1. Introduction

In this lab activity, we will work with two implementations of the Queue ADT as well as an application of queues. The specification for the Queue ADT to be used here is as given in the textbook and in lectures. It is specified as follows:

```
public interface Queue<E> {
   /** Accessor Method. Returns the size of the current instance **/
   int size();

   /** Accessor Method. Returns true if the current instance is empty; false, if not.
   **/
   boolean isEmpty();

   /** Accessor Method.
       Accesses the first element in the current instance of the queue.
       The affected element is the one that has been in the queue
       for the longest time among all its current elements.
       Returns reference to the element being accessed.
       Throws EmptyQueueException if the queue is empty.
   **/
   E getFront();

   /** Mutator Method.
       Adds a new element to the queue.
   **/
   void enqueue(E element);

   /** Mutator Method.
       Similar to the getFron() method, but this time, the queue is
       altered since the accessed element is also removed from
       the queue.
       Throws EmptyQueueException if the queue is empty.
   **/
   E dequeue();
}
```

2. Implementation

As part of this activity you will write the Java classes for the two implementations of the queue that were discussed in lectures and in the textbook; these are:

- The array-based implementation: based on the array being used in a circular mode. Your implementation should provide unlimited capacity. One full implementation is included at the end of this document. Your task will be to verify that it works as specified (see Exercise 1.B at the end of document).
- The linked-list implementation: based on a single linked list with internal references to the first node and the last node of that linked list.

A partial implementation of the one based on the singly linked list is as follows (or as discussed in lectures):

```
/**
   A partial implementation of the Queue using a singly linked list with references
   to the first and to the last node.
**/
public class LinkedQueue<E> implements Queue<E> {
       // inner class for nodes in singly linked lists
       private static class Node<E> {   .
              private E element;
              private Node<E> next;
              ... other methods of class Node
       } // END CLASS NODE

       private Node<E> first, last;   // references to first and last node
       private int size;

       public LinkedQueue() {         // initializes instance as empty queue
              first = last = null;
              size = 0;
       }
       public int size() {
              return size;
       }
       public boolean isEmpty() {
              return size == 0;
       }
       public E getFront() {
              if (isEmpty())
                 throw new EmptyQueueException("Queue is empty");
              return first.getElement();
       }
       public E dequeue() {
              E etr = getFront();     // throws the ESE if needed.
              ...
       }
       public void enqueue(E e) {
              if (size == 0)
                     first = last = new Node<>(e);
              else {
                     ...
              }
              size++;
       }
}
```

3. An Application of the Queue

We now discuss a simple simulation system in which the queue plays a central role. Here, we work on a simple system to emulate a batch processing system. It executes jobs that arrive to the system following the scheme to be explained shortly. The system includes a queue (the system queue) in which jobs are placed to wait for service (execution). At any instance of time, the first job in the queue (if not empty) is in execution (or being served by the system). Only one job can be served at a time. New jobs can arrive to the system at the beginning of a unit of time. Times for this system are measured as integer values 0, 1, 2, 3, ...; where 0 is when the system starts. Also, time unit i

corresponds to the time interval [i, i+1). So, if a job arrives at time interval x, then at the beginning of the time interval y ($y \geq x$) that job (if it is still in the system) would have been in the system for y-x units of time. When a job arrives, it will immediately be placed in the system queue, and, if it is the only job there, it will start executing immediately without any further delay. In other words, the time the system consumes in placing a new job in the queue is negligible. Moreover, when a new job arrives, if the queue is not empty at that moment, the first job in the queue will continue executing one more unit of time with no delay caused by the new arrival and its placement at the end of the queue.

The system will serve as many jobs as possible using a time-shared approach. Under this approach, each job is served one unit of time. If not finished by then, the system puts it in the back of a queue and continues serving the next job in the queue. A job in the queue will be served for one unit of time when it reaches the first position in the queue as per the above scheme. Once completed, the job exits the system with no further delays. The above circular approach is repeated for each job as needed until the job execution is finished.

We assume that to switch from one job to another, including, if needed, to place a job back at the end of the queue, does not take any time at all (in reality, as you may already suspect, this time is not zero). For simplicity, we also assume that at most one job arrives at a particular unit of time. In the case that the queue is not empty when a new arrival occurs, the job that was executing at the previous time unit will have priority over the new arrival if needed to be placed at the end of the queue because it has not finished execution.

The above approach describes the First-Come-First-Served (FCFS) scheduling policy - once in the queue, jobs are served in the order in which they were enqueued. At any particular time unit, the job in front of the queue is the one being served. The above approach is a simplified version of Round-robin scheduling.

