**1. Inventory Management System:**

import java.util.HashMap;

class Product {

private int productId;

private String productName;

private int quantity;

private 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 int getProductId() {

return productId;

}

public String getProductName() {

return productName;

}

public void setProductName(String productName) {

this.productName = productName;

}

public int getQuantity() {

return quantity;

}

public void setQuantity(int quantity) {

this.quantity = quantity;

}

public double getPrice() {

return price;

}

public void setPrice(double price) {

this.price = price;

}

@Override

public String toString() {

return "ProductID: " + productId + ", Name: " + productName + ", Quantity: " + quantity + ", Price: " + price;

}

}

class Inventory {

private HashMap<Integer, Product> products;

public Inventory() {

products = new HashMap<>();

}

public void addProduct(Product product) {

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

}

public void updateProduct(int productId, String productName, Integer quantity, Double price) {

Product product = products.get(productId);

if (product != null) {

if (productName != null) {

product.setProductName(productName);

}

if (quantity != null) {

product.setQuantity(quantity);

}

if (price != null) {

product.setPrice(price);

}

} else {

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

}

}

public void deleteProduct(int productId) {

if (products.remove(productId) == null) {

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

}

}

@Override

public String toString() {

StringBuilder sb = new StringBuilder();

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

sb.append(product.toString()).append("\n");

}

return sb.toString();

}

public static void main(String[] args) {

Inventory inventory = new Inventory();

Product product1 = new Product(1, "Laptop", 10, 999.99);

Product product2 = new Product(2, "Smartphone", 20, 499.99);

inventory.addProduct(product1);

inventory.addProduct(product2);

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

System.out.println(inventory);

inventory.updateProduct(1, null, 15, 950.00);

System.out.println("\nInventory after updating product 1:");

System.out.println(inventory);

inventory.deleteProduct(2);

System.out.println("\nInventory after deleting product 2:");

System.out.println(inventory);

}

}

**Time Complexity:**

* **Add Operation**: O(1) on average, since adding an item to a HashMap is an average O(1) operation.
* **Update Operation**: O(1) on average, since accessing and updating an item in a HashMap is O(1).
* **Delete Operation**: O(1) on average, as removing an item from a HashMap is an average O(1) operation.

**2. E-Commerce Platform System:**

import java.util.Arrays;

class Product {

private int productId;

private String productName;

private String category;

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

public int getProductId() {

return productId;

}

public String getProductName() {

return productName;

}

public String getCategory() {

return category;

}

@Override

public String toString() {

return "ProductID: " + productId + ", Name: " + productName + ", Category: " + category;

}

}

class LinearSearch {

public static Product search(Product[] products, int productId) {

for (Product product : products) {

if (product.getProductId() == productId) {

return product;

}

}

return null;

}

}

class BinarySearch {

public static Product search(Product[] products, int productId) {

int left = 0;

int right = products.length - 1;

while (left <= right) {

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

if (products[mid].getProductId() == productId) {

return products[mid];

}

if (products[mid].getProductId() < productId) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

}

public class ECommercePlatformSearch {

public static void main(String[] args) {

Product[] products = {

new Product(1, "Laptop", "Electronics"),

new Product(2, "Smartphone", "Electronics"),

new Product(3, "Table", "Furniture"),

new Product(4, "Chair", "Furniture"),

new Product(5, "Headphones", "Electronics")

};

// For binary search, we need a sorted array

Arrays.sort(products, (p1, p2) -> Integer.compare(p1.getProductId(), p2.getProductId()));

int searchId = 3;

// Linear Search

Product result = LinearSearch.search(products, searchId);

System.out.println("Linear Search Result: " + (result != null ? result : "Product not found"));

// Binary Search

result = BinarySearch.search(products, searchId);

System.out.println("Binary Search Result: " + (result != null ? result : "Product not found"));

}

}

**Time Complexity:**

* **Linear Search:** O(n)
  + Best Case: O(1) (when the element is at the beginning)
  + Average Case: O(n/2) ≈ O(n)
  + Worst Case: O(n) (when the element is at the end or not found)
* **Binary Search:** O(log n)
  + Best Case: O(1) (when the middle element is the target)
  + Average Case: O(log n)
  + Worst Case: O(log n) (when the target is not found after log n comparisons)

**Suitability:**

* **Linear Search:** Suitable for small arrays or unsorted arrays since it does not require sorting.
* **Binary Search:** More suitable for large, sorted arrays as it significantly reduces the number of comparisons needed to find the target element.

**Conclusion:** For an e-commerce platform with a large inventory, **binary search** is more suitable due to its logarithmic time complexity, provided that the array of products is kept sorted. This optimization will result in faster search operations, improving the overall performance of the platform.

**3. Sorting Customer Orders:**

import java.util.Arrays;

class Order {

private int orderId;

private String customerName;

private double totalPrice;

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

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

public int getOrderId() {

return orderId;

}

public String getCustomerName() {

return customerName;

}

public double getTotalPrice() {

return totalPrice;

}

@Override

public String toString() {

return "OrderID: " + orderId + ", CustomerName: " + customerName + ", TotalPrice: " + totalPrice;

}

}

class BubbleSort {

public static void sort(Order[] orders) {

int n = orders.length;

boolean swapped;

for (int i = 0; i < n - 1; i++) {

swapped = false;

for (int j = 0; j < n - i - 1; j++) {

if (orders[j].getTotalPrice() > orders[j + 1].getTotalPrice()) {

Order temp = orders[j];

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

orders[j + 1] = temp;

swapped = true;

}

}

if (!swapped) break; // Optimization: stop if no elements were swapped in the inner loop

}

}

}

class QuickSort {

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

if (low < high) {

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

sort(orders, low, pi - 1);

sort(orders, pi + 1, high);

}

}

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

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

int i = low - 1;

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

if (orders[j].getTotalPrice() <= pivot) {

i++;

Order temp = orders[i];

orders[i] = orders[j];

orders[j] = temp;

}

}

Order temp = orders[i + 1];

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

orders[high] = temp;

return i + 1;

}

}

public class ECommercePlatformSort {

public static void main(String[] args) {

Order[] orders = {

new Order(1, "Alice", 250.0),

new Order(2, "Bob", 150.0),

new Order(3, "Charlie", 300.0),

new Order(4, "David", 100.0),

new Order(5, "Eve", 200.0)

};

System.out.println("Original Orders:");

printOrders(orders);

// Bubble Sort

BubbleSort.sort(orders);

System.out.println("\nOrders after Bubble Sort:");

printOrders(orders);

// Shuffle the array for Quick Sort

shuffleOrders(orders);

System.out.println("\nOrders after Shuffling:");

printOrders(orders);

// Quick Sort

QuickSort.sort(orders, 0, orders.length - 1);

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

printOrders(orders);

}

private static void printOrders(Order[] orders) {

for (Order order : orders) {

System.out.println(order);

}

}

private static void shuffleOrders(Order[] orders) {

for (int i = orders.length - 1; i > 0; i--) {

int j = (int) (Math.random() \* (i + 1));

Order temp = orders[i];

orders[i] = orders[j];

orders[j] = temp;

}

}

}

**Time Complexity:**

* **Bubble Sort:**
  + Best Case: O(n)
  + Average Case: O(n^2)
  + Worst Case: O(n^2)
* **Quick Sort:**
  + Best Case: O(n log n)
  + Average Case: O(n log n)
  + Worst Case: O(n^2) (when the smallest or largest element is always chosen as the pivot)

**Comparison:**

* **Bubble Sort:** Simple to implement but inefficient for large datasets due to its quadratic time complexity. It is only efficient when the dataset is small or nearly sorted.
* **Quick Sort:** Generally much faster than Bubble Sort due to its average-case logarithmic time complexity. It is more suitable for large datasets. The worst-case performance can be mitigated by using better pivot selection strategies (e.g., choosing the median as the pivot).

**Conclusion:** **Quick Sort** is generally preferred over Bubble Sort for sorting customer orders by total price on an e-commerce platform due to its superior average-case performance and better scalability with large datasets.

**4. EmployeeManagement System:**

**Employee.java:**

import java.util.Arrays;

class Employee {

private int employeeId;

private String name;

private String position;

private 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 int getEmployeeId() {

return employeeId;

}

public String getName() {

return name;

}

public String getPosition() {

return position;

}

public double getSalary() {

return salary;

}

@Override

public String toString() {

return "EmployeeID: " + employeeId + ", Name: " + name + ", Position: " + position + ", Salary: " + salary;

}

}

class EmployeeManagementSystem {

private Employee[] employees;

private int count;

public EmployeeManagementSystem(int capacity) {

employees = new Employee[capacity];

count = 0;

}

public void addEmployee(Employee employee) {

if (count == employees.length) {

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

}

employees[count++] = employee;

}

public Employee searchEmployee(int employeeId) {

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

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

return employees[i];

}

}

return null;

}

public void deleteEmployee(int employeeId) {

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

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

employees[i] = employees[--count];

employees[count] = null;

return;

}

}

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

}

public void traverseEmployees() {

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

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

}

}

public static void main(String[] args) {

EmployeeManagementSystem ems = new EmployeeManagementSystem(2);

Employee e1 = new Employee(1, "Alice", "Manager", 75000);

Employee e2 = new Employee(2, "Bob", "Developer", 60000);

Employee e3 = new Employee(3, "Charlie", "Designer", 50000);

ems.addEmployee(e1);

ems.addEmployee(e2);

ems.addEmployee(e3);

System.out.println("All Employees:");

ems.traverseEmployees();

System.out.println("\nSearching for Employee with ID 2:");

System.out.println(ems.searchEmployee(2));

System.out.println("\nDeleting Employee with ID 1:");

ems.deleteEmployee(1);

ems.traverseEmployees();

}

}

**Time Complexity:**

* **Add Operation:** O(1) on average if there is space in the array. O(n) in worst case if the array needs to be resized.
* **Search Operation:** O(n) since it requires checking each element.
* **Traverse Operation:** O(n) as it involves visiting each element.
* **Delete Operation:** O(n) because it may require shifting elements to fill the gap.

**Limitations of Arrays:**

* **Fixed Size:** The initial size of the array is fixed. Resizing involves creating a new array and copying elements, which is costly.
* **Inefficient Insertions/Deletions:** Adding or deleting elements in the middle of the array requires shifting elements, which is inefficient (O(n)).
* **Memory Usage:** Arrays may waste memory if not fully utilized or require resizing frequently.

**5. Task Management System:**

**Task.java:**

class Task {

private int taskId;

private String taskName;

private String status;

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

this.taskId = taskId;

this.taskName = taskName;

this.status = status;

}

public int getTaskId() {

return taskId;

}

public String getTaskName() {

return taskName;

}

public String getStatus() {

return status;

}

@Override

public String toString() {

return "TaskID: " + taskId + ", TaskName: " + taskName + ", Status: " + status;

}

}

class Node {

Task task;

Node next;

public Node(Task task) {

this.task = task;

this.next = null;

}

}

class SinglyLinkedList {

private Node head;

public SinglyLinkedList() {

head = null;

}

public void addTask(Task task) {

Node newNode = new Node(task);

if (head == null) {

head = newNode;

} else {

Node current = head;

while (current.next != null) {

current = current.next;

}

current.next = newNode;

}

}

public Task searchTask(int taskId) {

Node current = head;

while (current != null) {

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

return current.task;

}

current = current.next;

}

return null;

}

public void deleteTask(int taskId) {

if (head == null) return;

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

head = head.next;

return;

}

Node current = head;

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

current = current.next;

}

if (current.next != null) {

current.next = current.next.next;

} else {

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

}

}

public void traverseTasks() {

Node current = head;

while (current != null) {

System.out.println(current.task);

current = current.next;

}

}

public static void main(String[] args) {

SinglyLinkedList taskList = new SinglyLinkedList();

Task t1 = new Task(1, "Design System", "In Progress");

Task t2 = new Task(2, "Develop Module", "Not Started");

Task t3 = new Task(3, "Test Application", "Completed");

taskList.addTask(t1);

taskList.addTask(t2);

taskList.addTask(t3);

System.out.println("All Tasks:");

taskList.traverseTasks();

System.out.println("\nSearching for Task with ID 2:");

Task searchResult = taskList.searchTask(2);

System.out.println(searchResult != null ? searchResult : "Task not found");

System.out.println("\nDeleting Task with ID 1:");

taskList.deleteTask(1);

taskList.traverseTasks();

}

}

**Time Complexity:**

* **Add Operation:** O(n) (traversing to the end of the list to add the node)
* **Search Operation:** O(n) (traversing the list to find the task)
* **Traverse Operation:** O(n) (visiting each node)
* **Delete Operation:** O(n) (finding the node to delete)

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

* **Dynamic Size:** Linked lists can grow and shrink dynamically, unlike arrays which have a fixed size.
* **Efficient Insertions/Deletions:** Inserting or deleting elements in a linked list is generally more efficient than in an array, especially when dealing with dynamic data that changes frequently.
* **Memory Utilization:** Linked lists allocate memory as needed, whereas arrays may waste memory if they are not fully utilized or may need resizing, which is costly.

**6. Library Management System:**

**Book.java:**

import java.util.Arrays;

import java.util.Comparator;

class LibraryManagementSystem {

private Book[] books;

private int count;

public LibraryManagementSystem(int capacity) {

books = new Book[capacity];

count = 0;

}

public void addBook(Book book) {

if (count == books.length) {

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

}

books[count++] = book;

}

public Book linearSearchByTitle(String title) {

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

if (books[i].getTitle().equalsIgnoreCase(title)) {

return books[i];

}

}

return null;

}

public Book binarySearchByTitle(String title) {

Arrays.sort(books, 0, count, Comparator.comparing(Book::getTitle));

int left = 0, right = count - 1;

while (left <= right) {

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

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

if (cmp == 0) {

return books[mid];

} else if (cmp < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

public void traverseBooks() {

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

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

}

}

public static void main(String[] args) {

LibraryManagementSystem lms = new LibraryManagementSystem(2);

lms.addBook(new Book(1, "The Catcher in the Rye", "J.D. Salinger"));

lms.addBook(new Book(2, "To Kill a Mockingbird", "Harper Lee"));

lms.addBook(new Book(3, "1984", "George Orwell"));

System.out.println("All Books:");

lms.traverseBooks();

System.out.println("\nLinear Search for '1984':");

Book foundBook = lms.linearSearchByTitle("1984");

System.out.println(foundBook != null ? foundBook : "Book not found");

System.out.println("\nBinary Search for 'To Kill a Mockingbird':");

foundBook = lms.binarySearchByTitle("To Kill a Mockingbird");

System.out.println(foundBook != null ? foundBook : "Book not found");

}

}

**Time Complexity:**

* **Linear Search:**
  + Best Case: O(1)
  + Average Case: O(n)
  + Worst Case: O(n)
* **Binary Search:**
  + Best Case: O(1)
  + Average Case: O(log n)
  + Worst Case: O(log n)

**Comparison:**

* **Linear Search:**
  + **Advantages:** Simple to implement, works on unsorted data, no need for preprocessing.
  + **Disadvantages:** Inefficient for large datasets due to its linear time complexity.
* **Binary Search:**
  + **Advantages:** Efficient for large datasets due to its logarithmic time complexity.
  + **Disadvantages:** Requires the dataset to be sorted, and sorting adds overhead if the data is not already sorted.

**7. Financial Forecasting:**

**FinancialForecasting.java:**

public class FinancialForecasting {

// Method to calculate future value using recursion

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

// Base case: if years is 0, return the present value

if (years == 0) {

return presentValue;

}

// Recursive case: calculate future value for one less year

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

}

public static void main(String[] args) {

double presentValue = 1000.0; // Initial amount

double annualGrowthRate = 0.05; // Annual growth rate (5%)

int years = 10; // Number of years to forecast

double futureValue = calculateFutureValue(presentValue, annualGrowthRate, years);

System.out.println("The future value after " + years + " years is: " + futureValue);

}

}

**Optimized Approach:**

public class FinancialForecasting {

// Method to calculate future value using iteration

public static double calculateFutureValueIterative(double presentValue, double growthRate, int years) {

double futureValue = presentValue;

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

futureValue \*= (1 + growthRate);

}

return futureValue;

}

public static void main(String[] args) {

double presentValue = 1000.0; // Initial amount

double annualGrowthRate = 0.05; // Annual growth rate (5%)

int years = 10; // Number of years to forecast

double futureValue = calculateFutureValueIterative(presentValue, annualGrowthRate, years);

System.out.println("The future value after " + years + " years is: " + futureValue);

}

}

**Recursive Approach:** Simple and elegant, but can be inefficient due to function call overhead and redundant calculations for large inputs.

**Iterative Approach:** More efficient for this problem as it avoids the overhead of recursive calls and is straightforward to implement.

**Memoization:** Useful for more complex problems where subproblems overlap and redundant calculations are costly.