**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.**

**Introduction**

Managing inventories efficiently is a key challenge for businesses with large stocks of products.  
With growing inventories, we need a system to perform operations — adding, updating, retrieving, and removing products — quickly and efficiently.  
This is where choosing the **right data structures and algorithms** becomes crucial.

**Why Are Data Structures and Algorithms Important in Large Inventories?**

➥ Large inventories typically consist of **thousands or even millions of products**.  
➥ Operations like adding, updating, retrieving, or removing products need to be **fast and efficient** to avoid bottlenecks in the system.  
➥ If we use an inefficient data structure (like a simple array or a ArrayList) for these operations, the complexity can become O(n), which means the time taken grows **directly with the number of products**.  
➥ A more sophisticated structure, such as a HashMap, lets us perform these operations in **average O(1)** time — regardless of the total number of products.

**Types of Data Structures Suitable for the Inventory System**

**HashMap:**  
➥ Allows fast O(1) average time for adding, updating, retrieving, and removing products.  
➥ Ideal when we have a unique key (such as Product Name or Product ID) to identify each product quickly.

**ArrayList:**  
➥ Allows storage of products in a collection.  
➥ Less efficient for search and deletion, typically O(n), making it unsuitable for large inventories.

**TreeMap:**  
➥ Stores products in a sorted order by key.  
➥ Operations are O(log n), which is slightly slower than HashMap but useful when we need products in a particular order.

**Time Complexity Analysis**

| **Operation** | **Average Time** |
| --- | --- |
| **Add** | O(1) |
| **Update** | O(1) |
| **Remove** | O(1) |

**Possible Optimizations**

If we frequently need products in a **sorted order**, we might use TreeMap.  
 If we need to handle large inventories efficiently and frequently perform **look-up by key**, HashMap is a perfect match.

**Summary**

➥ The HashMap is an appropriate data structure for this inventory management system due to its O(1) average time complexity for adding, updating, and removing products.  
➥ It efficiently handles large inventories and makes retrieval and modification operations faster and more scalable.  
➥ The implementation can be further improved by directly mapping Product to String (productName) instead of using a List.

Full Code

import java.util.\*;

class ProductDetails {

String productName;

int price;

int quantity;

ProductDetails(String productName, int price, int quantity) {

this.productName = productName;

this.price = price;

this.quantity = quantity;

}

}

class InventoryManager {

static HashMap<Integer, ProductDetails> warHouse = new HashMap<>();

int productId;

String productName;

int price;

int quantity;

InventoryManager(int productId, String productName, int price, int quantity) {

this.productId = productId;

this.productName = productName;

this.price = price;

this.quantity = quantity;

}

InventoryManager(int productId) {

this.productId = productId;

}

void add() {

if (warHouse.containsKey(productId)) {

System.out.println("Product with this ID already exists. Use update instead.");

} else {

warHouse.put(productId, new ProductDetails(productName, price, quantity));

System.out.println("Product added.");

}

}

void update() {

if (warHouse.containsKey(productId)) {

warHouse.put(productId, new ProductDetails(productName, price, quantity));

System.out.println("Product updated.");

} else {

System.out.println("Product not found. Use add to create a new product.");

}

}

void delete() {

if (warHouse.containsKey(productId)) {

warHouse.remove(productId);

System.out.println("Product removed.");

} else {

System.out.println("Product ID not found.");

}

}

}

public class Main {

public static void main(String[] args) {

Scanner obj = new Scanner(System.in);

while (true) {

System.out.println("\nEnter command (add, update, delete, view, exit): ");

String cmd = obj.next();

if (cmd.equalsIgnoreCase("add") || cmd.equalsIgnoreCase("update")) {

System.out.print("Enter Product ID: ");

int productId = obj.nextInt();

System.out.print("Enter Product Name: ");

String productName = obj.next();

System.out.print("Enter Product Price: ");

int price = obj.nextInt();

System.out.print("Enter Product Quantity: ");

int quantity = obj.nextInt();

