GRAFOS

- Puede definirse como G = (V.A)
- **V**: conjunto de **vértices** o nodos
- **A**: conjunto de aristas/relaciones
- Por lo tanto, **Grafo** es un conjunto de **vértices** y **aristas** que los relacionan
- **Nodos**: Son los vértices, dado que un modelo computacional de grafos pueden incluir muchos valores
- **Relaciones**: aristas/arcos, relacionan a los nodos
- **Grado**: Es la cantidad de arcos que salen de un vértice **(grado positivo),** o la cantidad de arcos que llegan a un vértice **(grado negativo)**
- En los grados negativos, no cuenta el vértice entrante de un bucle; en el positivo si (con el que sale)
- **Grado de un grafo:** grado máximo que alcanza uno de sus vértices

Objetivo:

- **Modelizar un problema** específico a través de un **modelo abstracto** donde los elementos que participan en el problema son los vértices y las relaciones que pueden existir entre estos participantes son los arcos

- Los grafos son **estructuras abstractas** (**no existen realmente** sino que solo sirven como una modelización virtual de un problema real)

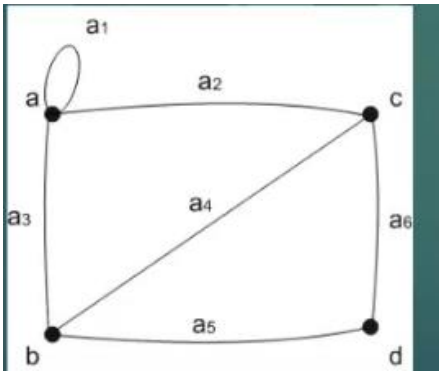
- Para su tratamiento computarizado, los grafos requerirán de una **representación computacional,** lo cual convertirá esta estructura abstracta en un **almacenamiento concreto** (memoria o disco) representado a través de alguna de las representaciones computacionales existentes:

Formas de representar un grafo:

- **Estática**: Se construyen sobre **estructuras computacionales rígidas** que utilizan el concepto de **contigüidad** (vectores, matrices), es decir, el siguiente vértice está a la derecha, el anterior a la izquierda de un vértice determinado, y lo mismo con los arcos (se debe contemplar la existencia de **TODAS** las relaciones posibles entre todos los vértices existentes)
- **Dinámica**: Acompaña la dinámica del grafo (**el espacio utilizado** por la representación va cambiando en **función** de cómo va cambiando el grafo)

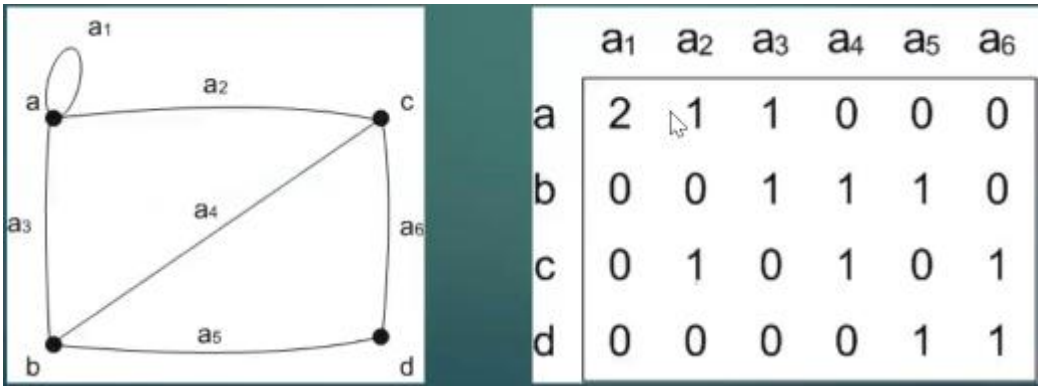
Estática:

Matriz de adyacencia: dados N vértices, es una matriz **NxN**



	a	b	c	d
a	2	1	1	0
b	1	0	1	1
c	1	1	0	1
d	0	1	1	0

Matriz de incidencia: Dados N vértices y M arcos, es una matriz **NxM**

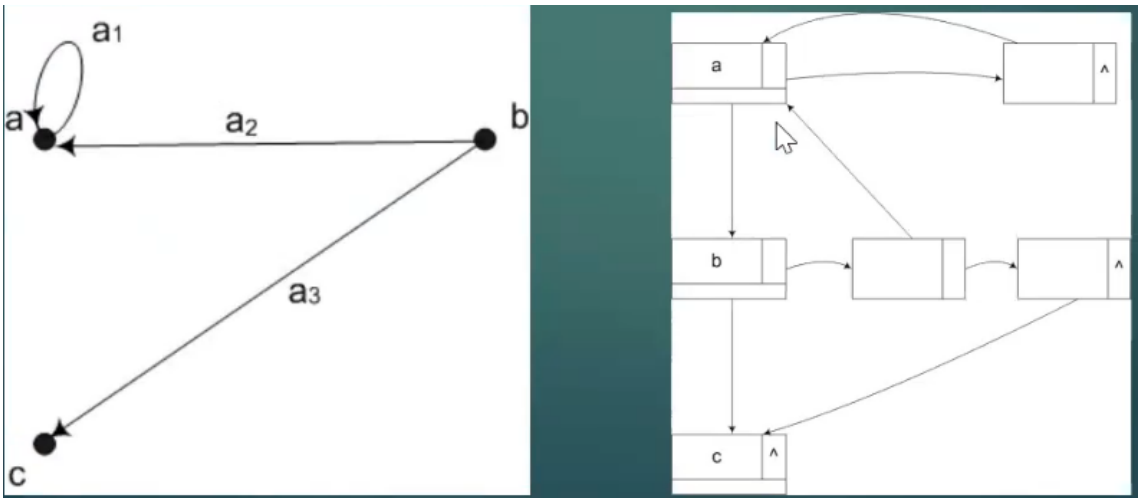


	a_1	a_2	a_3	a_4	a_5	a_6
a	2	1	1	0	0	0
b	0	0	1	1	1	0
c	0	1	0	1	0	1
d	0	0	0	0	1	1

Dinámica:

Listas de adyacencia: Una lista en la cual cada elemento representa los nodos del grafo. A su vez cada nodo mantiene otra lista asociada que representa los arcos o relaciones que salen de dicho nodo.

Cada nodo puede cargar datos respectivos al elemento (nombre, id, domicilio, etc…)



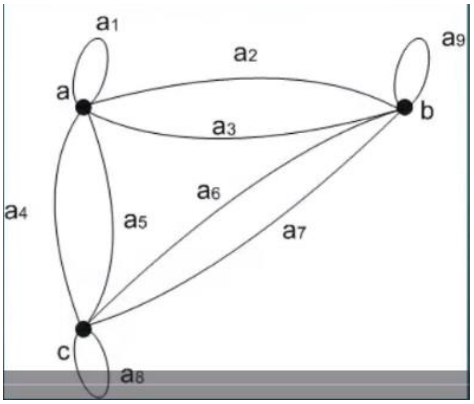
Dinámica con grafo fijo

Pfaltz

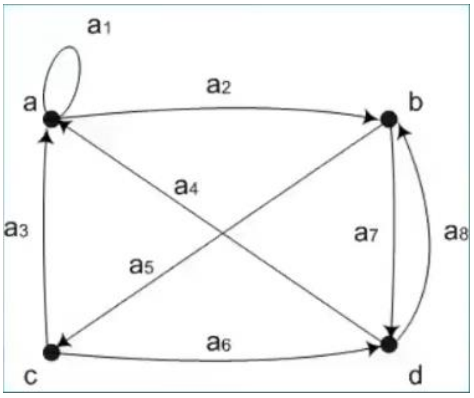
Caracterización de grafos:

Grafo libre: Grafo en el cual **no existen arcos**, todos sus vértices son aislados

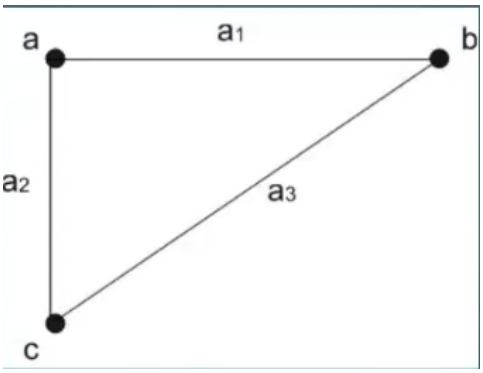
Grafo completo: Tiene **todos los arcos**/relaciones posibles entre nodos, cada vértice está conectado a todos los vértices que componen el grafo (incluido el mismo)



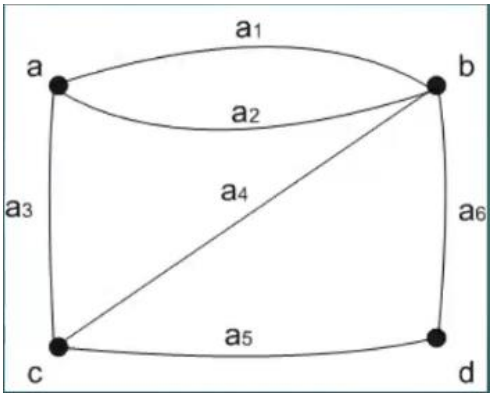
Grafo regular: cada uno de los nodos tiene el **mismo grado de salida** (positivo)



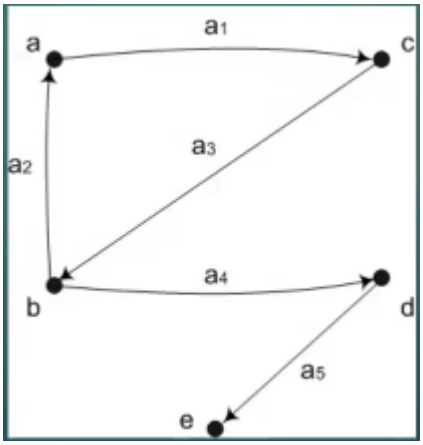
Grafo simple: Por cada par de vértices, hay un único arco que los relaciona



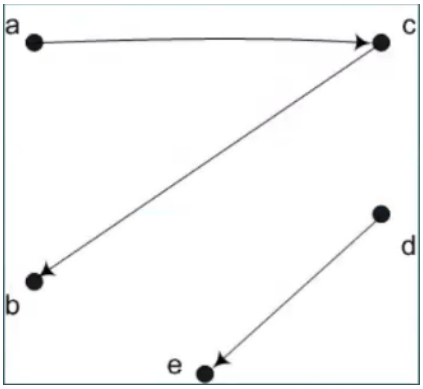
Grafo complejo: lo opuesto a simple, cualquier grafo que no es simple es complejo



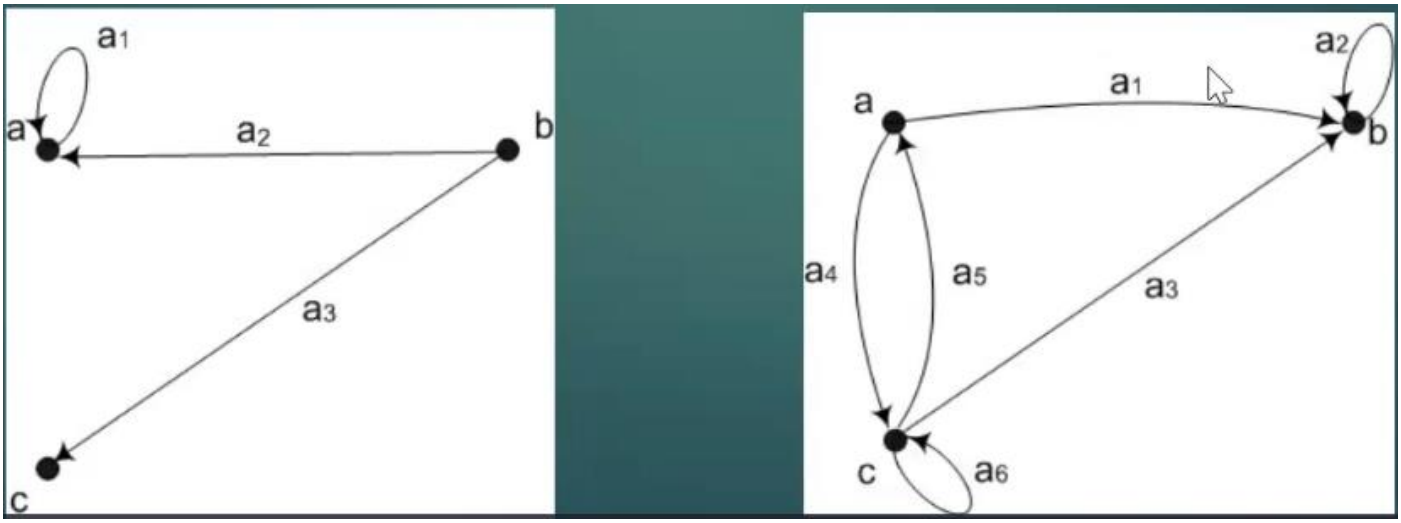
Grafo conexo: todo par de vértices está conectado por un **camino** (sin importar las direcciones)



Grafo No conexo: El grafo se encuentra (**desconectado**), cualquier grafo que no cumpla con la condición de grafo conexo (incluido un único vértice aislado)



Grafo complementario: dado un grafo G simple, su grafo complementario Gc está formado por los **mismos vértices** que G pero sus aristas son todas aquellas que le faltan a G para ser un **grafo completo**



Clasificación de grafos:

Grafo dirigido: se identifica el sentido de la relación (ej Instagram con seguidores vs seguidos)

Grafo no dirigido: No se identifica el sentido a la relación (ej Facebook con “amigo de”)

Restrictos: Grafos en los cuales la relación que se modela **NO DEBE CUMPLIR** con las propiedades de **reflexividad**, **simetría** y **transitividad**, (simplifica la ejecución/procesamiento y codificación de su implementación y algoritmos que se corren en el)

Grafo Irrestrictos: no se aplica ninguna restricción a la relación que se modela (requiere más codificación en su implementación para contemplar todas las situaciones que pueden darse)

Camino/paso/ciclo

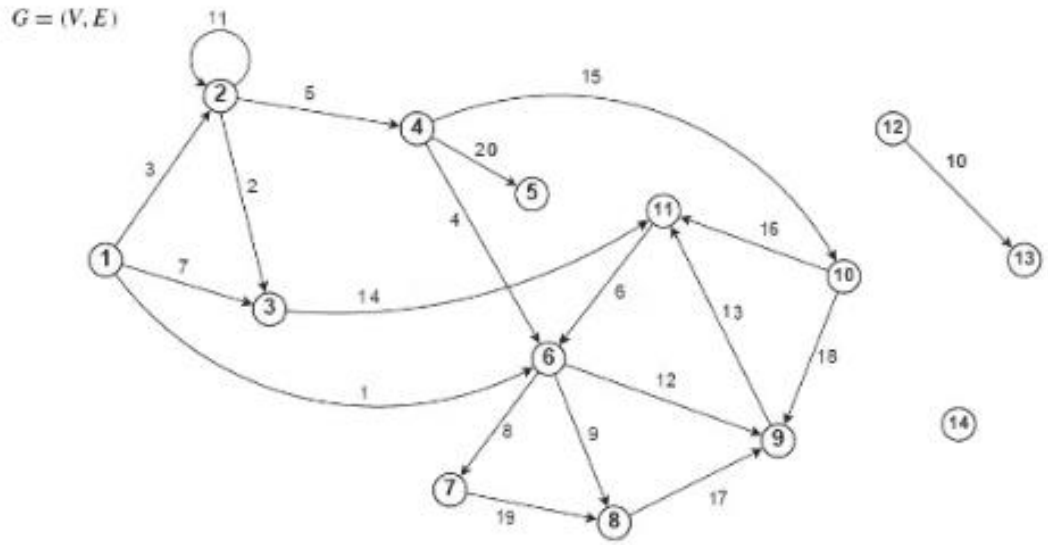
Camino: Entre dos nodos hay una vinculación **directa** o **indirecta** entre ambos (**independientemente del sentido**)

Paso: Entre dos nodos hay un camino, pero teniendo en cuenta el **sentido** (partiendo de un nodo a y siguiendo el sentido de los arcos puedo llegar al nodo b)

Ciclo: Es un paso o camino donde el origen y el destino son iguales (un ciclo puede estar compuesto por uno o más arcos)

Implementaciones – Representaciones dinámicas

Dado este grafo:

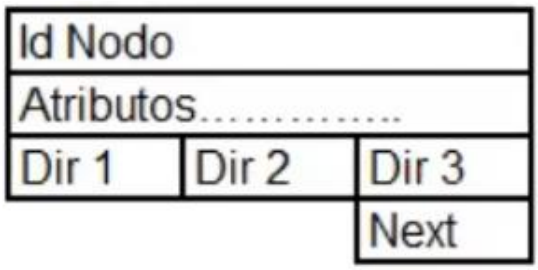


Dinámica con grado fijo:

Una única estructura que sirve **SOLO para los nodos**, no se modelan las relaciones

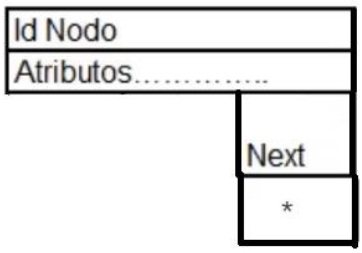
Como el grado del grafo es 3, la estructura tendrá 3 punteros para representar máximo 3 relaciones

El next marca el orden en el cual se cargaron los nodos, el 3 apunta al 4, el 4 al 5, el 5 al 6, etc… (el next de 14 es NULL por ser el último)



