Capítulo 1

Base de Datos: Una colección o conjunto de datos interrelacionados con un propósito específico vinculado a la resolución de un problema del mundo real.

Sistema de Gestión de Bases de Datos (SGBD): consiste en un conjunto de programas necesarios para acceder y administrar una BD.

Un **SGBD** posee dos tipos diferentes de lenguajes: uno para especificar el esquema de una BD, y el otro para la manipulación de los datos.

- **Lenguaje de Definición de Datos (LDD)**: La definición del esquema de una BD implica el diseño de la estructura que tendrá efectivamente la BD; describir los datos, las relaciones entre ellos, la semántica asociada y las restricciones de consistencia. El resultado de compilar lo escrito con el LDD es un archivo llamado Diccionario de Datos. Un **Diccionario de Datos** es un archivo con Metadatos, es decir, datos acerca de los datos.
- **Lenguaje de Manipulación de Datos (LMD)**: mediante este se puede recuperar información, agregar nueva información y modificar o borrar datos existentes. Existen dos tipos de LMD:
    - **LMD procedimentales**: el usuario debe especificar qué datos requiere y cómo obtener esos datos.
    - **LMD no procedimentales**: el usuario debe especificar qué datos requiere, sin detallar cómo obtener esos datos.

Objetivos de un **SGBD**:

- **Controlar la concurrencia**: varios usuarios pueden acceder a la misma información en un mismo periodo de tiempo.
- **Tener control centralizado**: tanto de los datos como de los programas que acceden a los datos.
- **Facilitar el acceso a los datos**: dado que provee un lenguaje de consulta para recuperación rápida de información.
- **Proveer seguridad para imponer restricciones de acceso:** se debe definir explícitamente quiénes son los usuarios autorizados a acceder a la BD.
- **Mantener la integridad de los datos**

Niveles de visión de los datos - 3 niveles

3. **Nivel de vista:** corresponde al nivel más alto de abstracción. En este nivel se describe parcialmente la BD, solo la parte que se desea ver. Es posible generar diferentes vistas de la BD, cada una correspondiente a la parte a consultar.
2. **Nivel Lógico:** en este nivel se describe la BD completa, indicando qué datos se almacenarán y las relaciones existentes entre esos datos. El resultado es una estructura simple que puede conducir a estructuras más complejas en el nivel físico.
1. **Nivel Físico:** este es el nivel más bajo de abstracción, en el cual se describe cómo se almacenan realmente los datos. Se detallan las estructuras de datos más complejas de bajo nivel.

Modelo de datos: El modelo de datos es un conjunto de herramientas conceptuales que permite describir los datos y su semántica, sus relaciones y las restricciones de integridad y consistencia. Existen 3 grupos de modelos de datos.

1. **Modelos lógicos basados en objetos - Nivel de vista y lógico**
   Estos modelos son utilizados para representar la información de un problema del mundo real con un esquema de alto nivel de abstracción. Dentro de estos modelos se destacan el modelo de entidades y relaciones, y el modelo orientado a objetos.
2. **Modelos lógicos basados en registros - Nivel lógico y físico**
   Estos modelos utilizan la estructura de datos de registro para almacenar la información en una Base de Datos. Existen tres modelos basados en registros: modelo jerárquico, modelo de red y modelo relacional.
3. **Modelos físicos**

Categorías de usuarios de bases de datos

- **Administrador de BD**
- **Programadores de aplicaciones**
- **Usuarios sofisticados:** realizan consultas a la BD a través de un lenguaje de consulta; generalmente, son profesionales del ámbito informático.
- **Usuarios especializados:** son aquellos usuarios que desarrollan aplicaciones no tradicionales
- **Usuarios normales**: Acceso indirecto

Capítulo 2

Un **archivo** es una colección de registros que abarcan entidades con un aspecto común y originadas para algún propósito particular. Es una estructura de datos homogénea. Residen en dispositivos de memoria secundaria y no en RAM.
Existen dos diferencias básicas entre las estructuras de datos que residen en la memoria RAM y aquellas que lo hacen en almacenamiento secundario:

• Capacidad: la memoria RAM tiene capacidad de almacenamiento más limitada que, por ejemplo, un disco rígido.
• Tiempo de acceso: la memoria RAM mide su tiempo de acceso en orden de nanosegundos (10–9 segundos), en tanto que el disco rígido lo hace en el orden de milisegundos (10–3 segundos).

Archivo físico: es el archivo residente en la memoria secundaria y es administrado (ubicación, tipos de operaciones disponibles) por el sistema operativo.

Archivo lógico: es el archivo utilizado desde el algoritmo. Cuando el algoritmo necesita operar con un archivo, genera una conexión con el sistema operativo, el cual será el responsable de la administración. Esta acción se denomina independencia física.

Archivo - Tipo de dato: tipo de dato simple: entero, real o carácter. Además, puede contener una estructura heterogénea como lo son los registros. Los registros, en general, poseen longitud fija, determinada por la suma de las longitudes de los campos que los componen.

Acceso a la información contenida en archivos

Secuencial: el acceso a cada elemento de datos se realiza luego de haber accedido a su inmediato anterior. El recorrido es, entonces, desde el primero hasta el último de los elementos, siguiendo el orden físico de estos.
Secuencial indizado: el acceso a los elementos de un archivo se realiza teniendo presente algún tipo de organización previa, sin tener en cuenta el orden físico.
Directo: es posible recuperar un elemento de dato de un archivo con un solo acceso, conociendo sus características, más allá de que exista un orden físico o lógico predeterminado.

Operaciones básicas sobre archivos

Definición de archivos
Como cualquier otro tipo de datos, los archivos necesitan ser definidos. Se reserva la palabra clave file para indicar la definición del archivo. La siguiente sentencia define una variable de tipo archivo:

Var **archivo_logico**: file of tipo_de_dato;

donde **archivo_logico** es el nombre de la variable, **file** es la palabra clave reservada para indicar la definición de archivos y tipo_de_dato indica el tipo de información que contendrá el archivo.

Correspondencia archivo lógico-archivo físico
Se debe, entonces, indicar que el archivo lógico utilizado por el algoritmo se corresponde con el archivo físico administrado por el sistema operativo. La sentencia encargada de hacer esta correspondencia es:

Assign **(nombre_logico, nombre_fisico)**

donde **nombre_logico** es la variable definida en el algoritmo, y **nombre_fisico** es una cadena de caracteres que representa el camino donde quedará (o ya se encuentra) el archivo y el nombre del mismo.

Apertura y creación de archivos
Para abrir un archivo, existen dos posibilidades:
• Que el archivo aún no exista porque aún no fue creado.
• Que el archivo exista y se quiera operar con él.
Para ello, se dispone de dos operaciones diferentes:
• La operación **rewrite** indica que el archivo va a ser creado y, por lo tanto, la única operación válida sobre el mismo es escribir información. Si existe, se borra su contenido.
• La operación **reset** indica que el archivo ya existe y, por lo tanto, las operaciones válidas sobre el mismo son lectura/escritura de información. Si no existe, arroja error.

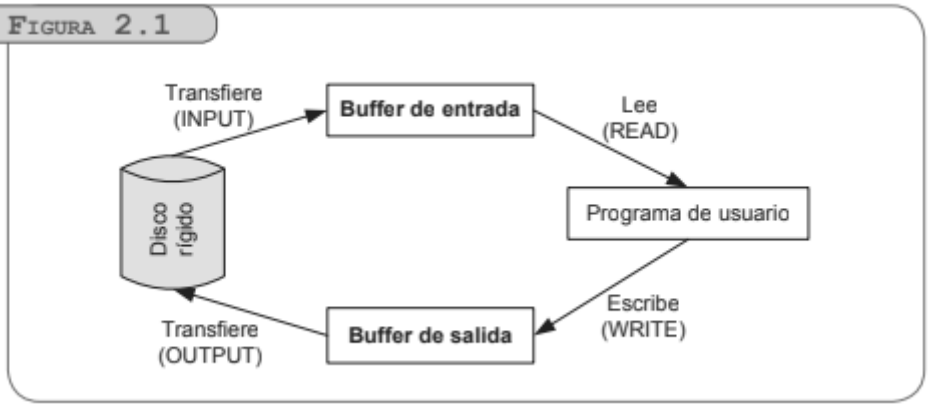
Cierre de archivos
Se realiza mediante la operación **close**(nombre_logico) y realiza una serie de eventos:

- Cada archivo necesariamente debe contener un fin. La marca de fin de archivo se conoce por su sigla en inglés **EOF** (end of file) y servirá posteriormente para controlar las lecturas sobre el archivo y no exceder la longitud del mismo.
- Transferir definitivamente la información volcada sobre el archivo a disco, es decir, transferir el buffer a disco.

Lectura y escritura de archivos
Las instrucciones son **read/write**(nombre_logico, var_dato); La variable **var_dato** y el tipo de dato de los elementos del archivo **nombre_logico** deben coincidir. Además, en caso de que **var_dato** corresponda a una variable de tipo registro, no debe realizarse una lectura o escritura campo a campo.

Buffers de memoria
Las lecturas y escrituras desde o hacia archivo se realizan sobre buffers. Se denomina buffer a una memoria intermedia (ubicada en RAM) entre un archivo y un programa, donde los datos residen provisoriamente hasta ser almacenados definitivamente en la memoria secundaria, o donde los datos residen una vez recuperados de dicha memoria secundaria.
La operación **read** lee desde un **buffer** y, en caso de no contar con información, el sistema operativo realiza automáticamente una operación **input**, trayendo más información al buffer. La diferencia radica en que cada operación input transfiere desde el disco una serie de registros. De esta forma, cada determinada cantidad de instrucciones read, se realiza una operación input.
De forma similar procede la operación **write**; en este caso, se escribe en el **buffer**, y si no se cuenta con espacio suficiente, se descarga el **buffer** a disco por medio de una operación **output**, dejándolo nuevamente vacío.

FIGURA 2.1



Creación de archivos
En la siguiente imagen vemos un ejemplo de código sobre la creación de un archivo de números reales.

