Assignment 4 – Selected Model Answers

Exercise 1.

(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 2.

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

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

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 $1 \leq i \leq \log_2 n$ 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 $\{r_i \mid i \geq 1\}$ is finite, it has a maximum m. Furthermore, the series $\displaystyle\sum_{i=1}^{\infty} \frac{i}{2^i}$ converges to some $d > 0$, so the sequence $\displaystyle\left\{ \frac{i}{2^i} \right\}_{i \geq 1}$ converges to 0, which implies that it is bounded by a constant c'. Taking this together we get

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

as claimed.

EXERCISE 3.

(i) Show that the running time of `siftUp`(n) is $O(\log n)$ and hence an insert into a heap takes time in $O(\log n)$.

(ii) The `siftDown` used in the *heapsort* algorithm requires about $2 \log n$ comparisons. Show how to reduce this to $\log n + O(\log \log n)$.

SOLUTION.

(i) `siftUp`(n) successively swaps an element in position n with the parent in position $\lfloor n/2 \rfloor$, if the parent is smaller. In the worst case the last swap involves the root. So the complexity is in $O(\ell)$, where ℓ is the length of the path from position n to position 1 of the root. As heaps are almost complete binary trees, we have $\ell \leq \log_2 n$, which shows the claimed time complexity in $O(\log n)$.

(ii) We can successively compare children nodes to determine a path from the root to a leaf, along which elements would be used in `siftDown` operations. The length of such a path is $\leq \log_2 n$, so the number of comparisons needed to determine the path is also at most $\log_2 n$. The elements along the path define an ordered list of length $\log_2 n$, and hence binary search in such a list requires a number of comparisons in $O(\log \log n)$. The actual inserton is then domne without further comparisons.

EXERCISE 4.

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

(ii) Implement heapsort using max-heaps.

SOLUTION. See the C++ header and program files in `Ass4_Ex4solution.zip`.