CS1027b Computer Science Fundamentals II

Assignment 3

Due Tuesday, March 10th, 11:55pm via Owl.

Overview

Simulation is a good method for evaluating supermarket checkout strategies. Which is best: a single queue feeding all checkout counters, a single regular queue feeding each checkout counter (all of equal status) and a number of express checkout counters (for customers buying 12 or fewer items) and the rest being regular checkout queues. Using realistic assumptions about customer arrival time, mean number of item to purchase, etc. it is possible to simulate the checkout process to determine which setup provides the minimum average (weighted by number of items) waiting customer times. The important thing in the above brief description of simulation is that one or more queues must be used. This assignment focuses on manipulation of a single event queue for a supermarket counter setup.

During simulation, we must be able to determine the next event that will occur. The problem is this event may not be the first event on the queue. One way to handle this problem is to use a `priority queue`. This strategy uses a heap to provide `O(log(n))` enqueuing and dequeuing operations for queues of size n. There are Java classes that implement this but for this assignment we will use `O(n)` enqueueing and dequeueing operations implemented using our QueueADT class.

Functional Specifications

You are provided with a zip file ass3Files.zip which contains a number of classes. I/O is done as on assignment 1, by using the `InStringFile.java` class. The standard class for queueing as presented in class, `LinkedQueue`, `QueueADT`, `LinearNode` are also provided.

An `Event` class is provided that gives the definition of an event. Its has fields for time (double), type (String) and number of items (int). It has the appropriate getters and setters and a `toString()` method. This class uses the `Service.java` class for a method called `doubleFormat` used in Event's `toString()` method (to perform minor pretty printing). Both of these classes are complete and do not need any modifications.

A complete `Main.java` class is also provided. The static `main` method reads an event file `ass3_events2015.txt`. This is an unordered queue of events. Each event has an eventTime field (double) and an eventType field (String, one of ARRIVAL, IN_SERVICE, DEPART, SNAPSHOT and ALL_DONE). The ARRIVAL, IN_SERVICE and DEPART events represent a customer's arrival at a checkout counter, the customer being served at the counter and the customer leaving the counter. Each has a 3rd field, eventNumItems (int) giving the number of items the customer has. The `main` method reads this file and prints out the data in unsorted order. It also computes the minimum eventTime value for all the ALL_DONE events (if there are more than one) in a local variable called `minAllDoneTime`.

You are required to add two methods to the `EventQueue`. This class has a constructor (provided). You are required to write two methods: `insertEvent()` and `deleteEvents()`.

1. The `insertEvent()` method adds one event object to an eventQueue object. This queue can be assumed to be sorted by time. We insert an event by dequeueing events from eventQueue until we find the insertion place, add the new event and restore the eventQueue to its original state (plus one event for the newly inserted event). It will be necessary to use use a second queue to perform this operation. Because you may (potentially) have to look at each event in the queue, this operation is `O(n)` for an `n` event queue. We also can use the queue's `toString()` method to print out the queue contents before and after insertion of the queue. A variable `debug` controls whether debugging printing occurs or not
2. The `deleteEvents()` method deletes all events with times >= the lowest eventTime value for an ALL_DONE event. That means there may be more than one ALL_DONE event!!! If there are two or more ALL_DONE events with the minimum value, we keep the first occurrence in the output list. There may also be no ALL_DONE event, in which case a WARNING message should be printed. Again, the queue's `toString()` method can be used to print the queue contents before and after insertion. As for `insertEvent`, a second queue may be needed to perform this O(n) operation.

Lastly, the `EmptyCollectionException.java` exception class handles the possibility of trying to dequeue and empty queue.

For a small event file (5 events), we read one line at a time in Main.java, and insert the event information into the eventQueue in the correct order. Then we delete all events in eventQueue with times greater than the smallest time for any ALL_DONE event from the sorted queue (but we do not delete the first ALL_DONE event at that time). For debugging purposes (debug=true in Main.java), we print the queue before and after each insert and delete operation. That code remains in the EventQueue.java file but the 2 places where insertion code and deletion code has been removed are indicated by comments. Put you code there. The debug output can be verbose, in that case set the debug variable to false (debug=false in Main.java) to reduce this output significantly.

For our small event file we get:

```
  1  Time:    9.90 Type: ARRIVAL      Number of items: 33

INSERT EVENT:
Event time:    9.90 Event type: ARRIVAL Event number of items: 33

Event Queue before new event enqueued
-------------------------------------
Empty Queue

Event Queue after new event enqueued
-------------------------------------
Time: 9.9 Type: ARRIVAL Number of items: 33

  2  Time:    0.90 Type: ARRIVAL      Number of items: 16

INSERT EVENT:
Event time:    0.90 Event type: ARRIVAL Event number of items: 16

Event Queue before new event enqueued
-------------------------------------
Time: 9.9 Type: ARRIVAL Number of items: 33

Event Queue after new event enqueued
-------------------------------------
Time: 0.9 Type: ARRIVAL Number of items: 16
Time: 9.9 Type: ARRIVAL Number of items: 33
```

```
  3  Time:    9.00 Type: ALL_DONE

INSERT EVENT:
Event time:    9.00 Event type: ALL_DONE

Event Queue before new event enqueued
-------------------------------------
Time: 0.9 Type: ARRIVAL Number of items: 16
Time: 9.9 Type: ARRIVAL Number of items: 33

Event Queue after new event enqueued
-------------------------------------
Time: 0.9 Type: ARRIVAL Number of items: 16
Time: 9.0 Type: ALL_DONE Number of items not specified
Time: 9.9 Type: ARRIVAL Number of items: 33

  4  Time:    0.10 Type: DEPART       Number of items: 8

INSERT EVENT:
Event time:    0.10 Event type: DEPART Event number of items: 8

Event Queue before new event enqueued
-------------------------------------
Time: 0.9 Type: ARRIVAL Number of items: 16
Time: 9.0 Type: ALL_DONE Number of items not specified
Time: 9.9 Type: ARRIVAL Number of items: 33

Event Queue after new event enqueued
-------------------------------------
Time: 0.1 Type: DEPART Number of items: 8
Time: 0.9 Type: ARRIVAL Number of items: 16
Time: 9.0 Type: ALL_DONE Number of items not specified
Time: 9.9 Type: ARRIVAL Number of items: 33

  5  Time:    4.00 Type: ALL_DONE

INSERT EVENT:
Event time:    4.00 Event type: ALL_DONE

Event Queue before new event enqueued
-------------------------------------
Time: 0.1 Type: DEPART Number of items: 8
Time: 0.9 Type: ARRIVAL Number of items: 16
Time: 9.0 Type: ALL_DONE Number of items not specified
Time: 9.9 Type: ARRIVAL Number of items: 33

Event Queue after new event enqueued
-------------------------------------
Time: 0.1 Type: DEPART Number of items: 8
Time: 0.9 Type: ARRIVAL Number of items: 16
Time: 4.0 Type: ALL_DONE Number of items not specified
Time: 9.0 Type: ALL_DONE Number of items not specified
Time: 9.9 Type: ARRIVAL Number of items: 33

5 events read and inserted in order in the eventQueue

Event Queue before all events with eventTime<=4.0 deleted
-------------------------------------------------------
Time: 0.1 Type: DEPART Number of items: 8
Time: 0.9 Type: ARRIVAL Number of items: 16
```

```
Time: 4.0 Type: ALL_DONE Number of items not specified
Time: 9.0 Type: ALL_DONE Number of items not specified
Time: 9.9 Type: ARRIVAL Number of items: 33

Event Queue after all events with eventTime<=4.0 deleted
--------------------------------------------------
Time: 0.1 Type: DEPART Number of items: 8
Time: 0.9 Type: ARRIVAL Number of items: 16
Time: 4.0 Type: ALL_DONE Number of items not specified
```

Non-functional Specifications

- Your program has to be compilable under Eclipse.
- Use Javadoc comments for each class and method. All significant variables must be commented.
- Use Java conventions and good Java programming techniques (meaningful variable names, conventions for variable and constant names, etc). Indent your code properly.
- Remember that assignments are to be done individually and must be your own work.