EJEMPLO 2.3

```
Program CrearArchivo;
Type
     ArchivodeReales     = file of real;

Var
     Reales         : ArchivodeReales;
     NroReal        : real;

Begin
     {enlace entre el nombre lógico y el nombre físico}
     Assign( Reales,'c:\archivos\numerosreales');

    {apertura del archivo para creación}
    rewrite( Reales );
```

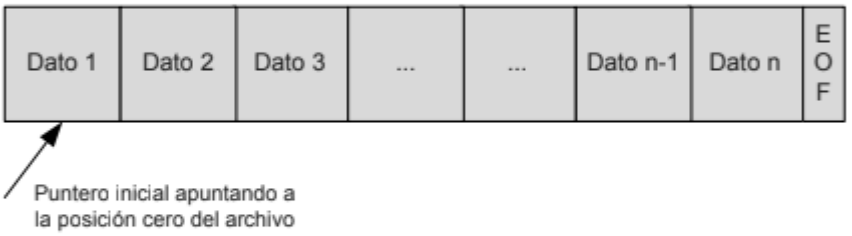
Operaciones adicionales con archivos

Puntero de trabajo de un archivo

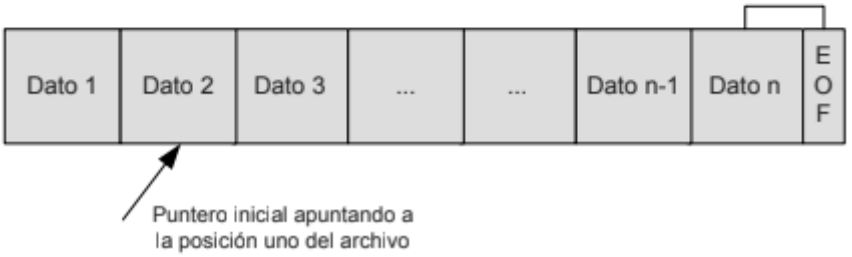
En el momento de ejecutarse la instrucción **reset**, el sistema operativo coloca un puntero direccionando al primer registro disponible dentro del archivo. Tanto la instrucción **read** como la **write** avanzan en forma automática el puntero una posición luego de ejecutarse.

Archivo antes de ejecutar un RESET

Archivo luego de ejecutar un RESET



Archivo luego de ejecutar un READ



Control de fin de archivo - EOF

Un algoritmo debe controlar el fin de archivo antes de realizar una operación de lectura/escritura. La función **eof**(nombre_logico);

Resultado de la función EOF() Falso

Resultado de la función EOF() Verdadero

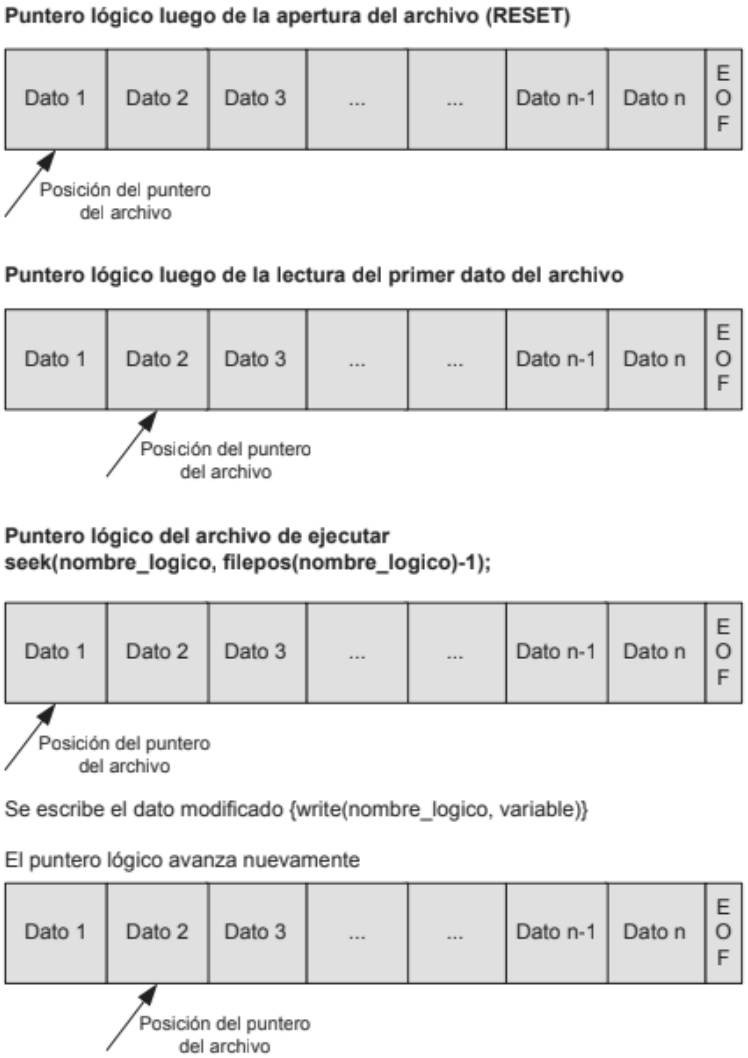
Control de tamaño del archivo
Devuelve un valor entero de la cantidad de elementos que contiene un archivo: filesize(nombre_logico)

Control de posición de trabajo dentro del archivo
Para conocer la posición actual del puntero del archivo, se utiliza la función filepos(nombre_logico) la cual retorna un número entero que indica la posición actual del puntero. Dicho valor estará siempre comprendido entre cero y la cantidad de elementos que tiene el archivo (filesize).

Ubicación física en alguna posición del archivo
Para modificar la posición del puntero, se dispone de la instrucción **seek**(nombre_logico, posición) donde **nombre_logico** indica el puntero del archivo a modificar y **posición** debe ser un valor entero (o variable de tipo entera) que indica el lugar donde será posicionado el puntero. Notar que **posición** debe estar entre 0 y filesize(nombre_logico), sino lanzará error.

IMPORTANTE: la instrucción seek(nombre_logico, filesize(nombre_logico)) dejará el puntero lógico del archivo referenciando a EOF, y las escrituras posteriores quitarán la marca.

Modificar la información de un archivo
En la siguiente imagen se muestra como trabaja el puntero ante la modificación del elemento **Dato 1** del archivo:



Capítulo 3

Archivo maestro: archivo que resume información sobre un dominio de problema específico. Ejemplo: el archivo de productos de una empresa que contiene el *stock* actual de cada producto.

Archivo detalle: archivo que contiene novedades o movimientos realizados sobre la información almacenada en el maestro. Ejemplo: el archivo con todas las ventas de los productos de la empresa realizadas en un día particular.

Actualización de un archivo maestro con un archivo detalle

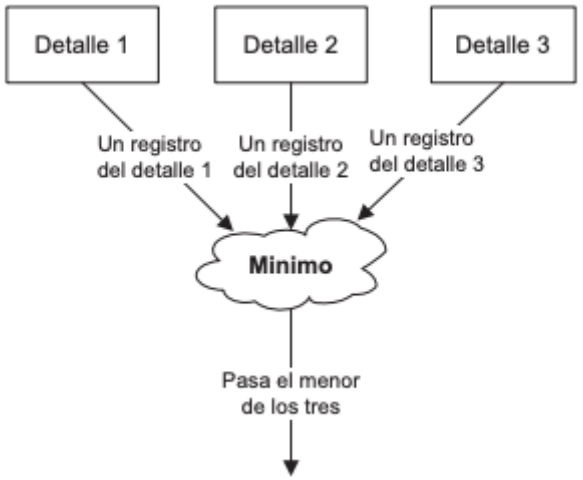
Presenta la variante más simple del proceso de actualización. Las pre- condiciones del problema son las siguientes:

- Existe un archivo maestro.
- Existe un único archivo detalle que modifica al maestro.
- Cada registro del detalle modifica a un registro del maestro. Esto significa que solamente aparecerán datos en el detalle que se correspondan con datos del maestro. Se descarta la posibilidad de generar altas en ese archivo.
- No todos los registros del maestro son necesariamente modificados.
- Cada elemento del maestro que se modifica es alterado por uno o más elementos del archivo detalle.
- Ambos archivos están ordenados por igual criterio. Esta precondición, considerada esencial, se debe a que hasta el momento se trabaja con archivos de datos de acuerdo con su orden físico.
- El proceso de actualización finaliza cuando se termina de recorrer el archivo detalle. Una vez procesados todos los registros del archivo detalle, el algoritmo finaliza, sin la necesidad de recorrer el resto del archivo maestro.

Actualización de un archivo maestro con N archivos detalle

Bajo las mismas consignas del ejemplo anterior, se plantea un proceso de actualización donde, ahora, la cantidad de archivos detalle se lleva a N (siendo N > 1) y el resto de las precondiciones son las mismas.

Para ello, se agrega un nuevo procedimiento, denominado **mínimo**, que actúa como filtro. El objetivo de este proceso a partir de la información recibida es retornar el elemento más pequeño de acuerdo con el criterio de ordenamiento del problema y avanzar una posición en ese archivo detalle. Cuando todos los detalles llegan a EOF, finaliza la actualización.



Merge - Proceso de generación de un nuevo archivo a partir de otros existentes
Este proceso recibe el nombre de merge o unión, y la principal diferencia con los casos previamente analizados radica en que el archivo maestro no existe, y por lo tanto debe ser generado.

