Programación II, 2018-2019
Escuela Politécnica Superior, UAM

Práctica 2: TAD Pila

OBJETIVOS

- Uso del TAD Nodo y Grafo.
- Familiarización con el TAD Pila (LIFO), aprendiendo su funcionamiento y potencial.
- Implementación en C del TAD Pila y del conjunto de primitivas necesarias para su manejo.
- Utilización del TAD Pila para resolver problemas.

NORMAS

Igual que en la práctica 1, en esta los ***programas*** que se entreguen deben:

- Estar escritos en ***ANSI C,*** siguiendo las **normas de programación** establecidas.
- Compilar sin errores ni *warnings* incluyendo las banderas "-ansi" y "-pedantic" al compilar.
- **Ejecutarse sin problema en una consola de comandos**.
- Incorporar un adecuado **control de errores**; es justificable que un programa no admita valores inadecuados, pero no que se comporte de forma anómala con dichos valores.
- Han de acompañarse de una ***memoria*** que debe elaborarse sobre el modelo propuesto y entregado con la práctica.

PLAN DE TRABAJO

Semana 1: código de P2_E1. Cada profesor indicará en clase si se ha de realizar alguna entrega y cómo: papel, e-mail, Moodle, etc.

Semana 2: código de P2_E1, P2_E2.

Semana 3: código de P2_E1, P2_E2, P2_E3.

Semana 4: todos los ejercicios.

La entrega final se realizará a través de Moodle, **siguiendo escrupulosamente las instrucciones** indicadas en el enunciado de la práctica 0 referentes a la organización y nomenclatura de ficheros y proyectos. Se recuerda que el fichero comprimido debe llamarse **Px_Prog2_Gy_Pz**, siendo '**x**' el número de la práctica, '**y**' el grupo de prácticas y '**z**' el número de pareja (ejemplo de entrega de la pareja 5 del grupo 2161: P1_Prog2_G2161_P05.zip).
Las **fechas de subida a Moodle del fichero** son las siguientes:

- Los alumnos de ***Evaluación Continua, la semana del 20 de marzo*** *(cada grupo puede realizar la entrega hasta las 23:30 h. de la noche anterior a su clase de prácticas*).
- Los alumnos de ***Evaluación Final,*** según lo especificado en la normativa.

INTRODUCCIÓN

Una **pila** (*stack* en inglés) es un tipo abstracto de datos (TAD) definido como una secuencia de datos homogéneos, **de carácter genérico**, ordenados implícitamente, en la que el modo de acceso a dichos datos

sigue una estrategia de tipo LIFO (del inglés *Last In First Out*), esto es, el último dato en entrar es el primero en salir.
En una pila hay dos operaciones básicas para el manejo de los datos almacenados en ella:

- **Apilar** (*Push*): Añade un elemento a la pila; el último elemento añadido siempre ocupa la cima o tope de la pila.
- **Desapilar** (*Pop*): Retira de la pila el último elemento apilado, es decir el que ocupa la cima de la pila.

Otras primitivas típicas del TAD Pila son las siguientes:

- Inicializar la pila.
- Comprobar si la pila está vacía.
- Comprobar si la pila está llena.
- Liberar la pila.

En esta práctica se implementará el TAD Pila en el lenguaje C eligiendo las estructuras de datos adecuadas, y se programarán las funciones asociadas a las principales primitivas para operar con la pila. Posteriormente, el TAD Pila programado se utilizará para resolver varios problemas.

Según la forma de implementar la pila que se ha visto en teoría, al apilar se reserva memoria y se introduce una copia del elemento en la pila, mientras que al desapilar basta con devolver el (último) elemento tal cual estaba guardado, sin necesidad de copiar ni liberar memoria dentro de la función de desapilar.

