1. OOPS: Diamond Problem & MRO

The Diamond Problem

This is a classic issue in languages that support multiple inheritance. It occurs when a class `D` inherits from both `B` and `C`, and both `B` and `C` inherit from a common base class `A`.

The Ambiguity: If class `A` has a method that both `B` and `C` override, and `D` calls that method, which version does it execute? Without a rule, the compiler/interpreter wouldn't know whether to look in `B` or `C`.

Method Resolution Order (MRO)

Python resolves this using an algorithm called **C3 Linearization**.

- It creates a linear order in which classes are searched when looking for a method.
- You can view this order by calling `ClassName.mro()` or `object.__mro__`.
- **The Rule:** Search is generally **Depth-First, Left-to-Right**, but it ensures that a class is always searched *before* its parents and that the order of parents is preserved.

2. OOPS Scenario: Predictive Behavior in Hierarchies

Problem: In a large corporate system, you have a `TechnicalManager` class that inherits from both `Developer` and `Manager`. Both parent classes have a `get_permissions()` method.

Analysis:

1. **Ambiguity:** If `TechnicalManager` doesn't define its own permissions, should it get the `Developer` permissions or the `Manager` permissions first?
2. **MRO Solution:** In Python, if you define `class TechnicalManager(Developer, Manager):`, the MRO will check `Developer` first. If you swap them to `(Manager, Developer)`, it checks `Manager` first.
3. **Predictability:** MRO ensures that no matter how complex the "diamond" gets, the behavior is predictable and doesn't lead to the "deadly diamond of death" found in older languages.

3. Programming: Stack Using Linked List

While an array-based stack is simple, it has a fixed size. A **Linked List** implementation allows the stack to grow and shrink dynamically.

Implementation Logic (LIFO)

To achieve $O(1)$ time complexity for both `push` and `pop`, we always perform operations at the **Head** of the linked list.

1. **Push Operation:**

- Create a new node.
- Set `new_node.next = head` .
- Update `head = new_node` .

2. **Pop Operation:**
    - Check for Underflow (`head == NULL`).
    - Store the current `head` data to return.
    - Update `head = head.next` .

Advantages:

- **Dynamic Size:** No need to pre-allocate memory.
- **Efficiency:** No "resizing" or "copying" overhead (which happens when an array/list reaches capacity).

Complexity: $O(1)$ Time for all operations.

4. SQL: JOIN with HAVING Clause

The `HAVING` clause is used specifically to filter the results of an aggregate function (`SUM` , `COUNT` , `AVG`).

WHERE vs. HAVING

- **WHERE:** Filters individual rows *before* the groups are created. It cannot be used with aggregate functions.
- **HAVING:** Filters the groups *after* the `GROUP BY` and aggregation have been performed.

Syntax Example:

Find customers who have placed more than 5 orders.

```
SELECT c.customer_name, COUNT(o.order_id) AS order_count
FROM customers c
INNER JOIN orders o ON c.customer_id = o.customer_id
GROUP BY c.customer_name
HAVING COUNT(o.order_id) > 5;
```

Key Logic:

If you want to filter by a specific city, use `WHERE` . If you want to filter by a "Total" or "Average" calculated across multiple rows, use `HAVING` .

Summary Table

Topic	Focus	Key Takeaway

OOPS	Diamond Problem	Resolved by MRO (Method Resolution Order) to prevent inheritance ambiguity.
DSA	Stack (Linked List)	Provides $O(1)$ operations with dynamic memory growth.
SQL	HAVING Clause	Filters aggregated results; it is to "groups" what `WHERE` is to "rows."

Day 13 complete. You've moved from static structures to dynamic memory and advanced logic resolution!