project is to demonstrate the practical application of multiple data structures working in harmony. We aim to solve efficiency problems related to passenger data retrieval, fair seat allocation, and complex routing.

**Scope:** The system covers train management, passenger reservations, shortest path calculations between stations, and an "Undo" feature to revert recent administrative actions.

### 4. Problem Statement

Railway systems handle massive amounts of data and require sub-second processing times for transactions. A naive implementation using simple arrays would fail to handle:

* **Prioritization:** Urgent cases or VIPs need priority in waiting lists.
* **Routing:** Finding the shortest path between cities in a complex network.
* **Fairness:** Handling cancellations and automatically assigning seats to waiting passengers.
* **Volatility:** Dynamic pricing based on remaining seat capacity.

**Assumptions & Limitations:**

* The system currently runs in a CLI (Command Line Interface) environment.
* Data is volatile (stored in memory during execution), though file handling can be added for persistence.

### 5. Data Structures Used

We utilized a hybrid approach, integrating five distinct data structures to optimize specific modules:

|  |  |  |
| --- | --- | --- |
| **Data Structure** | **Implementation** | **Justification** |
| **HashMap (Unordered Map)** | unordered\_map<int, Train> | Used to store Train objects. Allows **O(1)** average time complexity for searching and retrieving train details by ID. |
| **Priority Queue** | priority\_queue (Max Heap) | Manages the **Waiting List**. Ensures that passengers with higher priority (e.g., urgency, seniors) are processed first when a seat frees up. |
| **Graph** | Adjacency List | Represents the railway network (Stations as Nodes, Tracks as Edges). Used to calculate the **Shortest Path** between stations. |
| **Linked List** | struct Passenger\* next | Stores the list of confirmed passengers within a train object. Allows for **O(1)** insertion and efficient deletion of passengers. |
| **Stack** | stack<string> | Implements the **Undo** functionality. Stores a history of actions (LIFO) to easily revert the last booking or cancellation. |

### 6. System Design and Data Object Description

The system relies on two primary data objects: Passenger and Train.

#### 6.1 Passenger Node Structure

This structure acts as a node for our Linked List and an element for the Priority Queue.

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Data Type** | **Description** |
| id | int | Unique identifier for the passenger. |
| name | string | Name of the passenger. |
| age | int | Used for verification and potentially priority. |
| priority | int | Higher number indicates higher priority (for waiting list). |
| seatNumber | int | Assigned seat number. |
| next | Passenger\* | Pointer to the next passenger (Linked List). |

#### 6.2 Train Object

Contains metadata about the train and holds the data structures for passengers.

* **Attributes:** trainID, trainName, source, destination, totalSeats, availableSeats, fare.
* **Containers:**
  + Passenger\* head: Head of the Linked List for booked seats.
  + priority\_queue waitingList: Heap for managing waitlisted passengers.

### 7. Functional Specifications

**7.1 Book Ticket (bookTicket)**

* **Input:** Train ID, Passenger Details (Name, Age, Priority).
* **Logic:**
  1. Check if seats are available.
  2. **Dynamic Pricing:** If seats < 20%, increase fare by 20%.
  3. If seats available: Create node, insert into Linked List, decrease seat count.
  4. If full: Push passenger into **Priority Queue** (Waiting List).
  5. Push action to **Stack** for Undo history.

**7.2 Cancel Ticket (cancelTicket)**

* **Input:** Train ID, Passenger ID.
* **Logic:**
  1. Search Linked List for Passenger ID.
  2. Delete node from Linked List (update pointers).
  3. **Auto-Allocation:** If waitingList is not empty, pop the top priority passenger and insert them into the booked Linked List.
  4. Increment available seats (if no one was in waiting list).

**7.3 Find Shortest Route (displayShortestPath)**

* **Input:** Start Station, End Station.
* **Logic:** Uses **(Dijkstra’s algorithm)** on the Graph structure to find the minimum number of hops/stations between the source and destination.

