Illinois Institute of Technology
Department of Computer Science

Lectures 13–14: October 13–18, 2010

CS 430 Introduction to Algorithms
Fall Semester, 2010

1 Amortized Analysis

So far, we have analyzed best and worst case running times for an operation without considering its context. With amortized analysis, we study a sequence of operations rather than individual operations.

Consider the operation of a change machine that accepts dollar bills and gives out quarters. Every so often, the machine will run out of quarters and require the services of a technician to refill it – a process that could require a week of waiting. Thus, at the worst case, you could have to wait a week before you could get change; a worst-case analysis would then say that it might take you 10 weeks to get change for $10! This is clearly an absurd analysis, however, since the change machine stores enough quarters to run for several weeks without being refilled. Thus, the technician would need to be called at most one time (if you are unlucky enough) during the time you try to change your $10. This gives an amortized time of 1 week and a couple of minutes for you to get your change, which is somewhat better than our first analysis. We can tolerate one expensive operation (e.g. calling the technician), if the rest are guaranteed to be cheap thereafter.

Formally speaking, the amortized time for an operation is the *worst-case average* behavior of an operation over a sequence of operations. If, given **any** n operations $o_1, o_2, o_3, \ldots o_n$ we can bound their average cost:

$$\frac{1}{n}\sum_i \text{cost}(o_i) \leq \text{BOUND}$$

then we can say that BOUND is an amortized cost of one operation. Typically, we will rewrite the above equation:

$$\sum_i \text{cost}(o_i) \leq n \times \text{BOUND}$$

and analyze its left-hand side.

There are three different methods of amortized analysis: the *aggregate* method, the *accounting* method, and the method of *potential functions*. Each of these are equivalent in power, in that any analysis we run with one method can be reworked using another method. Nevertheless, we will tend to use the method of potential functions the most because it will conceal much of the menial labour associated with amortized analysis.

1.1 Aggregate method

Richard P. Feynman, a Noble-prize winning professor at Caltech, was once asked to explain the "Feynman technique" for solving a problem. He responded that his method of solving a problem is to:

- write down the problem
- stare at the problem really hard
- write down the solution to the problem

This is a degenerate form of the aggregate method. In the aggregate method, we try to deduce the total worst-case running time $T(n)$ for n operations, and then divide this number by n to get the amortized cost $\frac{T(n)}{n}$.

A stack Consider the example of an (originally empty) stack with two operations:

- `pop`—removes an element from the stack
- `push`—adds an element onto the stack

Clearly `pop` and `push` can be easily implemented in $O(1)$ time with a reasonable data-structure. Thus, any n such operations would require $O(1) \times n = O(n)$ time, giving an amortized running time of $O(1)$ per operation. This is not a particularly enlightening use of the aggregate method.

Let us introduce another operation to the stack:

- `multi-pop(k)` – remove k elements from the top of the stack, or simply empty the stack if there are fewer than k elements present.

Since k could be $\Theta(n)$ in the worst case, the `multi-pop` operation would require $O(n)$ time. Thus, n stack operations would give a worst-case running time of $n \times O(n) = O(n^2)$. Clearly, however, this is an overestimate. For one, we can only pop off the stack as many elements as we had originally "pushed" on; we cannot just keep `multi-pop`ping off the stack.

An aggregate analysis reveals that if we do n stack operations, we can put at most n items onto the stack with the `push` operation, so that, even if we take all of the items off the stack, we will be doing at most $O(n)$ work in total. Thus, the amortized time for a stack operation is still $O(n)/n = O(1)$.

A binary counter Consider, as a second example, a binary counter that is being implemented in hardware. Assume that the machine on which it is being run can flip a bit as its basic operation. We now want to analyze the cost of counting up from 0 to n (using k bits):

Decimal	Binary
1	000001
2	000010
3	000011
4	000100
5	000101
⋮	⋮
n	100110

A naive worst-case analysis would argue that each time we increment the counter, we can, at worst, flip each of the k bits giving $\Theta(kn)$ total bit flips for counting up to n.

Using the aggregate method, however, we can perform a more careful analysis for n increments:

- Every time we increment the counter, the ones bit flips = n flips in total
- Every second time we increment the counter, the twos bit flips = $\frac{n}{2}$ flips total
- Every fourth time we increment the counter, the fours bit flips = $\frac{n}{4}$ flips total

$\vdots$

Thus, altogether, the number of bit flips needed is:

$$n + \frac{n}{2} + \frac{n}{4} + \ldots \leq 2n$$

This gives an amortized time of $2n/n = 2$ bit flips per increment.

1.2 Accounting

Rather than looking at the whole picture, the accounting method looks at individual operations and assigns them amortized costs, just like an accountant would do.

A good analogy is that of a business. Suppose that you were running a business for data structures. You would charge a set price for each of your data structure operations; this price would be higher than your cost, so that, after the whole day is over, you will have made a profit, even if you take an individual loss on one particular operation. Let's take a look at a specific example.

Stack Consider the stack problem that we saw in the aggregate example, and suppose that you have a graduate student working for you who charges $1 for each manipulation of the stack.

You might set up the following price list:

item	price
`push`	$2
`pop`	$0
`multi-pop`	$0

Now, when a customer comes in with a request to `push` item onto the stack, you would pay the graduate student $1 for the job, and ask him to staple the second dollar onto the item. When the customer comes back with a `pop` request, you would instruct the graduate student to get the item, and keep the stapled dollar for himself as payment. For a `multi-pop`, the student would simply collect stapled dollars from all the items he takes off the stack.

You can readily see that you will not lose money on this scheme, and might even gain some money if people leave their items on the stack. Thus, the prices in your original list are amortized costs of the respective operations.

Binary counter We may use the accounting principle to pay for incrementing a binary counter in a similar fashion to the stack example. In this case, we will charge customers $2 for incrementing the counter. Since each increment converts a bit from zero to one, you can think of this as $2 to flip a bit from 0 to 1 and $0 to flip a bit from 1 to 0.

Again, when a customer comes pays $2 for an increment, you would pay your employee $1 for the bit that flip from 0 to 1. The other dollar is stapled to this bit. In order to pay for flipping the bits which change from 1 to 0, the student takes the dollar stapled to those bits.

1.3 Potential functions

Though the aggregate and accounting methods of amortized analysis are powerful, they also tend to involve a very tedious ands careful examination of the problem cases. The potential function method of amortized

analysis hides most of the tedium in a cleverly chosen potential function.

The potential function method involves maintaining a potential function of the data structure (akin to keeping all the extra bills in your pocket instead of stapling them to elements in the data structure). This potential function is entirely an artifact of the amortized analysis and is arbitrarily chosen to prove the desired amortized costs.

More formally, a potential function Φ is a function that maps a data structure to a positive integer. For example, one possible potential function on a binary tree would return the height of the tree.

Now, denote the amortized cost (i.e. what you would charge the customer) of the i'th operation in some problem by $\hat{c_i}$. Denote the actual cost of the i'th operation (i.e. what you would have to pay your graduate student for the work) by c_i. Then the amortized and actual costs are related to the potential function as follows:

$$\hat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) \tag{1}$$

where $\Phi(D_i)$ represents the potential function applied to the data structure after i operations.

Summing equation 1 over all i causes the potential functions to cancel in a telescoping fashion:

$$\sum_{i=1}^{n} \hat{c_i} = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

In other words,

$$\begin{aligned} \sum_{i=1}^{n} c_i &= \sum_{i=1}^{n} \hat{c_i} - \Phi(D_n) + \Phi(D_0) \\ \text{ACTUAL COST} &= \text{AMORTIZED COST} - \Phi(D_n) + \Phi(D_0) \end{aligned}$$

ACTUAL COST corresponds to the customer's charge and AMORTIZED COST corresponds to the graduate student's salary in the previous example.

Stack Let us return to our lovable stack example. Let us chose our potential function to be

$$\Phi(D_i) = \text{the number of items on the stack}$$

We furthermore assume that:

$$\begin{aligned} \Phi(D_0) &= 0 \\ \Phi(D_n) &\geq 0 \end{aligned}$$

We may now put equation 1 into action, lightly abusing the notation:

$$\begin{aligned} \text{AMORT}_{\texttt{pop}} &= \text{ACTUAL}_{\texttt{pop}} + |\text{stack}|_{(\text{after } \texttt{pop})} - |\text{stack}|_{(\text{before } \texttt{pop})} \\ \text{AMORT}_{\texttt{push}} &= \text{ACTUAL}_{\texttt{push}} + |\text{stack}|_{(\text{after } \texttt{push})} - |\text{stack}|_{(\text{before } \texttt{push})} \end{aligned}$$

We know that the actual cost of doing either a `push` or a `pop` is \$1. Moreover, by the very definitions of the stack operations,

$$\begin{aligned} |\text{stack}|_{(\text{after } \texttt{pop})} - |\text{stack}|_{(\text{before } \texttt{pop})} &= -1 \\ |\text{stack}|_{(\text{after } \texttt{push})} - |\text{stack}|_{(\text{before } \texttt{push})} &= 1 \end{aligned}$$

Thus, putting everything together, $\text{AMORT}_{\texttt{pop}} = 0$ and $\text{AMORT}_{\texttt{push}} = 2$, just as we had set our price list with the accounting method.

Binary counter We can use the potential function

$$\Phi(n) = \text{the number of 1 bits in the binary representation of } n$$

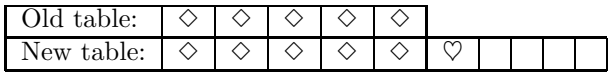
to compute the same amortized costs for the various tasks involved with incrementing a counter. The result is the same as they were for the accounting version of the analysis. It is a good exercise for you to try to fill in the steps yourself.

2 Hash tables

So far we have considered very simple problems for the purpose of introducing the concepts involved with amortized analysis. We will now look at a slightly more sophisticated example that uses hash tables.

Hash tables are covered in chapter 11 of CLRS, but, for the purposes of this example, you just need to remember that a hash table is a table of elements that grows and shrinks through insertions and deletions. The problem is that once the table gets completely filled up (the maximum size is fixed by the particular implementation), the program implementing it crashes.

We can, however, consider a simple amendment to the table that would avoid this problem. Specifically, suppose that each time we attempt to insert an item into a table that is already full (with n items), we allocate $2n$ new memory cells, copy the original table to this new location, and then insert our item accordingly. As an example, consider the insertion of the item ♡:



Since every item of the old table must be moved to enlarge the table, this operation needs time $\Theta(n)$. Insertion when there is room on the table, on the other hand, can be accomplished in $O(1)$ time.

A naive worst-case analysis of n insertions would give a time bound of $\Theta(n) \times n = O(n^2)$. By now, however, you should be fairly suspicious of this naive analysis and insist on an amortized calculation.

Aggregate method Consider inserting n items into the table. We want to analyze when we will be required to enlarge the table, and how many items will have to be moved. The first time we enlarge the table is after we have inserted 1 item (assuming the table starts out with size 1); this will require moving 1 item. The second time we enlarge the table is after we have inserted 2 items and will require 2 moving operations. The third time we enlarge the table will be when 8 items have been inserted into the table.

In general, after 2^{j-1} items have been inserted into the table, we will have to perform the j'th table enlargement and move all the items into the new table. Since we are inserting n items in total, it must be that $2^{j-1} \leq n$ which implies that $j \leq \lg(n) + 1$. Thus, the total number of items that will be moved by the enlargement operation would be:

$$\sum_{j=1}^{\lg n+1} 2^{j-1} \approx 2n$$

If we add the n operations needed for actually inserting items into a non-full table, then we get an overall running time of $3n$, or an amortized running time of $\frac{3n}{n} = 3$ steps per insertion.

Accounting We can analyze the same problem using the accounting method. Specifically, we can charge \$3 per insertion. When a customer comes to us with an insertion request, we will pay our graduate student \$1 for the labor, staple \$1 to the item, and keep \$1 in our hand.

If the graduate student has to enlarge the table, he will first be paid for the movements of the various items with the \$1 that is stapled to them. If an item does not have a stapled \$1, then we pay the graduate student from our hand. The question is: will we always have enough money in our hand to pay for these indigent items?

The key to the argument lies in the fact that every time we enlarge the table, we double its size. By indcution, if the table has n elements, half of the elements must be indigent and have no dollars stapled to them; the other half of the elements (the wealthy elements) can pay for their own movement with their stapled money. However, the wealthy elements also contributed $\frac{n}{2}$ dollars when they were inserted after the previous table enlargement. This money can now be used to subsidize the indigent items. It is a good exercise of your understanding of the accounting method to go through the previous paragraph and convince yourself that the accounting method works in this case.

Potential functions A final method of attacking our problem is with the use of potential functions. In this case, a potential function that works is:

$$\Phi(T) = 2 \times \text{the number of items in the table} - \text{the size of the table}$$

See the CLRS, page 466, for a proof with this potential function giving an amortized price of \$3 per insertion.

Deletion Just as we considered inserting elements into a hash table, we may also consider deleting them. A simple implementation would merely delete a specified item from the table when requested. However, it is often advantageous to contract the table upon a deletion if possible. For example, when deleting the ♡ from the previous hash-table example, it would be a good idea to contract the resulting table from 8 cells to 4 cells.

The question of when to contract the table is a difficult one. If we contract the table whenever it is half full, then we could end up with a very bad situation of repeated insert-delete combinations that cause the table to expand and contract. In this case, n operations could cost $\Theta(n^2)$ time.

Thus, it might make more sense to contract the table only when it is a quarter full so as to prevent this problem. Using a new potential function, we can see that insertion and deletion can be done in an amortized $O(1)$ time. This is discussed in further detail in section 17.4.2 of CLRS; read it carefully.