ASSIGNMENT-1

Data Structures & Algorithms

**Exercise 1: Inventory Management System**

**Explain why data structures and algorithms are essential in handling large inventories**

Data structures and algorithms are crucial for efficiently managing large inventories due to their ability to organize, access, and manipulate data effectively.

**Importance of Data Structures and Algorithms in Inventory Management**

**1. Efficiency in Data Handling:**

- Large inventories can contain thousands of items, making it essential to have a system that can quickly add, update, and retrieve product information. Efficient data structures minimize the time complexity of these operations, allowing for faster processing and better user experience.

**2. Scalability:**

- As inventory grows, the system must handle increased data without significant performance degradation. Algorithms that are optimized for scalability can manage larger datasets and support operations across distributed systems, ensuring that the inventory system remains responsive and efficient.

**3. Optimized Resource Utilization:**

- Properly chosen data structures can lead to better memory management and resource allocation. This is particularly important in environments with limited resources or when operating in cloud-based systems where costs can be tied to resource usage.

**4. Complex Data Analysis:**

- Data structures facilitate complex queries and data analysis, such as identifying trends in inventory turnover or predicting stock requirements. Algorithms can help in sorting and searching through large datasets to extract meaningful insights.

**5. Improved Decision-Making:**

- By efficiently organizing and processing inventory data, businesses can make informed decisions regarding stock levels, pricing strategies, and supply chain management, ultimately leading to enhanced operational performance.

**Discuss the types of data structures suitable for this problem.**

**Suitable Data Structures for Inventory Management:**

1**. HashMap:**

- A `HashMap` is ideal for storing inventory items where each product can be accessed via a unique key (e.g., product ID). This allows for average O(1) time complexity for add, update, and delete operations, making it highly efficient for inventory management.

2. **ArrayList:**

- An `ArrayList` can be used for scenarios where the order of items matters, or when the number of items is relatively small and known in advance. It allows for dynamic resizing and provides O(1) access time for indexed elements, though insertions and deletions can be O(n) in the worst case.

3. **LinkedList:**

- A `LinkedList` can be useful when frequent insertions and deletions are expected, as these operations are O(1) if the position is known. However, it has O(n) time complexity for access operations, which may not be ideal for all inventory systems.

4. **Tree Structures:**

- Balanced trees (like AVL or Red-Black Trees) can be employed for sorted inventory management, allowing for efficient searching, insertion, and deletion operations (O(log n)). This is beneficial when maintaining a sorted list of products is necessary.

5. **Priority Queue:**

- If certain products need to be prioritized (e.g., fast-moving items), a priority queue can be utilized. This structure allows for efficient retrieval of the highest (or lowest) priority item, which can be useful in managing stock levels dynamically.

6. **Graph Structures:**

- In complex inventory systems where relationships between products (like bundles or dependencies) are important, graph structures can be used. They allow for the representation of interconnected items and can facilitate complex queries regarding inventory relationships.

**2.CODE:**

//1.class Product with attributes like productId, productName, quantity, and price.

import java.util.HashMap;

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 String getProductName() {

return productName;

}

public int getQuantity() {

return quantity;

}

public double getPrice() {

return price;

}

public void setQuantity(int quantity) {

this.quantity = quantity;

}

public void setPrice(double price) {

this.price = price;

}

public String toString() {

return "Product{" +

"productId='" + productId + '\'' +

", productName='" + productName + '\'' +

", quantity=" + quantity +

", price=" + price +

'}';

}

}

//Choose an appropriate data structure to store the products (e.g., ArrayList, HashMap).

//Implement methods to add, update, and delete products from the inventory.

class InventoryManager {

private HashMap<String, Product> inventory;

public InventoryManager() {

this.inventory = new HashMap<>();

}

// Method to add a product

public void addProduct(Product product) {

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

}

// Method to update a product

public void updateProduct(String productId, int quantity, double price) {

Product product = inventory.get(productId);

if (product != null) {

product.setQuantity(quantity);

product.setPrice(price);

} else {

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

}

}

// Method to delete a product

public void deleteProduct(String productId) {

inventory.remove(productId);

}

// Method to display all products

public void displayProducts() {

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

System.out.println(product);

}

}

}

class Main {

public static void main(String[] args) {

InventoryManager inventoryManager = new InventoryManager();

// Adding products

Product product1 = new Product("P001", "Air pods", 10, 99.00);

Product product2 = new Product("P002", "Smartphone", 20, 499.99);

inventoryManager.addProduct(product1);

inventoryManager.addProduct(product2);

// Displaying products

System.out.println("Current Inventory:");

inventoryManager.displayProducts();

// Updating a product

inventoryManager.updateProduct("P001", 8, 899.99);

