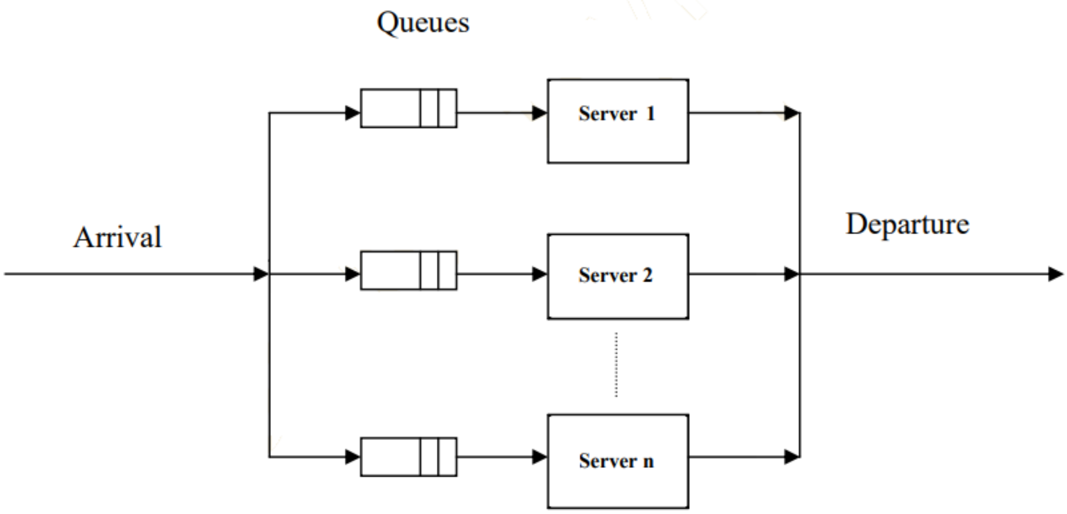
Lab #3

In the previous lab, we designed a discrete event simulator to simulate the changes in the state of a system across time, with each transition from one state of the system to another triggered via an event. Simulation statistics are also typically tracked to measure the performance of the system.

Such a system can be used to study many complex real-world systems such as queuing to order food at a fast-food restaurant. The following illustrates a multi-queue-multi-server system:



- There are n servers with its own queue.
- Each server can serve one customer at a time.
- Each customer has a service time (time taken to serve the customer).
- When a customer arrives (`ARRIVE` event):
  - the servers are scanned from `1` to `n` to find the first one that is idle (not serving any customer). This server starts serving the customer immediately (`SERVE` event).
  - the server is done (`DONE` event) after serving the customer.
  - if all servers are serving customers, then the customer that just arrived scans the queues from `1` to `n`, joins the first queue that is not full (not necessarily the shortest) and waits in the queue (`WAIT` event). Note that a customer that chooses to queue joins at the end of the queue.
  - if all servers are serving customers and all queues are full of waiting customers, then a new customer that arrives, just leaves (`LEAVE` event).
- If there is no waiting customer, then the server becomes idle again.

Notice from the above description that there are five events in the system, namely: `ARRIVE`, `SERVE`, `WAIT`, `LEAVE` and `DONE`. For each customer, these are the only possible event transitions:

- `ARRIVE` → `SERVE` → `DONE`
- `ARRIVE` → `WAIT` → `SERVE` → `DONE`
- `ARRIVE` → `LEAVE`

In essence, an event is tied to one customer. Depending on the current state of the system, triggering an event will result in the next state of the system, and possibly the next event. Events are also processed via a queue. At the start of the simulation, the queue only contains the customer arrival events. With every simultation time step, an event is retrieved from the queue to be processed, and any resulting event added to the queue. This process is repeated until there are no events left in the queue.

As an example, given an input of one server with a maximum queue length of `1` (i.e. only one waiting customer is allowed), followed by the arrival and *assuming that service times is `1.0`*,

```
1 1
0.500
```

```
0.600
0.700
```

the entire similation run results in the following output:

```
0.500 1 arrives
0.500 1 serves by 1
0.600 2 arrives
0.600 2 waits at 1
0.700 3 arrives
0.700 3 leaves
1.500 1 done serving by 1
1.500 2 serves by 1
2.500 2 done serving by 1
```

Finally, statistics of the system that we need to keep track of are:

1. the average waiting time for customers who have been served;
2. the number of customers served;
3. the number of customers who left without being served.

In our example, the end-of-simulation statistics are respectively, `[0.450 2 1]`.

On-Demand Service Time

Moreover, it is unrealistic that service time has to be pre-determined before the customer is being served. In the above example, customer `3` would not be served and hence a service time need not be specified. To faciliate on-demand data (or delayed data), we make use of the `Supplier` interface which specifies an abstract method `get()` to be defined by its implementation class.

