Amortize

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

Suppose the list has 15 values and the array has a *capacity* of 20 (`c.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 list 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 8, 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.

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¢.

Say it cost $100.00

image taken from sodastreamusa.com

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

Amortizing in ArrayList

Consider adding values to empty `ArrayList al` with initial capacity 20. Adding each of 20 values takes constant time. Adding a twentieth value causes a new array to be created of size 40 and the first 20 values to be copied into the new array.

We *amortize* each element copied over the add operation that caused it to be added. Thus, each of the first 20 adds cost (1) the time to add plus (2) the time to copy. Since adding and copying are constant time operations, that's still constant time.

But what about having to copy lots of times?

The diagram to the right shows an `ArrayList` filled to capacity, let's say with capacity 20. Each element cost 1 add.

1 add

The second diagram shows that the array size has been doubled and 20 more elements added to it. 20 elements required an add and a copy; 20 required only add.

1 add, 1 copy	1 add

The third diagram shows that the array has been doubled again and 40 more elements added to it. Now, a total of 60 copy operations have been done (20 elements were copied twice), but 80 elements have been added.

1 add, 2 copy	1 add, 1 copy	1 add

If you keep going this way, you will see that no matter how many times the array has to be doubled and then filled to capacity, the number of copies made is still less than the number of elements added. Therefore, it's still constant time per element added. This would not be the case if, when changing the size of the array, we added only one element instead of doubling the size.