Applied CS Concepts

1. Linux, Git or Version Control
2. SDLC and Agile
3. Big-O Notation
4. Tree and Graphs
5. Stack, Queue, Hash & Heaps
6. Sorting and Searching
7. Popular Algorithms
8. Database Concept
9. SQL
10. REST API

1. Linux

- **Open-source, Unix-like OS** is known for stability and flexibility.
- **Key Concepts**:
  - **Kernel**: Core that interacts with hardware.
  - **Shell**: Command-line interface (CLI) like Bash and Zsh.
  - **Filesystem**: Hierarchical structure starting with root (/).
  - **Permissions**: Control access (read, write, execute) for users/groups.
- **Common Commands**:
  - ls: List files.
  - cd: Change directory.
  - cp: Copy files.
  - rm: Remove files.
  - chmod: Change file permissions.
  - ps: View active processes.

2. Git (Version Control)

- **Distributed Version Control System** used to track code changes.
- **Key Concepts**:
  - **Repository**: Stores project files and history.
  - **Commit**: Snapshot of changes with a unique ID.
  - **Branch**: Parallel line of development.
  - **Merge**: Combine changes from branches.
  - **Clone**: Copy of a remote repo.
  - **Pull/Push**: Fetch or send changes to/from remote repo.
- **Basic Git Commands**:
  - git init: Initialize a repo.
  - git clone <url>: Clone remote repo.
  - git add <file>: Stage file for commit.

  - git commit -m "message": Commit changes.
  - git push: Push commits to remote repo.
  - git pull: Fetch and merge remote changes.
  - git branch: Manage branches.
  - git merge <branch>: Merge branches.

1. SDLC (Software Development Life Cycle)

- **Phases**: Steps followed to develop software from concept to deployment.
  - **Requirements Gathering**: Understand client needs.
  - **System Design**: Plan architecture and technology stack.
  - **Implementation (Coding)**: Actual development of the system.
  - **Testing**: Ensure software works as expected.
  - **Deployment**: Release the software to production.
  - **Maintenance**: Update and fix issues post-deployment.
- **Models**:
  - **Waterfall**: Linear, sequential approach.
  - **Iterative**: Repeated cycles (or iterations) for incremental development.
  - **V-Model**: Verification and validation at each stage.
  - **Spiral**: Focuses on risk analysis at every phase.

2. Agile Methodology

- **Iterative and Incremental Approach** for software development.
- **Key Principles**:
  - **Customer collaboration**: Continuous involvement from stakeholders.
  - **Responding to change**: Flexibility to adapt to changing requirements.
  - **Working software**: Deliver functioning software frequently.
  - **Individuals and interactions**: Focus on motivated, self-organizing teams.
- **Agile Frameworks**:
  - **Scrum**: Iterative, time-boxed sprints (2-4 weeks).
  - **Kanban**: Continuous flow and visual task management.
  - **Extreme Programming (XP)**: Emphasizes technical excellence and continuous feedback.
- **Benefits**:
  - **Faster Delivery**: Frequent releases.
  - **Flexibility**: Easy to adapt to changes.
  - **Customer Satisfaction**: Continuous feedback from clients.

Big-O Notation:

Big-O notation describes the efficiency of an algorithm, focusing on how time or space grows with input size.

