Gator Ticket Master
A Ticket Reservation System

Vishal Karthikeyan Setti
UFID: 47670880
Email: v.setti@ufl.edu

1 Project Overview

Gator Ticket Master is a seat booking service for Gator events, allowing users to reserve seats for attending these events. The service aims to develop a software system to efficiently manage seat allocation and reservation processes.

2 System Architecture

The code implementation is done using Red Black Trees and Min Heaps.

- **GatorTicketMaster**: Main class handling all ticket operations.
- **RedBlackTree**: The seat reservation system is primarily managed through a self-balancing Red-Black Tree. It stores the (userID, seatID) in its node.
- **MinHeap**: Priority queue implementation for waitlist management and available seats.

3 Data Structures

3.1 Red-Black Tree

Used for storing seat reservations with the following properties:

- Self-balancing binary search tree
- Guarantees O(log n) time complexity for operations
- Maintains user-seat mappings

3.2 Min Heap

Implements priority queue for waitlist management:

- Maintains users based on priority and timestamp
- Ensures highest priority users are served first
- O(log n) time complexity for insertions and deletions

4 Core Function Prototypes

4.1 GatorTicketMaster Class

The GatorTicketMaster class is a Java implementation of a ticket reservation system that efficiently manages seat bookings and user waitlists. At its core, it uses three main data structures: a Red-Black Tree for managing seat reservations, and two Min Heaps for handling available seats and the waitlist queue.

The system processes various commands through a file-based interface, supporting operations like seat initialization, reservations, cancellations, and waitlist management. When users attempt to reserve seats, they're either assigned an available seat or placed on a priority-based waitlist, where their position is determined by both their priority level and request timestamp. The system maintains data integrity through careful validation and provides feedback through an output file for each operation performed.

```
1 public void initialize(int seatCount)
```

The initialize method sets up a ticket reservation system by validating and creating a specified number of seats. It stores the total seat count and populates the availableSeats heap with sequential seat numbers (1 to seatCount), providing a foundation for the reservation system. The method includes basic error handling for invalid inputs and confirms successful initialization with a status message.

```
1 public void available()
```

The available method provides a simple status report of the ticketing system by printing the current count of available seats and the number of people in the waitlist. It directly accesses the size of both the availableSeats and waitlist data structures to display this information in a formatted output.

```
1 public void reserve(int userID, int userPriority)
```

The reserve method implements a dual-path ticket reservation system. If seats are available, it extracts the lowest-numbered seat from the availableSeats heap and assigns it to the user, recording this assignment in seatReservations. If no seats are available, it adds the user to a priority-based waitlist with their specified priority level. The method handles both direct reservations and waitlist management through a single interface, providing appropriate feedback through the output stream for both successful reservations and waitlist additions. This implementation ensures efficient seat allocation while maintaining a prioritized queue for handling overflow demand.

```
public void cancel(int seatID, int userID)
```

The cancel method handles ticket cancellation by first validating the user's reservation. If valid, it removes the reservation and either reassigns the seat to the highest-priority user from the waitlist or returns it to the available seats pool. The method includes error handling for invalid cancellation attempts and provides appropriate status messages for each outcome.

```
public void exitWaitlist(int userID)
```

The exitWaitlist method removes a user from the waitlist if present. It uses the waitlist's remove operation to find and remove the user by their ID, then prints a confirmation message for successful removal or notifies if the user wasn't found in the waitlist. The implementation provides simple, direct feedback for both successful and failed removal attempts.

```
public void updatePriority(int userID, int newPriority)
```

The updatePriority method modifies a user's priority in the waitlist system. It attempts to update the priority value for the specified userID and provides appropriate feedback through the output stream. If the user is found in the waitlist, their priority is updated and a confirmation message is displayed; if not found, it prints a failure notification. The method leverages the waitlist's internal updatePriority implementation to maintain proper heap ordering after the priority change.

```
public void addSeats(int count)
```

