Data Structures: Foundations

Data Structure Course
DGIST

November 1, 2025

Contents

Introduction

Why Foundations Matter

Course Philosophy

Build a strong programming foundation **before** diving into complex data structures

What You'll Master:

- Programming language fundamentals
- Control flow and recursion
- Memory models (stack vs heap)
- Debugging and testing skills
- Problem-solving patterns

Why This Matters:

- Data structures require solid coding skills
- Understanding memory is crucial
- Testing prevents subtle bugs
- Good habits accelerate learning

Key Principle

Implement every structure **from scratch** at least once to truly understand it

Choosing a Programming Language

Language Comparison

Language	Strengths	Best For	Learning Curve
C/C++	Maximum control, performance, STL	Competitive programming, systems	Steep (pointers, memory)
Java	Strong libraries, OOP, widely used	Interviews, enterprise	Moderate
Python	Readable, fast to write	Learning, prototyping	Easy

For Beginners

Python or **Java**

- Focus on concepts, not syntax
- Less memory management

For Performance/CP

C++

- Learn STL containers
- Manual memory control

Language-Specific Considerations

C++ Example

```cpp
#include <vector>
#include <iostream>

int main() {
    std::vector<int> arr = {1, 2, 3};
    arr.push_back(4);  // Dynamic resize

    // Manual memory for complex types
    int* ptr = new int[100];
    delete[] ptr;  // Must free!

    return 0;
}
```

Manual memory, explicit types

Python Example

```python
# Dynamic typing, GC
arr = [1, 2, 3]
arr.append(4)  # Auto-resize

# No manual memory management
# Objects cleaned up automatically

# But watch mutability!
list1 = [1, 2, 3]
list2 = list1  # Same reference
list2.append(4)
print(list1)  # [1, 2, 3, 4]
```

Automatic memory, dynamic types

Important

Data structures behave differently: manual memory in C/C++ vs garbage collection in Java/Python

Variables, Control Flow, and Loops

Core Concepts

Data Types & Scope

- Basic types: int, float, bool
- Type conversion and casting
- Variable scope and lifetime

Control Flow

```python
# If/else with early returns
def process(x):
    if x < 0:
        return "negative"
    elif x == 0:
        return "zero"
    else:
        return "positive"
```

Loop Patterns

- Index-based vs iterator-based
- Nested loops: $O(n^2)$, $O(n^3)$
- Break/continue usage
- Loop invariants

Common Pitfall

```python
# Off-by-one error!
for i in range(len(arr) - 1):  # Missing last!
    print(arr[i])

# Correct:
for i in range(len(arr)):
    print(arr[i])
```

Practice Problems

Nested Loops and Complexity

Time Complexity Examples

```python
# O(n) - Single loop
for i in range(n):
    print(i)

# O(n^2) - Nested loop
for i in range(n):
    for j in range(n):
        print(i, j)

# O(n^3) - Triple nested
for i in range(n):
    for j in range(n):
        for k in range(n):
            print(i, j, k)
```

Functions and Recursion

Function Fundamentals

Key Concepts

- Pass by value vs reference
- Return values and side effects
- Function purity
- Single responsibility principle

Best Practices

- Small, focused functions
- Clear, descriptive names
- Document preconditions
- Handle edge cases

Pure Function

```python
# Pure: no side effects
def add(a, b):
    return a + b

# Impure: modifies state
def append_item(lst, item):
    lst.append(item)  # Side effect!
```

C++ Reference

```cpp
// Pass by value (copy)
void func1(vector<int> v) {
    v.push_back(1);  // Original unchanged
}

// Pass by reference (no copy)
void func2(vector<int>& v) {
    v.push_back(1);  // Original modified
}
```

Recursion Basics

Essential Components

- **Base case**: Stopping condition (prevents infinite recursion)
- **Recursive case**: Progress toward base case
- **Stack frames**: Each call adds to call stack

Factorial

```
def factorial(n):
    # Base case
    if n <= 1:
        return 1
    # Recursive case
    return n * factorial(n - 1)

# Call stack for factorial(3):
# factorial(3) -> 3 * factorial(2)
# factorial(2) -> 2 * factorial(1)
# factorial(1) -> 1 (base case)
# Unwind: 2 * 1 = 2, then 3 * 2 = 6
```