// Displaying products after update

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

inventoryManager.displayProducts();

// Deleting a product

inventoryManager.deleteProduct("P002");

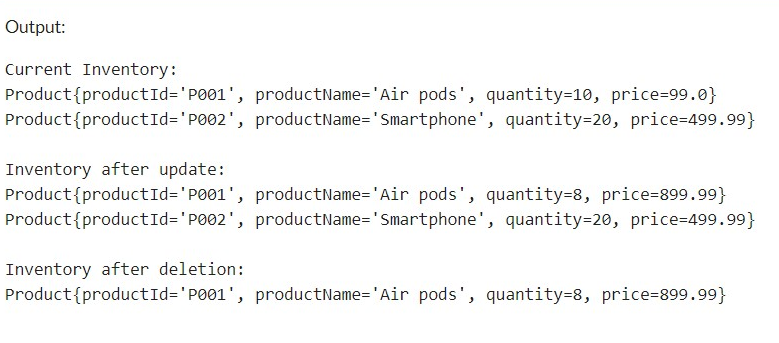
// Displaying products after deletion

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

inventoryManager.displayProducts();

}

}



**Time complexties:**

**Time Complexity Analysis:**

1. **Add Product:**
   * 1. Time Complexity: O(1)
     2. Inserting a product into a HashMap is O(1) due to the constant time complexity of hash-based data structures.
2. **Update Product:**
3. Time Complexity: O(1)
4. Updating a product in a HashMap is also O(1) since it involves accessing the element by key and replacing the value.
5. **Delete Product:**
   1. Time Complexity: O(1)
   2. Removing a product from a HashMap is O(1) as it involves finding the element by key and deleting it.

**Optimization:**

To optimize the above inventory management system, we should ensure that the `HashMap` is sized properly to avoid frequent rehashing, and also adjust the load factor for a balance between performance and memory usage.

For concurrent access, we can also consider using `ConcurrentHashMap` to prevent the chances of occurrence of synchronization issues.

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

**Explain Big O notation and how it helps in analyzing algorithms.**

**Big O Notation:**

Big O notation is a mathematical concept used to describe the performance or complexity of algorithms, particularly in terms of time and space. It provides a standardized way to analyze and compare the efficiency of different algorithms, focusing primarily on their worst-case scenarios.

**Importance**

- **Performance Comparison:** Big O notation allows developers to compare the efficiency of different algorithms, helping them choose the most suitable one for a specific problem.

- **Focus on Growth Rates:** It emphasizes how the algorithm's performance changes as the input size increases, rather than the exact number of operations, which can vary based on hardware and implementation.

- **Worst-case Analysis:** While it can also describe average and best-case scenarios, Big O primarily focuses on the worst-case performance, providing a conservative estimate of an algorithm's efficiency.

**Describe the best, average, and worst-case scenarios for search operations.**

**Scenarios for Search Operations**

When analyzing search operations, it is essential to consider different scenarios based on the input data and the algorithm used. The three primary scenarios are:

**1. Best-case Scenario:**

**-** This scenario describes the minimum time required for an algorithm to complete its operation. For a search operation, the best case occurs when the desired element is found immediately, typically at the first position in the dataset.

**- Example:** In a linear search, if the target element is the first element in the list, the time complexity is O(1) .

**2. Average-case Scenario:**

**-** The average-case scenario represents the expected time complexity for a search operation, assuming a uniform distribution of elements. It considers all possible positions of the target element and averages the time taken across these positions.

**- Example:** In a linear search, if the target element is equally likely to be found at any position, the average time complexity is O(n) , where n is the number of elements in the list.

**3. Worst-case Scenario:**

**-** The worst-case scenario describes the maximum time required for an algorithm to complete its operation. For a search operation, this occurs when the desired element is either not present in the dataset or is located at the last position.

**- Example:** In a linear search, if the target element is not found, the algorithm must check all elements, resulting in a time complexity of O(n) .

**Summary of Search Operation Complexities**

**- Best Case:** O(1) (element found immediately)

**- Average Case:** O(n) (element found after checking half the list)

**- Worst Case:** O(n) (element not found or found at the end)

2.**CODE:**

import java.util.Arrays;

//Product Class Implementation

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;

}

// Getters

public String getProductId() {

return productId;

}

public String getProductName() {

return productName;

}

public String getCategory() {

return category;

}

public String toString() {

return "Product{" +

"productId='" + productId + '\'' +

", productName='" + productName + '\'' +

", category='" + category + '\'' +

'}';

