Project 6. Heaps and Priority Queue (150 points)

Deadline: February 19th, 11:59pm.

Content:

Part 1. Implement d-ary min-heap.
Part 2. Priority Queue.
Part3. Test your priority queue.

The following files are (will be provided)

- dHeapInterface.html (Javadoc)
  - you need to implement dHeapInterface.java based on the html provided.
- dHeap.java (starter code)
- Record.java (starter code)
- EDF.java (starter code)

Part 1: (80 points)

Preamble:

Priority queues are an important data structure, with applications in areas such as job scheduling for printers and CPUs. Array-based binary heaps are one very efficient implementation of priority queues, with O(1) time findMin() and O(log n) time insert() and deleteMin(). Because the branching factor (the number of children a node can have) in a binary heap is 2, the base of the logarithm in O(log n) is 2 as well. In this project you will implement a *d-ary* heap, where **d** is the branching factor of the heap.

Why might it be a good idea to use d-ary heaps rather than binary heaps? The height of a complete binary tree containing 10,000 items is 13 (log base 2 of 10,000 is about 13), meaning that if the items are stored in a binary heap, insertions and deletions might take up to 13 swaps. If the same items are stored in a heap with branching factor 16, no more than 3 swaps will be required. The disadvantage of having 16 children for one node is that, for example, when

percolating down you have to find the smallest of the children, and the cost of this operation grows linearly with the number of children.

For a d-ary heap, the children of the node at array index j (when the root is at index 0) are at indices:

```
dj+1  to  dj+d
```

Also, the parent of the node at array index j is at:

floor((j - 1) / d)

Your heap will store data of a generic type T. Since heap relies on the order relation defined among elements, the heap will require that the type T implement the `Comparable` interface. Any particular T must satisfy the constraint that `T extends Comparable<? super T>`.

When defining your class dHeap, you should use the following syntax to ensure it compiles correctly:

```
public class dHeap<T extends Comparable<? super T>>
implements dHeapInterface<T>
```

To instantiate the array that you will use to implement a heap, you will need to downcast into the type T[] in a similar manner as you did for the escape room. However now you cannot just instantiate an Object[] array because the heap requires that the data all be `Comparable`; hence, instead you should instantiate the underlying array as:

```
T[ ] array = (T[ ]) new Comparable[123];
```

(The choice of 123 was arbitrary.)

Think about what other methods it might be useful to define in order to implement the required public methods. Since these other methods are not part of the public interface of your `dHeap` class, they should have **private** or **protected** visibility. For example, methods like:

private void trickleDown(int indx)
private void bubbleUp(int indx)
etc

Do not forget to create a JUnit tester file to test your methods: `dHeapTester.java`

Part 2. (20 points) Priority Queue

In this part you will create a `MyPriorityQueue` class that uses the `dHeap` class to provide the priority queue operations `add` and `poll` defined in the standard Java class PriorityQueue with a few exceptions:

- return type for add is void, not boolean
- ClassCastException is not necessary.

Your priority queue should a use **binary** heap for its implementation. Constructor for `MyPriorityQueue` takes one argument: the initial size of a queue.

Tester file is not required but make sure to test your class methods as well, since we will use our own tester to check your work.

Part 3. (50 points) Scheduling real time processes.

(Thank you, Prof. Biagioni for the assignment idea)

In part 3 you will write a class that creates a priority queue, reads a file (specified on the command line), and adds or removes processes from your priority queue.

The input file will have lines of two forms:

`schedule` *process deadline duration*

or

`run` *time*.

Where:

- **schedule** is a preset word that indicates that a new process is a about to start.
- *process* is an arbitrary string. Will be used to indicate the name of the process (lunch, work, play etc)
- *deadline* is a deadline of the process, represents time
- *duration* is the length of the process, represents time
- **run** is also a present word and indicates that you are ready to run the processes.
- *Time* represents a time frame (read further, it will make sense)

Input file example:

schedule breakfast 1000 45

schedule study 1400 280

schedule dinner 1725 30

run 350

The time units are not important, since the scheduler works the same either way. To make it easier to understand, you can think of the first two digits as *hours* and the last two digits as *minutes*.

