README

Gator Ticket Master System

Name: Bhanu Prasad Cherukuvada

UFID: 8697 4783

Email: b.cherukuvada@ufl.edu

This project implements a Gator Ticket Master system that manages seat reservations, cancellations, and a waitlist for users. It utilizes data structures like Red-Black Trees and Min-Heaps to efficiently handle the operations. The system processes commands from an input file and generates an output file with the results of each operation.

Table of Contents

Prerequisites

- **Python 3.6 or higher**: Ensure you have Python installed on your system.

Installation

1. **Clone or Download the Repository**

   Download the project files to your local machine.

2. **Install Required Packages and generate executable**

   Open a terminal in the project directory and run:

```
make
```

3. **Included Executable**

   For convenience, **the executable is already included in the root project directory**

Optional Installation with Docker

1. **Build the Docker Image**

   1. To build the Docker image, run the following command:

      ```
      docker build -t gator_ticket_master_image .
      ```

   2. Copy the Executable to Container to Host

      ```
      docker cp temp_container:/app/gatorTicketMaster
      ./gatorTicketMaster
      ```

   3. Remove the temporary container

      ```
      docker rm temp_container
      ```

   4. Give permissions to the executableq

      ```
      chmod +x gatorTicketMaster
      ```

Using the Executable

To use the executable, follow these steps:

1. **Prepare the Input File**

   - Create an input file (e.g., `testcase1.txt`) containing the commands for the ticket system.
   - Ensure the file is in the same directory as the executable. Run the Executable

   ```
   ./gatorTicketMaster testcase1.txt
   ```

   - Replace input1.txt with the name of your input file.

- On Windows, you may need to use `gatorTicketMaster.exe` instead.

2. **Output File**

    - The program generates an output file named `testcase1_output_file.txt`.
    - This file contains the results of the operations specified in the input file.

Input and Output Files

- Input File Format

  The input file should contain commands, one per line. Commands include:

    - `Initialize(<number_of_seats>)`
    - `Reserve(<userID>, <userPriority>)`
    - `Cancel(<seatID>, <userID>)`
    - `ExitWaitlist(<userID>)`
    - `UpdatePriority(<userID>, <newPriority>)`
    - `AddSeats(<number_of_seats>)`
    - `PrintReservations`
    - `ReleaseSeats(<userID_start>, <userID_end>)`
    - `Available`
    - `Quit`

- Output File

  The output file contains messages generated by the system in response to each command, following the specified formats.

Commands Overview

- **Initialize(N)**

    - Initializes the system with N seats.
    - Seats are numbered starting from 1.

- **Reserve(userID, priority)**

    - Reserves a seat for the user if available.
    - If no seats are available, the user is added to the waitlist.

- **Cancel(seatID, userID)**

    - Cancels the reservation for the specified seat and user.
    - The seat is reallocated to the next user in the waitlist if any.

- **ExitWaitlist(userID)**

    - Removes the user from the waitlist.

- **UpdatePriority(userID, newPriority)**

- Updates the user's priority in the waitlist.

- **AddSeats(N)**
  - Adds N new seats to the system.
  - Assigns seats to users in the waitlist if any.

- **PrintReservations**
  - Prints all current reservations in order of seat numbers.

- **ReleaseSeats(userID_start, userID_end)**
  - Releases reservations and waitlist entries for users within the specified range.

- **Available**
  - Displays the number of available seats and the length of the waitlist.

- **Quit**
  - Terminates the program.

Code Logic and Functionality

Overview

The system manages seat reservations and a waitlist using efficient data structures:

- Available Seats: Managed using a Min-Heap (`SeatHeap`) to always allocate the lowest available seat number.
- Reservations: Stored in a Red-Black Tree (`RedBlackTree`) for efficient insertion, deletion, and in-order traversal.
- Waitlist: Managed using a Min-Heap (`MinHeap`) based on user priority and timestamp to ensure fair allocation.

Pseudocode Explanation

Below is a high-level pseudocode explanation of the system's functionality:

Initialization

```
Initialize variables:
    seat_heap = new SeatHeap()
    waitlist = new MinHeap()
    reservations = new RedBlackTree()
    user_to_seat = empty map
    next_seat_number = 1
    timestamp = 0
    user_in_waitlist = empty set
```

```
For each command in input_file:
    Process the command accordingly
```

Command Processing

1. **Initialize(N)**

```
For seatID from next_seat_number to next_seat_number + N - 1:
    seat_heap.push(seatID)
next_seat_number += N
Output: "{N} Seats are made available for reservation"
```

2. **Available**

```
Output: "Total Seats Available : {len(seat_heap)}, Waitlist :
{len(waitlist)}"
```

3. **Reserve(userID, priority)**

```
If userID in user_to_seat or userID in user_in_waitlist:
    Do nothing (user cannot reserve twice or already in waitlist)
Else if seat_heap is not empty:
    seatID = seat_heap.pop()
    reservations.insert(seatID, userID)
    user_to_seat[userID] = seatID
    Output: "User {userID} reserved seat {seatID}"
Else:
    timestamp += 1
    waitlist.push(priority, timestamp, userID)
    user_in_waitlist.add(userID)
    Output: "User {userID} is added to the waiting list"
```

4. **Cancel(seatID, userID)**

```
If userID in user_to_seat and user_to_seat[userID] == seatID:
    reservations.delete_node(seatID)
    Remove userID from user_to_seat
    Output: "User {userID} canceled their reservation"
    If waitlist is not empty:
        next_user = waitlist.pop()
        Remove next_user.userID from user_in_waitlist
        reservations.insert(seatID, next_user.userID)
        user_to_seat[next_user.userID] = seatID
        Output: "User {next_user.userID} reserved seat {seatID}"
    Else:
```

```
        seat_heap.push(seatID)
Else:
    Output: "User {userID} has no reservation for seat {seatID} to cancel"
```

5. **ExitWaitlist(userID)**

```
If userID in user_in_waitlist:
    waitlist.remove(userID)
    user_in_waitlist.remove(userID)
    Output: "User {userID} is removed from the waiting list"
Else:
    Output: "User {userID} is not in waitlist"
```

6. **UpdatePriority(userID, newPriority)**

```
If userID in user_in_waitlist:
    waitlist.update_priority(userID, newPriority)
    Output: "User {userID} priority has been updated to {newPriority}"
Else:
    Output: "User {userID} priority is not updated"
```

7. **AddSeats(N)**

```
For seatID from next_seat_number to next_seat_number + N - 1:
    seat_heap.push(seatID)
next_seat_number += N
Output: "Additional {N} Seats are made available for reservation"
While seat_heap is not empty and waitlist is not empty:
    seatID = seat_heap.pop()
    next_user = waitlist.pop()
    Remove next_user.userID from user_in_waitlist
    reservations.insert(seatID, next_user.userID)
    user_to_seat[next_user.userID] = seatID
    Output: "User {next_user.userID} reserved seat {seatID}"
```

8. **PrintReservations**

```
reservations_list = reservations.inorder()
For each (seatID, userID) in reservations_list:
    Output: "[seat {seatID}, user {userID}]"
```

9. **ReleaseSeats(userID_start, userID_end)**

```
users_to_release = set of userIDs from userID_start to userID_end
For userID in users_to_release:
    If userID in user_in_waitlist:
        waitlist.remove(userID)
        user_in_waitlist.remove(userID)
assignment_messages = empty list
For userID in users_to_release:
    If userID in user_to_seat:
        seatID = user_to_seat[userID]
        reservations.delete_node(seatID)
        Remove userID from user_to_seat
        While waitlist is not empty:
            next_user = waitlist.pop()
            Remove next_user.userID from user_in_waitlist
            If next_user.userID not in users_to_release:
                reservations.insert(seatID, next_user.userID)
                user_to_seat[next_user.userID] = seatID
                assignment_messages.append("User {next_user.userID}
reserved seat {seatID}")
                Break
        Else:
            seat_heap.push(seatID)
If assignment_messages is empty:
    Output: "Reservations/waitlist of the users in the range
[{userID_start}, {userID_end}] have been released"
Else:
    Output: "Reservations of the Users in the range [{userID_start},
{userID_end}] are released"
    For msg in assignment_messages:
        Output: msg
```

10. **Quit**

```
Output: "Program Terminated!!"
Terminate the program
```

Data Structures Used

1. SeatHeap (Min-Heap)
    - Manages available seat numbers.
    - Ensures the lowest seat number is assigned first.
2. RedBlackTree
    - Stores current reservations.
    - Allows for efficient insertion, deletion, and in-order traversal.
    - Maintains reservations sorted by seat numbers.

3. MinHeap (Waitlist)

    - Manages users waiting for a seat.
    - Orders users based on priority and timestamp.
    - Ensures fair allocation when seats become available.

4. Dictionaries and Sets

    - **user_to_seat**: Maps user IDs to their reserved seat IDs.
    - **user_in_waitlist**: Keeps track of users currently in the waitlist.

Functional Flow

- **Reservation Process**

    - When a user requests a reservation, the system checks for available seats.
    - If a seat is available, it is assigned to the user.
    - If no seats are available, the user is added to the waitlist.

- **Cancellation and Reassignment**

    - When a reservation is canceled, the seat becomes available.
    - The system checks the waitlist to assign the seat to the next eligible user.
    - If the waitlist is empty, the seat is added back to the available seats.

- **Waitlist Management**

    - Users in the waitlist are ordered by priority (higher priority gets the seat first).
    - If priorities are equal, the user who joined the waitlist earlier gets preference.

- **Adding Seats**

    - New seats can be added to the system at any time.
    - The system automatically assigns new seats to users in the waitlist if any.

- **Releasing Seats**

    - Reservations and waitlist entries for a range of user IDs can be released.
    - Seats freed are assigned to the next eligible users not in the release range.

Time Complexity Analysis

The following are the time complexities for the various operations performed by the system:

- **Initialize(N)**

    - **Time Complexity:** O(N log N)
    - **Explanation:**
        - Inserting `N` seats into the `SeatHeap` (Min-Heap) requires O(log N) time per insertion.
        - Total time: O(N log N).

- **Reserve(userID, priority)**

    - **Time Complexity:**
        - **Successful Reservation:** O(log N)
        - **Added to Waitlist:** O(log N)
    - **Explanation:**
        - Checking dictionaries (`user_to_seat`, `user_in_waitlist`): O(1).
        - If a seat is available:
            - Pop seat from `SeatHeap`: O(log N).
            - Insert reservation into `RedBlackTree`: O(log N).
        - If no seat is available:
            - Insert user into `MinHeap` (waitlist): O(log N).

- **Cancel(seatID, userID)**

    - **Time Complexity:** O(log N)
    - **Explanation:**
        - Remove reservation from `RedBlackTree`: O(log N).
        - Update dictionaries: O(1).
        - If reassigning seat to next user in waitlist:
            - Pop user from `MinHeap`: O(log N).
            - Insert reservation into `RedBlackTree`: O(log N).
        - Else:
            - Push seat back into `SeatHeap`: O(log N).

- **ExitWaitlist(userID)**

    - **Time Complexity:** O(log N)
    - **Explanation:**
        - Remove user from `MinHeap`: O(log N).
        - Update `user_in_waitlist`: O(1).

- **UpdatePriority(userID, newPriority)**

    - **Time Complexity:** O(log N)
    - **Explanation:**
        - Update user's priority in `MinHeap`: O(log N).

- **AddSeats(N)**

    - **Time Complexity:** O(N log N)
    - **Explanation:**
        - Inserting `N` new seats into `SeatHeap`: O(N log N).
        - While assigning seats to waitlisted users:
            - Pop seat from `SeatHeap`: O(log N) per seat.
            - Pop user from `MinHeap`: O(log N) per user.
            - Insert reservation into `RedBlackTree`: O(log N) per reservation.
        - Total time depends on the number of users assigned seats (up to `N`): O(N log N).

- **PrintReservations**

    - **Time Complexity:** O(N)

  - **Explanation:**
    - In-order traversal of `RedBlackTree` visits each node once.

- **ReleaseSeats(userID_start, userID_end)**
  - **Time Complexity:** O(M log N), where `M` is the number of users in the specified range.
  - **Explanation:**
    - For each user in the range:
      - Remove from `user_in_waitlist` (if present): O(1).
      - If the user has a reservation:
        - Remove reservation from `RedBlackTree`: O(log N).
        - Update `user_to_seat`: O(1).
        - Attempt to reassign seat:
          - Pop user from `MinHeap`: O(log N) per user until a valid user is found.
          - Insert reservation into `RedBlackTree`: O(log N).
    - The number of operations depends on `M` and the size of the waitlist.

- **Available**
  - **Time Complexity:** O(1)
  - **Explanation:**
    - Access lengths of `SeatHeap` and `MinHeap`.

- **Quit**
  - **Time Complexity:** O(1)
  - **Explanation:**
    - Program termination.

Data Structures Time Complexities

- **Red-Black Tree Operations (Reservations):**
  - **Insertion:** O(log N)
  - **Deletion:** O(log N)
  - **Search:** O(log N)
  - **In-order Traversal:** O(N)

- **Min-Heap Operations (Waitlist and SeatHeap):**
  - **Insertion (push):** O(log N)
  - **Deletion (pop/remove):** O(log N)
  - **Update Priority:** O(log N)
  - **Access Minimum Element:** O(1)

Note:

- `N` refers to the total number of seats or users, depending on context.
- `M` refers to the number of users affected in operations like `ReleaseSeats`.

- The complexities assume that the operations on dictionaries and sets (like `user_to_seat` and `user_in_waitlist`) are O(1).
- The time complexities are given in terms of Big O notation, representing the upper bound of the operation's running time.