Programación II, 2016-2017
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 ésta 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 y comienzo de P2_E3.

Semana 3: código de P2_E1, P2_E2, P2_E3 y comienzo de P2_E4.

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 en la pila, mientras que al desapilar basta con devolver el (último) dato tal cual estaba guardado, sin necesidad de copiar y liberar memoria dentro de la función primitiva.

PARTE 1. EJERCICIOS

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

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_element.c** las primitivas del TAD Pila que se definen en el archivo de cabecera **stack_element.h** el cual, a su vez, hace uso de los elementos de **element.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_element.h */
typedef struct _Stack Stack;

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

Las primitivas de **stack_element.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 (**aunque no a la vez**), en esta versión de la Pila se trabaja con elementos de tipo Element. En este primer ejercicio vamos a trabajar con pilas de enteros para poder comprobar su correcto funcionamiento, por tanto:

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

```
typedef struct _Element Element;
```

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

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

En este último fichero hay que implementar las funciones definidas en **element.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 (*element_getInfo*) o añadir un puntero a cualquier tipo de dato a Element (*element_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 primitivas (mirar ejemplo más abajo e **imprimir exactamente eso para esos datos**[1]). 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):
```

[1] Aunque te parezca un detalle menor, cuando se especifica una librería C es fundamental mantener las especificaciones incluso en el formato de las salidas. En este sentido el no generar exactamente la misma salida que se describe se considerará un error grave que implicará la pérdida de la puntuación asignada a la impresión de los tipos de dato.

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

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 **Node**. 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 grafo (podéis utilizar la función **read_graph_from_file** que aparece en **graph_test.c** de la práctica 1). Después de leer y cargar dicho grafo, se accederán a las primitivas de **Graph** para ir recorriéndolo y obteniendo todas las conexiones que existen en dicho grafo (por cada nodo, hay que insertar sus conexiones entrantes y sus conexiones salientes, de una manera similar a como lo hacía la función **format_graph** de **graph_test.c**, pero en lugar de imprimir los nodos, habría que insertarlos en la pila), de manera que se inserten en una pila. Al terminar, se imprime la pila resultante (un nodo por cada línea) y se liberan todos los recursos. A continuación se muestra un ejemplo de ejecución de dicho programa:

```
> ./p2_e2 “g1.txt”
[2, b]
[1, a]
[3, c]
[1, a]
[3, c]
[2, b]
```

Ejercicio 3 (P2_E3). Búsqueda de caminos en un grafo

En este ejercicio recorreremos el grafo de forma que se visiten sus nodos entre dos nodos dados accediendo a las primitivas de **Graph** y **Node**, y haciendo uso de una pila para para el recorrido.

1. Implementación del algoritmo para recorrer un grafo

Para comenzar, se tiene que definir una función que permita recorrer un grafo dado a partir de un nodo origen. Podréis partir del siguiente pseudocódigo:

```
recorrer(Graph g, Node n):
        almacenar(S, n)
        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 grafo

Utilizando como base el pseudocódigo indicado de recorrer un grafo, realizad las modificaciones necesarias para responder a la pregunta de si **es posible encontrar un camino entre un nodo origen y el nodo destino para un grafo dado**. Este programa se escribirá en un fichero de nombre **p2_e3.c**.

El programa recibe como argumento un fichero donde se almacena un grafo (como en los ejercicios anteriores) y los identificadores de dos nodos (primero el de origen y después el de destino) e imprimirá el mensaje "Es posible encontrar un camino" o "No es posible encontrar un camino" según sea la salida de la búsqueda de caminos en un grafo usando el algoritmo implementado en este ejercicio.

Por ejemplo, en el grafo codificado en el fichero "g1.txt" no es posible ir del nodo 3 al nodo 1, pero sí del 1 al 3 o del 1 al 2.

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 de origen y destino, imprimiendo un nodo del grafo por línea. Discute la solución obtenida en la memoria.

Ejercicio 4 (P2_E4). Generalización del TAD Pila (Stack)

1. Modificación de las primitivas de pila

En este ejercicio modificaremos algunas funciones del TAD de la 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.

