**Fundamentos de Organización de Datos**

Introducción

Los datos no son persistentes al estar en la RAM. Es decir, si un dato se encuentra almacenado en la memoria principal, al por ejemplo desenchufar la computadora, estos datos se pierden. Un **archivo** es una estructura de datos donde los datos almacenados en ella persisten en **memoria secundaria**(disco duro,usb,etc).

CLASE 1

¿Base de datos?

Es una **colección de archivos** diseñados para servir a múltiples aplicaciones. Es una colección coherente de datos con significados inherentes. Un conjunto aleatorio de datos no puede considerarse una BD, debe tener cierta lógica.Una BD se diseña, construye y completa de datos para un propósito específico.

ARCHIVOS

Un **archivo** es una estructura de datos donde los datos almacenados en ella persisten en **memoria secundaria**(disco duro,usb,etc).

Organización

* Secuencia de Bytes: No se puede determinar fácilmente comienzo y fin del archivo. (Archivos de texto)
* **Campos** y **registros**(archivos estructurados): El campo es la unidad más pequeña de un archivo. El registro es un conjunto de campos. El acceso es secuencial. (file)

Tipos según su Acceso

* Secuencial **Físico**(archivos serie): acceso a los registros uno tras otro y en el orden físico en el que están guardados. Búsqueda secuencial y binaria.
* Secuencial **indizado o lógico**: acceso a los registros de acuerdo al orden establecido por otra estructura. Uso de índices, árboles.
* **Directo**: acceso a un registro determinado sin necesidad de haber accedido a los predecesores. Hashing.

Tipos según # de Cambios

* **Estáticos**: Pocos cambios; puede actualizarse en procesamientos por lotes. No necesita estructuras adicionales para agilizar cambios.
* **Volátiles**: Sometidos a operaciones frecuentes: agregar, borrar, actualizar. Necesita estructuras adicionales para mejorar los tiempos de acceso.

¿Qué es un Buffer?

Memoria intermedia(ubicada en RAM) entre un archivo y un programa,donde los datos residen provisoriamente hasta ser almacenados definitivamente en memoria secundaria o donde los datos residen una vez recuperados de dicha memoria secundaria.

Operaciones básicas

2 niveles:

* Físico(almacenamiento secundario).
* Lógico(dentro del programa):
  + Operaciones:
    - **Crear**
    - **Abrir**
    - **Read/Write**
    - **Eof(End of file)**
    - **Seek(localización)**

- Archivo **físico**: Existe en almacenamiento secundario. Es el archivo tal como lo conoce el SO y que aparece en su directorio de archivos.

- Archivo **lógico**: Es el archivo visto por el programa. Permite a un programa describir las operaciones a efectuarse en un archivo.

**Assign**

Sirve para ponerle nombre físico al archivo (como se verá cuando se abra el archivo). Ejemplo:

Assign (arch.nombre, “nombre.dat”(ruta)); Primero el nombre lógico luego el físico.

— — — — — — — —

**rewrite**(nombre lógico); crea un archivo, solo escritura.

**reset**(nombre lógico); abre un archivo, lectura-escritura.

**close**(nombre lógico); transfiere la información del buffer al disco. Pone el eof.

**eof**(nombre lógico); Es una función.

— — — — — — — —

**read**(nombre lógico, var\_dato); El puntero del archivo avanza automáticamente al siguiente.

**write**(nombre lógico, var\_dato); El puntero del archivo avanza automáticamente al siguiente.

LEEN Y/O ESCRIBEN sobre los BUFFERS.

El tipo de dato de la variable es igual al tipo de dato de los elementos del archivo.

— — — — — — — —

Funciones:

**fileSize**(nombre lógico); devuelve el tamaño del archivo

**filePos**(nombre lógico); Devuelve la pos actual del puntero en el archivo.

Procedimiento:

**seek**(nombre lógico, pos); establece la pos del puntero en el archivo.

CLASE 2

Algoritmia:

* **Básica**: Agregar nuevos elementos.
* **Clásica**: Actualización, Merge, Corte de Control. EN **FOD PRÁCTICA**

Proceso **Agregar** dato a Archivo:

Procedure agregar (var emp: empleados); → empleados = file of registro;

var e: registro;

begin

reset(emp);

seek(emp, filesize(emp));

leer (e);

while (e.nombre <> ‘ ‘) do begin

write (emp, e);

leer(e);

end;

close (emp);

end;

— —— —— —— —— —— —— —— —— —— —— —— —— —— —— —— —

Archivos:

* **Maestro**: Es el archivo que resume información sobre un dominio de problema específico.
* **Detalle**: Es el archivo que contiene movimientos realizados sobre la información almacenada.

CLASE 3

La RAM si bien es rápida(trabaja en nanosegundos) y de simple acceso, tiene sus desventajas frente al almacenamiento secundario(milisegundos):

* Capacidad limitada. Es más chica que un disco.
* Mayor costo. Es mucho más cara que un disco.
* **Volátil.** Frente a un fallo como un corte de luz la memoria RAM pierde su información.

ALMACENAMIENTO SECUNDARIO

El almacenamiento secundario necesita más tiempo para acceder a los datos. Su acceso es tan lento que es necesario enviar y recuperar datos con INTELIGENCIA. Al buscar un dato se espera encontrarlo en el primer intento(o en pocos). Si se buscan varios datos, se espera obtenerlos todos de una vez.

La información en el almacenamiento está organizada en archivos → Colección de **bytes** que representa la información.

Viaje de un Byte

Desde que se hace un write (archivo,variable) hasta que el byte llega al disco, se cumplen unos **ciclos** necesarios para escribir. Se involucran:

* **Administrador** de **archivos**: Es un conjunto de programas del SO (capas de procedimiento) que se encargan del procesamiento de la E/S de la información:
  + - Capas superiores: aspectos lógicos de datos
    - Capas inferiores: aspectos físicos de datos.

-**Buffers** de E/S: Agilizan la E/S de datos. Implica trabajar con grandes grupos de datos en RAM para reducir el acceso a almacenamiento secundario.

-**Procesador** de **E/S**: Dispositivo utilizado para la transmisión desde o hacia el almacenamiento externo. Independiente de la CPU.

-**Controlador** de **disco**: Encargado de controlar la operación de disco. Se refiere a la parte mecánica física del disco(magnético) que se encarga de transferir los datos al disco, para eso el brazo mecánico se coloca en la pista y luego en un sector.

En resumen,se encarga el SO de transferir la información al disco.

ARCHIVOS ESTRUCTURADOS

Campos

* Longitud **predecible**(fija): Desperdicio de espacio. Por ejemplo definimos un string[20], si usamos solo 15 de esos 20 bytes estamos desperdiciando espacio.
* Longitud **variable**: 2 formas:

- Indicador de longitud al principio de cada campo

- Delimitador al final de cada campo con un carácter especial.

Hoy en día las bases de datos trabajan con una longitud predecible. Es preferible eficiencia en recuperación de la información respecto al desperdicio de espacio.

Registros

* Longitud **predecible**: campos fijos o variables. El registro limita a los campos a un determinado número de bytes.
* Longitud **variable**: Indicador de longitud, delimitador con carácter, o segundo archivo con información de la dirección del byte de inicio de cada registro.

**Claves**

Permiten la identificación del registro. Deben permitir generar orden en el archivo por ese criterio.