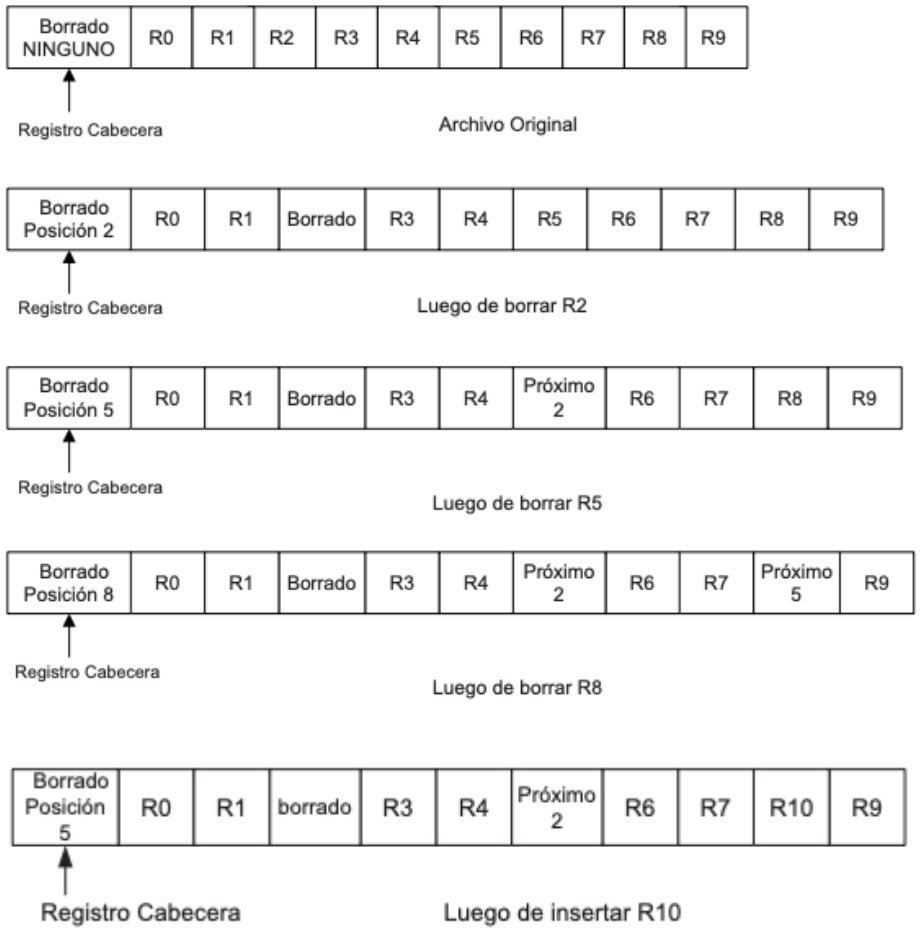
Corte de control
Proceso mediante el cual la información de un archivo es presentada en forma organizada de acuerdo con la estructura que tiene el archivo. El archivo tiene que en lo posible encontrarse ordenado para poder realizar este proceso.

Capítulo 4
Proceso de baja
Se denomina proceso de baja a aquel proceso que permite quitar información de un archivo.
Baja física: consiste en borrar efectivamente la información del archivo, recuperando el espacio físico. Tiene desventaja en cuanto a la performance. Existen dos técnicas algorítmicas para realizar esta operación:

- **Generar un nuevo archivo con los elementos válidos**, es decir, sin copiar los que se desea eliminar. Suponiendo que el archivo tiene **n** registros, **n** lecturas y **n**-1 escrituras; en ambos casos, las lecturas y escrituras se realizan en forma secuencial (es decir, para leer el registro **n**, antes se debe leer desde el registro 1 hasta el registro **n**-1) sobre ambos archivos. Una vez finalizado el proceso, coexisten en el disco dos archivos: el original y el nuevo. Significa que se debe disponer en memoria secundaria de la capacidad de almacenamiento suficiente para ambos archivos.
- **Utilizar el mismo archivo de datos**, generando los reacomodamientos que sean necesarios. El algoritmo requiere localizar, primero, al elemento a borrar. Luego, copiar sobre este registro el elemento siguiente y así sucesivamente, repitiendo esta operatoria hasta el final del archivo. El archivo no se cierra utilizando la instrucción **close**(); en su reemplazo se utilizó **truncate**(). La diferencia radica en que **close**() coloca la marca de fin luego del último elemento, y de este modo el **n-ésimo** registro quedaría repetido. La instrucción **truncate**(), en cambio, coloca la marca de fin en el lugar indicado por el puntero del archivo en ese momento, quitándose todo lo que hubiera desde esa posición en adelante.
  El **análisis de performance** determina que la cantidad de lecturas a realizar es n, en tanto que la cantidad de escrituras dependerá del lugar donde se encuentre el elemento a borrar; en el peor de los casos, deberán realizarse nuevamente **n-1 escrituras**, si el elemento a borrar apareciera en la primera posición del archivo. No es necesario contar con mayor capacidad en el disco rígido. Esto se debe a que se utiliza la ubicación original del archivo en memoria secundaria.

Baja lógica: consiste en borrar la información del archivo, pero sin recuperar el espacio físico respectivo. Esto es utilizando algún tipo de marca sobre el elemento que indique que se encuentra borrado. La ventaja del borrado lógico tiene que ver con la performance, basta con localizar el registro a eliminar y colocar sobre él una marca que indique que se encuentra no disponible. Entonces, la performance necesaria para llevar a cabo esta operación es de tantas lecturas como sean requeridas hasta encontrar el elemento a borrar, más una sola escritura que deja la marca de borrado lógico sobre el registro.
La desventaja de este método está relacionada con el espacio en disco. Al no recuperarse el espacio borrado, el tamaño del archivo tiende a crecer continuamente.

Recuperación de espacio
Periódicamente utilizar el proceso de baja física para realizar un proceso de compactación del archivo. El mismo consiste en quitar todos aquellos registros marcados como borrados (con baja lógica), utilizando para ello cualquiera de los algoritmos discutidos anteriormente para borrado físico.
Reasignación de espacio
Consiste en recuperar el espacio, utilizando los lugares indicados como borrados para el ingreso (altas) de nuevos elementos al archivo.
A medida que los datos se borran del archivo, se genera una lista encadenada invertida con las posiciones borradas. En la figura, primero se borra el registro 2, luego, el 5, y por último, el 8. Se puede observar cómo se lleva a cabo el proceso de borrado, indicando el espacio que queda libre. Al comienzo, en el registro cabecera del archivo solamente se indica que no hay registros borrados. A medida que se quitan elementos, este proceso se reitera en la lista de elementos borrados que se va construyendo.
Si el archivo no dispone de lugares para reutilizar, el registro cabecera no tiene ninguna dirección válida. En ese caso, para ingresar un nuevo elemento se debe acceder a la última posición del archivo, y agregarlo por ende al final.
Ejemplo:



Archivos con registros de longitud variable

La cantidad de espacio utilizada por cada elemento del archivo no está determinada a priori. Se tratará al archivo de datos como una secuencia de caracteres, donde EOF seguirá representando la marca de fin de archivo.

Características:

1. El archivo se encuentra definido como file y permite realizar la transferencia de la información carácter a carácter.
2. Cuando se termina de insertar un campo, se utiliza como marca de fin de campo el carácter #.
3. Cuando se termina de insertar un registro, se utiliza como marca de fin de campo el carácter @. Puede utilizarse explícitamente o no, depende del problema.

Mientras que con registros de longitud fija basta con definir el tipo de datos del archivo, para que tanto la operación read como write realicen la transferencia estructurada de información, con registros de longitud variable, la operatoria debe resolverse leyendo o escribiendo de a uno los caracteres que componen un registro.

La utilización de espacio en disco es optimizada, respecto del uso, con registros de longitud fija. Sin embargo, esto conlleva un algoritmo donde el programador debe resolver en forma mucho más minuciosa las operaciones de agregar y quitar elementos.

Proceso de baja física y lógica similar a longitud fija. Para la reasignación del espacio de bajas lógicas se genera una lista invertida donde a partir de un registro cabecera se dispone de las direcciones libres dentro del archivo. Es necesario ahora indicar, además, la cantidad de bytes disponibles en cada caso para su reutilización.

El proceso de inserción debe localizar el lugar dentro del archivo más adecuado al nuevo elemento. Existen tres formas genéricas para la selección de este espacio:

- **Primer ajuste**: consiste en seleccionar el primer espacio disponible donde entre. Tiende a ser la técnica más rápida y eficiente.
- **Mejor ajuste**: consiste en seleccionar el espacio más adecuado para el registro, aquel de menor tamaño donde quepa el registro.
- **Peor ajuste**: consiste en seleccionar el espacio de mayor tamaño, asignando para el registro sólo los bytes necesarios. Mantiene la filosofía de trabajo de longitud variable

Fragmentación interna: se produce cuando a un elemento de datos se le asigna mayor espacio del necesario. (Primer ajuste y mejor ajuste)

Fragmentación externa: el espacio disponible entre dos registros, pero que es demasiado pequeño para poder ser reutilizado. (Peor ajuste)

Modificación en registros de longitud variable

Se puede suponer que si el nuevo elemento ocupa menos espacio que el anterior, no se genera una situación problemática dado que el espacio disponible es suficiente, aunque en ese caso se generaría fragmentación interna.

Si la modificación aumenta el tamaño del registro, se estila dividir el proceso de modificación en dos etapas: en la primera se da de baja al elemento de dato viejo, mientras que en la segunda etapa el nuevo registro es insertado de acuerdo con la política de recuperación de espacio determinada.

Búsqueda de datos

Teniendo un archivo serie sin ningún orden preestablecido más que el físico. La performance depende de la cantidad de registros en el archivo, siendo de orden N:

Mejor caso **1 acceso**
Peor caso **N accesos**
Promedio **N/2 accesos**

Búsqueda binaria

Si se dispone de un archivo con registros de **longitud fija** y además **físicamente ordenado**, es posible mejorar esta performance de acceso si la búsqueda se realiza con el **mismo argumento que el utilizado para ordenar** este archivo.
La primera comparación del dato que se pretende localizar es contra el registro medio del archivo, es decir, el que tiene NRR=N/2. Si el registro no contiene ese dato, se descarta la mitad mayor o menor, según corresponda, reduciendo el espacio de búsqueda a los N/2 registros restantes (mitad restante). Nuevamente, se realiza la comparación pero con el registro medio de la mitad restante, repitiendo así este proceso hasta reducir el espacio de búsqueda a un registro.
En general, una búsqueda binaria en un archivo con N registros se realiza en a lo sumo log2 (N)+1 comparaciones y en promedio (log2 (N)+1)/2 comparaciones. Por lo tanto, se concluye que la búsqueda binaria es de orden log2 (N), logrando mejorar sustancialmente el caso anterior. No obstante, se debe considerar el costo adicional de mantener el archivo ordenado para posibilitar este criterio de búsqueda, y el número de accesos disminuye pero aún dista bastante de recuperar la información en un acceso a disco

Si disponemos del NRR del registro podemos acceder directamente al registro deseado. Cada registro tiene un número relativo dentro del archivo. Dicho número se denomina **NRR** y permite calcular la distancia en bytes desde el principio del archivo hasta cualquier registro. Esta situación, si bien posibilita acceso directo, no deja de ser especial, ya que es muy poco probable conocer el NRR del registro que contiene el dato a buscar.

