ASSIGNMENT 3: McMAHON'S BURGERS

Due Date: Tuesday Mar 22nd, 2022. 11:59 pm.

Goal: The goal of this assignment is to get some practice with combining the concepts learnt so far to choose appropriate data structures, with a focus on heaps and queues, for an efficient implementation of a simulation environment.

Problem Statement: You are the owner of a big and famous burger restaurant called McMahon's Burgers. The restaurant has K billing counters numbered 1 to K. Since your restaurant is a hit amongst youngsters, you get a lot of customers leading to long queues in billing as well as food preparation. You would like to know some statistics like average waiting time, average queue length etc., so that proper steps to improve customer convenience can be taken. You want to do this by running a simulation, as it is cost/time efficient and allows for a range of experimentation on the assumed model.

Assume that customers arrive randomly. You can assume that they are assigned contiguous integer ids, starting from 1. A new customer always joins the billing queue with the smallest length at that time. If there are multiple billing queues with the same smallest lengths, then the lowest numbered queue of those is chosen by the customer. If two customers arrive at the same time, you may assume that they came one after the other in the same instant, i.e., the first customer will already be in some queue, when the second customer is deciding which queue to join. When the arrival time is the same, the customer for whom the `arriveCustomer` function is called first will join the queue before the other customer who is arriving at the same time instant.

Different employees servicing the different billing queues are not equally efficient. The billing specialist (who takes the order and processes payment for a customer) in the billing queue k will take k units of time in completing the order. After the order is completed, the customer order is printed automatically and sent to the chef, who prepares the burgers in the sequence they receive the orders. If two orders arrive simultaneously then the chef chooses the order from the higher numbered billing queue first.

The chef has a large griddle on which at most M burger patties can be cooked simultaneously. Each burger patty gets cooked in exactly 10 units of time. Whenever a patty is cooked another patty can start cooking in that instant. Example, if a patty starts to be cooked at time 10, then it completes at time 20 and another patty starts cooking also at time 20.

Upon cooking, the burger is delivered to the customer in one unit of time. Whenever a customer gets all their burgers, they leave the restaurant instantaneously (it's a take-away restaurant with no dine-in).

Your goal is to simulate this whole process. The simulation has to be driven by events. Events in our simulation environment are arrival/departure of a customer, completion of payment for an order, completion of one or more burgers, putting burgers on the griddle,

etc. For each customer, you have to maintain their state: waiting in the queue, waiting for food, or have already left the building. You will also have to maintain a global clock, which will move forward after all events of a given time point are simulated. You should assume time as discrete integers starting at 0.

If there are multiple events happening at the same time instant, the events are executed in the following order:

1. Billing specialist prints an order and sends it to the chef; the customer leaves the queue.

2. A cooked patty/patties for a customer is/are removed from the griddle.

3. The chef puts another patty/patties for a customer on the griddle.

4. A newly arrived customer joins a queue.

5. Cooked burgers are delivered to customers and they leave.

Your goal of the assignment is to implement this simulation environment through the following operations:

```
public interface MMBurgersInterface {

    public boolean isEmpty();  /* Returns true if there is no further
events to simulate */

    public void setK(int k) throws IllegalNumberException;  /* The
number of billing queues in the restaurant is k. This will remain
constant for the whole simulation. Would be called only once */

    public void setM(int m) throws IllegalNumberException;  /* At
most m burgers can be cooked in the griddle at a given time. This will
remain constant for the whole simulation. Would be called only once */

    public void advanceTime(int t) throws IllegalNumberException;
/* Run the simulation forward simulating all events until (and
including) time t. */

    public void arriveCustomer(int id, int t, int numb) throws
IllegalNumberException;  /* A customer with ID=id arrives at time t.
They want to order numb number of burgers. Note that an id will not be
repeated in a simulation. Time cannot be lower than the time mentioned
in the previous command. Numb must be positive. IDs are consecutive*/
```

```
    public int customerState(int id, int t) throws
IllegalNumberException;  /* Print the state of the customer id at time
t. Output 0 if customer has not arrived until time t. Output the queue
number k (between 1 to K) if a customer is waiting in the kth billing
queue. Output K+1 if the customer is waiting for food. Output K+2 if
the customer has received their order by time t. Note that time cannot
be lower than the time mentioned in the previous command. */

    public int griddleState(int t) throws IllegalNumberException;
/* Print the number of burger patties on the griddle at time t. Note
that t cannot be lower than the time mentioned in the previous
command. */

    public int griddleWait(int t) throws IllegalNumberException;  /*
Print the number of burger patties waiting to be cooked at time t.
I.e., number of burgers for which order has been placed but cooking
hasn't started. Note that t cannot be lower than the time mentioned in
the previous command. */

    public int customerWaitTime(int id) throws
IllegalNumberException;  /* Print the total wait time of customer id
from arriving at the restaurant to getting the food. These queries
will be made at the end of the simulation */

    public float avgWaitTime();  /* Returns the average wait time per
customer after the simulation completes. This query will be made at
the end of the simulation. */

}
```

Write a program which implements such a data-structure. High credit will be given to choice of proper data-structures and efficiency. **For this program, you will need to make use of heaps, queues, and (growable) arrays.** You are not allowed to use internal implementation for any of these (other than using fixed size arrays), and you should implement them on your own - you are free to use any of the code that you have written in one of the earlier assignments.

Hint: A naive implementation which simulates the entire process at every time step may not be the most efficient. Instead, think about how you could make use of heaps to simulate the top priority events efficiently. In particular, your time complexity for simulating each event should be O(log K) where K is the number of billing counters.

Input Output:

The input output format is as follows: You need to add your code to MMBurgers.java file. DO NOT edit other files in the starter code. You can make other java files if required.

Here is an example input and expected output.

Example 1 -

Let Number of counters (K) = 3

Let Size of griddle (M) = 6

Initially, all the three counters are empty.

At t=0,

Customer 1 comes with an order of 3 burgers.

Customer 2 comes with an order of 4 burgers.

Customer 3 comes with an order of 5 burgers.

According to the condition of allocating customers to counters.

Customer1 goes to counter1 (q1)

Customer2 goes to counter2 (q2)

Customer3 goes to counter3 (q3)

Billing specialist in counter 1 takes 1 unit of time for placing an order.

Billing specialist in counter 2 takes 2 units of time for placing an order.

Billing specialist in counter 3 takes 3 units of time for placing an order.

1. Customer1 waits in counter1 for 1 unit of time (from t = 0 to 1).

2. At t = 1, 3 burgers of customer1 are put on the griddle.

3. At t = 2, order of Customer2 is printed and sent to the chef. 3 out of 4 burgers of Customer2 are put on griddle.

4. At t = 3, order of Customer3 is printed and sent to the chef.

5. At t = 11, 3 burgers of Customer1 get cooked. Now, 3 spaces are free. 1 burger of Customer2 and 2 burgers of Customer3 are put on the griddle.

6. At t = 12, 3 burgers of Customer2 are cooked. Now, 3 spaces are free. 3 burgers of Customer3 are put on the griddle. 3 burgers of Customer1 are delivered and Customer1 leaves.

7. At t = 13, Customer2 receives 3 burgers.

8. At t = 21, 1 burger of Customer2 and 2 burgers of Customer3 get cooked.

9. At t = 22, Customer3 receives 2 burger and its remaining 3 burger get cooked. Order of Customer2 is delivered and Customer2 leaves.

