```
/*
Alexandru Zaharia 918
- Practical work no. 1 -

Problem:
    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.


    Required operations:
        - get the number of vertices;
        - given two vertices, find out whether there is an edge from the first one to the second one, and retrieve the Edge_id if there is an edge (the latter is not required if an edge is represented simply as a pair of vertex identifiers);
        - get the in degree and the out degree of a specified vertex;
        - iterate through the set of outbound edges of a specified vertex (that is, provide an iterator). For each outbound edge, the iterator shall provide the Edge_id of the curren edge (or the target vertex, if no Edge_id is used).
        - iterate through the set of inbound edges of a specified vertex (as above);
        - get the endpoints of an edge specified by an Edge_id (if applicable);
        - retrieve or modify the information (the integer) attached to a specified edge.

    The operations must take no more than:
        - O(deg(x)+deg(y)) for: verifying the existence of an edge and for retrieving the edge between two given vertices.
        - O(1) for: getting the first or the next edge, inbound or outbound to a given vertex; get the endpoints, get or set the attached integer for an edge (given by an Edge_id or, if no Edge_id is defined, then given by its source and target); get the total number of vertices or edges; get the in-degree or the out-degree of a given vertex.

    Note: You are allowed to use, from existing libraries, data structures such as linked lists, double-linked lists, maps, etc. However, you are not allowed to use already-implemented graphs (though, you are encouraged to take a look at them).

*/

```

```
class DGraph {
private:
    unordered_map<int, vector<int>> inbounds;
    unordered_map<int, vector<int>> outbounds;

public:
    /* CONSTRUCTORS */
    DGraph(int n = 10);
    /* Default constructor for the DGraph class.
            Input: n (int) - the number of vertices. */
    DGraph(const DGraph &g);
    /* Copy constructor for the DGraph class.
            Input: g (const DGraph&) - the graph to be copied. */

    /* DESTRUCTOR */
    ~DGraph();
    /* Destructor for the DGraph class. */

    /* GETTERS */
    int getNoOfVertices();
    /*  Gets the number of vertices.
            Output: (int) = number of vertices */
    int getNoOfEdges();
    /*  Gets the number of edges.
            Output: (int) = number of edges */
    int getInDegree(int x);
    /*  Gets the inbound degree of a vertix.
            Input: x (int) - the vertix we get the inbound degree for
            Output: (int) - the inbound degree of 'x' */
    int getOutDegree(int x);
    /*  Gets the outbound degree of a vertix.
            Input: x (int) - the vertix we get the outbound degree for
            Output: (int) - the outbound degree of 'x' */
    vector<int> getInbounds(int x);
    /*  Gets the list of predecesors of x.
            Input: x (int) - the vertix we get the predecesor list for
            Output: (vector<int>) - the predecesor list of edges for 'x' */
    vector<int> getOutbounds(int x);
    /*  Gets the list of succesors of x.
            Input: x (int) - the vertix we get the succesor list for
            Output: (vector<int>) - the succesor list  of edges for 'x' */

    /* ADD */
    void addEdge(int x, int y);
    /*  Add an edge between two vertixes.
            Input:  x (int) - the start vertix
                    y (int) - the end vertix */
    bool isEdge(int x, int y);
    /*  Verify if there exists an edge between 'x' and 'y'
            Input:  x (int) - the start vertix
                    y (int) - the end vertix
            Output: true (bool) - if there exists and edge btw 'x' and 'y'
```

```
                        false (bool - otherwise) */

    /* ITERATORS */
    vector<int>::iterator iteratorInBegin(int x);
    /*  Get an iterator to the begining of the inbound list of 'x'.
            Input: x (int) - the vertix we work with
            Output: (vector<int>::iterator) - iterator to the begining
                                            of the inbound list of 'x' */
    vector<int>::iterator iteratorInEnd(int x);
    /*  Get an iterator to the end of the inbound list of 'x'.
            Input: x (int) - the vertix we work with
            Output: (vector<int>::iterator) - iterator to the end
                                            of the inbound list of 'x' */
    vector<int>::iterator iteratorOutBegin(int x);
    /*  Get an iterator to the begining of the outbound list of 'x'.
            Input: x (int) - the vertix we work with
            Output: (vector<int>::iterator) - iterator to the begining
                                            of the outbound list of 'x' */
    vector<int>::iterator iteratorOutEnd(int x);
    /*  Get an iterator to the end of the outbound list of 'x'.
            Input: x (int) - the vertix we work with
            Output: (vector<int>::iterator) - iterator to the end
                                            of the outbound list of 'x' */

};

class DGraphCost : public DGraph {
private:
    map<pair<int, int>, int> costs; //the cost
public:
    /*CONSTRUCTORS */
    DGraphCost(int n = 10);
    /*  Default constructor for the DGraphCost class.
            Input: n (int) - the number of vertices. */
    DGraphCost(const DGraphCost& g);
    /*  Copy constructor for the DGraphCost class.
            Input: g (const DGraphCost&) - the graph to be copied. */

    ~DGraphCost();
    /* Destructor of the DGraphCost class. */

    /* GETTERS */
    int getCost(pair<int, int> edge);
    /*  Get the cost of an edge.
            Input: edge (pair<int, int>) - the edge represented as a pair of vertices
            Output: (int) - the cost of the edge. */
    map<pair<int, int>, int> getCosts();
    /*  Get the list of costs.
            Output: (map<pair<int, int>, int>) - the list of costs represented as a mapping
                                                of pairs of ints to some ints. */

    /* SETTERS */
    void setCost(pair<int, int> edge, int cost);
    /*  Set the cost of an edge.
            Input:  edge (pair<int, int>) - the edge represented by a pair of ints
                    cost (int) - the new cost of the edge. */

    /* OPERATIONS */
    void addEdge(int x, int y, int z);
    /*  Add an edge between 'x' and 'y' with the cost 'z'.
            Input:  x (int) - the start vertix
                    y (int) - the end vertix
                    z (int) - the cost. */
};

/* ------- UI -------- */
void readEdge(DGraph &g) {
    /*  Read an edge from the user and add it to the graph.
            Input: g (DGraph&) - the graph we will add the edge to. */
}


string chooseFileG() {
    /*  Choose a file to initialize the costless graph.
            Output: (string) - the name of the file. */
}

string chooseFileGC() {
    /*  Choose a file to initialize the cost graph.
            Output: (string) - the name of the file. */
}



int chooseGraph() {
    /*  Choose a costless or a cost graph.
            Output: 1 - for costless graphs
                    2 - for cost graphs*/
}

DGraph initializeG() {
    /*  Initialize the costless graph.
            Output: (DGraph) - the initialized graph. */
}

void menuCommandsG() {
    /* Commands for the costless graph menu. */
}
void menuCommandsGC() {
    /* Commands for the cost graph menu. */
}
```

```cpp

int executeCommandG(string cmd, DGraph& g) {
    /*  Execute the given command on the given costless graph.
            Input:  cmd (string) - the command
                    g (DGraph&) - the costless graph.
            Output: 1 - for command 'x'
                    0 - otherwise. */
}
int executeCommandGC(string cmd, DGraphCost& g) {
    /*  Execute the given command on the given cost graph.
}

void mainMenu() {
    /* The main menu. Here we put all the other menus together. */
}
```