15-750 Graduate Algorithms January 23, 2001

* Amortized analysis
(Lecture notes by D. Sleator, adapted from those of A. Blum)

Amortized analysis

-- definition and motivation
-- example of counting
-- example of implementing a FIFO queue with two stacks
-- example of doubling-array for stack.
-- potential functions

Amortized analysis means analyzing time-averaged cost for a sequence of operations.

Motivation is that traditional worst-case-per-operation analysis can give overly pessimistic bound if the only way of having an expensive operation is to have a lot of cheap ones before it.

NOTE: this is DIFFERENT from our usual notion of "average case analysis" -- we're not making any assumptions about inputs being chosen at random -- we're just averaging over time.

The approach is going to be to somehow assign an artificial cost to each operation in the sequence. This artificial cost is called the _amortized cost_ of an operation. The key property required of amortized cost is that the total real cost of the sequence should be bounded by the total of the amortized costs of all the operations. Then, for purposes of analyzing an algorithm that, say, accesses a data structure, it is okay to just use the amortized cost instead of the actual cost of the operation. This will give you correct results.

Note: There is sometimes flexibility in the assignment of amortized costs.

There are going to be three approaches that we call:

The aggregate method
The banker's method (tokens in the data structure)
The physicist's method (potential functions)

(The physicist's method is just a slightly more formal version of the banker's method, as we'll see.)

We'll illustrate these methods through some examples.

Example1: a binary counter

Say we want to store a big binary counter in an array A: Start all entries at 0. The operation we are going to implement and analyze is that of counting.

The algorithm we'll use for incrementing this counter is the usual one. We toggle bit A[0]. If it changed from 0 to 1, then we toggle bit A[1], etc. We stop when a bit changes from 0 to 1. The cost of an increment is the number of bits that change.

A[m]	A[m-1]	.....	A[3]	A[2]	A[1]	A[0]	cost
0	0		0	0	0	0	
							1
0	0		0	0	0	1	

```
                                                    2
    0     0           0    0    1    0
                                                    1
    0     0           0    0    1    1
                                                    3
    0     0           0    1    0    0
                                                    1
    0     0           0    1    0    1
                                                    2
    0     0           0    1    1    0
```

The number of bits that change when the increment produces a number n is at most 1+floor(lg n). (That's just the number of bits in the binary representation of n.)

Thus, in a sequence of n increments, worst-case cost per increment is bounded by n(1+floor(lg n)) = O(n log n).

But, what is our *amortized* cost per increment? Answer: 2.

Proof 1 (aggregate method): how often do we flip A[0]? Answer: every time. how often do we flip A[1]? Answer: every other time. How often do we flip A[2]? Answer: every 4th time. Etc. So, total cost spent on flipping A[0] is n, total cost of A[1] is floor(n/2), total cost on A[2] is floor(n/4), etc. So, the total cost is:

total cost = n + floor(n/2) + floor(n/4) + ...

total cost <= n + n/2 + n/4 + n/8 + ... <= 2n

So the total cost is 2n, which means the amortized cost of an increment is 2.

Proof 2 (banker's method): Let's us a kind of accounting trick. On every bit that is a 1, let's keep a dollar on that bit. So for example, if the current count is 6, we'd have:

```
                      $    $
  array: 0  ....  0   1    1    0
```

We'll use the convention that whenever we toggle a bit, we must pay a dollar to do that.

Let's say we allocate $2 to do an increment. Let's see how much it costs to do the increment. In general, a bunch of low order bits change from 1 to 0, and then one bit changes from a 0 to a 1, and the process terminates. For each of the bits that changes from 1 to 0, we have a dollar sitting on the bit to pay for toggling that bit. For the bit that changes from a 0 to a 1, we have to pay a dollar to toggle the bit, then put a dollar on that bit (for future use). Thus, having allocated $2 for the increment always guarantees that we will have enough money to pay for the work, no matter how costly the increment actually is.

This completes proof 2 that the amortized cost of the increment is 2.

Example 2: Implementing a FIFO queue with two stacks

Say you have a stack data type, and you need to implement a FIFO queue. The stack has the usual POP and PUSH operations, and the cost of each operation is 1.

We can implement a FIFO queue using two stacks as follows.

The FIFO has two operations: ENQUEUE and DEQUEUE:

```
ENQUEUE(x):  Push x onto stack1.

DEQUEUE():   if stack2 is empty, then we POP the
             entire contents of stack1 and PUSH it
             into stack2.  Now simply POP from
             stack2 and return the result.
```

It's easy to see that this algorithm is correct. Here we'll just worry about the running time. (I'm going to ignore the cost of checking of stack2 is empty, and only measure the cost in terms of the number of PUSHs and POPs that are done.)

Claim: the amortized cost of ENQUEUE is 3 and DEQUEUE is 1.

Proof 1 (aggregate method): As an element flows through the two-stack data structure, it's PUSHed at most twice and POPPed at most twice. This shows that we can assign an amortized cost of 4 to ENQUEUE and 0 to DEQUEUE.

To get the desired result, note that if an element is not DEQUEUED it's only PUSHED twice and POPPED once. So the cost of 3 is paid for by the cost of 3 per ENQUEUE. The last POP is paid for by the DEQUEUE (if it happens).

Proof 2 (banker's method): Maintain a collection of tokens on stack1. In fact, keep 2 tokens for each thing in the stack.

When we ENQUEUE, we have three tokens to work with. We use one to push the element onto stack1, and the other two are put on the element for future use. When we have to move stack1 into stack2, we have enough tokens in the stack to pay for the move (one POP and one PUSH for each element). Finally, the last pop done by the DEQUEUE is paid for by the 1 we allocated for it. QED.

