**NAME** - Anisa Shaik
**UFID** – 49373585
**MAIL** – anisa.shaik@ufl.edu

# Gator Ticket Master Project Report

## 1. Project Structure

### 1.1 Overview of Project Architecture

#### Main Components:

***a) RedBlack Tree:***

- Manage reserved seat information.
- Each node contains User ID (key for the tree) and Seat ID

***b) Binary Min-Heaps:***

**Waitlist Heap:**

- Manage users waiting for a seat
- Each node contains User ID, User Priority (used as the key for the heap) and Timestamp (for tie-breaking)

**Available Seats Heap:**

- Manage unassigned seat numbers
- Each node contains Seat ID (used as the key for the heap).

#### Component Interactions:

***a) Seat Reservation Process:***

When a user requests a reservation, the system checks the Available Seats Heap
**If seats are available**: extract the minimum seat number and insert the user-seat pair into the RedBlack Tree.
**else**: Insert user into Waitlist Heap based on their priority

***b) Seat Cancellation Process:***

When a user cancels a reservation, Remove the user-seat pair from RedBlack Tree
**If the Waitlist Heap isn't empty:**

- Extract highest priority user from Waitlist Heap
- Assign emptied seat to this user, simultaneously insert new user-seat pair to the RedBlack

**If the Waitlist Heap is empty:**

- Insert the freed seat number back into the Available Seats Heap

***c) Adding New Seats:***

1. Insert new seat numbers into the Available Seats Heap

**NAME** - Anisa Shaik
**UFID** – 49373585
**MAIL** – anisa.shaik@ufl.edu

2. If there are users in Waitlist Heap: Extract users from Waitlist Heap and assign it to newly available seats. Then insert these user-seat pairs into RedBlack Tree.

***d) Releasing Seats:***

1. For each user in the specified range:
    - Remove the user-seat pair from RedBlack Tree and Insert freed seat number into Available Seats Heap
2. Remove any users in specified range from Waitlist Heap
3. Assign newly available seats to users in the Waitlist Heap.

## 1.2 Data Flow

### a) Initialization:
The system creates Available Seats Min-Heap with all seat numbers, RedBlack Tree for reservations and Waitlist Min-Heap initialized as empty.

### b) Seat Reservation:
The system checks the Available Seats Min-Heap.
- If seats are available - extract minimum seat number from Available Seats Min-Heap and insert the user-seat pair into RedBlack Tree.
- Else - insert user into Waitlist Min-Heap based on priority and timestamp.

### c) Seat Cancellation:
Will remove user-seat pair from RedBlack Tree.
If Waitlist Min-Heap is not empty:
- Extract highest priority user from Waitlist Min-Heap.
- Assign freed seat and insert new user-seat pair into RedBlack Tree.

If Waitlist Min-Heap is empty:
- Insert freed seat number back into Available Seats Min-Heap.

### d) Adding New Seats:
Insert new seat numbers into Available Seats Min-Heap.
While seats are available, and Waitlist Min-Heap is not empty:
- Extract highest priority user from Waitlist Min-Heap.
- Extract minimum seat number from Available Seats Min-Heap.
- Insert new user-seat pair into RedBlack Tree.

### e) Releasing Seats:
For each user in specified range:
- Remove user-seat pair from RedBlack Tree.
- Insert freed seat number into Available Seats Min-Heap.

**NAME** - Anisa Shaik
**UFID** – 49373585
**MAIL** – anisa.shaik@ufl.edu

Remove any users in specified range from Waitlist Min-Heap.
Assign newly available seats to users in Waitlist Min-Heap (if any).

### Interaction between RedBlack Tree and Min-Heaps:

***1. RedBlack Tree and Available Seats Min-Heap:***

- When a seat is reserved, it's removed from the Available Seats Min-Heap and added to the RedBlack Tree.
- When a reservation is canceled and no one is on the waitlist, the seat is removed from the RedBlack Tree and added back to the Available Seats Min-Heap.

***2. RedBlack Tree and Waitlist Min-Heap:***

When a seat becomes available (due to cancellation or addition of new seats) and the Waitlist Min-Heap is not empty:

a. The highest priority user is removed from the Waitlist Min-Heap.

b. A new reservation is created and added to the RedBlack Tree.

