**EXERCISES 1 TO 7**

**Exercise 1: Inventory Management System**

**Importance of Data Structures and Algorithms in Handling Large Inventories:**

1. **Efficient Storage and Retrieval**:
   * **Data Structures**: Use of arrays, hash tables, and trees for quick data access.
   * **Algorithms**: Search algorithms like binary search for fast retrieval.
2. **Optimal Resource Management**:
   * **Data Structures**: Priority queues and heaps for managing inventory priorities.
   * **Algorithms**: Sorting algorithms for organizing items by criteria like expiry dates.
3. **Scalability**:
   * **Data Structures**: Balanced trees (e.g., B-trees) maintain efficiency as data grows.
   * **Algorithms**: Efficient sorting and searching algorithms ensure system responsiveness.
4. **Data Integrity and Consistency**:
   * **Data Structures**: Sets and other structures prevent duplicates and maintain consistency.
   * **Algorithms**: Consistency checks to ensure accurate data.
5. **Real-Time Processing**:
   * **Data Structures**: Queues manage real-time data updates.
   * **Algorithms**: Fast processing algorithms handle real-time inventory tracking and alerts.
6. **Cost Efficiency**:
   * **Data Structures**: Space-efficient structures reduce storage costs.
   * **Algorithms**: Optimized algorithms minimize computational resources, saving costs.

**Types of Data Structures suitable for this problem are:**

1. **Arrays and Lists**: Simple storage for small inventories.
2. **Linked Lists**: Efficient for dynamic data with frequent insertions/deletions.
3. **Hash Tables**: Fast access using unique keys like SKU numbers.
4. **Binary Search Trees (BST)**: Sorted data storage with fast search.
5. **AVL Trees**: Self-balancing trees for efficient operations.
6. **B-trees and B+ trees**: Efficient for large-scale data storage and retrieval.
7. **Tries**: Fast prefix-based searching for strings.
8. **Heaps (Priority Queues)**: Manages items by priority, like expiration dates.
9. **Graphs**: Represents complex relationships and optimizes paths.
10. **Stacks and Queues**: Manages operations in LIFO or FIFO order.

**CODE IMPLEMENTATION:**

import java.util.HashMap;

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 getQuanity()

    {

        return quantity;

    }

    public void setQuantity(int quantity)

    {

        this.quantity=quantity;

    }

    public double getPrice()

    {

        return price;

    }

    public void setPrice(double price)

    {

        this.price=price;

    }

    public String toString(){

        return "Product {"+

        "product Id=" +productId+", "+"product Name=" +productName + ", "+"Quantity="+quantity+", "+"Price="+price+" }";

    }

}

class Inventory{

    private HashMap<String, Product> productInventory;

    public Inventory(){

        productInventory=new HashMap<>();

    }

    public void addProduct(Product product)

    {

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

        System.out.println("Product added: "+product);

    }

    public void updateProduct(String productId,Product updatedProduct){

        if (productInventory.containsKey(productId)) {

            productInventory.put(productId, updatedProduct);

            System.out.println("Product updated: " + updatedProduct);

        } else {

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

        }

    }

    public void deleteProduct(String productId) {

        Product removedProduct = productInventory.remove(productId);

        if (removedProduct != null) {

            System.out.println("Product removed: " + removedProduct);

        } else {

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

        }

    }

public void displayProducts() {

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

            System.out.println(product);

        }

    }

}

public class IMS{

    public static void main(String args[])

    {

        Inventory inv=new Inventory();

        Product p1=new Product("P001","Laptop",10,999.99);

        Product p2=new Product("P002","Mobile",20,599.99);

        System.out.println();

        inv.addProduct(p1);

        inv.addProduct(p2);

        System.out.println();

        inv.displayProducts();

        Product update=new Product("P002","Mobile",15,579.99);

        inv.updateProduct("P002",update);

        System.out.println();

        inv.displayProducts();

        inv.deleteProduct("P001");

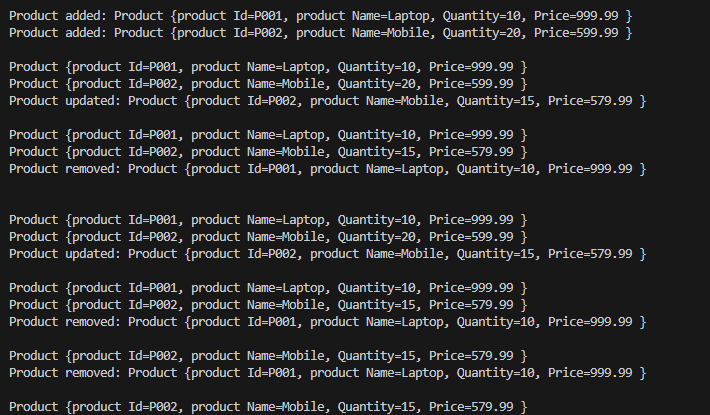
        System.out.println();

        inv.displayProducts();

    }

}

**Output:**

****

**Time Complexity:**

1. **Add Operation**: O(1) average case - Involves hashing and inserting the product.
2. **Update Operation**: O(1) average case - Access and replace the product by productId.
3. **Delete Operation**: O(1) average case - Find and remove the product using productId.

**Optimization Strategies:**

1. **Handling Collisions**: Use a good hash function and balanced trees within buckets to minimize worst-case complexity.
2. **Resizing**: Efficiently resize the HashMap when the load factor threshold is exceeded, choosing an appropriate initial capacity.
3. **Memory Management**: Optimize initial capacity and load factor, and manage unused entries to conserve memory.
4. **Custom Hash Function**: Tailor the hash function to your data for better key distribution. etc..

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

**Big O Notation:** Big O notation is a mathematical representation used to describe the upper bound of an algorithm's time complexity, i.e., how the runtime grows with the input size. It provides a way to classify algorithms according to their performance and efficiency.

* **Best-case scenario**: The minimum time taken by the algorithm for the given input size.
* **Average-case scenario**: The expected time taken by the algorithm for a typical input.
* **Worst-case scenario**: The maximum time taken by the algorithm, ensuring performance is understood even in the worst conditions.

**Time Complexities of Search Operations:**

**Linear Search**:

* Best-case: O(1) - The item is found at the first position.
* Average-case: O(n) - The item is found in the middle of the array.
* Worst-case: O(n) - The item is found at the last position or not present.

**Binary Search** (only applicable to sorted arrays):

* Best-case: O(1) - The item is found at the middle position.
* Average-case: O(log n) - The item is found somewhere in the array, requiring log n comparisons.
* Worst-case: O(log n) - The item is not found, requiring log n comparisons.

**CODE IMPLEMENTATION:**

import java.util.Arrays;

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 + '\'' +

               '}';

    }

}

public class Exercise2 {

    public static void main(String[] args) {

        Product[] products = {

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

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

            new Product("P003", "Book", "Literature"),

            new Product("P004", "Headphones", "Accessories"),

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

        };

//Linaer Search

        Product foundProduct = linearSearch(products, "P003");

        if (foundProduct != null) {

            System.out.println("Linear Search: Found " + foundProduct);

        } else {

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

        }

        Arrays.sort(products, (p1, p2) -> p1.getProductId().compareTo(p2.getProductId()));

//Binary Search

        foundProduct = binarySearch(products, "P002");

        if (foundProduct != null) {

            System.out.println("Binary Search: Found " + foundProduct);

        } else {

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

        }

    }

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

        for (Product product : products) {

            if (product.getProductId().equals(targetId)) {

                return product;

            }

        }

        return null;

    }

    public static Product binarySearch(Product[] products, String targetId) {

        int left = 0;

        int right = products.length - 1;

        while (left <= right) {

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

            Product midProduct = products[mid];

            int comparison = midProduct.getProductId().compareTo(targetId);

            if (comparison == 0) {

                return midProduct;

            } else if (comparison < 0) {

                left = mid + 1;

            } else {

                right = mid - 1;

            }

        }

        return null;

    }

}

**OUTPUT:**

****

**Analysis:**

**Time Complexity Comparison:**

* **Linear Search**:
  + Time Complexity: O(n)
  + Suitable for unsorted data or when the list is relatively small.
* **Binary Search**:
  + Time Complexity: O(log n)
  + Requires the data to be sorted but provides significantly faster search times for large datasets.

For an e-commerce platform, where the dataset of products can be large and efficient searching is crucial, **binary search** is generally more suitable. This is because it offers logarithmic time complexity (O(log n)), which scales much better with large numbers of products compared to the linear time complexity (O(n)) of a linear search.

**Exercise 3: Sorting Customer Orders**

**Sorting Algorithms:**

**Bubble Sort:**

* **Concept**: Repeatedly compares adjacent elements and swaps them if they're in the wrong order.
* **Time Complexity**: O(n²) in average and worst cases.
* **Best Use**: Simple to implement but inefficient for large datasets.

**Insertion Sort:**

* **Concept**: Builds a sorted array one element at a time by comparing each new element to the existing sorted ones.
* **Time Complexity**: O(n²) in average and worst cases.
* **Best Use**: Efficient for small or nearly sorted datasets.

**Quick Sort:**

* **Concept**: Uses a pivot to partition the array into smaller and larger elements, then recursively sorts the partitions.
* **Time Complexity**: O(n log n) on average, O(n²) in the worst case.
* **Best Use**: Efficient for large datasets, widely used due to its average-case efficiency.

**Merge Sort:**

* **Concept**: Splits the array into halves, recursively sorts each half, and then merges the sorted halves.
* **Time Complexity**: O(n log n) in all cases.
* **Best Use**: Consistent performance, but requires extra space for merging.

CODE IMPLEMENTATION:

import java.util.Arrays;

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 +

               '}';

    }

}

public class Exercise3 {

    public static void bubbleSort(Order[] orders) {

        int n = orders.length;

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

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

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

                    Order temp = orders[j];

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

                    orders[j + 1] = temp;

                }

            }

        }

    }

    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++;

                Order temp = orders[i];

                orders[i] = orders[j];

                orders[j] = temp;

            }

        }

        Order temp = orders[i + 1];

        orders[i + 1] = orders[high];

        orders[high] = temp;

        return i + 1;

    }

    public static void main(String[] args) {

        Order[] orders = {

            new Order("O001", "Alice", 250.00),

            new Order("O002", "Bob", 150.00),

            new Order("O003", "Charlie", 350.00),

            new Order("O004", "Diana", 200.00)

        };

        System.out.println("Orders before sorting:");

        for (Order order : orders) {

            System.out.println(order);

        }

        bubbleSort(orders);

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

        for (Order order : orders) {

            System.out.println(order);

        }

        orders = new Order[]{

            new Order("O001", "Alice", 250.00),

            new Order("O002", "Bob", 150.00),

            new Order("O003", "Charlie", 350.00),

            new Order("O004", "Diana", 200.00)

        };

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

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

        for (Order order : orders) {

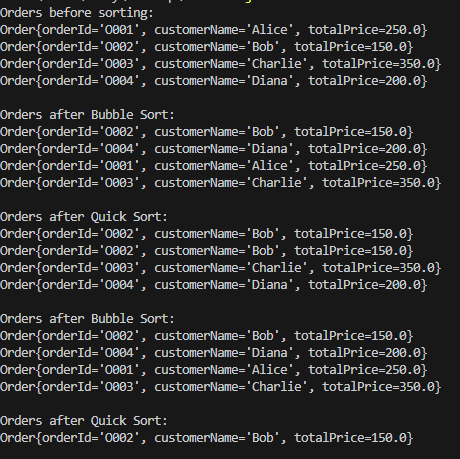
            System.out.println(order);

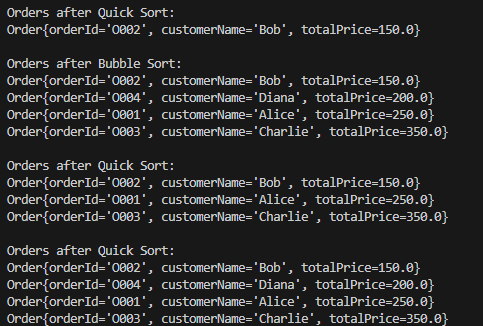
        }

    }

}

OUTPUT:





**Quick Sort vs. Bubble Sort:**

**Bubble Sort:**

* **Time Complexity**: O(n²) average and worst case.
* **Best Use**: Simple but inefficient for large datasets.

**Quick Sort:**

* **Time Complexity**: O(n log n) average case, O(n²) worst case.
* **Best Use**: Efficient for large datasets; generally faster due to better average-case performance.

**Quick sort is preferred because of the..**

 **Efficiency**: Quick Sort's O(n log n) average-case time complexity makes it faster for large datasets.

 **Scalability**: Handles larger datasets better than Bubble Sort.

 **Practicality**: Commonly used in practice; more efficient with memory and often faster on modern hardware.

**Exercise 4: Employee Management System**

**Array Representation in Memory**:

* Arrays are a contiguous block of memory where each element is of the same data type.
* The memory address of each element is calculated using the base address and the size of the data type, making access efficient.
* **Advantages**:
  + **Fast Access**: Constant time O(1) access to elements using indices.
  + **Efficient Memory Usage**: Fixed-size allocation ensures no memory overhead.

**CODE IMPLEMENTATION:**

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 +

               '}';

    }

}

public class Exercise4 {

    private Employee[] employees;

    private int count;

    public Exercise4(int capacity) {

        employees = new Employee[capacity];

        count = 0;

    }

    public boolean addEmployee(Employee employee) {

        if (count < employees.length) {

            employees[count++] = employee;

            return true;

        }

        return false;

    }

    public Employee searchEmployee(String employeeId) {

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

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

                return employees[i];

            }

        }

        return null;

    }

    public void traverseEmployees() {

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

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

        }

    }

    public boolean deleteEmployee(String employeeId) {

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

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

                employees[i] = employees[--count];

                employees[count] = null;

                return true;

            }

        }

        return false;

    }

    public static void main(String[] args) {

        Exercise4 ems = new Exercise4(10);

        ems.addEmployee(new Employee("E001", "Alice", "Manager", 70000));

        ems.addEmployee(new Employee("E002", "Bob", "Developer", 50000));

        ems.addEmployee(new Employee("E003", "Charlie", "Designer", 45000));

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

        ems.traverseEmployees();

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

        System.out.println(ems.searchEmployee("E002"));

        System.out.println("\nDelete Employee E003:");

        ems.deleteEmployee("E003");

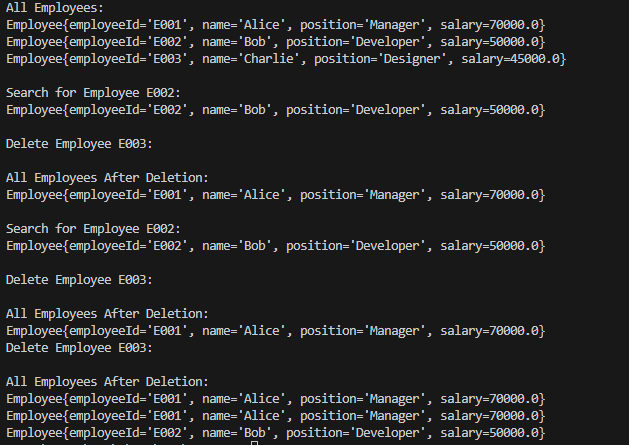
        System.out.println("\nAll Employees After Deletion:");

        ems.traverseEmployees();

    }

}

**OUTPUT:**

****

**Analysis:**

**Time Complexity**:

* **Add**: O(1) - Efficient, constant time if space is available.
* **Search**: O(n) - Linear search required, making it slower for large datasets.
* **Traverse**: O(n) - Requires visiting each element.
* **Delete**: O(n) - Finding and removing an element involves shifting, which can be time-consuming.

**Limitations of Arrays**:

* **Fixed Size**: Cannot change the size once allocated, leading to potential wasted or insufficient space.
* **Inefficient Insertions/Deletions**: Shifting elements for insertion or deletion results in O(n) time complexity.
* **Contiguous Memory Requirement**: Needs a contiguous block of memory, which can be problematic for large datasets.

**When to Use Arrays**:

* Suitable when the number of elements is fixed and known, and when fast access to elements is needed. For dynamic sizes or frequent updates, consider more flexible data structures like linked lists or dynamic arrays (ArrayList in Java).

**Exercise 5: Task Management System**

**Types of Linked Lists**:

* **Singly Linked List**:
  + Contains nodes with data and a reference to the next node.
  + Allows traversal in one direction (forward).
  + Memory-efficient but slower for operations requiring backward traversal or random access.
* **Doubly Linked List**:
  + Contains nodes with data, a reference to the next node, and a reference to the previous node.
  + Allows traversal in both directions (forward and backward).
  + Requires more memory due to additional pointers but provides faster insertion and deletion at both ends.

**CODE IMPLEMENTATION:**

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 + '\'' +

               '}';

    }

}

class TaskNode {

    Task task;

    TaskNode next;

    public TaskNode(Task task) {

        this.task = task;

        this.next = null;

    }

}

public class Exercise5 {

    private TaskNode head;

    public Exercise5() {

        this.head = null;

    }

    public void addTask(Task task) {

        TaskNode newNode = new TaskNode(task);

        if (head == null) {

            head = newNode;

        } else {

            TaskNode current = head;

            while (current.next != null) {

                current = current.next;

            }

            current.next = newNode;

        }

    }

    public Task searchTask(String taskId) {

        TaskNode current = head;

        while (current != null) {

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

                return current.task;

            }

            current = current.next;

        }

        return null;

    }

    public void traverseTasks() {

        TaskNode current = head;

        while (current != null) {

            System.out.println(current.task);

            current = current.next;

        }

    }

    public boolean deleteTask(String taskId) {

        if (head == null) {

            return false;

        }

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

            head = head.next;

            return true;

        }

        TaskNode current = head;

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

            current = current.next;

        }

        if (current.next == null) {

            return false;

        }

        current.next = current.next.next;

        return true;

    }

    public static void main(String[] args) {

        Exercise5 tms = new Exercise5();

        tms.addTask(new Task("T001", "Design UI", "Pending"));

        tms.addTask(new Task("T002", "Develop Backend", "In Progress"));

        tms.addTask(new Task("T003", "Testing", "Pending"));

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

        tms.traverseTasks();

        System.out.println("\nSearch for Task T002:");

        System.out.println(tms.searchTask("T002"));

        System.out.println("\nDelete Task T003:");

        tms.deleteTask("T003");

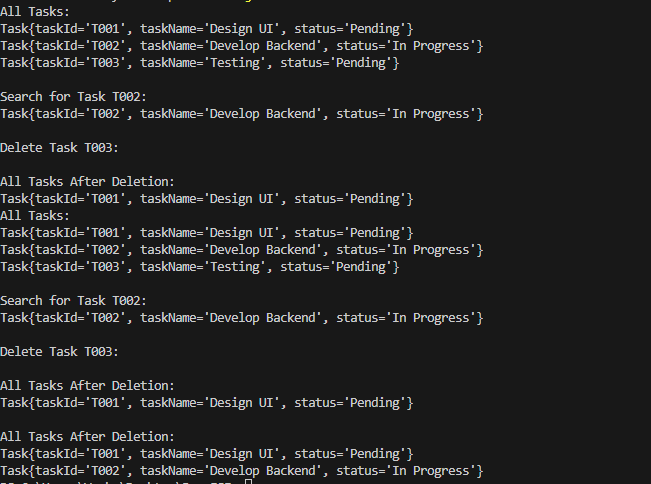
        System.out.println("\nAll Tasks After Deletion:");

        tms.traverseTasks();

    }

}

**OUTPUT:**

****

**Analysis**

**Time Complexity**:

* **Add**: O(1) for adding at the beginning, O(n) for adding at the end (traverse to the last node).
* **Search**: O(n) - Linear search through the list.
* **Traverse**: O(n) - Visits each node.
* **Delete**: O(n) - Finding the node to delete involves linear search.

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

* **Dynamic Size**: Linked lists can easily grow and shrink, unlike arrays which have a fixed size.
* **Efficient Insertions/Deletions**: Insertions and deletions in linked lists do not require shifting elements, making these operations more efficient, especially for large datasets.
* **Memory Utilization**: Linked lists do not require a contiguous block of memory, allowing for better utilization of memory in cases where memory is fragmented.

**Exercise 6: Library Management System**

**Search Algorithms:**

**Linear Search**:

* **Description**: A straightforward algorithm that checks each element in a list sequentially until the desired element is found or the list ends.
* **Time Complexity**: O(n) - It potentially requires checking every element, making it inefficient for large datasets.

**Binary Search**:

* **Description**: A more efficient algorithm that repeatedly divides a sorted list in half, comparing the target value to the middle element, and narrowing down the search interval.
* **Time Complexity**: O(log n) - Efficient for large, sorted datasets, as it reduces the number of comparisons significantly compared to linear search.

**CODE IMPLEMENTATION:**

import java.util.Arrays;

import java.util.Comparator;

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 + '\'' +

               '}';

    }

}

public class Exercise6 {

    private Book[] books;

    public Exercise6(Book[] books) {

        this.books = books;

    }

    public Book linearSearchByTitle(String title) {

        for (Book book : books) {

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

                return book;

            }

        }

        return null;

    }

    public Book binarySearchByTitle(String title) {

        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", "The Great Gatsby", "F. Scott Fitzgerald"),

            new Book("B002", "1984", "George Orwell"),

            new Book("B003", "To Kill a Mockingbird", "Harper Lee"),

            new Book("B004", "The Catcher in the Rye", "J.D. Salinger")

        };

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

        Exercise6 lms = new Exercise6(books);

        System.out.println("Search for '1984' using Linear Search:");

        System.out.println(lms.linearSearchByTitle("1984"));

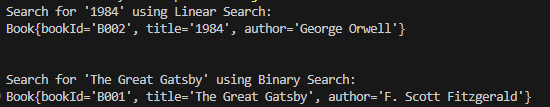
        System.out.println("\nSearch for 'The Great Gatsby' using Binary Search:");

        System.out.println(lms.binarySearchByTitle("The Great Gatsby"));

    }

}

**OUTPUT:**



**Analysis:**

**Time Complexity Comparison**:

* **Linear Search**: O(n)
  + Useful for unsorted or small datasets where setting up sorting isn't practical.
  + Simple and does not require data to be sorted.
* **Binary Search**: O(log n)
  + Much faster for large datasets but requires the list to be sorted.
  + More efficient as it quickly narrows down the search range.

**When to Use Each Algorithm**:

* **Linear Search**:
  + Use when the dataset is small or unsorted.
  + Useful if the list is frequently updated (insertion or deletion), as sorting would require additional overhead.
* **Binary Search**:
  + Preferable for large, sorted datasets where search speed is critical.
  + Useful when the cost of sorting the data initially is offset by the faster search times for multiple queries.

**Exercise 7: Financial Forecasting**

**Concept of Recursion**:

* **Recursion** is a technique where a function calls itself to solve smaller instances of the same problem.
* It can simplify problems by breaking them down into base cases (simple cases that can be solved directly) and recursive cases (more complex cases that are solved by calling the function on smaller instances).
* Example: Calculating the factorial of a number, Fibonacci sequence, etc.

**Method to Calculate Future Value Using Recursion**:

* The future value calculation often involves applying a growth rate to a current value over a number of periods. This can be represented recursively by applying the growth rate to the value from the previous period.

**CODE IMPLEMENTATION:**

public class Exercise7 {

    public static double Exercise7(double currentValue, double growthRate, int periods) {

        if (periods == 0) {

            return currentValue;

        }

        return Exercise7(currentValue \* (1 + growthRate), growthRate, periods - 1);

    }

    public static void main(String[] args) {

        double currentValue = 1000.0;

        double growthRate = 0.05;

        int periods = 10;

        double futureValueRecursive = Exercise7(currentValue, growthRate, periods);

        System.out.println("Future Value (Recursive): " + futureValueRecursive);

    }

}

**OUTPUT:**

****

**ANALYSIS:**

**Time Complexity**:

* The time complexity of this recursive algorithm is O(n), where n is the number of periods. This is because the function is called once for each period until the base case is reached.

**Optimizing the Recursive Solution**:

* **Avoiding Excessive Computation**: While the above recursive solution is straightforward, it can be inefficient if not managed properly, especially with a high number of periods or complex calculations.
* **Memoization**: One optimization technique is memoization, where results of expensive function calls are cached, and reused if the same inputs occur again, reducing the number of recursive calls.
* **Iterative Approach**: Another way to optimize is by using an iterative approach, which can reduce the overhead of function calls in recursion.