Array

This lesson talks about the complexity of operations on an array.

In this section of the course, we'll consider various data-structures and understand how different operations on them can be analyzed for complexity. We'll also digress into general discussions around trade-offs of space and time, so as to cultivate the readers' thought process for reasoning about complexity.

We'll start with the most basic data-structure - *the Array*. Almost all languages allow users to create arrays. Insert or retrieval operations on an array are well-known to be constant or O(1) operations. But what makes these operations constant time operations?

Contiguous memory allocation is the key to an array's constant time retrieval/insert of an element given the index. Any language platform that you code in would request the underlying operating system to allocate a contiguous block of memory for an array. Suppose the following is a block of memory allocated by the operating system for an array of 10 elements in computer memory.

Block of memory allocated by the operating system.

Suppose the array we requested above is for integers. The next part to the puzzle is the size of the *type* for which we requested the array. The type refers to the data type the array holds, it could be integers, characters or even objects. In the case of Java, an integer is 4 bytes so let's go with that. Realize that if given the starting address of the array in the computer memory, it is trivial to compute the starting memory address of any element in the array given the index of the element.

As an example, say the starting address of the array for the below snippet of code is 25.

```
// initialize array of 10 elements.
// The variable myArray points to memory address 25
int[] myArray = new int[10]
```

Arrays with 10 elements and start address of 25

The variable `myArray` literally points to **25** in the computer's memory. We also know each integer is 4 bytes in size. The first integer would lie at address **25**, the second integer would lie at **25** + **(1 * 4)** = **29**, the third integer would lie at **25** + **(2 * 4)** = **33**. To find out where the **nth** element of the array lies in the memory, we can use the following generalized formula:

$$Address\ of\ Nth\ Array\ Element = (Start\ Address\ of\ Array + (n * Size\ of\ Type))$$

For our array example the formula would become:

$$A[n] = Start\ Address\ of\ Array + (n * 4)$$

Memory representation of first 3 integers of the array

The final piece to the puzzle is to understand that only a mathematical calculation is required in order to access any element of the array. Ask yourself if the array size is 10, 100 or 1 million? Does the number of steps change for computing the formula to find the address of any element in the array? No, it doesn't. The size of the array has no bearing on how we retrieve

array? No, it doesn't. The size of the array has no bearing on how we retrieve the address for a given element. Once we compute the address, the OS and the underlying hardware will work in tandem to retrieve the value for the array element from the physical memory.

The mathematical calculation will always take the same amount of time whether we retrieve the address for the first element or the millionth element in the array. Hence retrieval or access operation on an array is constant time or O(1). Retrieving the last element or the middle element of the array requires the same number of steps to compute the element's address in memory.

Other Considerations

The astute reader would notice that an array forces us to block a chunk of memory. If most of our array is sparse then we end up wasting a lot of space. This is a well-known trade-off we make to gain time in exchange for space. Insert and retrieval become constant time operations, but then we may be wasting a lot of space.

Nuances in Scripting Languages

Folks with experience in Javascript or other languages which don't require declaring the size of the array at the time of initialization, may wonder if the same principles discussed above apply. Yes, they do. Behind the scenes, arrays have fixed sizes. If the user attempts to add items to an array which is completely filled, the platform will create a new array. The new array will have bigger capacity, and it copies elements from the previous array into the new array. This complexity is masked from the developer. These types of self-resizing arrays are called *dynamic arrays* and are discussed in later sections.

Pop Quiz

Q

In our analysis we reasoned that since the size of the type is fixed (integer was assumed to be 4 bytes), we can mathematically compute the starting address of any item in the array. Would the array access still be constant time if it were an array of types which could have varying sizes?

Assume you have an array of objects. These objects could have any number of fields including nested objects. Can we consider the access time to be constant?

Check Answers