

COMSATS UNIVERSITY ISLAMABAD

Project Report

For

TrainMate:

A Railway Management System

By

Hafsah Mehdi – SP22-BCT-016

Mohammad Fahad Malik – SP22-BCT-024

Supervisor

Mustafa Khattak

# INTRODUCTION:

A railway management system can be a complex application with many different components and operations. By dividing the system into smaller, more manageable parts, each of which can be implemented using an appropriate data structure, the system can be made more modular and easier to maintain and update. These data structures are also designed to efficiently manage memory. For example, linked lists can allocate memory dynamically, which allows them to efficiently manage a variable number of passengers on each train. This can help reduce memory wastage and improve the overall performance of the system. These data structures are specifically designed to efficiently manage and manipulate data in a specific way. For example, a stack is designed for LIFO operations, a queue is designed for FIFO operations, and a heap is designed for efficient priority-based operations. By using the appropriate data structure for each task, the system can perform operations more efficiently.

TrainMate is a program that efficiently handles all the processes in a Railway Management system, for both the administrators and clients.

# Problem Statement:

Railway management systems face several challenges and problems that can affect their efficiency and effectiveness. Here are some possible problems that current railway management systems may encounter:

Manual Processes: Many railway management systems still rely on manual processes such as paper-based ticketing and manual scheduling, which can be time-consuming and error-prone. This can lead to delays and inaccuracies in the system.

Lack of Automation: A lack of automation in railway management systems can also lead to inefficiencies, as tasks such as resource allocation and scheduling may not be optimized for efficiency and cost-effectiveness.

Limited Visibility: A lack of real-time visibility into the status of trains and resources can also be a problem for railway management systems. This can lead to difficulties in managing delays and disruptions and can result in a poor customer experience.

Security Concerns: Railway management systems may also face security concerns, such as cyberattacks or physical security threats, which can compromise the safety of passengers and staff and disrupt the operation of the railway system.

Overall, these challenges can impact the efficiency, safety, and reliability of railway management systems. Addressing these challenges and implementing modern technologies and practices can help improve the performance of railway systems and provide a better experience for passengers and staff.

# Problem Solution:

A railway management system can be a valuable tool for potential customers, such as railway operators and passengers, for several reasons:

Improved Efficiency: A well-designed railway management system can help improve the efficiency of a railway system by automating various tasks and improving the management of resources. This can help reduce delays, improve the accuracy of scheduling, and ultimately result in a better experience for passengers.

Improved Customer Experience: A railway management system can also improve the overall customer experience by providing features such as online ticket booking, real-time train schedules. This can help reduce waiting times and provide a more convenient and reliable service for passengers.

Data Analysis: A railway management system can also provide valuable data analysis capabilities, allowing railway operators to analyse trends, identify areas for improvement, and make changes in the system. This can help improve the overall performance and profitability of the railway system.

# Objectives:

* Reduce use of travel and expenses for ticket booking, as it can be done through the program.
* Reduce use of resources by storing all data in one place, therefore less use of memory too.
* Efficient and easy to use for the administrators, one place for checking and updating data.
* Customers will not have to wait in long queues to book tickets.

# Vision Statement:

For commuters and train administrators,

Who need a reliable and efficient train ticketing and scheduling system,

The TrainMate

Is a comprehensive train management system

That provides a user-friendly interface for booking tickets, managing train schedules, and optimizing train routes.

Unlike existing train ticketing and scheduling systems,

Our product offers real-time train tracking and scheduling updates, personalized booking options, and advanced analytics for administrators.

Our product is the primary choice for commuters and administrators seeking a streamlined, efficient, and user-friendly train management system that improves the overall efficiency, accuracy, and convenience of train travel.

# Scope:

The proposed project is a train ticketing and scheduling system, called TrainMate, which aims to provide a comprehensive and user-friendly interface for both commuters and administrators. The scope of the project includes the development of a platform that allows users to view train schedules, book tickets, and manage train routes and schedules.

The main functionalities of the TrainMate system include a train schedule viewer that displays up-to-date train schedules and arrival times, a ticket booking system that allows users to book tickets online, and a train management system that provides administrators with tools to manage train schedules, routes, and capacity.

The train schedule viewer will allow users to search for specific train schedules, view arrival and departure times, and see real-time updates for train delays or cancellations. The ticket booking system will allow users to search for available seats, select their preferred seats, and make payments online. Users will also be able to view their booking history and manage their personal information.

The train management system will provide administrators with tools to manage train schedules, routes, and capacity. This will include features such as train route optimization, real-time train tracking, and capacity planning. Administrators will also be able to view and manage train delays and cancellations and generate reports on train schedules and performance.