For this activity (**Exercise 2** at the end), the system reads a text file in which is line contains two integer values that are separated by a comma. For each line, the two numbers there correspond to an arriving job. On each such pair, the first value represents a unit of time corresponding to the *arrival time* of the particular job (job arrives at the beginning of that unit). The second number states how many units of time it needs to execute before completion (*service time*). The data about the jobs to be processed is read once; that is, no new jobs are generated by the system in addition to those in the input data. For more details about the input, see Section 3.2.

3.1. An Example

As an example, consider the following input:

Process ID	Arrival Time	Service Time
1	1	6
2	3	2
3	4	3
4	6	2

In the first column, the table shows the ID number that the system assigns to each arriving job (PID). It is not part of the input data, but it is automatically assigned by the system as per the arrival order. If the input file contains n jobs, the PID values will be 1, 2, 3, ..., n.

Time	Job	Content of	Events Happening at the beginning of the Time Unit t

Unit t	Served	Queue	
0	-	∅	Nothing
1	1	(1, 6)	Job 1 arrives and starts execution
2	1	(1, 5)	Job 1 continues execution
3	1	(1, 4), (2, 2)	Job 1 continues execution and job 2 arrives
4	2	(2, 2), (1, 3), (3, 3)	Job 1 goes to the end of the queue, job 2 starts execution, and job 3 arrives
5	1	(1, 3), (3, 3), (2, 1)	Job 2 goes to the end of the queue and job 1 continues execution
6	3	(3, 3), (2, 1), (1, 2), (4, 2)	Job 1 goes to the end of the queue, job 3 starts execution, and job 4 arrives
7	2	(2, 1), (1, 2), (4, 2), (3, 2)	Job 3 goes to the end of the queue and job 2 continues execution
8	1	(1, 2), (4, 2), (3, 2)	Job 2 finishes and job 1 continues execution
9	4	(4, 2), (3, 2), (1, 1)	Job 1 goes to the end of the queue and job 4 continues execution
10	3	(3, 2), (1, 1), (4, 1)	Job 4 goes to the end of the queue and job 3 continues execution
11	1	(1, 1), (4, 1), (3, 1)	Job 3 goes to the end of the queue and job 1 continues execution
12	4	(4, 1), (3, 1)	Job 1 finishes and job 4 continues execution
13	3	(3, 1)	Job 4 finishes and job 3 continues execution
14	-	∅	Job 3 finishes execution and system is empty

In the above table, a pair (i, t) in the queue corresponds to process having PID=i and which still needs t more units of time to complete execution. In the above scenario, for each process we have:

Process	Arrival Time	Departure Time	Total Time[1]
1	1	12	11
2	3	8	5
3	4	14	10
4	6	13	7

Then the average time that a process spends in the system is: (11+5+10+7)/4. This is what we want in this simple simulation.

3.2. Input Data

[1] The time that a process spends in the system: (departure time) - (arrival time).

The input for the program that you need to write is a text file that contains one line for each job. That file is to be named as "`input.txt`", and must be part of the directory from where the program is executed. Each line contains the following two values separated by a comma character: the *arrival time* of the corresponding job and the *service time* (time that the job needs to complete its task). The input file is assumed to be sorted by increasing order of the arrival times. For the example given, the input file would contain the following four lines:

1, 6
3, 2
4, 3
6, 2

Your Java program will read such file and process its data as per the above discussion. The input file is assumed to be correct, hence your program does not have to validate it; but keep in mind that the execution may terminate with an error if the input file is not valid.

3.3. The Processing System

One approach to achieve the desired behavior is as follows. NOTE: this can be made more efficient, and you should think about it, although that is not required for this activity.

1. **Read** input data and place relevant data for each job in a queue, say `inputQueue`.
2. **Initialize** another queue of jobs as an empty queue, say processingQueue.
3. **Create** an empty list of terminated jobs, say `terminatedJobs`.
4. **Set** time t=0
5. **while** at least one of the above two queues is not empty do:
    a. **if** `processingQueue` is not empty
        i. Service job at the front of that queue for 1 unit of time
        ii. **if** remaining time of that job is 0, do the following:
            1. **set** its departure time to t
            2. **remove** that job from `processingQueue`
            3. **place** that job in list `terminatedJobs`
        iii. **else** // for the if in b.
            1. move first job in `processingQueue` to the end of the queue
    b. **if** `inputQueue` is not empty and first job has arrival time t, remove that job from `inputQueue` and place corresponding processing job at the end of `processingQueue`
    c. **Increment** t by 1 unit
6. **Compute** final statistics as per the specifications given

3.4. Generated Output

The output for this activity is simple, just the final average time that jobs take in the system Once the input data is read by the program, after simulating the described system, the output produced is as follows:

```
Average Time in System is: ....
```

Consider the following cases as input on different instances; for each, the expected output is shown.