Sort interno

Descartando la posibilidad de ordenar directamente sobre memoria secundaria, por los costos. Esta estrategia consiste en los siguientes pasos:
1. Dividir el archivo en particiones de igual tamaño, de modo tal que cada partición quepa en memoria principal.
2. Transferir las particiones (de a una) a memoria principal. Esto implica realizar lecturas secuenciales sobre memoria secundaria, pero sin ocasionar mayores desplazamientos.
3. Ordenar cada partición en memoria principal y reescribirlas ordenadas en memoria secundaria. También en este caso la escritura es secuencial.

Luego del **sort interno**, se debe realizar el merge o fusión de las particiones, generando un nuevo archivo ordenado. Esto implica la lectura secuencial de cada partición nuevamente y la reescritura secuencial del archivo completo.
Se puede concluir que el **ordenamiento** es una operación de $O(N^2)$ evaluada en términos de desplazamiento (N (particiones) * N (cantidad de desplazamientos)).

Selección por reemplazo

Consiste en seleccionar siempre de memoria principal la clave menor de registro, enviarla a MS (partición generada) y reemplazarla por una nueva clave que está esperando ingresar a

MP. La clave que ingresa a MP debe ser marcada **como no disponible si es menor** a la enviada a la partición generada en MS.

Claves en memoria secundaria:

34, 19, 25, 59, 15, 18, 8, 22, 68, 13, 6, 48, 17

↑

Principio de la cadena de entrada

Resto de la entrada	Mem. principal	Partición generada
34, 19, 25, 59, 15, 18, 8, 22, 68, 13	6 48 17	-
34, 19, 25, 59, 15, 18, 8, 22, 68	13 48 17	6
34, 19, 25, 59, 15, 18, 8, 22	68 48 17	13, 6
34, 19, 25, 59, 15, 18, 8	68 48 22	17, 13, 6
34, 19, 25, 59, 15, 18	68 48 (8)	22, 17, 13, 6
34, 19, 25, 59, 15	68 (18) (8)	48 ,22, 17, 13, 6
34, 19, 25, 59	(15) (18) (8)	68, 48, 22, 17, 13, 6

Generalizando, se puede establecer que, en promedio, **este método aumenta el tamaño de las particiones al doble** (**o más** si se encuentra parcialmente ordenada la entrada de claves) de la cantidad de registros que se podrían almacenar en memoria principal (2P) como se ve en el ejemplo anterior.

Selección natural

Variante de selección por reemplazo que reserva y utiliza memoria secundaria en la cual se insertan los registros no disponibles. De este modo, la generación de una partición finaliza cuando este nuevo espacio reservado está en overflow (completo).

• Ventajas:
  – **Sort interno**: produce particiones de igual tamaño. Algorítmica simple. No es un detalle menor que las particiones tengan igual tamaño; cualquier variante del método de merge es más eficiente si todos los archivos que unifica son de igual tamaño.
  – **Selección natural y selección por reemplazo**: producen particiones con tamaño promedio igual o mayor al doble de la cantidad que caben en memoria principal.
• Desventajas:
  – **Sort interno**: es el más costoso en términos de performance.
  – **Selección natural y selección por reemplazo**: tienden a generar muchos registros no disponibles. Las particiones no quedan siempre de igual tamaño.

Merge en más de un paso

Ejemplo: (archivo de 800.000 registros y 1 Mb de memoria principal, donde cada registro ocupa 100 bytes y la clave, 10 bytes). En este caso, en vez de realizar el merge entre 80 particiones, se realizará el merge de 10 conjuntos de 8 particiones cada una. Se generan, así, 10 archivos intermedios, los cuales deberán sufrir otro merge para obtener el archivo original ordenado. Mejora la cantidad total de desplazamientos.

Índice
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. Es necesario que las claves permanezcan ordenadas.
Esta estructura de datos es otro archivo con registros de longitud fija, independientemente de la estructura del archivo original. La característica fundamental de un índice es que posibilita imponer orden en un archivo sin que realmente este se reacomode
Para encontrar un dato solo se realiza una búsqueda con potencial bajo costo en el índice y luego un acceso directo al archivo de datos, por lo que la cantidad de accesos a memoria secundaria está condicionada por la búsqueda en el índice.

Creación de índice primario
Al crearse el archivo, se crea también el índice asociado, ambos vacíos, solo con el registro encabezado.

Alta en índice primario
Se agrega el nuevo registro al final del archivo, a partir de esta operación, con el NRR o la dirección del primer byte, según corresponda, se genera un nuevo registro de datos a insertar de forma ordenada en el índice.

Modificaciones en índice primario
Se considera la posibilidad de cambiar cualquier parte del registro excepto la clave primaria. Si el archivo está organizado con registros de longitud fija, el índice no se altera.

Si el archivo está organizado con registros de longitud variable y el registro modificado no cambia de longitud, nuevamente el índice no se altera. Si el registro modificado cambia de longitud, particularmente agrandando su tamaño, este debe cambiar de posición, es decir, debe reubicarse. En este caso, esta nueva posición del registro es la que debe quedar asociada en la clave primaria respectiva del índice.

Bajas en índice primario
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 del índice primario
- Al ser de menor tamaño que el archivo asociado y tener registros de longitud fija posibilita mejorar la performance de búsqueda.
- Permite búsqueda binaria (Archivo ordenado y de long. fija)

Índices para claves candidatas
Las claves candidatas son claves que no admiten repeticiones de valores para sus atributos, similares a una clave primaria, pero que por cuestiones operativas no fueron seleccionadas como clave primaria.

Índices secundarios
Un índice secundario es una estructura adicional que permite relacionar una clave secundaria con una o más claves primarias, dado que varios registros pueden contener la misma clave secundaria.
Para acceder a un dato, primero se accede al índice secundario por la clave secundaria; allí se obtiene la clave primaria, y se accede con dicha clave al índice primario para obtener la dirección efectiva del registro buscado.
¿Por qué motivo el índice secundario no posee directamente la dirección física de elemento de dato? Al tener la dirección física solamente definida para la clave primaria, si el registro cambia de lugar en el archivo, solo debe actualizarse la clave primaria en el índice primario.

Clave secundaria	Clave primaria
A-ha	SON15
A-ha	VIR1323
A-ha	WAR23
Cock Robin	SON13
Eurythmics	ARI2313

Creación de índice secundario
Similar a la creación de índices primarios, se crean vacíos al crear el archivo.

Altas de índice secundario
Cualquier alta en el archivo de datos genera una inserción en índice secundario.

Modificaciones en índice secundario
Cuando cambia la clave secundaria, se reacomoda el índice secundario. Cuando cambia el resto del registro de datos (Menos clave secundaria y primaria), no cambia nada.

Bajas en índice secundario
- Cuando se elimina un registro del archivo de datos, se debe eliminar la referencia de ese registro del índice primario y todas las referencias en índices secundarios
- La eliminación en índice secundario, almacenado en un archivo de longitud fija, puede ser física o lógica.
- **Alternativa**: Sí se borra solo la referencia en el índice primario, el índice secundario no se altera, pero el índice secundario ocuparía datos que ya no existen (problema si no se dispone de espacio en memoria secundaria).

Índices selectivos
Existe otra posibilidad que consiste en disponer de índices que incluyan solo claves asociadas a una parte de la información existente, es decir, aquella información que tenga mayor interés de acceso. En el ejemplo presentado, se podría tener un índice con las canciones del grupo musical A-ha solamente.

Capítulo 6
Arboles binarios
Un **árbol binario** es una estructura de datos dinámica no lineal, en la cual cada nodo puede tener a lo sumo dos hijos. La estructura de datos árbol binario en general tiene sentido cuando está ordenado. Por lo general, a la izquierda de un elemento se encuentran los elementos menores que él, y a la derecha, los mayores. Presentan mejoras en el mantenimiento y la búsqueda.

La **búsqueda** en este tipo de estructuras se realiza a partir del nodo raíz y se recorre, explorando hacia los nodos hoja.
La búsqueda de un elemento es del **orden log2 (N)**, siendo N la cantidad de elementos distribuidos en el árbol (**Si se encuentra balanceado**). En caso de que el árbol **no se encuentre balanceado**, la performance de búsqueda tenderá a tener **orden lineal**.
Estructura del árbol binario y nodos

Raíz → 0

	Clave	Hijo Izq	Hijo Der
0	MM	1	2
1	GT	3	4
2	ST	8	11
3	BC	5	6
4	JF	7	14
5	AB	-1	-1
6	CD	-1	-1
7	HI	-1	-1

	Clave	Hijo Izq	Hijo Der
8	PR	9	10
9	OP	-1	-1
10	RX	-1	-1
11	UV	12	13
12	TR	-1	-1
13	ZR	-1	-1
14	KL	-1	-1

Cada nodo es un registro con su elemento de dato (clave primaria y NRR) y los NRR de el hijo izquierdo y derecho.

Inserción en un árbol binario
La principal ventaja del uso de árbol binario es la inserción, que no causa desorden. Realiza los siguientes pasos:

- Agrega el nuevo elemento al final del archivo
- Busca al padre de dicho elemento (De acuerdo con el orden del árbol)
- Actualiza el padre, haciendo referencia al nuevo hilo

Esta operación realiza **log2 (N) lecturas** y **2 escrituras**, la primera para agregar al final el elemento y la segunda para actualizar el elemento del padre con la nueva referencia.

Borrar un elemento en un árbol binario
Necesariamente debe ser un elemento terminal (una hoja). Performance (creo): **Log2 (N) lecturas** y **una escrituras**
Si no es terminal, se debe intercambiar por el menor de sus hijos mayores. En este caso la performance es de **Log2 (N) lecturas** y **dos escrituras**.

