**ASSIGNMENT 10**

**NAME: ANJALI**

**RG NO: 192324137**

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.**

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

public class Main {

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

Collections.sort(list);

}

public static void main(String[] args) {

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

stringList.add("Z");

stringList.add("A");

stringList.add("C");

stringList.add("B");

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

sortList(stringList);

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

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

integerList.add(5);

integerList.add(2);

integerList.add(8);

integerList.add(1);

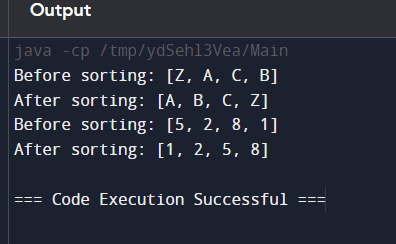
System.out.println("Before sorting: " + integerList);

sortList(integerList);

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

}

}



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.**

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<>();

}

public void addChild(TreeNode<T> child) {

this.children.add(child);

}

public void traverseDepthFirst() {

traverseDepthFirst(this);

}

private void traverseDepthFirst(TreeNode<T> node) {

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

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

traverseDepthFirst(child);

}

}

public TreeNode<T> findNodeByValue(T value) {

return findNodeByValue(this, value);

}

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

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

return node;

}

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

TreeNode<T> foundNode = findNodeByValue(child, value);

if (foundNode != null) {

return foundNode;

}

}

return null;

}

public static void main(String[] args) {

// Creating a tree of Strings

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

TreeNode<String> stringChild1 = new TreeNode<>("Child 1");

TreeNode<String> stringChild2 = new TreeNode<>("Child 2");

TreeNode<String> stringChild3 = new TreeNode<>("Child 3");

stringRoot.addChild(stringChild1);

stringRoot.addChild(stringChild2);

stringChild2.addChild(stringChild3);

System.out.println("Tree of Strings:");

stringRoot.traverseDepthFirst();

System.out.println("\nFinding node 'Child 2' in the tree of Strings:");

TreeNode<String> foundStringNode = stringRoot.findNodeByValue("Child 2");

if (foundStringNode != null) {

System.out.println("Found node: " + foundStringNode.value);

} else {

System.out.println("Node not found");

}

// Creating a tree of Integers

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

TreeNode<Integer> integerChild1 = new TreeNode<>(2);

TreeNode<Integer> integerChild2 = new TreeNode<>(3);

TreeNode<Integer> integerChild3 = new TreeNode<>(4);

integerRoot.addChild(integerChild1);

integerRoot.addChild(integerChild2);

integerChild2.addChild(integerChild3);

System.out.println("\n\nTree of Integers:");

integerRoot.traverseDepthFirst();

System.out.println("\nFinding node 3 in the tree of Integers:");

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

if (foundIntegerNode != null) {

System.out.println("Found node: " + foundIntegerNode.value);

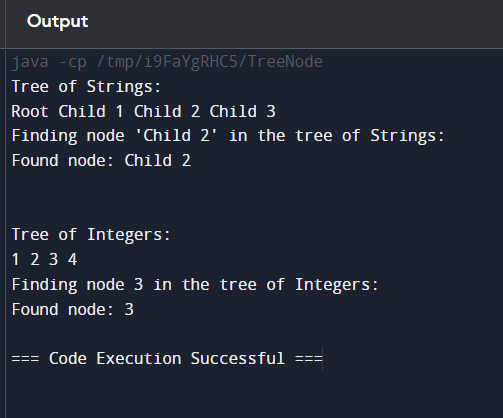
} else {

System.out.println("Node not found");

}

}

}



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.**

import java.util.PriorityQueue;

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

private PriorityQueue<T> queue;

public GenericPriorityQueue() {

this.queue = new PriorityQueue<>();

}

public void enqueue(T element) {

queue.add(element);

}

public T dequeue() {

return queue.poll();

}

public T peek() {

return queue.peek();

}

public static void main(String[] args) {

// Demonstration with Integer

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

integerPriorityQueue.enqueue(5);

integerPriorityQueue.enqueue(2);

integerPriorityQueue.enqueue(8);

integerPriorityQueue.enqueue(1);

System.out.println("Priority Queue of Integers:");

while (!integerPriorityQueue.queue.isEmpty()) {

System.out.print(integerPriorityQueue.dequeue() + " ");

}

System.out.println();

// Demonstration with String

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

stringPriorityQueue.enqueue("Z");

stringPriorityQueue.enqueue("A");

stringPriorityQueue.enqueue("C");

stringPriorityQueue.enqueue("B");

System.out.println("Priority Queue of Strings:");

while (!stringPriorityQueue.queue.isEmpty()) {

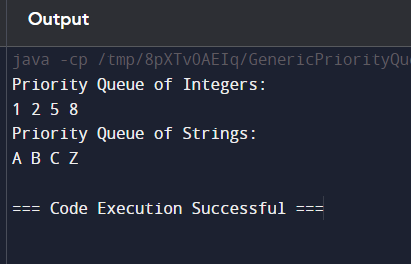
System.out.print(stringPriorityQueue.dequeue() + " ");

}

System.out.println();

}

}



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.**

import java.util.\*;

public class Graph<T> {

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

private boolean isDirected;

public Graph(boolean isDirected) {

this.adjacencyList = new HashMap<>();

this.isDirected = isDirected;

}

public void addNode(T node) {

if (!adjacencyList.containsKey(node)) {

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

}

}

public void addEdge(T source, T destination) {

if (adjacencyList.containsKey(source)) {

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

if (!isDirected) {

if (adjacencyList.containsKey(destination)) {

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

} else {

addNode(destination);