The addSeats method increases the venue's capacity by adding new seats. After validating the input, it processes each new seat by either assigning it to the highest-priority user in the waitlist or adding it to the available seats pool. The method maintains system integrity while providing feedback about seat assignments and capacity changes.

```
public void printReservations()
```

The printReservations method displays all current seat reservations in ascending seat number order. It retrieves reservations from the Red-Black Tree, sorts them by seatID, and prints each seat-user pairing in a formatted output.

```
public void releaseSeats(int userID1, int userID2)
```

The releaseSeats method handles batch cancellation of reservations for a range of user IDs. After validating the input range, it processes each user in the range by removing their seat reservations and waitlist entries. The released seats are then either reassigned to users from the waitlist (in priority order) or added back to the available seats pool if no waitlist exists. The method maintains system integrity by properly updating all data structures (seatReservations, waitlist, and availableSeats) while providing appropriate feedback messages for the batch operation's results.

```
public void quit()
```

Terminates the program and closes the output writer. This method ensures that all resources are properly released and output file is closed before the program exits.

```
public static void main(String[] args)
```

The main method drives the GatorTicketMaster system by processing commands from an input file and writing results to an output file. It parses each command line and routes it to the appropriate method through a switch statement, handling ten different operations ranging from initialization to seat management. The method includes basic error handling for file operations and ensures proper resource cleanup.

4.2 MinHeap Class

The MinHeap class implements a priority queue data structure with additional functionality for managing user waitlist entries. It maintains two main data structures: an ArrayList for the heap elements and a HashMap (userIndexMap) for quick access to user entries by their IDs. The class supports standard heap operations like insert, extractMin, and remove, while also providing specialized methods for handling WaitlistEntry objects with priority updates.

The implementation uses a compare method that can handle both Integer and WaitlistEntry objects, maintaining heap ordering based on either simple numerical comparison or a combination of priority and timestamp for waitlist entries. The heap property is maintained through promoteElement (bubble-up) and demoteElement (sink-down) operations, ensuring that parent nodes always have higher priority than their children.

```
public void insert(Object obj)
```

The insert method implements a heap insertion operation with an additional mapping feature for waitlist entries. It first adds the new object to the end of the heap array, then if the object is a WaitlistEntry, it maintains a separate mapping of userIDs to their heap indices for quick access.

Finally, it calls promoteElement to perform the standard heap "bubble-up" operation, ensuring the heap property is maintained by comparing and potentially swapping the new element with its parent nodes until it reaches its correct position.

```
private void promoteElement(int current)
```

The promoteElement method performs the classic heap "bubble-up" operation to maintain the min-heap property after insertion. Starting from the given index, it repeatedly compares the element with its parent and swaps them if the parent is larger, continuing this process until either reaching the root or finding a parent that's smaller than the current element. For WaitlistEntry objects, the comparison would be based on their priority values. The method maintains the fundamental heap property where each parent node must be smaller than or equal to its children, ensuring O(log n) complexity as the maximum number of swaps is limited by the heap's height.

```
public Object extractMin()
```

The extractMin method implements the standard heap extraction operation with additional handling for WaitlistEntry objects. It first retrieves the minimum element from the heap's root, updates the userIndexMap if it's a WaitlistEntry, then replaces the root with the last element in the heap. After removing the last element, it calls demoteElement to restore the heap property by "sinking down" the new root to its correct position. The method maintains a mapping of userIDs to their positions in the heap through userIndexMap, ensuring O(1) access to specific entries while preserving the overall O(log n) complexity of heap extraction. This implementation combines efficient priority queue operations with custom functionality for tracking waitlist entries.

```
private void demoteElement(int index)
```

The demoteElement method implements the heap's "sink-down" operation to maintain the min-heap property after root extraction. It starts at the given index and repeatedly compares the element with its children (left child at 2*index + 1 and right child at 2*index + 2), finding the smallest among the three nodes. If either child is smaller than the current node, it swaps with the smallest child and continues the process at the new position. This

process continues until either reaching a leaf node or finding a position where both children are larger than the current node. The method ensures the heap property is maintained where each parent must be smaller than its children.

```
private int compare(Object a, Object b)
```

The compare method provides a custom comparison logic for heap elements, handling two specific types: Integer and WaitlistEntry objects. For Integers, it uses the standard numerical comparison. For WaitlistEntry objects, it implements a two-tier comparison system: first comparing by priority (higher priority takes precedence) and then by timestamp (earlier timestamp wins) when priorities are equal. If the objects being compared are of different or unsupported types, it throws an IllegalArgumentException.

