Tutorial 2

Data Structures 1, 28. 2. 2025 `https://iuuk.mff.cuni.cz/~chmel/2425/ds1en/`

Exercise 1 (Stretching arrays)
During the lecture, you have seen that a flexible-sized array can be implemented in amortized constant time by letting the array fill up, and when it is full and we want to insert another element, we copy everything into an array that is double the size.
What would happen, if we changed the capacity C in a different way rather than doubling it every time? (Consider only inserts here.)

a) $C \rightsquigarrow C + k$, where $k \geq 1$ is a constant,

b) $C \rightsquigarrow C^2$,

c) $C \rightsquigarrow k \cdot C$ for a constant $k > 1$.

Solution
Adding a constant: we will show that we cannot get better than linear amortized cost. Let ℓ be the initial array capacity, and let us make $n \gg \ell$ operations, where we assume $n = l + \alpha \cdot k, \alpha \in \mathbb{N}$ for simplicity. Let us just calculate the cost of copying the elements: in total, we do $\sum\limits_{i=0}^{\alpha} \ell + i \cdot k$ operations, which is the sum of the first α elements of an arithmetic sequence, where $\alpha = \frac{n-\ell}{k}$. There is a formula for the sum which yields $\alpha \cdot \frac{\ell+(\ell+\alpha k)}{2} = \frac{n-\ell}{k} \cdot \frac{\ell+n}{2} \in \Omega(n^2)$ (because k, ℓ are constants). Therefore, we get that the sum of n operations has the total cost at least $\Omega(n^2)$, and thus we cannot get better amortized cost that $\Omega(n)$ per operation.
Squaring the capacity: here, our model matters a little bit. The result will be different depending on the time it takes to allocate the new array: whether it is constant-time, or linear-time.
If we can allocate the memory in constant time, we can use the coin method as when doubling. After doubling, we have an array with capacity C^2 that contains only C elements. Therefore, we need to use the remaining $C^2 - C$ inserts to save C^2 coins. We can do that just by saving three coins per insert: one for the insertin itself, and two for copying. As the allocation is constant-time, we only need to worry about the copying, for which we have $2(C^2 - C) \geq C^2$ coins (if C is at least 3).
If we require linear time to allocate the memory, we must realise that we have to pay the cost of the whole array beforehand. In particular, it might happen that we make just the exact number of inserts so that the array must grow, and we then stop right after that. The total cost is then immediately $\Theta(n^2)$, but we have done only $\Theta(n)$ operations, so we cannot have the amortized cost better than linear.
Changing the capacity to a constant multiple: we us a similar coin argument as when doubling. In general, after growing, we have an array of size C where $\frac{1}{k} \cdot C$ cells are filled, and another copy happens after $(1 - \frac{1}{k})C = \frac{k-1}{k}C$ inserts. In total, we must have saved up C coins, but we only have $\frac{k-1}{k}C$ insertions to do so. Thus, we will pay in a way where every insertion must pay for $\frac{C}{\frac{k-1}{k}C} = \frac{k}{k-1}$ copies and also its own insertion. We can do this by requiring that every insertion pays $1 + \frac{k}{k-1} \in \mathcal{O}(1)$ coins, where the first coin pays for the insertion and the other coins are saved up for the next copying.

Exercise 2 (Flexible arrays have potential!)
In the lecture, you have seen the analysis of the doubling flexible arrays. There, you used the aggregate method to argue that doubling the array when full and halving the array when at the quarter of its capacity gives amortized constant time.
Prove the same result using a potential defined as $\phi(\text{array}) = |\frac{\text{capacity of array}}{2} - \text{number of elements in array}|$.

Solution
Any non-resizing operation has constant cost and the potential changes by one at most. If we are resizing, then the potential is either $\frac{\text{capacity of array}}{2}$ if we are growing the array or $\frac{\text{capacity of array}}{4}$ if we are shrinking. Either way, we have a potential that is a constant multiple of the real cost of the operation. Thus, we can multiply the potential definition by a constant and the calculation works out in the end.

Exercise 3 (Bounded-balance-trees and aggregated analysis)
During the lecture, you have shown using a potential that the amortized complexity of all operations on BB[α]-trees is $\mathcal{O}(\log n)$. Prove the same result using aggregate analysis.

