Arrays

Types of Data Structures

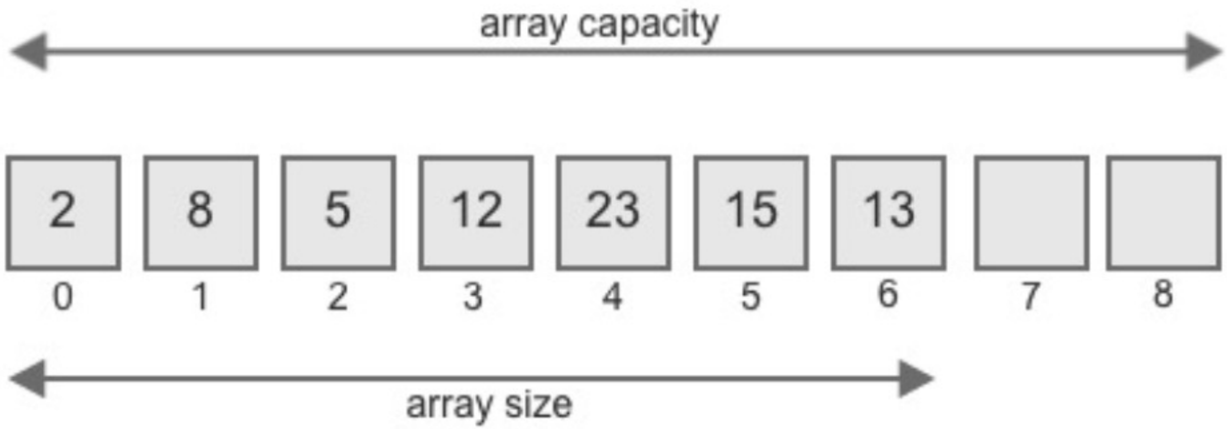
- Depending on the organization of the elements, data structures are classified into two types:
  1. **Linear data structures**: Elements are arranged in a sequential order like a straight line (or a column) but it is not compulsory to store all elements sequentially (say, Linked Lists). Examples: Arrays, Linked Lists, Stacks, and Queues.
  2. **Non–linear data structures**: Elements are not in a sequential order, but rather have a more complex relationship. They can be hierarchical (like a tree) or involve connections between elements (like a graph). Examples: Trees and graphs.

Array

- An array is a collection of elements stored in a contiguous block of memory.
- **Analogy**: Think of an array as a row of lockers, each with a unique number (index), where you can store items (data).

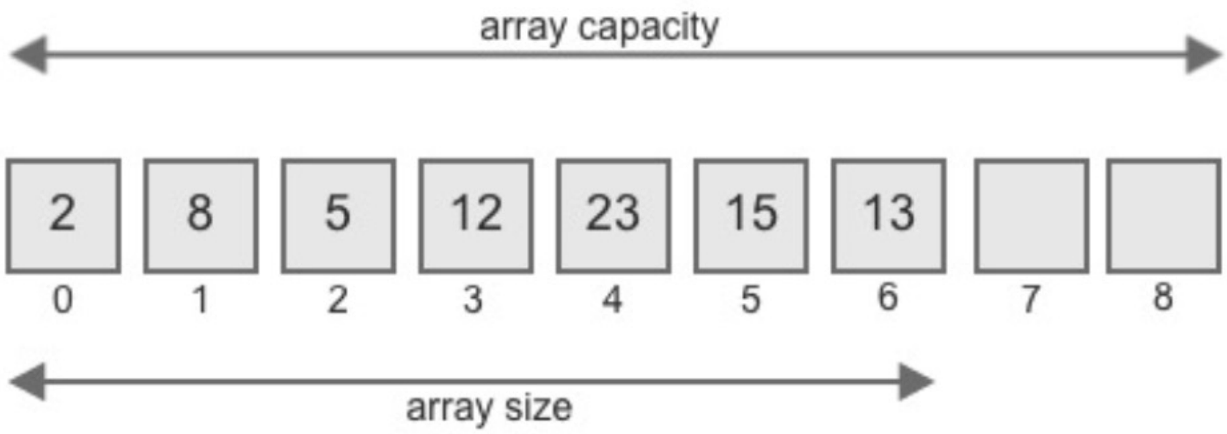
Array Properties

- **Fixed size**: Once defined, the size of an array cannot change.
- **Indexed**: Individual elements are accessed using indices (an index is an integer that usually starts from 0 and goes to $n - 1$, where n is the array's total capacity).