This comparison method is essential for maintaining proper ordering in the priority queue, ensuring that both simple integer priorities and complex waitlist entries with multiple comparison criteria are handled correctly.

```
private void swap(int i, int j)
```

The swap method performs a basic element exchange operation in the heap while maintaining the integrity of the userIndexMap. It first swaps two elements at the given indices in the heap array using a temporary variable. Then, for any WaitlistEntry objects involved in the swap, it updates their corresponding entries in the userIndexMap to reflect their new positions.

```
public boolean remove(int userID)
```

The remove method implements an efficient deletion operation in a heap data structure with an accompanying userID mapping system. It first checks if the userID exists in the mapping, then performs the standard heap deletion process: swapping the target element with the last element, removing the last element, and then restoring the heap property. The method handles both upward and downward heap adjustments by checking whether the replacement element needs to be promoted (bubbled up) or demoted (sunk down) based on its relationship with its parent.

```
public boolean updatePriority(int userID, int newPriority)
```

The updatePriority method modifies a WaitlistEntry's priority while maintaining the heap's ordering properties. It first locates the entry using the userIndexMap for O(1) access, then updates its priority value. Depending on whether the new priority is higher or lower than the old one, it either promotes (bubbles up) or demotes (sinks down) the element to restore the heap property.

This implementation ensures efficient priority updates without requiring removal and reinsertion of the element. The method returns false if the specified userID is not found in the heap, providing proper error handling for invalid updates.

```
public boolean isEmpty()
```

Checks if the heap is empty and returns true if the heap is empty, otherwise false.

```
public int size()
```

Returns the number of elements currently in the heap and returns the size of the heap.

4.3 RedBlackTree Class

The RedBlackTree class is a specialized implementation of a balanced binary search tree that automatically maintains its equilibrium during modifications. This data structure organizes nodes containing paired userID and seatID values, ensuring efficient search, insertion, and deletion operations through its self-balancing properties. The tree follows the Red-Black Tree principles, which include specific coloring rules and automatic rebalancing mechanisms to maintain optimal performance.

Each node in the tree stores two key pieces of information, a userID and a seatID, which are used to maintain proper ordering within the tree structure.

```
private void rotateLeft(Node node)
```

Performs a left rotation around node given in the argument of the function. The rotateLeft method performs a left rotation on a node, rebalancing the tree by making the node's right child its new parent, which helps maintain optimal tree structure and efficiency in search operations.

```
private void rotateRight(Node node)
```

Performs a right rotation around node given in the argument of the function. The rotateRight method perform a right rotation on a node, rebalancing the tree by making the node's left child its new parent, which helps maintain an efficient and balanced tree structure for optimized search operations.

```
public void insert(int userID, int seatID)
```

The insert method adds a new node with a given userID and seatID into the binary search tree, placing it in the correct position to maintain the tree's order, and then rebalancing the tree if necessary to preserve optimal structure.

It identifies the correct position based on userID, inserts the node while maintaining binary search order, and then calls rebalanceTreeAfterInsert to ensure the tree remains balanced.

```
private void rebalanceTreeAfterInsert(Node node)
```

The rebalanceTreeAfterInsert method rebalances the tree after a new node is inserted to maintain Red-Black Tree properties. It adjusts colors and performs rotations as needed. If the node's parent is red, it checks the color of the parent's sibling. If the sibling is red, it recolors the parent, sibling, and grandparent and moves up the tree. If the sibling is black or null, it rotates the tree to balance it. Finally, it ensures the root node remains black. This ensures the tree remains balanced and maintains Red-Black properties: no consecutive red nodes and equal black heights.

```
public Node findNode(int userID)
```