InventoryManager inv = new InventoryManager(productId, productName, price, quantity);

if (cmd.equalsIgnoreCase("add")) {

inv.add();

} else {

inv.update();

}

} else if (cmd.equalsIgnoreCase("delete")) {

System.out.print("Enter Product ID to delete: ");

int productId = obj.nextInt();

InventoryManager inv = new InventoryManager(productId);

inv.delete();

} else if (cmd.equalsIgnoreCase("view")) {

if (InventoryManager.warHouse.isEmpty()) {

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

} else {

for (Map.Entry<Integer, ProductDetails> entry : InventoryManager.warHouse.entrySet()) {

int id = entry.getKey();

ProductDetails details = entry.getValue();

System.out.println("Product ID: " + id +

", Name: " + details.productName +

", Price: " + details.price +

", Quantity: " + details.quantity);

}

}

} else if (cmd.equalsIgnoreCase("exit")) {

System.out.println("Exiting...");

break;

} else {

System.out.println("Unknown command.");

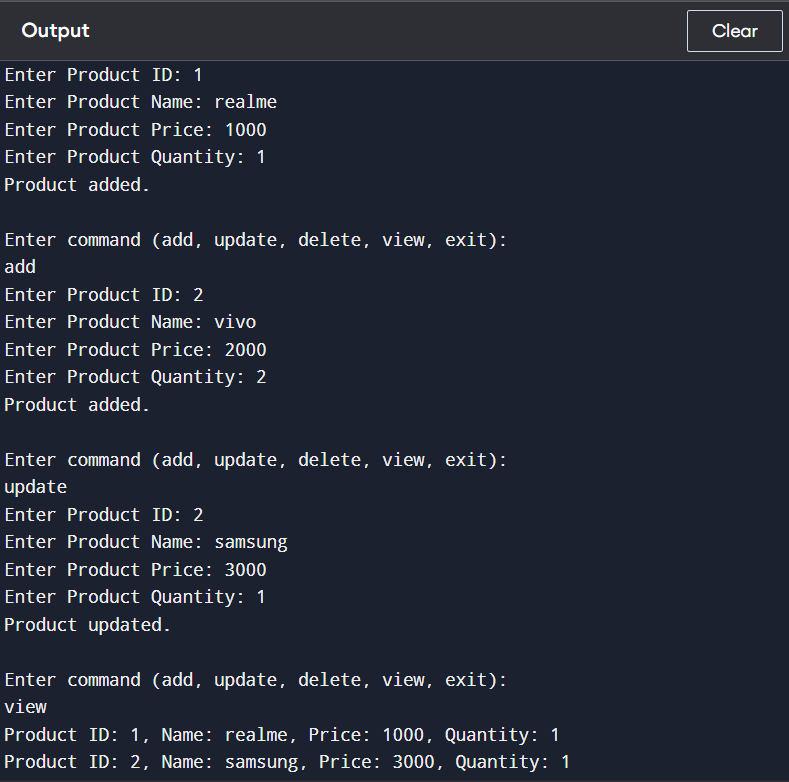
}

}

obj.close();

}

}



**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.**

Introduction  
Searching efficiently for products is a key feature for large e-commerce platforms.  
As inventories grow, we need methods to quickly find products by their IDs or attributes.

Why Are Asymotic Notation and Search Algorithm Important?

➥ Asymotic notation (Big O) lets us compare algorithm speeds, ignoring minor factors.  
➥ Large inventories can slow search if we use O(n) search.  
➥ Binary search performs faster, O(log n), if we have a **sorted array**.

Types of Search Algorithm Suitable for E-commerce:

**Linear Search:**  
➥ O(n) — simple but slow for large inventories.

**Binary Search:**  
➥ O(log n) — much faster, but requires **sorted array first**.

Time Complexity Analysis:

| **Algorithm** | **Best** | **Average** | **Worst** |
| --- | --- | --- | --- |
| Linear | O(1) | O(n) | O(n) |
| Binary | O(1) | O(log n) | O(log n) |

Possible Optimizations:

If we frequently search by productId, keep array **sorted by Id**.  
 If we frequently add and delete, consider **HashMap** instead.

