ALGORITHMS AND DATA STRUCTURES

**Exercise 1: Inventory Management System**

**SOLUTION:**

***Understanding the Problem:***

Algorithms and data structures assist in managing product data efficiently in big inventory systems. They facilitate fast addition, search, update, and removal of products. For instance, Java's HashMap facilitates fast access by product IDs. Without the right structures, the system becomes slow with growing inventory. Selecting the right structures such as HashMap, ArrayList, or TreeMap maintains the system organized and efficient.

***Setup:***

I am setting up this project in IntelliJ IDEA

***Implementation:***

Code:

***Product.java***

package com.inventory;

public class Product {

    private String productId;

    private String productName;

    private int quantity;

    private double price;

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

        this.productId = productId;

        this.productName = productName;

        this.quantity = quantity;

        this.price = price;

    }

    public String getProductId() { return productId; }

    public String getProductName() { return productName; }

    public int getQuantity() { return quantity; }

    public double getPrice() { return price; }

    public void setProductName(String productName) { this.productName = productName; }

    public void setQuantity(int quantity) { this.quantity = quantity; }

    public void setPrice(double price) { this.price = price; }

    public String toString() {

        return "ID: " + productId + ", Name: " + productName + ", Qty: " + quantity + ", Price: $" + price;

    }

}

***InventoryManager.java***

package com.inventory;

import java.util.HashMap;

public class InventoryManager {

    private HashMap<String, Product> inventory = new HashMap<>();

    public void addProduct(String id, String name, int qty, double price) {

        if (inventory.containsKey(id)) {

            System.out.println("Product already exists!");

        } else {

            inventory.put(id, new Product(id, name, qty, price));

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

        }

    }

    public void updateProduct(String id, int qty, double price) {

        Product p = inventory.get(id);

        if (p != null) {

            p.setQuantity(qty);

            p.setPrice(price);

            System.out.println("Product updated.");

        } else {

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

        }

    }

    public void deleteProduct(String id) {

        if (inventory.remove(id) != null) {

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

        } else {

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

        }

    }

    public void showInventory() {

        if (inventory.isEmpty()) {

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

        } else {

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

                System.out.println(p);

            }

        }

    }

    public static void main(String[] args) {

        InventoryManager im = new InventoryManager();

        im.addProduct("P001", "Monitor", 10, 15000.0);

        im.addProduct("P002", "Keyboard", 25, 1000.0);

        im.showInventory();

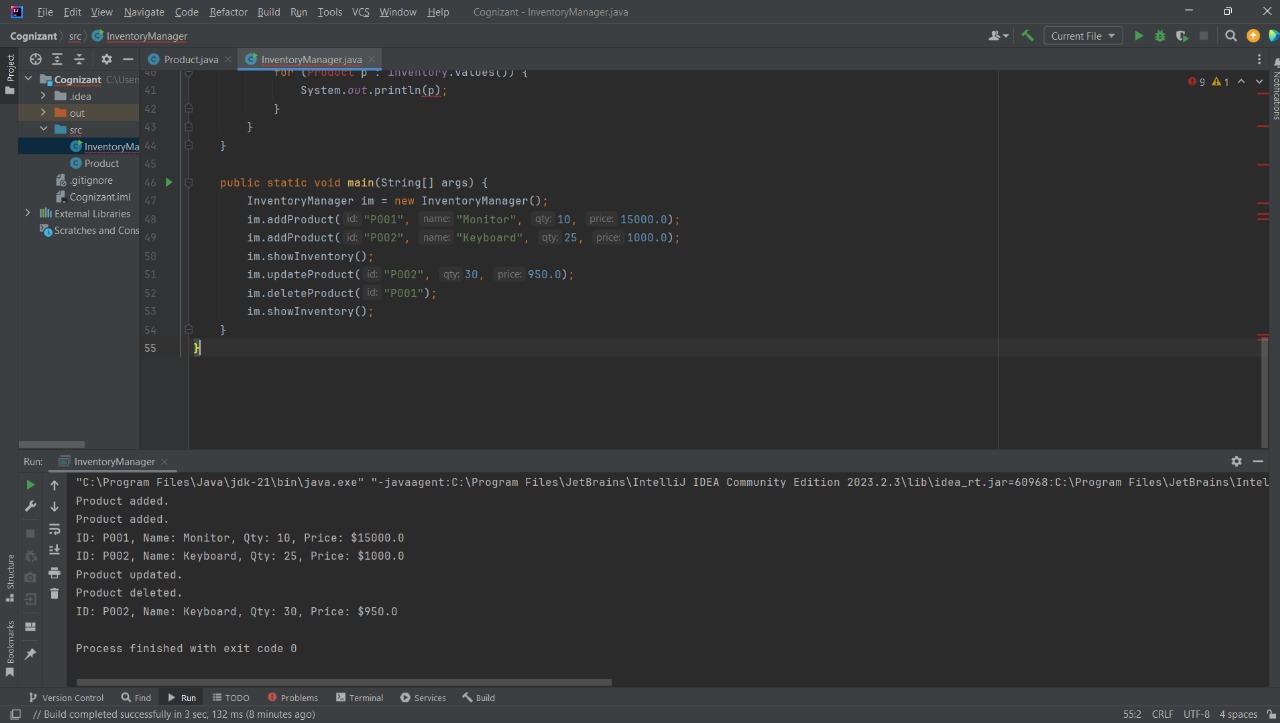
        im.updateProduct("P002", 30, 950.0);

