Lab 11 Queue Implementation and Client

Goal

In this lab you will work with queues. You will implement an event queue and then use it along with another queue in a simulation of customers waiting in a line at a bank.

Resources

- Chapter 10: Queues, Deques, and Priority Queues
- Chapter 11: Queue, Deque, and Priority Queue Implementations
- *VectorQueue.java*—A sample implementation of `Queue` (in *QueuePackage*)
- *Bank.jar*—The final animated application
- `Lab11Graphs.pdf` – Full size versions of the blank graphs for this lab
- Lab Manual Appendix —An Animation Framework

In `javadoc` directory

- *QueueInterface.html*—API documentation for the queue ADT
- *PriorityQueueInterface.html*—API documentation for the priority queue ADT
- *SimulationEventQueueInterface.html*—API documentation for the event queue ADT
- *SimulationEventInterface.html*—API documentation for the events on the event queue ADT
- *BankLine.html*—API documentation for a class representing a line of customers in a bank
- *Customer.html*—API documentation for a class representing a customer in a waiting line
- *Report.html*—API documentation for a class representing a class that will display a report for the simulation

Java Files

- *BankActionThread.java*
- *BankApplication.java*
- *BankLine.java*
- *Customer.java*
- *CustomerGenerator.java*
- *Report.java*
- *Teller.java*

There are other files used to animate the application. For a full description, see Appendix: An Animation Framework of this manual.

In *QueuePackage* directory

- *EmptyQueueException.java*
- *PriorityQueueInterface.java*
- *QueueInterface.java*
- *SimulationEvent.java*
- *SimulationEventInterface.java*
- *SimulationEventQueueInterface.java*
- *VectorQueue.java*

Introduction

A queue is a linear data structure that allows you to add items at the end and remove items from the front. It is a natural representation of a waiting line. A priority queue changes the add method. Instead of always adding at the end, it will insert the item into the queue according to a priority. The higher the priority, the closer to the front the item will be.

Event-Driven Simulations

One way of doing a simulation is to keep a list of all the events that have been scheduled to occur in the future. At each turn in the simulation, the event with the earliest time is removed from the list and processed. All of the events will be associated with an object. Processing the event will change the state of the object and may schedule new events to be processed at a later time. When designing a simulation, it will be important to have an idea of how the sequence of events flows in the system.

For example, consider a traffic light simulation where the light changes color every minute. What are the possible events in the simulation?

- The light turns green.
- The light turns red.
- The light turns yellow.
- A car arrives at the intersection.
- A car leaves the intersection.

Consider the traffic light first. What happens when the light turns red?

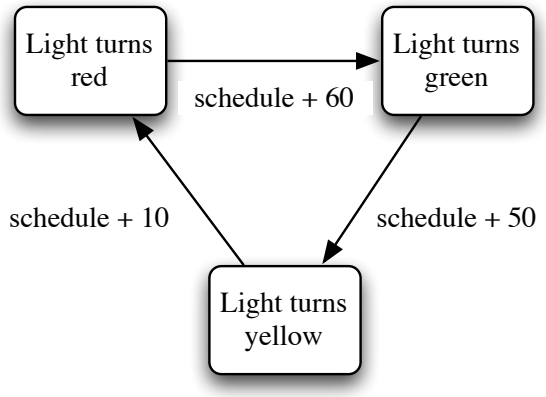
Event: Light turns red
State change: The color of the light becomes red.
Operations/Events to schedule: The light turns green in 60 seconds in the future.

Similarly, for the next two events,

Event: Light turns green
State change: The color of the light becomes green.
Operations/Events to schedule: The light turns yellow in 50 seconds in the future.

Event: Light turns yellow
State change: The color of the light becomes yellow.
Operations/Events to schedule: The light turns red in 10 seconds in the future.

