Sesión 08: *Structs* y cadenas enlazadas (*a mejorar*)

Hoja de problemas

Programación para Sistemas

Ángel Herranz
aherranz@fi.upm.es
Universidad Politécnica de Madrid

2020-2021

Ejercicio 1. Repasa las transparencias de clase. El trabajo con *structs* es muy natural, gran parte de su sintaxis es idéntica a la de otros lenguajes de programación y otra parte es muy parecida. Sin embargo, la memoria dinámica, especialmente cuando se viene de lenguajes con recolección automática de basura, se hace muy complicado. De nuevo, entender cada transparencia se hace fundamental.

Ejercicio 2. Utilizando la definición de `struct rectangulo`, tienes que escribir un programa que lea rectángulos de la entrada estándar, calcule su área y los imprima en orden, de mayor área a menor área.

La entrada estándar tendrá el siguiente formato:

- Un entero n que indica el número de rectángulos que habrá que leer. El dato n estará entre 1 y 1000.
- n filas con cuatro enteros A_x, A_y, B_x, B_y cada una que indican los puntos (A_x, A_y) y (B_x, B_y) que definen el rectángulo.

La salida estándar tiene que sacar n filas, cada una con un rectángulo, estando las filas ordenadas de menor a mayor área.

Veamos un ejemplo de entrada:

```
2
0 0 1 1
-1 -2 3 1
```

Y su correspondiente salida:

```
-1 -2 3 1
0 0 1 1
```

Ejercicio 3. Siguiendo las indicaciones del ejercicio sobre pilas de la sesión anterior, tu

trabajo es elaborar una implementación basada en cadenas enlazadas.

💻 **Ejercicio 4.** Lo cierto es que las pilas hacen un uso muy restringido de las cadenas enlazadas. Por ello, en este ejercicio, te proponemos elaborar un tipo abstracto de datos *listas*. Las funciones podrían ser algo como:

- `crear_vacía`
- `insertar_por_el_principio`
- `insertar_por_el_final`
- `insertar_por_posición`
- `longitud`
- `primero`
- `último`
- `iésimo`
- `borrar_primero`
- `borrar_último`
- `borrar_por_posición`

Ejercicio 5. En este ejercicio vamos a combinar conocimiento de varios temas para crear un tipo *tipo abstracto de datos*[1]. El tipo abstracto de datos que tendrás que implementar será el de las pilas usando cadenas enlazadas como estructura de datos.

De esta forma, las pilas serán **punteros a nodos de una cadena enlazada**. Vamos a construir un módulo para dicho tipo abstracto. Empezaremos con el *header*:

Listing 1: `pila.h`

```
#ifndef PILA_H
#define PILA_H

/* Las pilas serán punteros a struct nodo */
struct nodo {
  int dato;
  struct nodo *siguiente;
};

/* Crea una nueva pila vacía */
extern struct nodo *crear();

/* Decide si la pila está vacía */
extern int vacia(struct nodo *pila);

/* Coloca x en la cima de la pila y devuelve la "nueva" pila */
extern struct nodo *apilar(struct nodo *pila, int x);

/* Elimina el elemento en la cima de la pila y devuelve la "nueva" pila */
extern struct nodo *desapilar(struct nodo *pila);

/* Devuelve la cima de una pila no vacía */
extern int cima(struct nodo *pila);

#endif
```

[1] Aunque en C cuesta ser realmente abstracto, lo vamos a intentar.

Tu labor será completar e implementar correctamente las funciones en el fichero `pila.c`. Te ofrecemos aquí el esqueleto:

Listing 2: `pila.c`

```
#include <stdlib.h>
#include "pila.h"

struct nodo *crear() {
  return NULL;
}

struct nodo *apilar(struct nodo *pila, int dato) {
  struct nodo *nuevo;
  /* TODO: corregir y completar */
  return nuevo;
}

struct nodo *desapilar(struct nodo *pila) {
  struct nodo *siguiente;
  /* TODO: corregir y completar */
  return siguiente;
}

int cima(struct nodo *pila) {
  /* TODO: corregir y completar */
  return 0;
}

int es_vacia(struct nodo *pila) {
  return pila == NULL;
}
```

Para ayudarte a saber que vas por el buen camino hemos elaborado el siguiente programar de test:

Listing 3: `pila_test.c`

```
#include <assert.h>
#include <stdio.h>
#include "pila.h"

#define N 20

int main() {
  int i;
  struct nodo *pila;

  pila = crear();
```

```
  /* Tras inicializar la pila debería estár vacía */
  assert(vacia(pila));

  pila = apilar(pila,42);

  /* Tras apilar un elemento la pila no debería estar vacía */
  assert(!vacia(pila));
  /* y la cima debería ser el 42 apilado */
  assert(cima(pila) == 42);

  pila = desapilar(pila);

  /* Tras desapilar el único elemento la pila debería estar vacía */
  assert(vacia(pila));

  /* Este bucle llena la pila con N enteros */
  for (i = 0; i < N; i++) {
    pila = apilar(pila, 2*i);
    /* Tras cada apilado la pila no debería estar vacía */
    assert(!vacia(pila));
    /* y la cima debería ser lo último apilado */
    assert(cima(pila) == 2*i);
  }

  /* Este bucle casi vacía la pila */
  for (i = N - 1; i > 0; i--) {
    /* Tras cada desapilado la pila debería tener los elementos
       apilados en orden inverso */
    assert(cima(pila) == 2*i);

    pila = desapilar(pila);

    /* Tras cada desapilado la pila no deberia estar vacía */
    assert(!vacia(pila));
  }

  pila = desapilar(pila);

  /* Tras el último desapilado, la pila deberia estar vacía */
  assert(vacia(pila));

  fprintf(stderr, "¡Todos los tests pasados!\n");

  return 0;
}
```

Y para que lo tengas aún más fácil, puedes utilizar el siguiente `Makefile`:

```
CFLAGS=-Wall -Werror -g -pedantic

pila_test: pila.o pila_test.o
	$(CC) $(CFLAGS) -o $@ $^

pila.o: pila.c pila.h

test: pila_test
	./pila_test

clean:
	rm -f *.o pila_test
```

Cuando todo esté correcto deberías experimentar la siguiente sesión Bash:

```
$ make
cc -Wall -Werror -g -pedantic   -c -o pila.o pila.c
cc -Wall -Werror -g -pedantic   -c -o pila_test.o pila_test.c
cc -Wall -Werror -g -pedantic -o pila_test pila.o pila_test.o
$ ./pila_test
¡Todos los tests pasados!
$ |
```