CMPS 12B
Introduction to Data Structures
Winter 2009

Programming Assignment 3

Due Monday February 23, 10:00 pm

In this project you will implement a Queue ADT based on a linked list data structure. You will use your Queue ADT to simulate processing of jobs, where there are fewer processors than jobs and therefore some jobs may have to wait in a queue. Think of shoppers at a grocery store standing in lines at the check-out stands, or some manufacturing process where unfinished products wait their turn for one or more machines which perform some specific task. In these metaphors a job is a basket of goods to be purchased (or some collection of raw materials) and a processor is a check-out stand (or some machine which processes the raw materials.) Abstractly a job is an encapsulation of three quantities: *arrival time*, *duration*, and *finish time*. The arrival time is the time at which the job becomes available for processing. This is analogous to the time at which a shopper reaches check-out stand area. If there is a free Processor, the job may begin its "work", whatever that may be. If not, it must wait in a *processor queue*. The duration of a job is the amount of processor time it will consume once it reaches a processor. This is analogous to the amount of goods in the shopper's basket. Both arrival time and duration are intrinsic to the job and are known ahead of time, whereas the finish time is only known when the job reaches the head of a processor queue. The finish time can then be calculated as `start_time + duration`. Before a job is underway its finish time will be considered undefined. Once a given job's finish time is known, we can calculate the amount of time spent waiting in line, not counting processing time. Thus `wait_time = finish_time – arrival_time – duration`. Note that the simulation begins at time `0`, and all subsequent times are integer multiples of some unspecified unit.

The goal of this simulation is, given some batch of `m` jobs with specific arrival times and durations, determine three quantities: the total wait time (i.e. the total time spent by all `m` jobs waiting in queues), the maximum wait time, and the average wait time. Furthermore, determine these quantities for `n` processors, where `n` ranges from `1` to `m-1`. Observe that there is no point in considering `m` or more processors, since then the wait time for each job is necessarily `0`. The `n` processors in this simulation will be represented by an array of `n` processor queues. The job (if any) at the head of a processor queue is considered to be underway, while those jobs behind the head are considered to be waiting. When a job arrives it is assigned to a processor queue of minimum length. If there is more than one minimum length processor queue, the job will be assigned to the one whose array index is smallest. It is recommended that your simulation maintain one or more additional queues for storage of jobs which have not yet arrived, and for jobs which have been completed.

Your main program for this project will be contained in a file called `Simulation.java`. The Simulation class will contain a main method which will be the entry point for program execution. The Simulation class however is not to be considered a stand alone ADT. It is the client of two ADTs, one of which (Job) will be provided on the webpage, and the other (Queue) will be created by you. Your program will take one command line argument giving the name of an input file, and will write to two output files whose names are the name of the input file followed by the suffixes `.rpt` and `.trc` respectively. The `.rpt` (report) file will contain the results total wait, maximum wait, and average wait, for `n` processors where `n` ranges from `1` to `m-1`. The `.trc` (trace) file will contain a detailed trace of the state of the simulation at the points in time when either an arrival or finishing event occur. All `.class` files for this project will be placed in an executable jar file called `Simulation`. Thus your program will be run by doing

```
% Simulation input_file
```

and the result will be the creation of two files in your working directory called `input_file.rpt` and `input_file.trc`. During the initial design and construction phases of your project it may be helpful to send the contents of the trace file to `stdout` for diagnostic purposes. A file called `SimulationStub.java` is included on the website in the examples section. It contains a few lines of code and some high level pseudo-code which you may use as a starting point for this project. A makefile is also provided on the website which you may alter as you see fit.