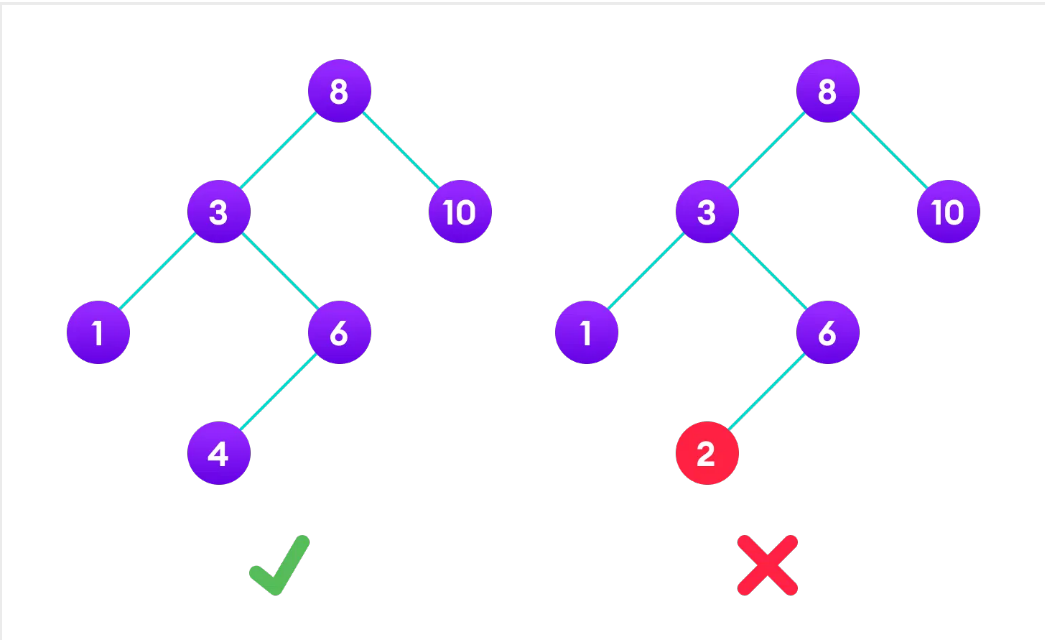
- **O(1)**: Constant time. Takes the same time regardless of input size.
- **O(n)**: Linear time. Time grows directly with input size.
- **O(n²)**: Quadratic time. Time grows exponentially with input size (e.g., nested loops).
- **O(log n)**: Logarithmic time. Time grows slower as input increases (e.g., binary search).
- **O(n log n)**: Linearithmic time. Found in efficient sorting algorithms.

Asymptotic Notation:

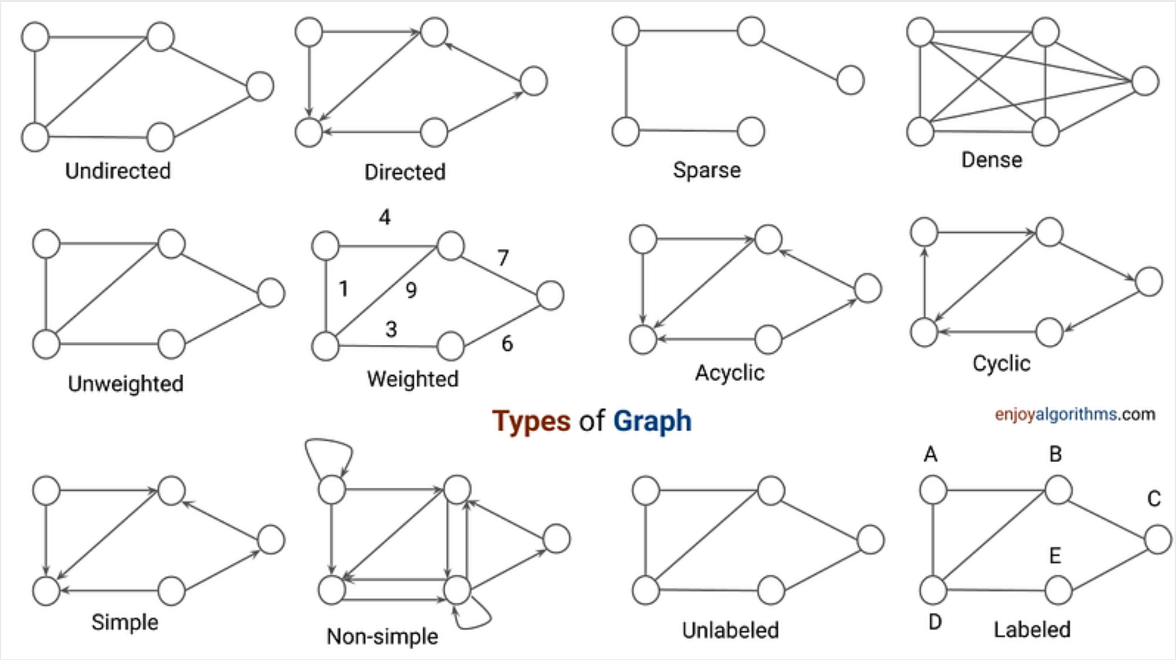
- **Big-O (O)**: Worst-case upper bound (e.g., **O(n²)**).
- **Big-Ω (Ω)**: Best-case lower bound (e.g., **Ω(n)**).
- **Big-Θ (Θ)**: Exact bound, both upper and lower (e.g., **Θ(n log n)**).

Trees:

- A tree is a hierarchical structure with nodes and edges, where each node has one parent (except the root) and zero or more children.
- **Binary Tree**: Each node has at most two children.
- **Binary Search Tree (BST)**: Left child is smaller, right child is larger than the parent node.



Graphs:

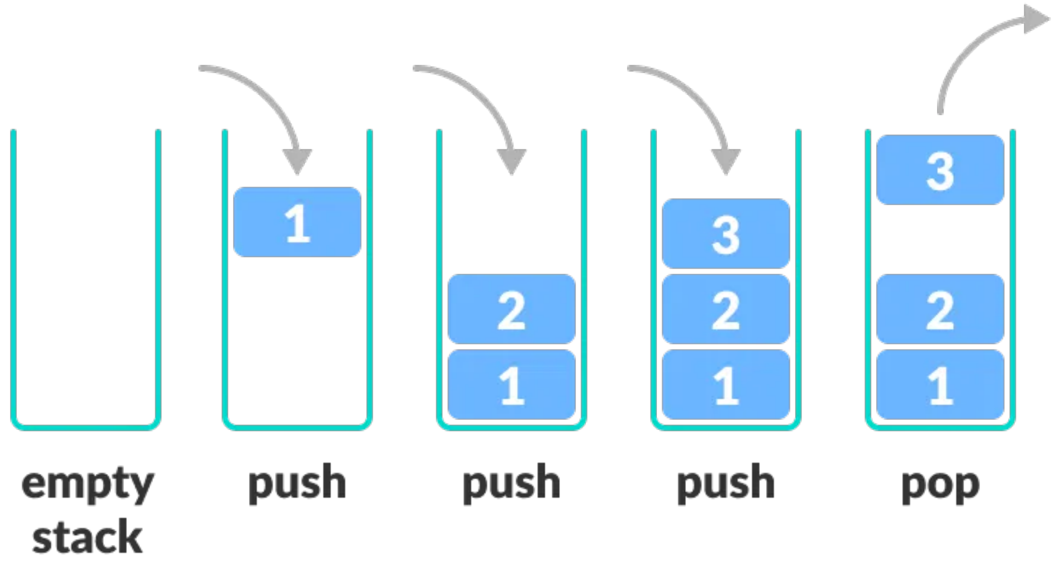


- A graph is a collection of nodes (vertices) and edges (connections between nodes).
- **Directed Graph (Digraph)**: Edges have a direction (e.g., A → B).
- **Undirected Graph**: Edges have no direction (e.g., A — B).
- **Weighted Graph**: Edges have weights (values).

Key Operations:

- **Traversal**: Visiting all nodes (e.g., DFS, BFS).
- **Shortest Path**: Finding the shortest route (e.g., Dijkstra's algorithm).

Stack



- **Definition**: A linear data structure that follows the **LIFO** (Last In, First Out) principle.
- **Operations**:
  - **Push**: Add an element to the top.
  - **Pop**: Remove the top element.
  - **Peek/Top**: View the top element without removing it.
- **Use Cases**: Undo operations in text editors, function calls in recursion, parsing expressions.

