Aspectos físicos de BD

SECCIÓN II

CAPÍTULO 2

Archivos, estructuras y operaciones básicas

Objetivo

El objetivo de este capítulo es presentar conceptos básicos de archivos, comenzando desde su definición, analizando sus propiedades esenciales, así como las operaciones elementales que permiten desarrollar la algorítmica más simple sobre ellos.

Si bien la orientación de los algoritmos propuestos es procedimental y cercana al lenguaje Pascal, se presentan los programas y procedimientos en un seudocódigo que permite al lector abordar los temas sin un conocimiento acabado de ese lenguaje de programación.

Se discuten sobre el final del capítulo, además, las alternativas para borrar información en el archivo, las cuales serán desarrolladas en detalle en el Capítulo 4.

Archivos

Los datos que necesitan ser preservados más allá de la ejecución de un algoritmo deben residir en archivos. Para que esto suceda, un archivo no puede residir en la memoria RAM, sino que debe estar en dispositivos de almacenamiento permanente de información. El ejemplo más natural como medio permanente para contener información lo constituye el disco rígido.

Definiciones

Se presentan a continuación algunas definiciones de archivo de datos.

Un **archivo** es una colección de registros semejantes, guardados en dispositivos de almacenamiento secundario de la computadora (Wiederhold, 1983).

Un **archivo** es una estructura de datos que recopila, en un dispositivo de almacenamiento secundario de una computadora, una colección de elementos del mismo tipo (De Giusti, 2001).

Un **archivo** es una colección de registros que abarcan entidades con un aspecto común y originadas para algún propósito particular.

Estas definiciones muestran que un archivo es una estructura de datos homogénea. Los archivos se caracterizan por el crecimiento y las modificaciones que se efectúan sobre éstos. El crecimiento indica la incorporación de nuevos elementos, y las modificaciones involucran alterar datos contenidos en el archivo, o quitarlos.

Aspectos físicos

Las estructuras de datos, en general, se definen y utilizan sobre la memoria RAM de la computadora y tienen existencia mientras el programa se encuentra activo. Por lo tanto, cuando es necesario tener persistencia en la información que maneja un algoritmo, esta información debe ser almacenada en archivos.

Los archivos residen en dispositivos de memoria secundaria o almacenamiento secundario, los cuales son medios que están ubicados fuera de la memoria RAM, y que son capaces de retener información luego que el programa finaliza o luego de apagar la computadora. Algunos ejemplos de dispositivos de almacenamiento secundario, entre otros, son los discos rígidos, los discos flexibles (*diskettes*), los discos ópticos, las cintas.

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).

No es propósito de este libro profundizar sobre los aspectos de *hardware* de una computadora. Queda para el lector obtener, en libros dedicados a esos temas, mayores detalles acerca de las diferencias entre las

memorias principales (RAM) y secundarias (discos magnéticos u ópticos, o memorias Flash). Solamente se enfatiza la diferencia sustancial existente en los tiempos de acceso a estos dispositivos. Los análisis en capítulos posteriores están orientados a crear estructuras de archivos que puedan brindar tiempos de respuesta razonables, teniendo en cuenta las diferencias de velocidades entre las memorias RAM y secundarias.

Administración de archivos

Se deben considerar dos aspectos fundamentales cuando se trabaja con archivos:

- Visión física
- Visión lógica

Los archivos residen en memoria secundaria. En un disco rígido pueden coexistir un gran número de archivos. La visión física de un archivo tiene que ver con el almacenamiento de este en memoria secundaria. Es responsabilidad del sistema operativo de una computadora resolver cuestiones relativas al lugar de almacenamiento de archivo en disco, cómo se ubicará la información en el mismo y cómo será recuperada. Nuevamente, para este libro no son aspectos a profundizar y no serán detallados.

Los archivos son manejados por algoritmos. Desde el punto de vista práctico, un algoritmo no referencia al lugar físico donde el archivo reside, sino que tiene una visión lógica de este; es decir que lo utiliza como si estuviera almacenado en la memoria RAM. Desde un algoritmo se debe establecer una "conexión" con el sistema operativo, y este será el responsable de determinar el lugar de residencia del archivo en disco y su completa administración.

A partir de las visiones anteriores, se deben distinguir dos conceptos diferentes de archivo pero interrelacionados:

- **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.

Antes que el programa pueda operar sobre el archivo, el sistema operativo debe recibir instrucciones para hacer un enlace entre el nombre lógico que utilizará el algoritmo y el archivo físico. Cada lenguaje de programación define una instrucción para tal fin.

Tipos de archivos

Los archivos de datos pueden analizarse desde diversas perspectivas. Es posible entonces definir el tipo de archivo por la información que contiene, por la estructura de información que contiene o por la organización física del mismo en el dispositivo de memoria secundaria.

Desde la perspectiva de la información que contiene, un archivo puede contener un tipo de dato simple: entero, real o carácter. Además, puede contener una estructura heterogénea como lo son los registros. De acuerdo con las necesidades de cada algoritmo, debe definirse el tipo de dato que el archivo necesite.

A partir de lo definido anteriormente, se infiere que un archivo contiene elementos del mismo tipo. Esto lo convierte en una estructura homogénea de datos. Los registros, en general, poseen longitud fija, determinada por la suma de las longitudes de los campos que los componen. Cuando un archivo se genera a partir de registros con este formato, se denomina archivo de longitud fija y predecible de los elementos de datos contenidos. El resto del capítulo tratará acerca de archivos con estas características. Además, es posible contar con registros con longitudes variables, que se adapten a las necesidades propias de cada dato. En estos casos, los archivos generados tendrán una lógica de trabajo diferente, la cual será discutida en el Capítulo 4.

Acceso a información contenida en los archivos

Básicamente, se pueden definir tres formas de acceder a los datos de un archivo:

- **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.

Suponga el lector que se dispone de un archivo con los nombres de los empleados de una empresa. Los datos de dicho archivo están almacenados en el orden en el cual fueron ingresados; sin embargo, a partir de un acceso secuencial indizado, es posible recuperar los datos en forma ordenada, es decir, como si hubiesen sido ingresados alfabéticamente.

- **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.

  Estos modos de acceso resultan de fundamental importancia para esta Sección Si bien en este capítulo se tratarán archivos con acceso secuencial para comprender su operatoria básica, en el resto de la Sección se presentarán las desventajas de este acceso, su bajo nivel de *performance* y la necesidad de contar con mejores organizaciones para la administración de archivos que representen en forma más adecuada a las bases de datos.

Operaciones básicas sobre archivos

Para poder operar con archivos, son necesarias una serie de operaciones elementales disponibles en todos los lenguajes de programación que utilicen archivos de datos. Estas operaciones incluyen:

- La definición del archivo lógico que utilizará el algoritmo y la relación del nombre lógico del archivo con su almacenamiento en el disco rígido.
- La definición de la forma de trabajo del archivo, que puede ser la creación inicial de un archivo o la utilización de uno ya existente.
- La administración de datos (lectura y escritura de información).

  Estas operaciones elementales se traducen en una serie de instrucciones en la que cada lenguaje define la operatoria detallada sobre archivos.

  El lector debe tener en cuenta que en los ejemplos del libro se utilizará una notación seudocódigo Pascal.

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.

Otra opción para definir archivos se presenta a continuación:

```
Type archivo= file of tipo_de_dato;
```

```
Var archivo_logico: archivo;
```

en este caso, se define un tipo `archivo` y, luego, una variable `archivo_logico` del tipo definido anteriormente.

El Ejemplo 2.1 muestra la definición de algunos archivos para almacenar tanto tipos de datos simples como compuestos.

EJEMPLO 2.1

```
Type
     Persona = record
          DNI                 :string[8];
          ApellidoyNombre     :string [30];
          Direccion           :string [40];
          Sexo                :char;
          Salario             :real;
     end;
     ArchivodeEnteros     = file of integer;
     ArchivodeReales      = file of real;
     ArchivodeCaracteres  = file of char;
     ArchivodeLinea       = file of string;
     ArchivodePersonas    = file of Persona;

Var
     Enteros      : ArchivodeEnteros;
     Reales       : ArchivodeReales;
     Caracteres   : ArchivodeCaracteres;
     Texto        : ArchivodeLinea;
     Personas     : ArchivodePersonas
```


Correspondencia archivo lógico-archivo físico

Como se indicó anteriormente, el sistema operativo de la computadora es el responsable de la administración del archivo en disco. El algoritmo utiliza un tipo de datos `file` también definido anteriormente, que representa el nombre lógico del mismo. 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.

