**Exercise 1: INVENTORY MANAGEMENT SYSTEM:**

**STEP 1: Understand the Problem**

**Importance of Data Structures and Algorithms:**

Data structures and algorithms are essential in handling large inventories for several reasons:

**Efficiency:** Proper data structures ensure that operations like adding, updating, and deleting products are performed quickly and efficiently.

**Scalability:** As the inventory grows, efficient data structures ensure the system can handle larger datasets without significant performance degradation.

**Organization:** They help in organizing the data in a way that makes retrieval and manipulation straightforward.

**Memory Management:** Efficient use of memory ensures that the system can handle large inventories without exhausting available resources.

**Suitable Data Structures:**

**ArrayList (Dynamic Array):** Allows dynamic resizing, fast access to elements, and easy iteration.

**HashMap (Hash Table):** Provides average constant-time complexity for insertions, deletions, and lookups. Suitable for quick access to products using unique identifiers like productId.

**LinkedList:** Useful for frequent insertions and deletions but less efficient for access operations.

**TreeMap (Red-Black Tree):** Provides sorted order of elements and guarantees O(log n) time complexity for basic operations.

**STEP 2: Setup**

Project Creation: Create a new project InventoryManagement

**STEP 3. Implementation**

Defining the Product Class:

**//PRODUCT.JAVA**

package Inventory;

import java.util.Scanner;

public class Product {

private String itemCode;

private String itemName;

private double itemPrice;

private int itemStock;

// Constructor for initialization with Scanner input

public Product() {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter Item Code: ");

this.itemCode = scanner.nextLine();

System.out.print("Enter Item Name: ");

this.itemName = scanner.nextLine();

System.out.print("Enter Item Quantity: ");

this.itemStock = scanner.nextInt();

System.out.print("Enter Item Price: ");

this.itemPrice = scanner.nextDouble();

}

// Constructor with parameters

public Product(String itemCode, String itemName, int itemStock, double itemPrice) {

this.itemCode = itemCode;

this.itemName = itemName;

this.itemStock = itemStock;

this.itemPrice = itemPrice;

}

public String getItemCode() {

return itemCode;

}

public String getItemName() {

return itemName;

}

public int getItemStock() {

return itemStock;

}

public double getItemPrice() {

return itemPrice;

}

public void setItemCode(String itemCode) {

this.itemCode = itemCode;

}

public void setItemName(String itemName) {

this.itemName = itemName;

}

public void setItemStock(int itemStock) {

this.itemStock = itemStock;

}

public void setItemPrice(double itemPrice) {

this.itemPrice = itemPrice;

}

@Override

public String toString() {

return "Item Code: " + itemCode + ", Name: " + itemName + ", Quantity: " + itemStock + ", Price: " + itemPrice;

}

}

**//InventoryManagement.java**

package Inventory;

import java.util.HashMap;

public class InventoryManagement {

private HashMap<String, Product> inventory;

public InventoryManagement() {

inventory = new HashMap<>();

}

public void addProduct(Product prod) {

inventory.put(prod.getItemCode(), prod);

}

public void updateInventory(String itemCode, int quantity, double price) {

Product product = inventory.get(itemCode);

if (product != null) {

product.setItemPrice(price);

product.setItemStock(quantity);

}

}

public void deleteProduct(String itemCode) {

inventory.remove(itemCode);

}

public Product viewProduct(String itemCode) {

return inventory.get(itemCode);

}

}

**//TestProduct.java**

package Inventory;

import java.util.Scanner;

public class TestProduct {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.println("Enter Product ID:");

String prodID = sc.nextLine();

System.out.println("Enter Product Name:");

String prodName = sc.nextLine();

System.out.println("Enter Product Price:");

double prodPrice = sc.nextDouble();

System.out.println("Enter Product Quantity:");

int prodQty = sc.nextInt();

Product prod = new Product(prodID, prodName, prodQty, prodPrice);

InventoryManagement inventory = new InventoryManagement();

inventory.addProduct(prod);

Product viewedProduct = inventory.viewProduct(prodID);

if (viewedProduct != null) {

System.out.println(viewedProduct);

} else {

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

}

sc.close();

}

}

**STEP 4. Analysis**

**Time Complexity:**

**Add :**

Using HashMap.put(): O(1) on average (constant time complexity).

