**ADDITIONAL HANDS-ON SOLUTIONS**

**DATA STRUCTURES & ALGORITHMS**

**Week 1**

**Exercise 1: Inventory Management System**

**Step 1: Understanding the Problem**

In a warehouse environment, managing large volumes of products demands efficient data storage and fast access to product details. DSA play a vital role in ensuring that operations like product addition, lookup, update, and deletion are handled with minimal time and space complexity.

It ensure speed and scalability in handling large inventories by enabling fast operations like search, update, and delete. They also support clean, maintainable code, making integration with UI or databases seamless.

Suitable Data Structures:

* ArrayList: Useful for simple product listings, best for sequential operations.
* HashMap<Integer, Product>: Ideal for fast lookup by productId — average time complexity: O(1) for add, update, delete.

**Step 2: Setup**

Project Name: InventoryManagementSystem

**Step 3: Implementation**

**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;

}

public void display() {

System.out.println("ID: " + productId + ", Name: " + productName +

", Quantity: " + quantity + ", Price: ₹" + price);

}

}

**Inventory.java**

import java.util.HashMap;

public class Inventory {

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

public void addProduct(Product p) {

inventory.put(p.productId, p);

System.out.println("Product added successfully.");

}

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

if (inventory.containsKey(productId)) {

Product p = inventory.get(productId);

p.quantity = newQty;

p.price = newPrice;

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

} else {

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

}

}

public void deleteProduct(int productId) {

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

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

} else {

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

}

}

public void displayAll() {

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

p.display();

}

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

Inventory inv = new Inventory();

Product p1 = new Product(1, "Mouse", 20, 499.99);

Product p2 = new Product(2, "Keyboard", 15, 999.50);

inv.addProduct(p1);

inv.addProduct(p2);

inv.displayAll();

inv.updateProduct(1, 25, 459.00);

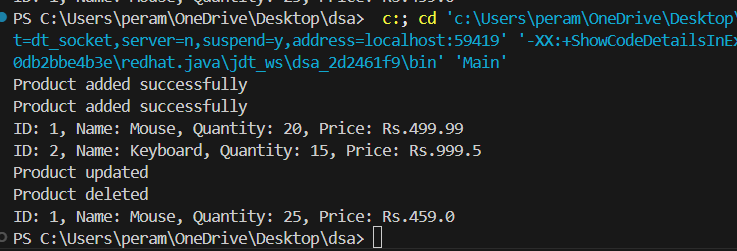
inv.deleteProduct(2);

inv.displayAll();

}

}

**OUTPUT:**

****

**Step 4: Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Data Structure | Time Complexity | Reason |
| Add Product | HashMap | O(1) | Direct insertion using unique productId as key |
| Update Product | HashMap | O(1) | Access by key is constant time |
| Delete Product | HashMap | O(1) | Removal by key also constant time |
| Display All | HashMap | O(n) | Need to iterate over all values |

**Exercise 3: Sorting Customer Orders**

**Step 1: Understanding Sorting Algorithms**

Sorting algorithms are used to arrange data in a specific order. Key ones include:

* Bubble Sort: Repeatedly steps through the list, compares adjacent elements, and swaps them if they’re in the wrong order.
  + Time Complexity: O(n²)
  + Simple but inefficient for large datasets.
* Insertion Sort: Builds the final sorted array one item at a time.
  + Time Complexity: O(n²)
  + Efficient for small datasets.
* Quick Sort: Divides and conquers by partitioning the array around a pivot.
  + Time Complexity: O(n log n) on average
  + Very efficient and widely used.
* Merge Sort: Divides the list into halves, sorts them, and merges.
  + Time Complexity: O(n log n)
  + Stable but uses extra space.

**Step 2: SetUp**

**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 void display() {

System.out.println("Order ID: " + orderId + ", Customer: " + customerName + ", Total Price: Rs." + totalPrice);

}

}

**SortUtils.java**

public class SortUtils {

public 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;

}

}

}

}

public 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);

}

}

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;

}

}

**OrderSortingDemo.java**

public class OrderSortingDemo {

public static void main(String[] args) {

Order[] orders = {

new Order(101, "Ravi", 5999.99),

new Order(102, "Anu", 2199.49),

new Order(103, "Kiran", 9999.00),

new Order(104, "Leela", 4500.75)

};

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

for (Order o : orders) o.display();

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

SortUtils.bubbleSort(orders);

for (Order o : orders) o.display();

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

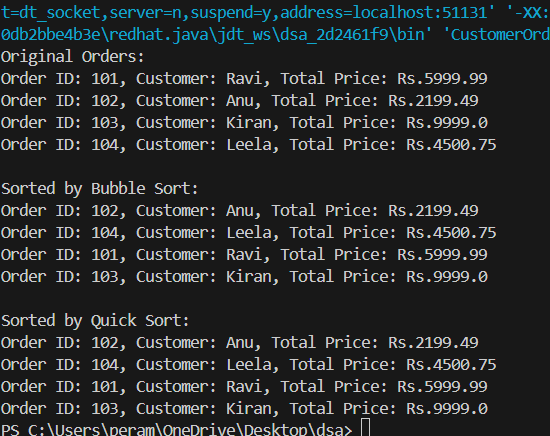
SortUtils.quickSort(orders, 0, orders.length - 1);

for (Order o : orders) o.display();

}

}

**OUTPUT:**



**Step 4: Analysis**

Bubble Sort:

* Time: O(n²)
* Easy to implement, but slow for large data.

Quick Sort:

* Time: O(n log n) on average
* Much faster due to better partitioning and recursion.

**Exercise 4: Employee Management System**

**Step 1: Understanding Array Representation**

Arrays are contiguous memory blocks where elements are stored at adjacent addresses. Each element is accessed via an index, enabling constant-time access (O(1)). They are efficient for fixed-size data but not ideal when dynamic resizing is required.

Advantages of Arrays:

* Fast random access using indices
* Simple and memory-efficient for static collections
* Easy to traverse and sort

**Implementation:**

**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 void display() {

System.out.println("ID: " + employeeId + ", Name: " + name +

", Position: " + position + ", Salary: ₹" + salary);

}

}

**EmployeeManager.java**

public class EmployeeManager {

private Employee[] employees = new Employee[100]; // Max 100 employees

private int count = 0;

public void addEmployee(Employee e) {

if (count < employees.length) {

employees[count++] = e;

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

} else {

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

}

}

public void searchEmployee(int id) {

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

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

employees[i].display();

return;

}

}

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

}

public void listEmployees() {

if (count == 0) {

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

return;

}

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

employees[i].display();

}

}

public 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];

}

employees[--count] = null;

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

return;

}

}

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

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

EmployeeManager manager = new EmployeeManager();

manager.addEmployee(new Employee(201, "Ruthvika", "Engineer", 50000));

manager.addEmployee(new Employee(202, "Rahul", "Analyst", 42000));

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

manager.listEmployees();

System.out.println("\nSearching for ID 201:");

manager.searchEmployee(201);

System.out.println("\nDeleting ID 202:");

manager.deleteEmployee(202);

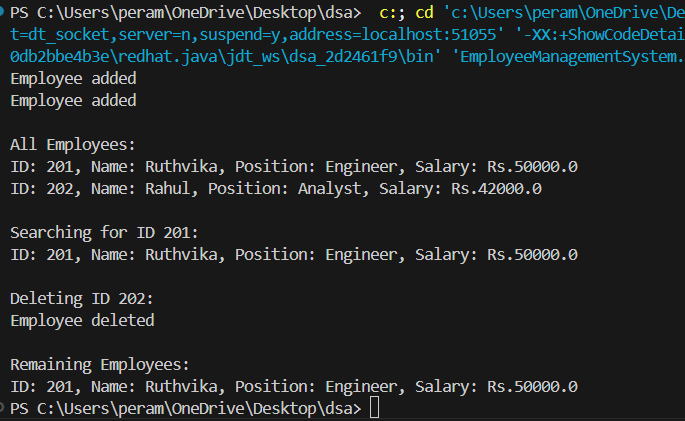
System.out.println("\nRemaining Employees:");

manager.listEmployees();

}

}

**OUTPUT:**



**Step 4: Analysis**

|  |  |
| --- | --- |
| Operation | Time Complexity |
| Add | O(1) |
| Search | O(n) |
| Traverse | O(n) |
| Delete | O(n) (due to shifting) |

Limitations of Arrays:

* Fixed size — can't dynamically resize
* Insertion/deletion is inefficient in the middle
* Not suitable when frequent modifications are needed

Better Alternatives:

* Use ArrayList or LinkedList for dynamic size and flexibility
* Use HashMap if search by ID needs to be faster (O(1))

**Exercise 5: Task Management System**

**Step 1: Understanding Linked Lists**

A Linked List is a linear data structure where elements are connected using pointers. Unlike arrays, elements are not stored in contiguous memory locations.

Types of Linked Lists:

* Singly Linked List: Each node points to the next node only.
* Doubly Linked List: Each node points to both the next and previous nodes.
* Circular Linked List: The last node points back to the first node.

For this task management system, we use a Singly Linked List.

**Step 2: Setup**

**Define Task Class**

**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 void display() {

System.out.println("Task ID: " + taskId + ", Name: " + taskName + ", Status: " + status);

}

}

**Step 3: Implementation**

**TaskNode.java**

public class TaskNode {

Task task;

TaskNode next;

public TaskNode(Task task) {

this.task = task;

this.next = null;

}

}

**TaskManager.java**

public class TaskManager {

private TaskNode head = null;

public void addTask(Task task) {

TaskNode newNode = new TaskNode(task);

if (head == null) {

head = newNode;

} else {

TaskNode temp = head;

while (temp.next != null) {

temp = temp.next;

}

temp.next = newNode;

}

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

}

public void searchTask(int taskId) {

TaskNode temp = head;

while (temp != null) {

if (temp.task.taskId == taskId) {

temp.task.display();

return;

}

temp = temp.next;

}

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

}

public void deleteTask(int taskId) {

if (head == null) {

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

return;

}

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

head = head.next;

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

return;

}

TaskNode prev = head;

TaskNode current = head.next;

while (current != null) {

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

prev.next = current.next;

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

return;

}

prev = current;

current = current.next;

}

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

}

public void displayTasks() {

if (head == null) {

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

return;

}

TaskNode temp = head;

while (temp != null) {

temp.task.display();

temp = temp.next;

}

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

TaskManager tm = new TaskManager();

tm.addTask(new Task(101, "Design UI", "Pending"));

tm.addTask(new Task(102, "Fix bugs", "In Progress"));

tm.addTask(new Task(103, "Deploy app", "Pending"))

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

tm.displayTasks();

System.out.println("\nSearching Task 102:");

tm.searchTask(102);

System.out.println("\nDeleting Task 101:");

tm.deleteTask(101);

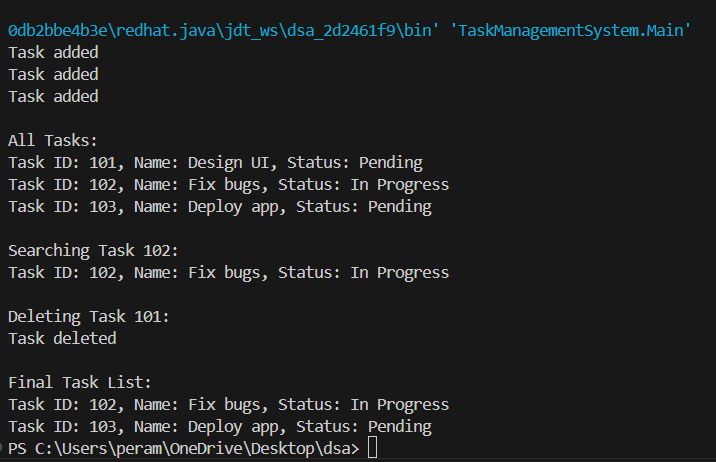
System.out.println("\nFinal Task List:");

tm.displayTasks();

}

}

**OUTPUT:**

****

**Step 4: Analysis**

|  |  |
| --- | --- |
| Operation | Time Complexity |
| Add Task | O(n) |
| Search Task | O(n) |
| Delete Task | O(n) |
| Display Tasks | O(n) |

Advantages over Arrays:

* Dynamic memory: No pre-defined size limit.
* Efficient insertions/deletions: No shifting required like arrays.

Limitations:

* No direct access by index
* Slightly more memory overhead due to node pointers

**Exercise 6: Library Management System**

**Step 1. Understanding Search Algorithms**

Linear Search

* Iterates through each element.
* Best case: O(1), Worst case: O(n)
* Simple, works on unsorted data.

Binary Search

* Divides the list in half repeatedly.
* Requires sorted data.
* Time complexity: O(log n)
* Efficient for large datasets.

**Step 2: Setup**

**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 void display() {

System.out.println("Book ID: " + bookId + ", Title: " + title + ", Author: " + author);

}

}

**Step 3: Implementation**

**LibrarySearch.java**

import java.util.Arrays;

import java.util.Comparator;

public class LibrarySearch {

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

for (Book b : books) {

if (b.title.equalsIgnoreCase(title)) {

return b;

}

}

return null;

}

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

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

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;

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

Book[] books = {

new Book(1, "Java Programming", "James Gosling"),

new Book(2, "Python Crash Course", "Eric Matthes"),

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

new Book(4, "Data Structures", "Seymour Lipschutz")

};

System.out.println("Linear Search Result:");

Book b1 = LibrarySearch.linearSearch(books, "Clean Code");

if (b1 != null) b1.display(); else System.out.println("Book not found");

System.out.println("\nBinary Search Result:");

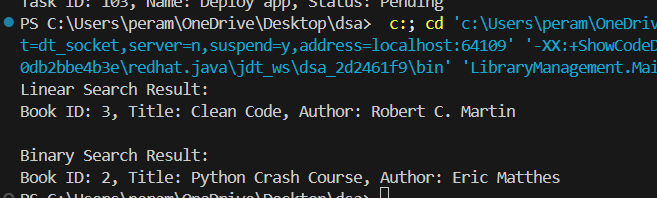
Book b2 = LibrarySearch.binarySearch(books, "Python Crash Course");

if (b2 != null) b2.display(); else System.out.println("Book not found");

}

}

**OUTPUT:**

****

**STEP 4: Analysis**

|  |  |  |
| --- | --- | --- |
| Algorithm | Time Complexity | Best For |
| Linear Search | O(n) | Unsorted/small datasets |
| Binary Search | O(log n) | Sorted/large datasets |