**Cognizant Deep Nurture 4.0 Hands-on Exercise    
   
   
   
   
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.

Define the Product Class

// File: Product.java

public 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;

}

@Override

public String toString() {

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

}

}

Inventory Management Class using HashMap

// File: InventoryManager.java

import java.util.HashMap;

public class InventoryManager {

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

// Add a product

public void addProduct(Product product) {

inventory.put(product.productId, product);

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

}

// Update a product

public void updateProduct(int id, String name, int quantity, double price) {

if (inventory.containsKey(id)) {

Product product = inventory.get(id);

product.productName = name;

product.quantity = quantity;

product.price = price;

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

} else {

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

}

}

// Delete a product

public void deleteProduct(int id) {

if (inventory.remove(id) != null) {

System.out.println("Product with ID " + id + " removed.");

} else {

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

}

}

// View all products

public void viewInventory() {

System.out.println("\nCurrent Inventory:");

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

System.out.println(p);

}

}

}

Test Class

// File: Main.java

public class Main {

public static void main(String[] args) {

InventoryManager manager = new InventoryManager();

// Add products

manager.addProduct(new Product(101, "Laptop", 20, 899.99));

manager.addProduct(new Product(102, "Phone", 50, 499.99));

// View inventory

manager.viewInventory();

// Update a product

manager.updateProduct(102, "Smartphone", 45, 549.99);

// Delete a product

manager.deleteProduct(101);

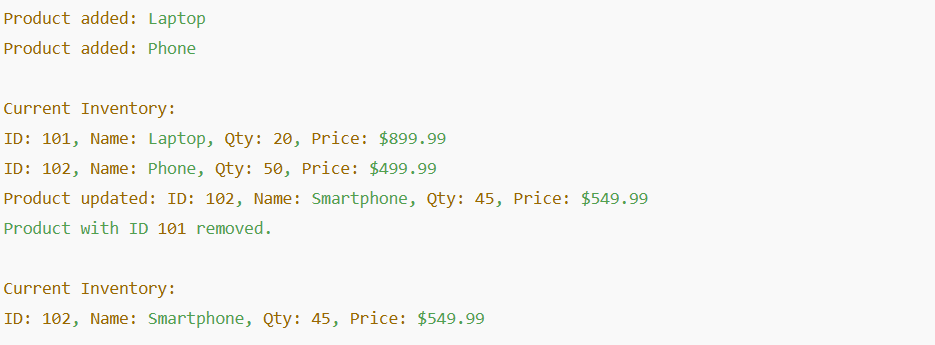
// Final inventory

manager.viewInventory();

}

}

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.

**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.

   
1.Setup Order Class

// File: Order.java

public 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;

}

public String toString() {

return "Order ID: " + orderId + ", Customer: " + customerName + ", Total: $" + totalPrice;

}

}

2. Implementation

Bubble Sort Implementation

// File: OrderSorter.java

public class OrderSorter {

public 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 - 1 - i; j++) {

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

// Swap

Order temp = orders[j];

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

orders[j + 1] = temp;

swapped = true;

}

}

if (!swapped) break; // Optimization for already sorted arrays

}

}

Quick Sort Implementation

public static void quickSort(Order[] orders, int low, int high) {

if (low < high) {

int pivotIndex = partition(orders, low, high);

quickSort(orders, low, pivotIndex - 1);

quickSort(orders, pivotIndex + 1, high);

}

}

private 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;

}

// Utility method to print orders

public static void printOrders(Order[] orders) {

for (Order order : orders) {

System.out.println(order);

}

}

}

3. Test the Sorting

// File: Main.java

public class Main {

public static void main(String[] args) {

Order[] orders = {

new Order(1, "Alice", 250.0),

new Order(2, "Bob", 100.0),

new Order(3, "Charlie", 300.0),

new Order(4, "Diana", 200.0)

};

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

OrderSorter.printOrders(orders);

// Test Bubble Sort

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

OrderSorter.bubbleSort(orders);

OrderSorter.printOrders(orders);

// Reinitialize for Quick Sort

orders = new Order[]{

new Order(1, "Alice", 250.0),

new Order(2, "Bob", 100.0),

new Order(3, "Charlie", 300.0),

new Order(4, "Diana", 200.0)

};

