JAVA FSE WEEK 1 SOLUTIONS

*PROBLEM 1:*

**1. Understanding the Problem**

I understand that efficient data storage and retrieval are crucial for managing large inventories.

The choice of data structures affects how quickly and efficiently I can add, update, or delete inventory items. I found that HashMap is a great choice for this scenario because it provides average O(1) time complexity for insertions, updates, and deletions, making it suitable for quick access and modifications.

**2. Setup**

I created a new project named InventoryManagementSystem

**3. Implementation**

import java.util.HashMap;

import java.util.Map;

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;

}

// Getters and Setters

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; }

}

class Inventory {

private Map<String, Product> productMap = new HashMap<>();

public void addProduct(Product product) {

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

}

public void updateProduct(String productId, Product updatedProduct) {

if (productMap.containsKey(productId)) {

productMap.put(productId, updatedProduct);

}

}

public void deleteProduct(String productId) {

productMap.remove(productId);

}

public Product getProduct(String productId) {

return productMap.get(productId);

}

}

**4. Analysis**

Time Complexity:

Add: O(1)

Update: O(1)

Delete: O(1)

Optimization: I’ve utilized HashMap, which is optimized for average O(1) operations for adding, updating, and deleting. For further optimization, I ensure hash collisions are minimized and use effective hash functions.

*PROBLEM 2:*

**1. Understanding Asymptotic Notation**

I have used Big O notation to describe the upper bound of an algorithm’s time complexity, which helps me analyze and compare the performance of different search algorithms.

Linear Search: I know it has O(n) time complexity in the worst case and O(1) in the best case if the item is at the beginning of the array.

Binary Search: I use binary search for its O(log n) time complexity, but it requires the data to be sorted.

**2. Setup**

Created a class Product with attributes for searching, such as productId, productName, and category

**3. Implementation**

import java.util.Arrays;

class Product {

private String productId;

private String productName;

private String category;

// Constructor, getters, and setters

}

class SearchUtil {

// Linear Search

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

for (Product product : products) {

if (product.getProductName().equalsIgnoreCase(searchTerm)) {

return product;

}

}

return null;

}

// Binary Search

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

int left = 0;

int right = products.length - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

int comparison = products[mid].getProductName().compareToIgnoreCase(searchTerm);

if (comparison == 0) {

return products[mid];

} else if (comparison < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

}

**4. Analysis**

Linear Search Time Complexity: O(n)

Binary Search Time Complexity: O(log n) for sorted arrays

Suitability: I prefer binary search for larger datasets if the data is sorted, as it provides faster search times compared to linear search.

*PROBLEM 3:*

**1. Understanding Sorting Algorithms**

I’ve implemented Bubble Sort with O(n^2) time complexity. It’s simple but inefficient for large datasets.

Also O(n^2) in the worst case, but it performs better with partially sorted data.I’ve opted for Quick Sort due to its O(n log n) average time complexity, which is faster for larger datasets compared to Bubble Sort.

Merge Sort: Another O(n log n) algorithm that is stable but uses additional space.

**2. Setup**

Created a class Order with attributes like orderId, customerName, and totalPrice

**3. Implementation**

import java.util.Arrays;

class Order {

private String orderId;

private String customerName;

private double totalPrice;

// Constructor, getters, and setters

}

class SortingUtil {

// Bubble Sort

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

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);

for (int j = low; j < high; j++) {

if (orders[j].getTotalPrice() <= 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;

}

}

**4. Analysis**

Bubble Sort Time Complexity: O(n^2)

Quick Sort Time Complexity: O(n log n) average, O(n^2) worst case

Quick Sort Preference: I prefer Quick Sort due to its efficiency with larger datasets compared to Bubble Sort.

*PROBLEM 4:*

**1. Understanding Array Representation**

I use arrays to store employee records because they provide O(1) time complexity for access. However, resizing and insertions/deletions can be inefficient.