PARTE 1. EJERCICIOS

Ejercicio 1 (P2_E1). Implementación y prueba del TAD Pila (Stack) como array de punteros a EleStack

1. Definición del tipo de dato Pila (Stack). Implementación: selección de estructura de datos e implementación de primitivas

Se pide implementar en el fichero **stack_elestack.c** las primitivas del TAD Pila definido en el archivo de cabecera **stack_elestack.h** el cual, a su vez, hace uso del TAD EleStack definido en **elestack.h**. Los tipos Status y Bool son los definidos en **types.h**. La constante MAXSTACK indica el tamaño máximo de la pila. Su valor debe permitir desarrollar los ejercicios propuestos. La estructura de datos elegida para implementar el TAD PILA consiste en un array de punteros a elementos de tipo genérico (EleStack) y un entero que almacena el índice (en el array) del elemento situado en la cima de la pila, es decir, el índice que ocupa el último elemento almacenado en la pila. Dicha estructura se muestra a continuación:

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

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

Los prototipos de las primitivas de **stack_elestack.c** se incluyen en el apéndice 1.

2. **Encapsulamiento del comportamiento tipo PILA**

Para encapsular el comportamiento de la Pila y lograr que se puedan crear pilas de distintos tipos (**aunque no a la vez**), en esta versión de la Pila se trabaja con elementos de tipo EleStack. En este primer ejercicio vamos a trabajar con pilas de Nodos para poder comprobar su correcto funcionamiento. Para ello hay que:

- Definir el **tipo EleStack como un TAD opaco** en el fichero **elestack.h** (ver apéndice 2):

```
typedef struct _EleStack EleStack;
```

- Definir el **la estructura _EleStack como un envoltorio para nodo** en el fichero **elestack-node.c**:

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

En este último fichero .c hay que implementar las funciones cuyos prototipos se encuentran en **elestack.h**.

Nótese que para lograr una abstracción mayor, varias de estas funciones usan un puntero a void (void *) lo cual permite o bien devolver un puntero de cualquier tipo (por ejemplo, en *elestack_getInfo*) o guardar un puntero a cualquier dato en EleStack (por ejemplo, en la función *elestack_setInfo*).

3. Comprobación de la corrección de la definición del tipo Stack y sus primitivas

Con el objetivo de evaluar el funcionamiento de las primitivas anteriores, desarrollar un programa de prueba **p2_e1.c** que trabaje con pilas de nodos. Utilizad el TAD Nodo de la práctica 1. Este programa deberá:

- Declarar e inicializar un nodo, un elemento de pila y una pila.

- Asignar al nodo el nombre “first” e id 111, e insertarlo en la pila.

- Asignar al nodo el nombre “second” e id 222, e insertarlo en la pila.

- Imprimir el contenido de la pila, indicando al final el número total de caracteres imprimidos.

- Extraer los elementos de la pila iterativamente hasta que la pila esté vacía. En cada iteración se imprimirá el elemento extraído de la pila.

- Imprimir el contenido de la pila, indicando al final el número total de caracteres imprimidos.

```
> ./p2_e1
Contenido de la pila:
[second, 222, 0]
[first, 111, 0]
Caracteres imprimidos: 32
Vaciando pila. Extracciones:
[second, 222, 0] [first, 111, 0]
Contenido de la pila después de vaciar:
Caracteres imprimidos: 0
```

Ejercicio 2 (P2_E2). Implementación y prueba del TAD Pila (Stack) con EleStack como envoltorio del tipo int

1. Modificación de los ficheros .h y .c que consideres necesarios

En este ejercicio haremos las modificaciones necesarias en determinados ficheros para que ahora el tipo de dato que maneje la pila sea de tipo **int**. Esto es:

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

Explica en la memoria qué ficheros has modificado o creado y por qué o para qué.

2. Comprobación de la corrección de la definición del tipo Stack y sus primitivas

Con el objetivo de evaluar el funcionamiento de las primitivas anteriores, desarrollar un programa **p2_e2.c** que:

1º) Reciba como argumento un entero positivo
2º) Cree una pila e introduzca en ella los números enteros desde el 0 hasta el nº recibido
3º) Imprima la pila por la salida estándar
4º) Ejecute una función derivada que reciba una pila de enteros y devuelva como un *double* el valor medio de sus elementos.
5º) Imprima por la salida estándar el valor de la media y la pila.