The findNode method searches for and returns the node with a specified userID in the binary search tree, or null if no such node exists. It traverses left or right based on comparisons with userID.

```
public void deleteNode(int userID)
```

The provided code implements a deletion operation in a Red-Black Tree, a self-balancing binary search tree data structure. The deleteNode method handles the removal of a node with a specified userID while maintaining the tree's critical properties. The algorithm first locates the target node and then handles three distinct cases: nodes with at most one child, nodes with two children (using a successor node), and special handling for the root node.

```
private Node successorNode(Node node)
```

The successorNode method implements the in-order successor finding algorithm in a binary search tree, which is crucial for maintaining tree ordering during operations like deletion. This method finds the next node in the tree's in-order traversal sequence by following two possible paths: either descending to the leftmost node in the right subtree (if a right child exists), or ascending through parent nodes until finding the first ancestor where the current node is in its left subtree.

The first case is straightforward - when a right child exists, the successor is found by moving

right once and then following left children as far as possible. The second case handles nodes without right children by traversing up the tree through parent pointers until finding a node that is a left child of its parent, with that parent being the successor. If no such parent is found (meaning we've reached the rightmost node in the tree), the method returns null.

```
private void rebalanceTreeAfterDelete(Node node, Node parent)
```

The rebalanceTreeAfterDelete method restores the Red-Black Tree's properties after node deletion by handling color violations and structural imbalances. It processes both left and right child cases symmetrically, dealing with three main scenarios: red siblings (requiring color swaps and rotations), black siblings with black children (requiring recoloring), and black siblings with at least one red child (requiring rotations and color adjustments).

The method uses a combination of tree rotations and color modifications to ensure all Red-Black Tree properties are maintained: black root, red nodes having black children, and consistent black-height across all paths. This rebalancing process is essential for maintaining the tree's logarithmic height guarantee and operational efficiency.

```
public List<Node> inorderTraversal()
```

Performs an inorder traversal of the Red-Black Tree. This method return a list of nodes in the tree sorted by their in-order sequence.

```
private void inorderTraversalHelper(Node node, List<Node> result)
```

This is a helper method for performing an inorder traversal of the Red Black Tree. This method recursively traverses the tree and adds each node to the result list.

5 WaitlistEntry Class

```
static class WaitlistEntry implements Comparable<WaitlistEntry> {
    // Fields
    private int userID;
    private int priority;
    private long timestamp;

    // Constructor
    public WaitlistEntry(int userID, int priority);
}
```

The WaitlistEntry class represents a user in the waiting list system, storing their userID, priority level, and entry timestamp. It implements Comparable to enable priority-based

ordering, where entries are compared first by priority (higher values take precedence) and then by timestamp (earlier entries win ties). The timestamp is automatically set using System.nanoTime() when an entry is created, ensuring precise chronological ordering when priorities are equal.

6 Node Class in RedBlackTree

```
static class Node {
    public Node(int userID, int seatID)
}
```

The Node class defines the structure for nodes in a Red-Black Tree, used for managing seat reservations. Each node contains a userID and seatID pair, along with color information (RED/BLACK) and pointers to its left child, right child, and parent nodes. The class initializes new nodes as RED by default, following Red-Black Tree insertion conventions. This structure enables efficient storage and retrieval of seat reservation information while maintaining the tree's self-balancing properties.

7 Makefile Targets and Usage

7.1 Default Target

The default target (invoked by running `make` without arguments) performs three sequential operations:

- Executes the clean target and all generated .class and output files
- Compiles all source files
- Processes given input files automatically

7.2 Target Specifications

7.2.1 make all

```
all: clean compile $(OUTPUT_FILE)
```

This target provides a complete build process by executing clean and compile sequentially.

7.2.2 make compile

```
compile:
    $(JAVAC) *.java
```

Handles the compilation of Java source files into class files.

7.2.3 Pattern rule

```
%_output_file.txt: %.txt
    $(JAVA) $(MAIN_CLASS) $<
```

Pattern rule to generate output files from input files.

