
PROJECT REPORT

Railway Ticket Booking System in C

Aaditya Gandhi [25MCA001] Rudra Makwana [25MCA021]

October 23, 2025

Contents

1	Introduction	3
1.1	Project Overview	3
1.2	Purpose of the Report	3
1.3	Core System Features	3
2	Core Data Structures	4
2.1	The <code>Train</code> Structure (Singly Linked List)	4
2.1.1	Purpose	4
2.1.2	Definition	4
2.1.3	Visual Representation	4
2.1.4	Key Operations	4
2.2	The <code>Passenger</code> Structure (Singly Linked List)	5
2.2.1	Purpose	5
2.2.2	Definition	5
2.2.3	Visual Representation	5
2.2.4	Key Operations	5
2.3	The <code>WaitingQueue</code> Structure (Queue)	6
2.3.1	Purpose	6
2.3.2	Implementation	6
2.3.3	Definition	6
2.3.4	Visual Representation	6
2.3.5	Key Operations	6
3	Key Functional Analysis	7
3.1	The Booking Process (<code>bookTicket()</code>)	7
3.2	The Cancellation Process (<code>cancelTicket()</code>)	7
4	System Architecture	8
5	Conclusion	9
5.1	Summary of Findings	9
5.2	Strengths and Weaknesses	9
5.2.1	Strengths	9
5.2.2	Potential Improvements	9

1 Introduction

1.1 Project Overview

This report provides a comprehensive analysis of a "Railway Ticket Booking System," a console-based application written entirely in the C programming language. The system simulates the core functionalities of a real-world railway reservation platform, managing train schedules, passenger bookings, and waiting lists.

The primary goal of this document is to dissect the underlying architecture of the program, with a specific focus on the data structures used to store and manipulate data dynamically in memory.

1.2 Purpose of the Report

The purpose of this analysis is to:

- Identify and describe the core data structures implemented in the project.
- Explain *why* these specific data structures were chosen for their respective tasks.
- Visually illustrate the state and flow of data within the system.
- Analyze the logic of key operations, such as ticket booking and cancellation, and how they interact with the data structures.
- Provide a high-level overview of the system's architecture.

1.3 Core System Features

The application provides a menu-driven interface with the following capabilities:

- **Train Management:** Add new trains to the system.
- **Information Display:** Display all available trains, all booked passengers, and the current waiting list.
- **Booking:** Book a ticket for a passenger on a specific train. If seats are full, the passenger is added to a waiting list.
- **Cancellation:** Cancel an existing ticket. This action frees a seat and automatically promotes the first passenger from the waiting list, if one exists.
- **Reporting:**
 - Search for all passengers on a specific train.
 - Generate a system-wide summary report of all trains and booking statistics.
 - Print an e-ticket for a specific passenger.

2 Core Data Structures

The system's dynamic nature is managed by three primary data structures, all implemented using C's `struct` and pointers. All data is stored in global variables, representing in-memory databases.

2.1 The Train Structure (Singly Linked List)

2.1.1 Purpose

To store and manage the list of all available trains in the system. A singly linked list is an appropriate choice because the total number of trains is not known at compile time, allowing the system to dynamically add new trains as needed.

2.1.2 Definition

The structure `Train` defines the properties of each train and contains a `next` pointer to chain trains together.

```
1 typedef struct Train
2 {
3     int train_no;
4     char train_name[50];
5     char source[30];
6     char destination[30];
7     int total_seats;
8     int available_seats;
9     struct Train *next;
10 } Train;
11
12 Train *head = NULL;
```

Listing 1: Train struct definition

2.1.3 Visual Representation

The trains are organized as a simple chain, starting from the global head pointer. Each `addTrain()` operation appends a new node to the end of this list.

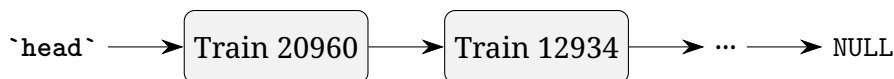


Figure 1: Visual representation of the `Train` singly linked list.

2.1.4 Key Operations

- `addTrain()`: Traverses the list to the end (where `temp->next == NULL`) and appends a new `Train` node.
- `searchTrain(int train_no)`: Traverses the list from the head. It returns a pointer to the `Train` node if the `train_no` matches, or `NULL` if not found. This is an $O(n)$ operation.

2.2 The Passenger Structure (Singly Linked List)

2.2.1 Purpose

To store a list of all passengers who have a **confirmed** booking. Similar to the `Train` list, a linked list is used because the number of booked passengers is dynamic and changes frequently.

2.2.2 Definition

The `Passenger` structure holds passenger details and a link to the next confirmed passenger.

```
1 typedef struct Passenger
2 {
3     int id;
4     char name[50];
5     int age;
6     int train_no;
7     struct Passenger *next;
8 } Passenger;
9
10 Passenger *bookedList = NULL;
```

Listing 2: Passenger struct definition

2.2.3 Visual Representation

This list, pointed to by `bookedList`, holds all confirmed passengers, regardless of which train they are on.