Nota: La función que calcula la media NO debe modificar la pila. Se deben gestionar los errores en todo momento. Asumir que un push precedido de un pop no causa error.

Incluye en la memoria el pseudocódigo de la función que calcula la media.

```
> ./p2_e2 3
Pila antes de la llamada a la función:
[3]
[2]
[1]
[0]
La media es 1,50000
Pila después de la llamada a la función:
[3]
[2]
[1]
[0]
```

Ejercicio 3 (P2_E3). Generalización del TAD Pila (Stack)

1. Modificación de las primitivas de pila

En este ejercicio modificaremos algunas funciones del TAD Pila para que pueda **almacenar cualquier tipo de dato de manera dinámica**, lo cual permitirá, en particular, tener varias pilas a la vez que almacenan elementos distintos (ej: una pila de nodos y una pila de enteros).

Para poder resolver este ejercicio se hará uso de punteros a funciones.

Los punteros a funciones permiten pasar una función como argumento de otra función e invocarla cuando sea preciso. Esto nos será útil, ya que la implementación de pila que estamos usando hasta ahora sólo necesita conocer el prototipo de tres funciones relativas a los elementos que almacena:

- La función que copia un elemento, puesto que stack_push llama a elestack_copy.
- La función que imprime un elemento, pues la función que imprime la pila (stack_print) llama a la función elestack_print para imprimir cada elemento.
- La función que destruye un elemento, pues la función que destruye la pila llama a elestack_destroy para liberar cada uno de los elementos.

El problema radica en que la estructura EleStack puede almacenar la dirección de cualquier tipo de objeto: en el primer ejercicio de esta práctica almacena la dirección de un nodo (Node *), mientras que en el segundo almacena la dirección de un entero (int *).

El compilador, en el momento de enlazar los diferentes módulos, necesita conocer exactamente tanto la definición de la estructura de los elementos (EleStack) como las definiciones de las funciones declaradas en EleStack.h (como habéis hecho en los ejercicios anteriores, primero con nodos y luego con enteros).

Esto impide que la implementación sea, de verdad, genérica: cuando desde la pila hay que copiar un elemento, imprimirlo o destruirlo, en la implementación de pila que tenemos hasta ahora hay que llamar a la función correspondiente del TAD EleStack, que a su vez tendrá que llamar a la función específica del objeto encapsulado en la estructura (struct _EleStack). Esto obliga a utilizar diferentes funciones dependiendo del tipo de datos que almacenemos en la pila (imprimir enteros no es igual que imprimir nodos, por ejemplo). La consecuencia es que, en esta implementación, además de ser engorrosa de gestionar, es imposible que en el mismo programa principal puedan convivir pilas de diferentes tipos de objetos (p. ej., una pila de enteros y otra de nodos).

La generalización que se plantea en este ejercicio considera que las funciones necesarias para estas tareas (solo para las que dependen del tipo de información que se quiere guardar en la pila) simplemente son atributos adicionales de la pila, a los que habrá que asignar valores en el momento de inicializar la pila. Esto es posible gracias a que C permite declarar variables de tipo puntero a función. En este caso la estructura para implementar la pila será:

```
struct _Stack{
    void * item[MAXSTACK];
    int top;

    P_stack_ele_destroy pf_destroy;
    P_stack_ele_copy  pf_copy;
    P_stack_ele_print pf_print;
};
```

Como se observa, además de las variables *item* y *top* habituales, aparecen otras tres variables llamadas *pf_destroy*, *pf_copy* y *pf_print*. Estas tres variables son, respectivamente, del tipo `P_stack_ele_destroy`, `P_stack_ele_copy` y `P_stack_ele_print`.

