Practical 3 - Handling exceptions. Templates

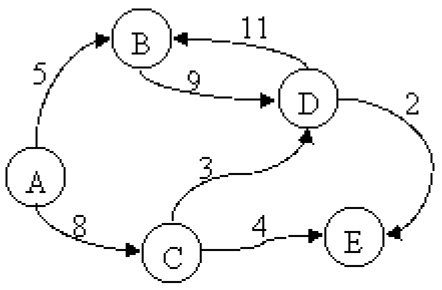
Instructions

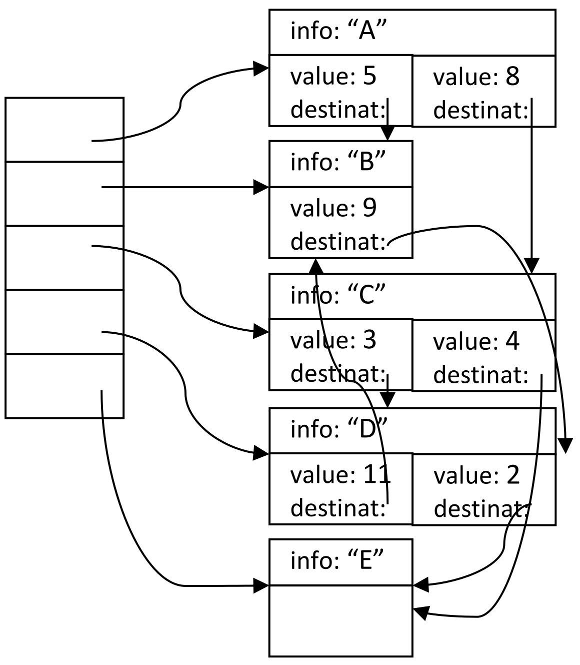
- Download the ***aeda2021_p03.zip*** file from the moodle page and unzip it (contains the ***lib*** folder, the ***Tests*** folder with files ***graph.h*** and ***tests.cpp***, and the ***CMakeLists*** and ***main.cpp*** files)
- Note that the ***unit tests for this project are commented***. Remove comments as you implement the tests.
- You must perform the exercise respecting the order of the questions
- Perform the implementation in the ***graph.h*** file

Exercise

The **Graph** class allows the representation of an oriented graph, composed of **nodes** connected by **edges**. The information contained in the nodes and edges of the graph can be associated with different types of data. The **Graph** class is a generic class with two arguments, nodes and edges. Consider that all nodes in the graph are different.

Each instance of the **Graph** class contains a vector of pointers to nodes. For each node, there is a vector of edges (ordered according to the destination node). The following figure shows the data structure for an example.





The **Graph** class declaration is as follows:

```
template <class N, class E>
class Node{
public:
      N info;
      vector< Edge<N,E> > edges;
      No(N inf) {
            info = inf;
      }
};

template <class N, class E>
class Edge {
public:
      E value;
      Node<N,A> *destination;
      Edge(No<N,E> *dest, E val) {
            value = val;
            destination = dest;
      }
};
```

```
template <class N, class E>
class Graph {
      vector< Node<N,E> *> nodes;
  public:
    Graph();
    ~Graph();
    Graph & addNode(const N &inf);
    Graph & addEdge(const N &begin, const N &end, const E &val);
    Graph & removeEdge(const N &begin, const N &end);
    E & edgeValue(const N &begin, const N &end);
    unsigned numEdges() const;
    unsigned numNodes() const;
    void print(std::ostream &os) const;
};
```

The implementation must be done in the *graph.h* file.

a) Implement:

- the *Graph* class constructor rand destructor
- the *numNodes()* method (which returns the number of nodes in the graph)
- the *numEdges()* method (which returns the number of edges in the graph)

b) Implement the *addNode (const N & inf)* method, which adds a new node to the graph and returns the changed graph (*this*). This method should throw the *NodeAlreadyExists* exception, if that node already exists (see unit test).

The *NodeAlreadyExists* exception is already implemented.

c) Implement the *addEdge(const N &begin, const N &end, const E &val)* method, which adds a new edge to the graph and returns the changed graph (*this*). This method should throw the appropriate exception if the edge already exists:

- *NodeDoesNotExist* exception: this exception is already implemented
- *EdgeAlreadyExists* exception:
  - Implement this exception. Implement the << operator, which prints the values of the begin and destination nodes of the edge

d) Implement the *edgeValue(const N &begin, const N &end)* method, which returns a reference to the specified edge data. This function should throw the appropriate exception if the edge does not exist in the graph (see unit test).

- *EdgeDoesNotExist* exception:
  - Implement the << operator, which prints the values of the begin and destination nodes of the edge

e) Implement the *removeEdge(const N &begin*, *const N &end)* method, which removes an edge of the graph and returns the changed graph (*this*). This function should throw the appropriate exception if the edge does not exist in the graph (identical to the previous paragraph).

f) Implement the *print*(*std::ostream &os)* method, which writes, for an output stream, the graph information. For the example indicated above, the method provides:

```
( A [B 5] [C 8] ) ( B [D 9] ) ( C [D 3] [E 4] ) ( D [B 11] [E 2] ) ( E )
```

g) Use the previous method to implement the << operator.

h) Document the implemented methods (use Doxygen).