CS61B Summer 2006
Instructor: Erin Korber
Lecture 24, 7 Aug.

1 Amortized Analysis

For some of the data structures we've discussed (namely hash tables and splay trees), it was claimed that the average time for certain operations is always better than the worst-case time.

The analysis technique that proves these claims is called *amortized analysis*. Amortized analysis is a way of proving that even if an operation is occasionally expensive, its cost is made up for by other, cheaper occurrences of the same operation.

1.1 The Averaging Method

Let's look at the average-case time for inserting an item into a hash table. Suppose a newly constructed hash table has just one bucket. Assume that the chains don't grow longer than O(1), and any operation that doesn't resize the table takes O(1) time–more precisely, suppose it takes at most one second.

The hash table has N buckets and n items. Suppose the insert operation doubles the size of the table (before adding the new item) if $n = N$. That way, the new item won't cause the load factor to exceed one. Suppose this resizing operation takes at most $2n$ seconds. When resizing is done, N doubles, the new item is inserted, and n increases by one.

After we construct an empty hash table and perform i insert operations, $n = i$ and N is the smallest power of two that is $\geq$ n. The total seconds for *all* the resizing operations is

$$2 + 4 + 8 + ... + N/4 + N/2 + N = 2N - 2.$$

So the cost of i insert operations is at most $i + 2N - 2$ seconds. Because $N < 2n = 2i$, the i insert operations take $O(5i - 2) = O(i)$ time, and so the *average* running time of an insertion operation is in O(1).

We say that the *amortized running time* of insertion is in O(1), even though the worst-case running time for a single insertion is in $Theta(n)$.

For almost any application, the amortized running time is more important than the worst-case running time, because the amortized running time determines the total running time of the application. The main exceptions

are some applications that require fast interaction (like video games), for which one slow operation might cause a noticeable glitch in response time.

1.2 The Accounting Method

Suppose our hash tables can resize in both directions: not only do they expand as the number of items increases, but they also shrink as the number decreases. Unfortunately, it's hard to analyze this with the averaging method, because you don't know what sequence of insert and remove operations an application might perform, so the number of operations won't tell us anything about the size of the table, which was key to the averaging analysis.

Let's examine a more sophisticated method. In the *accounting method*, we "charge" each operation a certain amount of time. Usually we overcharge, but we can save time in the bank for use during later operations.

We don't actually know how many seconds any computation takes, because it varies from computer to computer. However, everything a computer does can be broken down into constant-time computations, so we will let a "dollar" be a unit of time that's long enough to execute the slowest constant-time computation that comes up in the algorithm we're analyzing.

Each hash table operation has

- an *amortized cost*, which is the number of dollars that are "charged" to do that operation, and
- an *actual cost*, which is the actual number of constant-time computations the operation performs.

If an operation's amortized cost exceeds its actual cost, the extra dollars are saved in the bank to be spent on later operations. If an operation's actual cost exceeds its amortized cost, dollars are withdrawn from the bank to pay for an unusually expensive operation.

If the bank balance goes into surplus, it means that the actual total running time is even faster than the amortized costs imply.

The bank balance must never fall below zero. If it does, you are spending more total dollars than your budget claims, and you have failed to prove anything about the amortized running time of the algorithm.

Think of amortized costs as an allowance. If you have a $500 per month allowance, and you only spend $100 of it each month, you can save up the difference and eventually buy a car. The car may cost $30,000, but if you don't go into debt, your *average* spending obviously wasn't more than $500 per month.

1.3 "Accounting" Analysis of Hash Tables

Every operation (insert, find, remove) takes one dollar of actual running time unless the hash table is resized. We resize the table in two cases. An insert operation doubles the size if the load factor would otherwise exceed one, and the remove operation halves its size if the load factor would drop below 0.25. By trial and error, we might come up with the following amortized costs: insert and remove - \$5, find - \$1.

Is this accounting valid, or will we go broke?

Let's consider a hash table that is new, or has just been resized. If it has N buckets, then it has between $N/2 - 1$ and $N/2 + 1$ items. You can see this by checking all three cases: if the table is new, it has one bucket and zero items; if it's just been doubled, it has $N/2 + 1$ items; if it's just been halved, it has $N/2 - 1$ items.

