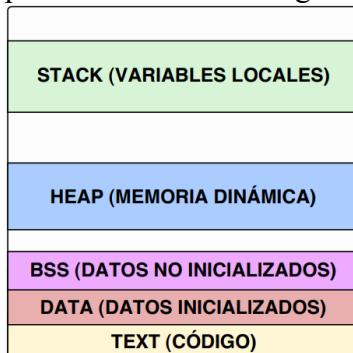


ALGO2-RESUMEN FINAL

1. MEMORIA

A. Estructura de la memoria de un programa

¿Cómo utilizo la RAM? Por bloques de memoria organización para trabajar coherentemente. El sistema operativo realiza esta organización.



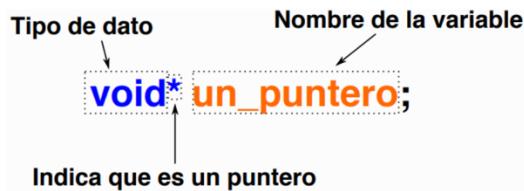
Detalles sobre la memoria:

- i. **Stack** de ejecución (pila): se almacenan todas las variables locales de las funciones que se encuentran en ejecución
- ii. **Heap** (memoria dinámica): memoria que puede estar disponible para ser utilizada por el programador en tiempo de ejecución. Hasta el momento de ser reservada, no se cuanto necesito
- iii. **Data**: constantes
- iv. **Text**: código fuente del programa (el que el programador escribe)

B. Punteros

Guarda direcciones de memoria (apunta a la caja).

Aclaración! → NO ASIGNAR NUNCA NUMEROS A PUNTEROS



- La memoria:
Donde almaceno datos.

La puedo pensar como un arreglo de celdas, que cada una posee un nombre asociado. Ese nombre es un número, la longitud del mismo depende de la computadora.

Solo puedo acceder a la memoria que me pertenece.

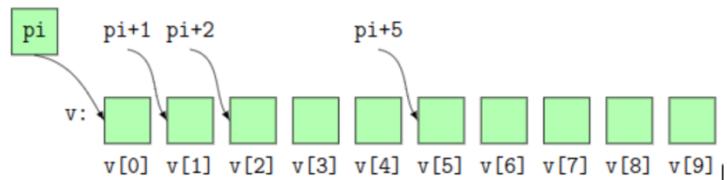
Tipo	Tamaño (bytes)
char, unsigned char	1
short int, unsigned short int	2
int, unsigned int, long int, unsigned long int	4
float	4
double	8
long double	12

- Punteros y notación de arreglos

Notación Arreglo	Notación de Puntero
array[0]	*array
array[1]	*(array+1)
array[2]	*(array+2)
array[3]	*(array+3)
array[n]	*(array+n)

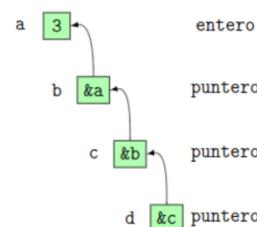
Figura 1: equivalencia notación arreglo-puntero

```
int v[10];
int *pi;
pi=v;
```



- Múltiples indirecciones

```
1 int      a = 3;
2 int      *b = &a;
3 int      **c = &b;
4 int      ***d = &c;
```



- Aritmética de punteros

1. **Operados de Dirección (&):** permite acceder a la dirección de memoria de una variable.

Ejemplo copado

```
int numero = 5;
int* p_numero;

p_numero = &numero //me da la dirección de memoria de numero (guardo direc)
*p_numero = 10 // ahora numero = 10, modiflico donde vive numero
```

2. **Operador de Indirección (*):** Además de permitir declarar un tipo de dato puntero, permite ver el valor que está en la dirección
3. **Incrementos (++) y decrementos (-):** se puede usar un puntero como si fuera un array

C. Memoria dinámica

Es aquella memoria que es reservada y utilizada únicamente en tiempo de ejecución. Es reservada en el heap. Debo solicitarla de forma explícita al sistema operativo

 IMPORTANTE: tener mucho cuidado con la memoria dinámica, se puede romper todo.

Dos funciones (relacionadas a memoria – son parte de stdio.h-)

- **malloc()** → Le digo al sistema operativo "che quiero usar memoria del heap"
- **free()** → Libero a esa memoria cuando ya la use #libresoy

Pasos para usar la memoria dinámica:

1. Reservo la memoria
2. Cuando no la utilizo mas y no necesito este espacio reservado, la libero

malloc():

- Reserva size bytes y devuelve un puntero a la memoria reservada.
- **Devuelve *void**
- **Tengo que saber el tipo de dato**

¿Cuanta memoria tengo que ir a buscar para números? Y bueno, yo quiero guardarme 5 del tipo int, vos fijate cuantos buscas (int: 4 bytes, entonces reservaría 4*5 bytes → donde 4 bytes es 1 int).

 Si falla devuelve NULL, osea que puede fallar: verificar su correcto funcionamiento. Es importante que no acceda a punteros inválidos.

```
if(numero == NULL){ // numeros de abajo, donde yo use malloc return -1; //  
también debería chequearlo para buffer }
```

ejemplo (pues sino crisis)

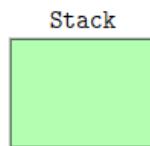
```

int main (){
    int valor_entero;
    int *ptr_entero;
    valor_entero = 5;
    ptr_entero = malloc( sizeof(int) );
    (*ptr_entero) = 10; // ptr_entero apunta a un array int que contiene 10 free(
    ptr_entero ); // acá ptr_entero tiene caca return 0;
}

```

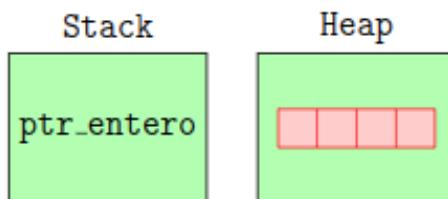
La variable `ptr_entero` se declara y vive durante la ejecución del programa en el **stack**.

`ptr_entero;`

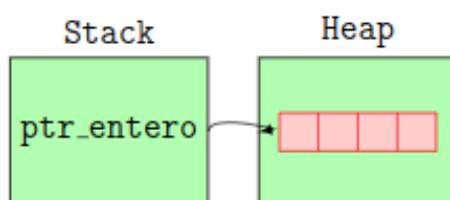


Ahora me pinto reservar memoria dinámica para un entero, entonces utilizo `malloc()`. Utilizo `sizeof()` que dado un tipo de dato devuelve la cantidad de bytes que este ocupa según la arquitectura.

`ptr_entero = malloc(sizeof(int));`



`malloc()` reserva la cantidad de bytes de memoria devuelta por `sizeof(int)`. Esta memoria pertenece al Heap. Una vez reservada la cantidad de bytes, la misma función, devuelve la dirección del primer byte de memoria.

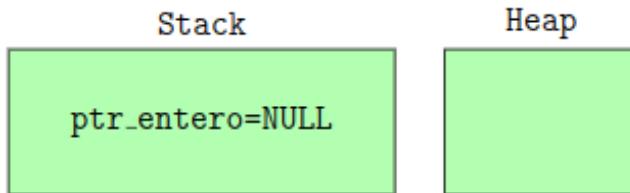


Si esta relación entre la memoria reservada en el heap y su dirección conocida desde el stack **se pierde**, la memoria pasa a ser **memoria no alcanzable** o perdida (no se como llegar a ella).

free()

Libera espacio de memoria apuntado por el puntero. Si es NULL, la operación no se realiza.

Primero libera la porción de memoria dinámica reservada con malloc() y luego inicializa al puntero que apuntaba a esa dirección con NULL (por eso la caca en el tutor).



calloc()

Inicializa a todos los punteros a NULL.

The calloc() function allocates memory for an array of nmemb elements of size bytes each and returns a pointer to the allocated memory. The memory is set to zero.

```
void *calloc(size_t nmemb, size_t sizeof(tipodedato)); //cantidad a reservar
```

realloc()

La función realloc() modifica el tamaño del bloque de memoria apuntado por el puntero en size bytes. Si el nuevo tamaño del bloque es mayor que el anterior, la memoria añadida no se encuentra inicializada en ningún valor.

⚠ Si falla devuelve NULL → estaría perdiendo la conexión entre mi stack y el heap, tengo que usar un auxiliar

Teorema fundamental de la Memoria Dinámica

"La memoria dinámica no se conserva! Siempre que se crea se debe destruir".
La memoria dinámica reservada debe liberarse cuando ya no se necesite

Tengo que lograr que siempre haya la misma cantidad de memoria dinámica reservada entre el inicio y fin: 0 bytes de memoria dinámica.

Herramienta: valgrind

D. C AVANZADO

Punteros a Funciones

Un puntero a una función es la dirección de memoria donde reside una determinada función en un programa en C. Estos pueden:

1. Ser elementos de un vector
2. Ser pasados como parámetros a una función
3. Ser devueltos por una función

```
void (*mi_funcion) (int,int)
```

EJ:

```
#include <stdio.h>
// declaracion de una funcion normal con un parametro entero
// y que devuelve un valor de tipo void.
void fun(int a) {
    printf("Value of a is %d\n", a);
}

int main(){
    // fun_ptr es un puntero a una funcion fun()
    void (*fun_ptr)(int) = &fun;

    /* la linea de arriba es equivalente a las siguientes dos lineas
        void (*fun_ptr)(int);
        fun_ptr = &fun;
    */

    // Invocando fun() usando fun_ptr
    (*fun_ptr)(10);

    return 0;
}
```

Los punteros a funciones:

- apuntan a código y no a datos, normalmente apuntan a la primera instrucción ejecutable de la función.
- no se debe reservar-liberar espacio

Conversión Explícita de Datos: Casteo

Convertir una expresión, que al ser evaluada corresponde a un tipo de dato determinado, como si perteneciera a otro tipo de dato.

El tipo de dato al cual quiere convertirse la expresión va entre paréntesis delante de la expresión convertida.

(HAY EJEMPLO)

Puntero Comodín: void*

A este tipo de puntero se le puede asignar la dirección de cualquier tipo de dato del lenguaje, ya que no está asociado a ningún tipo de dato en especial. Un puntero genérico puede apuntar a cualquier cosa

- NO puede ser desreferenciado
- NO pueden aplicarse las reglas de aritmética de punteros
- Puede ser convertido a cualquier otro tipo de dato sin una conversión explícita

```

int main()
{
    int a = 10;
    void *ptr = &a;
    printf("%d", *(int *)ptr);
    return 0;
}

```

Números hexadecimales

Dos dígitos son un byte

0x1234 // significa que esta en lenguaje hexadecimal donde
--> 34 es el menos significativo: primer byte
--> 12 es el mas significativo: segundo byte

Vector dinámico

Son como el string pero en lugar de terminar en '\0', terminan en NULL

Paso 0: creación

void** vector ——————> NULL

Paso 1: agregar elemento

Lo que agrego son PUNTEROS. Aca aparece el malloc, cambio el tamaño. Si antes no tenia nada y ahora tengo un elemento.

Le cambio el tamaño a mi vector dinámico y voy agregando cuantos elementos quiera

void** vector ——————> PUNTERO ——————> elemento que agrego el usuario

—————> PUNTERO ——————> NULL

Agrego uno mas que me apunte a NULL así se que ahí termina..

2. PUNTEROS A FUNCIONES

A. RECORDEMOS PUNTEROS

Un puntero es una variable que como dato tiene dónde está localizado algo → puede ser cualquier cosa, por lo que también puede apuntar a una función.

➡ En C se permite asignar un puntero a una dirección arbitraria → incluso un puntero a una dirección a memoria que no le pertenece al programa

Se diferencian dos tipos de punteros, que son **los punteros a datos y los punteros a funciones**. Los diferenciamos porque actúan distintos, y no podemos pasar de un puntero a otro

B. DEFINICION

Un puntero a función es una variable que almacena la dirección de memoria de una función.

Almacena la dirección de memoria de un dato que está en memoria.

Ahora, podemos acceder a una función a través de un puntero a ella.

i. ¿Para que las quiero?

Útiles cuando busco que mi código sea lo más genérico y modularizado posible.

Tengo un algoritmo, cuya lógica general es siempre la misma pero una parte varía, a veces A y a veces B → abastecer lo que cambia y parametrizarlo (recibirlo por parámetro). Logro aislar lo que cambia.

```
void funcionA(){
    < código común >
    A
    < código común >
}

void funcionB(){
    < código común >
    B
    < código común >
}
```

El tema es que ahora lo que difiere son líneas de código → utilizo **punteros a función**. Abstraigo A y B en dos funciones distintas y recibo por parámetro.

```
void funcion(void ( *funcion_extra)() ) {
    < código común >
    funcion_extra();
    < código común >
}
```

La función pasada por parámetro puede ser A, B o cualquiera que tenga esa firma.

Ahora tengo una función genérica que recibe por parámetro lo que hubiera sido dos funciones distintas: no duplico código.

ii. Sintaxis

Para uno tener un puntero a función, debo conocer que tipo de dato devuelve y cuáles parámetros recibe.

- Declaración de un puntero a función

```
char obtener_letra (int numero, bool chequeo);  
  
/*un puntero a esta función es*/  
  
char (*puntero_obtener_letra) (int,bool);  
puntero_obtener_letra = obtener_letra; // asignación  
char (*puntero_obtener_letra) (int,bool) = obtener_letra  
//inicialización
```

- Cómo recibir un puntero a una función por parámetro

```
void mostrar_letra(int repeticiones, char  
(*puntero_obtener_letra) (int,bool) );
```

- Cómo invoca una función si tengo el puntero de la misma

```
char mi_letra = puntero_obtener_letra(180, true);
```

iii. Vector de punteros a función

Supongamos que tengo estas dos funciones.

```
int* suma(int* primer_numero, int* segundo_numero);  
int* resta(int* primer_numero, int* segundo_numero);
```

- Cómo declarar un vector de punteros a función

```
int* (funciones[2] (int*,int*) = {suma, resta}
```

- Cómo invocar una función de un vector de punteros a función

Puedo invocarla de dos maneras distintas

Primera manera

```
tipo_de_dato mi_variable_resultado = vector_de_funciones[posicion]  
(variable1, variable2);  
  
//con el ejemplo anterior sería:  
int* suma_total = funciones[i] (valor1, valor2);
```

Segunda manera

```
tipo_de_dato (*funcion_actual)(tipo_de_variable1, tipo_de_variable2) =  
vector_de_funciones[posicion];  
  
tipo_de_dato nombre_variable = funcion_actual(variable1, variable2)
```

```
int* (*funcion_actual)(int*,int*) = funciones[i];
```

```
int* suma_total = funcion_actual(valor1,valor2);
```

C. OTRA MANERA DE DEFINIR PUNTEROS A FUNCION

Tengo la siguiente función a almacenar:

```
void caminar(int* x, int* y);
```

Pero la defino de la siguiente manera:

```
typedef void (*moverse)(int*,int*);
```

Entonces si la quiero almacenarla en un struct de personaje lo hago de la siguiente forma:

```
typedef struct personaje{
    size_t edad;
    char* nombre;
    int x;
    int y;
    moverse caminar; -> ahora es del tipo moverse
} personaje_t;
```

La función caminar ahora es del tipo moverse. A su vez, moverse es de tipo puntero a una función que recibe dos int* y devuelve void.

De esta manera, si mi personaje un día decide moverse en auto, esa función que recibiría los mismos parámetros y también devolvería void, podría ser fácilmente reemplazada.

D. ACLARACION DE SINTAXIS

Cuando hablamos de punteros a funciones, no necesitamos utilizar el operador & para acceder a la dirección de memoria de la función. Puedo hacerlo, pero no es necesario

```
char (*puntero_obtener_letra)(int, bool) = (&)obtener_letra;
```

Además no existen punteros dobles a funciones, o triples

```
puntero_obtener_letra = obtener_letra  
ambos
```

Son lo mismo

```
*****puntero_obtener_letra = obtener_letra
```

E. EJEMPLOS SALVADORES

Peleas de pokemones

Estamos programando un torneo de pokemones, donde se inscriben enternadores y hacen competir a sus pokemones con diferentes funciones de pelea. Tenemos varias funciones que hacen pelear a dos pokemones, éstas son algunas:

```
pokemont_t el_mas_fuerte(pokemon_t poke1, pokemon_t poke2); pokemont_t  
el_mas_inteligente(pokemon_t poke1, pokemon_t poke2); pokemont_t  
el_mas_rapido(pokemon_t poke1, pokemon_t poke2);
```

Tienen una estructura muy similar: todas reciben dos pokemones y devuelven un pokémon. Cada una tiene su lógica implementada pero de afuera se parecen.

Además de conocer su firma, conozco su implementación (la programé yo). En el torneo invocamos el tipo de pelea que queremos usar en ese momento.

Pero hay un problema, el usuario solamente puede usar las que nosotros programamos. Si quiere más tipos de pelea entre los pokemones, tiene que llamarnos a nosotros para que las agreguemos al .c y al .h... mmmm...

```
pokemon_t pelea_entre_pokemones (pokemon_t poke1, pokemon_t poke2,  
pokemon_t (*funcion) (pokemon_t, pokemon_t)) {  
  
    if (!poke1 || !poke2) return NULL; //no podemos hacer competir  
    a dos  
        //pokemones cuando al menos uno de ellos es NULL  
  
    if (!funcion) return NULL; //no puedo invocar una función nula  
    pokemont_t poke_ganador = funcion(poke1, poke2);  
        <...>  
}
```

De esta manera, recibo una función de pelea por parámetro y la invoco con los pokemones. Recibo el pokemon ganador y efectúo con éste cualquier operación necesaria. Esta función que se recibe por parámetro puede ser una de las que yo programé y agregué a mi .c, o podría ser una programada por el usuario que yo no conozco, pero que el usuario necesita.

3. ANALISIS DE ALGORITMOS

A. INTRODUCCION

Una vez analizado un problema y diseñado el algoritmo, es necesario poder determinar cuantos **recursos computacionales** consume → **tiempo o espacio**.

- Dependiendo que quiero, voy a priorizar una mejor utilización de los recursos. ¿Dónde quiero obtener una mayor eficacia?

Utilización de la memoria

- " el ancho de banda de las comunicaciones
- " del hardware

El tiempo de cómputo

Al no haber una sola de resolver los problemas → tengo que poder determinar cual es mejor algoritmo para ser utilizado en un determinado problema

¿Porque estudian algoritmos?

Se quiere saber cuanto tardará una implementación de un determinado algoritmo que se ejecuta en una determinada computadora y cuanto espacio va a ocupar.

Busca encontrar un modelo matemático para comparar los algoritmos → abstraer lo que importa y ver su comportamiento y complejidad

⚠ **No mido el tiempo de ejecución** ya que este depende del tamaño del problema. Mido de forma comparativa. Por ejemplo, cuando ordeno un vector, mientras mayor sea mi vector, mayor será mi problema

B. NOTACION BIG-O

Es una notación matemática que describe el comportamiento de una función en el límite: cuando el argumento tiende hacia un valor particular o infinito.

Big O describe específicamente el **peor de los casos** y puede usarse para describir el tiempo de ejecución requerido o el espacio usado.

- $T_{\min}(N)$ es el mejor de los casos (tiempo mínimo de ejecución)
- $T_{\max}(N)$ es el peor de los casos (tiempo máximo de ejecución)

Caracteriza a las funciones según su **tasa de crecimiento: orden de la función**.

- Ejemplo

Si un tiempo de ejecución es $O(f(n))$ entonces para un n lo suficientemente grande, el tiempo de ejecución es como mucho $k \cdot O(f(n))$ para alguna constante k dada.

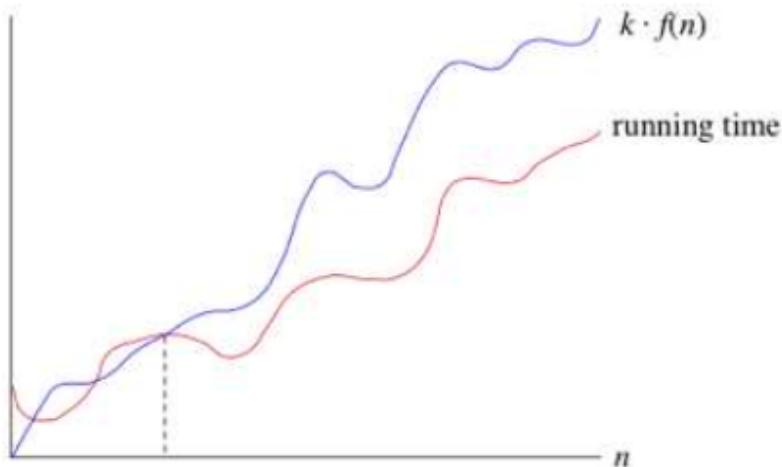
Sea $T(x)$ el tiempo de ejecución en función del Tamaño del problema.

- i. Big-O

Se dice que la tasa de crecimiento de un algoritmo es big-O:

✳️ $T(N) = O(f(N))$ si existe una constante positiva c y un n_0 tal que $T(N) \leq c * f(N)$ donde $N \geq n_0$

La tasa de crecimiento de $T(N)$ es menor o igual a $f(N)$. Está acotada por arriba: "no crece más que esto" (nunca va a tardar más)

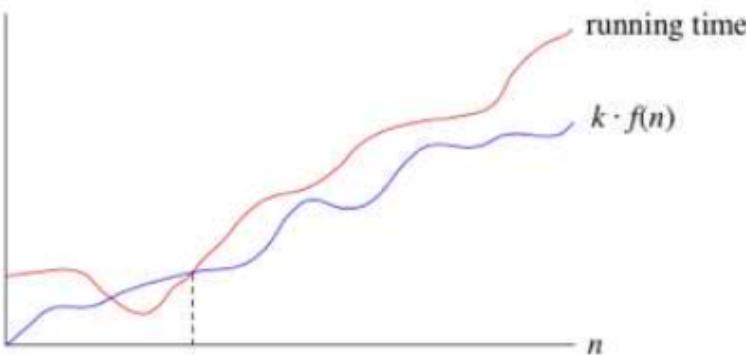


ii. Big-Omega

Se dice que la tasa de crecimiento de un algoritmo es big-Omega:

✳️ $T(N) = \Omega(g(N))$ si existe una constante positiva c y un n_0 tal que $T(N) \geq c * g(N)$ donde $N \geq n_0$

La tasa de crecimiento de $T(N)$ es mayor o igual a $f(N)$. Está acotada por abajo. El algoritmo toma *por lo menos* una cierta cantidad de tiempo en ejecutarse



iii. Propiedades de la notación

Símbolos de la notación:

$$f = \Theta(g) \quad f \text{ crece con la misma tasa que } g$$

$$f = O(g) \quad f \text{ no crece más rápido que } g$$

$$f = \Omega(g) \quad f \text{ crece a lo sumo como } g$$

$$f = o(g) \quad f \text{ crece más lentamente que } g$$

$$f \sim g \quad f/g \text{ se acerca a 1}$$

iv. Propiedades importantes

💡 Siempre se desprecian las constantes y funciones de menor orden

Regla 1

$$T_1(n) = O(f(n)), \text{ y } T_2(n) = O(g(n))$$

entonces:

a) $T_1(n) + T_2(n) = \max(O(f(n)), O(g(n)))$. Me quedo solo con el más grande de los dos (en el infinito el mayor predomina).

Ejemplo: $f(n) = n$ y $g(n) = n^2$. Me quedo con n^2

b) $T_1(n) * T_2(n) = \max(O(f(n) * (g(n))))$.

Ejemplo: $f(n) = n$ y $g(n) = n^2 \rightarrow T_1(n) * T_2(n) = n^3$

Regla 2

Si $T(n)$ es un polinomio de grado k , entonces $T(n) = O(n^k)$

Regla 3

$\log^k n = O(n)$ para cualquier constante $k \rightarrow$ tasa de crecimiento muy lenta

Valores:

Función	Nombre
c	Constante
$\log n$	Logarítmico
$\log_2 n$	Logaritmo base 2
n	Lineal
$n \log n$	Logaritmo iterado
n^2	Cuadrática
n^3	Cubica
2^n	Exponencial
$\log \log n$	Log logarítmica
$n!$	Factorial

Figura 6: Valores más comunes

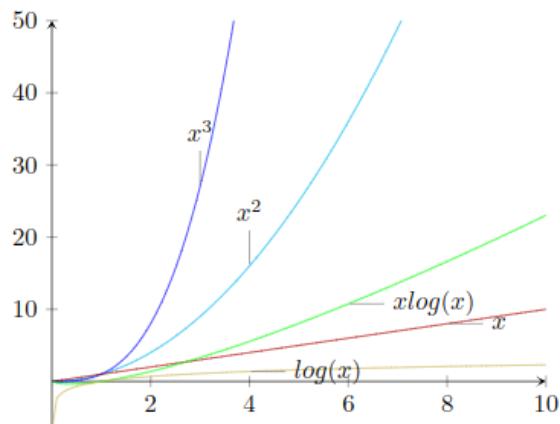


Figura 7: Comparativas de las funciones más comunes

Complejidad constante $O(1)$:

```
C ▾
void imprimir_primer_elemento(int arr[MAX_ELEMENTOS]){
    printf("Primer elemento del arreglo = %i", arr[0]);
}
```

Esta función se ejecuta en tiempo $O(1)$ (o "tiempo constante") en relación con su entrada. El arreglo de entrada podría ser 1 elemento o 1,000 elementos, pero esta función solo requeriría un paso.

Complejidad logarítmica $O(\log n)$:

```

int busqueda_binaria(int array[], int inicio, int fin, int elemento){
    if (fin >= inicio){
        int medio = inicio + (fin - inicio)/2;
        if (array[medio] == elemento)
            return medio;
        if (array[medio] > elemento)
            return busqueda_binaria(array, inicio, medio-1, elemento);
        return busqueda_binaria(array, medio+1, fin, elemento);
    }
    return -1;
}

```

Por regla general se asocia con algoritmos que "trocean" el problema para abordarlo.

Complejidad lineal $O(n)$:

```

C ▾

void imprimir_arreglo(int arr[MAX_ELEMENTOS], int tope){
    for(int i = 0; i < tope; i++)
        printf("%i\n", arr[i]);
}

```

Esta función se ejecuta en tiempo $O(n)$ (o "tiempo lineal"), donde n es el número de elementos del arreglo. Si la matriz tiene 10 elementos, tenemos que imprimir 10 veces. Si tiene 1000 elementos, tenemos que imprimir 1000 veces.
Ejemplo busqueda lineal.

Complejidad cuadrática $O(n^2)$:

Es cuando necesito iterar n veces por cada uno de los n elementos. Se incrementa al mismo ritmo que n^2 .

```

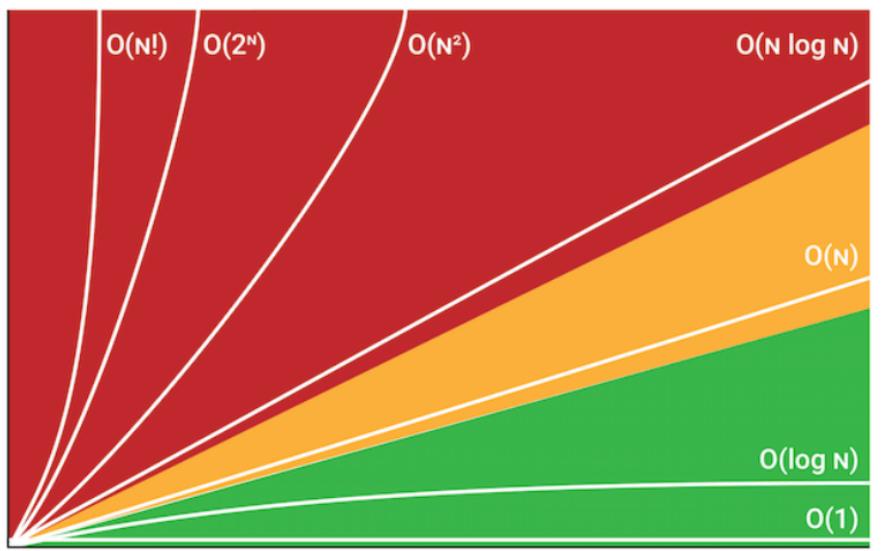
void burbujeo(int array[], int tope) {

    for (int i = 0; i < tope - 1; i++) { //O()
        for (int j = 0; j < tope - i - 1; j++) { //O(n)
            if (array[j] > array[j + 1]) {
                int temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
            }
        }
    }
}

```

Complejidad factorial $O(n!)$: Un algoritmo que siga esta complejidad es un algoritmo totalmente fallido. Una explosión combinatoria se dispara de tal manera

que cuando el conjunto crece un poco, lo normal es que se considere computacionalmente inviable.



C. CALCULAR EL TIEMPO

Para calcular el tiempo se deben determinar todas las instrucciones que se ejecutaron y multiplicarlas por el tiempo de ejecución de una instrucción: *analizar el algoritmo contabilizando las instrucciones*.

Reglas

i. 1. Iteraciones

El tiempo de ejecución de 1 iteración es como máximo el tiempo de ejecución de las instrucciones dentro de la iteración por el número de iteraciones.

$$O(k*n) = k(O(n)) = O(n) \text{ donde } k \text{ es el N° de instrucciones.}$$

ii. 2. Iteraciones Anidadas

Se analizan de dentro hacia afuera. El tiempo total es el tiempo de ejecución de la instrucción multiplicado por la cantidad de iteraciones de cada iteración.

$$O(k*n)O(cn)=k(O(n))*c(O(n))=(c+k)O(n^2)=O(n^2)$$

iii. 3. Instrucciones Consecutivas

El tiempo de ejecución es el máximo entre ambas instrucciones.

```
for ( i =0; i < n ; i ++ ) // O(n)
    w++;
    for ( i =0; i < n ; i ++ )      //O(n^2) --> máximo
        for ( j =0; j < n ; j ++ )
            k++;
```

Relaciones de Recurrencia

Expresar mis funciones en una conocida → lo uso para los algoritmos recursivos

Caso del factorial: no es directo. Pero yo sé que se va a llamar a sí mismo n veces (voy desde n hasta 0, restando de a 1). Puedo decir que la complejidad es $O(n)$

```
long factorial ( int n ) {  
    if ( n == 0 )  
        return 1;  
    else  
        factorial = n * factorial (n -1) ;  
    //n*(n-1)*(n-2)....*2*1 ->n veces  
}
```

Ecuaciones de recurrencia las busco plantear como me dice el "teorema maestro" (chan chan CHHannnn)

D. TEOREMA MAESTRO

Es una forma de resolver ecuaciones de recurrencia de la forma:

$T(n)=aT(n/b)+f(n)$ Donde $a \geq 1$ y $b > 1$, $f(n)$ es positiva

Consideraciones

1. Describe tiempos de ejecución de algoritmos que dividen un problema de tamaño n en subproblemas de tamaño n/b . "**Divide y conquistas**"
 1. Me quedo con fracciones de mi problema hasta llegar a una fácil de resolver. A cada fracción le aplico el mismo algoritmo!
 2. Cada subproblema es resuelto en un tiempo $T(n/b)$
 3. La función $f(n)$ abarca el costo de dividir el problema y combinarlo

Información

- n : **tamaño** del problema
- a : **cantidad de llamadas recursivas** que realiza el algoritmo (en la búsqueda binaria llamo 2 veces, pero SE EJECUTA SOLO 1, ya que voy para un lado o para el otro)
- b : en **cuantas partes se divide el problema** con cada llamada recursiva
- $f(n)$: es el costo en tiempo de lo que cuesta dividir y combinar, es decir, **todo lo que se realiza fuera de las llamadas recursivas**. Siempre se compara con $n^{\log_{ba}}$.

Resolución

Para resolver la ecuación tengo que calcular la complejidad de la parte recursiva del algoritmo, $n^{\log_b a}$, y compararla con la parte no recursiva de la siguiente manera:

$$\begin{cases} f(n) \text{ es menor que } n^{\log_b a} \text{ entonces } T(n) = \Theta(n^{\log_b a}). \\ f(n) \text{ es igual que } n^{\log_b a} \text{ entonces } T(n) = \Theta(n^{\log_b a} \lg n) \\ f(n) \text{ es polinómicamente mayor que } n^{\log_b a}, \text{ entonces } T(n) = \Theta(f(n)). \end{cases}$$

Ejemplo

Por ejemplo, sea el tiempo de ejecución de un algoritmo dado:

$$T(n) = 9T(n/3) + n$$

1. Identifico los valores de $a=9$, $b=3$ y $f(n)=n$
2. Ahora se calcula $n^{\log_b a} = n^{\log_3 9} = n^2$
3. Ahora miro la tabla $n < n^2 \rightarrow T(n) = O(n^2)$

4. RECURSIVIDAD + BACKTRACKING

A. Definición

La recursividad es una característica de ciertos problemas los cuales son resolubles mediante una nueva instancia del mismo problema.

Una función es recursiva si en el cuerpo de la función hay una llamada a sí misma.
Se contiene a si mismo en una versión más pequeña.

B. Recursividad Directa

Llamar a la función dentro de su ámbito. Se utiliza el stack de ejecución → parte del programa donde se mantiene la información sobre la cual la función está siendo ejecutada (sus variables y parámetros).

Se llama a si misma tantas veces sea necesario o lo permita la condición de corte.

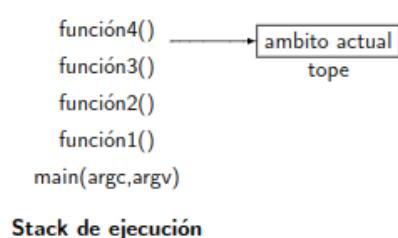
Condiciones de una función recursiva:

- Posee condición de corte
- Posee una llamada recursiva (se llama a si misma)

```

    llamada recursiva
    ↗   ↘
    ↗   ↘
1 long factorial (int un_numero){
2     if (un_numero > 0 )
3         return un_numero * factorial(un_numero-1);
4     else
5         return 1; <---- condición de corte
6     }
7

```

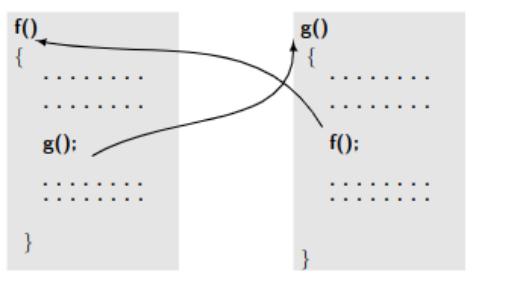


El ámbito de una función es una porción de memoria que se crea cuando la función es llamada y se destruye cuando la función termina su ejecución. En el stack se almacenan:

1. Argumentos pasados a la función
2. Variables locales declaradas en la función.

C. Recursividad Indirecta

Se invoca indirectamente a través de otra función. Una función f llama a una función g, la función g llama a la función f y así



D. Recursividad de Cola

Técnica para optimizar la recursividad eliminando las constantes llamadas recursivas.

Tail recursion se da cuando la llamada recursiva es la última instrucción de la función. Se evita la sobrecarga de cada llamada a la función y se evita el gasto de memoria de pila: **evitar el stack overflow** (los compiladores realizan esta optimización).

En cada llamado se pasa por parámetro el subtotal del cálculo que se está resolviendo

```
#include <stdio.h>
2
3 unsigned factorial_rec(unsigned n, unsigned parcial ){
4     if(n==0)
5         return parcial;
6
7     return factorial_rec(n-1, parcial*n );
8 }
9
10 unsigned factorial(unsigned n){
11     return factorial_rec(n,1);
12 }
13
14 int main(){
15     unsigned resultado=0;
16     unsigned n=15;
17     resultado = factorial(n);
18     printf("%u\n", resultado);
19 }
```

Podemos calcular sin esperar una llamada a una función recursiva para que nos devuelva un valor. En el ejemplo, el valor de n y parcial es independiente del número de llamadas recursivas.

Una función recursiva normal se puede convertir a tail recursive usando en la función original un parámetro adicional para ir guardando el resultado de tal manera que la llamada recursiva no tenga una operación pendiente.

Consideraciones:

Una llamada es tail recursive (recursiva por cola) si no tiene que hacer nada más después de la llamada de retorno.

La llamada recursiva es la última instrucción en la función.

La cantidad de información que debe ser almacenada durante el cálculo es independiente del número de llamadas recursivas.

E. DIVIDE Y CONQUISTA

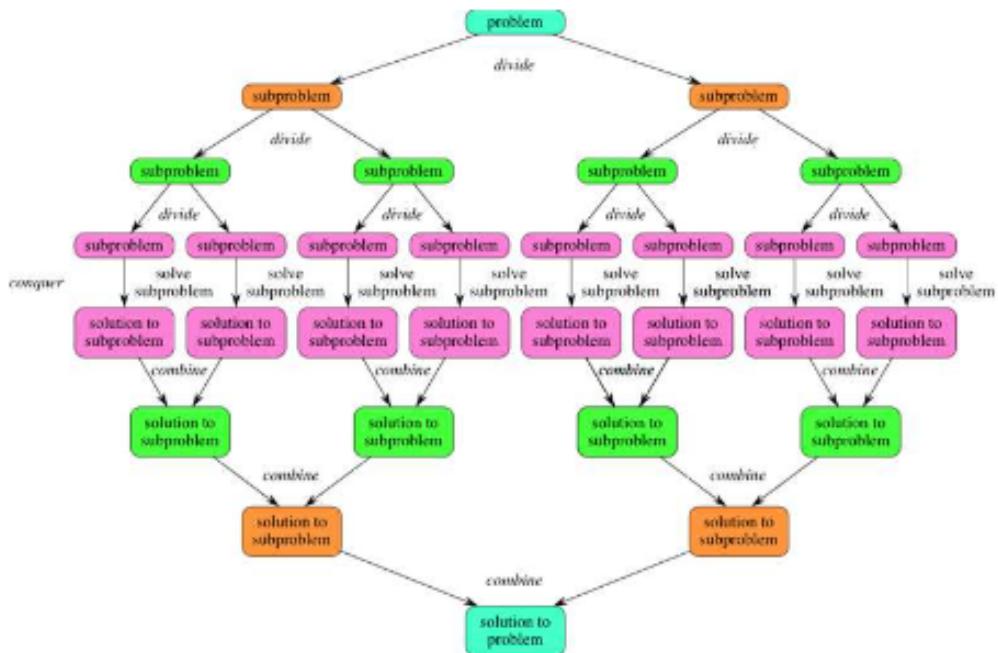
En el caso de los algoritmos, es una metodología para diseñarlos.

Dividir: el problema en un número de subproblemas que sean instancias menores del mismo problema.

Conquistar: los subproblemas para que sean resueltos recursivamente, si el tamaño del subproblema es lo suficientemente pequeño la solución es simple. Como el tamaño de los subproblemas es estrictamente menor que el tamaño original → existencia caso base

Combinar: las soluciones obtenidas de cada subproblema en la solución del problema original.

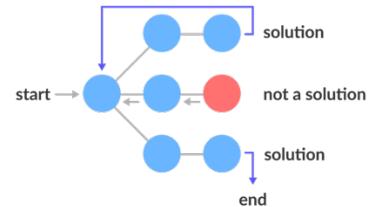
La solución de cada subproblema debe ser independiente.



F. Backtracking

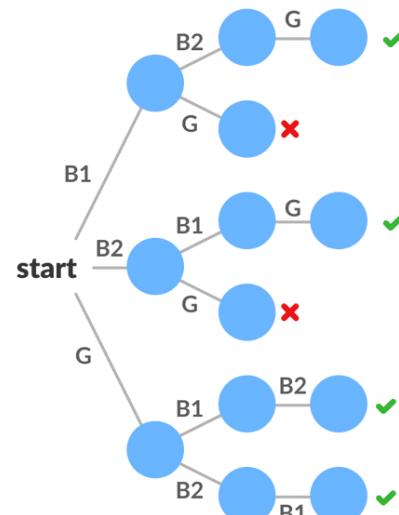
Un algoritmo de retroceso o backtracking es un algoritmo de resolución de problemas que utiliza un enfoque de fuerza bruta para encontrar el resultado deseado. El enfoque de la fuerza bruta **prueba todas las soluciones posibles** y elige las mejores/deseadas. El término retroceso sugiere que **si la solución actual no es adecuada, retroceda y pruebe otras soluciones**. Por lo tanto, en este enfoque se utiliza la *recursividad*.

Un árbol de estado espacial es un árbol que representa todos los estados posibles (solución o no solución) del problema desde la raíz como estado inicial hasta la hoja como estado terminal.



Por ejemplo: deseamos encontrar todas las formas posibles de colocar 2 chicos (B1 y B2) y 1 chica (G) en 3 bancos, pero la niña no debe estar en el banco del medio. Hay un total de $3! = 6$ posibilidades. Intentamos todas las posibilidades y obtenemos las posibles soluciones que son:

B1 B2 G	B2 G B1
B1 G B2	G B1 B2
B2 B1 G	G B2 B1



El árbol a la izquierda muestra las posibles soluciones.

5. TDA

A. Introducción

La abstracción es un proceso por el cual se separa a un problema o cosa de la complejidad que no es relevante para la solución de un problema.

Es un mecanismo que permite la expresión de los detalles relevantes y la supresión de los irrelevantes.

- La abstracción

Definición: separar aisladamente en la mente las características de un objeto o un hecho, dejando de prestar atención al mundo sensible para enfocarse solo en el pensamiento.

NO ES EL PORQUE ES EL PARA QUE.

- Los tipos de datos

Un tipo de dato está definido por:

El conjunto de todos los valores posibles que una variable de ese tipo puede tomar.

Las operaciones que las variables de ese tipo de dato pueden utilizar.

Un tipo de dato es primitivo cuando no se especifica información extra para definir una variable de este tipo o sobre las operaciones que pueden hacerse sobre las variables. Por ejemplo: *int*

Tipo	Conjunto de Valores	Operador	Operación	Resultado
int	-2^{31} y $(2^{31} - 1)$ implementado como complemento a 2	+ (suma) - (resta) * (multiplicación) / (división entera) % (resto)	$5 + 3$ $5 - 3$ $5 * 3$ $5 / 3$ $5 \% 3$	8 2 15 1 2

- Definición:

Un TDA define una clase de objetos abstractos los cuales están completamente caracterizados por las operaciones que pueden realizarse sobre esos objetos. Puede ser definido describiendo las operaciones características para ese tipo.

B. Un que y mil comos

- El qué: concepto de funcionalidad *¿qué hace esto?* **FUNCIONALIDAD**
 - La visión del usuario, que haga lo que dice que hace
- El como: forma en que algo está diseñado o implementado. La estructura interna o la forma en la cual la funcionalidad de algo es llevada a cabo *¿cómo lo hago?* **DISEÑO E IMPLEMENTACIÓN**
 - Visión del implementador → basarse en la forma que se diseña e implementa el tda.

El Qué y el Cómo

El Qué

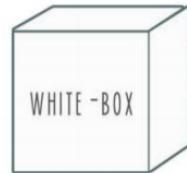
- Acá hablamos de las operaciones que podemos hacer con algo.
- Hablamos de funcionalidad, de ahí la pregunta: *¿Qué* hace esto?
- Hablamos de **caja negra**.



ZERO KNOWLEDGE

El Cómo

- Acá hablamos de la forma en que algo está diseñado o implementado.
- Nos interesa la estructura interna o la forma en la cual se lleva a cabo algo.
- La pregunta es: *¿Cómo* lo hago?
- Hablamos de **caja blanca**.



FULL KNOWLEDGE

Ambas partes (usuario e implementador) deben **cumplir el contrato**, el mismo está implícitamente aceptado en el archivo de cabecera (tda.h) del tda. Una vez el tda está implementado, puede ser usado por terceras partes → únicamente proporciona al usuario del tda el archivo de cabecera y el archivo objeto (no le importa al usuario como lo hice).

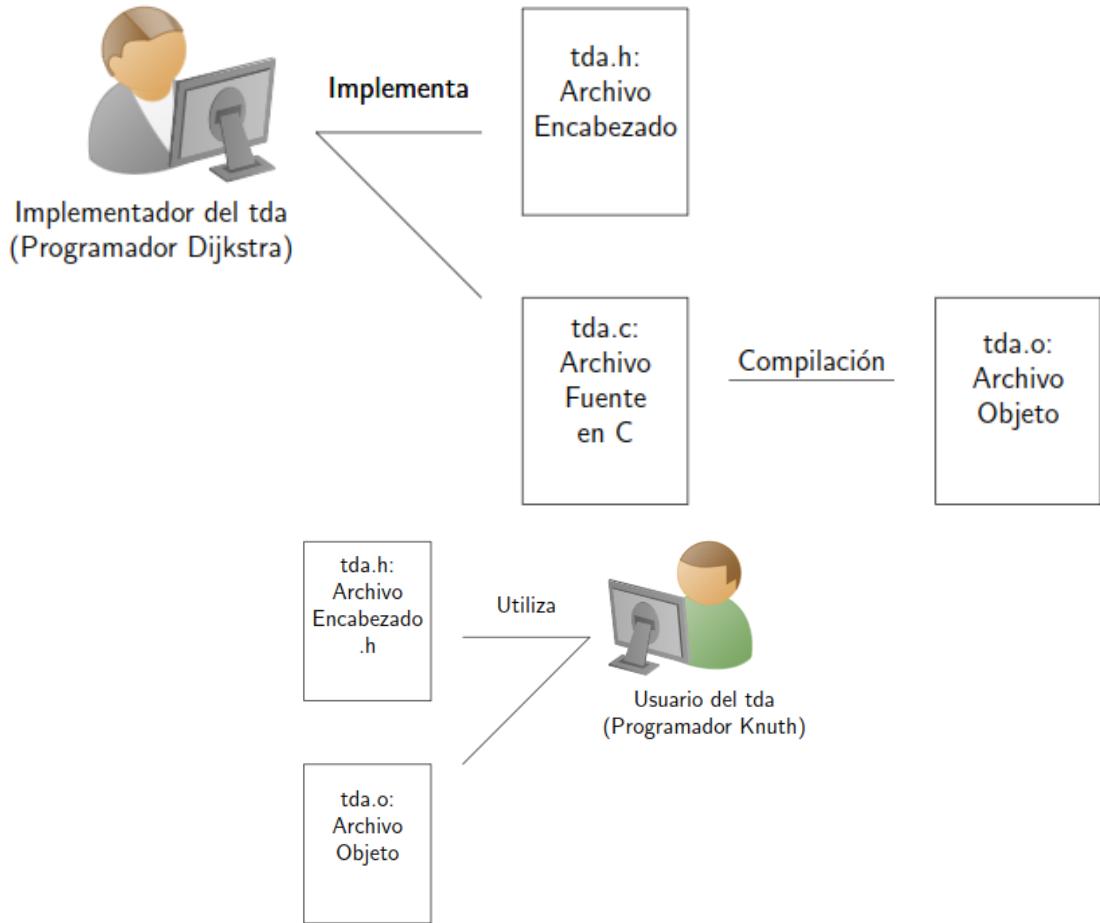
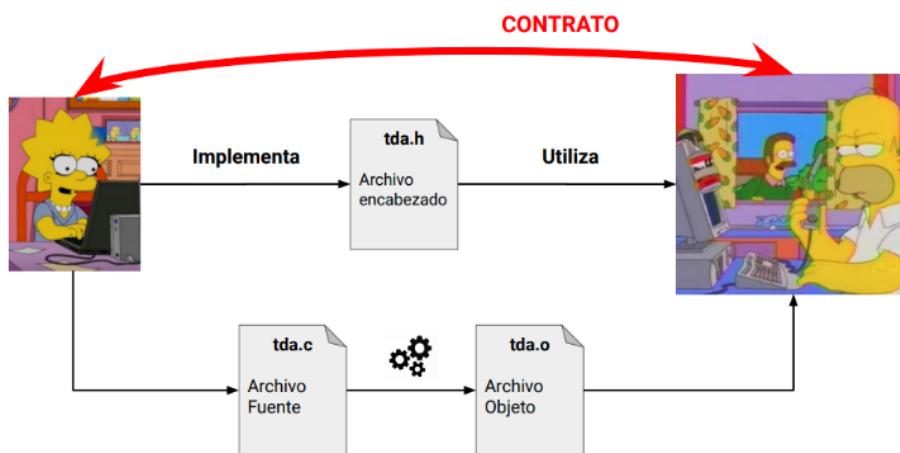


Figura 6: Utilización del tda

Existe un contrato tácito entre el implementador del tda y sus usuarios. El implementador debe garantizar que la funcionalidad expuesta en el archivo de cabecera se cumpla. La utilización de **Pre-Post** condiciones es **fundamental** para este fin.



C. Ventajas

- **Abstracción:** La abstracción permite simplificar la realidad mediante el despojo de complejidad que no es propio del problema que estamos resolviendo.
- **Encapsulamiento:** Un TDA debe exponer la menor cantidad posible de información del como esta implementado, haciendo que el usuario se base en las funciones que él mismo entiende. El implementador debe ocultar la mayor cantidad de detalles sobre la implementación y diseño del mismo.
- **Localización del cambio:** Cuando existen errores dentro de un programa, es más fácil detectarlo, pues la utilización de los TDAs fuerza la modularización.

D. Proceso de desarrollo de software

Los programas y las bibliotecas que se construyen en lenguaje C forman parte del concepto de Software.

El software se **desarrolla**. Incluye a un conjunto de etapas, muy parecidas a las utilizadas para la resolución de problemas computacionales.



El conjunto debe comprender las siguientes tareas:

1. **Análisis:** se pone foco en la comprensión del problema, determinar cuales son aquellas cosas que se requieren para la resolución del mismo. **¿Qué** es lo que hay que hacer?
2. **Diseño:** crear una solución teniendo en cuenta múltiples aspectos del software. Pone en interés el **Cómo**. Relacionado con el problema, el usuario y la solución.
3. **Implementación:** se desarrolla el software en base al análisis y diseño realizado.
4. **Pruebas:** prueba del sistema y la integración (pruebas de funcionalidad, integración de sistemas, de aceptación del usuario) para garantizar que el código esté limpio y se cumplan los objetivos comerciales de la solución. La **verificación y validación constituyen una parte vital** para garantizar que la aplicación / solución se complete con éxito.
5. **Instalación:** Esta es una etapa en la que tiene lugar la instalación real de la solución diseñada.
6. **Mantenimiento:** como el software es producto que una vez acabado sigue siendo susceptible al cambio → arreglas. mejoras o reparación.

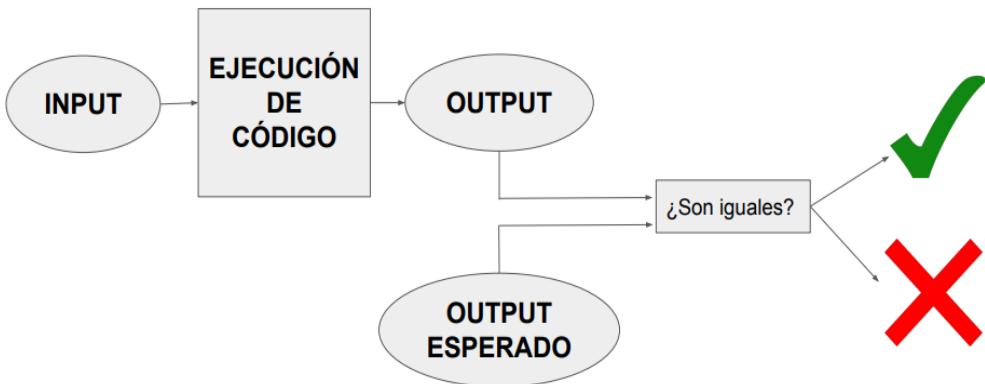
6. TESTING

A. Definición

Test: “Un procedimiento destinado a determinar la calidad, rendimiento o confiabilidad de algo, especialmente antes que sea llevado al uso masivo”

Las pruebas de software pueden definirse como el **proceso de verificación y validación** de que un software o una aplicación está libre de errores.

El proceso de prueba de software tiene como objetivo no solo encontrar fallas en el software existente, sino también encontrar medidas para mejorar el software en términos de eficiencia, precisión y usabilidad. Su objetivo principal es medir la especificación, la funcionalidad y el rendimiento de un programa o aplicación de software.



Importancia del testing

1. Software está en todo
2. Software es algo que está en constante cambio:
 - Clientes piden más
 - Necesidad de adaptarse a cambios del entorno
 - Necesidad de solucionar bugs

Beneficios (a largo plazo)

- Verificación de funcionalidad
 - Identificación de bugs
 - Comunicación entre módulos
- Red de seguridad ante cambios
- Cuando tengo un error → fácil de encontrar

Desventajas de su ausencia

- Aumento en el costo de mantenimiento
 - Costo corrección de bugs
 - Costo agregar nuevas funcionalidades

B. Tipos de pruebas de software

Las pruebas de software se pueden clasificar en:

1. **Prueba manual:** probar el software manualmente. El evaluador *asume el rol de usuario final* y prueba el software para identificar cualquier comportamiento o error inesperado.

1. **Pruebas unitarias:** se pone a prueba una única función o una pequeña funcionalidad. Sin pequeños indicadores del funcionamiento del código.
 2. **Pruebas de integración:** se pone a prueba la interacción entre varios módulos. Indican como se comporta la aplicación en conjunto → exponer fallas en la interacción entre las unidades.
 3. **Prueba del sistema:** un nivel del proceso de prueba de software en el que se prueba un sistema/software completo e integrado. El propósito de esta prueba es evaluar el cumplimiento del sistema con los requisitos especificados.
 4. **Prueba de aceptación:** nivel del proceso de prueba de software en el que se prueba la aceptabilidad de un sistema. El propósito de esta prueba es evaluar el cumplimiento del sistema con los requisitos comerciales y evaluar si es aceptable para la entrega.
2. **Pruebas de automatización:** las pruebas se escriben en código para que se ejecuten de forma automática. Se usa otro software para probar el producto. Se utilizan para probar la aplicación desde el punto de vista de la carga, el rendimiento y el estrés. Aumenta la cobertura de la prueba, mejora la precisión y ahorra tiempo y dinero en comparación con las pruebas manuales.

C. Técnicas de pruebas de software

Las técnicas de software se pueden clasificar principalmente en dos categorías:

1. **Prueba de caja negra:** la técnica de prueba en la que el evaluador no tiene acceso al código fuente del software y se realiza en la interfaz del software *sin preocuparse por la estructura lógica interna del software* se conoce como prueba de caja negra.
2. **Prueba de caja blanca:** la técnica de prueba en la que el probador es consciente del funcionamiento interno del producto, *tiene acceso a su código fuente* y se realiza asegurándose de que todas las operaciones internas se realizan de acuerdo con las especificaciones se conoce como prueba de caja blanca.

D. Estructura

Mientras hago cada función, hago un test de la misma a la par → ahí voy viendo como va fallando mi función y si cumple los casos borde.

Priorizo la legibilidad del código → que las pruebas las entienda y no estar modularizando todo (pruebas fáciles de entender)

1. **Inicialización:** creo lo que necesito
2. **Afirmación:** 1 o más afirmaciones que me permiten conocer el funcionamiento
3. **Destrucción**

GIVEN / DADO	WHEN / CUANDO	THEN / ENTONCES
<pre> 44 void DadaUnAprendizConVisionesPasadasYFuturas_CuandoSePidenLasVisionesFuturas_EntoncesSeDevuelveUnConjuntoDeVisionesFuturas() { 45 46 //Arrange / Organizar 47 cuervo_aprendiz_t aprendiz; 48 aprendiz.cantidad_visiones = 3; 49 aprendiz.visiones_adquiridas = (vision_t*) malloc(sizeof(vision_t) * aprendiz.cantidad_visiones); 50 aprendiz.visiones_adquiridas[0].epocha = FUTURO; 51 aprendiz.visiones_adquiridas[1].epocha = PASADO; 52 aprendiz.visiones_adquiridas[2].epocha = FUTURO; 53 54 //Act / Actuar 55 conjunto_visiones_t visiones_futuras = obtener_visiones_futuras(&aprendiz); 56 57 //Assert / Verificar 58 vision_t* visiones Esperadas[] = {aprendiz.visiones_adquiridas, 59 aprendiz.visiones_adquiridas + 2}; 60 ASSERT_EQUALS("Hay dos visiones futuras", 2, visiones_futuras.cantidad_visiones); 61 ASSERT_VECTOR_EQUALS("Los dos vectores son iguales", visiones Esperadas, visiones_futuras.visiones, visiones_futuras.cantidad_visiones); 62 } 63 </pre>		

7. TDAS

A. INTRODUCCION

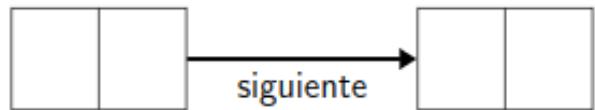
La abstracción permite que a partir de la combinación de ciertas herramientas primitivas que ofrecen los lenguajes de programación, crear nuevos tipos de datos:

- Tipos primitivos: numéricos, caracteres, booleanos.
- Funciones: permiten generar y encapsular nuevas acciones o funcionalidades sobre los datos.
- Bibliotecas: agrupan conceptualmente acciones y datos para generar nuevos tipos de datos.
- Caja negra: técnica que permite centrarse en ¿Qué? realiza cierta función y no en el ¿Cómo?.

Para cada tda ¡no existe una única implementación!

B. TDA NODO ENLAZADO

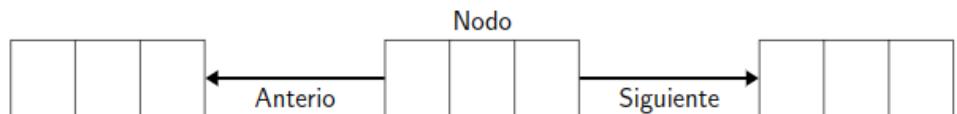
Abstira la idea de un contenedor → puede conocer a quien le sucede o antecede.



```

typedef struct nodo {
    dato_t elemento;
    nodo_t* siguiente;
} nodo_t;

```



```

typedef struct nodo {
    dato_t elemento;
    nodo_t* siguiente;
    nodo_t* anterior;
} nodo_t;

```

C. TDA PILA

Una pila es una colección ordenada de elementos que pueden insertarse y eliminarse por un extremo," denominado tope.

Pienso como una pila de platos → stack del programa.

⚠ "LIFO: Last in, first out" (ultimo en entrar, primero en salir)

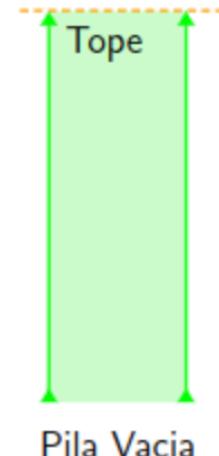
Conjunto mínimo de operaciones

Recordar: yo solo tengo referencia al último elemento de la pila → TOPE

- Crear

Se realizan operaciones que tienen que ver con el tamaño de la pila, es decir, la cantidad de elemento que la misma podrá albergar.

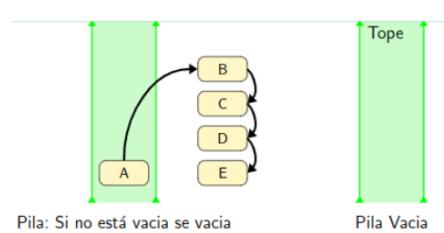
Complejidad: O(1)



- Destruir

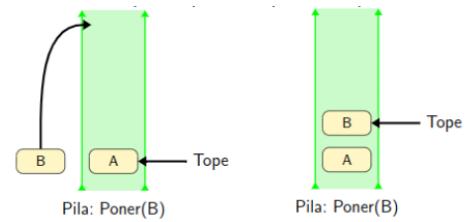
Se ocupa de liberar y limpiar todos los recursos que se utilizan para la creación de una pila. Si la misma posee elementos en su interior debe vaciarse en el proceso de destrucción.

Complejidad: O(n)



- Poner

Pone un elemento en la pila por el tope de la misma, haciendo el que tope pase a ser el nuevo elemento introducido. **Recordar cambiar el valor del tope.**



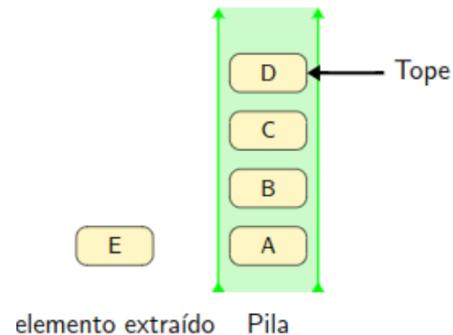
Complejidad: O(1)

- Tope: Observa el valor del tope de la pila. *En el caso anterior: valor del tope es B*

- Desapilar

Retira el elemento del tope de la pila y mueve el tope de la pila al elemento anterior al extraído, si el elemento extraído es el último la pila queda vacía.

Complejidad: O(1)



- Esta vacia:

Determina si una pila tiene o no elementos. Devuelve el estado de la pila como:

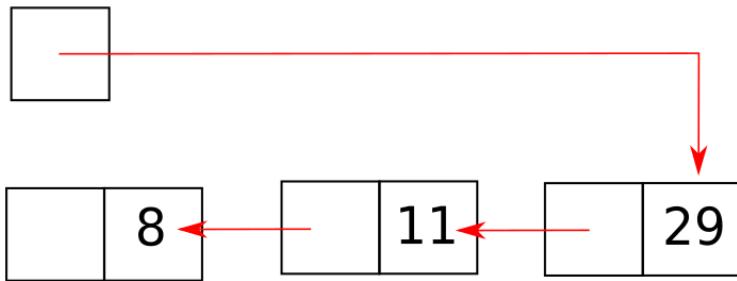
- Verdadero: pila no posee elementos apilados
- Falso: si posee elemento en su interior.

Puedo vaciar una pila sin la necesidad de saber cuantos elementos están apilados en su interior:

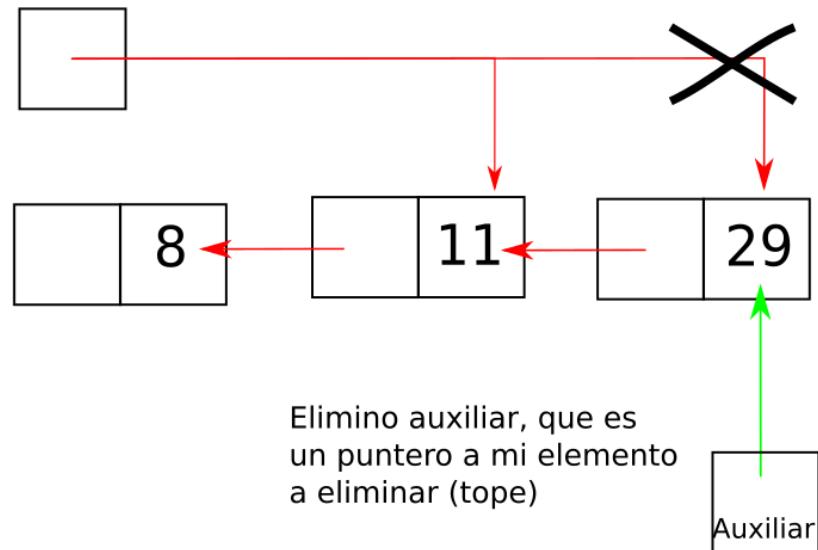
```
while(!esta_vacia(pila){
    sacar();
}
```

Nodo enlazado en pila

Únicamente puedo acceder al tope (al 29).

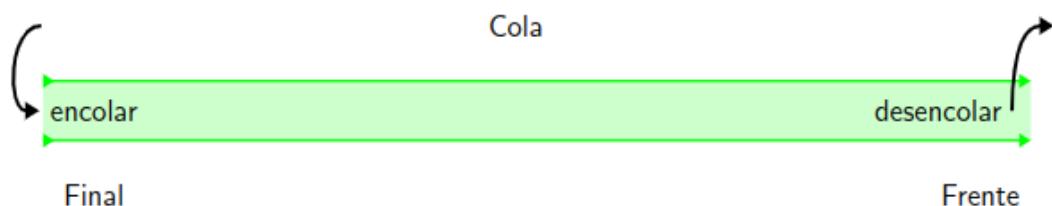


Cuando tenga que eliminar un elemento, me tengo que crear un nodo auxiliar así no pierdo referencia



D. TDA COLA

Posee dos extremos por los que se realizan operaciones. Un extremo es el inicio (frente) de la cola y el otro extremo es el final de la cola. *Ídem a la cola de un colectivo.*

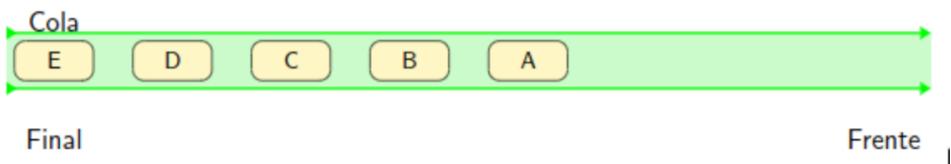


⚠ "FIFO: First in, First out" primero en entrar, primero en salir

Conjunto mínimo de operaciones

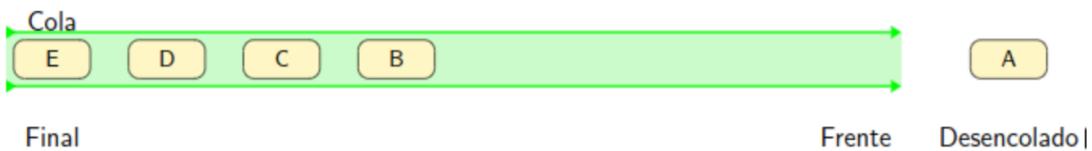
- **Encolar**

Los elementos se encolan por el final de la sola. La idea es la misma que la cola de supermercado, uno llega y se para en la final.



- **Desencolar**

Extrae del frente de la cola el elemento que se encuentra en el.



- **Primer**

Saber cual es el primer elemento del frente de la cola.



- **Esta vacia**

Determinar si la cola tiene o no elementos. Devuelve el estado de la cola como:

- Verdadero: si la cola no posee elementos encolados
- Falso: si la cola posee elementos en su interior encolados.

Puedo hacer lo mismo que la pila → vaciarla mientras no este vacía.

- **Crear**

Se realizan operaciones que tiene que ver con el tamaño de la cola, → cantidad de elemento que la misma podrá albergar

- **Destruir**

Se ocupa de liberar y limpiar todos los recursos que se utilizan para la creación de una cola. Si la misma posee elementos en su interior debe vaciarse en el proceso de destrucción.

1. La cola tiene elementos
2. Se deben eliminar cada uno de los elementos (vaciárla)
3. Una vez no posee elementos se destruye liberando toda la memoria que la estructura ocupa

Nodos enlazados

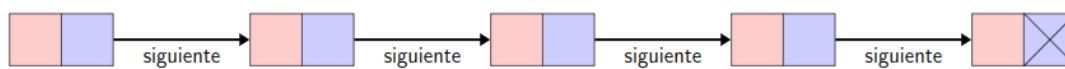
Crear:

```
nodo_t* principio; /*inicializar ambos elemementos a NULL, IMPORTANTE*/  
nodo_t* final;
```

E. TDA LISTA

Este tipo de dato está basado en los nodos enlazados. La idea reside en que un nodo al poder conocer su siguiente, puede crear una lista de nodos que termina cuando el último elemento apunta a NULL.

- Puedo meter un nodo en cualquier lugar
- Puedo ver todos sus elementos

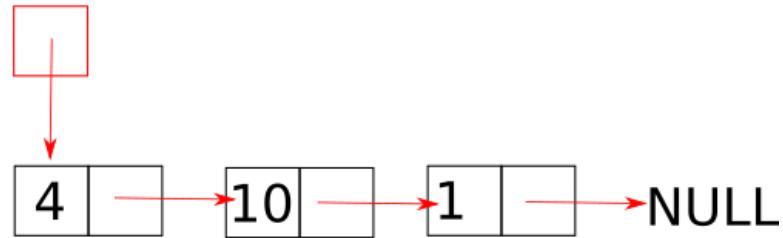


Listas simplemente enlazadas

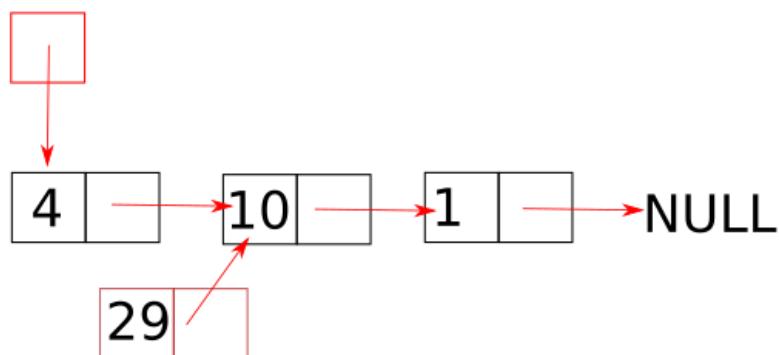
Cada nodo conoce al nodo siguiente.

Conjunto mínimo de operaciones

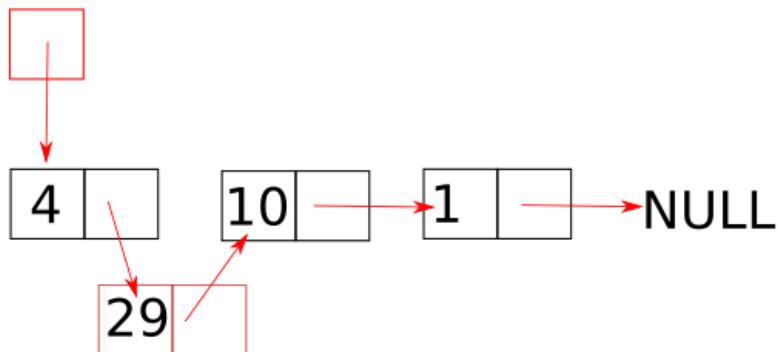
- **Crear:** Se inicializa la estructura lista_t, seteando todos sus campos en valores válidos. De realizarse su implementación con memoria dinámica se debe reservar la cantidad de memoria dinámica en el heap y posteriormente devolver la referencia a esa memoria reservada.
- **Insertar:** Esta operación inserta un elemento en la posición indicada, donde 0 es insertar como primer elemento y 1 es insertar luego del primer elemento y así... De no existir la posición indicada, lo inserta al final. Devuelve 0 si pudo insertar y -1 si no pudo.



Quiero insertar el 29, por lo que primero hago que el 29 apunte al 10

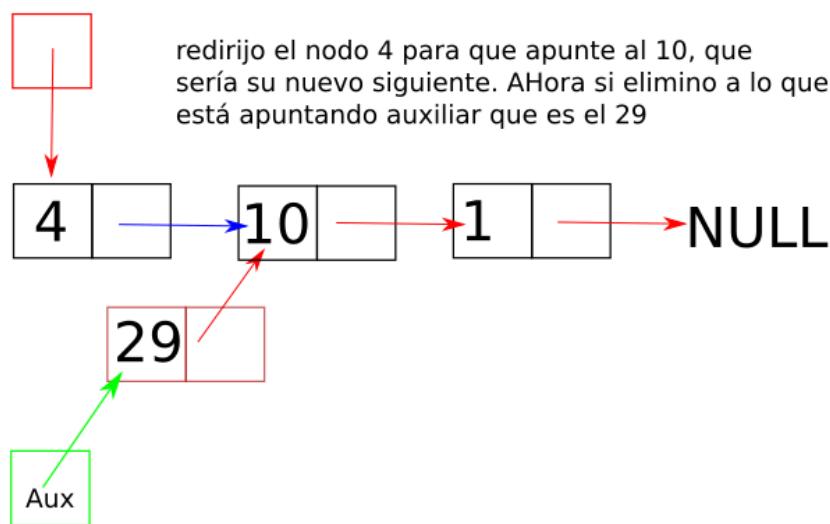
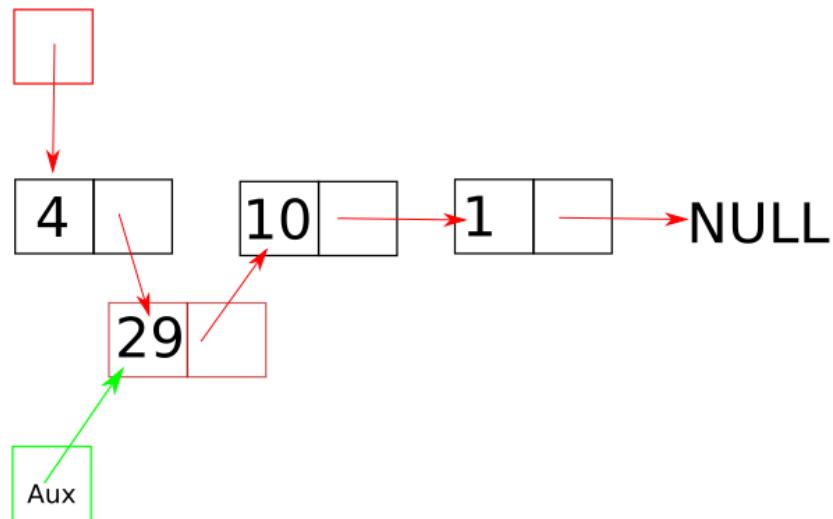


Luego hago que el 4 apunte al 29



- **Eliminar:** Esta operación quita de la lista el elemento que se encuentra en la posición indicada, donde 0 es el primer elemento. En caso de no existir esa posición se intentará borrar el último elemento. Devuelve 0 si pudo eliminar o -1 si no pudo.

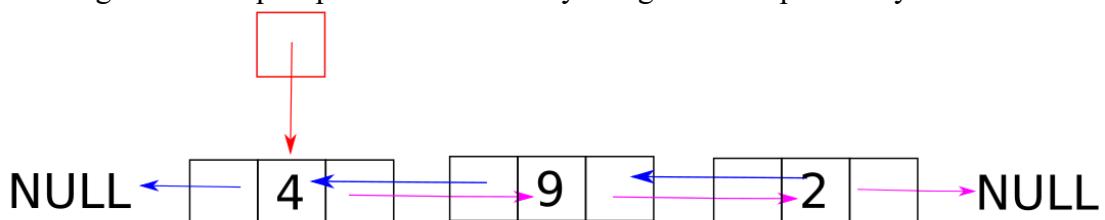
quiero eliminar el 29, debo crear un auxiliar que apunte al 29



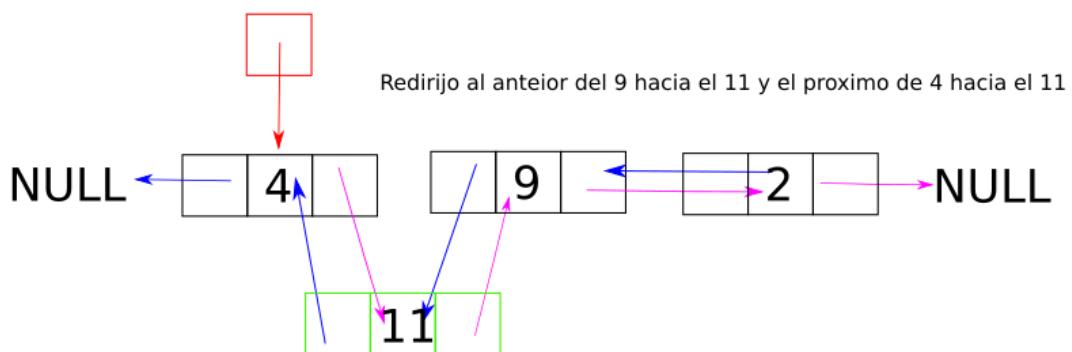
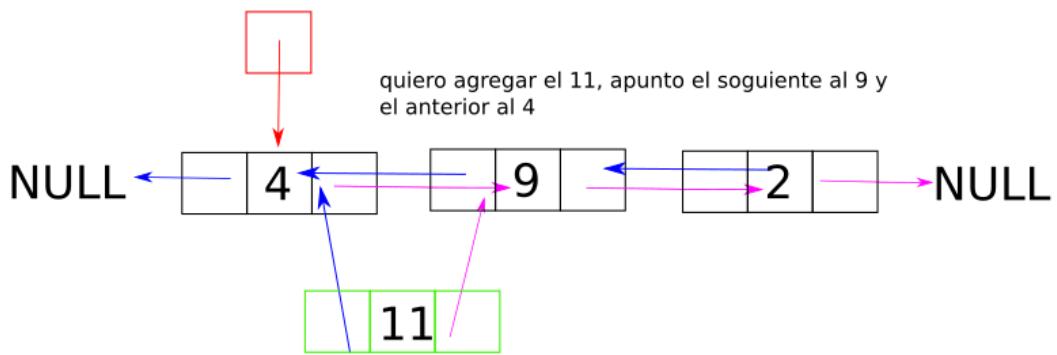
- **Elemento en posición:** Esta operación devuelve el elemento en la posición indicada, donde 0 es el primer elemento. Si no existe dicha posición devuelve NULL.
- **Destruir:** Libera la memoria reservada por la lista
- **Vacia:** Devuelve true si la lista esta vacía o false en caso contrario

Lista doblemente enlazada

Tengo un nodo que apunta a su anterior y al siguiente → puedo ir y volver

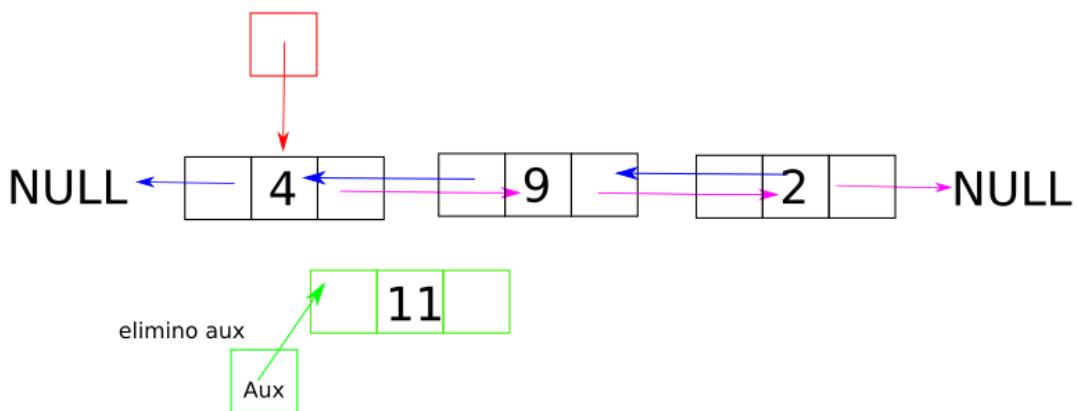


Agregar



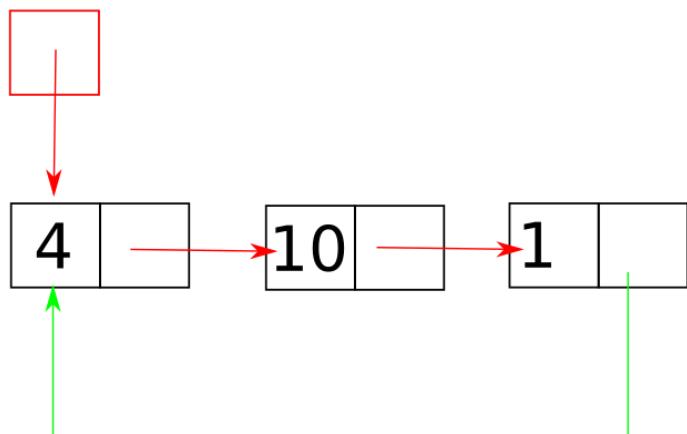
Eliminar

- Hago un auxiliar a lo que quiero eliminar (apunta al 11)
- Referencia el anterior del 11 al próximo del 11 (de 4 a 9)
- Referencia el próximo del 11 (el 9) al anterior del 11 (el 4)

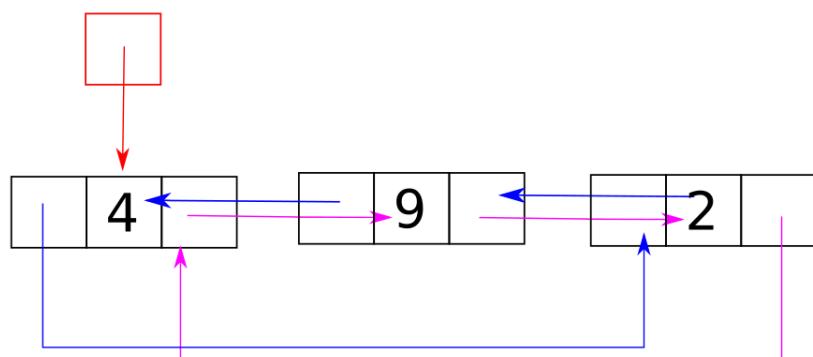


Lista circular simplemente enlazada

El último apunta al primero (en lugar de que apunte a NULL) → **tengo que tener un conteo de los elementos que tengo** para no quedarme en un loop infinito.



Lista circular doblemente enlazada



8. ARBOLES

A. INTRODUCCION

Un árbol es una colección de nodos, que puede estar conectado a múltiples nodos. Un árbol consiste de un nodo principal r, llamado raíz, y cero o muchos subárboles no vacíos, cada uno de ellos con su raíz conectada mediante un vértice al nodo raíz.

Este tipo de estructura es capaz de reducir el tiempo de acceso a los datos
→ $O(\log N)$ en promedio.

Nacen de la necesidad de presentar una jerarquía en la estructura de datos y optimizar la búsqueda lineal de una lista.

B. ESTRUCTURA

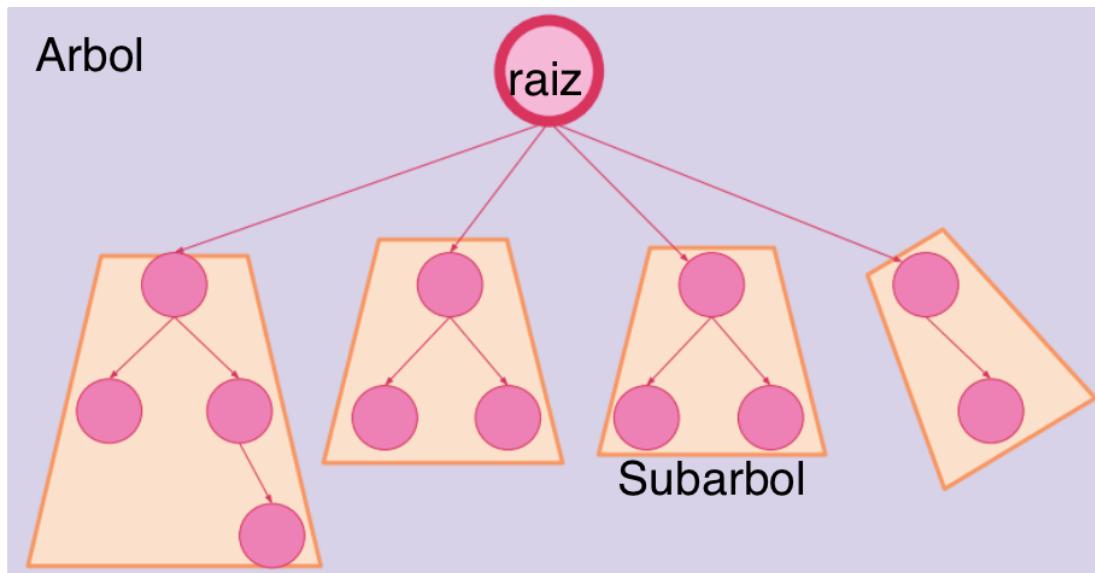
Árbol: colección de nodos.

Nodo: elemento del árbol.

1. Nodo padre: nodo que está inmediatamente superior → cada nodo tiene UN SOLO padre
2. Nodo hijo: elementos que están por debajo. Los nodos conectados en un nivel inferior son los hijos del nodo en el que estoy parado.
3. Nodo hojas: los que no tienen hijos.

Raíz: es el elemento de primer nivel del árbol.

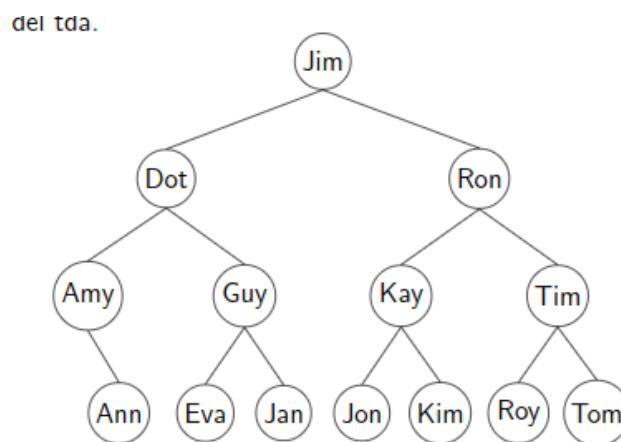
Sub-árbol: parándose desde un nodo puedo ver distintos árboles debajo de él.



C. ARBOL BINARIO

Estos árboles están íntimamente relacionados con las operaciones búsqueda, con el objetivo de aproximarse a la búsqueda binaria.

El nodo raíz está solamente conectado a dos subárboles, lo cual nos permite determinar la noción de izquierda y derecha.



Operaciones

1. Crear

2. Destruir
3. Vacío
4. Insertar
5. Eliminar
6. Buscar
7. Recorrer (operaciones más importantes)

Implementaciones

```
typedef struct nodo_arbol* arbol_binario_t;

struct nodo_arbol{
    void * elemento;
    nodo_arbol * izquierda;
    nodo_arbol * derecha
};
```

Recorridos

Recorrer un árbol significa pasar por cada uno de los nodos. Nombramos:

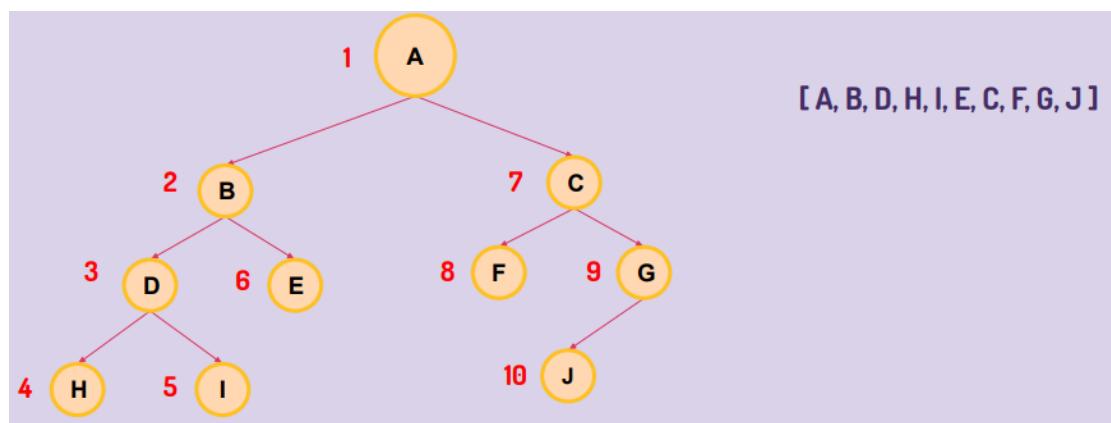
- N: nodo actual,
- D: subárbol derecho
- I: subárbol izquierdo



Preorder

Primero se visita el nodo actual, luego el subárbol izquierdo y luego el derecho

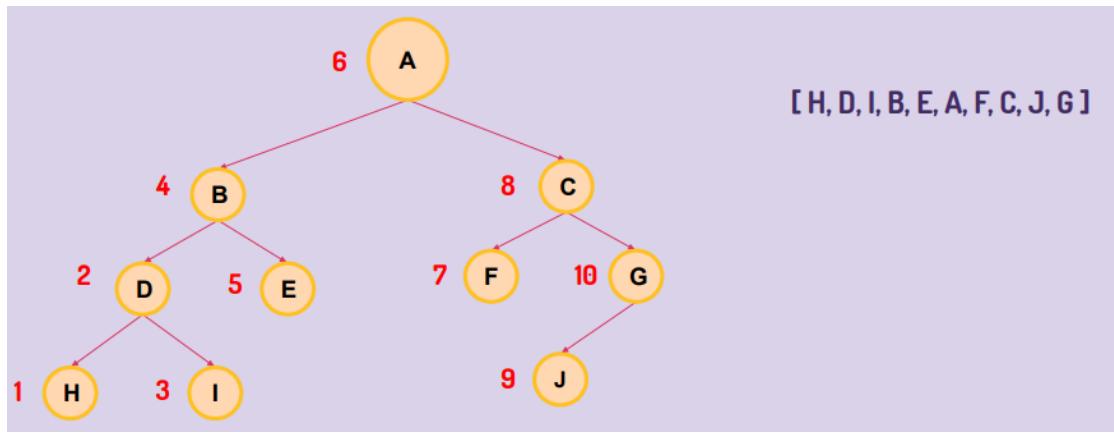
Se suele utilizar: Cuando quiero **clonar** árboles.



Inorder

Primero se visita el subárbol izquierdo, luego el nodo actual y por último el subárbol derecho

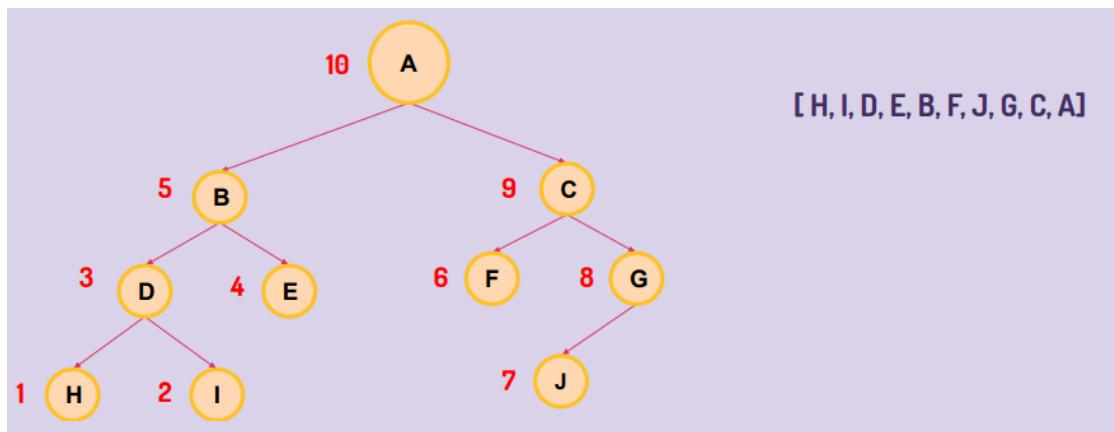
Se suele utilizar: Cuando quiero un recorrido **ordenado** del árbol.



Postorder

Primero se visita el subárbol izquierdo, luego el derecho y por último el nodo actual.

Se suele utilizar: Cuando quiero **destruir** el árbol.



D. ARBOL BINARIO DE BUSQUEDA

Este tipo de dato tiene un orden → forma de agrupar los elementos

Características

1. Las claves **mayores** se insertan en los subárboles **derecho**
2. Las claves **menores** se insertan en los subárboles **izquierdos**
3. Ambos subárboles también son árboles de búsqueda binaria

Operaciones

1. Crear

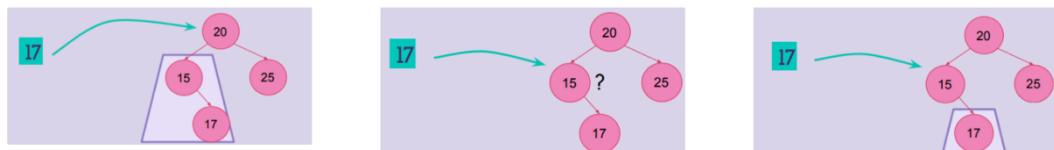
2. Destruir
3. Vacío
4. Insertar
5. Eliminar
6. Buscar
7. Recorrer
8. Complejidades

Operation	Best Case Complexity	Average Case Complexity	Worst Case Complexity
Search	$O(\log n)$	$O(\log n)$	$O(n)$
Insertion	$O(\log n)$	$O(\log n)$	$O(n)$
Deletion	$O(\log n)$	$O(\log n)$	$O(n)$

a) Búsqueda

La búsqueda comienza en el nodo raíz y sigue los siguientes pasos:

1. La clave buscada se compara con la clave del nodo raíz
2. Si las claves son iguales, la búsqueda se detiene
3. Si la clave buscada es mayor que la clave raíz, al búsqueda se reanuda en el subárbol derecho. Si la clave buscada es menor que la clave raíz, la búsqueda se reanuda en el subárbol izquierdo.



Detalle operación

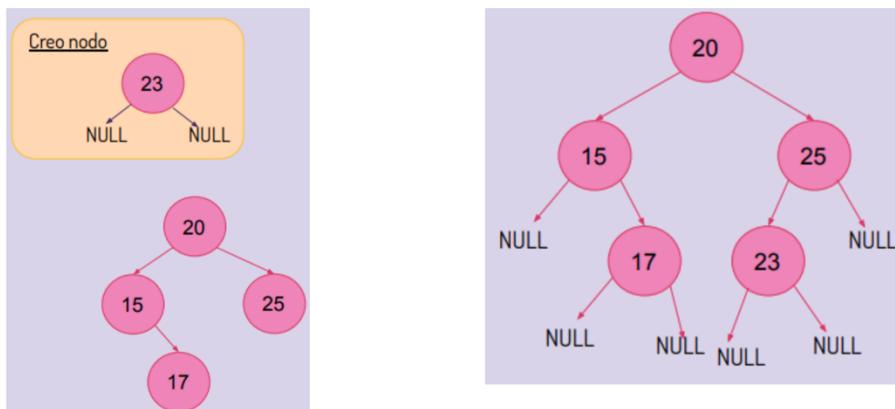
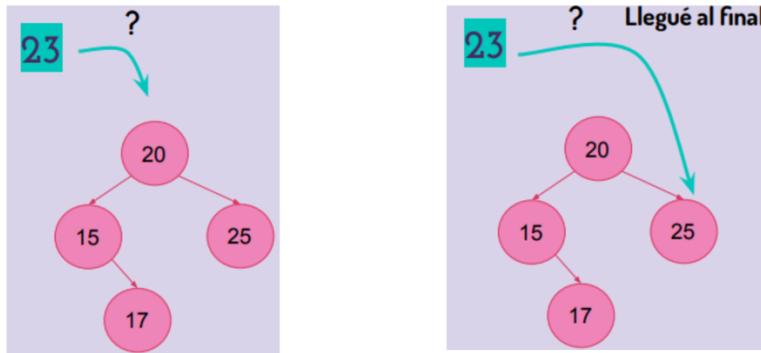
El algoritmo depende de la propiedad de ABB: cada subárbol izquierdo tiene valores por debajo del dato del nodo y cada subárbol derecho tiene valores por encima del dato del nodo. Si el valor está por debajo del nodo, podemos decir con certeza que el valor no está en el subárbol derecho, solo necesitamos buscar en el subárbol izquierdo. Y si el valor está por encima del nodo, el valor no está en el subárbol izquierdo, solo necesitamos buscar en el subárbol derecho.

b) Insertar

Tengo que mantener el orden, por lo que hay un lugar específico para insertarlo.

1. Comparo la clave del elemento a insertar con la clave del nodo raíz. Si es mayor avanco hacia el subárbol derecho, si es menor hacia el izquierdo.

2. Repetir el paso 1 hasta encontrar un elemento con clave igual o llegar al final del subárbol donde debo insertar el nuevo elemento
3. Cuando llego al final, creo un nuevo nodo asignado null a los punteros izquierdo y derecho del mismo. Luego coloco el nuevo nodo como hijo izquierdo o derecho del anterior según sea el valor de la clave.



Elementos repetidos: (Elijo yo que hacer si quiero insertar un elemento que ya está en mi árbol)

1. No insertar → no acepto duplicados
2. Aceptar y lo pongo a la izquierda

Detalle de operación

Insertar un valor en la posición correcta es similar a buscar, porque se mantiene la regla de que el subárbol izquierdo es menor que el dato nodo y el subárbol derecho es mayor que el dato nodo. Seguimos yendo al subárbol derecho o al subárbol izquierdo dependiendo del dato y cuando llegamos a un punto que el subárbol izquierdo o derecho es **nulo**, insertamos el nuevo nodo.

c) Eliminación

Esta operación es una extensión de la operación búsqueda. Me importa el estado en el que queda la estructura ya que tengo que mantener las reglas del TDA.

Hay tres escenarios distintos cuando quiero borrar un nodo:

1. Que sea un nodo hoja (no tiene hijos)
2. Que tenga un único hijo
3. Que tenga dos hijos.

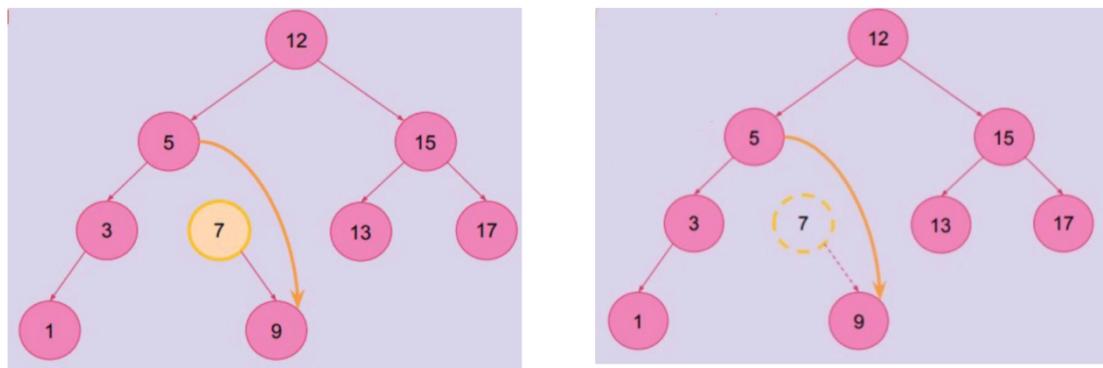
Nodo hoja (caso feliz)

1. Bajo por el árbol buscando el nodo a borrar
2. Al no tener hijos, lo borro tranquilamente y le asigno NULL al puntero de su antecesor que antes apuntaba al puntero eliminado.

Nodo con un hijo

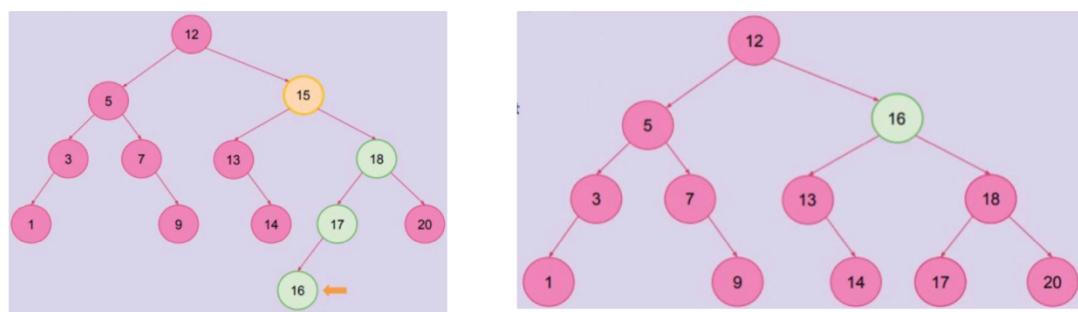
El elemento anterior se enlaza con el hijo del que queremos borrar.

1. Buscamos el nodo a borrar
2. Como tiene un hijo no puedo borrarlo de una
3. AL tener un solo hijo, lo linkeo al nodo padre del elemento borrado directamente con su hijo.
4. Borramos el nodo.



Nodo con 2 hijos

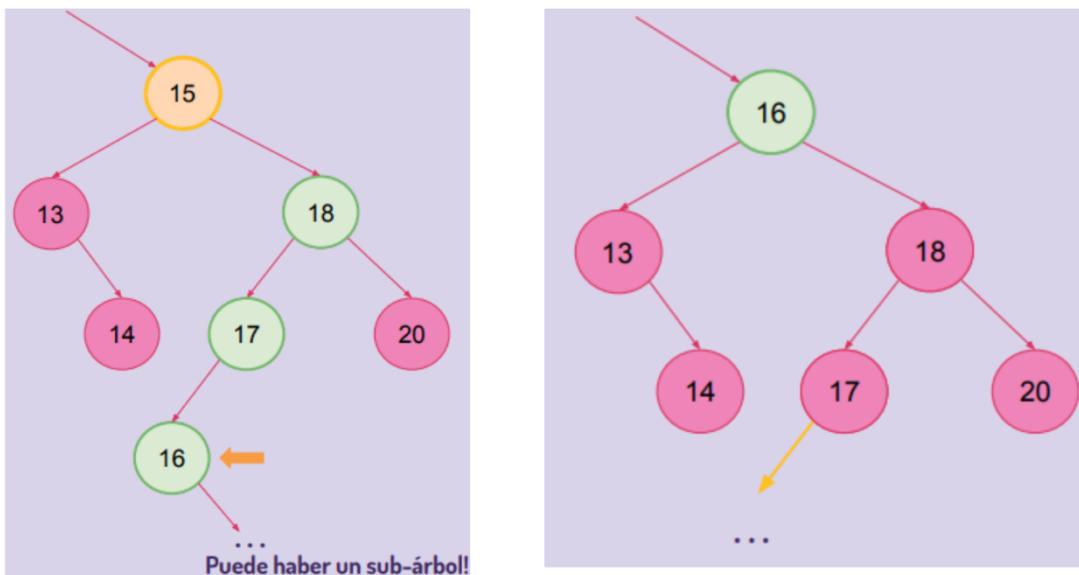
1. Buscamos el nodo a eliminar.
2. Busco su sucesor inmediato:
 1. Tomo el hijo derecho inmediato (rama de los más grandes)
 2. Bajo por los hijos izquierdos hasta que no haya más. Encuentro su sucesor inmediato, el que le sigue más grande.
3. El nodo encontrado es el que va a reemplazar al nodo a borrar
4. Borro el nodo a eliminar



Atención

¿Qué pasa si el 16 tiene un hijo derecho?

Linkeo ese hijo al padre el 16 (el 17) antes de realizar el reemplazo.

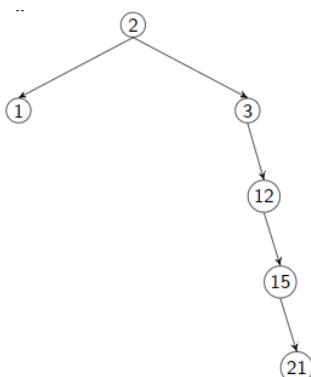


E. ARBOLES BINARIOS EQUILIBRADOS

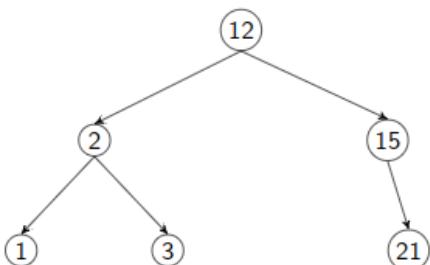
Puede suceder que luego de realizar varias operaciones de búsqueda / eliminación, la estructura quede desfavorable para la realización de ciertos algoritmos.

La búsqueda de un valor en un árbol no puede ser óptima si se encuentra la siguiente estructura. Las complejidades algorítmicas de las operaciones de buscar, insertar y borrar en un ABB, en el peor de los casos, es. Pero yo busco que sean $O(\log N)$.

Método para mantener un árbol balanceado.



Un **árbol binario equilibrado** es aquel en el que la altura de los subárboles izquierdo y derecho nunca difiere en más de una unidad.



Para determinar si un árbol binario está equilibrado, se calcula su factor de equilibrio. Factor de equilibrio: **diferencia de altura entre los subárboles derecho e izquierdo**

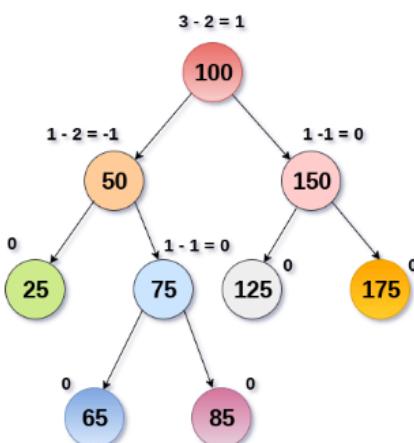
$F_e = \text{altura derecha} - \text{altura izquierda}$ (puedo hacer al revés la cuenta siempre y cuando lo respete para todos)

a) AVL

Árbol binario de búsqueda auto balanceado. Cada hijo difiere en altura de un valor de -1, 0 o 1

- 0 → el nodo está equilibrado y sus sub-arboles tienen exactamente la misma altura
- 1 → el nodo está equilibrado y su sub-arbol derecho es un nivel más alto
- -1 → el nodo está equilibrado y su sub-arbol izquierdo es un nivel más alto

Si el $|F_e| >= 2$ es necesario equilibrar.



Implementación

Un nodo de un AVL tiene los siguientes datos (como mínimo):

- Clave.
- Factor de balanceo.
- Puntero a un AVL a la derecha.
- Puntero a un AVL a la izquierda.

Rotaciones

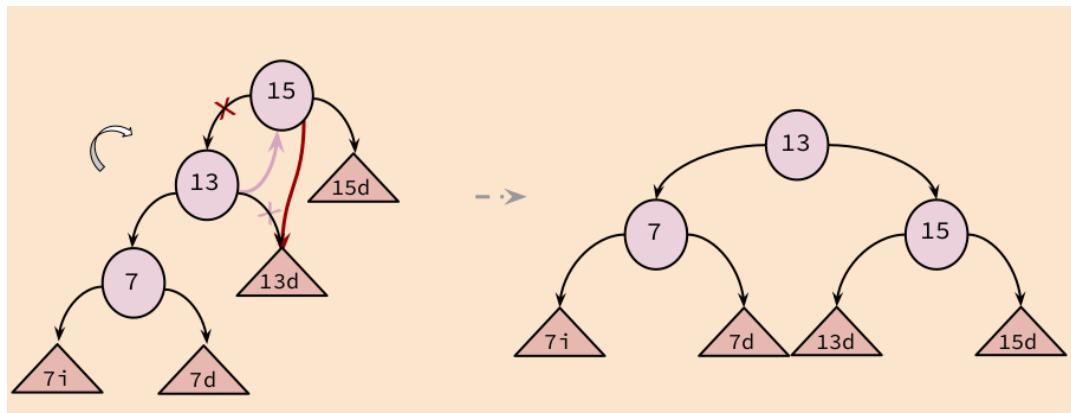
Las rotaciones reorganizan la estructura del árbol después de cada inserción o borrado.

1. Rotación simple: árbol se reorganiza sus nodos hacia la izquierda o hacia la derecha
2. Rotación Compuesta: implica realizar dos rotaciones simples, izq-der o der-izq

Me tengo que fijar como son los factores de equilibrio de los hijos para saber si es rotación simple o doble. Con el factor del "padre desequilibrado" se cual sería la instancia inicial (izquierda o derecha). Si los hijos no están desequilibrados para el mismo lado, entonces la rotación es doble.

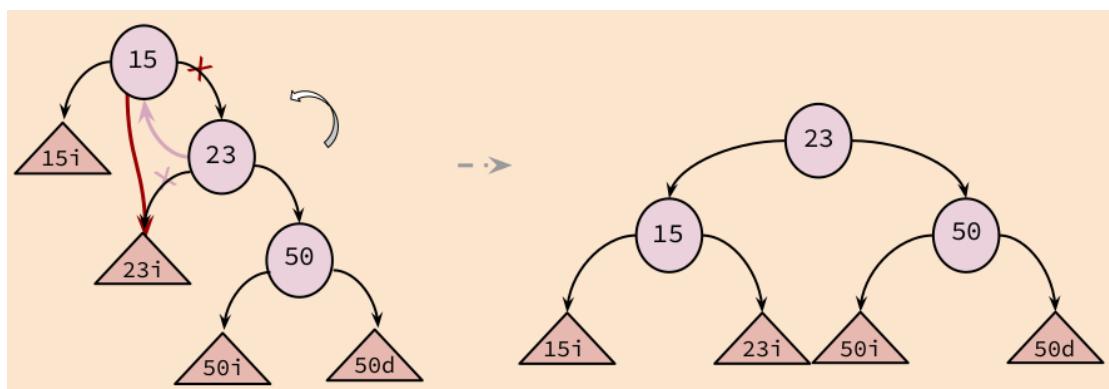
Caso 1: rotación simple a la derecha

Cuando mi árbol pesa más a la izquierda



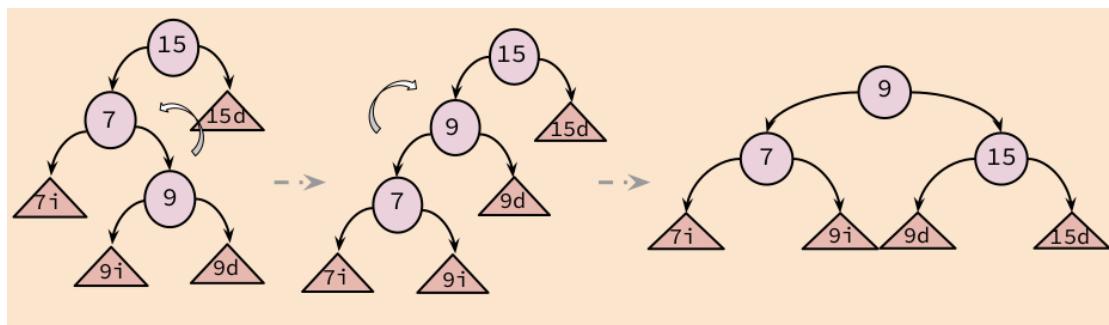
Caso 2: rotación simple a izquierda

Cuando mi árbol pesa más a la derecha.



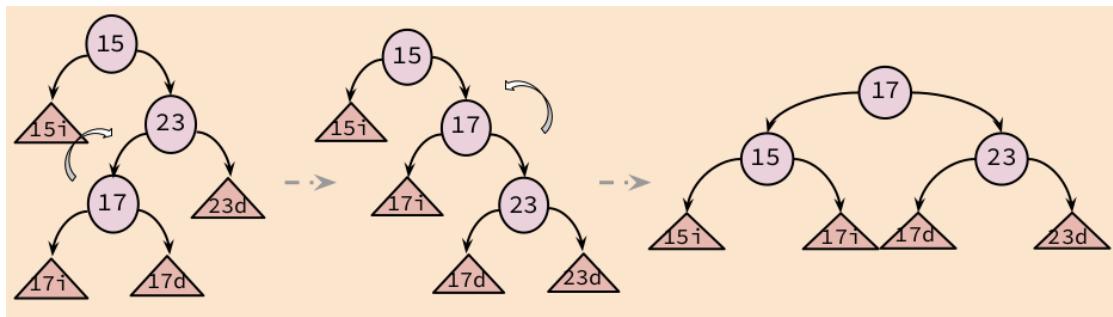
Caso 3: rotación izquierda + derecha

La altura del hijo izquierdo de un nodo es 2 mayor que la del hijo derecho y el hijo izquierdo pesa a la derecha



Caso 4: rotación derecha + izquierda

La altura del hijo derecho de un nodo se vuelve 2 veces mayor que la del hijo izquierdo y el hijo derecho pesa más en la izquierda.

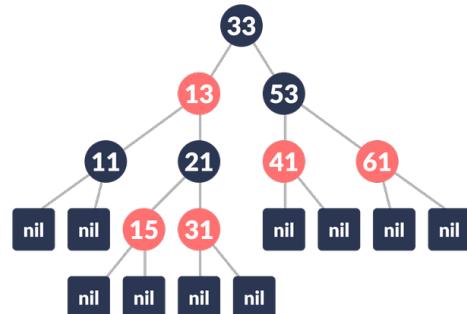


b) Árboles Rojo-Negro

El árbol rojo-negro es un **árbol de búsqueda binario autobalanceado** en el que cada nodo contiene información adicional para indicar el color del nodo, ya sea rojo o negro. Cada nodo tiene los siguientes atributos: color, clave, hijo izquierdo, hijo derecho y padre (excepto el nodo raíz).

→ La **altura** de un árbol rojo-negro es $O(\log N)$

→ La **profundidad de negro** del árbol es $O(\log Nb)$, donde nb es el número de nodos negros.



Propiedades

- Propiedad de rojo/negro:** cada nodo es de color, ya sea rojo o negro.
- Propiedad de la raíz:** la raíz es negra.
- Propiedad de la hoja:** cada hoja (NULL) es negra. No tienen ninguna clave, apuntan a NULL.
- Propiedad roja:** si un nodo rojo tiene hijos, los hijos siempre son negros. O desde otro punto de vista, cada nodo rojo debe tener un parente negro. Ningún camino desde la raíz hasta una hoja tiene dos nodos rojos consecutivos.
- Propiedad de altura:** cada camino desde un nodo dado a cualquiera de sus hojas tiene el mismo numero de nodos de color negro. → Cuando se calcula la altura negra o la profundidad negra se computan solo los nodos negros. No se computa el nodo sobre el cual estoy parado si es de color negro. Por ejemplo, la altura de 15 es 1, para 21 es 1 también o para 33 es 2.

```
typedef struct nodo{
    char color;
    void* dato;
    nodo_t *padre, *derecha, *izquierda;
    //El parente es necesario para las rotaciones e inserciones
}nodo_t;
```

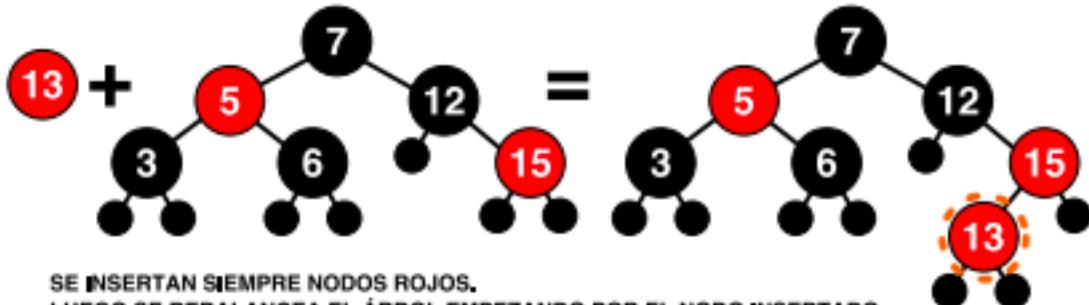
- **Nodo parente:** nodo→padre
- **Nodo abuelo:** nodo→padre→padre

- **Nodo tío:** nodo → padre → padre → izquierdo o nodo → padre → padre → derecho.
Normalmente se busca el izquierdo, y si no hay buscas el derecho.

Rotaciones

Para conservar las propiedades, en ciertos casos de la inserción y de la eliminación será necesario reestructurar el árbol.

Restructuración



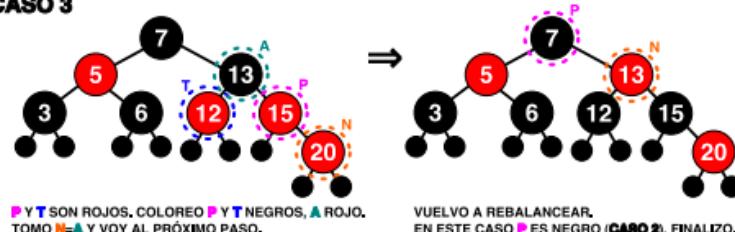
CASO 1



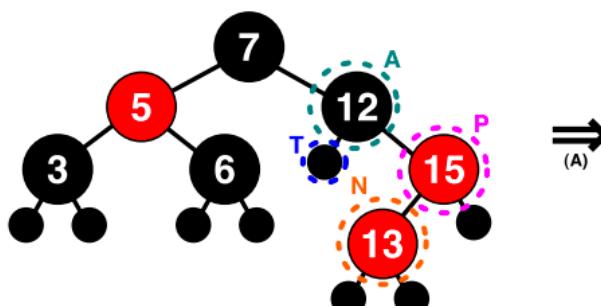
CASO 2



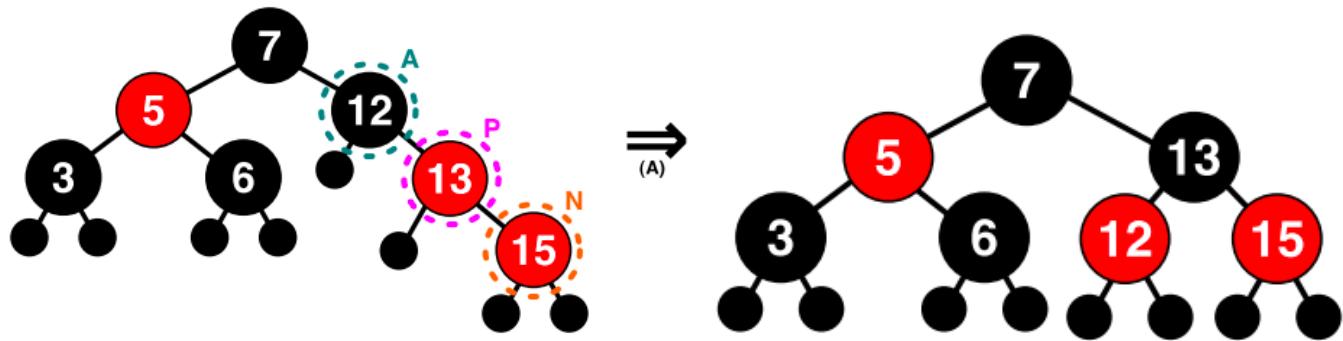
CASO 3



CASO 4



P ROJO, T NEGRO:
A) SI P^{DER} A y N^{Izq} P, ROTACION DERECHA P.
 TOMO N=P, IR AL PRÓXIMO PASO.
B) SI P^{Izq} A y N^{DER} P, ROTACION IZQUIERDA P.
 TOMO N=P, IR AL PRÓXIMO PASO.
C) IR AL PRÓXIMO PASO.



A) SI $N \xrightarrow{D} P$, ROTACIÓN IZQUIERDA A
COLOREO P NEGRO, A ROJO

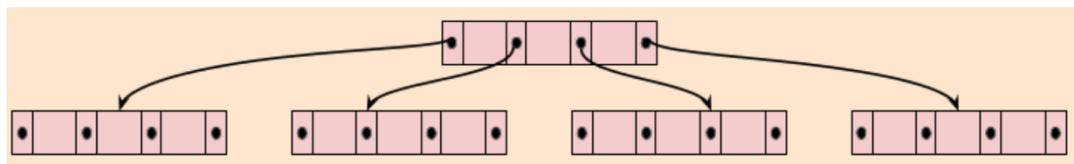
B) SI $N \xrightarrow{Izq} P$, ROTACIÓN DERECHA A
COLOREO P NEGRO, A ROJO

F. FAMILIA DE ARBOLES B

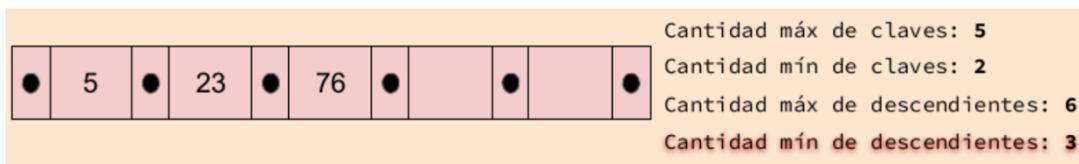
Los árboles B de búsqueda surgen a partir de la necesidad de contar con un gran número de elementos pero que todos ellos no pueden entrar en memoria

Características

1. Poca profundidad: crecen más a lo ancho que a lo alto.
2. Acceso es poco costoso
3. Claves están ordenadas.
4. Todas las ramas que parten de un determinado nodo tienen exactamente la misma altura. Todos los nodos hoja están al mismo nivel.
5. La **cantidad mínima de claves** es $k/2$ \ redondeado\ hacia \ abajo\\$ (excepto la raíz)
 1. k es la cantidad máxima de claves
6. Un nodo con k claves tiene como **máximo $m=k+1$ descendientes**
7. La cantidad mínima de descendientes es $m/2$ (excepto hojas y raíz)



Ejemplo

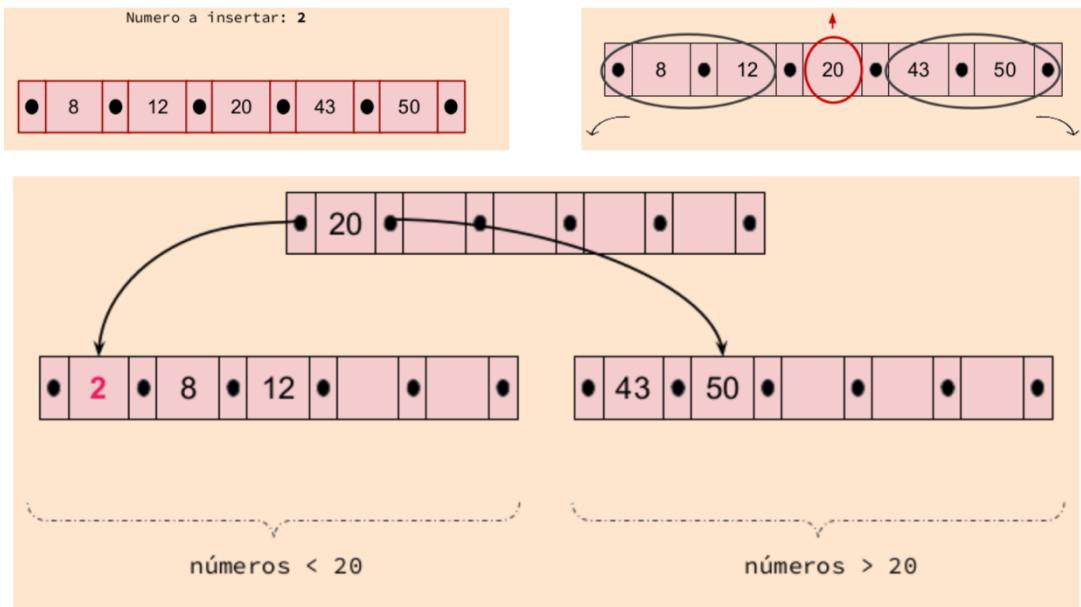


Inserción

El árbol B es una estructura que crece de abajo hacia arriba, es decir desde las hojas hacia la raíz. El elemento se inserta SIEMPRE en las hojas

1. Buscar en el árbol B la hoja nodo donde el nuevo valor clave debería ser insertado
2. Si el nodo hoja no está completo (contiene menos de k claves) entonces insertar el nuevo elemento manteniendo el orden de los elementos
3. Si el nodo hoja está lleno: **overflow**

Divido el nodo, generando 2 y promuevo la clave del medio al nivel superior



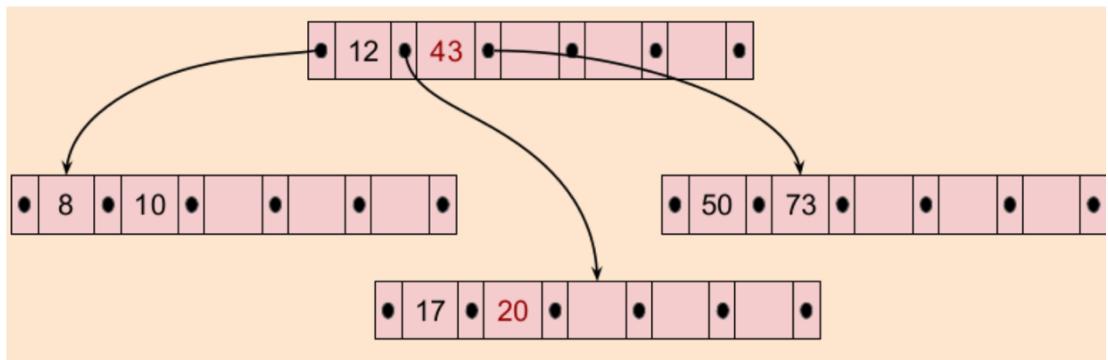
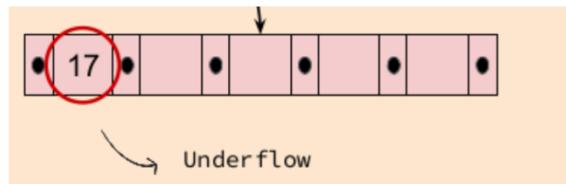
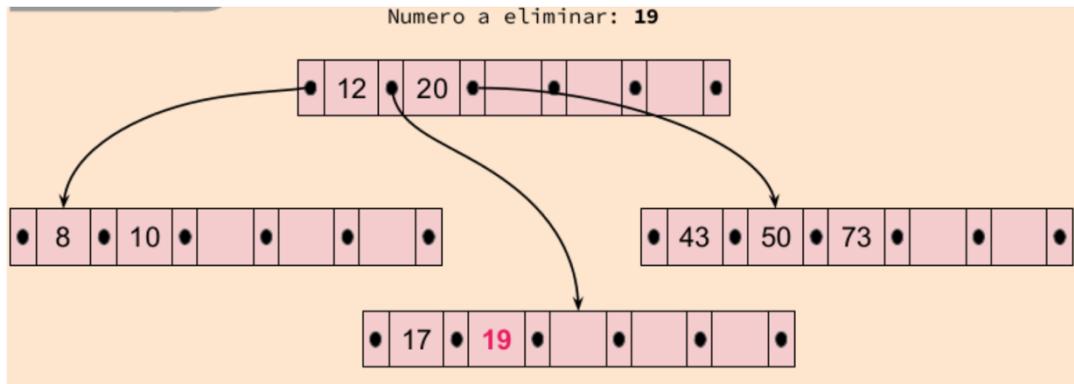
Eliminación

La eliminación de elementos se realiza desde los nodos hojas.

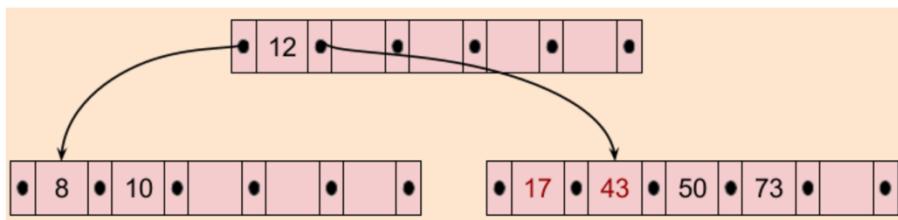
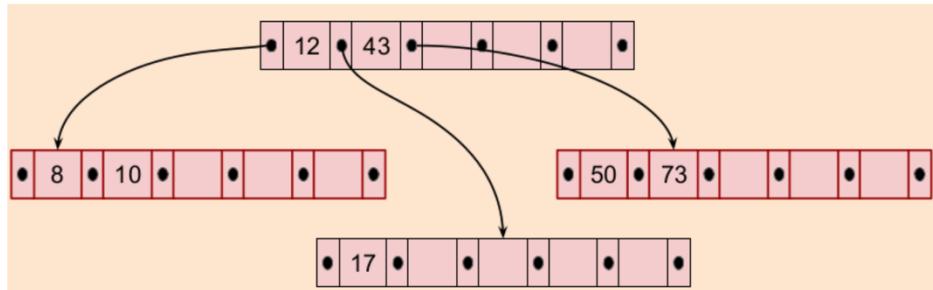
1. Localizar el nodo hoja en el que se debe eliminar.
2. Si tiene más claves del mínimo de número de claves (más de $m/2$) entonces se elimina el valor.
3. Sino, si el nodo hoja contiene justo $m/2$ elementos, entonces hay que completar el nodo tomando el elemento hermano de izquierda o de derecha

REDISTRIBUCIÓN: tengo un underflow

- Subo clave más chica del hermano derecho y bajo el padre que separa a ambos.
- Subo la clave más grande del hermano izquierdo y bajo el padre que separa a ambos

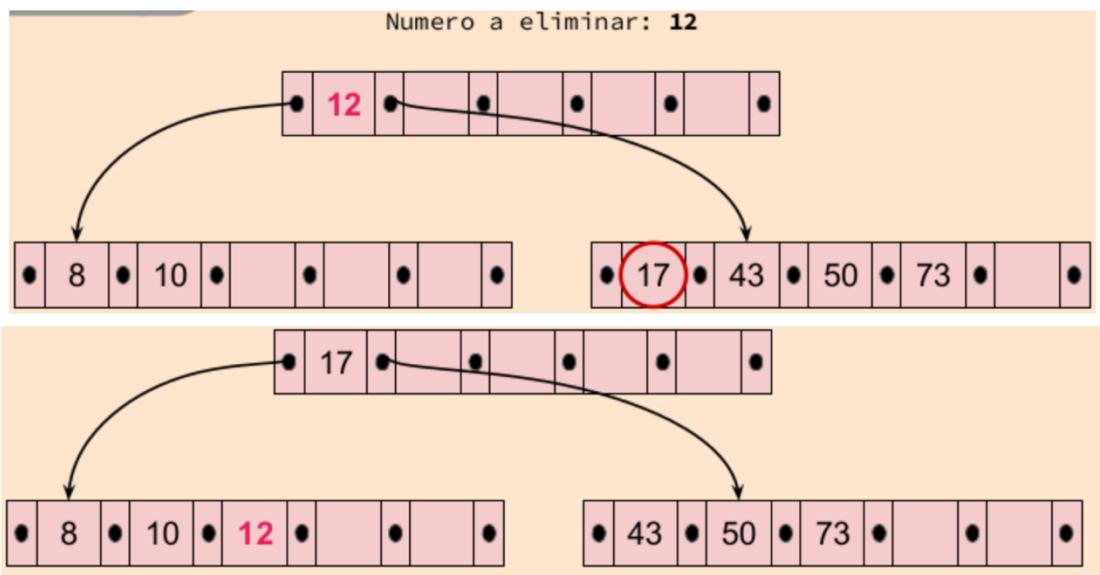


CONCATENACIÓN: tengo underflow de ambos hermanos

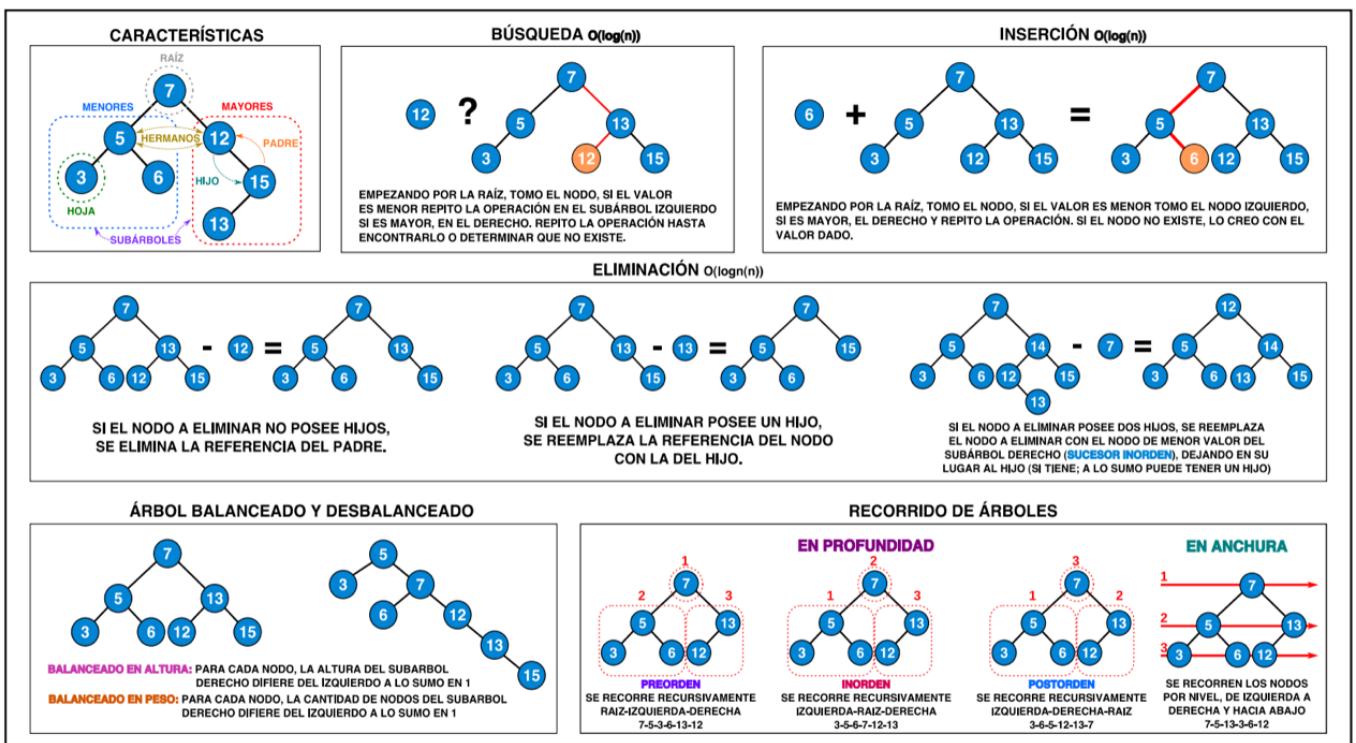


ELIMINAR NODO QUE NO ESTA EN LA HOJA: intercambio

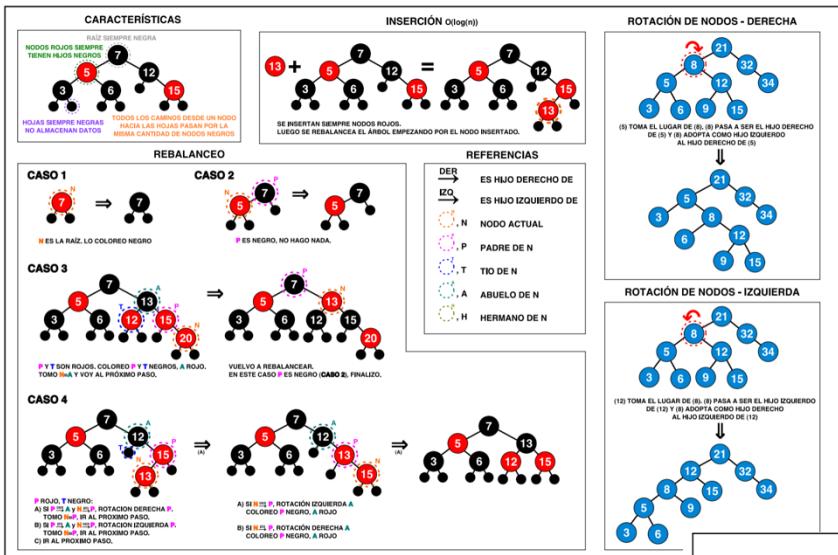
- intercambio el elemento con el inmediato superior y lo bajo a la hoja
- intercambio el elemento con el inmediato inferior y lo bajo a la hoja



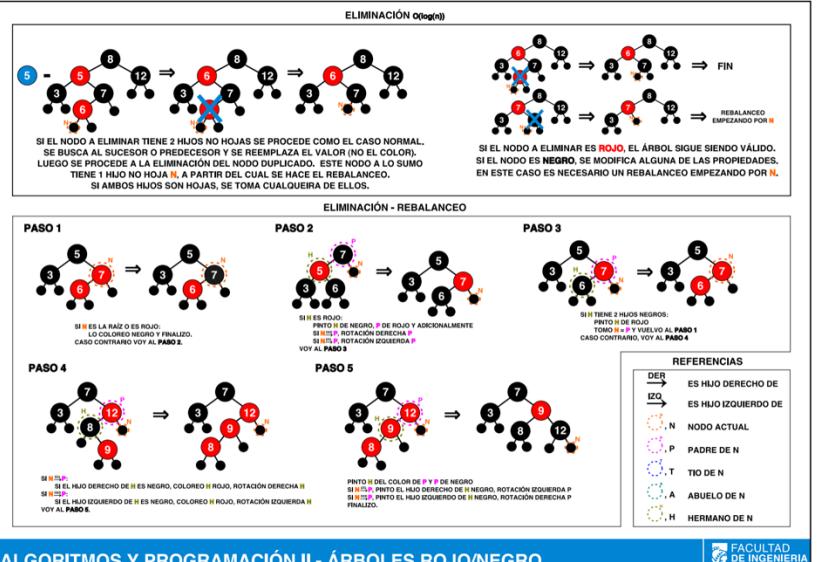
G. MACHETES



ARBOLES ROJO Y NEGRO MACHETE



ALGORITMOS Y PROGRAMACIÓN II - ÁRBOLES ROJO/NEGRO



ALGORITMOS Y PROGRAMACIÓN II - ÁRBOLES ROJO/NEGRO

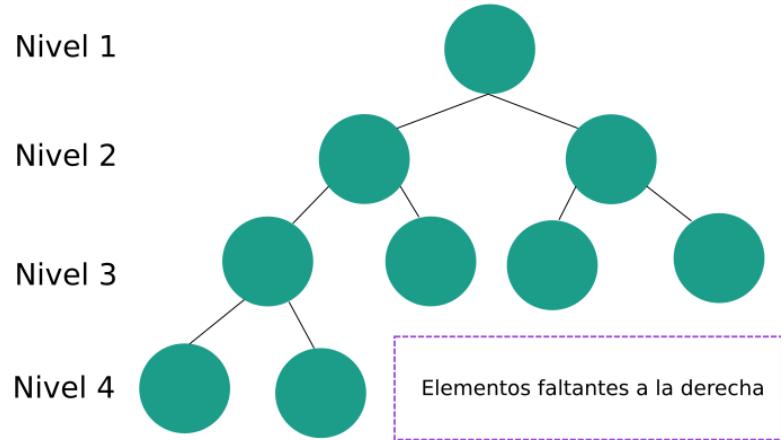
9. HEAP BINARIO

A. INTRODUCCION

Un heap binario es un árbol binario que tiene ciertas características. Presenta cierta relación entre elementos, ya que hay un **criterio de orden parcial**.

- No hay un criterio de orden total
- No hay relación entre los hermanos, sobrinos y tíos

Este tipo de árbol se lo denomina casi completo → si recorro por niveles solo puedo tener elementos faltantes en el último nivel. El "agujero" siempre está del lado derecho. Siempre se contemplan los niveles de forma ordenada, no puedo empezar un nuevo nivel si hay espacios en el actual.



Propiedades para ser Heap binario

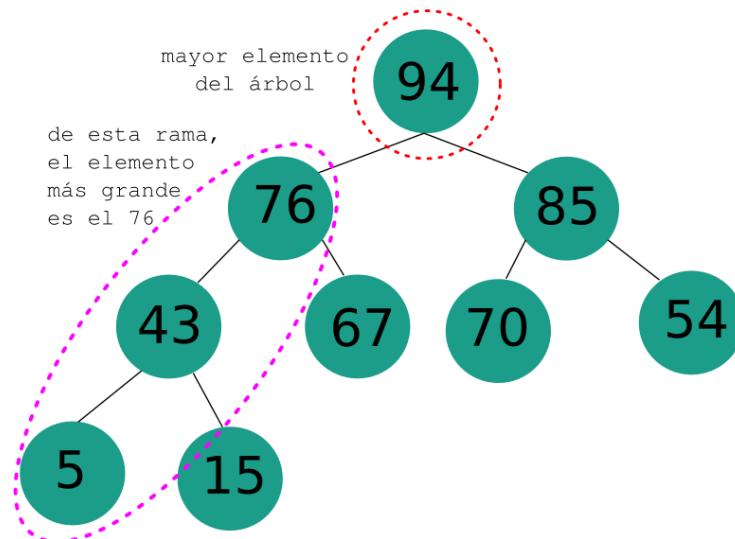
1. Árbol ordenado por rama (la raíz es del árbol es el menor o mayor elemento)+
2. Todos los niveles del heap, exceptuando el último, están completos (árboles casi completos)
3. Solo se puede leer, buscar y borrar la raíz del heap.

Heap maximal o minimal

Heap maximal

El nodo raíz de la rama actual es el mayor en comparación a los elementos que estén por debajo. El nodo con el mayor elemento se encuentra en la raíz de la rama actual.

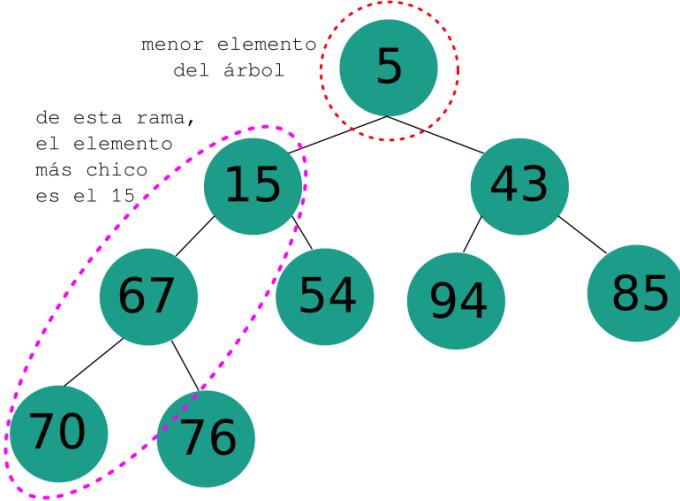
Por ende el **nodo raíz de TODO el árbol es el elemento más grande** de todos.



Heap minimal

El nodo con el menor elemento se encuentra en la raíz de la rama actual.

El **elemento menor del heap se encuentra en la raíz de todo el árbol**



B. IMPLEMENTACION

Un heap binario, a diferencia de un árbol abb, puede ser implementado con un vector. Si conozco cuantos elementos va a haber en nuestro heap, puedo inicializar el vector en el stack

- Menos riesgo
- Moverme en la estructura es más fácil

Un abb con un vector no es conveniente porque el abb no es una estructura del tipo casi completa y puede presentar agujeros en el medio de los niveles → necesitaría mucha memoria para terminar utilizando poca.

¿Para qué quiero un heap?

Un ejemplo es el método de ordenamiento heap sort. Realiza la conversión de un vector a un heap (de ser necesario) y el ordenamiento propiamente dicho.

La función presenta una complejidad de $O(n \log(n))$

No es un método estable: los elementos repetidos no mantienen un orden relativo entre sí incluso luego del ordenamiento.

C. OPERACIONES

Se utiliza el heap cuando queremos interactuar siempre con el más grande o más chico de los elementos.

Solo podemos acceder al elemento de la raíz del heap (leerlo, buscarlo y eliminarlo). Es la condición que nos permite tener la operación de búsqueda con complejidad logarítmica de $O(1)$.

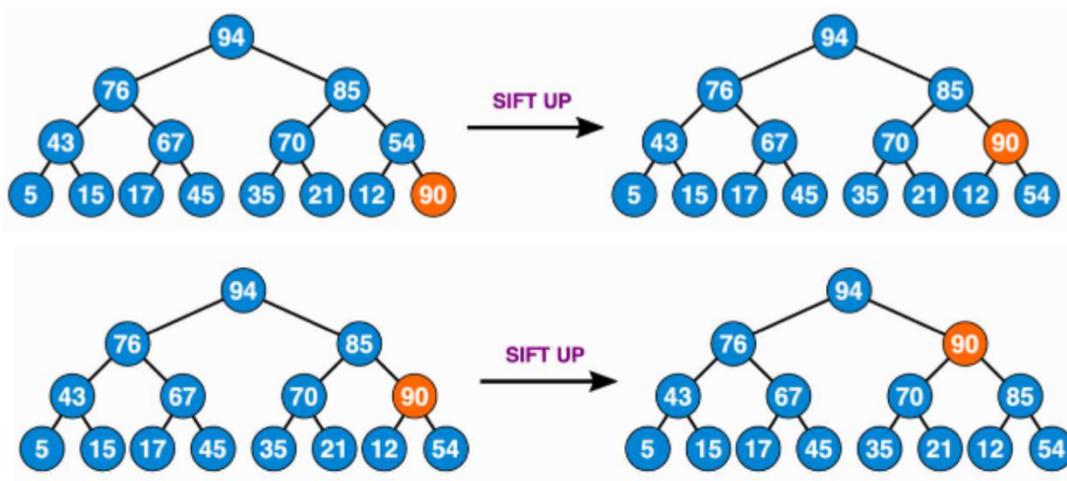
EXPLICO CON HEAP MAXIMAL

Insertar O(logn)

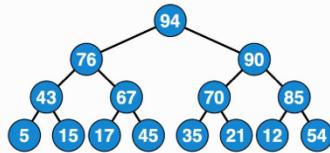
Vamos a insertar en donde tenemos el primer espacio libre, que será en mi último nivel. Una vez insertado, me fijo hasta donde tiene que flotar. Comparo mi elemento con el padre y sube el que es mayor y baja el menor. Cuando llego al punto donde no tiene que seguir, paro.

Realizo SIFT UP: el nodo actual le pregunta al padre, ¿soy mayor que vos?

- SI, se intercambian (el padre pasa a ser el hijo y el hijo el padre).
- NO, el padre es mayor que el nodo actual, así que el nodo actual está en la posición correcta.



En la tercera iteración, comparamos el 90 con el 94. Como el 94 es mayor, se mantiene como la raíz y encontramos el lugar el 90.

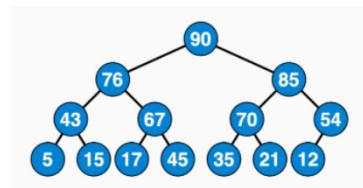
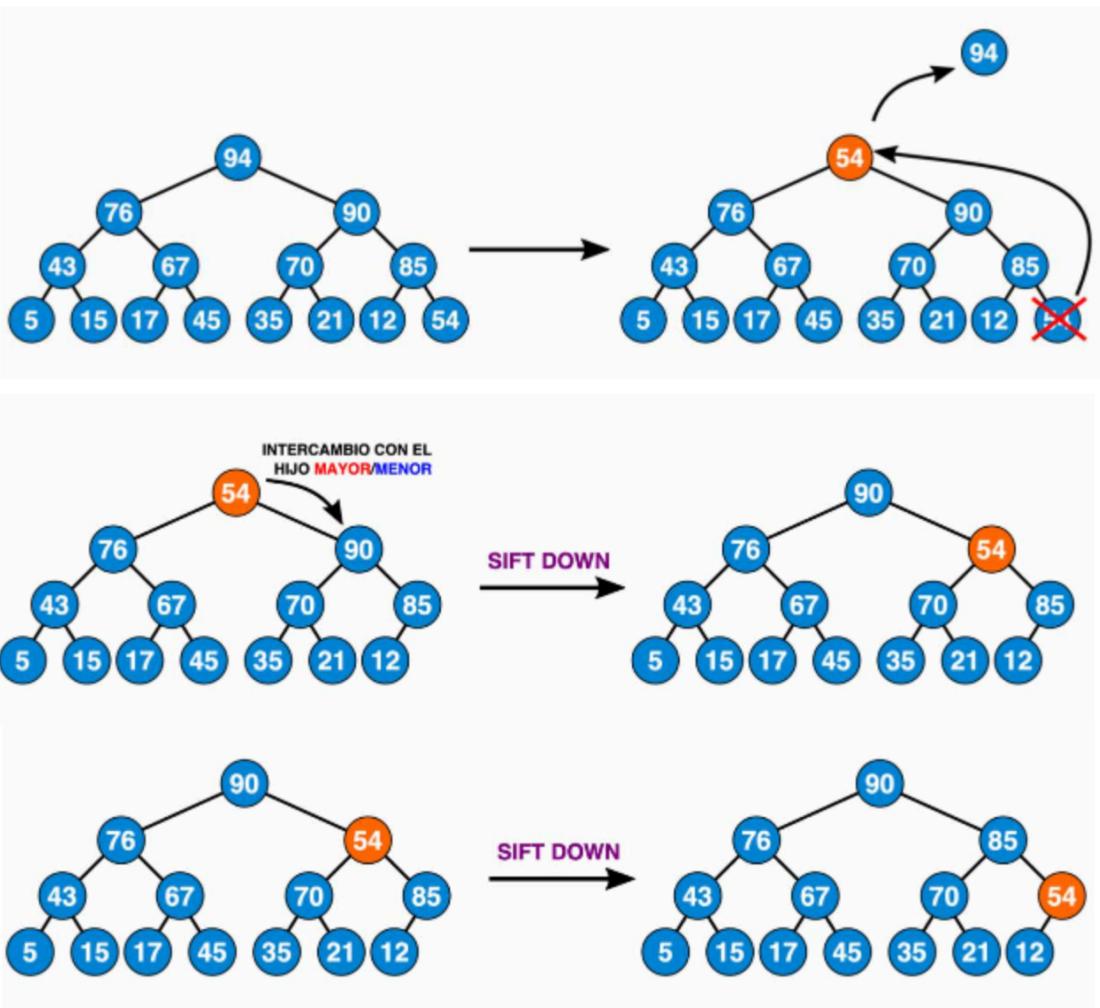


Eliminar O(logn)

Únicamente puedo borrar el elemento raíz del árbol.

Borro el elemento y lo reemplazo el último elemento disponible. A partir de ahí, dejo que se unda hasta que encuentre la posición correcta.

Realizo SIFT DOWN: esta operación compara el nodo actual con todos los hijos directos e intercambia al actual con el hijo mayor. Si no hay hijo mayor, el nodo actual es el mayor de los tres y encontramos el lugar correcto del nodo actual.



En la tercera iteración
comparamos el 54 con el
12. Como el nodo actual es
mayor a los hijos
encontramos el lugar
adecuado

```
void sift_down(int* elementos, size_t posicion, size_t
ultima_posicion){
    size_t pos_izq = posicion*2+1;
    size_t pos_der = posicion*2+2;
    size_t pos_a_intercambiar = posicion;

    if(pos_izq > ultima_posicion) //no tengo ni hijo izq ni
derecho
        return;
    if(pos_der > ultima_posicion) //no tengo hijo derecho
        pos_der = pos_izq;

    int elemento = elemento[posicion];
    int elemento_i = elemento[pos_izq];
    int elemento_d = elemento[pos_der];

    if(elemento_d > elemento_i) //busco el mayor de los dos hijos
```

```

        pos_a_intercambiar = pos_der;
    else
        pos_a_intercambiar = pos_izq;

    if(elementos[pos_a_intercambiar] > elemento){
        swap(elementos, posicion, pos_a_intercambiar)

        sift_down(elementos, pos_a_intercambiar, ultima_posicion);
    }
}

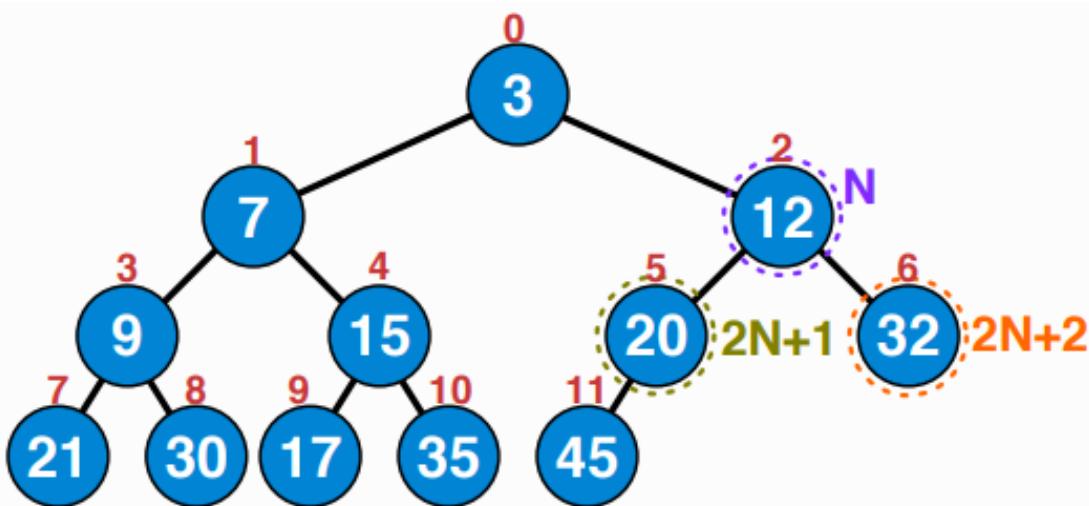
```

D. CONSTRUCCION

Método básico: Insertar elementos en un heap vacío uno a uno. Realizo n operaciones $\log(n)$

Complejidad: $n \log(n)$

Si el nodo actual está en la posición n, el hijo izquierdo está en $2n+1$ y el derecho en $2n+2$



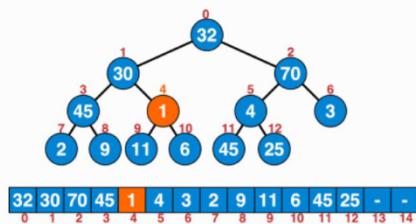
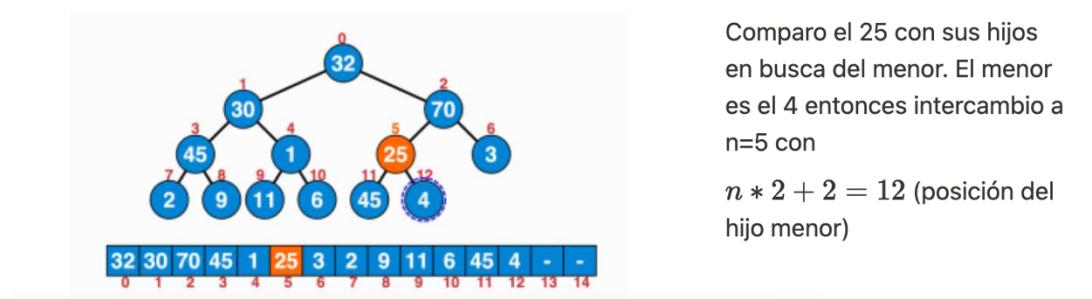
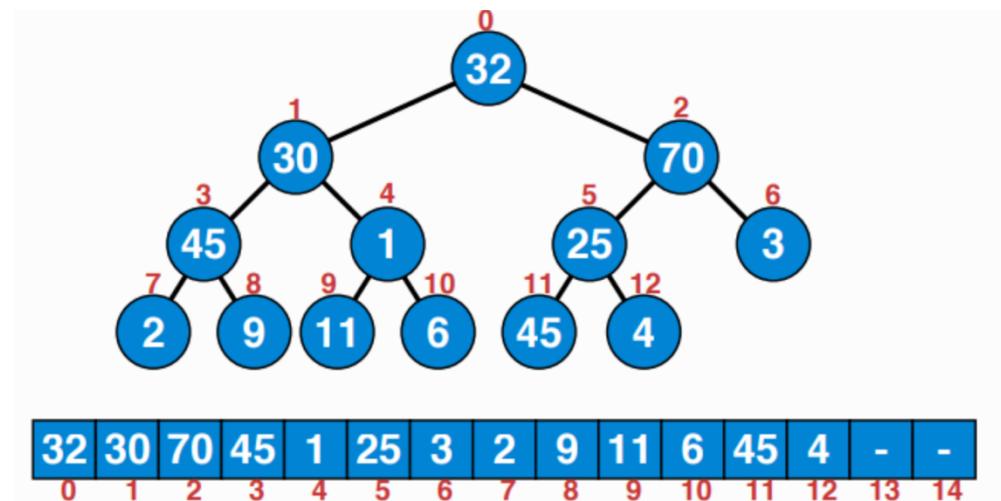
Desde un array: heapify

Ordeno el mismo vector mediante **heapify**. Esta función recorre un vector que no tiene agujeros y lo reordena para que cumpla las condiciones de un heap.

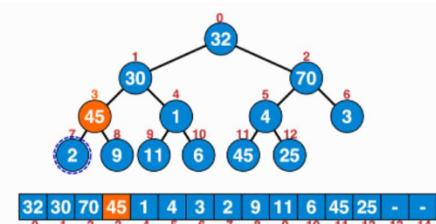
El algoritmo empieza averiguando cuantos elementos podemos tener en nuestro heap: $2^n - 1$ donde n es la cantidad de niveles. A cada elemento se la va a aplicar la función *sift down*, para generar un heap minimal. Pero, no tiene sentido aplicar esta función a los elementos de la última fila porque se que si o si TODOS son elementos hoja. Por lo tanto, empiezo desde la ante última fila en el último elemento: $(n-2)/2 = \text{inicio}$ donde n es la cantidad de elementos.

Ejemplo

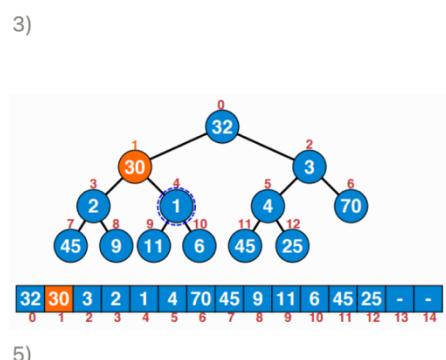
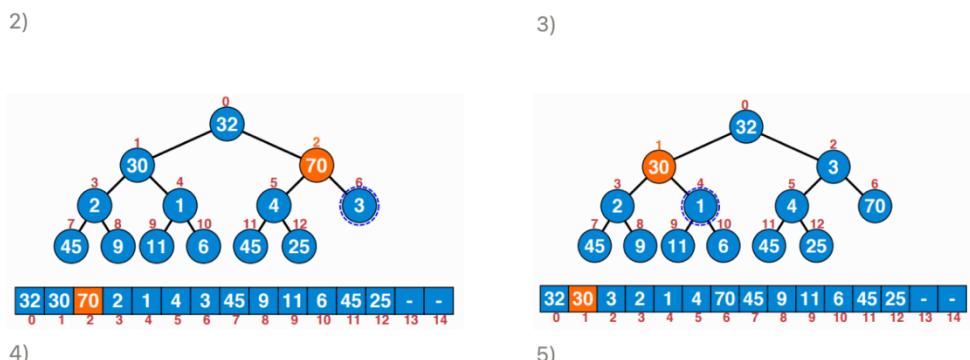
Quiero transformar este vector en un heap minimal.

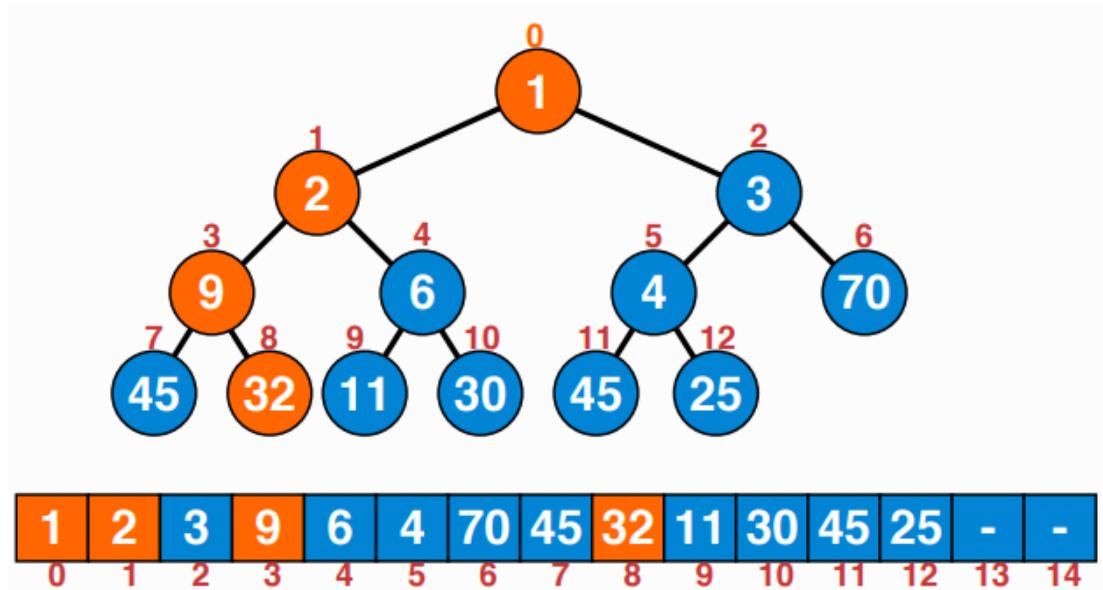
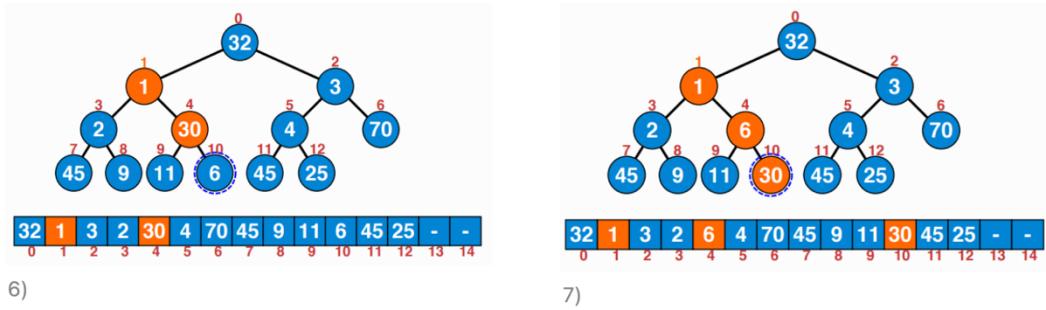


2)



3)





```
void heapify(int* elementos, size_t cantidad) {
    if(cantidad==0)
        return;

    for(size_r pos = (cantidad-2)/2; pos > 0; pos--)
        sift_down(elementos, pos, cantidad-1);

    sift_down(elementos, 0, cantidad-1);
}
```

Complejidad

Al analizar el peor caso, digo que tengo que aplicarle **sift down** a todos los elementos menos a la ultima fila $\rightarrow n/2$ elementos: osea que sera $n/2 \log n$ pero tomamos $O(n \log n)$

E. HEAP SORT

Mi elemento tiene que tener forma de heap, por lo que lo primero sería hacer un heapify a mi vector.

Complejidad: $O(n \log n)$

Luego aplico las operaciones de heap

- Hago operaciones desde la raíz → siempre saco el menor entonces tendría ordenado de menor a mayor.

Al extraer de la raíz, intercambio por el último. Luego de sacarlo tengo que restarle uno a cantidad y reordenar mi vector haciendo sift down

```
void heap_sort_rec(int* elementos, size_t cantidad){
    if(cantidad <=1) //vector con 1 elemento siempre esta ordenado.
        return;

    swap(elementos,0,cantidad-1);
    cantidad--;
    sift_down(elementos,0,cantidad-1);
    heap_sort_rec(elementos,cantidad);

}

void heap_sort(int* elementos size_t cantidad){
    heapify(elementos, cantidad);
    heap_sort_rec(elementos,cantidad);
}
```

F. MACHETE

CARACTERÍSTICAS	REPRESENTACIÓN COMO ARRAY
<p>ES UN ARBOL BINARIO (CASI) COMPLETO TIENE LA MENOR ALTURA POSIBLE NO ES UNA ESTRUCTURA ORDENADA SON ÚTILES PARA LA BÚSQUEDA DE EXTREMOS</p> <p>HEAP BINARIO MAXIMAL EL PADRE DE UN NODO ES MAYOR O IGUAL A SUS HIJOS</p>	<p>CADA ELEMENTO DEL HEAP SE ALMACENA EN UN ARRAY. LA RAÍZ SE ALMACENA EN LA POSICIÓN 0 DEL ARRAY. LOS HIJOS DE UN NODO N SE ALMACENAN EN LAS POSICIONES 2N Y 2N+1. ES LA REPRESENTACIÓN MAS EFICIENTE, NO REQUIERE REFERENCIAS ENTRE NODOS, ES UNA REPRESENTACIÓN SERIALIZADA.</p> <p>HEAP BINARIO MINIMAL EL PADRE DE UN NODO ES MENOR O IGUAL A SUS HIJOS</p>
<p>INSERCIÓN $O(\log(n))$</p> <p>AL INSERTAR UN NODO, SE AGREGA EL VALOR EN LA PRIMERA POSICIÓN DISPONIBLE EN EL HEAP.</p> <p>LUEGO DE INSERTAR EL NODO, SE LO VA INTERCAMBIANDO CON SU PADRE HASTA QUE CUMPLA CON LA PROPIEDAD DEL HEAP. SEGÚN SEA UN HEAP MAXIMAL O MINIMAL VERIFICAMOS QUE EL NODO SEA MAYOR O MENOR (O IGUAL) QUE EL PADRE Y SI NO CUMPLE SE INTERCAMBIAN.</p>	<p>ELIMINACIÓN DE LA RAÍZ $O(n \log(n))$</p> <p>PARA ELIMINAR LA RAÍZ, SE LA REEMPLAZA CON EL ÚLTIMO ELEMENTO DEL HEAP.</p> <p>LUEGO DE REALIZAR EL REEMPLAZO, SE VA DESPLAZANDO EL NODO HACIA ABAJO HASTA QUE VUELVA A CUMPLIR LA PROPIEDAD DEL HEAP. SEGÚN SEA UN HEAP MAXIMAL O MINIMAL SE INTERCAMBIA EL VALOR CON SU HIJO MAYOR O MENOR HASTA QUE YA NO SEA POSIBLE HACERLO.</p>
<p>HEAPSORT $O(n \log(n))$</p> <p>CONSISTE EN SIMPLEMENTE IR ELIMINANDO LA RAÍZ. TIENEMOS GARANTIZADO QUE LUEGO DE LA ELIMINACIÓN EL ELEMENTO DE LA RAÍZ VUELVE A SER EL MÁXIMO O MÍNIMO SEGÚN EL CASO. POR LO TANTO EN CADA PASO SE EXTRAÉ EL SIGUIENTE VALOR EN SECUENCIA.</p>	<p>FACULTAD DE INGENIERÍA Universidad de Buenos Aires</p>

10. METODOS DE ORDENAMIENTO

A. ASPECTOS GRALES

- El objetivo de un método de ordenamiento es del de reorganizar los ítems de forma tal que sus claves estén ordenadas de acuerdo a una regla bien definida.
- Si el archivo que tiene que ser ordenado cabe completamente en memoria, entonces el método de ordenamiento es interno. Si el archivo esta en disco, el metodo de ordenamiento es externo.
 - Diferencia: ordenamiento interno cualquier ítem puede ser accedido fácilmente. E el externo es accedido secuencialmente o por medio de grandes bloques.
- Los algoritmos pueden diferenciarse entre
 - **No adaptativos**
 - **Adaptativos**
- La primera característica de estudio de un algoritmo de ordenamiento es su tiempo de ejecución
- Existen algoritmos comparativos y no comparativos
- Link con diapos:
https://drive.google.com/drive/folders/1d9rDuyv6hn_05wRh_jjH9ByuThURWC38?usp=sharing

B. MERGE SORT

Es un algoritmo del tipo divide y conquista. Complejidad: $n \log(n)$

Trabaja particionando un arreglo en dos partes, llamándose recursivamente hasta que el arreglo tenga longitud 1 o cero. Una vez alcanzado el caso base, mezcla cada arreglo de longitud 1 en forma ordenada hasta obtener el arreglo de longitud original ordenado.

1. Divide a la mitad
2. Me llamo con la mitad izquierda
3. Me llamo con la mitad derecha
4. Merge

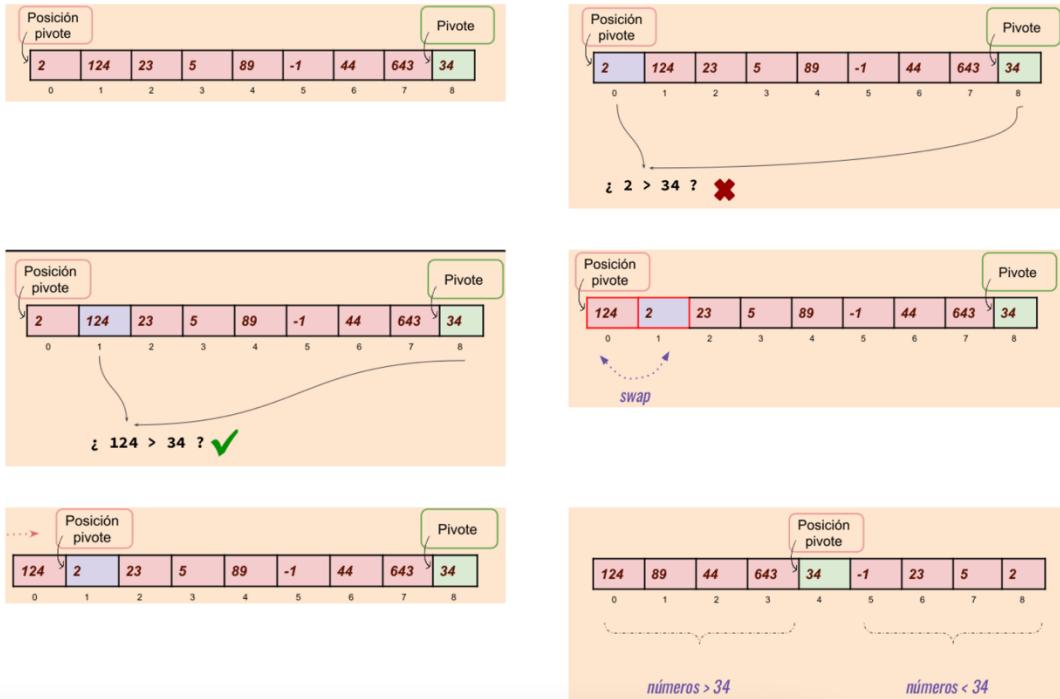
C. QUICKSORT

El algoritmo de ordenamiento Quicksort tiene la deseable característica de utilizar una pequeña pila auxiliar, solo requiere alrededor de $O(n \log n)$ operaciones en promedio para ordenar N elemento, y posee un ciclo interno extremadamente corto. Su desventaja es que no es un algoritmo de ordenamiento estable, es decir no preserva su orden relativo si la cantidad de claves se duplica, es decir que en el peor de los casos se necesitan $O(n^2)$ operaciones.

El Quicksort es un algoritmo de la familia divide y conquista. Trabaja particionando un arreglo en dos partes, después ordena cada una de las partes independientemente. El punto crucial del método es el proceso de partición.

1. Elijo un pivote (me conviene elegir el último elemento)
2. Voy iterando reubicando los mayores al pivote.
3. Ubico al pivote en su posición.
4. Me llamo recursivamente a la izquierda y luego a derecha.

Una vez ubique al pivote, repito lo mismo con las mitades mayores y menores al pivote.



D. EXTRA(+) (ver los apuntes de mpata2000)

11. HASH

A. INTRODUCCION

Diccionario:

- Colección de pares clave/valor
- Accedo a un elemento/lo defino mediante su clave

¿Por qué usarlo?

Aumento performance a la hora de acceder al dato.

No hay duplicación de entradas

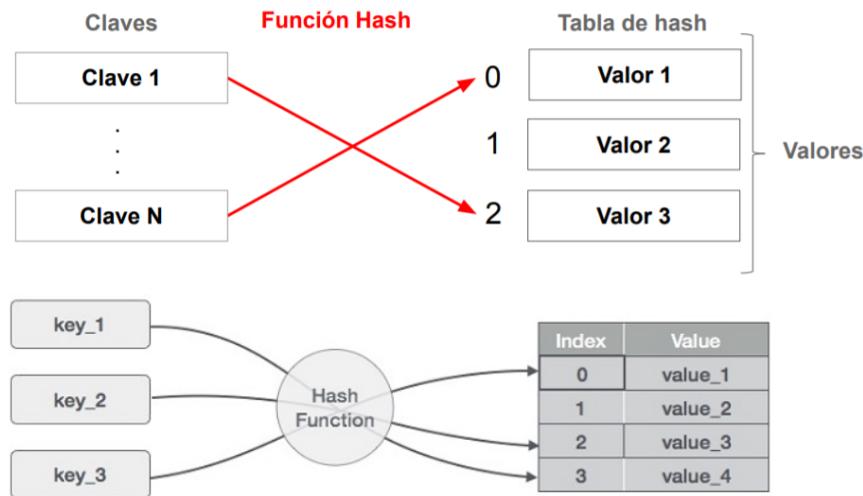
Tabla Hash:

Estructura que contiene **valores**

- datos asociados a las claves

Puedo hallar su valor a partir de una **clave**

- dato único que se utiliza para conectar con los valores (puede ser de cualquier tipo de dato siempre y cuando la función hash pueda interpretarla para generar un índice a partir de ella en la tabla)



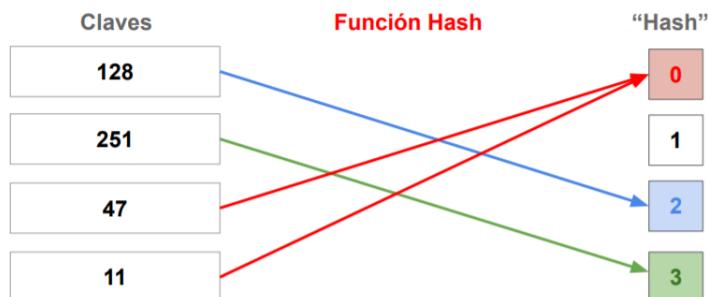
Función Hash:

Función Hash: transforma claves en un número asociado.

Toda función tiene cierto grado de dispersión → que tan buena es la función de Hash en direccionar.

Puede haber más claves que espacios en la tabla de hash

- Claves distintas dan el mismo valor de "hash"



Buena función Hash

1. El valor hash está completamente determinado por los datos que se procesan (datos de entrada).
2. La función utiliza todos los datos de entrada
3. Distribuye uniformemente los datos en todo el conjunto de posibles valores hash.
4. Genera valores muy diferentes para claves similares.

Colisiones

- **Misma clave, mismo hash**
- Cuando la función de hash genera el mismo índice para varias claves, habrá un conflicto (qué valor se almacenará en ese índice). A esto se le llama colisión de hash, no se pueden evitar, simplemente hay que manejar las colisiones. Hay que tener en cuenta que una colisión solo se da cuando la función hash genera un índice que ya está ocupado, si se trata de la misma clave se trata de actualización de valores.

B. TIPOS DE HASH

Direccionamiento cerrado → Hash abierto

Direccionamiento abierto → Hash cerrado

$$\text{"Hash"} = \text{clave \% Tamaño_tabla}$$

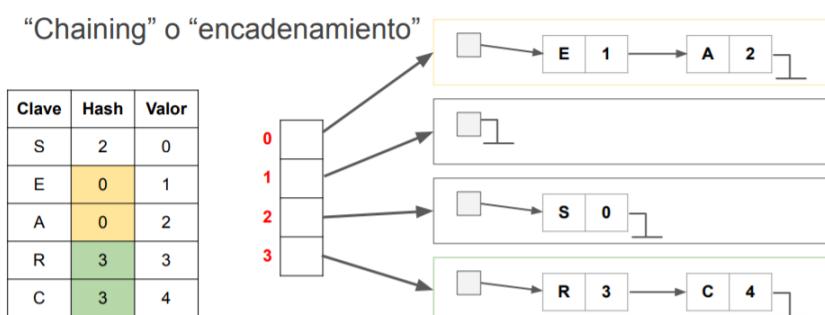
Hash abierto:

Se dice que es abierto porque guardo la información afuera → otro TDA (lista)

Direccionamiento es cerrado porque cuando accedo a una dirección, se queda en ese mismo índice de la tabla

- Cada vez que hay una colisión, igualmente voy a esa posición de memoria. Si mi Hash me dice que la clave está en la posición 0, entonces está ahí.

Colisión: resolución por encadenamiento → elementos se almacenan en el mismo índice utilizando una lista enlazada



Complejidad:

Para encontrar una clave con su correspondiente valor en caso de colisión es

- $O(n)$ donde n es la cantidad de elementos que colisionaron
- Voy a tener que recorrer la lista enlazada

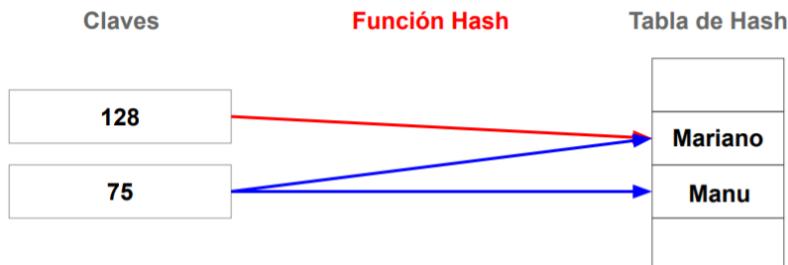
Hash cerrado

Todos los valores se guardan dentro de la misma tabla (no hay un TDA intermedio que funcione como bolsa de mis datos)

El tamaño de la tabla debe ser siempre mayor o igual a la cantidad de claves.

→ Ahora la clave puede estar en otro Hash (el hash no cambia!)

→ Si hay una colisión sigo recorriendo el array hasta encontrar el próximo espacio libre



Direccionamiento abierto no almacena varios elementos en un mismo índice. Cada espacio se llena con una sola clave o se deja en **NULL**.

Probing lineal

Se busca linealmente un espacio en la tabla. Si $\$h(k)\$$ es la función hash,

Si el índice $h(k)$ está ocupado, probamos $(h(k) + 1)$
Si $(h(k)+1)$ también está ocupado, probamos $(h(k) + 2)$
Si $(h(k) + 1)$ también está ocupado, probamos $(h(k)+3)$.
.....

Es decir, el índice aumenta linealmente hasta encontrar una celda vacía inmediata en la tabla para el nuevo elemento.

Área de desbordamiento

Dejo una zona donde voy a guardar todas las colisiones.

Se divide la tabla en dos secciones: el área principal a la que se asignan las claves y un área para colisiones

Cuando ocurre una colisión, se usa una celda de esta área para el nuevo elemento.

Diferencias entre direccionamientos:

1. El encadenamiento es más sencillo de implementar. Además la tabla Hash nunca se llana, siempre puedo agregar más elementos al encadenamiento.
 - En en direccionamiento abierto, la tabla se puede llenar

2. El encadenamiento es menos sensible a la función Hash o los factores de carga. El direccionamiento abierto requiere un cuidado especial para evitar la agrupación y el factor de carga desfavorable.
3. El encadenamiento se usa cuando se desconoce la cantidad de claves y con qué frecuencia se pueden insertar o eliminar. El direccionamiento abierto se utiliza cuando se conoce la frecuencia y el número de claves
4. El rendimiento de la caché del encadenamiento no es bueno ya que las claves se almacenan mediante una lista enlazada. El direccionamiento abierto proporciona un mejor rendimiento de la caché ya que todo se almacena en la misma tabla.
5. Desperdicio de espacio. Hay parte de la tabla de hash en el encadenamiento que nunca se utilizan. En el direccionamiento abierto se puede utilizar una celda incluso si una entrada no se asigna a ella.
6. El direccionamiento abierto requiere un cuidado especial para evitar la agrupación y el factor de carga desfavorable.

Factor de carga y rehash:

Factor de carga

$$\alpha = \frac{n}{m}, \quad 0 \leq \alpha \leq 1.$$

- n: número de claves almacenadas actualmente.
- m: capacidad de la tabla de hash.

Indica el grado de ocupación de la tabla de hash y qué tan probable es que haya una colisión. Debe mantenerse bajo de modo que el número de entradas en un hash sea pequeño y, por lo tanto, la complejidad sea casi constante O(1).

Valor es entre 0 y 1.

Rehash

Cuando $\alpha >= 0,75$ por ejemplo (lo determino el programador)

El tamaño del arreglo se aumenta (generalmente, se duplica) y a todos los valores se les vuelve a asignar un hash (un nuevo Índice en el arreglo) y se almacenan en un nuevo arreglo de tamaño mayor para mantener un factor de carga bajo y una complejidad baja.

¿Cómo se hace?

El rehashing se puede hacer de la siguiente manera:

- Para cada adición de una nueva entrada a la tabla, hay que verificar el factor de carga.
- Si es mayor o igual que su valor predefinido (o el valor predeterminado de 0,75 si no se proporciona), entonces **rehash**.
- Para **rehash**, hay que armar un nuevo arreglo del doble del tamaño anterior.

- Luego, hay que recorrer cada elemento en el antiguo arreglo e insertarlo en el nuevo arreglo mas grande.

Por ejemplo, si $m = 20$ entonces:

$$\begin{aligned} 0,75 &= \frac{n}{20} \\ 20 \cdot 0,75 &= n \\ n &= 15 \end{aligned}$$

cuando almacene la clave 15 habrá que aumentar la capacidad de la tabla y rehashear.

Recordatorio: se pueden tener colisiones rehashando. Rehashear no significa deshacerse de las colisiones.

C. OPERACIONES

- **Buscar:** siempre que se busque un elemento, hay que calcular el índice de la tabla hash de la *clave pasada* y ubicar el elemento en el arreglo. Se puede utilizar probing lineal para adelantar el elemento si no se encuentra en el índice calculado. Se sigue buscando hasta que se encuentre el elemento buscado o se encuentre una celda vacía.

- **Insertar:** siempre que se inserte un elemento, hay que calcular el índice de la tabla hash de la *clave pasada* y ubicar el elemento en el arreglo. Si el índice calculado esta ocupado, se pueden utilizar uno de los métodos mencionados arriba para manejar la colisión.

- **Eliminar:** siempre que se elimine un elemento, hay que calcular el índice de la tabla hash de la *clave pasada* y ubicar el elemento en el arreglo. Se puede utilizar probing lineal para adelantar el elemento si no se encuentra en el índice calculado. Cuando se encuentre, se puede almacenar un elemento ficticio allí para mantener intacto el rendimiento de la tabla hash o reemplazarlo con otro valor de la tabla (para ver como funciona el reemplazo ver clase teórica de Hash en minuto 2:28:44 <https://youtu.be/BGEews5U9Cc?t=8924> o un ejemplo en el apunte: <https://www.notion.so/Random-ee69875bebf744df99fac0aaa2483f8f>).

Practica Hash: <https://www.youtube.com/watch?v=X87hvHKKre0&t=4s>

12. PYTHON

A. INTRODUCCION

Características básicas:

1. Python es actualmente el lenguaje de programación multipropósito y de alto nivel más utilizado.
 1. Programación orientada a objetos
 2. Imperativa
 3. Funcional
2. Es un lenguaje más legible. Los programas suelen ser muchos mas cortos
3. Tipo dinámico: no tengo que definir el tipo de dato de la variable → el interprete mientras ejecuta sabe que tipo de dato es.
 1. Requiere más tiempo y memoria extra

4. No me tengo que ocupar de reservar/liberar memoria → lo hace el interprete
5. Me permite escribir módulos.c que puedo usar en otros programas luego.

a) Lenguaje interpretado

Un lenguaje interpretado es aquel que se ejecuta utilizando un programa de por medio llamado **interprete**, en lugar de compilar el código a lenguaje máquina que pueda comprender y ejecutar directamente una computadora (**lenguajes compilados**).

Pero, todo tiene un costo: la ejecución es más lenta ya que el intérprete ejecuta instrucción por instrucción.

Python tiene, no obstante, muchas de las características de los lenguajes compilados, por lo que se podría decir que es **semi-interpretado**. El código fuente se traduce a un pseudo-código máquina intermedio llamado **bytecode** la primera vez que se ejecuta, generando archivos .pyc o .pyo (bytecode optimizado), que son los que se ejecutarán en sucesivas ocasiones.

b) Tipado dinámico

No es necesario declarar el tipo de dato que va a contener una determinada variable, sino que su tipo de determinará en tiempo de ejecución según el tipo de valor al que se asigne y el tipo de esta variable puede cambiar si se le asigna un valor de otro tipo.

Almacena el valor en alguna ubicación de memoria y luego une ese nombre de variable a ese contenedor de memoria. Y hace que el contenido del contenedor sea accesible a través de ese nombre de variable.

```
x = 6
print(type(x))

x = 'holá'
print(type(x))

Salida:
<class 'int'>
<class 'str'>
```

Hacen ellos mismos las convenciones → ahorro mucho código

Sin embargo, disminuye la velocidad del programa porque la validación es tipos es en tiempo de ejecución o no te das cuenta de los errores de tipado hasta que el programa ejecute y me explote el error en la cara.

Nombres variables

Las variables las podemos pensar como “cajas” en donde se guarda información.

nombre	Válido
Num1	Válido
1num	Inválido: comienza con un número
_num	Válido
Num 1	Inválido: tiene un espacio
input	Inválido: es palabra reservada

Constantes

Las constantes las definimos en general al principio del programa o en un archivo aparte (más adelante veremos cómo incluir archivos a un proyecto). Por convención van completamente en mayúsculas. En el ejemplo del IVA, escribiríamos:



c) Fuertemente tipado

Significa que las variables tienen un tipo y que el tipo es importante cuando se realizan operaciones en una variable. los tipos deben ser compatibles con respecto al operando al realizar operaciones. Por ejemplo, Python permite agregar un número entero y un número de punto flotante, pero agregar un número entero a una cadena produce un error (no se puede, por ejemplo, sumar la cadena “9” y el número 8).

d) Orientado a objetos

Engloba las estructuras y objetos.

La programación orientada a objetos tiene como objetivo implementar entidades del mundo real en la programación. EL objetivo es unir los datos y las funciones que operan en ellos para que ninguna otra parte del código pueda acceder a estos excepto esa función.

i. clase

Una clase es un tipo de dato definido por el programador. Consiste en miembros de datos y funciones a los que se puede acceder y utilizar creando una instancia de esa clase.

Representa el conjunto de propiedades o métodos que son comunes a todos los objetos de un tipo, es como el plano de un objeto.

Por ejemplo un perro. Existe muchísimos perros distintos pero todos comparten ciertas propiedades como, raza, color, nombre, etc.

ii. objeto

Es una unidad básica de programación orientada a objetos y representa las entidades de la vida real.

Un objeto es una instancia de una clase; tiene:

- identidad
- estado
- comportamiento

Por ejemplo, "Perro" es un objeto de la vida real, que tiene algunas características como color, Raza, Ladrido, Dormir y Come.



iii. abstracción de datos

Es una de las características más esenciales de la programación orientada a objetos. Se refiere a proporcionar solo información esencial sobre los datos al mundo exterior, ocultando los detalles de fondo o la implementación.

- Ejemplo

Considere un ejemplo de la vida real de un hombre que conduce un automóvil. El hombre solo sabe que presionar el acelerador aumentará la velocidad del auto o aplicar los frenos parará el auto, pero no sabe cómo al presionar el acelerador aumenta la velocidad, no conoce el mecanismo interno del auto. o la implementación del acelerador, frenos, etc. en el automóvil. Eso es la abstracción.

iv. encapsulación

La agrupación de datos en una sola unidad. Es el mecanismo que une el código y los datos que manipula. Las variables o datos de una clase están ocultos de cualquier otra clase y solo se puede acceder a ellos a través de cualquier función miembro de su clase en la que están declarados.

v. herencia

La capacidad de una clase de derivar propiedades y características de otra clase. Al crear una clase, no es necesario escribir todas las propiedades y funciones una y otra vez, ya que pueden heredarse de otra clase que las posea.

Puedo reutilizar el código.

vi. polimorfismo

La capacidad de tomar más de una forma. Una operación puede presentar diferentes comportamientos en diferentes instancias, dependiendo de los tipos de datos utilizados en la operación.

- Ejemplo

Por ejemplo, una persona al mismo tiempo puede tener diferentes características, como un hombre al mismo tiempo es un padre, un esposo, un empleado. Entonces, la misma persona posee un comportamiento diferente en diferentes situaciones.

vii. enlace dinámico

El código asociado a una llamada de procedimiento no se conoce hasta el momento de la llamada en tiempo de ejecución

viii. paso de mensajes

Es una forma de comunicación en la programación orientada a objetos. Los objetos se comunican enviándose y recibiendo información entre sí. Un mensaje para un objeto es una solicitud de ejecución de un procedimiento y, por lo tanto, invocará a una función en el objeto receptor que genera los resultados deseados. El paso de mensajes implica especificar el nombre del objeto, el nombre de la función y la información que se enviará.

B. EJECUCION

a) Forma interactiva

Una forma muy utilizada de ejecutar código Python es a través de una sesión interactiva. Para iniciar una sesión interactiva de Python, simplemente se abre la terminal y luego se escribe `python3`.

A continuación, se muestra un ejemplo de cómo hacer esto en Linux:

```
$ python3
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

El mensaje estándar para el modo interactivo es `>>>`. Ahora, se puede escribir y ejecutar código Python como se desee, con el único inconveniente de que ***cuando cierra la sesión, el código desaparecerá***.

Cuando se trabaja de forma interactiva, cada expresión y declaración que escribe se evalúa y ejecuta de inmediato:

```

>>> print('Hola Mundo!')
Hola Mundo!
>>> 2 + 5
7
>>> print('Bienvenido a Python!')
Bienvenido a Python!

```

b) scripts

Creo un archivo en un editor de texto que esa .py

En el editor:

```

#La línea debajo indica qué programa se debe utilizar para
ejecutar el archivo.
#!/usr/bin/python3
print('Hola Mundo!')

```

Hay que guardar el archivo en un directorio de trabajo con el nombre hola.py.
Con el script de prueba listo hay que abrir la terminal y:

```

$ python3 hola.py
Hola Mundo!

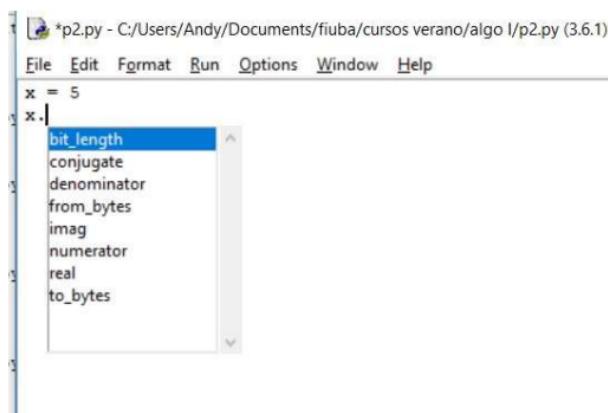
ipython3 --pylab #VISUAL STUDIO

```

C. TIPOS DE DATO

a) Números enteros

Al escribir `x = 5`, asume que `x` es una variable de tipo entera y la maneja como tal. En realidad, lo habíamos mencionado, no es una variable, sino un objeto. Hagamos esta prueba: escribimos `x = 5`, y debajo escribimos `x.`, un punto, y esperamos un par de segundos. Vemos:



La lista que aparece al costado son todos los métodos de los que disponemos. ¿Qué son los métodos? Son funciones que tiene el objeto, y que se pueden

utilizar. Por ejemplo, si escribimos `bit_length` y lo imprimimos, nos indica cuántos bits necesita para guardar el dato que tiene.

b) Complejos

Para crear un número complejo hay que poner `j` o `J` después de un número.

```
a = 2+3j  
b = -2j  
c = 0j
```

En general, el método `complex()` toma dos parámetros:

- `real` - parte real. Si se omite `real`, el valor predeterminado es 0.
- `imag` - parte imaginaria. Si se omite `imag`, el valor predeterminado es 0.

```
z = complex(2, -3)  
print(z) #Salida: (2-3j)  
  
z = complex(1)  
print(z) #Salida: (1+0j)  
  
z = complex()  
print(z) #Salida: 0j  
  
z = complex('5-9j')  
print(z) #Salida: (5-9j)
```

c) Cadenas

Si escribimos `x = "Juan"` nos dirá que es de tipo `string`. Nótese que no existe el tipo `char`. Se puede probar haciendo `x = 'a'`, que también lo tomará como un `string`.

Python permite la indexación negativa de sus secuencias. El índice `-1` se refiere al último elemento, `-2` al penúltimo elemento y así sucesivamente. Se puede acceder a un rango de elementos en una cadena usando el operador de corte : (dos punto

```
str = 'programar'  
  
print('str = ', str) # Salida: programar  
print('str[0] = ', str[0]) # Salida: p  
print('str[-1] = ', str[-1]) # Salida: r  
print('str[1:5] = ', str[1:5]) # Salida: rogr  
print('str[5:-2] = ', str[5:-2]) # Salida: am
```

Las cadenas **son inmutables**. Esto significa que los elementos de una cadena no se pueden cambiar una vez asignados. Simplemente se puede reasignar diferentes cadenas al mismo nombre. Tampoco se puede borrar ni eliminar caracteres de una cadena. Eliminar la cadena por completo es posible usando la palabra clave `del`.

d) Booleanos

Puede tener solo dos valores: `true` (1) o `false` (0)

e) Listas

Se crea una lista colocando todos los elementos entre corchetes. Puede tener cualquier número de elementos y pueden ser de diferentes tipos

```
# lista vacia
lista = []
# lista de enteros
lista = [1, 2, 3]
# lista mixta
lista = [1, "Hola", 3.4]
# lista con otra lista
lista = ["Hola", [8, 4, 6], ['a']]
```

Se puede usar el operador de índice [] para acceder a un elemento en una lista. A las listas anidadas se accede mediante la indexación anidada.

Las listas se pasan por referencia en una función, por lo tanto, si esta variable lista la pasamos a una función en donde la modificamos, la lista queda cambiada. Pero hacía falta aclarar algo más

- lista se encuentra en alguna dirección, supongamos en la 1000.
- Pero los valores que tiene: 123, “juan perez” y 18000 no están en esa dirección. Es decir, no están adentro de lista, se encuentran en un lugar aparte. Por ejemplo, en la dirección 2000.
- En la dirección 1000, lo que hay, entre otras cosas, es la dirección 2000

i. Funcionalidades

```
notas = []
notas.append(3) #agrego un 3 a mi lista
notas.count(3) #devuelve la cantidad de numeros 3 que hay en mi lista
notas.insert(4,7) #inserta un 7 en la posicion 4
suma = sum(notas) #suma de todos los numeros
cantidad = len(notas) #cantidad de elementos
copy #copia en la lista
index #devuelve la posicion en la que esta cierto elemento
```

Ordenamiento

Si quisieramos tener la lista ordenada, lo podemos hacer de dos maneras distintas:

1. Utilizando la función sorted que no modifica a la lista original, sino que devuelve una nueva lista ordenada.
2. Usando el método sort que sí modifica la lista original y no devuelve nada.

```
Usando sorted
notas_ordenadas = sorted(notas)
Usando sort
notas.sort()
```

Tanto sorted como sort ordenan, por defecto, de menor a mayor. Para ordenarlo de mayor a menor se puede ingresar un nuevo parámetro: reverse = True.

```
notas.sort(reverse = True)
```

Quitar

- Con el método clear quita todos los elementos y deja la lista vacía.
- Con remove quita un valor de la lista. Por ejemplo remove(3) quitaría una nota 3. Si hay más de una solo quita la primera. Si no hay ninguna da error, por eso este método debe usarse luego del método count. Si hay cantidad positiva, entonces se puede remover, de lo contrario, no.
- Con pop quita el elemento que esté en la posición indicada. Por ejemplo pop(2) quitaría el elemento que está en la posición 2. Si no le pasamos ningún parámetro, por defecto quita el último elemento de la lista.

Extend

```
notas_1 = [5, 4, 9, 7]
notas_2 = [2, 2, 4]
#Si queremos unir las dos listas debemos usar el método extend.
notas_1.extend(notas_2)
print(notas_1)
#la salida
[5, 4, 9, 7, 2, 2, 4]
```

Si usábamos append la salida iba a ser

```
[5, 4, 9, 7, [2, 2, 4]]
```

Agrega la segunda lista como un solo elemento. En lugar de agregar 3 elementos agrega 1 solo.

Recorriendo una lista

Lo más probable es que tengamos que recorrer una lista elemento por elemento para hacer cálculos o buscar algún dato, entre otras cosas.

UNO

Lo primero que podríamos pensar es hacer un rango desde 0 hasta la última posición de la lista, en donde vamos obteniendo cada uno de los elementos. Esto lo hacíamos poniendo el nombre de la lista y entre corchetes la posición. De esta forma recorremos toda la lista y la imprimimos elemento por elemento.

```
for i in range(len(notas)):
```

```
print(notas[i])
```

- Al no indicar la posición de arranque, por defecto da la posición 0
- `len(notas)` devuelve el tamaño de la lista. Por ejemplo, si la lista tiene 8 elementos, la última posición es la 7, pero hay que recordar que en `range` el límite superior lo deja afuera.

DOS

Recorrer una lista usando iteradores. En este caso nota es un iterador que va tomando los elementos de la lista de a uno por vez.

```
for nota in notas:  
    print(nota)
```

Listas + strings

Hay ciertas funciones y métodos que convierten un string en una lista y viceversa.

Split

```
s = "hola que tal"  
l = s.split()  
['hola', 'que', 'tal']  
  
#usando otro separador  
  
s = "hola,que,tal"  
l = s.split(',')  
['hola', 'que', 'tal']
```

Ordenar alfabéticamente

```
s = "camino"  
l = sorted(s)  
print(l)  
#En este caso sorted devuelve una lista ordenada  
alfabéticamente:  
['a', 'c', 'i', 'm', 'n', 'o']
```

Si quiero armar un string con esa lista. Armamos un string vacío, recorremos la lista y le vamos agregando con el + cada letra. El nuevo string queda con las letras de “camino” ordenadas en forma alfabética:

```
s2 = ""  
for letra in l:  
    s2 = s2 + letra  
acimno
```

Hay una forma de unir una lista de un string usando `join`

```
#arma un string vacío y la instrucción join indica que debe  
unir el parámetro  
#con ese string, como es vacío solo une los elementos de la  
lista.
```

```
s2 = "".join(l)

#con guiones
s3 = "-".join(l)
a-c-i-m-n-o
```

f) Tuplas

La genero colocando todos los elementos entre paréntesis, separados por comas.

En las tuplas no puedo modificar los valores.

```
tupla = (123, "juan perez", 18000)
```

puede representar una tupla (un registro) de un trabajador (“juan perez”) cuyo legajo es el 123, y tiene un sueldo de \$18.000.

Cada campo se lo referencia por una posición: la primera es la 0 (cero), la segunda la 1 (uno), etc.

```
print(tupla[0])
imprime 123.
```

Pero no podemos modificar ningún campo. Es decir, los valores son fijos. Por ejemplo, si queremos modificarle el sueldo y escribimos:

```
tupla[2] = 20000 #nos da un error
```

Podemos usar el operador de índice [] para acceder a un elemento en una tupla, donde el índice comienza desde 0.

g) Diccionarios

El diccionario de Python es una colección desordenada de elementos. Cada elemento de un diccionario tiene un par clave / valor. Los diccionarios están optimizados para devolver valores cuando se conoce la clave.

Crear un diccionario es tan simple como colocar elementos entre llaves {} separadas por comas. Un elemento tiene una clave y un valor correspondiente que se expresa como un par (clave: valor). Si bien los valores pueden ser de cualquier tipo de dato y pueden repetirse, las claves deben ser de tipo inmutable (cadena, número o tupla con elementos inmutables) y deben ser únicas.

```
# diccionario vacio
dicc = {}
# diccionario con claves enteras
dicc = {1: 'manzana', 2: 'pelota'}
# diccionario con claves mixtas
dicc = {'nombre': 'Lucas', 1: [2, 4, 3]}
```

Si usamos los corchetes [], KeyError se genera en caso de que no se encuentre una clave en el diccionario. Por otro lado, el método get() devuelve None si no se encuentra la clave.

D. CONSTRUCTORES

Existen distintos tipos de constructores

i. Constructor predeterminado

Es un constructor simple que no acepta ningún argumento. Su definición tiene solo un argumento que es una referencia a la instancia que se está construyendo.

```
class Profesor:  
  
    # constructor predeterminado  
    def __init__(self):  
        self.nombre = "Manuel"  
  
    # un método para imprimir miembros de datos  
    def print_Profesor(self):  
        print(self.nombre)  
  
    # creando objeto de la clase  
obj = Profesor()  
# llamar al método de instancia usando el objeto obj  
obj.print_Profesor()
```

¿Qué poronga es self?

Self es "si mismo". Es como que python agarra y con self se encarga de reservar memoria y todo cuando pongo self.

Siempre usá self como el nombre para el primer argumento en los métodos

ii. Constructor parametrizado

El constructor parametrizado toma su primer argumento como una referencia a la instancia que se está construyendo conocida como `self` y el resto de los argumentos son proporcionados por el programador.

E. METODOS

Un método es como una función de Python, pero debe llamarse en un objeto y, para crearlo, debe colocarse dentro de una clase. Como una función, un método tiene un nombre, puede tomar parámetros y tener una declaración de retorno.

i. `init`

El método `__init__` se utiliza para inicializar el estado del objeto cuando se crea una instancia de la clase.

```
class Persona:  
  
    def __init__(self, nombre):
```

```

        self.nombre = nombre

    def saludar(self):
        print('Hola, mi nombre es ', self.nombre)

p = Persona('Nico')
p.saludar()

```

En el ejemplo anterior, se crea un nombre de persona Nico. Al crear una persona, se pasa "Nico" como argumento, este argumento se pasará al método `__init__` para inicializar el objeto.

ii. self

`self` representa la instancia de la clase. Usando la [palabra clave](#) "self" podemos acceder a los atributos y métodos de la clase en Python. Vincula los atributos con los argumentos dados.

`self` es un parámetro en la función y el usuario puede usar otro nombre de parámetro en su lugar, pero es recomendable usar `self` porque aumenta la legibilidad del código.

```

class Auto():

    # método init o constructor
    def __init__(self, modelo, color):
        self.modelo = modelo
        self.color = color

    def mostrar(self):
        print ("El modelo es", self.modelo)
        print ("El color es", self.color)

audi = Auto("audi a4", "azul")
ferrari = Auto("ferrari 488", "verde")

audi.mostrar() # la misma salida que car.mostrar(audi)
ferrari.mostrar() # la misma salida que car.mostrar(ferrari)

```

iii. Input

Es como un `scanf()` en C. Esta función primero toma la entrada del usuario y luego evalúa la expresión, lo que significa que Python identifica automáticamente si el usuario ingresó una cadena, un número o una lista. Si la entrada proporcionada no es correcta, Python genera un error de sintaxis o una excepción.

```

val = input("Ingrese un valor: ")
print(val)

```

Lo que sea que ingrese como entrada, la función `input()` lo convierte en una cadena. Si se ingresa un valor entero, aún la función lo convertirá en una cadena. Hay que [convertirlo explícitamente](#) en un número entero en el código usando la conversión de tipos. El siguiente código toma dos entradas de la consola, las convierte en un número entero/floatante y luego imprime la suma.

```
num1 = int(input())
num2 = int(input())
print(num1 + num2)

num1 = float(input())
num2 = float(input())
print(num1 + num2)
```

F. FUNCIONES

Para realizar programas modulares como decíamos al principio necesitamos definir funciones que hagan ciertas tareas.

Para definir una función, simplemente ponemos la palabra `def` seguida del nombre de la función y una lista de parámetros que podría estar vacía. Debemos distinguir dos momentos: uno es el que la función se define (único), otro es cuando se llama (pueden ser uno o más llamados).

```
# Sintaxis
def funcion(parametros):
    expresion(s)
```

Arriba se muestra una definición de [función](#) que consta de los siguientes componentes.

1. Definición de palabra clave que marca el inicio del encabezado de la función.
2. Un nombre de función para identificar de forma exclusiva la función. La denominación de funciones sigue las mismas reglas de escritura de [identificadores](#) en Python.
3. Parámetros a través de los cuales pasamos valores a una función. Son opcionales.
4. Dos puntos (`:`) para marcar el final del encabezado de la función.
5. Una o más declaraciones de Python válidas que componen el cuerpo de la función. Las declaraciones deben tener el mismo nivel de indentación.
6. Una declaración de retorno opcional para devolver un valor de la función.

Una vez que hemos definido una función, podemos llamarla desde otra función, programa o incluso desde el indicador de Python. Para llamar a una función, simplemente escribimos el nombre de la función con los parámetros apropiados.

i. Función lambda (anónima)

En Python, una función anónima es una función que se define sin un nombre. Mientras que las funciones normales se definen usando la palabra clave `def` en Python, las funciones anónimas se definen usando la palabra clave `lambda`. Por lo tanto, las funciones anónimas también se denominan funciones `lambda`.

Una [función lambda](#) en Python tiene la siguiente sintaxis:

```
lambda argumentos: expresion
```

Las funciones Lambda pueden tener cualquier número de argumentos, pero solo una expresión. La expresión se evalúa y se devuelve.

```
double = lambda x: x * 2
print(double(5)) # Salida: 10

# expresion equivalente
def double(x):
    return x * 2
```

Las funciones Lambda pueden pasarse por parámetro al igual que funciones definidas con `def` (ver ejemplo en <https://youtu.be/wSkZwGzsPlc?t=3201>).

Ejemplo

Tengo una lista:

```
lista = [1,2,3,4,5,0]
min(lista) -> 0 #funcion que me devuelve el minimo

#puedo pasarle a minimo una función que evalue cada elemento
min(lista,key=lambda i: -i) -> 5
```

key es la forma de decirle para cada uno de estos elementos de la lista, la clave que vas a usar para comparar no es elemento mismo sino que a cada elemento aplícale la función y lo que te devuelva la función es lo que tenes que comparar.

En lugar de comparar 1 - 2 - 3 - 4 - 5 - 0 y quedarse con el 0, lo que va a comparar es el resultado de aplicarle una función a cada elemento

En este caso compara -1, -2, -3, -4, -5 y 0 y se queda con -5 que es el menor.

ii. Import

A medida que nuestro programa va creciendo, es una buena idea dividirlo en diferentes módulos. Un **módulo** es un archivo que contiene definiciones y declaraciones de Python. Los módulos de Python tienen un nombre de archivo y terminan con la extensión `.py`.

Las definiciones dentro de un módulo se pueden importar a otro módulo o al intérprete interactivo en Python. Usamos la palabra clave de importación para hacer esto. Por ejemplo, podemos importar el módulo matemático escribiendo la siguiente línea:

```
import math  
print(math.pi) # Salida: 3.141592653589793
```

Ahora todas las definiciones dentro del módulo `math` están disponibles.

También podemos importar solo algunos atributos y funciones específicos, utilizando la palabra clave `from`. Por ejemplo:

```
>>> from math import pi  
>>> pi  
3.141592653589793
```

iii. Control del flujo

- **if**
if evaluar expresion:
 declaracion(s)
- **if...else**
if evaluar expresion:
 declaracion(s)
else:
 declaracion(s)
- **if...elif...else**
if evaluar expresion:
 declaracion(s)
elif evaluar expresion:
 declaracion(s)
else:
 declaracion(s)
- **for**
for val in secuencia:
 declaracion(s)

Aquí, `val` es la variable que toma el valor del elemento dentro de la secuencia en cada iteración. El bucle continúa hasta que llega al último elemento de la secuencia.

Se puede generar una secuencia de números usando la función `range()`. Por ejemplo, `range(10)` generará números del 0 al 9 (10 números). Por lo tanto, se puede usar la función `range()` en bucles `for` para iterar a través de una secuencia de números combinada con la función `len()` para iterar a través de la secuencia usando indexación. Aquí hay un ejemplo.

```
genre = ['pop', 'rock', 'jazz']  
  
for i in range(len(genre)):  
    print("I like", genre[i])
```

Un bucle `for` también puede tener un bloque `else` opcional. La parte `else` se ejecuta si los elementos de la secuencia utilizada en el bucle `for` se agotan. La palabra clave `break` se puede utilizar para detener un bucle `for`. En tales casos, se ignora la parte `else`.

- **while**

```
while evaluar expresion:  
    declaracion(s)
```

Al igual que con los bucles `for`, los bucles `while` también pueden tener un bloque `else` opcional. La parte `else` se ejecuta si la condición en el ciclo `while` se evalúa como `False`. El ciclo `while` se puede terminar con una declaración `break`. En tales casos, se ignora la parte `else`. Por lo tanto, la parte `else` de un ciclo `while` se ejecuta si no se produce ninguna interrupción y la condición es falsa.

G. EJEMPLO

```
#defino una funcion en una linea(funcion anonima, sin nombre). No la  
puedo invocar nunca ya que no tiene nombre  
#no puede tener mas de una linea  
mi_funcion = lambda elemento,aux: print(elemento) #se la asigno a una  
variable, la puedo pasar por parametro  
  
contador = 0  
  
#ubo un error aca wtf  
def incrementar():  
    contador += 1  
  
def incrementarv2():  
    global contador  
    contador += 1  
  
#cortes comerciales. Puedo usar funciones declaradas mas abajo, a  
python no le importa  
#con que declare la funcion antes de usarla  
def funcion2(n):  
    return funcion1(n)+1  
  
def funcion1(n):  
    return 2*n  
  
def crear_funcion():  
    def funcion_auxiliar(n): #funcion creada dentro, es local  
        return n*2  
  
    return funcion_auxiliar  
  
#cuando invoco a crear_funcion, me devuleve la funcion auxiliar  
mi_funcion = crear_funcion()  
mi_funcion(2) #me va a devolver 2*2  
  
def crear_incremetador(n):  
    #closure: capturo una variable en mi entorno  
    def incrementar():
```

```

nonlocal n #cada vez que llame a mi incrementador
tengo un numero distinto
n += 1
return n
def obtener():
    return n
return [obtener,incrementar]

```

H. CHEATSHEET

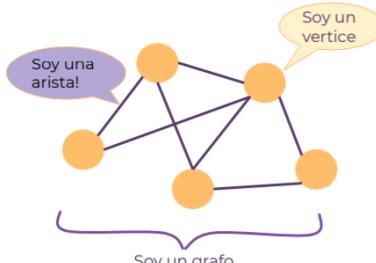
Main data types		List operations		List methods	
<code>boolean = True / False</code>		<code>list = []</code>	defines an empty list	<code>list.append(x)</code>	adds x to the end of the list
<code>integer = 10</code>		<code>list[i] = x</code>	stores x with index i	<code>list.extend(L)</code>	appends L to the end of the list
<code>float = 10.01</code>		<code>list[i]</code>	retrieves the item with index i	<code>list.insert(i,x)</code>	inserts x at i position
<code>string = "123abc"</code>		<code>list[-1]</code>	retrieves last item	<code>list.remove(x)</code>	removes the first list item whose value is x
<code>list = [value1, value2, ...]</code>		<code>list[i:j]</code>	retrieves items in the range i to j	<code>list.pop(i)</code>	removes the item at position i and returns its value
<code>dictionary = { key1:value1, key2:value2, ... }</code>		<code>del list[i]</code>	removes the item with index i	<code>list.clear()</code>	removes all items from the list
Numeric operators		Dictionary operations		String methods	
<code>+</code> addition	<code>==</code> equal	<code>dict = {}</code>	defines an empty dictionary	<code>string.upper()</code>	converts to uppercase
<code>-</code> subtraction	<code>!=</code> different	<code>dict[k] = x</code>	stores x associated to key k	<code>string.lower()</code>	converts to lowercase
<code>*</code> multiplication	<code>></code> higher	<code>dict[k]</code>	retrieves the item with key k	<code>string.count(x)</code>	counts how many times x appears
<code>/</code> division	<code><</code> lower	<code>del dict[k]</code>	removes the item with key k	<code>string.find(x)</code>	position of the x first occurrence
<code>**</code> exponent	<code>>=</code> higher or equal			<code>string.replace(x,y)</code>	replaces x for y
<code>%</code> modulus	<code><=</code> lower or equal			<code>string.strip(x)</code>	returns a list of values delimited by x
<code>//</code> floor division				<code>string.join(L)</code>	returns a string with L values joined by string
Boolean operators		Dictionary methods		String methods	
<code>and</code> logical AND	<code>#</code> coment	<code>dict.keys()</code>	returns a list of keys	<code>string.format(x)</code>	returns a string that includes formatted x
<code>or</code> logical OR	<code>\n</code> new line	<code>dict.values()</code>	returns a list of values		
<code>not</code> logical NOT	<code>\<char></code> scape char	<code>dict.items()</code>	returns a list of pairs (key,value)		
String operations		<code>dict.get(k)</code>	returns the value associated to the key k		
<code>string[i]</code>	retrieves character at position i	<code>dict.pop()</code>	removes the item associated to the key and returns its value		
<code>string[-1]</code>	retrieves last character	<code>dict.update(D)</code>	adds keys-values (D) to dictionary		
<code>string[i:j]</code>	retrieves characters in range i to j	<code>dict.clear()</code>	removes all keys-values from the dictionary		
		<code>dict.copy()</code>	returns a copy of the dictionary		

Legend: `x,y` stand for any kind of data values, `s` for a string, `n` for a number, `L` for a list where `i,j` are list indexes, `D` stands for a dictionary and `k` is a dictionary key.

13. GRAFOS

A. INTRODUCCION

Un grafo es un par ordenado $G = (V, E)$ donde:
- V es un conjunto de vértices (o nodos)
- E es un conjunto de aristas (o arcos)



i. Aplicaciones

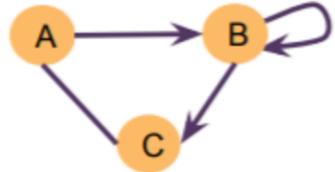
- Representar el flujo de cálculo o almacenar datos
- Google Maps usa grafos para construcción de sistemas de transporte, donde la intersección de dos (o más) carreteras se considera un vértice y la carretera que conecta dos vértices se considera una arista. El sistema de

- navegación se basa en el algoritmo para calcular el camino más corto entre dos vértices
- Facebook: usuarios son vértices y si son amigos, entonces hay una arista.

- Grafo dirigido

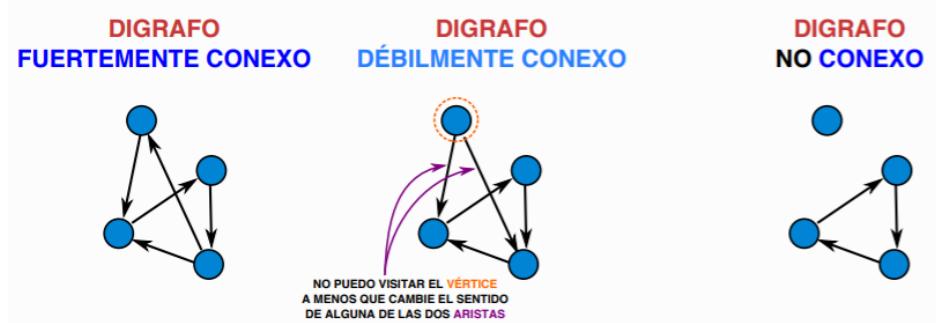
Si sus **aristas** tienen sentido.

Con que al menos una sea unidireccional, ya es dirigido.



Fuertemente conexo: un par de vértices {A,B} los son si existe un camino de A hacia B y otro de B hacia A.

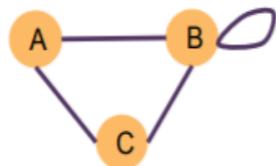
Débilmente conexo: si para lograr de ir A→B y de B→A es necesario reemplazar una o más aristas por aristas sin sentido.



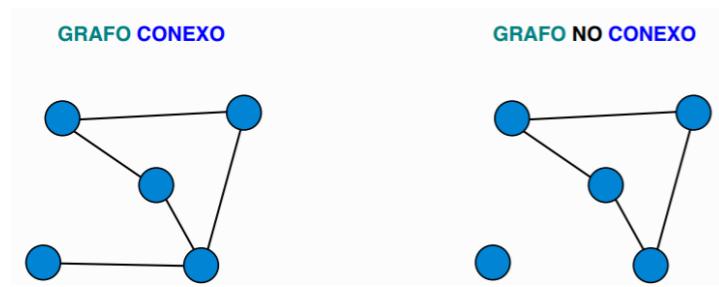
- Grafo no dirigido

Todas sus **aristas** con **bidireccionales**

→ se pueden recorrer en las dos direcciones



Componente conexa: conjunto de vértices donde existe un camino de un vértice a otro (**solo grafos no dirigidos**). Para cualquier par de vértices existe al menos un camino entre ellos.

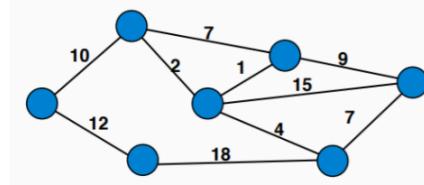


ii. Definiciones grafo:

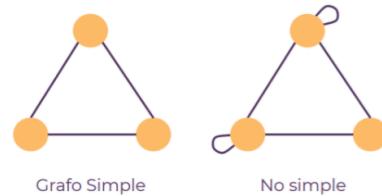
- **Orden de un grafo:** es la cantidad de vértices.
- **Tamaño de un grafo:** es la cantidad de aristas.
- **Grado de un vértice:** es la cantidad de aristas incidentes (ingresan al grafo)
- Grado de entrada: cantidad de aristas que entran al vértice
- Grado de salida: cantidad de aristas que salen de cada vértice

Grafo pesado: si sus **aristas** tienen pesos asignados.

Vendría a ser el "costo" a la hora de tener que transitar de un vértice hacia otro.



Grafo simple: es aquel que no posee aristas múltiples ni lazos. En un par de vértices hay solo una arista.

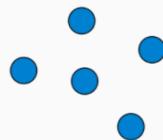


Grafo nulo: si no posee vértices ni aristas

Grafo vacío: si posee vértices pero no aristas

GRAFO NULO

GRAFO VACÍO



Densidad: un grafo denso es un grafo cuyo número de aristas está muy cerca del valor máximo de aristas que este puede tener.

$$D = \frac{2|E|}{|V|(|V|-1)}$$

- D: índice de densidad
- E: cantidad aristas
- V: cantidad vértices

Una **arista** es un bucle cuando conecta al vértice consigo mismo.

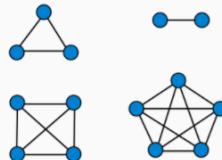
BUCLE



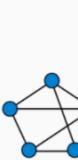
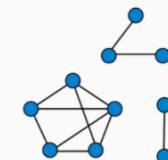
Grafo completo:

cuando contiene todas las aristas posibles.

GRAFOS COMPLETOS

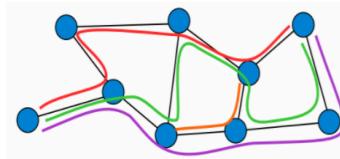


GRAFOS INCOMPLETOS



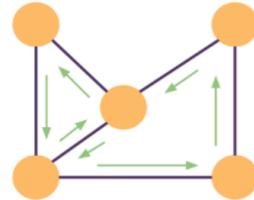
iii. Definiciones giales.:

Camino: un *recorrido a través de un grafo*. Es una secuencia de vértices (unidos por aristas). En un camino **no** puede haber **vértices repetidos**

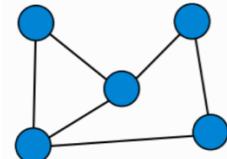


Ciclo: recorrido de aristas adyacentes que empieza y termina en el mismo lugar (pasando una sola vez por cada arista).

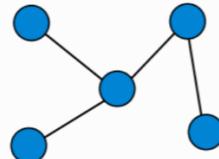
Cuando un **camino** empieza y termina en el mismo vértice.



GRAFO CON CICLOS

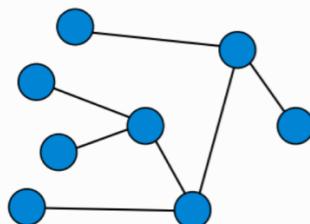


GRAFO ACÍCLICO

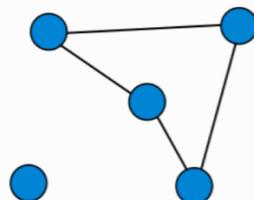


Grafo es un árbol: si es conexo y acíclico

ÁRBOL



GRAFO NO ÁRBOL



B. REPRESENTACION

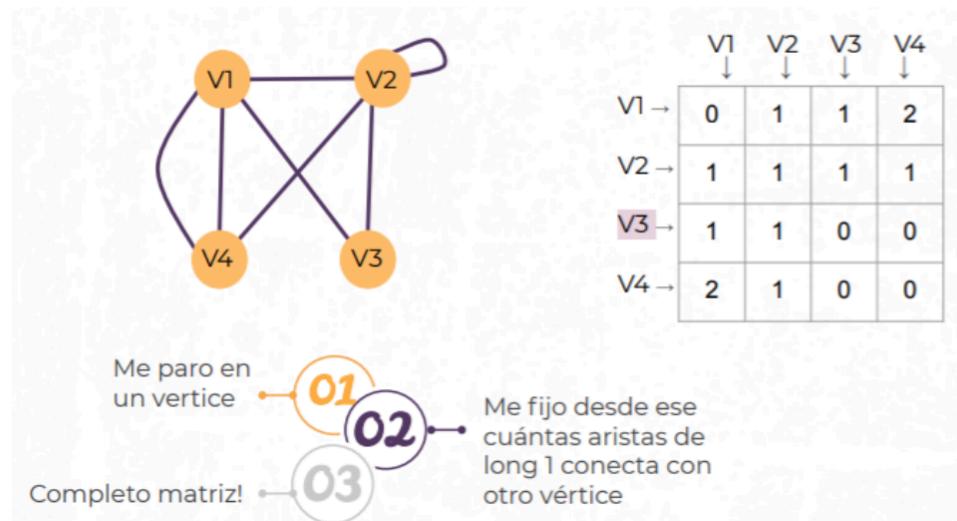
Las siguientes dos son las representaciones más utilizadas de un gráfo.

1. Matriz de adyacencia
2. Lista de adyacencia
3. Matriz de adyacencia

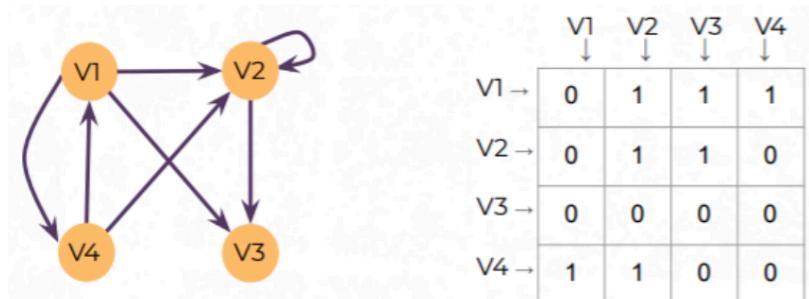
La elección de la representación del gráfo es específica de la situación. Depende totalmente del tipo de operaciones que se realizarán y la facilidad de uso.

i. Matriz de adyacencia

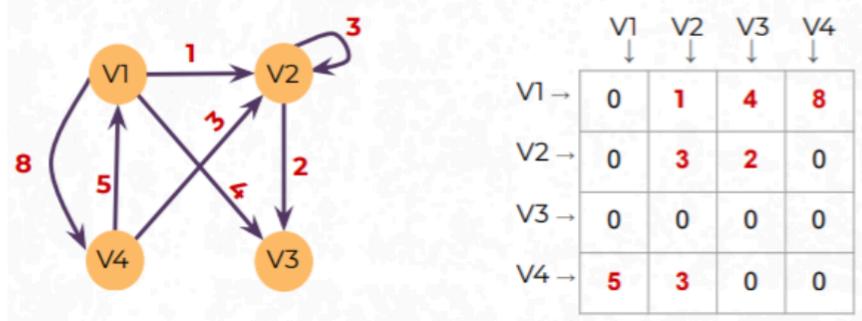
- La matriz de adyacencia para el grafo no dirigido es siempre simétrica.



- En un un grafo dirigido la matriz de adyacencia solo registra el vértice de inicio y al vértice final, $A[i][j]=1$, si y solo si la arista parte del vértice i hacia el vértice j:



- Si el grafo tiene pesos, completo con el peso



Ventajas

Las operaciones básicas como agregar una arista, eliminar una arista o verificar si hay una arista desde un vértice i a un vértice j son operaciones de tiempo constante $O(1)$.

Si el grafo es **denso**, la matriz de adyacencia debe ser la primera opción. La representación es más fácil de implementar y seguir.

Desventajas

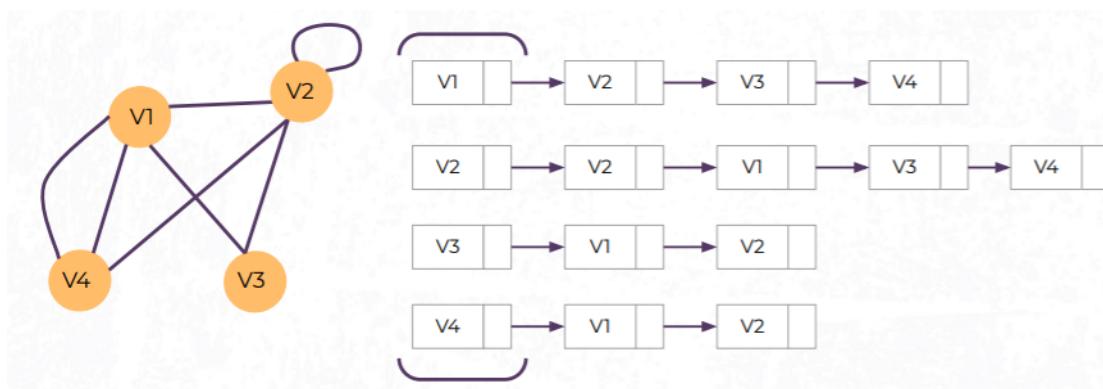
Se utiliza mucho espacio ya que hay que reservar espacio para cada enlace posible entre todos los vértices.

ii. Lista de adyacencia

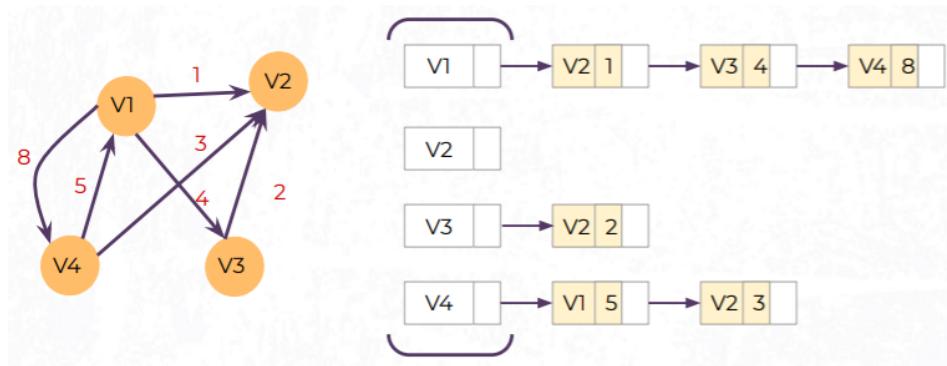
Cada arista posee una lista simplemente enlazada de a qué vértices está unida.

Una lista de adyacencia representa un grafo como un arreglo de listas vinculadas (ese arreglo, en vez de ser un arreglo, también puede ser otra lista y cada nodo de la misma, salen otras listas).

El tamaño del arreglo es igual al número de vértices. El índice del arreglo representa un vértice y cada elemento en su lista enlazada representa los otros vértices que forman una arista con el vértice.



Si el grafo es dirigido, represento las direcciones. Si tienen pesos las aristas, agrego ese dato:



Ventajas

Es eficiente en términos de almacenamiento porque solo se necesitan almacenar los valores para las aristas.

Usa espacio $O(|V| + |E|)$

Desventajas

Las consultas si hay una arista de vértice u a vértice v no son eficientes y pueden ser $O(n)$

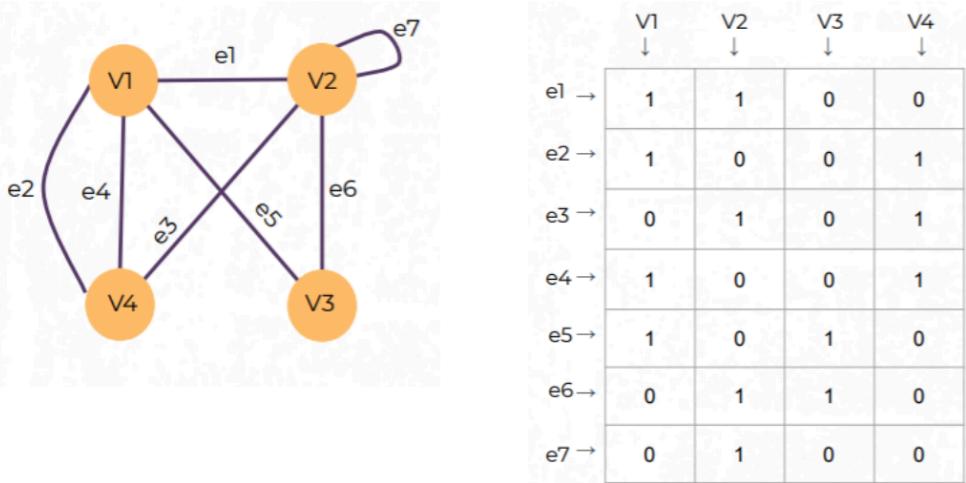
iii. Matriz de incidencia

Necesito identificar las aristas

Esta matriz de $Z^{m \times n}$ donde m es la cantidad de aristas y n la cantidad de vértices.

Para cada arista identificamos que nodos inciden con él y lo marcamos en el grafo no dirigido con un 1. Si el grafo a representar es dirigido: se le asigna -1 al vértice de salida y 1 al vértice de entrada.

Si el grafo fuera dirigido y con peso, en vez de utilizar el número 1 se utiliza el número del peso de cada arista.



C. RECORRIDOS

Proceso de búsqueda o forma de atravesar la estructura de datos de un grafo que involucra la visita de cada vértice en el grafo. El orden en el cual los vértices son visitados es la forma en que se clasifica el tipo de recorrido.

- **Visitar:** marcar como visitado el nodo.
- **Explorar:** política en la que vamos a definir cómo se comportará el algoritmo (explorar los vecinos, hijos, etc.)

Existen dos grandes recorridos: en anchura o en profundidad.

Recorrido en profundidad DFS

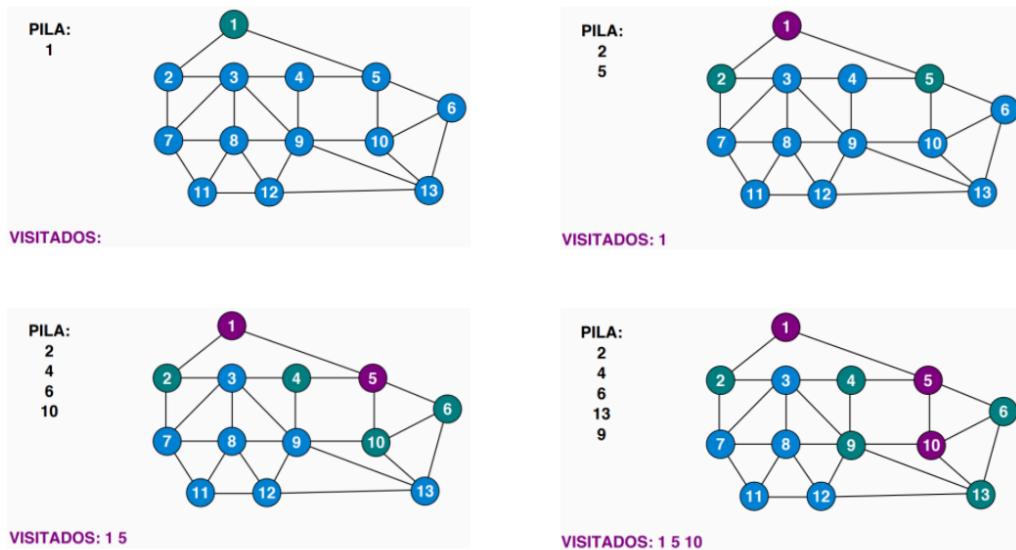
En inglés significa "Depth First Search"

Se visitan los nodos hijos primero, avanzando hasta que no se pueda continuar. Luego se "vuelve" hasta un hijo donde se tenían más caminos y se vuelve a realizar la misma lógica.

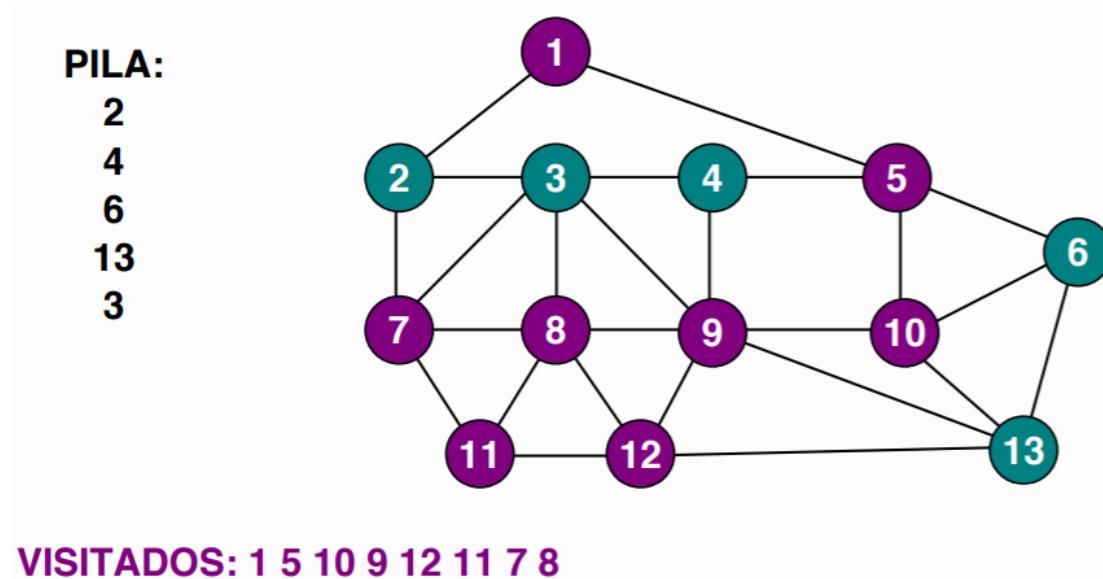
El recorrido consiste en ir recorriendo el grafo empezando desde un vértice cualquiera y, a cada paso, se visita un vértice adyacente al último visitado (se recorre lo más posible). Se continua el bucle hasta que no haya más vértices por visitar. Luego retrocede, se verifica si hay otros sin visitar y los recorre.

Mejor forma de pensarla es con una PILA

1. Apilar un vértice
2. Quitar un vértice de la pila, visitarlo
3. Apilar los vértices adyacentes al actual
4. Repetir desde 3 hasta quedarse sin vértices



Cuando ya no me quedan mas vértices por visitar, desapilo todos los elementos de la pila y los visito.



RECORRIDO DFS EN PYTHON:

```
def DFS(grafo, vertice_inicial):
    visitados = [vertice_inicial]
    pila = [vertice_inicial]

    while len(pila) > 0:
        vertice = pila.pop() #desapilo 1
```

```
print(vertice) #lo imprimo por pantalla
```

```
for ady in grafo.adyacencias(vertice):
```

```
    if ady not in visitados:
```

```
        pila.append(ady)
```

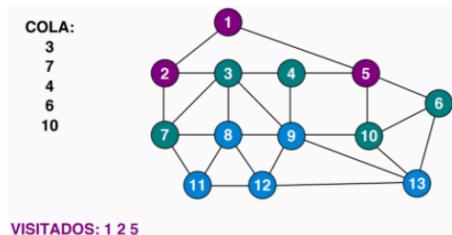
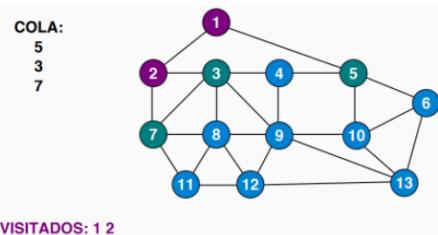
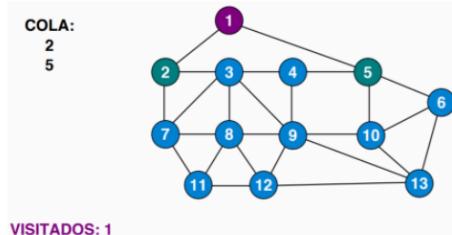
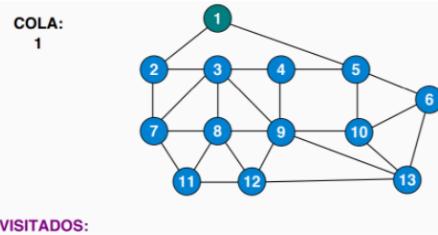
```
        visitados.append(ady)
```

Recorrido en ancho BFS

El recorrido a lo ancho consiste en ir recorriendo el grafo empezando desde un vértice cualquiera y luego se van visitando los vértices adyacentes más cercanos.

Mejor forma de pensarlo es con una COLA:

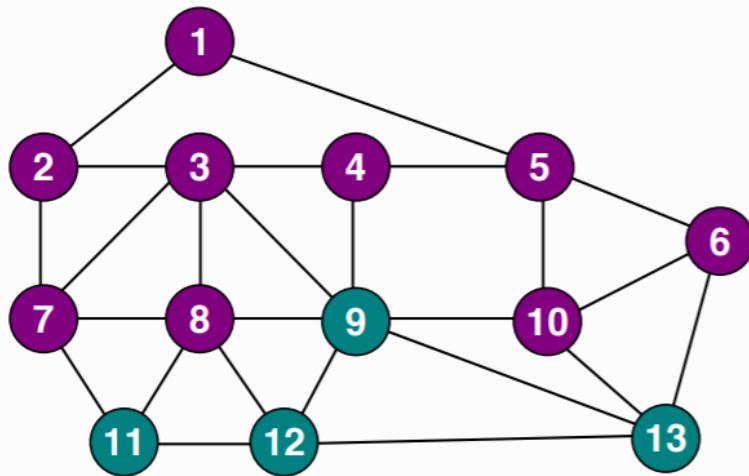
1. Encolar un vértice
2. Quitar un vértice de la cola y visitarlo.
3. Encolar los vértices adyacentes al actual
4. Repetir desde 2 hasta quedarse sin vértices



Cuando ya me quedaron todos los vértices encolados, los desencolo y visito.

COLA:

9
11
13
12



VISITADOS: 1 2 5 3 7 4 6 10 8

RECORRIDO BFS EN PYTHON:

```
def BFS(grafo, vertice_inicial):  
  
    visitados = [vertice_inicial]  
  
    cola = [vertice_inicial]  
  
    while len(cola) > 0:  
  
        vertice = cola.pop(0) #saco desde el principio de la cola  
  
        print(vertice) #lo imprimo por pantalla  
  
        for ady in grafo.adyacencias(vertice):  
  
            if ady not in visitados:  
  
                cola.append(ady)  
  
                visitados.append(ady)
```

D. ORDEN TOPOLOGICO

La idea del orden topológico es la de procesar los vértices de una grafo **acíclico** de forma tal que si el grafo contiene la arista dirigida uv entonces el nodo u aparece antes del nodo v

14. ALGOS CON GRAFOS

https://servicios.algoritmos7541mendez.com.ar/clases/11_-_diapos_grafos.pdf

ALGORITMOS PARA ENCONTRAR EL CAMINO MAS CORTO

A. DIJKSTRA

El algoritmo de Dijkstra es un algoritmo para la determinación del camino más corto, dado un vértice origen, hacia el resto de los vértices de un grafo que tiene peso en cada arista.

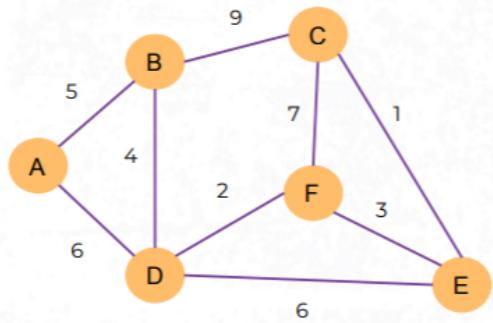
La idea consiste en ir exportando todos los caminos más cortos que parten del vértice origen y que llevan a todos los demás vértices.

Cuando se obtiene el camino más corto desde el vértice origen hasta el resto de los vértices que componen el grafo, el algoritmo se detiene.

i. Pasos

1. Elegir un vértice para comenzar.
2. Se crean dos listas de nodos, una lista de nodos Visitados y otra de nodos no visitados, que contiene a todos los nodos del grafo.
3. Se crea una tabla con 3 columnas: vértice, distancia mínima V y el nodo anterior por el cual se llegó.
4. Se toma el vértice V como vértice inicial y se calcula su distancia a sí mismo, que es 0 y la del resto de los vértices la inicializamos en ∞ .
5. Se visita el vértice NO VISITADO con menor distancia conocida desde el primer vértice V, que es el vértice con el que comenzamos ya que la distancia a ese es 0 y las demás infinito.
6. Se calcula la distancia entre los vértices sumando los pesos de cada uno con la distancia de V.
7. Si la distancia calculada de los vértices conocidos es menor a la que está en la tabla se actualiza y también los vértices desde donde se llegó.
8. Se pasa el vértice V a la lista de Vértices visitados y se continua con el vértice no visitado con menor distancia desde ese.

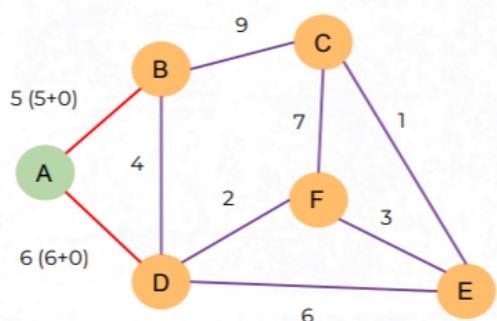
Vértice	Distancia	V. Anterior
A	0	-
B	∞	-
C	∞	-
D	∞	-
E	∞	-
F	∞	-



Visitados: []

No visitados: [A,B,C,D,E,F]

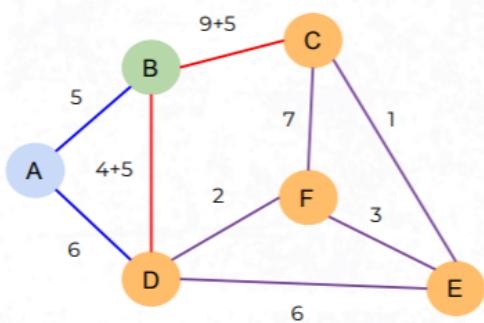
Vértice	Distancia	V. Anterior
A	0	-
B	5	A
C	∞	-
D	6	A
E	∞	-
F	∞	-



Visitados: [A]

No visitados: [B,C,D,E,F]

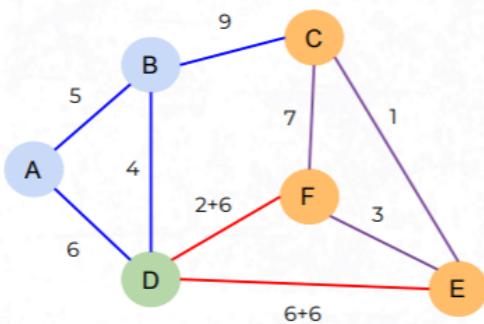
Vértice	Distancia	V. Anterior
A	0	-
B	5	A
C	14	B
D	6	A
E	∞	-
F	∞	-



Visitados: [A,B]

No visitados: [C,D,E,F]

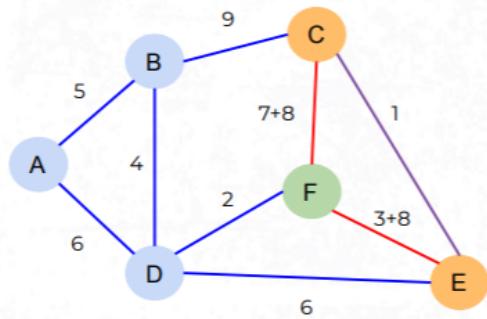
Vértice	Distancia	V. Anterior
A	0	-
B	5	A
C	14	B
D	6	A
E	12	D
F	8	D



Visitados: [A,B,D]

No visitados: [C,E,F]

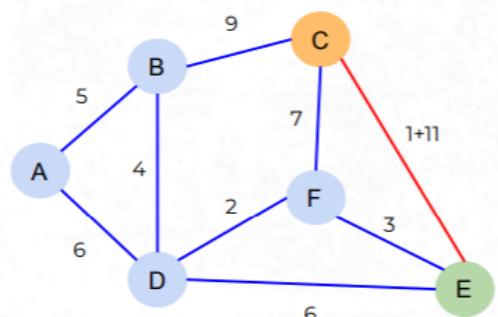
Vértice	Distancia	V. Anterior
A	0	-
B	5	A
C	14	B
D	6	A
E	11	F
F	8	D



Visitados: [A,B,D,F]

No visitados: [C,E]

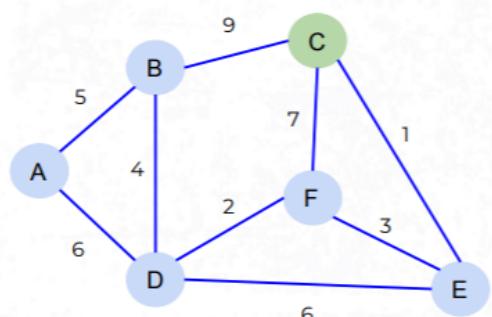
Vértice	Distancia	V. Anterior
A	0	-
B	5	A
C	12	E
D	6	A
E	11	F
F	8	D



Visitados: [A,B,D,F,E]

No visitados: [C]

Vértice	Distancia	V. Anterior
A	0	-
B	5	A
C	12	E
D	6	A
E	11	F
F	8	D

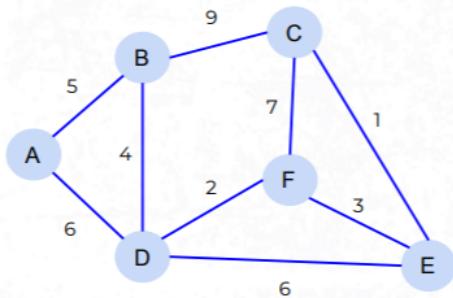


Visitados: [A,B,D,F,E,C]

No visitados: []



Vértice	Camino Minimo	Costo
A → B	A → B	5
A → C	A → D → F → E → C	12
A → D	A → D	6
A → E	A → D → F → E	11
A → F	A → D → F	8



```

def dij(grafo, vertice_inicial):
    tabla = {}
    visitados = []

    for v in grafo.vertices():
        tabla[v] = {}
        tabla[v]["distancia"] = math.inf
        tabla[v]["anterior"] = None

    tabla[vertice_inicial]["distancia"] = 0

    def minimo_no_visitado():
        no_visitados = [v for v in grafo.vertices() if v not
in visitados]
        #armo un listado de todos los no visitados

        menor = None
        distancia = math.inf
        for nv in no_visitados:
            if tabla[nv]["distancia"] < distancia:
                distancia = tabla[nv]["distancia"]
                menor = nv
        return menor

    while vertice := minimo_no_visitado():

        visitados.append(vertice)
        distancia_actual = tabla[vertice]["distancia"]

        #me fijo con que distancia llegue a ese vertice y despues de los
        #vecinos

        for ady in grafo.adyacencias(vertice):
            nueva_distancia = distancia_actual +
            grafo.distancia(vertice, ady)
            if nueva_distancia < tabla[ady]["distancia"]:
                tabla[ady]["distancia"] =
nueva_distancia
                tabla[ady]["anterior"] = vertice

    return tabla

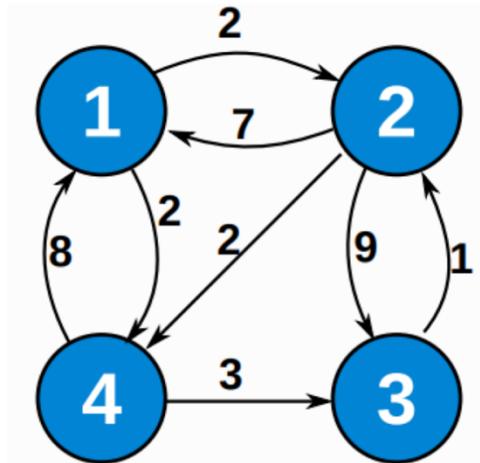
```

B. FLOYD-WARSHALL

Calcula el camino mínimo entre todos los pares de vértices. La implementación son 3 for anidados

Ejemplo: yo sé que de 1 a 3 es infinito (no tengo un camino directo), entonces es más barato:

- ir de 1 a 3 directo
- ir de 1 a 4 a 3
- ir de 1 a 2 a 3



Armo una matriz de distancias. Tomo un vértice v_0 y otro par de vértices v_1 y v_2 . Me quedo con la menor distancia entre:

- $v_1 \mapsto v_2$
- $v_1 \mapsto v_0 \mapsto v_2$

Por ejemplo, tomo $v_0 = 1, v_1 = 2, v_2 = 3$

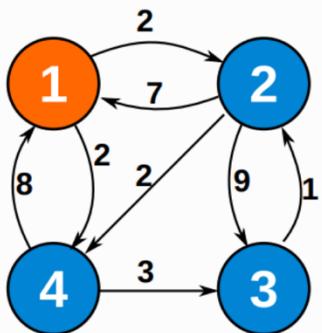
y me quedaría:

- $2 \mapsto 3 = 9$
- $2 \mapsto 1 \mapsto 3 = \text{infinito}$

Me quedo que ir directo de 2 hacia 3 es más corto

	1	2	3	4
1	0	2	∞	2
2	7	0	9	2
3	∞	1	0	∞
4	8	∞	3	0

Si los tomo de a pares a partir del 1, el 2,3,4 se van a modificar. Siempre me tengo que fijar en la matriz



	1	2	3	4
1	0	2	∞	2
2	7	0	9	2
3	∞	1	0	∞
4	8	∞	3	0

COMO 1 ES NUESTRO VERTICE INTERMEDIO, NO ME INTERESAN LAS ARISTAS QUE SALEN O LLEGAN DE EL.

	1	2	3	4
1	0	2	∞	2
2	7	0	9	2
3	∞	1	0	∞
4	8	∞	3	0

	1	2	3	4
1	0	2	∞	2
2	7	0	9	2
3	∞	1	0	∞
4	8	∞	3	0

? $2 \Rightarrow 1 \Rightarrow 3 < 2 \Rightarrow 3?$

? $2 \Rightarrow 1 \Rightarrow 4 < 2 \Rightarrow 4?$

	1	2	3	4
1	0	2	∞	2
2	7	0	9	2
3	∞	1	0	∞
4	8	∞	3	0

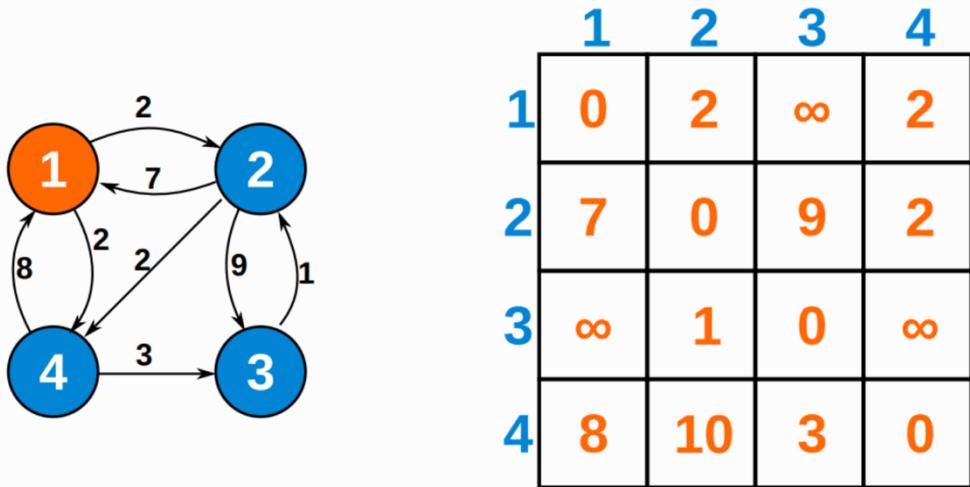
	1	2	3	4
1	0	2	∞	2
2	7	0	9	2
3	∞	1	0	∞
4	8	∞	3	0

$\dot{z}3 \Rightarrow 1 \Rightarrow 2 < 3 \Rightarrow 2?$

$\dot{z}3 \Rightarrow 1 \Rightarrow 4 < 3 \Rightarrow 4?$

	1	2	3	4
1	0	2	∞	2
2	7	0	9	2
3	∞	1	0	∞
4	8	∞	3	0

$\dot{z}4 \Rightarrow 1 \Rightarrow 2 < 4 \Rightarrow 2?$



ARBOL DE EXPANSION MINIMO

C. PRIM

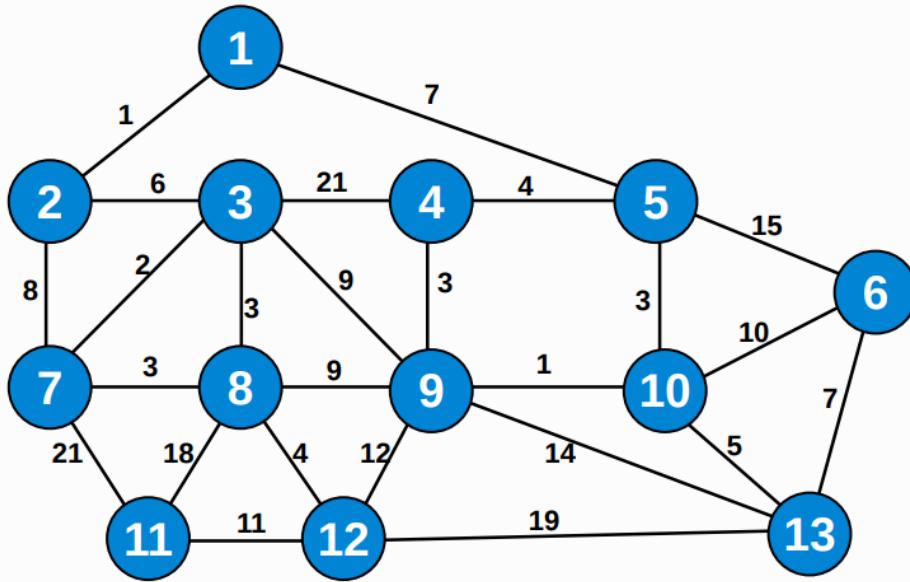
Un árbol (spanning tree) → grafo sin ciclos y conexo

Este algoritmo busca el spanning tree mínimo: al recorrer los pesos de las aristas, tenga que gastar lo menor posible para llegar de un vértice a otro. Quiero tener todos conectados entre sí con el menor peso posible.

- Forman un árbol que incluya todos los vértices
- Tiene la suma mínima de pesos entre todos los árboles que se pueden formar a partir del gráfo.

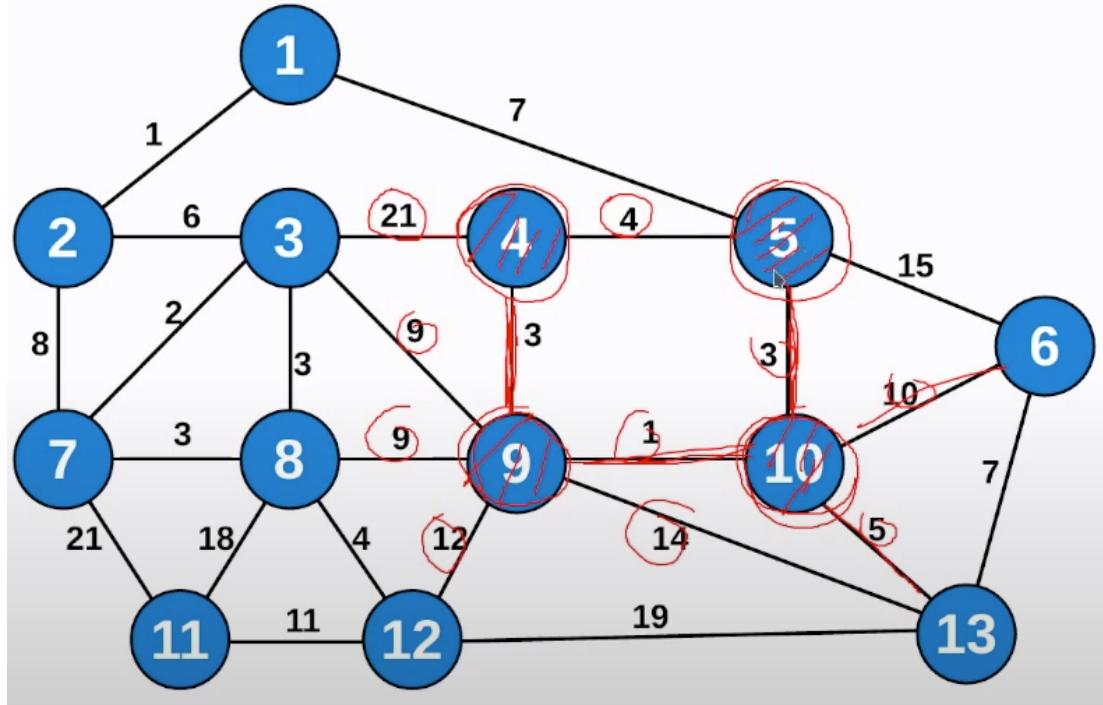
Los pasos para implementar el algoritmo de Prim son los siguientes:

1. Inicializar el árbol de expansión mínimo con un vértice elegido al azar.
2. Encontrar todas las aristas que conectan el vértice a nuevos vértices, elegir la mínima y agregarla al árbol. A medida que se agregan vértices al árbol se amplia la cantidad de aristas que podemos elegir, la única condición es que esa arista no lleve a un vértice ya agregado al arbol.
3. Seguir repitiendo el paso 2 hasta que obtengamos un árbol de expansión mínimo.



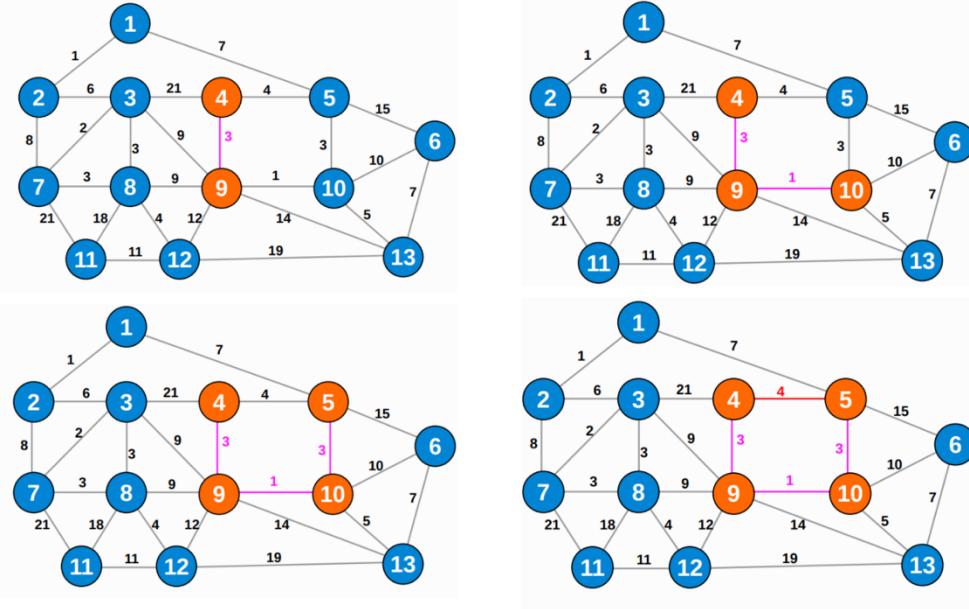
COMENZANDO POR UN VERTICE ARBITRARIO, MARCARLO COMO VISITADO E IR AGREGANDO LA ARISTA DE MENOR PESO QUE CONECTA UN VERTICE YA VISITADO CON UNO NO VISITADO.

Voy agregando al spanning tree un vértice a la vez. Cuando me quedo con la arista mínima, osea me quedo con el vértice también y repito. Empecé por el 4, me conecte al 9 después al 10 y por ultimo al 5 (esto sigue pero a modo de ejemplo). Después la mínima sería 4 (de 5 a 4) pero ya esta conectado entonces eso no lo puedo hacer.

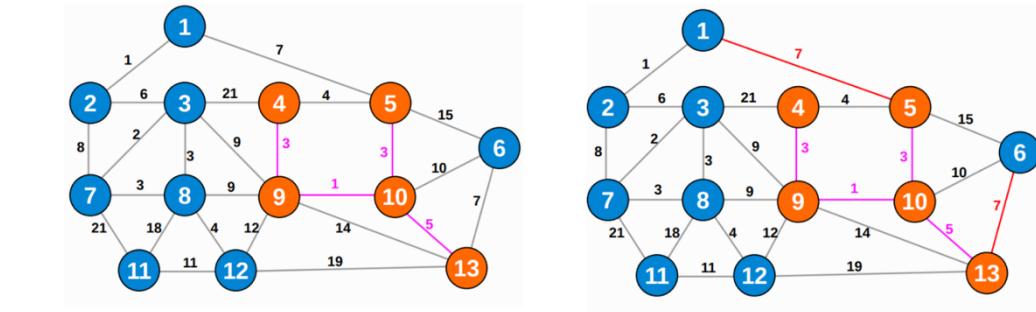


Ahora veamoslo más lindo. Lo que pasa es que empiezo con un vértice y hago crecer mi árbol agregando aristas (todas las posibles son las que tengo un vértice en mi

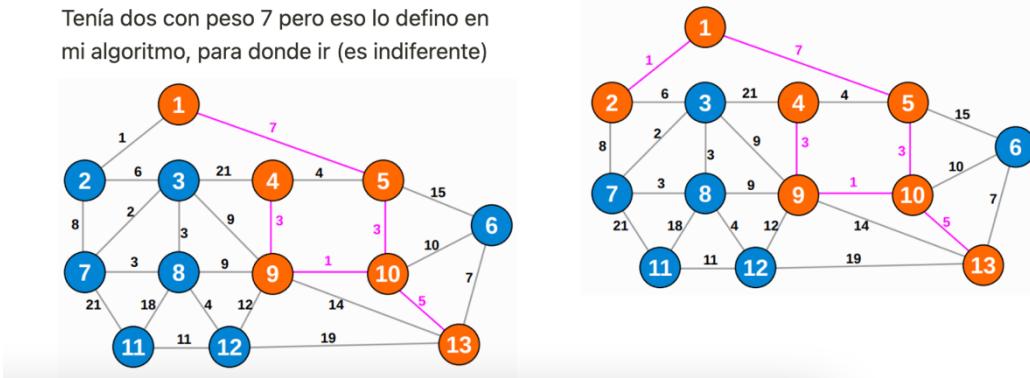
árbol)



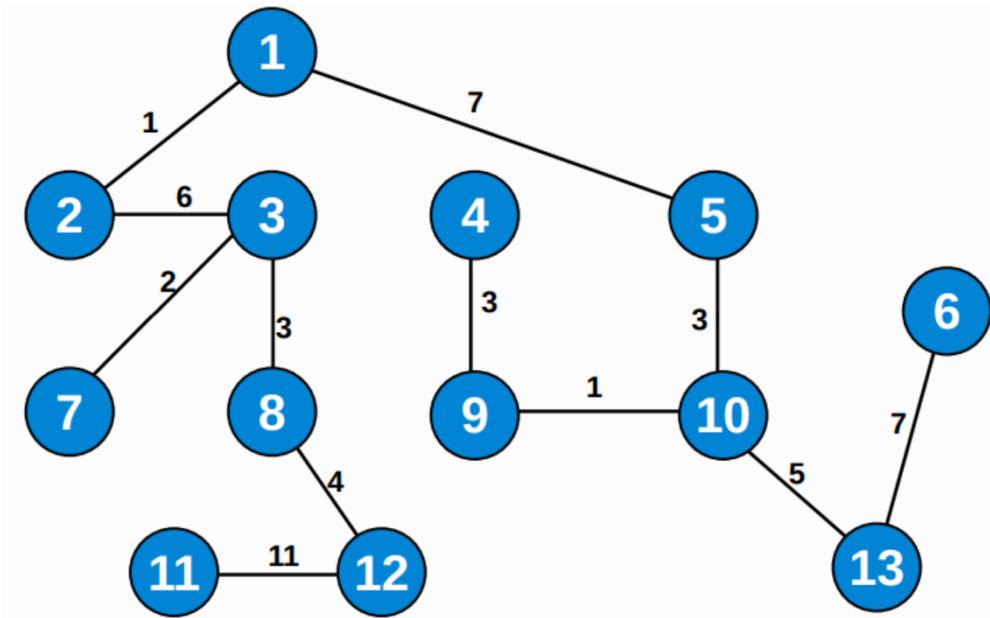
Esta arista que me genera un ciclo no la agrego, sino que agrego la que sigue menor



Tenía dos con peso 7 pero eso lo defino en mi algoritmo, para donde ir (es indiferente)



En síntesis, mi árbol queda así:

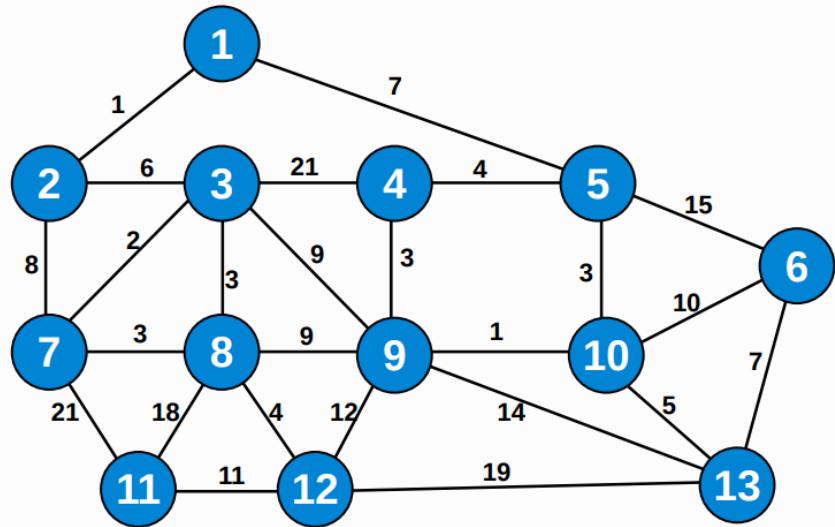


D. KRUNSKAL

En lugar de comenzar desde un vértice, el algoritmo de Kruskal ordena todos las aristas de bajo peso a alto y sigue agregando las aristas más bajas, ignorando aquellas aristas que crean un ciclo.

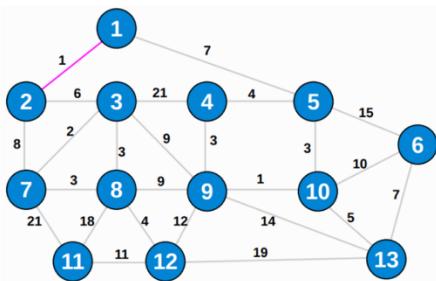
Se parte de las aristas con menor peso y se siguen agregando aristas hasta llegar al objetivo. Los pasos para implementar el algoritmo de Kruskal son los siguientes:

1. Ordenar todos las aristas de bajo peso a alto.
2. Tomar la arista con el peso más bajo y agregarlo al árbol de expansión. Si agregar la arista crea un ciclo, rechazar esta arista.
3. Seguir agregando aristas hasta llegar a todos los vértices.

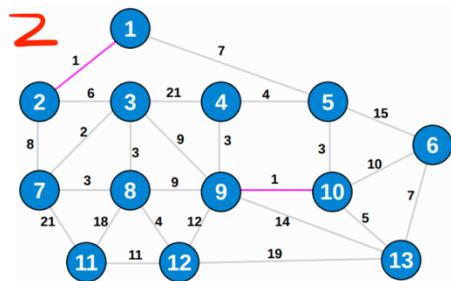


CREO UN ARBOL SEPARADO PARA CADA VERTICE.
VOY TOMANDO CADA ARISTA DE FORMA ASCENDENTE Y ME QUEDO CON LAS QUE UNEN 2 ARBOLES.

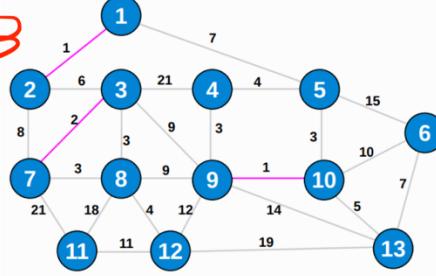
1



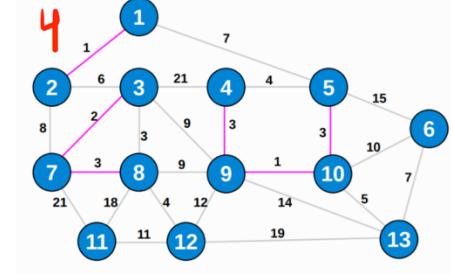
2

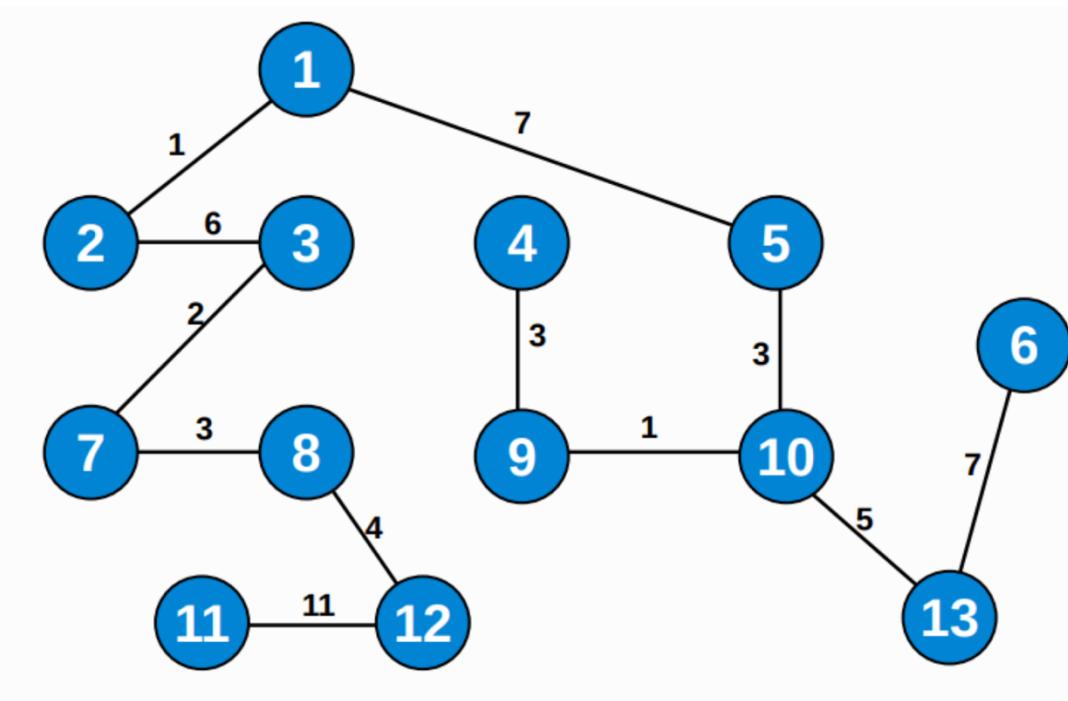


3



4





15. NOTAS- FINALES/PRACTICAS

Algoritmos 2, Curso Mendez ~ 1er Final, 2do Cuatrimestre 2021 ~ 2022-02-03

Apellido y nombre: _____

Nota final:		

Padrón: _____ Modalidad: Completo / Reducido

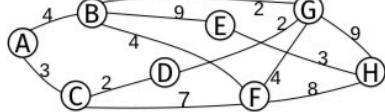
1) Explique qué es la complejidad computacional y para qué sirve. Muestre (y justifique) cómo calcular la complejidad computacional de los algoritmos **Merge sort** y **Quicksort**. Explique a qué se debe la diferencia en la complejidad de los algoritmos.

2) Dado el siguiente vector de enteros, conviértalo en un heap minimal. Muestre cada paso de la conversión y explique de cómo funciona. El algoritmo debe funcionar *in-place*.

$$V = [10, 6, 8, 1, 4, 7, 9, 2, 1]$$

3) Explique cómo funciona un árbol **AVL** y los diferentes tipos de rotaciones. ¿Qué ventaja tiene por sobre un **ABB**? Muestre paso a paso como insertar en un **AVL** vacío los elementos 4, 5, 6, 10, 8, 3 y 2 (en ese orden).

4) Explique cómo funciona el algoritmo de **Dijkstra** y aplíquelo paso a paso empezando por A:



5) Explique las diferencias y similitudes entre las tablas de Hash abiertas y cerradas. Utilice diagramas para explicar el funcionamiento de las diferentes operaciones para ambos casos.

Algoritmos 2, Curso Mendez ~ 2do Final, 2do Cuatrimestre 2021 ~ 2022-02-10

Apellido y nombre: _____

Nota final:			

Padrón: _____ Modalidad: Completo / Reducido

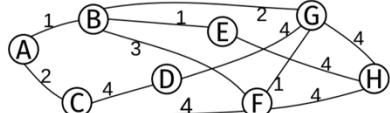
1) Explique qué es y en qué casos se aplica el **Teorema Maestro**. ¿Qué diferentes soluciones tiene? Dé un ejemplo de cada caso con su resolución.

2) Dado el siguiente vector de enteros, muestre cada paso del algoritmo **Merge sort** para ordenarlo de **mayor a menor**. Explique cómo funciona el algoritmo.

$$V = [0, 5, 1, 10, 8, 9, 4, 12, 25, 13, 2]$$

3) Explique cómo funciona un árbol **B** y sus propiedades. Cree un árbol B con 3 claves por nodo e inserte los elementos **9, 5, 1, 8, 12, 13, 15, 20**, y luego elimine **5, 1 y 20** (en ese orden). Muestre el estado del árbol en cada paso.

4) Explique cómo funciona el algoritmo de **Prim** y aplíquelo mostrando cada paso empezando por el nodo **A**:



5) Dada una tabla de **hash de direccionamiento abierto, sin zona de desborde**, con **tamaño inicial 3, factor de carga de 0.6** y función de hash $H(n) \rightarrow 3 \cdot n$:

- Explique cómo funciona esta tabla y qué significa cada característica de la misma.

- Inserte los pares **clave;valor** (mostrando cada paso): **1;A, 6;B, 3;C, 7;D, 2;E, 9;F, 1;G, 7;A**

- Elimine las claves **3, 5 y 7**

Algoritmos 2, Curso Méndez ~ 3er Final, 2do Cuatrimestre 2021 ~ 2022-02-17

Apellido y nombre: _____

Nota final:				

Padrón: _____ Modalidad: Completo / Reducido

1) Ordene de menor a mayor la complejidad computacional de los siguientes algoritmos.

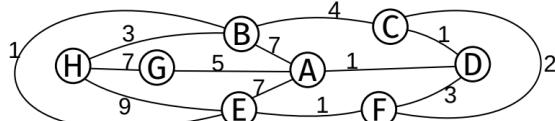
Explique cómo obtiene dicho resultado:

- Factorial
- Quicksort
- $T(n) = 4 T(n/3) + n$
- $T(n) = T(n/2) - O(1)$
- Búsqueda binaria
- $T(n) = 9 T(n/9) + O(n^9)$
- $O(n^2 \log(n))$

2) Dados los siguientes recorridos de un mismo **ABB**, reconstruyalo justificando cada paso.

Inorden=[S,N,P,C,A,T,L,D] Preorden = [C,N,S,P,D,A,T,L]

3) Explique cómo funciona el algoritmo de **Dijkstra** y aplíquelo mostrando cada paso empezando por el vértice **A**:



4) Explique cómo funciona la operación **sift-down** de un **heap binario**. Utilizando **C** o **Python**, escriba las estructuras y funciones (o clases y métodos) para implementar dicha operación.

5) Dada una tabla de **hash de direccionamiento abierto, sin zona de desborde**, con **tamaño inicial 5, factor de carga de 0.9**, crecimiento al doble y función de hash $H(n) \rightarrow n-1$:

- Explique cómo funciona esta tabla y qué significa cada característica de la misma.

- Inserte (+) y/o elimine(-) los pares **[clave;valor]** (mostrando la tabla en cada paso):

+[1;A], +[2;B], +[3;C], +[6;D], -[2], -[3], +[6;G], +[7;A], +[5;Y], +[0;M]