Prathamesh Arvind Jadhav

DSA for Placement

Ch-1: Introduction

1. Introduction

Data Structures and Algorithms (DSA) is a foundational subject in computer science.
It helps us organize and process data efficiently.

- **Data**: Any piece of information (like numbers, characters, files).
- **Structure**: A way to organize data logically and efficiently.

Together, **data structure** means organizing data so that it can be accessed and modified easily.

- **Algorithm**: A step-by-step method to solve a problem or perform a task (like searching or sorting data).

Why DSA?

- Improves **time** and **space** efficiency of programs.
- Crucial for **problem-solving** and **coding interviews**.
- Forms the base for many **real-world applications** (e.g., databases, OS, websites, etc.).

□ 2. Basic Terminology

Understanding some fundamental terms:

Term	Explanation
Data	Raw facts and figures (e.g., 10, 'A', 4.5)
Data Structure	A way of organizing and storing data (e.g., arrays, linked lists, trees)
Data Type	Tells what kind of data is being used (e.g., int, char, float)
Field	A single piece of data (like name, age in a student record)
Record	A collection of related fields (like a row in a table)
File	A collection of related records stored on disk
Operations	Actions performed on data structures (insert, delete, search, etc.)

□ 3. Elementary Data Organization

This refers to how data is **stored, accessed, and related** to each other.

- Data can be organized as:
  - **Linear structures**: Arrays, Linked Lists, Stacks, Queues
  - **Non-linear structures**: Trees, Graphs
- **Memory representation** is crucial: data structures are stored in RAM, and efficient layout saves time.

Example:

- An **array** stores data in contiguous memory.
- A **linked list** stores elements scattered in memory with links to the next node.

□ 4. Built-in Data Types in C

C language provides **primitive (built-in)** data types:

Data Type	Description	Example
int	Integer numbers	int x = 10;
float	Floating point (decimal) numbers	float pi = 3.14;
double	Double-precision float	double g = 9.8;
char	Character data	char ch = 'A';
void	No value	void function()

Additionally, we can build **user-defined data types** like:

- struct
- union
- enum
- typedef

These are helpful in creating complex data structures (like stacks, trees, etc.).

□ 5. Abstract Data Types (ADT)

An **Abstract Data Type (ADT)** is a **theoretical model** that defines what a data structure does, **not how** it does it.

Characteristics:

- Focuses on **what operations** are possible (insert, delete, search, etc.)
- Hides internal details (like implementation with arrays or linked lists)

- Promotes **modular programming**

Common ADTs:

ADT	Description
List	Ordered collection of items
Stack	LIFO (Last In First Out) structure
Queue	FIFO (First In First Out) structure
Deque	Double-ended queue
Tree	Hierarchical data structure
Graph	Nodes connected by edges
Map/Dictionary	Key-value pairs for fast lookups

Ch-2: Arrays

1. What is an Array?

Definition: An array is a linear data structure used to store multiple elements of the **same data type** in a **contiguous block of memory**.

➢ Why is this important in placements?

- It's the **foundation** for understanding memory layout, time complexity, and advanced structures (like trees, heaps, graphs).
- Often used in **problem-solving** for optimization and iteration.

---

2. Key Features of Arrays

Feature	Explanation
Fixed size	Size is defined at compile time (e.g., int arr[100];)
Random Access	Accessing arr[i] takes **O(1)** time
Homogeneous	Only one type of data per array (e.g., only integers)
Indexing	Arrays use **zero-based indexing** in C++

---

3. Types of Arrays

One-Dimensional Array (1D)

- Simple linear list.

```
int arr[5] = {1, 2, 3, 4, 5};
```

Two-Dimensional Array (2D)

- Used for matrices or grids.

```
int mat[2][3] = {{1,2,3}, {4,5,6}};
```

Multi-Dimensional Array (3D, 4D…)

- Rare in practice, used in scientific computing.

---

4. Array Memory Layout (Row-Major Order)

C++ stores multi-dimensional arrays in **row-major order**, i.e., row-by-row.