        im.deleteProduct("P001");

        im.showInventory();

    }

}

***Output:***

***Analysis:***

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Description** | **Time Complexity** | **Explanation** |
| Add | put(key, value) | O(1) average | Direct key-based insertion |
| Update | get(key) + set() | O(1) average | Accessing and modifying existing item by key |
| Delete | remove(key) | O(1) average | Removes item directly using its key |

How to optimize this operations

* Steer clear of using complex objects as keys unless hashCode() and equals() are appropriately overridden.
* If you are comfortable with O(log n) performance but still require sorted order, use TreeMap.  
  For more complex searches, such as by price or quantity, use indexing (in databases) or filtering with streams (in Java 8+).
* Use nested maps or lists (e.g., Map>) to divide inventory into categories for grouped access.

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

**SOLUTION:**

***Understanding Asymptotic Notation:***

The time or space complexity of an algorithm as the size of the input increases is expressed in Big O notation. It indicates the effectiveness of an algorithm.

* O(1): Constant time (optimal performance)
* Linear time, or O(n)
* Logarithmic time, or O(log n), is significantly faster than O(n) for large inputs.
* O(n2), or quadratic time (slower for large data)

|  |  |  |  |
| --- | --- | --- | --- |
| **Search Type** | **Best Case** | **Average Case** | **Worst Case** |
| **Linear** | O(1) (first item) | O(n/2) to O(n) | O(n) (last or not found) |
| **Binary** | O(1) (middle) | O(log n) | O(log n) |

***Setup:***

I am setting up this project in IntelliJ IDEA

***Implementation:***

Code:

***Product.java***

package com.search;

public class Product {

private String productId;

private String productName;

private String category;

public Product(String id, String name, String category) {

this.productId = id;

this.productName = name;

this.category = category;

}

public String getProductName() {

return productName;

}

public String toString() {

return productId + " | " + productName + " | " + category;

}

}

***SearchDemo.java***

package com.search;

import java.util.Arrays;

public class SearchDemo {

// Simple linear search over the array

public static int linearSearch(Product[] list, String target) {

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

if (list[i].getProductName().equalsIgnoreCase(target)) {

return i;

}

}

return -1;

}

// Binary search assumes the list is already sorted by name

public static int binarySearch(Product[] list, String target) {

int low = 0;

int high = list.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

String midName = list[mid].getProductName();

int comparison = midName.compareToIgnoreCase(target);

if (comparison == 0) {

return mid;

} else if (comparison < 0) {

low = mid + 1;

} else {

high = mid - 1;

}

}

return -1;

}

public static void main(String[] args) {

Product[] inventory = {

new Product("P001", "Apple", "Fruit"),

new Product("P002", "Banana", "Fruit"),

new Product("P003", "Carrot", "Vegetable"),

new Product("P004", "Dates", "Dry Fruit")

};

// Sort the inventory by product name for binary search

Arrays.sort(inventory, (a, b) -> a.getProductName().compareToIgnoreCase(b.getProductName()));

String searchTerm = "Carrot";

int linearIndex = linearSearch(inventory, searchTerm);

if (linearIndex != -1) {

System.out.println("Found (Linear): " + inventory[linearIndex]);

} else {

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

}

int binaryIndex = binarySearch(inventory, searchTerm);

if (binaryIndex != -1) {

System.out.println("Found (Binary): " + inventory[binaryIndex]);

