Programación II, 2015-2016
Escuela Politécnica Superior, UAM

Práctica 2: TAD Pila

OBJETIVOS

- Uso del TAD Punto y Laberinto.
- Familiarización con el TAD Pila (LIFO), conocer 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 algunos problemas de programación.

NORMAS

Igual que en la práctica 1, 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.
- La ***memoria*** que se entregue 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 y P2_E3.

Semana 3: 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 7 de marzo*** *(cada grupo puede realizar la entrega hasta las 23:30 h. de la noche anterior a su clase de prácticas*), excepto los grupos 2112 y 2162 que entregan el 16 de marzo para recuperar el día de clase perdido (25 de febrero). Los alumnos del 2163 sí entregan el jueves 10 a pesar de no tener clase el viernes 11 de marzo.
- 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 un conjunto de datos homogéneos, de carácter genérico, ordenados implícitamente, en el que el modo de acceso a dichos elementos es de tipo LIFO (del inglés *Last In First Out*, es decir el último en entrar es el primero en salir). En una pila hay dos operaciones básicas para el manejo de los datos:

- **Apilar** (*Push*): Añade un elemento a la pila; el último elemento añadido siempre ocupa la cima 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 en el lenguaje C el TAD Pila eligiendo las estructuras de datos adecuadas, y se programarán las funciones que implementen las principales primitivas para operar con la pila. El TAD Pila programado se utilizará para resolver varios problemas.

Tal como se ha visto en teoría al apilar hay que reservar memoria en la pila, mientras que al desapilar basta con devolver el elemento tal cual estaba guardado (no haría falta copiar ni liberar memoria).

PARTE 1. EJERCICIOS

Ejercicio 1 (P2_E1). Implementación y prueba del TAD Pila (Stack)

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.c las primitivas del TAD Pila que se definen en el archivo de cabecera stack.h el cual, a su vez, hace uso de los elementos de 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 elementos de tipo genérico y un entero que almacena el índice (en el array) del elemento situado en la cima de la pila. Dicha estructura se muestra a continuación:

```
#define MAXSTACK 100

/* En stack.h */
typedef struct _Stack Stack;

/* En stack.c */
struct _Stack {
        int top;
        EleStack* item[MAXSTACK];
};
```

Las primitivas de stack.c se incluyen en el apéndice 1.

2. Encapsulación del comportamiento tipo PILA.

Para encapsular el comportamiento de la Pila y lograr que se puedan crear pilas de distintos tipos, en la Pila se trabaja con elementos de tipo EleStack. En este primer ejercicio vamos a trabajar con pilas de enteros para poder comprobar su correcto funcionamiento, por tanto:

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

```
typedef struct _EleStack EleStack;
```

- Definid el **tipo _EleStack como un envoltorio de enteros** en el fichero elestack-int.c:

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

En este último fichero hay que implementar las funciones definidas en elestack.h.

Nótese que para lograr una abstracción mayor, dos de estas funciones usan un puntero a void (‘void *’) lo cual permite o bien devolver un puntero de cualquier tipo (*elestack_getInfo*) o añadir un puntero a cualquier tipo de dato a EleStack (*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, se desarrollará un programa **p2_e1.c** que trabajará con pilas de enteros. Este programa recibirá como argumento un número, que indicará cuántos enteros se tienen que leer por teclado, y los introducirá uno a uno en una pila. A continuación, utilizando la pila creada y las primitivas del TAD Pila, el programa crea dos pilas: una de números pares y otra con números impares. Durante el proceso también llamará al resto de funciones de su prototipo (mirar ejemplo más abajo e **imprimir exactamente eso para esos datos**). Al final imprimirá el contenido de las dos pilas y liberará todos los recursos. A continuación se muestra un ejemplo de ejecución de dicho programa:

```
> ./p2_e1 3
Pila total (no llena, vacía):

Introduce número: 1
Introduce número: 2
Introduce número: 3

Pila total (no llena, no vacía):
[3]
[2]
[1]

Imprimiendo la pila (no llena, no vacía) con números pares:
[2]
Imprimiendo la pila (no llena, no vacía) con números impares:
[1]
[3]

Pila total (no llena, vacía):
```

