**Array**

**Definition:**

- An array is a data structure consisting of a collection of elements, each identified by an array index starting from 0. An array is stored such that the position of each element can be computed from its index cell by a mathematical formula.

**Properties of Array:**

- Array can store data of specified data type.

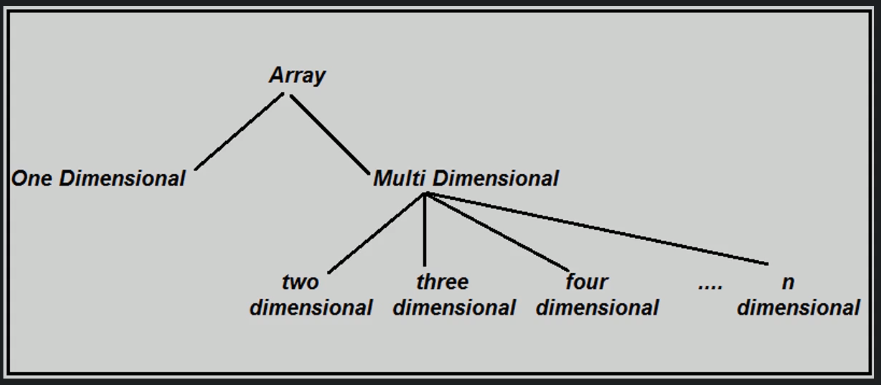
- It has contiguous memory location.

- Every ‘cell’ of an Array has a unique ‘Index’.

- ‘Index’ starts with 0.

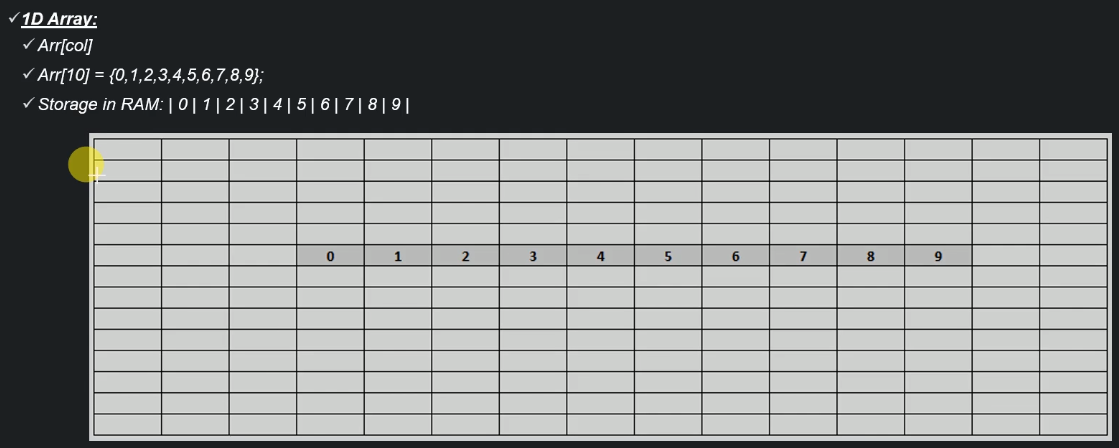
- ‘Size of Array’ needs to be specified mandatorily and cannot be modified.

