Programación II, 2018-2019
Escuela Politécnica Superior, UAM

PRACTICE 2: TAD Stack

GOALS

- Use of the TAD Node and Graph.
- Familiarization with the TAD Stack (LIFO), learning its potential and how it works.
- Implementation in C of the TAD Stack and the set of primitives necessary for its management.
- Using the TAD Stack to solve problems.

NORMS

As in practice 1, in this practice delivered **programs** must:

- Be written in **ANSI** C, following established programming **rules**.
- Compile without errors or warnings including "-ansi" and "-pedantic" flags when compiling.
- **Run without problem in a command console**.
- Include adequate error control; it is justifiable that a program does not admit inadequate values, but not that it behaves abnormally with these values.
- It must be accompanied by a **report** that must be prepared based on the proposed model and delivered with practice.

WORKING PLAN

Week 1: code of P2_E1. Each teacher will indicate in class if a delivery has to be made and how: paper, e-mail, Moodle, etc.

Week 2: code of P2_E1, P2_E2.

Week 3: to be designed…. Exercise 3.

Week 4: all the exercises (previous ones + exercise 4 to be designed).

The final delivery will be made through Moodle, following scrupulously the instructions indicated in the statement of practice 0 referring to the organization and nomenclature of files and projects. Remember that the compressed file must be called Px_Prog2_Gy_Pz, where '**x**' is the number of the practice, '**y**' is the practice group and '**z**' is the pair number (example of delivery of pair 5 of group 2161: P1_Prog2_G2161_P05.zip).
Moodle upload dates for the file are as follows:

- Students of Continuous Evaluation, the **week of March 20** (each group can make the delivery until 11:30 pm the night before their practice class).
- **Final Evaluation** students, as specified in the regulations.

INTRODUCTION

A **stack** is an abstract data type (TAD) defined as a sequence of homogeneous, **generic data**, implicitly ordered, in which the mode of access to the data follows a strategy of type LIFO (Last In First Out) , that is, the last entry is the first one to leave.
In a stack there are two basic operations for handling the data stored in it:

- ***Push**:* Adds an EleStack to the stack; the last EleStack added always occupies the top of the stack.
- ***Pop***: Removes the last stacked EleStack from the stack, ie. the one that occupies the top of the stack.

Other typical primitives of the TAD Stack are the following:

- Initialize the stack.
- Check if the stack is empty.
- Check if the stack is full.
- Free the stack.

In this practice, the TAD Stack will be implemented in the C language choosing the appropriate data structures, also, the functions associated with the main primitives to operate with the stack will be programmed. Subsequently, the programmed TAD Stack will be used to solve several problems.

According to the way to implement the stack that has been seen in theory, when stacking memory is stored in the stack, while unstacking is enough to return the (last) data as it was saved, without copying and freeing memory within the primitive function.

PART 1. EXERCICES

Exercise 1 (P2_E1). Implementation and testing of the TAD Stack as array of pointers to EleStack

Definition of the Stack data type. Implementation: selection of data structure and implementation of primitives

It is requested to implement in the file **stack_elestack.c** the TAD Stack primitives that are defined in the header file **stack_elestack.h** which, in turn, makes use of the TaD EleStack defined in **elestack.h**. The Status and Bool types are defined in types.h. The MAXSTACK constant indicates the maximum size of the stack. Its value should allow to develop the proposed exercises. The data structure chosen to implement the TAD STACK consists of an EleStack array of generic type and an integer that stores the index (in the array) of the EleStack located at the top of the stack. This structure is shown below:

```
/*
In stack_elestack.h */
typedef struct _Stack Stack;

/* in stack_elestack.c */
# define MAXSTACK 1024
struct _Stack {
    int top;
    EleStack * item[MAXSTACK];
};
```

The **stack_elestack.c** primitives are included in appendix 1.

1. Encapsulation of **the Stack behavior**

To encapsulate the behavior of the Stack and achieve that you can create stacks of different types (**although not at the same time), i**n this version of the Stack, EleStack elements are used. In this first exercise we will work with stacks of Nodes to be able to verify their correct operation. For it:

- Define the **EleStack type as an opaque TAD** in the **elestack.h** file (see appendix 2)):

```
typedef struct _EleStack EleStack;
```

- Define the **_EleStack type as a Node wrapper** in the **elestack-node.c** file:

```
struct _EleStack {
       Node* info;
};
```

In this last file you have to implement the functions defined in **elestack.h**.

Note that to achieve a greater abstraction, several of these functions use a pointer to void ('void *') which allows either returning a pointer of any type for example in (elestack_getInfo) or adding a pointer to any type of data to EleStack, for example in the function to elestack_setInfo.

2. Checking the correctness of the Stack type definition and its primitives

In order to evaluate the behaviour of the previous primitives, a test program p2_e1.c will be developed that will work with stacks of nodes. Use the TaD Node of practice 1. This program should:

- Declare and initialize a node, a stack element and a stack.

- Assign the name "first" and id 111 to the node and insert it into the stack.

- Assign the name "second" and id 222 to the node and insert it into the stack.

