Exercise 1: Inventory Management System

**1. Understand the Problem**

* **Why Data Structures and Algorithms are Essential:**  
  In a warehouse, thousands of products are tracked. Efficient data structures ensure quick access, insertion, and deletion of product data, which improves the performance of inventory operations.
* **Suitable Data Structures:**
  + **ArrayList:** Easy to traverse but slower for search and update.
  + **HashMap:** Allows fast retrieval, update, and deletion using product ID as key.

**2. Setup and Implementation**

* Create a Java project named “Inventory Management System”.

**CODE**:

Class Definition (Example in Java):

import java.util.HashMap;

import java.util.Map;

class Product {

int productId;

String productName;

int quantity;

double price;

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

this.productId = productId;

this.productName = productName;

this.quantity = quantity;

this.price = price;

}

@Override

public String toString() {

return productId + " - " + productName + " - Qty: " + quantity + " - Price: ₹" + price;

}

}

public class Main {

static Map<Integer, Product> inventory = new HashMap<>();

static void addProduct(Product p) {

inventory.put(p.productId, p);

}

static void updateProduct(Product p) {

inventory.put(p.productId, p);

}

static void deleteProduct(int productId) {

inventory.remove(productId);

}

public static void main(String[] args) {

Product p1 = new Product(101, "Laptop", 5, 55000);

Product p2 = new Product(102, "Mouse", 10, 500);

addProduct(p1);

addProduct(p2);

System.out.println("Inventory after adding:");

inventory.values().forEach(System.out::println);

p2.quantity = 15;

updateProduct(p2);

System.out.println("\nInventory after update:");

inventory.values().forEach(System.out::println);

deleteProduct(101);

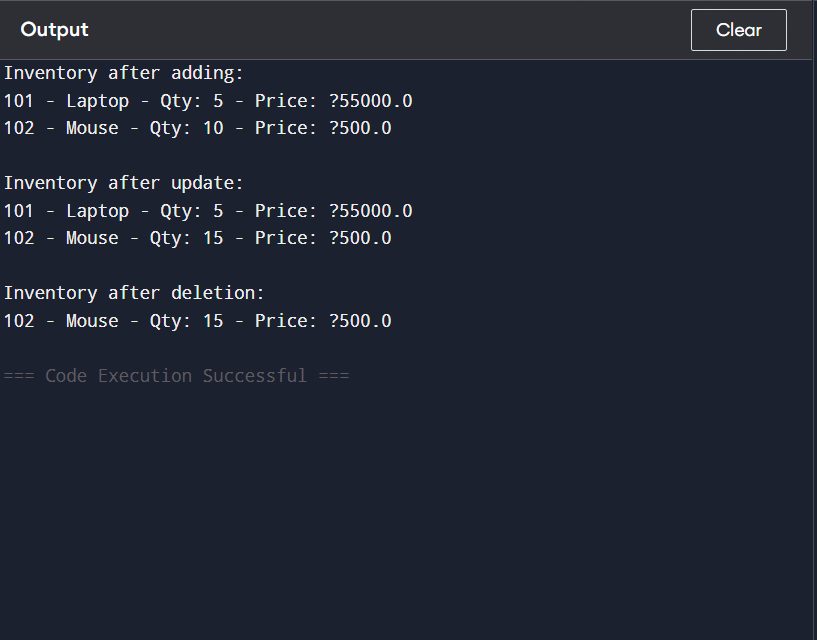
System.out.println("\nInventory after deletion:");

inventory.values().forEach(System.out::println);

}

}

**4. Analysis**

* **Add/Update/Delete (HashMap):**  
  Time Complexity = **O(1)** average, **O(n)** worst (due to hashing collisions).
* **Optimization:**
  + Use good hash functions.
  + Use Linked Hash Map if maintaining insertion order is needed. 

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

**1. Understand Asymptotic Notation**

* **Big O Notation:**  
  Describes how runtime grows with input size. Useful for comparing algorithm efficiency.
* **Best, Average, Worst Cases:**
  + **Linear Search:** Best O(1), Avg/Worst O(n)
  + **Binary Search:** Best O(1), Avg/Worst O(log n)

**2. Setup and Implementation**

**CODE:**

import java.util.Arrays;

class Product {

int productId;

String productName;

String category;

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

@Override

public String toString() {

return productId + " - " + productName + " - " + category;

}

}

public class Main {

static Product linearSearch(Product[] products, String name) {

for (Product p : products) {

if (p.productName.equals(name)) return p;

}

return null;