Binary Search

```
def binary_search(arr, target, left, right):
    # Base case: not found
    if left > right:
        return -1

    mid = left + (right - left) // 2

    # Base case: found
    if arr[mid] == target:
        return mid

    # Recursive cases
    if arr[mid] > target:
        return binary_search(arr, target,
                             left, mid - 1)
```

Recursion Patterns and Pitfalls

Common Patterns

- **Divide and Conquer**:
  Merge sort, quick sort
- **Tree Traversals**:
  In-order, pre-order, post-order
- **Backtracking**:
  Permutations, N-Queens
- **Dynamic Programming**:
  Fibonacci with memoization

Common Pitfalls

- Missing base case
  $\rightarrow$ Infinite recursion
- Incorrect base case
  $\rightarrow$ Wrong results
- Deep recursion
  $\rightarrow$ Stack overflow
- No progress to base
  $\rightarrow$ Infinite loop

Solution for Deep Recursion

Convert to iterative with explicit stack

Practice

Arrays and Strings Fundamentals

Arrays: Core Properties

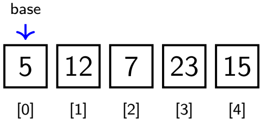
Key Characteristics

- Contiguous memory
- O(1) indexing
- Fixed size vs resizable
- Cache-friendly

Common Operations

Operation	Time
Access	O(1)
Search	O(n)
Insert (end)	O(1) amortized
Insert (middle)	O(n)
Delete (middle)	O(n)

Array Structure



Indices 0-4

Memory Address

`addr[i] = base + i × size`

Strings: Special Arrays

String Properties

- Arrays of characters
- Immutable (Java/Python) vs Mutable (C char arrays)
- Concatenation costs
- Substring operations

C++ (Mutable)

```
char s[] = "hello";
s[0] = 'H';  // OK: "Hello"

// C++ string class
string str = "hello";
str += " world";  // Efficient
str[0] = 'H';     // OK
```

Python (Immutable)

```
s = "hello"
# Can't modify: s[0] = 'H' # Error!

# Concatenation creates new string
s = s + " world"  # O(n)

# Better for multiple ops:
parts = []
parts.append("hello")
parts.append(" world")
```

Common Tasks

- Substring search
- Pattern matching
- Palindrome checking
- Reversal
- Character frequency

Memory Model: Stack vs Heap

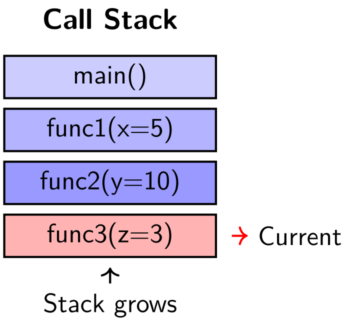
Stack Memory

Characteristics

- Stores function call frames
- Parameters and local variables
- Fast allocation/deallocation
- Limited size (typically 1-8 MB)
- Automatic cleanup
- LIFO (Last In, First Out)

Stack Overflow

Deep recursion can exhaust stack space
→ Convert to iteration or increase stack size



Heap Memory

Characteristics

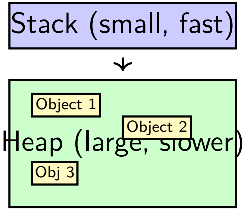
- Dynamic memory allocation
- Objects with longer lifetimes
- Larger capacity (GBs)
- Slower than stack
- Manual (C/C++) or GC (Java/Python)

Memory Management

C/C++: Manual
`malloc/free`, `new/delete`

Java/Python: Automatic
Garbage collection

Memory Layout



Pointers and References

C++ Pointers

```cpp
int x = 10;
int* ptr = &x;  // Pointer to x

*ptr = 20;      // Modify via pointer
cout << x;      // 20

// Heap allocation
int* arr = new int[100];
arr[0] = 5;
delete[] arr;   // Must free!

// Dangling pointer (BAD!)
int* p = new int(42);
delete p;
cout << *p;     // Undefined!
```

Python References

```python
# Objects are references
list1 = [1, 2, 3]
list2 = list1      # Same reference!

list2.append(4)
print(list1)       # [1, 2, 3, 4]

# To copy:
list3 = list1.copy()  # Shallow copy
list3.append(5)
print(list1)       # [1, 2, 3, 4]

# Deep copy for nested structures
import copy
nested = [[1, 2], [3, 4]]
deep = copy.deepcopy(nested)
```

Practice

C/C++: Allocate/free arrays, avoid leaks and dangling pointers

Java/Python: Understand when objects are shared and mutated

Debugging and Testing

Debugging Strategies

Essential Tools

- **Debugger**: Breakpoints, step through, watch variables
- **Assertions**: Capture invariants early
- **Logging**: Strategic print statements
- **Binary search**: Isolate bug location

Don't Just Print!

Use a proper debugger:

- Set breakpoints
- Inspect variable state
- Step line by line
- Understand execution flow

Debugging Process

1. **Reproduce**: Minimal test case
2. **Isolate**: Which function fails?
3. **Inspect**: Variable values at failure
4. **Hypothesize**: What could cause this?
5. **Test**: Verify hypothesis
6. **Fix**: Apply solution
7. **Validate**: Ensure fix works

Tools by Language

C/C++: gdb, lldb, valgrind
Java: JUnit, debugger
Python: pdb, unittest, pytest

Testing Best Practices

Testing Principles

- Start with unit tests
- Test edge cases
- Small and large inputs
- Property-based thinking
- Measure performance

Edge Cases to Test

- Empty input
- Single element
- Duplicates
- Negative numbers
- Boundary values

Python Unit Test

```python
import unittest

class TestArrayOps(unittest.TestCase):
    def test_reverse_normal(self):
        arr = [1, 2, 3, 4]
        reverse(arr)
        self.assertEqual(arr, [4, 3, 2, 1])

    def test_reverse_empty(self):
        arr = []
        reverse(arr)
        self.assertEqual(arr, [])

    def test_reverse_single(self):
        arr = [1]
        reverse(arr)
        self.assertEqual(arr, [1])

    def test_reverse_property(self):
        arr = [1, 2, 3]
        reverse(reverse(arr))
        self.assertEqual(arr, [1, 2, 3])

if __name__ == '__main__':
    unittest.main()
```

Using Online Judges

Online Judge Platforms

Popular Platforms

- **LeetCode**: Interview prep
- **HackerRank**: Competitions, hiring
- **Codeforces**: Competitive programming
- **AtCoder**: Japanese CP platform
- **TopCoder**: Algorithms, marathons

Benefits

- Immediate feedback
- Test against edge cases
- Compare solutions
- Track progress

Problem-Solving Approach

1. **Read** carefully, note constraints
2. **Design** algorithm with complexity
3. **Implement** cleanly
4. **Test** edge cases manually
5. **Submit** and iterate
6. **Reflect** on solution
7. **Document** patterns learned

Avoid Pitfalls

- Don't just copy solutions
- Re-implement after understanding
- Pay attention to constraints

Common Problem Patterns

Pattern	Description	Example Problems
Two Pointers	Use two indices moving through data	Palindrome, pair sum
Sliding Window	Fixed/variable size window	Max subarray sum
Fast & Slow Pointers	Detect cycles, find middle	Linked list cycle
Stack	LIFO for matching/parsing	Valid parentheses
BFS/DFS	Graph/tree traversal	Level-order, paths
Binary Search	Divide search space	Search sorted array

Summary

Key Takeaways

Language and Tools

- Choose one language and master it (Python/Java for learning, C++ for performance)
- Understand memory management for your language
- Learn debugging tools and use them effectively

Core Programming Skills

- Master control flow, loops, and avoid off-by-one errors
- Understand recursion: base case, recursive case, stack frames
- Know array and string fundamentals, including complexity

Memory Model

- Stack: fast, limited, automatic (function frames)
- Heap: larger, slower, manual or GC (dynamic objects)

Next Steps

Moving Forward

With these foundations in place, you're ready to:

- Study linear data structures (arrays, linked lists, stacks, queues)
- Implement each structure from scratch
- Analyze time and space complexity
- Apply structures to real-world problems
- Build toward more complex structures (trees, graphs, hash tables)

Recommended Practice

1. Implement basic array operations (reverse, rotate, search)
2. Write recursive solutions for factorial, Fibonacci, binary search
3. Practice two-pointer and sliding window problems
4. Solve 10-20 easy problems on LeetCode/HackerRank

Thank You!

Questions?

"The best way to learn data structures
is to implement them yourself."