// Test Quick Sort

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

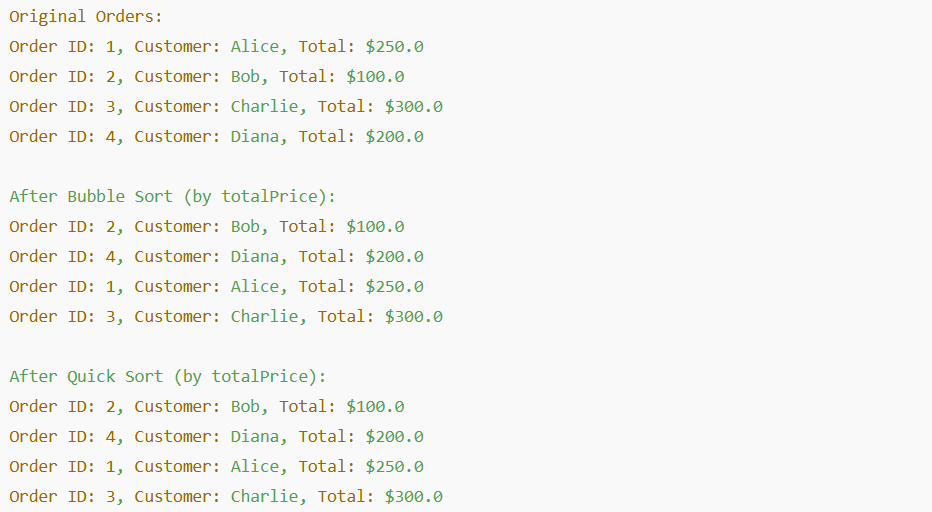
OrderSorter.quickSort(orders, 0, orders.length - 1);

OrderSorter.printOrders(orders);

}

}

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.

   
1.Employee Class

// File: Employee.java

public 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;

}

}

2. Implementation with Array

// File: EmployeeManager.java

public class EmployeeManager {

private Employee[] employees;

private int count;

public EmployeeManager(int capacity) {

employees = new Employee[capacity];

count = 0;

}

// Add employee

public void addEmployee(Employee emp) {

if (count < employees.length) {

employees[count++] = emp;

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

} else {

System.out.println("Cannot add employee: Array is full.");

}

}

// Search employee by ID

public Employee searchEmployee(int id) {

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

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

return employees[i];

}

}

return null;

}

// Traverse (display all)

public void displayEmployees() {

System.out.println("\nAll Employees:");

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

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

}

}

// Delete employee by ID

public void deleteEmployee(int id) {

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

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

// Shift elements to fill gap

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.");

}

}

Test Class

// File: Main.java

public class Main {

public static void main(String[] args) {

EmployeeManager manager = new EmployeeManager(5);

manager.addEmployee(new Employee(101, "Alice", "Developer", 75000));

manager.addEmployee(new Employee(102, "Bob", "Designer", 65000));

manager.addEmployee(new Employee(103, "Charlie", "Manager", 85000));

manager.displayEmployees();

// Search

System.out.println("\nSearching for employee with ID 102:");

Employee emp = manager.searchEmployee(102);

System.out.println(emp != null ? emp : "Employee not found.");

// Delete

manager.deleteEmployee(102);

manager.displayEmployees();

}

}

OUTPUT: 



**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.

   
1. Define the Task Class

// File: Task.java

public class Task {

int taskId;

String taskName;

String status;

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

this.taskId = taskId;

this.taskName = taskName;

this.status = status;

}

public String toString() {

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

}

}

2. Implementation Using Singly Linked List

Define Node and TaskList classes

// File: TaskNode.java

class TaskNode {

Task task;

TaskNode next;

public TaskNode(Task task) {

this.task = task;

this.next = null;

}

}

// File: TaskManager.java

public class TaskManager {

private TaskNode head;

// Add task at the end

public void addTask(Task task) {

TaskNode newNode = new TaskNode(task);

if (head == null) {

head = newNode;

} else {

TaskNode current = head;

while (current.next != null) {

current = current.next;

}

current.next = newNode;

}

System.out.println("Task added: " + task.taskName);

}

// Search task by ID

public Task searchTask(int id) {

TaskNode current = head;

while (current != null) {

if (current.task.taskId == id) {

return current.task;

}

current = current.next;

}

return null;

