**Exercise 1: Inventory Management System**

**Understand the Problem**

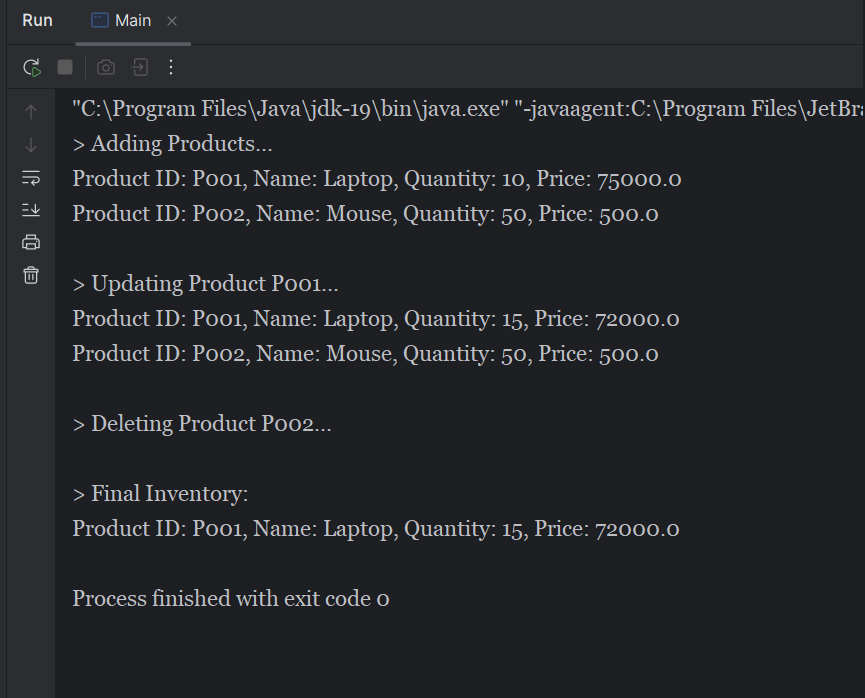
1. **Why Data Structures and Algorithms are Essential:**
   * In a warehouse, managing large inventories requires quick access, updates, and efficient storage.
   * Appropriate data structures and algorithms reduce time complexity for operations like searching, inserting, and deleting.
   * For example, using a HashMap allows constant time complexity (O(1)) for lookup operations compared to linear time (O(n)) in arrays or lists.
2. **Suitable Data Structures:**
   * **HashMap**: Efficient for key-based access (e.g., productId).
   * **ArrayList**: Maintains order, useful when sequential access is needed.
   * **TreeMap**: Maintains sorted order based on keys.

**Implementation**

**Main.java**

import java.util.HashMap;  
  
public class Main {  
 public static void main(String[] args) {  
 Inventory inv = new Inventory();  
  
 System.*out*.println("> Adding Products...");  
 Product p1 = new Product("P001", "Laptop", 10, 75000.0);  
 Product p2 = new Product("P002", "Mouse", 50, 500.0);  
 inv.addProduct(p1);  
 inv.addProduct(p2);  
 inv.printInventory();  
  
 System.*out*.println("\n> Updating Product P001...");  
 inv.updateProduct("P001", 15, 72000.0);  
 inv.printInventory();  
  
 System.*out*.println("\n> Deleting Product P002...");  
 inv.deleteProduct("P002");  
  
 System.*out*.println("\n> Final Inventory:");  
 inv.printInventory();  
 }  
}  
  
class Product {  
 String productId;  
 String productName;  
 int quantity;  
 double price;  
  
 public Product(String productId, String productName, int quantity, double price) {  
 this.productId = productId;  
 this.productName = productName;  
 this.quantity = quantity;  
 this.price = price;  
 }  
  
 public void display() {  
 System.*out*.println("Product ID: " + productId + ", Name: " + productName +  
 ", Quantity: " + quantity + ", Price: " + price);  
 }  
}  
  
class Inventory {  
 HashMap<String, Product> inventory = new HashMap<>();  
  
 public void addProduct(Product p) {  
 inventory.put(p.productId, p);  
 }  
  
 public void updateProduct(String id, int newQuantity, double newPrice) {  
 if (inventory.containsKey(id)) {  
 Product p = inventory.get(id);  
 p.quantity = newQuantity;  
 p.price = newPrice;  
 }  
 }  
  
 public void deleteProduct(String id) {  
 inventory.remove(id);  
 }  
  
 public void printInventory() {  
 for (Product p : inventory.values()) {  
 p.display();  
 }  
 }  
}

**Output**

****

**Analysis**

1. **Time Complexity:**
   * addProduct: O(1) — inserting into a HashMap.
   * updateProduct: O(1) — lookup and update.
   * deleteProduct: O(1) — removal from the HashMap.
2. **Optimization:**
   * Use indexing (e.g., productId as key) for direct access.
   * For frequently accessed products, consider a **cache layer** (e.g., LRU cache).
   * Periodically clean or optimize the HashMap size using rehashing if many deletions occur.

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

**Understand Asymptotic Notation**

1. **Big O Notation**:
   * Big O notation expresses the upper bound of an algorithm’s running time.
   * It describes how runtime grows relative to input size.
   * Examples:
     + O(1) – Constant time
     + O(n) – Linear time
     + O(log n) – Logarithmic time
     + O(n log n), O(n²), etc.
2. **Best, Average, Worst Case Scenarios for Search**:
   * **Linear Search (O(n))**:
     + Best Case: O(1) (element is first)
     + Average Case: O(n/2)
     + Worst Case: O(n) (element is last or not found)
   * **Binary Search (O(log n))**:
     + Best Case: O(1)
     + Average/Worst Case: O(log n)
     + Only works on sorted arrays.

**Implementation**

**Product.java**

class Product {

String productId;

String productName;

String category;

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

}

**SearchEngine.java**

class SearchEngine {

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

for (int i = 0; i < products.length; i++) {

if (products[i].productName.equalsIgnoreCase(key)) {

return i;

}

}

return -1;

}

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

int low = 0, high = products.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

int cmp = products[mid].productName.compareToIgnoreCase(key);

if (cmp == 0)

return mid;

else if (cmp < 0)

low = mid + 1;

else

high = mid - 1;

}

return -1;

}

}

**Main.java**

import java.util.Arrays;

import java.util.Comparator;

public class Main {

public static void main(String[] args) {

Product[] products = {

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

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

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

new Product("P004", "Camera", "Electronics")

};

System.out.println("> Searching for 'Phone' using Linear Search:");

int indexLinear = SearchEngine.linearSearch(products, "Phone");

System.out.println(indexLinear >= 0 ? "Found at index " + indexLinear : "Not found");

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

System.out.println("\n> Searching for 'Phone' using Binary Search:");

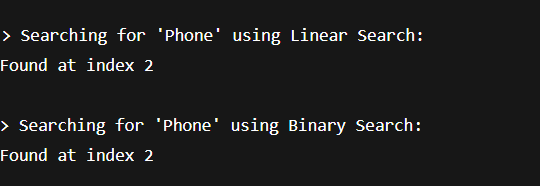
int indexBinary = SearchEngine.binarySearch(products, "Phone");

System.out.println(indexBinary >= 0 ? "Found at index " + indexBinary : "Not found");

}

}

**Output**

****

**Analysis**

1. **Time Complexities**:
   * **Linear Search**: O(n)
   * **Binary Search**: O(log n) after sorting (sorting takes O(n log n))
2. **Which to Use?**:
   * **Use Binary Search** when:
     + The list is large and sorted.
     + Frequent searches are required.
   * **Use Linear Search** when:
     + Data is unsorted.
     + Few items or one-time search.

**Exercise 3: Sorting Customer Orders**

**Understand Sorting Algorithms**

1. **Bubble Sort**:
   * Repeatedly swaps adjacent elements if they are in the wrong order.
   * Time Complexity: O(n²)
   * Simple but inefficient for large datasets.
2. **Quick Sort**:
   * Divide-and-conquer approach: selects a pivot and partitions the array.
   * Time Complexity:
     + Best/Average Case: O(n log n)
     + Worst Case (already sorted in bad pivot): O(n²)
   * Efficient and preferred in practice.