Para poder resolver este ejercicio se hará uso de los punteros a función. Los punteros a función nos permiten pasar una función como argumento e invocarla cuando sea preciso. Esto nos será útil ya que la implementación de pila que estamos usando hasta ahora sólo precisa conocer tres operaciones de los elementos que almacena, por ejemplo:

- Cómo copiar un elemento (tanto al insertarlo en la pila como al hacer top)
- Cómo imprimir un elemento (al imprimir la pila)
- Cómo destruir un elemento (al destruir la pila)

En la implementación actual, en estas tres situaciones se hace uso de las primitivas de Element. El problema, sin embargo, es que aunque Element puede almacenar cualquier puntero en su estructura, en el momento de enlazar para generar el programa final, el compilador precisa conocer cuál es la definición de la estructura de Element y cómo se van a definir las funciones declaradas en element.h (como habéis hecho en los ejercicios anteriores, primero con element-int.c y después con element-node.c). Esto impide que, de hecho, la implementación usada hasta ahora sea, de verdad, genérica, como se describe a continuación.

Cuando desde la pila hay que copiar un elemento, imprimirlo o destruirlo, en la implementación actual hay que llamar específicamente a una función del TAD Element (por ejemplo, en **stack_destroy**, cada elemento se libera con la llamada **element_destroy**, en **stack_print** cada elemento se imprime con la llamada a la función **element_print** y en **stack_push** el elemento se copia usando **element_copy**). Esto obliga a enlazar con diferentes funciones si la pila es de enteros o de nodos haciendo imposible que en el mismo programa principal se puedan tener simultáneamente pilas de enteros y de nodos. La generalización que se plantea en este ejercicio considera que las funciones necesarias para estas tareas (y que dependen del tipo de información que se quiere guardar en la pila) sólo son un dato más que la pila necesita almacenar. Esto es posible gracias a que C permite declarar variables de tipo puntero a función. La nueva pila que hay que programar debe contener de forma explícita las funciones que va a utilizar en los momentos descritos antes.

Por ello, en el momento de creación de la pila se aportan esas funciones y el código se transforma para que la pila llame a las funciones que tiene almacenadas cuando lo necesite. De esta forma, las llamadas que se han destacado antes se transforman en las siguientes en la nueva pila: en la función **stack_destroy** se llamará a la función guardada en **stc->destroy_element_function** (obsérvese que el atributo de la pila que guarda la función de destrucción se llama *destroy_element_function*); en la función **stack_print** la llamada que habría que realizar sería **stc->stack_elestack_print** y de manera similar al apilar elementos (usando **stc->copy_element_function**). De esta forma se consigue una pila realmente genérica.

Como se ha explicado, la nueva definición de la pila deberá almacenar los punteros a función en la propia estructura para poder usarlos posteriormente; esto lo haremos en los ficheros **stack_fp.h** y **stack_fp.c** (ver las definiciones completas en el apéndice 3):

```
/* En stack_fp.h */
typedef struct _Stack Stack;
/* Tipos de los punteros a función soportados por la pila */
typedef void (*destroy_element_function_type)(void*);
typedef void (*(*copy_element_function_type)(const void*));
typedef int (*print_element_function_type)(FILE *, const void*);

/* En stack_fp.c */
struct _Stack {
    int top;
    void * item[MAXSTACK];
    destroy_element_function_type   destroy_element_function;
    copy_element_function_type      copy_element_function;
    print_element_function_type     print_element_function;
};
```

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

Con el objetivo de comprobar las modificaciones recién realizadas, se repetirá el ejercicio P2_E2 (almacenar en una pila las conexiones entrantes y salientes para cada elemento de un grafo) pero con **dos pilas simultáneas** donde se inserten: en una de ellas los nodos (pila de nodos) y en otra sus ids (pila de enteros). Este programa se desarrollará en un fichero con nombre **p2_e4.c**. A continuación se muestra un ejemplo de la salida:

```
> ./p2_e4 “g1.txt”
Pila de enteros:
[2]
[1]
[3]
[1]
[3]
[2]
Pila de nodos:
[2, b]
[1, a]
[3, c]
[1, a]
[3, c]
[2, b]
```

Nótese que los punteros a función necesarios para la pila de nodos son simplemente llamadas a primitivas de Node. Es decir, para una declaración de una pila así:

```
Stack* snode = stack_ini(destroy_node_function,
                         copy_node_function,
                         print_node_function);
```

Bastaría con tener las siguientes funciones definidas en algún sitio (véase apéndice 4):

```
void destroy_node_function(void* e){
    node_destroy((Node *)e);
}
void * copy_node_function(const void* e){
    return node_copy((Node *)e);
}
int print_node_function(FILE * f, const void* e){
    return node_print(f, (Node *)e);
}
```

PARTE 2.A. PREGUNTAS SOBRE LA PRÁCTICA

Responded a las siguientes preguntas y adjuntadlas al fichero .zip que entreguéis.