7.2.4 make clean

```
clean:
ifeq ($(OS),Windows_NT)
# Windows environment command to delete files
  del /F /Q *.class 2>nul || true
  del /F /Q *output_file.txt 2>nul || true
else
# Unix environment command to delete files
  rm -f *.class *output_file.txt
endif
```

Removes all generated files to ensure a clean build environment. This clean command is written so that it can run in both Windows and Linux based environments.

7.2.5 make run

```
run:
    $(JAVA) $(MAIN_CLASS) $(INPUT_FILE)
```

Executes the program with a specified input file. Usage:

`make run INPUT_FILE=<filename>`

7.2.6 make help

Displays available commands and their descriptions:

- Lists all available targets
- Provides brief descriptions
- Shows usage instructions

8 Design Choice

- **Red-Black Tree for Seat Reservations**
  - O(log n) performance for insertions, deletions, lookups
  - Maintains ordered seat information
- **Min Heap + HashMap for Waitlist**
  - Priority-based queueing

  - O(1) user lookups
  - Uses timestamps for tie-breaking
- **Min Heap for Available Seats**
  - Tracks and allocates unassigned seats efficiently

Key Features

- Command Pattern architecture for maintainability
- O(log n) complexity for most operations
- Automatic seat reallocation from waitlist
- Comprehensive error handling

9 Time Complexity Analysis

9.1 GatorTicketMaster

The Table 1 gives the time complexity for the class GatorTicketMaster.

Function	**Time Complexity**	**Explanation**
`initialize`	$O(n)$	Creates a min heap with n seats. Each insertion takes $O(\log n)$, but building the entire heap takes $O(n)$
`available`	$O(1)$	Returns the size of two data structures which is maintained as a variable
`reserve`	$O(\log n)$	Extracts minimum from availableSeats heap ($O(\log n)$) or inserts into waitlist heap ($O(\log n)$), plus RB tree insertion ($O(\log n)$)
`cancel`	$O(\log n)$	Finding node in RB tree ($O(\log n)$), deletion ($O(\log n)$), and heap operation ($O(\log n)$)
`exitWaitlist`	$O(n)$	Searching through heap to find user ($O(n)$) and removal ($O(\log n)$)
`updatePriority`	$O(n)$	Searching through heap ($O(n)$) and update operation ($O(\log n)$)
`addSeats`	$O(k \log n)$	For k new seats, each requires heap extraction or insertion ($O(\log n)$)
`printReservations`	$O(n \log n)$	Inorder traversal of RB tree ($O(n)$) plus sorting ($O(n \log n)$)
`releaseSeats`	$O(m \log n)$	For range m (userID2 - userID1), each user requires RB tree and heap operations ($O(\log n)$)
`quit`	$O(1)$	Closes output stream

Table 1: Time Complexity Analysis of GatorTicketMaster Functions

9.2 MinHeap

The Table 2 gives the time complexity for the class MinHeap.

Function	**Complexity**	**Explanation**
`insert`	$O(\log n)$	Adds element to end of heap ($O(1)$) and bubbles up through height of tree ($O(\log n)$). HashMap operations are $O(1)$
`promoteElement`	$O(\log n)$	Traverses up the heap tree, maximum path length is height of tree
`extractMin`	$O(\log n)$	Removes root, moves last element to root ($O(1)$), then demotes it ($O(\log n)$). HashMap operations are $O(1)$
`demoteElement`	$O(\log n)$	Traverses down the heap tree, maximum path length is height of tree
`compare`	$O(1)$	Simple comparison operations for integers and WaitlistEntry objects
`swap`	$O(1)$	Constant time array element swaps and HashMap updates
`remove`	$O(\log n)$	HashMap lookup ($O(1)$), swap with last element ($O(1)$), then either promote or demote ($O(\log n)$)
`updatePriority`	$O(\log n)$	HashMap lookup ($O(1)$), then either promote or demote operation ($O(\log n)$)
`isEmpty`	$O(1)$	Simple check of ArrayList size
`size`	$O(1)$	Returns ArrayList size

Table 2: Time Complexity Analysis of MinHeap Operations

9.3 RedBlackTree

The Table 3 gives the time complexity for the class RedBlackTree.