Ejercicio 2 (P2_E2). Implementación y prueba del TAD Pila (Stack) con EleStack de tipo Point

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 **Point**. Explica en la memoria final qué ficheros has modificado o creado y por 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, se desarrollará un programa **p2_e2.c** que reciba como argumento un fichero que representa un laberinto. Después de leer y cargar dicho laberinto (como en el ejercicio P1_E2), se accederán a las primitivas de **Maze** para ir recorriéndolo y obteniendo todos los puntos que lo forman, de manera que se inserten en una pila. Al terminar, se imprime la pila resultante (un punto por cada línea) y se liberan todos los recursos. A continuación se muestra un ejemplo de ejecución de dicho programa:

```
> ./p2_e2 “m1.txt”
[(5, 6): +]
[(4, 6): +]
[(3, 6): +]
[(2, 6): +]
[(1, 6): +]
[(0, 6): +]
[(5, 5): +]
[(4, 5):  ]
[(3, 5): +]
[(2, 5): +]
[(1, 5): o]
[(0, 5): +]
...
[(0, 0): +]
```

Ejercicio 3 (P2_E3). Resolución del laberinto

En este ejercicio recorreremos el laberinto de forma que se visiten todas sus casillas accediendo a los prototipos de **Maze** y **Point**.

1. Implementación del algoritmo para recorrer un laberinto.

Para comenzar, se tiene que definir una función que permita recorrer un laberinto dado. Podréis partir del siguiente pseudocódigo:

```
recorrer(Maze m, Point o):
    almacenar(S, o)
    mientras que S no es vacío:
        v ← sacar(s)
        si v no ha sido descubierto:
            etiquetar(v, "descubierto")
            para cada vecino w de v:
                almacenar(S, w)
```

2. Aplicación del algoritmo a un laberinto.

Utilizando como base el algoritmo implementado de recorrer un laberinto, realizad las modificaciones necesarias para resolver un laberinto dado, es decir, responder a la pregunta de si **es posible encontrar un camino entre la casilla de entrada y la de salida**. Este programa se escribirá en un fichero de nombre **p2_e3.c**.

El programa recibe como argumento un fichero donde se almacena un laberinto (como en el ejercicio anterior) e imprimirá el mensaje "Es posible encontrar un camino" o "No es posible encontrar un camino" según sea la salida de la resolución del laberinto usando el algoritmo implementado en este ejercicio.

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 las casillas de entrada y salida, imprimiendo un punto del camino por línea. Discute la solución obtenida en la memoria.

Ejercicio 4 (P2_E4). Modificación del TAD Pila (Stack) con un puntero como cima

En este ejercicio realizaremos las modificaciones necesarias para que la cima de la pila sea ahora de tipo puntero en lugar de un entero. La cima (*top*) apuntará a la última posición del array que contenga un dato (es decir, guardará la dirección de dicha posición).

1. Definición del tipo de dato Pila (Stack).

Para comenzar, se tiene que definir en stack-pointer.c la nueva estructura elegida para el tipo abstracto de datos previamente declarado en stack.h:

```
struct _Stack{
        EleStack** top;
        EleStack* item[MAXSTACK];
}
```

2. Modificación de las primitivas necesarias.

Utilizando como base el ejercicio anterior, realizad las modificaciones necesarias en los ficheros .h y .c que consideréis necesarios para el correcto funcionamiento del ejercicio.

3. Comprobación de la corrección del ejercicio.

Con el objetivo de evaluar el funcionamiento de las primitivas anteriores, se generará un nuevo ejecutable usando el mismo programa main del ejercicio 3 (**p2_e3.c**) pero que utilice el nuevo tipo de dato. La salida esperada debe ser exactamente la misma.
Razonad si tenéis que realizar modificaciones en dicho programa main y, en caso positivo, entregadlo en un fichero de nombre **p2_e4.c** (discutiendo dichas modificaciones en la memoria).

PARTE 2.A. PREGUNTAS SOBRE LA PRÁCTICA

Responded a las siguientes preguntas **completando el fichero disponible en el .zip**. Renombrad el fichero para que se corresponda con vuestro nº de grupo y pareja de prácticas y adjuntadlo al fichero .zip que entreguéis.