El Ejemplo 2.2 presenta, a partir de la definición del Ejemplo 2.1, las asignaciones correspondientes. Observaciones a partir del ejemplo:

- En algunos casos, coinciden los nombres lógicos con la variable de tipo `string` definida para el nombre físico, pero esto no representa ningún inconveniente en la definición.
- Algunas definiciones de nombre físico contienen la extensión que se quiere dar al archivo.
- El archivo `archivodelinea` definido como `file of string` contendrá elementos de 255 caracteres de longitud en cada caso.

EJEMPLO 2.2

```
Type
     Persona = record
          DNI                  :string[8]
          ApellidoyNombre      :string [30];
          Direccion            :string [40];
          Sexo                 :char;
          Salario              :real;
     end;
     ArchivodeEnteros    = file of integer;
     ArchivodeReales     = file of real;
     ArchivodeCaracteres = file of char;
     ArchivodeLinea      = file of string;
     ArchivodePersonas   = file of Persona;

Var
     Enteros       : ArchivodeEnteros;
     Reales        : ArchivodeReales;
     Caracteres    : ArchivodeCaracteres;
     Texto         : ArchivodeLinea;
     Personas      : ArchivodePersonas

Begin
     {comienzo del programa}
     Assign( Enteros,'c:\archivos\enteros.dat');
     Assign( Reales,'c:\archivos\numerosreales');
     Assign( Caracteres,'c:\archivos\letras.dat');
     Assign( Texto, 'c:\archivos\linea');
     Assign( Personas, 'c:\archivos\pers.dat');
     ....

End.
```


Apertura y creación de archivos

Hasta el momento, se ha detallado cómo definir un archivo y se ha establecido la relación con el nombre físico. Para operar con un archivo desde un algoritmo, se debe realizar la apertura.

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.
- La operación `reset` indica que el archivo ya existe y, por lo tanto, las operaciones válidas sobre el mismo son lectura/escritura de información.

Estas operaciones definen entonces la modalidad de trabajo que se podrá realizar sobre el algoritmo. El lector debe notar la importancia y utilización de cada una de ellas. Si se intentara abrir como lectura/escritura un archivo inexistente, la operación `reset` generaría un error. Si se realizara la apertura de un archivo con la operación `rewrite`, y el archivo existiera previamente en disco, se borraría toda la información en él contenida, comenzando con un archivo vacío.

El formato de las dos operaciones es el siguiente:

```
rewrite(nombre_logico)

reset(nombre_logico)
```

donde `nombre_logico` es la variable definida previamente.

Cierre de archivos

Una vez finalizada la operatoria sobre un archivo, el mismo debe ser cerrado. Cerrar un archivo significa una serie de eventos:

- Cada archivo tiene un comienzo y 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.

  La operación de cierre de archivo es:

  ```
  close(nombre_logico)
  ```

  donde `nombre_logico` corresponde a la variable de tipo `file` del archivo que se desea cerrar.

Lectura y escritura de archivos

Para leer o escribir información en un archivo, las instrucciones son:

```
read(nombre_logico, var_dato);
```

```
write(nombre_logico, var_dato);
```

en ambos casos, `nombre_logico` corresponde al nombre lógico del archivo desde donde se desea leer o escribir. En tanto, `var_dato` corresponde a la variable del algoritmo que contendrá la información a transferir al archivo o la variable que recibirá la información desde el archivo, según el caso.

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.

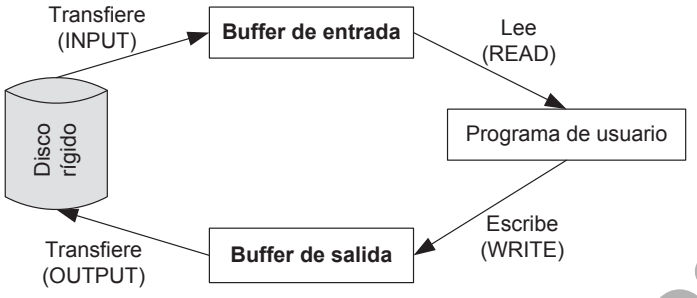
Buffers de memoria

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 justificación de lo anterior se debe a cuestiones de *performance*. Mientras que una operación sobre la memoria RAM tiene una demora del orden de nanosegundos, una operación sobre disco tiene una demora del orden de milisegundos. Por lo tanto, varias operaciones de lectura y/o escritura directamente sobre la memoria secundaria reducirían notablemente la *performance* de un algoritmo. Por este motivo, el sistema operativo, al controlar dichas operaciones, intenta minimizar el tiempo requerido para estas.

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`. Cada `read` se mide en nanosegundos, en tanto que los `input` se miden en milisegundos, pero se realizan con menor frecuencia.

Figura 2.1



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)

Esta forma de trabajo optimiza la *performance* de lectura y escritura sobre archivos.

Creación de archivos

Se presentan a continuación algunos ejemplos de algoritmos que crean archivos de datos, cuya información se lee desde teclado.

El Ejemplo 2.3 crea un archivo de números reales, bajo las siguientes precondiciones:

- Los números se obtienen desde teclado hasta recibir el número cero, el cual no se incorpora al archivo.
- El nombre del archivo es determinado desde el mismo algoritmo.

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 );
```

continúa >>>

```
    {lectura de un número real}
    read( NroReal );

    while (NroReal <> 0)do
      begin
         {escritura del número en el archivo}
          write( Reales, NroReal );

          {lectura de otro número real}
          read( nro );
      end;

    cierre del archivo}
    close( Reales );

end.
```

El Ejemplo 2.4 crea un archivo con datos de personas, bajo las siguientes precondiciones:

- Los datos se leen de teclado y finalizan cuando se lee un DNI nulo o vacío.
- El nombre del archivo se obtiene desde teclado.

En este algoritmo, el lector debe notar la diferencia entre leer información desde el teclado (lo cual debe resolverse campo a campo) y la escritura en el archivo, donde se indica solamente el registro.

EJEMPLO 2.4

```
Programa CrearArchivo;
Type
     Persona = record
          DNI                 :string[8]
          ApellidoyNombre     :string [30];
          Direccion           :string [40];
          Sexo                :char;
          Salario             :real;
     end;
     ArchivodePersonas   = file of Persona;

Var
     Personas        : ArchivodePersonas;
     Nombrefisico    : string[12];
     Per             : Persona;

Begin
    write( 'Ingrese el nombre del archivo:' );
    readln(NombreFisico)
```

continúa >>>

```
    {enlace entre el nombre lógico y el nombre físico}
    Assign( Personas, NombreFisico);

    {apertura del archivo para creación}
    rewrite( Personas );

    {lectura del DNI de una persona}
    readln( per.DNI );

    while (per.DNI <> '')do
      begin
          {lectura del resto de los datos de la persona}
          readln(per.ApellidoyNombre);
          readln(per.Direccion);
          readln(per.Sexo);
          readln(per.Salario);

          {escritura del registro de la persona en el archivo}
          write( Personas, per );

          {lectura del DNI de una nueva persona}
          readln( per.DNI );
      end;

    cierre del archivo}
    close( Personas );

