University of Freiburg
Dept. of Computer Science
Prof. Dr. F. Kuhn
P. Bamberger, P. Schneider

Algorithm Theory
Sample Solution Exercise Sheet 5

Due: Tuesday, 23rd of November, 2021, 4 pm

Exercise 1: Stack with Backup *(8 Points)*

Consider an (initially empty) stack S where every k operations the content of S is copied for backup purposes which takes time $\Theta(|S|)$.

(a) Given an *arbitrary* series of push and pop operations on S, what is the amortized cost of an operation? *(2 Points)*

(b) Assume we conduct a series of push and pop such that the size of S never exceeds N. Assign amortized running times (possibly depending on N and k) to the operations that are as small as possible. Prove your claim. *(6 Points)*

Sample Solution

(a) The amortized cost per push operation in an arbitrarily long sequence of push operations is unbounded since S can be arbitrarily large and thus the backup operation arbitrarily expensive.

A bit more precisely, we can argue as follows. Assume that we conduct n operations. In the worst case there are only push operations, which make the stack largest and thus the corresponding backups most expensive. The total cost for the push and backup operations is

$$n + \sum_{i=1}^{\lfloor n/k \rfloor} k \cdot i = n + k \cdot \sum_{i=1}^{\lfloor n/k \rfloor} i = n + k \cdot \frac{\lfloor n/k \rfloor \cdot \lfloor n/k + 1 \rfloor}{2} \overset{\text{for a } c>0}{\geq} c \cdot n^2/k.$$

So the amortized cost per operation is at least $c \cdot n/k$ (which is unbounded in n).

Remark: In particular, this means that in part (b) we will not be able to give an amortized time for push that is constant in N, the best we can hope for is $c \cdot N/k$ for some $c > 0$.

(b) W.l.o.g., we assume that the true cost of push and pop is 1 and that of backup is $|S|$ (one can always adjust these constants by some factor). We assign amortized cost of 1 to the pop operation and a cost of $1 + 2\frac{N+1}{k}$ to the push operation. We use the accounting method to show that this is a correct amortized cost assignment.

Let us first see how the account balance changes with given the amortized above. Consider the case that the current operation does *not* trigger a backup. Then the amortized cost of 1 for pop is sufficient to pay for its true cost and the account does not change. For a push operation without backup, we use 1 to pay for the true cost and pay $2\frac{N+1}{k}$ to the bank account. If there *is* a backup after an operation we take $|S|$ from the account to pay for it (after doing the same as above).

It remains to show that the account is always positive. Let us assume that there are no pop operations on an empty stack (we argue how to deal with that in a remark at the bottom). Then at any given operation in our sequence there were always at least as many pushes as pops in that subsequence, since S is initially empty.

Assume we just executed the n^{th} operation. On one hand, the total number of backups that have occurred so far is at most $\lceil n/k \rceil$,[1] with a total amount of at most $N \cdot \lceil n/k \rceil$ subtracted from the account. On the other hand there were at least $\lceil n/2 \rceil$ many `push` operations. Thus the amount we paid to the account is least

$$\lceil \tfrac{n}{2} \rceil \cdot 2 \cdot \tfrac{N+1}{k} \geq n \cdot \tfrac{N+1}{k} = (N+1) \cdot \tfrac{n}{k} \geq N \cdot \lceil \tfrac{n}{k} \rceil,$$

i.e., at least as much as we subtracted. Since this holds after every operation, the account always stays positive.

Remark: The only problem with `pop` operations on an empty stack is that the sequence could contain more `push`'s than `pop`'s, which we needed to exploit in our analysis above. Note that whenever we have such a sequence of `pop` operations on an empty stack, then the backups occurring during this sequence are also cheap.

In particular, we assumed that backup is for free in that situation as $|S| = 0$, so we need not subtract anything from the account (however, in case we want that backup always has a minimum constant cost, we can just increase the amortized cost of `pop` by that constant to pay for it). This means that backups during periods where we have only `pop` operations on an empty stack are taken care of (i.e., the account stays positive).

Now consider the period that starts after *a `pop` operation on an empty stack that is followed by a `push` and ends with the operation* before *the next `pop` operation on an empty stack. For this period we can guarantee that there are more `push`'s than `pop`'s and we simply apply the analysis above to show that the account stays positive during these periods as well.*

Exercise 2: Dynamic Array with Remove *(12 Points)*

In the lecture we saw a dynamically growing array that implements the `append` operation in amortized $\mathcal{O}(1)$ (reads/writes). For an array of size N that is already full, the `append` operation allocates a new array of size $2N$ ($\beta = 2$) before inserting an element at the first free array entry.

Additionally, we introduce another operation `remove`, which writes `Null` into the last non-empty entry of the dynamic array. However, by appending many elements and subsequently removing most of them, the ratio of unused space can become arbitrarily high. Therefore, when the dynamic array of size N contains $\frac{N}{4}$ or less elements after `remove`, we copy each element into a new array of size $\frac{N}{2}$.

(a) Given an array of size N which has at least $\frac{N}{2}$ elements, show that any series of `remove` operations has an amortized cost of $\mathcal{O}(1)$ (reads/writes) per operation. *(4 Points)*

(b) Use the potential function method to show that any series of `append` and `remove` operations has amortized cost of $\mathcal{O}(1)$ (reads/writes) per operation. Assume that the number of elements n in the array is initially 0 and assume that the array never shrinks below its initial size N_0 (we stop allocating smaller arrays in that case). *(8 Points)*

Remarks: You may assume that N is always a power of two. You may also assume that allocating an empty array is free, only copying elements costs one read and one write for each copied element. If you do part (b) absolutely correctly you automatically receive all points for part (a).

Sample Solution

(a) We pay 2 coins to the bank for each `remove` operation (1 coin = 1 read/write operation). When the number of elements in the array reaches $\frac{N}{4}$ (down from $\frac{N}{2}$) we conducted exactly $\frac{N}{4}$ `remove` operations and have $2 \cdot \frac{N}{4} = \frac{N}{2}$ coins on our bank account.

[1]Note that this is true even if we assume that a backup can occur with the very first operation. In fact, if the first backup happens only after k operations we could round down to $\lfloor n/k \rfloor$.

We use this amount to copy (one read and subsequently one write) each element into the new, smaller array for "free" (we make $\frac{N}{4}$ reads and $\frac{N}{4}$ writes). Each `remove` operation costs 1 coin (1 write) plus 2 coins (paid to bank) which makes an amortized cost of 3 coins $= \mathcal{O}(1)$ (read/writes).

Afterwards, the newly allocated array is again half full and the argument can be repeated.

(b) We define a potential function $\Phi(n, N)$ that is small when it contains exactly $\frac{N}{2}$ elements, and large when the number of elements approaches N or $N/4$ respectively, in order to pay for the imminent increase or decrease of the array size:

$$\Phi(n, N) := 2 \cdot |2n - N|.$$

Since we use absolute values, we obviously have $\Phi(n, N) \geq 0$ for any values n, N. For the amortized costs we make some case distinctions. We start with the amortized cost $a_{n,N}$ of `append`:

<u>Case $n = N$:</u>

$$\begin{aligned} a_{N,N} &= 1 + 2N + \Phi(N{+}1, 2N) - \Phi(N, N) \\ &= 1 + 2N + 2(2(N{+}1) - 2N) - 2(2N - N) = 5 \end{aligned}$$

<u>Case $n < N$, $n \geq N/2$:</u>

$$\begin{aligned} a_{n,N} &= 1 + \Phi(n{+}1, N) - \Phi(n, N) \\ &= 1 + 2(2(n{+}1) - N) - 2(2n - N) = 1 + 4 = 5 \end{aligned}$$

<u>Case $n < N$, $n < N/2$:</u>

$$\begin{aligned} a_{n,N} &= 1 + \Phi(n{+}1, N) - \Phi(n, N) \\ &= 1 + 2(N - 2(n{+}1)) - 2(N - 2n) = 1 - 4 \leq 0 \end{aligned}$$

Now the amortized cost $r_{n,N}$ of `remove`:

<u>Case $n = N/4{+}1$:</u>

$$\begin{aligned} r_{N/4+1,N} &= 1 + N/2 + \Phi(N/4, N/2) - \Phi(N/4{+}1, N) \\ &= 1 + N/2 + 2(N/2 - 2(N/4)) - 2(N - 2(N/4{+}1)) = 1 + N/2 - N + 4 \leq 5 \end{aligned}$$

<u>Case $n > N/4{+}1$, $n \leq N/2$:</u>

$$\begin{aligned} r_{n,N} &= 1 + \Phi(n{-}1, N) - \Phi(n, N) \\ &= 1 + 2(N - 2(n{-}1)) - 2(N - 2n) = 1 + 4 = 5 \end{aligned}$$

<u>Case $n > N/4{+}1$, $n > N/2$:</u>

$$\begin{aligned} r_{n,N} &= 1 + \Phi(n{-}1, N) - \Phi(n, N) \\ &= 1 + 2(2(n{-}1) - N) - 2(2n - N) = 1 - 4 \leq 0 \end{aligned}$$