**Update:**

Using HashMap.get() to retrieve the product: O(1) on average.

Updating product attributes: O(1).

Total time complexity: O(1).

**Delete:**

Using HashMap.remove(): O(1) on average.

**Optimization**:

**Concurrency**: Use concurrent data structures (e.g., ConcurrentHashMap) to handle multiple threads accessing the inventory simultaneously.

**Bulk** **Operations**: Implement bulk operations for adding, updating, and deleting multiple products at once to reduce the overhead of repeated operations.

**Caching**: Implement caching mechanisms to store frequently accessed products, reducing the need for repeated lookups.

**Indexing**: If using a database for persistent storage, ensure proper indexing on columns like productId to speed up retrieval operations.

By choosing efficient data structures and optimizing operations, we ensure the inventory management system can handle large inventories with minimal performance issues.

**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.

**Step 1: Understand Asymptotic Notation**

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

* **Big O Notation:** Big O notation is a mathematical representation used to describe the upper bound of an algorithm's running time. It characterizes the algorithm's performance in terms of the input size (n) and provides an abstract measure of the algorithm's efficiency by expressing the worst-case scenario.

**STEP 2: IMPLEMENTATION:**

**//SearchingDSA.java**

package searchingdsa;

import java.util.Arrays;

import java.util.Scanner;

public class SearchingDSA {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter number of products: ");

int numberOfProducts = scanner.nextInt();

scanner.nextLine(); // Consume newline

Product[] products = new Product[numberOfProducts];

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

System.out.print("Enter Item ID: ");

int id = scanner.nextInt();

scanner.nextLine(); // Consume newline

System.out.print("Enter Item Name: ");

String name = scanner.nextLine();

System.out.print("Enter Item Category: ");

String category = scanner.nextLine();

products[i] = new Product(id, name, category);

}

System.out.print("Enter target Item ID to search: ");

int targetId = scanner.nextInt();

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

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

// BINARY SEARCH

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

result = SearchAlgorithms.binarySearch(products, targetId);

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

scanner.close();

}

}

**//Product.java**

package searchingdsa;

public class Product {

private int itemId;

private String itemName;

private String itemCategory;

public Product(int itemId, String itemName, String itemCategory) {

this.itemId = itemId;

this.itemName = itemName;

this.itemCategory = itemCategory;

}

public int getItemId() {

return itemId;

}

public String getItemName() {

return itemName;

}

public String getItemCategory() {

return itemCategory;

}

@Override

public String toString() {

return "Product{" +

"itemId=" + itemId +

", itemName='" + itemName + '\'' +

", itemCategory='" + itemCategory + '\'' +

'}';

}

}

**//SearchAlgorithms.java**

package searchingdsa;

public class SearchAlgorithms {

// LINEAR SEARCH

public static Product linearSearch(Product[] items, int itemId) {

for (Product item : items) {

if (item.getItemId() == itemId) {

return item;

}

}

return null;

}

// BINARY SEARCH

public static Product binarySearch(Product[] items, int itemId) {

int left = 0;

int right = items.length - 1;

while (left <= right) {

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

if (items[mid].getItemId() == itemId) {

return items[mid];

} else if (items[mid].getItemId() < itemId) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

}

**Step 4: Analysis**

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

* **Linear Search Time Complexity:** O(n) in the worst and average cases, and O(1) in the best case.
* **Binary Search Time Complexity:** O(log n) in the worst and average cases, and O(1) in the best case.
* **Explanation:** Linear search checks each element sequentially, leading to linear time complexity. Binary search, on the other hand, divides the search space in half each time, resulting in logarithmic time complexity.

**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.

**UNDERSTAND SORTING ALGORITHMS:**

**Bubble Sort**

Bubble sort is a sorting algorithm that compares two adjacent elements and swaps them until they are in the intended order.

Just like the movement of air bubbles in the water that rise up to the surface, each element of the array move to the end in each iteration. Therefore, it is called a bubble sort.

**Working of Bubble Sort**

Suppose we are trying to sort the elements in ascending order.

1. First Iteration (Compare and Swap)

Starting from the first index, compare the first and the second elements.

If the first element is greater than the second element, they are swapped.

Now, compare the second and the third elements. Swap them if they are not in order.

