

Stephen Patpatyan

CS2400

Rick Ramirez

12 March 2025

GITHUB LINK TO PROJECT: <https://github.com/StevePatpatyan/CS2400-Project-3>

Queue Client

Goal

In this assignment 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 7: Queues, Deques, and Priority Queues
- Chapter 8: Queue, Deque, and Priority Queue Implementations
- *VectorQueue.java*—A sample implementation of *Queue* (in *QueuePackage*)
- *Bank.jar*—The final animated application
- *Graphs.pdf* – Full size versions of the blank graphs for this assignment
- *animatedapp* —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`
- `SimulationEventQueueTest.java`
- `Teller.java`

There are other files used to animate the application in the package `animatedapp`. For a full description, see [Appendix: An Animation Framework](#) at the end 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.

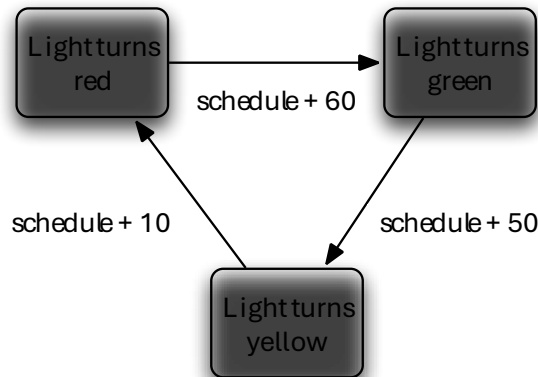
Operations/Events to schedule: The light turns yellow in 50 seconds in the future.

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 becomes green.

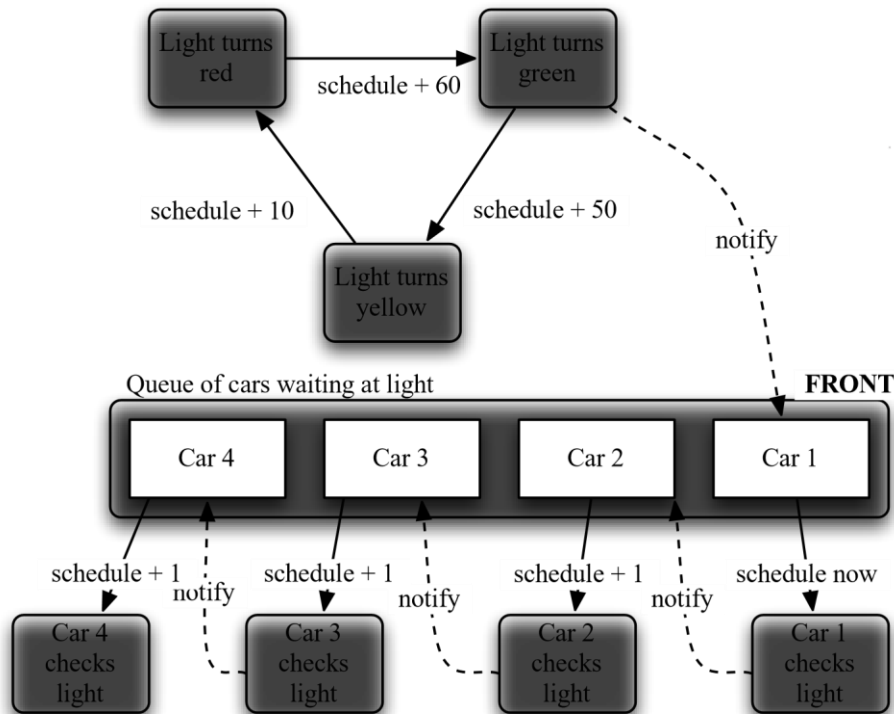
Operations/Events to schedule: The light turns yellow in 50 seconds. The light notifies the first car waiting in the queue that it can go now.

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 will 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.

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. We do have a design choice for how the clear method will work. It can either reset the simulation time back to the initial setting or it can leave it alone. We will choose the second option.

