HOMEWORK 5

5.1: (a) $O(n \log n) + O(k) = O(n \log n)$
(b) $O(n) + O(K \log n) = O(n)$
(c) $O(n) + O(n) + O(k \log k) = O(n)$
(d) $O(n \log n) + O(k \log n) = O(n \log n)$

5.2:

\textbf{HEAP-DELETE}(A, i)
\begin{itemize}
  \item \textbf{if} $i > \text{heapsize}[A] \text{ or } i < 1$
  \item \textbf{then} \textbf{error} "out of range"
  \item $A[i] \leftarrow A[\text{heapsize}[A]]$
  \item $\text{heapsize}[A] \leftarrow \text{heapsize}[A] - 1$
  \item \textbf{DOWN-HEAPIFY}(A, i)
\end{itemize}

5.3: (a) $O(k \log n)$

(b) $O(kn)$

(c) After the first find min, the min is placed at the top level (root) of the splay tree; after the first find max, the max becomes the root, and the min moves to either the third level, or the second level; then the min and the max will moves between the second/third level and the top level alternatively, by executing find min and find max alternatively (see figure 1 for illustration). The worst-case bound is $O(n) + O(k)$. 
(d) If $k = \Theta(\sqrt{n})$, balanced binary search tree is most efficient for the operations in this problem.

5.4: We need an auxiliary data structure $S$, which supports two operations \textsc{Insert} and \textsc{Extract-Max}, both with $O(\log n)$ running time. Such data structure can actually be implemented as a priority queue. Assume the heap in the problem stored in an array $A[1..n]$, we develop the following $O(k \log k)$-time algorithm to find the $k$th largest number in $A$.

1. Initialize $S$ to be empty;
2. \textsc{Insert} $A[1]$ into $S$ (here $A[i]$ means both its value and index in the array);
3. Apply \textsc{Extract-Max} on $S$, which returns $A[j]$;
5. Goto step 3, until the $k$th \textsc{Extract-Max}, which returns the $k$th largest number in the heap.

The algorithm runs in $k$ steps, and at each step no more than 2 elements are inserted into $S$, so the size of $S$ is bounded by $O(k)$. Because both \textsc{Insert} and \textsc{Extract-Max} are logarithmic time, the algorithm is therefore runs in $O(k \log k)$ time.