Courses Practice Roadmap Pro

## 2 - Static Arrays

14:35

Mark Lesson Complete View Code Prev Next

## Suggested Problems

| Status | Star | Problem | Difficulty | Video Solution | Code |
|---|---|---|---|---|---|
| ☐ | ☆ | Remove Duplicates From Sorted Array | Easy | | C++ |
| ☐ | ☆ | Remove Element | Easy | | C++ |

## Static Arrays

To recall, arrays are a way to store data contiguously. In strictly typed languages like Java, C++, C# etc, arrays have to have an allocated size when initialized. This means that the size of the array cannot change once declared and once its capacity is reached, it can store no more values. Loosely typed languages such as Python and JavaScript do not have static arrays, which we will discuss in the next chapter.

Having said all that, let's go over the basic operations that can be performed on an array, their efficiency and usage. The most common operations are:

- Reading
- Deletion

- Insertion

## Reading from an array

For this example, we will be initializing an array of size 3 called `myArray`. Array elements are accessed using indices, which start at $0$. So the first element resides at index $0$, second at index $1$ and so on.

```
// initialize myArray
myArray = [1,3,5]

// access an arbitrary element, where i is the index of the desired val
myArray[i]
```

As long as the index of an element is known, the access is instant. This is because each index of `myArray` is mapped to an address in the RAM. This is a single operation and in algorithm analysis, we say it runs in constant time, or $O(1)$ put more formally. This means that regardless of the size of the array, the time taken to access our element will be unaffected - it will always be a single operation.

## Traversing through an array

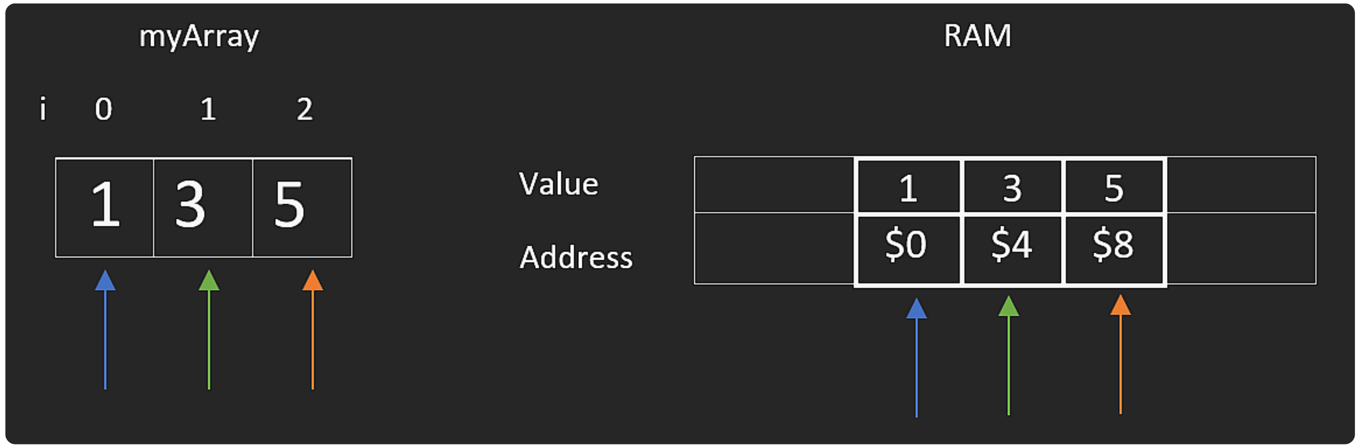
Reading all values from an array can be done using either a for loop or a while loop.

```
for i = 0 to myArray.size-1:
   print(myArray[i])

// OR

i = 0
while i < myArray.size:
   print(myArray[i])
   i++
```

> *In the for loop, we only iterate until* `myArray.size-1` *since that is the last accessible index of the array. The last index of the array is $n - 1$ where $n$ is the length of the array. This makes sense because the size of our array is 3, meaning it can hold three elements and if we start our index at 0, the last accessible index will be 2. If we try and access index 3, we will get an error.*



`myArray` easily accessible through indices in the memory.

