Exercise 1: Inventory Management System

The Significance of Data Structures and Algorithms in Managing Extensive Inventories:  
  
Efficiency: Time and computing resources required for tasks like finding, adding, updating, and removing inventory items are reduced by using effective data structures and algorithms.

Scalability: When data structures are selected carefully, the system may accommodate larger data volumes without seeing appreciable performance drops.

arrangement: Managing and retrieving inventory data is made easier by logical data arrangement.

Types of data Structures that Are Good for Inventory Management:  
  
ArrayList: Helpful for keeping a product list in order when fast index access is required. If the index is unknown, it is less effective for searches, insertions, and deletions.

HashMap: Uses keys (such as product IDs) to quickly access, insert, and remove entries. This is perfect for managing inventories when each product can be uniquely identified by an ID.

The implementation of the Inventory Management System involves creating two main classes namely ‘product’ and ‘InvetoryManagementSystem’

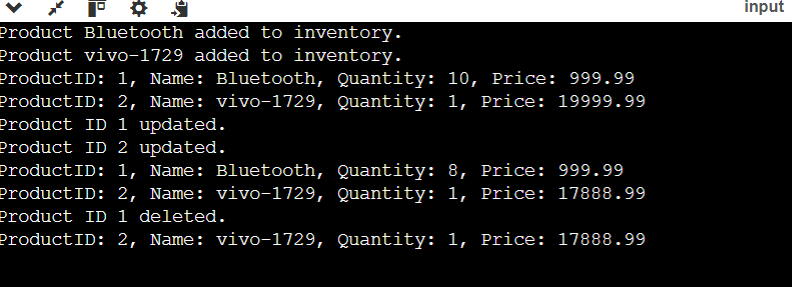
Product class contains:

* Attributes
* Constructor (Initializes new products)
* Getter Methods (To retrieve)
* Setter Methods (To Update)
* ToString Method

Analysis:

This implemented design ensures efficient data storage and retrieval, Employing the HashMap for constant-time complexity (O(1)) operations for adding, updating, and deleting products. The implementation focuses on maintaining a clear and organized structure, making it easy to manage and manipulate the inventory data.

Sample output:



Exercise 2: E-commerce Platform Search Function

Generally asymptotic notations are used to describe how the performance of an algorithm changes as the size of the input data increases.

How Big O notation helps in analysing algorithms?

Big Onotation is a way to express the upper bound of an algorithm's runtime or space requirement. It provides a way to describe the worst-case scenario of an algorithm's performance. Here's a simple explanation of the common Big O notations:

1. O(1) - Constant Time:

* The algorithm takes the same amount of time regardless of the size of the input.

Ex: Accessing an element in an array by index

1. O(n) - Linear Time:

* The algorithm's runtime grows linearly with the size of the input.

Ex: Iterating through an array to find an element.

1. O(log n) - Logarithmic Time:

* The algorithm's runtime grows logarithmically with the size of the input. This often happens with algorithms that divide the problem in half each time.

Ex: Binary search in a sorted array.

1. O(n^2) - Quadratic Time:

* The algorithm's runtime grows quadratically with the size of the input. This is often seen with algorithms that involve nested loops.

Ex: Bubble sort or selection sort.

Overall Design Efficiency:

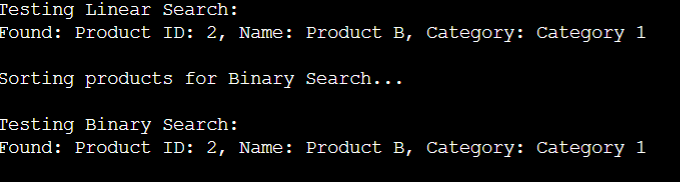
* Linear Search: Easy to implement, suitable for small or unsorted datasets. Performance drops with larger datasets.
* Binary Search: Efficient for large, sorted datasets, with faster search times compared to linear search.

Opted choices:

* Data Structure Choice: Linear search for simple or small datasets, binary search for large, sorted datasets.
* Performance: Binary search offers better performance for larger data, making it more suitable for e-commerce platforms with extensive product lists.

This approach ensures efficient searching and easy management of product data, enhancing the overall performance of the e-commerce platform.

Sample Output:



Exercise 3: Sorting Customer Orders

Sorting customer orders by their total price on an e-commerce platform to prioritize high value orders.

1. Understanding Sorting Algorithms:

Sorting algorithms help us organize data. Here’s a quick overview:

* Bubble Sort**:** This algorithm repeatedly steps through the list, compares adjacent elements, and swaps them if they’re in the wrong order. It’s simple but not very efficient for large datasets.
* Insertion Sort**:** This method builds the final sorted array one item at a time by repeatedly picking the next item and inserting it into its correct position. It’s efficient for small or nearly sorted datasets.
* Quick Sort**:** This is a highly efficient algorithm that works by selecting a 'pivot' element and partitioning the other elements into those less than and greater than the pivot. It’s generally faster than Bubble and Insertion Sort for larger datasets.
* Merge Sort**:** This algorithm divides the dataset into smaller units, sorts each unit, and then merges them back together. It’s very efficient and works well for large datasets.

2. Setup**:**

To manage and sort orders, start by creating a class Order. This class will hold details like the order ID, the customer’s name, and the total price of the order.

3. Implementation:

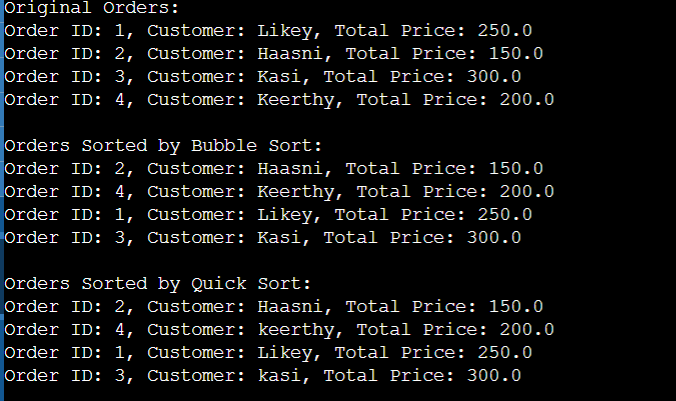
Implemented two sorting methods:

* Bubble Sort:
  + Compare each pair of adjacent orders.
  + Swap them if they are not in the correct order.
  + Repeat until the list is sorted.
* Quick Sort:
  + Choose a pivot element from the array.
  + Partition the array into two sub-arrays: elements less than the pivot and elements greater than the pivot.
  + Recursively apply the same steps to the sub-arrays until the whole array is sorted.

4**.** Analysis**:**

* Bubble Sort vs. Quick Sort:
  + Bubble Sort has a time complexity of O(n²), making it slower for large datasets.
  + Quick Sort has an average time complexity of O(n log n), making it much faster for large datasets compared to Bubble Sort.

Sample Output:



Exercise 4: Employee Management System

To Develop an employee management system for a company. Efficiently managing employee records is crucial.

1. Array Representation:

Arrays are like boxes lined up in a row, where each box (or element) holds a piece of data. They are stored in contiguous memory locations, which means each element is next to the previous one. This allows for quick access to any element using its index, but the size of the array is fixed once it’s created.

2. Setup**:**

* Employee Class**:** This class will hold details about an employee such as:
  + employeeId: A unique identifier for the employee.
  + name: The employee’s name.
  + position: The employee’s job position.
  + salary: The employee’s salary.

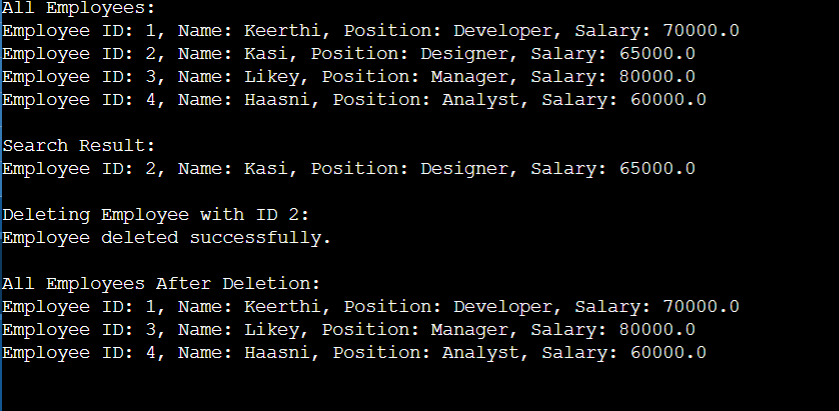
3. Implementation:

* Using an Array**:** We will use an array to store and manage employee records.
* Methods:
  + Add**:** Add a new employee to the array.
  + Search**:** Find an employee by their ID.
  + Traverse**:** Go through and print all employee records.
  + Delete**:** Remove an employee by their ID.

4. Analysis**:**

* Time Complexity**:**
  + Add Operation: O(1) – Constant time if there’s space in the array.
  + Search Operation**:** O(n) – Linear time as it may need to check each element.
  + Traverse Operation**:** O(n) – Linear time to go through all elements.
  + Delete Operation**:** O(n) – Linear time due to shifting elements to fill the gap.