This means that we can look at a hash table at any time, and guess a lower bound for how many dollars are in the bank from the values of n and N. We know that the last time the hash table was resized, the number of items n was in the range $N/2 - 1...N/2 + 1$. For every step n takes outside this range (by insert or remove), we accumulate another \$4 in the bank. So there must be at least $4(|n - N/2| - 1)$ dollars saved. Note that $4(|n - N/2| - 1)$ is a function of the data structure, and does not depend on the history of hash table operations performed.

We charge an amortized \$5 for an insert operation. Every insert operation costs one actual dollar and puts the other \$4 in the bank. An insert operation only resizes the table if the number of items n reaches $N + 1$. According to the formula above, there are at least $2N$ dollars in the bank. The cost of resizing the hash table from N to $2N$ buckets is N dollars, so we can afford it.

We charge an amortized \$5 for a remove operation. Every remove operation costs one actual dollar and puts the other \$4 in the bank. A remove operation only resizes the table if the number of items n drops to $N/4 - 1$. According to the formula above, there are at least N dollars in the bank. The cost of resizing the hash table from N to $N/2$ buckets is N dollars, so we can afford it.

The bank balance never drops below zero, so the amortized cost of all three operations is in O(1). Observe that if we alternate between inserting and deleting the same item over and over, the hash table is never resized, so we save up a lot of money in the bank. This isn't a problem; it just means the algorithm is faster (spends fewer dollars) than the amortized costs indicate.

1.4 Why Does Amortized Analysis Work?

Why does this metaphor about putting money in the bank tell us anything about the actual running time of an algorithm?

Suppose our accountant keeps a ledger with two columns: the total amortized cost of all operations so far, and the total actual cost of all operations so far. Our bank balance is the sum of all the amortized costs in the left column, minus the sum of all the actual costs in the right column. If the bank balance never drops below zero, the total actual cost is less than or equal to the total amortized cost.

Therefore, the total running time of all the actual operations never takes longer than the total amortized cost of all the operations.

Amortized analysis (as presented here) only tells us an upper bound (big-Oh) on the actual running time, and not a lower bound (big-Omega). It might happen that we accumulate a big bank balance and never spend it, and the total actual running time might be much less than the amortized cost. For example, splay tree operations take amortized O(log n) cost, but if your only operation is to access the same item n times in a row, the actual average running time is in O(1).

Amortized analysis of splay trees is rather complex and we will not go through it here; a treatment of the topic can be found in Weiss, among other places.

2 Randomized Analysis

Randomized analysis, like amortized analysis, is a mathematically rigorous way of saying, "The worst-case running time of this operation is slow, but nobody cares, because on average it runs fast." Unlike amortized analysis, the "average" is taken over an infinite number of runs of the program.

Randomized algorithms make decisions based on "rolls of the dice" or "coin flips". The random numbers actually help to keep the running time low. A randomized algorithm can occasionally run more slowly than expected, but the probability that it will run *asymptotically* slower is extremely low.

The randomized algorithms we've studied so far are quicksort and quickselect.

2.1 Expectation

Suppose a method x() flips a coin. If the coin comes up heads, x() takes one second to execute. If it comes up tails, x() takes three seconds.

Let X be the exact running time of one call to x(). With probability 0.5, X is 1, and with probability 0.5, X is 3. For obvious reasons, X is called a *random variable*.

The *expected* value of X is the average value X assumes in an infinite sequence of coin flips, $E[X] = 0.5 * 1 + 0.5 * 3 = 2$ seconds expected time.

Suppose we run the code sequence

```
x();     // takes time X
x();     // takes time Y
```

and let Y be the running time of the *second* call. The total running time is T = X + Y. (Y and T are also random variables.) What is the expected total running time $E[T]$?

The main idea from probability we need is called *linearity of expectation*, which says that the expected running times sum linearly, so $E[X + Y] = E[X] + E[Y] = 2 + 2 = 4$ seconds expected time.

The interesting thing is that linearity of expectation holds true whether or not X and Y are *independent*. Independence means that the first coin flip has no effect on the outcome of the second. If X and Y are independent, the code will take four seconds on average. But what if they're not? Suppose the second coin flip always matches the first–we always get two heads, or two tails. Then the code still takes four seconds on average. If the second coin flip is always the opposite of the first–we always get one head and one tail– the code still takes four seconds on average.

So if we determine the expected running time of each individual operation, we can determine the expected running time of a whole program by adding up the expected costs of all the operations.

2.2 Quicksort

Assume we're sorting an array in which all the keys are distinct (since this is the slowest case, and takes the same amount of time as when they are not, if we don't do anything special with keys that equal the pivot). Quicksort chooses a random pivot. The pivot is equally likely to be the smallest key, the second smallest, the third smallest, ..., or the largest. For each key, the probability is $1/n$.

Let $T(n)$ be a random variable equal to the running time of quicksort on n distinct keys. Suppose quicksort picks the ith smallest key as the pivot.

Then we run quicksort recursively on a list of length $i-1$ and on a list of length $n-i$. It takes O(n) time to partition and concatenate the lists–let's say at most n dollars–so the running time is

$$T(n) \leq n + T(i-1) + T(n-i).$$

Here i is a random variable that can be any number from 1 (pivot is the smallest key) to n (pivot is largest key), each chosen with probability $1/n$, so

$$E[T(n)] \leq \sum_{i=1}^{n} \frac{1}{n} - (n + E[T(i-1)] + E[T(n-i)])$$

This equation is called a *recurrence*, and you'll learn how to solve them in CS 170 (you do not need to know how for this class). The base cases for the recurrence are $T(0) = 1$ and $T(1) = 1$. This means that sorting a list of length zero or one takes at most one dollar (unit of time).

Here, we'll solve the recurrence by guessing an answer. Then we'll prove by induction that it works out. Let's guess that

$$E[T(j)] \leq 1 + 8j \log_2 j.$$

To prove this is true, we use induction. The base cases of the induction are T(0) and T(1), for which the guess is true. For the inductive step, assume that the inequality holds for all j ¡ n; we'll prove that it holds for j = n as well. From the original recurrence, we have

$$E[T(n)] \leq \sum_{j=0}^{n-1} (1 + \frac{2}{n} E[T(j)])$$

(setting $j = i-1$ in $\sum E[T(i-1)]$, and $j = n-i$ in $\sum E[T(n-i)]$).

$$\leq n + \frac{2}{n} - \sum_{j=0}^{n-1} (1 + 8j \log_2 j)$$

$$< n + \frac{2}{n}(n + \sum_{j=0}^{n/2} (8j \log_2(\frac{n}{2})) + \sum_{j=n/2+1}^{n-1} (8j \log_2(n))$$

$$= n + \frac{2}{n}(n + (n^2 + 2n)(\log_2(n) - 1) + (3n^2 - 6n) \log_2 n)$$

$$= 8n \log_2(n) - n - 8 \log_2(n) - 2.$$

$$< 1 + 8n \log_2(n)$$

which completes the inductive proof that $E[T(j)] \leq 1 + 8j \log_2 j$.

The expression $1 + 8j \log_2 j$ might be an overestimate, but it doesn't matter. The important point is that this proves that $E[T(n)]$ is in $O(n \log n)$. In other words, the expected running time of quicksort is in $O(n \log n)$.

2.3 Amortized Time vs. Expected Time

There's a subtle but important difference between amortized running time and expected running time.

Quicksort with random pivots takes $O(n \log n)$ expected running time, but its worst-case running time is in $\Theta(n^2)$. This means that there is a small possibility that quicksort will cost (n^2) dollars, but the probability that this will happen approaches zero as n grows large.

A splay tree operation takes $O(\log n)$ amortized time, but the worst-case running time for a splay tree operation is $\Theta(n)$. Splay trees are not randomized, and the "probability" of an n-time splay tree operation is not a meaningful concept. If you take an empty splay tree, insert the items 1...n in order, then run find(1), the find operation *will* cost n dollars. But a sequence of n splay tree operations, starting from an empty tree, *never* costs more than O(n log n) actual running time. Ever. (You might think, "but what if we keep trying to find the item furthest down the tree?". But the splaying that you do as you find items deep in the tree will make it more and more balanced.)

Hash tables are an interesting case, because they use both amortization and randomization. We can model the behavior of good hash codes with random numbers. Resizing takes $\Theta(n)$ time. With a "random" hash code, there is a tiny probability that every item will hash to the same bucket, so the worst-case running time of an operation is $\Theta(n)$ - even without resizing.

To account for resizing, we use amortized analysis. To account for collisions, we use randomized analysis. So when we say that hash table operations run in O(1) time, we mean they run in O(1) *expected, amortized* time.

Splay trees	$O(\log n)$ amortized time per operation
Quicksort	$O(n \log n)$ expected time
Quickselect	$\Theta(n)$ expected time
Hash tables	$\Theta(1)$ expected amortized time per operation