}

}

//Search Algorithms Implementation

class SearchAlgorithms {

// Linear search implementation

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

for (Product product : products) {

if (product.getProductId().equals(targetId)) {

return product; // Product found

}

}

return null; // Product not found

}

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

int left = 0;

int right = products.length - 1;

while (left <= right) {

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

if (products[mid].getProductId().equals(targetId)) {

return products[mid]; // Product found

}

if (products[mid].getProductId().compareTo(targetId) < 0) {

left = mid + 1; // Search in the right half

} else {

right = mid - 1; // Search in the left half

}

}

return null; // Product not found

}

}

//Main Class to Demonstrate Search

class Main {

public static void main(String[] args) {

Product[] products = {

new Product("P001", "Laptop", "Electronics"),

new Product("P002", "Smartphone", "Electronics"),

new Product("P003", "Tablet", "Electronics"),

new Product("P004", "Headphones", "Accessories"),

new Product("P005", "Smartwatch", "Wearables")

};

// Sorting the array for binary search

Arrays.sort(products, (a, b) -> a.getProductId().compareTo(b.getProductId()));

// Searching for a product using linear search

String targetId = "P003";

Product foundProductLinear = SearchAlgorithms.linearSearch(products, targetId);

System.out.println("Linear Search Result: " + (foundProductLinear != null ? foundProductLinear : "Not Found"));

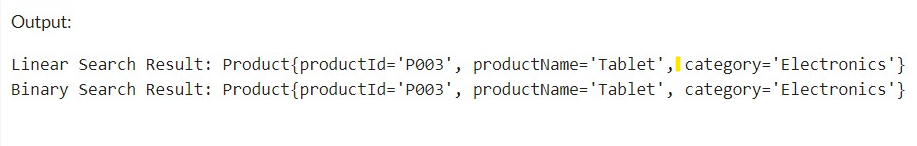
// Searching for a product using binary search

Product foundProductBinary = SearchAlgorithms.binarySearch(products, targetId);

System.out.println("Binary Search Result: " + (foundProductBinary != null ? foundProductBinary : "Not Found"));

}

}



**Compare the time complexity of linear and binary search algorithms.**

**Discuss which algorithm is more suitable for your platform and why.**

**Time Complexity:**

**Linear Search:**

1. Best Case: O(1) (when the element is at the beginning)
2. Average Case: O(n)
3. Worst Case: O(n)

**Binary Search:**

1. Best Case: O(1) (when the element is at the middle)
2. Average Case: O(log n)
3. Worst Case: O(log n)

**Suitable Algorithm for this Platform:**

Binary search is more suitable for the e-commerce platform because it has a lower time complexity of O(log n) compared to linear search O(n) for large datasets. However, it requires the dataset to be sorted. If the dataset is not sorted or frequently updated, linear search might be simpler to implement initially but less efficient for larger datasets.

**Exercise 3: Sorting Customer Orders**

**Explain different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort).**

**Bubble Sort**

Bubble Sort is a simple comparison-based sorting algorithm. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted.

* **Time Complexity**: O(n^2) in the average and worst case, O(n) in the best case
* **Space Complexity**: O(1)
* **Stability**: Stable

**Insertion Sort**

Insertion Sort builds the final sorted array one item at a time. It takes each element from the input and inserts it into the correct position within the already sorted part of the array.

* **Time Complexity**: O(n^2) in the average and worst case, O(n) in the best case
* **Space Complexity**: O(1)
* **Stability**: Stable

**Quick Sort**

Quick Sort is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

* **Time Complexity**: O(n log n) on average, O(n^2) in the worst case.
* **Space Complexity**: O(log n)
* **Stability**: Not stable

**Merge Sort**

Merge Sort is also a divide-and-conquer algorithm. It divides the array into two halves, recursively sorts them, and then merges the two sorted halves.

* **Time Complexity**: O(n log n)
* **Space Complexity**: O(n)
* **Stability**: Stable

2.**CODE:**

//Order Class Implementation

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;

}

// Getters

public String getOrderId() {

return orderId;

}

public String getCustomerName() {

return customerName;

}

public double getTotalPrice() {

return totalPrice;

}

public String toString() {

return "Order{" +

"orderId='" + orderId + '\'' +

", customerName='" + customerName + '\'' +

", totalPrice=" + totalPrice +

'}';

}

}

//Sorting Algorithms Implementation

class OrderSorting {

// Bubble Sort implementation

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()) {

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

Order temp = orders[j];

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

orders[j + 1] = temp;

}

}

}

}

// Quick Sort implementation

public static void quickSort(Order[] orders, int low, int high) {

if (low < high) {

int pi = partition(orders, low, high);

quickSort(orders, low, pi - 1); // Sort elements before partition

quickSort(orders, pi + 1, high); // Sort elements after partition

}

}

private static int partition(Order[] orders, int low, int high) {

double pivot = orders[high].getTotalPrice(); // Pivot

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;

