CS61B: Lecture 37
Wednesday, April 23, 2014

AMORTIZED ANALYSIS
==================
We've seen several data structures for which I claimed that the average time
for certain operations is always better than the worst-case time: hash tables,
tree-based disjoint sets, and splay trees.

The mathematics 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 earlier, cheaper operations.

The Averaging Method

Most hash table operations take O(1) time, but sometimes an operation forces
a hash table to resize itself, at great expense. What is the average time to
insert an item into a hash table with resizing? Assume that the chains never
grow longer than O(1), so any operation that doesn't resize the table takes
O(1) time--more precisely, suppose it takes at most one second.

Let n be the number of items in the hash table, and N the number of buckets.
Suppose it takes one second for the insert operation to insert the new item,
increment n, and then check if n = N. If so, it doubles the size of the table
from N to 2N, taking 2N additional seconds. This resizing scheme ensures that
the load factor n/N is always less than one.

Suppose every newly constructed hash table is empty and has just one bucket--
that is, initially n = 0 and N = 1. After i insert operations, n = i. The
number of buckets N must be a power of two, and we never allow it to be less
than or equal to n; so N is the smallest power of two > n, which is <= 2n.

The total time in seconds for _all_ the table 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 <= 5i - 2 seconds. Therefore, the
average running time of an insertion operation is (5i - 2)/i = 5 - 2/i
seconds, which is in O(1) time.

We say that the _amortized_running_time_ of insertion is in O(1), even though
the worst-case running time 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
really slow operation might cause a noticeable glitch in response time.

The Accounting Method

Consider hash tables that resize in both directions: not only do they expand
as the number of items increases, but they also shrink as the number of items
decreases. You can't analyze them with the averaging method, because you don't
know what sequence of insert and remove operations an application might
perform.

Let's try a more sophisticated method. In the _accounting_method_, we "charge"
each operation a certain amount of time. Usually we overcharge. When we
charge more time than the operation actually takes, we can save the excess time
in a bank to spend on later operations.

Before we start, let's stop using seconds as our unit of running time. 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 a sequence of constant-time computations. 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. A dollar is a real unit of
time, but it's different for different computers.

Each hash table operation has
- an _amortized_cost_, which is the number of dollars that we "charge" to do
  that operation, and
- an _actual_cost_, which is the actual number of constant-time computations
  the operation performs.

The amortized cost is usually a fixed function of n (e.g. $5 for insertion into
a hash table, or $2 log n for insertion into a splay tree), but the actual cost
may vary wildly from operation to operation. For example, insertion into a
hash table takes a long, long time when the table is resized.

When an operation's amortized cost exceeds its actual cost, the extra dollars
are saved in the bank to be spent on later operations. When 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 total 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 your dad gives you $500 a 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
saved that money and don't go into debt, your _average_ spending obviously
wasn't more than $500 a month.

Accounting of Hash Tables

Suppose every operation (insert, find, remove) takes one dollar of actual
running time unless the hash table is resized. We resize the table in two
circumstances.
- An insert operation doubles the table size if n = N AFTER the new item is
  inserted and n is incremented, taking 2N additional dollars of time for
  resizing to 2N buckets. Thus, the load factor is always less than one.
- The remove operation halves the table size if n = N/4 AFTER the item is
  deleted and n is decremented, taking N additional dollars of time for
  resizing to N/2 buckets. Thus, the load factor is always greater than 0.25
  (except when n = 0, i.e. the table is empty).

Either way, a hash table that has _just_ been resized has n = N/2.
A newly constructed hash table has n = 0 items and N = 1 buckets.

By trial and error, I came up with the following amortized costs.

    insert: 5 dollars
    remove: 5 dollars
    find: 1 dollar

Is this accounting valid, or will we go broke?

The crucial insight is that at any time, we can look at a hash table and know 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 exactly N/2. So if n != N/2, there have been subsequent insert/remove
operations, and these have put money in the bank.

We charge an amortized $5 for an insert or remove operation. Every insert or
remove operation costs one actual dollar (not counting resizing) and puts the
remaining $4 in the bank to pay for resizing. For every step n takes away from
N/2, we accumulate another $4. So there must be at least $4|n - N/2|$ dollars
saved (or 4n dollars for a never-resized one-bucket hash table).

IMPORTANT: Note that $4|n - N/2|$ is a function of the data structure, and does
NOT depend on the history of hash table operations performed. In general, the
accounting method only works if you can tell how much money is in the bank (or,
more commonly, a minimum bound on that bank balance) just by looking at the
current state of the data structure--without knowing how the data structure
reached that state.

An insert operation only resizes the table if the number of items n reaches N.
According to the formula $4|n - N/2|$, there are at least 2N dollars in the bank.
Resizing the hash table from N to 2N buckets costs 2N dollars, so we can afford
it. After we resize, the bank balance might be zero again, but it isn't
negative.

A remove operation only resizes the table if the number of items n drops to
N/4. According to the formula $4|n - N/2|$, there are at least N dollars in the
bank. Resizing the hash table from N to N/2 buckets costs N dollars, so we can
afford it.

The bank balance never drops below zero, so my amortized costs above are valid.
Therefore, 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 my amortized costs indicate.

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.

Total amortized cost	Total actual cost
$5	$1
$1	$1
$5	$3
.	.
.	.
.	.
$5	$1
$5	$2,049
$1	$1
$12,327	>= $10,333

Therefore, the total running time of all the actual operations is never 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) time, where n is the number of items in
the tree, but if your only operation is to find the same item n times in a row,
the actual average running time is in O(1).

If you want to see the amortized analysis of splay trees, Goodrich and Tamassia
have it. If you take CS 170, you'll see an amortized analysis of disjoint
sets. I am saddened to report that both analyses are too complicated to
provide much intuition about their running times. (Especially the inverse
Ackermann function, which is ridiculously nonintuitive, though cool
nonetheless.)