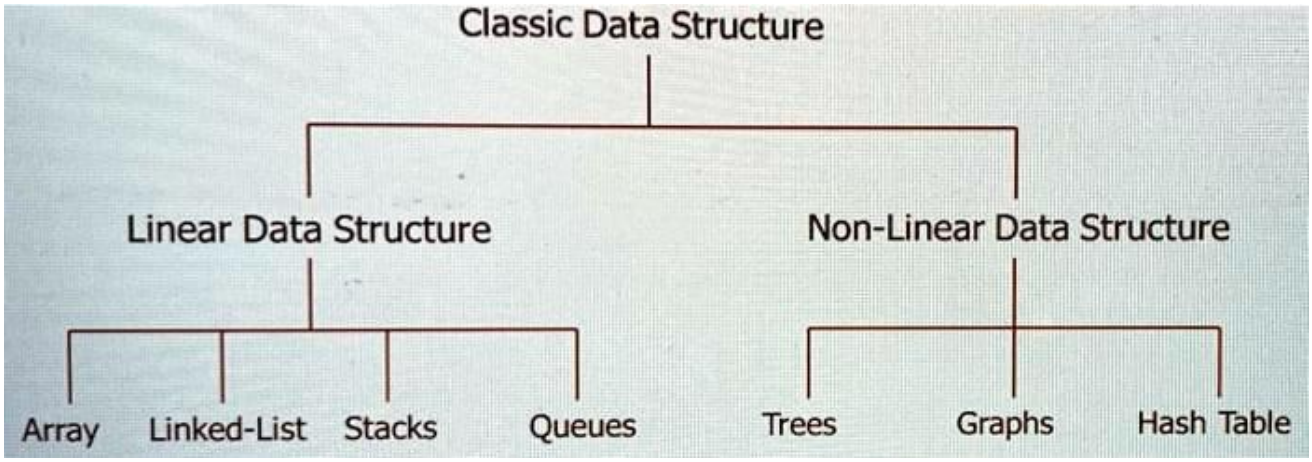
What is a Data Structure?:

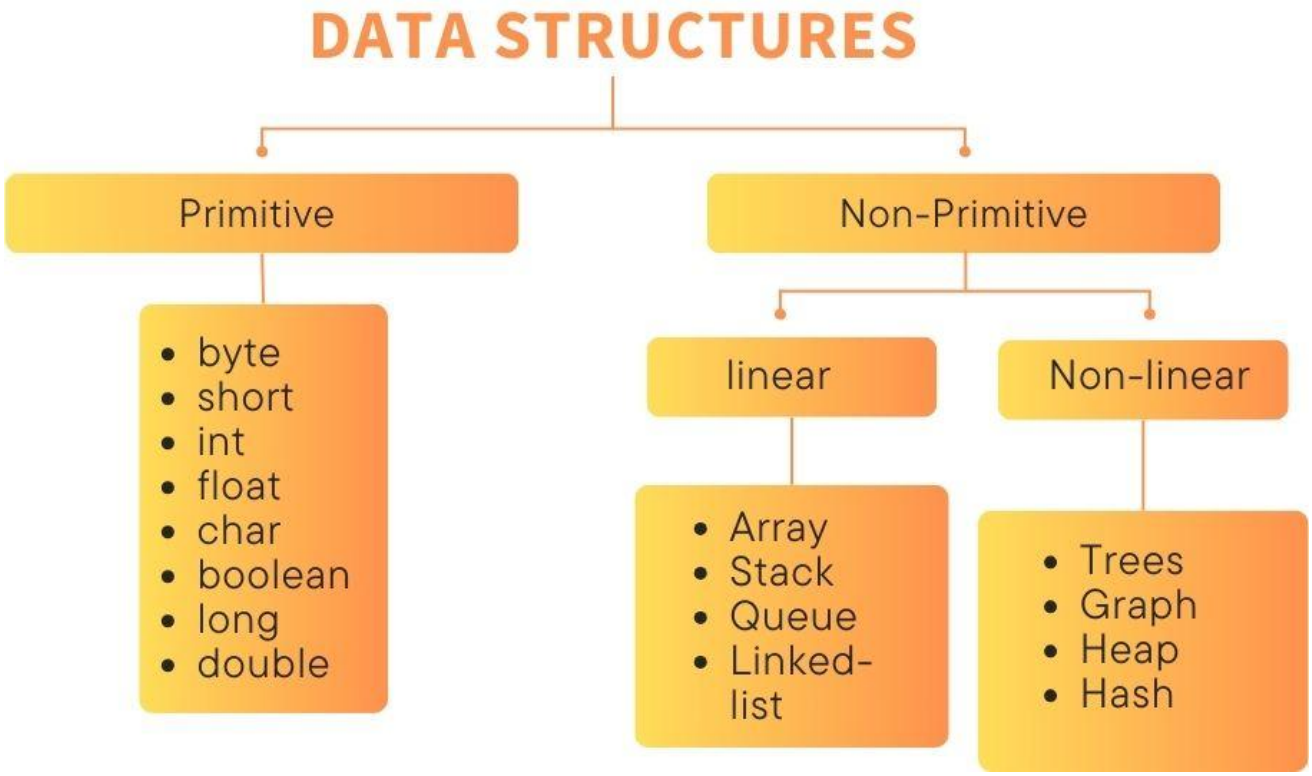
A data structure (DS) is a way of organizing data so that it can be used effectively.