***3. Coordinated Operations:***

During the ReleaseSeats operation, seats are removed from the RedBlack Tree and added to the Available Seats Min-Heap. Simultaneously, users in the specified range are removed from the Waitlist Min-Heap. Then, newly available seats are assigned to waiting users, moving data from the Available Seats Min-Heap to the RedBlack Tree, and removing users from the Waitlist Min-Heap.

## 2. Makefile

Since python is an interpreted language and not a compiled langauge, it doesn't require a makefile. I have added makefile with comments in the zip folder of the project (if applicable)

## 3. Function Prototypes

### 3.1 Core Functions

**1. def initialize(self, seat_count: int) -** Initializes the event with a specified number of seats.

**2. def available(self) -** Prints the number of available seats and the length of the waitlist.

3. **def reserve(self, user_id: int, user_priority: int)-** Reserves a seat for a user or adds them to the waitlist.

**NAME** - Anisa Shaik
**UFID** – 49373585
**MAIL** – anisa.shaik@ufl.edu

- user_id: Integer representing the unique identifier of the user.
- user_priority: Integer represents user's priority.

**4. def cancel(self, seat_id: int, user_id: int)** - Cancels a user's reservation and reassigns the seat if there's a waitlist.

- seat_id: Integer representing the seat to be canceled.
- user_id: Integer representing the user canceling the reservation

**5. def exit_waitlist(self, user_id: int)** - Removes a user from the waitlist.

- user_id: Integer representing the user to be removed from the waitlist.

**6. def update_priority(self, user_id: int, new_priority: int)** - Updates the priority of a user in the waitlist.

- user_id: Integer representing the user whose priority is to be updated.
- new_priority: Integer representing the new priority value.

**7. def add_seats(self, count: int) -** Adds new seats to the event and assigns them to waitlisted users if applicable.

- count: Integer representing the number of new seats to add.

**8. def print_reservations(self) -** Prints all assigned seats and their corresponding users.

**9. def release_seats(self, user_id1: int, user_id2: int)** - Releases seats assigned to users within a specified ID range.

- user_id1: Integer representing the lower bound of the user ID range.
- user_id2: Integer representing the upper bound of the user ID range.

## 3.2 Helper Functions

1. **def _heapify_up(self, i: int) :**
   - Purpose: Maintains the heap property by moving an element up the heap.
   - Role: Used in the MinHeap class to ensure correct ordering after insertion.
2. **def _heapify_down(self, i: int)**:
   - Purpose: Maintains the heap property by moving an element down the heap.
   - Role: Used in the MinHeap class to ensure correct ordering after extraction.
3. **def _fix_insert(self, node: Node):**
   - Purpose: Fixes the RedBlack tree properties after insertion.
   - Role: Ensures the RedBlack tree remains balanced in the RedBlackTree class.
4. **def _fix_delete(self, node: Node):**
   - Purpose: Fixes the RedBlack tree properties after deletion.

**NAME** - Anisa Shaik
**UFID** – 49373585
**MAIL** – anisa.shaik@ufl.edu

- Role: Ensures the  RedBlack tree remains balanced in the RedBlackTree class.

5. **def _left_rotate(self, node: Node)**:
    - Purpose: Performs a left rotation on the given node in the  RedBlack tree.
    - Role: Helper function for maintaining  RedBlack tree properties.
6. **def _right_rotate(self, node: Node)**:
    - Purpose: Performs a right rotation on the given node in the  RedBlack tree.
    - Role: Helper function for maintaining  RedBlack tree properties.
7. **def _compare(self, item1: tuple, item2: tuple) -> int**:
    - Purpose: Compares two items in the MinHeap based on priority and timestamp.
    - Role: Ensures correct ordering in the waitlist and available seats heaps.

