# Array of primitive values

When an array of primitive types is allocated, space is allocated for all of it's elements contiguously, as shown here.

You can see from this slide, that we have an array of **seven integers**.

The index position is in the left column, and that's the number we use, to access a specific array value.

So the first element, when we use index position 0, this will retrieve the value 34.

When we use index position 1, this gets the value of 18, and so on.

The addresses we show here are memory addresses, represented by these numbers.

| Index | Value | Address |
|---|---|---|
| 0 | 34 | 100 |
| 1 | 18 | 104 |
| 2 | 91 | 108 |
| 3 | 57 | 112 |
| 4 | 453 | 116 |
| 5 | 68 | 120 |
| 6 | 6 | 124 |

# Array of primitive values

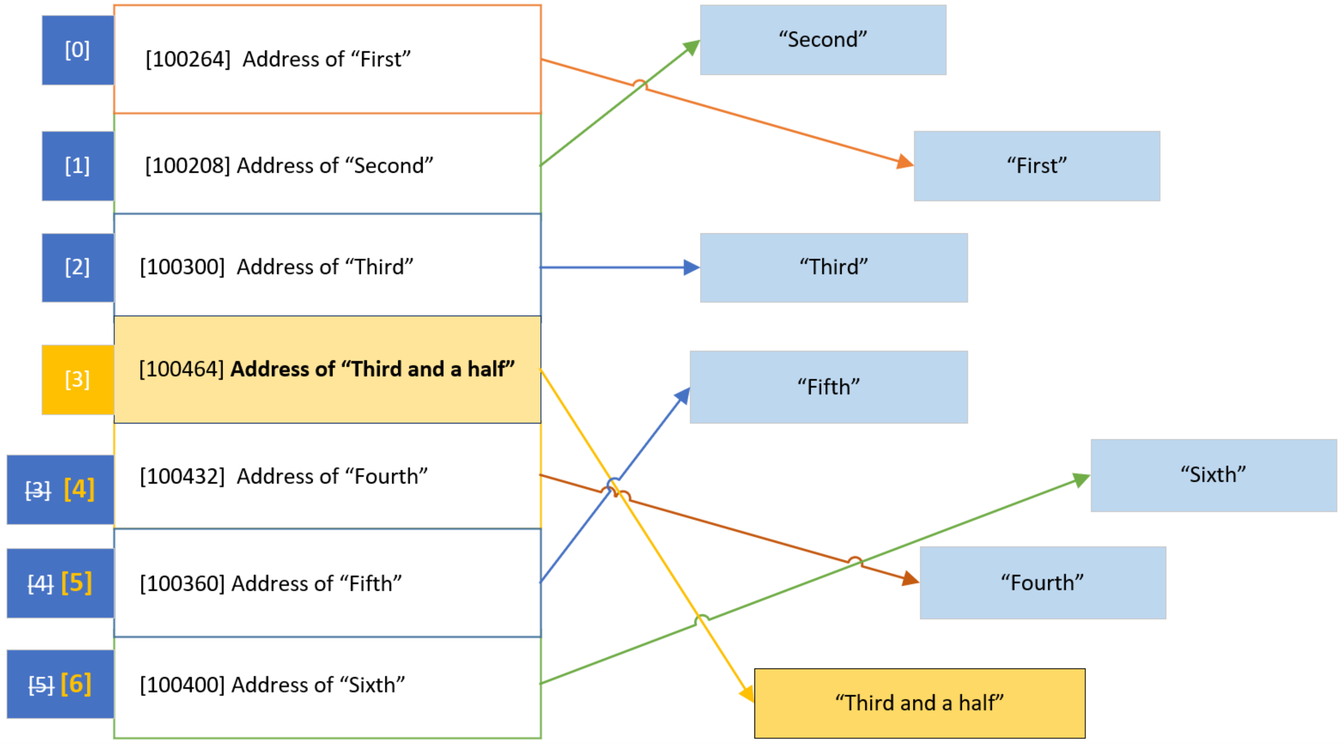
If 100 is the address of an integer, and we know an integer is 4 bytes, then the address of the next integer, if it's contiguous would be 104, as we show here, for the second element.

