Examen final 2021-08-13

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 Implementar en forma **recursiva** la función `void merge(int a[], size_t na, int b[], size_t nb, int r[])`. La función recibe dos arreglos ordenados (`a` y `b`) y guarda en `r` los elementos intercalados ordenadamente, en **tiempo lineal**. Asumir que `r` tiene espacio suficiente para guardar `na + nb` elementos.

Ejercicio 2 Implementar la función `void ordenar(int a[], size_t n)`, que ordena el arreglo utilizando el algoritmo según el siguiente pseudocódigo:

```
algoritmo ordenar(arreglo A con N elementos):
    repetir N veces:
        para i := 1 hasta N-1 inclusive:
            si A[i-1] > A[i] entonces:
                intercambiar(A[i-1], A[i])
```

Indicar (en un comentario) cuál es la complejidad computacional en tiempo (`T(N)`) y espacio (`E(N)`).

Ejercicio 3 Escribir un programa que recibe por parámetro de línea de comandos el nombre de un archivo de texto, e imprime el contenido del mismo línea por línea, en orden inverso (primero la última línea, luego la anteúltima, etc).

Nota: Asumir que se cuenta con memoria RAM suficiente para almacenar el contenido completo del archivo, y que cada línea de texto tiene a lo sumo `MAX_LINEA` bytes.

Nota: Para simplificar, en caso de error no es necesario liberar los recursos previamente reservados (pero sí es necesario detener el procesamiento del archivo y devolver al sistema operativo un número distinto de 0).

Nota: Se provee el archivo `texto.txt` para probar, invocando con `./ej3 texto.txt`.

Ejercicio 4 Implementar la función `uint8_t rotar(uint8_t v, int n)` que rota el valor `v` `n` bits a la derecha (si `n` es positivo) o a la izquierda (si `n` es negativo). Por ejemplo:

```
rotar(0x0a,  1) -> 0x05 // 00001010 -> 00000101
rotar(0x0a,  2) -> 0x82 // 00001010 -> 10000010
rotar(0x0a,  3) -> 0x41 // 00001010 -> 01000001
rotar(0x0a, -1) -> 0x14 // 00001010 -> 00010100
rotar(0x0a,  8) -> 0x0a // 00001010 -> 00001010
```

Ejercicio 5 Dado el TDA lista enlazada en el que los nodos guardan un dato de tipo `int`, implementar la función `int lista_buscar(lista_t *lista, int dato)`, que devuelve la posición en la que se encuentra el elemento dado, o -1 si no está en la lista.