**Exercise 1: Inventory Management System**

**Scenario:**

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

**Steps:**

1. **Understand the Problem:**
   * Explain why data structures and algorithms are essential in handling large inventories.
   * Discuss the types of data structures suitable for this problem.

A warehouse may store thousands of products. Therefore, efficient search, update, insert, and delete operations are crucial. Poorly chosen structures will lead to slow performance and higher memory consumption.

Suitable Data Structures for this problem:

HashMap<String, Product>:-

Pros: Fast access via productId (key).

Ideal for CRUD operations (average O(1) time complexity).

ArrayList<Product>:-

Cons: Linear search for updates or deletions (O(n)).

Not ideal for key-based lookups.

So we use HashMap with productId as the key.

1. **Setup:**
   * Create a new project for the inventory management system.
2. **Implementation:**
   * Define a class Product with attributes like **productId**, **productName**, **quantity**, and **price**.
   * Choose an appropriate data structure to store the products (e.g., ArrayList, HashMap).
   * Implement methods to add, update, and delete products from the inventory.

Product.java

package 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;

}

*// Getters and Setters*

public String 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 + "] " + productName + " | Qty: " + quantity + " | Price: $" + price;

}

}

InventoryManager.java

package inventory;

import java.util.HashMap;

import java.util.Map;

public class InventoryManager {

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

    public void addProduct(Product product) {

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

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

    }

    public void updateProduct(String productId, String name, int quantity, double price) {

        Product product = inventory.get(productId);

        if (product != null) {

            product.setProductName(name);

            product.setQuantity(quantity);

            product.setPrice(price);

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

        } else {

            System.out.println("Product not found: " + productId);

        }

    }

    public void deleteProduct(String productId) {

        Product removed = inventory.remove(productId);

        if (removed != null) {

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

        } else {

            System.out.println("Product not found: " + productId);

        }

    }

    public void displayInventory() {

        if (inventory.isEmpty()) {

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

        } else {

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

                System.out.println(product);

            }

        }

    }

}

InventoryTest.java

package inventory;

public class InventoryTest {

    public static void main(String[] args) {

        InventoryManager manager = new InventoryManager();

        Product p1 = new Product("P1", "Laptop", 10, 999.00);

        Product p2 = new Product("P2", "Mouse", 20, 20.00);

        Product p3 = new Product("P3", "Keyboard", 30, 50.00);

        manager.addProduct(p1);

        manager.addProduct(p2);

        manager.addProduct(p3);

        System.out.println("\n--- Current Inventory ---");

        manager.displayInventory();

        manager.updateProduct("P2", "Wireless Mouse", 24, 24.00);

        manager.deleteProduct("P1");

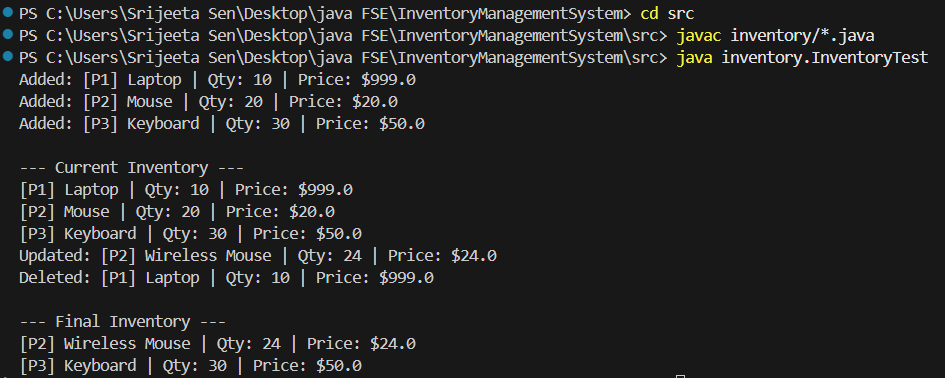
        System.out.println("\n--- Final Inventory ---");

        manager.displayInventory();

    }

}

Output



1. **Analysis:**
   * Analyze the time complexity of each operation (add, update, delete) in your chosen data structure.
   * Discuss how you can optimize these operations.

| **Operation** | **Time Complexity** | **Description** |
| --- | --- | --- |
| Add Product | O(1) | Using put(productId, product) |
| Update Product | O(1) | Using get(productId) then set |
| Delete Product | O(1) | Using remove(productId) |
| Search Product | O(1) | Fast lookup by productId |

**Optimizations:**

* Using TreeMap if sorted order is needed (O(log n)).
* Adding indexing or search filters for advanced queries.
* Using database backend for persistent, scalable storage.