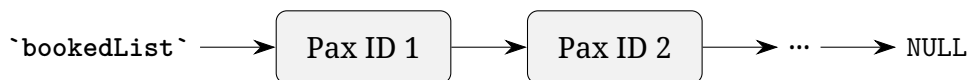


Figure 2: Visual representation of the `Passenger` singly linked list.

2.2.4 Key Operations

- `bookTicket()`: If a seat is available, a new `Passenger` node is created and appended to the end of this list.
- `cancelTicket()`: Searches the list for a specific `id`. This is a standard linked-list removal operation, which requires tracking the `prev` (previous) node to re-link the list correctly.

2.3 The WaitingQueue Structure (Queue)

2.3.1 Purpose

To manage passengers who attempt to book a ticket on a train that is already full. This structure is critical for ensuring fairness. The **First-In, First-Out (FIFO)** nature of a queue guarantees that the first person to be put on the waiting list is the first person to get a confirmed seat when one becomes available.

2.3.2 Implementation

The queue is implemented as a singly linked list, but it is managed using two global pointers: `front` and `rear`.

- `front`: Always points to the first element in the queue (the next to be served).
- `rear`: Always points to the last element in the queue (where new elements are added).

This design allows for $O(1)$ time complexity for both adding (enqueue) and removing (dequeue) elements.

2.3.3 Definition

```
1 typedef struct WaitingQueue
2 {
3     Passenger passenger; // Note: Contains the full passenger data
4     struct WaitingQueue *next;
5 } WaitingQueue;
6
7 WaitingQueue *front = NULL;
8 WaitingQueue *rear = NULL;
```

Listing 3: WaitingQueue struct definition

2.3.4 Visual Representation

New passengers are added at the `rear`, and passengers who get a confirmed seat are removed from the `front`.

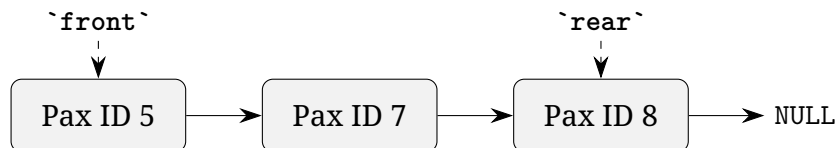


Figure 3: Visual representation of the WaitingQueue (FIFO).

2.3.5 Key Operations

- `enqueue(Passenger p)`: Creates a new `WaitingQueue` node, adds it to the end of the list, and updates the `rear` pointer.
- `dequeue()`: Removes the node from the `front`, returns the `Passenger` data it contained, and updates the `front` pointer.

3 Key Functional Analysis

The true complexity of the system lies in how these three data structures interact. The `bookTicket()` and `cancelTicket()` functions are the primary hubs for this interaction.

3.1 The Booking Process (`bookTicket()`)

This function demonstrates the decision-making logic of the system.

1. **Search:** The user provides a `train_no`. The system traverses the `Train` linked list (using `searchTrain()`) to find the corresponding train node.
2. **Check Seats:** It inspects the `available_seats` property of the found `Train` node.
3. **Case 1: Seats Available (`available_seats > 0`)**
 - The `available_seats` count on the `Train` node is decremented.
 - A new `Passenger` node is created with the user's details.
 - This new node is appended to the `bookedList` linked list.
 - A "Ticket booked successfully" message is shown.
4. **Case 2: No Seats Available (`available_seats == 0`)**
 - A `Passenger` object is created on the stack.
 - This object is passed to the `enqueue()` function.
 - The `enqueue()` function adds the passenger to the `WaitingQueue`.
 - A "Added to waiting list" message is shown.

3.2 The Cancellation Process (`cancelTicket()`)

This function is the most complex as it potentially involves all three data structures.

1. **Search and Remove:** The user provides a passenger `id`. The system traverses the `bookedList` linked list to find and remove the matching `Passenger` node.
2. **Find Train:** The `train_no` from the (now removed) passenger's data is used to search the `Train` linked list.
3. **Update Seats:** The `available_seats` count on that `Train` node is incremented.
4. **Check Waiting List:** The system calls `isEmptyQueue()` to check if the `WaitingQueue` has any passengers.
5. **Promote Passenger (if `!isEmptyQueue()`)**
 - The `dequeue()` function is called, removing the passenger from the front of the `WaitingQueue`.
 - The `available_seats` on the `Train` node is decremented again (as the new passenger just took the seat).
 - A new `Passenger` node is created from the dequeued data.
 - This new node is appended to the `bookedList` linked list.
 - A message is shown that a waiting passenger has been moved to the confirmed list.

This promotion logic correctly ensures that a cancelled ticket is automatically and fairly reassigned.

4 System Architecture

The system is built on a simple, monolithic architecture where the `main()` function's loop acts as the central controller. The data is stored in global variables, which are directly accessed and manipulated by all functions.