➢ **Example:**

```
int A[2][3] = {
  {1, 2, 3},
  {4, 5, 6}
};
```

Memory will be: 1 2 3 4 5 6

➢ **Formula for 2D array:**

LOC(A[i][j]) = Base + ((i * N) + j) * W

- Base = starting address
- N = total columns
- W = size of data type (in bytes)

---

5. Array Operations & Time Complexities

Operation	Description	Time Complexity
Traversal	Visiting all elements	O(n)
Insertion (end)	Add element at end (if space available)	O(1)
Insertion (middle)	Insert at position (need to shift)	O(n)
Deletion (end)	Remove last element	O(1)
Deletion (middle)	Remove from position (shift required)	O(n)
Searching (Linear)	Check each element	O(n)
Searching (Binary)	Only on sorted arrays	O(log n)
Updation	Change value at index	O(1)

---

6. Applications of Arrays in Real Life

- **Storage of static data**: Marks, prices, temperature.
- **Sorting Algorithms**: Bubble sort, Merge sort, Quick sort.
- **Matrix operations**: Used in image processing, ML.
- **Used in DS**: Arrays are the base of stacks, queues, heaps, etc.
- **Hashing**: Often uses arrays as buckets or hash tables.
- **Game Development**: Grids, boards, maps.

7. Advantages & Disadvantages

Advantages:

- Fast access using index (O(1))
- Easy to implement and understand
- Great for problems requiring static memory

Disadvantages:

- Fixed size – wastes memory or causes overflow
- Insertion/deletion is costly (O(n) time)
- Requires contiguous memory block

---

8. When to Use Arrays?

Use when:

- Number of elements is **known and fixed**.
- **Fast access** is required.
- Simple operations like search, sort, and traversal are needed.

Use other structures (e.g., vectors, linked lists) when:

- Size changes frequently
- Insertion/deletion is common

Ch-3: Linked lists

1. What is a Linked List?

A **Linked List** is a **linear data structure** in which elements are stored in **nodes**, and each node contains:

- **Data**
- A **pointer** (or reference) to the **next node**

It is **dynamic in size**, unlike arrays.

```
struct Node {
   int data;
   Node* next;
};
```

---

2. Why Use Linked List?

Arrays	Linked List
Fixed size	Dynamic size
Contiguous memory allocation	Non-contiguous memory allocation
Costly insert/delete operations	Efficient insert/delete (O(1) at head)

Arrays	Linked List
Random access	Sequential access only

---

3. Types of Linked Lists

Type	Description & Structure
Singly Linked List	Each node has a pointer to the next node only.
Doubly Linked List	Each node has two pointers: one to next, one to previous.
Circular Linked List	Last node points back to the head (first node). Can be singly or doubly circular.

---

4. Structure in C++

Singly Linked List:

```
struct Node {
    int data;
    Node* next;
};
```

Doubly Linked List:

```
struct Node {
    int data;
    Node* prev;
    Node* next;
};
```

---

5. Basic Operations and Their Time Complexity

Operation	Description	Time Complexity
Traversal	Visit all nodes	O(n)
Insertion at Head	Add new node at start	O(1)

Operation	Description	Time Complexity
Insertion at Tail	Add new node at end	O(n) (SLL), O(1) (DLL if tail pointer exists)
Insertion at Position	Insert node at a given index	O(n)
Deletion at Head	Remove first node	O(1)
Deletion at Tail	Remove last node	O(n) (SLL), O(1) (DLL with tail)
Search (Iterative)	Find node with given value	O(n)

---

6. Linked List Applications

- **Dynamic Memory Allocation**
- **Implementing Stacks and Queues**
- **Polynomial Arithmetic**
- **Graphs and Adjacency List**
- **Undo functionality in editors**
- **Browser history**

---

7. Advantages of Linked List

Dynamic size
Efficient insertions and deletions (especially at beginning)
Memory efficient for sparse data
No memory wastage (no need for resizing)

---

8. Disadvantages

No random access (O(n) to find element)
Extra memory for pointers
More complex than arrays
Cache performance is poor (due to non-contiguous memory)

Ch-4: Stack & Queue

1. What is a Stack?

- A **Stack** is a **linear data structure** which follows the **LIFO (Last In First Out)** principle.
- Elements are inserted (pushed) and removed (popped) **only from the top**.
- Think of it like a stack of plates: you can only take the top plate or put a new plate on top.

---

2. Basic Terminology

Term	Meaning
Push	Add an element to the top of the stack.
Pop	Remove the top element from the stack.
Top/Peek	Retrieve the top element without removing it.
Underflow	Trying to pop from an empty stack.
Overflow	Trying to push into a full stack (only in fixed-size stacks).