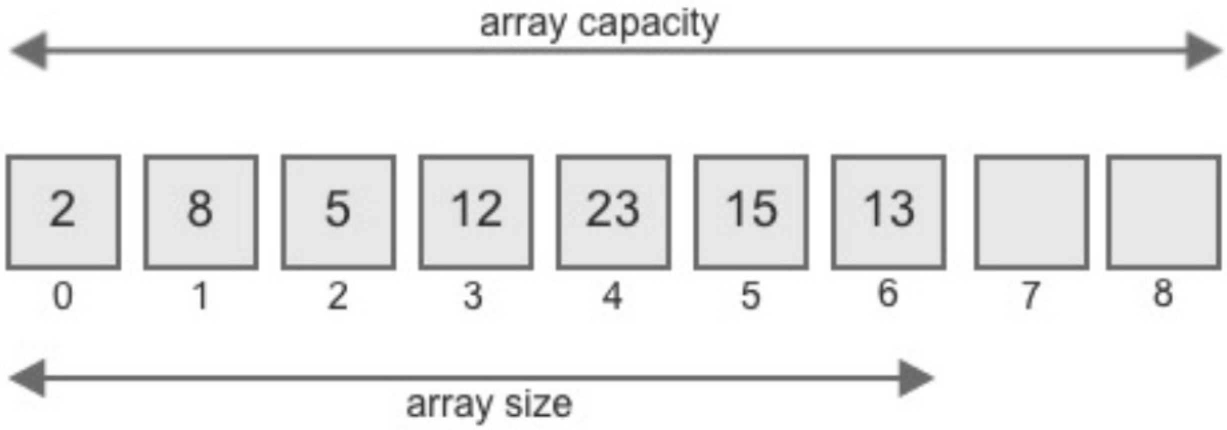
Array Size

- **Size**: The size of an array refers to the number of elements that are currently stored in the array.
- **Example**: In the figure below, there are 7 elements stored in the array. Thus, its size is 7.



Array Capacity

- **Capacity**: The capacity of an array refers to the total amount of space that has been allocated for the array. It indicates the maximum number of elements the array can hold before needing to resize.
- **Example**: In the figure below, the array capacity is 9.



Advantages of Array

- **Efficient Random Index-Based Access**:
  - Accessing any element by its index is very fast, taking constant time (O(1)).
  - **Analogy**: Just like you can quickly open any locker by its number without having to check each one sequentially.
- **Memory Efficiency due to Contiguous Memory Allocation**:
  - All elements are stored together in memory, reducing overhead and improving cache performance.

Accessing any element in O(1)

- An array's elements are stored in contiguous (continuous) memory locations.
- This means that once you know the starting address of the array, you can calculate the address of any element using simple arithmetic.

Accessing any element in O(1)

- **Address calculation**
- Suppose
  - The base (or starting) address of the array is $\mathbf{B}\boldsymbol{A}$.
  - The size of each element is $\boldsymbol{S}$ (in bytes).
  - The index of the element we want to access is $\boldsymbol{i}$.

  - The memory address of the ith element can be calculated using the formula:

$$\boldsymbol{Address\ of\ element\ at\ index\ i\ = BA\ +\ (i\ *\ S)}$$

Accessing any element in O(1)

- Example:
- Consider an integer array **arr** where each integer takes 4 bytes (on most systems):
- If the base address of **arr** is 1000 and you want to access the element at index 3, the address would be calculated as:

$$\boldsymbol{Address\ of\ arr[3] = 1000 + (3 * 4) = 1000 + 12 = 1012}$$

Accessing any element in O(1)

- The calculation $\mathbf{B}\boldsymbol{A} + (\boldsymbol{i} * \boldsymbol{S})$ involves only a few basic operations: addition and multiplication, both of which are performed in constant time.
- No matter how large the array is, the number of operations needed to calculate the address of any element remains the same.

