**Data structures and Algorithms – WEEK 1 HANDS ON(MANDATORY HANDS ON INCLUDED)**

**Exercise 1: Inventory Management System**

**Scenario:**

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

**UNDERSTANDING THE PROBLEM**:

**Suitable Data Structures**

* **ArrayList**: Easy to use, but searching takes O(n) time.
* **HashMap**: Ideal for key-value pairs like productId → Product, supports:
* O(1) average-case insertion, deletion, and search.
* Very efficient for managing large inventories.
* **We will use HashMap<Integer, Product>.**

**CODE:**

import java.util.HashMap;

public class InventorySystem {

static class Product {

private int productId;

private String productName;

private int quantity;

private double price;

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

this.productId = productId;

this.productName = productName;

this.quantity = quantity;

this.price = price;

}

public int getProductId() {

return productId;

}

public String getProductName() {

return productName;

}

public int getQuantity() {

return quantity;

}

public double getPrice() { return price; }

public void setQuantity(int quantity) {

this.quantity = quantity;

}

public void setPrice(double price) {

this.price = price;

}

public String toString() {

return "ID: " + productId + ", Name: " + productName + ", Qty: " + quantity + ", Price: ₹" + price;

}

}

static class InventoryManager {

private HashMap<Integer, Product> inventor

public InventoryManager() {

inventory = new HashMap<>();

}

public void addProduct(Product product) {

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

}

public void updateProduct(int productId, int newQuantity, double newPrice) {

if (inventory.containsKey(productId)) {

Product product = inventory.get(productId);

product.setQuantity(newQuantity);

product.setPrice(newPrice);

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

} else {

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

}

}

public void deleteProduct(int productId) {

if (inventory.containsKey(productId)) {

inventory.remove(productId);

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

} else {

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

}

}

public void displayInventory() {

if (inventory.isEmpty()) {

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

} else {

for (Product p : inventory.values()) {

System.out.println(p);

}

}

}

}

public static void main(String[] args) {

InventoryManager manager = new InventoryManager()

manager.addProduct(new Product(101, "Laptop", 10, 75000));

manager.addProduct(new Product(102, "Keyboard", 25, 700));

manager.addProduct(new Product(103, "Mouse", 50, 450));

System.out.println("\n-- Inventory After Adding --");

manager.displayInventory();

manager.updateProduct(102, 30, 800);

System.out.println("\n-- Inventory After Update --");

manager.displayInventory();

manager.deleteProduct(101);

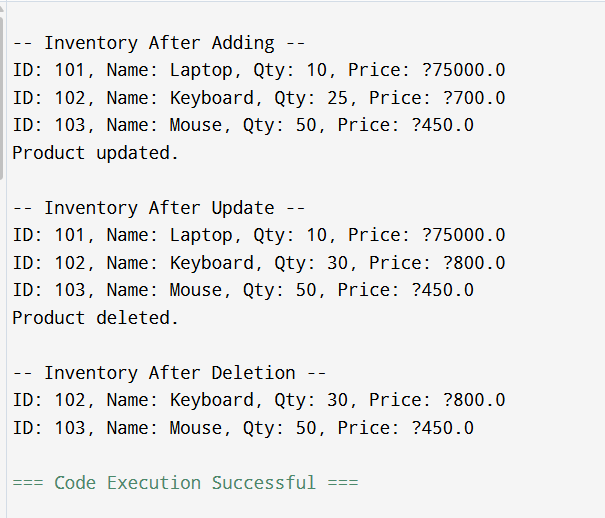
System.out.println("\n-- Inventory After Deletion --");

manager.displayInventory();

}

}

**OUTPUT:**



**Exercise 2: E-commerce Platform Search Function (MANDATORY HANDS ON)**

**Scenario:**

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

**UNDERSTANDING THE PROBLEM:**

Users expect instant results.

Inventory can contain thousands of product names.

Linear search becomes slow at scale.