Pictorially, the events for the light are



Now consider the events for the car. What happens when the car arrives at the light? It must check the color of the light to see if it can go through the intersection. If the light is red, the car will have to stop and wait. Can the car schedule when it will leave the intersection? No. Unlike with the light, the time that the car leaves has yet to be determined. There must be some place where the car will wait until it can be notified that it may continue. Once it receives notification, then it can schedule when it will leave the intersection.

Where will the car wait? The natural choice is a queue associated with the traffic light. What happens when the car arrives and the light is green? Can the car just go through the intersection? No! If there are cars waiting, a crash has just happened.

Event: Car arrives at the intersection

State change: None for now (arrival time could be recorded though).

Operations/Events to schedule: If the light is green and there are no cars waiting, schedule leaving the intersection 3 seconds in the future. Otherwise, put the car on the queue.

The question now is, "How do cars waiting at the light get going again?" Some event must trigger them. In this case, it will be when the light turns green. The design for that event must change.

Event: Light turns green (Version 2)

State change: The color of the light beomes green.

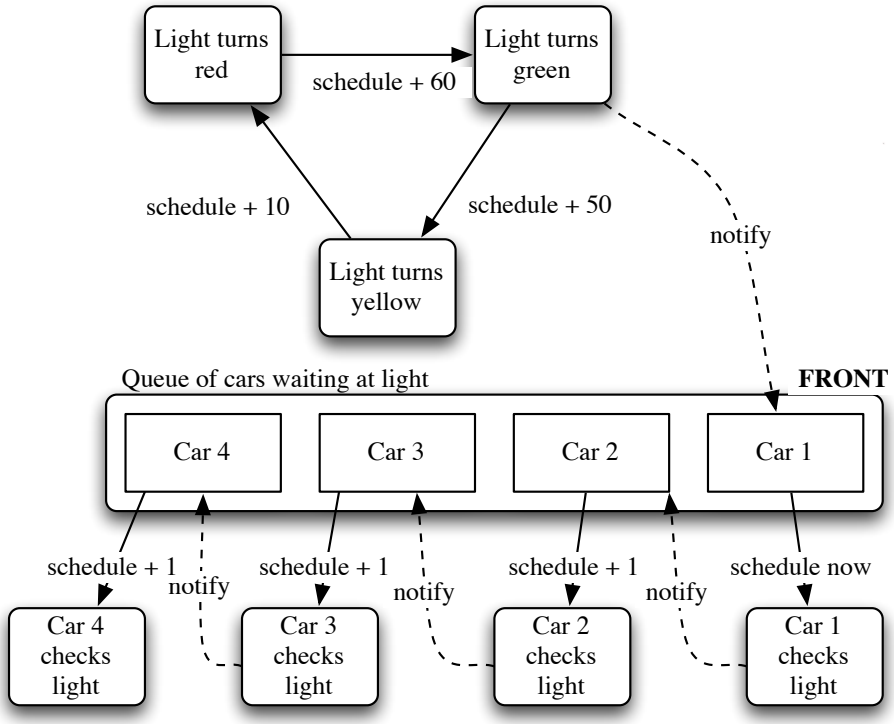
Operations/Events to schedule: The light turns yellow in 50 seconds. The light notifies the first car waiting in the queue that it can go now.

This takes care of the first car in the line, but what about the others? Each car in turn will notify the one behind it. This raises another question, "Can all the cars make it through the light in a single turn?" Sometimes the answer will be no. The time that the light is green will play a role. Clearly, all the cars cannot start at the same time but must be staggered. This indicates that the event queue must play a role. There must be another event.

Event: Waiting car checks intersection

State change: None.

Operations/Events to schedule: If the light is red or yellow, do nothing. Otherwise, schedule leave intersection 3 seconds in the future. Also, it will remove itself from the queue and notify the next car that it can check the intersection in 1 second.



There is only one event left to think about.

Event: Car leaves intersection
State change: None for now. (The time the car leaves the intersection could be recorded).
Operations/Events to schedule: None.

Another question is how do all the car arrival events get scheduled. One possibility is that they are all generated and placed on the event queue before the simulation starts. Another possibility is that there is a car generator object with a single event of its own.