}

}

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

Order temp = orders[i + 1];

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

orders[high] = temp;

return i + 1;

}

}

//Main Class

class Main {

public static void main(String[] args) {

Order[] orders = {

new Order("O001", "Alice", 250.75),

new Order("O002", "Bob", 150.50),

new Order("O003", "Charlie", 300.00),

new Order("O004", "David", 100.25),

new Order("O005", "Eve", 200.00)

};

// Sorting using Bubble Sort

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

OrderSorting.bubbleSort(orders);

for (Order order : orders) {

System.out.println(order);

}

// Resetting the orders for Quick Sort demonstration

orders = new Order[]{

new Order("O001", "Alice", 250.75),

new Order("O002", "Bob", 150.50),

new Order("O003", "Charlie", 300.00),

new Order("O004", "David", 100.25),

new Order("O005", "Eve", 200.00)

};

// Sorting using Quick Sort

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

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

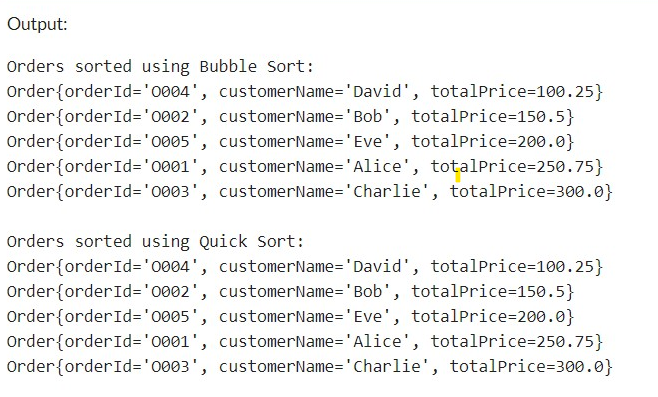
for (Order order : orders) {

System.out.println(order);

}

}

}



**Compare the performance (time complexity) of Bubble Sort and Quick Sort.Discuss why Quick Sort is generally preferred over Bubble Sort.**

**Time Complexity Comparison**

**Bubble Sort**

**- Best Case: O(n)**

**-** This occurs when the array is already sorted. The algorithm will make a single pass through the array, performing no swaps.

**- Average Case: O(n²)**

**-** On average, Bubble Sort will need to make about n/2 comparisons for each of the n elements, leading to a quadratic time complexity.

**- Worst Case: O(n²)**

**-** This occurs when the array is sorted in reverse order. The algorithm will need to make n passes, with each pass requiring n comparisons.

**Quick Sort**

**- Best Case: O(n log n)**

**-** This occurs when the pivot divides the array into two equal halves at each recursive step, leading to a balanced partitioning.

**- Average Case: O(n log n)**

**-** On average, Quick Sort performs well due to its divide-and-conquer strategy, resulting in logarithmic depth of recursion and linear work at each level.

**- Worst Case: O(n²)**

**-** This occurs when the pivot is the smallest or largest element repeatedly (e.g., when the array is already sorted or nearly sorted), leading to unbalanced partitions. However, this can often be mitigated with techniques such as choosing a random pivot or using the median-of-three method.

**Why Quick Sort is Generally Preferred Over Bubble Sort**

**1. Efficiency:**

**- Time Complexity:** Quick Sort has a much better average and worst-case time complexity (O(n log n)) compared to Bubble Sort (O(n²)). This makes Quick Sort significantly faster for larger datasets.

**- Divide-and-Conquer:** Quick Sort's divide-and-conquer approach allows it to efficiently sort large datasets by reducing the problem size at each step.

**2. Performance on Large Datasets:**

- Quick Sort is designed to handle large datasets efficiently. Bubble Sort, with its quadratic time complexity, becomes impractical for large arrays, as the number of comparisons and swaps increases dramatically.

Quick Sort is generally preferred over Bubble Sort due to its superior time complexity, efficiency with large datasets, and better cache performance. While Bubble Sort is simple and easy to implement, its inefficiency makes it unsuitable for practical applications involving large amounts of data. Quick Sort's versatility and performance make it a more robust choice for sorting tasks in most scenarios.

**Exercise 4: Employee Management System**

Arrays are a fundamental data structure in programming that store a collection of elements of the same data type. They are represented in memory as a contiguous block of memory locations, where each element is stored sequentially based on its index. Here's a more detailed explanation of how arrays are represented in memory and their advantages:

**#Representation in Memory**

**1. Contiguous Memory Locations:**

**-** An array is stored in a contiguous block of memory locations.

- The memory address of the first element is known as the base address of the array.

