PW2 - Small Project: Graph Library Representation by Adjacency Lists

Submission due before the 7-th PW session
Work to be done **in pairs**

Foreword

For all the practical work to be submitted, respect the following rules:

- **Precisely follow the topic statement**. Evaluation is based on this requirements specification document.
- **Variable names** cited in the topic statement must strictly be respected.
- The submission date for your work is a firm deadline. **Submission is to be done on Moodle**, before the PW session indicated. The exact deadline will be given by your teacher. If a report is to be provided, you can either join it as a PDF document to your submission, or alternatively print it and have it dated by the department secretary.
- If your submission is made of a single file, give it the name of both the students with the appropriate extension. If your submission is made of several files, group them as a ZIP or TGZ archive, then name it the same way. Don't forget to join the Makefile for compiling your source files.
- Pay careful attention to indentation and comments in your code. In particular, each function has to be *specified* by precisely indicating: what the function computes, what it returns, what are the input parameters and what are the output parameters (if any).

Subject: Graph Representation and Manipulation by Adjacency List

The objective of this small project is to develop in the C language a data structure describing a *graph*, with dynamic memory allocation according to the following principles:

- memory allocation only on demand,
- non-contiguous memory allocation,
- unused memory deallocation,
- insertion/deletion of nodes and edges,
- indirect access to successor nodes.

N.B. This small project is important because, not only will it be evaluated for itself, but it will serve in the second half of the semester for programming the full project.

Let $G = (V, E)$ be an undirected weighted graph. An example can be found in Fig. 1. This graphed is stored into memory by means of the array of adjacency lists shown in Fig. 2. Actually this structure can store directed as well as undirected graphs, since an undirected graph can be stored in the shape of a symmetric directed graph (which is the case in Fig. 2). We adopt the convention that an isolated node is represented as a node whose sole neighbour is the fictive node designated as "`-1`" (there are none in Fig. 2). The nodes whose array value is symbolized as $\not\Longrightarrow$ (which represents the NULL pointer) do not belong to the graph (e.g. nodes 9 and 10 in Fig. 2).

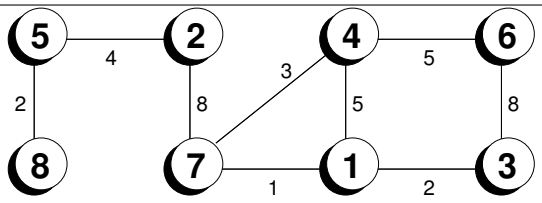

Figure 1: An Example of Undirected Graph G

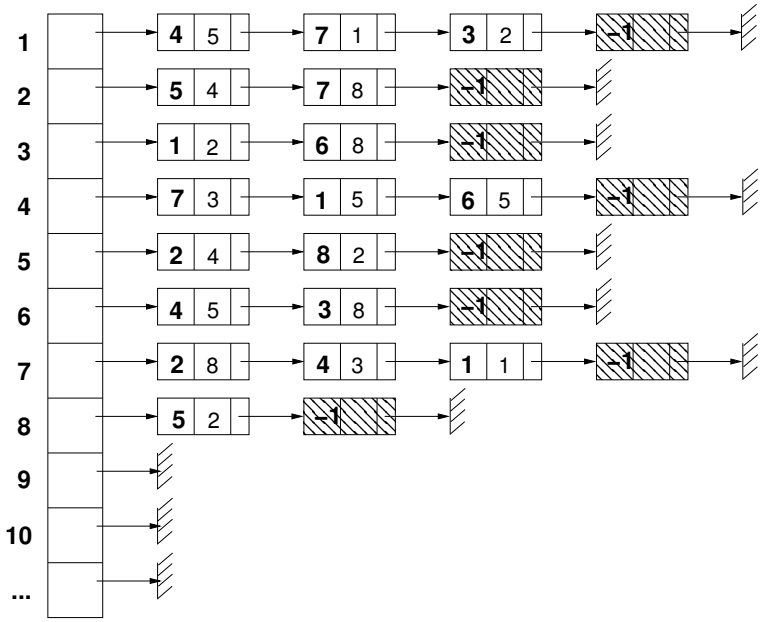

Figure 2: Representation of the graph G by means of an array of adjacency lists. Notice that it is stored as directed symmetric graph, i.e. there is a (y, x) edge for each (x, y) edge.

The list of neighbours of a node can also be coded as a doubly circular linked list with sentinel node (see Fig. 3). In this representation, a node that belongs to the graph is a node whose list of neighbours points *a minima* at an empty list (i.e. the sentinel). The advantage of this representation is that it facilitates programming the list operations.

