# Lecture 11

# Amortized Analysis

*Supplemental reading in CLRS: Chapter 17*

Data structures typically support several different types of operations, each with its own cost (e.g., time cost or space cost). The idea behind amortized analysis is that, even when expensive operations must be performed, it is often possible to get away with performing them rarely, so that the average cost per operation is not so high. It is important to realize that these "average costs" are not expected values—there needn't be any random events.[1] Instead, we are considering the worst-case average cost per operation in a sequence of many operations. In other words, we are interested in the asymptotic behavior of the function

$$C(n) = \frac{1}{n} \cdot \big(\text{worst-case total cost of a sequence of } n \text{ operations}\big)$$

(possibly with some condition on how many times each type of operation may occur). "Worst-case" means that no adversary could choose a sequence of $n$ operations that gives a worse running time.

In this lecture we discuss three methods of amortized analysis: aggregate analysis, the accounting method, and the potential method.

## 11.1 Aggregate Analysis

In **aggregate analysis**, one assumes that there is no need to distinguish between the different operations on the data structure. One simply asks, what is the cost of performing a sequence of $n$ operations, of any (possibly mixed) types?

**Example.** Imagine a stack[2] $S$ with three operations:

- PUSH$(S,x)$ – $\Theta(1)$ – pushes object $x$ onto the stack
- POP$(S)$ – $\Theta(1)$ – pops and returns the top object of $S$

[1] There could be random events, though. It makes sense to talk about the worst-case amortized expected running time of a randomized algorithm, for example: one considers the average expected cost per operation in the string of operations which has the longest expected running time.

[2] A **stack** is a data structure $S$ with two primitive operations: PUSH$(S,x)$, which stores $x$, and POP$(S)$, which removes and returns the most recently pushed object. Thus, a stack is "last in, first out," whereas a queue is "first in, first out."

- MULTIPOP$(S,k)$ – $\Theta(\min\{|S|,k\})$ – pops the top $k$ items from the stack (or until empty) and returns the last item:

```
while S is not empty and k > 0 do
    x ← POP(S)
    k ← k − 1
return x
```

Suppose we wish to analyze the the running time for a sequence of $n$ PUSH, POP, and MULTIPOP operations, starting with an empty stack. Considering individual operations without amortization, we would say that a MULTIPOP operation could take $\Theta(|S|)$ time, and $|S|$ could be as large as $n-1$. So in the hypothetical worst case, a single operation could take $\Theta(n)$ time, and $n$ such operations strung together would take $\Theta\left(n^2\right)$ time.

However, a little more thought reveals that such a string of operations is not possible. While a single POP could take $\Theta(n)$ time, it would have to be preceded by $\Theta(n)$ PUSH operations, which are cheap. Taken together, the $\Theta(n)$ PUSH operations and the one MULTIPOP operation take $\Theta(n)\cdot\Theta(1)+1\cdot\Theta(n)=\Theta(n)$ time; thus, each operation in the sequence takes $\Theta(1)$ time on average. In general, if there occur $r$ MULTIPOP operations with arguments $k_1,\ldots,k_r$, then there must also occur at least $k_1+\cdots+k_r$ PUSH operations, so that there are enough items in the stack to pop. (To simplify notation, we assume that $k$ is never chosen larger than $|S|$.) Thus, in a string of $n$ operations, the total cost of all non-$O(1)$ operations is bounded above by $O(n)$, so the total cost of all operations is $O(n)\cdot O(1)+O(n)=O(n)$. Thus, the amortized running time per operation is $O(1)$.

## 11.2 Accounting Method

Unlike aggregated analysis, the **accounting method** assigns a different cost to each type of operation. The accounting method is much like managing your personal finances: you can estimate the costs of your operations however you like, as long as, at the end of the day, the amount of money you have set aside is enough to pay the bills. The estimated cost of an operation may be greater or less than its actual cost; correspondingly, the surplus of one operation can be used to pay the debt of other operations.

In symbols, given an operation whose actual cost is $c$, we assign an amortized (estimated) cost $\widehat{c}$. The amortized costs must satisfy the condition that, for *any* sequence of $n$ operations with actual costs $c_1,\ldots,c_n$ and amortized costs $\widehat{c}_1,\ldots,\widehat{c}_n$, we have

$$\sum_{i=1}^{n}\widehat{c}_i \geq \sum_{i=1}^{n} c_i.$$

As long as this condition is met, we know that the amortized cost provides an upper bound on the actual cost of any sequence of operations. The difference between the above sums is the total surplus or "credit" stored in the data structure, and must at all times be nonnegative. In this way, the accounting model is like a debit account.

**Example.** Perhaps you have bought pre-stamped envelopes at the post office before. In doing so, you pay up-front for both the envelopes and the postage. Then, when it comes time to send a letter, no additional charge is required. This accounting can be seen as an amortization of the cost of sending a letter:

| Operation | Actual cost $c_i$ | Amortized cost $\widehat{c}_i$ |
|---|---|---|
| Buy an envelope | 5¢ | 49¢ |
| Mail a letter | 44¢ | 0¢ |

Obviously, for any valid sequence of operations, the amortized cost is at least as high as the actual cost. However, the amortized cost is easier to keep track of—it's one fewer item on your balance sheet.

**Example.** For the stack in Section 11.1, we could assign amortized costs as follows:

| Operation | Actual cost $c_i$ | Amortized cost $\widehat{c}_i$ |
|---|---|---|
| PUSH | 1 | 2 |
| POP | 1 | 0 |
| MULTIPOP | $\min\{\|S\|, k\}$ | 0 |

When an object is pushed to the stack, it comes endowed with enough credit to pay not only for the operation of pushing it onto the stack, but also for whatever operation will eventually remove it from the stack, be that a POP, a MULTIPOP, or no operation at all.

## 11.3 Potential Method

The **potential method** is similar in spirit to the accounting method. Rather than assigning a credit to each element of the data structure, the potential method assigns a credit to the entire data structure as a whole. This makes sense when the total credit stored in the data structure is a function only of its state and not of the sequence of operations used to arrive at that state. We think of the credit as a "potential energy" (or just "potential") for the data structure.

The strategy is to define a potential function $\Phi$ which maps a state $D$ to a scalar-valued potential $\Phi(D)$. Given a sequence of $n$ operations with actual costs $c_1, \ldots, c_n$, which transform the data structure from its initial state $D_0$ through states $D_1, \ldots, D_n$, we define heuristic costs

$$\widehat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}).$$

This rule can be seen as the conservation of energy: it says that the surplus (or deficit) cost $\widehat{c}_i - c_i$ caused by a given operation must be equal to the change in potential of the data structure caused by that operation. An operation which increases the potential of the system is assigned a positive heuristic cost, whereas an operation which decreases the potential of the system is assigned a negative heuristic cost.

Summing the heuristic costs of all $n$ operations, we find

$$\begin{aligned}\sum_{i=1}^{n} \widehat{c}_i &= \sum_{i=1}^{n} c_i + \Phi(D_i) - \Phi(D_{i-1}) \qquad \text{(a telescoping sum)} \\ &= \left(\sum_{i=1}^{n} c_i\right) + \Phi(D_n) - \Phi(D_0).\end{aligned}$$

Thus, the total credit stored in the data structure is $\Phi(D_n) - \Phi(D_0)$. This quantity must remain nonnegative at all times in order to ensure that the amortized cost provides an upper bound on the actual cost of any sequence of operations. (Any function $\Phi$ which satisfies this property can be used as the potential function.) One often chooses the potential function $\Phi$ so that $\Phi(D_0) = 0$; then one must check that $\Phi$ remains nonnegative at all times.

**Example.** Continuing the stack example from Sections 11.1 and 11.2, we define the potential of a stack $S$ to be $\Phi(S) = |S|$, the number of elements in the stack. An empty stack has zero potential, and clearly $\Phi$ is always nonnegative, so $\Phi$ is an admissible potential function. Then the heuristic costs of the stack operations are as follows:

- A PUSH operation increases the size of $S$ by 1, and has actual cost $c_{\text{PUSH}} = 1$. Thus, the amortized cost of a PUSH operation is

$$\widehat{c}_{\text{PUSH}} = c_{\text{PUSH}} + \Phi(D_{\text{new}}) - \Phi(D_{\text{old}}) = 1 + (|S_{\text{old}}| + 1) - |S_{\text{old}}| = 2.$$

- A POP operation decreases the size of $S$ by 1, and has actual cost $c_{\text{POP}} = 1$. Thus, the amortized cost of a POP operation is

$$\widehat{c}_{\text{POP}} = c_{\text{POP}} + \Phi(D_{\text{new}}) - \Phi(D_{\text{old}}) = 1 + (|S_{\text{old}}| - 1) - |S_{\text{old}}| = 0.$$

- The operation MULTIPOP$(S,k)$ decreases the size of $S$ by $\min\{|S|,k\}$, and has actual cost $c_{\text{MULTIPOP}} = \min\{|S|,k\}$. Thus, the amortized cost of a MULTIPOP operation is

$$\widehat{c}_{\text{MULTIPOP}} = c_{\text{MULTIPOP}} + \Phi(D_{\text{new}}) - \Phi(D_{\text{old}}) = \min\{|S|,k\} + (|S_{\text{new}}| - |S_{\text{old}}|) = 0.$$

Thus, the amortized costs for this application of the potential method are the same as those we came up with using the accounting method in Section 11.2:

| Operation | Actual cost $c_i$ | Amortized cost $\widehat{c}_i$ |
|---|---|---|
| PUSH | 1 | 2 |
| POP | 1 | 0 |
| MULTIPOP | $\min\{\|S\|,k\}$ | 0 |

## 11.4 Example: A Dynamically Resized Table

We now extend the example we started in Lecture 10: a dynamically doubled table $T$. This time we will forget about keys, inserting each new value into the next available block in the array $T.arr$. If $T.arr$ is full, then we create a new array of size $2|T.arr|$, copy over the old elements of $T$, perform the insertion, and reassign $T.arr$ to point to that array. Likewise, in §11.4.5 we ask you to devise an extension of our data structure which supports deletions (specifically, pops) as well as insertions, and halves the array $T.arr$ when appropriate in addition to doubling it when appropriate. In this way, $T$ never takes up much more space than it has to.

For now, assume that $T$ only supports insertions and lookups. We will store in $T$ two things: the underlying array $T.arr$, and an integer $T.num$ which keeps track of the number of entries in $T.arr$ which have yet been populated. Thus, the array is full and needs to be doubled upon the next insertion if and only if $|T.arr| = T.num$. So the table lookup will be a simple array lookup operation, which takes $O(1)$ time, and the insertion operation will be defined as follows:

**Algorithm:** TABLE-INSERT($T, x$)

```
1  if |T.arr| = 0 then
2       T.arr ← new array of size 1
3  ▷ If table is full, double it
4  if T.num = |T.arr| then
5       arr' ← new array of size 2|T.arr|
6       Migrate all items from T.arr to arr'
7       T.arr ← arr'
8  ▷ Insert x into the array
9  T.arr[T.num] ← x
10 T.num ← T.num + 1
```

### 11.4.1 Actual cost per operation

A lookup operation on $T$ is simply an array lookup, which takes $O(1)$ time. The running time of TABLE-INSERT, on the other hand, depends on whether the table needs to be doubled, which depends on the number of insertions which have happened prior to the current one. Specifically, starting from an initial table of size zero, the cost $c_i$ of the $i$th insertion is

$$c_i = \begin{cases} 1 & \text{if the table has space} \\ \Theta(i) & \text{if the table has to be expanded: } \begin{pmatrix} i-1 \text{ to allocate the new table} \\ i-1 \text{ to migrate the old entries} \\ 1 \text{ to make the insertion} \\ 1 \text{ to reassign } T.arr \end{pmatrix} \end{cases}$$

The "Chicken Little" analysis would say that the worst possible string of $n$ operations would consist entirely of operations which have the worst-case single-operation running time $\Theta(i)$, so that the sequence has running time

$$\Theta\left(1+2+\cdots+n\right) = \Theta\left(n^2\right).$$

This bound is not tight, however, because it is not possible for every operation to be worst-case.