Java can use simple math, using the index, and the address of the initial element in the array, to get the address, and retrieve the value of the element.

| Index | Value | Address |
|---|---|---|
| 0 | 34 | 100 |
| 1 | 18 | 104 |
| 2 | 91 | 108 |
| 3 | 57 | 112 |
| 4 | 453 | 116 |
| 5 | 68 | 120 |
| 6 | 6 | 124 |

# Arrays and ArrayLists of reference types

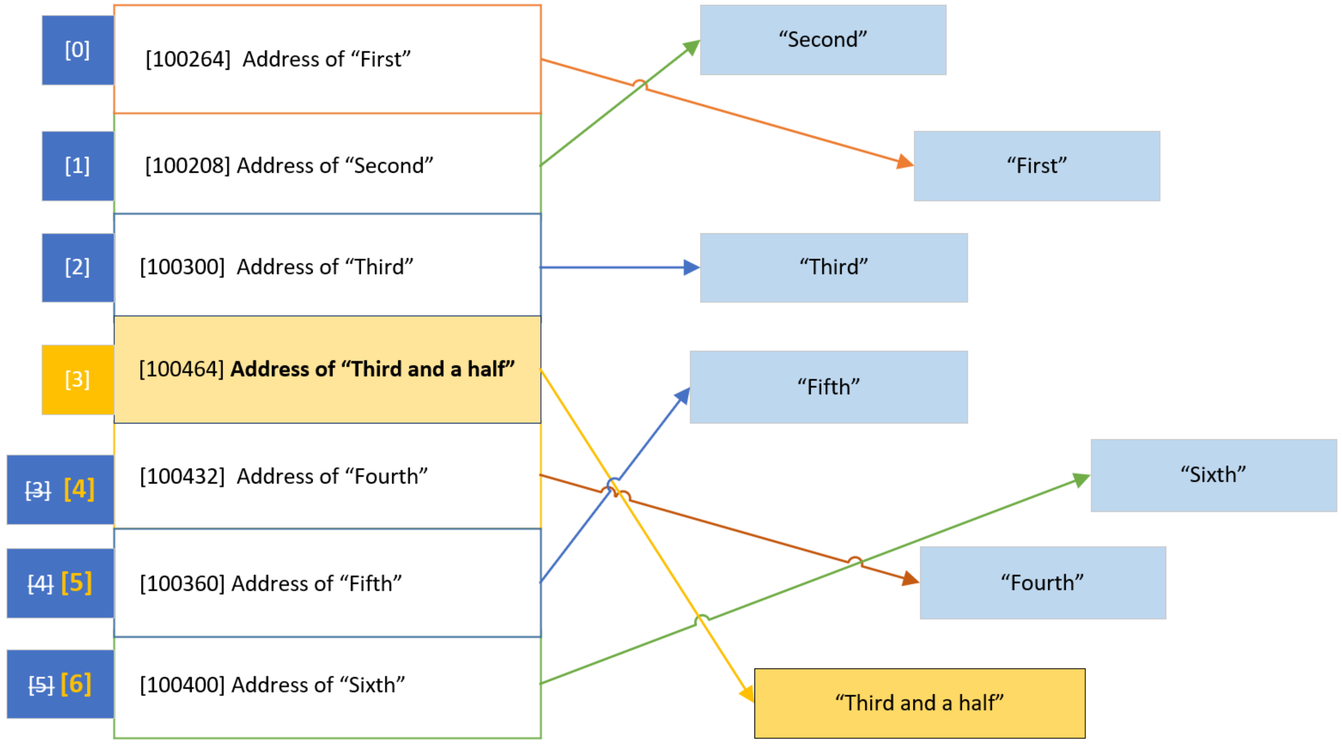
For reference types (meaning anything that's not a primitive type), like a String, or any other object, the array elements aren't the values, but the addresses of the referenced object or String.



# Arrays and ArrayLists of reference types

There's a level of indirection as I show on this slide.