**Types of Arrays:**

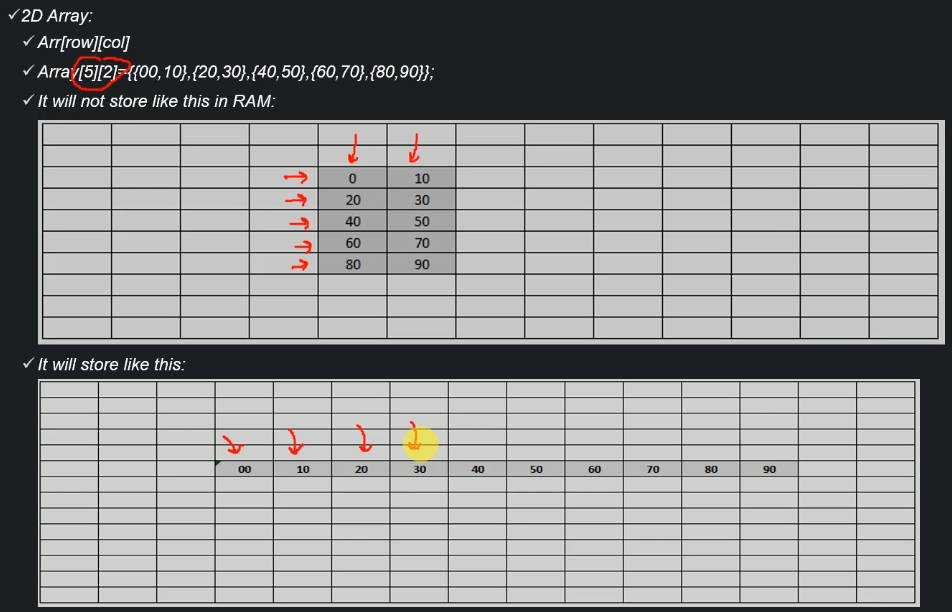


**Arrays in Memory:**

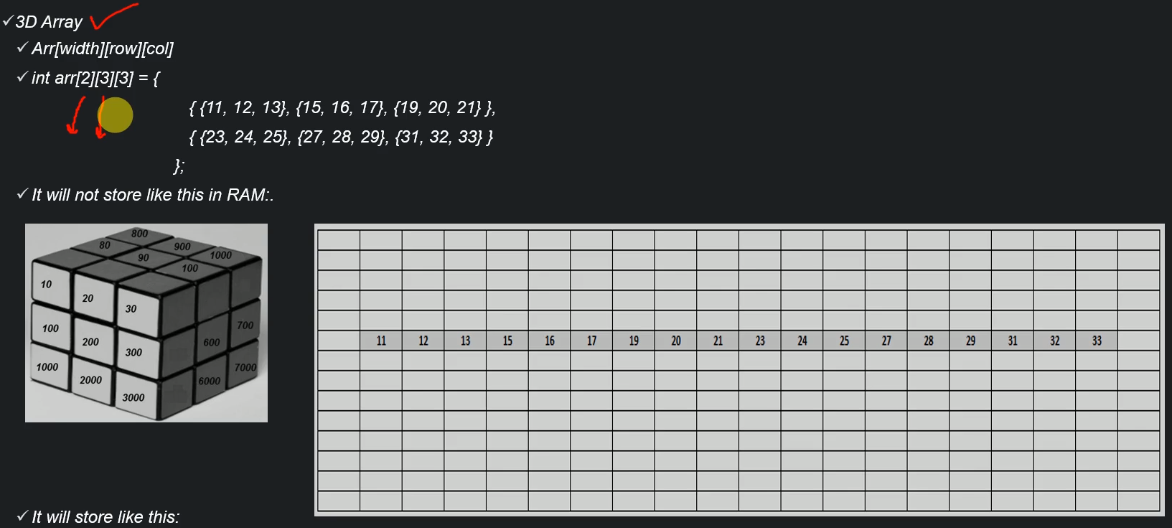
- For one-dimensional arrays:



- For two-dimensional arrays:



- For three-dimensional arrays:

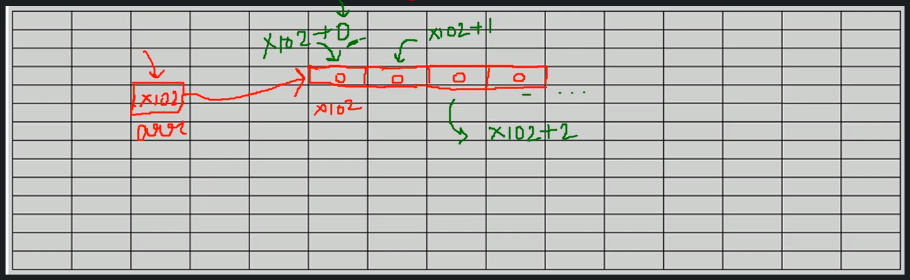


**Operations on and Time Complexity of a 1D array:**

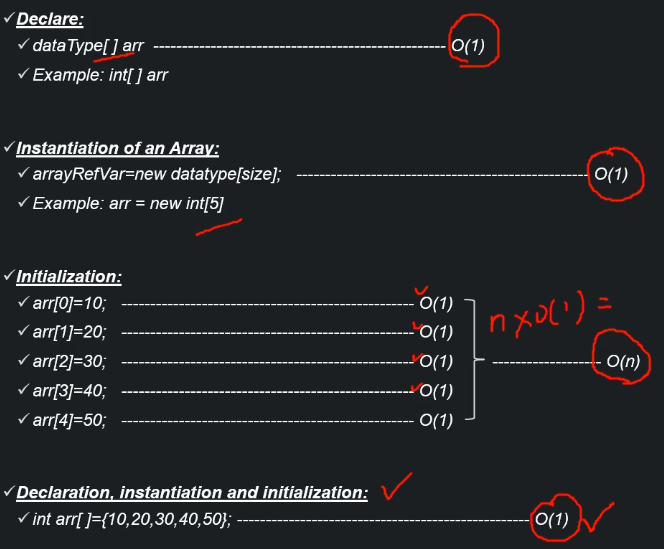
- Creation:

+ Consists of three smaller operations:

* Declaration: Creates a pointer to the array
* Instantiation of the array: Allocates memory to the array
* Initialization of the array: Assign values to cells within the array



+ Complexity:



- Insertion:

+ Note that insertion into arrays **DOES NOT** mean that more cells will be added to the array; Instead, we are re-using empty/recycled cells 🡺 The number of cells always stay the same.

If we need more cells, we have to allocate a new array and transfer all data from the old to the new, larger array.

+ Insertion algorithm:

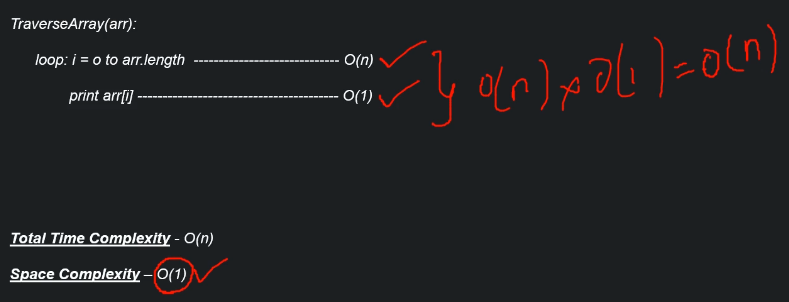


Space complexity is O(1) because we don’t need extra space to execute this operation. The array already exists in memory.

- Traversal:

+ To traverse an array is to visit all the cells of it.

+ Traversal algorithm:

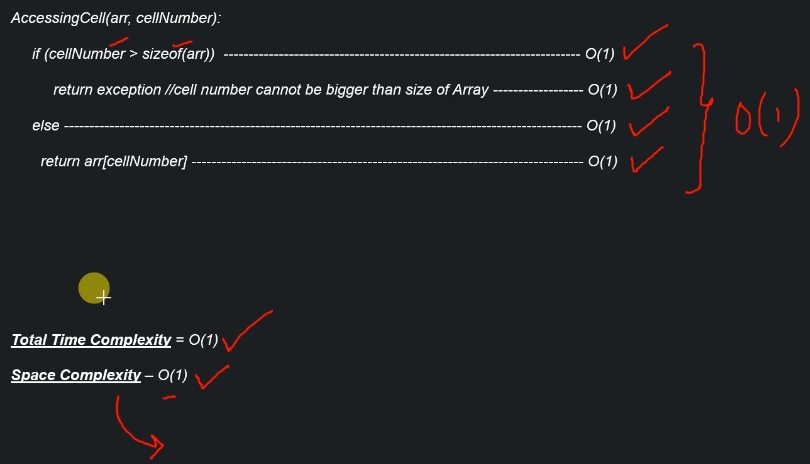


Space complexity is still O(1) because the array has already existed in memory.

- Accessing:

+ Simply access the value of a cell within an array using subscripting syntax [].

