N.B.K.R INSTITUTE OF SCIENCE & TECHNOLOGY

Vidyanagar-Tirupati Dt

ONLINE TICKET BOOKING

Course: Data structures

Branch: Computer Science

Section: F

Year: I

Semester: II

Submitted to: Ashok Selva Kumar

Submitted by:

Shaik. Ahadiya(24KB1A05GZ)

Kalanji. Nisha(24KB1A05M9)

Kalavalapudi. Yamini(24KB1A05N5)

Karampudi. Thulasi(24KB1A05Q4).

Acknowledgement

I would like to express my sincere gratitude to all those who contributed to the development of the *Online Ticket Booking System using Circular Queue in C*.

This project provided an excellent opportunity to implement and understand core data structure concepts such as circular queues, and how they can be applied in real-world scenarios like ticket management systems.

Special thanks to Mr. Ashok Selva Kumar Sir for his continuous guidance and support, encouragement throughout this learning experience in this project. I appreciate and thankful to my teammates, friends and family for support and motivation. Their feedback and encouragement played a crucial role in refining the logic and improving the user interaction aspects of the program.

I also extend my sincere thanks to my institute and all my faculty members who provided the necessary resources and a learning environment for developing this project.

This project helped enhance my skills in structured programming, user interface design via terminal menus, and memory-efficient queue handling techniques.

Abstract

This project presents the implementation of an **Online Ticket Booking System** using a **Circular Queue** data structure in the C programming language. The system allows users to simulate ticket booking requests with operations to **enqueue**, **dequeue**, and **process tickets** in a **round-robin fashion**. It efficiently manages up to a fixed number of ticket requests using a circular queue to optimize memory and performance. Key functionalities include adding new ticket requests, displaying current queue status, and processing requests in the order they were received. The program uses a simple menu-driven interface for user interaction and demonstrates foundational concepts of **queue management**, **circular array logic**, and **modular programming** in C. This implementation serves as a useful educational tool for understanding how real-time request processing systems can be structured using basic data structures.

Introduction

In today's fast-paced digital environment, efficient management of service requests is essential. This project presents a simulation of an **Online Ticket Booking System** implemented in the C programming language, utilizing a **circular queue** data structure. The circular queue enables effective handling of a fixed number of ticket requests in a round-robin manner, ensuring no space is wasted and requests are processed in the order they arrive.

The program provides a user-friendly, menu-driven interface allowing users to add ticket requests, view the queue, and process requests sequentially. By applying concepts such as queue operations (enqueue, dequeue), buffer wrapping, and condition checks for full or empty states, this system mirrors real-world queue management scenarios like ticket counters, customer service systems, and scheduling tasks.

Objective

The primary objective of this project is to design and implement a simplified **ticket booking system** using the **circular queue** data structure in the C programming language. The system aims to:

- Simulate a real-time ticket request queue with limited capacity.
- Implement core queue operations such as **enqueue**, **dequeue**, and **peek** in a circular buffer format.
- Ensure **efficient memory utilization** and prevent data loss or overflow through circular queue logic.
- Provide a **menu-driven interface** for users to interact with the system in real-time.
- Demonstrate the practical application of data structures in solving real-world problems like service request handling.

System Requirements:

Software Requirements:

Code: Blocks IDE or Turbo C++Compile

Windows/Linux Operating System (or online C compiler)

Hardware Requirements:

Minimum 2 GB RAM

Intel Core i3 Processor or higher

100 MB of disk space for storing files

Literature Review:

Online ticket booking systems, circular queues facilitate the management of ticket requests in a First-In-First-Out (FIFO) manner. This ensures fairness and orderliness in processing requests. The circular nature of the queue allows continuous handling of requests without the need for shifting elements, thereby optimizing performance. circular queues find applications in various domains such as CPU scheduling, memory management, and traffic systems. Their ability to efficiently manage data flow and handle tasks in a cyclical manner makes them a preferred choice in systems requiring predictable and efficient behaviour.

In Online ticket booking, the implementation of circular queues in C provides an effective solution for managing sequential tasks in systems like ticket booking, ensuring efficient memory utilization and consistent performance. circular queues can be implemented using arrays and two pointers: front and rear. These pointers are managed using modulo arithmetic to ensure the circular nature of the queue. This approach allows constant time complexity, 0(1), for both enqueue and dequeue operations, making it suitable for real-time applications.

Methodology:

Step by step process plan to build this project:

1. System Design and Data Structure Selection

- **Objective**: Develop a ticket booking system that efficiently manages multiple booking requests using a circular queue.
- **Data Structure**: Implement a circular queue to handle ticket requests. This structure allows for efficient use of memory by reusing space from completed bookings, as the last position is connected back to the first, forming a circle.

2. Queue Operations Implementation

- **Initialization**: Define a Circular Queue structure with an array to store ticket requests and two integer pointers, front and rear, initialized to -1 to indicate an empty queue.
- **Enqueue Operation**: Add a new ticket request to the queue. If the queue is full, reject the request; otherwise, insert the ticket at the rear position and update the rear pointer using modulo arithmetic to ensure circular behaviour.
- **Dequeue Operation**: Remove and return the ticket request from the front of the queue. If the queue becomes empty after the operation, reset both front and rear to -1.
- **Display Function**: Traverse the queue from front to rear, considering the circular nature, and display all pending ticket requests.

3. User Interface Development

- **Menu-Driven Interface**: Provide a user-friendly interface with options
    1. Add a new ticket request.
    2. Process all ticket requests.
    3. Display current queue status.
    4. Exit the system.
- **Input Handling**: Use standard input functions to capture user choices and ticket numbers, ensuring proper validation to prevent invalid entries.

4. Error Handling and Validation

- **Queue Full Condition**: Before adding a new ticket request, check if the queue is full using the condition (rear + 1) % MAX_QUEUE_SIZE == front. If true, display an error message and reject the request.
- **Queue Empty Condition**: Before processing or displaying tickets, check if the queue is empty by verifying if front == -1. If true, display an appropriate message indicating no pending requests.

5. Testing and Optimization

- **Functionality Testing**: Test all functionalities, including adding, processing, and displaying ticket requests, to ensure they work as expected.
- **Edge Case Handling**: Test scenarios such as attempting to add a ticket when the queue is full, processing tickets when the queue is empty, and displaying the queue when it contains only one ticket.
- **Performance Optimization**: Optimize the enqueue and dequeue operations to ensure constant time complexity, O(1), by utilizing modulo arithmetic for circular indexing.
- This methodology outlines a structured approach to developing a circular queue-based ticket booking system in C, ensuring efficient memory utilization and fair handling of ticket requests.

Project Description:

Problem Statement:

Develop a ticket booking system in the C programming language that efficiently manages booking requests using a circular queue data structure. The system should allow users to add new ticket requests, process existing requests in a First-In-First-Out (FIFO) manner and display the current queue status. By implementing a circular queue, the system aims to optimize memory utilization by reusing vacant spaces created after processing requests, thus overcoming the limitations of a linear queue.

Proposed solution:

The proposed solution involves developing a ticket booking system using the **circular queue** data structure to efficiently manage booking requests. This ensures **FIFO (First-In-First-Out) processing**, avoids memory wastage, and optimally utilizes vacant spots once requests are processed.

Key Features:

✅ **Efficient Memory Utilization**

✅ **FIFO Processing**

✅ **Dynamic Queue Management**

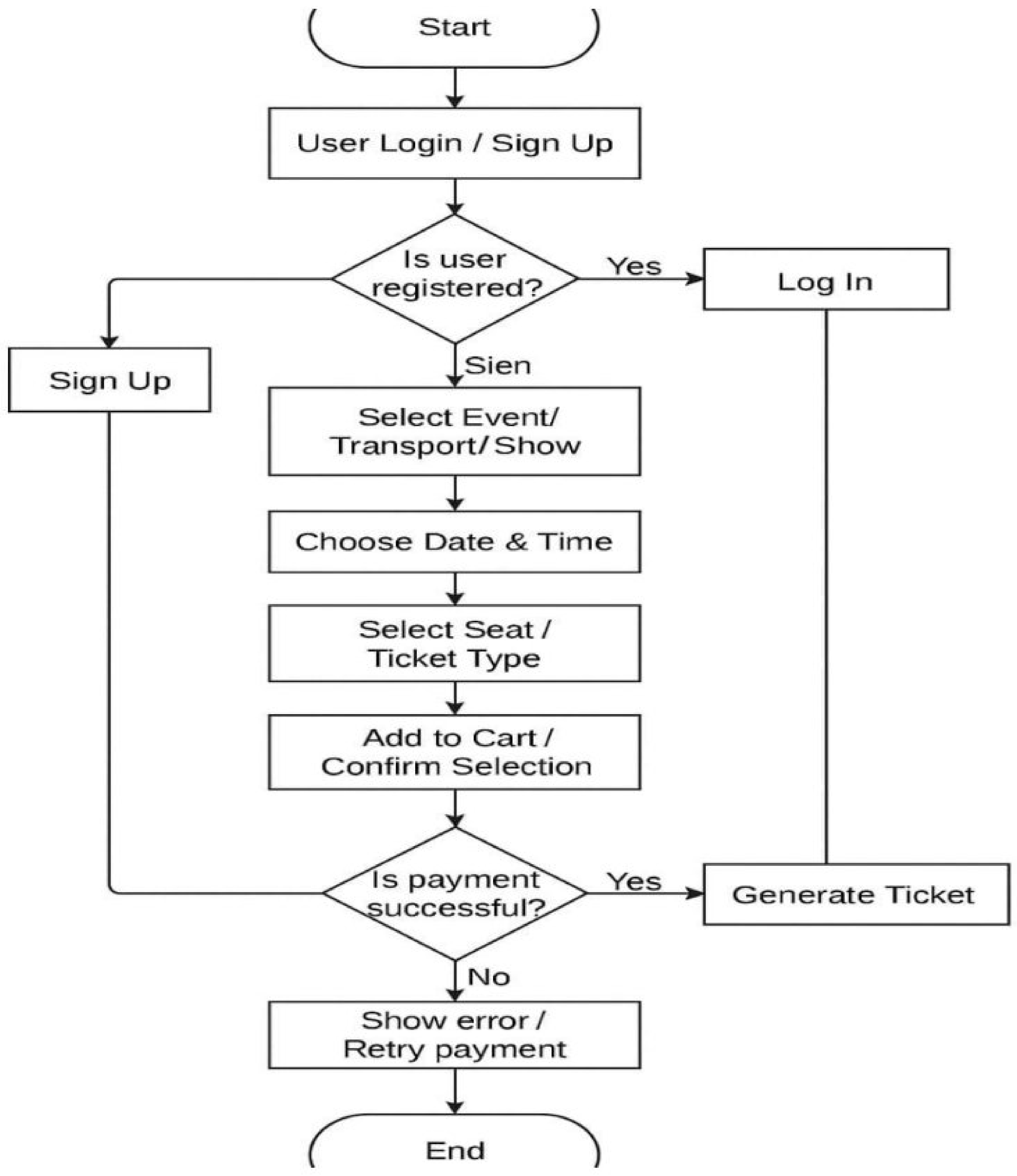
✅ **Fast Ticket Addition & Processing**

✅ **Queue Status Display**

✅ **Handling Full & Empty Conditions**

✅ **Scalability**

Flow chart:



Algorithm :

1. Start

2. Initialize front = -1, rear = -1, and define a fixed-size queue.

3. Repeat until user chooses to exit:

Display menu:

1. Add Ticket Request

2. Process Ticket Request

3. Show Queue

```
    4. Exit

 *  If choice is 1 (Add Request):
      a)Check if queue is full:
          If yes, display "Queue is full"
          Else:
            If queue is empty, set front = rear = 0
            Else, update rear = (rear + 1) % SIZE
           Add the request at queue[rear]
*If choice is 2 (Process Request):
        b)Check if queue is empty:
          If yes, display "No requests"
          Else:
            Display and remove queue[front]
            If only one element, reset front and rear to -1
           Else, update front = (front + 1) % SIZE
*If choice is 3 (Show Queue):
         If empty, show "Queue is empty"

          Else, display queue from front to rear
*If choice is 4 (Exit):
         Terminate the program
Else:
       Show "Invalid choice".
4.Exit.
 Repeat step 3 until the user exits.
End.
```

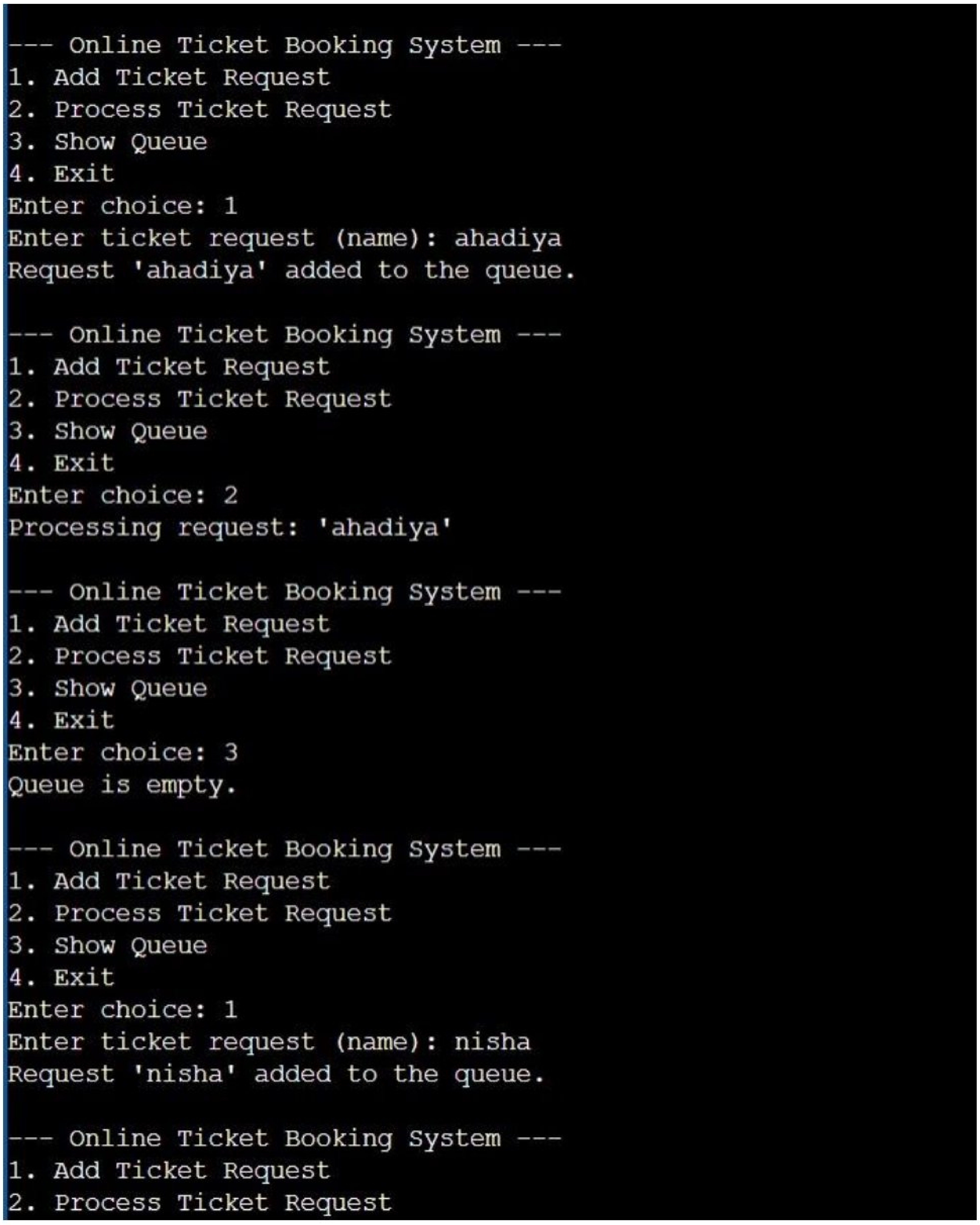
```
}
```

Program Code:

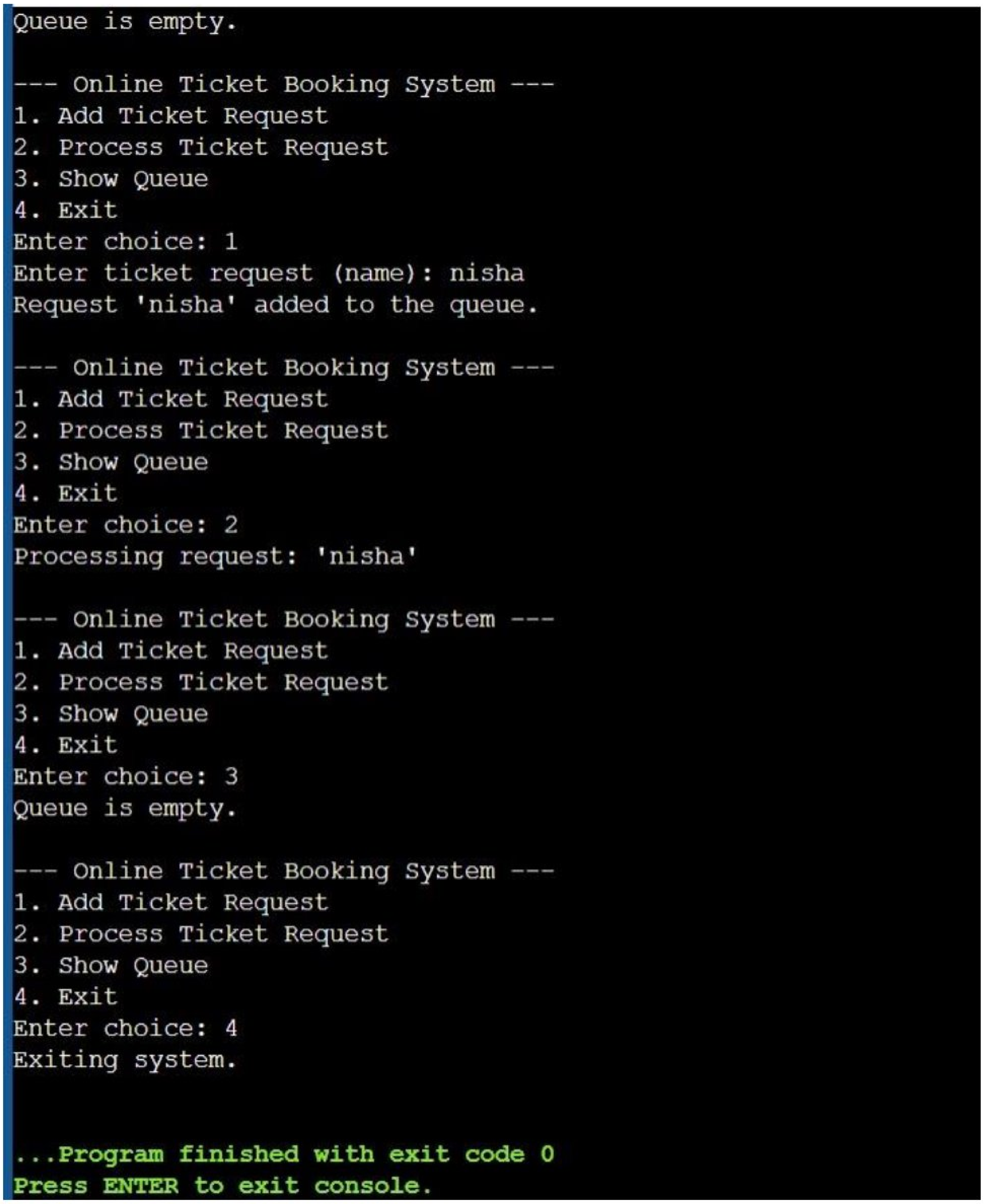
```
#include <stdio.h>
#include <string.h>

#define SIZE 5
#define MAX_NAME_LEN 30

// Circular Queue for storing ticket requests
char queue[SIZE][MAX_NAME_LEN];
int front = -1, rear = -1;

// Function to check if the queue is full
int isFull() {
   return (front == (rear + 1) % SIZE);
}

// Function to check if the queue is empty
int isEmpty() {
   return (front == -1);
}

// Enqueue a ticket request
void enqueue(char request[]) {
   if (isFull()) {
      printf("Request rejected! Queue is full.\n");
      return;
   }
```

```
    if (isEmpty()) {
        front = rear = 0;
    } else {
        rear = (rear + 1) % SIZE;
    }

    strcpy(queue[rear], request);
    printf("Request '%s' added to the queue.\n", request);
}

// Dequeue and process a ticket request
void dequeue() {
    if (isEmpty()) {
        printf("No ticket requests to process.\n");
        return;
    }

    printf("Processing request: '%s'\n", queue[front]);

    if (front == rear) {
        front = rear = -1;  // Queue becomes empty
    } else {
        front = (front + 1) % SIZE;
    }
}

// Display the current state of the queue
void display() {
    if (isEmpty()) {
        printf("Queue is empty.\n");
        return;
    }

    printf("Current Queue:\n");
    int i = front;
    while (1) {
        printf(" - %s\n", queue[i]);
        if (i == rear) break;

 i = (i + 1) % SIZE;
    }
}

int main() {
    int choice;
    char request[MAX_NAME_LEN];

    while (1) {
        printf("\n--- Online Ticket Booking System ---\n");
        printf("1. Add Ticket Request\n");
        printf("2. Process Ticket Request\n");
        printf("3. Show Queue\n");
        printf("4. Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);
        getchar(); // Consume newline

        switch (choice) {
            case 1:
               printf("Enter ticket request (name): ");
               fgets(request, MAX_NAME_LEN, stdin);

    request[strcspn(request, "\n")] = '\0'; // Remove newline
               enqueue(request);
               break;
            case 2:
               dequeue();
               break;
            case 3:
               display();
               break;
            case 4:
               printf("Exiting system.\n");
               return 0;
            default:
               printf("Invalid choice! Try again.\n");
         }
      }

      return 0;
    }
```

Output:



```
--- Online Ticket Booking System ---
1. Add Ticket Request
2. Process Ticket Request
3. Show Queue
4. Exit
Enter choice: 1
Enter ticket request (name): ahadiya
Request 'ahadiya' added to the queue.

--- Online Ticket Booking System ---
1. Add Ticket Request
2. Process Ticket Request
3. Show Queue
4. Exit
Enter choice: 2
Processing request: 'ahadiya'

--- Online Ticket Booking System ---
1. Add Ticket Request
2. Process Ticket Request
3. Show Queue
4. Exit
Enter choice: 3
Queue is empty.

--- Online Ticket Booking System ---
1. Add Ticket Request
2. Process Ticket Request
3. Show Queue
4. Exit
Enter choice: 1
Enter ticket request (name): nisha
Request 'nisha' added to the queue.

--- Online Ticket Booking System ---
1. Add Ticket Request
2. Process Ticket Request
```



```
Queue is empty.

--- Online Ticket Booking System ---
1. Add Ticket Request
2. Process Ticket Request
3. Show Queue
4. Exit
Enter choice: 1
Enter ticket request (name): nisha
Request 'nisha' added to the queue.

--- Online Ticket Booking System ---
1. Add Ticket Request
2. Process Ticket Request
3. Show Queue
4. Exit
Enter choice: 2
Processing request: 'nisha'

--- Online Ticket Booking System ---
1. Add Ticket Request
2. Process Ticket Request
3. Show Queue
4. Exit
Enter choice: 3
Queue is empty.

--- Online Ticket Booking System ---
1. Add Ticket Request
2. Process Ticket Request
3. Show Queue
4. Exit
Enter choice: 4
Exiting system.


...Program finished with exit code 0
Press ENTER to exit console.
```

Testing and validation:

🧪 **Test Case 1: Enqueue Until Full**

Description: Add ticket requests to the queue until it reaches its maximum capacity

Steps:

1. **Add "Alice"**
2. **Add "Bob"**
3. **Add "Charlie"**
4. **Add "David"**
5. **Add "Eve"**
6. **Attempt to add "Frank" (should be rejected as the queue is full)**

Expected Output:

```
Request 'Alice' added to the queue.

Request 'Bob' added to the queue.

Request 'Charlie' added to the queue.

Request 'David' added to the queue.

Request 'Eve' added to the queue.

Request rejected! Queue is full.
```

🧪 **Test Case 2: Dequeue and Add New Requests**

Description: Remove a ticket request and then add a new one to the queue.

Steps:

1. **Add "Alice"**
2. **Add "Bob"**
3. **Add "Charlie"**
4. **Dequeue (remove "Alice")**
5. **Add "David"**
6. **Add "Eve"**

Expected Output:

```
Request 'Alice' added to the queue.

Request 'Bob' added to the queue.

Request 'Charlie' added to the queue.

Processing request: 'Alice'

Request 'David' added to the queue.

Request 'Eve' added to the queue.
```

🧪 Test Case 3: Display Queue After Operations

Description: Display the queue's state after performing enqueue and dequeue operations.

Steps:

1. Add "Alice"
2. Add "Bob"
3. Dequeue (remove "Alice")
4. Display the queue

Expected Output:

```
Request 'Alice' added to the queue.

Request 'Bob' added to the queue.

Processing request: 'Alice'

Current Queue:

 - Bob
```

🧪 Test Case 4: Attempt to Dequeue from an Empty Queue

Description: Attempt to remove a ticket request from an empty queue.

Steps:

1. Dequeue (should display an error message)

Expected Output:

```
No ticket requests to process.
```

🧪 Test Case 5: Add Requests After Dequeue

Description: Add new ticket requests after the queue has been emptied

Steps:

1. **Add "Alice"**
2. **Add "Bob"**
3. **Dequeue (remove "Alice")**
4. **Dequeue (remove "Bob")**
5. **Add "Charlie"**
6. **Add "David"**

   Expected Output:

Request 'Alice' added to the queue.

Request 'Bob' added to the queue.

Processing request: 'Alice'

Processing request: 'Bob'

Request 'Charlie' added to the queue.

Request 'David' added to the queue.

🧪 **Test Case 6: Display Empty Queue**

Description: Display the queue's state when it is empty.

Steps:

1. **Display the queue**

Expected Output:

Queue is empty.

🧪 **Test Case 7: Add Requests After Queue is Full**

Description: Attempt to add ticket requests after the queue has reached its maximum capacity.

Steps:

1. **Add "Alice"**
2. **Add "Bob"**
3. **Add "Charlie"**

4. Add "David"
5. Add "Eve"
6. Attempt to add "Frank" (should be rejected as the queue is full)
7. Dequeue (remove "Alice")
8. Attempt to add "Frank" (should succeed now)

Expected Output:

```
Request 'Alice' added to the queue.
Request 'Bob' added to the queue.
Request 'Charlie' added to the queue.
Request 'David' added to the queue.
Request 'Eve' added to the queue.
Request rejected! Queue is full.
Processing request: 'Alice'
Request 'Frank' added to the queue.
```

🧪 **Test Case 8: Add and Remove Multiple Requests**

Description: Add multiple ticket requests and then remove them, checking the queue's state at each step.

Steps:

1. Add "Alice"
2. Add "Bob"
3. Add "Charlie"
4. Dequeue (remove "Alice")
5. Add "David"
6. Dequeue (remove "Bob")
7. Dequeue (remove "Charlie")
8. Dequeue (remove "David")

Expected Output:

```
Request 'Alice' added to the queue.
Request 'Bob' added to the queue.
Request 'Charlie' added to the queue.
Processing request: 'Alice'
Request 'David' added to the queue.
Processing request: 'Bob'
Processing request: 'Charlie'
Processing request: 'David'
```

These test cases comprehensively cover various scenarios your circular queue implementation might encounter. By running these tests, you can ensure that your program handles different situations correctly and efficiently.

Limitations:

1.No Graphical User Interface (GUI):The project runs in a command-line environment, which may not be user-friendly for non-technical users.

2.Lack of Real-Time Updates: The system does not support real-time seat availability updates, which can lead to conflicts in multi-user scenarios.

3.No Internet or Network Support:Being a standalone console application, it cannot connect to servers or handle online payments.

4.Limited Error Handling: The system has minimal input validation and may crash or behave unexpectedly with invalid input.

5.No Database Integration: Data is stored using simple text files, which is not secure or scalable for large systems.

6.Single User Environment: It is designed for single-user operation, making it unsuitable for concurrent multi-user use.

7.No Authentication System:The system does not include login or user roles (like admin or customer), which limits control and security.

Future Enhancements:

1.Graphical User Interface (GUI):Add a user-friendly interface using libraries like GTK or migrate to a language that supports GUI development.

2.Database Integration: Use databases like MySQL or SQLite for secure and scalable data management.

3.User Authentication System: Implement login functionality with roles for admin and users to enhance security.

4.Real-Time Booking and Updates: Enable real-time seat availability and booking updates, especially for multi-user access.

5.Online Payment Integration: Integrate secure payment gateways for actual ticket purchases.

6.Multi-User Support: Design the system for concurrent access over a network or the internet.

7.Email or SMS Notifications: Send booking confirmations and updates via email or SMS.

8.Reporting and Analytics: Generate reports on bookings, revenue, and user activity for admin review

Conclusion:

The Online Ticket Booking System developed using the C programming language demonstrates the practical implementation of fundamental programming concepts such as structures, file handling, arrays, and user-defined functions. This project provides a basic yet functional platform for users to book, view, and cancel tickets efficiently. While it may not include advanced features like database integration or a graphical interface, it effectively simulates the core functionality of real-world ticket booking systems. This project also highlights the potential of C language in building console-based management systems and serves as a strong foundation for more complex and user-friendly applications in the future.

References:

1.Algorithms and Data Structures: The Basic Toolbox by Kurt Mehlhorn and Peter Sanders.

2. C Data Structures and Algorithms by Alfred V. Aho, Jeffrey D. Ullman, and John E. Hopcroft.

3. Problem Solving with Algorithms and Data Structures" by Brad Miller and David Ranum .

4. Introduction to Algorithms by Thomas HCormen, Charles ELeiserson

, Ronald L. Rivest, and Clifford Stein.

5. Algorithms in C, Parts 1-5 (Bundle): Fundamentals, Data Structures, Sorting, Searching, and Graph Algorithms" by Robert Sedgewick.