* **Primaria/Unívoca**: Identifica un elemento particular dentro de un archivo.
* **Secundaria**: Reconoce un conjunto de elementos con igual valor.

**Forma canónica**: Es una representación única para la clave, ajustada a una regla. Ejemplo: clave solo en letras mayúsculas y sin espacios finales.

Estudio de la **performance** de un **ALGORITMO**

La performance de un algoritmo se puede medir en base al **N° de comparaciones** (cuantos ciclos de máquina ocupa el algoritmo para resolverse) y **midiendo cuántos accesos a discos hace**. Es menos eficiente hacer muchos accesos al disco qué hacer muchas comparaciones. Veremos el mejor caso, el peor caso y el caso promedio para cada algoritmo (mejor caso + peor caso / 2).

Performance Archivos Serie

* Mejor caso: leer 1 registro.
* Peor caso: leer n registros.
* Caso promedio: n/2 comparaciones.

Es de O(n). Depende de la cantidad de registros.

Recordemos que cuando se leen n registros no hacemos n accesos a discos gracias al buffer, que trae del disco bloques de registros. Ejemplo: si un archivo tiene 1.000.000 de registros y en cada buffer caben 1.000; en el peor caso se hacen 1.000 accesos a disco para recuperar la información.

Performance Archivos Directos

* Mejor, peor y caso promedio: 1 acceso y 1 lectura. O(1).
* Debo necesariamente conocer el lugar donde comienza el registro requerido.

Saber dónde comienza el registro es conocer el NRR.

**NRR**

Número relativo de registro: Indica la posición de un registro respecto al principio del archivo. Solo aplicable con registros de longitud fija. Ejemplo: NRR 10 y longitud de cada registro 8: 10\*8= 80 bytes de distancia

**Eliminación**

* Baja lógica: el registro queda marcado como borrado pero no se elimina realmente.
* Baja física: el registro se elimina realmente y queda un espacio en el archivo.

¿Qué hacer con la información desperdiciada en la Baja Lógica?

* **Recuperación de espacio**: periódicamente utilizar el proceso de baja física(FOD PRÁCTICA) para realizar un proceso de compactación del archivo.
* **Reasignación de espacio**: Recuperar el espacio utilizando los lugares indicados como borrados para el ingreso (altas) de nuevos elementos al archivo. Dependendiendo si se usan registros con longitud fija o variable se trabaja de formas distintas:
* **Fija**: Si hay una marca de borrado puedo colocar el registro ahí. Para mejorar la eficiencia tendríamos que conocer de antemano dónde están esas marcas de borrado para no tener que buscarlas secuencialmente. Para esto uso una Lista o pila:

En el encabezado estará NRR -4, el archivo tendrá→

alfa beta delta -6 gamma -1 epsilon

Si se borrara por ejemplo beta, el encabezado tendría el -2 y beta se reemplaza por -4.

* **Variable**: Si hay una marca de borrado puedo colocar el registro ahí siempre y cuando el registro nuevo **quepa**. Al igual que los tipos de registros de longitud fija se debe utilizar una lista(No una pila ya que se necesita buscar el registro borrado más adecuado) para conocer dónde están las marcas de borrado. Sin embargo, no se utilizan los NRR como enlace. Cada registro indica en su inicio la dirección del primer byte.

Fragmentación

Es hacer que un archivo ocupe más lugar del que necesita. 2 tipos:

* **Interna**: Se desperdicia espacio en un registro, se le asigna lugar pero no lo ocupa totalmente. Ocurre con registros de longitud fija.
* **Externa**: Espacios entre registros libres. Soluciones:
  + - Unir espacios libres pequeños adyacentes.
    - Minimizar la fragmentación eligiendo el espacio **más adecuado.**

Estrategias para elegir el espacio más adecuado (tienen sentido con regs de long variable):

* **Primer** ajuste: Se selecciona la primera entrada de la lista de disponibles que pueda almacenar al registro(se le asigna completa puede generar fragmentación interna). Minimiza la búsqueda, no se preocupa por la exactitud del ajuste.
* **Mejor** ajuste: Elige la entrada que más se aproxime al tamaño del registro y se le **asigna completa**. Se hace una búsqueda hasta terminar la lista. Puede generar fragmentación interna.
* **Peor** ajuste: Elige la entrada más grande para el registro, se le asigna sólo el espacio necesario. Se hace una búsqueda hasta terminar la lista. Genera fragmentación externa.

CLASE 4

Para mejorar la performance de la búsqueda de secuencial de archivos series, podemos realizar una **búsqueda binaria** siempre y cuando el archivo serie esté *ordenado por clave* y tenga *registros de longitud fija*(No se puede calcular la mitad en archivos de longitud variable). Ahora la eficiencia es de **Log2 n**.

Si bien se mejora la eficiencia, el costo por ello es mantener ordenado el archivo como ya dije, entonces:

¿Cómo ordenar un archivo?

* **En RAM**: Levanto todo el archivo a memoria RAM guardándolo en un vector, es decir si tiene N elementos hago N lecturas(recordar el buffer, no son n accesos). Ordeno el vector (despreciable la cantidad de comparaciones) y escribo el vector ordenado como mi nuevo archivo(N escrituras). En total **2N** operaciones, en cambio si el ordenamiento fuese directamente en el archivo serían N! operaciones (Recorro el archivo, busco el más chico y lo pongo en el primer lugar, repito y en el segundo y así).

Ahora, qué pasa si el archivo NO cabe en memoria RAM? →

* **Claves en RAM**: En vez de levantar el registro entero del archivo levanto una parte más chica: la clave, que me da el criterio de orden, y el NRR. Ahora el total de operaciones es de **3N**, ya que se leen las N claves, se ordenan, se leen N registros pero ahora en orden de las claves(se accede directamente por el NRR), y finalmente se escriben los N registros en un nuevo archivo.

¿Qué pasa si las claves TAMPOCO caben en RAM? →

* Ordenamiento para **Archivos muy grandes**:
  + Partir el archivo.
  + Ordenar cada parte.
  + Mergear las partes.

Es decir, levanto lo que quepa en RAM, lo ordeno y guardo en un nuevo archivo, tomo la siguiente parte del archivo original y hago lo mismo, así cuantas veces sea necesaria para procesar todo el archivo. Luego, utilizando el proceso de **Merge** junto todos los archivos ordenados para formar el archivo final.

El proceso de Merge es el que agrega una complejidad y un nivel de ineficiencia que hace que sea peor que los anteriores métodos. Aun así, sigue siendo mejor que ordenar el archivo directamente en disco.

Clase 5

Hasta ahora, sabemos que para mejorar la performance de la búsqueda de información tenemos que minimizar el número de accesos.

* Búsqueda secuencial es poco eficiente
* La binaria es más eficiente pero muy costosa, hay que mantener los archivos ordenados, además solo sirve para registros de longitud fija.
* Estructura auxiliar:

**Índices**

Un índice es una estructura de datos adicional que permite agilizar el acceso a la información almacenada en un archivo. En dicha estructura se almacenan las *claves* de los registros del archivo, junto a la referencia de acceso a cada registro asociado a la clave, por lo que es mucho más pequeño que el archivo de datos. Es necesario que las claves permanezcan ordenadas.

