Introduction to Data Structures

Minseok Jeon
DGIST

November 2, 2025

Contents

What Are Data Structures?

Definition

Definition

A **data structure** is a specialized format for organizing, storing, and managing data in a computer so that it can be accessed and modified efficiently.

Why We Need Them:

1. **Efficiency**
   Right structure = milliseconds
   Wrong structure = hours
2. **Organization**
   Systematic data management
3. **Reusability**
   Solve similar problems
4. **Abstraction**
   Hide implementation details

Real-World Analogies

- **Bookshelf** → Array
  Organized for easy retrieval
- **Filing cabinet** → Hash table
  Labeled drawers
- **Stack of plates** → Stack
  Take from top
- **Store line** → Queue
  First come, first served

Impact on Performance

Example: Checking if Element Exists

Using List (Array):

```
my_list = [1, 2, ..., 1000000]

if 999999 in my_list:  # O(n)
    print("Found!")

# Must scan through all elements
# For 1M elements: slow!
```

Using Set (Hash Table):

```
my_set = {1, 2, ..., 1000000}

if 999999 in my_set:   # O(1)
    print("Found!")

# Direct lookup
# For 1M elements: instant!
```

Key Insight

For 1 million elements, set lookup is nearly instantaneous while list scan takes significant time. **Choosing the right data structure matters!**

Time vs Space Trade-offs

The Fundamental Trade-off

Core Principle

Often, we can make an algorithm **faster** by using **more memory**, or **save memory** by accepting **slower execution**.

Time Complexity

How runtime grows with input size

- Focus: Speed
- Measure: Operations
- Goal: Minimize time

Space Complexity

How memory usage grows with input size

- Focus: Memory
- Measure: Bytes/storage
- Goal: Minimize space

Reality Check

There's **rarely** a solution that is both the fastest AND uses the least memory. Engineers must choose based on constraints.

Trade-off Example 1: Memoization

Without Memoization

Slower, Less Memory

```
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

# Time: O(2^n) - exponential!
# Space: O(n) - recursion stack
# fib(40) takes several seconds
```

With Memoization

Faster, More Memory

```
cache = {}
def fibonacci_memo(n):
    if n in cache:
        return cache[n]
    if n <= 1:
        return n
    result = fibonacci_memo(n-1) + fibonacci_memo(n
        -2)
    cache[n] = result
    return cache[n]

# Time: O(n) - much better!
# Space: O(n) - cache storage
# fib(40) is instant
```

Trade-off: Used extra memory (cache) to gain massive speed improvement

Trade-off Example 2: In-place vs Out-of-place

Out-of-place Algorithm

More Space, Simpler

```python
def reverse_list(arr):
    return arr[::-1]

# Creates new array
# Space: O(n)
# Original preserved
```

In-place Algorithm

Less Space, More Complex

```python
def reverse_inplace(arr):
    left, right = 0, len(arr)-1
    while left < right:
        temp = arr[left]
        arr[left] = arr[right]
        arr[right] = temp
        left += 1
        right -= 1

# Modifies original
# Space: O(1)
```

When to Choose What?

Optimize for Time When:

- Dealing with large datasets
- User experience matters
- Memory is abundant
- Real-time requirements
- Interactive applications

Optimize for Space When:

- Memory is limited
- Embedded systems
- Mobile devices
- Data too large for RAM
- Resource-constrained devices

Examples

- Web servers (caching)
- Video games
- Trading systems
- Search engines

Examples

- IoT devices
- Microcontrollers
- Big data processing
- Satellite systems

Asymptotic Analysis & Big-O

Why Asymptotic Analysis?

The Problem with Exact Counts

Actual runtime depends on:

- Hardware (CPU speed, memory)
- Programming language
- Compiler optimizations
- Current system load

The Solution: Asymptotic Analysis

Focus on **growth rate** as input size approaches infinity

- Hardware-independent
- Language-independent
- Describes **scalability**
- Constants become insignificant with large n

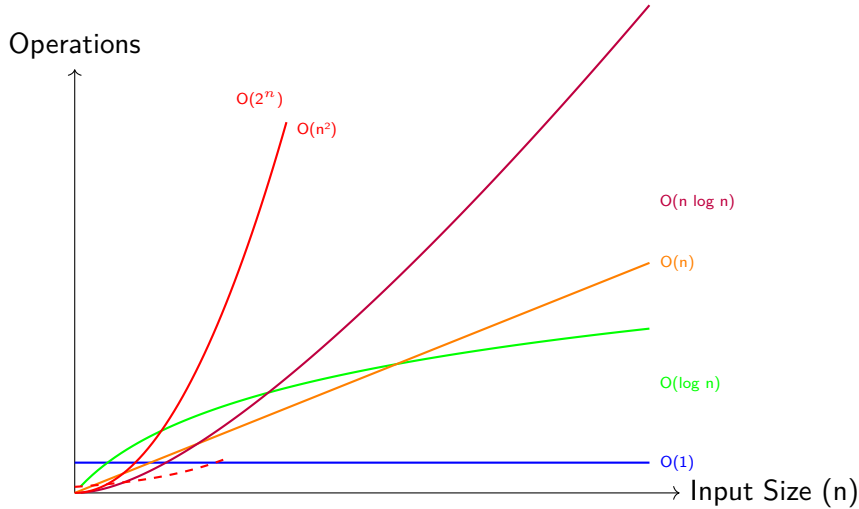
Common Big-O Classes

Notation	Name	Example	n=100
O(1)	Constant	Array access, hash lookup	1
O(log n)	Logarithmic	Binary search	≈ 7
O(n)	Linear	Linear search, scan	100
O(n log n)	Linearithmic	Merge sort, quicksort	≈ 664
$O(n^2)$	Quadratic	Nested loops, bubble sort	10,000
$O(n^3)$	Cubic	Triple nested loops	1,000,000
$O(2^n)$	Exponential	Recursive fibonacci	$\approx 10^{30}$
O(n!)	Factorial	All permutations	$\approx 10^{157}$

Visual Scale

For n = 100, $O(2^n)$ is **universe-ending**! Exponential growth is catastrophic.

Big-O Growth Visualization



From fastest (top) to slowest (bottom)

Big-O Rules

Rule 1: Drop Constants

$O(2n) \rightarrow O(n)$
$O(100) \rightarrow O(1)$
$O(n/2) \rightarrow O(n)$

Why? Constants don't affect growth rate

Rule 2: Drop Lower-Order Terms

$O(n^2 + n) \rightarrow O(n^2)$
$O(n^2 + 100*n + 50) \rightarrow O(n^2)$
$O(n^3 + n^2 + n) \rightarrow O(n^3)$

Why? Highest order dominates for large n

Rule 3: Different Variables

Two inputs? Use different variables!
Sequential: O(a + b)

Nested: O(a * b)

Rule 4: Worst Case

Big-O Examples

O(1) - Constant

```python
def get_first(arr):
    return arr[0]
# Always 1 operation
```

O(n) - Linear

```python
def find_max(arr):
    max_val = arr[0]
    for num in arr:  # n iterations
        if num > max_val:
            max_val = num
    return max_val
```

O(n^2) - Quadratic

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):        # n times
```

O(log n) - Logarithmic

```python
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
        # Halves search space each time!
    return -1
```

O(n log n) - Linearithmic

```python
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
```

Big-O of Common Operations

Sorting Algorithms Comparison

Algorithm	Average	Worst	Space	Stable?
Quick Sort	O(n log n)	$O(n^2)$	O(log n)	No
Merge Sort	O(n log n)	O(n log n)	O(n)	Yes
Heap Sort	O(n log n)	O(n log n)	O(1)	No
Bubble Sort	$O(n^2)$	$O(n^2)$	O(1)	Yes
Insertion Sort	$O(n^2)$	$O(n^2)$	O(1)	Yes

Quick Selection Guide

- Need **fast random access**? → Arrays
- Need **fast insertion/deletion at beginning**? → Linked lists
- Need **fast key-value lookup**? → Hash tables
- Need **sorted data with fast operations**? → Balanced BSTs
- Need **LIFO access**? → Stacks

When to Choose a Structure

Decision Framework

Questions to Ask

1. **What operations will be most frequent?**
   - Lookups? → Hash table or BST
   - Insertions/deletions? → Linked list or balanced tree
   - Processing in order? → Queue
   - Need to undo? → Stack
2. **Do you need ordered data?**
   - Yes, sorted → BST, sorted array
   - Yes, insertion order → Array, linked list
   - No → Hash table (fastest)
3. **Do you know the size in advance?**
   - Yes, fixed → Array
   - No, dynamic → Linked list, dynamic array, hash table
4. **What are your constraints?**
   - Limited memory → Space-efficient structures

Real-World Scenario 1: Phone Book

Requirements

- Fast lookup by name
- Alphabetical order useful
- Operations:
  - Search (frequent)
  - Insert (rare)
  - Delete (rare)

Decision

Hash table for O(1) lookup
or **BST** for O(log n) with ordering

Implementation

```
# Using hash table
phone_book = {
    "Alice": "555-1234",
    "Bob": "555-5678"
}

# O(1) lookup
number = phone_book["Alice"]
```

Real-World Scenario 2: Browser History

Requirements

- Recent pages accessed frequently
- Navigate back and forward
- Operations:
  - Push new page
  - Go back (pop)
  - Go forward (pop)

Decision

Two stacks:

- Back stack
- Forward stack

Implementation

```
back_stack = []
forward_stack = []

def visit_page(url):
    back_stack.append(current_page)
    forward_stack.clear()
    current_page = url

def go_back():
    if back_stack:
        forward_stack.append(current_page)
        current_page = back_stack.pop()

def go_forward():
    if forward_stack:
        back_stack.append(current_page)
        current_page = forward_stack.pop()
```

Real-World Scenario 3: Print Queue

Requirements

- First document submitted prints first
- Operations:
  - Add to queue (frequent)
  - Remove from queue (frequent)
- FIFO behavior essential

Decision

Queue
(First In, First Out)

Implementation