1D and 2D Arrays

- Arrays can also be classified as one-dimensional (1D) or two-dimensional (2D).

1D Arrays

- A one-dimensional array is a linear data structure that stores a sequence of elements in a single row (or column).
- Use when you need to store a list of items like grades, prices, temperatures, etc.

1D Array Memory Representation

- In memory, a 1D array is represented as a contiguous block of elements.
- The memory address of the ith element can be calculated using the formula:

$$\boldsymbol{Address\ of\ element\ at\ index\ i\ = BA + (i * S)}$$

- Where **BA** is the base address and **S** is size of each element

2D Arrays

- A two-dimensional array is a grid or matrix-like data structure that stores elements in rows and columns.
- Suitable for mathematical and scientific computations that require matrix representations.
- Like for representing tabular data like a spreadsheet or database table or grid-based applications such as image processing, games (like chess boards), or geographical data.

2D Array Memory Representation

- 2D arrays are also stored in a contiguous block of memory.
- This means that all elements are placed next to each other, without any gaps.
- For a 2D array, this can be done in two common ways:
  1. Row-major order
  2. Column-major order

Row-Major Order (More Common)

- In most languages like C and C++, elements are stored in row-major order.
- In row-major order, elements are stored row by row.
- Suppose we have the following 2D array

1	2	3	4
5	6	7	8
9	10	11	12

Row-Major Order

1
2
3
4
5
6
7
8
9
10
11
12

1	2	3	4
5	6	7	8
9	10	11	12

Row-Major Order

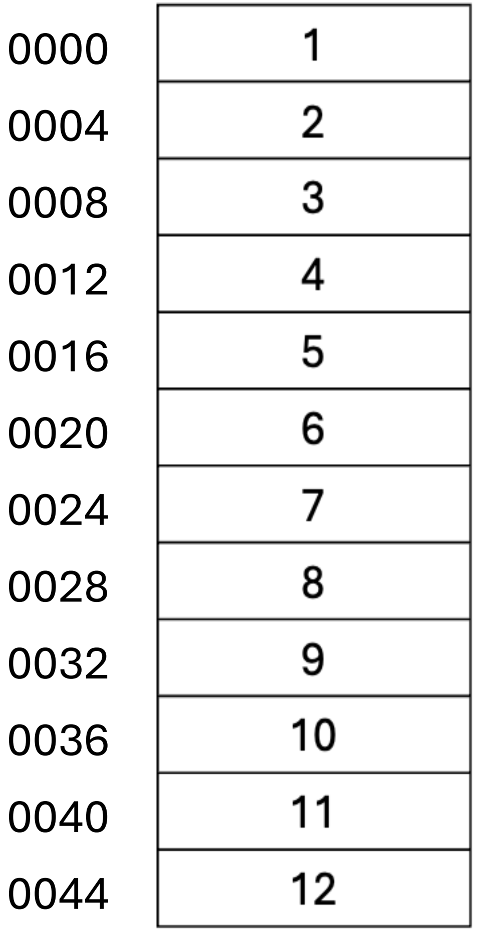
- To access an element in a 2D array stored in row-major order, you use the formula:

$$index = (rowNum \times numOfColumns + columnNo) \times size$$

Row-Major Order

- For example, if the integer size is 4 bytes, the starting address is 0000 and we want to access the element at index [1][2] i.e., row 1, column 2 (the value is 7)
- Then, the address would be:

index = (1*4 + 2) * 4 = (4+2)*4 = 24



0000	1
0004	2
0008	3
0012	4
0016	5
0020	6
0024	7
0028	8
0032	9
0036	10
0040	11
0044	12

