Lecture 3 (09-05) - Amortized Analysis

Sometimes, when we have an algorithm that is performed as a series of cheap operations followed by an occasional expensive operation, it can be extremely impractical to assume that every operation has the cost of the expensive operation.

Instead, **amortized analysis** "averages" out the cheap and expensive operations for the sequence overall.

- Note that this isn't average case analysis - we still consider the worst-case but have a less pessimistic method of taking the upper bound.
- Consider if the overall algorithm takes $T(m)$ total time, where m is the number of operations. The amortized cost per operation then is $\frac{T(m)}{m}$.

Basic Example: Binary Counter

Consider a binary counter represented by an unbounded array A ordered by the least significant bit first.
This counter has one operation: `increment` that increases the binary number counted by 1.

- Changing one bit of the array costs a unit amount of time.

Operation #	Counter State	Cost
0	`0`	0
1	`1`	1
2	`01`	2
3	`11`	1
4	`001`	3
5	`101`	1
6	`011`	2
7	`111`	1
8	`0001`	4

and so on...

What is the amortized cost of the `increment` operation?

Let us consider each bit. Notice the least significant bit gets flipped every operation. The 2^{nd} least significant bit gets flipped every 2 operations, and the 3^{rd} least significant bit gets flipped every 4 operations.

More generally, the k^{th} least significant bit gets flipped every $lg(k)+1$ operations
In this case, we can write our total cost, $T(m)$ as the following:

$$T(m) = m \sum_{i=1}^{lg(m)+1} (\frac{1}{2})^{i-1}$$

If we simply use ∞ instead of $lg(m)+1$ and treat it as a geometric series, we get the geometric series that converges to 2. Therefore, the amortized cost per operation is $\frac{2m}{m} = 2$.

Extension: Binary Counter with Scaling Cost

Suppose if instead of changing a bit taking unit time, changing the k^{th} least significant bit takes 2^k time.

Well, as we are only changing the cost, we may use the same summation but edit the cost:

$$T(m) = m \sum_{i=1}^{lg(m)+1} 2^{i-1}(\frac{1}{2})^{i-1} = m \sum_{i=1}^{lg(m)+1} 1 = m(lg(m)+1))$$

Given this, we then get $lg(m)+1$ for our amortized cost per operation.

Amortized dictionary

Motivating this, consider if we have a data structure for storing on SSDs, with the limitation that we want to write as much of the data to the drive in large chunks and as infrequently as possible due to the limited write cycles of the drive.

This data structure should still support relatively fast lookup and insert times.

First, let's consider a sorted array.

- We can lookup in $O(\log(n))$ time using binary search.
- However, to insert, we need $O(n)$ time due to moving elements.
  Second, let's consider an unsorted linked list.
- We can insert in $O(1)$ time by simply appending it to the head.
- However, to lookup, we take $O(n)$ time to traverse the list.

If we combine the two, however, we can achieve a relatively fast data structure that does writes in large chunks.

Consider a linked list of sorted arrays. The i^{th} node of the linked list has either an array of size 2^{i-1} or an empty array. When we insert an element, we start from the 1^{st} node and check if an array is empty. If it is, we simply insert. If not, we merge it into the next array until we get an empty array.

In this way, insertion is similar to the extended binary counter example with $O(lg(n))$ amortized cost.

Additionally, a search involves simply binary searching each non-empty array. In the worst case, we get $O\left(lg\left(\frac{n}{2}\right) + lg\left(\frac{n}{4}\right) + lg\left(\frac{n}{8}\right)\right) = O(\log^2(n))$

Charging Argument

One intuitive method of thinking about amortized analysis is that with each operation, we are given an allowance of c units per operation we do.

Most of the time, we won't have to pay our entire allowance to perform our operation - these are our "cheap" operations. In these cases, we will keep the excess allowance in a "bank" account.

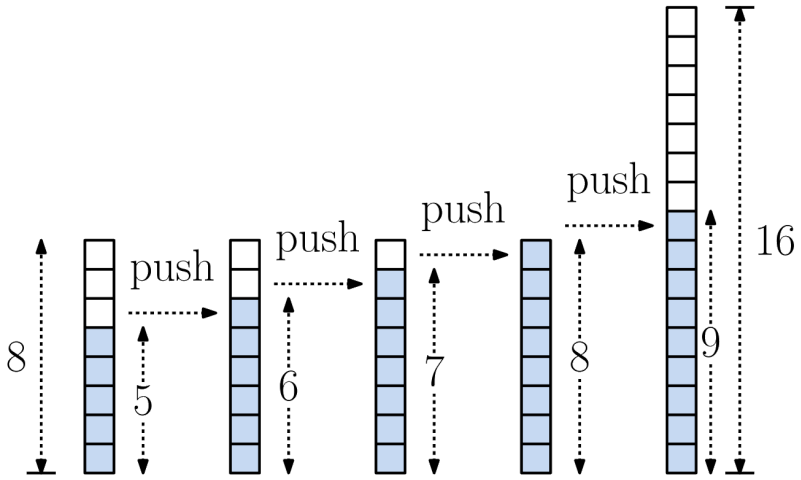
We would like to show, then, that when it comes to get charged for an expensive operation, we will have sufficient funds in our "bank" to charge for the operation.

The key to the charging argument is to break the sequence of operations down into "runs," usually beginning at the end of an expensive operation and ending with the next expensive operation.
We usually show some sort of congruence between runs.

Expanding Stacks

Consider an implementation using a simple array with some fixed size. We will push to this array until it is at capacity. When it is at capacity, we will copy all elements over to a new array of larger fixed size and add the element on top. We will consider allocating the new array a free operation.

In this model, let us consider both adding an element to an array and copying an element from one array to another to consume a unit cost.



Our claim is then, that this push operation takes amortized $O(1)$ time. More specifically, we argue that it takes at most 3 cost per operation.

Let's show this: consider a run starting at the $2^k + 1^{th}$ insert. Notice that we have $m = 2^k$ inserts before we must expand our stack (our expensive operation) where we must move a total of $2^{k+1} + 1$ for the move and insert respectively and 1 for each cheap operation.

Summing this, our claim is that $(m-1) + 2m + 1 \leq 3m \implies 3m \leq 3m$

Therefore, our claim holds.

Revisiting the binary counter

Let us revisit the binary counter in the context of our charging argument.
We can have each bit have its own allowance. The argument is that each bit's allowance is 1.

Consider: the first bit obviously flips every increment and costs 1 per flip.
The second bit flips every other increment and costs 2 per flip.
The third bit flips every four increments and costs 4 per flip, etc.

Therefore, after m operations, we have $lg(m) + 1$ bits that need flipping at an amortized cost of 1 per bit = $lg(m) + 1$ amortized cost for the m^{th} increment operation.

#amortized-cost #stack #algo #analysis #ds #binary-counter #dictionary