

Introduction to Data Structures and Core Concepts

Lecture 01

Data Structures and Algorithms

What Are Data Structures?

Imagine a library:

- Books thrown randomly on the floor?
- Stacked in piles?
- Organized on shelves by categories?

Data structures are ways to organize and store data in a computer so it can be used efficiently.

Formal Definition:

A specialized format for organizing, processing, retrieving, and storing data.

Why Do We Need Data Structures?

Problem: Store names of 100 students

Approach 1: Create 100 separate variables

```
char student1[50], student2[50], student3[50], ..., student100[50];
```

Approach 2: Use an array

```
char students[100][50];
```

Data structures help us:

- Organize data logically
- Access data efficiently
- Manage memory effectively
- Simplify complex problems

The Data-Algorithm Relationship

Data Structures + Algorithms = Programs

Think of cooking:

- **Data structures** = how you organize ingredients (pantry, fridge, spice rack)
- **Algorithms** = the recipe (steps to transform ingredients)
- **Program** = the complete meal preparation

Classification of Data Structures

Classification 1: Primitive vs. Non-Primitive

Primitive Data Structures

Basic building blocks provided by the language:

- `int` - integers (-2, 0, 42)
- `float` - decimal numbers (3.14, -0.5)
- `char` - single characters ('a', 'Z', '7')
- `double` - larger decimal numbers

Non-Primitive Data Structures

Complex structures built using primitive types:

- Arrays, Structures, Linked Lists
- Stacks, Queues, Trees, Graphs

Classification 2: Linear vs. Non-Linear

Linear Data Structures

Elements arranged sequentially:

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

Linked List: [10|•]-->[20|•]-->[30|•]-->[40|NULL]

Stack: [30] ← Top

[20]

[10] ← Bottom

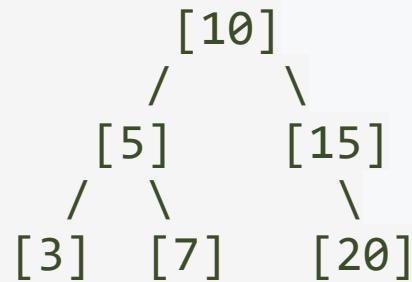
Queue: Exit ← [10] [20] [30] [40] ← Entry

Classification 2: Linear vs. Non-Linear (cont.)

Non-Linear Data Structures

Elements connected to multiple elements:

Tree (Hierarchical):



Graph (Network):



Classification 3: Static vs. Dynamic

Static Data Structures

- Fixed size at compile time
- Memory allocated when program starts

```
int numbers[10]; // Size is fixed at 10
```

- **Pros:** Faster access, simpler
- **Cons:** Wasteful, can't grow

Dynamic Data Structures

- Size changes during execution
- Memory allocated at runtime
- Example: Linked Lists

- **Pros:** Flexible, memory efficient

Importance in Programming

Performance Implications

Scenario: Find a student ID in 1,000,000 records

Unsorted Array + Linear Search:

- Check each element one by one
- Time: Up to 1,000,000 comparisons

Sorted Array + Binary Search:

- Divide and conquer
- Time: At most 20 comparisons ($\log_2 1,000,000 \approx 20$)

Hash Table:

- Direct access via computed index
- Time: Usually 1 comparison

Real-World Applications

Operating Systems

- Process Scheduling: Queues
- File Systems: Trees
- Memory Management: Linked Lists

Databases

- Indexing: B-trees
- Caching: Hash tables
- Query Processing: Graphs

Artificial Intelligence

- Search Algorithms: Trees and Graphs
- Neural Networks: Multi-dimensional Arrays
- Decision Making: Trees

Real-World Applications (cont.)

Web Development

- Browser History: Stack (back button)
- Social Networks: Graphs (friend connections)
- Autocomplete: Trees (tries)

Video Games

- Pathfinding: Graphs (shortest routes)
- Game State: Stacks (undo functionality)
- Collision Detection: Spatial data structures

Case Study: Music Streaming Service

Feature: "Recently Played" - show last 50 songs

Poor Choice - Unsorted Array:

- Search entire array for each new song
- If exists, remove and add to end
- If full, shift 49 elements
- Result: Slow

Better Choice - Circular Buffer + Hash Table:

- Circular buffer stores order
- Hash table for quick lookup
- Result: Fast insertions and lookups

Lesson: Right data structure = Better user experience

Basic Terminology

Core Terminology

1. Elements (or Nodes)

Individual items stored in a data structure

```
char students[5][20] = {"Alice", "Bob", "Charlie", "Diana", "Eve"};  
// "Alice", "Bob", etc. are elements
```

2. Operations

Actions performed on data structures:

- **Insertion** - Adding a new element
- **Deletion** - Removing an element
- **Traversal** - Visiting each element once
- **Searching** - Finding a specific element
- **Sorting** - Arranging elements in order
- **Updating** - Modifying an existing element

Core Terminology (cont.)