Column-Major Order

1
5
9
2
6
10
3
7
11
4
8
12

1	2	3	4
5	6	7	8
9	10	11	12

In column-major order, elements are stored column by column

Column-Major Order

- To access an element in a 2D array stored in column-major order, you use the formula:

$$index = (columnNum \times numOfRows + rowNum) \times size$$

Column-Major Order

- Again, if the integer size is 4 bytes, the starting address is 0000 and we want to access the element at index [1][2] i.e., row 1, column 2 (the value is 7)
- Then, the address would be:

index = (2*3 + 1) * 4 = (6+1)*4 = 28

0000	1
0004	5
0008	9
0012	2
0016	6
0020	10
0024	3
0028	7
0032	11
0036	4
0040	8
0044	12

Unordered vs Ordered Arrays

- Arrays can also be classified as either unordered (unsorted) or ordered (sorted).

Unordered Arrays

- An unordered array is a collection of elements stored in no particular sequence.
- New elements are typically added to the end of the array, making the insertion operation straightforward.
- Pros
    - **Fast Insertion (and Deletion at the end)**: Inserting an element at the end of the array is very efficient, typically $O(1)$ time complexity. Besides, deletion from the end is also efficient ($O(1)$)
    - **Simplicity**: Easier to implement as there is no need to maintain any order.

Unordered Arrays

- Cons
  - **Slow Search**: Searching for an element requires a linear search, which has $O(n)$ time complexity in the worst case as you need to compare with every element.
  - **Slow Deletion (not from the end)**: Finding the element to delete (from the beginning or middle) takes $O(n)$, and then you may need to shift elements to fill the gap, which also takes $O(n)$ time.

Unordered Arrays

- When to use
  - When insertion speed is critical and the order of elements does not matter.
  - For small datasets where the search and deletion time being $O(n)$ is acceptable.
  - When performing operations where the order is irrelevant, such as in simple storage or logging systems.

Unordered Arrays – Time Complexities

- **Search:** $O(n)$

	Add	Remove
Beginning	$O(n)$	$O(n)$
End	$O(1)$	$O(1)$
Middle	$O(n)$	$O(n)$

Ordered Arrays

- An ordered array is a collection of elements arranged in a specific sequence, typically sorted in ascending or descending order.
- Pros
    - **Fast Search**: Binary search can be used, providing $O(\log n)$ time complexity for search operations.
    - **Ordered Data**: Easier to find the minimum or maximum values directly from the array.

Ordered Arrays

- Cons
  - **Slower Insertion**: Maintaining the order requires finding the correct position and shifting elements, which can take $O(n)$ time in the worst case.
  - **Slower Deletion**: Finding the element to delete and then shifting elements to fill the gap can also take $O(n)$ time.

Ordered Arrays

- When to use
  - When search speed is critical and the dataset is relatively static with infrequent insertions and deletions.
  - For large datasets where fast searching is more important than fast insertion or deletion.
  - When the data needs to be processed in order frequently, such as in priority queues or maintaining a sorted list of records.

Ordered Arrays – Time Complexities

- **Search:** $O(log\ n)$ using binary search

	Add	Remove
Beginning	$O(n)$	$O(n)$
End	$O(1)$ [or $O(log\ n)$]	$O(1)$ [or $O(log\ n)$]
Middle	$O(n)$	$O(n)$

Static vs Dynamic Arrays

- Arrays can also be classified as static or dynamic

Static Arrays

- Static arrays have a fixed size that cannot be changed during the program's execution.
- The fixed size can be a limitation. If you underestimate the size, you may run out of space. If you overestimate, you waste memory.

Dynamic Arrays

- Dynamic arrays (sometimes also called as array lists or vectors) are flexible and their size can be adjusted at runtime, allowing the array to grow or shrink as needed.
- Resizing (especially growing) a dynamic array can be expensive because it may involve allocating new memory and copying elements from the old array to the new one.