## 4. Implementation Details

### 4.1 RedBlack Tree Implementation

The RedBlack Tree is used to manage reserved seat information. Each node in the tree contains:

- User ID (key for the tree)
- Seat ID
- Color (Red or Black)
- Parent, Left, and Right child pointers

**Key operations:**

1. Insert:
    - Standard BST insertion
    - Color the new node red
    - Perform fix-up to maintain RedBlack properties:
        - Recolor and rotate nodes as needed
        - Ensure the root is always black
2. Delete:
    - Standard BST deletion
    - If the deleted node was black, perform fix-up:
        - Handle cases of double-black nodes
        - Re-color and rotate as needed to restore RedBlack properties
3. Search:
    - Standard BST search operation
    - Traverse the tree based on the User ID

### 4.2 Binary Min-Heap Implementation

Two Binary Min-Heaps are used:

**NAME** - Anisa Shaik
**UFID** – 49373585
**MAIL** – anisa.shaik@ufl.edu

1. For managing the waitlist
2. For managing available seats

**Structure**:

- Implemented as an array
- Parent of node i is at (i-1)/2
- Left child of node i is at 2i + 1
- Right child of node i is at 2i + 2

**Key operations:**

1. Insert:
    - Add element to the end of the array
    - Perform heapify-up to maintain heap property
2. Extract-min:
    - Remove the root element (minimum)
    - Replace root with the last element
    - Perform heapify-down to maintain heap property

## 4.3 Seat Management

Available seats are managed using a Binary Min-Heap:

- Each seat is represented by its seat number
- The Min-Heap property ensures the lowest seat number is always at the root

### Seat assignment:

1. Extract the minimum seat number from the available seats heap
2. Assign this seat to the user
3. Insert the user-seat pair into the  RedBlack Tree

### Seat release:

1. Remove the user-seat pair from the  RedBlack Tree
2. If there are users in the waitlist:
    - Assign the seat to the highest priority user in the waitlist
    - Update the  RedBlack Tree
3. If no users in the waitlist:
    - Insert the seat number back into the available seats heap

This approach ensures efficient seat assignment and release operations.

## 4.4 Waitlist Management

The waitlist is implemented as a Binary Min-Heap:

- Each entry in the heap is a tuple: (-priority, timestamp, user_id)

**NAME** - Anisa Shaik
**UFID** – 49373585
**MAIL** – anisa.shaik@ufl.edu

- Using negative priority ensures higher priority users are at the top of the heap
- Timestamp is used as a tie-breaker for equal priorities

### Priority handling:

- Higher numeric priority values represent higher actual priority
- The negative of the priority is used in the heap to create a max-heap effect

### Timestamp handling:

- Timestamps are generated when a user is added to the waitlist
- They ensure a first-come-first-served order for users with equal priorities

When seats become available:

1. Extract the top entry from the waitlist heap
2. Assign the available seat to this user
3. Update the  RedBlack Tree with the new reservation

## 5. Operation Implementations

### 5.1 Reserve Operation

***Implementation Approach:***

- Check if there are available seats using a Min-Heap.
- If seats are available, extract the minimum seat number and assign it to the user.
- If no seats are available, add the user to the waitlist Min-Heap with their priority and a timestamp.

***Reasons for Choosing This Approach:***

- Using a Min-Heap ensures that the lowest seat number is always assigned first, adhering to the requirement of prioritizing lower seat numbers.
- The waitlist is managed using a priority queue (Min-Heap) to efficiently handle user priorities and timestamps.

***Advantages:***

- Efficient seat assignment with O(log n) complexity due to heap operations.
- Fair waitlist management based on user priority and arrival time.

***Disadvantages:***

- Complexity increases with many users due to heap operations.

**Time and Space Complexity:**

- Time Complexity: O(log n) for both seat assignment and adding to the waitlist.

**NAME** - Anisa Shaik
**UFID** – 49373585
**MAIL** – anisa.shaik@ufl.edu

- Space Complexity: O(n) for storing users in heaps.

## 5.2 Cancel Operation.

***Implementation Approach:***

- Search for the user's reservation in the RedBlack Tree.
- If found, remove the reservation and check the waitlist.
- If the waitlist is not empty, assign the seat to the highest priority user from the waitlist.
- If the waitlist is empty, add the seat back to available seats.

***Reasons for Choosing This Approach:***

- The RedBlack Tree allows efficient search and deletion of reservations.
- The Min-Heap for available seats ensures efficient reassignment of seats.

***Advantages*:**

- Efficient cancellation and reassignment with O(log n) complexity.
- Maintains data structure integrity with balanced tree operations.