Why Data Structures ? :

- ❖ They are essential ingredients in creating fast and powerful algorithms.
- ❖ They help to manage and organize data.
- ❖ They make code cleaner and easier to understand.



Also, broadly can be classified in above way too:



Abstract Data Type vs Data Structures :-

Abstract Data Type

An abstract data type is an abstraction of a data structure which provides only the interface to which a data structure must adhere to.

The interface doesn't give any specific details about how something should be implemented or in what programming language.

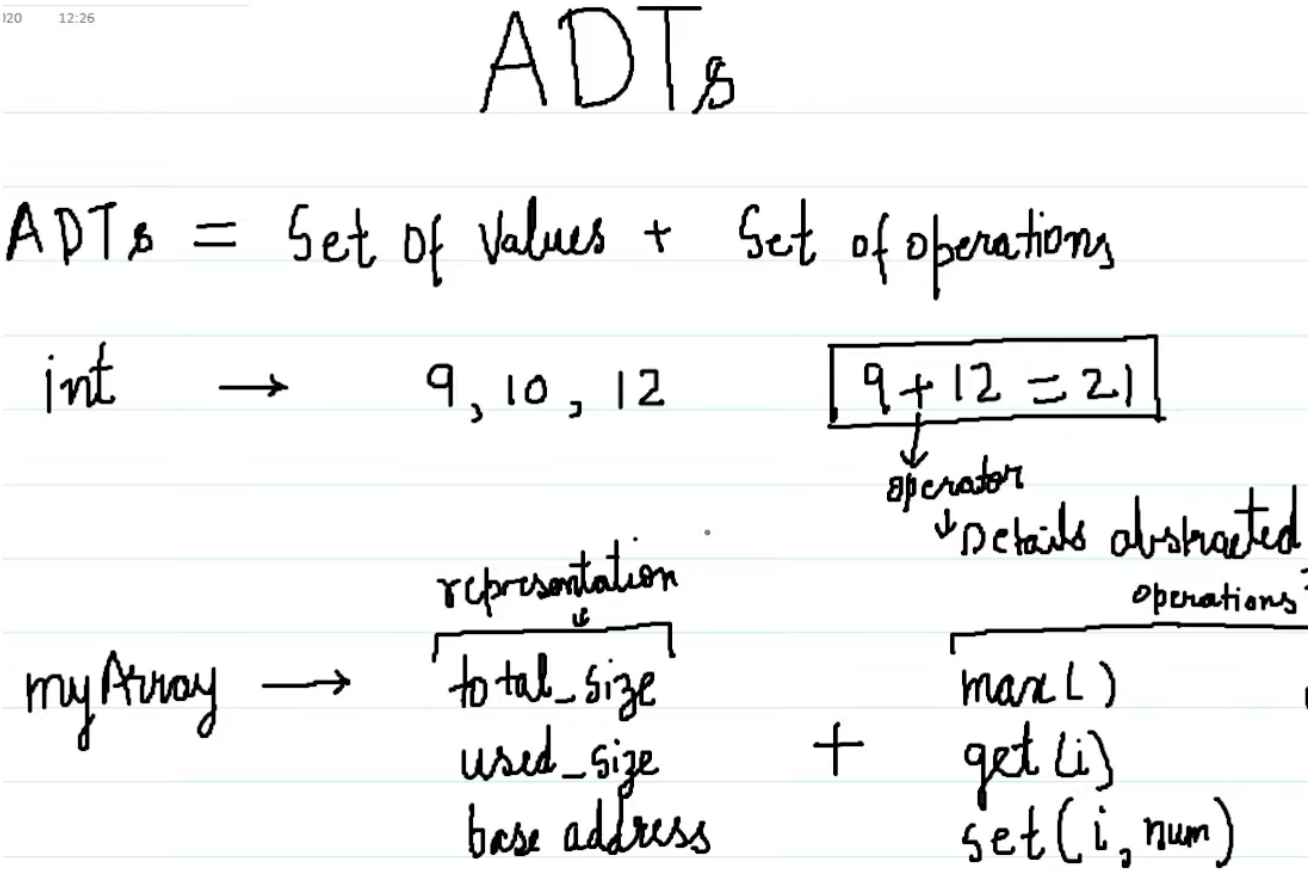
Note: We can connect / relate this concept with OOP concept of Abstraction, Interfaces and Encapsulation.

Examples

Abstraction (ADT)	Implementation (DS)
List	Dynamic Array Linked List
Queue	Linked List based Queue Array based Queue Stack based Queue
Map	Tree Map Hash Map / Hash Table
Vehicle	Golf Cart Bicycle Smart Car

ADT only defines how a data structure should behave and what methods it should have but not the details surrounding how those methods are implemented.