Function	**Time Complexity**	**Notes**
`rotateLeft()`	$O(1)$	Constant time operation, only updates pointers
`rotateRight()`	$O(1)$	Constant time operation, only updates pointers
`insert()`	$O(\log n)$	Tree traversal to find insertion point plus rebalancing
`rebalanceTreeAfterInsert()`	$O(\log n)$	Maximum height of tree traversal for recoloring and rotations
`findNode()`	$O(\log n)$	Binary search through balanced tree
`deleteNode()`	$O(\log n)$	Finding node plus rebalancing operations
`successorNode()`	$O(\log n)$	May need to traverse up to the height of the tree
`rebalanceTreeAfterDelete()`	$O(\log n)$	Maximum height of tree traversal for recoloring and rotations
`inorderTraversal()`	$O(n)$	Must visit every node in the tree
`inorderTraversalHelper()`	$O(n)$	Recursive function visiting all nodes

Table 3: Time Complexity Analysis of Red-Black Tree Operations

10 Output

Test Case 1

```
Initialize(4)
Available()
Reserve(8, 1)
Reserve(4, 3)
Reserve(5, 2)
Cancel(2, 4)
Reserve(1, 1)
Available()
PrintReservations()
Reserve(3, 1)
Reserve(2, 2)
Reserve(6, 3)
Available()
Cancel(1, 1)
Cancel(3, 5)
Available()
Reserve(11, 1)
Reserve(9, 2)
AddSeats(2)
Reserve(7, 2)
Cancel(1, 8)
Available()
ReleaseSeats(8, 10)
PrintReservations()
Quit()
```

Output for Test Case 1

```
4 Seats are made available for reservation
Total Seats Available : 4, Waitlist : 0
User 8 reserved seat 1
User 4 reserved seat 2
User 5 reserved seat 3
User 4 canceled their reservation
User 1 reserved seat 2
Total Seats Available : 1, Waitlist : 0
Seat 1, User 8
Seat 2, User 1
Seat 3, User 5
User 3 reserved seat 4
User 2 is added to the waiting list
User 6 is added to the waiting list
Total Seats Available : 0, Waitlist : 2
User 1 has no reservation for seat 1 to cancel
User 5 canceled their reservation
User 6 reserved seat 3
Total Seats Available : 0, Waitlist : 1
User 11 is added to the waiting list
```

```
User 9 is added to the waiting list
Additional 2 Seats are made available for reservation
User 2 reserved seat 5
User 9 reserved seat 6
User 7 is added to the waiting list
User 8 canceled their reservation
User 7 reserved seat 1
Total Seats Available : 0, Waitlist : 1
Reservations of the Users in the range [8, 10] are released
User 11 reserved seat 6
Seat 1, User 7
Seat 2, User 1
Seat 3, User 6
Seat 4, User 3
Seat 5, User 2
Seat 6, User 11
Program Terminated!!
```

Test Case 2

```
Initialize(10)
Reserve(2, 1)
Reserve(6, 2)
Reserve(13, 1)
Reserve(15, 3)
Reserve(9, 4)
Cancel(3, 13)
Reserve(4, 2)
Reserve(7, 1)
Reserve(8, 2)
Reserve(66, 1)
ReleaseSeats(10, 100)
Reserve(52, 1)
Reserve(41, 2)
Reserve(13, 1)
Reserve(1, 3)
Reserve(14, 1)
Reserve(15, 1)
Reserve(121, 3)
Reserve(161, 2)
Reserve(180, 4)
UpdatePriority(15, 2)
Reserve(72, 2)
Reserve(81, 3)
Reserve(66, 2)
Reserve(91, 1)
Reserve(99, 1)
Reserve(86, 2)
AddSeats(10)
Available()
PrintReservations()
Quit()
```