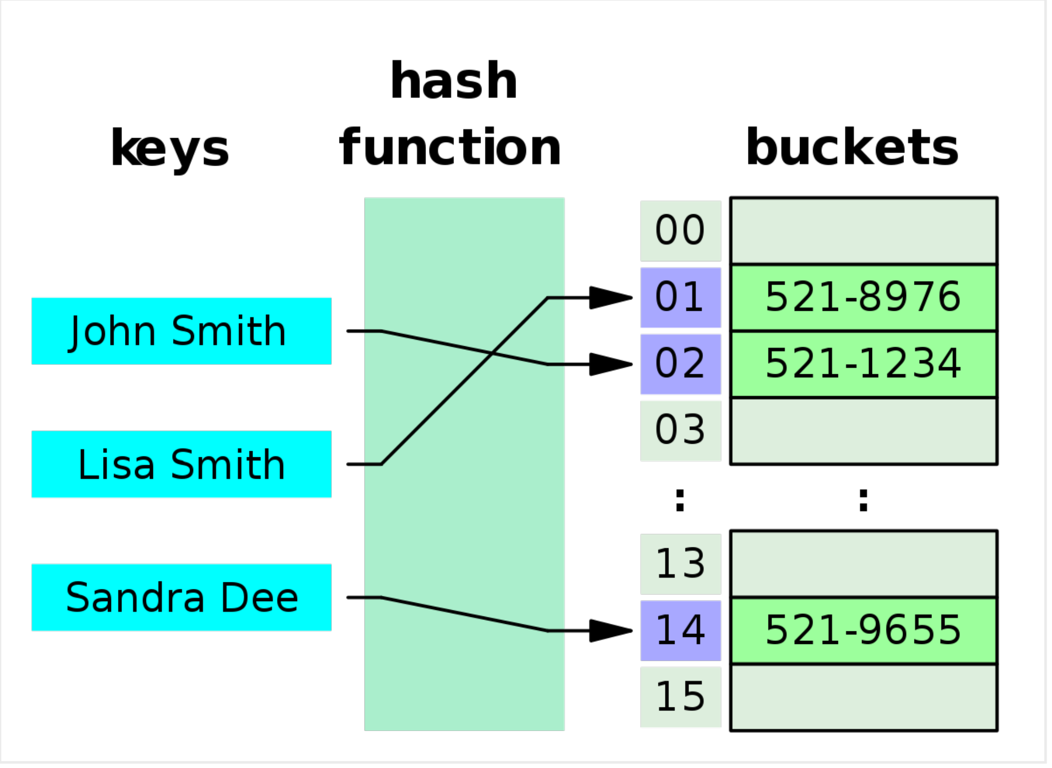
Queue

Queue Data Structure

- **Definition**: A linear data structure that follows the **FIFO** (First In, First Out) principle.
- **Operations**:

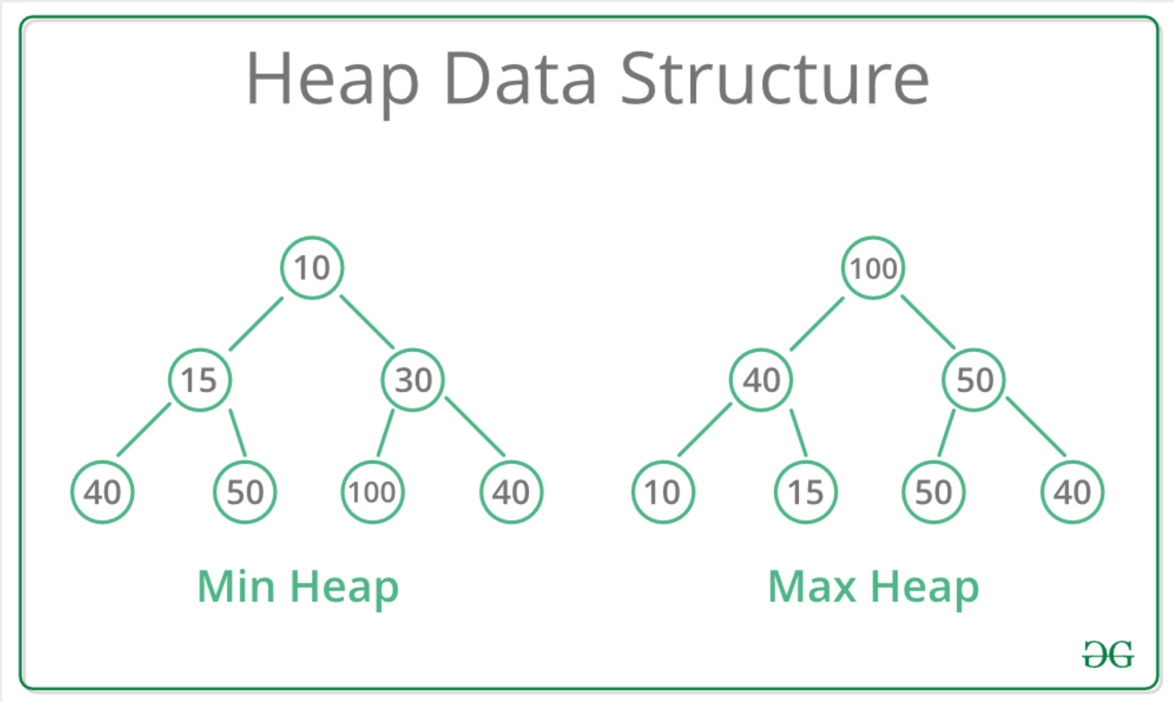
  - **Enqueue**: Add an element to the back.
  - **Dequeue**: Remove the element from the front.
  - **Peek/Front**: View the front element without removing it.
- **Use Cases**: Task scheduling, handling requests in servers, BFS in graphs.

Hash Table (Hash Map)



- **Definition**: A data structure that stores key-value pairs and uses a hash function to compute the index where the value is stored.
- **Operations**:
  - **Insert**: Add key-value pair.
  - **Search**: Retrieve value by key.
  - **Delete**: Remove key-value pair.
- **Use Cases**: Database indexing, caching, fast lookups.

Heap



- **Definition**: A special tree-based data structure that satisfies the **heap property**:
  - **Max Heap**: Parent nodes have greater values than their children.
  - **Min Heap**: Parent nodes have smaller values than their children.
- **Operations**:
  - **Insert**: Add an element and maintain heap property.
  - **Extract Max/Min**: Remove the root (largest or smallest).
  - **Heapify**: Rearrange the tree to maintain the heap property.
- **Use Cases**: Priority queues, heapsort algorithm, scheduling tasks based on priority.

Time Complexity:

- **Stack & Queue**:
  - Push/Pop/Enqueue/Dequeue: O(1)
- **Hash Table**:
  - Insert/Search/Delete: O(1) on average, O(n) in the worst case due to collisions.
- **Heap**:
  - Insert/Extract Min/Max: O(log n)

Sorting Algorithms (arrange data in order)

Algorithm	Time Complexity	Key Feature
Bubble Sort	$O(n^2)$	Repeatedly swaps adjacent elements.

Selection Sort	$O(n^2)$	Selects min/max each pass.
Insertion Sort	$O(n^2)$	Builds sorted array one item at a time.
Merge Sort	$O(n \log n)$	Divide and conquer (stable).
Quick Sort	$O(n \log n)$ avg, $O(n^2)$ worst	Fast, but not stable.
Heap Sort	$O(n \log n)$	Uses a heap to sort.

Searching Algorithms (find data)

Algorithm	Time Complexity	Key Feature
Linear Search	$O(n)$	Check each element one by one.
Binary Search	$O(\log n)$	Works only on sorted data; divide and search.

✅ **Efficient sorting** helps improve search speed.
🔍 **Binary Search** is preferred after sorting.

Sure! Here's a **detailed explanation of Sorting and Searching algorithms**, including how they work, their time complexities, and when to use them:

🔢 Sorting Algorithms

Sorting arranges data in a specific order (ascending or descending). It's essential for efficient searching and organizing data.