3. Data Organization

How elements are arranged and related

This determines:

- How we access elements
- How quickly we can perform operations
- How much memory is needed

Memory Management Fundamentals

Computer Memory: Simple Analogy

Memory is like an apartment building:

- Each apartment has a unique address (memory address)
- Each apartment stores one piece of information (a byte)
- You can only access apartments if you know their addresses

Storage Basics: Stack vs. Heap

The Stack

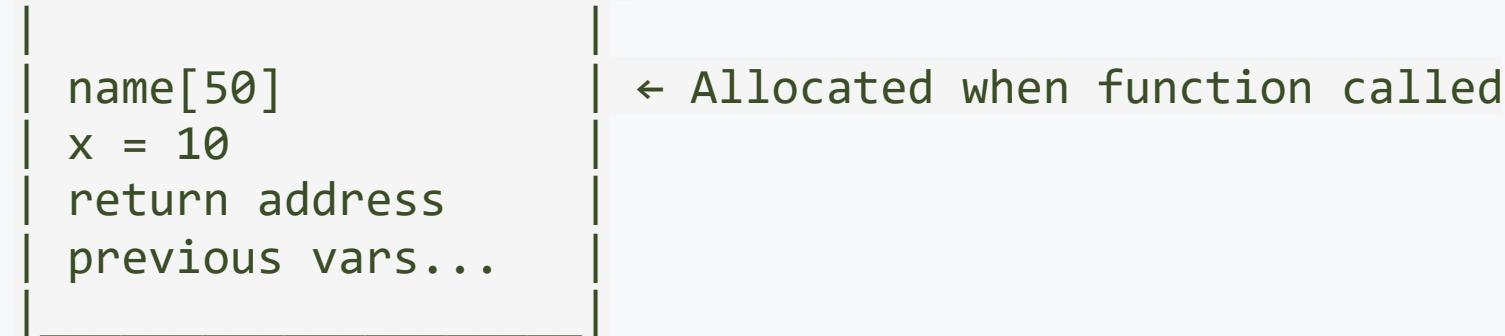
- Memory for local variables and function calls
- Automatically managed by compiler
- Limited, fixed size
- Very fast access
- Variables live only while function executes

```
void function() {  
    int x = 10;          // Stored on stack  
    char name[50];      // Stored on stack  
    // When function ends, memory automatically freed  
}
```

Storage Basics: Stack vs. Heap (cont.)

Stack Visualization:

Stack (grows downward):



Storage Basics: Stack vs. Heap (cont.)

The Heap

- Memory for dynamically allocated data
- Manually managed by programmer
- Much larger, can grow
- Slower access than stack
- Exists until explicitly freed

```
void function() {
    int *ptr = (int *)malloc(sizeof(int)); // Heap
    *ptr = 10;
    // You must free it when done
    free(ptr);
}
```

Heap Visualization:

Memory Allocation: Static vs. Dynamic

Static Allocation

- Size determined at compile time
- Memory allocated when program starts
- Cannot change size

```
int numbers[100]; // Always space for 100 integers  
// Even if you only use 10, space for 100 is reserved
```

Memory Allocation: Static vs. Dynamic (cont.)

Dynamic Allocation

- Size determined at runtime
- Memory allocated when needed
- Can request exactly the amount needed

```
int n;
printf("How many numbers? ");
scanf("%d", &n);
int *numbers = (int *)malloc(n * sizeof(int));
// Allocate exactly n integers

// Later, when done:
free(numbers);
```

Garbage Collection

The Problem: Memory Leaks

```
void bad_function() {
    int *ptr = malloc(100 * sizeof(int));
    // ... use ptr ...
    // Oops! Forgot to call free(ptr)
    // This memory is now leaked
}
// Called 1000 times = 1000 × 100 × 4 bytes = 400 KB lost
```

Garbage Collection:

Automatic memory management that reclaims unused memory

Important: C does **not** have automatic garbage collection

You must manually manage memory using `malloc()` and `free()`

Garbage Collection (cont.)

