DATA STRUCTURES AND ALGORITHMS

TABLE OF CONTENTS

---

1. INTRODUCTION TO DSA

WHAT IS A DATA STRUCTURE?

A data structure is a specialized format for organizing, processing, storing, and retrieving data in a computer. It defines the relationship

between data elements and the operations that can be performed on them.

Key Characteristics:

- Organization: How data is arranged in memory
- Access: Methods to retrieve stored data
- Operations: Insert, delete, update, search capabilities
- Efficiency: Time and space requirements

Think of it like organizing books in a library:

- By alphabetical order (easy to find specific book)
- By category (easy to find similar books)
- By publication date (easy to find recent books)

Each organization method serves different purposes and has different

trade-offs in terms of search speed and maintenance effort.

WHY DSA IS IMPORTANT?

---

1. EFFICIENCY

- Faster program execution
- Reduced computational resources
- Better user experience

2. PROBLEM SOLVING

- Breaks complex problems into manageable parts
- Provides proven solutions to common problems
- Enables systematic thinking

3. SCALABILITY

- Handles large datasets effectively
- Maintains performance as data grows
- Critical for real-world applications

4. CAREER OPPORTUNITIES

- Core of technical interviews
- Foundation for software engineering
- Required for competitive programming

5. REAL-WORLD APPLICATIONS

- Database management systems
- Operating systems (process scheduling)
- Networking (routing algorithms)
- Artificial Intelligence and Machine Learning
- Web browsers (history, cache)
- GPS navigation systems

---

2. TIME & SPACE COMPLEXITY

BIG O NOTATION

Big O notation describes the upper bound (worst-case) performance of an algorithm as the input size grows to infinity. It focuses on the dominant term and ignores constants.

Example: If algorithm takes $3n^2 + 5n + 10$ operations
→ Big O = $O(n^2)$ (only the largest term matters)

COMMON COMPLEXITIES (Best to Worst)

---

1. O(1) - CONSTANT

- Same execution time regardless of input size
- Example: Accessing array element by index
- Best possible complexity

2. O(n) - LINEAR

- Time grows proportionally with input
- Must check each element once
- Example: Linear search, simple loops
- Acceptable for most applications

3. $O(n^2)$ - QUADRATIC

- Time grows with square of input
- Nested loops over same data
- Example: Bubble Sort, Selection Sort
- Slow for large inputs

VISUAL COMPARISON (Operations for input size n)

	n = 10	n = 100	n = 1000
O(1)	1	1	1
O(n)	10	100	1,000
$O(n^2)$	100	10,000	1,000,000

BEST, WORST, AVERAGE CASE

BEST CASE (Omega Ω)

• Minimum time/space required

• Most favorable input scenario

• Example: Finding element at first position in linear search

AVERAGE CASE (Theta Θ)

• Expected performance over all possible inputs

• Most realistic measure

• Considers probability distribution

WORST CASE (Big O)

• Maximum time/space required

• Least favorable input scenario

• Most commonly used in analysis

• Example: Finding element at last position or not present

SPACE COMPLEXITY

Measures memory usage as input size grows.

Components:

1. Fixed Part: Space for constants, variables (independent of input)
2. Variable Part: Space that depends on input size

Examples:

• $O(1)$: Few variables regardless of input

• $O(n)$: Space grows linearly (copying array)

• $O(n^2)$: 2D matrix storage

TIME-SPACE TRADEOFF

Often, you can trade space for time or vice versa:

• More Memory → Faster Execution (caching, lookup tables)
• Less Memory → Slower Execution (recalculate instead of store)

---

3. ARRAYS

DEFINITION

An array is a collection of elements stored at contiguous (adjacent) memory locations. Each element is identified by an index or key.

