Examen final 2021-08-27

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 Un archivo binario contiene un listado de sucursales de una empresa, cada una con su ubicación (latitud y longitud). En el archivo, cada sucursal tiene el siguiente formato:

```
+----+----+-+-----------+
|lat |long|N|nombre     |
+----+----+-+-----------+
```

- `lat` y `long` son dos `float` indicando la latitud y longitud, respectivamente.
- `N` es un entero sin signo de 8 bits que representa la cantidad de caracteres de `nombre` (sin incluir el caracter nulo).
- `nombre` es el nombre de la sucursal (sin incluir el caracter nulo).

Se pide:

a. Declarar una estructura `sucursal_t` para guardar la información de una sucursal.
b. Escribir la función `void sucursal_destruir(sucursal_t *s)`
c. Escribir la función `bool leer_sucursales(const char *nombre_archivo, sucursal_t *sucursales[], size_t *cantidad)` que lea del archivo indicado todas las sucursales contenidas en el mismo, guardándolas en el arreglo `sucursales`, y guardando en `cantidad` la cantidad de sucursales leídas.

Ejercicio 2 En una lista doblemente enlazada, cada nodo contiene una referencia a los nodos anterior y próximo.

Dado el TDA `lista_t` implementado como:

```c
typedef struct nodo {
    int dato;
    struct nodo *prox;
    struct nodo *ant;
} nodo_t;

typedef struct {
    nodo_t *prim;
} lista_t;
```

Implementar la función `void lista_eliminar(lista_t *lista, int dato)`, que elimina el nodo que contiene el dato indicado, si es que existe. En caso de haber más de uno, se debe eliminar el primero únicamente.

Ejercicio 3 ROT47 es un sencillo algoritmo de cifrado utilizado para ocultar un texto sustituyendo cada caracter por el que está 47 posiciones por delante o por detrás en la tabla ASCII.

El algoritmo procesa caracter por caracter, y sólo tendrá en cuenta los caracteres con código ASCII entre 33 y 126 inclusive (que corresponden a los caracteres `!` y `~` respectivamente).

La tabla de conversión es la siguiente:

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

El caracter `!` (ASCII 33) se transforma en `P` (ASCII 33 + 47 = 80) y viceversa. El caracter `"` (34) se transforma en `Q` (81) y viceversa, etc. Cualquier caracter no comprendido en la tabla queda sin modificar.

Se pide escribir un programa que reciba una cantidad arbitraria de parámetros por línea de comandos, e imprima el resultado de codificar los mismos con ROT47. Ejemplo:

```
$ ./ej3 The Quick Brown Fox Jumps Over The Lazy Dog.
%96 "F:4< qC@H? u@I yF>AD ~G6C %96 {2KJ s@8]
$
```

Ejercicio 4 Implementar en forma **recursiva** la función `size_t union(const int a[], size_t na, const int b[], size_t nb, int r[])`, que dados los arreglos ordenados `a` y `b`, guarda en `r` la unión de los dos conjuntos, **en tiempo lineal**, y devuelve la cantidad de elementos de la unión. Asumir que el arreglo `r` tiene espacio suficiente. Cada elemento puede aparecer a lo sumo una vez en cada uno de los arreglos.

Ejercicio 5 Implementar la función `size_t cant_bits(uint32_t a[], size_t n)` que devuelva la cantidad de bits que están encendidos en el arreglo de `n` valores de 32 bits.