GATOR TICKET MASTER – PROJECT REPORT

Name: Bala Surya Krishna Vankayala

UFID: 82258857

UF-Email: b.vankayala@ufl.edu

1. PROJECT DESCRIPTION

Gator Ticket Master is a powerful yet user-friendly seat booking and reservation system designed to handle the demands of big, high-energy events where every seat counts. Built in Java, Gator Ticket Master makes it easy to manage reservations, waitlists, and changes to seating, all in real time. The system relies on advanced data structures — like Red-Black Trees, Min-Heaps, and a custom Priority Queue — to make sure seats are fairly and efficiently distributed, even when the pressure is on.

The main goal of Gator Ticket Master is to make things as smooth as possible for both event organizers and attendees. Whether it's a concert, a sports event, or an academic conference, Gator Ticket Master takes care of the logistics so everyone can focus on what matters: enjoying the event.

2. PROJECT STATEMENT AND REQUIREMENTS

The system should support the following key functionalities and operations:

1. Initialization of Event with Seats:

- **Operation**: Initialize(seatCount)
- **Description**: Initialize the event with the specified number of seats, seatCount. The seats will be sequentially numbered starting from 1 and added to the list of available seats.
- **Data Structure**: A Binary Min-Heap to store the available seats.

2. View Available Seats and Waitlist Length:

- **Operation**: Available()

- **Description**: Display the number of seats that are currently available for reservation, as well as the number of users on the waitlist.

3. Seat Reservation:

- **Operation**: Reserve(userID, userPriority)
- **Description**: Allow a user to reserve an available seat. If no seats are available, the user is added to the waitlist with a priority and timestamp. If a seat is assigned, the system should print the assigned seat number. If the user is added to the waitlist, print a message stating that the user is added to the waitlist.
- **Data Structures**:
    - A Red-Black Tree for managing reserved seats, where each node contains UserID (key) and SeatID (value).
    - A Binary Min-Heap to manage the waitlist, ordered by userPriority and timestamp.

4. Cancel Reservation:

- **Operation**: Cancel(seatID, userID)
- **Description**: Cancel the reservation of a seat, and reassign the seat to a user from the waitlist. If the waitlist is empty, the seat is added back to the available seats.
- **Data Structures**:
    - A Binary Min-Heap for unassigned available seats.
    - A Binary Min-Heap for the waitlist of users awaiting seat assignments.

5. Exit Waitlist:

- **Operation**: ExitWaitlist(userID)
- **Description**: If the user is on the waitlist, remove them. If the user has a reserved seat, they must cancel their reservation first before exiting the waitlist.

6. Update User Priority on Waitlist:

- **Operation**: UpdatePriority(userID, userPriority)
- **Description**: Update a user's priority if they are on the waitlist. Reorganize the waitlist heap to reflect the new priority while maintaining the original timestamp data.

7. Add New Seats:

- **Operation**: AddSeats(count)
- **Description**: Add count new seats to the available seats list. The new seat numbers should continue sequentially from the highest available seat number.

8. Print Current Reservations:

- **Operation**: PrintReservations()
- **Description**: Print a list of all assigned reservations, ordered by seat number. Each entry should show the UserID and their assigned SeatID.

9. Release Seats for a Range of User IDs:

- **Operation**: ReleaseSeats(userID1, userID2)
- **Description**: Release all seats assigned to users whose IDs fall within the range [userID1, userID2]. The seats should be re-added to the available seats heap. If any users in this range are on the waitlist, they should be removed.

10. Quit System:

- **Operation**: Quit()
- **Description**: Terminate the system. Any commands following this will not be processed, and the program will exit.

ADDITIONAL REQUIREMENTS

1. Data Structures:

- **Red-Black Tree**: Used to store reservations, where each node holds `UserID` and `SeatID`.
    - Ensure efficient insertion, deletion, and searching for reservations.
- **Binary Min-Heap**: Used for both the waitlist and the available seats.
    - The waitlist is ordered by `userPriority` (highest priority first) and then by timestamp in case of ties.
    - The available seats heap ensures that the seat with the lowest number is always assigned first.

2. Time Complexity:

- The operations involving seat reservation, cancellation, priority update, and seat release should be efficient in terms of time complexity, making use of the logarithmic properties of the Red-Black Tree and Binary Min-Heap.

3. Correctness:

- The system should maintain the correct state of reservations, available seats, and the waitlist at all times. The priority queue (min-heap) should accurately reflect user priorities and timestamps.