Key Properties:

• Fixed size (in static arrays)
• Same data type for all elements
• Direct access using index
• Contiguous memory allocation

MEMORY REPRESENTATION

Array: [10, 20, 30, 40, 50]

Memory:

```
┌────┬────┬────┬────┬────┐
| 10 | 20 | 30 | 40 | 50 |
└────┴────┴────┴────┴────┘
  0    1    2    3    4   ← Index
```

Base Address + (Index × Element Size) = Element Address

INDEXING

• Zero-based indexing: First element at index 0

• Random access: O(1) time to access any element

• Formula: memory_address = base_address + (index × size_of_element)

OPERATIONS & COMPLEXITY

---

1. ACCESS/READ - O(1)

   • Direct access using index

- Fastest operation

2. SEARCH - O(n)
   - Linear search: Check each element
   - Binary search (sorted array): O(log n)

3. INSERT
   - At end: O(1) if space available
   - At beginning: O(n) - shift all elements right
   - At middle: O(n) - shift elements from insertion point

4. DELETE
   - At end: O(1)
   - At beginning: O(n) - shift all elements left
   - At middle: O(n) - shift elements after deletion

5. UPDATE - O(1)
   - Direct access and modification

ADVANTAGES

✓ Fast random access (O(1))

✓ Simple and easy to use

✓ Cache-friendly (contiguous memory)

✓ Memory efficient (no extra pointers)

✓ Good for iteration

DISADVANTAGES

✗ Fixed size (static arrays)

✗ Expensive insertion/deletion at beginning

✗ Wasted memory if not fully utilized

✗ Difficult to resize

USE CASES

• Storing collections with known size

• Implementing other data structures (stack, queue)

• Lookup tables

• Buffer for I/O operations

• Matrix operations

• Storing sensor data, pixel values

4. LINKED LISTS

DEFINITION

A linked list is a linear data structure where elements (nodes) are not stored at contiguous locations. Each node contains data and a reference (pointer) to the next node.

WHY LINKED LISTS?

Arrays have limitations:

1. Fixed size
2. Expensive insertion/deletion at beginning
3. Memory waste if size unknown

Linked Lists solve these:

1. Dynamic size (grow/shrink as needed)
2. Efficient insertion/deletion at beginning: O(1)
3. Allocate memory as needed

NODE STRUCTURE

Each node contains:

```
┌──────────┬──────────┐
| Data  |  Next  |
└──────────┴──────────┘
  ↓      ↓
Value   Pointer to next node
```

1. SINGLY LINKED LIST

- One pointer: points to next node
- One-way traversal (forward only)
- Less memory per node

Structure:

[10|→] → [20|→] → [30|→] → [NULL]

2. DOUBLY LINKED LIST

- Two pointers: previous and next
- Bi-directional traversal
- More memory per node
- Easier deletion

Structure:

[NULL|←|10|→] ↔ [←|20|→] ↔ [←|30|→|NULL]

OPERATIONS & COMPLEXITY

---

1. INSERT AT BEGINNING - O(1)
   - Create new node
   - Point new node to current head
   - Update head to new node

2. INSERT AT END - O(n)
   - Traverse to last node
   - Add new node after last

3. INSERT AT POSITION - O(n)
   - Traverse to position-1
   - Adjust pointers

4. DELETE - O(n)
   - Search for node: O(n)
   - Adjust pointers: O(1)
   - Delete at beginning: O(1)

5. SEARCH - O(n)

- Must traverse from head
- No random access

6. ACCESS - O(n)

- Must traverse from beginning

ADVANTAGES

✓ Dynamic size

✓ Efficient insertion/deletion at beginning: O(1)

✓ No memory waste

✓ Easy to implement stack/queue

✓ Can easily insert in middle

DISADVANTAGES

✗ No random access (must traverse)

✗ Extra memory for pointers

✗ Not cache-friendly (scattered in memory)

✗ Traversal is sequential only

COMPARISON: ARRAY VS LINKED LIST

Aspect	Array	Linked List
Size	Fixed	Dynamic
Access	O(1)	O(n)
Insert (beginning)	O(n)	O(1)
Insert (end)	O(1)	O(n)
Search	O(n)	O(n)
Memory	Contiguous	Scattered
Extra Space	None	Pointers

USE CASES

- Implementation of stacks and queues
- Music/video playlist (doubly linked)
- Browser history (backward/forward)
- Undo functionality in applications
- Dynamic memory allocation
- Image viewer (next/previous)

---

5. STACK

DEFINITION

A stack is a linear data structure that follows the Last In First Out (LIFO) principle. The last element added is the first one to be removed.