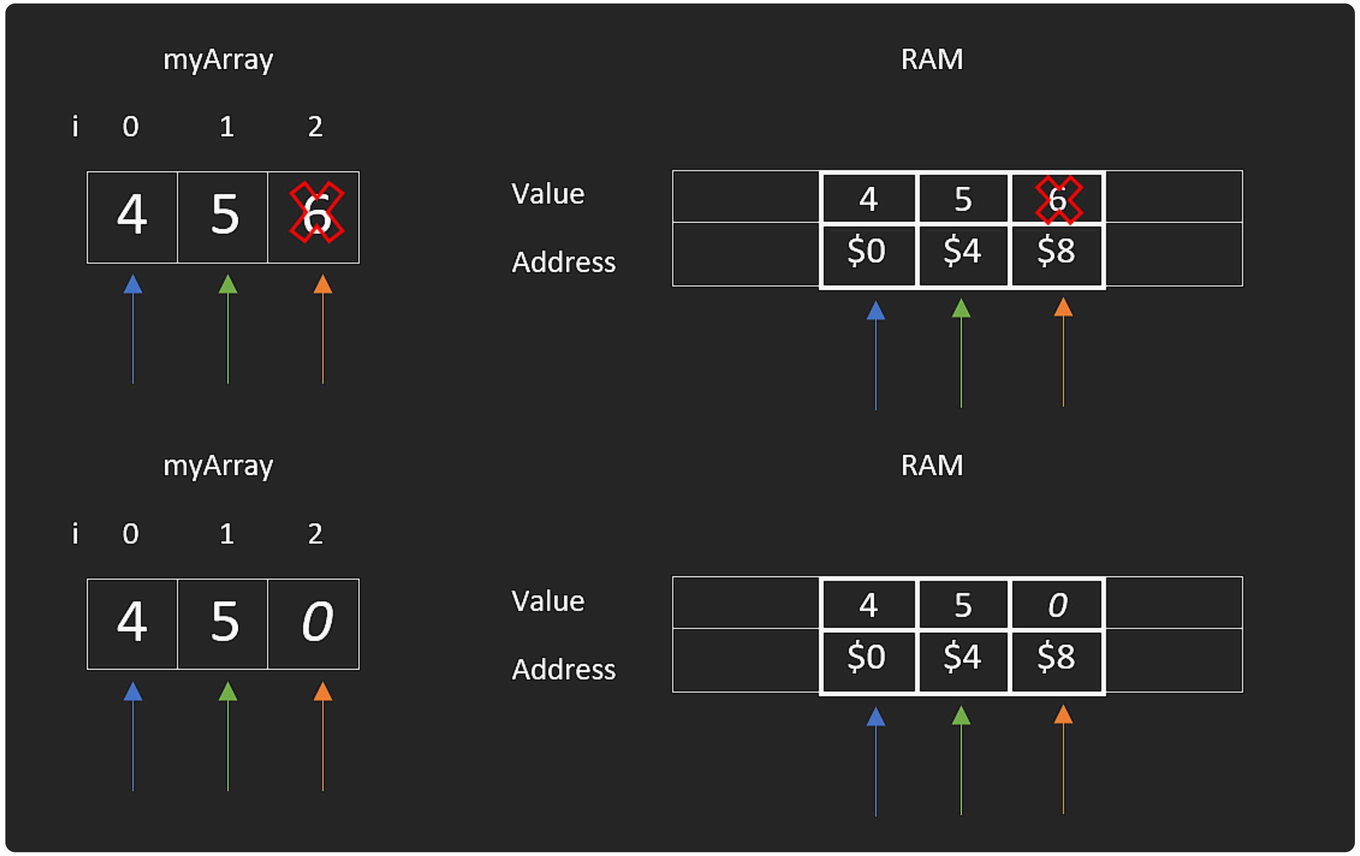
## Deleting from an array

### Removing from the end of the array

In strictly typed languages, all array indices are filled with `0s` or some default value upon initialization - this denotes an empty array. Taking this into consideration, when we want to remove an element from the ***end*** of an array, we set its value to either `0`, or `null`. It is not being "deleted" per se but rather overwritten by a value that denotes an empty index. The following pseudocode demonstrates the concept using `myArray = [4, 5, 6]` as an example.

```
fn removeEnd(myArray, length):
    if length > 0:
```

```
myArray[length - 1] = 0
```



