**WEEK1**

**Algorithms Data Structures**

**NAME : THILAGAR S**

**SUPERSET ID : 6420519**

**Exercise 1: Inventory Management System**

Warehouses deals with thousands of products in that kind of situation efficient search, update and deletion plays a crucial role. Poor data structures leads to slow performance**.**

**Suitable Data Structures:**

**ArrayList** : Good for sequential storage but search and update is slow it takes O(n) Time Complexity

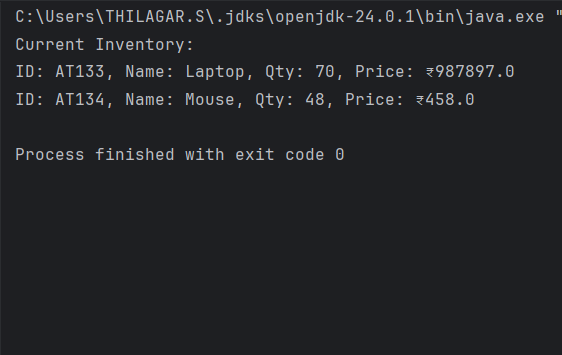
**HashMap :** It is used for fast access it takes O(n) Time Complexity

**TreeMap :** Maintain sorted order by key. Useful where sorted order is required. It takes O(log n) Time Complexity**.**

**CODE:**

package InventoryManagementSystem;  
import java.util.\*;  
public class InventoryManagementSystemClass {  
 public static void main(String[] args) {  
 Inventory inventory = new Inventory();  
 Product p1 = new Product("AT133", "Laptop", 3, 599999);  
 Product p2 = new Product("AT134", "Mouse", 48, 458);  
 Product p3 = new Product("AT135", "Keyboard", 20, 987);  
 inventory.addProduct(p1);  
 inventory.addProduct(p2);  
 inventory.addProduct(p3);  
 inventory.updateProduct("AT133", 70, 987897);  
 inventory.deleteProduct("AT135");  
 inventory.displayInventory();  
 }  
}  
class Product {  
 String productId;  
 String productName;  
 int quantity;  
 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 void display() {  
 System.*out*.println("ID: " + productId + ", Name: " + productName +  
 ", Qty: " + quantity + ", Price: ₹" + price);  
 }  
}  
class Inventory {  
 private Map<String, Product> products = new HashMap<>();  
 public void addProduct(Product product) {  
 products.put(product.productId, product);  
 }  
 public void updateProduct(String productId, int newQty, double newPrice) {  
 if (products.containsKey(productId)) {  
 Product p = products.get(productId);  
 p.quantity = newQty;  
 p.price = newPrice;  
 }  
 }  
 public void deleteProduct(String productId) {  
 products.remove(productId);  
 }  
 public void displayInventory() {  
 System.*out*.println("Current Inventory:");  
 for (Product p : products.values()) {  
 p.display();  
 }  
 }  
}

**OUTPUT:**



**Time Complexity Analysis:**

**Add Product – O(1)** Constant time for insertion by key.

**Update Product – O(1)** Direct update of details by key.

**Delete Product – O(1)** Constant time for deleting a product by accessing through key.

**Display All – O(n)** Must iterate through all products**.**

**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.

**Search operations:**

There are two types of searching methods  
 i) Linear Search

ii) Binary Search

**i)Linear Search:**

Best Case : O(1)

Average Case : O(n)

Worst Case : O(n)

**ii)Binary Search:**

Best Case : O(1)

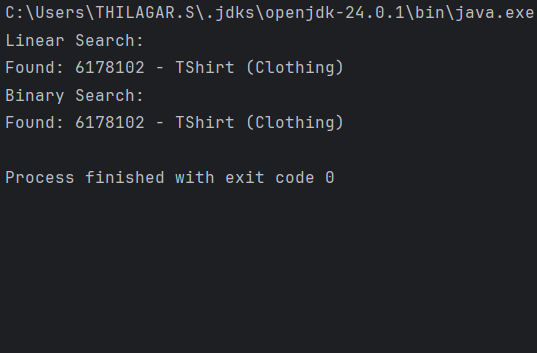
Average Case : O(log n)

Worst Case : O(log n)

**CODE :**

package ECommerceSearchExample;  
import java.util.Arrays;  
import java.util.Comparator;  
public class ECommerceSearchExampleClass {  
 public static void main(String[] args) {  
 Product[] products = {  
 new Product(6178101, "Computer", "Electronics"),  
 new Product(6178102, "TShirt", "Clothing"),  
 new Product(6178103, "Shoes", "Footwear"),  
 new Product(6178104, "Mouse", "Electronics"),  
 new Product(6178105, "Watch", "Accessories")  
 };  
 System.*out*.println("Linear Search:");  
 *linearSearch*(products, "TShirt");  
 Arrays.*sort*(products, Comparator.*comparing*(p -> p.productName));  
 System.*out*.println("Binary Search:");  
 *binarySearch*(products, "TShirt");  
 }  
 static void linearSearch(Product[] products, String productName) {  
 for (Product p : products) {  
 if (p.productName.equalsIgnoreCase(productName)) {  
 System.*out*.println("Found: " + p);  
 return;  
 }  
 }  
 System.*out*.println("Product not found.");  
 }  
 static void binarySearch(Product[] products, String productName) {  
 int left = 0, right = products.length - 1;  
 while (left <= right) {  
 int mid = (left + right) / 2;  
 int cmp = productName.compareToIgnoreCase(products[mid].productName);  
 if (cmp == 0) {  
 System.*out*.println("Found: " + products[mid]);  
 return;  
 } else if (cmp < 0) {  
 right = mid - 1;  
 } else {  
 left = mid + 1;  
 }  
 }  
 System.*out*.println("Product not found.");  
 }  
}  
class Product {  
 int productId;  
 String productName;  
 String category;  
 public Product(int productId, String productName, String category) {  
 this.productId = productId;  
 this.productName = productName;  
 this.category = category;  
 }  
 public String toString() {  
 return productId + " - " + productName + " (" + category + ")";  
 }  
}

**OUTPUT:**



**Linear Search:**

Time Complexity – O(n)

Space Complexity – O(1)

**Binary Search:**

Time Complexity – O(log n)

Space Complexity – O(1)

Linear Search is simple and works on unsorted data. Whereas binary search is more efficient for large datasets but requires sorted data.

For e-commerce platforms with large data sets Binary Search is better for fast performance. So, binary search is good algorithm for searching approach in an E-commerce platforms.

**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**.**

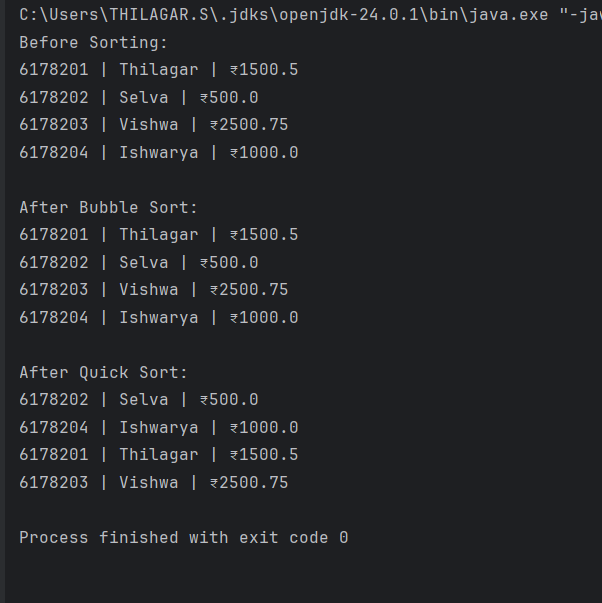
**Quick Sort :** A divide-and-conquer algorithm. Selects a pivot, partitions the array, and recursively sorts subarrays**.**

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

**CODE :**

package CustomerOrderSortingExample;  
public class CustomerOrderSortingExampleClass {  
 public static void main(String[] args) {  
 Order[] orders = {  
 new Order(6178201, "Thilagar", 1500.50),  
 new Order(6178202, "Selva", 500.00),  
 new Order(6178203, "Vishwa", 2500.75),  
 new Order(6178204, "Ishwarya", 1000.00)  
 };  
 System.*out*.println("Before Sorting:");  
 *displayOrders*(orders);  
 System.*out*.println("\nAfter Bubble Sort:");  
 *bubbleSort*(orders.clone());  
 *displayOrders*(orders);  
 System.*out*.println("\nAfter Quick Sort:");  
 Order[] quickSorted = orders.clone();  
 *quickSort*(quickSorted, 0, quickSorted.length - 1);  
 *displayOrders*(quickSorted);  
 }  
 static void bubbleSort(Order[] orders) {  
 int n = orders.length;  
 for (int i = 0; i < n - 1; i++) {  
 for (int j = 0; j < n - 1 - i; j++) {  
 if (orders[j].totalPrice > orders[j + 1].totalPrice) {  
 Order temp = orders[j];  
 orders[j] = orders[j + 1];  
 orders[j + 1] = temp;  
 }  
 }  
 }  
 }  
 static void quickSort(Order[] orders, int low, int high) {  
 if (low < high) {  
 int pivotIndex = *partition*(orders, low, high);  
 *quickSort*(orders, low, pivotIndex - 1);  
 *quickSort*(orders, pivotIndex + 1, high);  
 }  
 }  
 static int partition(Order[] orders, int low, int high) {  
 double pivot = orders[high].totalPrice;  
 int i = low - 1;  
 for (int j = low; j < high; j++) {  
 if (orders[j].totalPrice <= 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;  
 }  
 static void displayOrders(Order[] orders) {  
 for (Order o : orders) {  
 System.*out*.println(o.orderId + " | " + o.customerName + " | ₹" + o.totalPrice);  
 }  
 }  
}  
class Order {  
 int orderId;  
 String customerName;  
 double totalPrice;  
 public Order(int orderId, String customerName, double totalPrice) {  
 this.orderId = orderId;  
 this.customerName = customerName;  
 this.totalPrice = totalPrice;  
 }  
}

**OUTPUT:**



**Time Complexity and Performance:**

**Bubble Sort:**

**Time Complexity:**

i)Best Case – O(n)

ii)Average Case – O(n^2)

iii)Worst Case – O(n^2)

**Space Complexity –** O(1)

**Quick Sort:**

**Time Complexity:**

i)Best Case – O(n log n)

ii)Average Case – O(n log n)

iii)Worst Case – O(n^2)

**Space Complexity** – O(log n)

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 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).

Simple to use for fixed-size data.

Better cache locality, enhancing performance**.**

**CODE:**

package EmployeeManagementSystem;  
public class EmployeeManagementSystemClass {  
 public static void main(String[] args) {  
 EmployeeManager manager = new EmployeeManager(5);  
 manager.addEmployee(new Employee(1, "Thilagar", "Manager", 180000));  
 manager.addEmployee(new Employee(2, "Selva", "Developer", 160000));  
 manager.addEmployee(new Employee(3, "Senthil", "Tester", 150000));  
 System.*out*.println("All Employees:");  
 manager.traverseEmployees();  
 System.*out*.println("\nSearching for Employee with ID 2:");  
 manager.searchEmployeeById(2);  
 System.*out*.println("\nDeleting Employee with ID 2:");  
 manager.deleteEmployeeById(2);  
 System.*out*.println("\nAll Employees After Deletion:");  
 manager.traverseEmployees();  
 }  
}  
class Employee {  
 int employeeId;  
 String name;  
 String position;  
 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 void display() {  
 System.*out*.println(employeeId + " | " + name + " | " + position + " | ₹" + salary);  
 }  
}  
class EmployeeManager {  
 private Employee[] employees;  
 private int size;  
 public EmployeeManager(int capacity) {  
 employees = new Employee[capacity];  
 size = 0;  
 }  
 public void addEmployee(Employee emp) {  
 if (size < employees.length) {  
 employees[size++] = emp;  
 } else {  
 System.*out*.println("Cannot add: Employee array is full.");  
 }  
 }  
 public void searchEmployeeById(int id) {  
 for (int i = 0; i < size; i++) {  
 if (employees[i].employeeId == id) {  
 employees[i].display();  
 return;  
 }  
 }  
 System.*out*.println("Employee not found.");  
 }  
 public void traverseEmployees() {  
 for (int i = 0; i < size; i++) {  
 employees[i].display();  
 }  
 }  
 public void deleteEmployeeById(int id) {  
 int index = -1;  
 for (int i = 0; i < size; i++) {  
 if (employees[i].employeeId == id) {  
 index = i;  
 break;  
 }  
 }  
 if (index == -1) {  
 System.*out*.println("Employee not found.");  
 return;  
 }  
 for (int i = index; i < size - 1; i++) {  
 employees[i] = employees[i + 1];  
 }  
 employees[--size] = null;  
 }  
}

**OUTPUT:**

A screenshot of a computer program

AI-generated content may be incorrect.

**Time Complexity of Operations:**

Add-O(1)

Search- O(n)

Traverse- O(n)

Delete- O(n)

**Limitations of Arrays:**

Fixed Size cannot dynamically grow or shrink.

Inefficient Deletion/Insertion shifting elements takes time.

Wasted Space pre allocated space may go unused.

When to Use Arrays:

When the number of elements is known and fixed**.**

When fast index-based access is required.

When memory usage is predictable.

**Exercise 5: Task Management System**

**Understand Linked Lists**

**Singly Linked List:**

Each node contains data and a reference to the next node.

Traversal is possible in only one direction.

Efficient insertion/deletion at the beginning.

Used when memory efficiency is more important than bidirectional navigation.

**Doubly Linked List:**

Each node contains data, a reference to the next node, and a reference to the previous node.

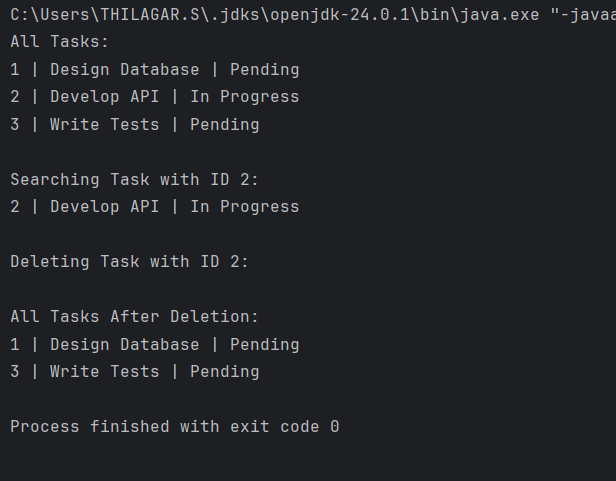
Allows traversal in both directions.

More flexible but requires extra memory per node**.**

**CODE:**

package TaskManagementSystem;  
public class TaskManagementSystemClass {  
 public static void main(String[] args) {  
 TaskList taskList = new TaskList();  
 taskList.addTask(new Task(1, "Design Database", "Pending"));  
 taskList.addTask(new Task(2, "Develop API", "In Progress"));  
 taskList.addTask(new Task(3, "Write Tests", "Pending"));  
 System.*out*.println("All Tasks:");  
 taskList.traverseTasks();  
 System.*out*.println("\nSearching Task with ID 2:");  
 taskList.searchTask(2);  
 System.*out*.println("\nDeleting Task with ID 2:");  
 taskList.deleteTask(2);  
 System.*out*.println("\nAll Tasks After Deletion:");  
 taskList.traverseTasks();  
 }  
}  
class Task {  
 int taskId;  
 String taskName;  
 String status;  
 public Task(int taskId, String taskName, String status) {  
 this.taskId = taskId;  
 this.taskName = taskName;  
 this.status = status;  
 }  
 public void display() {  
 System.*out*.println(taskId + " | " + taskName + " | " + status);  
 }  
}  
class TaskNode {  
 Task task;  
 TaskNode next;  
 public TaskNode(Task task) {  
 this.task = task;  
 this.next = null;  
 }  
}  
class TaskList {  
 private TaskNode head;  
 public void addTask(Task task) {  
 TaskNode newNode = new TaskNode(task);  
 if (head == null) {  
 head = newNode;  
 } else {  
 TaskNode temp = head;  
 while (temp.next != null) {  
 temp = temp.next;  
 }  
 temp.next = newNode;  
 }  
 }  
 public void traverseTasks() {  
 TaskNode temp = head;  
 while (temp != null) {  
 temp.task.display();  
 temp = temp.next;  
 }  
 }  
 public void searchTask(int id) {  
 TaskNode temp = head;  
 while (temp != null) {  
 if (temp.task.taskId == id) {  
 temp.task.display();  
 return;  
 }  
 temp = temp.next;  
 }  
 System.*out*.println("Task not found.");  
 }  
 public void deleteTask(int id) {  
 if (head == null) return;  
  
 if (head.task.taskId == id) {  
 head = head.next;  
 return;  
 }  
 TaskNode prev = head;  
 TaskNode curr = head.next;  
 while (curr != null) {  
 if (curr.task.taskId == id) {  
 prev.next = curr.next;  
 return;  
 }  
 prev = curr;  
 curr = curr.next;  
 }  
 System.*out*.println("Task not found.");  
 }  
}

**OUTPUT:**



**Exercise 6: Library Management System**

**Linear Search:**

Checks each element one by one until the target is found or end of list is reached.

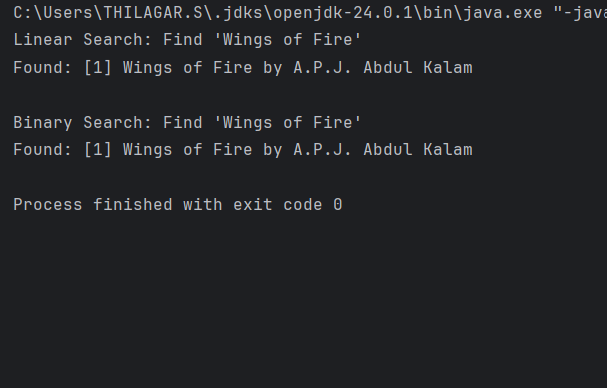
**Binary Search:**

Repeatedly divides the sorted list in half to search for the target.

**CODE:**

package LibraryManagementSystem;  
import java.util.Arrays;  
import java.util.Comparator;  
public class LibraryManagementSystemClass {  
 public static void main(String[] args) {  
 Book[] books = {  
 new Book(1, "Wings of Fire", "A.P.J. Abdul Kalam"),  
 new Book(2, "The Discovery of India", "Jawaharlal Nehru"),  
 new Book(3, "Godaan", "Munshi Premchand"),  
 new Book(4, "Train to Pakistan", "Khushwant Singh"),  
 new Book(5, "Midnight's Children", "Salman Rushdie")  
 };  
 System.*out*.println("Linear Search: Find 'Wings of Fire'");  
 *linearSearch*(books, "Wings of Fire");  
 Arrays.*sort*(books, Comparator.*comparing*(b -> b.title.toLowerCase()));  
 System.*out*.println("\nBinary Search: Find 'Wings of Fire'");  
 *binarySearch*(books, "Wings of Fire");  
 }  
 static void linearSearch(Book[] books, String title) {  
 for (Book book : books) {  
 if (book.title.equalsIgnoreCase(title)) {  
 System.*out*.println("Found: " + book);  
 return;  
 }  
 }  
 System.*out*.println("Book not found.");  
 }  
 static void binarySearch(Book[] books, String title) {  
 int left = 0, right = books.length - 1;  
 while (left <= right) {  
 int mid = (left + right) / 2;  
 int cmp = title.compareToIgnoreCase(books[mid].title);  
 if (cmp == 0) {  
 System.*out*.println("Found: " + books[mid]);  
 return;  
 } else if (cmp < 0) {  
 right = mid - 1;  
 } else {  
 left = mid + 1;  
 }  
 }  
 System.*out*.println("Book not found.");  
 }  
}  
class Book {  
 int bookId;  
 String title;  
 String author;  
  
 public Book(int bookId, String title, String author) {  
 this.bookId = bookId;  
 this.title = title;  
 this.author = author;  
 }  
 public String toString() {  
 return "[" + bookId + "] " + title + " by " + author;  
 }  
}

**OUTPUT:**



**Time Complexity:**

**i)Linear Search:**

Best Case : O(1)

Average Case : O(n)

Worst Case : O(n)

**ii)Binary Search:**

Best Case : O(1)

Average Case : O(log n)

Worst Case : O(log n)

**Use Linear Search:**

When the list is unsorted or small

When insertion/deletion happens frequently and sorting is costly

**Use Binary Search:**

When the list is already sorted

When you have a large static dataset

**Exercise 7: Financial Forecasting**

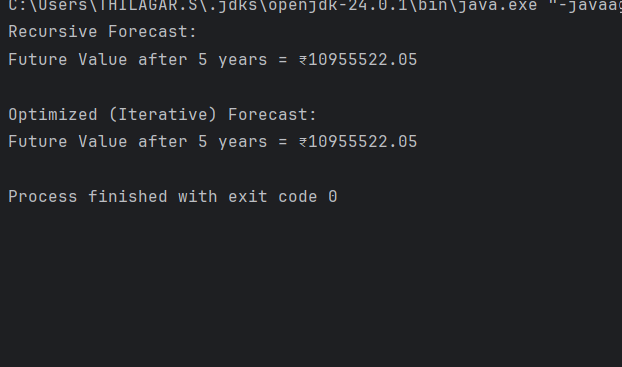
**Recursion:**

Recursion is a programming technique where a method calls itself to solve a smaller subproblem. It simplifies problems that have repetitive or self similar structure such as calculating factorial, Fibonacci numbers or compound interest over time

**CODE:**

package FinancialForecasting;  
public class FinancialForecastingClass {  
 public static void main(String[] args) {  
 double initialAmount = 1044800;  
 double growthRate = 0.6;  
 int years = 5;  
 System.*out*.println("Recursive Forecast:");  
 double result = *futureValueRecursive*(initialAmount, growthRate, years);  
 System.*out*.printf("Future Value after %d years = ₹%.2f\n", years, result);  
 System.*out*.println("\nOptimized (Iterative) Forecast:");  
 double resultIter = *futureValueIterative*(initialAmount, growthRate, years);  
 System.*out*.printf("Future Value after %d years = ₹%.2f\n", years, resultIter);  
 }  
 public static double futureValueRecursive(double amount, double rate, int years) {  
 if (years == 0) {  
 return amount;  
 }  
 return *futureValueRecursive*(amount, rate, years - 1) \* (1 + rate);  
 }  
 public static double futureValueIterative(double amount, double rate, int years) {  
 double result = amount;  
 for (int i = 1; i <= years; i++) {  
 result \*= (1 + rate);  
 }  
 return result;  
 }  
}

**OUTPUT:**



**Time Complexity – O(n)**

**Space Complexity – O(n)**

**Why Optimization is Needed?**

Although the recursion works here deeper or repeated recursion without caching may lead to

Redundant calculations.

Stack overflow for large n.

**Optimization Technique:**

Use memoization which store already computed values to avoid repeated calls.

For even better performance with low overhead, consider using iteration instead of recursion.