ADTs



Note: ADTs are just another custom data types which have set of values and operations which details can be hidden using concepts like Abstraction, Encapsulation etc.

The Relationship

Layer	What it is	Can you run it?
ADT (Stack)	Concept	No
Data Structure (Array)	Organization method	No
Implementation (Array-based Stack code)	Actual executable thing	Yes

The ADT is implemented USING data structures.

You take the **concept** (Stack), choose a **data structure** (Array or Linked List), write **code** that arranges memory according to that data structure while following the ADT's rules.

	ADT (Concept)	Can be implemented using these Data Structures
Stack	LIFO	Array, Linked List
Queue	FIFO	Array, Linked List
List	Ordered collection	Array, Linked List
Priority Queue	Highest priority first	Heap
Dictionary	Key-value pairs	Hash Table, Binary Search Tree
Set	Unique elements	Hash Table, Binary Search Tree
Graph	Vertices & edges	Adjacency Matrix, Adjacency List
Tree	Hierarchy	Node with pointers

Computational Complexity Analysis:

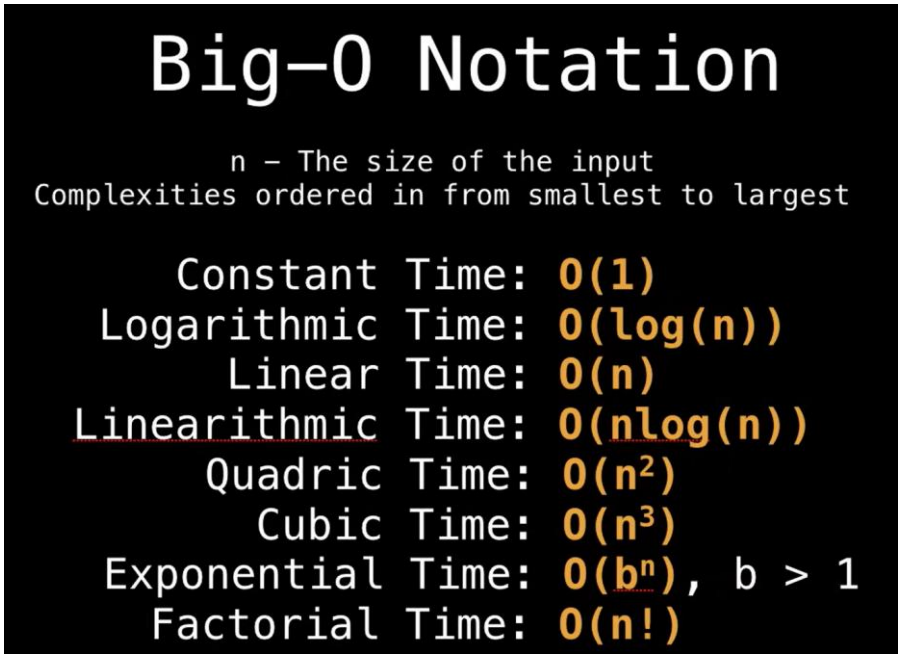
Complexity Analysis:

As programmers, we often and should !, find ourselves asking the same two questions over and over again:

- How much time does this algorithm need to finish?
- How much space does this algorithm need for it's computation?

Big-O Notation:

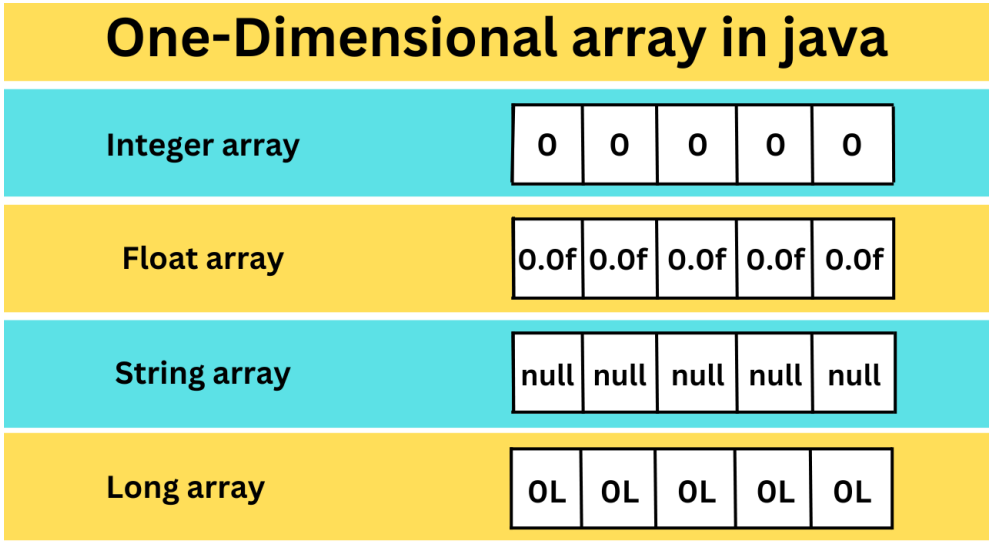
Big-O notation gives an upper bound of the complexity in the worst case, helping to quantify performance as the input size becomes arbitrarily large.



