**Cognizant - DN 4.0 Deep Skilling Week-1 Assignment solutions**

ALGORITHMS AND DATA STRUCTURES

Exercise 1: Inventory Management System Scenario:-

**1. Why Data Structures & Algorithms Are Essential:-**

Handling large inventories requires:

* Efficient retrieval/search.
* Fast update**s**
* Scalability for thousands of records.
* Efficient data structures prevent slowdowns:
* Linear searches become too slow as size grows.
* Hash-based structures allow constant time access.

**2. Suitable Data Structures:-** Array List,Treemap,Hashmap

**Best choice here:** HashMap<Integer, Product> using productId as the key.

**3. Java Implementation:-**

import java.util.HashMap;

import java.util.Map;

class Product {

int productId;

String productName;

int quantity;

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 String toString() {

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

}

}

class Inventory {

private Map<Integer, Product> products = new HashMap<>();

// Add a product

public void addProduct(Product product) {

products.put(product.productId, product);

System.out.println("Product added: " + product);

}

public void updateProduct(int productId, int quantity, double price) {

Product product = products.get(productId);

if (product != null) {

product.quantity = quantity;

product.price = price;

System.out.println("Product updated: " + product);

} else {

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

}

}

public void deleteProduct(int productId) {

if (products.containsKey(productId)) {

Product removed = products.remove(productId);

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

} else {

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

}

}

public void listProducts() {

if (products.isEmpty()) {

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

} else {

for (Product product : products.values()) {

System.out.println(product);

}

}

}

}

public class InventorySystem {

public static void main(String[] args) {

Inventory inventory = new Inventory();

inventory.addProduct(new Product(101, "Mouse", 50, 499.99));

inventory.addProduct(new Product(102, "Keyboard", 20, 899.50));

// Update a product

inventory.updateProduct(101, 40, 479.99);

// Delete a product

inventory.deleteProduct(102);

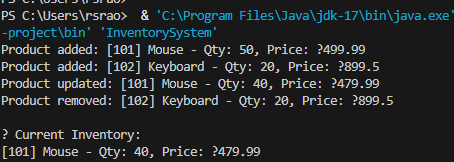
// List all products

inventory.listProducts();

}

}

OUTPUT:-



**4. Time Complexity Analysis:-**

Add:- O(1)avg

Update:- O(1)avg

Delete:-O(1)avg

List:-O(n)

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

1. Understand Asymptotic Notation:-

### **Big O Notation:-** Big O describes how the time or space an algorithm takes grows relative to input size.

* O(1):- Constant Time – Super fast, independent of input size
* O(n):- Linear Time – Time grows with input size
* O(log n):-Logarithmic Time – Fast for large input (binary search)
* O(n²):- Quadratic Time – Slow, especially for large inputs

### **Search Scenarios**

* **Best Case:**
* Linear: First item match → O(1)
* Binary: Middle match → O(1)
* **Average Case**:
* Linear: Random match → O(n/2) ≈ O(n)
* Binary: Halving → O(log n)
* **Worst Case**:
* Linear: Last/no match → O(n)
* Binary: Last comparison → O(log n)

**2. Java Implementation(Product Class for Search):-**

class Product {

String productId;

String productName;

String category;

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

@Override

public String toString() {

return productName + " (ID: " + productId + ", Category: " + category + ")";

}

}

**3. Implementation: Linear & Binary Search:-**

import java.util.Arrays;

import java.util.Comparator;

public class EcommerceSearch {

// Define the Product class

static class Product {

String productId;

String productName;

String category;

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

@Override

public String toString() {

return productName + " (ID: " + productId + ", Category: " + category + ")";

}

}

// Linear Search

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

for (Product product : products) {

if (product.productName.equalsIgnoreCase(name)) {

return product;

}

}

return null;

}

// Binary Search (Requires sorted array)

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

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

while (low <= high) {

int mid = (low + high) / 2;

int compare = products[mid].productName.compareToIgnoreCase(name);

if (compare == 0)

return products[mid];

else if (compare < 0)

low = mid + 1;

else

high = mid - 1;

}

return null;

}

public static void main(String[] args) {

Product[] productList = {

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

new Product("102", "Shampoo", "Personal Care"),

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

new Product("104", "Notebook", "Stationery"),

new Product("105", "Charger", "Electronics")

