Understanding Arrays in Data Structures

Mayank Pratap Singh

July 12, 2024

Introduction to Arrays

An **array** is a data structure consisting of a collection of elements (values or variables), each identified by at least one array index or key. Arrays are among the oldest and most important data structures, and they are used in various applications.

Array Representation in Memory

An array is stored in a **contiguous block of memory**. This means that each element of the array is stored sequentially in memory.

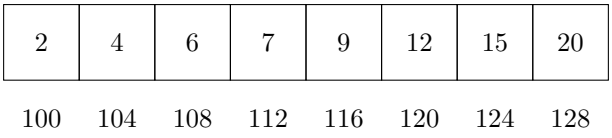


Figure 1: Representation of an array in memory

Base Address and Memory Address Calculation

Each element in the array is accessed via its **memory address**. The memory address of an element in an array can be calculated using the formula:

$$\text{Address} = \text{Base Address} + (i \times \text{size of element})$$

where i is the index of the element, and the **size of the element** depends on the data type. For an integer array, if the size of an integer is 4 bytes, then:

$$\text{Address of element at index } 0 = 100 + (0 \times 4) = 100$$
$$\text{Address of element at index } 1 = 100 + (1 \times 4) = 104$$
$$\text{Address of element at index } 2 = 100 + (2 \times 4) = 108$$

Properties of Arrays

- **Fixed Size:** The size of an array is fixed and must be defined at the time of array declaration.
- **Random Access:** Arrays allow random access to elements using their indices.
- **Homogeneous Elements:** Arrays store elements of the same data type.

Advantages of Arrays

- **Ease of Access:** Elements can be accessed randomly using the index number.
- **Memory Allocation:** Arrays use a contiguous block of memory.

Disadvantages of Arrays

- **Fixed Size:** The size of the array cannot be altered once declared.
- **Inefficient Insertion/Deletion:** Inserting or deleting elements can be inefficient as it requires shifting elements.

Accessing Elements in an Array

We can access elements in an array using their indices. For example, if the name of the array is `arr`:

$$\texttt{arr[5]} = 12$$
$$\texttt{arr[7]} = 20$$

The base address of an array refers to the address of the first element in the array.

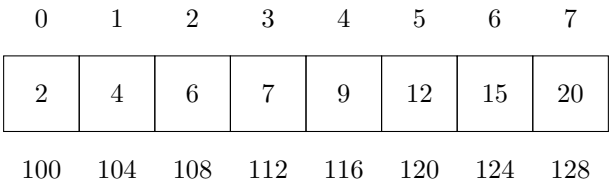


Figure 2: Accessing elements in an array

Time Complexity of Accessing Elements in an Array

Accessing an element in an array using its index is very efficient. The time complexity for this operation is **O(1)**, which is constant time. This means that no matter the size of the array, accessing an element by its index will always take the same amount of time.

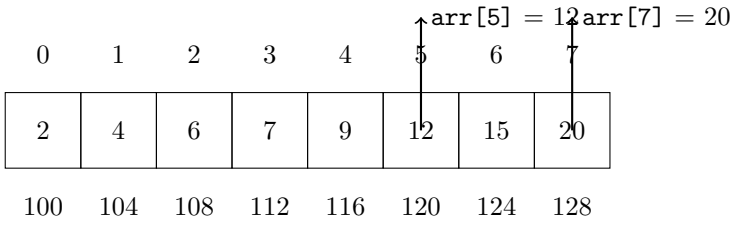


Figure: Array indices, memory addresses, and access examples

Why is the Time Complexity O(1)?

When you access an element in an array, you use its index to directly calculate its memory address using the formula:

$$\text{Address} = \text{Base Address} + (i \times \text{size of element})$$

This calculation is done in constant time because it involves a simple arithmetic operation, regardless of the size of the array.

$$\begin{aligned}
&\text{Address of element at index } 0 = 100 + (0 \times 4) = 100 \\
&\text{Address of element at index } 1 = 100 + (1 \times 4) = 104 \\
&\text{Address of element at index } 2 = 100 + (2 \times 4) = 108 \\
&\text{Address of element at index } 5 = 100 + (5 \times 4) = 120 \\
&\text{Address of element at index } 7 = 100 + (7 \times 4) = 128
\end{aligned}$$

Constant Time Complexity

Since this calculation is independent of the array size n, accessing any element by index will always take the same amount of time. Therefore, the time complexity is $\mathbf{O(1)}$.