} else {

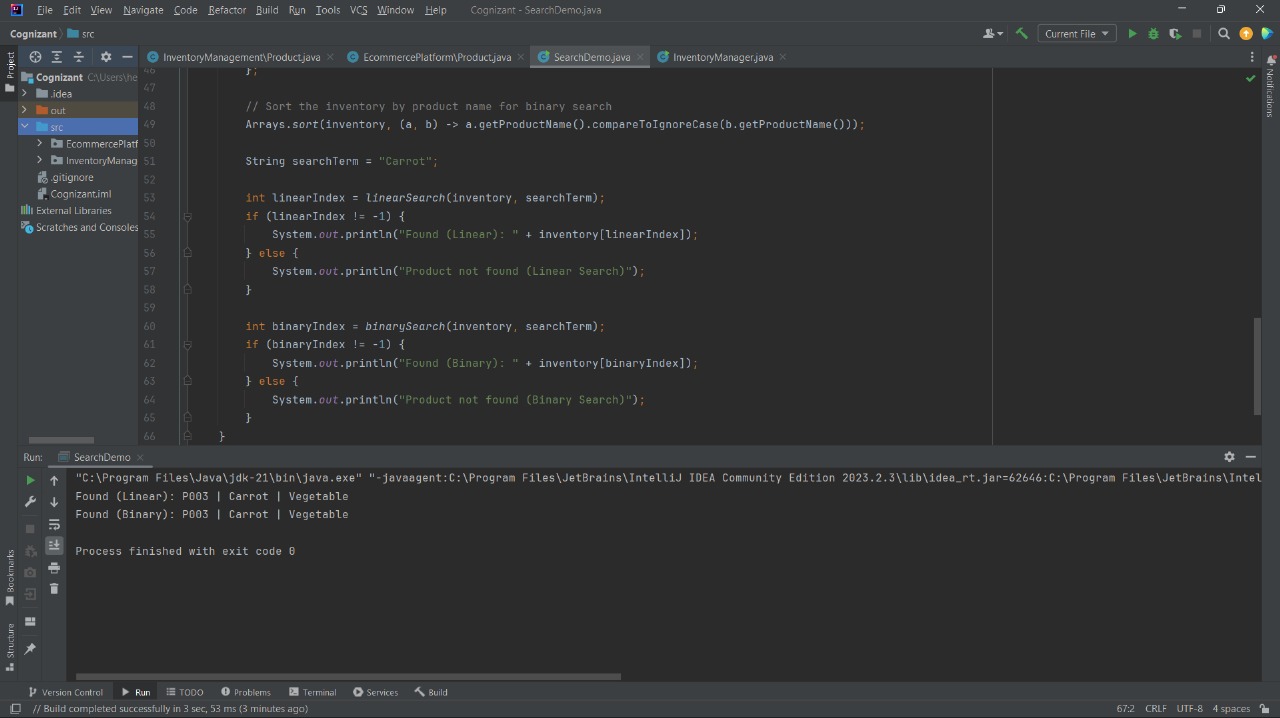
System.out.println("Product not found (Binary Search)");

}

}

}

***Output:***



***Analysis:***

Time Complexity Comparison

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | **Sorted Required** | **Suitable For** |
| **Linear Search** | O(n) | No | Small or unsorted datasets |
| **Binary Search** | O(log n) | Yes | Large, sorted datasets |

***Which is Better and Why?***

As long as the product list is sorted, binary search performs better with large datasets, making it more appropriate for an e-commerce platform where quick search is crucial. It is perfect for scalable platforms because it drastically cuts down on search time when compared to linear search. However, because linear search is straightforward and easy to use, it can be used if the data is small or unsorted.

**Exercise 3: Sorting Customer Orders**

**SOLUTION:**

***Understanding Sorting Algorithms:***

Data is sorted using sorting algorithms. Bubble Sort is a straightforward but slow method with an O(n²) time complexity that repeatedly compares and switches adjacent elements. Insertion Sort, which is O(n²) in the worst case, performs well on small or almost sorted data and creates the sorted list one item at a time. With an average time of O(n log n), Quick Sort is a divide-and-conquer algorithm that chooses a pivot and sorts around it; in the worst case, it takes O(n²). With a consistent O(n log n) time complexity and good stability, Merge Sort also employs divide and conquer, dividing the array and combining sorted portions.

***Setup:***

I am setting up this project in IntelliJ IDEA

***Implementation:***

Code:

***Order.java***

package com.orders;

public class Order {

private String orderId;

private String customerName;

private double totalPrice;

public Order(String id, String name, double price) {

this.orderId = id;

this.customerName = name;

this.totalPrice = price;

}

public double getTotalPrice() {

return totalPrice;

}

public String toString() {

return orderId + " | " + customerName + " | ₹" + totalPrice;

}

}

***OrderSorter.java***

package com.orders;

public class OrderSorter {

// Bubble Sort implementation

public static void bubbleSort(Order[] arr) {

for (int i = 0; i < arr.length - 1; i++) {

boolean swapped = false;

for (int j = 0; j < arr.length - i - 1; j++) {

if (arr[j].getTotalPrice() < arr[j + 1].getTotalPrice()) {

// Swap

Order temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

swapped = true;

}

}

// Optimization: stop if already sorted

if (!swapped) break;

