DM582 Exercises - Sheet 7

Mads Anker Nielsen Tobias Samsøe Sørensen

March 19, 2026

This document contains exercises from the course DM582 (spring 2025). Most exercises are from the book *Introduction to Algorithms, 4th edition* by Cormen, Leiserson, Rivest, and Stein (CLRS), the book *Algorithm Design, 1st edition* by J. Kleinberg and E. Tardos (KT), and the book *Discrete Mathematics and its Applications, 8th edition* by K. Rosen.

The solutions given here might differ from the solutions discussed in class. In class, we place more emphasis on the intuition leading to the correct answer. Please do not consider reading these solutions an alternative to attending the exercise classes.

References to CLRS refer to the book *Introduction to Algorithms, 4th edition* by Cormen, Leiserson, Rivest, and Stein.

References to KT refer to the book *Algorithm Design, 1st edition* by J. Kleinberg and E. Tardos.

References to Rosen refer to the book *Discrete Mathematics and its Applications, 8th edition* by K. Rosen.

References to BG refer to the book *Computer Algorithms: Introduction to Design and Analysis, 3rd edition* by Sara Baase and Allen Van Gelder.

This document will inevitably contain mistakes. If you find some, please report them to your TA so that we can correct them.

Sheet 7

Exercise 16.1-2

Exercise

Show that if a `DECREMENT` operation is included in the bit counter example, n operations can cost as much as $\Theta(nk)$ time.

Suggested solution

Let $A[k-1] = 1$ and $A[i] = 0$ for $i = 0, 1, \ldots, k-2$. That is, the value of the counter is 2^{k-1}. Now, consider the sequence of operations $\sigma = \langle \texttt{DECREMENT}, \texttt{INCREMENT} \rangle^n$ of length $2n$. The first pair of a `DECREMENT` and an `INCREMENT` operation will flip all bits (twice) and return the counter to its original value. This process repeats n times. Thus, the total running time is $\Theta(2nk)$.

Exercise 16.3-5

Exercise

Show how to implement a queue with two ordinary stacks (Exercise 10.1-7) so that the amortized cost of each `ENQUEUE` and each `DEQUEUE` operation is $O(1)$.

Suggested solution

We may implement the queue as follows. Let S_1 and S_2 be the two stacks. When we perform an `ENQUEUE` operation, we simply push the element onto S_1. When we perform a `DEQUEUE` operation, we first check if S_2 is empty. If it is, we pop all elements from S_1 and push them onto S_2. Then we pop the top element from S_2.

We assume that each `PUSH` and each `POP` operation on a stack has actual cost 1. The actual cost of the `ENQUEUE` operation is thus 1 and the actual cost of the `DEQUEUE` operation is 1 if S_2 is not empty and $2|S_1|$ if S_2 is empty.

We define the potential $\Phi_i = 2|S_1^i|$ where $|S_1^i|$ is the number of element on S_1 after the i-th operation. We have $\Phi_0 = 0$ and $\Phi_i \geq 0$ for all i. We now determine the amortized cost $\hat{c}_i$ of the i-th operation. If the i-th operation is an `ENQUEUE` operation, we have $\hat{c}_i = c_i + \Delta\Phi_i = 1 + 2 = 3$. If the i-th operation is a `DEQUEUE` operation, we have $\hat{c}_i = c_i + \Delta\Phi_i = 1 + 0 = 1$ when S_2 is not empty and $\hat{c}_i = c_i + \Delta\Phi_i = 2|S_1^{i-1}| - 2|S_1^{i-1}| = 0$ when S_2 is empty.

In any case, the amortized cost of the i-th operation is $O(1)$.

Exercise 16.3-3

Exercise

Consider an ordinary binary min-heap data structure supporting the instructions `INSERT` and `EXTRACT-MIN` that, when there are n items in the heap, implements each operation in $O(\log n)$ worst-case time. Give a potential function such that the amortized cost of `INSERT` is $O(\log n)$ and the amortized cost of `EXTRACT-MIN` is $O(1)$, and show that your potential function yields these amortized time bounds. Note that in the analysis, n is the number of items currently in the heap, and you do not know a bound on the maximum number of items that can ever be stored in the heap.

Suggested solution

We use the following idea. Suppose $f(n)$ is the runtime of the `EXTRACT-MIN` operation on a heap with n items. If we define $\Phi_i = \sum_{k=1}^{n} f(k)$, then the potential drops by $f(n)$ when an `EXTRACT-MIN` operation is performed. Thus, the cost incurred by an `EXTRACT-MIN` operation is exactly cancelled out by the change in potential and the amortized cost of the operation is $O(1)$. Furthermore, it is okay to increase the potential by $f(n)$ on each `INSERT` operation since $f(n) = O(\log n)$ and thus the amortized cost of the `INSERT` operation is $O(\log n) + O(\log n) = O(\log n)$.

The above is nearly the entire argument. Formally, define $\Phi_i = \sum_{k=1}^{n} f(k)$ where n is the number of items in the heap after the i-th operation. We have $\Phi_0 = 0$ and $\Phi_i \geq 0$ for all i since f is a non-negative function (it is the number of steps performed by an algorithm). Also, if the i-th operation is `INSERT`, then $\hat{c}_i = c_i + \Delta\Phi_i = O(\log n) + f(n) = O(\log n)$. If the i-th operation is an `EXTRACT-MIN` operation we have $\hat{c}_i = c_i + \Delta\Phi_i = f(n) - f(n) = 0 = O(1)$, which is what we wanted.

Exercise 16.3-2

Exercise

Redo Exercise 16.1-3 using a potential method of analysis.

Exercise 16.1-3: *Use aggregate analysis to determine the amortized cost per operation for a sequence of operations on a data structure in which the i-th operation costs i if is an exact power of 2, and 1 otherwise.*

Suggested solution

Note: Since this is the first exercise in which we use the potential method, we give an explanation which is much more verbose than needed in general. All we need in general is to define a potential function Φ, show that $\Phi_0 \leq \Phi_n$ for all $n \geq 0$ (we usually take $\Phi_0 = 0$), and then determine the amortized cost of the i-th operation to obtain the desired result.

It seems reasonable to guess that the amortized cost of n operations is $O(n)$ since, intuitively, most operations have constant cost while the costly operations are rare. We now try to prove this.

First, let us informally consider what we need the potential function to be. If we want the amortized cost of n operations to be $O(n)$, we need the amortized cost of each operation to be $O(1)$. When i is a power of 2, the cost c_i of the operation is i, so we need the potential function to change by $-i$ (possibly plus some constant) in order to the amortized cost to be constant. When i is not a power of 2, the cost c_i of the operation is 1, so we really don't need the potential function to change at all in order for the amortized cost to be constant. However, if we do not increase the potential, we cannot make the potential drop by i when i is a power of 2 without the potential becoming negative.[1] Thus, we need the potential function to increase. However, we cannot increase the potential function by more than a constant if we want the amortized cost to be constant.

With this in mind, we come up with the potential function $\Phi_0 = 0$ and $\Phi_i = 2(i - 2^k)$ for $i \geq 1$, where k is the largest integer such that $2^k \leq i$.

We briefly recall why the potential function method is sound. By definition, the amortized cost of the i-th operation is $\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}$ where c_i is the actual cost of the i-th operation. The total amortized cost of a sequence

[1] What we need is $\Phi_n \geq \Phi_0$ and we can always (and almost always do) define Φ such that $\Phi_0 = 0$

of n operations is then

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \Phi_n - \Phi_0.$$

If $\Phi_n - \Phi_0 \geq 0$ (i.e. $\Phi_n \geq \Phi_0$), then the amortized cost n operations is at least the actual cost of the same n operations. Thus, an upper bound on the amortized cost of performing n operations is an upper bound on the actual cost of performing n operations, which is what we want to derive.

In our case, we have $\Phi_0 = 0$ and $\Phi_i \geq 0$ for all i. Thus, all we have left to do is prove that $\hat{c}_i = O(1)$ for all i.

Let $\Delta\Phi_i = \Phi_i - \Phi_{i-1}$. There are 2 cases to consider. If i is not a power of 2, then the actual cost of the i-th operation is $c_i = 1$ and

$$\begin{aligned}\Delta\Phi_i &= 2(i - 2^k) - 2(i - 1 - 2^k) \\ &= 2i - 2^{k+1} - 2i + 2^{k+1} + 2 \\ &= 2,\end{aligned}$$

where we use that k takes the same value for both Φ_i and Φ_{i-1} since i is not a power of 2. In conclusion, $\hat{c}_i = c_i + \Delta\Phi_i = 1 + 2 = 3$.

