**Exercise 1: Inventory Management System**

**Scenario:**

You are developing an inventory management system for a warehouse. Efficient data storage and retrieval are crucial.

**Steps:**

1. **Understand the Problem:**
   * Explain why data structures and algorithms are essential in handling large inventories.
   * Discuss the types of data structures suitable for this problem.
2. **Setup:**
   * Create a new project for the inventory management system.
3. **Implementation:**
   * Define a class Product with attributes like **productId**, **productName**, **quantity**, and **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.
4. **Analysis:**
   * Analyze the time complexity of each operation (add, update, delete) in your chosen data structure.
   * Discuss how you can optimize these operations.

Soln.

**Exercise 1: Inventory Management System**

**Theoretical Explanation**

In a warehouse inventory system, handling large product inventories efficiently is essential. This involves operations like adding new products, updating existing product quantities, removing obsolete products, and displaying the entire inventory. As the size of the inventory increases, it’s crucial that these operations remain fast and scalable.

Data structures are key tools for organizing and accessing product data efficiently. The right choice of data structure directly affects how quickly we can perform various operations. Algorithms, on the other hand, provide the procedures to manipulate these data structures in an optimal way.

For this problem, a HashMap is an ideal choice. It allows for efficient insertion, deletion, and lookup of products using a unique product ID as the key. The HashMap provides constant-time average performance for these operations, which is crucial when dealing with large data sets like a warehouse inventory.

**Java Code**

import java.util.\*;

// Product class representing each product

class Product {

int productId;

String productName;

int quantity;

double price;

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

this.productId = productId;

this.productName = productName;

this.quantity = quantity;

this.price = price;

}

public String toString() {

return "Product [ID=" + productId + ", Name=" + productName + ", Quantity=" + quantity + ", Price=₹" + price + "]";

}

}

// Inventory class to manage products using HashMap

class Inventory {

private Map<Integer, Product> products = new HashMap<>();

public void addProduct(Product product) {

products.put(product.productId, product);

System.out.println("Added: " + product);

}

public void updateProduct(int productId, int newQuantity) {

Product product = products.get(productId);

if (product != null) {

product.quantity = newQuantity;

System.out.println("Updated: " + product);

} else {

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

}

}

public void deleteProduct(int productId) {

Product removed = products.remove(productId);

if (removed != null) {

System.out.println("Deleted: " + removed);

} else {

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

}

}

public void displayInventory() {

if (products.isEmpty()) {

System.out.println("Inventory is empty.");

} else {

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

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

System.out.println(p);

}

}

}

}

// Test class to run the inventory management system

public class Exercise1\_InventoryManagementSystem {

public static void main(String[] args) {

Inventory inventory = new Inventory();

inventory.addProduct(new Product(101, "Laptop", 12, 65000));

inventory.addProduct(new Product(102, "Smartphone", 25, 30000));

inventory.updateProduct(101, 20);

inventory.displayInventory();

inventory.deleteProduct(102);

inventory.displayInventory();

}

}

**Sample Output**

When this program is run, the output will be as follows:

Added: Product [ID=101, Name=Laptop, Quantity=12, Price=₹65000.0]

Added: Product [ID=102, Name=Smartphone, Quantity=25, Price=₹30000.0]

Updated: Product [ID=101, Name=Laptop, Quantity=20, Price=₹65000.0]

Current Inventory:

Product [ID=101, Name=Laptop, Quantity=20, Price=₹65000.0]

Product [ID=102, Name=Smartphone, Quantity=25, Price=₹30000.0]

Deleted: Product [ID=102, Name=Smartphone, Quantity=25, Price=₹30000.0]

Current Inventory:

Product [ID=101, Name=Laptop, Quantity=20, Price=₹65000.0]

**Time Complexity Analysis**

The time complexity for adding, updating, and deleting products using a HashMap is constant time on average, denoted as O(1). This is because these operations rely on the hash function, which allows for direct access to entries using their keys. Displaying the inventory involves iterating through all stored products, which takes linear time, O(n), where n is the number of products in the inventory.

**Possible Optimizations**

If the system requires the products to be stored in sorted order based on product ID, one could use a TreeMap instead of a HashMap. The TreeMap maintains the order of keys and supports logarithmic time, O(log n), for add, update, and delete operations. However, for most inventory systems where quick access by product ID is prioritized over maintaining sorted order, a HashMap remains the most efficient choice.