**Implementation**

**Order.java**

class Order {

String orderId;

String customerName;

double totalPrice;

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

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

public void display() {

System.out.println("Order ID: " + orderId + ", Customer: " + customerName + ", Total Price: " + totalPrice);

}

}

**Sorter.java**

class Sorter {

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;

}

}

}

}

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;

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

Order[] orders = {

new Order("O001", "Alice", 250.0),

new Order("O002", "Bob", 120.0),

new Order("O003", "Charlie", 450.0),

new Order("O004", "David", 300.0)

};

System.out.println("> Orders Sorted by Bubble Sort:");

Sorter.bubbleSort(orders);

for (Order o : orders) o.display();

orders = new Order[]{

new Order("O001", "Alice", 250.0),

new Order("O002", "Bob", 120.0),

new Order("O003", "Charlie", 450.0),

new Order("O004", "David", 300.0)

};

System.out.println("\n> Orders Sorted by Quick Sort:");

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

for (Order o : orders) o.display();

}

}

**Output**

****

**Analysis**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** | **Stable** | **In-place** |
| --- | --- | --- | --- | --- | --- |
| Bubble Sort | O(n) | O(n²) | O(n²) | Yes | Yes |
| Quick Sort | O(n log n) | O(n log n) | O(n²) | No | Yes |

**Why Quick Sort is Preferred:**

* + Much faster for large arrays due to average-case efficiency.
  + Optimized variants (like randomized or median-of-three pivot) reduce worst-case impact.

**Exercise 4: Employee Management System**

**Understand Array Representation**

1. **How Arrays Are Represented in Memory:**
   * Arrays are stored in **contiguous memory locations**.
   * Each element can be accessed in constant time using the index.
   * The array base address + index × size gives the location of any element.
2. **Advantages of Arrays:**
   * Fast access: O(1) for reading any element.
   * Simple to implement and use.
   * Efficient for fixed-size data.

**Implementation**

**Employee.java**

class Employee {

String employeeId;

String name;

String position;

double salary;

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

this.employeeId = employeeId;

this.name = name;

this.position = position;

this.salary = salary;

}

public void display() {

System.out.println("ID: " + employeeId + ", Name: " + name +", Position: " + position + ", Salary: " + salary);

}

}

**EmployeeSystem.java**

class EmployeeSystem {

Employee[] employees;

int count = 0;

public EmployeeSystem(int capacity) {

employees = new Employee[capacity];

}

public void addEmployee(Employee e) {

if (count < employees.length) {

employees[count++] = e;

} else {

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

}

}

public void searchEmployee(String id) {

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

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

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

employees[i].display();

return;

}

}

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

}

public void traverseEmployees() {

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

employees[i].display();

}

}

public void deleteEmployee(String id) {

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

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

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

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

}

employees[--count] = null;

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

return;

}

}

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

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

EmployeeSystem es = new EmployeeSystem(5);

es.addEmployee(new Employee("E001", "Alice", "Manager", 70000));

es.addEmployee(new Employee("E002", "Bob", "Developer", 55000));

es.addEmployee(new Employee("E003", "Charlie", "Designer", 48000));

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

es.traverseEmployees();

System.out.println("\n> Searching for E002:");

es.searchEmployee("E002");

System.out.println("\n> Deleting E002:");

es.deleteEmployee("E002");

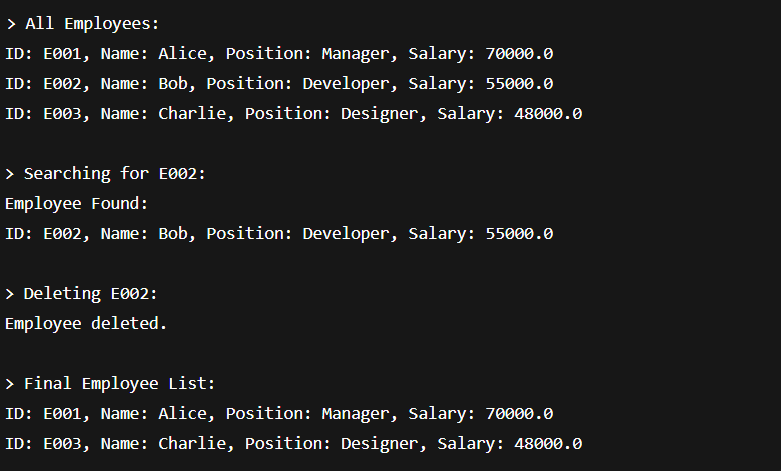
System.out.println("\n> Final Employee List:");

es.traverseEmployees();

}

}

**Output**

****

**Time Complexity Analysis**

| **Operation** | **Time Complexity** |
| --- | --- |
| Add | O(1) (amortized) |
| Search | O(n) |
| Traverse | O(n) |
| Delete | O(n) |

**Limitations of Arrays**

* Fixed size: cannot grow dynamically.
* Insertions/deletions from the middle require shifting elements.
* Searching is linear unless sorted and combined with binary search.

