COP 5536 Fall 2024

Programming Project
Due – November 14th

Gator Ticket Master

Problem Description

Gator Ticket Master is a seat booking service for Gator Events. Users utilize this service to secure a seat for attendance at a gator event. The service is seeking to develop a software system that effectively manages seat allocation and reservation management.

The service must adhere the following specific requirements for managing reservations:

- Every event commences with a specific **initial number of seats**, which can be **increased** in the event of a high anticipated demand for the event.
- When a user attempts to reserve a seat for the event, the system **prioritizes** and assigns the seat with the **lowest number** first from the available seats.
- Users have the option to **withdraw** their reservation, in which case the seat is reassigned to the user selected from the waitlist. If the waitlist is empty, the reservation is deleted, and the seat number is reintroduced into the list of unassigned seats.
- Users are assigned **priorities** (integer values), and the user with the highest priority is given preference when assigning seats from the waitlist heap(e.g., User with priority 2 gets priority over user with priority 1). Ties are resolved by considering the timestamp at which the reservation was made (first-come, first-served basis).
- It is assumed that users will not attempt to reserve a seat twice.
- User **priority can be modified** once they enter the waitlist. We then update the waitlist with the new priority while preserving the original timestamp data.
- If the event organizer finds out about any unusual booking activity from a group of users, they can **release the seats assigned to a range of user ID**'s and make those seats available again. If any of the users in the range are in waiting list, we remove the users from it.

Implement from scratch, a **Red-Black tree** to manage the reserved seat information. Each node in that tree contains the following information: *User ID* (unique identifier of the user and is the *Key* for the Red-Black Tree) and *Seat ID* (unique identifier of the seat). Implement from scratch, a priority-queue mechanism using a **Binary Min-Heap** as a data structure to manage the waitlist in the event that there are no seats currently unassigned. You should also implement a heap to keep track of the seat numbers that are currently unassigned, to satisfy the 2nd requirement. Since, there can be a scenarios where a user cancels his reservation and there is not waitlist, so the seat gets added back to the available seats

The system should support the following operations:

1. **Initialize(seatCount)** : Initialize the events with the specified number of seats, denoted as "seatCount". The seat numbers will be sequentially assigned as [1, 2, 3, …, seatCount] and added to the list of unassigned seats .
2. **Available() :** Print the number of seats that are currently available for reservation and the length of the waitlist.
3. **Reserve(userID, userPriority):** Allow a user to reserve the seat that is available from the unassigned seat list and update the reserved seats tree. If no seats are currently available, create a new entry in the waitlist heap as per the user's priority and timestamp. Print out the seat number if a seat is assigned. If the user is added to the waitlist Print out a message to the user stating that he is added to the waitlist.
4. **Cancel(seatID, userID):** Reassign the seat to user from the waitlist heap. If the waitlist is empty, delete the node and add it back to the available seats.
5. **ExitWaitlist(userID):** If the user is in the waiting list, remove him from the waiting list. If the user is already assigned a seat prior to this, the user must use the cancel function to cancel his reservation instead.
6. **UpdatePriority(userID, userPriority):** Modify the user priority only if he is in the waitlist heap. Update the heap with this modification.
7. **AddSeats(count):** We add the new seat numbers to the available seat list. The new seat numbers should follow the previously available range.
8. **PrintReservations():** We print the information of all the assigned seats and the users they are assigned to ordered by the seat Numbers.
9. **ReleaseSeats(userID1, userID2):** We release all the seats assigned to the users whose ID falls in the range [userID1, userID2]. It is guaranteed that userID2 >= userID1. We even remove the users from the waitlist if they are present there. The status of the change should be printed ordered by userID's in the range.
10. **Quit():** Anything below this command in the input file will not be processed. The program terminates either when the quit command is read by the system or when it reaches the end of the input commands, which ever happens first.

Please refer to the Function Output Format section for more details on how to handle output when the functions are invoked.