```
from collections import deque

print_queue = deque()

# Enqueue documents
print_queue.append("doc1.pdf")
print_queue.append("doc2.pdf")
print_queue.append("doc3.pdf")

# Dequeue - FIFO
current_job = print_queue.
    popleft()
# "doc1.pdf" prints first
```

Common Pitfalls

What NOT to Do

- × Using lists for frequent lookups
  (O(n) instead of O(1) with hash table)
- × Using arrays when size is unknown
  (Costly resizing)
- × Using BST when order doesn't matter
  (Hash table is faster)
- × Using recursive structures without considering stack overflow
  (Deep recursion can crash)

General Guidelines

- **Default to hash tables** for key-value storage (most versatile)
- **Use arrays** when you need index-based access
- **Use linked lists** when you have many insertions/deletions

Abstraction and ADTs

Abstract Data Types (ADT)

Definition

An **ADT** defines a data type by its **behavior** (operations) rather than its implementation. It specifies **what** operations can be performed, not **how**.

Key Principle

Separate the **interface** (what operations are available) from the **implementation** (how they work internally)

Benefits

1. **Flexibility**
   Change implementation without affecting users
2. **Simplicity**
   Hide internal complexity

Real-World Analogy

ADT = Car Interface

- Steering wheel
- Gas pedal
- Brake

ADT Example: Stack

Stack ADT Interface

```
class StackADT:
    def push(self, item): pass
    def pop(self): pass
    def peek(self): pass
    def is_empty(self): pass

# Behavior: LIFO
# (Last In, First Out)
```

Implementation 1: Array

```
class ArrayStack(StackADT):
    def __init__(self):
        self._data = []

    def push(self, item):
        self._data.append(item)

    def pop(self):
        return self._data.pop()
```

Implementation 2: Linked List

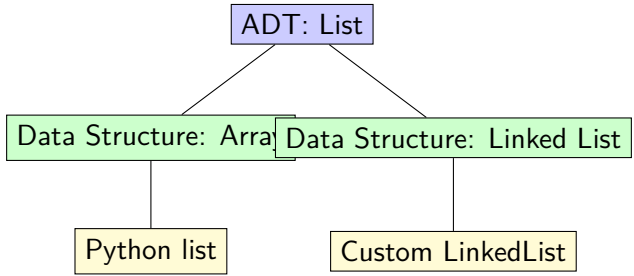
```
class LinkedStack(StackADT):
    def __init__(self):
        self._head = None
        self._size = 0

    def push(self, item):
        new_node = Node(item, self._head)
        self._head = new_node
        self._size += 1

    def pop(self):
        if not self._head:
            raise IndexError("empty")
        value = self._head.value
        self._head = self._head.next
        self._size -= 1
        return value

    def peek(self):
        if not self._head:
            raise IndexError("empty")
        return self._head.value
```

ADT Hierarchy



Three Levels

- **ADT** (Abstract): The concept/interface (e.g., "List")
- **Data Structure** (Logical): How data is organized (e.g., "Array" or "Linked List")
- **Implementation** (Physical): Actual code in a programming language

Encapsulation in ADTs

Good: Hidden Implementation

```
class Stack:
    def __init__(self):
        self._data = []  # Private

    def push(self, item):
        self._data.append(item)

    def pop(self):
        return self._data.pop()

# Users can't access _data directly
# Implementation can change!
```

✓ Implementation hidden
✓ Can swap to linked list easily
✓ Users can't break invariants

Bad: Exposed Implementation

```
class BadStack:
    def __init__(self):
        self.data = []  # Public!

    def push(self, item):
        self.data.append(item)

    def pop(self):
        return self.data.pop()

# Problem:
stack = BadStack()
stack.data = "oops!"  # Breaks it!
stack.data.clear()     # Ruins stack!
```

× Implementation exposed
× Users can break the stack
× Hard to change implementation

Summary

Key Takeaways

1. Data Structures Matter

- Right structure = milliseconds, Wrong structure = hours
- Choose based on most frequent operations
- Understand time vs space trade-offs

2. Big-O Analysis

- Describes growth rate, not exact time
- Focus on scalability for large inputs
- Remember: $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n)$
- Drop constants and lower-order terms

3. Common Operations

- Arrays: O(1) access, O(n) insert/delete in middle
- Hash tables: O(1) average lookup/insert/delete

Key Takeaways (continued)

4. Choosing Structures

- Ask: What operations are most frequent?
- Ask: Do you need ordering?
- Ask: What are your constraints (time/space)?
- Default to hash tables for key-value storage
- Use specialized structures when they match your pattern

5. Abstract Data Types

- Separate interface from implementation
- Hide internal details (encapsulation)
- Makes code flexible and maintainable
- Same interface, multiple implementations

Next Steps

Thank You!

Questions?

"Choosing the right data structure is half the battle in algorithm design."