Week - 1 : Algorithms & Data Structures

**Inventory System Management**

# Understanding the problem

**Importance of Data Structures and Algorithms in Handling Large Inventories**

Data structures and algorithms are crucial in handling large inventories for several reasons:

### Efficiency:

* + - **Time Complexity**: Efficient data structures and algorithms ensure that operations such as search, insert, update, and delete can be performed quickly, even as the inventory grows. For instance, a linear search in an unsorted list has a time complexity of O(n), whereas a binary search in a sorted list or operations in a balanced tree can be O(log n).
    - **Space Complexity**: Proper data structures can help manage memory usage more efficiently, which is essential for large inventories.

### Scalability:

* + - Efficient algorithms and data structures allow systems to scale and handle increased loads without a proportional increase in processing time. This is critical for maintaining performance as the number of items grows.

## Suitable Data Structures for Inventory Management

### ArrayList:

* + **Advantages**: Simple to implement, provides dynamic resizing, and is efficient for sequential access.
  + **Disadvantages**: Searching and deleting elements can be slow (O(n) in worst case).

### HashMap:

* + **Advantages**: Provides average O(1) time complexity for insert, delete, and search operations. Suitable for fast lookups.
  + **Disadvantages**: Does not maintain any order of elements, and the worst-case time complexity can degrade to O(n) if there are many hash collisions.

# Setup

* Create a new Java project for the inventory management system using any IDE such as Visual Studio Code, Eclipse, or a simple text editor with terminal.
* Inside the project directory, create two Java classes: **Inventory.java** and **Product.java**.
* Compile both Java files and run the Inventory class which contains the main method to interact with the system.

# Implementation

## Product Class Definition:

A seperate Product() class is defined to represent individual products in the inventory. It includes the following attributes:

* **productId (int)**: A unique identifier for each product
* **productName (String)**: The name of the product
* **quantity (int)**: The number of units available in the inventory
* **price (double)**: The cost of a single unit of the product

Get & Set methods are user for each attributes, along with a constructor and a **toString()** method to display product details.

## Data Structures Used:

To manage the inventory, a **HashMap<Integer, Product>** is used. The productId is used as the key, and the Product object is the value. This allows:

* Fast lookups for updating or deleting specific products
* Unique identification of each product
* Efficient storage and access using keys

## Core Functionalities Implemented:

### Add Product:

The user enters details for a new product. A Product object is created and added to the HashMap with its ID as the key. After each entry, the user is prompted whether they want to add another product.

### Update Product:

The product is searched using its ID. If found, the product name, quantity, and price can be updated using user input.

### Delete Product:

The product is removed from the map using its ID. If the ID does not exist, a message is displayed.

# Analysis:

## Time Complexity Analysis:

### Add Operation:

Inserting a new product into the HashMap using put() takes O(1) time on average. This is because hashing allows direct access to the storage location based on the key (product ID).

### Update Operation:

Updating a product involves retrieving it using get() and modifying its attributes. The retrieval is O(1) on average, so the total time for the update is also O(1).

### Delete Operation:

Removing a product using remove() takes O(1) time on average, as it directly accesses the product using the key and deletes it.

## Optimization Discussion:

Using a HashMap significantly improves the efficiency compared to an ArrayList. In an ArrayList, operations like update or delete require a linear search, resulting in **O(n)** time complexity. By switching to a HashMap, all major operations add, update, delete are optimized to **O(1)** on average.