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

1. **Efficient Data Storage and Retrieval:**
   * Large inventories can consist of thousands or even millions of items. Efficient data structures ensure that storage and retrieval operations (such as searching for a specific product) are performed quickly, minimizing latency and improving user experience.
2. **Optimal Use of Resources:**
   * Proper data structures help in using memory and computational resources effectively. For instance, using a HashMap for fast lookups or a balanced tree structure to maintain sorted data with logarithmic time complexity.
3. **Scalability:**
   * As the inventory grows, the chosen data structures and algorithms must scale efficiently. Inefficient structures can lead to performance degradation as the data size increases.
4. **Maintenance and Updation:**
   * Efficient algorithms facilitate quick updates and maintenance of inventory data, such as adding new products, updating quantities, or removing discontinued items.

**Suitable Data Structures for this Problem:**

1. **ArrayList:**
   * **Use:** Fixed size or infrequent changes.
   * **Pro:** Fast random access.
   * **Con:** Slow insertions/deletions.
2. **HashMap:**
   * **Use:** Key-value storage (e.g., product IDs).
   * **Pro:** O(1) average time complexity.
   * **Con:** No order maintained.
3. **Balanced Binary Search Tree:**
   * **Use:** Sorted data and range queries.
   * **Pro:** O(log n) time complexity.
   * **Con:** More complex implementation.
4. **Linked List:**
   * **Use:** Frequent insertions/deletions.
   * **Pro:** O(1) insertion/deletion.
   * **Con:** O(n) lookups.

**Analysis:**

**Time Complexity Analysis:-**

Here is the time complexity analysis of each operation in my code:

1. **Adding a Product (addProduct Method)**

* **Time Complexity:** O(1)
* **Explanation:** The addProduct method involves constant time operations: creating a new Product, setting its properties, and reading user input.

1. **Updating a Product (updateProduct** **Method)**

* **Time Complexity:** O(1)
* **Explanation:** The updateProduct method also involves constant time operations: creating a new Product, setting its properties, and reading user input.

1. **Deleting a Product (deleteProduct Method)**

* **Time Complexity:** O(n)
* **Explanation:** The deleteProduct method involves iterating through the ArrayList to find the product with the specified productId. In the worst case, it might need to check every element, resulting in O(n) complexity.

1. **Displaying Products (displayProduct Method)**

* **Time Complexity:** O(n)
* **Explanation:** The display method iterates through the ArrayList and prints each product. The time complexity is proportional to the number of products, resulting in O(n) complexity.

**Optimization:-**

### **Optimizing addProduct**

* **Current Complexity:** O(1)
* **Optimization:** This operation is already optimal as it involves constant-time operations for setting attributes and reading user input. No further optimization is needed.

### **Optimizing updateProduct**

* **Current Complexity:** O(1)
* **Optimization:** This operation is already optimal as it involves constant-time operations for setting attributes and reading user input. No further optimization is needed.

### **Optimizing deleteProduct**

* **Current Complexity:** O(n)
* **Optimization:** Use a **HashMap** for storing products, where the key is the product ID. This can bring the deletion operation down to O(1) because finding and removing an item from a HashMap is constant time.

### **Optimizing display**

* **Current Complexity:** O(n)
* **Optimization:** The display operation requires iterating through all elements, which is inherently O(n). If using a HashMap for other operations, convert the values to a list for display, which would still be O(n).