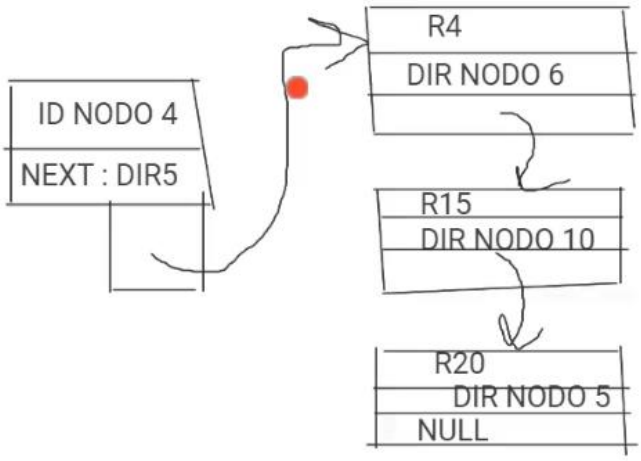
Lista de adyacencia:

Hay dos estructuras, una para los **nodos** y otra para las **relaciones**



Donde * es un **puntero** a la lista de arcos salientes de dicho nodo

- Ejemplo para el nodo 4:
- Id nodo 4
- dir 6, dir 10, dir 5
- next: dir 5
- * →R4(+atributos del arco)→R15(+atributos del arco)→ R20(+atributos del arco):



Atributos arcos:

- id_arco
- atributos
- puntero al nodo destino
- next(puntero al arco siguiente)

PFALTZ:

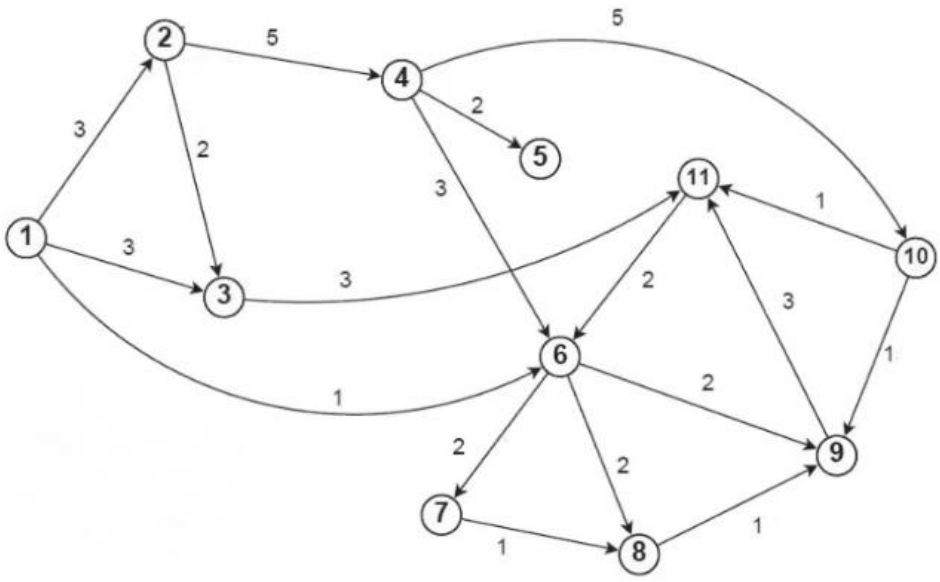
Algoritmos

Dado un grafo (en cualquier representación) correr un algoritmo me permitirá **recorrer** (obtener datos) o **modificar** algún grafo

En los algoritmos, hay dos maneras (ambas **recursivas**, es decir, se llama a si mismo x veces) para recorrer un grafo:

Recorrido en profundidad: Le paso por parámetro un solo nodo

Recorrido en anchura: Le paso por parámetro una colección de nodos



Ejemplo recorrido profundidad para grafo de arriba:

1. Empiezo por el nodo 1, obtengo el 2 (mientras el nodo 3 y 6 se quedan esperando)
2. tengo 1, 2(por el 1), obtengo el 3, 11(por el 3) y 6 se queda esperando
3. etc…

Ejemplo recorrido anchura para grafo de arriba:

1. Empiezo por el nodo 1, obtengo la colección 2, 3 y 6
2. obtengo la colección 1,2,4(por el 2), 3, 11(por el 3), 6, 7(por el 6), 8(por el 6) y 9(por el 6)
3. etc…

Grafo ejemplo (mismo que el de arriba pero con distancias random entre nodos, pueden ser metros por ej)

Dijktra:

Dado un grafo y un nodo, me permite calcular **la mínima distancia** desde ese nodo a todos los nodos por los cuales tenga **paso**

Ejemplo para nodo 11 del grafo de arriba y recorrido en profundidad:

1. Nodo origen = 11
2. **Distancia mínima 11 a 11 = 0**
3. Obtengo nodo 6 →distancia = 2
4. **Distancia mínima de 11 a 6 = 2**
5. Obtengo nodo 7, (nodo 8 y 9 se quedan esperando) →distancia 11 a 7 = 2 + 2
6. **Distancia mínima de 11 a 7 = 4** →(en espera de 6 a 8 y de 6 a 9, tiene que terminar el 7 por recorrido en prof.)
7. Obtengo nodo 8, (9 se queda esperando) → distancia 11 a 8 = 4(11 a 7) + 1(7 a 8)
8. **Distancia mínima 11 a 8 = 5** (por ahora)
9. Obtengo nodo 9 → distancia 11 a 9 = 5 (11 a 8) + 1(8 a 9)
10. **Distancia mínima 11 a 9 = 6** (por ahora)
11. **Distancia de 9 a 11 = 6 + 3 = 9**
12. Pero de 11 a 11 tengo distancia mínima 0 < 9 →por lo tanto corto la recursividad

13. Obtengo el primero que estaba en espera (de 6 a 8)
14. **Distancia 11 a 8** = 2(11 a 6) + 2(6 a 8 que estaba en espera) = **4**
15. 4 < 5(el de antes) → por lo tanto modifico con la nueva distancia mínima
16. Obtengo el otro que estaba en espera (de 6 a 9)
17. **Distancia 11 a 9** = 2(11 a 6) + 2 (6 a 9)
18. 4<6 → por lo tanto modifico con la nueva distancia mínima
19. **de 9 a 11** = 4 + 3 = 7
20. 11 a 11 = 0 < 7 → corto recursividad
21. Al no tener más hilos en espera termina el algoritmo

Resultado:

DISTANCIA MINIMA A 11 = 0

DISTANCIA MINIMA A 6 = 2

DISTANCIA MINIMA A 7 = 4

DISTANCIA MINIMA A 8 = 4

DISTANCIA MINIMA A 9 = 4

Floyd-Warshall de clausura transitiva

Parecido al dijktra pero en vez de un solo nodo de origen, para todos los nodos, la distancia hacia todos los nodos

El resultado es una matriz de NxN (N cantidad de nodos)

Filas: nodos de origen

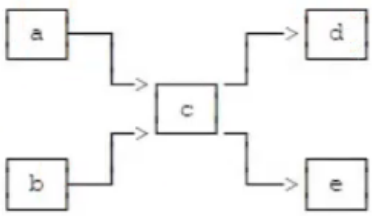
Columnas: nodo destino

Arboles

Un árbol es un grafo que cumple con las siguientes condiciones:

1. **Grafo dirigido**
2. **Aciclico** (no puede tener ciclos)
3. Existe un **único camino** (no se contemplan los sentidos de los arcos) entre cada par de nodos
4. $\sum V = \sum A + 1$ →(La cantidad de nodos es la cantidad de relaciones/arcos + 1)

Subárbol:



- **Subárbol principal derecho** (de un nodo): el subárbol que contiene todos los nodos desde los cuales hay **un paso** (**desde** ese nodo) → Ej árbol de arriba: subárbol principal derecho de nodo a: a, c, d, e. de nodo c: c, d, e. etc…

- **Subarbol principal izquierdo:** Conjunto de nodos desde los cuales hay **un paso hacia** ese nodo. ej: para nodo d: a, b, c, d

Árbol principal derecho: existe un **único nodo** cuyo subárbol principal derecho es el árbol completo. ej: no hay árbol principal derecho en el árbol de ejemplo. **Para nodo a**: b queda afuera && **para nodo b**: a queda afuera

- Árbol principal derecho = Árbol computacional = árbol → ese único nodo = raíz del árbol

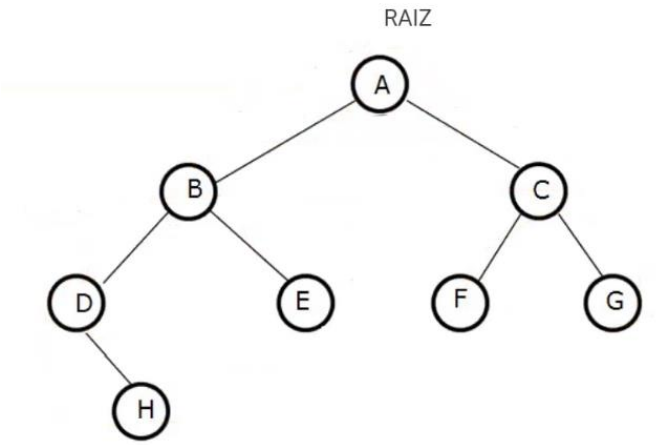
Grado:

Acá siempre por grado se entiende grado **positivo** (a la raíz no le llega ninguna relación y a los demás nodos por definición siempre les llega una única relación)

Al igual que con grafos, se toma el grado mayor

- Si grado = **2** → Árbol **binario**
- Si grado = **3** → Árbol **ternario**
- **4** = **cuaternario**, 5 = etc…

Árbol completo: todos los nodos **no maximales** (no hoja) tienen el mismo grado



Rama de un árbol: nodo que no es raíz ni hoja. → B, C, D

Profundidad: longitud del paso de la raíz a un nodo:

- Profundidad A: 0
- Profundidad B y C = 1
- Profundidad D, E, F y G = 2
- etc…

Nivel: Clase de equivalencia de nodos con misma profundidad:

Nivel 2: D, E, F, G

Altura: Cantidad de niveles

Arboles binarios

Árbol balanceado: Si me paro en la raíz, la cantidad de nodos en la rama izquierda y la cantidad de nodos en la rama derecha, coinciden o difieren en máximo 1

Árbol perfectamente balanceado: Se cumple lo anterior pero para todos los nodos (no solo para la raíz)

Árbol AVL:

En todos los nodos, la **altura** de la rama izquierda no difiere en más de una unidad de la altura de la rama derecha

Implementaciones de árboles en memoria

Estática:

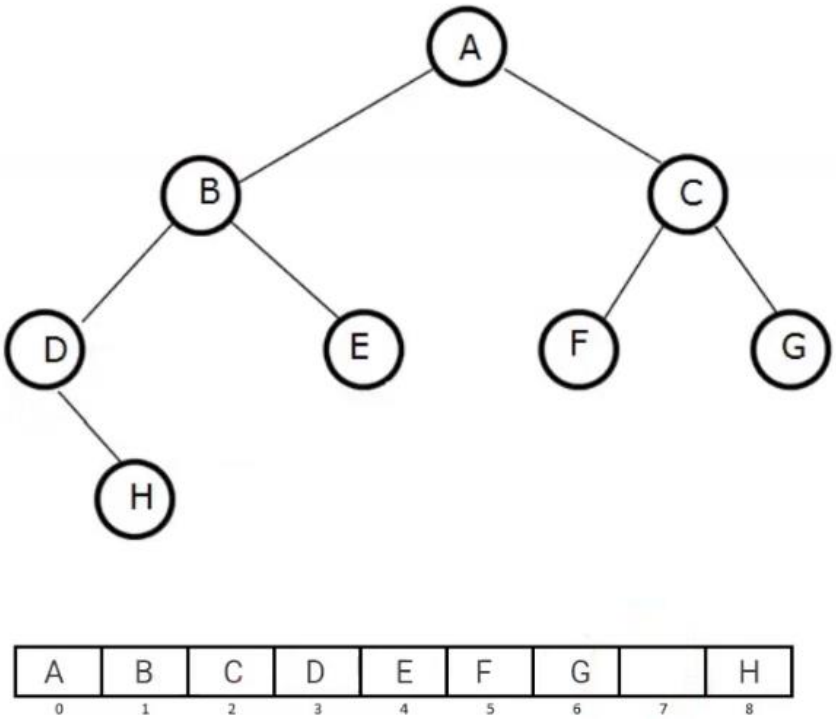
Vector:

Requiere saber de antemano el **grado** del árbol, ya que esto define su implementación (varía según el grado)

Lo que se implementa en memoria es el vector solamente, el árbol en si es una abstracción para entenderlo

Para un árbol **Binario**: (siendo i el subíndice del vector)

- Hijo **izquierdo** de i: [(**i**+1) * 2] – 1
- Hijo **Derecho** de i: (**i**+1) * 2
- **Padre** de i: Int[(**i**-1) / 2]



Si este árbol fuera ternario, sería:

- Hijo **izquierdo** de i: [(**i**+1) * 3]
- Hijo **medio** de i: [(**i**+1) * 3] – 2
- Hijo **derecho** de i: [(i+1) * 3] -1
- **Padre** de i: Int[(**i**-1) / 3]

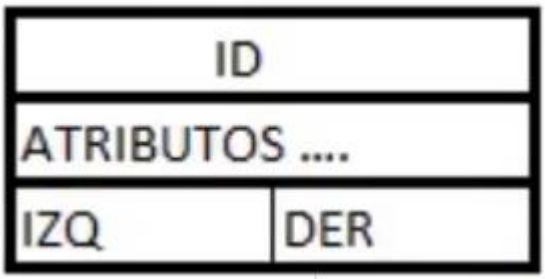
Idem cuaternario:

- Hijo **izquierdo** de i: [(**i**+1) * 4]
- Hijo **medio** de i: [(**i**+1) * 4] – 3
- Hijo **derecho** de i: [(i+1) * 4] -1
- **Padre** de i: Int[(**i**-1) / 4]

Dinámica:

Grado fijo

Para árbol **binario**:

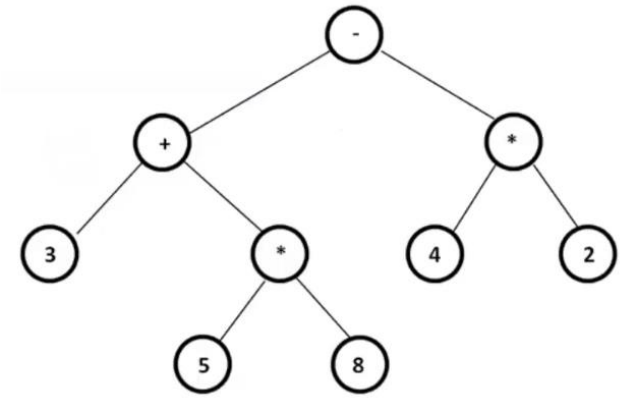


No necesito el puntero a next, ya que tengo la seguridad de que desde A (primer nodo que es apuntado por FRONT) puedo llegar a todo el árbol

Algoritmos

Para grado fijo

Arboles de expresión (se usan en cualquier calculadora /cálculos de una computadora):



Un operador (+, -, *, /) y dos operandos como máximo a la vez

Barridos/recorrido de árboles:

En profundidad

(le paso un único nodo):

- Preorden
- simétrico/in-orden **(SOLO PARA BINARIOS)**
- post orden

Algoritmo Preorden:

```
PreOrden (x nodo){
    Recursivo(x);
}

Recursivo(x nodo) {
    If x IS NULL Then
        Return;
    Print x-->ID;
    Recursivo (x-->izq);
    Recursivo (x-->der);
}
```

Algoritmo simétrico/inorden:

```
PreOrden (x nodo){
      Recursivo(x);
}

Recursivo(x nodo) {
      If x IS NULL Then
            Return;
      Recursivo (x-->izq);
      Print x-->ID;
      Recursivo (x-->der);
}
```

Algoritmo postOrden:

```
PreOrden (x nodo){
      Recursivo(x);
}

Recursivo(x nodo) {
      If x IS NULL Then
            Return;
      Recursivo (x-->izq);
      Recursivo (x-->der);
      Print x-->ID;
}
```

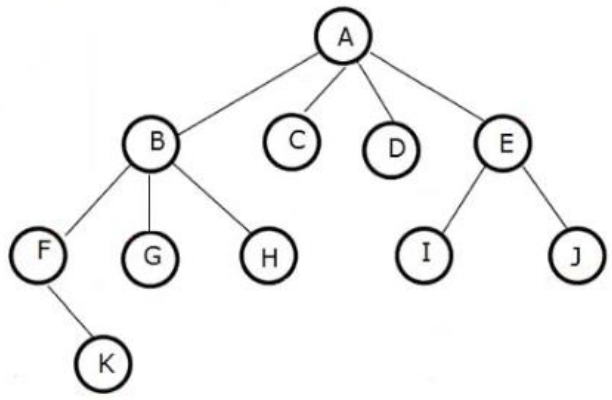
Por anchura

(Le paso una colección de nodos):

