Project 4: About Priority Queue...

We just studied the *Priority queue* structure, which is more general and useful as compared with the FIFO queue, as it also takes the priority of individual tasks into account. We will study its usefulness in other subsequent courses, especially in *CS 4310* *Operation Systems*. In this project, we will implement this structure in multiple ways and compare their behaviors.

1 The **maxPriorityQueue** interface

Below is an interface for the **maxPriorityQueue** ADT (Abstract Data Type), as given in Page 24 of the *Heapsort* notes.

```
public interface maxPriorityQueue{
    //Below inserts x into this maxPriorityQueue
    public void insert(Comparable x);

    //Below returns the element of this maxPriorityQueue with the largest key.
    public comparable maximum();

    //The following method removes and returns the element of this
    //maxPriorityQueue with the largest key.
    public comparable extractMax();

    //The following method increases the value of element located at
    //position i to the new value k, which is no smaller than its
    //original key value.
    public void increaseKey(int i, Comparable k);
}
```

2 What to do?

A very important feature of an abstract data structure is that it can be implemented in different ways. For this project, you have to go through the following steps:

1. An implementation of this **maxPriorityQueue** in terms of the *Heap* structure has been thoroughly discussed in the lecture notes on Heapsort. Study the pseudo codes of the above methods, and implement them in Java.¹

¹You need this implementation for a later project.
2. Provide two more implementations of `maxPriorityQueue`, one with a sorted array, the other with an unsorted array.

We have discussed these two implementations in a class. For example, with the initially empty unsorted array implementation, although you can `insert` the next item right in the first available place in $\Theta(1)$ time, you have to look for the `maximum` element in the list in $\Theta(n)$, where $n$ is the number of elements as contained in such a list. Once you have extracted the maximum element from the list, you also have to fill this “hole” by moving all the elements to the right of such a maximum element one position to the left, also in $\Theta(n)$. Finally, when you want to increase the value of an element, you can just do it in $\Theta(1)$.

3. In light of the above analysis for the unsorted case, make a theoretical analysis for the four operations in terms of $n$, the size of the `maxPriorityQueue`, for the sorted list implementation.

4. Practically, for $n = 10, 50, 100, 200, 500, 1000, 2000, 5000$, come up with a structure in each and every one of the above three implementations that can contain $n$ elements, and fill them with a randomly generated list of $n$ elements.

More specifically, for the above three implementations, find out the average number of comparisons and movements $^2$ as involved in the three operations of maximum, maximum extraction, and key increase $^3$; then compare them with the above theoretical data as given in Step 2, that you obtained in Step 3 and as you can find in the book/lecture notes.

3 What should be handed in?

Email me the source code, as well as one lab report containing 1) the theoretical results as mentioned in Step 3; 2) practical results, together with the comparison of the two kinds of results as mentioned in Step 4; and 3) most importantly, your justification behind such a comparison, i.e., why do you believe the practical results agree with the theoretical ones $^4$.

---

$^2$To iron out the noise, repeat each of the operations $n$ times, then take the average of these $n$ results. For example, when $n = 10$, repeat the process 10 times, then take the average of these ten runs as the average number for $n = 10$.

$^3$You might use two extra random number generators: one to identify an element, and the other for the increase value.

$^4$You might want to review the sampler for Project 3 and my comments made at the end....