Table of Contents

Output	Page 2
GraphTester	Page 3
Graph	Page 5
Positional	Page 7
ListPosition	Page 7
MatPosition	Page 8
PositionalList	Page 9
AdjacencyListGraph	Page 12
AdjacencyMatrixGraph	Page 15

To Professor Abdollahzadeh…

I know I went too far on this implementation, but I did so because I wasn't sure exactly how to start. I just tried implemented all the graph methods specified as part of the Graph ADT in the textbook (page 618).

The core of this implementation is an object known as a **Positional**, which is an object that holds a reference to an element. The children of Positionals can keep track of their positions in various **PositionalLists**. **ListPosition** keeps track of an element in PositionalLists, while **MatPosition** just holds one reference to a node in one list.

It was a pleasure taking data structures with you, professor.

- Kyle

```
Testing the AdjacencyListGraph structure
Add 5 verticies, adding edges 5,6,7,8
Edges on vertex "1": 5 6 8
Edges on vertex "2": 5
Edges on vertex "3": 6 7
Edges on vertex "4": 7
Edges on vertex "5": 8

Removing vertex 2
Edges on vertex "1": 6 8
Edges on vertex "3": 6 7
Edges on vertex "4": 7
Edges on vertex "5": 8

Removing edge 7
Edges on vertex "1": 6 8
Edges on vertex "3": 6
Edges on vertex "4":
Edges on vertex "5": 8

Testing the AdjacencyMatrixGraph structure
Edges on vertex "0": 8
Edges on vertex "1": 8 9
Edges on vertex "2":
Edges on vertex "3": 9

Removing edge 9
Edges on vertex "0": 8
Edges on vertex "1": 8
Edges on vertex "2":
Edges on vertex "3":

Removing vertex 0
Edges on vertex "0":
Edges on vertex "1":
Edges on vertex "2":
Edges on vertex "3":

Inserting edge 10
Edges on vertex "0":
Edges on vertex "1": 10
Edges on vertex "2": 10
Edges on vertex "3":
```

GraphTester.java

```
import java.util.ArrayList;

public class GraphTester {
   public static void main(String[] args) {
      list();
      matrix();
   }

   public static void matrix() {
      AdjacencyMatrixGraph<Integer> graph = new AdjacencyMatrixGraph<>(4);

      ArrayList<Graph<Integer, Integer>.Vertex> verticies = new ArrayList<>();
      // Since you can't get the vertex references directly, this should be enough.
      graph.vertices().forEachRemaining(v → verticies.add(v));

      graph.insertEdge(verticies.get(0), verticies.get(1), 8);
      Graph<Integer, Integer>.Edge e9 = graph.insertEdge(verticies.get(3),
verticies.get(1), 9);

      System.out.println("Testing the AdjacencyMatrixGraph structure");
      System.out.println(graph);
      System.out.println("Removing edge 9");
      graph.removeEdge(e9);
      System.out.println(graph);
      System.out.println("Removing vertex 0");
      graph.removeVertex(verticies.get(0));
      System.out.println(graph);

      // Gotta clear the vertex references, since they've changed
      verticies.clear();
      graph.vertices().forEachRemaining(v → verticies.add(v));

      System.out.println("Inserting edge 10");
      graph.insertEdge(verticies.get(2), verticies.get(1), 10);
      System.out.println(graph);
   }

   public static void list() {
      AdjacencyListGraph<Integer, Integer> graph = new AdjacencyListGraph<>();

      System.out.println("Testing the AdjacencyListGraph structure");
      Graph<Integer, Integer>.Vertex v1 = graph.insertVertex(1);
      Graph<Integer, Integer>.Vertex v2 = graph.insertVertex(2);
      Graph<Integer, Integer>.Vertex v3 = graph.insertVertex(3);
      Graph<Integer, Integer>.Vertex v4 = graph.insertVertex(4);
      Graph<Integer, Integer>.Vertex v5 = graph.insertVertex(5);
      System.out.println("Add 5 verticies, adding edges 5,6,7,8");
      graph.insertEdge(v1, v2, 5);
      graph.insertEdge(v1, v3, 6);
      Graph<Integer, Integer>.Edge e7 = graph.insertEdge(v3, v4, 7);
      graph.insertEdge(v5, v1, 8);
      System.out.println(graph);
      System.out.println("Removing vertex 2");
      graph.removeVertex(v2);
```

```
      System.out.println(graph);
      System.out.println("Removing edge 7");
      graph.removeEdge(e7);
      System.out.println(graph);
   }
}
```

