15-122: Principles of Imperative Computation

Recitation 12b Solutions — Josh Zimmerman

Practice!

Unbounded array insertion — aggregate analysis

Using aggregate analysis, show that adding an element to an unbounded array takes amortized $O(1)$ time. We'll only count the number of array writes, for simplicity.

Solution: NOTE: To more formally prove this, we'd need to use mathematical induction, but to make it easier to understand, this is just the core of the argument.

Consider n insertions to an array with limit $n-1$ (the array starts out empty).

Exactly one insertion will take n steps: $n-1$ to copy over all $n-1$ elements from the small array to the large array and one to insert the new element.

Every other insertion takes 1 step — we just insert the element.

So, over n insertions, we have a total of $(n-1)*1+1*(n) = 2n-1$ steps.

$$\frac{2n-1}{n} = 2 - \frac{1}{n} \in O(1)$$

Thus, we have an amortized runtime of $O(1)$ for insertion.

Unbounded array insertion — accounting analysis

Using an accounting analysis, show that adding an element to an unbounded array takes amortized $O(1)$ time (the array starts out empty).

Again, we'll only be counting array writes.

Assume that the limit is n, and that $n > 0$ (the analysis can also work when $n = 0$, but there's an annoying special case).

Solution: NOTE: To more formally prove this, we'd need to use mathematical induction, but to make it easier to understand, this is just the core of the argument.

When we insert, we need to pay 1 token (this reflects the cost of inserting into an array). We can also put two tokens aside. So, we lose 3 tokens for every insertion. Note that this is still a constant cost.

Then, when it comes time to make a new array we can reach into our stash of coins, which now has $2n$ tokens, since we've inserted into the array n times and put 2 tokens into the stash for each insertion. To help pay for the cost of copying elements over, we grab n tokens from the stash and use them. Now our stash has n tokens left in it. Then, we pay 1 token to insert the new element in the array, and set two tokens aside in the stash (leaving us with $n+2$ tokens).

On future resizings of the array, we'll have enough tokens since we insert two tokens every time — one of them will pay to copy the element that was inserted when we got it and the other will pay for copying one in the first half of the array.

In this way, we pay 3 tokens for every insert, which is a constant. So, insertion into the unbounded array is amortized constant time.

Binary counter

Consider the situation where we have an n-bit binary number. Assume that flipping a bit (changing it from 1 to 0, or from 0 to 1) is a constant-time operation.

What is the amortized time complexity of incrementing the number, in terms of n?

Solution: Consider incrementing a number repeatedly.

To go from $\underbrace{00\ldots00}_{n \text{ bits}}$ to $\underbrace{00\ldots00}_{n \text{ bits}}$ (we're ignoring the case of performing zero increments, since it doesn't make sense to consider the cost of no operations), we need 2^n increments. For example, if $n = 2$, we go: 00, 01, 10, 11, 00. This is a total of $4 = 2^2$ increments. (I'm assuming the normal rules for integer overflow here.)

We'll flip bit i $\frac{2^n}{2^i}$ times.

The proof of this is by induction on the bit (i).

Base case. The 0th bit is flipped on every increment. There are 2^n increments, and $\dfrac{2^n}{2^0} = 2^n$

Inductive hypothesis. Assume that for some fixed $k \in \mathbb{N}$ (where $k < n - 1$), we flip the kth bit $\frac{2^n}{2^k}$ times.

Inductive step. We wish to show that we flip bit $k + 1$ $\frac{2^n}{2^{k+1}}$ times.

To do this, we consider how many times we flip bit $k + 1$ in relation to bit k. If bit k is 0 when it is flipped for an increment, we do not flip bit $k + 1$, and if bit k is 1 when it is flipped for an increment, we do flip bit $k + 1$. Thus, we flip bit $k + 1$ exactly half as much as we flip bit k, and so, by the inductive hypothesis, we flip bit $k + 1$

$$\frac{2^n}{2^k} * \frac{1}{2} = \frac{2^n}{2^{k+1}}$$

times.

So, we flip bit k $\frac{2^n}{2^k}$ times.

Thus, we have a total of

$$\begin{aligned}
\sum_{i=0}^{n-1} \left(\frac{2^n}{2^i}\right) &= 2^n \sum_{i=0}^{n-1} \frac{1}{2^i} \\
&= 2^n \left(\frac{1 - (\frac{1}{2})^n}{\frac{1}{2}}\right) \\
&= 2^n(1 - \frac{1}{2^n})(2) \\
&= 2^n(2 - \frac{2}{2^n}) \\
&= 2^{n+1} - 2
\end{aligned}$$

flips. However, this was over 2^n increments, so we divide and see:

$$\begin{aligned}\frac{2^{n+1}-2}{2^n} &= \frac{2^{n+1}}{2^n} - \frac{2}{2^n} \\ &= 2 - \frac{1}{2^{n-1}} \in O(1)\end{aligned}$$

(The claim about big-O holds because $2 - \frac{1}{2^{n-1}} < 3$ for all n.)

We can also use an accounting analysis to show that n increments will cost $O(n)$ tokens. (And so each increment costs $O(1)$ token.)

Start by putting a token next to each bit. There are n bits, so this costs n tokens to start. Then, for each increment operation, we pay 2 tokens. For every bit that must flip from 0 to 1, we simply use the token sitting next to it to pay for the flip. Then, for the one bit that must flip from 1 to 0, we pay for that using one of tokens we've allocated for this increment operation and put the other token down next to it for future flips from 0 to 1.

In this way, we only need to contribute 2 new tokens for each increment operation, and our n increments only cost $O(n)$ tokens. Thus, each increment has an amortized cost of $O(1)$ tokens.