Esta estructura de datos es otro archivo con registros de longitud fija(puede hacerse búsqueda binaria), independientemente de la estructura del archivo original(el archivo original puede ser de longitud variable). La característica fundamental de un índice es que posibilita imponer **orden** en un archivo **sin que realmente este se reacomode.**

Generación y mantenimiento de índices **primarios**

* Creación: Cuando creo el archivo de datos también creo el archivo índice o tantos archivos índices como sean necesarios, ambos vacíos, solo con el registro cabecera.
* Altas: Se agrega un registro al final del archivo de datos. Se genera un nuevo registro a insertar de forma ordenada en el índice que contiene el NRR (long. fija) o la dirección del primer byte (long. variable) más la clave primaria.
* Modificaciones:
  + Sin modificar la clave:
    - Registros de long. fija: No pasa nada.
    - Registros de long. variable: Si el registro no se agranda, se almacena en la misma posición física y el índice no se ve afectado. Si el registro se agranda, se reubica en el archivo de datos y se debe guardar la nueva posición inicial en el índice.
  + Modificando la clave:

Se debe actualizar y reordenar el archivo de índices.

* Bajas: Eliminar un registro del archivo de datos implica borrar la información asociada en el índice primario. Así, se debe borrar física o lógicamente el registro correspondiente en el índice. Se debe notar que no tiene sentido recuperar el espacio físico en el índice con otro registro que se inserte, debido a que este archivo índice está ordenado y el elemento a insertar no debe alterar este orden.

Ventajas índice primario

La performance de búsqueda se mejora al realizarla sobre un archivo ordenado y de longitud fija. Si el archivo se levanta a memoria RAM es mucho más eficiente para mantenerlo ordenado. Sin embargo, en cuanto a **persistencia de datos** no conviene tenerlo en memoria RAM, ante un problema como por ejemplo un corte de luz podemos perder información modificada en el índice.

Para que el índice siempre esté en el disco: uso de ÁRBOLES. Estos trabajan directamente sobre los archivos en disco, perdemos EFICIENCIA pero aseguramos la PERSISTENCIA DE DATOS y además nos despreocupamos del tamaño del índice.

Generación y mantenimiento de índices **secundarios**

Las claves secundarias aceptan valores repetidos, en cambio las primarias no. Existen para ser recordadas fácilmente por el usuario para poder acceder a la información mediante ellas.

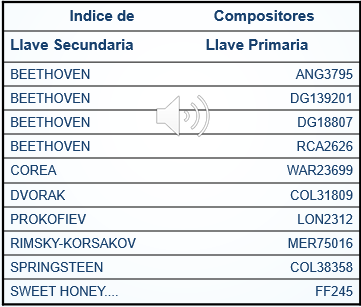
Un índice secundario es un archivo con registros de longitud fija que permite relacionar una clave secundaria con una o más claves primarias.

Entonces, para acceder a un dato deseado, primero se accede al índice secundario por clave secundaria; allí se obtiene la clave primaria, y se accede con dicha clave al índice primario para obtener finalmente la dirección efectiva del registro que contiene la información buscada.

* Creación: Al implantarse el archivo de datos, se deben crear todos los índices secundarios asociados, naturalmente vacíos, solo con el registro cabecera.
* Altas: Cualquier alta en el archivo de datos genera una inserción en el índice secundario, que implica reacomodar el archivo en el cual se almacena.
* Modificaciones: Cuando se produce un cambio en la clave secundaria se debe reacomodar el índice secundario. Cuando se produce un cambio en la clave primaria se debe modificar esta en el índice secundario. Si se cambia el resto del registro en el archivo de datos no pasa nada.
* Bajas: Cuando se elimina un registro del archivo de datos, esta operación implica eliminar la referencia a ese registro del índice primario más todas las referencias en índices secundarios. (Luego solo en archivo de la lista invertida)-

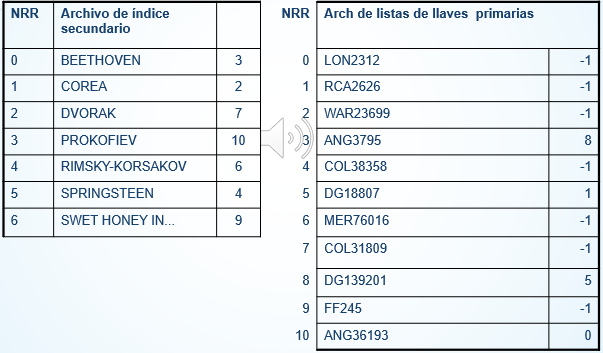
Organización de índices secundarios

Cuando se da de alta un registro al archivo de datos, si la clave secundaria es una que ya existía, en el índice secundario al momento de agregar esta clave, se debe reacomodar todo el archivo de índice secundario:



Supongamos que existían 3 Beethoven, al momento de agregar el 4to se tuvo que reacomodar el archivo para que quede ordenado por clave primaria.

Para evitar esto, utilizo un archivo auxiliar para los índices secundarios, es un archivo de listas de llaves primarias:



De esta forma se manejan las claves secundarias repetidas.

Ventajas índice secundario

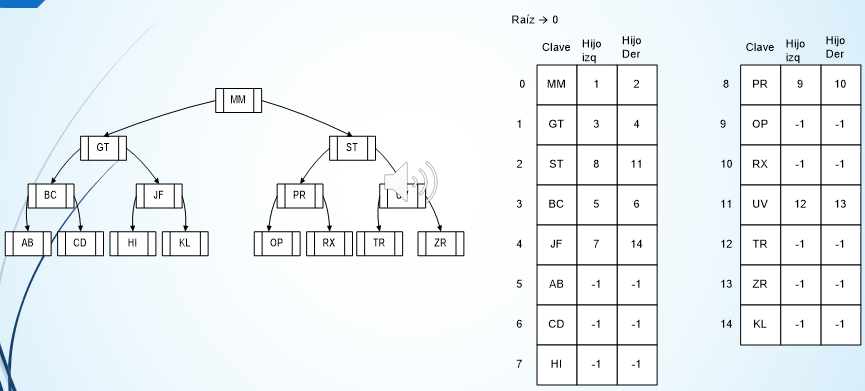
Si se agregan o borran datos de una clave secundaria ya existente, solo se debe modificar el archivo que contiene la lista. El único reacomodamiento en el índice se produce cuando se agrega o cambia una clave primaria. Igualmente el índice es más pequeño, por lo que dicho reacomodamiento resulta menos costoso.

Clase 6

Vamos a utilizar la estructura de datos de Árbol binario como soporte de los índices, para esto es necesario **implantar** el árbolsobre disco, o sea sobre un archivo, esta estructura es más eficiente para la búsqueda de información que la búsqueda binaria.

Construcción de archivo de árbol binario

* Hago un write del nuevo nodo al final del archivo.
* Comparo <> con la raíz, cuando encuentre -1 lo pongo ahi.