1. Bubble Sort

- **How it works**: Repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.
- **Example**: [4, 2, 1] → [2, 4, 1] → [2, 1, 4] → [1, 2, 4]
- **Time Complexity**:
  - Best: $O(n)$ (already sorted)
  - Worst: $O(n^2)$
- **Use Case**: Rare in practice; good for educational purposes.

2. Selection Sort

- **How it works**: Repeatedly finds the minimum (or maximum) element from unsorted part and puts it at the beginning.
- **Example**: [3, 1, 2] → [1, 3, 2] → [1, 2, 3]
- **Time Complexity**: $O(n^2)$ for all cases.
- **Use Case**: Simple and easy to implement, but inefficient.

3. Insertion Sort

- **How it works**: Builds the final sorted array one item at a time by inserting each element into its correct position.
- **Example**: [3, 1, 2] → [1, 3, 2] → [1, 2, 3]
- **Time Complexity**:
  - Best: $O(n)$
  - Worst: $O(n^2)$
- **Use Case**: Efficient for small datasets or nearly sorted data.

4. Merge Sort

- **How it works**: Divide the array into halves, sort each half recursively, and merge them.
- **Example**: [5, 3, 1, 2] → [5, 3], [1, 2] → [3, 5], [1, 2] → [1, 2, 3, 5]
- **Time Complexity**: $O(n \log n)$ in all cases.
- **Use Case**: Stable sort, used in external sorting (large files).

5. Quick Sort

- **How it works**: Selects a "pivot", partitions array into two parts (less than and greater than pivot), then sorts them recursively.
- **Example**: [4, 2, 5, 1] → pivot 2 → [1], 2, [4, 5] → [1, 2, 4, 5]
- **Time Complexity**:
  - Best/Average: $O(n \log n)$
  - Worst: $O(n^2)$ (when pivot is poorly chosen)
- **Use Case**: Very fast in practice; widely used in libraries.

6. Heap Sort

- **How it works**: Builds a max-heap, repeatedly removes the largest element, and rebuilds the heap.
- **Time Complexity**: $O(n \log n)$
- **Use Case**: No additional memory needed (in-place); good for real-time systems.

🔍 Searching Algorithms

Used to find the location or presence of an element in a data structure.

1. Linear Search

- **How it works**: Checks each element one by one.
- **Example**: Search 5 in [2, 3, 5, 7] → compares 2, 3, then finds 5.
- **Time Complexity**: $O(n)$
- **Use Case**: Unsorted or small arrays.

2. Binary Search

- **How it works**: Repeatedly divides a sorted array in half and compares the middle element.

- **Example**: Search 7 in [1, 3, 5, 7, 9] → check 5 → then right → finds 7.
- **Time Complexity**: O(log n)
- **Use Case**: Large sorted arrays.

Summary Table

Algorithm	Best Case	Average	Worst	Stable	In-Place
Bubble Sort	O(n)	O(n^2)	O(n^2)	Yes	Yes
Selection Sort	O(n^2)	O(n^2)	O(n^2)	No	Yes
Insertion Sort	O(n)	O(n^2)	O(n^2)	Yes	Yes
Merge Sort	O(n log n)	O(n log n)	O(n log n)	Yes	No
Quick Sort	O(n log n)	O(n log n)	O(n^2)	No	Yes
Heap Sort	O(n log n)	O(n log n)	O(n log n)	No	Yes
Linear Search	O(1)	O(n)	O(n)	-	-
Binary Search	O(1)	O(log n)	O(log n)	-	-

1. What is a Database?

A **database** is an organized collection of data that allows easy access, management, and updating.

2. DBMS (Database Management System)

Software that manages databases.
Examples: MySQL, PostgreSQL, MongoDB.

3. RDBMS (Relational DBMS)

Stores data in **tables** with rows and columns.
Uses **SQL** to interact with data.
Examples: MySQL, PostgreSQL, Oracle DB.