Input and Output File Formats

The first line of an input file will contain one integer giving the number `m` of jobs to be run. The next `m` lines will each contain two integers separated by a space. These integers will give the arrival time and duration of a job. The jobs will appear in the input file ordered by arrival times, with the earliest job first. Here is a sample input file called `ex1` representing a batch of 3 jobs:

```
3
2 2
3 4
5 6
```

As usual, you may assume that your project will be tested only on correctly formatted input, i.e. there is no need to consider what to do if the jobs are not ordered by finish times, or if the file does not contain at least `m+1` lines, etc. The report file corresponding to the above input file follows.

```
Report file: ex1.rpt
3 Jobs:
(2, 2, *) (3, 4, *) (5, 6, *)

***********************************************************
1 processor: totalWait=4, maxWait=3, averageWait=1.33
2 processors: totalWait=0, maxWait=0, averageWait=0.00
```

In the corresponding trace file below, the process queues are labeled `1` through `n`. The label `0` is reserved for a *storage queue*, which initially contains the jobs sorted by arrival time. Each job is represented as a triple: `(arrival, duration, finish)`. An undefined finish time is represented here as `*`. As jobs are finished, they are placed at the back of the storage queue, so that when the simulation is complete, the storage queue contains all jobs sorted by finish times. Once the simulation is complete for a given number of processors, the finish times are reset to undefined, and the same jobs are simulated again with one more processor.

```
Trace file: ex1.trc
3 Jobs:
(2, 2, *) (3, 4, *) (5, 6, *)

*****************************
1 processor:
*****************************
time=0
0: (2, 2, *) (3, 4, *) (5, 6, *)
1:

time=2
0: (3, 4, *) (5, 6, *)
1: (2, 2, 4)
```

```
time=3
0: (5, 6, *)
1: (2, 2, 4) (3, 4, *)

time=4
0: (5, 6, *) (2, 2, 4)
1: (3, 4, 8)

time=5
0: (2, 2, 4)
1: (3, 4, 8) (5, 6, *)

time=8
0: (2, 2, 4) (3, 4, 8)
1: (5, 6, 14)

time=14
0: (2, 2, 4) (3, 4, 8) (5, 6, 14)
1:

*****************************
2 processors:
*****************************
time=0
0: (2, 2, *) (3, 4, *) (5, 6, *)
1:
2:

time=2
0: (3, 4, *) (5, 6, *)
1: (2, 2, 4)
2:

time=3
0: (5, 6, *)
1: (2, 2, 4)
2: (3, 4, 7)

time=4
0: (5, 6, *) (2, 2, 4)
1:
2: (3, 4, 7)

time=5
0: (2, 2, 4)
1: (5, 6, 11)
2: (3, 4, 7)

time=7
0: (2, 2, 4) (3, 4, 7)
1: (5, 6, 11)
2:

time=11
0: (2, 2, 4) (3, 4, 7) (5, 6, 11)
1:
2:
```

Note that the above trace only prints the state of the simulation when it changes, i.e. when either an arrival or finish event occur. It is also possible for two or more jobs to have the same arrival time. Such jobs will necessarily appear next to each other in the input file. In such a case, the jobs are assigned to a processor queue in the order in which they appear in the input file. A number of other examples will posted on the webpage, including one in which distinct jobs have equal arrival times.

The format of the trace file seems to imply that there is one storage queue which contains both those jobs which have not yet arrived, and those which are complete. You may choose to implement the simulation with two different queues for these purposes. Whatever you choose as your implementation, the output to the trace and report files must look exactly as above. In addition to one or more storage queues, it will be helpful to maintain a *backup queue* which stores the jobs in the order in which they originally appeared in the input file. This way you can simply copy the backup queue to the storage queue, resetting finish times as you go, then run the next simulation with one more processor.

