Data Structures & Algorithms

Structured Academic Curriculum

Designed by

D Charan Jeet

A Comprehensive Course Syllabus for Engineering Students

Covering Foundations, Linear & Non-Linear Structures, and Algorithms

Contents

Introduction

Data Structures and Algorithms (DSA) form the backbone of computer science and software engineering. A thorough understanding of DSA is indispensable for writing efficient, scalable, and optimized software systems. Whether designing the architecture of a high-performance database, solving complex computational problems in competitive programming, or preparing for technical interviews at top-tier technology companies, DSA knowledge serves as the foundational skill set that distinguishes proficient engineers.

This syllabus presents a carefully structured curriculum that progressively builds the student's understanding from fundamental concepts to advanced problem-solving patterns. The course is divided into four comprehensive modules:

1. **Foundations of Data Structures & Complexity** — Establishing core concepts, complexity analysis, and foundational data structures.
2. **Linear Data Structures** — Exploring linked lists, stacks, queues, and hashing with practical applications.
3. **Non-Linear Data Structures** — Delving into trees, binary search trees, heaps, and graphs.
4. **Algorithms & Problem-Solving Patterns** — Mastering searching, sorting, greedy strategies, backtracking, dynamic programming, and sliding window techniques.

Each module is designed with clear learning outcomes, well-organized topic hierarchies, and emphasis on key concepts that are essential for both academic excellence and industry readiness.

Chapter 1

Foundations of Data Structures & Complexity

Module Overview

This module establishes the conceptual groundwork for the entire course. Students will learn what data structures and algorithms are, how to analyze their efficiency, and explore foundational structures including arrays, strings, and recursion.

1.1 Introduction to DSA

- **What is a Data Structure?**
  - Definition and purpose of organizing data efficiently
  - Classification: primitive vs. non-primitive, linear vs. non-linear
- **What is an Algorithm?**
  - Formal definition and characteristics of algorithms
  - Properties: finiteness, definiteness, input, output, effectiveness
- **Why DSA matters in interviews & real systems**
  - Role in technical interview processes at leading companies
  - Importance in designing scalable and performant systems
- **Abstract Data Types (ADT)**
  - Concept of abstraction in data structures
  - Separation of interface from implementation

1.2 Time & Space Complexity

- **Big-O Notation**

  - Formal definition and asymptotic analysis
  - Upper bound representation of algorithm growth
- **Best, Worst, and Average Case Analysis**
  - Understanding different scenarios of input distribution
  - Practical significance of worst-case analysis
- **Common Complexities**
  - $O(1)$ — Constant time
  - $O(\log n)$ — Logarithmic time
  - $O(n)$ — Linear time
  - $O(n \log n)$ — Linearithmic time
  - $O(n^2)$ — Quadratic time
- **Basic Mathematical Analysis**
  - Summation techniques and recurrence relations

1.3 Arrays

- **Static vs. Dynamic Arrays**
  - Memory allocation: compile-time vs. run-time
  - Trade-offs between fixed and resizable storage
- **Traversal**
  - Sequential access patterns and iteration techniques
- **Insertion & Deletion**
  - Positional operations and shifting overhead
  - Complexity analysis for different positions
- **Prefix Sum Concept**
  - Precomputation for range query optimization
- **Two Pointer Technique** (basic introduction)
  - Concept of converging pointers for efficient scanning

1.4 Strings

- **String Traversal**
  - Character-by-character processing
- **Palindrome Checking**
  - Iterative and recursive approaches
- **Frequency Counting**
  - Using arrays and hash maps for character frequency
- **Basic Pattern Matching**
  - Naïve string matching algorithm
  - Introduction to efficient pattern search concepts

1.5 Recursion (Important Foundation)

- **Recursive Thinking**
  - Decomposing problems into smaller subproblems
  - Identifying recursive structure in problems
- **Base Case & Recursive Case**
  - Termination conditions and progress toward base case
- **Stack Memory Concept**
  - Call stack behavior during recursive execution
  - Stack overflow and depth considerations
- **Classic Recursive Problems:**
  - Factorial computation
  - Fibonacci sequence generation
  - Sum of digits
  - Reverse an array recursively

Outcome

Student understands how to analyze problems, evaluate algorithmic complexity, and build foundational logic using arrays, strings, and recursion.