Estos tipos (`P_stack_ele_destroy`, `P_stack_ele_copy` y `P_stack_ele_print`) se han definido con la correspondiente sentencia `typedef` en el fichero `stack_fp.h`. incluido en el apéndice 3:

```
typedef void (*P_stack_ele_destroy)(void*);
typedef void* (*P_stack_ele_copy)(const void*);
typedef int (*P_stack_ele_print)(FILE *, const void*);
```

En la primera declaración estamos afirmando que en `P_stack_ele_destroy` almacenaremos la dirección de una función que reciba un puntero genérico (`void *)` y devuelva un objeto de tipo `void`. En `P_stack_ele_copy` guardaremos la dirección de una función que recibe un puntero genérico y devuelve la dirección de un dato sin especificar de qué tipo es ese dato (esto es, la función devuelve un objeto de tipo `void *`). Finalmente, En `P_stack_ele_print` guardaremos la dirección de una función que recibe un puntero a un fichero abierto y un puntero genérico y devuelve un entero.

En el apéndice 5 se muestran dos ejemplos de cómo utilizar los punteros a función (ver funciones *`stack_ini`* y *`stack_push`* del fichero *`stack_fp.c`*).

En este ejercicio debéis completar el fichero *`stack_fp.c`* y repetir los ejercicios 1 y 2 utilizando la nueva implementación de pila.

Ejercicio 4 (P2_E4). Búsqueda de caminos en un grafo utilizando la implementación genérica de Pila

En este ejercicio recorreremos el grafo de forma que se pueda llegar desde un nodo a otro visitando los nodos que se encuentran entre los mismos. Para ello utilizaremos los TADs **Graph, Node**, y **Stack.** Para este último utilizaremos la implementación genérica basada en punteros a funciones.

1. **Implementación del algoritmo para recorrer un grafo**

Para comenzar, es necesario definir una función que permita recorrer un grafo dado a partir de un nodo origen. Podéis partir del siguiente fragmento de pseudocódigo (ojo, no está completo → pensar):

```
graph_findDeepSearch (Graph g, Node v, Node to):
        push (P, v)
        while isEmpty (P)== FALSE:
                u ← pop (P)
                si getLabel (u) == "Blanco":
                        setLabel (u, "Negro")
                        for each neighbor w from u:
                                if (w==to) return EXIT
                                else if getLabel (w) == "Blanco"
                                        push (P, w)
```

2. **Aplicación del algoritmo a un grafo**

Utilizando como base el pseudocódigo dado para recorrer un grafo, realizad las modificaciones necesarias para responder a la pregunta de si **es posible encontrar un camino entre un nodo origen y un nodo destino para un grafo dado**. Si lo es, implementa la función correspondiente, de prototipo:

```
Node *graph_findDeepSearch (Graph *g, int from_id, int to_id);
```

Escribe un programa **p2_e4.c** que reciba como argumento un fichero donde se almacena un grafo (como en los ejercicios anteriores) y los identificadores de dos nodos (primero el del nodo origen y después el del nodo destino) e imprima el mensaje "Es posible encontrar un camino" o "No es posible encontrar un camino", según sea el resultado que devuelve la función de búsqueda de caminos en un grafo implementada en este ejercicio.

P. ej, en el grafo descrito en "g1.txt" no es posible ir del nodo 3 al nodo 1, pero sí del 1 al 3 o del 1 al 2.

Se sugiere utilizar las siguientes estructuras y tipos de datos:

```
/* En node.h */
typedef enum {
    BLANCO, NEGRO
} Label;

/* en node.c */
#define NAME_L 64
struct _Node {
 char name[NAME_L];  // nombre del nodo
 int id;             // id del nodo
 int nConnect;       // número de conexiones que parten del nodo

 Label etq;         // estado del nodo
 int antecesor_id;  // solo necesario para el ejercicio opcional
};
```

Nota: Deben incluirse en node.h graph.h las funciones "set" y "get" para cada uno de los atributos de las estructuras anteriores.