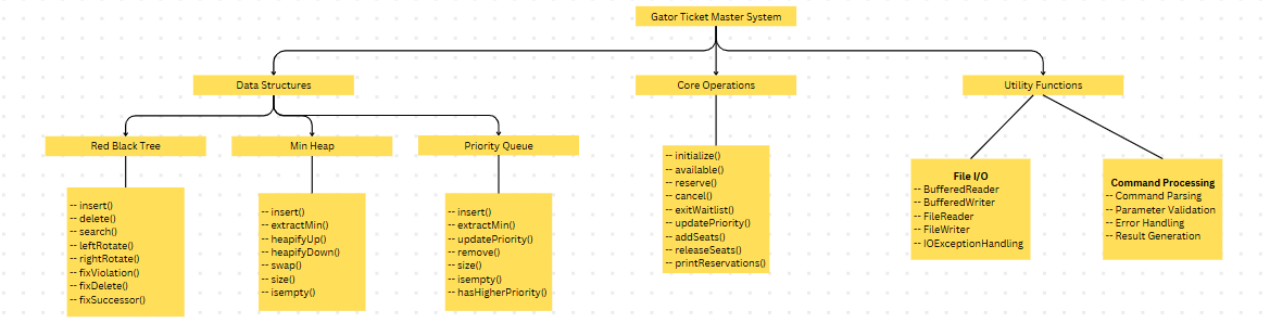
4. Edge Cases:

- Handle cases where no seats are available, when the waitlist is empty, when a user cancels a reservation, and when new seats are added dynamically.

5. User Experience:

- The system should provide clear feedback to the user regarding the reservation status, the position on the waitlist, and any changes to their reservation.

3. PROGRAM STRUCTURE



3.1 DATA STRUCTURES

3.1.1 Red Black Tree (Reservation Management)

The Red-Black Tree is used to store and manage seat reservations. This balanced binary search tree allows efficient insertion, deletion, and searching, crucial for handling high volumes of reservation data without sacrificing performance. Red-Black Trees maintain balance with minimal overhead, ensuring that operations remain efficient even as the number of reservations grows. This choice ensures that users can retrieve, add, or cancel

reservations quickly, which is essential for high-demand, real-time systems. Each operation—*insert*, *search*, and *delete* — is O(logn), which is optimal for balanced search trees.

Implemented Methods :

- **insert**: Adds a new reservation while maintaining balance in the tree, ensuring fast and orderly data entry.
- **search**: Retrieves reservation details by user ID, providing a quick lookup for user-specific data.
- **delete**: Removes a reservation and re-balances the tree, essential for real-time cancellations.
- **inorder_traversal**: Generates an ordered list of reservations, useful for creating reports or displaying reservation data.

3.1.2 Min Heap (Waitlist Management)

The Min Heap stores waitlisted users, ordered by priority level. This ensures that the highest-priority user (the user with the smallest priority value) is served first when a seat becomes available. Min Heaps efficiently manage dynamically changing priorities, making them ideal for prioritizing users in the waitlist based on urgency or other criteria. This approach allows quick reassignment of available seats to those with the highest need. Insertions, deletions, and extractMin operations all have O(logn) complexity, ensuring efficient management of waitlist updates.

Implemented Methods:

- **insert**: Adds a user to the waitlist with a specified priority, maintaining heap properties to ensure the waitlist remains ordered.
- **extractMin**: Retrieves and removes the highest-priority user, simplifying the allocation of available seats.
- **remove**: Removes a specific user from the waitlist, typically after they are assigned a seat.
- **update_priority**: Adjusts a user's priority and repositions them in the heap to maintain order.
- **isEmpty**: Checks if the waitlist has any remaining users, aiding seat management decisions.

3.1.3 Min Heap (Available Seat Management)

The SeatMinHeap manages available seats in ascending order by seat number, ensuring that the lowest-numbered seat is assigned first when a new reservation is made. This approach optimizes the seating arrangement by minimizing gaps and maximizing the use of contiguous seats. It also ensures that users are assigned seats systematically and fairly. Insertion, extraction, and lookup operations are O(logn), ensuring fast updates as seats are vacated or assigned.

Implemented Methods:

- **insert**: Adds a seat to the heap when it becomes available, maintaining order for efficient seat allocation.
- **extractMin**: Retrieves and removes the lowest-numbered seat, ensuring an orderly assignment to the next user.
- **contains**: Checks if a specific seat is available, aiding in confirming seat status.
- **isEmpty**: Determines if any seats are currently available, guiding waitlist actions when all seats are occupied.

3.1.4 Priority Queue (Task Prioritization)

The Priority Queue manages high-priority tasks within the reservation system, providing a flexible way to process urgent tasks based on priority levels. This structure is effective for managing dynamically prioritized tasks or users, especially in cases where specific actions or events need to be handled immediately. Both insertion and removal operations are O(logn), making it efficient for frequent priority-based operations.