---

3. Stack Representation

- Can be implemented using:
  - **Arrays** (fixed size)
  - **Linked List** (dynamic size)
  - **STL stack container** in C++ (recommended for quick coding)

---

4. Operations on Stack and Time Complexity

Operation	Description	Time Complexity
Push	Insert an element at the top	O(1)
Pop	Remove the top element	O(1)
Peek	Get the top element without removing	O(1)

Operation	Description	Time Complexity
IsEmpty	Check if stack is empty	O(1)
IsFull	Check if stack is full (array implementation)	O(1)

---

5. Linked List based Stack Implementation (Dynamic Size)

- Each node points to the next.
- Push and Pop happen at the head (top).

---

6. Applications of Stack

Application	Description
Expression Evaluation	Infix, Prefix, Postfix evaluation using stacks.
Expression Conversion	Convert Infix to Postfix or Prefix.
Backtracking Algorithms	Maze solving, puzzles.
Function Call Management	System call stack (function recursion).
Undo Mechanism	Text editors, browsers (back button).
Balanced Parentheses	Check for balanced delimiters in code.
Syntax Parsing	Used in compilers and interpreters.

Queue

1. What is a Queue?

- A **Queue** is a **linear data structure** that follows the **FIFO (First In First Out)** principle.
- Elements are inserted at the **rear (end)** and removed from the **front (start)**.
- Think of a queue like a line of people waiting for a service: the person who comes first is served first.

---

2. Basic Terminology

Term	Meaning
Enqueue	Insert an element at the rear of the queue.
Dequeue	Remove an element from the front of the queue.
Front	The first element in the queue (to be dequeued next).
Rear	The last element inserted in the queue.
Underflow	Trying to dequeue from an empty queue.
Overflow	Trying to enqueue into a full queue (array implementation).

---

3. Queue Representation

- Can be implemented using:
  - **Arrays** (fixed size)
  - **Linked Lists** (dynamic size)
  - **STL queue container** in C++ (recommended for quick coding)

---

4. Operations on Queue and Time Complexity

Operation	Description	Time Complexity
Enqueue	Add element at the rear	O(1)
Dequeue	Remove element from the front	O(1)
Front	Get the front element without removing	O(1)
IsEmpty	Check if queue is empty	O(1)
IsFull	Check if queue is full (array)	O(1)

---

5. Array-based Queue

- Simple queue uses a fixed-size array.
- Problem: After several dequeues, front moves forward but space behind is wasted.
- To solve this, use **Circular Queue**.

---

6. Circular Queue

- Circular queue solves the problem of wasted space by connecting the end of the queue back to the front.
- Rear and front indices wrap around using modulo operator.
- Efficient use of storage.

---

7. Linked List-based Queue

- Uses a linked list with pointers to front and rear nodes.
- Dynamic size, no overflow unless memory is full.
- Enqueue at rear, dequeue from front.
- Time complexity for operations: O(1).

---

8. Types of Queues

Type	Description
Simple Queue	Linear FIFO queue.
Circular Queue	Rear wraps to front to reuse empty space.
Deque (Double-Ended Queue)	Insertion and deletion at both ends.
Priority Queue	Elements dequeued based on priority, not order.

---

9. Applications of Queue

Application	Description
CPU Scheduling	Process scheduling in operating systems.

Application	Description
Buffer Handling	IO Buffers, keyboard buffers.
Breadth-First Search (BFS)	Graph and tree traversals.
Printer Queue	Managing print jobs.
Call Center Systems	Managing incoming calls in order.

Ch-5: Searching and Sorting

1. Searching

Searching means finding the position/index of a given element (key) in a data structure (usually an array or list).

Types of Searching

Searching Type	Description	Time Complexity (Average)
Linear Search	Check each element sequentially.	O(n)
Binary Search	Search in a **sorted** array by dividing the search space in half each time.	O(log n)

1.1 Linear Search

- Simple and straightforward.
- Start from the first element and compare with key until found or end reached.
- Works on **unsorted or sorted** data.
- Time Complexity: O(n)
- Space Complexity: O(1)

1.2 Binary Search

- Works only on **sorted arrays**.
- Find the middle element, compare with key:
    - If equal, return index.
    - If key < middle, search left half.
    - If key > middle, search right half.
- Recurse or iterate until element found or search space is empty.
- Time Complexity: O(log n)
- Space Complexity: O(1) (iterative), O(log n) (recursive)

---

2. Sorting

Sorting is arranging data in ascending or descending order. Sorting helps improve searching and data organization.

