CSC 325 Adv Data Structures

Lecture 14

Amortized Time

When analyzing algorithms, we normally focus on either best or worse case. However, this isn't always indicative of what happens in practice. Best case can be seen somewhat as a lower bound and worse case as an upper bound but what about what happens on average? How about if an operation sometimes performs fast and other times performs slow. Think about dynamic array resizing. To implement a list (an abstract data type you learned about in 220), we use an array. Let's start the array off with 4 empty spots:

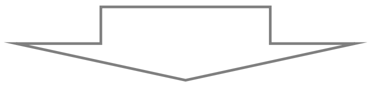
The next four insertions all happen in constant time:

`insert(5)`	5			
`insert(3)`	5	3		
`insert(7)`	5	3	7	
`insert(1)`	5	3	7	1

But for the next insertion, we must pay linear time. We pay a constant cost to create a new array double the size of the old one, but then a linear cost copying over all n values.

```
insert(9)
```

5	3	7	1



5	3	7	1	9			

So, is this insert function constant or linear? Well, in the best case it is constant, since this means no resizing has to occur. In the worse case it is linear since in this case we must make a bigger array and copy over all values. Is it fair to consider this function to be O(n) since this is the worse it can perform? Up until now we have been satisfied with this answer and while it is technically correct (since both the best and worse case run no slower than linear time), it is not indicative of what normally happens. The array resizing happens only once in a while and the far more frequent case is where we have space to put the new value. So, do all the quick insertions where there is still room for the new value make up for the infrequent nature of the expensive array doubling? In other words, do these cancel out and on average give us a better than linear time? How would we even show/prove such a thing. To do this, we must consider something called "amortized time".

Amortized time is a way of figuring out the average amount of time an operation (or even several different operations) take. In the case where the time taken doesn't change, an average cost is easier to compute. Take sequential search for example. At best it is O(1). At worst it is O(n). However this is completely dependent upon the value we are searching for and the items in the array. It is completely possible to hit the best case over and over again (we could search for the same value multiple times in a row). We could also hit the worst case over and over again (we could continually search for values that aren't in the array). If we average the best and worst case, it is simple to see that this is n / 2 (i.e. since part of the time we search 1 item and part of the time we search n items, on average we search half of this range, i.e. n / 2). Trying to apply this same logic to the dynamic array insertions doesn't work. That's because the operation naturally changes as you use it. Regardless of which value you insert, you will continue to pay constant time unless an array doubling is required, in which case you will pay linear time. It is not possible to continue to pay constant time indefinitely (as was the case with sequential search) nor is it possible to pay linear time indefinitely. So the average time is not simply n / 2 in this case. There is a pattern here that we can examine to get a better average time. Let's look at a table to see this pattern better. Assuming the dynamic array starts out with a size of 1 initially, here is what we will pay for successive insertions:

Cost	1	**n**	**n**	1	**n**	1	1	1	**n**	1	1	1	1	1	1	1	**n**	1
n	1	**2**	**3**	4	**5**	6	7	8	**9**	10	11	12	13	14	15	16	**17**	18
Capacity	1	**2**	**4**	4	**8**	8	8	8	**16**	16	16	16	16	16	16	16	**32**	32

This assumes we double on the next insertion after filling up the array. Perhaps it is easier to see the pattern if we double immediately once the array is full:

Cost	**n**	**n**	1	**n**	1	1	1	**n**	1	1	1	1	1	1	1	**n**	1	1
n	**1**	**2**	3	**4**	5	6	7	**8**	9	10	11	12	13	14	15	**16**	17	18
Capacity	**1**	**2**	4	**4**	8	8	8	**8**	16	16	16	16	16	16	16	**16**	32	32

Aggregate method

One straightforward way to see the amortized cost of this operation is to look at its average cost. If we insert some numbers, examine the cost for each insertion, then divide that by the number of insertions, we can see the average cost per insertion. This is known as the aggregate method.

Let's pick a number of values to insert. Let's say we insert (n + 1) values. We include the plus one such that at least one of those insertions would fill up the size n array and force an array doubling to occur. Given an empty n sized array, if we insert (n + 1) values, the cost would be the following:

Cost	1	1	1	1		1	n	1
Value	1st value	2nd value	3rd value	4th value	…	(n – 1)th value	nth value	(n + 1)th value

Basically, we pay constant cost for all values inserted except for the one value that causes an array expansion, which for that one insert will cost linear time. So the total time will be `n * O(1) + O(n)`. The (n * O(1)) part shows the n times we paid O(1) to insert a value into an empty slot and the (O(n)) part shows the one time we paid O(n) to expand the array's size and copy all values over. For the aggregate method, we simply take this sum and divide it by the number of times this operation was invoked, i.e.:

$$\frac{n * O(1) + O(n)}{n+1} = \frac{O(n) + O(n)}{n+1} = \frac{O(n)}{O(n)} = O(1)$$

In other words, the total time for (n + 1) insertions is (n * O(1) + O(n)). If we divide these we get a constant amount of time (on average) per insertion. This is similar to saying that if doing 10 things takes you 60 seconds, it took you, on average, 6 seconds to do each thing. If it takes n time to do n things, then on average, each thing took constant time.

Accounting method

Let’s use actual money to see if the cheap operations can “pay” for the expensive operations. Let’s say that every time we insert a value into our array, we must pay 1 coin. Every time we resize, we must pay 1 coin for each value we move over into the new array. However, to make this work, we are going to overpay to insert a value. We will pay 3 coins instead of 1, giving us a credit of 2 coins (that we will put into our “bank”). We will use these overpaid coins to pay for the resizing. This means that the credit in the bank (leftover money) that was paid for inserting values will be used to fully pay for the resizing, thereby making resizing free. Let’s see with this new coin system if the money paid for the insertions can fully pay for the number of values we must move every time we resize.

Cost	**n**	**n**	1	**n**	1	1	1	**n**	1	1	1	1	1	1	1	**n**	1	1
n	**1**	**2**	3	**4**	5	6	7	**8**	9	10	11	12	13	14	15	**16**	17	18
Capacity	**1**	**2**	4	**4**	8	8	8	**8**	16	16	16	16	16	16	16	**16**	32	32
Coins in bank	**1**	**1**	3	**1**	3	5	7	**1**	3	5	7	9	11	13	15	**1**	3	5

Let’s break down what is happening in more granular detail:

Action	**Coin Cost**	**Coins Paid**	**Coins in Bank**	
Inserting a value into empty slot	1	3	2	Inserting first value into array whose capacity is 1
Doubling the array	free			
Moving over single value	1	1	1	
Inserting a value into empty slot	1	3	3	Inserting second value into array whose capacity is 2
Doubling the array	free			
Moving over two values	2	2	1	
Inserting a value into empty slot	1	3	3	Inserting third and fourth value into array whose capacity is 4
Inserting a value into empty slot	1	3	5	
Doubling the array	free			
Moving over four values	4	4	1	
Inserting a value into empty slot	1	3	3	Inserting fifth, sixth, seventh and eighth value into array whose capacity is 8
Inserting a value into empty slot	1	3	5	
Inserting a value into empty slot	1	3	7	
Inserting a value into empty slot	1	3	9	
Doubling the array	free			
Moving over eight values	8	8	1	

It seems that we always have enough coins to pay for the values to move over to the new array. In fact, by the time we need to resize to 16, we should have built up 17 coins in the bank. When we resize to 32, we should have built up 33 coins. When we resize to k, we should have built up (k + 1) coins. This shows that the constant operations (inserting a value into an empty slot) actually happen frequently enough that they pay for the linear operations (moving over all values into the new array). Since we overpay a constant number of coins for every value inserted and those overpaid coins cover the expensive operations of moving over values, we can say that there is enough cheap operations (inserting a value takes constant time) to make up for the expensive operations (moving array items) and so overall the insert method takes constant time. We basically have enough constant operations by the time we have to resize that resizing becomes “free”, paid for by our already existing bank balance. So using amortized cost analysis (and the "accounting method"), we can say that insertion into a dynamic array takes amortized constant time, i.e. O(1).

Summary and Use Case

You might think this is cheating. However, this isn't a trick to make anything seem like it takes constant time. Imagine if we allowed the user to resize the array anytime they wished, even when it is not full. In that case, it would be possible to continually resize the array without having enough values inserted to justify the cost. In this case amortized analysis would NOT consider this to be a constant time operation, since we can't be sure we would have saved up enough coins to pay for the resizing in all cases. This would then be classified as a linear operation, since we would have to pay n coins to ensure the moving of array elements is paid for. So for this analysis to work, we have to make sure that over paying by a <u>constant</u> amount is enough to completely cover the cost of periodic expensive operations.

One area where this analysis really helps is with dictionaries. Dictionaries are implemented using hash tables, which are really just arrays. In 220 you learned that hash tables can insert, search and delete in constant time. Yet some of the operations didn't seem like they took constant time, at least not in all cases.

When using chaining as your collision resolution, the load factor α was the average length of a linked list (tied to a bucket). As this load factors grows, we can make a decision to double the table, thereby shrinking the load factor by giving us more empty buckets (i.e. the number of buckets, n, is now larger and when m, the number of values stored, is now divided into n, we get a smaller number, i.e. a smaller load factor). Since the load factor can be bound by a constant, for example we can say load factor should never be larger than 4, this means that our time complexity is really O(α) which is O(4) which is O(1). But what about that table doubling operation? Well this problem essentially reduces down to the dynamic array problem we have talked about during this entire lesson. We already proved that amortized time is constant for that operation, and so as long as the load factor is upper bounded by a constant, the entire operation is seen as O(1). A similar (but more involved) argument can be made if we are using open addressing for collision resolution. This one gets tricky with regards to probing, however amortized analysis still tells us that the insertion operation is constant.

References:

- https://www.youtube.com/watch?v=3MpzavN3Mco (Erik Demaine)
- https://www.youtube.com/watch?v=T7W5E-5mljc (NERDfirst)
- https://www.youtube.com/watch?v=VgLh4_4Bkhc (Geometry Lab)