**WEEK 1**

**2. Data Structures and Algorithms HandsOn**

**Exercise 1: Inventory Management System**

Efficient handling of large data: Quick access, update, and retrieval of inventory items.Reduce time complexity: Ensures operations like search, insert, and delete are fast.

**Suitable Data Structures:**

**ArrayList**: Maintains order; good for small data and sequential access.

**HashMap**: Fast access using keys (productId); ideal for large datasets and fast lookup.

**TreeMap**: Sorted order access based on keys; useful when sorted view is needed.

**Code:**

**Product.java**

public class Product {

String productId;

String productName;

int quantity;

double price;

public Product(String 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<String, Product> inventory = new HashMap<>();

public void addProduct(Product p) {

inventory.put(p.productId, p);

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

}

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

if (inventory.containsKey(id)) {

Product p = inventory.get(id);

p.quantity = newQty;

p.price = newPrice;

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

} else {

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

}

}

public void deleteProduct(String id) {

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

System.out.println("Product deleted: " + id);

} else {

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

}

}

public void displayInventory() {

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

p.display();

}

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

Inventory inventory = new Inventory();

Product p1 = new Product("P101", "Mouse", 50, 25.99);

Product p2 = new Product("P102", "Keyboard", 30, 45.00);

inventory.addProduct(p1);

inventory.addProduct(p2);

System.out.println("\nInventory after adding:");

inventory.displayInventory();

inventory.updateProduct("P101", 70, 27.50);

System.out.println("\nInventory after updating P101:");

inventory.displayInventory();

inventory.deleteProduct("P102");

System.out.println("\nInventory after deleting P102:");

inventory.displayInventory();

}

}

**Output:**

A screenshot of a computer

AI-generated content may be incorrect.

**Time Complexity of Operations:**

Add Product: O(1) (average case) in HashMap.

Update Product: O(1) if productId exists.

Delete Product: O(1) using key in HashMap.

**Optimization Techniques:**

Use HashMap with well-distributed hash function to avoid collisions. For large systems, use database indexing and caching mechanism

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

**Big O Notation**

* Describes algorithm performance in terms of input size n.
* Focuses on upper bound (worst-case) time complexity.
* Helps compare efficiency of algorithms regardless of hardware.

**Search operations:**

There are two types of searching methods  
 i) Linear Search

ii) Binary Search

**i)Linear Search:**

Best Case : O(1)

Average Case : O(n)

Worst Case : O(n)

**ii)Binary Search:**

Best Case : O(1)

Average Case : O(log n)

Worst Case : O(log n)

**Code:**

**Product.java**

public class Product {

int productId;

String productName;

String category;

public Product(int id, String name, String category) {

this.productId = id;

this.productName = name;

this.category = category;

}

public void display() {

System.out.println("ID: " + productId + ", Name: " + productName + ", Category: " + category);

}

}

**Search.java**

public class Search {

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

for (int i = 0; i < products.length; i++) {

if (products[i].productName.equalsIgnoreCase(name)) {

return i;

}

}

return -1;

}

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

int left = 0, right = products.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

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

if (cmp == 0) return mid;

else if (cmp < 0) right = mid - 1;

else left = mid + 1;

}

return -1;

}

}

**Main.java**

import java.util.Arrays;

import java.util.Comparator;

public class Main {

public static void main(String[] args) {

Product[] products = {

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

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

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

new Product(104, "Shoes", "Footwear")

};

String searchName1 = "Shoes";

int result1 = Search.linearSearch(products, searchName1);

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

if (result1 != -1) products[result1].display();

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

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

String searchName2 = "Laptop";

int result2 = Search.binarySearch(products, searchName2);

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

if (result2 != -1) products[result2].display();

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

}

}**Output:**

A screenshot of a computer

AI-generated content may be incorrect.

**Linear Search:**

* Time Complexity: O(n)
* Unsorted array

**Binary Search:**

* Time Complexity: O(log n)
* Requires sorted array

**Suitable Algorithm:**

Binary Search is more suitable:

* Faster for large datasets (O(log n))
* Requires pre-sorting but offers better performance.
* Linear Search only useful for small or unsorted datasets.

**Exercise 3: Sorting Customer Orders**

**Sorting Algorithms:**

**Bubble Sort:**

* Repeatedly swaps adjacent elements if out of order.
* Time Complexity: O(n²)

**Insertion Sort:**

* Builds sorted array one item at a time.
* Time Complexity: O(n²)

**Quick Sort:**

* Divide-and-conquer using pivot element.
* Time Complexity: O(n log n) average, O(n²) worst.

**Merge Sort:**

* Recursively divides and merges arrays.
* Time Complexity: O(n log n)

**Code:**

**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: $" + totalPrice);

}

}

**Sorter.java**

public class Sorter {

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;

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

Order[] orders = {

new Order(201, "Abi", 350.0),

new Order(202, "Balaji", 120.0),

new Order(203, "Charlie", 480.0),

new Order(204, "David", 260.0)

};

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

Sorter.bubbleSort(orders);

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

Order[] orders2 = {

new Order(201, "Abi", 350.0),

new Order(202, "Balaji", 120.0),

new Order(203, "Charlie", 480.0),

new Order(204, "David", 260.0)

};

System.out.println("\nSorting using Quick Sort:");

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

for (Order o : orders2) o.display();

}

}

**Output:**

A screen shot of a computer

AI-generated content may be incorrect.

**Time Complexity and Performance:**

**Bubble Sort:**

Time Complexity:

i)Best Case – O(n)

ii)Average Case – O(n^2)

iii)Worst Case – O(n^2)

Space Complexity – O(1)

**Quick Sort:**

Time Complexity:

i)Best Case – O(n log n)

ii)Average Case – O(n log n)

iii)Worst Case – O(n^2)

Space Complexity – O(log n)

**Quick Sort is Preferred over Bubble Sort:**

* Faster on average for large datasets.
* Uses less memory than Merge Sort.
* More scalable and practical than Bubble Sort.

**Exercise 4: Employee Management System**

**Array Representation in Memory:**

Arrays are stored in contiguous memory locations and accessed using indexing, where the address of the first element combined with an index allows quick access to any element in O(1) time using the formula: base address + index × size.

**Advantages:**

* Fast access: O(1) time for read/write by index.
* Simplicity in implementation.

**Code:**

**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);

}

}

**EmployeeManagement.java**

public class EmployeeManagement {

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

private int count = 0;

public void addEmployee(Employee emp) {

if (count < employees.length) {

employees[count++] = emp;

} else {

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

}

}

public Employee searchEmployee(int empId) {

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

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

return employees[i];

}

}

return null;

}

public boolean deleteEmployee(int empId) {

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

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

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

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

}

employees[--count] = null;

return true;

}

}

return false;

}

public void displayAll() {

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

employees[i].display();

}

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

EmployeeManagement em = new EmployeeManagement();

em.addEmployee(new Employee(1, "Abinaya", "Manager", 75000));

em.addEmployee(new Employee(2, "Balaji", "Developer", 60000));

em.addEmployee(new Employee(3, "Charlie", "Tester", 50000));

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

em.displayAll();

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

Employee emp = em.searchEmployee(2);

if (emp != null) emp.display(); else System.out.println("Not found.");

System.out.println("\nDeleting Employee ID 2:");

if (em.deleteEmployee(2)) System.out.println("Deleted successfully.");

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

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

em.displayAll();

} }

**Output:**

A screenshot of a computer

AI-generated content may be incorrect.

**Time Complexity of Operations:**

Add: O(1)

Search: O(n)

Traverse: O(n)

Delete: O(n)

**Limitations of Arrays:**

* Fixed size, not dynamically resizable without copying.
* Inefficient deletions/insertions, requires shifting.
* Not ideal for frequent updates or large-scale dynamic data.

**When to Use Arrays:**

* When the number of elements is known and fixed.
* When fast access by index is needed.
* Suitable for small or static datasets.

**Exercise 5: Task Management System**

**Types of Linked Lists:**

**Singly Linked List:**

* Each node points to the next node.
* Traversal in one direction only.

**Doubly Linked List:**

* Each node has pointers to both next and previous nodes.
* Allows bidirectional traversal.

**Code:**

**Task.java**

public 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 void display() {

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

}

}

**TaskManager.java**

public class TaskManager {

private Task head;

public void addTask(Task task) {

if (head == null) {

head = task;

} else {

Task current = head;

while (current.next != null) {

current = current.next;

}

current.next = task;

}

}

public Task searchTask(int id) {

Task current = head;

while (current != null) {

if (current.taskId == id) {

return current;

}

current = current.next;

}

return null;

}

public boolean deleteTask(int id) {

if (head == null) return false;

if (head.taskId == id) {

head = head.next;

return true;

}

Task current = head;

while (current.next != null) {

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

current.next = current.next.next;

return true;

}

current = current.next;

}

return false;

}

public void displayTasks() {

Task current = head;

while (current != null) {

current.display();

current = current.next;

}

}

} public class TaskManager {

private Task head;

public void addTask(Task task) {

if (head == null) {

head = task;

} else {

Task current = head;

while (current.next != null) {

current = current.next;

}

current.next = task;

}

}

public Task searchTask(int id) {

Task current = head;

while (current != null) {

if (current.taskId == id) {

return current;

}

current = current.next;

}

return null;

}

public boolean deleteTask(int id) {

if (head == null) return false;

if (head.taskId == id) {

head = head.next;

return true;

}

Task current = head;

while (current.next != null) {

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

current.next = current.next.next;

return true;

}

current = current.next;

}

return false;

}

public void displayTasks() {

Task current = head;

while (current != null) {

current.display();

current = current.next;

}

}

}

**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, "Develop Backend", "In Progress"));

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

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

manager.displayTasks();

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

Task found = manager.searchTask(2);

if (found != null) found.display(); else System.out.println("Not found.");

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

if (manager.deleteTask(2)) System.out.println("Task deleted.");

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

System.out.println("\nTasks After Deletion:");

manager.displayTasks();

}

}

**Output:**

A screen shot of a computer

AI-generated content may be incorrect.

**Time Complexity of Operations (Singly Linked List):**

Add (at end): O(n)

Add (at beginning): O(1)

Search: O(n)

Traverse: O(n)

Delete: O(n)

**Advantages of Linked Lists over Arrays:**

* Dynamic size: Grows and shrinks at runtime without resizing.
* Efficient insertion/deletion: Especially at beginning or middle
* No memory wastage: No need to predefine size like in arrays.

**Exercise 6: Library Management System**

**Linear Search:**

* Checks each element one by one.
* Does not require sorted data.

**Binary Search:**

* Repeatedly divides the sorted array to find the target.
* Requires sorted data.

**Code:**

**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("ID: " + bookId + ", Title: " + title + ", Author: " + author);

}

}

**LibraryManager.java**

import java.util.Arrays;

import java.util.Comparator;

public class LibraryManager {

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

for (Book book : books) {

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

return book;

}

}

return null;

}

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

int left = 0;

int right = books.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

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

if (comparison == 0) {

return books[mid];

} else if (comparison < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

public static void sortBooksByTitle(Book[] books) {

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

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

Book[] books = {

new Book(1, "Data Structures", "Narasimha Karumanchi"),

new Book(2, "Operating Systems", "Galvin"),

new Book(3, "Introduction to Algorithms", "CLRS"),

new Book(4, "Artificial Intelligence", "Stuart Russell")

};

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

Book foundLinear = LibraryManager.linearSearch(books, "Operating Systems");

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

LibraryManager.sortBooksByTitle(books);

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

Book foundBinary = LibraryManager.binarySearch(books, "Operating Systems");

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

}

}

**Output:**

A computer screen with white text

AI-generated content may be incorrect.

**Time Complexity Comparison:**

**Linear Search:**

* Best: O(1)
* Average/Worst: O(n)

**Binary Search:**

* Best: O(1)
* Average/Worst: O(log n)

**Use Linear Search:**

* When the list is unsorted or small
* When insertion/deletion happens frequently and sorting is costly

**Use Binary Search:**

* When the list is already sorted
* When you have a large static dataset

**Exercise 7: Financial Forecasting**

**Recursion Concept:**

* A function that calls itself to solve smaller subproblems.
* Simplifies problems that have repetitive or self-similar structure (e.g., financial projections, Fibonacci).
* Requires a base case to stop recursion and avoid infinite calls.

**Code:**

**FinancialForecast.java**

public class FinancialForecast {

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

if (years == 0)

return presentValue;

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

}

public static double futureValueMemo(double presentValue, double rate, int years, double[] memo) {

if (years == 0)

return presentValue;

if (memo[years] != 0)

return memo[years];

memo[years] = (1 + rate) \* futureValueMemo(presentValue, rate, years - 1, memo);

return memo[years];

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

double presentValue = 1000.0;

double rate = 0.1;

int years = 5;

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

System.out.printf("Future Value (Recursive): $%.2f%n", result);

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

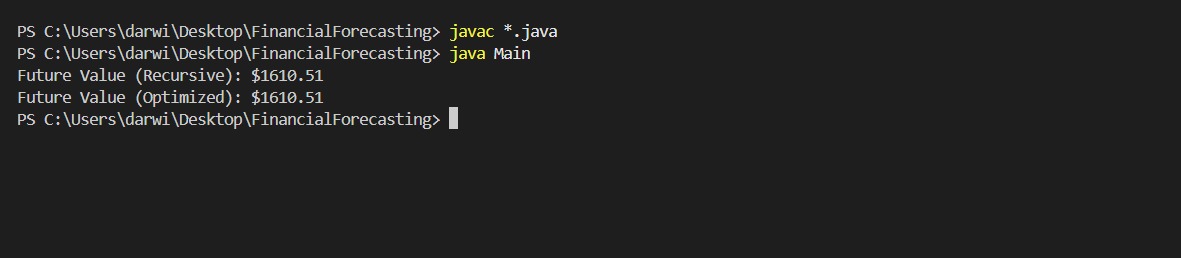
double optimizedResult = FinancialForecast.futureValueMemo(presentValue, rate, years, memo);

System.out.printf("Future Value (Optimized): $%.2f%n", optimizedResult);

}

}

**Output:**



**Recursive Algorithm:**

* Time Complexity – O(n)
* Space Complexity – O(n)

**Optimization Techniques:**

* Memoization: Store previously computed results to avoid redundant calls.
* Bottom-up (Dynamic Programming): Build solution iteratively.
* Tail Recursion: Converts recursion to iteration for optimization.