Overall, the TrainMate system will provide a comprehensive and user-friendly solution for managing train schedules and ticketing. It will improve the efficiency and accuracy of train travel and provide a more convenient experience for both commuters and administrators. The scope of the project will include the development of the platform and testing of its functionalities to ensure a seamless and reliable user experience.

# Modules:

Here are the modules for the proposed TrainMate project, grouped by system types:

**Client Web App:**

Train Schedule Viewer Module:

1. View train schedules and arrival times
2. Search for specific train schedules

Ticket Booking Module:

1. Search for available seats
2. Select preferred seats
3. Make payments online
4. View booking history
5. Manage information

**Admin Web App:**

Train Management Module:

1. Manage train schedules and routes
2. Optimize train routes
3. Manage train capacity and availability
4. View and manage train data

User Management Module:

1. View Train Schedules
2. Book tickets

# Limitations:

**Scalability:** As the system grows and the number of users increases, the code may become difficult to manage and maintain, which could impact the system's performance and reliability.

**Limited availability of train schedule data:** The accuracy and availability of train schedule data can vary depending on the region and transportation authority. This may limit the scope of the TrainMate system in certain areas or require additional data sourcing efforts.

**Integration with existing systems:** TrainMate may need to integrate with existing train ticketing and scheduling systems, which could pose technical challenges and require additional development effort.

**Security concerns**: TrainMate will need to store and manage sensitive user data, such as payment information and personal details. This introduces security concerns that will need to be addressed through secure coding practices, data encryption, and compliance with industry standards and regulations.

# Data Structures/Algorithms Used In this Program:

## Linked List:

### Purpose:

The Fare structure represents a fare for a specific transportation route, including the source and destination locations, fare amount, and pointers to the next and previous fares in the linked list.

The insertFare function is responsible for creating a new fare node and adding it to the linked list of fares.

The searchFare function allows searching for the fare amount based on a given source and destination.

### Operations:

* void insertFare(string source, string destination, int fare): This function creates a new Fare node and inserts it at the end of the linked list. If it is the first fare being added, it initializes both the first and last pointers. Otherwise, it updates the last pointer and adjusts the next and previous pointers accordingly.
* int searchFare(string source, string destination): This function traverses the linked list of fares, searching for a fare that matches the provided source and destination. If a matching fare is found, its fare amount is returned. If no match is found, the function returns 0.

### Usage:

To use the fare management functionality, you can call the insertFare function to add new fares to the list and the searchFare function to retrieve fare amounts based on the source and destination.

Initially, the first and last pointers are set to nullptr. As fares are inserted using the insertFare function, the linked list is constructed and maintained accordingly.

## Stacks:

### Usage 1:

### Purpose:

The purpose of the data structure is to store and manage train routes. Each train route is represented by a Train structure, which contains various attributes such as train name, source station, destination station, arrival time, departure time, distance, fare, capacity, and a stack of car numbers. The stack-based data structure allows for adding, removing, and accessing train routes.

### Operations:

* bool isempty(): This function checks if the train stack is empty and returns a boolean value accordingly.
* bool isfull(): This function checks if the train stack is full and returns a boolean value accordingly.
* void push(const Train& train): This function adds a new train route to the train stack, provided the stack is not already full.
* void pop(): This function removes the topmost train route from the stack, provided the stack is not already empty.
* Train& topElement(): This function returns a reference to the topmost train route in the stack.
* void displayStack(): This function displays all the train routes currently stored in the stack.
* void popTrainBySourceAndDestination(const string& source, const string& destination): This function pops a train route from the stack based on the given source and destination values. It removes the train route from the stack and shifts the remaining elements accordingly.

### Usage 1:

To use this data structure, you can call the provided functions accordingly. You can add train routes using push(), remove train routes using pop(), check if the stack is empty using isempty(), access the topmost train route using topElement(), display all train routes using displayStack(), and remove a train route based on source and destination using popTrainBySourceAndDestination().

You can initialize the stack by declaring an array of Train structures (Train trainStack[MAX\_SIZE];) and keeping track of the top index (int top = -1;).

Use the provided functions to interact with the train stack, add new train routes, remove train routes, and display the existing train routes.

### Usage 2:

Structure: Train

Cars Addition/Deletion from Trains.

Purpose: It stores a train’s cars in a stack so they are accessed in a suitable way, easy to add and remove cars from a train.

Operations: addCar(), removeCar().

Usage: Used to Add and Remove Cars from the Train.

## Graph (Adjacency List):

## Purpose:

Structures: Node and Train (vertices and edges of the graph)

The data structure is used to represent a graph where each station is represented as a vertex, and train schedules between stations are represented as edges.

It allows storing and managing train schedules, finding routes between stations, and performing operations related to the train network.

Operations and Usage:

* addTrain(const Train& train): Adds a train schedule to the graph. It checks if the source station already exists in the graph. If not, it creates a new node for the source station. Then, it adds an edge (train schedule) from the source to the destination station.
* deleteTrain(const string& source, const string& destination): Deletes a train schedule from the graph. It searches for the specified train schedule starting from the source station. If found, it removes the train schedule from the linked list of schedules.
* printGraph(): Prints all the train schedules in the graph. It iterates over each source station and prints the train schedules associated with it.
* searchTrain(const string& source, const string& destination): Searches for train routes between the given source and destination stations. It uses depth-first search (DFS) to traverse the graph and find all possible routes. It also calculates the shortest route based on the minimum distance and returns the fare for the shortest route.
* shortest\_route(int vertex, const string& destination, bool\* visited, const list<Train>& path, bool& found, int& shortestDistance, list<Train>& shortestRoute): A helper function used by searchTrain() to find the shortest route based on the minimum distance. It performs **a depth-first search** and keeps track of the shortest distance and route found so far.
* getShortestPathIndex(int\* dist, bool\* visited, int V): A helper function used by shortest\_route() to get the index of the vertex with the minimum distance (Dijkstra's algorithm).
* getSourceIndex(const string& source): Helper function to get the index of the source station in the adjacency list.
* getDestinationIndex(const string& destination): Helper function to get the index of the destination station in the adjacency list.

Other supporting functions include creating and deleting nodes, initializing visited arrays, and managing the adjacency list representation.

Overall, this data structure allows managing train schedules, finding routes between stations, and determining the shortest route based on distance. It utilizes a graph structure to represent the train network and implements various operations to manipulate and traverse the graph.

## Array:

Structures: trainStack, adjList

Purpose: It is used to implement the stack and adjacency list data structures.

Usage: Arrays are used to store the stack of train routes (trainStack) and the adjacency list of the graph (adjList).

## List(Dijisktra’s Algorithm/Shortest Route):

Structures: path, shortestRoute

Purpose: It is used to store the sequence of train routes during traversal and finding the shortest route.

Operations: Insertion, iteration

Usage: Lists are used to keep track of the routes and find the shortest route between source and destination stations.

Overall, these data structures are used together to represent and manage the train ticket reservation system efficiently. The linked list stores fare details, the stack stores train routes, the graph (adjacency list) represents the train network, and arrays/lists facilitate the implementation and traversal of these data structures.

## Queues:

Purpose:

The purpose of this data structure is to maintain a queue of tickets in the order they are booked. It allows adding tickets to the back (enqueue) and removing tickets from the front (dequeue) of the queue.

Operations:

* isEmpty(): Checks if the queue is empty.
* enqueue(Ticket ticket): Adds a new ticket to the back of the queue.
* dequeue(): Removes the ticket from the front of the queue.
* bookTicket(): Takes user input to book a ticket, calculates the ticket price based on train schedules, number of tickets, round trip, and class type, and enqueues the ticket.
* processTickets(): Processes all the tickets in the queue by dequeuing them one by one and printing the ticket information.

Usage:

The enqueue() operation is used to add newly booked tickets to the queue, and the dequeue() operation is used to process and remove tickets from the front of the queue. The isEmpty() function is used to check if there are any pending tickets in the queue. The bookTicket() function takes user input, calculates the ticket price, and enqueues the ticket. The processTickets() function dequeues each ticket from the queue, writes its information to an output file, and prints the ticket information to the console.

Overall, this data structure allows for the management and processing of ticket bookings in a first-in-first-out manner, ensuring that tickets are processed in the order they are booked.

## Heaps:

### Purpose:

The purpose of the TrainHeap class is to provide a data structure that efficiently maintains a collection of Train objects in a way that allows for efficient insertion, removal, and retrieval of the train with the highest capacity.

### Operations:

* void heapify(): This function performs the heapify operation on the trains vector, reorganizing its elements into a valid max heap structure based on the capacity of the trains.
* void push(Train train): This function adds a new Train object to the heap while maintaining the max heap property. The train is inserted at the appropriate position based on its capacity.
* Train pop(): This function removes and returns the Train object with the highest capacity from the heap while maintaining the max heap property. The root element is replaced with the last element in the heap, and the heap is reorganized accordingly.
* int size(): This function returns the number of trains currently stored in the heap.
* bool isEmpty() const: This function checks if the heap is empty and returns true if it is, false otherwise.

### Usage:

To use the TrainHeap class, you need to create an instance of it and then call its member functions as needed. You can push trains into the heap using the push function, retrieve the train with the highest capacity using the pop function, check the number of trains using the size function, and check if the heap is empty using the isEmpty function. The heapify function is useful for initially building the heap or reorganizing it after modifications to the trains vector.