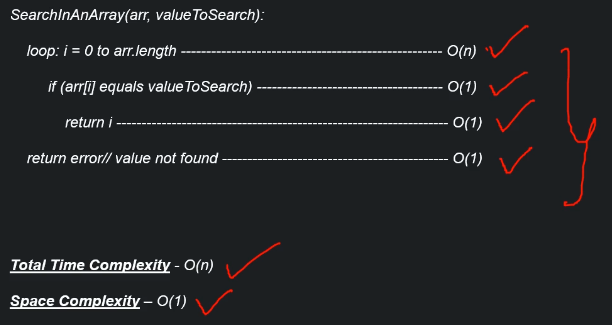
+ Accessing algorithm:



- Search:

+ Search for a specific value within the array. If the value exists, extract it; otherwise, we return null.

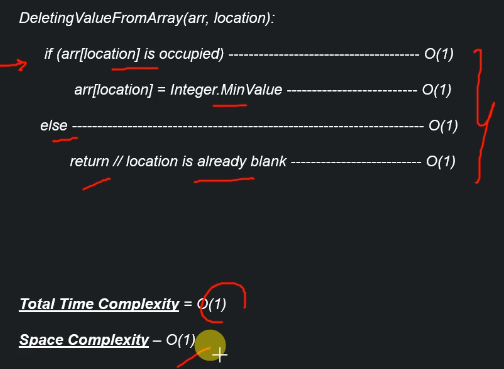
+ Search algorithm:



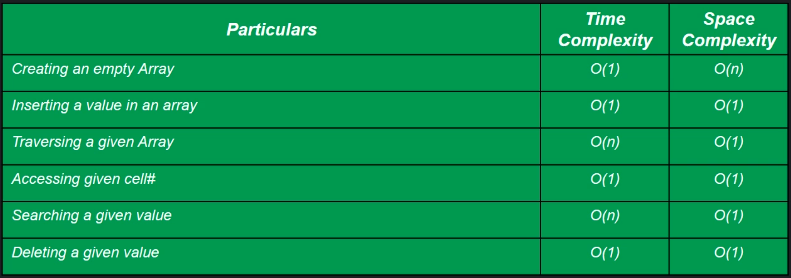
- Deletion:

+ We cannot actually delete a cell and decrease the size of the array in memory. Instead, we can choose a value so that whenever a cell is holding this value, it can be understood as being empty.

+ Deletion algorithm:



- Time and Space Complexity Summary:

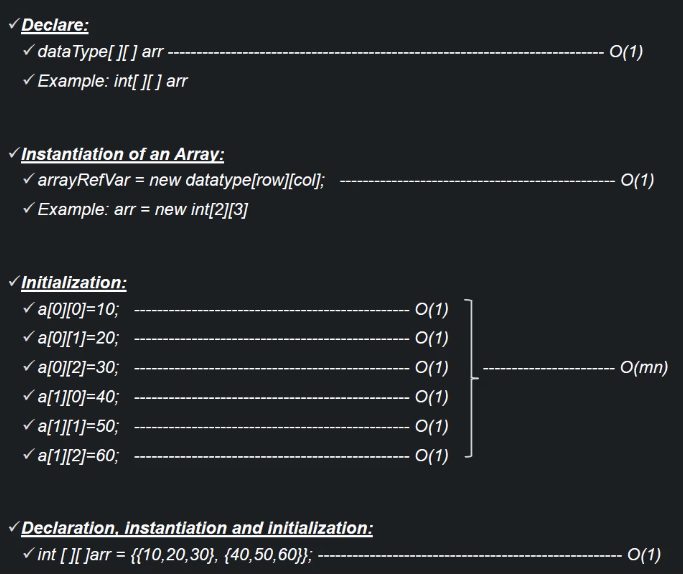


**Operations on and Time Complexity of a 2D array:**

- Creation:

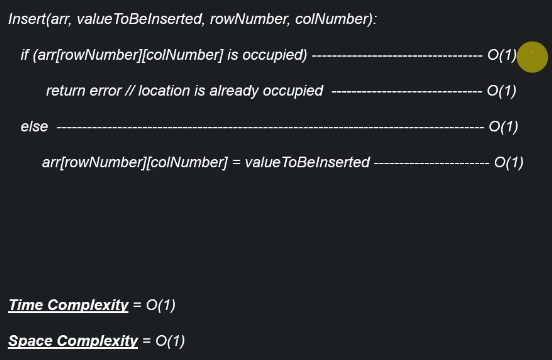
+ Consists of three smaller operations:

* Declaration: Creates a pointer to the array
* Instantiation of the array: Allocates memory to the array
* Initialization of the array: Assign values to cells within the array



- Insertion:

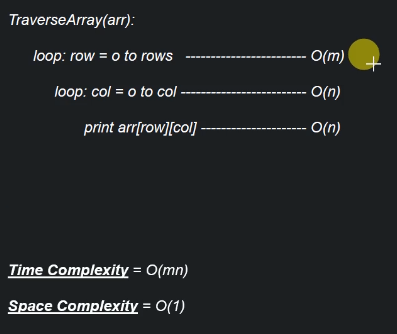
+ In similar manner to 1D array, the insertion algorithm for 2D array is:



- Traversal:

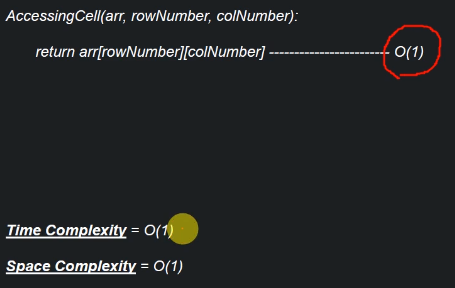
+ For 2D arrays, we need to use nested loop to be able to traverse all cells within them.

+ Traversal algorithm:



- Accessing:

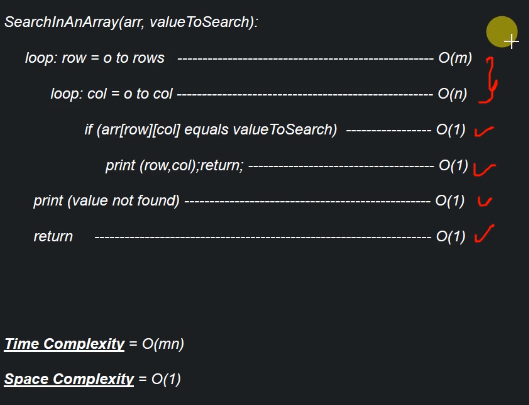
+ Accessing algorithm:



- Searching:

+ We make use of nested loops to traverse through each cell of the 2D array until we either obtain the wanted value or confirm its non-existence.

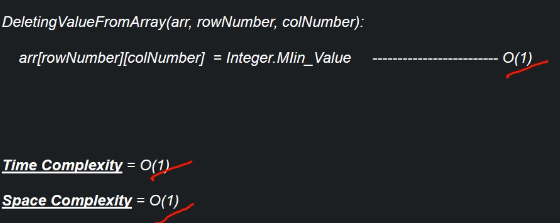
+ Search algorithm:



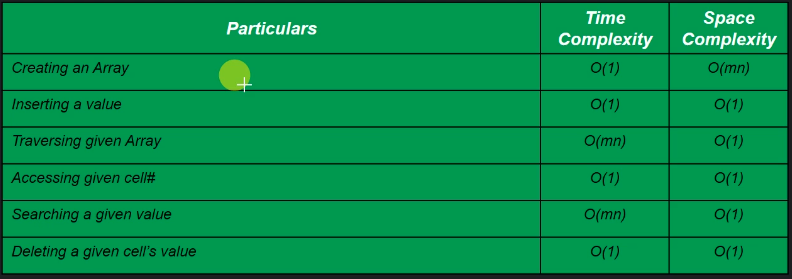
- Deletion:

+ We make use of the same deletion idea from the 1D array algorithm.

+ Deletion algorithm:



- Time and Space Complexity Summary:



**When to use/avoid arrays:**

- Use when:

+ When there is a need to store a large number of values of similar type of data.

+ When random access (fast access) is vital.

- Avoid when:

+ Data to be stored are non-homogenous i.e. not of the same data type.

+ The number of values to be stored are not known in advance.