}

}

// Quick Sort implementation

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

if (low < high) {

int pivotIndex = partition(arr, low, high);

quickSort(arr, low, pivotIndex - 1);

quickSort(arr, pivotIndex + 1, high);

}

}

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

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

int i = low - 1;

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

if (arr[j].getTotalPrice() >= pivot) {

i++;

// Swap

Order temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

// Swap pivot

Order temp = arr[i + 1];

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

arr[high] = temp;

return i + 1;

}

public static void main (String[] args) {

Order [] orders1 = {

new Order ("O101", "John", 4500.0),

new Order ("O102", "Alice", 11000.0),

new Order ("O103", "Meena", 3200.0),

new Order ("O104", "Rahul", 9500.0)

};

Order[] orders2 = orders1.clone(); // for quick sort

// Bubble Sort

bubbleSort(orders1);

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

for (Order o : orders1) {

System.out.println(o);

}

System.out.println();

// Quick Sort

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

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

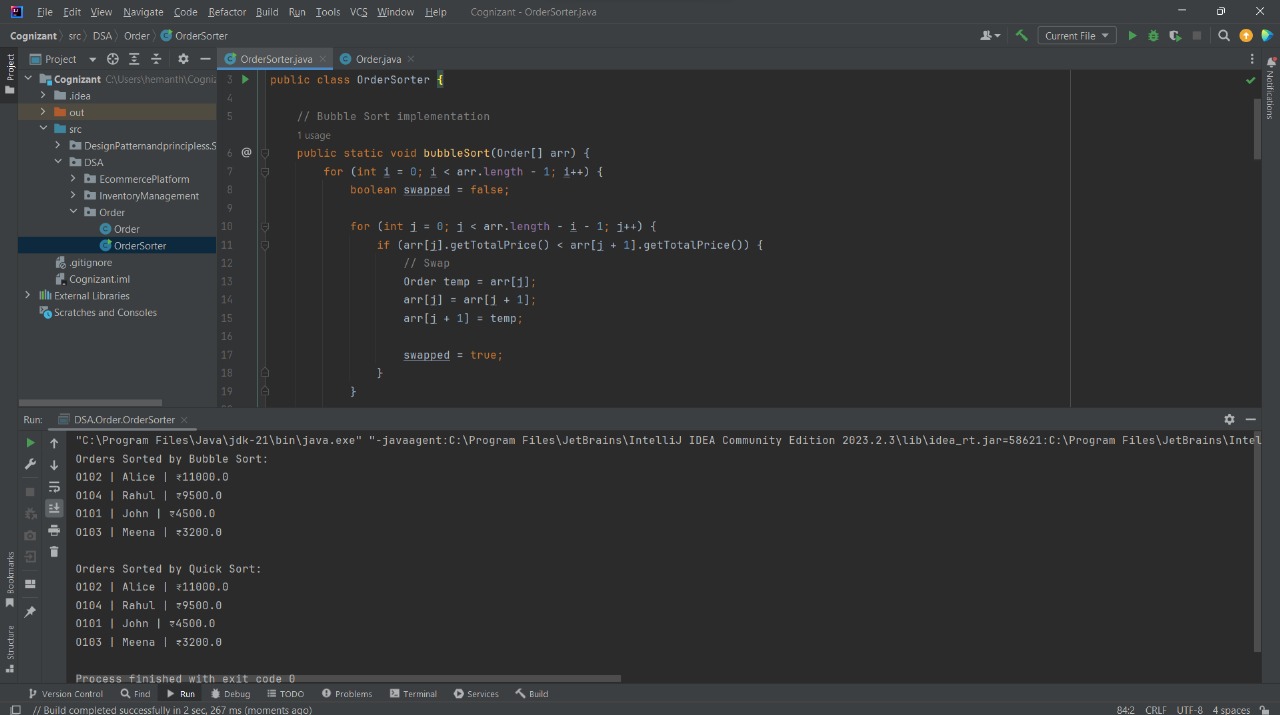
for (Order o : orders2) {

System.out.println(o);

}

}

}

***Output:***

***Analysis:***

Bubble Sort vs Quick Sort

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
| **Bubble Sort** | O(n) | O(n²) | O(n²) |
| **Quick Sort** | O (n log n) | O (n log n) | O(n²) |

***Why Quick Sort is generally preferred over Bubble Sort.***

Bubble Sort compares elements repeatedly, making it slow but simple. Quick Sort is a better option because it efficiently sorts smaller parts of the list and is faster for large data.

**Exercise 4: Employee Management System**

**SOLUTION:**

***Understand Array Representation:***

All of the elements in an array are arranged side by side in memory since they are stored in contiguous memory locations.