**CODE:**

import java.util.\*;

public class ECommerceSearch {

static class TrieNode {

Map<Character, TrieNode> children = new HashMap<>();

boolean isEndOfWord = false;

List<String> suggestions = new ArrayList<>();

}

static class ProductTrie {

TrieNode root;

public ProductTrie() {

root = new TrieNode();

}

public void insert(String productName) {

TrieNode node = root;

productName = productName.toLowerCase();

for (char c : productName.toCharArray()) {

node.children.putIfAbsent(c, new TrieNode());

node = node.children.get(c);

if (!node.suggestions.contains(productName)) {

node.suggestions.add(productName);

}

}

node.isEndOfWord = true;

}

public List<String> search(String prefix) {

TrieNode node = root;

prefix = prefix.toLowerCase();

for (char c : prefix.toCharArray()) {

if (!node.children.containsKey(c)) {

return new ArrayList<>();

}

node = node.children.get(c);

}

return node.suggestions;

}

}

public static void main(String[] args) {

ProductTrie searchSystem = new ProductTrie();

searchSystem.insert("Laptop");

searchSystem.insert("Lamp");

searchSystem.insert("Lantern");

searchSystem.insert("Mobile");

searchSystem.insert("Monitor");

searchSystem.insert("Mouse");

searchSystem.insert("Microwave");

Scanner sc = new Scanner(System.in);

System.out.println("Enter search prefix: ");

String input = sc.nextLine();

List<String> results = searchSystem.search(input);

if (results.isEmpty()) {

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

} else {

System.out.println("Search Results:");

for (String product : results) {

System.out.println("- " + product);

}

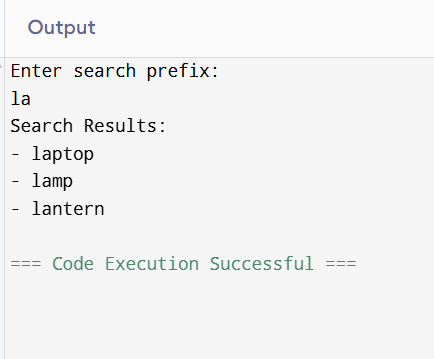
}

sc.close();

}

}

**OUTPUT:**



**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.

**UNDERSTANDING THE PROBLEM:**

Bubble Sort

* Repeatedly compares adjacent elements and swaps them if they are in the wrong order.
* Simple but slow for large datasets.

Time Complexity:

* Best: O(n) (already sorted)
* Average/Worst: O(n²)

**CODE:**

import java.util.\*;

public class OrderSorter {

static class Order {

int orderId;

String customerName;

double totalPrice;

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

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

@Override

public String toString() {

return "OrderID: " + orderId + ", Name: " + customerName + ", Total: ₹" + totalPrice;

}

}

static void bubbleSort(Order[] orders) {

int n = orders.length;

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

boolean swapped = false;

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

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

Order temp = orders[j];

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

orders[j + 1] = temp;

swapped = true;

}

}

if (!swapped) break;

}

}

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 display(Order[] orders) {

for (Order o : orders) {

System.out.println(o);

}

}

public static void main(String[] args) {

Order[] orders = {

new Order(101, "Alice", 5000),

new Order(102, "Bob", 1200),

new Order(103, "Charlie", 8500),

new Order(104, "David", 3000),

new Order(105, "Eve", 6500)

};

System.out.println("Sorted by Bubble Sort:");

Order[] bubbleSorted = orders.clone();

bubbleSort(bubbleSorted);

display(bubbleSorted);

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

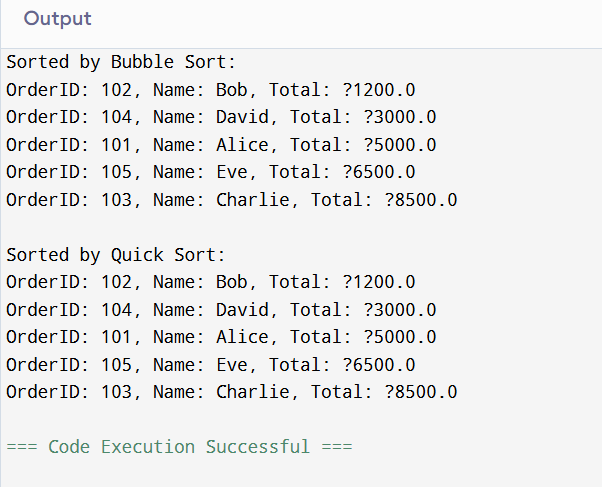
Order[] quickSorted = orders.clone();

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

display(quickSorted);

}

}



**Exercise 4: Employee Management System**

**Scenario:**

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

**UNDERSTANDING THE PROBLEM:**

Arrays:

* An array is like a row of boxes placed in a line — each box stores a value (like an employee record).
* Every box (element) has a fixed position (index), starting from 0.

Advantages of Arrays:

* Fast access: You can access any employee directly by their position — O(1) time.
* Simple structure: Easy to understand and implement.

Limitations:

* Fixed size: Once an array is created, you can’t change its size.
* Manual shifting: When you delete or insert, you may need to move other elements manually.

**CODE:**

import java.util.Scanner;

public class EmployeeManagement {

static class Employee {

int employee;

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 "ID: " + employeeId + ", Name: " + name + ", Position: " + position + ", Salary: ₹" + salary;

}

}

static final int MAX\_EMPLOYEES = 100;

static Employee[] employees = new Employee[MAX\_EMPLOYEES];

static int count = 0;

static void addEmployee(Employee emp) {

if (count < MAX\_EMPLOYEES) {

employees[count] = emp;

count++;

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

} else {

System.out.println("Employee array is full.");

}

}

static void searchEmployee(int id) {

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

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

System.out.println("Employee found: " + employees[i]);

return;

}

}

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

}

static void displayEmployees() {

if (count == 0) {

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

} else {

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

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

}

}

}

static void deleteEmployee(int id) {

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

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

// Shift elements to left

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

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

}

employees[count - 1] = null;

count--;

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

return;

}

}

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

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

addEmployee(new Employee(101, "Alice", "Developer", 50000));

addEmployee(new Employee(102, "Bob", "Designer", 45000));

addEmployee(new Employee(103, "Charlie", "Manager", 60000));

System.out.println("\n-- All Employees --");

displayEmployees();

System.out.println("\n-- Search Employee with ID 102 --");

searchEmployee(102);

System.out.println("\n-- Delete Employee with ID 101 --");

deleteEmployee(101);

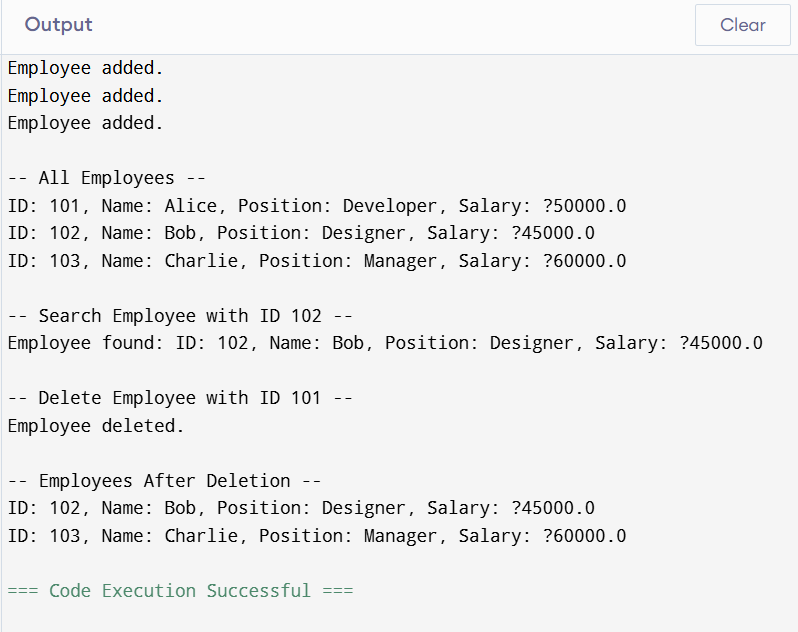
System.out.println("\n-- Employees After Deletion --");

displayEmployees(); sc.close();

}

}

**OUTPUT :**

****

**Exercise 5: Task Management System**

Scenario:

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

**UNDERSTANDING THE PROBLEM:**

Linked List is a linear data structure where each element (node) contains:

* Some data (e.g., taskId, taskName, etc.)
* A reference (link) to the next node in the sequence.

There are mainly two types of linked lists:

1. Singly Linked List:

Each node points to only the next node.

Efficient for forward traversal.

Simpler and uses less memory.

1. Doubly Linked List:

Each node has a link to the next and previous node.

Allows both forward and backward traversal.

**CODE:**

import java.util.LinkedList;

import java.util.Scanner;

public class TaskManager {

static class Task {

int taskId;

String description;

Task(int taskId, String description) {

this.taskId = taskId;

this.description = description;

}

@Override

public String toString() {

return "Task ID: " + taskId + ", Description: " + description;

}

}

static LinkedList<Task> taskList = new LinkedList<>();

static void addTask(int id, String desc) {

Task newTask = new Task(id, desc);

taskList.add(newTask);

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

}

static void deleteTask(int id) {

boolean found = false;

for (Task task : taskList) {

if (task.taskId == id) {

taskList.remove(task);

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

found = true;

break;

}

}

if (!found) {

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

}

}

static void displayTasks() {

if (taskList.isEmpty()) {

System.out.println("No tasks to show.");

} else {

for (Task task : taskList) {

System.out.println(task);

}

}

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

addTask(1, "Complete Java assignment");

addTask(2, "Study for math test");

addTask(3, "Buy groceries");

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

displayTasks();

deleteTask(2);

System.out.println("\nAfter Deletion:");

displayTasks();

sc.close();

}

}



**Exercise 6: Library Management System**

**Scenario:**

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

**UNDERSTANDING THE PROBLEM:**

**Linear Search**

* Checks each item one by one.
* Works on unsorted lists.
* Time complexity: O(n) — slower as list grows.

**Binary Search**

* Only works on sorted lists.
* Divides the list in half each time (like playing "Guess the Number").

Time complexity: O(log n) — much faster for large data**.**

**CODE:**

import java.util.\*;

public class LibrarySystem {

static 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 String toString() {

return "ID: " + bookId + ", Title: " + title + ", Author: " + author;

}

}

static void linearSearch(List<Book> books, String searchTitle) {

boolean found = false;

for (Book book : books) {

if (book.title.equalsIgnoreCase(searchTitle)) {

System.out.println("Book found (Linear Search): " + book);

found = true;

}

}

if (!found) System.out.println("Book not found (Linear Search).");

}

static void binarySearch(List<Book> books, String searchTitle) {

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

int low = 0, high = books.size() - 1;

boolean found = false;

while (low <= high) {

int mid = (low + high) / 2;

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

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

if (cmp == 0) {

System.out.println("Book found (Binary Search): " + books.get(mid));

found = true;

break;

} else if (cmp < 0) {

high = mid - 1;

} else {

low = mid + 1;

}

}

if (!found) System.out.println("Book not found (Binary Search).");

}

public static void main(String[] args) {

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

library.add(new Book(1, "Java Programming", "James Gosling"));

library.add(new Book(2, "Data Structures", "Robert Lafore"));

library.add(new Book(3, "Clean Code", "Robert C. Martin"));

library.add(new Book(4, "Artificial Intelligence", "Stuart Russell"));

linearSearch(library, "Clean Code");

binarySearch(library, "Clean Code");

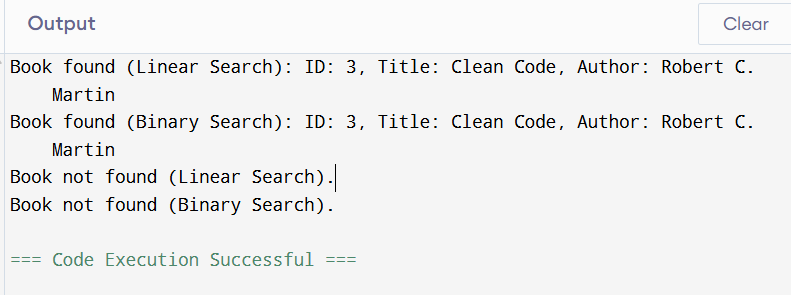
linearSearch(library, "Python Programming");

binarySearch(library, "Python Programming");

}

}

**OUTPUT:**



**Time Complexity Comparison:**

Linear Search : O(n)

Binary Search : O(logn)

**Exercise 7: Financial Forecasting (MANDATORY HANDS ON)**

**Scenario:**

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

**UNDERSTANDING THE PROBLEM:**

Recursion is when a function calls itself to solve a smaller part of the same problem.

It helps break down a big task into smaller repeatable steps — which is perfect for forecasting one year at a time.

**CODE:**

public class FinancialForecast {

static double predictFutureValue(double initialAmount, double growthRate, int years) {

if (years == 0) return initialAmount;

return predictFutureValue(initialAmount, growthRate, years - 1) \* (1 + growthRate);

}

public static void main(String[] args) {

double initial = 10000; // starting amount

double rate = 0.1; // 10% annual growth

int years = 5;

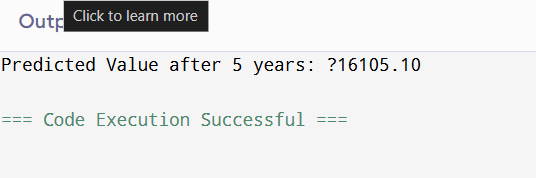
double futureValue = predictFutureValue(initial, rate, years);

System.out.printf("Predicted Value after %d years: ₹%.2f\n", years, futureValue);

}

}

**OUTPUT:**

****