Input Data	Output will be:

1, 6 3, 2 4, 3 6, 2	Average Time in System is: 8.25
1, 6 3, 2 4, 3 6, 2 7, 4 8, 9 9, 1	Average Time in System is: 11.71
0, 1 1, 1 2, 1 3, 1 4, 1 5, 1 6, 1 7, 1	Average Time in System is: 1.0
0, 2 1, 2 2, 2 3, 2 4, 2 5, 2 6, 2	Average Time in System is: 6.0
0, 1 1, 2 2, 3 3, 4 4, 5 5, 6 6, 7	Average Time in System is: 12.43
1, 50 2, 1 60, 1	Average Time in System is: 18.0

4. Exercises

Exercise 1. Work on the following two implementations of the Queue ADT:

A. Class `LinkedQueue` - implementation based on the singly linked list. Complete the implementation of class `LinkedQueue` partially given in an earlier section of this document.

B. Class `ArrayQueue` - implementation based on the **circular array.** In this case, the implementation must have a fixed initial capacity (the internal array has that initial length); and, whenever needed in operation `enqueue`, the capacity should increase by doubling the current one. The following is a partial implementation for this. You should study it and finish its implementation to have a full working implementation of the `Queue` ADT. Also, write the final Java class for it to be tested. You should be prepared to answer questions that your lab instructor may ask in order for you to show your understanding of this implementation.

```
public class ArrayQueue<E> implements Queue<E> {
	private final static int INITCAP = 10;
	private E[] elements;
	private int first, size;
	public ArrayQueue() {
		elements = (E[]) new Object[INITCAP];
		first = 0;
		size = 0;
	}
	public int size() {
		return size;
	}

	public boolean isEmpty() {
		return size == 0;
	}

	public E getFront() {
		if (isEmpty())
		   throw new EmptyQueueException("Queue is empty");
		return elements[first];
	}

	public E dequeue() {
		E etr = getFront();

		elements[first] = null;

		... adjust whatever needs to be adjusted ...

		return etr;
	}

	public void enqueue(E e) {
		if (size == elements.length)   // check capacity, double it if needed
			doubleCapacity();

		... finish the implementation of this method ...
	}

	private void doubleCapacity() {
		// doubles the capacity of the internal array

		... finish the implementation of this method ...
```

```
        }
}
```

For these two implementations of the `Queue`, you should provide appropriate testers to verify that your implementation works correctly.

Exercise 2. Write a Java program for the simulation system following the scheme suggested in Section 3.3. Test your program using the input data included in the example case shown in this document (Section 3.4). You must show the output results to your lab instructor. The instructor may ask you to try with different input data. Your implementation must work independently of the type of queue that is used, either `LinkedQueue` or `ArrayQueue`, or mixing them (one type for the input queue and another type for the processing queue), with no other changes required to the program.

The following class will be useful to represent jobs in your solution.

```
/**
 * A Job for this lab activity.
 * @author pedroirivera-vega
 *
 */
public class Job {
   private int pid;            // id of this job
   private int arrivalTime;    // arrival time of this job
   private int remainingTime;  // remaining service time for this job
   private int departureTime;  // time when the service for this job is completed
   public Job(int id, int at, int rt) {
       pid = id;
       arrivalTime = at;
       remainingTime = rt;
   }
   public int getDepartureTime() {
       return departureTime;
   }
   public void setDepartureTime(int departureTime) {
       this.departureTime = departureTime;
   }
   public int getPid() {
       return pid;
   }
   public int getArrivalTime() {
       return arrivalTime;
   }
   public int getRemainingTime() {
       return remainingTime;
   }

   /**
    * Registers an update of service received by this job.
    * @param q the time of the service being registered.
    */
   public void isServed(int q) {
```

```
      if (q < 0) return;   // only register positive value of q
         remainingTime -= q;
   }

   /**
    * Generates a string that describes this job; useful for printing
    * information about the job.
    */
   public String toString() {
      return "PID = " + pid +
             "  Arrival Time = " + arrivalTime +
             "  Remaining Time = " + remainingTime +
             "  Departure Time = " + departureTime;
   }
}
```

5. End