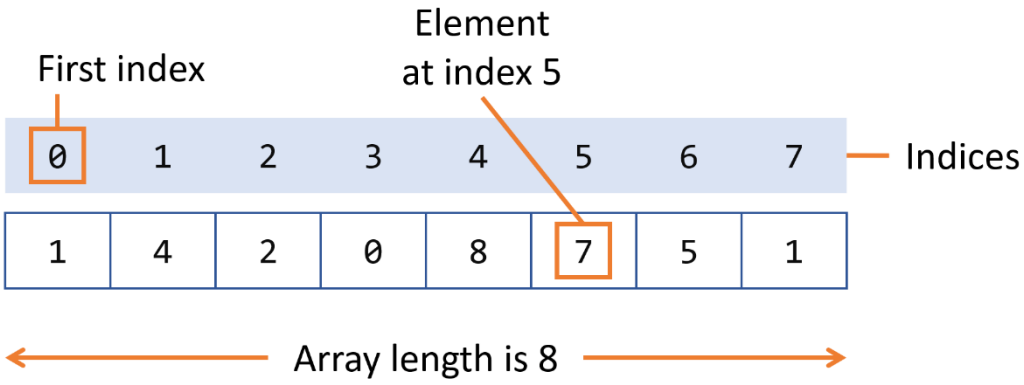
Basics of Array

Array is as data structure which contains similar / homogenous elements, data types or values. Array can of any data types including integer, char, String, float, but have to be of any particular data type but only one, can't contain different data types in an array.

Structure of an Array:



Representation of index and size of an integer array:



In an array these components are included to be in use ; index, data/value, size/length of an array, address of value or index.

Note: The index number in an any array is from 0 to (SizeofArray-1) index.

Declaration of an array:

- **Declare and then create with a fixed size:** This approach creates an array with default values (e.g., `0` for `int`, `null` for `String`, `false` for `boolean`).

```java
int[] numbers; // Declaration
numbers = new int[5]; // Creation/Instantiation with a size of 5
```

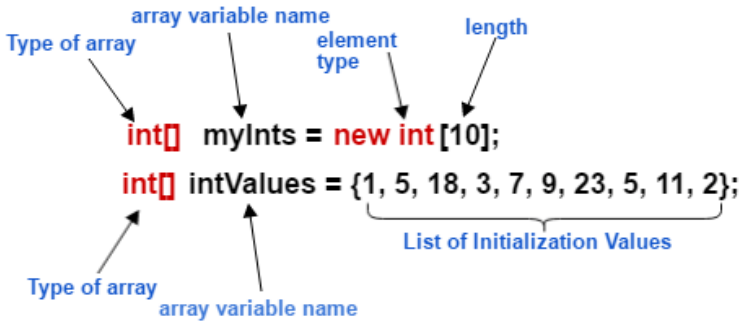
- **Declare first, then initialize with specific values (requires `new`):**

```java
String[] names;
names = new String[]{"John", "Mary", "David"}; // 'new String[]' is required
```

- **Declare and create in one line (fixed size), and manually assigning values :**

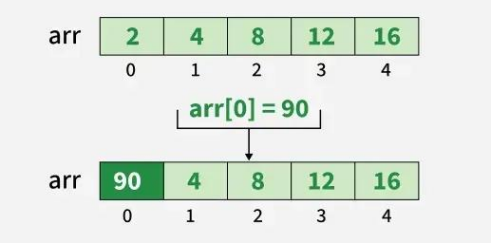
```
int[] nums = new int[4];
nums[0] = 31;
nums[1] = 64;
nums[2] = 11;
nums[3] = 100;
```

Another way is the second line ignore first one because it is we have seen already above, Here The size is automatically determined by the number of values provided.



Note :

The values or data of an array can be change even after assiging of having values declared for particular index in an array.

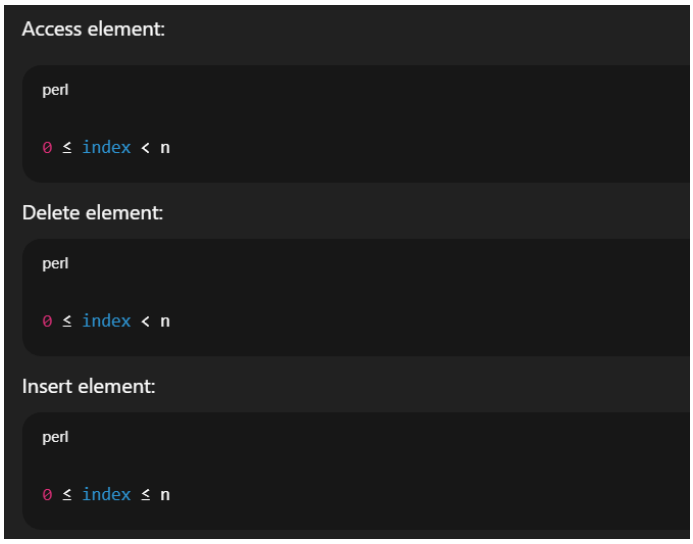


Array as an ADT (Abstract data type):

Here what I meant is instead of traditional method of performing some sort of operation and array creation in a single code why don't just hide those implementation and done all task:

Instead declaring array like: `int[] arr = new int[size];`

Trying to create an array but with different keyword like ARR, or whatever: `ARR[] a = new ARR[size];` n is the number of elements in array, and most important thing to consider is that in this ADT behaving as array we only dealt with the elements present , that's why there is size to track.



