Amortized Analysis

Chuck Cusack

These notes are based on chapter 18 of [1] and lectures from CSCE423/823, Spring 2001.

When analyzing the operations for a data structure, the usual practice is to analyze each operation separately. However, when a sequence of operations is to be performed, it might be more useful to know how much time is required to execute the entire sequence, and based on this, the average cost per operation. One method of performing this analysis is to multiply the worst-case complexity of the operations performed by the number of operations in the sequence. However, we shall see that this does not always produce a tight bound.

Amortized[1] **analysis** is a technique in which the time required to perform a sequence of operations is averaged over all operations performed. This type of analysis is not probabilistic, but guarantees the worst-case performance of the *average* operation. The key word here is *average*–some operations may take more time, and some less time.

We will discuss three different methods of amortized analysis: the *aggregate*[2] *method*, the *accounting method*, and the *potential method*. We will use three data structures as examples: MULTIPOP STACK, BINARY COUNTER, and DYNAMIC TABLE.

1 Sample Data Structures and Naive Analysis

Data Structure: MULTIPOP STACK

This is a stack with the added operation MULTIPOP. Thus, the operations are:

- PUSH(x) pushes x onto the top of the stack.
- POP() pops the top of the stack and returns the popped object.
- MULTIPOP(k) pops the top k items off the stack.

If a MULTIPOP STACK has n items on it, PUSH and POP each require $O(1)$ time, and MULTIPOP requires $O(min(k,n))$ operations.

We will analyze a sequence of n PUSH, POP, and MULTIPOP operations, assuming we are starting with an empty stack. The worst-case cost of a sequence of n operations is n times the worst-case cost of any of the operations. The worst-case cost of the most expensive operation, MULTIPOP(K), is $O(n)$, so the worst-case cost of a sequence of n operations is $O(n^2)$. We will see shortly that this bound is not tight.

[1]*amortize*: to pay off by periodic payments

[2]*aggregate*: combined, total

Data Structure: BINARY COUNTER

A BINARY COUNTER is k-bit counter that starts at 0, and supports the operation INCREMENT. The counter value x is represented by a binary array $A[0, \ldots, k-1]$, where $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$. To assign values to the bits, we use the operations SET and RESET, which set the value to 1 and 0, respectively. Each of these operations has a cost of 1. INCREMENT is implemented as follows:

```
Increment()
  i=0
  while(i<k and A[i]=1)
      Reset(A[i])
      i++
  if(i<k)
      Set(A[i])
```

Figure 1 shows the INCREMENT operator on a 6-bit counter from 0 to 16. It gives the cost of each operation, and the total cost of the first n operations.

Now we will analyze a sequence of n INCREMENT operations. Since at most k SET and RESET operations are performed by INCREMENT, its worst-case cost is $O(k)$ for a k-bit counter. Therefore, the worst-case cost of a sequence of n INCREMENTS is $nO(k) = O(nk)$. As with the MULTIPOP STACK, this is not a tight bound, as we will see in the next several sections.

Figure 1: Example of INCREMENT on a 6-bit counter from 0 to 16

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

Data structures like ARRAYS and HASH TABLES have many advantages over pointer-based data structures such as LINKED-LISTS. The biggest disadvantage to these data structures, however, is the fact that the size is fixed when the list is created. A DYNAMIC TABLE is a data structure that supports the operations INSERT and DELETE, and automatically expands and contracts the table as needed. It can be used in the implementation of any table-like data structure.

Data Structure: DYNAMIC TABLE
A DYNAMIC TABLE supports the operations INSERT and DELETE. Let $size$ be the number of slots in the table, and num be the number of items in the table. We call $\alpha = num/size$ the **load factor**. In other words, the load factor is the percentage of the table that is full. When INSERT is called on a table with $\alpha = 1$, the table size is doubled, and the new item inserted. When DELETE is called, the element is removed from the table, and if $\alpha < 1/4$, the table size is halved[3]. The following is pseudo-code for the INSERT operation.

```
Insert(item X)
  if(size=0)     // If the table is empty
     T=New table(1)    // Allocate table of size 1
     size=1            // Update size
  if(num=size)   // If the table is full
     size=size*2       // Double size
     S=New table(size) // Allocate table of new size
     PlaceElements(S,T) // Copy the elements from T to S
     Deallocate T      // Free the space from the old table
     T=S               // Assign table to new table
  PlaceElement(T,X) // Place new element X into the table T
  num++             // Increment number of elements
```

Since the table may be used to implement various different data structures, like an ARRAY or HASH TABLE, for instance, the details of PLACEELEMENT and PLACEELEMENTS are not specified. We assume, however, that the complexity of both is linear in the number of elements to be placed. We also assume that DEALLOCATE and NEW take constant time. The DELETE operation is left as an exercise.