};

// Linear Search

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

Product foundLinear = linearSearch(productList, "Keyboard");

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

// Binary Search requires sorted array

Arrays.sort(productList, Comparator.comparing(p -> p.productName));

System.out.println("\n Binary Search for 'Keyboard':");

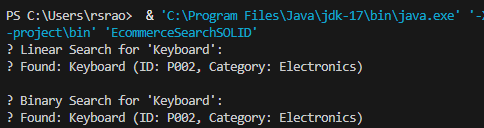
Product foundBinary = binarySearch(productList, "Keyboard");

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

}

}

OUTPUT:-



### **Which is Better?**

* **Small datasets** or **unsorted data**: use **Linear Search**
* **Large datasets** with **sorted data**: use **Binary Search**

For an **e-commerce platform**, **binary search is preferred** for its efficiency with sorted product names or IDs.

Exercise 3: Sorting Customer Orders Scenario:

Step 1: Understand Sorting Algorithms:-

### **Bubble Sort**

* Repeatedly swaps adjacent elements if they're in the wrong order.
* **Time Complexity**:
  + Worst & Avg:O(n²)
  + Best:(n) (if already sorted with optimization)
* **Simple but slow** for large data.

**Quick Sort**

* Divide-and-conquer using a **pivot**.
* Partitions array into two halves and recursively sorts them.
* **Time Complexity**:
  + Best & Avg: O(n log n)
  + Worst: O(n²) (rare, with bad pivots)
* **Much faster** in real-world scenarios.

## **Step 2: Create the Order class**

You’ll define:

* orderId (String)
* customerName (String)
* totalPrice (double)

Step 3: Full Java Code (Bubble Sort + Quick Sort):-

import java.util.\*;

// Order class

class Order {

String orderId;

String customerName;

double totalPrice;

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

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

public String toString() {

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

}

}

public class OrderSorting {

// Bubble Sort

static void bubbleSort(Order[] orders) {

int n = orders.length;

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

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;

}

}

}

}

// Quick Sort

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;

}

// Display orders

static void printOrders(String message, Order[] orders) {

System.out.println(message);

for (Order o : orders) {

System.out.println(o);

}

System.out.println();

}

public static void main(String[] args) {

Order[] orders1 = {

new Order("O001", "Spandana", 4500.0),

new Order("O002", "Riya", 2999.5),

new Order("O003", "Aarav", 6800.0),

new Order("O004", "Ravi", 1200.0)

};

Order[] orders2 = Arrays.copyOf(orders1, orders1.length); // for Quick Sort

// Bubble Sort

bubbleSort(orders1);

printOrders(" Orders sorted by Bubble Sort (Low -> High):", orders1);

// Quick Sort

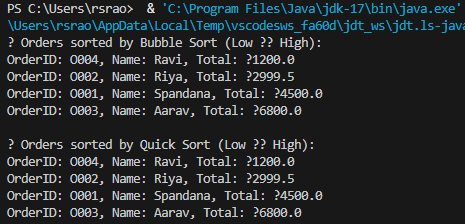
quickSort(orders2, 0, orders2.length - 1);

printOrders("Orders sorted by Quick Sort (Low -> High):", orders2);

}

}

OUTPUT:-



**Step 4: Time Complexity Analysis:-**

### **Why is Quicksort Better?**

* It divides the problem fast.
* Performs fewer comparisons and swaps.
* Faster for **large datasets**.
* Widely used in real-world platforms like Amazon, Flipkart, etc.

Exercise 4: Employee Management System Scenario:

**Step 1: Array Representation in Memory:-**

### **How Arrays Work:**

* Arrays are contiguous memory blocks.
* Each element is placed one after another.
* Accessing by index is super fast (O(1)).