Common Sorting Algorithms and Their Complexities

Algorithm	Time Complexity (Avg)	Time Complexity (Worst)	Space Complexity	Stable?	Notes
Bubble Sort	$O(n^2)$	$O(n^2)$	O(1)	Yes	Simple but inefficient
Selection Sort	$O(n^2)$	$O(n^2)$	O(1)	No	Simple, fewer swaps than bubble
Insertion Sort	$O(n^2)$	$O(n^2)$	O(1)	Yes	Efficient for nearly sorted data
Merge Sort	O(n log n)	O(n log n)	O(n)	Yes	Divide & conquer, stable
Quick Sort	O(n log n)	$O(n^2)$	O(log n)	No	Divide & conquer,

Algorithm	Time Complexity (Avg)	Time Complexity (Worst)	Space Complexity	Stable?	Notes
					faster in practice
Heap Sort	O(n log n)	O(n log n)	O(1)	No	Uses heap data structure
Counting Sort	O(n + k)	O(n + k)	O(k)	Yes	For integers in limited range
Radix Sort	O(d*(n + k))	O(d*(n + k))	O(n + k)	Yes	For integers, stable sort

(n = number of elements, k = range of input, d = number of digits in max number)

2.1 Bubble Sort

- Repeatedly swap adjacent elements if they are in wrong order.
- After each pass, the largest element moves to the end.
- Simple but slow for large data.

2.2 Selection Sort

- Select the minimum element from unsorted part and swap with the first unsorted element.
- Simple but inefficient for large data.

2.3 Insertion Sort

- Build sorted array one element at a time by inserting the current element into the correct position.
- Efficient for small or nearly sorted arrays.

2.4 Merge Sort

- Divide array into two halves, recursively sort both halves, then merge them.
- Stable and efficient with guaranteed O(n log n).

2.5 Quick Sort

- Choose a pivot, partition array so that left elements < pivot, right > pivot.
- Recursively apply to left and right partitions.
- Average O(n log n) but worst case $O(n^2)$ if pivot poorly chosen.

2.6 Heap Sort

- Build max heap, then swap root with last element and heapify the reduced heap.
- In-place, not stable.

3. When to Use Which Sorting Algorithm?

Scenario	Recommended Sorting Algorithm
Small or nearly sorted data	Insertion Sort
Large unsorted data	Quick Sort or Merge Sort
Stable sort needed	Merge Sort, Insertion Sort
Limited range integers	Counting Sort or Radix Sort
Memory constraints	Quick Sort or Heap Sort

Ch-6: Tree

1. What is a Tree?

- A **Tree** is a non-linear hierarchical data structure that consists of nodes connected by edges.
- It is a collection of nodes where:
  - One node is the **root** (starting point).
  - Each node can have zero or more child nodes.
  - There are no cycles (no node is revisited).
- Used to represent hierarchical data like file systems, organizational structures, XML/HTML DOM, etc.

2. Basic Terminology

Term	Description
Node	Basic unit containing data and references to children.
Root	Top node with no parent.
Edge	Connection between parent and child nodes.
Parent	Node connected above a node.
Child	Node connected below a node.
Leaf Node	Node with no children.
Internal Node	Node with at least one child.
Subtree	Tree consisting of a node and its descendants.
Height of Node	Number of edges on the longest path from that node to a leaf.
Height of Tree	Height of the root node.
Depth of Node	Number of edges from the root to that node.
Degree of Node	Number of children of a node.

3. Types of Trees

- **Binary Tree**: Each node has at most 2 children (left and right).
- **Binary Search Tree (BST)**: Binary tree with the left child < parent < right child property.
- **Balanced Tree**: Height is minimized to ensure operations are efficient (e.g., AVL Tree, Red-Black Tree).
- **Full Binary Tree**: Every node has 0 or 2 children.
- **Complete Binary Tree**: All levels except possibly last are fully filled, and nodes are as left as possible.
- **Perfect Binary Tree**: All internal nodes have two children and all leaves are at the same level.
- **Heap Tree**: Complete binary tree with heap property (max or min heap).

- **Trie, Segment Tree, Fenwick Tree**: Specialized trees for specific applications.

---

4. Representation of Trees

4.1 Using Nodes and Pointers (Linked Representation)

- Each node contains:
  - Data
  - Pointer/reference to left child
  - Pointer/reference to right child
- Dynamic size and easy to represent sparse trees.

4.2 Using Arrays (for Complete Binary Trees)

- Parent-child relationships are derived by indices:
  - Parent at index i
  - Left child at 2*i + 1
  - Right child at 2*i + 2
- Efficient for complete or nearly complete binary trees.