Implemented Methods:

- **offer**: Adds a task or user action to the queue based on its priority, ensuring that high-priority items are accessible.
- **poll**: Retrieves and removes the highest-priority task, facilitating timely handling of critical actions.
- **peek**: Accesses the highest-priority task without removing it, providing a view of the next task to be processed.

3.1.5 Buffered Writer (Output Management)

BufferedWriter handles logging for the reservation system, recording events such as new reservations, cancellations, and waitlist updates. This ensures data persistence and supports accurate system tracking. BufferedWriter provides efficient, buffered output to files, reducing I/O overhead and ensuring that log entries are written to the output file in a

controlled manner. *write*, *flush*, and *close* operations are O(1) on average, as they operate in constant time with buffered output.

Implemented Methods:

- **write**: Logs each system event (reservation or cancellation) to the output file, creating a persistent activity record.
- **flush**: Ensures all buffered data is written to the file immediately, improving the accuracy of real-time logs.
- **close**: Closes the output stream, securing data integrity and releasing file resources when logging is complete.

3.2 CORE OPERATIONS

The Core Operations of the GatorTicketMaster system manage seat reservations, cancellations, waitlist assignments, priority updates, and reporting. Each operation is designed to leverage the system's data structures — Red-Black Tree, Min Heap, SeatMinHeap, Priority Queue, and BufferedWriter — to ensure that the booking process is efficient, fair, and responsive to user needs. These operations allow the system to handle high volumes of requests with optimal time complexity, making it suitable for real-time seat management in high-demand scenarios.

3.2.1 Initialize System

This operation sets up all necessary components, preparing the reservation system for smooth functionality. It initializes the main data structures—Red-Black Tree for reservations, Min Heap for the waitlist, and SeatMinHeap for available seats—along with file handling for logging purposes.

Functions Used:

- **Constructor(gatorTicketMaster(String outputFile)):** Initializes reservations, waitlist, availableSeats, and output, setting up the system for operations.

3.2.2 Reserve Seat

This operation assigns a new reservation by selecting the next available seat and recording the reservation in the Red-Black Tree. By leveraging the SeatMinHeap, the system ensures orderly seat allocation, prioritizing lower seat numbers.

Functions Used:

- **extractMin (SeatMinHeap)**: Retrieves the lowest-numbered available seat.

- **insert (RedBlackTree)**: Stores the reservation details (user ID and seat ID).
- **write (BufferedWriter)**: Logs the reservation details to the output file.

3.2.3 Cancel Reservation

In cases where a user cancels, this operation efficiently removes the reservation from the system and returns the seat to the pool of available seats. The operation keeps the system responsive, allowing other users to access newly available seats.

Functions Used:

- **search (RedBlackTree)**: Finds the reservation by user ID.
- **delete (RedBlackTree)**: Removes the reservation while balancing the tree.
- **insert (SeatMinHeap)**: Returns the canceled seat to the available seats.
- **write (BufferedWriter)**: Logs the cancellation event.

3.2.4 Manage Waitlist

This set of operations allows the system to handle users waiting for seats, ensuring seats are allocated based on user priority.

- **Add to Waitlist**: Adds a user to the waitlist with a specific priority, maintaining an ordered waitlist.
    - **Functions Used**:
        - **insert (MinHeap)**: Adds the user to the waitlist based on priority.
- **Assign Seat to Waitlisted User**: Assigns an available seat to the highest-priority user in the waitlist, promoting fairness.
    - **Functions Used**:
        - **extractMin (MinHeap)**: Retrieves the highest-priority user.
        - **extractMin (SeatMinHeap)**: Gets the next available seat.
        - **insert (RedBlackTree)**: Creates a new reservation for the user.
        - **write (BufferedWriter)**: Logs the seat assignment.
- **Update Waitlist Priority**: Adjusts a user's priority level, reordering the waitlist dynamically.
    - **Functions Used**:
        - **remove (MinHeap)**: Temporarily removes the user.
        - **insert (MinHeap)**: Re-inserts the user with a new priority.
        - **write (BufferedWriter)**: Logs the priority update.

3.2.5 Generate Report

To provide a clear view of all active reservations, this operation creates an ordered report using the Red-Black Tree. It enables easy tracking of seat assignments and reservation status.

Functions Used:

- **inorder_traversal(RedBlackTree)**: Traverses the tree to generate a sorted reservation list.
- **write(BufferedWriter)**: Records each reservation entry in the output file for reporting.

