

## Amortize

An object `al` of class `java.util.ArrayList` maintains a list of values in an array `b` (say) and the number `size` of values in the list. It appears that `add` takes constant time: the values are maintained in `b[0..size-1]`, so just store the new value in `b[size]` and increment `size`.

Suppose the list has 15 values and the array has a *capacity* of 20 (`b.length` is 20). Therefore, 5 more values can be added to the list, using, say, method `al.add(...)`.





What happens when the array is filled to capacity (the list has 20 values) and a new value is added? A new array of twice the size is created, the values in the list are copied to the new array, and the new array is used from then on.

If that's the case, how can we say that method `add(...)` takes constant time? Well, here is what the API documentation for class `ArrayList`, version 14, says:

The `add` operation runs in *amortized constant time*, that is, adding  $n$  elements requires  $O(n)$  time.

### Explaining amortized time

A year ago, we bought a SodaStream fizz maker, let's say for \$100.00. We bought it to save money. We don't have to buy plastic bottles of fizzy water anymore, because this machine adds fizz to a glass of water.

	1 glass: \$100.00
	2 glasses: \$50.00 each
	100 glasses: \$1.00 each
	1000 glasses: 10¢ each

Think about it this way. After making one glass of fizzy water with the machine, we said that the glass cost us \$100.00. After making two glasses, we said each glass cost us \$50.00. After making 100 glasses, each glass cost us \$1.00; after 1,000 glasses, each glass cost us 10¢.



We *amortized* the initial cost of the machine over the glasses of fizzy water that we made with it.

### Amortizing the time in `ArrayList`

The implementation of `ArrayList` works like this: There is a backing array `b`, whose length is called the *capacity* of the array. The `ArrayList` has *size* elements, which are in `b[0..size-1]`.

	0	<i>size</i>	<i>capacity</i>
<code>b</code>	elements	?	

When *size* and *capacity* are the same, `b` has no room for more elements. If an element is to be added, a new array `b'` of twice the length (say) is created, the values in `b` are copied into the beginning of `b'`, and `b'` is assigned to `b` so that `b` now has twice the size.

Copying each element of array `b` into `b'` takes time, and it's important to know how much time all this copying takes. Below, we prove the following theorem.

**Theorem 1.** Suppose `b` has *capacity* 1 and *size* 1—it's a list of size 1. Suppose  $2^{k-1}$  additional elements are added to the `ArrayList` (for some  $k$ ), bringing the total to  $n = 2^k$ . The total number of individual elements copied is  $n-1$ , for any  $k \geq 0$ .

Here's the key point: Adding the additional  $n-1$  elements requires making a total of  $n-1$  copies, so the number of copies *per element added* is  $(n-1) / (n-1) = 1$ . Therefore, we can *amortize* the cost of all this copying by saying that each additional insertion cost not just the time for insertion but also the time for copying one element. Since both of these costs are constant, the amortized time for inserting an element is  $O(1)$ .

### Proof of theorem 1

In a math course, theorem 1 would be proved by mathematical induction on  $k$ . Instead, we give a more informal explanation of why the theorem is true.

The table on the next page shows the steps involved in adding elements to `b` until its size is  $n = 2^k$  elements.

## Amortize

size and capacity	Add this many elements	Since $b$ is full, first double $b$ to this size	Number of elements copied in doubling $b$
$2^0 = 1$	$2^0 = 1$	$2^1 = 2$	$2^0 = 1$
$2^1 = 2$	$2^1 = 2$	$2^2 = 4$	$2^1 = 2$
$2^2 = 4$	$2^2 = 4$	$2^3 = 8$	$2^2 = 4$
...	...	...	...
$2^{(k-1)}$	$2^{(k-1)}$	$2^k = n$	$2^{(k-1)}$

Look at the first row. The first column gives the initial size and capacity of  $b$ . The first step is to add 1 element to array  $b$  (see column 2), but since  $b$  has size 1, this causes its length to be doubled to 2 (see column 3). Column 4 shows the number of elements copied while doubling the length of  $b$ : 1.

Now look at row 2. The previous row filled array  $b$  to capacity 2. Adding 2 elements first doubles  $b$ 's length to 4 and then fills it to capacity. In doubling its size,  $2^1 = 2$  elements are copied. In turn, each row doubles the array length and fills it to capacity. The last row shows the last step in filling the array with  $n = 2^k$  elements.

Column 4 gives the number of elements copied at each step, so in total, the number of elements copied is  $2^0 + 2^1 + \dots + 2^{(k-1)} = 2^k - 1 = n - 1$ . QED (*quod erat demonstrandum*, or Quit End Done.)

### What about increasing the array size by 1?

We leave it to you to prove the following: Suppose that, when  $b$ 's length has to be increased, it is increased by only 1. To add  $n$  elements to the `ArrayList` requires  $n(n-1)/2$  elements to be copied. Amortizing this over the insertion of  $n$  elements into the list gives amortized time  $O(n)$ .

You can do this by building a table as done above. To help you out, suppose the size and capacity of  $b$  is 8, and one more element is added. What is the new size and capacity of  $b$ , and how many elements were copied?