# **ArrayList Operations**

## **ArrayList add(E e)**

**Overview**

* Once you’ve constructed an ArrayList, you can add values to it by calling its add(E e) method.
* When you ask an ArrayList to add a new value to the list, **it appends the new value to the end of the list**.
* Unlike with a simple array, an ArrayList can be thought of as having a dynamic size that automatically grows and shrinks to fit the elements you put inside it.
* So, you can think of the add method as performing two major tasks:

1. **Expanding the list’s size by one to accommodate the new element**

and

1. **Placing the new element at the end of the list**

**For Example**

// add values to a list

ArrayList<String> bands = new ArrayList<>();

bands.add("Tool");

bands.add("Phish");

bands.add("Pink Floyd");

A picture containing table

Description automatically generated

**When the ArrayList is Full**

* The ArrayList class manages an internal array of object references.
* When we want to add an object to the ArrayList, we **expand the list’s size by one** to accommodate the new element.
* Eventually, that internal arraywe are maintainingwill **run out of space**.
* What happens then?
* This is where array lists work their magic:

If you call add and the internal array is full, the ArrayList will automatically create a bigger array and copy all the objects from the smaller to the bigger array.

* Therefore, you will technically never run out of space in an ArrayList since its capacity will continue grow as you insert elements.

**Amortized Runtime**

* An ArrayList is implemented with an array.
* When the array hits capacity, the ArrayList class will create a new array with double the capacity and copy all the elements over to the new array.
* How do you describe the runtime of insertion? This is a tricky question.
* The array could be full.
  + If the array contains N elements, then inserting a new element will take O(N) time.
  + You will have to create a new array of size 2N and then copy N elements over.
  + This insertion will take O(N) time.
* However, we also know that this doesn't happen very often.
* The vast majority of the time insertion will be in O(l) time.
* We need a concept that takes both into account.
* This is what **amortized time** does.
* It allows us to describe that, yes, this worst case happens **every once in a while.**
* But once it happens, it won't happen again for so long that the cost is "amortized:'
* In this case, what is the amortized time?
* As we insert elements, we double the capacity when the size of the array is a power of 2.
* So after X elements, we double the capacity at array sizes 1, 2, 4, 8, 16, ..., X.
* That doubling takes, respectively, 1, 2, 4, 8, 16, 32, 64, ..., X copies.
* What is the sum of 1 + 2 + 4 + 8 + 16 + ... + X?
  + If you read this sum left to right, it starts with 1 and doubles until it gets to X.
  + If you read right to left, it starts with X and halves until it gets to 1.
* What is the sum of X + X/2 + X/4 + X/8 + ... + 1 ? This is roughly **2X**.
* Therefore, X insertions take O(2X) time. The amortized time for each insertion is O(1)

## **ArrayList add(int index, E e)**

* The ArrayList class also provides an overloaded version of the add method for adding a value at a particular index in the list.
* This version of add takes two parameters:

1. an index

and

1. a value to insert

* ArrayLists **preserve the order of the other list elements**.
* Any elements **to the right** of the given index are shifted **one to right** to make room for the new value.
* They use zero-based indexing, just as arrays and strings do, so if we insert a new object at index ***i***, then

The value that was previously at index *i* is now at index ***i+1*** after the new element ***e*** is inserted at index ***i***.

If you are adding an element to the end of the list, ***i*** would be equal to the size of the array.

If you try and insert past the size, then you will get an IndexOutOfBoundsException.

**For Example:**

Given the preceding list, consider the effect of inserting a value at index 1:

// the arraylist bands currently holds [Tool, Phish, Pink Floyd]

bands.add(1, "U2");

A picture containing timeline

Description automatically generated

* The call on add(int index, E e) instructs the list to insert the new element at index 1.
* Therefore, the old value at index 1 and everything that follows it gets shifted to the right.
* You can think of this method as performing three operations:
  1. expanding the list’s size by 1,
  2. shifting elements right to make room for a new element,

and

* 1. inserting the new element

## **ArrayList remove(int index)**

* The ArrayList also has a remove method that accepts an integer index as a parameter and removes the value at that index.
* ArrayLists **preserve the order of the other list elements**.
* The method preserves the order of the list by shifting values any values **to the right** of the given index are to the **left** to fill in the empty gap.

**For Example**

Consider what happens to the previous list if we remove the value at position 0 and then remove the value at position 1:

bands.remove(0);

bands.remove(1);

A picture containing application

Description automatically generated

* This result is a little surprising.
* We asked the list to remove the value at position 0 and then to remove the value at position 1.
* You might imagine that this would get rid of the strings "Tool" and "U2", since they were at positions 0 and 1, respectively, before this code was executed.
* However, an ArrayList is a dynamic structure whose values move around and shift into new positions in response to your commands.
* Therefore, **timing** **matters when removing or adding objects in an ArrayList**

## **ArrayList size()**

* You can pass an initial capacity to the ArrayList constructor:

ArrayList<Employee> staff = new ArrayList<>(100);

* Doing so allocates an internal array (or linked list) of 100 objects.
* Then, the first 100 calls to add will not involve any costly reallocation.
* If you want to **find the number of elements** in an ArrayList, you can call its **size** method
* The size() method returns the actual number of elements in the array list.

**For Example**

staff.size()

returns the current number of elements in the staff array list.

This is the equivalent of

a.length

if staff has an internal array called a.

## **ArrayList get(int index) and set(int index)**

* Unfortunately, the convenience we get from automatic growth convenience makes indexing the array more difficult.
* The reason is that the ArrayList class is not a part of the Java programming language; it is just a utility class programmed by someone and supplied in the standard library.
* Instead of the pleasant [] syntax to access or change the element of an array, you use the get() and set() methods.
* For example, to set the ith element, you use

staff.set(i, harry);

* This is equivalent to

a[i] = harry;

* for an array a. (As with arrays, the index values are zero based.)
* To get an array list element, use

Employee e = staff.get(i);

* This is equivalent to

Employee e = a[i];

**CAUTION:**

* Do not call list.set(i, x) until the *size* of the array list is larger than i.
* For example, the following code is wrong:

ArrayList<Employee> list = new ArrayList<>(100); // capacity 100, size 0

list.set(0, x); // no element 0 yet

* Use the add method instead of set to fill up an array, and use set only to replace a previously added element.