Examen final 2021-08-20

95.11/75.02 - Algoritmos y Programación I - Curso Essaya

Objetivo

Se dispone de los siguientes archivos:

- Archivos provistos con código:
  - `Makefile`
- Archivos a completar con código:
  - `ej1.c`, `ej2.c`, `ej3.c`, `ej4.c`, `ej5.c`: Implementación + pruebas de cada ejercicio

Al compilar con `make`, se genera un archivo ejecutable para cada ejercicio: `ej1`, `ej2`, etc. Cada uno de ellos, al ejecutarlo, corre las pruebas para verificar el correcto funcionamiento de la implementación.

El examen se aprueba con al menos 3 ejercicios correctamente resueltos. Un ejercicio se considera correctamente resuelto si:

- El programa `ej<n>` compila sin advertencias ni errores
- La implementación cumple con lo pedido en el enunciado

En algunos ejercicios se incluye un ejemplo de uno o dos casos de prueba y queda a cargo del alumno agregar más casos de prueba, para los que se provee sugerencias. En otros ejercicios se provee únicamente sugerencias. La implementación de las pruebas adicionales es **opcional**, pero se recomienda hacerlo ya que permite asegurar que la resolución del ejercicio es correcta.

Makefile

Para compilar el ejercicio `<n>`: `make ej<n>`. Por ejemplo, para compilar y ejecutar el ejercicio 1:

```
$ make ej1
gcc -Wall -pedantic -std=c99 ej1.c -o ej1
$ ./ej1
ej1.c: OK
```

Salida del programa

Al ejecutar `./ej<n>` se imprime el resultado de las pruebas. Si todas las pruebas pasan correctamente, se imprime `OK`. En caso contrario, cuando una de las verificaciones falla, se imprime un mensaje de error y el programa termina su ejecución. Por ejemplo:

```
$ ./ej1
ej1: ej1.c:45: main: Assertion `p != NULL' failed.
sh: “./ej1” terminated by signal SIGABRT (Abort)
```

Recordar: Correr cada uno de los ejercicios con `valgrind --leak-check=full` para mayor seguridad de que la implementación es correcta.

Pruebas

Se recomienda usar la función `assert` de la biblioteca estándar para verificar condiciones en las pruebas. Ejemplo de uso:

```c
#include <stdio.h>
#include <assert.h>

// funcion a probar
int sumar(int a, int b) {
    return a + b;
}

// pruebas
int main(void) {
    assert(sumar(0, 0) == 0);
    assert(sumar(2, 3) == 5);
    assert(sumar(2, -2) == 0);

    printf("%s: OK\n", __FILE__);
    return 0;
}
```

Nota: A veces para depurar un error en las pruebas es útil imprimir valores; se permite el uso de `printf()` para ello.

Nota: A veces para implementar las pruebas es útil utilizar números aleatorios. Se permite el uso de `rand()` para ello. En ese caso, se recomienda ejecutar `srand(0);` al inicio del programa para asegurar que la secuencia de números aleatorios sea siempre la misma, y así facilitar la depuración.

Ejrcicios

Ejercicio 1 Dada la implementación de una lista enlazada en la que cada nodo almacena un `char`, implementar una función que reciba dos listas enlazadas y determine el resultado de la comparación lexicográfica, suponiendo que cada nodo representa un caracter de una cadena de texto. El comportamiento debe ser similar a la función `strcmp`, es decir que la función devolverá un entero que será negativo, cero o positivo según si la cadena representada en la primera lista es menor, igual o mayor a la segunda, respectivamente.

Ejercicio 2 Al almacenar un número entero de más de 8 bits en un archivo binario, se debe decidir si se almacenará en formato *big endian* o *little endian*. Si se almacena como *big endian*, el primer byte contendrá los 8 bits más significativos, y el último byte los 8 bits menos significativos; y al revés si se almacena como *little endian*.

Se tiene un archivo binario que contiene una secuencia de números de 32 bits, y se desea invertir el modo de almacenamiento de los números; si estaban en big endian se debe pasar a little endian y viceversa. Notar que la operación es la misma en ambos casos, es simplemente invertir el orden de los bytes de cada número.

Implementar la función `bool invertir_endianness(const char *entrada, const char *salida)` que efectúa la conversión descrpita, leyendo del archivo cuya ruta es `entrada` y guardando el resultado en `salida`.

Ejercicio 3 Implementar en forma **recursiva** la función `void separar_con_guiones(const char *s, char r[])`, que dada la cadena `s`, guarda en `r` la cadena resultante de separar con un guión en todos los lugares donde haya dos caracteres consecutivos iguales. Asumir que `r` contiene espacio suficiente para almacenar la cadena resultante.

Ejemplos:

```
separar_con_guiones("hola") -> "hola"
separar_con_guiones("abbcdde") -> "ab-bcd-de"
separar_con_guiones("aaaa") -> "a-a-a-a"
```

Ejercicio 4 Un secuestrador quiere escribir una nota de rescate evitando que reconozcan su letra manuscrita. Encontró una revista y su idea es escribir la nota recortando letras de la revista.

a. Implementar la función `void contar_caracteres(const char *s, size_t cantidades[256])` que almacena en `cantidades` la cantidad de ocurrencias de cada caracter de la tabla ASCII en `s`.
b. Implementar la función `bool nota_rescate_posible(const char *nota, const char *revista)` que determina si es posible o no formar el texto de la nota dado el texto completo de la revista (es decir, si la revista contiene todos los caracteres necesarios).

Ejercicio 5 Sea el TDA `album` que representa un album discográfico, con un autor, un título y una secuencia de temas. Diseñar una estructura para representar el TDA, e implementar las funciones:

- `album_t *album_crear(const char *autor, const char *titulo)`
- `bool album_agregar_tema(album_t *album, const char *titulo_tema)`
- `size_t album_cantidad_temas(album_t *album)`
- `const char *album_autor(album_t *album)`
- `const char *album_titulo(album_t *album)`
- `const char *album_titulo_tema(album_t *album, int i)`
- `void album_destruir(album_t *album)`

Nota: el TDA debe almacenar su propia copia de las cadenas de texto. Se puede suponer que una cadena tiene a lo sumo `MAX_CADENA` caracteres.