```java
/*
 * HSLU / ICS/AIML : Modul ADS : Algorithmen & Datenstrukturen
 * Version: Mon Nov 18 14:30:09 CET 2024
 */

package uebung10.as.aufgabe02;


public class GraphTest {

  @SuppressWarnings("unused")
  public static void main(String[] args) {
    Graph<String, String> graph = new Graph<>();
    Vertex<String> u = graph.insertVertex("U");
    Vertex<String> v = graph.insertVertex("V");
    Vertex<String> w = graph.insertVertex("W");
    Edge<String> a = graph.insertEdge(u, v, "a");
    Edge<String> b = graph.insertEdge(v, w, "b");
    graph.print();

    if (graph.opposite(u, a) != v) {
      System.err.println("ERROR: v is not opposite of u!");
      System.exit(11);
    }
    if (!graph.areAdjacent(v, w)) {
      System.err.println("ERROR: v is not adjacent of w!");
      System.exit(22);
    }
  }
}

/* Session-Log:

Graph:
U -> (V,a)
V -> (U,a) (W,b)
W -> (V,b)

*/
```



```java
/*
 * HSLU / ICS/AIML : Modul ADS : Algorithmen & Datenstrukturen
 * Version: Mon Nov 18 14:30:09 CET 2024
 */

package uebung10.as.aufgabe02;

import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

@SuppressWarnings({"hiding", "rawtypes", "unchecked"})
public class Graph<V extends Comparable<V>, E extends Comparable<E>>
    implements GraphInterface<V, E> {

  private BackRefList<VertexImpl<V>> vertices;
  private BackRefList<EdgeImpl<E>> edges;

  Graph() {
    vertices = new BackRefList<>();
    edges = new BackRefList<>();
  }

  @Override
  public int numVertices() {
    return vertices.size();
  }

  @Override
  public int numEdges() {
    return edges.size();
  }

  @Override
  public Iterable<Vertex<V>> vertices() {
    return (List) vertices.getItems();
  }

  @Override
  public Iterable<Edge<E>> edges() {
    return (List) edges.getItems();
  }

  @Override
  public V replace(Vertex<V> v, V element) throws IllegalArgumentException {
    VertexImpl<V> vi = validate(v);
    return vi.setElement(element);
  }

  @Override
  public E replace(Edge<E> e, E element) throws IllegalArgumentException {
    EdgeImpl<E> ei = validate(e);
    return ei.setElement(element);
  }

  @Override
  public Iterable<Edge<E>> incidentEdges(Vertex<V> v)
      throws IllegalArgumentException {
    VertexImpl<V> vi = validate(v);
    return (List) vi.getIncidentEdges();
  }

  @Override
  public Vertex<V>[] endVertices(Edge<E> e) throws IllegalArgumentException {
    EdgeImpl<E> ei = validate(e);
    return ei.getEndpoints();
  }
```

```
68
69    /**
70    * @throws IllegalArgumentException if edge is not incident to vertex.
71    */
72    @Override
73    public Vertex<V> opposite(Vertex<V> v, Edge<E> e)
74        throws IllegalArgumentException {
75      validate(v);
76      validate(e);
77
78      // TODO: Implement here ...
79
80      return null;
81    }
82
83    @Override
84    public boolean areAdjacent(Vertex<V> v, Vertex<V> w)
85        throws IllegalArgumentException {
86      VertexImpl<V> vi = validate(v);
87      VertexImpl<V> wi = validate(w);
88      List<EdgeImpl<E>> incV = vi.getIncidentEdges();
89      List<EdgeImpl<E>> incW = wi.getIncidentEdges();
90
91      // TODO: Implement here ...
92
93      return false;
94    }
95
96    @Override
97    public Vertex<V> insertVertex(V element) {
98      VertexImpl<V> v = new VertexImpl<>(element);
99      Node<VertexImpl<V>> node;
100     node = vertices.insert(v);
101     v.setBackReference(node);
102     return v;
103   }
104
105   @Override
106   public Edge<E> insertEdge(Vertex<V> v, Vertex<V> w, E element)
107       throws IllegalArgumentException {
108     VertexImpl<V> vi = validate(v);
109     VertexImpl<V> wi = validate(w);
110     EdgeImpl<E> e = new EdgeImpl<>(v, w, element);
111     Node<EdgeImpl<E>> node;
112     node = edges.insert(e);
113     e.setBackReference(node);
114     node = vi.incidentList.insert(e);
115     e.backReferences[0] = node;
116     node = wi.incidentList.insert(e);
117     e.backReferences[1] = node;
118     return e;
119   }
120
121   @Override
122   public V removeVertex(Vertex<V> v) throws IllegalArgumentException {
123     VertexImpl<V> vi = validate(v);
124     List<EdgeImpl<E>> edges = vi.getIncidentEdges();
125     for (EdgeImpl<E> e : edges) {
126       removeEdge(e);
127     }
128     vertices.remove((Node<VertexImpl<V>>) vi.getBackReference());
129     vi.setBackReference(null); // mark vertex as <defunct>
130     return v.getElement();
131   }
```



```
132
133   @Override
134   public E removeEdge(Edge<E> e) throws IllegalArgumentException {
135     EdgeImpl<E> ei = validate(e);
136     Vertex<V>[] vertices = ei.getEndpoints();
137     Node<?>[] backRefs = ei.getBackReferences();
138     VertexImpl<V> v0 = (VertexImpl<V>) vertices[0];
139     v0.removeIncident(backRefs[0]);
140     VertexImpl<V> v1 = (VertexImpl<V>) vertices[1];
141     v1.removeIncident(backRefs[1]);
142     edges.remove((Node<EdgeImpl<E>>) ei.getBackReference());
143     ei.setBackReference(null); // mark edge as <defunct>
144     return e.getElement();
145   }
146
147   protected VertexImpl<V> validate(Vertex<V> v)
148       throws IllegalArgumentException {
149     VertexImpl<V> vi = (VertexImpl<V>) v;
150     if (!vi.validate(this))
151       throw new IllegalArgumentException("Invalid vertex");
152     return vi;
153   }
154
155   protected EdgeImpl<E> validate(Edge<E> e) throws IllegalArgumentException {
156     EdgeImpl<E> ei = (EdgeImpl<E>) e;
157     if (!ei.validate(this))
158       throw new IllegalArgumentException("Invalid edge");
159     return ei;
160   }
161
162   @Override
163   public String toString() {
164     StringBuilder sb = new StringBuilder();
165     for (VertexImpl<V> v : vertices.getItems()) {
166       sb.append(v.getElement() + " -> ");
167       for (EdgeImpl<E> e : v.getIncidentEdges()) {
168         Vertex<V> w = opposite(v, e);
169         sb.append("(" + w + "," + e + ") ");
170       }
171       sb.append("\n");
172     }
173     return sb.toString();
174   }
175
176   public void print() {
177     System.out.println("Graph:");
178     System.out.println(toString());
179   }
180
181   /**
182    * Prints the content of the vertex- and edge-sequences.
183    *
184    * listID:NodeNr|Item(BackRef)|prev<>next [: endpoints[2] : backrefs[2]]
185    *   listID: "v-seq" | "e-seq"
186    *   NodeNr: `listID`:"head"|`nr`|"tail"
187    *   Item: Node.item.toString()
188    *   BackRef: Back-Reference of Item
189    *   prev: Node.prev
190    *   next: Node.next
191    */
192   protected void printDiagnostic() {
193     Map<Integer, String> objIdMap = new LinkedHashMap<>();
194     fillUp(objIdMap);
195     System.out.println("v-seq:");
196     vertices.printDiagnostic("    ", objIdMap);
197     System.out.println("e-seq:");
198     edges.printDiagnostic("    ", objIdMap);
199   }
```

```
200
201   /**
202    * Traverses all lists and puts each object into the map (once!).
203    *
204    * @param objIdMap
205    *          Mapping of objects to its String-ID.
206    * @throws IllegalStateException
207    *           When an object is inserted more than once into the map.
208    */
209   protected void fillUp(Map<Integer, String> objIdMap)
210       throws IllegalStateException {
211     String listID = "v-seq";
212     Node head = vertices.getHead();
213     Node tail = vertices.getTail();
214     putInMap(head, listID + ":head", objIdMap);
215     int i = 0;
216     Node cursor = head.getNext();
217     while (cursor != tail) {
218       putInMap(cursor, listID + ":" + i, objIdMap);
219       putInMap(cursor.getItem(), cursor.getItem().getElement().toString(), objIdMap);
220       String incListID = "inc-seq";
221       VertexImpl v = (VertexImpl) cursor.getItem();
222       Node incHead = v.incidentList.getHead();
223       Node incTail = v.incidentList.getTail();
224       putInMap(incHead, incListID + ":" + i + "-head", objIdMap);
225       int j = 0;
226       Node incCursor = incHead.getNext();
227       while (incCursor != incTail) {
228         putInMap(incCursor, incListID + ":" + i + "-" + j, objIdMap);
229         incCursor = incCursor.getNext();
230         j++;
231       }
232       putInMap(incTail, incListID + ":" + i + "-tail", objIdMap);
233       cursor = cursor.getNext();
234       i++;
235     }
236     putInMap(tail, listID + ":tail", objIdMap);
237
238     listID = "e-seq";
239     head = edges.getHead();
240     tail = edges.getTail();
241     putInMap(head, listID + ":head", objIdMap);
242     i = 0;
243     cursor = head.getNext();
244     while (cursor != tail) {
245       putInMap(cursor, listID + ":" + i, objIdMap);
246       cursor = cursor.getNext();
247       i++;
248     }
249     putInMap(tail, listID + ":tail", objIdMap);
250   }
251
252   private static void putInMap(Object obj, String objId, Map<Integer, String> objIdMap
    )
253       throws IllegalStateException {
254     if (objIdMap.get(obj.hashCode()) == null) {
255       objIdMap.put(obj.hashCode(), objId);
256     } else {
257       throw new IllegalStateException(objId + ": exists already!");
258     }
259   }
```



```
260
261   class VertexImpl<V extends Comparable<V>> extends BackRefNodeItem<V>
262       implements Vertex<V> {
263
264     private BackRefList<EdgeImpl<E>> incidentList;
265
266     protected VertexImpl(V element) {
267       setElement(element);
268       incidentList = new BackRefList<>();
269     }
270
271     @Override
272     public V getElement() {
273       return super.getElement();
274     }
275
276     protected List<EdgeImpl<E>> getIncidentEdges() {
277       return incidentList.getItems();
278     }
279
280     protected void removeIncident(Node<?> backRef) {
281       incidentList.remove((Node) backRef);
282     }
283
284     protected BackRefList<EdgeImpl<E>> getIncidentList() {
285       return incidentList;
286     }
287
288     protected boolean validate(GraphInterface<V, E> graph) {
289       return (Graph.this == graph) && (getBackReference() != null);
290     }
291
292   }
```

```
293
294    class EdgeImpl<E extends Comparable<E>> extends BackRefNodeItem<E>
295        implements Edge<E> {
296
297      private Vertex<V>[] endpoints;
298      private Node<?>[] backReferences;
299
300      protected EdgeImpl(Vertex<V> v, Vertex<V> w, E element) {
301        setElement(element);
302        endpoints = new Vertex[] { v, w };
303        backReferences = new Node[2];
304      }
305
306      @Override
307      public E getElement() {
308        return super.getElement();
309      }
310
311      protected Vertex<V>[] getEndpoints() {
312        return endpoints;
313      }
314
315      protected Node<?>[] getBackReferences() {
316        return backReferences;
317      }
318
319      protected boolean validate(GraphInterface<V, E> graph) {
320        return (Graph.this == graph) && (getBackReference() != null);
321      }
322
323    }
324
325  }
```