CS2030 (2310) Lab #2

Tags & Categories

Tags:

Categories:

Related Tutorials

Task Content

Lab #2

The server and customer interactions from the previous lab can be broken down into a series of activities or events. Based on whether the server is available or busy serving another customer, there are two event transitions:

- `ARRIVE` → `SERVE`: a new customer arrives, then gets served immediately; or
- `ARRIVE` → `LEAVE`: a new customer arrives, then leaves.

We include another event transition:

- `SERVE` → `DONE`: a server is done serving a customer.

Notice that when a customer arrives and a server is available, the `SERVE` event follows right after `ARRIVE` and the `DONE` event occurs sometime in future.

```
ARRIVE → SERVE → after some time... → DONE
```

An event occurs at a particular time, and each event alters the state of the system and may generate more events. We call this a *discrete event simulator* as states remain unchanged between two events, which allows the simulation to jump from the time of one event to another.

Processing events *in the right way* requires the use of an event priority queue. While Java provides the `PriorityQueue` class which is a mutable collection (like `ArrayList`), we have provided our own `PQ` class that is immutable, as well as an accompanying `Pair` class.

The given programs `PQ.java`, `Pair.java` and `ImList.java` include javadoc comments. To automatically generate HTML documentation from the comments, issue the command:

```
$ javadoc -d doc Pair.java PQ.java ImList.java
```

You may then navigate through the documentation from `allclasses-index.html` found in the `doc` directory.

During the lecture, we have seen how an `ImList` of integers can be sorted by passing an integer `Comparator` to the `sort` method.

```
jshell> List<Integer> list = new ImList<Integer>(List.of(1, 2, 3))
list ==> [1, 2, 3]

jshell> class IntComp implements Comparator<Integer> {
   ...>    public int compare(Integer i, Integer j) {
   ...>       return j - i;
   ...>    }
   ...> }
|  created class IntComp

jshell> list.sort(new IntComp())
list ==> [3, 2, 1]
```

Similarly, a priority queue requires a sort order in which to prioritize elements in the queue. The following shows how a `PQ` of integers can be constructed by passing in an integer `Comparator`.

```
jshell> PQ<Integer> pq = new PQ<Integer>(new IntComp())
pq ==> []
```

We can add elements to the priority queue via the `add` method.

```
jshell> pq.add(1)
$.. ==> [1]

jshell> pq // note that pq is immutable
pq ==> []

jshell> pq = pq.add(1).add(2).add(3)
pq ==> [3, 1, 2]
```

You will also notice that the output of elements in a priority queue is not necessarily in order. What matters most is the values returned from the `poll` method.

```
jshell> pq.poll()
$.. ==> (3, [2, 1])
```

Due to the immutable nature of `PQ`, the `poll` method requires two *values* to be returned, the highest priority element as well as the priority queue after removing the element. These are encapsulated in a generic `Pair` object, and the individual values can be retrieved via the `first` and `second` methods.

```
jshell> Pair<Integer, PQ<Integer>> pr = pq.poll()
pr ==> (3, [2, 1])

jshell> pr.first()
$.. ==> 3

jshell> pq = pr.second()
pq ==> [2, 1]
```

The following shows how the priority queue can be subsequently polled until the queue becomes empty.

```
jshell> pr = pq.poll()
pr ==> (2, [1])

jshell> pq = pr.second()
pq ==> [1]

jshell> pq.isEmpty()
$.. ==> false

jshell> pr = pq.poll()
pr ==> (1, [])

jshell> pq = pr.second()
pq ==> []

jshell> pq.isEmpty()
$.. ==> true
```

We are now ready to process our events using `PQ`. As usual, we create arrival events for all customer arrival and service times. However, rather than processing them right away, we add them into a priority queue that is ordered in terms of events. Events are ordered by earliest occurrence of the event, and in the case of ties, by order in which the customer arrives.

Having included all arrival events in the priority queue, we can begin processing the queue:

- poll an event from the queue
- if necessary, generate the next event using the current state of the server and add into the queue

Repeate the above until the queue is empty. Remember to update the server along the way.