}

static Product binarySearch(Product[] products, String name) {

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

while (low <= high) {

int mid = (low + high) / 2;

int cmp = products[mid].productName.compareTo(name);

if (cmp == 0) return products[mid];

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

else high = mid - 1;

}

return null;

}

public static void main(String[] args) {

Product[] products = {

new Product(101, "Laptop", "Electronics"),

new Product(102, "Mouse", "Accessories"),

new Product(103, "Keyboard", "Accessories"),

new Product(104, "Monitor", "Electronics")

};

// Sort the array by productName before binary search

Arrays.sort(products, (a, b) -> a.productName.compareTo(b.productName));

// Linear Search

Product foundLinear = linearSearch(products, "Mouse");

System.out.println("Linear Search Result: " + (foundLinear != null ? foundLinear : "Not Found"));

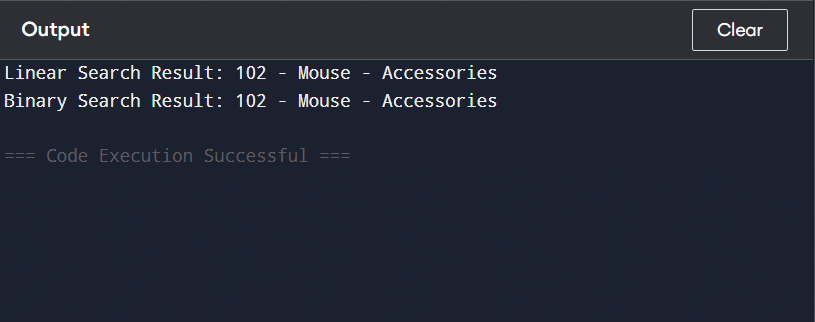
// Binary Search

Product foundBinary = binarySearch(products, "Mouse");

System.out.println("Binary Search Result: " + (foundBinary != null ? foundBinary : "Not Found"));

}

}**4. Analysis**

* **Linear Search:** O(n)
* **Binary Search:** O(log n) – requires sorted data.
* **Which to use:**
  + Use binary search for large, sorted datasets.
  + Use linear search if data is small or unsorted. 

**Exercise 3: Sorting Customer Orders**

**1. Understand Sorting Algorithms**

* **Bubble Sort:** Compares adjacent elements; swaps if needed. Inefficient.
* **Insertion Sort:** Good for small/partially sorted data.
* **Quick Sort:** Efficient divide-and-conquer sort.
* **Merge Sort:** Stable, consistent O(n log n) performance.

**2. Setup and Implementation**

**CODE:**

class Order {

int orderId;

String customerName;

double totalPrice;

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

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

@Override

public String toString() {

return orderId + " - " + customerName + " - ₹" + totalPrice;

}

}

public class Main {

static void bubbleSort(Order[] orders) {

for (int i = 0; i < orders.length - 1; i++) {

for (int j = 0; j < orders.length - i - 1; 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;

}

public static void main(String[] args) {

Order[] orders = {

new Order(201, "Alice", 499.99),

new Order(202, "Bob", 150.00),

new Order(203, "Charlie", 299.99),

new Order(204, "David", 700.00)

};

System.out.println("Original Orders:");

for (Order o : orders) System.out.println(o);

// Bubble Sort

bubbleSort(orders);

System.out.println("\nAfter Bubble Sort by totalPrice:");

for (Order o : orders) System.out.println(o);

// Shuffle for testing quick sort

orders = new Order[] {

new Order(201, "Alice", 499.99),

new Order(202, "Bob", 150.00),

new Order(203, "Charlie", 299.99),

new Order(204, "David", 700.00)

};

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

System.out.println("\nAfter Quick Sort by totalPrice:");

for (Order o : orders) System.out.println(o);

}

}

**4. Analysis**

* **Bubble Sort:** O(n²) – not efficient for large datasets.
* **Quick Sort:** Avg O(n log n), Worst O(n²) – better for large data.

**Why Prefer Quick Sort:**  
Faster in most real-world cases and more space-efficient than merge sort. 

**Exercise 4: Employee Management System**

**1. Understand Array Representation**

* **In Memory:** Arrays are stored in contiguous memory blocks.
* **Advantages:** Fast access using indices, predictable performance.

