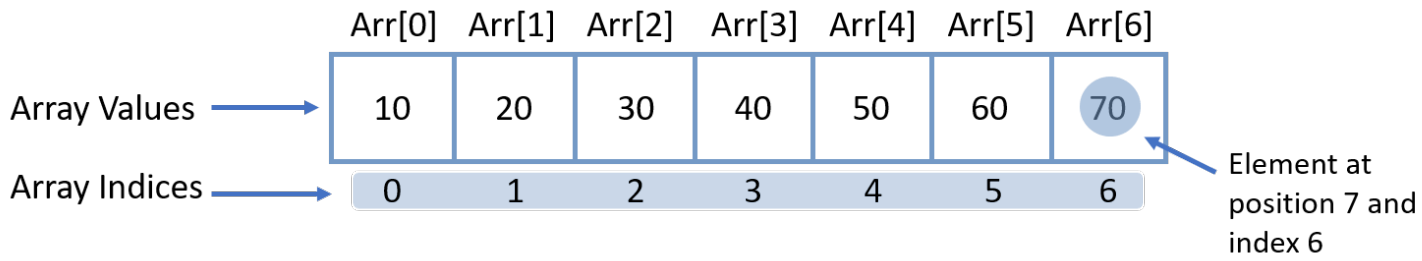
Arrays

Introduction

- An array is a collection of items of the same type i.e. a **collection of elements of a specific data type**.
- We can have arrays of different types:
    - An array of **Integers**.
    - An array of **Floats**.
    - An array of **Booleans**.
- At a memory level, arrays are stored in contiguous locations i.e. all the elements are stored in adjacent memory locations and in order.



Definition of Array

https://www.youtube.com/watch?v=55l-aZ7_F24

Types of Indexing

There are three common ways arrays are indexed:

1. **Zero-based Indexing**: The index of the array starts from 0. This type of indexing is the most common type and seen in languages such as Java, C, C++, etc.
2. **One-based Indexing**: The index of the array starts from 1. This type of indexing is seen in languages such as R, MATLAB.
3. **n-based Indexing**: The indices in such languages can start and end at any integer value. This type of indexing is seen in languages such as Ada.

Some additional reading on why 0 based indices are preferred:

Why we prefer 0 based indexing

Common Array Operations

Some commonly used array operations are:

- **Traversing**: This is visiting each element in the array in order.
- **Insertion**: An element can be inserted into the array at a specific index.
- **Deletion**: An element can be deleted from a specific index in an array.
- **Updates**: Update the value at a given index.
- **Sorting:** The array can be reordered based on the values.
- **Searching**: Find the index of an element given a value.

Searching and Sorting will be covered in a future module.

Traversing

```
algorithm Traverse(Arr[0...N-1]):
  for i -> 0 to N-1:
    // Access the element in the array using Arr[i]
    // Perform any required operation using the element
  end for
```

- In the above algorithm, N is the length of the array. Most languages will provide an interface to get the length of an array. For example, in Java, array.length (length property of the array class) will give you the length. In python, the len(list) function will give you the length of the list.
- **Note:** *Python lists are not pure arrays because they allow heterogeneous data types in them and don't always store elements in contiguous locations*

Insertion, Deletion, and Updation

- For Insertion the provided input should be the following:

    - Array, the position at which the operation must be performed and the element to be inserted.
- Note that for a 0-based indexed array, the index at which the insertion must be performed will be (position - 1).
- Another point to note is for this to work, the array must have an empty space in order to perform the insert. This is to prevent an issue that commonly occurs in C-like languages called **ArrayIndexOutOfBounds**. Read about it here: https://www.educative.io/edpresso/what-is-the-arrayindexoutofbounds-exception-in-java#:~:text=The%20ArrayIndexOutOfBounds%20exception%20is%20thrown,for%20this%20error%20during%20compilation.
- Let's look at the algorithm for this. In the algorithm, we make the assumption position N is empty.

```
algorithm InsertAtIndex(Arr[0...N], position, element):
    for i -> N to position:
        Arr[i] = Arr[i-1]
    end for
    Arr[position-1] = element
```

- For deletion, the input could be either of the following:
    - Array and the position at which deletion is to be performed.
    - Array and the element which is to be deleted.
      Let's look at the algorithm for both.

```
algorithm deleteAtPosition(Arr[0...N-1], position):
  for i <- position to N-1:
      Arr[position – 1] = Arr[position]
  End for
```

```
algorithm deleteElement(Arr[0…N-1], element):
  position <- Search(Arr[0…N-1], element)
  for i <- position to N-1:
      Arr[position – 1] = Arr[position]
  End for
```

- For updates, the input could be either of the following:
    - Array, updated value, and the position at which deletion is to be performed.
    - Array, updated value, and the element which is to be deleted.
      Let's look at the algorithm for both.

```
algorithm updateAtPosition(Arr[0…N-1], position, updatedValue):
   Arr[position -1] <- updatedValue
```

```
algorithm updateElement(Arr[0…N-1], element, updatedValue):
   position <- Search(Arr[0…N-1], element)
   Arr[position -1] <- updatedValue
```

In order to delete or update an element, we need to look at how to search in an array. We will cover that in a future module. For now, think of "search" as a function you can call in your update/delete routine.

Multidimensional Arrays

- Multidimensional arrays can be visualized as matrices.
- Let's consider a 2-D array:

	Column 1	Column 2	Column 3	Column 4
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]

https://www.youtube.com/watch?v=36z4qgN3GWw

- Multidimensional arrays support all of the same operations as arrays.
- They are most useful when dealing with matrix-related problems.

Summary - What we learnt and Further Reading

As part of this module:

1. We learned what is an array.
2. Different types of array indexing
3. Common array operations.
4. How to Traverse an array
5. Insert an element, Delete an element and Update an element in an array.
6. What is a multidimensional array?

For advanced topics related to arrays check the below link:

Arrays and Pointers

Multidimensional Pointer Arithmetic

Mapping 2D arrays to Linear memory