Graph.java

```
import java.util.Iterator;

/**
 * The Graph ADT
 *
 * @author Kyle Guarco
 */
public abstract class Graph<V, E> {

   public abstract int numVertices();
   public abstract Iterator<Vertex> vertices();
   public abstract int numEdges();
   public abstract Iterator<Edge> edges();

   public abstract Edge getEdge(Vertex uref, Vertex vref);
   public abstract int outDegree(Vertex vref);
   public abstract int inDegree(Vertex vref);
   public abstract Iterator<Edge> outgoingEdges(Vertex vref);
   public abstract Iterator<Edge> incomingEdges(Vertex vref);

   public abstract Vertex insertVertex(V data);
   public abstract Edge insertEdge(Vertex uref, Vertex vref, E data);
   public abstract void removeVertex(Vertex vref);
   public abstract void removeEdge(Edge eref);

   public Positional<Vertex>[] endVertices(Edge eref) {
      return eref.ends;
   }

   public Vertex opposite(Vertex vref, Edge eref) {
      Positional<Vertex> vpos = vertexpos(vref);

      return eref.ends[0] == vpos ? eref.ends[1].element : vref;
   }

   /** Sets the positional {@code posref} for this vertex. */
   protected void setvertexpos(Vertex vref, Positional<Vertex> posref) {
      vref.position = posref;
   }

   /** @return The positional related to vertex {@code vref} */
   @SuppressWarnings("unchecked")
   protected <P extends Positional<Vertex>> P vertexpos(Vertex vref) {
      return (P) vref.position;
   }

   /** Sets the positional {@code posref} for this edge. */
   protected void setedgepos(Edge eref, Positional<Edge> posref) {
      eref.position = posref;
   }


   /** @return The positional related to edge {@code eref} */
   @SuppressWarnings("unchecked")
```

```
  protected <P extends Positional<Edge>> P edgepos(Edge eref) {
     return (P) eref.position;
  }

  /**
   * A general vertex on a graph.
   */
  class Vertex {
     protected Positional<Vertex> position;

     public V data;

     public Vertex(V data) {
        this.data = data;
     }

     public Vertex() {
        this(null);
     }

     @Override
     public String toString() {
        return data.toString();
     }
  }

  /**
   * A general edge on a graph.
   */
  @SuppressWarnings("unchecked")
  class Edge {
     protected Positional<Edge> position;

     public E data;
     public final Positional<Vertex>[] ends;

     public Edge(Positional<Vertex> uref, Positional<Vertex> vref, E data) {
        this.ends = new Positional[] {uref, vref};
        this.data = data;
     }

     public Edge(Positional<Vertex> uref, Positional<Vertex> vref) {
        this(uref, vref, null);
     }

     @Override
     public String toString() {
        return data.toString();
     }
  }
}
```

Positional.java

```
/**
 * Abstraction of a position on a graph.
 *
 * @author Kyle Guarco
 */
public class Positional<T> {
   public T element;

   public Positional(T element) {
      this.element = element;
   }
}
```

ListPosition.java

```
import java.util.HashMap;
import java.util.Iterator;

/**
 * Describes a position on a graph. Can keep track of its position in any
 * positional linked list (see {@code PossitionalList<E>}).
 *
 * @author Kyle Guarco
 */
public class ListPosition<E> extends Positional<E> implements
Iterable<PositionalList<E>.Node>{
   private HashMap<PositionalList<E>, PositionalList<E>.Node> connections;

   public ListPosition(E element) {
      super(element);
      this.connections = new HashMap<>();
   }

   public ListPosition() {
      this(null);
   }

   /**
    * Iterates over all the connected nodes for this position.
    * @return The nodes this positional is connected to.
    */
   @Override
   public Iterator<PositionalList<E>.Node> iterator() {
      return connections.values().iterator();
   }

   /**
    * @param list
    * @return The connection this positional has to the {@code list}
    */
   public PositionalList<E>.Node get(PositionalList<E> list) {
      return connections.get(list);
   }
```

```
   /**
    * Connects this positional to a positional linked list.
    * @param list
    * @param node The node to relate the positional to.
    */
   public void connect(PositionalList<E> list, PositionalList<E>.Node node) {
      connections.put(list, node);
   }

   /**
    * Disconnects this positional from a postional linked list.
    * @param list
    */
   public void disconnect(PositionalList<E> list) {
      connections.remove(list);
   }
}
```