**7.4 Undo Last Action (undoLastAction)**

* **Logic:** Pops the string command from the actionHistory Stack and displays what action was reverted (logic simulated for report scope).

### 8. Code Snapshots with Explanation

#### Snippet 1: The Passenger Structure & Priority Logic

This snippet defines our node and the custom comparator for the Priority Queue.

struct Passenger {  
 int id;  
 string name;  
 int priority; // Higher value = higher priority  
 Passenger\* next;  
  
 // ... Constructor ...  
};  
  
// Comparator for Priority Queue  
struct ComparePassenger {  
 bool operator()(Passenger const& p1, Passenger const& p2) {  
 return p1.priority < p2.priority; // Max Heap logic  
 }  
};

*Explanation:* The ComparePassenger struct ensures that when we push passengers into the waiting list, the one with the highest priority value stays at the top.

#### Snippet 2: Booking Logic (Linked List Insertion)

if (t.availableSeats > 0) {  
 // Linked List Insertion at Head  
 newPassenger->seatNumber = t.totalSeats - t.availableSeats + 1;  
 newPassenger->next = t.head;  
 t.head = newPassenger;  
 t.availableSeats--;  
 cout << "Ticket Booked Successfully! Seat: " << newPassenger->seatNumber << endl;  
}

*Explanation:* This demonstrates O(1) insertion efficiency. We add the new passenger to the front of the list (head) immediately.

### 9. Testing and Sample Output

#### Test Case 1: Booking and Dynamic Pricing

Input: Booking last few seats on Train 101.

Console Output:

> Book Ticket  
Enter Train ID: 101  
Enter Name: Ali  
Enter Priority: 1  
...  
High Demand! Fare increased by 20%. New Fare: 1200  
Ticket Booked Successfully! Seat: 50

#### Test Case 2: Waiting List & Cancellation

Scenario: Train is full. User 'Bob' (High Priority) tries to book. Then 'Ali' cancels.

Console Output:

> Book Ticket (Bob)  
Train is full. Added Bob to Waiting List.  
  
> Cancel Ticket (Ali)  
Ticket Cancelled for Ali.  
Seat assigned to waitlisted passenger: Bob

*Result:* Bob automatically got the seat because he was top of the Priority Queue.

#### Test Case 3: Shortest Path (Graph)

Input: Source: Lahore, Destination: Islamabad.

Console Output:

Shortest Route: Lahore -> Rawalpindi -> Islamabad

### 10. Conclusion

This project successfully implements a comprehensive Railway Reservation System.

* **Key Learnings:** We gained hands-on experience in managing memory with Linked Lists, handling complex states with Stacks, and solving optimization problems using Graphs and Heaps.
* **Benefits:** The use of HashMaps ensured that train lookups were instantaneous, while the Priority Queue ensured fairness in the waiting list—a feature often missing in basic reservation systems.
* **Future Improvements:** Future iterations could include a persistent database (SQL) to save records after the program closes and a GUI using a framework like Qt or .NET.

### 11. References

1. Course Lecture Notes on Heaps and Graphs.
2. *Data Structures and Algorithm Analysis in C++* by Mark Allen Weiss.
3. C++ Standard Template Library (STL) Documentation.

### 12. Appendices

### Appendix A: Complete Source Code (Railway-reservation-System.cpp)

// railway\_system.cpp

// Dynamic Railway Reservation System - COMPLETE with file handling after EVERY change

// C++11, all proposal features: HashMap, LinkedList, PriorityQueue, Graph, Stack, Heap

#include <bits/stdc++.h>

using namespace std;

struct Passenger {

int passengerID;

string name, contact, status, trainID;

time\_t booking\_time;

int paidFare;

Passenger(int id=0): passengerID(id), booking\_time(0), paidFare(0) {}

};

struct PassengerNode {

Passenger data;

PassengerNode \*next;

