Assignment 4 – Selected Model Answers

EXERCISE 1.

(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 $\log n + O(\log\log n)$.

Hint: Determine first a path p along which elements need to be swapped, then perform a binary search on this path to find the proper position for the root element.

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 $\ell \le \log_2 n$, 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 $\le \log_2 n$, 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 2. Let R be a binary relation on the set of integers.

(i) Use an associative array to represent R such that it becomes easy to check whether R is symmetric.

(ii) Implement your symmetry checking algorithm.

SOLUTION. We only look at part (i). We can assume that $R \subseteq S \times S$ with $S \subseteq \mathbb{Z}$ and $|S| \le m$. Then we can assume a hash function $h : S \to \{0, \dots, m-1\}$.

Then we define a hash function $H : S \times S \to \{0, \dots, m^2 - 1\}$ by $H(x, y) = h(b) \cdot m + h(a)$. We use this function for hashing with linear probing to store the relation R in an array of length m^2.

In order to determine if R is symmetric, we have to check for every $(a, b) \in R$, if also $(b, a) \in R$ holds. For this take all pairs $(a, b) \in S \times S$. First check if $(a, b) \in R$, which is done by searching for (a, b) in the hash table using its hash value $H(a, b)$; the search procedure is the one for linear probing. If we do not find $(a, b) \in R$, we proceed with the next element.

If we find $(a,b) \in R$, we have to search for (b,a) in the hash table. We obtain $h(a) = H(a,b) \bmod m$ and $h(b) = (H(a,b) - h(a))/m$. With this we compute $H(b,a) = h(a) \cdot m + h(b)$, and finally use this hash value to search for (b,a) in the hash table.

Exercise 3. *Pairing heaps* are an efficient data structure using a very simple technique for rebalancing, though a full theoretical analysis is missing. They rebalance only in connection with the *deleteMin* operation. If $r_1, \dots, r_k$ is the sequence of root nodes stored in a location `roots`, then *deleteMin* combines r_1 with r_2, r_3 with r_4, etc., i.e. the roots are paired.

(i) Explain how to implement pairing heaps using three pointers per heap item i: one to the oldest child (i.e. the child linked first to i), one to the next younger sibling (if any), and one to the next older sibling. If there is no older sibling, the third pointer goes to the parent.

(ii) Explain how to implement pairing heaps using only two pointers per heap item: one to the oldest child and one to next younger sibling. If there is no younger sibling, the second pointer goes to the parent.

Solution. The operation *deleteMin* produces first a heap-ordered forest, then trees t_1, t_2 are combined, trees t_3, t_4 are combined, etc. So we have to look at the implementation of the *combine* operation for both data structures.

Assume that two trees t_1 and t_2 with roots r_1 and r_2 are combined. Without loss of generality we can assume $val(r_1) < val(r_2)$.

(i) We have to determine the youngest child c of the root r_1. For this first follow the link from r_1 to the oldest child, then successively follow links to the next younger sibling until no such sibling exists; this is the case for c. Create r_2 as a new youngest child, i.e. r_2 has a pointer to c, and c has a pointer to r_2. All other pointers in t_1 and t_2 are preserved.

(ii) In the case of only two pointers proceed analogously to determine the youngest child c of the root r_1. That is, first follow the link from r_1 to the oldest child, then successively follow links to the next younger sibling. However, there is always such a pointer, so we have to check, if the "younger sibling" is actually r_1, which is the case for c. Then create r_2 as a new youngest child, i.e. r_2 has a pointer to r_1, and c has a pointer to r_2. All other pointers in t_1 and t_2 are preserved.

Exercise 4. A *McGee heap* has the same structure as a Fibonacci heap, but supports just the mergeable heap operations. The implementations of the operations are the same as for Fibonacci heaps, except that insertion and union consolidate the root list as their last step. What are the worst-case running times of operations on McGee heaps?

Solution. The operations on a McGee heap are *insert*, *min*, *delete_min* and *union*. The operations *min* and *delete_min* are the same as for Fibonacci heaps, so there is no change.

The root list before the *consolidate* operation has the length $t(H) + 1$. As the number of iterations in the *consolidate* operation is bounded by the length of the root list, we obtain a complexity bound for *insert* in $O(t(H))$.

Furthermore, degrees are bound by $D(n)$, so after consolidation there will be at most $D(n)+1$ trees left. Thus, the potential before *insert* is $\Phi(H) = t(H) + 2m(H)$ and after the *insert* it will be $\Phi(H') \le D(n) + 1 + 2m(H)$, as the number of marked nodes can only decrease. Hence the amortised complexity for *insert* becomes

$$c \cdot t(H) + c(\Phi(H') - \Phi(H)) \le cD(n) + c \in O(D(n)) = O(\log n) \ .$$

Analogously, for the *union* operation the root list before the *consolidate* operation has the length $t(H_1) + t(H_2)$. As the number of iterations in the *consolidate* operation is bounded by the length of the root list, we obtain a complexity bound for *insert* in $O(t(H_1) + t(H_2)) = O(\max\{t(H_1), t(H_2)\})$.

Furthermore, degrees are bound by $D(n)$, so after consolidation there will be at most $D(n)+1$ trees left. Thus, the potential before *union* is $\Phi(H_1) + \Phi(H_2) = t(H_1) + 2m(H_1) + t(H_2) + 2m(H_2)$ and after the *insert* it will be $\Phi(H') \le D(n) + 1 + 2m(H_1) + 2m(H_2)$. Hence the amortised complexity for *union* becomes

$$c(\cdot t(H_1) + t(H_2)) + c(\Phi(H') - \Phi(H_1) - \Phi(H_2)) \le cD(n) + c \in O(D(n)) = O(\log n) \ .$$