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

**Scenario:**

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

**Steps:**

1. **Understand Asymptotic Notation:**
   * Explain Big O notation and how it helps in analyzing algorithms.
   * Describe the best, average, and worst-case scenarios for search operations.

Big O Notation describes how an algorithm’s runtime or space requirement grows relative to input size n. It helps in performance prediction without running code.

**Search Operations: Best, Average, Worst Case**

Search Type Best Case Average Case Worst Case

Linear Search O(1) O(n/2) ≈ O(n) O(n)

Binary Search O(1) O(log n) O(log n)

Binary Search is much faster if the data is sorted.

1. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
2. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.

Product.java

package ecommerce;

public class Product implements Comparable<Product> {

    private String productId;

    private String productName;

    private String category;

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

        this.productId = productId;

        this.productName = productName;

        this.category = category;

    }

    public String getProductId() { return productId; }

    public String getProductName() { return productName; }

    public String getCategory() { return category; }

    @Override

    public String toString() {

        return "[" + productId + "] " + productName + " (" + category + ")";

    }

    @Override

    public int compareTo(Product other) {

        return this.productName.compareToIgnoreCase(other.productName);

    }

}

SearchEngine.java

package ecommerce;

import java.util.Arrays;

public class SearchEngine {

*// Linear Search by product name*

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

        for (Product product : products) {

            if (product.getProductName().equalsIgnoreCase(name)) {

                return product;

            }

        }

        return null;

    }

*// Binary Search by product name*

    public static Product binarySearch(Product[] sortedProducts, String name) {

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

        while (left <= right) {

            int mid = (left + right) / 2;

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

            int cmp = midName.compareToIgnoreCase(name);

            if (cmp == 0)

                return sortedProducts[mid];

            else if (cmp < 0)

                left = mid + 1;

            else

                right = mid - 1;

        }

        return null;

    }

}

SearchTest.java

package ecommerce;

import java.util.Arrays;

public class SearchTest {

    public static void main(String[] args) {

        Product[] products = {

            new Product("P1", "Laptop", "Electronics"),

            new Product("P2", "Mouse", "Accessories"),

            new Product("P3", "Phone", "Electronics"),

            new Product("P4", "Charger", "Accessories"),

            new Product("P5", "Keyboard", "Accessories")

        };

*// Linear Search Test*

        System.out.println("Linear Search ");

        Product foundLinear = SearchEngine.linearSearch(products, "Phone");

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

*// Sort for Binary Search*

        Arrays.sort(products);

*// Binary Search Test*

        System.out.println("\n Binary Search ");

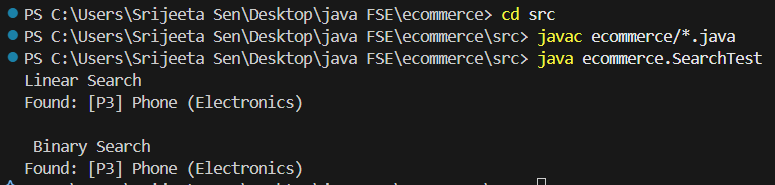
        Product foundBinary = SearchEngine.binarySearch(products, "Phone");

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

    }

}

Output



1. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.
   * Discuss which algorithm is more suitable for your platform and why.

**Time Complexity Comparison**

Algorithm Time Complexity Use Case

Linear Search O(n) Unsorted or small product lists

Binary Search O(log n) Sorted product lists, large datasets

Binary Search is faster for large datasets like in case of ecommerce platforms. However, it requires sorted data — ideally by product name or ID. If products change often (add/remove), maintaining sorted arrays can be costly. So in real systems, use search indexes (e.g., Trie, B-Trees, Hash Maps) or offload to search engines like Elasticsearch.

