Lecture Notes for CSCI 311: Algorithms
Set 15-Amortized Analysis

Professor Talmage

April 7, 2025

---

1 Motivating Example

We have been working to reduce the worst-case cost of data structure operations, specifically balancing search trees to reduce both the variability in cost and the worst-case cost. Sometimes it is not possible to keep worst-case costs down with this approach, though, as it becomes necessary at some point to perform an inherently expensive operation. We are next going to look at how we can analyze the runtime of a data structure at a higher level of abstraction, noting that rare expensive operations are less of a problem than frequent ones.

Consider an implementation of a Queue with runtimes

- *Enqueue*: $O(1)$ always
- *Dequeue*: $O(1)$ usually, sometimes $O(n)$.

How can we best represent the runtime of this implementation?

1. Best-case cost: Misleading.
2. Worst-case cost: Accurate, but if the bad case is very rare, may not reflect practical behavior.
3. Common-case: Can get you in trouble, similar to best-case.
4. Average-case: May not accurately represent outliers, and may not ever actually happen.

Consider the following pseudocode:

Algorithm 1 Queue from Two Stacks

```
function ENQUEUE($x$)
    $S_1$.push($x$)
end function

function DEQUEUE
    $x = S_2$.pop()
    if $x \neq \bot$ then return $x$
    $y = S_1$.pop()
    while $y \neq \bot$ do
        $S_2$.push($y$)
        $y = S_1$.pop()
    end while
    return $S_2$.pop()
end function
```

Complexity: `Enqueue` is always $O(1)$. `Dequeue` varies:

1. Best-case: $O(1)$
2. Worst-case: $O(n)$

Exercise: What is n in this context?

3. Common-case: $O(1)$
4. Average-case: ?

So, while it is accurate to say that `Enqueue` is $O(1)$ and `Dequeue` is $O(n)$ in the worst case, that does not communicate the fact that `Dequeue` is *usually* $O(1)$. To more precisely (not more accurately!) represent the cost, consider the lifecycle of a particular data element in the queue. That element will do four things: push onto S_1, pop from S_1, push onto S_2, pop from S_2. `Dequeue` is only expensive when it needs to perform these steps for many data elements at once.

First, we can analyze the total cost of a sequence of n `Enqueue` and `Dequeue` instances by noting that there will be at most n data elements, each of which incurs a total cost of at most 4, so the sequence's total cost is at most $4n = O(n)$. An alternative is to suppose that we could "pay ahead". Charge each `Enqueue`(x) 4 units. Then a `Dequeue` needs pay nothing, since the costs involved in moving its return value, and any other values that are in the way, from S_1 to S_2 have already been paid. Now what are our costs?

- $T($`Enqueue`$) = 4 = O(1)$
- $T($`Dequeue`$) = 0 = O(1)$
- $T(\text{any sequence of } n \text{ instances}) = n * O(1) = O(n)$.

2 Amortized Analysis

These two alternate approaches to considering the cost of a sequence of operations instances are types of *amortized analysis*, which is a collection of methods for looking at the total cost of all interactions with a data structure, instead of the specific costs of each operation instance. It is important to understand that amortized analysis is still a form of worst-case analysis, just at a different scale than that to which we are accustomed. We are now considering the worst-case cost of an entire sequence of operation instances. Another way to think of amortized analysis is as finding the worst-case average cost of a repeated operation.

Aside: Amortized analysis applies to any sequence of repeated operations, not just data structures, but data structures are where we most often see this pattern, so we will use the language of data structures.

2.1 Multipop Example

Suppose we add a new operation to a stack:

- `Push`(x)
- `Pop`$()$
- `Multipop`(k)

`Push` and `Pop` behave as normal, and can be efficiently implemented in $O(1)$ time each using linked lists. `Multipop` returns the k latest-pushed elements which have not already been returned, and we will implement it as a for loop, internally calling `Pop` k times, exiting early if we get a $\perp$.

To discuss the complexity of a data structure D, let n be the total number of operation instances in a run. Let $T(n)$ be the total worst-case cost of any sequence of n operation instances on D. The worst-case complexity of `Multipop` is thus $O(n)$, since we could `Push` $n-1$ values, then call `Multipop`$(n-1)$, which would cost $n-1$.

Claim 1. *For a stack with* `Multipop`*,* $T(n) = \Theta(n)$.

Proof. First, note that $T(n)$ is the cost of all `Push` instances plust the cost of all `Pop` instances, plus the cost of all `Multipop` instances: $T(n) = T(\texttt{Push}) + T(\texttt{Pop}) + T(\texttt{Multipop})$.