4. FUNCTION PROTOTYPES

The function prototypes in the GatorTicketMaster system define the main operations for managing reservations, handling available seats, and tracking the waitlist. The Red-Black Tree is used for efficient reservation management, and the SeatMinHeap organizes available seats in ascending order. Each function prototype supports essential operations that keep the system organized, allowing quick reservation lookups, efficient seat assignments, and clear management of available seats.

Red-Black Tree Operations

```
class RedBlackTree {
   void insert(int userId, int seatId); // Add a new reservation
   RBNode search(int userId); // Find reservation by user ID
   void delete(RBNode node); // Remove a reservation
}
```

SeatMinHeap Operations

```
class SeatMinHeap {
   void insert(int seatId, int priority); // Add available seat
   HeapNode extractMin(); // Get the lowest available seat number
   boolean isEmpty(); // Check if there are available seats
   int size(); // Get the number of available seats
```

}

Core Operations

public gatorTicketMaster(String outputFile) throws IOException; // Constructor to initialize system

public void initialize(int seatCount) throws IOException; // Initialize seats in the system

public void available() throws IOException; // Display available seats and waitlist count

public void reserve(int userId, int userPriority) throws IOException; // Reserve a seat for a user

public void cancel(int seatId, int userId) throws IOException; // Cancel a user's reservation

public void exitWaitlist(int userId) throws IOException; // Remove a user from the waitlist

public void updatePriority(int userId, int userPriority) throws IOException; // Update user's priority in waitlist

public void addSeats(int count) throws IOException; // Add additional seats and assign to waitlist users

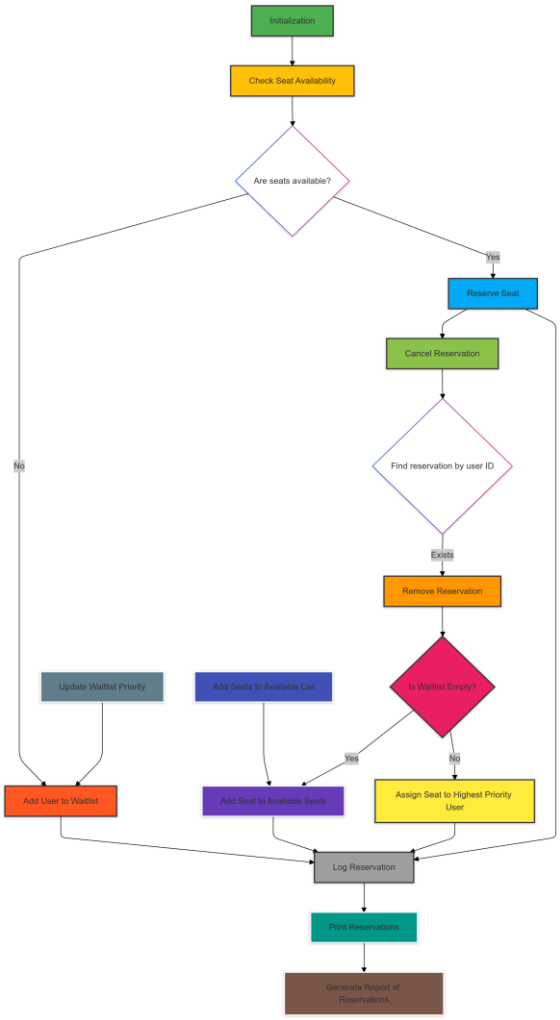
public void printReservations() throws IOException; // Print all current reservations

private void inorderTraversal(RBNode node, List<RBNode> result); // Helper for printReservations (inorder traversal)

public void releaseSeats(int userID1, int userID2) throws IOException; // Release seats for a range of user IDs

5. PROGRAM FLOW

The GatorTicketMaster system's program flow begins with initialization, followed by key operations such as checking seat availability, handling reservations and cancellations, managing the waitlist, and generating reports. Each operation uses specific data structures (Red-Black Tree and MinHeap) to ensure that seat assignments are orderly and responsive to user needs.



1. **Initialization**
    a. The system begins by initializing the primary data structures. The total number of seats is loaded into the SeatMinHeap in ascending order for efficient seat assignment. A RedBlackTree is set up to handle reservations, and a BufferedWriter is prepared to log each system action.
2. **Check Seat Availability**
    a. The available() function provides a quick snapshot of the current seat availability by checking the number of available seats in SeatMinHeap and the count of users on the waitlist.
3. **Seat Reservation**
    a. When a user requests a seat through the reserve() function:
        i. **If seats are available**: The system assigns the lowest-numbered seat by extracting it from SeatMinHeap and adds the reservation to the

RedBlackTree. The reservation details are logged using the BufferedWriter.

ii. **If no seats are available**: The user is added to a priority-based waitlist, ensuring that they are considered for the next available seat.

