Dynamic Array

This chapter discusses the time complexity of insert operations on a dynamic array.

What is it?

A dynamic array resizes itself automatically when it runs out of capacity and new elements need to be added to it. In Java, the ***ArrayList*** class is one such example. The official Java documentation states:

```
/**
* As elements are added to an ArrayList, its capacity grows automatically.
* The details of the growth policy are not specified beyond the fact that
* adding an element has constant amortized time cost.
*
*/
```

Note that a dynamic array only resizes itself to grow bigger, but not smaller. The claim is that the amortized cost of inserting n elements in a dynamic array is O(1) or constant time.

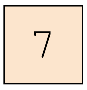
Implementation

We know that an array is a contiguous block of memory and can't be resized. Internally, a dynamic array implementation simply discards the previous allocation of memory and reallocates a bigger chunk. When the existing members are copied over to the new array and now new ones can also be added because of additional capacity.

One common strategy is to double the array size whenever it reaches full capacity. Below, we'll work out the complexity for this strategy.

First Element

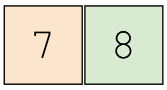
We'll trace how an array expands, starting with a size of 1 as new elements get added. The initial state of the array is shown below with one element.



Second Element Added

Now we add another element. Since the capacity is 1, we need to double it to 2 elements. We copy the previous element and insert the new one.

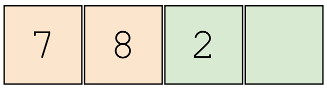
Cost = 1 copy + 1 new insert



Third Element Added

Since we don't have any more capacity, we'll need to double our array size again. Note that, at this point, we'll need to copy the previous two elements and add the new element.

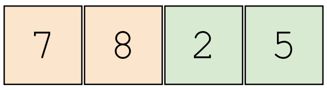
Cost = 2 copies + 1 new insert



Fourth Element Added

When we want to add the fourth element, there's one empty slot in our array where we can insert it without having to resize the array.

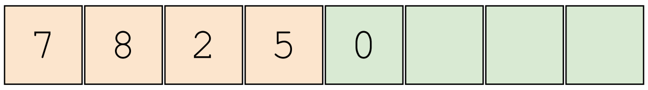
Cost = 1 new insert



Fifth Element Added

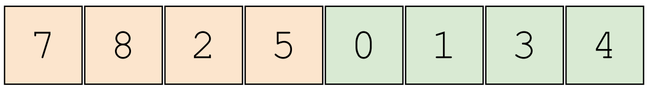
When we want to add the fifth element, there's no space in the array, so we need to double the array size from four to eight. We'll need to copy the four elements from the old array to the new array and insert the fifth element.

Cost: 4 copies + 1 new insert



Sixth, Seventh & Eight Element Added

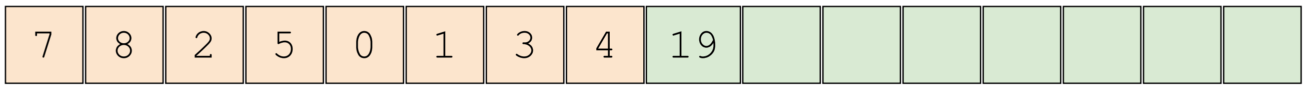
We have space for 3 more elements, so we can add the sixth, seventh and eighth element.
Cost for adding sixth, seventh and eighth element is just 3 new inserts.



Ninth Element Added

Adding the ninth element will cause us to double our array again, making the array the size of sixteen elements. Note that this time we'll need to copy the 8 elements from the older array to the new array, and also insert the ninth element.

Cost: 8 copies and 1 new insert



Adding Up the Costs

Let's take stock of our costs

Element #	Cost
1	1 insert
2	1 insert + 1 copy = 2
3	1 insert + 2 copies = 3
4	1 insert = 1
5	1 insert + 4 copies = 5
6	1 insert = 1

7	1 insert = 1
8	1 insert = 1
9	1 insert + 8 copies = 9 operations

You can probably see a pattern here. The highlighted rows incur additional copy operations. Coincidentally, they are also powers of 2 plus one.

$$2^n + 1$$

$$2^0 + 1 = 2,\ when\ n = 0$$

$$2^1 + 1 = 3,\ when\ n = 1$$

$$2^2 + 1 = 5,\ when\ n = 2$$

$$2^2 + 1 = 9,\ when\ n = 3$$

So now it becomes very easy for us to reason about the number of total operations that would take place when we add the ***(2^n+1)-th*** element to our dynamic array. It'll cause our array to double in size to ***(2^{n+1})*** slots.

For **2^n + 1** elements, the total operations would be

Total Operations = **Total Inserts** + **Total Copies**

Total Inserts

We know that we inserted **2^n + 1** elements.

Total Copies

In our example scenario:

- 1 copy when we doubled size from 1 to 2
- 2 copies when we doubled size from 2 to 4
- 4 copies when we doubled size from 4 to 8
- 8 copies when we doubled size from 8 to 16

Using the above information we can generalize that we will make that $\mathbf{2^n}$ copies whenever we increase the size of the array from $\mathbf{2^n}$ to $\mathbf{2^{n+1}}$. Now don't forget that our array was initially of size 0 and went through several doublings in size, so we need to sum up all the copy operations. When we added the **2^n + 1th** element, we would have performed the following copy operations starting all the way from the first doubling of the array.

$$1 + 2 + 4 ... 2^n$$

Total Cost

The total cost is just the total number of operations that we did to get the array to size $\mathbf{2^{n+1}}$ when we added the **(2^n+1)-th** element.

$$Total\ Cost = Total\ Inserts + Total\ Copies$$

$$Total\ Cost = (2^n + 1) + (1 + 2 + 4 ... 2^n)$$

Rearranging the terms, we can write

$$Total\ Cost = (1) + (1 + 2 + 4 ... 2^{n-1} + 2^n)$$

The arithematic series **(1 + 2 + 4 ... 2^{n-1} + 2^n)** is known to sum up to **2^{n+1} - 1**; for example:

$$1 + 2 = 3\ is\ equivalent\ to\ 2^{1+1} - 1 = 3$$

$$1 + 2 + 4 = 7\ is\ equivalent\ to\ 2^{2+1} - 1 = 7$$

$$1 + 2 + 4 + 8 = 15\ is\ equivalent\ to\ 2^{3+1} - 1 = 15$$

We can rewrite the total cost as:

$$Total\ Cost = 2^n + 1 + 2^{n+1} - 1$$

$$Total\ Cost = 2n + 2n + 1$$

$$Total\ Cost = 2^n + (2^n * 2)$$

$$Total\ Cost = 2^n(1 + 2)$$

$$Total\ Cost = 2^n * 3$$

Amortized Cost

To calculate the average or the amortized cost, all we need to do is to divide the total cost by the total number of elements added, similar to how we'll calculate average.

$$Amortized\ Cost = \frac{Total\ Cost}{Total\ Elements}$$

$$Amortized\ Cost = \frac{2^n * 3}{2^n + 1}$$

The equation 2^n / $2^n + 1$ would be roughly equal to 1 as n tends to infinity leaving the amoritzed cost as:

$$Amortized\ Cost = [\frac{2^n}{2^n + 1}] * 3$$

$$Amortized\ Cost = [1] * 3$$

$$Amortized\ Cost = 3$$

Hence we can claim that the amortized cost of insert operations is constant for a dynamic array.