18. AMORTIZED ANALYSIS

Alexandra UDRESCU, Mihai-Valentin DUMITRU
alexandra.udrescu01@upb.ro, mihai.dumitru2201@upb.ro

December 2024

In some cases, especially when dealing with data structures, we might want to analyze the complexity of a *sequence of n operations*; this sequence can even be heterogeneous, involving different types of operations, with different asymptotic complexities.

Our motivating example is a **dynamic array** which allows arbitrary amounts of insertions and deletions. The maximum number of elements is not known in advance, so the array should allow for resizing: when the array is filled, a new, larger array is allocated, all elements from the old array are copied to the new one, then the new element is inserted. Deletions are also arbitrary and might occur at any point. If we want to be memory-efficient and not waste space on a barely-populated array, it should also be able to *shrink*: when it gets too few elements, a new, smaller array is allocated, all elements from the old array are copied to the new one, except for the deleted element.

We will first consider a sequence of insertions, no deletions. An array A has two relevant measurements:

1. size(A): the capacity of the array – the maximum number of elements it can store.
2. elems(A): the number of elements currently in the array.

For simplicity, let's assume that initially size(A) $= 1$ and elems(A) $= 0$.

1 Naive resizing

Let us do a naive implementation first: we have an array in which we add elements, until it fills up. When full, in order to insert another element, we create a new array of size(A) $+ 1$ and copy the old array's elements in the new one, then we insert the new item:

Algorithm 18.1 Naive array resizing

```
1: function INSERT(A, new_item)
2:     if elems(A) = size(A) then
3:         new_array ← new array of size size(A) + 1
4:         for i ← 1 to elems(A) do
5:             new_array[i] ← A[i]
6:     new_array[elems(A) + 1] ← new_item
7:     free A
8:     return new_array
```

For the first insertion, we skip the `then` branch and directly insert into the array, but for all subsequent insertions, we will need to copy all the previously inserted elements into a new, larger array. For n inserts, we will need to copy $0 + 1 + 2 + 3 + ... + (n - 1) = \frac{n(n-1)}{2}$ elements in total. A sequence of n insertions thus have a total cost of $\Theta(n^2)$. Can we do better?

2 Doubling the size

The problem with the naive approach is that we end up resizing the array for every insertion, thus incurring a great cost for copying existing elements. Intuitively, we should allow the new array to have much more free space,

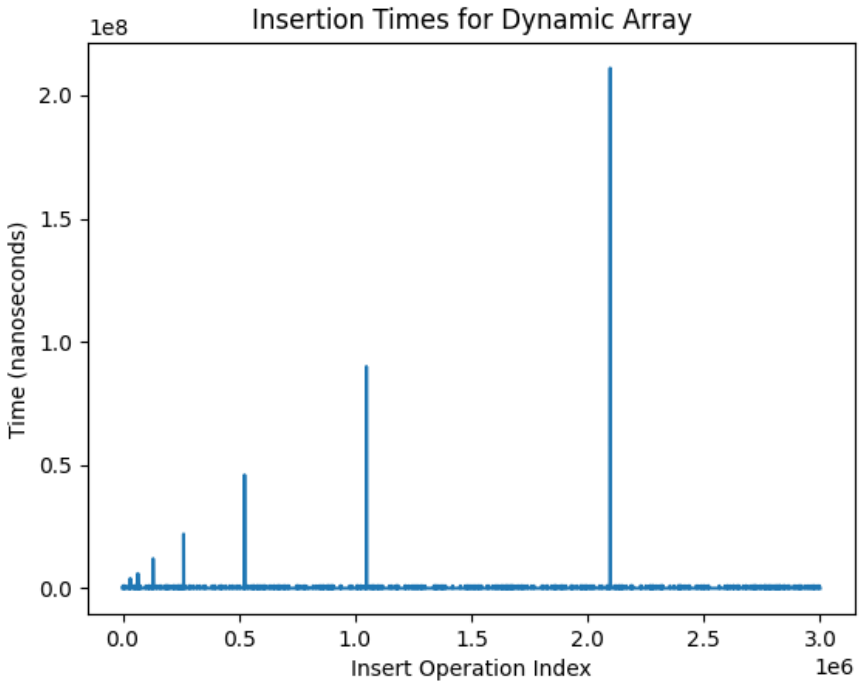

Figure 1: Insertion times for dynamic array

in order to resize less often: if instead we allocated an array with **two extra slots**, then we would need to copy elements only half as often as before; however, this is still asymptotically the same as the $+1$ approach.