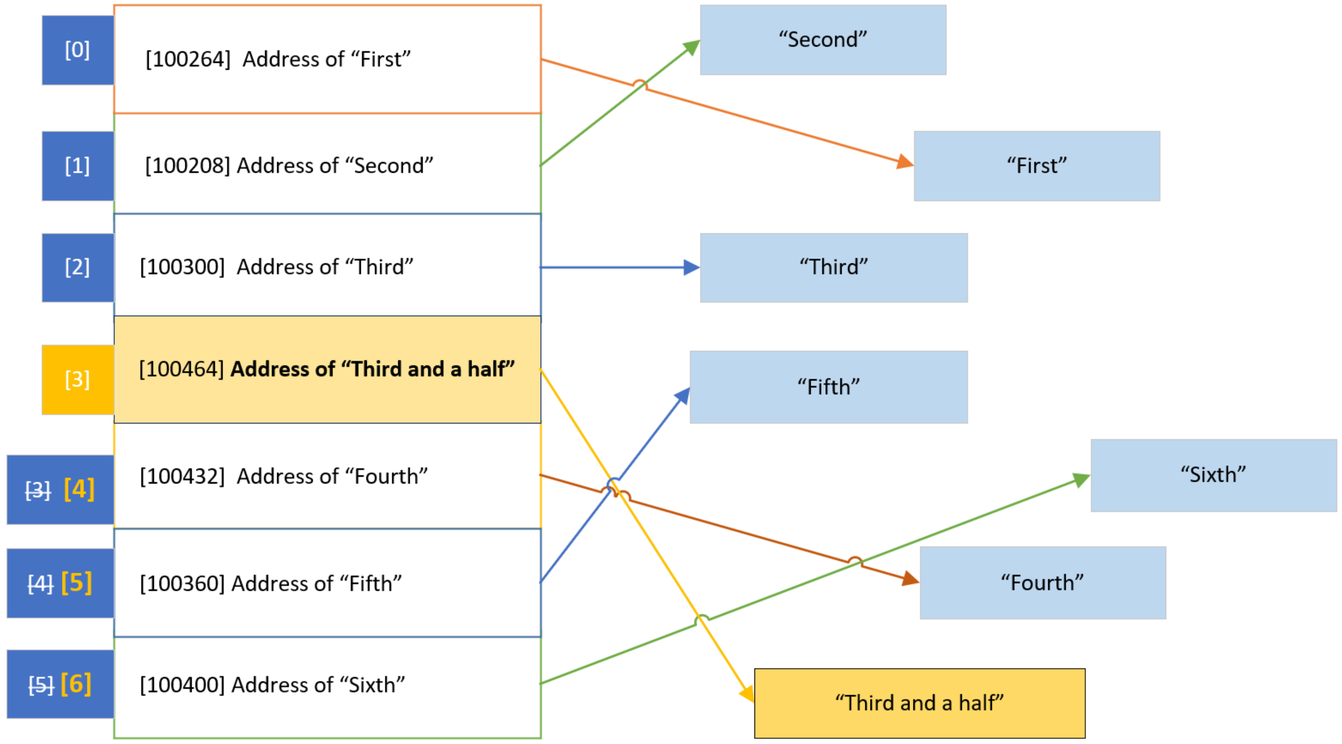
We've learned that ArrayLists are really implemented with arrays, under the covers.



# Arrays and ArrayLists of reference types

This means our objects aren't stored contiguously in memory, but their addresses are, in the array behind the ArrayList.

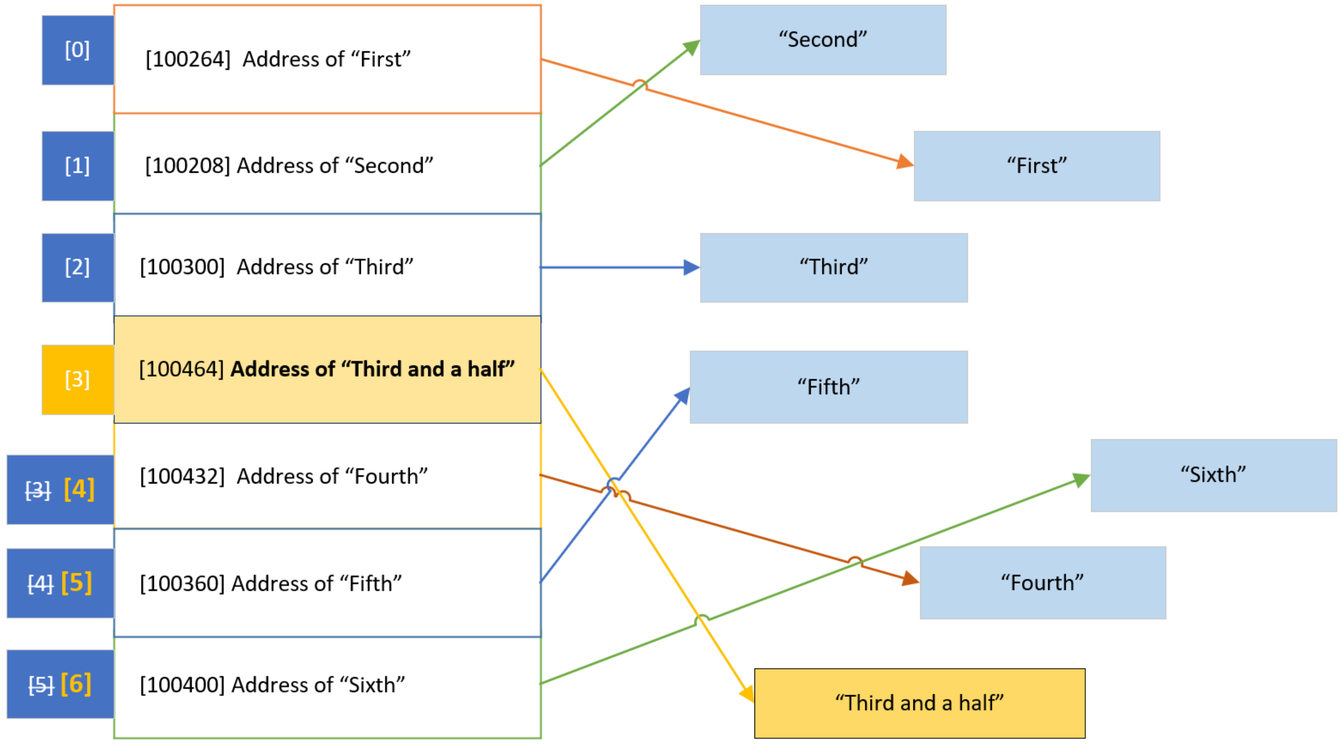
And again, the addresses can be easily retrieved with a bit of math, if we know the index of the element.



