

# CSC 378 Lecture 7

June 28, 2001

For another reference, see chapter 18 of the textbook (CLR).

## 1 Amortized Analysis

Often, we want to analyze the complexity of performing a sequence of operations on a particular data structure. In some cases, knowing the complexity of each operation in the sequence is important, so we can simply analyze the worst-case complexity of each operation. In other cases, only the time complexity for processing the entire sequence is important.

We can define the **worst-case sequence complexity** of a sequence of  $m$  operations as the maximum total time over all sequences of  $m$  operations (this is similar to the way that worst-case running time is defined). From this definition, it is obvious that the worst-case sequence complexity is less than or equal to  $m$  times the worst-case time complexity of a single operation in any sequence of  $m$  operations.

For example, suppose that we want to maintain a linked list of elements under the operations INSERT, DELETE, SEARCH, starting from an initially empty list. If we perform a sequence of  $m$  operations, what is the worst-case total time for all the operations? We know that the worst-case time for a single operation is  $\Theta(k)$  if the linked list contains  $k$  elements (INSERT takes time  $\Theta(1)$ . DELETE and SEARCH take time  $\Theta(k)$ ). Also, the maximum size of the linked list after  $k$  operations have been performed is  $k$ . Hence, the worst-case running time of operation number  $i$  is simply  $i - 1$  (the length of the list before operation  $i$ ), so the worst-case sequence complexity of the  $m$  operations is at most

$$\sum_{i=0}^{m-1} i = m(m-1)/2.$$

We could have been a lot more careful about analyzing the situation, since INSERT runs in time  $O(1)$  and the only way the list can grow is by inserting elements. Hence, there must either be a lot of constant-time operations or we must have a pretty short list. This kind of insight, however, would complicate the analysis and would not lead to a better asymptotic value for the worst-case sequence complexity.

The **amortized sequence complexity** of a sequence of  $m$  operations is defined as follows:

$$\text{amortized sequence complexity} = 1/m \times \text{worst-case sequence complexity} .$$

So the amortized complexity represents the average worst-case complexity of each operation. But be careful: contrary to the average-case time complexity of one operation, the amortized complexity involves no probability (the average is simply taken over the number of operations performed).

In our example above, the amortized sequence complexity is at most  $m(m-1)/2m = (m-1)/2$ . Amortized analyses make more sense than a plain worst-case time analysis in many situations, e.g.,

- A mail-order company employs a person to read customer's letters and process each order: we care about the time taken to process a day's worth of orders, for example, and not the time for each individual order.
- A symbol table in a compiler is used to keep track of information about variables in the program being compiled: we care about the time taken to process the entire program, i.e., the entire sequence of variables, and not about the time taken for each individual variable.

We will cover two basic methods for doing amortized analyses: the aggregate method and the accounting method. We've already seen an example of the aggregate method: simply compute the worst-case sequence complexity of the operations and divide by the number of operations in the sequence. We're going to look at another example to illustrate both methods.

## 1.1 MULTIPOP

Suppose we want to extend the standard Stack ADT (that has operations  $\text{PUSH}(S, x)$  and  $\text{POP}(S)$ ) with a new operation  $\text{MULTIPOP}(S, k)$  that removes the top  $k$  elements from the stack. The time complexity of each  $\text{PUSH}$  and  $\text{POP}$  operation is  $\Theta(1)$ , and the time complexity of  $\text{MULTIPOP}(S, k)$  is simply proportional to  $k$ , the number of elements removed (actually, it's proportional to  $\min(k, |S|)$ , where  $|S|$  is the number of elements in stack  $S$ ).

**The Aggregate Method:** In the aggregate method, we simply compute the worst-case sequence complexity of a sequence of operations and divide by the number of operations in the sequence.

For our  $\text{MULTIPOP}$  example, consider performing a total of  $n$  operations from among  $\text{PUSH}$ ,  $\text{POP}$ , and  $\text{MULTIPOP}$ , on a stack that is initially empty. In this case, we could at first try to say that since the stack will never contain more than  $n$  elements, the cost of each operation is  $O(n)$ , for a total of  $O(n^2)$ . This gives us an average of  $O(n)$ . In fact, we can do better if we realize that each object can be popped at most once for each time that it is pushed (including being popped by  $\text{MULTIPOP}$  operations). Since there can be at most  $n$   $\text{PUSH}$  operations, there can be at most  $n$   $\text{POP}$  operations (including counting the appropriate number  $\text{POP}$  operations for each  $\text{MULTIPOP}$ ), which means that the total time taken for the entire sequence is at most  $O(n)$ . This gives us that each operation takes on average  $O(1)$  time.

**The Accounting Method:** In the accounting method, we do the analysis as if we were an intermediate service providing access to the data structure. The *cost* to us for each operation is the operation's worst-case running time. We get to decide what we *charge* the customer for each operation. Obviously, we want to cover our costs with what we earn in charges. Unlike a store, however, we want the total charge to be as close as possible to the total cost—this will give us the best estimate of the true complexity.

Typically we will charge more than the cost for some types of operations and charge nothing for other types. When we charge more than the cost, the leftover amount can be stored with the elements in the data structure as *credit*. When we perform an “free” operation (i.e. no charge) on an element, we can use the credit stored with that element to pay for the cost of the operation.

If we assign charges and distribute credits carefully, we can ensure that each operation's cost will be paid and that the total credit stored in the data structure is never negative. This indicates

that the total amount charged for a sequence of operations is an upper bound on the total cost of the sequence, so we can use the total charge to compute an upper bound on the amortized complexity of the sequence.

For our **MULTIPOP** example, the cost of each operation (representing the time complexity of each operation) is as follows:

- $cost(PUSH(S, x)) = 1$
- $cost(POP(S)) = 1$
- $cost(MULTIPOP(S, k)) = \min(k, |S|)$

Since we know that each element can take part in at most two operations (one **PUSH** and one **POP** or **MULTIPOP**), the total "cost" for one element is 2, so we will assign charges as follows:

- $charge(PUSH) = 2$
- $charge(POP) = 0$
- $charge(MULTIPOP) = 0$

This might seem strange at first, since we are charging nothing for **POP** or **MULTIPOP**, but it works out if we distribute credits appropriately. When an element is pushed onto the stack, we charge 2: 1 is used to pay for the cost of the **PUSH**, and 1 is assigned to the element as credit. When we **POP** an element from the stack, we charge nothing: the cost of the **POP** is payed for by using the credit of 1 that was stored with the element. Similarly, for **MULTIPOP**, the cost of removing each element can be payed for by using the credit stored with each element.

Since we've shown that each operation can be payed for, and since the total credit stored in the stack is never negative (each element has a credit of 1 while it is in the stack, and there can never be a negative number of elements in the stack), we have shown that the total charge for a sequence of  $m$  operations is an upper bound on the total cost for that sequence. But the total charge for  $m$  operations is at most  $2m$ , so the total cost is  $O(m)$ . Dividing by the number of operations gives us an amortized complexity of  $O(1)$  for each operation.

## 1.2 Binary Counter

A Binary Counter is a sequence of  $k$  bits ( $k$  is fixed) on which a single operation can be performed: **INCREMENT**, which adds 1 to the integer represented in binary by the counter. The cost of a single **INCREMENT** operation is simply equal to the number of bits that need to be changed by the **INCREMENT**. For example, if  $k = 5$ ,

Initial counter:	00000	(value = 0)	
after INCREMENT:	00001	(value = 1)	cost = 1
after INCREMENT:	00010	(value = 2)	cost = 2
after INCREMENT:	00011	(value = 3)	cost = 1
after INCREMENT:	00100	(value = 4)	cost = 3
after INCREMENT:	00101	(value = 5)	cost = 1
⋮			
after INCREMENT:	11101	(value = 29)	cost = 1
after INCREMENT:	11110	(value = 30)	cost = 2
after INCREMENT:	11111	(value = 31)	cost = 1
after INCREMENT:	00000	(value = 0)	cost = 5

We can compute the amortized cost of a sequence of  $n$  **INCREMENT** operations, starting with value 0, as follows: Note that during the sequence of **INCREMENT** operations, we have the following situation (where we use the convention that bits of the counter are numbered from 0 (least significant bit) to  $k - 1$  (most significant bit)):

bit number	changes	total number of changes
0	every operation	$n$
1	every 2 operations	$\lfloor n/2 \rfloor$
2	every 4 operations	$\lfloor n/4 \rfloor$
$\vdots$		
$i$	every $2^i$ operations	$\lfloor n/2^i \rfloor$

Hence, the total number of bit-flips during the entire sequence is no more than the number of times bit  $i$  changes during the entire sequence, for bit numbers from 0 to  $\min\{k, \lfloor \log n \rfloor\}$  (the last bit that changes is bit number  $\lfloor \log n \rfloor$ , except that if  $\log n > k$ , there is no bit number  $\log n$ ). Hence, we get the following upper bound on the total number of bit-flips:

$$\sum_{i=0}^{\lfloor \log n \rfloor} \lfloor n/2^i \rfloor \leq \sum_{i=0}^{\lfloor \log n \rfloor} n/2^i \tag{1}$$

$$\leq n \sum_{i=0}^{\lfloor \log n \rfloor} 1/2^i \tag{2}$$

$$\leq n \sum_{i=0}^{\infty} 1/2^i \tag{3}$$

$$\leq 2n. \tag{4}$$

This gives us an amortized cost of  $2n/n = 2$  for each operation in the sequence.

Let's analyze the same problem using the accounting method instead of the aggregate method (which is what we did above, by finding the total cost directly). Consider what happens during one **INCREMENT** operation: a number of bits might be changed from 1 to 0 but exactly one bit will be changed from a 0 to a 1 (the rightmost bit with value 0).

For example, **INCREMENT**(00111) gives 01000, so three bits were changed from 1 to 0, but only one bit from 0 to 1. Hence, if we make sure that we have enough money stored in the counter to flip all the bits from 1 to 0, we can charge each operation only for the cost of flipping the 0 to a 1.

This is what we will do: even though the actual cost of an **INCREMENT** operation could be quite large, we charge each operation exactly 2: we use 1 to flip the 0 to a 1 and store the remaining 1 with the bit that was just changed to 1. Now, since we start the counter at 0, we can show the following **credit invariant**:

At any step during the sequence, each bit of the counter that is equal to 1 will have a credit of 1.

This can easily be proved by induction: initially, the counter is 0 and there is no credit, so the invariant is trivially true. Then, assuming that the invariant is true at a particular point, let's perform one **INCREMENT** operation: the cost of flipping bits from 1 to 0 is payed for by the credit stored with each 1, the cost of flipping a single bit from 0 to 1 is payed for with 1 from the 2 charged to the operation, and we store the remaining 1 together with the bit that was just changed to 1. None of the other bits are changed. Hence, the credit invariant is still true (every bit equal to 1 has a 1 credit).

This shows that the total charge for the sequence of operations is an upper bound on the total cost of the sequence, and since in this case the total charge is  $2n$ , we get that the amortized cost per operation is no more than  $2n/n = 2$  (same as before).