15-122: Principles of Imperative Computation

Recitation 8 — Josh Zimmerman

Amortized analysis

Amortized analysis lets us consider the runtime behavior of a sequence of operations of an algorithm.

It lets us take a more nuanced view of the runtime of an algorithm: if there's some incredibly rare operation that takes a long time to do, it doesn't make sense to characterize the entire performance of the algorithm by that one operation. By using amortized analysis, we can get a more accurate view of how the algorithm will actually run.

Unbounded arrays

Unbounded arrays are implemented as pointers to `struct uba_headers`:

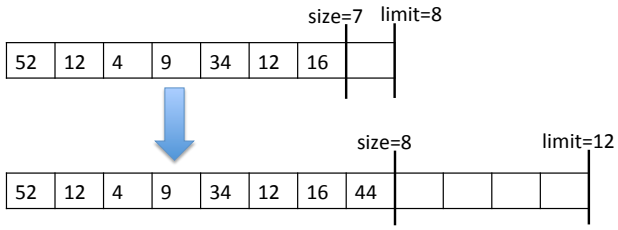
```
1 struct uba_header {
2   int size;
3   int limit;
4   elem[] data;
5 };
```

We're going to discuss a variation on the UBAs presented in class to give a different example of amortized analysis and discuss

When implementing unbounded arrays on an embedded device, a programmer is concerned that doubling the size of the array when we reach its limit may use precious memory resources too aggressively. So she decides to see if she can increase it by a factor of $\frac{3}{2} = 1.5$ instead, rounding down if the result is not an integral number.

This means that it won't make sense to have fewer than ___2___ elements in the array, because otherwise you might resize the array and get an array that wasn't any bigger. This would need to be reflected in the data structure invariant!

We're also going to resize the arrays a little bit earlier (which means we may use a bit more memory sometimes!) to make sure that we never end up with a `size` equal to `limit`.



Make sure you know how to write a data structure invariant for this modified UBA:

```
1 bool is_uba(struct uba_header* U) {
2   if (U == NULL) return false;
3   if (!(1 < U->limit)) return false;
4   if (!(0 <= U->size && U->size < U->limit)) return false;
5   //@assert \length(A) == U->limit;
6   return true;
7 }
```

We now carry out the the amortized analysis for this version of unbounded arrays. We'll start right after we've resized the array, when the size of the array is s and the limit of the array is $l = 3s/2$. We'll show that, assuming we start out with some non-zero number of tokens k, the next resizing of the array, from size l to size $l' = 3l/2 = 9s/4$, can be paid for by the cost (in tokens) of the operations that happen before that resize. (Just assume s is divisible by 4 for the purposes of this questions.)

Most of the operations require us to spend 1 token because we write to the `U->data` array exactly once.

In addition, we need to reserve 3 tokens for a total amortized cost of 4 tokens.

Starting from a $2/3$ full array, if we write into the old array every time, then after $l/3 = s/2$ insertions we will fill up the old array completely.

At this point, we have a total of $k + 3s/2$ tokens, and we need to copy $l = 3s/2$ tokens into the newly allocated array of size $l' = 3l/2$.

After all those copies, we have k tokens left. This is no smaller than k, and we have considered the worst case, so we will never run out of tokens.

You can try running through this analysis with other resizing factors:

If we triple the size of the array, we still need to reserve 2 tokens for a total amortized cost of 3 tokens. Starting from a $1/3$ full array, if we write into the old array every time, then after $2l/3 = 2s$ insertions we fill up the array. At this point we have $k+4s$ tokens, and we need to copy $l = 3s$ tokens to the newly allocted array of size $3l$. After all these copies we have $k + s$ tokens left. (If we tried to only reserve 1 token, we would end up iwith $k - s$ tokens. If we are allowed to have an amortized cost in fractions of a token, then it would suffice to reserve $1\frac{1}{2}$ tokens for a total amortized cost of $2\frac{1}{2}$ tokens.)

If we resize the array by a factor of $5/4$, we need to reserve 5 tokens for a total amortized cost of 6 tokens. Starting from a $4/5$ full array, if we write into the old array every time, then after $l/5 = s/4$ insertions we fill up the array. At this point we have $k + 5s/4$ tokens, and we need to copy $l = 5s/4$ tokens to the newly allocted array of size $5l/4$. After all these copies we have k tokens left.

If we resize the array by a factor of $4/3$, we need to reserve 4 tokens for a total amortized cost of 5 tokens. Starting from a $3/4$ full array, if we write into the old array every time, then after $l/4 = s/3$ insertions we fill up the array. At this point we have $k + 4s/3$ tokens, and we need to copy $l = 4s/3$ tokens to the newly allocted array of size $4l/3$. After all these copies we have k tokens left.

If we resize the array by a factor of $13/10 = 1.3$, we need to reserve 5 tokens for a total amortized cost of 6 tokens. Starting from a $10/13$ full array, if we write into the old array every time, then after $3l/13 = 3s/10$ insertions we fill up the array. At this point we have $k + 15s/10 = k + 3s/2$ tokens, and we need to copy $l = 13s/10$ tokens to the newly allocted array of size $13l/10$. After all these copies we have $k + 2s/10 = k + s/5$ tokens left. (If we tried to only reserve 4 tokens instead of 5, we would end up with $k - s/10$ tokens at the end. If we are allowed to reserve fractions of a token, then it would suffice to reserve $4\frac{1}{3}$ tokens for a total amortized cost of $5\frac{1}{3}$ tokens.)

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.

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).

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?

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 $\frac{2^n}{2^0} = 2^n$

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.