# Arrays and ArrayLists of reference types

This is a **cheap** lookup, and doesn't change, no matter what size the ArrayList is.

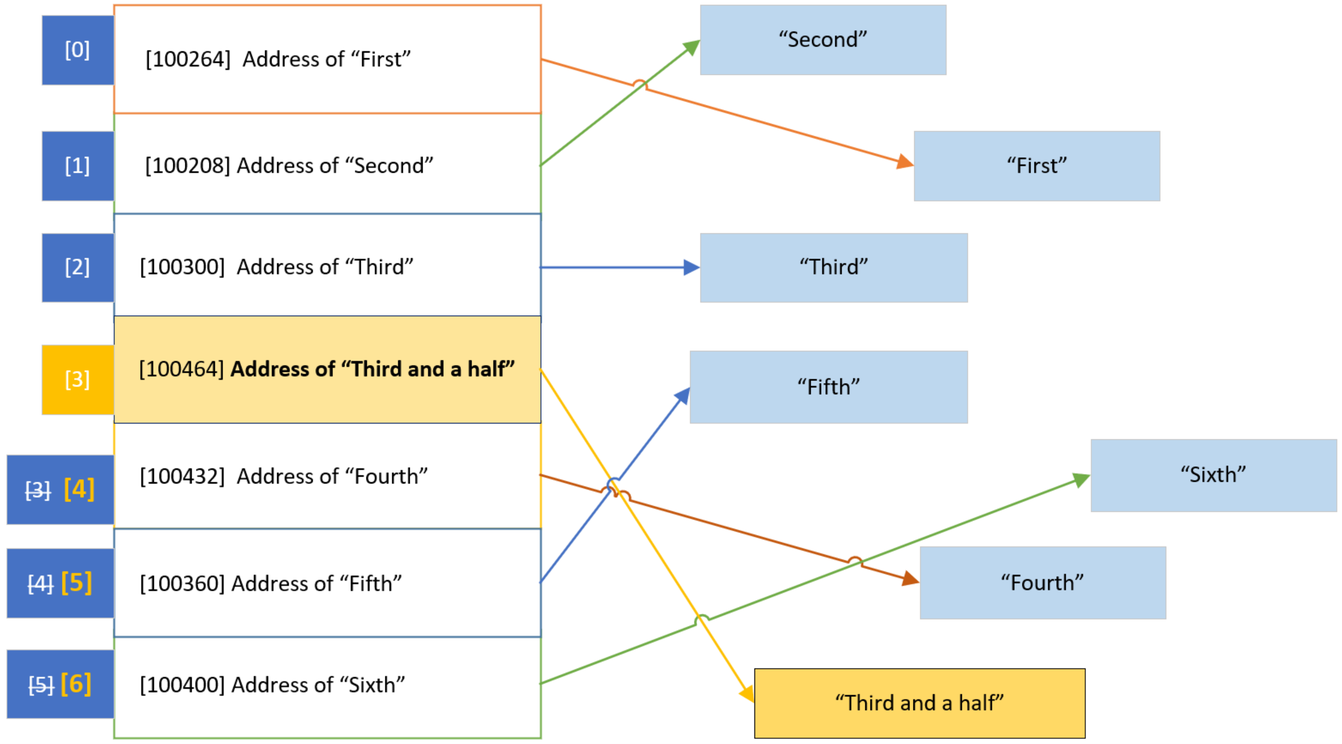
But to remove an element, the referenced addresses have to be re-indexed, or shifted, to remove an empty space.



# Arrays and ArrayLists of reference types

And when adding an element, the array that backs the ArrayList might be too small, and might need to be reallocated.