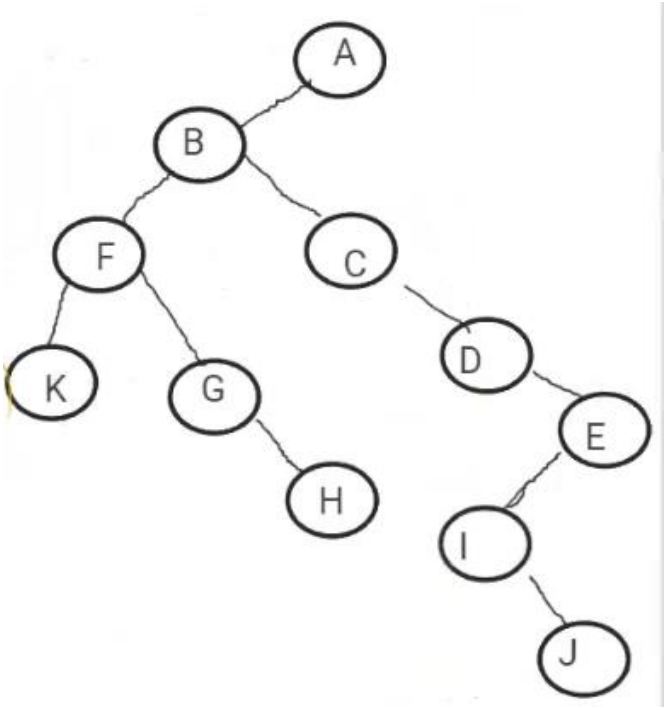
- Por niveles

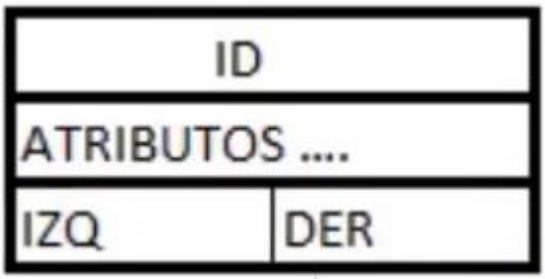
Para grado Variable: KNUTH

Dado un arbol n-ario abstracto:



Su implementación será un **árbol binario (es decir, transforma un árbol n-ario en uno binario):**





Se usa la siguiente estructura de nodo (misma que implementación dinámica con grado fijo):

El puntero **izq** representa el primogénito (B)

El puntero **der** es una lista de los hermanos del primogénito (C, D, E)

Lo mismo para cada nodo (tiene su primogénito y hermanos)

Implementaciones de grafos/árboles en DB

DB Relacional:

Dos tablas involucradas:

```
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
```

Dos FK, una para el nodo origen, otro para el destino

```
SELECT r.*, o.descripcion, d.descripcion
FROM relaciones r
INNER JOIN nodos o ON r.origen = o.idnodo
INNER JOIN nodos d ON r.destino = d.idnodo
```

Implementación en vida real

- Un filesystem/explorador de archivos es un **arbol** de **grado variable.** Al ser de grado variable, su implementación será un arbol binario (knuth)
- Google maps esta implementado como una matriz
- Chatgpt un grafo (en sus redes neuronales)
- Grafo/arbol algebraico para árboles genealógicos
- Parsers/autómatas utilizan arboles
- Algoritmos de compresión utilizan arboles

Algoritmos de compresión de datos:

Existen dos tipos:

Con pérdida: Al comprimir pierden información y por dicha perdida **no son reversibles** (este tipo de algoritmos se usa para archivos **multimedia**, las puedo recuperar pero al comprimir pierdo calidad de imagen)

Sin pérdida: Algoritmos que comprimen archivos sin perder información, por lo cual **son reversibles** (permiten volver al estado original del archivo), se utilizan para todos los archivos menos los multimedia

Por qué es posible comprimir un archivo?:

El alfabeto ASCII se creó de longitud fija (por más que un carácter se repita mucho más que otro, el espacio que ocupa es el mismo)

No todos los archivos contienen los 256 caracteres especificados en el archivo ASCII, por lo cual podría ocurrir que no se utilicen todos, desperdiciando bits en su representación

Para comprimir se puede llevar a cabo una contabilización de cuantos caracteres tenemos en un archivo, y por cada uno cuantas veces aparece

Algoritmo de Huffman

Principalmente para comprimir **archivos de texto**

Las computadoras codifican los caracteres mediante ASCII o Unicode (1byte y 4bytes respectivamente)

El algoritmo identifica cada uno de los caracteres distintos en el conjunto de archivos a comprimir, y le asigna un código de **longitud variable** según la frecuencia (mientras **más veces aparezca** un carácter, su código será **más pequeño)**

+ apariciones → código + pequeño

Compresión:

Se debe poder comprimir tanto el **contenido** como la **metadata** (usuario, fechas, etc…) y al descomprimir recuperar toda esta información (no solo el contenido)

Además debe ser capaz de comprimir varios archivos (no solo uno) y poder, dentro de su implementación, realizar la separación de c/u de ellos

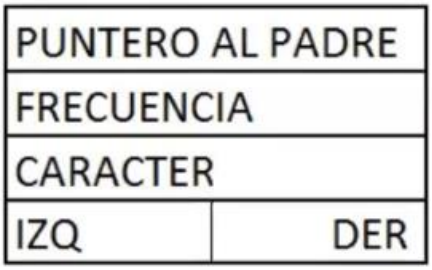
Para un archivo de tamaño normal, el porcentaje aprox de compresión es del 40% de su tamaño original

Si el archivo a comprimir es demasiado chico, su tamaño post compresión probablemente será más grande que el original

Estructuras:

1) Tabla de huffman → Dados N caracteres →**Filas**: N*2 -1
   Columnas:
   1.1 Carácter
   1.2 Frecuencia
   1.3 Código de longitud variable
   1.4 Puntero al árbol
   1.5 status (flag)

2) Arbol de Huffman



→ Tiene dos punteros (árbol binario)

3) Pila

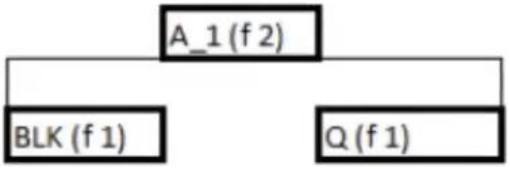
Ejemplo para compresión del contenido de un archivo = **"EN NEUQUEN"**

Tabla EJ:

Status	Char	Freq	Código	dir árbol
	E	3		
	N	3		
	U	2		
	BLK	1		
	Q	1		

Guardo dos variables:

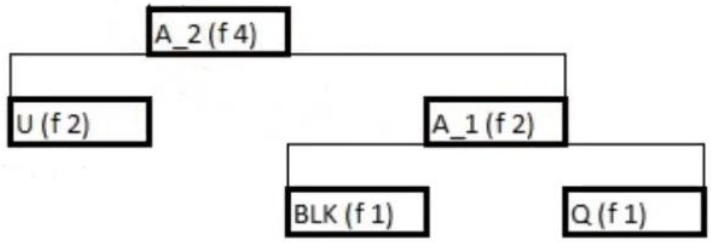
4) Posición de la tabla del ultimo valor (el menos frecuente) →EJ: Q
5) Frecuencia del valor en esa posición → EJ: 1
6) Voy a recorrer la tabla de abajo hacia arriba, buscando pares (por ser árbol binario) de caracteres con la misma frecuencia (1) y los coloco como hijos de un nodo que represente la suma de las frecuencias de sus hijos, en este caso tengo 2 caracteres con Frec = 1 → Frec. padre = 1 + 1 = 2
7) Cada vez que recorro la tabla, cambio la variable "posición" (primero pos = 4(Q), después pos = 3(BLK), etc…)
8) Primero encontré a Q en la tabla, por lo tanto Q va a la derecha



9) Ahora relleno el status de BLK y Q = 1 (porque ya tienen un padre asignado)
10) Agrego A_1 a la tabla (con su frec = 2) pero no le relleno el status (aún no tiene padre asignado)

Status	Char	Freq
	E	3
	N	3
	U	2
1	BLK	1
1	Q	1
	~~1~~	~~2~~

11) Ahora pos = 2 (U) →frec(U) = 2 != 1 → Bajo al principio de la nueva tabla modificada, y busco caracteres cuya frec = 2
12) Pos. tabla = 5 →**encuentro A_1 con frec = 2**
13) Salteo Q y BLK (porque ya están ubicados en el árbol y con su respectivo padre)
14) Encuentro U con frec = 2
15) Primero encontré a A_1 en la tabla, por lo tanto va a la derecha
16) A_2 nuevo padre con frec = 2 + 2 = 4



17) Agrego el nuevo padre a la tabla (sin status)

Status	Char	Freq
	E	3
	N	3
1	U	2
1	BLK	1
1	Q	1
1	~~1~~	~~2~~
	~~2~~	~~4~~

18) sigo recorriendo →pos(1) = N → frec(N) = 3 != 2 → repito lo mismo que antes para seguir completando el árbol y la tabla
19) Me van a quedar dos subárboles de este tipo:

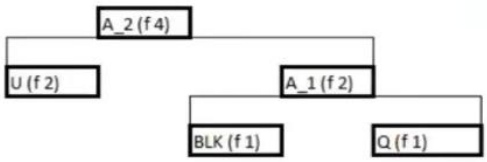
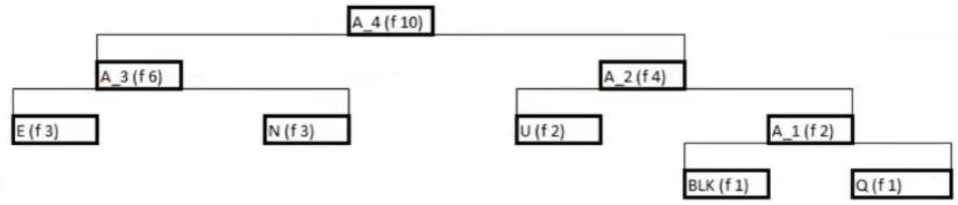


Tabla:

Status	Char	Freq
1	E	3
1	N	3
1	U	2
1	BLK	1
1	Q	1
1	∝1	2
	∝2	0
	∝3	0

20) Busco frec = 4 → encuentro A_2
21) Sigo buscando frec = 4 no encuentro nada → Ahora busco desde debajo de todo frec = 5 →no encuentro nada
22) Ahora busco frec = 6 → encuentro solo A_3
23) Creo el padre A_4 con frec = 4 + 6 = 10
24) primero encontré A_2 →va a la derecha



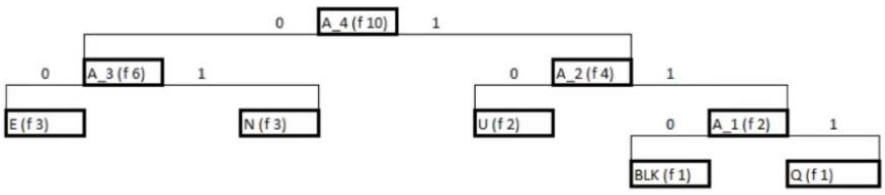
Status	Char	Freq
1	E	3
1	N	3
1	U	2
1	BLK	1
1	Q	1
1	∝1	2
1	∝2	0
1	∝3	0
	∝4	10

El algoritmo detecta que queda un único nodo, ese nodo será la raíz del árbol de huffman

Una vez completado el árbol, desde la raíz hacia el nodo del carácter a calcular su código variable, cada vez que me mueva hacia abajo a la **izq** codifico un **0**, y hacia **abajo-derecha** un **1**

Por eso, los caracteres con menos apariciones estarán más abajo en el árbol (un nivel más grande), por lo tanto su código obtenido será más largo (más espacio ocupa)

- Si tengo X caracteres, voy a tener X-1 alfas/padres y X*2-1 nodos en total



Por ultimo armo la última estructura (**pila**)

Para obtener el código de long. variable de cada carácter

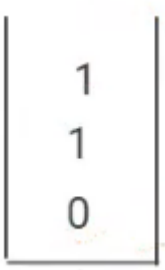
EJ: carácter BLK

Desde BLK subo a su padre (A_1) y guardo en una variable el puntero BLK

Consulto los hijos izq y der de A_1 →Como puntero BLK = puntero izq de A_1, sé que BLK es el hijo izq

Coloco un 0 en la pila

Repito hasta llegar a la raíz del árbol, hasta obtener la siguiente pila:



→Al desapilar, obtengo = 1,1,0 →código huffman del carácter BLK

Repito para todos los caracteres, hasta obtener sus códigos respectivos:

Status	Char	Freq	Código	dir árbol
1	E	3	00	
1	N	3	01	
1	U	2	10	
1	BLK	1	110	
1	Q	1	111	

EN NEUQUEN

Finalmente armo el archivo comprimido:

Cabecera:

- Cantidad de caracteres distintos (para saber cuándo termina la cabecera)
- Tabla de huffman original (2 columnas: carácter – frecuencia) x5 en el EJ

Cuerpo:

Reemplazo de cada caracter por su código: EJ: EN NEUQUEN = **00011100 10010111 10000100** →(últimos dos ceros de relleno)

Descompresión:

1) Identificar código cabecera

2) identificar fin código

3) identificar fin relleno

- Identificar fin cabecera →se usa la metadata de cantidad caracteres distintos
- Armo la tabla **carácter-frecuencia** en el mismo orden (ya que se descomprime usando el mismo .exe, por lo tanto tiene disponibles los métodos que se usaron para comprimir)
- Sumo todas las frecuencias y las guardo en una variable **pendientes**, obtengo como resultado la cantidad de caracteres pendientes a descomprimir
- En este caso **P = 10**
- Cuando llegue a **P = 0** sé que tengo que dejar de descomprimir (los números restantes serán relleno)
- Además sé que todas las hojas son caracteres
- Con la tabla obtenida, armo el árbol y procedo a descomprimir

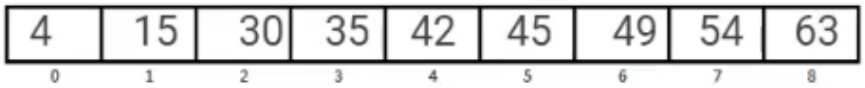
Desde la raíz (que tiene el front) voy recorriendo según cada código hasta llegar a una hoja, ahí corto y reemplazo por el carácter de esa hoja

Complejidad computacional

Complejidad computacional: métrica que mide cuanto procesa un determinado algoritmo de búsqueda para determinado tamaño de la estructura

Dividimos en **mejor caso**, **promedio**, **peor caso**

Búsqueda binaria:



Supongamos que quiero buscar el 15

1. Voy a la mitad, obtengo 42 → como 15 < 42 sigo por la izquierda del 42
2. Voy a la mitad de la mitad izquierda, obtengo 30 → como 15 < 30 sigo por la izquierda del 30
3. obtuve el dato buscado (15)

Complejidad computacional de la búsqueda binaria: cuantas veces voy a realizar lecturas sobre el vector, para llegar a obtener el dato buscado

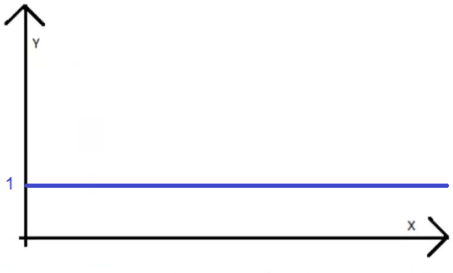
X: tamaño de la estructura (en este caso del vector)

Y: complejidad computacional (en este caso cantidad de lecturas del vector hasta encontrar el dato a buscar)

Mejor caso:

El dato a buscar justo se encontraba a la mitad del vector →encuentro el dato en la primera lectura

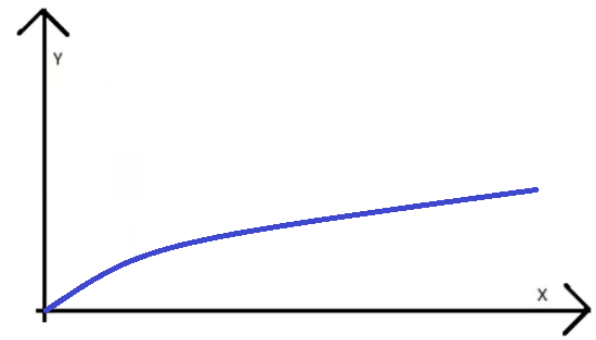
F(X) = 1 (CTE)



Peor Caso:

Encontrar el dato luego de haber hecho todas las lecturas posibles (o directamente no existe)

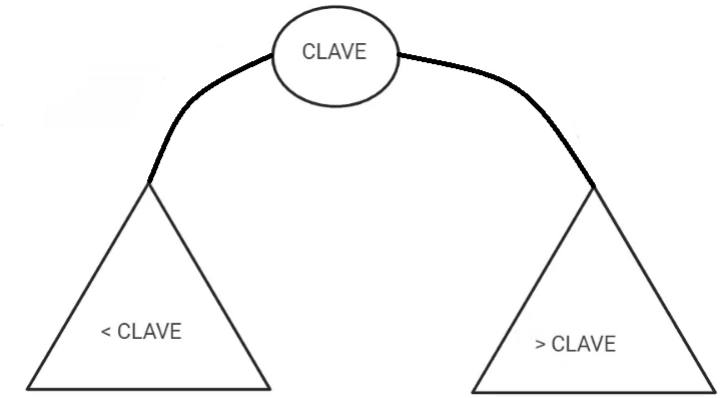
F(X) = $\log_2(X+1)$ →Siendo X el tamaño del vector



Árbol binario de búsqueda:

En una lista, al ser de tamaño dinámico, no puedo realizar búsqueda binaria para que sea tan eficiente como en un vector (ya que tiene tamaño fijo)

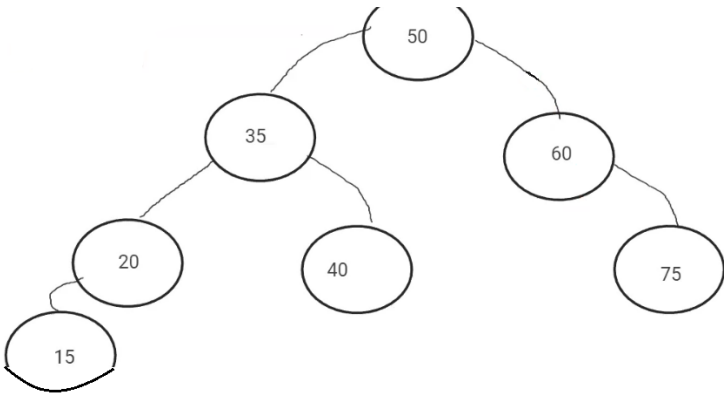
Para eso uso el árbol binario de búsqueda, el cual tendrá la misma eficiencia que la búsqueda binaria para los vectores, pero este lo será para las listas



Esta es su estructura básica

El subárbol izquierdo de un nodo con X clave, tendrá todas claves menores a la misma, y el subárbol derecho estará compuesto por nodos con clave mayor al padre

Para garantizar performance, el árbol debe estar balanceado



Ejemplo: 50, 60, 35, 40, 75, 20, 15

Supongamos busco el 40:

1. Leo 50. 40 < 50 →voy a la izq
2. leo 35. 40 > 35 → voy a la der
3. leo 40. lo encontré

Se verifica si esta balanceado/perfectamente balanceado

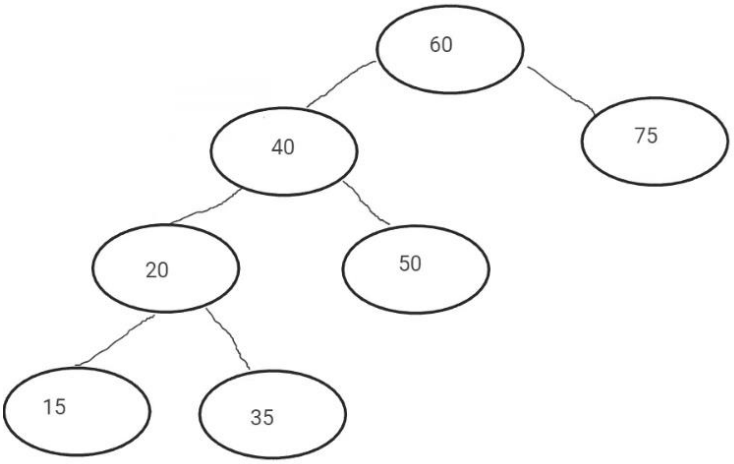
Si no está balanceado, se ejecuta sobre el árbol el **algoritmo de rotación** para terminar de balancearlo

Una vez que se ejecute dicho algoritmo, finalmente tendremos el árbol binario de búsqueda definitivo

Rotación a derecha:

Tiene que estar perfectamente balanceado (balanceado en cada nodo, no solo en la raíz)

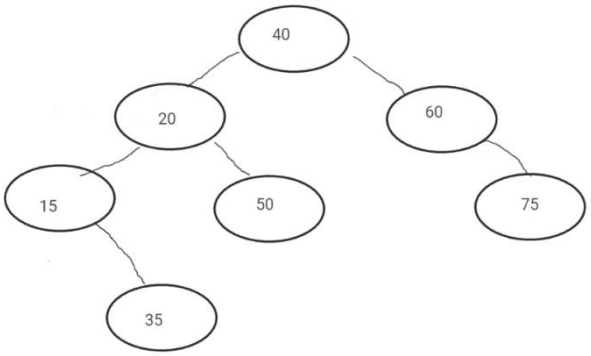
Supongamos este árbol:



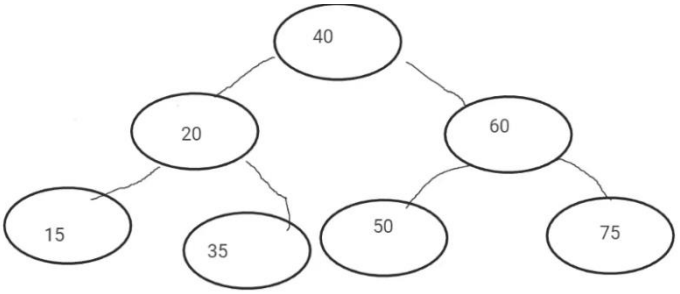
Los **nodos críticos** (que no están balanceados) son el 60 (5 nodos a izquierda != 1 a derecha) y el 40(3 nodos a izquierda != 1 a derecha)

Me fijo quien debe subir a la raíz para que quede balanceado. En este caso 40 (tendrá 3 nodos a izq (menores a 40) y 3 a der (mayores a 40))

Paso el 40 a la raíz. el 60 será hijo derecho (es mayor a 40):



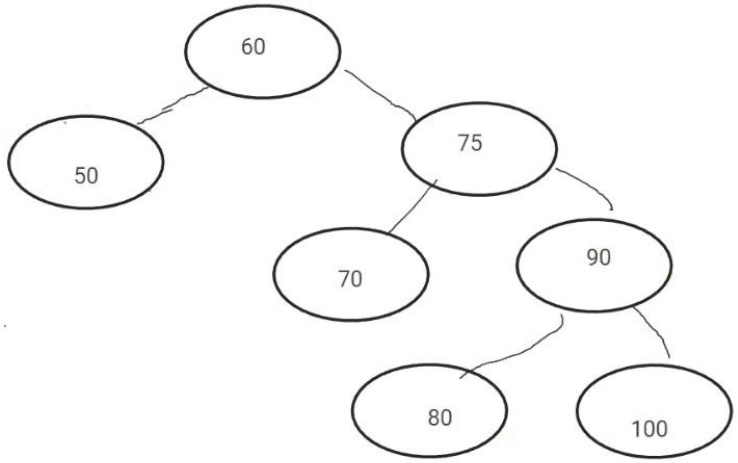
El hijo derecho del hijo izquierdo, pasa a ser el hijo izquierdo del derecho:



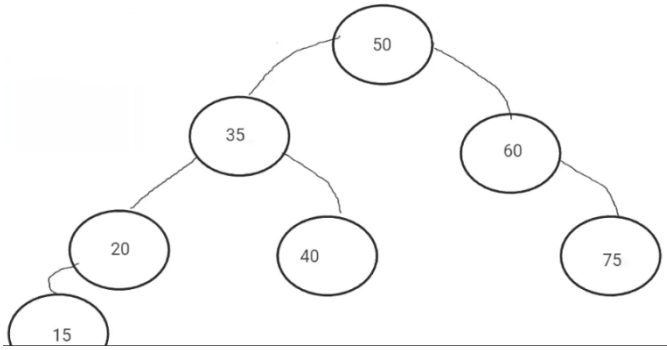
Finalmente me queda el árbol balanceado

Rotación a izquierda:

Lo mismo pero a izquierda xd



Otros Casos:

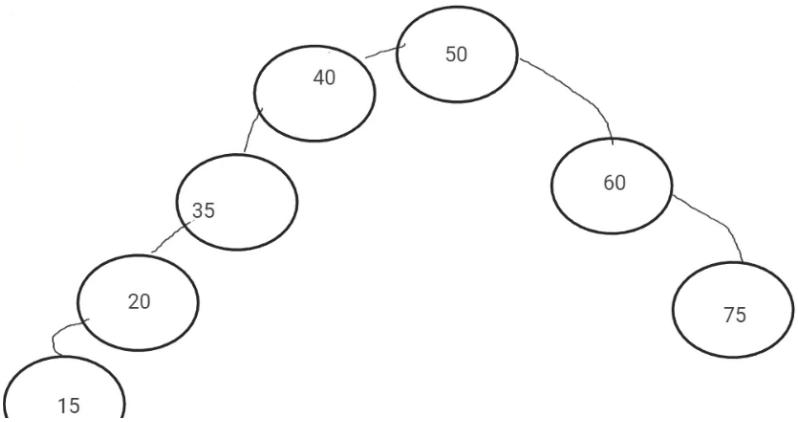


En este caso el árbol no se encuentra totalmente

Desbalanceado a izq o der (tomara más tiempo balancearlo)

Valor medio(nueva raíz para balancear el árbol): nodo 40 → debo correr el 35, y luego el 50 (antes solo debía correr un nodo, ahora dos)

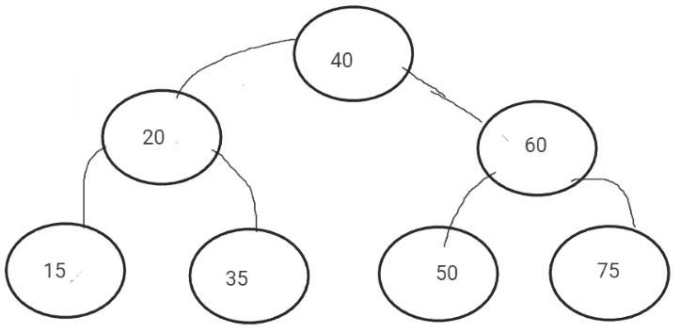
Corro el 35 y subo el 40:



Notar que ahora desbalancee un poco más el árbol, pero es necesario momentáneamente para finalmente balancear todo el árbol)

Procedo a balancear cada nodo (se pueden hacer en paralelo)

Resultado final:



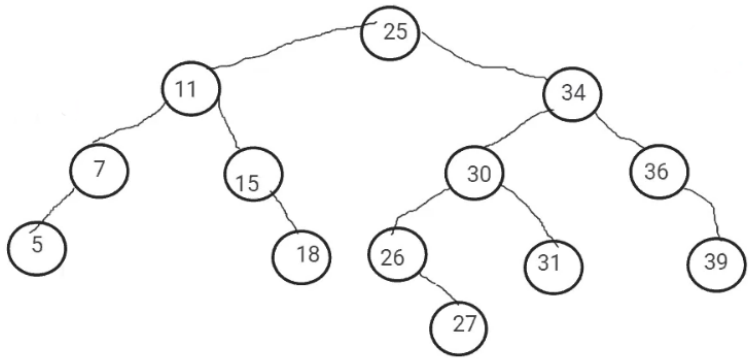
Mantenimiento:

Alta Arbol binario búsqueda:

Simplemente barrer el árbol hasta llegar a la posición del nuevo nodo y colocarlo, se modifican los atributos necesarios

Baja Arbol binario búsqueda:

Ejemplo: Baja de 5, 15 y 25 para este árbol AVL:



BAJAS → 3 casos posibles:

- Nodo sin hijos
- Nodo con un hijo

- Nodo con 2 hijos

1. Encontrar el nodo (lo mismo de siempre)
2. Se analiza en cuál de los 3 casos se encuentra ese nodo a eliminar:
   Tiene los dos punteros (izq. y der.) en NULL (no tiene hijos), no hay nada que reasignar: el 5 no tiene hijos, lo elimino directo
   Tiene un hijo: hay que resignar esa rama (el 15 tiene un hijo (18)
   Tiene dos hijos: hay que reasignar las dos ramas (el 25)
3. Se barre el árbol para cada nodo a eliminar, mientras se guarda en variables los padres de cada nodo

Nodo 5:

1) Bajo hasta el 5, pongo el puntero del hijo izq de su padre(7) en NULL →elimino el 5

Nodo 15:

1) Bajo hasta el 15: Hago que el hijo derecho del padre del 15 (11) ahora apunte al único hijo de 15(18) →elimino 15

Nodo 25:

1) El FRONT apunta al 25: tiene dos hijos
2) Bajo por la rama derecho y busco el menor nodo de toda la rama para que sea la raíz(ya que será mayor a toda la rama izquierda, pero menor a la rama derecha)
3) Piso el nodo a bajar con el nodo menor encontrado con todos los atributos 26(por un momento lo tendré repetido)
4) Ahora a partir de la rama del 34, hago la baja del 26 (tendré si o si o el criterio de sin hijos o un hijo máximo)

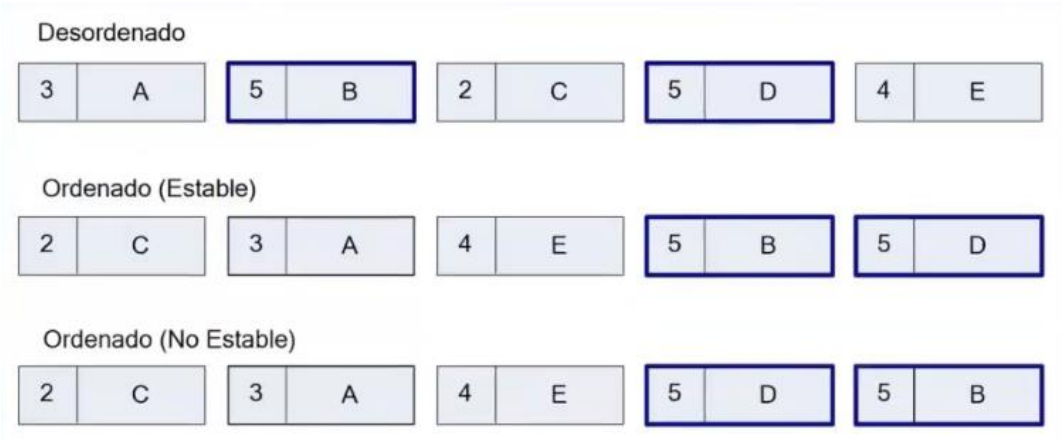
Métodos de clasificación/ordenamiento

- **Objetivo**: Dado un conjunto de valores desordenados del tipo {a1, a2, an}, devolver un conjunto ordenado de menor a mayor o de mayor a menor
- En la práctica muy pocas veces se ordenan números aislados. Estos números en realidad son las keys de los registros de una tabla
- Clave (valor a ordenar) + dato satélite (asociados a la key) = Registro

Clasificaciones de ordenamientos:

Estable: Ante registros con claves/keys iguales, se respeta el orden que tenían originalmente

No estable: no se garantiza lo que hace el estable (pero tampoco lo da vuelta a propósito, solo no lo garantiza)



In situ: Los métodos in situ son los que transforman una estructura de datos usando una cantidad extra de memoria, siendo esta **pequeña** y **constante**.

No in situ: requieren gran cantidad de memoria extra para transformar una entrada

La performance de algoritmos **in situ** es **mucho mayor** que las no in situ

Ejemplo in situ: uso simplemente un entero como variable para ordenar (invertir en este caso) un array

```
void invertirArray(int[] a)
  { int[] aux = new int[ a.length ];
    for(int c = 0; c < a.length ; c++)
          { aux[c] = a[a.length - c - 1];}
    a = aux;}

void invertirArrayInSitu(int[] a)
 {  int temp;
    for(int c = 0; c < a.length / 2; c++)
          { temp = a[c];
                a[c] = a[a.length - c - 1];
                a[a.length - c - 1] = temp;}}
```

Método interno: Si el archivo a ordenar cabe en **memoria principal,** entonces el método de clasificación es interno

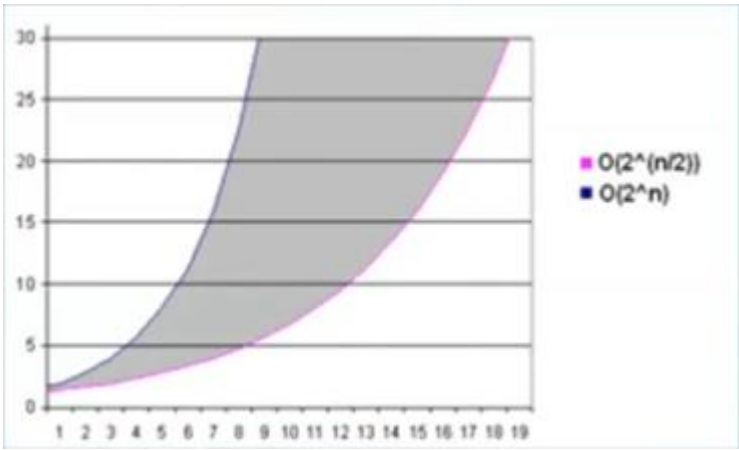
Método externo: Si ordenamos archivos desde un disco u **otro dispositivo** que no es memoria principal (generalmente volúmenes de info y tiempos de acceso mucho más grandes que los que ordena el método interno)

Complejidad computacional de un algoritmo de ordenamiento: cantidad de lecturas de claves necesarias para llevar a cabo determinado algoritmo

X: tamaño del vector

Y: cantidad de lecturas

Complejidad computacional = O(función), donde "función" es la función matemática que acota el comportamiento del algoritmo en función del tiempo y la cantidad de elementos



Como se evalúa la complejidad de un algoritmo:

- Una computadora solo puede realizar **operaciones matemáticas** y **comparaciones** (por estar hecha de electrónica booleana)
- Las **comparaciones** (if) pueden llegar a ser hasta **100 veces más lenta** que una operación matemática básica
- Por eso, para evaluar la complejidad de un algoritmo, se evalúa la cantidad de comparaciones realizadas

Métodos:

Para facilitar los ejemplos, se ordenan vectores. Recordar que en realidad se ordenan las claves de los registros (consecuentemente también sus datos asociados)

