Amortized Analysis: CLRS Chapter 17

Last revised: September 15, 2006

In amortized analysis we try to analyze the time required by a *sequence* of operations. There are many situations in which, even though it is possible for an individual operation to be very expensive, the average cost of an operation (taken over all operations performed) can be shown to be small.

Another way of saying this is that *amortized analysis guarantees the average case performance of each operation in the worst case*.

Note that this is very different from what is normally meant by *average-case analysis*; there is no probability at work here.

We will introduce the main ideas behind amortized analysis through the study of two different problems: stack operations with multipop and incrementing a binary counter.

Stack-operations

We allow three operations:

1. **PUSH(S, x):** Push x onto stack S.
2. **POP(S):** Pop top of S and returns popped item
3. **MULTIPOP(S, k):**

```
while $S$ is not empty and $k \neq 0$
    POP($S$)
    $k := k - 1$.
```

Note that **PUSH(S, x)** and **POP(S)** each take only $O(1)$ worst case time but **MULTIPOP(S, k)** can take up to n time in the worst case, where n is the total number of items in the stack.

Even though a **MULTIPOP** can take $\Theta(n)$ time it is not hard to show that *any* sequence of n operations, can use at most $O(n)$ time.

Incrementing a Binary Counter

The following code increments a counter:

```
INCREMENT($A$)
1  $i \leftarrow 0$
2  while $i < length[A]$ and $A[i] = 1$
3        do $A[i] \leftarrow 0$
4           $i \leftarrow i + 1$
5  if $i < length[A]$
6        then $A[i] \leftarrow 1$
```

Counter value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

An Increment operation can require as many as $k =$ length$[A]$ bit flips. We claim, though, that any *sequence* of n Increments requires only $O(n)$ bit flips.

3 different approaches to Amortized Analysis.

- **Aggregate Analysis:**
  Let $T(n)$ be the *total* cost of some sequence of n operations. In the worst case the average cost, or *amortized cost* per operation, is $T(n)/n$. In this type of analysis, every operation will have the same amortized cost.

- **The Accounting Method:**
  Each operation is charged an *amortized* cost, which can be different than actual cost.

  Difference between actual cost and amortized cost is expressed in credits. Extra credits are placed in specific parts of data structure. Lack of credits are made up by using credits previously placed in data structure.

  Different operations can have different amortized costs.
  Sum of the amortized costs of a sequence of operations is $\geq$ total cost of the sequence

- **The Potential Method:**
  The data structure is assigned a *potential (energy)* $\geq$ 0 based on its current configuration.

  Potential of original configuration is usually 0.
  The *amortized cost* of an operation will be the sum of its actual cost plus the difference between the potential before the operation and the potential after the operation.

  Different operations can have different amortized costs.
  Sum of the amortized costs of a sequence of operations is $\geq$ total cost of the sequence.

Aggregate Analysis: Stack-Operations

1. **PUSH(S, x):** Push x onto stack S.
2. **POP(S):** Pop top of S and returns popped item
3. **MULTIPOP(S, k):**

```
while $S$ is not empty and $k \neq 0$
    POP($S$)
    $k := k - 1$.
```

It is easy to see that *any* sequence of n **PUSH**, **POP**, and **MULTIPOP** operations takes $T(n) = O(n)$ time so the average time per operation is $T(n)/n = O(1)$.

We therefore can assign each operation in the sequence an amortized cost of $O(1)$.

Note that there is no *probability* involved. As stated **amortized analysis guarantees the average case performance of each operation in the worst case**.

Aggregate Analysis: Incrementing a counter

Counter value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Every		$A[0]$ bit gets flipped.
Every	2nd	$A[1]$ bit gets flipped.
Every	4th	$A[2]$ bit gets flipped.
Every	8th	$A[3]$ bit gets flipped.
Every	2^ith	$A[i]$ bit gets flipped.

So the total number of bits flipped after n increment operations will be

$$\sum_{i=0}^{k} \left\lfloor \frac{n}{2^i} \right\rfloor \leq n \sum_{i=0}^{k} \frac{1}{2^i} < 2n$$

Every		$A[0]$ bit gets flipped.
Every	2nd	$A[1]$ bit gets flipped.
Every	4th	$A[2]$ bit gets flipped.
Every	8th	$A[3]$ bit gets flipped.
Every	2^ith	$A[i]$ bit gets flipped.

So the total number of bits flipped after n increment operations will be

$$\sum_{i=0}^{k} \left\lfloor \frac{n}{2^i} \right\rfloor \leq n \sum_{i=0}^{k} \frac{1}{2^i} < 2n$$

This means that every operation requires at most $2n/n$ bit flips on average, i.e., has an amortized cost of $O(1)$.

The Accounting method: Stack-Operations

In the Accounting method we assign each operation an amortized cost $\hat{c}$, which can be different for the different operations.

Let c be the actual cost of some operation.

If $\hat{c} > c$ then the algorithm places $\hat{c} - c$ credits on items in the data structure.

If $\hat{c} < c$ then the algorithm must be able to take $c - \hat{c}$ credits off of the data structure at that time to pay for the rest of the operation.

Essentially, earlier actual cheap (real cost) operations are *prepaying* the expense of later more expensive (real cost) operations.

Let $\hat{c_i}$ be the given amortized cost of the ith operation and c_i its actual cost. Then

$$\sum_{i=1}^{n} \hat{c_i} \geq \sum_{i=1}^{n} c_i.$$

In the stack problem we use the following amortized costs:

Operation	**Real Cost** c	**Amortized Cost** $\hat{c}$
PUSH(S, x**)**	1	2
POP(S**)**	1	0
MULTIPOP(S, k**)**	$\min(k, \|S\|)$	0

When doing a **PUSH(**S, x**)**, 1 unit of amortized cost cost pays for the actual **PUSH(**S, x**)** and the 1 remaining unit of credit gets placed on the item in the stack that was just pushed.

Whenever an item in the stack is **(MULTI)POP**ed, it is paid for by the unit of credit sitting on it.

It is easy to see that there are always credits available to pay for a **POP** or **MULTIPOP** so the amortized costs are well defined.

The Accounting method: Incrementing a Counter

Counter value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Note that every increment flips exactly one 0 to be a 1; every 1 that is flipped to be a 0 was originally made into a 1 in a previous operation.

In this case we will charge every increment an amortized cost of 2 units of which 1 unit will be charged for flipping the one 0 to be a 1 and the other unit will be left as a credit on the 1.
All of the flips of 1s into 0s will be paid for using the credits already sitting on those bits.

The Potential Method

In the **The Potential Method**, we assign a **Potential** Φ to the data structure. The potential can be thought of as stored up energy, or credit, that can be used to pay for expensive operations.

The difference between this and the accounting method is that in the accounting method the credits were assigned to *objects* in the data structure while, in the potential method, the potential is a function of the total data structure,

The Potential Method (cont)

Starting with an initial data structure D_0.
Let D_i be the data structure after applying operation i in a sequence to D_{i-1}.
Φ maps every D to a real number $\Phi(D)$.

Let c_i be the real cost of operation i.
The amortized cost $\hat{c}_i$ of operation i is

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

The total amortized cost of n operations is then

$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)\end{aligned}$$

If we define Φ so that $\forall n, \Phi(D_n) \geq \Phi(D_0)$ we get

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i$$

so the amortized cost of a sequence of operations upper-bounds the real cost.

The Potential Method: Stack Operations

Let $\Phi(D)$ be the number of items on stack D.
Starting with empty stack D_0 gives us that
$\forall D, \Phi(D) \geq 0 = \Phi(D_0)$
so the amortized cost of a sequence of operations upper-bounds the real cost.

What are the amortized costs of the operations? Assume that D_{i-1} currently has s items on the stack.
We split into the cases that the i^{th} operation is:

PUSH(S, x):
$\Phi(D_i) - \Phi(D_{i-1}) = (s+1) - s = 1$.
Actual cost is $c_i = 1$. Amortized cost is
$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (s+1) - s = 2$.

Multipop(S, k):
Set $k' = \min(k, s)$, no. items popped off stack.
$\Phi(D_i) - \Phi(D_{i-1}) = (s - k') - s = -k'$.
Actual cost is $c_i = k'$. Amortized cost is
$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k' - k' = 0$.

POP(S):
0, same as **Multipop($S, 1$)**.

Combining, we get the same amortized costs as with the accounting method:

Operation	**Real Cost** c	**Amortized Cost** $\widehat{c}$
PUSH(S, x**)**	1	2
POP(S**)**	1	0
MULTIPOP(S, k**)**	$\min(k, \|S\|)$	0

The Potential Method: Incrementing a counter

In this case we let
$\Phi(D) =$ the number of 1s in the counter.

Suppose that the ith increment operation flips t_i 1 bits to 0; let b_i be the number of 1s in the counter after the operation.

Actual cost is $c_i \le t_i + 1$.

If $b_i = 0$ then increment totally resets the counter and $b_{i-1} = t_i$.
If $b_i > 0$ then $b_i = b_{i-1} - t_i + 1$.
In both cases $b_i \le b_{i-1} - t_i + 1$ so

$$\Phi(D_i) - \Phi(D_{i-1}) \le b_{i-1} - t_i + 1 - b_{i-1} = 1 - t_i.$$

Amortized cost is then,

$$\begin{aligned} \hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\le (t_i + 1) + (1 - t_i) = 2 \end{aligned}$$

We just saw that with the given potential function Φ the amortized cost per Increment is 2.

This means that

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i - (\Phi(D_n) - \Phi(D_0)),$$

Starting with an empty counter, $\Phi(D_0) = 0$ so

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i$$

Even if we don't start with an empty counter we can still analyze the run time by noting that

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i + \Phi(D_0) \leq 2n + \Phi(D_0)$$

So if $n > k$ (the counter size), the total cost of the sequence of operations will always be $O(n)$.

Procedure	Binary heap (worst-case)	Binomial heap (worst-case)	Fibonacci heap (amortized)
Make-Heap	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(\lg n)$	$O(\lg n)$	$\Theta(1)$
Minimum	$\Theta(1)$	$O(\lg n)$	$\Theta(1)$
Extract-Min	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$
Union	$\Theta(n)$	$O(\lg n)$	$\Theta(1)$
Decrease-Key	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
Delete	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$

We started this section with the table above. We now can understand the last column. In particular, Dijkstra's algorithm for solving the single-source shortest path problem and Prim's similar algorithm for constructing a minimum spanning-tree use

$|V|$ *Insert*s; $|V|$ *Extract-Min*s; $|E|$ *Decrease-Key*s.

The amortized time bounds given for Fibonacci heaps tells us that this sequence of operations can actually be performed in $\Theta(|V| \log |V| + |E|)$ instead of the $\Theta((|E| + |V|) \log |V|)$ that an analysis using worst-case operation time yields.

In the next part of this section we will see how to build Fibonacci heaps.