Practical Work nr. 1

Problem statement

Design and implement an abstract data type *directed graph* and a function (either a member function or an external one, as your choice) for reading a directed graph from a text file.

The vertices will be specified as integers from 0 to *n*-1, where *n* is the number of vertices.

Edges may be specified either by the two endpoints (that is, by the source and target), or by some abstract data type *Edge_id* (that data type may be a pointer or reference to the edge representation, but then care should be taken not to expose the implementation details of the graph).

Each edge will have an integer value (for instance, a cost) attached to it. The *directed graph* data type shall allow its users to retrieve and modify that integer and shall not interpret or restrain it in any way.

Feature list

F1.	Add a vertex
F2.	Remove a vertex
F3.	Add an edge
F4.	Remove an edge
F5.	Get number of vertices
F6.	Check if an edge exists
F7.	Get the degrees of a vertex
F8.	Show inbound edges of a vertex
F9.	Show outbound edges of a vertex
F10.	Get the cost of an edge
F11.	Set the cost of an edge
F12.	Help
F13.	Clear screen
F14.	Exit

Program format

The graph will be read from a text file having the following format:

- On the first line, **n** and **m**, **n** being the number of vertices and **m** the number of edges in the graph
- On each of the following **m** lines there are three numbers: **x**, **y** and **c**, describing an edge: the source, the target and the cost of the edge.

Menu:

```
> Select a command:
  1 - Add VERTEX
  2 - Remove VERTEX
  3 - Add EDGE
  4 - Remove EDGE
  5 - Get number of Vertices
  6 - Check if an EDGE exists
  7 - Degrees of VERTEX
  8 - Show the inBound edges of a vertex
  9 - Show the outBound edges of a vertex
 10 - Get the cost of an edge
 11 - Set the cost of an edge
  h - Help
  c - Clear screen
  0 - Exit
```

File input example:

test0.txt

```
5 6
0 0 1
0 1 7
1 2 2
2 1 -1
1 3 8
2 3 5
```

Running scenario

User	Program

	`> Select a command:`
	`  1 – Add VERTEX`
	`  2 – Remove VERTEX`
	`  3 – Add EDGE`
	`  4 – Remove EDGE`
	`  5 – Get number of Vertices`
	`  6 – Check if an EDGE exists`
	`  7 – Degrees of VERTEX`
	`  8 – Show inBound edges of a vertex`
	`  9 – Show outBound edges of a vertex`
	` 10 – Get the cost of an edge`
	` 11 – Set the cost of an edge`
	`  h – Help`
	`  c – Clear screen`
	`  0 – Exit`

1	
	`> Vertex:`
1	
	`> Vertex already exists.`
9	
	`> Vertex:`
2	

	`> Outbound edges:`
	`  1  3`

2	
	`> Vertex to be removed:`
1	
	`> Vertex was successfully removed.`
9	
	`> Vertex:`
2	

	`> Outbound edges:`
	`  3`

User	Program
5	
	> There are 4 vertices in the graph.
1	
	> Vertex:
6	
	> Vertex was successfully added.
5	
	> There are 5 vertices in the graph.
3	
	> Source vertex:
6	
	> Target vertex:
3	
	> Edge weight:
10	
	> Edge was successfully added.
4	
	> Source vertex:
6	
	> Target vertex:
3	
	> Edge was successfully removed.
6	
	> Source vertex:
6	
	> Target vertex:
3	
	> Inexistent edge.

User	Program
10	
	`> Source vertex:`
2	
	`> Target vertex:`
3	
	`> The cost from 2 to 3 is 5.`
11	
	`> Source vertex:`
2	
	`> Target vertex:`
3	
	`> The new cost:`
10	
	`> Update successful.`
10	
	`> Source vertex:`
2	
	`> Target vertex:`
3	
	`> The cost from 2 to 3 is 10.`
7	
	`> Vertex:`
2	

	`> inDegree of 2 is 0.`
	`> outDegree of 2 is 1.`

0	
	`> Exiting...`

Specification

We shall define a class named Graph representing a directed graph.

Graph

- a class that creates a graph using a dictionary with inbound and outbound edges for every vertex and also a dictionary with the cost for every edge.
- initially, the graph is empty; the user may choose to load a graph from a text file or input the data manually; each line of the text file contains 2 vertices and the cost of their edge.

The ***Graph class*** will provide the following methods:

1. **getVertices()** : shows the number of vertices in the graph
2. **isEdge(x, y)** : checks to see if there is an edge from x to y
3. **inBound(v)** : shows all the vertices u such that there is an edge from u to v
4. **outBound(v)** : shows all the vertices u such that there is an edge from v to u
5. **inDegree(v)** : shows the number of inbounds of vertex v
6. **outDegree(v)** : shows the number of outbounds of vertex v
7. **addVertex(v)** : adds a new vertex, v, to the graph
8. ***removeVertex(v)*** : removes vertex v from the graph (if such a vertex exists) and all the edges that vertex has
9. **addEdge(x, y)** : adds an edge from x to y to the graph
10. **removeEdge(x, y)** : removes the edge from x to y from the graph (if the edge exists)
11. **setCost(x, y, c)** : modifies the cost of the edge from x to y with c
12. **getCost(x, y)** : returns the cost of the edge from x to y

Implementation

- The inbound dictionary contains as key a vertex v and as its value, a list of vertices u, such that there is an edge from u to v.
- The outbound dictionary contains as key a vertex v and as its value, a list of vertices u, such that there is an edge from v to u.
- The costs dictionary contains as key a vertex v and as its value a list of tuples (u, c) such that there is an edge from v to u of cost c
- Initially all three dictionaries are empty. When a vertex is added, a key with the specified value is created in the costs dictionary along with an empty list. When an edge is added, the other two dictionaries are initialized (in case they do not exist already) and a tuple is created in the costs dictionary.

The implementation for adding a vertex

```
def addVertex(self, v):
    # adds a vertex to the graph, but only to the vertices dictionary
          try:
          v = int(v)
          except ValueError:
                return "> Invalid input."

          if v in self.vertices.keys():
                return "> Vertex already exists."

          self.vertices[v] = []

          return "> Vertex was successfully added."
```

The implementation for removing a vertex

```
def removeVertex(self, v):
      # removes a vertex from the graph along with all the edges that
      # come into contact with that vertex

      try:
          v = int(v)
      except ValueError:
          return "> Invalid input."

      if v not in self.vertices.keys():
          return "> Vertex does not exist."

      if v in self.outEdges:
          for node in self.outEdges[v]:
              self.inEdges[node].remove(v)

      if v in self.inEdges:
          for node in self.inEdges[v]:
              self.outEdges[node].remove(v)
          for node in self.inEdges[v]:
              for t in self.vertices[node]:
                  if t[0] == v:
                      self.vertices[node].remove(t)

      if v in self.inEdges:
          self.inEdges.pop(v)
      if v in self.outEdges:
          self.outEdges.pop(v)
      if v in self.vertices:
          self.vertices.pop(v)

      return "> Vertex was successfully removed."
```

The implementation for adding an edge

```
def addEdge(self, x, y, c):
        # adds an edge from x to y with the cost c to the graph

        try:
            x = int(x)
            y = int(y)
            c = int(c)
        except ValueError:
            return "> Invalid input."

        if x not in self.vertices.keys():
            return "> Vertex " + str(x) + "does not exist."

        if y not in self.vertices.keys():
            return "> Vertex " + str(y) + "does not exist."

        if self.isEdge(x, y):
            return "> Edge already exists, cannot overwrite."

        self.vertices[x].append((y, c))

        if y in self.inEdges.keys():
            self.inEdges[y].append(x)
        else:
            self.inEdges[y] = [x]

        if x in self.outEdges.keys():
            self.outEdges[x].append(y)
        else:
            self.outEdges[x] = [y]

        return "> Edge was successfully added."
```

The implementation for removing an edge

```
  def removeEdge(self, x, y):
        # removes an edge from the graph

        x = int(x)
        y = int(y)

        # Remove y from the OutEdges of x
        # Remove x from the InEdges of y
        # Remove y from self.vertices[x]

        self.outEdges[x].remove(y)

        self.inEdges[y].remove(x)

        for t in self.vertices[x]:
            if t[0] == y:
                self.vertices[x].remove(t)

        return "> Edge was successfully removed."
```