Insert deals with gaps → n+1 possibilities
Delete deals with elements → n possibilities

If:

size = 0

There are:

- 0 elements
- 1 insertion position (index 0)
- 0 deletion positions

So: insertAt(0, value) ✓ valid deleteAt(0) ❌ invalid

And that right there proves you now understand the difference between:

- element indexes (0 to size-1) * insertion positions (0 to size)

- # Understanding Array as an Abstract Data Type: The Shift from Structure to Behavior
- The fundamental difference between a raw array and an array implemented as an Abstract Data Type lies not in what they store, but in how they *behave* and *think* about themselves.
- When we declare a raw array using `int[] arr = new int[size];`, we are working directly with memory. The array is just a contiguous block of storage locations, each identified by an index. Every operation—every insertion, every deletion, every search—must be manually implemented in the same code where the array lives. The programmer bears the full burden of tracking the current number of elements, remembering which indices are valid, and ensuring that operations respect the logical state of the collection. The raw array has no awareness of itself; it simply *is*.
- But when we envision an array as an ADT—perhaps created with something like `ARR[] a = new ARR[size];`—we are no longer working with raw memory. We are working with a *concept* that understands itself. This abstracted array knows its own size. It knows the difference between an element position and an insertion gap. It provides operations like `insertAt()` and `deleteAt()` that embody this understanding, and it enforces the rules that make those operations meaningful.
- Consider the case of an empty array. When `size = 0`, there are zero elements but there is exactly one gap—the position at index 0 where an element could be inserted. An ADT-aware array understands this distinction: `insertAt(0, value)` is a perfectly valid operation because a gap exists, while `deleteAt(0)` is invalid because no element occupies that position. This distinction is not merely academic; it reveals that the ADT has internalized the logical structure of a collection. It knows that insertion deals with gaps between elements (hence `size + 1` possible positions), while deletion deals with existing elements (hence `size` possible positions). The raw array knows none of this—it simply allows `arr[0] = value` regardless of whether that assignment constitutes insertion, overwrite, or something else entirely.
- What we are really describing is the difference between *exposing structure* and *providing behavior*. The raw array shows us everything—every index, every memory location—but does nothing for us. The array as an ADT shows us nothing of its internal workings but does everything we need, wrapping the raw structure in a layer of meaning that transforms how we interact with it. We no longer think about memory locations and indices; we think about operations and intentions. And in that transformation lies the true power of abstraction—not merely hiding complexity, but creating a new level of understanding where the data structure itself becomes an active participant in our programs, aware of its own rules and capable of enforcing them.

For more clarification we can see programs or implemented code in java :

https://github.com/Sandesh775/DSA_BCA_Course/blob/main/Data%20Structures/Array/ADT_implementation_in_JAVA/ArrayADT.java

https://github.com/Sandesh775/DSA_BCA_Course/blob/main/Data%20Structures/Array/ADT_implementation_in_JAVA/ArrayADTextended.java

In ArrayADTextended.java :

MyArray2 is **static storage** but **dynamic behavior**. But until upto n-1 size !!!

Think of it like this:

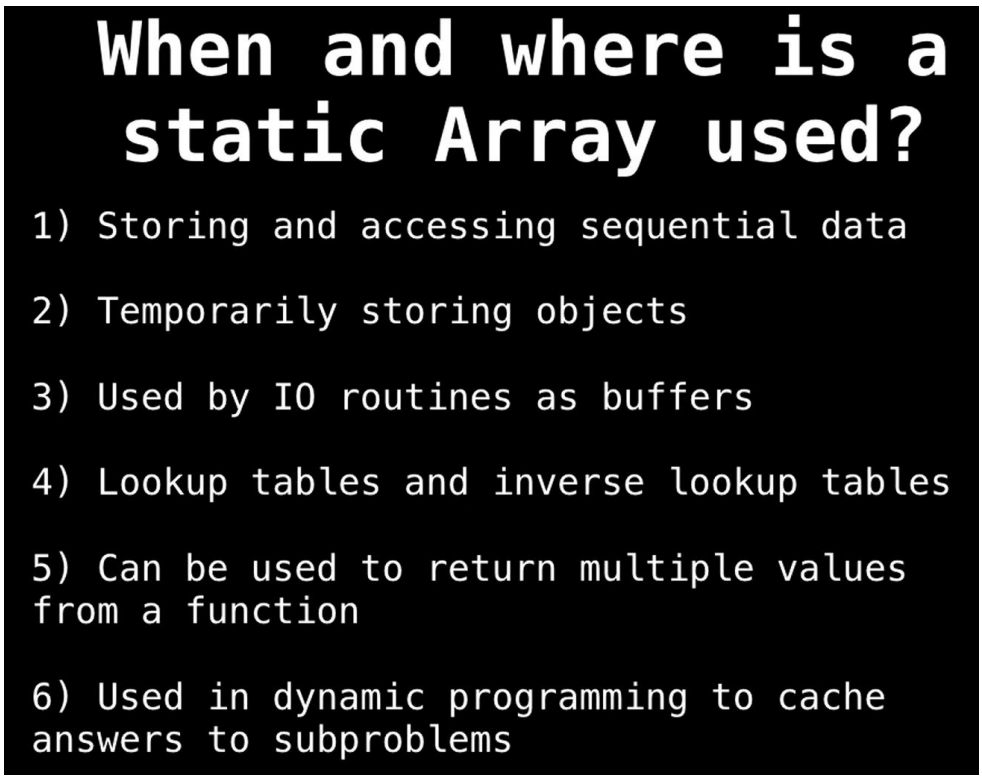
- **Under the hood:** Fixed memory block (static array)
- **In terms of behavior:** Dynamic list that grows and shrinks (until hitting capacity)