Homogeneous vs. Heterogeneous Arrays

Arrays can be classified into **homogeneous** and **heterogeneous** based on the type of elements they store.

Homogeneous Arrays

Homogeneous arrays store elements of the same data type. This is the traditional form of arrays used in many programming languages.

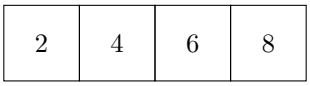


Figure: Homogeneous Array (same data type)

Heterogeneous Arrays

Heterogeneous arrays can store elements of different data types. In Python, lists can be used as heterogeneous arrays.

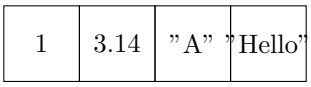


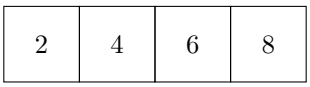
Figure: Heterogeneous Array (different data types)

Static vs. Dynamic Arrays

Arrays can also be classified into **static** and **dynamic** based on their size flexibility.

Static Arrays

Static arrays have a fixed size, which must be declared at the time of array creation. They are not flexible in terms of size, and attempting to access an index out of bounds will result in an error.



Static Array

Fixed Size

Index Out of Bounds Error

Dynamic Arrays

Dynamic arrays can change their size at runtime, allowing for more flexibility. Elements can be appended or removed from the array as needed.

In Python, a dynamic array can be represented using a list, which allows for adding elements at runtime.

1	3.14	”A”	”Hello”	10

Dynamic Array
Flexible Size
Can Change at Runtime

Summary

- **Homogeneous Arrays:** Store elements of the same data type.
- **Heterogeneous Arrays:** Store elements of different data types, like Python lists.
- **Static Arrays:** Have a fixed size and can result in an index out of bounds error.
- **Dynamic Arrays:** Have flexible size and can change during runtime.

Understanding the differences between these types of arrays helps in choosing the appropriate data structure for different applications.

Static vs. Dynamic Arrays: Which is Better?

The choice between static and dynamic arrays depends on the specific requirements of the application.

Static Arrays

Advantages:

- **Fixed Size:** Since the size is fixed, memory allocation is straightforward and efficient.
- **Performance:** Accessing and manipulating elements in static arrays can be faster compared to dynamic arrays.
- **Predictable:** The fixed size ensures that the array's memory usage is predictable.

Disadvantages:

- **Inflexibility:** The size of the array cannot be changed once it is defined.
- **Index Out of Bounds Error:** Attempting to access an index outside the defined range will result in an error.

Dynamic Arrays

Advantages:

- **Flexible Size:** The size of the array can be changed at runtime, allowing for dynamic addition and removal of elements.
- **Convenient:** They are more convenient to use when the number of elements is not known in advance.

Disadvantages:

- **Overhead:** Dynamic arrays may incur overhead due to resizing operations.
- **Performance:** Accessing and manipulating elements can be slower compared to static arrays.

Recommendation

We should use **static arrays** when the size of the array is known and fixed. This is because static arrays offer better performance due to their predictable memory allocation and faster access times.

For example, if someone is writing similar logic in C or Java, they might use static arrays, which would be faster compared to Python's dynamic arrays. This is because C and Java are statically typed languages, and their static arrays can be more efficient than Python's dynamic arrays, which offer more flexibility but at the cost of performance.

In summary, for performance-critical applications where the size of the array is known in advance, static arrays are preferable. However, for applications requiring flexibility and where the number of elements is not known beforehand, dynamic arrays are more suitable.

Dynamic Arrays and Load Factor

Dynamic arrays are designed to provide flexibility by allowing their size to change at runtime. This flexibility, however, comes at a cost of performance due to the internal operations required for resizing.

Internal Working of Dynamic Arrays

When a dynamic array reaches its capacity, it undergoes a resizing operation. The resizing process involves creating a new array of double the size and copying the existing elements into the new array. Here's a step-by-step breakdown:

1. **Creation of New Array:** A new array is created with double the size of the current array.

2. **Copying Elements:** All elements from the current array are copied to the new array.
3. **Adding New Element:** The new element is then added to the newly created array at the next available position.

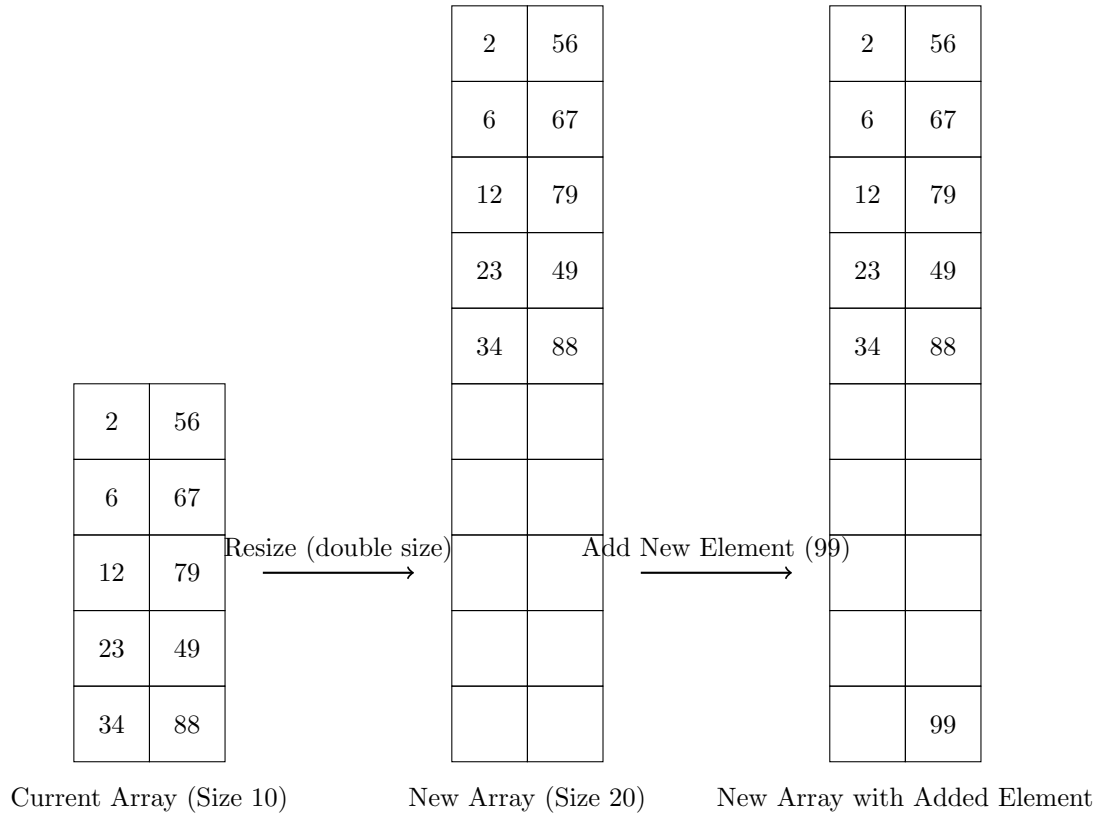


Figure 1: Steps involved in resizing a dynamic array and adding a new element.

Impact on Performance

This resizing operation introduces a delay in response time due to:

1. The creation of a new array with double the size.
2. The copying of all elements from the old array to the new array.

Load Factor

The concept of **load factor** is used to determine when to resize the dynamic array. The load factor is the ratio of the number of elements to the size of

the array. When the load factor exceeds a certain threshold (typically 0.8), the resizing process is triggered.

$$\text{Load Factor} = \frac{\text{Number of Elements}}{\text{Size of Array}}$$

If the load factor exceeds 0.8, the array size is doubled to accommodate more elements. This helps in maintaining efficient performance for dynamic array operations.

Reversal of a List

Reversing a list is a common interview question that can be solved efficiently using the two-pointer approach. Given an array with elements, the goal is to reverse the array.

Example

Input: `[2, 2, 7, 9, 12, 4]`
Output: `[4, 12, 9, 7, 2, 2]`

Negative Indexing in Python

Python supports negative indexing, which allows you to access elements from the end of the list. For example, `list[-1]` will access the last element of the list.

