import java.util.\*;

import java.util.Map.Entry;

import java.io.PrintWriter;

import java.io.File;

interface Visitor<T>

{

public void visit(T obj);

}

// --- Vertex class -----------

class Vertex<E>

{

public static final double INFINITY = Double.MAX\_VALUE;

public HashMap<E, Pair<Vertex<E>, Double> > adjList

= new HashMap<E, Pair<Vertex<E>, Double> >();

public E data;

public double dist; // used for particular graph problems, NOT the graph itself

public boolean visited;

public Vertex<E> nextInPath; // used for particular graph problems, NOT the graph itself

public Vertex( E x )

{

data = x;

dist = INFINITY;

nextInPath = null;

}

public Vertex() { this(null); }

public E getData(){ return data; }

public double getDistance(){ return dist; }

public boolean isVisited(){ return visited; }

public void visit(){ visited = true; }

public void unvisit(){ visited = false; }

public Iterator<Map.Entry<E, Pair<Vertex<E>, Double>>> iterator()

{

return adjList.entrySet().iterator();

}

public void addToAdjList(Vertex<E> neighbor, double cost)

{

if( adjList.get(neighbor.data) == null)

adjList.put(neighbor.data, new Pair<Vertex<E>, Double> (neighbor, cost) );

// Note: if you want to change the cost, you'll need to remove it and then add it back

}

public void addToAdjList(Vertex<E> neighbor, int cost)

{

addToAdjList( neighbor, (double)cost );

}

public boolean equals(Object rhs)

{

if( !( rhs instanceof Vertex<?>) )

return false;

Vertex<E> other = (Vertex<E>)rhs;

return (data.equals(other.data));

}

public int hashCode()

{

return (data.hashCode());

}

public void showAdjList()

{

Iterator<Entry<E, Pair<Vertex<E>, Double>>> iter ;

Entry<E, Pair<Vertex<E>, Double>> entry;

Pair<Vertex<E>, Double> pair;

System.out.print( "Adj List for " + data + ": ");

iter = adjList.entrySet().iterator();

while( iter.hasNext() )

{

entry = iter.next();

pair = entry.getValue();

System.out.print( pair.first.data + "("

+ String.format("%3.1f", pair.second)

+ ") " );

}

System.out.println();

}

}

//--- Graph class ------------------------------------------------------

public class Graph<E>

{

// the graph data is all here --------------------------

protected HashMap<E, Vertex<E> > vertexSet;

// public graph methods --------------------------------

public Graph ()

{

vertexSet = new HashMap<E, Vertex<E> >();

}

public Graph( Edge<E>[] edges )

{

this();

int k, numEdges;

numEdges = edges.length;

for (k = 0; k < numEdges; k++)

addEdge( edges[k].source.data,

edges[k].dest.data, edges[k].cost);

}

public void addEdge(E source, E dest, double cost)

{

Vertex<E> src, dst;

// put both source and dest into vertex list(s) if not already there

src = addToVertexSet(source);

dst = addToVertexSet(dest);

// add dest to source's adjacency list

src.addToAdjList(dst, cost);

dst.addToAdjList(src, cost); // ADD THIS IF UNDIRECTED GRAPH

}

public void addEdge(E source, E dest, int cost)

{

addEdge(source, dest, (double)cost);

}

// adds vertex with x in it, and always returns ref to it

public Vertex<E> addToVertexSet(E x)

{

Vertex<E> retVal=null;

Vertex<E> foundVertex;

// find if Vertex already in the list:

foundVertex = vertexSet.get(x);

if ( foundVertex != null ) // found it, so return it

{

return foundVertex;

}

// the vertex not there, so create one

retVal = new Vertex<E>(x);

vertexSet.put(x, retVal);

return retVal; // should never happen

}

public boolean remove(E start, E end)

{

Vertex<E> startVertex = vertexSet.get(start);

boolean removedOK = false;

if( startVertex != null )

{

Pair<Vertex<E>, Double> endPair = startVertex.adjList.remove(end);

removedOK = endPair!=null;

}

// Add if UNDIRECTED GRAPH:

Vertex<E> endVertex = vertexSet.get(end);

if( endVertex != null )

{

Pair<Vertex<E>, Double> startPair = endVertex.adjList.remove(start);

removedOK = startPair!=null ;

}

return removedOK;

}

public void showAdjTable()

{

Iterator<Entry<E, Vertex<E>>> iter;

System.out.println( "------------------------ ");

iter = vertexSet.entrySet().iterator();

while( iter.hasNext() )

{

(iter.next().getValue()).showAdjList();

}

System.out.println();

}

public void clear()

{

vertexSet.clear();

}

// reset all vertices to unvisited

public void unvisitVertices()

{

Iterator<Entry<E, Vertex<E>>> iter;

iter = vertexSet.entrySet().iterator();

while( iter.hasNext() )

{

iter.next().getValue().unvisit();

}

}

/\*\* Breadth-first traversal from the parameter startElement\*/

public void breadthFirstTraversal(E startElement, Visitor<E> visitor)

{

unvisitVertices();

Vertex<E> startVertex = vertexSet.get(startElement);

breadthFirstTraversalHelper( startVertex, visitor );

}

/\*\* Depth-first traversal from the parameter startElement \*/

public void depthFirstTraversal(E startElement, Visitor<E> visitor)

{

unvisitVertices();

Vertex<E> startVertex = vertexSet.get(startElement);

depthFirstTraversalHelper( startVertex, visitor );

}

protected void breadthFirstTraversalHelper(Vertex<E> startVertex,

Visitor<E> visitor)

{

LinkedQueue<Vertex<E>> vertexQueue = new LinkedQueue<>();

E startData = startVertex.getData();

startVertex.visit();

visitor.visit(startData);

vertexQueue.enqueue(startVertex);

while( !vertexQueue.isEmpty() )

{

Vertex<E> nextVertex = vertexQueue.dequeue();

Iterator<Map.Entry<E, Pair<Vertex<E>, Double>>> iter =

nextVertex.iterator(); // iterate adjacency list

while( iter.hasNext() )

{

Entry<E, Pair<Vertex<E>, Double>> nextEntry = iter.next();

Vertex<E> neighborVertex = nextEntry.getValue().first;

if( !neighborVertex.isVisited() )

{

vertexQueue.enqueue(neighborVertex);

neighborVertex.visit();

visitor.visit(neighborVertex.getData());

}

}

}

} // end breadthFirstTraversalHelper

public void depthFirstTraversalHelper(Vertex<E> startVertex, Visitor<E> visitor)

{

Vertex<E> nextVertex;

startVertex.visit();

visitor.visit(startVertex.data);

Iterator<Entry<E, Pair<Vertex<E>, Double>>> iter;

for(iter = startVertex.iterator(); iter.hasNext(); )

{

nextVertex = iter.next().getValue().first;

if(!nextVertex.isVisited())

depthFirstTraversalHelper(nextVertex, visitor);

}

}

public void writeToFile(PrintWriter pw)

{

pw.println("Vertices");

for(Vertex<E> v : vertexSet.values())

pw.println(v.data);

pw.println("\nAdjacency List");

for(Vertex<E> v : vertexSet.values()) {

pw.printf("(%s) ->", v.data);

for(Pair<Vertex<E>, Double> adj : v.adjList.values())

pw.printf("[%s] ", adj.first.data);

pw.println();

}

return;

}

}