Best Practice in C:

```
void good_function() {
    int *ptr = malloc(100 * sizeof(int));
    if (ptr == NULL) {
        // Handle allocation failure
        return;
    }
    // ... use ptr ...
    free(ptr); // Always free what you allocate
}
```

Memory Compaction

The Problem: Fragmentation

Like a bookshelf with small gaps that can't fit a thick book:

Memory after many allocations/deallocations:
[Used][Free][Used][Free][Free][Used][Free][Used]
10 5 8 3 4 15 6 12

Total free: $5 + 3 + 4 + 6 = 18$ units

But can't allocate 10 continuous units!

Memory Compaction (cont.)

Compaction: Move allocated blocks together

After compaction:

[Used]	[Used]	[Used]	[Used]	[Free]
10	8	15	12		18 units	

Now you can allocate the 10 units needed!

Note: Compaction is complex and expensive

Typically handled by the operating system

More relevant in managed-memory languages

Problem-Solving Approaches

Iterative Approaches

Definition:

Repeating instructions using loops until a condition is met

Key Characteristics:

- Uses loops: `for`, `while`, `do-while`
- Explicitly manages loop counter
- Generally more memory efficient

Example 1: Sum of Array Elements

Problem: Find sum of array elements

```
int sum_array(int arr[], int n) {  
    int sum = 0;  
    for (int i = 0; i < n; i++) {  
        sum += arr[i];  
    }  
    return sum;  
}  
  
// Usage  
int numbers[] = {5, 10, 15, 20, 25};  
int total = sum_array(numbers, 5);  
// total = 75
```

Example 1: How It Works

Step 0: sum = 0

Step 1: sum = 0 + 5 = 5

Step 2: sum = 5 + 10 = 15

Step 3: sum = 15 + 15 = 30

Step 4: sum = 30 + 20 = 50

Step 5: sum = 50 + 25 = 75

Example 2: Finding Maximum Element

```
int find_max(int arr[], int n) {
    int max = arr[0]; // Assume first is maximum

    for (int i = 1; i < n; i++) {
        if (arr[i] > max) {
            max = arr[i]; // Update if larger found
        }
    }

    return max;
}
```

Recursive Approaches

Definition:

A function calls itself to solve a smaller version of the same problem

Key Components:

1. **Base Case:** Simplest case solved directly (stops recursion)
2. **Recursive Case:** Function calls itself with simpler input

Analogy:

In a queue, you ask the person in front for their position.

They ask the person in front of them.

Continues until someone at front says "I'm first!"

Then the answer propagates back.

Example 1: Factorial

Mathematical definition:

- $n! = n \times (n-1) \times (n-2) \times \dots \times 2 \times 1$
- $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

Recursive definition:

- $n! = n \times (n-1)!$
- $1! = 1$ (base case)

```
int factorial(int n) {
    // Base case
    if (n == 1 || n == 0) {
        return 1;
    }
    // Recursive case: n! = n × (n-1)!
    return n * factorial(n - 1);
}
```

How factorial(5) Works

```
factorial(5)
= 5 × factorial(4)
= 5 × (4 × factorial(3))
= 5 × (4 × (3 × factorial(2)))
= 5 × (4 × (3 × (2 × factorial(1))))
= 5 × (4 × (3 × (2 × 1)))           ← Base case
= 5 × (4 × (3 × 2))
= 5 × (4 × 6)
= 5 × 24
= 120
```

Function Call Visualization

```
factorial(5)
  ↓
  calls factorial(4)
  ↓
  calls factorial(3)
  ↓
  calls factorial(2)
  ↓
  calls factorial(1)
  ↓
  returns 1 ← Base case
  returns  $2 \times 1 = 2$ 
  returns  $3 \times 2 = 6$ 
  returns  $4 \times 6 = 24$ 
returns  $5 \times 24 = 120$ 
```

Example 2: Fibonacci Sequence

Sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21...

