CSE 15
Introduction to Data Structures
Programming Assignment 4

In this project you will implement an IntegerQueue ADT in C based on a linked list data structure. You will use your IntegerQueue to simulate a set of jobs performed by a set of processors, where there are more jobs than processors, and therefore some jobs may have to wait in a queue. Think of shoppers at a grocery store waiting in line at check-out stands. In this metaphor, a job is a basket of goods to be purchased and a processor is a clerk at a check-out stand ringing up the purchase. Abstractly, a job is an encapsulation of three quantities: *arrival time*, *duration*, and *finish time*. The arrival time is the time at which a job becomes available for processing. This is analogous to the time at which a shopper reaches the check-out stand area. If there is a free Processor, that job's "work" may begin. 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 quantity 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 before the simulation begins. The finish time, by contrast, is only known when the job reaches the head of a processor queue. The finish time can then be calculated as

```
finish_time = 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 its time spent waiting in line, not counting processing time. Thus

```
wait_time = finish_time – arrival_time – duration.
```

In this simulation, time is considered a discrete quantity, starting at 0 and assuming only non-negative integer values. Note that time 0 is reserved for initialization, and all job arrival times will be greater than or equal to 1.

The goal of the simulation will be as follows: given a batch of m jobs, each with specific arrival times and durations, determine (1) the total wait time (the total time spent by all m jobs waiting in queues), (2) the maximum wait time (the longest wait any of the m jobs endured) and (3) the average wait time over all m jobs. 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. (If there are as many or more checkers than shoppers, no one ever has to wait.) The n processors in this simulation will be represented by an array of n processor queues. The job (if any) at the front of a processor queue is considered to be underway, while the jobs (if any) behind the front job are waiting. When a job arrives it is assigned to a processor queue of minimum length. (Shoppers naturally go to the shortest line.) If there is more than one minimum length processor queue, the job will be assigned to the one whose array index is smallest. (Shoppers go to the closest among several shortest lines.) It is recommended that your simulation maintain one or more additional queues for storage of those jobs that have not yet arrived, and for those jobs that have been completed.

Your function `main()` for this project will be contained in a file called Simulation.c. The Simulation module is not to be considered a stand-alone ADT however. It is the client of two ADTs, one of which (Job) will be provided on the webpage, and the other (IntegerQueue) 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 those points in time when either an arrival or finish event occur. Your Makefile for this project will create an executable file called Simulation. Thus your program will be run by doing: `$ Simulation input_file`. The result will be the creation of two files in your current 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.c is included in the examples section of the course website. It contains a helper function and some high level pseudo-code that you may use as a starting point. 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 arrival times, or if the file does not contain at least m 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 processor queues are labeled 1 through n. The label 0 is reserved for a *storage queue*, which initially contains all jobs, sorted by arrival time. Each job is represented as a triple: `(arrival, duration, finish)`. An undefined finish time is represented as `*`. As jobs are finished, they are placed at the back of the storage queue. If more than one job finishes at the same time, they are placed in the storage queue in order of their appearance in the queue array, i.e. minimum index first. When the simulation is complete for a given number of processors, the storage queue will contain all jobs sorted by finish times. At this point the finish times will be reset to undefined, all jobs will again be arranged in the storage queue sorted by arrival times, and 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 (or both) occur. It is 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 this case, the jobs are assigned to a processor queue in the order in which they appear in the input file. It is also possible that some jobs finish at the same time, or finish at the same time as some arrival event. In such cases, do the following steps in order.

- If any jobs finish at the same time, complete their finish events in the order that they appear in the queue array, i.e. lowest index first.

- If any jobs arrive at the same time, complete their arrival events by placing them at the back of the processor queue that is (1) shortest, and if there are multiple shortest queues then (2) choose the one with lowest index in the queue array.

Always complete the finish events first, then complete the arrival events. A number of other examples are posted on the class webpage, including cases in which distinct jobs have the same arrival times, and in which some finish and arrival events occur simultaneously. The format of the trace file seems to imply that there is one storage queue that contains both those jobs which have not yet arrived, and those that are complete. You may choose to implement the simulation with two different queues for these purposes. However you choose to implement the simulation, the output to the trace and report files must look exactly as above.

Notice that your IntegerQueue ADT is expected to manipulate jobs, which are not integers but instances of the separate Job ADT, provided on the webpage. How is this to be accomplished? Function `main()` in your client Simulation.c will read the first line of the input file to determine the number m of jobs. It will then allocate an array of Job references of length m from heap memory. Each subsequent line will contain two integers to be parsed, then passed to the Job constructor. (A helper function is included in SimulationStub.c for this purpose, which you may use or discard as you like.) The resulting Job reference will then be placed in the aforementioned array. Thus, each job is associated with a specific integer in the range 0 to $m - 1$, namely is its index in this array. It is these indices that will be stored and manipulated by your IntegerQueue ADT. You can call this array of jobs anything you like, but we refer to it here as the *backup array*. It will continue to store all Jobs in their original input file order, which is necessary to set up each new simulation.

The IntegerQueue ADT

Your IntegerQueue ADT will be based on a linked list data structure, and will use the following (abbreviated) header file. The complete header IntegerQueue.h is posted on the webpage, and will be submitted unchanged with your project.

```
// Exported type --------------------------------------------------------------

// IntegerQueue
// Exported reference type
typedef struct IntegerQueueObj* IntegerQueue;


// Constructors-Destructors ---------------------------------------------------

// newIntegerQueue()
// Constructor for the IntegerQueue ADT
IntegerQueue newIntegerQueue();

// freeIntegerQueue()
// Destructor for the Queue ADT
void freeIntegerQueue(IntegerQueue* pQ);


// ADT operations -------------------------------------------------------------

// isEmpty()
// Returns 1 (true) if Queue Q is empty, 0 (false) otherwise.
int isEmpty(IntegerQueue Q);

// length()
// Returns the number of elements in Q
int length(IntegerQueue Q);
```

```
// enqueue()
// Inserts x at back of Q.
void enqueue(IntegerQueue Q, int x);

// dequeue()
// Deletes and returns the item at front of Q.
// Pre: !isEmpty()
int dequeue(IntegerQueue Q);

// peek()
// Returns the item at front of Q.
// Pre: !isEmpty()
int peek(IntegerQueue Q);

// dequeueAll()
// Resets Q to the empty state.
void dequeueAll(IntegerQueue Q);


// Other Operations -----------------------------------------------------------

// IntegerQueueToString()
// Determines a text representation of IntegerQueue Q. Returns a pointer to a
// char array, allocated from heap memory, containing the integer sequence
// represented by Q. The sequence is traversed from front to back, each integer
// is added to the string followed by a single space. The array is terminated
// by a NUL '\n' character. It is the responsibility of the calling function to
// free this heap memory.
char* IntegerQueueToString(IntegerQueue Q);

// equals()
// Returns true (1) if Q and R are matching sequences of integers, false (0)
// otherwise.
int equals(IntegerQueue Q, IntegerQueue R);
```

This file is almost identical to the one found at

/Examples/Lecture/ADT-Examples/IntegerQueueADT/Array/

which is based on an array (with array doubling). Other than the underlying data structure, there are only two differences between these two versions of the IntegerQueue ADT. First, your IntegerQueue will contain an additional function called `length()` returning the number of elements in an IntegerQueue. Second, and more significantly, this version has no `printIntegerQueue()` function printing a text representation to an output stream. Instead, your version will contain an `IntegerQueueToString()` function that returns a `NUL '\0'` terminated char array allocated from heap memory. The text representation contained in the returned C string will be a space separated list of decimal integers, in the order front to back. Note that each integer in the queue, including the last, is followed by a space, and there is no newline `'\n'` character included in the returned string. The main difficulty in implementing this function over `printIntegerQueue()`, is knowing the correct length of the char array to allocate. (A similar problem arises designing function `DictionaryToString()` from pa3.) One way to proceed is to include a private helper function that goes through the IntegerQueue and counts the total number of decimal digits and spaces needed to create the required string. Another approach is to maintain the string length as a field in the `struct IntegerQueueObj`. Then every call to `enqueue()` will add to, and every call to `dequeue()` will subtract from, this accumulated string length. Design decisions like these are exactly the type of thing to include in your README file. Write your own client IntegerQueueTest.c to rigorously exercise your version of the IntegerQueue, and submit this test client with your project. A somewhat weaker test of the IntegerQueue ADT, called IntegerQueueClient.c, is included on the webpage.

The Job ADT

The Job ADT is contained in the files Job.h and Job.c, which are included on the webpage and should be submitted unchanged with your project. Job is a basic ADT encapsulating the triple (arrival time, duration, finish time). It includes a few access functions and manipulation procedures for doing things like computing the finish time. It proper use should be clear from the function headings. No tester file is included. Submit this ADT unaltered with your project.

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 file called Simulation, and includes a clean utility. Submit the following files to the assignment pa4 before the due date.

README	created by you
Simulation.c	created by you
IntegerQueue.h	provided, do not alter
IntegerQueue.c	created by you
IntegerQueueTest.c	created by you
Job.h	provided, do not alter
Job.c	provided, do not alter
Makefile	provided, alter as you see fit

This project is by far the most difficult one of the quarter. Creating and testing the IntegerQueue ADT should be routine, given your experience with lab3 (Stack ADT) and pa3 (Dictionary ADT). The principle difficulty is in writing Simulation.c, which is a non-trivial application of the IntegerQueue ADT. A file called SimulationStub.c is included on the webpage containing (an outline of) an algorithm that you may find useful. Finish the IntegerQueue as soon as possible, so you can begin working on the difficult part. As usual, start early and ask plenty of questions.