Advantages:

* Index-based quick access to elements (O (1) time).
* For small, fixed-size collections, it is simple to implement and effective.***Setup:***

***Setup:***

I am setting up this project in IntelliJ IDEA

***Implementation:***

Code

***Employee.java***

package com.orders;

public class Employee {

private String employeeId;

private String name;

private String position;

private double salary;

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

this.employeeId = id;

this.name = name;

this.position = position;

this.salary = salary;

}

public String getEmployeeId() {

return employeeId;

}

public String toString() {

return employeeId + " | " + name + " | " + position + " | ₹" + salary;

}

}

***EmployeeArrayManager.java***

package com.orders;

public class EmployeeArrayManager {

private Employee[] employees = new Employee[10];

private int count = 0;

public void addEmployee(Employee emp) {

if (count < employees.length) {

employees[count++] = emp;

} else {

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

}

}

public Employee searchEmployee(String id) {

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

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

return employees[i];

}

}

return null;

}

public void displayAll() {

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

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

}

}

public void deleteEmployee(String id) {

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

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

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

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

}

employees[--count] = null;

return;

}

}

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

}

public static void main(String[] args) {

EmployeeArrayManager manager = new EmployeeArrayManager();

manager.addEmployee(new Employee("E001", "Anil", "Manager", 75000));

manager.addEmployee(new Employee("E002", "Neha", "Developer", 60000));

manager.addEmployee(new Employee("E003", "Ravi", "Tester", 50000));

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

manager.displayAll();

System.out.println("\nSearching for E002:");

Employee found = manager.searchEmployee("E002");

System.out.println(found != null ? found : "Not found");

System.out.println("\nDeleting E001...");

manager.deleteEmployee("E001");

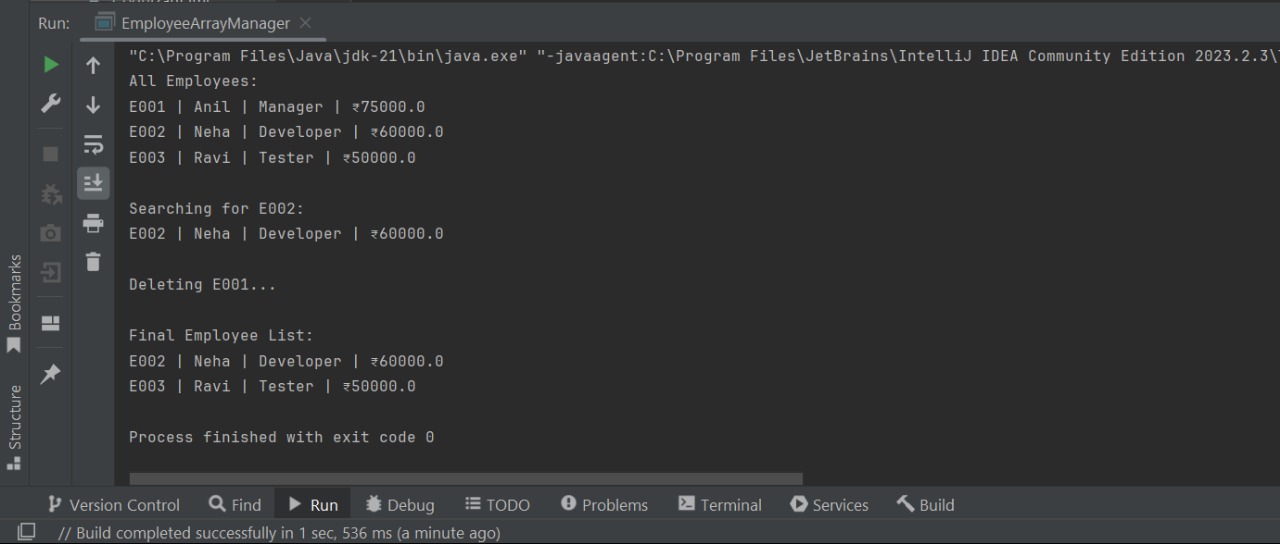
System.out.println("\nFinal Employee List:");

manager.displayAll();

}

}

***Output:***



***Analysis:***

Time Complexity Analysis:

|  |  |  |
| --- | --- | --- |
| **Operation** | **Time Complexity** | **Notes** |
| Add | O (1) | Add at end if space is available |
| Search | O(n) | Linear scan |
| Traverse | O(n) | Visit all elements |
| Delete | O(n) | Shift elements after deletion |

Limitations:

* Fixed Size: Unless you manually resize the array, you won't be able to add more elements once it is full.
* Slow Deletion/Search: It takes longer to delete or search because it requires moving or scanning elements.
* Better Options: Use ArrayList or HashMap for faster operations and dynamic size.

**Exercise 5: Task Management System**

**SOLUTION:**

**Understand Linked Lists:**

* **Singly Linked List:** Every node in a singly linked list points to the node after it. It is simple to implement and requires less memory.
* **Doubly Linked List:** Every node in a doubly linked list points to both the node before it and the node after it. uses more memory but permits traversal in both directions.

