Problem Statement: Efficient Array Manipulations and Big O Analysis in Java
Objective: To implement and optimize various array operations in Java, analyze their time and space complexities using Big O notation, and demonstrate the ability to select the most efficient algorithms for given tasks.

Requirements
Implement Basic Array Operations:

Write Java methods to perform the following operations on arrays:
Insertion: Insert an element at a specific position.
Deletion: Delete an element from a specific position.
Search: Search for an element and return its index if found.
Reverse: Reverse the array.
Find Maximum/Minimum: Find the maximum and minimum elements in the array.

Implement Advanced Array Operations:

Write Java methods to perform the following advanced operations on arrays:
Merge: Merge two sorted arrays into one sorted array.
Remove Duplicates: Remove duplicate elements from a sorted array.
Find Duplicates: Identify duplicate elements in the array.
Analyze Time and Space Complexities:

For each array operation, analyze and document the time and space complexities using Big O notation.
Optimize and Compare:

Implement optimized versions of the operations where possible.
Compare the performance of the optimized versions with the initial implementations.
Performance Testing:

Use a large dataset to demonstrate the efficiency of your implementations.
Provide a report comparing the performance of different methods, supported by empirical data (e.g., execution time).
Implementation Steps
Setup:

Create a new Java project.
Define a class ArrayOperations to encapsulate the array operations.
Implement Basic Array Operations:

Implement the basic versions of the required operations.
Ensure proper handling of edge cases (e.g., inserting/deleting at the bounds of the array).
Implement Advanced Array Operations:

Implement the advanced versions of the required operations.
Ensure the methods work efficiently for large datasets.
Analyze and Document Complexities:

Write comments or a separate documentation file explaining the time and space complexities of each method using Big O notation.

Optimize Operations:

Identify opportunities to optimize the initial implementations.
Implement optimized versions and explain the improvements.
Performance Testing:

Write a test class ArrayOperationsTest to benchmark the performance of the different implementations.
Use large arrays to illustrate the efficiency gains of the optimized methods.
Report:

Compile a report that includes:
Descriptions of each operation and its implementation.
Analysis of time and space complexities.
Comparison of initial and optimized implementations.
Performance data from benchmarking.

2.Problem Statement: Array Manipulations and Big O Analysis
Objective: Implement various operations on arrays in Java, analyze their time and space complexities using Big O notation, and demonstrate the ability to select efficient algorithms for given tasks.

Requirements
Implement Basic Array Operations:

Write Java methods to perform the following operations on arrays:
Insertion: Insert an element at the beginning, middle, and end.
Deletion: Delete an element from the beginning, middle, and end.
Search: Linear search for an element and return its index if found.
Reverse: Reverse the array.
Find Maximum/Minimum: Find the maximum and minimum elements in the array.
Implement Advanced Array Operations:

Write Java methods to perform the following advanced operations on arrays:
Sort: Sort the array using both Bubble Sort and Quick Sort.
Rotate: Rotate the array to the right by a given number of steps.
Find Pair with Sum: Find all pairs of elements that sum to a specific value.
Analyze Time and Space Complexities:

For each array operation, analyze and document the time and space complexities using Big O notation.
Performance Testing:

Use a large dataset to demonstrate the efficiency of your implementations.
Provide a report comparing the performance of different methods, supported by empirical data (e.g., execution time).
Implementation Steps
Setup:

Create a new Java project.
Define a class ArrayOperations to encapsulate the array operations.

Implement Basic Array Operations:

Implement the basic versions of the required operations.
Ensure proper handling of edge cases (e.g., inserting/deleting at the bounds of the array).
Implement Advanced Array Operations:
Implement the advanced versions of the required operations.
Ensure the methods work efficiently for large datasets.

Analyze and Document Complexities:
Write comments or a separate documentation file explaining the time and space complexities of each method using Big O notation.

Performance Testing:
Write a test class ArrayOperationsTest to benchmark the performance of the different implementations.
Use large arrays to illustrate the efficiency gains of the optimized methods.
Report:

Compile a report that includes:
Descriptions of each operation and its implementation.
Analysis of time and space complexities.
Comparison of initial and optimized implementations.
Performance data from benchmarking.

3.Problem Statement: Sorting Algorithms and Their Analysis in Java
Objective: Implement various sorting algorithms in Java, analyze their time and space complexities using Big O notation, and demonstrate the ability to choose the most efficient algorithm for different datasets and scenarios.

Requirements
Implement Basic Sorting Algorithms:

Write Java methods to perform the following sorting algorithms:
Bubble Sort
Selection Sort
Insertion Sort
Implement Advanced Sorting Algorithms:
Write Java methods to perform the following advanced sorting algorithms:
Merge Sort
Quick Sort
Heap Sort

Implement Special Sorting Algorithms:
Write Java methods to perform the following special sorting algorithms:
Counting Sort
Radix Sort
Bucket Sort

Analyze Time and Space Complexities:
For each sorting algorithm, analyze and document the time and space complexities using Big O notation.

Optimize and Compare:
Implement optimized versions of the sorting algorithms where applicable.
Compare the performance of different sorting algorithms on various datasets.

Performance Testing:
Use datasets of varying sizes and characteristics to demonstrate the efficiency of your implementations.
Provide a report comparing the performance of different sorting algorithms, supported by empirical data (e.g., execution time).
Implementation Steps
Setup:
Create a new Java project.
Define a class SortingAlgorithms to encapsulate the sorting algorithms.

Implement Basic Sorting Algorithms:
Implement the basic versions of Bubble Sort, Selection Sort, and Insertion Sort.
Ensure proper handling of edge cases (e.g., sorting an already sorted array).
Implement Advanced Sorting Algorithms:

Implement the advanced versions of Merge Sort, Quick Sort, and Heap Sort.
Ensure the methods work efficiently for large datasets.

Implement Special Sorting Algorithms:
Implement the special versions of Counting Sort, Radix Sort, and Bucket Sort.
Ensure the methods handle non-comparative sorting scenarios.

Analyze and Document Complexities:
Write comments or a separate documentation file explaining the time and space complexities of each method using Big O notation.

Optimize Algorithms:
Identify opportunities to optimize the initial implementations.
Implement optimized versions and explain the improvements.

Performance Testing:
Write a test class SortingAlgorithmsTest to benchmark the performance of the different implementations.
Use datasets with different sizes and characteristics to illustrate the efficiency gains of the optimized methods.
Report:

Compile a report that includes:
Descriptions of each sorting algorithm and its implementation.
Analysis of time and space complexities.
Comparison of initial and optimized implementations.
Performance data from benchmarking.

4.Problem Statement: List Operations and Their Analysis in Java

Objective: Implement various operations on different types of lists in Java, analyze their time and space complexities using Big O notation, and demonstrate the ability to choose the most efficient data structure for different tasks.

Requirements
Implement Basic List Operations:
Write Java methods to perform the following operations on lists:
Add: Add an element at the beginning, middle, and end of the list.
Remove: Remove an element from the beginning, middle, and end of the list.
Get: Retrieve an element by its index.
Set: Update an element at a specific index.
Contains: Check if the list contains a specific element.

Implement Different Types of Lists:
Implement and compare the following types of lists:
ArrayList: Backed by a dynamic array.
LinkedList: Backed by a doubly-linked list.
CustomLinkedList: Implement a singly-linked list from scratch.
Analyze Time and Space Complexities:

For each list operation, analyze and document the time and space complexities using Big O notation.

Compare List Implementations:
Compare the performance of ArrayList, LinkedList, and CustomLinkedList for various operations.
Determine the most efficient list implementation for different scenarios.

Performance Testing:
Use datasets of varying sizes and characteristics to demonstrate the efficiency of your implementations.
Provide a report comparing the performance of different list implementations, supported by empirical data (e.g., execution time).
Implementation Steps
Setup:
Create a new Java project.
Define a class ListOperations to encapsulate the list operations.

Implement Basic List Operations:
Implement the basic versions of the required operations for ArrayList and LinkedList.
Ensure proper handling of edge cases (e.g., adding/removing elements at the bounds of the list).
Implement CustomLinkedList:
Implement a singly-linked list from scratch with methods for adding, removing, getting, setting, and checking elements.
Analyze and Document Complexities:

Write comments or a separate documentation file explaining the time and space complexities of each method using Big O notation.

Compare List Implementations:
Benchmark the performance of ArrayList, LinkedList, and CustomLinkedList for various operations.
Identify the most efficient list implementation for different scenarios.

Performance Testing:
Write a test class ListOperationsTest to benchmark the performance of the different list implementations.
Use datasets with different sizes and characteristics to illustrate the efficiency gains of the optimized methods.
Report:

Compile a report that includes:
Descriptions of each list operation and its implementation.
Analysis of time and space complexities.
Comparison of different list implementations.
Performance data from benchmarking.

1.Employee Management System
Design a Java program for managing employees using Comparator interface to sort employees based on different criteria. The program should include the following classes: Employee, EmployeeComparator, EmployeeManagement, and EmployeeManagementSystem.

Employee Class
Attributes: id, name, age, salary.
Methods: Getters, setters, and toString() method.
EmployeeComparator Class

Implementations:
CompareById: Compares employees based on id in ascending order.
CompareByName: Compares employees based on name in alphabetical order.
CompareByAge: Compares employees based on age in ascending order.
CompareBySalary: Compares employees based on salary in ascending order.
EmployeeManagement Class
Attributes: employees (an array or list of Employee objects).
Methods:
addEmployee(Employee employee): Adds a new employee to the list.
removeEmployee(int id): Removes an employee from the list by id.
sortEmployees(Comparator<Employee> comparator): Sorts employees based on the provided Comparator.
displayEmployees(): Displays all employees in the list.
EmployeeManagementSystem Class
Attributes: employeeManagement (an instance of EmployeeManagement).
Methods:
addNewEmployee(int id, String name, int age, double salary): Creates a new Employee object and adds it to the management system.
removeEmployee(int id): Removes an employee from the management system.
sortEmployeesById(): Sorts employees by id.
sortEmployeesByName(): Sorts employees by name.
sortEmployeesByAge(): Sorts employees by age.
sortEmployeesBySalary(): Sorts employees by salary.
displayEmployees(): Displays all employees managed by the system.
Requirements

Implement appropriate constructors and methods for each class.
Use Comparator interface to define multiple comparison strategies (CompareById, CompareByName, CompareByAge, CompareBySalary) in the EmployeeComparator class.
Demonstrate the "has-a" relationship between EmployeeManagementSystem and EmployeeManagement, and between EmployeeManagement and Employee.
Design the classes inside a package named employeeManagement.
Design a client application (EmployeeManagementSystemApp) with a menu-driven interface for performing operations on the employee management system.

2.Product Inventory Management System

Design a Java program for managing a product inventory using Comparator interface to sort products based on different criteria. The program should include the following classes: Product, ProductComparator, Inventory, and InventoryManagementSystem.

Product Class
Attributes: productId, productName, price, quantity.
Methods: Getters, setters, and toString() method.
ProductComparator Class
Implementations:
CompareById: Compares products based on productId in ascending order.
CompareByName: Compares products based on productName in alphabetical order.
CompareByPrice: Compares products based on price in ascending order.
CompareByQuantity: Compares products based on quantity in ascending order.
Inventory Class
Attributes: inventory (an array or list of Product objects).
Methods:
addProduct(Product product): Adds a new product to the inventory.
removeProduct(int productId): Removes a product from the inventory by productId.
sortProducts(Comparator<Product> comparator): Sorts products based on the provided Comparator.
displayInventory(): Displays all products in the inventory.
InventoryManagementSystem Class
Attributes: inventory (an instance of Inventory).
Methods:
addNewProduct(int productId, String productName, double price, int quantity): Creates a new Product object and adds it to the inventory.
removeProduct(int productId): Removes a product from the inventory.
sortProductsById(): Sorts products by productId.
sortProductsByName(): Sorts products by productName.
sortProductsByPrice(): Sorts products by price.
sortProductsByQuantity(): Sorts products by quantity.
displayInventory(): Displays all products in the inventory.
Requirements
Implement appropriate constructors and methods for each class.
Use Comparator interface to define multiple comparison strategies (CompareById, CompareByName, CompareByPrice, CompareByQuantity) in the ProductComparator class.
Demonstrate the "has-a" relationship between InventoryManagementSystem and Inventory, and between Inventory and Product.
Design the classes inside a package named inventoryManagement.
Design a client application (InventoryManagementSystemApp) with a menu-driven interface for performing operations on the product inventory.

3.Student Grade Management System

Design a Java program for managing student grades using Comparable interface to sort students based on their grades. The program should include the following classes: Student, GradeComparator, StudentManagement, and StudentManagementSystem.

Student Class
Attributes: studentId, studentName, grade.
Methods: Getters, setters, and toString() method.
GradeComparator Class
Implementations:
CompareByGrade: Compares students based on grade in descending order.
StudentManagement Class
Attributes: students (an array or list of Student objects).
Methods:
addStudent(Student student): Adds a new student to the list.
removeStudent(int studentId): Removes a student from the list by studentId.
sortStudents(): Sorts students based on their grades using Comparable.
displayStudents(): Displays all students in the list.
StudentManagementSystem Class
Attributes: studentManagement (an instance of StudentManagement).
Methods:
addNewStudent(int studentId, String studentName, int grade): Creates a new Student object and adds it to the management system.
removeStudent(int studentId): Removes a student from the management system.
sortStudentsByGrade(): Sorts students by grade.
displayStudents(): Displays all students managed by the system.
Requirements
Implement appropriate constructors and methods for each class.
Use Comparable interface to define the natural ordering of Student objects based on their grade in descending order.
Demonstrate the "has-a" relationship between StudentManagementSystem and StudentManagement, and between StudentManagement and Student.
Design the classes inside a package named studentManagement.
Design a client application (StudentManagementSystemApp) with a menu-driven interface for performing operations on the student management system.

4.Music Album Management System

Design a Java program for managing music albums using Comparable and Comparator interfaces to sort albums based on different criteria. The program should include the following classes: Album, AlbumComparator, AlbumManagement, and AlbumManagementSystem.

Album Class
Attributes: title, artist, releaseYear, numberOfTracks.
Methods: Getters, setters, and toString() method.
AlbumComparator Class
Implementations:
CompareByTitle: Compares albums based on title in alphabetical order.
CompareByArtist: Compares albums based on artist in alphabetical order.
CompareByReleaseYear: Compares albums based on releaseYear in ascending order.

From: K H, Shivaprasada
Sent: Friday, July 5, 2024 11:58 AM
To: Kumar, Nishant <nishant.k.kumar@capgemini.com>; Kumar, Anubhav <anubhav.b.kumar@capgemini.com>; Mandal, Swapnanil <swapnanil.mandal@capgemini.com>; Tripathy, Sumit <sumit.tripathy@capgemini.com>; Kumar, Tapan <tapan.kumar@capgemini.com>; Pratap, Shakti <shakti.pratap@capgemini.com>; Upadhyay, Kaustubh <kaustubh.upadhyay@capgemini.com>; Kumar, Ayush <ayush.f.kumar@capgemini.com>; Sahu, Satya <satya.sahu@capgemini.com>; Bharti, Om Prakash <om-prakash.bharti@capgemini.com>; Tah, Hritam <hritam.tah@capgemini.com>; Chakrabarty, Arindam <arindam.chakrabarty@capgemini.com>; Gupta, Shahil <shahil.gupta@capgemini.com>; Dubey, Ashutosh <ashutosh.dubey@capgemini.com>; Thakur, Suman <suman.a.thakur@capgemini.com>; Mahato, Rajiv Kumar <rajiv-kumar.mahato@capgemini.com>; Dubey, Saurav Kumar <saurav-kumar.dubey@capgemini.com>; S, Vijay Vignesh <vijay-vignesh.s@capgemini.com>; Chavan, Vedant Mukesh <vedant-mukesh.chavan@capgemini.com>; Verma, Vivek <vivek.b.verma@capgemini.com>; Bouri, Ranjit <ranjit.bouri@capgemini.com>; Behera, Rudra Prasad <rudra-prasad.behera@capgemini.com>; SHARMA, MOHIT <mohit.h.sharma@capgemini.com>; Gupta, Ratan <ratan.gupta@capgemini.com>; Singh, Rishabh <rishabh.d.singh@capgemini.com>; Khuntia, Soumesh <soumesh.khuntia@capgemini.com>; Kanungo, Souravashree <souravashree.kanungo@capgemini.com>; Jain, Parteek <parteek.jain@capgemini.com>; J, SATISH <satish.a.j@capgemini.com>; RAJ, RACHIT <rachit.raj@capgemini.com>; Kumar, Reshav <reshav.kumar@capgemini.com>; Bihari, Madhav <madhav.bihari@capgemini.com>
Cc: ., Vishalarani <vishalarani.vishalarani@capgemini.com>
Subject: Implement Basic Array Operations:

Problem Statement: Efficient Array Manipulations and Big O Analysis in Java
Objective: To implement and optimize various array operations in Java, analyze their time and space complexities using Big O notation, and demonstrate the ability to select the most efficient algorithms for given tasks.

Requirements
Implement Basic Array Operations:

Write Java methods to perform the following operations on arrays:
Insertion: Insert an element at a specific position.
Deletion: Delete an element from a specific position.
Search: Search for an element and return its index if found.
Reverse: Reverse the array.
Find Maximum/Minimum: Find the maximum and minimum elements in the array.

Implement Advanced Array Operations:

Write Java methods to perform the following advanced operations on arrays:
Merge: Merge two sorted arrays into one sorted array.
Remove Duplicates: Remove duplicate elements from a sorted array.
Find Duplicates: Identify duplicate elements in the array.
Analyze Time and Space Complexities:

For each array operation, analyze and document the time and space complexities using Big O notation.
Optimize and Compare:

Implement optimized versions of the operations where possible.
Compare the performance of the optimized versions with the initial implementations.
Performance Testing:

Use a large dataset to demonstrate the efficiency of your implementations.
Provide a report comparing the performance of different methods, supported by empirical data (e.g., execution time).
Implementation Steps
Setup:

Create a new Java project.
Define a class ArrayOperations to encapsulate the array operations.
Implement Basic Array Operations:

Implement the basic versions of the required operations.
Ensure proper handling of edge cases (e.g., inserting/deleting at the bounds of the array).
Implement Advanced Array Operations:

Implement the advanced versions of the required operations.
Ensure the methods work efficiently for large datasets.
Analyze and Document Complexities:

Write comments or a separate documentation file explaining the time and space complexities of each method using Big O notation.
Optimize Operations:

Identify opportunities to optimize the initial implementations.
Implement optimized versions and explain the improvements.
Performance Testing:

Write a test class ArrayOperationsTest to benchmark the performance of the different implementations.
Use large arrays to illustrate the efficiency gains of the optimized methods.
Report:

Compile a report that includes:
Descriptions of each operation and its implementation.
Analysis of time and space complexities.
Comparison of initial and optimized implementations.
Performance data from benchmarking.

2.Problem Statement: Array Manipulations and Big O Analysis
Objective: Implement various operations on arrays in Java, analyze their time and space complexities using Big O notation, and demonstrate the ability to select efficient algorithms for given tasks.

Requirements
Implement Basic Array Operations:

Write Java methods to perform the following operations on arrays:

Insertion: Insert an element at the beginning, middle, and end.
Deletion: Delete an element from the beginning, middle, and end.
Search: Linear search for an element and return its index if found.
Reverse: Reverse the array.
Find Maximum/Minimum: Find the maximum and minimum elements in the array.
Implement Advanced Array Operations:

Write Java methods to perform the following advanced operations on arrays:
Sort: Sort the array using both Bubble Sort and Quick Sort.
Rotate: Rotate the array to the right by a given number of steps.
Find Pair with Sum: Find all pairs of elements that sum to a specific value.
Analyze Time and Space Complexities:

For each array operation, analyze and document the time and space complexities using Big O notation.
Performance Testing:

Use a large dataset to demonstrate the efficiency of your implementations.
Provide a report comparing the performance of different methods, supported by empirical data (e.g., execution time).
Implementation Steps
Setup:

Create a new Java project.
Define a class ArrayOperations to encapsulate the array operations.
Implement Basic Array Operations:

Implement the basic versions of the required operations.
Ensure proper handling of edge cases (e.g., inserting/deleting at the bounds of the array).
Implement Advanced Array Operations:
Implement the advanced versions of the required operations.
Ensure the methods work efficiently for large datasets.

Analyze and Document Complexities:
Write comments or a separate documentation file explaining the time and space complexities of each method using Big O notation.

Performance Testing:
Write a test class ArrayOperationsTest to benchmark the performance of the different implementations.
Use large arrays to illustrate the efficiency gains of the optimized methods.
Report:

Compile a report that includes:
Descriptions of each operation and its implementation.
Analysis of time and space complexities.
Comparison of initial and optimized implementations.
Performance data from benchmarking.

3.Problem Statement: Sorting Algorithms and Their Analysis in Java

Objective: Implement various sorting algorithms in Java, analyze their time and space complexities using Big O notation, and demonstrate the ability to choose the most efficient algorithm for different datasets and scenarios.

Requirements
Implement Basic Sorting Algorithms:

Write Java methods to perform the following sorting algorithms:
Bubble Sort
Selection Sort
Insertion Sort
Implement Advanced Sorting Algorithms:
Write Java methods to perform the following advanced sorting algorithms:
Merge Sort
Quick Sort
Heap Sort

Implement Special Sorting Algorithms:
Write Java methods to perform the following special sorting algorithms:
Counting Sort
Radix Sort
Bucket Sort

Analyze Time and Space Complexities:
For each sorting algorithm, analyze and document the time and space complexities using Big O notation.

Optimize and Compare:
Implement optimized versions of the sorting algorithms where applicable.
Compare the performance of different sorting algorithms on various datasets.

Performance Testing:
Use datasets of varying sizes and characteristics to demonstrate the efficiency of your implementations.
Provide a report comparing the performance of different sorting algorithms, supported by empirical data (e.g., execution time).
Implementation Steps
Setup:
Create a new Java project.
Define a class SortingAlgorithms to encapsulate the sorting algorithms.

Implement Basic Sorting Algorithms:
Implement the basic versions of Bubble Sort, Selection Sort, and Insertion Sort.
Ensure proper handling of edge cases (e.g., sorting an already sorted array).
Implement Advanced Sorting Algorithms:

Implement the advanced versions of Merge Sort, Quick Sort, and Heap Sort.
Ensure the methods work efficiently for large datasets.

Implement Special Sorting Algorithms:
Implement the special versions of Counting Sort, Radix Sort, and Bucket Sort.

Ensure the methods handle non-comparative sorting scenarios.

Analyze and Document Complexities:
Write comments or a separate documentation file explaining the time and space complexities of each method using Big O notation.

Optimize Algorithms:
Identify opportunities to optimize the initial implementations.
Implement optimized versions and explain the improvements.

Performance Testing:
Write a test class SortingAlgorithmsTest to benchmark the performance of the different implementations.
Use datasets with different sizes and characteristics to illustrate the efficiency gains of the optimized methods.
Report:

Compile a report that includes:
Descriptions of each sorting algorithm and its implementation.
Analysis of time and space complexities.
Comparison of initial and optimized implementations.
Performance data from benchmarking.

4.Problem Statement: List Operations and Their Analysis in Java
Objective: Implement various operations on different types of lists in Java, analyze their time and space complexities using Big O notation, and demonstrate the ability to choose the most efficient data structure for different tasks.

Requirements
Implement Basic List Operations:
Write Java methods to perform the following operations on lists:
Add: Add an element at the beginning, middle, and end of the list.
Remove: Remove an element from the beginning, middle, and end of the list.
Get: Retrieve an element by its index.
Set: Update an element at a specific index.
Contains: Check if the list contains a specific element.

Implement Different Types of Lists:
Implement and compare the following types of lists:
ArrayList: Backed by a dynamic array.
LinkedList: Backed by a doubly-linked list.
CustomLinkedList: Implement a singly-linked list from scratch.
Analyze Time and Space Complexities:

For each list operation, analyze and document the time and space complexities using Big O notation.

Compare List Implementations:
Compare the performance of ArrayList, LinkedList, and CustomLinkedList for various operations.
Determine the most efficient list implementation for different scenarios.

Performance Testing:
Use datasets of varying sizes and characteristics to demonstrate the efficiency of your implementations.
Provide a report comparing the performance of different list implementations, supported by empirical data (e.g., execution time).
Implementation Steps
Setup:
Create a new Java project.
Define a class ListOperations to encapsulate the list operations.

Implement Basic List Operations:
Implement the basic versions of the required operations for ArrayList and LinkedList.
Ensure proper handling of edge cases (e.g., adding/removing elements at the bounds of the list).
Implement CustomLinkedList:
Implement a singly-linked list from scratch with methods for adding, removing, getting, setting, and checking elements.
Analyze and Document Complexities:

Write comments or a separate documentation file explaining the time and space complexities of each method using Big O notation.

Compare List Implementations:
Benchmark the performance of ArrayList, LinkedList, and CustomLinkedList for various operations.
Identify the most efficient list implementation for different scenarios.

Performance Testing:
Write a test class ListOperationsTest to benchmark the performance of the different list implementations.
Use datasets with different sizes and characteristics to illustrate the efficiency gains of the optimized methods.
Report:

Compile a report that includes:
Descriptions of each list operation and its implementation.
Analysis of time and space complexities.
Comparison of different list implementations.
Performance data from benchmarking.

1.A library wants to automate the management of its books and members. The details provided by the library are as follows:

Book Details:

BookTitle
Author
ISBN
MaxCopies

AvailableCopies
Member Details:

MemberName
MemberID
BorrowedBooks (List of borrowed book ISBNs)
The automated software should enable the librarian to perform the following tasks:

Add a new book
Update the number of available copies of a book
Register a new member
Borrow a book
Return a book
Display the details of all books
Display the details of all members
The solution should:

Add appropriate constructors and methods to the classes.
Demonstrate the "has-a" relationship between the library class and the member and book classes.
Use the ArrayList collection class to add the books and the members to the application dynamically.
The complete design of the classes should be inside a package.
Design the client application as a menu-driven application for the various tasks that the librarian performs.

2.A gym wants to automate the management of its members and the classes they attend. The details provided by the gym are as follows:

Class Details:

ClassName
Instructor
MaxParticipants
CurrentParticipants
Member Details:

MemberName
MemberID
EnrolledClasses (List of enrolled class names)
The automated software should enable the gym admin to perform the following tasks:

Add a new class
Update the number of current participants of a class
Register a new member
Enroll a member in a class
Remove a member from a class
Display the details of all classes
Display the details of all members
The solution should:

Add appropriate constructors and methods to the classes.
Demonstrate the "has-a" relationship between the gym class and the member and class classes.

Use the ArrayList collection class to add the classes and the members to the application dynamically.
The complete design of the classes should be inside a package.
Design the client application as a menu-driven application for the various tasks that the gym admin performs.

3.A bookstore management system has the following classes: Book, Purchase, and PurchaseList. A PurchaseList has many Purchase objects for a given customer. A Purchase object is created whenever a book is purchased and contains information about the Book being bought. The Book class includes the title, author, price, and description. The PurchaseList needs to use an ArrayList for storing multiple Purchase objects.

The BillingSystem class has a method called generateInvoice to which an object of PurchaseList is provided. This method calculates the grand total for the purchase list by considering the quantity and discount (from Purchase) and the price (from Book). Each time a purchase is made, some taxes need to be applied. For this purpose, there is a TaxCalculator class that helps in calculating the sales tax. The current tax rate is 8%, which can be revised frequently.

Requirements
Create appropriate constructors and methods for the classes.
Demonstrate the "has-a" relationship between PurchaseList and Purchase, and between Purchase and Book.
Use the ArrayList collection class to manage the list of purchases dynamically.
Design the classes inside a package.
Design a client application with a menu-driven interface for various tasks performed by the bookstore admin.

4.A grocery store management system has the following classes: Product, Purchase, and PurchaseList. A PurchaseList has many Purchase objects for a given customer. A Purchase object is created whenever a product is purchased and contains information about the Product being bought. The Product class includes the name, price, and description. The PurchaseList needs to use a LinkedList for storing multiple Purchase objects.

The BillingSystem class has a method called generateInvoice to which an object of PurchaseList is provided. This method calculates the grand total for the purchase list by considering the quantity and discount (from Purchase) and the price (from Product). Each time a purchase is made, some taxes need to be applied. For this purpose, there is a TaxCalculator class that helps in calculating the sales tax. The current tax rate is 7%, which can be revised frequently.

Requirements
Create appropriate constructors and methods for the classes.
Demonstrate the "has-a" relationship between PurchaseList and Purchase, and between Purchase and Product.
Use the LinkedList collection class to manage the list of purchases dynamically.
Design the classes inside a package.
Design a client application with a menu-driven interface for various tasks performed by the grocery store admin.

5.A student management system has the following classes: Student, Course, and StudentSystem. The StudentSystem class maintains a collection of unique students using a TreeSet. Each Student has a

name, student ID, and a set of courses they are enrolled in. Each Course has a course code and a course name.

The StudentSystem class has methods for:

Adding a new student to the system.
Adding a new course for a student.
Removing a course for a student.
Displaying all students in the system.
Displaying all courses for a specific student.
Displaying all unique courses across all students.
Requirements
Create appropriate constructors and methods for the classes.
Demonstrate the "has-a" relationship between Student and Course, and between StudentSystem and Student.
Use the TreeSet collection class to manage the list of unique students.
Design the classes inside a package.
Design a client application with a menu-driven interface for various tasks performed by the student system admin.

6.A movie database system has the following classes: Movie, Actor, and MovieDatabase. The MovieDatabase class maintains a collection of unique movies using a HashSet. Each Movie has a title, director, release year, and a set of actors starring in it. Each Actor has a name and a birth year.

The MovieDatabase class has methods for:

Adding a new movie to the database.
Adding an actor to a movie.
Removing an actor from a movie.
Displaying all movies in the database.
Displaying all actors for a specific movie.
Displaying all unique actors in the entire database.
Requirements
Create appropriate constructors and methods for the classes.
Demonstrate the "has-a" relationship between Movie and Actor, and between MovieDatabase and Movie.
Use the HashSet collection class to manage the list of unique movies.
Design the classes inside a package.
Design a client application with a menu-driven interface for various tasks performed by the movie database admin.

7.A task management system has the following classes: Task, TaskManager, and TaskSystem. The TaskSystem class maintains a queue of tasks using a Queue interface, implemented typically with LinkedList. Each Task has a title, description, priority, and status (e.g., pending, completed).

The TaskManager class has methods for:

Adding a new task to the system.
Removing the next task from the queue (removing the highest priority task first).
Marking a task as completed.

Displaying all tasks in the system.
Displaying the next task to be completed without removing it.
Displaying all completed tasks.
Requirements
Create appropriate constructors and methods for the classes.
Demonstrate the "has-a" relationship between Task and TaskManager, and between TaskManager and TaskSystem.
Use the Queue interface (implemented with LinkedList) to manage tasks in a first-in, first-out (FIFO) manner.
Design the classes inside a package.
Design a client application with a menu-driven interface for various tasks performed by the task manager.

8.A university administration wants to develop a Student Records Management System using Java with the following classes: Student, Course, StudentRecord, and UniversitySystem.

Student Class
Attributes: studentId, name, email, phone.
Methods: Getters, setters, and toString() method.
Course Class
Attributes: courseCode, courseName, creditHours.
Methods: Getters, setters, and toString() method.
StudentRecord Class
Attributes: student, coursesEnrolled (a Map of courses and grades).
Methods:
addCourse(Course course, double grade): Adds a course and its grade to the record.
removeCourse(Course course): Removes a course from the record.
displayCourses(): Displays all courses enrolled along with grades.
UniversitySystem Class
Attributes: studentRecords (a Map with studentId as key and StudentRecord as value).
Methods:
addStudent(Student student): Adds a new student to the system.
removeStudent(String studentId): Removes a student from the system.
enrollStudent(String studentId, Course course): Enrolls a student in a course.
dropStudentCourse(String studentId, Course course): Drops a course for a student.
displayStudentCourses(String studentId): Displays all courses enrolled by a student.
Requirements
Implement appropriate constructors and methods for each class.
Ensure that the UniversitySystem class manages student records using a Map (typically HashMap), with studentId as the key and StudentRecord as the value.
Demonstrate the "has-a" relationship between UniversitySystem and StudentRecord, and between StudentRecord and Student and Course.
Design the classes inside a package named universitySystem.
Design a client application (UniversitySystemApp) with a menu-driven interface for performing operations on student records.

9.Design a Java program for managing a book catalog using LinkedHashMap to maintain insertion order. The program should include the following classes: Book, BookCatalog, and CatalogSystem.

Book Class
Attributes: isbn (unique identifier), title, author, price.

Methods: Getters, setters, and toString() method.
BookCatalog Class
Attributes: catalog (a LinkedHashMap with isbn as key and Book as value).
Methods:
addBook(Book book): Adds a new book to the catalog.
removeBook(String isbn): Removes a book from the catalog.
displayAllBooks(): Displays all books in the catalog in the insertion order.
CatalogSystem Class
Attributes: bookCatalog (an instance of BookCatalog).
Methods:
addNewBook(String isbn, String title, String author, double price): Creates a new book object and adds it to the catalog.
removeBook(String isbn): Removes a book from the catalog.
displayAllBooks(): Displays all books in the catalog.
Requirements
Implement appropriate constructors and methods for each class.
Use LinkedHashMap to maintain the insertion order of books in the catalog.
Demonstrate the "has-a" relationship between CatalogSystem and BookCatalog, and between BookCatalog and Book.
Design the classes inside a package named bookCatalog.
Design a client application (CatalogSystemApp) with a menu-driven interface for performing operations on the book catalog.

10.Design a Java program for managing a dictionary using TreeMap to maintain natural ordering by keys (words). The program should include the following classes: Word, Dictionary, and DictionarySystem.

Word Class
Attributes: word (unique identifier), meaning.
Methods: Getters, setters, and toString() method.
Dictionary Class
Attributes: dictionary (a TreeMap with word as key and Word as value).
Methods:
addWord(Word word): Adds a new word to the dictionary.
removeWord(String word): Removes a word from the dictionary.
searchWord(String word): Searches for a word in the dictionary and displays its meaning.
displayAllWords(): Displays all words in the dictionary in alphabetical order.
DictionarySystem Class
Attributes: dictionary (an instance of Dictionary).
Methods:
addNewWord(String word, String meaning): Creates a new Word object and adds it to the dictionary.
removeWord(String word): Removes a word from the dictionary.
searchWord(String word): Searches for a word in the dictionary and displays its meaning.
displayAllWords(): Displays all words in the dictionary.
Requirements
Implement appropriate constructors and methods for each class.
Use TreeMap to maintain natural ordering by keys (words) in the dictionary.
Demonstrate the "has-a" relationship between DictionarySystem and Dictionary, and between Dictionary and Word.
Design the classes inside a package named dictionary.

Design a client application (DictionarySystemApp) with a menu-driven interface for performing operations on the dictionary.

From: K H, Shivaprasada
Sent: Friday, July 5, 2024 11:58 AM
To: Kumar, Nishant <nishant.k.kumar@capgemini.com>; Kumar, Anubhav <anubhav.b.kumar@capgemini.com>; Mandal, Swapnanil <swapnanil.mandal@capgemini.com>; Tripathy, Sumit <sumit.tripathy@capgemini.com>; Kumar, Tapan <tapan.kumar@capgemini.com>; Pratap, Shakti <shakti.pratap@capgemini.com>; Upadhyay, Kaustubh <kaustubh.upadhyay@capgemini.com>; Kumar, Ayush <ayush.f.kumar@capgemini.com>; Sahu, Satya <satya.sahu@capgemini.com>; Bharti, Om Prakash <om-prakash.bharti@capgemini.com>; Tah, Hritam <hritam.tah@capgemini.com>; Chakrabarty, Arindam <arindam.chakrabarty@capgemini.com>; Gupta, Shahil <shahil.gupta@capgemini.com>; Dubey, Ashutosh <ashutosh.dubey@capgemini.com>; Thakur, Suman <suman.a.thakur@capgemini.com>; Mahato, Rajiv Kumar <rajiv-kumar.mahato@capgemini.com>; Dubey, Saurav Kumar <saurav-kumar.dubey@capgemini.com>; S, Vijay Vignesh <vijay-vignesh.s@capgemini.com>; Chavan, Vedant Mukesh <vedant-mukesh.chavan@capgemini.com>; Verma, Vivek <vivek.b.verma@capgemini.com>; Bouri, Ranjit <ranjit.bouri@capgemini.com>; Behera, Rudra Prasad <rudra-prasad.behera@capgemini.com>; SHARMA, MOHIT <mohit.h.sharma@capgemini.com>; Gupta, Ratan <ratan.gupta@capgemini.com>; Singh, Rishabh <rishabh.d.singh@capgemini.com>; Khuntia, Soumesh <soumesh.khuntia@capgemini.com>; Kanungo, Souravashree <souravashree.kanungo@capgemini.com>; Jain, Parteek <parteek.jain@capgemini.com>; J, SATISH <satish.a.j@capgemini.com>; RAJ, RACHIT <rachit.raj@capgemini.com>; Kumar, Reshav <reshav.kumar@capgemini.com>; Bihari, Madhav <madhav.bihari@capgemini.com>
Cc: ., Vishalarani <vishalarani.vishalarani@capgemini.com>
Subject: Implement Basic Array Operations:

Problem Statement: Efficient Array Manipulations and Big O Analysis in Java
Objective: To implement and optimize various array operations in Java, analyze their time and space complexities using Big O notation, and demonstrate the ability to select the most efficient algorithms for given tasks.

Requirements
Implement Basic Array Operations:

Write Java methods to perform the following operations on arrays:
Insertion: Insert an element at a specific position.
Deletion: Delete an element from a specific position.
Search: Search for an element and return its index if found.
Reverse: Reverse the array.
Find Maximum/Minimum: Find the maximum and minimum elements in the array.

Implement Advanced Array Operations:

Write Java methods to perform the following advanced operations on arrays:
Merge: Merge two sorted arrays into one sorted array.
Remove Duplicates: Remove duplicate elements from a sorted array.
Find Duplicates: Identify duplicate elements in the array.
Analyze Time and Space Complexities:

For each array operation, analyze and document the time and space complexities using Big O notation.
Optimize and Compare:

Implement optimized versions of the operations where possible.
Compare the performance of the optimized versions with the initial implementations.
Performance Testing:

Use a large dataset to demonstrate the efficiency of your implementations.
Provide a report comparing the performance of different methods, supported by empirical data (e.g., execution time).
Implementation Steps
Setup:

Create a new Java project.
Define a class ArrayOperations to encapsulate the array operations.
Implement Basic Array Operations:

Implement the basic versions of the required operations.
Ensure proper handling of edge cases (e.g., inserting/deleting at the bounds of the array).
Implement Advanced Array Operations:

Implement the advanced versions of the required operations.
Ensure the methods work efficiently for large datasets.
Analyze and Document Complexities:

Write comments or a separate documentation file explaining the time and space complexities of each method using Big O notation.
Optimize Operations:

Identify opportunities to optimize the initial implementations.
Implement optimized versions and explain the improvements.
Performance Testing:

Write a test class ArrayOperationsTest to benchmark the performance of the different implementations.
Use large arrays to illustrate the efficiency gains of the optimized methods.
Report:

Compile a report that includes:
Descriptions of each operation and its implementation.
Analysis of time and space complexities.
Comparison of initial and optimized implementations.
Performance data from benchmarking.

2.Problem Statement: Array Manipulations and Big O Analysis
Objective: Implement various operations on arrays in Java, analyze their time and space complexities using Big O notation, and demonstrate the ability to select efficient algorithms for given tasks.

Requirements

Implement Basic Array Operations:

Write Java methods to perform the following operations on arrays:
Insertion: Insert an element at the beginning, middle, and end.
Deletion: Delete an element from the beginning, middle, and end.
Search: Linear search for an element and return its index if found.
Reverse: Reverse the array.
Find Maximum/Minimum: Find the maximum and minimum elements in the array.
Implement Advanced Array Operations:

Write Java methods to perform the following advanced operations on arrays:
Sort: Sort the array using both Bubble Sort and Quick Sort.
Rotate: Rotate the array to the right by a given number of steps.
Find Pair with Sum: Find all pairs of elements that sum to a specific value.
Analyze Time and Space Complexities:

For each array operation, analyze and document the time and space complexities using Big O notation.
Performance Testing:

Use a large dataset to demonstrate the efficiency of your implementations.
Provide a report comparing the performance of different methods, supported by empirical data (e.g., execution time).
Implementation Steps
Setup:

Create a new Java project.
Define a class ArrayOperations to encapsulate the array operations.
Implement Basic Array Operations:

Implement the basic versions of the required operations.
Ensure proper handling of edge cases (e.g., inserting/deleting at the bounds of the array).
Implement Advanced Array Operations:
Implement the advanced versions of the required operations.
Ensure the methods work efficiently for large datasets.

Analyze and Document Complexities:
Write comments or a separate documentation file explaining the time and space complexities of each method using Big O notation.

Performance Testing:
Write a test class ArrayOperationsTest to benchmark the performance of the different implementations.
Use large arrays to illustrate the efficiency gains of the optimized methods.
Report:

Compile a report that includes:
Descriptions of each operation and its implementation.
Analysis of time and space complexities.
Comparison of initial and optimized implementations.
Performance data from benchmarking.

3.Problem Statement: Sorting Algorithms and Their Analysis in Java
Objective: Implement various sorting algorithms in Java, analyze their time and space complexities using Big O notation, and demonstrate the ability to choose the most efficient algorithm for different datasets and scenarios.

Requirements
Implement Basic Sorting Algorithms:

Write Java methods to perform the following sorting algorithms:
Bubble Sort
Selection Sort
Insertion Sort
Implement Advanced Sorting Algorithms:
Write Java methods to perform the following advanced sorting algorithms:
Merge Sort
Quick Sort
Heap Sort

Implement Special Sorting Algorithms:
Write Java methods to perform the following special sorting algorithms:
Counting Sort
Radix Sort
Bucket Sort

Analyze Time and Space Complexities:
For each sorting algorithm, analyze and document the time and space complexities using Big O notation.

Optimize and Compare:
Implement optimized versions of the sorting algorithms where applicable.
Compare the performance of different sorting algorithms on various datasets.

Performance Testing:
Use datasets of varying sizes and characteristics to demonstrate the efficiency of your implementations.
Provide a report comparing the performance of different sorting algorithms, supported by empirical data (e.g., execution time).
Implementation Steps
Setup:
Create a new Java project.
Define a class SortingAlgorithms to encapsulate the sorting algorithms.

Implement Basic Sorting Algorithms:
Implement the basic versions of Bubble Sort, Selection Sort, and Insertion Sort.
Ensure proper handling of edge cases (e.g., sorting an already sorted array).
Implement Advanced Sorting Algorithms:

Implement the advanced versions of Merge Sort, Quick Sort, and Heap Sort.
Ensure the methods work efficiently for large datasets.

Implement Special Sorting Algorithms:
Implement the special versions of Counting Sort, Radix Sort, and Bucket Sort.
Ensure the methods handle non-comparative sorting scenarios.

Analyze and Document Complexities:
Write comments or a separate documentation file explaining the time and space complexities of each method using Big O notation.

Optimize Algorithms:
Identify opportunities to optimize the initial implementations.
Implement optimized versions and explain the improvements.

Performance Testing:
Write a test class SortingAlgorithmsTest to benchmark the performance of the different implementations.
Use datasets with different sizes and characteristics to illustrate the efficiency gains of the optimized methods.
Report:

Compile a report that includes:
Descriptions of each sorting algorithm and its implementation.
Analysis of time and space complexities.
Comparison of initial and optimized implementations.
Performance data from benchmarking.

4.Problem Statement: List Operations and Their Analysis in Java
Objective: Implement various operations on different types of lists in Java, analyze their time and space complexities using Big O notation, and demonstrate the ability to choose the most efficient data structure for different tasks.

Requirements
Implement Basic List Operations:
Write Java methods to perform the following operations on lists:
Add: Add an element at the beginning, middle, and end of the list.
Remove: Remove an element from the beginning, middle, and end of the list.
Get: Retrieve an element by its index.
Set: Update an element at a specific index.
Contains: Check if the list contains a specific element.

Implement Different Types of Lists:
Implement and compare the following types of lists:
ArrayList: Backed by a dynamic array.
LinkedList: Backed by a doubly-linked list.
CustomLinkedList: Implement a singly-linked list from scratch.
Analyze Time and Space Complexities:

For each list operation, analyze and document the time and space complexities using Big O notation.

Compare List Implementations:
Compare the performance of ArrayList, LinkedList, and CustomLinkedList for various operations.

Determine the most efficient list implementation for different scenarios.

Performance Testing:
Use datasets of varying sizes and characteristics to demonstrate the efficiency of your implementations.
Provide a report comparing the performance of different list implementations, supported by empirical data (e.g., execution time).
Implementation Steps
Setup:
Create a new Java project.
Define a class ListOperations to encapsulate the list operations.

Implement Basic List Operations:
Implement the basic versions of the required operations for ArrayList and LinkedList.
Ensure proper handling of edge cases (e.g., adding/removing elements at the bounds of the list).
Implement CustomLinkedList:
Implement a singly-linked list from scratch with methods for adding, removing, getting, setting, and checking elements.
Analyze and Document Complexities:

Write comments or a separate documentation file explaining the time and space complexities of each method using Big O notation.

Compare List Implementations:
Benchmark the performance of ArrayList, LinkedList, and CustomLinkedList for various operations.
Identify the most efficient list implementation for different scenarios.

Performance Testing:
Write a test class ListOperationsTest to benchmark the performance of the different list implementations.
Use datasets with different sizes and characteristics to illustrate the efficiency gains of the optimized methods.
Report:

Compile a report that includes:
Descriptions of each list operation and its implementation.
Analysis of time and space complexities.
Comparison of different list implementations.
Performance data from benchmarking.

1.Library Management System
Design and implement a Library Management System in Java with the following requirements:

Book Class: Create a Book class with attributes such as bookId, title, author, genre, and quantityAvailable. Include appropriate getters and setters.

Member Class: Create a Member class with attributes such as memberId, name, phoneNumber, and booksBorrowed. booksBorrowed should be a list of Book objects that the member has borrowed. Include appropriate methods to add and remove books from this list.

Library Class:

Maintain an array or list of Book objects to represent the collection of books in the library.
Maintain an array or list of Member objects to represent the library members.
Implement methods to add new books to the library, display all books, and search for a book by its title or author.
Implement methods to add new members to the library, display all members, and search for a member by their name or phone number.
Implement methods for a member to borrow a book (reduce quantityAvailable for the book and add it to the member's booksBorrowed) and return a book (increase quantityAvailable and remove it from the member's booksBorrowed).
Handle exceptions where appropriate, such as when trying to borrow a book that is not available or when trying to return a book that the member has not borrowed.
Main Class (LibraryManagementSystem):

Create instances of Library, Book, and Member to test the functionalities.
Demonstrate adding books and members, borrowing and returning books, searching for books and members, and displaying their details.

Book Class: Represents a book with attributes such as bookId, title, author, genre, and quantityAvailable. Includes methods for setting/getting attributes and overriding toString() for easy printing.

Member Class: Represents a library member with attributes such as memberId, name, phoneNumber, and booksBorrowed (list of Book objects). Includes methods to borrow and return books, and overrides toString() for easy printing.

Library Class: Manages a collection of Book objects (books) and Member objects (members). Includes methods to add books/members, display all books/members, search books/members by title/author/name/phone number.

LibraryManagementSystem Class: Main class demonstrating the functionalities of the Library Management System. It creates instances of Library, Book, and Member, adds books and members to the library, displays them, and tests borrowing and returning books with exception handling (BookNotAvailableException).

2.Student Management System
Design and implement a Student Management System in Java with the following requirements:

Student Class: Create a Student class with attributes such as studentId, name, age, gender, and coursesEnrolled. Include appropriate methods for adding/removing courses and displaying student details.

Course Class: Create a Course class with attributes such as courseId, courseName, instructor, and maxStudents. Include methods to enroll/unenroll students, display course details, and check if a course is full.

University Class:

Maintain an array or list of Student objects to represent the students in the university.
Maintain an array or list of Course objects to represent the courses offered by the university.
Implement methods to add new students/courses, display all students/courses, and search for a student/course by their ID/name or courseId/courseName.
Implement methods to enroll a student in a course and unenroll a student from a course. Handle exceptions where appropriate, such as when trying to enroll in a full course or unenroll from a course not enrolled in.
Main Class (StudentManagementSystem):

Create instances of University, Student, and Course to test the functionalities.
Demonstrate adding students and courses, enrolling/unenrolling students, searching for students/courses, and displaying their details.

Explanation:
Student Class: Represents a student with attributes such as studentId, name, age, gender, and coursesEnrolled (list of Course objects). Includes methods to enroll/unenroll in courses and overrides toString() for easy printing.

Course Class: Represents a course with attributes such as courseId, courseName, instructor, maxStudents, and enrolledStudents (list of Student objects). Includes methods to check if the course is full, add/remove students, and overrides toString() for easy printing.

University Class: Manages a collection of Student objects (students) and Course objects (courses). Includes methods to add students/courses, display all students/courses, search students/courses by ID/name/courseId/courseName, and enroll/unenroll students in/from courses. Handles exceptions (CourseFullException) where appropriate.

StudentManagementSystem Class: Main class demonstrating the functionalities of the Student Management System. It creates instances of University, Student, and Course, adds students and courses to the university, displays them, and tests enrolling/unenrolling students in/from courses with exception handling (CourseFullException).

3.Banking System
Design and implement a Banking System in Java with the following requirements:

Account Class: Create an Account class with attributes such as accountId, accountType (Savings or Checking), balance, and owner (instance of Customer class). Include methods for deposit, withdraw, and view balance.

Customer Class: Create a Customer class with attributes such as customerId, name, address, and accounts (list of Account objects). Include methods to add/remove accounts and display customer details.

Bank Class:

Maintain an array or list of Customer objects to represent the customers of the bank.
Implement methods to add new customers/accounts, display all customers/accounts, and search for a customer/account by their ID/name.
Implement methods for performing transactions such as deposit and withdraw across accounts.
Handle exceptions where appropriate, such as insufficient balance for a withdrawal.
Main Class (BankingSystem):

Create instances of Bank, Customer, and Account to test the functionalities.
Demonstrate adding customers/accounts, performing transactions, searching for customers/accounts, and displaying their details.

Account Class: Represents a bank account with attributes such as accountId, accountType, balance, and owner (a Customer object). Includes methods for deposit, withdraw, view balance, and overrides toString() for easy printing.

Customer Class: Represents a bank customer with attributes such as customerId, name, address, and accounts (list of Account objects). Includes methods to add/remove accounts, display accounts, and overrides toString() for easy printing.

Bank Class: Manages a collection of Customer objects (customers). Includes methods to add customers, display all customers, search customers by ID/name, and perform transactions (deposit and withdraw) across accounts. Handles exceptions (InsufficientBalanceException) where appropriate.

BankingSystem Class: Main class demonstrating the functionalities of the Banking System. It creates instances of Bank, Customer, and Account, adds customers and accounts to the bank, displays them, and tests transactions (deposit and withdraw) with exception handling (InsufficientBalanceException).

4.Online Shopping System
Design and implement an Online Shopping System in Java with the following requirements:

Product Class: Create a Product class with attributes such as productId, productName, price, quantityAvailable, and category. Include methods to add/remove products, display product details, and check availability.

Customer Class: Create a Customer class with attributes such as customerId, name, email, and address. Include methods to add/remove products to/from cart, place orders, view order history, and display customer details.

ShoppingCart Class: Create a ShoppingCart class to manage the items added by a customer. Include methods to add/remove items, calculate total price, and display cart details.

Order Class: Create an Order class to represent an order placed by a customer. Include attributes such as orderId, orderDate, customer, items (list of Product objects), and totalAmount. Implement methods to display order details.

OnlineShoppingSystem Class:

Maintain an array or list of Product objects to represent the products available in the online store.
Maintain an array or list of Customer objects to represent the registered customers.

Implement methods to add new products/customers, display all products/customers, and search for a product/customer by their ID/name or customerId/name.
Implement methods for customers to add/remove products to/from cart, place orders, view order history, and display their details.

5.School Management System
Design and implement a School Management System in Java with the following requirements:

Student Class: Create a Student class with attributes such as studentId, name, grade, address, and coursesEnrolled (list of Course objects). Include methods to enroll/drop courses, display student details, and view enrolled courses.

Teacher Class: Create a Teacher class with attributes such as teacherId, name, address, and coursesTaught (list of Course objects). Include methods to assign/unassign courses, display teacher details, and view assigned courses.

Course Class: Create a Course class with attributes such as courseId, courseName, teacher (instance of Teacher class), studentsEnrolled (list of Student objects), and maxCapacity. Include methods to add/remove students, display course details, and check availability.

School Class:

Maintain arrays or lists of Student, Teacher, and Course objects to represent the entities in the school.
Implement methods to add new students/teachers/courses, display all students/teachers/courses, and search for a student/teacher/course by their ID/name.
Implement methods for enrolling/dropping students from courses, assigning/unassigning teachers to courses, and handling exceptions where appropriate, such as enrolling a student in a full course.
Main Class (SchoolManagementSystem):

Create instances of School, Student, Teacher, and Course to test the functionalities.
Demonstrate adding students/teachers/courses, enrolling/dropping students from courses, assigning/unassigning teachers to courses, searching for students/teachers/courses, and displaying their details.

From: K H, Shivaprasada
Sent: Friday, July 5, 2024 12:01 PM
To: Kumar, Nishant <nishant.k.kumar@capgemini.com>; Kumar, Anubhav <anubhav.b.kumar@capgemini.com>; Mandal, Swapnanil <swapnanil.mandal@capgemini.com>; Tripathy, Sumit <sumit.tripathy@capgemini.com>; Kumar, Tapan <tapan.kumar@capgemini.com>; Pratap, Shakti <shakti.pratap@capgemini.com>; Upadhyay, Kaustubh <kaustubh.upadhyay@capgemini.com>; Kumar, Ayush <ayush.f.kumar@capgemini.com>; Sahu, Satya <satya.sahu@capgemini.com>; Bharti, Om Prakash <om-prakash.bharti@capgemini.com>; Tah, Hritam <hritam.tah@capgemini.com>; Chakrabarty, Arindam <arindam.chakrabarty@capgemini.com>; Gupta, Shahil <shahil.gupta@capgemini.com>; Dubey, Ashutosh <ashutosh.dubey@capgemini.com>; Thakur, Suman <suman.a.thakur@capgemini.com>; Mahato, Rajiv Kumar <rajiv-kumar.mahato@capgemini.com>; Dubey, Saurav Kumar <saurav-kumar.dubey@capgemini.com>; S, Vijay Vignesh <vijay-vignesh.s@capgemini.com>; Chavan, Vedant Mukesh <vedant-mukesh.chavan@capgemini.com>; Verma, Vivek <vivek.b.verma@capgemini.com>; Bouri, Ranjit <ranjit.bouri@capgemini.com>; Behera, Rudra

Prasad <rudra-prasad.behera@capgemini.com>; SHARMA, MOHIT <mohit.h.sharma@capgemini.com>; Gupta, Ratan <ratan.gupta@capgemini.com>; Singh, Rishabh <rishabh.d.singh@capgemini.com>; Khuntia, Soumesh <soumesh.khuntia@capgemini.com>; Kanungo, Souravashree <souravashree.kanungo@capgemini.com>; Jain, Parteek <parteek.jain@capgemini.com>; J, SATISH <satish.a.j@capgemini.com>; RAJ, RACHIT <rachit.raj@capgemini.com>; Kumar, Reshav <reshav.kumar@capgemini.com>; Bihari, Madhav <madhav.bihari@capgemini.com>
Cc: ., Vishalarani <vishalarani.vishalarani@capgemini.com>
Subject: RE: Implement Basic Array Operations:

1.A library wants to automate the management of its books and members. The details provided by the library are as follows:

Book Details:

BookTitle
Author
ISBN
MaxCopies
AvailableCopies
Member Details:

MemberName
MemberID
BorrowedBooks (List of borrowed book ISBNs)
The automated software should enable the librarian to perform the following tasks:

Add a new book
Update the number of available copies of a book
Register a new member
Borrow a book
Return a book
Display the details of all books
Display the details of all members
The solution should:

Add appropriate constructors and methods to the classes.
Demonstrate the "has-a" relationship between the library class and the member and book classes.
Use the ArrayList collection class to add the books and the members to the application dynamically.
The complete design of the classes should be inside a package.
Design the client application as a menu-driven application for the various tasks that the librarian performs.

2.A gym wants to automate the management of its members and the classes they attend. The details provided by the gym are as follows:

Class Details:

ClassName
Instructor
MaxParticipants

CurrentParticipants
Member Details:

MemberName
MemberID
EnrolledClasses (List of enrolled class names)
The automated software should enable the gym admin to perform the following tasks:

Add a new class
Update the number of current participants of a class
Register a new member
Enroll a member in a class
Remove a member from a class
Display the details of all classes
Display the details of all members
The solution should:

Add appropriate constructors and methods to the classes.
Demonstrate the "has-a" relationship between the gym class and the member and class classes.
Use the ArrayList collection class to add the classes and the members to the application dynamically.
The complete design of the classes should be inside a package.
Design the client application as a menu-driven application for the various tasks that the gym admin performs.

3.A bookstore management system has the following classes: Book, Purchase, and PurchaseList. A PurchaseList has many Purchase objects for a given customer. A Purchase object is created whenever a book is purchased and contains information about the Book being bought. The Book class includes the title, author, price, and description. The PurchaseList needs to use an ArrayList for storing multiple Purchase objects.

The BillingSystem class has a method called generateInvoice to which an object of PurchaseList is provided. This method calculates the grand total for the purchase list by considering the quantity and discount (from Purchase) and the price (from Book). Each time a purchase is made, some taxes need to be applied. For this purpose, there is a TaxCalculator class that helps in calculating the sales tax. The current tax rate is 8%, which can be revised frequently.

Requirements
Create appropriate constructors and methods for the classes.
Demonstrate the "has-a" relationship between PurchaseList and Purchase, and between Purchase and Book.
Use the ArrayList collection class to manage the list of purchases dynamically.
Design the classes inside a package.
Design a client application with a menu-driven interface for various tasks performed by the bookstore admin.

4.A grocery store management system has the following classes: Product, Purchase, and PurchaseList. A PurchaseList has many Purchase objects for a given customer. A Purchase object is created whenever a product is purchased and contains information about the Product being bought. The Product class includes the name, price, and description. The PurchaseList needs to use a LinkedList for storing multiple Purchase objects.

The BillingSystem class has a method called generateInvoice to which an object of PurchaseList is provided. This method calculates the grand total for the purchase list by considering the quantity and discount (from Purchase) and the price (from Product). Each time a purchase is made, some taxes need to be applied. For this purpose, there is a TaxCalculator class that helps in calculating the sales tax. The current tax rate is 7%, which can be revised frequently.

Requirements
Create appropriate constructors and methods for the classes.
Demonstrate the "has-a" relationship between PurchaseList and Purchase, and between Purchase and Product.
Use the LinkedList collection class to manage the list of purchases dynamically.
Design the classes inside a package.
Design a client application with a menu-driven interface for various tasks performed by the grocery store admin.

5.A student management system has the following classes: Student, Course, and StudentSystem. The StudentSystem class maintains a collection of unique students using a TreeSet. Each Student has a name, student ID, and a set of courses they are enrolled in. Each Course has a course code and a course name.

The StudentSystem class has methods for:

Adding a new student to the system.
Adding a new course for a student.
Removing a course for a student.
Displaying all students in the system.
Displaying all courses for a specific student.
Displaying all unique courses across all students.
Requirements
Create appropriate constructors and methods for the classes.
Demonstrate the "has-a" relationship between Student and Course, and between StudentSystem and Student.
Use the TreeSet collection class to manage the list of unique students.
Design the classes inside a package.
Design a client application with a menu-driven interface for various tasks performed by the student system admin.

6.A movie database system has the following classes: Movie, Actor, and MovieDatabase. The MovieDatabase class maintains a collection of unique movies using a HashSet. Each Movie has a title, director, release year, and a set of actors starring in it. Each Actor has a name and a birth year.

The MovieDatabase class has methods for:

Adding a new movie to the database.
Adding an actor to a movie.
Removing an actor from a movie.
Displaying all movies in the database.
Displaying all actors for a specific movie.
Displaying all unique actors in the entire database.

Requirements
Create appropriate constructors and methods for the classes.
Demonstrate the "has-a" relationship between Movie and Actor, and between MovieDatabase and Movie.
Use the HashSet collection class to manage the list of unique movies.
Design the classes inside a package.
Design a client application with a menu-driven interface for various tasks performed by the movie database admin.

7.A task management system has the following classes: Task, TaskManager, and TaskSystem. The TaskSystem class maintains a queue of tasks using a Queue interface, implemented typically with LinkedList. Each Task has a title, description, priority, and status (e.g., pending, completed).

The TaskManager class has methods for:

Adding a new task to the system.
Removing the next task from the queue (removing the highest priority task first).
Marking a task as completed.
Displaying all tasks in the system.
Displaying the next task to be completed without removing it.
Displaying all completed tasks.
Requirements
Create appropriate constructors and methods for the classes.
Demonstrate the "has-a" relationship between Task and TaskManager, and between TaskManager and TaskSystem.
Use the Queue interface (implemented with LinkedList) to manage tasks in a first-in, first-out (FIFO) manner.
Design the classes inside a package.
Design a client application with a menu-driven interface for various tasks performed by the task manager.

8.A university administration wants to develop a Student Records Management System using Java with the following classes: Student, Course, StudentRecord, and UniversitySystem.

Student Class
Attributes: studentId, name, email, phone.
Methods: Getters, setters, and toString() method.
Course Class
Attributes: courseCode, courseName, creditHours.
Methods: Getters, setters, and toString() method.
StudentRecord Class
Attributes: student, coursesEnrolled (a Map of courses and grades).
Methods:
addCourse(Course course, double grade): Adds a course and its grade to the record.
removeCourse(Course course): Removes a course from the record.
displayCourses(): Displays all courses enrolled along with grades.
UniversitySystem Class
Attributes: studentRecords (a Map with studentId as key and StudentRecord as value).
Methods:
addStudent(Student student): Adds a new student to the system.
removeStudent(String studentId): Removes a student from the system.

enrollStudent(String studentId, Course course): Enrolls a student in a course.
dropStudentCourse(String studentId, Course course): Drops a course for a student.
displayStudentCourses(String studentId): Displays all courses enrolled by a student.
Requirements
Implement appropriate constructors and methods for each class.
Ensure that the UniversitySystem class manages student records using a Map (typically HashMap), with studentId as the key and StudentRecord as the value.
Demonstrate the "has-a" relationship between UniversitySystem and StudentRecord, and between StudentRecord and Student and Course.
Design the classes inside a package named universitySystem.
Design a client application (UniversitySystemApp) with a menu-driven interface for performing operations on student records.

9.Design a Java program for managing a book catalog using LinkedHashMap to maintain insertion order. The program should include the following classes: Book, BookCatalog, and CatalogSystem.

Book Class
Attributes: isbn (unique identifier), title, author, price.
Methods: Getters, setters, and toString() method.
BookCatalog Class
Attributes: catalog (a LinkedHashMap with isbn as key and Book as value).
Methods:
addBook(Book book): Adds a new book to the catalog.
removeBook(String isbn): Removes a book from the catalog.
displayAllBooks(): Displays all books in the catalog in the insertion order.
CatalogSystem Class
Attributes: bookCatalog (an instance of BookCatalog).
Methods:
addNewBook(String isbn, String title, String author, double price): Creates a new book object and adds it to the catalog.
removeBook(String isbn): Removes a book from the catalog.
displayAllBooks(): Displays all books in the catalog.
Requirements
Implement appropriate constructors and methods for each class.
Use LinkedHashMap to maintain the insertion order of books in the catalog.
Demonstrate the "has-a" relationship between CatalogSystem and BookCatalog, and between BookCatalog and Book.
Design the classes inside a package named bookCatalog.
Design a client application (CatalogSystemApp) with a menu-driven interface for performing operations on the book catalog.

10.Design a Java program for managing a dictionary using TreeMap to maintain natural ordering by keys (words). The program should include the following classes: Word, Dictionary, and DictionarySystem.

Word Class
Attributes: word (unique identifier), meaning.
Methods: Getters, setters, and toString() method.
Dictionary Class
Attributes: dictionary (a TreeMap with word as key and Word as value).
Methods:

addWord(Word word): Adds a new word to the dictionary.
removeWord(String word): Removes a word from the dictionary.
searchWord(String word): Searches for a word in the dictionary and displays its meaning.
displayAllWords(): Displays all words in the dictionary in alphabetical order.
DictionarySystem Class
Attributes: dictionary (an instance of Dictionary).
Methods:
addNewWord(String word, String meaning): Creates a new Word object and adds it to the dictionary.
removeWord(String word): Removes a word from the dictionary.
searchWord(String word): Searches for a word in the dictionary and displays its meaning.
displayAllWords(): Displays all words in the dictionary.
Requirements
Implement appropriate constructors and methods for each class.
Use TreeMap to maintain natural ordering by keys (words) in the dictionary.
Demonstrate the "has-a" relationship between DictionarySystem and Dictionary, and between Dictionary and Word.
Design the classes inside a package named dictionary.
Design a client application (DictionarySystemApp) with a menu-driven interface for performing operations on the dictionary.

From: K H, Shivaprasada
Sent: Friday, July 5, 2024 11:58 AM
To: Kumar, Nishant <nishant.k.kumar@capgemini.com>; Kumar, Anubhav <anubhav.b.kumar@capgemini.com>; Mandal, Swapnanil <swapnanil.mandal@capgemini.com>; Tripathy, Sumit <sumit.tripathy@capgemini.com>; Kumar, Tapan <tapan.kumar@capgemini.com>; Pratap, Shakti <shakti.pratap@capgemini.com>; Upadhyay, Kaustubh <kaustubh.upadhyay@capgemini.com>; Kumar, Ayush <ayush.f.kumar@capgemini.com>; Sahu, Satya <satya.sahu@capgemini.com>; Bharti, Om Prakash <om-prakash.bharti@capgemini.com>; Tah, Hritam <hritam.tah@capgemini.com>; Chakrabarty, Arindam <arindam.chakrabarty@capgemini.com>; Gupta, Shahil <shahil.gupta@capgemini.com>; Dubey, Ashutosh <ashutosh.dubey@capgemini.com>; Thakur, Suman <suman.a.thakur@capgemini.com>; Mahato, Rajiv Kumar <rajiv-kumar.mahato@capgemini.com>; Dubey, Saurav Kumar <saurav-kumar.dubey@capgemini.com>; S, Vijay Vignesh <vijay-vignesh.s@capgemini.com>; Chavan, Vedant Mukesh <vedant-mukesh.chavan@capgemini.com>; Verma, Vivek <vivek.b.verma@capgemini.com>; Bouri, Ranjit <ranjit.bouri@capgemini.com>; Behera, Rudra Prasad <rudra-prasad.behera@capgemini.com>; SHARMA, MOHIT <mohit.h.sharma@capgemini.com>; Gupta, Ratan <ratan.gupta@capgemini.com>; Singh, Rishabh <rishabh.d.singh@capgemini.com>; Khuntia, Soumesh <soumesh.khuntia@capgemini.com>; Kanungo, Souravashree <souravashree.kanungo@capgemini.com>; Jain, Parteek <parteek.jain@capgemini.com>; J, SATISH <satish.a.j@capgemini.com>; RAJ, RACHIT <rachit.raj@capgemini.com>; Kumar, Reshav <reshav.kumar@capgemini.com>; Bihari, Madhav <madhav.bihari@capgemini.com>
Cc: ., Vishalarani <vishalarani.vishalarani@capgemini.com>
Subject: Implement Basic Array Operations:

Problem Statement: Efficient Array Manipulations and Big O Analysis in Java
Objective: To implement and optimize various array operations in Java, analyze their time and space complexities using Big O notation, and demonstrate the ability to select the most efficient algorithms for given tasks.

Requirements
Implement Basic Array Operations:

Write Java methods to perform the following operations on arrays:
Insertion: Insert an element at a specific position.
Deletion: Delete an element from a specific position.
Search: Search for an element and return its index if found.
Reverse: Reverse the array.
Find Maximum/Minimum: Find the maximum and minimum elements in the array.

Implement Advanced Array Operations:

Write Java methods to perform the following advanced operations on arrays:
Merge: Merge two sorted arrays into one sorted array.
Remove Duplicates: Remove duplicate elements from a sorted array.
Find Duplicates: Identify duplicate elements in the array.
Analyze Time and Space Complexities:

For each array operation, analyze and document the time and space complexities using Big O notation.
Optimize and Compare:

Implement optimized versions of the operations where possible.
Compare the performance of the optimized versions with the initial implementations.
Performance Testing:

Use a large dataset to demonstrate the efficiency of your implementations.
Provide a report comparing the performance of different methods, supported by empirical data (e.g., execution time).
Implementation Steps
Setup:

Create a new Java project.
Define a class ArrayOperations to encapsulate the array operations.
Implement Basic Array Operations:

Implement the basic versions of the required operations.
Ensure proper handling of edge cases (e.g., inserting/deleting at the bounds of the array).
Implement Advanced Array Operations:

Implement the advanced versions of the required operations.
Ensure the methods work efficiently for large datasets.
Analyze and Document Complexities:

Write comments or a separate documentation file explaining the time and space complexities of each method using Big O notation.
Optimize Operations:

Identify opportunities to optimize the initial implementations.
Implement optimized versions and explain the improvements.
Performance Testing:

Write a test class ArrayOperationsTest to benchmark the performance of the different implementations.
Use large arrays to illustrate the efficiency gains of the optimized methods.
Report:

Compile a report that includes:
Descriptions of each operation and its implementation.
Analysis of time and space complexities.
Comparison of initial and optimized implementations.
Performance data from benchmarking.

2.Problem Statement: Array Manipulations and Big O Analysis
Objective: Implement various operations on arrays in Java, analyze their time and space complexities using Big O notation, and demonstrate the ability to select efficient algorithms for given tasks.

Requirements
Implement Basic Array Operations:

Write Java methods to perform the following operations on arrays:
Insertion: Insert an element at the beginning, middle, and end.
Deletion: Delete an element from the beginning, middle, and end.
Search: Linear search for an element and return its index if found.
Reverse: Reverse the array.
Find Maximum/Minimum: Find the maximum and minimum elements in the array.
Implement Advanced Array Operations:

Write Java methods to perform the following advanced operations on arrays:
Sort: Sort the array using both Bubble Sort and Quick Sort.
Rotate: Rotate the array to the right by a given number of steps.
Find Pair with Sum: Find all pairs of elements that sum to a specific value.
Analyze Time and Space Complexities:

For each array operation, analyze and document the time and space complexities using Big O notation.
Performance Testing:

Use a large dataset to demonstrate the efficiency of your implementations.
Provide a report comparing the performance of different methods, supported by empirical data (e.g., execution time).
Implementation Steps
Setup:

Create a new Java project.
Define a class ArrayOperations to encapsulate the array operations.
Implement Basic Array Operations:

Implement the basic versions of the required operations.
Ensure proper handling of edge cases (e.g., inserting/deleting at the bounds of the array).
Implement Advanced Array Operations:

Implement the advanced versions of the required operations.
Ensure the methods work efficiently for large datasets.

Analyze and Document Complexities:
Write comments or a separate documentation file explaining the time and space complexities of each method using Big O notation.

Performance Testing:
Write a test class ArrayOperationsTest to benchmark the performance of the different implementations.
Use large arrays to illustrate the efficiency gains of the optimized methods.
Report:

Compile a report that includes:
Descriptions of each operation and its implementation.
Analysis of time and space complexities.
Comparison of initial and optimized implementations.
Performance data from benchmarking.

3.Problem Statement: Sorting Algorithms and Their Analysis in Java
Objective: Implement various sorting algorithms in Java, analyze their time and space complexities using Big O notation, and demonstrate the ability to choose the most efficient algorithm for different datasets and scenarios.

Requirements
Implement Basic Sorting Algorithms:

Write Java methods to perform the following sorting algorithms:
Bubble Sort
Selection Sort
Insertion Sort
Implement Advanced Sorting Algorithms:
Write Java methods to perform the following advanced sorting algorithms:
Merge Sort
Quick Sort
Heap Sort

Implement Special Sorting Algorithms:
Write Java methods to perform the following special sorting algorithms:
Counting Sort
Radix Sort
Bucket Sort

Analyze Time and Space Complexities:
For each sorting algorithm, analyze and document the time and space complexities using Big O notation.

Optimize and Compare:
Implement optimized versions of the sorting algorithms where applicable.
Compare the performance of different sorting algorithms on various datasets.

Performance Testing:
Use datasets of varying sizes and characteristics to demonstrate the efficiency of your implementations.
Provide a report comparing the performance of different sorting algorithms, supported by empirical data (e.g., execution time).
Implementation Steps
Setup:
Create a new Java project.
Define a class SortingAlgorithms to encapsulate the sorting algorithms.

Implement Basic Sorting Algorithms:
Implement the basic versions of Bubble Sort, Selection Sort, and Insertion Sort.
Ensure proper handling of edge cases (e.g., sorting an already sorted array).
Implement Advanced Sorting Algorithms:

Implement the advanced versions of Merge Sort, Quick Sort, and Heap Sort.
Ensure the methods work efficiently for large datasets.

Implement Special Sorting Algorithms:
Implement the special versions of Counting Sort, Radix Sort, and Bucket Sort.
Ensure the methods handle non-comparative sorting scenarios.

Analyze and Document Complexities:
Write comments or a separate documentation file explaining the time and space complexities of each method using Big O notation.

Optimize Algorithms:
Identify opportunities to optimize the initial implementations.
Implement optimized versions and explain the improvements.

Performance Testing:
Write a test class SortingAlgorithmsTest to benchmark the performance of the different implementations.
Use datasets with different sizes and characteristics to illustrate the efficiency gains of the optimized methods.
Report:

Compile a report that includes:
Descriptions of each sorting algorithm and its implementation.
Analysis of time and space complexities.
Comparison of initial and optimized implementations.
Performance data from benchmarking.

4.Problem Statement: List Operations and Their Analysis in Java
Objective: Implement various operations on different types of lists in Java, analyze their time and space complexities using Big O notation, and demonstrate the ability to choose the most efficient data structure for different tasks.

Requirements
Implement Basic List Operations:

Write Java methods to perform the following operations on lists:
Add: Add an element at the beginning, middle, and end of the list.
Remove: Remove an element from the beginning, middle, and end of the list.
Get: Retrieve an element by its index.
Set: Update an element at a specific index.
Contains: Check if the list contains a specific element.

Implement Different Types of Lists:
Implement and compare the following types of lists:
ArrayList: Backed by a dynamic array.
LinkedList: Backed by a doubly-linked list.
CustomLinkedList: Implement a singly-linked list from scratch.
Analyze Time and Space Complexities:

For each list operation, analyze and document the time and space complexities using Big O notation.

Compare List Implementations:
Compare the performance of ArrayList, LinkedList, and CustomLinkedList for various operations.
Determine the most efficient list implementation for different scenarios.

Performance Testing:
Use datasets of varying sizes and characteristics to demonstrate the efficiency of your implementations.
Provide a report comparing the performance of different list implementations, supported by empirical data (e.g., execution time).
Implementation Steps
Setup:
Create a new Java project.
Define a class ListOperations to encapsulate the list operations.

Implement Basic List Operations:
Implement the basic versions of the required operations for ArrayList and LinkedList.
Ensure proper handling of edge cases (e.g., adding/removing elements at the bounds of the list).
Implement CustomLinkedList:
Implement a singly-linked list from scratch with methods for adding, removing, getting, setting, and checking elements.
Analyze and Document Complexities:

Write comments or a separate documentation file explaining the time and space complexities of each method using Big O notation.

Compare List Implementations:
Benchmark the performance of ArrayList, LinkedList, and CustomLinkedList for various operations.
Identify the most efficient list implementation for different scenarios.

Performance Testing:
Write a test class ListOperationsTest to benchmark the performance of the different list implementations.
Use datasets with different sizes and characteristics to illustrate the efficiency gains of the optimized methods.
Report:

Compile a report that includes:
Descriptions of each list operation and its implementation.
Analysis of time and space complexities.
Comparison of different list implementations.
Performance data from benchmarking.

Problem Statement:
1.A hospital wants to maintain details of its staff members. Some of these staff members are doctors, and some are nurses. Both types of staff members have certain common attributes like name, age, id, and department. Additionally, doctors have attributes like specialization and consultation fee, while nurses have attributes like shift timings and hourly wage.
Demonstrate the above scenario by having Staff as the base class and Doctor and Nurse as the subclasses. Let the search method in the Staff class be an abstract method. Have appropriate data members and member functions in these classes to solve the following queries:
a) Get the total count of the staff members for the hospital. b) Get the count of the doctors and nurses. c) Given a staff id, display the details.

2.A library wants to maintain details of its members. Some of these members are students, and some are faculty members. Both types of members have certain common attributes like name, age, id, and membership type. Additionally, students have attributes like grade and borrowed books, while faculty members have attributes like department and research publications.
Demonstrate the above scenario by having LibraryMember as the base class and StudentMember and FacultyMember as the subclasses. Let the search method in the LibraryMember class be an abstract method. Have appropriate data members and member functions in these classes to solve the following queries:
a) Get the total count of the library members. b) Get the count of the student members and faculty members. c) Given a member id, display the details.

3.A software development company wants to maintain details of its employees. Some of these employees are developers, and some are testers. Both types of employees have certain common attributes like name, age, id, and department. Additionally, developers have attributes like programming languages known and projects handled, while testers have attributes like testing tools known and bugs reported.
Demonstrate the above scenario by having Employee as the base class and Developer and Tester as the subclasses. Let the search method in the Employee class be an abstract method. Have appropriate data members and member functions in these classes to solve the following queries:
a) Get the total count of the employees in the company. b) Get the count of the developers and testers. c) Given an employee id, display the details.

From: K H, Shivaprasada
Sent: Friday, July 5, 2024 12:01 PM
To: Kumar, Nishant <nishant.k.kumar@capgemini.com>; Kumar, Anubhav <anubhav.b.kumar@capgemini.com>; Mandal, Swapnanil <swapnanil.mandal@capgemini.com>; Tripathy, Sumit <sumit.tripathy@capgemini.com>; Kumar, Tapan <tapan.kumar@capgemini.com>; Pratap, Shakti <shakti.pratap@capgemini.com>; Upadhyay, Kaustubh <kaustubh.upadhyay@capgemini.com>; Kumar, Ayush <ayush.f.kumar@capgemini.com>; Sahu,

Satya <satya.sahu@capgemini.com>; Bharti, Om Prakash <om-prakash.bharti@capgemini.com>; Tah, Hritam <hritam.tah@capgemini.com>; Chakrabarty, Arindam <arindam.chakrabarty@capgemini.com>; Gupta, Shahil <shahil.gupta@capgemini.com>; Dubey, Ashutosh <ashutosh.dubey@capgemini.com>; Thakur, Suman <suman.a.thakur@capgemini.com>; Mahato, Rajiv Kumar <rajiv-kumar.mahato@capgemini.com>; Dubey, Saurav Kumar <saurav-kumar.dubey@capgemini.com>; S, Vijay Vignesh <vijay-vignesh.s@capgemini.com>; Chavan, Vedant Mukesh <vedant-mukesh.chavan@capgemini.com>; Verma, Vivek <vivek.b.verma@capgemini.com>; Bouri, Ranjit <ranjit.bouri@capgemini.com>; Behera, Rudra Prasad <rudra-prasad.behera@capgemini.com>; SHARMA, MOHIT <mohit.h.sharma@capgemini.com>; Gupta, Ratan <ratan.gupta@capgemini.com>; Singh, Rishabh <rishabh.d.singh@capgemini.com>; Khuntia, Soumesh <soumesh.khuntia@capgemini.com>; Kanungo, Souravashree <souravashree.kanungo@capgemini.com>; Jain, Parteek <parteek.jain@capgemini.com>; J, SATISH <satish.a.j@capgemini.com>; RAJ, RACHIT <rachit.raj@capgemini.com>; Kumar, Reshav <reshav.kumar@capgemini.com>; Bihari, Madhav <madhav.bihari@capgemini.com>
Cc: ., Vishalarani <vishalarani.vishalarani@capgemini.com>
Subject: RE: Implement Basic Array Operations:

1.A library wants to automate the management of its books and members. The details provided by the library are as follows:

Book Details:

BookTitle
Author
ISBN
MaxCopies
AvailableCopies
Member Details:

MemberName
MemberID
BorrowedBooks (List of borrowed book ISBNs)
The automated software should enable the librarian to perform the following tasks:

Add a new book
Update the number of available copies of a book
Register a new member
Borrow a book
Return a book
Display the details of all books
Display the details of all members
The solution should:

Add appropriate constructors and methods to the classes.
Demonstrate the "has-a" relationship between the library class and the member and book classes.
Use the ArrayList collection class to add the books and the members to the application dynamically.
The complete design of the classes should be inside a package.
Design the client application as a menu-driven application for the various tasks that the librarian performs.

2.A gym wants to automate the management of its members and the classes they attend. The details provided by the gym are as follows:

Class Details:

ClassName
Instructor
MaxParticipants
CurrentParticipants
Member Details:

MemberName
MemberID
EnrolledClasses (List of enrolled class names)
The automated software should enable the gym admin to perform the following tasks:

Add a new class
Update the number of current participants of a class
Register a new member
Enroll a member in a class
Remove a member from a class
Display the details of all classes
Display the details of all members
The solution should:

Add appropriate constructors and methods to the classes.
Demonstrate the "has-a" relationship between the gym class and the member and class classes.
Use the ArrayList collection class to add the classes and the members to the application dynamically.
The complete design of the classes should be inside a package.
Design the client application as a menu-driven application for the various tasks that the gym admin performs.

3.A bookstore management system has the following classes: Book, Purchase, and PurchaseList. A PurchaseList has many Purchase objects for a given customer. A Purchase object is created whenever a book is purchased and contains information about the Book being bought. The Book class includes the title, author, price, and description. The PurchaseList needs to use an ArrayList for storing multiple Purchase objects.

The BillingSystem class has a method called generateInvoice to which an object of PurchaseList is provided. This method calculates the grand total for the purchase list by considering the quantity and discount (from Purchase) and the price (from Book). Each time a purchase is made, some taxes need to be applied. For this purpose, there is a TaxCalculator class that helps in calculating the sales tax. The current tax rate is 8%, which can be revised frequently.

Requirements
Create appropriate constructors and methods for the classes.
Demonstrate the "has-a" relationship between PurchaseList and Purchase, and between Purchase and Book.
Use the ArrayList collection class to manage the list of purchases dynamically.
Design the classes inside a package.

Design a client application with a menu-driven interface for various tasks performed by the bookstore admin.

4.A grocery store management system has the following classes: Product, Purchase, and PurchaseList. A PurchaseList has many Purchase objects for a given customer. A Purchase object is created whenever a product is purchased and contains information about the Product being bought. The Product class includes the name, price, and description. The PurchaseList needs to use a LinkedList for storing multiple Purchase objects.

The BillingSystem class has a method called generateInvoice to which an object of PurchaseList is provided. This method calculates the grand total for the purchase list by considering the quantity and discount (from Purchase) and the price (from Product). Each time a purchase is made, some taxes need to be applied. For this purpose, there is a TaxCalculator class that helps in calculating the sales tax. The current tax rate is 7%, which can be revised frequently.

Requirements
Create appropriate constructors and methods for the classes.
Demonstrate the "has-a" relationship between PurchaseList and Purchase, and between Purchase and Product.
Use the LinkedList collection class to manage the list of purchases dynamically.
Design the classes inside a package.
Design a client application with a menu-driven interface for various tasks performed by the grocery store admin.

5.A student management system has the following classes: Student, Course, and StudentSystem. The StudentSystem class maintains a collection of unique students using a TreeSet. Each Student has a name, student ID, and a set of courses they are enrolled in. Each Course has a course code and a course name.

The StudentSystem class has methods for:

Adding a new student to the system.
Adding a new course for a student.
Removing a course for a student.
Displaying all students in the system.
Displaying all courses for a specific student.
Displaying all unique courses across all students.
Requirements
Create appropriate constructors and methods for the classes.
Demonstrate the "has-a" relationship between Student and Course, and between StudentSystem and Student.
Use the TreeSet collection class to manage the list of unique students.
Design the classes inside a package.
Design a client application with a menu-driven interface for various tasks performed by the student system admin.

6.A movie database system has the following classes: Movie, Actor, and MovieDatabase. The MovieDatabase class maintains a collection of unique movies using a HashSet. Each Movie has a title, director, release year, and a set of actors starring in it. Each Actor has a name and a birth year.

The MovieDatabase class has methods for:

Adding a new movie to the database.
Adding an actor to a movie.
Removing an actor from a movie.
Displaying all movies in the database.
Displaying all actors for a specific movie.
Displaying all unique actors in the entire database.
Requirements
Create appropriate constructors and methods for the classes.
Demonstrate the "has-a" relationship between Movie and Actor, and between MovieDatabase and Movie.
Use the HashSet collection class to manage the list of unique movies.
Design the classes inside a package.
Design a client application with a menu-driven interface for various tasks performed by the movie database admin.

7.A task management system has the following classes: Task, TaskManager, and TaskSystem. The TaskSystem class maintains a queue of tasks using a Queue interface, implemented typically with LinkedList. Each Task has a title, description, priority, and status (e.g., pending, completed).

The TaskManager class has methods for:

Adding a new task to the system.
Removing the next task from the queue (removing the highest priority task first).
Marking a task as completed.
Displaying all tasks in the system.
Displaying the next task to be completed without removing it.
Displaying all completed tasks.
Requirements
Create appropriate constructors and methods for the classes.
Demonstrate the "has-a" relationship between Task and TaskManager, and between TaskManager and TaskSystem.
Use the Queue interface (implemented with LinkedList) to manage tasks in a first-in, first-out (FIFO) manner.
Design the classes inside a package.
Design a client application with a menu-driven interface for various tasks performed by the task manager.

8.A university administration wants to develop a Student Records Management System using Java with the following classes: Student, Course, StudentRecord, and UniversitySystem.

Student Class
Attributes: studentId, name, email, phone.
Methods: Getters, setters, and toString() method.
Course Class
Attributes: courseCode, courseName, creditHours.
Methods: Getters, setters, and toString() method.
StudentRecord Class
Attributes: student, coursesEnrolled (a Map of courses and grades).

Methods:
addCourse(Course course, double grade): Adds a course and its grade to the record.
removeCourse(Course course): Removes a course from the record.
displayCourses(): Displays all courses enrolled along with grades.
UniversitySystem Class
Attributes: studentRecords (a Map with studentId as key and StudentRecord as value).
Methods:
addStudent(Student student): Adds a new student to the system.
removeStudent(String studentId): Removes a student from the system.
enrollStudent(String studentId, Course course): Enrolls a student in a course.
dropStudentCourse(String studentId, Course course): Drops a course for a student.
displayStudentCourses(String studentId): Displays all courses enrolled by a student.
Requirements
Implement appropriate constructors and methods for each class.
Ensure that the UniversitySystem class manages student records using a Map (typically HashMap), with studentId as the key and StudentRecord as the value.
Demonstrate the "has-a" relationship between UniversitySystem and StudentRecord, and between StudentRecord and Student and Course.
Design the classes inside a package named universitySystem.
Design a client application (UniversitySystemApp) with a menu-driven interface for performing operations on student records.

9.Design a Java program for managing a book catalog using LinkedHashMap to maintain insertion order. The program should include the following classes: Book, BookCatalog, and CatalogSystem.

Book Class
Attributes: isbn (unique identifier), title, author, price.
Methods: Getters, setters, and toString() method.
BookCatalog Class
Attributes: catalog (a LinkedHashMap with isbn as key and Book as value).
Methods:
addBook(Book book): Adds a new book to the catalog.
removeBook(String isbn): Removes a book from the catalog.
displayAllBooks(): Displays all books in the catalog in the insertion order.
CatalogSystem Class
Attributes: bookCatalog (an instance of BookCatalog).
Methods:
addNewBook(String isbn, String title, String author, double price): Creates a new book object and adds it to the catalog.
removeBook(String isbn): Removes a book from the catalog.
displayAllBooks(): Displays all books in the catalog.
Requirements
Implement appropriate constructors and methods for each class.
Use LinkedHashMap to maintain the insertion order of books in the catalog.
Demonstrate the "has-a" relationship between CatalogSystem and BookCatalog, and between BookCatalog and Book.
Design the classes inside a package named bookCatalog.
Design a client application (CatalogSystemApp) with a menu-driven interface for performing operations on the book catalog.

10.Design a Java program for managing a dictionary using TreeMap to maintain natural ordering by keys (words). The program should include the following classes: Word, Dictionary, and DictionarySystem.

Word Class
Attributes: word (unique identifier), meaning.
Methods: Getters, setters, and toString() method.
Dictionary Class
Attributes: dictionary (a TreeMap with word as key and Word as value).
Methods:
addWord(Word word): Adds a new word to the dictionary.
removeWord(String word): Removes a word from the dictionary.
searchWord(String word): Searches for a word in the dictionary and displays its meaning.
displayAllWords(): Displays all words in the dictionary in alphabetical order.
DictionarySystem Class
Attributes: dictionary (an instance of Dictionary).
Methods:
addNewWord(String word, String meaning): Creates a new Word object and adds it to the dictionary.
removeWord(String word): Removes a word from the dictionary.
searchWord(String word): Searches for a word in the dictionary and displays its meaning.
displayAllWords(): Displays all words in the dictionary.
Requirements
Implement appropriate constructors and methods for each class.
Use TreeMap to maintain natural ordering by keys (words) in the dictionary.
Demonstrate the "has-a" relationship between DictionarySystem and Dictionary, and between Dictionary and Word.
Design the classes inside a package named dictionary.
Design a client application (DictionarySystemApp) with a menu-driven interface for performing operations on the dictionary.

From: K H, Shivaprasada
Sent: Friday, July 5, 2024 11:58 AM
To: Kumar, Nishant <nishant.k.kumar@capgemini.com>; Kumar, Anubhav <anubhav.b.kumar@capgemini.com>; Mandal, Swapnanil <swapnanil.mandal@capgemini.com>; Tripathy, Sumit <sumit.tripathy@capgemini.com>; Kumar, Tapan <tapan.kumar@capgemini.com>; Pratap, Shakti <shakti.pratap@capgemini.com>; Upadhyay, Kaustubh <kaustubh.upadhyay@capgemini.com>; Kumar, Ayush <ayush.f.kumar@capgemini.com>; Sahu, Satya <satya.sahu@capgemini.com>; Bharti, Om Prakash <om-prakash.bharti@capgemini.com>; Tah, Hritam <hritam.tah@capgemini.com>; Chakrabarty, Arindam <arindam.chakrabarty@capgemini.com>; Gupta, Shahil <shahil.gupta@capgemini.com>; Dubey, Ashutosh <ashutosh.dubey@capgemini.com>; Thakur, Suman <suman.a.thakur@capgemini.com>; Mahato, Rajiv Kumar <rajiv-kumar.mahato@capgemini.com>; Dubey, Saurav Kumar <saurav-kumar.dubey@capgemini.com>; S, Vijay Vignesh <vijay-vignesh.s@capgemini.com>; Chavan, Vedant Mukesh <vedant-mukesh.chavan@capgemini.com>; Verma, Vivek <vivek.b.verma@capgemini.com>; Bouri, Ranjit <ranjit.bouri@capgemini.com>; Behera, Rudra Prasad <rudra-prasad.behera@capgemini.com>; SHARMA, MOHIT <mohit.h.sharma@capgemini.com>; Gupta, Ratan <ratan.gupta@capgemini.com>; Singh, Rishabh <rishabh.d.singh@capgemini.com>; Khuntia, Soumesh <soumesh.khuntia@capgemini.com>; Kanungo, Souravashree <souravashree.kanungo@capgemini.com>; Jain, Parteek <parteek.jain@capgemini.com>; J, SATISH <satish.a.j@capgemini.com>; RAJ, RACHIT

<rachit.raj@capgemini.com>; Kumar, Reshav <reshav.kumar@capgemini.com>; Bihari, Madhav <madhav.bihari@capgemini.com>
Cc: ., Vishalarani <vishalarani.vishalarani@capgemini.com>
Subject: Implement Basic Array Operations:

Problem Statement: Efficient Array Manipulations and Big O Analysis in Java
Objective: To implement and optimize various array operations in Java, analyze their time and space complexities using Big O notation, and demonstrate the ability to select the most efficient algorithms for given tasks.

Requirements
Implement Basic Array Operations:

Write Java methods to perform the following operations on arrays:
Insertion: Insert an element at a specific position.
Deletion: Delete an element from a specific position.
Search: Search for an element and return its index if found.
Reverse: Reverse the array.
Find Maximum/Minimum: Find the maximum and minimum elements in the array.

Implement Advanced Array Operations:

Write Java methods to perform the following advanced operations on arrays:
Merge: Merge two sorted arrays into one sorted array.
Remove Duplicates: Remove duplicate elements from a sorted array.
Find Duplicates: Identify duplicate elements in the array.
Analyze Time and Space Complexities:

For each array operation, analyze and document the time and space complexities using Big O notation.
Optimize and Compare:

Implement optimized versions of the operations where possible.
Compare the performance of the optimized versions with the initial implementations.
Performance Testing:

Use a large dataset to demonstrate the efficiency of your implementations.
Provide a report comparing the performance of different methods, supported by empirical data (e.g., execution time).
Implementation Steps
Setup:

Create a new Java project.
Define a class ArrayOperations to encapsulate the array operations.
Implement Basic Array Operations:

Implement the basic versions of the required operations.
Ensure proper handling of edge cases (e.g., inserting/deleting at the bounds of the array).
Implement Advanced Array Operations:

Implement the advanced versions of the required operations.

Ensure the methods work efficiently for large datasets.
Analyze and Document Complexities:

Write comments or a separate documentation file explaining the time and space complexities of each method using Big O notation.
Optimize Operations:

Identify opportunities to optimize the initial implementations.
Implement optimized versions and explain the improvements.
Performance Testing:

Write a test class ArrayOperationsTest to benchmark the performance of the different implementations.
Use large arrays to illustrate the efficiency gains of the optimized methods.
Report:

Compile a report that includes:
Descriptions of each operation and its implementation.
Analysis of time and space complexities.
Comparison of initial and optimized implementations.
Performance data from benchmarking.

2.Problem Statement: Array Manipulations and Big O Analysis
Objective: Implement various operations on arrays in Java, analyze their time and space complexities using Big O notation, and demonstrate the ability to select efficient algorithms for given tasks.

Requirements
Implement Basic Array Operations:

Write Java methods to perform the following operations on arrays:
Insertion: Insert an element at the beginning, middle, and end.
Deletion: Delete an element from the beginning, middle, and end.
Search: Linear search for an element and return its index if found.
Reverse: Reverse the array.
Find Maximum/Minimum: Find the maximum and minimum elements in the array.
Implement Advanced Array Operations:

Write Java methods to perform the following advanced operations on arrays:
Sort: Sort the array using both Bubble Sort and Quick Sort.
Rotate: Rotate the array to the right by a given number of steps.
Find Pair with Sum: Find all pairs of elements that sum to a specific value.
Analyze Time and Space Complexities:

For each array operation, analyze and document the time and space complexities using Big O notation.
Performance Testing:

Use a large dataset to demonstrate the efficiency of your implementations.
Provide a report comparing the performance of different methods, supported by empirical data (e.g., execution time).

Implementation Steps
Setup:

Create a new Java project.
Define a class ArrayOperations to encapsulate the array operations.
Implement Basic Array Operations:

Implement the basic versions of the required operations.
Ensure proper handling of edge cases (e.g., inserting/deleting at the bounds of the array).
Implement Advanced Array Operations:
Implement the advanced versions of the required operations.
Ensure the methods work efficiently for large datasets.

Analyze and Document Complexities:
Write comments or a separate documentation file explaining the time and space complexities of each method using Big O notation.

Performance Testing:
Write a test class ArrayOperationsTest to benchmark the performance of the different implementations.
Use large arrays to illustrate the efficiency gains of the optimized methods.
Report:

Compile a report that includes:
Descriptions of each operation and its implementation.
Analysis of time and space complexities.
Comparison of initial and optimized implementations.
Performance data from benchmarking.

3.Problem Statement: Sorting Algorithms and Their Analysis in Java
Objective: Implement various sorting algorithms in Java, analyze their time and space complexities using Big O notation, and demonstrate the ability to choose the most efficient algorithm for different datasets and scenarios.

Requirements
Implement Basic Sorting Algorithms:

Write Java methods to perform the following sorting algorithms:
Bubble Sort
Selection Sort
Insertion Sort
Implement Advanced Sorting Algorithms:
Write Java methods to perform the following advanced sorting algorithms:
Merge Sort
Quick Sort
Heap Sort

Implement Special Sorting Algorithms:
Write Java methods to perform the following special sorting algorithms:
Counting Sort
Radix Sort

Bucket Sort

Analyze Time and Space Complexities:
For each sorting algorithm, analyze and document the time and space complexities using Big O notation.

Optimize and Compare:
Implement optimized versions of the sorting algorithms where applicable.
Compare the performance of different sorting algorithms on various datasets.

Performance Testing:
Use datasets of varying sizes and characteristics to demonstrate the efficiency of your implementations.
Provide a report comparing the performance of different sorting algorithms, supported by empirical data (e.g., execution time).
Implementation Steps
Setup:
Create a new Java project.
Define a class SortingAlgorithms to encapsulate the sorting algorithms.

Implement Basic Sorting Algorithms:
Implement the basic versions of Bubble Sort, Selection Sort, and Insertion Sort.
Ensure proper handling of edge cases (e.g., sorting an already sorted array).
Implement Advanced Sorting Algorithms:

Implement the advanced versions of Merge Sort, Quick Sort, and Heap Sort.
Ensure the methods work efficiently for large datasets.

Implement Special Sorting Algorithms:
Implement the special versions of Counting Sort, Radix Sort, and Bucket Sort.
Ensure the methods handle non-comparative sorting scenarios.

Analyze and Document Complexities:
Write comments or a separate documentation file explaining the time and space complexities of each method using Big O notation.

Optimize Algorithms:
Identify opportunities to optimize the initial implementations.
Implement optimized versions and explain the improvements.

Performance Testing:
Write a test class SortingAlgorithmsTest to benchmark the performance of the different implementations.
Use datasets with different sizes and characteristics to illustrate the efficiency gains of the optimized methods.
Report:

Compile a report that includes:
Descriptions of each sorting algorithm and its implementation.
Analysis of time and space complexities.
Comparison of initial and optimized implementations.

Performance data from benchmarking.

4.Problem Statement: List Operations and Their Analysis in Java
Objective: Implement various operations on different types of lists in Java, analyze their time and space complexities using Big O notation, and demonstrate the ability to choose the most efficient data structure for different tasks.

Requirements
Implement Basic List Operations:
Write Java methods to perform the following operations on lists:
Add: Add an element at the beginning, middle, and end of the list.
Remove: Remove an element from the beginning, middle, and end of the list.
Get: Retrieve an element by its index.
Set: Update an element at a specific index.
Contains: Check if the list contains a specific element.

Implement Different Types of Lists:
Implement and compare the following types of lists:
ArrayList: Backed by a dynamic array.
LinkedList: Backed by a doubly-linked list.
CustomLinkedList: Implement a singly-linked list from scratch.
Analyze Time and Space Complexities:

For each list operation, analyze and document the time and space complexities using Big O notation.

Compare List Implementations:
Compare the performance of ArrayList, LinkedList, and CustomLinkedList for various operations.
Determine the most efficient list implementation for different scenarios.

Performance Testing:
Use datasets of varying sizes and characteristics to demonstrate the efficiency of your implementations.
Provide a report comparing the performance of different list implementations, supported by empirical data (e.g., execution time).
Implementation Steps
Setup:
Create a new Java project.
Define a class ListOperations to encapsulate the list operations.

Implement Basic List Operations:
Implement the basic versions of the required operations for ArrayList and LinkedList.
Ensure proper handling of edge cases (e.g., adding/removing elements at the bounds of the list).
Implement CustomLinkedList:
Implement a singly-linked list from scratch with methods for adding, removing, getting, setting, and checking elements.
Analyze and Document Complexities:

Write comments or a separate documentation file explaining the time and space complexities of each method using Big O notation.

Compare List Implementations:
Benchmark the performance of ArrayList, LinkedList, and CustomLinkedList for various operations.
Identify the most efficient list implementation for different scenarios.

Performance Testing:
Write a test class ListOperationsTest to benchmark the performance of the different list implementations.
Use datasets with different sizes and characteristics to illustrate the efficiency gains of the optimized methods.
Report:

Compile a report that includes:
Descriptions of each list operation and its implementation.
Analysis of time and space complexities.
Comparison of different list implementations.
Performance data from benchmarking.

1.Create an interface called LibraryRecord with the functions addRecord, displayRecords, and searchRecords. Create a class called BookRecord that implements the interface. The class should maintain a record of all the books, comprising the book title, author name, and number of copies available.
Define the functions of the interface appropriately.
Now the program should accept an author's name as an input, and within the searchRecords function, display the titles of all books by that author along with the number of copies available. Also, display the title of the book with the highest number of copies available.

2.Create an interface called PatientRecord with the methods addPatient, displayPatients, and searchPatientByDoctor. Create a class called HospitalRecord that implements the interface. The class should maintain a record of all the patients, including the patient name, doctor name, and diagnosis.
Define the methods of the interface appropriately.
The program should accept a doctor's name as input, and within the searchPatientByDoctor method, display the names of all patients being treated by that doctor along with their diagnoses. Also, display the name of the patient with the most diagnoses.

3.Create an interface called Inventory with methods addItem, displayItems, and searchItem. Create a class called StoreInventory that implements the interface. The class should maintain a record of all items in the store, including item name, category, and quantity.
Define the methods of the interface appropriately.
The program should accept a category as input, and within the searchItem method, display the names of all items belonging to that category along with their quantities. Also, display the item with the highest quantity in that category.

From: K H, Shivaprasada
Sent: Friday, July 5, 2024 12:03 PM
To: Kumar, Nishant <nishant.k.kumar@capgemini.com>; Kumar, Anubhav <anubhav.b.kumar@capgemini.com>; Mandal, Swapnanil <swapnanil.mandal@capgemini.com>; Tripathy, Sumit <sumit.tripathy@capgemini.com>; Kumar, Tapan <tapan.kumar@capgemini.com>;

Pratap, Shakti <shakti.pratap@capgemini.com>; Upadhyay, Kaustubh <kaustubh.upadhyay@capgemini.com>; Kumar, Ayush <ayush.f.kumar@capgemini.com>; Sahu, Satya <satya.sahu@capgemini.com>; Bharti, Om Prakash <om-prakash.bharti@capgemini.com>; Tah, Hritam <hritam.tah@capgemini.com>; Chakrabarty, Arindam <arindam.chakrabarty@capgemini.com>; Gupta, Shahil <shahil.gupta@capgemini.com>; Dubey, Ashutosh <ashutosh.dubey@capgemini.com>; Thakur, Suman <suman.a.thakur@capgemini.com>; Mahato, Rajiv Kumar <rajiv-kumar.mahato@capgemini.com>; Dubey, Saurav Kumar <saurav-kumar.dubey@capgemini.com>; S, Vijay Vignesh <vijay-vignesh.s@capgemini.com>; Chavan, Vedant Mukesh <vedant-mukesh.chavan@capgemini.com>; Verma, Vivek <vivek.b.verma@capgemini.com>; Bouri, Ranjit <ranjit.bouri@capgemini.com>; Behera, Rudra Prasad <rudra-prasad.behera@capgemini.com>; SHARMA, MOHIT <mohit.h.sharma@capgemini.com>; Gupta, Ratan <ratan.gupta@capgemini.com>; Singh, Rishabh <rishabh.d.singh@capgemini.com>; Khuntia, Soumesh <soumesh.khuntia@capgemini.com>; Kanungo, Souravashree <souravashree.kanungo@capgemini.com>; Jain, Parteek <parteek.jain@capgemini.com>; J, SATISH <satish.a.j@capgemini.com>; RAJ, RACHIT <rachit.raj@capgemini.com>; Kumar, Reshav <reshav.kumar@capgemini.com>; Bihari, Madhav <madhav.bihari@capgemini.com>
Cc: ., Vishalarani <vishalarani.vishalarani@capgemini.com>
Subject: RE: Implement Basic Array Operations:

Problem Statement:
1.A hospital wants to maintain details of its staff members. Some of these staff members are doctors, and some are nurses. Both types of staff members have certain common attributes like name, age, id, and department. Additionally, doctors have attributes like specialization and consultation fee, while nurses have attributes like shift timings and hourly wage.
Demonstrate the above scenario by having Staff as the base class and Doctor and Nurse as the subclasses. Let the search method in the Staff class be an abstract method. Have appropriate data members and member functions in these classes to solve the following queries:
a) Get the total count of the staff members for the hospital. b) Get the count of the doctors and nurses. c) Given a staff id, display the details.

2.A library wants to maintain details of its members. Some of these members are students, and some are faculty members. Both types of members have certain common attributes like name, age, id, and membership type. Additionally, students have attributes like grade and borrowed books, while faculty members have attributes like department and research publications.
Demonstrate the above scenario by having LibraryMember as the base class and StudentMember and FacultyMember as the subclasses. Let the search method in the LibraryMember class be an abstract method. Have appropriate data members and member functions in these classes to solve the following queries:
a) Get the total count of the library members. b) Get the count of the student members and faculty members. c) Given a member id, display the details.

3.A software development company wants to maintain details of its employees. Some of these employees are developers, and some are testers. Both types of employees have certain common attributes like name, age, id, and department. Additionally, developers have attributes like programming languages known and projects handled, while testers have attributes like testing tools known and bugs reported.
Demonstrate the above scenario by having Employee as the base class and Developer and Tester as the subclasses. Let the search method in the Employee class be an abstract method. Have appropriate data members and member functions in these classes to solve the following queries:

a) Get the total count of the employees in the company. b) Get the count of the developers and testers. c) Given an employee id, display the details.

From: K H, Shivaprasada
Sent: Friday, July 5, 2024 12:01 PM
To: Kumar, Nishant <nishant.k.kumar@capgemini.com>; Kumar, Anubhav <anubhav.b.kumar@capgemini.com>; Mandal, Swapnanil <swapnanil.mandal@capgemini.com>; Tripathy, Sumit <sumit.tripathy@capgemini.com>; Kumar, Tapan <tapan.kumar@capgemini.com>; Pratap, Shakti <shakti.pratap@capgemini.com>; Upadhyay, Kaustubh <kaustubh.upadhyay@capgemini.com>; Kumar, Ayush <ayush.f.kumar@capgemini.com>; Sahu, Satya <satya.sahu@capgemini.com>; Bharti, Om Prakash <om-prakash.bharti@capgemini.com>; Tah, Hritam <hritam.tah@capgemini.com>; Chakrabarty, Arindam <arindam.chakrabarty@capgemini.com>; Gupta, Shahil <shahil.gupta@capgemini.com>; Dubey, Ashutosh <ashutosh.dubey@capgemini.com>; Thakur, Suman <suman.a.thakur@capgemini.com>; Mahato, Rajiv Kumar <rajiv-kumar.mahato@capgemini.com>; Dubey, Saurav Kumar <saurav-kumar.dubey@capgemini.com>; S, Vijay Vignesh <vijay-vignesh.s@capgemini.com>; Chavan, Vedant Mukesh <vedant-mukesh.chavan@capgemini.com>; Verma, Vivek <vivek.b.verma@capgemini.com>; Bouri, Ranjit <ranjit.bouri@capgemini.com>; Behera, Rudra Prasad <rudra-prasad.behera@capgemini.com>; SHARMA, MOHIT <mohit.h.sharma@capgemini.com>; Gupta, Ratan <ratan.gupta@capgemini.com>; Singh, Rishabh <rishabh.d.singh@capgemini.com>; Khuntia, Soumesh <soumesh.khuntia@capgemini.com>; Kanungo, Souravashree <souravashree.kanungo@capgemini.com>; Jain, Parteek <parteek.jain@capgemini.com>; J, SATISH <satish.a.j@capgemini.com>; RAJ, RACHIT <rachit.raj@capgemini.com>; Kumar, Reshav <reshav.kumar@capgemini.com>; Bihari, Madhav <madhav.bihari@capgemini.com>
Cc: ., Vishalarani <vishalarani.vishalarani@capgemini.com>
Subject: RE: Implement Basic Array Operations:

1.A library wants to automate the management of its books and members. The details provided by the library are as follows:

Book Details:

BookTitle
Author
ISBN
MaxCopies
AvailableCopies
Member Details:

MemberName
MemberID
BorrowedBooks (List of borrowed book ISBNs)
The automated software should enable the librarian to perform the following tasks:

Add a new book
Update the number of available copies of a book
Register a new member
Borrow a book
Return a book
Display the details of all books

Display the details of all members
The solution should:

Add appropriate constructors and methods to the classes.
Demonstrate the "has-a" relationship between the library class and the member and book classes.
Use the ArrayList collection class to add the books and the members to the application dynamically.
The complete design of the classes should be inside a package.
Design the client application as a menu-driven application for the various tasks that the librarian performs.

2.A gym wants to automate the management of its members and the classes they attend. The details provided by the gym are as follows:

Class Details:

ClassName
Instructor
MaxParticipants
CurrentParticipants
Member Details:

MemberName
MemberID
EnrolledClasses (List of enrolled class names)
The automated software should enable the gym admin to perform the following tasks:

Add a new class
Update the number of current participants of a class
Register a new member
Enroll a member in a class
Remove a member from a class
Display the details of all classes
Display the details of all members
The solution should:

Add appropriate constructors and methods to the classes.
Demonstrate the "has-a" relationship between the gym class and the member and class classes.
Use the ArrayList collection class to add the classes and the members to the application dynamically.
The complete design of the classes should be inside a package.
Design the client application as a menu-driven application for the various tasks that the gym admin performs.

3.A bookstore management system has the following classes: Book, Purchase, and PurchaseList. A PurchaseList has many Purchase objects for a given customer. A Purchase object is created whenever a book is purchased and contains information about the Book being bought. The Book class includes the title, author, price, and description. The PurchaseList needs to use an ArrayList for storing multiple Purchase objects.

The BillingSystem class has a method called generateInvoice to which an object of PurchaseList is provided. This method calculates the grand total for the purchase list by considering the quantity and discount (from Purchase) and the price (from Book). Each time a purchase is made, some taxes

need to be applied. For this purpose, there is a TaxCalculator class that helps in calculating the sales tax. The current tax rate is 8%, which can be revised frequently.

Requirements
Create appropriate constructors and methods for the classes.
Demonstrate the "has-a" relationship between PurchaseList and Purchase, and between Purchase and Book.
Use the ArrayList collection class to manage the list of purchases dynamically.
Design the classes inside a package.
Design a client application with a menu-driven interface for various tasks performed by the bookstore admin.

4.A grocery store management system has the following classes: Product, Purchase, and PurchaseList. A PurchaseList has many Purchase objects for a given customer. A Purchase object is created whenever a product is purchased and contains information about the Product being bought. The Product class includes the name, price, and description. The PurchaseList needs to use a LinkedList for storing multiple Purchase objects.

The BillingSystem class has a method called generateInvoice to which an object of PurchaseList is provided. This method calculates the grand total for the purchase list by considering the quantity and discount (from Purchase) and the price (from Product). Each time a purchase is made, some taxes need to be applied. For this purpose, there is a TaxCalculator class that helps in calculating the sales tax. The current tax rate is 7%, which can be revised frequently.

Requirements
Create appropriate constructors and methods for the classes.
Demonstrate the "has-a" relationship between PurchaseList and Purchase, and between Purchase and Product.
Use the LinkedList collection class to manage the list of purchases dynamically.
Design the classes inside a package.
Design a client application with a menu-driven interface for various tasks performed by the grocery store admin.

5.A student management system has the following classes: Student, Course, and StudentSystem. The StudentSystem class maintains a collection of unique students using a TreeSet. Each Student has a name, student ID, and a set of courses they are enrolled in. Each Course has a course code and a course name.

The StudentSystem class has methods for:

Adding a new student to the system.
Adding a new course for a student.
Removing a course for a student.
Displaying all students in the system.
Displaying all courses for a specific student.
Displaying all unique courses across all students.
Requirements
Create appropriate constructors and methods for the classes.
Demonstrate the "has-a" relationship between Student and Course, and between StudentSystem and Student.

Use the TreeSet collection class to manage the list of unique students.
Design the classes inside a package.
Design a client application with a menu-driven interface for various tasks performed by the student system admin.

6.A movie database system has the following classes: Movie, Actor, and MovieDatabase. The MovieDatabase class maintains a collection of unique movies using a HashSet. Each Movie has a title, director, release year, and a set of actors starring in it. Each Actor has a name and a birth year.

The MovieDatabase class has methods for:

Adding a new movie to the database.
Adding an actor to a movie.
Removing an actor from a movie.
Displaying all movies in the database.
Displaying all actors for a specific movie.
Displaying all unique actors in the entire database.
Requirements
Create appropriate constructors and methods for the classes.
Demonstrate the "has-a" relationship between Movie and Actor, and between MovieDatabase and Movie.
Use the HashSet collection class to manage the list of unique movies.
Design the classes inside a package.
Design a client application with a menu-driven interface for various tasks performed by the movie database admin.

7.A task management system has the following classes: Task, TaskManager, and TaskSystem. The TaskSystem class maintains a queue of tasks using a Queue interface, implemented typically with LinkedList. Each Task has a title, description, priority, and status (e.g., pending, completed).

The TaskManager class has methods for:

Adding a new task to the system.
Removing the next task from the queue (removing the highest priority task first).
Marking a task as completed.
Displaying all tasks in the system.
Displaying the next task to be completed without removing it.
Displaying all completed tasks.
Requirements
Create appropriate constructors and methods for the classes.
Demonstrate the "has-a" relationship between Task and TaskManager, and between TaskManager and TaskSystem.
Use the Queue interface (implemented with LinkedList) to manage tasks in a first-in, first-out (FIFO) manner.
Design the classes inside a package.
Design a client application with a menu-driven interface for various tasks performed by the task manager.

8.A university administration wants to develop a Student Records Management System using Java with the following classes: Student, Course, StudentRecord, and UniversitySystem.

Student Class
Attributes: studentId, name, email, phone.
Methods: Getters, setters, and toString() method.
Course Class
Attributes: courseCode, courseName, creditHours.
Methods: Getters, setters, and toString() method.
StudentRecord Class
Attributes: student, coursesEnrolled (a Map of courses and grades).
Methods:
addCourse(Course course, double grade): Adds a course and its grade to the record.
removeCourse(Course course): Removes a course from the record.
displayCourses(): Displays all courses enrolled along with grades.
UniversitySystem Class
Attributes: studentRecords (a Map with studentId as key and StudentRecord as value).
Methods:
addStudent(Student student): Adds a new student to the system.
removeStudent(String studentId): Removes a student from the system.
enrollStudent(String studentId, Course course): Enrolls a student in a course.
dropStudentCourse(String studentId, Course course): Drops a course for a student.
displayStudentCourses(String studentId): Displays all courses enrolled by a student.
Requirements
Implement appropriate constructors and methods for each class.
Ensure that the UniversitySystem class manages student records using a Map (typically HashMap), with studentId as the key and StudentRecord as the value.
Demonstrate the "has-a" relationship between UniversitySystem and StudentRecord, and between StudentRecord and Student and Course.
Design the classes inside a package named universitySystem.
Design a client application (UniversitySystemApp) with a menu-driven interface for performing operations on student records.

9.Design a Java program for managing a book catalog using LinkedHashMap to maintain insertion order. The program should include the following classes: Book, BookCatalog, and CatalogSystem.

Book Class
Attributes: isbn (unique identifier), title, author, price.
Methods: Getters, setters, and toString() method.
BookCatalog Class
Attributes: catalog (a LinkedHashMap with isbn as key and Book as value).
Methods:
addBook(Book book): Adds a new book to the catalog.
removeBook(String isbn): Removes a book from the catalog.
displayAllBooks(): Displays all books in the catalog in the insertion order.
CatalogSystem Class
Attributes: bookCatalog (an instance of BookCatalog).
Methods:
addNewBook(String isbn, String title, String author, double price): Creates a new book object and adds it to the catalog.
removeBook(String isbn): Removes a book from the catalog.
displayAllBooks(): Displays all books in the catalog.
Requirements

Implement appropriate constructors and methods for each class.
Use LinkedHashMap to maintain the insertion order of books in the catalog.
Demonstrate the "has-a" relationship between CatalogSystem and BookCatalog, and between BookCatalog and Book.
Design the classes inside a package named bookCatalog.
Design a client application (CatalogSystemApp) with a menu-driven interface for performing operations on the book catalog.

10.Design a Java program for managing a dictionary using TreeMap to maintain natural ordering by keys (words). The program should include the following classes: Word, Dictionary, and DictionarySystem.

Word Class
Attributes: word (unique identifier), meaning.
Methods: Getters, setters, and toString() method.
Dictionary Class
Attributes: dictionary (a TreeMap with word as key and Word as value).
Methods:
addWord(Word word): Adds a new word to the dictionary.
removeWord(String word): Removes a word from the dictionary.
searchWord(String word): Searches for a word in the dictionary and displays its meaning.
displayAllWords(): Displays all words in the dictionary in alphabetical order.
DictionarySystem Class
Attributes: dictionary (an instance of Dictionary).
Methods:
addNewWord(String word, String meaning): Creates a new Word object and adds it to the dictionary.
removeWord(String word): Removes a word from the dictionary.
searchWord(String word): Searches for a word in the dictionary and displays its meaning.
displayAllWords(): Displays all words in the dictionary.
Requirements
Implement appropriate constructors and methods for each class.
Use TreeMap to maintain natural ordering by keys (words) in the dictionary.
Demonstrate the "has-a" relationship between DictionarySystem and Dictionary, and between Dictionary and Word.
Design the classes inside a package named dictionary.
Design a client application (DictionarySystemApp) with a menu-driven interface for performing operations on the dictionary.

From: K H, Shivaprasada
Sent: Friday, July 5, 2024 11:58 AM
To: Kumar, Nishant <nishant.k.kumar@capgemini.com>; Kumar, Anubhav <anubhav.b.kumar@capgemini.com>; Mandal, Swapnanil <swapnanil.mandal@capgemini.com>; Tripathy, Sumit <sumit.tripathy@capgemini.com>; Kumar, Tapan <tapan.kumar@capgemini.com>; Pratap, Shakti <shakti.pratap@capgemini.com>; Upadhyay, Kaustubh <kaustubh.upadhyay@capgemini.com>; Kumar, Ayush <ayush.f.kumar@capgemini.com>; Sahu, Satya <satya.sahu@capgemini.com>; Bharti, Om Prakash <om-prakash.bharti@capgemini.com>; Tah, Hritam <hritam.tah@capgemini.com>; Chakrabarty, Arindam <arindam.chakrabarty@capgemini.com>; Gupta, Shahil <shahil.gupta@capgemini.com>; Dubey, Ashutosh <ashutosh.dubey@capgemini.com>; Thakur, Suman <suman.a.thakur@capgemini.com>; Mahato, Rajiv Kumar <rajiv-kumar.mahato@capgemini.com>; Dubey, Saurav Kumar <saurav-

kumar.dubey@capgemini.com>; S, Vijay Vignesh <vijay-vignesh.s@capgemini.com>; Chavan, Vedant Mukesh <vedant-mukesh.chavan@capgemini.com>; Verma, Vivek <vivek.b.verma@capgemini.com>; Bouri, Ranjit <ranjit.bouri@capgemini.com>; Behera, Rudra Prasad <rudra-prasad.behera@capgemini.com>; SHARMA, MOHIT <mohit.h.sharma@capgemini.com>; Gupta, Ratan <ratan.gupta@capgemini.com>; Singh, Rishabh <rishabh.d.singh@capgemini.com>; Khuntia, Soumesh <soumesh.khuntia@capgemini.com>; Kanungo, Souravashree <souravashree.kanungo@capgemini.com>; Jain, Parteek <parteek.jain@capgemini.com>; J, SATISH <satish.a.j@capgemini.com>; RAJ, RACHIT <rachit.raj@capgemini.com>; Kumar, Reshav <reshav.kumar@capgemini.com>; Bihari, Madhav <madhav.bihari@capgemini.com>
Cc: ., Vishalarani <vishalarani.vishalarani@capgemini.com>
Subject: Implement Basic Array Operations:

Problem Statement: Efficient Array Manipulations and Big O Analysis in Java
Objective: To implement and optimize various array operations in Java, analyze their time and space complexities using Big O notation, and demonstrate the ability to select the most efficient algorithms for given tasks.

Requirements
Implement Basic Array Operations:

Write Java methods to perform the following operations on arrays:
Insertion: Insert an element at a specific position.
Deletion: Delete an element from a specific position.
Search: Search for an element and return its index if found.
Reverse: Reverse the array.
Find Maximum/Minimum: Find the maximum and minimum elements in the array.

Implement Advanced Array Operations:

Write Java methods to perform the following advanced operations on arrays:
Merge: Merge two sorted arrays into one sorted array.
Remove Duplicates: Remove duplicate elements from a sorted array.
Find Duplicates: Identify duplicate elements in the array.
Analyze Time and Space Complexities:

For each array operation, analyze and document the time and space complexities using Big O notation.
Optimize and Compare:

Implement optimized versions of the operations where possible.
Compare the performance of the optimized versions with the initial implementations.
Performance Testing:

Use a large dataset to demonstrate the efficiency of your implementations.
Provide a report comparing the performance of different methods, supported by empirical data (e.g., execution time).
Implementation Steps
Setup:

Create a new Java project.

Define a class ArrayOperations to encapsulate the array operations.
Implement Basic Array Operations:

Implement the basic versions of the required operations.
Ensure proper handling of edge cases (e.g., inserting/deleting at the bounds of the array).
Implement Advanced Array Operations:

Implement the advanced versions of the required operations.
Ensure the methods work efficiently for large datasets.
Analyze and Document Complexities:

Write comments or a separate documentation file explaining the time and space complexities of each method using Big O notation.
Optimize Operations:

Identify opportunities to optimize the initial implementations.
Implement optimized versions and explain the improvements.
Performance Testing:

Write a test class ArrayOperationsTest to benchmark the performance of the different implementations.
Use large arrays to illustrate the efficiency gains of the optimized methods.
Report:

Compile a report that includes:
Descriptions of each operation and its implementation.
Analysis of time and space complexities.
Comparison of initial and optimized implementations.
Performance data from benchmarking.

2.Problem Statement: Array Manipulations and Big O Analysis
Objective: Implement various operations on arrays in Java, analyze their time and space complexities using Big O notation, and demonstrate the ability to select efficient algorithms for given tasks.

Requirements
Implement Basic Array Operations:

Write Java methods to perform the following operations on arrays:
Insertion: Insert an element at the beginning, middle, and end.
Deletion: Delete an element from the beginning, middle, and end.
Search: Linear search for an element and return its index if found.
Reverse: Reverse the array.
Find Maximum/Minimum: Find the maximum and minimum elements in the array.
Implement Advanced Array Operations:

Write Java methods to perform the following advanced operations on arrays:
Sort: Sort the array using both Bubble Sort and Quick Sort.
Rotate: Rotate the array to the right by a given number of steps.
Find Pair with Sum: Find all pairs of elements that sum to a specific value.
Analyze Time and Space Complexities:

For each array operation, analyze and document the time and space complexities using Big O notation.
Performance Testing:

Use a large dataset to demonstrate the efficiency of your implementations.
Provide a report comparing the performance of different methods, supported by empirical data (e.g., execution time).
Implementation Steps
Setup:

Create a new Java project.
Define a class ArrayOperations to encapsulate the array operations.
Implement Basic Array Operations:

Implement the basic versions of the required operations.
Ensure proper handling of edge cases (e.g., inserting/deleting at the bounds of the array).
Implement Advanced Array Operations:
Implement the advanced versions of the required operations.
Ensure the methods work efficiently for large datasets.

Analyze and Document Complexities:
Write comments or a separate documentation file explaining the time and space complexities of each method using Big O notation.

Performance Testing:
Write a test class ArrayOperationsTest to benchmark the performance of the different implementations.
Use large arrays to illustrate the efficiency gains of the optimized methods.
Report:

Compile a report that includes:
Descriptions of each operation and its implementation.
Analysis of time and space complexities.
Comparison of initial and optimized implementations.
Performance data from benchmarking.

3.Problem Statement: Sorting Algorithms and Their Analysis in Java
Objective: Implement various sorting algorithms in Java, analyze their time and space complexities using Big O notation, and demonstrate the ability to choose the most efficient algorithm for different datasets and scenarios.

Requirements
Implement Basic Sorting Algorithms:

Write Java methods to perform the following sorting algorithms:
Bubble Sort
Selection Sort
Insertion Sort
Implement Advanced Sorting Algorithms:
Write Java methods to perform the following advanced sorting algorithms:

Merge Sort
Quick Sort
Heap Sort

Implement Special Sorting Algorithms:
Write Java methods to perform the following special sorting algorithms:
Counting Sort
Radix Sort
Bucket Sort

Analyze Time and Space Complexities:
For each sorting algorithm, analyze and document the time and space complexities using Big O notation.

Optimize and Compare:
Implement optimized versions of the sorting algorithms where applicable.
Compare the performance of different sorting algorithms on various datasets.

Performance Testing:
Use datasets of varying sizes and characteristics to demonstrate the efficiency of your implementations.
Provide a report comparing the performance of different sorting algorithms, supported by empirical data (e.g., execution time).
Implementation Steps
Setup:
Create a new Java project.
Define a class SortingAlgorithms to encapsulate the sorting algorithms.

Implement Basic Sorting Algorithms:
Implement the basic versions of Bubble Sort, Selection Sort, and Insertion Sort.
Ensure proper handling of edge cases (e.g., sorting an already sorted array).
Implement Advanced Sorting Algorithms:

Implement the advanced versions of Merge Sort, Quick Sort, and Heap Sort.
Ensure the methods work efficiently for large datasets.

Implement Special Sorting Algorithms:
Implement the special versions of Counting Sort, Radix Sort, and Bucket Sort.
Ensure the methods handle non-comparative sorting scenarios.

Analyze and Document Complexities:
Write comments or a separate documentation file explaining the time and space complexities of each method using Big O notation.

Optimize Algorithms:
Identify opportunities to optimize the initial implementations.
Implement optimized versions and explain the improvements.

Performance Testing:
Write a test class SortingAlgorithmsTest to benchmark the performance of the different implementations.

Use datasets with different sizes and characteristics to illustrate the efficiency gains of the optimized methods.
Report:

Compile a report that includes:
Descriptions of each sorting algorithm and its implementation.
Analysis of time and space complexities.
Comparison of initial and optimized implementations.
Performance data from benchmarking.

4.Problem Statement: List Operations and Their Analysis in Java
Objective: Implement various operations on different types of lists in Java, analyze their time and space complexities using Big O notation, and demonstrate the ability to choose the most efficient data structure for different tasks.

Requirements
Implement Basic List Operations:
Write Java methods to perform the following operations on lists:
Add: Add an element at the beginning, middle, and end of the list.
Remove: Remove an element from the beginning, middle, and end of the list.
Get: Retrieve an element by its index.
Set: Update an element at a specific index.
Contains: Check if the list contains a specific element.

Implement Different Types of Lists:
Implement and compare the following types of lists:
ArrayList: Backed by a dynamic array.
LinkedList: Backed by a doubly-linked list.
CustomLinkedList: Implement a singly-linked list from scratch.
Analyze Time and Space Complexities:

For each list operation, analyze and document the time and space complexities using Big O notation.

Compare List Implementations:
Compare the performance of ArrayList, LinkedList, and CustomLinkedList for various operations.
Determine the most efficient list implementation for different scenarios.

Performance Testing:
Use datasets of varying sizes and characteristics to demonstrate the efficiency of your implementations.
Provide a report comparing the performance of different list implementations, supported by empirical data (e.g., execution time).
Implementation Steps
Setup:
Create a new Java project.
Define a class ListOperations to encapsulate the list operations.

Implement Basic List Operations:
Implement the basic versions of the required operations for ArrayList and LinkedList.
Ensure proper handling of edge cases (e.g., adding/removing elements at the bounds of the list).

Implement CustomLinkedList:
Implement a singly-linked list from scratch with methods for adding, removing, getting, setting, and checking elements.
Analyze and Document Complexities:

Write comments or a separate documentation file explaining the time and space complexities of each method using Big O notation.

Compare List Implementations:
Benchmark the performance of ArrayList, LinkedList, and CustomLinkedList for various operations.
Identify the most efficient list implementation for different scenarios.

Performance Testing:
Write a test class ListOperationsTest to benchmark the performance of the different list implementations.
Use datasets with different sizes and characteristics to illustrate the efficiency gains of the optimized methods.
Report:

Compile a report that includes:
Descriptions of each list operation and its implementation.
Analysis of time and space complexities.
Comparison of different list implementations.
Performance data from benchmarking.