Event: Generate Car
State change: None.
Operations/Events to schedule: Create a car and schedule it to enter the intersection now. Generate delta (a random time interval). Schedule a generate car event at time delta in the future.

This is similar to the operation of the traffic light except that the events occur at a random interval instead of a fixed one.

Events

Essentially, an event encapsulates a time and what to do at that time. Events will support the interface in `SimulationEventInterface`. The four methods that must be supported are:

getTime(): Get the time for the event.
getDescription(): Get a string describing the event.
getPostActionReport(): Get a string describing the results of the event.
process(): Do the actions required for the event.

The time and description of the event will be set when the event is created. The `process` method will be specialized for each particular event class. The last thing that `process` should do is to set a string, which the `getPostActionReport` method wil return. Strictly speaking, the two methods that return the strings are not needed for the simulation, but they are useful in the animated application. The only really interesting method is `process()`.

To make it easier to create new events, the abstract class `SimulationEvent` has been created. It defines all of the methods, except for process, which is abstract. All a subclass needs to do is to have a constructor and a `process` method. An example of such a class is the inner class `GenerateCustomerEvent` in the `CustomerGenerator` class. Making the event class an inner class eases the coding marginally. Because the inner class will have access to all of the private variables of the class it is inside, the state of the object can be changed directly if needed.

Pre-Lab Visualization

The Simulation Event Queue

The event queue is very similar to a priority queue. It has two major differences. The first difference is that the event queue acts as a timekeeper for the simulation. Every time an event is removed from the event queue, the simulation time moves forward to be the same as the time for the event that was just removed.

The second difference is that events that are before the current time of the event queue will not be added. If that were allowed, the arrow of time would not always go in the forward direction.

The major operation is the `add()` method. Suppose the following event is received by the add method.

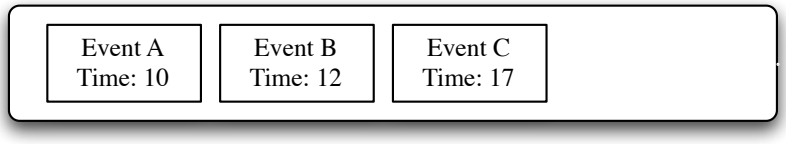
[3]

In each of the following event queues, show where it would be added.