3. **(opcional) Devolver el camino encontrado**

Opcionalmente y a partir del ejercicio anterior, mostrad por pantalla (en caso de que sea posible encontrar un camino) el camino válido que permita ir entre los nodos origen y destino, imprimiendo un nodo del grafo por línea.

Nota: Se puede resolver utilizando una función recursiva o bien una pila adicional en la que se almacenen los ids de los nodos.

Discute la solución adoptada en la memoria.

PARTE 2. MEMORIA

Responded las siguientes preguntas y adjuntad las respuestas en el fichero .PDF correspondiente a la memoria de esta práctica, que ha de incluirse en el fichero .ZIP que entreguéis.

1. Una aplicación habitual del TAD PILA es para evaluar expresiones posfijo. Describe en detalle qué TADs habría que definir/modificar para poder adaptar la pila de enteros (P3_E1) de modo que tengamos una que permita evaluar expresiones posfijo.

2. Explicad las decisiones de diseño y alternativas que se han considerado durante los distintos ejercicios de la práctica. En particular, explicad las claves de la implementación y de las decisiones de diseño de los ejercicios **P2_E2**, **P2_E3** y **P2_E4**.

3. **CONCLUSIONES FINALES**

   Incluir, al final de la memoria, unas breves conclusiones sobre la práctica, indicando los beneficios que os ha aportado la práctica, qué aspectos vistos en las clases de teoría han sido reforzados, qué apartados de la práctica han sido más fácilmente resolubles y cuáles han sido los más complicados (o no se han podido resolver finalmente), qué aspectos nuevos de programación se han aprendido, qué dificultades se han encontrado, etc.

Apéndice 1: stack_elestack.h

```
typedef struct _Stack Stack;

/**----------------------------------------------------------------
Inicializa la pila reservando memoria. Salida: NULL si ha habido error o el puntero a la pila si ha ido bien
----------------------------------------------------------------*/
Stack * stack_ini();

/**----------------------------------------------------------------
Elimina la pila Entrada: puntero a la pila que se desea eliminar
----------------------------------------------------------------*/
void stack_destroy(Stack *);

/**----------------------------------------------------------------
Inserta un elemento en la pila. Entrada: elemento a insertar y pila donde insertarlo. Salida: OK si logra insertarlo o ERROR si no.
 ----------------------------------------------------------------*/
Status stack_push(Stack *, const EleStack *);

/**----------------------------------------------------------------
Extrae un elemento de la pila. Entrada: la pila de donde extraerlo. Salida: NULL si no logra extraerlo, o el elemento extraído si lo logra. Nótese que la pila quedará modificada.
----------------------------------------------------------------*/
EleStack * stack_pop(Stack *);

/**----------------------------------------------------------------
Comprueba si la pila esta vacía. Entrada: puntero a la pila. Salida: TRUE si está vacía o FALSE si no lo esta
----------------------------------------------------------------*/
Bool stack_isEmpty(const Stack *);

/**----------------------------------------------------------------
Comprueba si la pila esta llena. Entrada: puntero a la pila. Salida: TRUE si está llena o FALSE si no lo esta
----------------------------------------------------------------*/
Bool stack_isFull(const Stack *);

/**----------------------------------------------------------------
Imprime toda la pila, comenzando por el último elemento insertado (aquel en la cima o tope) y terminando por el primero que se insertó. Imprime un elemento por línea. Entrada: pila y fichero abierto donde imprimir. Salida: número de caracteres escritos en total.
----------------------------------------------------------------*/
int stack_print(FILE*, const Stack*);
```