***Disadvantages:***

- Complexity in maintaining  RedBlack Tree properties during deletion.

***Time and Space Complexity:***

- Time Complexity: O(log n) for both searching in the tree and heap operations.
- Space Complexity: O(n) for storing reservations and available seats.

## 5.3 ReleaseSeats Operation

***Implementation Approach:***

- Traverse RedBlack Tree to find users within the specified ID range.
- Remove their reservations and add seats back to available seats.
- Remove users from the waitlist if they fall within the range.

***Reasons for Choosing This Approach:***

- Efficient traversal of  RedBlack Tree maintains order by User ID.

***Advantages:***

- Efficient release of multiple reservations in one operation.

***Disadvantages:***

- Traversal can become complex with a large number of nodes.

***Time and Space Complexity*:**

**NAME** - Anisa Shaik
**UFID** – 49373585
**MAIL** – anisa.shaik@ufl.edu

- Time Complexity: O(n) for traversing nodes within range.
- Space Complexity: O(n) for storing released seats.

### 5.4 AddSeats Operation

***Implementation Approach:***

- Add new seat numbers sequentially to the Min-Heap of available seats.

***Reasons for Choosing This Approach:***

- Ensures new seats are added efficiently while maintaining order.

***Advantages*:**

- Simple addition of seats with minimal overhead.

***Disadvantages:***

- Requires careful management of seat numbering to avoid conflicts.

***Time and Space Complexity:***

- Time Complexity: O(log n) per seat addition due to heap operations.
- Space Complexity: O(n) for storing new seats.

## 6. Technical Specifications

### 6.1 Priority Handling

User priorities are managed using a Binary Min-Heap for the waitlist, with the following specification:

- In the waitlist heap, priorities are stored as negative values to create a max-heap effect within the min-heap structure.

***Implementation of priority handling:***

1. When a user is added to the waitlist:
    - The priority is negated and stored in the heap tuple: (-priority, timestamp, user_id)
    - This ensures that higher priority users with more negative values are at top of heap.
2. Priority updates:
    - The UpdatePriority(userID, userPriority) function is used to modify a user's priority in the waitlist.
    - Function searches for the user in the waitlist heap.- If found, it updates the priority value, by maintaining original timestamp.

- After updating, the heap is re-balanced using _heapify_up and _heapify_down operations to ensure correctness.

## 6.2 Timestamp Management

Timestamps are important for breaking ties between users with the same priority in the waitlist:

- When a user is added to the waitlist, a timestamp is generated using datetime.now().timestamp()+ time.time_ns() % 1 * 1e-9
- The timestamp is stored as the second element in the heap tuple: (-priority, timestamp, user_id)

***Tie-breaking process:***

1. When comparing two users in the waitlist with the same priority, the timestamp is used as a secondary comparison.
2. The user with the earlier timestamp is given higher preference to implement a first-come-first-served policy for equal priorities.

This approach ensures fairness in the system, where users who joined the waitlist earlier are served first when priorities are equal.

## 6.3 Edge Case Handling

The system handles various edge cases to ensure robust operation:

1. Canceling non-existent reservations:
    - In the Cancel(seatID, userID) function, the system first checks if the user has a reservation for the specified seat.
    - If no such reservation exists, it prints: "User <userID> has no reservation for seat <seatID> to cancel"
2. Updating priority for users not in the waitlist:
    - The UpdatePriority(userID, userPriority) function first checks if the user is in the waitlist.
    - If the user is not found, it prints: "User <userID> priority is not updated"
3. Exiting waitlist for users not in the waitlist:
    - The ExitWaitlist(userID) function checks if the user is in the waitlist.
    - If not found, it prints: "User <userID> is not in waitlist"
4. Adding invalid number of seats:
    - The AddSeats(count) function checks if the input is a positive integer.
    - If not, it prints: "Invalid input. Please provide a valid number of seats."
5. Releasing seats with invalid user ID range:
    - The ReleaseSeats(userID1, userID2) function checks if the range is valid (userID2 >= userID1 and both are non-negative).
    - If invalid, it prints: "Invalid input. Please provide a valid range of users."