The size variable is the **bridge** between static storage and dynamic behavior. It tells the array:

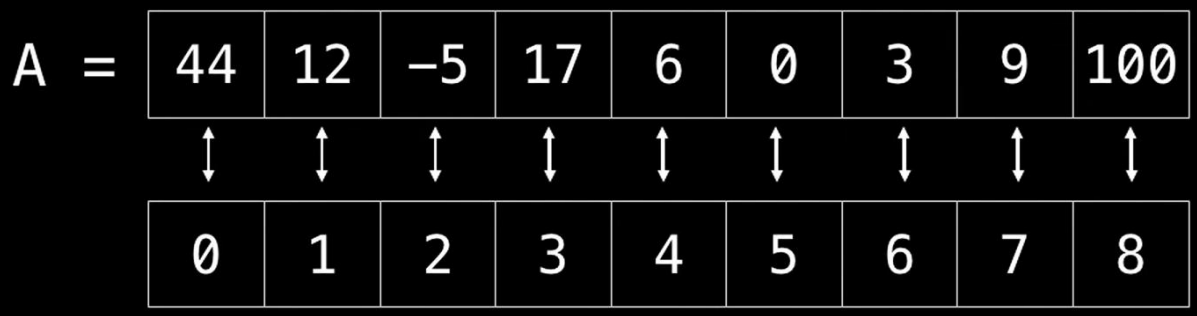
"These indices are actually in use. The rest are just empty space waiting to be filled."

Static and Dynamic Arrays:

A static array is a fixed length container containing n elements indexable from the range [0, n-1].



Static Array



Elements in *A* are referenced by their index. There is no other way to access elements in an array. Array indexing is zero-based, meaning the first element is found in position zero.

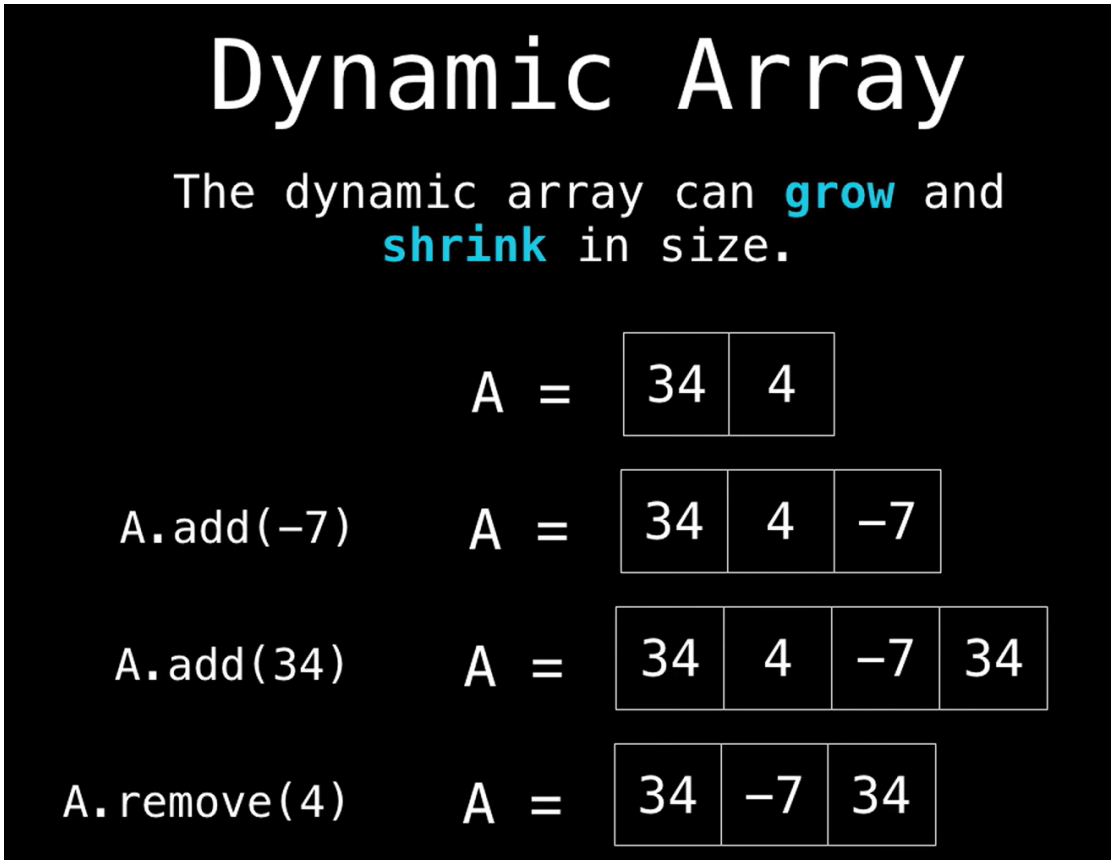
```
A[0] = 44
A[1] = 12
A[4] = 6
A[7] = 9
A[9] => index out of bounds!
```