***Setup:***

I am setting up this project in IntelliJ IDEA

***Implementation:***

Code

***Task.java***

package com.orders;

public class Task {

    int taskId;

    String taskName;

    String status;

    Task next; // pointer to the next task

    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;

    }

}

***TaskLinkedList.java***

package com.orders;

public class TaskLinkedList {

    private Task head = null;

    // Add task at the end

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

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

        if (head == null) {

            head = newTask;

        } else {

            Task current = head;

            while (current.next != null) {

                current = current.next;

            }

            current.next = newTask;

        }

    }

    // Search task by ID

    public Task searchTask(int id) {

        Task current = head;

        while (current != null) {

            if (current.taskId == id) {

                return current;

            }

            current = current.next;

        }

        return null;

    }

    // Display all tasks

    public void displayTasks() {

        Task current = head;

        if (current == null) {

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

            return;

        }

        while (current != null) {

            System.out.println(current);

            current = current.next;

        }

    }

    // Delete task by ID

    public void deleteTask(int id) {

        if (head == null) return;

        if (head.taskId == id) {

            head = head.next;

            return;

        }

        Task current = head;

        while (current.next != null) {

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

                current.next = current.next.next;

                return;

            }

            current = current.next;

        }

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

    }

    public static void main(String[] args) {

        TaskLinkedList list = new TaskLinkedList();

        list.addTask(101, "Write documentation", "Pending");

        list.addTask(102, "Fix bug #302", "In Progress");

        list.addTask(103, "Review PR", "Completed");

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

        list.displayTasks();

        System.out.println("\nSearching for Task ID 102:");

        Task found = list.searchTask(102);

        System.out.println(found != null ? found : "Not found");

        System.out.println("\nDeleting Task ID 101...");

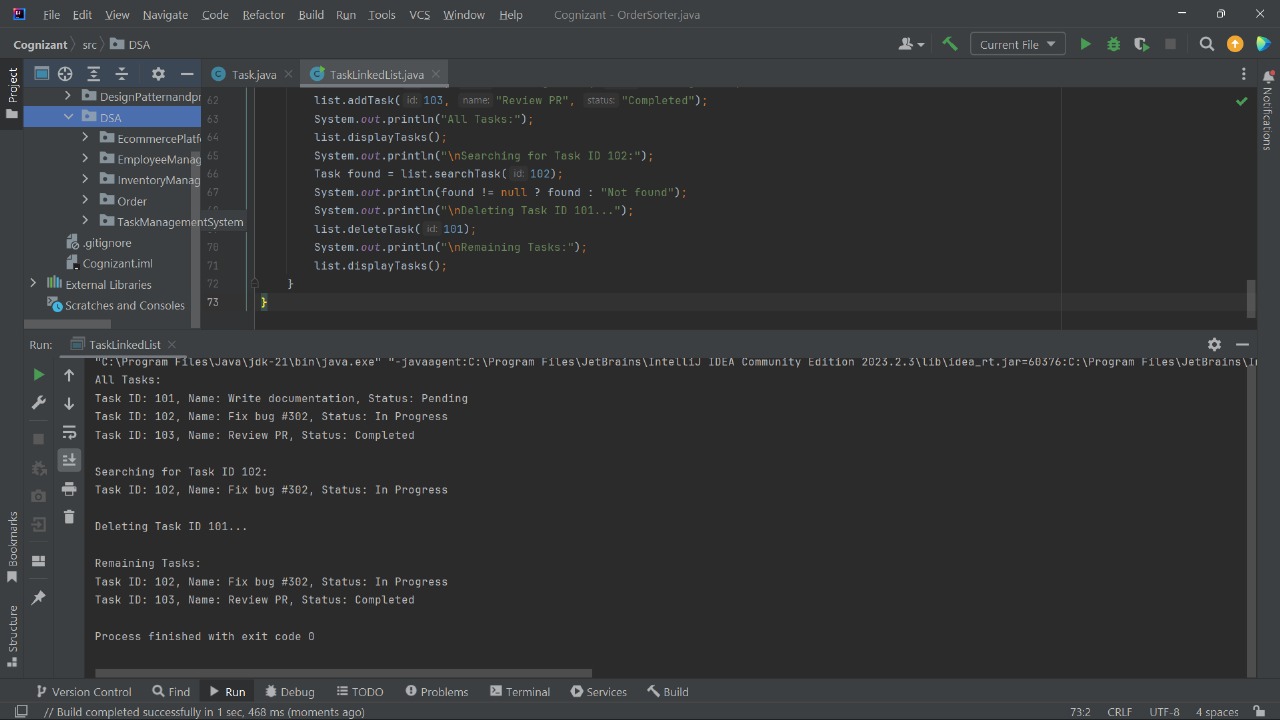
        list.deleteTask(101);