Solution
We know that the path to any leaf is always $\mathcal{O}(\log n)$. Thus, each insertion/deletion may only change $\mathcal{O}(\log n)$ nodes in the tree.
Let us consider any particular node in the tree. This node has the size of the subtree denoted by s. Let us consider the evolution of the size of the subtree and the necessary rebuilds. After a node is rebuilt (either because of itself or some of its ancestors), it requires at least $(\frac{1}{2} - \alpha)s$ insertions/deletions in its subtree after which, an operation can happen at the earliest. Thus, each rebuild for size s occurs after $\mathcal{O}(s)$ operations and costs $\mathcal{O}(s)$, thus the total cost after n operations *that touch the node* can be $\mathcal{O}(n)$. In total, each insertion/deletion touches $\mathcal{O}(\log n)$ nodes, and therefore the total cost can be at most $\mathcal{O}(m \log n)$, which yields the required amortized cost.

Bonus exercises

Exercise 4 (Binary counter)
Recall the proof that the binary counter (staring at zero) has amortized constant complexity of adding one. How does the amortized cost change if we allow not only adding one, but also subtracting one?
Is it possible to modify the counter so that the amortized cost remains constant?

Solution
It is $\Theta(\ell)$ in total, as the cost for ℓ operations is roughly $\sum_{k\geq 1} \frac{\ell}{2^k} = 2\ell$, yielding a constant amortized cost.
Subtracting one: not as easy, we can get to some 2^k, and then we can repeat subtracting one and adding one until the end.
We can fix the counter by having two counters: P and N (positive and negative) such that every bit is set to one in at most one of the two. The result would then be $P - N$.
When counting, if adding one, we add one to P, and when subtracting, we add one to N. The invariant is eady to maintain: if we have ones in both P and N on the i-th bit (only one such bit can be new per addition), we just set both bits to zero.
For the amortization, the potential will be again the number of ones in both counters. The real cost is the number of changes from 0 to 1 + the number of changes from 1 to 0, the difference of potential is the number of changes from 0 to 1 - the number of changes from 1 to 0. Thus, the amortized complexity is twice the number of changed bits from 0 to 1. However, each addition to one of the counters changes at most one bit from zero to one, so the amortized cost is constant.

Exercise 5 (Binary counter II)
We are in the same situation as in the previous exercise (Binary counter) and only want to add ones.
Show that when we pay 2^k for swapping the k-th bit (and the 0th bit is the least significant bit), then adding ℓ ones costs $\mathcal{O}(\ell \cdot n)$.

Solution
For ℓ operations, we have $\sum_{i=1}^{\lceil \log \ell \rceil} \frac{2^i \cdot \lfloor \ell \rfloor}{2^i} \leq \ell \cdot \lceil \log \ell \rceil$.

Exercise 6 (A sublinear set)
We have the following implementation of a set (of integers) that supports operations Insert, Lookup.
The set `M` is implemented as an array of arrays, where the array `M[i]` has length 2^i. Every array `M[i]` is either empty or completely full, and every array is sorted in increasing order (though there is no relationship between distinct arrays).
As an example, if we have 9 elements in our set, the arrays `M[0], M[3]` are full and all others are empty (this corresponds to the binary representation of the number 9). See, for example the following:

```
M[0] = {5}
M[1] = {}
M[2] = {}
M[3] = {3, 4, 8, 10, 11, 15, 19, 22}
```

a) What is the complexity of the Lookup operation, that is implemented by binary searching on every full array?

b) The INSERT operation will be implemented as follows: we start by creating an array with the single inserted element. Next, we look at the array `M[0]`. If it is empty, we add our new array there. If it is full, then we merge our array with the array `M[0]` and we continue with the array `M[1]`, until we find an empty array `M[i]`. What is the (amortized) cost of the INSERT operation?

Hint: the previous exercise might be helpful.

Solution a) Worstcase complexity is $\sum_{i=0}^{\log n} \log(2^i) = \sum_{i=0}^{\log n} i = \frac{\log(n)\cdot(\log(n)+1)}{2} \in \Theta(\log^2 n)$.

b) Worstcase, this can be linear, e.g. inserting for $n = 2^k - 1$ means that we have to merge all arrays into one, but using the previous exercise, we can amortize this. One merge costs roughly 2^{k+1}, so we have (up to a constant factor) the same situation as in the Binary counter II. There, we have amortized cost $\mathcal{O}(\text{counter length})$, which is $\log(n)$, which yields amortized logarithmic cost.