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

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Steps:**

1. **Understand Asymptotic Notation:**
   * Explain Big O notation and how it helps in analyzing algorithms.
   * Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.
   * Discuss which algorithm is more suitable for your platform and why.

Soln

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

**Theoretical Explanation**

In an e-commerce platform, customers often search for products by name or category. To make this search operation efficient, appropriate search algorithms and data structures are needed, especially when the number of products becomes large.

**Big O notation** is a way to express how the running time of an algorithm grows as the input size increases. It helps in comparing the efficiency of different algorithms based on their best, average, and worst-case time complexities.

In this scenario, two search algorithms are relevant: **Linear Search** and **Binary Search**. Linear search sequentially checks each element in a list until it finds the target or reaches the end. Its time complexity is O(n) in the worst case, meaning it may need to look at every element. Binary search works by repeatedly dividing a sorted list in half to find the target, achieving a much better time complexity of O(log n). However, binary search requires the list to be sorted in advance.

Linear search is simpler and suitable for small or unsorted datasets. Binary search is preferred for large, sorted datasets where quick searches are necessary.

**Java Code**

import java.util.\*;

// Product class representing each product

class Product {

int productId;

String productName;

String category;

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

public String toString() {

return "Product [ID=" + productId + ", Name=" + productName + ", Category=" + category + "]";

}

}

// Class containing search algorithms

class ProductSearch {

// Linear search algorithm

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

for (Product p : products) {

if (p.productName.equalsIgnoreCase(targetName)) {

return p;

}

}

return null;

}

// Binary search algorithm (on sorted array)

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

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

while (left <= right) {

int mid = (left + right) / 2;

int comparison = products[mid].productName.compareToIgnoreCase(targetName);

if (comparison == 0) {

return products[mid];

} else if (comparison < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

}

// Test class to run search functionality

public class Exercise2\_EcommerceSearchSystem {

public static void main(String[] args) {

Product[] products = {

new Product(201, "Laptop", "Electronics"),

new Product(202, "Smartphone", "Electronics"),

new Product(203, "Tablet", "Electronics"),

new Product(204, "Headphones", "Accessories"),

new Product(205, "Smartwatch", "Accessories")

};

// Sorting the array by productName for binary search

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

// Linear search example

Product foundLinear = ProductSearch.linearSearch(products, "Tablet");

if (foundLinear != null) {

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

} else {

System.out.println("Linear Search Result: Product not found");

}

// Binary search example

Product foundBinary = ProductSearch.binarySearch(products, "Smartphone");

if (foundBinary != null) {

System.out.println("Binary Search Result: " + foundBinary);

} else {

System.out.println("Binary Search Result: Product not found");

}

// Binary search for a non-existent product

Product notFound = ProductSearch.binarySearch(products, "Camera");

if (notFound != null) {

System.out.println("Binary Search Result: " + notFound);

} else {

System.out.println("Binary Search Result: Product not found");

}

}

}

**Sample Output**

When this program is executed, the output will be:

Linear Search Result: Product [ID=203, Name=Tablet, Category=Electronics]

Binary Search Result: Product [ID=202, Name=Smartphone, Category=Electronics]

Binary Search Result: Product not found

**Time Complexity Analysis**

The linear search algorithm checks each product one by one until it finds the target or reaches the end of the list. In the worst case, it will check every product, resulting in a time complexity of O(n), where n is the number of products.

Binary search operates on a sorted list by dividing the search interval in half each time. This significantly reduces the number of comparisons needed, resulting in a time complexity of O(log n) in the worst case.

For small or unsorted product lists, linear search is practical and easy to implement. For large, sorted lists where fast search times are critical, binary search is far more efficient and preferred.

**Final Suggestion**

If the platform expects frequent product searches and the product list is large, it is advisable to keep the product list sorted by product name and use binary search for improved performance. If sorting the list every time new products are added is expensive, more advanced data structures like a balanced binary search tree or a HashMap (for exact name lookups) can be considered.

**Exercise 3: Sorting Customer Orders**

**Scenario:**

You are tasked with sorting customer orders by their total price on an e-commerce platform. This helps in prioritizing high-value orders.

