**ASSIGNMENT-10**

**P.HARSHA VARDHAN REDDY**

**192311078**

1. Create a generic method sortList that takes a list of comparable elements and sorts it. Demonstrate this method with a list of Strings and a list of Integers.

**PROGRAM:**

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

public class GenericSort {

// Generic method to sort a list of comparable elements

public static <T extends Comparable<T>> void sortList(List<T> list) {

Collections.sort(list);

}

public static void main(String[] args) {

// Demonstrate sorting a list of Strings

List<String> stringList = new ArrayList<>();

stringList.add("Orange");

stringList.add("Apple");

stringList.add("Banana");

stringList.add("Grape");

System.out.println("Before sorting (Strings): " + stringList);

sortList(stringList);

System.out.println("After sorting (Strings): " + stringList);

// Demonstrate sorting a list of Integers

List<Integer> integerList = new ArrayList<>();

integerList.add(34);

integerList.add(12);

integerList.add(56);

integerList.add(7);

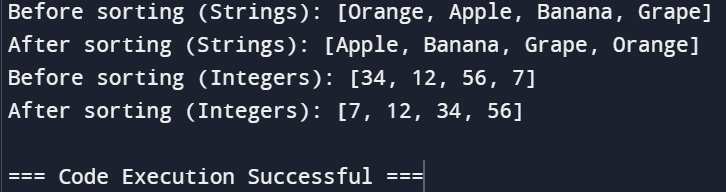
System.out.println("Before sorting (Integers): " + integerList);

sortList(integerList);

System.out.println("After sorting (Integers): " + integerList);

}

}

**OUTPUT:**  


1. Write a generic class TreeNode&lt;T&gt; representing a node in a tree with children. Implement methods to add children, traverse the tree (e.g., depth-first search), and find a node by value. Demonstrate this with a tree of Strings and Integers.

**PROGRAM:**

import java.util.ArrayList;

import java.util.List;

class TreeNode<T> {

private T value;

private List<TreeNode<T>> children;

// Constructor

public TreeNode(T value) {

this.value = value;

this.children = new ArrayList<>();

}

// Method to add a child node

public void addChild(TreeNode<T> child) {

children.add(child);

}

// Depth-First Search (DFS) traversal

public void traverse() {

System.out.println(value); // Visit the node

for (TreeNode<T> child : children) {

child.traverse(); // Recursively visit children

}

}

// Find a node by value using DFS

public TreeNode<T> findNodeByValue(T value) {

if (this.value.equals(value)) {

return this;

}

for (TreeNode<T> child : children) {

TreeNode<T> result = child.findNodeByValue(value);

if (result != null) {

return result;

}

}

return null; // Return null if the value is not found

}

public T getValue() {

return value;

}

public List<TreeNode<T>> getChildren() {

return children;

}

}

public class TreeDemo {

public static void main(String[] args) {

// Demonstrate with a tree of Strings

TreeNode<String> rootString = new TreeNode<>("Root");

TreeNode<String> childA = new TreeNode<>("Child A");

TreeNode<String> childB = new TreeNode<>("Child B");

TreeNode<String> childC = new TreeNode<>("Child C");

rootString.addChild(childA);

rootString.addChild(childB);

childA.addChild(childC);

System.out.println("String Tree Traversal:");

rootString.traverse();

System.out.println("Finding 'Child B' in String Tree:");

TreeNode<String> foundStringNode = rootString.findNodeByValue("Child B");

System.out.println(foundStringNode != null ? "Found: " + foundStringNode.getValue() : "Not Found");

// Demonstrate with a tree of Integers

TreeNode<Integer> rootInteger = new TreeNode<>(1);

TreeNode<Integer> child1 = new TreeNode<>(2);

TreeNode<Integer> child2 = new TreeNode<>(3);

TreeNode<Integer> child3 = new TreeNode<>(4);

rootInteger.addChild(child1);

rootInteger.addChild(child2);

child1.addChild(child3);

System.out.println("Integer Tree Traversal:");

rootInteger.traverse();

System.out.println("Finding '3' in Integer Tree:");

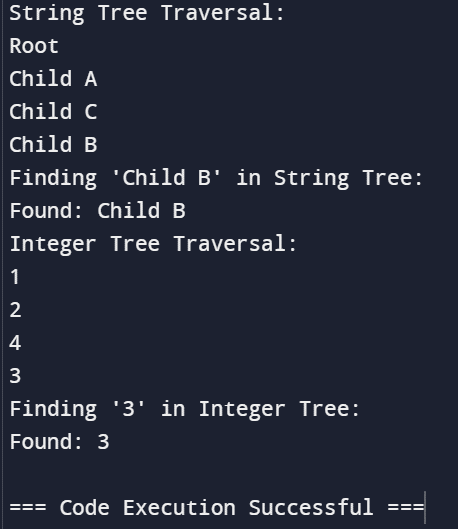
TreeNode<Integer> foundIntegerNode = rootInteger.findNodeByValue(3);

System.out.println(foundIntegerNode != null ? "Found: " + foundIntegerNode.getValue() : "Not Found");

}

}

**OUTPUT:**



1. Implement a generic class GenericPriorityQueue&lt;T extends

Comparable&lt;T&gt;&gt; with methods like enqueue, dequeue, and peek.