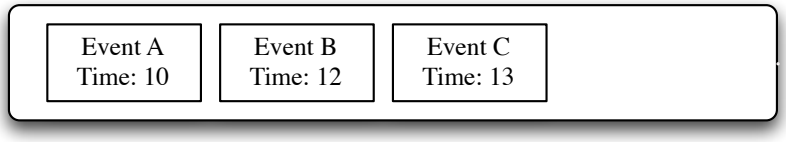
The Events The current time: 10

The Events The current time: 10

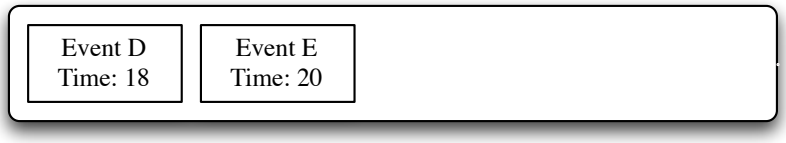
The Events The current time: 10

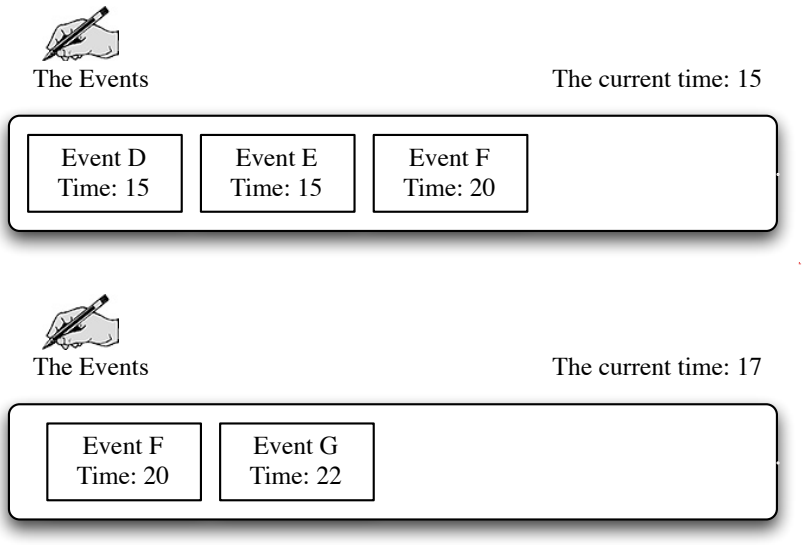


The Events The current time: 10



The Events The current time: 15





Give an algorithm for inserting an event into an event queue.

Show the operation of the algorithm on the previous event queue examples.

Using an Event Queue in a Simulation

Suppose that the event queue is working. Some code is needed to drive the simulation forward. (It is called an event loop.)

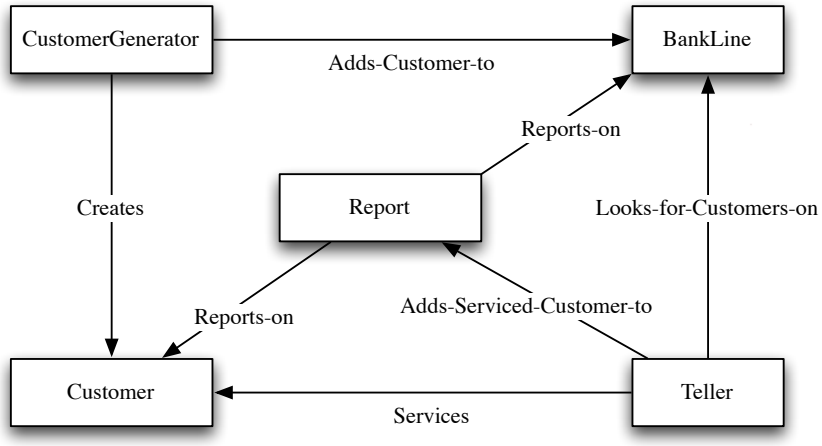
When should the simulation stop?

At each step in the simulation, what must happen?

Given an algorithm for the event loop.

The Bank Line Simulation

In this lab, an event simulation will be created. It will simulate customers waiting to be served at a bank by tellers. Beside the animation and event classes, there will be five main classes that implement the bank simulation. Four of them will have associated animation displays. Pictorially, their relations are



A `Customer` in this simulation has very few responsibilities. The major responsibility of this class is to be able to draw a graphic representation of itself. When the customer is created, it will be given a name and the current time. At some time in the future it will be notified (by the teller) that it has been serviced. Once that has happened, it has the responsibility to be able to compute the time that it waited. The time it starts waiting and the time it is serviced will both be displayed. Consult *`Customer.html`* for names of the methods it implements.

The `BankLine` class is nearly as free of responsibilities as `Customer`. Besides its responsibilities as a queue, it has the additional responsibility of being able to draw itself.

The `Report` class has the responsibility for presenting the results of the simulation. It will produce two averages. The first is the average time that the customers currently in the line have waited. The second is the average time that the serviced customers waited. To accomplish the first task, the `Report` class will have access to the bank line. It will iterate over the customers in the line, requesting the time they started waiting. It will use these values to compute the average time waited. To satisfy the second requirement, it will keep a list of customers that have finished. It will iterate over them, requesting the time they waited. This places a requirement on `Teller` to give the customer to the `Report` object when the teller removes the customer from the bank line. The last requirement of `Report` is to display the current time of the simulation. This means that the simulation loop will need to inform the `Report` object of the current time after every step.

The `CustomerGenerator` class is one of the two classes that interacts with the event queue. It has an event for customer generation.

Event: Generate Customer

State change: Customer name/count is updated.