When the array is full, we will allocate a new array with **double the size** of the old one.

Algorithm 18.2 Doubling the array when resizing

```
1: function INSERT(A, new_item)
2:     if elems(A) = size(A) then
3:         new_array ← new array of size 2 · size(A)
4:         for i ← 1 to elems(A) do
5:             new_array[i] ← A[i]
6:     new_array[elems(A) + 1] ← new_item
7:     free A
8:     return new_array
```

By doing this, we improve the asymptotic complexity of our sequence of n insertions to $\Theta(n)$. The crux of the "trick" is that most insertion operations are cheap, taking constant time; only once in a while will we get an expensive insertion that requires linear time. Using the doubling strategy, we ensure that this costly operations come so rarely, that their cost is *amortized* over all the cheap operations. In other words, we can consider each insertion to have an *amortized cost* that is constant.

Next we shall formally prove this complexity, by introducing three popular, alternative methods for performing amortized analysis on sequences of data structure operations: the aggregate method, the accounting method, and the potential method.

3 The Aggregate Method

Aggregate analysis computes the worst-case time complexity of an entire sequence, rather than only one operation. As such, given a sequence of n with $T(n)$ worst-case total time, each operation is considered to have an amortized time cost of $T(n)/n$.

Let S be a sequence of n insert operations op_1, op_2, ..., op_n performed on an array. We will examine the aggregate

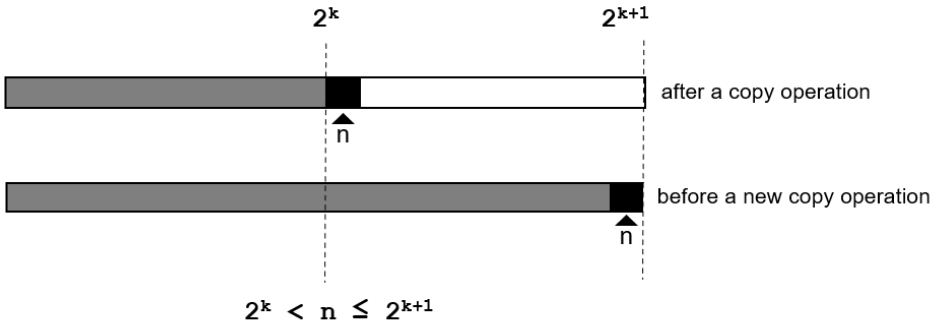

Figure 2: Array - before and after doubling its size

costs associated with these operations, specifically distinguishing between the actual insertion and the cost of copying elements when the array needs to be resized. This analysis is demonstrated in the following table 1, which outlines the costs for a sequence of 9 operations:

	Total cost	**Copying cost**	**Insertion cost**
Operation 1	1	0	1
Operation 2	2	1	1
Operation 3	3	2	1
Operation 4	1	0	1
Operation 5	5	4	1
Operation 6	1	0	1
Operation 7	1	0	1
Operation 8	1	0	1
Operation 9	9	8	1

Table 1: Costs of insert operations in a dynamic array

We can define the cost of sequence S as:

$$\text{cost}(S) = \text{cost}_{\text{insert}}(S) + \text{cost}_{\text{copy}}(S)$$

In this formula, the $\text{cost}_{\text{insert}}(S)$ is always equal to n, the number of operations in the sequence. The difficulty comes from computing $\text{cost}_{\text{copy}}(S)$.