Analogy: Stack of plates - you add and remove plates from the top only.

LIFO PRINCIPLE

Last In, First Out

Operations happen at one end only (called "top"):

```
┌─────┐
│  3  │ ← Top (most recent)
├─────┤
│  2  │
├─────┤
│  1  │ ← Bottom (oldest)
└─────┘
```

Push 4 → 4 goes on top

Pop → 4 comes out first (not 1)

BASIC OPERATIONS

1. PUSH - O(1)
   • Add element to top of stack
   • Increment top pointer

2. POP - O(1)
   • Remove and return top element
   • Decrement top pointer
   • Check for underflow (empty stack)

3. PEEK/TOP - O(1)
   • View top element without removing
   • No modification to stack

4. IS_EMPTY - O(1)
   • Check if stack has no elements
   • Returns true/false

5. SIZE - O(1)
   • Return number of elements

- Track with counter variable

IMPLEMENTATION METHODS

---

1. USING ARRAY
   - Fixed size
   - Fast operations
   - Risk of overflow

2. USING LINKED LIST
   - Dynamic size
   - No overflow risk
   - Extra memory for pointers

ADVANTAGES

✓ Simple operations: O(1)

✓ Memory efficient (only store data, no random access needed)

✓ Easy to implement

✓ Backtracking becomes natural

DISADVANTAGES

✗ Limited access (only top element)

✗ No random access

✗ Fixed size (array implementation)

✗ Stack overflow possible

APPLICATIONS

---

1. FUNCTION CALLS (Call Stack)

- Store return addresses
- Local variables storage
- Recursion management

SYNTAX PARSING

- Checking balanced parentheses
- Compiler syntax checking
- XML/HTML tag matching

UNDO/REDO FUNCTIONALITY

- Text editors
- Photoshop operations
- Browser history

BROWSER HISTORY

• Back button functionality

REAL-WORLD EXAMPLES

• Stack of plates in a cafeteria
• Stack of books on a desk
• Browser back button
• Ctrl+Z (undo) in applications
• Function execution in programming

6. QUEUE

DEFINITION

A queue is a linear data structure that follows the First In First Out (FIFO) principle. The first element added is the first one to be removed.

Analogy: Queue at a ticket counter - first person in line is served first.

FIFO PRINCIPLE

First In, First Out

Operations happen at two ends:

• Rear (back): Where elements are added (enqueue)

• Front: Where elements are removed (dequeue)

```
 Front              Rear
  ↓                  ↓
 ┌────┬────┬────┬────┬────┐
 | 1 | 2 | 3 | 4 | 5 |
 └────┴────┴────┴────┴────┘
  ↑              ↑
 Remove         Add
 (Dequeue)     (Enqueue)
```

BASIC OPERATIONS

---

1. ENQUEUE - O(1)

   • Add element at rear

   • Increment rear pointer

2. DEQUEUE - O(1)

   • Remove element from front

   • Increment front pointer

• Check for underflow (empty queue)

3. FRONT/PEEK - O(1)

• View front element without removing

4. IS_EMPTY - O(1)

• Check if queue has no elements

5. IS_FULL - O(1)

• Check if queue is full (array implementation)

6. SIZE - O(1)

• Return number of elements

TYPES OF QUEUES

---

1. SIMPLE QUEUE (Linear Queue)

• Basic FIFO structure

• Front and rear move forward only

• Memory waste possible

2. CIRCULAR QUEUE

- Rear connects back to front
- Efficient memory usage
- No waste of space
- Used in CPU scheduling