Operations/Events to schedule: Create a new customer. Add the customer to the bank line. Schedule a generate customer event at a random time in the future.

The `Teller` class is the other class that interacts with the event queue. It will check the line for a customer to service.

What will happen if there is a customer in line?

What will happen if there isn't a customer in line?

Similar to generating a customer, the amount of time required to handle the customer will be random. The maximum time will be one of the parameters of the constructor. The other time used by the teller class is the period between checking the line. We shall assume that the tellers are very vigilant and the line will be checked every second.

Complete the event specification. (`Teller` has a method `serve()`, which encapsulates its responsibilities for serving the customer.)

Event: Check Bank Line for a Customer

 **State change**:

 **Operations/Events to schedule**:

Give an algorithm for the process method of the event.

Directed Lab Work

Implementing the Event Queue

All but two of the classes needed in today's lab exist. The first class that will be created is the SimulationEventQueue.

Step 1. In the `QueuePackage`, create a new class named `SimulationEventQueue`.

Step 2. In the class declaration, make it implement `SimulationEventQueueInterface`.

Step 3. Create method stubs for each of the methods in the interface.

Checkpoint: The class should compile now.

Step 4. Create a private variable to store the current simulation time.

Step 5. Create a private variable to store the contents of the queue.

Step 6. Implement all of the methods except for `add()`. You may find the class `VectorQueue` helpful.

Step 7. In the `remove()` method, add code to change the current time of the event queue.

Step 8. Refer to the pre-lab exercises and implement the `add()` method.

Checkpoint: The class should compile.

The Bank Line Animation

Checkpoint: The bank application should run. At the very start of the set up phase init() will be called. The customer generator in its constructor puts its initial event on the event queue. That should show up as the next event. No customers should be in the line. The bank teller Fred should be waiting patiently for customers to show up. The report should indicate that there are no customers waiting or served. The simulation time is 0.0.

At this point, it would be nice to see the event queue in operation. Code to drive the simulation will be added into the BankActionThread.

Creating the Event Loop

Step 9. In the method `executeApplication()` in `BankActionThread`, add code that will repeatedly take events from the event queue and process them. Refer to the pre-lab exercises.

The display for the simulation has a few requirements for what happens in the event loop.

Step 10. Inside the loop after the event has been processed, get the post action report from the event and use it to set `lastEventReport`.

Step 11. If there is a next event, get the description and use it to set `nextEventAction`.

Step 12. Update the time for the report.

Step 13. The last code in the loop should be the line that will pause the animation.

```
animationPause();
```

Checkpoint: Compile and run the application. Press go. Customers should be generated and placed one at a time into the line. You should see them. Unfortunately, Fred is busy drinking coffee and is not yet helping the customers. The simulation should stop once it hits 1000.

Completing the Teller Event

It is time for Fred to get to work. The process method for CheckForCustomerEvent inside the Teller class needs to be completed.

Step 14. Refer to the pre-lab exercises and complete the code for the method `process()`.

Step 15. If no customer was served, set `serving` to `null`.

Step 16. At the end of processing, make sure to set `postActionReport` with a string describing the actions taken by the event.

Step 17. In the constructor for `Teller`, add code that will generate the first `CheckForCustomerEvent`. (This is similar to how the customer generator operates.)

Checkpoint: The teller should now take customers from the line. As customers are serviced, the report should change. Step and carefully trace the operation of the simulation. Verify that it is operating correctly.

