Assignment 3 – Selected Model Answers

Exercise 1.

(i) Show that any comparison-based algorithm for determining the smallest of n elements requires $n-1$ comparisons.

(ii) Show also that any comparison-based algorithm for determining the smallest and second smallest elements of n elements requires at least $n-1+\log n$ comparisons.

Note. You must consider that an arbitrary *algorithm* contains these number of comparisons, whereas on a specific input the number may still be lower (see the proof on the number of comparisons in a sorting algorithm).

(iii) Give an algorithm with this performance.

Solution.

(i) Let $\ell = [x_1, \dots, x_n]$. As for the proof we build a binary decision tree, where the leaves are labelled with one element x_k and the non-leaf vertices are labelled by comparisons $x_i \le x_j$. The edge from a non-lef vertex to its left successor is labelled with x_j, and the edge to its right successor is labelled with x_i. If we have a path $v_0, \dots, v_d$ from the root to a leaf, then all edges on such a path are labelled by x_j that have been discarded for being the minimum. That is, we must have $d \ge n-1$.

(ii) As in the proof of the lower bound of comparison-based sorting algortithms let $\ell = [x_1, \dots, x_n]$. Build a binary decision tree with non-leaf-vertices labelled by comparisons. A leaf is labelled by a pair (x_i, X) with $1 \le i \le n$ and $X \subseteq \{x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n\}$.

If $v_0, \dots, v_d$ is a path from the root to a leaf, where the label of v_k is $x_{k_i} \le x_{k_j}$ (for $0 \le k \le d-1$), then v_k corresponds to

$$\varphi_k = \begin{cases} x_{k_i} \le x_{k_j} & \text{if } v_{k+1} \text{ is the left successor of } v_k \\ x_{k_j} < x_{k_i} & \text{if } v_{k+1} \text{ is the right successor of } v_k \end{cases} .$$

If v_d is labelled by (x_i, X), then $\{\varphi_0, \dots, \varphi_{d-1}\}$ imply $x_i = \min\{x_1, \dots, x_n\}$ and $x_i \le x_j < x$ for all $x \in X$, where $j \ne i$ and x_j is the second smallest element of $\{x_1, \dots, x_n\}$.

Then there are $n \cdot 2^{n-1}$ leaves. There can be at most 2^d leaves in a binary tree of depth d. This implies $2^d \ge n \cdot 2^{n-1}$, i.e. $d \ge \log_2(n \cdot 2^{n-1}) = \log_2 n + n - 1$ as claimed.

(iii) If ℓ is the input list, then we should assume $|\ell| \ge 2$. For $|\ell| = 2$ a single comparison suffices to determine the smallest and second smallest element, for $|\ell| = 3$ three comparisons are required.

For $|\ell| > 3$ split ℓ into two sublists ℓ_1 and ℓ_2, where $|\ell_1|$ is a power of 2 and $|\ell_2| \ge 2$. Then apply the algorithm recursively to ℓ_1 and ℓ_2. Given $x_1^1 \le x_2^1$ and $x_1^2 \le x_2^2$ for the resulting smallest and second smallest elements of ℓ_1 and

ℓ_2, respectively, two comparisons—first x_1^1 with x_1^2, then the larger of these two with one of the remaining elements—suffice to determine the smallest and second smallest elements of ℓ.

In this way we obtain a linear recurrence equation, and then we can proceed to show that the number of comparisons is $n - 1 + \log n$. We omit further details and the implementation.

EXERCISE 2.

(i) Implement max-heaps using arrays. In particular, implement *build_heap* and *sift-down*.

(ii) Implement heapsort using max-heaps.

EXERCISE 3.

(i) Show how addressable priority queues using doubly linked lists can be realised, where each list item represents an element in the queue, and a handle is a handle of a list item.

(ii) Determine and the complexity of queue operations for two different options using sorted lists or unsorted lists.

SOLUTION.

