University of Freiburg
Dept. of Computer Science
Prof. Dr. F. Kuhn
S. Faour, M. Fuchs, A. Mályusz

Algorithm Theory
Sample Solution Exercise Sheet 7

Due: Friday, 6th of December, 2024, 10:00 am

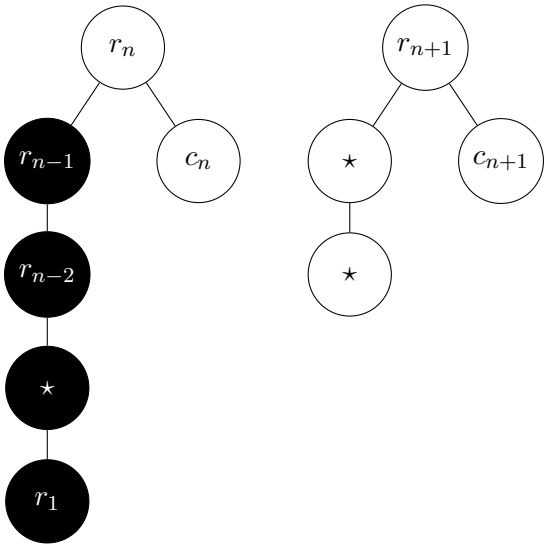
Exercise 1: Worst Case Decrease *(4 Points)*

We've seen in the lecture that Fibonacci heaps are only efficient in an *amortized* sense. However, the time to execute a single, individual operation can be large. Show that in the worst case, the `decrease-key` operation can require time $\Omega(n)$ (for any heap size n).

Hint: Describe an execution in which there is a decrease-key operation that requires linear time.

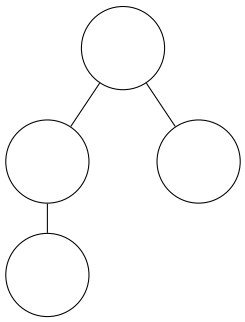
Sample Solution

A costly *decrease-key* operation:
We construct a degenerated tree. Assume we already have a tree T_n in which the root r_n has two children r_{n-1} and c_n, where c_n is unmarked and r_{n-1} is marked and has a single child r_{n-2} that is also marked and has a single child r_{n-3} and so on, until we reach a (marked or unmarked) leaf r_1. In other words, T_n consists of a line of marked nodes, plus the root and one further unmarked child of the root. We give the root r_n some key k_n.
We now add another 5 nodes to the heap and delete the minimum of them, causing a *consolidate*. In more detail let us add a node r_{n+1} with key $k_{n+1} \in (0, k_n)$, one with key 0 and 3 with keys $k' \in (k_{n+1}, k_n)$. When we delete the minimum, first both pairs of singletons are combined to two trees of rank 1, which are combined again to one binomial tree of rank 2, with the node r_{n+1} as the root and we name its childless child c_{n+1} (confer the picture for the current state).



Since also T_n has rank 2 we now combine it with the new tree and r_{n+1} becomes the new root. We now decrease the key of c_n to 0 as well as the keys of the two unnamed nodes and delete the minimum after each such operation, as to cause no further effect from *consolidate*. Decreasing the key of c_n, however, will now mark its parent r_n, as it is not a root anymore. Thus the remaining heap is of

exactly the same shape as T_n, except that its depth did increase by one: a T_{n+1}.
Can we create such trees? We sure can by starting with an empty heap, adding 5 nodes, deleting one, resulting in a tree of the following form:



We cut off the lowest leaf and now have a T_1. The rest follows via induction.
Obviously, a *decrease-key* operation on r_1 will cause a cascade of $\Omega(n)$ cuts if applied to a heap consisting of such a T_n.

Exercise 2: Fibonacci Heaps Modifications - Amortized I *(5 Points)*

Suppose we "simplify" Fibonacci heaps such that we do *not* mark any nodes that have lost a child and consequentially also do *not* cut marked parents of a node that needs to be cut out due to a `decrease-key`-operation. Is the *amortized* running time

(a) ... of the `decrease-key`-operation still $\mathcal{O}(1)$? *(1 Point)*

(b) ... of the `delete-min`-operation still $\mathcal{O}(\log n)$? *(4 Points)*
Hint: Can we still guarantee the recursive property (proved in the lecture) i.e. a given node with rank i has i children that have at least ranks $i-2, i-3$, ..., respectively?

Explain your answers.

Sample Solution

Two reasonable answers would be as follows.

(a) Yes. Not having to cut all your marked ancestor nodes only makes `decrease-key` faster. In fact each individual `decrease-key` operation has now runtime $\mathcal{O}(1)$. *(2 Points)*

(b) No. The reason is that we loose the recursive property that a given node with rank i has i children that have at least ranks $i-2, i-3, ...$, respectively. This was required to show that each tree of a given rank has a *minimum size* of F_{i+2} (where F is the Fibonacci series) which grows exponential in i. Consequentially the maximum rank can not be too large, just $\mathcal{O}(\log n)$, as a tree with higher rank would require more than n nodes.