As an example using JShell, the following `Supplier` implementation can be defined that generates a service time (say, a constant `1.0`)

```
jshell> class DefaultServiceTime implements Supplier<Double> {
   ...>    public Double get() {
   ...>       System.out.println("generating service time..."); // output for tracing purposes
   ...>       return generateServiceTime();
   ...>    }
   ...> }
|  created class DefaultServiceTime
```

`DefaultServiceTime` can be used in a simple `Customer` class as follows:

```
jshell> class Customer {
   ...>    private final Supplier<Double> serviceTime;
   ...>    Customer(Supplier<Double> serviceTime) {
   ...>       this.serviceTime = serviceTime;
   ...>    }
   ...>    double getServiceTime() {
   ...>       return this.serviceTime.get();
   ...>    }
   ...> }
|  created class Customer

jshell> DefaultServiceTime serviceTime = new DefaultServiceTime()
serviceTime ==> DefaultServiceTime@52cc8049

jshell> Customer adele = new Customer(serviceTime) // adele is created first
adele ==> Customer@1753acfe

jshell> Customer billie = new Customer(serviceTime)
billie ==> Customer@2a2d45ba
```

```
jshell> Customer charlie = new Customer(serviceTime)
charlie ==> Customer@34c45dca

jshell> billie.getServiceTime() // billie is the first to get a service time
generating service time...
$.. ==> 1.0

jshell> adele.getServiceTime()
generating service time...
$.. ==> 1.0
```

Notice that even though `adele` is being created first before `billie`, the latter invokes `getServiceTime` (and hence `generateServiceTime`) first. Moreover, `charlie` is created but no service time is generated.

Also notice that rather than having `Customer` depend on `DefaultServiceTime` directly, it is dependent on the `Supplier` interface instead. Doing this allows the program to accept different types of `Supplier<Double>` implementations. In this lab, you can make use of `DefaultServiceTime` since the sample run below assumes a default service time of `1.0`.

Task

The utility classes `Pair`, `PQ` and `ImList` have been provided to you. Moreover, the `Main` client with `DefaultServiceTime` are provided for you.

```
import java.util.Scanner;
import java.util.function.Supplier;

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

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

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

Sample runs of the program is given below. All of them assumes a default service time of `1.0`. You should create your own input files using `vim`. You may also create other implementations of `Supplier<Double>` to test out different service times. Note that your program will be tested against test cases where the service times could be different when serving different customers.

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

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

```
0.600 2 arrives
0.600 2 waits at 1
0.700 3 arrives
0.700 3 leaves
1.500 1 done serving by 1
1.500 2 serves by 1
2.500 2 done serving by 1
[0.450 2 1]

$ cat 2.in
1 1
0.500
0.600
0.700
1.500
1.600
1.700

$ java Main < 2.in
0.500 1 arrives
0.500 1 serves by 1
0.600 2 arrives
0.600 2 waits at 1
0.700 3 arrives
0.700 3 leaves
1.500 1 done serving by 1
1.500 2 serves by 1
1.500 4 arrives
1.500 4 waits at 1
1.600 5 arrives
1.600 5 leaves
1.700 6 arrives
1.700 6 leaves
2.500 2 done serving by 1
2.500 4 serves by 1
3.500 4 done serving by 1
[0.633 3 3]

$ cat 3.in
2 1
0.500
0.600
0.700
1.500
1.600
1.700

$ 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 waits at 1
1.500 1 done serving by 1
1.500 3 serves by 1
1.500 4 arrives
```

```
1.500 4 waits at 1
1.600 2 done serving by 2
1.600 5 arrives
1.600 5 serves by 2
1.700 6 arrives
1.700 6 waits at 2
2.500 3 done serving by 1
2.500 4 serves by 1
2.600 5 done serving by 2
2.600 6 serves by 2
3.500 4 done serving by 1
3.600 6 done serving by 2
[0.450 6 0]

$ cat 4.in
2 2
0.500
0.600
0.700
1.500
1.600
1.700

$ java Main < 4.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 waits at 1
1.500 1 done serving by 1
1.500 3 serves by 1
1.500 4 arrives
1.500 4 waits at 1
1.600 2 done serving by 2
1.600 5 arrives
1.600 5 serves by 2
1.700 6 arrives
1.700 6 waits at 1
2.500 3 done serving by 1
2.500 4 serves by 1
2.600 5 done serving by 2
3.500 4 done serving by 1
3.500 6 serves by 1
4.500 6 done serving by 1
[0.600 6 0]
```