Project 4: About Priority Queue...

1 Why do we do it?

We just studied the Priority queue structure, which is much more useful, practically speaking, as compared with the FIFO queue, as it also takes the priority of individual tasks into account. We will have a closer look at its application in some of the subsequent courses, especially in CS4310 Operation Systems.

We will implement this structure in multiple ways and compare their behaviors in this project, which we will also use in a later project when we implement and study the Huffman code in the context of greedy algorithms.

2 The maxPriorityQueue interface

Below is an interface for the maxPriorityQueue ADT (Abstract Data Type), as given in Page 22 of the Heapsort notes.

```java
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);
}
```
3 What to do?

A very important feature of an abstract data structure is that it can be implemented in different ways. For this project, we will implement the above interface\(^1\) in three ways.

1. We discussed in class two simple implementations of \texttt{maxPriorityQueue}, one with a \textit{sorted array}, the other with an \textit{unsorted array}.

   For example, with the initially empty unsorted array implementation, although you can \textit{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 \textit{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 \textit{increase} the value of an element, you can just do it in $\Theta(1)$.

   An implementation with a sorted array can be similarly done.

2. An implementation of this \texttt{maxPriorityQueue} in terms of the \textit{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\(^2\).

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 \texttt{maxPriorityQueue}, for the \textit{sorted list} implementation.

4. Practically speaking, 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.

   Then, for the above three implementations, find out the average number of comparisons and movements\(^3\) as involved in the three operations of maximum, maximum extraction, and key increase\(^4\) and compare them with the theoretical results as achieved in the lecture notes.

4 What should be handed in?

Email me the source code, as well as a lab report containing the comparisons between the theoretical results as mentioned in Step 3 and practical ones as mentioned in Step 4, as well

---

1. Your implementations have to share the same interface.
2. You need this implementation for a later project.
3. 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 $n = 10$.
4. You might use two extra random number generators: one to identify an element, and the other for the new value, as defined in the \texttt{increaseKey} function as discussed in Section 2.
as your justification behind such a comparison, i.e., why do you believe the practical results agree with the theoretical ones.

5 A general grading guideline

≥ 1: Successful implementation of both the sorted and unsorted list based priority queue, as required in Step 1

≤ 2: Successful implementation of the heap based priority structure, as shown in Step 2

≤ 3: Theoretical analysis of the sorted list based structure, as shown in Step 3

≤ 4: Collected data showing comparison of the average number of comparisons and movements among the three implementations as shown in Step 4

≤ 5: A satisfactory justification of the relationship between the practically obtained data and the theoretically derived results (Step 4)

---

5You might want to review the sampler for Project 3, and my comments made at the end as how to provide a satisfactory justification.