Examen final 2021-09-03

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

a. Definir el tipo de dato `persona_t` que representa a una persona en el padrón electoral nacional. Una persona tiene un DNI (`int`), un nombre y una dirección (cadenas de texto de tamaño indefinido).
b. Escribir las funciones `persona_crear` y `persona_destruir`.
c. Escribir la función `persona_t *persona_buscar(int dni, persona_t *personas[], size_t n)` que recibe un arreglo de personas ordenado por DNI, y un DNI a buscar, devuelve un puntero a la persona correspondiente (si es que se encuentra) en tiempo **mejor que lineal**.

Ejercicio 2 Escribir un programa que reciba mediante argumentos de línea de comandos los nombres de dos archivos y copie el contenido del primero al segundo. Por ejemplo:

```
./ej2 foto.jpg copia.jpg
```

debe crear el archivo `copia.jpg` (o sobreescribirlo si ya existía) con el contenido exacto de `foto.jpg`.

Notas:

- Asumir que el archivo a copiar es binario.
- En caso de error (cualquier tipo de error que pueda ocurrir), mostrar un mensaje y terminar la ejecución del programa.
- Para simplificar, se permite no liberar los recursos en caso de error.

Ejercicio 3 Una lista enlazada contiene un ciclo si algún nodo referencia a algún otro nodo que se encuentra ”atrás” en la secuencia.

El algoritmo de Floyd permite detectar ciclos en una lista enlazada, y funciona mediante una *tortuga* y una *liebre*. La tortuga es una referencia a un nodo que avanza ”despacio”: en cada iteración avanza un nodo hacia adelante. La liebre es una referencia a un nodo que avanza ”rápido”: en cada iteración avanza dos nodos hacia adelante.

La tortuga y la liebre arrancan en el primer nodo. Si en algún momento se vuelven a encontrar en algún nodo significa que hay un ciclo. Si llegan al final de la lista sin volver a encontrarse es que no hay ciclos.

Dado el TDA lista enlazada implementado como:

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

Implementar la función `bool hay_ciclo(lista_t *lista)` que determine si hay o no un ciclo, mediante el algoritmo de Floyd.

Ejercicio 4 Se tiene un registro de 16 bits guarda la fecha actual en el siguiente formato:

```
M M M M D D D D D Y Y Y Y Y Y Y
15            8 7             0
```

`MMMM` representa los 4 bits del mes (siendo enero el mes 1), `DDDDD` representa los 5 bits del día, y `YYYYYYY` son los 7 bits del año (comenzando desde el año 2000).

Por ejemplo, el 2 de abril de 2001 se representaría en este registro como:

```
0 1 0 0|0 0 0 1 0|0 0 0 0 0 0 1
   4   |    2    |     2001
```

Escribir las funciones:

- `uint16_t empaquetar_fecha(unsigned int dia, unsigned int mes, unsigned int anio)` que devuelve la fecha codificada en el formato descripto.
- `void desempaquetar_fecha(uint16_t reg, unsigned int *dia, unsigned int *mes, unsigned int *anio)` que decodifica el registro `reg` y guarda en `dia`, `mes` y `anio` los valores correspondientes.

Ejercicio 5 Escribir en forma **recursiva** la función `void intercalar(char a[], char b[], char r[])` que guarda en `r` el resultado de intercalar las cadenas `a` y `b` caracter por caracter. Asumir que `r` tiene espacio suficiente para guardar el resultado.

Por ejemplo, `intercalar("hola", "mundo!", r)` debe guardar en `r` la cadena `"hmoulnado!"`.