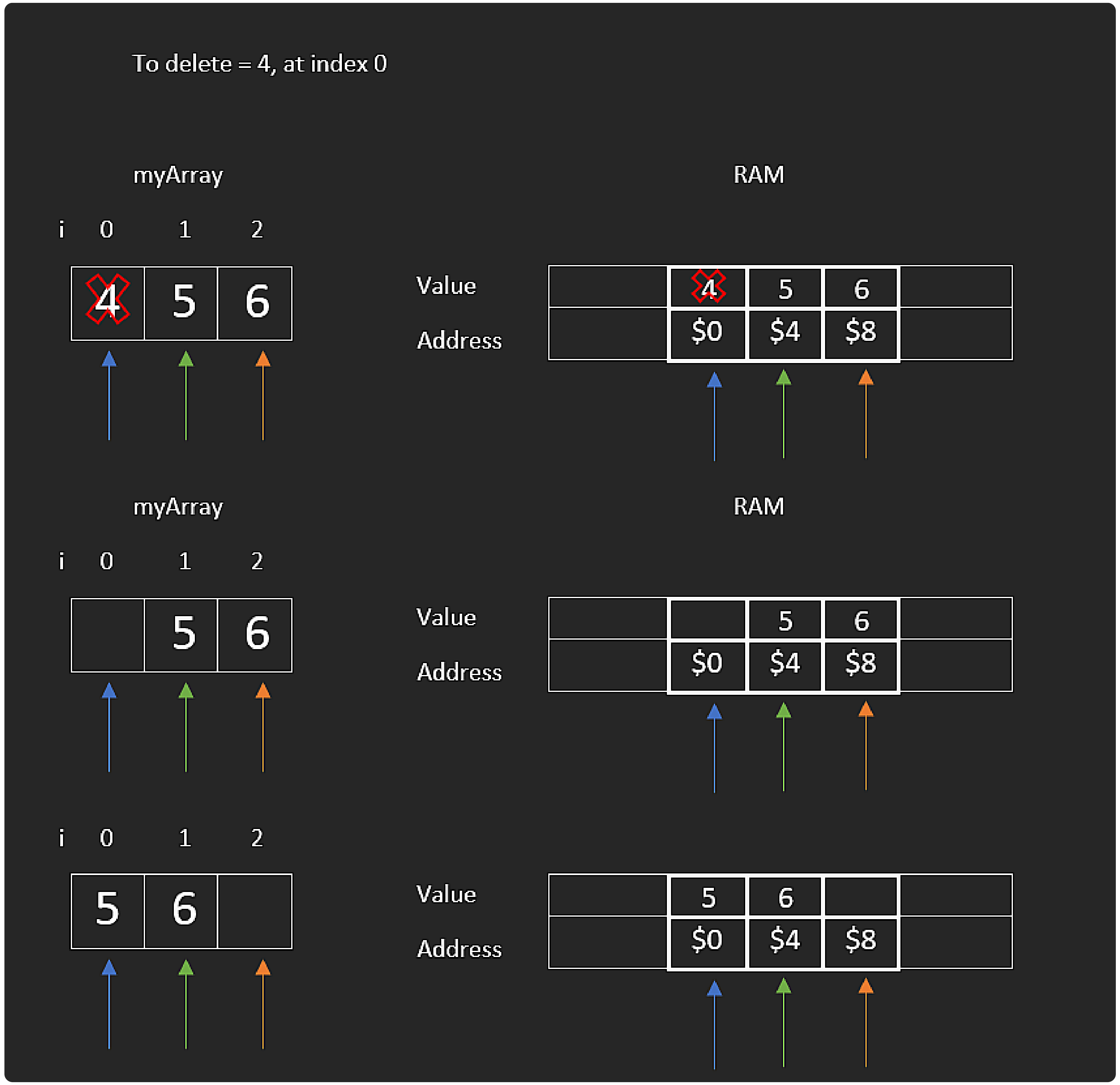
6 is deleted/overwritten by either 0 or -1 to denote that it does not exist anymore.

## Removing at the `ith` index

Using the same instance of `myArray` from the previous example, let's say that instead of deleting at the end, we wanted to delete an element at a random index. Would we be able to perform this in $O(1)$? We could just replace it with a zero and call it a day. However, what if the index we wanted to delete at was $0$?. If replaced with a $0$, we would have an empty first index, which does not make sense.

In the worst case, the random index might be the $0$-th index, in which case, upon deletion, all the elements from index $1$ all the way till $n-1$ will shift one position to the left, where $n$ is the size of the array. This is seen in the pseudocode and visual below.

```
fn removeMiddle(myArray, i, length):
    for index = i + 1 to length:
        myArray[index - 1] = myArray[index]
```


To delete = 4, at index 0
myArray
RAM
i 0 1 2
4 5 6
Value
Address
4 5 6
$0 $4 $8
myArray
RAM
i 0 1 2
5 6
Value
Address
5 6
$0 $4 $8
i 0 1 2
5 6
Value
Address
5 6
$0 $4 $8

> *In the worst case, `n` shifts may be required, therefore the above code will be in $O(n)$.*

## Insertion

### Inserting at the end of the array

Since we can always access the last index of the array, inserting an element to the end of an array is $O(1)$ time. Below is pseudocode demonstrating the concept.