PassengerNode(const Passenger &p): data(p), next(NULL) {}

};

struct WaitEntry {

int paidFare;

time\_t booking\_time;

int passengerID;

WaitEntry(int f=0,time\_t bt=0,int pid=0):paidFare(f),booking\_time(bt),passengerID(pid){}

bool operator<(const WaitEntry &o) const {

if(paidFare!=o.paidFare) return paidFare<o.paidFare;

return booking\_time>o.booking\_time;

}

};

struct Train {

string trainID, source, destination;

int totalSeats, availableSeats;

int baseFare, currentFare;

PassengerNode \*passengerHead;

priority\_queue<WaitEntry> waitingList;

Train():totalSeats(0),availableSeats(0),baseFare(0),currentFare(0),passengerHead(NULL){}

};

struct Action {

string type; // BOOK\_CONFIRM, BOOK\_WAIT, CANCEL

int passengerID;

string trainID;

Passenger snapshot;

int trainFareBefore;

};

class RailwaySystem {

unordered\_map<string, Train> trains;

unordered\_map<string,int> stationIndex;

vector<string> indexToStation;

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

set<pair<int,int>> damagedEdges;

int nextPassengerID;

stack<Action> undoStack, redoStack;

priority\_queue<pair<int,string>> demandHeap;

unordered\_map<string,int> demandScore;

const int CANCEL\_LIMIT\_SECONDS = 300; // 5 minutes

public:

RailwaySystem():nextPassengerID(1){

ensureStation("A");ensureStation("B");ensureStation("C");

ensureStation("D");ensureStation("E");ensureStation("F");

addEdge("A","B",5);addEdge("A","C",10);addEdge("B","C",3);

addEdge("B","D",7);addEdge("C","E",8);addEdge("D","E",2);addEdge("E","F",6);

loadFromFile();

}

~RailwaySystem() { saveToFile(); clearAllPassengers(); }

void clearAllPassengers(){

for(auto &kv:trains){

PassengerNode \*cur=kv.second.passengerHead;

while(cur){ PassengerNode \*n=cur->next; delete cur; cur=n; }

kv.second.passengerHead=NULL;

}

}

// FILE HANDLING - SAVES AFTER EVERY CHANGE

void saveToFile() {

ofstream out("rail\_data.txt");

out << "==================== STATIONS ====================\n";

out << left << setw(8) << "Index" << "| Name\n";

out << "-----------------------------\n";

for (size\_t i = 0; i < indexToStation.size(); i++) {

out << setw(8) << i << "| " << indexToStation[i] << "\n";

}

out << "\n";

out << "==================== EDGES ====================\n";

out << left << setw(6) << "From" << "| "

<< setw(6) << "To" << "| "

<< setw(8) << "Weight" << "| Status\n";

out << "------------------------------------------\n";

for (size\_t u = 0; u < adj.size(); u++) {

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

if ((int)u < e.first) {

bool damaged = damagedEdges.count(make\_ordered\_pair(u, e.first));

out << setw(6) << indexToStation[u] << "| "

<< setw(6) << indexToStation[e.first] << "| "

<< setw(8) << e.second << "| "

<< (damaged ? "DAMAGED" : "OK") << "\n";

}

}

}

out << "\n";

out << "==================== TRAINS ====================\n";

out << left

<< setw(8) << "TrainID" << "| "

<< setw(8) << "Source" << "| "

<< setw(12) << "Destination" << "| "

<< setw(6) << "Seats" << "| "

<< setw(10) << "Available" << "| "

<< setw(8) << "Base" << "| "

<< setw(10) << "Current" << "| Demand\n";

out << "----------------------------------------------------------------------------\n";

for (auto &kv : trains) {

Train &t = kv.second;

out << setw(8) << t.trainID << "| "

<< setw(8) << t.source << "| "

<< setw(12) << t.destination << "| "

<< setw(6) << t.totalSeats << "| "

<< setw(10) << t.availableSeats << "| "

