

Exercise 1: Inventory Management System

1. Understand the Problem

Why Data Structures and Algorithms Are Essential:

- **Efficient Storage and Retrieval:** As inventory size grows, efficient data structures allow for quick access, modification, and deletion of product records.
- **Performance:** Algorithms help in optimizing operations like searching for a product or sorting products based on different criteria, which is crucial for managing large inventories.
- **Memory Management:** Choosing the right data structure ensures optimal use of memory, preventing waste and ensuring scalability.

Suitable Data Structures:

- **ArrayList:** Good for maintaining a list of products where order is important, and you need fast access by index.
- **HashMap:** Ideal for scenarios where quick lookup, addition, and deletion by key (productId) are required.
- **TreeMap:** Provides sorted order of products based on keys, useful if products need to be sorted by productId.

2. Setup

Create a new Java project in your IDE (e.g., IntelliJ IDEA, Eclipse).

3. Implementation

```
import java.util.HashMap;
```

```
import java.util.Map;
```

```
class Product {
```

```
    private String productId;
```

```
    private String productName;
```

```
    private int quantity;
```

```
    private double price;
```

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

```
        this.productId = productId;
```

```
        this.productName = productName;
```

```
    this.quantity = quantity;  
    this.price = price;  
}
```

```
public String getProductId() {  
    return productId;  
}
```

```
public void setProductId(String productId) {  
    this.productId = productId;  
}
```

```
public String getProductName() {  
    return productName;  
}
```

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

```
public int getQuantity() {  
    return quantity;  
}
```

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

```
public double getPrice() {
```

```
        return price;
    }

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

    @Override
    public String toString() {
        return "Product{" +
            "productId='" + productId + "\" +
            ", productName='" + productName + "\" +
            ", quantity=" + quantity +
            ", price=" + price +
            "'}";
    }
}

class Inventory {
    private Map<String, Product> productMap;

    public Inventory() {
        productMap = new HashMap<>();
    }

    public void addProduct(Product product) {
        productMap.put(product.getProductId(), product);
    }
}
```

```
public void updateProduct(String productId, Product updatedProduct) {  
    if (productMap.containsKey(productId)) {  
        productMap.put(productId, updatedProduct);  
    } else {  
        System.out.println("Product not found.");  
    }  
}
```

```
public void deleteProduct(String productId) {  
    if (productMap.containsKey(productId)) {  
        productMap.remove(productId);  
    } else {  
        System.out.println("Product not found.");  
    }  
}
```

```
public Product getProduct(String productId) {  
    return productMap.get(productId);  
}
```

```
public void listAllProducts() {  
    for (Product product : productMap.values()) {  
        System.out.println(product);  
    }  
}
```

```
public class InventoryManagementSystem {  
    public static void main(String[] args) {
```

- **Add Product:**

- **HashMap:** Average case $O(1)$, Worst case $O(n)$ due to possible hash collisions.
- **Update Product:**
 - **HashMap:** Average case $O(1)$, Worst case $O(n)$ due to hash collisions.
- **Delete Product:**
 - **HashMap:** Average case $O(1)$, Worst case $O(n)$ due to hash collisions.
- **Retrieve Product:**
 - **HashMap:** Average case $O(1)$, Worst case $O(n)$.

Optimizations:

- **Reduce Hash Collisions:** Use a good hash function to minimize the chance of collisions, which can degrade performance.
- **Rehashing:** Ensure the load factor of the HashMap is optimal to maintain performance during additions.
- **Concurrency:** For a concurrent environment, use ConcurrentHashMap to handle multiple threads.

Exercise 2: E-commerce Platform Search Function

1. Understand Asymptotic Notation

Big O Notation:

- **Definition:** Big O notation is used to describe the upper bound of an algorithm's running time, providing a way to express its performance in terms of input size.
- **Purpose:** It helps us understand how the runtime of an algorithm increases with the size of the input, allowing us to compare the efficiency of different algorithms.

Best, Average, and Worst-Case Scenarios:

- **Best Case:** The minimum time an algorithm takes to complete, typically when the desired element is found immediately.
- **Average Case:** The expected time an algorithm takes to complete, averaged over all possible inputs.
- **Worst Case:** The maximum time an algorithm takes to complete, usually when the desired element is not present, or the input is arranged unfavorably.

2. Setup

```
class Product {
    private String productId;
    private String productName;
```

```
private String category;
```

```
public Product(String productId, String productName, String category) {  
    this.productId = productId;  
    this.productName = productName;  
    this.category = category;  
}
```

```
public String getProductId() {  
    return productId;  
}
```

```
public String getProductName() {  
    return productName;  
}
```

```
public String getCategory() {  
    return category;  
}
```

```
@Override
```

```
public String toString() {  
    return "Product{" +  
        "productId='" + productId + "'" +  
        ", productName='" + productName + "'" +  
        ", category='" + category + "'" +  
        '}';  
}
```

```
}
```

3. Implementation

Linear Search Algorithm:

```
public class LinearSearch {  
  
    public static int linearSearch(Product[] products, String productName) {  
        for (int i = 0; i < products.length; i++) {  
            if (products[i].getProductName().equalsIgnoreCase(productName)) {  
                return i;  
            }  
        }  
        return -1; // Not found  
    }  
}
```

Binary Search Algorithm:

Binary search requires the array to be sorted. It works by repeatedly dividing the search interval in half and comparing the middle element with the target value.

```
import java.util.Arrays;  
  
public class BinarySearch {  
  
    public static int binarySearch(Product[] products, String productName) {  
        int left = 0;  
        int right = products.length - 1;  
  
        while (left <= right) {  
            int mid = left + (right - left) / 2;  
            int comparison = products[mid].getProductName().compareToIgnoreCase(productName);  
  
            if (comparison == 0) {
```



```

        return mid;

    } else if (comparison < 0) {

        left = mid + 1;

    } else {

        right = mid - 1;

    }

}

return -1; // Not found

}

}

```

4. Analysis

Time Complexity Comparison:

- **Linear Search:**
 - **Best Case:** $O(1)$ when the desired element is at the beginning of the array.
 - **Average Case:** $O(n)$ where n is the number of elements.
 - **Worst Case:** $O(n)$ when the desired element is at the end of the array or not present.
- **Binary Search:**
 - **Best Case:** $O(1)$ when the desired element is at the middle of the array.
 - **Average Case:** $O(\log n)$ where n is the number of elements.
 - **Worst Case:** $O(\log n)$ because the search space is halved with each step.

Screenshot-

The screenshot shows a Java IDE with a file named 'Main.java'. The code defines a 'Product' class with attributes 'productId', 'productName', and 'category', and methods to get these values. The output window shows the results of running the code, indicating that the product was found at index 1 for linear search and index 3 for binary search.

```

Main.java
1 import java.util.Arrays;
2
3 class Product {
4     private String productId;
5     private String productName;
6     private String category;
7
8     public Product(String productId, String productName, String category) {
9         this.productId = productId;
10        this.productName = productName;
11        this.category = category;
12    }
13
14    public String getProductId() {
15        return productId;
16    }
17
18    public String getProductName() {
19        return productName;
20    }
21
22    public String getCategory() {
23        return category;
24    }
25
26    @Override

```

```

Output
java -cp /tmp/gzq1GsPsY1/SearchAlgorithms
Linear Search: Product found at index 1
Binary Search: Product found at index 3

=== Code Execution Successful ===

```

Exercise 3: Sorting Customer Orders

1. Understand Sorting Algorithms

Bubble Sort:

- **Description:** Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted.
- **Time Complexity:**
 - Best Case: $O(n)$ (when the list is already sorted)
 - Average Case: $O(n^2)$
 - Worst Case: $O(n^2)$
- **Space Complexity:** $O(1)$ (in-place sorting)

Insertion Sort:

- **Description:** Insertion Sort builds the final sorted array one item at a time. It takes each element from the input and inserts it into its correct position in a sorted part of the array.
- **Time Complexity:**
 - Best Case: $O(n)$ (when the list is already sorted)
 - Average Case: $O(n^2)$
 - Worst Case: $O(n^2)$
- **Space Complexity:** $O(1)$ (in-place sorting)

Quick Sort:

- **Description:** Quick Sort is a divide-and-conquer algorithm. It picks an element as a pivot and partitions the array into two halves, with elements smaller than the pivot on one side and elements larger on the other. It then recursively sorts the partitions.
- **Time Complexity:**
 - Best Case: $O(n \log n)$
 - Average Case: $O(n \log n)$
 - Worst Case: $O(n^2)$ (when the smallest or largest element is always chosen as the pivot)
- **Space Complexity:** $O(\log n)$ (due to recursive stack space)

Merge Sort:

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

- **Time Complexity:**
 - Best Case: $O(n \log n)$
 - Average Case: $O(n \log n)$
 - Worst Case: $O(n \log n)$
- **Space Complexity:** $O(n)$ (due to temporary arrays used for merging)

2. Setup

```
class Order {  
    private String orderId;  
    private String customerName;  
    private double totalPrice;  
  
    public Order(String orderId, String customerName, double totalPrice) {  
        this.orderId = orderId;  
        this.customerName = customerName;  
        this.totalPrice = totalPrice;  
    }  
  
    public String getOrderId() {  
        return orderId;  
    }  
  
    public String getCustomerName() {  
        return customerName;  
    }  
  
    public double getTotalPrice() {  
        return totalPrice;  
    }  
}
```

@Override

```
public String toString() {  
    return "Order{" +  
        "orderId='" + orderId + '\'' +  
        ", customerName='" + customerName + '\'' +  
        ", totalPrice=" + totalPrice +  
        '}';  
}  
}
```

3. Implementation

```
public class BubbleSort {  
  
    public static void bubbleSort(Order[] orders) {  
        int n = orders.length;  
        boolean swapped;  
        for (int i = 0; i < n - 1; i++) {  
            swapped = false;  
            for (int j = 0; j < n - 1 - i; j++) {  
                if (orders[j].getTotalPrice() > orders[j + 1].getTotalPrice()) {  
                    // Swap orders[j] and orders[j + 1]  
                    Order temp = orders[j];  
                    orders[j] = orders[j + 1];  
                    orders[j + 1] = temp;  
                    swapped = true;  
                }  
            }  
            if (!swapped) break; // Stop if the array is already sorted  
        }  
    }  
}
```

```
}
```

Quick Sort Implementation:

```
public class QuickSort {
```

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

```
        if (low < high) {
```

```
            int pi = partition(orders, low, high);
```

```
            quickSort(orders, low, pi - 1);
```

```
            quickSort(orders, pi + 1, high);
```

```
        }
```

```
    }
```

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

```
        double pivot = orders[high].getTotalPrice();
```

```
        int i = low - 1;
```

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

```
            if (orders[j].getTotalPrice() <= pivot) {
```

```
                i++;
```

```
                // Swap orders[i] and orders[j]
```

```
                Order temp = orders[i];
```

```
                orders[i] = orders[j];
```

```
                orders[j] = temp;
```

```
            }
```

```
        }
```

```
        // Swap orders[i+1] and orders[high] (pivot)
```

```
        Order temp = orders[i + 1];
```

```
        orders[i + 1] = orders[high];
```

```
        orders[high] = temp;
```

```
        return i + 1;
    }
}
```

4. Analysis

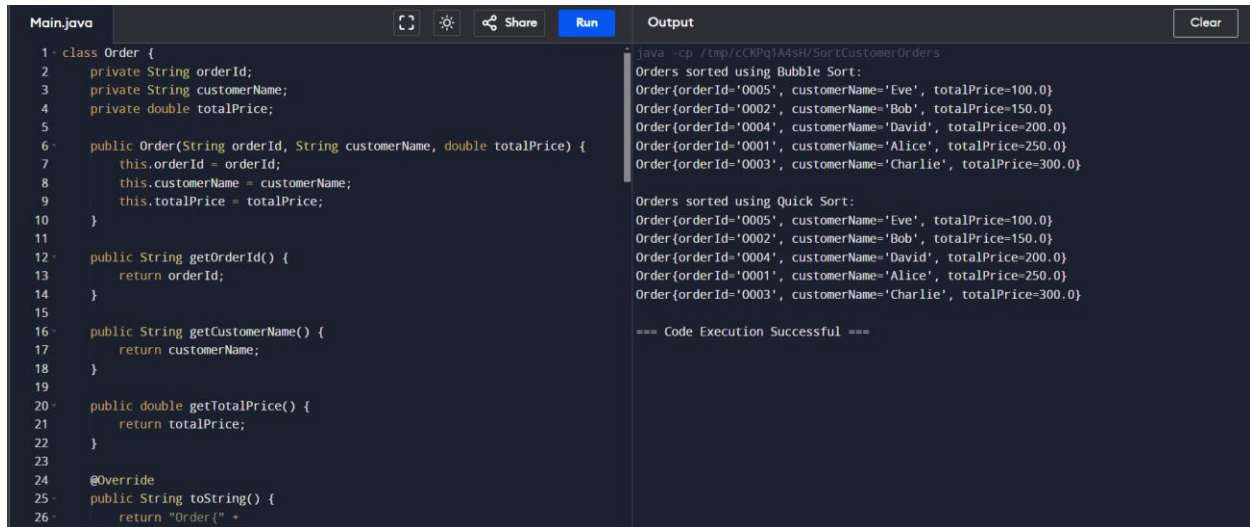
Performance Comparison:

- **Bubble Sort:**
 - Time Complexity: $O(n^2)$ for average and worst cases.
 - Space Complexity: $O(1)$ as it is an in-place sorting algorithm.
 - **Drawbacks:** Inefficient for large datasets due to its quadratic time complexity.
- **Quick Sort:**
 - Time Complexity: $O(n \log n)$ on average and best cases; $O(n^2)$ in the worst case.
 - Space Complexity: $O(\log n)$ due to the recursive calls.
 - **Advantages:** Faster than Bubble Sort for large datasets and is generally efficient due to its divide-and-conquer nature.

Why Quick Sort is Preferred:

- **Efficiency:** Quick Sort is generally more efficient than Bubble Sort due to its average-case time complexity of $O(n \log n)$, compared to Bubble Sort's $O(n^2)$.
- **Practical Performance:** Quick Sort has good cache performance and is often faster in practice, especially for large datasets.
- **Flexibility:** It can be implemented as an in-place sort with low additional memory overhead compared to Merge Sort.

Screenshot



The screenshot shows a Java IDE with a file named 'Main.java'. The code defines an 'Order' class with attributes 'orderId', 'customerName', and 'totalPrice'. It includes a constructor and three getter methods. The output window shows the results of running the code, displaying two sorted lists of 'Order' objects: one sorted using Bubble Sort and another using Quick Sort. Both lists show the same five objects in the same order. The output ends with '=== Code Execution Successful ==='.

```
1 class Order {
2     private String orderId;
3     private String customerName;
4     private double totalPrice;
5
6     public Order(String orderId, String customerName, double totalPrice) {
7         this.orderId = orderId;
8         this.customerName = customerName;
9         this.totalPrice = totalPrice;
10    }
11
12    public String getOrderId() {
13        return orderId;
14    }
15
16    public String getCustomerName() {
17        return customerName;
18    }
19
20    public double getTotalPrice() {
21        return totalPrice;
22    }
23
24    @Override
25    public String toString() {
26        return "Order{" +
```

```
java -cp /tmp/cCKPq1AdSH/SortCustomerOrders
Orders sorted using Bubble Sort:
Order{orderId='0005', customerName='Eve', totalPrice=100.0}
Order{orderId='0002', customerName='Bob', totalPrice=150.0}
Order{orderId='0004', customerName='David', totalPrice=200.0}
Order{orderId='0001', customerName='Alice', totalPrice=250.0}
Order{orderId='0003', customerName='Charlie', totalPrice=300.0}

Orders sorted using Quick Sort:
Order{orderId='0005', customerName='Eve', totalPrice=100.0}
Order{orderId='0002', customerName='Bob', totalPrice=150.0}
Order{orderId='0004', customerName='David', totalPrice=200.0}
Order{orderId='0001', customerName='Alice', totalPrice=250.0}
Order{orderId='0003', customerName='Charlie', totalPrice=300.0}

=== Code Execution Successful ===
```

Exercise 4: Employee Management System

1. Understand Array Representation

Arrays in Memory:

- **Definition:** An array is a collection of elements of the same data type, stored in contiguous memory locations. Each element can be accessed using an index, which is calculated based on the starting address of the array.
- **Memory Layout:** Arrays are stored sequentially in memory. This sequential storage allows for fast access to elements using indices. For an array with a starting address base and element size s , the address of the i -th element is given by $\text{base} + i * s$.
- **Advantages:**
 - **Fast Access:** Constant time complexity $O(1)$ for accessing elements using an index.
 - **Predictable Memory Usage:** Since the size is fixed, memory allocation is straightforward.
 - **Cache-Friendly:** Contiguous memory storage makes arrays cache-efficient.

Limitations:

- **Fixed Size:** Arrays have a fixed size, meaning you cannot add more elements than the predefined size.
- **Costly Insertions/Deletions:** Inserting or deleting elements can be costly, as it may require shifting elements.
- **Inefficient for Dynamic Data:** Not suitable for scenarios where the number of elements changes frequently.

Setup

Let's create an Employee class with the necessary attributes.

```
class Employee {  
    private String employeeId;  
    private String name;  
    private String position;  
    private double salary;  
  
    public Employee(String employeeId, String name, String position, double salary) {  
        this.employeeId = employeeId;  
        this.name = name;  
        this.position = position;  
        this.salary = salary;  
    }  
  
    public String getEmployeeId() {  
        return employeeId;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public String getPosition() {  
        return position;  
    }  
  
    public double getSalary() {  
        return salary;  
    }  
}
```


@Override

```
public String toString() {  
    return "Employee{" +  
        "employeeId='" + employeeId + "\" +  
        ", name='" + name + "\" +  
        ", position='" + position + "\" +  
        ", salary=" + salary +  
        "'";  
}
```

```
}
```

Implementation

```
public class EmployeeManagementSystem {  
    private Employee[] employees;  
    private int count;  
  
    public EmployeeManagementSystem(int capacity) {  
        employees = new Employee[capacity];  
        count = 0;  
    }
```

// Add an employee

```
public boolean addEmployee(Employee employee) {  
    if (count < employees.length) {  
        employees[count++] = employee;  
        return true;  
    }  
    System.out.println("Array is full, unable to add employee.");  
    return false;
```

```
}
```

```
// Search for an employee by ID
```

```
public Employee searchEmployee(String employeeId) {  
    for (int i = 0; i < count; i++) {  
        if (employees[i].getEmployeeId().equals(employeeId)) {  
            return employees[i];  
        }  
    }  
    return null;  
}
```

```
// Traverse and list all employees
```

```
public void listEmployees() {  
    for (int i = 0; i < count; i++) {  
        System.out.println(employees[i]);  
    }  
}
```

```
// Delete an employee by ID
```

```
public boolean deleteEmployee(String employeeId) {  
    for (int i = 0; i < count; i++) {  
        if (employees[i].getEmployeeId().equals(employeeId)) {  
            // Shift elements to the left  
            for (int j = i; j < count - 1; j++) {  
                employees[j] = employees[j + 1];  
            }  
            employees[--count] = null; // Nullify the last element  
            return true;  
        }  
    }  
}
```

```
    }  
}  
System.out.println("Employee not found.");  
return false;  
}  
  
public static void main(String[] args) {  
    EmployeeManagementSystem system = new EmployeeManagementSystem(10);  
  
    // Add employees  
    system.addEmployee(new Employee("E001", "Alice", "Manager", 75000));  
    system.addEmployee(new Employee("E002", "Bob", "Developer", 60000));  
    system.addEmployee(new Employee("E003", "Charlie", "Analyst", 55000));  
  
    // List all employees  
    System.out.println("All Employees:");  
    system.listEmployees();  
  
    // Search for an employee  
    Employee emp = system.searchEmployee("E002");  
    System.out.println("Searched Employee: " + emp);  
  
    // Delete an employee  
    system.deleteEmployee("E003");  
  
    // List all employees after deletion  
    System.out.println("Employees after deletion:");  
    system.listEmployees();  
}
```

}

4. Analysis

Time Complexity:

- **Add Operation:**
 - Time Complexity: $O(1)$ for adding at the end if there is space.
 - Limitation: Adding beyond the fixed size is not possible without resizing.
- **Search Operation:**
 - Time Complexity: $O(n)$ as it involves checking each element until the match is found.
- **Traverse Operation:**
 - Time Complexity: $O(n)$ as it requires visiting each element in the array.
- **Delete Operation:**
 - Time Complexity: $O(n)$ as it may require shifting elements to fill the gap left by the deleted element.

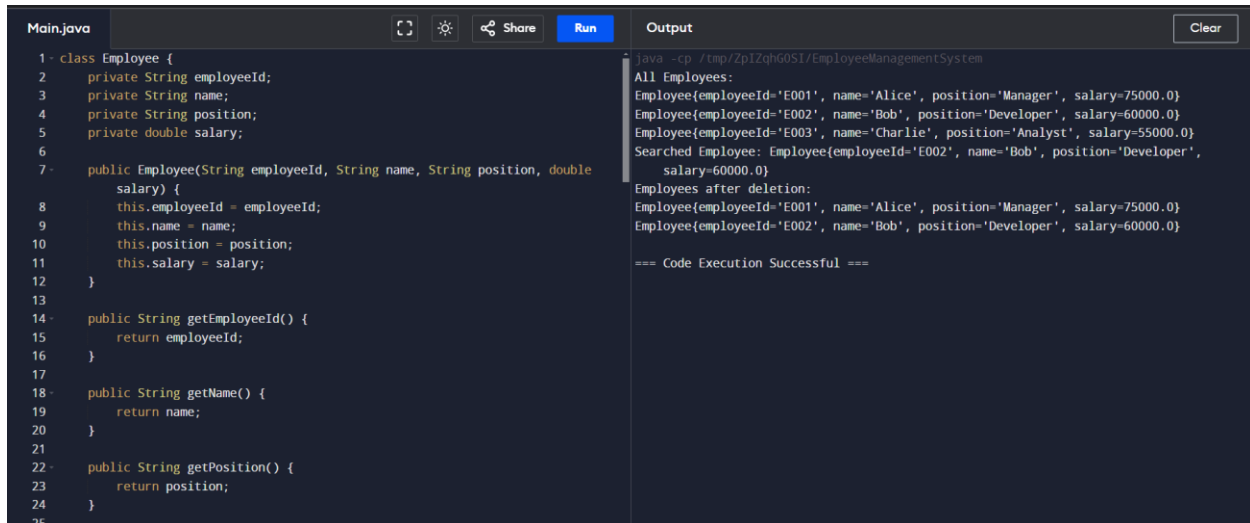
Limitations of Arrays and When to Use Them:

- **Static Size:** Arrays have a fixed size, so they are not suitable for scenarios where data grows dynamically.
- **Insertion and Deletion Overheads:** Inserting or deleting elements can be inefficient due to the need for shifting elements.
- **Use Cases:** Arrays are ideal when the number of elements is known in advance, and fast access to elements using indices is required.

For more dynamic scenarios, consider using other data structures like ArrayLists or LinkedLists, which provide more flexibility for resizing and modifying collections.

This code provides a basic implementation of an employee management system using arrays and highlights the advantages and limitations of using arrays for managing dynamic data.

Screenshot



The screenshot shows a Java IDE with a file named 'Main.java'. The code defines an 'Employee' class with attributes 'employeeId', 'name', 'position', and 'salary'. It includes a constructor and three getter methods. The 'Output' window shows the results of running the program, displaying a list of three employees, a search result for employee 'E002', and the state of the employee list after deleting employee 'E002'. The output concludes with '=== Code Execution Successful ==='.

```
1 class Employee {
2     private String employeeId;
3     private String name;
4     private String position;
5     private double salary;
6
7     public Employee(String employeeId, String name, String position, double
8         salary) {
9         this.employeeId = employeeId;
10        this.name = name;
11        this.position = position;
12        this.salary = salary;
13    }
14
15    public String getEmployeeId() {
16        return employeeId;
17    }
18
19    public String getName() {
20        return name;
21    }
22
23    public String getPosition() {
24        return position;
25    }
26 }
```

```
java -cp /tmp/Zp1Zqhg0S1/EmployeeManagementSystem
All Employees:
Employee{employeeId='E001', name='Alice', position='Manager', salary=75000.0}
Employee{employeeId='E002', name='Bob', position='Developer', salary=60000.0}
Employee{employeeId='E003', name='Charlie', position='Analyst', salary=55000.0}
Searched Employee: Employee{employeeId='E002', name='Bob', position='Developer',
    salary=60000.0}
Employees after deletion:
Employee{employeeId='E001', name='Alice', position='Manager', salary=75000.0}
Employee{employeeId='E002', name='Bob', position='Developer', salary=60000.0}

=== Code Execution Successful ===
```

Exercise 5: Task Management System

1. Understand Linked Lists

Singly Linked List:

- **Structure:** Consists of nodes where each node contains data and a pointer to the next node.
- **Traversal:** Begins from the head node and continues through each node following the next pointers.
- **Characteristics:**
 - Dynamic size, allowing efficient insertion and deletion.
 - Lack of backward traversal due to unidirectional links.

Doubly Linked List:

- **Structure:** Similar to a singly linked list but each node contains two pointers: one to the next node and one to the previous node.
- **Traversal:** Allows traversal in both directions (forward and backward).
- **Characteristics:**
 - Dynamic size, allowing efficient insertion and deletion.
 - Greater memory usage due to the additional pointer.
 - Supports operations that require backward traversal or quick access to previous nodes.

2. Setup

```
class Task {  
  
    private String taskId;
```

```
private String taskName;

private String status;

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

    this.taskId = taskId;

    this.taskName = taskName;

    this.status = status;

}

public String getTaskId() {

    return taskId;

}

public String getTaskName() {

    return taskName;

}

public String getStatus() {

    return status;

}

@Override

public String toString() {

    return "Task{" +

        "taskId=" + taskId + "\" +

        ", taskName=" + taskName + "\" +

        ", status=" + status + "\" +

        '}'

}

}
```

```
}
```

3. Implementation

```
// Node class for singly linked list
```

```
class Node {
```

```
    Task task;
```

```
    Node next;
```

```
    public Node(Task task) {
```

```
        this.task = task;
```

```
        this.next = null;
```

```
    }
```

```
}
```

```
public class TaskManagementSystem {
```

```
    private Node head;
```

```
    public TaskManagementSystem() {
```

```
        head = null;
```

```
    }
```

```
// Add a task to the linked list
```

```
public void addTask(Task task) {
```

```
    Node newNode = new Node(task);
```

```
    if (head == null) {
```

```
        head = newNode;
```

```
    } else {
```

```
        Node current = head;
```

```
        while (current.next != null) {
```

```
            current = current.next;
```

```

    }

    current.next = newNode;
}
}

// Search for a task by ID
public Task searchTask(String taskId) {
    Node current = head;
    while (current != null) {
        if (current.task.getTaskId().equals(taskId)) {
            return current.task;
        }
        current = current.next;
    }
    return null;
}

// Traverse and list all tasks
public void listTasks() {
    Node current = head;
    while (current != null) {
        System.out.println(current.task);
        current = current.next;
    }
}

// Delete a task by ID
public boolean deleteTask(String taskId) {
    if (head == null) {

```



```
        return false;
    }

    if (head.task.getTaskId().equals(taskId)) {
        head = head.next;
        return true;
    }

    Node current = head;
    while (current.next != null) {
        if (current.next.task.getTaskId().equals(taskId)) {
            current.next = current.next.next;
            return true;
        }
        current = current.next;
    }
    return false;
}

public static void main(String[] args) {
    TaskManagementSystem system = new TaskManagementSystem();

    // Add tasks
    system.addTask(new Task("T001", "Design Database", "Pending"));
    system.addTask(new Task("T002", "Implement API", "In Progress"));
    system.addTask(new Task("T003", "Test Application", "Pending"));

    // List all tasks
    System.out.println("All Tasks:");
```

```

system.listTasks();

// Search for a task
Task task = system.searchTask("T002");
System.out.println("Searched Task: " + task);

// Delete a task
system.deleteTask("T003");

// List all tasks after deletion
System.out.println("Tasks after deletion:");
system.listTasks();
}
}

```

4. Analysis

Time Complexity:

- **Add Operation:** $O(n)$ as it requires traversal to the end of the list to add a new task.
- **Search Operation:** $O(n)$ as it involves checking each node until the match is found.
- **Traverse Operation:** $O(n)$ as it requires visiting each node in the list.
- **Delete Operation:** $O(n)$ as it may require traversal to find the node to be deleted.

Advantages of Linked Lists over Arrays for Dynamic Data:

- **Dynamic Size:** Linked lists can grow and shrink dynamically, making them more flexible for managing data with unpredictable size changes.
- **Efficient Insertions/Deletions:** Inserting or deleting elements in linked lists is generally more efficient than in arrays, as it doesn't require shifting elements.
- **Memory Usage:** Linked lists use memory proportionate to the number of elements, whereas arrays may have unused space

Exercise 6: Library Management System

1. Understand Search Algorithms

Linear Search:

- **Algorithm:** Sequentially checks each element of the list until the target element is found or the list ends.
- **Use Case:** Suitable for unsorted or small data sets.
- **Characteristics:**
 - Simple and easy to implement.
 - Inefficient for large data sets ($O(n)$ time complexity).

Binary Search:

- **Algorithm:** Repeatedly divides a sorted list in half to locate the target element.
- **Use Case:** Efficient for sorted data sets.
- **Characteristics:**
 - Requires the list to be sorted.
 - Much faster than linear search for large data sets ($O(\log n)$ time complexity).

2. Setup

```
class Book {

    private String bookId;

    private String title;

    private String author;


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

        this.bookId = bookId;

        this.title = title;

        this.author = author;

    }


    public String getBookId() {

        return bookId;

    }


    public String getTitle() {
```

```
        return title;
    }
}
```

```
public String getAuthor() {
    return author;
}
```

```
@Override
public String toString() {
    return "Book{" +
        "bookId=\"" + bookId + "\" +
        ", title=\"" + title + "\" +
        ", author=\"" + author + "\" +
        '}';
}
}
```

```
}
```

3. Implementation

```
import java.util.Arrays;
```

```
import java.util.Comparator;
```

```
public class LibraryManagementSystem {
    private Book[] books;

    public LibraryManagementSystem(Book[] books) {
        this.books = books;
    }
}
```

```
// Linear search for books by title
```

```
public Book linearSearchByTitle(String title) {
```

```
for (Book book : books) {  
    if (book.getTitle().equalsIgnoreCase(title)) {  
        return book;  
    }  
}  
return null;  
}
```

// Binary search for books by title

```
public Book binarySearchByTitle(String title) {  
    // Ensure the list is sorted by title  
    Arrays.sort(books, Comparator.comparing(Book::getTitle));  
  
    int left = 0;  
    int right = books.length - 1;  
    while (left <= right) {  
        int mid = left + (right - left) / 2;  
        int comparison = books[mid].getTitle().compareToIgnoreCase(title);  
        if (comparison == 0) {  
            return books[mid];  
        } else if (comparison < 0) {  
            left = mid + 1;  
        } else {  
            right = mid - 1;  
        }  
    }  
    return null;  
}
```

```

public static void main(String[] args) {

    Book[] books = {

        new Book("B001", "Java Programming", "Alice Smith"),

        new Book("B002", "Python Essentials", "Bob Johnson"),

        new Book("B003", "Data Structures", "Charlie Brown"),

        new Book("B004", "Algorithms Unlocked", "David Wilson")

    };


    LibraryManagementSystem library = new LibraryManagementSystem(books);


    // Linear search

    String searchTitle = "Python Essentials";

    Book foundBookLinear = library.linearSearchByTitle(searchTitle);

    System.out.println("Linear Search Result: " + foundBookLinear);


    // Binary search

    Book foundBookBinary = library.binarySearchByTitle(searchTitle);

    System.out.println("Binary Search Result: " + foundBookBinary);

}
}

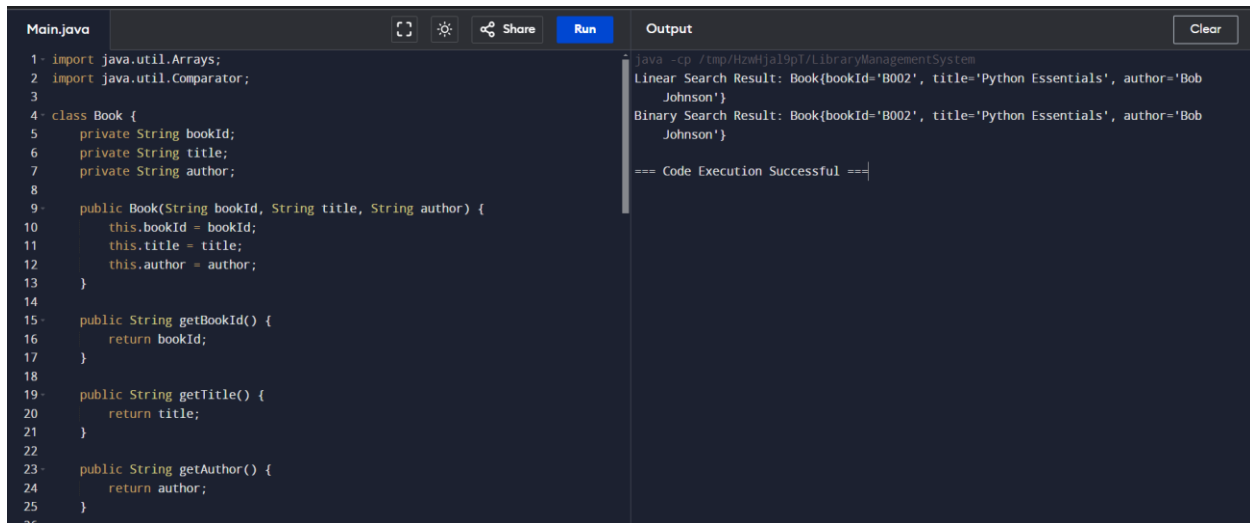
```

4. Analysis

Time Complexity:

- **Linear Search:** $O(n)$, where n is the number of books. It checks each book sequentially, making it less efficient for large data sets.
- **Binary Search:** $O(\log n)$, where n is the number of books. It divides the sorted list in half at each step, making it much faster for large data sets.

Screenshot

A screenshot of a Java IDE. The left pane shows a file named 'Main.java' with the following code:

```
1- import java.util.Arrays;
2- import java.util.Comparator;
3-
4- class Book {
5-     private String bookId;
6-     private String title;
7-     private String author;
8-
9-     public Book(String bookId, String title, String author) {
10-         this.bookId = bookId;
11-         this.title = title;
12-         this.author = author;
13-     }
14-
15-     public String getBookId() {
16-         return bookId;
17-     }
18-
19-     public String getTitle() {
20-         return title;
21-     }
22-
23-     public String getAuthor() {
24-         return author;
25-     }
26- }
```

The right pane shows the 'Output' window with the following text:

```
java -cp /tmp/HzwHja19pT/LibraryManagementSystem
Linear Search Result: Book(bookId='B002', title='Python Essentials', author='Bob
Johnson')
Binary Search Result: Book(bookId='B002', title='Python Essentials', author='Bob
Johnson')
=== Code Execution Successful ===
```

Exercise 7: Financial Forecasting

1. Understand Recursive Algorithms

Recursion:

- **Concept:** A recursive function calls itself to solve smaller instances of the same problem until a base case is reached.
- **Simplification:** Recursion can simplify complex problems by breaking them down into more manageable subproblems.

Advantages of Recursion:

- Elegant and concise code.
- Natural fit for problems that can be divided into similar subproblems (e.g., factorial calculation, Fibonacci sequence).

Disadvantages of Recursion:

- May lead to high memory usage due to call stack growth.
- Can be less efficient for certain problems if not optimized.

2. Setup

Let's create a method to calculate future values based on a given annual growth rate and the number of years. We'll use recursion to compute the future value.

3. Implementation

```
public class FinancialForecastingTool {
```

```
    // Recursive method to calculate future value
```

```

public static double calculateFutureValue(double presentValue, double growthRate, int years) {
    // Base case: no more years to predict
    if (years == 0) {
        return presentValue;
    }
    // Recursive case: calculate the future value for the next year
    return calculateFutureValue(presentValue * (1 + growthRate), growthRate, years - 1);
}

```

```

public static void main(String[] args) {
    double presentValue = 10000; // Initial amount
    double annualGrowthRate = 0.05; // 5% growth rate
    int years = 5; // Number of years to predict

    double futureValue = calculateFutureValue(presentValue, annualGrowthRate, years);
    System.out.println("Future Value after " + years + " years: $" + futureValue);
}
}

```

4. Analysis

Time Complexity:

- **Recursive Algorithm:** $O(n)$, where n is the number of years. The algorithm makes one recursive call per year, leading to linear complexity.
- **Space Complexity:** $O(n)$ due to the call stack growth in the recursion.

Screenshot

Main.java	Output
<pre>1- public class FinancialForecastingTool { 2 3 // Iterative method to calculate future value 4- public static double calculateFutureValueIterative(double presentValue, 5 double growthRate, int years) { 6 double futureValue = presentValue; 7 for (int i = 0; i < years; i++) { 8 futureValue *= (1 + growthRate); 9 } 10 return futureValue; 11 } 12- public static void main(String[] args) { 13 double presentValue = 10000; // Initial amount 14 double annualGrowthRate = 0.05; // 5% growth rate 15 int years = 5; // Number of years to predict 16 17 double futureValueIterative = calculateFutureValueIterative(presentValue, 18 annualGrowthRate, years); 19 System.out.println("Future Value after " + years + " years (Iterative): 20 \$" + futureValueIterative); 21 } 22 }</pre>	<pre>java -cp /tmp/FNOKB1wCuG/FinancialForecastingTool Future Value after 5 years (Iterative): \$12762.815625000001 === Code Execution Successful ===</pre>