Exercise 2: E-commerce Platform Search Function

1. Understand the Problem:

In an e-commerce platform with a large number of products, managing and searching the inventory efficiently is critical. When the number of items grows from a few hundreds to thousands, poor data structures can make the system slow and unresponsive. That's why understanding and using the right data structures and algorithms is essential.

Efficient Searching:

If we use a simple list to store products, then finding a product like "Laptop" or an item with productId 303 will require us to check each element one by one. This takes linear time, which means if we have n products, we may need to do up to n checks. This is called **O(n)** time complexity. On the other hand, if we use structures like HashMap where we can access items by their key (like productId), then the search time becomes **O(1)** – meaning it takes the same short time regardless of how many products there are.

Faster Updates and Deletions:

Updating or deleting a product also requires locating it first. With a simple array or list, this would again take O(n) time because we'd have to search through the list to find the correct product. But if we store products in a HashMap using the productId as the key, we can directly access, update, or remove it in constant time, which is O(1).

Organized Storage:

The data structure also helps us organize the product records. Arrays or ArrayLists are good for storing items in order, but for large systems where we need fast access,

HashMaps are more suitable. If we want our inventory sorted by ID, TreeMap can help because it keeps data sorted with O(log n) time complexity for most operations.

Memory Efficiency:

Choosing the right data structure avoids memory wastage. For instance, using a HashMap with a good initial size avoids unnecessary resizing. Poor choices lead to unused space or slow operations when the data grows.

Scalability and Performance:

When the system scales up and data grows from 100 to 10,000 products, a well-designed structure like HashMap or TreeMap ensures that performance remains stable. Otherwise, the system becomes too slow to use.

2. Understand Asymptotic Notation:

Big O Notation:

Big O notation describes how the performance of an algorithm grows as the size of the input grows. For example, O(1) means constant time – it doesn't matter how big the input is, the time stays the same. O(n) means the time grows in proportion to the size of the input. O(log n) means time grows slowly even as input grows a lot.

Best, Average, and Worst Case in Search:

For **linear search**, the best case is O(1) when the item is found at the beginning. The average case is O(n/2), but we simplify it to O(n), and the worst case is O(n) when the item is at the end or not present.

For **binary search**, the best case is O(1) when the item is exactly in the middle. The average and worst case are both O(log n), because each time we cut the search range in half. But binary search requires the array to be sorted.

3. Discuss the Types of Data Structures:

In an inventory system, we need to support operations like adding a product, updating details, deleting a product, searching by ID or name, and displaying everything.

HashMap is the best choice. It allows us to search, add, delete, and update in O(1) time. We can use productId as the key, and store the full product object as the value.

ArrayList is not ideal for large inventories. Searching by ID means we have to loop through every item until we find a match, which takes O(n) time. It's okay for small data but not scalable.

TreeMap is good if we want to keep our data sorted by productId. It takes O(log n) time for all operations, which is slower than HashMap but acceptable when sorting is required.

LinkedList is not recommended for this problem because access by index or key takes O(n) time and it’s not built for fast search.

4. Analysis:

Let’s compare linear search and binary search in terms of time complexity.

Linear Search:

In this method, we go one by one through each product in the list. If we have 1,000 products, in the worst case we might have to look at all 1,000. So, the time complexity is O(n). It's simple and works for any list, sorted or not.

Binary Search:

In binary search, we first sort the array (if not already sorted). Then we check the middle element. If it's not what we want, we look at either the left or right half, and keep cutting the search range in half. So, the time complexity is O(log n), which is much faster than O(n) for large inputs. But binary search only works on sorted data.

Which Algorithm is Better for My Platform:

In our e-commerce inventory system, binary search is better for search operations **if** the data is stored in a sorted array and not changing frequently. But for most real-world systems where products are added, removed, and updated constantly, maintaining sorted order is inefficient.

That's why I prefer using a **HashMap**. Even though binary search is faster than linear search, HashMap is faster than both with O(1) time for search, update, and delete. So it is the most suitable for large inventory handling.

5. Final Optimization Strategy:

I used a HashMap with productId as the key. This gives me instant access to any product using get(productId). Before adding or updating, I check if the ID already exists using containsKey(). Before deleting, I do the same to avoid errors.

Also, when I know that I will store thousands of products, I give an initial capacity like new HashMap<>(1000) to avoid rehashing, which improves performance.

By using these practices and a proper understanding of algorithm complexity and data structure behavior, my e-commerce platform remains fast, even with a huge number of products.