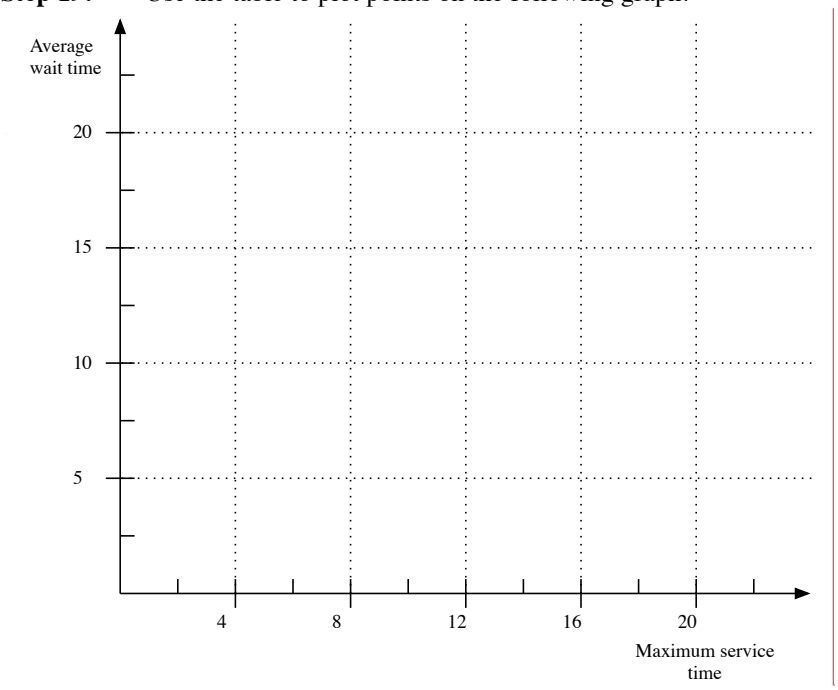
Change the service interval time and verify that customers are handled quicker.

Graphing the Results

Step 18. Run the simulation with a maximum interval of 20 and a simulation time limit of 1000. Fill in the following table.

MAXIMUM SERVICE TIME	AVERAGE WAIT TIME FOR CUSTOMERS SERVED
6	
8	
10	
12	
14	
16	
18	
20	
22	
24	

Step 19. Use the table to plot points on the following graph.



Suppose the service time is greater than the interval that customers appear. You expect that the teller will fall behind and the length of the line will increase without bound. If the service time is less, you expect that the teller will be able to keep up.

The interesting question is what happens when the service and interval times are the same. How long will the line be on average in this case? It turns out that the average, as the simulation time increases, will approach infinity. This will also have an effect on the average waiting time.

Step 20. Run the application 20 times with the initial settings and record the average. (Warning: If you set the animation delay time to be too small, the animation may not stop at the end of the simulation but restart at time zero.)

Step 21. What was the maximum average wait?

Step 22. What was the minimum average wait?

Post-Lab Follow-Ups

1. Implement a new version of `SimulationEventQueue`, which uses a linked list.

2. Add two private queues to the `CustomerGenerator` that will store name syllables. When generating a name, take one syllable from each queue and concatenate them together. Put the syllables back on the ends of their respective queues. Set the queues so that each has a length that is a different prime number.

3. Modify the simulation so that it has two tellers that take customers from a single line.

4. Modify the simulation so that it has two tellers, each with their own separate line.

5. Add an event to the `Report` class to collect statistics on the customer waiting line. Each time the event occurs, the length of the line will be recorded in order to compute the average, minimum, and maximum length of the line.

6. One of the good things about an event-driven simulation is that it will jump over times where nothing is happening. In the animation, this can cause sudden jumps in time. Add a class that will generate dummy events every second. The only responsibility of the event is to schedule the next dummy event.

7. *For those familiar with statistics and calculus*: Change the `CustomerGenerator` class so that the time between customers is determined according to a Gaussian distribution with a given mean and standard deviation. Make a similar change for the service time for the `Teller` class. There are other distributions you can try as well. (Hint: To do this, consult a book on statistics and find a table that gives the cumulative distribution for a Gaussian distribution. Generate a value between 0 and 1 and interpolate to find a z score. From this, use the mean and standard deviation for the desired distribution to find your value.)

8. Change the `BankLine` to be a priority queue. Each customer generated will have one of two priorities, high or low. Have the `Report` class report the average waiting time for each priority level.

9. *For those familiar with Java graphics: M*ake the event queue a displayed object by the animation.