- Each subsequent element is stored in the next consecutive memory location.

**2. Memory Allocation:**

**-** In most programming languages, the size of an array must be specified when it is declared.

- The compiler allocates a contiguous block of memory locations based on the size of the array and the size of each element.

- For example, if an array of integers is declared with a size of 10, the compiler will allocate 40 bytes of memory (assuming each integer occupies 4 bytes).

**3. Accessing Elements:**

- Elements in an array are accessed using their index.

- The memory address of an element can be calculated using the base address and the index of the element.

- The formula for calculating the memory address is: `base\_address + (index \* size\_of\_each\_element)`

**Advantages of Arrays**

**1. Random Access:**

**-** Arrays allow for constant-time access to any element using its index.

- This means that accessing an element at a specific index takes the same amount of time regardless of the index value or the size of the array.

**2. Simplicity and Efficiency:**

- Arrays are simple to implement and use in programming.

- They provide efficient storage and access to data, especially when the size of the data is known in advance.

However, arrays also have some limitations, such as a fixed size (which can lead to memory wastage or the need for dynamic resizing) and the requirement that all elements be of the same data type. To overcome these limitations, other data structures like linked lists, hash tables, and trees are often used in addition to arrays.

**2.CODE:**

//Employee Class Implementation

class Employee {

private String employeeId;

private String name;

private String position;

private double salary;

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

this.employeeId = employeeId;

this.name = name;

this.position = position;

this.salary = salary;

}

// Getters

public String getEmployeeId() {

return employeeId;

}

public String getName() {

return name;

}

public String getPosition() {

return position;

}

public double getSalary() {

return salary;

}

public String toString() {

return "Employee{" +

"employeeId='" + employeeId + '\'' +

", name='" + name + '\'' +

", position='" + position + '\'' +

", salary=" + salary +

'}';

}

}

//Employee Management Class

class EmployeeManagement {

private Employee[] employees;

private int size;

private int capacity;

public EmployeeManagement(int capacity) {

this.capacity = capacity;

this.employees = new Employee[capacity];

this.size = 0;

}

// Method to add an employee

public void addEmployee(Employee employee) {

if (size < capacity) {

employees[size] = employee;

size++;

System.out.println("Employee added: " + employee);

} else {

System.out.println("Employee array is full. Cannot add more employees.");

}

}

// Method to search for an employee by employeeId

public Employee searchEmployee(String employeeId) {

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

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

return employees[i]; // Employee found

}

}

return null; // Employee not found

}

// Method to traverse and display all employees

public void traverseEmployees() {

if (size == 0) {

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

return;

}

System.out.println("Employee List:");

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

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

}

}

// Method to delete an employee by employeeId

public void deleteEmployee(String employeeId) {

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

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

// Shift elements to the left

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

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

}

employees[size - 1] = null; // Clear the last element

size--;

System.out.println("Employee deleted: " + employeeId);

return;

}

}

System.out.println("Employee not found: " + employeeId);

}

}

class Main {

public static void main(String[] args) {

EmployeeManagement employeeManagement = new EmployeeManagement(5);

// Adding employees

employeeManagement.addEmployee(new Employee("E001", "Alice Smith", "Developer", 60000));

employeeManagement.addEmployee(new Employee("E002", "Bob Johnson", "Manager", 80000));

employeeManagement.addEmployee(new Employee("E003", "Charlie Brown", "Designer", 55000));

// Traversing employees

employeeManagement.traverseEmployees();

// Searching for an employee

Employee foundEmployee = employeeManagement.searchEmployee("E002");

System.out.println("Found Employee: " + (foundEmployee != null ? foundEmployee : "Not Found"));

// Deleting an employee

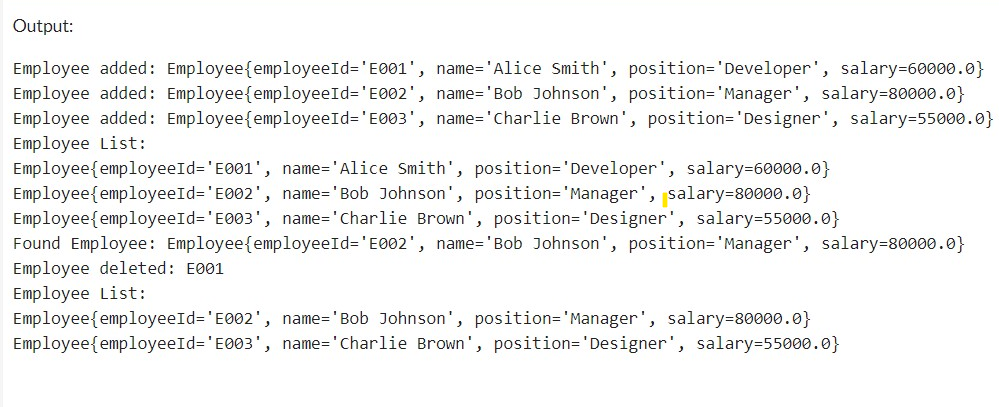
employeeManagement.deleteEmployee("E001");

