Analysis of the time complexity of each operation (add, update, delete)

* In this inventory management , I am using a HashMap to store products by their unique productId offers excellent performance.
* The **time complexity for adding a product** is **O (1)** on average because the HashMap uses a hashing algorithm to directly compute the index where the product should be stored. This means the system does not need to search through the entire list of products; it simply calculates the hash and stores or retrieves the item at that position.
* **Updating a product** also takes **O (1)** time on average since it involves locating the product using its ID and modifying its details, which again relies on direct access through hashing.
* Similarly, **deleting a product** is done in constant time by locating the product via its key and removing it from the map. However, in rare cases where multiple keys produce the same hash value (called **collisions**), the HashMap uses techniques like chaining or balanced trees internally, which can degrade performance to **O(n)** in the worst case. Fortunately, Java’s implementation optimizes this scenario so well that average-case **O (1)** remains practically consistent even for large datasets.
* Overall, using a HashMap ensures that the essential operations like add, update, and delete are executed in constant time, making it ideal for handling large warehouse inventories where performance and scalability are crucial.

Analysis to Optimize these operations

To optimize the operations of adding, updating, and deleting products in an inventory management system, the first and most important step is to **choose the right data structure**, such as a HashMap, which already provides constant time complexity (O(1)) for these operations in most cases.

However, further optimization is possible. For instance, before adding a new product, the system should first **check whether the product ID already exists** to prevent overwriting existing data, which maintains data integrity. During updates, ensure that only changed fields are modified instead of reassigning the entire object, which reduces unnecessary memory operations.

For deletions, avoid calling removal logic on non-existent IDs by checking their presence first, which prevents wasted computation. If the inventory is expected to grow significantly, a proper **initial capacity and load factor** should be set for the HashMap to **reduce rehashing overhead**, which can slow down performance when the map resizes internally.

In scenarios where frequent sorting or range queries are needed, consider using a TreeMap for **logarithmic time complexity with automatic sorting**, though it comes at the cost of slightly slower operations compared to a HashMap. Additionally, for very large inventories, combining in-memory storage with persistent databases and using **batch processing or caching strategies** can further enhance overall efficiency. Overall, smart usage patterns, pre-checks, appropriate configurations, and combining the right tools based on the use case can optimize these operations effectively.