UNIT 4

**1. Self-Balancing Trees**

Self-balancing trees are a type of **binary search tree (BST)** that automatically keeps their height (depth) small to maintain **efficient operations** (insert, delete, search) in **O(log n)** time.

**Types of Self-Balancing Trees:**

* **AVL Tree** (Adelson-Velsky and Landis)
  + Balancing is done using **rotation** after insertion/deletion.
  + Balance factor of each node is either -1, 0, or +1.
* **Red-Black Tree**
  + Every node has a color (red or black), and the tree ensures properties like no two consecutive red nodes.
  + Allows faster insertions/deletions than AVL in practice but more relaxed balance.
* **Splay Tree**
  + Uses "splaying" (rotating recently accessed element to the root) to keep frequently accessed elements near the top.
* **Treap** (Tree + Heap)
  + Combines properties of binary search tree and heap using priority values.

**2. B-Trees**

A **B-tree** is a self-balancing search tree designed for **systems that read and write large blocks of data**, like databases and file systems.

**Properties:**

* Generalization of a binary search tree where a node can have more than 2 children.
* Each node contains **keys** and **pointers to children**.
* Nodes are kept in sorted order.
* **Balanced**, meaning all leaf nodes are at the same depth.
* High branching factor reduces tree height, minimizing disk I/O.

**Order m B-Tree:**

* Each node can have **at most m children** and **at least ⌈m/2⌉ children** (except root).
* Internal nodes contain between ⌈m/2⌉ and m children.

**3. B+ Trees**

**B+ Trees** are an extension of B-Trees, commonly used in **databases and file systems**.

**Key Differences from B-Trees:**

* **All data is stored in leaf nodes**.
* Internal nodes only store keys used for routing.
* Leaf nodes are **linked** for efficient **range queries** (sorted linked list).
* Better suited for **sequential access** and **indexing large datasets**.

**4. Single Source Shortest Path Algorithms**

These algorithms find the **shortest path** from a **single source** vertex to all other vertices in a **weighted graph**.

**Key Algorithms:**

**a. Dijkstra's Algorithm**

* Works for graphs with **non-negative edge weights**.
* Greedy approach using **priority queue (min-heap)**.
* Time complexity: O((V + E) log V)

**b. Bellman-Ford Algorithm**

* Works with **negative weights**.
* Slower than Dijkstra but detects **negative weight cycles**.
* Time complexity: O(V × E)

**c. SPFA (Shortest Path Faster Algorithm)**

* Improvement over Bellman-Ford using a queue.
* Efficient in practice but worst-case same as Bellman-Ford.

**5. All-Pairs Shortest Path Algorithms**

These algorithms compute the shortest paths between **every pair of vertices** in a graph.

**Key Algorithms:**

**a. Floyd-Warshall Algorithm**

* Dynamic programming algorithm.
* Works for **negative weights**, but not negative cycles.
* Time complexity: O(V³)
* Good for **dense graphs**.

**b. Johnson’s Algorithm**

* Handles graphs with negative weights.
* Combines **Bellman-Ford + Dijkstra**.
* Reweights edges to eliminate negative weights.
* Efficient for **sparse graphs**: O(V² log V + VE)

**6. Network Flow Algorithms**

These algorithms deal with problems where **flow is pushed through a network**, such as **maximum flow**, **minimum cut**, or **bipartite matching**.

**Key Algorithm:**

**a. Ford-Fulkerson Algorithm**

* Finds **maximum flow** in a flow network.
* Uses **augmenting paths** and **residual graphs**.
* Complexity depends on path-finding strategy (O(max\_flow × E))

**b. Edmonds-Karp Algorithm**

* Specific implementation of Ford-Fulkerson using **BFS** to find shortest augmenting paths.
* Time complexity: O(VE²)

**c. Dinic's Algorithm**

* Uses level graphs and blocking flows.
* Efficient for dense graphs: O(V²E)

**Applications:**

* Network routing
* Image segmentation
* Bipartite graph matching
* Project selection

UNIT 6

**✅ 1. Approximation Algorithms**

**🔍 What is it?**

An **approximation algorithm** is used to find **near-optimal solutions** to **optimization problems** (usually NP-hard) where finding the exact solution is computationally expensive or impossible in a reasonable time.

**💡 Why use them?**

* For **NP-Hard** problems (e.g., Traveling Salesman, Knapsack), **exact algorithms** are too slow.
* Approximation gives a solution that’s **"good enough"** within some **guaranteed bound**.

**🧮 Approximation Ratio:**

Let:

* C be the cost of the solution by the approximation algorithm.
* C\* be the cost of the optimal solution.

Then the **approximation ratio** is:

* *Max(C / C)*\* for minimization problems
* *Min(C / C)*\* for maximization problems

**🛠 Examples:**

* **Vertex Cover**: 2-approximation algorithm exists.
* **Knapsack Problem**: Fully Polynomial Time Approximation Scheme (FPTAS).
* **TSP (metric)**: 1.5-approximation using Christofides' algorithm.

**🎲 2. Randomized Algorithms**

**🔍 What is it?**

An algorithm that makes **random choices** during execution to improve performance or simplicity.

**🔁 Two Types:**

1. **Las Vegas Algorithm**
   * Always produces **correct output**, but runtime may vary.
   * Example: Randomized QuickSort.
2. **Monte Carlo Algorithm**
   * May produce **incorrect results** with small probability, but usually runs faster.
   * Example: Randomized primality tests (Miller-Rabin).

**✅ Advantages:**

* Simpler logic
* Faster in average case
* Good for **large inputs or distributed systems**

**🧪 Example:**

* **QuickSort (Random Pivot):** Improves performance and avoids worst-case O(n²).
* **Karger's Algorithm:** Finds a minimum cut in a graph using edge contractions randomly.

**🧠 3. Class of Problems Beyond NP – PSPACE**

**🔍 What is PSPACE?**

* **PSPACE** is the class of decision problems that can be solved using **polynomial amount of memory**, regardless of time.
* It may take **exponential time**, but not more than polynomial **space**.

**📚 Relationships:**

P ⊆ NP ⊆ PSPACE ⊆ EXPTIME

**⚖️ PSPACE-Complete Problems:**

* Hardest problems in PSPACE (analogous to NP-Complete in NP).
* Examples:
  + Quantified Boolean Formula (QBF)
  + Generalized Chess/Go/Tic-Tac-Toe (for n×n boards)

**⏳ Why is it important?**

* Many problems in **game theory**, **AI**, and **logic** fall into PSPACE.
* It tells us that **even if a problem is not in NP**, it might still be solvable with limited memory.

**⚛️ 4. Introduction to Quantum Algorithms**

**🔍 What is it?**

Quantum algorithms run on **quantum computers**, leveraging principles of quantum mechanics like **superposition** and **entanglement**.

**💡 Why quantum algorithms?**

They can **solve some problems faster** than classical algorithms.

**⚙️ Key Quantum Algorithms:**

| **Algorithm** | **Use** | **Classical Time** | **Quantum Time** |
| --- | --- | --- | --- |
| **Shor’s Algorithm** | Integer factorization (RSA breaking) | Exponential | Polynomial |
| **Grover’s Algorithm** | Unstructured search | O(n) | O(√n) |

**🧬 Quantum Complexity Classes:**

* **BQP** (Bounded-error Quantum Polynomial time) — quantum version of class P.
* Not fully understood how BQP compares to NP, PSPACE, etc.

**💭 Applications:**

* Cryptography
* Search and optimization
* Quantum simulations (chemistry, physics)

**🧵 5. Parallel Algorithms**

**🔍 What is it?**

Algorithms designed to **run on multiple processors/cores simultaneously**, aiming to solve problems **faster** by breaking them into **independent tasks**.

**💡 Why use them?/**

* Efficient use of modern hardware (multi-core CPUs, GPUs, clusters).
* Huge performance gains for large data sets.

**📚 Key Concepts:**

**a. Speedup (S)**

S=TserialTparallelS = \frac{T\_{serial}}{T\_{parallel}}

**b. Amdahl’s Law**

Defines the **maximum speedup** possible using parallel processing: S=1(1−P)+PNS = \frac{1}{(1 - P) + \frac{P}{N}}  
Where:

* *P* = parallel portion of the algorithm
* *N* = number of processors

**c. PRAM Model**

Parallel Random Access Machine – a theoretical model used to analyze parallel algorithms.

**🧠 Examples:**

* **Parallel Merge Sort**
* **Matrix Multiplication**
* **MapReduce (Big Data)**

**🧾 Summary Table:**

| **Topic** | **Key Idea** | **Example** |
| --- | --- | --- |
| **Approximation Algorithms** | Near-optimal solution to NP-Hard problems | Vertex Cover (2-approx) |
| **Randomized Algorithms** | Random decisions in logic | QuickSort, Miller-Rabin |
| **PSPACE** | Problems solvable in polynomial space | QBF |
| **Quantum Algorithms** | Uses quantum bits and logic | Shor’s Algorithm |
| **Parallel Algorithms** | Multiple tasks run concurrently | Parallel Matrix Multiplication |