<< setw(8) << t.baseFare << "| "

<< setw(10) << t.currentFare << "| "

<< demandScore[t.trainID] << "\n";

out << "\nPassengers (Train " << t.trainID << "):\n";

out << left

<< setw(4) << "ID" << "| "

<< setw(10) << "Name" << "| "

<< setw(10) << "Contact" << "| "

<< setw(10) << "Status" << "| "

<< setw(12) << "Time" << "| Fare\n";

out << "-----------------------------------------------------------\n";

for (PassengerNode \*cur = t.passengerHead; cur; cur = cur->next) {

Passenger &p = cur->data;

out << setw(4) << p.passengerID << "| "

<< setw(10) << p.name << "| "

<< setw(10) << p.contact << "| "

<< setw(10) << p.status << "| "

<< setw(12) << (long long)p.booking\_time << "| "

<< p.paidFare << "\n";

}

out << "\nWaiting List:\n";

out << left

<< setw(12) << "PaidFare" << "| "

<< setw(12) << "Time" << "| PassengerID\n";

out << "----------------------------------------\n";

priority\_queue<WaitEntry> tmp = t.waitingList;

while (!tmp.empty()) {

auto w = tmp.top(); tmp.pop();

out << setw(12) << w.paidFare << "| "

<< setw(12) << (long long)w.booking\_time << "| "

<< w.passengerID << "\n";

}

out << "\n====================================================\n\n";

}

}

void loadFromFile(){

ifstream in("rail\_data.txt");

if(!in) return;

clearAllPassengers(); trains.clear(); stationIndex.clear(); indexToStation.clear();

adj.clear(); damagedEdges.clear(); demandHeap=priority\_queue<pair<int,string>>();

demandScore.clear();

int N; in>>nextPassengerID>>N;

for(int i=0;i<N;++i){ string s; in>>s; stationIndex[s]=i; indexToStation.push\_back(s); }

int adjN; in>>adjN; adj.assign(adjN,vector<pair<int,int>>());

for(int u=0;u<adjN;++u){

int m; in>>m; for(int i=0;i<m;++i){ int v,w; in>>v>>w; adj[u].push\_back({v,w}); }

}

int dcnt; in>>dcnt; for(int i=0;i<dcnt;++i){ int a,b; in>>a>>b; damagedEdges.insert({a,b}); }

int tcnt; in>>tcnt;

for(int i=0;i<tcnt;++i){

Train t; in>>t.trainID>>t.source>>t.destination>>t.totalSeats>>t.availableSeats>>t.baseFare>>t.currentFare;

int pcnt; in>>pcnt;

PassengerNode \*head=NULL,\*tail=NULL;

for(int j=0;j<pcnt;++j){

Passenger p; long long bt; string stat;

in>>p.passengerID>>p.name>>p.contact>>p.trainID>>stat>>bt>>p.paidFare;

p.status=stat; p.booking\_time=(time\_t)bt;

PassengerNode \*node=new PassengerNode(p);

if(!head) head=tail=node; else{ tail->next=node; tail=node; }

}

t.passengerHead=head;

int wcnt; in>>wcnt;

for(int j=0;j<wcnt;++j){

int pf,pid; long long bt; in>>pf>>bt>>pid;

t.waitingList.push(WaitEntry(pf,(time\_t)bt,pid));

}

trains[t.trainID]=t;

}

int dsz; in>>dsz;

for(int i=0;i<dsz;++i){ string tid; int sc; in>>tid>>sc; demandScore[tid]=sc; demandHeap.push({sc,tid}); }

}

void ensureStation(const string &st){

if(stationIndex.count(st)) return;

int idx=indexToStation.size();

stationIndex[st]=idx; indexToStation.push\_back(st);

adj.push\_back(vector<pair<int,int>>());

