

# Índice

---

Grafos	1
Estructuras de datos	7
Árboles	8
Árboles de Búsqueda	14
Compresión de datos (Huffman)	19
Métodos de Ordenamiento	25
Hashing	33
Árbol B	38
Teoría de Base de Datos	44
Teoría de Objetos de Base de Datos	49
Data Warehouse	52

# Grafos

---

## CONCEPTO

**Definición:** un grafo puede definirse como  $G = (V, A)$ , donde  $V$  representa a un conjunto de vértices/nodos y  $A$  a las aristas (que son las relaciones entre los vértices).

**Nodos simples y compuestos:** en la computación se considera que los nodos pueden estar compuestos por muchos valores (formando así una estructura de datos) o por un único valor.

- Si un nodo contiene un único valor, se llama nodo simple.
- Si hay un conjunto de valores que caracteriza a un nodo, se llama nodo compuesto.

**Grado de los nodos:** cantidad de aristas que afectan a un nodo. En los grafos dirigidos distinguimos:

- Grado positivo: cantidad de aristas que salen de un nodo.
- Grado negativo: cantidad de aristas que llegan a un nodo.

**Objetivo:** el objetivo de los grafos es modelizar un problema a través de un modelo abstracto donde los elementos del problema son los vértices y las relaciones entre ellos las aristas. Hablamos de modelo abstracto dado que un grafo es una estructura que no existe realmente. Para que se convierta en un almacenamiento concreto habrá que representarlo en una estructura computacional.

## REPRESENTACIÓN COMPUTACIONAL DE GRAFOS

**Definición:** existen dos formas de representar a los grafos computacionalmente:

- Estática: se establece un espacio fijo, el cual no cambia en función de cómo cambia el grafo. En cambio, ya desde un principio contempla todas las relaciones posibles entre todos los vértices del grafo, existan o no. Su representación computacional se construye sobre estructuras computacionales rígidas (como vectores y matrices):
  - Matriz de adyacencia.
  - Matriz de incidencia.
- Dinámica: acompañan la dinámica del grafo. El espacio utilizado va cambiando en función de cómo va cambiando el grafo. Para su implementación es necesario el uso de punteros que vinculen las diferentes posiciones de memoria que representan los vértices y los arcos. Entonces, el espacio ocupado es el espacio que ocupan en memoria los vértices y aristas, más el espacio necesario para los punteros. Sus representaciones posibles son:
  - Listas de adyacencia.
  - Dinámica con grado fijo.
  - Pfaltz.

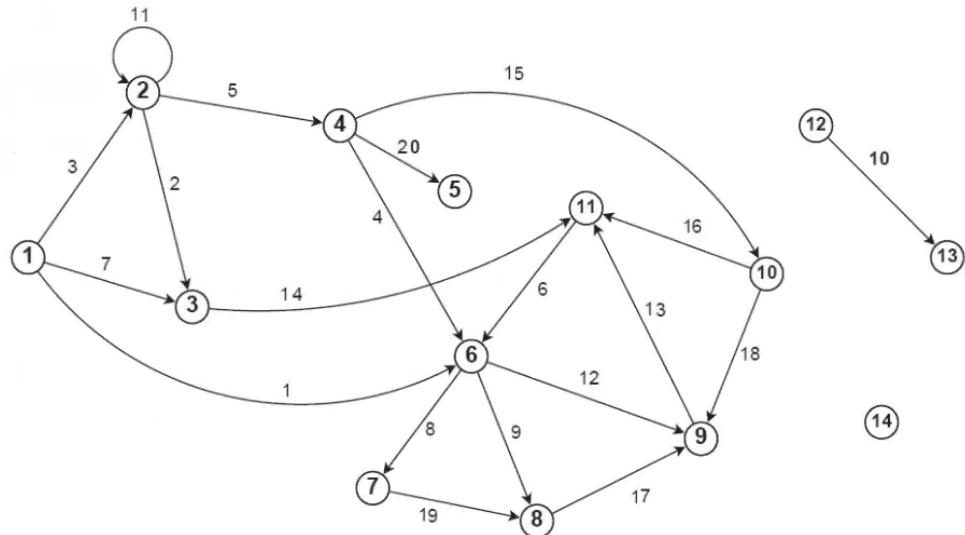
**Comparación de ambos métodos (estático y dinámico) según varios criterios:**

- Espacio ocupado: las representaciones dinámicas no siempre ocupan menos espacio que las estáticas. Esto depende de la cantidad de aristas que tenga el grafo en función de todas las aristas posibles. Cuantas más aristas posea un grafo, menor será el espacio desperdiciado por las representaciones estáticas y mayor será el desperdicio representado por los punteros en las representaciones dinámicas.
- Velocidad: las representaciones estáticas suelen ser más rápidas que las dinámicas debido a que en las estáticas el acceso es directo, mientras que en las dinámicas debo ir saltando de nodo en nodo para acceder al elemento deseado.

- Cuándo usarlos:
  - Estáticas: se recomiendan para grafos más pequeños donde pueda utilizarse la potencia de la velocidad de acceso, o también para grafos densos, donde existe una gran cantidad de relaciones definidas dentro de todas las posibles.
  - Dinámicas: cuando hay un grafo grande, con gran cantidad de vértices. También cuando se tiene un grafo disperso (con muchos nodos pero pocas relaciones entre ellos). También se recomienda usarla si lo que queremos modelar tiende a cambiar mucho con el tiempo.

## REPRESENTACIÓN ESTÁTICA Y DINÁMICA

Representación estática:



- Matriz de adyacencia: es una matriz cuadrada donde las filas y columnas representan los nodos (la fila es el nodo origen y la columna el nodo destino) y las celdas representan cuántas aristas permiten ir del nodo origen al nodo destino (para los bucles se pone "2"). Tener cuidado si el grafo es direccionalizado: capaz se puede ir de 1 a 2 pero no de 2 a 1.

	1	2	3	4	5	6	...	14
1	0	1	1	0	0	1	0	0
2	0	2	1	1	0	0	0	0
...								
14								

- Matriz de incidencia: las filas representan los nodos y las columnas las aristas. Si ese nodo se relaciona con esa arista, en el nodo **origen** se pone un "**1**" y en el nodo **destino** se pone un "**2**". En caso de una **relación bucle** se pone un "**3**".

	R1	...	R11	...	14
1	1		0		0

2	0		3		0
...					
6	2		0		0
14	0		0		0

Limitaciones: no podemos poner atributos en las relaciones.

### Representación dinámica:

- Lista de adyacencia: es una lista que tiene un nodo por cada vértice que compone al grafo. A su vez, cada vértice tiene una sublista con los arcos que salen de él.  
Los NODOS TIPO VÉRTICE tienen tres componentes: uno para sus atributos, otro para apuntar al siguiente nodo de la lista y un tercer puntero a su sublista de aristas.  
En cambio, los NODOS TIPO ARISTA tienen sus atributos, un puntero a la arista siguiente y un puntero al vértice al cual dirige esa arista.

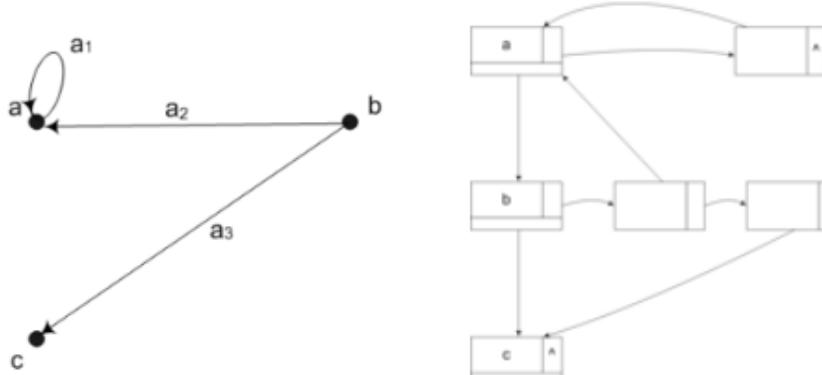
#### NODO TIPO VÉRTICE

idNodo  
struct atributos  
next\*  
subList\*

#### NODO TIPO ARISTA

idArco  
struct atributosDeLaRelacion  
next\*  
nodoDestino\*

#### Ejemplo:



- Dinámica con grado fijo: si nosotros supiéramos de antemano que el grafo tiene un grado fijo, podemos pasar a la siguiente implementación:

Id Nodo		
Atributos.....		
Dir 1	Dir 2	Dir 3
		Next

En vez de tener un NODO TIPO VÉRTICE y un NODO TIPO ARISTA vamos a tener un único tipo de estructura, que es la que vemos en la imagen. Vamos a seguir manteniendo el puntero next\* para poder enlazar toda la lista de nodos pero ya no vamos a tener más una sublista de aristas. La vamos a reemplazar por n punteros (siendo n el grado, en el ejemplo, 3).

Limitación: al reemplazar al NODO TIPO ARISTA por un simple puntero, no vamos a poder guardar los atributos de la arista.

- Pfaltz: esta implementación es una ampliación de lo que es la lista de adyacencia. En vez de tener un puntero a una lista de arcos salientes del nodo, vamos a tener dos cosas: un puntero a una lista de arcos salientes y un puntero a una lista de arcos entrantes.

Tengamos en cuenta que la cantidad de nodos arcos no se van a duplicar, sino que al nodo de tipo arco se le va a agregar un puntero al próximo entrante, al próximo saliente y un puntero al nodo origen.

## ALGORITMO DE DIJKSTRA

**Definición:** seguimos con el mismo grafo de ejemplo de antes pero ahora vamos a considerar que los números arriba de las aristas son distancias.

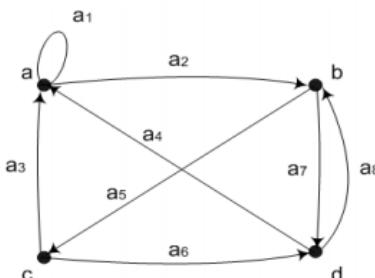
El algoritmo de dijkstra nos permite calcular la distancia mínima para llegar de un nodo origen a todos los nodos destino que se pueda. Hay dos formas de hacerlo y ambas son recursivas.

- Búsqueda en profundidad (depth first): comienza en el nodo origen del paso o camino y avanza al primer vértice que está conectado con él. Verifica si ese vértice es quien se está buscando como destino, de no ser así se traslada a este nuevo nodo a través del arco utilizado y vuelve a realizar la misma operación. Realiza esta operación hasta que no puede avanzar más por no existir arcos que salgan del nodo en que se encuentra. En ese momento regresa al nodo anterior que lo llevó hasta allí y repite la operación realizada tomando otro arco. Esta operación se realiza hasta que se encuentra el destino buscado o cuando no haya más aristas que recorrer, en cuyo caso se puede afirmar que no existe paso o camino entre los nodos propuestos.

De existir varios pasos o caminos entre los dos nodos esta técnica no garantiza que encuentre el más corto u óptimo, sino que encuentra un camino posible.

Para evitar que el algoritmo entre en un loop infinito, se debe verificar que el nodo al cual se le pasará la búsqueda no haya sido ya evaluado, en cuyo caso se pasa a la siguiente relación.

Ejemplo: paso entre d y c: ( $d \rightarrow (a4) \rightarrow a \rightarrow (a1) \rightarrow a \rightarrow (a2) \rightarrow b \rightarrow (a5) \rightarrow c$ )

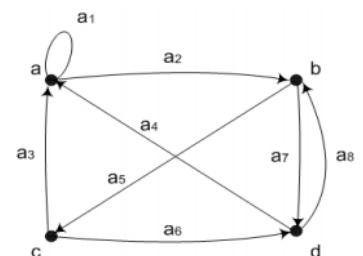


- Búsqueda en anchura (breadth first): evalúa primero todos los destinos de todos los arcos que parten del vértice origen del paso o camino a evaluar. Esta técnica no requiere retroceder. Una vez que se cargaron en cola todos los nodos vecinos directos a procesar, se continúa trasladando la búsqueda al primer elemento a procesar de la cola, ingresando ahora en la cola todos los nodos relacionados directamente con este nodo al final de la estructura.

El paso encontrado es el más corto porque la técnica va buscando primero pasos de longitud uno, después de longitud dos, después tres y así sucesivamente. Cuando las aristas tienen distancias esto no es así.

El método finaliza cuando se encuentra el paso buscado o cuando no quedan más nodos para procesar.

Ejemplo: paso entre d y c: ( $d \rightarrow$  analiza ( $a4$ ) y ( $a8$ ) → ninguno lleva a c → se mueve a a y agrega b a la pila → analiza ( $a1$ ) y ( $a2$ ) → ninguno lleva a c → se mueve a b que estaba en la pila → analiza ( $a5$ ) y ( $a7$ ) → ( $a5$ ) lleva a c entonces termina).



## ALGORITMO DE FLOYD-WARSHALL DE CLAUSURA TRANSITIVA

**Definición:** es una matriz cuadrada dimensionada por la cantidad de nodos (las filas son los nodos origen). Lo que indica esta matriz es lo mismo que Dijkstra pero “todos contra todos”. Es decir, no hay un único nodo origen sino que vamos a calcularlo para todos los nodos. En el caso de los nodos que no tengan relación (es decir, que no se pueda llegar de uno a otro) se va a colocar  $\infty$  en la matriz.

## CARACTERIZACIÓN DE GRAFOS

**Definición:** los grafos se diferencian según las características de sus arcos:

- Grafo libre: grafo que no posee aristas/relaciones ( $A = \{\emptyset\}$ ). Todos los vértices son aislados.
- Grafo completo: es aquel grafo que contiene todas las aristas posibles. Cada vértice está conectado a todos los vértices que componen al grafo, incluido él mismo.
- Grafo regular: es aquel grafo donde cada vértice tiene el mismo grado (positivo o negativo).
- Grafo simple: existe sólo una arista que une a dos vértices específicos.
- Grafo complejo: puede existir más de un arco que vincule dos vértices cualesquiera. Cualquier grafo que no cumpla con la condición de ser simple se considera complejo.
- Grafo conexo: existe al menos una conexión entre todos los nodos que conforman el grafo, sea esta directa (a través de un arco entre ambos) o indirecta (a través de más de un arco entre ambos). De cualquier vértice puedo llegar a cualquier otro.
- Grafo no conexo: un grupo de vértices no está conectado con el resto de los vértices. Cualquier grafo que no cumpla con la condición de ser conexo se considera no conexo.
- Grafo complementario ( $G^c$ ): es aquel que está compuesto por los mismos vértices que  $G$  pero su conjunto de aristas son todas aquellas que le faltan a  $G$  para ser un grafo completo.

## CLASIFICACIÓN DE GRAFOS

**Según la dirección de las aristas:**

- Grafos dirigidos: las aristas tienen sentido.
- Grafos no dirigidos: las aristas no tienen sentido.

**Según las restricciones que pueden ser aplicadas a las relaciones que modelan:**

- Grafos restrictos: la relación que se modela no debe cumplir la propiedad de reflexividad (no tiene bucles), simetría (no pueden existir ciclos simples) y transitividad. Si, en cambio, un grafo cumple con estas tres condiciones, se dice que es equivalente.
- Grafos irrestrictos: no se aplica ninguna restricción a la relación que se modela. Son los grafos más complicados de administrar, dado que es común la aparición de ciclos.

## CAMINOS, PASOS Y CICLOS

**Camino (entre nodos):** existe una vinculación directa o indirecta entre ambos nodos. En los grafos dirigidos existe camino entre dos vértices cuando existe vinculación entre ellos, independientemente del sentido de los arcos.

**Paso (entre nodos):** existe un camino entre ambos nodos pero con un sentido preestablecido. Como en este caso es relevante el sentido, solo se evalúan pasos en los grafos dirigidos, dado que en los grafos no dirigidos el concepto de paso se iguala al de camino.

**Ciclo (entre nodos):** es un paso o un camino donde el origen y el destino son iguales. Un ciclo puede estar compuesto por uno o más arcos.

## PREGUNTAS DE PARCIAL

1. Si se usara Pfaltz para implementar un árbol, el puntero link sería siempre nulo. V o F.

Verdadero. Los nodos en los árboles tienen un sólo camino desde la raíz hasta ellos, por lo que ese campo estaría vacío.

2. En Pfaltz, el llink del último arco agregado siempre será nulo. V o F.

Verdadero.

3. Si el puntero ledge de un nodo no es nulo, entonces el nodo es un terminal. V o F.

4. El algoritmo de Pfaltz se basa en el concepto de grafo bipartito. V o F.

5. Detalle las estructuras del algoritmo de Pfaltz.

6. Pfaltz. Si el grado de un grafo es 2, entonces existe al menos una celda de tipo arco donde el puntero rlink es no nulo. V o F.

7. Detalle las estructuras de las listas de adyacencia (explique qué información tienen los nodos de las listas).

# Estructuras de datos

---

## ESTRUCTURAS

### Clasificación:

- Biunívocas: son unívocas en ambos sentidos de la relación manteniendo uno o ningún predecesor y uno o ningún sucesor.
  - Pilas: es una estructura de datos que tiene como característica diferencial que su dinámica de ingreso y egreso es de tipo LIFO de forma tal que la forma de ingresar los datos es por un extremo de la pila y por el mismo extremo se realizan las extracciones.
  - Colas: una cola es una estructura de datos que se caracteriza por privilegiar el orden y la jerarquía manteniendo una dinámica FIFO, debido a que el primer elemento en entrar será también el primero en salir respetando el orden de llegada.
  - Listas: la lista es una estructura de datos que tiene una dinámica abierta. Esto quiere decir que dentro de una lista, a la hora de insertar un elemento, se recorre toda la lista y se coloca el elemento en la posición que se requiera, dependiendo de si se desea o no mantener la lista ordenada por algún valor de los nodos que la componen.
    - Lista lineal: es la lista tradicional. Comienza con un elemento y el último puntero del nodo apunta a NULL.
    - Lista circular: el último nodo apunta al primero.
    - Lista doblemente enlazada: son las listas que se implementan con la posibilidad de que los nodos, aparte de tener un apuntador al nodo siguiente, tengan un apuntador al nodo anterior.
- Unívocas: mantienen un solo predecesor pero pueden tener más de un sucesor.
  - Árboles: es unívoco dado que solo cumple la unicidad en un sentido: cada elemento tiene un solo predecesor pero puede tener más de un sucesor.

# Árboles

---

## ÁRBOL

**Definición:** un árbol es un grafo que tiene que cumplir las siguientes condiciones:

- Grafo dirigido: las relaciones tienen que tener una dirección.
- Acíclico: no puede haber ciclos.
- Existe un camino único entre cada par de nodos.
- Los nodos tienen que superar en 1 la cantidad de arcos:  $(\#V) = (\#E) + 1$

Árbol computacional: para que sea computacional hay que agregar una condición a las otras 4:

- Tiene raíz única.

### Conceptos:

- Grado: máxima cantidad de hijos o subárboles que puede tener cada nodo. El mayor grado de los nodos va a dar el grado del árbol.
  - Grado 2: árbol binario.
  - Grado mayor a 2: árbol n-ario.
- Nivel: posición en la que se encuentra cada nodo con respecto a la raíz, considerando que la raíz se encuentra en el nivel 0.
- Profundidad: cantidad de niveles que tiene un árbol.
- Subárbol principal izquierdo de un nodo: es el subárbol que contiene a todos los nodos de los cuales puedo llegar a él.
- Subárbol principal derecho de un nodo: es el subárbol compuesto por todos los nodos a los cuales se puede llegar desde él.
- Árbol principal izquierdo: un árbol es principal izquierdo si existe un único nodo cuyo subárbol principal izquierdo es el árbol completo.
- Árbol principal derecho: un árbol es principal derecho si existe un único nodo cuyo subárbol principal derecho es el árbol completo. Este nodo va a ser la raíz. *Un árbol computacional es un árbol principal derecho.*
- Hoja/nodo maximal: nodos que no tienen ninguna relación saliente.

### Características:

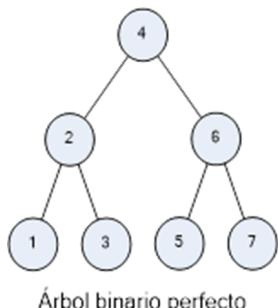
- Árbol completo: todos los nodos que no son hojas tienen el mismo grado, excepto posiblemente el último último nivel, el cual es completado de izquierda a derecha.
  - Árbol lleno: es un árbol completo y todas sus hojas se encuentran en el mismo nivel.
- Árbol balanceado: si me ubico en la raíz, la cantidad de nodos de la rama izquierda es igual a la de la rama derecha (pudiendo diferir en 1). Cualquier camino desde la raíz a una hoja tiene la misma longitud.
- Árbol perfectamente balanceado: la misma definición de balanceado pero para cada nodo.
- Árbol algebráico/de expresión: árbol binario usado para representar expresiones algebraicas. Las hojas representan las constantes (los números) y los no terminales representan los operadores. Los paréntesis no aparecen en el árbol ya que se deducen de las posiciones de los operadores en el árbol. Se lo recorre con inorden, postorden y preorden.

## CRECIMIENTO

**Definición:** el crecimiento de un árbol es exponencial en función del grado del mismo, o sea, que en cada nivel puede crecer en función del grado definido.

La máxima cantidad de elementos posibles de un árbol es: máx elementos = (grado^niveles) - 1

Ejemplo:



Si tomamos el árbol de la figura veremos que la máxima cantidad de elementos que puede tener es 7 que responde a  $2^3 - 1$  que es el grado elevado a 3 que son los niveles menos 1 que está dado por el grado de imparidad de la raíz

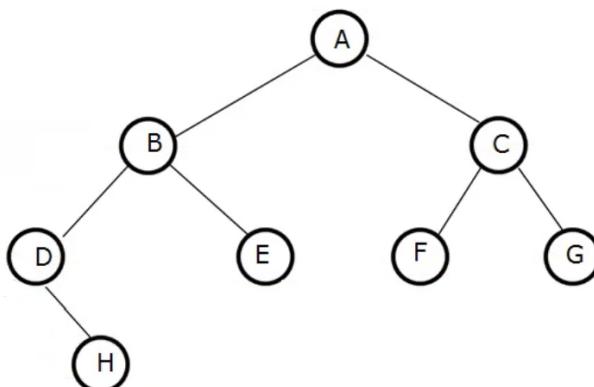
## IMPLEMENTACIONES EN MEMORIA DE LOS ÁRBOLES

**Estática:** se representa al árbol en un vector. Se carga el árbol en el vector de acuerdo a los niveles. El nodo en la posición 0 es la raíz y luego se carga de izquierda a derecha pasando por todos los niveles del árbol. En el array se ponen todos los nodos por más que no estén en el árbol real, poniendo NULL si es que no están.

Pueden hacerse cálculos para encontrar los hijos y el padre de un nodo:

- Hijo izquierdo:  $[(i+1)*2]-1$
- Hijo derecho:  $(i+1)*2$
- Padre:  $\text{int}((i-1)/2)$

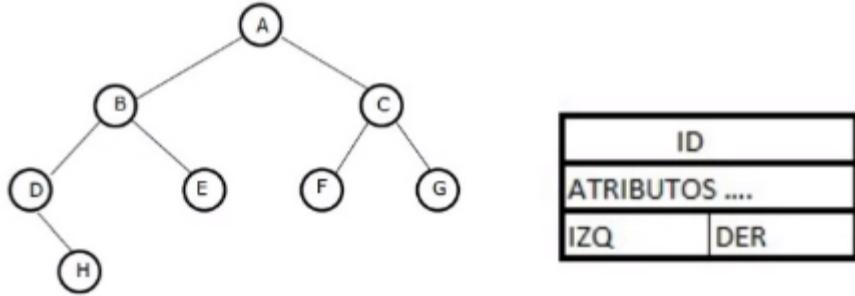
Esas fórmulas son para grado 2. Para grado 3 se agrega una ecuación (para cada grado también se agrega otra ecuación) y se multiplica por el grado y se resta uno. La  $i$  representa el subíndice del vector. Si algún cálculo da con coma se toma la parte entera. Como todas las fórmulas son aplicables al grado, el árbol debe ser de grado fijo.



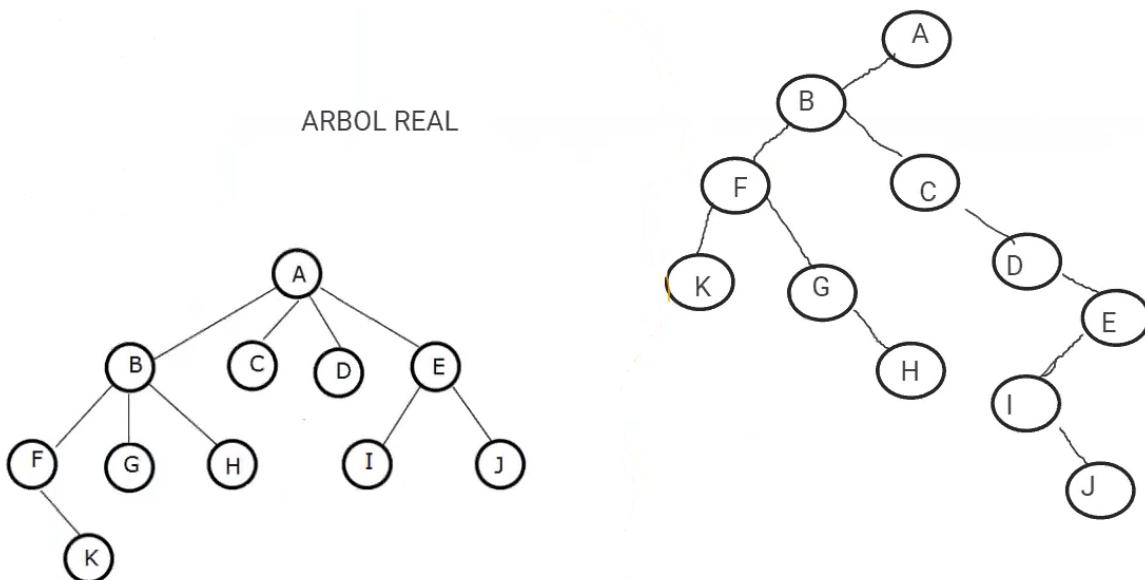
A	B	C	D	E	F	G		H
---	---	---	---	---	---	---	--	---

**Dinámica:**

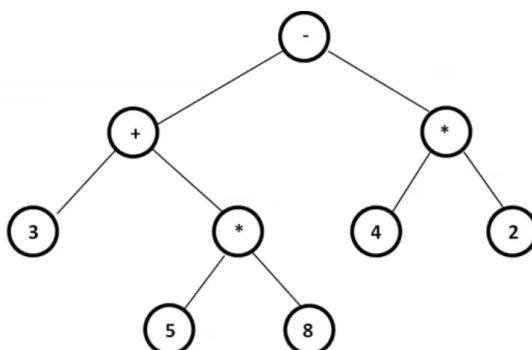
- Grado fijo: se define una estructura de tipo nodo que tiene los atributos y los dos punteros (uno al hijo izquierdo y otro al derecho). También se guarda un puntero a la raíz (llamado "front") para saber por dónde empezar.



- Grado variable/Knuth: a la izquierda vemos el árbol real. Ese árbol así como esta no puede guardarse en memoria. El que esta guardado en memoria es el que está a la derecha, que representa al mismo árbol con la diferencia de que es binario.  
Todo árbol tiene un hijo primogénito, que es el primero de la izquierda. De esa manera vamos a colocar al primer hijo a la izquierda y todos los hermanos irán a la derecha.



### RECORRIDO/BARRIDO DE ÁRBOLES



**Definición:** proceso de moverse entre los nodos de un árbol. Hay dos métodos principales:

- Recorrido en profundidad: son definidos en forma recursiva.
  - Preorden (RID): puede aplicarse a árboles no binarios.
  - Inorden/Simétrico (IRD): sólo para árboles binarios.
  - Postorden (IDR): puede aplicarse a árboles no binarios.

Preorden (RID)	In orden/Simétrico (IRD)	Post orden (IDR)
<pre>Barrido(x nodo) {     Recursivo(x); }  Recursivo(x nodo) {     if x IS NULL then         return;     print x--&gt;ID;     Recursivo(x--&gt;izq);     Recursivo(x--&gt;der); }</pre>	<pre>Barrido(x nodo) {     Recursivo(x); }  Recursivo(x nodo) {     if x IS NULL then         return;     Recursivo(x--&gt;izq);     print x--&gt; ID;     Recursivo(x--&gt;der); }</pre>	<pre>Barrido(x nodo) {     Recursivo(x); }  Recursivo(x nodo) {     if x IS NULL then         return;     Recursivo(x--&gt;izq);     Recursivo(x--&gt;der);     print x--&gt; ID; }</pre>
<b>Output:</b> -3*58*42	<b>Output:</b> 3+5*8-4*2	<b>Output:</b> 358*+42*-

- Recorrido en anchura: no son considerados recursivos. Se lee por niveles. **Output: +\*3\*4258.** Si se quisiera encontrar un elemento en un árbol buscando por niveles, podemos tomar la fórmula de crecimiento y despejar de ella la cantidad de niveles y llegaremos a que un árbol tiene una búsqueda logarítmica:

$$\begin{aligned}
 \text{elementos} &= \text{grado}^{\text{niveles}} - 1 \\
 \text{elementos} + 1 &= \text{grado}^{\text{niveles}} \\
 \log(\text{elementos} + 1) &= \text{niveles} * \log(\text{grado}) \\
 \text{niveles} &= \log(\text{elementos} + 1 - \text{grado}) \\
 \text{niveles} &> \log(\text{elementos})
 \end{aligned}$$

## IMPLEMENTACIÓN DE GRAFOS Y ÁRBOLES EN BASES DE DATOS

**Base de datos relacional:** se tienen dos tablas: una de nodos y una de relaciones. La tabla de relaciones debería tener dos FKs a la de nodos.

```

CREATE TABLE nodos(
    idnodo INT PRIMARY KEY,
    descripcion VARCHAR (20) NOT NULL,
    atributo1 VARCHAR (50) NULL
)
GO

CREATE TABLE relaciones(
    idrelacion INT PRIMARY KEY,
    origen INT NOT NULL REFERENCES NODOS,
    destino INT NOT NULL REFERENCES NODOS,
    atributo1 VARCHAR (50) NULL,
)
GO

```

**Base de datos noSQL:** en general se usa una base de datos orientada a grafos (como neo4j).

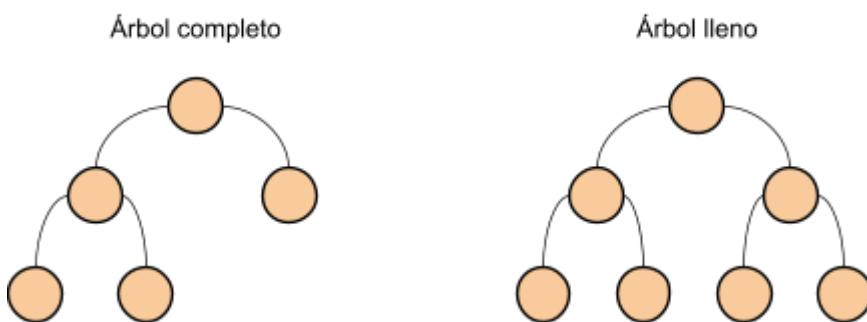
## PREGUNTAS DE PARCIAL

1. El barrido simétrico sólo puede realizarse en árboles binarios. V o F.

Verdadero. El barrido simétrico consiste en recorrer el árbol de la siguiente manera: IRD. Para que exista un nodo hijo izquierdo y uno derecho, el grado de todos los nodos debe ser 2, por lo que se trata de árboles binarios.

2. Explique en no más de 10 renglones la diferencia entre árbol completo y árbol lleno.

En un árbol completo todos los nodos que no son hojas tienen el mismo grado, excepto posiblemente el último último nivel, el cual es completado de izquierda a derecha. El árbol lleno es un árbol completo y todas sus hojas se encuentran en el mismo nivel. Gráficamente:



3. Explique para qué sirven los algoritmos de rotación a izquierda o derecha de un subárbol dado.

Los algoritmos de rotación sirven para balancear un árbol en caso de estar desbalanceado. Si la futura raíz está a la derecha entonces se rota el árbol a izquierda, pero si la futura raíz está a la izquierda entonces se rota el árbol a la derecha.

4. El barrido de pre-orden se puede hacer en todo tipo de árboles pero el simétrico es sólo para árboles binarios. V o F.

Verdadero.

5. Un grafo que posee  $n$  nodos y  $(n-1)$  arcos siempre es un árbol. V o F.

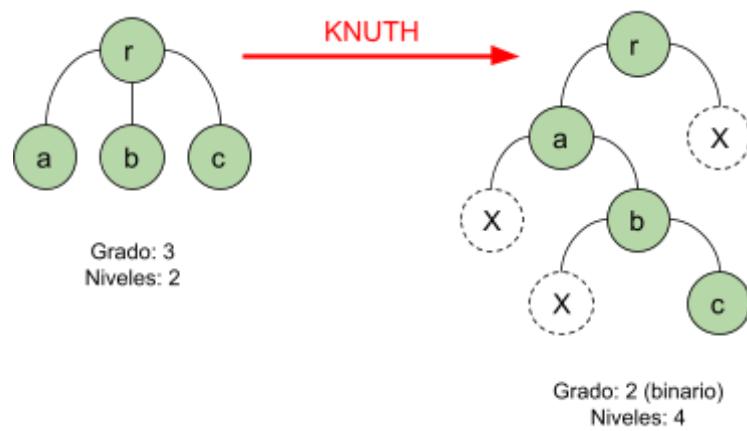
Falso. Para ser un árbol además debe cumplirse que: sea acíclico, exista un único camino entre un par de nodos y que sea dirigido (en un único sentido).

6. La transformada de Knuth es un algoritmo cuyo objetivo es representar en una estructura genérica un árbol  $n$ -ario. V o F.

Falso. Knuth transforma un árbol  $n$ -ario en un árbol binario para que pueda ser representado computacionalmente.

7. Knuth. Si un árbol  $n$ -ario tiene 2 niveles, entonces la profundidad máxima que puede alcanzar un nodo en la transformada de Knuth es  $n$ .

Falso. Ejemplo:



**8. Un grafo que posee un sólo nodo siempre es un árbol.**

Verdadero. Cumple con las 4 condiciones necesarias para ser árbol.

# Árboles de Búsqueda

## COMPLEJIDAD COMPUTACIONAL Y ESPACIAL

**Complejidad computacional:** cuánto procesa un algoritmo, un procedimiento o un programa.

**Complejidad espacial:** cuánto espacio físico necesitamos para que un algoritmo pueda correr/ejecutarse. Se mide en líneas de código representativas (como los for o llamados recursivos) que se ejecutan en producción.

La complejidad espacial y la computacional son inversamente proporcionales:



Solamente consideramos el primer cuadrante porque cualquier otro punto (incluso el origen) no tienen sentido. Hoy en día es más importante reducir la complejidad computacional (procesamiento).

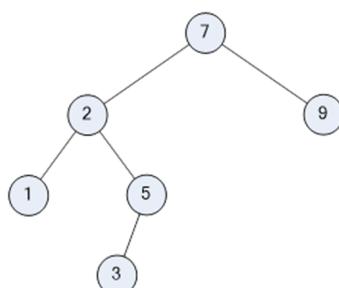
## ÁRBOL DE BÚSQUEDA

**Definición:** es un árbol que soporta las operaciones de búsqueda, inserción y eliminación de forma eficiente. En estos árboles las claves no aparecen en los nodos de forma arbitraria, sino que hay un criterio de orden que determina dónde una determinada clave puede ubicarse en el árbol.

**Criterios para que un árbol sea considerado de búsqueda:** las vamos a explicar en detalle más abajo

- Balanceado: si me ubico en la raíz, la cantidad de nodos de la rama izquierda es igual a la de la rama derecha (pudiendo diferir en 1).
- Perfectamente balanceado: es la misma definición de antes pero para cada nodo.
- AVL: en vez de contar la cantidad de nodos contamos la cantidad de niveles. Puede diferir en 1. Esto es para todo nodo.

**Árbol binario de búsqueda (ABB)**: es un árbol binario que internamente se guarda en un vector donde los elementos menores se ingresan a la izquierda y los mayores a la derecha. Para cada nodo X del árbol, los valores de todas las claves de su subárbol izquierdo son menores que la clave de X y los valores de todas las claves de su subárbol derecho son mayores que la clave de X. Si se lo recorre en inorden se muestran los elementos ordenados de forma ascendente.



Búsqueda:  $\ln 2 (n+1)$  siendo  $n$  la cantidad de claves totales.

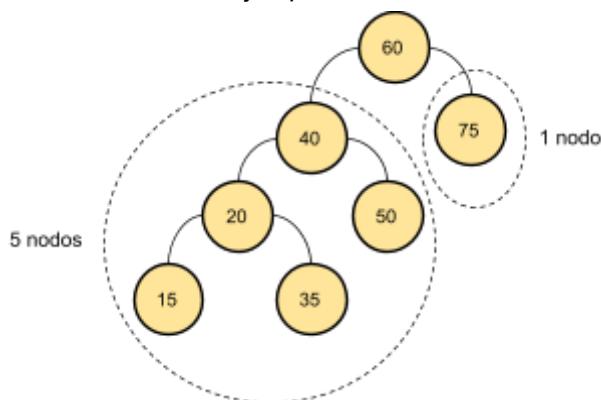
Para hacer una búsqueda hay que empezar leyendo por la raíz y hay que ir por las ramas de acuerdo a si es mayor o menor (por cada nivel paso una sola vez). Entonces hay que lograr que el árbol tenga la menor cantidad de niveles posibles. En el peor caso de todos hay que pasar por todos los niveles.

## CRITERIOS PARA QUE UN ÁRBOL SEA CONSIDERADO DE BÚSQUEDA

**Árbol balanceado:** si me ubico en la raíz, la cantidad de nodos de la rama izquierda es igual a la de la rama derecha (pudiendo diferir en 1).

**Árbol perfectamente balanceado:** si me ubico en cada nodo, la cantidad de nodos de la rama izquierda es igual a la de la rama derecha (pudiendo diferir en 1).

Algoritmos de rotación: sirven para balancear un árbol en caso de estar desbalanceado. Si la futura raíz está a la derecha entonces se rota el árbol a izquierda, pero si la futura raíz está a la izquierda entonces se rota el árbol a la derecha. Por ejemplo, vamos a balancear el siguiente árbol.

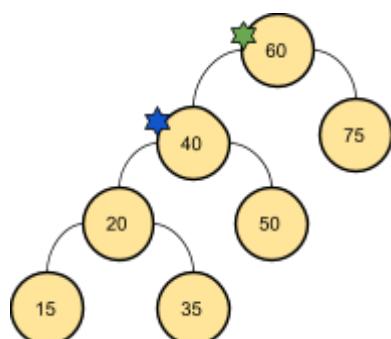


Primero empezamos equilibrando la raíz y una vez que eso esté hecho podemos empezar a equilibrar cada nivel. Tenemos 5 nodos de un lado y 1 del otro, por lo que no está balanceado. ¿Cuál debería ser el elemento raíz para que el árbol quede con 3 elementos a un lado y 3 al otro? Recordar que debemos respetar la regla de que los elementos menores estén a izquierda y los mayores a derecha.

Siempre vamos a elegir al valor medio (en valor). En este caso es el 40 porque la cantidad de valores es impar. Si fuera par tendríamos 2 valores medios y en ese caso tenemos que elegir el que esté ubicado más cerca de la raíz.

Para hacer este cambio nunca vamos a pisar datos. Solamente vamos a usar dos punteros:

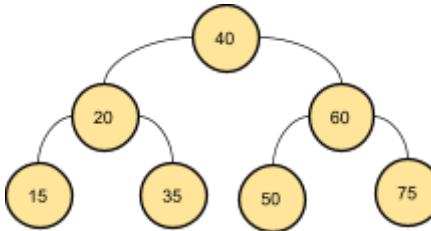
- Puntero verde: apunta a la posición a balancear (en este caso, la raíz).
- Puntero celeste: apunta a la clave que va a pasar a ocupar ese lugar (el 40).



Los cambios que hay que hacer son:

1. El hijo izquierdo va a ser la raíz.
2. La raíz va a pasar a ser el hijo derecho de la nueva raíz.

3. El hijo derecho del que va a ser la nueva raíz pasa a ser el hijo izquierdo de la antigua raíz.



Con esto nos aseguramos que la raíz quede balanceada. Después hay que hacer un recorrido por niveles para ver si todos los nodos están balanceados. Este procedimiento es para cuando la nueva raíz se encontraba a izquierda. Si hubiera estado a derecha sería lo mismo pero para el otro lado. También este caso es el más fácil porque la nueva raíz estaba en el nivel 1. Si hubiera estado más abajo tendríamos que haber repetido el procedimiento hasta llevar la nueva clave hasta la raíz.

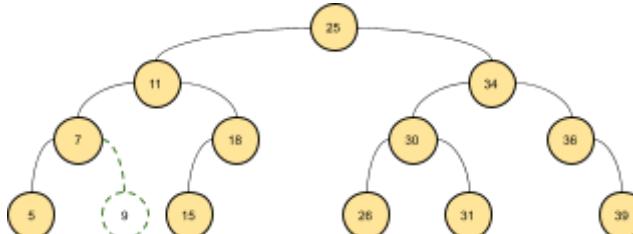
Ejemplo de búsqueda: supongamos que queremos leer el valor 15. Vamos a tener que hacer 3 lecturas: 40-20-15. Usando la fórmula  $\ln 2(8)=3$ . Esta fórmula siempre devuelve el peor de los casos, que es suponer que esté en el último nivel.

**Árbol AVL:** para cada nodo se debe cumplir que la cantidad de niveles de su rama izquierda es igual a la de su rama derecha. Como las otras condiciones, también puede diferir en 1.

Este algoritmo es largo (en tiempo de procesamiento) y muchas veces no “ganamos” nada al ejecutarlo dado que, si nuestro objetivo es reducir la cantidad de niveles pero todos los niveles ya están completos, no hay forma de reducirlos. En todo caso el algoritmo podría mover la hoja de un nivel como hoja de otro nivel para que quede AVL.

## ALTAS Y BAJAS EN UN ABB

**Alta ABB:** se busca un hueco libre yendo por la rama que corresponda (de acuerdo a si el valor es mayor o menor al que queremos ingresar). Por ejemplo: se quiere insertar el valor “9”.



**Baja ABB:** vamos a ver 3 casos:

1. Baja de nodo sin hijos
  - Recorrer el árbol y encontrar al elemento.
  - Guardar la dirección de su padre.
  - Guardar si es hijo izquierdo o derecho de su padre.
  - Acceder a los punteros del padre y setear el puntero izquierdo o derecho (según corresponda) a NULL.
2. Baja de nodo con un hijo
  - Recorrer el árbol y encontrar al elemento.
  - Guardar la dirección de su padre.
  - Guardar si es hijo izquierdo o derecho de su padre.
  - Libera nodo que se quiere eliminar pero antes dice que el puntero derecho del padre apunte al que era su único hijo.

3. Baja de nodo con 2 hijos: este es el primer ejemplo en el que se van a pisar datos en vez de mover punteros.

- a. Recorrer el árbol y encontrar al elemento.
- b. Guardar la dirección de su padre.
- c. Guardar si es hijo izquierdo o derecho de su padre.
- d. Se para en la rama derecha y busca al menor de todos los valores. Este valor va a pasar a ocupar el lugar del que queríamos eliminar.
- e. Ahora elimino al nodo donde estaba el valor seleccionado en el paso anterior.
- f. Ahora se verifica el balanceo. Si está balanceado no tiene que volver a balancear al árbol. Si no hubiera quedado balanceado debería balancearlo.

## PREGUNTAS DE PARCIAL

### **1. Explique la diferencia entre árbol balanceado y AVL en menos de 10 renglones.**

Un árbol balanceado es un árbol que, si nos paramos en la raíz, la cantidad de nodos de la rama izquierda es igual a la cantidad de nodos de la rama derecha (pudiendo diferir en 1). En cambio, en un árbol AVL para cada nodo se debe cumplir que la cantidad de niveles de su rama izquierda sea igual a la de su rama derecha. También puede diferir en 1.

### **2. Si un ABB es lleno, entonces también está balanceado. V o F.**

Verdadero. Un árbol lleno es un árbol completo en el cual todas las hojas se encuentran en el mismo nivel. Para ser balanceado, la cantidad de nodos de la rama izquierda a la raíz debe coincidir con la cantidad de nodos de la rama derecha.

# Compresión de datos (Huffman)

---

## ALGORITMOS DE COMPRESIÓN

**Tipos:** existen dos tipos de algoritmos de compresión:

- Con pérdida de datos: son algoritmos que pierden información al comprimir y, por esa razón, no son reversibles. Este tipo de algoritmo se utiliza para los archivos multimedia.
- Sin pérdida de datos: son algoritmos que comprimen archivos sin perder información, por lo cual son reversibles. Se utilizan para todos los archivos menos los multimedia. En la materia vamos a estudiar estos.

**¿Por qué es posible comprimir un archivo?**: la tabla ASCII se creó de longitud fija. Contiene 256 caracteres, por lo que para representar a un carácter se necesita 1 byte ( $2^{8 \text{ bits}} = 256 \text{ caracteres}$ ). De forma tal que un carácter va a ocupar si o si un byte, por más que no estemos usando los 256 caracteres del alfabeto ASCII. Pero en la realidad, la mayoría de los archivos no contienen los 256 caracteres ASCII, por lo que estamos desperdiciando bits en la representación de caracteres. Recordar que para el caso de UNICODE se usan 4 bytes.

## HUFFMAN

**Definición:** Huffman es un algoritmo de compresión de datos sin pérdida, que es muy eficiente con archivos de texto (txt, excel, word, pdf).

Lo que hace el algoritmo es identificar cada uno de los caracteres distintos en el archivo a comprimir y le asigna un código de longitud variable según qué tan frecuente sea. Cuanto mayor sea su frecuencia, menor será la longitud de su código.

**Estructuras usadas:** Huffman hace uso de tres estructuras

- Tabla de Huffman: esta tabla va a tener tantas filas como:  $(2 * \text{cant. caracteres}) - 1$ . Tiene las siguientes columnas:
  - Status (flag): indica si los caracteres están ubicados en el árbol.
  - Carácter.
  - Frecuencia: cuántas veces aparece el carácter en el archivo.
  - Código de longitud variable: va a arrancar en NULL.
  - Puntero al árbol.
- Árbol de Huffman: tiene un “puntero al padre” porque es posible recorrer el árbol desde las hojas hasta la raíz.

PUNTERO AL PADRE	
FRECUENCIA	
CARÁCTER	
IZQ	DER

- Pila

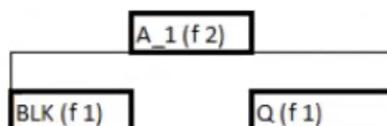
**Método de compresión de Huffman**: supongamos un archivo que tiene escrito “NEUQUEN”. Lo primero que hay que hacer es leer el archivo e identificar los caracteres usados. Estos caracteres se cargan en la tabla y se ordenan por frecuencia (que es la cantidad de veces que aparece el carácter en el archivo).

Status	Char	Frecuencia	Código	Dirección árbol
	E	3		
	N	3		
	U	2		
	BLK	1		
	Q	1		

La cantidad final de filas de esta tabla va a ser:  $(2 * \text{cant. caracteres}) - 1 = 9$ . Ya vamos a ver cómo se cargan las filas que faltan:

	Status	Char	Frec
0		E	3
1		N	3
2		U	2
3		BLK	1
4		Q	1
...			
8			

- 1) A partir de esta tabla el algoritmo tiene que construir el árbol de Huffman. Para esto, lo primero que va a hacer es partir de la frecuencia más chica (1) y la última posición llena de la tabla (en este caso, 4).  
El algoritmo va a ir armando un árbol binario desde las hojas hacia la raíz, buscando elementos que tengan la misma frecuencia. El primer par que va a encontrar es la letra Q con el carácter blanco BLK y va a armar un árbol de la siguiente manera, creando un padre ficticio ( $\alpha_1$ ) cuya frecuencia va a ser la suma de la frecuencia de sus hijos.



- 2) Agregamos ese  $\alpha_1$  en la tabla y le completamos la frecuencia. Ahora también vamos a completar la columna "Status" con un 1 (que quiere decir que ya están ubicados en el árbol).

3	1	BLK	1
4	1	Q	1
5	1	$\alpha_1$	2

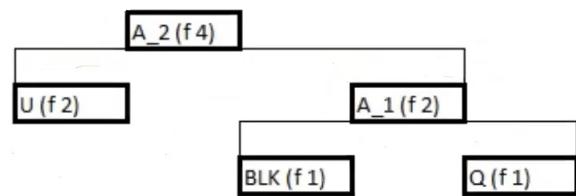
- 3) Ahora, ya analizamos los caracteres Q y BLK que se encuentran en las posiciones 3 y 4, por lo

que tenemos que movernos a la posición 2 (en realidad hay una variable que va marcando qué posición de la tabla hay que analizar y tenemos que actualizar esta variable a 2). Lo que no vamos a actualizar es la frecuencia. Veníamos con una frecuencia de 1 y la vamos a mantener (también tenemos una variable que guarda la frecuencia). Miremos la fila 2:

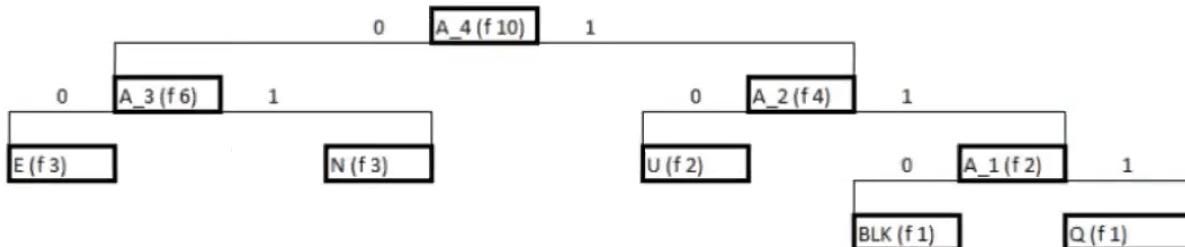
2		U	2
---	--	---	---

En la fila 2 encontramos a la letra U, pero que tiene una frecuencia distinta a la que tiene la variable. Cuando ocurre esto el algoritmo dice que hay que avanzar la frecuencia en 1 y volver a empezar a leer la tabla desde abajo de todo (que ahora la fila de abajo de todo es la 5 porque habíamos agregado a α1). Entonces ahora parte de α1 y empieza a subir por la tabla hasta encontrar a uno que tenga la misma frecuencia. Matchea con la U, entonces vuelve a hacer lo mismo de antes para agregarlos al árbol:

2	1	U	2
3	1	BLK	1
4	1	Q	1
5	1	α1	2
6	1	α2	4



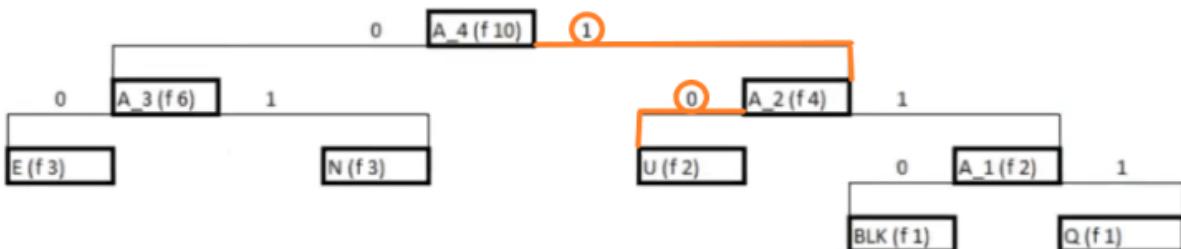
4) Finalmente la tabla y el árbol quedan así:



Status	Char	Frec	Código	*
0	1	E	3	
1	1	N	3	
2	1	U	2	
3	1	BLK	1	
4	1	Q	1	
5	1	α1	2	
6	1	α2	4	
7	1	α3	6	
8	1	α4	10	

→ El último valor será la raíz del árbol.

- 5) Podemos ver que en las aristas del árbol colocamos un cero en las que van hacia la izquierda y un 1 en las que van hacia la derecha. En realidad estos ceros y unos no están guardados en ningún lado, sino que se interpreta que un movimiento a la izquierda es un 0 y un movimiento a la derecha es 1. Estos valores nos van a permitir representar a los caracteres con un código de longitud variable. En el paso siguiente vamos a ver cómo.
- 6) Vamos a incorporar la estructura que nos faltaba: la pila. Vamos a obtener el código de U.



Nos paramos en U (que es una hoja. Todos los caracteres son hojas). Avanzamos al padre y ahí el padre va a saber si estábamos a izquierda o derecha. En este caso la U está a la izquierda entonces pusheamos un 0 en la pila. A su vez a2 asciende a su padre y detectamos que ascendimos por la derecha, entonces ahora se apila un 1. Como a4 no tiene padre, finaliza y el código lo obtendremos haciendo pop de los elementos de la pila. Obtenemos el código "10" (recordar que la pila lee primero el último). De esta manera obtenemos todos los códigos:

Status	Char	Frec	Código	*
0	1	E	3	00
1	1	N	3	01
2	1	U	2	10
3	1	BLK	1	110
4	1	Q	1	111

- 7) Esta información la vamos a usar para formar el archivo comprimido, el cual sigue la siguiente estructura:

- Cabecera (sin codificar):
  - Cantidad de caracteres distintos.
  - Tabla de Huffman original (solamente 2 columnas: char y frecuencia).
- Cuerpo (codificado):
  - Reemplazo cada carácter por su nuevo código.

Es importante remarcar que no se guardan los códigos ni el árbol. Ahora en la descompresión vamos a ver cómo es posible volver al archivo original sin guardar estas cosas.

**Método de descompresión de Huffman:** inicialmente hay que identificar 3 cosas:

- El fin de la cabecera. Nunca guardamos donde termina la cabecera pero lo que sí guardamos es cuál es su tamaño.
- El fin de un código. Recordar que los códigos son de longitud variable. Podemos reconstruir el árbol con la tabla de dos columnas que guardamos en la cabecera.
- El código de relleno (estos son los ceros que se agregan cuando hay que completar un byte). Podemos deducir que todas las hojas son caracteres reales (y todos los caracteres reales

son hojas), por lo cual podemos saber en qué nivel están los caracteres. Sabiendo eso, por ejemplo, si Q es 111 (hoja), sabemos que no puede existir un carácter con código 11, 1 o 111x.

Algoritmo: para cada bit que lee va a ir a la raíz del árbol y va a seguir leyendo bits hasta que llegue a una hoja. Ahí decodifica el carácter y vuelve a empezar. Va hallando los caracteres uno por uno.

## PREGUNTAS DE PARCIAL

### 1. Detalle las estructuras del algoritmo de Huffman.

Las estructuras son:

- Tabla de Huffman: tiene las siguientes columnas: status, carácter, frecuencia, código y puntero.
- Árbol: árbol que se arma con la tabla de Huffman y que nos permitirá obtener los códigos de cada carácter. Estructuralmente se trata de un árbol binario que se construye desde las hojas hasta la raíz y donde todas las hojas son caracteres (y todos los caracteres son hojas). Para obtener los códigos hay que tener en cuenta que las aristas de hijos izquierdos representan un 0 y las de un hijo derecho, 1.
- Pila: estructura de datos que sirve para obtener el código de un carácter. A medida que se recorre el árbol desde una hoja hasta la raíz, se van guardando en la pila ceros y unos (según si se trata de un hijo izquierdo o derecho). Finalmente se leerán los elementos de la pila (LIFO) y el valor obtenido será el código del carácter.

### 2. La longitud variable del código no puede superar los 8 dígitos. V o F.

Falso. En el caso de la tabla ASCII (donde hay 256 caracteres entonces se usa 1 byte para representar a cada carácter) no sucedería que el código supere los 8 dígitos, pero en un archivo que usa el alfabeto UNICODE (donde un carácter ocupa 4 bytes), ahí sí podría ocupar más de 8 dígitos.

# Métodos de Ordenamiento

---

## INTRODUCCIÓN

**Definición:** son técnicas que ordenan un conjunto de valores en un momento determinado. Hablamos de un momento determinado porque las altas o bajas de elementos generan que el conjunto deje de estar ordenado.

Existen muchos algoritmos pero la elección del mejor de ellos depende de varios factores, como por ejemplo, la cantidad de elementos a ordenar, el grado de orden con el que ya vienen dados los elementos, si van a ser ordenados en memoria RAM, en cinta o en disco, etc.

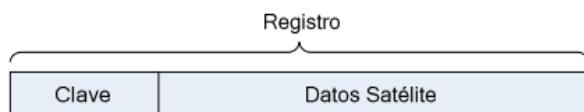
**Objetivo:** dado un conjunto de datos desordenados, el objetivo es devolver un conjunto ordenado de menor a mayor o mayor a menor.

**Operaciones:** la clasificación de los datos requiere al menos de dos operaciones fundamentales

- Comparación de valores (el tipo de dato debe permitir determinar si un valor es menor, mayor o igual a otro)
- Movimiento de los valores a su posición ordenada.

## CLASIFICACIÓN

**Registros:** en la práctica, los valores a ordenar suelen ser parte de una colección de datos llamado registro. Cada registro contiene una key que es el valor a ser ordenado (ya sea un número, letra o un string, o más preciso, la PK de un registro), y el resto del registro contiene los datos satélites.



**Estabilidad:** un ordenamiento se considera estable si mantiene el orden relativo que tenían originalmente los elementos con claves iguales.

**Ejemplo:** si se tienen dos registros A y B con la misma clave (clave duplicada) y, originalmente, A aparece primero que B, entonces el método se considera estable cuando A aparece primero que B en el archivo ordenado.

Una de las ventajas de los métodos estables es que permiten que un conjunto de valores se ordenen usando claves múltiples, por ejemplo, por orden alfabetico del apellido, luego el nombre, luego el documento, etc.

**Métodos In situ:** son los que ordenan una estructura de datos usando el mismo espacio ocupado originalmente o una cantidad extra de memoria, siendo ésta pequeña y constante.

En estos métodos generalmente la entrada es sobreescrita por la salida a medida que se ejecuta el algoritmo, es decir, no se usan estructuras extras para realizar el ordenamiento. Esta característica es fundamental en lo que respecta a la optimización de algoritmos, debido a que el hecho de utilizar la misma estructura disminuye los tiempos de ejecución.

### Clasificación interna y externa:

- Método interno: el ordenamiento es realizado en RAM. Se puede acceder a los elementos fácilmente y de forma aleatoria.

- Método externo: el ordenamiento es realizado en disco o algún otro dispositivo. Acá se debe acceder a los elementos de forma secuencial o al menos en grandes bloques de datos.

## TEORÍA DE LA COMPLEJIDAD/ORDEN DE CRECIMIENTO DE ALGORITMOS

**Teoría de la complejidad computacional:** estudia el costo de ejecución de un algoritmo determinado. Para realizar este estudio se consideran la cantidad de operaciones que realiza una computadora en función del tiempo y la cantidad de elementos a procesar.

Los problemas que tienen una solución con orden de complejidad lineal son los problemas que se resuelven en un tiempo que se relaciona linealmente con su tamaño. Actualmente las computadoras sólo son capaces de resolver (en tiempo y forma) problemas que tengan un orden de complejidad lineal o a lo sumo polinómico.

**Teoría de la computabilidad:** se ocupa de la factibilidad de expresar problemas como algoritmos efectivos sin tomar en cuenta los recursos necesarios para ello.

## MERGE SORT

### Características:

- Es un algoritmo recursivo.
- Requiere de memoria extra proporcional a la cantidad de elementos del vector a ordenar.
- Partes del algoritmo se pueden ejecutar en forma concurrente.

**Método:** el algoritmo básico se basa en fusionar dos vectores de entrada A y B (previamente ordenados) en un único vector de salida C. Como A y B ya están ordenados, lo que hace el algoritmo es sacar un elemento de A y uno de B y preguntar cuál es más chico. Ese lo va a agregar en C. Despues saca otro elemento de ese vector y lo vuelve a comparar con el elemento que sacó del otro vector y así sucesivamente hasta que ambos vectores queden vacíos.

**Algoritmo:** si el vector está desordenado y tiene más de 1 elemento, dividirlo en 2. Si tiene 1 elemento se lo considera ordenado y hay que mergearlo con otro. Luego se mergean los vectores ordenados.

**Tiempo de ejecución:** tiene un tiempo de ejecución  $n \log n$  sin importar cuál sea la entrada.

### Ejemplo:

25	11	9	30	36	18	39	4
0	1	2	3	4	5	6	7
25	11	9	30	36	18	39	4
0	1	2	3	4	5	6	7
25	11	9	30	36	18	39	4
0	1	2	3	4	5	6	7
11	25	9	30	18	36	4	39
0	1	2	3	4	5	6	7
9	11	25	30	4	18	36	39
0	1	2	3	4	5	6	7
4	9	11	18	25	30	36	39
0	1	2	3	4	5	6	7

## INSERTION SORT

### Características:

- La complejidad computacional es variable.
- Es eficiente para ordenar vectores que tienen pocos elementos y están semiordenados.
- Es lento por la cantidad de veces que tiene que intercambiar elementos adyacentes.

**Método:** partimos del vector que hay que ordenar. Vamos a tener un puntero a una posición donde a su izquierda se considera que están los elementos parcialmente ordenados (es decir ordenados entre ellos pero no necesariamente en sus posiciones finales).

El algoritmo comienza analizando el elemento apuntado y lo compara con los elementos ordenados para saber en qué posición insertarlo. A veces el elemento a analizar ya está en la posición correcta (por ser el número mayor), en cuyo caso quedará ya ordenado sin hacer ningún cambio y el puntero se moverá al siguiente elemento a analizar.

Esto nos lleva a dos conclusiones respecto al tiempo de ejecución.

### Tiempo de ejecución:

- Mejor de los casos: en el mejor de los casos el vector ya viene ordenado, entonces cuando se compare al elemento apuntado con los que están a su izquierda siempre va a indicar que el elemento apuntado es el mayor, por lo que no habría que realizar ningún cambio y el puntero avanzaría a la posición siguiente y así, teniendo un tiempo lineal:  $n$ .
- Peor de los casos: en el peor de los casos el vector viene ordenado en el orden opuesto y la complejidad computacional sería muy alta,  $n^2$  - cuadrática.

### Ejemplo:

25	11	30	7	36	18	32
0	1	2	3	4	5	6
25	11	30	7	36	18	32
0	1	2	3	4	5	6
11	25	30	7	36	18	32
0	1	2	3	4	5	6
7	11	25	30	36	18	32
0	1	2	3	4	5	6
7	11	18	25	30	36	32
0	1	2	3	4	5	6
7	11	18	25	30	32	36

## SELECTION SORT

### Características:

- Es muy antiperformante.
- No se puede hacer en forma concurrente.
- Es in situ.

**Método:** selecciona al elemento más chico y lo coloca en su lugar. Luego busca al segundo más pequeño y lo coloca en la posición siguiente y así. Cabe mencionar que los elementos cambian posición en el array, no se usa una estructura adicional.

## QUICK SORT

### Características:

- Es un algoritmo recursivo.
- Se puede hacer en forma concurrente.
- Es in situ, ya que usa solo una pila auxiliar.

**Método:** el algoritmo consiste en elegir un pivot que es un valor aleatorio del array (en general se toma el primer elemento) y “se coloca en el medio del array”. Luego se compara este valor con todos los otros valores del array, dejando los valores que son menores a él a su izquierda (aunque no necesariamente ordenados) y a los mayores a su derecha. El pivot ya quedó en el centro, en su posición final. Después se va a llamar recursivamente a este algoritmo para el array de la izquierda y el de la derecha, quedando el array total ordenado.

### Algoritmo:

1. Elegimos al pivot. Puede ser cualquier elemento del array.



2. Comparamos el pivot con todos los elementos del array, separando los que son mayores a él y los que son menores.



3. Invocamos al algoritmo nuevamente pasándole como parámetro el array de la izquierda y el de la derecha. Esto puede hacerse en forma concurrente.

**Tiempo de ejecución:** es el algoritmo que mejor responde en la mayoría de los casos. Su tiempo de ejecución varía de la siguiente manera:

- Caso promedio:  $n \log n$ . Este caso se da cuando el valor del pivot elegido se encuentra en el centro de la lista.
- Peor caso:  $n^2$ . Este caso se da cuando el valor del pivot elegido se encuentra en alguno de los extremos (porque la lista está ordenada o inversamente ordenada).

## BUBBLE SORT

### Características:

- Es uno de los más lentos y poco recomendables.
- Es el método más elemental.
- Complejidad computacional variable.

**Método:** consiste en hacer  $N-1$  pasadas sobre los datos, donde en cada paso, los elementos adyacentes son comparados e intercambiados si es necesario. Durante la primera pasada, el elemento más grande va “burbujeando” a la última posición del vector.

En general, luego de K pasadas por el vector, los últimos K elementos del vector se consideran bien ordenados por lo que no se los tendrá en cuenta. A veces no hace falta hacer N-1 pasadas sobre el vector para que éste quede ordenado.

### Tiempo de ejecución:

- Peor caso: es de orden cuadrático  $n^2$  y ocurre cuando el vector viene en orden inverso.
- Mejor caso: es de orden lineal y ocurre cuando el vector está previamente ordenado.

## SHELL SORT

### Características:

- Mejora notablemente el método de inserción (se hace una modificación ganando velocidad al intercambiar elementos que estén muy distantes).
- Se puede hacer en paralelo.

**Método:** la idea es mejorar el algoritmo de inserción. Recordemos que en inserción el peor de los casos era cuando el vector venía ordenado en forma inversa. Este algoritmo lo que hace es primero “ordenar” a los elementos (es decir, darles un cierto “preorden” que si bien -obviamente- no ordena al vector completamente, evita que se produzca el peor caso de ordenamiento).

Entonces, la idea es aplicar un algoritmo de pre-orden que deje al vector listo para luego aplicar el algoritmo de inserción. Para eso va a hacer una división del vector en partes que no son contiguas (esto se llama “distancia”. Puede usar distancias de distintos tamaños).

Es lo mismo que un for pero en vez de avanzar de a 1 elemento avanza de a 5. Entonces, lo que va a hacer es ordenar estos elementos que se encuentran a la misma distancia. Estos pueden ordenarse en paralelo. Con esto lo que vamos a lograr es que en cada conjunto (y por ende en la totalidad del vector), los valores más grandes vayan tendiendo a la derecha.

### Ejemplo:

Ordenado con distancia de 5														
12	15	96	33	21	25	93	65	23	45	29	41	39	17	59
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	15	35	17	21	25	41	65	23	45	29	93	96	33	59

Después va a repetir lo mismo pero con una distancia más pequeña:

Ordenado con distancia de 3														
12	15	35	17	21	25	41	65	23	45	29	93	96	33	59
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	15	23	17	21	25	41	29	35	45	33	59	96	65	93

Y ahora lo ordenamos con distancia 1, que sería lo mismo que el algoritmo de inserción.

**Tiempo de ejecución:** si bien no hay un consenso sobre la eficiencia del Shell Sort, se considera que este varía entre  $O(n^{3/2})$  y  $O(n^{7/6})$ . Es uno de los mejores en términos de procesamiento.

## HEAP SORT

### Características:

- No requiere espacio de memoria adicional dado que es un algoritmo que se ejecuta in situ.

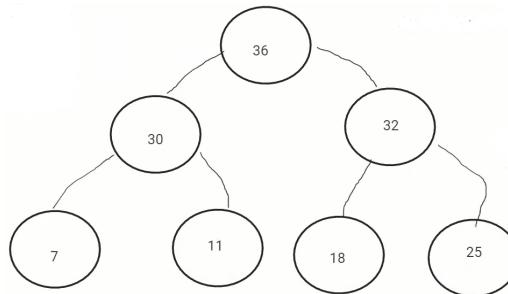
- Se implementa sobre un árbol binario.

**Método:** este algoritmo, al igual que los otros, trabaja con un vector pero este vector va a estar representando un árbol binario. Se trata de un árbol cargado por niveles y de orden parcial.

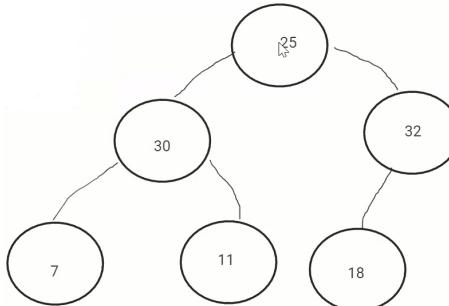
- Árbol cargado por niveles: se carga comenzando por el nivel 0 (raíz) y se sigue avanzando por los niveles de izquierda a derecha hasta finalizar. No hace falta que el árbol esté completo.
- Árbol de orden parcial: cada nodo satisface la “condición de montículo” que establece que la clave de cada nodo debe ser mayor o igual a las claves de sus hijos, si es que tiene. Esto implica que la clave más grande está en la raíz.

#### Algoritmo:

1. Cargar el árbol (1 sola vez): partimos del vector 25, 11, 30, 7, 36, 18, 32 y se construye el árbol leyendo los valores del vector uno a uno y haciendo las modificaciones necesarias.



2. Ordenamiento (n-1 veces) siendo n la cantidad de claves a ordenar: por las características del árbol, el algoritmo ya sabe que el mayor valor va a estar en la raíz, por lo que ya puede colocar este valor en su posición final. Ahora lo que va a hacer es mover el último valor que fue cargado (o sea la hoja más a la derecha, en este caso el 25) a la raíz.



Lo que pasa es que ahora dejó de ser un árbol de orden parcial. Entonces ahora vamos a comparar el 25 con sus hijos y hacer las modificaciones para que sea un árbol de orden parcial nuevamente. Vamos a intercambiar al 25 por el 32.

Ahora volvemos a hacer lo mismo que antes. Sacamos a la nueva raíz (32), la ordenamos en el vector en su posición final y subimos el 18. Y vamos a seguir haciendo esto hasta terminar.

**Tiempo de ejecución:**  $\log_2(n+1)*n$

#### ORDEN DE COMPLEJIDAD DE LOS ALGORITMOS

Nombre	Mejor caso	Caso medio	Peor caso	Estable	Comentarios
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Si	El más lento de todos. Uso pedagógico.
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Si	Apto si queremos que consumir siempre la misma cantidad de tiempo.
Insertion Sort	$O(n)$	$O(n)$	$O(n^2)$	Si	Conveniente cuando el array está casi ordenado.
Shell Sort	$O(n^{5/4})$	$O(n^{3/2})$	$O(n^2)$	No	Dependiente de la secuencia de incrementos.
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Si	Adecuado para trabajos en paralelo.
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	No	El método acotado en el tiempo muy utilizado para grandes volúmenes de datos
QuickSort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	No	El más rápido en la práctica. Implementado en gran cantidad de sistemas.

## PREGUNTAS DE PARCIAL

- 1. El algoritmo Quicksort sobre un conjunto de datos desordenados tiene una complejidad menor que si se ejecuta sobre datos ordenados. V o F.**

Verdadero. En el mejor de los casos el algoritmo tiene un orden de complejidad de  $n \log n$ . Esto se da cuando el valor del pivot se encuentra en el medio. En el peor de los casos el orden de complejidad pasa a  $n^2$  y se da cuando el pivot se encuentra en los extremos o cuando la lista está casi ordenada.

- 2. El algoritmo de quicksort no puede ser recursivo. V o F.**

Falso. Una vez que se selecciona el pivot el algoritmo compara todos los valores del array con este, separando a su izquierda los que son menores a él y a su derecha los que son mayores a él, aunque estos no están necesariamente ordenados. Ahí es donde entra en juego la recursividad. Se aplicará el mismo algoritmo a estos arrays (que, además, puede hacerse en forma concurrente, disminuyendo el tiempo de procesamiento).

- 3. Heap sort está implementado sobre árbol binario. V o F.**

Verdadero. El algoritmo trabaja con un vector que representa un árbol binario que tiene 2 características: es cargado por niveles y es de orden parcial.

- 4. Mencionar un algoritmo de ordenamiento iterativo y uno recursivo.**

El quicksort es un algoritmo de ordenamiento recursivo in situ que por medio de dos variables  $i$  y  $j$  va recorriendo el array desde ambos extremos hasta dejar en el medio al elemento llamado "pivot", el cual se considera en su posición final. A su izquierda se encontrarán los valores menores a él y a su derecha los mayores, no necesariamente ordenados. Luego se aplicará recursivamente este algoritmo sobre cada array (pudiendo realizarse en forma concurrente) hasta que el vector quede completamente ordenado.

El selection sort es un algoritmo iterativo muy antiperformante que lo que hace es seleccionar al elemento más chico y colocarlo en su posición final (mediante intercambios) hasta ordenar al vector en su totalidad.

# Hashing

---

## ACCESO A DATOS

**Definición:** anteriormente mencionamos al ABB que es un árbol armado de forma tal de poder hacer búsquedas sobre esa misma estructura sin necesidad de algo adicional. En acceso a los datos ese concepto va a cambiar porque por un lado vamos a tener los datos originales y después vamos a tener estructuras a parte para poder acceder a esos datos de una forma más eficiente.

- Elementos que va a tener la estructura:
  - Claves: que son las claves por las que se está ordenando.
  - Punteros: la dirección de dónde comienza un registro de una base de datos.
- Tipos de accesos:
  - Hashing Dinámico y Estático
    - Permite claves únicas y búsquedas por un sólo valor.
    - No permite búsquedas por rangos.
  - Árbol B y Árbol B+:
    - Permite claves duplicadas.
    - Permite búsquedas por rangos.

## HASHING ESTÁTICO (ABIERTO)

**Implementación:** tenemos un conjunto de datos, cada uno identificado por una clave. Vamos a crear una tabla de hash de tamaño estático. Esta tabla va a tener la siguiente estructura:

- Clave.
- Puntero.
- Baja lógica (bit).

Para el ejemplo supongamos una tabla de hash de 100 entradas (0 a 99).

**Función de hash:** es una función que toma una clave y genera una posición válida de la matriz para ubicarla. Supongamos que la función recibe las siguientes claves y devuelve las siguientes posiciones:

- Clave: 1 → Posición 45
- Clave: 2 → Posición 39
- Clave: i → Posición 12
- Clave: j → Posición 39

	Clave	Puntero	
0			
1			
2			
12	clave i		
39	clave 2		
45	clave 1		
51	clave j		

¿Qué pasó? para claves distintas nos devolvió la misma posición. Esto se llama “colisión de hash” y para solucionarlo hay que aplicar una función de re-hash, que, a diferencia de la función de hash, va a recibir la posición, no una clave, en este caso va a recibir el 39. La función de re-hash nos va a devolver una nueva posición, supongamos que devuelve la 51.

**Búsqueda de una clave:** supongamos que queremos buscar la clave i.

- Le aplicamos la función de hash y obtenemos la posición 12.
- Comparamos la clave que hallamos en esa posición con la clave que estamos buscando.
- Si hay coincidencia, el puntero de la tabla va a apuntar a donde comienzan los datos. En caso de no coincidir esto nos indica que ha ocurrido una colisión (que sería el caso de la clave j). En este caso lo que hay que hacer es aplicar la función de re-hash y volver a hacer esta comparación de la clave hasta que encontremos la clave que estamos buscando.

**Performance:** está dada por la cantidad de veces máxima que tenemos que aplicar la función de re-hash. Por ejemplo, en el caso anterior tuvimos que aplicar la función de re-hash 1 vez pero tuvimos que hacer 2 lecturas a la tabla, por lo que la cantidad de lecturas a la tabla siempre va a ser una más que la cantidad de veces que se tiene que aplicar la función de re-hash.

## FUNCIÓN DE RE-HASH

**Dispersión:** hashing quiere decir dispersión. Lo que esperamos de una función de hash para que sea eficiente es que genere dispersión y ocupe todas las posiciones vacías, y no que genere valores dentro de un mismo rango limitado. Acá también entra en juego el tamaño de la tabla de hash.

**Límite de una función de re-hash:** supongamos que recibe la posición 39 y devuelve las siguientes posiciones al seguir re-hasheando: 39, 51, ..., 39. ¿Qué pasó? volvió al 39. Supongamos que tardó 20 iteraciones en volver al 39, entonces vamos a decir que hay 80 posiciones de esta matriz (100-20) que nunca nos va a devolver la función de re-hash. Con lo cual nos encontramos con un problema funcional y de performance porque nos puede decir que la tabla está llena cuando no es así.

Lo ideal sería lograr que la tabla de re-hash vuelva a la posición inicial en la iteración 100, habiendo pasado por todas las posiciones de la tabla.

## FUNCIÓN DE HASH Y RE-HASH

**¿Cómo lograr que la función de re-hash pase por todas las posiciones de la tabla?:** inicialmente el caso más sencillo es pensar que la función de re-hash sume 1 a la posición que recibe hasta llegar al final de la tabla.

**Tamaño de la tabla:** supongamos que tenemos 300 claves de tipo de dato entero. ¿Cuál tiene que ser el tamaño de la tabla de hash? el primer número primo (o el segundo o tercero según la cantidad de dispersión que le queramos dar) mayor a la cantidad de claves. En nuestro caso el tamaño será 307 (entre 0 y 306).

Ahora vamos a ver el tipo de función que tiene que ser la función de hash. Hay dos. ¿Cómo sabemos que estas funciones generan dispersión? por pruebas de fuerza bruta.

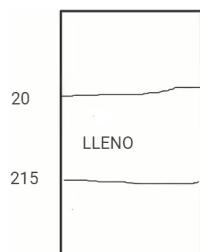
- Dobles: se toma la clave numérica y se la parte en dos. Se pasan ambas partes a binario y se efectúa una “suma or exclusivo” (que suma 1 cuando los bits son distintos). Luego pasamos este número a decimal y tenemos que ver si esta posición está dentro del rango de posiciones de la tabla. Si no lo está tenemos que aplicar la función mod(307).
- Cuadrado medio: elevamos el valor recibido al cuadrado. Ahora, tenemos que tomar los k dígitos del medio de este número (siendo k la cantidad de dígitos que tiene 307, que es el tamaño de la tabla de hash). Como 307 tiene 3 dígitos, tenemos que tomar los 3 valores del medio del número hallado. De ser necesario se puede completar con ceros a la izquierda. Si el valor obtenido no está dentro del rango, de nuevo, tenemos que aplicar la función mod(307).

Ahora analicemos el caso de la función de re-hash de cuadrado medio. Las siguientes claves nos otorgan la misma posición (colisionan):

- Clave 2869 → 4
- Clave 2 → 4

¿Qué es, entonces, lo que tiene que lograr una función de re-hash? Que partiendo de 4 y ejecutando la función 307 veces recorra las 307 posiciones de la matriz. Para asegurar esto tenemos que hacer lo siguiente: a la posición 4 que obtuvimos le vamos a sumar un número primo (1) y a ese resultado le tenemos que aplicar el mod(307).

Pero, ¿qué nos va a generar usar el número primo 1? En algún momento va a pasar que tengamos, por ejemplo, de la posición 20 a la 215 lleno. Entonces, si nos llega a tocar la posición 30 vamos a tener que aplicar la función de re-hash e ir sumando 1 hasta llegar a 216.



A este amontonamiento se lo conoce como "clustering". Para evitarlo lo que hay que hacer es evitar que el número primo que se suma sea 1. Tiene que ser un número primo relativamente grande.

**Baja de una clave:** supongamos que queremos dar de baja una clave. El primer paso es buscarla. En el caso básico borramos esa clave y ya está. Pero en el caso en que nuestra clave formara parte de un encadenamiento (es decir de la colisión de otros hash), el borrar esta clave nos va a romper el encadenamiento y no nos va a permitir llegar a las otras claves porque en su lugar va a haber un NULL, entonces el algoritmo lo va a interpretar como la clave no existe en vez de compararla con la clave que está buscando y aplicar la función de re-hash.

Para solucionar esto vamos a usar la tercera columna de la tabla "baja lógica". En hashing estático todas las bajas son lógicas.

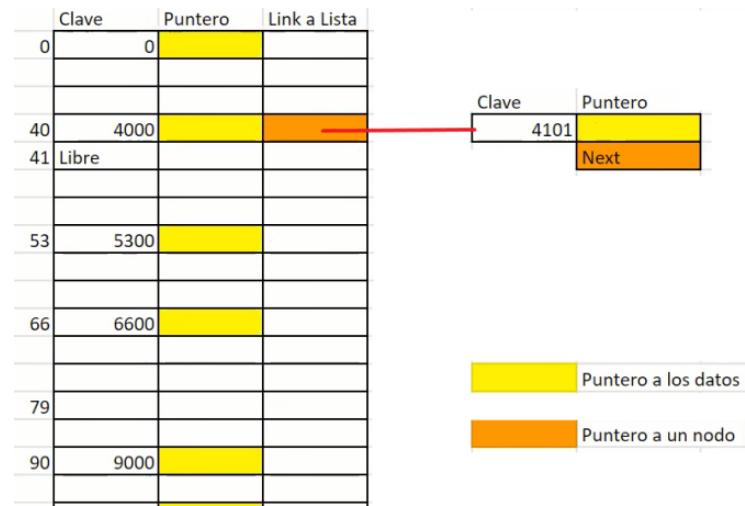
## HASHING DINÁMICO (CHAINING)

**Implementación:** tenemos un conjunto de datos, cada uno identificado por una clave. Inicialmente vamos a crear una tabla de hash de un tamaño determinado (podemos usar el mismo criterio que en hashing estático y usar el primer número primo mayor a la cantidad de claves). A diferencia del método anterior, acá no vamos a tener función de re-hash y todas las bajas van a ser físicas. Esta tabla va a tener la siguiente estructura:

- Clave.
- Puntero.
- Link a lista: en esta lista vamos a tener todas las claves que colisionaron.

Clave	Puntero
	Next

Por ejemplo:



## HASHING DENTRO DE ÍNDICES

**Implementación:** ¿cómo resolvería hashing la siguiente consulta?

```
select *
from Personas
where idPersona=40
```

Se quiere buscar la clave 40. Entonces se va a aplicar la función de hash, lo cual va a devolver una posición de la tabla. Las bases de datos usan hashing dinámico, por lo que se irá a buscar la clave a la tabla y si no se encuentra se buscará en la lista.

La eficiencia de esto está dada por la cantidad de nodos ( $n$ ) en esa lista. En el peor de los casos se encontrará la clave en  $n + 1$  lecturas (se suma 1 por la primera lectura a la tabla de hash).

Ahora analicemos la siguiente consulta:

```
select *
from Personas
where idPersona between 40 and 41
```

En este caso queremos buscar por un rango. Acá sólo serían dos claves, la 40 y la 41. Esto técnicamente se resuelve buscando una por una, primero la clave 40 como vimos arriba y, una vez que termina, la clave 41. Pero esto es ineficiente dado que hay que hacer todo el procedimiento para todos los valores del rango. Es peor que un full scan. Por eso decimos que hashing no sirve para búsquedas por rango.

## PREGUNTAS DE PARCIAL

- 1. El método de “dobles” permite resolver el problema de colisiones entre claves distintas. V o F.**

Falso. El método de dobles es simplemente un método para implementar una función de hashing. El método consiste en dividir la clave en dos partes, luego pasar cada parte a binario y realizar la operación “suma or exclusivo”. Luego habrá que pasar el resultado de la suma a decimal y si se pasará del rango aplicar la función mod().

- 2. El método del “cuadrado medio” permite resolver el problema de colisiones entre claves distintas. V o F.**

Falso. El método del cuadrado medio es simplemente un método para implementar una función de hash. El método consiste en elevar la clave al cuadrado y luego quedarse con sus  $n$  dígitos del medio (siendo  $n$  la cantidad de dígitos del tamaño de la tabla de hash). Si el valor obtenido excediera el tamaño de la tabla se deberá aplicar la función módulo.

Puede haber claves distintas que, al aplicarse esta función, se obtenga el mismo valor, lo que generaría una colisión de hash que deberá resolverse de alguna otra manera (rehashing, chaining, etc.).

- 3. No existe función inversa para Hash. V o F.**

Verdadero. Al aplicarse la función de hash sobre una cadena es imposible “revertir” la operación y obtener la cadena original.

- 4. Al producirse una colisión en direccionamiento abierto se debe agregar un nuevo nodo en la lista para esa posición. V o F.**

Falso. Eso es así para hashing dinámico. En hashing estático/abierto, al producirse una colisión se aplica la función de re-hash las veces que sean necesarias hasta encontrar una posición vacía de la tabla.

- 5. ¿Qué es una colisión?**

Una colisión es un fenómeno que ocurre cuando una función de hash devuelve el mismo valor para dos entradas diferentes. Por ejemplo, si

$\text{Hash}(\text{clave } i) = n$

$\text{Hash}(\text{clave } j) = n$

Estamos ante una colisión. El tratamiento de las colisiones difiere según se trate de hashing estático o dinámico. En hashing dinámico se aplica la función de re-hash las veces que sean necesarias hasta obtener una posición de la tabla que esté libre. En hashing dinámico simplemente se agrega un nodo a la lista de claves que colisionaron en esa posición.

# Árbol B

## ÁRBOLES M-ARIOS

**Árbol m-ario:** árbol de grado mayor a 2.

**Introducción:** si se tiene un conjunto de datos muy grande, como puede ser una DB, no podemos colocarlo en la RAM, por lo cual es imposible implementar un árbol de búsqueda. La tabla va a estar en disco y vamos a utilizar un índice que va a tener en su estructura una estructura de tipo árbol. Ese árbol va a variar en su grado entre 50 y 2000 en base al tamaño de las claves y de la página del disco.

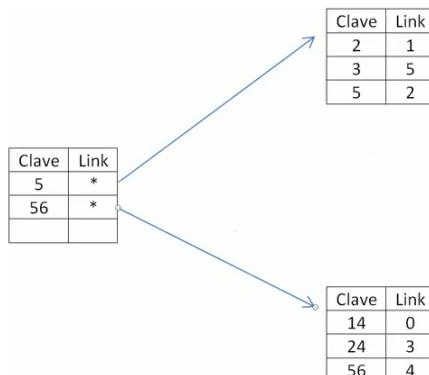
## ÁRBOL B

**Definición:** es un tipo de árbol m-ario destinado a la creación de índices físicos para el acceso a la información. Este árbol es físicamente y lógicamente independiente de la tabla.

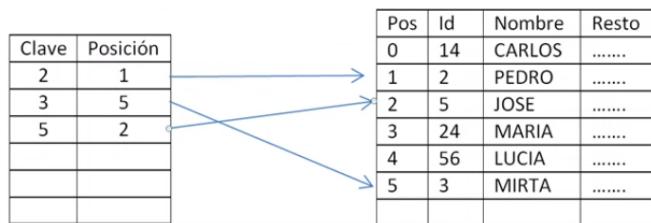
El objetivo principal es minimizar las operaciones de E/S hacia el disco. Al imponer la condición de balance, el árbol es restringido de manera tal que se garantice que la búsqueda, la inserción y la eliminación sean todos de tiempo log n.

**Nodos del árbol:** el árbol B es uno de los pocos árboles con dos tipos de nodos diferentes:

- El nodo raíz o rama: tiene dos cosas:
  - Clave: van los valores de las claves ordenados de menor a mayor.
  - Puntero/Link: apunta al nodo que contiene claves menores o iguales que ella.



- El nodo hoja: tiene dos cosas:
  - Clave: van los valores de las claves ordenados de menor a mayor.
  - Puntero/Posición: contiene la posición relativa de los datos correspondientes a esa clave (el rowId).



## OPERACIONES EN ÁRBOL B

**Búsqueda:** buscar en un árbol B es muy parecido a buscar en un ABB, excepto que en vez de hacer una decisión binaria en cada nodo, hacemos una decisión en base al número de hijos del nodo.

**Inserción:** para insertar un elemento x, comenzamos en la raíz y realizamos una búsqueda para él. Asumiendo que el elemento no está previamente en el árbol, la búsqueda sin éxito terminará en un nodo hoja, donde se lo va a insertar.

**Split:** si ocurre que cuando se llega a la hoja no hay espacio para insertar el nodo se produce lo que se denomina “split” que es un proceso que divide el nodo en dos dejando la mitad de elementos en cada uno respetando el orden de menor a mayor, quedando la mitad de los elementos más chicos en un nodo y la mitad de los elementos más grandes en el otro.

**Eliminación:** para eliminar un elemento x, comenzamos en la raíz y realizamos una búsqueda para él. Si el elemento existe se llegará a la hoja donde está y se borrará, sino se dirá que no existe.

**Fusión:** si ocurre que cuando se elimina el elemento x el nodo queda vacío, debe eliminarse el nodo, lo que puede generar una baja potencial en todos los antecesores de dicho nodo.

## CASO PRÁCTICO

**Situación:** dados los siguientes números de claves posibles realizaremos el proceso de inserción, búsqueda y eliminación de elementos del árbol.

22 – 3 – 5 – 11 – 23 – 54 – 10 – 14 – 15 – 7 – 3 – 9

A modo práctico lo realizaremos con un árbol de grado 3 para ver las diferentes situaciones que pueden ocurrir.

Como es de grado 3, lo primero que hacemos es tomar 3 claves: 22 – 3 – 5, ordenarlas y armar la siguiente estructura.

Nodo hoja	
Clave	Link
3	1
5	2
22	0

Pos	Id	Resto
0	22	.....
1	3	.....
2	5	.....

Ahora tomamos otra clave, la 11 y se va a producir el primer split porque el nodo no tiene más espacio. 3 y 5 van a quedar por un lado y 11 y 22 por otro. Por partir este nodo también vamos a tener que crearle un parent (nodo raíz). El parent va a tener como claves las claves más grandes de cada hijo.

Clave	Link
3	1
5	2

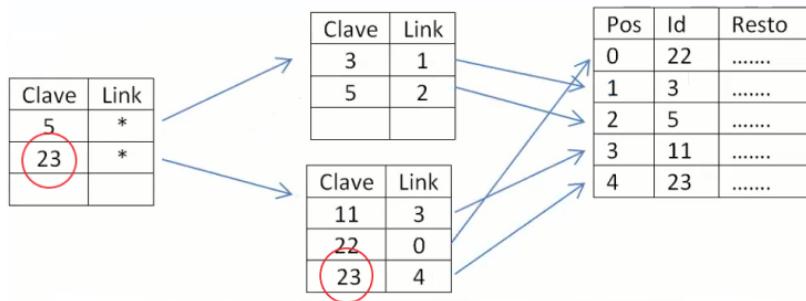
  

Clave	Link
11	3
22	0

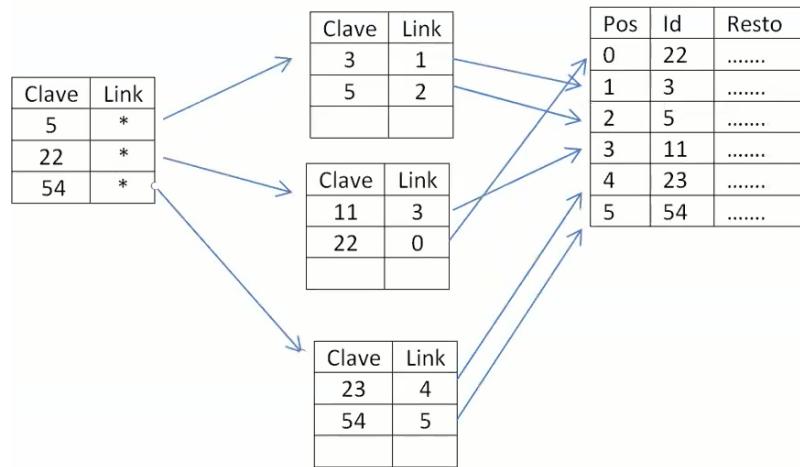
  

Pos	Id	Resto
0	22	.....
1	3	.....
2	5	.....
3	11	.....

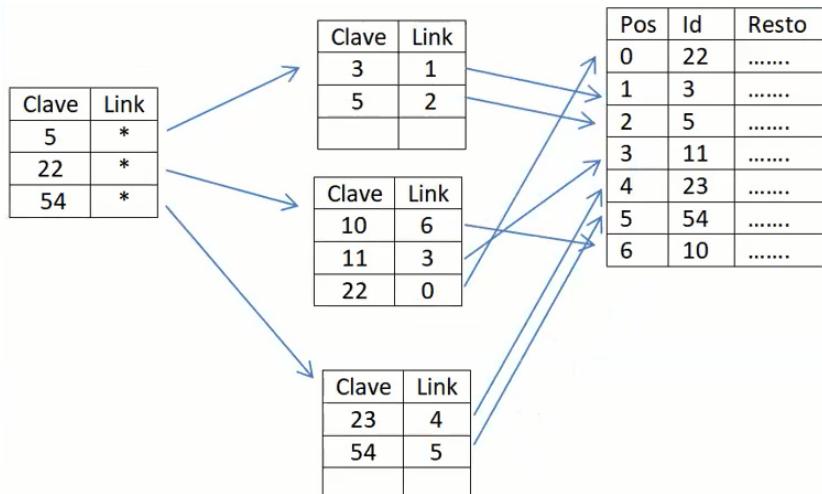
Ahora llega la clave 23, que corresponde ubicarla en el nodo donde está el 22, pero ahora el mayor valor de ese nodo cambió, por lo que el parent deberá cambiar también.



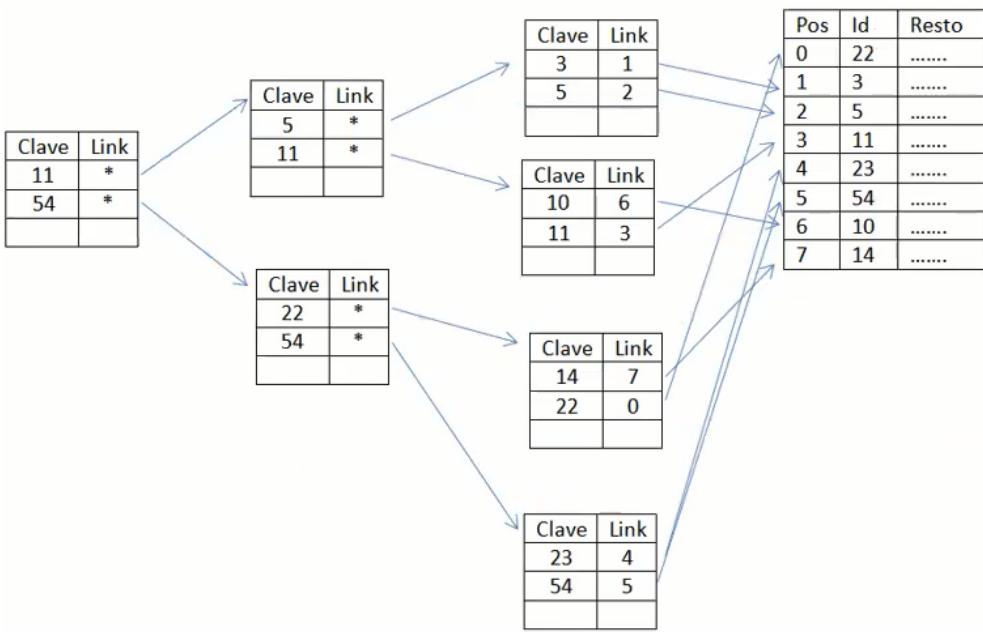
Ahora llega la clave 54 y si bien tenemos lugar en el nodo de arriba, no corresponde meterla ahí. Entonces se va a producir otro split en el nodo de abajo y van a quedar el 11 y el 22 por un lado y el 23 y el 54 por otro. También se vuelven a cambiar las claves del padre.



Ahora llega el 10 y se ubica en el nodo del medio, desplazando a las otras claves:



Ahora llega el 14 que se ubica entre el 11 y el 22, por lo que hay otro split. El problema es que ahora el padre también está lleno. Pero el padre sólo puede referenciar a 3 nodos, no 4. Entonces el padre también va a sufrir un split.

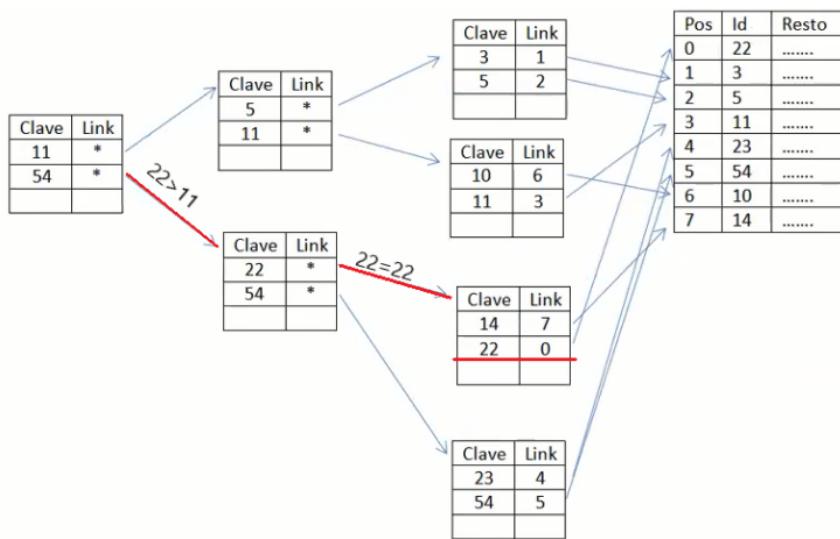


Y así sigue sucesivamente.

## OPERACIONES EN ÁRBOL B

**Búsqueda de una clave única:** búsqueda de la clave 22.

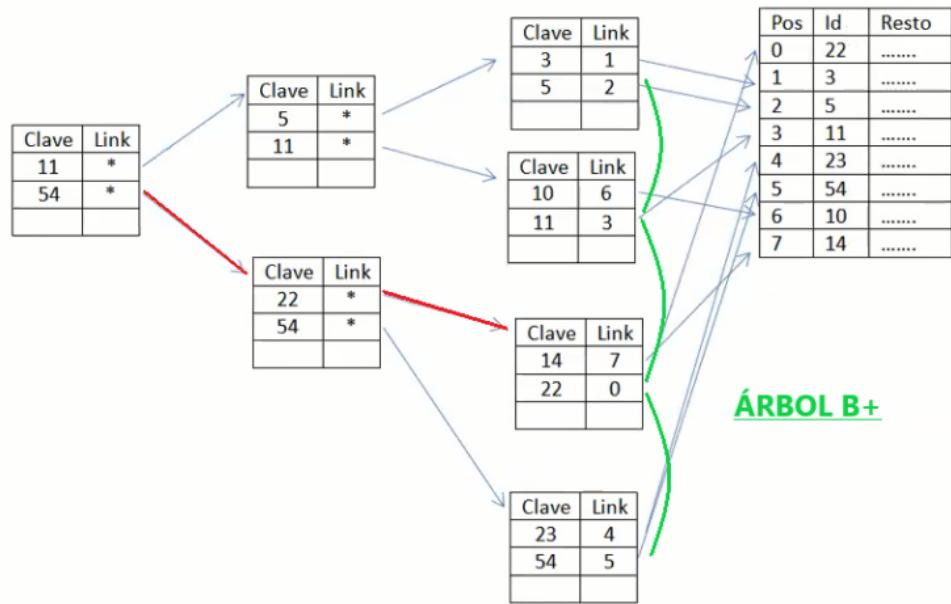
```
select *
from Personas
where idPersona=22
```



## ÁRBOL B+

**Búsqueda de un rango de claves:**

```
select *
from Personas
where idPersona between 20 and 41
```



En el nodo solamente tenemos una clave dentro del rango: la 22. Pero la 23 que también la necesitamos está en otro nodo. Acá entra en juego el concepto de árbol B+. Este árbol en realidad es un árbol B+ y tiene unos punteros extra entre los hermanos, que nos permite pasar de un nodo a los otros (ver flechas verdes). Estos punteros solamente están presentes en los nodos hojas. Además, en un árbol B+ toda la información se encuentra almacenada en las hojas.

Es importante remarcar que en algunos casos los índices basados en árboles B+ le pueden ahorrar al motor de base de datos la tarea de ORDER BY. Supongamos que se quieren leer los IDs del 20 al 41: el índice ya los va a devolver en orden porque así es como los lee, entonces el motor ya los recibe ordenados y no tiene que ejecutar un algoritmo de ordenamiento.

## OTROS CONCEPTOS

**Dispersión (solamente para índices de claves duplicadas):** es una estadística de cuántos valores distintos hay. En base a eso, si una tabla tiene varios índices, va a elegir al que tenga más dispersión para resolver una consulta sql.

## ÍNDICES

**Definición:** un índice es una estructura en disco o en memoria asociada a una tabla o vista que acelera la búsqueda de filas de la tabla o vista. Por lo general, los índices están implementados sobre un árbol B+, donde se almacenan las claves (que pueden ser generadas a partir de una o varias columnas de la tabla) para que el motor de base de datos pueda buscar de forma rápida y eficiente la fila o un rango de filas asociadas a los valores de cada clave.

Se puede prescindir de un índice sobre todo cuando no haya una búsqueda que se haga de manera recurrente sobre esa columna, dado que no tendría sentido armar la estructura y sólo empeoraría la performance del motor. Los índices deben estar en su justa medida dado que es muy costoso (en términos de performance) mantenerlos actualizados.

## PREGUNTAS DE PARCIAL

- 1. El árbol B es más performante que el árbol B+ en el manejo de consultas por clave puntual. V o F.**

Falso.

- 2. Explique el objetivo de los índices y su tipo.**

El objetivo de los índices es acceder de forma más rápida a los datos almacenados en las tablas. Los índices son estructuras opcionales (generalmente implementados con un árbol B o árbol B+) asociadas a estas que son independientes física y lógicamente de los datos almacenados. Una tabla puede tener muchos índices sobre distintas columnas, aunque no es recomendable dado que empeora la performance general del motor. Los índices deben existir en su justa medida, dado que, como deben ser actualizados ante cualquier operación que cambie la estructura del árbol B asociado al índice (claramente es muy costoso actualizar los índices ante cualquier insert/update/delete).

- 3. El árbol B+ permite búsquedas por rango con claves duplicadas. V o F.**

Verdadero.

- 4. El árbol B es más performante para búsquedas por rango que el árbol B+. V o F.**

Falso. El árbol B+ tiene punteros extra entre los hermanos, que permite pasar de un nodo hoja a los otros, lo que hace que la búsqueda por rangos sea más performante.

- 5. En un árbol B todas las hojas se encuentran en el mismo nivel.**

Verdadero. Un árbol B se mantiene balanceado porque requiere que todos los nodos hoja se encuentren a la misma altura.

- 6. Explique el acceso a datos por árbol B+ y mencione de qué modo resuelve las búsquedas por rango.**

El acceso a datos se hace partiendo del nodo raíz y comparando la clave que se está buscando con las claves que tiene ese nodo. Sabiendo que el puntero asociado apunta a un nodo que contiene claves menores o iguales, se llega hasta el nodo hoja que contiene (si la clave existe) a la clave. Este nodo hoja tendrá un puntero a los datos.

En el caso de una búsqueda por rangos es de vital importancia el uso de árboles B+ dado que los nodos hojas tienen unos punteros extra entre los hermanos que permiten saltar entre nodos.

- 7. ¿Qué es un índice? De un ejemplo de cuando podría prescindir de uno.**

Un índice es una estructura en disco o en memoria asociada a una tabla o vista que acelera la búsqueda de filas de la tabla o vista. Por lo general, los índices están implementados sobre un árbol B+, donde se almacenan las claves (que pueden ser generadas a partir de una o varias columnas de la tabla) para que el motor de base de datos pueda buscar de forma rápida y eficiente la fila o un rango de filas asociadas a los valores de cada clave.

Se puede prescindir de un índice sobre todo cuando no haya una búsqueda que se haga de manera recurrente sobre esa columna, dado que no tendría sentido armar la estructura y sólo empeoraría la performance del motor. Los índices deben estar en su justa medida dado que es muy costoso (en términos de performance) mantenerlos actualizados.

# Teoría de Base de Datos

## INTRODUCCIÓN

**Diferencia entre un repositorio de base de datos y una base de datos:** una DB nos da una determinada cantidad de funcionalidades, entre ellas DBMS, transacciones, concurrencias, que un repositorio no ofrece y tendríamos que programarlo nosotros.

**Programas/Procesos:** controlan todas las actividades del DBMS.

- **DBMS:** es el motor de base de datos y es el que otorga todas las funcionalidades. Tiene que cumplir un conjunto de propiedades:
  - Atomicidad: garantiza que una operación se ejecute en su totalidad o que no se ejecute. Logra que varios comandos sean considerados como un solo comando o transacción. Ante un error se vuelve a la última instancia segura conocida y es como si la operación nunca se hubiera ejecutado.
  - Consistencia: tiene que ver con la integridad referencial. El motor debe pasar de un estado consistente a otro consistente (por ejemplo, no se pueden tener FKs que apunten a cosas que no existen). Si el motor ve que algo va a quedar inconsistente, no lo ejecuta.
  - Integridad:
  - Aislamiento: consiste en aislar transacciones, secciones o instancias. Un motor es concurrente, pero el motor es uno solo, la base de datos es una sola. Muchas personas quieren acceder a estos datos al mismo tiempo, por lo que el motor separa en sesiones, permisos y se setean niveles de aislamientos para que los usuarios y sesiones no se pisen entre sí.
  - Durabilidad: es la posibilidad que tiene un motor de persistir la información, mantenerla guardada y poder recuperarla (backup).

Si un motor cumple con estas propiedades (denominadas ACID), se dice que cumple con las reglas básicas para ser un motor.

### Datos:

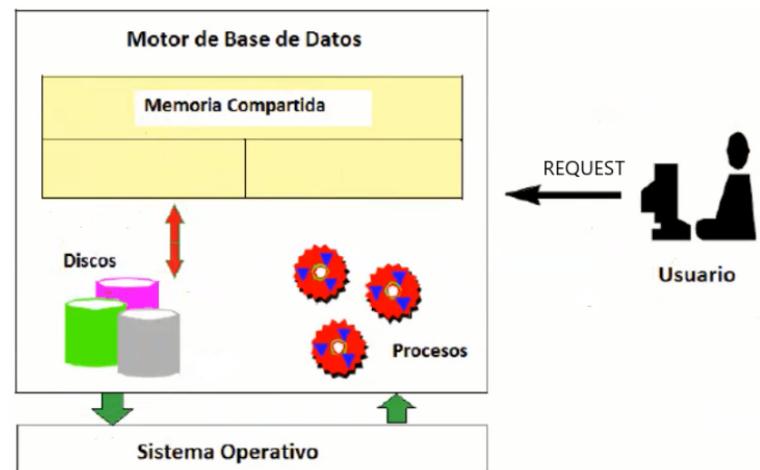
- Integrados
- Compartidos (concurrencia)

### Tecnología:

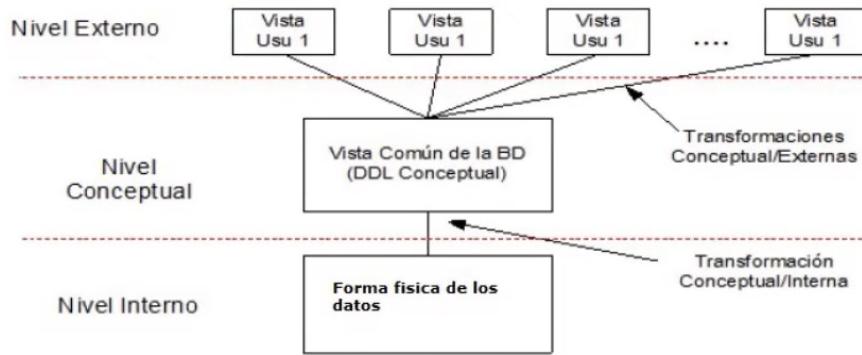
- HW.
- SO.

### Usuarios:

- Programador de aplicaciones.
- DBA (DB Administrator).
- Usuarios finales.
- Otros (tester, arquitecto, etc.).



## ARQUITECTURA ANSI SPARC



**Nivel interno/físico:** se describe cómo los datos están almacenados físicamente y cómo son accedidos.

**Nivel conceptual:** puente entre ambos niveles. Es la estructura lógica global que representa las estructuras de datos y las relaciones entre ellos. Hay una única vista en este nivel y se lo define con el DDL (data definition language).

**Nivel externo:** es la forma en la cual los usuarios ven los datos. Está conformado por las múltiples vistas de los datos almacenados en la base de datos. Estas vistas están adecuadas a las necesidades de información de cada usuario. Se define con el DML (data manipulation language).

## FUNCIONALIDADES DEL DBMS

**1. Diccionario de datos/metadata:** son los datos de los datos. Estamos hablando de las tablas y vistas del sistema. Para acceder a ellas usamos sys.xxx

**2. Seguridad/DCL:** estamos hablando de qué se puede y qué no se puede hacer. El encargado de la seguridad es el Data Control Language, que es el encargado de dar y quitar permisos.

GRANT/REVOKE USER/ROL PERMISO OBJETO

### Tipos de permisos:

- Conexión: es el permiso básico. Si no tengo permiso para conectarme a la base de datos no puedo hacer nada.
- Permisos para dar permisos
- Sobre tablas: select, insert, delete, alter, drop, truncate
- Sobre vistas: select, insert, delete, alter, drop,
- Sobre SP/FX: exec, alter, drop
- Sobre índices y triggers: alter, drop

**3. Integridad:** este concepto refiere a ciertas reglas de aplicación (que muchas veces no coinciden con las reglas de negocio) que los datos deben cumplir. Hablamos de tres tipos de integridad:

- Integridad de las entidades (Primary Keys): la clave primaria de una tabla debe tener un valor único (no nulo) para cada fila de la tabla que permita identificarla únicamente. Si no fuera así, la base de datos perdería su integridad.
- Integridad referencial (Foreign Keys): busca asegurar la consistencia de las relaciones entre tablas. La integridad referencial determina que todos los valores que toma una clave foránea deben ser valores nulos o valores que existan como clave primaria de la tabla que se los referencia.

Acá también entra en juego la baja de registros de una tabla, dado que si su PK es referenciada por otra tabla la base de datos cancelará la transacción para, justamente, mantener la integridad de la base de datos. La base de datos siempre debe pasar de un estado consistente a otro consistente.

- Integridad semántica: tiene que ver con el sentido de los datos. Los datos deben respetar las restricciones definidas sobre los dominios o sobre los atributos, es decir, deben cumplir con las condiciones de negocio, por ejemplo, no tendría sentido que en la columna "Edad" haya un número negativo, o que se acepte un email sin el carácter '@' o, por ejemplo, la aplicación de un Banco no puede permitir que el campo "Nombre" de una persona sea NULL.

Para garantizar la integridad semántica se pueden usar constraints del tipo:

- CHECK.
- DATA TYPE.
- DEFAULT.
- UNIQUE.
- NOT NULL.

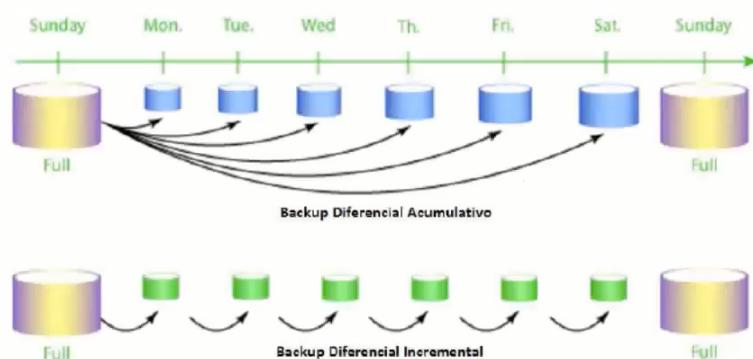
Los objetos de bases de datos que se utilizan para manejar la integridad son:

- Constraints: las constraints sólo sirven para dar integridad. No tienen otra funcionalidad.
- SP/FX/Triggers
- Views

**4. Resguardo, backup y restore:** cuando se hace un backup de una base de datos se backupea absolutamente todo: datos, permisos, funciones, usuarios, etc. Hay dos tipos de backup elementales:

- Backup full: en un determinado momento se copia la base de datos entera.
- Diferencial acumulativo: consiste en backupear las novedades respecto al último full. Como las novedades se van acumulando, el backup de un día me inutiliza el del día anterior. Cuando llegue el próximo backup full, el último backup acumulativo va a quedar inutilizado.
- Diferencial incremental: cada backup tiene las novedades respecto al backup anterior. Es decir, el lunes guarda las diferencias respecto al domingo, el martes guarda sólo las diferencias respecto al lunes, etc. La gran diferencia con el método anterior es que el backup de un día no inutiliza al del día anterior.

\*Nota: en general los full anteriores se mantienen por un tiempo.



Replicación: por resguardo, la información que consideramos esencial y que no puede perderse (como las transacciones en un banco), se tiene que replicar con una transacción distribuida en un servidor en un disco externo. Es decir, dentro del trigger que se ejecuta cuando se inserta una transacción, esos mismos inserts los vamos a replicar afuera en otro lado.

**5. Consistencia:** no es lo mismo que integridad. Nosotros debemos partir de datos válidos, finalizar en datos válidos y que todos esos datos sean vistos por todos los usuarios.

**6. Administración física:** los tablespaces permiten agrupar lógicamente los datafiles o archivos de datos donde se almacenan físicamente los datos de las tablas de usuarios y del sistema.

En un tablespace podemos guardar objetos con una volumetría constante y en otro podemos guardar objetos incrementales (y acá le podemos indicar, por ejemplo, que dispare una alarma cuando el espacio ocupado llegue a cierto porcentaje). Cuando creamos un tablespace podemos setear el tamaño inicial, por lo que ese espacio se reservará en disco (probablemente espacio contiguo) para ese tablespace. Otro parámetro importante son las “extensiones” que es espacio adicional que se asigna al tablespace cuando se agota el espacio inicial asignado.

**7. Optimización de acceso a los datos:** es lo que vimos antes con índices. El DBMS debe encargarse de llevar a cabo el plan de ejecución más performante.

**8. Logs, auditoría, chequeo de recursos:** se deben guardar logs de todo lo que ocurre en la base de datos.

## PREGUNTAS DE PARCIAL

- 1. Explique en no más de una carilla todo lo relacionado con la funcionalidad de integridad.**

Ver integridad, ítem 3.

- 2. Detallar en una carilla todo lo que sepa del objeto de base de datos constraint y su relación con integridad.**

- 3. Explicar en menos de una carilla todo sobre Back-Up y Restore.**

Cuando se hace el backup de una base de datos no sólo se copian los datos sino que también se copian las funciones, los permisos, todo.

Hay dos tipos de backups elementales. El primero es el acumulativo, en el cual siempre se guardan las novedades respecto al último full. Esto quiere decir que el backup de un día inutiliza al del día anterior. El otro tipo de backup es el incremental, en el cual se guardan las novedades respecto al último backup.

- 4. Explicar el modelo ANSI SPARC.**

# Teoría de Objetos de Base de Datos

---

## SNAPSHOTS Y VISTAS

**Vistas:** una vista es una tabla virtual cuyo contenido está definido por una consulta. Los datos de esta tabla proceden de las tablas o vistas a las que se referencia en la consulta y pueden pertenecer a otra base de datos. De esta forma, es una presentación adaptada de los datos contenidos en una o más tablas, o en otras vistas.

Características:

- No aloca espacio de almacenamiento.
- No contiene datos almacenados.
- Está definida por una query que consulta datos de una o varias tablas o vistas.

El uso de vistas tiene diversas motivaciones:

- Para ocultar la complejidad de los datos. O simplemente para simplificar y personalizar la percepción de la base de datos para cada usuario.
- Como mecanismo de seguridad dado que una vista permite a los usuarios acceder a los datos, pero sin tener necesariamente acceso a las tablas subyacentes de la vista. Asimismo, quizás se quiere que un usuario solamente vea ciertas columnas de una tabla y no la tabla completa.
- Pueden utilizarse para proporcionar una interfaz compatible con versiones anteriores con el fin de emular una tabla que existía pero cuyo esquema ha cambiado.

**Diferencias entre una vista y un snapshot:** al igual que una view, un snapshot tiene un select a una o más tablas pero cada vez que se invoque a este snapshot en un from en vez de mostrar los datos al momento va a mostrar los datos a la última "foto" que se les haya sacado. El snapshot en sí tiene una sentencia de refresh que le indica cada cuánto tiene que volver a ejecutarse.

## DBLINK

**Definición:** permite conectar una base de datos con otra. Es un objeto unidireccional.

## SINÓNIMO

**Definición:** va a ser un alias permanente de un determinado objeto.

## IOT (INDEX ORGANIZED TABLE)

**Definición:** tenemos en una misma estructura la tabla y el índice. No es tan difícil de implementar. En el árbol B, en el nodo hoja en lugar de tener un puntero al registro vamos a tener el registro en sí. Esta tabla solamente va a poder ser accedida de dos maneras: orden secuencial o por el índice.

## TABLA TEMPORAL

**Definición:** es una tabla cuyos datos son de existencia temporal. Esta tabla se crea con la siguiente sintaxis: #tabla\_temporal y tiene las siguientes características:

- Solo es visible para la sesión actual. No puede ser vista o utilizada por procesos o consultas fuera de la sesión en la que esta se declara.

- No son registradas en las tablas de diccionario de datos.
- No es posible alterarlas, pero sí eliminarlas y crear los índices temporales que necesite.

Conviene usarlas, por ejemplo, como almacenamiento intermedio en consultas muy grandes ya que pueden servir para guardar resultados intermedios basados en consultas de menor tamaño.

También suelen usarse en ciclos, para almacenar elementos que el ciclo necesite leer. Proporciona un medio rápido y eficiente para hacerlo.

## PREGUNTAS DE PARCIAL

### 1. ¿Qué es una tabla temporal? Mencione un ejemplo concreto de su uso.

Es una tabla cuyos datos son de existencia temporal. Esta tabla se crea con la siguiente sintaxis: #tabla\_temporal y tiene las siguientes características:

- Solo es visible para la sesión actual. No puede ser vista o utilizada por procesos o consultas fuera de la sesión en la que esta se declara.
- No son registradas en las tablas de diccionario de datos.
- No es posible alterarlas, pero sí eliminarlas y crear los índices temporales que necesite.

Conviene usarlas, por ejemplo, como almacenamiento intermedio en consultas muy grandes ya que pueden servir para guardar resultados intermedios basados en consultas de menor tamaño.

También suelen usarse en ciclos, para almacenar elementos que el ciclo necesite leer. Proporciona un medio rápido y eficiente para hacerlo.

### 2. Mencione y detalle un modo de auto numerar una tabla.

### 3. Detallar en una carilla todo lo que sepa del objeto de base de datos vista y su relación con seguridad.

# Data Warehouse

---

## DATOS, INFORMACIÓN Y CONOCIMIENTO

**Introducción:** la inteligencia de negocio se define como el conjunto de metodologías, herramientas y estructuras de almacenamiento que permiten la reunión, depuración y transformación de los datos en una información integrada que se pueda analizar y convertir en conocimiento para la optimización del proceso de toma de decisiones.

**Diferencia entre dato, información y conocimiento:** los datos son valores ya conocidos que se encuentran diseminados en diferentes partes. La información, en cambio, es un dato asociado a una relación que, vinculado y acumulado luego de un proceso de análisis, se transforma en conocimiento.

**Datos:** son la mínima unidad semántica que se corresponden con los elementos primarios de la información que en sí mismos no tienen ningún valor. Para brindar algún tipo de información necesitan que se los vincule con alguna relación. No pueden aportar nada que contribuya con la toma de decisiones.

Provienen de diferentes orígenes: internos (es decir, de la propia organización) o externos (extraídos del contexto). A la vez, pueden ser objetivos, subjetivos y de tipo cualitativo o cuantitativo.

**Información:** conjunto de datos procesados o relacionados con un significado específico. Si se le añade alguna relación, los datos se pueden convertir en información. Un dato se puede transformar en información por diferentes maneras:

- Contextualizando: se sabe en qué contexto y para qué propósito se generaron.
- Categorizando: se conocen las unidades de medida que ayudan a interpretarlos.
- Calculando: los datos fueron procesados matemática o estadísticamente.
- Corrigiendo: habiendo eliminando errores o inconsistencias de los datos.
- Condensando: resumiendo los datos de forma más concisa (agregación de datos).

**Conocimiento:** es el marco conceptual adecuado para la incorporación de nueva información. A medida que se va incorporando más información se generan nuevos conocimientos que contribuirán con la toma de decisiones.

Para que la información se convierta en conocimiento se deben llevar adelante las siguientes acciones:

- Comparación con otros elementos.
- Predicción de consecuencias.
- Búsqueda de conexiones.
- Conversación con otros portadores de conocimiento.

## TECNOLOGÍAS OLAP Y OLTP

**Introducción:** la evolución de la informática a finales de los '90 trajo una acumulación masiva de datos en el entorno empresarial. Esta gran cantidad de datos proviene en su mayor parte de la aplicación de la informática en las actividades de la empresa.

Pero hace falta un modo de estructurar esta información para que aporte una nueva perspectiva. Nace de este modo la tecnología OLAP basada en la utilización de tecnología de Bases de Datos Multidimensionales, para diferenciarse de OLTP (On Line Transaction Processing), que se basan en bases de datos relacionales.

### **OLAP y OLTP:**

- OLAP (On Line Analytical Processing): también llamado modelo relacional, debido a que analiza y relaciona la información analizada. Las aplicaciones en OLAP son usadas por analistas y gerentes que frecuentemente quieren vistas de *alto nivel* de los datos, tales como las ventas de una línea de productos, por región, etc. La base de datos OLAP es usualmente actualizada por bloques, generalmente de múltiples fuentes, y provee poderosas aplicaciones multiusuario de poder analítico. Por lo tanto, las bases de datos OLAP son optimizadas para el análisis.
- OLTP (On Line Transaction Processing): también llamado modelo transaccional, debido a que se basa en la ejecución de un conjunto de transacciones para obtener el resultado esperado. Las aplicaciones con OLTP están caracterizadas en que muchos usuarios crean, actualizan, o retienen registros individuales. Entonces, las bases de datos con OLTP son optimizadas para las actualizaciones de las transacciones.

### **Características OLAP:**

- Su ejecución se basa en el análisis.
- Conforman el 1% de los sistemas existentes.
- Son sistemas para la toma de decisiones.
- Procesan información.
- La información se almacena desnormalizada.
- Registran información global por patrones de interés también conocidos como "dimensiones".
- La información es persistente o "no volátil".

### **Características OLTP:**

- Su ejecución se basa en transacciones.
- Conforman el 99% de los sistemas existentes.
- Son sistemas "operativos".
- Procesan datos.
- Los datos se almacenan normalizados.
- Registran datos a nivel de detalle de cada transacción.

**Estructura OLAP:** la mayoría de los datos que se usan en aplicaciones OLAP son originarios de otros sistemas y aplicaciones. De cualquier modo, en casi la totalidad de las aplicaciones OLAP, los datos son capturados directamente por la aplicación OLAP. Cuando los datos proceden de otras aplicaciones es necesario duplicarlos y almacenarlos separadamente de los originales de los cuales proceden, para poder ser utilizados de manera activa por la aplicación OLAP de manera independiente.

Algunas de las razones que obligan a duplicar los datos para formar el MODELO OLAP son:

- Ejecución: se debe poder acceder a los datos de manera muy rápida, lo cual obliga a que se guarden separados, y a disponer de una estructura de datos optimizada que pueda ser accedida sin perjudicar la respuesta operativa del sistema.
- Múltiples fuentes de datos: el proceso para unir y combinar estos datos procedentes de distintas aplicaciones o sistemas puede ser extremadamente complejo, porque estas aplicaciones o sistemas suelen usar sistemas de codificación diferentes y además pueden disponer de periodicidades distintas.
- Filtrado de datos: en la gran mayoría de sistemas transaccionales nos encontramos con mucha frecuencia gran cantidad de datos que necesitan ser filtrados antes para poder realizar un buen análisis que nos permita generar informes adecuados.

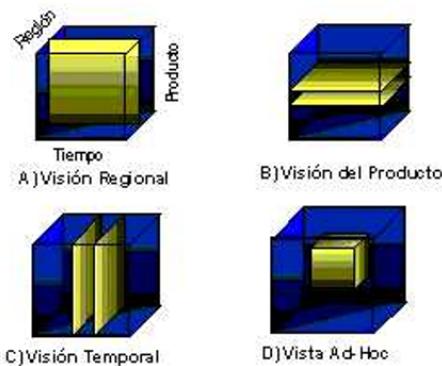
- Ajuste y modificación de datos: hay varias razones por la cuales los datos deben ser ajustados antes de realizar el análisis. Algunas de ellas, podrían ser:
  - Sucursales situadas en otros países operan con contabilidades distintas y los datos puede que necesiten ser modificados antes de usarse en el análisis.
  - Las distintas estructuras de la compañía no siempre son iguales.
  - Se pueden realizar análisis que no parten de datos operativos como pueden ser los que se obtienen de las características demográficas, publicidad televisiva, etc.
- Actualización y consistencia de datos: las aplicaciones de las cuales proceden los datos pueden estar en diferentes estados de actualización. El análisis que realiza un OLAP depende en gran medida de la consistencia de los datos y por lo tanto es necesaria una plataforma que garantice esa consistencia.
- Historia de los datos: la gran mayoría de aplicaciones OLAP incluyen el tiempo como una dimensión. El uso del tiempo como una dimensión permite obtener resultados muy provechosos en cuanto a análisis temporales cuando se dispone de datos de varios años atrás.
- Distintas perspectivas o vistas: los datos operacionales tienen que ser necesariamente muy detallados, pero muchas de las actividades de toma de decisiones requieren una visión a más alto nivel, no tan estructurada. Interesa, por lo tanto, combinar almacenes de datos, ajustar la información según el nivel de resumen o el nivel de visión que se quiere alcanzar.
- Actualización de datos: si la aplicación dispone de varias entradas de datos es necesario separar la base de datos de OLAP para que no se sobreesciban los datos operacionales que se están usando en un determinado momento.

## BASES DE DATOS MULTIDIMENSIONALES

**Definición:** en este caso tenemos 3 dimensiones aunque podemos tener muchas más. Es importante remarcar que una vez que definimos la granularidad no podemos obtener ciertos datos. Por ejemplo, si en la dimensión "tiempo" usamos semanas, podremos sacar datos mensuales o trimestrales pero no diarios.



De este modo la información puede analizarse dentro del cubo formado por la intersección de las dimensiones.



Es simplemente fijar una variable y obtener los resultados. Por ejemplo, si fijamos el tiempo, podemos sacar cuantos productos se vendieron por región en ese tiempo.

Problema: a medida que se agregan dimensiones a una base de datos multidimensional, el número de puntos de datos o "celdas" crece rápidamente. Por ejemplo, considerando que no se venden todos

los productos en todas las sucursales todos los días, y que las sucursales más pequeñas solo pueden manejar el 20% de todos los productos, el 80% de las celdas estarán vacías. En la práctica, muchas bases de datos tienen el 95% de las celdas vacías o en cero. Esto es conocido como "sparsely populated" o dispersión de datos.

#### **Dispersión de datos:**

- Hipercubo: la información se guarda en un único cubo.
- Multicubo: la información se almacena dividiendo los datos en grupos más pequeños y densos (objetos multidimensionales), donde la base de datos multidimensional consiste en un número de objetos separados normalmente con diferentes dimensiones.

#### **BDM vs BDR**

	<b>Base de Datos Relacional</b>	<b>Base de Datos Multidimensional</b>
<b>Depósito de datos, acceso y visión</b>	Relacional Tablas de Columnas e hileras Lenguajes SQL con ampliaciones Herramientas de terceros que usan API	Dimensional Arreglos: Hipercubo, Multicubo Tecnología de matriz dispersa Propietario de hoja de cálculo
<b>Utilización e incorporación</b>	OLTP Motor RDBMS Profundización a nivel de detalle Desempeño de consultas: rango amplio	OLAP Motor multidimensional Profundización a nivel de resumen/adición Desempeño de consultas: rápido
<b>Tamaño y actualización de bases de datos</b>	Gigabytes a terrabytes El depósito de índices y el retiro de normas que incrementan tamaño Consulta y cargas paralelas Actualización durante uso	Gigabytes Compresión y adición de datos dispersos Difícil actualizar durante uso; los cambios pequeños pueden requerir reorganización

#### **DATAWAREHOUSE**

**Tabla de hechos:** contiene las métricas. Su PK está formada por varios campos, que son las dimensiones y son todas FKs.

#### **Dimensiones:**

- Al definir una unidad, después no se puede sacar información de una unidad menor.

#### **Data mart:**

- Si todas las dimensiones tienen 1 sola tabla, se llama modelo estrella.
- Si al menos 1 dimensión tiene más de 1 tabla, se llama modelo copo de nieve.

#### **Datawarehouse:**

- Si todas las dimensiones tienen 1 sola tabla, se llama constelación de estrellas.