// Traversing employees after deletion

employeeManagement.traverseEmployees();

}

}



**Analysis:**

**Time Complexity of Operations**

* Add Employee:

Time Complexity: O(1)

* Search Employee by ID:

Time Complexity: O(n)

* Traverse Employees:

Time Complexity: O(n)

* Delete Employee by ID:
  + Time Complexity: O(n)

**Limitations of Arrays and When to Use Them**

* **Fixed Size**: Arrays have a fixed size, making them unsuitable when the number of elements is unknown or changes frequently.
* **Inefficient for Frequent Insertions/Deletions:** Operations like insertion and deletion are costly (O(n)) compared to dynamic data structures like ArrayList or LinkedList.
* **When to Use Arrays:**
* When the number of elements is known and fixed.
* For applications requiring fast access to elements by index.
* When memory efficiency and performance of access are critical.
* **Exercise 5: Task Management System**

Linked lists are a fundamental data structure used in computer science to store collections of elements. Unlike arrays, linked lists do not require contiguous memory allocation, allowing for efficient insertion and deletion of elements. There are several types of linked lists, but the two most common types are Singly Linked Lists and Doubly Linked Lists.

**1. Singly Linked List**

A Singly Linked List consists of a sequence of nodes, where each node contains two components:

- Data: The value or information stored in the node.

- Next Pointer: A reference (or pointer) to the next node in the sequence.

The list is accessed starting from a head node, and the last node's next pointer is typically set to null, indicating the end of the list.

Advantages

- Dynamic Size: Unlike arrays, linked lists can grow and shrink in size dynamically, allowing for efficient memory usage.

- Efficient Insertions/Deletions: Inserting or deleting nodes can be done in O(1) time if the position is known (e.g., at the head or tail), as it only requires updating pointers.

Disadvantages

- Memory Overhead: Each node requires additional memory for the pointer to the next node.

- Sequential Access: Accessing elements is not as efficient as arrays, as it requires traversing the list from the head to the desired node, resulting in O(n) time complexity for access operations.

**2. Doubly Linked List**

A Doubly Linked List is similar to a singly linked list but has an additional pointer in each node:

- Data: The value or information stored in the node.

- Next Pointer: A reference to the next node in the sequence.

- Previous Pointer: A reference to the previous node in the sequence.

This allows traversal in both directions—from the head to the tail and from the tail to the head.

Advantages

- Bidirectional Traversal: Nodes can be traversed in both directions, making operations like searching and deletion more flexible.

- Easier Deletion: Deleting a node is easier because you can access the previous node directly without needing to traverse the list from the head.

Disadvantages

- Increased Memory Usage: Each node requires additional memory for the previous pointer, leading to higher memory overhead compared to singly linked lists.

- More Complex Implementation: The implementation of a doubly linked list is more complex due to the need to manage two pointers.

In conclusion, both singly linked lists and doubly linked lists have their own strengths and weaknesses. The choice between them depends on the specific requirements of the application, such as the need for bidirectional traversal, memory constraints, and the frequency of insertion and deletion operations.

2.**CODE:**

class Task {

private String taskId;

private String taskName;

private String status;

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

this.taskId = taskId;

this.taskName = taskName;

this.status = status;

}

// Getters

public String getTaskId() {

return taskId;

}

public String getTaskName() {

return taskName;

}

public String getStatus() {

return status;

}

public String toString() {

return "Task{" +

"taskId='" + taskId + '\'' +

", taskName='" + taskName + '\'' +

", status='" + status + '\'' +

'}';

}

}

class Node {

Task task;

Node next;

public Node(Task task) {

this.task = task;

this.next = null;

}

}

class TaskLinkedList {

private Node head;

// Method to add a task to the end of the linked list

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;

}

System.out.println("Task added: " + task);

}

// Method to search for a task by taskId

public Task searchTask(String taskId) {

Node current = head;

while (current != null) {

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

return current.task; // Task found

}

current = current.next;

}

return null; // Task not found

}

// Method to traverse and display all tasks

public void traverseTasks() {

if (head == null) {

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

return;

}

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

Node current = head;

while (current != null) {

System.out.println(current.task);

current = current.next;

