Chapter 26: Advanced Data Structures (e.g., Trees, Graphs)

Introduction

As programs grow in complexity and deal with large-scale data, basic structures like arrays and linked lists become insufficient for efficient data manipulation and storage. **Advanced data structures** such as **trees**, **heaps**, **tries**, and **graphs** enable optimal solutions for complex problems including parsing expressions, route finding, database indexing, compiler construction, and much more.

This chapter explores these structures in depth—how they are designed, how they work, and how they are implemented in real-world systems. You will also learn about the time and space complexity of various operations on these structures and see code-level examples and algorithmic applications.

26.1 Trees

26.1.1 Overview of Trees

A **tree** is a hierarchical data structure with a root node and sub-nodes (children), where each node (except the root) has exactly one parent. It is an abstract model of hierarchical structures.

Key Terms:

- **Root**: Topmost node.
- **Leaf**: Node with no children.
- **Internal Node**: Node with at least one child.
- **Depth**: Length of the path from root to the node.
- **Height**: Longest path from the node to a leaf.

26.1.2 Binary Trees

A **binary tree** is a tree in which each node has at most two children, typically called **left** and **right**.

Traversal Methods:

- **In-order (LNR)**: Left → Node → Right
- **Pre-order (NLR)**: Node → Left → Right
- **Post-order (LRN)**: Left → Right → Node
- **Level-order**: Breadth-first traversal using a queue.

26.1.3 Binary Search Trees (BSTs)

A **Binary Search Tree** maintains a sorted structure:

- Left subtree contains nodes with values **less than** the parent node.
- Right subtree contains nodes with values **greater than** the parent node.

Operations:

- **Insert**: O(log n) average
- **Search**: O(log n) average
- **Delete**: O(log n) average (*Worst-case for unbalanced trees: O(n)*)

26.1.4 Balanced Trees

AVL Tree (Adelson-Velsky and Landis)

- A self-balancing BST.
- Balance factor (height left - height right) must be in [-1, 0, 1].

Red-Black Tree

- A binary tree with nodes marked red or black.
- Ensures O(log n) time for insertion, deletion, and lookup.

26.1.5 Heaps

A **heap** is a complete binary tree used to implement priority queues.

- **Min-Heap**: Parent ≤ children
- **Max-Heap**: Parent ≥ children

Operations:

- **Insert**: O(log n)
- **Extract-Min/Max**: O(log n)
- **Build-Heap**: O(n)

Used in **Heap Sort** and **Dijkstra's algorithm**.

26.1.6 Tries (Prefix Trees)

- A tree-based data structure for storing strings, used especially for autocomplete and spell checking.
- Each node represents a character of the string.
- Fast lookup: O(length of word)

26.2 Graphs

26.2.1 Introduction to Graphs

A **graph** is a non-linear data structure consisting of **vertices (nodes)** and **edges (connections)**. It can be:

- **Directed** or **Undirected**
- **Weighted** or **Unweighted**
- **Cyclic** or **Acyclic**

26.2.2 Representation of Graphs

1. **Adjacency Matrix**:
    - 2D array: `matrix[i][j] = 1` if edge exists.
    - Space: $O(V^2)$
2. **Adjacency List**:
    - Each vertex stores a list of adjacent vertices.
    - Space: O(V + E)

26.2.3 Graph Traversal

Breadth-First Search (BFS):

- Uses a queue.
- Explores neighbors level by level.
- Time: O(V + E)

Depth-First Search (DFS):

- Uses a stack or recursion.
- Explores one branch deeply before backtracking.
- Time: O(V + E)

26.2.4 Applications of Graphs

- **Shortest Path** (Dijkstra, Bellman-Ford)
- **Cycle Detection**
- **Topological Sorting** (for DAGs)
- **Minimum Spanning Tree** (Kruskal's, Prim's)
- **Network Flow** (Ford-Fulkerson)

26.2.5 Dijkstra's Algorithm (Shortest Path)

Used in weighted graphs (non-negative edges) to find the shortest path from a source to all vertices.

- Time: O((V + E) log V) with Min-Heap

26.2.6 Minimum Spanning Tree

Prim's Algorithm

- Starts from any node, adds the cheapest edge to the growing MST.
- Uses priority queue.

Kruskal's Algorithm

- Sorts all edges and adds the smallest edge that doesn't form a cycle.

- Uses Union-Find.

26.3 Comparative Analysis of Data Structures

Structure	Use Case	Avg. Time Complexity	Space
Binary Search Tree	Sorted data, fast lookup	O(log n)	O(n)
AVL / Red-Black Tree	Self-balancing trees	O(log n)	O(n)
Heap	Priority queue, scheduling	O(log n)	O(n)
Trie	String search, autocomplete	O(k), k = word length	High
Graph (Adj List)	Real-world networks	O(V + E) traversal	O(V + E)

26.4 Real-World Applications

- **Trees**: Compilers (parse trees), AI (decision trees), File systems.
- **Heaps**: Task scheduling, bandwidth management, event-driven simulators.
- **Tries**: Search engines, IP routing, dictionary implementations.
- **Graphs**: Social networks, navigation systems, recommendation systems.

Summary

Advanced data structures like trees and graphs empower developers to solve non-trivial problems with optimal time and space efficiency. Mastery of these structures, along with their algorithms and use cases, is essential for tackling complex software challenges in fields ranging from databases and networking to machine learning and artificial intelligence.

By understanding and applying trees, heaps, tries, and graphs, you unlock the ability to architect systems that are not only functional but scalable, efficient, and intelligent.