Output for Test Case 2

```
10 Seats are made available for reservation
User 2 reserved seat 1
User 6 reserved seat 2
User 13 reserved seat 3
User 15 reserved seat 4
User 9 reserved seat 5
User 13 canceled their reservation
User 4 reserved seat 3
User 7 reserved seat 6
User 8 reserved seat 7
User 66 reserved seat 8
Reservations/waitlist of the users in the range [10, 100] have been
   released
User 52 reserved seat 4
User 41 reserved seat 8
User 13 reserved seat 9
User 1 reserved seat 10
User 14 is added to the waiting list
User 15 is added to the waiting list
User 121 is added to the waiting list
User 161 is added to the waiting list
User 180 is added to the waiting list
User 15 priority has been updated to 2
User 72 is added to the waiting list
User 81 is added to the waiting list
User 66 is added to the waiting list
User 91 is added to the waiting list
User 99 is added to the waiting list
User 86 is added to the waiting list
Additional 10 Seats are made available for reservation
User 180 reserved seat 11
User 121 reserved seat 12
User 81 reserved seat 13
User 15 reserved seat 14
User 161 reserved seat 15
User 72 reserved seat 16
User 66 reserved seat 17
User 86 reserved seat 18
User 14 reserved seat 19
User 91 reserved seat 20
Total Seats Available : 0, Waitlist : 1
Seat 1, User 2
Seat 2, User 6
Seat 3, User 4
Seat 4, User 52
Seat 5, User 9
Seat 6, User 7
Seat 7, User 8
Seat 8, User 41
Seat 9, User 13
Seat 10, User 1
Seat 11, User 180
```

```
Seat 12, User 121
Seat 13, User 81
Seat 14, User 15
Seat 15, User 161
Seat 16, User 72
Seat 17, User 66
Seat 18, User 86
Seat 19, User 14
Seat 20, User 91
Program Terminated!!
```

Test Case 3

```
Initialize(3)
Reserve(6, 2)
Reserve(1, 1)
Reserve(9, 3)
Reserve(2, 2)
Reserve(5, 1)
Reserve(4, 3)
Reserve(3, 2)
Available()
ExitWaitlist(9)
UpdatePriority(6, 3)
UpdatePriority(3, 3)
AddSeats(3)
Available()
Cancel(3, 9)
Reserve(9, 3)
Reserve(12, 2)
Reserve(18, 1)
Reserve(17, 3)
ReleaseSeats(6, 11)
UpdatePriority(18, 3)
AddSeats(2)
PrintReservations()
Quit()
Release(1, 12)
PrintReservations()
```

Output for Test Case 3

```
3 Seats are made available for reservation
User 6 reserved seat 1
User 1 reserved seat 2
User 9 reserved seat 3
User 2 is added to the waiting list
User 5 is added to the waiting list
User 4 is added to the waiting list
User 3 is added to the waiting list
Total Seats Available : 0, Waitlist : 4
```

```
User 9 is not in waitlist
User 6 priority is not updated
User 3 priority has been updated to 3
Additional 3 Seats are made available for reservation
User 4 reserved seat 4
User 3 reserved seat 5
User 2 reserved seat 6
Total Seats Available : 0, Waitlist : 1
User 9 canceled their reservation
User 5 reserved seat 3
User 9 is added to the waiting list
User 12 is added to the waiting list
User 18 is added to the waiting list
User 17 is added to the waiting list
Reservations of the Users in the range [6, 11] are released
User 17 reserved seat 1
User 18 priority has been updated to 3
Additional 2 Seats are made available for reservation
User 18 reserved seat 7
User 12 reserved seat 8
Seat 1, User 17
Seat 2, User 1
Seat 3, User 5
Seat 4, User 4
Seat 5, User 3
Seat 6, User 2
Seat 7, User 18
Seat 8, User 12
Program Terminated!!
```

11 Conclusion

The GatorTicketMaster system provides an efficient and robust solution for ticket management, utilizing optimal data structures and algorithms to ensure performance and reliability.