Implementation of a simple graph with an edge list

Variables

vertices: set of vertices
edges: set of edges
For each vertex, we keep track of the element associated with the vertex and the degree, the in-degree and the out-degree of the vertex, that is, a 4-tuple [*element*, *degree*, *in-degree*, *out-degree*]. For each edge, we keep track of the element associated with the edge, whether the edge is directed and the end vertices of the edge, that is, a 4-tuple [*element*, *directed*?, $vertex_1$, $vertex_2$] where $vertex_1$ is the origin of the edge and $vertex_2$ is the destination if the edge is directed.
invariant: *vertices* is the set of vertices of the graph and *edges* is the set of edges of the graph

Initialization

vertices $\leftarrow \emptyset$
edges $\leftarrow \emptyset$

Algorithms

size():
  output: size of the graph
return numVertices() + numEdges()

isEmpty():
  output: graph is empty?
return size() = 0

elements():
  output: collection of elements stored in positions of graph
col ← empty collection
for each vertex *vertex* in *vertices* **do**
    add element stored in *vertex* to *col*
for each edge *edge* in *edges* **do**
    add element stored in *edge* to *col*
return *col*

positions():
  output: collection of positions of graph
col ← empty collection
for each vertex *vertex* in *vertices* **do**
    add *vertex* to *col*
for each edge *edge* in *edges* **do**
    add *edge* to *col*
return *col*

swapElements(*first*, *second*):
  postcondition: elements of *first* and *second* have been swapped
  input: positions elements of which are to be swapped
swap elements of *first* and *second*

replaceElement(*position*, *element*):
  postcondition: element at *position* in graph has been replaced with *element*
  input: *position* element of which is to be replaced with *element*
  output: replaced element
temp ← element of *position*
element of *position* ← *element*
return *temp*

```
numVertices():
   output: number of vertices of the graph
return (size of vertices)

numEdges():
   output: number of edges of the graph
return (size of edges)

vertices():
   output: collection of the vertices of the graph
col ← empty collection
for each vertex vertex in vertices do
      add vertex to col
return col

edges():
   output: collection of the edges of the graph
col ← empty collection
for each edge edge in edges do
      add edge to col
return col

aVertex():
   precondition: the graph is nonempty
   output: a vertex of the graph
vertex ← a vertex in vertices
return vertex

degree(vertex):
   input: vertex of which the degree is to be returned
   output: degree of vertex
return degree of vertex

adjacentVertices(vertex):
   input: vertex the adjacent vertices of which are returned
   output: collection of vertices adjacent to vertex
col ← empty collection
for each edge in edges do
      if vertex is an end vertex of edge then
            add other end vertex of edge to col
return col

incidentEdges(vertex):
   input: vertex whose incident edges are returned
   output: collection of edges incident on vertex
col ← empty collection
for each edge in edges do
      if vertex is an end vertex of edge then
            add edge to col
return col

endVertices(edge):
   input: edge of which the end vertices are returned
   output: end vertices of edge
return end vertices of edge

opposite(vertex, edge):
   input: vertex and edge
```

```
   output: the end vertex of edge different from vertex
   precondition: vertex is an end vertex of edge
(first, second) ← end vertices of edge
if vertex = first then
    return second
else
    return first
```

```
areAdjacent(first, second):
   input: vertices
   output: first and second are adjacent?
found ← false
for each edge in edges do
    found ← found or (first and second are the end vertices of edge)
return found
```

```
directedEdges():
   output: collection of directed edges of the graph
col ← empty collection
for each edge edge in edges do
    if edge is directed then
        add edge to col
return col
```

```
undirectedEdges():
   output: collection of undirected edges of the graph
col ← empty collection
for each edge edge in edges do
    if edge is not directed then
        add edge to col
return col
```

```
destination(edge):
   input: edge
   output: destination of edge
   precondition: edge is directed
return destination of edge
```

```
origin(edge):
   input: edge
   output: origin of edge
   precondition: edge is directed
return origin of edge
```

```
isDirected(edge):
   input: edge
   output: edge is directed?
return edge is directed?
```

```
inDegree(vertex):
   input: vertex of which the indegree is to be returned
   output: indegree of vertex
return indegree of vertex
```

```
outDegree(vertex):
   input: vertex of which the outdegree is to be returned
```

output: outdegree of *vertex*
return outdegree of *vertex*

inIncidentEdges(*vertex*):
input: vertex
output: collection of incoming edges of *vertex*
col ← empty collection
for each edge *edge* in *edges* **do**
  if *edge* is directed **then**
    if *vertex* is destination of *edge* **then**
      add *edge* to *col*
return *col*

outIncidentEdges(*vertex*):
input: vertex
output: collection of outgoing edges of *vertex*
col ← empty collection
for each edge *edge* in *edges* **do**
  if *edge* is directed **then**
    if *vertex* is origin of *edge* **then**
      add *edge* to *col*
return *col*

inAdjacentVertices(*vertex*):
input: vertex
output: collection of vertices adjacent to *vertex* along incoming edges
col ← empty collection
for each edge *edge* in *edges* **do**
  if *edge* is directed **then**
    if *vertex* is destination of *edge* **then**
      add origin of *edge* to *col*
return *col*

outAdjacentVertices(*vertex*):
input: vertex
output: collection of vertices adjacent to *vertex* along outgoing edges
col ← empty collection
for each edge *edge* in *edges* **do**
  if *edge* is directed **then**
    if *vertex* is origin of *edge* **then**
      add destination of *edge* to *col*
return *col*

insertEdge(*first*, *second*, *element*):
input: vertices and element
output: undirected edge with end vertices *first* and *second* and element *element*
precondition: there is no edge between *first* and *second*, *first* and *second* are different
postcondition: undirected edge with end vertices *first* and *second* and element *element* has been added to the graph
edge ← undirected edge with end vertices *first* and *second* and element *element*
add to *edge* to *edges*
degree of *first* ← degree of *first* + 1
degree of *second* ← degree of *second* + 1
return *edge*

insertDirectedEdge(*first*, *second*, *element*):
input: vertices and element

output: directed edge from *first* to *second* with element *element*
precondition: there is no undirected edge between *first* and *second*, there is no directed edge from *first* to *second*, *first* and *second* are different
postcondition: directed edge from *first* to *second* with element *element* has been added to the graph
edge ← directed edge from *first* to *second* with element *element*
add to *edge* to *edges*
degree of *first* ← degree of *first* + 1
degree of *second* ← degree of *second* + 1
outdegree of *first* ← outdegree of *first* + 1
indegree of *second* ← indegree of *second* + 1
return *edge*

insertVertex(*element*):
input: element
output: vertex with element *element*
postcondition: vertex with element *element* has been added to graph
vertex ← vertex with element *element* and degree, indegree and outdegree all 0
add *vertex* to *vertices*
return *vertex*

removeVertex(*vertex*):
input: vertex to be removed
postcondition: *vertex* and edges incident on *vertex* have been removed from graph
for each edge *edge* in *edges* **do**
  if *vertex* is end vertex of *edge* **then**
    removeEdge(*edge*)
remove *vertex* from *vertices*

removeEdge(*edge*):
input: edge to be removed
postcondition: *edge* has been removed from graph
first, *second* ← end vertices of *edge*
degree of *first* ← degree of *first* − 1
degree of *second* ← degree of *second* − 1
if *edge* is directed **then**
  outdegree of *first* ← outdegree of *first* − 1
  indegree of *second* ← indegree of *second* − 1
remove *edge* from *edges*

makeUndirected(*edge*):
input: edge
postcondition: *edge* is undirected
if edge *edge* is directed **then**
  first, *second* ← end vertices of *edge*
  out-degree of *first* ← out-degree of *first* − 1
  in-degree of *second* ← in-degree of *second* − 1
  set *edge* to be undirected

reverseDirection(*edge*):
input: edge
precondition: *edge* is directed
postcondition: direction of *edge* has been reversed
first, *second* ← end vertices of *edge*
out-degree of *first* ← out-degree of *first* − 1
in-degree of *first* ← in-degree of *first* + 1
in-degree of *second* ← in-degree of *second* − 1

out-degree of *second* $\leftarrow$ out-degree of *second* + 1
swap origin and destination of *edge*

setDirectionFrom(*edge*, *vertex*):
  input: edge and vertex
  precondition: *vertex* is an end vertex of *edge*
  postcondition: *edge* has been directed away from *vertex*
left as an exercise

setDirectionTo(*edge*, *vertex*):
  input: edge and vertex
  precondition: *vertex* is an end vertex of *edge*
  postcondition: *edge* has been directed to *vertex*
left as an exercise