Exercise 1: Inventory Management System

Understanding the Problem

Why Data Structures and Algorithms are needed:

Data structures and algorithms lie at the core of any efficient handling and manipulation of large inventories. They enable optimized storage, quick retrieval, and effective manipulation of data so that inventory records remain updated and accurate.

Types of Data Structures Applicable:

ArrayList: For maintaining a list of products dynamically, ordered in nature.

HashMap: Best for fast access, insert and delete by key, like the productId.

Setup

Create a project in your development environment.

Implementation

class Product {

String productId;

String productName;

int quantity;

double price;

public Product(String productId, String productName, int quantity, double price) {

this.productId =

ThisHLT.productName = productName;

this.quantity = quantity;

this.price = price;

}

}

oler

import java.util.HashMap;

class Inventory {

HashMap<String, Product> products = new HashMap<>();

public void addProduct(Product product) {

products.put(product.productId,

It keeps track of products through a HashMap.

```java

import java.util.HashMap;

public class ProductInventory {

HashMap<String, Product> products;

public ProductInventory() {

this.products = new HashMap<>();

}

public void updateProduct(Product product) {

products.put(product.productId, product);

}

public void deleteProduct(String productId) {

products.remove(productId);

}

}

Analysis

Time Complexity:

Add Product: O(1) for HashMap

Update Product: O(1) for HashMap

Delete Product: O(1) for HashMap

Optimization Discussion :

It will provide an efficient O(1) average time complexity for add, update, and delete operations using HashMap. If the inventory is very large, then ensuring that hash functions are well distributed maintains performance.

Exercise 2: E-commerce Platform Search Function

**Setup**

**Class Definition:**

java

Copy code

class Product {

String productId;

String productName;

String category;

// Constructor

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

}

**Implementation**

**Linear Search:**

java

Copy code

public Product linearSearch(Product[] products, String targetName) {

for (Product product : products) {

if (product.productName.equals(targetName)) {

return product;

}

}

return null;

}

**Binary Search (Assuming sorted array):**

java

Copy code

import java.util.Arrays;

public Product binarySearch(Product[] products, String targetName) {

int left = 0;

int right = products.length - 1;

while (left <= right) {

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

int cmp = products[mid].productName.compareTo(targetName);

if (cmp == 0) {

return products[mid];

} else if (cmp < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

**Analysis**

**Time Complexity:**

* **Linear Search:** O(n)
* **Binary Search:** O(log n)

**Suitability Discussion:** Binary search is more efficient than linear search for large, sorted datasets due to its O(log n) complexity. However, if the dataset is not sorted, linear search is necessary unless the data can be sorted beforehand.

**Exercise 3: Sorting Customer Orders**

**Understand Sorting Algorithms**

**Sorting Algorithms:**

* **Bubble Sort:** Simple but inefficient for large datasets. O(n^2)
* **Quick Sort:** Efficient and commonly used. O(n log n) average case.

**Setup**

**Class Definition:**

java

Copy code

class Order {

String orderId;

String customerName;

double totalPrice;

// Constructor

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

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

}

**Implementation**

**Bubble Sort:**

java

Copy code

public 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) {

// swap orders[j] and orders[j+1]

Order temp = orders[j];

orders[j] = orders[j+1];

orders[j+1] = temp;

}

}

}

}

**Quick Sort:**

java

Copy code

public 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 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++;

// swap orders[i] and orders[j]

Order temp = orders[i];

orders[i] = orders[j];

orders[j] = temp;

}

}

// swap orders[i+1] and orders[high] (pivot)

Order temp = orders[i+1];

orders[i+1] = orders[high];

orders[high] = temp;

return i+1;

}

**Analysis**

**Time Complexity:**

* **Bubble Sort:** O(n^2)
* **Quick Sort:** O(n log n) average case

**Discussion:** Quick Sort is generally preferred over Bubble Sort due to its better average-case performance. Bubble Sort can be used for small datasets or nearly sorted data.

**Exercise 4: Employee Management System**

**Understand Array Representation**

**Array Representation:** Arrays are contiguous memory locations that allow constant time access. They are simple and efficient for static datasets.

**Setup**

**Class Definition:**

java

Copy code

class Employee {

String employeeId;

String name;

String position;

double salary;

// Constructor

public Employee(String employeeId, String name, String position, double salary) {

this.employeeId = employeeId;

this.name = name;

this.position = position;

this.salary = salary;

}

}

**Implementation**

**Array Operations:**

java

Copy code

class EmployeeManagement {

Employee[] employees;

int size;

// Constructor

public EmployeeManagement(int capacity) {

employees = new Employee[capacity];

size = 0;

}

// Add employee

public void addEmployee(Employee employee) {

if (size < employees.length) {

employees[size++] = employee;

} else {

// Resize array if needed

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

employees[size++] = employee;

}

}

// Search employee

public Employee searchEmployee(String employeeId) {

for (Employee employee : employees) {

if (employee != null && employee.employeeId.equals(employeeId)) {

return employee;

}

}

return null;

}

// Traverse employees

public void traverseEmployees() {

for (Employee employee : employees) {

if (employee != null) {

System.out.println(employee);

}

}

}

// Delete employee

public void deleteEmployee(String employeeId) {

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

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

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

employees[size] = null;

break;

}

}

}

}

**Analysis**

**Time Complexity:**

* **Add:** O(1) amortized if resizing
* **Search:** O(n)
* **Traverse:** O(n)
* **Delete:** O(n)

**Limitations:** Arrays have fixed size and resizing can be costly. They are not ideal for dynamic datasets where frequent insertions and deletions are required.

**Exercise 5: Task Management System**

**Understand Linked Lists**

**Types of Linked Lists:**

* **Singly Linked List:** Nodes have a single pointer to the next node.
* **Doubly Linked List:** Nodes have pointers to both next and previous nodes.

**Setup**

**Class Definition:**

class Task {

String taskId;

String taskName;

String status;

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

this.taskId = taskId;

this.taskName = taskName;

this.status = status;

}

}

**Implementation**

**Singly Linked List Operations:**

class SinglyLinkedList {

class Node {

Task task;

Node next;

Node(Task task) {

this.task = task;

}

}

**Binary Search**

import java.util.Arrays;

public Product binarySearch(Product[] products, String targetName) {

int left = 0;

int right = products.length - 1;

while (left <= right) {

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

int cmp = products[mid].productName.compareTo(targetName);

if (cmp == 0) {

return products[mid];

} else if (cmp < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

**Analysis**

**Time Complexity:**

* **Linear Search:** O(n)
* **Binary Search:** O(log n)

Exercise 3: Sorting Customer Orders

**Implementation**

**Bubble Sort:**

public 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;

}

}

}

}

**Quick Sort:**

public 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 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++;

// 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;

}

**Analysis**

**Time Complexity:**

* **Bubble Sort:** O(n^2)
* **Quick Sort:** O(n log n)

Top of Form

Bottom of Form