Visualization:

```
 [2] → [3]
  ↑     ↓
 [1] ← [4]
```

3. PRIORITY QUEUE

- Elements have priority
- Higher priority served first
- Not strict FIFO
- Used in scheduling algorithms

IMPLEMENTATION METHODS

---

1. USING ARRAY

- Fixed size
- Fast operations

2. USING LINKED LIST

- Dynamic size
- No size limitation
- Extra memory for pointers

ADVANTAGES

✓ Fair ordering (FIFO)

✓ Fast operations: O(1)

✓ Natural for scheduling/buffering

✓ Easy to implement

DISADVANTAGES

✗ Limited access (only front/rear)

✗ No random access

✗ Fixed size (array implementation)

✗ Queue overflow possible

APPLICATIONS

---

1. CPU SCHEDULING

- Process scheduling
- Round-robin scheduling

- Job scheduling

2. I/O BUFFERING
   - Keyboard buffer
   - Printer queue
   - Disk scheduling

3. HANDLING REQUESTS
   - Web server request handling
   - Call center systems
   - Customer service queues

4. REAL-TIME SYSTEMS
   - Handling asynchronous data
   - Message queues
   - Event handling

5. DATA STREAMING
   - Audio/video buffering
   - Network packets

6. SIMULATION

• Airport systems

• Bank queuing systems

REAL-WORLD EXAMPLES

• People waiting in line at a store

• Cars at a toll booth

• Print jobs sent to printer

• Customer support ticket system

• Playlist (song queue)

• Messages in messaging apps

QUEUE VS STACK

Aspect	Stack	Queue
Principle	LIFO	FIFO
Operations	Push/Pop	Enqueue/Dequeue
Access Points	One (top)	Two (front/rear)
Use Case	Backtracking	Scheduling
Example	Undo feature	Printer queue

7. SEARCHING ALGORITHMS

WHAT IS SEARCHING?

Searching is the process of finding a particular element in a collection of elements. The goal is to determine whether the element exists and, if so, its position.

LINEAR SEARCH (Sequential Search)

CONCEPT:

Check each element one by one from beginning to end until target is found
or end is reached.

Process:

Array: [64, 34, 25, 12, 22]

Target: 12

Step 1: Check 64 ≠ 12

Step 2: Check 34 ≠ 12

Step 3: Check 25 ≠ 12

Step 4: Check 12 = 12 ✓ Found!

COMPLEXITY:

• Time Complexity:

- Best Case: O(1) - element at first position
- Average Case: O(n) - element in middle
- Worst Case: O(n) - element at end or not present

• Space Complexity: O(1) - no extra space needed

CHARACTERISTICS:

• Works on both sorted and unsorted arrays

• Simple to implement

• Inefficient for large datasets

• No preprocessing required

ADVANTAGES:

✓ Simple and easy to understand

✓ Works on unsorted data

✓ Good for small datasets

✓ No extra memory needed

DISADVANTAGES:

✗ Slow for large datasets

✗ Inefficient compared to other methods

✗ Time increases linearly with size

BINARY SEARCH

CONCEPT:

Efficiently search in a SORTED array by repeatedly dividing the search interval in half. Compare target with middle element and eliminate half of the remaining elements.

PREREQUISITE: Array must be sorted!

Process:

Sorted Array: [11, 12, 22, 25, 34, 64, 90]

Target: 25

Step 1: Middle = 25, Found! ✓

Another example, Target: 90

Step 1: Middle = 25, 90 > 25, search right half

Step 2: Middle = 64, 90 > 64, search right half

Step 3: Middle = 90, Found! ✓

ALGORITHM STEPS:

1. Find middle element
2. If target = middle, return position
3. If target < middle, search left half
4. If target > middle, search right half
5. Repeat until found or no elements left

COMPLEXITY:

• Time Complexity:
  - Best Case: O(1) - element at middle
  - Average Case: O(log n)
  - Worst Case: O(log n)

• Space Complexity:
  - Iterative: O(1)
  - Recursive: O(log n) - call stack

WHY O(log n)?

Each comparison eliminates half the elements:

$n \to n/2 \to n/4 \to n/8 \to \ldots \to 1$

Number of divisions = $\log_2(n)$

CHARACTERISTICS:

• Only works on sorted arrays

• Much faster than linear search

• Divide and conquer approach

• Can be implemented iteratively or recursively

ADVANTAGES:

✓ Very efficient: O(log n)

✓ Much faster for large datasets

✓ Predictable performance

DISADVANTAGES:

✗ Requires sorted array

✗ Sorting takes O(n log n) time if unsorted

✗ More complex than linear search

✗ Not suitable for frequently changing data

COMPARISON: LINEAR VS BINARY SEARCH

Aspect	Linear	Binary

Data Requirement	Any	Sorted only
Time Complexity	O(n)	O(log n)
Space Complexity	O(1)	O(1) or O(log n)
Best For	Small/unsorted	Large/sorted
Implementation	Very simple	Moderate

Performance Comparison (n = 1,000,000):

• Linear Search: ~1,000,000 comparisons (worst case)

• Binary Search: ~20 comparisons (worst case)

WHEN TO USE WHICH?

USE LINEAR SEARCH when:

• Array is unsorted

• Array is small

• Search is infrequent

• Simplicity is priority

USE BINARY SEARCH when:

• Array is sorted

• Array is large

• Frequent searches needed

• Performance is critical

---

8. SORTING ALGORITHMS

WHAT IS SORTING?

Sorting is arranging elements in a specific order (ascending or descending).

Sorted data enables efficient searching, makes data analysis easier, and improves overall algorithm performance.

BUBBLE SORT

CONCEPT:

Repeatedly compare adjacent elements and swap them if they're in wrong order.

Largest element "bubbles up" to the end in each pass.

HOW IT WORKS:

Pass 1: Compare all adjacent pairs, swap if needed

Pass 2: Repeat, but last element already in place

Continue until no swaps needed

Example: [5, 1, 4, 2]

Pass 1: [1, 4, 2, 5] - 5 bubbles to end

Pass 2: [1, 2, 4, 5] - 4 in place

Pass 3: No swaps - sorted!

COMPLEXITY:

• Time Complexity:

  - Best Case: $O(n)$ - already sorted

  - Average Case: $O(n^2)$

  - Worst Case: $O(n^2)$ - reverse sorted

• Space Complexity: $O(1)$ - in-place sorting

CHARACTERISTICS:

• Simple to understand and implement

• Stable sort (maintains relative order)

• Adaptive (faster if partially sorted)

• In-place sorting

ADVANTAGES:

✓ Very simple implementation

✓ Stable sorting

✓ Works well for small datasets

✓ Detects if already sorted

DISADVANTAGES:

✗ Very slow for large datasets

✗ $O(n^2)$ makes it impractical

✗ Many unnecessary comparisons

SELECTION SORT

CONCEPT:

Find the minimum element from unsorted part and place it at the beginning.

Repeat for remaining unsorted part.

HOW IT WORKS:

1. Find minimum in entire array
2. Swap with first position
3. Find minimum in remaining array
4. Swap with second position
5. Repeat until sorted

Example: [64, 25, 12, 22, 11]

Step 1: Find min (11), swap with first → [11, 25, 12, 22, 64]

Step 2: Find min (12), swap with second → [11, 12, 25, 22, 64]

Step 3: Find min (22), swap with third → [11, 12, 22, 25, 64]

Step 4: Find min (25), swap with fourth → [11, 12, 22, 25, 64]

COMPLEXITY:

• Time Complexity:

  - Best Case: $O(n^2)$

  - Average Case: $O(n^2)$

  - Worst Case: $O(n^2)$

• Space Complexity: $O(1)$

CHARACTERISTICS:

• Always makes same number of comparisons

• Minimizes number of swaps

• Not stable by default

• In-place sorting

ADVANTAGES:

✓ Simple to implement

✓ Minimum number of swaps: $O(n)$

✓ Works well for small datasets

✓ Memory efficient

DISADVANTAGES:

✗ $O(n^2)$ even for sorted array

✗ Not adaptive

✗ Unstable sort

INSERTION SORT

CONCEPT:

Build sorted array one element at a time by inserting each element into its correct position, similar to sorting playing cards in hand.

