**Exercise 1: Inventory Management System**

**Scenario:**

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

**Steps:**

1. **Understand the Problem:**
   1. Explain why data structures and algorithms are essential in handling large inventories.
   2. Discuss the types of data structures suitable for this problem.
2. **Setup:**
   1. Create a new project for the inventory management system.
3. **Implementation:**
   1. Define a class Product with attributes like **productId**, **productName**, **quantity**, and **price**.
   2. Choose an appropriate data structure to store the products (e.g., ArrayList, HashMap).
   3. Implement methods to add, update, and delete products from the inventory.
4. **Analysis:**
   1. Analyze the time complexity of each operation (add, update, delete) in your chosen data structure.
   2. Discuss how you can optimize these operations.

**Solution :**

### **Why are Data Structures and Algorithms Important?**

In a warehouse, we may have thousands of products. If our system doesn’t manage and search through data quickly and efficiently, it will become slow, and users will get frustrated.

That’s why we use proper data structures and algorithms:

* They help in storing data neatly.
* They allow fast access, search, updates, and deletions.
* They make the system scalable, i.e., it works even when data grows big.

### **Suitable Data Structures:**

|  |  |
| --- | --- |
| **Data Structure** | **Why It's Useful** |
| **ArrayList** | Easy to store a list of products and maintain order |
| **HashMap** | Very fast to find products using unique keys like productId |
| **TreeMap** | Sorted map; keeps products ordered by keys |

For this system, HashMap is best because each product has a unique productId, and we want fast lookup, insert, and delete.

**Code:**

import java.util.\*;

class Product {

int productId;

String productName;

int quantity;

double price;

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

this.productId = productId;

this.productName = productName;

this.quantity = quantity;

this.price = price;

}

@Override

public String toString() {

return "Product ID: " + productId + ", Name: " + productName +

", Quantity: " + quantity + ", Price: ₹" + price;

}

}

class InventoryManager {

HashMap<Integer, Product> inventory = new HashMap<>();

public void addProduct(Product p) {

inventory.put(p.productId, p);

System.out.println("Product added successfully.");

}

public void updateProduct(int id, int newQuantity, double newPrice) {

if (inventory.containsKey(id)) {

Product p = inventory.get(id);

p.quantity = newQuantity;

p.price = newPrice;

System.out.println("Product updated.");

} else {

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

}

}

public void deleteProduct(int id) {

if (inventory.containsKey(id)) {

inventory.remove(id);

System.out.println("Product deleted.");

} else {

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

}

}

public void displayInventory() {

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

System.out.println(p);

}

}

}

public class Main {

public static void main(String[] args) {

InventoryManager im = new InventoryManager();

im.addProduct(new Product(101, "Laptop", 10, 55000));

im.addProduct(new Product(102, "Mouse", 50, 499));

im.updateProduct(101, 12, 54000);

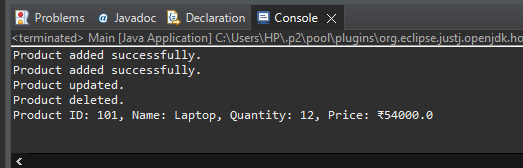
im.deleteProduct(102);

im.displayInventory();

}

}

**Output:**



### **Time Complexity of Each Operation in HashMap:**

|  |  |  |
| --- | --- | --- |
| **Operation** | **Time Complexity** | **Explanation** |
| addProduct() | O(1) (Average) | Directly adds using key |
| updateProduct() | O(1) (Average) | Gets the product by ID, updates values |
| deleteProduct() | O(1) (Average) | Removes by key instantly |
| displayInventory() | O(n) | Goes through all values once |

**Note:** In the worst-case scenario, due to hash collisions, these can go up to O(n), but in practice, Java handles it well with internal optimizations.

### **Optimization**

* Use a HashMap for constant-time access.
* If we need the products sorted (like by name or price), we can use a TreeMap or sort the values manually.
* If frequent searching by other fields (like productName) is needed, consider using additional HashMaps keyed by those fields.

**Exercise 3: Sorting Customer Orders**

**Scenario:**

You are tasked with sorting customer orders by their total price on an e-commerce platform. This helps in prioritizing high-value orders.

**Steps:**

1. **Understand Sorting Algorithms:**
   1. Explain different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort).
2. **Setup:**
   1. Create a class **Order** with attributes like **orderId**, **customerName**, and **totalPrice**.
3. **Implementation:**
   1. Implement **Bubble Sort** to sort orders by **totalPrice**.
   2. Implement **Quick Sort** to sort orders by **totalPrice**.
4. **Analysis:**
   1. Compare the performance (time complexity) of Bubble Sort and Quick Sort.
   2. Discuss why Quick Sort is generally preferred over Bubble Sort.

**Solution:**

## **1. Bubble Sort**

**Definition:**  
 Bubble Sort is a simple sorting algorithm that compares adjacent elements in a list and repeatedly swaps them if they are in the wrong order. The process is repeated until no further swaps are needed, indicating that the list is sorted.

**Working Principle:**  
 In each pass through the array, the largest unsorted element "bubbles up" to its correct position at the end. This continues until the entire list is sorted.

**Step-by-Step Example:**  
 Given an array: [5, 1, 4, 2, 8]

* First Pass:  
   Compare 5 and 1 → Swap → [1, 5, 4, 2, 8]  
   Compare 5 and 4 → Swap → [1, 4, 5, 2, 8]  
   Compare 5 and 2 → Swap → [1, 4, 2, 5, 8]  
   Compare 5 and 8 → No Swap
* Repeat this process until no swaps are required.

**Time Complexity:**

* Best Case: O(n) — when the array is already sorted.
* Average Case: O(n²)
* Worst Case: O(n²)

**Space Complexity:**  
 O(1) — in-place sorting.

**Use Case:**  
 Generally used for educational purposes and small data sets due to its simplicity.

## **2. Insertion Sort**

**Definition:**  
 Insertion Sort is a comparison-based algorithm that builds the final sorted array one element at a time by comparing each new element with those already sorted and inserting it at the correct position.

**Working Principle:**  
 It starts from the second element, compares it with the elements before it, and inserts it in the correct place, shifting the larger elements to the right.

**Step-by-Step Example:**  
 Given an array: [7, 3, 5, 2]

* Start with index 1 (element 3).  
   Compare with 7 → Insert before → [3, 7, 5, 2]
* Index 2 (element 5).  
   Compare with 7 → Move 7 → Compare with 3 → Insert after → [3, 5, 7, 2]
* Index 3 (element 2).  
   Move 7, 5, 3 → Insert at start → [2, 3, 5, 7]

**Time Complexity:**

* Best Case: O(n) — nearly sorted input.
* Average and Worst Case: O(n²)

**Space Complexity:**  
 O(1) — in-place sorting.

**Use Case:**  
 Efficient for small or nearly sorted datasets. It is often used in real-time systems where data is received continuously.

## **3. Quick Sort**

**Definition:**  
 Quick Sort is an efficient divide-and-conquer sorting algorithm. It works by selecting a 'pivot' element and partitioning the array such that elements less than the pivot are moved to its left and greater elements to its right. The same process is then recursively applied to the left and right subarrays.

**Working Principle:**

* Choose a pivot (e.g., last element).
* Rearrange the array so that all elements smaller than the pivot come before it and those greater come after.
* Recursively apply the same logic to the left and right partitions.

**Step-by-Step Example:**  
 Given an array: [6, 3, 9, 5, 2]

* Choose pivot = 2
* Partition: [2] (pivot is at correct place)
* Left: [6, 3, 9, 5]
* Repeat process for left and right parts until fully sorted.

**Time Complexity:**

* Best and Average Case: O(n log n)
* Worst Case: O(n²) — occurs when pivot is always the smallest or largest element

**Space Complexity:**  
 O(log n) — due to recursive call stack.

**Use Case:**  
 Preferred for large datasets where fast performance is needed. Common in system libraries.

## **4. Merge Sort**

**Definition:**  
 Merge Sort is a stable and reliable sorting algorithm that uses the divide-and-conquer technique. It divides the array into two halves, sorts each half recursively, and finally merges the sorted halves to produce the final sorted array.

**Working Principle:**

* Divide the array into two equal halves.
* Recursively sort both halves.
* Merge the sorted halves into one sorted array.

**Step-by-Step Example:**  
 Given an array: [8, 4, 3, 7, 6]

* Split into [8, 4] and [3, 7, 6]
* Sort each half: [4, 8] and [3, 6, 7]
* Merge both: [3, 4, 6, 7, 8]

**Time Complexity:**

* Best, Average, Worst Case: O(n log n)

**Space Complexity:**  
 O(n) — requires extra space for merging.

**Use Case:**  
 Ideal for scenarios where stability is required, or for sorting linked lists. Performs consistently regardless of data order.

**Code:**

class Order {

int orderId;

String customerName;

double totalPrice;

public Order(int id, String name, double price) {

this.orderId = id;

this.customerName = name;

this.totalPrice = price;

}

public String toString() {

return "Order ID: " + orderId + ", Customer: " + customerName + ", Total: ₹" + totalPrice;

}

void bubbleSort(Order[] orders) {

for (int i = 0; i < orders.length - 1; i++) {

for (int j = 0; j < orders.length - i - 1; j++) {

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

Order temp = orders[j];

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

orders[j + 1] = temp;

}

}

}

}

void quickSort(Order[] arr, int low, int high) {

if (low < high) {

int pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

int partition(Order[] arr, int low, int high) {

double pivot = arr[high].totalPrice;

int i = low - 1;

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

if (arr[j].totalPrice <= pivot) {

i++;

Order temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

Order temp = arr[i + 1];

arr[i + 1] = arr[high];

arr[high] = temp;

return i + 1;

}

}

public class Main {

public static void main(String[] args) {

// TODO Auto-generated method stub

Order[] orders = new Order[] {

new Order(101, "Ravi", 2500.50),

new Order(102, "Neha", 1499.99),

new Order(103, "Amit", 3200.00),

new Order(104, "Priya", 1999.75),

new Order(105, "Rahul", 1800.00)

};

System.out.println("Original Orders:");

for (Order o : orders) {

System.out.println(o);

}

Order sorter = new Order(0, "", 0);

sorter.bubbleSort(orders);

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

for (Order o : orders) {

System.out.println(o);

}

orders = new Order[] {

new Order(101, "Ravi", 2500.50),

new Order(102, "Neha", 1499.99),

new Order(103, "Amit", 3200.00),

new Order(104, "Priya", 1999.75),

new Order(105, "Rahul", 1800.00)

};

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

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

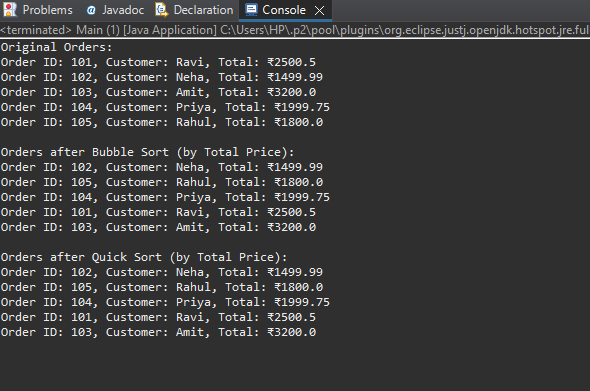
for (Order o : orders) {

System.out.println(o);

}

}}

**Output:**



### **Analysis**

|  |  |  |
| --- | --- | --- |
| **Algorithm** | **Time Complexity** | **Performance** |
| Bubble Sort | O(n²) | Very slow for large data |
| Quick Sort | O(n log n) avg | Fast and efficient |

Quick Sort is better because it handles big data faster and uses divide-and-conquer.

**Exercise 4: Employee Management System**

**Scenario:**

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

**Steps:**

1. **Understand Array Representation:**
   1. Explain how arrays are represented in memory and their advantages.
2. **Setup:**
   1. Create a class Employee with attributes like **employeeId**, **name**, **position**, and **salary**.
3. **Implementation:**
   1. Use an array to store employee records.
   2. Implement methods to **add**, **search**, **traverse**, and **delete** employees in the array.
4. **Analysis:**
   1. Analyze the time complexity of each operation (add, search, traverse, delete).
   2. Discuss the limitations of arrays and when to use them.

**Solution:**

## **Array Representation in Memory**

An array is a fundamental data structure in programming that stores a collection of elements of the same data type under a single name. What makes arrays particularly useful is how they are represented in the computer's memory.

### **Memory Representation:**

When an array is created, the system allocates a fixed block of contiguous memory locations for its elements. This means that all the elements of the array are stored one after another in adjacent memory cells.

## **Advantages of Arrays**

Arrays provide several advantages that make them a preferred choice in many situations:

### **1. Direct Access (Random Access):**

Arrays allow access to any element in constant time O(1). Since the memory locations are contiguous, the address of any element can be calculated directly using the index. This is extremely efficient for reading or writing data.

Example: Accessing arr[3] is just as fast as accessing arr[0].

### **2. Memory Efficiency:**

When the size of the data is known in advance, arrays are a memory-efficient structure because they allocate memory only once, and there's no extra overhead per element.

Especially useful in performance-critical systems or embedded environments.

### **3. Simplicity and Ease of Use:**

Arrays are simple to understand and easy to implement. Their syntax is straightforward, and most programming languages support them natively.

For beginners, arrays are usually the first data structure introduced.

### **4. Support for Algorithms:**

Many standard algorithms such as searching, sorting, and traversal work very efficiently with arrays due to their predictable structure.

Sorting techniques like Quick Sort, Merge Sort, and Binary Search are optimized for array use.

**Code:**

class Employee {

int employeeId;

String name;

String position;

double salary;

Employee(int id, String name, String pos, double sal) {

this.employeeId = id;

this.name = name;

this.position = pos;

this.salary = sal;

}

public String toString() {

return "ID: " + employeeId + ", Name: " + name + ", Position: " + position + ", Salary: ₹" + salary;

}

}

class EmployeeManager {

Employee[] employees = new Employee[100];

int count = 0;

void addEmployee(Employee e) {

employees[count++] = e;

}

Employee search(int id) {

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

if (employees[i].employeeId == id)

return employees[i];

}

return null;

}

void delete(int id) {

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

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

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

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

}

employees[--count] = null;

break;

}

}

}

void display() {

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

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

}

}

}

public class Main {

public static void main(String[] args) {

// TODO Auto-generated method stub

EmployeeManager manager = new EmployeeManager();

manager.addEmployee(new Employee(101, "Ravi Kumar", "Software Engineer", 55000));

manager.addEmployee(new Employee(102, "Neha Singh", "Project Manager", 75000));

manager.addEmployee(new Employee(103, "Arjun Mehta", "Data Analyst", 48000));

manager.addEmployee(new Employee(104, "Priya Sharma", "HR Executive", 42000));

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

manager.display();

int searchId = 103;

Employee found = manager.search(searchId);

if (found != null) {

System.out.println("\nEmployee Found (ID: " + searchId + "):");

System.out.println(found);

} else {

System.out.println("\nEmployee with ID " + searchId + " not found.");

}

int deleteId = 102;

System.out.println("\nDeleting Employee with ID: " + deleteId);

manager.delete(deleteId);

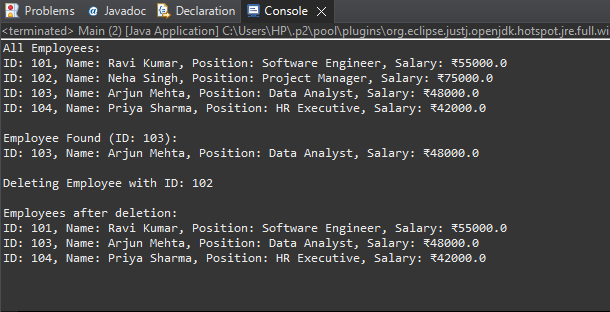
System.out.println("\nEmployees after deletion:");

manager.display();

}

}

**Output:**



### **Analysis**

|  |  |
| --- | --- |
| **Operation** | **Time Complexity** |
| Add | O(1) |
| Search | O(n) |
| Traverse | O(n) |
| Delete | O(n) |

**Limitation**: Fixed size, slow deletion, not dynamic  
Use arrays for small and fixed datasets. For dynamic data, prefer ArrayList or LinkedList.

**Exercise 5: Task Management System**

**Scenario:**

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

**Steps:**

1. **Understand Linked Lists:**
   1. Explain the different types of linked lists (Singly Linked List, Doubly Linked List).
2. **Setup:**
   1. Create a class **Task** with attributes like **taskId**, **taskName**, and **status**.
3. **Implementation:**
   1. Implement a singly linked list to manage tasks.
   2. Implement methods to **add**, **search**, **traverse**, and **delete** tasks in the linked list.
4. **Analysis:**
   1. Analyze the time complexity of each operation.
   2. Discuss the advantages of linked lists over arrays for dynamic data.

**Solution:**

## **Linked Lists:**

A **linked list** is a linear data structure in which elements, known as **nodes**, are connected through pointers. Unlike arrays, linked lists do not store data in contiguous memory locations. Instead, each node holds the data and a reference (or link) to the next node in the sequence.

Linked lists are dynamic in nature, meaning they can easily grow or shrink in size without the need to allocate or deallocate memory manually, as is required in arrays.

There are several types of linked lists, but the two most commonly used are:

## **1. Singly Linked List**

### **Definition:**

A Singly Linked List is a type of linked list where each node contains two parts:

* **Data**: The value to be stored.
* **Next**: A pointer/reference to the next node in the list.

The list starts with a head node, and the last node points to null, indicating the end of the list.

### **Working Principle:**

Traversal always begins from the head node and moves forward using the next pointers. You can move in one direction only from the first to the last node.

### **Advantages:**

* Simple to implement and understand.
* Efficient for operations like insertion and deletion at the beginning of the list.

### **Disadvantages:**

* Traversal is only possible in one direction.
* Accessing an element requires sequential traversal (no direct access).

## **2. Doubly Linked List**

### **Definition:**

A Doubly Linked List is a more advanced type of linked list in which each node contains three parts:

* **Data**: The value to be stored.
* **Prev**: A pointer to the previous node.
* **Next**: A pointer to the next node.

The list starts with a head node (whose prev is null) and ends with a tail node (whose next is null).

### **Working Principle:**

Traversal can be done in both directions forward and backward .

**Advantages:**

* Allows bidirectional traversal.
* More flexible for operations like insertion and deletion at both ends or in the middle.
* Deleting a node is easier if a pointer to it is given (no need to traverse from head).

### **Disadvantages:**

* Slightly more complex to implement.
* Requires more memory because of the extra pointer (prev).

**Code:**

class Task {

int taskId;

String taskName;

String status;

Task next;

Task(int id, String name, String status) {

this.taskId = id;

this.taskName = name;

this.status = status;

this.next = null;

}

}

class TaskManager {

Task head = null;

void addTask(int id, String name, String status) {

Task newTask = new Task(id, name, status);

if (head == null) head = newTask;

else {

Task temp = head;

while (temp.next != null)

temp = temp.next;

temp.next = newTask;

}

}

Task search(int id) {

Task temp = head;

while (temp != null) {

if (temp.taskId == id)

return temp;

temp = temp.next;

}

return null;

}

void delete(int id) {

if (head == null) return;

if (head.taskId == id) {

head = head.next;

return;

}

Task prev = head, curr = head.next;

while (curr != null && curr.taskId != id) {

prev = curr;

curr = curr.next;

}

if (curr != null) prev.next = curr.next;

}

void displayTasks() {

Task temp = head;

while (temp != null) {

System.out.println("ID: " + temp.taskId + ", Name: " + temp.taskName + ", Status: " + temp.status);

temp = temp.next;

}

}

}

public class Main {

public static void main(String[] args) {

// TODO Auto-generated method stub

TaskManager manager = new TaskManager();

manager.addTask(1, "Design Module", "Pending");

manager.addTask(2, "Implement Logic", "In Progress");

manager.addTask(3, "Test Application", "Pending");

manager.addTask(4, "Deploy Project", "Not Started");

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

manager.displayTasks();

int searchId = 3;

Task found = manager.search(searchId);

if (found != null) {

System.out.println("\nTask Found (ID: " + searchId + "):");

System.out.println("ID: " + found.taskId + ", Name: " + found.taskName + ", Status: " + found.status);

} else {

System.out.println("\nTask with ID " + searchId + " not found.");

}

int deleteId = 2;

System.out.println("\nDeleting Task with ID: " + deleteId);

manager.delete(deleteId);

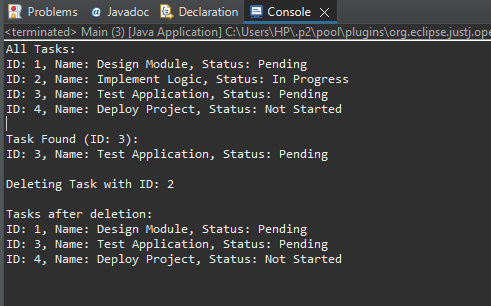
System.out.println("\nTasks after deletion:");

manager.displayTasks();

}

}

**Output:**



### **Analysis**

|  |  |
| --- | --- |
| **Operation** | **Time Complexity** |
| Add | O(n) (at end) |
| Search | O(n) |
| Delete | O(n) |
| Traverse | O(n) |

Linked Lists are better than arrays when you need frequent insert/delete. No need to shift elements like in arrays.

**Exercise 6: Library Management System**

**Scenario:**

You are developing a library management system where users can search for books by title or author.

**Steps:**

1. **Understand Search Algorithms:**
   1. Explain linear search and binary search algorithms.
2. **Setup:**
   1. Create a class **Book** with attributes like **bookId**, **title**, and **author**.
3. **Implementation:**
   1. Implement linear search to find books by title.
   2. Implement binary search to find books by title (assuming the list is sorted).
4. **Analysis:**
   1. Compare the time complexity of linear and binary search.
   2. Discuss when to use each algorithm based on the data set size and order.

**Solution:**

## **Linear Search**

### **Definition:**

Linear Search, also known as sequential search, is the simplest method of searching for an element in a list or array. It involves checking each element one by one from the beginning until the desired value is found or the end of the list is reached.

### **How It Works:**

1. Start from the first element of the array.
2. Compare the current element with the target value.
3. If a match is found, return the index or the element.
4. If no match is found by the end of the array, return a failure message (e.g., -1 or null).

### **Time Complexity:**

* **Best Case:** O(1) (if the element is the first one)
* **Average Case:** O(n/2) ≈ O(n)
* **Worst Case:** O(n) (if the element is at the end or not present)

### **When to Use:**

* When the list is unsorted.
* When the list is small and performance is not critical.
* When there's no frequent need to perform repeated searches.

## **Binary Search**

### **Definition:**

Binary Search is a highly efficient algorithm used to search for an element in a sorted list. It works by repeatedly dividing the search interval in half until the element is found or the interval becomes empty.

### **How It Works:**

1. Start by identifying the middle element of the array.
2. Compare the middle element with the target:
   1. If it matches, return it.
   2. If the target is less than the middle element, repeat the search on the left half.
   3. If the target is greater, repeat the search on the right half.
3. This process continues until the element is found or there are no elements left to search.

### **Time Complexity:**

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

### **When to Use:**

* When the list is sorted in advance.
* For large datasets where performance matters.
* In cases requiring frequent searching, such as in search engines or database indexing.

**Code:**

import java.util.Arrays;

import java.util.Comparator;

class Book {

int bookId;

String title;

String author;

public Book(int id, String title, String author) {

this.bookId = id;

this.title = title;

this.author = author;

}

public String toString() {

return "ID: " + bookId + ", Title: " + title + ", Author: " + author;

}

Book linearSearch(Book[] books, String title) {

for (Book b : books) {

if (b.title.equalsIgnoreCase(title)) return b;

}

return null;

}

int binarySearch(Book[] books, String title) {

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

while (left <= right) {

int mid = (left + right) / 2;

int cmp = books[mid].title.compareToIgnoreCase(title);

if (cmp == 0) return mid;

else if (cmp < 0) left = mid + 1;

else right = mid - 1;

}

return -1;

}

}

public class Main {

public static void main(String[] args) {

// TODO Auto-generated method stub

Book[] books = new Book[] {

new Book(101, "Java Programming", "James Gosling"),

new Book(102, "Data Structures", "Seymour Lipschutz"),

new Book(103, "Operating Systems", "Abraham Silberschatz"),

new Book(104, "Computer Networks", "Andrew S. Tanenbaum"),

new Book(105, "Database Systems", "Ramez Elmasri")

};

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

for (Book b : books) {

System.out.println(b);

}

Book searcher = new Book(0, "", "");

String searchTitle1 = "Operating Systems";

Book foundBook = searcher.linearSearch(books, searchTitle1);

if (foundBook != null) {

System.out.println("\nLinear Search Result:");

System.out.println(foundBook);

} else {

System.out.println("\nLinear Search: Book titled \"" + searchTitle1 + "\" not found.");

}

Arrays.sort(books, Comparator.comparing(b -> b.title.toLowerCase()));

String searchTitle2 = "Data Structures";

int index = searcher.binarySearch(books, searchTitle2);

if (index != -1) {

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

System.out.println(books[index]);

} else {

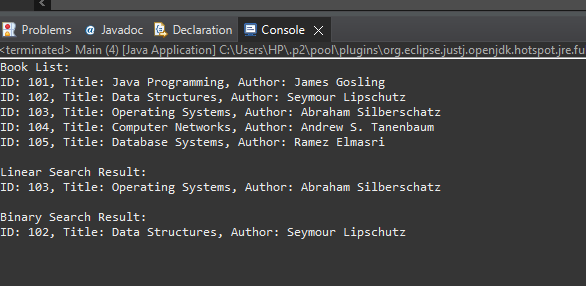
System.out.println("\nBinary Search: Book titled \"" + searchTitle2 + "\" not found.");

}

}

}

**Output:**



### **Analysis**

|  |  |  |
| --- | --- | --- |
| **Algorithm** | **Time Complexity** | **Suitable For** |
| Linear Search | O(n) | Small or unsorted data |
| Binary Search | O(log n) | Sorted large data |

Use Linear Search for small/simple systems. Use Binary Search when working with sorted data for faster results.