end.
```

En ambos casos, la apertura del archivo se realiza mediante la instrucción `rewrite` dado que el archivo aún no existe, se está creando.

Operaciones adicionales con archivos

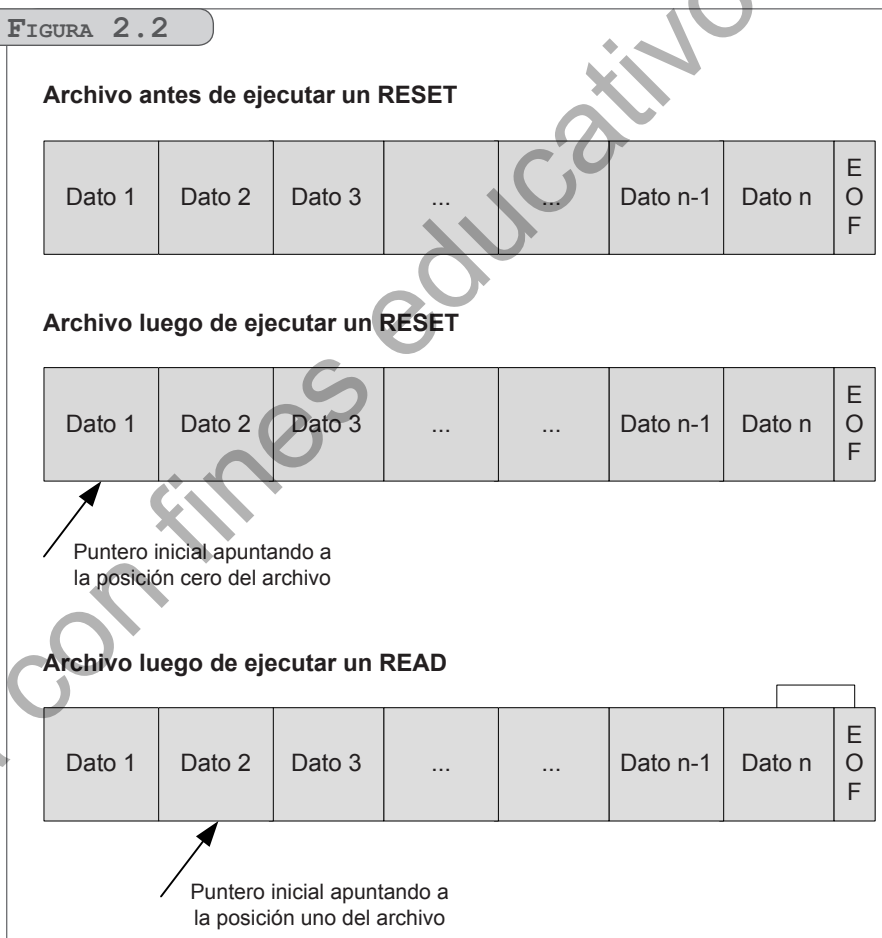
Hasta el momento, fueron presentadas las operaciones básicas sobre archivos de datos. Son necesarias, además, una serie de operaciones adicionales que permitirán operar con el archivo una vez creado. Estas operaciones son:

- Control de fin de datos.
- Control de tamaño del archivo.
- Control de posición de trabajo dentro del archivo.
- Ubicación física en alguna posición del archivo.

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. Este puntero, relacionado al nombre lógico del archivo, indicará en todo momento la posición actual de trabajo. Esta posición actual direcciona al elemento que retornará una instrucción `read` o el lugar donde se escribirá un dato con la instrucción `write`. Tanto la instrucción `read` como la `write` avanzan en forma automática el puntero una posición luego de ejecutarse. (Figura 2.2)

Figura 2.2



Control de fin de datos

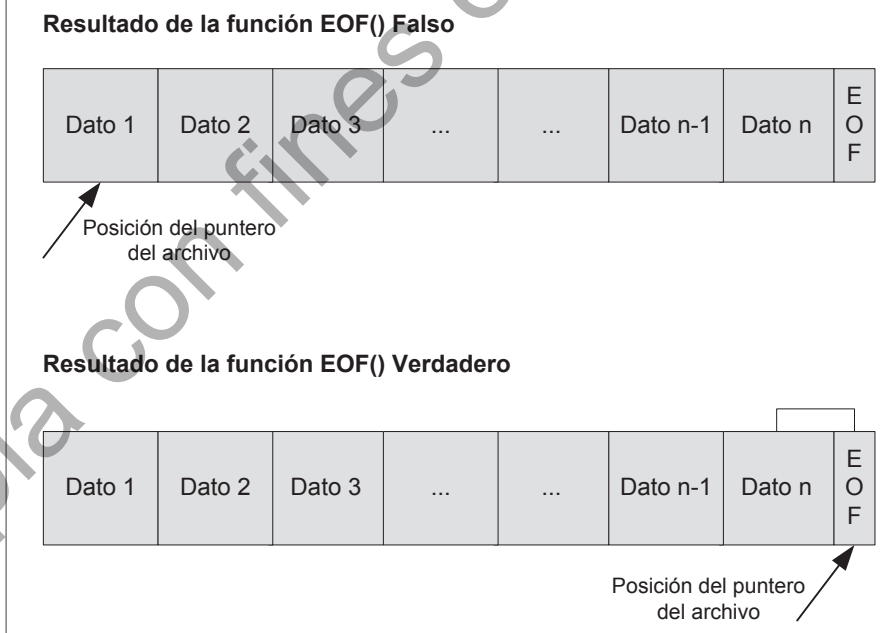
Para poder recorrer un archivo de datos existente, se debe realizar la lectura de cada uno de sus elementos. Como se dijo anteriormente, el tipo de acceso que tendrán los archivos en los Capítulos 2 y 3 corresponde al acceso secuencial. De esta forma, los datos son obtenidos en el orden en que fueron ingresados, desde el comienzo y hasta el final. Es necesario, entonces, contar con una operación que permita controlar la recuperación de datos sin exceder el fin de archivo.

Un algoritmo debe controlar el fin de archivo antes de realizar una operación de lectura. La función

```
eof(nombre_logico);
```

es la encargada de resolver este control. Aquí, `nombre_logico` corresponde al archivo que se está evaluando. La función retornará verdadero si el puntero del archivo referencia a EOF, y falso en caso contrario. (Figura 2.3.)

Figura 2.3



Note el lector que si un algoritmo realiza una operación de lectura y el puntero del archivo se encuentra apuntando a EOF, se genera un error en la ejecución del programa.

Control de tamaño del archivo

En determinados algoritmos, es necesario conocer la cantidad de elementos que conforman un archivo. Se dispone para tal fin de una función que retorna un valor entero indicando cuántos elementos se encuentran almacenados en un archivo de datos.

```
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

Con cada operación de lectura y/o escritura, avanza automáticamente el puntero lógico del archivo. Este movimiento del puntero debe ser, en algunas circunstancias, manejado por el algoritmo. 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.

Note el lector que `posición` debe ser un valor comprendido entre cero y `filesize(nombre_logico)`. En su defecto, la instrucción generará un error de ejecución.

Lectura de archivos

El algoritmo siguiente al de creación de un archivo debe permitir leer los elementos contenidos en el mismo. Así, el Ejemplo 2.5 presenta el código correspondiente a leer el archivo generado en el Ejemplo 2.4.

EJEMPLO 2.5

```
Procedure Recorrido(var Personas: ArchivodePersonas );

var Per: Persona; {para leer elementos del archivo}

begin

     {dado que el archivo está creado, debe abrirse como de
     lectura/escritura}
     reset(Personas);

     while not eof(Personas)do
       begin

          {se obtiene un elemento desde archivo}
          read( Personas, Per );

          {se presenta por pantalla cada dato del elemento leído}
          writeln( Per.DNI );
          writeln( Per.ApellidoyNombre );
          writeln( Per.Direccion );
          writeln( Per.Sexo );
          writeln( Per.Salario );
          writeln;
       end;

     {cierre del archivo}
     close( Personas );

end;
```


A partir del Ejemplo 2.5:

- El lector puede observar que el ejemplo se presenta como un módulo, en lugar de un programa. En este módulo es importante notar la forma en que se pasa el parámetro correspondiente al nombre lógico del archivo. Si bien el algoritmo no modifica el archivo, y por lo tanto el parámetro podría ser pasado por valor, el mismo se envía por referencia. Se debe notar, entonces, que el parámetro correspondiente al nombre lógico de un archivo no se maneja de forma similar que el resto de los parámetros. El nombre lógico del archivo siempre representa un nexo con el nombre físico y, por ende, es necesario pasarlo siempre a un módulo por referencia.
- Además, no se realiza la asignación nombre lógico-nombre físico dentro del módulo. Esta debe, en general, resolverse en el cuerpo del programa principal.
- El archivo debe abrirse como lectura. La instrucción prevista para tal fin es `reset`.
- La lectura del archivo se realiza registro a registro, pero la impresión en pantalla debe resolverse campo a campo.

Agregar más información a un archivo

Otro tipo de algorítmica básica sobre archivos consiste en incorporar nueva información a los mismos. Siguiendo el Ejemplo 2.4, sobre el archivo de personas se incorporan nuevos datos; el Ejemplo 2.6 muestra esta acción.

EJEMPLO 2.6

```
Procedure Agregar (var Personas: ArchivodePersonas );

     Procedure Leer( var Per:Persona);
     Begin
          readln( Per.DNI );
          readln( Per.ApellidoyNombre );
          readln( Per.Direccion );
          readln( Per.Sexo );
          readln( Per.Salario );
     End,

var  Per: Persona; {para leer elementos del archivo}
begin
     {dado que el archivo está creado, debe abrirse como de
     lectura/escritura}
     reset(Personas);
     {posicionamiento al final del archivo}
     seek( Personas, filesize(Personas));
     {lectura de datos de una persona}
     Leer(Per)
     while (Per.DNI<>'')do
        begin
           {se agrega una persona al archivo}
            write(Personas, Per)
           {lectura de datos de otra persona}
           Leer(Per)
     end;
   {cierre del archivo}
   close( Personas );
end;
```


Las conclusiones, en este caso, son:

- Nuevamente, el problema se resuelve con un módulo, donde se respeta la forma de pasaje de parámetros anteriormente descripta.
- El archivo debe abrirse nuevamente con la instrucción `reset`, esto dado a que debe incorporar nuevos datos manteniendo los anteriores. Si se hubiese realizado una apertura con la instrucción `rewrite`, entonces los datos ya almacenados en el archivo desaparecerían.

- La instrucción `filesize(nombre_logico)` determina cuántos registros tiene el archivo. El lector debe recordar que, en general, los archivos numeran a sus registros a partir del registro cero. Por lo tanto, la instrucción `seek(nombre_logico, filesize(nombre_logico))` dejará el puntero lógico del archivo referenciando a EOF, y las escrituras posteriores quitarán la marca.
- La orden `close(nombre_logico)` incorpora nuevamente la marca sobre el nuevo final del archivo.

Modificar la información de un archivo

El último ejemplo, correspondiente a la denominada algorítmica básica sobre archivos, corresponde a su vez a aquel problema donde se plantea la necesidad de modificar los datos contenidos en un archivo. Manteniendo la estructura del problema inicial, el Ejemplo 2.7 presenta un módulo que permite modificar el salario de las personas, realizando a este un incremento de 15%.

EJEMPLO 2.7

```
Procedure Actualizar(var Personas: ArchivodePersonas );
var Per: Persona; {para leer elementos del archivo}
begin
    {dado que el archivo está creado, debe abrirse como de
    lectura/escritura}
    reset(Personas);
    while not eof (Personas)do
      begin
        Read( Personas, Per);
        {modificación del salario}
        Per.salario := Per.salario * 1.15;
        {ubicar al puntero del archivo en el registro leído}
        Seek( Personas,filepos(Personas) -1 );
        {se graba la persona con salario modificado}
        write(Personas, Per)
      end;
   {cierre del archivo}
   close( Personas );
end;
```


Las conclusiones, en este caso, son:

- Luego de efectuar una operación de lectura sobre el archivo, el puntero lógico se desplaza una posición hacia adelante. Por lo tanto, es necesario retroceder una posición antes de proceder a la escritura del registro modificado.

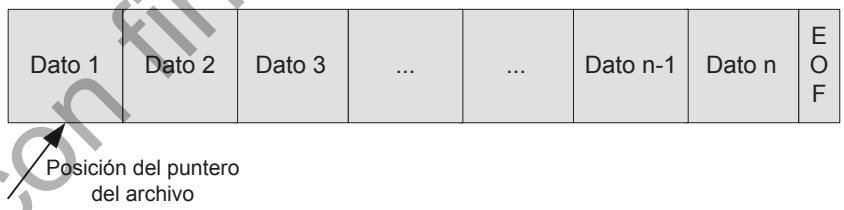
- Luego, la operación de escritura lleva al puntero nuevamente sobre el siguiente registro. La Figura 2.4 presenta gráficamente el caso anterior.

Figura 2.4

Puntero lógico luego de la apertura del archivo (RESET)

Puntero lógico luego de la lectura del primer dato del archivo

Puntero lógico del archivo de ejecutar seek(nombre_logico, filepos(nombre_logico)-1);



Se escribe el dato modificado {write(nombre_logico, variable)}

El puntero lógico avanza nuevamente

Puntero lógico del archivo de ejecutar

```
seek(nombre_logico, filepos(nombre_logico)
-1);
```

Se escribe el dato modificado

```
{write(nombre_logico, variable) }
```

Proceso de bajas

Las operaciones esenciales sobre archivos son:

- **Alta:** ingresar nuevos datos al archivo.
- **Modificación:** alterar el contenido de algún dato del archivo.
- **Consulta:** presentar el contenido total o parcial del archivo.
- **Baja:** quitar información del archivo.

Hasta el momento, se describieron, en líneas generales, las tres primeras operaciones. El proceso de baja debe ser discutido con mucho mayor nivel de detalle. Esto se debe a que, en general, cuando se opera con bases de datos, las circunstancias donde se desea quitar información de un archivo están muy limitadas. El proceso de baja será discutido en detalle en el Capítulo 4, así como las circunstancias que lo hacen poco frecuente.

Cuestionario del capítulo

1. ¿Qué es un archivo? ¿En qué se diferencia de otras estructuras de datos conocidas por el lector?
2. ¿Cuáles son las principales características a tener en cuenta cuando se opera con un archivo? ¿Qué es el *buffering*?
3. ¿Qué tipos de accesos reconoce el lector sobre un archivo?
4. ¿En qué se diferencia el nombre lógico de un archivo del nombre físico?
5. ¿Por qué existen dos formas de abrir un archivo, `reset` y `rewrite`?
6. ¿Por qué es importante que un algoritmo contenga la instrucción `close`?
7. ¿Cómo trabaja el puntero lógico del archivo?

Ejercitación

1. Siguiendo la lógica del Ejercicio 2.4, el lector deberá realizar un algoritmo que permita crear un archivo que contenga números enteros.
2. Realice un algoritmo que permita guardar en un archivo todos los productos que actualmente se encuentran en venta en un determinado negocio. La información que es importante considerar es: nombre del producto, cantidad actual, precio unitario de venta, tipo de producto (que puede ser comestible, de limpieza o vestimenta).
3. Teniendo como base al ejercicio anterior, plantee un algoritmo que permita presentar en pantalla los productos del archivo que correspondan al rubro de limpieza o cuya cantidad actual supere las 100 unidades.
4. Realice un algoritmo que permita modificar los precios unitarios de los productos del archivo generado en el Ejercicio 2, incrementando en 10% los correspondientes al rubro comestibles, en 5% a los de limpieza y en 20% a los de vestimenta.

Capítulo 3

Algorítmica clásica sobre archivos

Objetivo

El objetivo del Capítulo 3 está relacionado con el desarrollo de algoritmos considerados clásicos en la operatoria de archivos secuenciales. Estos algoritmos se resumen en tres tipos: de actualización, *merge* y corte de control.

El primer caso, con todas sus variantes, permite introducir al alumno en problemas donde se actualiza el contenido de un archivo resumen o "maestro", a partir de un conjunto de archivos con datos vinculados a ese archivo maestro.

En el segundo caso, se dispone de información distribuida en varios archivos que se reúne para generar un nuevo archivo, producto de la unión de los anteriores.

Por último, el corte de control, muy presente en la operatoria de BD, determina situaciones donde, a partir de información contenida en archivos, es necesario generar reportes que resuman el contenido, con un formato especial.

Proceso de actualización de archivos

Si bien el Capítulo 2 presentó algoritmos denominados de actualización, esto se realizó bajo la consigna de cambiar algún dato de un archivo. Esto es, y siguiendo con el Ejemplo 2.7, modificar el salario de las personas contenidas en el archivo. En este apartado, el contenido de un archivo será modificado por el contenido de otro archivo.

Se deben introducir, en este punto, los conceptos de archivo maestro y archivo detalle, que se continuarán utilizando en el resto del capítulo. Se denomina archivo maestro al 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.

Por otra parte, se denomina archivo detalle al 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.

En el resto del apartado, se plantean diversas situaciones que ejemplifican el proceso de actualización. Es muy importante que el lector analice las precondiciones que en cada caso se plantean. Los algoritmos propuestos tienen en cuenta tales precondiciones que, en caso de no ser cumplidas, determinan la falla de su ejecución.

Actualización de un archivo maestro con un archivo detalle (I)

Presenta la variante más simple del proceso de actualización. Las precondiciones 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 y solo un elemento 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. Más adelante, se discutirán situaciones donde los archivos respetan un orden lógico de datos.

El Ejemplo 3.1 presenta un caso práctico donde se tienen en cuenta las precondiciones anteriores.

EJEMPLO 3.1

```
program ejemplo_3_1;
  type
    producto = record
      cod: str4;
      descripcion: string[30];
      pu: real;
      stock: integer;
    end;
    venta_prod = record
      cod: str4;
      cant_vendida: integer;
    end;
    detalle = file of venta_prod;
    maestro = file of producto;
  var
    regm: producto;
    regd: venta_prod;
    mae1: maestro;
    det1: detalle;
begin
  assign (mae1, 'maestro');
  assign (det1, 'detalle');
  reset (mae1);
  reset (det1);
  while (not eof(det1)) do begin
    read(mae1,regm);
    read(det1,regd);
    {se busca en el maestro el producto del detalle}
    while (regm.cod <> regd.cod) do
      read (mae1,regm);
   {se modifica el stock del producto con la cantidad
    vendida de ese producto}
    regm.stock := regm.stock - regd.cant_vendida;

    {se reubica el puntero en el maestro}
    seek (mae1, filepos(mae1)-1);

    {se actualiza el maestro}
    write(mae1,regm);
  end;
  close(det1);
  close(mae1);
end.
```


Algunas consideraciones del ejemplo anterior son las siguientes:

- 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.

- Es necesario buscar aquel registro del archivo maestro que se actualiza a partir del archivo detalle. Esto se debe a que no todos los elementos del archivo maestro necesariamente serán modificados.
- Nótese que las lecturas sobre el archivo maestro se realizan sin controlar el fin de archivo. Si bien esta operatoria se indicó como incorrecta en el Capítulo 2, el lector debe notar que, a partir de la tercera precondición planteada, el archivo detalle no genera altas. Por lo tanto, para cada registro del archivo detalle debe necesariamente existir el correspondiente en el archivo maestro.

Actualización de un archivo maestro con un archivo detalle (II)

Este es otro caso, levemente diferente del anterior; solo se modifica una precondición del problema y hace, de esta forma, variar el algoritmo resolutorio. Las precondiciones, en este caso, 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 el maestro.
- Cada elemento del archivo maestro puede no ser modificado, o ser modificado por uno o más elementos del detalle.
- Ambos archivos están ordenados por igual criterio. Esta precondición, considerada muy fuerte, se debe a que hasta el momento se manipulan archivos de datos de acuerdo con su orden físico. Más adelante, se discutirán situaciones donde los archivos respetan un orden lógico de datos.

El Ejemplo 3.2 presenta un seudocódigo donde se intenta resolver el problema planteado.

EJEMPLO 3.2

```
program ejemplo_3_2;
  type
       producto = record
         cod: str4;
         descripcion: string[30];
         pu: real;
         stock: integer;
```

continúa >>>

```
    end;
    venta_prod = record
      cod: str4;
      cant_vendida: integer;
    end;
    detalle = file of venta_prod;
    maestro = file of producto;
var
  regm: producto;
  regd: venta_prod;
  mae1: maestro;
  det1: detalle;
  cod_actual: str4;
  tot_vendido: integer;
begin
  assign (mae1, 'maestro');
  assign (det1, 'detalle');
  reset (mae1);
  reset (det1);
  while (not eof(det1)) do
    begin
      read(mae1,regm);
      read(det1,regd);
      {se busca en el maestro el producto del detalle}
      while (regm.cod <> regd.cod) do
       read (mae1,regm);

      {se totaliza la cantidad vendida del detalle}
       cod_actual := regd.cod;
       tot_vendido := 0;
       while (regd.cod = cod_actual) do
        begin
          tot_vendido := tot_vendido + regd.cant_vendida;
          read(det1,regd);
        end;

       {se modifica el stock del producto con la cantidad vendida
       de ese producto}
       regm.stock := regm.stock - tot_vendido;

      {se reubica el puntero en el maestro}
       seek (mae1, filepos(mae1)-1);

      {se actualiza el maestro}
       write(mae1,regm);

    end;
    close(det1);
    close(mae1);
end.
```

Si se realiza un análisis detallado de la solución, se encontrará que, entre este ejemplo y el anterior, solo se agrega una iteración que permite agrupar todos los registros del detalle que modificarán a un elemento del maestro, y a partir de ese agrupamiento, actualizar efectivamente el registro correspondiente del maestro.

Sin embargo, esta solución presenta algunos inconvenientes. Se puede observar que la segunda operación read sobre el archivo detalle se hace sin controlar el fin de datos de este último. Esto podría solucionarse agregando una selección `(if)` que permita controlar dicha operación. Pero cuando finaliza la iteración interna, al retornar a la iteración principal, se lee otro registro del archivo detalle, perdiendo de esa forma un dato ya leído.

La solución presentada en el Ejemplo 3.2 no es correcta: genera inconvenientes, y las posibles soluciones son modificaciones forzadas para lograr una solución.

Por la forma de trabajo de los archivos en lenguajes como Pascal, este tipo de problemas requiere implantar soluciones diferentes. El inconveniente se plantea con la operación `read` sobre el archivo detalle. Es necesario leer fuera de la iteración principal del algoritmo y, más adelante, dentro de la iteración. Por lo tanto, el control del `EOF` debe resolverse de otra manera. El Ejemplo 3.3 presenta una solución alternativa.

EJEMPLO 3.3

```
program ejemplo_3_3;
  const valoralto='9999';
  type str4 = string[4];
       producto = record
         cod: str4;
         descripcion: string[30];
         pu: real;
         cant: integer;
       end;
       venta_prod = record
         cod: str4;
         cant_vendida: integer;
       end;
       detalle = file of venta_prod;
       maestro = file of producto;
  var
    regm: producto;
    regd: venta_prod;
    mae1: maestro;
    det1: detalle;
    total: integer;
    aux: str4;
```

continúa >>>

```
procedure leer (var archivo:detalle; var dato:venta_prod);
    begin
      if (not eof(archivo))
       then
         read (archivo,dato)
       else
         dato.cod:= valoralto;
    end;

{programa principal}
begin
     assign (mae1, 'maestro');
     assign (det1, 'detalle');
     reset (mae1);
     reset (det1);
     read(mae1,regm);
     leer(det1,regd);

    {se procesan todos los registros del archivo detalle}
     while (regd.cod <> valoralto) do
       begin
        aux := regd.cod;
        total := 0;

       {se totaliza la cantidad vendida de productos iguales
       en el archivo de detalle}
        while (aux = regd.cod ) do
          begin
            total := total + regd.cant_vendida;
            leer(det1,regd);
          end;

        {se busca en el maestro el producto del detalle}
        while (regm.cod <> aux) do
          read (mae1,regm);

        {se modifica el stock del producto con la cantidad
        total vendida de ese producto}
        regm.cant := regm.cant - total;

        {se reubica el puntero en el maestro}
        seek (mae1, filepos(mae1)-1);

        {se actualiza el maestro}
        write(mae1,regm);

        {se avanza en el maestro}
        if (not eof (mae1))
          then
            read(mae1,regm);

     end;

  close (det1);
  close (mae1);

end.
```

Puede observarse ahora que se presenta una solución que utiliza un procedimiento de lectura, y un control de fin de proceso que va más allá del control del `EOF`.

El procedimiento de lectura, denominado `leer`, es el responsable de realizar el `read` correspondiente sobre el archivo, en caso de que este tuviera más datos. En caso de alcanzar el fin de archivo, se asigna a la variable `dato.cod`, por la cual el archivo está ordenado, un valor imposible de alcanzar en condiciones normales de trabajo. Este valor indicará que el puntero del archivo ha llegado a la marca de fin.

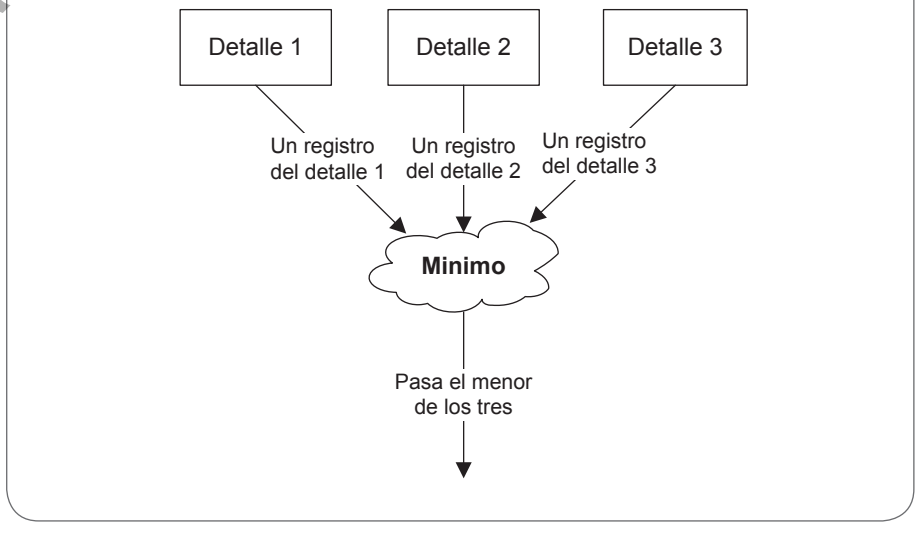
En el módulo principal se controla el fin del procesamiento evaluando el valor que tiene la variable por la cual el archivo está ordenado (regd. cod), controlando que no tome el valor que indique indirectamente `EOF`.

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.

El Ejemplo 3.4 presenta la resolución de un algoritmo de actualización a partir de tres archivos detalle. Para ello, se agrega un nuevo procedimiento, denominado `minimo`, 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. En el Ejemplo 3.4, el procedimiento mínimo retorna el código mínimo de producto. La Figura 3.1 presenta la situación gráficamente.

Figura 3.1



El objetivo del procedimiento `minimo` es determinar el menor de los tres elementos recibidos de cada archivo (para poder retornarlo como el más pequeño) y leer otro registro del archivo desde donde provenía ese elemento.

De esta forma, el resto del algoritmo del Ejemplo 3.3 se mantiene igual. El fin de datos se controla con el mismo criterio. En el momento en que los tres archivos detalle se hayan recorrido hasta su último elemento, el menor elemento retornado corresponderá a la marca de fin utilizada. De este modo finaliza el procesamiento.

EJEMPLO 3.4

```
program ejemplo_3_4;

  const valoralto='9999';

  type str4 = string[4];

       producto = record
         cod: str4;
         descripcion: string[30];
         pu: real;
         cant: integer;
       end;

       venta_prod = record
         cod: str4;
         cant_vendida: integer;
       end;

       detalle = file of venta_prod;
       maestro = file of producto;
var

 regm: producto;
 min, regd1, regd2,regd3: venta_prod;
 mae1: maestro;
 det1,det2,det3: detalle;
 aux: str4;
 total_vendido:integer;

procedure leer (var archivo:detalle; var dato:venta_prod);
begin
  if (not eof(archivo))then
    read (archivo,dato)
  else
    dato.cod:= valoralto;
end;

procedure minimo (var r1,r2,r3,min:venta_prod);
begin
  if (r1.cod<=r2.cod) and (r1.cod<=r3.cod)then
    begin
      min := r1;
      leer(det1,r1);
    end
  else
    if (r2.cod<=r3.cod)then
      begin
```

continúa >>>

```
        min := r2;
        leer(det2,r2);
      end
    else
      begin
        min := r3;
        leer(det3,r3)
      end;
end;

{programa principal}
begin
  assign (mae1, 'maestro');
  assign (det1, 'detalle1');
  assign (det2, 'detalle2');
  assign (det3, 'detalle3');
  reset (mae1);
  reset (det1);
  reset (det2);
  reset (det3);
  read(mae1,regm);
  leer(det1, regd1);
  leer(det2, regd2);
  leer(det3, regd3);
  minimo(regd1,regd2,regd3,min);

{se procesan todos los registros de los archivos detalle}
  while (min.cod <> valoralto) do
   begin
     {se totaliza la cantidad vendida de productos iguales en
     el archivo de detalle}
     aux := min.cod;
     total_vendido := 0
     while (aux = min.cod ) do
       begin
         total_vendido := total_vendido + min.cantvendida;
         minimo(regd1,regd2,regd3,min);
       end;

     {se busca en el maestro el producto del detalle}
     while (regm.cod <> min.cod) do
       read(mae1,regm);

     {se modifica el stock del producto con la cantidad total
     vendida de ese producto}
     regm.cant := regm.cant - total;

     {se reubica el puntero en el maestro}
     seek (mae1, filepos(mae1)-1);

     {se actualiza el maestro}
     write(mae1,regm);

     {se avanza en el maestro}
     if (not eof (mae1))
       then read(mae1,regm);
   end;

  close (det1);
  close (det2);
  close (det3);
  close (mae1);

end.
```

Proceso de generación de un nuevo archivo a partir de otros existentes. *Merge*

El segundo grupo de problemas presentado en este capítulo está vinculado con la generación de un archivo que resuma información de uno o varios archivos 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.

Se muestran algunas variantes de este tipo de problemas.

Primer ejemplo

El primer ejemplo plantea un problema muy simple. Las precondiciones son las siguientes:

- Se tiene información en tres archivos detalle.
- Esta información se encuentra ordenada por el mismo criterio en cada caso.
- La información es disjunta, esto significa que un elemento puede aparecer una sola oportunidad en todo el problema. Si el elemento 1 está en el archivo detalle1, solo puede aparecer una vez en este y no podrá estar en el resto de los archivos.

EJEMPLO 3.5

```
program ejemplo_3_5;

  const valoralto='9999';

  type str4 = string[4];

       producto = record
         codigo: str4;
         descripcion: string[30];
         pu: real;
         cant: integer;
       end;

       detalle = file of producto;
var

 min, regd1, regd2,regd3: producto;
 det1, det2, det3, mae1: detalle;

procedure leer (var archivo:detalle; var dato:producto);
begin
```

continúa >>>

```
  if (not eof(archivo)) then
    read (archivo,dato)
  else
    dato.codigo:= valoralto;
end;

procedure minimo (var r1,r2,r3,min:producto);
begin
  if (r1.codigo<=r2.codigo) and (r1.codigo<=r3.codigo) then
    begin
      min := r1;
      leer(det1,r1);
    end
  else
    if (r2.cod<=r3.cod) then
      begin
        min := r2;
        leer(det2,r2);
      end
    else begin
        min := r3;
        leer(det3,r3)
    end;
 end;

{programa principal}
begin
  assign (mae1, 'maestro');
  assign (det1, 'detalle1');
  assign (det2, 'detalle2');
  assign (det3, 'detalle3');

  rewrite (mae1);
  reset (det1);
  reset (det2);
  reset (det3);

  leer(det1, regd1);
  leer(det2, regd2);
  leer(det3, regd3);

  minimo(regd1,regd2,regd3,min);

  {se procesan todos los registros de los archivos detalle}
   while (min.codigo <> valoralto) do
    begin
      write(mae1, min);
      minimo(regd1,regd2,regd3,min);
    end;

  close (det1);
  close (det2);
  close (det3);
  close (mae1);

end.
```

El Ejemplo 3.5 presenta la solución al problema planteado. Este algoritmo tiene aspectos similares al ejemplo anterior, dado que se utilizan tres archivos de detalle y los procedimientos `minimo` y `leer` se resuelven en forma análoga.

Segundo ejemplo

Como segundo ejemplo se presenta un problema similar, pero ahora los elementos se pueden repetir dentro de los archivos detalle, modificando de esta forma la tercera precondición del ejemplo anterior. El resto de las precondiciones permanecen inalteradas.

EJEMPLO 3.6

```
program ejemplo_3_6;

  const valoralto='9999';

  type str4 = string[4];

       producto = record
          codigo: str4;
          nombre: string[30];
          cant: integer;
       end;
       detalle =  file of producto;
var

 min, regd1, regd2, regd3, regmae: ventas;
 det1, det2, det3, mae1: detalle;
 codprod: str4;
 cantotal: integer;

 procedure leer (var archivo:detalle; var dato:producto);
 begin
   if (not eof(archivo)) then
     read (archivo,dato)
   else
     codigo:= valoralto;
end;

procedure minimo (var r1,r2,r3,min:producto);
begin
  if (r1.codigo<=r2.codigo) and (r1.cod<=r3.codigo) then
    begin
      min := r1;
      leer(det1,r1);
    end
   else
```

continúa >>>

```
      if (r2.codigo<=r3.codigo)then
        begin
          min := r2;
          leer(det2,r2);
        end
      else
        begin
          min := r3;
          leer(det3,r3)
        end;
 end;

{programa principal}
begin
  assign (mae1, 'maestro');
  assign (det1, 'detalle1');
  assign (det2, 'detalle2');
  assign (det3, 'detalle3');

  rewrite (mae1);
  reset (det1);
  reset (det2);
  reset (det3);

  leer(det1, regd1);
  leer(det2, regd2);
  leer(det3, regd3);

  minimo(regd1,regd2,regd3,min);

  {se procesan todos los registros de los archivos detalle}
   while (min.codigo <> valoralto) do
   begin
     codprod := min.codigo;
     cantotal := 0;
     {se procesan todos los registros del mismo producto}
     while (codprod=min.codigo)
       begin
          cantotal:= cantotal + min.cant;
          minimo(regd1,regd2,regd3,min);
       end;

     write(mae1, min);

    end;

  close (det1);
  close (det2);
  close (det3);
  close (mae1);

end.
```

El algoritmo del ejemplo 3.6 expone la solución al problema. Se deberá notar que:

- Los elementos que correspondan al mismo código de producto deben agruparse para resumir la información. Una vez agrupados y resumidos, son almacenados en el archivo maestro.
- La solución utiliza la misma estrategia utilizada para ejemplos anteriores.

Corte de control

Se denomina corte de control al proceso mediante el cual la información de un archivo es presentada en forma organizada de acuerdo con la estructura que tiene el archivo.

Suponga que se almacena en un archivo la información de ventas de una cadena de electrodomésticos. Dichas ventas han sido efectuadas por los vendedores de cada sucursal de cada ciudad de cada provincia del país. Luego, es necesario informar al gerente de ventas de la empresa el total de ventas producidas de acuerdo con el siguiente formato:

```
Provincia: ……
Ciudad: ……
Sucursal: ……
Vendedor 1        $$
Vendedor n        $$
Total sucursal:   $$
Sucursal: ……
Vendedor 1        $$
Vendedor n        $$
Total sucursal:   $$
Total ciudad:     $$
Ciudad: ……
Total ciudad:     $$
Total prov.:      $$
Prov.: ……..……
Total ciudad:     $$
Total prov.:      $$
Total empresa:    $$
```

Deben tenerse en cuenta las siguientes precondiciones:

- El archivo se encuentra ordenado por provincia, ciudad, sucursal y vendedor.
- Se debe informar el total de vendido en cada sucursal, ciudad y provincia, así como el total final.
- En diferentes provincias pueden existir ciudades con el mismo nombre, o en diferentes ciudades pueden existir sucursales con igual denominación.

El Ejemplo 3.7 presenta una solución para el problema planteado. El lector podrá notar que el algoritmo utiliza el proceso de lectura definido previamente en este capítulo.

EJEMPLO 3.7

```
program ejemplo_3_7;

  const valoralto=”ZZZ”;

  type nombre = string[30];

        RegVenta = record
             Vendedor: integer;
             MontoVenta: real;
             Sucursal: nombre;
             Ciudad: nombre;
             Provincia: nombre;
        end;
        Ventas =  file of RegVenta;
var

  reg: RegVenta;
  archivo: Ventas;
   total, totprov, totciudad, totsuc: integer;
   prov, ciudad, sucursal: nombre;

procedure leer (var archivo: Ventas; var dato:RegVenta);
begin
  if (not eof(archivo)) then
    read (archivo,dato)
  else
    dato.provincia:= valoralto;
end;

{programa principal}
begin
  assign (archivo, ‘archivoventas’);
  reset (archivo);
```

continúa >>>

```
leer(archivo, reg);
total:= 0;

while (reg.Provincia <> valoralto) do
  begin
    writeln("Provincia:", reg.Provincia);
    prov := reg.Provincia;
    totprov := 0;

    while (prov=reg.Provincia)do
      begin
        writeln("Ciudad:", reg.Ciudad);
        ciudad := reg.Ciudad;
        totciudad := 0

        while (prov=reg.Provincia) and (Ciudad=reg.Ciudad)do
          Begin
            writeln("Sucursal:", reg.Sucursal);
            sucursal := regSucursal;
            totsuc := 0;

            while ((prov=reg.Provincia) and
                   (Ciudad=reg.Ciudad) and
                   (Sucursal=regSucursal)) do
              Begin
                write("Vendedor:", reg.Vendedor);
                writeln(reg.MontoVenta);
                totsuc := totsuc + reg.MontoVenta;
                leer(archivo, reg);
              end;

            writeln("Total Sucursal", totsuc);
            totciudad := totciudad + totsuc;
          end;

        writeln("Total Ciudad", totciudad);
        totprov := totprov + totciudad;
      end;

    writeln("Total Provincia", totprov);
    total := total + totprov,
  end;

writeln("Total Empresa", total);

close (archivo);

end.
```

Cuestionario del capítulo

1. ¿Cuáles son los procesos denominados algorítmica clásica sobre archivos?
2. ¿Por qué fue necesario utilizar el procedimiento para leer del archivo en algunos ejercicios del capítulo?
3. ¿Por qué el proceso de *merge* utilizó una apertura del archivo maestro con `rewrite`?
4. ¿Por qué se consideró que los archivos están físicamente ordenados?

Ejercitación

1. Una empresa posee un **archivo ordenado por código de promotor** con información de las ventas realizadas por cada uno de ellos (código de promotor, nombre y monto de venta). Sabiendo que en ese archivo pueden existir uno o más registros por cada promotor, realice un procedimiento que reciba el archivo anteriormente mencionado y lo compacte (esto es, generar un nuevo archivo donde cada promotor aparezca una única vez con sus ventas totales).
2. El encargado de ventas de un negocio desea administrar el *stock* de productos que vende. Para ello, genera un archivo maestro donde figuran todos los productos comercializados. Se maneja la siguiente información: código de barras, nombre comercial, proveedor, *stock* actual, *stock* mínimo. Diariamente se genera un archivo detalle donde se asientan todas las ventas de productos realizadas. Se pide generar un algoritmo que permita actualizar el archivo maestro con el detalle sabiendo que: i) ambos archivos están ordenados por código de barras del producto; ii) cada registro del maestro puede ser actualizado por 0, 1 o más registros del detalle; iii) el archivo detalle solamente contiene registros que están en el archivo maestro.
   a) Realice un proceso que reciba al archivo maestro y genere un listado donde figuren los productos cuyo *stock* está por debajo del mínimo permitido.
   b) Ídem anterior, generando, ahora, un archivo donde figure el nombre de los productos cuyo *stock* está por debajo del mínimo permitido.

3. El servicio meteorológico provincial posee un archivo con información de lluvias registradas mensualmente en las ciudades de la provincia durante el año 2007. Dicho organismo recibe un archivo con la información de los servicios regionales, los cuales indican la cantidad llovida para un mes determinado en su ciudad.

   a. Lea cuidadosamente el ejercicio y defina la estructura de ambos archivos.

   b. Genere ambos archivos.

   c. Actualice el archivo de servicios meteorológicos de la provincia con la información recibida de los servicios regionales.

   Nota: al generar los archivos, suponga que la información ingresa ordenada por ciudad. Además, el archivo de los servicios regionales puede contener varios datos de una misma ciudad.

4. a. Se necesita contabilizar los votos de las diferentes mesas electorales por provincia y localidad. Para ello, se posee un archivo con la siguiente información: código de provincia, código de localidad, número de mesa y cantidad de votos. Realice un listado como se muestra a continuación:

Código de provincia:
Código de localidad Total de votos
....
....
Total de votos por provincia:

Código de provincia:
Código de localidad Total de votos
....
Total de votos por provincia:
....

Nota: la información viene ordenada por código de provincia y código de localidad.

   b. En caso de que tuviera que resolver el inciso a pero sin la precondición de tener el archivo ordenado, ¿cómo lo resolvería? Justifique.

5. Una gran proveeduría deportiva tiene organizado su sistema de forma tal que cada sección genera su propio archivo con código de vendedor, nombre del vendedor, producto vendido, cantidad vendida y precio unitario del producto. Mensualmente, envía sus archivos a la sección administración. Se sabe que la proveeduría está dividida en cinco secciones (zapatillería, vestimenta, pesca, caza y *camping*) y que los vendedores están capacitados para realizar ventas en cualquier sector. Además, se sabe que los archivos están organizados por código de vendedor y que estos pueden realizar varias ventas. Se pide obtener un archivo que registre la información de cada vendedor y el monto total obtenido por sus ventas del mes correspondiente.

Capítulo 4

Eliminación de datos. Archivos con registros de longitud variable

Objetivo

Este capítulo tiene dos objetivos fundamentales. El primero de ellos expone y discute el problema de eliminación de información contenida en archivos. Este proceso se puede llevar a cabo mediante algoritmos que operen de manera física o lógica sobre un archivo con registros de longitud fija. A lo largo de la primera parte del capítulo, se presentan las dos soluciones y se discute en detalle la eficiencia de estas. Asimismo, se presentan propuestas alternativas, que, en gran parte, mejoran la *performance* final del proceso de baja.

El segundo objetivo del capítulo está vinculado con archivos que contienen registros de longitud variable. En relación con este tema, se analizarán alternativas para realizar la algorítmica básica –altas, bajas, modificaciones y consultas–, comparando la eficiencia obtenida con la presentada anteriormente para archivos con registros de longitud fija.

Proceso de bajas

Se denomina **proceso de baja** a aquel proceso que permite quitar información de un archivo.

El proceso de baja puede ser analizado desde dos perspectivas diferentes: aquella ligada con la algorítmica y *performance* necesarias para borrar la información, y aquella que tiene que ver con la necesidad real de quitar información de un archivo en el contexto informático actual.

Históricamente, el proceso de baja era necesario para no mantener información poco útil o relevante en un archivo y, por consiguiente, recuperar espacio en dispositivos de almacenamiento secundario. En la actualidad, las organizaciones que disponen de BD consideran la información como su bien más preciado. De esta forma, el concepto de borrar información queda condicionado. En general, el conocimiento adquirido (información) no se quita, sino que se preserva en archivos o repositorios históricos. Teniendo en cuenta esta situación, el proceso de baja se relativiza en importancia. Sin embargo, en esta sección se presenta el proceso algorítmico inherente, con un análisis general de *performance*.

El proceso de baja puede llevarse a cabo de dos modos diferentes:

- **Baja física:** consiste en borrar efectivamente la información del archivo, recuperando el espacio físico.
- **Baja lógica:** consiste en borrar la información del archivo, pero sin recuperar el espacio físico respectivo.

Baja física

Se realiza baja física sobre un archivo cuando un elemento es quitado del archivo y es reemplazado por otro elemento del mismo archivo, decrementando en uno su cantidad de elementos.

La ventaja de este método consiste en administrar, en cada momento, un archivo de datos que ocupe el lugar mínimo necesario. La desventaja, como se discutirá a continuación, tiene que ver con la *performance* final de los algoritmos que implementan esta solución.

