Gator Ticker Master

Varun Aiyaswamy Kannan
UFID: 31555559
varunaiyaswamyka@ufl.edu

Introduction

The Reservation Management System is designed to efficiently handle seat reservations, cancellations, and waitlist management. This system supports operations such as adding or releasing seats, canceling reservations, updating priorities, and printing current reservations. The underlying architecture of the system leverages advanced data structures, including a **Red-Black Tree** for active reservations and a **Min Heap** for managing the waitlist. These structures enable efficient, scalable, and dynamic management of reservation data.

Core Functionalities

- **Reservation Management**: Facilitates seat reservations and cancellations based on availability.
- **Priority Queue Waitlist**: Manages users on a waitlist by assigning seats based on priority.
- **Red-Black Tree**: Ensures efficient management of active reservations through fast lookups, insertions, and deletions.
- **Seat Addition and Release**: Dynamically adds new seats and reallocates them as necessary.
- **Reservation Overview**: Displays all current reservations in sorted order.

Core Functions

1. `__init__`(self, initial_seat_count: int)

Purpose: Initializes the system with a specified number of seats and sets up the data structures for managing reservations and the waitlist.

2. `Reserve(self, userID: int, userPriority: int)`

Purpose: Reserves a seat for a user, either immediately or by adding them to the waitlist if no seats are available. Users on the waitlist are prioritized based on their priority level.

3. `Cancel(self, seatID: int, userID: int)`

Purpose: Cancels an existing reservation and reassigns the seat to the highest-priority user on the waitlist, if applicable.

4. `ExitWaitlist(self, userID: int)`

Purpose: Removes a user from the waitlist, re-heapifying the waitlist afterward.

5. `UpdatePriority(self, userID: int, userPriority: int)`

Purpose: Updates the priority of a user on the waitlist and ensures that the queue is reordered accordingly.

6. `AddSeats(self, count: int)`

Purpose: Adds new seats to the system and assigns them to the highest-priority users from the waitlist.

7. `PrintReservations(self)`

Purpose: Prints a sorted list of current reservations by seat number, utilizing an in-order traversal of the Red-Black Tree.

8. `ReleaseSeats(self, userID1: int, userID2: int)`

Purpose: Releases all reservations for users within a specified ID range, returning the seats to the available pool and reallocating them to users on the waitlist.

9. `Quit(self)`

Purpose: Terminates the program, ensuring a graceful shutdown of the system.

Data Structures Used

1. Red-Black Tree

Overview: A **Red-Black Tree** is a self-balancing binary search tree that maintains a balanced structure through color-coded nodes. This ensures logarithmic time complexity for all basic operations, such as searching, inserting, and deleting.

Usage in the System: The Red-Black Tree is used to efficiently store and manage reservations, allowing for quick lookup, insertion, and deletion of seat reservations. Each node in the tree represents a reservation, indexed by userID and seatID. This ensures that reservations are easily searchable and maintain optimal performance even with large datasets.

Benefits:

- **Efficient Operations**: Guarantees that search, insertion, and deletion operations all have a time complexity of $O(\log n)$.
- **Balanced Structure**: Maintains balance throughout the lifetime of the tree, ensuring consistent performance even as reservations are added or removed.

2. Min Heap

Overview: A **Min Heap** is a complete binary tree where each node's value is smaller than or equal to its children's values, ensuring that the root node contains the smallest element. In this system, the heap is used to manage the waitlist, where the user with the highest priority (lowest priority value) is always at the root.

Usage in the System: The Min Heap is employed to manage users waiting for seats. Each user on the waitlist is represented as a tuple consisting

of **priority**, **timestamp**, and **userID**. When a seat becomes available, the user at the root of the heap (the one with the highest priority) is assigned the seat. If the seat is not available immediately, the user is kept in the heap until a seat is released.

Benefits:

- **Efficient Priority Management**: Accessing and assigning seats to the highest-priority user is done in constant time, $O(1)$.
- **Logarithmic Time Complexity**: Both insertions and deletions from the heap take $O(\log n)$ time, making the system scalable even as the number of users grows.
- **Dynamic Priority Updates**: The heap supports dynamic priority changes, ensuring that the user priorities can be efficiently updated as needed.

3. PriorityHeap (Custom Class)

Overview: The **PriorityHeap** is a custom implementation of a Min Heap designed specifically for handling the priority and timestamp of users on the waitlist.

Usage in the System: This class handles operations such as inserting users into the waitlist based on their priority and timestamp, and efficiently removing the user with the highest priority when a seat becomes available.

Benefits:

- **Tailored for Waitlist Management**: Provides specialized handling of user priorities and timestamps, maintaining the heap property for efficient operations.
- **Simple Implementation**: The custom heap class simplifies operations related to priority-based user management without additional complexity.

Conclusion

The Reservation Management System leverages efficient data structures—specifically a **Red-Black Tree** and a **Min Heap**—to manage seat reservations and waitlist operations. The Red-Black Tree ensures fast and balanced management of reservations, while the Min Heap provides efficient handling of the waitlist, ensuring that users are seated according to priority. These data

structures collectively enable the system to perform efficiently and scale well with increasing user demands.