Summary:

➥ Linear search is simple and effective for small or unsorted inventories.  
➥ Binary search is faster for large, **sorted inventories**.

Full Code:

import java.util.\*;

class Product {

int productId;

String productName;

String category;

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

}

public class Main {

static Product linearSearch(Product[] products, int id) {

for (Product p : products) {

if (p.productId == id) return p;

}

return null;

}

static Product binarySearch(Product[] products, int id) {

int left = 0;

int right = products.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

if (products[mid].productId == id) return products[mid];

else if (products[mid].productId < id) left = mid + 1;

else right = mid - 1;

}

return null;

}

public static void main(String[] args) {

Product[] products = {

new Product(1, "Shoes", "Fashion"),

new Product(2, "Book", "Stationery"),

new Product(3, "Watch", "Accessories"),

new Product(4, "Bag", "Fashion"),

new Product(5, "Mobile", "Electronics"),

};

Arrays.sort(products, Comparator.comparingInt(p -> p.productId));

Scanner sc = new Scanner(System.in);

System.out.println("Choose search method:");

System.out.println("1. Linear Search");

System.out.println("2. Binary Search");

System.out.print("Enter choice (1 or 2): ");

int choice = sc.nextInt();

System.out.print("Enter ProductId to search: ");

int id = sc.nextInt();

Product result = null;

if (choice == 1) {

result = linearSearch(products, id);

} else if (choice == 2) {

result = binarySearch(products, id);

} else {

System.out.println("Invalid choice.");

sc.close();

return;

}

if (result != null) {

System.out.println("Product found: " + result.productName + " (" + result.category + ")");

} else {

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

}

sc.close();

}

}

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

**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.**

Introduction  
Sorting is a key functionality when we need to prioritize orders by total price.  
For large inventories, choosing an efficient algorithm makes a huge difference in performance.

Why Are Sorting Algorithm Important?

➥ Bubble Sort is simple but O(n²), slow for large n.  
➥ Quick Sort is O(n log n), much faster and more efficient.  
➥ Sorting lets us quickly view, search, or prioritize orders.

Types of Sorting Algorithm:

**Bubble Sort:**  
➥ O(n²) — slow for large n  
➥ Suitable for small or nearly-sorted lists

**Quick Sort:**  
➥ O(n log n) — faster and more efficient  
➥ Divide-and-conquer approach

Time Complexity Analysis:

| **Algorithm** | **Best** | **Average** | **Worst** |
| --- | --- | --- | --- |
| Bubble | O(n) | O(n²) | O(n²) |
| Quick | O(n log n) | O(n log n) | O(n²) |

Possible Optimizations:

If nearly sorted, Bubble Sort performs well.  
 Otherwise, Quick Sort is more efficient.

Summary:

➥ Bubble Sort is simple but slow.  
➥ Quick Sort is faster and more scalable for large datasets.

Full Code:

import java.util.\*;

class Order {

int orderId;

String customerName;

double totalPrice;

Order(int orderId, String customerName, double totalPrice) {

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

}

public class Main {

static void bubbleSort(Order[] orders) {

int n = orders.length;

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

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

if (orders[j].totalPrice > orders[j + 1].totalPrice) {

Order temp = orders[j];

orders[j] = orders[j + 1];

orders[j + 1] = temp;

}

}

}

}

static void quickSort(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);

}

}

static int partition(Order[] orders, int low, int high) {

double pivot = orders[high].totalPrice;

int i = low - 1;

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

if (orders[j].totalPrice <= pivot) {

i++;

Order temp = orders[i];

orders[i] = orders[j];

orders[j] = temp;

}

}

Order temp = orders[i + 1];

orders[i + 1] = orders[high];

orders[high] = temp;

return i + 1;

}

static void printOrders(Order[] orders) {

for (Order o : orders) {

System.out.println("OrderID: " + o.orderId + ", Customer: " + o.customerName + ", Total: " + o.totalPrice);

}

}

public static void main(String[] args) {

Order[] orders = {

new Order(101, "Alice", 450.0),

new Order(102, "Bob", 300.0),

new Order(103, "Charlie", 700.0),

new Order(104, "Diana", 150.0)

};

Scanner sc = new Scanner(System.in);

System.out.println("Choose sorting method:");

System.out.println("1. Bubble Sort");

System.out.println("2. Quick Sort");

System.out.print("Enter choice (1 or 2): ");

int choice = sc.nextInt();

if (choice == 1) {

bubbleSort(orders);

} else if (choice == 2) {

quickSort(orders, 0, orders.length - 1);

} else {

System.out.println("Invalid choice.");

sc.close();

return;

}

System.out.println("\nSorted Orders by Total Price:");

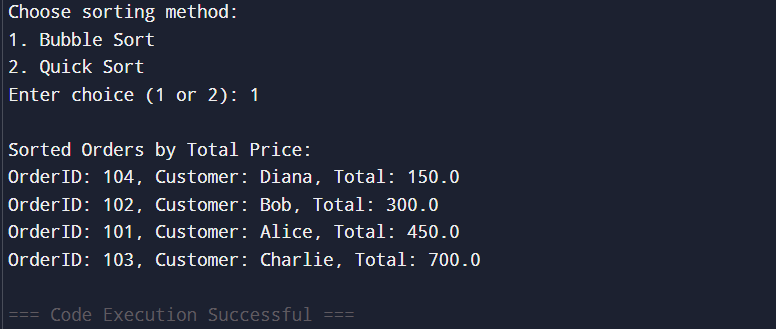
printOrders(orders);

sc.close();

}

}

Output:



A screenshot of a computer program

AI-generated content may be incorrect.

**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.**

Introduction  
Managing employees efficiently is a key requirement for companies.  
Using **arrays** is a simple way to implement this when the number of employees is small and well defined.

Why Are Arrays Important?

➥ Allows for **direct indexing**, O(1) to access.  
➥ Stores elements contiguously in memory.  
➥ But it's **fixed in size** — adding or removing is not efficient.

Types:

**Array:**  
➥ Stores all employees in a contiguous block.  
➥ Operations (search, add, delete) typically O(n) except accessing by index which is O(1).

Time Complexity Analysis:

| **Operations** | **Time** |
| --- | --- |
| add | O(1) |
| search | O(n) |
| delete | O(n) |
| view all | O(n) |

Possible Optimizations:

If frequently adding or removing, consider **ArrayList or HashMap**.

Summary:

➥ The array is simple and efficient for small, static collections.  
➥ Operations scale with n; for large datasets, it's often better to use more sophisticated structures.

Full Code:

import java.util.\*;

class Employee {

int employeeId;

String name;

String position;

double salary;

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

this.employeeId = employeeId;

this.name = name;

this.position = position;

this.salary = salary;

}

}

public class Main {

static void addEmployee(Employee[] employees, int index, Employee newEmployee) {

if (index >= 0 && index < employees.length) {

employees[index] = newEmployee;

System.out.println("Employee added at position " + index);

} else {

System.out.println("Index out of bounds. Cannot add employee.");

}

}

static Employee searchEmployee(Employee[] employees, int id) {

for (Employee e : employees) {

if (e != null && e.employeeId == id)

return e;

}

return null;

}

static void deleteEmployee(Employee[] employees, int id) {

for (int i = 0; i < employees.length; i++) {

if (employees[i] != null && employees[i].employeeId == id) {

employees[i] = null;

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

return;

}

}

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

}

static void viewAllEmployees(Employee[] employees) {

System.out.println("Current Employees:");

boolean found = false;

for (Employee e : employees) {

if (e != null) {

System.out.println(e.employeeId + " " + e.name + " " + e.position + " $" + e.salary);

found = true;

}

}

if (!found) System.out.println("No employees found.");

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

Employee[] employees = new Employee[5];

while (true) {

System.out.println("\nMenu:");

System.out.println("1. Add Employee");

System.out.println("2. Search Employee");

System.out.println("3. Delete Employee");

System.out.println("4. View All Employees");

System.out.println("5. Exit");

System.out.print("Enter choice: ");

int choice = sc.nextInt();

if (choice == 1) {

System.out.print("Enter Index (0-4): ");

int index = sc.nextInt();

System.out.print("Enter Employee ID: ");

int id = sc.nextInt();

sc.nextLine();

System.out.print("Enter Name: ");

String name = sc.nextLine();

System.out.print("Enter Position: ");

String position = sc.nextLine();

System.out.print("Enter Salary: ");

double salary = sc.nextDouble();

addEmployee(employees, index, new Employee(id, name, position, salary));

} else if (choice == 2) {

System.out.print("Enter Employee ID to search: ");

int id = sc.nextInt();

Employee emp = searchEmployee(employees, id);

if (emp != null) {

System.out.println("Found: " + emp.employeeId + " " + emp.name + " " + emp.position + " $" + emp.salary);

} else {

System.out.println("Employee not found.");

}

} else if (choice == 3) {

System.out.print("Enter Employee ID to delete: ");

int id = sc.nextInt();

deleteEmployee(employees, id);

} else if (choice == 4) {

viewAllEmployees(employees);

} else if (choice == 5) {

System.out.println("Exiting...");

break;

} else {

System.out.println("Invalid choice.");

}

}

sc.close();

}

}

A screenshot of a computer

AI-generated content may be incorrect.

**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.**

Introduction  
Managing tasks efficiently is a key requirement in a to-do application.  
Using **ArrayList** lets us easily add, view, and delete tasks without worrying about array size.

Why Are ArrayList and List Important?

➥ Allows **dynamic resizing**, adding or removing elements without needing a new array.  
➥ Operations like adding, removing, and retrieving are O(1) or O(n) depending on the action.

Types:

**ArrayList:**  
➥ Stores elements in a resizable array.  
➥ Allows fast addition at the end; removal and search are O(n).

Time Complexity Analysis:

| **Operations** | **Time** |
| --- | --- |
| add | O(1) |
| view | O(n) |
| complete | O(n) |
| delete | O(n) |

Possible Optimizations:

If frequently accessing by ID, consider **HashMap** instead.

Summary:

➥ ArrayList is convenient for small and medium-sized task lists.  
➥ Allows adding, removing, and updating efficiently.

Full Code:

import java.util.Scanner;

class Task {

int taskId;

String description;

boolean isDone;

Task next;

Task(int taskId, String description) {

this.taskId = taskId;

this.description = description;

this.isDone = false;

this.next = null;

}

}

class TaskList {

Task head;

void addTask(int taskId, String description) {

Task newTask = new Task(taskId, description);

if (head == null) {

head = newTask;

} else {

Task temp = head;

while (temp.next != null) {

temp = temp.next;

}

temp.next = newTask;

}

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

}

void viewAllTasks() {

Task temp = head;

if (temp == null) {

System.out.println("No tasks found.");

return;

}

while (temp != null) {

System.out.println(temp.taskId + " " + temp.description + " Completed: " + temp.isDone);

temp = temp.next;

}

}

void completeTask(int id) {

Task temp = head;

while (temp != null) {

if (temp.taskId == id) {

temp.isDone = true;

System.out.println("Task " + id + " marked as completed.");

return;

}

temp = temp.next;

}

System.out.println("Task ID not found.");

}

void deleteTask(int id) {

if (head == null) {

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

return;

}

if (head.taskId == id) {

head = head.next;

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

return;

}

Task prev = null;

Task current = head;

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

prev = current;

current = current.next;

}

if (current == null) {

System.out.println("Task not found.");

} else {

prev.next = current.next;

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

}

}

Task searchTask(int id) {

Task temp = head;

while (temp != null) {

if (temp.taskId == id) return temp;

temp = temp.next;

}

return null;

}

}

public class Main {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

TaskList taskList = new TaskList();

int choice;

do {

System.out.println("\n--- Task Manager ---");

System.out.println("1. Add Task");

System.out.println("2. View All Tasks");

System.out.println("3. Complete Task");

System.out.println("4. Delete Task");

System.out.println("5. Search Task");

System.out.println("6. Exit");

System.out.print("Enter your choice: ");

choice = sc.nextInt();

sc.nextLine(); // consume leftover newline

switch (choice) {

case 1:

System.out.print("Enter Task ID: ");

int id = sc.nextInt();

sc.nextLine(); // consume newline

System.out.print("Enter Task Description: ");

String desc = sc.nextLine();

taskList.addTask(id, desc);

break;

case 2:

taskList.viewAllTasks();

break;

case 3:

System.out.print("Enter Task ID to mark as completed: ");

int compId = sc.nextInt();

taskList.completeTask(compId);

break;

case 4:

System.out.print("Enter Task ID to delete: ");

int delId = sc.nextInt();

taskList.deleteTask(delId);

break;

case 5:

System.out.print("Enter Task ID to search: ");

int searchId = sc.nextInt();

Task found = taskList.searchTask(searchId);

if (found != null)

System.out.println("Found: " + found.taskId + " " + found.description + " Completed: " + found.isDone);

else

System.out.println("Task not found.");

break;

case 6:

System.out.println("Exiting Task Manager...");

break;

default:

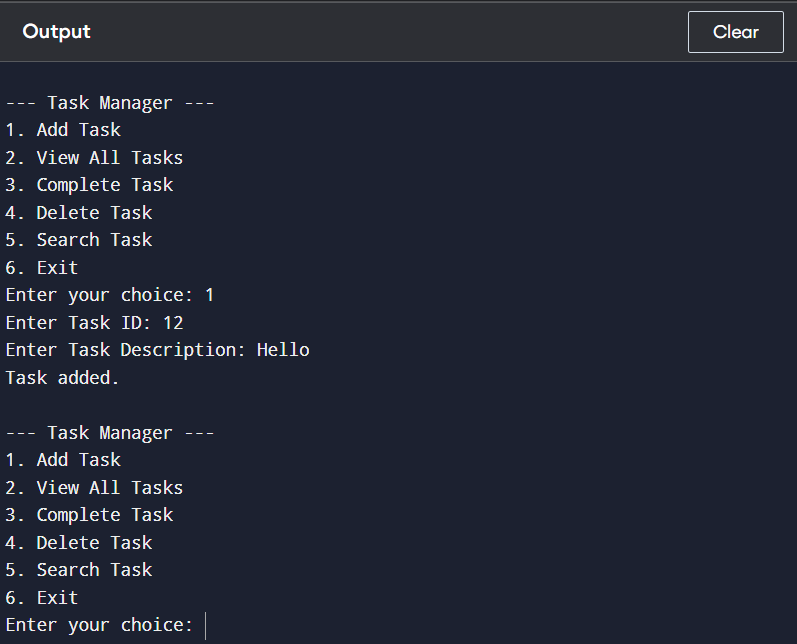
System.out.println("Invalid choice.");

}

} while (choice != 6);

sc.close();

}

}

**Exercise 6: Library Book 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.**

Introduction  
Managing books in a library efficiently is a key requirement.  
Using **HashMap** lets us quickly retrieve, add, or delete books by their unique IDs.

Why Are HashMap Important?

➥ Allows O(1) average time operations.  
➥ Enables fast search, addition, and deletion by key.

Types:

**HashMap:**  
➥ Stores key-value pairs.  
➥ Allows fast operations by bookId.

Summary:

➥ HashMap efficiently handles large libraries with fast operations.  
➥ Allows quick addition, deletion, search by IDs.