---

5. Tree Traversals

Traversal is visiting all nodes in a specific order.

Traversal	Order Description	Use Case
Preorder	Root → Left → Right	Copy tree, prefix expression
Inorder	Left → Root → Right	Retrieve sorted data in BST
Postorder	Left → Right → Root	Delete tree, postfix expression
Level-order	Visit nodes level by level (BFS)	Shortest path, serialization

---

6. Applications of Trees

- **Hierarchical Data**: File systems, org charts.
- **Binary Search Trees**: Fast searching, insertion, deletion (O(log n) average).
- **Expression Trees**: Represent arithmetic expressions.

- **Heaps**: Priority queues.
- **Tries**: Efficient string retrieval.
- **Syntax trees**: Compilers.
- **Routing tables** in networks.

---

7. Complexity of Tree Operations (Binary Search Tree)

Operation	Average Time Complexity	Worst Case Complexity
Search	O(log n)	O(n)
Insertion	O(log n)	O(n)
Deletion	O(log n)	O(n)
Traversal (all nodes)	O(n)	O(n)

Note: Worst case occurs if tree becomes skewed (like a linked list). Balanced trees avoid this.

Ch-7: Graphs

1. What is a Graph?

- A **Graph** is a non-linear data structure consisting of a set of **vertices (nodes)** and a set of **edges** connecting pairs of vertices.
- Used to represent relationships, networks like social networks, maps, computer networks, dependency graphs, etc.

---

2. Basic Terminology

Term	Description
Vertex (Node)	Fundamental unit representing an entity in a graph.
Edge	Connection between two vertices.
Adjacent Vertices	Two vertices connected directly by an edge.
Degree of Vertex	Number of edges connected to a vertex.

Term	Description
Path	Sequence of vertices where each adjacent pair is connected by an edge.
Cycle	Path where the first and last vertices are the same, with at least one edge.
Connected Graph	There is a path between every pair of vertices.
Disconnected Graph	Some vertices cannot be reached from others.
Weighted Graph	Edges have weights or costs.
Directed Graph (Digraph)	Edges have direction (from one vertex to another).
Undirected Graph	Edges have no direction (bidirectional).

---

3. Types of Graphs

Graph Type	Description
Undirected Graph	Edges do not have direction.
Directed Graph	Edges have direction (arcs).
Weighted Graph	Edges carry weights (e.g., distances, costs).
Unweighted Graph	Edges have no weights.
Simple Graph	No loops or multiple edges between the same vertices.
Multigraph	Multiple edges allowed between the same pair of vertices.
Complete Graph	Every vertex is connected to every other vertex.

---

4. Graph Representation

4.1 Adjacency Matrix

- A 2D array adj[V][V], where V is the number of vertices.
- adj[i][j] = 1 (or weight) if edge exists from vertex i to vertex j, else 0.
- Easy to implement.
- Space complexity: $O(V^2)$.
- Good for dense graphs.

4.2 Adjacency List

- Each vertex stores a list of adjacent vertices.
- Space efficient for sparse graphs.
- Faster to iterate over neighbors.
- Space complexity: O(V + E), where E = number of edges.

---

5. Graph Traversal Algorithms

5.1 Depth-First Search (DFS)

- Explores as far as possible along a branch before backtracking.
- Uses recursion or stack.
- Useful for connectivity, cycle detection, topological sorting.

5.2 Breadth-First Search (BFS)

- Explores all neighbors at current depth before going deeper.
- Uses queue.
- Finds shortest path in unweighted graphs.
- Useful for level-order traversal.

---

6. Important Graph Concepts

6.1 Cycle Detection

- Detect cycles using DFS or Union-Find.
- In directed graphs, detect cycles by tracking recursion stack.
- Important for topological sorting and deadlock detection.

6.2 Connected Components

- Number of disconnected subgraphs in an undirected graph.
- Can be found using DFS or BFS.

6.3 Topological Sorting

- Linear ordering of vertices such that for every directed edge u → v, u comes before v.
- Applicable only for Directed Acyclic Graphs (DAGs).
- Used in scheduling tasks, build systems.

---

7. Weighted Graph Algorithms

7.1 Dijkstra's Algorithm

- Finds shortest path from a source to all vertices in a graph with non-negative weights.
- Uses priority queue (min-heap).
- Time complexity: $O((V + E) \log V)$.

7.2 Bellman-Ford Algorithm

- Finds shortest path even with negative weights.
- Detects negative weight cycles.
- Time complexity: $O(V * E)$.

