Examen final 2021-04-17

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:

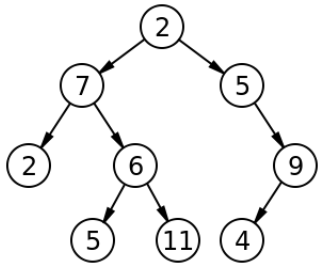
```c
#include <stdio.h>
#include <assert.h>

// 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 *árbol binario* es una estructura enlazada en la que cada nodo contiene referencias a otros dos nodos, llamados *hijo izquierdo* y *derecho* (pudiendo cualquiera de ellos ser una referencia nula).



Dado un *camino* formado por una secuencia de unos y ceros, se puede recorrer un árbol binario de la siguente manera:

- Se comienza en el nodo raíz (el primer nodo del árbol).
- Por cada elemento del camino:
  - Si es un 0, continuar hacia el hijo izquierdo
  - Si es un 1, continuar hacia el hijo derecho

Ejemplo: para el árbol mostrado arriba, el camino `"010"` conduce al valor 5.

Sea la estructura `nodo_t` que representa un nodo del árbol, cuyo dato es de tipo `int`. Se pide implementar la función `int arbol_recorrer(nodo_t *nodo, const char *camino)`, que recibe un puntero a un nodo del árbol y un camino a recorrer, y devuelve el dato almacenado en el nodo resultante de recorrer el camino, o 0 si el camino no conduce a un nodo.

Sugerencia: pensar la función en forma recursiva.

Ejercicio 2 Base32 es un algoritmo de codificación diseñado para codificar datos binarios de longitud arbitraria, convirtiéndolo en una cadena de caracteres imprimibles.

El alfabeto de caracteres se define de la siguiente manera:

Valor	Caracter	Valor	Caracter	Valor	Caracter	Valor	Caracter
0	A	9	J	18	S	27	3
1	B	10	K	19	T	28	4
2	C	11	L	20	U	29	5
3	D	12	M	21	V	30	6
4	E	13	N	22	W	31	7
5	F	14	O	23	X		
6	G	15	P	24	Y		
7	H	16	Q	25	Z		
8	I	17	R	26	2		

Para codificar un arreglo de bytes:

- El arreglo se divide en bloques de 5 bytes (40 bits).
- Cada bloque de 5 bytes se divide en 8 valores de 5 bits (0-31), y cada valor se convierte en un caracter según la tabla.

Escribir la función `void base32_codificar(uint8_t bytes[], size_t n, char s[])`, que guarda en `s` la cadena codificada en Base32. Asumir que `n` es múltiplo de 5, y que `s` tiene capacidad suficiente.

Ejercicio 3 Implementar la función `void interseccion(const int a[], size_t na, const int b[], size_t nb, int r[], size_t *nr)`, que dados los arreglos ordenados `a` y `b`, guarda en `r` la intersección de los dos conjuntos, **en tiempo lineal**, y en `nr` la cantidad de elementos de la intersección. Asumir que el arreglo `r` tiene espacio suficiente.

Nota: los arreglos pueden contener elementos duplicados; la intersección debe incluir tantas copias del elemento como veces que aparece en ambos arreglos.

Ejemplos:

```
interseccion({2},          {3})          -> {}
interseccion({2},          {2})          -> {2}
interseccion({2, 2},       {2})          -> {2}
interseccion({2, 2},       {2, 2})       -> {2, 2}
interseccion({1, 2, 2, 4}, {1, 2, 2, 3}) -> {1, 2, 2}
```

Ejercicio 4 Escribir la función `bool uniq(const char *entrada, const char *salida)`, donde `entrada` es la ruta a un archivo de texto existente, y `salida` es la ruta a un archivo para escribir. La función debe copiar el contenido línea por línea, salteando las líneas duplicadas consecutivas. Ejemplo:

```
Entrada:                          Salida:
I like to move it, move it        I like to move it, move it
I like to move it, move it        Ya like to move it
I like to move it, move it        I like to move it, move it
Ya like to move it                Ya like to move it
I like to move it, move it
I like to move it, move it
I like to move it, move it
Ya like to move it
```

Asumir que las líneas del archivo tienen como máximo 1024 caracteres.

Ejercicio 5 Implementar el TDA *cuenta bancaria*, que cumpla con el siguiente comportamiento:

```c
cuenta_t *c = cuenta_crear("Perez");
printf("Cuenta de %s\n", cuenta_nombre(c)); // imprime "Cuenta de Perez"
cuenta_saldo(c);                            // devuelve 0
cuenta_acreditar(c, 100, "Sueldo");
cuenta_extraer(c, 60, "Shopping");          // debita la cantidad y devuelve true
cuenta_saldo(c);                            // devuelve 40
cuenta_extraer(c, 100, "Deudas");           // devuelve false

void f(int64_t cantidad, const char *descripcion) {
    printf("%d %s\n");
}

cuenta_movimientos(c, f); // por cada movimiento registrado, ejecuta la función f
                          // dada la función f de arriba imprime:
                          //   100 Sueldo
                          //   -60 Shopping

cuenta_destruir(c); // libera la memoria
```

Observaciones:

- La cuenta lleva un registro de todos los movimientos efectuados exitosamente. La cantidad de movimientos registrados no tiene límite.
- El saldo nunca puede ser negativo.
- La función `cuenta_movimientos` recorre los movimientos registrados, y por cada uno llama a la función recibida. La función recibe un número con signo; si el movimiento es una extracción, el número recibido será negativo.