If $i = 2^k$ for some natural number k, then the actual cost of the i-th operation is $c_i = i = 2^k$ and

$$\begin{aligned}\Delta\Phi_i &= 2(i - 2^k) - 2(i - 1 - 2^{k-1}) \\ &= 2i - 2^{k+1} - 2i + 2^k + 2 \\ &= -2^k + 2\end{aligned}$$

where we use that $\Phi_{i-1} = 2(i - 2^{k-1})$ since i is a power of 2. In conclusion, $\hat{c}_i = c_i + \Delta\Phi_i = 2^k - 2^k + 2 = 2$.

Thus, the total amortized cost of n operations is at most $3n$, which is an upper bound on the actual cost of performing n operations. Thus, the actual cost performing n operations is $O(n)$ which is what we wanted.

Exercise 16.4-3

Exercise

Discuss how to use the accounting method to analyze both the insertion and deletion operations, assuming that the table doubles in size when its load factor exceeds 1 and the table halves in size when its load factor goes below 1/4.

Suggested solution

We give a semi-formal argument that charging 3 for insertions and 2 for deletions suffices.

Every time an insert operation is performed, we pay 1 credit immediately to perform the operation and save 2 for the next expansion.

Every time a delete operation is performed, we pay 1 credit immediately to perform the operation and save 1 for the next contraction.

First, consider an insert operation that causes an expansion and let n be the size of the table after the insertion. Since the table doubles in size every time, there has been at least $n/2$ insertion operations since the last expansion, and we have saved 2 credit for each operation. Thus, we have at least n credit saved up to perform the expansion, which is exactly the cost of the operation.

Next, consider a deletion operation that causes a contraction and let n be the size of the table after the delete operation. We observe that whenever a contraction or expansion is performed, the table is exactly half full after the operation. Since there must have been at least one expansion before any contraction, the table must have been at least half full at some point. Thus, there must have been at least $n/4$ deletion operations since the last contraction/expansion. Thus, we have at least $n/4$ credit saved up to perform the contraction, which is exactly the cost of the operation.

Exercise 16.4-4

Exercise

Suppose that instead of contracting a table by halving its size when its load factor drops below 1/4, you contract the table by multiplying its size by 2/3 when its load factor drops below 1/3. Using the potential function

$$\Phi(T) = |2(T.num - T.size/2)|$$

show that the amortized cost of a `TABLE-DELETE` that uses this strategy is bounded above by a constant.

Suggested solution

Let s_i be the size of the table after the i-th operation and let n_i be the number of elements in the table after the i-th operation.

We think of deletion and contraction as two different operations. Whenever a deletion causes a contraction, we think of this as first performing the contraction and then performing the deletion.

Now, rewrite the potential function as $\Phi_i = |2n_i - s_i|$. When a deletion is performed, we have $s_i = s_{i-1}$ and $n_i = n_{i-1} - 1$. When a contraction is performed, we have $s_i = \frac{2}{3}s_{i-1}$ and $n_i = n_{i-1}$.

For a deletion, the amortized cost is

$$\begin{aligned}
\hat{c}_i &= 1 + |2n_i - s_i| - |2n_{i-1} - s_{i-1}| \\
&= 1 + |2n_i - s_i| - |2(n_i + 1) - s_i| \\
&= 1 + |2n_i - s_i| - |2n_i + 2 - s_i| \\
&\leq 1 + |2n_i - s_i| - |2n_i - s_i| + 2 \\
&= 3.
\end{aligned}$$

In order to analyze the case where a contraction is performed, we start by observing the following. We have $s_{i-1} = 3n_i$ since this is the criterion for a contraction. Also, we have $s_i = \frac{2}{3}s_{i-1} = \frac{2}{3}3n_i = 2n_i$.

Now the amortized cost of a contraction is

$$\begin{aligned}
\hat{c}_i &= n_i + |2n_i - s_i| - |2n_{i-1} - s_{i-1}| \\
&= n_i + |2n_i - 2n_i| - |2n_i - s_{i-1}| \\
&= n_i - |2n_i - s_{i-1}| \\
&= n_i - |2n_i - \frac{3}{2}2n_i| \\
&= n_i - |2n_i - 3n_i| \\
&= n_i - |-n_i| \\
&= n_i - n_i = 0
\end{aligned}$$

which is what we wanted.