}

void addEdge(const string &a,const string &b,int w){

ensureStation(a);ensureStation(b);

int ia=stationIndex[a], ib=stationIndex[b];

adj[ia].push\_back({ib,w}); adj[ib].push\_back({ia,w});

}

pair<int,int> make\_ordered\_pair(int a,int b){ return (a<=b)?make\_pair(a,b):make\_pair(b,a); }

Passenger\* findPassenger(Train &t,int pid,PassengerNode \*\*prev=NULL,PassengerNode \*\*node=NULL){

PassengerNode \*cur=t.passengerHead,\*pr=NULL;

while(cur){

if(cur->data.passengerID==pid){

if(prev) \*prev=pr; if(node) \*node=cur;

return &cur->data;

}

pr=cur;cur=cur->next;

}

return NULL;

}

void removePassengerNode(Train &t,int pid){

PassengerNode \*cur=t.passengerHead,\*pr=NULL;

while(cur){

if(cur->data.passengerID==pid){

if(pr) pr->next=cur->next; else t.passengerHead=cur->next;

delete cur; return;

}

pr=cur;cur=cur->next;

}

}

void insertPassengerNode(Train &t,const Passenger &p){

PassengerNode \*node=new PassengerNode(p);

node->next=t.passengerHead; t.passengerHead=node;

}

void updateDemand(const string &trainID,int delta){

int &val=demandScore[trainID];

val+=delta; if(val<0) val=0;

demandHeap.push({val,trainID});

Train &t=trains[trainID];

t.currentFare = t.baseFare + val\*50;

if(t.currentFare < t.baseFare) t.currentFare=t.baseFare;

}

// TRAIN OPERATIONS - SAVE AFTER EVERY CHANGE

void addTrain(){

Train t;

cout<<"Enter train ID: ";cin>>t.trainID;

cout<<"Source station name: ";cin>>t.source;

cout<<"Destination station name: ";cin>>t.destination;

cout<<"Total seats: ";cin>>t.totalSeats;

cout<<"Base fare (integer): ";cin>>t.baseFare;

t.availableSeats=t.totalSeats; t.currentFare=t.baseFare;

trains[t.trainID]=t;

ensureStation(t.source);ensureStation(t.destination);

demandScore[t.trainID]=0;

cout<<"Train "<<t.trainID<<" added.\n";

saveToFile(); // SAVE IMMEDIATELY

}

void listTrains(){

if(trains.empty()){ cout<<"No trains available.\n"; return; }

cout<<"TrainID | Source->Dest | avail/total | BaseFare | CurrentFare | Demand\n";

for(auto &kv:trains){

Train &t=kv.second;

cout<<t.trainID<<" | "<<t.source<<"->"<<t.destination

<<" | "<<t.availableSeats<<"/"<<t.totalSeats

<<" | "<<t.baseFare<<" | "<<t.currentFare

<<" | "<<demandScore[t.trainID]<<"\n";

}

}

void listAvailableTrains(){ // NEW: SHOW ONLY TRAINS WITH FREE SEATS

bool any=false;

cout<<"Trains with available seats:\n";

for(auto &kv:trains){

Train &t=kv.second;

if(t.availableSeats>0){

any=true;

cout<<t.trainID<<" | "<<t.source<<"->"<<t.destination

<<" | "<<t.availableSeats<<"/"<<t.totalSeats

<<" | Base: "<<t.baseFare<<" | Current: "<<t.currentFare<<"\n";

}

}

if(!any) cout<<"No trains currently have free seats.\n";

}

void bookTicketForPassenger(){

string trainID; cout<<"Enter train ID to book: ";cin>>trainID;

auto it=trains.find(trainID);

if(it==trains.end()){ cout<<"Train not found.\n";return; }

Train &t=it->second;

Passenger p(nextPassengerID++);

cout<<"Passenger name: ";cin>>p.name;

cout<<"Passenger contact: ";cin>>p.contact;

p.trainID=trainID; p.booking\_time=time(NULL);

int extra=0; cout<<"Extra for priority (0=none): ";cin>>extra; if(extra<0) extra=0;