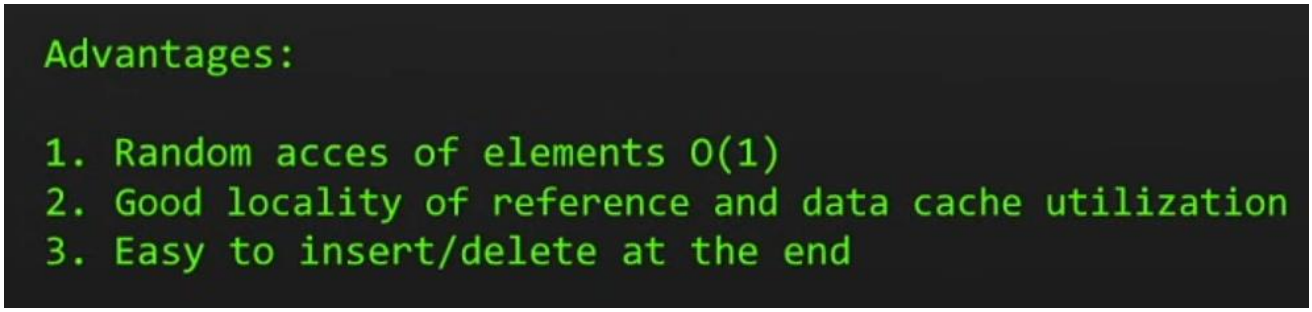
What a Static Array Can Do

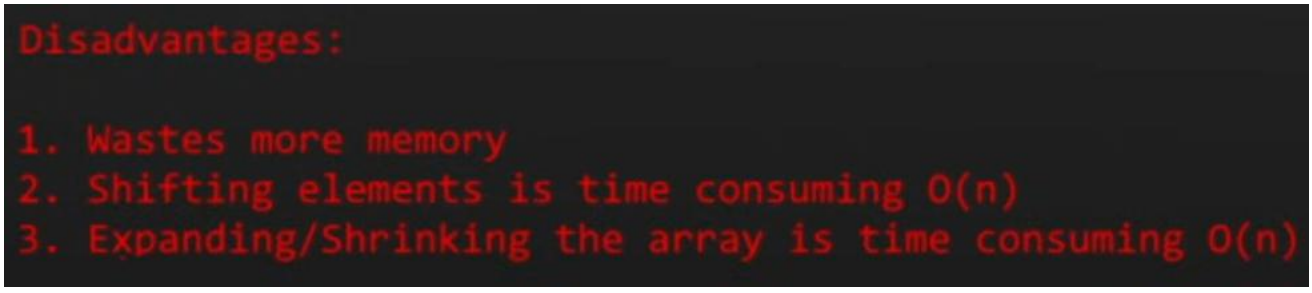
Operation	Syntax	What It Does
Access	`arr[index]`	Read value at that position
Assign	`arr[index] = value`	Write/change value at that position