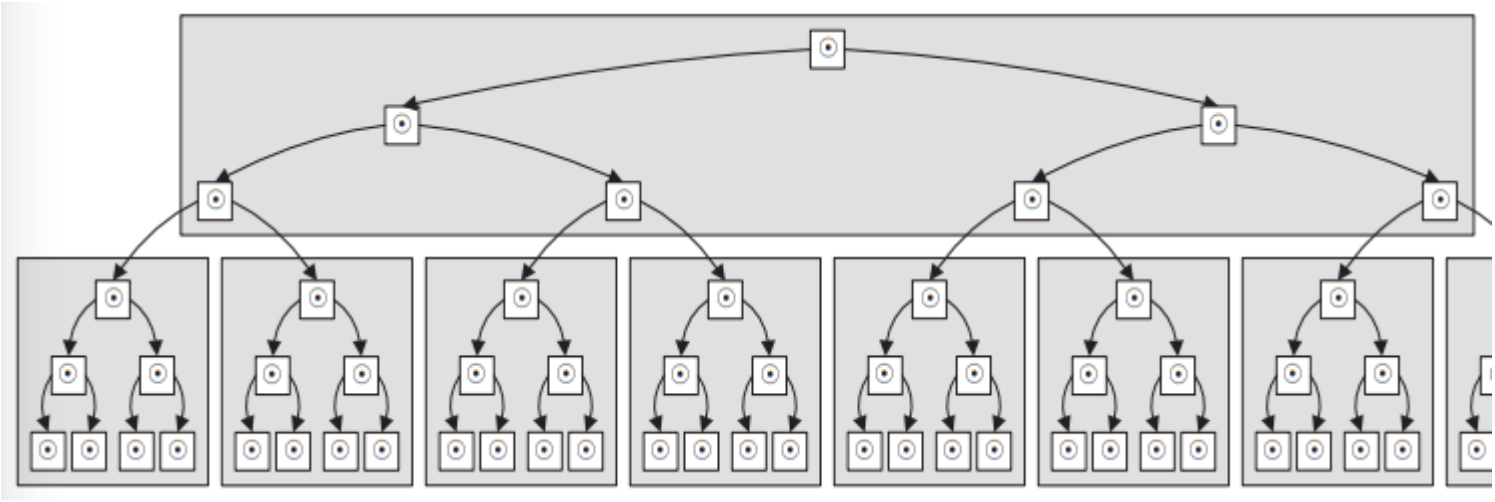
Archivos de datos vs Índices de datos
Si un archivo tiene 4 criterios de ordenamiento, **se requerirán 4 árboles diferentes** (c/u con el atributo por el cual se ordena). Estos sirven de índice para el archivo de datos(archivo serie).

Árbol balanceado
Se entiende por árbol balanceado a aquel árbol donde la trayectoria de la raíz a cada una de las hojas está representada por igual cantidad de nodos. Es decir, todos los nodos hoja se encuentran a igual distancia del nodo raíz.
Si el árbol binario está desbalanceado, la performance de búsqueda será de orden lineal, ya que se transforma en una estructura tipo lista.
¿Cuando un árbol Binario es buena elección? Cuando el árbol se encuentra balanceado.

Arboles AVL
Los árboles balanceados en altura son árboles binarios cuya construcción se determina respetando un precepto muy simple: la diferencia entre el camino más corto y el más largo entre un nodo terminal y la raíz no puede diferir en más que un determinado **N**, y dicho **N** es el nivel de balanceo en altura del árbol.
Es una estructura capaz de mantener el balanceo del árbol, pero asumiendo mayores costos de inserción y borrado, ya que ante un desbalanceo, se debe rebalancear el árbol y esto tiene un costo alto de performance en cuanto accesos a disco. Por esto último, no presentan una solución viable para los índices de archivos de datos.

Paginación de Árboles Binarios
El árbol se divide en páginas (se pagina) y cada página contiene un conjunto de nodos ubicados en direcciones físicas cercanas. De esta manera, se almacenan las páginas en el buffer y el buffer transfiere al disco duro (desde o hacia). De esta forma, se mejora la performance ya que se hacen menos lecturas en el disco.

Ejemplo: Una organización de este tipo reduce el número de accesos a disco necesarios para poder recuperar la información. En el árbol de la imagen se puede notar que, al transferir la primera página, se están recuperando siete nodos del árbol, los que representan los primeros tres niveles del árbol. Así, y siguiendo dicha figura, con solo dos accesos a disco es posible recuperar un nodo específico entre 63.



La **performance de búsqueda** es de Orden $\mathbf{Log_{k+1}(N)}$ siendo K la cantidad de nodos por página y N la cantidad de claves por archivo.

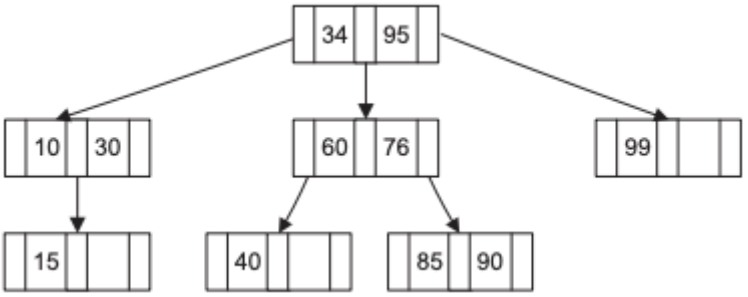
Dividir el árbol en páginas implica un costo extra necesario para su reacomodamiento y para mantener su balanceo interno. Insertar y borrar en un **árbol binario paginado** es muy costoso de implementar y bajo en performance para su mantenimiento.

Arboles multicamino

Un árbol multicamino es una estructura de datos en la cual cada nodo puede contener **K** elementos y **K+1** hijos.

Se define el concepto de orden de un árbol multicamino como la máxima cantidad de descendientes posibles de un nodo. Un árbol multicamino de orden M contendrá un máximo de M descendientes por nodo.

También tenemos el problema de los costos (performance) en el balanceo del mismo.

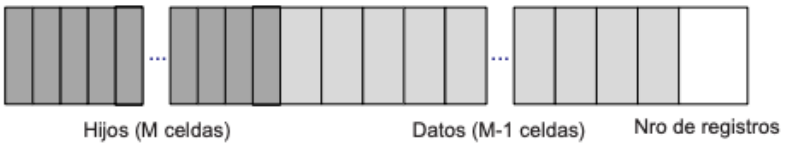


Capítulo 7

Arboles B

Los árboles B son árboles multicamino con una construcción especial que permite mantenerlos balanceados a bajo costo. Un árbol B de orden M posee las siguientes propiedades básicas:

1. Cada nodo del árbol puede contener, como máximo, M descendientes y M-1 elementos.
2. La raíz no posee descendientes directos o tiene al menos dos.
3. Un nodo con x descendientes directos contiene x-1 elementos.
4. Los nodos terminales (hojas) tienen, como mínimo, [M/2] – 1 elementos, y como máximo, M-1 elementos.
5. Los nodos que no son terminales ni raíz tienen, como mínimo, [M / 2] elementos.
6. Todos los nodos terminales se encuentran al mismo nivel.



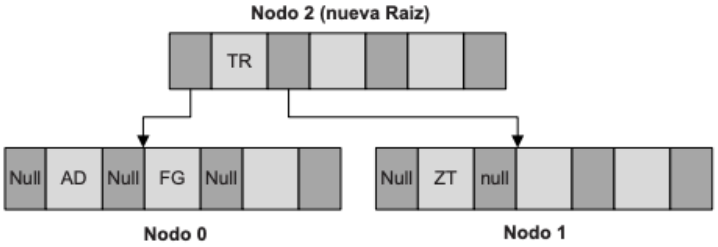
Formato del nodo para archivo del índice árbol b

Formato gráfico del nodo del índice árbol B

Creación de árboles B
Los nodos terminales (hojas) siempre están al mismo nivel.
Se comienza con una estructura vacía, ingresan elementos a la raíz hasta que esta se llene, cuando se produce un **overflow**:

- Se crea un nuevo nodo.
- La primera mitad se queda en el nodo viejo.
- La segunda mitad de las claves se traslada al nuevo nodo.
- La menor de las claves de la segunda mitad se promociona al nodo padre.

Llega el elemento ZT y produce overflow



Búsqueda en un árbol B
Comenzando por la raíz, se procede a buscar el elemento (al igual que el proceso de inserción). Si no se encuentra, se procede a buscar en el nodo inmediato siguiente que debería contener al elemento, procediendo de esa forma hasta encontrar dicho elemento, o hasta encontrar un nodo sin hijos que no incluya al elemento. El proceso de búsqueda en un árbol B no difiere mucho al de un árbol binario.

Eficiencia de búsqueda en Árbol B
La eficiencia de búsqueda en un árbol B consiste en contar los accesos al archivo de datos, que se requieren para localizar un elemento o para determinar que el elemento no se encuentra. El resultado es un valor acotado en el rango entero [1, H], siendo H la altura del árbol.

Eficiencia de inserción en Árbol B
Todos los elementos se insertan en nodos terminales. **H** lecturas para encontrar ese nodo. Por lo tanto, si no posee overflow, solamente necesita una escritura en ese nodo. Si posee overflow, se hacen 2 escrituras por cada overflow + 1 escritura de la nueva raíz.
Resumiendo:

H lecturas, 1 escritura (Mejor caso)
H lecturas, (2 * H) + 1 escrituras (Peor caso)

Definiciones
Se denomina **nodos hermanos** a aquellos nodos que tienen el mismo nodo padre.
Se denomina **nodos hermanos adyacentes** a aquellos nodos que, siendo hermanos, son además dependientes de punteros consecutivos del padre.

Eliminación en Árbol B
El elemento a borrar debe estar localizado en un nodo terminal. Si el elemento no está en una hoja, debe ser intercambiado por el elemento mayor de sus claves menores, o el menor de sus claves mayores (mejor opción).
Al eliminar el elemento del nodo terminal puede ocurrir 2 situaciones:

- Si al borrar el elemento del nodo terminal la cantidad de elementos no queda por debajo de la mínima[M / 2] - 1 elementos), el proceso finaliza con éxito.
- Al borrar el elemento se genera un underflow (nodo queda debajo de la mínima):
    - Lo más inmediato es concatenar (ambos nodos y el nodo padre) pero se debe chequear que sea posible. La concatenación puede propagarse hasta la raíz, disminuyendo en 1 la altura/nivel del árbol.
    - La otra alternativa es redistribuir. Esto es, que algún hermano adyacente comparta sus elementos, una mitad de los elementos involucrados va a la izquierda, otro a la derecha y un elemento sube como separador al nodo padre. LA CANTIDAD DE NODOS NO VARÍA.