The above process goes on until the last element. Compare the Adjacent Elements

1. Remaining Iteration

The same process goes on for the remaining iterations.

After each iteration, the largest element among the unsorted elements is placed at the end.

Put the largest element at the end

In each iteration, the comparison takes place up to the last unsorted element.

Compare the adjacent elements

The array is sorted when all the unsorted elements are placed at their correct positions.

The array is sorted if all elements are kept in the right order

**Bubble Sort Algorithm**

bubbleSort(array)

for i <- 1 to indexOfLastUnsortedElement-1

if leftElement > rightElement

swap leftElement and rightElement

end bubbleSort

**Bubble Sort Complexity:**

|  |  |
| --- | --- |
| Time Complexity |  |
| Best | O(n) |
| Worst | O(n2) |
| Average | O(n2) |
| Space Complexity | O(1) |

**Insertion Sort Algorithm**

Insertion sort is  a sorting algorithm that places an unsorted element at its suitable place in each iteration.

Insertion sort works similarly as we sort cards in our hand in a card game.

We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put in their right place.

A similar approach is used by insertion sort.

**Working of Insertion Sort:**

Suppose we need to sort the following array.

The first element in the array is assumed to be sorted. Take the second element and store it separately in key.  
  
Compare key with the first element. If the first element is greater than key, then key is placed in front of the first element. If the first element is greater than key, then key is placed in front of the first element.

Now, the first two elements are sorted.  
  
Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array. Place 1 at the beginning

Similarly, place every unsorted element at its correct position. Place 4 behind 1Place 3 behind 1 and the array is sorted

**Insertion Sort Algorithm**

insertionSort(array)

mark first element as sorted

for each unsorted element X

'extract' the element X

for j <- lastSortedIndex down to 0

if current element j > X

move sorted element to the right by 1

break loop and insert X here

end insertionSort

**Insertion Sort Complexity**

|  |  |
| --- | --- |
| **Time Complexity** |  |
| Best | O(n) |
| Worst | O(n2) |
| Average | O(n2) |
| **Space Complexity** | O(1) |

**Merge Sort Algorithm:**

Merge Sort is one of the most popular sorting algorithms that is based on the principle of **“DIVIDE AND CONQUER ALGORITHM”**

Here, a problem is divided into multiple sub-problems. Each sub-problem is solved individually. Finally, sub-problems are combined to form the final solution.

Divide and Conquer Strategy

Using the Divide and Conquer technique, we divide a problem into subproblems. When the solution to each subproblem is ready, we 'combine' the results from the subproblems to solve the main problem.

Suppose we had to sort an array A. A subproblem would be to sort a sub-section of this array starting at index p and ending at index r, denoted as A[p..r].

**Divide**

If q is the half-way point between p and r, then we can split the subarray A[p..r] into two arrays A[p..q] and A[q+1, r].

**Conquer**

In the conquer step, we try to sort both the subarrays A[p..q] and A[q+1, r]. If we haven't yet reached the base case, we again divide both these subarrays and try to sort them.

**Combine**

When the conquer step reaches the base step and we get two sorted subarrays A[p..q] and A[q+1, r] for array A[p..r], we combine the results by creating a sorted array A[p..r] from two sorted subarrays A[p..q] and A[q+1, r].

**MergeSort Algorithm:**

The MergeSort function repeatedly divides the array into two halves until we reach a stage where we try to perform MergeSort on a subarray of size 1 i.e. p == r.

After that, the merge function comes into play and combines the sorted arrays into larger arrays until the whole array is merged.

MergeSort(A, p, r):

if p > r

return

q = (p+r)/2

mergeSort(A, p, q)

mergeSort(A, q+1, r)

merge(A, p, q, r)

To sort an entire array, we need to call MergeSort(A, 0, length(A)-1).

The merge sort algorithm recursively divides the array into halves until we reach the base case of array with 1 element. After that, the merge function picks up the sorted sub-arrays and merges them to gradually sort the entire array.

The merge Step of Merge Sort

Every recursive algorithm is dependent on a base case and the ability to combine the results from base cases. Merge sort is no different. The most important part of the merge sort algorithm is, you guessed it, merge step.

The merge step is the solution to the simple problem of merging two sorted lists(arrays) to build one large sorted list(array).