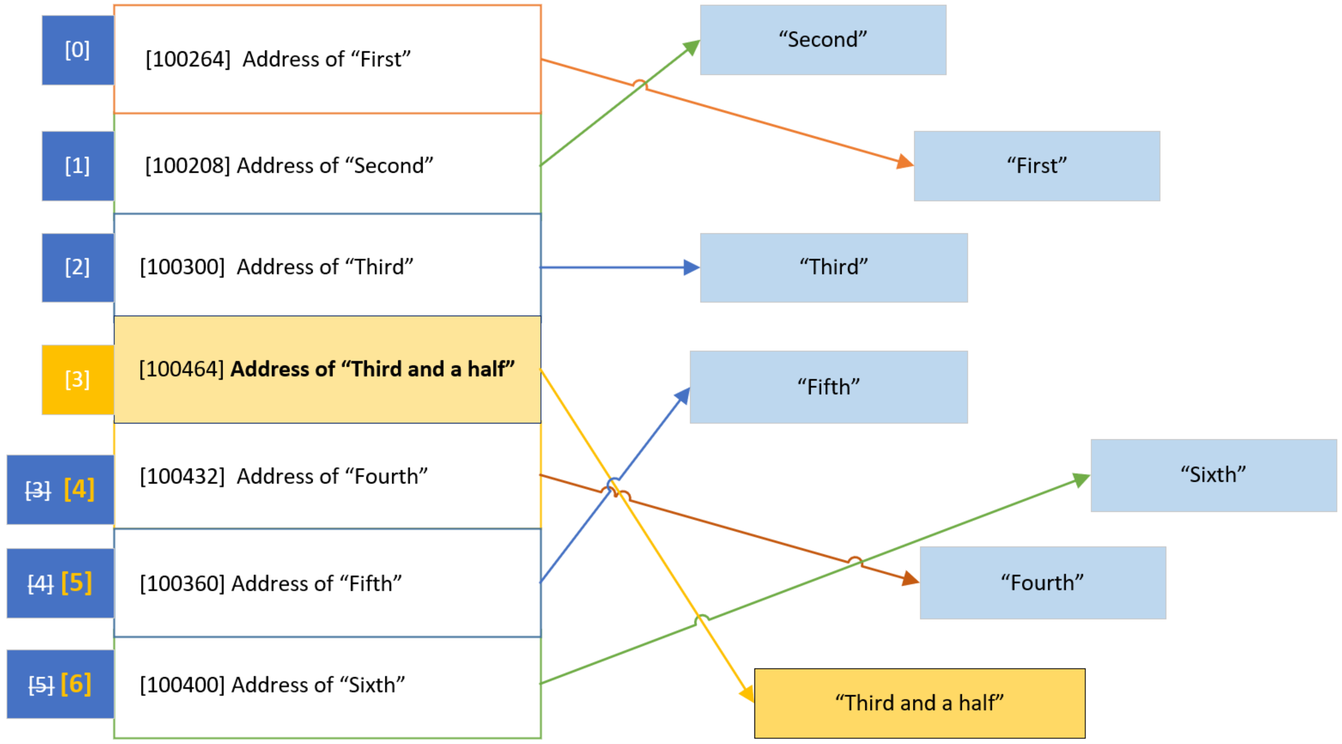
Either of these operations can be an **expensive** process, if the number of elements is large.



# Arrays and ArrayLists of reference types

In this slide, the String, "Third and a half', represents a new element we want inserted, at index position 3.

This means all the elements below this point, need to be moved and re-indexed.



# ArrayList capacity

An ArrayList is created with an initial capacity, depending on how many elements we create the list with, or if you specify a capacity when creating the list.

On this slide, I show an ArrayList that has a capacity of 10, because we're passing 10 in the constructor of this list.

We then add 7 elements.

```
ArrayList<Integer> intList = new ArrayList<>(10);
for (int i = 0; i < 7; i++) {
    intList.add((i + 1) * 5);
}
```

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Value | 5 | 10 | 15 | 20 | 25 | 30 | 35 | | | |

# ArrayList capacity

We can add 3 more elements, using the ArrayList add method, and the array that is used to store the data, doesn't need to change.

```
ArrayList<Integer> intList = new ArrayList<>(10);
for (int i = 0; i < 7; i++) {
    intList.add((i + 1) * 5);
}

intList.add(40);
intList.add(45);
intList.add(50);
```

The elements at indices 7, 8, and 9, get populated.

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Value | 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50 |

# ArrayList capacity is reached

But if the number of elements exceeds the current capacity, Java needs to reallocate memory, to fit all the elements, and this can be a costly operation, especially if your ArrayList contains a lot of items.

```
ArrayList<Integer> intList = new ArrayList<>(10);
for (int i = 0; i < 7; i++) {
    intList.add((i + 1) * 5);
}
intList.add(40);
intList.add(45);
intList.add(50);

intList.add(55);              // This add exceeds the ArrayList capacity,
                              // assuming an initial capacity of 10,
                              // as an example.
```

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Value | 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50 | 55 | | | | |

# ArrayList capacity is reached

So now, if our code simply calls add on this ArrayList, the next operation is going to create a new array, with more elements, but copy the existing 10 elements over.

Then the new element is added.  You can imagine this add operation costs more, in both time and memory, than the previous add methods did.

When Java re-allocates new memory for the ArrayList, it automatically sets the capacity to a greater capacity.

But the Java language doesn't really specify exactly how it determines the new capacity, or promise that it will continue to increase the capacity in the same way in future versions.

We can't actually get this capacity size, from the ArrayList.

| **Index** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Value | 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50 | 55 | | | | |

# ArrayList capacity is reached

From their own documentation, Java states that, "The details of the growth policy, are not specified beyond the fact that adding an element, has constant amortized time cost".