- Print the contents of the stack, also indicating the characters printed

- Extract the elements of the stack iteratively until the stack is empty. In each iteration, the element extracted from the stack will be printed.

- Print the contents of the stack, indicating the characters printed

```
> ./p2_e1
Print the contents of the stack
[second, 222, 0]
[first, 111, 0]
printed characters: 32
Emptying stack
[second, 222, 0] [first, 111, 0]
Print the contents of the stack after emptying:

printed characters:0
```

Exercise 2 (P2_E2). Implementation and testing of the TAD Stack with EleStack as wrapper of type int

Modification of the .h and .c files that you consider necessary

In this exercise we will make the necessary modifications in certain files so that now the type of data that handles the stack is of type int. This is:

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

Explain in the final report what files you have modified or created and why.

1. Checking the correctness of the Stack type definition and its primitives

In order to evaluate the operation of the previous primitives, a program **p2_e2.c** will be developed that (i) receives as an argument a positive integer, (ii) creates a stack with the integer numbers from the received integer to 0 (iii)) print the stack by the standard output (iv) execute a derivative function that receives a stack of integers and returns as a double the average value of its elements and (v) prints the value of the mean and the state of the stack.

Note: The function that calculates the average must NOT modify the stack. Errors must be managed at all times. Assume that a push preceded by a pop does not cause an error.

The pseudocode of the function must be included in the practice report.

```
> ./p2_e2 3
Stack before the call to the function
[3]
[2]
[1]
[0]
The average is 1,50000
Stack after the call to the function
[3]
[2]
[1]
[0]
```

Appendix 1: stack_elestack.h

```
#define MAXSTACK 100
typedef struct _Stack Stack;

/**-----------------------------------------------------------------
Initialize the stack reserving memory. Output: NULL if there was an error or the stack if it went well
-----------------------------------------------------*/
Stack * stack_ini();

/**-----------------------------------------------------------------
Remove the stack Input: the stack to be removed
-----------------------------------------------------------------*/
void stack_destroy(Stack *);

/**-----------------------------------------------------------------
Insert an EleStack in the stack. Input: an EleStack and the stack where to insert it. Output: NULL if you can
not insert it or the resulting stack if it succeeds
-----------------------------------------------------------------*/
Stack * stack_push(Stack *, const EleStack *);

/**-----------------------------------------------------------------
Extract an EleStack in the stack. Input: the stack from which to extract it. Output: NULL if you can not extract it
or the extracted EleStack if it succeeds. Note that the stack will be modified
-----------------------------------------------------------------*/
EleStack * stack_pop(Stack *);

/**-----------------------------------------------------------------
Check if the stack is empty. Input: stack. Output: TRUE if it is empty or FALSE if it is not
-----------------------------------------------------------------*/
Bool stack_isEmpty(const Stack *);

/**-----------------------------------------------------------------
Check if the stack is full. Input: stack. Exit: TRUE if it is full or FALSE if it is not
-----------------------------------------------------------------*/
Bool stack_isFull(const Stack *);

/**-----------------------------------------------------------------
Print the entire stack, placing the EleStack on top at the beginning of printing (and one EleStack per line).
Input: stack and file where to print it. Output: Returns the number of written characters.
-----------------------------------------------------------------*/
int stack_print(FILE*, const Stack*);
```

Appendix 2: elestack.h

```
typedef struct _EleStack EleStack;

/**-------------------------------------------------------------------
Initialize a stack EleStack. Output: Pointer to the initialized EleStack or NULL in case of error
-------------------------------------------------------------------*/
EleStack * EleStack_ini();

/**-------------------------------------------------------------------
Remove an EleStack. Entry: EleStack to destroy.
-------------------------------------------------------------------*/
void EleStack_destroy(EleStack *);

/**-------------------------------------------------------------------
Modify the data of an EleStack. Entry: The EleStack to be modified and the content to be saved in that
EleStack. Output: OK or ERROR
-------------------------------------------------------------------*/
Status EleStack_setInfo(EleStack *, void*);

/**-------------------------------------------------------------------
Returns the EleStack content. Entry: The EleStack. Output: The content of EleStack or NULL if there has been
an error.
-------------------------------------------------------------------*/
void * EleStack_getInfo(EleStack *);

/**-------------------------------------------------------------------
Copy one EleStack in another, reserving memory. Input: the EleStack to copy. Output: Returns a pointer to the
copied EleStack or NULL in case of error.
-------------------------------------------------------------------*/
EleStack * EleStack_copy(const EleStack *);

/**-------------------------------------------------------------------
Compares two EleStack. Entry: two EleStack to compare. Output: Returns TRUE if they are the same and if
not FALSE
-------------------------------------------------------------------*/
Bool EleStack_equals(const EleStack *, const EleStack *);

/**-------------------------------------------------------------------
Print the EleStack in a file that is already open. Input: File in which it is printed and the EleStack to print.
Output: Returns the number of written characters.
-------------------------------------------------------------------*/
int EleStack_print(FILE *, const EleStack *);
```