The elements should be dequeued in priority order. Demonstrate

with Integer and String.

**PROGRAM:**

import java.util.PriorityQueue;

class GenericPriorityQueue<T extends Comparable<T>> {

private PriorityQueue<T> queue;

// Constructor

public GenericPriorityQueue() {

this.queue = new PriorityQueue<>();

}

// Method to add an element to the queue (enqueue)

public void enqueue(T element) {

queue.offer(element);

}

// Method to remove and return the highest priority element (dequeue)

public T dequeue() {

return queue.poll(); // Returns null if the queue is empty

}

// Method to peek at the highest priority element without removing it

public T peek() {

return queue.peek(); // Returns null if the queue is empty

}

// Method to check if the queue is empty

public boolean isEmpty() {

return queue.isEmpty();

}

}

public class PriorityQueueDemo {

public static void main(String[] args) {

// Demonstrate with Integer

GenericPriorityQueue<Integer> integerQueue = new GenericPriorityQueue<>();

integerQueue.enqueue(5);

integerQueue.enqueue(1);

integerQueue.enqueue(3);

integerQueue.enqueue(2);

System.out.println("Dequeuing from Integer Queue:");

while (!integerQueue.isEmpty()) {

System.out.println(integerQueue.dequeue());

}

// Demonstrate with String

GenericPriorityQueue<String> stringQueue = new GenericPriorityQueue<>();

stringQueue.enqueue("Banana");

stringQueue.enqueue("Apple");

stringQueue.enqueue("Cherry");

stringQueue.enqueue("Date");

System.out.println("Dequeuing from String Queue:");

while (!stringQueue.isEmpty()) {

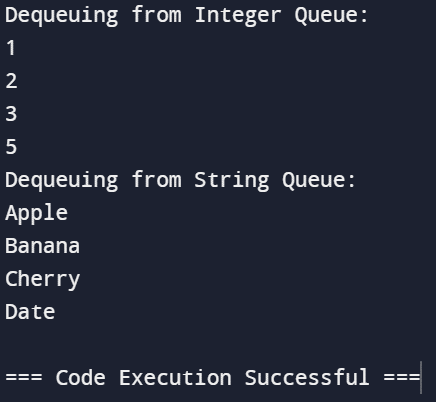
System.out.println(stringQueue.dequeue());

}

}

}

**OUTPUT:**



1. Design a generic class Graph&lt;T&gt; with methods for adding nodes,

adding edges, and performing graph traversals (e.g., BFS and DFS).

Ensure that the graph can handle both directed and undirected

graphs. Demonstrate with a graph of String nodes and another graph

of Integer nodes.

**PROGRAM:**

import java.util.\*;

class Graph<T> {

private Map<T, List<T>> adjacencyList;

private boolean isDirected;

// Constructor to initialize the graph type (directed or undirected)

public Graph(boolean isDirected) {

this.adjacencyList = new HashMap<>();

this.isDirected = isDirected;

}

// Method to add a node

public void addNode(T node) {

adjacencyList.putIfAbsent(node, new ArrayList<>());

}

// Method to add an edge

public void addEdge(T src, T dest) {

adjacencyList.get(src).add(dest);

if (!isDirected) {

adjacencyList.get(dest).add(src);

}

}

// Method to perform BFS

public void bfs(T startNode) {

Set<T> visited = new HashSet<>();

Queue<T> queue = new LinkedList<>();

queue.add(startNode);

visited.add(startNode);

while (!queue.isEmpty()) {

T currentNode = queue.poll();

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

for (T neighbor : adjacencyList.get(currentNode)) {

if (!visited.contains(neighbor)) {

queue.add(neighbor);

visited.add(neighbor);

}

}

}

System.out.println();

}

// Method to perform DFS

public void dfs(T startNode) {

Set<T> visited = new HashSet<>();

dfsHelper(startNode, visited);

System.out.println();

}

private void dfsHelper(T currentNode, Set<T> visited) {

visited.add(currentNode);

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

for (T neighbor : adjacencyList.get(currentNode)) {

if (!visited.contains(neighbor)) {

dfsHelper(neighbor, visited);

}

}

}

}

public class GraphDemo {

public static void main(String[] args) {

// Demonstrate with a graph of Strings (undirected)

Graph<String> stringGraph = new Graph<>(false);

stringGraph.addNode("A");

stringGraph.addNode("B");

stringGraph.addNode("C");

stringGraph.addNode("D");

stringGraph.addEdge("A", "B");

stringGraph.addEdge("A", "C");

stringGraph.addEdge("B", "D");

stringGraph.addEdge("C", "D");

System.out.println("BFS traversal of String Graph:");

stringGraph.bfs("A");

System.out.println("DFS traversal of String Graph:");

stringGraph.dfs("A");

// Demonstrate with a graph of Integers (directed)

Graph<Integer> integerGraph = new Graph<>(true);

integerGraph.addNode(1);

integerGraph.addNode(2);

integerGraph.addNode(3);

integerGraph.addNode(4);

integerGraph.addEdge(1, 2);

integerGraph.addEdge(1, 3);

integerGraph.addEdge(2, 4);

integerGraph.addEdge(3, 4);

System.out.println("BFS traversal of Integer Graph:");

integerGraph.bfs(1);

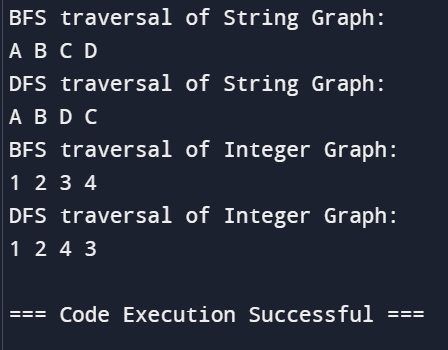
System.out.println("DFS traversal of Integer Graph:");

integerGraph.dfs(1);

}

}

**OUTPUT:**



1. Create a generic class Matrix&lt;T extends Number&gt; that represents a

matrix and supports operations like addition, subtraction, and

multiplication of matrices. Ensure that the operations are type-safe

and efficient. Demonstrate with matrices of Integer and Double.

**PROGRAM:**

import java.util.Arrays;

public class Matrix<T extends Number> {

private final int rows;

private final int cols;

private final T[][] data;

@SuppressWarnings("unchecked")

public Matrix(int rows, int cols) {

this.rows = rows;

this.cols = cols;

this.data = (T[][]) new Number[rows][cols];

}

// Method to set a value in the matrix

public void set(int row, int col, T value) {

data[row][col] = value;

}

// Method to get a value from the matrix

public T get(int row, int col) {

return data[row][col];

}

// Method to add two matrices

public Matrix<T> add(Matrix<T> other) {

checkDimensions(other);

Matrix<T> result = new Matrix<>(rows, cols);

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

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

result.set(i, j, addValues(this.get(i, j), other.get(i, j)));

}

}

return result;

}

// Method to subtract two matrices

public Matrix<T> subtract(Matrix<T> other) {

checkDimensions(other);

Matrix<T> result = new Matrix<>(rows, cols);

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

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

result.set(i, j, subtractValues(this.get(i, j), other.get(i, j)));

}

}

return result;

}

// Method to multiply two matrices

public Matrix<T> multiply(Matrix<T> other) {

if (this.cols != other.rows) {

throw new IllegalArgumentException("Matrix dimensions do not match for multiplication");

}

Matrix<T> result = new Matrix<>(this.rows, other.cols);

for (int i = 0; i < this.rows; i++) {

for (int j = 0; j < other.cols; j++) {

T sum = createZero();

for (int k = 0; k < this.cols; k++) {

sum = addValues(sum, multiplyValues(this.get(i, k), other.get(k, j)));

}

result.set(i, j, sum);

}

}

return result;

}

// Method to check if dimensions of two matrices match

private void checkDimensions(Matrix<T> other) {

if (this.rows != other.rows || this.cols != other.cols) {

throw new IllegalArgumentException("Matrix dimensions do not match");

}

}

// Method to add two Number values

@SuppressWarnings("unchecked")

private T addValues(T a, T b) {

if (a instanceof Double || b instanceof Double) {

return (T) Double.valueOf(a.doubleValue() + b.doubleValue());

} else if (a instanceof Integer || b instanceof Integer) {

return (T) Integer.valueOf(a.intValue() + b.intValue());

} else if (a instanceof Long || b instanceof Long) {

return (T) Long.valueOf(a.longValue() + b.longValue());

} else if (a instanceof Float || b instanceof Float) {

return (T) Float.valueOf(a.floatValue() + b.floatValue());

} else {

throw new UnsupportedOperationException("Unsupported number type");

}

}

// Method to subtract two Number values

@SuppressWarnings("unchecked")

private T subtractValues(T a, T b) {

if (a instanceof Double || b instanceof Double) {

return (T) Double.valueOf(a.doubleValue() - b.doubleValue());

} else if (a instanceof Integer || b instanceof Integer) {

return (T) Integer.valueOf(a.intValue() - b.intValue());

} else if (a instanceof Long || b instanceof Long) {

return (T) Long.valueOf(a.longValue() - b.longValue());

} else if (a instanceof Float || b instanceof Float) {

return (T) Float.valueOf(a.floatValue() - b.floatValue());

} else {

throw new UnsupportedOperationException("Unsupported number type");

}

}

// Method to multiply two Number values

@SuppressWarnings("unchecked")

private T multiplyValues(T a, T b) {

if (a instanceof Double || b instanceof Double) {

return (T) Double.valueOf(a.doubleValue() \* b.doubleValue());

} else if (a instanceof Integer || b instanceof Integer) {

return (T) Integer.valueOf(a.intValue() \* b.intValue());

} else if (a instanceof Long || b instanceof Long) {

return (T) Long.valueOf(a.longValue() \* b.longValue());

} else if (a instanceof Float || b instanceof Float) {

return (T) Float.valueOf(a.floatValue() \* b.floatValue());

} else {

throw new UnsupportedOperationException("Unsupported number type");

}

}

// Method to create zero value of type T

@SuppressWarnings("unchecked")

private T createZero() {

if (data[0][0] instanceof Double) {

return (T) Double.valueOf(0);

} else if (data[0][0] instanceof Integer) {

return (T) Integer.valueOf(0);

} else if (data[0][0] instanceof Long) {

return (T) Long.valueOf(0);

} else if (data[0][0] instanceof Float) {

return (T) Float.valueOf(0);

} else {

throw new UnsupportedOperationException("Unsupported number type");

}

}

// Method to print the matrix

public void printMatrix() {

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

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

System.out.print(data[i][j] + " ");

}

System.out.println();

}

}

public static void main(String[] args) {

// Demonstrate with Integer matrices

Matrix<Integer> intMatrix1 = new Matrix<>(2, 2);

Matrix<Integer> intMatrix2 = new Matrix<>(2, 2);

intMatrix1.set(0, 0, 1);

intMatrix1.set(0, 1, 2);

intMatrix1.set(1, 0, 3);

intMatrix1.set(1, 1, 4);

intMatrix2.set(0, 0, 5);

intMatrix2.set(0, 1, 6);

intMatrix2.set(1, 0, 7);

intMatrix2.set(1, 1, 8);

System.out.println("Integer Matrix 1:");

intMatrix1.printMatrix();

System.out.println("Integer Matrix 2:");

intMatrix2.printMatrix();

System.out.println("Addition of Integer Matrices:");

Matrix<Integer> intAddResult = intMatrix1.add(intMatrix2);

intAddResult.printMatrix();

System.out.println("Subtraction of Integer Matrices:");

Matrix<Integer> intSubResult = intMatrix1.subtract(intMatrix2);

intSubResult.printMatrix();

System.out.println("Multiplication of Integer Matrices:");

Matrix<Integer> intMulResult = intMatrix1.multiply(intMatrix2);

intMulResult.printMatrix();

// Demonstrate with Double matrices

Matrix<Double> doubleMatrix1 = new Matrix<>(2, 2);

Matrix<Double> doubleMatrix2 = new Matrix<>(2, 2);

doubleMatrix1.set(0, 0, 1.5);

doubleMatrix1.set(0, 1, 2.5);

doubleMatrix1.set(1, 0, 3.5);

doubleMatrix1.set(1, 1, 4.5);

doubleMatrix2.set(0, 0, 5.5);

doubleMatrix2.set(0, 1, 6.5);

doubleMatrix2.set(1, 0, 7.5);

doubleMatrix2.set(1, 1, 8.5);

System.out.println("Double Matrix 1:");

doubleMatrix1.printMatrix();

System.out.println("Double Matrix 2:");

doubleMatrix2.printMatrix();

System.out.println("Addition of Double Matrices:");

Matrix<Double> doubleAddResult = doubleMatrix1.add(doubleMatrix2);

doubleAddResult.printMatrix();

System.out.println("Subtraction of Double Matrices:");

Matrix<Double> doubleSubResult = doubleMatrix1.subtract(doubleMatrix2);

doubleSubResult.printMatrix();

System.out.println("Multiplication of Double Matrices:");

Matrix<Double> doubleMulResult = doubleMatrix1.multiply(doubleMatrix2);

doubleMulResult.printMatrix();

}

}

**OUTPUT:**