7.3 Floyd-Warshall Algorithm

- Finds shortest paths between all pairs of vertices.
- Dynamic programming approach.
- Time complexity: $O(V^3)$.

---

8. Special Graph Structures

- **Minimum Spanning Tree (MST)**: Connect all vertices with minimum total edge weight without cycles.
  - Algorithms: Kruskal's, Prim's.
- **Bipartite Graph**: Vertices can be divided into two sets such that edges only go between sets, no edges within a set.
- **Planar Graph**: Can be drawn on a plane without edges crossing.

---

9. Applications of Graphs

- Social Networks (friends, followers).
- Network Routing.
- Dependency Resolution (package management).
- GPS Navigation and Maps.
- Scheduling and Task Ordering.

- Cycle detection in deadlocks.
- Web Crawlers.

---

10. Complexity Summary

Operation	Adjacency Matrix	Adjacency List
Space	$O(V^2)$	O(V + E)
Add Edge	O(1)	O(1)
Remove Edge	O(1)	O(E)
Check Edge Exists	O(1)	O(degree of vertex)
Traverse Neighbors	O(V)	O(degree of vertex)

Ch-8: Hashing

1. What is Hashing?

- **Hashing** is a technique to map data of arbitrary size (like strings, keys) to fixed-size values (usually integers), called **hash codes** or **hash values**.
- The hash code is used as an **index** in an array (called a **hash table**) to store the actual data.
- Main goal: Provide **fast data retrieval**, ideally **O(1)** time complexity for search, insert, and delete operations.

---

2. Hash Table

- A **hash table** is a data structure that implements an associative array or dictionary, mapping keys to values using a hash function.
- It contains:
  - **Buckets or slots**: Storage locations indexed by hash codes.
  - **Hash function**: Computes the bucket index for each key.

---

3. Hash Function

- A hash function takes input (key) and returns an integer (hash code).

- Good hash function properties:
  - **Deterministic:** Same input always yields the same output.
  - **Uniform distribution:** Minimizes collisions by spreading keys evenly.
  - **Fast computation.**

Example: For integer keys, a simple hash function can be:
hash(key) = key % table_size

---

4. Collisions in Hashing

- Collision occurs when **two different keys hash to the same index**.
- Since hash table size is limited, collisions are inevitable.

---

5. Collision Resolution Techniques

5.1 Separate Chaining

- Each bucket contains a **linked list** (or another data structure) of elements hashed to the same index.
- Insert the new key at the head/tail of the list.
- Search requires scanning the linked list.
- Simple to implement.

Pros:

- Easy to handle collisions.
- Table size can be small.

Cons:

- Extra memory for pointers.
- Search time can degrade to O(n) in worst case.

5.2 Open Addressing

- All elements are stored **within the hash table array itself**.
- On collision, probe other slots using a probing sequence until an empty slot is found.

Types of probing:

- **Linear probing:** Check next slots one by one (hash + i) % table_size.
- **Quadratic probing:** Use quadratic function (hash + i^2) % table_size.
- **Double hashing:** Use a second hash function for probing.

Pros:

- No extra memory for pointers.
- Cache-friendly due to contiguous memory.

Cons:

- Clustering problems (especially linear probing).
- Table must not be too full (load factor < 0.7) for efficiency.

---

6. Load Factor (α)

- Load factor is the ratio of number of elements to the number of buckets:

$$\alpha = \frac{\text{Number of elements}}{\text{Number of buckets}}$$

- A high load factor means more collisions.
- Generally, resize and rehash the table when α exceeds a threshold (e.g., 0.7).

---

7. Rehashing

- When the load factor is too high, increase table size (usually double) and re-insert all keys.
- Expensive but necessary for performance.
- New table size is usually a prime number to reduce collisions.

---

8. Applications of Hashing

- Implementing dictionaries, sets, and maps.
- Database indexing.
- Caching and memoization.
- Password storage (using cryptographic hash functions).
- Detecting duplicates.
- Symbol tables in compilers.

9. Hashing in C++ STL

- unordered_map and unordered_set use hashing internally.
- Average time complexity for insert, search, delete is O(1).
- Under the hood, uses separate chaining or open addressing depending on implementation.

10. Time Complexity Summary

Operation	Average Case	Worst Case (With Poor Hash Function or High Load)
Search	O(1)	O(n)
Insert	O(1)	O(n)
Delete	O(1)	O(n)