Application: **Discrete Event Simulator**

Source: Deptt. of CSE, IIT Delhi (with some changes)

My Burger Shop

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

Problem Statement: You are the owner of a big and famous burger restaurant. 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 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.

You are required to implement this simulation environment in Python through the following operations as member functions of a suitable class:

1. isEmpty()

Returns 1(int) if there are no further events to simulate.

2. setK(int *k*)

The parameter ***k*** specifies the number of billing queues in the restaurant. The function returns with an appropriate error message if an attempt is made to modify the value of ***k***, i.e., the function is called more than once.

3. setM(int *m*)

The parameter ***m*** specifies maximum number of burgers that can be cooked in the griddle simultaneously. The function returns with an appropriate error message if an attempt is made to modify the value of ***m***, i.e., the function is called more than once.

4. advanceTime(int *t*)

Runs the simulation forward simulating all events upto (including) time ***t***.

5. arriveCustomer(int *id*, int *t*, int *numb*)

A customer with ID = ***id*** arrives at time ***t*** and orders ***numb*** number of burgers. The function returns with appropriate error messages in the following scenarios:

- the arrival time of a customer is lower than that of a previous customer
- ***numb*** is negative
- the IDs are not consecutive

6. customerState(int *id*, int *t*)

Prints the state of the customer with ID = ***id*** at time ***t***. Outputs 0 if the customer has not yet arrived; the queue number k (between 1 and K) if the customer is waiting in the k^{th} billing queue; k+1 if the customer is waiting for food; k+2 if the customer has received her order by time ***t***. The function returns with an appropriate error message if the value of ***t*** is smaller than that in the previous command.

7. griddleState(int *t*)

Returns the number of burger patties on the griddle at time ***t***. Returns with an appropriate error message if the value of ***t*** is smaller than that in the previous command.

8. griddleWait(int *t*)

Returns the number of burger patties waiting to be cooked at time ***t***, i.e. the number of burgers for which order has been placed but cooking has not yet started. Returns with an appropriate error message if the value of ***t*** is smaller than that in the previous command.

9. customerWaitTime(int *id*)

Returns the total wait time of customer ***id*** from arriving at the restaurant to getting the food. This query will be made at the end of the simulation.

10. avgWaitTime()

Returns the average wait time per customer after the simulation completes. This query will be made at the end of the simulation.