**2. Setup and Implementation**

**CODE:**

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;

}

@Override

public String toString() {

return employeeId + " - " + name + " - " + position + " - ₹" + salary;

}

}

public class Main {

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

static int size = 0;

static void add(Employee e) {

employees[size++] = e;

}

static Employee search(int id) {

for (int i = 0; i < size; i++) {

if (employees[i].employeeId == id) return employees[i];

}

return null;

}

static void traverse() {

for (int i = 0; i < size; i++) {

System.out.println(employees[i]);

}

}

static void delete(int id) {

for (int i = 0; i < size; i++) {

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

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

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

}

size--;

break;

}

}

}

public static void main(String[] args) {

// Adding employees

add(new Employee(1, "Alice", "Manager", 70000));

add(new Employee(2, "Bob", "Developer", 50000));

add(new Employee(3, "Charlie", "Designer", 45000));

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

traverse();

// Search

System.out.println("\nSearching for employee with ID 2:");

Employee found = search(2);

System.out.println(found != null ? found : "Not Found");

// Delete

delete(2);

System.out.println("\nEmployees after deleting ID 2:");

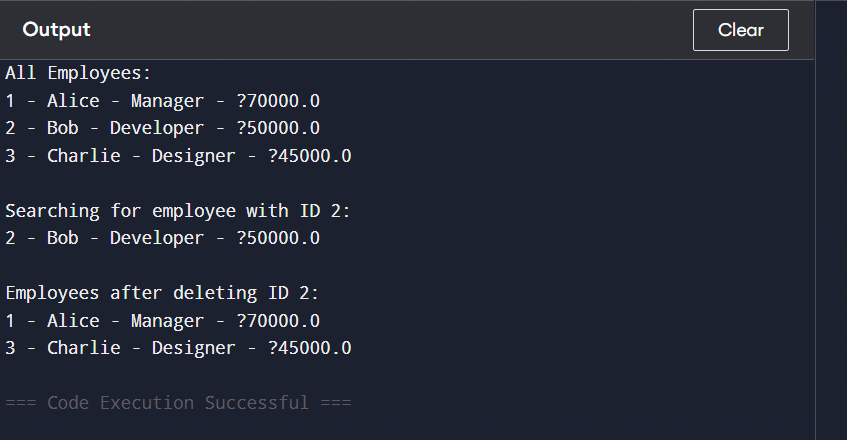
traverse();

}

}

**4. Analysis**

* Add: O(1)
* Search: O(n)
* Traverse: O(n)
* Delete: O(n)

**Limitations:**  
Fixed size, inefficient delete, not dynamic. Use ArrayList or LinkedList for better flexibility. 

**Exercise 5: Task Management System**

**1. Understand Linked Lists**

* **Singly Linked List:** Each node points to the next.
* **Doubly Linked List:** Each node has both next and previous pointers.

**2. Setup & Implementation**

**CODE:**

class Task {

int taskId;

String taskName;

String status;

Task next;

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

this.taskId = taskId;

this.taskName = taskName;

this.status = status;

this.next = null;

}

@Override

public String toString() {

return taskId + " - " + taskName + " - " + status;

}

}

public class Main {

static Task head = null;

static void add(Task t) {

t.next = head;

head = t;

}

static Task search(int id) {

Task temp = head;

while (temp != null) {

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

temp = temp.next;

}

return null;

}

static void traverse() {

Task temp = head;

while (temp != null) {

System.out.println(temp);

temp = temp.next;

}

}

static void delete(int id) {

Task temp = head, prev = null;

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

prev = temp;

temp = temp.next;

}

if (temp != null) {

if (prev == null) head = temp.next;

else prev.next = temp.next;

}

}

public static void main(String[] args) {

// Adding tasks

add(new Task(1, "Design", "Pending"));

add(new Task(2, "Development", "In Progress"));

add(new Task(3, "Testing", "Not Started"));

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

traverse();

// Search

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

Task found = search(2);

System.out.println(found != null ? found : "Task Not Found");

// Delete

delete(2);

System.out.println("\nTasks after deleting Task ID 2:");

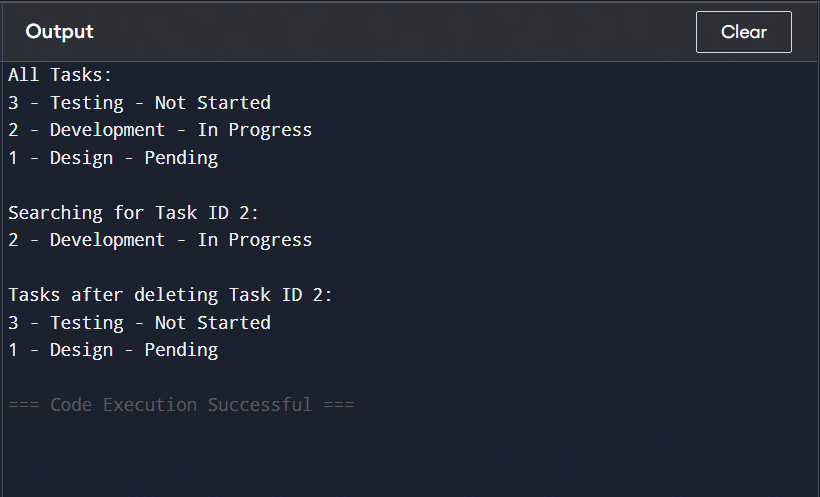
traverse();

