Assignment 10

Java programming

k.vamsi

192372062

20-8-24

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.Collections;

import java.util.List;

public class GenericSorter {

// Generic method to sort a list of comparable elements

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

Collections.sort(list);

}

}

import java.util.ArrayList;

import java.util.List;

public class Main {

public static void main(String[] args) {

// List of Strings

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

stringList.add("Banana");

stringList.add("Apple");

stringList.add("Orange");

stringList.add("Mango");

// List of Integers

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

intList.add(42);

intList.add(23);

intList.add(99);

intList.add(7);

// Sorting the String list

GenericSorter.sortList(stringList);

System.out.println("Sorted String List: " + stringList);

// Sorting the Integer list

GenericSorter.sortList(intList);

System.out.println("Sorted Integer List: " + intList);

}

}

Output:

Sorted String List: [Apple, Banana, Mango, Orange]

Sorted Integer List: [7, 23, 42, 99]

2. 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;

public class TreeNode<T> {

private T value;

private List<TreeNode<T>> children;

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);

}

// Getter for value

public T getValue() {

return value;

}

// Method to traverse the tree using Depth-First Search (DFS)

public void traverse() {

traverse(this, "");

}

private void traverse(TreeNode<T> node, String indent) {

System.out.println(indent + node.getValue());

for (TreeNode<T> child : node.children) {

traverse(child, indent + " ");

}

}

// Method to find a node by value using DFS

public TreeNode<T> find(T value) {

return find(this, value);

}

private TreeNode<T> find(TreeNode<T> node, T value) {

if (node.getValue().equals(value)) {

return node;

}

for (TreeNode<T> child : node.children) {

TreeNode<T> found = find(child, value);

if (found != null) {

return found;

}

}

return null; // If the value is not found

}

}

public class Main {

public static void main(String[] args) {

// Creating a tree of Strings

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

TreeNode<String> child1String = new TreeNode<>("Child1");

TreeNode<String> child2String = new TreeNode<>("Child2");

TreeNode<String> grandchild1String = new TreeNode<>("GrandChild1");

rootString.addChild(child1String);

rootString.addChild(child2String);

child1String.addChild(grandchild1String);

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

rootString.traverse();

TreeNode<String> foundStringNode = rootString.find("Child2");

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

// Creating a tree of Integers

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

TreeNode<Integer> child1Int = new TreeNode<>(2);

TreeNode<Integer> child2Int = new TreeNode<>(3);

TreeNode<Integer> grandchild1Int = new TreeNode<>(4);

rootInt.addChild(child1Int);

rootInt.addChild(child2Int);

child1Int.addChild(grandchild1Int);

System.out.println("\nTraversing Integer Tree:");

rootInt.traverse();

TreeNode<Integer> foundIntNode = rootInt.find(3);

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

}

}

Output:

Traversing String Tree:

Root

Child1

GrandChild1

Child2

Found String Node: Child2

Traversing Integer Tree:

1

2

4

3

Found Integer Node: 3

3. 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;

public class GenericPriorityQueue<T extends Comparable<T>> {

private PriorityQueue<T> queue;

public GenericPriorityQueue() {

this.queue = new PriorityQueue<>();

}

// Method to add an element to the queue

public void enqueue(T element) {

queue.add(element);

}

// Method to remove and return the element with the highest priority

public T dequeue() {

return queue.poll(); // Returns and removes the head of the queue

}

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

public T peek() {

return queue.peek(); // Returns the head of the queue without removing it

}

// Method to check if the queue is empty

public boolean isEmpty() {

return queue.isEmpty();

}

}

public class Main {

public static void main(String[] args) {

// Demonstrating with Integer

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

intQueue.enqueue(5);

intQueue.enqueue(1);

intQueue.enqueue(3);

intQueue.enqueue(2);

System.out.println("Peek Integer Queue: " + intQueue.peek()); // Should print 1

while (!intQueue.isEmpty()) {

System.out.println("Dequeued from Integer Queue: " + intQueue.dequeue());

}

// Demonstrating with String

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

stringQueue.enqueue("Banana");

stringQueue.enqueue("Apple");

stringQueue.enqueue("Cherry");

stringQueue.enqueue("Date");

System.out.println("\nPeek String Queue: " + stringQueue.peek()); // Should print "Apple"

while (!stringQueue.isEmpty()) {

System.out.println("Dequeued from String Queue: " + stringQueue.dequeue());

}

}

}

Output:

Peek Integer Queue: 1

Dequeued from Integer Queue: 1

Dequeued from Integer Queue: 2

Dequeued from Integer Queue: 3

Dequeued from Integer Queue: 5

Peek String Queue: Apple

Dequeued from String Queue: Apple

Dequeued from String Queue: Banana

Dequeued from String Queue: Cherry

Dequeued from String Queue: Date

4. 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.\*;

public class Graph<T> {

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

private boolean isDirected;

// Constructor

public Graph(boolean isDirected) {

this.isDirected = isDirected;

this.adjacencyList = new HashMap<>();

}

// Method to add a node to the graph

public void addNode(T node) {

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

}

// Method to add an edge to the graph

public void addEdge(T source, T destination) {

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

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

adjacencyList.get(source).add(destination);

if (!isDirected) {

adjacencyList.get(destination).add(source);

}

}

// Method to perform Breadth-First Search (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 node = queue.poll();

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

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

if (!visited.contains(neighbor)) {

visited.add(neighbor);

queue.add(neighbor);

}

}

}

System.out.println();

}

// Method to perform Depth-First Search (DFS)

public void dfs(T startNode) {

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

dfsRecursive(startNode, visited);

