WEEK 1: DATA STRUCTURES AND ALGORITHMS

# Exercise 1: Inventory Management System

**Scenario:** To develop an inventory management system for a warehouse with efficient data storage and retrieval.

**Understand the Problem:**

**Why Data Structures and Algorithms are Essential in Handling Large Inventories**

Data structures and algorithms are essential for managing large inventories due to several key factors:

* Efficiency:Appropriate data structures facilitate efficient storage, retrieval, and manipulation of data. Algorithms ensure that operations like searching, adding, updating, and deleting items are performed optimally.
* Scalability: Efficient data structures and algorithms help maintain performance as the inventory size grows, preventing slowdowns and ensuring smooth operations.
* Memory Management:Optimized data structures make effective use of memory, preventing wastage and enabling the system to handle large datasets efficiently.
* Complexity Management: They simplify the inherent complexity of managing large datasets, making the system easier to implement and maintain.**Discuss the types of data structures suitable for this problem.**

The suitable types of data structures for an Inventory Management System are as follows:

* \*\*ArrayList:\*\* Provides dynamic arrays that can grow as needed. It is good for scenarios where the number of items is variable, and accessing and iterating through the list is frequent.
* \*\*Binary Search Tree (BST):\*\* Allows for sorted storage and efficient in-order traversal. It is suitable for scenarios requiring ordered data.
* \*\*HashMap:\*\* Also known as a Hash Table, it provides efficient key-value pair storage. It is excellent for quick lookups, additions, and deletions based on unique identifiers like product IDs.
* \*\*Linked List:\*\* Useful for constant-time insertions and deletions. However, it provides linear-time access, which might be a drawback for large datasets.

Among these, choosing the HashMap would be more appropriate in terms of time complexity for insertion, deletion, and updating operations.

**Setup:**

import java.util.HashMap;

class Item {

    private String itemId;

    private String itemName;

    private int stockQuantity;

    private double unitPrice;

    // Constructor to initialize an Item instance

    public Item(String itemId, String itemName, int stockQuantity, double unitPrice) {

        this.itemId = itemId;

        this.itemName = itemName;

        this.stockQuantity = stockQuantity;

        this.unitPrice = unitPrice;

    }

    // Getter for the item ID

    public String getItemId() {

        return itemId;

    }

    // Setter for the item ID

    public void setItemId(String itemId) {

        this.itemId = itemId;

    }

    // Getter for the item name

    public String getItemName() {

        return itemName;

    }

    // Setter for the item name

    public void setItemName(String itemName) {

        this.itemName = itemName;

    }

    // Getter for the stock quantity

    public int getStockQuantity() {

        return stockQuantity;

    }

    // Setter for the stock quantity

    public void setStockQuantity(int stockQuantity) {

        this.stockQuantity = stockQuantity;

    }

    // Getter for the unit price

    public double getUnitPrice() {

        return unitPrice;

    }

    // Setter for the unit price

    public void setUnitPrice(double unitPrice) {

        this.unitPrice = unitPrice;

    }

    @Override

    public String toString() {

        return "Item ID: " + itemId + "\n" +

               "Item Name: " + itemName + "\n" +

               "Stock Quantity: " + stockQuantity + "\n" +

               "Unit Price: " + unitPrice + "\n";

    }

}

class StockController {

    private HashMap<String, Item> itemCatalog;

    // Constructor to initialize the StockController with an empty item catalog

    public StockController() {

        itemCatalog = new HashMap<>();

    }

    // Adds an item to the catalog

    public void addItem(Item item) {

        if (item != null) {

            itemCatalog.put(item.getItemId(), item);

            System.out.println("Item added: " + item.getItemName());

        } else {

            System.out.println("Cannot add null item.");

        }

    }

    // Updates an item in the catalog

    public void updateItem(Item item) {

        if (item != null && itemCatalog.containsKey(item.getItemId())) {

            itemCatalog.put(item.getItemId(), item);

            System.out.println("Item updated: " + item.getItemName());

        } else if (item == null) {

            System.out.println("Cannot update null item.");

        } else {

            System.out.println("Item with ID " + item.getItemId() + " not found.");

        }

    }

    // Removes an item from the catalog

    public void removeItem(String itemId) {

        if (itemId != null && itemCatalog.containsKey(itemId)) {

            itemCatalog.remove(itemId);

            System.out.println("Item removed with ID: " + itemId);

        } else if (itemId == null) {

            System.out.println("Item ID cannot be null.");

        } else {

            System.out.println("Item with ID " + itemId + " not found.");

        }

    }

    // Displays all items in the catalog

    public void showAllItems() {

        if (itemCatalog.isEmpty()) {

            System.out.println("No items in the catalog.");

        } else {

            for (Item item : itemCatalog.values()) {

                System.out.println(item);

            }

        }

    }

    public static void main(String[] args) {

        StockController controller = new StockController();

        // Adding items to the catalog

        Item item1 = new Item("I001", "Item\_One", 50, 999.99);

        Item item2 = new Item("I002", "Item\_Two", 30, 899.99);

        Item item3 = new Item("I003", "Item\_Three", 100, 349.99);

        Item item4 = new Item("I004", "Item\_Four", 25, 1249.99);

        Item item5 = new Item("I005", "Item\_Five", 15, 2399.99);

        controller.addItem(item1);

        controller.addItem(item2);

        controller.addItem(item3);

        controller.addItem(item4);

        controller.addItem(item5);

        // Displaying all items

        controller.showAllItems();

        // Updating an item

        Item updatedItem1 = new Item("I001", "Item\_One", 60, 949.99);

        controller.updateItem(updatedItem1);

        // Displaying all items after update

        controller.showAllItems();

        // Removing an item

        controller.removeItem("I002");

        // Displaying all items after removal

        controller.showAllItems();

    }

}

**Time Complexity Analysis:**

1. **Add Product:**

* Time Complexity: O(1)
  + Inserting a product into a HashMap is O(1) due to the constant time complexity of hash-based data structures.

1. **Update Product:**
   * Time Complexity: O(1)
   * Updating a product in a HashMap is also O(1) since it involves accessing the element by key and replacing the value.
2. **Delete Product:**
   * Time Complexity: O(1)
   * Removing a product from a HashMap is O(1) as it involves finding the element by key and deleting it.

**Optimization:**

To optimize the inventory management system, we should ensure that the `HashMap` is sized properly to avoid frequent rehashing and adjust the load factor for a balance between performance and memory usage. For concurrent access, using `ConcurrentHashMap` can help prevent synchronization issues and ensure thread-safe operations.

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

**Scenario:** To work on the search functionality of an e-commerce platform with optimized performance.

**Understand Asymptotic Notation:**

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

Big O notation is a mathematical representation used to describe the upper bound of an algorithm's running time or space requirements in terms of the size of the input data. It also helps in analyzing the efficiency of algorithms by providing an approximation of the worst-case scenario in terms scales as the input size increases. This allows developers to predict performance and make informed decisions about which algorithms to use.

Big O Notation helps in analyzing the Algorithms

* **Algorithm Comparison**: Standardizes efficiency comparison, e.g., O(n log n) vs. O(n^2).
* **Scalability**: Assesses how well an algorithm handles larger inputs.
* **Performance Prediction**: Predicts how an algorithm scales with input size, guiding suitability for large datasets.
* **Worst-Case Analysis**: Ensures the system can handle the algorithm's maximum resource needs.
* **Optimization**: Guides code optimization by highlighting less efficient algorithms.

**Describe the best, average, and worst-case scenarios for search operations.**

**Best, Average, and Worst-Case Scenarios for Search Operations**

1. **Best Case**: This is the best-case scenario where the search operation completes in the shortest possible time, usually when the desired element is at the beginning of the collection.
2. **Average Case**: The scenario of average case, represents a typical run where the position of the desired element is uniformly distributed in the list of elements.
3. **Worst Case**: The scenario of worst case is where the search operation takes the longest time, this is usually considered when the desired element is at the end of the collection or not present in the list at all.

**Setup:**

import java.util.Arrays;

import java.util.Comparator;

// Class representing a product with an ID, name, and type

class Product {

    private String id;

    private String name;

    private String type;

    // Constructor to initialize a Product instance

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

        this.id = id;

        this.name = name;

        this.type = type;

    }

    // Getter for the product ID

    public String getId() {

        return id;

    }

    // Getter for the product name

    public String getName() {

        return name;

    }

    // Getter for the product type

    public String getType() {

        return type;

    }

    @Override

    public String toString() {

        return "ID: " + id + ", Name: " + name + ", Type: " + type;

    }

}

// Class containing search algorithms for products

class SearchAlgorithms {

    // Performs a linear search to find a product by its name

    public static Product findByNameLinear(Product[] productArray, String nameToFind) {

        for (Product product : productArray) {

            if (product.getName().equalsIgnoreCase(nameToFind)) {

                return product;

            }

        }

        return null; // Product not found

    }

    // Performs a binary search to find a product by its name

    static Product findByNameBinary(Product[] productArray, String nameToFind) {

        // Sort products based on their names to enable binary search

        Arrays.sort(productArray, Comparator.comparing(Product::getName));

        int start = 0;

        int end = productArray.length - 1;

        while (start <= end) {

            int middle = start + (end - start) / 2;

            int comparisonResult = productArray[middle].getName().compareToIgnoreCase(nameToFind);

            if (comparisonResult == 0) {

                return productArray[middle]; // Product found

            } else if (comparisonResult < 0) {

                start = middle + 1;

            } else {

                end = middle - 1;

            }

        }

        return null; // Product not found

    }

}

// Main class to demonstrate the usage of Product and SearchAlgorithms classes

class Main {

    public static void main(String[] args) {

        Product[] products = {

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

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

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

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

            new Product("P005", "Keyboard", "Accessories")

        };

        // Demonstrating linear search to find a product by name

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

        String nameToSearch = "Tablet";

        Product resultProduct = SearchAlgorithms.findByNameLinear(products, nameToSearch);

        if (resultProduct != null) {

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

        } else {

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

        }

        // Demonstrating binary search to find a product by name

        System.out.println("\nBinary Search:");

        nameToSearch = "Monitor";

        resultProduct = SearchAlgorithms.findByNameBinary(products, nameToSearch);

        if (resultProduct != null) {

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

        } else {

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

        }

    }

}

**Time Complexity:**

**Linear Search:**

* Best Case: O(1) (when the element is at the beginning)
* Average Case: O(n)
* Worst Case: O(n)

**Binary Search:**

* Best Case: O(1) (when the element is at the middle)
* Average Case: O(log n)
* Worst Case: O(log n)

**Suitable Algorithm for this Platform:**

Binary search is more suitable for the e-commerce platform because it has a lower time complexity of O(log n) compared to linear search O(n) for large datasets. However, it requires the dataset to be sorted. If the dataset is not sorted or frequently updated, linear search might be simpler to implement initially but less efficient for larger datasets.

# Exercise 3: Sorting Customer Orders

**Scenario:** To sort customer orders by their total price on an e-commerce platform which helps in prioritizing high-value orders.

**Understand Sorting Algorithms:**

**Bubble Sort**

Bubble Sort is a simple comparison-based sorting algorithm. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted.

* **Time Complexity**: O(n^2) in the average and worst case, O(n) in the best case
* **Space Complexity**: O(1)
* **Stability**: Stable

**Insertion Sort**

Insertion Sort builds the final sorted array one item at a time. It takes each element from the input and inserts it into the correct position within the already sorted part of the array.

* **Time Complexity**: O(n^2) in the average and worst case, O(n) in the best case
* **Space Complexity**: O(1)
* **Stability**: Stable

**Quick Sort**

Quick Sort is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

* **Time Complexity**: O(n log n) on average, O(n^2) in the worst case.
* **Space Complexity**: O(log n)
* **Stability**: Not stable

**Merge Sort**

Merge Sort is also a divide-and-conquer algorithm. It divides the array into two halves, recursively sorts them, and then merges the two sorted halves.

* **Time Complexity**: O(n log n)
* **Space Complexity**: O(n)
* **Stability**: Stable

**Setup:**

**Create a class Order**

// Represents an order with an ID, customer name, and total cost

class Sorting\_Customer\_Orders {

    private String id;

    private String customer;

    private double cost;

    // Constructor to initialize an Order instance

    public Sorting\_Customer\_Orders(String id, String customer, double cost) {

        this.id = id;

        this.customer = customer;

        this.cost = cost;

    }

    // Getter for the order ID

    public String getId() {

        return id;

    }

    // Getter for the customer name

    public String getCustomer() {

        return customer;

    }

    // Getter for the total cost

    public double getCost() {

        return cost;

    }

    @Override

    public String toString() {

        return "Order ID: " + id + ", Customer: " + customer + ", Total Cost: $" + cost;

    }

}

// Contains sorting algorithms to sort an array of orders

class SortingAlgorithms {

    // Sorts an array of orders using bubble sort based on total cost

    public static void bubbleSortOrders(Sorting\_Customer\_Orders[] 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].getCost() > orders[j + 1].getCost()) {

                    // Swap orders[j] with orders[j + 1]

                    Sorting\_Customer\_Orders temp = orders[j];

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

                    orders[j + 1] = temp;

                }

            }

        }

    }

    // Sorts an array of orders using quick sort based on total cost

    public static void quickSortOrders(Sorting\_Customer\_Orders[] orders, int low, int high) {

        if (low < high) {

            int partitionIndex = partitionOrders(orders, low, high);

            quickSortOrders(orders, low, partitionIndex - 1);

            quickSortOrders(orders, partitionIndex + 1, high);

        }

    }

    // Partitions the array for quick sort

    private static int partitionOrders(Sorting\_Customer\_Orders[] orders, int low, int high) {

        double pivotValue = orders[high].getCost();

        int i = (low - 1);

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

            if (orders[j].getCost() <= pivotValue) {

                i++;

                // Swap orders[i] with orders[j]

                Sorting\_Customer\_Orders temp = orders[i];

                orders[i] = orders[j];

                orders[j] = temp;

            }

        }

        // Swap the pivot element with the element at i + 1

        Sorting\_Customer\_Orders temp = orders[i + 1];

        orders[i + 1] = orders[high];

        orders[high] = temp;

        return i + 1;

    }

}

// Demonstrates sorting algorithms on an array of orders

class Main {

    public static void main(String[] args) {

        Sorting\_Customer\_Orders[] ordersArray = {

            new Sorting\_Customer\_Orders("O001", "Alice", 250.50),

            new Sorting\_Customer\_Orders("O002", "Bob", 150.75),

            new Sorting\_Customer\_Orders("O003", "Charlie", 300.10),

            new Sorting\_Customer\_Orders("O004", "David", 175.20),

            new Sorting\_Customer\_Orders("O005", "Eve", 210.80)

        };

        System.out.println("Before Bubble Sort:");

        displayOrders(ordersArray);

        // Perform bubble sort and display the sorted orders

        SortingAlgorithms.bubbleSortOrders(ordersArray);

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

        displayOrders(ordersArray);

        // Reset orders array

        ordersArray = new Sorting\_Customer\_Orders[]{

            new Sorting\_Customer\_Orders("O001", "Alice", 250.50),

            new Sorting\_Customer\_Orders("O002", "Bob", 150.75),

            new Sorting\_Customer\_Orders("O003", "Charlie", 300.10),

            new Sorting\_Customer\_Orders("O004", "David", 175.20),

            new Sorting\_Customer\_Orders("O005", "Eve", 210.80)

        };

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

        displayOrders(ordersArray);

        // Perform quick sort and display the sorted orders

        SortingAlgorithms.quickSortOrders(ordersArray, 0, ordersArray.length - 1);

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

        displayOrders(ordersArray);

    }

    // Displays the details of all orders in the array

    public static void displayOrders(Sorting\_Customer\_Orders[] orders) {

        for (Sorting\_Customer\_Orders order : orders) {

            System.out.println(order);

        }

    }

}

**Analysis**

**Time Complexity Comparison**

* 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)

**Quick Sort is Preferred Over Bubble Sort**

Quick Sort is generally preferred over Bubble Sort because it has a much better average-case time complexity of O(n log n) compared to Bubble Sort's O(n^2).

Even though Quick Sort can degrade to O(n^2) in the worst case, this can be mitigated with good pivot selection strategies, such as choosing the median or using randomization.

Quick Sort also tends to have better cache performance and is more efficient in practice, making it more suitable for sorting large datasets on an e-commerce platform.

# Exercise 4: Employee Management System

**Scenario:** To develop an employee management system for a company and efficiently manage employee records.

**Understand Array Representation:**

**Arrays are Represented in Memory**

Arrays are a fundamental data structure in Java, where elements are stored in contiguous memory locations. This arrangement provides efficient indexing and quick access to elements.

* Contiguous Memory Allocation: Elements are stored in adjacent memory blocks, allowing direct access via an index.
* Fixed Size: The size of an array is defined at the time of its creation and cannot be changed.

**Advantages:**

* Fast Access: O(1) time complexity for accessing elements by index.
* Simplicity: Easy to use and understand, with a straightforward syntax.
* Memory Efficiency: Efficient memory usage due to contiguous allocation.

**Setup:**

**Create a class Employee**

// Represents an employee with an ID, name, position, and salary

class Employee {

    private int id;

    private String fullName;

    private String jobTitle;

    private double annualSalary;

    // Constructor to initialize an Employee instance

    public Employee(int id, String fullName, String jobTitle, double annualSalary) {

        this.id = id;

        this.fullName = fullName;

        this.jobTitle = jobTitle;

        this.annualSalary = annualSalary;

    }

    // Getter for employee ID

    public int getId() {

        return id;

    }

    // Getter for employee name

    public String getFullName() {

        return fullName;

    }

    // Getter for employee position

    public String getJobTitle() {

        return jobTitle;

    }

    // Getter for employee salary

    public double getAnnualSalary() {

        return annualSalary;

    }

    @Override

    public String toString() {

        return "Employee ID: " + id + ", Name: " + fullName + ", Position: " + jobTitle + ", Salary: $" + annualSalary;

    }

}

// Manages a list of employees using an array

class EmployeeManager {

    private Employee[] employeeList;

    private int currentSize;

    // Constructor to initialize EmployeeManager with a specified capacity

    public EmployeeManager(int capacity) {

        employeeList = new Employee[capacity];

        currentSize = 0;

    }

    // Adds a new employee to the array

    public void addEmployee(Employee employee) {

        if (currentSize < employeeList.length) {

            employeeList[currentSize++] = employee;

        } else {

            System.out.println("The employee list is full. Unable to add more employees.");

        }

    }

    // Searches for an employee using their ID

    public Employee findEmployeeById(int id) {

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

            if (employeeList[i].getId() == id) {

                return employeeList[i];

            }

        }

        return null;

    }

    // Prints the details of all employees in the list

    public void displayAllEmployees() {

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

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

        }

    }

    // Removes an employee from the list using their ID

    public boolean removeEmployeeById(int id) {

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

            if (employeeList[i].getId() == id) {

                // Shift employees to the left to fill the gap

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

                    employeeList[j] = employeeList[j + 1];

                }

                employeeList[--currentSize] = null;

                return true;

            }

        }

        return false;

    }

    public static void main(String[] args) {

        EmployeeManager manager = new EmployeeManager(10);

        // Adding sample employees to the list

        manager.addEmployee(new Employee(1, "Alice Johnson", "Software Developer", 70000));

        manager.addEmployee(new Employee(2, "Bob Smith", "Project Manager", 85000));

        manager.addEmployee(new Employee(3, "Charlie Brown", "Business Analyst", 60000));

        manager.addEmployee(new Employee(4, "David Wilson", "UI/UX Designer", 65000));

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

        manager.displayAllEmployees(); // Display all employees

        System.out.println("\nSearching for employee with ID 3:");

        Employee employee = manager.findEmployeeById(3); // Find an employee by ID

        if (employee != null) {

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

        } else {

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

        }

        System.out.println("\nRemoving employee with ID 2:");

        boolean wasRemoved = manager.removeEmployeeById(2); // Remove an employee by ID

        System.out.println("Removal successful: " + wasRemoved);

        // Display all employees after removal

        System.out.println("\nEmployee List after removal:");

        manager.displayAllEmployees();

    }

}

**Analysis:**

**Time Complexity of Operations**

* Add Employee: Time Complexity: O(1)
* Search Employee by ID: Time Complexity: O(n)
* Traverse Employees: Time Complexity: O(n)
* Delete Employee by ID: Time Complexity: O(n)

**Limitations of Arrays and When to Use Them**

* **Fixed Size**: Arrays have a fixed size, making them unsuitable when the number of elements is unknown or changes frequently.
* **Inefficient for Frequent Insertions/Deletions:** Operations like insertion and deletion are costly (O(n)) compared to dynamic data structures like ArrayList or LinkedList.
* **When to Use Arrays:**
* When the number of elements is known and fixed.
* For applications requiring fast access to elements by index.
* When memory efficiency and performance of access are critical.

# Exercise 5: Task Management System

**Scenario:** To develop a task management system where tasks need to be added, deleted, and traversed efficiently.

**Understand Linked Lists**

Linked lists offer better management of dynamic data due to their flexible size and efficient insertions/deletions, despite having a higher time complexity for search operations compared to arrays.

**Types of Linked Lists;**

* **Singly Linked List:**
* **Structure**: Each node contains data and a reference to the next node.
* **Traversal:** Can only traverse in one direction (forward).
* **Advantages:** Simple implementation, uses less memory compared to doubly linked lists.
* **Doubly Linked List:**
* **Structure:** Each node contains data, a reference to the next node, and a reference to the previous node.
* **Traversal:** Can traverse in both directions (forward and backward).
* **Advantages:** Easier to implement certain operations (like deletion) and more flexible traversal.

**Setup**

**Create a class Task**

// Represents a Task with an ID, name, and status

class Task {

    private int taskId;

    private String taskName;

    private String status;

    // Constructor to initialize a Task instance

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

        this.taskId = taskId;

        this.taskName = taskName;

        this.status = status;

    }

    // Getter for task ID

    public int getTaskId() {

        return taskId;

    }

    // Getter for task name

    public String getTaskName() {

        return taskName;

    }

    // Getter for task status

    public String getStatus() {

        return status;

    }

    @Override

    public String toString() {

        return "Task ID: " + taskId + ", Task Name: " + taskName + ", Status: " + status;

    }

}

// Represents a node in the linked list, containing a Task and a reference to the next node

class Node {

    Task task;

    Node next;

    // Constructor to initialize a Node with a Task

    public Node(Task task) {

        this.task = task;

        this.next = null;

    }

}

// Manages a list of tasks using a linked list

class TaskLinkedList {

    private Node head;

    // Constructor to initialize an empty TaskLinkedList

    public TaskLinkedList() {

        this.head = null;

    }

    // Adds a task to the end of the linked list

    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;

        }

    }

    // Searches for a task by its ID

    public Task searchTaskById(int taskId) {

        Node current = head;

        while (current != null) {

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

                return current.task;

            }

            current = current.next;

        }

        return null;

    }

    // Traverses the linked list and prints all tasks

    public void traverseTasks() {

        Node current = head;

        while (current != null) {

            System.out.println(current.task);

            current = current.next;

        }

    }

    // Deletes a task by its ID

    public boolean deleteTaskById(int taskId) {

        if (head == null) return false;

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

            head = head.next;

            return true;

        }

        Node current = head;

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

            current = current.next;

        }

        if (current.next == null) return false;

        current.next = current.next.next;

        return true;

    }

}

// Demonstrates the usage of the TaskLinkedList class

class Main {

    public static void main(String[] args) {

        TaskLinkedList taskList = new TaskLinkedList();

        // Adding new tasks to the list

        taskList.addTask(new Task(1, "Design UI", "In Progress"));

        taskList.addTask(new Task(2, "Develop Backend", "Not Started"));

        taskList.addTask(new Task(3, "Write Tests", "Not Started"));

        taskList.addTask(new Task(4, "Deploy Application", "Completed"));

        // Display all tasks

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

        taskList.traverseTasks();

        // Search for a specific task by its ID

        System.out.println("\nSearching for task with ID 3:");

        Task task = taskList.searchTaskById(3);

        if (task != null) {

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

        } else {

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

        }

        // Delete a task by its ID

        System.out.println("\nDeleting task with ID 2:");

        boolean isDeleted = taskList.deleteTaskById(2);

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

        // Display all tasks after deletion

        System.out.println("\nAll tasks after deletion:");

        taskList.traverseTasks();

    }

}

**Analysis:**

**Time Complexity of Operations:**

* + **Add Task:**
  + Time Complexity: O(n)
  + **Search Task by ID:**
    - Time Complexity: O(n)
  + **Traverse Tasks:**
    - Time Complexity: O(n)
  + **Delete Task by ID:**
* Time Complexity: O(n)

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

* Dynamic Size: Linked lists can grow and shrink dynamically, unlike arrays that have a fixed size.
* Efficient Insertions/Deletions: Insertions and deletions can be more efficient (O(1)) if the position is known, as there's no need to shift elements.
* Memory Usage: Linked lists use memory more efficiently for dynamic data as they allocate memory as needed, whereas arrays may allocate more memory than necessary.
* Flexibility: Linked lists provide more flexibility with dynamic data structures, making them more suitable for tasks where the size of the dataset changes frequently.

# Exercise 6: Library Management System

**Scenario:** To develop a library management system where users can search for books by title or author.

**Understand Search Algorithms**

**Explain linear search and binary search algorithms.**

**Linear Search:**

Linear search is a simple search algorithm that checks every element in the list sequentially until the desired element is found or the list ends.

* Time Complexity: O(n)
* Space Complexity: O(1)
* Best Case: O(1) (if the element is at the beginning)
* Worst Case: O(n) (if the element is at the end or not present)
* Use Case: Suitable for unsorted or small lists.

**Binary Search:**

Binary search is a more efficient search algorithm for sorted lists. It repeatedly divides the search interval in half, comparing the middle element with the target value.

* Time Complexity: O(log n)
* Space Complexity: O(1)
* Best Case: O(1) (if the middle element is the target)
* Worst Case: O(log n) (if the element is not present)
* Use Case: Suitable for large, sorted lists.

Linear search is straightforward and suitable for small or unsorted datasets, while binary search is more efficient for larger, sorted datasets due to its significantly lower time complexity.

**Setup & Implementation:**

**Create a class Book**

import java.util.Arrays;

import java.util.Comparator;

// Represents a book with an ID, title, and author

class Book {

    private int id;

    private String name;

    private String writer;

    // Constructor to initialize a Book object

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

        this.id = id;

        this.name = name;

        this.writer = writer;

    }

    // Getter for book ID

    public int getId() {

        return id;

    }

    // Getter for book title

    public String getName() {

        return name;

    }

    // Getter for book author

    public String getWriter() {

        return writer;

    }

    @Override

    public String toString() {

        return "Book ID: " + id + ", Title: " + name + ", Author: " + writer;

    }

}

// Manages a collection of books and provides search functionalities

class LibraryManager {

    private Book[] catalog;

    private int count;

    // Constructor to initialize the LibraryManager with a specific capacity

    public LibraryManager(int capacity) {

        catalog = new Book[capacity];

        count = 0;

    }

    // Adds a new book to the catalog

    public void addBook(Book book) {

        if (count < catalog.length) {

            catalog[count++] = book;

        } else {

            System.out.println("The catalog is full. Unable to add more books.");

        }

    }

    // Finds a book by its title using linear search

    public Book searchByTitleLinear(String title) {

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

            if (catalog[i].getName().equalsIgnoreCase(title)) {

                return catalog[i];

            }

        }

        return null;

    }

    // Sorts the books by title for efficient binary search

    public void sortBooksByTitle() {

        Arrays.sort(catalog, 0, count, Comparator.comparing(Book::getName, String.CASE\_INSENSITIVE\_ORDER));

    }

    // Finds a book by its title using binary search

    public Book searchByTitleBinary(String title) {

        int left = 0;

        int right = count - 1;

        while (left <= right) {

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

            int comparison = catalog[middle].getName().compareToIgnoreCase(title);

            if (comparison == 0) {

                return catalog[middle];

            } else if (comparison < 0) {

                left = middle + 1;

            } else {

                right = middle - 1;

            }

        }

        return null;

    }

}

// Main class to demonstrate functionality

class Main {

    public static void main(String[] args) {

        // Initialize the LibraryManager with a capacity of 5 books

        LibraryManager manager = new LibraryManager(5);

        // Add books to the catalog

        manager.addBook(new Book(101, "To Kill a Mockingbird", "Harper Lee"));

        manager.addBook(new Book(102, "1984", "George Orwell"));

        manager.addBook(new Book(103, "The Great Gatsby", "F. Scott Fitzgerald"));

        manager.addBook(new Book(104, "Moby Dick", "Herman Melville"));

        manager.addBook(new Book(105, "The Catcher in the Rye", "J.D. Salinger"));

        // Search for a book by title using linear search

        System.out.println("Searching for '1984' using linear search:");

        Book book = manager.searchByTitleLinear("1984");

        if (book != null) {

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

        } else {

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

        }

        // Sort books by title for binary search

        manager.sortBooksByTitle();

        // Search for a book by title using binary search

        System.out.println("\nSearching for 'The Great Gatsby' using binary search:");

        book = manager.searchByTitleBinary("The Great Gatsby");

        if (book != null) {

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

        } else {

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

        }

    }

}

**Analysis**

**Time Complexity of Search Algorithms**

**Linear Search:**

* Best Case: O(1)
* Average Case: O(n)
* Worst Case: O(n)
* Space Complexity: O(1)

**Binary Search:**

* Best Case: O(1)
* Average Case: O(log n)
* Worst Case: O(log n)
* Space Complexity: O(1)

**When to Use Each Algorithm**

**Linear Search:**

* Use for unsorted or small datasets.
* Simple to implement and does not require sorting.
* Efficient for cases where the dataset size is small or the target element is frequently near the beginning.

**Binary Search:**

* Use for large, sorted datasets.
* Much more efficient for large datasets due to its O(log n) time complexity.
* Requires the list to be sorted, adding an additional step if the data is not already sorted.

# Exercise 7: Financial Forecasting

**Scenario:** To develop a financial forecasting tool that predicts future values based on past data.

**Understand Recursive Algorithms**

**Concept of Recursion**

Recursion is a technique where a function calls itself to solve smaller instances of the same problem. It can simplify complex problems by breaking them down into more manageable subproblems.

* **Base Case**: The condition under which the recursion stops.
* **Recursive Case**: The part of the function where it calls itself with a smaller or simpler input.
* **Advantages**:
  + Simplifies code for problems that have repetitive structures.
  + Often more intuitive for problems like tree traversal, factorial calculation, etc.
* **Disadvantages**:
  + Can lead to excessive memory use due to function call stack.
  + Potential for stack overflow if not properly controlled.

**Setup**

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

// This class performs financial forecasting using recursive calculations

class FinancialForecasting {

    // Recursive method to calculate the future value of an investment

    public static double computeFutureValue(double initialAmount, double annualGrowthRate, int numPeriods) {

        // Base case: if the number of periods is zero, return the initial amount

        if (numPeriods == 0) {

            return initialAmount;

        }

        // Recursive case: calculate the value after one period and call the method with reduced periods

        return computeFutureValue(initialAmount \* (1 + annualGrowthRate), annualGrowthRate, numPeriods - 1);

    }

    // Main method to test the future value prediction

    public static void main(String[] args) {

        double initialAmount = 1000.0;  // Initial amount of money

        double annualGrowthRate = 0.05; // Annual growth rate of 5%

        int numPeriods = 10;            // Number of periods (years) for growth

        // Calculate the future value after the given number of periods

        double futureValue = computeFutureValue(initialAmount, annualGrowthRate, numPeriods);

        // Output the predicted future value

        System.out.println("Projected Future Value: $" + futureValue);

    }

}

**Analysis**

**Time Complexity of Recursive Algorithm**

* **Time Complexity**: O(n), where n is the number of periods.
* **Space Complexity**: O(n), due to the function call stack. Each recursive call adds a new frame to the stack.

**Optimizing the Recursive Solution**

To avoid excessive computation and potential stack overflow, memoization can be used to store previously computed results. Memoization helps in cases where the same subproblems are solved multiple times, reducing redundant calculations. However, in this simple growth rate model, memoization is not necessary because each computation only depends on the immediate previous step and does not involve overlapping subproblems. Each recursive call in this model is unique to its period and does not repeat previous calculations, so the benefits of memoization are minimal in this specific scenario.