Examen final 2021-03-15

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 El algoritmo de *búsqueda por interpolación* es una variación de búsqueda binaria. En cada paso, en lugar de seleccionar el elemento que está en la mitad del arreglo, se hace una interpolación lineal entre los elementos de los extremos.

$$med = izq + \left\lfloor (der - izq) \frac{x - A[izq]}{A[der] - A[izq]} \right\rfloor$$

Por ejemplo, si buscamos el número $x = 22$ en un arreglo A entre las posiciones $izq = 0$ y $der = 99$, el primer paso sería inspeccionar los elementos de los extremos. Suponiendo que $A[izq] = 15$ y $A[der] = 84$:

$$med = 0 + \left\lfloor (99 - 0) \frac{22 - 15}{84 - 15} \right\rfloor = \left\lfloor 99 \frac{7}{69} \right\rfloor = \lfloor 10.0434 \rfloor = 10$$

El próximo paso será inspeccionar el elemento en $A[med] = A[10]$, repitiendo los pasos, como en búsqueda binaria.

Se pide: escribir una función que reciba un arreglo de números (`int`) y un número a buscar, y devuelva el índice del número o -1 si no está en el arreglo, utilizando búsqueda por interpolación.

EJERCICIO 2 Un archivo binario contiene un listado de sucursales de una cadena de supermercados. En el archivo, cada sucursal tiene el siguiente formato:

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

- `N` es un entero sin signo de 8 bits que representa la cantidad de caracteres de `nombre` (sin incluir el caracter nulo). Se garantiza que $0 \leq N \leq 31$.
- `nombre` es el nombre de la sucursal (sin incluir el caracter nulo).
- `lat` y `long` son dos `float` que representan la ubicación (latitud, longitud) de la sucursal.

Se pide:

a. Escribir una función que recibe un archivo previamente abierto y los datos de una sucursal, y escribe en el archivo los datos con el formato descripto.
b. Escribir una función que recibe un archivo previamente abierto, lee los datos de una sucursal y los devuelve (ya sea por el valor de retorno o mediante los parámetros).

EJERCICIO 3 Se tiene un TDA que representa una lista enlazada que contiene números enteros.

Escribir la función `bool esta_ordenada(lista_t *lista)`, que determine **en forma recursiva** si los números están ordenados en forma ascendente o no.

EJERCICIO 4 Escribir la función `void hex_codificar(uint32_t n, char *s)` que guarda en `s` el número recibido convertido a su representación en notación hexadecimal. Asumir que `s` tiene espacio suficiente. Ejemplo:

```c
char s[10];
hex_codificar(0x3f4, s);
// s contiene "3f4"
```

En la implementación de esta función **no** se permite usar funciones de la biblioteca estandar.

EJERCICIO 5 Implementar la función `void merge3(const int a[], size_t na, const int b[], size_t nb, const int c[], size_t nc, int r[])`, que dados los arreglos ordenados `a`, `b`, y `c`, guarda en `r` todos los elementos ordenados, **en tiempo lineal**. Asumir que el arreglo `r` tiene `na + nb + nc` elementos.

Ayuda: Si tenemos la función `merge` que intercala ordenadamente dos arreglos de números, podemos implementar `merge3(a, b, c)` como `merge(merge(a, b), c)`.