**2. Setup**

I created a class Employee with attributes like employeeId, name, position, and salary

**3. Implementation**

import java.util.Arrays;

class Employee {

private String employeeId;

private String name;

private String position;

private double salary;

// Constructor, getters, and setters

}

class EmployeeManagement {

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

private int size = 0;

public void addEmployee(Employee employee) {

if (size >= employees.length) {

// Resize the array if necessary

employees = Arrays.copyOf(employees, employees.length \* 2);

}

employees[size++] = employee;

}

public Employee searchEmployee(String employeeId) {

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

if (employees[i].getEmployeeId().equals(employeeId)) {

return employees[i];

}

}

return null;

}

public void traverseEmployees() {

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

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

}

}

public void deleteEmployee(String employeeId) {

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

if (employees[i].getEmployeeId().equals(employeeId)) {

employees[i] = employees[size - 1];

employees[size - 1] = null;

size--;

return;

}

}

}

}

**4. Analysis**

Time Complexity:

Add: O(1) (average case, resizing is O(n))

Search: O(n)

Traverse: O(n)

Delete: O(n)

Limitations: I find arrays less suitable for dynamic data due to their fixed size and inefficient insertion/deletion operations. For more flexibility, I would use data structures like linked lists or ArrayLists.

*PROBLEM 5:*

1. Understand Linked Lists

Singly Linked List: Nodes point only to the next node, allowing traversal in one direction.

Doubly Linked List: Nodes point to both next and previous nodes, allowing bidirectional traversal.

**2. Setup**

I created a class Task with attributes like taskId, taskName, and status

**3. Implementation**

class Task {

private String taskId;

private String taskName;

private String status;

// Constructor, getters, and setters

}

class SinglyLinkedList {

private Node head;

private class Node {

Task task;

Node next;

Node(Task task) {

this.task = task;

this.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(String taskId) {

Node current = head;

while (current != null) {

if (current.task.getTaskId().equals(taskId)) {

return current.task;

}

current = current.next;

}

return null;

}

public void traverseTasks() {

Node current = head;

while (current != null) {

System.out.println(current.task.getTaskName());

current = current.next;

}

}

public void deleteTask(String taskId) {

if (head == null) return;

if (head.task.getTaskId().equals(taskId)) {

head = head.next;

return;

}

Node current = head;

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

current = current.next;

}

if (current.next != null) {

current.next = current.next.next;

}

}

}

**4. Analysis**

Time Complexity:

Add: O(n) (if appending at the end)

Search: O(n)

Traverse: O(n)

Delete: O(n)

Advantages: Linked lists are beneficial for dynamic data where the size changes frequently, as they offer efficient insertions and deletions compared to arrays.

*PROBLEM 6:*

**1. Understand Search Algorithms**

Linear Search: Simple and straightforward with O(n) time complexity in the worst case.

Binary Search: Efficient with O(log n) time complexity, but requires a sorted array.

**2. Setup**

I created a class Book with attributes like bookId, title, and author

**3. Implementation**

import java.util.Arrays;

class Book {

private String bookId;

private String title;

private String author;

// Constructor, getters, and setters

}

class LibraryUtil {

// Linear Search

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

for (Book book : books) {

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

return book;

}

}

return null;

}

// Binary Search

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

int left = 0;

int right = books.length - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

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

if (comparison == 0) {

return books[mid];

} else if (comparison < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

}

**4. Analysis**

Linear Search Time Complexity: O(n)

Binary Search Time Complexity: O(log n) for sorted arrays

Use Case: Binary search is preferred for larger datasets when the data is sorted due to its faster search times compared to linear search.

*PROBLEM 7:*

**1. Understanding Recursive Algorithms**

Recursion allows solving problems by breaking them down into smaller sub-problems. It simplifies problem-solving but can lead to deep recursive calls and stack overflow if not optimized.

**2. Setup**

Implemented a method to calculate the future value using recursion

**3. Implementation**

public class FinancialForecasting {

// Recursive approach to calculate future value

public static double calculateFutureValue(double initialValue, double growthRate, int years) {

if (years <= 0) {

return initialValue;

}

return calculateFutureValue(initialValue \* (1 + growthRate), growthRate, years - 1);

}

public static void main(String[] args) {

double initialValue = 1000;

double growthRate = 0.05; // 5%

int years = 10;

double futureValue = calculateFutureValue(initialValue, growthRate, years);

System.out.println("Future Value: " + futureValue);

}

}

**4. Analysis**

Time Complexity: O(n) due to recursive calls

Optimization: To avoid excessive computation, I could use memoization or iterative approaches to handle large inputs more efficiently and prevent stack overflow issues.