The total number of `Pop` calls, whether called directly or as part of a `Multipop` must be at most n, since there are no more than n elements ever in the stack, since there are at most n `Push` instances. Thus, the cost of all `Pop` instances plus all `Multipop` instances is at most n, and the cost of all `Push` instances is at most n, so $T(n) \le n + n = O(n)$. □

2.2 Types of Amortized Analysis

There are typically three types of amortized analysis. these are (mostly) just different ways to express the same fundamental concepts, and you can usually use whichever fits your problem best.

1. Aggregate Method: Sum the total cost $T(n)$ of a worst-case sequence of n intances. Amortized cost of an instance is then $T(n)/n$.
2. Accounting Method: Overcharge some operation instances to pay costs in later instances. This method typically associates extra charge with particular data values.
3. Potential Method: Again, overcharge some instances, but store total overcharge in one "bank account", instead of keeping it tied to particular data values.

3 Aggregate Method

To use the aggregate method of amortized analysis, compute the most **any** sequence of n operation instances could cost. Either present your result as the total cost of a sequence or, if appropriate, divide by n to get an amortized cost per instance. This works best when considering a sequence of n instances of the same operation, as direct division could be misleading if the costs are not evenly distributed among different operations in the sequence.

Example: k-bit binary counter.

Suppose we have a binary counter with k bits of memory and one operation: `increment()`. The counter starts at 0, and counts in binary:

$$0000 \to 0001 \to 0010 \to 0011 \to 0100 \to 0101 \to \cdots$$

We will measure cost as the number of bit flips an operation instance must perform.

Exercise: What is the worst-case cost of a single `increment` instance? What is the worst-case cost of a sequence of n instances?

- Could have to flip all k bits, for example from $011111 \to 100000$.
- Can conclude that n instances cost a total of $O(kn)$.

- Most instances flip nowhere near k bits, so we might be able to do a better analysis.

Let us label the bits $b_0, b_1, \dots, b_{k-1}$ and consider their individual behaviors.

000000	000110
000001	000111
000010	001000
000011	001001
000100	001010
000101	...

We see that b_0 (the rightmost bit) flips in every increment. b_1 flips in every second increment, which makes sense since it represents 2's. b_2 flips every fourth increment, b_3 every eighth, and so on. In general, b_i flips every 2^ith increment. We can add these up and say that in a sequence of n `increment` instances, we have

$$\begin{aligned} T(n) &= \sum_{i=0}^{k-1} (\#\text{ times } b_i \text{ flips}) \\ &= n + n/2 + n/4 + \cdots + n/(2^{\log_2 n}) \\ &\leq n + n/2 + n/4 + \cdots \\ &= \frac{n}{1 - 1/2} = 2n \end{aligned}$$

Thus, $T(n) = O(n)$, so the amortized cost of n `increment` instances is $O(n)$, and since the instances are all the same, we could say that the amortized cost of a single `increment` instance is $O(n)/n = O(1)$.

4 Accounting Method

The idea for the next method of amortized analysis is that we will overcharge some (cheap) operation instances to pay for later expensive operation instances. The particular approach of the accounting method is that the extra charge, beyond paying for the cheaper instance itself, is stored on a particular data element in the structure, to pay for later operations which interact with that data element. We call the amount we charge each operation instance, whether over or under its actual cost, its *amortized cost*.

For amortized cost to make sense, it is essential that we do actually "pay for" all real costs, in all possible sequences. Recall that any prefix of a legal sequence of operations is itself legal, so in any prefix of a sequence, the amortized cost must be at least as large as real cost. In non-computing terms, we can never be in debt. Mathematically, if the amortized cost of the ith operation instance in a sequence is $\hat{c}_i$ and its real cost is c_i, we must have

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

Or for a prefix, $\sum_{i=1}^{k} \hat{c}_i \geq \sum_{i=1}^{k} c_i, \forall 1 \leq k \leq n$. The reason this holds is that being in debt, or having lower total amortized cost than real cost, would be borrowing time, which we cannot do. If we had a sequence that went into debt, then paid itself off later, what would happen if we terminated the sequence in the middle, while it was in debt? Our amortized analysis would not account for the true cost of that prefix, and would thus be a lie.

Example: We already used accounting analysis on the queue from two stacks. Next, let us go back to the k-bit counter and see how to apply this technique to get the same amortized cost.