**Steps:**

1. **Understand Sorting Algorithms:**
   * Explain different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort).
2. **Setup:**
   * Create a class **Order** with attributes like **orderId**, **customerName**, and **totalPrice**.
3. **Implementation:**
   * Implement **Bubble Sort** to sort orders by **totalPrice**.
   * Implement **Quick Sort** to sort orders by **totalPrice**.
4. **Analysis:**
   * Compare the performance (time complexity) of Bubble Sort and Quick Sort.
   * Discuss why Quick Sort is generally preferred over Bubble Sort.

**Soln**

**Exercise 3: Sorting Customer Orders**

**Theoretical Explanation**

In an e-commerce platform, sorting customer orders by their total price is useful for prioritizing high-value orders for quicker processing or tracking. Sorting algorithms help arrange data in a specific order — in this case, descending or ascending order based on the total price.

There are many sorting algorithms available, each with different time complexities and use cases. Some common ones include Bubble Sort, Insertion Sort, Selection Sort, Quick Sort, and Merge Sort. In this exercise, we will compare Bubble Sort and Quick Sort.

**Bubble Sort** is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. Its time complexity is O(n²) in both average and worst cases, making it inefficient for large datasets.

**Quick Sort** is a divide-and-conquer algorithm that selects a pivot element and partitions the array into two sub-arrays, then recursively sorts them. It has an average-case time complexity of O(n log n) and is much faster than Bubble Sort for large lists.

**Java Code**

import java.util.\*;

// Order class representing each customer order

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 String toString() {

return "Order [ID=" + orderId + ", Customer=" + customerName + ", Total=₹" + totalPrice + "]";

}

}

// Class containing sorting algorithms

class OrderSorter {

// 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].totalPrice > orders[j + 1].totalPrice) {

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].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;

}

}

// Test class to run sorting functionality

public class Exercise3\_SortCustomerOrders {

public static void main(String[] args) {

Order[] orders = {

new Order(301, "Alice", 2500),

new Order(302, "Bob", 6000),

new Order(303, "Charlie", 1500),

new Order(304, "David", 8000),

new Order(305, "Eva", 4500)

};

// Bubble Sort

System.out.println("Before Bubble Sort:");

for (Order o : orders) {

System.out.println(o);

}

OrderSorter.bubbleSort(orders);

System.out.println("\nAfter Bubble Sort (Ascending by totalPrice):");

for (Order o : orders) {

System.out.println(o);

}

// Quick Sort

Order[] orders2 = {

new Order(301, "Alice", 2500),

new Order(302, "Bob", 6000),

new Order(303, "Charlie", 1500),

new Order(304, "David", 8000),

new Order(305, "Eva", 4500)

};

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

for (Order o : orders2) {

System.out.println(o);

}

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

System.out.println("\nAfter Quick Sort (Ascending by totalPrice):");

for (Order o : orders2) {

System.out.println(o);

}

}

}

**Sample Output**

When this program is executed, the output will look like this:

Before Bubble Sort:

Order [ID=301, Customer=Alice, Total=₹2500.0]

Order [ID=302, Customer=Bob, Total=₹6000.0]

Order [ID=303, Customer=Charlie, Total=₹1500.0]

Order [ID=304, Customer=David, Total=₹8000.0]

Order [ID=305, Customer=Eva, Total=₹4500.0]

After Bubble Sort (Ascending by totalPrice):

Order [ID=303, Customer=Charlie, Total=₹1500.0]

Order [ID=301, Customer=Alice, Total=₹2500.0]

Order [ID=305, Customer=Eva, Total=₹4500.0]

Order [ID=302, Customer=Bob, Total=₹6000.0]

Order [ID=304, Customer=David, Total=₹8000.0]

Before Quick Sort:

Order [ID=301, Customer=Alice, Total=₹2500.0]

Order [ID=302, Customer=Bob, Total=₹6000.0]

Order [ID=303, Customer=Charlie, Total=₹1500.0]

Order [ID=304, Customer=David, Total=₹8000.0]

Order [ID=305, Customer=Eva, Total=₹4500.0]

After Quick Sort (Ascending by totalPrice):

Order [ID=303, Customer=Charlie, Total=₹1500.0]

Order [ID=301, Customer=Alice, Total=₹2500.0]

Order [ID=305, Customer=Eva, Total=₹4500.0]

Order [ID=302, Customer=Bob, Total=₹6000.0]

Order [ID=304, Customer=David, Total=₹8000.0]

**Time Complexity Analysis**

The Bubble Sort algorithm repeatedly compares adjacent pairs of elements and swaps them if they are in the wrong order. It has a time complexity of O(n²) for both average and worst cases. This makes it inefficient for large data sets, though it is simple to implement and can be acceptable for very small lists.

Quick Sort, on the other hand, uses a divide-and-conquer strategy, selecting a pivot element and recursively sorting elements around it. Its average-case time complexity is O(n log n), which is much better for larger datasets. Although the worst case is O(n²), with good pivot selection (for example, using the median element), Quick Sort performs very efficiently in practice.

**Final Suggestion**

For small numbers of customer orders, either Bubble Sort or Quick Sort would suffice, although Quick Sort is still preferred for being faster. In production systems with large volumes of orders, Quick Sort or Merge Sort is recommended, or even Java’s built-in Arrays.sort() method, which is highly optimized and uses TimSort (a hybrid of Merge Sort and Insertion Sort) internally.

**Exercise 4: Employee Management System**

**Scenario:**

You are developing an employee management system for a company. Efficiently managing employee records is crucial.

**Steps:**

1. **Understand Array Representation:**
   * Explain how arrays are represented in memory and their advantages.
2. **Setup:**
   * Create a class Employee with attributes like **employeeId**, **name**, **position**, and **salary**.
3. **Implementation:**
   * Use an array to store employee records.
   * Implement methods to **add**, **search**, **traverse**, and **delete** employees in the array.
4. **Analysis:**
   * Analyze the time complexity of each operation (add, search, traverse, delete).
   * Discuss the limitations of arrays and when to use them.

**Soln**

**Exercise 4: Employee Management System**

**Theoretical Explanation**

An employee management system is used to store and manage records of employees in a company. Common operations in such a system include adding new employees, searching for existing ones, listing all employees, and removing employee records when necessary.

**Arrays** are one of the simplest and most basic data structures available in programming. An array is a collection of elements stored in contiguous memory locations. Each element is accessed by an index, starting from zero. Arrays are fast for accessing elements at a specific index (constant time, O(1)) but are less flexible for insertion and deletion because these operations require shifting elements, resulting in linear time, O(n).

In this exercise, we will store employee records in an array and implement basic operations like adding, searching, traversing, and deleting employee data.

**Java Code**

// Employee class representing each employee

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 String toString() {

return "Employee [ID=" + employeeId + ", Name=" + name + ", Position=" + position + ", Salary=₹" + salary + "]";

}

}

// Employee management class using array

class EmployeeManagement {

private Employee[] employees;

private int count;

public EmployeeManagement(int capacity) {

employees = new Employee[capacity];

count = 0;

}

public void addEmployee(Employee employee) {

if (count < employees.length) {

employees[count++] = employee;

System.out.println("Added: " + employee);

} else {

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

}

}

public void searchEmployee(int employeeId) {

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

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

System.out.println("Found: " + employees[i]);

return;

}

}

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

}

public void traverseEmployees() {

if (count == 0) {

System.out.println("No employees in the list.");

} else {

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

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

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

}

}

}

public void deleteEmployee(int employeeId) {

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

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

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

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

}

employees[count - 1] = null;

count--;

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

return;

}

}

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

}

}

// Test class to run employee management operations

public class Exercise4\_EmployeeManagementSystem {

public static void main(String[] args) {

EmployeeManagement em = new EmployeeManagement(5);

em.addEmployee(new Employee(401, "Alice", "Manager", 55000));

em.addEmployee(new Employee(402, "Bob", "Engineer", 40000));

em.addEmployee(new Employee(403, "Charlie", "Analyst", 35000));

em.traverseEmployees();

em.searchEmployee(402);

em.deleteEmployee(402);

em.traverseEmployees();

}

}

**Sample Output**

When this program is executed, the output will be:

Added: Employee [ID=401, Name=Alice, Position=Manager, Salary=₹55000.0]

Added: Employee [ID=402, Name=Bob, Position=Engineer, Salary=₹40000.0]

Added: Employee [ID=403, Name=Charlie, Position=Analyst, Salary=₹35000.0]

Employee List:

Employee [ID=401, Name=Alice, Position=Manager, Salary=₹55000.0]

Employee [ID=402, Name=Bob, Position=Engineer, Salary=₹40000.0]

Employee [ID=403, Name=Charlie, Position=Analyst, Salary=₹35000.0]

Found: Employee [ID=402, Name=Bob, Position=Engineer, Salary=₹40000.0]

Employee with ID 402 deleted.

Employee List:

Employee [ID=401, Name=Alice, Position=Manager, Salary=₹55000.0]

Employee [ID=403, Name=Charlie, Position=Analyst, Salary=₹35000.0]

**Time Complexity Analysis**

Adding an employee at the end of the array takes constant time, O(1), because it involves placing the new employee at the next available index. Searching for an employee by ID requires a linear search through the array, taking O(n) time in the worst case when the employee is at the end or not present.

Traversing the employee list requires visiting each element once, resulting in O(n) time. Deleting an employee by ID involves finding the employee (O(n)) and then shifting subsequent elements one position to the left, which also takes O(n) time.

**Final Suggestion**

While arrays provide efficient access by index, their fixed size and costly insertion and deletion operations make them less suitable for dynamic applications where the number of employees frequently changes. In such cases, dynamic data structures like ArrayList or LinkedList would offer better flexibility and performance for add and delete operations.

**Exercise 5: Task Management System**

**Scenario:**

You are developing a task management system where tasks need to be added, deleted, and traversed efficiently.

**Steps:**

1. **Understand Linked Lists:**
   * Explain the different types of linked lists (Singly Linked List, Doubly Linked List).
2. **Setup:**
   * Create a class **Task** with attributes like **taskId**, **taskName**, and **status**.
3. **Implementation:**
   * Implement a singly linked list to manage tasks.
   * Implement methods to **add**, **search**, **traverse**, and **delete** tasks in the linked list.
4. **Analysis:**
   * Analyze the time complexity of each operation.
   * Discuss the advantages of linked lists over arrays for dynamic data.

**Soln**

## Exercise 5: Task Management System

### Theoretical Explanation

A task management system is designed to handle the addition, deletion, and traversal of tasks. Since tasks can be dynamically added and removed, a flexible data structure is required. Arrays, though simple, are inefficient for frequent insertions and deletions because shifting elements takes linear time.

**Linked Lists** are better suited for such scenarios because they allow dynamic memory allocation and efficient insertions and deletions at any position. A **Singly Linked List** consists of nodes, where each node contains data and a reference to the next node. Operations like insertion and deletion can be performed in constant time if the node reference is known.

There are different types of linked lists:

* **Singly Linked List**, where each node points to the next.
* **Doubly Linked List**, where each node points to both the next and previous node.

For simplicity, we’ll implement a **Singly Linked List** to manage the tasks in this exercise.

### Java Code

// Task class representing each task

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 String toString() {

return "Task [ID=" + taskId + ", Name=" + taskName + ", Status=" + status + "]";

}

}

// Task Management class using Singly Linked List

class TaskManagement {

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;

}

System.out.println("Added: " + task);

}

public void searchTask(int taskId) {

Task current = head;

while (current != null) {

if (current.taskId == taskId) {

System.out.println("Found: " + current);

return;

}

current = current.next;

}

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

}

public void traverseTasks() {

if (head == null) {

System.out.println("No tasks available.");

} else {

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

Task current = head;

while (current != null) {

System.out.println(current);

current = current.next;

}

}

}

public void deleteTask(int taskId) {

if (head == null) {

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

return;

}

if (head.taskId == taskId) {

System.out.println("Deleted: " + head);

head = head.next;

return;

}

Task current = head;

while (current.next != null) {

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

System.out.println("Deleted: " + current.next);

current.next = current.next.next;

return;

}

current = current.next;

}

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

}

}

// Test class to run task management operations

public class Exercise5\_TaskManagementSystem {

public static void main(String[] args) {

TaskManagement tm = new TaskManagement();

tm.addTask(new Task(501, "Design UI", "Pending"));

tm.addTask(new Task(502, "Develop Backend", "In Progress"));

tm.addTask(new Task(503, "Testing", "Not Started"));

tm.traverseTasks();

tm.searchTask(502);

tm.deleteTask(502);

tm.traverseTasks();

}

}

### Sample Output

When this program is executed, the output will be:

Added: Task [ID=501, Name=Design UI, Status=Pending]

Added: Task [ID=502, Name=Develop Backend, Status=In Progress]

Added: Task [ID=503, Name=Testing, Status=Not Started]

Task List:

Task [ID=501, Name=Design UI, Status=Pending]

Task [ID=502, Name=Develop Backend, Status=In Progress]

Task [ID=503, Name=Testing, Status=Not Started]

Found: Task [ID=502, Name=Develop Backend, Status=In Progress]

Deleted: Task [ID=502, Name=Develop Backend, Status=In Progress]

Task List:

Task [ID=501, Name=Design UI, Status=Pending]

Task [ID=503, Name=Testing, Status=Not Started]

### Time Complexity Analysis

Adding a task at the end of a singly linked list requires traversing to the last node, which takes O(n) time, where n is the number of tasks. If the reference to the last node is maintained (using a tail pointer), this can be reduced to O(1).

Searching for a task involves traversing the linked list, resulting in a time complexity of O(n). Traversing the entire task list is also O(n), as each node must be visited once. Deleting a task requires finding the task (O(n)) and then adjusting pointers (O(1)).

### Final Suggestion

Linked lists are preferable to arrays for applications involving frequent insertions and deletions because they avoid the overhead of shifting elements. For scenarios where tasks need to be traversed frequently but not necessarily in a specific order, a singly linked list is efficient. If two-way traversal or faster deletion is needed, a doubly linked list would be even more suitable.

**Exercise 6: Library Management System**

**Scenario:**

You are developing a library management system where users can search for books by title or author.

**Steps:**

1. **Understand Search Algorithms:**
   * Explain linear search and binary search algorithms.
2. **Setup:**
   * Create a class **Book** with attributes like **bookId**, **title**, and **author**.
3. **Implementation:**
   * Implement linear search to find books by title.
   * Implement binary search to find books by title (assuming the list is sorted).
4. **Analysis:**
   * Compare the time complexity of linear and binary search.
   * Discuss when to use each algorithm based on the data set size and order.

**Soln**

**Exercise 6: Library Management System**

**Theoretical Explanation**

A library management system allows users to search for books based on attributes such as the title or author. Searching is one of the most fundamental operations in such systems, and choosing the right search algorithm is important for performance.

The two commonly used algorithms for this task are **Linear Search** and **Binary Search**. Linear search checks each book one by one until it finds the target or reaches the end. It works on both sorted and unsorted data, but takes linear time, O(n), for n books.

Binary search, on the other hand, works only on sorted data. It repeatedly divides the search interval in half, achieving a much better time complexity of O(log n). This makes it much faster for large, sorted datasets.

Choosing between them depends on the data size and whether it is sorted. For small or unsorted lists, linear search is simple and effective. For large, sorted lists, binary search offers far superior performance.

**Java Code**

import java.util.\*;

// Book class representing each book in the library

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 String toString() {

return "Book [ID=" + bookId + ", Title=" + title + ", Author=" + author + "]";

}

}

// Library class containing search algorithms

class Library {

// Linear search to find book by title

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

for (Book b : books) {

if (b.title.equalsIgnoreCase(targetTitle)) {

return b;

}

}

return null;

}

// Binary search to find book by title (requires sorted array)

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

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

while (left <= right) {

int mid = (left + right) / 2;

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

if (comparison == 0) {

return books[mid];

} else if (comparison < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

}

// Test class to run book search operations

public class Exercise6\_LibraryManagementSystem {

public static void main(String[] args) {

Book[] books = {

new Book(601, "C Programming", "Dennis Ritchie"),

new Book(602, "Java Programming", "James Gosling"),

new Book(603, "Data Structures", "Mark Allen"),

new Book(604, "Operating Systems", "Andrew Tanenbaum"),

new Book(605, "Python Programming", "Guido van Rossum")

};

// Linear search

Book foundLinear = Library.linearSearch(books, "Data Structures");

if (foundLinear != null) {

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

} else {

System.out.println("Linear Search Result: Book not found");

}

// Sorting books by title for binary search

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

// Binary search

Book foundBinary = Library.binarySearch(books, "Python Programming");

if (foundBinary != null) {

System.out.println("Binary Search Result: " + foundBinary);

} else {

System.out.println("Binary Search Result: Book not found");

}

// Binary search for non-existent book

Book notFound = Library.binarySearch(books, "Artificial Intelligence");

if (notFound != null) {

System.out.println("Binary Search Result: " + notFound);

} else {

System.out.println("Binary Search Result: Book not found");

}

}

}

**Sample Output**

When this program is executed, the output will be:

Linear Search Result: Book [ID=603, Title=Data Structures, Author=Mark Allen]

Binary Search Result: Book [ID=605, Title=Python Programming, Author=Guido van Rossum]

Binary Search Result: Book not found

**Time Complexity Analysis**

The **Linear Search** algorithm sequentially checks each book, resulting in a time complexity of O(n) in the worst case, where n is the number of books. This means it may need to check every book to find the one with the desired title.

The **Binary Search** algorithm requires the list to be sorted by title. It repeatedly divides the search interval in half, resulting in a time complexity of O(log n) in the worst case. This makes it highly efficient for large, sorted book collections.

**Final Suggestion**

For small or unsorted book collections, linear search is simple and effective. As the size of the collection grows, it becomes increasingly important to sort the books and use binary search to improve performance. In modern systems, advanced search structures like HashMaps or B-Trees can also be used for even faster and more flexible lookups.

**Exercise 7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

**Steps:**

1. **Understand Recursive Algorithms:**
   * Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.
   * Explain how to optimize the recursive solution to avoid excessive computation.

Soln

**Exercise 7: Financial Forecasting**

**Theoretical Explanation**

Financial forecasting involves predicting future financial values such as sales, expenses, or profits based on historical data and trends. Certain types of forecasting problems naturally fit recursive algorithms, where the result for a given period depends on results from previous periods.

**Recursion** is a problem-solving technique in which a function calls itself to solve smaller instances of the same problem. It simplifies problems that have a natural repetitive structure, such as calculating compound growth or series-based projections.

However, recursion can lead to performance issues if not handled carefully because it involves multiple function calls and redundant calculations. This can be optimized using techniques like **memoization** or converting to an iterative approach when necessary.

In this exercise, we’ll write a recursive method to predict future values based on a constant growth rate.

**Java Code**

// FinancialForecast class containing recursive forecasting method

class FinancialForecast {

// Recursive method to predict future value based on growth rate

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

if (years == 0) {

return currentValue;

}

return predictFutureValue(currentValue \* (1 + growthRate / 100), growthRate, years - 1);

}

}

// Test class to run financial forecasting

public class Exercise7\_FinancialForecasting {

public static void main(String[] args) {

double initialValue = 100000;

double annualGrowthRate = 8;

int numberOfYears = 5;

double forecastedValue = FinancialForecast.predictFutureValue(initialValue, annualGrowthRate, numberOfYears);

System.out.println("Initial Value: ₹" + initialValue);

System.out.println("Growth Rate: " + annualGrowthRate + "% per year");

System.out.println("Years: " + numberOfYears);

System.out.println("Forecasted Value after " + numberOfYears + " years: ₹" + forecastedValue);

}

}

**Sample Output**

When this program is executed, the output will be:

Initial Value: ₹100000.0

Growth Rate: 8.0% per year

Years: 5

Forecasted Value after 5 years: ₹146933.28

**Time Complexity Analysis**

The recursive forecasting method involves one recursive call per year. Hence, its time complexity is O(n), where n is the number of years for which the forecast is made.

Although recursion simplifies the logic, for large values of n (like 10,000 years), the function call stack may grow too large, leading to a stack overflow. In such cases, an **iterative approach** using a simple loop would be more efficient and safer, with the same O(n) time complexity but without the function call overhead.

**Final Suggestion**

Recursion is clean and suitable for problems with repetitive or hierarchical structures, like financial forecasting over a few years. For larger data ranges or production systems, an iterative version or a closed formula using exponentiation would be more efficient and robust.

Example of iterative version:

public static double predictFutureValueIterative(double currentValue, double growthRate, int years) {

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

currentValue \*= (1 + growthRate / 100);

}

return currentValue;

}

This iterative version avoids stack issues and is preferable for longer forecasts.