### **Pros:**

* Fast access
* Easy to traverse
* Cache-friendly

### **Limitations:**

* Fixed size
* Costly to insert/delete in between (O(n))

**Step 2 & 3: Full Java Code (Add, Search, Traverse, Delete):-**

import java.util.Scanner;

class Employee {

int employeeId;

String name;

String position;

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 String toString() {

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

}

}

public class EmployeeManagement {

static final int MAX = 100; // Max employees

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

static int count = 0;

// Add employee

static void addEmployee(Employee emp) {

if (count < MAX) {

employees[count++] = emp;

System.out.println("Employee added: " + emp.name);

} else {

System.out.println(" Employee list full!");

}

}

// Search by ID

static void searchEmployee(int id) {

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

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

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

return;

}

}

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

}

// Traverse

static void listEmployees() {

if (count == 0) {

System.out.println(" No employees in the system.");

return;

}

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

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

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

}

}

// Delete by ID

static void deleteEmployee(int id) {

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

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

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

employees[j] = employees[j + 1]; // Shift left

}

employees[--count] = null; // Remove last

System.out.println(" Deleted employee with ID: " + id);

return;

}

}

System.out.println(" No employee with ID: " + id);

}

public static void main(String[] args) {

addEmployee(new Employee(101, "Spandana", "Manager", 50000));

addEmployee(new Employee(102, "Riya", "Developer", 42000));

addEmployee(new Employee(103, "Arjun", "Tester", 30000));

listEmployees();

searchEmployee(102);

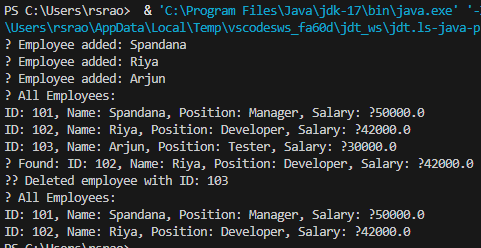
deleteEmployee(103);

listEmployees();

}

}

OUTPUT:-



**Step 4: Time Complexity Analysis:-**

Add:- O(1):- Direct insertion at count index.

Search:-O(n):-Linear search over array.

Traverse:-O(n):-Print each element.

Delete:-O(n):-Shift elements after deletion

* **Limitations of Arrays:-**

Fixed size:-Use ArrayList, LinkedList in real apps

Deletion is costly (O(n)):-Use HashMap for fast access

Insertion in middle is slow:- Use dynamic data structures (LinkedList)

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

Exercise 5: Task Management System Scenario:-

## **Step 1: Understanding Linked Lists:-**

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

Singly Linked List:-Each node links to the next

Doubly Linked List:-Each node links to both next & previous

**We’re using Singly Linked List** for lighter & perfect for simple dynamic data

**Step 2 & 3: Full Java Code:-**

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 String toString() {

return "ID: " + taskId + ", Name: " + taskName + ", Status: " + status;

}

}

class TaskManager {

Task head = null;

// Add Task at end

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: " + taskName);

}

// Search Task by ID

public void searchTask(int taskId) {

Task temp = head;

while (temp != null) {

if (temp.taskId == taskId) {

System.out.println(" Found Task: " + temp);

return;

}

temp = temp.next;

}

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

}

// Traverse all tasks

public void listTasks() {

if (head == null) {

System.out.println(" No tasks in the list.");

return;

}

System.out.println("Task List:");

Task temp = head;

while (temp != null) {

System.out.println(temp);

temp = temp.next;

}

}

// Delete Task by ID

public void deleteTask(int taskId) {

if (head == null) {

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

return;

}

if (head.taskId == taskId) {

System.out.println("Deleted Task: " + head.taskName);

head = head.next;

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 not found with ID: " + taskId);

return;

}

previous.next = current.next;

System.out.println(" Deleted Task: " + current.taskName);

}

}

public class TaskManagement {

public static void main(String[] args) {

TaskManager manager = new TaskManager();

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

manager.addTask(2, "Implement backend", "In Progress");

manager.addTask(3, "Testing", "Pending");

manager.listTasks();

manager.searchTask(2);

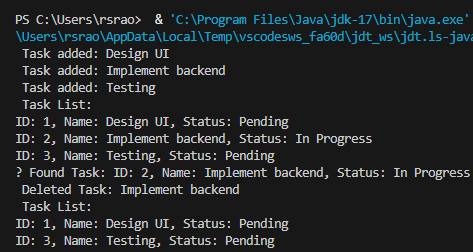
manager.deleteTask(2);

manager.listTasks();

}

}

OUTPUT:-



**Linked List vs Array (When to Use?):-**

Arrays have a **fixed size** and offer **fast access** through indexing (O(1)), but **adding or deleting elements** can be costly due to shifting. They are also **memory-efficient**. On the other hand, linked lists are **dynamic in size**, making it easier to **add or delete elements** by simply relinking nodes, though **searching is slower (O(n))** and they use **more memory** due to additional pointer storage.

