Priority Queue Group Exercises – KEY
– scaled to 50 pts

By Ali Ebrahimpour

Please work in groups of 2 or 3 to work the following problems. Use additional paper as needed, and staple the sheets together before turning them in. ONLY TURN IN 1 WORKSHEET/ANSWERS PER TEAM.

1. **12 pts** For a data structure to be a *min-priority queue* we must have these functions:
   i. **Insert**: To insert a new element to the priority queue
   ii. **FindMin**: To find the minimum element of the priority queue
   iii. **ExtractMin**: To find and extract the minimum element of the priority queue, then re-build it if necessary.

   For any 2 of the data structures below, describe how we can use them to implement a priority queue and the time complexity of each of the above three functions when we use them.
   
   a) A simple array (list)  
   b) A linked list  
   c) A simple list and a pointer to the element of the list with minimum value  
   d) A sorted list  
   e) A sorted linked list  
   f) Heap

A: Insert in O(1) – findMin in O(n) – extractMin in O(n)
B: Insert in O(1) – findMin in O(n) – extractMin in O(n)
C: Will be discussed at the beginning of the class
D: (If it’s in decreasing order) Insert in O(n) – findMin in O(1) – extractMin in O(1)
   (If it’s in increasing order) Insert in O(n) – findMin in O(1) – extractMin in O(n)
E: (If it’s in decreasing order) Insert in O(n) – findMin in O(1) – extractMin in O(1)
   (If it’s in increasing order) Insert in O(n) – findMin in O(1) – extractMin in O(1)
F: (for min-heap) Insert in O(log n) – findMin in O(1) – extractMin in O(log n)
2. **6 pts** Suppose we are implementing a heap by using an array for which the \(i\)th element of the array is the parent node of the nodes \(2i\) and \(2i+1\) of the array (if not empty), in the tree representation of the heap. Suppose we have the array below:

\[
\begin{array}{cccccccccccc}
1 & 5 & 8 & 11 & 6 & 10 & 14 & 13 & 15 & 8 & 9 & 12 & 15 & 14 & 16 & 8 & 18
\end{array}
\]

a) Draw the tree of the above array.

b) Is this tree a min-heap? Why?

No, because the leaf node 8 is a child of 13 which is a bigger number.

c) Write the steps that should be taken to fix the tree to become a min-heap.

Heapify-up the leaf node 8 two times.

3. **3 pts** If we had a sorted array in the previous question, was it a heap? Explain your answer.

Yes, because children come after their parent, and since the array is sorted they should be bigger than their parent. So, we have a heap.

4. **3 pts** Compute the total number of elements in a complete (i.e. full) heap with a tree of height \(h\).
It has $2^n$ nodes. If the root is not considered as an element itself, then number of elements will be one less than this.

5. **9 pts** Suppose that we want to use two heaps, one min-heap and one max-heap, to make a data structure that provides us with these functions, which have the complexities shown in parentheses:
   
   i. Find the median (in $O(1)$)
   
   ii. Extract median (in $O(\log n)$) – this function also rebuilds the heaps
   
   iii. Insert a new element (in $O(\log n)$)

   Explain how we can use a max- and min-heap in each of the operations

   *Hint:* Look at the max-heap and min-heaps below, and try to do each of the 3 operations on them. (The median for these numbers is 11)

   ![Max-heap](image1)

   ![Min-heap](image2)

   When we want to add a new element, if both the heaps have the same size we arbitrarily choose one and add the element to that. If one of the heaps is bigger, we add the element to the smaller heap. Now we should check the invariant that all the elements of the min-heap should be bigger than all the elements of the max-heap. By adding the new element to one of the trees, the new root may not satisfy this invariant. Let’s say we have added the new element to the max-heap, and then the new number was big enough to become the root. So now we have to compare the new root to the root of the other heap. If this root is now bigger than the root of the min-heap, we should extract the root of the min-heap and add it to the max-heap, and extract the root of the max-heap and add it to the min-heap. This way we preserve the invariant. The median at all time will be either the root of the bigger tree, if the total number of elements is odd, or the mean of the elements of the root, if the total number of elements is even.

   A (easier) way to solve this problem is as follows. When we want to add a new element, we compare it to the root of the max-heap. If it’s smaller we add the new element to the max-heap, but if it’s bigger, we add it to the min-heap. So, this way the invariant is always satisfied. Then since we want heaps to have almost the same size, we check the number of elements. If one of them has at least 2 more elements than the other one, we extract one of its elements and insert it to the other heap.