To determine $copy_{cost}(S)$, we note that if k denotes the number of copy operations required following n insert operations, then: $2^{k-1} < n \leq 2^k \Rightarrow k = \lceil \log_2(n) \rceil$. This idea is visually presented in Figure 2.

Considering all these, the cost of the sequence S becomes:

$$\text{cost}(S) = n + \sum_{i=1}^{\lceil \log(n) \rceil - 1} 2^i \leq n + \sum_{i=1}^{\log(n)} 2^i = 3n - 1$$

As such, the amortized cost of an insert operation becomes:

$$\text{cost}(op_i) = \frac{\text{cost}(S)}{n} = \frac{\Theta(n)}{n} = \Theta(1)$$

4 The Accounting Method

The accounting method, also known as the banker's method, assigns a **constant** cost to each operation to represent its average execution time. Certain operations are priced below their actual cost, creating a reserve of funds, or credit. The credit accrued from the less expensive operations is then used to subsidize the more costly ones that arise later on. The base idea is to determine an uniform amortized cost, the same for each operation, such that the surplus left after inexpensive operations is then allocated to offset the cost of more expensive operations that may occur later on in the sequence. It is important to note that an expensive operation cannot consume more than the credit already available from previous operations.

Operation	Amortized cost	Real cost	Credit
			0
insert 3	3	1	2
insert 4	3	1	4
copy	3	4	0
insert 5	3	1	2

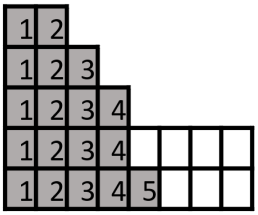

Figure 3: Accounting Method credit example

For an insert operation, we associate an amortized cost $\hat{c}$ such that:

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i$$

(where c_i is the real cost of op_i).

The initial phase involves determining the amortized cost per operation. Given a half-full list (of size 2^k, but with 2^{k-1} elements), which was just resized to double its previous size, we can carry out up to 2^{k-1} insertions before a resize operation is necessary again. When it's time to resize the list again and insert the $(2^{k-1}+1)$-th element, we must replicate the entire existing list — both the elements that were already in place prior to the sequence of insertions, as well as the elements added by the sequence. As such, when adding 2^{k-1} elements, when the list becomes full, we will have to copy 2^k elements. Consequently, each insertion operation should account for:

- adding the current element.
- making a future copy of the element that was just inserted.
- making a future copy of an already existing element.

As such, we can assume that an insert operation has amortized cost $\hat{c} = 3$.

Let us look at an example of how crediting works in the accounting method. Figure 3 presents the values of the credit when performing a sequence of 3 insertion operations in a half-full list of size 4.

At operation $i+1$, the credit can be computed as:

$$\text{credit}_{i+1} = \text{credit}_i + \hat{c} - c_i$$

Subsequently, we need to confirm that the amortized cost is sufficient to cover the actual cost of a sequence of operations.

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c} \Rightarrow \sum_{i=1}^{n} c_i \leq 3n$$

From our earlier aggregate analysis, we have established that $\text{cost}(S) = \sum_{i=1}^{n} c_i = 3n - 1$. Therefore, it is appropriate to conclude that the amortized cost per insertion is indeed 3.

5 The Potential Method

The potential method is akin to the accounting method, but rather than allocating amortized costs, it defines a potential function that tracks the variations in the data structure's credit reserve. As such, cheap operations make the potential grow, while expensive operations consume the accumulated potential of the data-structure. At any given moment in time, the potential difference between the initial state and any subsequent state of a data structure must be non-negative.

We define a potential function $\Phi : State \to \mathbb{N}$ that represents the potential function defined on the states of a data structure.

Let h_0 be the initial state of the data structure. Then the potential function Φ at this initial state is $\Phi(h_0) = 0$. Furthermore, for any state h_t of the data structure occurring during the course of the computation, the potential function Φ is non-negative, that is, $\Phi(h_t) \geq 0$ for all such states h_t.

