Examen final 2021-09-09

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 RC5 es un protocolo de comunicación estándar utilizado para transmitir datos mediante un canal infrarrojo, por ejemplo en controles remotos.

Un ”paquete” de datos RC5 tiene 16 bits, y tiene el formato:

```
0011TAAAAACCCCCC
```

donde:

- los 4 bits más pesados son siempre 0011.
- `T` es el *toggle bit*, que cambia cada vez que se presiona un botón.
- `AAAAA` (5 bits) es la *dirección* del dispositivo al que va dirigido el comando.
- `CCCCCC` (6 bits) es el *comando* que se desea que el dispositivo ejecute.

Escribir las funciones:

- `uint16_t rc5_empaquetar(bool toggle, uint8_t direccion, uint8_t comando)` que empaqueta los datos en el formato RC5.
- `bool rc5_desempaquetar(uint16_t paquete, bool *toggle, uint8_t *direccion, uint8_t *comando)` que desempaqueta los datos, y devuelve un booleano indicando si el formato es correcto o no.

Ejercicio 2 Dado el TDA lista enlazada implementado como:

```
typedef struct nodo { int dato; struct nodo *prox; } nodo_t;
typedef struct { nodo_t *prim; } lista_t;
```

Escribir la función `bool lista_repetir(lista_t *lista)` que duplique el contenido de la lista, concatenando al final.

Por ejemplo, si la lista contiene `10 -> 20 -> 30`, luego de invocar a `lista_repetir`, debe quedar como `10 -> 20 -> 30 -> 10 -> 20 -> 30`.

Ejercicio 3 Escribir la función `bool invertir_lineas(char *entrada, char *salida)` que recibe dos rutas de archivos de texto y escribe en `salida` el contenido de `entrada` invirtiendo cada una de las líneas.

Asumir que una línea tiene a lo sumo `MAX_LINEA` caracteres.

Ayuda: tener en cuenta que lo que hay que invertir en cada línea es todo lo que está *antes* del caracter `\n`.

Ejercicio 4 Implementar la función `void rellenar(size_t m, size_t n, int matriz[m][n], int i, int j, int color)` que ofrece una funcionalidad similar a la herramienta ”balde de pintura” del famoso Paint: pintar una región de pixels contiguos que sean del mismo color, con un color diferente.

Los argumentos son: una matriz de dimensiones `m x n`, una posición `(i, j)` y un color C.

La función debe efectuar el siguiente algoritmo:

- Sea P el color actual del casillero `(i, j)`
- Pintar el casillero con el color C.
- Repetir la operación para cada uno de los 4 casilleros contiguos (norte, sur, este, oeste) *que tengan color P*.

Sugerencia: pensarlo en forma recursiva.

Ejercicio 5 Se desea implementar un TDA ”vector dinámico de floats”. Se pide:

- Definir el tipo de dato `vecf_t`.
- **Implementar las funciones:**
  - `vecf_t *vecf_crear()`
  - `void vecf_destruir(vecf_t *v)`

- `size_t vecf_cantidad(const vecf_t *v)`
- `bool vecf_agregar(vecf_t *v, float f)`
- `float vecf_suma(const vecf_t *v)` que devuelve la suma de todos los floats