Los árboles binarios de búsqueda tienen eficiencia **logarítmica** solamente cuando están balanceados. Un árbol está **balanceado** cuando la altura de la trayectoria más corta hacia una hoja no difiere de la altura de la trayectoria más grande. El mayor inconveniente de los árboles binarios es que se **desbalancean** fácilmente

Árbol AVL

Los árboles avl son árboles binarios balanceados en altura. Esto quiere decir que para cada nodo → La diferencia entre la altura de su camino izquierdo y su camino derecho no puede ser mayor que determinado límite. Por ejemplo BA(1) quiere decir que el límite de la diferencia de altura es 1:

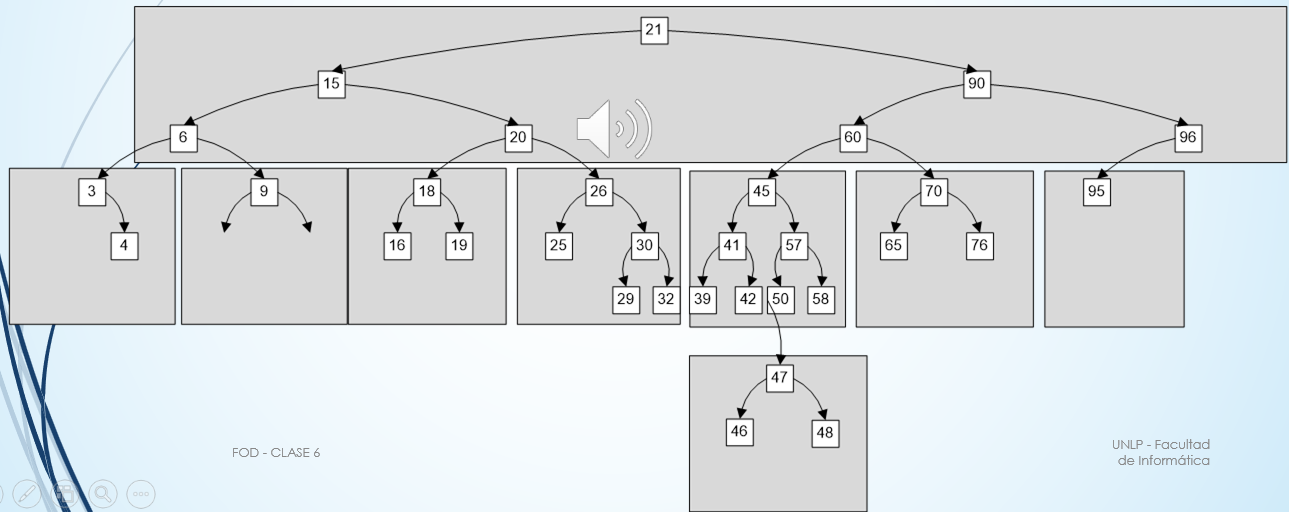


La diferencia de altura para el nodo raíz entre su camino izquierdo y su camino derecho es 1. En cuanto al nodo hijo derecho de raíz, la diferencia de altura es 0, por debajo del límite. Por lo que este árbol es balanceado en altura.

**Eficiencia**: parecido a log2 n (1.44 log2 n), el árbol no se desbalancea mucho. Aunque no nos sirven tanto estos árboles porque para mantenerlos balanceados en altura requieren algoritmos COMPLEJOS que hacen muchos accesos a disco.

Árbol binarios Paginados

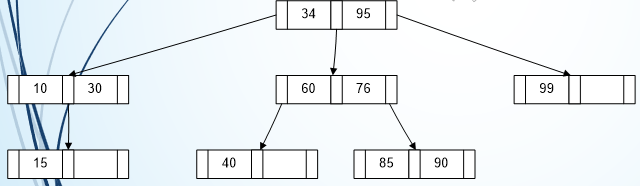
Los árboles binarios en memoria secundaria están separados en **páginas**, una página de un árbol tiene tantos nodos como cantidad de registros que entren en un buffer, es decir, cuando se va a buscar al disco un nodo, se trae la página entera. Es por esto que los nodos de una misma página tienen que estar ubicados *físicamente cerca* (no es fácil de lograr).



La cuestión a analizar ahora es cómo generar un árbol binario paginado. Para que el uso de las páginas tenga sentido, los elementos correspondientes a la misma deberían llegar consecutivos para almacenarse en el mismo sector del disco.

Árboles multicamino

Generalización de árboles binarios, cada nodo tiene **k punteros** y **k-1 claves**, disminuye la profundidad del árbol.



Árboles **balanceados B**

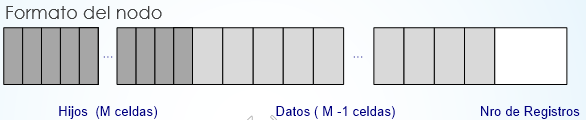
Son árboles multicamino con una construcción especial en forma ascendente que permite mantenerlo balanceado a bajo costo.

**PROPIEDADES** de un árbol B de orden M(máx cantidad de hijos, punteros):

* Cada nodo que no sea ni raíz ni terminal tiene como mínimo M/2(integer) hijos.
* La raíz tiene como mínimo 2 hijos (o sino ninguno).
* Todos los nodos terminales al mismo nivel.
* Los nodos terminales tienen como mínimo M/2 -1 registros.

La idea es que cada **nodo** reemplace a una **página** de nodos.

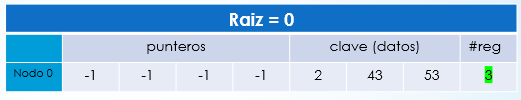
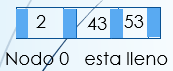
Formato en Archivo



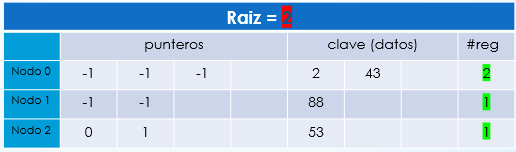
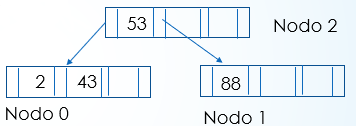
**Creación**

Los elementos siempre se insertan en nodos terminales.

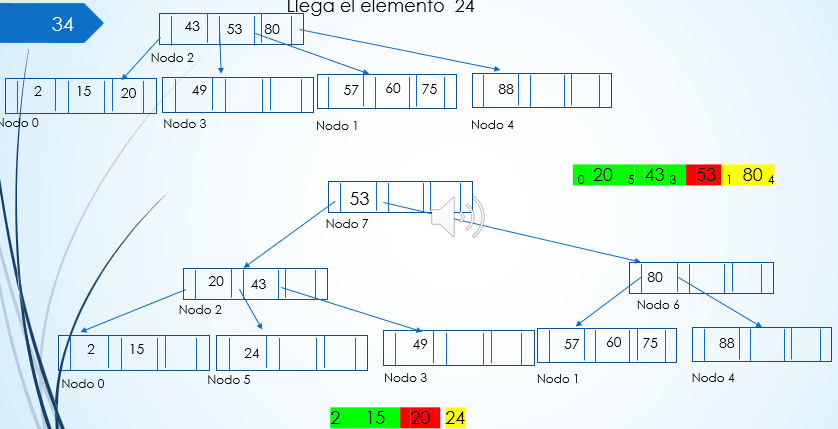
Árbol: Archivo:

En un nodo la inserción de claves se hace de forma ordenada

Supongamos que llega el 88. Como el nodo está lleno, se produce **overflow**. Esta clave se posiciona imaginariamente al lado del 53, a su derecha por ser más grande, se divide la cantidad de elementos del nodo más este nuevo por 2, es decir 4/2, la mitad sube a ser raíz, el lado derecho pasa a ser hijo derecho de la raíz y el lado izquierdo hijo izquierdo.



