Gator Ticket Master Project Report

COP 5536 Advanced Data Structures

- Name: Divyam Dubey
- UFID: 45227758
- UF Email: divyamdubey@ufl.edu

Project Overview

Gator Ticket Master streamlines seat reservations for Gator events, automating seat allocation, cancellations, and waitlist management. The system begins with a set number of seats, dynamically adding more if demand increases. Seats are assigned in ascending order, and users with the highest priority and earliest request timestamp are prioritized if on a waitlist. Canceled seats are reassigned from the waitlist or returned to available inventory if no waitlist exists. Event organizers can release seats within a user ID range if unusual activity is detected.

The system uses a Red-Black Tree to track active reservations, with each node storing User ID and Seat ID, while two Min-Heaps manage the waitlist and available seats for efficient reallocation. This ensures a responsive and fair reservation process for high-demand events.

Note:

We are not using a Makefile in this project because Python is an interpreted language, so there's no compilation step required. The code can be run directly using the Python interpreter. The project structure is relatively simple, with a main script (gatorTicketMaster.py) that imports the necessary modules. Python's import system handles dependencies automatically, eliminating the need for a build process that a Makefile would typically manage.

Project Structure

Main Components

1. gatorTicketMaster.py: Main program file
   - Handles user interactions and orchestrates overall functionality
   - Implements core functions like Initialize, Reserve, Cancel, etc.
2. BinaryHeap.py: Implementation of the Binary Heap data structure
   - Used for managing unassigned seats efficiently
3. RedBlackTree.py: Implementation of the Red-Black Tree data structure

   - Manages seat reservations and provides efficient search, insert, and delete operations

4. WaitlistMaxBinaryHeap.py: Implementation of the Max Binary Heap for the waitlist
   - Manages the waitlist with priority-based ordering

Key Data Structures

Binary Heap (BinaryHeap.py)

- Stores unassigned seat IDs
- Provides O(log n) insertion and extraction of minimum element
- Key methods:
  - insert(item)
  - extract_min()
  - remove_arbitrary(item)

Red-Black Tree (RedBlackTree.py)

- Stores user reservations (userId as key, seatId as value)
- Ensures O(log n) time complexity for search, insert, and delete operations
- Key methods:
  - insert(key, seatId)
  - delete(key)
  - search(key)
  - inorder_traversal()

Max Binary Heap (WaitlistMaxBinaryHeap.py)

- Manages the waitlist with user priorities
- Allows O(log n) insertion and extraction of maximum priority element
- Key methods:
  - insert(userId, priority, timestamp)
  - extract_max()
  - update_priority(userId, new_priority)
  - remove_user(userId)

Core Functions in gatorTicketMaster.py

1. Initialize(seatCount: int):
    a) Sets up the initial seating arrangement for the venue
    b) Creates a pool of available seats numbered from 1 to seatCount
2. Reserve(userId: int, userPriority: int):
    a) Attempts to reserve a seat for a user or adds them to the waitlist
    b) Assigns the lowest-numbered available seat if seats are available
    c) Adds the user to a priority-based waitlist if no seats are available
3. Cancel(seatId: int, userId: int):
    a) Cancels a user’s reservation for a specific seat
    b) Makes the seat available for the next reservation or waitlisted user
    c) Assigns the newly available seat to the highest-priority waitlisted user, if any
4. Available():
    a) Provides a status report of the ticketing system
    b) Displays the number of seats currently available
    c) Shows the number of users on the waitlist
5. ExitWaitlist(userId: int):
    a) Removes a specific user from the waitlist
    b) Useful when a user no longer wishes to wait for a seat
6. UpdatePriority(userId: int, userPriority: int):
    a) Changes the priority of a waitlisted user
    b) Adjusts the user’s position in the waitlist based on the new priority
7. AddSeats(count: int):
    a) Increases the total number of available seats in the system
    b) Assigns newly added seats to waitlisted users if any exist, based on their priority
8. PrintReservations():
    a) Displays a list of all current seat reservations
    b) Shows each reserved seat number along with the user ID who reserved it
    c) Lists reservations in order of seat numbers
9. ReleaseSeats(userId1: int, userId2: int):
    a) Cancels reservations for a range of user IDs (from userId1 to userId2, inclusive)
    b) Makes these seats available for new reservations or waitlisted users
    c) Assigns newly available seats to waitlisted users based on their priority
10. Quit():
    a) Performs any necessary cleanup operations
    b) Returns True to signal the main loop to end the program execution