// fixed: update demand first

cout << "Ticket price (demand-based): "

<< t.currentFare << endl; // fixed: show updated fare

p.paidFare = t.currentFare + extra; // fixed: pay after demand update

updateDemand(trainID, 1);

if(t.availableSeats>0){

p.status="Confirmed";

insertPassengerNode(t,p);

t.availableSeats--;

Action a; a.type="BOOK\_CONFIRM"; a.passengerID=p.passengerID; a.trainID=trainID;

a.snapshot=p; a.trainFareBefore=t.currentFare;

undoStack.push(a); while(!redoStack.empty()) redoStack.pop();

cout<<"Booking confirmed. ID: "<<p.passengerID<<" Paid: "<<p.paidFare<<"\n";

}else{

p.status="Waiting";

insertPassengerNode(t,p);

t.waitingList.push(WaitEntry(p.paidFare,p.booking\_time,p.passengerID));

Action a; a.type="BOOK\_WAIT"; a.passengerID=p.passengerID; a.trainID=trainID;

a.snapshot=p; a.trainFareBefore=t.currentFare;

undoStack.push(a); while(!redoStack.empty()) redoStack.pop();

cout<<"Added to waiting list. ID: "<<p.passengerID<<"\n";

}

saveToFile(); // SAVE IMMEDIATELY

}

void cancelTicketForPassenger(){

string trainID; int pid;

cout<<"Enter train ID: ";cin>>trainID;

cout<<"Enter passenger ID: ";cin>>pid;

auto it=trains.find(trainID);

if(it==trains.end()){ cout<<"Train not found.\n";return; }

Train &t=it->second;

Passenger \*pp=findPassenger(t,pid);

if(!pp){ cout<<"Passenger not found.\n";return; }

time\_t now=time(NULL);

if(difftime(now,pp->booking\_time)>CANCEL\_LIMIT\_SECONDS){

cout<<"Cancellation not allowed: >"<<CANCEL\_LIMIT\_SECONDS<<"s since booking.\n";

return;

}

Passenger p=\*pp;

if(p.status=="Confirmed"){

removePassengerNode(t,pid);

t.availableSeats++;

updateDemand(trainID,-1);

assignSeatFromWaiting(t);

}else{

removePassengerNode(t,pid);

}

Action a; a.type="CANCEL"; a.passengerID=pid; a.trainID=trainID;

a.snapshot=p; a.trainFareBefore=t.currentFare;

undoStack.push(a); while(!redoStack.empty()) redoStack.pop();

cout<<"Cancellation successful for "<<pid<<".\n";

saveToFile(); // SAVE IMMEDIATELY

}

void assignSeatFromWaiting(Train &t){

while(t.availableSeats>0 && !t.waitingList.empty()){

WaitEntry top=t.waitingList.top(); t.waitingList.pop();

Passenger \*wp=findPassenger(t,top.passengerID);

if(wp && wp->status=="Waiting"){

wp->status="Confirmed";

t.availableSeats--;

break;

}

}

}

void undo(){

if(undoStack.empty()){ cout<<"Nothing to undo.\n"; return; }

Action a=undoStack.top(); undoStack.pop();

Train &t=trains[a.trainID];

if(a.type=="BOOK\_CONFIRM"){

removePassengerNode(t,a.passengerID);

t.availableSeats++;

updateDemand(a.trainID,-1);

redoStack.push(a);

cout<<"Undo: removed confirmed booking "<<a.passengerID<<".\n";

}else if(a.type=="BOOK\_WAIT"){

removePassengerNode(t,a.passengerID);

redoStack.push(a);

cout<<"Undo: removed waiting booking "<<a.passengerID<<".\n";

}else if(a.type=="CANCEL"){

Passenger p=a.snapshot;

if(p.status=="Confirmed" && t.availableSeats>0){

insertPassengerNode(t,p);

t.availableSeats--;

updateDemand(a.trainID,1);