**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).

|  | **Time Complexity** | | |  |  |
| --- | --- | --- | --- | --- | --- |
| **Algorithm** | **Best** | **Average** | **Worst** | **Space complexity** |  |
| **Bubble Sort** | O(n) | O(n²) | O(n²) | O(1) |  |
| **Insertion Sort** | O(n) | O(n²) | O(n²) | O(1) |  |
| **Quick Sort** | O(n log n) | O(n log n) | O(n²) | O(log n) |  |
| **Merge Sort** | O(n log n) | O(n log n) | O(n log n) | O(n) |  |

1. **Setup:**
   * Create a class **Order** with attributes like **orderId**, **customerName**, and **totalPrice**.
2. **Implementation:**
   * Implement **Bubble Sort** to sort orders by **totalPrice**.
   * Implement **Quick Sort** to sort orders by **totalPrice**.

Order.java

package ecommerce;

public class Order {

private String orderId;

private String customerName;

private double totalPrice;

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

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

public double getTotalPrice() {

return totalPrice;

}

public String toString() {

return "[" + orderId + "] " + customerName + ": $" + totalPrice;

}

}

OrderSorter.java

package ecommerce;

public class OrderSorter {

*// Bubble Sort*

    public static void bubbleSort(Order[] orders) {

        int n = orders.length;

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

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

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

*// Swap*

                    Order temp = orders[j];

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

                    orders[j + 1] = temp;

                }

            }

        }

    }

*// Quick Sort*

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

        if (low < high) {

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

            quickSort(orders, low, pivotIndex - 1);

            quickSort(orders, pivotIndex + 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) { // Descending

                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;

    }

}

OrderTest.java

package ecommerce;

import java.util.Arrays;

public class OrderTest {

public static void main(String[] args) {

Order[] orders = {

new Order("O01", "Emily", 250.00),

new Order("O02", "Derek", 150.00),

new Order("O03", "Charlie", 300.00),

new Order("O04", "David", 200.00)

};

Order[] ordersBubble = Arrays.copyOf(orders, orders.length);

Order[] ordersQuick = Arrays.copyOf(orders, orders.length);

*// Bubble Sort*

System.out.println("--- Bubble Sort (Descending by totalPrice) ---");

OrderSorter.bubbleSort(ordersBubble);

for (Order o : ordersBubble) System.out.println(o);

*// Quick Sort*

System.out.println("\n--- Quick Sort (Descending by totalPrice) ---");

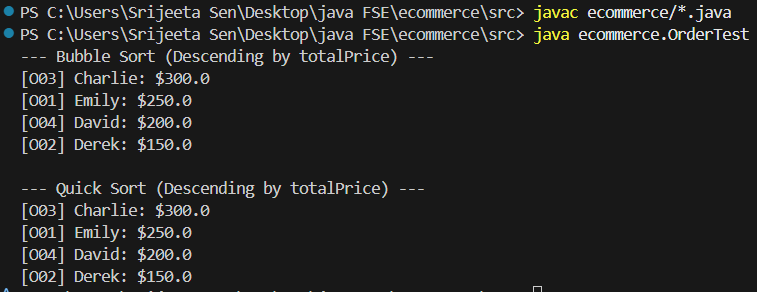
OrderSorter.quickSort(ordersQuick, 0, ordersQuick.length - 1);

for (Order o : ordersQuick) System.out.println(o);

}

}

Output



1. **Analysis:**
   * Compare the performance (time complexity) of Bubble Sort and Quick Sort.
   * Discuss why Quick Sort is generally preferred over Bubble Sort.

**Time Complexity Comparison:**

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)

Quick Sort is preferred over Bubble Sort because:

* Quick Sort is significantly faster for large datasets due to its divide-and-conquer approach.
* Bubble Sort performs too many swaps and comparisons.
* For small or nearly sorted data, Insertion Sort is often better than Bubble.

**Exercise 4: Employee Management System**

**Scenario:**

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

**Steps:**

1. **Understand Array Representation:**
   * Explain how arrays are represented in memory and their advantages.

Arrays Are Represented in Memory as:

* Arrays in Java are contiguous blocks of memory.
* Each element is accessible via an index (0-based).
* Array size is fixed at the time of creation.

Advantages of Arrays:

* Fast access (O(1)) via indexing.
* Easy to traverse sequentially.
* Memory-efficient for static-sized data

1. **Setup:**
   * Create a class Employee with attributes like **employeeId**, **name**, **position**, and **salary**.
2. **Implementation:**
   * Use an array to store employee records.
   * Implement methods to **add**, **search**, **traverse**, and **delete** employees in the array.

Employee.java

package employee;

public 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 toString() {

return "[" + employeeId + "] " + name + " - " + position + ": $" + salary;

}

}

EmployeeManager.java

package employee;

public class EmployeeManager {

    private Employee[] employees;

    private int size;

    public EmployeeManager(int capacity) {

        employees = new Employee[capacity];

        size = 0;

    }

*// Add employee*

    public void addEmployee(Employee e) {

        if (size < employees.length) {

            employees[size++] = e;

            System.out.println("Employee added: " + e);

        } else {

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

        }

    }

*// Search employee by ID*

    public Employee searchEmployee(int id) {

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

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

                return employees[i];

            }

        }

        return null;

    }

*// Traverse all employees*

    public void listEmployees() {

        System.out.println("\nAll Employees:");

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

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

        }

    }

*// Delete employee by ID*

    public boolean deleteEmployee(int id) {

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

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

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

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

                }

                employees[--size] = null;

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

                return true;

            }

        }

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

        return false;

    }

}

EmployeeTest.java

package employee;

public class EmployeeTest {

    public static void main(String[] args) {

        EmployeeManager manager = new EmployeeManager(5);

        manager.addEmployee(new Employee(001, "Emily", "Manager", 80000));

        manager.addEmployee(new Employee(002, "Suzy", "Developer", 60000));

        manager.addEmployee(new Employee(003, "Sophia", "Accountant", 45000));

        manager.listEmployees();

        System.out.println("\nSearching for employee ID 002:");

        Employee e = manager.searchEmployee(002);

        if (e != null) {

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

        } else {

            System.out.println("Not found.");

        }

        System.out.println("\nDeleting employee ID 001:");

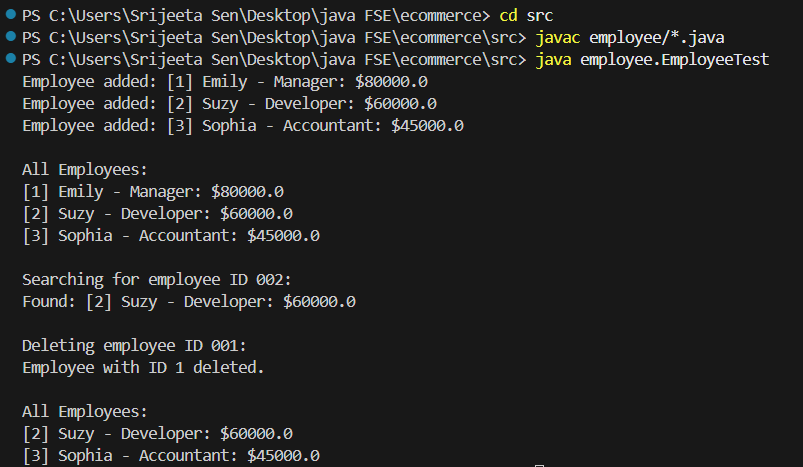
        manager.deleteEmployee(001);

        manager.listEmployees();

    }

}

Output



1. **Analysis:**
   * Analyze the time complexity of each operation (add, search, traverse, delete).
   * Discuss the limitations of arrays and when to use them.

**Time Complexity Analysis**

Operation Time Complexity

Add O(1) (amortized, at end)

Search by ID O(n) (linear scan)

Traverse O(n)

Delete by ID O(n) (due to shift)

**Limitations of Arrays**

* Fixed Size - Must define size at creation
* Inefficient Delete - Requires shifting elements (O(n))
* Inefficient Insert at Index - Also requires shifting (O(n))
* No Built-in Dynamic Growth - Needs manual resizing or switch to ArrayList

**When to Use Arrays**

* When the maximum number of elements is known.
* When random access (by index) is required.
* When you want simple memory-efficient storage.

**Exercise 5: Task Management System**

**Scenario:**

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

**Steps:**

1. **Understand Linked Lists:**
   * Explain the different types of linked lists (Singly Linked List, Doubly Linked List).

Singly Linked List: Each node contains data and a pointer to the next node. Simple, uses less memory.

Doubly Linked List: Each node contains data, a pointer to the next node, and a pointer to the previous node. Enables backward traversal but uses more memory.