}

}

// Method to delete a task by taskId

public void deleteTask(String taskId) {

if (head == null) {

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

return;

}

// If the task to be deleted is the head

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

head = head.next;

System.out.println("Task deleted: " + taskId);

return;

}

Node current = head;

while (current.next != null) {

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

current.next = current.next.next; // Bypass the node to delete it

System.out.println("Task deleted: " + taskId);

return;

}

current = current.next;

}

System.out.println("Task not found: " + taskId);

}

}

class Main {

public static void main(String[] args) {

TaskLinkedList taskList = new TaskLinkedList();

// Adding tasks

taskList.addTask(new Task("T001", "Design Database", "In Progress"));

taskList.addTask(new Task("T002", "Implement API", "Not Started"));

taskList.addTask(new Task("T003", "Write Documentation", "Completed"));

// Traversing tasks

taskList.traverseTasks();

// Searching for a task

Task foundTask = taskList.searchTask("T002");

System.out.println("Found Task: " + (foundTask != null ? foundTask : "Not Found"));

// Deleting a task

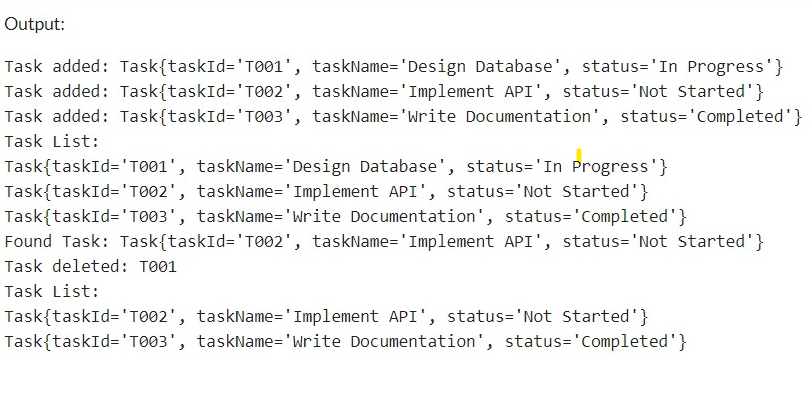
taskList.deleteTask("T001");

// Traversing tasks after deletion

taskList.traverseTasks();

}

}



**Analysis:**

**Time Complexity of Operations:**

* + **Add Task:**

Time Complexity: O(n)

* + **Search Task by ID:**

Time Complexity: O(n)

* + **Traverse Tasks:**

Time Complexity: O(n)

* + **Delete Task by ID:**

Time Complexity: O(n)

**Advantages of Linked Lists Over Arrays for Dynamic Data**

* Dynamic Size: Linked lists can grow and shrink dynamically, unlike arrays that have a fixed size.
* Efficient Insertions/Deletions: Insertions and deletions can be more efficient (O(1)) if the position is known, as there's no need to shift elements.
* Memory Usage: Linked lists use memory more efficiently for dynamic data as they allocate memory as needed, whereas arrays may allocate more memory than necessary.
* Flexibility: Linked lists provide more flexibility with dynamic data structures, making them more suitable for tasks where the size of the dataset changes frequently.

**Exercise 6: Library Management System**

**Explain linear search and binary search algorithms.**

**Linear Search:**

Linear search is a simple search algorithm that checks every element in the list sequentially until the desired element is found or the list ends.

* Time Complexity: O(n)
* Space Complexity: O(1)
* Best Case: O(1) (if the element is at the beginning)
* Worst Case: O(n) (if the element is at the end or not present)
* Use Case: Suitable for unsorted or small lists.

**Binary Search:**

Binary search is a more efficient search algorithm for sorted lists. It repeatedly divides the search interval in half, comparing the middle element with the target value.

* Time Complexity: O(log n)
* Space Complexity: O(1)
* Best Case: O(1) (if the middle element is the target)
* Worst Case: O(log n) (if the element is not present)
* Use Case: Suitable for large, sorted lists.

Linear search is straightforward and suitable for small or unsorted datasets, while binary search is more efficient for larger, sorted datasets due to its significantly lower time complexity.