Now, if a node can loose an arbitrary number of children without being cut, the above property can not be guaranteed anymore. In particular, in extreme cases we could end up with a tree with rank $n-1$. Since `delete-min` has amortized runtime linear in the maximum rank, it will have a higher amortized running time (i.e., $\omega(\log n)$). *(4 Points)*

Exercise 3: Fibonacci Heaps Modifications - Amortized II *(11 Points)*

(a) Assume that operation `decrease-key` never occurs. Show that in this case, the maximum rank $D(n)$ of a Fibonacci heap is at most $\lfloor \log_2(n) \rfloor$. *(4 Points)*

(b) We want to augment the Fibonacci heap data structure by adding an operation `increase-key`(v, k) to increase the key of a node v (given by a direct pointer) to the value k. The operation should have an amortized running time of $\mathcal{O}(\log n)$. Describe the operation `increase-key`(v, k) in sufficient detail and prove the correctness and amortized running time. *(7 Points)*

Remark: You can use the same potential function as for the standard Fibonacci heap data structure. Note however that after conducting ***increase-key****(v, k) the Fibonacci heap must still be a list of heaps, where the maximum rank $D(n) \in O(\log n)$.*

Sample Solution

(a) First we show inductively that when there are *no* `decrease-key` operations, then a heap of rank i in the rootlist has exactly 2^i nodes (we call the number of nodes the size of the heap in the following). A heap of rank 0 is just a single node, thus it has size $2^0 = 1$.

Given a heap h of rank $i > 0$ which might also be a sub-heap attached to some parent node. The only way the degree i of h can be created is by linking two heaps h_1, h_2 of rank $i-1$. By induction hypothesis heaps of rank $i-1$ have 2^{i-1} elements. Therefore, the size of the heap $h = \texttt{link}(h_1, h_2)$ is the sum of the sizes of h, h_2, i.e., $2^{i-1} + 2^{i-1} = 2^i$.

Remark: When we execute a ***delete-min*** *operation, then smaller heaps that are attached to the current minimum are cut and reinserted into the rootlist. But this does not change the the form of the subheaps of the root in any way, so the induction argument above remains valid.*

Since we have only n nodes in the Fibonacci heap in total, the heap with the biggest rank $D(n)$ must fulfill the inequality

$$2^{D(n)} \leq n \Longleftrightarrow D(n) \leq \log_2 n.$$

Since $D(n)$ is an integer value we have $D(n) \leq \lfloor \log_2 n \rfloor$.

(b) **Implementation and correctness arguments**: As suggested in the remark, we try to design the `increase-key`(v, k) operation to maintain the same conditions of the Fibonacci heap. Specifically, we ensure in the following that each node looses at most one rank by loosing a child. First we assert that for `increase-key`(v, k) the new key k is larger than the current key of v. If k is smaller than or equal to all the keys of its child nodes, the heap condition is not violated after changing the key to k and we do nothing else.

Otherwise we first cut out and reinsert all child-heaps of v into the rootlist. Since v has lost too many children (each node can loose at most one) we also cut v from its parent and reinsert it as single node into the rootlist. Since v's former parent now lost a child, we run the cascading cut procedure on v's former parent, meaning that all successive marked ancestors of v are cut out and reinserted into the rootlist. The closest previously unmarked ancestor of v is marked.

Finally we have to consider a special case that forces us to do another step. If the node v whose key we increased is the current minimum, then we have to go through the whole rootlist to find the new minimum (or to confirm that v is still the minimum). But then we also have to run a consolidate like for `delete-min`. The reason for that is technical: we have to shrink the size of the rootlist R back down to $D(n)$ in order to "pay" that costly search (and consolidate) with the associated decrease in potential.

Runtime: The *actual cost* of our implementation of `increase-key`(v, k) is composed of the following components. We have $t_1 \leq D(n)$ steps for cutting and reinserting all child-heaps of v, since $D(n)$ is the maximum number of children v can have.

The next costly step is the cascading cuts procedure, which takes t_2, where t_2 is the number of successively marked ancestors of v plus one or, alternatively, the distance to the closest unmarked ancestor of v.

Finally, let us assume that we actually increase the key of current the minimum v. Then we have to find a new minimum and also consolidate, which takes time to the order of $t_3 \leq |H.\texttt{rootlist}|$, where R is the size of the rootlist.

The potential of the Fibonacci heap changes as follows:

$$\begin{aligned} R_{new} &= R_{old} + D(n) + 1 - |H.\texttt{rootlist}| \\ M_{new} &= M_{old} - (t_2 - 1) \\ \Phi_{new} &= \Phi_{old} + D(n) + 1 - |H.\texttt{rootlist}| - 2(t_2 - 1). \end{aligned}$$

The difference $\Phi_{new} - \Phi_{old}$ can be used to offset or more precisely *amortise* our true costs $t_1 + t_2 + t_3$:

$$\begin{aligned} a_i &= t_1 + t_2 + t_3 + \Phi_{new} - \Phi_{old} \\ &\leq D(n) + t_2 + |H.\texttt{rootlist}| + \Phi_{new} - \Phi_{old} \\ &= D(n) + t_2 + |H.\texttt{rootlist}| + D(n) + 1 - |H.\texttt{rootlist}| - 2(t_2 - 1) \\ &= 2D(n) + 3 - t_2 \\ &\leq 2D(n) + 3 \in \mathcal{O}(\log n). \end{aligned}$$