For example

- breakfast would be scheduled to complete no later than 10:00am and 45 minutes are needed for breakfast,
- study would be scheduled with a deadline of 14:00 pm and you will need 280 minutes for that.
- dinner with a deadline of 17:25pm and 30 min are need.

Each line beginning with `schedule` tells your program to **insert** the corresponding item into the priority queue, with a priority corresponding to its **deadline**. Items with earlier (smaller) deadlines should be executed first -- this is called Earliest Deadline First scheduling, and is often used in real-time systems (you can read about it in Wikipedia if you are interested).

Your program must have a long integer variable to represent time (internal variable in your code). That variable is initially set to zero, and is increased as the program runs.

Each line beginning with `run` time tells your program to advance your internal time variable until it reaches `time`. You do this by repeatedly removing the first item from the priority queue, and advancing the internal time variable by the **duration** of the process, until the time is reached.

If the process completes before the time is reached, a new process is taken from the priority queue. If there are no processes left in the priority queue, the internal time is simply set to `time`. If, on the other hand, the current process would run past time, it is re-inserted in the priority queue with a duration equal to the remaining time.

Sample run

So in the example above, time starts at zero, and when your program reads the line that says

```
run 350
```

- it starts to run the first process in the queue, which will be "*breakfast*". Breakfast takes 45 minutes, so your program will remove "*breakfast* " from the queue and set its time variable to 45.
- The next item removed from the queue will be "*study*", which takes time 280. So, the time variable is set to 325 (280+45).

Now, "dinner" is removed from the queue. The time is now 325, and *dinner* is scheduled to take 30 time units. **But**, this run ends at time 350, whereas if the dinner was done now, the run would end at time 355. So, the time variable is set to 350, and your program must re-insert "dinner" into the priority queue with a deadline of 1725 (the deadline hasn't changed) and a duration of 5 (which is 30-(350-325)), the duration minus the difference between the end time and the start time).

Output requirements:

- Every time a process is inserted into the queue, your program should print

`current time`: *adding* `process` *with deadline* `deadline` *and duration* `duration`.

- Every time a process is removed from the queue, your program should print

`current time`: *busy with* `process` *with deadline* `deadline` and *duration* `duration`.

- When a process completes, your program should print

`current time`: done with `process`.

- If the process completes after its deadline, your program should print

`current time`: done with process (late).

where `current time` is the value of the time variable when the process is first removed from the queue.

Some examples

Given the above example file, your program should print:

```
0: adding breakfast with deadline 1000 and duration 45

0: adding study with deadline 1400 and duration 280

0: adding dinner with deadline 1725 and duration 30

0: busy with breakfast with deadline 1000 and duration 45

45: done with breakfast
```

```
45: busy with study with deadline 1400 and duration 280
325: done with study
325: busy with dinner with deadline 1725 and duration 30
350: adding dinner with deadline 1725 and duration 5
```

For another example, suppose this was your schedule:

schedule sleep 70 95

schedule coffee 80 20

run 70

schedule facebook 90 35

run 100

```
0: adding sleep with deadline 70 and duration 95
0: adding coffee with deadline 80 and duration 20
0: busy with sleep with deadline 70 and duration 95
70: adding sleep with deadline 70 and duration 25
70: adding facebook with deadline 90 and duration 35
70: busy with sleep with deadline 70 and duration 25
95: done with sleep (late)
95: busy with coffee with deadline 80 and duration 20
100: adding coffee with deadline 80 and duration 15
```

Another test run is in a separate text file.

What to submit:

- dHeapInterface.java
- dHeap.java
- dHeapTester.java
- MyPriorityQueue.java
- Record.java
- EDF.java //stands for Earliest Deadline First

Make sure to look at your submission report!!