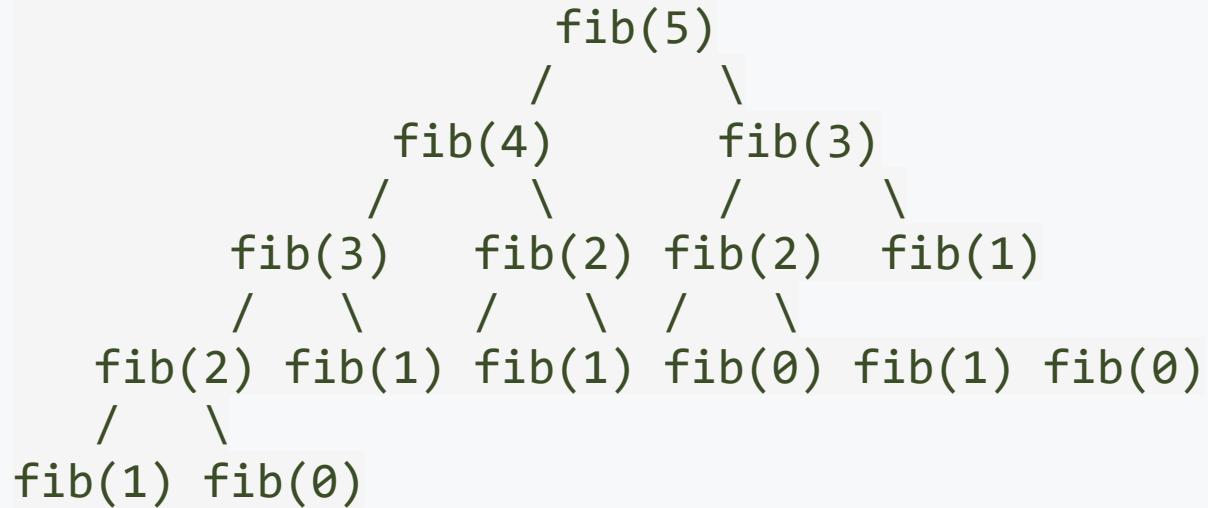
Rule:

- $F(0) = 0$
- $F(1) = 1$
- $F(n) = F(n-1) + F(n-2)$

```
int fibonacci(int n) {
    // Base cases
    if (n == 0) return 0;
    if (n == 1) return 1;

    // Recursive case
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Fibonacci Call Tree



Each $\text{fib}(1)$ returns 1, $\text{fib}(0)$ returns 0

Results combine: $\text{fib}(5) = 5$

When to Use Recursion vs. Iteration

Use Recursion When:

- Problem has natural recursive structure
- Solution is more elegant and understandable
- Examples: Tree traversals, quicksort, mergesort

Use Iteration When:

- Performance is critical (no function call overhead)
- Memory is limited (recursion uses stack)
- Iterative solution is straightforward
- Examples: Simple loops, array processing

Recursion vs. Iteration Comparison

Aspect	Iteration	Recursion
Code Clarity	Sometimes verbose	Often elegant
Memory Usage	Lower	Higher (stack)
Speed	Faster	Slightly slower
Risk	Infinite loops	Stack overflow
Best For	Simple traversals	Tree problems

Same Problem, Both Approaches

Sum of first n numbers:

```
// Iterative
int sum_iterative(int n) {
    int sum = 0;
    for (int i = 1; i <= n; i++) {
        sum += i;
    }
    return sum;
}

// Recursive
int sum_recursive(int n) {
    if (n == 1) return 1;          // Base case
    return n + sum_recursive(n - 1); // Recursive
}

// Both: sum(5) = 15
```

Key Takeaways

What We Learned Today

1. Data Structures Definition

- Organized ways to store and manage data
- Essential for efficient programming

2. Classification

- Primitive vs. Non-primitive
- Linear vs. Non-linear
- Static vs. Dynamic

3. Importance

- Direct impact on program performance
- Used everywhere in real-world applications

What We Learned Today (cont.)

4. Memory Concepts

- Stack vs. Heap
- Static vs. Dynamic allocation
- Importance of managing memory in C
- Garbage collection (not in C!)
- Memory compaction

5. Problem-Solving Approaches

- Iteration: Using loops to repeat operations
- Recursion: Functions calling themselves
- Each has its place depending on the problem

Review Questions

1. What is a data structure?
2. Give three examples of linear data structures.
3. What is the difference between stack and heap memory?
4. What is the base case in recursion, and why is it important?
5. When would you prefer iteration over recursion?
6. What is garbage collection, and does C provide it automatically?