System.out.println();

}

private void dfsRecursive(T node, Set<T> visited) {

visited.add(node);

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

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

if (!visited.contains(neighbor)) {

dfsRecursive(neighbor, visited);

}

}

}

}

public class Main {

public static void main(String[] args) {

// Graph with String nodes (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 for String Graph starting from A:");

stringGraph.bfs("A");

System.out.println("DFS for String Graph starting from A:");

stringGraph.dfs("A");

// Graph with Integer nodes (directed)

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

intGraph.addNode(1);

intGraph.addNode(2);

intGraph.addNode(3);

intGraph.addNode(4);

intGraph.addEdge(1, 2);

intGraph.addEdge(1, 3);

intGraph.addEdge(2, 4);

intGraph.addEdge(3, 4);

System.out.println("\nBFS for Integer Graph starting from 1:");

intGraph.bfs(1);

System.out.println("DFS for Integer Graph starting from 1:");

intGraph.dfs(1);

}

}

Output:

BFS for String Graph starting from A:

A B C D

DFS for String Graph starting from A:

A B D C

BFS for Integer Graph starting from 1:

1 2 3 4

DFS for Integer Graph starting from 1:

1 2 4 3

5. 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.lang.reflect.Array;

public class Matrix<T extends Number> {

private T[][] data;

private int rows;

private int cols;

// Constructor

@SuppressWarnings("unchecked")

public Matrix(Class<T> clazz, int rows, int cols) {

this.rows = rows;

this.cols = cols;

this.data = (T[][]) Array.newInstance(clazz, rows, cols);

}

// Method to set the value at a specific position

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

data[row][col] = value;

}

// Method to get the value at a specific position

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<>(data[0][0].getClass(), rows, cols);

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

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

result.set(i, j, addNumbers(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<>(data[0][0].getClass(), rows, cols);

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

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

result.set(i, j, subtractNumbers(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 multiplication is not possible with these dimensions.");

}

Matrix<T> result = new Matrix<>(data[0][0].getClass(), this.rows, other.cols);

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

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

T sum = zeroValue();

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

sum = addNumbers(sum, multiplyNumbers(this.get(i, k), other.get(k, j)));

}

result.set(i, j, sum);

}

}

return result;

}

// Utility methods for type-safe operations on Number

private T addNumbers(T a, T b) {

if (a instanceof Integer) {

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

} else if (a instanceof Double) {

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

} else if (a instanceof Long) {

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

} else if (a instanceof Float) {

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

} else if (a instanceof Short) {

return (T) Short.valueOf((short) (a.shortValue() + b.shortValue()));

} else if (a instanceof Byte) {

return (T) Byte.valueOf((byte) (a.byteValue() + b.byteValue()));

} else {

throw new UnsupportedOperationException("Operation not supported for this type.");

}

}

private T subtractNumbers(T a, T b) {

if (a instanceof Integer) {

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

} else if (a instanceof Double) {

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

} else if (a instanceof Long) {

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

} else if (a instanceof Float) {

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

} else if (a instanceof Short) {

return (T) Short.valueOf((short) (a.shortValue() - b.shortValue()));

} else if (a instanceof Byte) {

return (T) Byte.valueOf((byte) (a.byteValue() - b.byteValue()));

} else {

throw new UnsupportedOperationException("Operation not supported for this type.");

}

}

private T multiplyNumbers(T a, T b) {

if (a instanceof Integer) {

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

} else if (a instanceof Double) {

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

} else if (a instanceof Long) {

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

} else if (a instanceof Float) {

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

} else if (a instanceof Short) {

return (T) Short.valueOf((short) (a.shortValue() \* b.shortValue()));

} else if (a instanceof Byte) {

return (T) Byte.valueOf((byte) (a.byteValue() \* b.byteValue()));

} else {

throw new UnsupportedOperationException("Operation not supported for this type.");

}

}

// Method to return zero value for the type T

private T zeroValue() {

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

return (T) Integer.valueOf(0);

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

return (T) Double.valueOf(0.0);

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

return (T) Long.valueOf(0L);

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

return (T) Float.valueOf(0.0f);

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

return (T) Short.valueOf((short) 0);

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

return (T) Byte.valueOf((byte) 0);

} else {

throw new UnsupportedOperationException("Operation not supported for this type.");

}

}

// Method to check if the dimensions of the matrices match

private void checkDimensions(Matrix<T> other) {

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

throw new IllegalArgumentException("Matrices dimensions do not match.");

}

}

// 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();

}

System.out.println();

}

}

public class Main {

public static void main(String[] args) {

// Integer matrix demonstration

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

intMatrix1.set(0, 0, 1);

intMatrix1.set(0, 1, 2);

intMatrix1.set(1, 0, 3);

intMatrix1.set(1, 1, 4);

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

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();

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

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

intSum.printMatrix();

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

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

intDifference.printMatrix();

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

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

intProduct.printMatrix();

// Double matrix demonstration

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

doubleMatrix1.set(0, 0, 1.1);

doubleMatrix1.set(0, 1, 2.2);

doubleMatrix1.set(1, 0, 3.3);

doubleMatrix1.set(1, 1, 4.4);

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

doubleMatrix2.set(0, 0, 5.5);

doubleMatrix2.set(0, 1, 6.6

output:

Double Matrix 1:

1.1 2.2

3.3 4.4

Double Matrix 2:

5.5 6.6

7.7 8.8

Sum of Double Matrices:

6.6 8.8

11.0 13.2

Difference of Double Matrices:

-4.4 -4.4

-4.4 -4.4

Product of Double Matrices:

24.2 27.5

53.9 61.6