}

}

**4. Analysis**

* Add: O(1)
* Search: O(n)
* Traverse: O(n)
* Delete: O(n)

**Advantages Over Arrays:**  
Dynamic size, easier insertions/deletions. 

**Exercise 6: Library Management System**

**1. Understand Search Algorithms**

* **Linear Search:** No sorting needed, checks each element.
* **Binary Search:** Requires sorted data; more efficient.

**2. Setup & Implementation**

CODE:

import java.util.Arrays;

class Book {

int bookId;

String title;

String author;

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

this.bookId = bookId;

this.title = title;

this.author = author;

}

@Override

public String toString() {

return bookId + " - " + title + " by " + author;

}

}

public class Main {

static Book linearSearch(Book[] books, String title) {

for (Book b : books) {

if (b.title.equals(title)) return b;

}

return null;

}

static Book binarySearch(Book[] books, String title) {

int low = 0, high = books.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

int cmp = books[mid].title.compareTo(title);

if (cmp == 0) return books[mid];

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

else high = mid - 1;

}

return null;

}

public static void main(String[] args) {

Book[] books = {

new Book(1, "C Programming", "Dennis Ritchie"),

new Book(2, "Java Basics", "James Gosling"),

new Book(3, "Python Crash Course", "Eric Matthes"),

new Book(4, "Algorithms", "Robert Sedgewick")

};

// Sort the array before using binary search

Arrays.sort(books, (a, b) -> a.title.compareTo(b.title));

System.out.println("Linear Search for 'Java Basics':");

Book foundLinear = linearSearch(books, "Java Basics");

System.out.println(foundLinear != null ? foundLinear : "Book Not Found");

System.out.println("\nBinary Search for 'Java Basics':");

Book foundBinary = binarySearch(books, "Java Basics");

System.out.println(foundBinary != null ? foundBinary : "Book Not Found");

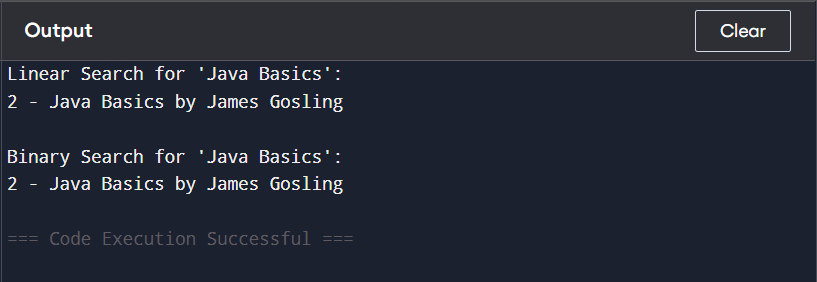
}

}

**4. Analysis**

* Linear: O(n)
* Binary: O(log n)

**Use Binary Search** when data is sorted and large.



**Exercise 7: Financial Forecasting**

**1. Understand Recursive Algorithms**

* **Recursion:** Function calls itself with a base condition.
* **Use:** Simplifies problems like Fibonacci series, tree traversal, financial growth prediction.

**2. Setup and Implementation**

**CODE:**

public class Main {

static double predictValue(int year, double current, double rate) {

if (year == 0) return current;

return predictValue(year - 1, current, rate) \* (1 + rate);

}

static double futureValue(int years, double value, double growthRate) {

if (years == 0) return value;

return futureValue(years - 1, value, growthRate) \* (1 + growthRate);

}

public static void main(String[] args) {

double current = 1000;

double rate = 0.08;

int year = 5;

double predicted = predictValue(year, current, rate);

double future = futureValue(year, current, rate);

System.out.println("Predicted Value after " + year + " years: ₹" + predicted);

System.out.println("Future Value after " + year + " years: ₹" + future);

}

}**4. Analysis**

* Time Complexity: O(n)
* **Optimization:** Use **memoization** or **iterative method** to reduce stack calls.
* 