Ok, maybe you're interested in what constant amortized time is.

Let's start with how to determine cost, which in this case is generally considered in terms of time, but may include memory usage and processing costs, etc.

| **Index** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Value | 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50 | 55 | | | | |

# Big O Notation

Maybe you've heard people talking about Big O Notation, or Big O, and wondered what this means.

I won't get too deep into it, but there are a couple of concepts that are fairly easy to grasp, and will help us understand how **cheap** or **expensive** an operation is, in terms of time and memory usage, as the operation scales.

This means it's a way to express how well the operation performs, when applied to more and more elements.

Big O approximates the cost of an operation, for a certain number of elements, called n.

Cost is usually determined by the time it takes, but it can include memory usage, and complexity for example.

# Big O Notation

As n (the number of elements) gets bigger, an operation's cost can stay the same.

But cost often grows, as the number of elements grow.

Costs can grow linearly, meaning the cost stays in step, with the magnitude of the number of elements.

Or costs can grow exponentially, or by some other non-linear fashion.

In a perfect world, an operation's time and complexity would never change. This ideal world, in Big O Notation is O(1), sometimes called constant time.

In many situations, an operation's cost is in direct correlation to the number of elements, n. In Big O Notation this is O(n), sometimes called linear time.

# Big O Notation

So if we have 10 elements, the cost is 10 times what it would be for 1 element, because the operation may have to execute some functions, up to 10 times vs. just once, and 100 times for 100 elements, for example.

O(n) is generally our worst case scenario for List operations, but there are Big O Notations, for worse performers.

# Constant Amortized Time Cost

Another scenario, is the one the Java docs declared for the growth of the ArrayList, that adding an element has constant amortized time cost.

In our case, we'll designate this constant amortized time as O(1)*.

This means that in the majority of cases, the cost is close to O(1), but at certain intervals, the cost is O(n).

If we add an element to an ArrayList, where the capacity of the List is already allocated, and space is available, the cost is the same each time, regardless of how many elements we add.

# Constant Amortized Time Cost

But as soon as we reach the capacity, and all the elements (all n elements) need to be copied in memory, this single add would have a maximum cost of O(n).

After this operation, that forced a reallocation, any additional add operations go back to O(1), until the capacity is reached again.

As the expensive intervals decrease, the cost gets closer to O(1), so we give it the notation O(1)*.

# ArrayList Operations - Big O

This slide shows the Big O values, for the most common ArrayList operations or methods.

Let's just talk about one example, the contains method, which looks for a matching element, and needs to traverse through the ArrayList to find a match.

It could find a match at the very first index, this is the best case scenario, so it's O(1).

It might not find a match until the last index, this is the worst case scenario, so it's O(n).

| Operation | Worst Case | Best Case |
|---|---|---|
| add(E element) | O(1)* | |
| add(int index, E element) | O(n) | O(1)* |
| contains(E element) | O(n) | O(1) |
| get(int index) | O(1) | |
| indexOf(E Element) | O(n) | O(1) |
| remove(int index) | O(n) | O(1) |
| remove(E element) | O(n) | |
| set(int index, E element) | O(1) | |

O(1) - constant time - operation's cost (time) should be constant regardless of number of elements.

O(n) - linear time - operation's cost (time) will increase linearly with the number of elements n.

O(1)* - constant amortized time - somewhere between O(1) and O(n), but closer to O(1) as efficiencies are gained.

# ArrayList Operations - Big O

In general, the cost will be something in between, for the contains method, because the element will be found somewhere between the first and nth (or last) element.

You'll notice that the indexed methods are usually O(1), remembering that finding an element by its index, is a simple calculation.

It only gets costly with indexed add or remove methods, if the ArrayList needs to be re-indexed, or re-sized.

| Operation | Worst Case | Best Case |
|---|---|---|
| add(E element) | O(1)* | |
| add(int index, E element) | O(n) | O(1)* |
| contains(E element) | O(n) | O(1) |
| get(int index) | O(1) | |
| indexOf(E Element) | O(n) | O(1) |
| remove(int index) | O(n) | O(1) |
| remove(E element) | O(n) | |
| set(int index, E element) | O(1) | |

O(1) - constant time - operation's cost (time) should be constant regardless of number of elements.

O(n) - linear time - operation's cost (time) will increase linearly with the number of elements n.