Chapter 2

Linear Data Structures

Module Overview

This module explores linear data structures that store elements in sequential order. Students will study linked lists, stacks, queues, and hashing, gaining insight into memory management, pointer manipulation, and efficient data retrieval.

2.1 Linked List

- **Singly Linked List**
  - Node structure: data and next pointer
  - Traversal and basic operations
- **Doubly Linked List**
  - Bidirectional traversal with previous and next pointers
- **Circular Linked List**
  - Circular connectivity and applications
- **Insert / Delete Operations**
  - Insertion and deletion at beginning, end, and specific positions
  - Complexity analysis for each operation
- **Reverse a Linked List**
  - Iterative and recursive reversal techniques
- **Detect Cycle**
  - Floyd's cycle detection algorithm (Tortoise and Hare)

2.2 Stack

- **Stack using Array**
  - Array-based implementation with push, pop, and peek
- **Stack using Linked List**
  - Dynamic memory allocation for stack operations
- **Applications:**
  - **Parenthesis Checking** — Balanced brackets validation
  - **Infix to Postfix Conversion** — Expression transformation
  - **Next Greater Element** — Monotonic stack approach

2.3 Queue

- **Simple Queue**
  - FIFO principle and basic operations
- **Circular Queue**
  - Efficient use of fixed-size buffer
- **Deque** (Double-Ended Queue)
  - Insertion and deletion from both ends
- **Priority Queue** (introduction)
  - Element ordering based on priority values
- **Applications:**
  - **BFS Concept Preview** — Queue-driven graph traversal
  - **Sliding Window Maximum** — Deque-based optimization

2.4 Hashing

- **Hash Table Basics**
  - Hash function design and key-to-index mapping
- **HashMap / Dictionary Concept**

  - Key-value pair storage and retrieval in $O(1)$ average time

- **Collision Handling**
  - **Chaining** — Linked list at each bucket
  - **Open Addressing** — Linear probing, quadratic probing
- **Frequency-Based Problems**
  - Counting occurrences and pattern detection using hash maps

Outcome

Student understands memory-linked data structures, pointer manipulation, and real-world applications of linear data structures in system design and problem solving.

Chapter 3

Non-Linear Data Structures

Module Overview

This module introduces hierarchical and network-based data structures. Students will explore trees, binary search trees, heaps, and graphs — structures essential for modeling complex relationships and solving optimization problems.

3.1 Trees

- **Binary Tree**
  - Node structure with left and right children
  - Properties: full, complete, perfect, balanced binary trees
- **Tree Traversals:**
  - **Inorder** — Left → Root → Right
  - **Preorder** — Root → Left → Right
  - **Postorder** — Left → Right → Root
  - **Level Order** — Breadth-first traversal using queue
- **Height, Depth, and Leaf Nodes**
  - Definitions and computation methods
  - Relationship between height and node count
- **Diameter of Tree** (introductory logic)
  - Longest path between any two nodes
  - Recursive computation approach

3.2 Binary Search Tree (BST)

- **Insert / Delete / Search Operations**
  - Maintaining the BST property during modifications
  - Average and worst-case complexity analysis
- **Valid BST Check**
  - In-range validation using recursive bounds
- **Lowest Common Ancestor (LCA)**
  - Leveraging BST properties for efficient LCA computation

3.3 Heap

- **Min Heap / Max Heap**
  - Heap property and complete binary tree representation
  - Array-based storage of heap structure
- **Heap Operations**
  - Insertion (heapify up) and extraction (heapify down)
  - Building a heap from an unsorted array — $O(n)$ analysis
- **Kth Largest Element Problem**
  - Min-heap approach for efficient selection