**Exercise 5: Task Management System**

**Understand Linked Lists**

1. **Types of Linked Lists**:
   * **Singly Linked List**: Each node points to the next.
   * **Doubly Linked List**: Each node points to both previous and next.
   * **Circular Linked List**: Last node points back to the head.
2. **Why Linked Lists?**
   * Dynamic size.
   * Easy insertions/deletions (especially in the middle or end).
   * No need to shift elements as in arrays.

**Implementation**

**Task.java**

class Task {

String taskId;

String taskName;

String status;

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

this.taskId = taskId;

this.taskName = taskName;

this.status = status;

}

public void display() {

System.out.println("Task ID: " + taskId + ", Name: " + taskName + ", Status: " + status);

}

}

**TaskNode.java**

class TaskNode {

Task task;

TaskNode next;

public TaskNode(Task task) {

this.task = task;

this.next = null;

}

}

class TaskList {

TaskNode head = null;

public void addTask(Task task) {

TaskNode newNode = new TaskNode(task);

if (head == null) {

head = newNode;

} else {

TaskNode temp = head;

while (temp.next != null) temp = temp.next;

temp.next = newNode;

}

}

public void searchTask(String id) {

TaskNode temp = head;

while (temp != null) {

if (temp.task.taskId.equals(id)) {

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

temp.task.display();

return;

}

temp = temp.next;

}

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

}

public void traverseTasks() {

TaskNode temp = head;

while (temp != null) {

temp.task.display();

temp = temp.next;

}

}

public void deleteTask(String id) {

if (head == null) return;

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

head = head.next;

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

return;

}

TaskNode prev = head;

TaskNode curr = head.next;

while (curr != null) {

if (curr.task.taskId.equals(id)) {

prev.next = curr.next;

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

return;

}

prev = curr;

curr = curr.next;

}

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

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

TaskList taskList = new TaskList();

taskList.addTask(new Task("T001", "Design UI", "Pending"));

taskList.addTask(new Task("T002", "Develop Backend", "In Progress"));

taskList.addTask(new Task("T003", "Testing", "Pending"));

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

taskList.traverseTasks();

System.out.println("\n> Searching for Task T002:");

taskList.searchTask("T002");

System.out.println("\n> Deleting Task T002:");

taskList.deleteTask("T002");

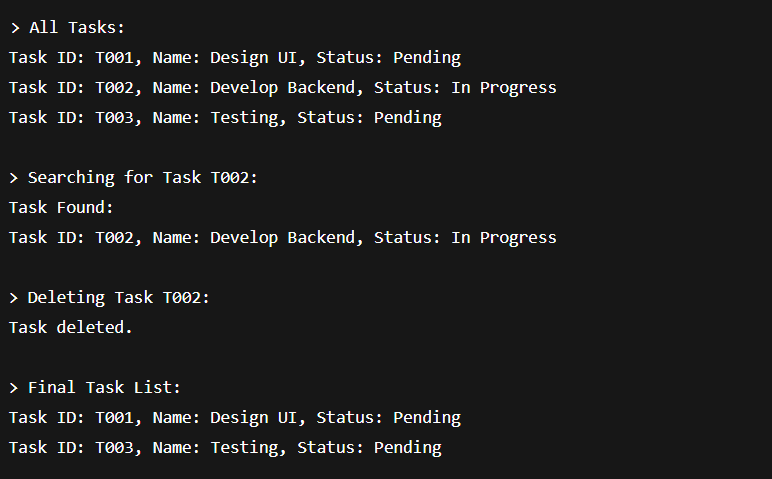
System.out.println("\n> Final Task List:");

taskList.traverseTasks();

}

}

**Output**

****

**Time Complexity Analysis**

| **Operation** | **Time Complexity** |
| --- | --- |
| Add | O(n) (to end) |
| Search | O(n) |
| Traverse | O(n) |
| Delete | O(n) |

**Advantages over Arrays**

* **Dynamic size**: No need to define fixed size upfront.
* **Efficient deletions**: No shifting required.
* Better suited for frequent insert/delete operations.