### 11.4.2 Aggregate analysis

Aggregate analysis of our table's operations amounts to keeping track of precisely which operations will be worst-case. The table has to be expanded if and only if the table is full, so

$$c_i = \begin{cases} \Theta(i) & \text{if } i-1 \text{ is zero or an exact power of 2} \\ 1 & \text{otherwise.} \end{cases}$$

Thus, a string of $n$ operations, $m$ of which are insertion operations, would take time

$$T\left(n-m \text{ lookups}\right) + T\left(m \text{ insertions}\right)$$

$$= \Big(\big(n-m\big)\cdot\Theta(1)\Big) + \left(\sum_{\substack{k-1 \text{ is not an} \\ \text{exact power of 2}}} \Theta(1) \quad + \sum_{\substack{k-1 \text{ is an} \\ \text{exact power of 2}}} \Theta(k)\right)$$

$$= \Big( (n-m) \cdot \Theta(1) \Big) + \left( \Big( m - \big(2 + \lfloor \lg(m-1) \rfloor\big) \Big) \cdot \Theta(1) \quad + \sum_{j=0}^{\lfloor \lg(m-1) \rfloor} \Theta\left(2^j\right) \right)$$

$$= \Theta(n-m) + \big( \Theta(m) - \Theta(\lg m) + \Theta(2m) \big)$$

$$= \Theta(n) + \Theta(m)$$

$$= \Theta(n) \quad (\text{since } m \le n).$$

Thus, a string of $n$ operations takes $\Theta(n)$ time, regardless of how many of the operations are insertions and how many are lookups. Thus, aggregate analysis gives an amortized running time of $O(1)$ per operation.

### 11.4.3 Accounting analysis

We can set up a balance sheet to pay for the operations on our table as follows:

- Lookups cost \$1
- Insertions cost \$3
  - \$1 to insert our element into the array itself
  - \$2 to save up for the next time the array has to be expanded
    - One dollar will pay for migrating the element itself
    - One dollar will go to charity.

Because the current incarnation of $T.arr$ started with $\frac{1}{2}|T.arr|$ elements pre-loaded, it follows that, when $T.arr$ is full, only $\frac{1}{2}|T.arr|$ elements will have enough money in the bank to pay for their migration. However, there will also be $\$\frac{1}{2}|T.arr|$ of charity money saved up from the $\frac{1}{2}|T.arr|$ most recent insertions. This charity is exactly sufficient to pay for migrating the elements that don't have their own money. Thus, our balance sheet checks out; our assigned costs of \$1 and \$3 work. In particular, each operation's cost is $O(1)$.

### 11.4.4 Potential analysis

In our potential analysis, the potential energy will play the role of money in the bank: when the array needs to be doubled, the potential energy will exactly pay for migrating the old elements into the new array. Thus, we want to find a function which equals zero immediately after doubling the table and grows to size $|T.arr|$ as it is filled up. (To simplify notation, assume that copying an array of size $k$ takes exactly $k$ steps, rather than always using the more precise notation $\Theta(k)$.) The simplest function which satisfies these properties, and the one that will be most convenient to work with, is

$$\Phi(T) = 2 \cdot T.num - |T.arr|.$$

Right after expansion, we have $T.num = \frac{1}{2}|T.arr|$, so $\Phi(T) = 0$; right before expasnion, we have $T.num = |T.arr|$, so $\Phi(T) = 2 \cdot |T.arr| - |T.arr| = |T.arr|$.

The actual cost of a lookup is, as we said, $c_{\text{lookup}} = 1$. (Again, to simplify notation, let's just write 1 instead of $\Theta(1)$.) Using the potential $\Phi$, the amortized cost of a lookup is

$$\widehat{c}_{\text{lookup}} = c_{\text{lookup}} + \Phi(T_{\text{new}}) - \Phi(T_{\text{old}}) = c_{\text{lookup}} = 1,$$

since $T_{\text{new}} = T_{\text{old}}$.