4. **Cancel Reservation**
    a. The cancel() function handles seat cancellations:
        i. The system first searches the RedBlackTree to verify the reservation.
        ii. **If the reservation exists**, it is removed from the tree, and the seat is returned to the system.
            1. **If the waitlist is not empty**: The system assigns the newly available seat to the highest-priority user on the waitlist.
            2. **If the waitlist is empty**: The seat is re-added to SeatMinHeap.
        iii. All actions are logged to maintain an accurate record.
5. **Update Waitlist Priority**
    a. Users on the waitlist can adjust their priority through the updatePriority() function. The system reorders the user's position in the waitlist based on the new priority, ensuring the highest-priority users are first in line for available seats.
6. **Add Seats**
    a. The addSeats() function adds new seats to the system:
        i. New seats are first assigned to users on the waitlist, with the highest-priority users given seats first.
        ii. Any remaining seats are added to SeatMinHeap to be available for future reservations.
7. **Print Reservations**
    a. The printReservations() function creates a report of all active reservations by performing an inorder traversal of the RedBlackTree. This function outputs a sorted list of reservations by seat number, creating a clear record of all current bookings.
8. **Release Seats for a Range of User IDs**
    a. The releaseSeats() function releases seats for a specified range of user IDs:
        i. Seats associated with users in the range are freed, and these users are removed from both the reservation list and waitlist.
        ii. The freed seats are then reassigned to waitlisted users based on priority, with remaining seats added back to SeatMinHeap for general availability.

6. TIME & SPACE COMPLEXITY

6.1 Red-Black Tree Operations

Operation	Time Complexity	Space Complexity
Insert	O(log n)	O(1)
Delete	O(log n)	O(1)
Search	O(log n)	O(1)
LeftRotate	O(1)	O(1)
RightRotate	O(1)	O(1)
FixViolation	O(log n)	O(1)
FixDelete	O(log n)	O(1)
FindSuccessor	O(log n)	O(1)

6.2 Heap Operations – MinHeap and Priority Queue

Operation	Time Complexity	Space Complexity
Insert	O(log n)	O(1)
ExtractMin	O(log n)	O(1)
HeapifyUp	O(log n)	O(1)
HeapifyDown	O(log n)	O(1)
UpdatePriority	O(n)	O(1)
Remove	O(n)	O(1)
Size/IsEmpty	O(1)	O(1)

6.3 Core Operations

Operation	Time Complexity	Space Complexity
Initialize	O(n)	O(n)
Available	O(1)	O(1)
Reserve	O(log n)	O(1)
Cancel	O(log n)	O(1)
ExitWaitlist	O(n)	O(1)
UpdatePriority	O(n)	O(1)
AddSeats	O(k log k)	O(k)
PrintReservations	O(n log n)	O(n)
ReleaseSeats	O(m log n)	O(m)

where,

- **n**: Total number of seats/reservations in the system
- **k**: Number of new seats being added in AddSeats operation

- **m**: Range of users in ReleaseSeats operation (userID2 - userID1 + 1)

7. PROGRAM EXECUTION

Step-1: Download and Extract

Extract the contents from BalaSuryaKrishna_Vankayala.zip Navigate to the extracted directory BalaSuryaKrishna_Vankayala

Step-2: Compilation

Build the program using ***make*** command

Step-3: Execution

Run the program by providing test files:

java gatorTicketMaster test1.txt

java gatorTicketMaster test2.txt

java gatorTicketMaster test3.txt

Step-4: View Results

Check output in the generated files:

test1_output_file.txt

test2_output_file.txt

test3_output_file.txt

Step-5: Cleanup

Remove all generated files using ***make clean*** command

8. CONCLUSION

The **GatorTicketMaster** system is a well-structured, efficient solution for managing seat reservations, cancellations, and waitlists in high-demand environments like event booking and ticketing systems. Leveraging optimized data structures—such as the Red-Black Tree for quick reservation management and MinHeap for priority-based seat allocation—the system ensures that seat assignments are handled in an organized and responsive manner. Its comprehensive program flow, which includes real-time updates, logging, and reporting, provides a transparent record of activities, enhancing reliability and accountability.

Through this project, I gained valuable experience in implementing data structures for high-performance systems, learning how balanced and priority-based structures contribute to both speed and fairness in resource allocation. The project reinforced the importance of designing scalable and user-focused systems, and it underscored the practical benefits of transparency in operations. Overall, GatorTicketMaster not only met the goals of creating a responsive reservation system but also deepened my skills in data management and program flow design for real-world applications.