Full Code:

import java.util.\*;

class Book {

int bookId;

String title;

String author;

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

this.bookId = bookId;

this.title = title;

this.author = author;

}

}

public class Main {

static void addBook(List<Book> library, Book book) {

library.add(book);

}

static void deleteBook(List<Book> library, int id) {

library.removeIf(b -> b.bookId == id);

}

static void viewAllBooks(List<Book> library) {

for (Book b : library) {

System.out.println(b.bookId + " " + b.title + " by " + b.author);

}

}

static Book linearSearchByTitle(List<Book> library, String title) {

for (Book b : library) {

if (b.title.equalsIgnoreCase(title)) {

return b;

}

}

return null;

}

static Book binarySearchByTitle(List<Book> library, String title) {

Collections.sort(library, Comparator.comparing(b -> b.title.toLowerCase()));

int left = 0, right = library.size() - 1;

while (left <= right) {

int mid = (left + right) / 2;

String midTitle = library.get(mid).title.toLowerCase();

int cmp = midTitle.compareTo(title.toLowerCase());

if (cmp == 0) return library.get(mid);

else if (cmp < 0) left = mid + 1;

else right = mid - 1;

}

return null;

}

public static void main(String[] args) {

List<Book> library = new ArrayList<>();

addBook(library, new Book(1, "1984", "George Orwell"));

addBook(library, new Book(2, "To Kill a Mockingbird", "Harper Lee"));

addBook(library, new Book(3, "Brave New World", "Aldous Huxley"));

addBook(library, new Book(4, "Fahrenheit 451", "Ray Bradbury"));

System.out.println("All books:");

viewAllBooks(library);

System.out.println("\nLinear Search Result:");

Book foundLinear = linearSearchByTitle(library, "Brave New World");

if (foundLinear != null)

System.out.println("Found: " + foundLinear.title + " by " + foundLinear.author);

else

System.out.println("Book not found.");

System.out.println("\nBinary Search Result:");

Book foundBinary = binarySearchByTitle(library, "Brave New World");

if (foundBinary != null)

System.out.println("Found: " + foundBinary.title + " by " + foundBinary.author);

else

System.out.println("Book not found.");

deleteBook(library, 1);

System.out.println("\nAfter Deleting Book with ID 1:");

viewAllBooks(library);

}

}



**Exercise 7: Financial Forecast Application**

**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.**

Introduction  
A financial forecast application lets businesses track transactions and compute total profits or losses.  
Using **ArrayList** makes adding transactions and calculating totals convenient and flexible.

Why Are ArrayList Important?

➥ Allows adding transactions without predefined size.  
➥ Enables fast iteration for computing total.

Types:

**ArrayList:**  
➥ Stores transactions in a resizable array.  
➥ Allows adding transactions efficiently and calculating total afterwards.

Time Complexity Analysis:

| **Operations** | **Time** |
| --- | --- |
| add | O(1) |
| total | O(n) |
| view all | O(n) |

Possible Optimizations:

If frequently needing total, we can maintain a **running total**.

Summary:

➥ ArrayList is convenient for adding transactions and calculating total afterwards.  
➥ Operations scale well for small to medium sized financial data.

Full Code:

import java.util.\*;

class Transaction {

int transactionId;

double amount;

Transaction(int transactionId, double amount) {

this.transactionId = transactionId;

this.amount = amount;

}

}

class Main {

static void addTransaction(List<Transaction> transactions, Transaction transaction) {

transactions.add(transaction);

}

static double total(List<Transaction> transactions) {

double sum = 0;

for (Transaction t : transactions) {

sum += t.amount;

}

return sum;

}

static void viewAll(List<Transaction> transactions) {

for (Transaction t : transactions) {

System.out.println(t.transactionId + " " + t.amount);

}

}

static double forecast(double currentValue, double rate, int years) {

if (years == 0) return currentValue;

return forecast(currentValue \* (1 + rate), rate, years - 1);

}

public static void main(String[] args) {

List<Transaction> transactions = new ArrayList<>();

addTransaction(transactions, new Transaction(1, 500.0));

addTransaction(transactions, new Transaction(2, -200.0));

viewAll(transactions);

double current = total(transactions);

System.out.println("Total: " + current);

double growthRate = 0.1; // 10%

int years = 3;

double future = forecast(current, growthRate, years);

System.out.println("Forecasted Value after " + years + " years: " + future);

}

}

A screenshot of a computer

AI-generated content may be incorrect.