The algorithm maintains three pointers, one for each of the two arrays and one for maintaining the current index of the final sorted array.

**Merge Sort Complexity**

|  |  |
| --- | --- |
| Time Complexity |  |
| Best | O(n\*log n) |
| Worst | O(n\*log n) |
| Average | O(n\*log n) |
| Space Complexity | O(n) |

**Quicksort Algorithm:**

Quicksort is a sorting algorithm based on the divide and conquer approach where

An array is divided into subarrays by selecting a pivot element (element selected from the array).  
  
While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.

The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.

At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

**Working of Quicksort Algorithm**

1. **Select the Pivot Element**

There are different variations of quicksort where the pivot element is selected from different positions. Here, we will be selecting the rightmost element of the array as the pivot element.

Select a pivot element

1. **Rearrange the Array**

Now the elements of the array are rearranged so that elements that are smaller than the pivot are put on the left and the elements greater than the pivot are put on the right.

Put all the smaller elements on the left and greater on the right of pivot element

Here's how we rearrange the array:

A pointer is fixed at the pivot element. The pivot element is compared with the elements beginning from the first index. Comparison of pivot element with element beginning from the first index

If the element is greater than the pivot element, a second pointer is set for that element. If the element is greater than the pivot element, a second pointer is set for that element.

Now, pivot is compared with other elements. If an element smaller than the pivot element is reached, the smaller element is swapped with the greater element found earlier. Pivot is compared with other elements.

Again, the process is repeated to set the next greater element as the second pointer. And, swap it with another smaller element. The process is repeated to set the next greater element as the second pointer.

The process goes on until the second last element is reached. The process goes on until the second last element is reached.

Finally, the pivot element is swapped with the second pointer. Finally, the pivot element is swapped with the second pointer.

3. Divide Subarrays

Pivot elements are again chosen for the left and the right sub-parts separately. And, step 2 is repeated. Select pivot element of in each half and put at correct place using recursion

The subarrays are divided until each subarray is formed of a single element. At this point, the array is already sorted.

**Quick Sort Algorithm**

quickSort(array, leftmostIndex, rightmostIndex)

if (leftmostIndex < rightmostIndex)

pivotIndex <- partition(array,leftmostIndex, rightmostIndex)

quickSort(array, leftmostIndex, pivotIndex - 1)

quickSort(array, pivotIndex, rightmostIndex)

partition(array, leftmostIndex, rightmostIndex)

set rightmostIndex as pivotIndex

storeIndex <- leftmostIndex - 1

for i <- leftmostIndex + 1 to rightmostIndex

if element[i] < pivotElement

swap element[i] and element[storeIndex]

storeIndex++

swap pivotElement and element[storeIndex+1]

return storeIndex + 1

**Quicksort Complexity**

|  |  |
| --- | --- |
| Time Complexity |  |
| Best | O(n\*log n) |
| Worst | O(n2) |
| Average | O(n\*log n) |
| Space Complexity | O(log n) |

**SETUP AND IMPLEMENTATION:**

**//Order.java**

package purchase;

import java.util.Scanner;

public class Order {

private String orderNumber;

private String clientName;

private double totalAmount;

public Order(String orderNumber, String clientName, double totalAmount) {

this.orderNumber = orderNumber;

this.clientName = clientName;

this.totalAmount = totalAmount;

}

public String getOrderNumber() {

return orderNumber;

}

public String getClientName() {

return clientName;

}

public double getTotalAmount() {

return totalAmount;

}

@Override

public String toString() {

return "Order{" +

"orderNumber='" + orderNumber + '\'' +

", clientName='" + clientName + '\'' +

", totalAmount=" + totalAmount +

'}';

}

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].getTotalAmount() > orders[j + 1].getTotalAmount()) {

Order temp = orders[j];

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

orders[j + 1] = temp;

}

}

}

}

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].getTotalAmount();

int i = (low - 1);

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

if (orders[j].getTotalAmount() < 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;

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter number of orders: ");

int numberOfOrders = scanner.nextInt();

scanner.nextLine(); // Consume newline

Order[] orders = new Order[numberOfOrders];

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

System.out.print("Enter Order Number: ");

String orderNumber = scanner.nextLine();

System.out.print("Enter Client Name: ");

