**WEEK-1**

**Data Structures and Algorithms HandsOn**

**Exercise 1: Inventory Management System**

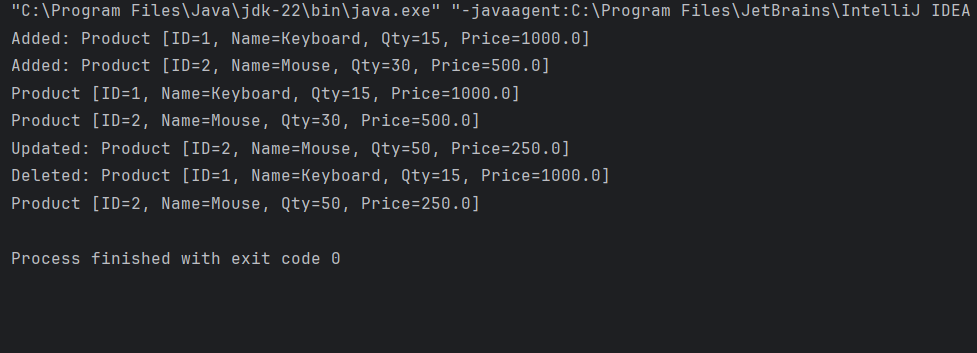
In warehouse systems that handle thousands of products, choosing the right data structure is critical. Operations like searching, updating, and deleting records must be fast and reliable to avoid performance bottlenecks.

Using an inefficient data structure can lead to slow lookups, delayed updates, and a poor user experience, especially as the dataset grows.

**Code:**

import java.util.\*;  
class Product {  
 private int productId;  
 private String productName;  
 private int quantity;  
 private double price;  
  
 public Product(int productId, String productName, int quantity, double price) {  
 this.productId = productId;  
 this.productName = productName;  
 this.quantity = quantity;  
 this.price = price;  
 }  
  
 public int getProductId() { return productId; }  
 public String getProductName() { return productName; }  
 public int getQuantity() { return quantity; }  
 public double getPrice() { return price; }  
  
 public void setQuantity(int quantity) { this.quantity = quantity; }  
 public void setPrice(double price) { this.price = price; }  
  
 @Override  
 public String toString() {  
 return "Product [ID=" + productId + ", Name=" + productName +  
 ", Qty=" + quantity + ", Price=" + price + "]";  
 }  
}  
  
  
class InventoryManager {  
 private List<Product> inventory = new ArrayList<>();  
  
 public void add(Product product) {  
 for (Product p : inventory) {  
 if (p.getProductId() == product.getProductId()) {  
 System.*out*.println("Product with ID already exists.");  
 return;  
 }  
 }  
 inventory.add(product);  
 System.*out*.println("Added: " + product);  
 }  
  
 public void update(int productId, int newQty, double newPrice) {  
 for (Product p : inventory) {  
 if (p.getProductId() == productId) {  
 p.setQuantity(newQty);  
 p.setPrice(newPrice);  
 System.*out*.println("Updated: " + p);  
 return;  
 }  
 }  
 System.*out*.println("Product not found.");  
 }  
  
 public void delete(int productId) {  
 Iterator<Product> iterator = inventory.iterator();  
 while (iterator.hasNext()) {  
 Product p = iterator.next();  
 if (p.getProductId() == productId) {  
 iterator.remove();  
 System.*out*.println("Deleted: " + p);  
 return;  
 }  
 }  
 System.*out*.println("Product not found.");  
 }  
  
 public void display() {  
 for (Product product : inventory) {  
 System.*out*.println(product);  
 }  
 }  
}  
  
class InventoryManagementSystem {  
 public static void main(String[] args) {  
 InventoryManager manager = new InventoryManager();  
  
 manager.add(new Product(1, "Keyboard", 15, 1000));  
 manager.add(new Product(2, "Mouse", 30, 500));  
  
 manager.display();  
  
 manager.update(2, 50, 250);  
 manager.delete(1);  
  
 manager.display();  
 }  
  
  
}

**Output:**

****

**Time Complexity Analysis:**

* Add Product – O(n) Must search the entire list to check for duplicate productId before insertion.
* Update Product – O(n) Linear search to locate the product by productId, then update in constant time.
* Delete Product – O(n) Uses an iterator to linearly search for the product by productId and remove it.
* Display All – O(n) Must iterate through all products in the list to display them.

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

**Big O Notation:**

Mathematical notation used to describe the performance or complexity of an algorithm. Specifically, how its runtime or memory usage increases as the input size increases.

**Inventory Operations (Using ArrayList):**

There are four major operations:  
     i) Add Product  
     ii) Update Product  
     iii) Delete Product  
     iv) Display All Products

i) Add Product  
     Best Case: O(1) – If product is added without duplicates  
     Average Case: O(n) – Linear check for existing productId  
     Worst Case: O(n) – Duplicate check found at the end of list

ii) Update Product  
     Best Case: O(1) – Product found at beginning of list  
    Average Case: O(n) – Product in middle of the list  
     Worst Case: O(n) – Product not found after full scan

iii) Delete Product  
     Best Case: O(1) – Product found and removed immediately  
     Average Case: O(n) – Product is in middle of list  
     Worst Case: O(n) – Product not found

iv) Display All Products  
     Best Case: O(n)  
     Average Case: O(n)  
     Worst Case: O(n)

**Code:**

import java.util.\*;  
class item {  
 private int productId;  
 private String productName;  
 private String category;  
  
 public item(int productId, String productName, String category) {  
 this.productId = productId;  
 this.productName = productName;  
 this.category = category;  
 }  
  
 public int getProductId() { return productId; }  
 public String getProductName() { return productName; }  
 public String getCategory() { return category; }  
  
 @Override  
 public String toString() {  
 return "Product [ID=" + productId + ", Name=" + productName + ", Category=" + category + "]";  
 }  
}  
  
  
class SearchEngine {  
  
 public static item linearSearch(item[] products, String name) {  
 for (item p : products) {  
 if (p.getProductName().equalsIgnoreCase(name)) {  
 return p;  
 }  
 }  
 return null;  
 }  
  
 public static item binarySearch(item[] products, String name) {  
 int low = 0, high = products.length - 1;  
  
 while (low <= high) {  
 int mid = (low + high) / 2;  
 int compare = products[mid].getProductName().compareToIgnoreCase(name);  
  
 if (compare == 0)  
 return products[mid];  
 else if (compare < 0)  
 low = mid + 1;  
 else  
 high = mid - 1;  
 }  
 return null;  
 }  
  
  
 public static void sortByName(item[] products) {  
 Arrays.*sort*(products, Comparator.*comparing*(item::getProductName, String.*CASE\_INSENSITIVE\_ORDER*));  
 }  
}  
  
class Ecommerce {  
 public static void main(String[] args) {  
 item[] products = {  
 new item(1, "Laptop", "Electronics"),  
 new item(2, "Football", "Sports")  
 };  
  
  
 item foundLinear = SearchEngine.*linearSearch*(products, "Football");  
 System.*out*.println("Found " + (foundLinear != null ? foundLinear : "Not Found")+" using Linear Search");  
  
  
 SearchEngine.*sortByName*(products);  
 item foundBinary = SearchEngine.*binarySearch*(products, "Laptop");  
 System.*out*.println("Found " + (foundBinary != null ? foundBinary : "Not Found")+" using Binary Search");  
 }  
}

**Output:**

**A screen shot of a computer

AI-generated content may be incorrect.**

**Exercise 3: Sorting Customer Orders**

**Sorting Algorithms:**

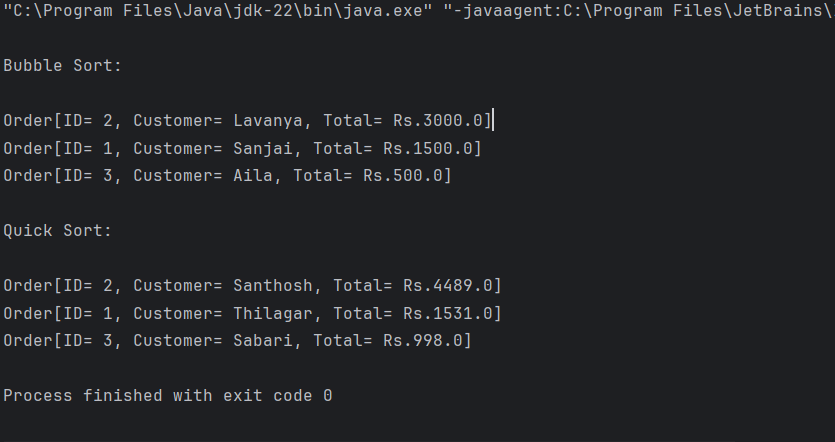
**Bubble Sort :** Repeatedly compares adjacent elements and swaps them if they are in the wrong order.

**Insertion Sort :** Builds the sorted array one element at a time by inserting elements into their correct position.

**Quick Sort :** A divide-and-conquer algorithm. Selects a pivot, partitions the array, and recursively sorts subarrays.

**Merge Sort :** Divides array into halves, recursively sorts, and merges sorted halves.

**Code:**

****

Bubble Sort is simple but inefficient for large datasets. Quick sort is generally faster because it reduces the number of comparison via divide and conquer approach. So, quick sort is preferred.

**Exercise 4: Employee Management System**

**Arrays:**

Arrays are contiguous blocks of memory where each element is stored one after another. The address of the first element and an index allow quick access to any element in O(1) time using base + index \* size.

**Advantages of Array:**

 Fast random access using index – O(1) time.  
    Simple and easy to implement for fixed-size data.  
    Better cache locality – improves performance due to sequential memory allocation**.**

**Disadvantages of Array:**

Fixed size – cannot grow or shrink at runtime without creating a new array.  
    Insertion/deletion is costly – O(n) in worst case due to shifting elements.  
    Wastage of memory – if allocated size is more than required.

**Code:**

class Employee { private int employeeId;  
 private String name;  
 private String position;  
 private double salary;  
  
 public Employee(int employeeId, String name, String position, double salary) {  
 this.employeeId = employeeId;  
 this.name = name;  
 this.position = position;  
 this.salary = salary;  
 }  
  
 public int getEmployeeId() { return employeeId; }  
 public String getName() { return name; }  
 public String getPosition() { return position; }  
 public double getSalary() { return salary; }  
  
 @Override  
 public String toString() {  
 return "Employee[ID=" + employeeId + ", Name=" + name +  
 ", Position=" + position + ", Salary=" + salary + "]";  
 }  
}  
class EmployeeManager {  
 private Employee[] employe;  
 private int size;  
  
 public EmployeeManager(int total) {  
 employe = new Employee[total];  
 size = 0;  
 }  
  
 public void add(Employee emp) {  
 if (size < employe.length) {  
 employe[size++] = emp;  
 System.*out*.println("Added: " + emp);  
 } else {  
 System.*out*.println("Cannot add: Array is full.");  
 }  
 }  
  
 public Employee search(int empId) {  
 for (int i = 0; i < size; i++) {  
 if (employe[i].getEmployeeId() == empId) {  
 return employe[i];  
 }  
 }  
 return null;  
 }  
  
  
 public void traverse() {  
 System.*out*.println("\nEmployee List:");  
 for (int i = 0; i < size; i++) {  
 System.*out*.println(employe[i]);  
 }  
 }  
  
 public void delete(int empId) {  
 for (int i = 0; i < size; i++) {  
 if (employe[i].getEmployeeId() == empId) {  
 for (int j = i; j < size - 1; j++) {  
 employe[j] = employe[j + 1];  
 }  
 employe[--size] = null;  
 System.*out*.println("Deleted employee with ID: " + empId);  
 return;  
 }  
 }  
 System.*out*.println("Employee not found.");  
 }  
}  
  
class EmployeeManagementSystem {  
 public static void main(String[] args) {  
 EmployeeManager manager = new EmployeeManager(3);  
  
 manager.add(new Employee(1, "Sanjai", "Web Scraper", 100000));  
 manager.add(new Employee(2, "Lavanya", "Developer", 65000));  
 manager.add(new Employee(3, "Aila", "Designer", 60000));  
  
 manager.traverse();  
  
 System.*out*.println("\nSearching for employee with ID 2:");  
 Employee found = manager.search(3);  
 System.*out*.println(found != null ? found : "Not Found");  
  
 manager.delete(2);  
 manager.traverse();  
 }  
}

**Output:**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**Exercise 5: Task Management System**

**Singly Linked List:**

A singly linked list is a linear data structure where each element (node) contains data and a reference (pointer) to the next node in the sequence.

**Characteristics of Singly Linked List:**

Each node holds two parts: data and next.  
    Traversal is only possible in one direction — from head to tail.  
    The last node points to null, indicating the end of the list.  
    No backward traversal is possible.

**Advantages of Singly Linked List:**

Efficient insertion and deletion at the beginning — O(1) time.  
    Dynamic size — grows or shrinks at runtime.  
    Memory-efficient compared to doubly linked list (uses one pointer per node).

**Disadvantages of Singly Linked List:**

No direct access by index — requires O(n) traversal.  
    Cannot traverse backward.  
    Inserting/deleting at the end takes O(n) time (unless tail is tracked).

**Code:**

class Task {  
 private int taskId;  
 private String taskName;  
 private String status;  
  
 public Task(int taskId, String taskName, String status) {  
 this.taskId = taskId;  
 this.taskName = taskName;  
 this.status = status;  
 }  
  
 public int getTaskId() { return taskId; }  
 public String getTaskName() { return taskName; }  
 public String getStatus() { return status; }  
  
 @Override  
 public String toString() {  
 return "Task[ID=" + taskId + ", Name=" + taskName + ", Status=" + status + "]";  
 }  
}  
class TaskNode {  
 Task task;  
 TaskNode next;  
  
 public TaskNode(Task task) {  
 this.task = task;  
 this.next = null;  
 }  
}  
class TaskManager {  
 private TaskNode head;  
  
 public void add(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;  
 }  
 System.*out*.println("Added: " + task);  
 }  
  
 public Task search(int taskId) {  
 TaskNode current = head;  
 while (current != null) {  
 if (current.task.getTaskId() == taskId) {  
 return current.task;  
 }  
 current = current.next;  
 }  
 return null;  
 }  
  
 public void traverse() {  
 System.*out*.println("\nAll Tasks:");  
 TaskNode current = head;  
 while (current != null) {  
 System.*out*.println(current.task);  
 current = current.next;  
 }  
 }  
  
 public void delete(int taskId) {  
 if (head == null) return;  
  
 if (head.task.getTaskId() == taskId) {  
 System.*out*.println("Deleted: " + head.task);  
 head = head.next;  
 return;  
 }  
  
 TaskNode current = head;  
 while (current.next != null) {  
 if (current.next.task.getTaskId() == taskId) {  
 System.*out*.println("Deleted: " + current.next.task);  
 current.next = current.next.next;  
 return;  
 }  
 current = current.next;  
 }  
  
 System.*out*.println("Task with ID " + taskId + " not found.");  
 }  
}  
  
class TaskManagementSystem {  
 public static void main(String[] args) {  
 TaskManager manager = new TaskManager();  
  
 manager.add(new Task(1, "Develop Bot", "Completed"));  
 manager.add(new Task(2, "Connect DB", "In Progress"));  
 manager.add(new Task(3, "Deploy Bot", "Pending"));  
  
 manager.traverse();  
  
 System.*out*.println("\nSearch Task with ID 3:");  
 Task result = manager.search(2);  
 System.*out*.println(result != null ? result : "Not Found");  
  
 manager.delete(2);  
 manager.traverse();  
  
 }  
}

**Output:A screenshot of a computer program

AI-generated content may be incorrect.**

**Exercise 6: Library Management System**

**Code:**

import java.util.\*;class Book {  
 private int bookId;  
 private String title;  
 private String author;  
  
 public Book(int bookId, String title, String author) {  
 this.bookId = bookId;  
 this.title = title;  
 this.author = author;  
 }  
  
 public int getBookId() { return bookId; }  
 public String getTitle() { return title; }  
 public String getAuthor() { return author; }  
  
 @Override  
 public String toString() {  
 return "Book[ID=" + bookId + ", Title=\"" + title + "\", Author=" + author + "]";  
 }  
}  
  
  
class Library {  
  
 public static Book linear(Book[] books, String title) {  
 for (Book book : books) {  
 if (book.getTitle().equalsIgnoreCase(title)) {  
 return book;  
 }  
 }  
 return null;  
 }  
  
 public static void sort(Book[] books) {  
 Arrays.*sort*(books, Comparator.*comparing*(Book::getTitle, String.*CASE\_INSENSITIVE\_ORDER*));  
 }  
  
 public static Book binary(Book[] books, String title) {  
 int low = 0, high = books.length - 1;  
  
 while (low <= high) {  
 int mid = (low + high) / 2;  
 int cmp = books[mid].getTitle().compareToIgnoreCase(title);  
  
 if (cmp == 0)  
 return books[mid];  
 else if (cmp < 0)  
 low = mid + 1;  
 else  
 high = mid - 1;  
 }  
  
 return null;  
 }  
  
 public static void display(Book[] books) {  
 for (Book book : books) {  
 System.*out*.println(book);  
 }  
 }  
}  
  
class LibraryManagementSystem {  
 public static void main(String[] args) {  
 Book[] books = {  
 new Book(1, "The Alchemist", "Paulo Coelho"),  
 new Book(2, "To Kill a Mockingbird", "Harper Lee"),  
 new Book(3, "1984", "George Orwell")  
 };  
  
 System.*out*.println("All Books:");  
 Library.*display*(books);  
  
 System.*out*.println("\nLinear Search for '1984':");  
 Book result1 = Library.*linear*(books, "1984");  
 System.*out*.println(result1 != null ? result1 : "Book not found");  
  
  
 System.*out*.println("\nBinary Search for 'Moby Dick':");  
 Book result2 = Library.*binary*(books, "Moby Dick");  
 System.*out*.println(result2 != null ? result2 : "Book not found");  
  
 }  
}

**Output:**

**A computer screen shot of a computer code

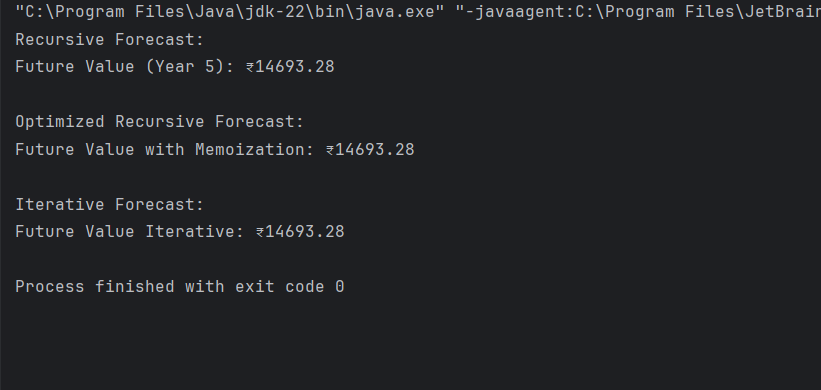
AI-generated content may be incorrect.**

**Exercise 7: Financial Forecasting**

**Code:**

class Financial{  
  
 public static double futureValue(double amount, double interest, int years) {  
 if (years == 0) {  
 return amount;  
 }  
 return (1 + interest) \* *futureValue*(amount, interest, years - 1);  
 }  
  
 public static double futureValueMemo(double amount, double interest, int years, Double[] memo) {  
 if (years == 0) return amount;  
 if (memo[years] != null) return memo[years];  
  
 memo[years] = (1 + interest) \* *futureValueMemo*(amount, interest, years - 1, memo);  
 return memo[years];  
 }  
  
 public static double futureValueIterative(double amount, double interest, int years) {  
 double result = amount;  
 for (int i = 0; i < years; i++) {  
 result \*= (1 + interest);  
 }  
 return result;  
 }  
}  
  
class FinancialForecasting {  
 public static void main(String[] args) {  
 double amount = 10000;  
 double interest = 0.08;  
 int years = 5;  
  
 System.*out*.println("Recursive Forecast:");  
 double future = Financial.*futureValue*(amount, interest, years);  
 System.*out*.printf("Future Value (Year %d): ₹%.2f\n", years, future);  
  
 System.*out*.println("\nOptimized Recursive Forecast:");  
 Double[] memo = new Double[years + 1];  
 double futureMemo = Financial.*futureValueMemo*(amount, interest, years, memo);  
 System.*out*.printf("Future Value with Memoization: ₹%.2f\n", futureMemo);  
  
 System.*out*.println("\nIterative Forecast:");  
 double futureIter = Financial.*futureValueIterative*(amount, interest, years);  
 System.*out*.printf("Future Value Iterative: ₹%.2f\n", futureIter);  
  
 }  
}

**Output:**

****