1. Indicad qué modificaciones habría que hacer a la pila del ejercicio 3 para que la cima de la pila fuera de tipo **puntero**, en particular especificad qué ficheros habría que modificar y cuáles serían estos cambios:

En **stack_fp.h,** antes:	Ahora:
En **stack_fp.c,** antes:	Ahora:
En **p2_e3.c**, antes:	Ahora:
En …, 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) en una que permita evaluar expresiones posfijo. Realiza el mismo ejercicio pero con la pila general (P2_E4).

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**, **P2_E3** y **P2_E4**.

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_element.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_destroy(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 Element *);
/**-------------------------------------------------------------------
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
-------------------------------------------------------------------*/
Element * 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
-------------------------------------------------------------------*/
Element * 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: element.h

```
typedef struct _Element Element;

/**-------------------------------------------------------------------
Inicializa un elemento de pila. Salida: Puntero al elemento inicializado o NULL en caso de error
-------------------------------------------------------------------*/
Element * element_ini();
/**-------------------------------------------------------------------
Elimina un elemento. Entrada: Elemento a destruir.
-------------------------------------------------------------------*/
void element_destroy(Element *);

/**-------------------------------------------------------------------
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.
-------------------------------------------------------------------*/
Element * element_setInfo(Element *, void*);
/**-------------------------------------------------------------------
Devuelve el contenido de Element. Entrada: El elemento. Salida: El contenido de Element o NULL si ha
habido error.
-------------------------------------------------------------------*/
void * element_getInfo(Element *);

/**-------------------------------------------------------------------
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.
-------------------------------------------------------------------*/
Element * element_copy(const Element *);
/**-------------------------------------------------------------------
Compara dos elementos. Entrada: dos elementos a comparar. Salida: Devuelve TRUE en caso de ser iguales
y si no FALSE
-------------------------------------------------------------------*/
Bool element_equals(const Element *, const Element *);

/**-------------------------------------------------------------------
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 element_print(FILE *, const Element *);
```

Apéndice 3: stack_fp.h

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

/* Tipos de los punteros a función soportados por la pila */
typedef void (*destroy_element_function_type)(void*);
typedef void (*(*copy_element_function_type)(const void*));
typedef int (*print_element_function_type)(FILE *, const void*);

/**-------------------------------------------------------------------
Inicializa la pila reservando memoria y almacenando los tres punteros a función pasados como parámetro (el primero para destruir elementos, el segundo para copiar elementos y el tercero para imprimir elementos). Salida: NULL si ha habido error o la pila si ha ido bien
-------------------------------------------------------------------*/
Stack * stack_ini(destroy_element_function_type f1, copy_element_function_type f2, print_element_function_type f3);
/**-------------------------------------------------------------------
Elimina la pila Entrada: la pila que se va a eliminar
-------------------------------------------------------------------*/
void stack_destroy(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 void *);
/**-------------------------------------------------------------------
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
-------------------------------------------------------------------*/
void * 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
-------------------------------------------------------------------*/
void * 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 4: functions.c

```
/*
 * En este fichero se definen las funciones de destrucción, copia e impresión de elementos a almacenar en
 * una pila para distintos tipos de datos
*/

/* Las siguientes funciones se usarán cuando se quieran guardar enteros en la pila. Ojo! Estas funciones
reciben un puntero a entero, no una estructura con un puntero a entero (como en el ejercicio P2_E1) */
void destroy_intp_function(void* e){
        free((int*)e);
}

void * copy_intp_function(const void* e){
        int * dst;
        if (e == NULL)
                return NULL;
        dst = (int*)malloc(sizeof(int));
        /*Copiamos el elemento*/
        *(dst) = *((int*)e);
        return dst;
}

int    print_intp_function(FILE * f, const void* e){
        if (f != NULL && e != NULL)
                return fprintf(f, "[%d]", *((int*)e));
        return -1;
}

/* Las siguientes se usarán cuando se quieran guardar nodos en la pila */
void destroy_node_function(void* e){
        node_destroy((Node *)e);
}
void * copy_node_function(const void* e){
        return node_copy((Node *)e);
}
int    print_node_function(FILE * f, const void* e){
        return node_print(f, (Node *)e);
}
```