**Exercise 1: Inventory Management System**

**1. Why Data Structures & Algorithms?**

* Large inventories need fast operations (add/search/delete).
* Efficient algorithms reduce time complexity; data structures like HashMap provide O(1) access.

**2. Suitable Data Structures:**

* ArrayList: Good for ordered data.
* HashMap: Best for fast access using productId.

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;

}

public String toString() {

return productName + " (Qty: " + quantity + ", Price: " + price + ")";

}

}

public class InventorySystem {

public static void main(String[] args) {

java.util.Map<Integer, Product> inventory = new java.util.HashMap<>();

inventory.put(1, new Product(1, "Laptop", 5, 70000));

inventory.put(2, new Product(2, "Mouse", 50, 500));

inventory.get(1).quantity = 10; // Update

inventory.remove(2); // Delete

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

System.out.println(p);

}

}

}

Output:- Laptop (Qty: 10, Price: 70000.0)

**4. Time Complexity:**

* Add: O(1)
* Update: O(1)
* Delete: O(1)

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

**1. Big O Notation:**

* Best, Avg, Worst case time to predict performance.
* Linear Search: O(n), Binary Search: O(log n)

class Product {

int id;

String name;

String category;

Product(int id, String name, String cat) {

this.id = id;

this.name = name;

this.category = cat;

}

}

public class SearchDemo {

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

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

if (products[i].name.equals(name)) return i;

}

return -1;

}

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

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

while (low <= high) {

int mid = (low + high) / 2;

int cmp = products[mid].name.compareTo(name);

if (cmp == 0) return mid;

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

else high = mid - 1;

}

return -1;

}

public static void main(String[] args) {

Product[] sortedProducts = {

new Product(1, "Keyboard", "Accessories"),

new Product(2, "Laptop", "Electronics"),

new Product(3, "Monitor", "Display")

};

System.out.println(linearSearch(sortedProducts, "Laptop")); // Output: 1

System.out.println(binarySearch(sortedProducts, "Laptop")); // Output: 1

}

}

Output:- 1

1

**3. Binary is better for sorted data (O(log n)), linear for unsorted (O(n)).**

**Exercise 3: Sorting Customer Orders**

**1. Sorting Algorithms:**

* Bubble Sort: O(n^2), simple
* Quick Sort: O(n log n), efficient

class Order {

int orderId;

String customer;

double totalPrice;

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

this.orderId = id;

this.customer = name;

this.totalPrice = price;

}

}

public class SortOrders {

public static 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;

}

}

}

}

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[] 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 static void main(String[] args) {

Order[] orders = {

new Order(1, "Alice", 5000),

new Order(2, "Bob", 3000),

new Order(3, "Carol", 7000)

};

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

for (Order o : orders) {

System.out.println(o.customer + " - " + o.totalPrice);

}

}

}

Output:- Bob - 3000.0

Alice - 5000.0

Carol - 7000.0

**3. Quick Sort is faster for large datasets.**

**Exercise 4: Employee Management System**

**1. Array Representation:**

* Arrays are stored in contiguous memory blocks.
* Direct access via index (O(1)).

class Employee {

int employeeId;

String name;

String position;

double salary;

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

this.employeeId = id;

this.name = name;

this.position = pos;

this.salary = salary;

}

public String toString() {

return name + " (" + position + ", Salary: " + salary + ")";

}

}

public class EmployeeSystem {

public static void main(String[] args) {

Employee[] employees = new Employee[3];

employees[0] = new Employee(101, "Amit", "Manager", 80000);

employees[1] = new Employee(102, "Neha", "Developer", 60000);

employees[2] = new Employee(103, "Ravi", "Designer", 50000);

// Traverse

for (Employee e : employees) {

System.out.println(e);

}

// Search

for (Employee e : employees) {

if (e.name.equals("Neha")) {

System.out.println("Found: " + e);

}

}

// Delete (set to null)

employees[1] = null;

System.out.println("After Deletion:");

for (Employee e : employees) {

if (e != null)

System.out.println(e);

}

}

}

Output:- Amit (Manager, Salary: 80000.0)

Neha (Developer, Salary: 60000.0)

Ravi (Designer, Salary: 50000.0)

Found: Neha (Developer, Salary: 60000.0)

After Deletion:

Amit (Manager, Salary: 80000.0)

Ravi (Designer, Salary: 50000.0)

**3. Time Complexity:**

* Add: O(1)
* Search: O(n)
* Delete: O(1) (setting to null)
* Traverse: O(n)

**Exercise 5: Task Management System**

**1. Linked Lists:**

* Singly Linked List: one-way connection.
* Doubly Linked List: two-way navigation.

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;

}

}

public class TaskList {

Task head;

public 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;

}

}

public void traverseTasks() {

Task temp = head;

while (temp != null) {

System.out.println(temp.taskName + " - " + temp.status);

temp = temp.next;

}

}

public void deleteTask(int id) {

if (head == null) return;

if (head.taskId == id) head = head.next;

else {

Task temp = head;

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

temp = temp.next;

}

if (temp.next != null) temp.next = temp.next.next;

}

}

public static void main(String[] args) {

TaskList list = new TaskList();

list.addTask(1, "Design UI", "Pending");

list.addTask(2, "Write Backend", "In Progress");

list.traverseTasks();

list.deleteTask(1);

System.out.println("After Deletion:");

list.traverseTasks();

}

}

Output:- Design UI - Pending

Write Backend - In Progress

After Deletion:

Write Backend - In Progress

**3. Time Complexity:**

* Add: O(n)
* Traverse: O(n)
* Delete: O(n)
* Search: O(n)

**Advantages:**

* Dynamic size, easy insertion/deletion without shifting.

**Exercise 6: Library Management System**

**1. Search Algorithms:**

* Linear Search: O(n), simple, unsorted.
* Binary Search: O(log n), needs sorted data.

class Book {

int bookId;

String title;

String author;

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

this.bookId = id;

this.title = title;

this.author = author;

}

}

public class LibrarySearch {

public 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;

}

public 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 static void main(String[] args) {

Book[] sortedBooks = {

new Book(1, "Data Structures", "Mark Allen"),

new Book(2, "Java Programming", "Kathy Sierra"),

new Book(3, "Operating Systems", "Galvin")

};

System.out.println("Linear Search index: " + linearSearch(sortedBooks, "Java Programming"));

System.out.println("Binary Search index: " + binarySearch(sortedBooks, "Java Programming"));

}

}

Output:- Linear Search index: 1

Binary Search index: 1

**3. Use Binary Search for large, sorted datasets; Linear Search for small or unsorted ones.**

**Exercise 7: Financial Forecasting**

**1. Recursive Algorithms:**

* Recursion simplifies repeated computations.
* Base case + recursive call = clean logic.

public class Forecast {

public static double predictGrowth(double baseValue, double rate, int years) {

if (years == 0) return baseValue;

return predictGrowth(baseValue \* (1 + rate), rate, years - 1);

}

public static void main(String[] args) {

double predicted = predictGrowth(10000, 0.10, 3);

System.out.println("Predicted value after 3 years: " + predicted);

}

}

Output:- Predicted value after 3 years: 13310.0

**3. Time Complexity:**

* O(n) where n = years.
* Can optimize using memoization or iteration to avoid stack overflow in deeper recursion.