Pudo haber subido el 43 y el 53 haber quedado a la derecha



Performance de búsqueda

* Mejor caso:1 lectura.
* Peor caso: h lecturas (siendo h altura). El tema es saber el valor de h.

**AXIOMA**: Árbol balanceado de orden M, si el número de elementos (claves) es N → Hay N+1 punteros nulos en nodos terminales (-1).

¿Cómo calcular h?

Para calcular h tendríamos que analizar el peor caso, en un árbol balanceado el peor caso sería que cada nodo tenga la cantidad mínima de hijos, de esta forma el árbol crece en altura(al tener más hijos, menor altura). Entonces:

Nivel Núm mín de hijos

1 2

2                  2\*(M/2)

3                      2\*(M/2)\*(M/2)

… …

h 2\*(M/2)h-1

Esto quiere decir que el nivel h (nivel de las hojas) tendría como mínimo 2\*(M/2)h-1 hijos, pero como son hojas, estos son los punteros nulos.

Relacionando el **axioma** y el número de punteros nulos de los nodos terminales:

N+1>= 2\*(M/2)h-1

Despejando:

h<= 1+ log base m/2 ((N+1)/2).

Conclusión: conociendo M y N (orden y cantidad de claves) podemos obtener la altura del árbol en el peor caso. Despejando h obtenemos:

**h <= 1+ logM/2 ((N+1)/2)**

Performance de inserción

Mejor caso (sin overflow):

* H lecturas
* 1 escritura

Peor caso (overflow hasta la raíz, aumenta en uno el nivel del árbol)

* H lecturas
* 2h +1 escrituras (2 por nivel + la raíz)

**Eliminación**

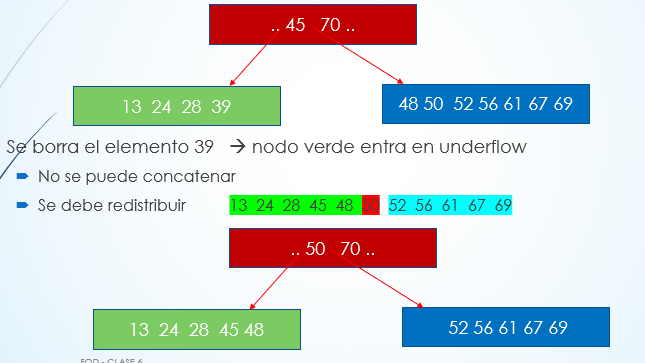
* Siempre eliminar de nodos terminales.
* Si se requiere eliminar un elemento que no está en un nodo terminal, llevarlo allí primero.

Posibilidades ante eliminación:

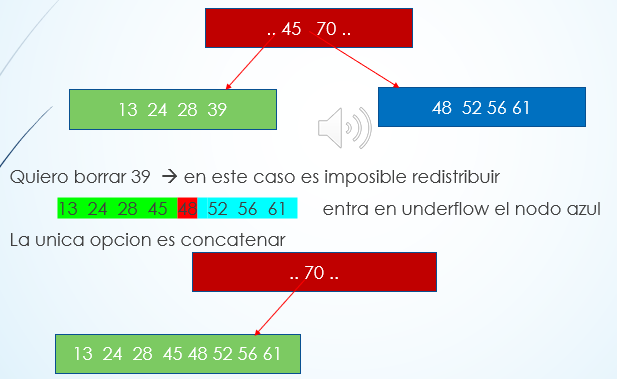
* Mejor caso: borrar un elemento del nodo y que no produzca **underflow**, solo reacomodos en el nodo. (El underflow rompe la propiedad que dice: cantidad de registros en un nodo terminal M/2 -1).
* Peor caso: Se produce underflow.

Soluciones para **underflow**:

* **Redistribuir**. Ejemplo con orden del árbol 10.



* **Concatenar**. Orden 10:



Performance de eliminación

* Mejor caso(borra un nodo terminal):
  + H lecturas
  + 1 escritura
* Peor caso(concatenar lleva a decrementar el nivel del árbol en 1)
  + 2h -1 lecturas
  + H +1 escrituras

**ÁRBOL B\***

Se crean árboles balanceados más anchos. Es decir, tiene menos altura, al tener menos altura la búsqueda de información se hace en menos accesos. Esto es gracias a que en la inserción puedo redistribuir, y al hacerlo pospongo la creación (división) de nuevos nodos, que esto me genera el aumento de altura del árbol (vas tirando el overflow para arriba).

Un B\* es un árbol B en el cual cada nodo está lleno por lo menos ⅔ partes.

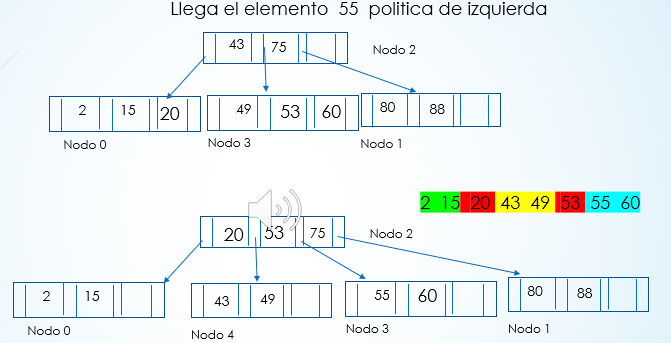
**Propiedades** (orden M)

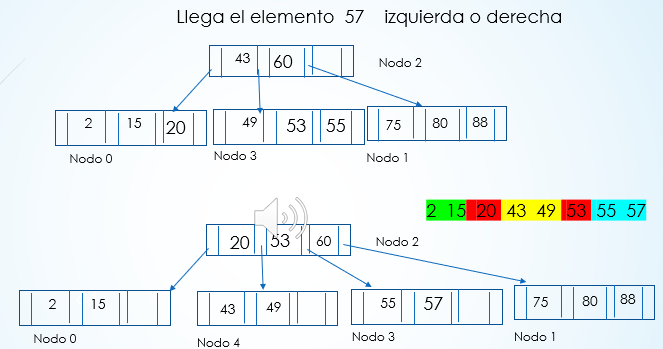
* Cada página (nodo) tiene máximo M descendientes.
* La raíz tiene al menos 2 descendientes o ninguno.
* Todas las hojas aparecen al mismo nivel.
* Una página que no sea hoja si tiene K descendientes contiene K-1 llaves.
* Cada página, menos la raíz y las hojas, tienen al menos ( (2M - 1) / 3 ) descendientes.
* Una página hoja tiene por lo menos ( (2M - 1) / 3 )-1 claves.

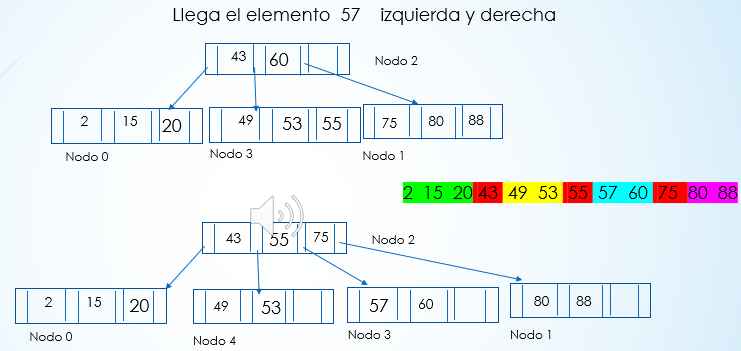
Operaciones

* Búsqueda: igual que B.
* Inserción: división (intento después de intentar redistribuir) o 3 casos posibles para redistribución:
  + **Derecha**: redistribuir con nodo adyacente hermano de la derecha (o izquierda solo si es el último)
  + **Izquierda o derecha**: si el nodo de la izquierda está lleno y no se puede redistribuir, se busca el de la derecha. Si los 2 están llenos entonces se divide con el nodo de la izquierda.
  + **Izquierda y derecha**: si el nodo de la izquierda está lleno y no se puede redistribuir, se busca el de la derecha. Si los 2 están llenos se divide entre los 3 nodos, estos **tendrán un ¾ parte llena.**

Ejemplo:







Recordar izq y der. dividir en ¾.

**ÁRBOL B+**

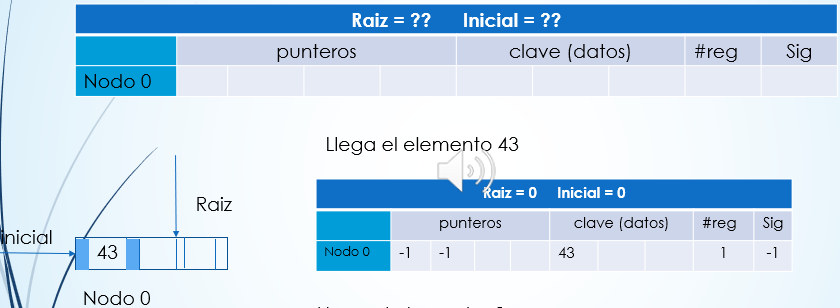
Está formado por un conjunto de grupos de registros ordenados por clave en **forma secuencial** (páginas terminales) y con un conjunto de índices que proporciona **rápido acceso** a los registros (nodos no terminales).

**Propiedades**

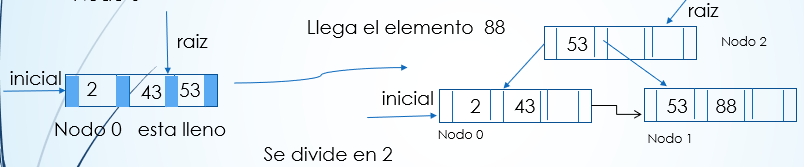
* Cada página tiene máximo M descendientes.
* Cada página, menos la raíz y las hojas, tienen entre M/2 y M hijos.
* La raíz tiene al menos 2 descendientes (o ninguno).
* Todas las hojas aparecen en el mismo nivel.
* Una página que no sea hoja si tiene K descendientes contiene K-1 llaves.
* Los nodos terminales representan un conjunto de datos y son linkeados juntos.

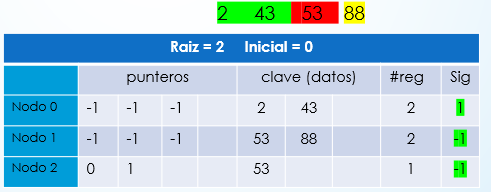
Operaciones

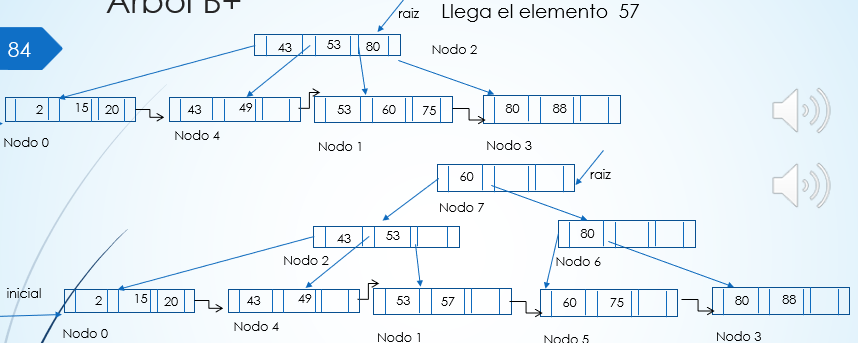
* Creación/Inserción



Se agrega en el registro del archivo el campo Sig.

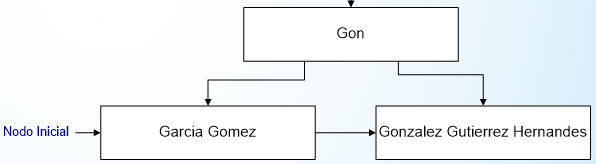




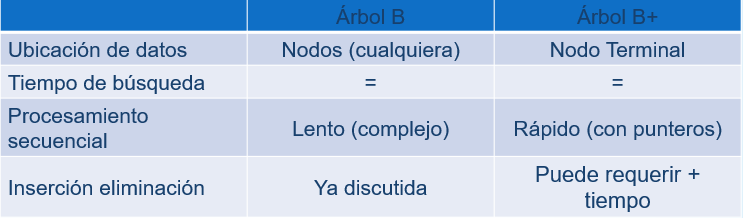


Árboles B+ de prefijos simples

El conjunto índice está formado por separadores más cortos.



Conclusiones

****

**Hashing**

Número de accesos según la forma en que se encuentre el archivo hasta ahora:

* Secuencia: N/2 promedio.
* Ordenado: Log2 N.
* Árboles: 3 o 4.

Necesitamos un mecanismo de acceso a registros con una única lectura → Hashing

El **hashing** o **dispersión** es una técnica que convierte la llave del registro en un número aleatorio, el que sirve después para determinar dónde se almacena el registro.

Diferencias entre:

Archivos secuenciales indizados → Mejor caso 2 accesos

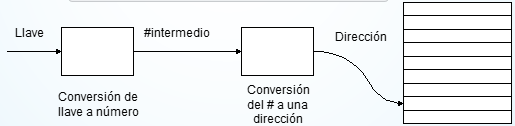
* Un archivo de datos.
* Un archivo de índice primario.
* Un archivo de índices secundarios.

Archivos directos → Un acceso a disco

* No hay estructuras adicionales.
* El archivo de datos es el único que se organiza y por un único criterio (clave primaria).

Atributos del hash

* No requiere almacenamiento adicional para el índice.
* Facilita la inserción y eliminación de los registros.
* Encuentra registros con MUY POCOS accesos a disco en promedio.



Si la dirección está ocupada se produce **colisión** (pepe sinónimo de epep) u **overflow**(no entran más registros en la dirección).

Costo del hashing

* No podemos usar registros de longitud variable.
* No puede haber orden físico de datos.
* No hay llaves duplicadas.

Parámetros que definen la eficiencia

1. **Función de hash** → Es una función matemática, tiene que ser una función que devuelva un número único para la clave pasada. Entrada: llave. Salida: dirección de memoria. No hay relación aparente entre llave y dirección.

* **Colisión**: Un registro es asignado a una dirección que está utilizada por otro registro.
* **Overflow**: Un registro es asignado a una dirección que está utilizada por otro registro y no queda espacio para este nuevo.

Pasos

* Representar la llave en forma numérica en caso que no lo sea.
* Aplicar la función.
* Relacionar el número resultante con el espacio disponible(ejemplo, espacio disponible 256 bytes, y la función me da 257, hay que ver cómo mapearlo).

1. **Tamaño de los nodos** (o **cubeta**, es la dirección de memoria) → Pueden tener más de un registro.

A mayor tamaño:

* Menor overflow.
* Mayor fragmentación.
* Búsqueda más lenta dentro del nodo. Aunque esto no afecta a la eficiencia ya que se hace en RAM.

1. **Densidad de empaquetamiento** → Es el cociente entre : N° de registros del archivo / capacidad total del archivo(direcciones de memoria \* # de registros que entran en cada dirección). DE menor: menor overflow y más desperdicio de espacio.

Es necesario analizar el comportamiento de un archivo directo : Cuando un registro requiere un solo acceso y cuando requiere más accesos → **Estimar el overflow**: analizar probabilísticamente si la inserción de un registro genera colisión o no y si esa colisión genera overflow. Entonces sabiendo que:

* N: # de nodos.
* C: capacidad de 1 nodo.
* R: # de registros del archivo.
* DE: R / C x N.

Probabilidad de que un nodo reciba i registros (**distribución de Poisson**):

P(i) = (R! /(i! (R - i)) ) \* (1/N)i \* (1- 1/N)R-i. Forma **simplificada:**

P(i)= ((R/N)i \* e-(R/N)) / i!

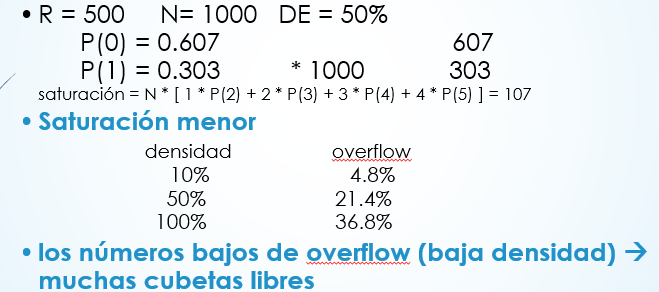
Entonces si reemplazamos i por C+1 estaremos obteniendo la probabilidad

de que un nodo esté en overflow.

Una vez que obtenemos **P(i) si lo multiplicamos por N**, siendo N el número

de direcciones, estaremos obteniendo el # esperado de direcciones con i

registros asignados.



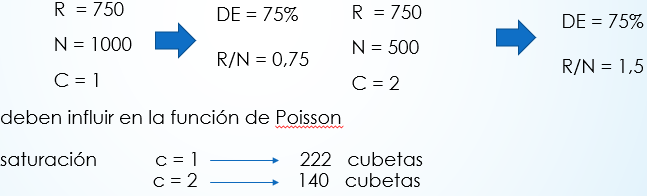
Saturación es lo mismo que overflow. La cuenta de saturación da como

resultado la cantidad de registros en overflow. A mayor densidad, mayor

porcentaje de overflow. Es decir que conviene disminuir la densidad de

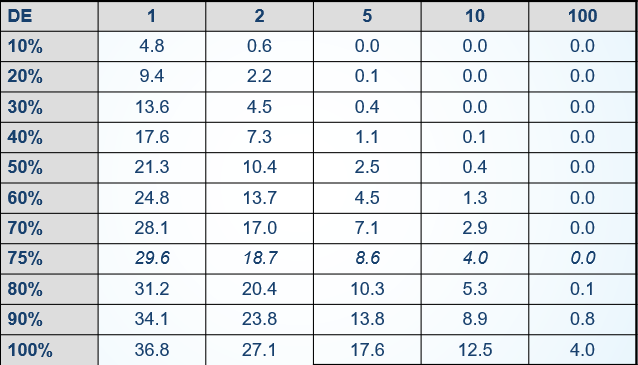
empaquetamiento aunque esto deja muchas cubetas libres.

Cuando tenemos la mitad de direcciones y la capacidad de registros por cubeta aumenta, vemos que el overflow disminuye.



**Conclusión**:

A mayor DE y mayor capacidad de registros por nodo, el overflow va a ser menor



1. **Método de tratamiento de overflow**:

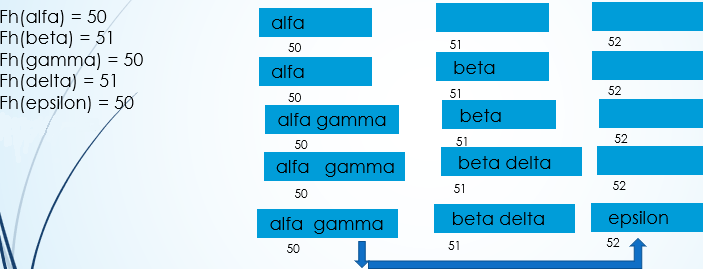
Si bien el % de overflow se reduce, no llega a 0%, por lo cual, existen métodos para el tratamiento de overflow. 4 de ellos son:

* **Saturación progresiva**: Se busca el próximo nodo que esté libre.

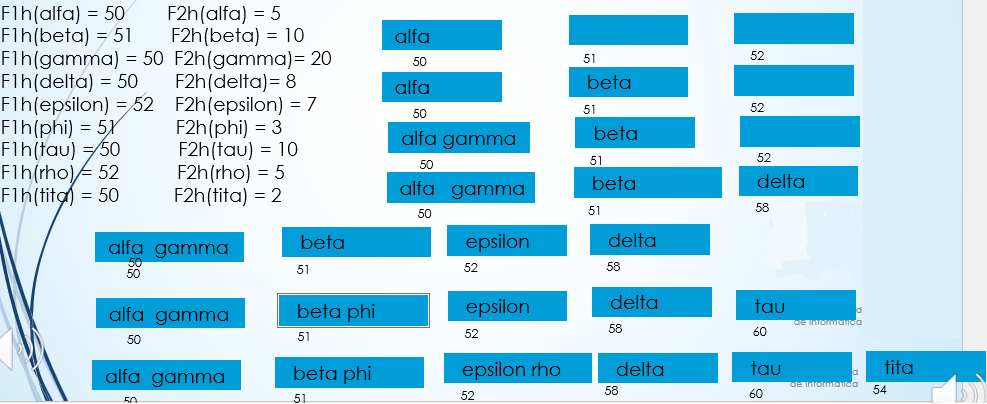


1. Búsqueda: Si el nodo se llenó y tiene una marca de overflow, se busca en los siguientes hasta encontrar el elemento o ̶h̶a̶s̶t̶a̶ ̶q̶u̶e̶ ̶s̶e̶ ̶e̶n̶c̶u̶e̶n̶t̶r̶e̶ ̶u̶n̶ ̶e̶s̶p̶a̶c̶i̶o̶ ̶v̶a̶c̶í̶o̶ hasta que se encuentre un espacio vacío que nunca se haya llenado.
2. Eliminación: Se elimina el elemento.

* **Saturación progresiva encadenada**: Lo mismo que la anterior pero se trabaja con punteros cuando hay overflow. La búsqueda es más eficiente.



* **Dispersión doble**: La saturación tiende a agrupar en zonas contiguas, y las búsquedas son largas cuando la densidad tiende a 1. Por ello en este método se busca almacenar registros de overflow en zonas no relacionadas: para esto se utiliza una segunda función de hash, el resultado se le suma a la dirección original tantas veces sean necesarias hasta encontrar espacio libre.