Programming Environment

You may use either Java, C++, or Python for this project. Your program will be tested using the Java or g++ compiler or python interpreter on the thunder.cise.ufl.edu server. So, you should verify that it compiles and runs as expected on this server, which may be accessed via the Internet. Your submission must include a makefile that creates an executable file named **gatorTicketMaster**.

Your program should execute using the following :

For **C/C++:**
```
$ ./ gatorTicketMaster file_name
```

For **Java:**
```
$ java gatorTicketMaster file_name
```

For **Python:**
$ python3 gatorTicketMaster file_name

Where file_name is the name of the file that has the input test data.

Input and Output Requirements:

- Read Input from a text file where **input_filename** is specified as a command-line argument.
- All Output should be written to a text file having filename as concatenation of **input_filename + "_" + "output_file.txt"**
  (e.g., input_filename = "test1.txt", output_filename = "test1_output_file.txt")
- The program should terminate when the operation encountered in the input file is **Quit()**

Function Output Format:

Initialize(seatCount):

```
# If a valid integer is provided

<seatCount> Seats are made available for reservation

# If invalid integer is provided

Invalid input. Please provide a valid number of seats.
```

Available():

```
Total Seats Available : <available seat count>, Waitlist : <waitlist
length>
```

Reserve(userID, userPriority):

```
# If seat is available for reservation

User <userID> reserved seat <seatID>

# If the user is added to the waiting list

User <userID> is added to the waiting list
```

Cancel(seatID, userID):

```
# If the user had a booking prior and the waitlist is empty

User <userID> canceled their reservation

# If the user had a booking prior and the waitlist is not empty
```

```
User <userID> canceled their reservation
User <userIDx> reserved seat <seatIDx>

# If the user has no reservation

User <userID> has no reservation to cancel

# The seat does not belong to the user

User <userID> has no reservation for seat <seatID> to cancel
```

ExitWaitlist(userID):

```
# If the user is in the waiting list

User <userID> is removed from the waiting list

# If the user is not in the waiting list

User <userID> is not in waitlist
```

UpdatePriority(userID, userPriority):

```
# If the user is in the waiting list
User <userID> priority has been updated to <userPriority>

# If the user is not in the waiting list

User <userID> priority is not updated
```

AddSeats(count):

```
# If a valid integer is provided and the waitlist is empty

Additional <count> Seats are made available for reservation

# If a valid integer is provided and the waitlist is not empty

Additional <count> Seats are made available for reservation
User <userID1> reserved seat <seatID1>
User <userID2> reserved seat <seatID2>
.
.
.

# If invalid integer is provided

Invalid input. Please provide a valid number of seats.
```

PrintReservations():

```
[<seatID1>, <userID1>]
[<seatID2>, <userID2>]
[<seatID3>, <userID3>]
.
.
.
```

ReleaseSeats(userID1, userID2):

```
# If a valid range is provided and the waitlist is empty
Reservations/waitlist of the users in the range [userID1, userID2] have
been released

# If a valid range is provided and the waitlist is not empty
Reservations of the Users in the range [userID1, userID2] are released
User <userIDa> reserved seat <seatIDa>
User <userIDb> reserved seat <seatIDb>
.
.
.

# If invalid range is provided

Invalid input. Please provide a valid range of users.
```

Quit():

```
Program Terminated!!
```

Example 1:

Input

```
Initialize(5)
Available()
Reserve(1, 1)
Reserve(2, 1)
Cancel(1, 1)
Reserve(3, 1)
PrintReservations()
Quit()
```

Output

```
5 Seats are made available for reservation
Total Seats Available : 5, Waitlist : 0
User 1 reserved seat 1
```

```
User 2 reserved seat 2
User 1 canceled their reservation
User 3 reserved seat 1
Seat 1, User 3
Seat 2, User 2
Program Terminated!!
```

Example 2:

Input

```
Initialize(3)
Reserve(1, 1)
Reserve(2, 1)
Reserve(3, 1)
Reserve(4, 1)
Available()
Reserve(5, 2)
Available()
PrintReservations()
Cancel(3, 3)
Available()
PrintReservations()
AddSeats(2)
Reserve(6, 1)
Reserve(7, 1)
Reserve(8, 3)
Reserve(9, 1)
UpdatePriority(7,  2)
AddSeats(2)
ExitWaitlist(6)
Available()
Reserve(10,1)
Reserve(11,1)
ReleaseSeats(1,4)
Quit()
AddSeats(111)
```

Output

```
3 Seats are made available for reservation
User 1 reserved seat 1
User 2 reserved seat 2
User 3 reserved seat 3
User 4 is added to the waiting list
Total Seats Available : 0, Waitlist : 1
User 5 is added to the waiting list
Total Seats Available : 0, Waitlist : 2
[seat 1, user 1]
[seat 2, user 2]
[seat 3, user 3]
User 3 canceled their reservation
```

```
User 5 reserved seat 3
Total Seats Available : 0, Waitlist : 1
[seat 1, user 1]
[seat 2, user 2]
[seat 3, user 5]
Additional 2 Seats are made available for reservation
User 4 reserved seat 4
User 6 reserved seat 5
User 7 is added to the waiting list
User 8 is added to the waiting list
User 9 is added to the waiting list
User 7 priority has been updated to 2
Additional 2 Seats are made available for reservation
User 8 reserved seat 6
User 7 reserved seat 7
User 6 is not in waitlist
Total Seats Available : 0, Waitlist : 1
User 10 is added to the waiting list
User 11 is added to the waiting list
Reservations of the Users in the range [1, 4] are released
User 9 reserved seat 1
User 10 reserved seat 2
User 11 reserved seat 4
Program Terminated!!
```

Submission Requirements:

- Include a makefile for easy compilation.
- Provide well-commented source code.
- Submit a PDF report that includes project details, function prototypes, and explanations.
- Follow the input/output and submission requirements as described in the reference project.

Do not use nested directories. All your files must be in the first directory that appears after unzipping. You must submit the following:
1. Makefile: You must design your makefile such that 'make' command compiles the source code and produces an executable file. (For java class files that can be run with java command)
2. Source Program: Provide comments.
3. Report:
● The report should be in PDF format.
● The report should contain your basic info: **Name, UFID and UF Email**
● Present function prototypes showing the structure of your programs. Include the structure of program.

To submit, please compress all your files together using a zip utility and submit to the Canvas system. You should look for the "Assignment Project" for the submission. Your submission should be named "**LastName_FirstName.zip**".

Important: *Please make sure the name you provided is the same as the same that appears on the Canvas system.* ***Please do not submit directly to a TA.*** *All email submissions will be ignored without further notification. Please note that the due date is a hard deadline. No late submission will be allowed. Any submission after the deadline will not be accepted.*

Grading Policy

Grading will be based on the correctness and efficiency of algorithms. Below are some details of the grading policy.

- Correct implementation and execution: 25 pts
- Comments and readability (Source code and naming conventions: 15 pts
- Report: 20 pts
- Testcases: 40 pts

Important: *Your program will be graded based on the produced output. You must make sure to produce the correct output to get points. There will be a threshold for the running time of your program. If your program runs slow, we will assume that you have not implemented the required data structures properly.*
You will get ***negative points*** *if you do not follow the* ***input/output or submission requirements*** *above.*

Following is the clear guidance on how your marks will be deducted:

- Source files are not in a single directory after unzipping: -5 points
- Incorrect output file name: -5 points
- Error in make file : -5 points
- Make file does not produce an executable file that can be run with one of the commands mentioned in the Programming Environment Section: -5 points
- Hard coded input file name instead of taking as an argument from the command prompt: -5 points
- Not following the Output formatting specified in examples: -5 points
- Any other input/output or submission requirement mentioned in the document: -3 points

Also, we may ask you to fix the above problems and demonstrate your projects.

Miscellaneous:

Implement Red-Black tree and Binary min-heap from scratch, **without using built-in libraries**.
Your implementation should be your own. You must work by yourself for this assignment (discussion is allowed). **Your submission will be checked for plagiarism!!**