1. En el ejercicio 4 hemos realizado una serie de modificaciones para que la cima de la pila sea de tipo **puntero**. Indicad qué ficheros habéis tenido que modificar y cuáles han sido estos cambios:

En **elestack.h**, antes:	Ahora:
En **elestack.c**, antes:	Ahora:
En **stack.h,** antes:	Ahora:
En **stack.c,** antes:	Ahora:
En **p2_e3.c**, antes:	Ahora:

2. 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 (P2_E1) o de puntos (P2_E2) en una que permita evaluar expresiones posfijo.

PARTE 2.B. MEMORIA SOBRE LA PRÁCTICA

Explica las decisiones de diseño y alternativas que se han considerado durante la práctica. En particular, explica en detalle la implementación y decisiones de diseño de los ejercicios **P2_E2** y **P2_E3**.

CONCLUSIONES FINALES

Se reflejará 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, qué aspectos nuevos de programación se han aprendido, dificultades, etc.

Apéndice 1: stack.h

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

/**-------------------------------------------------------------------
Inicializa la pila reservando memoria. Salida: NULL si ha habido error o la pila si ha ido bien
-------------------------------------------------------------------*/
Stack * stack_ini();
/**-------------------------------------------------------------------
Elimina la pila Entrada: la pila que se va a eliminar
-------------------------------------------------------------------*/
void stack_free(Stack *);

/**-------------------------------------------------------------------
Inserta un elemento en la pila. Entrada: un elemento y la pila donde insertarlo. Salida: NULL si no logra
insertarlo o la pila resultante si lo logra
-------------------------------------------------------------------*/
Stack * stack_push(Stack *, const EleStack *);
/**-------------------------------------------------------------------
Extrae un elemento en la pila. Entrada: la pila de donde extraerlo. Salida: NULL si no logra extraerlo o el
elemento extraido si lo logra. Nótese que la pila quedará modificada
-------------------------------------------------------------------*/
EleStack * stack_pop(Stack *);
/**-------------------------------------------------------------------
Copia un elemento (reservando memoria) sin modificar el top de la pila. Entrada: la pila de donde copiarlo.
Salida: NULL si no logra copiarlo o el elemento si lo logra
-------------------------------------------------------------------*/
EleStack * stack_top(const Stack *);
/**-------------------------------------------------------------------
Comprueba si la pila esta vacia. Entrada: pila. Salida: TRUE si está vacia o FALSE si no lo esta
-------------------------------------------------------------------*/
Bool stack_isEmpty(const Stack *);
/**-------------------------------------------------------------------
Comprueba si la pila esta llena. Entrada: pila. Salida: TRUE si está llena o FALSE si no lo esta
-------------------------------------------------------------------*/
Bool stack_isFull(const Stack *);
/**-------------------------------------------------------------------
Imprime toda la pila, colocando el elemento en la cima al principio de la impresión (y un elemento por línea).
Entrada: pila y fichero donde imprimirla. Salida: Devuelve el número de caracteres escritos.
-------------------------------------------------------------------*/
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. Entrada: Elemento a destruir.
-------------------------------------------------------------------*/
void elestack_free(EleStack *);

/**-------------------------------------------------------------------
Modifica los datos de un elemento. Entrada: El elemento a modificar y el contenido a guardar en dicho
elemento. Salida: El elemento a modificar o NULL si ha habido error.
-------------------------------------------------------------------*/
EleStack * elestack_setInfo(EleStack *, void*);
/**-------------------------------------------------------------------
Devuelve el contenido de EleStack. Entrada: El elemento. Salida: El contenido de EleStack o NULL si ha
habido error.
-------------------------------------------------------------------*/
void * elestack_getInfo(EleStack *);

/**-------------------------------------------------------------------
Copia un elemento en otro, reservando memoria. Entrada: el elemento a copiar. Salida: Devuelve un puntero
al elemento copiado o NULL en caso de error.
-------------------------------------------------------------------*/
EleStack * elestack_copy(const EleStack *);
/**-------------------------------------------------------------------
Compara dos elementos. Entrada: dos elementos a comparar. Salida: Devuelve TRUE en caso de ser iguales
y si no FALSE
-------------------------------------------------------------------*/
Bool elestack_equals(const EleStack *, const EleStack *);

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