# Data Structures & Algorithms Exercises

**Exercise 1: Inventory Management System**

1. Understanding the Problem:

Efficient data structures and algorithms are essential for handling large inventories as they directly influence the speed and memory usage of storage, search, and update operations. In real-time warehouse environments, quick access to product details is critical for operations and customer satisfaction.

Suitable data structures include:

- HashMap: Best for key-based access such as using product ID.

- ArrayList: Useful for simple sequential access or small inventories.

2. Setup:

A new Java project is created for managing inventory.

3. Implementation:

A Product class is defined with attributes: productId, productName, quantity, and price.

A HashMap<Integer, Product> is used for storing products, with the following methods:

- addProduct(Product p)

- updateProduct(int id, Product updatedProduct)

- deleteProduct(int id)

Program

import java.util.\*;

class Product {

int productId;

String productName;

int quantity;

double price;

Product(int id, String name, int qty, double price) {

this.productId = id;

this.productName = name;

this.quantity = qty;

this.price = price;

}

}

class InventoryManager {

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

void addProduct(Product p) {

inventory.put(p.productId, p);

}

void updateProduct(int id, Product updated) {

inventory.put(id, updated);

}

void deleteProduct(int id) {

inventory.remove(id);

}

void displayInventory() {

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

System.out.println(p.productId + " - " + p.productName + " - " + p.quantity + " - $" + p.price);

}

}

}

public class Main {

public static void main(String[] args) {

InventoryManager manager = new InventoryManager();

Product p1 = new Product(101, "Laptop", 10, 75000.0);

Product p2 = new Product(102, "Mouse", 50, 500.0);

Product p3 = new Product(103, "Keyboard", 30, 1500.0);

manager.addProduct(p1);

manager.addProduct(p2);

manager.addProduct(p3);

System.out.println("Initial Inventory:");

manager.displayInventory();

Product updatedMouse = new Product(102, "Wireless Mouse", 40, 800.0);

manager.updateProduct(102, updatedMouse);

manager.deleteProduct(103);

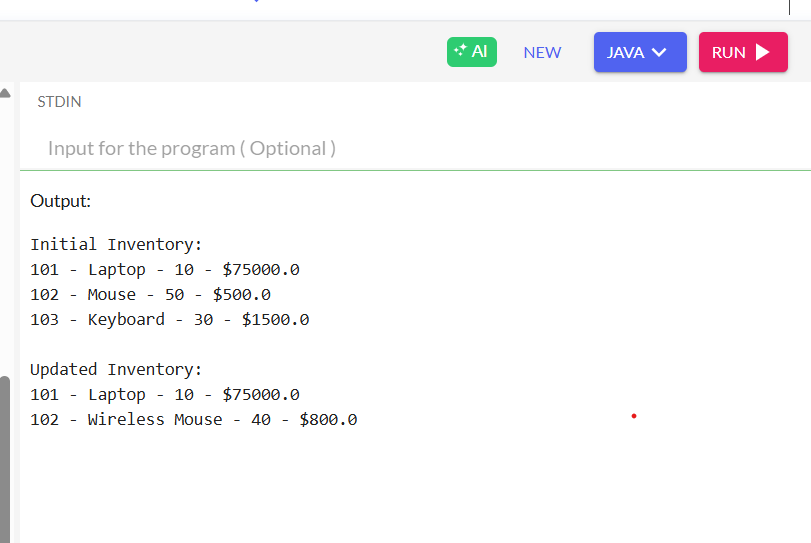
System.out.println("\nUpdated Inventory:");

manager.displayInventory();

}

}

Output



4. Analysis:

- Add/Update/Delete in a HashMap: O(1) average time.

- Optimization includes input validation and consistent hash keys.

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

1. Understanding Asymptotic Notation:

Big O notation describes the performance or complexity of an algorithm.

- Best case: Optimal condition (e.g., first match in linear search).

- Average case: Expected performance over typical inputs.

- Worst case: Most time-consuming scenario (e.g., item not found).

2. Setup:

Product class includes productId, productName, and category.

3. Implementation:

- Linear Search: Iterates through unsorted array.

- Binary Search: Requires sorted array of products by name or ID.

Program

import java.util.Arrays;

class SearchProduct {

int productId;

String productName;

String category;

SearchProduct(int id, String name, String category) {

this.productId = id;

this.productName = name;

this.category = category;

}

}

class SearchEngine {

static int linearSearch(SearchProduct[] arr, String key) {

for (int i = 0; i < arr.length; i++) {

if (arr[i].productName.equalsIgnoreCase(key)) return i;

}

return -1;

}

static int binarySearch(SearchProduct[] arr, String key) {

int low = 0, high = arr.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

int comp = arr[mid].productName.compareToIgnoreCase(key);

if (comp == 0) return mid;

else if (comp < 0) low = mid + 1;

else high = mid - 1;

}

return -1;

}

}

public class Main {

public static void main(String[] args) {

SearchProduct[] products = {

new SearchProduct(1, "Laptop", "Electronics"),

new SearchProduct(2, "Mouse", "Accessories"),

new SearchProduct(3, "Keyboard", "Accessories"),

new SearchProduct(4, "Monitor", "Electronics"),

new SearchProduct(5, "Camera", "Photography")

};

// Linear Search Example

String searchKey1 = "Keyboard";

int index1 = SearchEngine.linearSearch(products, searchKey1);

System.out.println("Linear Search: " + (index1 != -1 ? "Found at index " + index1 : "Not Found"));

// Sort array before Binary Search

Arrays.sort(products, (a, b) -> a.productName.compareToIgnoreCase(b.productName));

// Binary Search Example

String searchKey2 = "Camera";

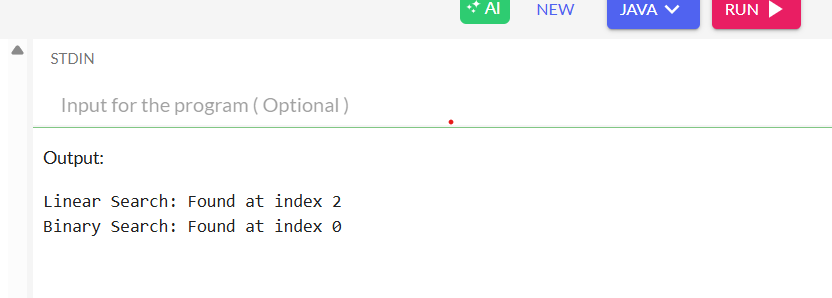
int index2 = SearchEngine.binarySearch(products, searchKey2);

System.out.println("Binary Search: " + (index2 != -1 ? "Found at index " + index2 : "Not Found"));

}

}

Output



4. Analysis:

- Linear Search: O(n)

- Binary Search: O(log n)

Binary search is preferable when data is sorted and large in size.

**Exercise 3: Sorting Customer Orders**

1. Understanding Sorting Algorithms:

- Bubble Sort: Simple but slow (O(n²)).

- Quick Sort: Efficient with average complexity O(n log n), faster in most cases.

2. Setup:

An Order class is created with orderId, customerName, and totalPrice.

3. Implementation:

Two sorting functions are implemented:

- bubbleSort(Order[] orders)

- quickSort(Order[] orders, int low, int high)

Program

class Order {

int orderId;

String customerName;

double totalPrice;

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

this.orderId = id;

this.customerName = name;

this.totalPrice = price;

}

void display() {

System.out.println(orderId + " - " + customerName + " - ₹" + totalPrice);

}

}

class SortOrders {

static void bubbleSort(Order[] arr) {

int n = arr.length;

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

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

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

Order temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

static 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);

}

}

static 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) {

Order[] orders = {

new Order(101, "Alice", 4500.0),

new Order(102, "Bob", 12000.0),

new Order(103, "Charlie", 3200.0),

new Order(104, "David", 5000.0)

};

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

for (Order o : orders) o.display();

// Bubble Sort

SortOrders.bubbleSort(orders);

System.out.println("\nAfter Bubble Sort (Ascending by Price):");

for (Order o : orders) o.display();

// Recreate unsorted orders

orders = new Order[] {

new Order(101, "Alice", 4500.0),

new Order(102, "Bob", 12000.0),

new Order(103, "Charlie", 3200.0),

new Order(104, "David", 5000.0)

};

// Quick Sort

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

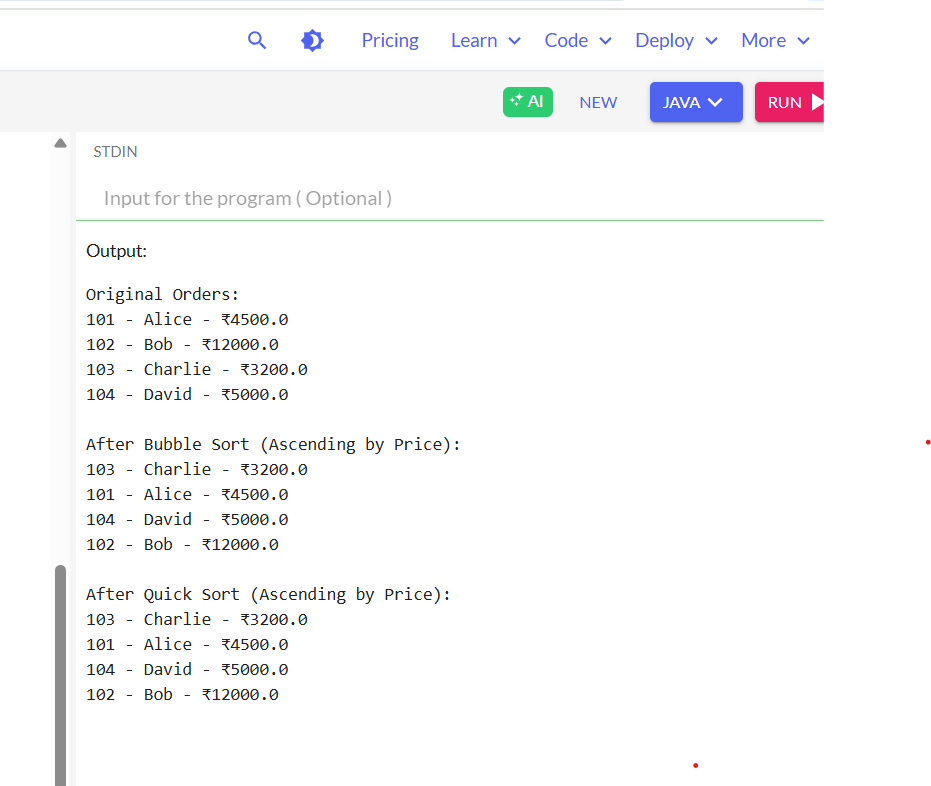
System.out.println("\nAfter Quick Sort (Ascending by Price):");

for (Order o : orders) o.display();

}

}

Output



4. Analysis:

Quick Sort is generally preferred over Bubble Sort for better performance, especially with larger data sets.

**Exercise 4: Employee Management System**

1. Understanding Array Representation:

Arrays are stored in contiguous memory, allowing fast index-based access. However, they have a fixed size and require shifting during deletions or insertions.

2. Setup:

An Employee class is created with fields employeeId, name, position, and salary.

3. Implementation:

Operations on an Employee[] array include:

- Add employee

- Search employee by ID

- Traverse list

- Delete employee by shifting elements

Program

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;

}

}

class EmployeeManager {

Employee[] employees = new Employee[100];

int count = 0;

void addEmployee(Employee e) {

if (count < employees.length) {

employees[count++] = e;

} else {

System.out.println("Employee list is full!");

}

}

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 displayAll() {

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

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

System.out.println(employees[i].employeeId + " - " + employees[i].name + " - " + employees[i].position + " - ₹" + employees[i].salary);

}

}

}

public class Main {

public static void main(String[] args) {

EmployeeManager manager = new EmployeeManager();

// Add Employees

manager.addEmployee(new Employee(101, "Alice", "Manager", 75000));

manager.addEmployee(new Employee(102, "Bob", "Developer", 60000));

manager.addEmployee(new Employee(103, "Charlie", "Tester", 50000));

// Display all

manager.displayAll();

// Search

System.out.println("\nSearching for Employee with ID 102:");

Employee emp = manager.search(102);

if (emp != null) {

System.out.println(emp.employeeId + " - " + emp.name + " - " + emp.position + " - ₹" + emp.salary);

} else {

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

}

// Delete

System.out.println("\nDeleting Employee with ID 102...");

manager.delete(102);

// Display after deletion

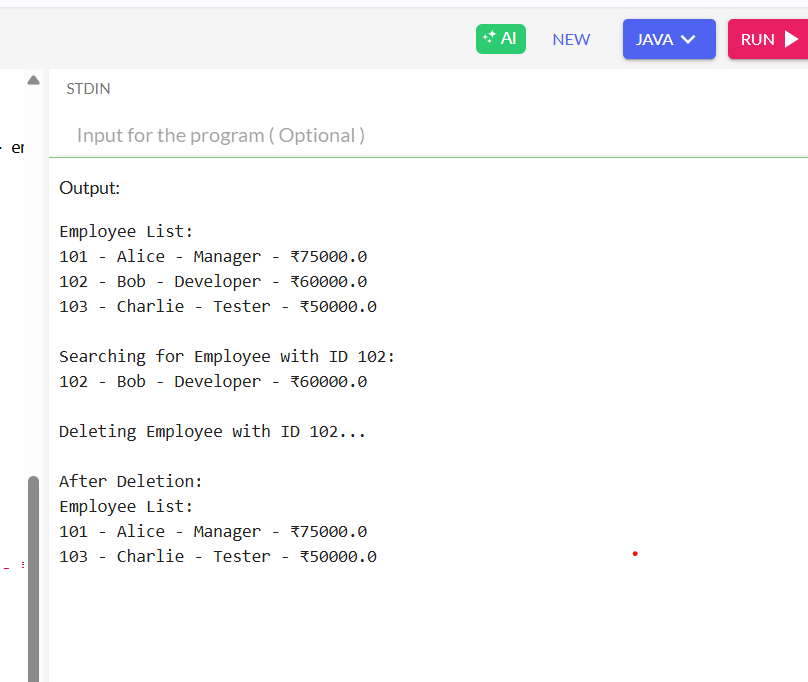
System.out.println("\nAfter Deletion:");

manager.displayAll();

}

}

Output



4. Analysis:

- Add: O(1) if space available

- Search/Delete: O(n)

Arrays are suitable for small datasets with minimal deletions.

**Exercise 5: Task Management System**

1. Understanding Linked Lists:

Linked lists allow dynamic memory allocation and ease of insertion and deletion compared to arrays. Types:

- Singly Linked List: Nodes point to the next node.

- Doubly Linked List: Nodes have both previous and next references.

2. Setup:

The Task class contains taskId, taskName, and status.

3. Implementation:

A singly linked list is used with methods to:

- Add tasks at the beginning or end

- Search by taskId

- Traverse all tasks

- Delete a task

Program

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;

void addTask(Task t) {

t.next = head;

head = t;

}

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) {

Task temp = head, prev = null;

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

prev = temp;

temp = temp.next;

}

if (temp != null) {

if (prev == null) head = temp.next;

else prev.next = temp.next;

}

}

void traverse() {

Task temp = head;

while (temp != null) {

System.out.println(temp.taskId + ": " + temp.taskName + " [" + temp.status + "]");

temp = temp.next;

}

}

}

public class Main {

public static void main(String[] args) {

TaskManager manager = new TaskManager();

// Adding tasks

manager.addTask(new Task(1, "Design UI", "Pending"));

manager.addTask(new Task(2, "Develop Backend", "In Progress"));

manager.addTask(new Task(3, "Write Tests", "Pending"));

// Traversing list

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

manager.traverse();

// Searching task

System.out.println("\nSearching Task with ID 2:");

Task found = manager.search(2);

if (found != null) {

System.out.println("Found: " + found.taskName + " [" + found.status + "]");

} else {

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

}

// Deleting a task

System.out.println("\nDeleting Task with ID 1...");

manager.delete(1);

// Traversing list after deletion

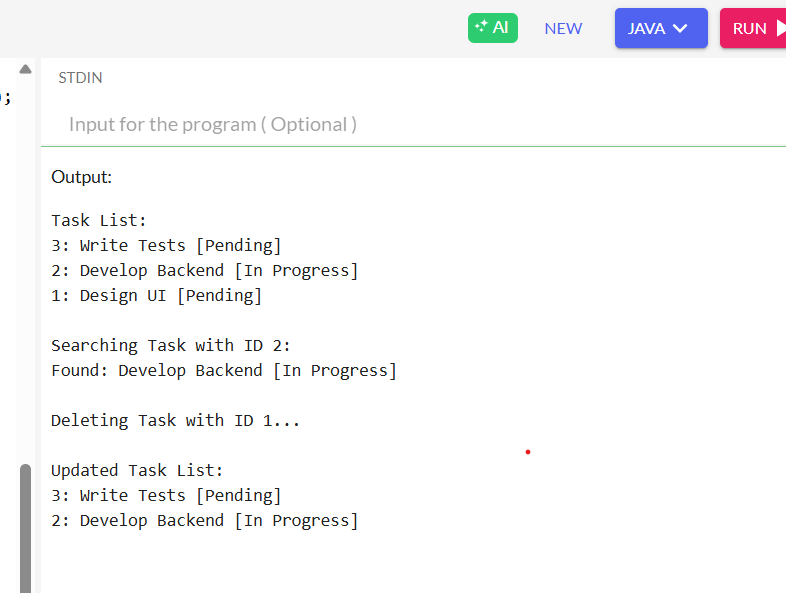
System.out.println("\nUpdated Task List:");

manager.traverse();

}

}

Output



4. Analysis:

- Add/Delete/Search: O(n) (O(1) if done at head)

Linked lists are better suited for dynamic lists with frequent changes.

**Exercise 6: Library Management System**

1. Understanding Search Algorithms:

- Linear Search: Suitable for small or unsorted datasets.

- Binary Search: Efficient for large, sorted data.

2. Setup:

A Book class is created with bookId, title, and author.

3. Implementation:

- Linear search loops through the array.

- Binary search applies divide-and-conquer on a sorted list.

Program

import java.util.Arrays;

class Book {

int bookId;

String title;

String author;

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

this.bookId = id;

this.title = title;

this.author = author;

}

void display() {

System.out.println(bookId + " - " + title + " by " + author);

}

}

class LibrarySearch {

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

for (int i = 0; i < books.length; i++) {

if (books[i].title.equalsIgnoreCase(title)) return i;

}

return -1;

}

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

int low = 0, high = books.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

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

if (cmp == 0) return mid;

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

else high = mid - 1;

}

return -1;

}

}

public class Main {

public static void main(String[] args) {

Book[] library = {

new Book(1, "The Alchemist", "Paulo Coelho"),

new Book(2, "Clean Code", "Robert C. Martin"),

new Book(3, "Atomic Habits", "James Clear"),

new Book(4, "Java Programming", "Herbert Schildt"),

new Book(5, "1984", "George Orwell")

};

// Linear Search Test

String searchTitle1 = "Atomic Habits";

int index1 = LibrarySearch.linearSearch(library, searchTitle1);

System.out.println("Linear Search:");

if (index1 != -1) library[index1].display();

else System.out.println("Book not found");

// Binary Search requires sorted array

Arrays.sort(library, (a, b) -> a.title.compareToIgnoreCase(b.title));

// Binary Search Test

String searchTitle2 = "Clean Code";

int index2 = LibrarySearch.binarySearch(library, searchTitle2);

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

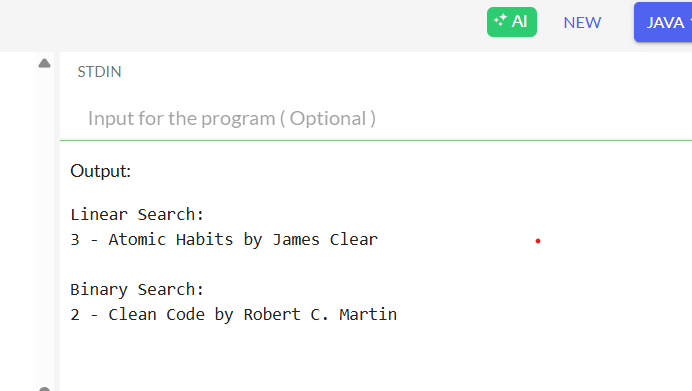
if (index2 != -1) library[index2].display();

else System.out.println("Book not found");

}

}

Output



4. Analysis:

- Linear: O(n)

- Binary: O(log n)

Binary search is more efficient, but only applicable to sorted data.

**Exercise 7: Financial Forecasting**

1. Understanding Recursive Algorithms:

Recursion solves problems by breaking them into smaller subproblems. It helps express logic in a concise way but can be inefficient without optimization.

2. Setup:

A recursive method is defined to calculate future value using:

futureValue = presentValue \* (1 + rate)^years

3. Implementation:

A recursive method calculates value over time:

Program

class Forecasting {

static double forecast(int years, double currentValue, double rate) {

if (years == 0) return currentValue;

return forecast(years - 1, currentValue, rate) \* (1 + rate);

}

}

public class Main {

public static void main(String[] args) {

double currentValue = 10000.0; // starting amount

double growthRate = 0.10; // 10% annual growth

int years = 5; // forecast for 5 years

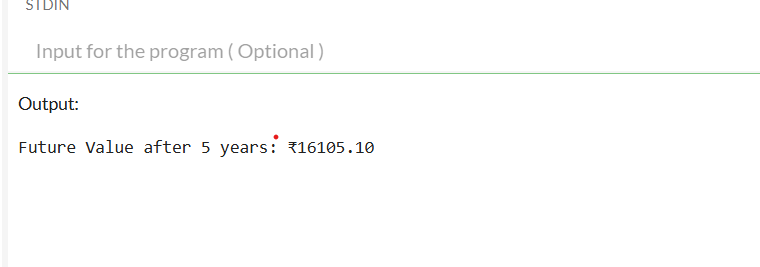
double futureValue = Forecasting.forecast(years, currentValue, growthRate);

System.out.printf("Future Value after %d years: ₹%.2f\n", years, futureValue);

}

}

Output



4. Analysis:

- Time complexity: O(n)

- Optimizable using memoization or iteration to avoid stack overflow.