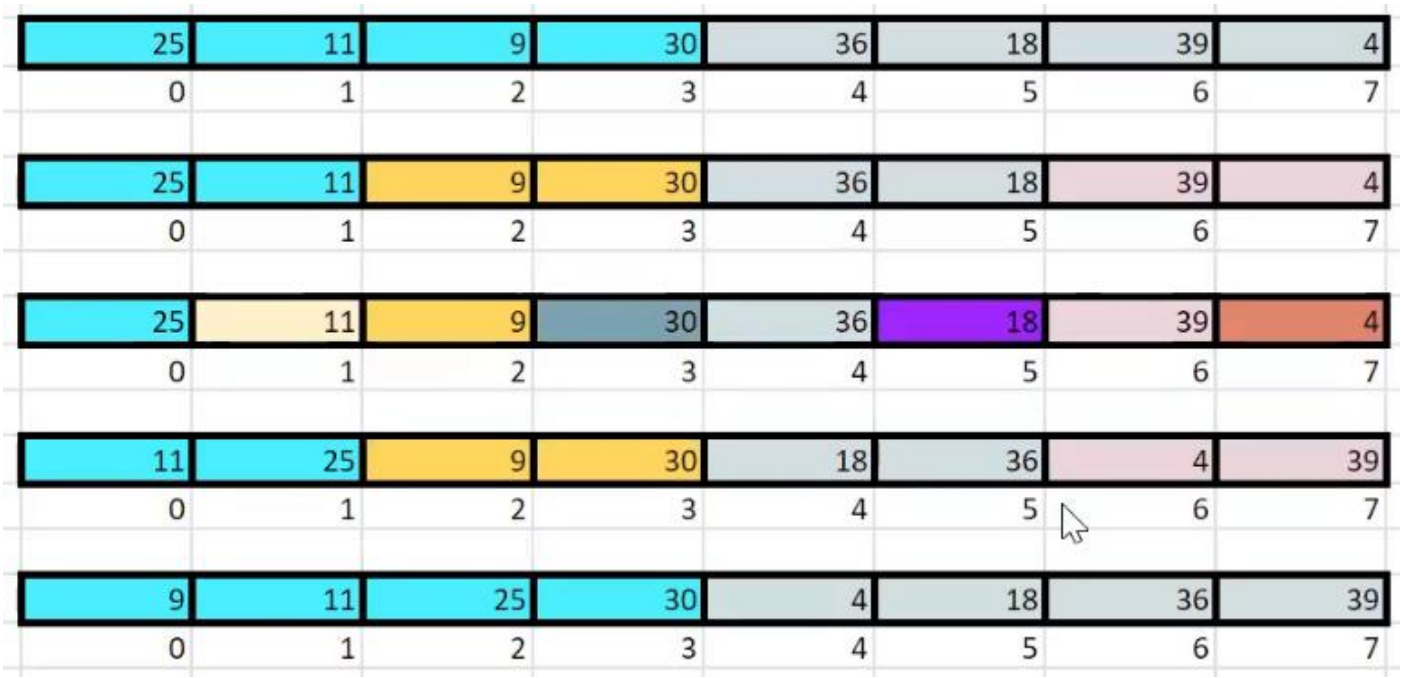
Los de mejor performance son: Merge Sort, Quick Sort y Heap Sort

Merge Sort:

Recursivo

Partimos de la base de que tendremos 2 o más conjuntos de datos, donde cada uno estará ordenado

1. Divido el vector hasta tener divisiones de un elemento (premisa de que un conjunto de un único elemento ya está ordenado)
2. Ordeno cada parte
3. hago el merge entre las partes ordenadas hasta llegar a un merge de todo el vector



1. Divido hasta tener divisiones de un elemento c/u
2. ordeno con la siguiente división (25 y 11), repito lo mismo para las siguientes dos divisiones(9 y 30) (se puede hacer en paralelo, (36,18) y (39, 4))
3. Hago el merge y lo ordeno (11,25,9,30 = 9,11,25,30) →en paralelo ordeno el otro merge

Inserción:

- Iterativo (un for adentro de otro)
- Emular la forma en que una persona ordena un conjunto de cosas
- También parto de la premisa de que un conjunto de un solo elemento esta ordenado
- Inserto de a uno, con cada inserción ordeno/intercalo donde vaya

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	
0	1	2	3	4	5	6

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

Selección:

Iterativo

Recorro todo el vector, busco el menor, lo llevo al principio → N veces (baja performance)

Por cada paso, se deja un elemento en su lugar definitivo (lugar predefinido de antemano, a la izq de todo)

Se guarda en variables:

- posición mínimo
- clave del mínimo

25	11	30	7	36	18	32		
0	1	2	3	4	5	6	pos min	1
							clave minim	11
7	11	30	25	36	18	32		
0	1	2	3	4	5	6		

(Lo mismo para el resto del vector)

Quick Sort:

Recursivo

Es mas performante si se aplica a datos desordenados, y menos si los datos ya están ordenados

A diferencia de **selección** ahora no predefino de antemano el lugar, sino que la **clave**

Tengo las variables izq y der, en los dos extremos del vector →debo compararlos para ver cual queda en su pos definitiva, el otro (guardado en una FLAG) será comparado en el paso siguiente (por única vez)

ej: FLAG = der

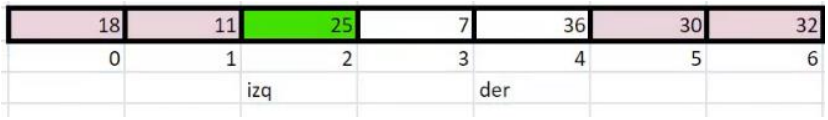
Comparo las claves de los extremos → 25 < 32, por lo tanto:

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

18 < 25: cambio el 18 al 25, y cambio el FLAG

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

11 < 25, y está a la izq: quedan en su lugar, corro el izq



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

30 > 25, pero 30 está a la izq, los intercambio, muevo der a la izquierda, cambio FLAG a der:

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

7 < 25, pero está a la derecha, intercambio y cambio flag:

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

Ahora izq = der = pos 3 →esa clave queda en su posición definitiva

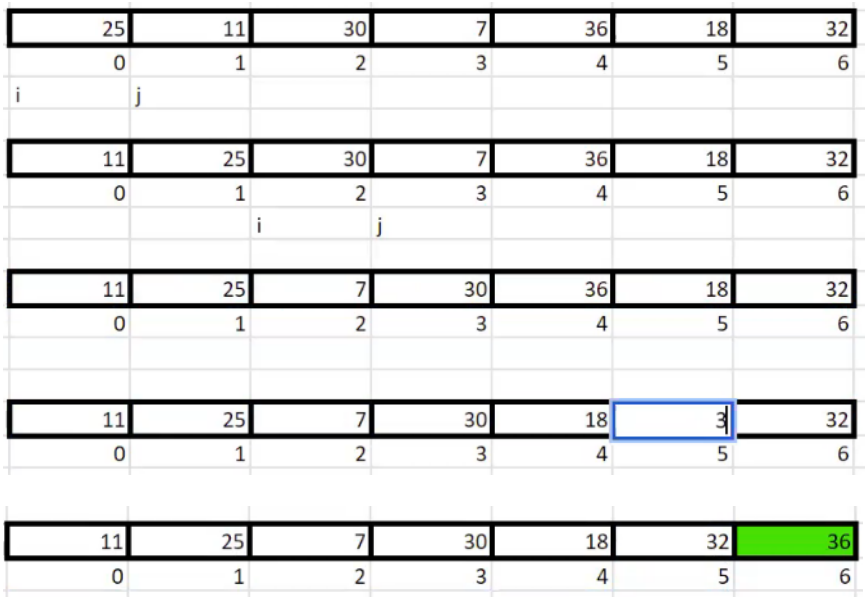
Ahora se ordena por un lado el vector izquierdo a 25 (18,11,7) y por otro el derecho (36,30,32)

Burbujeo (Bubble Sort)

Comparo de a posiciones adyacentes, de izquierda a derecha

Al finalizar, voy a encontrar el mayor a la derecha de todo (y el resto del vector por ordenarse)

FLAG (hubo intercambio) → cuando recorro todo y el flag = false: termine de ordenar



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

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

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

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

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

Repito N veces

Los siguientes dos algoritmos son para una volumetría de datos mayor:

Shell Sort:

Pensado para ordenar volumetría grande

Ejecuto un algoritmo de **prearmado** a los datos, para luego aplicar **inserción** y la cantidad de desplazamientos sea muy pequeña.

Los valores predefinidos para ir subdividiendo el vector (distancia) son 5 y 3 y 1(es inserción básicamente)

EJ Distancia = 5:

- Agarro los valores cuyas distancias sean 5 y los ordeno
- En este caso empiezo con 12, 25 y 45 (ya están ordenados)
- Paso al 15, 59 y 39 → los ordeno pero en su respectiva posición de distancia 5 desde la pos del 15: 15, 39, 59
- Lo mismo con todo el vector, y lo mismo en caso de distancia = 3

Ordenado con distancia de 5

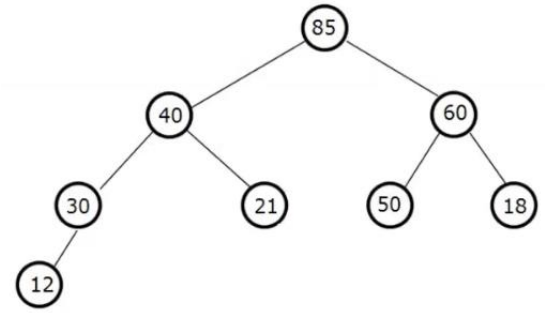
12	15	96	33	21	25	93	65	23	45	29	41	39	17	59
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	15	39	17		25	41	65	23		29	93	96	33	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Heap Sort:

Trabaja sobre **vectores** que representen **arboles binarios**

Árbol cargado por niveles: primero la raíz, después todo el nivel 1 cargado de izq a derecha, después todo el nivel 2 cargado de izquierda a derecha, etc… (Siempre binario)

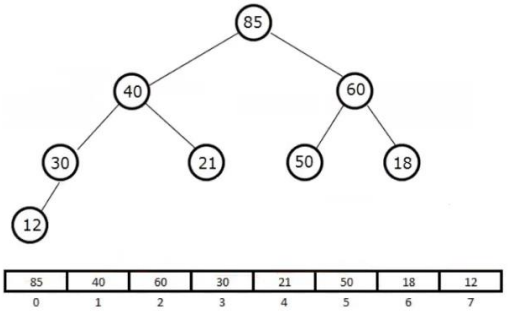
Árbol orden parcial: todos los padres son mayores a su subárbol (un sobrino puede ser mayor a su tío (50>40), pero nunca un hijo > padre)



Árbol implementado en vector:

Como se vio antes, se implementa con un barrido por niveles usando la siguientes formulas

- Hijo izq: ((i+1) * 2) – 1
- Hijo derecho: (i+1) * 2
- Padre: int[(i-1)/2]



Algoritmo Heap Sort:

Recursivo y **recurrente** (tiene que esperar a que termine un paso para arrancar el otro)

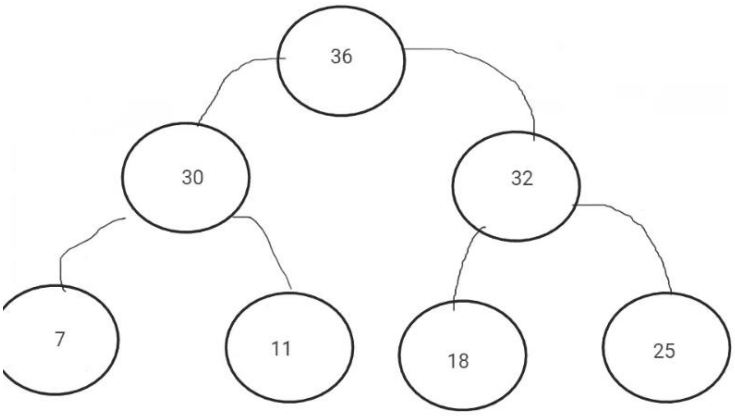
Contra: como dice arriba, no se puede trabajar en paralelo

Utiliza el **árbol cargado por niveles** (recordado arriba), de **orden parcial** e **implementado en un vector** (recordar que el árbol en si es una abstracción/no existe)

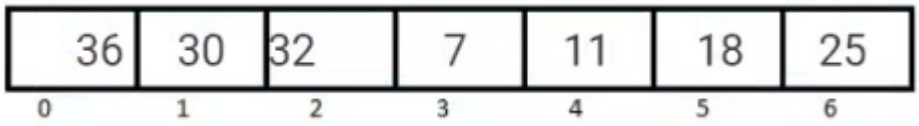
1. Cargo el árbol pre-armado (1 sola vez)
2. ordenamiento (n-1) veces (mucha performance para mucha volumetría)

Ejemplo: 25, 11, 30, 7, 36, 18, 32

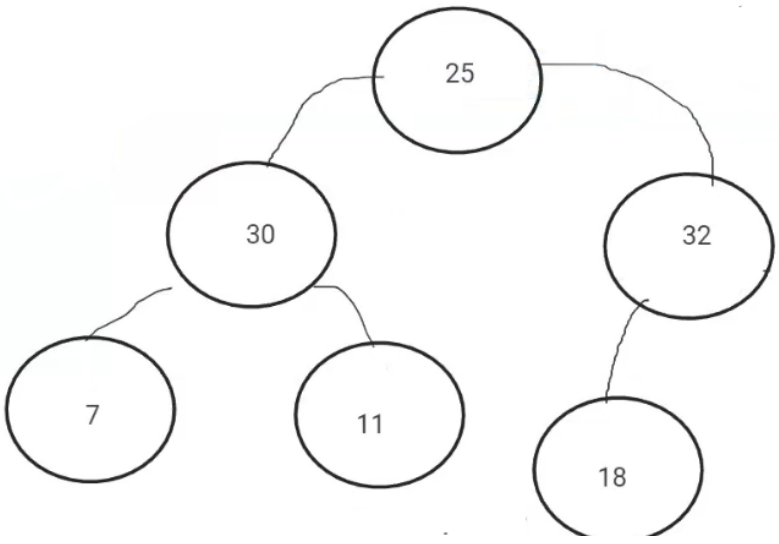
Cargo árbol por niveles:



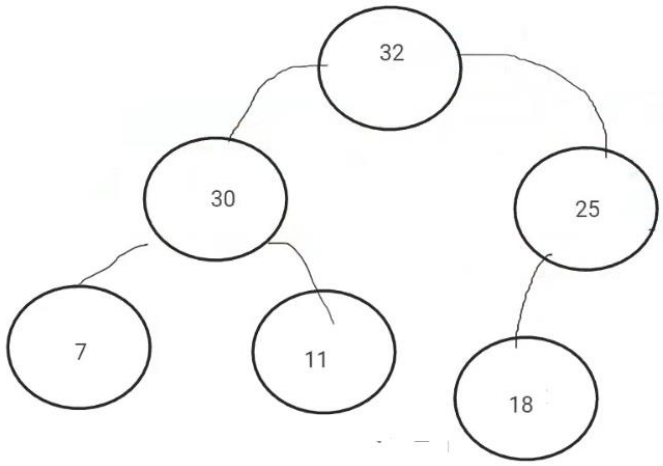
El árbol no existe, pero es una abstracción del vector, el cual queda así:



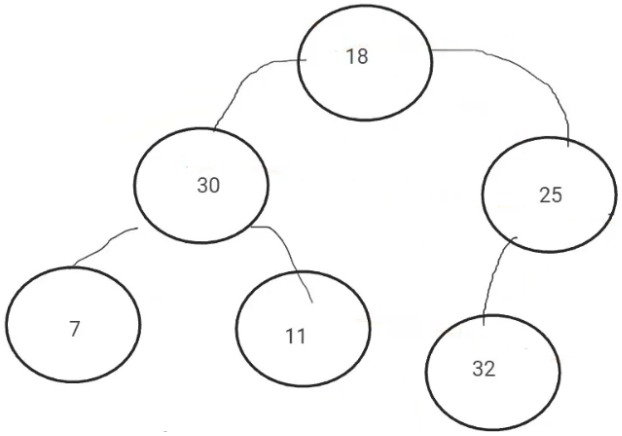
2) ordeno (en el árbol)
   36 > 25 → paso 25 a la raíz, saco el 36 del árbol



Comparo con nivel 1: 25 < 30 y 32. 32 > 30:



32 > 18:



32 ya es pos ordenada, lo elimino

sigo hasta ordenar

Comparaciones:

- **Bubble Sort(burbujeo)** :El más lento de todos (uso pedagógico nada mas)
- **Selection Sort:** Otro muy lento, uso pedagógico
- **Inserción**: Conveniente cuando sabemos que el vector está casi ordenado en el mismo criterio que queremos ordenar
- **Quick sort**: El más rápido en la practica
- **Heap** o **Shell Sort**: muy utilizado en grandes volúmenes de datos
- **Merge**: el más conveniente en cuanto a estabilidad y trabajos en paralelos

Acceso a datos:

Elementos en la estructura (lo que tiene el contenido):

- **Claves** (por las cuales de indexa/ordena)
- **Punteros** (direcciones de memoria o disco donde comience la instancia del registro que se va a buscar)

Colisión

Situación en la que dadas dos claves distintas, obtengo la misma posición al aplicar la función de hash

La forma en que se administra esa colisión (elegir una de las dos clave-puntero para esa posición) varía

> **Función rehash:** En vez de recibir la clave como dominio (F(clave)), recibe la posición, y el codominio (resultado) será una posición valida del vector. (ej: hay colisión en la pos 44, le paso el 44 a la función de rehash y me devuelve por ejemplo la pos + 1 = 45)

Luego al querer acceder a ese dato alojado mediante X rehashes, se generara la misma colisión, y se aplicara la función rehash X veces para ubicar su posición real

Tipos de acceso:

Hashing y **Arbol B (la estructura mas utilizada por motores DB para crear índices)**

Hashing

Muy performante para trabajar en una **única** clave y que **no sea** por **rangos** (buscar claves entre 40 y 50). Una vez que encontré una clave, no tengo forma de acceder a las adyacentes.