Exercise: Write pseudocode for `increment`, treating the bits as an array and looping from b_0 up to b_{k-1}. How do we know when to stop flipping bits?

Observe that any single `increment` instance will only flip a single bit $0 \to 1$, while potentially flipping many bits $1 \to 0$. We use this as the foundation of our amortization. Charge \$2 for flipping a bit $0 \to 1$, leaving the extra \$1 on the 1. Charge \$0 for flipping a bit $1 \to 0$, instead using the \$1 left on the 1. Now, since each `increment` instance only does one $0 \to 1$ flip, we charge each instance only \$2. The charges left on the 1 bits pay for the rest of the work each `increment` has to do, so the total actual cost is no more than the total amortized cost, and we have a valid amortization, with $\hat{(}\texttt{increment}) = 2$ and $T(n) = 2n$.

Exercise: Perform an accounting-method analysis on the stack with `Multipop`.

Exercise: Recall from CSCI 204 (or other Data Structures course) that an array implementation of a list has `append` cost $O(1)$ most of the time, when the array is not full, then $O(n)$ to extend the array (allocate new space and copy data) when it is full. Use an accounting analysis to argue that the amortized cost of `append` is $O(1)$.

5 Potential Method

The third and final method of amortized analysis is named for gravitational potential energy, as it has some similar properties. Recall that gravitational potential energy is a function of the height of an object in a gravity well. In day-to-day usage, this means that the higher an object is, the more energy it has stored. If you drop something, that potential energy becomes kinetic energy as the object falls. One of the important features of gravitational potential energy is that the path an object takes to a particular position is irrelevant–the same position will have the same energy, whether one got there from above, below, sideways, or anything else.

Like the accounting method, the potential method overcharges cheap operation instances, saving the extra charge to pay for later costs. However, we will store the extra charge in one lump sum, instead of distributing it throughout the data structure. Further, the stored charge, or potential, is always the same when the structure is in the same state, however it got there. Maintaining this property will influence how we choose our amortized costs, but it is useful because, when set up correctly, it implies that there is always enough potential stored up to pay for any expensive operation instances.

Notation: If D is a state of a data structure, we use $\phi(D)$ to denote the potential of that state. Recall that an execution on a data structure is a sequence of steps (operation instances) s_i from the initial state D_0. We can then denote the sequence of states as $D_0, D_1, \ldots$, with $s_i(D_{i-1})) = D_i$. Potential is a function $\phi : \{D_i\}_{i \geq 0} \to \mathbb{R}$.

Definition 1. The amortized cost of a step s_i with real cost c_i is

$$\hat{c_i} = c_i + \phi(D_i) - \phi(D_{i-1}).$$

That is, amortized cost is the real cost, adjusted by the change in potential $\Delta(\phi) = \phi(D_i) - \phi(D_{i-1})$ (new potential minus old potential).

If the change in potential is positive, we have overcharged this step and stored more potential for later. If the change is negative, we are using potential to help pay for this step. In fact, $\hat{c_i}$ may be 0 or negative if potential changes enough.

Claim 2. *If a sequence of steps S takes a data structure from state D to state D', then* $\sum_{s_i \in S} \hat{c_i} = \sum_{s_i \in S}(c_i) + \phi(D') - \phi(D)$.

We omit the proof, since it is just canceling out alternating positive and negative appearances of the potential of each intermediate state.

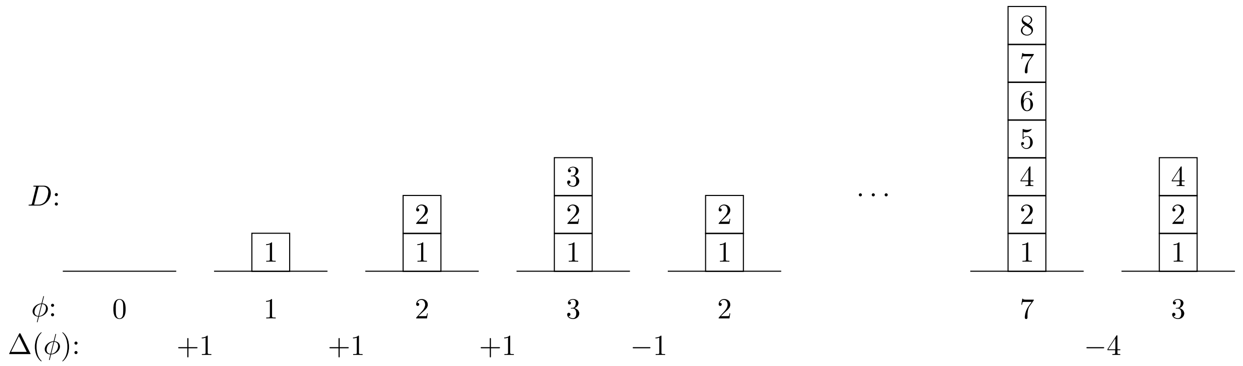
Rule: For amortized cost to be valid or meaningful (actually describing the real cost), we must have $\phi(D) \geq \phi(D_0)$ for all reachable states D. That is to say, at no point can we use more potential to pay for a step than we have already saved up. In general, $\phi(D_0)$ can be any value, but since whatever our starting potential is provides a floor beneath which we cannot go, we typically normalize $\phi(D_0) = 0$ and show that our potential is never negative to prove its validity.

5.1 Examples

We will revisit a couple of previous examples with this new method, then move to a more complex example which would not be as feasible to analyze with either the aggregate or accounting methods.

Example: Stack with Multipop. In an arbitrary state of the structure, we must have enough potential saved up to pay for a potential Multipop, so define

$$\phi(D) = \text{ the size of the stack in state } D$$



First, we check validity of our analysis: $\phi(D_0)$ is the initial stack size, which is 0. Since the size of a stack is never negative, $\phi(D) \geq 0 = \phi(D_0)$ for any reachable state D. Next, we compute the amortized cost of each operation. Let D denote the state immediately before the operation, D' the state immediately after, and s the size of the stack in state D:

- `Push`: $\hat{c}(push) = c(push) + \phi(D') - \phi(D) = 1 + (s+1) - (s) = 2$
- `Pop`: $\hat{c}(pop) = c(pop) + \phi(D') - \phi(D) = 1 + (s-1) - (s) = 0$
- `Multipop`: $\hat{c}(multipop) = c(multipop) + \phi(D') - \phi(D) = k + (s-k) - (s) = 0$

That is, the amortized cost of each operation is $O(1)$ and the amortized cost of any sequence of n operation instances is $O(n)$.

Example: k-Bit Counter.

Exercise: Perform a potential analysis of the k-bit counter, generalizing our accounting analysis.

- $\phi(D)$ will be the number of 1's in the counter.
- $\hat{c}(increment) = c(increment) + \Delta(\phi) = (\text{\#flips}) + (\text{one } 0 \to 1 \text{ flip}) - (\text{\# } 1 \to 0 \text{ flips}) = 2$

Example: Allocating space for a dynamic table. Consider a generalization of our algorithm for implementing a list type using arrays. Now, "table" means any block of memory holding many items, whether a list, 2-dimensional table, etc. "Dynamic" means that the size of the structure will change over time. We want to ignore the specific properties of the table (such as re-connecting linked lists, shifting elements in arrays, etc.) and just focus on memory allocation costs. Thus, we will primarily consider the size of the table–the number of elements currently stored, versus the capacity–the amount of space currently allocated.

We consider two operations. First, `Insert`(x) inserts a new value x, ensuring that there is enough space allocated to store the now-increased size. Assume the table has capacity 0 on creation and that the cost of allocation is linear in the current size of the table, since we will need to copy everything to a newly-allocated space in memory. Second, `Delete`(x) deletes some value x, and will ensure that there is not "too much" space allocated.

We will use the following heuristics ("rules of thumb" or solution ideas):

- If the table is full, `Insert` will double its capacity (increase to 1 if it had capacity 0).
- If the table is less than 1/4 full, `Delete` will halve its capacity.

Exercise: We have discussed in the case of simple lists why doubling when full leads to good amortized performance of `Insert`. Why would we not halve when half empty?

Before we dive fully into potential analysis, consider the real and aggregate cost of `Insert`. $c(insert) = 1$ when the table is is not full (size less than capacity), or the size of the table if it is full. We can then see that in a sequence of n inserts, the total cost is