Example 3: doubling array

We'll use an array to implement a stack that allows push and pop operations. We represent the stack as an array, A[] and an integer variable top that points to the top of the stack. Here is a naive implementation of push and pop:

```
push(x):  top++;  A[top] = x;
pop;      top--;  return A[top+1]
```

These operations are both constant time (cost=1). The problem is, what happens when the array gets full? To deal with this, we keep another variable L, storing the size of the array, and a variable k keeping count of the number of elements of the array that are in use. When we are about to overflow the available space (k=L), we allocate an array twice as large, move the data over to the new array, free the old array, and double L. This operation is called "doubling". It will be convenient to analyze it as thought it was a separate operation on the data strucure, that occurs only when k=L.

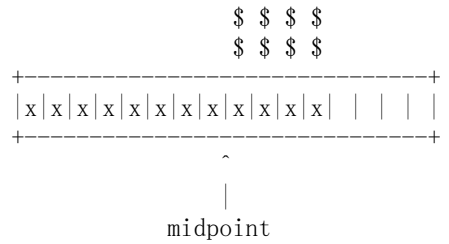
If the stack is full, and has size L, and we apply the doubling operation, the cost of the operation is L, because we have to move L items into the new array. (The cost of allocating and freeing the arrays is O(L).)

Starting from an empty stack, doing any sequence n pushes

and pops, how much does this cost?

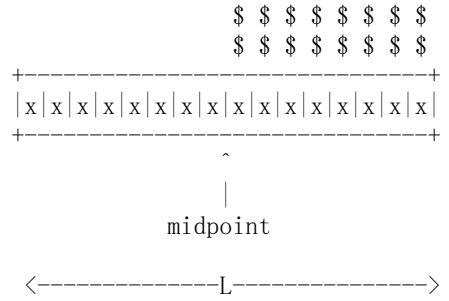
First analysis: Total cost for pushes and pops is n. Total cost for doubling is at worst 1 + 2 + 4 + ... n/2 + n < 2n So, the total cost is 3n. Therefore we can say that the amortized cost of an operation is 3.

Second analysis: Again, we use the financial approach. We'll keep money lying around the data structure in the following way.

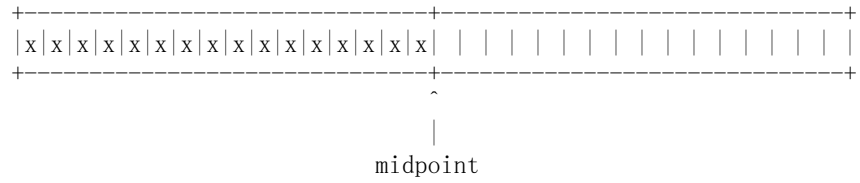


We'll keep $2 on each element stored in the stack that is BEYOND THE MIDPOINT of the current array. In the normal case when we do a push (the array is not full), we need $1 to do the work, and then at most $2 to put onto the the new item we just pushed. Thus the cost is $3. (pop is even cheaper.) Now what happens if we have to do a doubling operation?

Before doubling:



After doubling:



You can see that the money that we had on the items (L) was sufficient to pay for all the work of doubling the array (L). We conclude that if we allocate $3 for each operation, we'll never run out of money. Thus the amortized cost of an operation is 3.

In these examples, the financial model was not really necessary to get the desired results. However, later we'll see examples where this approach is necessary.

Potential Functions. (The Physicist's method)

Let's say that instead of distributing our money all over the data structure (as we did above), we keep it all in a piggy bank. What's really important is how much money is in

the piggy bank. We'll call the amount of money in this bank the "potential function" (Phi) for the problem.

So for the two problems above, we used the following two potential functions:

Phi(counter) = # of one bits in the counter

Phi(queue) = 2 * the size of stack 1.

$$\text{Phi(stack)} = \begin{cases} 2(k-L/2) & \text{if } k >= L/2 \\ 0 & \text{otherwise} \end{cases}$$

(Here L is the current array size, and k is the number of elements currently in the stack.)

Using this formalism we can define the amortized cost of an operation. Say the system changes from state S to state S' as a result of doing some operation. We define the amortized cost of the operation as follows:

amortized cost = actual cost + Delta(Phi) =
= actual cost + Phi(S') - Phi(S)

This is simply the amount of additional money that we need to maintain our piggy bank and to pay for the work.

For the counter, with the potential function given above:

amortized cost of increment <= 2

For the stack, with the potential function given above:

amortized cost of pop <= 3

How is this amortized cost related to actual cost? Let's sum the above definition of amortized cost over all the operations:

Sigma(amortized cost) = Sigma(actual cost) + Phi(final) - Phi(initial)
or
Sigma(actual cost) = Sigma(amortized cost) + Phi(initial) - Phi(final)

If the potential is always non-negative, and starts at zero (as it is in our examples), then

Sigma(actual cost) <= Sigma(amortized cost)

In this more general framework, the potential can be negative, and may not start at 0. So in general we have to worry about the initial and final potentials.

Summary of using potential functions to do amortized analysis:

(1) Pick a potential function that's going to work (this is art)

(2) Using your potential function, bound the amortized cost of the operations you're interested in.

(3) Bound Phi(initial) - Phi(final)

In terms of (1), one obvious point is that if the actual cost of an operation is HIGH, and you want the amortized

cost to be LOW, then in this case the potential must DECREASE by a lot to pay for it. This is illustrated in both of the examples in this lecture.

We'll see in the next lecture just how essential this formalism is for analyzing splay trees.