Universidad Autónoma de San Luis Potosí

Facultad de Ingeniería

Área en Ciencias de la Computación

Estructuras de Datos II

Semestre: Agosto – Diciembre 2022

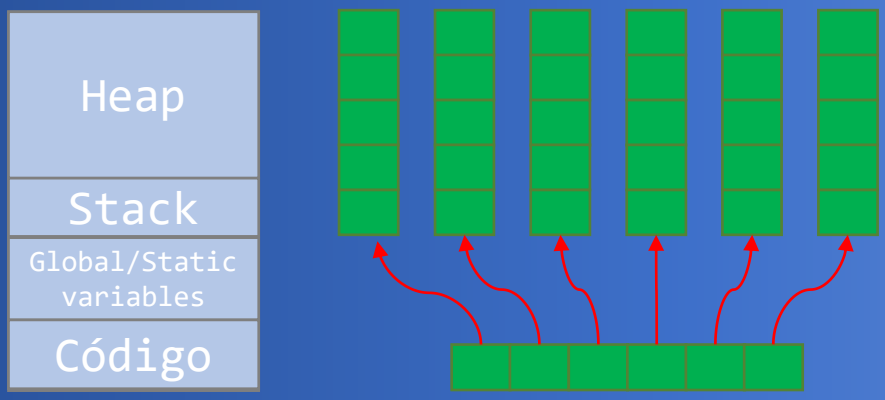
Alumno: Esparza Castañeda Hugo

Profesor: M. en C. Froylán Eloy Hernández Castro

Índice General de Temas

Martes 16 Agosto 2022

1. APUNTADORES

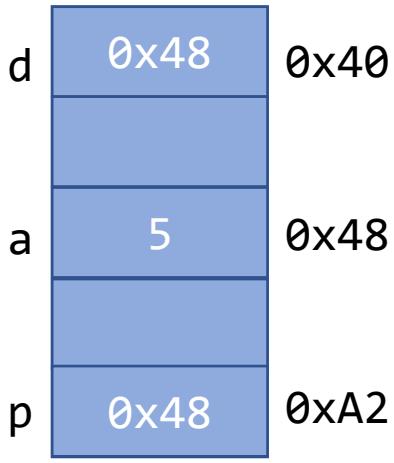


1.1 Introducción a apuntadores

Objetivo: Conocer el concepto de apuntadores 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.

```c
int main(){
    int a;
    a = 5;
    int *p;
    p = &a;
    int *d;
    d = p;
    return 0;
}
```




```c
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
```

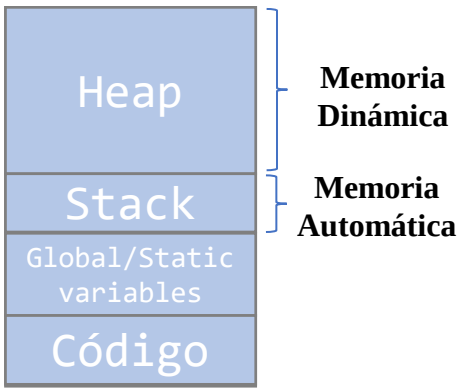
a = 10; ⇔ *p = 10

scanf("%d", p) ⇔ scanf("%d", &a)

1.2 Parámetros por valor/referencia

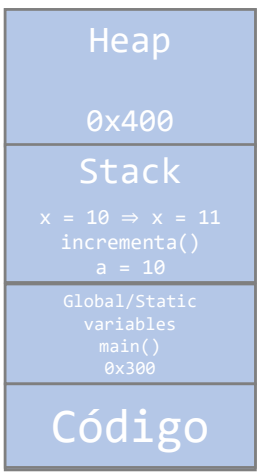
Jueves 18 Agosto 2022

Memoria

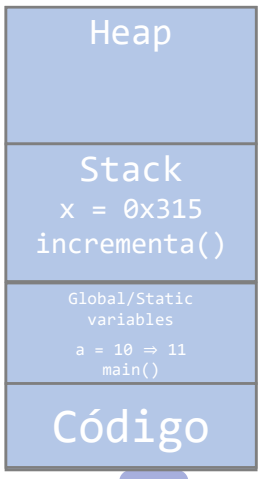



```c
#include <stdio.h>
void incrementa(int *);
int main(){
    int a;
    a = 10;
    incrementa(&a);
    printf("%d", a);
    return 0;
}

void incrementa(int *x){
    *x = *x + 1;
    return;
}
```

Por valor



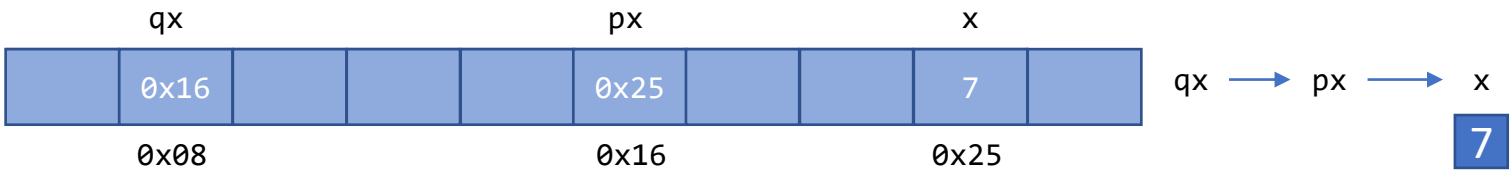
Por referencia



```
int *r;   ⇒   int *r = NULL;
```

1.3 Apuntadores a apuntadores

Viernes 19 Agosto 2022



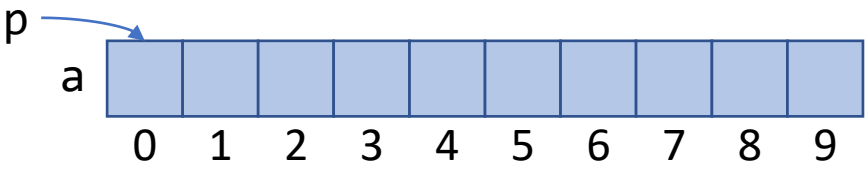

```c
#include <stdio.h>
int main(){
    int x = 7; //entero
    int *px = &x; //apuntador
    int **qx = &px; //apuntador a apunt
    printf("%p\n", px);  //output: 0x25
    printf("%d\n", *px); //output: 7
    printf("%p\n", &x);  //output: 0x25
    printf("%p\n", &px); //output: 0x16
    printf("%p\n", &qx); //output: 0x08
    printf("%p\n", qx);  //output: 0x16
    printf("%p\n", *qx); //output: 0x25
    return 0;
}
```

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

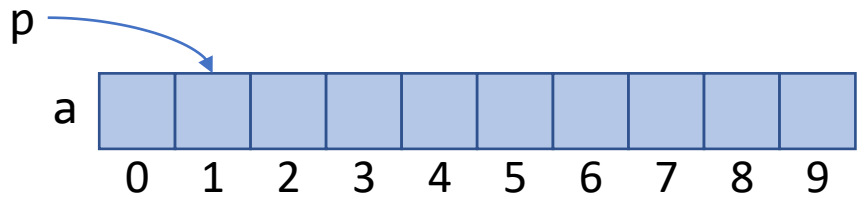
Los apuntadores a apuntadores se usan para matrices dinámicas.

1.4 Apuntadores y arreglos

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

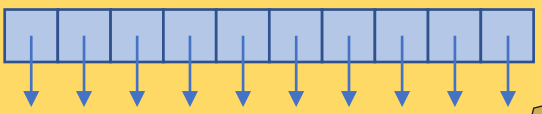



```
p = p + 1;
```



Arreglo de apuntadores

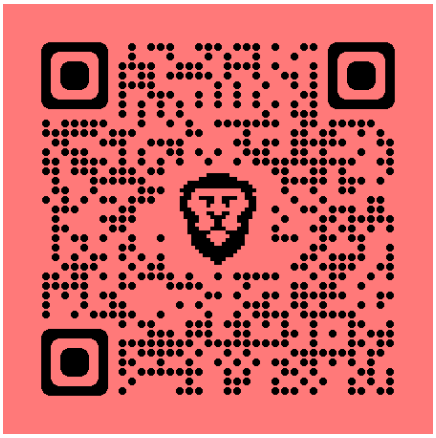
```
int *b[10];
```



Arreglo de apuntadores apuntando a NULL

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

Tarea #0: Resumen del video “Pointers”



Los punteros es uno de los temas más difíciles, sim 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.

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

Entonces, un puntero es un tipo que 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.

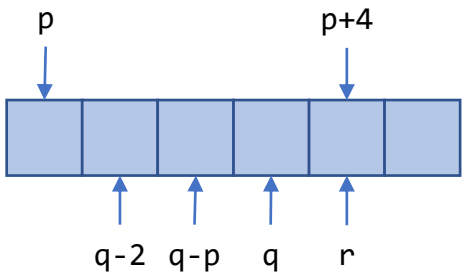
1.5 Aritmética de apuntadores

Lunes 22 Agosto 2022

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

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:

```c
#include <stdio.h>
int main(){
    int a[] = {28, 41, 7};
    int *pi = a;
    printf("%d\n", *pi); //output: 28
    pi += 1;
    printf("%d\n", *pi); //output: 41
    pi++;
    printf("%d\n", *pi); //output: 7
    ++pi;
    printf("%d\n", *pi); //output: ???
    return 0;
}
```

Ejemplo 2:

```c
#include <stdio.h>
int main(){
    short s;
    short *ps = &s;
    char c;
    char *pc = &c;
    printf("%p\n", ps); //output: 0x7ffcca0af8c6
    ps = ps + 1;
    printf("%p\n", ps); //output: 0x7ffcca0af8c8
    printf("%p\n", pc); //output: 0x7ffcca0af8c5
    pc = pc + 1;
    printf("%p\n", pc); //output: 0x7ffcca0af8c6
    return 0;
}
```

short = 2 bytes, char = 1 byte

Martes 23 Agosto 2022

Ejemplo 3: Función strlen

```c
int strlen(char *s){
    char *p = s;
    while(*p != '\0'){
        p++;
    }
    return p-s;
}
```

Ejemplo 4: Función strcpy

```c
void strcpy(char *s, char *t){
    while((*s = *t) != '\0'){
        s++;
        t++;
    }
    return;
}
```

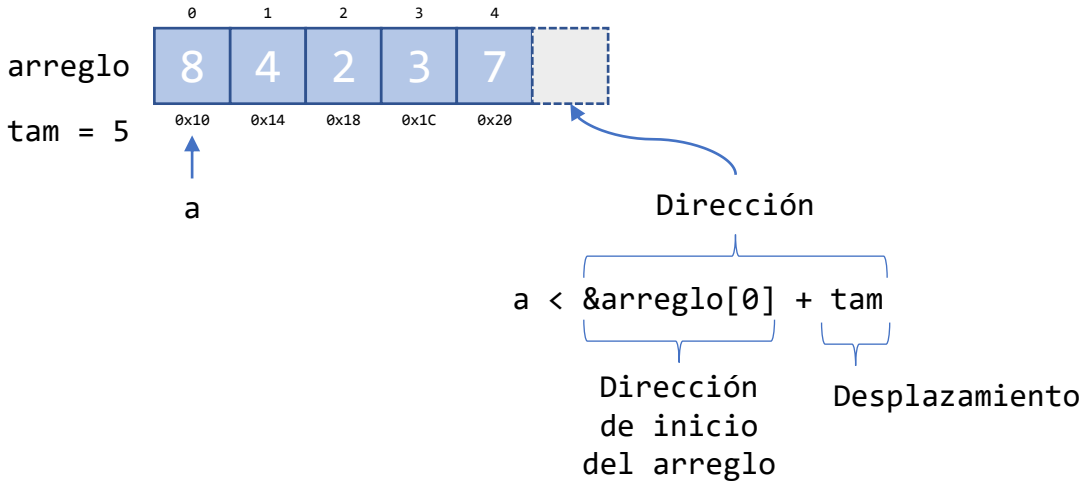
Ejemplo 5: Función strcmp

```c
int strcmp(char *s, char *t){
    for(;*s == *t; s++, t++){
        if(*s == '\0'){
            return 0;
        }
    }
    return *s - *t;
}
```

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

```c
#include <stdio.h>
int suma(int, int []);
int main(){
    int tam = 5;
    int arreglo[5] = {8, 4, 2, 3, 7};
    int p = suma(tam, arreglo);
    printf("%d\n", p);
    return 0;
}

int suma(int tam, int arreglo[tam]){
    int sum = 0;
    int *a = arreglo;
    for(int *a = arreglo; a - arreglo < tam; a++){
        sum += *a;
    }
    return sum;
}
```

Con ciclo while (más elegante)

```c
#include <stdio.h>
int suma(int, int []);
int main(){
    int tam = 5;
    int arreglo[5] =
    {8, 4, 2, 3, 7};
    int p = suma(tam, arreglo);
    printf("%d\n", p);
    return 0;
}

int suma(int tam, int arreglo[tam]){
    int sum = 0;
    int *a = arreglo;
    int *b = &arreglo[tam];
    while(a < b){
        sum += *a++;
    }
    return sum;
}
```

```
arreglo ⇒ &arreglo[0]

&arreglo[tam] ⇒ 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

```c
#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

```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)));
}
```

```c
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

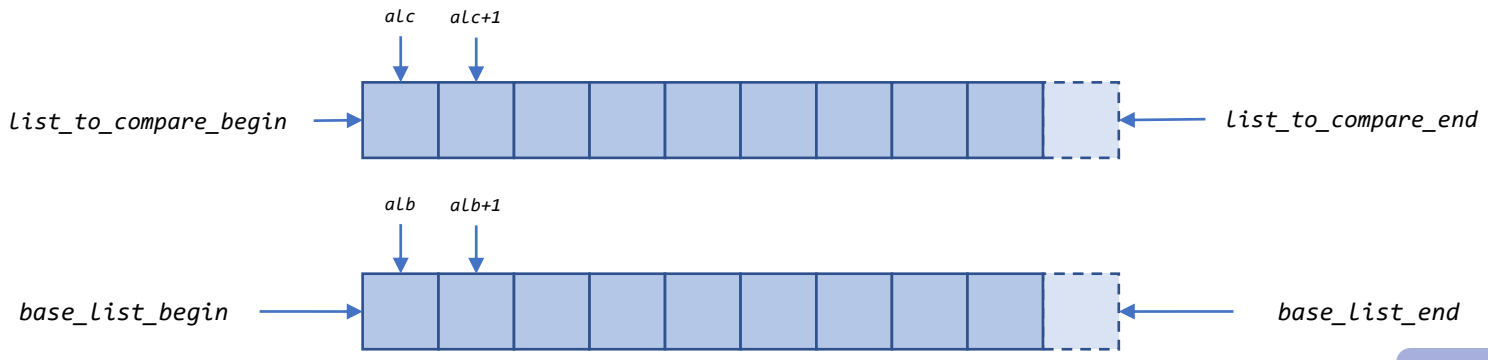
```c
#include "sublist.h"
comparison_result_t check_lists(int *list_to_compare_begin, int *list_to_compare_end,
                                int *base_list_begin, int *base_list_end){
    //Test 01
    //Primero verificamos que no esten vacios los dos
    if(list_to_compare_begin == NULL && base_list_begin == NULL){
        return EQUAL;
    }
    //Test 02
    //Revisamos si la lista a comparar esta vacia, pero otra no
    else if(base_list_begin != NULL && list_to_compare_begin == NULL){
        return SUBLIST;
    }
    //Test 03
    //Revisamos si la lista base esta vacia, pero la segunda no
    else if(list_to_compare_begin != NULL && base_list_begin == NULL){
        return SUPERLIST;
    }
    //Cuando las dos listas no estan vacias
    else{
        //Cuando son del mismo tamaño
        if(list_to_compare_end - list_to_compare_begin == base_list_end - base_list_begin){
            int *alc = list_to_compare_begin;
            int *alb = base_list_begin;
            //Compara uno por uno
            while(alc < list_to_compare_end){
                if(*alc == *alb){
                    alc++;
                    alb++;
                }
                //Si hay por lo menos uno diferente
                //Test 05, 16, 17, 19
                else{
                    return UNEQUAL;
                }
            }
            //Si el ciclo termina sin diferencias
            //Test 04
            return EQUAL;
        }
        //Cuando la lista base es mas grande
        else if(base_list_end - base_list_begin > list_to_compare_end - list_to_compare_begin){
            int *alc = list_to_compare_begin;
            int *alb = base_list_begin;
            int t = 0;
            int *tmp = NULL;
            while(alb < base_list_end){
                if(*alc == *alb){
                    alc++;
                    t++;
                    if(t == 1){
                        tmp = alb;
                    }
                    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 (podía 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, esta 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 sublista, a continuación pongo cada función.

Prototipos de funciones y función principal

```c
#include "sublist.h"

comparison_result_t iguales(int *alc, int *alb, int *blc);

comparison_result_t tipo_arr(int *alc, int *alb, int *blc, int *blb);

comparison_result_t check_lists(int *list_to_compare_begin, int *list_to_compare_end,
                                int *base_list_begin, int *base_list_end){
    if(list_to_compare_begin == NULL && base_list_begin == NULL){
        return EQUAL;
    }
    else if(base_list_begin != NULL && list_to_compare_begin == NULL){
        return SUBLIST;
    }
    else if(list_to_compare_begin != NULL && base_list_begin == NULL){
        return SUPERLIST;
    }
    else{
        if(list_to_compare_end - list_to_compare_begin == base_list_end - base_list_begin){
            return iguales(list_to_compare_begin, base_list_begin, list_to_compare_end);
        }
        else if(base_list_end - base_list_begin > list_to_compare_end - list_to_compare_begin){
            return tipo_arr(list_to_compare_begin, base_list_begin, list_to_compare_end, base_list_end);
        }
        else{
            if(tipo_arr(base_list_begin, list_to_compare_begin, base_list_end, list_to_compare_end) == 1){
                return UNEQUAL;
            }
            else{
                return SUPERLIST;
            }
        }
    }
}
```

Función para comprobar si son iguales

```c
comparison_result_t iguales(int *alc, int *alb, int *blc){
    while(alc < blc){
        if(*alc == *alb){
            alc++;
            alb++;
        }
        else{
            return UNEQUAL;
        }
    }
    return EQUAL;
}
```

Función para saber si es sublista o superlista

```c
comparison_result_t tipo_arr(int *alc, int *alb, int *blc, int *blb){
    int *beg = alc;
    int *beg2 = blc;
    int t = 0;
    int *tmp = NULL;
    while(alb < blb){
        if(*alc == *alb){
            alc++;
            t++;
            if(t == 1){
                tmp = alb;
            }
            else if(t == beg2 - beg){
                return SUBLIST;
            }
            alb++;
        }
        else{
            alc =  beg;
            if(t == 0){
                alb++;
            }
            else{
                t = 0;
                alb = tmp + 1;
            }
        }
    }
    return UNEQUAL;
}
```

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 reescrbir mi código.

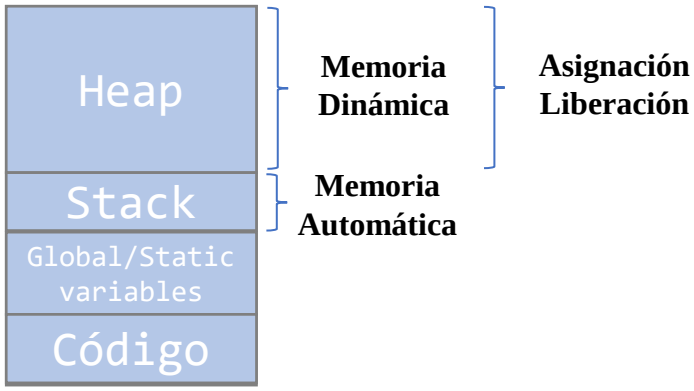
10. ¿Qué conocimientos nuevos adquirió?

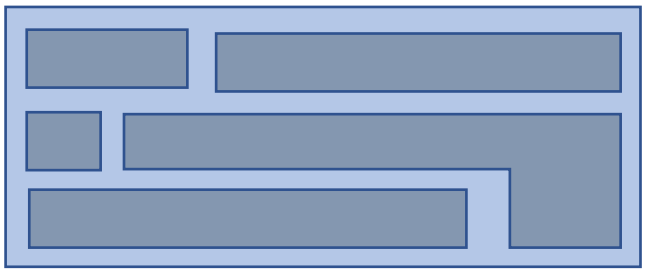
Primeramente, el `enum`, porque no lo conocía, en segunda, esta tarea me ayudo a terminar de entender la aritmética de apuntadores.

1.6 Gestión de memoria dinámica

Lunes 29 Agosto 2022

Memoria





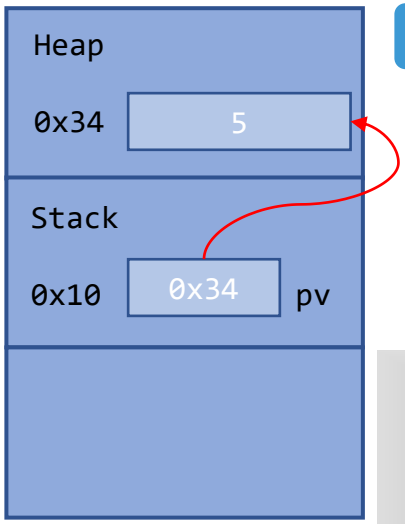
Fragmentación

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 especifico 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.



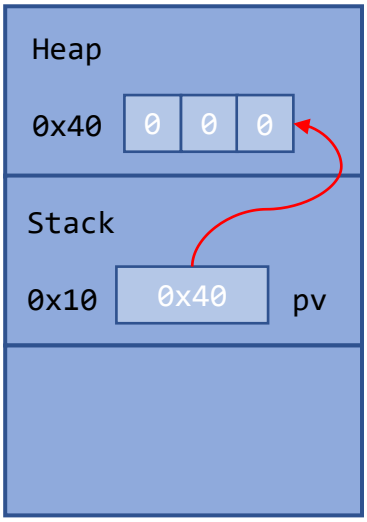
Liberación

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

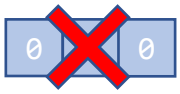
free() devuelve la memoria al sistema, ya no te pertenece

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);

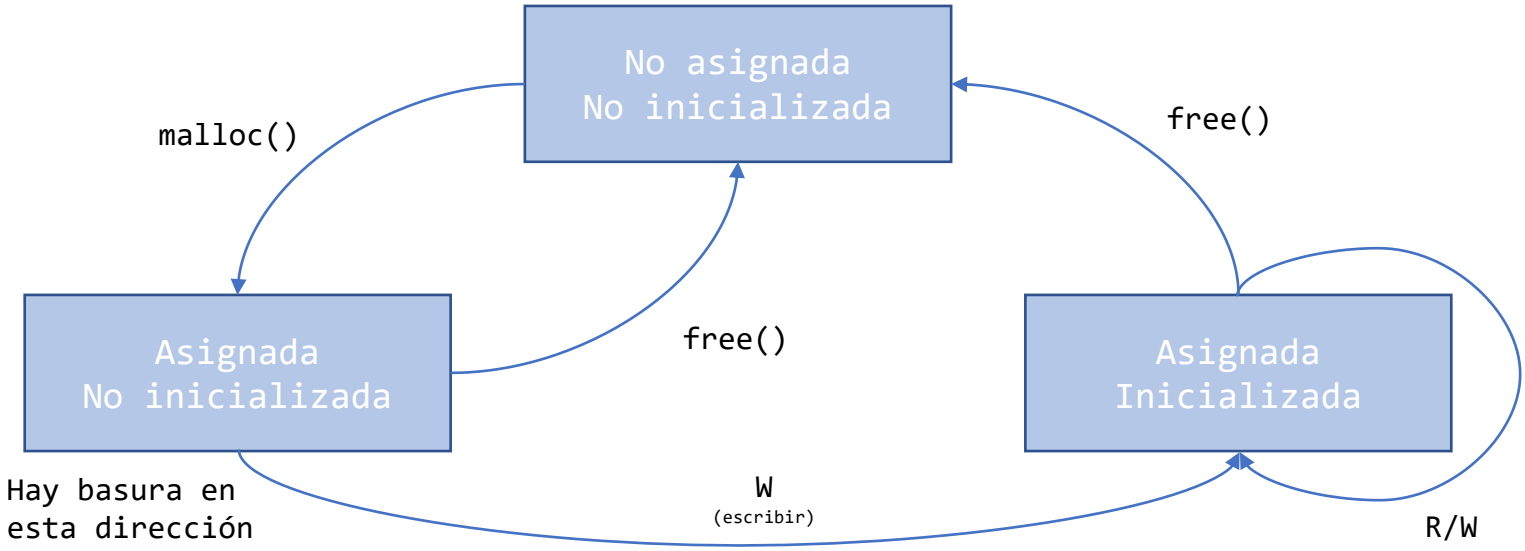


¿Cómo enterarnos si la asignación de memoria falla?

```c
#include <stdio.h>
#include <stdlib.h>
int main(){
    int *pv = NULL;
    pv = (int *)malloc(sizeof(int));
    if(pv == NULL){
        printf("Error: no hay memoria disponible");
        exit(EXIT_FAILURE);
    }
    /*codigo*/
    return 0;
}
```

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)

```c
char *bytes;
while(1){
    bytes = (char *)malloc(10);
}
```

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

```c
double *d = NULL;
d = (double *)malloc(100);
double pi = 3.1416;
d = &pi;
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.

```c
double *d = NULL;
d = (double *)malloc(100);
double pi = 3.1416;
double *f = d;
d = &pi;
free(f);
```

Por cada malloc() asigna un free()

2) Wild pounters (apuntadores no inicializados)
Apuntadores que no apuntan a nada.

Cada puntero se inicializa con un NULL

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

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

Asignar NULL al apuntador después de free()

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

1.9 Arreglos dinámicos

Jueves 01 Septiembre 2022

```c
#include <stdio.h>
#include <stdlib.h>
int *crea_arreglo(int);
int main(){
    int n;
    scanf("%d", &n);
    int *a = crea_arreglo(n);
    for(int i = 0; i < n; i++){
        scanf("%d", &a[i]);
    }
    for(int j = 0; j < n; j++){
        printf("%d ", a[j]);
    }
    printf("\n");
    free(a);
    return 0;
}

int *crea_arreglo(int tam){
    int *arreglo = NULL;
    arreglo = (int *)malloc(tam * sizeof(int));
    if(arreglo == NULL){
        printf("No hay memoria\n");
        exit(EXIT_FAILURE);
        return NULL;
    }
    return arreglo;
}
```

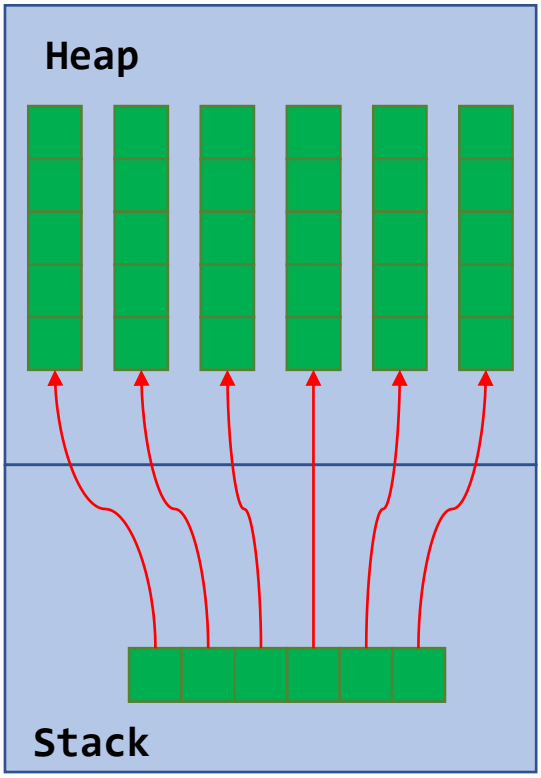
Las líneas 9 y 12 las podemos cambiar, o bien, cambiar los dos ciclos for, tal como se muestra a continuación y exactamente lo mismo:

```c
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

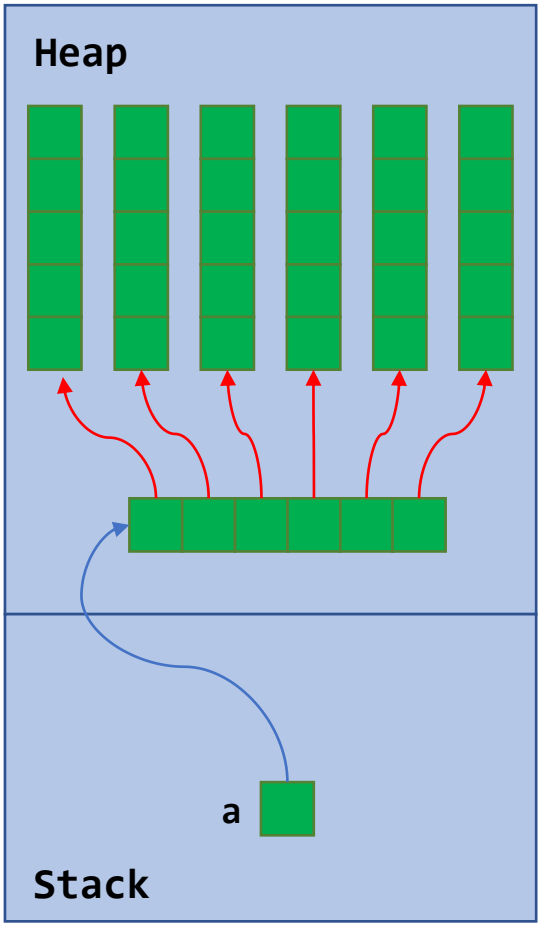
Viernes 02 Septiembre 2022

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
int main(){
    int *a[6] = {NULL};
    for(int i = 0; i < 6; i++){
        a[i] = (int *)malloc(5 * sizeof(int));
        assert(a[i] != NULL);
    }
    for(int j = 0; j < 6; j++){
        free(a[j]);
    }
    return 0;
}
```



También podemos liberar la matriz de esta forma:

```c
    int **p = a;
    for(int j = 0; j < 6; j++){
        free(*(p + j));
    }
    return 0;
}
```

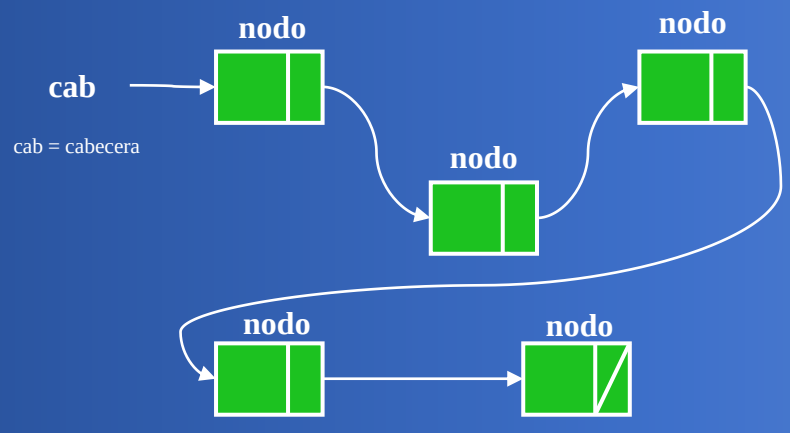



```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
int main(){
    int **a = NULL;
    a = (int **)malloc(6 * sizeof(int *));
    assert(a != NULL);
    for(int i = 0; i < 6; i++){
        a[i] = (int *)malloc(5 * sizeof(int));
        assert(a[i] != NULL);
    }
    int **p = a;
    for(int j = 0; j < 6; j++){
        free(*(p + j));
    }
    free(a);
    return 0;
}
```

Implementación de una matriz dinámica con funciones

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
int **crea_matriz(int, int);
void llenar_matriz(int **, int, int);
void imprimir_matriz(int **, int, int);
void liberar_matriz(int **);
int main(){
    int r, c;
    scanf("%d", &r);
    scanf("%d", &c);
    int **a = crea_matriz(r, c);
    llenar_matriz(a, r, c);
    imprimir_matriz(a, r, c);
    liberar_matriz(a);
    return 0;
}

int **crea_matriz(int r, int c){
    int **a = NULL;
    a = (int **)malloc(c * sizeof(int *));
    assert(a != NULL);
    for(int i = 0; i < c; i++){
        a[i] = (int *)malloc(r * sizeof(int));
        assert(a[i] != NULL);
    }
    return a;
}

void llenar_matriz(int **matriz, int r, int c){
    int **a = matriz;
    for(int i = 0; i < c; i++){
        for(int j = 0; j < r; j++){
            scanf("%d", &a[i][j]);
        }
    }
}

void imprimir_matriz(int **matriz, int r, int c){
    int **a = matriz;
    for(int i = 0; i < c; i++){
        for(int j = 0; j < r; j++){
            printf("%d ", a[i][j]);
        }
        printf("\n");
    }
}

void liberar_matriz(int **matriz){
    int **p = matriz;
    for(int j = 0; j < 6; j++){
        free(*(p + j));
    }
    free(matriz);
}
```

2. LISTAS ENLAZADAS

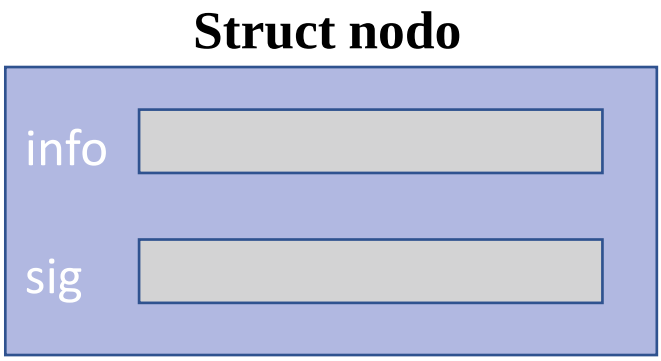


2.1 Listas simples

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

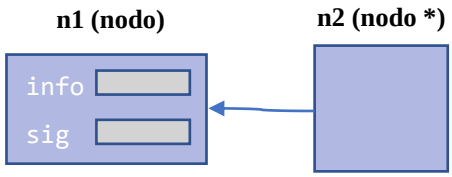
En una lista podemos guardar cualquier cosa, para este ejemplo de lista simple, haremos una lista de números enteros, cada casilla es un nodo, y para cada nodo hacemos una estructura con los datos que almacenara cada nodo.

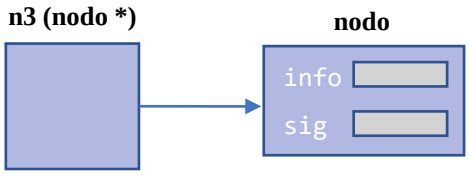
2.1.1 Definir la estructura del nodo

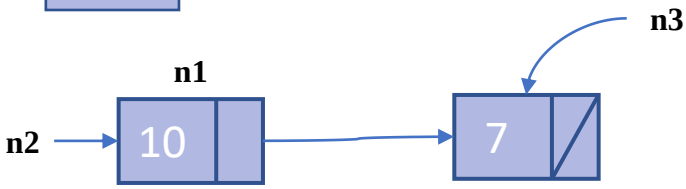



```c
struct nodo{
    int info;
    struct nodo *sig;
};
typedef struct nodo nodo_t;
```

```c
int main(){
    nodo_t n1;
    nodo_t *n2, *n3;
    n1.info = 5;
    n1.sig = NULL;
    n2 = &n1;
    n2->info = 10;
    printf("%d", n1.info);
    n3 = (nodo_t *)malloc(sizeof(nodo_t));
    assert(n3 != NULL);
    n3->info = 7;
    n3->sig = NULL;
    n2->sig = n3;
    return 0;
}
```







Output

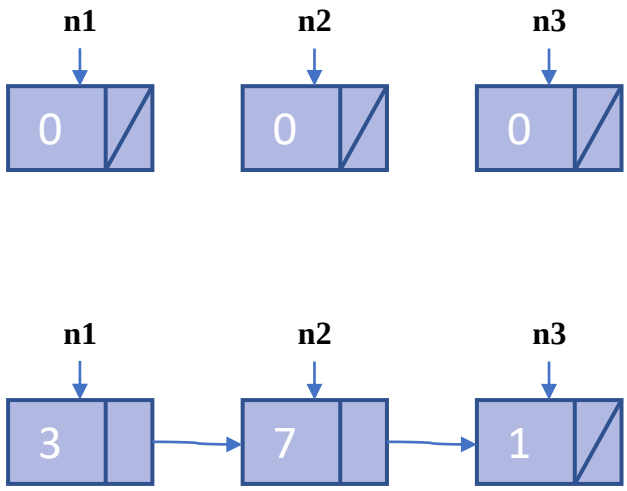
10

2.1.2 Función para crear un nodo

```c
nodo_t *crea_nodo(){
    nodo_t *nodo = NULL;
    nodo = (nodo_t*)malloc(sizeof(nodo_t));
    if(nodo == NULL){
        printf("Error: no hay memoria suficiente");
        exit(EXIT_FAILURE);
    }
    nodo->info = 0;
    nodo->sig = NULL;
    return nodo;
}
```

Ejemplo Burdo

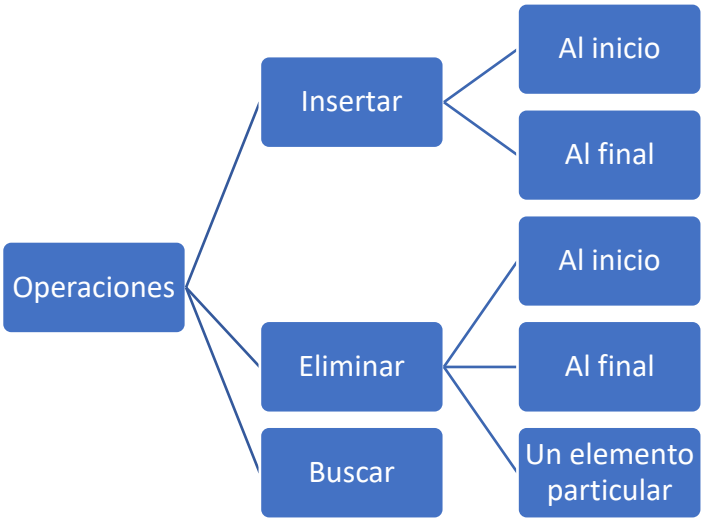
```c
int main(){
    nodo_t *n1 = crea_nodo();
    nodo_t *n2 = crea_nodo();
    nodo_t *n3 = crea_nodo();
    n1->info = 3;
    n2->info = 7;
    n3->info = 1;
    n1->sig = n2;
    n2->sig = n3;
    printf("%d ", n1->info);
    printf("%d ", n1->sig->info);
    printf("%d", n1->sig->sig->info);
    free(n1);
    free(n2);
    free(n3);
    return 0;
}
```



Output

3 7 1

2.1.3 Operaciones en listas enlazadas



2.1.3 Creando un menú de opciones

```c
int main(){
    nodo_t *lista = NULL;
    int opcion = 0, num = 0;
    do{
        opcion = selecciona_opcion();
        switch(opcion){
            case 1: printf("Valor del numero: ");
                scanf("%d", &num);
                insertar_inicio(&lista, num);
                break;
            case 2: printf("Valor del numero: ");
                scanf("%d", &num);
                insertar_final(&lista, num);
                break;
            case 3: eliminar_inicio(&lista);
                break;
            case 4: eliminar_final(&lista);
                break;
            case 5: imprimir_lista(lista);
                break;
            case 6: printf("Valor del numero: ");
                scanf("%d", &num);
                elimina_nodo(&lista, num);
                break;
            case 7: liberar_lista(&lista);
                break;
            case 8: printf("Valor del numero: ");
                scanf("%d", &num);
                printf("%p\n", buscar_dato(lista, num));
                break;
            default: puts("Opcion no valida");
                break;
        }
    }while(opcion != 0);
    return 0;
}
```

2.1.5 Función “int selecciona_opción();”

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

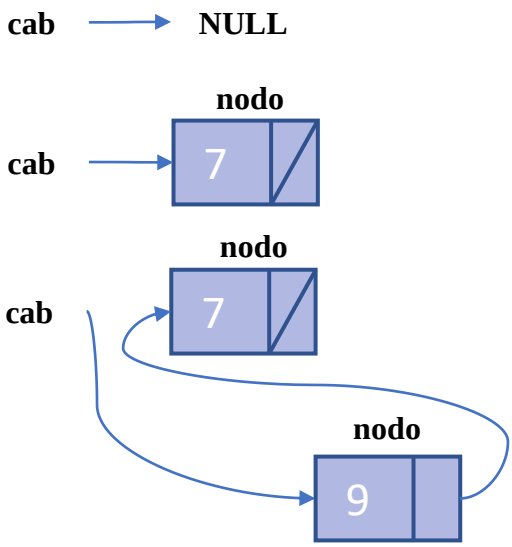
```c
int selecciona_opcion(){
    puts("Selecciona una opcion");
    puts("0. Salir");
    puts("1. Insertar inicio");
    puts("2. Insertar al final");
    puts("3. Eliminar al inicio");
    puts("4. Eliminar al final");
    puts("5. Imprimir toda la lista");
    puts("6. Eliminar nodo arbitrario");
    puts("7. Eliminar/Liberar toda la lista");
    puts("8. Buscar dato");
    int opcion;
    scanf("%d", &opcion);
    return opcion;
}
```

2.1.5 Función “void insertar_inicio(nodo_t **, int);”

```c
void insertar_inicio(nodo_t **cab, int dato){
    nodo_t *nodo = crea_nodo();
    nodo->info = dato;
    nodo->sig = *cab;
    *cab = nodo;
}
```

Ejemplo:

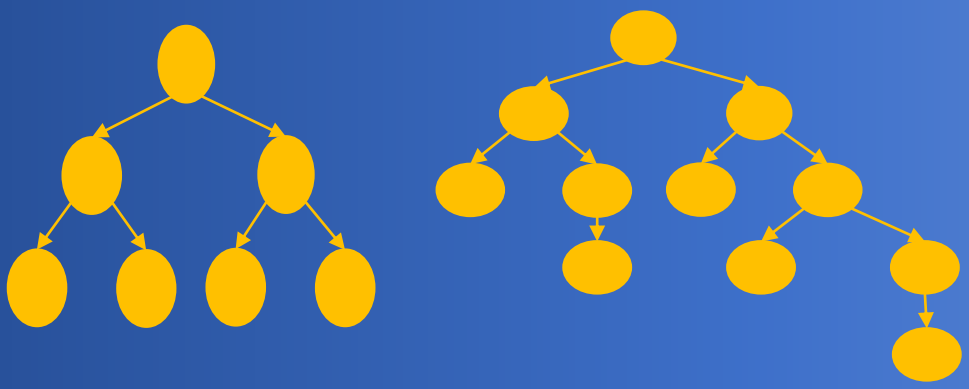
```c
int main(){
    nodo_t *lista = NULL;
    insertar_inicio(&lista, 7);
    insertar_inicio(&lista, 9);
    return 0;
}
```



3. GRAFOS

Viernes 04 Noviembre 2022

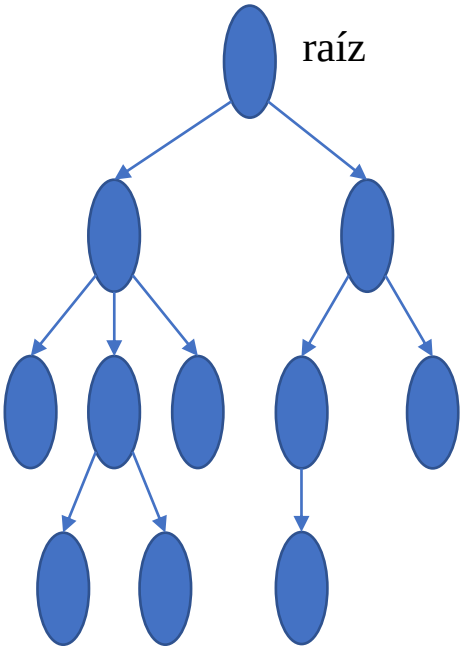
4. ÁRBOLES BINARIOS



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

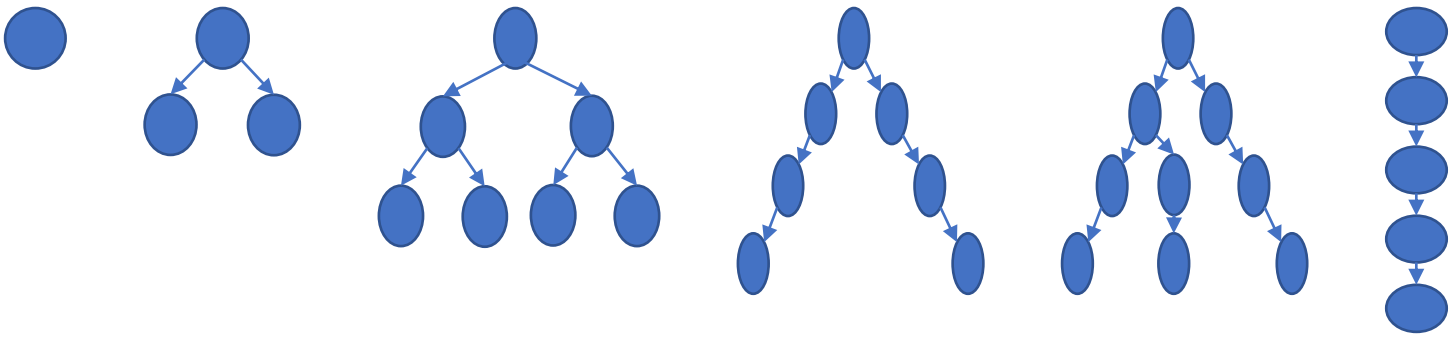


Terminología	
Raíz	Nodo principal, cabecera.
Hijos	Nodos izquierdo y derecho de un nodo.
Padre	Nodo con hijos.
Hermanos	Nodos que tienen el mismo padre.
Antecesor y descendiente	Si podemos ir desde el nodo A hacia el nodo B, entonces A es un antecesor de B, y B es un descendiente de A.
Hoja	Nodos sin 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:

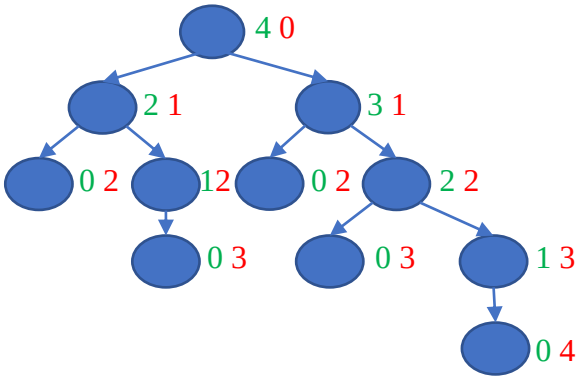
- Información con estructura jerárquica.
- Navegación web y redes.
- Bases de datos (árboles B y B+).
- Son la base para las estructuras eficientes como los conjuntos (sets).

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 $2^{n+1} - 1$ nodos.

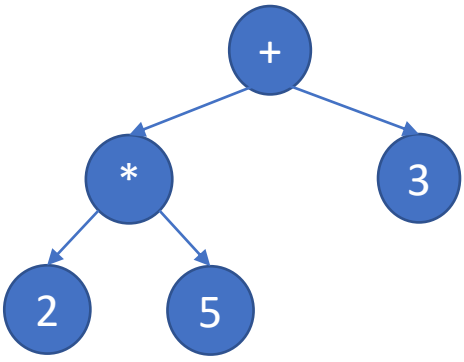
¿Cuál es la altura de un árbol binario perfecto que tiene N nodos?

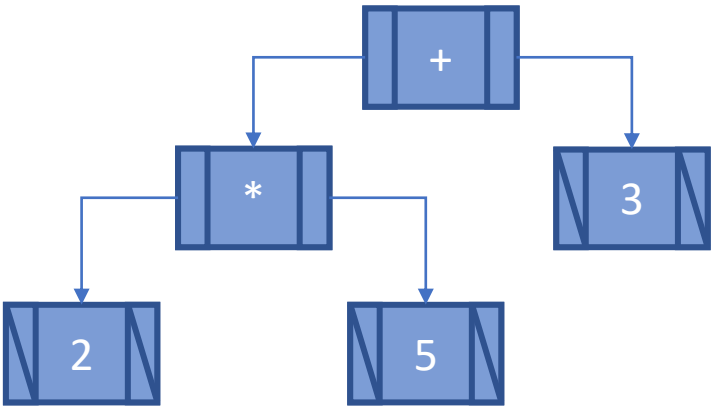
$$2^{n+1} - 1 = N \Rightarrow 2^{n+1} = N + 1 \qquad \log a^b = b \log a \qquad \log_x x = 1$$

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

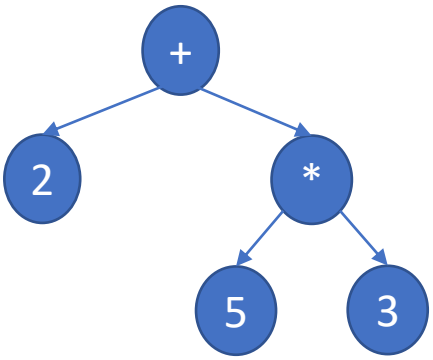
4.2 Árboles binarios de expresiones

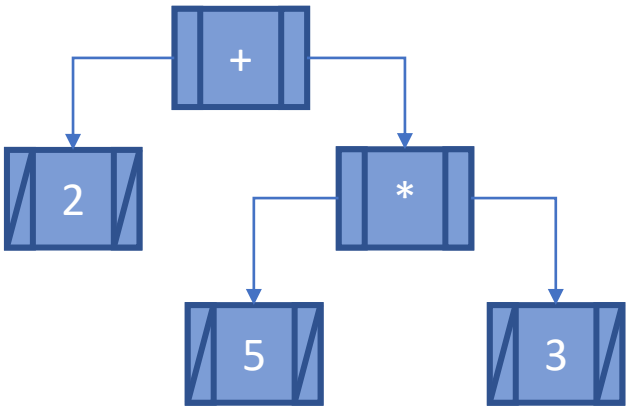
Jueves 10 Noviembre 2022



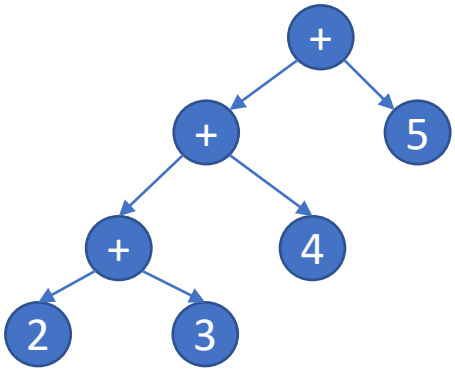


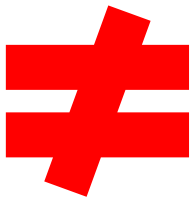
$$2 * 5 + 3 \Rightarrow 13$$

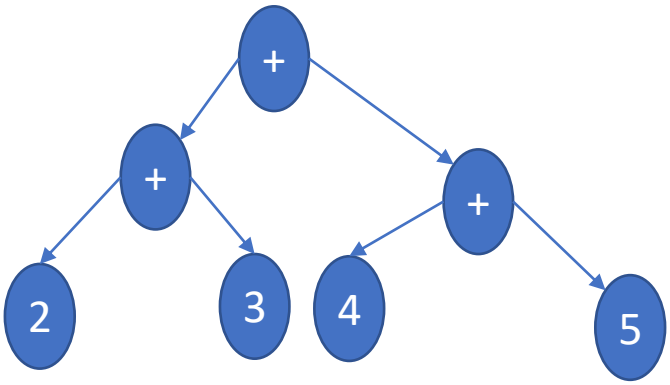




$$2 * (5 + 3) \Rightarrow 16$$







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

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

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

4.2.1 Definir la estructura del nodo

```c
struct nodo{
    int info;
    struct nodo *izq;
    struct nodo *der;
};
typedef struct nodo nodo_t;
```

5. ÁRBOLES MULTICAMINOS