String clientName = scanner.nextLine();

System.out.print("Enter Total Amount: ");

double totalAmount = scanner.nextDouble();

scanner.nextLine(); // Consume newline

orders[i] = new Order(orderNumber, clientName, totalAmount);

}

System.out.println("Original Order Array:");

for (Order order : orders) {

System.out.println(order);

}

Order[] bubbleSortedOrders = orders.clone();

bubbleSort(bubbleSortedOrders);

System.out.println("\nBubble Sorted Order Array:");

for (Order order : bubbleSortedOrders) {

System.out.println(order);

}

Order[] quickSortedOrders = orders.clone();

quickSort(quickSortedOrders, 0, quickSortedOrders.length - 1);

System.out.println("\nQuick Sorted Order Array:");

for (Order order : quickSortedOrders) {

System.out.println(order);

}

scanner.close();

}

}

**Performance Analysis**

Time Complexity Comparison

* **Bubble Sort:**
  + **Best Case:** O(n) - Occurs when the array is already sorted.
  + **Average Case:** O(n^2) - Involves comparing and swapping elements in a nested loop.
  + **Worst Case:** O(n^2)- Occurs when the array is sorted in reverse order.
* **Quick Sort:**
  + **Best Case:** O(nlogn) - Occurs when the pivot divides the array into two nearly equal halves.
  + **Average Case:** O(nlogn) - Generally, the pivot will partition the array reasonably well.
  + **Worst Case:** O(n^2) - Occurs when the pivot is the smallest or largest element, resulting in highly unbalanced partitions. This can be mitigated by using techniques like choosing a random pivot or the median-of-three.

Why Quick Sort is Generally Preferred Over Bubble Sort

1. **Efficiency on Large Datasets:**
   * Quick Sort has an average time complexity of O(nlogn), making it significantly faster than Bubble Sort's O(n^2)for large datasets. The logarithmic factor makes a considerable difference in execution time as the input size increases.
2. **Divide and Conquer Strategy:**
   * Quick Sort's divide-and-conquer strategy is efficient at breaking down the problem into smaller subproblems, leading to faster sorting overall. This makes it adaptable to various types of input data.
3. **Fewer Comparisons and Swaps:**
   * Quick Sort generally performs fewer comparisons and swaps than Bubble Sort. Bubble Sort repeatedly compares and swaps adjacent elements, leading to a higher number of operations, especially in large or reversed arrays.
4. **Better Performance on Average:**
   * Despite its worst-case time complexity of O(n^2), Quick Sort's average performance is much better due to its balanced partitioning approach in most cases. Using strategies like random pivot selection or median-of-three can help avoid the worst-case scenario.
5. **In-Place Sorting:**
   * Quick Sort is an in-place sort (requiring only a small, constant amount of extra storage space), which is beneficial for memory efficiency. While Bubble Sort is also in-place, its performance drawback outweighs this benefit.

**EXERCISE 4: EMPLOYEE MANAGEMENT SYSTEM:**

**STEP 1. Understanding Array Representation:**

Arrays are represented in memory as contiguous blocks, allowing efficient access through indexing. They offer constant-time access and are simple to use, making them suitable for storing fixed-size collections of similar elements.

**STEP 2. Setup:**

Create a class Employee with attributes such as employeeId, name, position, and salary to encapsulate employee data.

class Employee {

int id;

String name;

String position;

double salary;

public Employee(int id, String name, String position, double salary) {

this.id = id;

this.name = name;

this.position = position;

this.salary = salary;

}

public String toString() {

return "Employee{id=" + id + ", name='" + name + "', position='" + position + "'}";

}

public String tostring() {

return "Employee Id: " + id + ", Name: " + name + ", Position: " + position + ", Salary: " + salary;

}

}

**STEP 3. Implementation:**

Use an array to store employee records, implementing methods to add, search, traverse, and delete employees, ensuring efficient management of the data.

public class EmployeeManager {

private Employee[] employees;

private int count;

public EmployeeManager(int capacity) {

employees = new Employee[capacity];

count = 0;

}

public void addEmployee(int id, String name, String position, double salary) {

if (count >= employees.length) {

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

return;

}

employees[count++] = new Employee(id, name, position, salary);

System.out.println("Employee " + name + " added.");

}

public Employee searchEmployee(int id) {

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

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

return employees[i];

}

}

return null;

}

public void deleteEmployee(int id) {

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

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

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

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

}

employees[count--] = null;

System.out.println("Employee " + id + " deleted");

return;

}

}

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

}

public void traverseEmployee() {

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

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

}

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

EmployeeManager employeemanager = new EmployeeManager(20);

while (true) {

System.out.println("---------------------");

System.out.println("1. ADD EMPLOYEE");

System.out.println("2. SEARCH EMPLOYEE");

System.out.println("3. DELETE EMPLOYEE");

System.out.println("4. TRAVERSE EMPLOYEES");

System.out.println("----------------------");

System.out.println("Enter choice");

int ch = sc.nextInt();

sc.nextLine();

switch (ch) {

case 1: {

System.out.println("Enter employee id: ");

int id = sc.nextInt();

sc.nextLine();

System.out.println("Enter employee name: ");

String name = sc.nextLine();

System.out.println("Enter employee salary: ");

double salary = sc.nextDouble();

sc.nextLine();

System.out.println("Enter employee position: ");

String position = sc.nextLine();

employeemanager.addEmployee(id, name, position, salary);

break;

}

case 2: {

System.out.println("Enter employee id to search: ");

int id = sc.nextInt();

Employee employee = employeemanager.searchEmployee(id);

if (employee != null) {

System.out.println("Found employee: " + id);

} else {

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

}

break;

}

case 3: {

System.out.println("Enter employee id to delete: ");

int id = sc.nextInt();

employeemanager.deleteEmployee(id);

break;

}

case 4: {

employeemanager.traverseEmployee();

break;

}

default: {

System.out.println("Invalid choices");

break;

}

}

}

}

}

**4. Analysis**

* **Add**: O(1) if there's space; O(n) if resizing is needed.
* **Search**: O(n) in the worst case.
* **Traverse**: O(n) as each element is accessed once.
* **Delete**: O(n) due to shifting elements after removal.

Arrays are limited by their fixed size and inefficient resizing, making them ideal for scenarios with a known number of elements.

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

**STEP 1. Understand Linked Lists**

**Singly Linked List**

A singly linked list is a type of linked list where each node points to the next node in the sequence and the last node points to null. It allows traversal in one direction, from the head node to the end.

**Doubly Linked List**

A doubly linked list is a type of linked list where each node has two pointers: one pointing to the next node and one pointing to the previous node. This allows traversal in both directions, from the head to the end and from the end to the head.

**STEP 2. Setup**

**Task Class**

The Task class will have attributes for taskId, taskName, and status. For managing tasks using a singly linked list, we'll also have a next pointer to point to the next task in the list.

**STEP 3. Implementation**

We will implement a singly linked list to manage tasks with methods to add, search, traverse, and delete tasks.

**//Task.java:**

public class Task {

int id;

String name;

String state;

Task nextTask;

public Task(int id, String name, String state) {

this.id = id;

this.name = name;

this.state = state;

this.nextTask = null;

}

}

import java.util.\*;

public class TaskManager {

private Task firstTask;

public TaskManager() {

this.firstTask = null;

}

public void addTask(int id, String name, String state) {

Task newTask = new Task(id, name, state);

if (firstTask == null) {

firstTask = newTask;

} else {

Task currentTask = firstTask;

while (currentTask.nextTask != null) {

currentTask = currentTask.nextTask;

}

currentTask.nextTask = newTask;

}

System.out.println("Task " + name + " added.");

}

public Task searchTask(int id) {

Task currentTask = firstTask;

while (currentTask != null) {

if (currentTask.id == id) {

return currentTask;

}

currentTask = currentTask.nextTask;

}

return null;

}

public void deleteTask(int id) {

Task currentTask = firstTask;

Task previousTask = null;

while (currentTask != null) {

if (currentTask.id == id) {

if (previousTask != null) {

previousTask.nextTask = currentTask.nextTask;

} else {

firstTask = currentTask.nextTask;

}

System.out.println("Task " + id + " deleted.");

return;

}

previousTask = currentTask;

currentTask = currentTask.nextTask;