The potential function serves as a mechanism to monitor the accumulated "precharged time" at any given moment in the computation. Essentially, it quantifies the reservoir of "saved-up time" that can be utilized to offset the cost of time-intensive operations. This concept is similar to maintaining a bank balance in the banker's method of accounting. A key characteristic of the potential function is its dependence solely on the current state of the data structure. This means that the function's value is independent of the computation's historical path that led to the current state.

Using the potential function, the amortized cost for op_i can be calculated using the formula: $\hat{c}_i = c_i + \Phi(h_i) - \Phi(h_{i-1})$, where notations remain the same as in the previous sections.

For our implementation, we need the potential to be maximum right before doubling, when the size is equal to the number of elements to be copied. Then, it should be 0 after doubling the size and copying the elements in the new array. The non-expensive insert operations would grow the potential by 2, so that they can account for a later copy of an already-existing element and the element added. As such, we can define the potential function as:

$$\Phi(A_0) = 0 \tag{1}$$
$$\Phi(A_i) = 2 \cdot \text{elems}(A_i) - \text{size}(A_i) \tag{2}$$

From the way we wrote the implementation, the number of elements in the list is always bigger than half of its size. This means that the chosen potential function yields non-negative values for any state of the array.

Furthermore, in order to find out the amortized cost $\hat{c}$ we define the following situations.

1. **Insertion does not double the array**: $\text{size}(A_i) = \text{size}(A_{i-1})$
   Here, the cost of an insertion is $c_i = 1 + \text{size}(A_{i-1})$.

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(A_i) - \Phi(A_{i-1}) && (3)\\
&= c_i + (2 \cdot \text{elems}(A_i) - \text{size}(A_i)) - (2 \cdot \text{elems}(A_{i-1}) - \text{size}(L_{i-1})) && (4)\\
&= c_i + 2(\text{elems}(A_i) - \text{elems}(A_{i-1})) && (5)\\
&= c_i + 2 && (6)\\
&= 1 + 2 && (7)\\
&= 3 && (8)
\end{aligned}$$

2. **Insertion doubles the array**: $\text{size}(A_i) = 2 \cdot \text{size}(A_{i-1})$. Here, the cost of an insert is $c_i = 1$.

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(A_i) - \Phi(A_{i-1}) && (9)\\
&= c_i + (2 \cdot \text{elems}(A_i) - \text{size}(A_i)) - (2 \cdot \text{elems}(A_{i-1}) - \text{size}(L_{i-1})) && (10)\\
&= c_i + 2(\text{elems}(A_i) - \text{elems}(A_{i-1})) + (\text{size}(A_{i-1}) - \text{size}(A_i)) && (11)\\
&= (1 + \text{size}(A_{i-1})) + 2 - \text{size}(L_{i-1}) && (12)\\
&= 3 && (13)
\end{aligned}$$

6 Amortized deletion

Let us now consider a sequence of both insertion and deletion operations. The naive solution is try to do the "reverse" of our resizing strategy: when the array gets more than half empty ($\text{elems}(A) = \frac{\text{size}(A)}{2}$), we allocate a new one that is half its size and copy all the elements except the deleted one.

However, think about a sequence that contains sufficient insertions to trigger the doubling resizing, followed by a string of alternating deletions and insertions: the array will be resized (and the elements copied to a new copy) for each operation!

The correct solution is to allow the array to get even more depleted than just "half empty".

Think about the accounting method: each "normal" insertion brought enough credit for the insertion itself, as well as the future copying of the inserted element **and** the future copying of another, already-present element.

Similarly, we want to allow "normal" deletions to bring enough credit for the deletion itself, as well as the future copying of an element from the first half of the array. This should also happen right after the array size doubling was triggered; so we need to allow half of the elements to be "normally" deleted, without triggering any copies. We can do this by resizing the array when it gets less than a quarter full (elems$(A) = \frac{\text{size}(A)}{4}$).

7 References and further reading

The content of these notes was adapted from "Introduction to Algorithms", fourth edition, chapter 16[1].

Bibliography

[1] Thomas H Cormen et al. *Introduction to algorithms*. 4th. MIT press, 2022. Chap. 16, pp. 448–475.