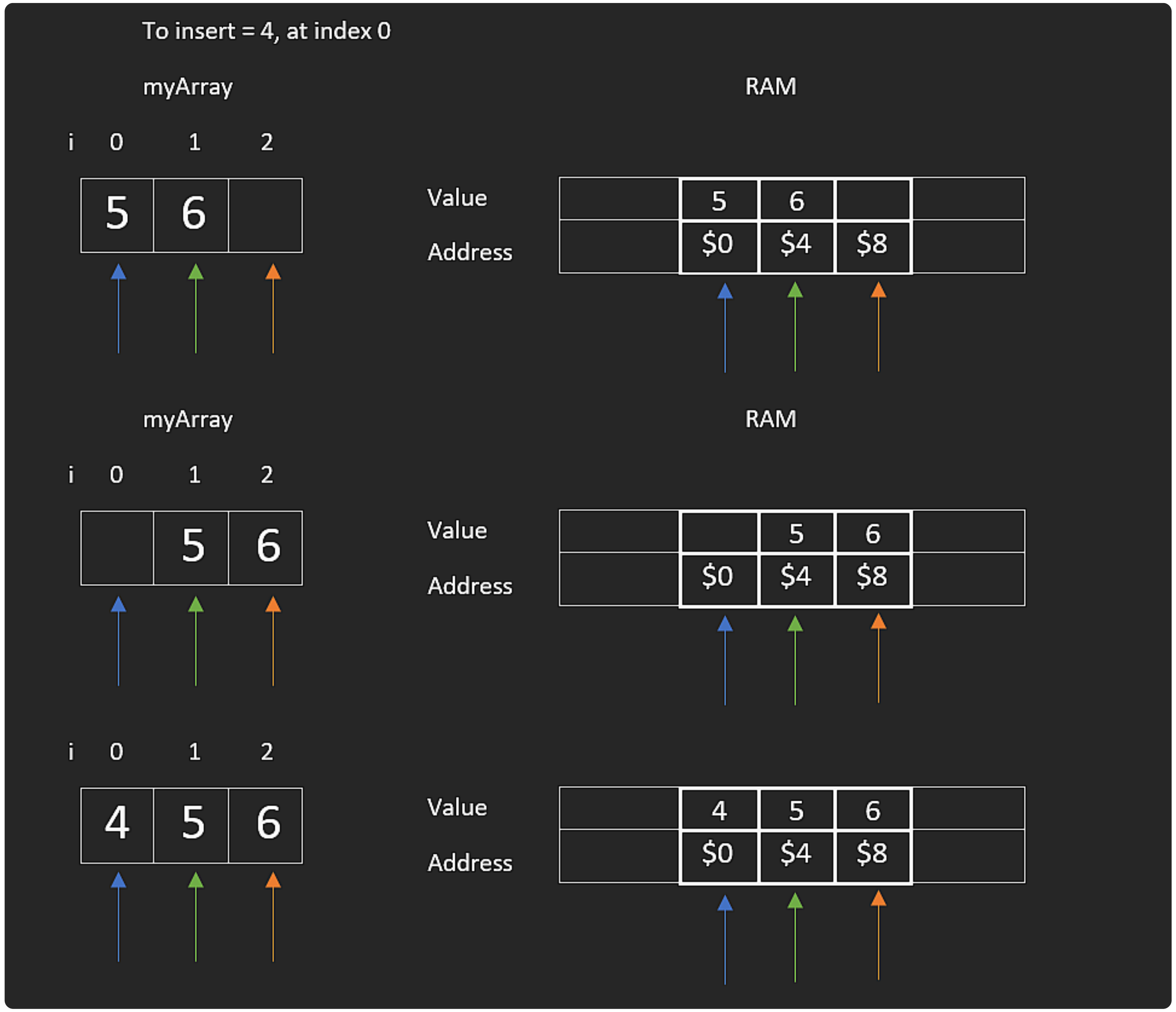
```
fn insertEnd(myArray, n, length, capacity):
    if length < capacity:
        myArray[length] = n
```

> *Here, length is the number of indices that are non-empty and capacity is the actual size of the array that is declared upon instantiation. This makes sense because we are assigning the `length` index, which is the current last index, to `n`, which is the desired value.*

### Inserting at the `ith` index

Inserting in the `ith` index is a little bit more tricky. With the current state of myArray, we have `[4, 5, 0]` with the last index being empty. If the ith index is the last index, we know it will be $O(1)$. But what if it is at index `1` , or `0` ? We cannot overwrite because we would lose our current values. What we can do is shift the current values one position to the right before we insert our new value, say, at index $0$. Below is the pseudocode and visual demonstrating this.

```
fn insertMiddle(myArray, i, n, length):
    for index = length - 1 and index > i - 1:
        myArray[index + 1] = myArray[index]
    myArray[i] = n
```



> *First we create space, and then we insert into the desired index.*

## Closing Notes

| **Operation** | **Big-O Time** | **Notes** |
|---|---|---|
| Access | $O(1)$ | |
| Insertion | $O(n)$* | If inserting at the end of the array, $O(1)$ |
| Deletion | $O(n)$* | If deleting at the end of the array, $O(1)$ |