* **Encadenamiento en áreas separadas**: Los elementos que generan overflow van a un conjunto de nodos especiales.



**Hashing dinámico**

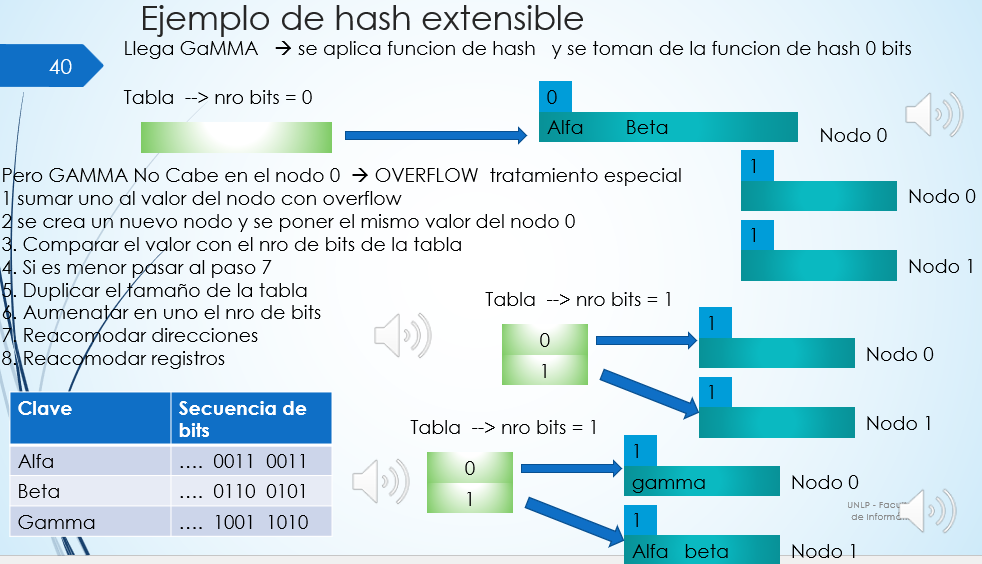
Cuando se utiliza espacio de direccionamiento estático llega un punto en el que se va a requerir **redispersar** los registros del archivo, ya que el archivo se llenará y se tendrá que generar un nuevo archivo, aumentando la cantidad de nodos, esto implica tener una nueva función de hash, por lo que todo el archivo cambia. Entonces, lo que buscamos es tener una **cantidad variable de direcciones con una única función de hash**.

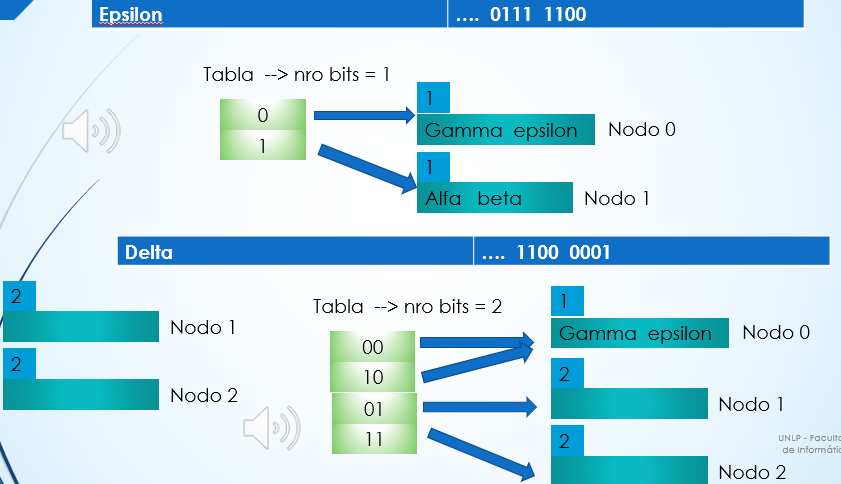
Solución → espacio de direccionamiento dinámico.

* Reorganizar el archivo sin mover muchos registros
* Técnicas que asumen bloques físicos, pueden utilizarse dinámicamente

.Varias posibilidades

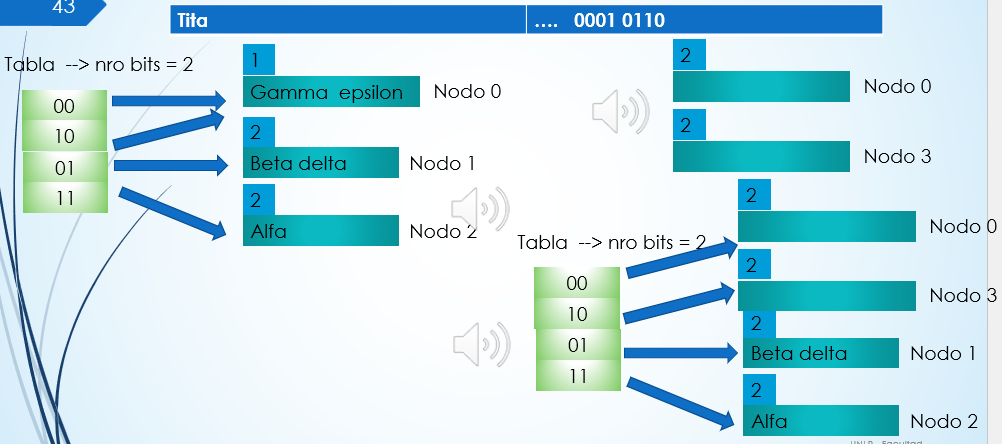
* Hash virtual
* Hash dinámico
* Hash **Extensible**: Garantiza UN solo acceso para la BÚSQUEDA. La inserción a veces requiere más de un acceso. **PASOS A SEGUIR**



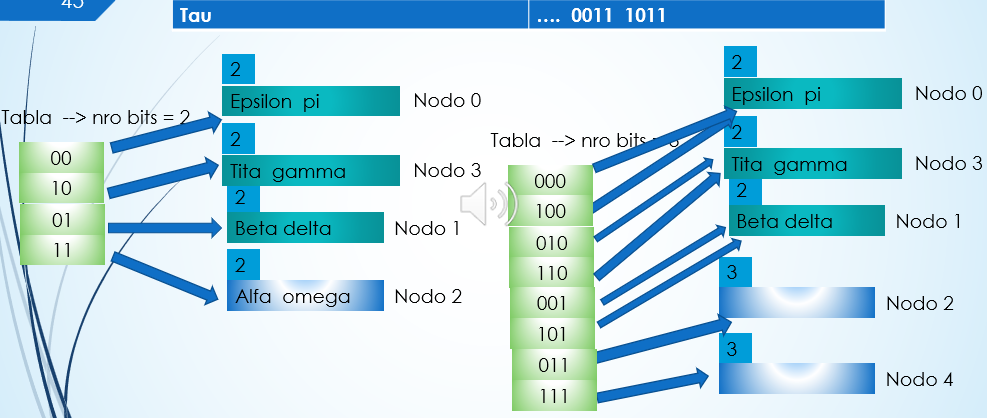


La que no genera overflow es apuntada por 2 direcciones. Ahora alfa va en el nodo 2 y beta y delta en nodo 1.

Caso en el que el valor del nodo es = al número de bits de la tabla:



Ahora con 3 bits:



ver últimas 2 diapositivas hash.