**Solution 1: Inventory Management System**

Scenario: You are developing an inventory management system for a warehouse. Efficient data storage and retrieval are crucial.

**>> Explain why data structures and algorithms are essential in handling large inventories.**

* **Efficiency**: Efficient data structures and algorithms ensure fast data retrieval and updates, which is critical in large inventories where operations need to be performed quickly.
* **Scalability**: Proper data structures can handle the increasing size of data without a significant drop in performance.
* **Resource Management**: Optimized algorithms and data structures use resources effectively, minimizing memory and processing power requirements.

**>> Discuss the types of data structures suitable for this problem.**

* **ArrayList (Dynamic Array):** Suitable for scenarios where the order of insertion needs to be preserved and random access is frequent.
* **HashMap (Hash Table):** Ideal for quick lookups, insertions, and deletions. Allows for fast access to items based on unique keys (e.g., productId).
* **Linked List:** Useful for situations where frequent insertions and deletions occur, especially in the middle of the list.
* **Tree (e.g., Binary Search Tree, AVL Tree):** Good for maintaining a sorted order of products and providing efficient range queries.

We will use a HashMap to store the products, where the key is productId and the value is the Product object. This provides O(1) average time complexity for add, update, and delete operations.

**Analysis**

**1. Add Product**

Method: addProduct(Product product)

* Operation: Adds a Product object to the HashMap with the product ID as the key.
* Time Complexity: O(1) on average.

Explanation: Inserting a key-value pair into a HashMap generally takes constant time because the underlying data structure is a hash table. However, in worst-case scenarios (e.g., many collisions), the complexity can degrade to O(n), where n is the number of entries.

**2. Update Product**

Method: updateProduct(String productId, Product updatedProduct)

* Operation: Updates an existing Product object in the HashMap if it exists.
* Time Complexity: O(1) on average.

Explanation: Since HashMap provides constant time complexity for lookups and updates, updating a product involves retrieving the product by its ID (constant time) and then replacing it with the updated product (constant time).

**3. Delete Product**

Method: deleteProduct(String productId)

* Operation: Removes a Product from the HashMap using its product ID.
* Time Complexity: O(1) on average.

Explanation: Removing a key-value pair from a HashMap is generally a constant time operation due to the hash table structure. As with insertion and updates, the worst-case scenario can be O(n) if many collisions occur.

**4. Get Product**

Method: getProduct(String productId)

* Operation: Retrieves a Product from the HashMap using its product ID.
* Time Complexity: O(1) on average.

Explanation: Accessing a value in a HashMap by key is an average constant-time operation. This is due to the hash-based indexing used internally.

**5. Display All Products**

Method: displayAllProducts()

* Operation: Iterates over all Product objects in the HashMap and prints them.
* Time Complexity: O(n), where n is the number of products in the inventory.

**Explanation**: Iterating through all entries in a HashMap involves visiting each entry exactly once. Therefore, the time complexity is linear with respect to the number of entries.

**How to Optimize**

* **HashMap Optimizations**: Properly initialize capacity and load factor.
* **Batch Operations**: Minimize redundant operations and consider batching.
* **Concurrency**: Use thread-safe data structures or synchronization.
* **Pagination and Caching**: Implement for better performance with large datasets.
* **Persistence**: Use databases or file storage for data persistence and optimize I/O.

**How to Run the code**

* Simply run the Main.java file

Or,

* Go till \_01\_Answer directory
* Javac \*.java
* Java InventoryManagement