No me permite hacer búsquedas por más de una clave

Tiene dos maneras de trabajar:

- **Estático** (Direccionamiento abierto)
- **Dinámico** (Encadenamiento)

Ambas tienen un conjunto de datos, y cada uno tiene asociada su key (a ordenar)

Estático:

- **Tabla** de **hash** (matriz común y corriente de tamaño N definido e inmodificable)
- Dada una determinada clave, le aplico una función de hash de tal forma que el codominio (resultado) de esa función de hash sea un número valido entre 0 y N (una posición valida de la matriz).
- En esa posición, es donde encontraremos la **clave** y el **puntero**

Objetivo:

Armar la estructura de BD de tal manera que, una vez cargadas todas las claves, la función de rehash se aplique la **menor** cantidad de veces posible

Para eso:

- Definir un **tamaño** correcto a la tabla de hash: según criterio
- Definir que **función de hash** que tenga como objetivo que el rehash se aplique a la menor cantidad de claves posibles
- Definir una buena **función de rehash**

Para esto existen ciertos criterios:

Tamaño:

A priori: **1er numero primo > cantidad de claves** (ej 300 claves →Tamaño = 307)

Cuanta más dispersión queramos, será 2do numero primo > cantidad de claves, 3ro, etc…

Función de Hash:

Estas dos, estadísticamente son las que más dispersión logran para un conjunto de claves

Dobles:

Solo para claves numéricas

1. Dividir la clave en 2 (colocando ceros a la izq. de ser necesario para que queden en igual cant de dígitos)
2. Pasar a binario
3. Suma XOR
4. Pasar a decimal
5. Aplicar modulo (función modulo tipo discreta, para que quede un valor entre 0 y N(tamaño tabla hash))

EJ: 2967 en tabla de tamaño = 307

29 = 0011101

67 = 1000011

Los sumo con XOR: 1011110 = 94(307) = **94**

Cuadrado Medio:

Solo para claves numéricas

1. Elevo la clave al cuadrado
2. Toma los valores medios
3. Aplico modulo (igual que en dobles)

Ej: 2869 en tabla de tamaño = 307

(2869)^2 = 8231161

La pos más grande de la tabla es la 306 →3 dígitos

Agarro los 3 dígitos del medio del 8231161, tal que a la izq y a la derecha de esos 3 dígitos, queden la misma cant de dígitos, en caso de no lograrlo, se agregan ceros a la izquierda hasta lograrlo:

82 311 61

311 > 306 → 311(307) = **4**

Ej: clave 2 en misma tabla

(2)^2 = 4

La pos más grande de la tabla es la 306 →3 dígitos

_ 004 _ → 4(307) = **4** → habrá colisión entre 2869 y 2

Función de Rehash:

- Asegurar encontrar un lugar vacío lo más posible (luego de una colisión)
- A una clave que generó una colisión, le sumo un numero primo y le aplico el modulo del tamaño (otro primo), miméticamente logramos que, repitiendo esto tantas veces como el tamaño de la tabla)

Bajas:

Existe la baja lógica (No se borra el contenido, simplemente se borra la clave, y se usa un FLAG que indique que ese registro esta libre)
Cuando se de una **alta** en ese registro, simplemente se pisa el FLAG, se pone la clave nueva y se pisa el puntero viejo

Dinámico:

Se usa cuando partimos con una determinada cantidad de claves, pero a futuro puede llegar a aumentar (Como en los índices de una DB relacional)

Parecido a Estático, con la diferencia de que **no hay función de rehashing**

Arbol B

Permite **claves duplicadas** y **búsqueda por rangos** (clásico between)

Arboles M-arios (siendo M el grado que va entre mínimo 50 y máximo 2000)

Cuando se implementa?: Se implementa el árbol de búsqueda en un **disco** o **almacenamiento secundario**, cuando se tiene un conjunto de datos tan grande, que no se pueden colocar en memoria principal

Las características de un disco a comparación con memoria principal (tiempo de acceso), hacen que sea necesario utilizar valores de M más grandes para que la implementación de los árboles en disco sea eficiente

Arbol B

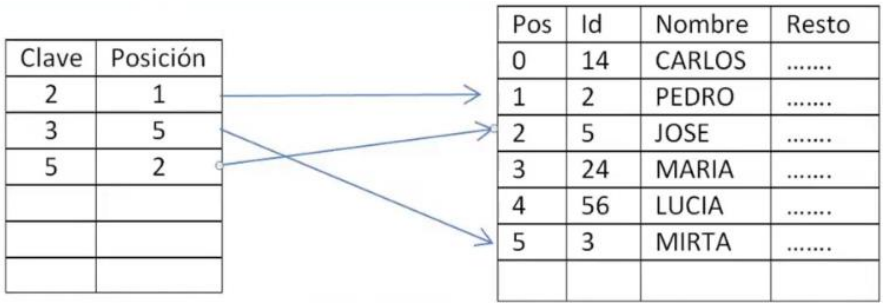
Es un tipo de árbol M-ario destinado a **la creación de índices físicos** para el **acceso** a la **información** en DB relacionales

Objetivo: minimizar las operaciones de entrada y salida 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 $O(\log_x(n))$ →n = cant. De estructura/cantidad de registros y x = M

Es de los pocos arboles con dos tipos diferentes de nodos: **raíz/rama** y **hoja,** c/u con su respectiva estructura:

- El nodo raíz o rama tendrá punteros a sus hijos
- El nodo hoja al no tener hijos, tiene punteros a los datos

Nodo hoja: Tiene una componente de datos donde van los valores de las claves ordenadas de menor a mayor y un componente puntero que contiene la posición relativa de los datos secuenciales correspondientes a esa clave

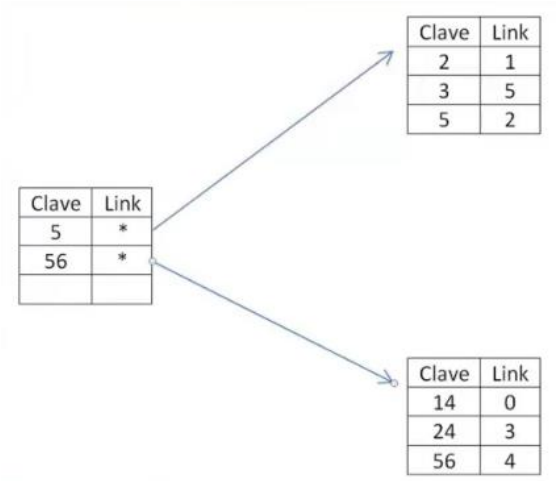


Clave	Posición
2	1
3	5
5	2

Pos	Id	Nombre	Resto
0	14	CARLOS	.....
1	2	PEDRO	.....
2	5	JOSE	.....
3	24	MARIA	.....
4	56	LUCIA	.....
5	3	MIRTA	.....

pos = “row id” → posición física en el disco donde se encuentra almacenado ese registro

Nodo raíz o rama: Tiene una componente de datos donde van los valores de las claves ordenados de menor a mayor y un componente puntero que apunta al nodo que contiene claves menores o iguales que ella



Se usan 3 como ejemplo, pero mínimo son 50 y máximo 2000

Búsqueda en Arbol B:

Es muy parecido a buscar en un árbol binario de búsqueda, excepto que en vez de hacer una decisión binaria (o de dos caminos en cada nodo), hacemos una decisión multicamino en base al número de hijos del nodo

Inserción en Arbol B:

1. Se comienza por la raíz y realizamos una búsqueda del elemento X a insertar (asumiendo que el elemento no está en el árbol)
2. La búsqueda sin éxito terminara en un nodo hoja → ese es el punto en el árbol donde X va a ser insertado si hay lugar

Si cuando se llega a la hoja no hay espacio para insertar el nodo, se produce un **Split**

Split: 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 otra mitad (de los más grandes) en el otro

Eliminación en Arbol B:

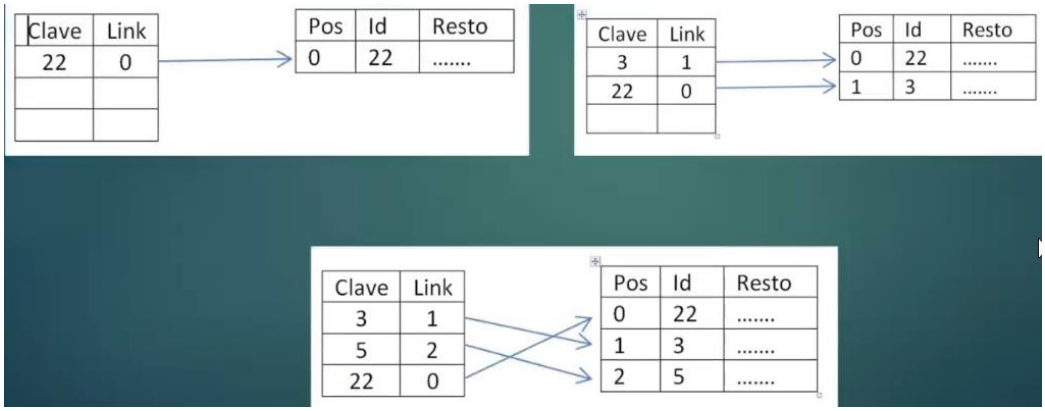
1. Comenzamos en la raíz y realizamos una búsqueda para el (asumiendo que el elemento existe en el árbol)
2. Si existe se llega a la hoja donde está y se borra, de lo contrario se dirá que no existe
3. Si ocurre que cuando se elimina el elemento X, el nodo queda vacío (es decir, acabamos de eliminar la última clave del nodo), debe eliminarse el nodo, lo que puede generar una baja potencial en todos los antecesores de dicho nodo

Ejemplo armado(inserción), **búsqueda** y **eliminación** de elementos de árbol B:

dado un conjunto de claves: 22-3-5-11-23-54-10-14-15-7-3-9 →hay claves duplicadas, como el 3

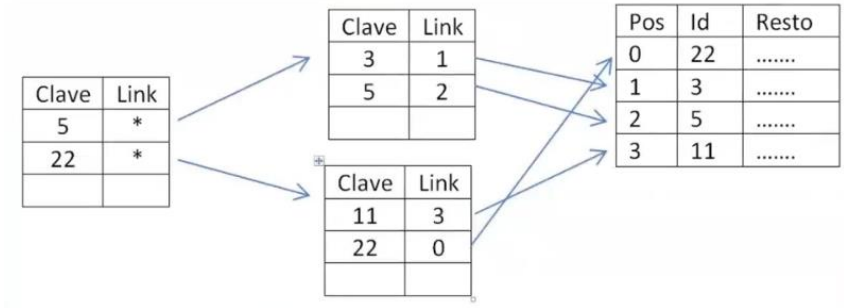
Aunque el grado mínimo es 50, vamos a hacer el ejemplo para un árbol de grado M=3

1. Como M = 3, tomo los 3 primeros y los ordeno de menor a mayor



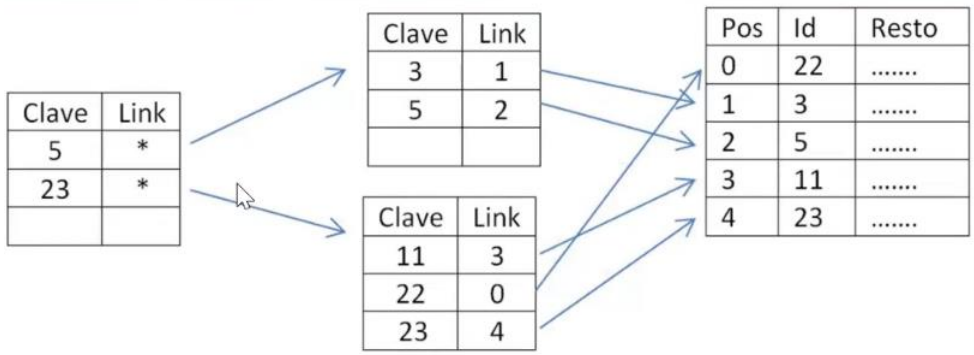
link = row id (pos en disco)

2. Cuando quiera ingresar el 11, al no haber lugar para ingresarlo, hago un **Split** → las dos menores (3 y 5) van a un nodo, y las dos mayores (11 y 22) a otro:
   Antes teníamos un solo nodo (no hacía falta un padre), ahora son 2 y tenemos que agregar el nodo padre
   Al nodo padre le agregamos las claves de los últimos atributos de cada nodo hijo:

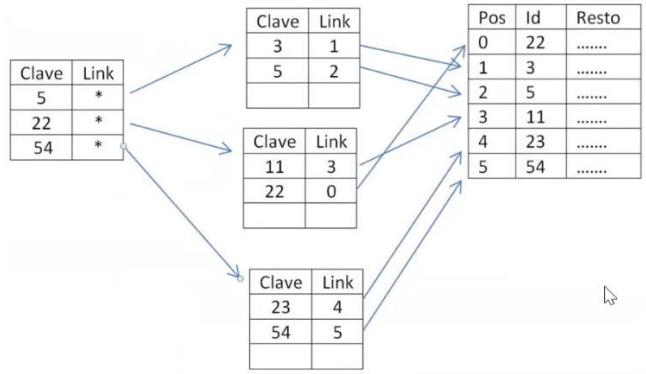


   Cada puntero(link) del padre apuntara a un subárbol que contenga a todas las claves menores o iguales a él. ejemplo el 5 apunta al subárbol de las claves <= 5 → 3 y 5

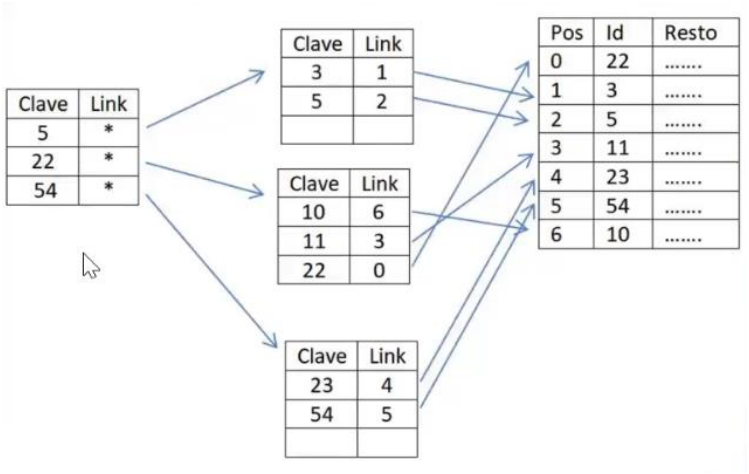
3. Quiero insertar el 23, al haber lugar simplemente se inserta pero se debe modificar la clave del nodo padre, que ahora en vez de tener como clave al 22, tiene al 23 (por ser el nuevo mayor):



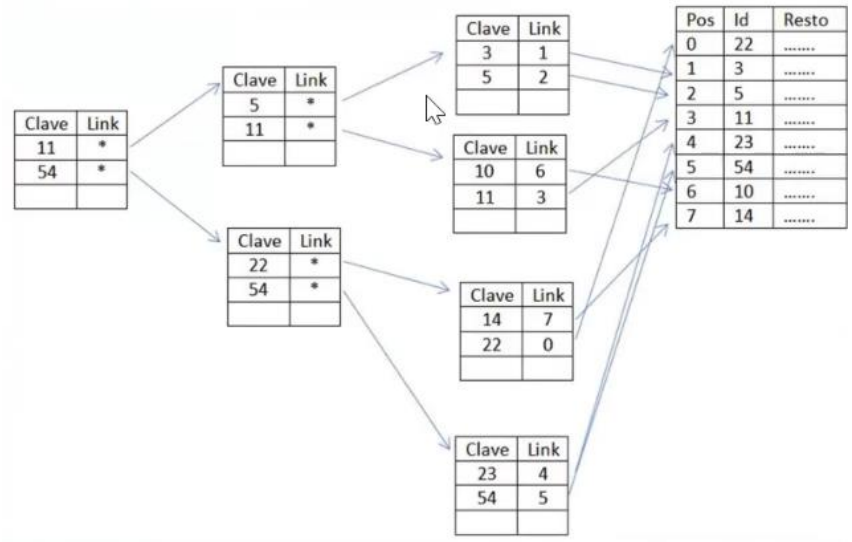
4. Quiero insertar el 54, no entra en su respectivo nodo por lo que debo hacer un **Split** → Un nodo tendrá los dos menores (11 y 24) y el otro los dos mayores (23, 54). Además agrego la clave y su puntero al nodo padre:



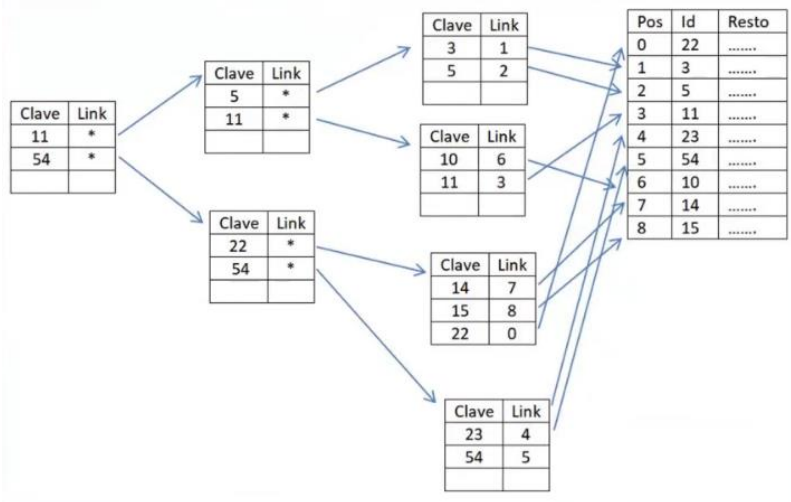
5. Quiero insertar el 10, su nodo es el del medio, al haber lugar lo inserto:



6. Quiero insertar el 14, iría también en el nodo del medio, no hay lugar →hago **Split** de ese nodo
   Ahora tengo 4 hijos (no entran en el nodo padre) → hago **Split** del padre:

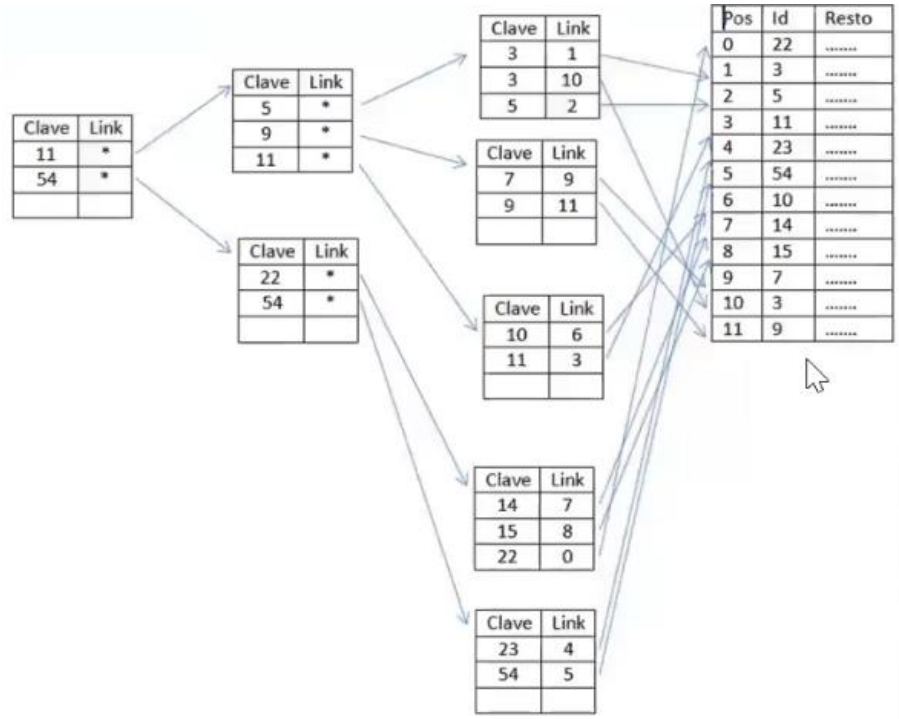


7. agrego resto de valores hasta el 9:



8.

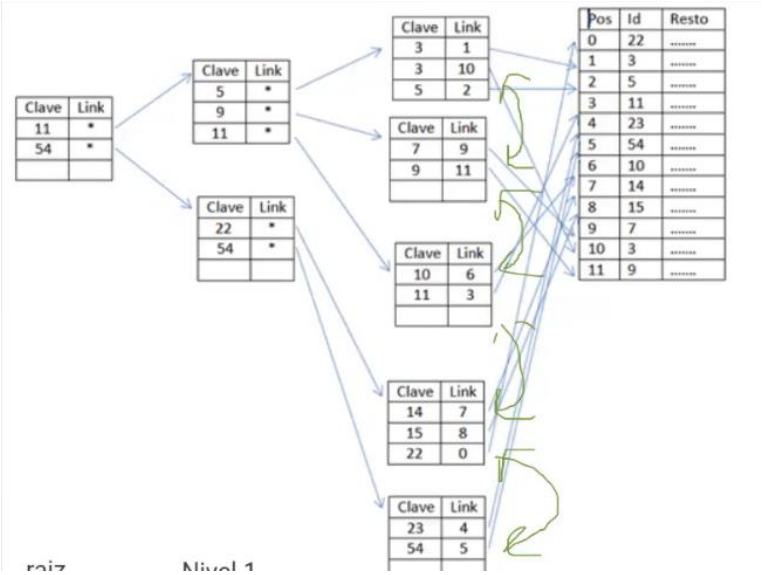
   para ubicar al 9 (como a todos los demás), parto desde la raíz:
   9 <= 11 →voy al nodo de claves 5 y 11
   9 > 5 y <= 11 →voy al nodo de claves 7, 10 y 11
   No hay lugar → **Split** y agrego le clave en el nodo padre**:**



Notemos que todas las hojas están en el mismo nivel (2)

Como el árbol-B permite claves duplicadas y búsqueda por rangos, voy a necesitar:

punteros para pasar de una hoja a otra (al anterior y al siguiente)



Implementación de queries con este árbol de ejemplo:

Importante: acá son pocos nodos, pero en caso de muchas claves (lo común) se realizan búsquedas binarias para ir ubicando los nodos

Indices:

El objetivo es acceder de forma mas rápida a los datos almacenados en las tablas

Son estructuras opcionales asociadas a estas que son independientes física y lógicamente de los datos almacenados

Se pueden crear distintos tipos de índices sobre uno o mas campos

Parámetros de los índices:

Load/fill factor: parámetro que indica, al momento de crear un índice, el porcentaje de M para el cual la cantidad de nodos se va a llenar, en vez de hasta el máximo (me aseguro siempre tener claves disponibles para evitar **splits,** ya que los splits son muy costosos en cuanto a performance de índices)

Esto lo puedo implementar para **tablas transaccionales** y cuando las claves **no son incrementales**

Tipos de índices:

- hashing
- árbol B+
- B tree cluster index
- Bit map index

Características de los índices:

- claves únicas o duplicadas
- 1 campo o compuesto

Transacciones:

Transacción: conjunto de instrucciones que cumplen con las propiedades **ACID**

- **A**: **Atomicidad** (o se confirman todas las instrucciones[commit] o se vuelve para atrás[rollback])
- **C**: **Consistencia** (dos get en el mismo momento obtengan lo mismo), ejemplo cajeros cant. plata
- **I**: **Isolation(aislación)** (Como los procesos cooperan entre sí para acceder a los datos compartidos (datos/tablas)
- **D**: **Durabilidad** (queda persistido en el tiempo, contraej: las cosas en memoria no son durables)

begin transaction

select …

set @...

delete …

insert …

commit (confirmar/persistir) **|| rollback** (volver hacia atrás/los insert y los delete no persistan en la DB)

Cuando se ejecuta el begin transaction, se crea la tabla con las modificaciones, y a cada cambio se le asocia la tabla UNDO por si se ejecuta el rollback

Si se ejecuta un commit, simplemente se borra la tabla UNDO

ej:

TRX 1

1	NUEVO	TRX1
2	NUEVO 2	TRX 1
	VIEJO 3	
4	VIEJO 4	

```
BEGIN TRANSACTION
  UPDATE CLIENTE SET NOMBRE = 'NUEVO'
WHERE CODIGO = 1

UPDATE CLIENTE SET NOMBRE = 'NUEVO 2'
WHERE CODIGO = 2

COMMIT
```

TRX1:

1	VIEJO	1
2	VIEJO 2	1

1	VIEJO	
2	VIEJO 2	
	VIEJO 3	
4	VIEJO 4	

Si un proceso está modificando un registro, y al mismo tiempo otro proceso quiere acceder a él, va a obtener el dato viejo o el nuevo dependiendo del nivel de aislación (**isolation**) configurado:

1. **caso**: tenés que esperar a que se termine de modificar para acceder o no (ejemplo estoy en centralpasajes rellenando datos de la compra de un pasaje, otro usuario no puede acceder al mismo pasaje)
2. **caso**: Lees el último dato confirmado
3. **caso:** lees el dato sucio (el menos frecuente, vas directo a la tabla)qsy

El nivel de aislación se configura con: set transaction isolation level [nivel]

el [nivel] por defecto es read commited →me lee el dato confirmado (o leo el último dato confirmado o espera a que se ejecute un commit o rollback)

Dato sucio: set transaction isolation level uncommited

dato repetible:(si tomo un dato para consultarlo, que otro no pueda acceder en ese momento a modificarlo, por ej mientras lleno los datos de un pasaje, que no venga otro a reservarlo más rápido) **set transaction isolation level REPETEABLE read**

Serializable: **set transaction isolation level SERIALIZABLE** (el más restrictivo, traba la segunda query que quiera hacer un insert luego de que una primera query tenga este nivel de aislamiento y haya hecho un select antes), para leer está todo bien, pero el problema es cuando uno quiere modificar y el otro leer

```
NIVEL
AISLAMIENTO/ PROBLEMA  |   DATO SUCIO | LECTURA NO REPETIBLE | DATO FANTASMA
READ UNCOMMITTED              SI              SI                   SI
READ COMMITTED                NO              SI                   SI
REPEATABLE READ               NO              NO                   SI
SERIALIZABLE                  NO              NO                   NO

-- SE PUEDEN DAR DEADLOCKS PRODUCTO DEL BLOQUEO COMPARTIDO,
```

Estructura DB

Componentes DB:

Datos:

- Están **integrados** →para minimizar redundancias (en un 80%/90% con la normalización)
- Están **compartidos** →concurrentemente pueden ser accedidos por múltiples usuarios

Tecnología:

- **Hardware** → discos
- **SO**

Programas/procesos:

DBMS (Data base managment system) →componente más importante de la DB, son todos los procesos y programas que ya vienen embebidos con el motor de la DB, manejan distintas funcionalidades como transacciones, niveles de aislamiento, concurrencia, creación y administración de distintos objetos, etc…

A estos programas/procesos que ya vienen embebidos, podemos agregarles los nuestros:

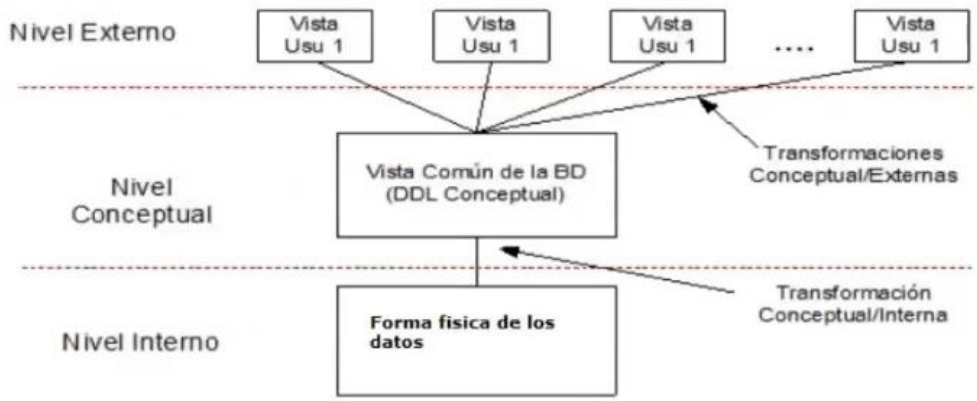
4. funciones
5. Stored prodcedures
6. Triggers

Usuarios/grupos de usuarios/roles:

- Programador de aplicaciones
- DBA → DataBase administrator
- Usuarios finales
- Otros perfiles funcionales

Estructura DB desde el punto de vista arquitectónico:

Arquitectura ANSI SPARC:



Capa externa: Las vistas que los usuarios tienen de los datos (a través de reportes, vistas, aplicaciones para ver en pantalla, etc…)

Capa interna: forma física de cómo están grabados los datos (paginación, distribución, nivel de raid, índices, tamaños de página, etc…)

Nivel conceptual: Tiene que, (en caso de consultas, insert, delete, etc…), hacer los **pasajes** del nivel **interno al externo** y **viceversa** (parecido a MVC)

Funcionalidades DBMS:

1. Diccionario de datos – METADATA
2. Seguridad – DCL →Lo que pueden y no pueden hacer los distintos usuarios
3. Integridad →de los datos
4. Durabilidad →resguardo de los datos (backup y restore)
5. Consistencia
6. Recovery
7. Administración física
8. Optimización de acceso a los datos
9. Logs, auditoria, chequeo de recursos

Diccionario de datos - METADATA:

Tablas y vistas del sistema, solo el **DBMS** las puede **modificar**. Nosotros (con los permisos correspondientes) solo las podemos **consultar**

Todo lo que sea objetos físicos (tablas (de esquemas por ej), vistas, snapshots, índices, stored procedures, etc…) tienen que estar guardados (en cuanto a su estructura) en la DB

Seguridad – DCL:

En SQL vimos

- DML (Data manipulation language) →manipulacion de datos
- DDL (Data definition language) → recuperación, insert, update y delete de datos

Ahora agregamos:

- DCL (Data control language) →creación, alteración y dropeo de objetos →permite identificar para cada usuario, que puede y que no puede hacer

Permisos de:

- **CREATE**: Apuntan a un **tipo** de objeto →Un usuario tiene permiso para crear ciertos tipos de objetos (vistas, tablas, etc…)
- **ALTER – DROP:** Para objetos en particular, y para todos los usuarios

para ciertos objetos:

- Tablas:

  Permisos de (además de CREATE y ALTER-DROP):

  - SELECT
  - INSERT
  - UPDATE
  - DELETE
  - TRUNCATE

- Vistas: mismas que Tablas menos TRUNCATE
  - SELECT (siempre se puede hacer)
  - INSERT (depende de cómo la vista este creada)
  - UPDATE (depende de cómo la vista este creada)
  - DELETE (depende de cómo la vista este creada)

Los permisos de un usuario en una vista, son solo para el objeto que ve

- Stored Procedures y Funciones:
  - Permiso de EXEC (ejecución)

un DBA puede dar permisos a un usuario, para que ese usuario de permisos a otros, en la sintaxis de DCL:

dar permisos: GRANT (cuál es el objeto al cual le damos permiso, a que usuario/s y tipo/s de permisos)

quitar: REVOKE (lo mismo que GRANT)

También puedo dar permisos generales:

SELECT ANY TABLE (permiso a hacer select a todas las tablas de la db (incluso las aun no creadas)

Integridad:

Reglas de negocio de los datos

1. Integridad de las entidades (PK)
2. Integridad referencial (FK)
3. Integridad semántica (significado de los datos)

Objetos usados para la integridad (ordenados por importancia):

1. **CONSTRAINTS**: ya que existen justamente y exclusivamente para asegurar integridad)
2. **Stored procedures** y **triggers**
3. **Vistas con restricciones** propias (VIEW WITH CHECK OPTIONS) → Cada vez que se haga un insert o un update sobre esa vista, los nuevos datos tienen que poder dar verdadero en la condición de WHERE de la vista (si es que la tuviera)

Durabilidad:

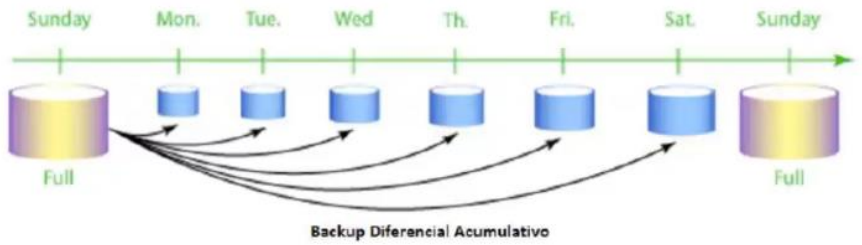
Backup: mapear la DB completa (objetos, usuarios, permisos, logs, etc…) TODO

Políticas de backup:

- **Backup full:**
  se backupea absolutamente todo

- **Backup diferencial acumulativo:**

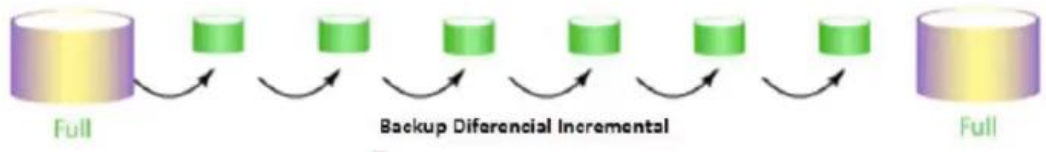
  Cada backup se hace con las novedades desde el **último backup full**, pisando a los anteriores



Backup Diferencial Acumulativo

- **Backup diferencial incremental:**

Cada backup se hace con las novedades desde el **último backup diferencial,** al llegar al siguiente backup full, se backupea pisando todos los anteriores



Restore: Operación inversa al backup

ejemplo se rompe el disco el jueves

- **Backup diferencial acumulativo**: se debe hacer un restore del ultimo full, luego un restore del ultimo diferencial acumulativo, y luego se cargan a mano los datos que se trabajaron en el día y se perdieron por la rotura (ya que no se habían backupeado)
- **Backup diferencial incremental:** Se debe hacer un restore del ultimo full, luego un restore del diferencial siguiente al full, luego del siguiente diferencial y así hasta llegar al último. finalmente cargar a mano los datos que se trabajaron en el día y se perdieron a causa de la rotura del disco

Hay casos en los que existen datos que **no pueden** perderse bajo ninguna circunstancia (operaciones de facturación bancaria por ejemplo) → usan las replicas

En el caso de estas operaciones que no pueden perderse, se manejan en transacciones distribuidas realizando replicas en otra base de datos externa. Ante cualquier inconveniente, se pueden recuperar los datos de ahí

Consistencia:

Cada transacción:

1. Parte de un estado valido de los datos (cumpliendo la integridad)
2. Termina en un estado valido de los datos (cumpliendo la integridad)
3. Los datos son vistos por cualquier usuario que los solicite

Cuando yo hice el commit (confirmé), cualquier usuario que haga un select debe ver los datos actualizados

Recovery:

Tiene que ver con la consistencia

Transacción de ejemplo:

BEGIN

SQL1

SQL2

SQL i →Se cae la DB

SQL N

COMMIT

Cuando se vuelva a levantar la DB, se va a ejecutar el **recovery** → Va a ir al rollback segment y va a ejecutar todos los SQL inversos de las sesiones que estaban con transacciones abiertas (sin confirmar) al momento de crearse la DB.