Eficiencia de eliminación en Árbol B
Mejor caso: Eliminar elemento de nodo terminal sin que genere underflow.
Peor caso: Concatenar propagando a la raíz.

	Mejor caso	Peor caso
Cantidad de lecturas	H	$2 * H - 1$
Cantidad de escrituras	1	$H - 1$

Modificación en un Árbol B
La opción más simple es proceder como una baja del elemento anterior y una alta del nuevo elemento.

Conclusiones sobre Árboles B

- Buena solución para el manejo de índices asociados a un archivo de datos. VIABLE
- Usado para administrar índices de claves primarias, candidatas o secundarias. El árbol B está formado por claves, hijos y referencias del resto del registro en el archivo de datos original (archivo serie).
- Índices de claves secundarias o candidatas siempre referencian a la clave primaria (Por performance). Esto es porque si se modifica la posición física de un registro, solamente se modifica en el índice de las claves primarias.

Arboles B*

Variante del árbol B*, plantea una redistribución en caso de overflow también. A diferencia del árbol B que solo es ante underflow. Esto demorará la generación de nuevos nodos. El árbol crece más lento y mejora la performance.

La división de árboles B* en caso de overflow produce nodos completos en ⅔ partes y no con ½ de elementos como en el árbol B. Es decir, a partir de 2 nodos completos, se generan 3 nodos completos en ⅔ partes.

Propiedades de un árbol B*:

- Cada nodo del árbol puede contener, como máximo, M descendientes y M-1 elementos.
- La raíz no posee descendientes o tiene al menos dos.
- Un nodo con x descendientes contiene x-1 elementos.
- Los nodos terminales tienen, como mínimo, [(2M-1)/3] -1 elementos, y como máximo, M-1 elementos.
- Los nodos que no son terminales ni raíz tienen, como mínimo, [(2M-1) / 3] descendientes.
- Todos los nodos terminales se encuentran al mismo nivel.

IMPORTANTE: Cuando se completa la raíz de un árbol B* y se divide, es el único momento que los dos nodos hijos creados solo completarán su espacio a la mitad, es decir, ½ y no ⅔.

La búsqueda y baja en Árbol B* es igual a la del árbol B. Lo único que cambia es la cantidad mínima que tiene que tener cada nodo en caso de underflow (Para la baja).

Inserción sobre árboles B*

La inserción sin overflow es igual a la de un árbol B. Ante un caso de overflow, existe la posibilidad de redistribuir antes de dividir. Esta alternativa de redistribución es regulada mediante 3 políticas de redistribución posibles: De un lado, de un lado u otro lado, o política de un lado y otro lado. Esto es, que nodo se debe tener en cuenta al redistribuir.

Ejemplos:

- Política a un lado: Teniendo **política de derecha**, si se completa un nodo se intenta redistribuir con el hermano adyacente derecho, si está lleno entonces se debe dividen los 2 nodos a 3 nodos con ⅔ partes de elementos. Como excepción, si el nodo **no tiene** hermano adyacente derecho, se intenta redistribuir o se divide con el izquierdo.
- Política de un lado o otro lado: Alternativa a la anterior. Teniendo **política de derecha o izquierda**, en caso de producirse una saturación en un nodo, se intenta primero redistribuir con un adyacente hermano. De no ser posible la redistribución, se intenta con el otro adyacente hermano. Si nuevamente la redistribución no es posible, la alternativa es dividir de 2 nodos llenos a 3 nodos ⅔ llenos. Siguiendo la política de derecha o izquierda presentada en este problema, se divide con el **nodo adyacente hermano derecho**.
- Política de un lado y otro lado: Alternativa a la anterior. Primero, redistribuir hacia un lado y, si no es posible, con el otro hermano. La diferencia aparece cuando los tres nodos están completos. Aquí se toman los tres nodos y se generan cuatro nodos con ¾ partes completas cada uno. El problema de esta política es la cantidad de operaciones de entrada-salida.

Performance en inserción Árbol B*

Como **mínimo cada una de las políticas requiere dos lecturas** (el nodo que se satura y un adyacente hermano) **y tres escrituras**. No se contabiliza la lectura en el nodo padre, ya que una inserción se realiza sobre nodos terminales y para acceder a los mismos necesariamente se hace a través del padre.

En caso de necesitar realizar una división:

- Política de un lado: 4 escrituras (Más las dos lecturas)
- Política de un lado u otro lado: 4 escrituras (Más las dos lecturas)
- Política de un lado y otro lado: 4 escrituras (Más las dos lecturas)

Manejo de buffers. Arboles B virtuales

- La performance del árbol depende de su altura (H)
- La capacidad de cada nodo está determinada inicialmente por la capacidad que el SO puede manejar a través de Buffers
- Política LRU (Last Recently Used): Se mantienen en memoria principal los Buffers (que contienen nodos) más utilizados últimamente, esto implica que un árbol con acceso frecuente contenga la raíz en el buffer, disminuyendo los accesos al disco.

Acceso secuencial indizado

Se denomina archivo con acceso secuencial indizado a aquel que permite dos formas para visualizar la información:

1. **Indizada:** el archivo puede verse como un conjunto de registros ordenados por una clave o llave.
2. **Secuencial:** se puede acceder secuencialmente al archivo, con registros físicamente contiguos y ordenados nuevamente por una clave o llave.

Mostrar los datos de forma secuencial ordenada con un **árbol B** es muy ineficiente.

Archivos físicamente ordenados a bajo costo

A raíz del problema con el acceso secuencial, se llega a la conclusión de que debe tratarse a cada nodo como un nodo terminal de un árbol B y que cada nodo terminal debe estar enlazado a su hermano.

Esta solución presenta una ventaja concreta, si bien el archivo no se encuentra físicamente ordenado, cada bloque si lo está, y como cada nodo se encuentra enlazado con el siguiente, el recorrido secuencial se realiza a muy bajo costo.

Ejemplo: Teniendo el nodo de la Figura 7.21 b), se ingresa “Estevez”, esto produce overflow y se realiza la división de árbol B, manteniendo el orden secuencial y los enlaces.

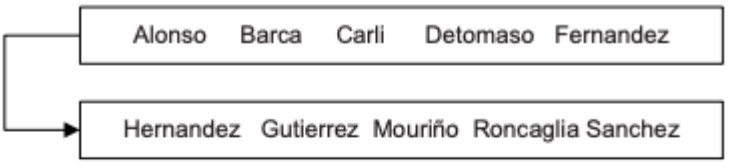
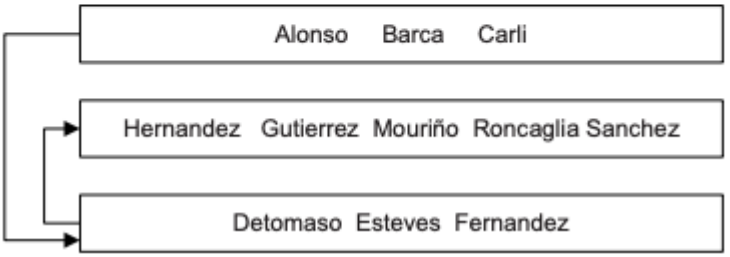


Figura 7.21 b)



Arboles B+

El árbol B+ incorpora las características del Árbol B además del tratamiento secuencial ordenado del archivo.

Propiedades del árbol B+:

- Cada nodo del árbol puede contener, como máximo, M descendientes y M-1 elementos.
- La raíz no posee descendientes o tiene al menos dos.
- Un nodo con x descendientes contiene x-1 elementos.
- Los nodos terminales tienen, como mínimo, ([M/2] – 1) elementos, y como máximo, M-1 elementos.
- Los nodos que no son terminales ni raíz tienen, como mínimo, [M/2] descendientes.
- Todos los nodos terminales se encuentran al mismo nivel.
- Los nodos terminales representan un conjunto de datos y son enlazados entre ellos.

Arriba se establece la principal diferencia entre un árbol B y un árbol B+. Para poder realizar **acceso secuencial ordenado** a todos los registros del archivo, es necesario que cada elemento (clave asociada a un registro de datos) aparezca almacenado en un nodo terminal.

Cuando ocurre la primera división, el nodo raíz contendrá un elemento que actúa como separador (es una copia), y la clave o elemento real (de la copia) estará en un nodo terminal. Solamente cuando ocurre una división en un nodo terminal se promociona una copia.

En caso de dividir un nodo no terminal, se debe promocionar hacia el padre el elemento en sí y no una copia del mismo.

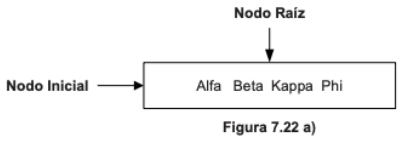
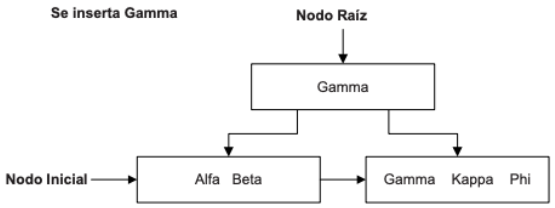


Figura 7.22 a)



Borrado en árbol B+: Al borrar siempre se hace a nivel hoja (nodo terminal), y si hubiera una copia del elemento a eliminar, esa copia no se borra ya que actúa como separador.

Árboles B+ de prefijos simples

Hacen que los separadores utilicen el mínimo espacio necesario que permita decidir si buscar a la derecha o izquierda y así aprovechar el espacio físico. Por ejemplo, si se separa utilizando la clave Gamma y solamente con la letra G alcanza para utilizarla como separador, se utiliza solo G. Así, se aprovecha el espacio al máximo.

