



Universidad Autónoma de San Luis Potosí

Facultad de Ingeniería

Área en Ciencias de la Computación



Estructuras de Datos II

Semestre: Agosto – Diciembre 2022

Alumno: Esparza Castañeda Hugo

Profesor: M. en C. Froylán Eloy Hernández Castro



Unidad 1	APUNTADORES	1
	1.1. Introducción a apuntadores	1
	1.2. Parámetros por valor/referencia	1
	1.3. Apuntadores a apuntadores	2
	1.4. Apuntadores y arreglos	2
TAREA	Tarea #0: Resumen del video “Pointers”	3
	1.5 Aritmética de apuntadores	4
TAREA	Tarea #1: Aritmética de apuntadores	6
	1.6. Gestión de memoria dinámica	19
	1.7. Estados de la memoria dinámica	20
	1.8. Problemas en el manejo de la memoria dinámica	20
	1.9. Arreglos dinámicos	22
	1.10. Matriz dinámica	23
TAREA	Tarea #2: Arreglos dinámicos	25
	1.11. Apuntadores por “referencia”	37
	1.12. realloc	37
	1.13. Pila dinámica	38
	1.14. Estructuras dinámicas	39
	1.15. Apuntadores genéricos	40
TRABAJO	Trabajo en clase: “Arreglo dinámico”	41
TAREA	Tarea #3: Arreglos redimensionables	46
TRABAJO	Trabajo en clase: “Farmacia con medicamentos”	62
EXAMEN	Examen Primer Parcial	64
Unidad 2	LISTAS ENLAZADAS	67
	2.1. Listas simples	67
	2.1.1. Definir la estructura del nodo	67
	2.1.2. Funciones de listas simples	68
TAREA	Tarea #4: Listas simples enlazadas	
	2.2. Pilas, colas y conjuntos	
	2.2.1. Pilas con listas	
	2.2.2. Colas con listas	
	2.2.3. Conjuntos con listas	

**TAREA****Tarea #5:** Conjuntos con listas enlazadas

2.3. Listas circulares

2.3.1. Funciones de listas circulares

2.4. Listas dobles

2.4.1. Funciones de listas dobles

2.5. Listas circulares doblemente enlazadas

2.5.1. Funciones de listas circulares doblemente enlazadas

2.6. Listas circulares doblemente enlazadas con centinela

2.6.1. Funciones de listas circulares doblemente enlazadas con centinela

TAREA**Tarea #6:** Listas doblemente enlazadas con centinela**EXAMEN****Examen Segundo Parcial**

2.7. Listas de listas

2.7.1. Funciones de listas de listas

TRABAJO**Trabajo en clase:** “Actividad Vuelos de Avión”**TRABAJO****Trabajo en clase:** “Actividad Biblioteca”**TAREA****Tarea #7:** Listas de listas**Unidad 3****GRAFOS**

3.1. Estructuras de grafos

3.2. Ejercicio, matriz de adyacencia

3.3. Ejercicio biblioteca

3.4. Operaciones sobre grafos

3.5. Grado de un vértice

TAREA**Tarea #8:** Operaciones sobre Grafos**EXAMEN****Examen Tercer Parcial****Unidad 4****ÁRBOLES BINARIOS**

4.1. Conceptos básicos de árboles binarios

4.2. Árboles binarios de expresiones

4.2.1. Definir la estructura del nodo

4.2.2. Función evalúa

4.3. Construcción de un árbol binario de expresiones

TAREA**Tarea #9:** Árboles de Expresiones

4.4. Árboles binarios de búsqueda



4.5. Búsqueda en un árbol binario de búsqueda

4.5.1. Tipos de recorridos

4.6. Eliminación ABB

TAREA**Tarea #10: Árboles Binarios de Búsqueda**

4.7. Árboles binarios balanceados

4.7.1. Rotaciones simples

4.8. Ejemplo de árboles AVL

4.9. Casos generales de rotación AVL

TAREA**Tarea #11: Árboles AVL****Unidad 1****ÁRBOLES MULTICAMINOS**

5.1. Árboles B

5.2. Definiciones/propiedades de un árbol B

5.3. Ejemplo de inserción en un árbol B de orden 2

5.4. Ejemplo de inserción en Árboles B

5.5. Eliminación en Árboles B

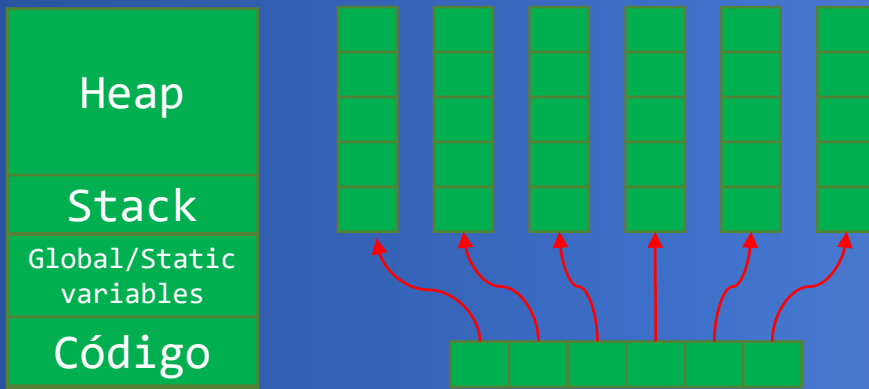
5.6. Árboles B+

5.7. Inserción en Árboles B+

5.8. Eliminación en Árboles B+

EXAMEN**Examen Cuarto Parcial**

1. APUNTADORES

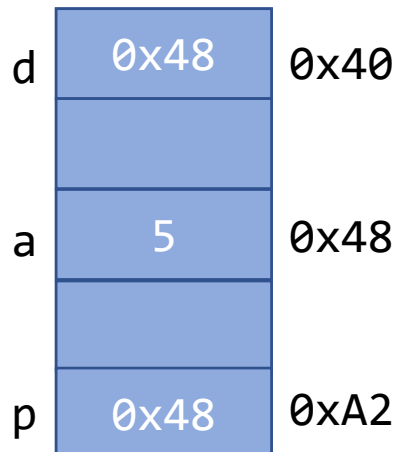


1.1 Introducción a apunadores

Objetivo: Conocer el concepto de apunadores en el paso de parámetros de funciones y gestión de memoria (Dinámica), así como ser capaz de utilizarlos.

Apuntador: un apuntador es una variable que contiene una dirección de memoria pero su contenido es otra dirección de memoria.

```
1 int main(){
2     int a;
3     a = 5;
4     int *p;
5     p = &a;
6     int *d;
7     d = p;
8     return 0;
9 }
```



```
printf("%d", a); //output: 5
printf("%d", *p); //output: 5
printf("%p", p); //output: 0x48
printf("%d", *d); //output: 5
printf("%p", d); //output: 0x48
```

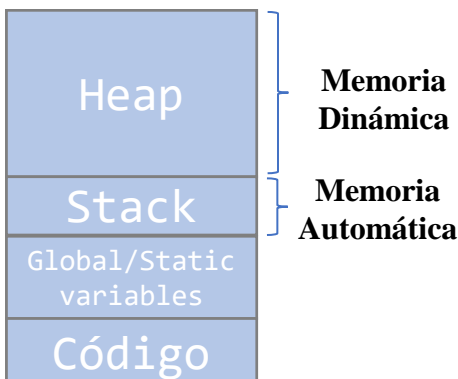
$a = 10; \Leftrightarrow *p = 10$

$\text{scanf}(\text{"\%d"}, p) \Leftrightarrow \text{scanf}(\text{"\%d"}, \&a)$

1.2 Parámetros por valor/referencia

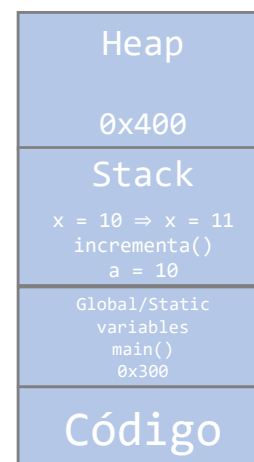
Jueves 18 Agosto 2022

Memoria

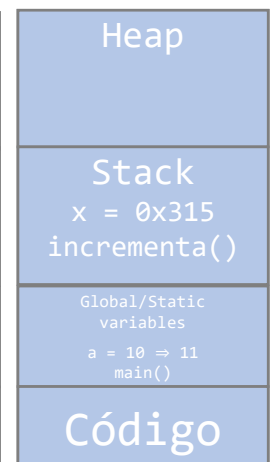


```
1 #include <stdio.h>
2 void incrementa(int *);
3 int main(){
4     int a;
5     a = 10;
6     incrementa(&a);
7     printf("%d", a);
8     return 0;
9 }
10
11 void incrementa(int *x){
12     *x = *x + 1;
13     return;
14 }
```

Por valor



Por referencia



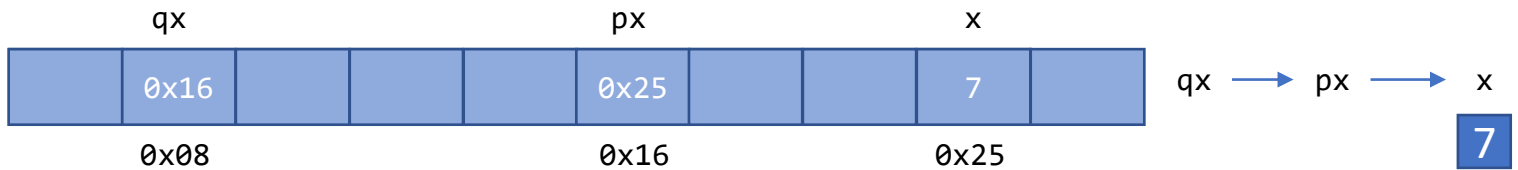


```
int *r;  ⇒  int *r = NULL;
```



1.3 Apuntadores a apuntadores

Viernes 19 Agosto 2022



```

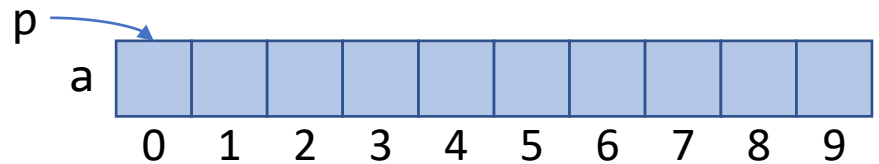
1  #include <stdio.h>
2  int main(){
3      int x = 7; //entero
4      int *px = &x; //apuntador
5      int **qx = &px; //apuntador a apunt
6      printf("%p\n", px); //output: 0x25
7      printf("%d\n", *px); //output: 7
8      printf("%p\n", &x); //output: 0x25
9      printf("%p\n", &px); //output: 0x16
10     printf("%p\n", &qx); //output: 0x08
11     printf("%p\n", qx); //output: 0x16
12     printf("%p\n", *qx); //output: 0x25
13     return 0;
14 }
```

* = indirección = Obtener el valor almacenado en una dirección.
 & = dirección = Obtener la dirección de una variable.

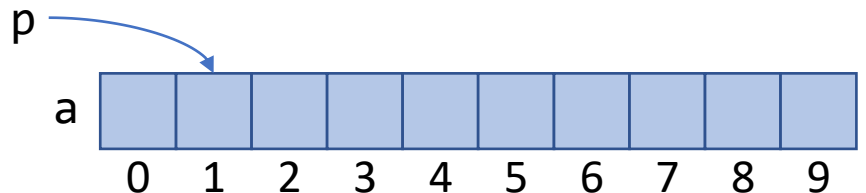
Los apuntadores a apuntadores se usan para matrices dinámicas.

1.4 Apuntadores y arreglos

```
int a[10];
int *p = a; ⇒ int *p = &a[0];
```

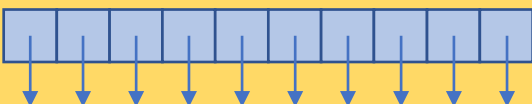


```
p = p + 1;
```



Arreglo de apuntadores

```
int *b[10];
```



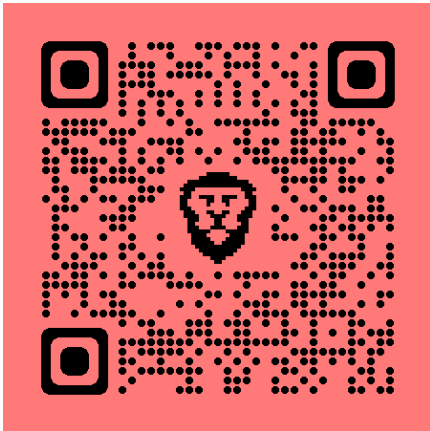
Arreglo de apuntadores apuntando a NULL

```
for(int i = 0; i < 10; i++){
    b[i] = NULL;
}
```

⇔ `int *b[10] = {NULL};`



Tarea #0: Resumen del video "Pointers"



Los punteros es uno de los temas más difíciles, sin embargo son de mucha utilidad, como por ejemplo al pasar datos a una función, modificarlos y luego regresarlos, cosa que no podría hacerse de otra manera.

Cuando pasamos un dato por valor a una función, lo que estamos pasando es en realidad una copia, sin embargo, cuando pasamos el dato por referencia haciendo el uso de punteros, lo que hacemos es darle acceso a la memoria en donde esta almacenado dicho valor, haciendo que todos los cambios que realicemos en la función, le sucedan al valor mismo.

Cada archivo de nuestra computadora esta almacenado en el disco duro, o en un disco de estado solido, según sea el caso. Esos discos son solo de almacenamiento, por lo tanto, no podemos trabajar directamente ahí, la

manipulación de los datos únicamente puede hacerse en la memoria RAM, por lo que tenemos que movernos ahí. La memoria RAM es como un arreglo enorme. La memoria RAM es Memoria de Acceso Aleatorio, y una vez que apagamos la computadora, todos los datos de la memoria RAM se destruyen.

Retomando lo que dijimos hace un momento que la memoria RAM es un arreglo enorme dividido en celdas de bytes (dependiendo del tamaño del tipo de dato), así como los arreglos tienen índices para cada valor guardado, la memoria tiene una dirección para cada dato almacenado.

Las variables de tipo str o cadena de texto, necesitan llevar el '\0' para poder saber donde terminan. Una vez aclarado el tema de la memoria, lo más importante de recordar sobre los punteros es, que son solo direcciones de memoria.

Entonces, un puntero es un tipo de dato que guarda una dirección, y el tipo únicamente nos describe el dato guardado en esa dirección de memoria.

Al saber la dirección de la memoria en donde esta guardada una variable, podemos ir directamente a esa ubicación en la memoria y manipular el dato, es por eso que no es necesario hacer una copia cuando enviamos un valor por referencia en una función.

El puntero más simple en C es el puntero NULL, que como su nombre o indica, no apunta a nada (lo cual en realidad puede ser muy útil). Siempre que declaramos un puntero, y no le asignamos una dirección, debemos hacer NULL el valor de ese puntero (son buenas practicas de programación). Podemos extraer la dirección de una variable con el operador & (como en los ejemplos anteriores).

El propósito principal de un puntero es permitirnos modificar o inspeccionar la ubicación a la que apunta.

Si tenemos un puntero pc, entonces *pc es el dato que vive en la dirección de memoria guardada en pc.

Usado en contexto, * es el operador que nos permite acceder al dato almacenado en la dirección, dicho de otro modo, no nos sirve únicamente saber la dirección, también debemos ir allí, y el operador * nos permite hacerlo.

Por lo tanto, cuando tenemos un puntero que apunta a NULL, es decir a nada, e intentamos ir a esa dirección con el operador * nos aparecerá el error segmentation fault.

Es por esto la importancia de declarar todos los apuntadores a NULL cuando no se les asigna inmediatamente una dirección significativa, ya que si no lo hacemos, nuestro apuntador podría apuntar a cualquier espacio de memoria y manipularlo accidentalmente.

Tipo de dato	Tamaño (en bytes)
int	4
char	1
float	4
double	8
long long	8
char*	4 u 8



1.5

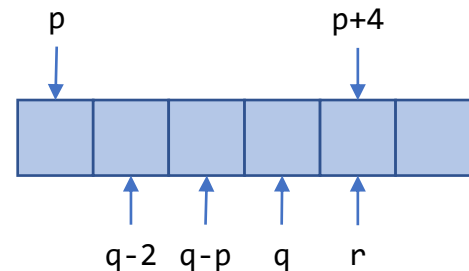
Aritmética de apuntadores

Lunes 22 Agosto 2022

Operador	Nombre
*	Indirección
&	Dirección
->	Apunta a
+	Suma
-	Resta
==	Igualdad
!=	Desigualdad
>	Comparación de direcciones
>=	
<	
<=	
(tipo de dato)	Cast ó conversión explícita

Operaciones aritméticas sobre apuntadores

1. Sumar un entero a un apuntador.
2. Restar un entero a un apuntador.
3. Restar dos apuntadores.
4. Comparaciones.



$p < q$ True
 $p == q$ False
 $q >= p$ True

$q == r + 1$ False
 $q + 1 == r$ True

Ejemplo 1:

```

1  #include <stdio.h>
2  int main(){
3      int a[] = {28, 41, 7};
4      int *pi = a;
5      printf("%d\n", *pi); //output: 28
6      pi += 1;
7      printf("%d\n", *pi); //output: 41
8      pi++;
9      printf("%d\n", *pi); //output: 7
10     ++pi;
11     printf("%d\n", *pi); //output: ???
12     return 0;
13 }
```

Ejemplo 2:

```

1  #include <stdio.h>
2  int main(){
3      short s;
4      short *ps = &s;
5      char c;
6      char *pc = &c;
7      printf("%p\n", ps); //output: 0x7ffcca0af8c6
8      ps = ps + 1;
9      printf("%p\n", ps); //output: 0x7ffcca0af8c8
10     printf("%p\n", pc); //output: 0x7ffcca0af8c5
11     pc = pc + 1;
12     printf("%p\n", pc); //output: 0x7ffcca0af8c6
13     return 0;
14 }
```

short = 2 bytes, char = 1 byte

Martes 23 Agosto 2022

Ejemplo 3: Función strlen

```

1  int strlen(char *s){
2      char *p = s;
3      while(*p != '\0'){
4          p++;
5      }
6      return p-s;
7 }
```

Ejemplo 4: Función strcpy

```

1  void strcpy(char *s, char *t){
2      while((*s = *t) != '\0'){
3          s++;
4          t++;
5      }
6      return;
7 }
```

Ejemplo 5: Función strcmp

```

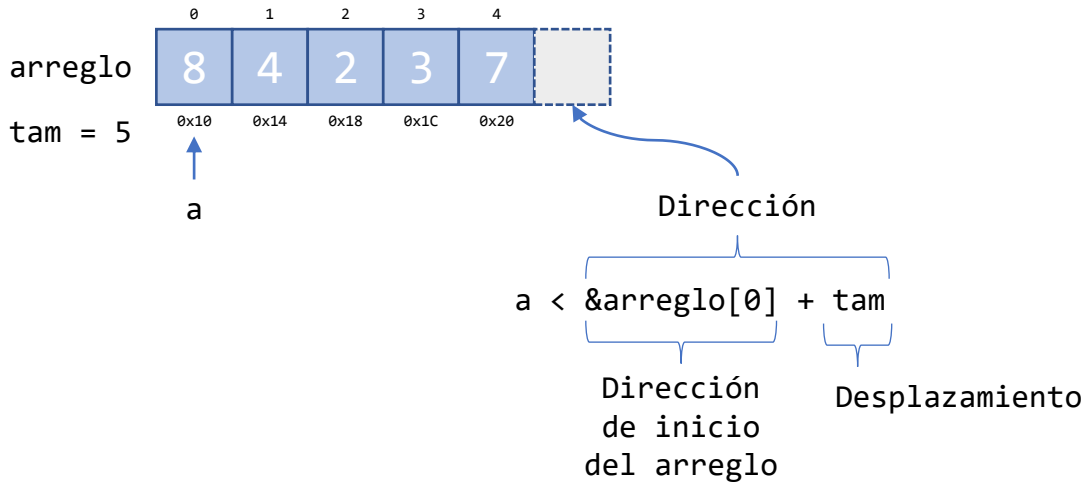
1  int strcmp(char *s, char *t){
2      for(; *s == *t; s++, t++){
3          if(*s == '\0'){
4              return 0;
5          }
6      }
7      return *s - *t;
8 }
```



Problema

Obtener la suma de los elementos de un arreglo de enteros de un tamaño determinado, usando aritmética de apuntadores.

```
int arreglo[5] = {8 4, 2, 3, 7};
```



Con ciclo for

```
1  #include <stdio.h>
2  int suma(int, int []);
3  int main(){
4      int tam = 5;
5      int arreglo[5] = {8, 4, 2, 3, 7};
6      int p = suma(tam, arreglo);
7      printf("%d\n", p);
8      return 0;
9  }
10
11 int suma(int tam, int arreglo[tam]){
12     int sum = 0;
13     int *a = arreglo;
14     for(int *a = arreglo; a - arreglo < tam; a++){
15         sum += *a;
16     }
17     return sum;
18 }
```

Con ciclo while (más elegante)

```
1  #include <stdio.h>
2  int suma(int, int []);
3  int main(){
4      int tam = 5;
5      int arreglo[5] =
6      {8, 4, 2, 3, 7};
7      int p = suma(tam, arreglo);
8      printf("%d\n", p);
9      return 0;
10 }
11
12 int suma(int tam, int arreglo[tam]){
13     int sum = 0;
14     int *a = arreglo;
15     int *b = &arreglo[tam];
16     while(a < b){
17         sum += *a++;
18     }
19     return sum;
20 }
```

`arreglo` \Rightarrow `&arreglo[0]`

`&arreglo[tam]` \Rightarrow too-far pointer
no forma parte
del arreglo



Tarea #1: Aritmética de apuntadores

Instructions

Given any two lists **A** and **B**, determine if:

- List **A** is equal to list **B**; or
- List **A** contains list **B** (**A** is a superlist of **B**); or
- List **A** is contained by list **B** (**A** is a sublist of **B**); or
- None of the above is true, thus lists **A** and **B** are unequal

Specifically, list **A** is equal to list **B** if both lists have the same values in the same order.

List **A** is a superlist of **B** if **A** contains a sub-sequence of values equal to **B**.

List **A** is a sublist of **B** if **B** contains a sub-sequence of values equal to **A**.

Examples:

- If **A** = [] and **B** = [] (both lists are empty), then **A** and **B** are equal
- If **A** = [1, 2, 3] and **B** = [], then **A** is a superlist of **B**
- If **A** = [] and **B** = [1, 2, 3], then **A** is a sublist of **B**
- If **A** = [1, 2, 3] and **B** = [1, 2, 3, 4, 5], then **A** is a sublist of **B**
- If **A** = [3, 4, 5] and **B** = [1, 2, 3, 4, 5], then **A** is a sublist of **B**
- If **A** = [3, 4] and **B** = [1, 2, 3, 4, 5], then **A** is a sublist of **B**
- If **A** = [1, 2, 3] and **B** = [1, 2, 3], then **A** and **B** are equal
- If **A** = [1, 2, 3, 4, 5] and **B** = [2, 3, 4], then **A** is a superlist of **B**
- If **A** = [1, 2, 4] and **B** = [1, 2, 3, 4, 5], then **A** and **B** are unequal
- If **A** = [1, 2, 3] and **B** = [1, 3, 2], then **A** and **B** are unequal



sublist.h

```
#ifndef SUBLIST_H
#define SUBLIST_H

#include <stddef.h>

typedef enum { EQUAL, UNEQUAL, SUBLIST, SUPERLIST } comparison_result_t;

comparison_result_t check_lists(int *list_to_compare_begin, int *list_to_compare_end,
                                int *base_list_begin, int *base_list_end);

#endif
```



test_sublist.c

```
#include "test-framework/unity.h"
#include "sublist.h"
#define ELEMENT_COUNT(array) (sizeof(array) / sizeof(array[0]))
void setUp(void){
}

void tearDown(void){
}

static void test_empty_lists(void){
    TEST_IGNORE(); // delete this line to run test
    TEST_ASSERT_EQUAL(EQUAL, check_lists(NULL, NULL, NULL, NULL));
}

static void test_empty_list_within_non_empty_list(void){
    TEST_IGNORE();
    int base_list[] = { 1, 2, 3 };

    TEST_ASSERT_EQUAL(SUBLIST,
        check_lists(NULL, NULL, base_list, base_list+ELEMENT_COUNT(base_list)));
}

static void test_non_empty_list_contains_empty_list(void){
    TEST_IGNORE();
    int list_to_compare[] = { 1, 2, 3 };

    TEST_ASSERT_EQUAL(SUPERLIST, check_lists(list_to_compare,
        list_to_compare+ELEMENT_COUNT(list_to_compare),
        NULL, NULL));
}

static void test_list_equals_itself(void){
    TEST_IGNORE();
    int list_to_compare[] = { 1, 2, 3 };
    int base_list[] = { 1, 2, 3 };

    TEST_ASSERT_EQUAL(EQUAL, check_lists(list_to_compare,
        list_to_compare+ELEMENT_COUNT(list_to_compare),
        base_list,
        base_list+ELEMENT_COUNT(base_list)));
}

static void test_different_lists(void){
    TEST_IGNORE();
    int list_to_compare[] = { 1, 2, 3 };
    int base_list[] = { 2, 3, 4 };

    TEST_ASSERT_EQUAL(UNEQUAL, check_lists(list_to_compare,
        list_to_compare+ELEMENT_COUNT(list_to_compare),
        base_list,
        base_list+ELEMENT_COUNT(base_list)));
}
```



```
static void test_false_start(void){
    TEST_IGNORE();
    int list_to_compare[] = { 1, 2, 5 };
    int base_list[] = { 0, 1, 2, 3, 1, 2, 5, 6 };

    TEST_ASSERT_EQUAL(SUBLIST, check_lists(list_to_compare,
                                            list_to_compare+ELEMENT_COUNT(list_to_compare),
                                            base_list,
                                            base_list+ELEMENT_COUNT(base_list)));
}

static void test_consecutive(void){
    TEST_IGNORE();
    int list_to_compare[] = { 1, 1, 2 };
    int base_list[] = { 0, 1, 1, 1, 2, 1, 2 };

    TEST_ASSERT_EQUAL(SUBLIST, check_lists(list_to_compare,
                                            list_to_compare+ELEMENT_COUNT(list_to_compare),
                                            base_list,
                                            base_list+ELEMENT_COUNT(base_list)));
}

static void test_sublist_at_start(void){
    TEST_IGNORE();
    int list_to_compare[] = { 0, 1, 2 };
    int base_list[] = { 0, 1, 2, 3, 4, 5 };

    TEST_ASSERT_EQUAL(SUBLIST, check_lists(list_to_compare,
                                            list_to_compare+ELEMENT_COUNT(list_to_compare),
                                            base_list,
                                            base_list+ELEMENT_COUNT(base_list)));
}

static void test_sublist_at_middle(void){
    TEST_IGNORE();
    int list_to_compare[] = { 2, 3, 4 };
    int base_list[] = { 0, 1, 2, 3, 4, 5 };

    TEST_ASSERT_EQUAL(SUBLIST, check_lists(list_to_compare,
                                            list_to_compare+ELEMENT_COUNT(list_to_compare),
                                            base_list,
                                            base_list+ELEMENT_COUNT(base_list)));
}

static void test_sublist_at_end(void){
    TEST_IGNORE();
    int list_to_compare[] = { 3, 4, 5 };
    int base_list[] = { 0, 1, 2, 3, 4, 5 };

    TEST_ASSERT_EQUAL(SUBLIST, check_lists(list_to_compare,
                                            list_to_compare+ELEMENT_COUNT(list_to_compare),
                                            base_list,
                                            base_list+ELEMENT_COUNT(base_list)));
}
```



```
static void test_at_start_of_superlist(void){
    TEST_IGNORE();
    int list_to_compare[] = { 0, 1, 2, 3, 4, 5 };
    int base_list[] = { 0, 1, 2 };

    TEST_ASSERT_EQUAL(SUPERLIST, check_lists(list_to_compare,
                                              list_to_compare+ELEMENT_COUNT(list_to_compare),
                                              base_list,
                                              base_list+ELEMENT_COUNT(base_list)));
}

static void test_in_middle_of_superlist(void){
    TEST_IGNORE();
    int list_to_compare[] = { 0, 1, 2, 3, 4, 5 };
    int base_list[] = { 2, 3 };

    TEST_ASSERT_EQUAL(SUPERLIST, check_lists(list_to_compare,
                                              list_to_compare+ELEMENT_COUNT(list_to_compare),
                                              base_list,
                                              base_list+ELEMENT_COUNT(base_list)));
}

static void test_at_end_of_superlist(void){
    TEST_IGNORE();
    int list_to_compare[] = { 0, 1, 2, 3, 4, 5 };
    int base_list[] = { 3, 4, 5 };

    TEST_ASSERT_EQUAL(SUPERLIST, check_lists(list_to_compare,
                                              list_to_compare+ELEMENT_COUNT(list_to_compare),
                                              base_list,
                                              base_list+ELEMENT_COUNT(base_list)));
}

static void test_first_list_missing_element_from_second_list(void){
    TEST_IGNORE();
    int list_to_compare[] = { 1, 3 };
    int base_list[] = { 1, 2, 3 };

    TEST_ASSERT_EQUAL(UNEQUAL, check_lists(list_to_compare,
                                              list_to_compare+ELEMENT_COUNT(list_to_compare),
                                              base_list,
                                              base_list+ELEMENT_COUNT(base_list)));
}

static void test_second_list_missing_element_from_first_list(void){
    TEST_IGNORE();
    int list_to_compare[] = { 1, 2, 3 };
    int base_list[] = { 1, 3 };

    TEST_ASSERT_EQUAL(UNEQUAL, check_lists(list_to_compare,
                                              list_to_compare+ELEMENT_COUNT(list_to_compare),
                                              base_list,
                                              base_list+ELEMENT_COUNT(base_list)));
}
```



```
static void test_first_list_missing_additional_digits_from_second_list(void){
    TEST_IGNORE();
    int list_to_compare[] = { 1, 2 };
    int base_list[] = { 1, 22 };

    TEST_ASSERT_EQUAL(UNEQUAL, check_lists(list_to_compare,
                                            list_to_compare+ELEMENT_COUNT(list_to_compare),
                                            base_list,
                                            base_list+ELEMENT_COUNT(base_list)));
}

static void test_order_matters_to_a_list(void){
    TEST_IGNORE();
    int list_to_compare[] = { 1, 2, 3 };
    int base_list[] = { 3, 2, 1 };

    TEST_ASSERT_EQUAL(UNEQUAL, check_lists(list_to_compare,
                                            list_to_compare+ELEMENT_COUNT(list_to_compare),
                                            base_list,
                                            base_list+ELEMENT_COUNT(base_list)));
}

static void test_same_digits_but_different_numbers(void){
    TEST_IGNORE();
    int list_to_compare[] = { 1, 0, 1 };
    int base_list[] = { 10, 1 };

    TEST_ASSERT_EQUAL(UNEQUAL, check_lists(list_to_compare,
                                            list_to_compare+ELEMENT_COUNT(list_to_compare),
                                            base_list,
                                            base_list+ELEMENT_COUNT(base_list)));
}

static void test_different_signs(void){
    TEST_IGNORE();
    int list_to_compare[] = { 1, 2, 3 };
    int base_list[] = { 1, -2, 3 };

    TEST_ASSERT_EQUAL(UNEQUAL, check_lists(list_to_compare,
                                            list_to_compare+ELEMENT_COUNT(list_to_compare),
                                            base_list,
                                            base_list+ELEMENT_COUNT(base_list)));
}

int main(void){
    UnityBegin("test_sublist.c");
    RUN_TEST(test_empty_lists);
    RUN_TEST(test_empty_list_within_non_empty_list);
    RUN_TEST(test_non_empty_list_contains_empty_list);
    RUN_TEST(test_list_equals_itself);
    RUN_TEST(test_different_lists);
    RUN_TEST(test_false_start);
    RUN_TEST(test_consecutive);
    RUN_TEST(test_sublist_at_start);
    RUN_TEST(test_sublist_at_middle);
    RUN_TEST(test_sublist_at_end);
    RUN_TEST(test_at_start_of_superlist);
    RUN_TEST(test_in_middle_of_superlist);
    RUN_TEST(test_at_end_of_superlist);
    RUN_TEST(test_first_list_missing_element_from_second_list);
    RUN_TEST(test_second_list_missing_element_from_first_list);
    RUN_TEST(test_first_list_missing_additional_digits_from_second_list);
    RUN_TEST(test_order_matters_to_a_list);
    RUN_TEST(test_same_digits_but_different_numbers);
    RUN_TEST(test_different_signs);
    return UnityEnd();
}
```




sublist.c

```

1  #include "sublist.h"
2  comparison_result_t check_lists(int *list_to_compare_begin, int *list_to_compare_end,
3                                  int *base_list_begin, int *base_list_end){
4
5      //Test 01
6      //Primero verificamos que no esten vacios los dos
7      if(list_to_compare_begin == NULL && base_list_begin == NULL){
8          return EQUAL;
9      }
10     //Test 02
11     //Revisamos si la lista a comparar esta vacia, pero otra no
12     else if(base_list_begin != NULL && list_to_compare_begin == NULL){
13         return SUBLIST;
14     }
15     //Test 03
16     //Revisamos si la lista base esta vacia, pero la segunda no
17     else if(list_to_compare_begin != NULL && base_list_begin == NULL){
18         return SUPERLIST;
19     }
20     //Cuando las dos listas no estan vacias
21     else{
22         //Cuando son del mismo tamaño
23         if(list_to_compare_end - list_to_compare_begin == base_list_end - base_list_begin){
24             int *alc = list_to_compare_begin;
25             int *alb = base_list_begin;
26             //Compara uno por uno
27             while(alc < list_to_compare_end){
28                 if(*alc == *alb){
29                     alc++;
30                     alb++;
31                 }
32                 //Si hay por lo menos uno diferente
33                 //Test 05, 16, 17, 19
34                 else{
35                     return UNEQUAL;
36                 }
37             }
38             //Si el ciclo termina sin diferencias
39             //Test 04
40             return EQUAL;
41         }
42         //Cuando la lista base es mas grande
43         else if(base_list_end - base_list_begin > list_to_compare_end - list_to_compare_begin){
44             int *alc = list_to_compare_begin;
45             int *alb = base_list_begin;
46             int t = 0;
47             int *tmp = NULL;
48             while(alb < base_list_end){
49                 if(*alc == *alb){
50                     alc++;
51                     t++;
52                     if(t == 1){
53                         tmp = alb;
54                     }
55                 }
56                 else if(t == list_to_compare_end - list_to_compare_begin){

```



```
55         return SUBLIST;
56     }
57     alb++;
58 }
59 else{
60     alc = list_to_compare_begin;
61     if(t == 0){
62         alb++;
63     }
64     else{
65         t = 0;
66         alb = tmp + 1;
67     }
68 }
69 }
70 return UNEQUAL;
71 }
72 //Cuando la lista a comparar es más grande
73 else{
74     int *alc = list_to_compare_begin;
75     int *alb = base_list_begin;
76     int t = 0;
77     int *tmp = NULL;
78     while(alc < list_to_compare_end){
79         if(*alc == *alb){
80             alb++;
81             t++;
82             if(t == 1){
83                 tmp = alc;
84             }
85             else if(t == base_list_end - base_list_begin){
86                 return SUPERLIST;
87             }
88             alc++;
89         }
90         else{
91             alb = base_list_begin;
92             if(t == 0){
93                 alc++;
94             }
95             else{
96                 t = 0;
97                 alc = tmp + 1;
98             }
99         }
100     }
101     return UNEQUAL;
102 }
103 }
104 }
```



1. Explique su algoritmo para determinar que dos arreglos son iguales.

```

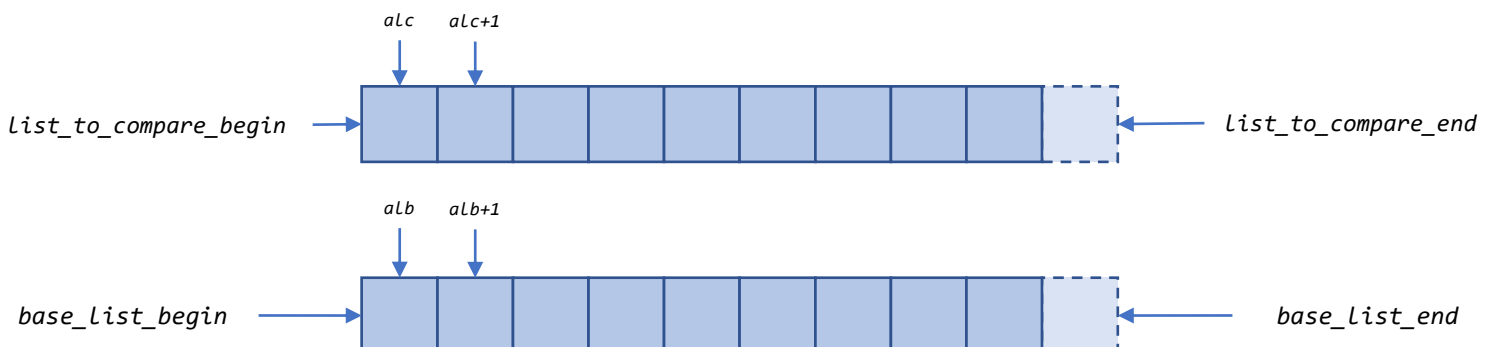
5 //Primero verificamos que no esten vacios los dos
6 if(list_to_compare_begin == NULL && base_list_begin == NULL){
7     return EQUAL;
8 }
19 //Cuando las dos listas no estan vacias
20 else{
21     //Cuando son del mismo tamaño
22     if(list_to_compare_end - list_to_compare_begin == base_list_end - base_list_begin){
23         int *alc = list_to_compare_begin;
24         int *alb = base_list_begin;
25         //Compara uno por uno
26         while(alc < list_to_compare_end){
27             if(*alc == *alb){
28                 alc++;
29                 alb++;
30             }
31             //Si hay por lo menos uno diferente
32             //Test 05, 16, 17, 19
33             else{
34                 return UNEQUAL;
35             }
36         }
37         //Si el ciclo termina sin diferencias
38         //Test 04
39         return EQUAL;
40     }

```

Primero pongo un caso base para cuando los dos arreglos están vacíos (líneas de código de la 6 a la 8), si los apuntadores del inicio apuntan a NULL, quiere decir que no hay elementos en ninguno y son iguales.

Después, para cuando los arreglos no están vacíos y son del mismo tamaño (líneas de código de la 22 a la 40), primero confirmo eso con un condicional `if`, en el que utilizo aritmética de apuntadores restando el final con el principio de cada arreglo y luego los comparo. Una vez hecho eso, creo dos apuntadores, donde cada uno apunta al principio de cada arreglo, esto para no mover de lugar los originales y para escribir menos. Luego doy inicio a un ciclo `while` que se va a repetir desde el principio del arreglo a comparar (podría ser cualquiera, ya que son del mismo tamaño), hasta que el apuntador quede en donde termina el mismo arreglo.

Dentro del ciclo `while` hay un condicional `if` en donde comparo si lo que hay en el principio del arreglo base, es lo mismo que hay en el arreglo a comparar, en caso de que sea verdad, le sumo uno a cada apuntador para comparar los siguientes, en caso de que haya una diferencia, la función regresa `UNEQUAL` y por ende termina la función, en caso de que nunca haya uno diferente, el ciclo `while` termina sin entrar en el `else`, y regresa `EQUAL`.





2. Explique su algoritmo para determinar que un arreglo es subarreglo de otro.

```
41 //Cuando la lista base es mas grande
42 else if(base_list_end - base_list_begin > list_to_compare_end - list_to_compare_begin){
43     int *alc = list_to_compare_begin;
44     int *alb = base_list_begin;
45     int t = 0;
46     int *tmp = NULL;
47     while(alb < base_list_end){
48         if(*alc == *alb){
49             alc++;
50             t++;
51             if(t == 1){
52                 tmp = alb;
53             }
54             else if(t == list_to_compare_end - list_to_compare_begin){
55                 return SUBLIST;
56             }
57             alb++;
58         }
59         else{
60             alc = list_to_compare_begin;
61             if(t == 0){
62                 alb++;
63             }
64             else{
65                 t = 0;
66                 alb = tmp + 1;
67             }
68         }
69     }
70     return UNEQUAL;
71 }
```

Lo primero es comprobar que el arreglo base sea más grande, eso lo hice con aritmética de apuntadores (línea 42). Aquí lo que hago es un ciclo para recorrer todo el arreglo base para buscar que estén todos los números del arreglo a comparar, así mismo, debo corroborar que estén en el mismo orden, sin importar en que parte del arreglo base, pero que vayan seguidos y en el mismo orden, entonces cada vez que encuentro al primero en coincidir, guardo esa dirección en un apuntador temporal, esto por si el orden se rompe, y así poder volver a comenzar una casilla después de esa. También puse un contador t, el cual me decía si los encontraba a todos, porque podía darse el caso de que encontrara solo a la mitad en el mismo orden, y al final, entonces eso me permitía saber si fueron todos, en el momento en que los encontraba a todos, regresaba SUBLIST, y en caso contrario, UNEQUAL.



3. Explique su algoritmo para determinar que un arreglo es superarreglo de otro.

```
72      //Cuando la lista a comparar es más grande
73      else{
74          int *alc = list_to_compare_begin;
75          int *alb = base_list_begin;
76          int t = 0;
77          int *tmp = NULL;
78          while(alc < list_to_compare_end){
79              if(*alc == *alb){
80                  alb++;
81                  t++;
82                  if(t == 1){
83                      tmp = alc;
84                  }
85                  else if(t == base_list_end - base_list_begin){
86                      return SUPERLIST;
87                  }
88                  alc++;
89              }
90              else{
91                  alb = base_list_begin;
92                  if(t == 0){
93                      alc++;
94                  }
95                  else{
96                      t = 0;
97                      alc = tmp + 1;
98                  }
99              }
100          }
101          return UNEQUAL;
102      }
```

De la línea 73 a la 102, está el algoritmo para una superlista, que es el caso cuando el arreglo a comparar es más grande que el arreglo base, básicamente es el mismo código que el de la sublista, con la diferencia de que cambio de lugar las variables del arreglo base con las del arreglo a comparar, y viceversa.



4. ¿Qué es una enumeración (enum) en el lenguaje C?

Es definir un conjunto de cadenas de texto, en donde cada cadena de texto equivale a un valor entero (int), que en este caso, será el índice en el que fue colocado en enum.



5. ¿Cuál es el nombre de la prueba (test) que le resultó más difícil pasar?

Pues en sí no estuvieron difíciles, pero digamos que las que me tomaron más tiempo en pasar, fueron las de sublistas y superlistas, esto fue porque leí mal las indicaciones y tuve que volver a escribirlas porque al principio solo checaba que estuvieran todos los valores, no que estuvieran en el mismo orden y todos seguidos.



6. ¿Hubo alguna prueba (test) que no pudo pasar? ¿Cuál fue? ¿Qué errores obtuvo?

No, si las pude pasar todas.



7. ¿Aplicó la refactorización en su código? Explique de qué manera lo hizo

Sí, al principio resolví todo dentro de esa misma función, entonces creé más funciones para separar el código, dejando únicamente los casos base en la función original, eso también ayudo a no repetir el código de superlista y sublistas, a continuación pongo cada función.

Prototipos de funciones y función principal

```
1  #include "sublist.h"
2
3  comparison_result_t iguales(int *alc, int *alb, int *blc);
4
5  comparison_result_t tipo_arr(int *alc, int *alb, int *blc, int *blb);
6
7  comparison_result_t check_lists(int *list_to_compare_begin, int *list_to_compare_end,
8                                int *base_list_begin, int *base_list_end){
9      if(list_to_compare_begin == NULL && base_list_begin == NULL){
10         return EQUAL;
11     }
12     else if(base_list_begin != NULL && list_to_compare_begin == NULL){
13         return SUBLIST;
14     }
15     else if(list_to_compare_begin != NULL && base_list_begin == NULL){
16         return SUPERLIST;
17     }
18     else{
19         if(list_to_compare_end - list_to_compare_begin == base_list_end - base_list_begin){
20             return iguales(list_to_compare_begin, base_list_begin, list_to_compare_end);
21         }
22         else if(base_list_end - base_list_begin > list_to_compare_end - list_to_compare_begin){
23             return tipo_arr(list_to_compare_begin, base_list_begin, list_to_compare_end, base_list_end);
24         }
25         else{
26             if(tipo_arr(base_list_begin, list_to_compare_begin, base_list_end, list_to_compare_end) == 1){
27                 return UNEQUAL;
28             }
29             else{
30                 return SUPERLIST;
31             }
32         }
33     }
34 }
```



Función para comprobar si son iguales

```
1  comparison_result_t iguales(int *alc, int *alb, int *blc){
2      while(alc < blc){
3          if(*alc == *alb){
4              alc++;
5              alb++;
6          }
7          else{
8              return UNEQUAL;
9          }
10     }
11     return EQUAL;
12 }
```

Función para saber si es sublista o superlista

```
1  comparison_result_t tipo_arr(int *alc, int *alb, int *blc, int *blb){
2      int *beg = alc;
3      int *beg2 = blc;
4      int t = 0;
5      int *tmp = NULL;
6      while(alb < blb){
7          if(*alc == *alb){
8              alc++;
9              t++;
10             if(t == 1){
11                 tmp = alb;
12             }
13             else if(t == beg2 - beg){
14                 return SUBLIST;
15             }
16             alb++;
17         }
18         else{
19             alc = beg;
20             if(t == 0){
21                 alb++;
22             }
23             else{
24                 t = 0;
25                 alb = tmp + 1;
26             }
27         }
28     }
29     return UNEQUAL;
30 }
```



8. ¿Qué hizo particularmente bien en esta tarea?

Aplicar correctamente aritmética de apuntadores en mi código, así como la implementación de apuntadores en funciones.



9. ¿Qué pudo haber hecho mejor en la tarea?

Tomarme el tiempo suficiente para leer a detalle los requerimientos de la tarea, para no tener que volver a reescribir mi código.

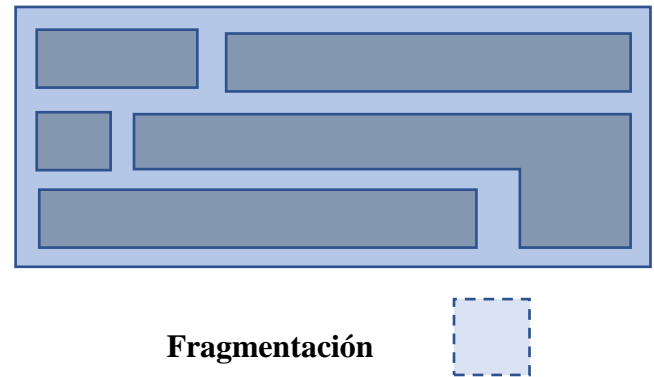
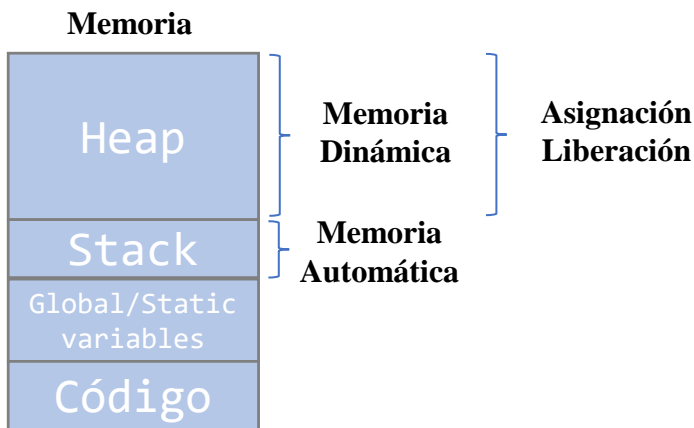


10. ¿Qué conocimientos nuevos adquirió?

Primeramente, el enum, porque no lo conocía, en segunda, esta tarea me ayudo a terminar de entender la aritmética de apuntadores.

1.6 Gestión de memoria dinámica

Lunes 29 Agosto 2022



Asignación

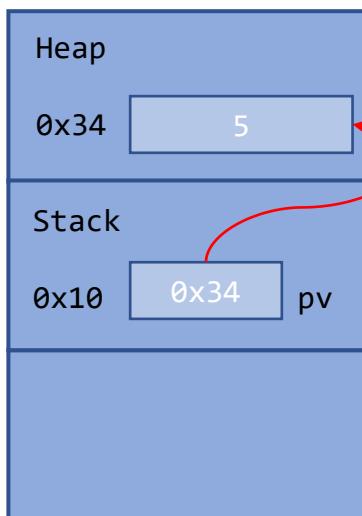
`void *malloc(size_t);`
 Regresa un apuntador con la dirección del primer elemento del bloque.

Ejemplo:

```
int *pv = (int *)malloc(sizeof(int));
*pv = 5;
printf("%d\n", *pv); //output: 5
```

size_t es un tipo de dato unsigned int. En este caso específico la cantidad de bytes por asignar.

void * regresa un apuntador con la dirección del primer elemento del bloque, y es void para poder usarlo en todo apuntador genérico.



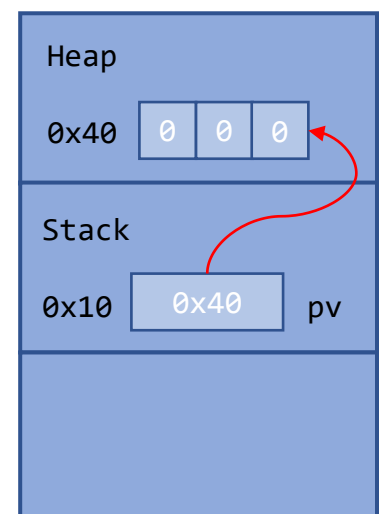
Liberación

```
void free(void *);
free(pv);
printf("%d\n", *pv); //output: random number
```

`free()` devuelve la memoria al sistema, ya no te pertenece

Para poder utilizar `malloc()` y `free()`, se debe agregar la librería `stdlib.h`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int *pv = (int *)malloc(3*sizeof(int));
5     *pv = 0;
6     *(pv + 1) = 0;
7     *(pv + 2) = 0;
8     free(pv);
9     return 0;
10 }
```



`free(pv);`





¿Cómo enterarnos si la asignación de memoria falla?

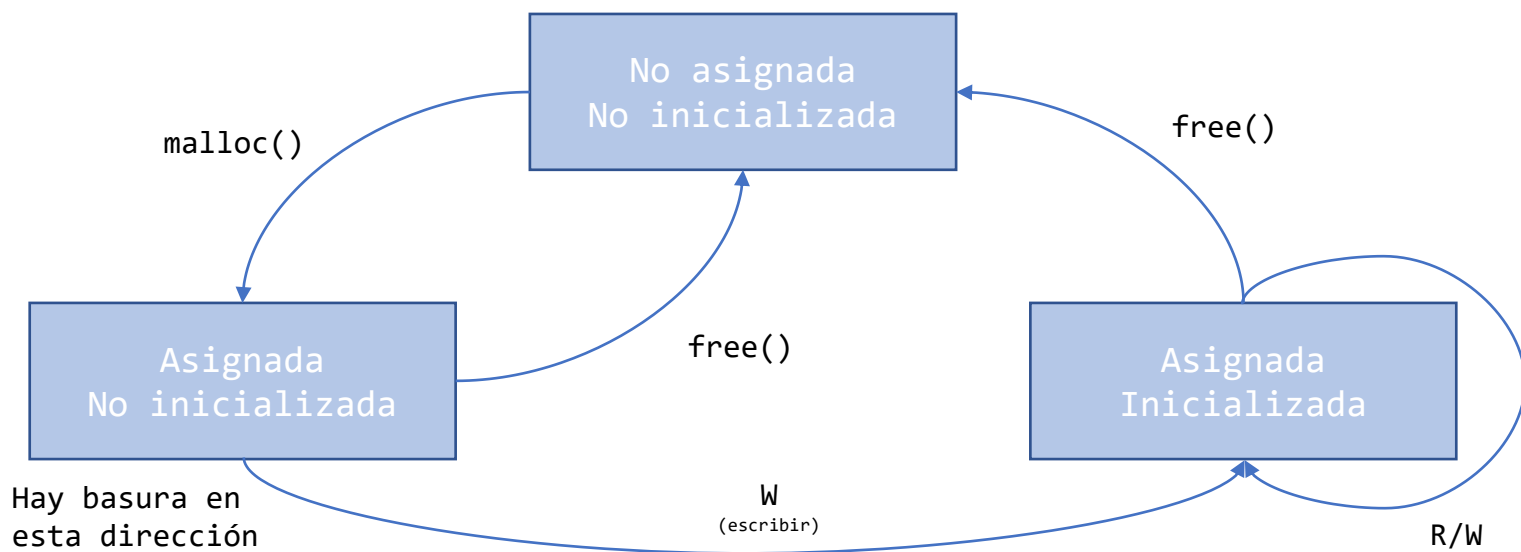
```

1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(){
4      int *pv = NULL;
5      pv = (int *)malloc(sizeof(int));
6      if(pv == NULL){
7          printf("Error: no hay memoria disponible");
8          exit(EXIT_FAILURE);
9      }
10     /*codigo*/
11     return 0;
12 }

```

1.7 Estados de memoria dinámica

Martes 30 Agosto 2022



1.8 Problemas en el manejo de memoria dinámica

Fragmentación de memoria, fuera de nuestro alcance, no hay un bloque contiguo lo suficientemente grande.

1) Fugas de memoria (memory leaks)

```

1  char *bytes;
2  while(1){
3      bytes = (char *)malloc(10);
4  }

```

Un ciclo infinito que en algún punto creará una fuga de memoria porque no hay memoria infinita.



```
1 double *d = NULL;
2 d = (double *)malloc(100);
3 double pi = 3.1416;
4 d = &pi;
5 free(d);
```

Al hacer eso, perdemos la dirección de memoria en donde teníamos guardado los 100 de memoria.

Lo correcto es hacer otro apuntador (en este caso f) en donde guardemos la dirección antes de cambiar el apuntador.

```
1 double *d = NULL;
2 d = (double *)malloc(100);
3 double pi = 3.1416;
4 double *f = d;
5 d = &pi;
6 free(f);
```

Por cada malloc() asigna un free()

2) Wild pointers (apuntadores no inicializados)

Apuntadores que no apuntan a nada.

Cada puntero se inicializa con un NULL

```
int *p = NULL;
```

3) Dangling pointers (apuntadores a una dirección de memoria que ya fue liberada)

Asignar NULL al apuntador después de free()

```
int *p = NULL;
p = (int *)malloc(sizeof(int));
free(p);
p = NULL;
```



```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <assert.h>
4  int *crea_arreglo(int);
5  int main(){
6      int n;
7      scanf("%d", &n);
8      int *a = crea_arreglo(n);
9      for(int i = 0; i < n; i++){
10         scanf("%d", &a[i]);
11     }
12     for(int j = 0; j < n; j++){
13         printf("%d ", a[j]);
14     }
15     printf("\n");
16     free(a);
17     return 0;
18 }
19 int *crea_arreglo(int tam){
20     int *arreglo = NULL;
21     arreglo = (int *)malloc(tam * sizeof(int));
22     assert(arreglo != NULL);
23     return arreglo;
24 }
```

Las líneas 9 y 12 las podemos cambiar, o bien, cambiar los dos ciclos for, tal como se muestra a continuación y exactamente lo mismo:

```
9      scanf("%d", a + i);
12     printf("%d ", *(a + j));

8      for(int *pi = a; pi < a + n; pi++){
9          scanf("%d", pi);
10     }
11     for(int *pj = a; pj < a + n; pj++){
12         printf("%d\n", *pj);
13     }
```

1.10

Matriz dinámica

Viernes 02 Septiembre 2022

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <assert.h>
4  int main(){
5      int *a[6] = {NULL};
6      for(int i = 0; i < 6; i++){
7          a[i] = (int *)malloc(5 * sizeof(int));
8          assert(a[i] != NULL);
9      }
10     for(int j = 0; j < 6; j++){
11         free(a[j]);
12     }
13     return 0;
14 }

```

También podemos liberar la matriz de esta forma:

```

10     int **p = a;
11     for(int j = 0; j < 6; j++){
12         free(*(p + j));
13     }
14     return 0;
15 }

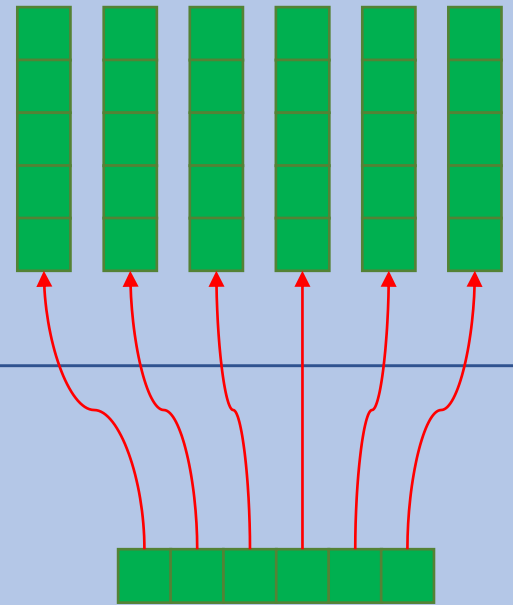
```

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <assert.h>
4  int main(){
5      int **a = NULL;
6      a = (int **)malloc(6 * sizeof(int *));
7      assert(a != NULL);
8      for(int i = 0; i < 6; i++){
9          a[i] = (int *)malloc(5 * sizeof(int));
10         assert(a[i] != NULL);
11     }
12     int **p = a;
13     for(int j = 0; j < 6; j++){
14         free(*(p + j));
15     }
16     free(a);
17     return 0;
18 }

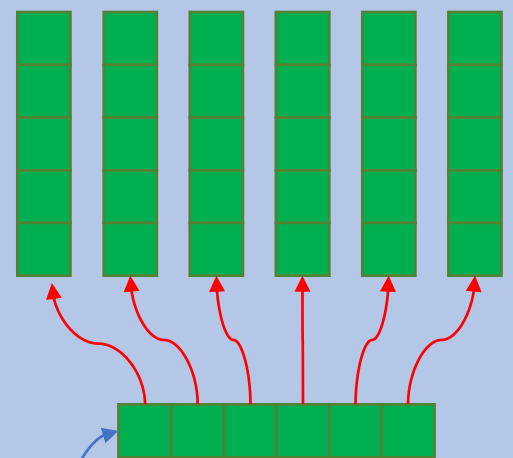
```

Heap



Stack

Heap



Stack



Implementación de una matriz dinámica con funciones

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <assert.h>
4  int **crea_matriz(int, int);
5  void llenar_matriz(int **, int, int);
6  void imprimir_matriz(int **, int, int);
7  void liberar_matriz(int **);
8  int main(){
9      int r, c;
10     scanf("%d", &r);
11     scanf("%d", &c);
12     int **a = crea_matriz(r, c);
13     llenar_matriz(a, r, c);
14     imprimir_matriz(a, r, c);
15     liberar_matriz(a);
16     return 0;
17 }
18
19 int **crea_matriz(int r, int c){
20     int **a = NULL;
21     a = (int **)malloc(c * sizeof(int *));
22     assert(a != NULL);
23     for(int i = 0; i < c; i++){
24         a[i] = (int *)malloc(r * sizeof(int));
25         assert(a[i] != NULL);
26     }
27     return a;
28 }
29
30 void llenar_matriz(int **matriz, int r, int c){
31     int **a = matriz;
32     for(int i = 0; i < c; i++){
33         for(int j = 0; j < r; j++){
34             scanf("%d", &a[i][j]);
35         }
36     }
37 }
38
39 void imprimir_matriz(int **matriz, int r, int c){
40     int **a = matriz;
41     for(int i = 0; i < c; i++){
42         for(int j = 0; j < r; j++){
43             printf("%d ", a[i][j]);
44         }
45         printf("\n");
46     }
47 }
48
49 void liberar_matriz(int **matriz){
50     int **p = matriz;
51     for(int j = 0; j < 6; j++){
52         free(*(p + j));
53     }
54     free(matriz);
55 }
```



Tarea #2: Arreglos Dinámicos

Minesweeper

If you need help running the tests or submitting your code, check out [HELP.md](#).

Instructions

Add the mine counts to a completed Minesweeper board.

Minesweeper is a popular game where the user has to find the mines using numeric hints that indicate how many mines are directly adjacent (horizontally, vertically, diagonally) to a square.

In this exercise you have to create some code that counts the number of mines adjacent to a given empty square and replaces that square with the count.

The board is a rectangle composed of blank space (' ') characters. A mine is represented by an asterisk ('*') character.

If a given space has no adjacent mines at all, leave that square blank.

Examples

For example, you may receive a 5 x 4 board like this (empty spaces are represented here with the '.' character for display on screen):

```
.*.*.
..*..
..*..
.....
```

And your code will transform it into this:

```
1*3*1
13*31
.2*2.
.111.
```



minesweeper.h

```
#ifndef MINESWEEPER_H
#define MINESWEEPER_H
#include <stddef.h>

char **annotate(const char **minefield, const size_t rows);
void free_annotation(char **annotation, const size_t rows);

#endif
```



test_minesweeper.c

```
#include "test-framework/unity.h"
#include "minesweeper.h"
#include <stdlib.h>
#define ARRAY_SIZE(arr) (sizeof(arr) / sizeof(arr[0]))
void setUp(void){
}

void tearDown(void){
}

static void test_annotate_no_rows(void){
    TEST_IGNORE(); // delete this line to run test
    const char **minefield = NULL;
    const size_t rows = 0;
    char **actual = annotate(minefield, rows);
    TEST_ASSERT_NULL(actual);
}

static void test_annotate_no_columns(void){
    TEST_IGNORE();
    const char *minefield[] = { "" };
    const char *expected[] = { "" };
    const size_t rows = ARRAY_SIZE(expected);
    char **actual = annotate(minefield, rows);
    TEST_ASSERT_EQUAL_STRING_ARRAY(expected, actual, rows);
    free_annotation(actual, rows);
}

static void test_annotate_no_mines(void){
    TEST_IGNORE();
    const char *minefield[] = {
        // clang-format off
        " ",
        " ",
        " "
        // clang-format on
    };
    const char *expected[] = {
        // clang-format off
        " ",
        " ",
        " "
        // clang-format on
    };
    const size_t rows = ARRAY_SIZE(expected);
    char **actual = annotate(minefield, rows);
    TEST_ASSERT_EQUAL_STRING_ARRAY(expected, actual, rows);
    free_annotation(actual, rows);
}
```




```
static void test_annotate_minefield_with_only_mines(void){
    TEST_IGNORE();
    const char *minefield[] = {
        // clang-format off
        "***",
        "***",
        "***"
        // clang-format on
    };
    const char *expected[] = {
        // clang-format off
        "***",
        "***",
        "***"
        // clang-format on
    };
    const size_t rows = ARRAY_SIZE(expected);
    char **actual = annotate(minefield, rows);
    TEST_ASSERT_EQUAL_STRING_ARRAY(expected, actual, rows);
    free_annotation(actual, rows);
}

static void test_annotate_mine_surrounded_by_spaces(void){
    TEST_IGNORE();
    const char *minefield[] = {
        // clang-format off
        "  ",
        " * ",
        "  "
        // clang-format on
    };
    const char *expected[] = {
        // clang-format off
        "111",
        "1*1",
        "111"
        // clang-format on
    };
    const size_t rows = ARRAY_SIZE(expected);
    char **actual = annotate(minefield, rows);
    TEST_ASSERT_EQUAL_STRING_ARRAY(expected, actual, rows);
    free_annotation(actual, rows);
}

static void test_annotate_space_surrounded_by_mines(void){
    TEST_IGNORE();
    const char *minefield[] = {
        // clang-format off
        "***",
        "* *",
        "***"
        // clang-format on
    };
};
```



```

const char *expected[] = {
    // clang-format off
    "***",
    "*8*",
    "***"
    // clang-format on
};
const size_t rows = ARRAY_SIZE(expected);
char **actual = annotate(minefield, rows);
TEST_ASSERT_EQUAL_STRING_ARRAY(expected, actual, rows);
free_annotation(actual, rows);
}

static void test_annotate_horizontal_line(void){
    TEST_IGNORE();
    const char *minefield[] = {
        // clang-format off
        " * * "
        // clang-format on
    };
    const char *expected[] = {
        // clang-format off
        "1*2*1"
        // clang-format on
    };
    const size_t rows = ARRAY_SIZE(expected);
    char **actual = annotate(minefield, rows);
    TEST_ASSERT_EQUAL_STRING_ARRAY(expected, actual, rows);
    free_annotation(actual, rows);
}

static void test_annotate_horizontal_line_mines_at_edges(void){
    TEST_IGNORE();
    const char *minefield[] = {
        // clang-format off
        "*      *"
        // clang-format on
    };
    const char *expected[] = {
        // clang-format off
        "*1 1*"
        // clang-format on
    };
    const size_t rows = ARRAY_SIZE(expected);
    char **actual = annotate(minefield, rows);
    TEST_ASSERT_EQUAL_STRING_ARRAY(expected, actual, rows);
    free_annotation(actual, rows);
}

static void test_annotate_vertical_line(void){
    TEST_IGNORE();
    const char *minefield[] = {
        // clang-format off
        " ",
        "*",
        " ",
        "*",
        " "
        // clang-format on
    };
};

```



```

const char *expected[] = {
    // clang-format off
    "1",
    "*",
    "2",
    "*",
    "1"
    // clang-format on
};
const size_t rows = ARRAY_SIZE(expected);
char **actual = annotate(minefield, rows);
TEST_ASSERT_EQUAL_STRING_ARRAY(expected, actual, rows);
free_annotation(actual, rows);
}

static void test_annotate_vertical_line_mines_at_edges(void){
    TEST_IGNORE();
    const char *minefield[] = {
        // clang-format off
        "*",
        " ",
        " ",
        " ",
        " "
        // clang-format on
    };
    const char *expected[] = {
        // clang-format off
        "*",
        "1",
        " ",
        "1",
        "*"
        // clang-format on
    };
    const size_t rows = ARRAY_SIZE(expected);
    char **actual = annotate(minefield, rows);
    TEST_ASSERT_EQUAL_STRING_ARRAY(expected, actual, rows);
    free_annotation(actual, rows);
}

static void test_annotate_cross(void){
    TEST_IGNORE();
    const char *minefield[] = {
        // clang-format off
        " * ",
        " * ",
        "*****",
        " * ",
        " * "
        // clang-format on
    };
    const char *expected[] = {
        // clang-format off
        " 2*2 ",
        "25*52",
        "*****",
        "25*52",
        " 2*2 "
        // clang-format on
    };
    const size_t rows = ARRAY_SIZE(expected);
    char **actual = annotate(minefield, rows);
    TEST_ASSERT_EQUAL_STRING_ARRAY(expected, actual, rows);
    free_annotation(actual, rows);
}

```



```
static void test_annotate_large_minefield(void){
    TEST_IGNORE();
    const char *minefield[] = {
        // clang-format off
        " * * ",
        " *  ",
        "  * ",
        " * * ",
        " * * ",
        " "
        // clang-format on
    };
    const char *expected[] = {
        // clang-format off
        "1*22*1",
        "12*322",
        " 123*2",
        "112*4*",
        "1*22*2",
        "111111"
        // clang-format on
    };
    const size_t rows = ARRAY_SIZE(expected);
    char **actual = annotate(minefield, rows);
    TEST_ASSERT_EQUAL_STRING_ARRAY(expected, actual, rows);
    free_annotation(actual, rows);
}

int main(void){
    UnityBegin("test_minesweeper.c");
    RUN_TEST(test_annotate_no_rows);
    RUN_TEST(test_annotate_no_columns);
    RUN_TEST(test_annotate_no_mines);
    RUN_TEST(test_annotate_minefield_with_only_mines);
    RUN_TEST(test_annotate_mine_surrounded_by_spaces);
    RUN_TEST(test_annotate_space_surrounded_by_mines);
    RUN_TEST(test_annotate_horizontal_line);
    RUN_TEST(test_annotate_horizontal_line_mines_at_edges);
    RUN_TEST(test_annotate_vertical_line);
    RUN_TEST(test_annotate_vertical_line_mines_at_edges);
    RUN_TEST(test_annotate_cross);
    RUN_TEST(test_annotate_large_minefield);
    return UnityEnd();
}
```



minesweeper.c

```
1  #include "minesweeper.h"
2  #include <string.h>
3  #include <stdlib.h>
4  #include <assert.h>
5
6  char **annotate(const char **minefield, const size_t rows){
7      //Caso base cuando no existe la matriz
8      if(minefield == NULL){
9          return NULL;
10     }
11     int c = strlen(minefield[0]);
12     int cont = 0;
13     int r = rows;
14     //Crear la matriz
15     char **a = NULL;
16     a = (char **)malloc(r * sizeof(char *));
17     assert(a != NULL);
18     for(int i = 0; i < r; i++){
19         a[i] = (char *)malloc((c + 1) * sizeof(char));
20         assert(a[i] != NULL);
21     }
22     //Copiando la matriz para ya tener asteriscos y espacios
23     for(int i = 0; i < r; i++){
24         for(int j = 0; j < c + 1; j++){
25             if(j != c){
26                 a[i][j] = minefield[i][j];
27             }
28             else{
29                 a[i][j] = '\0';
30             }
31         }
32     }
33     //Caso general para todas las matrices
34     for(int i = 0; i < r; i++){
35         for(int j = 0; j < c; j++){
36             //Si no es un asterisco
37             if(a[i][j] != '*'){
38                 //Fijamos los limites para revisar
39                 int r_begin, r_end, c_begin, c_end;
40                 //Si estamos al principio de un renglon
41                 if(i == 0){
42                     r_begin = 0;
43                 }
44                 else{
45                     r_begin = i - 1;
46                 }
47                 //Si estamos al final de un renglon
48                 if(i == r - 1){
49                     r_end = r - 1;
50                 }
51             }
52         }
53     }
54 }
```



```
51         else{
52             r_end = i + 1;
53         }
54         //Si estamos en la primer columna
55         if(j == 0){
56             c_begin = 0;
57         }
58         else{
59             c_begin = j - 1;
60         }
61         //Si estamos en la ultima columna
62         if(j == c - 1){
63             c_end = c - 1;
64         }
65         else{
66             c_end = j + 1;
67         }
68         //Revisamos si los de alrededor son asteriscos
69         for(int k = r_begin; k < r_end + 1; k++){
70             for(int l = c_begin; l <= c_end; l++){
71                 if(a[k][l] == '*'){
72                     cont++;
73                 }
74             }
75         }
76         //Si hay por lo menos un asterisco cerca
77         if(cont != 0){
78             a[i][j] = cont + '0';
79             cont = 0;
80         }
81     }
82 }
83 }
84 return a;
85 }
86
87 void free_annotation(char **annotation, const size_t rows){
88     char **p = annotation;
89     int r = rows;
90     for(int i = 0; i < r; i++){
91         free(*(p + i));
92     }
93     free(annotation);
94 }
```



1. Explica tus algoritmos para crear y liberar la memoria dinámica utilizada.

```

14 //Crear la matriz
15 char **a = NULL;
16 a = (char **)malloc(r * sizeof(char *));
17 assert(a != NULL);
18 for(int i = 0; i < r; i++){
19     a[i] = (char *)malloc((c + 1) * sizeof(char));
20     assert(a[i] != NULL);
21 }

87 void free_annotation(char **annotation, const size_t rows){
88     char **p = annotation;
89     int r = rows;
90     for(int i = 0; i < r; i++){
91         free(*(p + i));
92     }
93     free(annotation);
94 }

```

Para la creación de la memoria dinámica (líneas de la 14 a la 21), y la liberación de memoria (líneas de la 87 a la 94) utilicé el código aprendido el viernes 02 de septiembre, en donde primero es crear un apuntador que apunta a otro apuntador, primero, ya sabemos cuantos renglones hay, entonces creo un arreglo dinámico de apuntadores del mismo tamaño de los renglones, luego cada casilla de memoria (cada uno de los renglones), apunta a otra cadena de memoria del tamaño de cada renglón + 1 (el más uno es para poder el final de cadena ‘\0’) el tamaño de cada renglón lo obtuve en la línea 11 con la función `strlen()`. Así mismo, aplico el `assert()` para comprobar que hay memoria suficiente.

Para la liberación de la memoria, también aplico el código visto en clase el día 02, al liberar primero cada uno de los renglones, y posteriormente liberar la matriz de renglones.



2. Explica tu algoritmo para determinar cuántas minas hay alrededor de cada celda.

```

34 for(int i = 0; i < r; i++){
35     for(int j = 0; j < c; j++){
36         //Si no es un asterisco
37         if(a[i][j] != '*'){
38             //Fijamos los limites para revisar
39             int r_begin, r_end, c_begin, c_end;
40             //Si estamos al principio de un renglon
41             if(i == 0){
42                 r_begin = 0;
43             }
44             else{
45                 r_begin = i - 1;
46             }
47             //Si estamos al final de un renglon
48             if(i == r - 1){
49                 r_end = r - 1;

```



```
50     }
51     else{
52         r_end = i + 1;
53     }
54     //Si estamos en la primer columna
55     if(j == 0){
56         c_begin = 0;
57     }
58     else{
59         c_begin = j - 1;
60     }
61     //Si estamos en la ultima columna
62     if(j == c - 1){
63         c_end = c - 1;
64     }
65     else{
66         c_end = j + 1;
67     }
68     //Revisamos si los de alrededor son asteriscos
69     for(int k = r_begin; k < r_end + 1; k++){
70         for(int l = c_begin; l <= c_end; l++){
71             if(a[k][l] == '*'){
72                 cont++;
73             }
74         }
75     }
76     //Si hay por lo menos un asterisco cerca
77     if(cont != 0){
78         a[i][j] = cont + '0';
79         cont = 0;
80     }
81 }
82 }
83 }
```

Primero hago dos ciclos for para hacer el recorrido por toda la matriz y compruebo que estoy en una casilla en blanco, al llegar a una, hago varios condicionales para saber si estoy en alguna de las orillas de la matriz o no, para saber que de donde a donde debo checar, y de este modo, no acceder a memoria que no me pertenece.

Luego de fijar los limites a revisar, hago otros dos ciclos for para revisar todas las casillas de alrededor en busca de asteriscos, en caso de encontrar una, aumento en uno mi contador de minas, y al final pongo ese numero en forma de char, y reinicio mi contador de bombas para la siguiente casilla en blanco.



3. ¿Qué es apuntador constante (const) en el lenguaje C? ¿En cuáles situaciones se recomienda utilizarlo?

Es cuando el valor al que apunta el apuntador va a ser constante

```
const int *p; // El contenido apunta al puntero es constante
int const *P; // El contenido apunta al puntero es constante.
int * const page; // puntero es constante
const int * const p; // El contenido apunta al puntero y el puntero es constante.
```

Se recomienda utilizarlo cuando solo vamos a leer los datos y no a modificarlos.



4. ¿Cuál es el nombre de la prueba (test) que te resultó más difícil pasar? ¿Por qué consideras que fue difícil?

Ninguna.



5. ¿Hubo alguna prueba (test) que no se pudo pasar? ¿Cuál fue? ¿Qué errores obtuviste?

No.



6. Inventa una nueva prueba y agrégala al conjunto de pruebas del proyecto. Agrega el código fuente y explicación de tu prueba en el reporte.

```
1 static void test_esquinas(void){
2     //TEST_IGNORE();
3     const char *minefield[] = {
4         // clang-format off
5         "* *",
6         "  ",
7         "* *"
8         // clang-format on
9     };
10    const char *expected[] = {
11        // clang-format off
12        "*2*",
13        "242",
14        "*2*"
15        // clang-format on
16    };
17    const size_t rows = ARRAY_SIZE(expected);
18    char **actual = annotate(minefield, rows);
19    TEST_ASSERT_EQUAL_STRING_ARRAY(expected, actual, rows);
20    free_annotation(actual, rows);
21 }
22
```



```

23 static void test_one_void(void){
24     //TEST_IGNORE();
25     const char *minefield[] = { " " };
26     const char *expected[] = { " " };
27     const size_t rows = ARRAY_SIZE(expected);
28     char **actual = annotate(minefield, rows);
29     TEST_ASSERT_EQUAL_STRING_ARRAY(expected, actual, rows);
30     free_annotation(actual, rows);
31 }
32
33 static void test_one_bomb(void){
34     //TEST_IGNORE();
35     const char *minefield[] = { "*" };
36     const char *expected[] = { "*" };
37     const size_t rows = ARRAY_SIZE(expected);
38     char **actual = annotate(minefield, rows);
39     TEST_ASSERT_EQUAL_STRING_ARRAY(expected, actual, rows);
40     free_annotation(actual, rows);
41 }

```

Agregué 3 pruebas, la primera es cuando únicamente las esquinas tienen bombas, la segunda cuando solo hay una casilla en blanco, y la tercera cuando solo hay una bomba.



7. ¿Aplicaste la refactorización en tu código? Explica de qué manera lo hiciste

```

38         //Fijamos los limites para revisar
39         int r_begin = (i == 0) ? 0: i - 1;
40         int r_end = (i == r - 1) ? r - 1 : i + 1;
41         int c_begin = (j == 0) ? 0: j - 1;
42         int c_end = (j == c - 1) ? c - 1 : j + 1;

```

Para no ocupar tantas líneas en los if else para fijar los limites, lo cambie por esa otra estructura.



8. ¿Qué hiciste particularmente bien en esta tarea?

Aplicar los conocimientos de los arreglos dinámicos para la creación y liberación de memoria.



9. ¿Qué pudiste haber hecho mejor en esta tarea?

Honestamente, no lo sé.



10. ¿Qué nuevos conocimientos y experiencias adquiriste con esta tarea?

A utilizar el ? y los : en vez de if y else.



1.11

Apuntadores por “referencia”

Lunes 05 Septiembre 2022

```

1  int *crea_arreglo(int tam){
2      int *arreglo = NULL;
3      arreglo = (int *)malloc(tam * sizeof(int));
4      assert(arreglo != NULL);
5      return arreglo;
6  }

```

V1

```

1  void crea_arreglo(int **arreglo, int tam){
2      *arreglo = (int *)malloc(tam * sizeof(int));
3      assert(*arreglo != NULL);
4  }

```

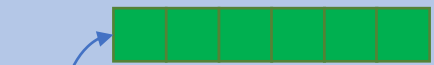
V2

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <assert.h>
4  void crea_arreglo(int **, int);
5  int main(){
6      int *arreglo = NULL;
7      crea_arreglo(&arreglo, 10);
8      for(int i = 0; i < 10; i++){
9          scanf("%d", arreglo + i);
10     }
11     for(int i = 0; i < 10; i++){
12         printf("%d", *(arreglo + i));
13     }
14     free(arreglo);
15     return 0;
16 }
17 void crea_arreglo(int **arreglo, int tam){
18     *arreglo = (int *)malloc(tam * sizeof(int));
19     assert(*arreglo != NULL);
20 }

```

Heap



Stack

a



1.12

realloc

Martes 06 Septiembre 2022

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(){
4      int *p = (int *)malloc(4 * sizeof(int));
5      p = (int *)realloc(p, 6 * sizeof(int));
6      p = (int *)realloc(p, 2 * sizeof(int));
7      free(p);
8      return 0;
9  }

```

La función realloc es para cambiar el tamaño de la memoria almacenada.



1.13

Pila dinámica

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <assert.h>
4  typedef struct{
5      int *arreglo;
6      int tope;
7      int max;
8  }pila_dinamica;
9
10 void inicializa_pila(pila_dinamica *);
11 void push(pila_dinamica *, int);
12 void pop(pila_dinamica *);
13 void imprimir_pila(pila_dinamica *);
14
15 int main(){
16     pila_dinamica pila;
17     inicializa_pila(&pila);
18     push(&pila, 2);
19     push(&pila, 34);
20     push(&pila, 146);
21     imprimir_pila(&pila);
22     pop(&pila);
23     push(&pila, 2);
24     imprimir_pila(&pila);
25     free(pila.arreglo);
26     return 0;
27 }
28
29 void inicializa_pila(pila_dinamica *p){
30     p->arreglo = NULL;
31     p->tope = p->max = 0;
32 }
33
34 void push(pila_dinamica *p, int dato){
35     if(p->tope == p->max){
36         p->max = p->max == 0 ? 1 : p->max * 2;
37         int *q = (int *)realloc(p->arreglo, p->max * sizeof(int));
38         assert(q != NULL);
39         p->arreglo = q;
40     }
41     p->arreglo[p->tope++] = dato;
42 }
43
44 void pop(pila_dinamica *p){
45     if(p->tope > 0){
46         int *q = (int *)realloc(p->arreglo, (p->tope - 1) * sizeof(int));
47         assert(q != NULL);
48         p->arreglo = q;
49         p->tope--;
50     }
51 }
52
53 void imprimir_pila(pila_dinamica *p){
54     if(p->tope != 0){
55         for(int i = 0; i < p->tope; i++){
56             printf("%d ", p->arreglo[i]);
57         }
58         printf("\n");
59     }
60     else{
61         printf("La pila esta vacia\n");
62     }
63 }

```



```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <assert.h>
4  #include <string.h>
5
6  typedef struct{
7      char nombre[50];
8      int edad;
9  }persona_t;
10
11 typedef struct{
12     char *nombre;
13     int edad;
14 }dpersona_t;
15
16 int main(){
17     persona_t persona;
18     printf("Nombre: ");
19     scanf("%[^\n]", persona.nombre);
20     printf("Edad: ");
21     scanf("%d", &persona.edad);
22     persona_t persona2 = persona;
23     printf("Nombre: %s\n", persona2.nombre);
24     printf("Edad: %d\n", persona2.edad);
25     dpersona_t persona_d;
26     char nombre[50];
27     char espacio;
28     scanf("%c", &espacio);
29     scanf("%[^\n]", nombre);
30     persona_d.nombre = (char *)malloc(strlen(nombre) + 1);
31     strcpy(persona_d.nombre, nombre);
32     printf("Nombre dinamico: %s\n", persona_d.nombre);
33     persona_t personas[10];
34     dpersona_t *personas_d = NULL;
35     personas_d = (dpersona_t *)malloc(10 * sizeof(dpersona_t));
36     return 0;
37 }
```



1.15

Apuntadores genéricos

Viernes 09 Septiembre 2022

void * = apuntador genérico porque puede apuntar a cualquier cosa

No se puede usar indirección sobre void * (se debe hacer cast)

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <assert.h>
4  #include <string.h>
5
6  typedef struct{
7      char nombre[50];
8      int edad;
9  }persona_t;
10
11 int main(){
12     int a = 100;
13     float b = 3.1416;
14     persona_t c;
15     strcpy(c.nombre, "Hugo Esparza");
16     c.edad = 26;
17     void *p = NULL;
18     p = &a;
19     printf("%d\n", *(int *)p);
20     p = &b;
21     printf("%f\n", *(float *)p);
22     p = &c;
23     printf("%s\n", ((persona_t *)p)->nombre);
24     return 0;
25 }
```

p->nombre ⇒ ((persona_t *)p)->nombre

(*p).nombre ⇒ (*(persona_t *)p).nombre

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5      void *p;
6      p = malloc(10);
7      printf("Dame el valor del entero: ");
8      scanf("%d", (int *)p);
9      printf("El entero es: %d\n", *(int *)p);
10     printf("Dame el valor del flotante: ");
11     scanf("%f", (float *)p);
12     printf("El flotante es: %f\n", *(float *)p);
13     return 0;
14 }
```



Trabajo en clase: “Arreglo dinámico”

Lunes 12 Septiembre 2022

Escribir un programa para escribir/leer datos en un arreglo dinámico por medio de un apuntador genérico. El usuario especificará el tamaño del arreglo y el tipo de dato seleccionado una de las siguientes opciones: 1 = int, 2 = char, 3 = float, 4 = alumno con clave (int) y nombre (char *). Se debe reservar memoria para el arreglo de acuerdo al tamaño y tipo de dato especificado por el usuario (todos los datos que contiene el arreglo serán del mismo tipo). Después de elegir el tipo de dato el usuario podrá escribir información en el arreglo e imprimir su contenido para verificar que la información se guardó correctamente. En cualquiera de los casos el arreglo se accederá por medio de una misma variable (void *). El programa terminará hasta que el usuario elija la opción 0 para salir, por lo que se deberá liberar la memoria antes de elegir un nuevo tipo de dato. Utilice notación/aritmética de apuntadores para acceder a los elementos del arreglo.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <assert.h>
4  #include <string.h>
5  #include <stdbool.h>
6
7  typedef enum{
8      SALIR,
9      ENTERO,
10     LETRA,
11     FLOTANTE,
12     ALUMNO
13 }tipo t;
14
15 typedef struct{
16     int clave;
17     char *nombre;
18 }alumno t;
19
20 tipo t seleccionar_tipo();
21 void crea_arreglo(void **, size_t, tipo t);
22 int menu();
23 void escribir(void **, size_t, tipo t);
24 void imprimir(void **, size_t, tipo t);
25 void liberar_arreglo(void **, size_t, tipo t);
26
27 int main(){
28     void *arreglo = NULL;
29     size_t tam;
30     bool flag = true;
31     int opt, o = 0;
32     tipo t tipo;
33     do{
```



```
34     printf("Ingrese valor del tamaño: ");
35     scanf("%lu", &tam);
36     tipo = seleccionar_tipo();
37     crea_arreglo(&arreglo, tam, tipo);
38     do{
39         opt = menu();
40         o = 0;
41         if(opt == 3){
42             o = 1;
43         }
44         else if(opt == 0){
45             flag = false;
46             o = 1;
47         }
48         else if(opt == 1){
49             escribir(&arreglo, tam, tipo);
50         }
51         else if(opt == 2){
52             imprimir(&arreglo, tam, tipo);
53         }
54         else{
55             puts("opcion incorrecta");
56         }
57     }while(o != 1);
58     liberar_arreglo(&arreglo, tam, tipo);
59 }while(flag);
60 puts("Saliendo del programa...");
61 return 0;
62 }
63
64 tipo_t seleccionar_tipo(){
65     tipo_t t;
66     int b = 1;
67     do{
68         puts("Elige un tipo");
69         puts("1. int");
70         puts("2. char");
71         puts("3. float");
72         puts("4. Alumno");
73         printf("Ingresa numero de tipo: ");
74         scanf("%u", &t);
75         if(t == ENTERO){
76             return ENTERO;
77         }
78         else if(t == LETRA){
79             return LETRA;
```




```
80     }
81     else if(t == FLOTANTE){
82         return FLOTANTE;
83     }
84     else if(t == ALUMNO){
85         return ALUMNO;
86     }
87     else{
88         puts("Opcion no valida, vuelva a escoger");
89     }
90 }while(b != 0);
91 }
92
93 void crea_arreglo(void **arreglo, size_t tam, tipo t tipo){
94     if(tipo == ENTERO){
95         *arreglo = malloc(tam * sizeof(int));
96         assert(*arreglo != NULL);
97     }
98     if(tipo == LETRA){
99         *arreglo = malloc(tam * sizeof(char));
100        assert(*arreglo != NULL);
101    }
102    if(tipo == FLOTANTE){
103        *arreglo = malloc(tam * sizeof(float));
104        assert(*arreglo != NULL);
105    }
106    if(tipo == ALUMNO){
107        *arreglo = malloc(tam * sizeof(alumno_t));
108        assert(*arreglo != NULL);
109    }
110 }
111
112 int menu(){
113     int op;
114     puts("Que desea hacer?");
115     puts("0. Salir");
116     puts("1. Escribir datos en el arreglo");
117     puts("2. Imprimir el arreglo");
118     puts("3. Cambiar tipo de dato (debera volver a crear la matriz)");
119     printf("Ingrese el numero de la opción: ");
120     scanf("%d", &op);
121     return op;
122 }
123
124 void escribir(void **a, size_t tam, tipo t tipo){
125     if(tipo == ENTERO){
```



```
126     for(size_t i = 0; i < tam; i++){
127         scanf("%d", ((int *)(*a) + i));
128     }
129 }
130 if(tipo == LETRA){
131     char s;
132     for(size_t i = 0; i < tam; i++){
133         scanf("%c", &s);
134         scanf("%c", ((char *)(*a) + i));
135     }
136 }
137 if(tipo == FLOTANTE){
138     for(size_t i = 0; i < tam; i++){
139         scanf("%f", ((float *)(*a) + i));
140     }
141 }
142 if(tipo == ALUMNO){
143     char nombre[50], s;
144     int clave;
145     for(size_t i = 0; i < tam; i++){
146         printf("Ingreso clave: ");
147         scanf("%d", &((alumno_t *)(*a) + i)->clave);
148         scanf("%c", &s);
149         printf("Ingreso nombre: ");
150         scanf("%s", nombre);
151         ((alumno_t *)(*a) + i)->nombre = malloc(strlen(nombre) + 1);
152         assert(((alumno_t *)(*a) + i)->nombre != NULL);
153         strcpy(((alumno_t *)(*a) + i)->nombre, nombre);
154     }
155 }
156 }
157
158 void imprimir(void **a, size_t tam, tipo_t tipo){
159     if(tipo == ENTERO){
160         for(size_t i = 0; i < tam; i++){
161             printf("%d ", *((int *)(*a) + i));
162         }
163     }
164     if(tipo == LETRA){
165         for(size_t i = 0; i < tam; i++){
166             printf("%c ", *((char *)(*a) + i));
167         }
168     }
169     if(tipo == FLOTANTE){
170         for(size_t i = 0; i < tam; i++){
171             printf("%f ", *((float *)(*a) + i));
```



```
172     }
173 }
174 if(tipo == ALUMNO){
175     char nombre[50];
176     for(size_t i = 0; i < tam; i++){
177         printf("Nombre: %s  Clave: %d", ((alumno_t *)(*a) + i)->nombre,
178 ((alumno_t *)(*a) + i)->clave);
179         printf("\n");
180     }
181     printf("\n");
182 }
183
184 void liberar_arreglo(void **a, size_t tam, tipo_t tipo){
185     if(tipo != ALUMNO){
186         free(*a);
187     }
188     else{
189         for(size_t i = 0; i < tam; i++){
190             free(((alumno_t *)(*a) + i)->nombre);
191         }
192         free(*a);
193     }
194 }
```



Tarea #3: Arreglos Redimensionables

Grade School

If you need help running the tests or submitting your code, check out [HELP.md](#).

Instructions

Given students' names along with the grade that they are in, create a roster for the school.

In the end, you should be able to:

- Add a student's name to the roster for a grade
 - "Add Jim to grade 2."
 - "OK."
- Get a list of all students enrolled in a grade
 - "Which students are in grade 2?"
 - "We've only got Jim just now."
- Get a sorted list of all students in all grades. Grades should sort as 1, 2, 3, etc., and students within a grade should be sorted alphabetically by name.
 - "Who all is enrolled in school right now?"
 - "Let me think. We have Anna, Barb, and Charlie in grade 1, Alex, Peter, and Zoe in grade 2 and Jim in grade 5. So the answer is: Anna, Barb, Charlie, Alex, Peter, Zoe and Jim"

Note that all our students only have one name (It's a small town, what do you want?) and each student cannot be added more than once to a grade or the roster.

In fact, when a test attempts to add the same student more than once, your implementation should indicate that this is incorrect.



grade_school.h

```
#ifndef GRADE_SCHOOL_H
#define GRADE_SCHOOL_H

#include <stddef.h>
#include <stdint.h>
#include <stdbool.h>
```



```
typedef struct {
    uint8_t grade;
    char *name;
} student_t;

typedef struct {
    size_t count;
    size_t max_students;
    student_t *students;
} roster_t;

void init_roster(roster_t *roster);
void free_roster(roster_t *roster);
bool add_student(roster_t *roster, const char *name, uint8_t grade);
roster_t get_grade(const roster_t *roster, uint8_t desired_grade);

#endif
```



test_grade_school.c

```
#include "test-framework/unity.h"
#include "grade_school.h"

void setUp(void){
}

void tearDown(void){
}

static void check_roster_names(roster_t expected, roster_t actual){
    for (size_t i = 0; i < expected.count; ++i)
        TEST_ASSERT_EQUAL_STRING(expected.students[i].name,
                                   actual.students[i].name);
}

static void check_rosters(roster_t expected, roster_t actual){
    TEST_ASSERT_EQUAL_size_t_MESSAGE(expected.count, actual.count,
                                       "Incorrect number of students");
    TEST_ASSERT_EQUAL_size_t_MESSAGE(expected.max_students, actual.max_students,
                                       "Incorrect maximum space for students");
    check_roster_names(expected, actual);
}

static void test_roster_is_empty_when_no_student_added(void){
    TEST_IGNORE(); // delete this line to run test
    roster_t actual;
    init_roster(&actual);
    TEST_ASSERT_EQUAL(0, actual.count);
    TEST_ASSERT_EQUAL(0, actual.max_students);
    TEST_ASSERT_NULL(actual.students);
    free_roster(&actual);
}
```



```
static void test_add_student(void){
    TEST_IGNORE(); // delete this line to run test
    roster_t actual;
    init_roster(&actual);
    TEST_ASSERT_TRUE(add_student(&actual, "Aimee", 2));
    free_roster(&actual);
}

static void test_student_added_to_roster(void){
    TEST_IGNORE();
    student_t expected_students[] = { { 2, "Aimee" } };
    roster_t expected = { 1, 1, expected_students };
    roster_t actual;
    init_roster(&actual);
    add_student(&actual, "Aimee", 2);
    check_rosters(expected, actual);
    free_roster(&actual);
}

static void test_adding_multiple_students_in_same_grade_in_roster(void){
    TEST_IGNORE();
    roster_t actual;
    init_roster(&actual);
    TEST_ASSERT_TRUE(add_student(&actual, "Blair", 2));
    TEST_ASSERT_TRUE(add_student(&actual, "James", 2));
    TEST_ASSERT_TRUE(add_student(&actual, "Paul", 2));
    free_roster(&actual);
}

static void test_multiple_students_in_same_grade_are_added_to_roster(void){
    TEST_IGNORE();
    student_t expected_students[] = {
        { 2, "Blair" },
        { 2, "James" },
        { 2, "Paul" }
    };
    roster_t expected = { 3, 4, expected_students };
    roster_t actual;
    init_roster(&actual);
    add_student(&actual, "Blair", 2);
    add_student(&actual, "James", 2);
    add_student(&actual, "Paul", 2);
    check_rosters(expected, actual);
    free_roster(&actual);
}

static void test_cannot_add_student_to_same_grade_more_than_once(void){
    TEST_IGNORE();
    roster_t actual;
    init_roster(&actual);
    TEST_ASSERT_TRUE(add_student(&actual, "Blair", 2));
    TEST_ASSERT_TRUE(add_student(&actual, "James", 2));
    TEST_ASSERT_FALSE(add_student(&actual, "James", 2));
    TEST_ASSERT_TRUE(add_student(&actual, "Paul", 2));
    free_roster(&actual);
}
```



```
static void test_student_not_added_to_same_grade_in_roster_more_than_once(void){
    TEST_IGNORE();
    student_t expected_students[] = {
        { 2, "Blair" },
        { 2, "James" },
        { 2, "Paul" }
    };
    roster_t expected = { 3, 4, expected_students};
    roster_t actual;
    init_roster(&actual);
    add_student(&actual, "Blair", 2);
    add_student(&actual, "James", 2);
    add_student(&actual, "James", 2);
    add_student(&actual, "Paul", 2);
    check_rosters(expected, actual);
    free_roster(&actual);
}

static void test_adding_students_in_multiple_grades(void){
    TEST_IGNORE();
    roster_t actual;
    init_roster(&actual);
    TEST_ASSERT_TRUE(add_student(&actual, "Chelsea", 3));
    TEST_ASSERT_TRUE(add_student(&actual, "Logan", 7));
    free_roster(&actual);
}

static void test_students_in_multiple_grades_are_added_to_roster(void){
    TEST_IGNORE();
    student_t expected_students[] = {
        { 3, "Chelsea" },
        { 7, "Logan" }
    };
    roster_t expected = {2, 2, expected_students};
    roster_t actual;
    init_roster(&actual);
    add_student(&actual, "Chelsea", 3);
    add_student(&actual, "Logan", 7);
    check_rosters(expected, actual);
    free_roster(&actual);
}

static void test_cannot_add_same_student_to_multiple_grades_in_roster(void){
    TEST_IGNORE();
    roster_t actual;
    init_roster(&actual);
    TEST_ASSERT_TRUE(add_student(&actual, "Blair", 2));
    TEST_ASSERT_TRUE(add_student(&actual, "James", 2));
    TEST_ASSERT_FALSE(add_student(&actual, "James", 3));
    TEST_ASSERT_TRUE(add_student(&actual, "Paul", 3));
    free_roster(&actual);
}
```



```
static void test_student_not_added_to_multiple_grades_in_roster(void){
    TEST_IGNORE();
    student_t expected_students[] = {
        { 2, "Blair" },
        { 2, "James" },
        { 3, "Paul" }
    };
    roster_t expected = { 3, 4, expected_students};
    roster_t actual;
    init_roster(&actual);
    add_student(&actual, "Blair", 2);
    add_student(&actual, "James", 2);
    add_student(&actual, "James", 3);
    add_student(&actual, "Paul", 3);
    check_rosters(expected, actual);
    free_roster(&actual);
}

static void test_students_are_sorted_by_grades_in_roster(void){
    TEST_IGNORE();
    student_t expected_students[] = {
        { 1, "Anna" },
        { 2, "Peter" },
        { 3, "Jim" }
    };
    roster_t expected = { 3, 4, expected_students};
    roster_t actual;
    init_roster(&actual);
    add_student(&actual, "Jim", 3);
    add_student(&actual, "Peter", 2);
    add_student(&actual, "Anna", 1);
    check_rosters(expected, actual);
    free_roster(&actual);
}

static void test_students_are_sorted_by_name_in_roster(void){
    TEST_IGNORE();
    student_t expected_students[] = {
        { 2, "Alex" },
        { 2, "Peter" },
        { 2, "Zoe" }
    };
    roster_t expected = { 3, 4, expected_students};
    roster_t actual;
    init_roster(&actual);
    add_student(&actual, "Peter", 2);
    add_student(&actual, "Zoe", 2);
    add_student(&actual, "Alex", 2);
    check_rosters(expected, actual);
    free_roster(&actual);
}
```




```
static void test_students_are_sorted_by_grades_and_then_by_names_in_roster(void){
    TEST_IGNORE();
    student_t expected_students[] = {
        { 1, "Anna" },
        { 1, "Barb" },
        { 1, "Charlie" },
        { 2, "Alex" },
        { 2, "Peter" },
        { 2, "Zoe" },
        { 3, "Jim" }
    };
    roster_t expected = { 7, 8, expected_students};
    roster_t actual;
    init_roster(&actual);
    add_student(&actual, "Peter", 2);
    add_student(&actual, "Anna", 1);
    add_student(&actual, "Barb", 1);
    add_student(&actual, "Zoe", 2);
    add_student(&actual, "Alex", 2);
    add_student(&actual, "Jim", 3);
    add_student(&actual, "Charlie", 1);
    check_rosters(expected, actual);
    free_roster(&actual);
}

static void test_grade_empty_if_no_students_in_roster(void){
    TEST_IGNORE();
    uint8_t desired_grade = 1;
    roster_t roster;
    init_roster(&roster);
    roster_t actual = get_grade(&roster, desired_grade);
    TEST_ASSERT_EQUAL(0, actual.count);
    free_roster(&roster);
    free_roster(&actual);
}

static void test_grade_empty_if_no_students_in_grade(void){
    TEST_IGNORE();
    uint8_t desired_grade = 1;
    roster_t roster;
    init_roster(&roster);
    add_student(&roster, "Peter", 2);
    add_student(&roster, "Zoe", 2);
    add_student(&roster, "Alex", 2);
    add_student(&roster, "Jim", 3);
    roster_t actual = get_grade(&roster, desired_grade);
    TEST_ASSERT_EQUAL(0, actual.count);
    free_roster(&roster);
    free_roster(&actual);
}
```



```
static void test_student_not_added_to_same_grade_more_than_once(void){
    TEST_IGNORE();
    uint8_t desired_grade = 2;
    student_t expected_students[] = {
        { 2, "Blair" },
        { 2, "James" },
        { 2, "Paul" }
    };
    roster_t expected = { 3, 4, expected_students};
    roster_t roster;
    init_roster(&roster);
    add_student(&roster, "Blair", 2);
    add_student(&roster, "James", 2);
    add_student(&roster, "James", 2);
    add_student(&roster, "Paul", 2);
    roster_t actual = get_grade(&roster, desired_grade);
    check_rosters(expected, actual);
    free_roster(&roster);
    free_roster(&actual);
}

static void test_student_not_added_to_multiple_grades(void){
    TEST_IGNORE();
    uint8_t desired_grade = 2;
    student_t expected_students[] = {
        { 2, "Blair" },
        { 2, "James" }
    };
    roster_t expected = {2, 2, expected_students};
    roster_t roster;
    init_roster(&roster);
    add_student(&roster, "Blair", 2);
    add_student(&roster, "James", 2);
    add_student(&roster, "James", 3);
    add_student(&roster, "Paul", 3);
    roster_t actual = get_grade(&roster, desired_grade);
    check_rosters(expected, actual);
    free_roster(&roster);
    free_roster(&actual);
}

static void test_student_not_added_to_other_grade_for_multiple_grades(void){
    TEST_IGNORE();
    uint8_t desired_grade = 3;
    student_t expected_students[] = {
        { 3, "Paul" }
    };
    roster_t expected = { 1, 1, expected_students};
    roster_t roster;
    init_roster(&roster);
    add_student(&roster, "Blair", 2);
    add_student(&roster, "James", 2);
    add_student(&roster, "James", 3);
    add_student(&roster, "Paul", 3);
    roster_t actual = get_grade(&roster, desired_grade);
    check_rosters(expected, actual);
    free_roster(&roster);
    free_roster(&actual);
}
```



```
static void test_students_are_sorted_by_name_in_grade(void){
    TEST_IGNORE();
    uint8_t desired_grade = 5;
    student_t expected_students[] = {
        { 5, "Bradley" },
        { 5, "Franklin" }
    };
    roster_t expected = {2, 2, expected_students};
    roster_t roster;
    init_roster(&roster);
    add_student(&roster, "Franklin", 5);
    add_student(&roster, "Bradley", 5);
    add_student(&roster, "Jeff", 1);
    roster_t actual = get_grade(&roster, desired_grade);
    check_rosters(expected, actual);
    free_roster(&roster);
    free_roster(&actual);
}

int main(void){
    UnityBegin("test_grade_school.c");
    RUN_TEST(test_roster_is_empty_when_no_student_added);
    RUN_TEST(test_add_student);
    RUN_TEST(test_student_added_to_roster);
    RUN_TEST(test_adding_multiple_students_in_same_grade_in_roster);
    RUN_TEST(test_multiple_students_in_same_grade_are_added_to_roster);
    RUN_TEST(test_cannot_add_student_to_same_grade_more_than_once);
    RUN_TEST(test_student_not_added_to_same_grade_in_roster_more_than_once);
    RUN_TEST(test_adding_students_in_multiple_grades);
    RUN_TEST(test_students_in_multiple_grades_are_added_to_roster);
    RUN_TEST(test_cannot_add_same_student_to_multiple_grades_in_roster);
    RUN_TEST(test_student_not_added_to_multiple_grades_in_roster);
    RUN_TEST(test_students_are_sorted_by_grades_in_roster);
    RUN_TEST(test_students_are_sorted_by_name_in_roster);
    RUN_TEST(test_students_are_sorted_by_grades_and_then_by_names_in_roster);
    RUN_TEST(test_grade_empty_if_no_students_in_roster);
    RUN_TEST(test_grade_empty_if_no_students_in_grade);
    RUN_TEST(test_student_not_added_to_same_grade_more_than_once);
    RUN_TEST(test_student_not_added_to_multiple_grades);
    RUN_TEST(test_student_not_added_to_other_grade_for_multiple_grades);
    RUN_TEST(test_students_are_sorted_by_name_in_grade);
    return UnityEnd();
}
```



grade_school.c

```
1  #include "grade_school.h"
2  #include <stdlib.h>
3  #include <assert.h>
4  #include <string.h>
5  void sort_by_grade(roster_t *);
6  void sort_by_name(roster_t *);
7  void init_roster(roster_t *roster){
8      roster->count = 0;
9      roster->max_students = 0;
10     roster->students = NULL;
11 }
12
13 void free_roster(roster_t *roster){
14     for(size_t i = 0; i < roster->count; i++){
15         free(roster->students[i].name);
16     }
17     free(roster->students);
18 }
19
20 bool add_student(roster_t *roster, const char *name, uint8_t grade){
21     if(roster->count == roster->max_students){
22         if(roster->max_students == 0){
23             roster->max_students = 1;
24             student_t *r = (student_t *)realloc(roster->students, roster->max_students * sizeof(student_t));
25             assert(r != NULL);
26             roster->students = r;
27             roster->students[roster->count].grade = grade;
28             roster->students[roster->count].name = malloc(strlen(name) + 1);
29             assert(roster->students[roster->count].name != NULL);
30             strcpy(roster->students[roster->count].name, name);
31             roster->count++;
32             return true;
33         }
34         else{
35             for(size_t i = 0; i < roster->count; i++){
36                 if(strcmp(roster->students[i].name, name) == 0){
37                     return false;
38                 }
39             }
40             roster->max_students = roster->max_students * 2;
41             student_t *r = (student_t *)realloc(roster->students, roster->max_students * sizeof(student_t));
```



```
42     assert(r != NULL);
43     roster->students = r;
44     roster->students[roster->count].grade = grade;
45     roster->students[roster->count].name = malloc(strlen(name) + 1);
46     assert(roster->students[roster->count].name != NULL);
47     strcpy(roster->students[roster->count].name, name);
48     roster->count++;
49     sort_by_grade(roster);
50     sort_by_name(roster);
51     return true;
52 }
53 }
54 else{
55     for(size_t i = 0; i < roster->count; i++){
56         if(strcmp(roster->students[i].name, name) == 0){
57             return false;
58         }
59     }
60     roster->students[roster->count].grade = grade;
61     roster->students[roster->count].name = malloc(strlen(name) + 1);
62     assert(roster->students[roster->count].name != NULL);
63     strcpy(roster->students[roster->count].name, name);
64     roster->count++;
65     sort_by_grade(roster);
66     sort_by_name(roster);
67     return true;
68 }
69 }
70
71 void sort_by_grade(roster_t *roster){
72     student_t tmp;
73     for(size_t i = 0; i < roster->count; i++){
74         for(size_t j = i + 1; j < roster->count; j++){
75             if(roster->students[i].grade > roster->students[j].grade){
76                 tmp = roster->students[i];
77                 roster->students[i] = roster->students[j];
78                 roster->students[j] = tmp;
79                 i = 0;
80             }
81         }
82     }
83 }
84
85 void sort_by_name(roster_t *roster){
86     student_t tmp;
87     for(size_t i = 0; i < roster->count; i++){
```



```
88     for(size_t j = i + 1; j < roster->count; j++){
89         if(roster->students[i].grade == roster->students[j].grade){
90             if(strcmp(roster->students[i].name, roster-
>students[j].name) > 0){
91                 tmp = roster->students[j];
92                 roster->students[j] = roster->students[i];
93                 roster->students[i] = tmp;
94             }
95         }
96     }
97 }
98 }
99
100 roster_t get_grade(const roster_t *roster, uint8_t desired_grade){
101     roster_t answer;
102     init_roster(&answer);
103     for(size_t i = 0; i < roster->count; i++){
104         if(roster->students[i].grade == desired_grade){
105             add_student(&answer, roster->students[i].name, desired_grade);
106         }
107     }
108     return answer;
109 }
```



1. Explica tu algoritmo para agregar un estudiante a una lista. ¿Cómo crece tu lista dinámicamente? ¿Cómo determinas de que no puede haber dos estudiantes en la lista que tengan el mismo nombre? ¿Cómo determinas que un mismo estudiante no puede estar en dos grados distintos?

```

20 bool add_student(roster_t *roster, const char *name, uint8_t grade){
21     if(roster->count == roster->max_students){
22         if(roster->max_students == 0){
23             roster->max_students = 1;
24             student_t *r = (student_t *)realloc(roster->students, roster->max_students * sizeof(student_t));
25             assert(r != NULL);
26             roster->students = r;
27             roster->students[roster->count].grade = grade;
28             roster->students[roster->count].name = malloc(strlen(name) + 1);
29             assert(roster->students[roster->count].name != NULL);
30             strcpy(roster->students[roster->count].name, name);
31             roster->count++;
32             return true;
33         }
34         else{
35             for(size_t i = 0; i < roster->count; i++){
36                 if(strcmp(roster->students[i].name, name) == 0){
37                     return false;
38                 }
39             }
40             roster->max_students = roster->max_students * 2;
41             student_t *r = (student_t *)realloc(roster->students, roster->max_students * sizeof(student_t));
42             assert(r != NULL);
43             roster->students = r;
44             roster->students[roster->count].grade = grade;
45             roster->students[roster->count].name = malloc(strlen(name) + 1);
46             assert(roster->students[roster->count].name != NULL);
47             strcpy(roster->students[roster->count].name, name);
48             roster->count++;
49             sort_by_grade(roster);
50             sort_by_name(roster);
51             return true;
52         }
53     }
54     else{
55         for(size_t i = 0; i < roster->count; i++){
56             if(strcmp(roster->students[i].name, name) == 0){
57                 return false;
58             }
59         }
60         roster->students[roster->count].grade = grade;
61         roster->students[roster->count].name = malloc(strlen(name) + 1);
62         assert(roster->students[roster->count].name != NULL);
63         strcpy(roster->students[roster->count].name, name);
64         roster->count++;
65         sort_by_grade(roster);
66         sort_by_name(roster);
67         return true;
68     }
69 }

```

Primero reviso si el último elemento de la lista ya alcanza al tamaño máximo, de ser así, hay dos opciones, la primera, que este vacía, en ese caso el tamaño máximo cambia de cero a uno, en caso contrario, el tamaño máximo aumenta al doble, y se utiliza `realloc()` para cambiar el tamaño. En caso de que no, simplemente agrego el elemento. Para revisar que no haya dos estudiantes repetidos, solo necesito hacer una búsqueda de todos los alumnos, así no importa si son del mismo grupo o no.



2. Explica tu algoritmo para obtener la lista de estudiantes de un grado específico. ¿La lista que se obtiene como resultado de este algoritmo es dinámica? ¿Cómo determinas su tamaño?

```
100 roster_t get_grade(const roster_t *roster, uint8_t desired_grade){
101     roster_t answer;
102     init_roster(&answer);
103     for(size_t i = 0; i < roster->count; i++){
104         if(roster->students[i].grade == desired_grade){
105             add_student(&answer, roster->students[i].name, desired_grade);
106         }
107     }
108     return answer;
109 }
```

Lo que hago es crear otra lista llamada `answer`, la inicializo con `init_roster()`, y luego hago un ciclo `for` que recorre todos los elementos de la lista, cada vez que encuentra uno con el mismo grado, lo agrega a la lista `answer` con la función `add_student()`, al utilizar esa función, la lista crece dinámicamente, por lo que no necesito determinar su tamaño.



3. ¿Qué algoritmo de ordenamiento utilizaste para esta tarea? ¿Cómo aplicaste el criterio de ordenar primero por grado y luego por nombre?

```
71 void sort_by_grade(roster_t *roster){
72     student_t tmp;
73     for(size_t i = 0; i < roster->count; i++){
74         for(size_t j = i + 1; j < roster->count; j++){
75             if(roster->students[i].grade > roster->students[j].grade){
76                 tmp = roster->students[i];
77                 roster->students[i] = roster->students[j];
78                 roster->students[j] = tmp;
79                 i = 0;
80             }
81         }
82     }
83 }
```

Utilicé el método de ordenamiento burbuja() en el que va de forma iterativa uno por uno ordenándolos por grado, utilicé este método porque los `tests` no contenían muchos alumnos, por lo cual no hay mucha diferencia entre utilizar este o algún algoritmo más rápido como `quicksort()`, el método burbuja aprendido en clase suele utilizar otra función llamada `swap()`, pero decidí implementarlo ahí mismo.

Para el ordenamiento por nombre primero reviso si los elementos que trato de ordenar (el actual y el siguiente) son del mismo grado, ya que de lo contrario no tiene sentido, después reviso con la función `strcmp()` si es mayor a 0, en cuyo caso los cambio de lugar.



```

85 void sort_by_name(roster_t *roster){
86     student_t tmp;
87     for(size_t i = 0; i < roster->count; i++){
88         for(size_t j = i + 1; j < roster->count; j++){
89             if(roster->students[i].grade == roster->students[j].grade){
90                 if(strcmp(roster->students[i].name, roster->students[j].name) > 0){
91                     tmp = roster->students[j];
92                     roster->students[j] = roster->students[i];
93                     roster->students[i] = tmp;
94                 }
95             }
96         }
97     }
98 }

```



4. Imagina por un momento que se te pide agregar una nueva funcionalidad para dar de baja estudiantes. Explica detalladamente los cambios que consideres son necesarios para poder eliminar estudiantes de la lista dinámica.

```

71 void remove_student(roster_t *roster, const char *name){
72     for(size_t i = 0; i < roster->count; i++){
73         if(strcmp(roster->students[i].name, name) == 0){
74             for(size_t j = i; j < roster->count - 1; j++){
75                 student_t tmp = roster->students[j];
76                 roster->students[j] = roster->students[j + 1];
77                 free(tmp.name);
78             }
79             roster->count--;
80             return;
81         }
82     }
83 }

```

Para eliminar un alumno, podemos hacer una función `remove_student()` a la que le pasemos como parámetros la lista y el nombre del alumno que queremos eliminar, el nombre puede ser `const`, ya que, no solo es de lectura. Una vez entro de la función inicio una ciclo `for` para recorrer todos los elementos (ya que no hay alumnos repetidos, será el primero que encontremos), con la función `strcmp()` revisamos si ya lo encontramos, una vez que lo encontramos, primero lo guardamos en una estructura temporal, luego hacemos que se recorran todos de lugar, y hacemos el `free()` del nombre del temporal para no perderlo. También es importante agregar el prototipo de la función al archivo `.h` porque si la agregamos al archivo `.c` nos aparecerá un `warning` por ser llamado desde ahí.



5. ¿Cuál es el nombre de la prueba (test) que te resultó más difícil pasar? ¿Por qué consideras que fue difícil? Al finalizar la tarea, ¿cuántas pruebas tuviste que ignorar? ¿Cuáles fueron? ¿Qué errores obtuviste?

Ninguna, todas fueron sencillas de pasar, todas las pruebas fueron pasadas.



6. Inventa una nueva prueba y agrégala al conjunto de pruebas del proyecto. Agrega el código fuente y explicación de tu prueba en el reporte.

```
static void test_delete_student(void){
    //TEST_IGNORE();
    uint8_t desired_grade = 5;
    student_t expected_students[] = {
        { 5, "Andres" },
        { 5, "Franklin" },
        { 5, "Hugo" }
    };
    roster_t expected = {3, 4, expected_students};
    roster_t roster;
    init_roster(&roster);
    add_student(&roster, "Franklin", 5);
    add_student(&roster, "Bradley", 5);
    add_student(&roster, "Jeff", 1);
    remove_student(&roster, "Bradley");
    add_student(&roster, "Hugo", 5);
    add_student(&roster, "Andres", 5);
    roster_t actual = get_grade(&roster, desired_grade);
    check_rosters(expected, actual);
    free_roster(&roster);
    free_roster(&actual);
}
```

En este test decidí incorporar mi función para eliminar, agrego varios estudiantes, elimino a uno, y después sigo agregando. El test paso la prueba utilizando make memcheck.



7. ¿Aplicaste la refactorización en tu código? Explica de qué manera lo hiciste

Sí, me di cuenta que mi función para agregar alumnos era muy larga y repetitiva, así que la simplifique.

```
22 bool add_student(roster_t *roster, const char *name, uint8_t grade){
23     for(size_t i = 0; i < roster->count; i++){
24         if(strcmp(roster->students[i].name, name) == 0){
25             return false;
26         }
27     }
28     if(roster->count == roster->max_students){
29         if(roster->max_students == 0){
30             roster->max_students = 1;
31         }
32         else{
33             roster->max_students = roster->max_students * 2;
34         }
35     }
36 }
```



```
35     student_t *r = NULL;
36     r = (student_t *)realloc(roster->students, roster->max_students *
37 sizeof(student_t));
38     assert(r != NULL);
39     roster->students = r;
40 }
41 roster->students[roster->count].grade = grade;
42 roster->students[roster->count].name = malloc(strlen(name) + 1);
43 assert(roster->students[roster->count].name != NULL);
44 strcpy(roster->students[roster->count].name, name);
45 roster->count++;
46 sort_by_grade(roster);
47 sort_by_name(roster);
48 return true;
49 }
```



8. ¿Qué hiciste particularmente bien en esta tarea?

Aplicar los conocimientos adquiridos en clase de arreglos dinámicos utilizando `realloc()`, así como los conocimientos adquiridos el semestre pasado de ordenamiento.



9. ¿Qué pudiste haber hecho mejor en esta tarea?

Tal vez hay una manera más fácil de simplificar la función para agregar estudiantes.



10. ¿Qué nuevos conocimientos y experiencias adquiriste con esta tarea?

El utilizar los métodos de ordenamiento en punteros, aunque siento que más que aprender, esta tarea fue para poder terminar de entender el tema de arreglos dinámicos de manera practica.

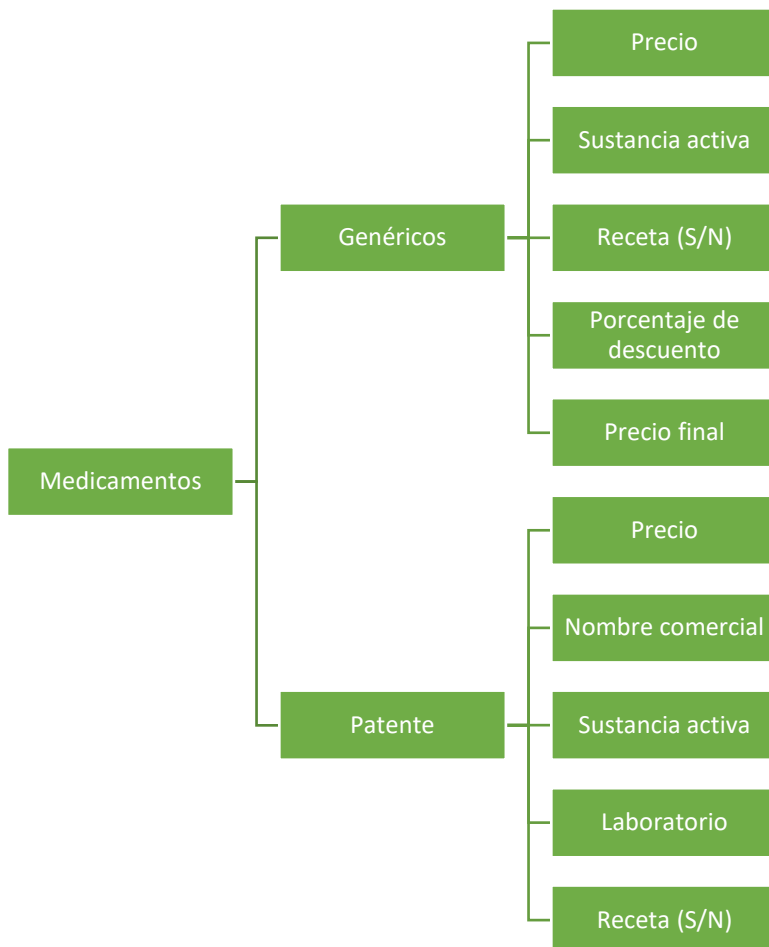


Trabajo en clase: "Farmacia con medicamentos"

Martes 13 Septiembre 2022

En una farmacia se venden 2 tipos de medicamentos: genéricos y de patente, Los datos de los tipos genéricos son: el precio, sustancia activa, si requiere receta médica ("s" o "n"), porcentaje de descuento (si no hay se asigna un cero) y precio final. Los datos para los medicamentos de patente son: precio, nombre comercial (por ejemplo "aspirina"), sustancia activa (por ejemplo "ácido acetilsalicílico"), laboratorio (por ejemplo "Bayer") y si requiere receta médica ("s" o "n").

a. Diseñe las estructuras de datos necesarias para poder almacenar la información de varios medicamentos en un arreglo dinámico de estructuras, en el cual se puedan almacenar datos tanto de medicamentos genéricos como de medicamentos de patente.



```

typedef enum{
    GENERICO,
    PATENTE
}tipo_medimento_t;

typedef struct{
    float precio;
    char *sustancia_activa;
    char requiere_receta;
    void *info_adicional;
    tipo_medimento_t tipo;
}medimento_t;

typedef struct{
    float porcentaje_descuento;
    float precio_final;
}generico_t;

typedef struct{
    char *nombre_comercial;
    char *laboratorio;
}patente_t;
  
```

b. Escriba una función para reservar memoria para el arreglo de medicamentos. La función recibe dos parámetros: el tamaño del arreglo y un apuntador por referencia donde se asignará la memoria dinámica para el arreglo.

```

medimento_t *crea_arreglo(size_t tam){
    medicamento_t *arreglo = NULL;
    arreglo = (medimento_t *)malloc(tam * sizeof(medimento_t));
    assert(arreglo != NULL);
}
  
```



c. Escribir una función para capturar la información medicamentos genéricos y/o de patente. La función recibe dos parámetros: un apuntador al arreglo de medicamentos y un entero con el tamaño del arreglo. Utilice notación/aritmética de apuntadores para acceder a los elementos del arreglo.

```

1 void captura_medicamentos(medicamento_t *arreglo, size_t tam){
2     char s[40];
3     for(size_t i = 0; i < tam; i++){
4         scanf("%f", &arreglo[i].precio);
5         scanf("%u", &arreglo[i].tipo);
6         scanf("\n%c", &arreglo[i].requiere_receta);
7         scanf("\n%[^\n]", s);
8         arreglo[i].sustancia_activa = malloc(strlen(s) + 1);
9         assert(arreglo[i].sustancia_activa != NULL);
10        strcpy(arreglo[i].sustancia_activa, s);
11        if(arreglo[i].tipo == GENERICO){
12            arreglo[i].info_adicional = malloc(sizeof(generico_t));
13            scanf("%f", &((generico_t *) (arreglo[i].info_adicional))-
14                >porcentaje_descuento);
15            float descuento = (arreglo[i].precio) * ((generico_t
16                *) (arreglo[i].info_adicional))->porcentaje_descuento;
17            ((generico_t *) (arreglo[i].info_adicional))->precio_final =
18                arreglo[i].precio - descuento;
19        }
20        else{
21            char nc[50];
22            char lab[50];
23            arreglo[i].info_adicional = malloc(sizeof(patente_t));
24            scanf("\n%[^\n]", nc);
25            scanf("\n%[^\n]", lab);
26            ((patente_t *) (arreglo[i].info_adicional))->laboratorio =
27                malloc(strlen(lab) + 1);
28            assert(((patente_t *) (arreglo[i].info_adicional))->laboratorio
29                != NULL);
30            strcpy(((patente_t *) (arreglo[i].info_adicional))->laboratorio,
31                lab);
32            ((patente_t *) (arreglo[i].info_adicional))->nombre_comercial =
33                malloc(strlen(nc) + 1);
34            assert(((patente_t *) (arreglo[i].info_adicional))->nombre_comercial
35                != NULL);
36            strcpy(((patente_t *) (arreglo[i].info_adicional))->nombre_comercial, nc);
37        }
38    }
39 }

```



Primer Examen Parcial

Jueves 15 Septiembre 2022

Problema 1 (25 puntos)

Escribir una función para obtener los elementos de la diagonal principal de una matriz dinámica. La función recibe como parámetros el apuntador a una matriz dinámica de $N \times N$ números enteros, previamente creada y rellena con datos, y regresa un arreglo dinámico con el contenido de la diagonal principal. **Utilice aritmética de apuntadores.**

Por ejemplo, si la matriz dinámica contiene la siguiente información:

1	2	3
4	5	6
7	8	9

Entonces regresa un arreglo dinámico que contiene:

1	5	9
---	---	---

Solución

Asumiendo que también nos dan N como un parámetro, por ejemplo, en el ejemplo sería $N = 3$

```
int *diagonal_Principal(int **matriz, int N){
    int *diagonal = NULL;
    crea_arreglo(&diagonal, N);
    int j = 0;
    for(int i = 0; i < N; i++){
        *(diagonal + j) = (*(matriz + i) + j);
        j = i + 1;
    }
    return diagonal;
}

void crea_arreglo(int **arreglo, int tam){
    *arreglo = (int *)malloc(tam * sizeof(int));
    assert(*arreglo != NULL);
}
```

Problema 2

Se requiere implementar un prototipo de una pequeña parte de una aplicación de red social, la cual mostrará publicaciones que aparecerán en pantalla cuando el usuario abra la página principal. Existen dos tipos de publicaciones en la red social: publicaciones de texto (mensajes) y publicaciones fotográficas. Las publicaciones de texto contienen el nombre del autor y el texto del mensaje. Las publicaciones fotográficas contienen el nombre del autor, el número de personas a las que les ha gustado la publicación y el tamaño del archivo de la imagen que será visualizada.

a. (10 puntos) Diseñe y escriba el código de las estructuras requeridas para almacenar las publicaciones de la red social en una estructura de datos dinámica.



```
#define MAX_TEXTO 30

typedef enum{
    MENSAJE,
    FOTO
}tipo_post_t;

typedef struct{
    char username[MAX_TEXTO];
    tipo_post_t tipo;
    void *info;
}post_t;

typedef struct{
    char mensaje[MAX_TEXTO];
}mensaje_t;

typedef struct{
    int tam_archivo;
    int likes;
}foto_t;
```

b. (10 puntos) Completar la siguiente función para inicializar la estructura dinámica de publicaciones a un tamaño determinado. **Conteste únicamente sobre la hoja del examen.**

```
void init_posts(post_t **arreglo, int tam){
    *arreglo = (post_t *)malloc(tam * sizeof(post_t));
    assert(*arreglo != NULL);
}
```

c. (40 puntos) Completar la siguiente función leer la información dentro de la estructura dinámica de publicaciones. **Conteste únicamente sobre la hoja del examen y use aritmética de apuntadores.**

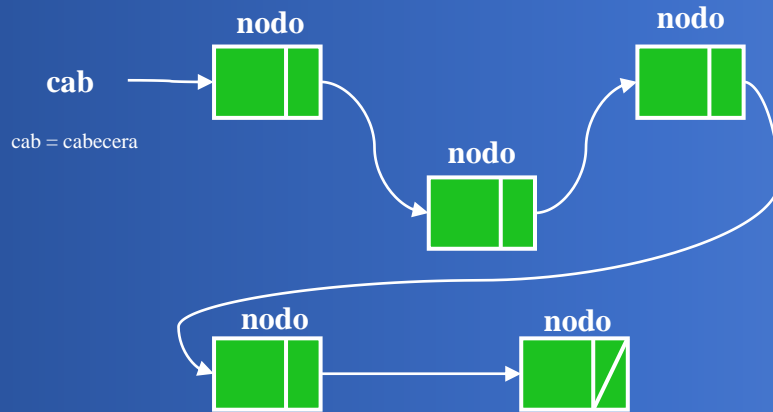
```
void read_posts(post_t *posts, int tam){
    for(post_t *p = posts; p < posts + tam; p++){
        printf("Username: ");
        scanf("%s", p->username);
        printf("Post type (0 = text, 1 = photo): ");
        scanf("%u", &p->tipo);
        if(p->tipo == MENSAJE){
            p->info = malloc(sizeof(mensaje_t));
            printf("Message text: ");
            scanf("\n%[^\n]", ((mensaje_t *)p->info)->mensaje);
        } else if(p->tipo == FOTO){
            p->info = malloc(sizeof(foto_t));
            printf("File size: ");
            scanf("%d", &((foto_t *)p->info)->tam_archivo);
            printf("Likes count: ");
            scanf("%d", &((foto_t *)p->info)->likes);
        }
    }
}
```



d. (15 puntos) Completar la siguiente función para obtener la cantidad de “me gusta” de la publicación más popular. Si ninguna publicación tiene “me gusta” regresa un valor especial -1. **Conteste únicamente sobre la hoja del examen y use aritmética de apuntadores.**

```
int most_popular_count(post_t *posts, int n){
    int result = -1;
    for(post_t *p = posts; p < posts + n; p++){
        if(p->tipo == FOTO){
            int likes = ((foto_t *)p->info)->likes;
            if(likes > 0 && likes > result){
                result = likes;
            }
        }
    }
    return result;
}
```


2. LISTAS ENLAZADAS



2.1 Listas simples

Objetivo: Ser capaz de diseñar diversos tipos de listas enlazadas y programar las principales operaciones para su manipulación.

En una lista podemos guardar cualquier cosa, para este ejemplo de lista simple, haremos una lista de números enteros, cada casilla es un nodo, y para cada nodo hacemos una estructura con los datos que almacenara cada nodo.

2.1.1 Definir la estructura del nodo

Struct nodo

info

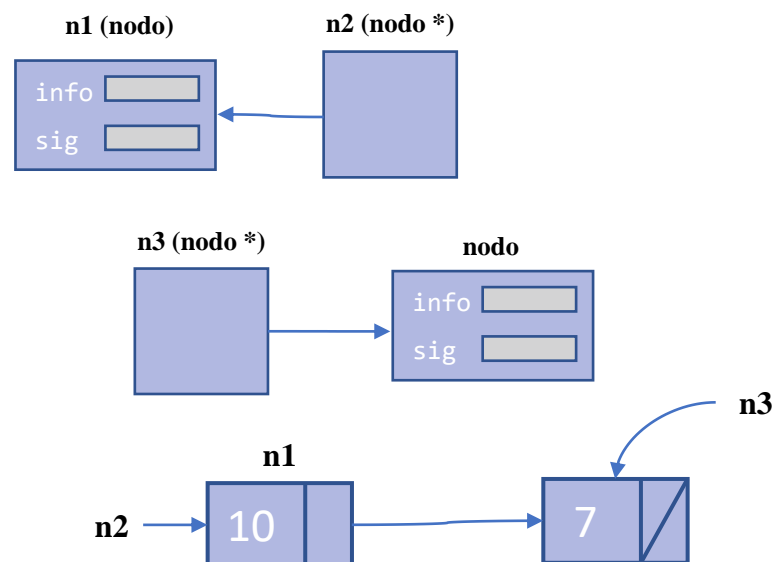
sig

```
1 struct nodo{
2     int info;
3     struct nodo *sig;
4 };
5 typedef struct nodo nodo_t;
```

```
1 int main(){
2     nodo_t n1;
3     nodo_t *n2, *n3;
4     n1.info = 5;
5     n1.sig = NULL;
6     n2 = &n1;
7     n2->info = 10;
8     printf("%d", n1.info);
9     n3 = (nodo_t *)malloc(sizeof(nodo_t));
10    assert(n3 != NULL);
11    n3->info = 7;
12    n3->sig = NULL;
13    n2->sig = n3;
14    return 0;
15 }
```

Output

10





2.1.2 Funciones de listas simples

Función para crear un nodo

```

1  nodo_t *crea_nodo(){
2      nodo_t *nodo = NULL;
3      nodo = (nodo_t*)malloc(sizeof(nodo_t));
4      assert(nodo != NULL);
5      nodo->info = 0;
6      nodo->sig = NULL;
7      return nodo;
8  }

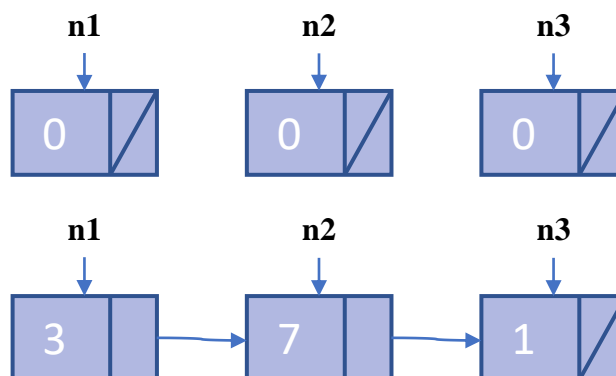
```

Ejemplo Burdo

```

1  int main(){
2      nodo_t *n1 = crea_nodo();
3      nodo_t *n2 = crea_nodo();
4      nodo_t *n3 = crea_nodo();
5      n1->info = 3;
6      n2->info = 7;
7      n3->info = 1;
8      n1->sig = n2;
9      n2->sig = n3;
10     printf("%d ", n1->info);           //output: 3
11     printf("%d ", n1->sig->info);       //output: 3
12     printf("%d", n1->sig->sig->info);    //output: 3
13     free(n1);
14     free(n2);
15     free(n3);
16     return 0;
17 }

```





```
1  int main(){
2      nodo t *lista = NULL;
3      int opcion = 0, num = 0;
4      do{
5          opcion = selecciona_opcion();
6          switch(opcion){
7              case 0: puts("Saliendo del programa...");
8                  break;
9              case 1: printf("Valor del numero: ");
10                 scanf("%d", &num);
11                 insertar_inicio(&lista, num);
12                 break;
13              case 2: printf("Valor del numero: ");
14                 scanf("%d", &num);
15                 insertar_final(&lista, num);
16                 break;
17              case 3: eliminar_inicio(&lista);
18                 break;
19              case 4: eliminar_final(&lista);
20                 break;
21              case 5: imprimir_lista(lista);
22                 break;
23              case 6: printf("Valor del numero: ");
24                 scanf("%d", &num);
25                 elimina_nodo(&lista, num);
26                 break;
27              case 7: liberar_lista(&lista);
28                 break;
29              case 8: printf("Valor del numero: ");
30                 scanf("%d", &num);
31                 printf("%p\n", buscar_dato(lista, num));
32                 break;
33              case 9: printf("%d\n", tam_lista(lista));
34                 break;
35              default: puts("Opcion no valida");
36                      break;
37          }
38      }while(opcion != 0);
39      liberar_lista(&lista);
40      return 0;
41 }
```



Función para seleccionar una opción

```

1  int selecciona_opcion(){
2      puts("Selecciona una opcion");
3      puts("0. Salir");
4      puts("1. Insertar inicio");
5      puts("2. Insertar al final");
6      puts("3. Eliminar al inicio");
7      puts("4. Eliminar al final");
8      puts("5. Imprimir toda la lista");
9      puts("6. Eliminar nodo arbitrario");
10     puts("7. Eliminar/Liberar toda la lista");
11     puts("8. Buscar dato");
12     puts("9. Imprimir tamaño de la lista");
13     int opcion;
14     scanf("%d", &opcion);
15     return opcion;
16 }

```

puts() funciona como printf, pero es para imprimir en pantalla puro texto, y ya imprime el salto de línea sin necesidad de ponerlo.

Función para insertar al inicio de la lista

```

1  void insertar_inicio(nodo_t **cab, int dato){
2      nodo_t *nodo = crea_nodo();
3      nodo->info = dato;
4      nodo->sig = *cab;
5      *cab = nodo;
6  }

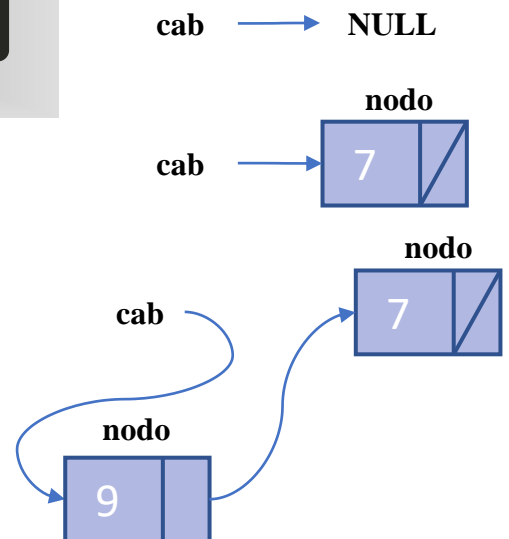
```

Ejemplo:

```

1  int main(){
2      nodo_t *lista = NULL;
3      insertar_inicio(&lista, 7);
4      insertar_inicio(&lista, 9);
5      return 0;
6  }

```





Función para imprimir la lista (iterativo)

```
1 void imprimir_lista(const nodo_t *cab){
2     const nodo_t *it = cab;
3     while(it != NULL){
4         printf("%d ", it->info);
5         it = it->sig;
6     }
7 }
```

Función para imprimir la lista (recursivo)

```
1 void imprimir_lista(const nodo_t *cab){
2     if(cab == NULL){
3         return;
4     }
5     printf("%d ", cab->info);
6     imprimir_lista(cab->sig); //Si lo intercambiamos, se imprime al revés.
7 }
```

Función para buscar un dato (iterativo)

```
1 nodo_t *buscar_dato(const nodo_t *cab, int dato){
2     for(const nodo_t *it = cab; it != NULL; it = it->sig){
3         if(it->info == dato){
4             return (nodo_t *)it;
5         }
6     }
7     return NULL;
8 }
```

Función para liberar lista (iterativo)

```
1 void liberar_lista(nodo_t **cab){
2     nodo_t *it = *cab;
3     while(it != NULL){
4         nodo_t *tmp = it;
5         it = it->sig;
6         free(tmp);
7     }
8     *cab = NULL;
9 }
```



Función para liberar lista (recursivo)

```
1 void liberar_lista(nodo_t **cab){
2     if(*cab == NULL){
3         return;
4     }
5     else{
6         liberar_lista(&((*cab)->sig));
7         free(*cab);
8     }
9     *cab = NULL;
10 }
```

Función para insertar al final de la lista (iterativo)

Jueves 22 Septiembre 2022

```
1 void insertar_final(nodo_t **cab, int dato){
2     nodo_t *nodo = crea_nodo();
3     nodo->info = dato;
4     if(*cab == NULL){
5         *cab = nodo;
6     }
7     else{
8         nodo_t *ultimo = *cab;
9         while(ultimo->sig != NULL){
10             ultimo = ultimo->sig;
11         }
12         ultimo->sig = nodo;
13     }
14 }
```

Función para insertar al final de la lista (recursivo)

```
1 void insertar_final(nodo_t **cab, int dato){
2     nodo_t *nodo = crea_nodo();
3     nodo->info = dato;
4     if(*cab == NULL){
5         *cab = nodo;
6         return;
7     }
8     else{
9         insertar_final(&((*cab)->sig), dato);
10     }
11 }
```



Función para eliminar al inicio de la lista

```

1 void eliminar_inicio(nodo_t **cab){
2     nodo_t *tmp;
3     if(*cab != NULL){
4         tmp = *cab;
5         *cab = (*cab)->sig;
6         free(tmp);
7     }
8 }

```



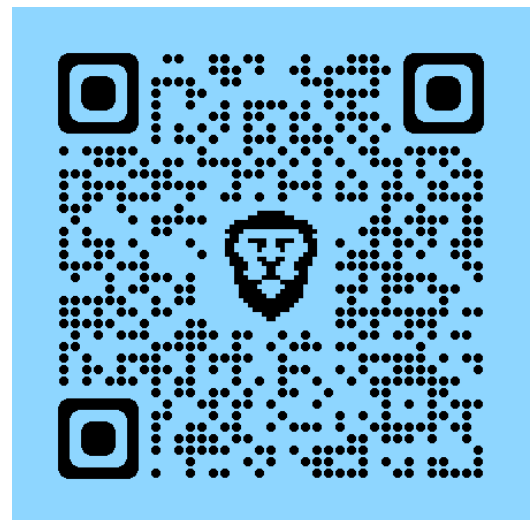
Ejercicio de OmegaUp: 14867. Listas enlazadas

Viernes 23 Septiembre 2022

Descripción

Dada una lista de enteros A inicialmente vacía, imprimir el contenido de A después de realizar N de las siguientes operaciones:

Operación	Explicación
push_front v	Agrega v al inicio de A
pop_front	Elimina el primer elemento de A
push_back v	Agrega v al final de A
pop_back	Elimina el último elemento de A
erase v	Elimina el primer elemento de A con valor igual a v
Clear	Elimina todos los elementos de A



Entrada

La primer línea contiene un entero N ($1 \leq N \leq 100$), el número de operaciones a realizar. Las siguientes N líneas contienen las operaciones a realizar.

Salida

Una sola línea con el contenido final de A , cada elemento separado por un espacio.

Ejemplo 1

Entrada	Salida
<pre> 6 push_back 3 push_front 7 clear push_front 5 erase 2 push_front 9 </pre>	<pre> 9 5 </pre>

Ejemplo 2

Entrada	Salida
<pre> 9 push_front 1 push_back 2 push_back 5 push_back 1 push_front 2 erase 1 push_front 6 push_back 3 erase 2 </pre>	<pre> 6 2 5 1 3 </pre>



```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <assert.h>
4  #include <string.h>
5
6  struct nodo{
7      int info;
8      struct nodo *sig;
9  };
10 typedef struct nodo nodo_t;
11
12 void insertar_inicio(nodo_t **, int);
13 void insertar_final(nodo_t **, int);
14 void imprimir_lista(const nodo_t *);
15 void eliminar_inicio(nodo_t **);
16 void eliminar_final(nodo_t **);
17 void elimina_nodo(nodo_t **, int);
18 void liberar_lista(nodo_t **);
19 nodo_t *crea_nodo();
20
21 int main(){
22     nodo_t *lista = NULL;
23     int n, x;
24     char comando[12];
25     scanf("%d", &n);
26     for(int i = 0; i < n; i++){
27         scanf("\n%s", comando);
28         if(strcmp(comando, "push_front") == 0){
29             scanf("%d", &x);
30             insertar_inicio(&lista, x);
31         }
32         else if(strcmp(comando, "pop_front") == 0){
33             eliminar_inicio(&lista);
34         }
35         else if(strcmp(comando, "push_back") == 0){
36             scanf("%d", &x);
37             insertar_final(&lista, x);
38         }
39         else if(strcmp(comando, "pop_back") == 0){
40             eliminar_final(&lista);
41         }
42         else if(strcmp(comando, "erase") == 0){
43             scanf("%d", &x);
44             elimina_nodo(&lista, x);
45         }
46         else{
```




```
47         liberar_lista(&lista);
48     }
49 }
50 imprimir_lista(lista);
51 liberar_lista(&lista);
52 return 0;
53 }
54
55 nodo_t *crea_nodo(){
56     nodo_t *nodo = NULL;
57     nodo = (nodo_t*)malloc(sizeof(nodo_t));
58     assert(nodo != NULL);
59     nodo->info = 0;
60     nodo->sig = NULL;
61     return nodo;
62 }
63
64 void insertar_inicio(nodo_t **cab, int dato){
65     nodo_t *nodo = crea_nodo();
66     nodo->info = dato;
67     nodo->sig = *cab;
68     *cab = nodo;
69 }
70
71 void insertar_final(nodo_t **cab, int dato){
72     nodo_t *nodo = crea_nodo();
73     nodo->info = dato;
74     if(*cab == NULL){
75         *cab = nodo;
76     }
77     else{
78         nodo_t *ultimo = *cab;
79         while(ultimo->sig != NULL){
80             ultimo = ultimo->sig;
81         }
82         ultimo->sig = nodo;
83     }
84 }
85
86 void imprimir_lista(const nodo_t *cab){
87     const nodo_t *it = cab;
88     while(it != NULL){
89         printf("%d ", it->info);
90         it = it->sig;
91     }
92     printf("\n");
```



```
93 }
94
95 void eliminar_inicio(nodo_t **cab){
96     nodo_t *tmp;
97     if(*cab != NULL){
98         tmp = *cab;
99         *cab = (*cab)->sig;
100         free(tmp);
101     }
102 }
103
104 void liberar_lista(nodo_t **cab){
105     nodo_t *it = *cab;
106     while(it != NULL){
107         nodo_t *tmp = it;
108         it = it->sig;
109         free(tmp);
110     }
111     *cab = NULL;
112 }
113
114 void eliminar_final(nodo_t **cab){
115     if(*cab == NULL){
116         return;
117     }
118     else if((*cab)->sig == NULL){
119         nodo_t *tmp = *cab;
120         free(tmp);
121         *cab = NULL;
122     }
123     else{
124         nodo_t *ultimo = *cab;
125         nodo_t *penultimo = NULL;
126         while(ultimo->sig != NULL){
127             penultimo = ultimo;
128             ultimo = ultimo->sig;
129         }
130         free(ultimo);
131         penultimo->sig = NULL;
132     }
133 }
134
135 void elimina_nodo(nodo_t **cab, int x){
136     if(*cab == NULL){
137         return;
138     }
```



```
139     else if((*cab)->info == x){
140         nodo_t *tmp = *cab;
141         *cab = (*cab)->sig;
142         free(tmp);
143         return;
144     }
145     else{
146         nodo_t *prev = *cab, *it = (*cab)->sig;
147         while(it != NULL && it->info != x){
148             prev = it;
149             it = it->sig;
150         }
151         if(it != NULL){
152             prev->sig = it->sig;
153             free(it);
154         }
155     }
156 }
```

Función para eliminar al final de la lista (iterativo)

Lunes 26 Septiembre 2022

```
1 void eliminar_final(nodo_t **cab){
2     if(*cab == NULL){
3         return;
4     }
5     else if((*cab)->sig == NULL){
6         nodo_t *tmp = *cab;
7         free(tmp);
8         *cab = NULL;
9     }
10    else{
11        nodo_t *ultimo = *cab;
12        nodo_t *penultimo = NULL;
13        while(ultimo->sig != NULL){
14            penultimo = ultimo;
15            ultimo = ultimo->sig;
16        }
17        free(ultimo);
18        penultimo->sig = NULL;
19    }
20 }
```



Función para eliminar al final de la lista (recursivo)

```
1 void eliminar_final(nodo_t **cab){
2     if(*cab == NULL){
3         return;
4     }
5     else if((*cab)->sig == NULL){
6         nodo_t *tmp = *cab;
7         free(tmp);
8         *cab = NULL;
9         return;
10    }
11    else{
12        eliminar_final(&(*cab)->sig);
13    }
14 }
```

Función para eliminar nodo arbitrario (iterativo)

Martes 27 Septiembre 2022

```
1 void elimina_nodo(nodo_t **cab, int x){
2     if(*cab == NULL){
3         return;
4     }
5     else if((*cab)->info == x){
6         nodo_t *tmp = *cab;
7         *cab = (*cab)->sig;
8         free(tmp);
9         return;
10    }
11    else{
12        nodo_t *prev = *cab, *it = (*cab)->sig;
13        while(it != NULL && it->info != x){
14            prev = it;
15            it = it->sig;
16        }
17        if(it != NULL){
18            prev->sig = it->sig;
19            free(it);
20        }
21    }
```



Función para contar el número de elementos (iterativo)

Jueves 29 Septiembre 2022

```
1 int tam_lista(const nodo_t *cab){
2     int cont = 0;
3     const nodo_t *it = cab;
4     while(it != NULL){
5         cont++;
6         it = it->sig;
7     }
8     return cont;
9 }
```

Función para contar el número de elementos (recursivo)

```
1 int tam_lista(const nodo_t *cab){
2     if(cab == NULL){
3         return 0;
4     }
5     const nodo_t *resto = cab->sig;
6     int tam_resto = tam_lista(resto);
7     return 1 + tam_resto;
8 }
```

Función para eliminar nodo arbitrario (recursivo)

```
1 void elimina_nodo(nodo_t **cab, int x){
2     if(*cab == NULL){
3         return;
4     }
5     else if((*cab)->info == x){
6         nodo_t *tmp = *cab;
7         *cab = (*cab)->sig;
8         free(tmp);
9         return;
10    }
11    else{
12        elimina_nodo(&(*cab)->sig, x);
13    }
14 }
```



Función para buscar un dato (recursivo)

```
1  nodo_t *buscar_dato(const nodo_t *cab, int dato){
2      if(cab == NULL){
3          return NULL;
4      }
5      else if(cab->info == dato){
6          return cab;
7      }
8      else{
9          buscar_dato(&(*cab)->sig, dato);
10     }
11 }
```

3. GRAFOS



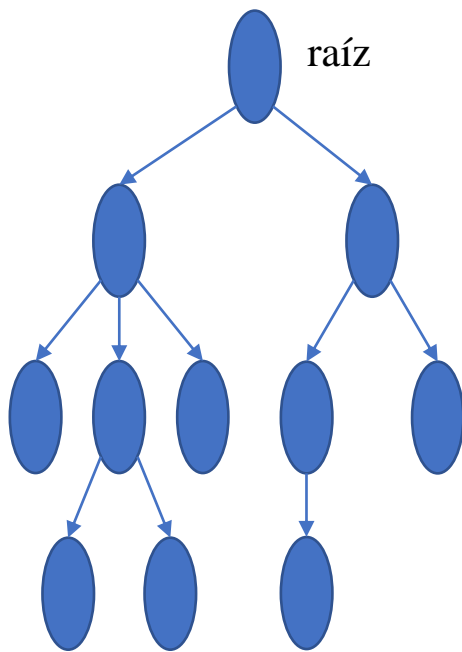
4. ÁRBOLES BINARIOS



4.1 Conceptos básicos de árboles

Objetivo: Diseñar y programar las estructuras y las operaciones básicas para la manipulación de grafos.

Lunes 07 Noviembre 2022

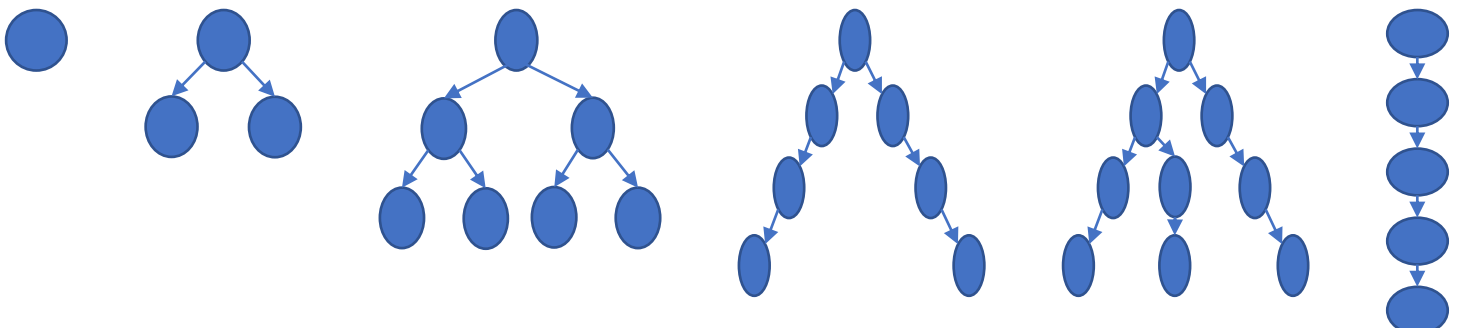


Terminología	
Raíz	Nodo principal, cabecera.
Hijos	Nodos izquierdo y derecho de un nodo.
Padre	Nodo con hijos.
Hermanos	Nodos que tienen el mismo padre.
Antecesor y descendiente	Si podemos ir desde el nodo A hacia el nodo B, entonces A es un antecesor de B, y B es un descendiente de A.
Hoja	Nodos sin hijos.
Nodo interno	Nodos que no son hojas.
Camino	Secuencia de aristas desde un nodo antecesor hasta un nodo descendiente.

Definiciones y aplicaciones

Un **árbol binario** es un árbol en el cual cada nodo puede tener a lo máximo 2 hijos.

Ejemplos:





Aplicaciones de los árboles:

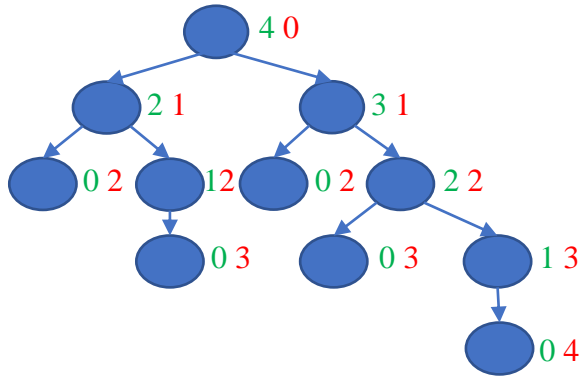
- Información con estructura jerárquica.
- Navegación web y redes.
- Bases de datos (árboles B y B+).
- Son la base para las estructuras eficientes como los conjuntos (sets).

Árbol binario completo

Todos los niveles excepto posiblemente el último están llenos y todos los nodos están lo más posible a la izquierda.

Altura de un nodo

Martes 08 Noviembre 2022



Altura de un nodo x

Es el número de aristas en el camino más largo desde x hasta una hoja. Las hojas tienen altura cero.

La altura de un árbol es la altura de la raíz (en este caso es 4).

La profundidad de un nodo x

Es la longitud del camino desde la raíz hasta el nodo x (cantidad de aristas). La profundidad de la raíz es cero.

Los nodos que tienen la misma profundidad están en un mismo nivel. En un árbol binario perfecto, todos los niveles están completamente llenos.

El número máximo de nodos en el nivel i es 2^i . Un árbol perfecto de altura h tiene exactamente $2^{h+1} - 1$ nodos.

¿Cuál es la altura de un árbol binario perfecto que tiene N nodos?

$$2^{h+1} - 1 = N \Rightarrow 2^{h+1} = N + 1$$

$$\log a^b = b \log a$$

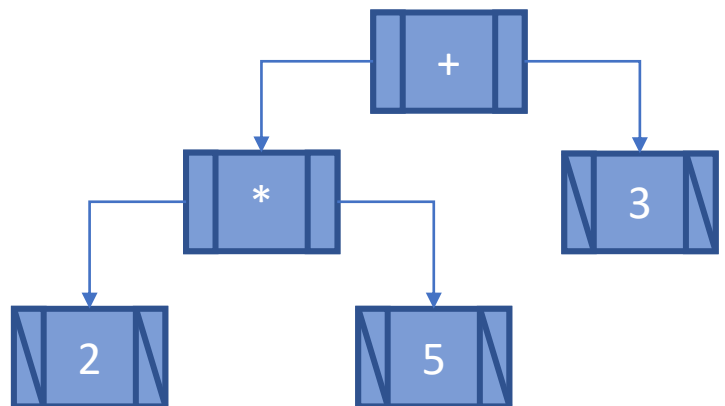
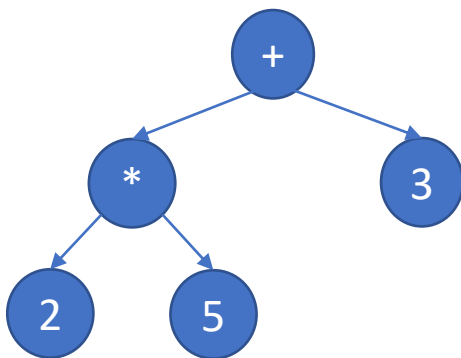
$$\log_x x = 1$$

$$\log_2(2^{h+1}) = \log_2(N + 1) \Rightarrow (h + 1) \log_2 2 = \log_2(N + 1) \Rightarrow h = \log_2(N + 1) - 1$$

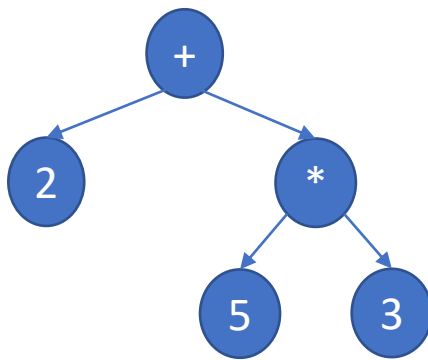
4.2

Árboles binarios de expresiones

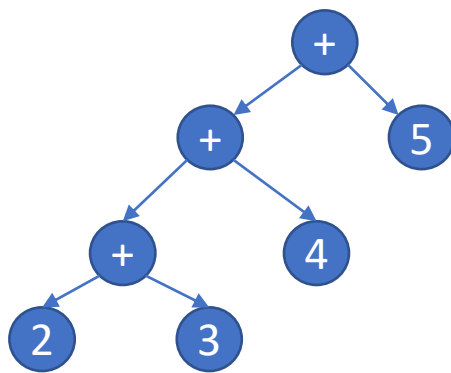
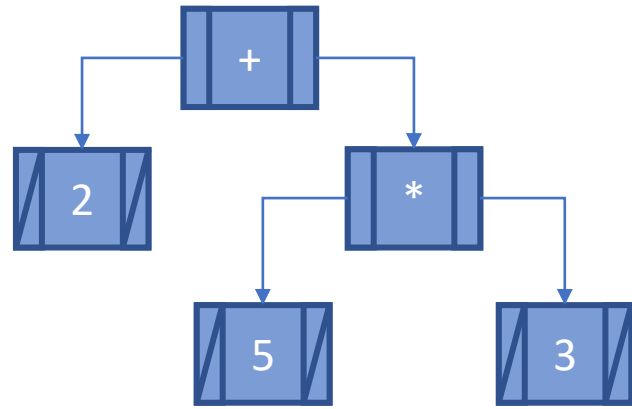
Jueves 10 Noviembre 2022



$$2 * 5 + 3 \Rightarrow 13$$

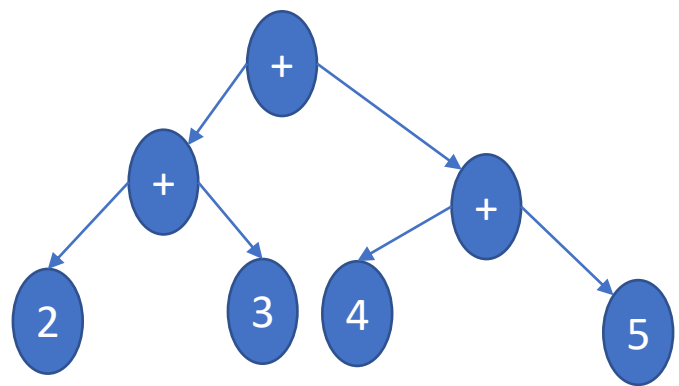


$$2 * (5 + 3) \Rightarrow 16$$



$$2 + 3 + 4 + 5 \Rightarrow 14$$

≠



$$(2 + 3) + (4 + 5) \Rightarrow 14$$

Aunque para nosotros sea la misma operación matemática, son dos árboles completamente diferentes por los paréntesis.

4.2.1

Definir la estructura del nodo

```

1 struct nodo{
2     int info;
3     struct nodo *izq;
4     struct nodo *der;
5 };
6 typedef struct nodo nodo_t;
```

5. ÁRBOLES MULTICAMINOS