MatPosition.java

```
/**
 * Describes a positional that holds one node reference. Note that the
 * MatPosition should be used ONLY IF the node in question is used in one list.
 */
public class MatPosition<T> extends Positional<T> {
   private PositionalList<T>.Node noderef;

   public MatPosition(T element) {
      super(element);
   }

   public void setref(PositionalList<T>.Node noderef) {
      this.noderef = noderef;
   }

   public PositionalList<T>.Node getref() {
      return noderef;
   }
}
```

PositionalList.java

```
import java.util.Iterator;

/**
 * Keeps track of a list of positionals in a linked list. It's important to note
 * that positionals are different from a node, in that positionals can be related to
 * a node in a positional linked list. The positional IS NOT a node in a linked list,
 * but an element of it.
 */
public class PositionalList<E> implements Iterable<E> {
   private Node head, tail;
   private int size;

   public PositionalList() {
      this.size = 0;
   }

   /**
    * @return An iterable of all the nodes in the list.
    */
   @Override
   public Iterator<E> iterator() {
      return new Iterator<E>() {

         private Node current = head;

         @Override
         public boolean hasNext() {
            return current ≠ null;
         }

         @Override
         public E next() {
            E element = current.pos.element;
            current = current.next;
            return element;
         }
      };
   }

   /**
    * Adds a positional node to the linked list. It's important to note that the
    * relation of the positional to this list isn't done automagically, and requires
    * a manual call to {@code Position.connect()}. This ensures that the connections
    * to each list aren't "opaque" (not hidden from the user).
    * @param posref A reference to a positional.
    * @return The node in this list representing {@code posref}
    */
   public Node add(Positional<E> posref) {
      Node newnode = new Node(posref);
      size++; // Increment here, since both outcomes of the function add a node.

      if (head == null) {
         head = newnode;
         tail = head;
```

```
      return newnode;
   }

   tail.next = newnode;
   tail.next.prev = tail;
   tail = tail.next;
   return newnode;
}

/**
 * Removes a node from the linked list. The node is garbage collected when
 * this function returns.
 * @param noderef A reference to a node.
 * @return Did the removal succeed?
 */
public boolean remove(Node noderef) {
   if (noderef == null)
      return false;

   Node prevref = noderef.prev;
   boolean hasPrev = (prevref != null);
   Node nextref = noderef.next;
   boolean hasNext = (nextref != null);

   if (hasPrev)
      prevref.next = nextref;

   if (hasNext)
      nextref.prev = prevref;

   if (head == noderef) {
      head = null;
      if (hasNext)
         head = nextref;
      else
         tail = null;
   } else if (tail == noderef && hasPrev) {
      Node current = prevref;
      while (current.next != null)
         current = current.next;

      tail = current;
   }

   size--;
   return true;
}

public Node head() {
   return head;
}

public Node tail() {
   return tail;
}
```

```
   public E element(Node noderef) {
      return noderef.pos.element;
   }

   public Node next(Node noderef) {
      return noderef.next;
   }

   public Node previous(Node noderef) {
      return noderef.prev;
   }

   public int size() {
      return size;
   }

   /**
    * A node in a positional linked list. The member {@code pos} holds a
    * reference to a positional.
    */
   class Node {
      public Node next, prev;
      public Positional<E> pos;

      public Node(Positional<E> pos) {
         this.pos = pos;
      }

      public Node() {
         this(null);
      }
   }
}
```