10. At t = 23, Customer3 receives all burgers and leaves.

Finish time of Customer1 = 12 units

Finish time of Customer2 = 22 units

Finish time of Customer3 = 23 units

Input Output for example 1

```
[In] : arriveCustomer(1, 0, 3)
[In]: arriveCustomer(2, 0, 4)
[In]: arriveCustomer(3, 0, 5)
[In] customerState(2 , 1)
[Out]: 2
[In]: griddleState(1)
[Out]: 3
[In]:griddleWait(1)
[Out]: 0
[In]: griddleState(2)
[Out]: 6
[In]: customerState( 1, 3)
[Out]: 2
[In] customerState(2, 7)
[Out]: 3
```

```
[In]:griddleWait(10)
[Out]: 6
[In]: griddleState(14)
[Out]: 6
[In]: griddleState(21)
[Out]: 3
[In]: isEmpty()
[Out]: False
[In]: advanceTime(23)
[In]: isEmpty()
[Out]: True
[In]: customerWaitTime(1)
[Out]: 12
[In]: customerWaitTime(2)
[Out]: 22
[In]: customerWaitTime(3)
[Out]: 23
[In]: avgWaitTime()
[Out]: 19.00
```

Example 2:

Let no. Of counters (K) = 2

Let size of griddle (M) = 8

Initially, all the counters are empty.

1. Customer1 comes at t = 0 with order of 5 burgers.
2. Customer2 comes at t = 0 with order of 6 burgers.
3. Customer3 comes at t = 1 with order of 4 burgers.
4. Customer4 comes at t = 1 with order of 5 burgers.

Billing specialist in counter 1 take 1 unit of time for placing an order.

Billing specialist in counter 2 take 2 units of time for placing an order.

1. At t = 0, Customer1 and Customer2 arrive. Customer1 goes to counter1 and Customer2 goes to counter2.

2. At t = 1, Order of Customer1 is placed and sent to the chef. 5 burgers are put on the griddle. Customer3 and Customer4 arrive . Customer3 goes to counter1. As counter1 and counter2 have the same length, Customer4 goes to counter1.

3. At t = 2, Order of Customer2 and Customer3 are placed and sent to the chef. Chef considers order of customer2 (high counter value) first and put 3 burgers of customer2 on the griddle.

4. At t = 3, Order of Customer4 is placed and sent to the chef.

Chef's Queue – (2,3) -> (3,4) -> (4,5)

where (val1, val2) = (customer id, number of burgers waiting to be put on griddle for this customer)

5. At t = 11, 5 burgers of Customer1 are cooked and removed from griddle. Now 3 burgers of customer2 and 2 burgers of customer3 are put on griddle.

Chef's Queue – (3,2) -> (4,5)

6. At t = 12, 3 burgers of customer2 are cooked. Now, 2 burgers of customer3 and 1 burger of customer4 are put on griddle. Order of customer1 is delivered and he leaves.

Chef's Queue – (4,4)

7. At t = 13, 3 burgers of customer2 are delivered to him.

8. At t = 21, 3 burgers of customer2 and 2 burgers of customer3 are cooked. Now, 4 burgers of customer4 are put on griddle.

Chef's Queue - empty

9. At t = 22, 2 burgers of customer3 and 1 burger of customer4 are cooked. Order of customer2 is finished and he leaves.

10. At t = 23, Order of customer3 is finished and he leaves.

11. At t = 31, 4 burgers of customer4 are cooked.

12. At t = 32, Order of customer4 is delivered and he leaves.

Finishing time of customer1 = 12 units

Finishing time of customer2 = 22 units

Finishing time of customer3 = 23 units

Finishing time of customer4 = 32 units

Input Output for example 2

```
[In]: arriveCustomer(1, 0, 5)
[In]: arriveCustomer(2, 0, 6)
[In]: arriveCustomer(3, 1, 4)
[In]: arriveCustomer(4, 1, 5)
[In]: customerState(3, 1)
[Out]: 1
[In]: customerState(4, 1)
[Out]: 1
[In]: griddleState(1)
[Out]: 5
[In]: customerState(1, 5)
[Out]: 2
[In]: griddleState(5)
[Out]: 8
[In]: customerState(2, 6)
[Out]: 3
[In]:griddleWait(6)
[Out]: 12
[In]:griddleWait(12)
[Out]: 7
[In]: griddleState(25)
[Out]: 4
[In]: advanceTime(32)
[In]: isEmpty()
[Out]: True
[In]: customerWaitTime(1)
[Out]: 12
```

```
[In]: customerWaitTime(2)

[Out]: 22

[In]: customerWaitTime(3)

[Out]: 22

[In]: customerWaitTime(4)

[Out]: 31

[In]: avgWaitTime()

[Out]: 21.75
```