**NAME** - Anisa Shaik
**UFID** – 49373585
**MAIL** – anisa.shaik@ufl.edu

6. Handling empty waitlist when canceling a reservation:
    - If a user cancels their reservation and the waitlist is empty, the seat is added back to the available seats heap instead of being reassigned.

These edge case handling mechanisms ensure that the system remains stable and provides appropriate feedback even when faced with unexpected or invalid inputs.

## 7. Performance Considerations

### 7.1 Time Complexity Analysis

1. ***RedBlack Tree Operations***:
    - Insert, Delete, Search: O(log n) Where n is the number of reserved seats.
2. ***Binary Min-Heap Operations***:
    - Insert, Extract-min: O(log m) Where m is the number of elements in the heap.

***Major Operations:***

- Initialize(seatCount): O(n log n), where n is the number of seats
- Reserve(userID, userPriority): O(log n)
- Cancel(seatID, userID): O(log n)
- UpdatePriority(userID, userPriority): O(m), where m is the size of the waitlist
- AddSeats(count): O(k log m), where k is the number of new seats and m is the size of the waitlist
- PrintReservations(): O(n log n), where n is the number of reserved seats
- ReleaseSeats(userID1, userID2): O((k + m) log n), where k is the number of seats released and m is the size of the waitlist

### 7.2 Space Complexity Analysis

- RedBlack Tree: O(n), where n is the number of reserved seats
- Available Seats Heap: O(k), where k is the number of available seats
- Waitlist Heap: O(m), where m is the number of users in the waitlist

**Total Space Complexity**: O(n + k + m)

## 8. Testing Approach

### 8.1 Test Cases

My testing approach involved a comprehensive set of test cases designed to cover various scenarios and edge cases. Here are the key test cases and their purposes:

Testcases 1, Testcases 2, Testcases 3 are from provided TestCases_v2.pdf

**NAME** - Anisa Shaik
**UFID** – 49373585
**MAIL** – anisa.shaik@ufl.edu

***Basic Functionality Test (Testcase 1):***

- Checks the system's ability to handle basic operations in sequence such as tests initialization, reservation, cancellation, and seat availability.

***Complex Scenario Test (Testcase 2):***

- Tests larger number of seats and more complex operations such as priority updates, releasing seats, and adding new seats. And also verifies the system's ability to handle a mix of operations with varying priorities.

***Edge Case and Priority Management Test (Testcase 3):***

- Focuses on waitlist management, priority updates, and edge cases like tests exiting the waitlist and updating priorities of waitlisted users.

***Key Edge Cases Tested:***

1. Canceling non-existent reservations:
   - Example: Cancel(2, 4) in Testcase 1 when seat 2 is not reserved.
2. Updating priority for users not in the waitlist:
   - Example: UpdatePriority(6, 3) in Testcase 3 after seat assignment.
3. Exiting waitlist for users not in the waitlist:
   - Example: ExitWaitlist(9) in Testcase 3 after seat assignment.
4. Releasing seats with a range of user IDs:
   - Example: ReleaseSeats(10, 100) in Testcase 2 to test bulk seat release.
5. Adding seats when there's a waitlist:
   - Example: AddSeats(3) in Testcase 3 to test seat assignment to waitlisted users.
6. Handling operations after program termination:
   - Example: Release(1, 12) and PrintReservations() after Quit() in Testcase 3.

Note – Output files are included in the project zip file (All example and usecases mentioned are attached)

## 9. Conclusion

The Gator Ticket Master project successfully implemented - A robust seat booking service for GatorEvents that meets all the specified requirements. The system effectively handles various scenarios such as seat reservations, cancellations, priority updates, and dynamic seat additions, also maintaining efficiency and data integrity. The use of advanced data structures like RedBlack Trees and Binary Heaps ensured optimal performance for various operations.

**NAME** - Anisa Shaik
**UFID** – 49373585
**MAIL** – anisa.shaik@ufl.edu

## 10. References

- Mehta, D.P., & Sahni, S. (Eds.). (2017). Handbook of Data Structures and Applications (2nd ed.). Chapman and Hall/CRC. https://doi.org/10.1201/9781315119335
- https://www.cise.ufl.edu/~sahni/cop5536/index.html