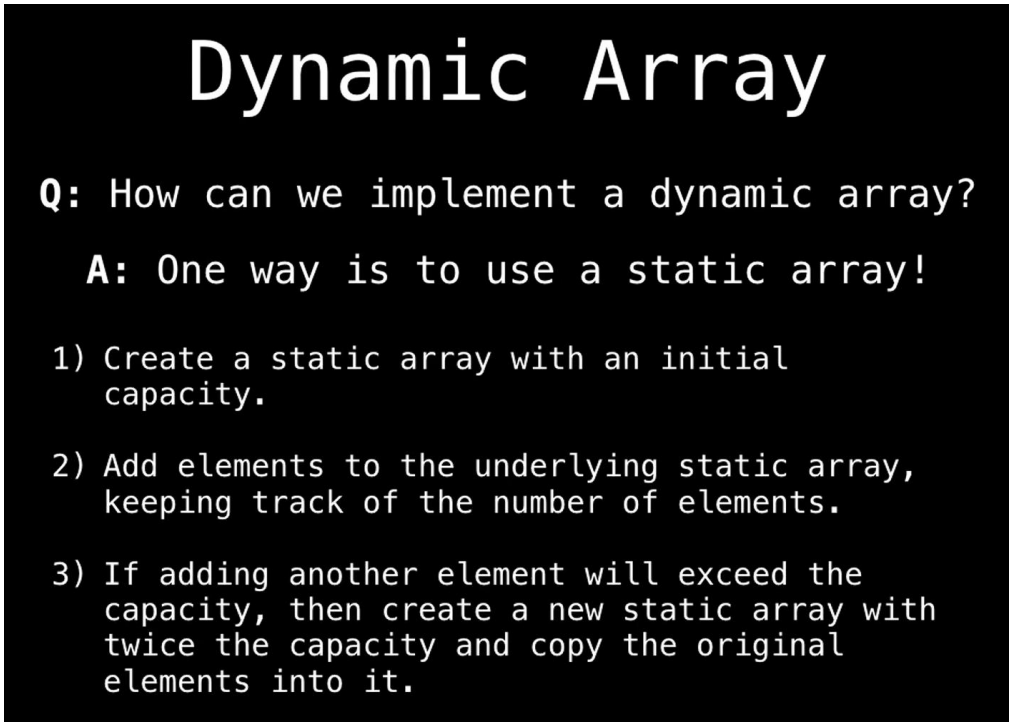
Dynamic Array:

A **dynamic array** is a data structure that provides **array-like access** with the ability to **automatically resize itself** when elements are added or removed, overcoming the fixed-size limitation of static arrays.

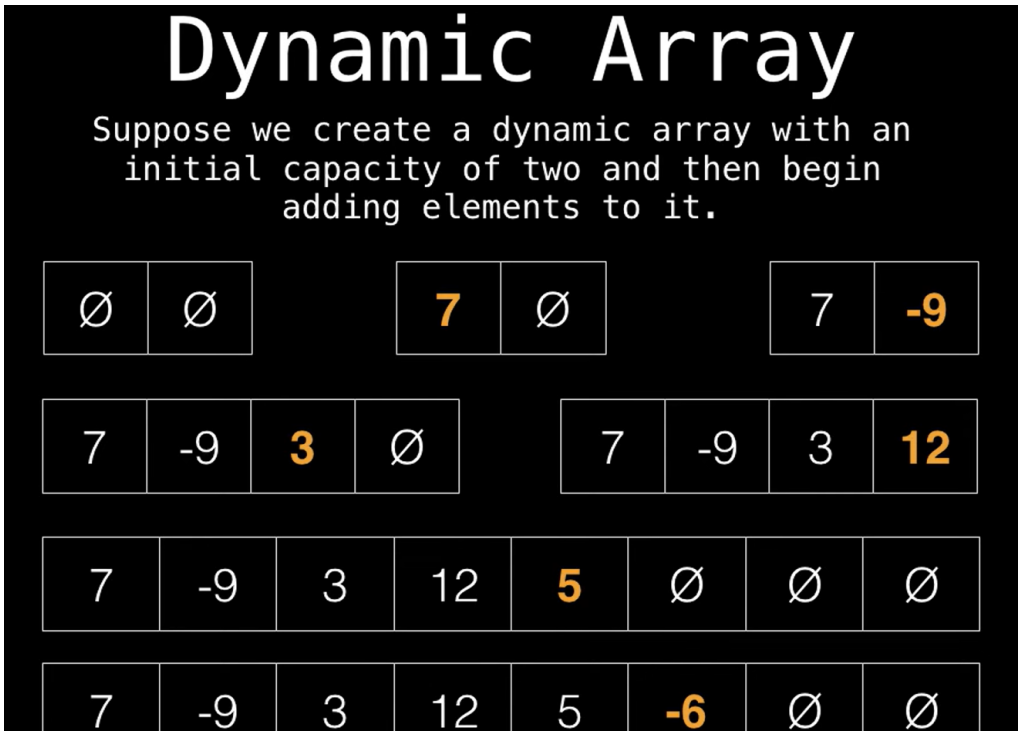








Demonstration in figure example:



The figure illustrates how a dynamic array grows by **allocating a new, larger static array** when capacity is exceeded. Starting with an **initial capacity of 2**, the array holds elements 7 and -9. When 3 is added, there's no space, so a **new array of double capacity (4)** is created, elements are copied over, and 3 is inserted. This pattern repeats: when adding 12 (now at capacity 4), a **new array of capacity 8** is created, all four elements are copied, and 12 is added. Subsequent additions 5 and -6 fit without resizing. This **copy-on-growth** strategy gives dynamic arrays their **amortized O(1)** insertion while maintaining **contiguous memory** for fast index access.

For further more clarification about Dynamic Array check out code here:

https://github.com/Sandesh775/DSA_BCA_Course/blob/main/Data%20Structures/Array/ADT_implementation_in_JAVA/Dynamic_Array_Demo.java

https://github.com/Sandesh775/DSA_BCA_Course/blob/main/Data%20Structures/Array/ADT_implementation_in_JAVA/DynamicArrayUsingStaticArrayDemo.java