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 (First In First Out) queue, as it also takes the priority of individual tasks into account. The difference between the normal FIFO and the priority queue is that, when we take something out from a FIFO queue, we always take out the one that has been there the longest time; while, when we take something out a priority queue, we choose the one with the highest priority.

We will have a closer look at the application of priority queues in some of the subsequent courses, especially in CS4310 Operation Systems, when a processor become available, we want to give it to the “most important” process first, i.e., the one attached with the highest priority.

We will implement this priority queue 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 coding in the context of greedy algorithms.

2 The maxPriorityQueue interface

The following is an interface for the maxPriorityQueue ADT (Abstract Data Type), as given on Page 27 of the HeapSort notes.

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

    //This one returns, but not removes, 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?

As we saw in Project 2, a sorting algorithm can be implemented in various ways, an abstract
data type (ADT) can be implemented in different ways. For this project, we will implement
this priority queue ADT \(^1\) in three ways, all based on an array of int, as declared with the
following, since we need to know exactly where each item is located, e.g.,

```java
int sortedPQ[];
// or the following should also work
int[] sortedPQ;
```

1. An `maxPriorityQueue` can clearly be implemented with an unsorted array, as we dis-
cussed in a class:

   - For the `insert` operation, you can insert the next item right in the first available
     place in \(\Theta(1)\) time.
   - For `maximum` operation, since this list is not sorted, you have to look for the
     maximum element in the array, with \(O(n)\) comparisons, where \(n\) is the number of
     elements as currently contained in such an array.
   - Once you have extracted, or removed, the maximum element from the array,
     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, leading to \(O(n)\) movements.
   - Finally, when you want to increase the value of an element, you can just do it
     in \(\Theta(1)\), again since this list is not sorted.

You need to implement the above algorithms, in Java, as an `unsortedList` implemen-
tation of the `maxPriorityQueue` interface as given in Section 2.

2. In light of the above discussion and analysis for the unsorted case, come up with algo-
rithms for these four operations, i.e., `insert`, `maximum`, `extraction`, and `increase`
its value, when implementing the `maxPriorityQueue` ADT with a sorted array, where
everything is sorted as we know it, and make a theoretical analysis for the four oper-
ations in terms of \(n\), the size of the `maxPriorityQueue`, i.e., how many comparisons,
insertions, and movements in terms of \(n\), the number of elements in the list.

You also need to implement the above algorithms, in Java, as a `sortedList` implemen-
tation of the `maxPriorityQueue` interface.

3. Algorithms of the four operations for this `maxPriorityQueue` ADT in terms of the `max-
Heap` structure have been thoroughly discussed in the lecture notes on HeapSort. Study
the pseudo codes of the above methods, and come up with a `maxHeap` implementation
of the `maxPriorityQueue` interface \(^2\).

---

\(^1\)Your implementations have to share the same interface, similar to what you did with the sorting mech-
anisms in Project 2.

\(^2\)You need this implementation for a later project.
4. Practically speaking, for \( n = 10, 50, 100, 200, 500, 1000, 2000, 5000 \), come up with an array of type \texttt{int}, in each and every one of the above three implementations, and fill them with a randomly generated list of \( n \) elements, similar to what you have done in Project 3.

For the above three implementations in an unsorted array, a sorted array, and in maxHeap, find out the average number of comparisons and movements as involved in the three operations of \texttt{maximum}, \texttt{extractMax}, and \texttt{increaseKey}.

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 \( n = 10 \). Same thing goes to the other values of \( n \in \{50, 100, 200, 500, 1000, 2000, 5000\} \).

Finally, use a spreadsheet to collect all these data and compare them with the theoretical results as shown on Pages 30 and 31 in \textit{HeapSort} notes for the maxHeap implementation, those as shown in Step 1 for the unsorted case, and what you have done for Step 2 for the sorted case.

5. You have to justify your conclusion.

4 What should be handed in?

Send in the whole collection of program files, and a lab report containing the data that you have collected, comparisons between the theoretical results and practical ones as mentioned in Step 4, as well 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

\[ \geq 1: \] Successful implementation of both the sorted, and unsorted, array based priority queue, as required in Step 1

\[ \leq 2: \] Theoretical analysis of the sorted array based structure, as shown in Step 2

\[ \leq 3: \] Successful implementation of the maxHeap based priority structure, as shown in Step 3

\[ \leq 4: \] Collected data showing comparison of the average number of comparisons and movements among the three implementations as shown in 4

\[ \leq 5: \] A satisfactory justification of the relationship between the practically obtained data and the theoretically derived results as shown in Step 5

---

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.

You might want to review the samplers for Project 3 for a satisfactory justification.