Sample Output:



Exercise 5: Task Management System

1. Understand Linked Lists:

* Singly Linked List:
  + Each node contains data and a reference (or link) to the next node.
  + Allows traversal in one direction (from head to end).
  + Simpler to implement but only supports forward traversal.
* Doubly Linked List:
  + Each node contains data, a reference to the next node, and a reference to the previous node.
  + Allows traversal in both directions (forward and backward).
  + More complex due to additional pointers but offers more flexibility.

2. Setup**:**

* Task Class: This class will hold details about a task such as:
  + taskId: A unique identifier for the task.
  + taskName: The name or description of the task.
  + status: The status of the task (e.g., pending, completed).

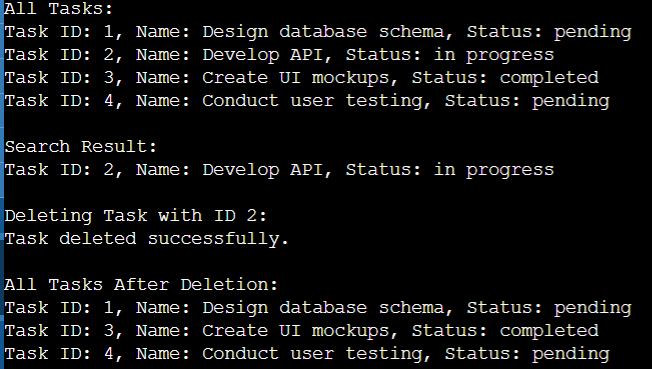
**3**. Implementation:

* Using a Singly Linked List**:** We will use a singly linked list to manage tasks.
* Methods:
  + Add**:** Adds a new task to the end of the list.
  + Search**:** Find a task by its ID.
  + Traverse: Go through and print all tasks.
  + Delete: Remove a task by its ID.

4. Analysis:

* Time Complexity:
  + Add Operation: O(1) if adding at the head, O(n) if adding at the end (without tail pointer).
  + Search Operation: O(n) as it may need to check each node.
  + Traverse Operation**:** O(n) to go through all nodes.
  + Delete Operation**:** O(n) as it may need to find the node first.

Sample Output:



Exercise 6: Library Management System

To develop a library management system where users can search books by title or author

1. Understand Search Algorithms:

* Linear Search:
  + Searches through each element in the list one by one.
  + Simple but can be slow for large lists.
  + Time Complexity: O(n), where n is the number of elements.
* Binary Search:
  + Searches by repeatedly dividing the sorted list in half.
  + Fast but requires the list to be sorted.
  + Time Complexity: O(log n), where n is the number of elements.

2. Setup:

* Book Class: This class will hold details about a book such as:
  + bookId: A unique identifier for the book.
  + title: The title of the book.
  + author: The author of the book.

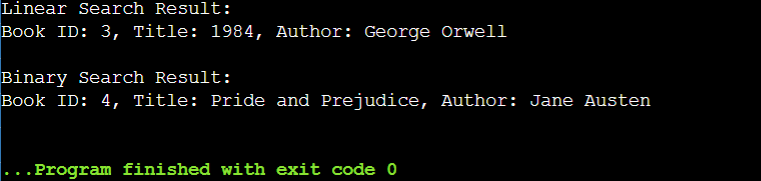
3. Implementation:

* Using Linear Search**:** We will implement a method to find books by title using linear search.
* Using Binary Search**:** We will implement a method to find books by title using binary search, assuming the list is sorted by title.

4.Analysis:

* Time Complexity:
  + Linear Search: O(n) – Searches each element one by one.
  + Binary Search: O(log n) – Divides the list in half each time.
* When to Use Each Algorithm:
  + Linear Search**:** Use when the list is small or unsorted.
  + Binary Search**:** Use when the list is large and sorted for faster search times.

Sample output:



Exercise 7: Financial Forecasting

To develop a financial forecasting tool that predicts future values based on past data:

1. Understand Recursive Algorithms:
   * Concept of Recursion:
     + Recursion is a method of solving a problem where the solution involves solving smaller instances of the same problem. A recursive function calls itself to solve these smaller instances.
     + Recursion can simplify complex problems by breaking them down into simpler sub-problems.
2. Setup:
   * Future Value Calculation Method:
     + Created a method that uses a recursive approach to calculate the future value of an investment based on past growth rates.
3. Implementation:
   * Recursive Algorithm:
     + Implemented a recursive algorithm that predicts future values based on past growth rates.
4. Analysis:

TimeComplexityofcalculateFutureValue **:** O(n)

* The function makes one recursive call per year, resulting in a linear time complexity with respect to the number of years.

Sample output:

