15-122: Principles of Imperative Computation

Recitation 12b   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.

There are a few ways to determine the amortized cost of an algorithm. We'll go over two of them here.

The first, called *aggregate analysis*, is a method in which we first determine that n operations take at *WORST* $T(n)$ steps. Then, we can conclude that the *amortized* cost of the algorithm is

$$\frac{T(n)}{n}$$

There's another method, called the *accounting method*, where we pay some number of tokens for each operation, and can optionally put some tokens aside at each step. Then, we can use the tokens we set aside balance out the cost of a later expensive operation. We look at the total number of tokens we pay over n operations to see what the amortized cost of each operation is.

There's another way of thinking about the accounting method where we start with some number of coins (say, n) and then use that to pay for our operations. We must be able to show that the number of coins we have never goes negative when doing such an analysis. If we show that, then we've shown that the operations we do will have total cost at most n.

You can use either of these methods depending on what you find easiest in a particular situation. Different problems call for different methods of doing amortized analysis.

If these descriptions of methods don't make complete sense to you, don't worry about it yet—we're going to go through a couple of examples during this recitation.

Unbounded arrays

In an unbounded array, we insert elements into an array `A` until it's full and then allocate an array `B` that is twice the size of `A`, copy each element of `A` to `B`, and insert the new element into the new array.

The worst case time complexity for a single insertion is $O(n)$, but the *amortized* time complexity of insertion into an unbounded array is $O(1)$.

We'll show this two ways: One using an *aggregate analysis* and one using the *accounting method*. (As usual, this problem is at the end of the handout.)

Next, we'll talk about why we wrote the removal code the way we did. Here's the code from lecture:

```
elem uba_rem(uba L)
//@requires is_uba(L);
//@requires L->size > 0;
//@ensures is_uba(L);
{
  if (L->size <= L->limit/4 && L->limit >= 2) {
    uba_resize(L, L->limit/2);
  }
  L->size--;
  elem e = L->A[L->size];
  return e;
}
```

But why do we resize when `L->size <= L->limit/4` instead of when `L->size <= L->limit/2`? In our implementation, we can get amortized constant time for removal and insertion, while in an implementation where we resize when `L->size <= L->limit/2`, we cannot get that amortized time—in fact, we get worst case amortized $O(n)$ time for insertion and removal, where n is the size of the unbounded array.

To see this, consider an unbounded array that has size $\frac{n}{2}$ and limit n. Then, consider a sequence of k removals and insertions. If we remove, we must resize the array, which takes $\frac{n}{2} - 1$ steps to copy the remaining elements into the new array. Then if we insert into the array, we must resize it again, which takes $\frac{n}{2}$ steps, for the same reason that removal takes that many steps. This pattern repeats, so overall we'll take $O(k\frac{n}{2})$ steps for all k operations. Thus, insertion and removal are amortized $O(n)$ — much slower than we want.

However, if we wait to resize the array when we have `limit/4` items in it, we will be able to add many items before we have to resize again. This allows us to spread the cost of the insertions and removals out over many operations and maintain amortized $O(1)$ insertion and removal time.

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.

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).

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?