Apéndice 2: elestack.h

```
typedef struct _EleStack EleStack;

/**-------------------------------------------------------------------
Inicializa un elemento de pila. Salida: Puntero al elemento inicializado o NULL en caso de error
-------------------------------------------------------------------*/
EleStack * EleStack_ini();

/**-------------------------------------------------------------------
Elimina un elemento de pila. Entrada: elemento a destruir.
-------------------------------------------------------------------*/
void EleStack_destroy(EleStack *);

/**-------------------------------------------------------------------
Modifica los datos de un elemento de pila o EleStack. Entrada: El puntero al elemento a modificar y el contenido a
guardar en dicho elemento. Salida: OK o ERROR
-------------------------------------------------------------------*/
Status EleStack_setInfo(EleStack *, void*);

/**-------------------------------------------------------------------
Devuelve el contenido almacenado en un elemento de pila (del tipo que sea). Entrada: El elemento de pila. Salida: El
contenido de ese elemento, o NULL si ha habido error.
-------------------------------------------------------------------*/
void * EleStack_getInfo(EleStack *);

/**-------------------------------------------------------------------
Copia un elemento de pila en otro, reservando memoria. Entrada: el elemento a copiar. Salida: puntero al nuevo
elemento, copia del recibido, o NULL en caso de error.
-------------------------------------------------------------------*/
EleStack * EleStack_copy(const EleStack *);

/**-------------------------------------------------------------------
Compara dos elementos de pila (su contenido). Entrada: los dos elementos a comparar. Salida: TRUE en caso de ser
iguales y FALSE en caso contrario.
-------------------------------------------------------------------*/
Bool EleStack_equals(const EleStack *, const EleStack *);

/**-------------------------------------------------------------------
Imprime un elemento de pila en un fichero ya abierto. Entrada: Fichero en el que se imprime y elemento a imprimir.
Salida: número de caracteres escritos en total.
-------------------------------------------------------------------*/
int EleStack_print(FILE *, const EleStack *);
```

Apéndice 3: stack_fp.h

```
#ifndef   STACK_H
#define  STACK_H

#include "types.h"

typedef struct _Stack Stack;

/* Tipos de los punteros a función soportados por la pila */
typedef void (*P_stack_ele_destroy)(void*);
typedef void* (*P_stack_ele_copy)(const void*);
typedef int (*P_stack_ele_print)(FILE *, const void*);

/* La función stack_ini recibirá los valores para los campos de la pila que son punteros a función, es decir, recibirá las
funciones para, respectivamente, destruir, copiar e imprimir los elementos de la pila creada*/

Stack * stack_ini (P_stack_ele_destroy f1, P_stack_ele_copy f2, P_stack_ele_print f3);
void stack_destroy(Stack *);

Status stack_push(Stack *, const void *);
void * stack_pop(Stack *);

Bool stack_isEmpty(const Stack *);
Bool stack_isFull(const Stack *);

int stack_print(FILE*, const Stack*);
```

Apéndice 4: stack_fp.c (ejemplo incompleto)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

#include "stack_fp.h"

#define MAXSTACK 100
#define EMPTY_STACK -1

extern int errno;

struct _Stack {
    int top;
    void * item[MAXSTACK];

    P_stack_ele_destroy  pf_destroy;
    P_stack_ele_copy   pf_copy;
    P_stack_ele_print  pf_print;
};

Stack *stack_ini (P_stack_ele_destroy  fd,  P_stack_ele_copy  fc,  P_stack_ele_print  fp) {
    Stack *s;
    int i;

    s = (Stack*) malloc(sizeof(Stack));
    if (!s) {
        fprintf(stderr, "%s", strerror(errno));
        return NULL;
    }
    // Inicializo tope y asigno los punteros a función
    s->top = -1;
    s->pf_copy = fc;
    s->pf_destroy = fd;
    s->pf_print = fp;

    // Asigno los punteros de los elementos
    for (i=0; i< MAXSTACK; i++)
        s->item[i] = NULL;

    return s;
}
```

```
void stack_destroy(Stack* stc) {
    if (!stc) return;

    while (stc->top != EMPTY_STACK) {
        stc->pf_destroy( stc→item[stc→top]);   // Llamada a la función cuyo puntero se almacenó como campo
                                               // de la pila, es decir, utilizando el puntero a la función
        stc->top --;
    }
    free(stc);
}
```