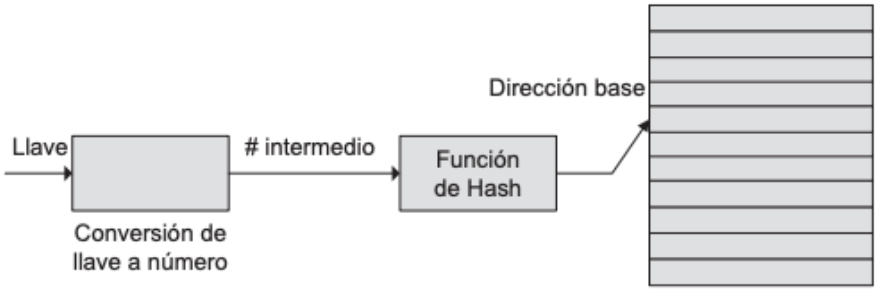
Arboles balanceados - Conclusiones
Características compartidas entre los árboles de la familia B:

- Manejo de nodos → facilita operaciones e/s y mantiene un número bajo de accesos
- Todos los nodos terminales están a la misma altura → balanceados
- Crecen de abajo hacia arriba
- Son bajos y anchos
- Se pueden implementar con registros de longitud variable

Arboles B* mejoran la eficiencia de los árboles b, aumentando la cantidad mínima de elementos en un nodo → disminuye la altura final. El costo de esto es que las operaciones de inserción son un poco más lentas.
En los árboles B+ toda la información está en los nodos terminales, los nodos no terminales actúan como separadores y poseen una copia de los elementos contenidos en los nodos terminales. Y además poseen un acceso secuencial a bajo costo.

Capítulo 8
Dispersión (Hashing)
Un archivo directo es un archivo en el cual cualquier registro puede ser accedido directamente. El hashing es un mecanismo que trata de asegurar la rápida recuperación de registros, en un solo acceso promedio.
Métodos de búsqueda más eficientes:

- En los árboles B+ son necesarios en promedio 3 o 4 accesos a memoria secundaria para recuperar el registro. El hashing asegura un promedio de 1 acceso para recuperar la información.
- Hashing es una técnica que usa una función para mapear registros en direcciones de almacenamiento en memoria secundaria.
  Dada una clave, esta se convierte en un número, sobre el cual se aplica una función de hash que permite obtener la dirección de memoria



- NO se requiere almacenamiento adicional y NO es necesaria una estructura auxiliar (Utiliza el mismo archivo a diferencia de un índice por ejemplo). Facilita la inserción y eliminación rápida de registros en el archivo. Localiza registros dentro del archivo con un solo acceso a disco (no siempre pero casi).
- NO se puede aplicar hashing a registros de longitud variable
- NO es posible obtener un orden lógico de los datos
- NO es posible tratar con claves duplicadas, no sirve para claves secundarias

Tipos de dispersión
Con espacio de direccionamiento estático: El espacio para dispersar está definido previamente (espacio asignado para los registros de un archivo de datos).
Con espacio de direccionamiento dinámico: El espacio disponible para dispersar los registros de un archivo de datos aumenta o disminuye según las necesidades.

Parámetros de la dispersión (direccionamiento estático)

1. Función de hash

Recibe como entrada una clave, y produce una dirección de memoria secundaria donde almacenar el registro asociado a la clave en el archivo de datos.
El conjunto de direcciones físicas sobre el disco rígido es fijo y finito.
Si la función de hash es muy sencilla, tiende a generar colisiones (almacenar un registro en una dirección de memoria ocupada).
Claves sinónimas → Producen la misma dirección de memoria al calcular la función de hash

Solución para que no haya colisiones:

- Elegir función de hash perfecta, que no genere colisiones. Muy difícil, opción inválida
- Minimizar el número de colisiones y tratar estas como una condición excepcional. Existen varios modos de tratar colisiones:
    - **Distribuir los registros de la forma más aleatoria**.
    - **Utilizar más espacio en disco**. Ej: 100 direcciones disponibles para 10 registros. Poco eficiente en cuanto a espacio.
    - A**lmacenar más de un registro por cada dirección física**, así, 2 claves en situación de colisión podrían almacenarse en la misma dirección física asignada por la función de hash.
      Si a pesar de la mayor capacidad del nodo, un registro no cabe, se dice que ese nodo está saturado o en overflow. Esta situación debe tratarse.

2. Tamaño de cada nodo de almacenamiento

Agrandar la capacidad del nodo es una alternativa para evitar colisiones.
La capacidad del nodo queda determinada por la posibilidad de transferencia de información en cada operación de E/S desde RAM a Disco y viceversa.

3. Densidad de empaquetamiento (DE)

Es la relación entre el espacio disponible para el archivo (n) y la cantidad de registros que componen dicho archivo (r).
El espacio disponible se define como la cantidad de nodos direccionables (n) por la función de hash, y la cantidad de registros que cada nodo puede almacenar, **Registros por Nodo (RPN)**.

n = nodos direccionables
r = cantidad de registros del archivo
RPN = cantidad de registros por nodo

$$DE = \frac{r}{RPN * n}$$

Si se debieran esparcir 30 registros entre 10 direcciones con capacidad de cinco registros por cada dirección, la DE sería de 0.6 o 60% (30 / 5 * 10).

Cuanto mayor sea la DE, mayor será la probabilidad de colisiones.
Cuando la DE se mantiene baja, hay menos colisiones, pero se desperdicia espacio en disco (fragmentación). La DE sube a medida que se agregan registros.

Si se quiere aumentar el espacio de direcciones disponibles se deben reubicar todos los registros ya almacenados (con la función de hash ya aplicada) utilizando una nueva función de hash apropiada para la nueva cantidad de direcciones disponibles.

4. Métodos de tratamiento de desbordes (Overflow)

El **overflow** ocurre cuando un registro es direccionado a un nodo que no dispone de capacidad para almacenarlo. Cuando esto ocurre deben realizarse dos acciones:

1. Encontrar lugar para el registro en otra dirección
2. Asegurarse de que el registro posteriormente sea encontrado en esa nueva dirección.

Estudio de la ocurrencia de overflow

P(i); La probabilidad de que un nodo reciba i claves entre las K disponibles será:

$$P(i) = \frac{(K/N)^i * e^{-(K/N)}}{i!}$$

Siendo:

N representa el número de direcciones de nodos disponibles en memoria secundaria.
K determina la cantidad de registros a dispersar.
i determina la cantidad de registros que contendrá un nodo en un momento específico.
Ejemplo: El cálculo de la probabilidad de que un nodo no reciba ninguna clave (i = 0)

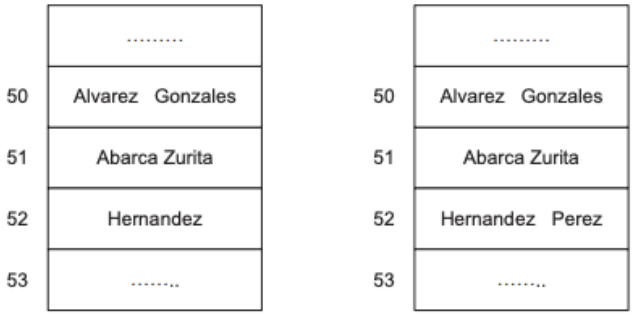
Resolución de colisiones con overflow

Aunque se cuente con una **DE** baja, es probable que las colisiones produzcan overflow, por ello se debe contar con un método para reubicar esos registros. Existen 4 métodos: saturación progresiva, saturación progresiva encadenada, doble dispersión y área de desbordes por separado.

Saturación progresiva

Consiste en almacenar el registro en la dirección siguiente más próxima (que haya espacio en el nodo) desde donde se produce el overflow. Ejemplo ingresando el registro “Perez”:

Se inserta un nuevo registro, Perez, y la función de hash retorna como dirección base al **nodo 50**. Esta situación genera una saturación, debido a que el nodo 50 se encuentra completo.
La primera dirección del nodo posterior que tiene capacidad para contener a **Perez** es la dirección **52**.

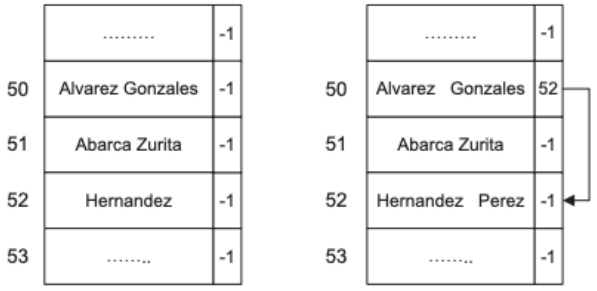


El proceso de búsqueda consiste en ir a la dirección dada por la función y, si el elemento a buscar no se encuentra allí, y el nodo está completo, se sigue buscando en los nodos siguientes, hasta encontrar el elemento, o hasta encontrar un nodo que no esté completo y no contenga el registro buscado.
IMPORTANTE: El método necesita indicar con una marca si una dirección estuvo completa anteriormente, para no impedir la búsqueda de otros registros. Cuando se elimina un valor, se deja una marca “**#**” en esa posición para que el nodo figure como completo y no frene otras búsquedas.

Es un método poco eficiente, **N** accesos o lecturas en el peor de los casos. El método podría requerir chequear todas las direcciones disponibles en un caso extremo, para poder localizar un registro.

Saturación progresiva encadenada
Igual que en la saturación progresiva, pero luego de localizar la nueva dirección disponible, esta se enlaza con la dirección base inicial, generando una cadena de búsqueda de elementos.
La búsqueda se hace a través de los enlaces, y la inserción también.
Tiene mejoras en la **performance** respecto a Saturación Progresiva, sin embargo, requiere que cada nodo manipule información extra (enlace). Ejemplo de inserción de “Perez”:

A partir del ejemplo anterior, es posible observar que con saturación progresiva fueron necesarios tres accesos para recuperar a Perez, en tanto que con saturación progresiva encadenada solamente se requirieron dos accesos.



Doble dispersión
El método consiste en disponer de 2 funciones de hash, la primera obtiene la dirección de memoria a partir de la clave, en el cual el registro es ubicado. **De producirse overflow,** se utilizará una segunda función de hash, esta segunda función no retorna una dirección, sino que retorna un desplazamiento. Este desplazamiento se suma a la dirección base obtenida con la primera función, generando así la nueva dirección donde se intentará ubicar al registro. En caso de **generarse nuevamente overflow**, se deberá sumar de manera reiterada el desplazamiento obtenido, y así sucesivamente hasta encontrar una dirección con espacio suficiente para albergar al registro.
Aumenta el tiempo de respuesta ya que los registros tienden a ubicarse lejos de sus direcciones y esto implica mayor desplazamiento de la cabeza lectora del disco

Área de desbordes por separado
Se distinguen dos tipos de nodos, los direccionables por la función de hash y los de reserva que serán alcanzados en caso de saturación (no son alcanzados por la función de hash).
Al producirse saturación, el registro es reubicado en la primera dirección disponible del área de desbordes, y la dirección base original se enlaza con la dirección del área de reserva utilizada.
Si no hay lugar disponible en el nodo del área de desbordes, se redirecciona el elemento a un nuevo nodo (dentro del área de desbordes), y se enlaza con el primero.

