# Practice 2

Laura de Paz Carbajo & Paula Samper López
Double degree Mathematics & Computer Science
Programming II, Group 2192
UAM 2018/2019

# Report: Practice 2

## EXERCISE 1

This is the ADT we implemented in exercise P2_E2:

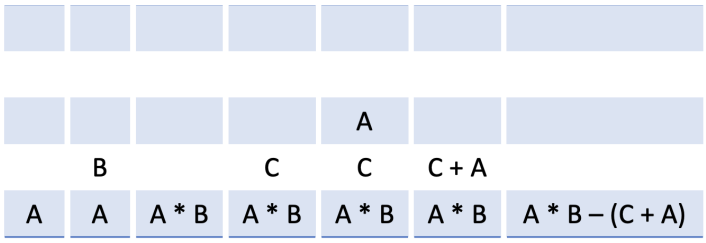
```
struct _EleStack {
        int *e;
};
```

This implementation would be enough to evaluate postfix expressions since we are only pushing integers into our stack. In case we wanted to work with other type of numbers such as decimal point numbers, we would have to change the structure so that it can receive and store decimal points without returning any error. In that case, instead of `int *e` we would write `double *e`.

We will show the behavior of this function with an example:

Let A, B and C be 3 integers. We will evaluate the following postfix expression using a stack:

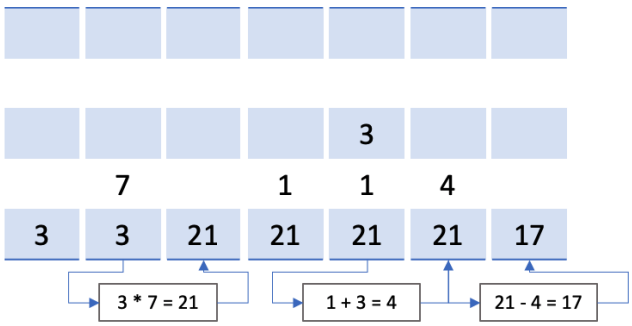
**A B * C A + -**




Popping the last element we get the result: **A*B - (C + A)**

Let assign integer values to A, B and C: A = 3 B = 7 C = 1

**3 7 * 1 3 + -**




With this example we have proven that the element ADT does not need anything but an integer (or any other type of number).

## EXERCISE 2

**P2_E1**

This exercise was quite straightforward because we had studied the implementation of the primitives of the stack in the theoretical lessons of this course and we had already done node.c for practice 1. Even so, we had some difficulties while writing the code for elestack file and later while correcting all valgrind errors. These last issues were mainly due to our lack of knowledge of the meaning of the flags in the makefile. For example, at the beginning we could not find the errors in our code because valgrind did not show the line they were in, until we realized we had to use -g flag.

**P2_E2**

We copied stack_elestack.c file from previous exercise and added meanElementStack function but before that we had to modify elestack file so that it would allow us to pop and push integers to the stack instead of nodes. It was not too complicated although we had to go through the entire file a couple of times because we always missed some references to nodes. Regarding the main function, we managed to do it correctly from the beginning and went ahead to fix errors and warnings.

In this exercise we just struggled with the mean calculation because it always returned an integer even though we had initialized a double. We realized that we were dividing two integers and assigning it to a double, so we simply converted those two integer numbers to double and made the corresponding operation.

**P2_E3**

We found this exercise quite easy, since we only had to adapt our previous ones to accept every kind of data. We create stack_fp.c, and stack_fp.h in order to achieve that, and we also change the two main.c. The program compiled quickly, and we just found a few errors in valgrind that we easily solved.

**P2_E4**

Firstly, we copied the Graph, Node and Stack´s ADTs from the previous exercises. Afterwards we had to modify node.c and node.h, in order to include the new members of the node structure (the label and the antecessor id), and we also created some new primitive functions within this ADT. We then changed graph.c and graph.h to fit the new node structure and moved to create the new function: Node *graph_findDeepSearch (Graph *g, int from_id, int to_id).
Even though we followed the instructions given, we struggled a little to make this function work properly, since we did not understand how it worked at the beginning.
When compiling the exercise, we realized we were dragging a problem on the function graph_GetConnectionsFrom, which made the function stack_push to fail. After solving the issue, we added the function void graph_printPath (FILE *pf, Graph *g, int idNode) to the exercise. We decided to design it as a recursive function rather than using a stack as we found it easier. Then we ran valgrind and found some errors that were easily corrected.

## EXERCISE 3

With this practice, we have really learnt the concept of stack´s scope, by reinforcing the theory given in the lectures; and the importance of it on various areas such as programming and technology.

We also wanted to highlight the fluency we have acquire when it comes to designing an ADT, being now so much more aware of its benefits and limitations. The implementation of the ADT´s primitives are completely clear for us now, and we are perfectly capable of using them correctly without any problem within other functions.

However, we found several difficulties at the beginning of the practice. The worst of them was valgrind. Even though finding every error that appeared when we ran it was quite complicated, as the time went by, we became more fluent when using it, and solving these errors was not difficult any more. Besides, we have understood its importance, and now thanks to it our codes are clearer and more organized.

One the other hand, we found that compiling our programs was the easiest part for us. Although this was our greatest concern when we were coursing Programming I, we have realized the level and fluency we have acquire on the matter, since now we can manage accurately easy C concepts and structures without any issue.