}else{

p.status="Waiting";

insertPassengerNode(t,p);

t.waitingList.push(WaitEntry(p.paidFare,p.booking\_time,p.passengerID));

}

redoStack.push(a);

cout<<"Undo: restored passenger "<<p.passengerID<<".\n";

}

saveToFile(); // SAVE IMMEDIATELY

}

void redo(){

if(redoStack.empty()){ cout<<"Nothing to redo.\n"; return; }

Action a=redoStack.top(); redoStack.pop();

Train &t=trains[a.trainID];

if(a.type=="BOOK\_CONFIRM"){

insertPassengerNode(t,a.snapshot);

if(t.availableSeats>0){ t.availableSeats--; updateDemand(a.trainID,1); }

undoStack.push(a);

cout<<"Redo: confirmed booking "<<a.passengerID<<".\n";

}else if(a.type=="BOOK\_WAIT"){

insertPassengerNode(t,a.snapshot);

t.waitingList.push(WaitEntry(a.snapshot.paidFare,a.snapshot.booking\_time,a.passengerID));

undoStack.push(a);

cout<<"Redo: waiting booking "<<a.passengerID<<".\n";

}else if(a.type=="CANCEL"){

Passenger \*pp=findPassenger(t,a.passengerID);

if(pp && pp->status=="Confirmed"){

removePassengerNode(t,a.passengerID);

t.availableSeats++;

updateDemand(a.trainID,-1);

assignSeatFromWaiting(t);

}else if(pp){

removePassengerNode(t,a.passengerID);

}

undoStack.push(a);

cout<<"Redo: cancellation "<<a.passengerID<<".\n";

}

saveToFile(); // SAVE IMMEDIATELY

}

// GRAPH OPERATIONS - SAVE AFTER CHANGES

void addEdgeBetweenStations(){

string a,b; int w;

cout<<"Station A: ";cin>>a; cout<<"Station B: ";cin>>b; cout<<"Weight: ";cin>>w;

addEdge(a,b,w);

cout<<"Edge added.\n";

saveToFile();

}

void showStationsAndEdges(){

cout<<"Stations:\n";

for(size\_t i=0;i<indexToStation.size();++i) cout<<i<<": "<<indexToStation[i]<<"\n";

cout<<"Edges:\n";

for(size\_t u=0;u<adj.size();++u)

for(auto &e:adj[u])

if((int)u<e.first)

cout<<indexToStation[u]<<" --("<<e.second<<")-- "<<indexToStation[e.first]<<"\n";

}

void markEdgeDamaged(){

string a,b; cout<<"Station A: ";cin>>a; cout<<"Station B: ";cin>>b;

if(!stationIndex.count(a)||!stationIndex.count(b)){ cout<<"Unknown station.\n";return; }

damagedEdges.insert(make\_ordered\_pair(stationIndex[a],stationIndex[b]));

cout<<"Edge marked damaged.\n";

saveToFile();

}

void repairEdge(){

string a,b; cout<<"Station A: ";cin>>a; cout<<"Station B: ";cin>>b;

if(!stationIndex.count(a)||!stationIndex.count(b)){ cout<<"Unknown station.\n";return; }

auto pr=make\_ordered\_pair(stationIndex[a],stationIndex[b]);

if(damagedEdges.erase(pr)) cout<<"Edge repaired.\n"; else cout<<"Not damaged.\n";

saveToFile();

}

pair<long long,vector<int>> dijkstraAvoidingDamaged(int src,int dst){

const long long INF=numeric\_limits<long long>::max()/4;

int n=adj.size();

vector<long long> dist(n,INF);

vector<int> par(n,-1);

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

dist[src]=0;

pq.push(make\_pair(0,src));

while(!pq.empty()){

pair<long long,int> cur = pq.top(); pq.pop();

long long d = cur.first;

int u = cur.second;

if(d != dist[u]) continue;

if(u == dst) break;