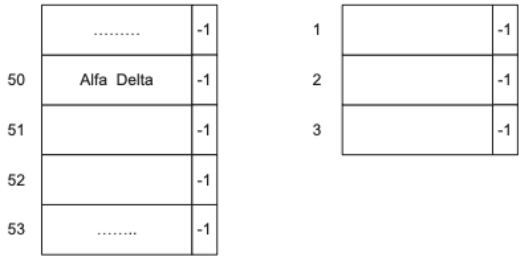


Figura 8.6 a)

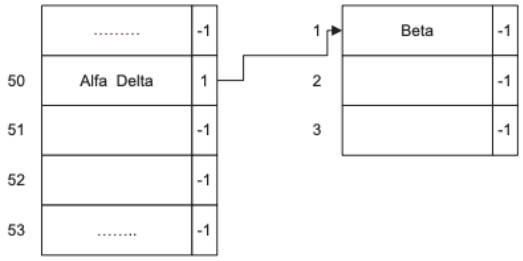


Figura 8.6 b)

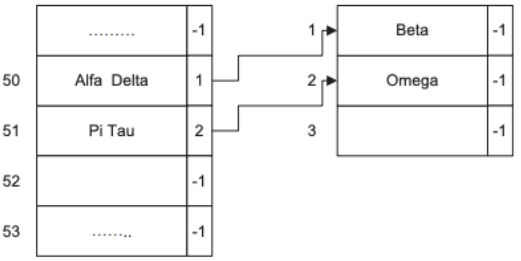


Figura 8.6 c)

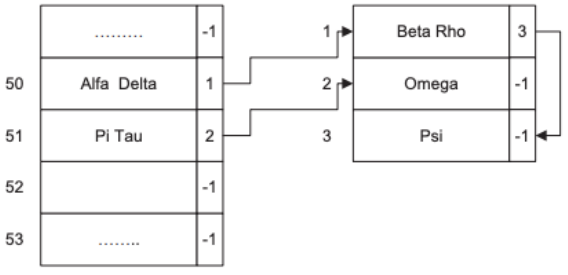


Figura 8.6 d)

Hash asistido por tabla
Asegura acceder a un registro de datos en 1 solo acceso, pero necesita una estructura adicional.
Utiliza 3 funciones (hash, desplazamiento y bits):

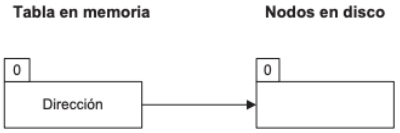
1. Dirección física
2. Desplazamiento
3. Secuencia de bits que no pueden ser todos unos

Necesita utilizar espacio extra en memoria principal para la administración de la tabla.
Complejidad adicional en caso de saturación.
Este método coloca la búsqueda por sobre las otras operaciones.

Hash con espacio de Direccionamiento Dinámico
Cuando se analizó la probabilidad de saturación, quedó establecido un umbral (de 75%) deseado para la DE; es decir, mientras la DE se encuentre en un valor inferior a ese tope, la probabilidad de overflow disminuirá considerablemente y tenderá a cero.
Si el espacio de direcciones de memoria se completa (o casi), es necesario obtener mayor cantidad de direcciones para nodos, y también es necesario que la función de hash direcciones más nodos. (Significa redispersar todos los registros del archivo :(Tarea difícil y costosa).
El hashing con espacio de direccionamiento dinámico es una alternativa para que el espacio de direcciones crezca de manera dinámica.

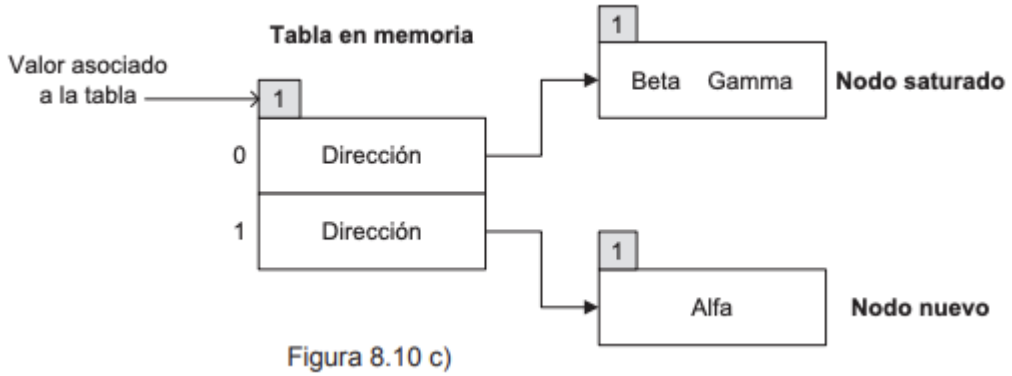
Hash extensible
El principio del método consiste en comenzar a trabajar con un único nodo para almacenar registros e ir aumentando la cantidad de direcciones disponibles a medida que los nodos se completan.
No utiliza el concepto de DE. Esto se debe a que el espacio en disco utilizado aumenta o disminuye en función de la cantidad de registros de que dispone el archivo en cada momento.
La función de hash retorna un string de bits, la cantidad de bits que retorna determina la cantidad máxima de direcciones a las que puede acceder el método.
Este método **requiere de una estructura auxiliar**, una tabla que se administra en memoria principal y contiene la dirección física de cada nodo.

Comienza con un solo nodo en el disco y una tabla que solamente contiene una dirección, la del único nodo disponible. La tabla indica la cantidad de bits a direccionar:



Supongamos un problema donde la capacidad del nodo es de 2 registros. Se debe dispersar la clave “Alfa”. Se **calcula la función de hash** y se **toman tantos bits como indica el valor asociado a la tabla**. En este caso, dicho valor es 0; esto indica que hay una sola dirección del nodo disponible y Alfa debería insertarse en dicho nodo, sin tener en cuenta el bit menos significativo de la función aplicada a “Alfa”. De igual manera trabaja al insertar “Beta”, ya que no se genera saturación (overflow).

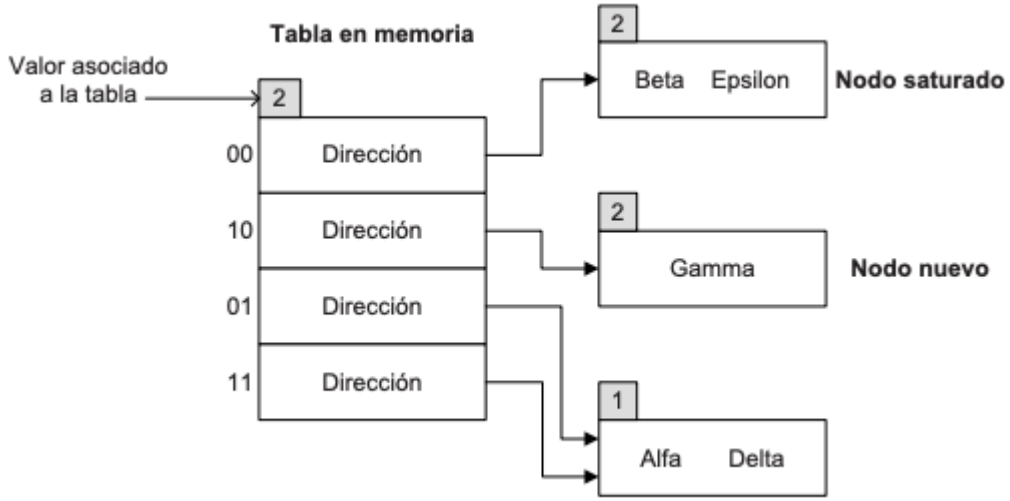
Se debe dispersar la clave “Gamma”. En este caso se genera un **overflow**. Entonces se debe incrementar en 1 el valor asociado al nodo saturado; luego se chequea si el valor asociado a la tabla es menor (En este caso es 0). Al ser menor, se deben generar más direcciones y redireccionar.
El valor asociado a la tabla indica la cantidad de bits que es necesario tomar de la función de hash. A partir de este momento es necesario tomar uno de ellos, el menos significativo.
La primera celda de la tabla direcciona al nodo saturado, y la nueva celda apunta al nuevo nodo generado.
Los elementos de la cubeta saturada (Alfa y Beta) más el elemento que generó la saturación (Gamma) son **redispersados** entre ambos nodos de acuerdo con el valor que determina el bit menos significativo, resultante de la función de hash.



Clave	FH(Clave)
Alfa	00.....1001
Beta	00.....0100
Gamma	00.....0010
Delta	00.....1111
Epsilon	00.....0000

Figura 8.10 c)

Si se ingresa “Delta” y “Epsilon”:



Características:

- Se utilizan sólo los bits necesarios (FH).
- Los bits tomados forman la dirección del nodo que se utilizará
- Al haber saturación, se genera un nuevo nodo y se redirecciona con 1 bit más
- La tabla tendrá tantas direcciones de nodos como 2^i (siendo i el número de bits actuales para el sistema)
- La búsqueda se realiza en 1 solo acceso

Conclusiones

- **Búsqueda secuencial** → baja performance, proceso simple
- Los **árboles balanceados** son aceptables en términos de eficiencia
    - Árbol B y B* → mismo comportamiento pero el B* completa mas los nodos (tiene menos accesos, pero las operaciones de inserción son más lentas)
    - Arbol B+ → búsqueda eficiente al igual que B y recorrido secuencial eficiente.
- **Hashing estático** CASI siempre encuentra un elemento en 1 acceso a disco.
- **Hashing extensible** utiliza espacio de direccionamiento dinámico y SIEMPRE recupera (encuentra) la información en 1 acceso a disco. El costo de esta técnica es su complejo procesamiento cuando una inserción genera overflow.