$$\sum_{i=1}^{n} c_i \leq n + \sum_{i=0}^{\lfloor \log_2 n \rfloor} 2^j = 3n$$

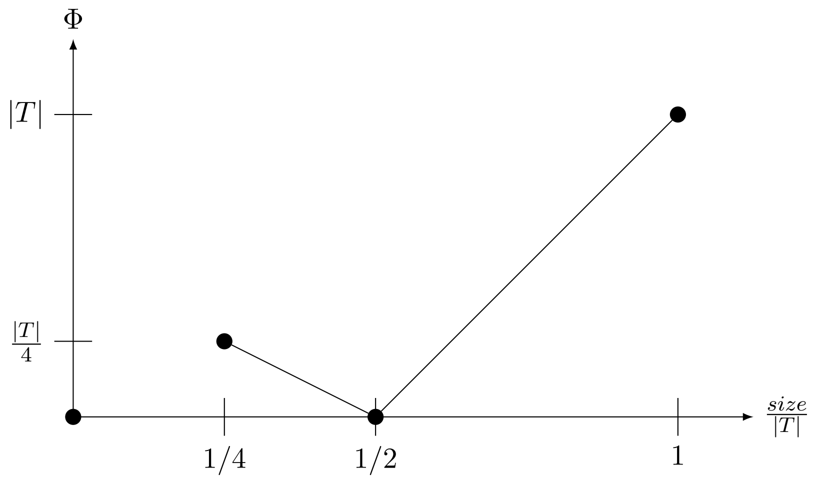
Adding `Delete` instances to the sequence can only decrease the cost of `Insert` instances (since there is less data to copy), so in any sequence of n instances, the total cost of `Insert` instances, divided by the number of `Insert` instances is $O(1)$. (This is how you can sometimes use aggregate analysis to get per-operation amortized times.)

We want to fully analyze `Delete` at the same time as `Insert`, and we would like to have a per-instance amortized cost for each, not just aggregate costs. To do this, we will find a potential function that saves enough to pay for any expensive instances, whether `Insert` or `Delete`. Consider when expensive instances happen: `Insert` is expensive when the table is full. `Delete` is expensive when the table is 1/4 full. Note that expensive `Insert` instances are rarer, but cost more. The final observation we need is that immediately after doubling or halving capacity, the table is half full, so we can be confident that there will be quite a few operation instances before the next expensive instance, so we do not need to have any potential left in this state.

We can now use these observations to describe the potential function. When the table is 1/4 or entirely full, we need to have as much potential saved up as the size of the table. When it is 1/2 full, we set the potential to 0. Denote the capacity of the table as $|T|$. Then, graphically, our potential function looks like this:

Observe that the first portion of the graph has slope -1, since it takes $|T|/4$ `Insert` instances to fill the table from 1/4 to 1/2 full (or an equal number of `Delete` instances to go the other way), and the second part has slope 2, since we have to store $|T|$ potential in $|T|/2$ instances. We do not need to define ϕ for tables less than 1/4 full, as we shrink the allocation, resetting the table to half full, when the table would be less than 1/4 full. The only exception is the special case of an empty table, with $\phi(D_0) = 0$.

We can now determine the amortized cost of each operation. Note that there are three cases for each operation: when the table is between 1/4 and 1/2 full, when it is more than 1/2 full, and when we re-allocate and copy. We have previously avoided having different amortized costs in different cases, instead



rounding up and overestimating the cost. We can instead have different amortized costs in different cases and look at the largest of those as the bound we provide.

$$\hat{c}(\texttt{Insert}) = \begin{cases} 1 + \phi(size+1) - \phi(size) = 1 - 1 = 0 & \text{if } 1/4 \leq \frac{size}{|T|} < 1/2 \text{ (less than half full)} \\ 1 + \phi(size+1) - \phi(size) = 1 + 2 = 3 & \text{if } 1/2 \leq \frac{size}{|T|} < 1 \text{ (at least half full)} \\ (size + 1) + \phi(1/2 \text{ full}) - \phi(\text{full}) = & \text{if } size = |T| \text{ (full)} \\ size + 1 + 0 - size = 1 & \end{cases}$$

$$\hat{c}(\texttt{Delete}) = \begin{cases} 1 + \phi(size-1) - \phi(size) = 1 + 1 = 2 & \text{if } 1/4 < \frac{size}{|T|} \leq 1/2 \text{ (at most half full)} \\ 1 + \phi(size-1) - \phi(size) = 1 - 2 = -1 & \text{if } 1/2 < \frac{size}{|T|} \leq 1 \text{ (more than half full)} \\ (size + 1) + \phi(1/2 \text{ full}) - \phi(1/4 \text{ full}) = & \text{if } \frac{size}{|T|} = 1/4 \text{ (min size)} \\ size + 1 + 0 - size = 1 & \end{cases}$$

In all cases, the cost of each operation is at most $2 = O(1)$, so we have constant amortized cost for both `Insert` and `Delete`.