Solutions to Problem Set 4

Solution to Problem 1 Operations 1 and 2 are the usual queue operations. Operations 1 and 3 are the usual min-priority queue operations. So, we will use a queue $Q$ (implemented as say a linked list with pointers to the back and front) and a min-heap $H$ (to implement the priority queue). In addition, each item in $Q$ will maintain a pointer (index) to the corresponding item in $H$, and vice versa.

To implement Find-Min, we just access the root item of $H$. That will take constant time.

To implement Insert($x$), we just insert $x$ into the queue and into the heap, and create pointers between the queue item $x$ and the heap item $x$. Inserting into a queue takes constant time, and inserting into a heap takes $O(\log n)$ time, so the total time is $O(\log n)$.

To implement Delete, we first extract the front item from $Q$. Then, using the pointer to $H$, we delete the corresponding item in $H$. (Namely, we promote the last item in $H$ to where the deleted item was in $H$, and then do a “float down”.) Whenever we move an item in $H$, we update the pointers between $Q$ and $H$. Again, deleting from a queue takes constant time, and floating down a heap takes $O(\log n)$ time, so the total time is $O(\log n)$.

There is a data structure that can implement all 3 operations in amortized constant time. The idea is to replace the min-heap with a deque containing a subset of the items, similar to the solution to Problem 3. The details are omitted.

Solution to Problem 2 We will use an augmented 2–3 tree in which each internal node $N$ maintains the number of leaves in the subtree rooted at $N$.

The Member operation is done as usual in a 2–3 tree.

The Insert($x$) operation is done as usual in a 2–3 tree, with extra code to increment the leaf counts of the nodes on the path from the root to the new leaf, and to update the leaf counts during node splittings.

To implement the Delete($k$) operation, we first need to find the $k$ smallest item. We can use the leaf counts as a guide to decide at each step whether to go left, to the middle, or right. Once we find the leaf to be deleted, we delete it as usual in a 2–3 tree, with extra code to decrement the leaf counts on the path from the root to the deleted leaf, and to update the leaf counts during node fusions.

As in 2–3 trees, the running time of each operation is $O(\log n)$.

I probably should fill in some details here . . .

Solution to Problem 3 We will store the items in a deque (double-ended queue) $D$. We will maintain $D$ so that, from front to back, the items are in
increasing order. A deque can be implemented by a doubly-linked list, with
two pointers to the front and back, so that each deque operation runs in worst-case
constant time.

To implement the Extract-Min operation, we just Extract the front of \( D \).

To implement the Insert-Max(\( x \)) operation, use the following code:

\[
\text{while not empty}(D) \text{ and } \text{back}(D) > x \text{ do } \\
\quad \text{Delete-back}(D); \\
\quad \text{Insert-back}(D, x)
\]

It is easy to see, by induction on the number of operations, that \( D \) will be
in increasing order, and that both operations are implemented correctly.

Running Time: Extract-Min just involves one deque operation, and so runs
in worst-case constant time. What about Insert-Max? Well, after the first
\( k \) Insert-Max operations, there were exactly \( k \) Insert-backs performed. Hence
there could be at most \( k \) Delete-backs performed, because an item cannot be
deleted unless it was first inserted. Hence the number of deque operations
performed is \( O(k) \), which means the total time is \( O(k) \).

Solution to Problem 4 Imagine a tournament among the \( n \) items, in which
the smaller item wins each game. Store the results in a binary tree or heap
array. (Each result gives us the minimum item over a certain interval.) The
number of games is \( n - 1 \), so the total space is \( O(n) \).

To implement Find-Min(\( i,j \)), we just partition \( i..j \) into intervals that we have
stored the answers to. It is easy to see that every interval can be partitioned
into \( O(\log n) \) intervals that we stored the answers to. (To see why, consider the
paths from \( i \) and \( j \) to the root in the tournament.) So all we have to do is take
the minimum of those \( O(\log n) \) answers. That will take \( O(\log n) \) time.

I probably should fill in some details here...