Railway Reservation System

Submitted by

Shivaya Pandey RA2311056010032

Under the Guidance of

Dr. Rajkumar K.
Assistant Professor, Data Science and Business Systems

In partial satisfaction of the requirements for the degree of

BACHELORS OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING - DATA SCIENCE

DATA SCIENCE AND BUSINESS SYSTEMS

COLLEGE OF ENGINEERING AND TECHNOLOGY SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR - 603203

October 2024

SRM INSTITUTION OF SCIENCE AND TECHNOLOGY KATTANKULATHUR-603203

BONAFIDE CERTIFICATE

Certified that this Course Project Report titled **"Railway Reservation System"** is the Bonafide work done by **Shivaya Pandey(RA2311056010032)** of II Year/ III Sem B.TECH - CSE-DS who carried out under my supervision for the course **21CSC201J DATA STRUCTURES AND ALGORITHMS**. Certified further, that to the best of my knowledge the work reported herein does not form part of any other work.

SIGNATURE

Faculty In-Charge
Dr. Rajkumar K
AssistantProfessor
Department of DSBS
SRM Institute of Science and Technology
Kattankulathur Campus, Chennai

HEAD OF THE DEPARTMENT
Dr. Kavitha V.
Professor and Head ,
Department of DSBS,
SRM Institute of Science and Technology
Kattankulathur Campus, Chennai

DATA STRUCTURES & ALGORITHMS

1.PROBLEM STATEMENT

Problem Statement: Railway Reservation System Using Circular Queue

Objective: The objective of this project is to develop a Railway Reservation System that efficiently manages seat bookings in a cyclic order using a circular queue data structure. In this system, passengers are assigned seats sequentially, and once all seats are occupied, the booking process wraps around to reuse available seats from the beginning. This cyclic nature of the circular queue allows for seamless reservation management without wasting memory or running into overflow/underflow issues. The system enables passengers to book seats, cancel reservations, and view all current bookings, with canceled seats becoming available immediately for new bookings. Through this implementation, we demonstrate the practical application of a circular queue to manage continuous seat availability in a real-world reservation scenario.

2.DATA STRUCTURES USED

Data Structure: Circular Queue

In this Railway Reservation System, we utilize a **Circular Queue** to manage seat allocations in an efficient and cyclical manner. A circular queue operates like a linear queue but with the key difference that the last position wraps around to the first, creating a circular structure. This means that when new reservations reach the maximum seat count, they continue from the start, utilizing available seats that might have been freed by cancellations. The circular queue structure is ideal for this project as it ensures optimal memory utilization and prevents issues like memory overflow and underflow, which are common in linear queue implementations when they reach their capacity.

In this implementation:

- The **front** and **rear** pointers keep track of the start and end of the queue, respectively.
- **Front** moves forward with each cancellation, marking the next available seat, while **rear** moves with each booking, marking the most recently booked seat.

- When **rear** reaches the maximum seat limit and new bookings are made, it wraps back to the beginning of the array, allowing continuous use of the array without requiring memory reallocation.

The primary functionality of this system includes booking a seat, canceling a reservation, and displaying all active reservations. When a booking is made, the passenger is added to the circular queue; if a seat is canceled, it becomes immediately available for the next booking.

3. JUSTIFICATION

A **Circular Queue** is particularly suited for this project because of the following reasons:

1. **Memory Efficiency**: A circular queue is designed to utilize memory in a continuous, looped fashion. Since the queue “wraps around” at the end of its allocated space, it eliminates the need for additional memory. In a standard linear queue, reaching the end would require additional memory for more bookings or a complete reset of the queue. The circular queue’s ability to manage fixed memory space and “recycle” seats makes it highly efficient and avoids the problem of wasted or inaccessible memory.
2. **Cyclic Seat Availability**: In a railway reservation system, seats are often canceled and become available frequently. A circular queue fits this need as it allows the system to continue using previously occupied seats as soon as they are freed. Once the last seat is occupied, bookings continue from the beginning of the queue, allowing a seamless cyclic reservation process. This setup is ideal for scenarios like train reservations, where limited seats must be reused efficiently to accommodate new passengers without downtime or empty seats.

3. **FIFO (First-In-First-Out) Processing**: Circular queues maintain the order of reservations in a First-In-First-Out manner, which is crucial in real-world scenarios. Passengers are given seats in the order they request them, and seats are freed in a way that respects this order. This ensures a fair reservation system where each passenger is processed in the sequence of their booking, providing both order and clarity to the reservation process.
4. **Avoidance of Overflow/Underflow Issues**: A linear queue would encounter overflow when it reaches the end of its allocated space, even if there are available seats at the beginning due to cancellations. A circular queue, however, prevents this by reusing the memory space of the canceled bookings automatically. This guarantees that the system can handle a continuous influx of new bookings without running into overflow issues, as long as the total number of bookings doesn't exceed the total seats.

4. TOOLS TO IMPLEMENT

In developing the Railway Reservation System using a Circular Queue, several tools and technologies were employed to ensure efficient implementation and demonstration of the project.

- **C Programming Language**: The core development of the Railway Reservation System was done using the C programming language. C provides low-level memory manipulation capabilities, making it ideal for implementing data structures like circular queues. Its efficiency and speed are crucial for real-time applications such as reservation systems.
- **Online Compiler**: An online C compiler, such as Repl.it or OnlineGDB, was used for coding, compiling, and executing the C code. These platforms facilitate easy testing and debugging, allowing for rapid development cycles without the need for local setup. The online compiler also provides the convenience of accessing the project from anywhere.
- **Integrated Development Environment (IDE)**: An online C compiler, such as Repl.it or OnlineGDB, was used for coding, compiling, and executing the C code. These platforms facilitate easy testing and debugging, allowing for rapid development cycles without the need for local setup.

These tools collectively provide a robust environment for developing and demonstrating the functionality of the Railway Reservation System, ensuring that the project is not only effective but also easy to manage and maintain.

5.SOURCE CODE

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_SEATS 20
#define MAX_NAME_LENGTH 30
#define MAX_DEST_LENGTH 30

typedef struct {
    int id;
    char name[MAX_NAME_LENGTH];
    char destination[MAX_DEST_LENGTH];
    int seat_number;
    double fare;
} Passenger;

typedef struct {
    Passenger passengers[MAX_SEATS];
    int front;
    int rear;
    int count;
    double total_revenue;
} CircularQueue;

void initializeQueue(CircularQueue* queue) {
    queue->front = 0;
    queue->rear = -1;
    queue->count = 0;
    queue->total_revenue = 0.0;
}

int isFull(CircularQueue* queue) {
    return queue->count == MAX_SEATS;
}

int isEmpty(CircularQueue* queue) {
    return queue->count == 0;
}
```

```c
double calculateFare(const char* destination) {
    return (strlen(destination) > 5) ? 150.0 : 100.0;
}

int searchPassenger(CircularQueue* queue, int id) {
    if (isEmpty(queue)) return -1;

    int index = queue->front;
    for (int i = 0; i < queue->count; i++) {
        if (queue->passengers[index].id == id) {
            return index;
        }
        index = (index + 1) % MAX_SEATS;
    }
    return -1;
}

void bookTicket(CircularQueue* queue, int id, const char* name, const char*
destination) {
    if (isFull(queue)) {
        printf("Booking failed: All seats are full.\n");
        return;
    }

    if (searchPassenger(queue, id) != -1) {
        printf("Booking failed: Passenger ID already exists.\n");
        return;
    }

    queue->rear = (queue->rear + 1) % MAX_SEATS;
    queue->passengers[queue->rear].id = id;
    strncpy(queue->passengers[queue->rear].name, name, MAX_NAME_LENGTH - 1);
    strncpy(queue->passengers[queue->rear].destination, destination,
MAX_DEST_LENGTH - 1);
    queue->passengers[queue->rear].seat_number = queue->count + 1;
    queue->passengers[queue->rear].fare = calculateFare(destination);

    queue->count++;
    queue->total_revenue += queue->passengers[queue->rear].fare;

    printf("Ticket booked for %s (ID: %d)\n", name, id);
    printf("Seat Number: %d\n", queue->passengers[queue->rear].seat_number);
```

```
    printf("Fare: %.2f\n", queue->passengers[queue->rear].fare);
}

void cancelTicket(CircularQueue* queue, int id) {
    if (isEmpty(queue)) {
        printf("Cancellation failed: No bookings available.\n");
        return;
    }

    int index = searchPassenger(queue, id);
    if (index == -1) {
        printf("Cancellation failed: Passenger ID not found.\n");
        return;
    }

    printf("Ticket cancelled for %s (ID: %d)\n",
           queue->passengers[index].name,
           queue->passengers[index].id);

    if (index == queue->front) {
        queue->front = (queue->front + 1) % MAX_SEATS;
    } else {
        int current = index;
        int next = (index + 1) % MAX_SEATS;
        while (current != queue->rear) {
            queue->passengers[current] = queue->passengers[next];
            current = next;
            next = (next + 1) % MAX_SEATS;
        }
        queue->rear = (queue->rear - 1 + MAX_SEATS) % MAX_SEATS;
    }
    queue->count--;
}

void displayReservations(CircularQueue* queue) {
    if (isEmpty(queue)) {
        printf("No reservations to display.\n");
        return;
    }

    printf("\nCurrent Reservations:\n");
    printf("%-4s %-20s %-20s %-8s %-8s\n",
           "ID", "Name", "Destination", "Seat No", "Fare");
```

```
    printf("-------------------------------------------------\n");

    int index = queue->front;
    for (int i = 0; i < queue->count; i++) {
        printf("%-4d %-20s %-20s %-8d %-8.2f\n",
               queue->passengers[index].id,
               queue->passengers[index].name,
               queue->passengers[index].destination,
               queue->passengers[index].seat_number,
               queue->passengers[index].fare);
        index = (index + 1) % MAX_SEATS;
    }
    printf("\nTotal Revenue: %.2f\n", queue->total_revenue);
}

int main() {
    CircularQueue queue;
    initializeQueue(&queue);

    int choice, id;
    char name[MAX_NAME_LENGTH];
    char destination[MAX_DEST_LENGTH];

    while (1) {
        printf("\n=== Railway Reservation System ===\n");
        printf("1. Book Ticket\n");
        printf("2. Cancel Ticket\n");
        printf("3. Display Reservations\n");
        printf("4. Exit\n");
        printf("Enter choice: ");

        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter Passenger ID: ");
                scanf("%d", &id);
                printf("Enter Name: ");
                scanf("%s", name);
                printf("Enter Destination: ");
                scanf("%s", destination);
                bookTicket(&queue, id, name, destination);
                break;
```

```
            case 2:
                printf("Enter Passenger ID to cancel: ");
                scanf("%d", &id);
                cancelTicket(&queue, id);
                break;

            case 3:
                displayReservations(&queue);
                break;

            case 4:
                printf("Thank you for using Railway Reservation System.\n");
                exit(0);

            default:
                printf("Invalid choice! Try again.\n");
        }
    }
    return 0;
}
```

Explanation:

The code implements a basic reservation system using a **circular queue** to manage a set number of passengers. It allows users to book and cancel tickets, search for specific reservations, and display all current reservations.

- **Passenger and Queue Structures**: Two main structures organize the data. The `Passenger` structure holds individual information like ID, name, destination, seat number, and fare. The `CircularQueue` structure represents the booking queue, which can hold up to

`MAX_SEATS` passengers. It includes fields for tracking the queue's front and rear, total passenger count, and total revenue generated.

- **Queue Initialization**: The `initializeQueue` function sets up the queue, preparing it to accept bookings by setting the starting positions of `front` and `rear`, along with initializing the revenue.
- **Booking Tickets**: In the `bookTicket` function, a new passenger is added if seats are available and the ID is unique. It calculates the fare based on the destination length and adds the passenger to the queue, updating both seat assignments and the total revenue.
- **Cancelling Tickets**: The `cancelTicket` function allows a passenger to cancel their ticket by ID. The program searches for the ID, removes the passenger from the queue, and shifts remaining bookings as necessary to maintain order.
- **Displaying Reservations**: The `displayReservations` function lists all current passengers and their details, as well as the total revenue. This is useful for checking the overall status of bookings.
- **Main Menu**: The program runs in a loop, displaying a menu where users can choose to book, cancel, view reservations, or exit. This interactive setup makes it easy to use, with prompts guiding each action.

This system efficiently manages reservations for a small, fixed number of seats, ensuring that bookings are processed in order and allowing passengers to book or cancel as needed. The use of a circular queue helps make the

system both fast and organized.

6. CODE OUTPUT

```
=== Railway Reservation System ===
1. Book Ticket
2. Cancel Ticket
3. Display Reservations
4. Exit
Enter choice:
```

Fig 1: Screenshot of program

```
=== Railway Reservation System ===
1. Book Ticket
2. Cancel Ticket
3. Display Reservations
4. Exit
Enter choice: 1
Enter Passenger ID: 23
Enter Name: Shivaya
Enter Destination: Dehradun
Ticket booked for Shivaya (ID: 23)
Seat Number: 1
Fare: 150.00
```

Fig 2: Screenshot of program

```
Phone Directory Menu:
1. Add Contact
2. Display Directory (Sorted by Name)
3. Display Upcoming Birthdays
4. Delete Contact
5. Search for Contact
6. Edit Contact
7. Exit
Enter your choice: 3
Upcoming Birthdays:
```

Fig 3: Screenshot of program

```
=== Railway Reservation System ===
1. Book Ticket
2. Cancel Ticket
3. Display Reservations
4. Exit
Enter choice: 2
Enter Passenger ID to cancel: 23
Ticket cancelled for Shivaya (ID: 23)
```

Fig 4: Screenshot of program

```
=== Railway Reservation System ===
1. Book Ticket
2. Cancel Ticket
3. Display Reservations
4. Exit
Enter choice: 3

Current Reservations:
ID   Name                Destination          Seat No  Fare
--------------------------------------------------
1    ram                 delhi                1        100.00
2    shreya              mumbai               2        150.00

Total Revenue: 400.00
```

Fig 5: Screenshot of program

TIME COMPLEXITY:

- The system uses a circular queue with a fixed maximum size (MAX_SEATS), which affects all operations.
- Time Complexities of Core Operations:
    - Initializing the queue: O(1) - constant time as it only sets initial values
    - Checking if queue is full/empty: O(1) - simple comparison

operations
    - Calculating fare: O(1) - basic string length check and calculation
- Ticket Booking Operations:
    - Searching for existing passenger ID: O(n) - needs to check through all current passengers
    - Adding a new passenger: O(1) - direct insertion at rear position
    - Overall booking operation: O(n) - dominated by the search complexity
- Ticket Cancellation Operations:
    - Finding the passenger: O(n) - linear search through passengers
    - Removing and shifting passengers: O(n) - may need to shift up to n-1 elements
    - Overall cancellation: O(n) - combination of search and shift operations
- Display Operations:
    - Displaying all reservations: O(n) - needs to traverse all current passengers
    - Displaying total revenue: O(1) - direct access to stored value
- Space Complexity:
    - Fixed space: O(MAX_SEATS) - array size is predetermined

    - Auxiliary space: O(1) - only uses a few additional variables
- The circular queue implementation is efficient for a fixed-size reservation system, but performance may degrade with larger MAX_SEATS values due to linear search operations.
- The system is optimized for quick insertions and deletions at the ends (O(1)) but requires linear time for internal deletions due to necessary shifting of elements.