The Queue ADT

Your Queue ADT will implement the following interface file.

```
// QueueInterface.java
public interface QueueInterface{

   // isEmpty
   // pre: none
   // post: returns true if this Queue is empty, false otherwise
   public boolean isEmpty();

   // length
   // pre: none
   // post: returns the length of this Queue.
   public int length();

   // enqueue
   // adds newItem to back of this Queue
   // pre: none
   // post: !isEmpty()
   public void enqueue(Object newItem);

   // dequeue
   // deletes and returns item from front of this Queue
   // pre: !isEmpty()
   // post: this Queue will have one fewer element
   public Object dequeue() throws QueueEmptyException;

   // peek
   // pre: !isEmpty()
   // post: returns item at front of Queue
   public Object peek() throws QueueEmptyException;

   // dequeueAll
   // sets this Queue to the empty state
   // pre: !isEmpty()
   // post: isEmpty()
   public void dequeueAll() throws QueueEmptyException;

   // toString
   // overrides Object's toString() method
   public String toString();
}
```

There are four main differences between the Queue ADT for this project and the IntegerQueue ADT posted on the website. First, your Queue ADT will be based on an underlying linked list data structure, not an array. Second, the queue elements are Objects not integers. This means that the item field in your inner Node class must be of type `Object`, and several parameter lists and return types are now `Object` instead of `int`. Third, there is an additional access function called `length()` which merely returns the length of a Queue object. This is necessary so that your simulation can determine which processor queue is shortest, and therefore where a newly arrived job should be assigned. Fourth, the `toString()` function has been included in the implementation file. This function should assemble a `String` consisting of the Objects in the `Queue` separated by spaces. It should do this by explicitly calling the `toString()` function of whatever type of Object is stored in the Queue. Your Queue ADT will be comprised of three files. File `QueueInterface.java` above is provided on the webpage, and must not be altered. You will write the files `QueueEmptyException.java`, and `Queue.java`. As usual, you should write a test client for your Queue ADT, called `QueueTest.java` for independent testing of it's operations.

The Job ADT

The file `Job.java` below contains the definition of the Job class, and will also be posted on the website.

```
// Job.java
import java.io.*;

public class Job{
   public static final int UNDEF = -1;
   private int arrival;
   private int duration;
   private int finish;

   // default constructor
   public Job(int arrival, int duration){
      this.arrival = arrival;
      this.duration = duration;
      this.finish = UNDEF;
   }

   // access functions
   public int getArrival(){return arrival;}
   public int getDuration(){return duration;}
   public int getFinish(){return finish;}
   public int getWaitTime(){
      if( finish==UNDEF ){
         System.err.println("Job: getWaitTime(): undefined finish time");
         System.exit(1);
      }
      return finish-duration-arrival;
   }

   // manipulation procedures
   public void computeFinishTime(int timeNow){finish = timeNow + duration;}
   public void resetFinishTime(){finish = UNDEF;}

   // toString(): overrides Object's toString() method
   public String toString(){
      return "("+arrival+", "
                +duration+", "
                +(finish==UNDEF?"*":String.valueOf(finish))+")";
   }
}
```

Observe that this is a very simple ADT with a number of self explanatory access functions and manipulation procedures. In particular the methods `getWaitTime()`, `computeFinishTime()`, and `resetFinishTime()` automate some basic tasks you will need to perform the simulation. Observe also that function `toString()` assembles a `String` representation of a Job which is consistent with the output file formats above. Thus if you write the `toString()` function in your Queue class correctly, you can print a properly formatted Queue to `stdout` by just doing: `System.out.println(myQueue)`. To print to one of your output files, say the report file, do

```
PrintWriter report = new PrintWriter(new FileWriter(args[0]+".rpt"));
```

then

```
report.println(myQueue);
```

Notice that there are no interface or exception files associated with the Job ADT. As previously mentioned the file `Job.java` is posted on the webpage. It should not be altered in any way.

What to turn in

You may alter the provided Makefile to include submit and test utilities, or alter it in any other way you see fit, as long as it creates an executable jar file called `Simulation`. To repeat however, you are not to alter the files `QueueInterface.java` or `Job.java`. Submit the files:

```
README
Simulation.java
QueueInterface.java
QueueEmptyException.java
Queue.java
QueueTest.java
Job.java
makefile
```

to the assignment name `pa3`. Use peek or go to

```
/afs/cats.ucsc.edu/class/cmps012b-pt.w09/pa3
```

to check that all files were successfully submitted. As always start early and ask plenty of questions.