# Examen final 2021-04-05

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;
}
```

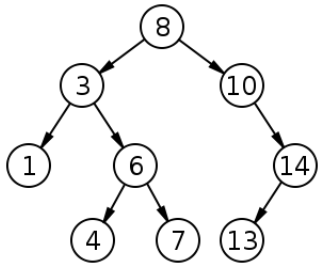
Nota: A veces para depurar un error en las pruebas es útil imprimir valores; se permite el uso de `printf()` para ello.

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.

# Ejercicios

**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).

Un *árbol binario de búsqueda* (ABB) es un árbol binario en el que el dato en cada nodo es mayor o igual que el dato de cualquier nodo del sub-árbol izquierdo, y menor o igual que el dato de cualquier nodo del sub-árbol derecho.




Si queremos buscar un elemento en el árbol (por ejemplo el número 6 en el árbol de arriba), no hace falta recorrer todos los nodos, ya que al visitar cada nodo podemos comparar el elemento buscado con el del nodo y así decidir si el elemento podría estar en el subárbol derecho o izquierdo. Como $6 < 8$, no hace falta visitar los nodos del subárbol derecho. Esta operación se repite en cada nodo visitado hasta encontrar el elemento, o determinar que no existe.

Sea la estructura `nodo_t` que representa un nodo del ABB, cuyo dato es de tipo `int`. Se pide implementar la función `nodo_t *abb_buscar(nodo_t *nodo, int dato)`, que recibe un puntero a un nodo del árbol y busca el `dato` en el nodo y sus hijos, devolviendo un puntero al nodo que contiene el dato, o `NULL` si no se encuentra.

Recomendación: pensar la función en forma recursiva.

**Ejercicio 2** Implementar el TDA *registro de desplazamiento*, que representa un valor de `n` bits (`n` múltiplo de 8). Se considera que el bit menos significativo es el que está en el extremo derecho, y el más significativo en el extremo izquierdo. Implementar las funciones:

- `reg_t *reg_crear(size_t n)`: crea un registro de `n` bits. Devuelve `NULL` si `n` es inválido, o si la operación falla. Todos los bits arrancan en 0.
- `bool reg_rshift(reg_t *r, bool v)`: desplaza a la derecha el registro. El bit más significativo quedará con el valor `v`. El valor previo del bit menos significativo es devuelto.
- `bool reg_lshift(reg_t *r, bool v)`: desplaza a la izquierda el registro. El bit menos significativo quedará con el valor `v`. El valor previo del bit más significativo es devuelto.
- `void reg_destruir(reg_t *r)`

**Ejercicio 3** Implementar las funciones:

- `bool matriz_guardar(const char *nombre, size_t n, size_t m, float matriz[n][m])` que guarda en el archivo `nombre` la matriz de `n x m` números de tipo `float`.
- `float *matriz_cargar(const char *nombre, size_t *n, size_t *m)` que lee del archivo la matriz, devolviendo el puntero al bloque reservado de memoria dinámica, y guardando en `n` y `m` las dimensiones.

El formato del archivo queda a libre criterio.

Nota: `matriz_cargar` devuelve `float*` porque al no saber las dimensiones de la matriz no tenemos otra opción. El casteo al tipo de matriz correspondiente lo hará quien haya llamado a la función (se incluye un ejemplo en las pruebas). Recordar que en el fondo los valores de un arreglo multidimensional se guardan en memoria como un arreglo común y corriente. Es decir, `{{1, 2, 3}, {4, 5, 6}}` es equivalente a `{1, 2, 3, 4, 5, 6}`.

**Ejercicio 4** Sea la siguiente estructura que representa un libro:

```c
typedef struct libro {
    char titulo[MAX_TITULO];
    char autor[MAX_AUTOR];
    uint16_t anno_publicacion;
} libro_t;
```

Escribir la función `void ordenar_libros(libro_t libros[], size_t n)` que ordena los libros en forma creciente según el valor de `anno_publicacion`, y para los libros publicados en el mismo año, según el `titulo`.

**Ejercicio 5** Implementar la función `void unir(char destino[], char delimitador, char *cadenas[], size_t n)`, que a partir de un arreglo de cadenas guarda en `destino` la cadena formada por todas las cadenas separadas por el delimitador. Asumir que `destino` tiene espacio suficiente. Ejemplo:

```c
char *cadenas[] = { "2021", "04", "05" };
char s[16];
unir(s, '-', cadenas, 3);
// s contiene "2021-04-05"
```