Data Structures: Arrays

Minseok Jeon
DGIST

November 2, 2025

Contents

Introduction to Arrays

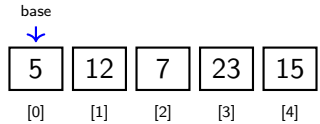
What is an Array?

Definition

An **array** is a contiguous memory block that stores elements of the same type, supporting **O(1) indexing**.

Key characteristics:

- Fixed or dynamic size
- Elements stored consecutively in memory
- Direct access via index
- Cache-friendly data structure
- Foundation for many other data structures



Array of 5 elements

Memory Address Calculation

Address of `arr[i]` = `base_address + i × sizeof(element)`

Static vs Dynamic Arrays

Static Arrays

Definition

Fixed size determined at compile time or initialization

Advantages:

- No resize overhead
- Predictable memory usage
- Slightly faster access
- Simple memory management

Disadvantages:

- Cannot grow or shrink
- Must know size in advance
- Wasted space if overallocated

Examples

C:

```c
int arr[100];  // Fixed size
```

Java:

```java
int[] arr = new int[100];
```

C++:

```cpp
int arr[100];
std::array<int, 100> arr;
```

Use Case

Dynamic Arrays

Definition

Resizable arrays that grow as needed

Growth Strategy:

- Start with small capacity
- Double capacity when full (typical)
- Allocate new array and copy elements
- Free old array

Examples:

- C++: `std::vector`
- Java: `ArrayList`
- Python: `list`
- C#: `List<T>`

Python Implementation

```python
class DynamicArray:
    def __init__(self):
        self.capacity = 1
        self.size = 0
        self.array = [None] * self.
             capacity

    def append(self, item):
        if self.size == self.capacity:
            self._resize()
        self.array[self.size] = item
        self.size += 1

    def _resize(self):
        self.capacity *= 2
        new_array = [None] * self.capacity
        for i in range(self.size):
            new_array[i] = self.array[i]
        self.array = new_array
```

Comparison: Static vs Dynamic Arrays

Operation	Static Array	Dynamic Array
Access	O(1)	O(1)
Append	N/A	O(1) amortized
Insert (middle)	O(n)	O(n)
Delete (middle)	O(n)	O(n)
Memory	Exact	1.5-2× actual size
Resize	No	Yes (expensive)

Static Arrays Best For

- Fixed-size data
- Memory-constrained systems
- Real-time systems

Dynamic Arrays Best For

- Unknown or changing size
- Frequent additions
- General-purpose use

Amortized Analysis

Dynamic Array Resizing Cost

Resizing Strategy

When array is full, allocate new array with $2\times$ capacity

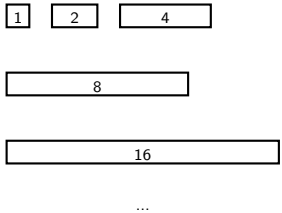
Single Resize Cost: O(n)

- Allocate new array: O(capacity)
- Copy all n elements: O(n)
- Free old array: O(1)

But resizes are infrequent!

- Resizes at sizes: 1, 2, 4, 8, 16, ..., n
- Total resizes: $\log_2 n$
- Total copy cost: $1 + 2 + 4 + ... + n = 2n - 1$

Growth Sequence



Amortized Analysis Result

Accounting Method for Amortized Analysis

Intuition

Charge extra "credits" for cheap operations to pay for expensive ones

Charge $3 per insertion:

- $1 for actual insertion
- $2 saved as credit for future resize

When resize happens:

- Need to copy n elements
- Each element has $2 credit saved
- Total credits: $2n
- Resize cost: $n (copying)
- Credits cover the cost!

Growth Factor Comparison

Factor	Memory	Frequency
1.5×	Lower	More
2×	Higher	Less

Shrinking Strategy

Shrink when `size < capacity/4`
(not `capacity/2` to avoid thrashing)

Trade-off: Space efficiency vs resize frequency

Array Operations

Access and Search Operations

Access by Index: O(1)

```
element = arr[i]
```

Direct memory calculation:
`base + i × sizeof(element)`

Linear Search: O(n)

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1
```

Check each element sequentially

Binary Search: O(log n)

Only for sorted arrays!

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

Key Insight

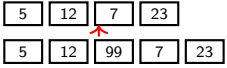
Binary search eliminates half the search space each iteration

Insert and Delete Operations

Insert at Position i: O(n)

```python
def insert(arr, index, value):
    arr.append(None)  # Ensure space
    # Shift elements right
    for i in range(len(arr)-1, index, -1):
        arr[i] = arr[i-1]
    arr[index] = value
```

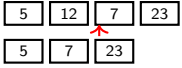
Insert 99 at index 2



Delete at Position i: O(n)

```python
def delete(arr, index):
    # Shift elements left
    for i in range(index, len(arr)-1):
        arr[i] = arr[i+1]
    arr.pop()  # Remove last
```

Delete at index 1



Performance Summary

- Insert/delete at end: O(1) amortized
- Insert/delete at beginning: O(n) - worst case
- Insert/delete in middle: O(n) - must shift elements

Multidimensional Arrays

2D Arrays (Matrices)

Declaration

Python:

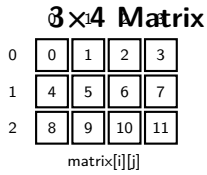
```python
matrix = [[0]*cols for _ in range(rows)]
```

Java:

```java
int[][] matrix = new int[rows][cols];
```

C++:

```cpp
vector<vector<int>> matrix(rows,
    vector<int>(cols));
```



Accessing Elements

`element = matrix[i][j]`

Time: O(1)

Memory Layout: Row-Major vs Column-Major

Row-Major Order

Used in: C, C++, Python, Java

Store rows contiguously:
`[row0][row1][row2]...`

Address calculation:
`base + (i × cols + j) × size`

Better for:

- Iterating by rows
- Row-wise operations

Column-Major Order

Used in: Fortran, MATLAB

Store columns contiguously:
`[col0][col1][col2]...`

Address calculation:
`base + (j × rows + i) × size`

Better for:

- Iterating by columns
- Column-wise operations

Performance Impact

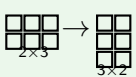
Accessing memory in the wrong order can cause 10-100× slowdown due to cache misses!

Common Matrix Operations

Transpose: O(rows × cols)

```python
def transpose(matrix):
    rows = len(matrix)
    cols = len(matrix[0])
    result = [[0]*rows for _ in range(cols)]
    for i in range(rows):
        for j in range(cols):
            result[j][i] = matrix[i][j]
    return result
```

Example



Rotate 90° Clockwise

```python
def rotate_90(matrix):
    n = len(matrix)
    # 1. Transpose
    for i in range(n):
        for j in range(i, n):
            temp = matrix[i][j]
            matrix[i][j] = matrix[j][i]
            matrix[j][i] = temp

    # 2. Reverse each row
    for i in range(n):
        matrix[i].reverse()
```

Matrix Multiplication

Naive: $O(n^3)$

Strassen: $O(n^{2.37})$

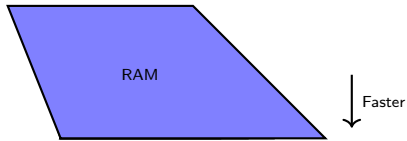
Coppersmith-Winograd: $O(n^{2.376})$

Memory Layout & Cache Locality

Cache Hierarchy and Performance

Memory Hierarchy

Level	Size	Latency
L1 Cache	32-64 KB	1-4 cycles
L2 Cache	256 KB-1 MB	10-20 cycles
L3 Cache	8-32 MB	40-75 cycles
RAM	GB	200+ cycles



Memory Hierarchy

Cache Line

- Typical size: 64 bytes
- Fetching one element loads entire cache line
- Sequential access benefits from prefetching

Spatial Locality: Good vs Bad Access Patterns

Good: Sequential Access

```python
# Excellent cache locality
sum = 0
for i in range(n):
    sum += arr[i]
```

- Accesses consecutive memory
- Cache prefetching works well
- Minimizes cache misses

Bad: Random Access

```python
# Poor cache locality
sum = 0
for i in random_indices:
    sum += arr[i]
```

- Jumps around memory
- No benefit from prefetching
- Frequent cache misses

Matrix Traversal Example (C/C++ row-major)

```c
// GOOD: Row-wise iteration (follows memory layout)
for (int i = 0; i < rows; i++)
    for (int j = 0; j < cols; j++)
        sum += matrix[i][j];
```

Optimization: Array of Structures vs Structure of Arrays

Array of Structures (AoS)

```
struct Point {
    float x, y, z;
};
Point points[1000];

// Access x coordinates
for (int i = 0; i < 1000; i++)
    sum += points[i].x;
```

Memory layout:

`x0 y0 z0 | x1 y1 z1 | x2 y2 z2 ...`

Issue: Loading unnecessary y, z values

Structure of Arrays (SoA)

```
struct Points {
    float x[1000];
    float y[1000];
    float z[1000];
};
Points points;

// Access x coordinates
for (int i = 0; i < 1000; i++)
    sum += points.x[i];
```

Memory layout:

`x0 x1 x2 ... | y0 y1 y2 ... | z0 z1 z2 ...`

Benefit: Better cache locality for single field

When to Use SoA

Applications & Limitations

When to Use Arrays

Common Applications

- **Lookup tables**: ASCII mappings, precomputed values
- **Buffers**: Fixed-size data storage, I/O buffers
- **Stacks/Queues**: Array-based implementations
- **Dynamic programming**: Memoization tables
- **Sorting algorithms**: In-place sorting
- **Image processing**: Pixel arrays (2D/3D)
- **Scientific computing**: Vectors, matrices, tensors

Advantages

- ✓ O(1) random access by index
- ✓ Cache-friendly (contiguous memory)
- ✓ Simple and intuitive
- ✓ Low memory overhead
- ✓ Excellent for iteration

Limitations

- × Fixed size or resize overhead
- × O(n) insertion/deletion in middle
- × Wasted space if sparse
- × Cannot efficiently grow in middle

When to Use Alternatives

Use Case	Better Alternative	Reason
Frequent insertions/deletions	Linked List	O(1) insert/delete
Unknown size, many insertions	Dynamic Array	Amortized O(1) append
Key-value lookups	Hash Table	O(1) average lookup
Sorted data with insertions	Binary Search Tree	O(log n) operations
Priority queue	Heap	O(log n) insert/extract-min
Sparse data	Hash Table / Sparse Matrix	Space efficient

Summary

Key Takeaways

Array Fundamentals

- Contiguous memory blocks with O(1) indexing
- Foundation for many data structures
- Direct memory address calculation

Static vs Dynamic

- Static: fixed size, no overhead, predictable
- Dynamic: resizable, O(1) amortized append, flexible
- Amortized analysis shows O(1) average cost for dynamic growth

Performance Considerations

- Cache locality is critical: 10-100$\times$ performance difference
- Access patterns matter: sequential $>$ random
- Row-major vs column-major order affects cache performance

Thank You!

Questions?