Cuando el **recovery** finaliza su ejecución, el usuario o DBA podrá acceder a los datos (select)

Administración física:

Los objetos de DB con contenido (tablas, índices y snapshots) se almacenan físicamente en **table spaces**.

Para una DB tengo definidos varios **table spaces** donde irán almacenados mis distintos objetos de esa DB

Los objetos se van a clasificar según las siguientes características para saber en que **table space** guardarlo**:**

- **Objetos de tamaño constante**
- **Tablas de tamaño incremental:** acá necesitamos funcionalidades de administración, para que el DBA pueda administrar y controlar que no se le vaya a llenar el disco y no enterarse o no tomar medidas preventivas → se le agrega una alarma que avise cuando quede menos del 20% de espacio libre para poder tomar acciones preventivas y así evitar el llenado de la DB (agregar disco, depurar tablas, etc…)
  Parámetros de las tablas:
  1. Table space
  2. Tamaño inicial
  3. Extensiones (que pasa cuando ese tamaño inicial se llena) →puedo definir N extensiones pero del mismo tamaño

Optimización de acceso a los datos:

Cómo el motor de base de datos resuelve un SELECT

Ese "cómo" es una funcionalidad del DBMS que es el **optimizador** (cada vez que se hace un select, calcula el plan de ejecución, es decir, la manera más óptima de resolver esa consulta)

En una **store procedure**, en su compilación, se guarda (entre otras cosas) el plan de ejecución. Luego cuando ejecutamos la stored procedure, no se tiene que recalcular dicho plan →a diferencia de un select SQL embebido

Logs, auditoria, chequeo de recursos

Ej: chequear cuantas sesiones están abiertas, hace cuanto, cuanto se está utilizando de memoria, cuanto se está utilizando de acceso a disco y como vino esa utilización en un determinado de tiempo anterior, etc…

Posibilidad de logear y hacer auditorias → sobre distintas ocurrencias en la DB

Objetos DB:

1. **Snapshot**:
   Es una vista materializada de objetos en una DB → Estructura de un select + los datos
   Cuanto alguien quiere obtener un snapshot, obtiene los resultados de ese select en su última actualización, en vez del actual como un select común (tabla temporal, cada X tiempo se actualiza) →las operaciones que tiene son **update** y **select**
2. **IOT (INDEX ORGANIZED TABLE):**
   Estructura donde el índice y la tabla es un mismo objeto (tiene la forma de un árbol B, y en vez de tener un puntero a los datos, tiene en ese lugar el registro completo) →Al tener esa estructura, la única forma de acceder a todos los datos es a través de esa única clave por la cual esa indexada
3. **Secuencias**:
   No en todos los motores de bases de datos la autonumeracion se hace a través de un identity o un serial, sino con objetos separados

Las secuencias me sirven para hacer más de una numeración a una tabla y para poder a llegar a numerar algún dato que no sea numérico sino alfanumérico (las facturas por ejemplo, que tendrán una numeración distinta según el tipo de factura y según el punto de venta)

4. **DBLINK**
   Link de una DB a otra (unidireccional)
   en la DB destino, voy a tener obviamente usuarios, los cuales tendrán como en cualquier db un usuario + password
   en la DB origen se configura/parametriza el DBLINK con ese usuario+password para poder acceder a ella
   Debido a que al acceder a otra DB pueden haber duplicidad en cuanto a los nombres de los objetos (como en el caso de las **REPLICACIONES (las tablas se llamaran igual)**, entonces en dblink, el nombre real de c/u de esos objetos será nombre_del_dbLink.nombre_de_la_tabla → es decir, la reconocerá con este nombre y no con el nombre original que se encuentre en la DB destino
5. **Sinónimos**
   Alias permanente a un objeto de base de datos para acortar el nombre en caso de una DBLINK o en caso de en un ambiente de desarrollo tener varios esquemas
6. **DTS (DATA TRANSFORMATION SERVICES)**
   Es la posibilidad de hacer y parametrizar un proceso para leer archivos y ponerlos en una proceso .bach que complete tablas o viceversa; en base a una tabla/conjunto de tablas, que generen un determinado archivo
7. **Package**
   Crea un paquete y dentro de ese paquete creo N stored procedures

DATAWAREHOUSE

Inteligencia del negocio:

- 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**
- Transformación de **datos** en **información**, y transformar la **info** en **conocimiento**. Con la intención de mejorar al máximo el proceso de toma de decisiones de la organización
- Un dato pasa a ser información cuando se lo vincula con alguna **relación**
- El dato es la **mínima unidad semántica** que se corresponde con los elementos primarios de la información (dato por sí solo no tienen ningún valor)

Los datos provienen de diferentes orígenes:

- **Interno**: de la propia organización
- **Externos**: extraídos del contexto (por ejemplo de la competencia)

A la vez pueden ser objetivos, subjetivos, cualitativos, cuantitativos

Información: conjunto de datos procesados o relacionados con un significado especifico

De qué manera un dato se puede transformar en información:

- **contextualizando**: se sabe en qué contexto y para que propósito se generaron
- **categorizando**: se conocen las unidades de medida que ayudan a interpretarlos
- **calculando**: los datos fueron procesados matemática o estadísticamente
- **Corrigiendo**: eliminando errores/inconsistencias de los datos
- **condensando**: resumiendo los datos de forma más concisa (agregación de datos)

Conocimiento

Conocimiento: Fusión de **valores**, **información** y **experiencia**, 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

- Puedo aplicar "función inversa" a la información para volver a los datos que la originaron: datos ←→información
- Con el conocimiento, no puedo volver a la información que la originó, por eso es una fusión: información →conocimiento

Para convertir información en conocimiento:

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

Tecnologías OLAP (On-line Analytical Processing)

Tecnología basada en la utilización de tecnologías de bases de datos **multidimensionales**, que surge por la **acumulación masiva de datos** en los 90s, y que se diferencia de OLTP (on-line Transaction Processing), la cual se fundamenta en las DB relacionales

OLAP VS OLTP

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

- Su ejecución se basa en **transacciones** (las tablas transaccionales son las que se llenan por un proceso o relación entre otras tablas de la DB)
- Conforman el **99%** de los sistemas existentes
- Son sistemas **para la operación (operativos)**
- Procesan **datos**
- Los **datos** se almacenan **normalizados**
- Registran **datos** nivel de **detalle** en cada transacción
- Los datos son **volátiles** (en algún momento se pueden eliminar)

OLAP: "On-line analytical processing" también llamado **modelo relacional**, debido a que analiza y **relaciona** la información analizada

- Su ejecución se basa en el **análisis**
- Conforman el **1%** de los sistemas existentes
- Son sistemas para la **toma de decisiones/análisis**
- 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"**

OLTP:

- Muchos usuarios crean, actualizan o retienen registros individuales.

- Son DBs optimizadas para las actualizaciones de las transacciones (rápidamente)|

OLAP:

- las aplicaciones OLAP son usadas por analistas y gerentes que frecuentemente quieren vistas de alto nivel de los datos. Pocos usuarios a comparación de OLTP, pero estos pocos usuarios tienen un perfil alto dentro de la organización.
- Son actualizadas de a bloques, de múltiples fuentes, y proveen poderosas aplicaciones multiusuario de análisis
- Optimizadas para el análisis
- Facilidad y beneficio de utilizar DBs multidimensionales pero tienen un costo de mantenimiento muy alto

Estructura 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 DB OLAP de manera independiente

filtrado de datos: En la 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

Razones por las cuales los datos deben ser ajustados antes de realizar el análisis:

- 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. Existe diferencias en los modos de trabajar de las direcciones departamentales, en las estructuras operativas…
- Se pueden realizar análisis que no parten de datos operativos como pueden ser los que se obtienen de las características demográficas, publicidad televisiva…

Actualización y consistencias de datos:

Al usar datos procedentes de distintas fuentes, es probable que esas fuentes (DBs) se encuentren en diferentes estados de actualización. El análisis que realiza un OLAP depende de la consistencia de los datos y por lo tanto es necesaria una plataforma que garantice esa consistencia

Historial de datos:

La mayoría de aplicaciones OLAP incluyen el **tiempo** como una dimensión, esto permite obtener resultados provechosos en cuanto a análisis temporales cuando se dispone de datos de varios años atrás

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

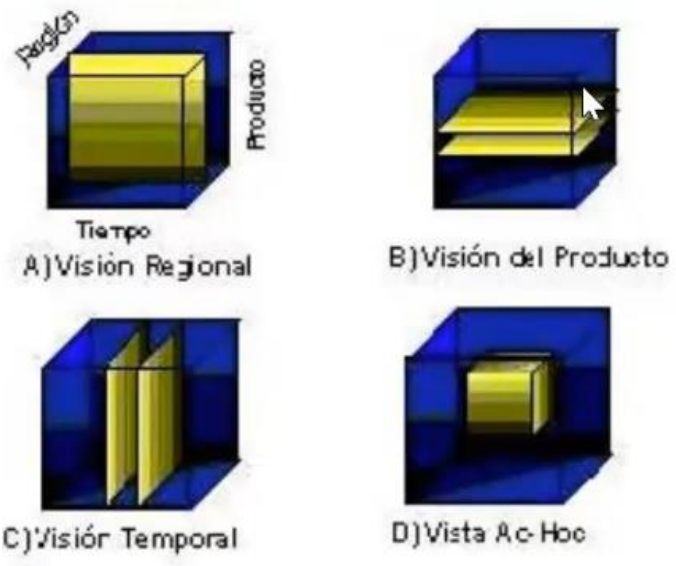
Actualización de datos: Si la DB dispone de varias entradas de datos, es obvio que es necesario separar la base de datos de OLAP para que no se sobreescriban los datos operacionales que se están usando en un determinado momento.

Diseño de las tablas en OLAP

Las dimensiones (**tiempo**, **región** y **producto**) determinan la estructura de la información almacenada y definen adicionalmente caminos de consolidación. Dimensión >= 3

De este modo, la información puede analizarse **dentro del cubo** formado por la **intersección** de las **dimensiones** de la variable particular

Ejemplos:



A) Para cada **región**, todos los **productos** vendidos a lo largo del **tiempo**
B) Para ropa deportiva (un **producto**), todo lo vendido a lo largo del **tiempo** en todas las **regiones geográficas** en las cuales hay sucursales
C) Todas las ventas de todos los **productos** de todas las **regiones** en un **año (2012)**
D) **Ac-hoc** : combinación de las tres anteriores cuanto calzado deportivo se vendió para el año 2012 en una región

Lo anterior es para una DB multidimensional de 3 dimensiones

Sparsely populated/dispersión de datos: A medida que se agregan dimensiones a una DB multidimensional, crece el número de puntos de datos (celdas). Ejemplo, considerando que no se venden todos los productos en todas las sucursales todos los días, si se considerara que las sucursales más pequeñas solo pueden manejar el 20% del total de productos, el 80% de las **celdas** estarán **vacías**, eso es la **dispersión de datos**

La dispersión de datos trae 2 problemas:

1. Cantidad de espacio innecesario ocupado
2. Problemas en la performance de algoritmos: los algoritmos de búsqueda o selección van a procesar igual todas las celdas en cero

Modos de dispersión de datos en multidimensional:

- **Hipercubo**: La información se guarda implícitamente en un gran y único cubo, donde todos los datos en la aplicación aparecen como una sencilla estructura multidimensional (como el cubo de arriba)

- **Multicubo**: La información se almacena dividiendo los datos en grupos (cubos) más pequeños (menos cantidad de datos) la base de datos multidimensional consiste en un numero de objetos separados normalmente con diferentes dimensiones

Clasificaciones:

Con ejemplo empresa viaje egresados

- **Sistemas transaccionales:** Más detallado: datos personales, fecha nacimiento, genero, domicilio, cobros cuotas, datos del colegio, etc…
- **Sistemas de logística:** micros, hoteles, habitaciones, transporte en Bariloche, etc…
- **Sistemas estratégicos:** rango de edad (17 a 20 años), localidad, etc… (datos que no se migran, datos que se migran sumarizados/por rangos, datos que se migran tal cual)

- **aplicativos in house**: Aplicativos donde tenes todo el código fuente (lo programaste, lo compraste, etc…)
- **Sistemas licenciados**: sistemas que se compra la licencia (SAP por ejemplo, en la nube)
- **Repositorio de datos:** lo que era visual basic por ejemplo

Proceso ETL (extract – transform – load)

Pasar de un archivo o conjunto de archivos (repositorio de datos) a un **stagin área/área intermedia** donde aplicaciones licenciadas que deben pagarse, **combinan**, **estandarizan** y **depuran**, antes de migrarlo al Data Warehouse (que es una base de datos)

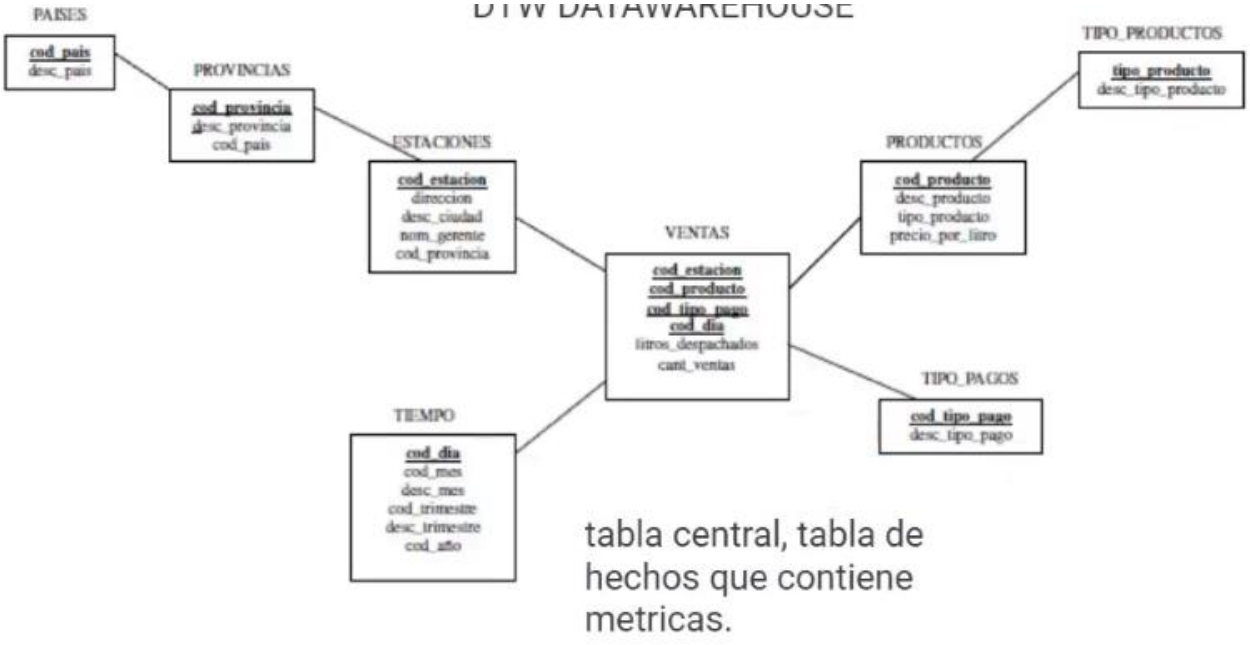
- **Combinación:** porque el cliente viene de varias fuentes, y tengo que combinar/integrar para pasarlo al DTW como un único cliente
- Estandarización: A causa de las múltiples fuentes, podemos tener datos que semánticamente significan lo mismo pero están escritos diferente (por ej genero M –F vs V-M)
- **Depuración**: Tenemos datos históricos, hay que manejar (decidir si migrar o no, los corrijo, los trato de completar, etc…) los datos incompletos, datos erróneos, datos que en su momento cumplían las reglas de negocio de la época pero ya no, etc…

DATA WAREHOUSE

Formado por una **tabla central**, llamada **tabla de hechos** cuyos atributos son las **métricas** (son todo lo que puedo medir)

Lo que importa es la **performance**, no la cantidad de info

Ejemplo de estación de servicio:



En este caso mide litros despachados y cantidad de ventas/clientes, no es un DATA WAREHOUSE, al solo medir ventas, se llama **DATA MART** (es solo una parte de un DTW)

Al unir todas las áreas y obtener la base de datos completa, se obtiene el **DATA WAREHOUSE**

En este caso hay una única tabla central, entonces estamos viendo un **hipercubo** →para que sea un multicubo debe haber mínimo 2

Nomenclatura:

DATA MART:

- Si todas las dimensiones del DATA MART son de 1 tabla, se llama **STAR schema** (modelo estrella)
- Si al menos una dimensión tiene más de una tabla, se llama **SNOW FLAKE (**copo de nieve)

DATAWAREHOUSE:

- Si todas las dimensiones de un DTW son de 1 tabla, se llama **constelación de estrellas**
- SI al menos una dimensión del DTW tiene más de una tabla se lama **snow flake**