1. **Setup:**
   * Create a class **Task** with attributes like **taskId**, **taskName**, and **status**.
2. **Implementation:**
   * Implement a singly linked list to manage tasks.
   * Implement methods to **add**, **search**, **traverse**, and **delete** tasks in the linked list.

Task.java

package task;

public 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 toString() {

        return "[" + taskId + "] " + taskName + " - " + status;

    }

}

TaskList.java

package task;

public class TaskList {

    private Node head;

    private class Node {

        Task task;

        Node next;

        Node(Task task) {

            this.task = task;

            this.next = 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;

        }

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

    }

*// Search for a task by ID*

    public Task searchTask(int taskId) {

        Node current = head;

        while (current != null) {

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

                return current.task;

            }

            current = current.next;

        }

        return null;

    }

*// Traverse all tasks*

    public void traverseTasks() {

        Node current = head;

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

        while (current != null) {

            System.out.println(current.task);

            current = current.next;

        }

    }

*// Delete a task by ID*

    public boolean deleteTask(int taskId) {

        if (head == null)

return false;

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

            head = head.next;

            System.out.println("Deleted task with ID " + taskId);

            return true;

        }

        Node current = head;

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

            current = current.next;

        }

        if (current.next != null) {

            System.out.println("Deleted task with ID " + taskId);

            current.next = current.next.next;

            return true;

        }

        System.out.println("Task with ID " + taskId + " not found.");

        return false;

    }

}

TestTaskList.java

package task;

public class TestTaskList {

    public static void main(String[] args) {

        TaskList taskList = new TaskList();

        taskList.addTask(new Task(1, "Design database", "Pending"));

        taskList.addTask(new Task(2, "Develop UI", "In Progress"));

        taskList.addTask(new Task(3, "Test application", "Pending"));

        taskList.traverseTasks();

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

        Task found = taskList.searchTask(2);

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

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

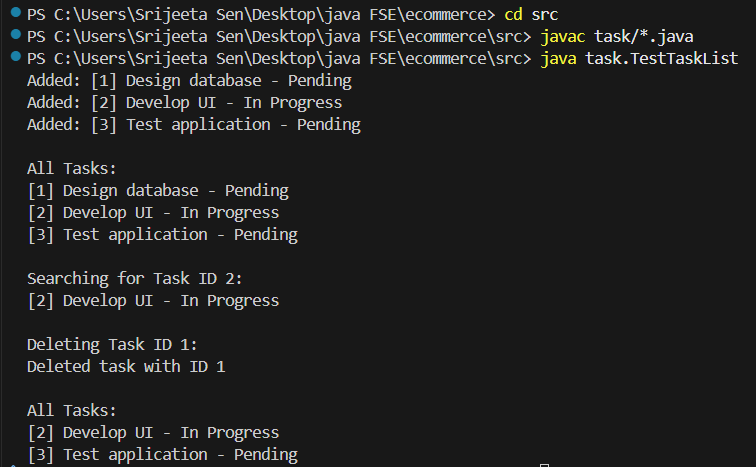
        taskList.deleteTask(1);

        taskList.traverseTasks();

    }

}

Output



1. **Analysis:**
   * Analyze the time complexity of each operation.
   * Discuss the advantages of linked lists over arrays for dynamic data.

**Operation Time Complexity**

Add Task O(n)

Search Task O(n)

Traverse Tasks O(n)

Delete Task O(n)

**Advantages of Linked Lists over Arrays**

* Dynamic Growth: No fixed size required, can grow/shrink as needed.
* Efficient Insertions and Deletions: Insertion and deletion don’t require shifting elements.
* Better for Frequently Changing Lists: Ideal when data is inserted/deleted often.

**Limitations of Linked Lists**

* No Random Access: Must traverse sequentially.
* More Memory Usage: Requires additional space for pointers.

**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.

**Linear Search:** Goes through each element until it finds the match.

Time Complexity: O(n)

**Binary Search:** Works only on sorted data. Divides the array/list into halves repeatedly until the match is found.

Time Complexity: O(log n)

1. **Setup:**
   * Create a class **Book** with attributes like **bookId**, **title**, and **author**.
2. **Implementation:**
   * Implement linear search to find books by title.
   * Implement binary search to find books by title (assuming the list is sorted).

