**Exercise 1: Inventory Management System**

**1. Understand the Problem**

**Why Data Structures and Algorithms are Essential in Handling Large Inventories**

Data structures and algorithms play a important role in handling large inventories because they determine how efficiently data can be stored, accessed, and changed. The choice of data structure affects the performance of the system by considering the factors of time and space complexity. Efficient algorithms ensure that operations such as adding, updating, and deleting products are performed quickly, which is essential for maintaining an up-to-date and accurate inventory.

**Types of Data Structures Suitable for This Problem**

* **ArrayList**: Provides fast access to elements using indexes. Suitable for scenarios where random access is more frequent than insertions and deletions.
* **LinkedList**: Provides efficient insertions and deletions but slower random access. Useful when frequent additions and deletions are expected.
* **HashMap**: Provides average constant time complexity (O(1)) for insertions, deletions, and lookups. Suitable for scenarios where fast access to elements by key is required.
* **TreeMap**: Provides log(n) time complexity for insertions, deletions, and lookups. Useful when elements need to be stored in a sorted order.

4. Analysis

Time Complexity of Each Operation for HashMap algorithm

We'll consider the best case, average case, and worst case for each operation.

i.Add Product

* Best Case: O(1)
  + When there are no hash collisions and the hash table does not need resizing, adding an element involves a single insertion operation.
* Average Case: O(1)
  + In a well-distributed hash table, adding an element typically takes constant time because hash collisions are minimized.
* Worst Case: O(n)
  + In case of many hash collisions, insertion may degrade to O(n). Additionally, if the hash table needs to resize, the insertion time can temporarily degrade.

ii.Update Product

* Best Case: O(1)
  + If the product exists in the hash table and there are no hash collisions, updating the element involves a single retrieval and insertion operation.
* Average Case: O(1)
  + Generally, updating an element in a well-distributed hash table takes constant time.
* Worst Case: O(n)
  + Similar to the worst-case for addition, if many elements hash to the same bucket, retrieving and updating an element can degrade to O(n).

iii.Delete Product

* Best Case: O(1)
  + When there are no hash collisions and the product exists in the hash table, removing an element involves a single deletion operation.
* Average Case: O(1)
  + In a well-distributed hash table, deleting an element typically takes constant time.
* Worst Case: O(n)
  + If many elements hash to the same bucket, deleting an element can degrade to O(n).

Optimization

* Load Factor and Resizing: Ensure that the load factor of the HashMap is managed properly (default is 0.75) to avoid frequent resizing and hash collisions. Adjusting the initial capacity and load factor can help maintain O(1) operations in most cases.
* Hash Function: Use a good hash function to distribute keys uniformly across the hash table, reducing the likelihood of collisions and improving average-case performance.
* Concurrency: If the system will be accessed by multiple threads, use ConcurrentHashMap to handle concurrent modifications efficiently without locking the entire table.