AdjacencyListGraph.java

```
import java.util.HashMap;
import java.util.Iterator;

public class AdjacencyListGraph<V, E> extends Graph<V, E> {
   private PositionalList<Vertex> vertexlist;
   private PositionalList<Edge> edgelist;
   private HashMap<ListPosition<Vertex>, PositionalList<Edge>> incidence;

   public AdjacencyListGraph() {
      this.vertexlist = new PositionalList<>();
      this.edgelist = new PositionalList<>();
      this.incidence = new HashMap<>();
   }

   @Override
   public String toString() {
      StringBuilder builder = new StringBuilder();

      PositionalList<Edge> edges;
      for (Vertex vertex : vertexlist) {
         edges = incidence.get(vertexpos(vertex));

         builder.append("Edges on vertex \"" + vertex.toString() + "\": ");

         for (Edge edge : edges) {
            builder.append(edge.toString() + " ");
         }

         builder.append('\n');
      }

      return builder.toString();
   }

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

   @Override
   public Iterator<Vertex> vertices() {
      return vertexlist.iterator();
   }

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

   @Override
   public Iterator<Edge> edges() {
      return edgelist.iterator();
   }
```

```
  @Override
  public Edge getEdge(Vertex uref, Vertex vref) {
     PositionalList<Edge> uedges = incidence.get(vertexpos(uref)), vedges =
incidence.get(vertexpos(vref));
     for (Edge uedge : uedges) {
        for (Edge vedge : vedges) {
           if (uedge == vedge)
              return uedge;
        }
     }
     return null;
  }

  @Override
  public int outDegree(Vertex vref) {
     PositionalList<Edge> edges = incidence.get(vertexpos(vref));
     return edges.size();
  }

  @Override
  public int inDegree(Vertex vref) {
     return outDegree(vref);
  }

  @Override
  public Iterator<Edge> outgoingEdges(Vertex vref) {
     PositionalList<Edge> edges = incidence.get(vertexpos(vref));
     return edges.iterator();
  }

  @Override
  public Iterator<Edge> incomingEdges(Vertex vref) {
     return outgoingEdges(vref);
  }

  @Override
  public Vertex insertVertex(V data) {
     Vertex vertex = new Vertex(data);

     ListPosition<Vertex> pos = new ListPosition<>(vertex);
     pos.connect(vertexlist, vertexlist.add(pos));
     setvertexpos(vertex, pos);

     PositionalList<Edge> edges = new PositionalList<Edge>();
     incidence.put(pos, edges);

     return vertex;
  }

  @Override
  public Edge insertEdge(Vertex uref, Vertex vref, E data) {
     ListPosition<Vertex> upos = vertexpos(uref), vpos = vertexpos(vref);
     if (upos == vpos)
        return null;
```

```
      Edge edge = new Edge(upos, vpos, data);
      ListPosition<Edge> edgepos = new ListPosition<Edge>(edge);
      setedgepos(edge, edgepos);

      PositionalList<Edge> uedges = incidence.get(upos), vedges = incidence.get(vpos);

      edgepos.connect(uedges, uedges.add(edgepos));
      edgepos.connect(vedges, vedges.add(edgepos));

      return edge;
   }

   @Override
   public void removeVertex(Vertex vref) {
      ListPosition<Vertex> vertexpos = vertexpos(vref);

      PositionalList<Edge> vedges = incidence.get(vertexpos);
      for (Edge edge : vedges) {
         removeEdge(edge);
      }

      removefromlist(vertexlist, vertexpos);
      vertexpos.disconnect(vertexlist);

      incidence.remove(vertexpos);
   }

   @Override
   public void removeEdge(Edge eref) {
      ListPosition<Edge> edgepos = edgepos(eref);

      removefromlist(edgelist, edgepos);
      edgepos.disconnect(edgelist);

      PositionalList<Edge> vedges;
      for (Positional<Vertex> vertexpos : eref.ends) {
         vedges = incidence.get(vertexpos);
         removefromlist(vedges, edgepos);
         edgepos.disconnect(vedges);
      }
   }

   /**
    * Removes a positional from this linked list. The relation to this
    * linked list isn't undone automagically, and requires a manual call to
    * {@code Position.disconnect()}.
    * @param listref The list to delete from
    * @param posref A reference to a positional (this isn't deleted, only the node).
    * @return Did the removal succeed?
    */
   private <T> boolean removefromlist(PositionalList<T> listref, ListPosition<T> posref)
{
      return listref.remove(posref.get(listref));
   }
}
```

AdacencyMatrixGraph.java

```
import java.util.Iterator;

public class AdjacencyMatrixGraph<E> extends Graph<Integer, E> {
   private PositionalList<Vertex> vertexlist;
   private PositionalList<Edge> edgelist;
   private Edge[][] edges;

   @SuppressWarnings("unchecked")
   public AdjacencyMatrixGraph(int size) {
      this.vertexlist = new PositionalList<>();
      this.edgelist = new PositionalList<>();
      this.edges = new Graph.Edge[size][size];

      for (int i = 0; i < size; i++)
         insertVertex();
   }

   @Override
   public String toString() {
      StringBuilder builder = new StringBuilder();

      for (Vertex vertex : vertexlist) {
         builder.append("Edges on vertex \"" + vertex.toString() + "\": ");

         for (Edge edge : edges[vertex.data]) {
            if (edge != null)
               builder.append(edge.data.toString() + " ");
         }

         builder.append('\n');
      }

      return builder.toString();
   }

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

   @Override
   public Iterator<Vertex> vertices() {
      return vertexlist.iterator();
   }

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

   @Override
   public Iterator<Edge> edges() {
      return edgelist.iterator();
   }
```

```
@Override
public Edge getEdge(Vertex uref, Vertex vref) {
   return edges[uref.data][vref.data];
}

@Override
public int outDegree(Vertex vref) {
   int count = 0;

   for (Edge edge : edges[vref.data]) {
      if (edge ≠ null)
         count++;
   }

   return count;
}

@Override
public int inDegree(Vertex vref) {
   return outDegree(vref);
}

@Override
public Iterator<Edge> outgoingEdges(Vertex vref) {
   return new Iterator<Edge>(){

      Edge[] edgeit = edges[vref.data];
      int index = 0;

      @Override
      public boolean hasNext() {
         return index < edgeit.length;
      }

      @Override
      public Edge next() {
         return edgeit[index++];
      }
   };
}

@Override
public Iterator<Edge> incomingEdges(Vertex vref) {
   return outgoingEdges(vref);
}

@Override
public Vertex insertVertex(Integer data) {
   Vertex vertex = new Vertex(data);
   MatPosition<Vertex> vertexpos = new MatPosition<>(vertex);
   setvertexpos(vertex, vertexpos);

   vertexpos.setref(vertexlist.add(vertexpos));
```

```
    return vertex;
  }

  public Vertex insertVertex() {
    return insertVertex(vertexlist.size());
  }

  @Override
  public Edge insertEdge(Vertex uref, Vertex vref, E data) {
    MatPosition<Vertex> upos = vertexpos(uref), vpos = vertexpos(vref);

    Edge edge = new Edge(upos, vpos, data);
    MatPosition<Edge> edgepos = new MatPosition<>(edge);
    setedgepos(edge, edgepos);

    edgepos.setref(edgelist.add(edgepos));

    edges[uref.data][vref.data] = edge;
    edges[vref.data][uref.data] = edge;

    return edge;
  }

  @Override
  public void removeVertex(Vertex vref) {
    //MatPosition<Vertex> vertexpos = vertexpos(vref);

    for (int y = 0; y < vertexlist.size(); y++)
      edges[y][vref.data] = null;

    for (int x = 0; x < vertexlist.size(); x++)
      edges[vref.data][x] = null;

    //vertexlist.remove(vertexpos.getref());
    //vertexlist.forEach(v -> v.data--);
  }

  @Override
  public void removeEdge(Edge eref) {
    MatPosition<Edge> edgepos = edgepos(eref);

    Positional<Vertex> upos = eref.ends[0], vpos = eref.ends[1];

    edges[upos.element.data][vpos.element.data] = null;
    edges[vpos.element.data][upos.element.data] = null;

    edgelist.remove(edgepos.getref());
  }
}
```