**WEEK-1**

**Data Structures and Algorithms**

**Exercise 1: Inventory Management System**

**Problem Understanding:**

Fast operations needed for large inventory systems.

**Data Structure Used:**

Used HashMap with product Id as key (O(1)).

**Code Implementation Summary:**

Product stores details.

Warehouse Inventory manages all actions.

Inventory shows usage flow.

**Time Complexity:**

Add/Update/Delete – O(1)

Display – O(n)

**Optimization Discussion:**

Use TreeMap for sorted output (O(log n)).

**Code:**

**Java Project (Package) : Inventory Management System**

**Product.java:**

import java.util.HashMap;

class Product {

private final String productId;

private String productName;

private int availableUnits;

private double unitPrice;

public Product(String productId, String productName, int availableUnits, double unitPrice) {

this.productId = productId;

this.productName = productName;

this.availableUnits = availableUnits;

this.unitPrice = unitPrice;

}

public String getProductId() {

return productId;

}

public void modifyDetails(String name, int units, double price) {

this.productName = name;

this.availableUnits = units;

this.unitPrice = price;

}

public String getDetails() {

return String.format("ID: %s | Name: %s | Qty: %d | Price: ₹%.2f", productId, productName, availableUnits, unitPrice);

}

}

class WarehouseInventory {

private final HashMap<String, Product> productRegistry = new HashMap<>();

public void registerProduct(Product product) {

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

}

public void updateProduct(String id, String newName, int newQty, double newPrice) {

Product existing = productRegistry.get(id);

if (existing != null) {

existing.modifyDetails(newName, newQty, newPrice);

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

} else {

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

}

}

public void deleteProduct(String id) {

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

System.out.println("Product " + id + " removed from inventory.");

} else {

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

}

}

public void displayInventory() {

if (productRegistry.isEmpty()) {

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

return;

}

System.out.println("=== Inventory List ===");

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

System.out.println(p.getDetails());

}

}

}

**Inventory.java:**

public class Inventory {

public static void main(String[] args) {

WarehouseInventory warehouse = new WarehouseInventory();

Product prod1 = new Product("ITM1001", "HDMI Cable", 30, 250.75);

Product prod2 = new Product("ITM1002", "Laptop Stand", 15, 1150.00);

warehouse.registerProduct(prod1);

warehouse.registerProduct(prod2);

warehouse.displayInventory();

warehouse.updateProduct("ITM1001", "Premium HDMI Cable", 25, 299.99);

warehouse.deleteProduct("ITM1002");

warehouse.displayInventory();

}

**Output:**

**A screenshot of a computer program

AI-generated content may be incorrect.**

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

**Problem Understanding:**

Search system lets users find products by name using two methods.

**Data Structure Used:**

Used two arrays: rawData (linear) and orderedData (binary).

**Code Implementation Summary:**

Product stores info.

Search Engine adds, sorts, and searches items.

**Time Complexity:**

Add – O(1)

Sort – O(n log n)

Linear – O(n)

Binary – O(log n)

**Optimization Discussion:**

Binary is faster but needs sorting. Use Array List/Tree Set for dynamic data.

**Code:**

**Java Project (Package) : E Commerce Platform**

**Product.java:**

import java.util.Arrays;

import java.util.Comparator;

class Product {

String productId;

String productName;

String category;

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

public String toString() {

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

}

}

class SearchEngine {

private Product[] rawData;

private Product[] orderedData;

private int count;

SearchEngine(int max) {

rawData = new Product[max];

orderedData = new Product[max];

count = 0;

}

void addItem(Product p) {

if (count < rawData.length) {

rawData[count] = p;

orderedData[count] = p;

count++;

}

}

void prepareSorted() {

Arrays.sort(orderedData, 0, count, Comparator.comparing(x -> x.productName.toLowerCase()));

}

Product searchLinear(String target) {

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

if (rawData[i].productName.equalsIgnoreCase(target)) {

return rawData[i];

}

}

return null;

}

Product searchBinary(String target) {

int start = 0, end = count - 1;

while (start <= end) {

int mid = (start + end) / 2;

int res = orderedData[mid].productName.compareToIgnoreCase(target);

if (res == 0) return orderedData[mid];

if (res < 0) start = mid + 1;

else end = mid - 1;

}

return null;

}

void showCatalog() {

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

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

System.out.println("- " + rawData[i]);

}

}

}

**Ecommerce.java:**

public class Ecommerce {

public static void main(String[] args) {

SearchEngine engine = new SearchEngine(10);

engine.addItem(new Product("PRD001", "Tablet", "Electronics"));

engine.addItem(new Product("PRD002", "Backpack", "Accessories"));

engine.addItem(new Product("PRD003", "Camera", "Electronics"));

engine.addItem(new Product("PRD004", "Jeans", "Clothing"));

engine.prepareSorted();