Exercise 6: Library Management System Scenario:

**Step 1: Understanding Search Algorithms:**-

### **Linear Search**

* **Checks each element one by one**
* Works on **unsorted** data
* **Time Complexity:**
  + Best: O(1)
  + Average/Worst: O(n)

**Binary Search**

* Requires **sorted** data
* Divides the array in half each time
* **Time Complexity:**
* Best: O(1)
* Average/Worst: O(log n)

**Step 2 & 3: Full Java Code:-**

import java.util.Arrays;

import java.util.Comparator;

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 String toString() {

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

}

}

public class LibraryManagement {

// Linear Search

public static void linearSearch(Book[] books, String targetTitle) {

System.out.println(" Linear Search for '" + targetTitle + "':");

for (Book book : books) {

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

System.out.println(" Found: " + book);

return;

}

}

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

}

// Binary Search (Assumes sorted by title)

public static void binarySearch(Book[] books, String targetTitle) {

System.out.println(" Binary Search for '" + targetTitle + "':");

int left = 0;

int right = books.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

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

if (cmp == 0) {

System.out.println(" Found: " + books[mid]);

return;

} else if (cmp < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

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

}

public static void main(String[] args) {

Book[] library = {

new Book(101, "The Alchemist", "Paulo Coelho"),

new Book(102, "Harry Potter", "J.K. Rowling"),

new Book(103, "Wings of Fire", "A.P.J. Abdul Kalam"),

new Book(104, "Zero to One", "Peter Thiel"),

new Book(105, "Atomic Habits", "James Clear")

};

// Linear Search

linearSearch(library, "Wings of Fire");

// Sort before Binary Search

Arrays.sort(library, Comparator.comparing(book -> book.title.toLowerCase()));

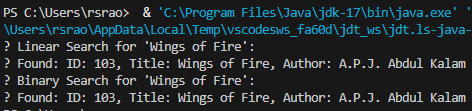
// Binary Search

binarySearch(library, "Wings of Fire");

}

}

**OUTPUT:-**

****

**Step 4: Time Complexity Comparison:-**

Linear Search:-O(1)

Binary Search:-O(1)

Exercise 7: Financial Forecasting Scenario:

**Step 1: Understand Recursion:-**

Recursion is when a method calls itself to solve smaller parts of a larger problem.

Example:-

int factorial(int n) {

if (n == 1) return 1;

return n \* factorial(n - 1);

}

### **Why Use Recursion?**

* Breaks down complex problems into simpler ones
* Great for tree structures, factorials, Fibonacci, and growth modeling

**Step 2–3: Full Java Code:-**

public class FinancialForecast {

// Recursive method to calculate future value

public static double forecast(double currentValue, double growthRate, int years) {

// Base case

if (years == 0) return currentValue;

// Recursive case

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

}

// Optimized version using memoization (optional)

public static double forecastMemo(double currentValue, double growthRate, int years, double[] memo) {

if (years == 0) return currentValue;

if (memo[years] != 0) return memo[years];

memo[years] = forecastMemo(currentValue \* (1 + growthRate), growthRate, years - 1, memo);

return memo[years];

}

public static void main(String[] args) {

double presentValue = 10000.0; // Rs. 10,000

double rate = 0.10; // 10% growth rate

int futureYears = 5;

// Recursive calculation

double futureValue = forecast(presentValue, rate, futureYears);

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

// Optional memoized version

double[] memo = new double[futureYears + 1];

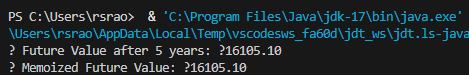
double memoValue = forecastMemo(presentValue, rate, futureYears, memo);

System.out.printf(" Memoized Future Value: ₹%.2f\n", memoValue);

}

}

OUTPUT:-



**Step 4: Analysis:-**

### Time Complexity:

* Recursive approach:
  + Time: O(n) (one call per year)
  + Space: O(n) (call stack depth)
* Memoized version:
  + Time: O(n)
  + Space: O(n) (memo array + call stack)

### **Optimization Tip:**

If you use dynamic programming or iterative loops, you avoid the risk of stack overflow in deep recursion.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |