WEEK 1

DATA STRUCTURES AND ALGORITHMS

EXERCISE 1:

1.Inventory Management System:

1.1 Introduction:

The Inventory Management System is designed to efficiently manage and track products in a warehouse. This document provides a detailed overview of the system's implementation, focusing on the use of appropriate data structures to handle large inventories.

1.2 Problem Understanding:

Efficient data structures and algorithms are crucial for handling large inventories due to their impact on performance, scalability, and data integrity. Suitable data structures include ArrayList, HashMap, LinkedList, and TreeMap. For this project, HashMap is chosen due to its average O(1) time complexity for key operations.

1.3 Setup:

Project Name	InventoryManagementSystem
Package	com.inventory
IDE	Eclipse

1.4 Implementation:

Product class

```
package com.inventory;


public class Product {
    private String productId;
    private String productName;
    private int quantity;
    private 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 String getProductId() {
        return productId;
    }


    public void setProductId(String productId) {
        this.productId = productId;
```

```
}

public String getProductName() {
    return productName;
}

public void setProductName(String productName) {
    this.productName = productName;
}

public int getQuantity() {
    return quantity;
}

public void setQuantity(int quantity) {
    this.quantity = quantity;
}

public double getPrice() {
    return price;
}

public void setPrice(double price) {
    this.price = price;
    }
}
```

Inventory Class

```
   package com.inventory;

import java.util.HashMap;
import java.util.Map;

public class Inventory {
    private Map<String, Product> products;

    public Inventory() {
        this.products = new HashMap<>();
    }

    public void addProduct(Product product) {
        products.put(product.getProductId(), product);
    }

    public void updateProduct(String productId, Product
updatedProduct) {
        if (products.containsKey(productId)) {
            products.put(productId, updatedProduct);
        }
    }

    public void deleteProduct(String productId) {
        products.remove(productId);
    }
}
```

1.5 Analysis

Time Complexity

- Add Product: Insertion into a HashMap has an average time complexity of O(1).
- Update Product: Updating a value in a HashMap also has an average time complexity of O(1).
- Delete Product: Deletion from a HashMap has an average time complexity of O(1).

Optimization

- Concurrent Access: If the system requires concurrent access, consider using ConcurrentHashMap for thread-safe operations.
- Memory Usage: Regularly clean up obsolete entries to free up memory. This can be done using weak references or by periodically scanning and removing unused items.

EXERCISE 2:

2. E-commerce Platform Search Function:

2.1 Introduction:

This document details the implementation and analysis of search algorithms within an e-commerce platform, focusing on linear search and binary search methods.

2.2 Problem Understanding:

Big O notation is used to describe the upper bound of an algorithm's running time. It helps classify algorithms based on their efficiency. Linear and binary search algorithms are compared for their best, average, and worst-case scenarios.

Common Big O notations:

- **O(1):** Constant time, independent of input size.
- **O(n):** Linear time, grows proportionally with input size.
- **O(n^2):** Quadratic time, grows with the square of the input size.
- **O(log n):** Logarithmic time, grows logarithmically as input size increases.

Best, Average, and Worst-Case Scenarios:

- **Best Case:** The scenario where the algorithm performs the minimum number of operations (e.g., finding the element immediately).
- **Average Case:** The expected scenario, taking into account the probability of different inputs (e.g., the element is somewhere in the middle).
- **Worst Case:** The scenario where the algorithm performs the maximum number of operations (e.g., the element is not found or is the last element).

2.3 Setup:

Project Name: EcommercePlatform

Package: com.ecommerce

IDE: Eclipse

2.4 Implementation:

Product Class

```
package com.ecommerce;
public class Product {
    private String productId;
    private String productName;
    private String category;

    public Product(String productId, String productName, String 
category) {
        this.productId = productId;
        this.productName = productName;
        this.category = category;
    }

    public String getProductId() {
        return productId;
    }

    public void setProductId(String productId) {
        this.productId = productId;
    }

    public String getProductName() {
        return productName;
    }

    public void setProductName(String productName) {
        this.productName = productName;
    }

    public String getCategory() {
        return category;
    }

    public void setCategory(String category) {
        this.category = category;
    }
}
```

Linear Search

```
package com.ecommerce;
public class LinearSearch {
    public static Product linearSearch(Product[] products, String
productId) {
        for (Product product : products) {
            if (product.getProductId().equals(productId)) {
                return product;
            }
        }
        return null;
    }
}
```

Binary Search

```
 package com.ecommerce;
import java.util.Arrays;


public class BinarySearch {
    public static Product binarySearch(Product[] products, String
productId) {
        Arrays.sort(products, (a, b) ->
a.getProductId().compareTo(b.getProductId()));
        int left = 0;
        int right = products.length - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;
            int comparison =
products[mid].getProductId().compareTo(productId);
```

```
            if (comparison == 0) {
                return products[mid];
            } else if (comparison < 0) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return null;
    }
}
```

2.5 Analysis:

Linear Search Time Complexity:

- Best Case: O(1)
- Average Case: O(n)
- Worst Case: O(n)

Binary Search Time Complexity:

- Best Case: O(1)
- Average Case: O(log n)
- Worst Case: O(log n)

For an e-commerce platform, binary search is generally more suitable due to its logarithmic time complexity, making it more efficient for larger datasets.

EXERCISE 3:

3. Sorting Customer Orders:

3.1 Introduction:

This document details the implementation and analysis of sorting algorithms within an e-commerce platform, focusing on Bubble Sort and Quick Sort methods for sorting customer orders by total price.

3.2 Problem Understanding:

Sorting algorithms are essential for organizing data efficiently. Different sorting algorithms have varying time and space complexities, which affect their performance on large datasets. This exercise focuses on comparing Bubble Sort and Quick Sort.

1. Understand Sorting Algorithms

Bubble Sort:

- Time Complexity: O(n^2) for all cases.
- Space Complexity: O(1).

Insertion Sort:

- Time Complexity: O(n^2) average/worst, O(n) best.
- Space Complexity: O(1).

Quick Sort:

- Time Complexity: O(n log n) average, O(n^2) worst.

- Space Complexity: O(log n) average.

Merge Sort:

- Time Complexity: O(n log n) for all cases.
- Space Complexity: O(n).

3.3 Setup:

Project Name: EcommercePlatform

Package: com.ecommerce

IDE: Eclipse

3.4 Implementation:

Order class

```
package com.ecommerce;
public class Order {
private String orderId;
private String customerName;
private double totalPrice;

public Order(String orderId, String customerName, double totalPrice) {
    this.orderId = orderId;
    this.customerName = customerName;
    this.totalPrice = totalPrice;
}

public String getOrderId() {
        return orderId;
    }

    public void setOrderId(String orderId) {
        this.orderId = orderId;
    }

    public String getCustomerName() {
        return customerName;
    }

    public void setCustomerName(String customerName) {
        this.customerName = customerName;
    }

    public double getTotalPrice() {
        return totalPrice;
    }

    public void setTotalPrice(double totalPrice) {
        this.totalPrice = totalPrice;
    }
}
```

Bubble Sort

```
package com.ecommerce;
public class BubbleSort {
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].getTotalPrice() > orders[j + 1].getTotalPrice())
{
                Order temp = orders[j];
                orders[j] = orders[j + 1];
                orders[j + 1] = temp;
            }
        }
    }
}
}
```

Quick Sort

```
package com.ecommerce;
public class QuickSort {
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].getTotalPrice();
        int i = (low - 1); // Index of smaller element
        for (int j = low; j < high; j++) {
            if (orders[j].getTotalPrice() <= pivot) {
                i++;
                // Swap orders[i] and orders[j]
                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;
    }
}
```

3.5. Analysis:

Bubble Sort

- Time Complexity: O(n^2) in all cases.
- Space Complexity: O(1).
- Performance: Slow for large datasets due to its quadratic time complexity.

Quick Sort

- Time Complexity: O(n log n) on average, O(n^2) in the worst case.
- Space Complexity: O(log n) due to recursive stack space.
- Performance: Generally faster than Bubble Sort, especially for large datasets, due to its average-case time complexity of O(n log n).

Quick Sort is generally preferred over Bubble Sort for sorting large datasets due to its more efficient average-case time complexity. While Quick Sort can degrade to O(n^2) in the worst case, this can be mitigated with optimizations like randomizing the pivot element. Bubble Sort, on the other hand, consistently performs poorly on larger datasets due to its O(n^2) time complexity.

EXERCISE 4:

4. Employee Management System:

4.1 Introduction:

This document details the development and analysis of an Employee Management System. It covers the use of arrays to manage employee records efficiently, focusing on add, search, traverse, and delete operations.

4.2 Problem Understanding:

Arrays are a collection of elements stored in contiguous memory locations. Each element can be accessed using an index, which represents its position in the array.

Advantages:

- **Direct Access:** Elements can be accessed directly using their index, making retrieval operations very fast (O(1) time complexity).
- **Cache-Friendly:** Contiguous memory allocation improves cache performance, speeding up access times.
- **Simple and Efficient:** Easy to implement and manage due to their simplicity.

4.3 Setup:

Project Name: EmployeeManagementSystem

Package Name: com.company

4.4 Implementation:

Employee Class

```
 package com.company;
public class Employee {
private int employeeId;
private String name;
private String position;
private 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 int getEmployeeId() {
    return employeeId;
}

public void setEmployeeId(int employeeId) {
    this.employeeId = employeeId;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public String getPosition() {
    return position;
}

public void setPosition(String position) {
    this.position = position;
}

public double getSalary() {
    return salary;
}

public void setSalary(double salary) {
    this.salary = salary;
}
}
```

Array Operations

```
            package com.company;

import java.util.Arrays;

public class EmployeeManagement {
    private Employee[] employees;
    private int size;

    public EmployeeManagement(int capacity) {
        employees = new Employee[capacity];
        size = 0;
    }

    public boolean addEmployee(Employee employee) {
        if (size == employees.length) {
            return false; // Array is full
        }
        employees[size++] = employee;
        return true;
    }

    public Employee searchEmployee(int employeeId) {
        for (int i = 0; i < size; i++) {
            if (employees[i].getEmployeeId() == employeeId) {
                return employees[i];
            }
        }
        return null; // Employee not found
    }

    public void traverseEmployees() {
        for (int i = 0; i < size; i++) {
            System.out.println(employees[i].getEmployeeId() + ": " +
employees[i].getName());
        }
    }

    public boolean deleteEmployee(int employeeId) {
        for (int i = 0; i < size; i++) {
            if (employees[i].getEmployeeId() == employeeId) {
                // Shift elements to the left to fill the gap
                for (int j = i; j < size - 1; j++) {
                    employees[j] = employees[j + 1];
                }
                employees[--size] = null;
                return true;
            }
        }
        return false;
    }

    public static void main(String[] args) {
        EmployeeManagement management = new
EmployeeManagement(5);
        management.addEmployee(new Employee(1, "Alice",
"Manager", 75000));
        management.addEmployee(new Employee(2, "Bob",
"Developer", 60000));
        management.addEmployee(new Employee(3, "Charlie",
"Analyst", 55000));
        System.out.println("Traversing employees:");
        management.traverseEmployees();
        System.out.println("\nSearching for employee with ID 2:");
        Employee employee = management.searchEmployee(2);
        if (employee != null) {
            System.out.println("Found: " + employee.getName());
        } else {
            System.out.println("Employee not found.");
        }
        System.out.println("\nDeleting employee with ID 2:");
        if (management.deleteEmployee(2)) {
            System.out.println("Employee deleted.");
        } else {
            System.out.println("Employee not found.");
      }
        System.out.println("\nTraversing employees after deletion:");
        management.traverseEmployees();
    }
}
```

4.5. Analysis:

Time Complexity

- **Add Operation:**

  Time Complexity: O(1) (direct insertion at the end of the array if space is available).

- **Search Operation:**

  Time Complexity: O(n) (linear search).

- **Traverse Operation:**

  Time Complexity: O(n) (visiting each element).

- **Delete Operation:**

Time Complexity: O(n) (finding the element and shifting remaining elements).

Limitations of Arrays

- **Fixed Size:** Arrays have a fixed size, which means you need to specify the maximum number of elements beforehand.
- **Costly Insertions and Deletions:** Inserting or deleting elements in the middle of the array requires shifting elements, which can be costly in terms of time complexity (O(n)).

- **Better for Static Data:** Arrays are more suitable for static data where the size and elements do not change frequently.

When to Use Arrays

- When the number of elements is known and fixed.
- When fast access to elements is required using an index.
- When the dataset is relatively small or the array size can be predetermined.

EXERCISE 5:

5. Task Management System:

5.1 Introduction:

This document details the development and analysis of a Task Management System using a singly linked list. It focuses on adding, searching, traversing, and deleting tasks efficiently.

5.2 Problem Understanding:

Types of Linked Lists

Singly Linked List:

- Description: Each node contains data and a reference to the next node in the list.
- Advantages: Simple to implement and requires less memory for storing references.

Doubly Linked List:

- Description: Each node contains data, a reference to the next node, and a reference to the previous node.
- Advantages: Allows traversal in both directions and easier deletion of nodes.

5.3 Setup:

Project Name: TaskManagementSystem

Package Name: com.taskmanager

5.4 Implementation:

Task Class

```
package com.taskmanager;
public class Task {
    private int taskId;
    private String taskName;
    private String status;

    public Task(int taskId, String taskName, String status) {
        this.taskId = taskId;
        this.taskName = taskName;
        this.status = status;
    }

    public int getTaskId() {
        return taskId;
    }
    public void setTaskId(int taskId) {
        this.taskId = taskId;
    }

    public String getTaskName() {
        return taskName;
    }

    public void setTaskName(String taskName) {
        this.taskName = taskName;
    }

    public String getStatus() {
        return status;
    }

    public void setStatus(String status) {
        this.status = status;
    }
}
```

Singly Linked List Operations

```java
package com.taskmanager;

public class TaskManagement {
    private Node head;

    private static class Node {
        Task task;
        Node next;

        Node(Task task) {
            this.task = task;
            next = null;
        }
    }

    public void addTask(Task task) {
        Node newNode = new Node(task);
        if (head == null) {
            head = newNode;
        } else {
            Node current = head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = newNode;
        }
    }

    public Task searchTask(int taskId) {
        Node current = head;
        while (current != null) {
            if (current.task.getTaskId() == taskId) {
                return current.task;
            }
            current = current.next;
        }
        return null;
    }

    public void traverseTasks() {
        Node current = head;
        while (current != null) {
            System.out.println("Task ID: " + current.task.getTaskId() + ",
Task Name: " + current.task.getTaskName() + ", Status: " +
current.task.getStatus());
            current = current.next;
        }
    }
```

```
public boolean deleteTask(int taskId) {
    if (head == null) {
        return false; // List is empty
    }

    if (head.task.getTaskId() == taskId) {
        head = head.next;
        return true;
    }

    Node current = head;
    while (current.next != null) {
        if (current.next.task.getTaskId() == taskId) {
            current.next = current.next.next;
            return true;
        }
        current = current.next;
    }

    return false; // Task not found
}

public static void main(String[] args) {
        TaskManagement management = new TaskManagement();
        management.addTask(new Task(1, "Task One", "Pending"));
        management.addTask(new Task(2, "Task Two", "Completed"));
        management.addTask(new Task(3, "Task Three", "In
Progress"));
        System.out.println("Traversing tasks:");
        management.traverseTasks();
        System.out.println("\nSearching for task with ID 2:");
        Task task = management.searchTask(2);
        if (task != null) {
            System.out.println("Found: " + task.getTaskName());
        } else {
            System.out.println("Task not found.");
        }
        System.out.println("\nDeleting task with ID 2:");
        if (management.deleteTask(2)) {
            System.out.println("Task deleted.");
        } else {
            System.out.println("Task not found.");
        }
        System.out.println("\nTraversing tasks after deletion:");
        management.traverseTasks();
    }
}
```

5.5. Analysis:

Time Complexity

- Add Operation:
  - Time Complexity: O(n)
- Search Operation:
  - Time Complexity: O(n)
- Traverse Operation:
  - Time Complexity: O(n)
- Delete Operation:
  - Time Complexity: O(n)

Advantages of Linked Lists over Arrays

- Dynamic Size: Linked lists can grow and shrink in size dynamically, making them more flexible for dynamic data.
- Efficient Insertions/Deletions: Inserting or deleting nodes can be done without shifting elements, making these operations faster compared to arrays (no need to shift elements).
- Memory Utilization: Linked lists can utilize memory more efficiently as they do not require contiguous memory allocation like arrays.

EXERCISE 6:

6. Library Management System:

6.1 Introduction:

This document details the development and analysis of a Library Management System using linear and binary search algorithms. It focuses on searching for books by title efficiently within a dataset of book information.

6.2 Problem Understanding:

Types of Search Algorithms

<u>Linear Search:</u>

- Description: A straightforward search algorithm that checks each element in a list sequentially until the desired element is found or the list ends.
- Advantages: Simple to implement and works on unsorted datasets.

<u>Binary Search:</u>

- Description: An efficient search algorithm that works on sorted lists by repeatedly dividing the search interval in half until the target element is found or the interval is empty.
- Advantages: Significantly faster than linear search for large datasets, but requires the list to be sorted.

6.3 Setup:

Project Name: LibraryManagementSystem

Package Name: com.librarymanager

6.4 Implementation:

Book Class

```
package com.librarymanager;

public class Book {
    private int bookId;
    private String title;
    private String author;

    public Book(int bookId, String title, String author) {
        this.bookId = bookId;
        this.title = title;
        this.author = author;
    }

    public int getBookId() {
        return bookId;
    }

    public void setBookId(int bookId) {
        this.bookId = bookId;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }

    @Override
    public String toString() {
        return "Book{" +
                "bookId=" + bookId +
                ", title='" + title + '\'' +
                ", author='" + author + '\'' +
                '}';
    }
}
```

Linear Search Implementation

```
 package com.librarymanager;
import java.util.List;
public class LinearSearch {
    public static Book linearSearchBooksByTitle(List<Book> books,
String title) {
        for (Book book : books) {
            if (book.getTitle().equals(title)) {
                return book;
            }
        }
        return null;
    }
}
```

Binary Search Implementation

```
package com.librarymanager;
import java.util.List;
public class BinarySearch {
    public static Book binarySearchBooksByTitle(List<Book> books,
String title) {
        int left = 0, right = books.size() - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            Book midBook = books.get(mid);
            if (midBook.getTitle().equals(title)) {
                return midBook;
            }
            if (midBook.getTitle().compareTo(title) < 0) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return null;
    }
}
```

6.5 Analysis:

Time Complexity

- Add Operation:

  Time Complexity: O(n) for adding to a sorted list, O(1) for unsorted.

- Search Operation:

  Linear Search: O(n)

  Binary Search: O(log n)

- Traverse Operation:

  Time Complexity: O(n)

- Delete Operation:

Time Complexity: O(n) for both linear and binary search implementations when updating the list structure.

Advantages of Linked Lists over Arrays

- Dynamic Size: Linked lists can grow and shrink in size dynamically, making them more flexible for dynamic data.
- Efficient Insertions/Deletions: Inserting or deleting nodes can be done without shifting elements, making these operations faster compared to arrays (no need to shift elements).
- Memory Utilization: Linked lists can utilize memory more efficiently as they do not require contiguous memory allocation like arrays.

EXERCISE 7:

7. Financial Forecasting:

7.1 Introduction:

This document details the development and analysis of a financial forecasting tool that predicts future values based on past data using recursive algorithms. Recursion is a powerful technique that can simplify complex problems by breaking them down into simpler sub-problems. This document will explain the concept of recursion, implement a recursive algorithm for financial forecasting, and analyze its time complexity.

7.2 Understanding Recursive Algorithms:

Recursion is a method of solving problems where a function calls itself as a subroutine. This technique allows the function to break down a complex problem into smaller, more manageable problems of the same type. Each recursive call should move towards a base case, which is a condition that stops the recursion.

Advantages of Recursion:

- Simplifies code for problems that can be divided into similar sub-problems.
- Reduces the need for complex looping constructs.

Disadvantages of Recursion:

- Can lead to excessive memory usage if not implemented correctly.
- May result in stack overflow if the base case is not reached.

7.3 Setup:

- **Project Name**: FinancialForecasting
- **Package Name**: com.financialforecast

7.4 Implementation:

FinancialForecast class

```
package com.financialforecast;

public class FinancialForecast {

    public static double predictFutureValue(double presentValue, double growthRate, int years) {

        if (years == 0) {

            return presentValue;

        }

        return predictFutureValue(presentValue * (1 + growthRate), growthRate, years - 1);

    }


    public static void main(String[] args) {

        double presentValue = 1000.0;

        double growthRate = 0.05;

        int years = 10;

        double futureValue = predictFutureValue(presentValue, growthRate, years);

        System.out.println("Predicted future value: " + futureValue);

    }

}
```

Optimized Recursive Solution with Memoization

```
package com.financialforecast;

import java.util.HashMap;

import java.util.Map;

public class FinancialForecast {

    private static Map<Integer, Double> memo = new HashMap<>();

    public static double predictFutureValue(double presentValue,
double growthRate, int years) {

        if (years == 0) {

            return presentValue;

        }

        if (memo.containsKey(years)) {

            return memo.get(years);

        }

        double futureValue = predictFutureValue(presentValue * (1 +
growthRate), growthRate, years - 1);

        memo.put(years, futureValue);

        return futureValue;

    }


    public static void main(String[] args) {

        double presentValue = 1000.0;

        double growthRate = 0.05;

        int years = 10;
        double futureValue = predictFutureValue(presentValue, growthRate, years);
        System.out.println("Predicted future value: " + futureValue);
    }
}
```

7.5 Analysis:

Time Complexity

The time complexity of the recursive algorithm is O(n), where n is the number of years. This is because the algorithm makes one recursive call for each year until it reaches the base case.

Optimizing the Recursive Solution

To optimize the recursive solution and avoid excessive computation, we can use memoization. Memoization involves storing the results of expensive function calls and reusing them when the same inputs occur again. With memoization, the algorithm's efficiency improves significantly, making it more suitable for large datasets and deeper recursive calls.