Product result1 = engine.searchLinear("Backpack");

if (result1 != null)

System.out.println("Linear Search >> Match Found: " + result1);

else

System.out.println("Linear Search >> No matching item");

Product result2 = engine.searchBinary("Camera");

if (result2 != null)

System.out.println("Binary Search >> Match Found: " + result2);

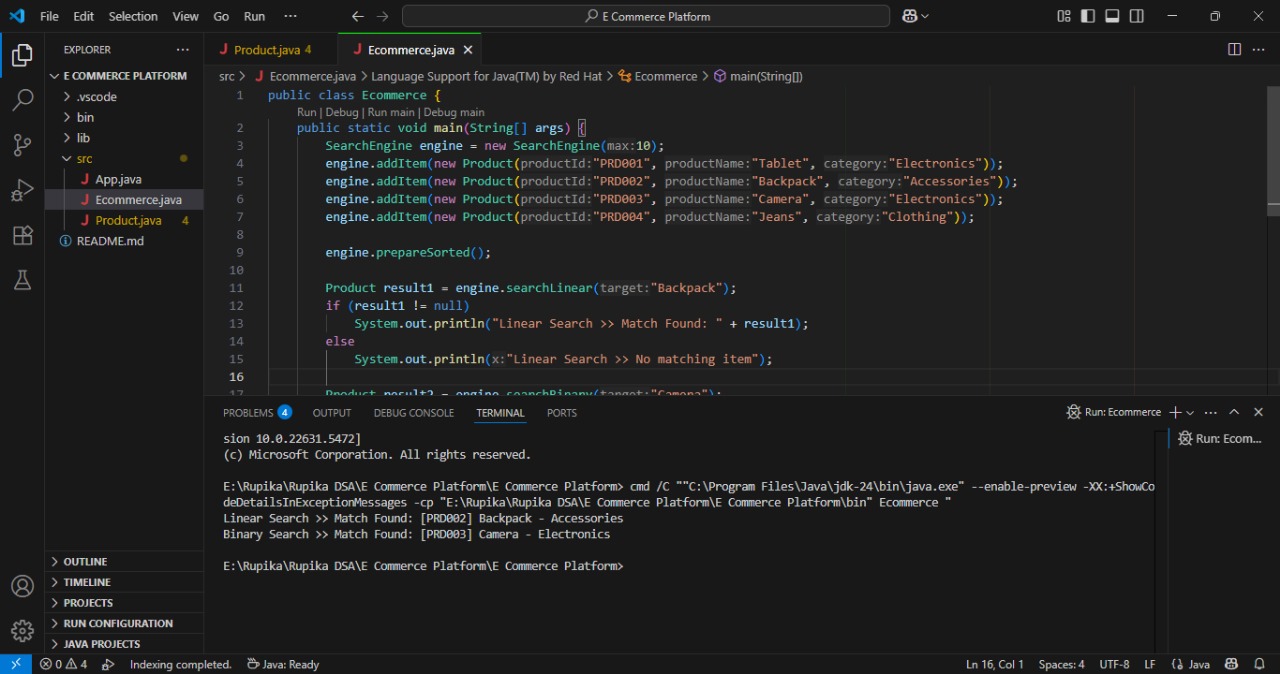
else

System.out.println("Binary Search >> No matching item");

}

}

**Output:**



**Exercise 3: Sorting Customer Orders**

**Problem Understanding:**

Sorting orders by total price helps in billing and analysis.

**Data Structure Used:**

Used an array of Order objects for sorting.

**Code Implementation Summary:**

Order stores details.

SortManager performs Bubble and Quick Sort.

Main class tests both methods.

**Time Complexity:**

Bubble – O(n²)

Quick – O(n log n)

print – O(n)

**Optimization Discussion:**

Quick Sort is faster. Use Arrays.sort() for real apps**.**

**Code:**

**Java Project (Package)** :Sorting Customer Orders

**Order.java:**

class Order {

String orderId;

String customerName;

double totalPrice;

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

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

public String toString() {

return "Order#" + orderId + " | Customer: " + customerName + " | Amount: ₹" + totalPrice;

}

}

class SortManager {

void sortByBubble(Order[] items) {

int len = items.length;

for (int round = 0; round < len - 1; round++) {

boolean flag = false;

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

if (items[i].totalPrice > items[i + 1].totalPrice) {

Order swap = items[i];

items[i] = items[i + 1];

items[i + 1] = swap;

flag = true;

}

}

if (!flag) break;

}

}

void sortByQuick(Order[] array, int low, int high) {

if (low < high) {

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

sortByQuick(array, low, pivotIndex - 1);

sortByQuick(array, pivotIndex + 1, high);

}

}

int partition(Order[] data, int start, int end) {

double pivotValue = data[end].totalPrice;

int i = start - 1;

for (int j = start; j < end; j++) {

if (data[j].totalPrice <= pivotValue) {

i++;

Order tmp = data[i];

data[i] = data[j];

data[j] = tmp;

}

}

Order temp = data[i + 1];

data[i + 1] = data[end];

data[end] = temp;

return i + 1;

}

void printAll(Order[] dataset) {

for (Order ord : dataset) {

System.out.println(ord);

}

}

}

**OrdersorterApp.java:**

import java.util.Arrays;

public class OrderSorterApp {

public static void main(String[] args) {

Order[] sampleOrders = {

new Order("ORD201", "Meera", 750.00),

new Order("ORD202", "Ayaan", 320.40),

new Order("ORD203", "Zara", 1299.99),

new Order("ORD204", "Ravi", 560.75)

};

SortManager handler = new SortManager();

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

Order[] firstPass = Arrays.copyOf(sampleOrders, sampleOrders.length);

handler.sortByBubble(firstPass);

handler.printAll(firstPass);

System.out.println("\n>>> Sorted using Quick Sort:");

Order[] secondPass = Arrays.copyOf(sampleOrders, sampleOrders.length);

handler.sortByQuick(secondPass, 0, secondPass.length - 1);

handler.printAll(secondPass);

}

}

**Output:**

A screenshot of a computer program

AI-generated content may be incorrect.

**Exercise 4: Employee Management System**

**Problem Understanding:**

Manages employee records with operations like add, search, delete, and view.

**Data Structure Used:**

Uses a fixed-size array to store Employee objects.

**Code Implementation Summary:**

Employee holds ID, name, role, salary.

HRSystem handles register, find, discard, and list actions**.**

**Time Complexity:**

Register – O(1)

Search/Delete – O(n)

List – O(n)

**Optimization Discussion:**

Using ArrayList or HashMap would allow better scalability and faster access.

**Code:**

**Java Project (Package)** :Employee Management System

**Employee.java:**

class Employee {

String employeeId;

String name;

String position;

double salary;

Employee(String 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 + " | Role: " + position + " | Pay: ₹" + salary;

}

}

class HRSystem {

private Employee[] directory;

private int filled;

HRSystem(int max) {

directory = new Employee[max];

filled = 0;

}

boolean register(Employee emp) {

if (filled >= directory.length) return false;

directory[filled++] = emp;

return true;

}

Employee find(String id) {

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

if (directory[i].employeeId.equals(id)) return directory[i];

}

return null;

}

**HR,java:**

boolean discard(String id) {

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

if (directory[i].employeeId.equals(id)) {

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

directory[j] = directory[j + 1];

}

directory[--filled] = null;

return true;

}

}

return false;

}

void list() {

if (filled == 0) {

System.out.println(">>> No employee data present.");

return;

}

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

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

}

}

}

**EmployeeApp.java:**

public class EmployeeApp {

public static void main(String[] args) {

HRSystem system = new HRSystem(5);

system.register(new Employee("EM101", "Ananya", "Team Lead", 82000));

system.register(new Employee("EM102", "Raj", "Engineer", 61000));

system.register(new Employee("EM103", "Isha", "Analyst", 49000));

System.out.println(">>> Current Employee List:");

system.list();

System.out.println("\n>>> Searching for EM102:");

Employee result = system.find("EM102");

System.out.println(result != null ? result : "No match found.");

System.out.println("\n>>> Deleting EM101:");

boolean removed = system.discard("EM101");

System.out.println(removed ? "Entry removed successfully." : "Employee ID not found.");

System.out.println("\n>>> Employee List After Update:");

system.list();

}

}

**Output:**

A screen shot of a computer

AI-generated content may be incorrect.

**Exercise 5: Task Management System**

**Problem Understanding:**

Manages tasks with add, search, delete, and display operations using a linked list.

**Data Structure Used:**

Used a singly linked list (Task nodes) to dynamically manage tasks.

**Code Implementation Summary:**

Task holds ID, name, status, and link to next.

TaskChain handles all task operations with traversal logic.

**Time Complexity:**

Add – O(n)

Search – O(n)

Delete – O(n)

Display – O(n)

**Optimization Discussion:**

For faster operations, a HashMap could be added for direct ID access.

**Code:**

**Java Project (Package)** :Sorting Customer Orders

**Task.java:**

class Task {

String taskId;

String taskName;

String status;

Task next;

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

this.taskId = taskId;

this.taskName = taskName;

this.status = status;

this.next = null;

}

public String toString() {

return "Task[" + taskId + "] - " + taskName + " | Status: " + status;

}

}

class TaskChain {

private Task start;

void addTask(String id, String name, String state) {

Task entry = new Task(id, name, state);

if (start == null) {

start = entry;

} else {

Task ptr = start;

while (ptr.next != null) ptr = ptr.next;

ptr.next = entry;

}

}

boolean deleteTask(String id) {

if (start == null) return false;

if (start.taskId.equals(id)) {

start = start.next;

return true;

}

Task prev = start;

Task current = start.next;

while (current != null) {

if (current.taskId.equals(id)) {

prev.next = current.next;

return true;

}

prev = current;

current = current.next;

}

return false;

}

Task searchTask(String id) {

Task node = start;

while (node != null) {

if (node.taskId.equals(id)) return node;

node = node.next;

}

return null;

}

void displayAll() {

if (start == null) {

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

return;

}

Task temp = start;

while (temp != null) {

System.out.println(temp);

temp = temp.next;

}

}

}

**TaslFlow.java:**

public class Mai

public class TaskFlow {

public static void main(String[] args) {

TaskChain flow = new TaskChain();

flow.addTask("TSK001", "UI Mockup", "To Do");

flow.addTask("TSK002", "Logic Layer", "In Progress");

flow.addTask("TSK003", "Final Testing", "To Do");

System.out.println(">>> Task Overview:");

flow.displayAll();

System.out.println("\n>>> Search Result for TSK002:");

Task result = flow.searchTask("TSK002");

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

System.out.println("\n>>> Delete Attempt for TSK001:");

boolean status = flow.deleteTask("TSK001");

System.out.println(status ? "Successfully removed." : "Deletion failed.");

System.out.println("\n>>> Tasks After Update:");

flow.displayAll();

}

}

**Output:**

A screenshot of a computer

AI-generated content may be incorrect.

**Exercise 6: Library Management System**

**Problem Understanding:**

Provides a way to search books by title using linear and binary search methods.

**Data Structure Used:**

Uses a List<Book> for flexible and efficient book storage and access.

**Code Implementation Summary:**

Book holds ID, title, and author.

LibrarySearch performs linear and binary search by title.

**Time Complexity:**

Linear – O(n)

Binary – O(log n)

after sort O(n log n)

**Optimization Discussion:**

Use HashMap<String, Book> for constant-time title search in large libraries.

**Code:**

**Java Project (Package)** :Library Management System

**Book.java:**

public class Book {

int bookId;

String title;

String author;

public Book(int id, String title, String author) {

this.bookId = id;

this.title = title;

this.author = author;

}

public String toString() {

return "[" + bookId + "] " + title + " by " + author;

}

}

**LibrarySearch.java:**

import java.util.\*;

public class LibrarySearch {

public static Book linearSearch(List<Book> books, String title) {

for (Book b : books) {

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

}

return null;

}

public static Book binarySearch(List<Book> books, String title) {

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

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

while (low <= high) {

int mid = (low + high) / 2;

Book midBook = books.get(mid);

int cmp = midBook.title.compareToIgnoreCase(title);

if (cmp == 0) return midBook;

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

else high = mid - 1;

}

return null;

}

}

**Main.java:**

import java.util.\*;

public class Main {

public static void main(String[] args) {

List<Book> books = List.of(

new Book(1, "Clean Code", "Robert C. Martin"),

new Book(2, "Design Patterns", "Erich Gamma"),

new Book(3, "Algorithms", "Cormen")

);

Book result1 = LibrarySearch.linearSearch(books, "Clean Code");

System.out.println("Linear Search: " + result1);

Book result2 = LibrarySearch.binarySearch(new ArrayList<>(books), "Clean Code");

System.out.println("Binary Search: " + result2);

}

}

**Output:**

A screenshot of a computer program

AI-generated content may be incorrect.

**Exercise 7: Financial Forecasting**

**Problem Understanding:**

Calculates the future value of money using compound interest over time.

**Data Structure Used:**

No complex structures; uses recursion to simulate yearly compounding.

**Code Implementation Summary:**

futureValue() recursively multiplies value with growth rate.

main() calls it with preset values.

**Time Complexity:**

Recursive calls – O(n) where n = number of years

**Optimization Discussion:**

Use iteration for better efficiency and avoid stack overflow in deep recursion.

**Code:**

**Java Project (Package) : Financial Forecasting**

**ForecastForecast.java:**

public class FinancialForecast {

// Recursive method

public static double futureValue(double pv, double rate, int years) {

if (years == 0) return pv; // base case

return futureValue(pv, rate, years - 1) \* (1 + rate);

}

public static void main(String[] args) {

double presentValue = 10000; // ₹10,000

double rate = 0.05; // 5% growth rate

int years = 5;

double result = futureValue(presentValue, rate, years);

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

}

}

**Output:**

