Albert-Ludwigs-Universität, Inst. für Informatik
Prof. Dr. Fabian Kuhn
H. Ghodselahi, Y. Maus December 08, 2016

Algorithm Theory, Winter Term 2016/17
Problem Set 4 - Sample Solution

Exercise 1: Amortization (using Accounting) (8 points)

Suppose we perform a sequence of n operations on an (unknown) data structure in which the i-th operation costs i if i is an exact power of 2, and 1 otherwise.

Use the **accounting** method to determine the amortized cost per operation.

Solution:

We impose an extra charge on inexpensive operations which we keep in the so called *bank account*. Then we use this bank account to pay for more expensive operations. It is crucial to show that there is always enough credit on the bank account when we need it to pay for an expensive operation.

To get a picture of how costs are distributed lets take a look at the actual costs of operations.

Operation	1	2	3	4	5	6	7	8	9	...	15	16	17	...
Actual Cost	1	2	1	4	1	1	1	8	1	...	1	16	1	...

Table 1: Operations and their actual costs

We claim that the amortized cost of each operation is 3. Obviously operation i is expensive if and only if i is a power of 2 and the actual cost of every cheap operation is 1. We will store 2 additional units in the bank account for each cheap operation which imposes

$$actual\ cost + bank\ account\ change = 1 + 2 = 3$$

amortized cost for each cheap operation. Let $i = 2^j$ for some $j \in \mathbb{N}$ be an expensive operation. Then we take $2^j - 3$ units from the bank account to pay for its execution which leads to

$$actual\ cost + bank\ account\ change = 2^j - (2^j - 3) = 2^j - (2^j - 3) = 3$$

amortized cost.

It remains to show that the bank account always has sufficiently many credits to pay for any expensive operation. Let $i = 2^j$ be any expensive operation. The number of consecutive operations that are not an exact power of 2 (i.e., cheap operations) and are performed immediately before operation (2^j) is $2^j - 2^{j-1} - 1$. Each of these operations adds 2 units to the bank account. So before the execution of the expensive operation 2^j there are at least $2 \cdot (2^j - 2^{j-1} - 1) = 2^j - 2$ units on the bank account (during those operations nothing is taken from the bank account). Thus, there is enough credit on the account to pay the $2^j - 3$ credits for operation $i = 2^j$.

Exercise 2: Amortization (using Potential Function) (8 points)

We are given a data structure $\mathcal{D}$, which supports the operations `put` and `flush`. The operation `put` stores a data item in $\mathcal{D}$ and has a running time of 1. Further, if $\mathcal{D}$ contains $k \geq 0$ items, the operation `flush` deletes $\lceil k/2 \rceil$ of the k data items stored in $\mathcal{D}$ and its running time is equal to k.

Prove that both operations have constant amortized running time by using the **potential function** method.

Solution

Goal:

$\mathcal{O}(1)$ amortized time per operation.

Intuition:

Flushing every item costs at most 2. Hence, a potential function should be increased by at least 2 whenever we `put` an item. Therefore, we are guaranteed to have enough potential to do a `flush` operation.

Definition of potential function:

A potential function is a function mapping a *possible configuration of the data structure* to a non negative real number. It is important that the potential function can be computed from the status of the data structure. In particular the information about previous operations that have been performed is not necessary to determine its value.

Define $\Phi = 2N$, where N is the current number of elements in the data structure $\mathcal{D}$.

Correctness of potential function:

The above potential function is never less than 0 since the number of elements in $\mathcal{D}$ can not be negative.

Amortized cost for i-th operation:

For the analysis, fix an arbitrary sequence of operations. Then let N_{i-1} denote the number of elements in $\mathcal{D}$ before the i-th operation and N_i the number of elements after the i-th operation [1]. In the following a_i and t_i are amortized and actual costs, respectively, for operation i. For the i-th operation we consider the cases that it is a `put` or a `flush` operation separately:

[1] N_i depends on the type of operation that is performed in step i. Thus N_i differs in the two considered cases

If the i-th operation is `put`, then:

$N_i = N_{i-1} + 1.$
$t_i = 1$ (the actual cost of operation `put`).
$a_i = t_i + \Phi_i - \Phi_{i-1} = 1 + 2N_{i-1} + 2 - 2N_{i-1} = 3 \in \mathcal{O}(1).$

If the i-th operation is `flush`, then:

$N_i = N_{i-1} - \lceil N_{i-1}/2 \rceil.$
$t_i = N_{i-1}$ (the actual cost of operation `flush`) as given in the exercise.
$a_i = t_i + \Phi_i - \Phi_{i-1} = N_{i-1} + (2N_{i-1} - 2\lceil N_{i-1}/2 \rceil) - 2N_{i-1} = N_{i-1} - 2\lceil N_{i-1}/2 \rceil \leq 0 \in \mathcal{O}(1).$

Hence, the amortized costs for both operations `put` and `flush` are constant.