Book.java

package bookbook;

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 int getBookId() {

        return bookId;

    }

    public String getTitle() {

        return title;

    }

    public String getAuthor() {

        return author;

    }

    public String toString() {

        return "[" + bookId + "] " + title + " by " + author;

    }

}

BookSearch.java

package bookbook;

import java.util.\*;

public class BookSearch {

*// Linear Search by Title*

    public static Book linearSearch(List<Book> books, String title) {

        for (Book book : books) {

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

                return book;

            }

        }

        return null;

    }

*// Binary Search by Title*

    public static Book binarySearch(List<Book> books, String title) {

        int low = 0;

        int high = books.size() - 1;

        while (low <= high) {

            int mid = (low + high) / 2;

            Book midBook = books.get(mid);

            int compareResult = midBook.getTitle().compareToIgnoreCase(title);

            if (compareResult == 0) {

                return midBook;

            } else if (compareResult < 0) {

                low = mid + 1;

            } else {

                high = mid - 1;

            }

        }

        return null;

    }

}

TestLibrary.java

package bookbook;

import java.util.\*;

public class TestLibrary {

    public static void main(String[] args) {

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

        books.add(new Book(1, "The Alchemist", "Paulo Coelho"));

        books.add(new Book(2, "Wuthering Heights", "Emily Bronte"));

        books.add(new Book(3, "Anna Karenina", "Leo Tolstoy"));

        books.add(new Book(4, "Moby Dick", "Herman Melville"));

        books.add(new Book(5, "Pride and Prejudice", "Jane Austen"));

*// Sort for binary search*

        books.sort(Comparator.comparing(Book::getTitle));

*// Linear Search*

        Book found1 = BookSearch.linearSearch(books, "Anna Karenina");

        System.out.println("Linear Search Result: " + (found1 != null ? found1 : "Not Found"));

*// Binary Search*

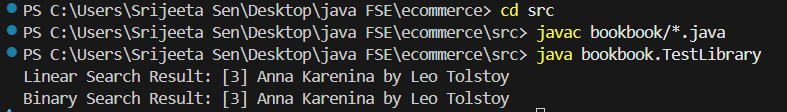
        Book found2 = BookSearch.binarySearch(books, "Anna Karenina");

        System.out.println("Binary Search Result: " + (found2 != null ? found2 : "Not Found"));

    }

}

Output



1. **Analysis:**
   * Compare the time complexity of linear and binary search.
   * Discuss when to use each algorithm based on the data set size and order.

| **Algorithm** | **Time Complexity** | **Usage** |
| --- | --- | --- |
| Linear Search | O(n) | Best for small or unsorted datasets. |
| Binary Search | O(log n) | Best for large, sorted datasets. |

**Exercise 7: Financial Forecasting**

**Scenario:**

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

**Steps:**

1. **Understand Recursive Algorithms:**
   * Explain the concept of recursion and how it can simplify certain problems.

Recursion is when a method calls itself to solve smaller instances of a problem until it reaches a base case. It's especially useful for problems with a repetitive structure, like forecasting a value repeatedly over a period.

1. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
2. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.

FinancialForecasting.java

public class FinancialForecasting {

*// Recursive method*

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

if (years == 0) {

return principal; *// Base case*

}

*// Recursive case*

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

}

public static void main(String[] args) {

double principal = 1000.0;

double rate = 0.05; *// 5% annual growth*

int years = 10;

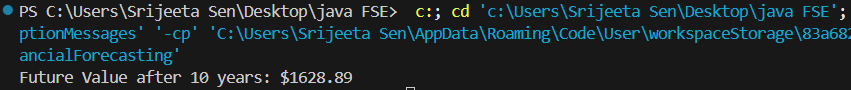
double result = futureValue(principal, rate, years);

System.out.println("Future Value after " + years + " years: $" + String.format("%.2f", result));

}

}

Output



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

**Time Complexity**

* The function makes **one call per year** until years == 0.
* Time Complexity = **O(n)**, where n = years.

**Optimization**

For long forecasting periods (large n), recursion can cause:

* **Stack Overflow** for very large n.
* **Repetitive Multiplication** that can be computed in O(1) using math.

We can also use the direct formula:

FV=PV×(1+rate) years