Once again, the `Main` class has been provided to you.

```
import java.util.Scanner;

class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        ImList<Double> arrivalTimes = new ImList<Double>();
        ImList<Double> serviceTimes = new ImList<Double>();

        int numOfServers = sc.nextInt();
        while (sc.hasNextDouble()) {
            arrivalTimes = arrivalTimes.add(sc.nextDouble());
            serviceTimes = serviceTimes.add(sc.nextDouble());
        }

        Simulator sim = new Simulator(numOfServers, arrivalTimes, serviceTimes);
        System.out.println(sim.simulate());
        sc.close();
    }
}
```

Sample runs of the program using the same input as in Lab 1 is given below.

```
$ cat 1.in
3
0.500 1.000
0.600 1.000
0.700 1.000

$ java Main < 1.in
0.500 1 arrives
0.500 1 serves by 1
0.600 2 arrives
0.600 2 serves by 2
0.700 3 arrives
0.700 3 serves by 3
1.500 1 done serving by 1
1.600 2 done serving by 2
1.700 3 done serving by 3
[3 0]

$ cat 2.in
3
0.500 1.000
0.600 1.000
0.700 1.000
1.500 1.000
1.600 2.000
1.700 3.000

$ java Main < 2.in
0.500 1 arrives
0.500 1 serves by 1
0.600 2 arrives
0.600 2 serves by 2
```

```
0.700 3 arrives
0.700 3 serves by 3
1.500 1 done serving by 1
1.500 4 arrives
1.500 4 serves by 1
1.600 2 done serving by 2
1.600 5 arrives
1.600 5 serves by 2
1.700 3 done serving by 3
1.700 6 arrives
1.700 6 serves by 3
2.500 4 done serving by 1
3.600 5 done serving by 2
4.700 6 done serving by 3
[6 0]

$ cat 3.in
2
0.500 1.000
0.600 1.000
0.700 1.000

$ java Main < 3.in
0.500 1 arrives
0.500 1 serves by 1
0.600 2 arrives
0.600 2 serves by 2
0.700 3 arrives
0.700 3 leaves
1.500 1 done serving by 1
1.600 2 done serving by 2
[2 1]

$ cat 4.in
2
0.500 1.000
0.600 1.000
1.500 1.000

$ java Main < 4.in
0.500 1 arrives
0.500 1 serves by 1
0.600 2 arrives
0.600 2 serves by 2
1.500 1 done serving by 1
1.500 3 arrives
1.500 3 serves by 1
1.600 2 done serving by 2
2.500 3 done serving by 1
[3 0]

$ cat 5.in
2
0.500 1.100
0.600 0.900
0.700 0.700
1.500 0.100
1.600 0.200
1.700 0.300

$ java Main < 5.in
0.500 1 arrives
0.500 1 serves by 1
0.600 2 arrives
0.600 2 serves by 2
0.700 3 arrives
0.700 3 leaves
1.500 2 done serving by 2
1.500 4 arrives
1.500 4 serves by 2
1.600 1 done serving by 1
1.600 4 done serving by 2
1.600 5 arrives
1.600 5 serves by 1
1.700 6 arrives
1.700 6 serves by 2
1.800 5 done serving by 1
2.000 6 done serving by 2
[5 1]
```

Some tips...

As the problem increases in complexity, you have to learn how to simplify the original problem so that you can solve them incrementally. For example, using JShell

- start by experimenting with the `PQ` and be familiar with how `add` and `poll` works. You can just follow through the preamble section of this document.
- define a general event class with a timestamp and experiment with the `PQ` of events.
- make the event class a parent and include arrival event as the child; don't worry about the other events. At this point of time, your event `PQ` should just work (adding and polling) with arrival events since arrival events are events.
- include the leave event (since it's simplest) such that when an arrival event gets polled, it should generate a leave event and add to the `PQ`. This leave event should be the next one polled from the `PQ`.
- till now, we did not pay much attention to the status of the servers. Include the serve event (skip done event for the moment) which will update the "state" of the servers. Think of how the "update" should be done in order to make it effect-free. Since a new state of servers has to be created, think of how to include this into `PQ` processing which ultimately drives the simulation.
- if you are able to reach this point, you will realize that the output of your program will be no different from the previous lab. Now it is a matter of including the done event.

Although you could have gotten the correct output without using events and/or the priority queue, do note that there will be more events later, so it is always good to prepare early.