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

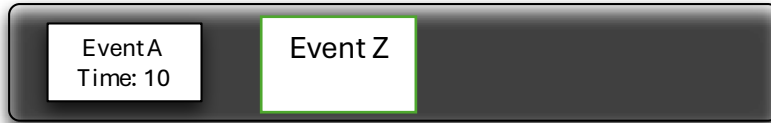
Event Z Time: 15

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



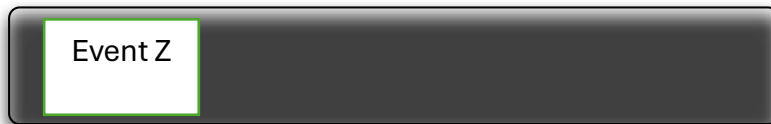
TheEvents

Thecurrenttime: 10



TheEvents

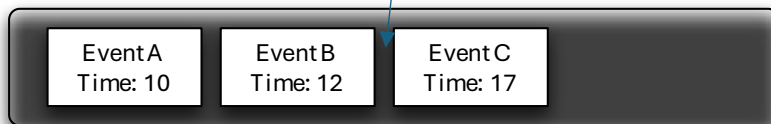
Thecurrenttime: 10



TheEvents

Event Z

Thecurrenttime: 10



TheEvents

Thecurrenttime: 10

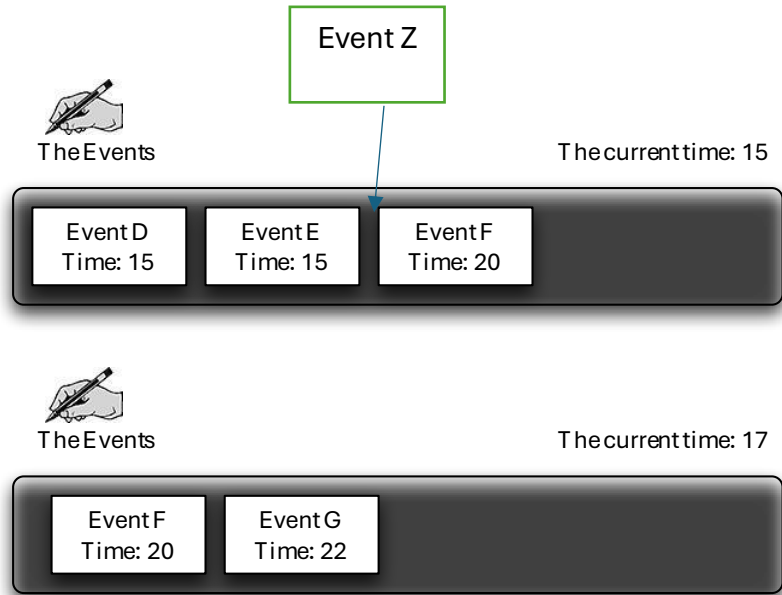


TheEvents

Thecurrenttime: 15

Event Z





DON'T ADD HERE

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



Check if new event time is less than current simulation time:

yes: don't add new event

no: move on to adding new event

Check if queue is empty

yes: add event

no: loop while queue has an element next on iteration and larger time event is not found

iterate position to add new event

iterate event to check against new event

if larger time found

add new event in front of larger time/at recorded position

else:

add new event at end of queue

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



EXAMPLE 1:

- New event time < current time? NO, so move to adding
- Empty? NO, so move to loop

Current time: 10

Event A

- 1st event. Is time > new event time? No
- End loop because last element of queue reached

Event A

Event Z

- Was larger time found? NO, so add new event to end of queue

EXAMPLE 2:

Current time: 10

- New event time < current time? NO, so move to adding

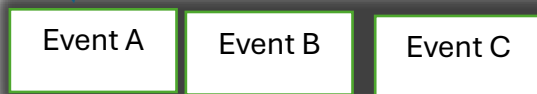
Event Z

- Queue empty? YES, so add new event to front of queue

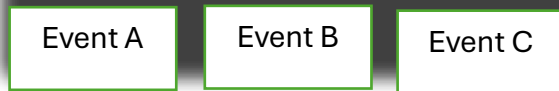
EXAMPLE 3:

Current time: 10

- New event time < current time? NO, so move to adding
- Empty? NO, so move on to iterations



- 1st event. Is time > new event time? NO, move to next event



- 2nd event. Is time > new event time? NO, move to next event

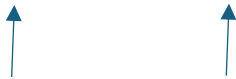
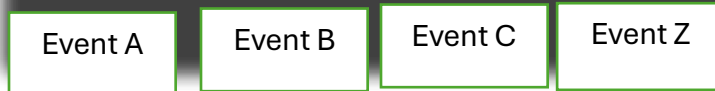


- 3rd event. Is time > new event time? YES. Loop ends and add new event in front

EXAMPLE 4:

Current Time: 10

- New event time < current time? NO, so move to adding



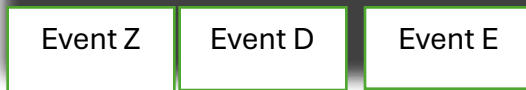
- Queue not empty. Move to iterating
- 1st, 2nd, and 3rd event has time not greater than new event time.
- No larger time found. Add new event to end of loop.

EXAMPLE 5:



Current Time: 15

- New event time < current time? NO, so move to adding



- Queue not empty. Move to iterating.
- 1st event time greater than new event time.

- Larger time found. Add new event in front of first event.

EXAMPLE 6:

Current time: 15

- New event time < current time? NO, so move to adding



- Queue not empty. Move to iterating.
- 1st and 2nd event time not greater than new event time
- 3rd event time is greater.
- Larger time found. Add new event in front of 3rd event.

EXAMPLE 7:

Current time: 17

- New event time < current time. YES, so do not add



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?



The simulation should stop when it reaches a certain time limit, or if it is explicitly stopped.

At each step in the simulation, what must happen?



Event that are scheduled to be executed at that time should be executed and the time should be iterated by one (generally).

Given an algorithm for the event loop.



EVENT LOOP ALGORITHM BELOW

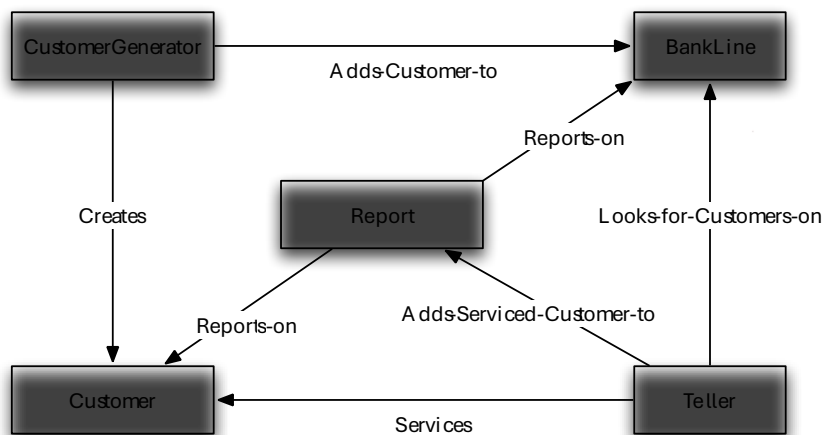
Loop: While current time \leq max time or if simulation is not explicitly stopped
 Loop: While time of event in front == current time
 Execute event
 End loop once the event in front has a time greater than the current time

 Add 1 to the current time

End loop/simulation

The Bank Line Simulation

In this assignment, 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?



Teller services the customer in the front of the line, and adds serviced customer to the report. Report computes new average waiting times.

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



There are no customers to service, so average customer waiting times stays the same.

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: Customer is removed from, added to serviced customers on report, and marked as served (ONLY IF THERE IS A CUSTOMER TO BE SERVED)



Operations/Events to schedule:

Serve customer: Add served customer to report, calculate new average waiting time for served customers and customers waiting in line, Mark customer as served at the current time.

Give an algorithm for the process method of the event.



Check if bank line is empty

Yes: move to next event

No:

Serve customer (remove from line)

generate next check customer event

Directed Work

An Animated Simulation

In this assignment you will complete an animated application that simulates customers being served in a bank. Approximately half of the classes in the application implement the framework that animates the application. A description of the classes in the framework is given in Appendix: An Animation Framework in this manual. For the most part, these classes can be ignored as the animated application is developed as long as appropriate care is taken. The appendix also gives instructions for using the framework to create new animated applications.

All but two of the classes needed in this assignment exist. The first class that will be created is the `SimulationEventQueue`.

Implementing the Event Queue

- 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 visualization exercises and implement the `add()` method.

Checkpoint: The class should compile. Run the `SimulationEventQueueTest` program. While it is not an exhaustive test of the `SimulationEventQueue` class, it does basic tests. If any of the tests fail, diagnose the problem.

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 visualization 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 visualization 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.

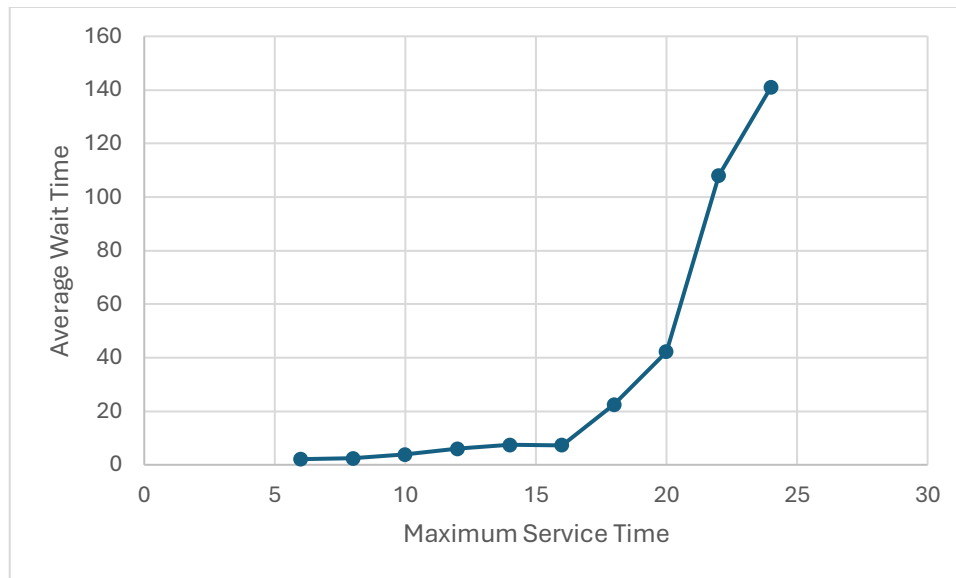
Change the service interval time and verify that customers are handled quicker.

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	2.11
8	2.38
10	3.82
12	5.99
14	7.46
16	7.36
18	22.54
20	42.15
22	108.09
24	140.97

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.)

54.84

Step 21. What was the maximum average wait?

119.35

Step 22. What was the minimum average wait?

18.24

Stack Client

Goal

In this assignment you will complete an application that uses the Abstract Data Type (ADT) stack.

Resources

- Chapter 5: Stacks

In javadoc directory

- *StackInterface.html*—Interface documentation for the interface `StackInterface`

Java Files

- *StackInterface.java*
- *StackSort.java*
- *VectorStack.java*

Introduction

In computer science, one of the important basic structures is the stack. It is of both theoretical and practical use.

In its simplest form it has three operations: push, pop, and empty. Push places a value on the top of the stack. Pop removes the top value from the stack. Empty is a test to determine if the stack has any values in it. Some specifications give a fourth operation called peek (or top). Peek will return the top value on the stack but leaves the number of

items unchanged. Strictly speaking, peek is unnecessary because a pop followed by a push will mimic its operation. Before continuing the assignment you should review the material in Chapter 5. In particular, review the documentation of the interface

StackInterface.java.

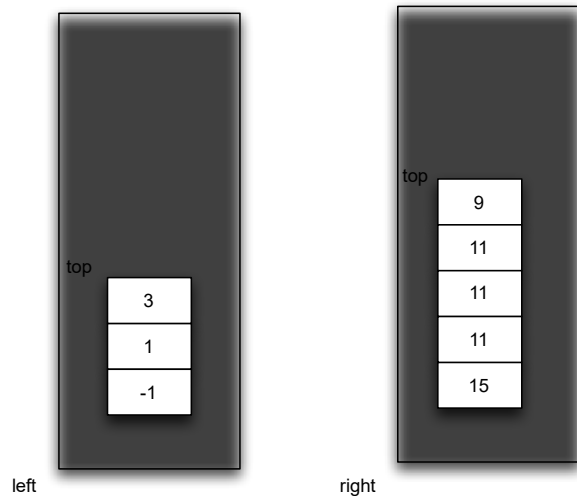
In theoretical computer science, one of the problems of interest is recognizing words in a language. In this context, a language is a set of words that follow some pattern. For example, one language is all the words from the alphabet $\{0, 1\}$ that have equal numbers of zeros and ones. The word 001011 is in the language, but the word 00111 is not. There are a number of primitive models of computation that have different abilities. One kind of model is a machine called a pushdown automata (PDA). It has a finite control (program) and a single stack that it can use for memory. While fairly powerful, a PDA does have some surprising limits. For example, while a PDA can recognize words of the form $0^n 1^n$, it cannot recognize $0^n 1^n 0^n$. Modern computer languages are often recursively defined by a grammar, which can be recognized by a PDA.

The application you will complete implements a sorting algorithm. Sorting is a general problem where given a collection of items, you arrange them in order from smallest to largest. We will restrict ourselves to a collection of integer values in an array. For example, if given the integers 8, 2, 9, 1, 1, 3; their sorted order is 1, 1, 2, 3, 8, 9. You will examine a number of different sorting techniques in Chapters 8 and 9. While it is not obvious, the sort that we will be doing in this assignment is equivalent to the insertion sort from Chapter 8 and has the same performance. As with any of the sorts from Chapter 8, the stack sort should not be used in general applications.

Visualization

Stack Sort

In order to sort values we will use two stacks which will be called the left and right stacks. The values in the stacks will be sorted and the values in the left stack will all be less than or equal to the values in the right stack. The following example illustrates a possible state for our two stacks. Notice that the values in the left stack are sorted so that the smallest value is at the bottom of the stack. The values in the right stack are sorted so that the smallest value is at the top of the stack. If we read the values up the left stack and then down the right stack, we get -1, 1, 3, 9, 11, 11, 11, 15, which is in sorted order.



Suppose that we have a new value that we want to put into our sorted collection. We will want to put it on the top of one of the two stacks, but we may have to first move values around.

No moves required:

Consider adding the value 5 to the example shown above. We do not have to move any values and can place the 5 on the top of either stack and still have a sorted collection.

Which values would not require that the contents of the stacks be changed?



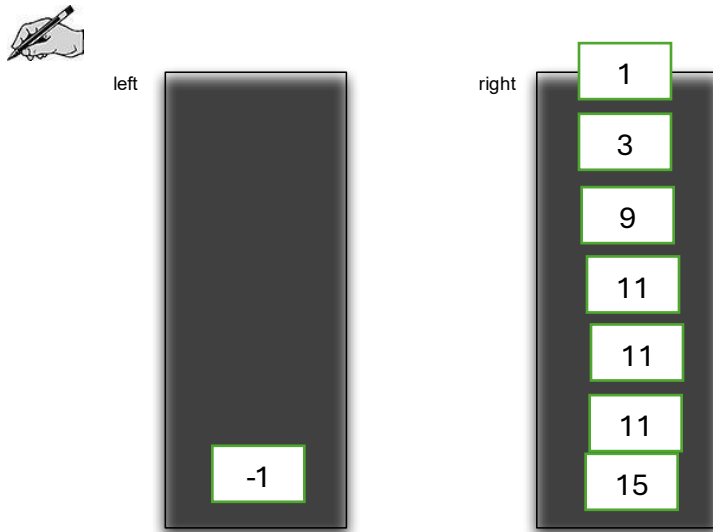
3, 4, 5, 6, 7, 8, 9

Moves from left to right required:

Consider adding the value 0 to the example shown above. We must move values from the left stack to the right stack.

How many values must be moved and what is the state of the two stacks before we add the value 0?

2 values must be moved, being the 3 and 1 from the left stack to the right stack.



What condition should we use to determine if enough values have been moved?



If the value we are inserting is greater than or equal to the top value of the left stack or if either stack is empty.

Consider adding the value -2 to the example shown above. Again must move values from the left stack to the right stack.

How many values must be moved and what is the state of the two stacks before we add the value -2?

3 values must be moved to the right stack, being all of the left stack values.

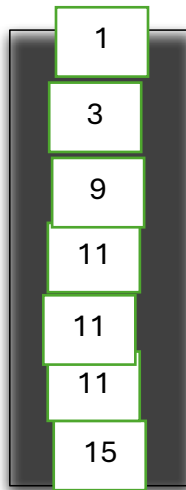




left



right



What condition should we use to determine if enough values have been moved?

If the value we are inserting is greater than or equal to the top value of the left stack or if either stack is empty.



Write code using iteration that will move values from the left to the right stack as required.



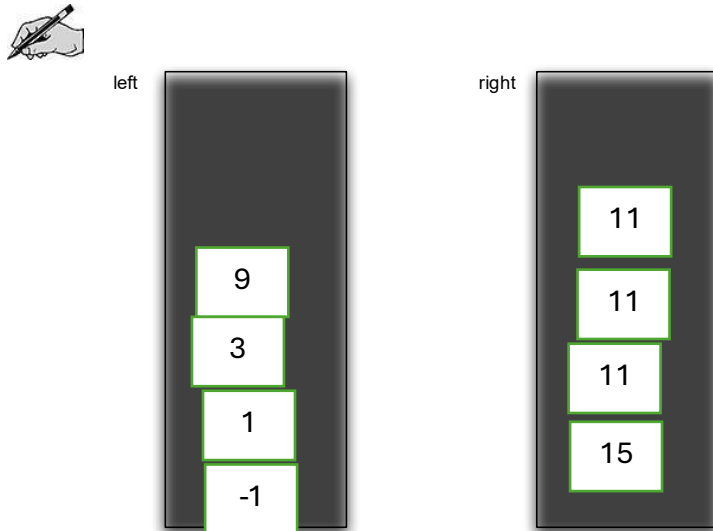
```
while (!left.isEmpty() && value < left.peek()) {  
    right.push(left.pop());  
}
```

Moves from right to left required:

Consider adding the value 11 to the example shown above. We must move values from the right stack to the left stack.

How many values must be moved and what is the state of the two stacks before we add the value 11?

1 value must be moved, being the 9 from the right stack.



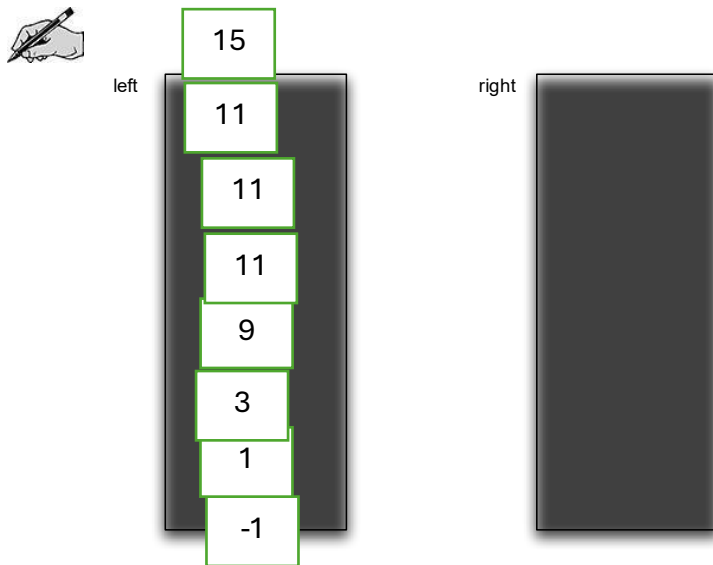
What condition should we use to determine if enough values have been moved?

If the value we are inserting is less than or equal to the top value of the right stack or if either stack is empty.

Consider adding the value 20 to the example shown above. Again we must move values from the right stack to the left stack.

How many values must be moved and what is the state of the two stacks before we add the value 20?

All 5 values from the right must be moved to the left stack.



What condition should we use to determine if enough values have been moved?

If the value we are inserting is less than or equal to the top value of the right stack or if either stack is empty.



Write code using iteration that will move values from the right to the left stack as required.



```
while (!right.isEmpty() && right.peek() < value) {  
  
    left.push(right.pop());  
}
```

Adding all the values from an array into the two stacks:

We can add the values from the array one at a time to the stacks. Putting together the pieces from the previous questions, write an algorithm for this task.



DO THIS FOR EACH VALUE IN THE ARRAY

```
if (!left.isEmpty() && value < left.peek()){ // new value is less than the top of the left
stack, meaning move left stack values to right
```

```
    while (!left.isEmpty() && value < left.peek()) {
        right.push(left.pop());
    } // end while
```

```
} // end if
```

```
else if (!right.isEmpty() && value > right.peek()) { // new value is greater than top of right
stack, meaning move right to left
```

```
    while (!right.isEmpty() && value > right.peek()) {
```

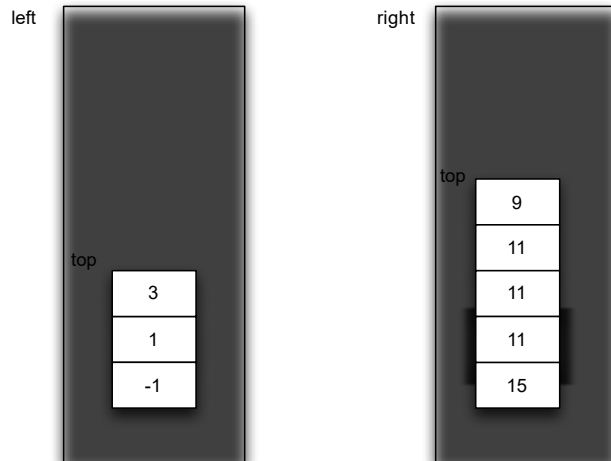
```
        left.push(right.pop());
    } // end while
```

```
} // end else if
```

```
// now push value into appropriate spot between the stacks after room is made
right.push(value);
```

Putting the values into a new array:

For our particular sorting algorithm, we are going to create a second array with the values from the original array in sorted order. Therefore, the final task we need to do before we return is to put the values into the `result` array. Consider again our example.



Suppose we pop the values off of the left

stack one at a time. What order do we get?

3, 1, -1 (descending order)

Suppose we pop the values off of the right stack one at a time. What order do we get?



9, 11, 11, 11, 15 (ascending order)

This suggests that if we move the values from the left stack to the right stack, we can then directly pop them off of the right stack into the `result` array. Write an algorithm that accomplishes this task.



```
T[] result = new T[numberOfEntries];
while (!left.isEmpty()) {

    right.push(left.pop());

} // end while

int index = 0;
```



```
while (!right.isEmpty()) {  
  
    result[index] = right.pop();  
  
    index++;  
  
} // end while  
  
return result;
```

Directed Work

Stack Sort

Pieces of the `StackSort` class already exist and are in `StackSort.java`. Take a look at that code now if you have not done so already. Also before you start, make sure you are familiar with the methods available to you in the `VectorStack` class (check `StackInterface.html`).

Step 1. Compile the classes `StackSort` and `VectorStack`. Run the `main` method in `StackSort`.

Checkpoint: If all has gone well, the program will run. It will create arrays of various sizes and print out the result of `StackSort` method. At the end, the program will ask you to enter an integer value. It will use the value to create and then sort an array of that size. Enter any value. The sorted arrays as reported by the program should all be empty. Our first goal is to get values into a stack and then move them to the result array.

Step 2. Create a new `VectorStack<Integer>` and assign it to `lowerValues`.

Step 3. Create a new `VectorStack<Integer>` and assign it to `upperValues`.

Step 4. Using a loop, scan over the values in the argument array `data` and push them onto the `upperValues` stack.

Step 5. Using a loop, pop all the values from the `upperValues` stack and place them into the array `result`.

Checkpoint: Compile and run the program. Again it should run and ask for a size. Any value will do. This time, you should see results for each of the calls to the StackSort method. The order that values are popped off the stack should be in the reverse order that they were put on the stack. If all has gone well, you should see the values in reverse order in the results array. We will now complete the StackSort method

Step 6. Inside the loop that scans over the `data` array, we need to move the data between the two stacks before we push the value. Refer to the visualization exercise to complete the body of the loop.

Step 7. Before the loop that pops the data values from the `upperValues` stack, we need to move any data values from the `lowerValues` stack. Refer to the visualization exercise to implement this loop.

Checkpoint: Compile and run the program. The output of the StackSort method should be the original arrays in sorted order. If not, debug until the results are correct.

OUTPUTS

QUEUE: SimulationEventQueue Implementation

BASIC TESTING OF THE SIMULATION EVENT QUEUE

TESTING GETSIZE

Checking to see if testEventQueue0 has 0 items
Passed test

Add 1 item and see if the size is now 1

Checking to see if testEventQueue0 has 1 items
Passed test

Add 1 item and see if the size is now 2

Checking to see if testEventQueue0 has 2 items
Passed test

TESTING CLEAR

Add 1 item

Add another item

Checking to see if testEventQueue1 has 0 items after clear
Passed test

TESTING ISEMPY

Checking to see if testEventQueue2 is empty to start
Passed test

Add 1 item and check to see if testEventQueue2 is not empty
Passed test

Add 1 item and check to see if testEventQueue2 is not empty
Passed test

Checking to see if testEventQueue2 is empty after clear
Passed test

TESTING PEEK

Add two events to eventQueue1

Checking to see if testEventQueue1 has the item at time 1 at front
Passed test

Add two events to eventQueue2 in opposite order

Checking to see if testEventQueue2 has the item at time 1 at front
Passed test

TESTING REMOVE

Add two events to eventQueue3

Checking to see if remove gets the item at time 1 at front
Passed test

Checking to see if the size is now 1
Passed test

Checking to see if the current time is now 1.0
Passed test

Checking to see if remove gets the item at time 2 at front
Passed test

Checking to see if the size is now 0
Passed test

Checking to see if the current time is now 2.0
Passed test

Add an item at time 5, then clear

Checking to see if the size is now 0
Passed test

Checking to see if the current time is still 2.0
Passed test

```
TESTING ADD
Add two events to eventQueue4
Remove the first event at time 1
Add a new event at time 1
Checking to see if first item is at time 1 at front
    Passed test

Checking to see if the size is now 2
    Passed test
Checking to see if the current time is now 1.0
    Passed test

Remove both events
Checking to see if the size is now 0
    Passed test
Checking to see if the current time is now 2.0
    Passed test

Try to add an event to eventQueue1 at time 1 (should fail)
Checking to see if the size is still 0
    Passed test
Checking to see if the current time is still 2.0
    Passed test

Try to add an event to eventQueue1 at time 2 (should work)
Checking to see if the size is now 1
    Passed test
Checking to see if the current time is still 2.0
    Passed test

Remove the item at time 2
    Passed test
```

```
Add 8 events to queue (all but one should get added)
Checking to see if the size is now 7
    Passed test
Checking to see if the current time is still 2.0
    Passed test

Remove events from queue one at a time
Did we remove 7 items?
    Passed test
Checking to see if the size is now 1
    Passed test
Checking to see if the current time is still 2.0
    Passed test

Remove the item at time 2
    Passed test

Add 8 events to queue (all but one should get added)
Checking to see if the size is now 7
    Passed test
Checking to see if the current time is still 2.0
    Passed test

Remove events from queue one at a time
Did we remove 7 items?
    Passed test
    Passed test

Remove the item at time 2
    Passed test

Add 8 events to queue (all but one should get added)
Checking to see if the size is now 7
```

```
    Passed test
Checking to see if the current time is still 2.0
    Passed test

Remove events from queue one at a time
Did we remove 7 items?
    Passed test
Checking to see if the size is now 7
    Passed test
Checking to see if the current time is still 2.0
    Passed test

Remove events from queue one at a time
Did we remove 7 items?
    Passed test
Checking to see if the current time is still 2.0
    Passed test

Remove events from queue one at a time
Did we remove 7 items?
    Passed test

Remove events from queue one at a time
Did we remove 7 items?
    Passed test
Remove events from queue one at a time
Did we remove 7 items?
    Passed test
Were the items in the corrent order?
    Passed test
Were the items in the corrent order?
    Passed test
```

```
Did the time update correctly?
    Passed test
Did the time update correctly?
Did the time update correctly?
    Passed test
```

QUEUE: BankLine Implementation (BELOW)

Bank Simulation (Skeleton)

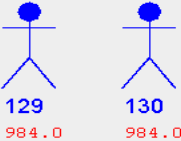
ResetGoPauseStep


Step delay (units of 0.01 second)1

Delay is 0.01 seconds: Warning not recommended

Step 870

Generate the next customer
Next event: Check for next customer





Simulation time is: 988.0
Customers waiting: 2
Average time spent waiting: 4.0
Customers served: 128
Average time spent waiting: 1.7890625

Max customer interval (integer > 0):20
Max service time (integer > 0):6
Max simulation time (integer > 0):1000
Ending at time 1000

STACK: STACK SORT

Checkpoint 1 (BELOW)

```

This program sorts an array of integer values.
Original array is: < >
Sorted array is: < >

Original array is: < 6 >
Sorted array is: < 0 >

Original array is: < 1 0 >
Sorted array is: < 0 0 >

Original array is: < 8480 133 163 4515 6191 4084 9788 4231 2716 1567 >
Sorted array is: < 0 0 0 0 0 0 0 0 0 0 >

Original array is: < 1 3 2 1 6 4 4 8 5 1 5 4 1 7 2 2 3 1 8 5 >
Sorted array is: < 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 >

Please enter the number of values to sort
    It should be an integer value greater than or equal to 1.
3
Original array is: < 6 46 88 >
Sorted array is: < 0 0 0 >

```

Checkpoint 2

```

This program sorts an array of integer values.
Original array is: < >
Sorted array is: < >

Original array is: < 2 >
Sorted array is: < 2 >

Original array is: < 5 4 >
Sorted array is: < 4 5 >

Original array is: < 7015 6083 91 8851 4245 2077 9664 8951 3391 2468 >
Sorted array is: < 2468 3391 8951 9664 2077 4245 8851 91 6083 7015 >

Original array is: < 4 6 6 7 4 2 5 4 8 3 3 2 0 8 0 3 1 5 3 0 >
Sorted array is: < 0 3 5 1 3 0 8 0 2 3 3 8 4 5 2 4 7 6 6 4 >

Please enter the number of values to sort
    It should be an integer value greater than or equal to 1.
3
Original array is: < 86 45 27 >
Sorted array is: < 27 45 86 >

```

Checkpoint 3 (BELOW)