It is not too difficult to see that both INSERT and DELETE require $O(size)$ operations in the worst case. Since a sequence of n INSERT and DELETE operations may create a table with $size$ as large as n, the worst-case cost of the sequence is $nO(n) = O(n^2)$ operations. We will show that this bound is not tight.

2 The Aggregate Method

With the **aggregate method**, we show that for all n, a sequence of n operations takes worst-case time $T(n)$. Therefore, the average operation takes worst-case time $T(n)/n$. We call this average cost the **amortized cost**, and assign each operation the same amortized cost, regardless of the actual cost.

Example: MULTIPOP STACK
Given a sequence of n operations, the total number of elements that can be pushed onto the stack

[3]Why not use $\alpha < 1/2$?

is at most n. Thus, the total cost of all POP and MULTIPOP operations is no more than n, since the cost of these is the number of elements popped. The total worst-case cost of n operations is $O(n)$, and the average cost per operation is $O(n)/n = O(1)$.

Example: BINARY COUNTER
Notice that although the worst-case cost of an INCREMENT is k, usually the cost is much less. Notice that every INCREMENT requires the first bit to be flipped. However, only every other INCREMENT flips the second bit, and only every fourth INCREMENT flips the third bit. In general, bit i is only flipped once every 2^{i-1} INCREMENT. The total number of bit flips is therefore

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

so the worst-case cost of n INCREMENTs is $O(n)$, and the average cost is $O(n)/n = O(1)$.

Example: DYNAMIC TABLE
The DYNAMIC TABLE is a good example of the limitations of the aggregate method. It is not easy to see a good upper bound on the cost of a sequence of INSERT and DELETE operations. However, if we consider only the operation INSERT, we can easily do an analysis. Notice that the table expands only when num is a power of 2, in which case the cost of INSERT is $O(num)$. Otherwise, the cost is $O(1)$. If c_i is the cost of the i^{th} operation, then the cost of n INSERT operations is

$$\sum_{k=1}^{n} c_k = \sum_{1 \le k \le n : k = 2^i} k \; + \sum_{1 \le k \le n : k \ne 2^i} 1 \; < \sum_{i=0}^{\lfloor \log n \rfloor} 2^i + \sum_{i=0}^{n} 1 < \frac{2^{\lfloor \log n \rfloor + 1} - 1}{2 - 1} + n < 2n + n = 3n.$$

Since the cost of a sequence of n INSERT operations is $3n = O(n)$, the average cost of a single INSERT is $O(n)/n = O(1)$.

3 The Accounting Method

With the **accounting method**, we assign different amortized costs to each operation. When an operation's amortized cost exceeds its actual cost the difference is assigned to some object of the data structure as credit. The credit can be used later to pay for an operation whose actual cost exceeds its amortized cost. The amortized costs need to be assigned so that whenever an operation's actual cost exceeds its amortized cost, there is enough credit in the appropriate object of the data structure to pay the difference. This needs to be true no matter what n operations are performed. If we can assign amortized costs so that this is true, then we can use the total amortized cost as an upper bound on the actual cost.

Example: MULTIPOP STACK
Recall that we can only POP an item from a stack which has an item on it. In other words, we must PUSH items onto the stack before POP or MULTIPOP can incur cost. Therefore, we will assign the amortized costs as follows

Operation	Cost	Amortized Cost
PUSH	1	2
POP	1	0
MULTIPOP(K)	$k' = min(k, n)$	0

The amortized cost of PUSH is one more than its actual cost, so whenever a PUSH is executed, we associate the credit with the pushed item. When a POP is executed, its amortized cost is 1 less than its actual cost, but the item to be popped has a credit associated with it that can pay for the POP. Similarly, each of the k' items to be removed by a MULTIPOP have a credit associated with them, so the MULTIPOP can be paid for by these credits. Thus, each operation can be paid for, and the total amortized cost of any n operations is an upper bound on the actual cost. Since the most expensive operation is PUSH with amortized cost of 2, an upper bound on the worst-case cost of a sequence of n operations is $2n = O(n)$.

Example: BINARY COUNTER
For the BINARY COUNTER, we assign amortized costs as follows:

Operation	Cost	Amortized Cost
SET	1	2
RESET	1	0

When a SET operation is performed, the amortized cost exceeds the actual cost by one, so we place a credit on the bit that is set to 1. When a RESET operation is performed, the bit must be a 1, so it has a credit associated with it, which is used to pay for the operation. Thus, each operation can be paid for, and the total amortized cost is an upper bound on the actual cost. An upper bound on the actual cost is 2 times the number of SET operations performed, since the RESET operations have an amortized cost of 0. Since at most 1 SET operation is performed in each INCREMENT, a sequence of n INCREMENT operations executes at most n SET operations, so the worst-case cost of the sequence is $2n = O(n)$ operations.

Example: DYNAMIC TABLE
Again, we will consider only the INSERT operation. (We will do a complete analysis using the **potential method** in the next section.) We assign an amortized cost of 3 for INSERT. When we perform an operation, we use 1 to pay for the INSERT, place 1 on the item so we can pay to place the item in an expanded table, and place 1 on an item that has no credit on it (because it was placed from a smaller table) to pay to place it in an expanded table.

When an item is inserted into a table of size 2^i, all of the elements are copied to a table of size 2^{i+1}, and only 2 items (the item inserted and some other item) have any credit. The table will not expand again until another $2^i - 1$ more items are inserted. After each is inserted, they each have a credit, and give credit to $2^i - 1$ other items which don't have credit. When the table has size 2^{i+1}, $2 + 2(\cdot 2^i - 1) = 2^{i+1}$ items (all of them) have a credit on them, so we can pay to place them into a new table.

Thus, every INSERT can be paid for, and the amortized cost is an upper bound on the actual cost of n INSERT operations. Thus, n INSERT operations has a worst-case cost of $3n = O(n)$ operations.

4 The Potential Method

The **potential method** is similar to the **accounting method**, in that each operation is assigned an amortized cost. However, instead of assigning a credit to particular objects of the data structure, we assign it to the entire data structure, calling it the **potential** of the data structure.

Let D_0 be the initial data structure, and for each $i = 1, \dots, n$, we denote by D_i the data structure after the i^{th} operation is performed on D_{i-1}. Let c_i be the actual cost of the i^{th} operation. Let $\Phi(D_i)$ be the potential of data structure D_i. We call Φ the **potential function**. The amortized cost, $\widehat{c_i}$, of the i^{th} operation is defined by

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

In other words, the amortized cost is the actual cost plus the increase in potential due to the operation. The total amortized cost of a sequence of n operations is

$$\begin{aligned}\sum_{i=1}^{n} \widehat{c_i} &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \sum_{i=1}^{n} \Phi(D_i) - \sum_{i=0}^{n-1} \Phi(D_i) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)\end{aligned}$$

Therefore, as long as $\Phi(D_n) \geq \Phi(D_0)$, the amortized cost is an upper bound on the actual cost. Since we want this analysis to hold for any $n \geq 1$, we will define Φ so that $\Phi(D_i) \geq \Phi(D_0)$ for all $i > 0$. One way to easily do this is to define $\Phi(D_0) = 0$, and guarantee that $\Phi(D_i) \geq 0$ for all $i > 0$.

Example: MULTIPOP STACK
Define $\Phi(D_i)$ to be the number of elements on the stack. Clearly $\Phi(D_0) = 0$, and since there can't be a negative number of elements on the stack,

$$\Phi(D_i) \geq 0 = \Phi(D_0) \text{ for all } i = 1, \dots, n.$$

Therefore the total amortized cost is an upper bound on the actual cost. Now we need to compute the amortized cost of each operation. Consider the stack after $i - 1$ operations. If the number of items on the stack is s, $\Phi(D_{i-1}) = s$ by definition.
If the i^{th} operation is a PUSH, then $\Phi(D_i) = s + 1$, and the amortized cost is

$$\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (s + 1) - s = 2.$$

If the i^{th} operation is a MULTIPOP(k), and $k' = min(s, k)$, then $\Phi(D_i) = s - k'$, and the amortized cost is

$$\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k' + (s - k') - s = 0.$$

Since POP is essentially MULTIPOP(1), its amortized cost is also 0. Since the most expensive operation, PUSH, has an amortized cost of 2, an upper bound on the worst-case cost of a sequence of n stack operations is $2n = O(n)$.