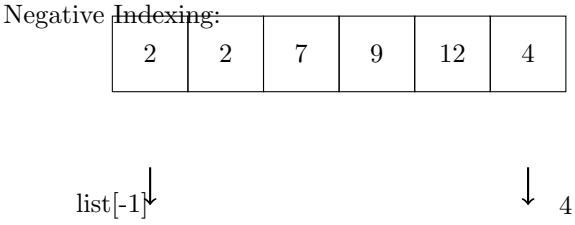


Figure 2: Accessing the last element using negative indexing.

Example:
If we have `list2 = [2, 1, 7, 9, 12, 4]`, using `list2[::-1]` will reverse the list:

```
list2 = [2, 1, 7, 9, 12, 4]
result = list2[::-1]
print(result)
```

Original Array:

2	1	7	9	12	4

Input: [2, 1, 7, 9, 12, 4]

Reversed Array:

4	12	9	7	1	2

Output: [4, 12, 9, 7, 1, 2]

Figure 3: Reversing a list using slicing with negative step.

This will output:

```
[4, 12, 9, 7, 1, 2]
```

Slicing with Steps:
You can use slicing to skip elements by specifying a step. For example, `list2[::-2]` will reverse the list and skip every second element:

```
list2 = [2, 1, 7, 9, 12, 4]
result = list2[::-2]
print(result)
```

This will output:

```
[4, 9, 2]
```

Using the slicing technique, you can manipulate lists in various ways, such as reversing them or skipping elements.

Original Array:

2	1	7	9	12	4

Input: [2, 1, 7, 9, 12, 4]

Sliced Array:

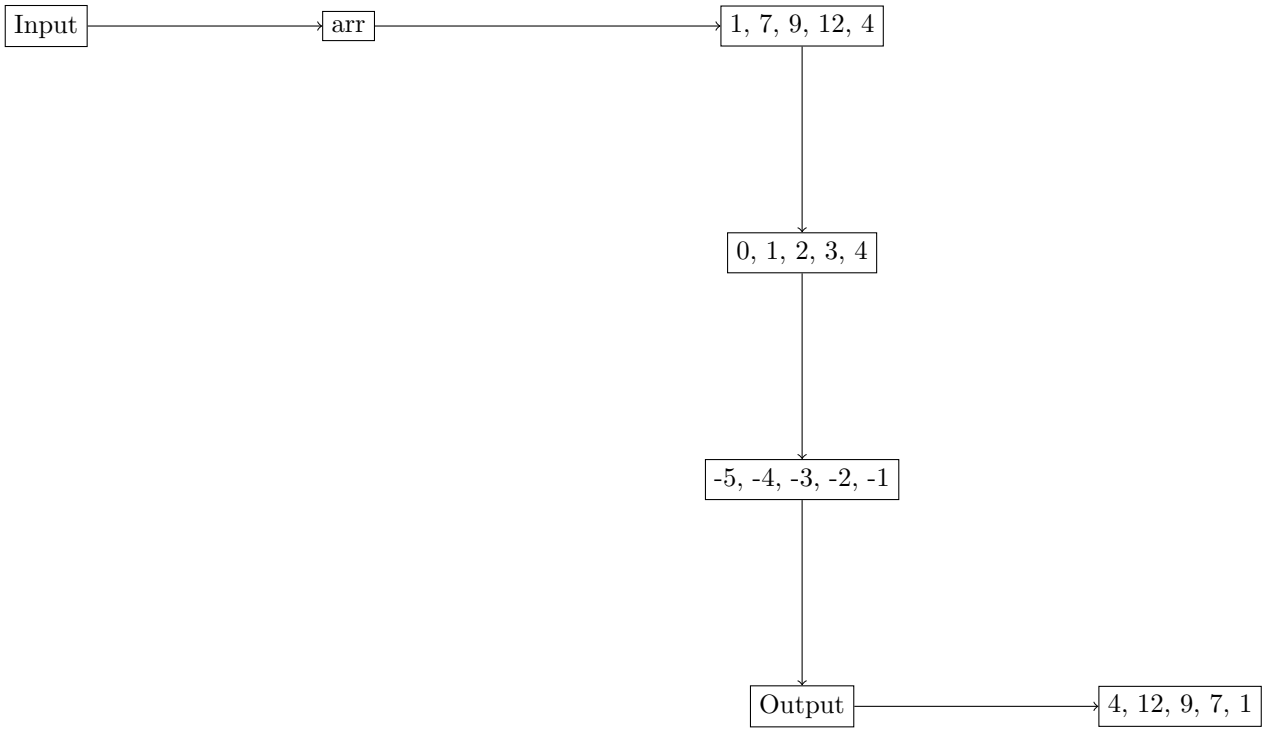
4	9	2

Output: [4, 9, 2]

Figure 4: Using slicing with steps to reverse a list and skip elements.