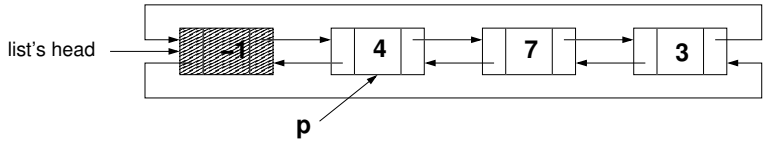

Figure 3: Example of a Doubly Circular Linked List with Sentinel

Implementation Instructions. Store the graph G as an array of adjacency lists. The indexes of the array are the node numbers, possibly minus 1 (decide what suits you best) since the indexes usually start from 0 whereas the node numbers usually start from 1. Each array element contains the memory address of the list of neighbours of the corresponding node. That list is in any of the shapes described above, and is built according to the order in which the nodes are inserted, rather than in the nodes' numerical order. Inserting a node at the end or the beginning has the same complexity with a doubly circular linked list (with sentinel or not). But with a simply linked list, insertion must be done at the list's beginning for efficiency reasons.

In order to facilitate the graph manipulations, the array data structure is completed with the

maximal potential number of nodes of the graph. Thus you will have to define a data structure named `Graph` with the following fields:

1. `isDirected`: indicates whether the graph is directed or not,
2. `nbMaxNodes`: the graph's maximum number of nodes,
3. `adjList`: an array whose elements have the type Neighbour.

   The type `Neighbour` is made of the following fields:

   - `neighbour`: an integer indicating the neighbour's number of the current node,
   - `weight`: an integer giving the weight of an edge,
   - `nextNeighbour`: a pointer to the following cell in the list's order, i.e. to the next neighbour of the current node,
   - `previousNeighbour`: a pointer to the previous cell in the list's order, i.e. to the previous neighbour of the current node (only in the case of a doubly circular linked list with sentinel).

Work to be done:

Build two libraries as general as possible in C. The first one is for manipulating the linked lists of the type you have chosen to implement. The second one is for manipulating the graphs stored as arrays of adjacency lists. The aim of these libraries is to allow easy graph or lists manipulation by providing all the basic operations on such structures such as insertion, deletion, graph traversal, etc. These libraries have to be thought of in a spirit of reuse.

The C program you are asked to write for this PW will make use of these libraries. It has to provide the user with a menu that offers the following functionalities:

- `create graph`: graph creation, if not previously done, according to a maximum number of nodes provided by the user;
- `load graph`: graph loading from a text file whose structure is as described in Fig. 4;
- `add node`: insertion of a node in the graph (with a number lower than the graph's maximum number of nodes);
- `add edge`: insertion of an edge in the graph after verification that both its endpoints are nodes of the graph and that the edge is not already in the graph. The user is asked if the edge is symmetric "`y`" or not "`n`" (except when the graph is totally undirected in which case each edge will automatically be seen as symmetric);
- `remove node`: deletion of a node from the graph (and of the edges it was an endpoint of);
- `remove edge`: deletion of an edge from the graph;
- `view graph`: graph display in the same format as the input file;
- `save graph`: graph saving in the text format illustrated in Fig. 4;
- `quit`: exit the program.

These operations must pay attention to whether the graph is already created or not. Indeed it is not created yet, the only allowed choices are to manually create it, or to load it from a file, or to quit the program.

All the cells that were dynamically allocated have be liberated, either when deletion operations are called or at the end of the program. This will be checked by your teacher thanks to the valgrind utility. No operation other than "quit" must end the program. Each operation execution must display a message indicating if it has succeeded or not (and preferably why not in case it has failed). We recommend that each of your functions returns an error code to its calling function, so that this function can either return it itself, or display an appropriate message to the user. Such messages can for example be "*error - edge (x, y) not added: (x, y) already exits*", "*error - unknown node x: not deleted*", "*OK - edge (x, y) successfully added*, etc.

```
# maximum number of nodes
11
# directed
n
# node: neighbours
1: (4/5), (7/1), (3/2)
2: (5/4), (7/8)
3: (1/2), (6/8)
4: (7/3), (1/5), (6/5)
5: (2/4), (8/2)
6: (4/5), (3/8)
7: (2/8), (4/3), (1/1)
8: (5/2)
```

Figure 4: Structure of the Text File for Defining and Recording a Graph

NB 1: The implementation of this subject necessitates many functions to be written. Your teacher, when he marks you work, will appreciate that these functions are clearly written and named, are kept of small size and can be reused. Ergonomy will also be taken into account. The menu must not be implemented directly inside the main program: dedicated functions to display the menu and manage it have to be written, that the main program will call.

NB 2: the libraries define an API used by the main program or the main functions. When compiling your source code, the libraries must appear as such (`-libgraph -liblist`) in the compilation line respectively for the libraries in reverse dependencies order for the files `libgraph.a` et `liblist.a`), and not as `.o` files. To this end, your `makefile` must build the libraries when necessary depending on the dependencies, thanks to an appropriate compilation line. It has to recompile only the source files whose compilation is required due to the modifications since the last compilation. Thus a particular attention must be paid to the dependencies between rules, to the usage of variables and to the genericity of compilation. The codes and additional compilation results will respectively be stored in directories named `SRC`, `OBJ`, `BIN`, `LIB`. The directory `INCLUDE` is optional.