4. Table Components

- **Row** = Record
- **Column** = Field
- **Primary Key** = Unique identifier for rows
- **Foreign Key** = Links one table to another

✅ 5. ACID Properties

Ensures reliable database transactions:

- **A**tomicity – All or nothing
- **C**onsistency – Data stays valid
- **I**solation – Transactions don't interfere
- **D**urability – Data survives failures

📐 6. Normalization

Process to reduce redundancy and improve data integrity.

🔍 7. SQL (Structured Query Language)

Used to:

- **SELECT** data
- **INSERT** new rows
- **UPDATE** existing records
- **DELETE** rows
- **JOIN** multiple tables

🧮 8. Indexing

Speeds up search queries on large datasets.

🔄 9. Transactions

Group of SQL operations executed together. Should follow ACID rules.

📦 10. NoSQL Databases

Non-relational, used for unstructured or semi-structured data.
Types: Document (MongoDB), Key-Value, Column, Graph.

✅ 5. ACID Properties

These ensure **reliable and safe** transactions in databases.

🔶 1. Atomicity – All or Nothing

If one part of the transaction fails, the whole transaction fails.
Example:
Transferring ₹100 from A to B:

- Deduct ₹100 from A
- Add ₹100 to B
  If the second step fails, the first one rolls back — no money is lost or created.

🔶 2. Consistency – Valid Data Only

Data must follow rules before and after the transaction.
Example:
A bank rule: an account can't go below ₹0.

If A has ₹100, a transaction withdrawing ₹150 will fail — keeping the data valid.

🔶 3. Isolation – No Interference

Multiple transactions can happen together without messing up each other.
Example:
Two users book the **last movie ticket** at the same time.
Isolation ensures **only one booking goes through**, avoiding double booking.

🔶 4. Durability – Survives Failures

Once committed, data stays even after crashes.
Example:
If a message is sent and committed to the DB, it won't disappear even if the server crashes right after.

✅ SQL (Structured Query Language)

A standard language used to **store, retrieve, and manipulate data** in relational databases like MySQL, PostgreSQL, and SQLite.

🔷 Basic SQL Operations (CRUD)

Create – INSERT new records
Read – SELECT data
Update – UPDATE existing records
Delete – DELETE records

🔷 Common SQL Commands

- SELECT * FROM users; → Get all data from the users table
- INSERT INTO users (name, age) VALUES ('Alex', 25); → Add a new user
- UPDATE users SET age = 26 WHERE name = 'Alex'; → Change Alex's age
- DELETE FROM users WHERE age < 18; → Remove users under 18
- WHERE, ORDER BY, GROUP BY, JOIN, LIMIT → Filters, sorts, and connects tables

🔷 Example Query with ACID

```
BEGIN TRANSACTION;

UPDATE accounts SET balance = balance - 100 WHERE name = 'Alice';
UPDATE accounts SET balance = balance + 100 WHERE name = 'Bob';

COMMIT;
```

- Ensures **Atomicity** – both updates happen or none
- **Consistency** – rules like balance ≥ 0 are maintained
- **Isolation** – runs safely with other transactions
- **Durability** – after COMMIT, changes persist

✅ 10. REST API (Representational State Transfer)

A **REST API** allows communication between client and server over **HTTP** using standard operations.

🔷 Key Concepts

- **Client**: The app that makes requests (e.g., browser or mobile app)
- **Server**: The system that handles the request and returns data
- **Resource**: Any object/data (like user, product) identified by a **URL**
- **Stateless**: Each request is independent – server doesn't remember previous ones

🔷 HTTP Methods (CRUD)

Method	Action	Example
GET	Read data	GET /users
POST	Create data	POST /users
PUT	Update data	PUT /users/1
DELETE	Delete data	DELETE /users/1

🔷 JSON Format

Data is sent and received in **JSON** (JavaScript Object Notation):

```json
{
 "name": "Alice",
 "age": 25
}
```

🔷 Example

Request:

```http
GET /products/123
```

Response:

```json
{
 "id": 123,
 "name": "Laptop",
 "price": 999.99
}
```

REST APIs are widely used in **web apps**, **mobile apps**, and **microservices** due to their simplicity and scalability.