**Exercise 6: Library Management System**

**Understand Search Algorithms**

1. **Linear Search**:
   * Sequentially checks each element until a match is found.
   * Works on unsorted data.
   * Time Complexity: O(n)
2. **Binary Search**:
   * Works on **sorted** arrays.
   * Repeatedly divides the array to locate the element.
   * Time Complexity: O(log n)

**Implementation**

**Book.java**

class Book {

String bookId;

String title;

String author;

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

this.bookId = bookId;

this.title = title;

this.author = author;

}

public void display() {

System.out.println("Book ID: " + bookId + ", Title: " + title + ", Author: " + author);

}

}

**Library.java**

import java.util.Arrays;

import java.util.Comparator;

class Library {

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

for (int i = 0; i < books.length; i++) {

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

return i;

}

}

return -1;

}

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

int low = 0, high = books.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

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

if (cmp == 0) return mid;

else if (cmp < 0) low = mid + 1;

else high = mid - 1;

}

return -1;

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

Book[] books = {

new Book("B001", "The Alchemist", "Paulo Coelho"),

new Book("B002", "Clean Code", "Robert Martin"),

new Book("B003", "Effective Java", "Joshua Bloch"),

new Book("B004", "1984", "George Orwell")

};

System.out.println("> Linear Search for 'Clean Code':");

int index1 = Library.linearSearch(books, "Clean Code");

System.out.println(index1 >= 0 ? "Found at index " + index1 : "Not found");

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

System.out.println("\n> Binary Search for 'Clean Code':");

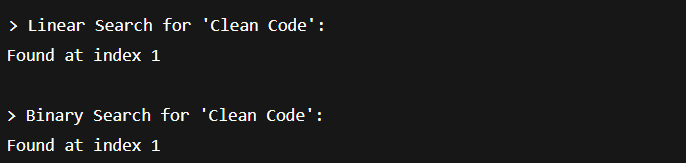
int index2 = Library.binarySearch(books, "Clean Code");

System.out.println(index2 >= 0 ? "Found at index " + index2 : "Not found");

}

}

**Output**

****

**Time Complexity Comparison**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| Linear Search | O(1) | O(n/2) | O(n) |
| Binary Search | O(1) | O(log n) | O(log n) |

**When to Use Which?**

* **Use Linear Search**:
  + When the dataset is **small** or **unsorted**.
  + One-time or infrequent search.
* **Use Binary Search**:
  + When the dataset is **large** and **sorted**.
  + Frequent lookups.

**Exercise 7: Financial Forecasting**

**Understand Recursive Algorithms**

1. **What is Recursion?**
   * A function that calls itself to solve a smaller sub-problem.
   * Base case stops recursion.
   * Useful for problems that have a repetitive, divide-and-conquer nature.
2. **Example Use in Forecasting**:
   * Predict future value based on a fixed annual growth rate.
   * Formula:  
     FutureValue = PresentValue × (1 + Rate)^n  
     This naturally fits a recursive model.

**Implementation**

**Forecast.java**

class Forecast {

public static double futureValue(double presentValue, double rate, int years) {

if (years == 0)

return presentValue;

return (1 + rate) \* futureValue(presentValue, rate, years - 1);

}

public static double futureValueMemo(double presentValue, double rate, int years, double[] memo) {

if (years == 0) return presentValue;

if (memo[years] != 0) return memo[years];

memo[years] = (1 + rate) \* futureValueMemo(presentValue, rate, years - 1, memo);

return memo[years];

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

double presentValue = 10000;

double rate = 0.10;

int years = 5;

System.out.println("> Predicting with Recursion:");

double future = Forecast.futureValue(presentValue, rate, years);

System.out.println("Future Value (Year " + years + "): ₹" + future);

System.out.println("\n> Predicting with Memoized Recursion:");

double[] memo = new double[years + 1];

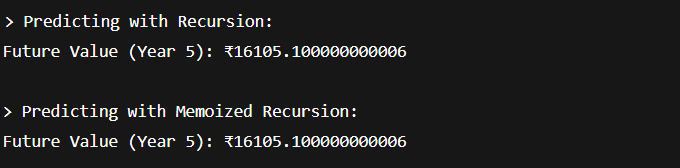
double futureMemo = Forecast.futureValueMemo(presentValue, rate, years, memo);

System.out.println("Future Value (Year " + years + "): ₹" + futureMemo);

}

}

**Output**

****

**Time Complexity Analysis**

| **Version** | **Time Complexity** | **Space Complexity** |
| --- | --- | --- |
| Recursive | O(n) | O(n) (call stack) |
| Memoized | O(n) | O(n) (array + stack) |

* Recursive solution is elegant but **can recompute** values.
* Memoization avoids repeated work and is faster for large n.

**Optimizing Further**

We can also implement this **iteratively** with O(1) space if needed:

public static double futureValueIterative(double presentValue, double rate, int years) {

for (int i = 0; i < years; i++)presentValue \*= (1 + rate);

return presentValue;

}