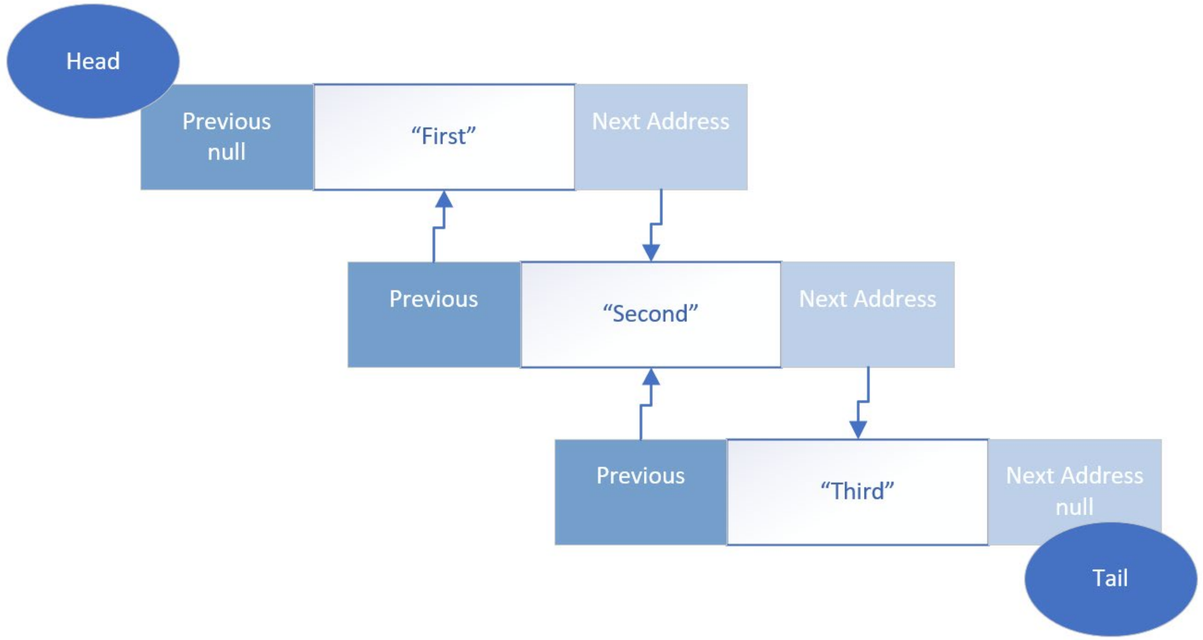
O(1)* - constant amortized time - somewhere between O(1) and O(n), but closer to O(1) as efficiencies are gained.

# LinkedList

The LinkedList is not indexed at all.

There is no array, storing the addresses in a neat ordered way, as we saw with the ArrayList.

Instead, each element that's added to a linked list, forms a chain, and the chain has links to the previous element, and the next element.

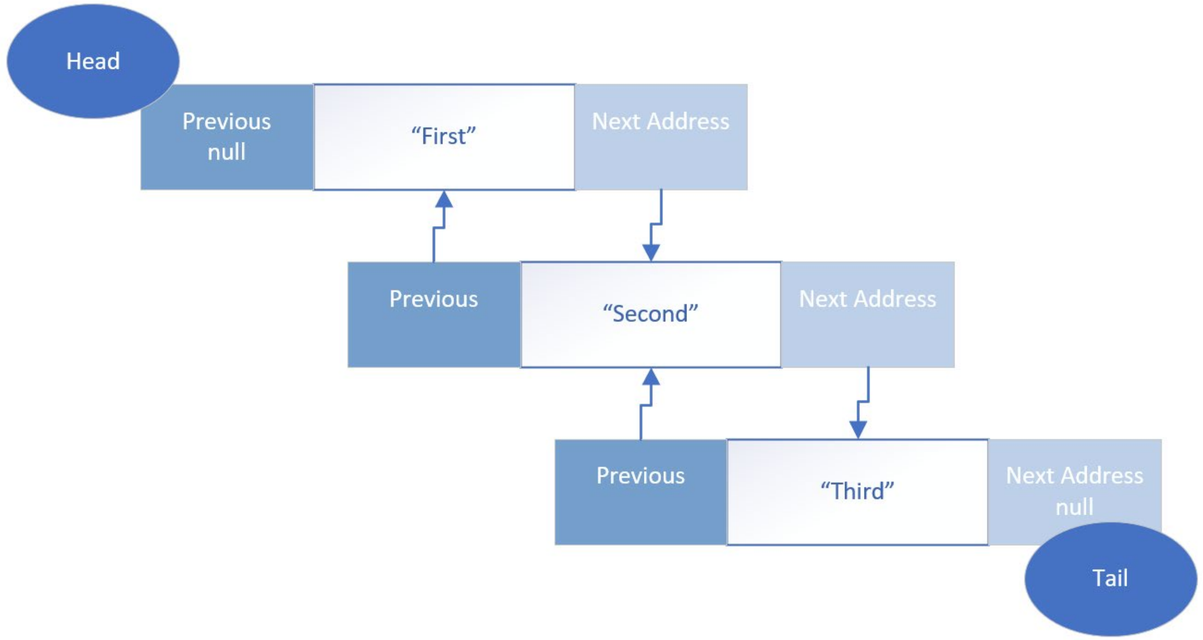


# LinkedList

This architecture is called a doubly linked list, meaning an element is linked to the next element, but it's also linked to a previous element, in this chain of elements.

