1. OOPS: SOLID Principles (Overview)

SOLID is an acronym for five design principles intended to make software designs more understandable, flexible, and maintainable.

1. **S - Single Responsibility Principle (SRP):** A class should have one, and only one, reason to change.
2. **O - Open/Closed Principle (OCP):** Software entities should be open for extension but closed for modification.
3. **L - Liskov Substitution Principle (LSP):** Objects of a superclass should be replaceable with objects of its subclasses without breaking the application.
4. **I - Interface Segregation Principle (ISP):** No client should be forced to depend on methods it does not use.
5. **D - Dependency Inversion Principle (DIP):** Depend on abstractions, not on concretions.

2. OOPS Scenario: Violating SRP

Problem: You have a `UserAuth` class that handles user login logic, calculates user permissions, **and** logs activity to a text file.

The Violation: This class has three distinct responsibilities. If the logging format changes, you have to modify the `UserAuth` class. If the database schema for permissions changes, you modify the same class. This creates a "God Object" that is fragile and hard to test.

The SOLID Solution (Applying SRP): Split the class into three:

1. `Authenticator` : Handles the login/credentials logic.
2. `Authorizer` : Handles permission and role logic.
3. `Logger` : Handles writing to the text file.

The Result: Now, if you want to switch from logging to a file to logging to a Cloud API, you only change the `Logger` class. The login logic remains untouched and safe.

3. Programming: Queue Using Linked List

Unlike an array-based queue, a linked list implementation is dynamic and doesn't require a "Circular" logic to reuse space, as we can simply add and remove nodes.

Implementation Logic (FIFO)

To achieve $O(1)$ efficiency, we maintain two pointers: `front` (to dequeue) and `rear` (to enqueue).

1. **Enqueue Operation (Add to Tail):**
   - Create a new node.
   - If `rear` is `NULL` , both `front` and `rear` point to the new node.

- Else, `rear.next = new_node` and update `rear = new_node`.

2. **Dequeue Operation (Remove from Head):**
   - Check for Underflow (`front == NULL`).
   - Store the `front` node data.
   - Move `front` forward: `front = front.next`.
   - If `front` becomes `NULL`, set `rear` to `NULL` as well (Queue is empty).

Complexity: $O(1)$ for both Enqueue and Dequeue.

4. SQL: JOIN with CASE Statement

You can use `CASE` expressions within or after a `JOIN` to categorize data dynamically based on values from multiple tables.

Scenario: Categorizing Order Priority

We want to join `Orders` and `Shipping` and label the order priority based on the shipping speed and order value.

Syntax Example:

```
SELECT
    o.order_id,
    c.customer_name,
    CASE
        WHEN s.shipping_method = 'Express' THEN 'Critical'
        WHEN o.total_amount > 1000 THEN 'High Priority'
        ELSE 'Standard'
    END AS fulfillment_status
FROM orders o
INNER JOIN customers c ON o.customer_id = c.customer_id
LEFT JOIN shipping s ON o.order_id = s.order_id;
```

Why use it?

- **Business Logic in Data:** It moves simple labeling logic out of the application code and into the database query, which is often more efficient for large reports.
- **Custom Buckets:** Excellent for creating custom "tiers" (e.g., Gold/Silver/Bronze) based on joined aggregate data.

Summary Table

Topic	Focus	Key Takeaway
OOPS	SOLID (SRP)	A class should do one thing; splitting responsibilities reduces "ripple effects" during changes.

DSA	Queue (Linked List)	Uses `front` and `rear` pointers to ensure $O(1)$ time for both ends.
SQL	JOIN + CASE	Allows conditional logic and custom labeling based on data from combined tables.

Quarter-way through the journey! You've moved from basic OOPS to professional-grade design principles.