**Exercise 1: Inventory Management System**

**Scenario:**

You are developing an inventory management system for a warehouse. Efficient data storage and retrieval are crucial.

**Steps:**

1. **Understand the Problem:**
   * Explain why data structures and algorithms are essential in handling large inventories.
   * Discuss the types of data structures suitable for this problem.
2. **Setup:**
   * Create a new project for the inventory management system.
3. **Implementation:**
   * Define a class Product with attributes like **productId**, **productName**, **quantity**, and **price**.
   * Choose an appropriate data structure to store the products (e.g., ArrayList, HashMap).
   * Implement methods to add, update, and delete products from the inventory.
4. **Analysis:**
   * Analyze the time complexity of each operation (add, update, delete) in your chosen data structure.
   * Discuss how you can optimize these operations.

**SOLUTION:**

**Product.java**

**package** com.inventory.model;

**public** **class** Product {

**private** String productId;

**private** String productName;

**private** **int** quantity;

**private** **double** price;

**public** Product(String productId, String productName, **int** quantity, **double** price) {

**this**.productId = productId;

**this**.productName = productName;

**this**.quantity = quantity;

**this**.price = price;

}

**public** String getProductId() {

**return** productId;

}

**public** String getProductName() {

**return** productName;

}

**public** **int** getQuantity() {

**return** quantity;

}

**public** **double** getPrice() {

**return** price;

}

**public** **void** setQuantity(**int** quantity) {

**if** (quantity >= 0) {

**this**.quantity = quantity;

} **else** {

System.***err***.println("Warning: Quantity cannot be negative for product " + productId);

}

}

**public** **void** setPrice(**double** price) {

**if** (price >= 0) {

**this**.price = price;

} **else** {

System.***err***.println("Warning: Price cannot be negative for product " + productId);

}

}

@Override

**public** String toString() {

**return** "Product [ID=" + productId + ", Name=" + productName +

", Quantity=" + quantity + ", Price=$" + String.*format*("%.2f", price) + "]";

}

@Override

**public** **boolean** equals(Object o) {

**if** (**this** == o) **return** **true**;

**if** (o == **null** || getClass() != o.getClass()) **return** **false**;

Product product = (Product) o;

**return** productId.equals(product.productId);

}

@Override

**public** **int** hashCode() {

**return** productId.hashCode();

}

}

**InventoryManager.java**

**package** com.inventory.model;

**import** com.inventory.model.Product;

**import** java.util.HashMap;

**import** java.util.Map;

**import** java.util.Set;

**public** **class** InventoryManager {

**private** Map<String, Product> inventory;

**public** InventoryManager() {

**this**.inventory = **new** HashMap<>();

}

**public** **boolean** addProduct(Product product) {

**if** (inventory.containsKey(product.getProductId())) {

System.***out***.println("Error: Product with ID " + product.getProductId() + " already exists.");

**return** **false**;

}

inventory.put(product.getProductId(), product);

System.***out***.println("Added: " + product.getProductName() + " (ID: " + product.getProductId() + ")");

**return** **true**;

}

**public** **boolean** updateProduct(String productId, **int** newQuantity, **double** newPrice) {

Product product = inventory.get(productId);

**if** (product == **null**) {

System.***out***.println("Error: Product with ID " + productId + " not found for update.");

**return** **false**;

}

**if** (newQuantity != -1) {

product.setQuantity(newQuantity);

}

**if** (newPrice != -1.0) {

product.setPrice(newPrice);

}

System.***out***.println("Updated product ID " + productId);

**return** **true**;

}

**public** **boolean** deleteProduct(String productId) {

**if** (inventory.remove(productId) != **null**) {

System.***out***.println("Deleted product with ID: " + productId);

**return** **true**;

}

System.***out***.println("Error: Product with ID " + productId + " not found for deletion.");

**return** **false**;

}

**public** Product getProduct(String productId) {

**return** inventory.get(productId);

}

**public** **void** listAllProducts() {

**if** (inventory.isEmpty()) {

System.***out***.println("Inventory is empty.");

**return**;

}

System.***out***.println("\n--- Current Inventory ---");

**for** (Product product : inventory.values()) {

System.***out***.println(product);

}

System.***out***.println("-------------------------");

}

**public** **int** getTotalProductCount() {

**return** inventory.size();

}

}

**InventoryTest.java**

**package** com.inventory.model;

**import** com.inventory.model.InventoryManager;

**import** com.inventory.model.Product;

**public** **class** InventoryTest {

**public** **static** **void** main(String[] args) {

InventoryManager manager = **new** InventoryManager();

System.***out***.println("--- Adding Products ---");

manager.addProduct(**new** Product("P001", "Laptop", 10, 1200.00));

manager.addProduct(**new** Product("P002", "Mouse", 50, 25.50));

manager.addProduct(**new** Product("P003", "Keyboard", 30, 75.00));

manager.addProduct(**new** Product("P001", "Duplicate Laptop", 5, 1100.00));

manager.listAllProducts();

System.***out***.println("\n--- Getting Product P002 ---");

Product p002 = manager.getProduct("P002");

**if** (p002 != **null**) {

System.***out***.println("Found: " + p002);

} **else** {

System.***out***.println("Product P002 not found.");

}

System.***out***.println("\n--- Getting Product P999 (Non-existent) ---");

Product p999 = manager.getProduct("P999");

**if** (p999 != **null**) {

System.***out***.println("Found: " + p999);

} **else** {

System.***out***.println("Product P999 not found.");

}

System.***out***.println("\n--- Updating Products ---");

manager.updateProduct("P001", 15, -1.0);

manager.updateProduct("P003", -1, 80.00);

manager.updateProduct("P002", 45, 27.00);

manager.updateProduct("P005", 10, 100.00);

manager.listAllProducts();

System.***out***.println("\n--- Deleting Products ---");

manager.deleteProduct("P003");

manager.deleteProduct("P005");

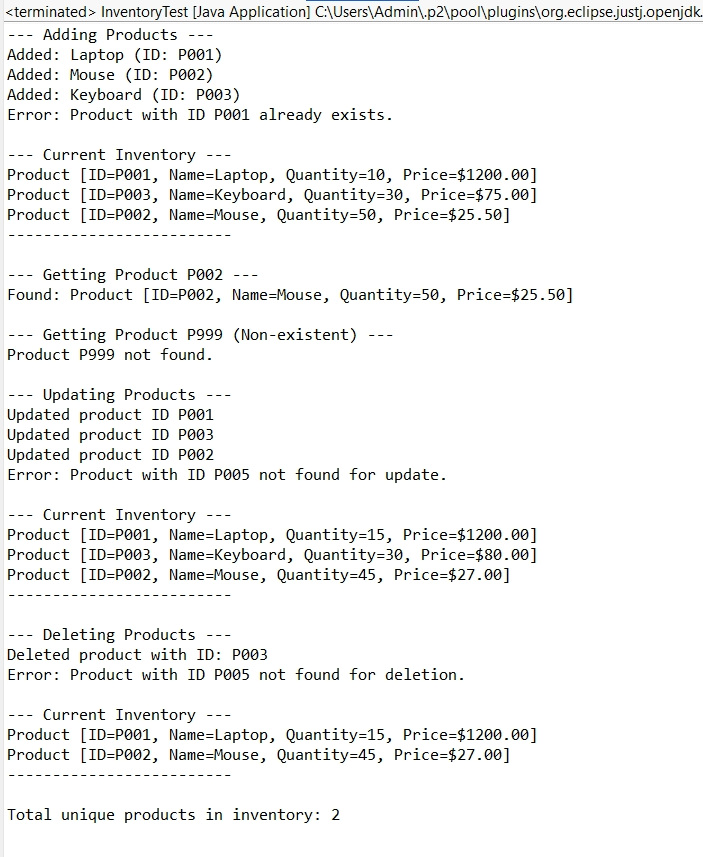
manager.listAllProducts();

System.***out***.println("\nTotal unique products in inventory: " + manager.getTotalProductCount());

}

}

**OUTPUT:**



**Exercise 2: E-commerce Platform Search Function**

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Steps:**

1. **Understand Asymptotic Notation:**
   * Explain Big O notation and how it helps in analyzing algorithms.
   * Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.
   * Discuss which algorithm is more suitable for your platform and why.

**SOLUTION:**

**Big O Notation**

In programming, Big O describes how an algorithm's run time or space grows with the size of the input n.

|  |  |  |
| --- | --- | --- |
| **Case** | **Description** | **Example** |
| Best Case | Minimal time to search (e.g., first element match) | O(1) |
| Average Case | Average position match | O(n) for linear, O(log n) for binary |
| Worst Case | Last or no match | O(n) for linear, O(log n) for binary |

**Product.java**

**package** com.ecommerce.search;

**public** **class** Product {

**private** **int** productId;

**private** String productName;

**private** String category;

**public** Product(**int** productId, String productName, String category) {

**this**.productId = productId;

**this**.productName = productName;

**this**.category = category;

}

**public** **int** getProductId() {

**return** productId;

}

**public** String getProductName() {

**return** productName;

}

**public** String getCategory() {

**return** category;

}

@Override

**public** String toString() {

**return** "[" + productId + "] " + productName + " - " + category;

}

}

**SearchService.java**

**package** com.ecommerce.search;

**public** **class** SearchService {

**public** **static** Product linearSearch(Product[] products, String targetName) {

**for** (Product p : products) {

**if** (p.getProductName().equalsIgnoreCase(targetName)) {

**return** p;

}

}

**return** **null**;

}

**public** **static** Product binarySearch(Product[] products, String targetName) {

**int** low = 0, high = products.length - 1;

**while** (low <= high) {

**int** mid = (low + high) / 2;

**int** compare = products[mid].getProductName().compareToIgnoreCase(targetName);

**if** (compare == 0) {

**return** products[mid];

} **else** **if** (compare < 0) {

low = mid + 1;

} **else** {

high = mid - 1;

}

}

**return** **null**;

}

}

**SearchTest.java**

**package** com.ecommerce.search;

**import** java.util.Arrays;

**import** java.util.Comparator;

**public** **class** SearchTest {

**public** **static** **void** main(String[] args) {

Product[] products = {

**new** Product(11, "Sandals", "Footwear"),

**new** Product(12, "Frock", "Clothing"),

**new** Product(13, "Mouse", "Electronics"),

**new** Product(14, "TextBooks", "Education"),

**new** Product(15, "Keychain", "Accessories")

};

String target1 = "Mouse";

Product result1 = SearchService.*linearSearch*(products, target1);

System.***out***.println("Linear Search Result: " + (result1 != **null** ? result1 : "Product not found"));

Arrays.*sort*(products, Comparator.*comparing*(Product::getProductName));

String target2 = "Mouse";

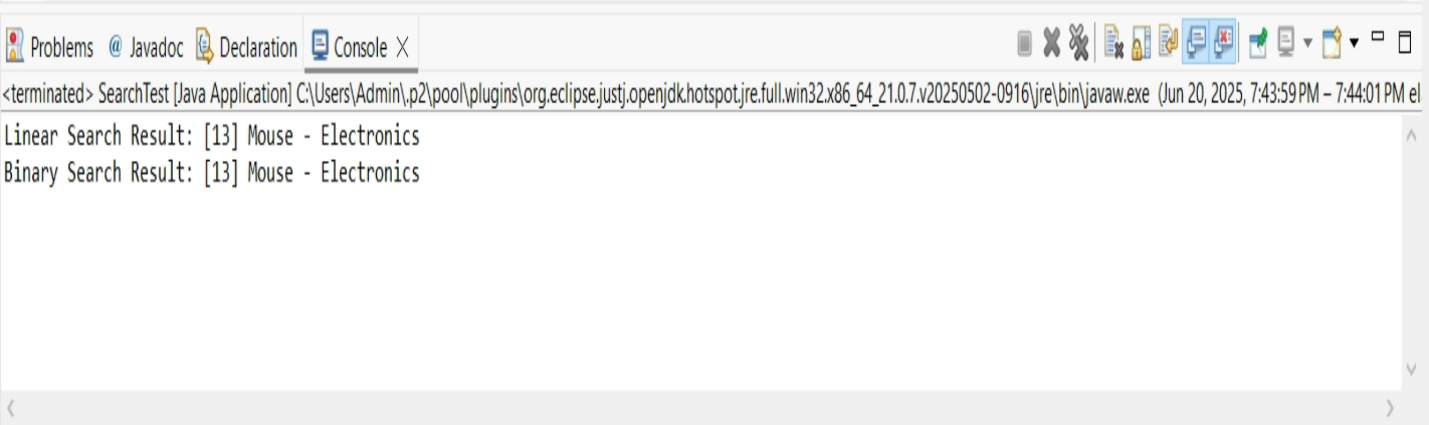
Product result2 = SearchService.*binarySearch*(products, target2);

System.***out***.println("Binary Search Result: " + (result2 != **null** ? result2 : "Product not found"));

}

}

**OUTPUT:**

****

### Comparing Time Complexity

|  |  |  |
| --- | --- | --- |
| **Criteria** | **Linear Search** | **Binary Search** |
| **Time Complexity** | O(n) | O(log n) |
| **Best Case** | O(1) (first element match) | O(1) (middle element match) |
| **Average Case** | O(n/2) → O(n) | O(log n) |
| **Worst Case** | O(n) (last or no match) | O(log n) |
| **Data Requirement** | Works on unsorted data | Requires sorted data |
| **Implementation** | Simple | Slightly more complex |
| **Flexibility** | Good for small or dynamic collections | Best for large, static, sorted collections |

### B. Which Algorithm is More Suitable for Your Platform?

1. **Large Product Catalogs**: E-commerce platforms typically store thousands or millions of products. Binary search’s O(log n) time makes it faster and scalable.
2. **Sorted Indexes Are Common**: Products can be sorted by name, category, or price, enabling the use of binary search or tree-based structures.
3. **High Performance Is Critical**: Fast search leads to better user experience and conversion rates.

#### **When Linear Search Might Be Used**:

* If the dataset is **small**.
* If the data **is not sorted** and you want a **quick and simple implementation**.
* For **temporary lists** or **testing purposes**.

**Exercise 3: Sorting Customer Orders**

**Scenario:**

You are tasked with sorting customer orders by their total price on an e-commerce platform. This helps in prioritizing high-value orders.

**Steps:**

1. **Understand Sorting Algorithms:**
   * Explain different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort).
2. **Setup:**
   * Create a class **Order** with attributes like **orderId**, **customerName**, and **totalPrice**.
3. **Implementation:**
   * Implement **Bubble Sort** to sort orders by **totalPrice**.
   * Implement **Quick Sort** to sort orders by **totalPrice**.
4. **Analysis:**
   * Compare the performance (time complexity) of Bubble Sort and Quick Sort.
   * Discuss why Quick Sort is generally preferred over Bubble Sort.

**SOLUTION:**

**Order.java**

**package** com.orders.model;

**public** **class** Order {

**private** String orderId;

**private** String customerName;

**private** **double** totalPrice;

**public** Order(String orderId, String customerName, **double** totalPrice) {

**this**.orderId = orderId;

**this**.customerName = customerName;

**this**.totalPrice = totalPrice;

}

**public** String getOrderId() {

**return** orderId;

}

**public** String getCustomerName() {

**return** customerName;

}

**public** **double** getTotalPrice() {

**return** totalPrice;

}

@Override

**public** String toString() {

**return** "Order [ID=" + orderId + ", Customer=" + customerName + ", Total Price=$" + String.*format*("%.2f", totalPrice) + "]";

}

}

**OrderSorter.java**

**package** com.orders.model;

**import** com.orders.model.Order;

**import** java.util.List;

**import** java.util.Collections;

**public** **class** OrderSorter {

**public** **static** **void** bubbleSort(List<Order> orders) {

**int** n = orders.size();

**for** (**int** i = 0; i < n - 1; i++) {

**for** (**int** j = 0; j < n - i - 1; j++) {

**if** (orders.get(j).getTotalPrice() > orders.get(j + 1).getTotalPrice()) {

Collections.*swap*(orders, j, j + 1);

}

}

}

}

**public** **static** **void** quickSort(List<Order> orders) {

**if** (orders == **null** || orders.size() <= 1) {

**return**;

}

*quickSort*(orders, 0, orders.size() - 1);

}

**private** **static** **void** quickSort(List<Order> orders, **int** low, **int** high) {

**if** (low < high) {

**int** pi = *partition*(orders, low, high);

*quickSort*(orders, low, pi - 1);

*quickSort*(orders, pi + 1, high);

}

}

**private** **static** **int** partition(List<Order> orders, **int** low, **int** high) {

**double** pivot = orders.get(high).getTotalPrice();

**int** i = (low - 1);

**for** (**int** j = low; j < high; j++) {

**if** (orders.get(j).getTotalPrice() <= pivot) {

i++;

Collections.*swap*(orders, i, j);

}

}

Collections.*swap*(orders, i + 1, high);

**return** i + 1;

}

}

**OrderSortingTest.java**

**package** com.orders.model;

**import** com.orders.model.Order;

**import** com.orders.model.OrderSorter;

**import** java.util.ArrayList;

**import** java.util.Arrays;

**import** java.util.List;

**public** **class** OrderSortingTest {

**public** **static** **void** main(String[] args) {

List<Order> orders = Arrays.*asList*(

**new** Order("O101", "Alice", 150.75),

**new** Order("O105", "Bob", 50.00),

**new** Order("O103", "Charlie", 200.00),

**new** Order("O102", "David", 12.30),

**new** Order("O104", "Eve", 150.75),

**new** Order("O106", "Frank", 99.99)

);

System.***out***.println("--- Original Orders ---");

*printOrders*(orders);

List<Order> bubbleSortOrders = **new** ArrayList<>(orders);

System.***out***.println("\n--- Sorting using Bubble Sort ---");

**long** startTimeBubble = System.*nanoTime*();

OrderSorter.*bubbleSort*(bubbleSortOrders);

**long** endTimeBubble = System.*nanoTime*();

*printOrders*(bubbleSortOrders);

System.***out***.println("Bubble Sort Time: " + (endTimeBubble - startTimeBubble) + " ns");

List<Order> quickSortOrders = **new** ArrayList<>(orders);

System.***out***.println("\n--- Sorting using Quick Sort ---");

**long** startTimeQuick = System.*nanoTime*();

OrderSorter.*quickSort*(quickSortOrders);

**long** endTimeQuick = System.*nanoTime*();

*printOrders*(quickSortOrders);

System.***out***.println("Quick Sort Time: " + (endTimeQuick - startTimeQuick) + " ns");

}

**private** **static** **void** printOrders(List<Order> orders) {

**for** (Order order : orders) {

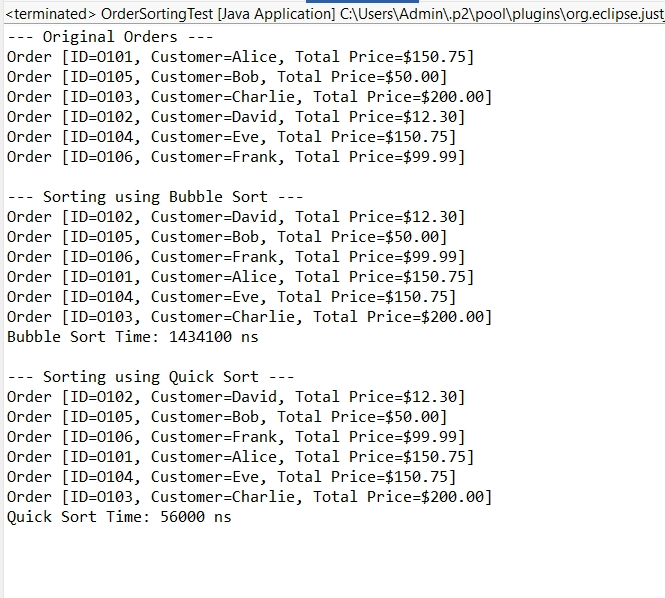
System.***out***.println(order);

}

}

}

**OUTPUT:**



**Exercise 4: Employee Management System**

**Scenario:**

You are developing an employee management system for a company. Efficiently managing employee records is crucial.

**Steps:**

1. **Understand Array Representation:**
   * Explain how arrays are represented in memory and their advantages.
2. **Setup:**
   * Create a class Employee with attributes like **employeeId**, **name**, **position**, and **salary**.
3. **Implementation:**
   * Use an array to store employee records.
   * Implement methods to **add**, **search**, **traverse**, and **delete** employees in the array.
4. **Analysis:**
   * Analyze the time complexity of each operation (add, search, traverse, delete).
   * Discuss the limitations of arrays and when to use them.

**SOLUTION:**

## **Understand Array Representation**

### How arrays are represented in memory:

* Arrays are **contiguous blocks of memory**.
* Each element is accessed using an **index**.
* All elements must be of the **same data type**.
* Accessing any element is **O(1)** (constant time).

### Advantages:

* Fast data access via indexing.
* Simple and efficient for fixed-size collections.
* Low overhead compared to dynamic data structures.

**Employee.java**

**package** com.company.ems;

**public** **class** Employee {

**private** **int** employeeId;

**private** String name;

**private** String position;

**private** **double** salary;

**public** Employee(**int** employeeId, String name, String position, **double** salary) {

**this**.employeeId = employeeId;

**this**.name = name;

**this**.position = position;

**this**.salary = salary;

}

**public** **int** getEmployeeId() {

**return** employeeId;

}

**public** **void** display() {

System.***out***.println("ID: " + employeeId + ", Name: " + name +

", Position: " + position + ", Salary: " + salary);

}

}

**EmployeeManager.java**

**package** com.company.ems;

**public** **class** EmployeeManager {

**private** **static** Employee[] *employees* = **new** Employee[100];

**private** **static** **int** *count* = 0;

**public** **static** **void** addEmployee(Employee emp) {

**if** (*count* < *employees*.length) {

*employees*[*count*++] = emp;

System.***out***.println("Employee added successfully.");

} **else** {

System.***out***.println("Error: Employee list is full.");

}

}

**public** **static** **void** searchEmployee(**int** id) {

**for** (**int** i = 0; i < *count*; i++) {

**if** (*employees*[i].getEmployeeId() == id) {

System.***out***.println("Employee Found:");

*employees*[i].display();

**return**;

}

}

System.***out***.println("Employee with ID " + id + " not found.");

}

**public** **static** **void** displayAllEmployees() {

**if** (*count* == 0) {

System.***out***.println("No employees to display.");

} **else** {

System.***out***.println("All Employee Records:");

**for** (**int** i = 0; i < *count*; i++) {

*employees*[i].display();

}

}

}

**public** **static** **void** deleteEmployee(**int** id) {

**for** (**int** i = 0; i < *count*; i++) {

**if** (*employees*[i].getEmployeeId() == id) {

**for** (**int** j = i; j < *count* - 1; j++) {

*employees*[j] = *employees*[j + 1];

}

*employees*[--*count*] = **null**;

System.***out***.println("Employee with ID " + id + " deleted.");

**return**;

}

}

System.***out***.println("Employee with ID " + id + " not found.");

}

**public** **static** **void** main(String[] args) {

*addEmployee*(new Employee(1, "Alice", "Developer", 50000));

*addEmployee*(new Employee(2, "Bob", "Manager", 70000));

*addEmployee*(new Employee(3, "Charlie", "Analyst", 60000));

System.***out***.println();

*displayAllEmployees*();

System.***out***.println("\nSearching for Employee ID 2:");

*searchEmployee*(2);

System.***out***.println("\nDeleting Employee ID 1:");

*deleteEmployee*(1);

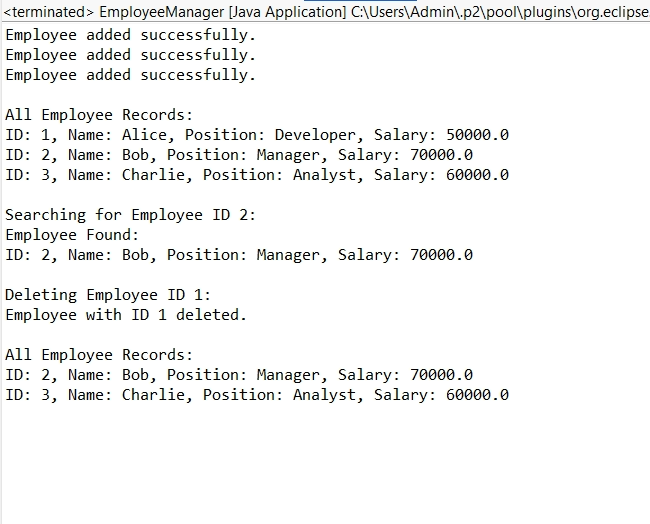
System.***out***.println();

*displayAllEmployees*();

}

}

**OUTPUT:**



## **Time Complexity:**

|  |  |  |
| --- | --- | --- |
| **Operation** | **Description** | **Time Complexity** |
| Add | Insert at end | O(1) |
| Search | Linear search by ID | O(n) |
| Traverse | Print all records | O(n) |
| Delete | Shift array elements after deletion | O(n) |

## **Limitations of Using Arrays**

* Fixed size (not dynamic).
* Insertion and deletion (not at end) are costly.
* Manual shifting required for deletion.
* Better alternatives: ArrayList, LinkedList.

**Exercise 5: Task Management System**

**Scenario:**

You are developing a task management system where tasks need to be added, deleted, and traversed efficiently.

**Steps:**

1. **Understand Linked Lists:**
   * Explain the different types of linked lists (Singly Linked List, Doubly Linked List).
2. **Setup:**
   * Create a class **Task** with attributes like **taskId**, **taskName**, and **status**.
3. **Implementation:**
   * Implement a singly linked list to manage tasks.
   * Implement methods to **add**, **search**, **traverse**, and **delete** tasks in the linked list.
4. **Analysis:**
   * Analyze the time complexity of each operation.
   * Discuss the advantages of linked lists over arrays for dynamic data.

**SOLUTION:**

### ****Understand Linked Lists****

#### **Types of Linked Lists:**

* **Singly Linked List**: Each node stores data and a reference to the next node.
* **Doubly Linked List**: Each node stores data, a reference to the next **and** previous nodes.

#### **Singly Linked List Benefits:**

* Dynamic size (no need to pre-allocate memory).
* Efficient insert/delete at beginning or middle (no shifting like arrays).
* Slower access to random elements (no index-based access).

**Task.java**

**package** com.company.tms;

**public** **class** Task {

**int** taskId;

String taskName;

String status;

Task next;

**public** Task(**int** taskId, String taskName, String status) {

**this**.taskId = taskId;

**this**.taskName = taskName;

**this**.status = status;

**this**.next = **null**;

}

**public** **void** display() {

System.***out***.println("Task ID: " + taskId + ", Name: " + taskName + ", Status: " + status);

}

}

**TaskManager.java**

**package** com.company.tms;

**public** **class** TaskManager {

Task head = **null**;

**public** **void** addTask(**int** taskId, String taskName, String status) {

Task newTask = **new** Task(taskId, taskName, status);

**if** (head == **null**) {

head = newTask;

} **else** {

Task temp = head;

**while** (temp.next != **null**) {

temp = temp.next;

}

temp.next = newTask;

}

System.***out***.println("Task added.");

}

**public** **void** searchTask(**int** taskId) {

Task temp = head;

**while** (temp != **null**) {

**if** (temp.taskId == taskId) {

System.***out***.println("Task found:");

temp.display();

**return**;

}

temp = temp.next;

}

System.***out***.println("Task with ID " + taskId + " not found.");

}

**public** **void** displayTasks() {

**if** (head == **null**) {

System.***out***.println("No tasks to display.");

**return**;

}

Task temp = head;

System.***out***.println("All Tasks:");

**while** (temp != **null**) {

temp.display();

temp = temp.next;

}

}

**public** **void** deleteTask(**int** taskId) {

**if** (head == **null**) {

System.***out***.println("Task list is empty.");

**return**;

}

**if** (head.taskId == taskId) {

head = head.next;

System.***out***.println("Task deleted.");

**return**;

}

Task current = head;

Task previous = **null**;

**while** (current != **null** && current.taskId != taskId) {

previous = current;

current = current.next;

}

**if** (current == **null**) {

System.***out***.println("Task with ID " + taskId + " not found.");

} **else** {

previous.next = current.next;

System.***out***.println("Task deleted.");

}

}

**public** **static** **void** main(String[] args) {

TaskManager manager = **new** TaskManager();

manager.addTask(1, "Design UI", "Pending");

manager.addTask(2, "Develop Backend", "In Progress");

manager.addTask(3, "Test System", "Pending");

System.***out***.println();

manager.displayTasks();

System.***out***.println("\nSearching for Task ID 2:");

manager.searchTask(2);

System.***out***.println("\nDeleting Task ID 1:");

manager.deleteTask(1);

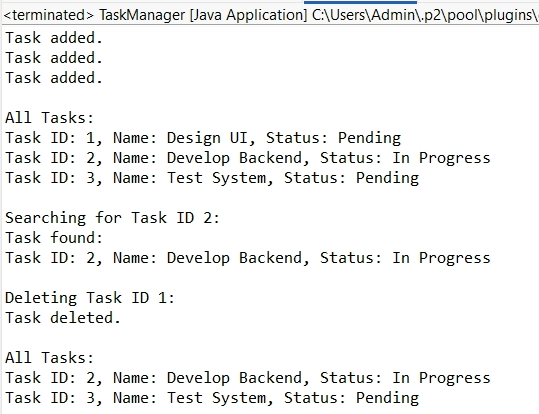
System.***out***.println();

manager.displayTasks();

}

}

**OUTPUT:**



## **Analysis**

|  |  |
| --- | --- |
| **Operation** | **Time Complexity** |
| Add Task | O(n) (add at end) |
| Search Task | O(n) |
| Display Tasks | O(n) |
| Delete Task | O(n) |

### Advantages of Linked Lists Over Arrays

|  |  |
| --- | --- |
| **Linked List** | **Array** |
| Dynamic size | Fixed size |
| Easy insertion/deletion at front | Costly shifting |
| No wasted memory | May allocate more than required |
| No random access | Fast random access (O(1)) |

**Exercise 6: Library Management System**

**Scenario:**

You are developing a library management system where users can search for books by title or author.

**Steps:**

1. **Understand Search Algorithms:**
   * Explain linear search and binary search algorithms.
2. **Setup:**
   * Create a class **Book** with attributes like **bookId**, **title**, and **author**.
3. **Implementation:**
   * Implement linear search to find books by title.
   * Implement binary search to find books by title (assuming the list is sorted).
4. **Analysis:**
   * Compare the time complexity of linear and binary search.
   * Discuss when to use each algorithm based on the data set size and order.

## **Solution:**

### ****Understand Search Algorithms****

#### **Linear Search:**

* **Checks each element one by one** until the target is found.
* Works on **unsorted** lists.
* Use when: the list is **small or unsorted.**

#### **Binary Search:**

* Works by **repeatedly dividing a sorted list** into halves.
* Must be used on a **sorted list.**
* Use when: the list is **large and sorted.**

**Book.java**

**package** com.company.lms;

**public** **class** Book {

**int** bookId;

String title;

String author;

**public** Book(**int** bookId, String title, String author) {

**this**.bookId = bookId;

**this**.title = title;

**this**.author = author;

}

**public** **void** display() {

System.***out***.println("ID: " + bookId + ", Title: " + title + ", Author: " + author);

}

**public** String getTitle() {

**return** title.toLowerCase();

}

}

**Library.java**

**package** com.company.lms;

**import** java.util.Arrays;

**import** java.util.Comparator;

**public** **class** Library {

Book[] books;

**int** count;

**public** Library(**int** size) {

books = **new** Book[size];

count = 0;

}

**public** **void** addBook(Book book) {

**if** (count < books.length) {

books[count++] = book;

} **else** {

System.***out***.println("Library is full.");

}

}

**public** **void** linearSearchByTitle(String title) {

**boolean** found = **false**;

**for** (**int** i = 0; i < count; i++) {

**if** (books[i].getTitle().equals(title.toLowerCase())) {

System.***out***.println("Book found using Linear Search:");

books[i].display();

found = **true**;

**break**;

}

}

**if** (!found) {

System.***out***.println("Book not found using Linear Search.");

}

}

**public** **void** sortBooksByTitle() {

Arrays.*sort*(books, 0, count, Comparator.*comparing*(Book::getTitle));

}

**public** **void** binarySearchByTitle(String title) {

**int** left = 0;

**int** right = count - 1;

title = title.toLowerCase();

**while** (left <= right) {

**int** mid = left + (right - left) / 2;

String midTitle = books[mid].getTitle();

**if** (midTitle.equals(title)) {

System.***out***.println("Book found using Binary Search:");

books[mid].display();

**return**;

} **else** **if** (midTitle.compareTo(title) < 0) {

left = mid + 1;

} **else** {

right = mid - 1;

}

}

System.***out***.println("Book not found using Binary Search.");

}

**public** **void** displayAllBooks() {

System.***out***.println("Books in Library:");

**for** (**int** i = 0; i < count; i++) {

books[i].display();

}

}

**public** **static** **void** main(String[] args) {

Library library = **new** Library(10);

library.addBook(**new** Book(1, "The Alchemist", "Paulo Coelho"));

library.addBook(**new** Book(2, "Clean Code", "Robert C. Martin"));

library.addBook(**new** Book(3, "Harry Potter", "J.K. Rowling"));

library.addBook(**new** Book(4, "Atomic Habits", "James Clear"));

System.***out***.println();

library.displayAllBooks();

System.***out***.println("\n🔎 Linear Search for 'Clean Code':");

library.linearSearchByTitle("Clean Code");

library.sortBooksByTitle();

System.***out***.println("\n🔎 Binary Search for 'Clean Code':");

library.binarySearchByTitle("Clean Code");

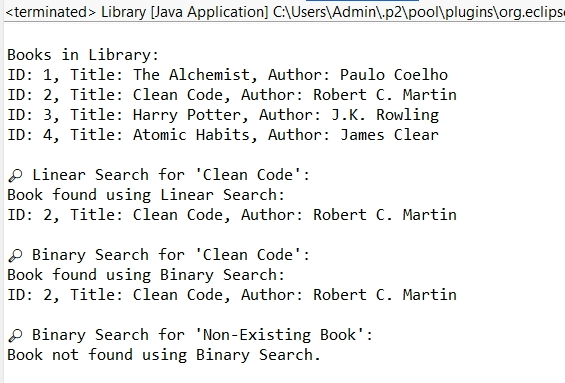
System.***out***.println("\n🔎 Binary Search for 'Non-Existing Book':");

library.binarySearchByTitle("Invisible Man");

}

}

**OUTPUT:**



## **Analysis**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Search Type** | **Time Complexity** | **Works on Sorted?** | **Pros** | **Cons** |
| **Linear Search** | O(n) | No | Simple, works on unsorted data | Slow on large datasets |
| **Binary Search** | O(log n) | Yes | Fast on large sorted datasets | Requires sorted data |

### When to Use Which?

|  |  |
| --- | --- |
| **Scenario** | **Use** |
| Small or unsorted dataset | Linear Search |
| Large and sorted dataset | Binary Search |
| Frequent updates to list (unsorted) | Linear Search |
| Static, sorted list for read-heavy apps | Binary Search |

**Exercise 7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

**Steps:**

1. **Understand Recursive Algorithms:**
   * Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.
   * Explain how to optimize the recursive solution to avoid excessive computation.

**SOLUTION:**

### Concept of Recursion:

Recursion is when a function **calls itself** to solve smaller subproblems. It's often useful for problems like:

* Factorial calculation
* Fibonacci series
* Tree/graph traversal
* Financial predictions with recurring calculations (like compound interest or growth modeling)

**Example :**  
To predict value in n years:  
**futureValue(n) = futureValue(n-1) × (1 + growthRate)**

**ForecastCalculator.java**

**package** com.financial.forecast;

**public** **class** ForecastCalculator {

**public** **static** **double** calculateFutureValue(**double** currentValue, **double** growthRate, **int** years) {

**if** (years == 0) {

**return** currentValue;

}

**return** *calculateFutureValue*(currentValue \* (1 + growthRate), growthRate, years - 1);

}

**public** **static** **void** main(String[] args) {

**double** initialValue = 1000.0;

**double** growthRate = 0.08;

**int** years = 5;

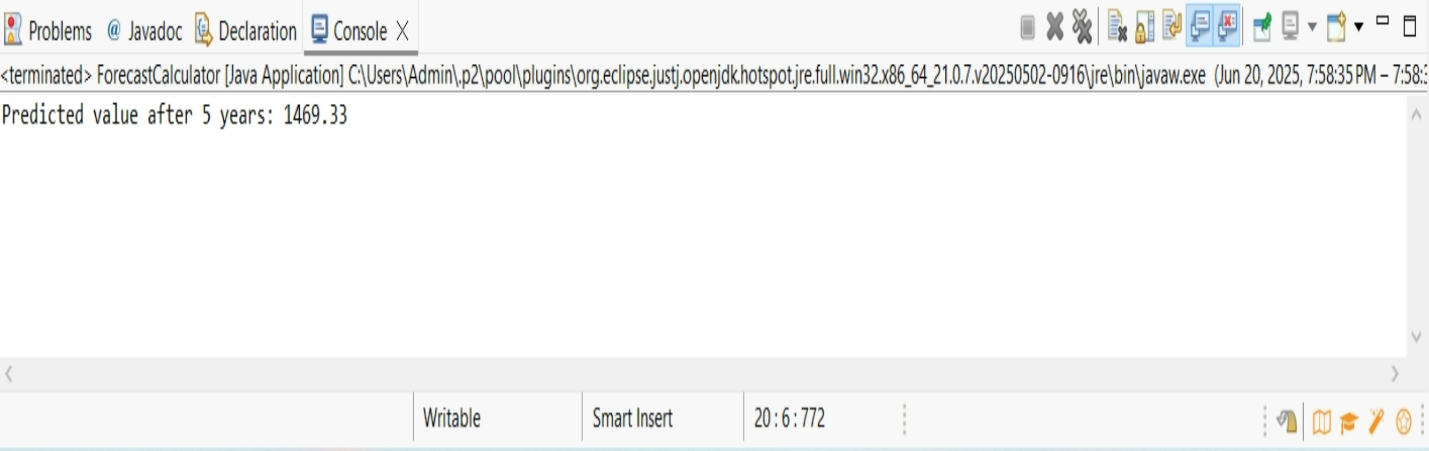
**double** predictedValue = *calculateFutureValue*(initialValue, growthRate, years);

System.***out***.printf("The Predicted value after %d years: %.2f%n", years, predictedValue);

}

}

**OUTPUT:**

****

### A. ****Time Complexity****

The recursive function calls itself **once per year**, so:

* **Time Complexity:** O(n), where n = number of years
* **Space Complexity:** O(n) (due to call stack in recursion)

### B. ****Optimization Discussion****

#### **Problem with Recursion:**

* For large n, recursion can cause **stack overflow** or unnecessary recomputation.
* Example: If we used recursion for a Fibonacci-based forecast, we’d face **exponential time** without memoization.

#### **Solution:**

1. **Use Memoization**: Store results of previous calculations.
2. **Use Iterative Approach** (preferred for long-term forecasting).