Data Flow

1. Input Processing:
   - Read commands from an input file
   - Parse each command and call corresponding functions
2. Seat Management:
   - Unassigned seats stored in Binary Heap
   - Reserved seats managed in Red-Black Tree
3. Waitlist Management:
   - Waitlisted users stored in Max Binary Heap
   - Priority updates handled within the heap
4. Reservation Process:
   - Check Binary Heap for available seats
   - If seat available, remove from Binary Heap and add to Red-Black Tree
   - If no seat available, add user to Waitlist Max Binary Heap
5. Cancellation Process:
   - Remove reservation from Red-Black Tree
   - Check Waitlist Max Binary Heap for highest priority user
   - If waitlist not empty, assign seat to top waitlisted user
   - If waitlist empty, return seat to Binary Heap
6. Output Generation:
   - Write results of each operation to an output file

This structure ensures efficient seat management, waitlist handling, and reservation processing, with each component optimized for its specific role in the system.

Implementation Choices and Rationale

1. Data Structure Selection

We chose to implement three main data structures for this system:

a) Binary Heap (Min Heap) for Unassigned Seats:

- Advantage: Provides O(log n) time complexity for both insertion and extraction of the minimum element.
- Rationale: Allows us to efficiently assign the lowest-numbered available seat during reservations.
- Disadvantage: Doesn't support efficient arbitrary element deletion, which is mitigated by our usage pattern.

b) Red-Black Tree for Active Reservations:

- Advantage: Guarantees O(log n) time complexity for search, insertion, and deletion operations.
- Rationale: Efficiently manages active reservations, allowing quick lookups and updates.
- Disadvantage: More complex implementation compared to simpler data structures.

c) Max Binary Heap for Waitlist:

- Advantage: Provides O(log n) time complexity for insertion and extraction of the highest priority element.
- Rationale: Efficiently manages the waitlist, always having the highest priority user ready for assignment.
- Disadvantage: Updating priorities or removing arbitrary users is more complex, requiring additional bookkeeping.

2. Timestamp Usage in Waitlist

We incorporated timestamps in the waitlist to break ties between users with the same priority:

- Advantage: Ensures fairness by giving preference to earlier requests when priorities are equal.
- Rationale: Provides a deterministic way to handle equal priorities without relying on insertion order.
- Disadvantage: Requires additional storage and comparison logic.

3. User Position Tracking in Waitlist

We maintain a separate dictionary to track user positions in the waitlist:

- Advantage: Allows O(1) access to a user's position, facilitating efficient updates and removals.
- Rationale: Significantly improves the performance of operations like UpdatePriority and ExitWaitlist.

- Disadvantage: Requires additional memory and needs careful management to keep in sync with the heap.

4. Dynamic Seat Addition

The AddSeats function dynamically increases the seat capacity:

- Advantage: Allows flexibility in managing venue capacity.
- Rationale: Accommodates scenarios where additional seats become available after initial setup.
- Implementation Note: New seats are first assigned to waitlisted users (if any) before being added to unassigned seats.

5. Batch Seat Release

The ReleaseSeats function handles releasing multiple seats in a single operation:

- Advantage: Efficiently handles bulk cancellations or administrative seat releases.
- Rationale: Provides a way to quickly free up a range of seats, useful for handling no-shows or system errors.
- Implementation Note: Released seats are immediately offered to waitlisted users, maintaining system responsiveness.

6. Error Handling and Input Validation

We implemented input validation for functions like Initialize, AddSeats, and ReleaseSeats:

- Advantage: Improves system robustness by preventing invalid operations.
- Rationale: Ensures data integrity and prevents system state corruption.
- Implementation Note: Invalid inputs are caught and appropriate error messages are generated.

Time Complexity Analysis

- Initialize: O(n log n) where n is the number of seats
- Reserve: O(log n)
- Cancel: O(log n)
- Available: O(1)
- ExitWaitlist: O(log n)
- UpdatePriority: O(log n)
- AddSeats: O(m log n) where m is the number of new seats
- PrintReservations: O(n)
- ReleaseSeats: O((k + m) log n) where k is the range of user IDs and m is the number of waitlisted users assigned seats

Space Complexity

The space complexity of the system is O(n), where n is the total number of seats. This accounts for:

- Binary Heap for unassigned seats
- Red-Black Tree for reservations
- Max Binary Heap for the waitlist

Performance Considerations

- The use of efficient data structures (Binary Heap, Red-Black Tree) ensures optimal performance for large-scale events.
- The system is designed to handle dynamic seat additions and priority updates efficiently.

Error Handling and Edge Cases

- The system handles invalid inputs by providing appropriate error messages.
- Edge cases such as canceling non-existent reservations or updating priorities for users not in the waitlist are properly managed.