3.4 Graphs

- **Graph Representation:**
  - **Adjacency Matrix** — Dense graph storage
  - **Adjacency List** — Sparse graph storage
  - Trade-offs in space and time complexity
- **Breadth-First Search (BFS)**
  - Level-by-level exploration using a queue
  - Applications: shortest path in unweighted graphs

- **Depth-First Search (DFS)**
  - Recursive and iterative implementations
  - Applications: path finding, topological concepts
- **Cycle Detection** (basic)
  - Detecting cycles in directed and undirected graphs
- **Connected Components**
  - Identifying disjoint subgraphs using BFS/DFS

Outcome

Student can model, traverse, and solve problems involving hierarchical and network-based data structures, including trees, heaps, and graphs.

Chapter 4

Algorithms & Problem-Solving Patterns

Module Overview

This module focuses on core algorithmic strategies and problem-solving paradigms. Students will master searching, sorting, greedy approaches, backtracking, dynamic programming, and sliding window techniques — skills essential for competitive programming and technical interviews.

4.1 Searching Algorithms

- **Linear Search**
  - Sequential scan with $O(n)$ complexity
- **Binary Search**
  - Divide-and-conquer on sorted data with $O(\log n)$ complexity
  - Iterative and recursive implementations
- **Binary Search on Answer**
  - Applying binary search to optimization problems
  - Identifying monotonic predicates for search space reduction

4.2 Sorting Algorithms

- **Bubble Sort**
  - Repeated adjacent swaps — $O(n^2)$ complexity
- **Selection Sort**
  - Finding minimum element iteratively — $O(n^2)$ complexity

- **Insertion Sort**
  - Building sorted subarray incrementally — efficient for small inputs
- **Merge Sort**
  - Divide-and-conquer with $O(n \log n)$ guaranteed performance
  - Stable sort with additional space requirements
- **Quick Sort**
  - Partition-based sorting with $O(n \log n)$ average case
  - Pivot selection strategies and worst-case mitigation
- **Comparison of Sorting Techniques**
  - Time complexity, space complexity, and stability comparison
  - Choosing the appropriate sorting algorithm for different scenarios

4.3 Greedy Algorithms

- **Activity Selection Problem**
  - Selecting maximum non-overlapping activities
  - Greedy choice property and optimal substructure
- **Coin Change** (greedy case)
  - Minimum coins using greedy strategy for canonical systems
- **Fractional Knapsack**
  - Maximizing value with fractional item selection
  - Value-to-weight ratio optimization

4.4 Recursion to Backtracking

- **Subsets Generation**
  - Generating all subsets using recursive inclusion/exclusion
- **Permutations**
  - Generating all arrangements using swap-based recursion

- **N-Queens Problem** (conceptual)
  - Constraint satisfaction and pruning strategies
  - Backtracking framework for placement problems

4.5 Introduction to Dynamic Programming

- **Memoization** (Top-Down Approach)
  - Caching recursive subproblem solutions
  - Reducing exponential time to polynomial time
- **Tabulation** (Bottom-Up Approach)
  - Iterative table construction for subproblem solutions
- **Classic Dynamic Programming Problems:**
  - **Fibonacci DP** — Transforming $O(2^n)$ to $O(n)$
  - **Climbing Stairs** — Counting distinct paths
  - **0/1 Knapsack** (introductory idea) — Optimal item selection with constraints

4.6 Sliding Window & Two Pointer Pattern

- **Fixed Window**
  - Maintaining a window of constant size across an array
  - Maximum/minimum sum subarray of size k
- **Variable Window**
  - Expanding and shrinking window based on conditions
  - Smallest subarray with a given sum
- **Subarray Problems**
  - Longest substring without repeating characters

Outcome

Student has developed strong algorithmic thinking and can apply searching, sorting, greedy, backtracking, dynamic programming, and sliding window strategies to solve complex computational problems efficiently.