Example: BINARY COUNTER
Define $\Phi(D_i)$ to be the number of 1's in the counter. Clearly $\Phi(D_i) \geq 0 = \Phi(D_0)$ for all $i = 1\ldots, n$, so the total amortized cost is an upper bound on the actual cost. Let t_i be the number of RESET operations performed by the i^{th} INCREMENT. Then the cost of INCREMENT is $t_i + 1$, and number of 1's in the counter is $t_i - 1$ less than it was. That is, $\Phi(D_i) - \Phi(D_{i-1}) = 1 - t_i$, so the amortized cost of INCREMENT is

$$\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) = (t_i + 1) + (1 - t_i) = 2.$$

Thus, a sequence of n INCREMENT operations takes $2n = O(n)$ operations in the worst-case.

Example: DYNAMIC TABLE
The potential function for a DYNAMIC TABLE is a little more complex than the previous examples. It makes sense that the potential be 0 immediately after a contraction or expansion, and increases as the load factor approaches $1/4$ or 1 to make sure that we can pay for a contraction or expansion. Thus, whenever $num = size$ ($\alpha = 1$), we need $\Phi(D_i) = size$ to pay for an expansion, and whenever $num = size/4$ ($\alpha = 1/4$), we need $\Phi(D_i) = size/4$ to pay for a contraction. We will use the following potential function.

$$\Phi(D_i) = \begin{cases} 2 \cdot num - size & \text{if } \alpha \geq 1/2 \\ size/2 - num & \text{if } \alpha < 1/2. \end{cases}$$

It is not hard to prove that this potential function satisfies to above requirements, so that the total amortized cost is an upper bound on the actual cost of a sequence of n operations. We will now compute the amortized costs.

Let $size_i$ denote the size of the table after the i^{th} operation, and num_i denote the number of items in the table after the i^{th} operation. If the i^{th} operation is an INSERT, then clearly $num_i = num_{i-1} + 1$. There are 4 cases to consider

1. If $\alpha_{i-1} = 1$, an expansion is triggered. In this case, $size_i/2 = size_{i-1} = num_i - 1$, so
$$\begin{aligned} \widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= num_i + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\ &= num_i + 2 \cdot num_i - (2 \cdot num_i - 2) - (2 \cdot num_i - 2) + (num_i - 1) \\ &= 3. \end{aligned}$$

2. If $1/2 \leq \alpha_{i-1} < \alpha_i < 1$, then $size_i = size_{i-1}$, so
$$\begin{aligned} \widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\ &= 1 + (2 \cdot num_i - size_i) - 2(num_i - 1) + size_i \\ &= 3. \end{aligned}$$

3. If $\alpha_{i-1} < \alpha_i < 1/2$, then $size_i = size_{i-1}$, so
$$\begin{aligned} \widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\ &= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i - 1)) \\ &= 0. \end{aligned}$$

4. If $\alpha_{i-1} < 1/2 \le \alpha_i$, then $size_i = size_{i-1}$, and $num_{i-1} < size_{i-1}/2$
$$\begin{aligned} \widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + (2 \cdot num_i - size_i) - (size_{i-1}/2 - num_{i-1}) \\ &= 1 + (2(\cdot num_{i-1} + 1) - size_{i-1}) - (size_{i-1}/2 - num_{i-1}) \\ &= 3 + 3 \cdot num_{i-1} - \tfrac{3}{2} size_{i-1} \\ &< 3 + \tfrac{3}{2} size_{i-1} - \tfrac{3}{2} size_{i-1} \\ &= 3. \end{aligned}$$

In any case, the amortized cost of INSERT is no larger than 3. If the i^{th} operation is a DELETE, then $num_i = num_{i-1} - 1$. There are 3 cases to consider

1. If $\alpha_{i-1} = 1/4$, then $\alpha_i < 1/4$, so a contraction is triggered. Thus, $size_i/2 = size_{i-1}/4 = num_i + 1$, and
$$\begin{aligned} \widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= (num_i + 1) + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\ &= (num_i + 1) + ((num_i + 1) - num_i) - ((2 \cdot num_i + 2) - (num_i + 1)) \\ &= 1. \end{aligned}$$

2. If $1/4 \le \alpha_i < \alpha_{i-1} < 1/2$, then $size_i = size_{i-1}$,
$$\begin{aligned} \widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\ &= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i + 1)) \\ &= 2. \end{aligned}$$

3. If $1/2 \le \alpha_{i-1}$, then $\widehat{c_i} = O(1)$. The proof of this is left as an exercise.

In any case, the amortized cost of DELETE is $O(1)$.

The amortized cost of both INSERT and DELETE is $O(1)$, so a sequence of n of these operations has a worst-case cost of $O(n)$.

References

[1] Cormen, Leiserson, and Rivest, *Introduction to Algorithms*, McGraw Hill, 1990.