CS526
Homework Assignment 4

This assignment has two primary goals. The first goal is to give students an opportunity to study the implementation of an adaptable heap priority queue, which is discussed in Chapter 9. We discuss the outline of the implementation in the class. However, we do not discuss implementation-level details. Students are expected to acquire a deeper understanding of the implementation through this assignment

The second goal is to give students an opportunity to implement an application using a priority queue. The application to be implemented is a simulation of a process scheduler of a computer system. Though the scheduler, which is simulated by this assignment, is a small, simplified version, it reflects some of the basic operation of a typical process scheduler.

Implement this program as *ProcessScheduling.java*. You may have multiple classes and multiple files but your main program and the main part of the program must be in the *ProcessScheduling.java* file.

The following describes the scheduling system that is simulated.

> Processes arrive at a computer system and the computer system executes the processes one at a time based on some priority criterion. Each process has a *process id*, *priority*, *arrival time*, and *duration*. The *duration* is the amount of time that takes to completely execute the corresponding process. The system keeps a priority queue to keep arriving processes and prioritize the execution of processes. When a process arrives, it is inserted into the priority queue. Then, each time the system is ready to execute a process, the system removes a process with the smallest *priority* from the priority queue and executes it for the *duration* of the process.
>
> Suppose that a process *p* with a very large *priority* arrives at the system while the system is executing another process. While *p* is waiting in the queue, another process *q* with a smaller *priority* arrives. After the execution of the current process is finished, the system will remove and execute *q* (because *q* has a smaller *priority*), and *p* must wait until *q* finishes. If another process with a smaller *priority* arrives while *q* is being executed, *p* will have to wait again after *q* is completed. If this is repeated, *p* will have to wait for a very long time. One way of preventing this is as follows: If a process has waited longer than a predetermined maximum wait time, its *priority* is decremented.

For the purpose of this simulation, we assume the followings:

- We use the priority queue implemented in the *HeapAdaptablePriorityQueue.java*. This class implements an adaptable priority queue that uses a heap data structure.
- Each entry in the priority queue keeps (*key*, *value*) pair, which represents a process.

- The *key* of an entry is the *priority* of the corresponding process and it is of Integer type. The value of *priority* is between 1 and 10, inclusively. A process with a smaller *priority* is executed before a process with a larger *priority*.
- The *value* of an entry is the reference to the corresponding process.
- Each process must have, at the minimum, the following attributes:

```
id: integer              // process id
arrivalTime: integer     // the time when the process arrives at the system
duration: integer        // execution of a process takes this amount of time
```

The simulation program uses a logical time to keep track of the simulation process and the same logical time is used to represent the *arrivalTime* and *duration*. The simulation goes through a series of iterations and each iteration represents the passage of one logical time unit (in what follows we will use *time unit* only to refer to *logical time unit*). At the beginning, the current time is set to time 0. Each iteration implements what occurs during one time unit and, at the end of each iteration, the current time is incremented.

The following describes the simulation program:

- All processes are stored in a certain data structure *D*, which is supposed to be external to the computer system.
- In each iteration, the following occurs:
  - We compare the current time with the arrival time of a process with the earliest arrival time in *D*. If the arrival time of that process is equal to or smaller than the current time, we remove the process from *D* and insert it into the priority queue *Q* (this represents the arrival of a process at the system).
  - If no process is being executed at this time and there is at least one process in *Q*, then a process with the smallest *priority* is removed from *Q* and executed.
  - If the currently running process is completed, then we update the *priorities* of some processes. The update is performed as follows. If there is any process in *Q* that has been waiting longer than the maximum wait time, then the *priority* of that process is decreased by one.
  - The current time is increased by one time unit.
- The above is repeated until *D* is empty. At this time, all processes have arrived at the system. Some of them may have been completed and removed from *Q* and some may still wait in *Q*.
- If there are any remaining processes in *Q*, these processes are removed and executed one at a time.

A pseudocode of the simulation is given below. This pseudocode is a high-level code and detailes are not included. The omission of the details is intentional and students are expected to figure out the details. In the pseudocode, *Q* is a priority queue, which stores (*priority*, *process*) pairs. There is a Boolean variable *running* in the pseudocode. It is used to indicate whether the system is currently executing a process or not. It is true if the system is currently executing a process and false otherwise.

```
Read all processes from the input file and store them in an appropriate data structure, D
currentTime = 0
While D is not empty // while loop runs once every time unit until D is empty
    Get (don't remove) a process p from D that has the earliest arrival time
    If the arrival time of p <= currentTime
        Remove p from D and insert it into Q
    If Q is not empty and the flag running is false
        Remove a process with the smallest priority from Q
        Set a flag running to true
    currentTime = currentTime + 1
    If currently running process has finished
        Set a flag running to false
        Update priorities of processes that have been waiting longer than max. wait time

// At this time all processes in D have been moved to Q.
If there are any remaining processes in Q
    Remove them one at a time and execute it.
```

As mentioned in the first line of the pseudocode, an input file stores information about all processes. A sample input file is shown below:

```
1 9 19 2
2 9 6 8
3 9 9 13
4 6 15 21
5 7 19 25
6 7 12 32
7 2 11 42
8 5 20 49
9 1 9 59
10 7 10 69
```

Each line in the input file represents a process and it has four integers separated by a space(s). The four integers are the *process id*, *priority*, *duration*, and *arrival time*, respectively, of a process. Your program must read all processes from the input file and store them in an appropriate data structure. You can use any data structure that you think is appropriate.

However, for the priority queue, you must use the priority queue implemented in *HeapAdaptablePriorityQueue.java*. The *HeapAdaptablePriorityQueue* class inherits from or implements a few superclasses and interfaces. You may need to study some or all of these superclasses and interfaces to better understand the *HeapAdaptablePriorityQueue* class. On Blackboard, I will post only the *HeapAdaptablePriorityQueue.java* file. You can obtain other files from the textbook's web site (http://www.wiley.com/WileyCDA/WileyTitle/productCd-EHEP002900.html?filter=TEXTBOOK).

While your program is performing the simulation, whenever a process is removed from the priority queue (to be executed), your program must display information about the removed process. After your program finishes the simulation of executing all processes in the input file, it must display the average waiting time of all processes. A sample output is shown below, which is the expected output for the above sample input. In this example, the maximum wait time is 10.

Process removed from queue is: id = 1, at time 2, wait time = 0
Process id = 1
    Priority = 9
    Arrival = 2
    Duration = 19

Process removed from queue is: id = 4, at time 21, wait time = 0
Process id = 4
    Priority = 6
    Arrival = 21
    Duration = 15

Process removed from queue is: id = 5, at time 36, wait time = 11
Process id = 5
    Priority = 6
    Arrival = 25
    Duration = 19

Process removed from queue is: id = 7, at time 55, wait time = 13
Process id = 7
    Priority = 1
    Arrival = 42
    Duration = 11

Process removed from queue is: id = 9, at time 66, wait time = 7
Process id = 9
    Priority = 1
    Arrival = 59
    Duration = 9

List is empty
Process removed from queue is: id = 8, at time 75, wait time = 26
Process id = 8
    Priority = 4
    Arrival = 49
    Duration = 20

Process removed from queue is: id = 6, at time 95, wait time = 63
Process id = 6
    Priority = 4

Arrival = 32
Duration = 12

Process removed from queue is: id = 2, at time 107, wait time = 99
Process id = 2
Priority = 3
Arrival = 8
Duration = 6

Process removed from queue is: id = 3, at time 113, wait time = 100
Process id = 3
Priority = 3
Arrival = 13
Duration = 9

Process removed from queue is: id = 10, at time 122, wait time = 53
Process id = 10
Priority = 3
Arrival = 69
Duration = 10

Total wait time = 372.0
Average wait time = 37.2

Note that the *priority* of a process shown in the output is not necessarily the initial *priority* of the process. As mentioned earlier, the *priority* of a process may be updated if it has waited beyond the maximum wait time. So, the *priorities* in the output are the final ones that might have been updated.

Your program must write the output to an output file named *process_scheduling_out.txt*.

On the Blackboard, I posted two sample inputs and corresponding outputs (one is shown in this assignment). You can use these two sample inputs to test your program. However, since you cannot conclude your program is correct based on only two test results, you may want to test your program with some more test inputs.

Before you start your program, you may want to study the *HeapAdaptablePriorityQueue* class in the *HeapAdaptablePriorityQueue.java* file and its superclasses/interfaces carefully.

Documentation

No separate documentation is needed. However, you must include a brief description of the data structure *D* at the beginning of your source code as comments. You must also include the specification of each method in your program, right above each method, and you must include sufficient inline comments within your program.

Grading

Program correctness is worth 80% and documentation is worth 20%.

Your simulation program will be tested with different inputs and points will be deducted if your program does not produce the correct output.

Points will be deducted if you do not include a description of *D*, specifications of methods, or sufficient inline comments.

Deliverables

If you have a single file, name it as *ProcessScheduling.java* and upload it to Blackboard. If you have multiple files, combine them into a single archive file and name it *LastName_FirstName_hw4.EXT*, where *EXT* is an appropriate file extension (such as *zip* or *rar*). Upload this file to Blackboard.