        System.out.println("\nRemaining Tasks:");

        list.displayTasks();

    }

}

***Output:***

***Analysis:***

Analysis of Time Complexity

|  |  |  |
| --- | --- | --- |
| **Operation** | **Time Complexity** | **Notes** |
| Add Task | O(n) | Traverse to the end |
| Search Task | O(n) | Linear search |
| Traverse | O(n) | Visit all elements |
| Delete Task | O(n) | Need to find the node |

Advantages of linked lists over arrays:

* Dynamic size: No fixed size needs to be declared.
* Effective insert/delete: Shifting elements is not necessary for inserting or deleting.
* Slower access, however, is not possible by index (no random access).

**Exercise 6: Library Management System**

**SOLUTION:**

***Understanding Search Algorithms:***

**Linear Search:**

A simple search method called linear search goes through each item in a list one at a time until it finds the desired value or reaches the end. It performs best with small or unsorted datasets, but as data size grows, it slows down. In the worst scenario, the time complexity is O(n), where n is the number of elements. **Binary Search:**

Binary Search is a far faster algorithm. It compares the middle element with the target after repeatedly halving the search range. It continues the search in the appropriate half based on the size of the target. With a time complexity of O(log n), this method works well with big, sorted datasets**.**

***Setup:***

I am setting up this project in IntelliJ IDEA

***Implementation:***

Code

***Book.java***

package com.orders;

public class Book {

private int bookId;

private String title;

private String author;

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

this.bookId = bookId;

this.title = title;

this.author = author;

}

public String getTitle() {

return title;

}

public String toString() {

return "ID: " + bookId + ", Title: " + title + ", Author: " + author;

}

}

***Library.java***

package com.orders;import java.util.Arrays;

import java.util.Comparator;

public class Library {

    private Book[] books;

    public Library(Book[] books) {

        this.books = books;

    }

    // Linear Search

    public Book linearSearch(String title) {

        for (Book b : books) {

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

                return b;

            }

        }

        return null;

    }

    // Binary Search (Assuming books are sorted by title)

    public Book binarySearch(String title) {

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

        while (left <= right) {

            int mid = (left + right) / 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 sortBooksByTitle() {

        Arrays.sort(books, Comparator.comparing(Book::getTitle, String.CASE\_INSENSITIVE\_ORDER));

    }

    public void showAllBooks() {

        for (Book b : books) {

            System.out.println(b);

        }

    }

}

***Main.java***

package com.orders;

public class Main {

    public static void main(String[] args) {

        Book[] bookArray = {

            new Book(101, "The Alchemist", "Paulo Coelho"),

            new Book(102, "Atomic Habits", "James Clear"),

            new Book(103, "Clean Code", "Robert C. Martin"),

            new Book(104, "The Silent Patient", "Alex Michaelides")

        };

        Library lib = new Library(bookArray);

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

        lib.showAllBooks();

        // Linear Search Example

        System.out.println("\nSearching (Linear) for 'Clean Code':");

        Book foundLinear = lib.linearSearch("Clean Code");

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

        // Sort before Binary Search

        lib.sortBooksByTitle();

        // Binary Search Example

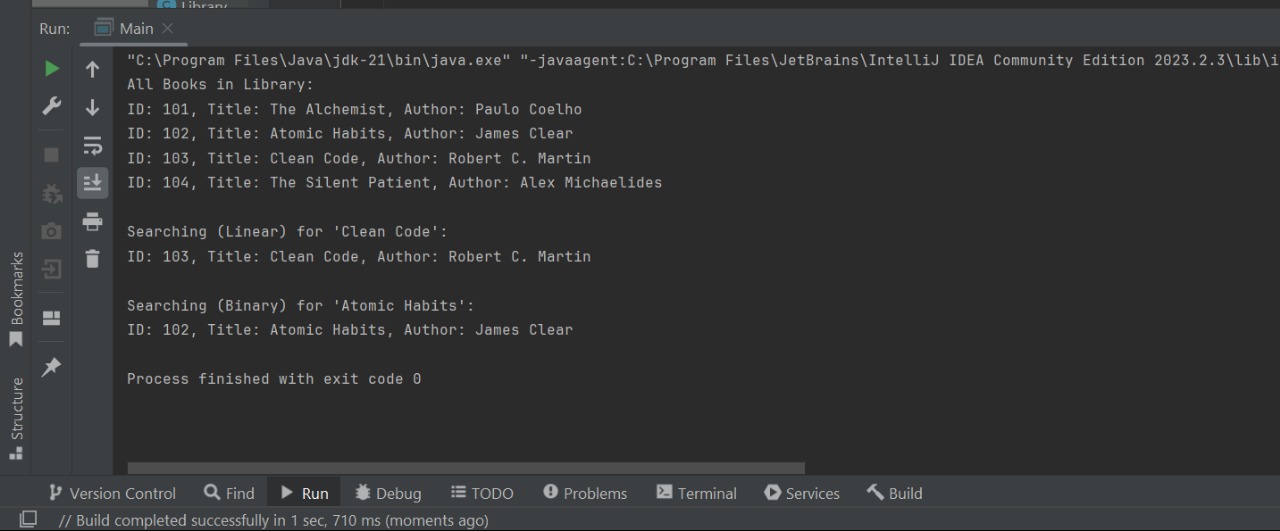
        System.out.println("\nSearching (Binary) for 'Atomic Habits':");