}

// Traverse tasks

public void displayTasks() {

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

TaskNode current = head;

while (current != null) {

System.out.println(current.task);

current = current.next;

}

}

// Delete task by ID

public void deleteTask(int id) {

if (head == null) {

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

return;

}

if (head.task.taskId == id) {

head = head.next;

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

return;

}

TaskNode current = head;

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

current = current.next;

}

if (current.next != null) {

current.next = current.next.next;

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

} else {

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

}

}

}

3. Testing the Task Management System

// File: Main.java

public class Main {

public static void main(String[] args) {

TaskManager manager = new TaskManager();

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

manager.addTask(new Task(2, "Write Backend", "In Progress"));

manager.addTask(new Task(3, "Test Application", "Pending"));

manager.displayTasks();

// Search

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

Task result = manager.searchTask(2);

System.out.println(result != null ? result : "Task not found.");

// Delete

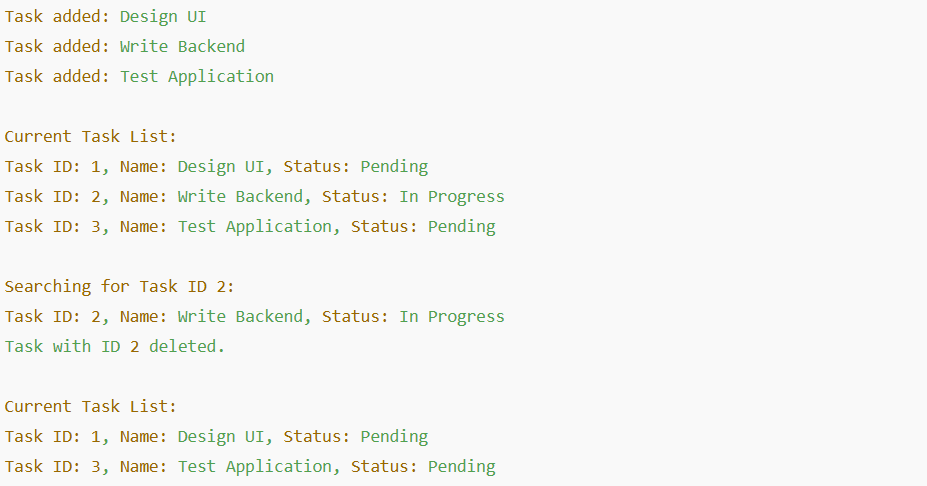
manager.deleteTask(2);

manager.displayTasks();

}

}

OUTPUT: 



   
   
**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.

1.Define the Book Class

// File: Book.java

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 String toString() {

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

}

}

2. Implementation – Linear and Binary Search

// File: LibraryManager.java

import java.util.Arrays;

import java.util.Comparator;

public class LibraryManager {

// Linear search by title

public static Book linearSearchByTitle(Book[] books, String title) {

for (Book book : books) {

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

return book;

}

}

return null;

}

// Binary search by title (must be sorted by title first)

public static Book binarySearchByTitle(Book[] books, String title) {

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

while (low <= high) {

int mid = (low + high) / 2;

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

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

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

else high = mid - 1;

}

return null;

}

// Sort books by title

public static void sortBooksByTitle(Book[] books) {

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

}

// Print all books

public static void printBooks(Book[] books) {

for (Book book : books) {

System.out.println(book);

}

}

}

3. Testing the Library System

// File: Main.java

public class Main {

public static void main(String[] args) {

Book[] books = {

new Book(1, "The Alchemist", "Paulo Coelho"),

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

new Book(3, "1984", "George Orwell"),

new Book(4, "Moby Dick", "Herman Melville"),

new Book(5, "Pride and Prejudice", "Jane Austen")

};

System.out.println("=== All Books ===");

LibraryManager.printBooks(books);

// Linear Search

System.out.println("\n🔍 Linear Search for '1984':");

Book foundLinear = LibraryManager.linearSearchByTitle(books, "1984");

System.out.println(foundLinear != null ? foundLinear : "Book not found.");

// Sort before Binary Search

LibraryManager.sortBooksByTitle(books);

// Binary Search

System.out.println("\n🔍 Binary Search for '1984':");

Book foundBinary = LibraryManager.binarySearchByTitle(books, "1984");

System.out.println(foundBinary != null ? foundBinary : "Book not found.");

}

}

OUTPUT:

