GatorTicketMaster Project Report

Name: Pranaya Yadav Palleboyina
UFID: 11602431
UF Email: p.palleboyina@ufl.edu

Project Overview
GatorTicketMaster is a seat booking service implementation that manages seat reservations and waitlist operations for Gator Events. The system uses custom implementations of Red-Black Trees and Binary Min-Heaps to provide efficient seat allocation and priority-based waitlist management.

Data Structures Used

1. Red-Black Tree
Used to manage seat reservations with (O(log n)) time complexity.

```
class RedBlackTree:
    def insert(self, user_id, seat_id)         # O(log n)
    def delete_node(self, node)                # O(log n)
    def find_by_user_id(self, user_id)         # O(log n)
    def find_by_seat_id(self, seat_id)         # O(log n)
```

2. Binary Min-Heap (Waitlist)

Implements a priority queue for waitlist management with (O(log n)) operations.

```
class MinHeap:
    def insert(self, priority, timestamp, user_id)       # O(log n)
    def extract_min()                                    # O(log n)
    def remove_user(self, user_id)                       # O(log n)
    def update_priority(self, user_id, new_priority) # O(log n)
```

3. Binary Min-Heap (Available Seats)

Manages available seat numbers, ensuring the lowest-numbered seats are assigned first.

```
class AvailableSeatsHeap:
    def insert(self, seat_id)       # O(log n)
    def extract_min()               # O(log n)
```

Program Structure

Core Classes and Their Relationships

1. SeatReservationSystem:
- Main coordinator class
- Manages all reservation operations
- Integrates the three data structures
- Handles I/O operations

2. Supporting Classes:
- Color: Enum for Red-Black Tree node colors
- RBNode: Node class for Red-Black Tree
- MinHeapNode: Node class for Binary Heaps

Function Prototypes and Program Structure

RedBlackTree Class

```
class RedBlackTree:
    def __init__(self): ...
    def left_rotate(self, x: RBNode): ...
    def right_rotate(self, x: RBNode): ...
    def insert_fixup(self, z: RBNode): ...
    def insert(self, user_id: int, seat_id: int): ...
    def find_by_user_id(self, user_id: int) -> RBNode: ...
    def find_by_seat_id(self, seat_id: int) -> RBNode: ...
    def delete_node(self, z: RBNode): ...
```

MinHeap Class (Waitlist)

```
class MinHeap:
    def __init__(self): ...
    def insert(self, priority: int, timestamp: int, user_id: int): ...
    def extract_min(self) -> MinHeapNode: ...
    def remove_user(self, user_id: int) -> bool: ...
    def heapify_up(self, i: int): ...
    def heapify_down(self, i: int): ...
```

AvailableSeatsHeap Class

```
class AvailableSeatsHeap:
    def __init__(self): ...
    def insert(self, seat_id: int): ...
    def extract_min(self) -> int: ...
    def _heapify_up(self, i: int): ...
    def _heapify_down(self, i: int): ...
```

SeatReservationSystem Class

```
class SeatReservationSystem:
    def __init__(self, output_file: str): ...
    def initialize(self, seat_count: int): ...
    def reserve(self, user_id: int, user_priority: int): ...
    def cancel(self, seat_id: int, user_id: int): ...
    def update_priority(self, user_id: int, new_priority: int): ...
    def add_seats(self, count: int): ...
    def exit_waitlist(self, user_id: int): ...
    def release_seats(self, user_id1: int, user_id2: int): ...
```

Helper Classes

RBNode Class

```
class RBNode:
    def __init__(self, user_id: int, seat_id: int): ...
```

MinHeapNode Class

```
class MinHeapNode:
    def __init__(self, priority: int, timestamp: int, user_id: int): ...
```

Program Flow

Initialization Flow

```
Initialize(seat_count)
├── Create RedBlackTree for reservations
├── Create MinHeap for waitlist
└── Create AvailableSeatsHeap with seats 1 to seat_count
```

Reservation Flow
Reserve(user_id, priority)
├── Check available seats
│   ├── If seat available: Assign seat and update RBTree
│   └── If no seats: Add to waitlist MinHeap
└── Update system state

Cancellation Flow
Cancel(seat_id, user_id)
├── Verify reservation in RBTree
├── Remove reservation
├── Check waitlist
│   ├── If waitlist empty: Add to available seats
│   └── If users waiting: Assign to highest priority
└── Update system state

Implementation Details

1. Seat Reservation Process
- Check available seats using `AvailableSeatsHeap`
- If a seat is available: Assign using `RedBlackTree`
- If no seats: Add to waitlist with priority
- Time Complexity: \(O(\log n) \)

2. Priority Management
- Waitlist uses negative priorities for max-heap behavior
- Timestamps ensure FIFO ordering within the same priority
- Priority updates preserve original timestamps

3. Seat Release Process
- Identify affected seats using `RedBlackTree`
- Remove reservations
- Remove users from the waitlist
- Reassign seats to waiting users by priority

4. Error Handling
- Invalid seat/user IDs
- Non-existent reservations
- Invalid priority updates
- File I/O errors

Time Complexity Analysis

Operation	**Time Complexity**	**Description**
Initialize	O(n)	Creating initial available seats heap
Reserve	O(log n)	Seat assignment or waitlist addition
Cancel	O(log n)	Reservation removal and reassignment
UpdatePriority	O(log n)	Priority modification in the waitlist
AddSeats	O(m log n)	m = new seats added
ReleaseSeats	O(k log n)	k = seats in range

Testing and Validation

The system was tested using various test cases covering:
1. Basic seat reservations
2. Priority-based waitlist operations
3. Seat cancellations and reassignments
4. Bulk seat releases
5. Edge cases and error conditions

Conclusion

The implementation successfully meets all project requirements while maintaining efficient time complexities. Red-Black Trees and Binary Heaps ensure optimal performance for all operations, making the system scalable for large numbers of seats and users.