        Book foundBinary = lib.binarySearch("Atomic Habits");

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

    }

}

***Output:***

***Analysis***

Time Complexity Comparison

|  |  |  |
| --- | --- | --- |
| **Operation** | **Linear Search** | **Binary Search** |
| Best Case | O(1) | O(1) |
| Average/Worst | O(n) | O(log n) |
| Requirements | Unsorted | Sorted |

Which and when to use?

The best use cases for linear search are small or unsorted datasets. It is easy to implement and doesn't require any particular order. Despite being slower for large datasets, linear search is a good option if sorting the data isn't required or practical.  
However, when the dataset is large and sorted, Binary Search should be used. It only functions if the data is already sorted, but it is far quicker than linear search. Binary search is therefore the better choice if performance is important and the data can be organized.

**Exercise 7: Financial Forecasting**

**SOLUTION:**

***Understanding Recursive Algorithms:***

Recursion is a technique in which a function calls itself to resolve subproblems.

It streamlines issues that can be divided into recurring subtasks, like compound interest computation or pattern-based future value prediction.

***Setup:***

I am setting up this project in IntelliJ IDEA

Recursive Forecast Formula

futureValue(years) = currentValue × (1 + growthRate)^years

***Implementation:***

Code

***RecursiveForecaster.java***

package com.orders;

public class RecursiveForecaster {

// Recursive method to forecast future value

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

if (years == 0) {

return currentValue;

}

return forecastFutureValue(currentValue \* (1 + growthRate), growthRate, years - 1);

}

public static void main(String[] args) {

double current = 10000; // Starting value (e.g., ₹10,000)

double rate = 0.08; // 8% growth rate

int years = 5;

System.out.println("Current Value: ₹" + current);

System.out.println("Expected Growth Rate: " + (rate \* 100) + "% per year");

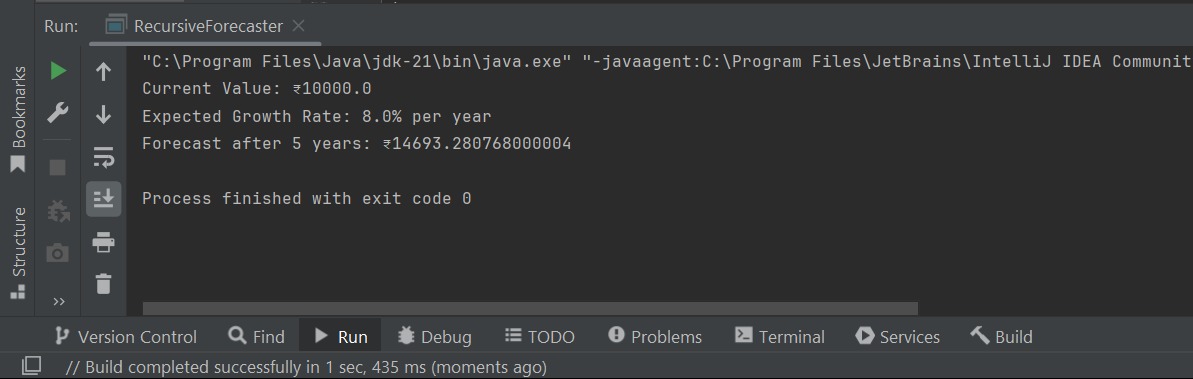
double forecast = forecastFutureValue(current, rate, years);

System.out.println("Forecast after " + years + " years: ₹" + forecast);

}

}

***Output:***



***Analysis:***

Time Complexity

Until the desired year is reached, the recursive function executes once for every year.

The time complexity, then, is O(n), where n is the number of years that need to be predicted.

Optimization

Even though the recursion is straightforward, Java stack overflow can occur when deep recursion is used (for example, for large n).

In order to prevent overcalculating:

* To improve memory usage, use an iterative method rather than recursion.
* Use dynamic programming or memoization to store results if the function (like Fibonacci) requires repeated calculations.