The assignment for week 5 and 6 of CXX01 is to implement a graph as a reusable component. This assignment should be completed in groups of at most two students.

1 Graph

Your group will write a reusable component which implements a graph data structure. Your library will have to adhere to the requirements listed below.

- The library implements a graph by building on top of the doubly linked list library you produced in week 4.
- The library supports:
  - undirected and directed graphs;
  - unweighted and weighted graphs.
- The graph consists of a set of vertices. This set is stored in a doubly linked list.
- Each vertex contains a name (`char *`), a generic data pointer (`void *`) and a set of outgoing edges. This set is stored in a doubly linked list.
- Each edge contains a weight (`double`) and a pointer to the vertex it connects to.
- A vertex can be added to the graph.
- The graph can be searched for a vertex with a specific name.
- The graph can be searched for a vertex which contains a generic data pointer for which a specific predicate is true.
- An edge can be added to a given vertex.
- A given vertex can be deleted from the graph.
- The graph itself can be disposed of properly (without any memory leaks!)

Of course you are totally free to implement extra functionalities.

Design the API (Application Programmers Interface) for your graph carefully. Discuss it with a teacher to verify your design.

2 Testing

All functions should have proper unit tests and you must use a test framework (e.g. µnit, see: `https://nemequ.github.io/munit/`.) to manage your testing. Use a test driven approach. So write the unit test for a specific function *before* you write the implementation of that specific function.

3 Documentation

Document the doubly linked library and the graph library in a short report. Use proper citations and references when appropriate. Also explain the design decisions you made while implementing the libraries. Describe your testing approach and testing results in your report. Commit and push your report in pdf format into your repository. If you like you can use Doxygen[1] to generate the documentation of the API (Aplications Programmers Interface) of your libraries.

4 Implementation ideas

You should be able to use more than one graph in your program, therefore all functions should have a parameter which identifies the graph upon which the operation must be performed. It is a good idea to use a pointer to a specific `struct` e.g. `Graph` to identify a graph. You can use the data members of the struct for bookkeeping. A minimal example is given in Listing 1 but feel free to add more data members.

[1] See: `http://www.doxygen.org/`.

```c
typedef struct
{
    char* name;
    void* data;
    DoublyLinkedList edges;
} Vertex;

typedef struct
{
    Vertex* destination;
    double costs;
} Edge;

typedef struct
{
    DoublyLinkedList vertices;
} Graph;
```

Listing 1: A possible setup for your graph data structure.