HOW IT WORKS:

1. Start with second element
2. Compare with elements before it
3. Shift larger elements right
4. Insert element in correct position
5. Repeat for all elements

Example: [12, 11, 13, 5, 6]

Start: [12] | 11, 13, 5, 6

Step 1: [11, 12] | 13, 5, 6

Step 2: [11, 12, 13] | 5, 6

Step 3: [5, 11, 12, 13] | 6

Step 4: [5, 6, 11, 12, 13]

COMPLEXITY:

• Time Complexity:

- Best Case: $O(n)$ - already sorted
- Average Case: $O(n^2)$
- Worst Case: $O(n^2)$ - reverse sorted

• Space Complexity: $O(1)$

CHARACTERISTICS:

• Simple and intuitive

• Stable sort

• Adaptive (efficient for nearly sorted data)

• In-place sorting

• Online (can sort as data arrives)

ADVANTAGES:

✓ Simple implementation

✓ Efficient for small datasets

✓ Adaptive - O(n) for nearly sorted

✓ Stable sorting

✓ Online algorithm

✓ Low overhead

DISADVANTAGES:

✗ $O(n^2)$ for large datasets

✗ Not efficient for large unsorted data

---

9. WHEN TO USE WHICH DATA STRUCTURE

DECISION FRAMEWORK

Choosing the right data structure is crucial for efficient programs.

Consider these factors:

1. OPERATION FREQUENCY
   - What operations are most common?

- How often do you insert/delete/search?

2. DATA SIZE
   - Small dataset (< 100 elements)
   - Medium dataset (100 - 10,000)
   - Large dataset (> 10,000)

3. MEMORY CONSTRAINTS
   - Is memory limited?
   - Can you afford extra space?

4. ACCESS PATTERNS
   - Random access needed?
   - Sequential access only?
   - Both ends access?

5. ORDERING REQUIREMENTS
   - Must maintain insertion order?
   - Need sorted data?
   - Order doesn't matter?

DATA STRUCTURE SELECTION GUIDE

ARRAY

WHEN TO USE:

✓ Size is known and fixed

✓ Need fast random access by index

✓ Simple collection of elements

✓ Cache performance matters

✓ Sequential access

BEST FOR:

• Storing sensor readings

• Fixed-size lookup tables

• Matrix operations

• Image pixels

• Buffer for I/O

EXAMPLES:

• Student grades (fixed class size)

• Days of week

• RGB color values

• Board game grid

LINKED LIST

____

WHEN TO USE:

✓ Size changes frequently

✓ Many insertions/deletions at beginning

✓ Don't need random access

✓ Memory fragmentation okay

BEST FOR:

• Dynamic collections

• Implementing stack/queue

• When insertion at beginning is frequent

• Unknown size at compile time

EXAMPLES:

• Music playlist (add/remove songs)

• Photo gallery (next/previous)

• Browser tabs

• Train coaches (add/remove)

AVOID WHEN:

✗ Need fast random access

✗ Searching frequently

✗ Memory is very limited

STACK

———

WHEN TO USE:

✓ Need LIFO behavior

✓ Backtracking required

✓ Nested structures

✓ Function calls

BEST FOR:

- Undo/Redo operations
- Expression evaluation
- Syntax checking
- Backtracking problems
- Function call management

EXAMPLES:

- Text editor undo

• Browser back button

• Balanced parentheses checker

REAL-WORLD:

• Stack of plates

• Books on a desk

• Email drafts

QUEUE

———

WHEN TO USE:

✓ Need FIFO behavior

✓ Processing in order

✓ Scheduling tasks

✓ Buffering data

BEST FOR:

• Task scheduling

• Request handling

• Buffer management

EXAMPLES:

• Print job queue

• Customer service line

• CPU task scheduling

• Message queue

• Call center system

REAL-WORLD:

• Ticket counter line

• Restaurant waiting list

• Hospital emergency queue (priority)

HASH TABLE (Dictionary/Map)

WHEN TO USE:

✓ Need O(1) lookup

✓ Key-value associations

✓ Uniqueness checking

✓ Counting frequencies

BEST FOR:

• Fast lookups

• Caching

• Removing duplicates

• Counting occurrences

• Database indexing

EXAMPLES:

• Phone book (name → number)

• Student records (ID → data)

• Word frequency counter

• Cache implementation

• Symbol tables

AVOID WHEN:

✗ Need sorted data

✗ Order matters

✗ Memory is very limited

TREE (Binary Search Tree)

WHEN TO USE:

✓ Hierarchical data

✓ Fast search, insert, delete (O(log n))

✓ Sorted data needed

✓ Range queries

BEST FOR:

• Hierarchical relationships

• Sorted data with modifications

• Quick search operations

• File systems

EXAMPLES:

• Organization chart

• File system directories

• Decision trees

• Expression parsing

• Database indexing

REAL-WORLD:

• Family tree

• Company hierarchy

• Tournament bracket

GRAPH

———

WHEN TO USE:

✓ Complex relationships

✓ Network connections

✓ Many-to-many relationships

✓ Path finding needed

BEST FOR:

• Social networks

• Maps and navigation

• Network topology

• Dependency resolution

• Recommendation systems

EXAMPLES:

• Social media connections

• Google Maps

• Flight routes

• Web page links

• Project dependencies

COMPLEXITY-BASED DECISIONS

IF PRIORITY IS FAST ACCESS:

• Array: O(1) with index

• Hash Table: O(1) with key

IF PRIORITY IS FAST INSERTION AT BEGINNING:

• Linked List: O(1)

• Stack: O(1) (if only at top)

IF PRIORITY IS FAST SEARCH:

• Hash Table: O(1) average

• Binary Search Tree: O(log n)

• Sorted Array with Binary Search: O(log n)

IF PRIORITY IS SORTED DATA:

• Binary Search Tree

• Sorted Array (if few updates)

IF PRIORITY IS MEMORY EFFICIENCY:

• Array: No pointer overhead

• Avoid linked structures if possible

REAL-WORLD SCENARIO MAPPING

SCENARIO: Social Media Platform

Feature	Data Structure
Friend List	Graph / Hash Table
News Feed	Queue / Array
Like Count	Hash Table
Comments	Linked List / Array
User Cache	Hash Table
Recently Viewed	Stack / Array

SCENARIO: Text Editor

Feature	Data Structure
Document Text	Array / Rope (special tree)

Undo History	Stack
Redo History	Stack
Find & Replace	Hash Table
Line Numbers	Array

SCENARIO: E-commerce Website

Feature	Data Structure
Product Catalog	Hash Table
Shopping Cart	Array / Linked List
Order History	Array / List
Product Search	Tree / Hash Table
Recently Viewed	Queue (circular)
Recommendations	Graph

KEY TAKEAWAYS

1. DATA STRUCTURES

- Choose based on operation frequency and requirements
- No single "best" structure - depends on use case
- Understand trade-offs between time and space

2. TIME COMPLEXITY

- $O(1) > O(\log n) > O(n) > O(n^2)$
- Always consider worst case
- Aim for lowest possible complexity

3. SPACE COMPLEXITY

- Consider memory constraints
- Sometimes worth using more space for speed
- In-place algorithms are memory efficient

4. SEARCHING

- Linear: $O(n)$, works on unsorted
- Binary: $O(\log n)$, requires sorted
- Always sort first if multiple searches needed

Remember: Understanding concepts is more important than memorizing!

Focus on when and why to use each structure.