**Object-Oriented Programming (OOP) in Java with UML**

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects," which can contain data and code:

* Data, in the form of fields (often called attributes or properties)
* Code, in the form of methods (procedures or functions)

A key advantage of OOP is that it promotes modularity and reusability, making it easier to manage and scale complex software systems.

**Core OOP Concepts**

1. Class
   * A blueprint or template that defines the characteristics and behavior of objects.
   * It specifies the data type and methods that objects of that class will have.
   * In Java, a class is declared using the class keyword.

UML Representation

* + A rectangle divided into three sections:
    - Top section: Class name
    - Middle section: Class attributes (fields)
    - Bottom section: Class operations (methods)

class Dog {

String name; // Attribute

int age; // Attribute

void bark() { // Method

System.out.println("Woof!");

}

}

UML Diagram

1. Object
   * An instance of a class. You can create many objects from a single class.
   * Objects have specific values for the attributes defined by the class.
   * In Java, objects are created using the new keyword.
2. Dog myDog = new Dog(); // Creates an object of the Dog class
3. myDog.name = "Buddy";
4. myDog.age = 3;
5. myDog.bark(); // Calls the bark() method of the myDog object
6. Encapsulation
   * Bundling the data (attributes) and the code (methods) that operate on that data into a single unit, or "object".
   * It also involves controlling the visibility of the internal state of an object and hiding it from the outside.
   * In Java, encapsulation is achieved using access modifiers (private, protected, public).
7. class BankAccount {
8. private double balance; // Private attribute
9. public void deposit(double amount) { // Public method
10. balance += amount;
11. }
12. public double getBalance() { // Public method
13. return balance;
14. }
15. }

UML Representation

* + Attributes are preceded by '-' for private, '+' for public, and '#' for protected.

UML Diagram

1. Inheritance
   * A mechanism where a new class (subclass or derived class) inherits properties and behaviors from an existing class (superclass or base class).
   * Promotes code reuse and establishes a relationship between classes.
   * In Java, inheritance is achieved using the extends keyword.
2. class Animal { // Superclass
3. String name;
4. void eat() { System.out.println("Eating"); }
5. }
6. class Dog extends Animal { // Subclass
7. void bark() { System.out.println("Woof!"); }
8. }
9. Dog myDog = new Dog();
10. myDog.name = "Rover"; // Inherited from Animal
11. myDog.eat(); // Inherited from Animal
12. myDog.bark();

UML Representation

* + An arrow with a hollow triangle pointing from the subclass to the superclass.

UML Diagram

1. Polymorphism
   * The ability of an object to take on many forms.
   * Allows objects of different classes to be treated as objects of a common type.
   * In Java, polymorphism is achieved through method overriding and method overloading.
   * Method Overriding: A subclass provides a different implementation of a method that is already defined in its superclass.
   * Method Overloading: A class has multiple methods with the same name but different parameters.
2. class Shape {
3. double area() { return 0; }
4. }
5. class Circle extends Shape {
6. double radius;
7. double area() { return Math.PI \* radius \* radius; } // Overriding
8. }
9. class Rectangle extends Shape {
10. double length, width;
11. double area() { return length \* width; } // Overriding
12. }
13. Shape s1 = new Circle(); // Polymorphism
14. Shape s2 = new Rectangle(); // Polymorphism
15. System.out.println(s1.area()); // Calls Circle's area()
16. System.out.println(s2.area()); // Calls Rectangle's area()

UML Representation

* + Overridden methods are indicated in the subclass.

UML Diagram

1. Abstraction
   * The process of hiding complex implementation details and showing only the essential features of an object.
   * Simplifies the interface and reduces complexity.
   * In Java, abstraction is achieved using abstract classes and interfaces.
2. abstract class Vehicle {
3. abstract void start(); // Abstract method
4. void drive() { System.out.println("Driving"); }
5. }
6. class Car extends Vehicle {
7. void start() { System.out.println("Starting car"); } // Implementation
8. }
9. Vehicle v = new Car(); // Abstraction
10. v.start();
11. v.drive();

UML Representation

* + Abstract classes and methods are shown in italics.

UML Diagram

**OOP in Industry Use Cases**

* E-commerce:
  + Classes:
  + class Product {
  + String name;
  + double price;
  + String description;
  + }
  + class ShoppingCart {
  + List<Product> products;
  + void addProduct(Product product) { /\* ... \*/ }
  + double getTotalPrice() { /\* ... \*/ }
  + }
  + class Customer {
  + String name;
  + String email;
  + String address;
  + }
  + class Order {
  + Customer customer;
  + List<Product> products;
  + double totalPrice;
  + String orderStatus;
  + }
  + class Payment {
  + String paymentMethod;
  + double amount;
  + String transactionId;
  + private String creditCardNumber; // Encapsulated
  + // ...
  + }
  + Encapsulation: Hiding sensitive customer data (e.g., credit card information) within the Payment class.
  + Inheritance:
  + class Electronics extends Product {
  + String brand;
  + double warranty;
  + }
  + class Clothing extends Product {
  + String size;
  + String color;
  + }
  + Polymorphism:
  + // In ShoppingCart:
  + double getTotalPrice() {
  + double total = 0;
  + for (Product product : products) {
  + total += product.price; // Could be overridden in subclasses
  + }
  + return total;
  + }
* Banking Systems:
  + Classes:
  + class Account {
  + private double balance; // Encapsulated
  + String accountNumber;
  + Customer customer;
  + public void deposit(double amount) { /\* ... \*/ }
  + public void withdraw(double amount) { /\* ... \*/ }
  + public double getBalance() { /\* ... \*/ }
  + // ...
  + }
  + class Transaction {
  + String transactionId;
  + Date transactionDate;
  + double amount;
  + String type; // "Deposit", "Withdrawal", "Transfer"
  + }
  + class Customer {
  + String name;
  + String customerId;
  + // ...
  + }
  + class Loan {
  + String loanId;
  + double loanAmount;
  + double interestRate;
  + // ...
  + }
  + class Mortgage extends Loan {
  + String propertyAddress;
  + // ...
  + }
  + Encapsulation: Protecting account balances and transaction details within the Account class.
  + Inheritance:
  + class SavingsAccount extends Account {
  + double interestRate;
  + // ...
  + }
  + class CheckingAccount extends Account {
  + double overdraftLimit;
  + // ...
  + }
  + Polymorphism:
  + // In a transaction processing system:
  + void processTransaction(Transaction transaction, Account account) {
  + if (transaction.type.equals("Deposit")) {
  + account.deposit(transaction.amount);
  + } else if (transaction.type.equals("Withdrawal")) {
  + account.withdraw(transaction.amount); // Could be overridden
  + } // ...
  + }
* Social Media:
  + Classes:
  + class User {
  + private String profileDetails; //Encapsulated
  + String username;
  + String email;
  + List<Post> posts;
  + List<User> friends;
  + // ...
  + }
  + class Post {
  + String postId;
  + User author;
  + Date timestamp;
  + String content;
  + List<Comment> comments;
  + // ...
  + }
  + class Comment {
  + String commentId;
  + User author;
  + Date timestamp;
  + String text;
  + // ...
  + }
  + class Message {
  + String messageId;
  + User sender;
  + User receiver;
  + String text;
  + Date timestamp;
  + // ...
  + }
  + class Group {
  + String groupId;
  + String name;
  + List<User> members;
  + List<Post> groupPosts;
  + // ...
  + }
  + Encapsulation: Hiding user profile details and controlling access to them.
  + Inheritance:
  + class TextPost extends Post {
  + //...
  + }
  + class ImagePost extends Post {
  + String imageUrl;
  + }
  + class VideoPost extends Post{
  + String videoUrl;
  + }
  + Polymorphism:
  + // Displaying content
  + void displayPost(Post post) {
  + System.out.println(post.content); // Could be overridden in subclasses
  + }
* Gaming:
  + Classes:
  + abstract class GameObject {
  + int x, y;
  + int health;
  + abstract void update();
  + abstract void render();
  + // ...
  + }
  + class Player extends GameObject {
  + String name;
  + Weapon weapon;
  + void update() { /\* ... \*/ }
  + void render() { /\* ... \*/ }
  + // ...
  + }
  + class Enemy extends GameObject {
  + String enemyType;
  + void update() { /\* ... \*/ }
  + void render() { /\* ... \*/ }
  + // ...
  + }
  + class Weapon {
  + String weaponType;
  + int damage;
  + // ...
  + }
  + class Level {
  + int levelNumber;
  + List<GameObject> objects;
  + // ...
  + }
  + Encapsulation: Managing the state of game objects (e.g., position, health) internally.
  + Inheritance:
  + class Zombie extends Enemy {
  + // ...
  + }
  + class Boss extends Enemy {
  + //...
  + }
  + Polymorphism:
  + //Handling collisions
  + void handleCollision(GameObject obj1, GameObject obj2) {
  + // ...
  + }

**Abstract Classes vs. Interfaces in Java**

Here's a comparison of abstract classes and interfaces in Java:

**Abstract Class**

* A class that cannot be instantiated. It can contain both abstract (methods without implementation) and concrete (methods with implementation) methods.
* Used to define a common template or behavior for its subclasses.
* A subclass must either implement all the abstract methods of the superclass or be declared abstract itself.
* Java only supports single inheritance. A class can inherit from only one abstract class.
* Can have constructors, instance variables, and non-static methods.

**Interface**

* A collection of abstract methods. Before Java 8, it could only contain abstract methods. From Java 8 onwards, it can also contain default and static methods.
* Used to define a contract for what a class should do. Any class that implements an interface must provide implementations for all its abstract methods.
* Java supports multiple inheritance of interfaces. A class can implement multiple interfaces.
* Cannot have constructors. Variables in an interface are implicitly public, static, and final.

**Key Differences**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Abstract Class** | **Interface** |
| Instantiation | Cannot be instantiated | Cannot be instantiated |
| Inheritance | Single inheritance | Multiple inheritance |
| Methods | Can have abstract and concrete methods | Can have abstract, default, and static methods |
| Variables | Can have instance variables | Variables are implicitly public, static, and final |
| Constructors | Can have constructors | Cannot have constructors |
| Use Case | To define a common template or behavior | To define a contract for what a class should do |
| Evolution | Easier to evolve (can add concrete methods) | Adding a new abstract method breaks existing implementations |

**Example**

// Abstract Class

abstract class Shape {

String color;

public Shape(String color) {

this.color = color;

}

abstract double area(); // Abstract method

public String getColor() { // Concrete method

return color;

}

}

class Circle extends Shape {

double radius;

public Circle(String color, double radius) {

super(color);

this.radius = radius;

}

@Override

double area() {

return Math.PI \* radius \* radius;

}

}

// Interface

interface Drawable {

void draw(); // Abstract method

}

class Rectangle implements Drawable {

int width, height;

public Rectangle(int width, int height) {

this.width = width;

this.height = height;

}

@Override

public void draw() {

System.out.println("Drawing rectangle");

}

}

**Java 8 Features with Examples**

Java 8 introduced several significant features that changed how Java code is written. Here are some of the key ones:

**1. Lambda Expressions**

* Lambda expressions provide a concise way to represent anonymous functions. They enable you to write more functional-style code.
* // Before Java 8
* Thread t1 = new Thread(new Runnable() {
* @Override
* public void run() {
* System.out.println("Hello from thread t1");
* }
* });
* // With Java 8 Lambda Expressions
* Thread t2 = new Thread(() -> System.out.println("Hello from thread t2"));
* t2.start();

**2. Functional Interfaces**

* Functional interfaces are interfaces with a single abstract method (SAM). Lambda expressions can be used to provide implementations for these methods. The @FunctionalInterface annotation can be used to enforce this.
* @FunctionalInterface
* interface MyFunction {
* int apply(int x);
* }
* MyFunction square = x -> x \* x;
* int result = square.apply(5); // result is 25

**3. Default Methods in Interfaces**

* Java 8 allows interfaces to have default methods, which provide a default implementation for a method. This allows you to add new methods to an interface without breaking existing implementations.
* interface MyInterface {
* void myMethod();
* default void myDefaultMethod() {
* System.out.println("Default implementation");
* }
* }
* class MyClass implements MyInterface {
* @Override
* public void myMethod() {
* System.out.println("Implementation in MyClass");
* }
* }
* MyClass obj = new MyClass();
* obj.myMethod(); // Output: Implementation in MyClass
* obj.myDefaultMethod(); // Output: Default implementation

**4. Static Methods in Interfaces**

* Interfaces can also have static methods, which can be called directly on the interface itself.
* interface MyInterface {
* static void myStaticMethod() {
* System.out.println("Static method in interface");
* }
* }
* MyInterface.myStaticMethod(); // Output: Static method in interface

**5. Method References**

* Method references provide a shorthand for lambda expressions that call an existing method.
* // Lambda Expression
* List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
* names.forEach(name -> System.out.println(name));
* // Method Reference
* names.forEach(System.out::println);

**6. Stream API**

* The Stream API provides a powerful way to process collections of data in a functional style. It supports operations like filtering, mapping, and reducing.
* List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
* // Calculate the sum of even numbers
* int sum = numbers.stream()
* .filter(n -> n % 2 == 0)
* .mapToInt(Integer::intValue)
* .sum(); // sum is 6

**7. Optional Class**

* The Optional class is used to represent the presence or absence of a value. It helps to avoid NullPointerExceptions.
* String str = null;
* Optional<String> optional = Optional.ofNullable(str);
* if (optional.isPresent()) {
* System.out.println(optional.get());
* } else {
* System.out.println("String is null");
* }
* // Using orElse
* String result = optional.orElse("Default Value"); // result is "Default Value"

**8. Date and Time API (java.time)**

* Java 8 introduced a new Date and Time API that is more intuitive and easier to use than the old java.util.Date and java.util.Calendar classes.
* LocalDate today = LocalDate.now();
* System.out.println("Today's date: " + today);
* LocalTime now = LocalTime.now();
* System.out.println("Current time: " + now);
* LocalDateTime dateTime = LocalDateTime.now();
* System.out.println("Current date and time: " + dateTime);
* LocalDate specificDate = LocalDate.of(2024, 1, 1);
* System.out.println("Specific date: " + specificDate);
* DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
* String formattedDateTime = dateTime.format(formatter);
* System.out.println("Formatted date and time: " + formattedDateTime);

Okay, here are some examples of Data Structure algorithms with Java:

* **Arrays:**
  + Sorting:
    - Bubble Sort
    - Selection Sort
    - Insertion Sort
    - Merge Sort
    - Quick Sort
  + Searching:
    - Linear Search
    - Binary Search
* **Linked Lists:**
  + Singly Linked List:
    - Insertion
    - Deletion
    - Traversal
  + Doubly Linked List:
    - Insertion
    - Deletion
    - Traversal
* **Stacks:**
  + Push
  + Pop
  + Peek
* **Queues:**
  + Enqueue
  + Dequeue
* **Trees:**
  + Binary Tree:
    - Traversal (Inorder, Preorder, Postorder)
    - Search
    - Insertion
    - Deletion
  + Binary Search Tree:
    - Search
    - Insertion
    - Deletion
* **Graphs:**
  + Traversal:
    - Breadth-First Search (BFS)
    - Depth-First Search (DFS)
  + Shortest Path:
    - Dijkstra's Algorithm
    - Bellman-Ford Algorithm

**Key Points and Explanations:**

* **Arrays:** Demonstrates basic array operations and initialization.
* **Sorting:** Includes implementations of Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, and Quick Sort. Each method sorts the given array in ascending order.
* **Searching:** Covers Linear Search (for unsorted arrays) and Binary Search (for sorted arrays).
* **Linked Lists:** Provides implementations for Singly and Doubly Linked Lists, including insertion, deletion, and traversal methods.
* **Stacks and Queues:** Shows basic stack and queue operations (push, pop, peek, enqueue, dequeue).
* **Trees:** Demonstrates Binary Tree traversals (inorder, preorder, postorder), search, insertion and deletion. Also includes Binary Search Tree operations.
* **Graphs:** Implements Breadth-First Search (BFS) and Depth-First Search (DFS) for graph traversal. Also includes Dijkstra's and Bellman-Ford algorithms for finding shortest paths in a graph.

import java.util.\*;

import java.lang.\*;

class Main {

public static void main(String[] args) {

// Arrays

System.out.println("Arrays Examples:");

int[] arr = {5, 2, 8, 1, 9, 4};

System.out.println("Original Array: " + Arrays.toString(arr));

// Sorting

System.out.println("\nSorting Algorithms:");

int[] arrBubble = arr.clone();

bubbleSort(arrBubble);

System.out.println("Bubble Sort: " + Arrays.toString(arrBubble));

int[] arrSelection = arr.clone();

selectionSort(arrSelection);

System.out.println("Selection Sort: " + Arrays.toString(arrSelection));

int[] arrInsertion = arr.clone();

insertionSort(arrInsertion);

System.out.println("Insertion Sort: " + Arrays.toString(arrInsertion));

int[] arrMerge = arr.clone();

mergeSort(arrMerge, 0, arrMerge.length - 1);

System.out.println("Merge Sort: " + Arrays.toString(arrMerge));

int[] arrQuick = arr.clone();

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

System.out.println("Quick Sort: " + Arrays.toString(arrQuick));

// Searching

System.out.println("\nSearching Algorithms:");

System.out.println("Linear Search for 9: " + linearSearch(arr, 9));

System.out.println("Binary Search for 9: " + binarySearch(arr, 9)); // Requires sorted array

// Linked Lists

System.out.println("\nLinked Lists Examples:");

SinglyLinkedList sll = new SinglyLinkedList();

sll.insertAtEnd(1);

sll.insertAtEnd(2);

sll.insertAtEnd(3);

sll.insertAtBeginning(0);

sll.insertAtPosition(2, 5); // Insert 5 at position 2

System.out.print("Singly Linked List: ");

sll.display();

sll.deleteAtBeginning();

sll.deleteAtEnd();

sll.deleteAtPosition(1);

System.out.print("Singly Linked List after deletions: ");

sll.display();

DoublyLinkedList dll = new DoublyLinkedList();

dll.insertAtEnd(1);

dll.insertAtEnd(2);

dll.insertAtEnd(3);

dll.insertAtBeginning(0);

dll.insertAtPosition(2, 5); // Insert 5 at position 2

System.out.print("Doubly Linked List: ");

dll.display();

dll.deleteAtBeginning();

dll.deleteAtEnd();

dll.deleteAtPosition(1);

System.out.print("Doubly Linked List after deletions: ");

dll.display();

// Stacks

System.out.println("\nStacks Examples:");

StackExample stack = new StackExample();

stack.push(1);

stack.push(2);

stack.push(3);

System.out.println("Stack Peek: " + stack.peek());

System.out.println("Stack Pop: " + stack.pop());

System.out.println("Stack Pop: " + stack.pop());

System.out.println("Stack Peek after pops: " + stack.peek());

// Queues

System.out.println("\nQueues Examples:");

QueueExample queue = new QueueExample();

queue.enqueue(1);

queue.enqueue(2);

queue.enqueue(3);

System.out.println("Queue Dequeue: " + queue.dequeue());

System.out.println("Queue Dequeue: " + queue.dequeue());

System.out.println("Queue Dequeue: " + queue.dequeue());

// Trees

System.out.println("\nBinary Tree Examples:");

BinaryTree bt = new BinaryTree();

bt.root = new Node(1);

bt.root.left = new Node(2);

bt.root.right = new Node(3);

bt.root.left.left = new Node(4);

bt.root.left.right = new Node(5);

System.out.print("Inorder Traversal: ");

bt.inorderTraversal(bt.root);

System.out.println();

System.out.print("Preorder Traversal: ");

bt.preorderTraversal(bt.root);

System.out.println();

System.out.print("Postorder Traversal: ");

bt.postorderTraversal(bt.root);

System.out.println();

System.out.println("Search for 4 in Binary Tree: " + bt.search(bt.root, 4));

System.out.println("Search for 6 in Binary Tree: " + bt.search(bt.root, 6));

BinarySearchTree bst = new BinarySearchTree();

bst.insert(5);

bst.insert(3);

bst.insert(7);

bst.insert(2);

bst.insert(4);

bst.insert(6);

bst.insert(8);

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

System.out.println("Search for 4 in BST: " + bst.search(bst.root, 4));

System.out.println("Search for 9 in BST: " + bst.search(bst.root, 9));

System.out.print("Inorder Traversal of BST: ");

bst.inorderTraversal(bst.root);

System.out.println();

bst.delete(3);

System.out.print("Inorder Traversal of BST after deleting 3: ");

bst.inorderTraversal(bst.root);

System.out.println();

// Graphs

System.out.println("\nGraphs Examples:");

Graph graph = new Graph(6); // 6 vertices

graph.addEdge(0, 1);

graph.addEdge(0, 2);

graph.addEdge(1, 3);

graph.addEdge(2, 4);

graph.addEdge(3, 5);

graph.addEdge(4, 5);

System.out.print("BFS Traversal starting from vertex 0: ");

graph.bfs(0);

System.out.println();

System.out.print("DFS Traversal starting from vertex 0: ");

graph.dfs(0);

System.out.println();

// Shortest Path Algorithms

System.out.println("\nShortest Path Algorithms (Example Graph):");

int[][] graphMatrix = {

{0, 4, 0, 0, 0, 0, 0, 8, 0},

{4, 0, 8, 0, 0, 0, 0, 11, 0},

{0, 8, 0, 7, 0, 4, 0, 0, 2},

{0, 0, 7, 0, 9, 14, 0, 0, 0},

{0, 0, 0, 9, 0, 10, 0, 0, 0},

{0, 0, 4, 14, 10, 0, 2, 0, 0},

{0, 0, 0, 0, 0, 2, 0, 1, 6},

{8, 11, 0, 0, 0, 0, 1, 0, 7},

{0, 0, 2, 0, 0, 0, 6, 7, 0}

};

Dijkstra dijkstra = new Dijkstra(graphMatrix);

int source = 0;

int[] distancesDijkstra = dijkstra.dijkstra(source);

System.out.println("Dijkstra's Algorithm - Shortest distances from source " + source + ":");

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

System.out.println("Vertex " + i + ": " + distancesDijkstra[i]);

}

BellmanFord bellmanFord = new BellmanFord(graphMatrix, graphMatrix.length);

bellmanFord.bellmanFord(source);

}

// Sorting Algorithms

// Bubble Sort

public static void bubbleSort(int[] 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] > arr[j + 1]) {

// Swap arr[j] and arr[j+1]

int temp = arr[j];

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

arr[j + 1] = temp;

}

}

}

}

// Selection Sort

public static void selectionSort(int[] arr) {

int n = arr.length;

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

int min\_idx = i;

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

if (arr[j] < arr[min\_idx])

min\_idx = j;

}

// Swap the found minimum element with the first

// element

int temp = arr[min\_idx];

arr[min\_idx] = arr[i];

arr[i] = temp;

}

}

// Insertion Sort

public static void insertionSort(int[] arr) {

int n = arr.length;

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

int key = arr[i];

int j = i - 1;

/\* Move elements of arr[0..i-1], that are

greater than key, to one position ahead

of their current position \*/

while (j >= 0 && arr[j] > key) {

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

j = j - 1;

}

arr[j + 1] = key;

}

}

// Merge Sort

public static void mergeSort(int[] arr, int l, int r) {

if (l < r) {

// Find the middle point

int m = (l + r) / 2;

// Sort first and second halves

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

// Merge the sorted halves

merge(arr, l, m, r);

}

}

public static void merge(int[] arr, int l, int m, int r) {

// Find sizes of two subarrays to be merged

int n1 = m - l + 1;

int n2 = r - m;

/\* Create temp arrays \*/

int L[] = new int[n1];

int R[] = new int[n2];

/\* Copy data to temp arrays \*/

for (int i = 0; i < n1; ++i)

L[i] = arr[l + i];

for (int j = 0; j < n2; ++j)

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

/\* Merge the temp arrays \*/

// Initial indexes of first and second subarrays

int i = 0, j = 0;

// Initial index of merged subarray array

int k = l;

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

k++;

}

/\* Copy remaining elements of L[] if any \*/

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

/\* Copy remaining elements of R[] if any \*/

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

// Quick Sort

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

if (low < high) {

/\* pi is partitioning index, arr[pi] is

now at right place \*/

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

// Recursively sort elements before

// partition and after partition

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

public static int partition(int[] arr, int low, int high) {

// pivot

int pivot = arr[high];

// Index of smaller element and

// indicates the right position of pivot found so far

int i = (low - 1);

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

// If current element is smaller than the pivot

if (arr[j] < pivot) {

i++; // increment index of smaller element

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

// swap arr[i+1] and arr[high] (or pivot)

int temp = arr[i + 1];

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

arr[high] = temp;

return (i + 1);

}

// Searching Algorithms

// Linear Search

public static int linearSearch(int[] arr, int target) {

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

if (arr[i] == target) {

return i; // Return the index if found

}

}

return -1; // Return -1 if not found

}

// Binary Search

public static int binarySearch(int[] arr, int target) {

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

while (low <= high) {

int mid = low + (high - low) / 2; // More robust against overflow

if (arr[mid] == target)

return mid;

if (arr[mid] < target)

low = mid + 1;

else

high = mid - 1;

}

return -1;

}

// Linked List Classes

// Singly Linked List

static class SinglyLinkedList {

Node head;

static class Node {

int data;

Node next;

Node(int d) {

data = d;

next = null;

}

}

// Insertion at the beginning

public void insertAtBeginning(int data) {

Node newNode = new Node(data);

newNode.next = head;

head = newNode;

}

// Insertion at the end

public void insertAtEnd(int data) {

Node newNode = new Node(data);

if (head == null) {

head = newNode;

return;

}

Node last = head;

while (last.next != null) {

last = last.next;

}

last.next = newNode;

}

// Insertion at a given position

public void insertAtPosition(int position, int data) {

if (position == 0) {

insertAtBeginning(data);

return;

}

Node newNode = new Node(data);

Node current = head;

int count = 0;

while (current != null && count < position - 1) {

current = current.next;

count++;

}

if (current == null) {

System.out.println("Position is out of bounds");

return;

}

newNode.next = current.next;

current.next = newNode;

}

// Deletion at the beginning

public void deleteAtBeginning() {

if (head == null) return;

head = head.next;

}

// Deletion at the end

public void deleteAtEnd() {

if (head == null) return;

if (head.next == null) {

head = null;

return;

}

Node secondLast = head;

while (secondLast.next.next != null) {

secondLast = secondLast.next;

}

secondLast.next = null;

}

// Deletion at a given position

public void deleteAtPosition(int position) {

if (head == null) return;

if (position == 0) {

deleteAtBeginning();

return;

}

Node current = head;

int count = 0;

while (current != null && count < position - 1) {

current = current.next;

count++;

}

if (current == null || current.next == null) {

System.out.println("Position is out of bounds");

return;

}

current.next = current.next.next;

}

// Traversal

public void display() {

Node current = head;

while (current != null) {

System.out.print(current.data + " -> ");

current = current.next;

}

System.out.println("null");

}

}

// Doubly Linked List

static class DoublyLinkedList {

Node head;

static class Node {

int data;

Node next;

Node prev;

Node(int d) {

data = d;

next = null;

prev = null;

}

}

// Insertion at the beginning

public void insertAtBeginning(int data) {

Node newNode = new Node(data);

newNode.next = head;

if (head != null) {

head.prev = newNode;

}

head = newNode;

}

// Insertion at the end

public void insertAtEnd(int data) {

Node newNode = new Node(data);

if (head == null) {

head = newNode;

return;

}

Node last = head;

while (last.next != null) {

last = last.next;

}

last.next = newNode;

newNode.prev = last;

}

// Insertion at a given position

public void insertAtPosition(int position, int data) {

if (position == 0) {

insertAtBeginning(data);

return;

}

Node newNode = new Node(data);

Node current = head;

int count = 0;

while (current != null && count < position - 1) {

current = current.next;

count++;

}

if (current == null) {

System.out.println("Position is out of bounds");

return;

}

newNode.next = current.next;

newNode.prev = current;

if (current.next != null) {

current.next.prev = newNode;

}

current.next = newNode;

}

// Deletion at the beginning

public void deleteAtBeginning() {

if (head == null) return;

head = head.next;

if (head != null) {

head.prev = null;

}

}

// Deletion at the end

public void deleteAtEnd() {

if (head == null) return;

if (head.next == null) {

head = null;

return;

}

Node last = head;

while (last.next != null) {

last = last.next;

}

Node secondLast = last.prev;

secondLast.next = null;

last.prev = null;

}

// Deletion at a given position

public void deleteAtPosition(int position) {

if (head == null) return;

if (position == 0) {

deleteAtBeginning();

return;

}

Node current = head;

int count = 0;

while (current != null && count < position) {

current = current.next;

count++;

}

if (current == null) {

System.out.println("Position is out of bounds");

return;

}

Node previous = current.prev;

Node nextNode = current.next;

if (previous != null) {

previous.next = nextNode;

}

if (nextNode != null) {

nextNode.prev = previous;

}

}

// Traversal

public void display() {

Node current = head;

while (current != null) {

System.out.print(current.data + " <-> ");

current = current.next;

}

System.out.println("null");

}

}

// Stack and Queue Classes

// Stack

static class StackExample {

int[] stack;

int top;

int capacity;

public StackExample() {

capacity = 1000; // You can change the capacity as needed.

stack = new int[capacity];

top = -1;

}

public void push(int data) {

if (top == capacity - 1) {

System.out.println("Stack Overflow");

return;

}

stack[++top] = data;

}

public int pop() {

if (top == -1) {

System.out.println("Stack Underflow");

return -1; // Or throw an exception

}

return stack[top--];

}

public int peek() {

if (top == -1) {

System.out.println("Stack is empty");

return -1; // Or throw an exception

}

return stack[top];

}

public boolean isEmpty() {

return top == -1;

}

}

// Queue

static class QueueExample {

int[] queue;

int front, rear;

int capacity;

public QueueExample() {

capacity = 1000; // You can change the capacity

queue = new int[capacity];

front = rear = -1;

}

public void enqueue(int data) {

if (rear == capacity - 1) {

System.out.println("Queue Overflow");

return;

}

if (front == -1)

front = 0;

queue[++rear] = data;

}

public int dequeue() {

if (front == -1 || front > rear) {

System.out.println("Queue Underflow");

return -1;

}

int data = queue[front++];

return data;

}

public int peek() {

if (front == -1 || front > rear) {

System.out.println("Queue is empty");

return -1;

}

return queue[front];

}

public boolean isEmpty() {

return front == -1 || front > rear;

}

}

// Tree Classes

// Binary Tree

static class BinaryTree {

Node root;

static class Node {

int data;

Node left, right;

Node(int d) {

data = d;

left = right = null;

}

}

// Inorder Traversal: Left -> Root -> Right

public void inorderTraversal(Node node) {

if (node == null)

return;

inorderTraversal(node.left);

System.out.print(node.data + " ");

inorderTraversal(node.right);

}

// Preorder Traversal: Root -> Left -> Right

public void preorderTraversal(Node node) {

if (node == null)

return;

System.out.print(node.data + " ");

preorderTraversal(node.left);

preorderTraversal(node.right);

}

// Postorder Traversal: Left -> Right -> Root

public void postorderTraversal(Node node) {

if (node == null)

return;

postorderTraversal(node.left);

postorderTraversal(node.right);

System.out.print(node.data + " ");

}

// Search in Binary Tree

public boolean search(Node node, int data) {

if (node == null)

return false;

if (node.data == data)

return true;

boolean leftSearch = search(node.left, data);

if (leftSearch)

return true;

boolean rightSearch = search(node.right, data);

return rightSearch;

}

}

// Binary Search Tree

static class BinarySearchTree {

Node root;

static class Node {

int data;

Node left, right;

Node(int d) {

data = d;

left = right =null;

}

}

// Search in Binary Search Tree

public boolean search(Node root, int data) {

if (root == null)

return false;

if (root.data == data)

return true;

if (data < root.data)

return search(root.left, data);

return search(root.right, data);

}

// Insertion in Binary Search Tree

public void insert(int data) {

root = insertRec(root, data);

}

Node insertRec(Node root, int data) {

if (root == null) {

root = new Node(data);

return root;

}

if (data < root.data)

root.left = insertRec(root.left, data);

else if (data > root.data)

root.right = insertRec(root.right, data);

return root;

}

// Deletion in Binary Search Tree

public void delete(int data) {

root = deleteRec(root, data);

}

Node deleteRec(Node root, int data) {

if (root == null)

return root;

if (data < root.data) {

root.left = deleteRec(root.left, data);

} else if (data > root.data) {

root.right = deleteRec(root.right, data);

} else {

// Node with only one child or no child

if (root.left == null)

return root.right;

else if (root.right == null)

return root.left;

// Node with two children: Get the inorder successor (smallest

// in the right subtree)

root.data = minValue(root.right);

// Delete the inorder successor

root.right = deleteRec(root.right, root.data);

}

return root;

}

int minValue(Node root) {

int minv = root.data;

while (root.left != null) {

minv = root.left.data;

root = root.left;

}

return minv;

}

// Inorder Traversal for BST (Sorted order)

public void inorderTraversal(Node node) {

if (node == null)

return;

inorderTraversal(node.left);

System.out.print(node.data + " ");

inorderTraversal(node.right);

}

}

// Graph Class

static class Graph {

private int V; // Number of vertices

private LinkedList<Integer> adj[]; // Adjacency List representation

Graph(int v) {

V = v;

adj = new LinkedList[v];

for (int i = 0; i < v; ++i)

adj[i] = new LinkedList();

}

void addEdge(int v, int w) {

adj[v].add(w);

adj[w].add(v); // For an undirected graph

}

// Breadth-First Search (BFS)

void bfs(int s) {

boolean visited[] = new boolean[V];

LinkedList<Integer> queue = new LinkedList();

visited[s] = true;

queue.add(s);

while (queue.size() != 0) {

s = queue.poll();

System.out.print(s + " ");

Iterator<Integer> i = adj[s].listIterator();

while (i.hasNext()) {

int n = i.next();

if (!visited[n]) {

visited[n] = true;

queue.add(n);

}

}

}

}

// Depth-First Search (DFS)

void dfs(int s) {

boolean visited[] = new boolean[V];

dfsUtil(s, visited);

}

void dfsUtil(int v, boolean visited[]) {

visited[v] = true;

System.out.print(v + " ");

Iterator<Integer> i = adj[v].listIterator();

while (i.hasNext()) {

int n = i.next();

if (!visited[n])

dfsUtil(n, visited);

}

}

}

// Dijkstra's Algorithm

static class Dijkstra {

private int[][] graph;

private int V;

public Dijkstra(int[][] graph) {

this.graph = graph;

this.V = graph.length;

}

public int[] dijkstra(int src) {

int[] dist = new int[V];

Boolean[] sptSet = new Boolean[V];

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

dist[i] = Integer.MAX\_VALUE;

sptSet[i] = false;

}

dist[src] = 0;

for (int count = 0; count < V - 1; count++) {

int u = minDistance(dist, sptSet);

sptSet[u] = true;

for (int v = 0; v < V; v++) {

if (!sptSet[v] && graph[u][v] != 0 &&

dist[u] != Integer.MAX\_VALUE &&

dist[u] + graph[u][v] < dist[v])

dist[v] = dist[u] + graph[u][v];

}

}

return dist;

}

private int minDistance(int[] dist, Boolean[] sptSet) {

int min = Integer.MAX\_VALUE, min\_index = -1;

for (int v = 0; v < V; v++)

if (sptSet[v] == false && dist[v] <= min) {

min = dist[v];

min\_index = v;

}

return min\_index;

}

}

// Bellman-Ford Algorithm

static class BellmanFord {

private int[][] graph;

private int V;

public BellmanFord(int[][] graph, int V) {

this.graph = graph;

this.V = V;

}

public void bellmanFord(int src) {

int[] dist = new int[V];

for (int i = 0; i < V; i++)

dist[i] = Integer.MAX\_VALUE;

dist[src] = 0;

for (int i = 1; i < V; ++i) {

for (int j = 0; j < V; ++j) {

for (int k = 0; k < V; ++k) {

if (graph[j][k] != 0 && dist[j] != Integer.MAX\_VALUE && dist[j] + graph[j][k] < dist[k])

dist[k] = dist[j] + graph[j][k];

}

}

}

for (int j = 0; j < V; ++j) {

for (int k = 0; k < V; ++k) {

if (graph[j][k] != 0 && dist[j] != Integer.MAX\_VALUE && dist[j] + graph[j][k] < dist[k]) {

System.out.println("Graph contains negative weight cycle");

return;

}

}

}

System.out.println("Bellman-Ford Algorithm - Shortest distances from source " + src + ":");

for (int i = 0; i < V; ++i)

System.out.println("Vertex " + i + ": " + dist[i]);

}

}

}