



Año 2022

Indice

Características	3
Primeros pasos en C++	4
Sintaxis básica	5
Tipos de datos derivados (no primitivos)	6
Canal de entrada	9
Funciones	11
Proceso de compilación	14
Memoria dinámica	15
Punteros	16
Pedido (new) y liberación (delete y nullptr) de memoria	18
Dobles punteros	22
Valgrid	23
Líneas de comandos	24
Programación estructurada	25
Programación orientada a objetos	26
Vector dinámico	32
Herencia	34
Polimorfismo	37
Diagrama de clases con UML	39
Templates	48
Estructuras lineales	56
Complejidad temporal	63
Recursión	68
Divide y vencerás	71
Árboles	73
Grafos	91
Hashing	96



Características

El lenguaje de programación C++ es un superset de C.

Existen estándares de compilación, los principales son ANSI C++ y ISO C++.

Tipos de lenguaje

Compilado	Interpretado
<ul style="list-style-type: none">• Debe ser compilado antes de ejecutarse.• El compilador genera un binario ejecutable a partir del código.• Se compila para una arquitectura y sistema operativo específico.	<ul style="list-style-type: none">• No requiere compilación• Un programa llamado “intérprete” se ocupa de traducir en tiempo real el código a medida que se quiere ejecutar.• Se necesita tener el intérprete instalado para poder ejecutar el código.

Algunos compiladores de utilidad son gcc (linux) y MinGW (Windows).

Tipos de datos

Es necesario definir los tipos de datos (**int**, **char**, **float**, etc.).

Enteros (numérico)	short	%i	± 32.767	2 bytes
	int		$\pm 2.147.483.647$	4 bytes
	long		$\pm 2.147.483.647$	4 bytes
	unsigned	Establece que sólo se puedan utilizar números positivos, duplicando el alcance máximo del rango		
Reales (numérico)	float	%f	6 decimales	4 bytes
	double	%d	15 decimales	8 bytes
	long double		19 decimales	10 bytes
Caracteres	char	%c	ASCII	1 byte
Lógicos	bool		true - false	

Se pueden castear los tipos de datos:

```
char c; // Declaración de la variable
c = 'A'; // Inicialización de la variable
          o definición de variable

cout << c; // ----> A
cout << (int)c; // ----> 65 (ASCII)
```



Primeros pasos en C++

```
#include <iostream> // g++  
  
int main() // donde inicia el código  
{  
    std::cout << "Hola Mundo!" << std::endl;  
    return 0;  
}
```

Donde **cout** es “canal de salida” que imprime algo (como un `printf`), y **endl** significa “final de línea”, lo cual agrega un salto de línea.

Si se agrega la línea **using namespace std;**
luego de los `#include`, permite omitir el uso de `std::`

→ standard

Si la función **main** devuelve un **0**, por convención significa que el programa se ejecutó correctamente.



Sintaxis básica

Los bloques de código se escriben entre llaves. { }

Las variables no pueden iniciar con números, pero si pueden contener guiones bajos y signo dólar (\$).

Los nombres siempre deben corresponder con lo que la variable va a guardar o lo que la función hará. Deben ser nombres completos, y si tiene más de una palabra, separar con guión bajo o utilizar camelCase.

En C++ se pueden utilizar constantes mediante el prefijo **const**. De esta manera, queda completamente en desuso el **#define**. Es una buena práctica definirlos luego de los **#include** y esté escrito en mayúsculas. **const int IVA = 21;**

En C++ se permite usar el incremento (++) y el decremento (--) de una variable por detrás o por delante de la variable. La diferencia es la siguiente:

```
int x = 5;  
int y;
```

```
// Postincremento: asigna y luego incrementa  
y = x++; // x = 6, y = 5  
  
// Preincremento: incrementa y asigna  
y = ++x; // x = 6, y = 6
```

La palabra reservada **sizeof** permite obtener el tamaño en bytes de una variable.

```
unsigned int edad = 27;  
cout << sizeof(edad); // 4
```

Tipos de datos derivados

Vectores

Al trabajar con un vector como parámetro de una función, no es necesario declarar la cantidad de elementos que tiene.

Sin embargo, no hay que olvidar que los vectores tienen un tope definido.

```
void cargar(int vec[], int n) {  
    for(int i = 0; i < n; i++) {  
        ...  
    }  
}
```

```
const int MAXIMO = 5;  
int main() {  
    int vector[MAXIMO];  
    cargar(vector, MAXIMO);  
    return 0;  
}
```





Matrices

Una matriz es un vector de vectores, es decir, un vector cuyos elementos también son vectores.

Al trabajar con una matriz como parámetro de una función, no es necesario declarar la cantidad de filas que tiene. Sin embargo, es obligatorio declarar el número de las columnas.

```
const int COLUMNAS = 5;           filas columns
void cargar(int mat[][COLUMNAS], int m, int n) {
    for(int i = 0; i < m; i++) {
        for(int j = 0; j < n; j++) {
            ...
        }
    }
}
```

Es requerido declarar las columnas debido a que, al declarar una variable de tipo vector (una matriz es un vector de vectores), es necesario definir la cantidad de espacios de memoria que necesita cada vector del vector de vectores.

Estructuras

Permite el agrupamiento de datos en una estructura. Se puede acceder a los datos mediante el uso de un punto.

Por convención, se utiliza la primera palabra de la estructura en mayúsculas.

```
// Estructura
struct Empleado {
    int legajo;
    float sueldo;
};
```



```
Empleado empleado_nuevo;
empleado_nuevo.legajo = 109619;
empleado_nuevo.sueldo = 105.21;
```

Enumerados

Similar a una estructura que, básicamente, permite la declaración de varias constantes.

```
// Enumerado
enum Estado {
    NO_COMENZADO,
    INICIADO,
    FINALIZADO
};
```

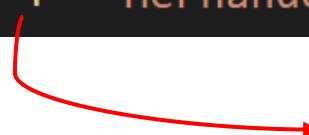


```
Estado partido;
partido = NO_COMENZADO;
...
if(partido == INICIADO)
    consultar_puntuacion();
```

Cadena de caracteres

Utilizando la librería de strings: `#include <string>`, se pueden utilizar cadenas de caracteres en C++ (y ya no como un vector de char).

```
string nombre;
nombre = "Guillermo";
nombre = nombre + " Hernandez";
```

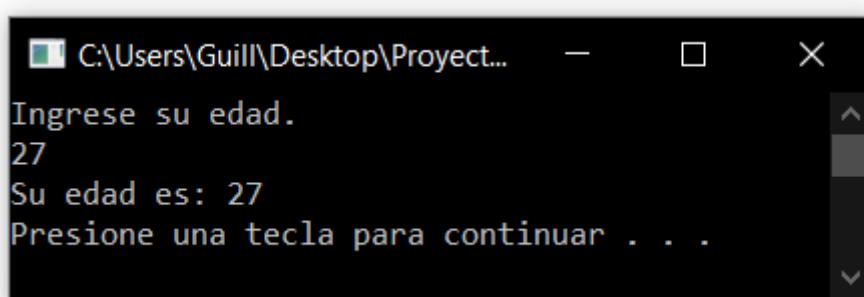


permite concatenar
cadenas de texto

Canal de entrada

Para pedir datos por pantalla al usuario, se utiliza **cin**.

```
#include <iostream> // g++  
  
using namespace std;  
  
int main() {  
    unsigned int edad;  
  
    cout << "Ingrese su edad." << endl;  
  
    cin >> edad; // Se pide dato al usuario  
  
    cout << "Su edad es: " << edad << endl;  
  
    return 0;  
}
```



A screenshot of a terminal window titled 'C:\Users\Guill\Desktop\Proyect...'. The window contains the following text:
Ingrrese su edad.
27
Su edad es: 27
Presione una tecla para continuar . . .



Optimización del canal de entrada

Se puede obtener una cadena de caracteres del canal de entrada (incluyendo los espacios en blanco) con la siguiente línea:

```
string texto_ingresado;
getline(cin >> ws, texto_ingresado);
```

Optimización de la obtención de un entero por canal de entrada

```
const unsigned int MAX_CARACTERES_ENTERO = 10;

/*
 * Pre-condiciones: El usuario debe ingresar un número entero por pantalla.
 * Pos-condiciones: Asigna el número entero ingresado por pantalla a numero_entero.
 */
void obtener_entero_ingresado(int & numero_entero) {
    unsigned int longitud_entero = MAX_CARACTERES_ENTERO;
    unsigned int caracter = 0;
    string valor_ingresado;
    char numero_ingresado[MAX_CARACTERES_ENTERO + 2];

    getline(cin >> ws, valor_ingresado);

    if(valor_ingresado[0] == '-') {
        numero_ingresado[0] = valor_ingresado[0];
        caracter++;
        longitud_entero++;
    }

    if(isdigit(valor_ingresado[caracter]) && valor_ingresado.length() <= longitud_entero) {
        while(isdigit(valor_ingresado[caracter]) && caracter < longitud_entero) {
            numero_ingresado[caracter] = valor_ingresado[caracter];
            caracter++;
        }
    } else {
        numero_ingresado[0] = '0';
        caracter = 1;
    }

    numero_ingresado[caracter] = '\0';
    numero_entero = stoi(numero_ingresado);
}
```



Funciones

Las funciones deben declararse y definirse por encima del `main`.

```
tipo_de_dato nombre_de_la_funcion ( datos de entrada ) {  
    . . .  
    return dato de salida;  
}
```

Se llaman funciones a aquellas que devuelven un dato de salida, y procedimiento a las que no devuelven nada (`void`).

Parámetros por referencia

Para poder editar directamente el valor de una variable a través de una función, se utiliza el pasaje por referencia definido con un &.

```
int x = 5;  
int y = 8;  
  
intercambiar(x,y);  
  
// x = 8;  
// y = 5;
```

```
void intercambiar(int & x, int & y) {  
    int aux = x;  
    x = y;  
    y = aux;  
}
```



Cabecera o firma de la función (*header*)

Se puede declarar una función por encima del `main`, y definirla por debajo.

Una forma óptima y de buena práctica, es establecer la declaración y la definición de las funciones en archivos por separado. Luego, para poder utilizarlos en nuestro código, se agrega un `#include` con el nombre del archivo de cabecera entre comillas dobles.

```
#include "suma.h"
```

```
// Declaración de función  
int suma(int op1, int op2);
```

```
int main() {  
    ...  
}
```

```
// Definición de la función  
int suma(int op1, int op2) {  
    return op1+op2;  
}
```

suma.h

suma.cpp

este archivo también
debe incluir el `#include`

De forma práctica, quedaría de la siguiente manera:

```
↳ suma.cpp > ...  
1   #include "suma.h"  
2  
3   int suma(int op1, int op2) {  
4       return op1+op2;  
5   }
```

```
↳ suma.h > ...  
1   #ifndef __suma_h__  
2   #define __suma_h__  
3  
4   int suma(int op1, int op2);  
5  
6   #endif /* __suma_h__ */
```

```
↳ main.cpp > ⚙ main()  
1   #include <iostream> // g++  
2   #include "suma.h"  
3  
4   using namespace std;  
5  
6   int main() {  
7       int x = 5, y = 3;  
8       int resultado = suma(x,y);  
9       // resultado == 8  
10 }
```



Proceso de compilación

Preprocesamiento

El precompilador busca el contenido de los `#include`, `#define`, `#ifndef`, `#endif`, y los carga (pega en nuestro código para utilizarlo). Es decir, reemplaza las palabras reservadas con código.

Esto posibilita la división del código fuente en módulos independientes.

¿Dónde se encuentra la librería iostream?

Compilación

Un compilador traduce directamente el código fuente en instrucciones que la computadora pueda entender.

En este proceso se crean los archivos objeto (.o)

Linkedición

Une los archivos objeto creados en el proceso de compilación en un solo archivo ejecutable que el usuario pueda utilizar.



Archivos

Abrir

Abro el archivo para lectura o para escritura según lo necesario.

```
Lectura: ifstream miArchivo("ejemplo.txt");
Escritura: ofstream miArchivo("");
fstream miArchivo("", tipo); // requiere include <fstream>
```

Chequeo de errores

Verificar que el archivo se abrió correctamente. Si no lo hizo notificar al usuario.

```
if (miArchivo.is_open() == true) ...
```

Procesamiento

Realizar las acciones necesarias.

Por defecto, salto de linea

```
Lectura por línea    getline(miArchivo, línea, delimitador);
                                         Devuelve la cantidad de caracteres que pudo leer
```

Lectura por palabra: miArchivo >> palabra; // lee hasta

Escritura por palabra: miArchivo << fraseAGuardar << endl;

Cerrar

Siempre cerrar los archivos al terminar de utilizarlos. miArchivo.close();

Memoria dinámica

División de la memoria (segmentos)

Heap (extra segment): Zona de la memoria dinámica. Los bloques de memoria que se soliciten deben liberarse antes finalizar el programa, en caso contrario, quedará memoria ocupada en la computadora incluso tras finalizar el programa.

Stack segment (pila): Almacena el contenido de las variables locales en la invocación de cada función, incluyendo las de la función main.

Data segment: Almacena el contenido de las variables definidas como externas (globales) y las estáticas.

Code segment: Donde se localiza el código resultante de compilar nuestra aplicación.



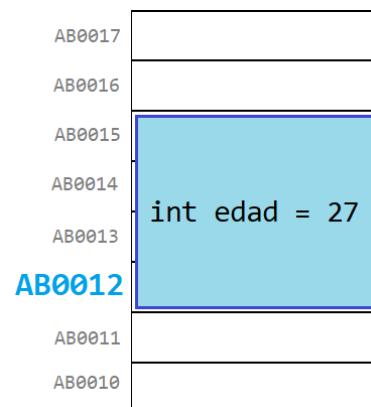
Se encarga el compilador de forma automática

Consideremos a un byte como una celda. Recordemos que un byte = 8 bits, por lo tanto, en cada byte se pueden almacenar hasta $2^8 = 256$ valores diferentes.

Cada byte (celda) se identifica con un número que llamamos **dirección**, este numero es correlativo. En general, se representa en formato hexadecimal por ser múltiplo de 8.

<code>short</code>	2 bytes
<code>int</code>	4 bytes
<code>long</code>	4 bytes
<code>float</code>	4 bytes
<code>double</code>	8 bytes
<code>long double</code>	10 bytes
<code>char</code>	1 byte
<code>bool</code>	1 byte

Cada variable ocupa uno o más bytes (celdas) dependiendo del tipo de variable y la plataforma de la máquina.



Por ejemplo, si una variable de tipo int ocupa 4 bytes, se reservan cuatro celdas **contiguas**, siendo la dirección de la primera celda la dirección correspondiente a esa variable. Si una variable no ocupa completamente el byte (como un bool), rellena el resto de bites con ceros.

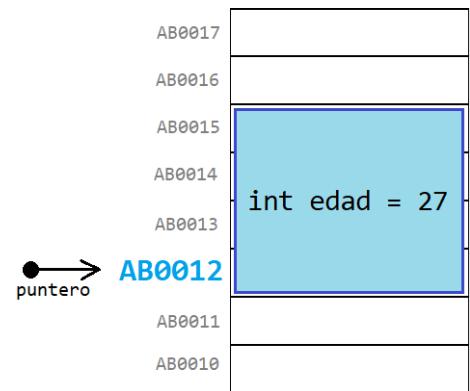
Si queremos saber en qué dirección está una variable, debemos usar el operador `&` (aspersen), pero para imprimir su valor hay que indicar que lo tome como un entero sin signo.

```
int var = 8;
cout << (unsigned) &var << endl;
```

Punteros

Tipo especial de variable que almacena direcciones de memoria. Es decir, en lugar de guardar datos, almacena direcciones de memoria de la propia máquina.

Cuando una variable de tipo puntero guarda una dirección de memoria, decimos que apunta a esa dirección.



Un puntero tiene que conocer el **tipo de dato** de la variable que esté en la dirección de memoria que almacenará su valor. Esto es para saber cuántas celdas de memoria deben reservarse y para saber cómo interpretar los datos almacenados (debido a que en cada celda sólo se guardan bits). Un puntero de enteros suele ocupar 8 bytes.



Se declaran indicando su tipo de dato junto con un asterisco y un identificador de la variable (nombre). No se puede declarar más de una variable de tipo puntero en una misma línea.

```
int* punteroEntero;
```

No se puede definir / asignar de forma explícita / literal un espacio de memoria, sin embargo, se puede asignar la dirección de memoria de otra variable.

```
int* puntero_edad;
int edad = 27;

puntero_edad = &edad;
```

Para obtener la dirección de memoria de una variable, se utiliza el aspersor (**&**).

Teniendo un puntero asignado a la dirección de memoria de una variable, se puede modificar dicha variable a través de su puntero. Para ello, se debe utilizar el asterisco, ya que funciona como operador de *desreferenciación*.

```
*puntero_edad = 52; // edad == 52
```



Punteros y vectores

Cuando declaramos un vector, el compilador reserva la memoria solicitada y guarda sólo la *dirección del primer elemento*.

```
cout << vector[0] << endl; // imprime el valor del primer elemento del vector  
cout << vector << endl; // imprime la dirección del primer elemento del vector
```

Por lo tanto, como en la variable `vector` se guarda una dirección, se puede pensar como un puntero constante (no se puede cambiar su dirección de apuntado).

Sabiendo esto, podemos crear un puntero con la dirección del vector y trabajarla directamente desde un puntero.

```
int* puntero;  
int vector[3];  
  
puntero = vector;  
  
vector[0] = 1; /* == */ puntero[0] = 1;  
  
vector[1] = 2; /* == */ puntero[1] = 2;  
  
vector[2] = 3; /* == */ puntero[2] = 3;
```

Definición de un tipo de dato de tipo puntero

Se puede establecer, al inicio de nuestro código, el tipo de dato puntero de enteros mediante el siguiente código:

```
typedef int* Pint;
```

De esta manera, se puede abreviar a escritura en la creación de variables de este tipo:

```
Pint punteroEntero; // declaración de variable
```



Uso de la memoria dinámica (heap)

Se utiliza la palabra reservada **new** y un tipo de dato para crear una variable anónima. De esta manera, se solicita al **heap** que se guarde una porción de memoria, la cual debe guardarse en un puntero.

```
int* punteroEntero = new int;
```

De esta forma, queda guardada la dirección de esa variable anónima (perteneciente al heap) dentro de nuestro puntero (perteneciente al stack) y puede accederse mediante el operador asterisco para guardar / recuperar valores.

Cuando trabajamos con memoria dinámica, debemos tener cuidado porque la memoria solicitada hay que liberarla, ya que permanecerá incluso finalizado nuestro programa. Se hace mediante el operador **delete**. Además, es buena práctica asignar el puntero a **nullptr** para que se borre la dirección del puntero.

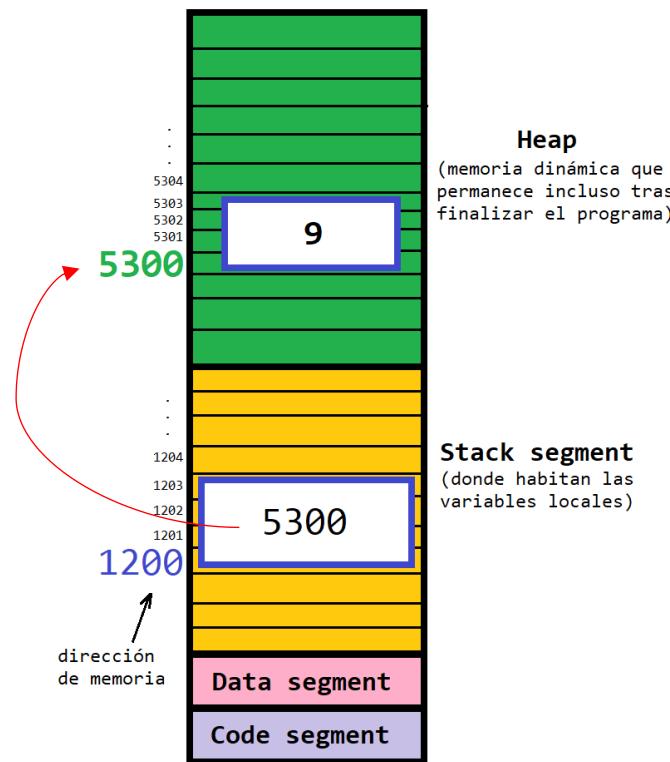
```
delete punteroEntero;  
punteroEntero = nullptr;
```

Para comprender cómo funciona la interacción entre la memoria heap y la memoria stack por medio de punteros, se puede representar de la siguiente manera:

```
int* p;
p = new int;
*p = 9;

delete p;

// p == 5300
// *p == 9
// &p == 1200
```



Se traduce como que la variable `p` (la cual es un puntero de tipo entero) está ubicada dentro del **stack segment**, en la dirección **1200** (la cual se puede obtener mediante el operando `&` en `p`).

Dicha variable `p` almacena una dirección de memoria que pertenece al **heap**, y dicha dirección puede alterarse mediante la **desreferenciación (*)** de `p`.



Vectores dinámicos

La potencia de los punteros se aprecia en las estructuras dinámicas. Si quisiéramos crear un vector de enteros de forma dinámica, podemos solicitar todo un bloque al **heap**.

```
int* puntero = new int [5]; // bloque contiguo de 5 enteros
```

Luego, para liberar dicho bloque de la memoria, se utililiza:

```
delete [] puntero;
```

Al trabajar con vectores en memoria dinámica y a dicho vector se le agota el espacio, se crea un nuevo vector dinámico con el doble de tamaño, pasar todos los elementos del vector viejo al nuevo, y luego se elimina el vector viejo. Así, continuar trabajando con el vector nuevo con mayor espacio.

```
int* puntero_auxiliar;
int* vector_dinamico = new int [2]; // bloque contiguo de 2 enteros

vector_dinamico[0] = valorRandom;
vector_dinamico[1] = valorRandom;

/* El vector dinámico se llenó. Para continuar agregando elementos,
   es necesario crear un nuevo vector dinámico con el doble de espacio
   y destruir el ya creado cuyo espacio fue insuficiente. */

puntero_auxiliar = vector_dinamico;

vector_dinamico = new int[4]; // nuevo bloque con el doble de tamaño
for(int i = 0; i < 2; i++)// pasar los elementos del viejo vector al nuevo
    vector_dinamico[i] = puntero_auxiliar[i];
delete puntero_auxiliar;// Destruir el viejo vector dinámico
puntero_auxiliar = nullptr;

vector_dinamico[2] = valorRandom;
vector_dinamico[3] = valorRandom;
```



Estructuras y punteros

```
struct Libro {  
    string titulo;  
    char genero;  
    int puntaje;  
};
```

```
Libro* libro = new Libro;
```

```
(*libro).titulo = "C";  
libro->titulo = "C++";
```

```
delete libro;  
libro = nullptr;
```

} Son lo mismo



Dobles punteros

Se utilizan para crear matrices dinámicas.

```
Int** p = new int[10];
```



Valgrid

Es un programa que permite verificar que toda la memoria del Heap solicitada es correctamente liberada durante la ejecución del programa.



Linea de comandos

Igual que en C, argumentos en el main



Programación Estructurada

En programación estructurada, tenemos por un lado funciones y por otro datos.
Los datos se pasan a las funciones como parámetros.

Modalidad top – down, desde los módulos principales hacia los más simples.

Difícil para hacer desarrollos complejos.

Difícil para ubicar y corregir errores.

Difícil de mantener (realizar cambios).

Programación Orientada a Objetos

Todos los elementos son objetos que interactúan entre sí a través de mensajes (métodos/funciones para comunicarse).

La programación se hace de abajo hacia arriba: se construyen en primer lugar los objetos más simples para luego ir construyendo los objetos que utilizan a esos objetos más simples.

Los objetos pueden representar elementos.

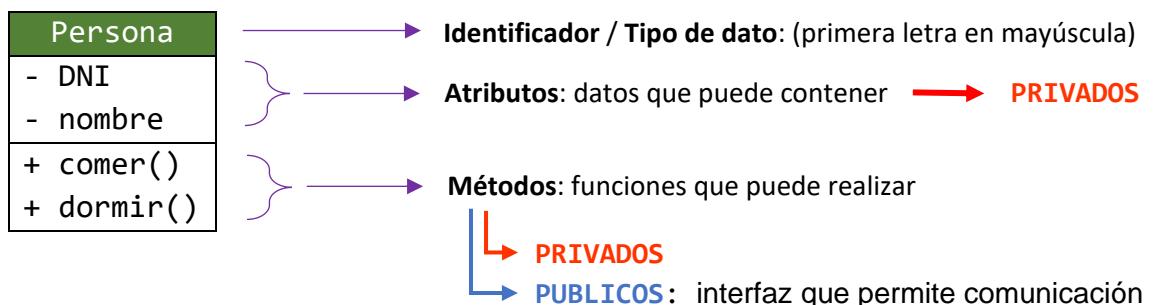
- Concretos: auto, estudiante, empleado, etc.
- Abstractos: un número complejo

Estructurado	Orientado a objetos
<pre>int vector[MAX_VECTOR]; int tope = 0; cargar(vector, tope); ordenar(vector, tope); mostrar(vector, tope);</pre>	<p>Se tiene un objeto de tipo vector, con métodos internos que serán inicializados</p> <pre>vector.cargar(); vector.ordenar(); vector.mostrar();</pre> <p>El tope del vector se encontrará dentro del mismo objeto Vector</p>

Etapas

Diseño	Implementación	Utilización
TDA (tipo de dato abstracto)	Clase	Objeto
Diseñador	Implementador	Usuario

→ Diseño UML (unified modeling language) estandarizado de un objeto



Clase

Es un tipo de dato que se define con sus propios atributos (privados) y métodos (privados por defecto).

Cada clase es un espacio de nombre ([spacename](#)).

```
class Persona {
    private: string nombre; // atributo
    public: void comer(); // método público (interfaz)
} private: void orinar(); // método privado
```

Objeto

Representación de una clase en una variable.

Características principales de POO

- **Encapsulamiento** Presentación del código de manera que el usuario pueda utilizarlo con sólo la interfaz, sin necesidad de conocer la implementación del código. Es decir, sólo mediante el uso de los métodos públicos.
- **Ocultamiento de la implementación** Mediante el uso de métodos y atributos privados.
- **Herencia**
- **Polimorfismo**

Diferencia principal con un **struct**



Formato de una clase básica en POO

1. Se realiza el diseño inicial de la clase.
2. Se declara la clase en un archivo HEADER y se define la clase en un archivo CPP.
3. Se utilizan objetos de la clase.

TDA	
EMPLEADO	
- nombre	
- salario	
+ asignar_nombre()	
+ asignar_salario()	
+ sueldo()	
- sueldo_neto()	

C Empleado.h > ...

```
1  #ifndef __EMPLEADO_H__
2  #define __EMPLEADO_H__
3
4  class Empleado {
5
6      // Atributos
7      private:
8          char* nombre;
9          int salario;
10
11     // Métodos públicos (interfaz)
12     public:
13         void asignar_nombre(char* nombre);
14         void asignar_salario(int salario);
15         int sueldo();
16
17     // Métodos privados
18     private:
19         int sueldo_neto();
20 };
21
22 #endif /* __EMPLEADO_H__ */
```

C Empleado.cpp > ...

```
1  #include "Empleado.h"
2
3  const int porcentaje_impuestos = 24;
4
5  void Empleado::asignar_nombre(char* nombre_empleado) {
6      nombre = nombre_empleado;
7  }
8
9  void Empleado::asignar_salario(int salario_empleado) {
10    salario = salario_empleado;
11 }
12
13 int Empleado::sueldo_neto() {
14     return (salario * 100 / porcentaje_impuestos);
15 }
16
17 int Empleado::sueldo() {
18     return sueldo_neto();
19 }
```

```
int main() {
    Empleado empleado;
    empleado.asignar_nombre(NOMBRE);
    empleado.asignar_salario(95000);

    cout << "El sueldo del empleado es: " << empleado.sueldo() << endl;

    return 0;
}
```



Interfaz

Los métodos públicos de una clase corresponden a la “interfaz”, es decir, los métodos que el usuario podrá usar.

Dichos métodos deben contener 3 comentarios:

- ❖ **Descripcion** Breve explicación de lo que hace el método (palabras sencillas).
- ❖ **Pre-condiciones** Estado del objeto y restricciones de los parámetros.
- ❖ **Post-condiciones** Lo que sucederá, y como estará el objeto luego.

Métodos getter y setter

Para asignar o acceder a un valor de los atributos de una clase, es necesario crear un método que permita realizar dicha acción debido a que los atributos no pueden ser accedidos directamente (por ser privados).

Para ello, se crean los métodos **obtener (get)** y **asignar (set)**.

Constructor

Debe tener el mismo nombre que la clase, no tiene tipo de dato (ni siquiera void).

Se llama automáticamente cuando el objeto se crea y se utiliza para inicializar los atributos.

Se dice que el constructor está sobrecargado cuando existe más de uno, pero ninguno debe tener los mismos parámetros que otro constructor existente.

```
Ejemplo > C punto_r2.h > ...
1  #ifndef __PUNTO_R2__
2  #define __PUNTO_R2__
3
4  class Punto_en_R2 {
5
6      // Constructor sin parámetros
7  public:
8      Punto_en_R2();
9
10     // Constructor con parámetros
11 public:
12     Punto_en_R2(double x, double y);
13
14     // Destructor
15 public:
16     ~Punto_en_R2();
17
18     // Atributos
19 private:
20     double x;
21     double y;
22 };
23
24 #endif /* __PUNTO_R2__ */
```

```
Ejemplo > C punto_r2.cpp > ...
1  #include "punto_r2.h"
2
3  // Constructor sin parámetros
4  Punto_en_R2::Punto_en_R2() {
5      x = 0;
6      y = 0;
7 }
8
9  // Constructor con parámetros
10 Punto_en_R2::Punto_en_R2(double x, double y) {
11     this->x = x;
12     this->y = y;
13 }
14
15 // Destructor
16 Punto_en_R2::~Punto_en_R2() {
17     x = 0;
18     y = 0;
19 }
```

```
int main() {
    Punto_en_R2 punto_en_origen();
    Punto_en_R2 punto_especifico(0,0);

    return 0;// Destructor se ejecuta
}
```

Destructor

Método que destruye el objeto y se llama automáticamente cuando finaliza el método main (o delete si es que se creó de forma dinámica).

Constructor de copia

Se ejecuta automáticamente cuando se crea un objeto igualado a otro.

```
int main() {
    Punto_en_R2 punto_en_origen();
    Punto_en_R2 otro_punto = punto_en_origen;

    return 0; // Destructor se ejecuta 2 veces
}

un equivalente es:
Punto_en_R2 otro_punto(punto_en_origen);
```

Para poder tener control sobre este tipo de copias, se puede sobreescribir el método del **constructor de copia** en la clase.

```
// Constructor de copia
// Pre-condiciones: -
// Pos-condiciones: Crea una copia del objeto
Punto_en_R2 (const Punto_en_R2 & objeto);
```

Se utiliza **const** para asegurar que el objeto original (el cual se intenta copiar) no pueda ser modificado.

Se pasa por **referencia** debido a que, si no fuera de esta forma, se estaría pasando una copia del objeto. Y como estamos trabajando en el constructor de copia, no puede utilizarse.

Inicializador

En el constructor, y antes de iniciar las llaves, VIDEO DE 15/9 AL FINAL

Se utiliza para construir un objeto que tiene otro dentro.

Vector dinámico

Con el uso de programación orientada a objetos y memoria dinámica, se puede crear un vector dinámico con el uso de clases.

Es importante tener en cuenta la sobreescritura (*sobrecarga*) del **Constructor de copia** para evitar que, al igualar un objeto Vector con otro, no se utilice el mismo espacio de memoria en el Heap para ambos.

C **vector.h >**

```
#ifndef __VECTOR_H__
#define __VECTOR_H__

typedef int Type; // define el tipo de dato del vector

class Vector {

private: // Atributos
    int longitud;
    Type* elementos; // puntero que señala el primer elemento de un vector

public: // Métodos
    // Constructor sin parámetros
    // Pos-condiciones: crea un objeto Vector vacío con longitud 0
    Vector();

    // Constructor con parámetros
    // Pre-condiciones: cantidad debe ser mayor que 0
    // Pos-condiciones: crea un objeto Vector de longitud = cantidad
    Vector(int longitud);

    // Constructor de copia
    // Pos-condiciones: Crea una copia del objeto Vector con direcciones de memoria diferentes
    Vector(const Vector & vector);

    // Destructor
    // Pos-condiciones: libera la memoria solicitada
    ~Vector();

    // Pre-condiciones: 0 <= posicion < longitud
    // Pos-condiciones: asigna elemento en posicion el Vector
    void asignar(Type elemento, int posicion);

    // Pre-condiciones: 0 <= posicion < longitud
    // Pos-condiciones: devuelve el elemento de posicion
    Type obtener(int posicion);

    // Pre-condiciones: -
    // Pos-condiciones: devuelve el atributo longitud de objeto Vector
    int obtener_longitud();

};

#endif /* __VECTOR_H__ */
```

C++ vector.cpp >

```
#include "vector.h"

// Constructor sin parámetros
Vector::Vector() {
    longitud = 0;
    elementos = 0; // nullptr
}

// Constructor con parámetros
Vector::Vector(int longitud) {
    this->longitud = longitud;
    elementos = new Type[longitud];
}

// Constructor de copia
Vector::Vector(Vector vector) {
    longitud = vector.longitud;
    if(longitud > 0) {
        elementos = new Type[longitud];
        for(int i = 0; i < longitud; i++)
            elementos[i] = vector.elementos[i];
    } else elementos = 0;
}

// Destructor
Vector::~Vector() {
    if(longitud > 0)
        delete [] elementos;
}

void Vector::asignar(Type elemento, int posicion) {
    elementos[posicion] = elemento;
}

Type Vector::obtener(int posicion) {
    return elementos[posicion];
}

int Vector::obtener_longitud() {
    return longitud;
}
```

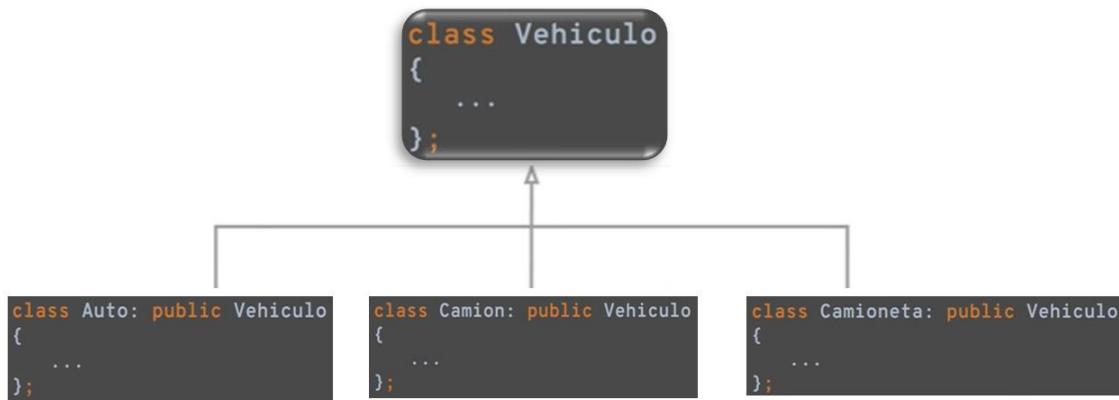
Herencia

Es una relación entre clases por la cual se define una clase que puede ser un caso particular de otra.

Tenemos una clase más general (**clase madre**) que le pasa sus características (métodos y atributos) a las clases más particulares (**clase hija**).

Una forma sencilla de detectar una herencia es si se puede expresar con la frase “**es un**”. Por ejemplo: un **Auto** **es un** **Vehículo**.

También se utiliza la expresión “Auto hereda (es subclase de) la clase Vehículo”.



La herencia sólo debe usarse en los casos en que el objeto de la clase hija utilizará todos los atributos y métodos de la clase madre.

Es decir, se debe utilizar la herencia si la clase padre tiene TODOS los atributos y métodos que la clase hija necesita y que, además, la clase hija necesite los suyos propios. No deben sobrar métodos ni atributos de la clase padre (no pueden quedar atributos de la clase padre sin utilizar).

Además, no es recomendable crear una clase hija si sólo se utilizará una vez. Es decir, si sólo existen vehículos de tipo Auto en el programa, no es necesario crear una clase madre “Vehículos” y una clase hija “Auto”. Simplemente crear una clase Auto que englobe todo lo que se necesita será suficiente.

Es importante saber que existen desventajas en el uso de las herencias. Entre ellas están que es estático, poco flexible en comparación con la composición y que puede romper el encapsulamiento.

Constructores y destructores

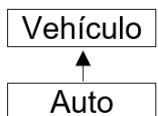
- La construcción se realiza, en primer lugar, con la parte de los objetos “ancestros” (de la clase madre) y luego la parte correspondiente a las subclases.
- La destrucción se realiza en orden inverso; en primer lugar, se destruye lo correspondiente a los herederos (clases hijas) y luego se va “subiendo” y destruyendo la parte correspondiente a las clases padre (ancestros).

Inicializadores

Si **Auto** hereda de **Vehículo** (Auto es la clase hija y Vehículo es la clase madre), y **Vehículo** no tiene el constructor por defecto, se debe llamar al constructor necesario antes de realizar cualquier otra tarea.

```
Auto(string patente, bool esManual) : Vehiculo(patente)
{
    ...
};
```

Inicializador



Protected (#)

Los atributos y métodos definidos con la palabra reservada **protected** serán capaces de heredarse a clases hijas.

No pueden ser accedidos por otras clases que no sean las clases hijas o la propia clase en donde se está definiendo.

```
class Vehiculo
{
protected:
    int ruedas;
    ...
};
```

Punteros

Punteros de tipo MASCOTA pueden apuntar a la clase madre MASCOTA y a las clases hijas GATO y PERRO, pero sólo puede usar los métodos y atributos de la clase MASCOTA.

Punteros de tipo GATO sólo puede apuntar a la clase GATO, no a MASCOTA.

- | → M: ref_mascota
- | → P: ref_perro
- | → G: ref_gato

Escrito en pseudocódigo.

M := P // válido

M es un puntero a una clase de tipo Mascota.

M := G // válido

P es un puntero a una clase de tipo Perro.

M := objeto_perro

G es un puntero a una clase de tipo Gato.

P := M // inválido

M.ladrar () // inválido

Polimorfismo

Permite la posibilidad de que distintos objetos respondan de manera diferente ante la llegada del mismo mensaje.

Clase concreta

Una clase se considera concreta cuando se pueden crear objetos a partir de ella, es decir **puede instanciarse**.

Clase abstracta

Una clase se considera abstracta cuando **no puede tener instancias** en forma directa. Para ello, es suficiente con que uno de sus métodos sea abstracto mediante la palabra reservada **virtual**.

Los atributos y métodos abstractos pueden o no ser definidos en la clase madre, pero siempre deben ser declarados.

```
class Figura {
protected:
    *int color[3];
public:
    virtual void dibujar() = 0;
    void renderizar();
};

class Cuadrado: public Figura {
private:
    int base;
public:
    void dibujar();
};
```

virtual void dibujar() = 0;

Método declarado como abstracto y no definido. La igualación a 0 hace que, en la clase heredada, se tenga que definir la función **dibujar** obligatoriamente y, por lo tanto, en la clase madre no deba definirse.

En la clase hija “Cuadrado” se define (se especifica la funcionalidad) del método **dibujar**.

Si en la clase hija sigue sin definirse algún método abstracto declarado en la clase madre con igualación a cero, la clase hija también se considerará clase abstracta.

Si no se agrega la igualación a cero en la clase madre, obligatoriamente debe definirse el método en dicha clase para, en caso que la clase hija no lo defina, utilice el método definido en la clase madre.

Los métodos predefinidos (como los constructores y destructores) no se igualan a cero nunca.

virtual ~Figura();

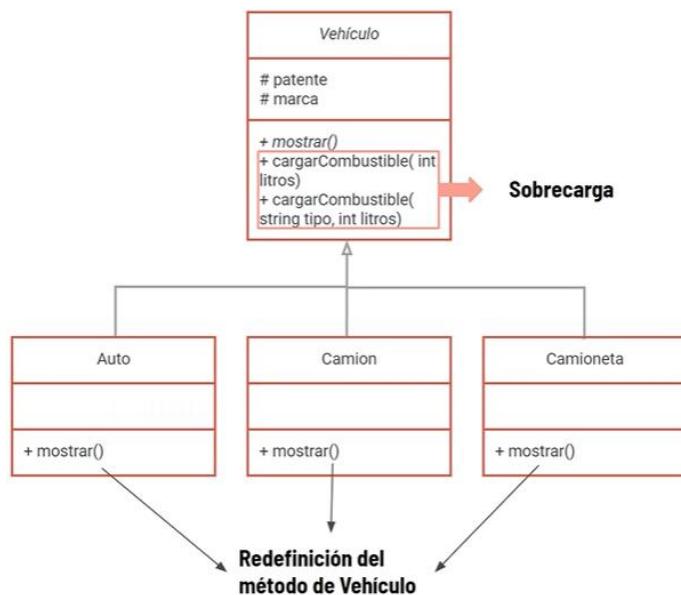
Herencia

Sobrecarga y Redefinición

La sobrecarga es cuando se crean métodos con el mismo nombre, pero con distinta cantidad y/o tipo de parámetros.

En la herencia y polimorfismo, se pueden redefinir los métodos (con los mismos parámetros). Es decir, se crea un método abstracto en la clase padre y en las clases hijas se define cómo van a funcionar. Cada clase hija puede tener una funcionalidad diferente del mismo método.

Es importante notar que, en los TDA y UML, las clases, atributos y métodos abstractos se escriben en cursiva (italica).



Usos del polimorfismo

- Estructuras de datos polimórficas
- Pasaje de parámetros en funciones.

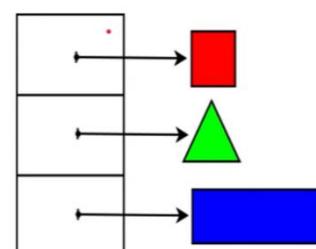


Diagrama de clases con UML

UML es un tipo de diseño de un sistema, estandarizado de forma global. Es un lenguaje unificado y modelado.

Lenguaje gráfico para visualizar, especificar, construir y documentar artefactos del software.

Se puede utilizar **DRAW.IO** para realizarlos.

Clasificación

Diagramas estáticos

- Casos de uso / User history lo que el sistema debe hacer y para qué
- Clases
- Objetos
- Componentes
- Despliegue

Diagramas dinámicos

- Estados
- Actividad
- Secuencia
- Colaboración

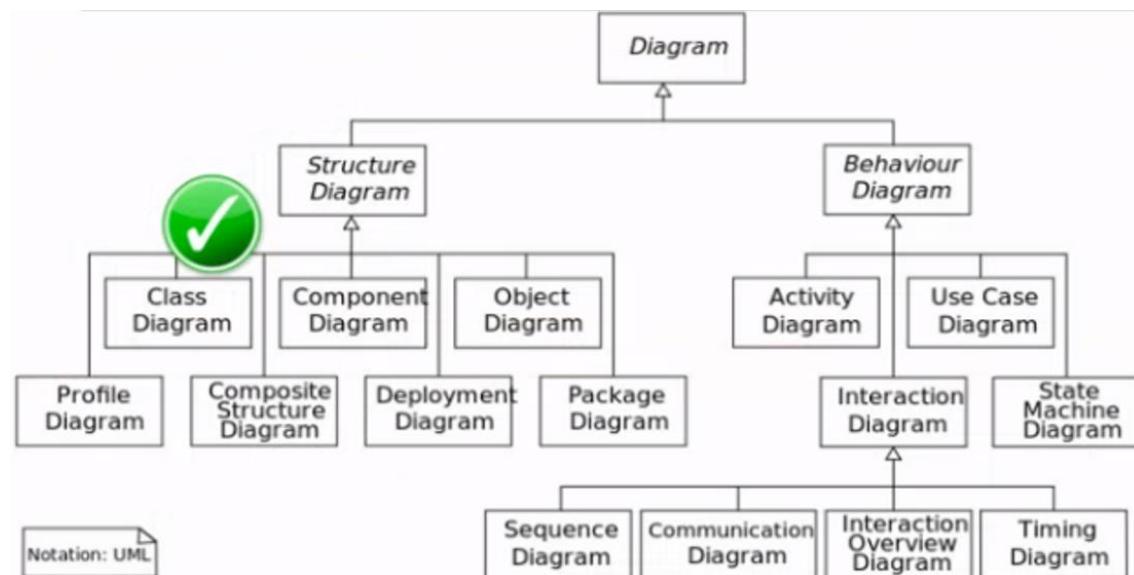




Diagrama de clase (tarea de un analista funcional)

Describe las clases (conjunto de objetos que comparten los mismos atributos y métodos) que conforman el modelo de un determinado sistema.

Es creado y refinado durante las fases de análisis y diseño, estando presente como guía en la implementación (programación) del sistema.

Perspectivas desde las cuales se pueden utilizar los diagramas de clase:

- **Conceptual** Conceptos en el dominio del problema que se está estudiando, se arma de forma preliminar en las primeras entrevistas que se realizan con el usuario. Se crea con la mayor independencia posible de la implementación final del sistema (sin tener en cuenta el software que lo implementa, es decir, es independiente del lenguaje de programación). Se dibujan rectángulos con los nombres de los objetos (draft / borrador) y se unen mediante flechas para representar las relaciones que tengan entre ellos.
- **Especificación** Refleja las interfaces de las clases, pero no su implementación. Se muestran las clases más cercanas a los tipos de datos.
- **Implementación** Representa las clases tal cual aparecen en el entorno de implementación. Es el que finalmente pasa a manos del programador.

A diferencia de un TDA (el cual es más abstracto / no se mete en los detalles), el diagrama de clases trabajado con UML tiene en cuenta el lenguaje de programación con el cual se está trabajando.

TDA	UML
Tipo de Dato Abstracto	Lenguaje de Modelado Unificado
INDEPENDIENTE del código y del lenguaje	DEPENDIENTE del código y del lenguaje
Abstracción del código	Documentación del código

Scoope o Alcance de clase

Al crear una clase para un determinado sistema, es necesario tener en cuenta la funcionalidad de dicho sistema para el armado de los atributos y métodos de la clase. Es decir, deben adaptarse a la necesidad del usuario / funcionalidad del sistema.

Por ejemplo: si un sistema es sólo para contar las ruedas de un vehículo, no es necesario agregar un método que lea la patente.

Representación de una clase en UML

Se escribe en forma de cuadro como un TDA.

En los atributos se debe especificar su visibilidad (encapsulamiento) y el rango de valores asociados, así como su tipo de dato (ya sea primitivo u otra clase).

Recordando que el encapsulamiento permite hacer que no se puedan alterar atributos o métodos desde otras clases.

Vehículo
+color: String
#patente: String
-velocidad_maxima: Integer
~peso: Float
+acelerar(aFondo: Boolean): void
+frenar(): void
<u>+patente valida(): Boolean</u>

⊕ Tipos de visibilidad

- **(privado)** No visible fuera de la clase ni en clases derivadas (heredadas).
- + **(público)** Visible desde cualquier clase (puede romperse el encapsulamiento).
- # **(protegido)** No visible fuera de la clase, pero visible desde clases derivadas.
- ~ **(paquete)** Sólo accesible desde clases en el mismo paquete.

Las operaciones / métodos de la clase son los servicios que expone la clase (el “contrato” que cumple la clase).

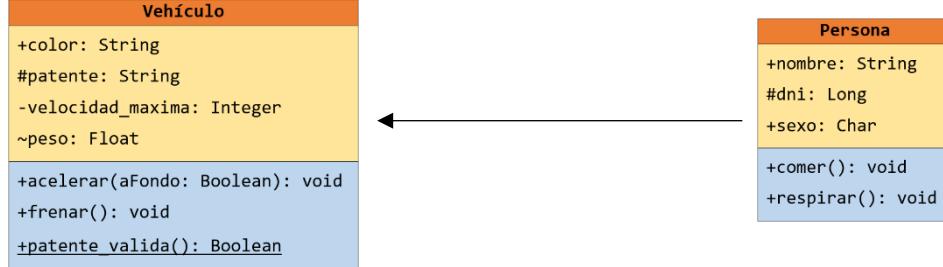
⊕ Tipos de métodos

- **Métodos de instancia** Requieren que se crea un objeto de esa clase para poder utilizarlo
- **Métodos de clase** **Static** No requieren que el objeto sea creado. Se representan subrayados.

Recordar que las clases abstractas (así como sus métodos abstractos) se escriben en cursiva.

Relaciones de clases en UML

Es la forma en que las clases se conectan (asocian) unas con otras, con el fin de resolver determinada funcionalidad. Se representan mediante distintos tipos de flecha.



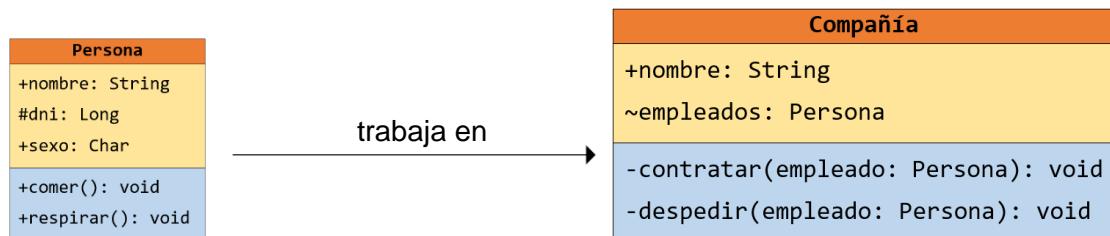
Existen muchas relaciones entre clases, y dichas relaciones pueden nombrarse como: tiene, crea, usa, es un...

Un ejemplo es: **silla es un mueble** (por herencia de clases).

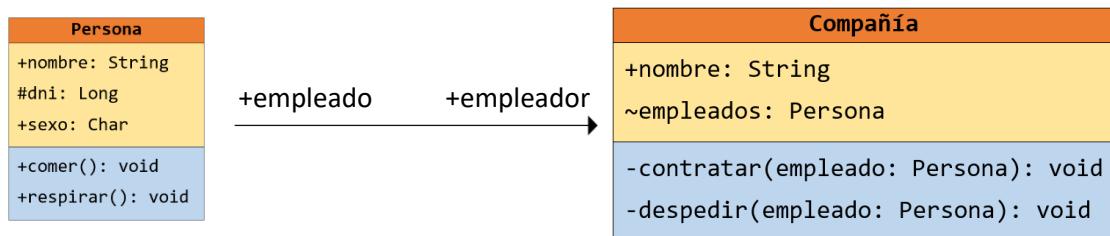
Se puede saber que una clase usa herencia cuando utiliza objetos de otra, crea otra clase o utiliza otra clase.

Para detallar el tipo de relación que hay entre dos clases, se puede utilizar:

- Nombre de la relación



- Rol



En este caso, dentro de la clase Compañía probablemente exista una lista / vector de objetos Persona en un atributo llamado “empleados”.

Se suele aclarar el “rol” o el “nombre de la relación”, no ambos al mismo tiempo.

Multiplicidad de relaciones en UML

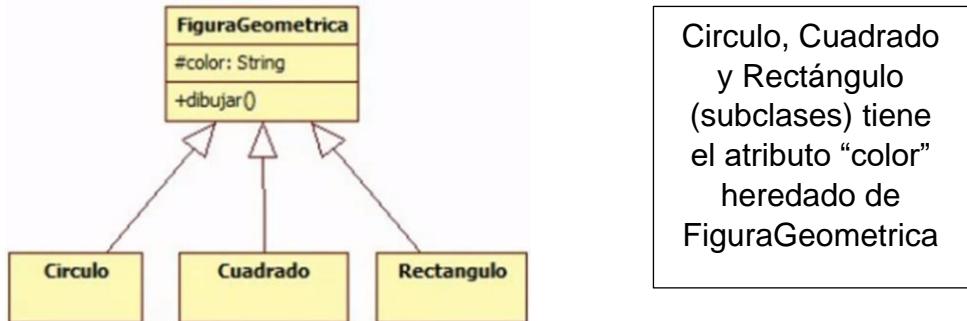
La multiplicidad relaciona una cierta cantidad de objetos de una clase con otra.

- 1 El atributo debe tener un único valor.
- 0..1 El atributo puede o no tener un valor.
- 0..* El atributo puede tener varios valores o ninguno.
- 1..* El atributo puede tener varios valores, pero debe tener al menos uno.
- * El atributo puede tener varios valores.
- M..N El atributo puede tener entre M y N valores.



Tipos de relaciones

- General



- Complejas

- Composición Una clase posee como atributo un objeto y comparten el tiempo de vida.



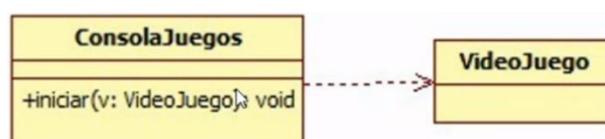
En este caso: Si destruyo el objeto Libro, se destruye el objeto Capítulo ya que el objeto Capítulo sólo se utiliza con Libro.

- Agregación Una clase posee como atributo un objeto, pero no comparten el tiempo de vida.



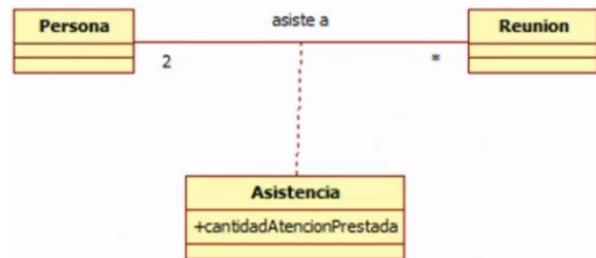
En este caso: Se puede destruir Agenda, pero el Contacto (su información) puede ser utilizado por otra clase.

- Dependencia Una clase depende de otra para poder ejecutar cierta funcionalidad. Es decir, dentro de un objeto se utiliza como parámetro otra clase.



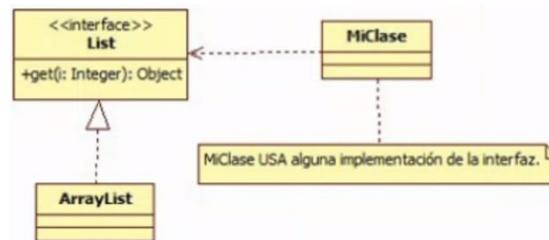
- Association class

Permite agregar atributos y operaciones a una asociación.

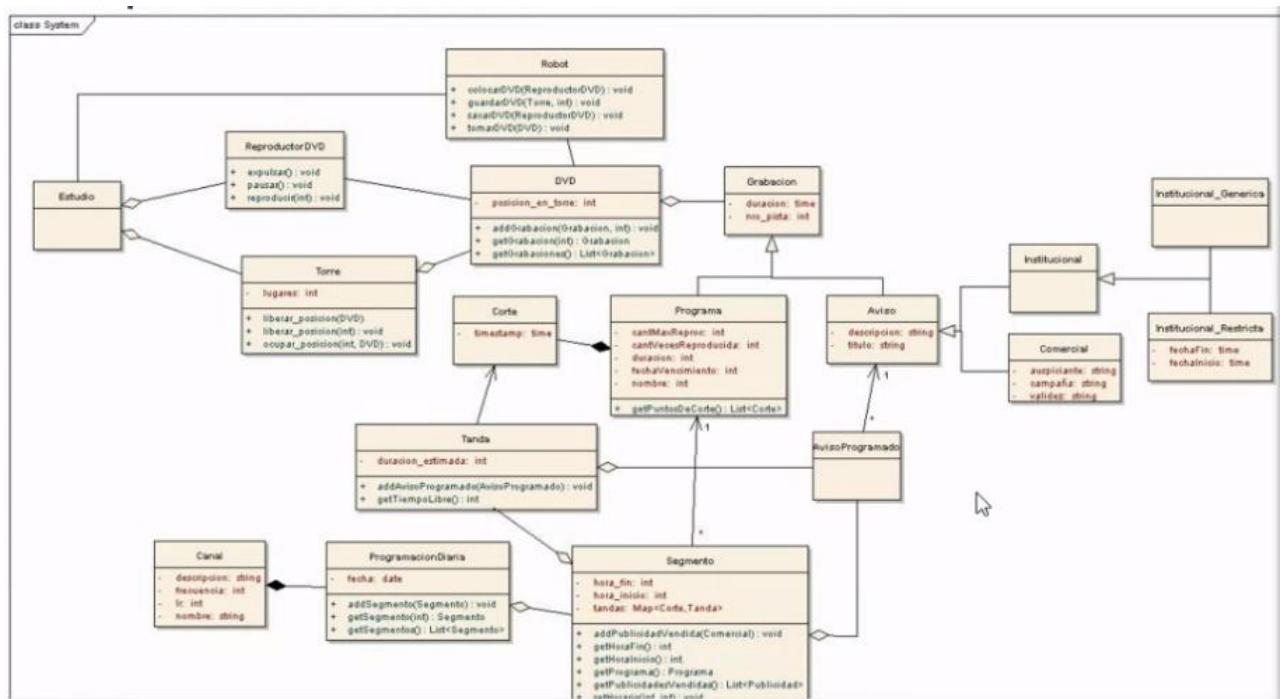


En este caso: una instancia de la association class corresponde exactamente al par Persona – Reunion.

- Interfaces: clase abstracta con todos sus métodos abstractos.



Ejemplo de un UML de clases



Ejemplo de un TDA

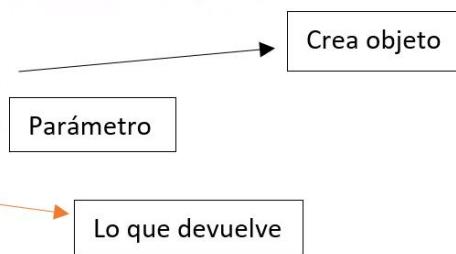
En lenguaje natural, donde no se especifican los tipos de datos, etc.

Una operación que siempre debe aclararse es la creación del objeto.

En detalle de operaciones se debe detallar la descripción, precondición y poscondicion.

TDA Persona

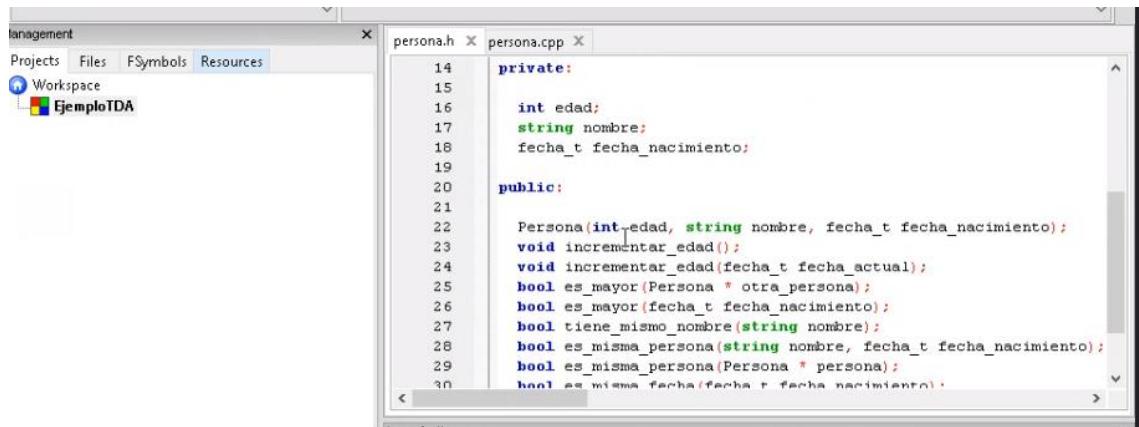
- Nombre: Persona
- Invariantes:
 - Edad ∈ Entero
 - Nombre, conjunto de caracteres alfabéticos
 - Fecha de nacimiento, fecha conformada por día, mes y año (dd/mm/aaaa)
- Operaciones:
 - Persona: Entero X cadena X Fecha → Persona
 - Incrementar edad
 - Incrementar edad: Fecha → booleano
 - Es mayor: Persona → booleano
 - Es mayor: Fecha → booleano
 - Tiene mismo nombre: cadena → booleano
 - Es la misma persona: Persona → booleano
 - Es la misma persona: cadena X Fecha → booleano
 - Tiene mismo nacimiento: Fecha → booleano



Detalle de las operaciones:

- Operación: Persona
 - Descripción: construye una Persona en base a 3 parámetros. El primero es la edad, el segundo es el nombre y el tercero es la fecha de nacimiento.
 - Precondición: la edad debe ser un número entero mayor a 0, el nombre debe estar compuesto por caracteres alfabéticos y la fecha debe tener un formato correcto (dd/mm/aaaa).
 - Postcondición: construye una Persona con la edad, el nombre y la fecha apropiados.
- Operación: Incrementar edad
 - Descripción: aumenta la edad de la Persona en 1.
 - Precondición: -
 - Postcondición: Incrementa la edad de la Persona en 1.
- Operación: Incrementar edad
 - Descripción aumenta la edad de la Persona en 1 si el día y el mes de la fecha indicada por parámetro son los mismos que los de la fecha de nacimiento.
 - Precondición: -
 - Postcondición: Si el día y el mes de la fecha indicada por parámetro son los mismos que los de la fecha de nacimiento, incrementa la edad de la persona en 1.

Con el TDA terminado, se puede proceder a la implementación de la siguiente manera:



```
14     private:  
15  
16         int edad;  
17         string nombre;  
18         fecha_t fecha_nacimiento;  
19  
20     public:  
21  
22         Persona(int edad, string nombre, fecha_t fecha_nacimiento);  
23         void incrementar_edad();  
24         void incrementar_edad(fecha_t fecha_actual);  
25         bool es_mayor(Persona * otra_persona);  
26         bool es_mayor(fecha_t fecha_nacimiento);  
27         bool tiene_mismo_nombre(string nombre);  
28         bool es_misma_persona(string nombre, fecha_t fecha_nacimiento);  
29         bool es_misma_persona(Persona * persona);  
30         bool es_misma_fecha(fecha_t fecha_nacimiento);
```

Templates

Son plantillas, definiciones genéricas de una estructura.

Permite hablar, por ejemplo, de “vectores de enteros” y “vectores de strings” sin necesidad de reescribir el código fuente del objeto.

Cuando usamos templates como datos, decimos que hacemos polimorfismo estático, ya que se define en tiempo de compilación.

Existen tanto clases template como funciones template.

Función Template

```
template < typename Tipo >
Tipo mayor(Tipo n1, Tipo n2){
    if(n1 > n2)
        return n1;
    return n2;
}
```

Notas:

typename también puede ser “class”

Se pueden usar más de un “Tipo” de dato, separándolo con comas

No necesariamente todos los tipos de dato de los argumentos deben ser “Tipo”.

Ejemplo:

```
int main(){
    int numeroInt1 = 7, numeroInt2 = 12;
    string palabra1 = "Carolina", palabra2 = "Andres";
    int numeroInt3 = 7, numeroInt4 = -1;

    cout << mayor<int>(numeroInt1, numeroInt2) << endl;
    cout << mayor(palabra1, palabra2) << endl;
    cout << mayor<unsigned>(numeroInt3, numeroInt4) << endl;

    return 0;
}
```

El primer cout imprime 12

En el segundo cout, no se pasaron los tipos de datos, entonces el compilador intenta deducirlo. Si puede deducirlo, entonces lo realiza sin problemas.
Entonces, en este caso, imprime Andres (compara los ascii, strcmp)

En el tercer cout, se fuerza que sea unsigned (a pesar que el segundo parámetro es un -1), de esta forma “transforma” el -1 en el numero MAX_INTEGER – 1 y por lo tanto lo tomará como el mayor, y lo devuelve (2 a la 32 – 1).

Observación: una misma función pudo utilizarse para 3 tipos de datos diferentes gracias a template.

Clase template

Una clase que utiliza templates debe escribirse todo dentro de un mismo archivo (en el header).

El template existe solamente cuando se instancia, por lo tanto, no pueden separarse su definición y su declaración.

Esto es porque no generan código hasta que se instancian y, por lo tanto, si se separaran, sería necesario definir todos los métodos con todos los tipos de datos posibles y sus combinaciones.

Ejemplo:

```
int main(){
    int numeroInt1 = 7, numeroInt2 = 12;
    double numeroDouble1 = 0.69, numeroDouble2 = 1.79;

    Mayor<int> mayorEnteros;
    Mayor<double> mayorDouble;

    cout << mayorEnteros.mayor(numeroInt1, numeroInt2) << endl;
    cout << mayorDouble.mayor(numeroDouble1, numeroDouble2) << endl;

    return 0;
}
```

Al instanciarse, se designa el tipo de dato (al crear el objeto).

```
#ifndef _MAYOR_TEMPLATE_
#define _MAYOR_TEMPLATE_

template < typename Tipo >
class Mayor{

public:
    Tipo mayor(Tipo n1, Tipo n2);

};

template < typename Tipo >
Tipo Mayor<Tipo>::mayor(Tipo n1, Tipo n2){
    if(n1 > n2)
        return n1;
    return n2;
}

#endif
```

Standard Template Library (STL)

Librería que contiene templates de diferentes estructuras, como listas o vectores. También tiene funciones templates, como lo son los algoritmos de ordenamiento.

Todas las estructuras definidas en STL tienen un iterador asociado que sabe cómo recorrer la estructura. Dicho iterador es cargado con punteros a los objetos de la estructura.

Al recorrer con un iterador, este puede ser aumentado como iterador normal (`++`) y automáticamente apunta al siguiente elemento de la estructura.

Nota: La memoria dinámica guardada en una estructura STL debe ser liberada por el usuario.

Ejemplo de STL lista

```
#include <iostream>
#include <list> → Librería que incluye
using namespace std;

int main(){
    list <int> lista;
    list <int> :: iterator iterador;
    lista.push_back(5); // Agregar 5 al final
    lista.push_back(9); // Agregar 9 al final
    lista.push_front(3); // Agregar 3 al principio
    for(iterador = lista.begin(); iterador != lista.end(); iterador++){
        cout << *iterador << endl;
    }
    // Imprime 3, 5, 9
    return 0;
}
```

Devuelve un iterador apuntando al principio de la lista

Devuelve un iterador apuntando al teórico elemento que le sigue al último elemento de la lista

```
advance(iterador, 1); → Mueve al
                        iterador una
                        posición
                        hacia
                        adelante
lista.insert(iterador, 36);
for(int elemento : lista){
    cout << elemento << endl;
}
```

Auto

```
list <int> :: iterator iterador = lista.begin();
```

equivalencia

```
auto iterator1 = lista.begin();
```

Por ejemplo:

```
int main(){

    list <int> lista;

    for(int i = 0; i < 10; i++){
        lista.push_back(i);
    }

    auto iterator1 = lista.begin();
    auto iterator2 = lista.begin();

    advance(iterator1, 6);
    iterator2++;

    iterator1 = lista.erase(iterator1);
    iterator2 = lista.erase(iterator2);

    cout << *(--iterator1) << endl;
    cout << *(iterator2--) << endl;

    lista.remove(9);

    for(auto entero : lista){
        cout << entero << endl;
    }

    return 0;
}
```

Ejemplo de un programa usando templates

```
C animal.h > ...
1  #ifndef __ANIMAL_H__
2  #define __ANIMAL_H__
3
4  #include <string>
5
6  using namespace std;
7
8  class Animal {
9
10     // Atributo
11     protected:
12         string nombre;
13
14     // Métodos
15     protected:
16         virtual void moverse() = 0;
17 };
18
19 #endif /* __ANIMAL_H__ */
```

```
C aves.h > ...
1  #ifndef __AVES_H__
2  #define __AVES_H__
3
4  #include "animal.h"
5
6  class Aves: public Animal {
7
8      protected:// Métodos
9          virtual void moverse() = 0;
10 };
11
12 #endif /* __AVES_H__ */
```

```
C peces.h > ...
1  #ifndef __PECES_H__
2  #define __PECES_H__
3
4  #include "animal.h"
5
6  class Peces: public Animal {
7
8      protected:// Métodos
9          virtual void moverse() = 0;
10 };
11
12 #endif /* __PECES_H__ */
```

```
C gaviota.h > ...
1  #ifndef __GAVIOTA_H__
2  #define __GAVIOTA_H__
3
4  #include "aves.h"
5
6  class Gaviota: public Aves {
7
8      // Constructor
9      public:
10          Gaviota(string nombre);
11          Gaviota();
12
13      // Métodos
14      public:
15          void moverse();
16 };
17
18 #endif /* __GAVIOTA_H__ */
```

```
C pelicano.h > ...
1  #ifndef __PELICANO_H__
2  #define __PELICANO_H__
3
4  #include "aves.h"
5
6  class Pelicano: public Aves {
7
8      // Constructor
9      public:
10          Pelicano(string nombre);
11          Pelicano();
12
13      // Métodos
14      public:
15          void moverse();
16 };
17
18 #endif /* __PELICANO_H__ */
```

```
C pez_payaso.h > ...
1  #ifndef __PEZPAYASO_H__
2  #define __PEZPAYASO_H__
3
4  #include "peces.h"
5
6  class PezPayaso: public Peces {
7
8      // Constructor
9      public:
10          PezPayaso(string nombre);
11          PezPayaso();
12
13      // Métodos
14      public:
15          void moverse();
16 };
17
18 #endif /* __PEZPAYASO_H__ */
```

```
C tiburon.h > ...
1  #ifndef __TIBURON_H__
2  #define __TIBURON_H__
3
4  #include "peces.h"
5
6  class Tiburon: public Peces {
7
8      // Constructor
9      public:
10          Tiburon(string nombre);
11          Tiburon();
12
13      // Métodos
14      public:
15          void moverse();
16 };
17
18 #endif /* __TIBURON_H__ */
```

Donde cada archivo cpp correspondiente tiene la siguiente configuración (aplicable a cada especie):

```
G gaviota.cpp > ...
1  #include "gaviota.h"
2  #include <iostream>
3
4  void Gaviota::moverse() {
5      std::cout << "Soy una gaviota llamada " << nombre << " y ";
6      std::cout << "vuelo." << std::endl;
7  }
8
9  Gaviota::Gaviota(string nombre) {
10     this -> nombre = nombre;
11 }
12
13 Gaviota::Gaviota() {
14     this -> nombre = "gaviota";
15 }
```

La clase de tipo template utilizada es la siguiente:

```

C manada.h > ...
1  #ifndef __TEMPLATE_MANADA__
2  #define __TEMPLATE_MANADA__
3
4  #include <list>
5  #include <string>
6
7  template < typename Especie >
8  class Manada {
9
10     private: list < Especie > lista_manada; // Atributo STL
11
12     public: // Métodos
13         void agregar_animal(string nombre);
14         void mover_manada();
15
16    };
17
18    template < typename Especie >
19    void Manada<Especie>::agregar_animal(string nombre) {
20        Especie animal(nombre);
21        lista_manada.push_back(animal);
22    }
23
24    template < typename Especie >
25    void Manada<Especie>::mover_manada() {
26        for(auto i = lista_manada.begin(); i != lista_manada.end(); i++) {
27            i -> moverse();
28        }
29    }
30 #endif

```

De esta manera, se puede implementar una lista con distintos tipos de animales representados por distintos tipos de datos, sin necesidad de crear el objeto específico de cada uno.

Esta función permite agregar cualquier animal (cualquier tipo de dato) sin necesidad de especificarlo cada vez. Sólo será necesario cuando se cree una instancia de clase de tipo Manada. Es decir, en cada manada sólo se permitirá un tipo de especie, determinado por **Especie**.

El main del programa anterior queda de la siguiente manera:

```

C main.cpp > ...
1 #include <iostream>
2
3 #include <list> // STL
4
5 #include "gaviota.h"
6 #include "pelicano.h"
7 #include "pez_payaso.h"
8 #include "tiburon.h"
9
10 #include "manada.h"
11
12 int main() {
13
14     Manada<Gaviota> gaviotas;
15     gaviotas.agregar_animal("Gaviota1");
16     gaviotas.agregar_animal("Gaviota2");
17
18     Manada<Pelicano> pelicanos;
19     pelicanos.agregar_animal("Pelicano1");
20     pelicanos.agregar_animal("Pelicano2");
21
22     Manada<PezPayaso> peces;
23     peces.agregar_animal("Pez1");
24     peces.agregar_animal("Pez2");
25
26     Manada<Tiburon> tiburones;
27     tiburones.agregar_animal("Tiburon1");
28     tiburones.agregar_animal("Tiburon2");
29
30     gaviotas.mover_manada();
31     pelicanos.mover_manada();
32     peces.mover_manada();
33     tiburones.mover_manada();
34
35     return 0;
36 }
```

Manada de gaviotas

Manada de pelícanos

Manada de pez payaso

Manada de tiburones

PROBLEMAS	SALIDA	CONSOLA DE DEPURACIÓN	TERMINAL	JUPYTER	Code	+	~	☰	✖
-----------	--------	-----------------------	----------	---------	------	---	---	---	---

```
PS C:\Users\Guill\Desktop\Proyectos\Templates c++> cd "c:\Users\Guill\Desktop\Proyectos\Templates c++\" ; if (
$?) { g++ *.cpp -o main } ; if ($?) { ./main }
Soy una gaviota llamada Gaviota1 y vuelo.
Soy una gaviota llamada Gaviota2 y vuelo.
Soy un pelícano llamado Pelicano1 y vuelo.
Soy un pelícano llamado Pelicano2 y vuelo.
Soy un pez payaso llamado Pez1 y nado.
Soy un pez payaso llamado Pez2 y nado.
Soy un tiburón llamado Tiburon1 y nado.
Soy un tiburón llamado Tiburon2 y nado.
```

Estructuras lineales

(Listas, pilas y colas)

Una lista es una estructura de datos lineal flexible, ya que puede crecer a medida que se insertan nuevos elementos o acortarse cuando se borren.

- Pilas representada de abajo hacia arriba
- Colas representada de izquierda a derecha
- Listas propiamente dichas

Cada elemento de una lista está conformado por **nodos**, donde cada nodo (objeto de una clase) contiene el dato que se desea almacenar y un **apuntador** (puntero) que indica cuál es el siguiente elemento.

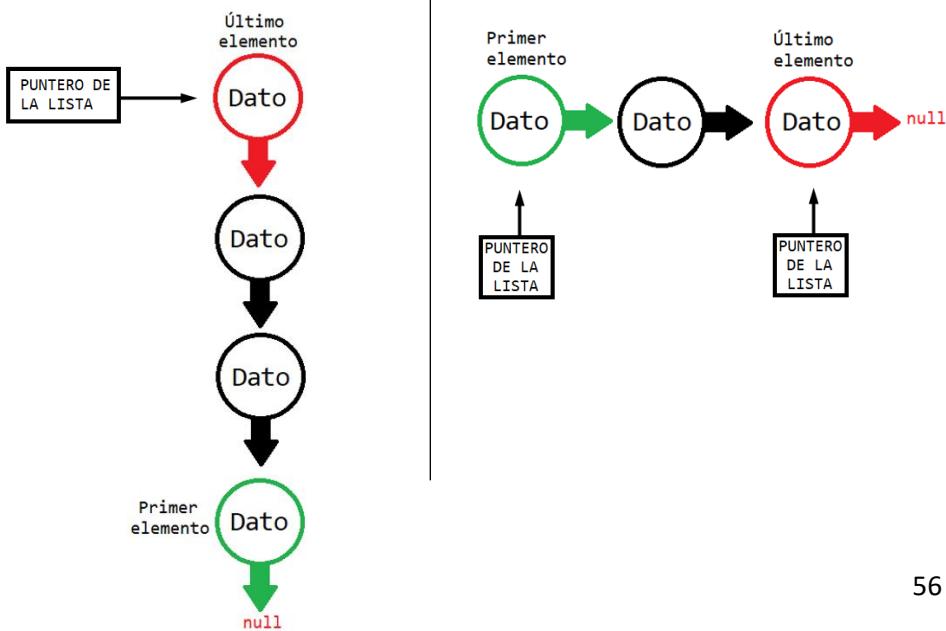


Operaciones de una lista

- Alta
- Baja
- Consulta
- Modificación (opcional)

En general, la operación de baja también realiza la operación de consulta.

	Pilas	Colas
Alta	Agrega al último elemento	Agrega al último elemento
Baja	Elimina el último elemento	Elimina un elemento del principio
Consulta	Devuelve el último elemento	Devuelve el elemento del principio



Implementación del nodo

HEADER

```
#ifndef __NODO_H__
#define __NODO_H__

typedef char Dato;

class Nodo {
    // Atributos
private:
    Dato dato;
    Nodo* siguiente;

    // Metodos
public:
    // Constructor con parámetro
    // Pre-condiciones: -
    // Pos-condiciones: crea un nodo con dato = info
    Nodo(Dato info);

    // Establece un nuevo Nodo siguiente
    // Pre-condiciones: -
    // Pos-condiciones: establece siguiente en nodo
    void cambiar_siguiente(Nodo* nodo);

    // Mostrar dato
    // Pre-condiciones: -
    // Pos-condiciones: devuelve dato
    Dato obtener_dato();

    // Mostrar el siguiente nodo
    // Pre-condiciones: -
    // Pos-condiciones: devuelve siguiente
    Nodo* obtener_siguiente();
};

#endif /* __NODO_H__ */
```

CPP

```
#include "nodo.h"

Nodo::Nodo(Dato info) {
    dato = info;
    siguiente = 0;
}

void Nodo::cambiar_siguiente(Nodo* nodo) {
    siguiente = nodo;
}

Dato Nodo::obtener_dato() {
    return dato;
}

Nodo* Nodo::obtener_siguiente() {
    return siguiente;
}
```

Pilas

- Usos
 - Funciones recursivas
 - Editores de texto, cuando uno presiona el botón de deshacer, vuelve al estado anterior inmediato. Esos cambios se guardan en una pila.
 - La pila (stack) que usa la memoria RAM para almacenar variables locales.
 - Los navegadores de Internet, cuando uno está en un sitio y presiona un link, va a otro sitio. Y cuando presiona la flecha regresar, vuelve a la página anterior. Los sitios visitados los guarda en una pila.
- Implementación
 - Estáticas
 - Dinámicas (usando memoria del Heap)

HEADER de una Pila

```
#ifndef __PILA_H__
#define __PILA_H__


#include "nodo.h"

const int CAPACIDAD_INICIAL = 10;

class Pila {

    // Atributos (privados)
private:
    Nodo* ultimo;

    // Interfaz
public:
    // Constructor con parámetro con valor por defecto
    // Pre-condicion: -
    // Pos-condicion: construye una Pila vacía
    Pila();

    // Destructor
    // Pre-condicion: -
    // Pos-condicion: libera memoria utilizada por los elementos de la pila
    ~Pila();

    // Agrega dato como último elemento a la pila
    // Pre-condicion: -
    // Pos-condicion: Agrega el elemento dato al final de la Pila
    void alta(Dato dato);

    // Elimina el último elemento de la pila y lo devuelve
    // Pre-condicion: Pila no debe estar vacía (cantidad == 0)
    // Pos-condicion: Elimina el último elemento de datos y lo devuelve
    Dato baja();

    // Devuelve el último elemento de la Pila
    // Pre-condicion: Pila no debe estar vacía (cantidad == 0)
    // Pos-condicion: Devuelve el último elemento de la Pila
    Dato consulta();

    // Indica si la Pila está vacía
    // Pre-condicion: -
    // Pos-condicion: devuelve Verdadero si la pila está vacía
    //                 devuelve Falso si la pila no está vacía
    bool esta_vacia();

};

#endif /* __PILA_H__ */
```



CPP de una Pila

```
#include "pila.h"

Pila::Pila() {
    ultimo = 0;
}

Pila::~Pila() {
    while(! esta_vacia())
        baja();
}

void Pila::alta(Dato dato) {
    Nodo* nodo = new Nodo(dato);
    nodo -> cambiar_siguiente(ultimo);
    ultimo = nodo;
}

Dato Pila::consulta() {
    return (ultimo -> obtener_dato());
}

Dato Pila::baja() {
    Dato dato = ultimo -> obtener_dato();
    Nodo* auxiliar = ultimo;
    ultimo = auxiliar -> obtener_siguiente();
    delete auxiliar;
    return dato;
}

bool Pila::esta_vacia() {
    return (ultimo == 0);
}
```

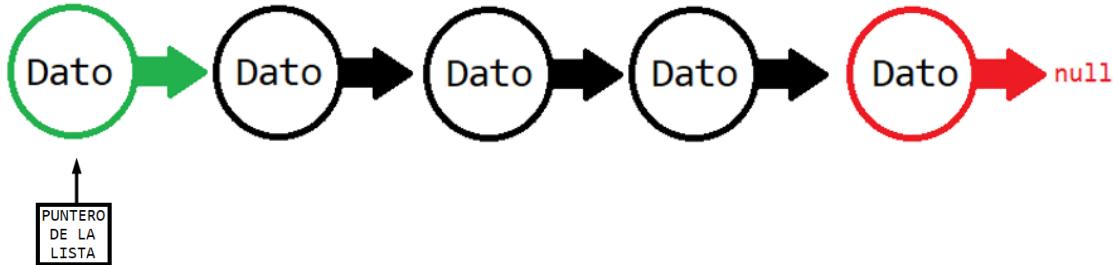


Cola

Lista

Se puede añadir un elemento en cualquier parte de la lista.

Se puede eliminar un elemento de cualquier parte de la lista.



Si se desea tener una lista ordenada (por ejemplo, en orden alfabético), hay que adaptar la operación de alta acorde al orden deseado de la lista.

- Alta

- Desordenado:

- ✓ Al principio
- ✓ Al final
- ✓ En algún lugar del medio

El más eficiente será en donde haya un puntero de la lista.

- Ordenado

- ✓ Alfabéticamente
- ✓ De menor a mayor
- ✓ Según decisión del usuario (*con un parámetro*)

- Baja

- Desordenado

- ✓ Del principio
- ✓ Del final
- ✓ De algún lugar del medio

El más eficiente será en donde haya un puntero de la lista.

- Ordenado

- ✓ De un elemento en particular (*con un parámetro de nombre*)
 - De la primera recurrencia
 - De la última recurrencia
 - De todas las recurrencias

}

Para eliminar elementos

- ✓ De una posición de la lista específica (*con un parámetro numérico*)

Complejidad Temporal

Tiempo de ejecución de un algoritmo. Varía en base al tipo de computadora, por dicho motivo, se utiliza sólo las operaciones elementales para analizarse.

Existen dos formas de estudiarla

- Posteriori (empírico) Se mide el tiempo de ejecución de un programa (depende de la computadora donde se ejecuta).
- A priori (teórico) Determina una función que se relaciona con el tiempo de ejecución del algoritmo para cierto valor de entrada.

Conceptos

		Ejemplo
Problema	Necesidad inicial por la que se necesita una solución.	Ordenar un vector de elementos
Ejemplar	Problema específico del que se busca solución.	[2, 5, 7, 1, 3] con n = 5
Algoritmo	Serie de pasos ordenados y finitos cuyo objetivo es hallar la solución al problema para cualquier tipo de ejemplar.	Selección Burbujeo Inserción

El tamaño del problema de

- Función factorial: n (número del que se busca el factorial)
- Buscar un elemento en un vector: tamaño del vector
- Ordenamiento de un vector: tamaño del vector

Tiempo de ejecución

Depende de:

- Datos de entrada
- Algoritmo
- Código que genere el compilador o el intérprete
- Arquitectura de la máquina

• Principio de invarianza

Dado un algoritmo y dos implementaciones del mismo (en la misma máquina o en dos distintas) que tardarán un tiempo $T_1(n)$ y $T_2(n)$ respectivamente, entonces existe una constante real $C > 0$, y $n_0 \in N$ tales que $\forall n \geq n_0$ se verifica que:

$$T_1(n) \leq C \cdot T_2(n)$$

Por ejemplo

Algoritmo: ordenamiento por selección

Ejemplar: vector de 100 millones de elementos aleatorios

Implementación: en C++ y en Python
en M₁ (6 GB RAM) y en M₂ (2 GB RAM)

	M1	M2
C++	40s	75s
Python	1211s	2325s

La constante de invarianza entre M₁ y M₂ es aproximadamente C = 2.

Por otro lado, la constante de invarianza entre un lenguaje y el otro es C = 30.

Operaciones elementales

- Asignación de variables
- Operaciones aritméticas básicas (suma, resta, división, multiplicación)
- Comparaciones lógicas
- Llamadas y retornos de funciones
- Acceso a una estructura indexada (vector)

Un ejemplo es:

```
int a, b = 5; // asignación           -> 1 O.E.
a = b + 1; // asignación y suma     -> 2 O.E.
a++; /* a = a + 1 */ // asignación y suma -> 2 O.E.
```

} } T(n) = 5
n es el tamaño del problema

Puede haber algoritmos que dependan de un dato externo (como una lectura de un csv o el ingreso de datos por parte del usuario, lo cual hace que la cantidad de operaciones elementales varíe. Por ejemplo:

```
int a, b = 2; // asignación      -> 1 O.E.
cin >> a; // asignación y suma -> 2 O.E.
if(a > 5) { // comparación    -> 1 O.E.
    a = a + 3; // asignación y suma -> 2 O.E.
    b = a + 1; // asignación y suma -> 2 O.E.
} else
    b = 1; // asignación       -> 1 O.E.
```

Donde, si a es mayor que 5, se obtendría un total de $T(n) = 7$.

Sin embargo, si a es menor o igual a 5, se tendría $T(n) = 5$.

Podemos definir un “mejor” y “peor” caso según la traza (secuencia del algoritmo).

- Peor caso traza del algoritmo donde se ejecuta la cantidad máxima de operaciones.
- Mejor caso traza del algoritmo donde se ejecuta la cantidad mínima de operaciones.
- Promedio corresponde al promedio de todas las trazas ponderado según su probabilidad. Si la probabilidad es 50%, se divide por 2 el $T(n)$.

Es importante estudiar siempre el peor caso, debido a que define el tiempo máximo de ejecución del programa (una cota superior).

```

// n es el tamaño del vector
int busqueda_en_vector(int vector[], int n, int dato_buscado) {
    bool encontrado = false;      // 1 OE
    int i = 0;                   // 1 OE

    while((i < n) && (!encontrado)) { // 3 OE (3 comparaciones incluida el &&
        if(vector[i] == dato_buscado) // 2 OE (acceso al vector y comparación ==)
            encontrado = true;     // 1 OE (asignación)
        else i++;                 // 2 OE (asignación y suma)
    } // Al salir del while, hace una última comparación en el while que dará false -> 1 OE

    if(encontrado)             // 1 OE
        return i;               // 1 OE
    return -1;                  // 1 OE
}

```

Medidas asintóticas

- Cota superior: big O

Se calcula desde un valor en específico (un n_0) y se puede multiplicar por una constante C.

- Cota inferior: Ω (omega)

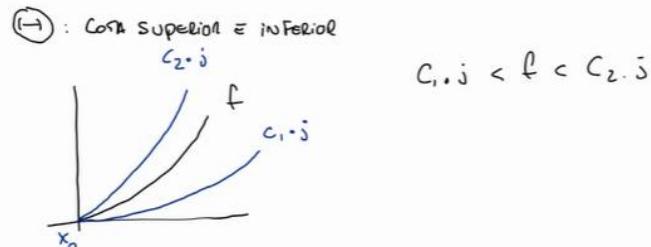
Se calcula desde un valor en específico (un n_0).

- Orden exacto: Θ (theta) (cota superior e inferior)

La función se encuentra siempre por encima y por debajo desde un valor en específico (un X_0). Se puede multiplicar por una constante C_1 y C_2 .

Donde Y es la cantidad de operaciones elementales.

Donde X es el tiempo





Espacial (memoria necesaria para ejecutar un algoritmo)

Recursión

En general, es una función que se llama a sí misma al menos una vez, de forma directa o indirecta.



Clasificaciones

Según dónde está el llamado recursivo

- De cola El llamado recursivo está al final de la función
- No de cola El llamado recursivo no está al final de la función

Por el modo de invocación

- Directa Se llama a sí misma.
- Indirecta Hay una función de por medio.
- Anidada Para hacer el llamado recursivo, requiere resolver previamente otro llamado recursivo.

Según la cantidad de llamados

- Simple Sólo una vez se llama a sí misma en cada recursión.
- Múltiple Se llama a sí misma varias veces en cada recursión.

Recordando que las funciones ocupan espacio en el Stack de la memoria, la cual no es tan grande. Por tanto, si una función recursiva se llama múltiples veces, puede llegar a llenarse la memoria y dejar de funcionar.

Tail Call Optimization

La optimización de los llamados de cola permite manejar mejor la memoria del stack para evitar el sobreuso de la misma.

Por ejemplo, para el caso del cálculo de un factorial.

```
// Pre-condición: n >= 0
int factorial(int n) {
    if(n == 0 || n == 1) return 1;// condición de corte
    return (factorial(n - 1) * n); // llamado a sí misma
}
```

En este caso, en cada iteración de la función, se crea una nueva función en la memoria stack, guardando la información de la función anterior.

La forma de optimizarlo es la siguiente:

```
int _factorial(int numero, int resultado) {

    // Condición de corte
    if(numero == 1 || numero == 0) return resultado;

    // Llamado a sí misma
    return _factorial(numero - 1, resultado * numero);
}

// Pre-condición: n >= 0
int factorial(int n) {
    return _factorial(n, 1);
}
```

En este caso, por cada iteración de la función `_factorial`, se almacena la información dentro de la función `factorial` (la inicial), por lo tanto, se elimina la información de `_factorial` liberando la memoria del stack para poder reutilizar ese mismo espacio con una nueva función `_factorial` (recursiva) sin necesidad de llenar la memoria de muchas funciones `_factorial` repetidas como en el primer caso.

Divide y vencerás

Si el tamaño del problema a tratar es $N > L$, siendo L un valor arbitrario, se debe dividir el problema en m subproblemas no solapados y que tengan la misma estructura que el problema original. Luego, a cada uno de esos m subproblemas se los dividirá nuevamente hasta que cada uno alcance un tamaño L o menor, los cuales se resolverán por cualquier otro método.

Por último, la solución del problema original se obtiene por combinación de las m soluciones obtenidas de los subproblemas en que se lo había dividido.

Divide un problema grande en problemas más pequeños hasta llegar a un “caso base”. Luego, resolviendo los problemas pequeños se llega a resolver el problema general / original.

Búsqueda binaria

Se utiliza en vectores / listas / arreglos ordenados para la búsqueda de un elemento específico.

Por ejemplo, se tiene un vector:

0	1	2	3	4	5	6	7
1	3	5	7	8	9	10	14

Y se quiere buscar el elemento con valor 10.

centro

Se comienza buscando en el elemento del centro (en este caso, 4º). Se le pregunta si es el número 10. Como no lo es, y el 8 es menor a 10, se sabe que el elemento buscado puede estar en vectores más altos.

El siguiente paso será analizar los elementos siguientes del subvector:

5	6	7
9	10	14

De esta manera, se busca el centro [el cual se puede calcular como 7 (el último) + 5 (el siguiente del centro anterior) dividido dos]. En este caso: 6º. Se le consulta si es el 10 y entonces lo retorna ya que sí lo es.

La cantidad de comparaciones que se realiza es $\log_2(n)$ porque en cada comparación se “ pierden” la mitad de los elementos.

Al encontrar el resultado buscado en el subproblema (subvector), se obtiene el resultado del problema (vector) original.

Según lo visto en “divide y vencerás”, este nuevo sub-vector es un subproblema a resolver.

Tomó una subdivisión

$O(\log n)$

Potencia

Forma bruta: $2^4 : 2 \times 2 \times 2 \times 2$ (requiere 4 operaciones elementales)

Forma optimizada: 2^4

↓
Subdividir el problema

$$2^2 = 2^1 \times 2^1$$

$$2^1 = 2$$

```
int potencia(int base, int exponente) {
    if(exponente == 0) return 1;
    if(exponente == 1) return base;

    int resultado = potencia(base, exponente / 2);
    resultado *= resultado;

    if((exponente % 2) == 1)
        resultado *= base;

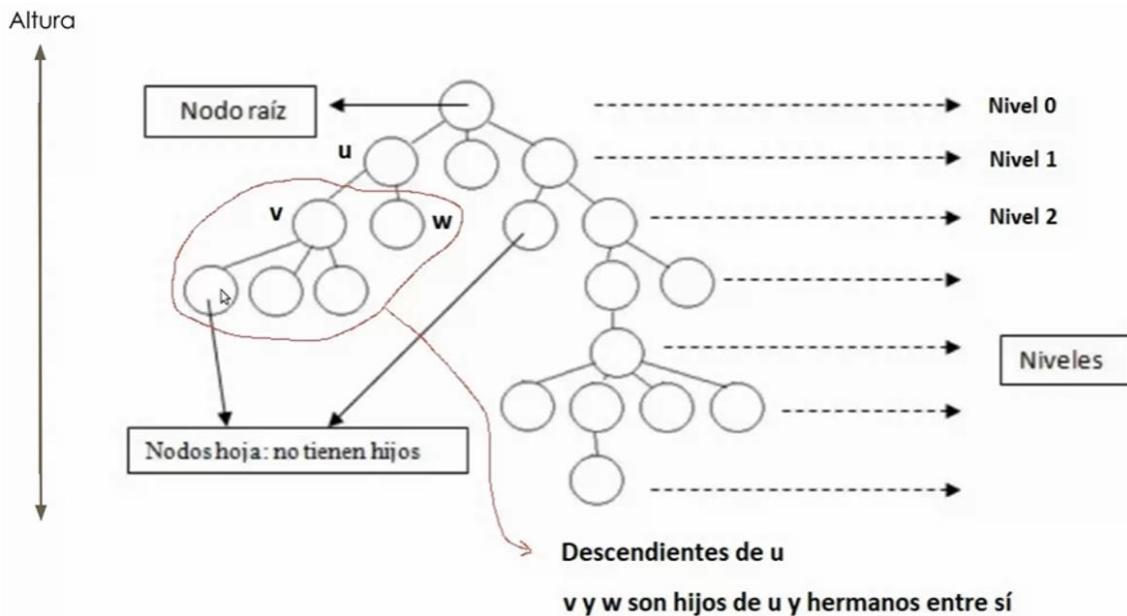
    return resultado;
}
```

Ordenamiento (mergesort o quicksort)

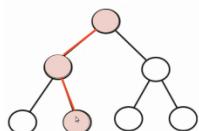
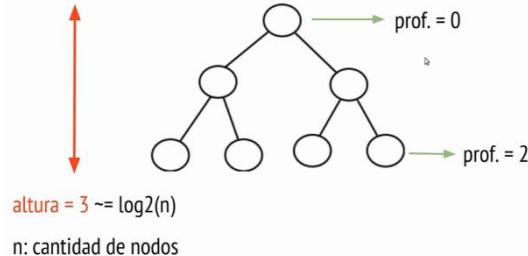
```
void ordenar(vec, min, max){
    if (max - min > 1) {
        particionar vec en vec_1 y vec_2
        y obtener punto_corte
        ordenar(vec_1, min, punto_corte)
        ordenar(vec_2, punto_corte+1, max)
        combinar(vec_1, vec_2)
    }
}
```

Árboles

Estructura de datos que tiene un único “padre” con varios “hijos”



- Camino: secuencia de nodos que se encuentren entre-lazados entre sí
- Altura y profundidad



Es logaritmo en base dos porque cada nodo tiene 2 hijos.

Si el árbol no es vacío, existe un único nodo que no tiene padre (el nodo raíz).

Cada nodo puede tener, a lo sumo, **n** hijos. Se llama árbol **n**-ario.

Si el árbol puede tener a lo sumo 2 hijos, se considera árbol binario.

Se dice que el árbol está completo si cada nodo tiene el máximo cantidad de hijos posibles (n).

Cuando un nodo no tiene hijos, se llama “hoja”.

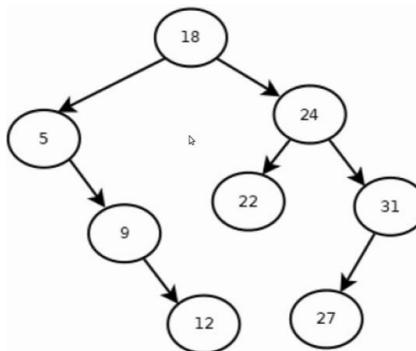
Un árbol degenerado es aquel arbol que tiene 1 solo hijo por nodo (una lista).

Árboles binarios de búsqueda (ABB)

Es un árbol binario en donde cada clave será más grande que las claves que estén en el subárbol izquierdo (si hubiera) y más chica que las del subárbol derecho.

Cada subárbol es, a su vez, un ABB.

Si llegasen las claves 18 – 5 – 9 – 24 – 12 – 31 – 22 – 27 (en ese orden), se construirá el siguiente árbol:



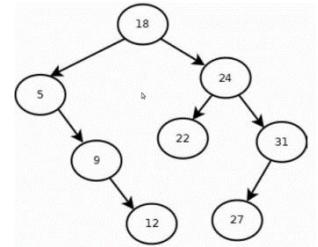
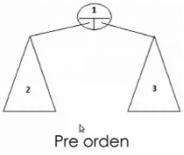
1. El **18** es el primero y por tanto será la raíz del árbol (el único sin padre).
2. El **5** es menor que el **18** y por tanto se ubicará en la izquierda.
3. El **9** es menor que el **18**, se ubicará en la izquierda pero como ya existe un hijo en la izquierda, se ubicará como “nieto” (hijo del hijo) del lado derecho porque es mayor que **5**.
4. El **24** es mayor que el **18**, por tanto se ubicará en la derecha.
5. El **12** es menor que el **18**, por tanto se ubicará como el último hijo en la izquierda, y como es mayor que **9**, se ubicará por la derecha.
6. El **31** es mayor que el **18**, por tanto se ubicará en la derecha. Como es mayor que el **24**, será un hijo del lado derecho.
7. El **22** es mayor que el **18**, por tanto irá por la derecha. Es menor que el **24**, por tanto se ubicará por la izquierda.
8. El **27** es mayor que el **18**, por tanto irá por la derecha. Es mayor que el **24**, por tanto se irá por la derecha. Como es menor que el **31** ya ubicado allí, se irá por la izquierda como su hijo.

Recorridos de un ABB

Pre-Orden

Primero se recorre la raíz, luego el subárbol izquierdo y luego el derecho. Se aplica recursivamente en cada nivel / profundidad.

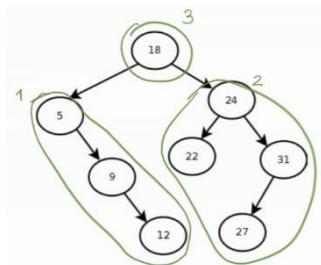
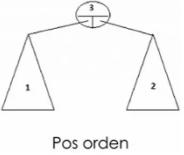
En este caso, el orden será: 18 – 5 – 9 – 12 – 24 – 22 – 31 – 27



Pos-Orden

Comienza recorriendo el sub-árbol izquierdo, luego el sub-árbol derecho y finalmente la raíz.

En este caso, el orden será: 12 – 9 – 5 – 22 – 27 – 31 – 24 – 18

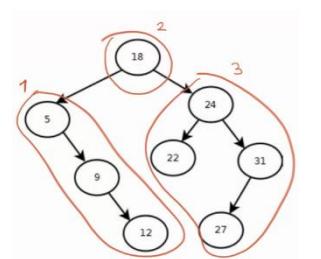
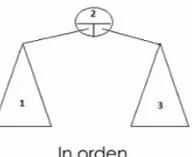


In-Orden

Comienza recorriendo el sub-árbol izquierdo, luego la raíz y finalmente el subárbol derecho.

En este caso, el orden será: 5 – 9 – 12 – 18 – 22 – 24 – 27 – 31

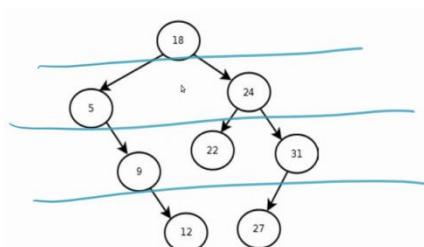
De esta manera, se ordenan los nodos en orden creciente.



En ancho

Imprime por nivel en orden de izquierda a derecha.

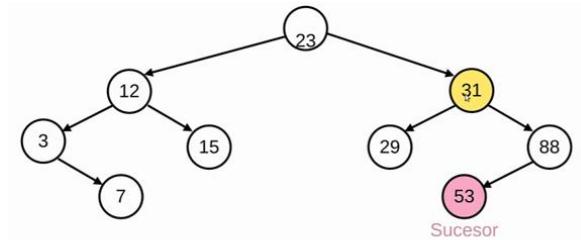
En este caso, el orden será: 18 – 5 – 24 – 9 – 22 – 31 – 12 – 27



Todos estos recorridos son de tipo recursivo.

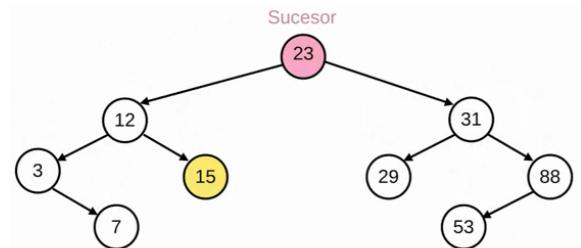
Sucesor

- Si el nodo tiene un subárbol derecho, su sucesor será el que tenga el mínimo valor en ese subárbol derecho.



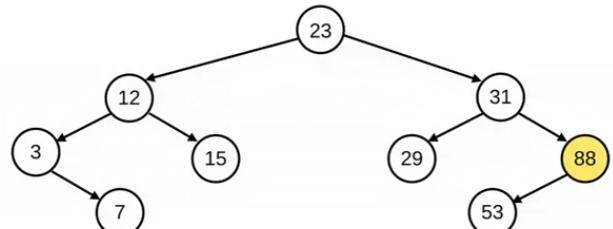
53 es el numero mas chico del subarbol derecho a 31

- Si el nodo no tiene un subárbol derecho, tenemos que atravesar a los ancestros del nodo hasta encontrar el primer nodo que sea mayor al actual.



23 es el primer ancestro que es mas grande que 15

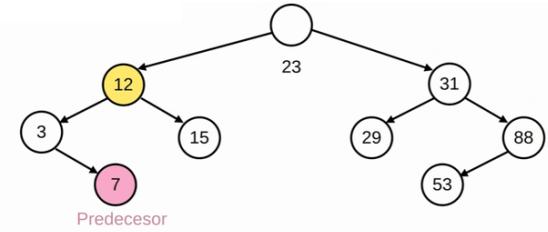
- Si el nodo es el número máximo en el árbol, entonces no tiene sucesor.



88 es el numero máximo. No tiene sucesor.

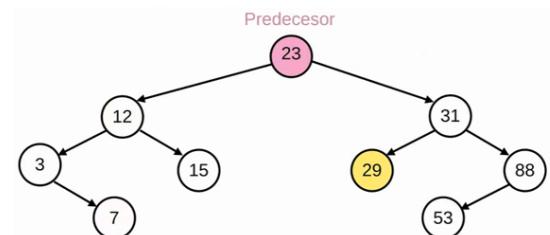
Predecesor

- Si el nodo tiene un subárbol izquierdo, su predecesor será el que tenga el máximo valor en ese subárbol izquierdo.



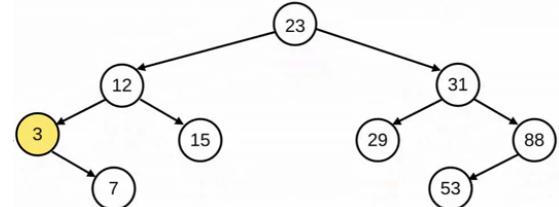
7 es el numero mas grande del subárbol izquierdo a 12

- Si el nodo no tiene un subárbol izquierdo, tenemos que atravesar a los ancestros del nodo hasta encontrar el primer nodo que sea menor al actual.



23 es el primer ancestro que es mas mas chico que 29

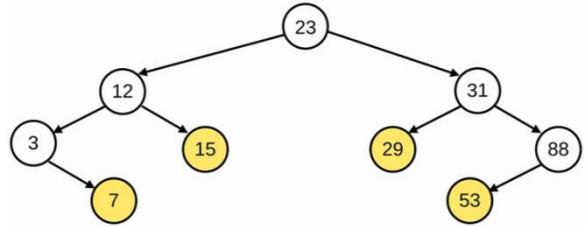
- Si el nodo es el número mínimo en el árbol, entonces no tiene predecesor.



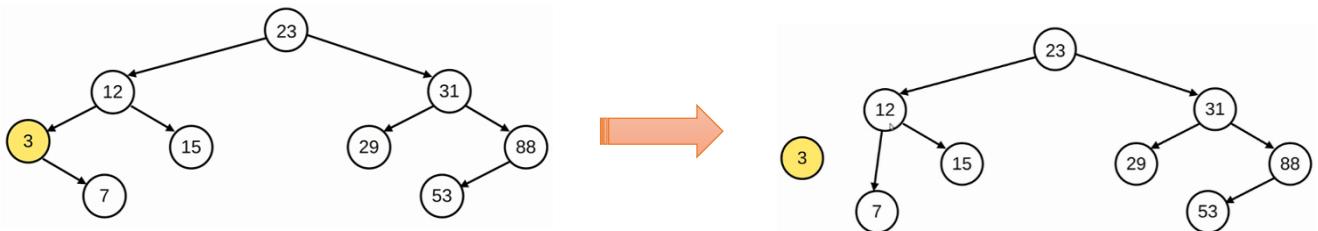
3 es el numero mínimo. No tiene predecesor.

Remove

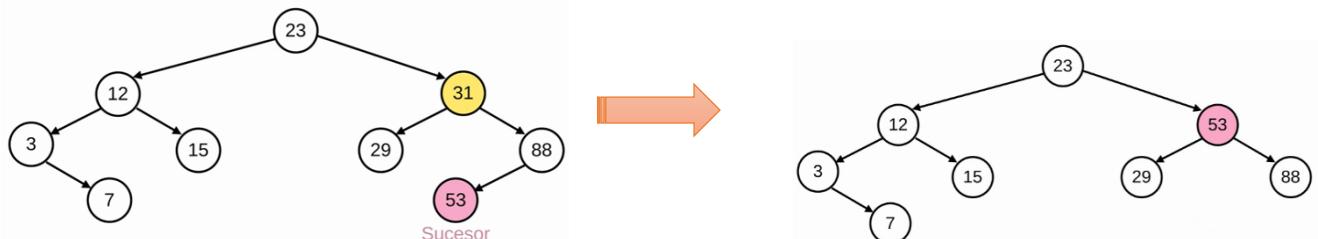
- El nodo a eliminar es una hoja, es el caso más simple. El nodo padre quedará apuntando a null.



- Eliminar un nodo que sólo tiene un hijo.



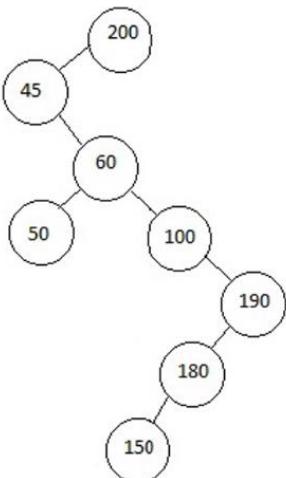
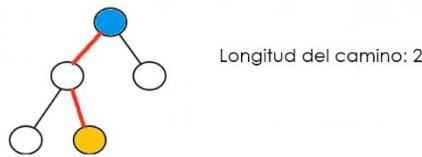
- Eliminar un nodo que tiene dos hijos



Arboles binarios balanceados

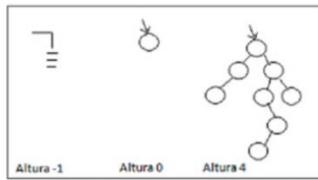
La complejidad de búsqueda en un árbol balanceado (con forma de triángulo) es de $\log_2(n)$

- Longitud de un camino: es la cantidad de aristas que hay que hay que recorrer para ir desde el nodo raíz hasta un nodo cualquiera.

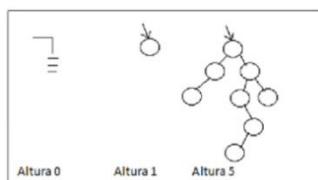


- Peso: cantidad de nodos que tiene un determinado bloque del árbol
- Altura

Este árbol, si se le quitara el nodo "50", queda un árbol degenerado (una lista). La búsqueda de datos es ineficiente, con una complejidad de n

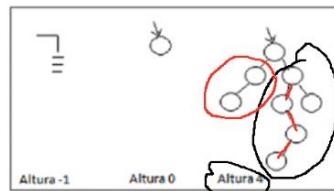


Cormen: longitud del camino más largo.

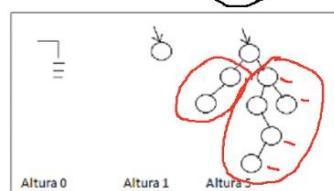


Drozdek: cantidad de niveles.

Para el balanceo de un árbol, vamos a requerir medir el subárbol derecho y el subárbol izquierdo y calcular su diferencia (por lo tanto, da igual qué autor utilicemos para medir la altura).



$$3 - 1 = 2$$



$$4 - 2 = 2$$

Diferencias de alturas

- Nodo A = $\text{abs}(4 - 2) = 2$
- Nodo B = $\text{abs}(0 - 1) = 1$
- Nodo C = $\text{abs}(1 - 3) = 2$
- Nodo D = $\text{abs}(0 - 0) = 0$
- Nodo E = $\text{abs}(2 - 0) = 2$
- Nodo F = $\text{abs}(0 - 0) = 0$
- Nodo G = $\text{abs}(0 - 1) = 1$
- Nodo H = $\text{abs}(0 - 0) = 0$

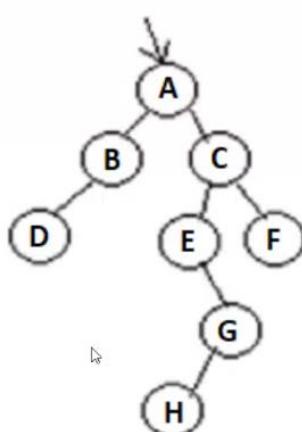


ABB balanceado por altura



Definición:



Un ABB está balanceado por su altura si y solo si, para todo nodo N del árbol se verifica que:



$$| h(\text{subárbol_derecho}(N)) - h(\text{subárbol_izquierdo}(N)) | \leq d$$



Siendo d un entero mayor a 0.

AVL



Llamados así por haber sido diseñados por Adelson-Velski y Landis.



Cada nodo tiene un factor de balanceo (FB)



FB puede tomar solo tres valores: -1, 0, 1.

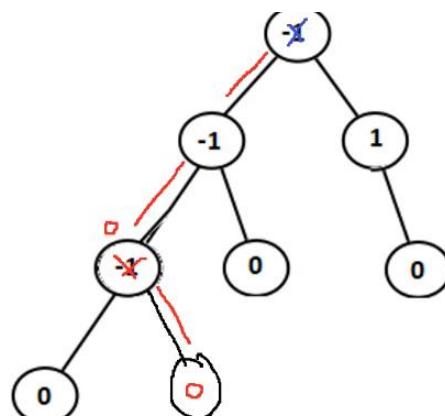


Si FB toma algún valor mayor a 1 o menor que -1, hay que rebalancear.

Para ser considerado un **AVL**, la diferencia de altura no debe ser mayor que 1. Es decir, subárbol derecho – subárbol izquierdo ≤ 1 .

Es decir, $d = 1$

Para que un árbol sea balanceado, al agregar un nodo es necesario tener en cuenta el “factor de balanceo” del árbol para saber dónde se puede agregar el nodo dentro del arbol. Si quedara desbalanceado, se “ajustan” los punteros para que el árbol quede balanceado. Cada nodo debe tener un “factor de balanceo” entre -1 y 1.



Rebalanceo



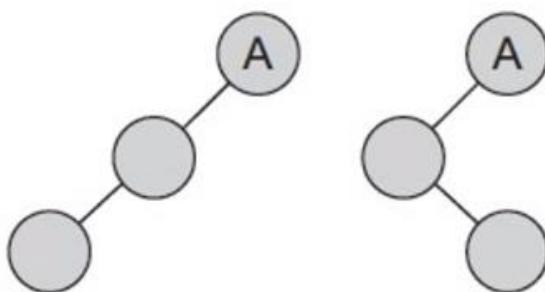
Se da el alta (o baja) habitual.



Se recalculan los FB desde el nodo insertado hasta la raíz.



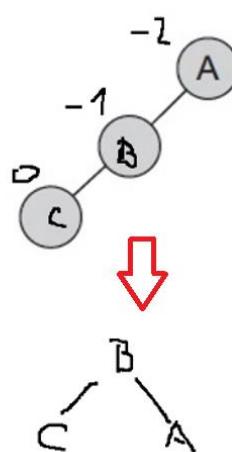
Si hay desbalanceo se procede a realizar las rotaciones correspondientes.



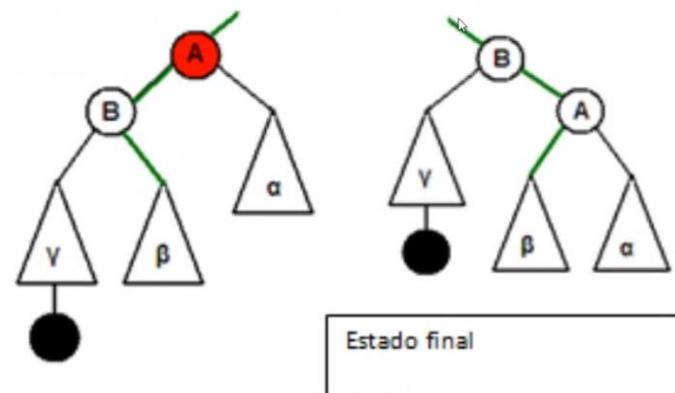
- Rotación simple derecha
- Rotación doble: izquierda – derecha

Un ejemplo:

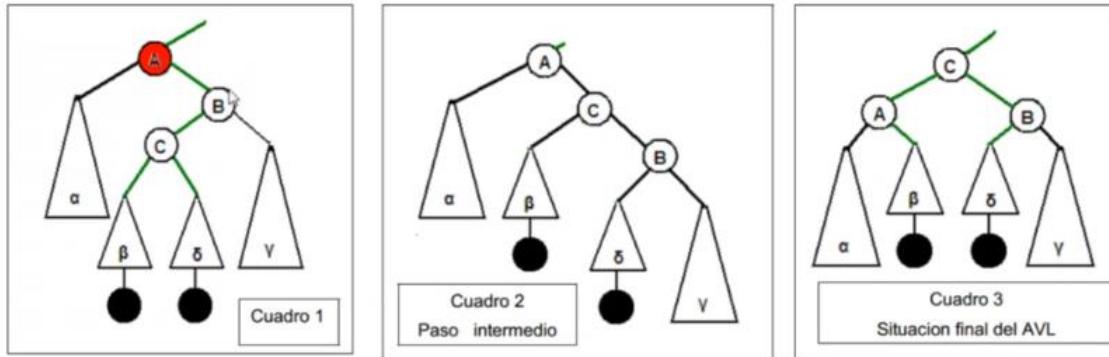
Rotación simple derecha



Rotación SD con hijos



Rotación D-DI con hijos



Rotaciones – FB



Rotación	FB (p)	FB (q)	FB (r)
SD	- 2	0 o - 1	0
SI	2	0 o 1	0
D-ID	- 2	1	0
D-DI	2	- 1	0

Borrar un nodo



Se borra de la misma forma que un árbol ABB.



Una extracción genera un cambio en el factor de balanceo del nodo padre de esa rama.



Requiere las mismas rotaciones simples y dobles de la inserción.



El costo de este algoritmo es proporcional a la altura del arbol, o sea, $O(\log n)$.

Método en el borrado



BUSCAR NODO X
A BORRAR (IGUAL
QUE EN ABB)



BUSCAR
REEMPLAZO DE X
EN NODO HOJA



REEMPLAZAR X
CON Y



ELIMINAR Y
DE LA HOJA



ACTUALIZAR FB.

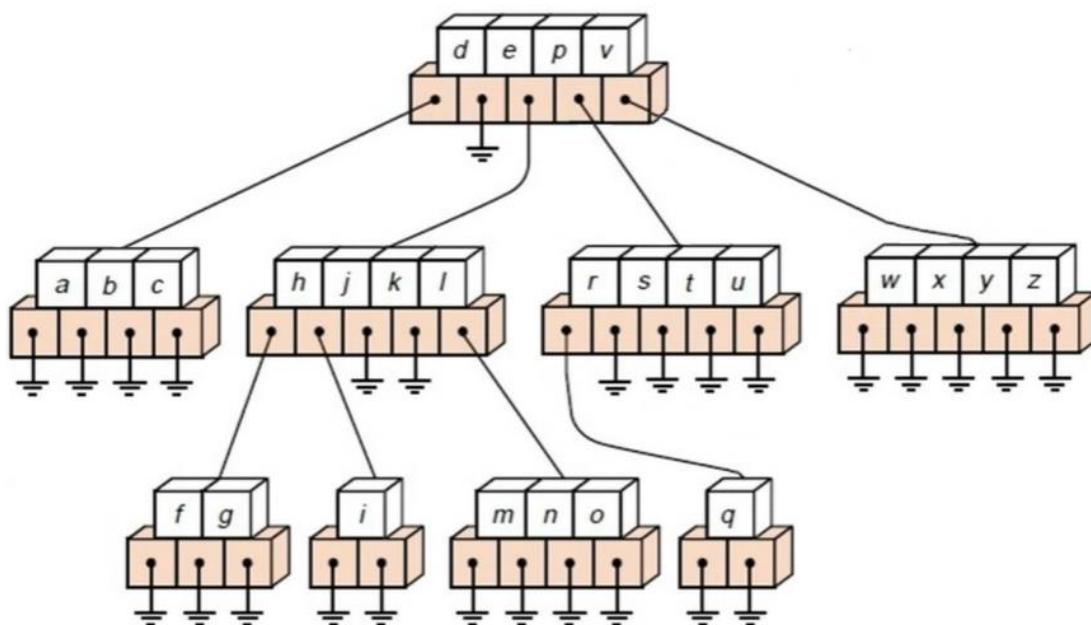
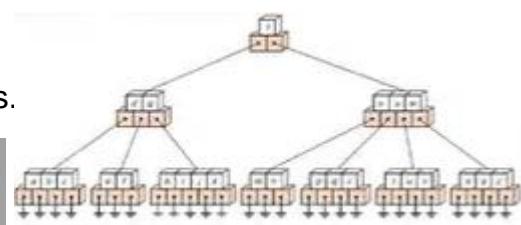


SUBIR Y
EQUILIBRAR.

Árbol multivias

Es un árbol con n cantidad de claves y m cantidad de hijos.

- ▶ Problema: cuando los datos se guardan en disco, se busca minimizar los accesos al mismo.
- ▶ Los ABB evolucionan a árboles de búsqueda de múltiples (m) vías:
- ▶ Un nodo tiene a lo sumo m hijos y $m - 1$ claves.
- ▶ Las claves en cada nodo están ordenadas de menor a mayor en espacios contiguos.
- ▶ Cada clave tiene un puntero izquierdo a un nodo con claves menores (puede ser null) y uno derecho a un nodo con claves mayores (puede ser null).



Si las claves se ingresaran en orden alfabético (en este caso), se estaría creando una especie de “lista”, por lo tanto, importa le orden en que se ingresen los datos.

Árbol B

Definición:

Un árbol B de orden m es un árbol de búsqueda de m-vías en el que

1. La raíz, o bien es hoja, o bien tiene al menos dos hijos.
2. Todas las hojas están al mismo nivel.
3. Cada nodo, a excepción quizás del nodo raíz, tiene $k - 1$ claves, donde $m/2 \leq k \leq m$.
4. Cada nodo interno, a excepción quizás del nodo raíz, tiene k hijos, donde $m/2 \leq k \leq m$.

$$5\text{-vías} \Rightarrow m = 5$$

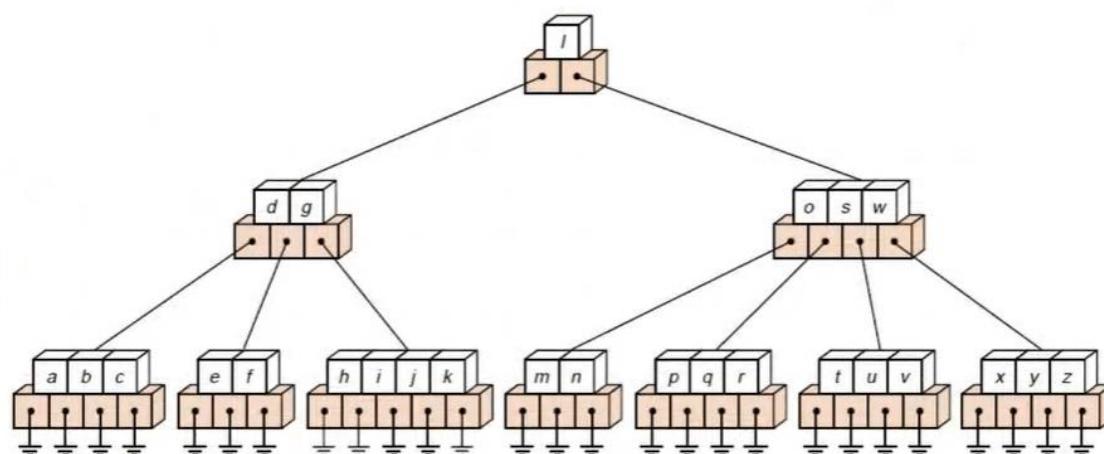
$$\frac{5}{2} \leq k \leq 5 \rightarrow 2,5 \leq k \leq 5 \rightarrow 3 \leq k \leq 5$$

sólo valores
enteros

\Rightarrow 2 a 4 claves y 3 a 5 hijos, a excepción de la

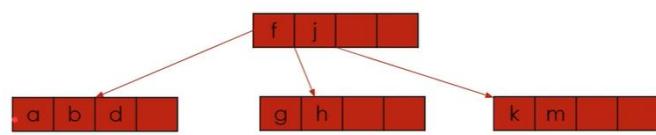
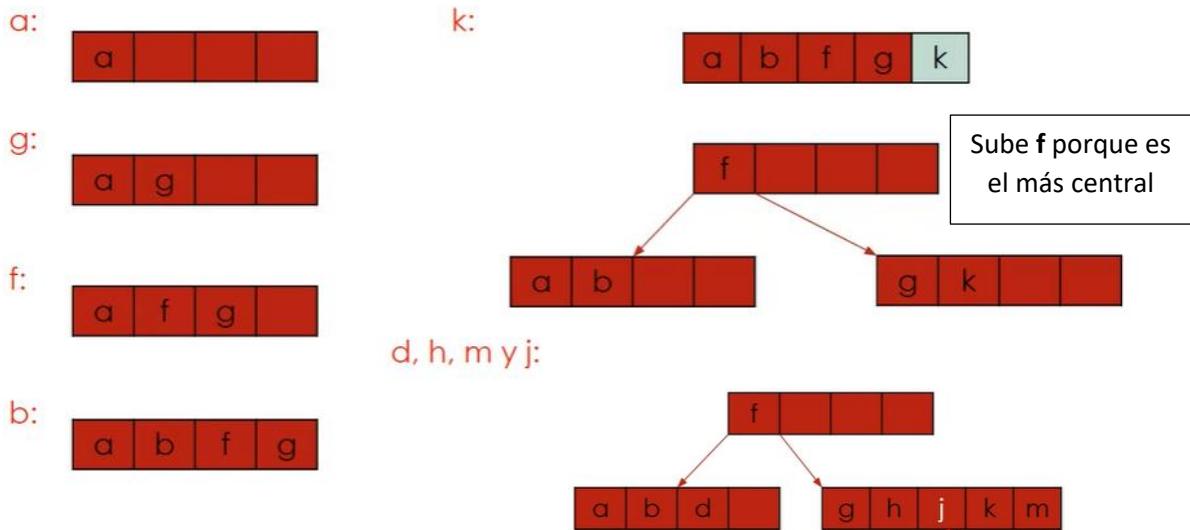
Raiz

La excepción que se da en la raíz es cuando no hay suficientes datos para que el árbol sea creado (o bien que se eliminaron).

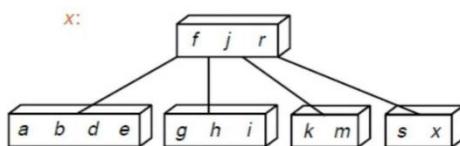
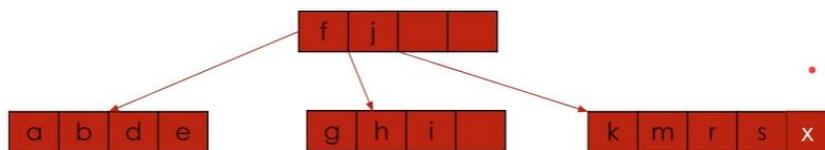


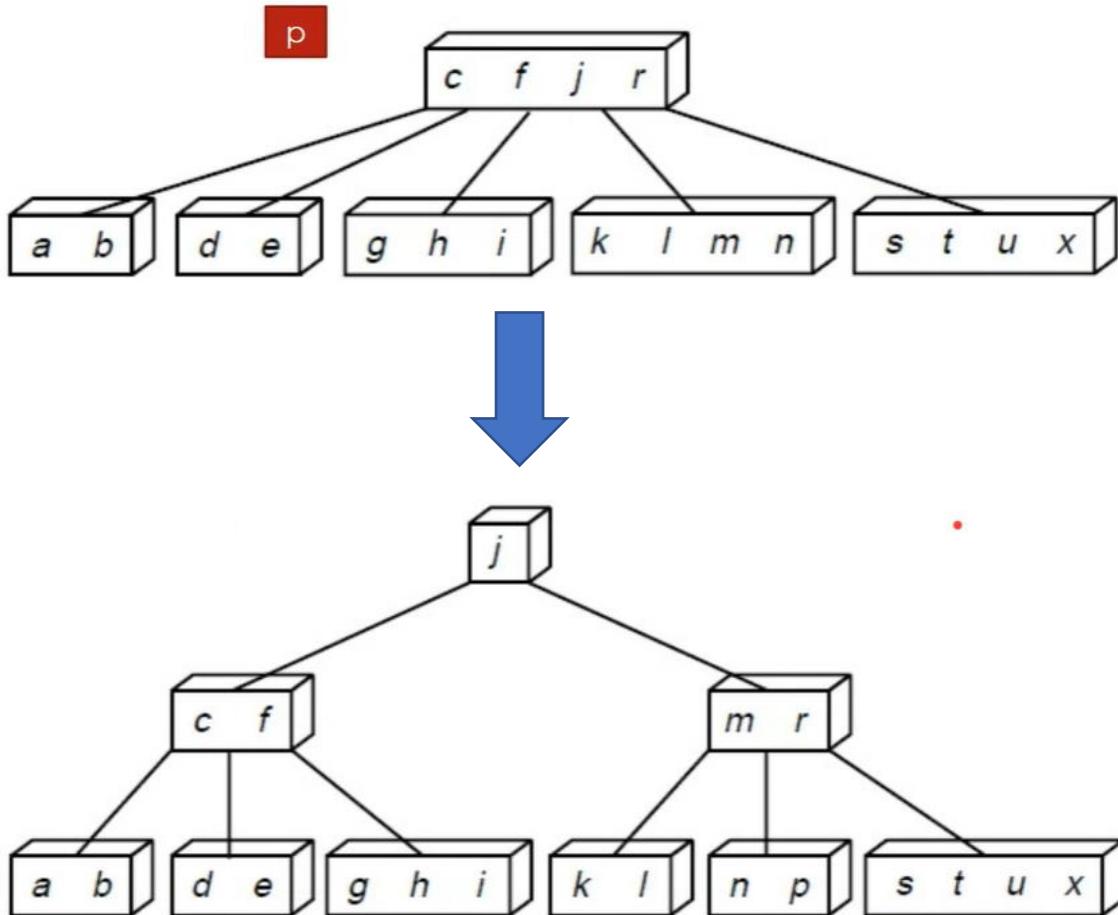
Un ejemplo: se ingresan datos en el siguiente orden

a, g, f, b, k, d, h, m, j, e, s, i, r, x, c, l, n, t, u, p



Cada hijo sería un puntero a un nodo, y cada nodo tiene 2 claves y 3 hijos

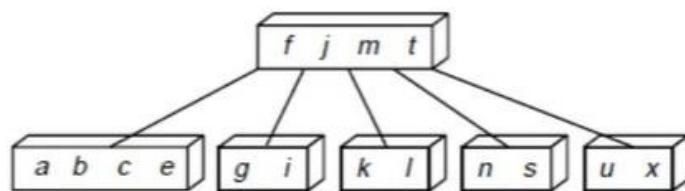
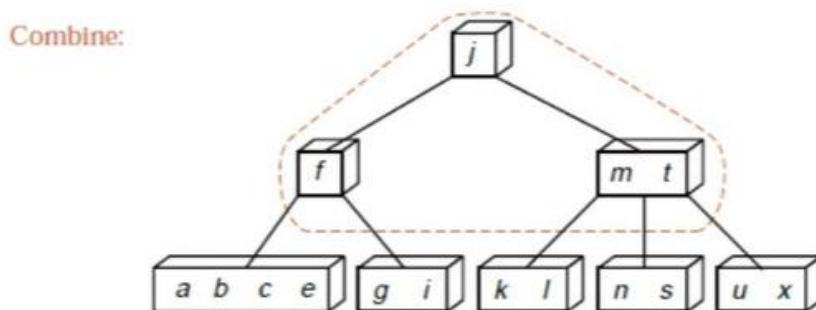
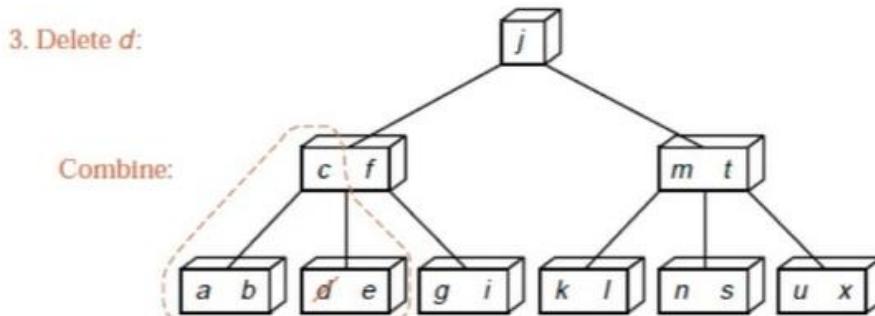
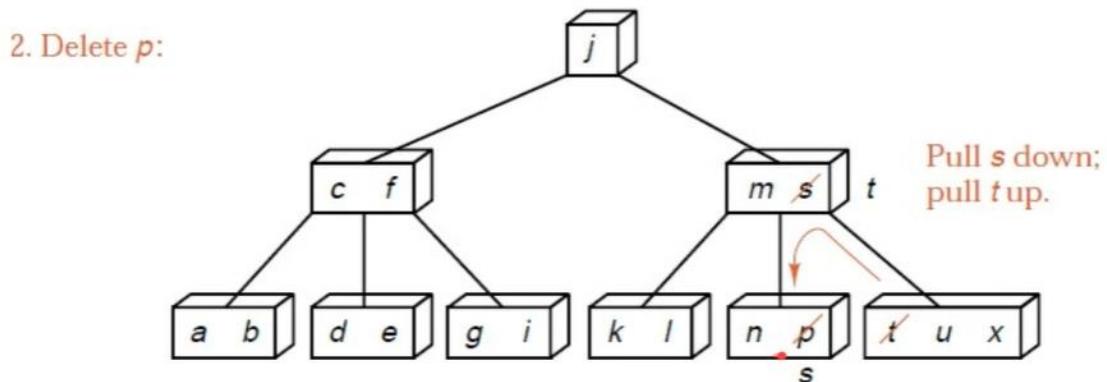
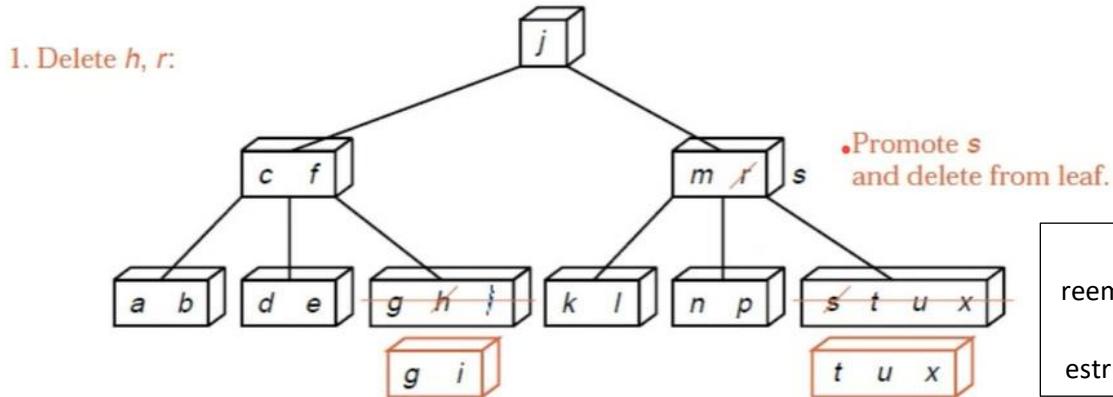




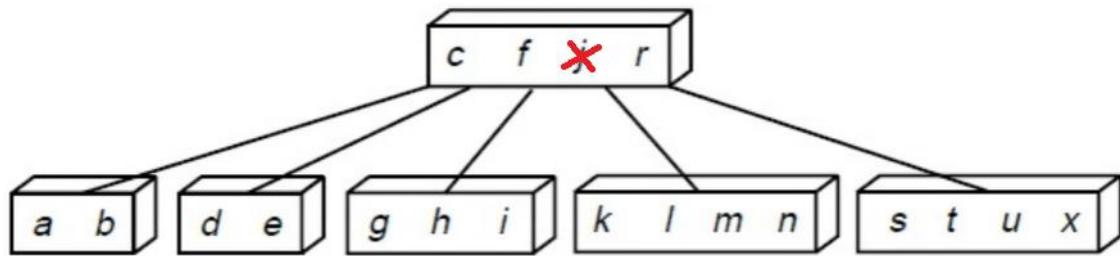
Al insertar a **p** se hace el siguiente proceso:

1. P < C? no
2. P < F? no
3. P < J? no
4. P < R? si, entonces se va por la rama izquierda de R
5. P < K? no
6. P < L? no
7. P < M? no
8. P < N? no
9. Como no hay más claves, entonces se asume que P es el mayor de todos de ese nodo. Por lo tanto se ubica a la derecha.
10. El nodo quedó con 5 claves, lo cual supera la cantidad máxima que puede tener. Por lo tanto, la clave de en medio del nodo debe subir, en este caso M (es el que queda en medio tras sumar a P) y splitando el nodo.
11. M queda justo a la izquierda de R (tras preguntar a todos si es menor que...)
12. Como el nodo queda con 5 claves (superá la cantidad maxima que puede tener), sube el de en medio (en este caso la J), acomodando todas las vías.

Un ejemplo de borrado en un árbol de 5 vías:



Caso de borrado de una clave en una no hoja:

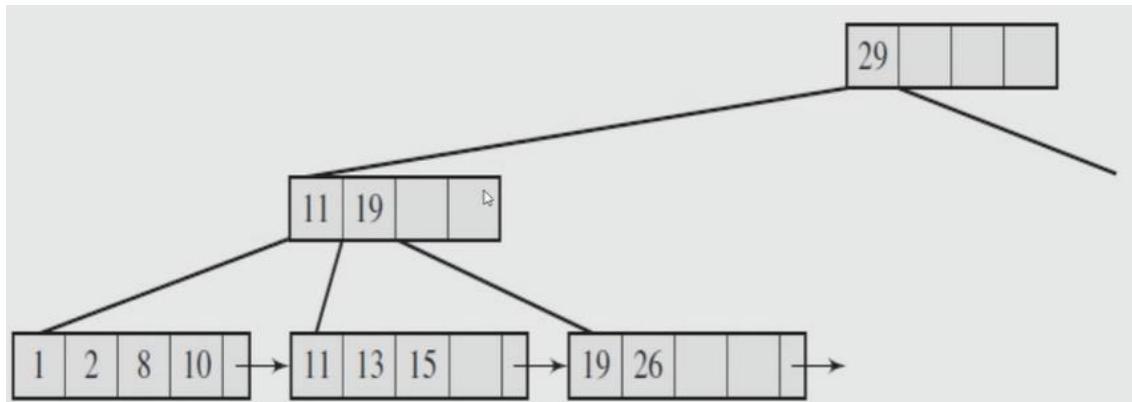


Sí o si, una clave debe ir a reemplazar a la j para que pueda mantenerse en orden y balanceado. En este caso, será la i por ser mayor que la f y menor que la r.

Árbol B+

Cumple con las características de árbol B y agrega:

- Toda la información está almacenada en los nodos hoja.
- Los nodos forman una lista simplemente enlazada.



En este caso, el 11 y el 19 están repetidos y funcionan como identificador.

Todos los hijos son mayores o iguales a la derecha.

Las hojas arman una lista.

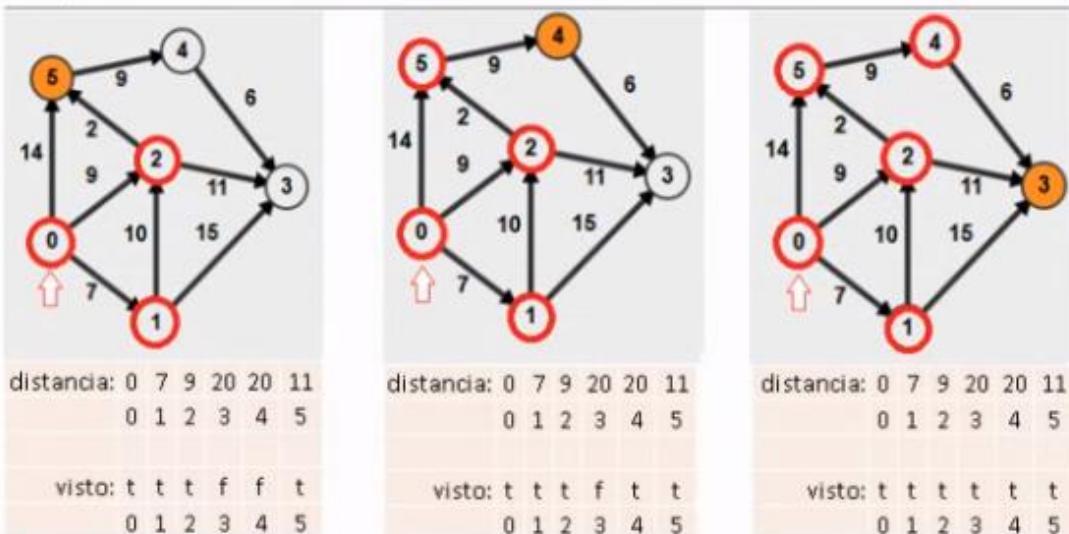
Grafos

<https://www.youtube.com/watch?v=XOvzxkZM-j0>

Camino mínimo según Dijkstra

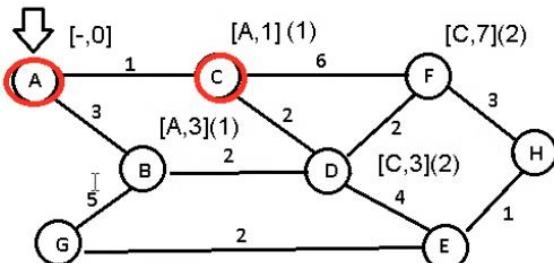
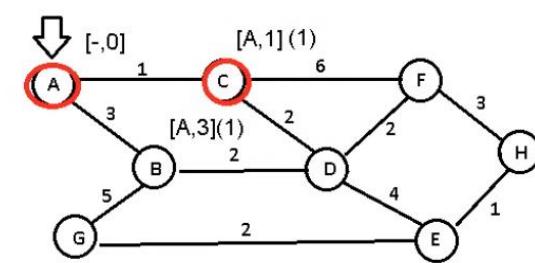
Iniciando en el nodo etiquetado como inicial o 0.

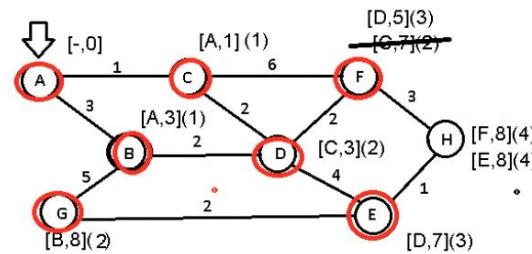
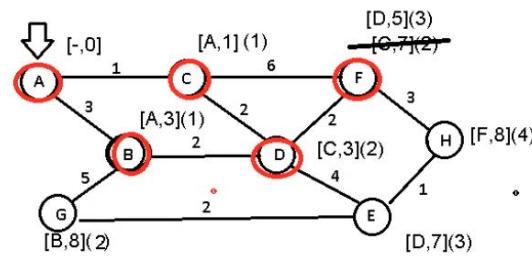
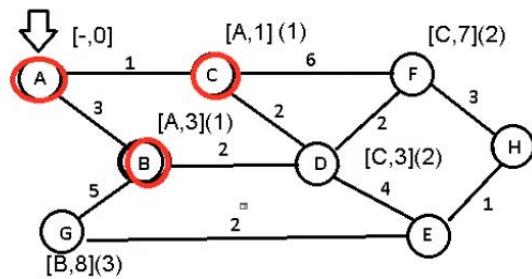
Dijkstra



- Cola de prioridad

Se etiqueta cada nodo con: [desde dónde venís , distancia acumulada] y entre paréntesis el número de iteración. Los círculos rojos son para marcar / etiquetar como visitado.





Para este caso:

C_{min1} : A – C – D – F – H

C_{min2} : A – C – D – E – H

Camino mínimo según Floyd-Warshall

Arma de matrices de recorrido para conocer el camino mínimo

- Formar las matrices iniciales D y R, donde D es la matriz de adyacencia o *distancias*, y R es una matriz de *recorridos* del mismo tamaño vacía.
- Se toma k=1.
- Se selecciona la fila y la columna k de la matriz D y entonces, para i y j, con i=k, j=k e i=j, hacemos:

Si $(D_{ik} + D_{kj}) < D_{ij} \rightarrow D_{ij} = D_{ik} + D_{kj}$ y $R_{ij} = k$

En caso contrario, dejamos las matrices como están.



- Si $k \leq n$, aumentamos k en una unidad y repetimos el paso anterior, en caso contrario páramos las interacciones.
- La matriz final D contiene los costos óptimos para ir de un vértice a otro, mientras que la matriz R contiene los penúltimos vértices de los caminos óptimos que unen dos vértices.

Un camino “infinito” de un nodo a otro se dice cuando no son adyacentes.



(los cuadros en rojo es donde hubo un remplazo por encontrar una distancia menor)

(el resultado amarillo es donde nos encontramos ubicados)

iteracion1

distancias:

	A	B	C	D	E
A	0	4	8	∞	∞
B	4	0	1	2	∞
C	8	12	0	4	2
D	∞	2	4	0	7
E	∞	∞	2	7	0

recorridos:

	A	B	C	D	E
A	-	B	C	D	E
B	A	-	C	D	E
C	A	A	-	D	E
D	A	B	C	-	E
E	A	B	C	D	-

iteracion2

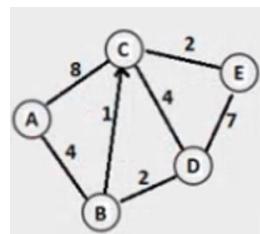
distancias:

	A	B	C	D	E
A	0	4	5	6	∞
B	4	0	1	2	∞
C	8	12	0	4	2
D	6	2	3	0	7
E	∞	∞	2	7	0

recorridos:

	A	B	C	D	E
A	-	B	B	B	E
B	A	-	C	D	E
C	A	A	-	D	E
D	B	B	B	-	E
E	A	B	C	D	-

Finalmente



iteracion4

distancias:

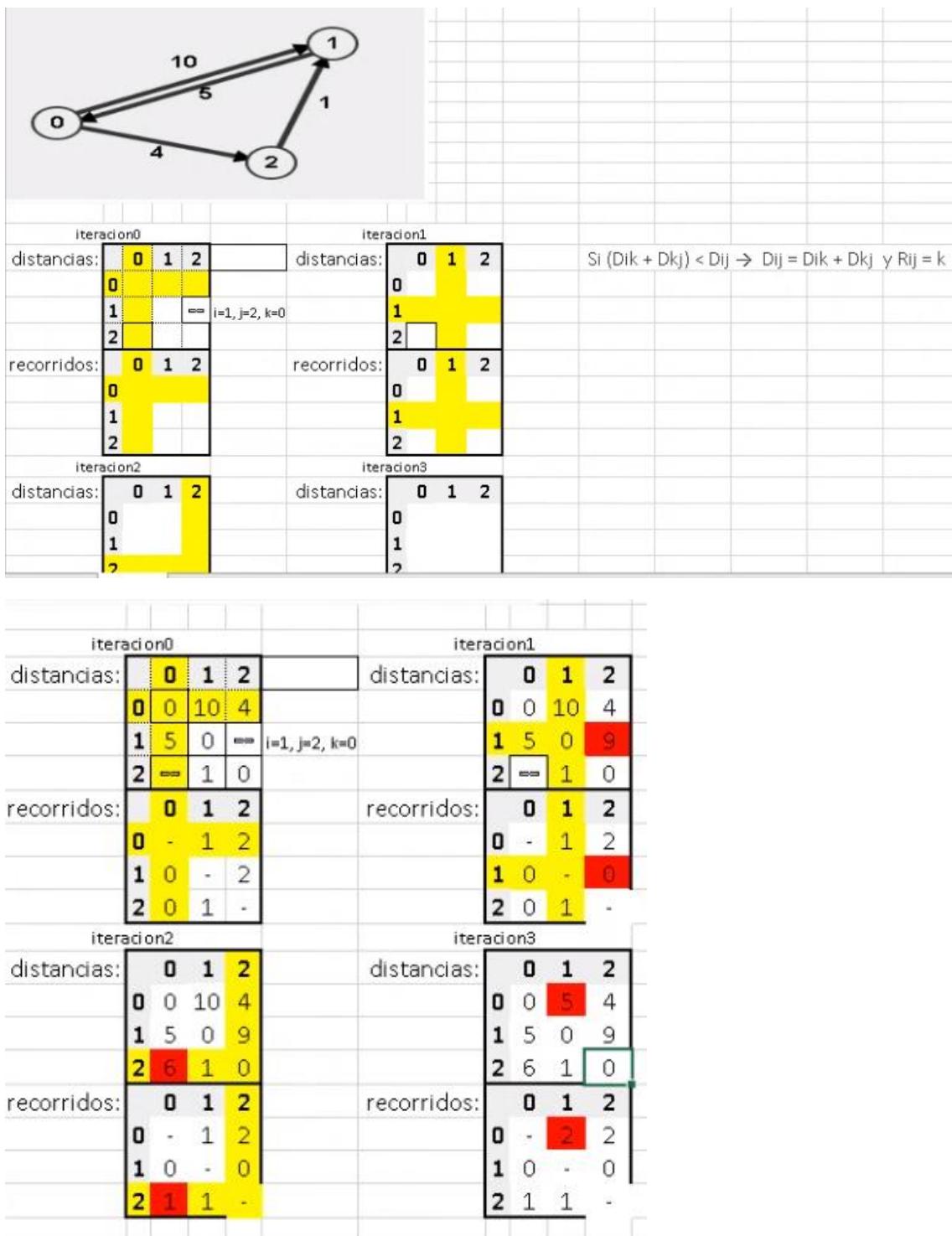
	A	B	C	D	E
A	0	4	5	6	7
B	4	0	1	2	3
C	8	6	0	4	2
D	6	2	3	0	5
E	11	8	2	6	0

recorridos:

	A	B	C	D	E
A	-	B	B	B	C
B	A	-	C	D	C
C	A	D	-	D	E
D	B	B	B	-	C
E	C	D	C	C	-

Para llegar de A a E (distancia mínima 7), primero hay que pasar por C, pero para llegar a C hay que pasar por B.

Otro ejemplo:

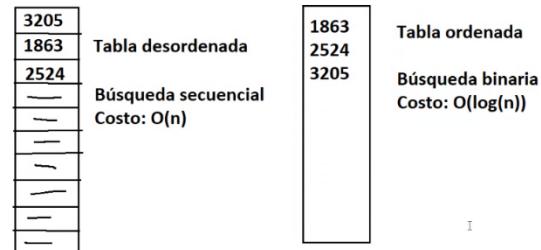


Hashing

(Tablas de dispersión)

Si se tiene una tabla de vectores, la cual está desordenada, se debe realizar una búsqueda secuencial para encontrar un vector específico, lo cual tiene una complejidad de $O(n)$.

Si se mantuviera la tabla siempre ordenada, bastaría con una búsqueda binaria, con una complejidad de $O(\log_2(n))$.



- Tamaño de la tabla (vector) de hashing
 - o Hay que saber cuántas claves (datos) hay
 - o Tomo esa cantidad y la divido por 0,8 (factor de carga λ)
 - o Elijo un número primo inmediatamente superior al valor