To find the amortized cost of an insertion, we must split into cases. In the first case, suppose the table does not need to be doubled as a result of the insertion. Then the amortized cost is

$$\begin{aligned}
\widehat{c} &= c + \Phi(T_{\text{new}}) - \Phi(T_{\text{old}}) \\
&= 1 + \left(2 \cdot T_{\text{new}}.num - |T_{\text{new}}.arr|\right) - \left(2 \cdot T_{\text{old}}.num - |T_{\text{old}}.arr|\right) \\
&= 1 + 2\left(T_{\text{new}}.num - T_{\text{new}}.num\right) - \left(|T_{\text{new}}.arr| - |T_{\text{old}}.arr|\right) \\
&= 1 + 2(1) + 0 \\
&= 3.
\end{aligned}$$

In the second case, suppose the table does need to be doubled as a result of the insertion. Then the actual cost of insertion is $|T_{\text{new}}.num|$, so the amortized cost of insertion is

$$\begin{aligned}
\widehat{c} &= c + \Phi(T_{\text{new}}) - \Phi(T_{\text{old}}) \\
&= T_{\text{new}}.num + \left(2 \cdot T_{\text{new}}.num - |T_{\text{new}}.arr|\right) - \left(2 \cdot T_{\text{old}}.num - |T_{\text{old}}.arr|\right) \\
&= \left(|T_{\text{old}}.arr| + 1\right) + 2\left(T_{\text{new}}.num - T_{\text{old}}.num\right) - \left(|T_{\text{new}}.arr| - |T_{\text{old}}.arr|\right) \\
&= \left(|T_{\text{old}}.arr| + 1\right) + 2(1) - \left(2|T_{\text{old}}.arr| - |T_{\text{old}}.arr|\right) \\
&= 3.
\end{aligned}$$

In both cases, the amortized running time of insertion is 3. (If we had picked a less convenient potential function $\Phi$, the amortized running time would probably be different in the two cases. The potential analysis would still be valid if done correctly, but the resulting amortized running times probably wouldn't be as easy to work with.)

### 11.4.5 Exercise: Allowing deletions

Suppose we wanted our dynamically resized table to support pops (deleting the most recently inserted element) as well as insertions. It would be reasonable to want the table to halve itself when fewer than half the slots in the array are occupied, so that the table only ever takes up at most twice the space it needs to. Thus, a first attempt at a pop functionality might look like this (with the TABLE-INSERT procedure unchanged):

**Algorithm:** NAÏVE-POP($T$)

```
1  if T.num = 0 then
2      error "Tried to pop from an empty table"
3  r ← T.arr[T.num − 1]
4  Unset T.arr[T.num − 1]
5  T.num ← T.num − 1
6  if T.num ≤ ½ |T.arr| then
7      arr′ ← new array of size ⌊½ |T.arr|⌋
8      Migrate all items from T.arr to arr′
9      T.arr ← arr′
10 return r
```

Unfortunately, the worst-case running time of NAÏVE-POP is not so good.

**Exercise 11.1.** *Given $n$ sufficiently large, produce a sequence of $n$* TABLE-INSERT*, lookup, and/or* NAÏVE-POP *operations which have total running time $\Theta(n^2)$. Thus, in any amortized analysis, at least one of the operations must have amortized running time at least $\Omega(n)$.*

**Exercise 11.2.** *Devise a way of supporting pops which still retains $O(1)$ worst-case amortized running time for each operation, and do an amortized analysis to prove that this is the case. You may redefine* TABLE-INSERT *or add more data fields to $T$ if you wish (and of course you will have to define* TABLE-POP*), but it must still remain the case that, for every possible sequence of operations, $T$ only takes up $O(T.num)$ space in memory at any given point in time.*

*[Hint: An adversary can choose a series of operations for which* TABLE-INSERT *and* NAÏVE-POP *run slowly by making sure $T.num$ repeatedly crosses the critical thresholds for table doubling and table halving. (You must have used this fact somehow in your solution to Exercise 11.1.) So your first step will be to figure out how to place decisions about doubling and halving outside the control of your adversary.]*

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2012