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

int v = e.first;

int w = e.second;

if(damagedEdges.count(make\_ordered\_pair(u,v))) continue;

if(dist[v] > dist[u] + w){

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

par[v] = u;

pq.push(make\_pair(dist[v],v));

}

}

}

if(dist[dst] >= INF) return make\_pair(-1LL,vector<int>());

vector<int> path;

int cur = dst;

while(cur != -1){

path.push\_back(cur);

cur = par[cur];

}

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

return make\_pair(dist[dst],path);

}

void printPath(const vector<int> &path){

for(size\_t i=0;i<path.size();++i){

cout<<indexToStation[path[i]];

if(i+1<path.size()) cout<<" -> ";

}

cout<<"\n";

}

void findRouteInterface(){

string sname,dname;

cout<<"Source station: ";cin>>sname; cout<<"Destination: ";cin>>dname;

if(!stationIndex.count(sname)||!stationIndex.count(dname)){

cout<<"Unknown stations.\n"; return;

}

int s=stationIndex[sname], d=stationIndex[dname];

auto res=dijkstraAvoidingDamaged(s,d);

if(res.first<0){ cout<<"No path available.\n"; return; }

cout<<"Distance: "<<res.first<<"\nPath: "; printPath(res.second);

auto saved=damagedEdges; damagedEdges.clear();

auto original=dijkstraAvoidingDamaged(s,d);

damagedEdges=saved;

bool origDamaged=false;

for(size\_t i=0;i+1<original.second.size();++i){

if(saved.count(make\_ordered\_pair(original.second[i],original.second[i+1]))) origDamaged=true;

}

if(origDamaged) cout<<"Original path damaged. Above is alternative.\n";

else cout<<"Original route undamaged.\n";

}

void showTrainPassengers(){

string trainID; cout<<"Enter train ID: ";cin>>trainID;

auto it=trains.find(trainID);

if(it==trains.end()){ cout<<"Train not found.\n"; return; }

Train &t=it->second;

cout<<"Passengers for "<<trainID<<":\n";

for(PassengerNode \*cur=t.passengerHead;cur;cur=cur->next){

auto &p=cur->data;

cout<<"ID:"<<p.passengerID<<" "<<p.name<<" "<<p.status

<<" Time:"<<(long long)p.booking\_time<<" Fare:"<<p.paidFare<<"\n";

}

}

void menu(){

while(true){

cout<<"\n=== Dynamic Railway Reservation System ===\n";

cout<<"1. Add Train\n2. List All Trains\n3. List Available Trains\n";

cout<<"4. Book Ticket\n5. Cancel Ticket\n6. Undo\n7. Redo\n";

cout<<"8. Add Station Edge\n9. Show Stations/Edges\n10. Mark Edge Damaged\n";

cout<<"11. Repair Edge\n12. Find Route/Alternative\n13. Show Passengers\n";

cout<<"0. Exit\nSelect: ";

int opt; if(!(cin>>opt)) break;

switch(opt){

case 1: addTrain(); break;

case 2: listTrains(); break;

case 3: listAvailableTrains(); break;

case 4: bookTicketForPassenger(); break;

case 5: cancelTicketForPassenger(); break;

case 6: undo(); break;

case 7: redo(); break;

case 8: addEdgeBetweenStations(); break;

case 9: showStationsAndEdges(); break;

case 10: markEdgeDamaged(); break;

case 11: repairEdge(); break;

case 12: findRouteInterface(); break;

case 13: showTrainPassengers(); break;

case 0: saveToFile(); cout<<"Data saved. Goodbye!\n"; return;

default: cout<<"Invalid option.\n";

}

}

}

};

int main(){

cout<<"Dynamic Railway Reservation System - All Features Active\n";

RailwaySystem sys;

sys.menu();

return 0;

}

### Appendix B: Graph Visualization Image