Exercise 3: Fibonacci Heaps (12 points)

Fibonacci heaps are only efficient in an **amortized** sense. The time to execute a single, individual operation can be large. Show that in the worst case, the `delete-min` and `decrease-key` operations can require time $\Omega(n)$ (for any heap size n).

Hint: *Describe an execution in which there is a delete-min operation that requires linear time. Also, describe an execution in which there is a decrease-key operation that requires linear time.*

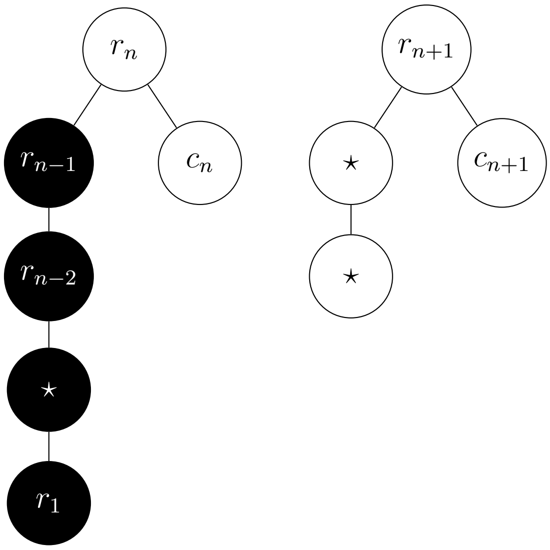
Solution

A costly *delete-min*:

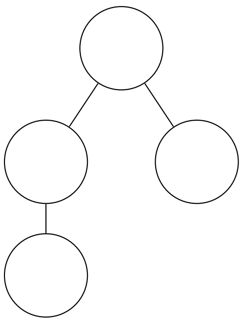
First n elements are added to the heap, which causes them all to be roots in the root list. Deleting the minimum causes a *consolidate* call, which combines the remaining $n-1$ elements, which need at least $n-2$ *merge* operations, i.e., it costs $\Omega(n)$ time.

A costly *decrease-key* operation: (more difficult)

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 4: Union-Find (12 points)

Assume that we are given a union-find data structure which is implemented as a disjoint-set forest. In the lecture, we have seen that when using path compression and the union-by-rank heuristics, the total running time of any m operations is $\Theta(m \cdot \alpha(m, n))$ (where $\alpha(m, n)$ is the inverse of the Ackermann function and n is the number of `make-set` operations).

We now consider any sequence of m union-find operations, where all the `make-set` and `union` operations appear before any of the `find-set` operations. Let f be the number of `find-set` operations. Show that the total running time of the f `find-set` operations is only $\mathcal{O}(f + n)$ if both path compression and union-by-rank heuristics are used. What happens in the same situation if we use only the path compression heuristic (without union-by-rank)?

Remark: In the union-by-rank heuristic, each tree of the disjoint-forest representation has a rank which is computed as follows. When a tree of size 1 is created in a `make-set` operation, its rank is 0. Further, whenever two trees T_1 and T_2 are merged in a `union` operation, the tree of smaller rank is attached to the tree of larger rank. If T_1 and T_2 have different ranks, the rank of the combined tree is equal to the larger of the two ranks of T_1 and T_2. Otherwise, if they both have the same rank, the rank of the combined tree is the rank of the two trees plus 1.

Solution

Observation 1: Using path compression, once a node x appears on a find path, x will be either a root or a child of a root at all times thereafter since all the `union` operations appear before any of the `find-set` operation.

We use the *accounting method* to obtain a constant amortized cost of a `find-set` operation when amortizing over all n `make-set` operations. We charge a `make-set` operation two dollars. One dollar pays for the `make-set`, and one dollar remains on the node x that is created. The latter pays for the first time that x appears on a find path and is turned into a child of a root. We charge the actual cost of a `union` operation as its amortized cost. Obviously, this pays for the actual linking of one node to another. We charge one dollar for a `find-set`. This dollar pays for visiting the root and its child, and for the path compression of these two nodes, during the `find-set`. All other nodes on the find path use their stored dollar to pay for their visitation and path compression. With respect to the Observation 1, after the `find-set`, all nodes on the find path become children of a root (except for the root itself), and so whenever they are visited during a subsequent `find-set`, the `find-set` operation itself will pay for them. Since the credit in our bank account always remains positive, the total actual cost of all operations is upper bounded by the total amortized cost of all operations. The total actual cost of all `union` operations equals the total amortized cost of them. Hence, the total actual cost of all `make-set` and `find-set` operations is at most $2n + f$ since the number of `make-set` operations is n. Consequently, the total actual cost of all `find-set` operations is $\mathcal{O}(n + f)$.

Observe that nothing in the above argument requires union by rank. Therefore, we get an $\mathcal{O}(f + n)$ time bound regardless of whether we use union by rank.