Reversal of List

Slicing Operation



Using Slicing for Reversal

The array can be reversed using the slicing operation:

```
arr[::-1]
```

Example:

```
arr = [1, 7, 9, 12, 4]
reversed_arr = arr[::-1]
print(reversed_arr)  # Output: [4, 12, 9, 7, 1]
```

Time and Space Complexity

- **Time Complexity:** The time complexity of slicing operation for reversal is $O(n)$ as it iterates through the entire list once.
- **Space Complexity:** The space complexity of slicing operation for reversal is $O(1)$ if we do not consider the space required for the output list, otherwise it is $O(n)$.

Alternative Approach: Using a Loop

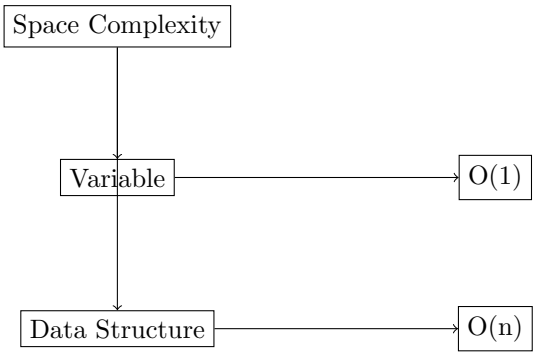
```
def reverse(arr):
    ans = []
    for idx in range(1, len(arr) + 1):
        ans.append(arr[-idx])
    return ans

arr = [1, 4, 6, 8, 11]
result = reverse(arr)
print(result)  # Output: [11, 8, 6, 4, 1]
```

- **Time Complexity:** The time complexity of this loop-based reversal is $O(n)$.
- **Space Complexity:** The space complexity is $O(n)$ due to the additional list used to store the reversed elements.

Understanding Space Complexity

- **Definition:** Space complexity refers to the amount of extra space or memory used by an algorithm.
- **Components:**
  - **Variables:** Using a constant number of variables results in $O(1)$ space complexity.

– **Data Structures:** Using additional data structures (like lists, arrays, etc.) results in $O(n)$ space complexity.



Two-Pointer Approach

The two-pointer approach involves using two indices that start at the beginning and end of the list, respectively, and move towards the center, swapping elements as they go.

Original Array:

2	2	7	9	12	4

Input: [2, 2, 7, 9, 12, 4]

Reversed Array:

4	12	9	7	2	2

Output: [4, 12, 9, 7, 2, 2]

Figure 5: Reversing a list using the two-pointer approach.

```
def reverse_list(arr):
    left = 0
    right = len(arr) - 1
    while left < right:
        arr[left], arr[right] = arr[right], arr[left]
        left += 1
        right -= 1
    return arr

arr = [2, 2, 7, 9, 12, 4]
result = reverse_list(arr)
print(result)
```

Let's say we have an array:

$$[2, 7, 12, 19, 21, 27, 52]$$

We will use two pointers: - i starts from 0 - j starts from $n - 1$

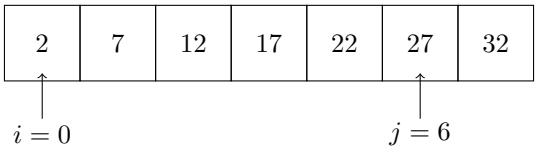


Figure 6: Initial positions of pointers i and j.

We will swap the values of $arr[i]$ and $arr[j]$ until $i < j$.
We increment i and decrement j after each swap:

$$i = i + 1$$
$$j = j - 1$$

Time Complexity: $O(n)$
Space Complexity: $O(1)$

[language=Python, caption=Python code for reversing an array using two-pointer approach, frame=single] def reverse(arr): n = len(arr) i = 0 j = n - 1

while i ¡ j: swap the elements between arr[i] and arr[j] arr[i], arr[j] = arr[j], arr[i] i += 1 j -= 1

return arr

Example with Even Number of Elements

Consider the array:

$$[2, 5, 8, 12, 14, 19]$$

Using the same two-pointer approach, we swap the elements as follows:

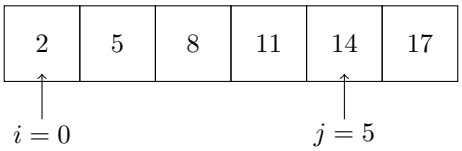


Figure 7: Array with even number of elements before swapping.

After swapping, the array becomes:

$$[19, 14, 12, 8, 5, 2]$$

1 Array Operations

1.1 Accessing Elements

To access an element in an array, use its index. The time complexity for this operation is $O(1)$.

$$\text{arr}[i]$$

For a 2D array, elements are accessed using two indices: row and column.

$$\text{arr}[i][j]$$

1.2 Searching

To search for an element in an array, use a loop to iterate through the elements and compare each one with the target value.

$$\text{arr} = [2, 4, 9, 12, 15]$$

$$\text{target} = 12 \Rightarrow \text{output} = 3$$

$$\text{target} = 17 \Rightarrow \text{output} = -1 \text{ (element not found)}$$

```
int search(int arr[], int n, int target) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == target) {
            return i;
        }
    }
    return -1;
}
```

1.3 Insertion and Deletion

Insertion and deletion operations in arrays require shifting elements, leading to a time complexity of $O(n)$. For frequent insertions and deletions, linked lists are more efficient.

$$\text{arr} = [2, 7, 9, 12, 15]$$

$$\text{Insert 14 at index 2}$$

$$\text{arr} = [2, 7, 14, 9, 12, 15]$$

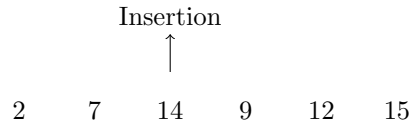


Figure 8: Array After Insertion

1.4 Sorting

Sorting arranges elements in a specific order, either ascending or descending. Sorting algorithms can be classified into two types:

- **Inplace Sorting:** Does not require extra space. Example: Quick Sort.
- **Outplace Sorting:** Requires extra space. Example: Merge Sort.

$$\text{arr} = [2, 1, 5, 3, 9]$$
$$\text{sorted} = [1, 2, 3, 5, 9]$$

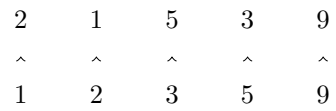


Figure 9: Array Before and After Sorting

2 2D Arrays

2D arrays are used to store data in a tabular format with rows and columns.

$$\begin{bmatrix} 23 & 47 & 65 \\ 46 & 24 & 99 \\ 29 & 32 & 42 \end{bmatrix}$$

To access the element at the second row and first column:

$$\text{arr}[2][1] = 32$$

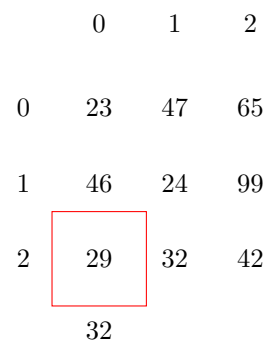


Figure 10: 2D Array with Element Access Highlighted

3 Applications of Arrays

3.1 Searching

Arrays are often used for searching due to their efficiency in accessing elements.

```
int search(int arr[], int n, int target) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == target) {
            return i;
        }
    }
    return -1;
}
```

3.2 Matrix Computation

2D arrays are commonly used in matrix computations. In computer vision, images are represented as 2D arrays of pixel values.

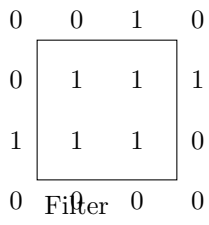


Figure 11: Image Segmentation Using 2D Arrays

4 Complexity Analysis

- **Access:** $O(1)$
- **Search:** $O(n)$
- **Insertion:** $O(n)$
- **Deletion:** $O(n)$

5 Conclusion

Arrays are a fundamental data structure with wide applications in computer science. They offer efficient access and are suitable for operations that do not involve frequent insertions and deletions.