**1.Understand the Problem:**

**Why Data Structures and Algorithms Are Essential in Handling Large Inventories:**

**Efficient Searching**

In a large inventory, there may be **thousands of products**.Without proper data structures, finding a single product (like by product ID or name) will take a long time.

* Example: Searching in a normal list takes **O(n)** time.  
  But using a **HashMap**, it takes **O(1)** time.

**Faster Updates and Deletions**

Products may need to be **updated frequently** (e.g., quantity or price changes).If we store products in a normal list, we must **search through the list** to find the item before updating = **slow**.

With the right data structure (like a HashMap):

* Updating and deleting become **instant operations** (O(1)).

**Organized Storage**

Data structures help to **store data logically**.

* Arrays or Lists for sequential access
* HashMaps for key-based access (like product ID)
* Trees for sorted access

This organization improves the performance and clarity of the code.

**Memory Efficiency**

* Good data structures **reduce memory wastage** and allow better **space utilization**, especially in systems handling thousands of records.

**Scalability and Performance**

* When inventory size grows from 100 to 10,000:
* Poor data structures make the system **slow**
* Good ones (like HashMap or TreeMap) ensure that the system remains **fast and responsive**

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

In an inventory system, we need to perform operations like:

* Add a new product
* Update product quantity or price
* Delete a product
* Search by product ID
* Display all products

To handle these efficiently, the following data structures are commonly used:

**HashMap ( Best Choice)**

**Why:**

* Fast lookup using productId as the key
* Supports constant time: **O(1)** for add, update, delete, and search
* Perfect for large inventories where each product has a unique ID

**Example:**

HashMap<Integer, Product> inventory;

**ArrayList ( Not Preferred for This Use Case)**

**Why not:**

* Works well for small lists
* But searching and updating by ID takes **O(n)** time
* You have to loop through the entire list

**TreeMap (Optional - for Sorted Inventory)**

**Why:**

* Stores elements in **sorted order of keys**
* Slightly slower than HashMap (**O(log n)**), but useful if you want the product list to be automatically sorted by ID

**LinkedList (Not Suitable)**

**Why not:**

* Accessing elements by ID is slow (**O(n)**)
* Not designed for key-based access

**4.Analyses**

**Analyze the Time Complexity of Each Operation (Using HashMap)**

**I chose:**

HashMap<Integer, Product> inventory;

Here, productId is used as the key to access each Product object.

**Add Product**

inventory.put(productId, product);

* Time Complexity: O(1) (Average case)
* Reason: HashMap stores the object directly at the hash index of the key. No need to search or loop.

**Update Product**

Product p = inventory.get(productId);

p.quantity = newQty;

p.price = newPrice;

* Time Complexity: O(1)
* Reason: We can directly access the object using the key. Updating fields is also constant time.

**Delete Product**

inventory.remove(productId);

* Time Complexity: O(1)
* Reason: Remove operation uses the key to find and delete the product in constant time.

**Discuss how you can optimize these operations.**

To optimize add, update, and delete operations in my inventory system, I used **HashMap** with **productId as the key**. Since HashMap gives us constant time **O(1)** access, it’s already efficient, but still I followed a few steps to improve it further.

**I used productId as the key**

* Example:  
  If I want to find product 101, I directly do:

inventory.get(101);

No need to loop or search. It's instant access.

**I checked if the product already exists before adding or updating**

* Example:

if (!inventory.containsKey(101)) {

inventory.put(101, newProduct);

}

This prevents **duplicate entries** and also avoids overwriting by mistake.

**Before deleting, I checked if the ID exists**

* Example:

if (inventory.containsKey(101)) {

inventory.remove(101);

}

This avoids errors like trying to delete something that doesn't exist.

**If I expect many products, I can give an initial capacity in HashMap**

* Example:

HashMap<Integer, Product> inventory = new HashMap<>(1000);

This avoids rehashing and makes performance better for large data.

So overall, I optimized my operations by using proper keys, checking before update/delete, and planning for large data. This helps my system run faster and safer even if 1000+ products are there.