**2.CODE:**

import java.util.Arrays;

class Book {

private String bookId;

private String title;

private String author;

public Book(String bookId, String title, String author) {

this.bookId = bookId;

this.title = title;

this.author = author;

}

// Getters

public String getBookId() {

return bookId;

}

public String getTitle() {

return title;

}

public String getAuthor() {

return author;

}

public String toString() {

return "Book{" +

"bookId='" + bookId + '\'' +

", title='" + title + '\'' +

", author='" + author + '\'' +

'}';

}

}

class BookSearch {

// Linear search implementation

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

for (Book book : books) {

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

return book; // Book found

}

}

return null; // Book not found

}

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

int left = 0;

int right = books.length - 1;

while (left <= right) {

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

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

if (comparison == 0) {

return books[mid]; // Book found

} else if (comparison < 0) {

left = mid + 1; // Search in the right half

} else {

right = mid - 1; // Search in the left half

}

}

return null; // Book not found

}

}

class Main {

public static void main(String[] args) {

Book[] books = {

new Book("B001", "The Great Gatsby", "F. Scott Fitzgerald"),

new Book("B002", "To Kill a Mockingbird", "Harper Lee"),

new Book("B003", "1984", "George Orwell"),

new Book("B004", "Pride and Prejudice", "Jane Austen"),

new Book("B005", "The Catcher in the Rye", "J.D. Salinger")

};

// Demonstrating Linear Search

String searchTitleLinear = "1984";

Book foundBookLinear = BookSearch.linearSearch(books, searchTitleLinear);

System.out.println("Linear Search Result: " + (foundBookLinear != null ? foundBookLinear : "Not Found"));

// Sorting books by title for Binary Search

Arrays.sort(books, (b1, b2) -> b1.getTitle().compareToIgnoreCase(b2.getTitle()));

// Demonstrating Binary Search

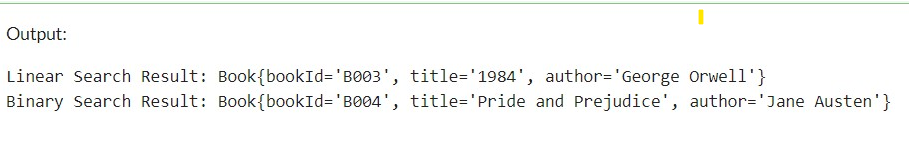
String searchTitleBinary = "Pride and Prejudice";

Book foundBookBinary = BookSearch.binarySearch(books, searchTitleBinary);

System.out.println("Binary Search Result: " + (foundBookBinary != null ? foundBookBinary : "Not Found"));

}

}



**Time Complexity of Search Algorithms**

**Linear Search:**

* Best Case: O(1)
* Average Case: O(n)
* Worst Case: O(n)
* Space Complexity: O(1)

**Binary Search:**

* Best Case: O(1)
* Average Case: O(log n)
* Worst Case: O(log n)
* Space Complexity: O(1)

**When to Use Each Algorithm**

**Linear Search:**

* Use for unsorted or small datasets.
* Simple to implement and does not require sorting.
* Efficient for cases where the dataset size is small or the target element is frequently near the beginning.

**Binary Search:**

* Use for large, sorted datasets.
* Much more efficient for large datasets due to its O(log n) time complexity.
* Requires the list to be sorted, adding an additional step if the data is not already sorted.

**Exercise 7: Financial Forecasting**

**Understand Recursive Algorithms**

**Concept of Recursion**

Recursion is a technique where a function calls itself to solve smaller instances of the same problem. It can simplify complex problems by breaking them down into more manageable subproblems.

* **Base Case**: The condition under which the recursion stops.
* **Recursive Case**: The part of the function where it calls itself with a smaller or simpler input.
* **Advantages**:
  + Simplifies code for problems that have repetitive structures.
  + Often more intuitive for problems like tree traversal, factorial calculation, etc.
* **Disadvantages**:
  + Can lead to excessive memory use due to function call stack.
  + Potential for stack overflow if not properly controlled.

**CODE:**

class FutureValueCalculator {

public static double calculateFutureValue(double presentValue, double interestRate, int years) {

if (years == 0) {

return presentValue;

} else {

return calculateFutureValue(presentValue, interestRate, years - 1) \* (1 + interestRate);

}

}

}

class Main {

public static void main(String[] args) {

double presentValue = 1000;

double interestRate = 0.05;

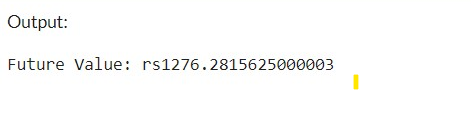
int years = 5;

double futureValue = FutureValueCalculator.calculateFutureValue(presentValue, interestRate, years);

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

}

}



**Analysis**

**Time Complexity of Recursive Algorithm**

* **Time Complexity**: O(n), where n is the number of periods.
  + Each call to the function handles one period, leading to n recursive calls.
* **Space Complexity**: O(n), due to the function call stack. Each recursive call adds a new frame to the stack.

**Optimizing the Recursive Solution**

To avoid excessive computation and potential stack overflow, we can use memoization to store previously computed results. However, in this simple growth rate model, memoization is not necessary as each step only depends on the previous step and does not repeat subproblems.