The beginning of the chain is called the head of the list, and the end is called the tail.

This can also be considered a queue, in this case, a double ended queue, because we can traverse both backwards and forwards, through these elements.



# LinkedList - Retrieval of an Element costs more than an ArrayList retrieval

Getting an element from the list, or setting a value of element, isn't just simple math anymore, with the LinkedList type.

To find an element, we'd need to start at the head or tail, and check if the element matches, or keep track of the number of elements traversed, if we are matching by an index, because the index isn't stored as part of the list.

For example, even if you know, you want to find the 5th element, you'd still have to traverse the chain this way, to get that fifth element.

This type of retrieval is considered expensive in computer currency, which is processing time and memory usage.

On the other hand, inserting and removing an element, is much simpler for this type of collection.

# LinkedList - Inserting or Removing an Element may be less costly than using an ArrayList

In contrast to an ArrayList, inserting or removing an item in a LinkedList, is just a matter of breaking two links in the chain, and re-establishing two different links.

No new array needs to be created, and elements don't need to be shifted into different positions.

A reallocation of memory to accommodate all existing elements, is never required.

So for a LinkedList, inserting and removing elements, is generally considered **cheap** in computer currency, compared to doing these functions in an ArrayList.

# LinkedList and ArrayList Operations - Big O

This slide shows the Big O values, for the most common shared List operations or methods, for both types.

For a LinkedList, adding elements to the start or end of the List, will almost always be more efficient than an ArrayList.

| | Linked List | | ArrayList | |
|---|---|---|---|---|
| **Operation** | **Worst Case** | **Best Case** | **Worst Case** | **Best Case** |
| add() | O(1) | | O(1)* | |
| add(int index, E element) | O(n) | O(1) | O(n) | O(1)* |
| contains(E element) | O(n) | O(1) | O(n) | O(1) |
| get(int index) | O(n) | O(1) | O(1) | |
| indexOf(E Element) | O(n) | O(1) | O(n) | O(1) |
| remove(int index) | O(n) | O(1) | O(n) | O(1) |
| remove(E element) | O(n) | O(1) | O(n) | |
| set(int index, E element) | O(n) | O(1) | O(1) | |

O(1) - constant time - operation's cost (time) should be constant regardless of number of elements.

O(n) - linear time - operation's cost (time) will increase linearly with the number of elements n.

O(1)* - constant amortized time - somewhere between O(1) and O(n), but closer to O(1) as efficiencies are gained.

# LinkedList and ArrayList Operations - Big O

When removing elements, a LinkedList will be more efficient, because it doesn't require re-indexing, but the element still needs to be found, using the traversal mechanism, which is why it is O(n), as the worst case.

Removing elements from the start or end of the List, will be more efficient for a LinkedList.

| | Linked List | | ArrayList | |
|---|---|---|---|---|
| **Operation** | **Worst Case** | **Best Case** | **Worst Case** | **Best Case** |
| add() | O(1) | | O(1)* | |
| add(int index, E element) | O(n) | O(1) | O(n) | O(1)* |
| contains(E element) | O(n) | O(1) | O(n) | O(1) |
| get(int index) | O(n) | O(1) | O(1) | |
| indexOf(E Element) | O(n) | O(1) | O(n) | O(1) |
| remove(int index) | O(n) | O(1) | O(n) | O(1) |
| remove(E element) | O(n) | O(1) | O(n) | |
| set(int index, E element) | O(n) | O(1) | O(1) | |

O(1) - constant time - operation's cost (time) should be constant regardless of number of elements.

O(n) - linear time - operation's cost (time) will increase linearly with the number of elements n.

O(1)* - constant amortized time - somewhere between O(1) and O(n), but closer to O(1) as efficiencies are gained.

# Things to Remember when considering whether to use an ArrayList vs LinkedList

The ArrayList is usually the better default choice for a List, especially if the List is used predominantly for storing and reading data.

If you know the maximum number of possible items, then it's probably better to use an ArrayList, but set it's capacity.

This code demonstrates how to set the capacity of your ArrayList to 500,000.

```
int capacity = 500_000;
ArrayList<String> stringArray = new ArrayList<>(capacity);
```

# Things to Remember when considering whether to use an ArrayList vs LinkedList

An ArrayList's index is an int type, so an ArrayList's capacity is limited to the maximum number of elements an int can hold, Integer.MAX_VALUE = 2,147,483,647.

You may want to consider using a LinkedList if you're adding and processing or manipulating a large amount of elements, and the maximum elements isn't known, but may be great, or if your number of elements may exceed Integer.MAX_VALUE.

A LinkedList can be more efficient, when items are being processed predominantly from either the head or tail of the list.

```
int capacity = 500_000;
ArrayList<String> stringArray = new ArrayList<>(capacity);
```