Note that while in both examples, customers arrived first, and all queries regarding state etc were issued later, this may not be true in general. And customer arrivals and queries will be interleaved.

What is being provided?

Your assignment folder contains these files:

1. IllegalNumberException.java : Defines a custom exception (Do not modify this file)
2. MMBurgersInterface.java : Defines the simulation interface (Do not modify this file)
3. MMBurgers.java : You have to implement all the functions of MMBurgersInterface in this file and you can add your own classes and methods also.
4. Main.java : File with the main function to test your implementation.
5. Makefile

How to run?

The starter folder contains a Makefile with the necessary commands to run the program, if you are unable to run the makefile just copy-paste the commands in the all: section and run sequentially in the shell/cmd/terminal.

What to submit?

1. Submit your code in a .zip file named in the format **<EntryNo>.zip e.g., 2018ME10000.zip.** Make sure that when we run “unzip yourfile.zip”, there should be a directory <EntryNo> created which should contain all the files which are

provided in the starter code, including modified MMBurgers.java file. You can create additional java class files as per your need/requirements. In addition to your code, “writeup.txt” also be present in the directory. Thus the directory structure should be as follows after unzipping:

<EntryNo>

- All the files provided in the Starter code (unmodified) - other than MMBurgers.java
- Modified MMBurgers.java
- Any other files needed for implementation
- writeup.txt

2. You will be penalized for any submissions that do not conform to this requirement.

3. The writeup.txt should have a line that lists names of all students you discussed/collaborated with (see guidelines on collaboration vs. cheating on the course home page). If you never discussed the assignment with anyone say None.

After this line, you are welcome to write something about your code, though this is not necessary.

What is allowed? What is not?

1. This is an individual assignment.
2. Your code must be your own. You can browse online resources for any general ideas/concepts, but you are supposed to search for/look at specific code meant to solve these or related problems.
3. You should develop your algorithm using your own efforts. You should not Google search for direct solutions to this assignment. However, you are welcome to Google search for generic Java-related syntax.
4. You must not discuss this assignment with anyone outside the class. **Make sure you mention the names in your write-up in case you discuss with anyone from within the class.** Please refer to the plagiarism related guidelines covered in the first lecture and follow them carefully. In case of any doubts, you are free to contact the TAs or the instructors.
5. You are not allowed to use built-in (or anyone else’s) implementations of stacks, queues, vectors, growable arrays and/or other similar data structures - it is ok to

use fixed size arrays as covered in the class. A key aspect of the course is to have you learn how to implement these data structures. You are free to use your own implementation of any data structures from one of the earlier assignments.

6. Your submitted code will be automatically evaluated against another set of benchmark problems. You get a significant penalty if your output is not automatically parsable and does not follow input-guidelines.
7. We will run plagiarism detection software. Anyone found guilty will be awarded a suitable penalty as per IIT rules.

Evaluation Criteria

The assignment is worth 12 points. Your code will be autograded at the demo time against a series of tests. A separate demo will be taken and points will be reserved for correctness of the code (in terms of whether any built-in functions are being used or not), efficiency as well as your ability to answer questions related to your own code. Your code should be as efficient as possible (primarily think about efficiency in terms of the time and memory complexity, and removing any obvious inefficiencies/redundant operations resulting in very slow implementations). Marks will be deducted for inefficient code/implementations.