}

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

}

public void traverseTasks() {

Task currentTask = firstTask;

while (currentTask != null) {

System.out.println("Task ID: " + currentTask.id + ", Name: " + currentTask.name + ", State: " + currentTask.state);

currentTask = currentTask.nextTask;

}

}

public static void main(String[] args) {

TaskManager manager = new TaskManager();

Scanner scanner = new Scanner(System.in);

while (true) {

System.out.println(" ------------------------");

System.out.println("1. ADD TASK ");

System.out.println("2. SEARCH TASK");

System.out.println("3. DELETE TASK");

System.out.println("4. TRAVERSE TASK");

System.out.println("---------------------------");

System.out.println("Enter choice: ");

int choice = scanner.nextInt();

scanner.nextLine();

switch (choice) {

case 1: {

System.out.println("1. Add task");

System.out.println("Enter task id to be added: ");

int id = scanner.nextInt();

scanner.nextLine();

System.out.println("Enter task name to be added: ");

String name = scanner.nextLine();

System.out.println("Enter task state to be added: ");

String state = scanner.nextLine();

manager.addTask(id, name, state);

break;

}

case 2: {

System.out.println("2. Search task");

System.out.println("Enter task id to search the task: ");

int searchId = scanner.nextInt();

Task task = manager.searchTask(searchId);

if (task != null) {

System.out.println("Found task: " + task.name + " with state " + task.state);

} else {

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

}

break;

}

case 3: {

System.out.println("3. Delete task");

System.out.println("Enter task id to delete task: ");

int deleteId = scanner.nextInt();

manager.deleteTask(deleteId);

break;

}

case 4: {

System.out.println("4. Traverse tasks");

manager.traverseTasks();

break;

}

default:

System.out.println("Invalid choice. Please enter a number between 1 and 4.");

}

}

}

}

**STEP 4. Analysis**

**Time Complexity of Operations**

* **Add Task**: O(1) - Adding a task to the head of the list takes constant time.
* **Search Task**: O(n) - In the worst case, you might have to traverse the entire list to find the task.
* **Delete Task**: O(n) - In the worst case, you might have to traverse the entire list to find and delete the task.
* **Traverse Tasks**: O(n) - Traversing the list requires visiting each node once.

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

* **Dynamic Size**: Linked lists can easily grow and shrink in size by adding or removing nodes, whereas arrays have a fixed size.
* **Memory Allocation**: Linked lists use dynamic memory allocation, which can be more efficient when the number of elements is not known in advance.
* **Insertion and Deletion**: Insertion and deletion operations are more efficient in linked lists because they don't require shifting elements as in arrays.

This project showcases the implementation and management of tasks using a singly linked list in Java, highlighting the basic operations and their complexities.

**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.

**STEP 1. Understand Search Algorithms**

**Linear Search:**

* Linear search is a straightforward algorithm that checks each element of a list sequentially until the desired element is found or the list ends.
* Time Complexity: O(n), where n is the number of elements in the list.

**Binary Search:**

* Binary search is a more efficient algorithm that only works on sorted lists. It repeatedly divides the list in half, comparing the target value to the middle element and discarding the half where the target cannot be.
* Time Complexity: O(log n), where n is the number of elements in the list.

**STEP 2. Setup**

//**Book.java**

public class Book {

private int id;

private String name;

private String writer;

public Book(int id, String name, String writer) {

this.id = id;

this.name = name;

this.writer = writer;

}

public int getId() {

return id;

}

public String getName() {

return name;

}

public String getWriter() {

return writer;

}

@Override

public String toString() {

return "Book{" +

"id=" + id +

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

", writer='" + writer + '\'' +

'}';

}

}

**// LINEAR SEARCH**

import java.util.List;

public class Library {

public Book linearSearchByTitle(List<Book> bookList, String searchTitle) {

for (Book book : bookList) {

if (book.getName().equalsIgnoreCase(searchTitle)) {

return book;

}

}

return null;

}

}

**// BINARY SEARCH**

import java.util.Collections;

import java.util.Comparator;

import java.util.List;

public class Library {

public Book linearSearchByTitle(List<Book> bookList, String searchTitle) {

for (Book book : bookList) {

if (book.getName().equalsIgnoreCase(searchTitle)) {

return book;

}

}

return null;

}

public Book binarySearchByTitle(List<Book> bookList, String searchTitle) {

Collections.sort(bookList, Comparator.comparing(Book::getName));

int left = 0;

int right = bookList.size() - 1;

while (left <= right) {

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

Book midBook = bookList.get(mid);

int cmp = midBook.getName().compareToIgnoreCase(searchTitle);

if (cmp == 0) {

return midBook;

} else if (cmp < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

}

import java.util.ArrayList;

import java.util.List;

import java.util.Scanner;

public class Main {

public static void main(String[] args) {

List<Book> bookList = new ArrayList<>();

Library library = new Library();

Scanner scanner = new Scanner(System.in);

while (true) {

System.out.println("Enter the book ID (or type -1 to finish):");

int id = scanner.nextInt();

if (id == -1) break;

scanner.nextLine();

System.out.println("Enter the book name:");

String name = scanner.nextLine();

System.out.println("Enter the book writer:");

String writer = scanner.nextLine();

bookList.add(new Book(id, name, writer));

System.out.println("Book added.");

}

System.out.println("Enter the book title to search for:");

String searchTitle = scanner.nextLine();

Book foundBookLinear = library.linearSearchByTitle(bookList, searchTitle);

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

Book foundBookBinary = library.binarySearchByTitle(bookList, searchTitle);

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

scanner.close();

}

}**STEP 4. Analysis**

**Time Complexity:**

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

**Usage:**

* Linear Search:
  + Used when the dataset is small or unsorted.
  + Simple to implement and does not require the list to be sorted.
* Binary Search:
  + Used when the dataset is large and sorted.
  + More efficient than linear search for large datasets but requires initial sorting.

**EXERCISE 7: FINANCIAL FORECASTING:**

**Scenario:**

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

**STEP 1: Understand Recursive Algorithms:**

* Explain the concept of recursion and how it can simplify certain problems.

**STEP 2:Setup:**

* Create a method to calculate the future value using a recursive approach.

**STEP 3: Implementation:**

* Implement a recursive algorithm to predict future values based on past growth rates.

**STEP 4:Analysis:**

* Discuss the time complexity of your recursive algorithm.
* Explain how to optimize the recursive solution to avoid excessive computation.

**STEP 1. Understanding Recursive Algorithms**

**Recursion** is a method of solving problems where a function calls itself as a subroutine. This can simplify problems that have a natural recursive structure, such as mathematical sequences, tree traversals, and dynamic programming problems. In essence, recursion breaks down a problem into smaller sub-problems, each of which is a simpler instance of the original problem.

**Advantages of recursion:**

* Simplifies the code for problems that can be divided into similar sub-problems.
* Easy to implement and understand for problems like factorial computation, Fibonacci series, etc.

**Disadvantages of recursion:**

* Can lead to excessive memory usage due to the call stack.
* May result in performance issues for problems requiring a large number of recursive calls.

**STEP 2. Setup: Creating a Method to Calculate Future Value Using Recursion**

Let's consider a simple scenario where we have a constant growth rate and need to predict future values. We'll use the formula:

Future Value = Current Value × (1+Growth Rate)^n

where n is the number of periods.

**STEP 3. Implementation: Recursive Algorithm**

import java.util.HashMap;

import java.util.Map;

import java.util.Scanner;

public class FinancialForecast {

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

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

if (years == 0) {

return initialValue;

}

if (cache.containsKey(years)) {

return cache.get(years);

}

double futureValue = calculateFutureValue(initialValue \* (1 + rate), rate, years - 1);

cache.put(years, futureValue);

return futureValue;

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.println("Enter the current value:");

double initialValue = scanner.nextDouble();

System.out.println("Enter the growth rate (as a decimal):");

double rate = scanner.nextDouble();

System.out.println("Enter the number of periods:");

int years = scanner.nextInt();

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

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

scanner.close();

}

}**STEP 4. Analysis**

**Time Complexity:**

* The time complexity of this recursive algorithm is O(n)O(n)O(n), where nnn is the number of periods. This is because each call to calculateFutureValue results in another call with one less period, leading to a total of nnn calls.

**Optimizing the Recursive Solution:**

* To optimize the recursive solution and avoid excessive computation, we can use **memoization** or **iterative approach**.

**Memoization:** Memoization involves storing the results of expensive function calls and reusing them when the same inputs occur again. Here's how to implement it: