

**Tipos abstractos y Estructuras de datos. Organizaciones de ficheros. Algoritmos. Formatos de información y ficheros.**

### **1. TIPOS ABSTRACTOS DE DATOS (TAD)**

Se entiende por **abstracción** la capacidad de manejar un objeto como un concepto general, sin considerar la enorme cantidad de detalles que pueden estar asociados con dicho objeto.

En programación, esto significa que, de un TAD concreto, sabemos su objetivo (para lo que vale) y sabemos las operaciones que podemos realizar con él, pero no conocemos como está implementado, ni importa en realidad, solo queremos usar el TAD mediante las operaciones que permita, esto nos abstrae de la capa de implementación propia del TAD.

Se pueden **definir los TAD** como modelos matemáticos sobre los que se definen:

- Un conjunto de datos.
- Un conjunto de operaciones que se pueden realizar sobre esos datos.
- Todo esto sin especificar cómo se implementan internamente

Se centra en el comportamiento y no en la implementación. Es decir, describe qué hace una operación, no cómo lo hace.

Permiten razonar sobre el comportamiento del software sin preocuparse por los detalles técnicos. Son esenciales para el diseño de algoritmos eficientes y estructuras de datos robustas. Son la base de la programación orientada a objetos, donde las clases son una forma de TAD.

Aunque el TAD define el comportamiento, su implementación puede variar:

- Una pila puede implementarse (por ejemplo) con un array o una lista enlazada.
- Un diccionario puede ser (por ejemplo) una tabla hash o un árbol binario de búsqueda.

La abstracción permite cambiar la implementación sin alterar el código que lo usa, dicho de otro modo, puedo implementar por ejemplo el TAD Pila o bien con un array o bien con una lista enlazada, lógicamente al escribir el código (implementarlo) será distinto en un caso que, en el otro, pero el uso que se hace posteriormente del TAD es el mismo (son las mismas operaciones), esté implementado de una manera u otra.

#### **Características generales de los TAD**

- Una vez definido se puede usar como un tipo de dato primitivo
- Un TAD puede incluir otro TAD
- Las operaciones de un TAD pueden involucrar otros Tasa
- Encapsulamiento: Oculta los detalles internos de implementación.
- Interfaz definida: Solo se puede interactuar con el TAD mediante sus operaciones públicas.

- Independencia de implementación: Puedes cambiar la estructura interna sin afectar el uso externo.
- Modularidad: Facilita el diseño de software organizado y reutilizable.
- Se pueden definir en lenguaje natural, pseudocódigo o directamente usando algún lenguaje de programación.

**Ejemplos clásicos de TAD**

TAD	Operaciones típicas	Ejemplo de uso
Pila (Stack)	push, pop, top, isEmpty	Deshacer acciones en un editor
Cola (Queue)	enqueue, dequeue, front, isEmpty	Gestión de procesos en sistemas
Lista	insert, delete, search, traverse	Manejo de elementos ordenados
Conjunto	union, intersection, add, remove	Operaciones matemáticas de grupos
Diccionario	insert, delete, search (clave-valor)	Almacenamiento de configuraciones

**2. ESTRUCTURAS DE DATOS**

En primer lugar, no se debe confundir TAD con estructura de datos. El TAD representa un objeto concreto (una lista, una pila, etc....) y la estructura de datos es el mecanismo usado para implementar el TAD, y puede ser distinta según la implementación.

**Ejemplo:**

Supongamos que estás enseñando el concepto de **Pila** a tus alumnos:

Como TAD, defines que una pila permite insertar en la cima de la pila (push) y eliminar de la cima de la pila (pop), siguiendo el orden LIFO.

Como estructura de datos, puedes implementarla con:

- Un array.
- Una lista enlazada.

Podemos decir que los TAD hacen uso de las ESTRUCTURAS DE DATOS.

**2.1 TIPOS DE DATOS**

Las estructuras de datos están formadas, obviamente, por datos que tienen que ser de un tipo determinado. Existen básicamente dos tipos de datos:

- **Simples**  $\equiv$  Es aquél cuyo contenido se trata como una unidad que no puede separarse en partes más elementales. En programación se les conoce también como **tipos primitivos**. Ejemplos: entero, real, booleano, carácter, puntero

- **Estructurados** ≡ Son aquellos que permiten almacenar un conjunto de elementos bajo una estructura particular, darle un único nombre, pero con la posibilidad de acceder en forma individual a cada componente.

## 2.2 ESTRUCTURAS DE DATOS

Las **estructuras de datos** son métodos de organización de datos que permiten un almacenamiento eficiente de la información en la memoria principal.

Son un conjunto de variables de un determinado tipo agrupadas y organizadas para representar un comportamiento. Pretenden facilitar el esquema lógico para manipular los datos en función del problema y del algoritmo.

Según su comportamiento durante la ejecución del programa, existen dos tipos de estructuras:

- **Estáticas** ≡ tamaño de memoria fijo, definido en tiempo de compilación.
- **Dinámicas** ≡ tamaño de memoria variable, definido en tiempo de ejecución.

### 2.2.1 ESTRUCTURAS DE DATOS ESTÁTICAS

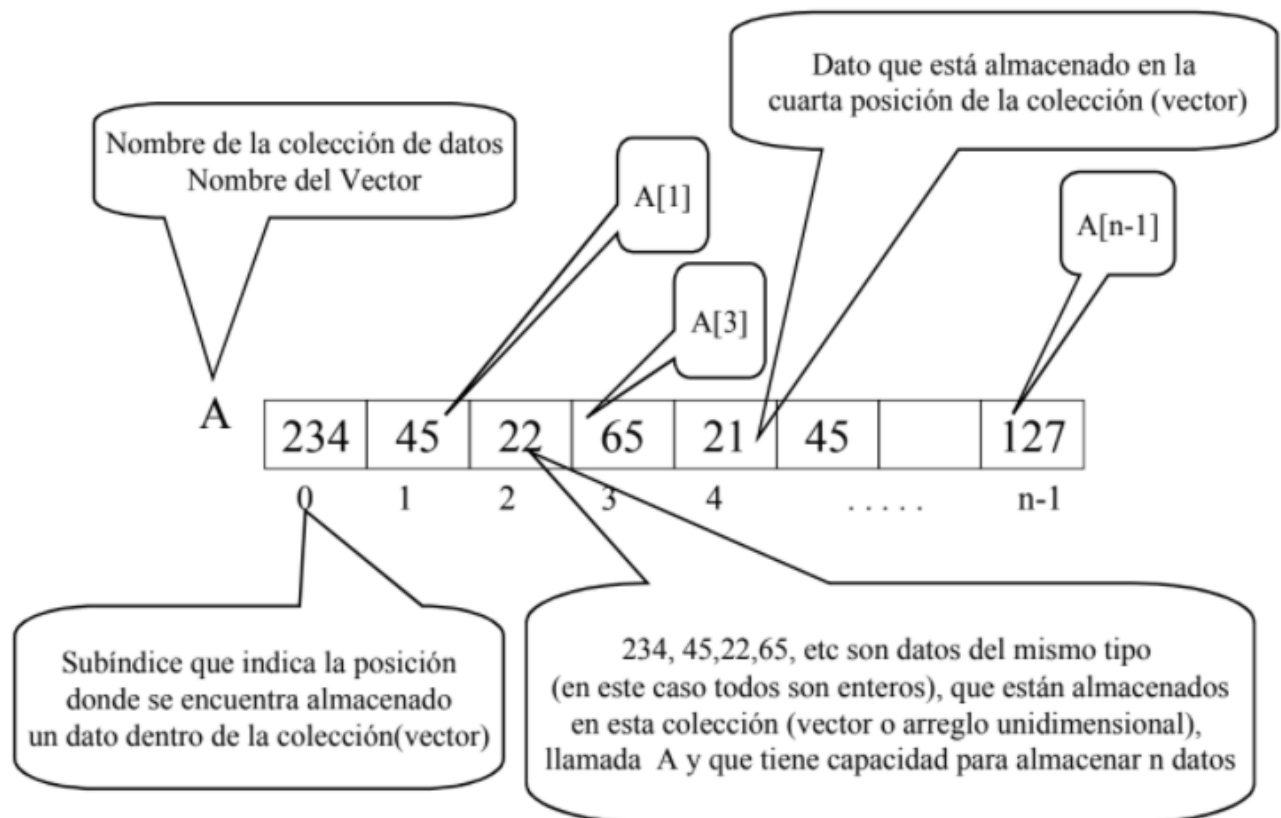
Son aquellas cuyo tamaño o longitud y estructura quedan fijados en tiempo de definición o compilación (al escribir el código fuente), y permanecen inalterables durante la ejecución del proceso en el que fueron declaradas. Al decir que pertenecen inalterables nos referimos a estructura y tamaño, no al contenido en sí, que puede variar, lógicamente.

#### Tipos de estructuras estáticas de datos

**ARRAY** ≡ También llamado vector o arreglo. Son colecciones de datos del mismo **tipo**, referenciadas **con un solo nombre**, y que para acceder a cada uno de los datos de la colección se hace uso de subíndices.

#### Características:

- N.º fijo de elementos del mismo tipo
- Almacenados en posiciones contiguas de memoria
- Acceso directo a través de un índice



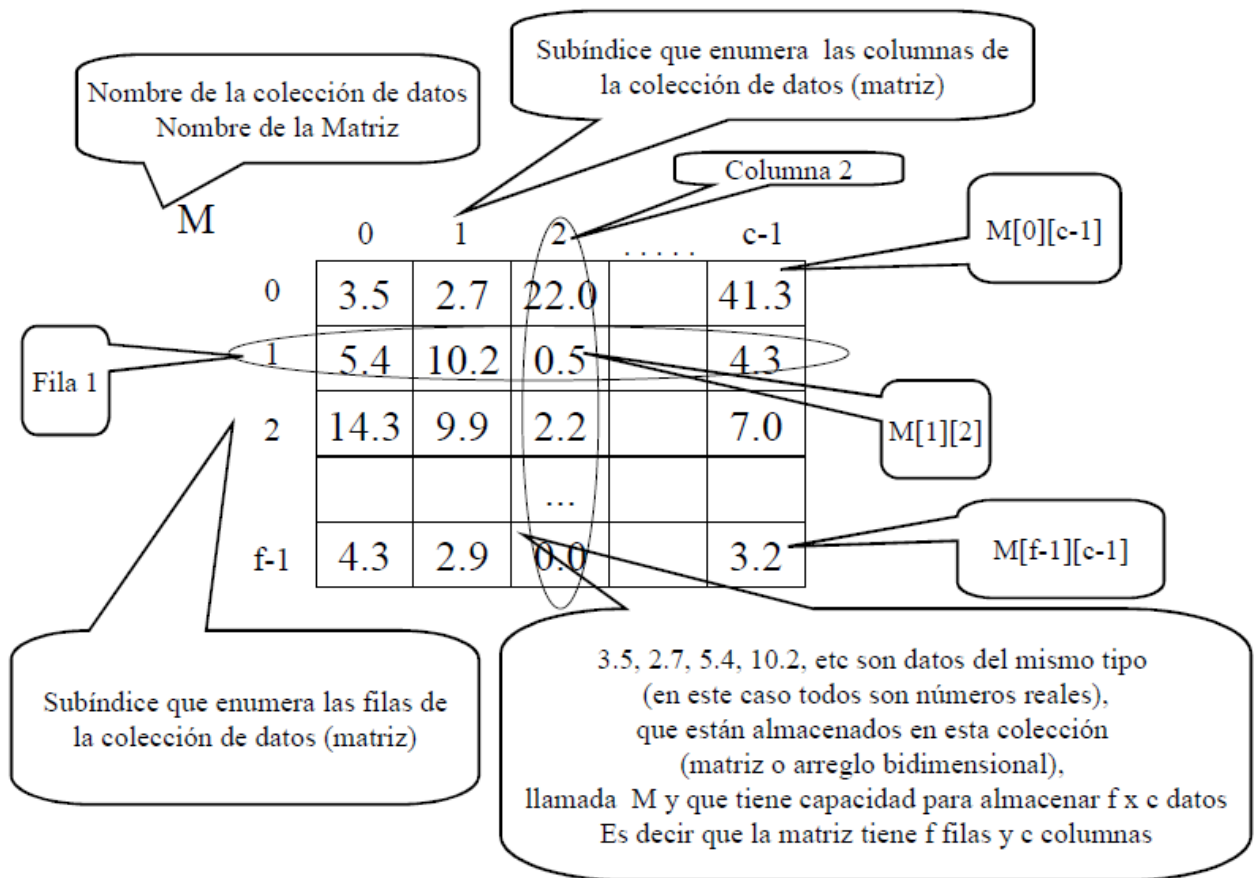
EL número de índices de un vector puede ser más de uno, a esto se le conoce como **DIMENSIÓN**. En función de la dimensión:

1 dimensión (un solo índice) → Array, vector o arreglo. (Términos sinónimos).

2 dimensiones (dos índices) → Matriz

3 o más dimensiones (tres índices o más) → Poliedro

Una Matriz o array bidimensional es una colección de datos del mismo tipo, referenciada con un solo nombre y que para acceder a cada uno de sus datos se necesita dos subíndices que indican la posición (la fila y la columna) donde se encuentra almacenado un dato, dentro de la colección.



**REGISTRO**  $\equiv$  Es una estructura de datos formada por yuxtaposición de elementos que contienen información relativa a un mismo ente.

A los elementos que componen el registro los llamamos **campos**, cada uno de los cuales es de un **determinado tipo, simple o estructurado**. Los campos dentro del registro aparecen en un orden determinado y se identifican por un nombre. Para definir el registro es necesario especificar el nombre y tipo de cada campo.

Por ejemplo: consideremos un registro, referido a Empleado, que está constituido por tres campos: Nombre (cadena), Edad (entero) y Porcentaje de impuestos (real).

### 2.2.2 ESTRUCTURAS DE DATOS DINAMICAS

Las estructuras dinámicas estarán formadas por un número **indeterminado** de datos, es decir, el número de elementos que existirá a lo largo del desarrollo del programa no se sabrá antes de la ejecución del mismo, por tanto, las estructuras dinámicas se dice que tienen **cardinalidad infinita**.

Como consecuencia de eso, es imposible asignar una cantidad de espacio definido y fijo de memoria para almacenar las variables dinámicas declaradas, surgiendo así el concepto de **asignación dinámica de memoria**.

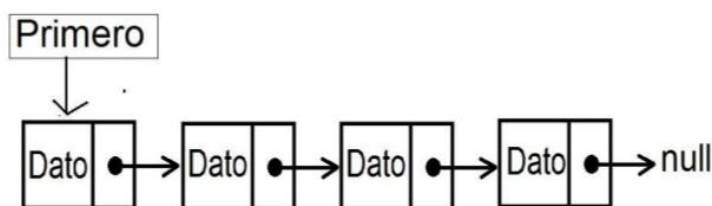
De esta forma, cada vez que se necesite un nuevo componente de la estructura, se deberá asignar de forma dinámica, es decir, el sistema operativo se encargará de asignar un hueco no ocupado de memoria para ese elemento.

Una primera impresión que se puede sacar de esto es que las distintas componentes de una estructura dinámica **no tienen por qué ocupar posiciones consecutivas** de memoria debido a que el sistema operativo puede asignar huecos situados en **posiciones arbitrarias**.

#### Concepto de Puntero

Debido a que los distintos "valores" que forman una estructura dinámica pueden estar almacenados en posiciones arbitrarias de memoria, se necesita un mecanismo que nos permita enlazar unos elementos con otros, para poder acceder a todos los datos que formen parte de una misma estructura.

Para ellos se utilizan los **punteros**, es una variable especial que almacena una dirección de memoria que corresponde a otra variable, es decir, un puntero "apunta" a otra variable dentro de la memoria. De esa manera cada componente de la estructura debe tener un puntero que almacene la dirección de memoria del elemento siguiente o el anterior, o ambos.

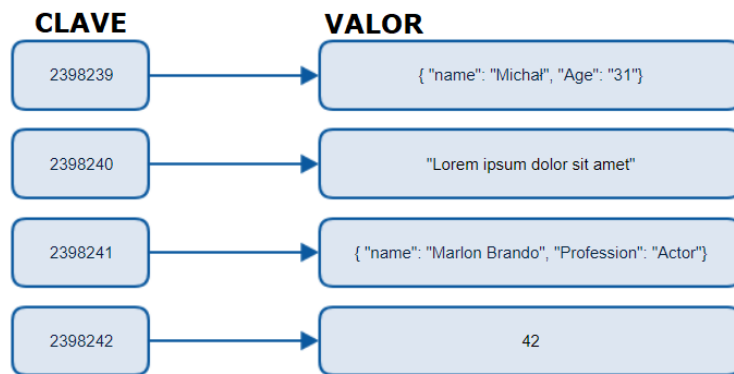


**2.2.2.1 Tipos de estructuras de datos dinámicas**

**CONJUNTO**  $\equiv$  Es una colección de valores, sin **ningún orden concreto** ni **valores repetidos**. Su correspondencia en las matemáticas sería el conjunto finito. Los valores pueden ser datos primitivos o datos estructurados (otro conjunto incluso).

**MULTICONJUNTO O BOLSA**  $\equiv$  Una variación del conjunto es el multiconjunto o bolsa, que es lo mismo que una estructura de datos de conjunto, pero que admite valores repetidos.

**MAPA**  $\equiv$  Conjunto de pares clave-valor, cuyas **claves** no tienen **ningún orden concreto**. Se pueden repetir **los valores**, pero no las claves. En algunos lenguajes de programación (PHP, por ejemplo), se les conoce también con el nombre de vectores asociativos. Un ejemplo de serían las tablas Hash.



**DICCIONARIO**  $\equiv$  Un diccionario es una estructura MAPA al que se le otorga un **orden** interno en la clave.

**LISTAS**  $\equiv$  Secuencia de **ordenada** de valores que pueden **repetirse** y en la que todos los componentes son del mismo tipo, y cada elemento puede ir seguido de otro o de ninguno.

Existen dos criterios generales de calificación de listas:

❖ Por la forma de acceder a sus elementos.

- Listas densas  $\rightarrow$  Cuando la estructura que contiene la lista es la que determina la posición del siguiente elemento. Los elementos siguen una secuencia física. Se sabe cuál es el siguiente elemento porque para acceder a él haya que pasar previamente por todos los anteriores.
- Listas enlazadas o encadenadas  $\rightarrow$  es una de las estructuras de datos fundamentales, y puede ser usada para implementar otras estructuras de datos. Consiste en una secuencia de nodos, en los que se guardan campos de datos arbitrarios y una o dos referencias, enlaces o punteros al nodo anterior o posterior.

El orden de almacenamiento puede ser diferente al orden del recorrido. Son estructuras que almacenan elementos auto referenciados, dado que contiene un puntero (referencia) a otro elemento del mismo tipo. Permiten inserciones y borrados **en cualquier parte de la lista**, pero no permiten un acceso directo o aleatorio a un nodo concreto, hay que recorrer la lista, según donde vayan apuntando los punteros, para poder llegar al elemento que nos interese.

#### 2.2.2.2 Tipos de estructuras de datos de listas densas

**PILA** ≡ Es una estructura en la que se añaden elementos, y pueden ser recuperados de forma que el último en llegar es el primero en salir (**LIFO**. - Last in, first out. Último en entrar, primero en salir). Solo se puede acceder al elemento que esté en la cima de la pila. Se puede implementar con un array o con una lista. Las operaciones usuales sobre una pila son las siguientes:

- Apilar (push) → Almacena un elemento nuevo en la cima de la pila. Salvo que la pila esté llena.
- Desapilar (pop) → Obtiene el elemento que está en la cima de la pila. Salvo que la pila esté vacía.

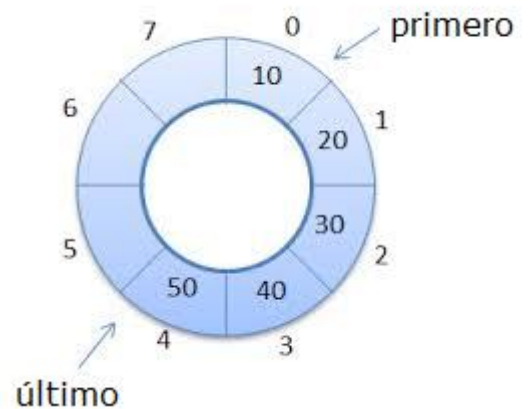
A la cima de la pila se le suele llamar TOS (Top of stack). Si la pila tiene un tamaño máximo especificado (por ejemplo, cuando se usa un array para implementarla) se le llama pila acotada. Si se implemente la pila con una lista que se vaya ampliando dinámicamente, salvo que se especifique lo contrario, no tiene un tamaño máximo (recordar concepto previamente comentado de cardinalidad infinita).

**COLA** ≡ Conocida también como FILA. Las operaciones de inserción (push) se realizan por un extremo, y las operaciones de extracción (pop) por el otro (FIFO. - First In, First Output. Primero en entrar, primero en salir). Solo se puede acceder al primer y último elemento. Las operaciones usuales son las siguientes:

- Encolar → Se añade un elemento a la cola. Se añade al final de esta.
- Desencolar → Se elimina el elemento frontal de la cola, es decir, el primer elemento que entró.
- Frente → Se devuelve el elemento frontal de la cola, es decir, el primer elemento que entró. (se devuelve, pero no se elimina!).

**Tipos de Colas**

- Cola circular o anillo → Los elementos están de forma circular y cada elemento tiene un sucesor y un predecesor. El último elemento precede al primero y viceversa.



- Cola de prioridad → Los elementos de la cola tiene, de forma opcional, una prioridad asignada, de forma que serán desencolados los elementos de mayor prioridad en primer lugar. En caso de igualdad de prioridad se sigue el orden natural de la cola.

Existen dos formas de implementar la prioridad:

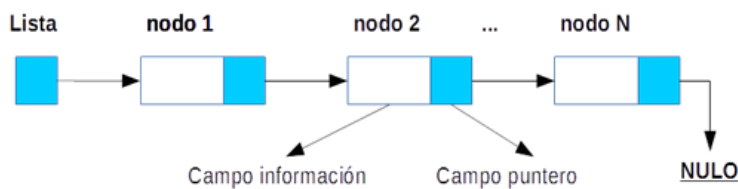
- Asignar la prioridad como un dato más del elemento. En ese caso conviene tener la cola ordenada por prioridad.
  - Hacer tantas colas distintas como prioridades haya.
- Bicolos, doble cola o colas doblemente terminadas → Tipo de cola que permiten la inserción y eliminación de elementos de ambos extremos de la cola, tanto por el principio como por el final. Pueden ser de dos tipos:
    - De Entrada restringida → Insertar por el final y Eliminar por el principio o final. En general, permite insertar por un extremo y eliminar por los dos.
    - De Salida restringida → Insertar por el principio o el final y Eliminar por el final. En general, permite insertar por ambos extremos y borrar solo por uno.

**2.2.2.3 Tipos de estructuras de datos de listas encadenadas (o enlazadas)**

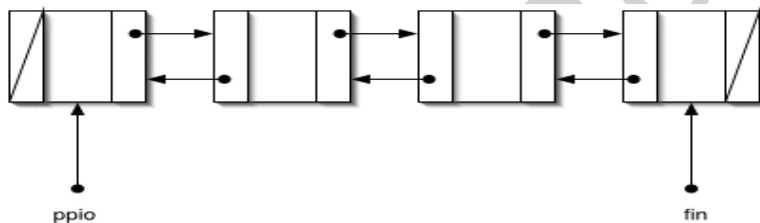
**LISTAS ENLAZADAS LINEALES**  $\equiv$  Existe un puntero al primero elemento de la lista. El último elemento de la lista no apunta a ningún otro elemento.

Se distinguen dos tipos:

- Lista simplemente enlazada  $\rightarrow$  Cada nodo tiene un único campo de enlace. Cada nodo, excepto el último, enlaza con el nodo siguiente, y el enlace del último nodo contiene NULL para indicar el final de la lista.

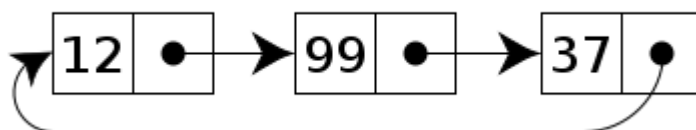
**Lista simplemente enlazada**

- Lista doblemente enlazada (o lista enlazada de dos vías)  $\rightarrow$  Cada nodo tiene dos enlaces: uno apunta al nodo anterior, o apunta al valor NULL si es el primer nodo; y otro que apunta al nodo siguiente, o apunta al valor NULL si es el último nodo.

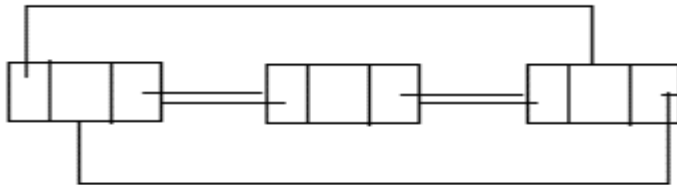


**LISTAS ENLAZADAS CIRCULARES**  $\equiv$  En estas listas el último elemento apunta al primero. Al igual que en el caso anterior, podemos distinguir:

- Lista circular simplemente enlazada  $\rightarrow$  Cada nodo tiene un enlace al nodo siguiente, pero a diferencia de las listas enlazadas lineales, el último nodo apunta al primero, no a NULL. Se suele almacenar la referencia (puntero) al último nodo de la lista, dado que a partir de él podemos acceder al primer nodo.



- Lista circular doblemente enlazada → Cada nodo tiene dos enlaces, similares a los de la lista doblemente enlazada, excepto que el enlace anterior del primer nodo apunta al último y el enlace siguiente del último nodo, apunta al primero.



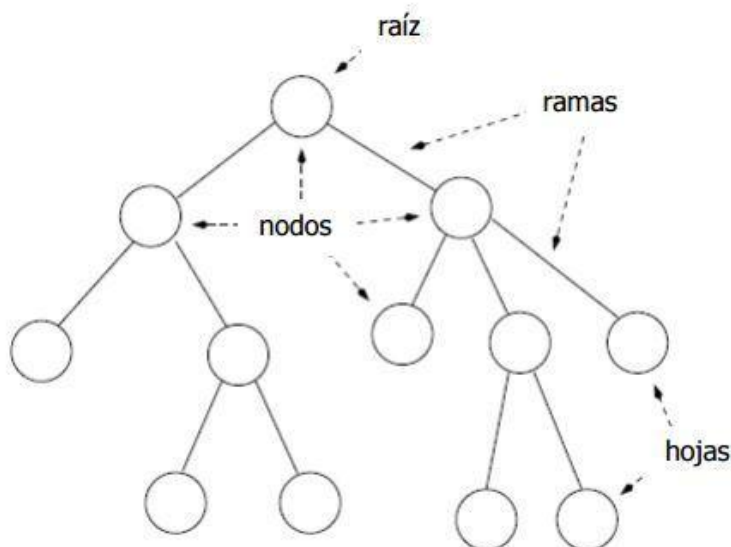
En general, a veces las listas enlazadas tienen un **nodo centinela** (también llamado falso nodo o nodo ficticio) al principio o al final de la lista, el cual no es usado para guardar datos. Su propósito es simplificar o agilizar algunas operaciones, asegurando que cualquier nodo tiene otro anterior o posterior, y que toda la lista (incluso alguna que no contenga datos) siempre tenga un "primer y último" nodo.

**ARBOLES** ≡ Es una estructura de datos muy similar a las listas doblemente enlazadas, en el sentido que tienen punteros que apuntan a otros elementos, pero no tienen una estructura lógica de tipo lineal o secuencial como aquellas, sino ramificada o jerárquica.

Un árbol que no tiene ningún nodo se llama **árbol vacío o nulo**.

Formada por nodos, vértices y aristas y es **acíclica**. Un nodo puede tener cero o más hijos, y uno o ningún padre. El **único** nodo que no tiene padre es el nodo **raíz** del árbol.

Si el orden de los subárboles importa, entonces forman una lista, y se denomina árbol ordenado (por defecto un árbol se supone que es ordenado). En caso contrario los subárboles forman un conjunto, y se denomina árbol no ordenado.



**Terminología usada en el manejo de árboles**

- **Raíz:** El nodo superior de un árbol.
- **Hijo:** Un nodo conectado directamente con otro cuando se aleja de la raíz.
- **Padre:** La noción inversa de *hijo*.
- **Hermanos:** Un conjunto de nodos con el mismo padre.
- **Descendiente:** Un nodo accesible por descenso repetido de padre a hijo.
- **Ancestro:** Un nodo accesible por ascenso repetido de hijo a padre.
- **Hoja:** Un nodo sin hijos.
- **Nodo interno:** Un nodo con al menos un hijo.
- **Grado de un nodo:** Número de subárboles de un nodo.
- **Grado de un árbol:** El máximo grado que tenga cualquiera de sus nodos.
- **Brazo o Arista:** La conexión entre un nodo y otro.
- **Camino:** Una secuencia de nodos y brazos conectados con un nodo descendiente.
- **Nivel:** El nivel de un nodo se define por  $1 +$  (el número de brazos entre el nodo y la raíz).
- **Altura de un nodo:** La altura de un nodo es el número de brazos en el camino más largo entre ese nodo y una hoja.
- **Altura de un árbol:** La altura de un árbol es la altura de su nodo raíz.
- **Anchura de un árbol:** Mayor número de nodos que hay en un nivel.
- **Profundidad:** La profundidad de un nodo es el número de brazos desde la raíz del árbol hasta un nodo.
- **Bosque:** Un bosque es un conjunto o unión disjunta de árboles.
- **Rama:** Una ruta del nodo raíz a cualquier otro nodo.

En teoría de grafos, un árbol se define como un grafo acíclico conectado, en el que cualesquiera 2 vértices, están conectados por exactamente 1 camino.

**Recorrido de un árbol**

El recorrido de un árbol se refiere al proceso de visitar de una manera sistemática, exactamente una vez, cada nodo del árbol. Estos recorridos se clasifican en función del orden en el cual son visitados los nodos.

En general, un árbol puede ser recorrido de dos formas: Por profundidad y por amplitud.

- Por profundidad  $\equiv$  Varios tipos de recorridos.
  - 1) **Preorden:** Se pasa primero por la raíz, y después (para cada nodo) se recorre primero el subárbol izquierdo y luego el subárbol derecho, de forma recursiva.
  - 2) **Postorden:** Para cada nodo primero se visita el subárbol izquierdo, luego el derecho y finalmente la raíz, de manera recursiva.
  - 3) **Inorden:** Para cada nodo, primero se visita el árbol izquierdo, luego la raíz, y luego el árbol derecho, de manera recursiva.

**BLOQUE II****ANEXO****TEMA 03**

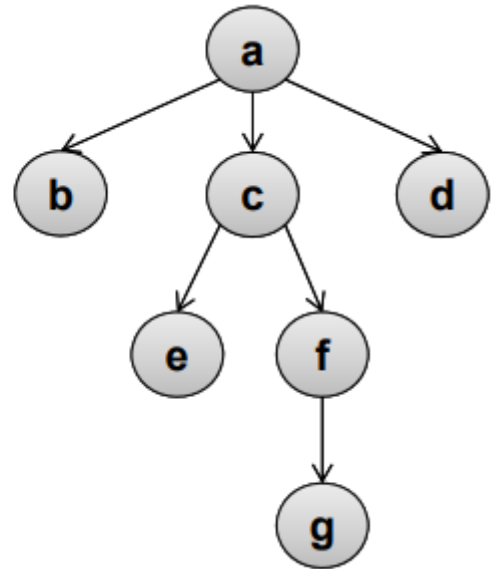
- Por amplitud, anchura o niveles  $\equiv$  Recorridos en orden por nivel (de nivel en nivel), donde visitamos cada nodo en un nivel antes de ir a un nivel inferior. Esto también es llamado recorrido en anchura-primero. En el mismo nivel los nodos se recorren de izquierda a derecha.

**Preorden:** a,b,c,e,f,g,d

**Postorden:** b,e,g,f,c,d,a

**Inorden:** b,a,e,c,g,f,d

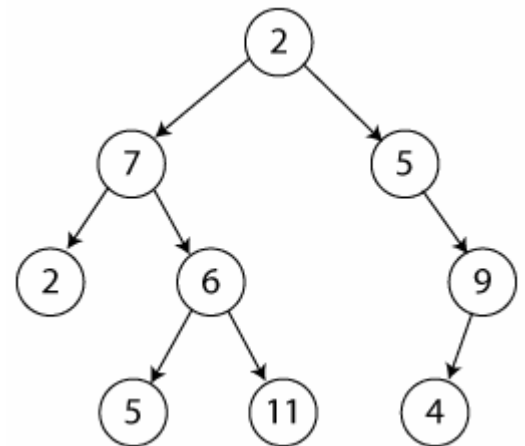
**Por Niveles:** a,b,c,d,e,f,g

**Tipos de árboles**

**ARBOL BINARIO**  $\equiv$  un árbol binario es una estructura de datos en la cual cada nodo siempre tiene un hijo izquierdo y un hijo derecho. No pueden tener más de dos hijos (de ahí el nombre "binario").

En teoría de grafos, se usa la siguiente definición: «Un árbol binario es un grafo conexo, acíclico y no dirigido tal que el grado de cada vértice no es mayor a 2». De esta forma sólo existe un camino entre un par de nodos.

- Por definición, el grado de un árbol binario **como máximo** es 2, es decir, cada nodo tiene como máximo dos hijos.
- Árbol binario **lleno**  $\equiv$  También llamado árbol binario estricto o propio.
  - Cada nodo tiene exactamente 0 o 2 hijos. No hay nodos con solo un hijo.
  - Los nodos que no son hojas (es decir, que tienen hijos) deben tener sus dos hijos.



- Árbol binario **completo**  $\equiv$  Todos los niveles están completamente llenos, excepto el último. El último nivel se llena **de izquierda a derecha**, sin huecos intermedios. Un árbol completo puede tener nodos con solo un hijo en el último nivel.



Característica	Árbol binario lleno	Árbol binario completo
Hijos por nodo	0 o 2	0, 1 o 2
Niveles inferiores	No necesariamente llenos	Todos llenos excepto el último
Posición de nodos	No importa	Último nivel alineado a la izquierda
Uso común	Árboles balanceados, heaps	Implementación de heaps, arrays

- Un árbol binario **perfecto**  $\equiv$  es un árbol binario lleno en el que todas las hojas (vértices con cero hijos) están a la misma profundidad (distancia desde la raíz, también llamada altura).



- Todos los nodos internos tienen exactamente dos hijos
  - Todas las hojas están en el mismo nivel.
- Un árbol binario es **equilibrado** si la altura entre el subárbol izquierdo y el derecho se diferencian como máximo en una unidad. Esta propiedad garantiza que el árbol no se incline demasiado hacia un lado, lo que permite mantener operaciones eficientes como búsqueda, inserción y eliminación en tiempo logarítmico (notación asintótica).

**ARBOL BINARIO DE BUSQUEDA (ABB)**  $\equiv$  Es caso especial de árbol binario que cumple que el subárbol izquierdo de cualquier nodo (si no está vacío) contiene valores menores que el que contiene dicho nodo, y el subárbol derecho (si no está vacío) contiene valores mayores.

El interés de los árboles binarios de búsqueda (ABB) radica en que su recorrido en orden proporciona los elementos ordenados de forma ascendente y en que la búsqueda de algún elemento suele ser muy eficiente.

#### Tipos de árboles binarios equilibrados

**ARBOL AVL**  $\equiv$  es un tipo de árbol binario de búsqueda auto equilibrado, propuesto en 1962 por Adelson-Velskii y Landis (de ahí el nombre AVL).

Su regla principal es:



La diferencia de altura entre los subárboles izquierdo y derecho de cualquier nodo debe ser -1, 0 o +1

Esto garantiza que el árbol se mantenga equilibrado, lo que permite mantener operaciones en tiempo logarítmico  $O(\log n)$ .

El **factor de equilibrio** es la diferencia entre las alturas del árbol derecho y el izquierdo.

#### **Ventajas**

- Mantiene búsqueda eficiente incluso con inserciones desordenadas.
- Evita que el árbol se degrade a una lista enlazada.
- Ideal para aplicaciones donde se realizan muchas operaciones dinámicas.

**ARBOL ROJO-NEGRO**  $\equiv$  Es un árbol binario de búsqueda auto equilibrado en el que cada nodo tiene un atributo de color: rojo  o negro . Este color no representa datos, sino que se usa para mantener el equilibrio del árbol de forma eficiente (eficiencia logarítmica).

En los árboles rojo-negro las hojas no son relevantes y no contienen datos.

Además de los requisitos impuestos a los árboles binarios de búsqueda convencionales, se deben satisfacer las siguientes reglas para tener un árbol rojo-negro válido:

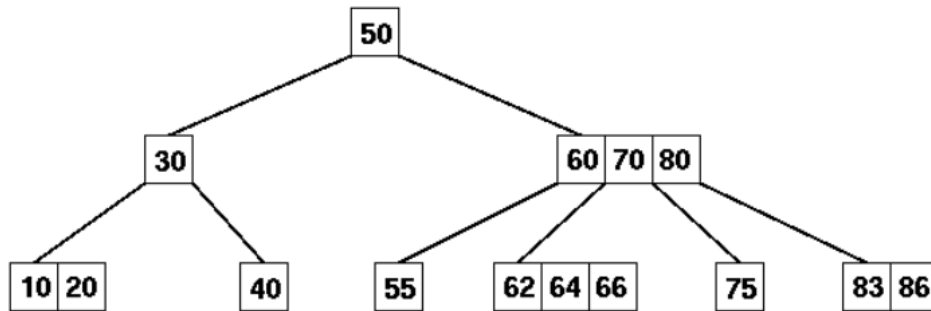
- Cada nodo es **rojo** o **negro**.
- La raíz siempre es **negra**.
- Todas las hojas (NULL) son **negras**.
- Todo nodo rojo debe tener dos nodos hijos negros, dicho de otro modo, un nodo rojo no puede tener hijos rojos (no hay dos rojos consecutivos).
- Cada camino desde un nodo dado a sus hojas descendientes contiene el mismo número de nodos negros.

**ARBOL 2-3-4**  $\equiv$  Es una estructura de datos auto equilibrada que pertenece a la familia de los árboles B, concretamente de orden 4. Es muy útil para implementar diccionarios, bases de datos y sistemas de archivos, y tiene una relación directa con los árboles rojo-negro.

Es un árbol en el que cada nodo puede tener:

Tipo de nodo	Elementos de datos	Hijos posibles
2-nodo	1	2
3-nodo	2	3
4-nodo	3	4

- Auto equilibrado: todos los nodos hoja están al mismo nivel.
- Tienen hasta 4 hijos por nodo.
- No hay nodos con un solo hijo.
- Tienen hasta 3 datos por nodo.
- Ordenado: los elementos dentro de cada nodo están en orden creciente.
- Operaciones eficientes: búsqueda, inserción y eliminación en  $O(\log n)$ .
- Las reestructuraciones se realizan desde la raíz hacia las hojas.

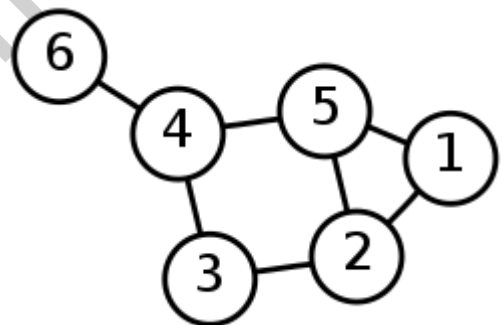


Otros tipos de árboles binarios conocidos son:

- Árboles AA  $\equiv$  AAB auto-balanceable. Variación del árbol rojo-negro.
- Árboles de segmento  $\equiv$  Permite almacenar segmentos (intervalos) en cada nodo.
- Árboles multicamino o multirrama  $\equiv$  Posee un grado "g" mayor a dos, donde cada nodo de información del árbol tiene un máximo de "g" hijos.
- Árboles B  $\equiv$  Son ABB, pero cada nodo puede contener más de dos hijos.

**GRAFOS**  $\equiv$  Es un conjunto de objetos llamados vértices o nodos unidos por enlaces llamados aristas o arcos, que permiten representar relaciones binarias entre elementos de un conjunto.

Se puede definir de forma más sencilla como una estructura de datos no lineal en la que un nodo puede apuntar a varios y a su vez puede ser apuntado por varios.



#### **Terminología usada en los grafos**

Un grafo se compone de un conjunto "V" de **vértices** y de un conjunto "A" de **aristas**. Cada arista se identifica con el par de vértices que une. Los vértices de una arista son entre sí nodos adyacentes.

**Grado de un vértice**  $\equiv$  Número de aristas que inciden en él. Si el grado es 0 se dice que es un nodo aislado.

**Grado de un grafo**  $\equiv$  Número de vértices de ese grafo. También llamado **orden**

**Camino**  $\equiv$  Secuencia de vértices conectados por aristas. La **longitud** o distancia es el número de aristas que comprende el camino.

**Tipos de Grafos**

**Grafo no dirigido**  $\equiv$  Las aristas no tienen dirección. Si hay una arista entre A y B, puedes ir de A hasta B y también de B hasta A, es decir, se puede recorrer libremente en ambos sentidos. Se representa con líneas simples entre nodos.

**Grafo Dirigido**  $\equiv$  Las aristas tienen dirección, como flechas. Una arista de  $A \rightarrow B$  no implica que puedas ir de  $B \rightarrow A$ . Se representa con flechas entre nodos

**Grafo Conectado o conexo**  $\equiv$  Existe un camino simple entre 2 cualesquiera de sus nodos, es decir, existencia de caminos entre todos los pares de vértices.

**Grafo ponderado**  $\equiv$  Es un tipo de grafo en el que cada arista tiene un valor o peso asociado. Este peso puede representar cosas como Distancia, coste, tiempo, etc.

**Algoritmos importantes en la teoría de grafos**

**Búsqueda en anchura**  $\equiv$  BFS. Algoritmo de búsqueda no informada para recorrer o buscar elementos en un grafo.

**Búsqueda en profundidad**  $\equiv$  DFS. Algoritmo de búsqueda no informada para recorrer **todos** los nodos de un árbol o grafo de forma ordenada, pero no uniforme.

**Algoritmo Dijkstra**  $\equiv$  También llamado algoritmo de caminos mínimos. Determina el camino más corto, dado un vértice origen, hacia el resto de los vértices en un grafo que tiene pesos en cada arista. Intenta encontrar el camino de menor peso.

**Algoritmo Bellman-Ford**  $\equiv$  Genera el camino más corto en un grafo dirigido ponderado, en el que el peso de alguna de las aristas puede ser negativo. El algoritmo de Dijkstra resuelve este mismo problema en un tiempo menor, pero requiere que los pesos de las aristas no sean negativos, salvo que el grafo sea dirigido y sin ciclos. Por lo que el Algoritmo Bellman-Ford normalmente se utiliza cuando hay aristas con peso negativo.

**Algoritmo PRIM**  $\equiv$  Permite encontrar un árbol recubridor mínimo en un grafo conexo, no dirigido y cuyas aristas están etiquetadas.

**Algoritmo KRUSKAL**  $\equiv$  Encontrar un árbol recubridor mínimo en un grafo conexo y ponderado. Es decir, busca un subconjunto de aristas que, formando un árbol, incluyen todos los vértices y donde el valor de la suma de todas las aristas del árbol es el mínimo.

**Algoritmo de Ford-Fulkerson**  $\equiv$  Propone buscar caminos en los que se pueda aumentar el flujo, hasta que se alcance el flujo máximo.

**Algoritmo de Floyd-Warshall**  $\equiv$  Es un algoritmo de análisis sobre grafos para encontrar el camino mínimo en grafos dirigidos ponderados. El algoritmo encuentra el camino entre todos los pares de vértices en una única ejecución. (Ej: Programación dinámica).

**Algoritmos de Tarjan y Kosaraju**  $\equiv$  Detección de componentes fuertemente conexas en grafos dirigidos.

### **3. ORGANIZACIÓN DE FICHEROS**

Un **fichero o archivo** (términos sinónimos) es un conjunto de datos relacionados lógicamente y organizados según ciertos criterios, que se almacena en memoria secundaria (discos, cintas, memorias flash, etc.).

EL **formato de un archivo** define la organización lógica del mismo, la estructura de los datos, posición que ocupan dentro del archivo, longitud, etc.

#### **3.1 REGISTROS**

Los ficheros se organizan de diversas formas, pueden contener desde una simple secuencia de bytes hasta complejas estructuras de datos.

Una forma muy habitual es dividir un fichero en **registros**. Los registros pueden ser:

**De longitud fija**  $\equiv$  Caracterizado por que siempre va a ocupar el mismo espacio en el disco, tenga o no información el registro.

**De longitud indefinida**  $\equiv$  Un registro lógico formado por varios campos de tamaño variables. En los ficheros de este tipo el ordenador desconoce el tamaño (indefinido) de sus registros, y por tanto no puede acceder a la información (registro) directamente, debido a que al no saber el tamaño tampoco puede calcular la posición. El sistema de acceso a este tipo de registros es recorriendo secuencialmente los que le preceden.

**De longitud variable**  $\equiv$  Pueden contener cualquier tamaño en bytes, se puede especificar previamente un máximo y un mínimo. El tamaño del registro oscila entre el máximo y el mínimo.

Se usan métodos para poder predefinir la longitud de los registros con el fin de poder acceder a ellos de forma correcta y sin posibles errores.

- *Separadores de campos (banderas)*: Se sitúa al inicio y final del campo un carácter especial y único que identifique el principio y el final del campo. Este carácter especial no se puede dar dentro del propio contenido de los campos. El carácter elegido será el usado siempre para esa función en todo el registro y fichero.
- *Indicadores de longitud*: Se sitúa al inicio y final del campo un campo auxiliar que almacena el tamaño de cada campo, con el fin de identificar su tamaño y por tanto su dimensión.
- *Mascaras*: La ausencia o presencia de campos se indica en el primer campo del registro, utilizando subcampos conteniendo cero o uno según exista o no, cada uno de los campos del registro.

**Definiciones importantes**

**Registro lógico**  $\equiv$  Es un conjunto de datos relacionados, referidos a la misma entidad, al que se accede y se trata de forma unitaria. Un registro está formado por uno más campos.

**Campo**  $\equiv$  Unidad elemental de información contenida en un registro, o sea, cada dato individual de un registro, que tienen significado por sí mismo en el contexto del registro del que forma parte.

**Clave**  $\equiv$  En muchas ocasiones uno de los campos que forman un registro contiene un valor que nunca se repite, denominado clave, y que permite distinguir un registro de otro. Pueden existir ficheros que no tengan registros con campos clave, en ese caso una combinación de varios campos del registro puede simular este mismo propósito.

**Registro físico o Bloque**  $\equiv$  Es la cantidad de información que se trasvasa, en una única operación de Entrada/Salida, entre memoria secundaria (disco, etc.) a memoria (RAM) o viceversa.

Un registro físico contiene varios registros lógicos, a este número se le conoce como **Factor de bloqueo**. Puede ocurrir lo contrario, que un registro lógico sea tan grande que necesite más de un registro físico para poder transportar todo el registro a la memoria central, y por lo tanto más de una operación de entrada/salida. A esto se le conoce como **registros expandidos**.

Por lo tanto, el **factor de bloqueo** puede ser:

- Menor que 1  $\rightarrow$  El registro lógico ocupa más que el físico, se transfiere menos de 1 registro lógico en cada operación de E/S.
- Igual a 1  $\rightarrow$  El tamaño del registro lógico y físico coincide, se transfiere 1 registro lógico en cada operación de entrada y salida.
- Mayor que 1  $\rightarrow$  Lo más común, varios registros lógicos en un registro físico, se transfiere más de un registro lógico en cada operación de E/S.

**Medidas de utilización de los archivos**

**ACTIVIDAD**  $\equiv$  Es el porcentaje de registros procesados en relación con el número total de registros.

Tasa de actividad =  $n^{\circ}$  reg. procesados /  $n^{\circ}$  reg. totales \* 100

**VOLATILIDAD**  $\equiv$  Es el porcentaje de registros que se adicionan, suprimen, o modifican en un fichero respecto al número medio de registros del fichero (en un periodo de tiempo concreto).

Se dice que un fichero es:

- Volátil: % de adiciones y supresiones alto.
- Estático: % de adiciones y supresiones bajo.

Depende de varias tasas:

- a) Tasa de adición =  $\text{n}^\circ \text{ reg. añadidos} / \text{n}^\circ \text{ reg. totales} * 100$
- b) Tasa de Supresión =  $\text{n}^\circ \text{ reg. eliminados} / \text{n}^\circ \text{ reg. totales} * 100$
- c) Tasa de modificación =  $\text{n}^\circ \text{ reg. modificados} / \text{n}^\circ \text{ reg. totales} * 100$
- d) Tasa de crecimiento = Tasa de adición— Tasa de supresión

### 3.2 CLASIFICACION DE FICHEROS

Según la función que cumple cada archivo, se pueden clasificar de la siguiente manera:

**FICHEROS PERMANENTES**  $\equiv$  Contienen información relevante para una aplicación. Es decir, los datos necesarios para el funcionamiento de ésta. Tienen un periodo de permanencia en el sistema amplio, la información o varía muy poco o no varía a lo largo del tiempo. Se subdividen en:

- *Ficheros maestros o de situación*: contienen el estado actual de los datos que pueden modificarse desde la aplicación.
- *Ficheros de consulta o constantes*: son aquellos que incluyen datos fijos para la aplicación. No suelen ser modificados y se accede a ellos para realización de consultas. (Por ejemplo, un archivo de códigos postales).
- *Ficheros históricos*: contienen datos que fueron considerados como actuales en un periodo o situación anterior. Se utilizan para la reconstrucción de situaciones o consulta de estadísticas.

**FICHEROS TEMPORALES**  $\equiv$  Se utilizan para almacenar información útil para una parte de la aplicación, no para toda ella. Son generados a partir de datos de ficheros permanentes. Tienen un corto periodo de existencia. Estos se subdividen en:

- *Ficheros intermedios*: almacenan resultados de una aplicación que serán utilizados por otra.
- *Ficheros de maniobras*: almacenan datos de una aplicación que no pueden ser mantenidos en memoria principal por falta de espacio.
- *Ficheros de resultados*: almacenan datos que van a ser transferidos a un dispositivo de salida.
- *Fichero de registro o transacciones*: se utiliza para almacenar temporalmente nuevos registros que deben añadirse a un archivo maestro. Estos registros se acumulan en un archivo separado, y posteriormente se integran en el archivo principal mediante un proceso de actualización o consolidación. Se organiza como una **Pila** (archivo de pila), muy habitual en transacciones de muchos registros o muchas operaciones.

**3.3 ORGANIZACIÓN DE FICHEROS****3.3.1 SOPORTES DE INFORMACIÓN**

Antes de ver las distintas formas en las que se organiza un fichero, es necesario saber en qué tipo de soportes se almacenarán estos archivos, dado que, según el tipo, el archivo podrá estar organizado de una u otra forma.

Los **soportes de información** son los dispositivos que almacenan los datos, existen dos tipos de soportes.

**Soportes secuenciales o de acceso secuencial**  $\equiv$  Cuando se abre un fichero en este tipo de soportes, se accede inicialmente al primer registro para luego ir pasando de forma secuencial por todos los demás de forma consecutiva, hasta alcanzar el registro buscado.

Se usan principalmente para copias de seguridad, y también por razones de antigüedad. Ejemplo: cintas magnéticas.

**Soportes direccionales o de acceso directo**  $\equiv$  Este soporte permite acceder de forma directa a cualquier registro sin necesidad de pasar por los anteriores. Ejemplo: Disco Duro.

**3.3.2 ORGANIZACIÓN**

**ORGANIZACIÓN SECUENCIAL**  $\equiv$  Un fichero con organización secuencial se caracteriza porque sus registros están almacenados de forma contigua, en el mismo orden en el que se ingresan, de manera, que la única forma de acceder a él, es leyendo un registro tras otro desde el principio hasta el final.

En los ficheros secuenciales suele haber una marca indicativa del fin del fichero, que suele denominarse EOF (End of File). Para detectar el final del fichero sólo es necesario encontrar la marca EOF.

Registro 1
Registro 2
Registro i-1
Registro i-2
Registro n-1
Registro n

**Características**

- La lectura siempre se realiza hacia delante.
- El acceso al fichero es monousuario, no permiten el acceso simultáneo de varios usuarios.
- Se pueden grabar en cualquier tipo de soporte, tanto en secuenciales como direccionales.

**Ventajas**

- Muy eficiente para procesamiento en bloque o lectura total.
- Aprovechan al máximo el soporte de almacenamiento, al no dejar huecos vacíos.

**Inconvenientes**

- No se pueden insertar registros entre los que ya están grabados.
- Lento para búsquedas individuales.

Los registros almacenados se identifican por medio de una información ubicada en uno de sus campos, a este campo se le denomina clave o llave. Si se ordena un archivo secuencial por su clave, es más rápido realizar cualquier operación de lectura o escritura.

**ORGANIZACION INDEXADA**  $\equiv$  Se basa en la utilización de un **índice separado**, que contiene claves y punteros a los registros contenido en el archivo de datos, de manera que permiten el acceso a un registro individual del fichero de forma directa, sin tener que leer los anteriores.

Se busca **primero en el índice**, luego se accede al registro. Por tanto, existirá una zona de registros en la que se encuentran los datos del archivo y una zona de índices, que contiene una tabla con las claves de los registros y las posiciones donde se encuentran los mismos, dentro del archivo de datos. La tabla de índices estará ordenada por el campo clave.

La tabla de índices será cargada en memoria principal (memoria RAM, búsquedas mucho más rápidas), para realizar en ella la búsqueda de la fila correspondiente a la clave del registro a encontrar, obteniéndose así la dirección donde se encuentra el registro, dentro del archivo de datos.

Una vez localizada la dirección, sólo hay que acceder a la zona de registros en el soporte de almacenamiento y posicionarnos en la dirección indicada.

Este tipo de organización divide el espacio del soporte de almacenamiento en 3 áreas o zonas:

- 1) Área primaria o de datos: Es la zona donde está el contenido ordenado ascendentemente por el valor de su clave.
- 2) Área de índices: Contiene claves y punteros que permiten localizar bloques o registros en el área primaria.
- 3) Área overflow o desborde: Almacena registros que no caben en el área primaria o que se insertan después de su creación.

El área primaria y el índice no se alteran después de ser creado el fichero, el overflow si, este va aumentando con todos los registros que son insertados.

El sistema accede primero al área primaria, y si no encuentra el registro, consulta el área de overflow.

### Ventajas

- Permite acceso directo y secuencial.
- Facilita búsquedas por múltiples campos si hay varios índices.

**Desventajas**

- Mayor complejidad y uso de espacio por mantener el índice.
- Solamente se puede grabar en un soporte direccionable.

**ORGANIZACION DIRECTA o ALEATORIA**  $\equiv$  En este tipo de ficheros se puede acceder a un registro indicando la posición relativa del mismo dentro del archivo o, más comúnmente, a través de una clave que forma parte del registro como un campo más.

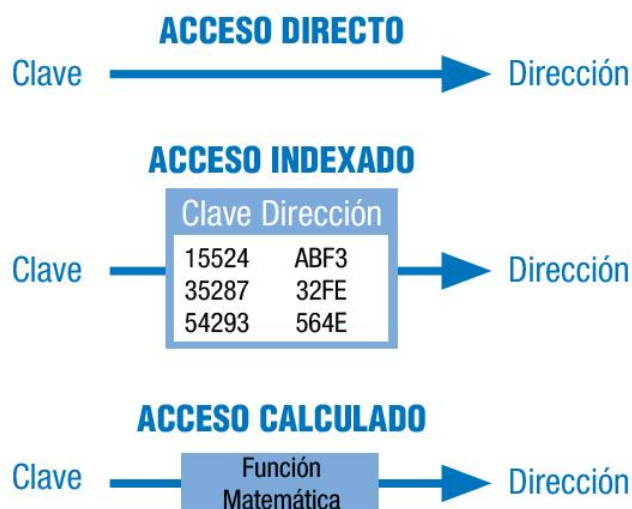
Se accede directamente al registro sin tener que recorrer el archivo.

A La organización directa también se le conoce como **aleatoria o relativa**.

Campo clave: campo que permite identificar y localizar un registro de manera ágil y organizada.

Cada uno de los registros se guarda en una posición física, que dependerá del espacio disponible en memoria masiva, de ahí que la distribución de los registros sea aleatoria dentro del soporte de almacenamiento. Para acceder a la posición física de un registro se utiliza una dirección o índice, no siendo necesario recorrer todo el fichero para encontrar un determinado registro.

A través de una transformación específica aplicada a la clave se obtendrá la dirección física en la que se encuentra el registro. Según la forma de realizar esta transformación, existen diferentes modos de acceso:



En el acceso directo la clave coincide con la dirección, debiendo ser numérica y comprendida dentro del rango de valores de las direcciones. Es el método más rápido.

El acceso indexado es similar a la organización indexada.

El acceso calculado se realiza utilizando funciones hash (ver más abajo).

**Ventajas**

- Muy rápido para búsquedas individuales.
- Ideal para sistemas que requieren acceso inmediato.

**Desventajas**

- Estos archivos deben almacenarse en dispositivos de memoria masiva de acceso directo, no se pueden usar soportes de acceso secuencial.
- Puede haber colisiones (dos claves que apuntan a la misma posición).
- Menos eficiente para recorridos secuenciales.

**Características**

- Registros de longitud fija.
- Permiten múltiples usuarios utilizándolos.
- Los registros no se borran físicamente, se coloca una marca de borrado.
- En los ficheros de acceso directo los registros siempre se encuentran en posiciones contiguas dentro del soporte de almacenamiento.

**ORGANIZACION SECUENCIAL INDEXADA**  $\equiv$  También llamados parcialmente indexados, al igual que en los ficheros indexados existe una zona de índices y otra zona de registros de datos, pero esta última se encuentra dividida en segmentos (bloques de registros) ordenados.

En la tabla de índices, cada fila hace referencia a cada uno de los segmentos. La clave corresponde al último registro y el índice apunta al registro inicial. Una vez que se accede al primer registro del segmento, dentro de él se localiza (de forma secuencial) el registro buscado.

**Características**

- Permite el acceso secuencial. Esto es muy interesante cuando la tasa de actividad es alta. En el acceso secuencial, además, los registros se leen ordenados por el campo clave.
- Permite el acceso directo a los registros. Realmente emula el acceso directo, empleando para ello las tablas de índices. Primero busca la clave en el área de índices y luego va a leer al área de datos en la dirección que le indica la tabla.

**Ventajas**

- Equilibra eficiencia en lectura masiva y búsquedas individuales.
- Permite mantener orden lógico sin sacrificar rendimiento.

**Desventajas**

- Más compleja de mantener que la secuencial pura.
- El índice puede volverse grande si hay muchos bloques.
- Solo se puede utilizar soportes direccionables.

**Comparación rápida**

Organización	Acceso rápido	Recorrido secuencial	Inserción intermedia	Uso típico
Secuencial	✗	✓	✗	Procesamiento por lotes
Directa	✓	✗	✓ (con colisiones)	Sistemas de consulta inmediata
Indexada	✓	✓	✓	Bases de datos
Secuencial indexada	✓	✓	✓	Archivos mixtos (consulta + lote)

**ACCESO CALCULADO O HASH**  $\equiv$  Cuando se usan ficheros indexados es necesario siempre tener que consultar una tabla para obtener la dirección de almacenamiento a partir de la clave. La técnica del acceso calculado o hash, permite accesos más rápidos, ya que, en lugar de consultar una tabla, se utiliza una transformación o función matemática (función de hashing) conocida, que a partir de la clave genera la dirección de cada registro del archivo. Si la clave es alfanumérica, deberá previamente ser transformada en un número.

El mayor problema que presenta este tipo de ficheros es que a partir de diferentes claves se obtenga la misma dirección al aplicar la función matemática o transformación. A este problema se le denomina **colisión**, y las claves que generan la misma dirección se conocen por **sinónimos**. Para resolver este problema se aplican diferentes métodos, como tener un bloque de excedentes o zona de sinónimos, o crear un archivo de sinónimos, etc.

Para llevar a cabo la transformación existen multitud de métodos, siendo algunos:

- **Módulo:** La dirección será igual al resto de la división entera entre la clave y el número de registros.
- **Extracción o truncamiento:** La dirección será igual a una parte de las cifras que se extraen de la clave.
- **División por un número primo:** Si N es el número máximo estimado de registros en el fichero y P es el número primo más próximo a N, se divide la llave por P y se utiliza como dirección el resto de la operación.
- **Cuadrado de la llave:** Si la llave no es un número muy alto, se eleva al cuadrado y se extraen como dirección unas cuantas cifras intermedias.
- **Plegamiento:** Se utiliza cuando la llave toma valores numéricos muy grandes. Consiste en recortar la llave en trozos y después sumar dichos trozos. Al resultado de la suma se le puede volver a aplicar cualquier método de los descritos.

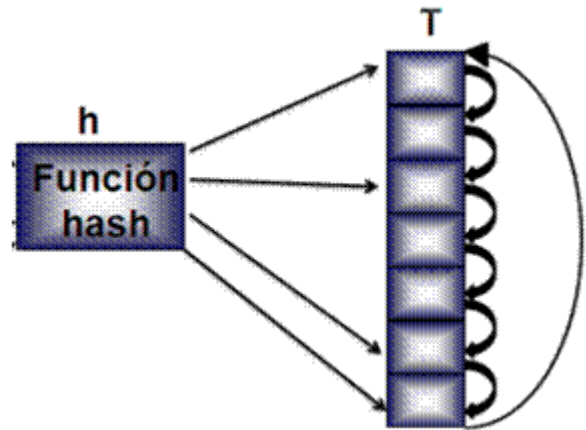
Una buena transformación o función de hash, será aquella que produzca el menor número de colisiones. En este caso hay que buscar una función, a ser posible biunívoca, que relacione los posibles valores de la clave con el conjunto de números correlativos de dirección. Esta función consistirá en realizar una serie de cálculos matemáticos con

el valor de la clave hasta obtener un número entre 1 y  $n$ , siendo  $n$  el número de direcciones que tiene el fichero.

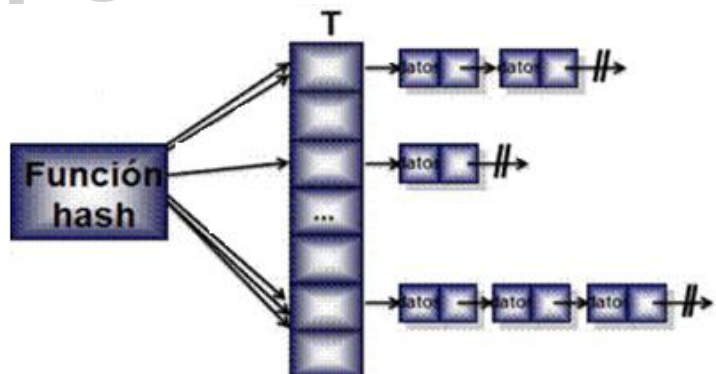
Para resolver las colisiones se usan principalmente dos métodos:

**HASHING CERRADO**  $\equiv$  Mantiene una tabla en memoria que, a medida que se le va introduciendo las claves, se van dispersando por la tabla mediante una función ya preestablecida (función hash).

Cuando ocurre una colisión, es decir, que la clave, al aplicarle la función hash, da una posición en el vector que ya estaba ocupada el programa aplicara una fórmula previamente determinada para encontrar otra casilla desocupada o hasta que llegue a la misma posición en la cual colisiono, si esto sucede y no se encontró una casilla disponible, puede realizar una **exploración lineal**, a partir de la posición que colisiono, hasta encontrar una posición vacía, si vuelve a la misma posición en la que colisionó, entonces quiere decir que el vector está lleno y no se admiten más claves.



**HASHING ABIERTO O DISPERSION ABIERTA**  $\equiv$  se basa en colocar en una lista enlazada todos los elementos que se dispersan en el mismo hueco. Así todos los elementos almacenados en una lista enlazada tendrán la misma clave. En este caso los huecos de la tabla dispersa no almacenan ya elementos, sino punteros a listas enlazadas asignadas dinámicamente que almacenan todos los elementos que se dispersan en ese hueco. En teoría no hay límite de espacio, dado que se asigna la memoria de forma dinámica. Lógicamente la memoria no es infinita, por lo que El número de elementos que pueden ser almacenados solo está limitado por la cantidad de memoria disponible en la computadora.



**ORGANIZACIÓN POR AGRUPAMIENTO O CLUSTERING**  $\equiv$  En este tipo de almacenamiento, se agrupan tablas cuyos ficheros comparten algunos atributos (campos), a los que se llama claves de agrupamiento.

**4. ALGORITMOS**

Se trata de un conjunto de instrucciones o reglas definidas y no-ambiguas, ordenadas y finitas que permiten solucionar un problema, realizar un cómputo, procesar datos y llevar a cabo otras tareas o actividades. Dados un estado inicial y una entrada, siguiendo los pasos sucesivos se llega a un estado final y se obtiene una solución. Los algoritmos son el objeto de estudio de la algoritmia.

**4.1 CONCEPTOS DE ALGORITMIA ELEMENTAL**

Se dice que, para resolver un problema concreto, un algoritmo requiere un tiempo del orden de  $t(n)$  para una función dada  $t$ , si existe una constante multiplicativa  $c$  y una implementación del algoritmo capaz de resolver todos los casos de tamaño  $n$  en un tiempo que no sea superior a  $c \cdot t(n)$  segundos.

Este importante concepto, es conocido como **complejidad temporal asintótica o notación asintótica** y mide **la complejidad computacional o algorítmica** de un algoritmo, o sea lo eficiente que es en cuanto a usar el menor número de recursos posibles para realizar su tarea.

- Permite comparar algoritmos independientemente del hardware o lenguaje.
- Nos enfocamos en el comportamiento de crecimiento, no en detalles de implementación.
- Es clave para determinar si un algoritmo es escalable.

La complejidad computacional puede ser, para cada una de estas anotaciones asintóticas:

**$O(1) \equiv$  constante**

**$O(\log n) \equiv$  logarítmica**

**$O(n) \equiv$  lineal**

**$O(n \log n) \equiv$  Casi Lineal**

**$O(n^2) \equiv$  Cuadrática**

**$O(n^3) \equiv$  Cúbica**

**$O(2^n) \equiv$  Exponencial**

**$O(n!) \equiv$  Factorial**

Están ordenadas de mayor eficiencia (constante) a menor eficiencia (factorial)

Estas notaciones asintóticas se calculas siempre **para el mejor caso, el caso promedio y el peor caso**, de cada algoritmo.

Ejemplo: si estudiamos la complejidad computacional de un algoritmo que ordene el contenido de una tabla, el mejor caso será cuando la tabla ya viene ordenada, el caso promedio será cuando tiene elementos ordenados y desordenados, y el peor caso cuando está todo desordenado.

**Técnicas de diseño de algoritmos**

**Algoritmos voraces (greedy)**  $\equiv$  Seleccionan los elementos más prometedores del conjunto de candidatos hasta encontrar una solución, dicho de otro modo, tomar la mejor decisión en cada paso, esperando que eso lleve a una solución global óptima.

En la mayoría de los casos la solución no es óptima.

Aplicación típica: Problemas de optimización.

**Ejemplos**

- Algoritmo de Kruskal (árbol de recubrimiento mínimo).
- Algoritmo de Prim.

**Divide y vencerás**  $\equiv$  Divide el problema en subproblemas independientes, resolverlos recursivamente y combinar sus soluciones, logrando así la solución al problema completo.

Aplicación típica: Ordenamiento, búsqueda, multiplicación de matrices.

**Ejemplos**

- Mergesort.
- Quicksort.
- Búsqueda binaria.

**Vuelta atrás (backtracking)**  $\equiv$  Explorar todas las soluciones posibles de forma sistemática, retrocediendo cuando una opción no lleva a solución válida. Garantiza encontrar todas las soluciones. Costoso en tiempo si no se optimiza.

Aplicación típica: Problemas de decisión, combinatoria, juegos.

**Ejemplos**

- Laberintos.
- Generación de combinaciones/permutaciones.

**Ramificación y poda (Branch & Bound)**  $\equiv$  Similar al backtracking, pero se descartan ramas del árbol de búsqueda si no pueden mejorar la solución actual.

**Programación dinámica**  $\equiv$  Resolver subproblemas solapados y almacenar sus soluciones para evitar recomputación.

**Algoritmos paralelos**  $\equiv$  Permiten la división de un problema en subproblemas de forma que se puedan ejecutar de forma simultánea en varios procesadores.

**Algoritmos determinísticos**  $\equiv$  el comportamiento del algoritmo es lineal: cada paso del algoritmo tiene únicamente un paso sucesor y otro antecesor.

**Algoritmos no determinísticos**  $\equiv$  el comportamiento del algoritmo tiene forma de árbol y a cada paso del algoritmo puede bifurcarse a cualquier número de pasos inmediatamente posteriores, además todas las ramas se ejecutan simultáneamente.

**4.2 ALGORITMOS DE BÚSQUEDA**

Un algoritmo de búsqueda es un conjunto de instrucciones que están diseñadas para localizar un elemento con ciertas propiedades dentro de una estructura de datos.

**Tipos de algoritmos de búsqueda**

**BUSQUEDA LINEA o SECUENCIAL**  $\equiv$  Consiste en ir comparando el elemento a buscar con cada elemento de la estructura hasta encontrarlo o hasta que se llegue al final.

No necesita que la estructura de datos está previamente ordenada o no.

La existencia se puede asegurar cuando el elemento es localizado, pero no se puede asegurar la no existencia hasta no haber analizado todos los elementos de la estructura.

- Complejidad:  $O(n)$
- Ventaja: No requiere orden
- Desventaja: Ineficiente en listas grandes

**BUSQUEDA BINARIA, DICOTÓMICA O DE LAS DICOTOMÍAS**  $\equiv$  El término dicotómica proviene de "dicotomía", que significa división en dos partes.

En cada paso del algoritmo, el conjunto de datos se divide en dos mitades, y se descarta una de ellas según el valor buscado. Este enfoque sigue el principio de divide y vencerás.

Requiere que la lista de elementos esté previamente ordenada por el campo de búsqueda.

Se compara el elemento a buscar con el elemento situado en la mita de la estructura, si este es mayor que el que buscamos, nos quedamos con la lista izquierda (los menores), si es mayor nos quedamos con la lista derecha (los mayores), y se repite el proceso hasta encontrar el elemento o no poder dividir cada una de las listas en dos mitades.

- Complejidad:  $O(\log n)$
- Ventaja: Muy eficiente
- Desventaja: Requiere orden previo

Ejemplo: queremos encontrar el número 28 en este array.

**Búsqueda Binaria**

0	1	2	3	4	5	6	7	8	9	
4	6	10	12	17	25	29	30	41	44	$28 > 17$
					25	29	30	41	44	$28 < 30$
					25	29				$28 > 25$
					29					$28 \neq 29$

Se determinó que el elemento no está haciendo solo 4 comparaciones!

Programación, 19/10/13-2014

15

**BUSQUEDA POR INTERPOLACIÓN**  $\equiv$  Es un algoritmo de búsqueda diseñado para arreglos ordenados numéricamente, especialmente cuando los datos están distribuidos de forma uniforme. Es una mejora sobre la búsqueda binaria en ciertos casos, ya que estima la posición probable del valor buscado en lugar de simplemente dividir el arreglo por la mitad.

Requiere que la estructura de datos esté previamente ordenada por un valor numérico.

Imagina que estás buscando un número en una lista ordenada. En lugar de ir al centro (como en la búsqueda binaria), la interpolación calcula una posición estimada basada en el valor buscado y los extremos del arreglo.

- Complejidad:  $O(\log \log n)$  en promedio
- Ventaja: Más rápida que binaria en datos uniformes
- Desventaja: Ineficiente en datos no uniformes

**OTROS**  $\equiv$  Jump Search, Exponential Search, búsqueda en árbol binario, búsqueda mediante tablas hash.

Algoritmo	Tipo de búsqueda	Complejidad	Requiere orden
Búsqueda lineal	No informada	$O(n)$	✗ No
Búsqueda binaria	No informada	$O(\log n)$	✓ Sí
Interpolación	No informada	$O(\log \log n)$ en promedio	✓ Sí (uniforme)
Jump Search	No informada	$O(\sqrt{n})$	✓ Sí
Exponential Search	No informada	$O(\log n)$	✓ Sí

## 4.3 ALGORITMOS DE ORDENACIÓN

### 4.3.1 CLASIFICACION DE ALGORITMOS DE ORDENACIÓN

Los algoritmos de ordenación se pueden clasificar en las siguientes maneras:

Según el **lugar** donde se realice la ordenación:

- **Algoritmos de ordenamiento interno**  $\equiv$  en la memoria del ordenador.
- **Algoritmos de ordenamiento externo**  $\equiv$  en un lugar externo como un disco duro.

Según el **tiempo** que tardan en realizar la ordenación, dadas entradas ya ordenadas o inversamente ordenadas:

- **Algoritmos de ordenación natural**  $\equiv$  Tarda lo mínimo posible cuando la entrada está ordenada.
- **Algoritmos de ordenación no natural**  $\equiv$  Tarda lo mínimo posible cuando la entrada está inversamente ordenada.

Según la **estabilidad**:

- **Algoritmos estables**  $\equiv$  Un algoritmo de ordenación es estable si mantiene el orden relativo de los elementos con valores iguales.

Ejemplo: Tenemos una lista de objetos con dos atributos: nombre y edad.

```
[("Ana", 25), ("Luis", 25), ("Carlos", 30)]
```

Si ordenamos por edad con un algoritmo estable, el resultado será:

```
[("Ana", 25), ("Luis", 25), ("Carlos", 30)]
```

Ana sigue antes que Luis, como en la lista original, se mantiene el orden en los elementos que tiene igual valor.

- **Algoritmos no estables**  $\equiv$  Un algoritmo no estable puede reordenar elementos iguales arbitrariamente, perdiendo su orden original.

Siguiendo el ejemplo anterior, un algoritmo no estable podría darnos una ordenación resultado como esta:

```
[("Luis", 25), ("Ana", 25), ("Carlos", 30)]
```

Aquí Luis aparece antes que Ana, aunque originalmente Ana estaba primero.

#### ¿Por qué importa la estabilidad?

En ordenaciones multicriterio (por ejemplo, primero por edad, luego por nombre), la estabilidad permite aplicar ordenaciones sucesivas sin perder la anterior.

Es crucial cuando los elementos tienen más información que el campo por el que se ordena.

### 4.3.2 ALGORITMOS DE ORDENACIÓN MAS CONOCIDOS

**METODO DE LA BURBUJA (BUBBLESORT)**  $\equiv$  Estable. Complejidad cuadrática  $O(n^2)$ . Recorre el array eligiendo parejas de elementos, si no están ordenados, los intercambia. Repite el proceso hasta que estén todos ordenados. Cada pasada consigue ordenar un elemento del array. El número necesario de pasadas es  $N-1$ , siendo  $N$  el número de elementos totales del array o estructura a ordenar.

- Método: Compara pares adyacentes y los intercambia si están en orden incorrecto.
- Ventajas: Fácil de implementar.
- Desventajas: Muy lento para listas grandes.

**ORDENACIÓN POR SELECCION (SELECTION SORT)**  $\equiv$  No estable. Complejidad cuadrática  $O(n^2)$ . Este método consiste en buscar el elemento más pequeño y ponerlo en la primera posición, buscar el siguiente más pequeño y ponerlo en la segunda posición, y así con todos los elementos hasta que estén ordenados.

- Método: Encuentra el mínimo y lo coloca en su posición final.
- Ventajas: Pocos intercambios.
- Desventajas: Muchas comparaciones.

**ORDENACIÓN POR INSERCIÓN DIRECTA (INSERTION SORT)**  $\equiv$  Complejidad cuadrática  $O(n^2)$ . En este método se supone una sublista ordenada de elementos, sobre la cual se insertan el resto de elementos en el lugar adecuado, para que la sublista no pierda el orden.

- Método: Inserta cada elemento en su lugar correcto en una lista ordenada.
- Ventajas: Eficiente para listas pequeñas o casi ordenadas.
- Desventajas: Ineficiente para listas grandes.

Existe una variación llamada de **inserción binaria**, que funciona exactamente igual, simplemente que para buscar el orden del elemento a insertar en la sublista usa el método de búsqueda binaria o dicotómica.

**METODO SHELL (SHELL)**  $\equiv$  No estable. Complejidad  $O(n \log n)$ . El Shell Sort es una generalización del ordenamiento por inserción, teniendo en cuenta dos observaciones:

1. El ordenamiento por inserción es eficiente si la entrada está "casi ordenada".
2. El ordenamiento por inserción es ineficiente, en general, porque mueve los valores sólo una posición cada vez.
3. Mucho más rápido que Insertion Sort para listas medianas.

El algoritmo Shell sort mejora el ordenamiento por inserción. No se compara a cada elemento con el de su izquierda o derecha, si no con el que está a un cierto número de lugares (llamado **salto o gaps**) a su izquierda. Este salto es constante y su valor inicial es  $N/2$  (siendo  $N$  el número de elementos y teniendo en cuenta la división entera). Esto permite que un elemento haga "pasos más grandes" hacia su posición esperada.

- Método: Generaliza el Insertion Sort usando saltos (gaps) entre elementos.
- Ventajas: Mucho más rápido que Insertion Sort para listas medianas.
- Desventajas: No es estable.

**ORDENACIÓN POR MEZCLA (MERGESORT)**  $\equiv$  Estable, **recursivo** y de complejidad  $O(n \log n)$ . Consiste en dividir en dos partes iguales el array a ordenar (divide y vencerás), ordenar cada una por separado, y luego mezclas ambos arrays, manteniendo la ordenación en un único array.

- Método: Divide la lista en mitades, ordena cada mitad y las fusiona.
- Ventajas: Muy eficiente y estable.
- Desventajas: Requiere memoria adicional.

**ORDENACIÓN RAPIDA (QUICKSORT)**  $\equiv$  No estable, **recursivo** y de complejidad  $O(n \log n)$ . Basado también en divide y vencerás. Consiste en coger un elemento **pivote** central, y subdividir la estructura en dos mitades, a la izquierda los elementos menores que el pivote, y a la derecha los elementos mayores que el pivote. El pivote se puede incluir en cualquiera de las dos sublistas. Aplicando de forma **recursiva** este mismo método a cada mitad resultante, finalmente el array quedará ordenado.

- Método: Elige un pivote, divide la lista y ordena recursivamente.
- Ventajas: Muy rápido en la práctica.
- Desventajas: Peor caso  $O(n^2)$  si el pivote es mal elegido.

**ORDENACIÓN POR MONTÍCULO (HEAPSORT)**  $\equiv$  No estable, **no recursivo** y de complejidad  $O(n \log n)$ . Consiste en almacenar todos los elementos en un montículo (heap) y extraerlos de la cima de uno en uno para obtener el arreglo ordenado.

- Método: Convierte la lista en un heap y extrae el máximo/mínimo.
- Ventajas: No requiere memoria adicional.
- Desventajas: No es estable.

**MÉTODOS DE ORDENAMIENTO POR DISTRIBUCIÓN**  $\equiv$  Estos métodos de ordenación utilizan **urnas** para depositar en ellas los registros en el proceso de ordenación. En cada recorrido de la lista se depositan en una urna aquellos registros cuya clave tienen una cierta correspondencia con "i"

Los métodos de distribución más importantes son:

- COUNTING SORT  $\equiv$  Estable. Complejidad  $O(n + k)$
- BUCKET SORT  $\equiv$  Estable. Complejidad  $O(nk)$
- RADIX SORT  $\equiv$  Estable. Complejidad  $O(n + k)$

Tipo de algoritmo	Método principal	Estabilidad	Complejidad promedio	Uso típico
Bubble Sort	Comparación directa	✓ Estable	$O(n^2)$	Educación, listas pequeñas
Selection Sort	Selección mínima	✗ No estable	$O(n^2)$	Simplicidad, pocos intercambios
Insertion Sort	Inserción directa	✓ Estable	$O(n^2)$	Listas casi ordenadas
Merge Sort	Divide y vencerás	✓ Estable	$O(n \log n)$	Grandes volúmenes de datos
Quick Sort	Divide y vencerás	✗ No estable	$O(n \log n)$	Rendimiento general
Heap Sort	Árbol de montículo	✗ No estable	$O(n \log n)$	Ordenación sin recursión
Counting Sort	Conteo de frecuencias	✓ Estable	$O(n + k)$	Datos enteros pequeños
Radix Sort	Ordenación por dígitos	✓ Estable	$O(nk)$	Códigos, números largos
Bucket Sort	Distribución en cubos	✓ Estable	$O(n + k)$	Datos uniformemente distribuidos

**5. SISTEMAS DE FICHEROS**

Los diferentes sistemas de archivos son simplemente diferentes formas de organizar y almacenar archivos en un disco duro, unidad flash o cualquier otro dispositivo de almacenamiento. Cada dispositivo de almacenamiento tiene una o más particiones y cada partición está "formateada" con un sistema de archivos. El proceso de formateado simplemente crea un sistema de archivos vacío de ese tipo en el dispositivo.

**Definición técnica**  $\equiv$  Un sistema de archivos define cómo se nombran, almacenan, jerarquizan y recuperan los archivos en un medio físico.

**Funciones**

- Organización: Divide el disco en bloques y asigna espacio a cada archivo.
- Jerarquía: Permite estructuras como carpetas, subcarpetas y rutas.
- Gestión de metadatos: Guarda información como nombre, tamaño, fecha de creación, permisos.
- Acceso eficiente: Optimiza la lectura/escritura de datos.
- Seguridad y permisos: Controla quién puede acceder o modificar archivos.
- Integridad: Algunos sistemas usan journaling para evitar corrupción tras apagones o fallos.

A continuación, se muestra una breve descripción de algunos de los sistemas de archivos más comunes que se encuentran, clasificados por el sistema operativo para el que suele ser usual, aunque no de forma exclusiva. No es una clasificación exhaustiva - hay muchos otros diferentes formatos, que se nombran posteriormente.

**5.1 Principales sistemas de archivos en Windows**

**FAT32 (File Allocation Table 32)**  $\equiv$  FAT32 es el sistema de archivos de Windows más antiguo, pero todavía se utiliza en dispositivos de medios extraíbles, solo en los dispositivos más pequeños.

- Ventajas: Altísima compatibilidad con casi todos los sistemas y dispositivos.
- Limitaciones:
  - No admite archivos mayores de 4 GB.
  - No soporta permisos ni cifrado.
- Ideal para: Pendrives, tarjetas SD, dispositivos multimedia.

**exFAT (Extended File Allocation Table)**  $\equiv$  Se trata de una actualización del FAT32, introducida por Microsoft en Windows Vista, para acabar con la limitación de 4 GB de FAT32. Se puede usar en Windows, macOS o GNU/Linux, aunque sólo en las versiones más recientes como a partir de Windows XP SP3 u OS X 10.6.5 Snow Leopard.

- Ventajas:
  - Supera las limitaciones de FAT32.
  - Compatible con Windows, macOS y Linux (con soporte).
- Limitaciones: No tiene cifrado ni journaling (registro de cambios, explicado posteriormente), por lo que es menos robusto ante fallos (menor tolerancia a fallos).
- Ideal para: Discos externos, almacenamiento portátil de gran capacidad.

**BLOQUE II****ANEXO****TEMA 03**

**NTFS (New Technology File System)** ≡ Las versiones modernas de Windows - desde Windows XP - utilizan el sistema de archivos NTFS para su partición del sistema. Las unidades externas se pueden formatear con FAT32 o NTFS. Sin los límites del tamaño máximo del FAT32, es una muy buena opción para discos duros y otras unidades externas. Su mayor desventaja está en las compatibilidades. Por ejemplo, de forma nativa macOS puede leer las unidades formateadas con él, pero no puede escribir en ellas. Linux puedes leer y escribir usando controladores de terceros.

- Ventajas: Soporta permisos, cifrado, compresión, journaling, cuotas de disco.
- Limitaciones:
  - macOS solo puede leerlo (sin escribir, salvo con software adicional).
  - Linux puede leer/escribir con controladores como ntfs-3g.
  - Ideal para: Discos internos, particiones del sistema, almacenamiento seguro.

**ReFS (Resilient File System)** ≡ El sistema de archivos ReFS (que significa sistema de archivos resistente) es una de las últimas novedades del sistema operativo Windows de Microsoft, y está diseñado para reemplazar en algunos casos al conocido y muy utilizado NTFS.

ReFS presenta nuevas funciones para mejorar la resistencia a la corrupción de datos mediante la detección con precisión de la misma de manera que se pueda corregir «on the fly». Dicho de otra manera, con este sistema de archivos se pueden detectar y corregir errores de corrupción de datos en las unidades de almacenamiento sin que el usuario tenga que intervenir para nada, de manera automática y sin paradas de servicio.

- Ventajas:
  - o Diseñado para alta disponibilidad, integridad de datos y tolerancia a fallos.
  - o Soporta corrección automática de errores y volúmenes grandes.
  - o Longitud máxima de archivo de 35 PB (ReFS) frente a 256 TB (NTFS)
- Limitaciones:
  - o No compatible con versiones domésticas de Windows-
  - o No tiene compresión ni cifrado de archivos (NTFS si los tiene)
  - o No se puede usar como partición de arranque.
- Ideal para: Servidores, almacenamiento redundante, entornos empresariales

**Compatibilidad con otros sistemas operativos**

Sistema de archivos	Windows	macOS	Linux	Android
FAT32	✓	✓	✓	✓
exFAT	✓	✓	✓ *	✓
NTFS	✓	Lectura	✓ *	✗
ReFS	✓	✗	✗	✗

**5.2 Principales sistemas de archivos en Mac**

**HFS+ (Mac OS Extended / Plus)** ≡ Admite unidades de almacenamiento internas y externas.

Los sistemas GNU/Linux pueden trabajar con HFS+ sin problemas.

Los sistemas Windows sólo podrán leer el contenido de los discos formateados con HFS+, pero no escribir en ellos.

Obsoleto en versiones modernas de macOS

- Características
  - o Journaling (registro de cambios)
  - o Compatible con Time Machine
- Limitaciones: Menor rendimiento que APFS en SSD

**APFS (Apple File System)** ≡ Remplazo del sistema de archivos HFS+ usado por Apple. Está especialmente optimizado para dispositivos de estado sólido (SSD) y almacenamiento flash.

Características:

- Tamaño máximo por partición: 7,5 EB
- Tamaño máximo por archivo: 2 EB
- Tamaño máximo del nombre de los archivos: 255 caracteres
- Copy-on-write (protección contra corrupción)
- Cifrado nativo
- Snapshots (copias de seguridad instantáneas)

Limitaciones:

- No compatible con Windows ni Linux sin software adicional
- No recomendado para discos externos si se usan en otros sistemas

Los sistemas operativos macOS también son compatibles con particiones formateadas en sistemas de archivos **FAT32 y exFAT**.

Sistema de archivos	macOS	Windows	Linux	Android
APFS	✓	✗	✗*	✗
HFS+	✓	✗	✓*	✗
exFAT	✓	✓	✓	✓
FAT32	✓	✓	✓	✓

### 5.3 Principales sistemas de archivos en Linux

**Ext2 / Ext3 / Ext4**  $\equiv$  A menudo se verán los sistemas de archivos Ext2, Ext3 y Ext4 en Linux. Ext2 es un sistema de archivos más antiguo, y carece de características importantes como el registro en diario (si se apaga el equipo o se bloquea una computadora al escribir en una unidad ext2, es posible que se pierdan datos.) Ext3 añade estas características de robustez a costa de cierta velocidad. Ext4 es más moderno y rápido (ahora es el sistema de archivos predeterminado en la mayoría de las distribuciones de Linux, y es más rápido.) Windows y Mac **no son compatibles** con estos sistemas de archivos; necesitará una herramienta de terceros para acceder a los archivos de dichos sistemas de archivos. Por esta razón, a menudo es ideal para formatear las particiones del sistema Linux como ext4 y dejar dispositivos extraíbles formateados con FAT32 o NTFS si se necesita compatibilidad con otros sistemas operativos. **Linux puede leer y escribir en FAT32 o NTFS.**

- EXT3  $\rightarrow$  Es un sistema de archivos principalmente utilizado en distribuciones Linux con registro por diario (**journaling**). Está siendo reemplazado por su sucesor, ext4, aunque todavía se utiliza.

#### ¿Qué es el journaling?

El journaling se basa en llevar un registro diario en el que se almacena la información necesaria para restablecer los datos del sistema afectados por un cambio, en caso de que falle.

- EXT4  $\rightarrow$  Es una mejora compatible de ext3 que utiliza menos CPU y mejora la velocidad de lectura y escritura, aunque es más lento en la eliminación de archivos que Ext3.

En ext4 se introducen los **exents**, que se utilizan para reemplazar al tradicional esquema de bloques utilizado por ext2 y ext3. Los exents mejoran el rendimiento al trabajar con ficheros de gran tamaño y evitan la fragmentación.

Tanto macOS como Windows tienen aplicaciones de terceros para poder acceder a los discos duros formateados con EXT3/EXT4 pese a **no haber soporte nativo**.

**Btrfs (B-tree FS)**  $\equiv$  No es el predeterminado en la mayoría de las distribuciones de Linux en este momento, pero probablemente reemplazará a Ext4. El objetivo es proporcionar características adicionales que permitan a Linux escalar a mayores cantidades de almacenamiento.

- Ventajas
  - o Snapshots, compresión, RAID nativo.
  - o Verificación de integridad.
- Limitaciones:
  - o Aún en evolución, no tan robusto como ext4 en todos los casos.
  - o Ideal para: Sistemas avanzados, entornos de prueba, NAS.

**XFS** ≡ Es un sistema de archivos de alto rendimiento para Linux, diseñado para manejar grandes volúmenes de datos con eficiencia (archivos grandes), especialmente en entornos de servidor y almacenamiento intensivo.

- Ventajas
  - o Alto rendimiento en lectura/escritura secuencial.
  - o Muy usado en servidores y bases de datos.
  - o Soporta Journaling
- Limitaciones:
  - o No permite reducir el tamaño de la partición.
  - o Ideal para: Volúmenes grandes, servidores.

**F2FS (Flash-Friendly FS)** ≡ Optimizado para dispositivos SSD.

- Ventajas
  - o Optimizado para NAND flash y SSD.
  - o Menor desgaste de bloques.
- Limitaciones:
  - o No tan extendido como ext4.
- Ideal para: Smartphones, Raspberry Pi, SSDs.

**OTROS SISTEMAS** ≡ para conocimiento, existen otros sistemas de archivos para Linux, menos utilizados.

- ZFS, ReiserFS

**COMPATIBILIDADES DE LOS SISTEMAS OPERATIVOS MÁS USUALES**

Se muestra una tabla en la que se relacionan los sistemas de ficheros más usuales, y cuáles son los sistemas operativos compatibles con cada uno de ellos.

SISTEMA DE ARCHIVOS	WIN 8/10/11	macOS	GNU/LINUX
<b>FAT32</b>	Sí	Sí	Sí
<b>NTFS</b>	Sí	Con apps de terceros	Sí, aunque puede necesitar drivers
<b>EXFAT</b>	Sí	Sí	Sí, aunque puede necesitar drivers
<b>HFS+</b>	Con apps de terceros	Sí	Sí
<b>APFS</b>	Con apps de terceros	Sí	No inicialmente, aunque hay drivers para intentarlo
<b>EXT4</b>	Con apps de terceros	Con apps de terceros	Sí

El "más compatible" es FAT32, pero tiene el inconveniente de las limitaciones de tamaño citada anteriormente, por lo que, aunque aún se usa muchísimo, de está dejando de usarse de forma paulatina.

A simple vista el sistema **exFAT** es el más versátil, dado que es compatible con el resto de sistemas operativos. Algunas combinaciones no son compatibles de forma nativa, se necesitan herramientas de terceros.

## **6. FORMATOS DE INFORMACION**

Por regla general, aunque no es obligatorio, los archivos están formados por un nombre, un punto y una extensión (Ejemplo: PROGRAMA.EXE).

El nombre nos sirve para diferenciar unos archivos de otros y la extensión para atribuirle unas propiedades concretas.

Estas propiedades asociadas o "formato de archivo" vienen dadas por las letras que conforman la extensión. Normalmente su máximo son tres letras aunque existen algunas excepciones (.jpeg, .html, .java, etc.). Cada uno de estos pequeños grupos de caracteres está asociado a un tipo de archivo.

Podemos dividir los archivos en dos grandes grupos. Éstos son los ejecutables y los no ejecutables o archivos de datos.

A continuación se muestra una lista de tipos de archivos (extensiones), clasificadas según la naturaleza del archivo al que corresponden (audio, texto, video, etc.).

### **6.1 FORMATOS DE IMAGEN**

#### **Tipos de archivos de imagen – Rasterizados vs Vectoriales**

**Formatos rasterizados**  $\equiv$  Imagen compuesta por píxeles. Cada píxel tiene color. Dependen de la resolución. Ejemplos: PEG, PNG, BMP, GIF, TIFF, WEBP

**Formatos vectoriales**  $\equiv$  Imagen basada en fórmulas matemáticas (líneas, curvas, formas). Escalable sin pérdida. Tamaño pequeño. Ejemplos: SVG, EPS, PDF, AI (Adobe Illustrator)

La compresión con pérdidas es un proceso que elimina parte de los datos de la imagen. Aunque esto reduce significativamente el tamaño del archivo, también disminuye la calidad de la imagen.

La compresión sin pérdidas sólo elimina los metadatos no esenciales. Sólo reduce ligeramente el tamaño del archivo, pero conserva la calidad de la imagen.

Las imágenes rasterizadas pueden ser con o sin pérdidas, mientras que las vectoriales no son ninguna de las dos porque su tamaño ya es pequeño, por lo que no necesitan ninguna compresión.

### 6.1.2 Los principales formatos rasterizados

1. **JPEG y JPG** ≡ Son el mismo formato de archivo, con extensión de archivo ligeramente distinta.

Es una imagen de trama con compresión con pérdida. JPEG elimina algunos datos para reducir el tamaño de su archivo, lo que disminuye a su vez la calidad de la imagen. JPEG es un formato de imagen plano, lo que significa que todas las ediciones se guardan en una sola capa, y no se pueden revertir las modificaciones. No admite transparencias, a diferencia de PNG y GIF.

Los principales navegadores, como Google Chrome, Safari y Mozilla Firefox, admiten este tipo de archivo de imagen desde su primera versión.

2. **PNG** ≡ Portable Network Graphics (PNG) es una trama con compresión sin pérdidas.

Cómo no tiene pérdidas, conserva los datos originales y su calidad sigue siendo la misma. Esto hace que PNG tenga una mayor calidad de imagen que JPEG. PNG puede manejar hasta 16 millones de colores, mientras que GIF sólo admite 256 colores. PNG también pueden conservar la transparencia

PNG es compatible con los principales navegadores y visores estándar del sistema operativo.

3. **BMP** ≡ Los archivos de imagen de mapa de bits (BMP) son rásteres que mapean píxeles individuales, lo que resulta en poca o ninguna compresión en una imagen dada.

Los archivos BMP son más grandes y poco prácticos de almacenar o procesar, y su calidad no es significativamente mejor que la de los formatos de imágenes rasterizadas como PNG o WebP.

Los principales navegadores y sistemas operativos admiten BMP.

BMP solía ser uno de los formatos de archivo de imagen más comunes pero, hoy en día, se considera obsoleto debido a su naturaleza no optimizada.

4. **GIF** ≡ El formato de intercambio de gráficos (GIF) es una trama que utiliza la compresión sin pérdidas.

Sin embargo, los archivos GIF son de 8 bits y sólo pueden mostrar 256 colores.

GIF es compatible con los principales navegadores y sistemas operativos.

5. **TIFF** ≡ El formato de archivo de imagen con etiquetas (TIFF) es una imagen de trama que admite la compresión con pérdidas, pero la gente suele utilizar TIFF como formato de imagen sin pérdidas. TIFF y TIF son los mismos formatos.

A pesar de su alta calidad, el formato TIFF no es compatible automáticamente con los principales navegadores. Tienes que instalar complementos o extensiones para renderizar un archivo TIFF en tu navegador.

## BLOQUE II

## TEMA 03

6. HEIF / HEIC ≡ El formato de archivo de imagen de alta eficiencia (HEIF) es un tipo de trama basado en el mapeo de píxeles, lo que significa que la calidad de la imagen disminuirá al ampliarla.

HEIF tiene el doble de eficiencia de compresión que el formato JPEG. Con el mismo tamaño de archivo, HEIF puede proporcionar una calidad de imagen mucho mejor.

La desventaja de HEIF es que tiene una compatibilidad limitada con los sistemas operativos y no es compatible con los navegadores web.

7. RAW ≡ RAW es un formato de archivo de imagen utilizado por las cámaras digitales para almacenar imágenes de alta calidad. Sin embargo, estas imágenes de alta calidad hacen que los archivos RAW tengan un gran tamaño. Un solo archivo de imagen RAW puede pesar cientos de megabytes.

8. PSD ≡ El documento de Photoshop (PSD) es un tipo de archivo nativo de Adobe Photoshop para guardar imágenes y trabajos en curso. Es una trama con compresión sin pérdidas.

### 6.1.3 Los principales formatos vectoriales

1. SVG ≡ Los gráficos vectoriales escalables (SVG) son un formato de archivo basado en vectores. Esto significa que, cuando se amplía una imagen SVG, no se pierde nada de su calidad de imagen.

SVG es un formato de imagen basado en XML.

Es posible insertar SVG directamente en una página web como código CSS.

SVG admite imágenes transparentes y puede incluir animaciones.

Todos los principales navegadores web admiten este formato de archivo de imagen.

2. EPS ≡ Encapsulated PostScript (EPS) es un vector con compresión sin pérdidas.

Al igual que el TIFF, los archivos EPS también se utilizan ampliamente para la impresión.

EPS no es compatible con los principales navegadores web y no se puede ver con los visores de imágenes predeterminados.

3. PDF ≡ El formato de documento portátil (PDF) puede resultar más familiar como formato de documento, pero también puede utilizarse para guardar imágenes e ilustraciones.

Un archivo PDF se basa en el mismo lenguaje PostScript que el EPS. Por lo tanto, el PDF es una excelente opción para la impresión. Es un vector con compresión sin pérdidas, lo que te permite ampliar una imagen PDF tanto como quieras.

Todos los principales navegadores admiten el formato PDF.

**BLOQUE II****TEMA 03**

4. **INDD**  $\equiv$  InDesign Document (INDD) es un formato de imagen vectorial utilizado por Adobe InDesign para guardar los archivos de los proyectos. Un archivo INDD puede contener varias páginas, lo que da lugar a archivos de gran tamaño.

INDD tampoco es un formato apto para la web, lo que significa que no puedes abrirlo directamente en ningún navegador.

5. **AI**  $\equiv$  También de la familia de software de Adobe, Illustrator Artwork (AI) es un formato nativo del software de gráficos vectoriales Adobe Illustrator.

Como AI es un vector, es posible escalar las imágenes AI tan grandes o tan pequeñas como se desee.

AI no es compatible con ningún navegador ni con los visores de imágenes predeterminados del sistema operativo. La única manera de ver este formato es a través del propio Adobe Illustrator.

Formato	Compresión	Pérdida	Transparencia	Nº de colores aprox.
<b>JPEG (.jpg)</b>	✓ Sí	▼ Con pérdida	✗ No	Hasta 16 millones (24 bits)
<b>PNG (.png)</b>	✓ Sí	✓ Sin pérdida	✓ Sí	Hasta 16 millones (24 bits)
<b>GIF (.gif)</b>	✓ Sí	✓ Sin pérdida (limitado)	✓ Sí (1 bit de transparencia)	Máx. 256 colores (8 bits)
<b>TIFF (.tif)</b>	✓ Opcional	✓ Sin pérdida o ▼ Con pérdida	✓ Sí (según configuración)	Hasta 16 millones (24-48 bits)
<b>BMP (.bmp)</b>	✗ No (sin compresión)	✓ Sin pérdida	✗ No	Hasta 16 millones (24 bits)
<b>WebP (.webp)</b>	✓ Sí	▼ Con pérdida o ✓ Sin pérdida	✓ Sí	Hasta 16 millones (24 bits)
<b>HEIF (.heic)</b>	✓ Sí	▼ Con pérdida o ✓ Sin pérdida	✓ Sí	Hasta 16 millones (24 bits)
<b>RAW (varios)</b>	✗ No	✓ Sin pérdida	✗ No	Depende del sensor (12-14 bits por canal)

**NOTA**  $\equiv$  GIF solo es sin pérdida si la imagen tiene  $\leq 256$  colores. Si tiene más, se pierde información de color.

## 6.2 FORMATOS DE AUDIO

Al igual que con las imágenes, aparte de conocer las distintas extensiones de formatos de audio, lo fundamental es saber si disponen de compresión o no y si tienen pérdida o no.

**MP3** ≡ Compresión con pérdida. Buena calidad con tamaño reducido.

**WAV** ≡ Sin compresión. Alta calidad, gran tamaño.

**FLAC** ≡ Compresión sin pérdida.

**AAC** ≡ Compresión con pérdida.

**OGG** ≡ Compresión con pérdida.

**AIFF** ≡ Sin compresión. Similar a WAV, pero desarrollado por Apple.

**ALAC** ≡ Compresión sin pérdida.

**OPUS** ≡ Compresión con pérdida. Usado en WhatsApp, Discord.





**PCM** ≡ Sin compresión.

## 6.3 FORMATOS DE VIDEO

**Todos** los formatos que se relacionan a continuación cumplen dos condiciones:

- Son formatos contenedores.
- Admiten compresión.

Un contenedor es un formato de archivo que agrupa diferentes flujos de datos:

-  Video (codificado con códecs como H.264, VP9...)
-  Audio (AAC, MP3, Opus...)
-  Subtítulos (SRT, ASS...)
-  Metadatos (autor, duración, capítulos...)

Ejemplo: Un archivo .mp4 puede contener Video en H.264, Audio en AAC y Subtítulos en SRT

**MP4** ≡ Con pérdida. Muy usado. Compatible con casi todo.

**MKV** ≡ Con pérdida o sin pérdida. Flexible. Soporta múltiples pistas.

**AVI** ≡ Con pérdida o sin pérdida. Antiguo. Menos eficiente que MP4.

**MOV** ≡ Con pérdida o sin pérdida. Usado por Apple. Buena calidad.

**WMV** ≡ Con pérdida. Alta compresión. Formato de Microsoft.

**FLV** ≡ Con pérdida Usado en streaming antiguo. Obsoleto.

## BLOQUE II

## TEMA 03

**WEBM** ≡ Con pérdida. Libre y abierto. Optimizado para web.

**MPEG/.MPG** ≡ Con pérdida. Usado en DVDs y transmisiones.

**MTS/.M2TS** ≡ Con pérdida. Alta definición en cámaras. Basado en MPEG-2.

**3GP** ≡ Con pérdida Usado en móviles antiguos. Muy comprimido.

### 6.4 FORMATOS DE COMPRESIÓN DE ARCHIVOS

**zip** ≡ Muy popular, compatible con casi todos los sistemas operativos. Soporta compresión sin pérdida.

**rar** ≡ Formato propietario con buena compresión. Requiere software específico como WinRAR.

**7z** ≡ Alta tasa de compresión. Usado por 7-Zip, libre y eficiente.

**tar** ≡ No comprime por sí solo, pero se usa para empaquetar archivos en Linux/Unix.

**tar.gz** ≡ Archivo TAR comprimido con gzip. Muy común en distribuciones de software.

**tar.bz2** ≡ Archivo TAR comprimido con bzip2. Mejor compresión que gzip, pero más lento.

**Gz** ≡ Compresión con gzip. Usado para archivos individuales.

**bz2** ≡ Compresión con bzip2. Más eficiente que gzip en algunos casos.

**xz** ≡ Compresión moderna, muy eficiente. Usado en distribuciones Linux.

**cab** ≡ Formato de Microsoft para empaquetar archivos en Windows.

**iso** ≡ Imagen de disco que puede contener archivos comprimidos internamente.

**dmg** ≡ Imagen de disco para macOS, puede incluir compresión.

**apk** ≡ Paquete de instalación para Android, basado en ZIP.

**Jar** ≡ Archivo Java empaquetado, también basado en ZIP.

### 6.5 FICHEROS PDF

**PDF** ≡ Portable Document Format. Desarrollado por Adobe para representar documentos de forma independiente del software, hardware o sistema operativo. Su objetivo es preservar el diseño, tipografía, imágenes y estructura del documento original.

- Extensión: .pdf
- Compresión: Sí
- Ventajas: Universal, seguro, imprimible, admite firmas digitales y protección por contraseña

## BLOQUE II

## TEMA 03

**PDF/A** ≡ PDF for Archiving. Es una versión especializada de PDF diseñada para **preservación a largo plazo** de documentos digitales. Es una norma ISO (19005) que prohíbe elementos que dificultan la conservación, como: Enlaces externos, Elementos multimedia como audio o vídeo, Javascript, Fuentes no incrustadas.

### Versiones de PDF/A

- PDF/A-1: Basado en PDF 1.4. No permite compresión JPEG2000 ni transparencia.
- PDF/A-2: Basado en PDF 1.7. Permite JPEG2000, capas, y archivos adjuntos.
- PDF/A-3: Permite adjuntar cualquier tipo de archivo (como XML, CSV, etc.).
- PDF/A-4: Basado en PDF 2.0. Mejora compatibilidad y estructura.

OTROS FORMATOS ≡ algunos formatos adicionales relacionados con PDF son:

**PDF/X** ≡ Para impresión profesional. Elimina elementos no imprimibles.

**PDF/E** ≡ Para ingeniería. Soporta CAD, modelos 3D.

**PDF/UA** ≡ Para accesibilidad. Compatible con lectores de pantalla.

**PDF 2.0** ≡ Última versión del estándar. Mejora seguridad, estructura y metadatos.

## 6.6 FORMATOS DE ARCHIVOS DE INTERCABIO DE DATOS

**JSON** (.json) ≡ JavaScript Object Notation: ligero, legible, ampliamente usado en APIs y aplicaciones web. Soporta estructuras anidadas.

**XML** (.xml) ≡ eXtensible Markup Language: más complejo que JSON, pero muy flexible. Usado en configuración, documentos, y servicios SOAP.

**CSV** (.csv) ≡ Comma-Separated Values: formato de texto plano para datos tabulares. Ideal para hojas de cálculo y bases de datos simples.

**YAML** (.yaml / .yml) ≡ YAML Ain't Markup Language: legible para humanos, usado en configuraciones (Docker, GitHub Actions, etc.).

**TOML** (.toml) ≡ Tom's Obvious Minimal Language: formato de configuración simple, usado en proyectos como Rust.

**XLSX** (.xlsx) — Formato de Excel basado en XML. Usado para datos tabulares con formato y fórmulas.

**OTROS** ≡ Avro, NDJSON, GeoJSON