

```
1  /*
2  Alexandru Zaharia 918
3  - Practical work no. 1 -
4
5  Problem:
6      Design and implement an abstract data type directed graph and a function      ↗
7      (either a member function or an external one, as your choice) for reading a ↗
8      directed graph from a text file.
9      The vertices will be specified as integers from 0 to n-1, where n is the      ↗
10     number of vertices.
11     Edges may be specified either by the two endpoints (that is, by the source   ↗
12     and target), or by some abstract data type Edge_id (that data type may be a ↗
13     pointer or reference to the edge representation, but then care should be     ↗
14     taken not to expose the implementation details of the graph).
15     Each edge will have an integer value (for instance, a cost) attached to it.  ↗
16     The directed graph data type shall allow its users to retrieve and modify    ↗
17     that integer and shall not interpret or restrain it in any way.
18
19
20
21     Required operations:
22     - get the number of vertices;
23     - given two vertices, find out whether there is an edge from the first      ↗
24     one to the second one, and retrieve the Edge_id if there is an edge          ↗
25     (the latter is not required if an edge is represented simply as a pair      ↗
26     of vertex identifiers);
27     - get the in degree and the out degree of a specified vertex;
28     - iterate through the set of outbound edges of a specified vertex (that     ↗
29     is, provide an iterator). For each outbound edge, the iterator shall        ↗
30     provide the Edge_id of the current edge (or the target vertex, if no        ↗
31     Edge_id is used).
32     - iterate through the set of inbound edges of a specified vertex (as        ↗
33     above);
34     - get the endpoints of an edge specified by an Edge_id (if applicable);
35     - retrieve or modify the information (the integer) attached to a            ↗
36     specified edge.
37
38     The operations must take no more than:
39     -  $O(\deg(x) + \deg(y))$  for: verifying the existence of an edge and for      ↗
40     retrieving the edge between two given vertices.
41     -  $O(1)$  for: getting the first or the next edge, inbound or outbound to a    ↗
42     given vertex; get the endpoints, get or set the attached integer for an     ↗
43     edge (given by an Edge_id or, if no Edge_id is defined, then given by      ↗
44     its source and target); get the total number of vertices or edges; get     ↗
45     the in-degree or the out-degree of a given vertex.
46
47     Note: You are allowed to use, from existing libraries, data structures such  ↗
48     as linked lists, double-linked lists, maps, etc. However, you are not      ↗
49     allowed to use already-implemented graphs (though, you are encouraged to    ↗
50     take a look at them).
51
52  */
53
```

```
29 class DGraph {
30 private:
31     unordered_map<int, vector<int>> inbounds;
32     unordered_map<int, vector<int>> outbounds;
33
34 public:
35     /* CONSTRUCTORS */
36     DGraph(int n = 10);
37     /* Default constructor for the DGraph class.
38        Input: n (int) - the number of vertices. */
39     DGraph(const DGraph &g);
40     /* Copy constructor for the DGraph class.
41        Input: g (const DGraph&) - the graph to be copied. */
42
43     /* DESTRUCTOR */
44     ~DGraph();
45     /* Destructor for the DGraph class. */
46
47     /* GETTERS */
48     int getNoOfVertices();
49     /* Gets the number of vertices.
50        Output: (int) = number of vertices */
51     int getNoOfEdges();
52     /* Gets the number of edges.
53        Output: (int) = number of edges */
54     int getInDegree(int x);
55     /* Gets the inbound degree of a vertex.
56        Input: x (int) - the vertex we get the inbound degree for
57        Output: (int) - the inbound degree of 'x' */
58     int getOutDegree(int x);
59     /* Gets the outbound degree of a vertex.
60        Input: x (int) - the vertex we get the outbound degree for
61        Output: (int) - the outbound degree of 'x' */
62     vector<int> getInbounds(int x);
63     /* Gets the list of inbound edges of x.
64        Input: x (int) - the vertex we get the inbound edges list for
65        Output: (vector<int>) - the inbound list of edges for 'x' */
66     vector<int> getOutbounds(int x);
67     /* Gets the list of outbound edges of x.
68        Input: x (int) - the vertex we get the outbound edges list for
69        Output: (vector<int>) - the outbound list of edges for 'x' */
70
71     /* ADD */
72     void addEdge(int x, int y);
73     /* Add an edge between two vertexes.
74        Input: x (int) - the start vertex
75              y (int) - the end vertex */
76     bool isEdge(int x, int y);
77     /* Verify if there exists an edge between 'x' and 'y'
78        Input: x (int) - the start vertex
79              y (int) - the end vertex
80        Output: true (bool) - if there exists and edge btw 'x' and 'y'
```

```

81         false (bool - otherwise) */
82
83     /* ITERATORS */
84     vector<int>::iterator iteratorInBegin(int x);
85     /* Get an iterator to the beginning of the inbound list of 'x'.
86        Input: x (int) - the vertex we work with
87        Output: (vector<int>::iterator) - iterator to the beginning
88               of the inbound list of 'x' */
89     vector<int>::iterator iteratorInEnd(int x);
90     /* Get an iterator to the end of the inbound list of 'x'.
91        Input: x (int) - the vertex we work with
92        Output: (vector<int>::iterator) - iterator to the end
93               of the inbound list of 'x' */
94     vector<int>::iterator iteratorOutBegin(int x);
95     /* Get an iterator to the beginning of the outbound list of 'x'.
96        Input: x (int) - the vertex we work with
97        Output: (vector<int>::iterator) - iterator to the beginning
98               of the outbound list of 'x' */
99     vector<int>::iterator iteratorOutEnd(int x);
100    /* Get an iterator to the end of the outbound list of 'x'.
101       Input: x (int) - the vertex we work with
102       Output: (vector<int>::iterator) - iterator to the end
103              of the outbound list of 'x' */
104
105 };
106
107 class DGraphCost : public DGraph {
108 private:
109     map<pair<int, int>, int> costs; //the cost
110 public:
111     /*CONSTRUCTORS */
112     DGraphCost(int n = 10);
113     /* Default constructor for the DGraphCost class.
114        Input: n (int) - the number of vertices. */
115     DGraphCost(const DGraphCost& g);
116     /* Copy constructor for the DGraphCost class.
117        Input: g (const DGraphCost&) - the graph to be copied. */
118
119     ~DGraphCost();
120     /* Destructor of the DGraphCost class. */
121
122     /* GETTERS */
123     int getCost(pair<int, int> edge);
124     /* Get the cost of an edge.
125        Input: edge (pair<int, int>) - the edge represented as a pair of
126               vertices
127        Output: (int) - the cost of the edge. */
128     map<pair<int, int>, int> getCosts();
129     /* Get the list of costs.
130        Output: (map<pair<int, int>, int>) - the list of costs represented as
131               a mapping
132               of pairs of ints to some ints. */

```

```
131
132     /* SETTERS */
133     void setCost(pair<int, int> edge, int cost);
134     /* Set the cost of an edge.
135        Input: edge (pair<int, int>) - the edge represented by a pair of
               ints
               cost (int) - the new cost of the edge. */
136
137
138     /* OPERATIONS */
139     void addEdge(int x, int y, int z);
140     /* Add an edge between 'x' and 'y' with the cost 'z'.
141        Input: x (int) - the start vertex
142               y (int) - the end vertex
143               z (int) - the cost. */
144 };
145
146 /* ----- UI ----- */
147 void readEdge(DGraph &g) {
148     /* Read an edge from the user and add it to the graph.
149        Input: g (DGraph&) - the graph we will add the edge to. */
150 }
151
152
153 string chooseFileG() {
154     /* Choose a file to initialize the costless graph.
155        Output: (string) - the name of the file. */
156 }
157
158 string chooseFileGC() {
159     /* Choose a file to initialize the cost graph.
160        Output: (string) - the name of the file. */
161 }
162
163
164
165 int chooseGraph() {
166     /* Choose a costless or a cost graph.
167        Output: 1 - for costless graphs
168               2 - for cost graphs*/
169 }
170
171 DGraph initializeG() {
172     /* Initialize the costless graph.
173        Output: (DGraph) - the initialized graph. */
174 }
175
176 void menuCommandsG() {
177     /* Commands for the costless graph menu. */
178 }
179 void menuCommandsGC() {
180     /* Commands for the cost graph menu. */
181 }
```

```
182
183 int executeCommandG(string cmd, DGraph& g) {
184     /* Execute the given command on the given costless graph.
185         Input:  cmd (string) - the command
186                g (DGraph&) - the costless graph.
187         Output: 1 - for command 'x'
188                0 - otherwise. */
189 }
190 int executeCommandGC(string cmd, DGraphCost& g) {
191     /* Execute the given command on the given cost graph.
192     */
193 }
194 void mainMenu() {
195     /* The main menu. Here we put all the other menus together. */
196 }
```