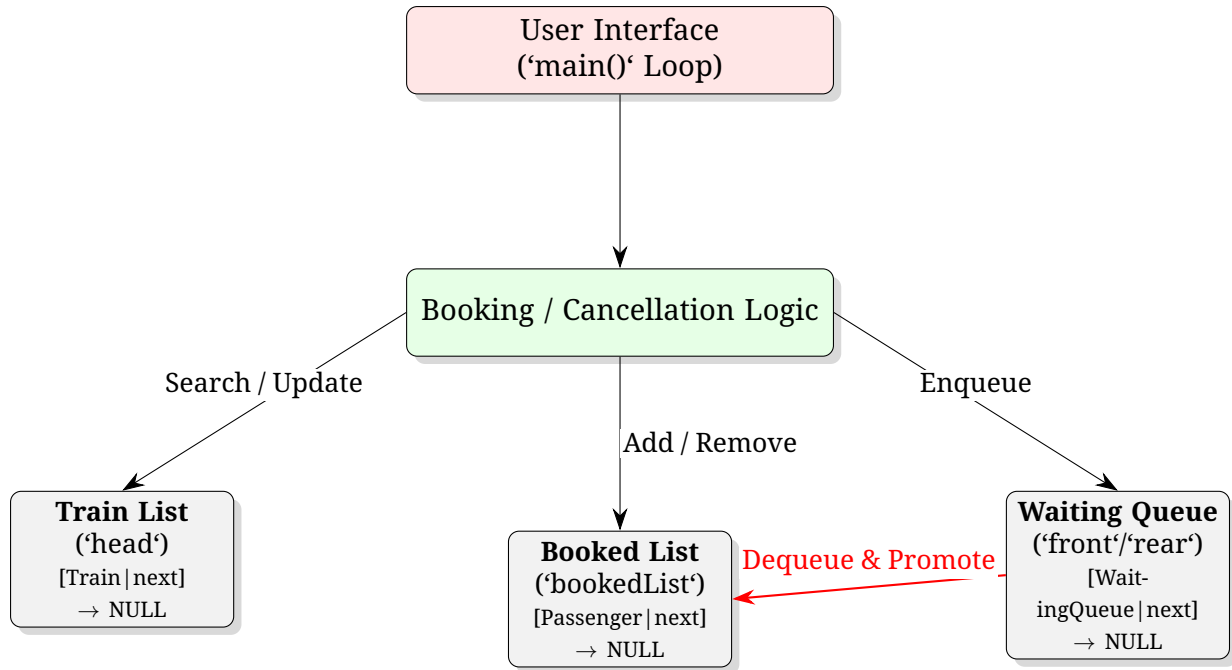


Figure 4: Improved diagram of data flow between system components.

As shown in Figure 4, the `main()` function calls the core logic functions. These functions, in turn, operate on the three global data structures. The `cancelTicket()` logic creates a flow of data from the `WaitingQueue` back to the `bookedList`, completing the system's dynamic cycle.

5 Conclusion

5.1 Summary of Findings

This C-based Railway Booking System effectively utilizes fundamental data structures to manage a dynamic and interactive state.

- **Singly Linked Lists** (for `Train` and `Passenger`) provide a simple and flexible way to store an unknown number of trains and confirmed bookings.
- **A Queue** (for `WaitingQueue`, implemented as a linked list with `front/rear` pointers) is the perfect data structure for managing a waiting list, ensuring FIFO-based fairness.

The project's logic correctly handles the complex interactions between these structures, particularly during the ticket cancellation and passenger promotion process.

5.2 Strengths and Weaknesses

5.2.1 Strengths

- **Appropriate Data Structures:** The choice of data structures (linked lists and a queue) is highly appropriate for the problem.
- **Dynamic:** The system is not limited by fixed-size arrays and can handle any number of trains or passengers, limited only by system memory.
- **Correctness:** The FIFO logic for the waiting list is correctly implemented, which is a critical business rule for a reservation system.

5.2.2 Potential Improvements

- **Search Efficiency:** Searching for a train or a passenger is an $O(n)$ operation. For a real-world system with thousands of passengers, this would be too slow. A **hash table** (using `train_no` or `id` as the key) would provide $O(1)$ average-case lookup time, dramatically improving performance.
- **Data Persistence:** All data is lost when the program exits. The system could be improved by saving the linked lists and queue to a file (e.g., in binary or CSV format) and loading them back at startup.
- **Global Variables:** The heavy reliance on global variables (`head`, `bookedList`, `front`, `rear`) makes the code harder to maintain and test. A better design would be to encapsulate this data within a main "System" struct and pass pointers to it as needed.
- **Input Validation:** The `scanf` calls are not robust and can fail (or cause infinite loops) if the user enters non-numeric data when a number is expected.

Overall, the project serves as an excellent practical demonstration of how linked lists and queues are applied to solve real-world problems.