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

}

}

}

}

public void bfsTraversal(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)) {

queue.add(neighbor);

visited.add(neighbor);

}

}

}

System.out.println();

}

public void dfsTraversal(T startNode) {

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

dfsHelper(startNode, visited);

System.out.println();

}

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

visited.add(node);

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

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

if (!visited.contains(neighbor)) {

dfsHelper(neighbor, visited);

}

}

}

public static void main(String[] args) {

// Demonstration with a graph of String nodes

Graph<String> stringGraph = new Graph<>(false); // Undirected graph

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("Undirected graph of Strings:");

stringGraph.bfsTraversal("A");

stringGraph.dfsTraversal("A");

// Demonstration with a graph of Integer nodes

Graph<Integer> integerGraph = new Graph<>(true); // Directed graph

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);

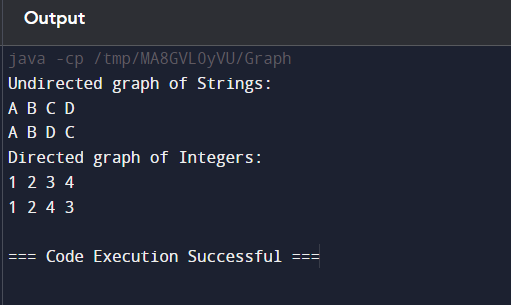
System.out.println("Directed graph of Integers:");

integerGraph.bfsTraversal(1);

integerGraph.dfsTraversal(1);

}

}



1. **Create a generic class Matrix<T extends Number> 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. simple java program**

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

class Matrix<T extends Number> {

private final int rows;

private final int cols;

private final List<List<T>> data;

public Matrix(int rows, int cols, List<List<T>> data) {

this.rows = rows;

this.cols = cols;

this.data = data;

}

// Addition of two matrices

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

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

throw new IllegalArgumentException("Matrix dimensions must match for addition.");

}

List<List<T>> result = new ArrayList<>();

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

List<T> row = new ArrayList<>();

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

row.add(addNumbers(this.data.get(i).get(j), other.data.get(i).get(j)));

}

result.add(row);

}

return new Matrix<>(rows, cols, result);

}

// Subtraction of two matrices

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

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

throw new IllegalArgumentException("Matrix dimensions must match for subtraction.");

}

List<List<T>> result = new ArrayList<>();

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

List<T> row = new ArrayList<>();

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

row.add(subtractNumbers(this.data.get(i).get(j), other.data.get(i).get(j)));

}

result.add(row);

}

return new Matrix<>(rows, cols, result);

}

// Multiplication of two matrices

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

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

throw new IllegalArgumentException("Number of columns of first matrix must equal number of rows of second matrix.");

}

List<List<T>> result = new ArrayList<>();

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

List<T> row = new ArrayList<>();

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

T sum = null;

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

T product = multiplyNumbers(this.data.get(i).get(k), other.data.get(k).get(j));

sum = sum == null ? product : addNumbers(sum, product);

}

row.add(sum);

}

result.add(row);

}

return new Matrix<>(rows, other.cols, result);

}

// Helper method to add two numbers

private T addNumbers(T a, T b) {

if (a instanceof Integer) {

return (T) (Integer) (((Integer) a) + ((Integer) b));

} else if (a instanceof Double) {

return (T) (Double) (((Double) a) + ((Double) b));

}

throw new UnsupportedOperationException("Unsupported type");

}

// Helper method to subtract two numbers

private T subtractNumbers(T a, T b) {

if (a instanceof Integer) {

return (T) (Integer) (((Integer) a) - ((Integer) b));

} else if (a instanceof Double) {

return (T) (Double) (((Double) a) - ((Double) b));

}

throw new UnsupportedOperationException("Unsupported type");

}

// Helper method to multiply two numbers

private T multiplyNumbers(T a, T b) {

if (a instanceof Integer) {

return (T) (Integer) (((Integer) a) \* ((Integer) b));

} else if (a instanceof Double) {

return (T) (Double) (((Double) a) \* ((Double) b));

}

throw new UnsupportedOperationException("Unsupported type");

}

@Override

public String toString() {

StringBuilder sb = new StringBuilder();

for (List<T> row : data) {

for (T value : row) {

sb.append(value).append(" ");

}

sb.append("\n");

}

return sb.toString();

}

}

public class Main {

public static void main(String[] args) {

// Integer matrices

Matrix<Integer> intMatrix1 = new Matrix<>(2, 2, Arrays.asList(

Arrays.asList(1, 2),

Arrays.asList(3, 4)

));

Matrix<Integer> intMatrix2 = new Matrix<>(2, 2, Arrays.asList(

Arrays.asList(5, 6),

Arrays.asList(7, 8)

));

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

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

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

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

System.out.println(intAddition);

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

System.out.println(intSubtraction);

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

System.out.println(intMultiplication);

// Double matrices

Matrix<Double> doubleMatrix1 = new Matrix<>(2, 2, Arrays.asList(

Arrays.asList(1.5, 2.5),

Arrays.asList(3.5, 4.5)

));

Matrix<Double> doubleMatrix2 = new Matrix<>(2, 2, Arrays.asList(

Arrays.asList(5.5, 6.5),

Arrays.asList(7.5, 8.5)

));

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

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

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

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

System.out.println(doubleAddition);

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

System.out.println(doubleSubtraction);

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

System.out.println(doubleMultiplication);

}

}