(i) We use a doubly linked list storing in each node a unique identifier as handle and a key value. For *insert* we create a new handle and add the new node at the end of the list (if unsorted) or after the last node with a smaller key value. For *delete* scan the list for the given handle, then remove the node. For *decrease* also scan the list for the given handle. Then either update the key value in the found node (unsorted case) or delete the node and insert a new one with the decreased key value (sorted case). For *delete_min* either delete the first node in the list (sorted case) or scan the list until the node with minimum key value is found and delete this node.

(ii) In the unsorted case an insertion requires time in $O(1)$, as only a new node is added to a doubly linked list. A *delete* or a *decrease* require a linear search through the whole list until the node is found, which requires $O(n)$ time. Similarly, a *delete_min* requires a search of the list for the minimum key value in linear time and a deletion in constant time, so the time complexity is in $O(n)$.

In the sorted case an insertion requires a linear search through the list with complexity in $O(n)$. A *delete* or a *decrease* require a linear search through the whole list until the node is found, which requires $O(n)$ time. For *decrease* we further have to insert the updated node, which also requires time in $O(n)$. However, *delete_min* only deletes the first node, which can be done in time $O(1)$.

EXERCISE 4.

(i) Design an algorithm for inserting k new elements into a max-heap with n elements.

(ii) Give an algorithm with time complexity in $O(k + \log n)$.

Hint. Use an approach similar to the building of a heap.

SOLUTION.

(i) We add the k elements at the end of the representing array and use *sift-up* to restore the max-heap property. The complexity of *sift-up* for a tree with n nodes in in $O(\log n)$, because $\log_2 n$ is the height of the tree, which bounds the number of swaps. Then the algorithm will have complexity in $O(k \log n)$.

(ii) In order to improve the complexity to be in $O(k + \log n)$ we proceed differently. Again we start with adding the k new elements to the end of the representing array, but the max-heap property is restored in a different way. We proceed in a bottom up way starting with the nodes that have at least one child among the k new elements.

There are $\lceil k/2 \rceil$ such nodes. Then we *sift-down* the roots of the trees rooted at these nodes. These subtrees have height 1, so the complexity is $c \cdot \lceil k/2 \rceil$ with some fixed constant $c > 0$. We iterate this taking next those nodes that have at least one child among the nodes treated in the previous step. There are $\lceil k/4 \rceil$ such nodes. Then we *sift-down* the roots of the trees rooted at these nodes. These subtrees have height 2, so the complexity is $c \cdot \lceil 2k/4 \rceil$. We iterate first until we reach the root. Summing up the complexity for each step we obtain

$$c \cdot \sum_{i=1}^{\log_2 n} \left\lceil \frac{k}{2^i} \right\rceil i \ .$$

For any $1 \le i \le \log_2 n$ we can write $k = a_i \cdot 2^i + r_i$ with $r_i < 2^i$, which gives us

$$\left\lceil \frac{k}{2^i} \right\rceil = \frac{k}{2^i} + \frac{r_i}{2^i} \ .$$

As the set $\{r_i \mid i \ge 1\}$ is finite, it has a maximum m. Furthermore, the series $\sum_{i=1}^{\infty} \frac{i}{2^i}$ converges to some $d > 0$, so the sequence $\left\{ \frac{i}{2^i} \right\}_{i \ge 1}$ converges to 0, which implies that it is bounded by a constant c'. Taking this together we get

$$\begin{aligned} c \cdot \sum_{i=1}^{\log_2 n} \left\lceil \frac{k}{2^i} \right\rceil i \ &= \ c \cdot \sum_{i=1}^{\log_2 n} k \frac{i}{2^i} + c \cdot \sum_{i=1}^{\log_2 n} \frac{i}{2^i} r_i \\ &\le \ c \cdot k \cdot \sum_{i=1}^{\infty} \frac{i}{2^i} + c \cdot m \cdot \log_2 n \cdot c' \\ &\le \ c \cdot d \cdot k + c \cdot m \cdot c' \cdot \log_2 n \ \in \ O(k + \log n) \end{aligned}$$

as claimed.