Para realizar el proceso de baja física existen, básicamente, dos técnicas algorítmicas:

- Generar un nuevo archivo con los elementos válidos, es decir, sin copiar los que se desea eliminar.
- Utilizar el mismo archivo de datos, generando los reacomodamientos que sean necesarios.

Baja física generando nuevo archivo de datos

El algoritmo del Ejemplo 4.1 presenta el caso de borrado físico generando un nuevo archivo de datos. La precondición del problema presentado en el ejemplo es:

- Se dispone de un archivo de empleados donde figura el empleado “Carlos García”.

EJEMPLO 4.1

```
program ejemplo_4_1;
  const valoralto="ZZZ";
  type
      empleado = record;
            Nombre         : string[50];
            Direccion      : string[50];
            Documento      : string[12];
            Edad           : integer;
            Observaciones: string[200];
      end;
     archivoemple = file of Empleado;
var
 reg: empleado;
 archivo, archivonuevo: archivoemple;
procedure leer (var archivo: archivoemple; var dato:empleado);
    begin
      if (not eof(archivo))
        then
            read (archivo,dato)
        else
            dato.Nombre := valoralto;
    end;

{programa principal}
begin
  assign (archivo, 'archemple');
  assign (archivonuevo, 'archnuevo');
  reset(archivo);
  rewrite(archivonuevo);
  leer(archivo,reg)

  {se copian los registros previos a Carlos Garcia}
  while (reg.Nombre<>"Carlos Garcia")
    begin
       write(archivonuevo,reg)
       leer(archivo,reg)
    end

  {se descarta a Carlos Garcia}
  leer(archivo,reg)

  {se copian los registros restantes}
  while (reg.Nombre<>valoralto) do
    begin
       write(archivonuevo,reg)
       leer(archivo,reg)
    end

  close(archivonuevo);
  close(archivo);
end.
```

El algoritmo presentado consiste en recorrer el archivo original de datos, copiando al nuevo archivo toda la información, menos la correspondiente al elemento que se desea quitar.

Una vez finalizado el proceso, se debe eliminar el archivo inicial del dispositivo de memoria secundaria, renombrando luego al generado con el nombre original.

Un análisis de *performance* básico determina que este método necesita leer tantos datos como tenga el archivo original y escribir todos los datos, salvo el que se elimina. Esto significa, 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, en el orden en el cual aparecen en el archivo (es decir, para leer el registro *n*, antes se debe leer desde el registro 1 hasta el registro *n*-1) sobre ambos archivos. Este nivel de *performance* será de utilidad para comparar la eficiencia de esta solución con las que se presentarán más adelante en este capítulo.

Se debe notar que, una vez finalizado el proceso de generación del nuevo archivo, coexisten en el disco rígido dos archivos: el original y el nuevo sin el registro borrado. Esto significa que se debe disponer en memoria secundaria de la capacidad de almacenamiento suficiente para ambos archivos.

Baja física utilizando el mismo archivo de datos

El caso de borrado físico utilizando el mismo archivo de datos se presenta en el Ejemplo 4.2, donde se quita del archivo al empleado sin generar otro archivo. La precondición del problema presentado en el Ejemplo 4.2 es la misma que la presentada para el Ejemplo 4.1.

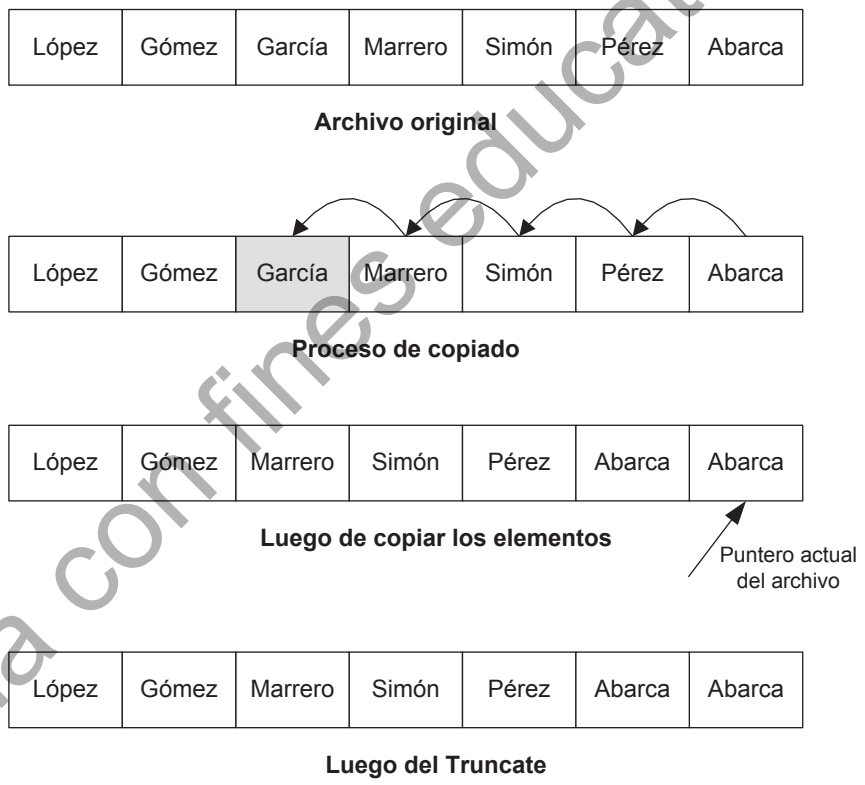
EJEMPLO 4.2

```
program ejemplo_4_2;
  const valoralto="ZZZ";
  type
      empleado = record;
            Nombre         : string[50];
            Direccion      : string[50];
            Documento      : string[12];
            Edad           : integer;
            Observaciones:   string[200];
     end;
    archivoemple = file of Empleado;
var
 reg: empleado;
 archivo: archivoemple;

procedure leer (var archivo: archivoemple; var dato:empleado);
    begin
      if (not eof(archivo))
        then
            read (archivo,dato)
        else
            dato.Nombre := valoralto;
    end;

{programa principal}
begin
  assign (archivo, 'archemple');
  reset(archivo);
  leer(archivo,reg);

 {se avanza hasta Carlos Garcia}
 while (reg.Nombre<>"Carlos Garcia")
  begin
   leer(archivo,reg);
  end;

{se avanza hasta el siguiente a Carlos Garcia}
leer(archivo,reg);

{se copian los registros restantes}
While (reg.Nombre<>valoralto) do
  begin
     Seek( archivo, filepos(archivo)-2 );
     write(archivo,reg);
     Seek( archivo, filepos(archivo)+1 );
     leer(archivo,reg);
  end;

 truncate(archivo);
end.
```

Para poder quitar el elemento a borrar, el algoritmo requiere localizar, primero, al empleado Carlos García. Luego, copiar sobre este registro el elemento siguiente y así sucesivamente, repitiendo esta operatoria hasta el final del archivo.

La Figura 4.1 presenta en forma gráfica el proceso. Se debe notar que el archivo no fue cerrado utilizando la instrucción close(); en su reemplazo se utilizó truncate(). La diferencia entre ambas 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.

Figura 4.1



El análisis de *performance*, para este ejemplo, 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. Es de notar que, a diferencia del método anterior, 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

Se realiza una baja lógica sobre un archivo cuando el elemento que se desea quitar es marcado como borrado, pero sigue ocupando el espacio dentro del archivo. 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. Como se verá en este capítulo, esto puede ser compensado con técnicas alternativas para el alta de nuevos elementos.

El Ejemplo 4.3 presenta el algoritmo que elimina de manera lógica a un registro del archivo. En este caso, la precondición del problema es similar a los ejemplos anteriores de este capítulo.

EJEMPLO 4.3

```
program ejemplo_4_3;

  const valoralto="ZZZ";

  type
      empleado = record;
                Nombre       : string[50];
                Direccion    : string[50];
                Documento    : string[12];
                Edad         : integer;
                Observaciones: string[200];
       end;
       archivoemple = file of Empleado;
var
 reg: empleado;
 archivo: archivoemple;

procedure leer (var archivo: archivoemple; var dato:empleado);
begin
  if (not eof(archivo)) then
    read (archivo,dato)
  else
    dato.Nombre := valoralto;
end;

{programa principal}
begin
  assign (archivo, 'archemple');
  reset(archivo);
  leer(archivo,reg);
```

continúa >>>

```
  {se avanza hasta Carlos Garcia}
  while (reg.Nombre<>"Carlos Garcia")do
    begin
      leer(archivo,reg);
    end;

  {se genera una marca de borrado }
  reg.nombre := "***"

  {se borra lógicamente a Carlos Garcia}
  Seek( archivo, filepos(archivo)-1 );
  write(archivo,reg);

  close(archivo);
end.
```

El Ejemplo 4.4 muestra la forma en que debería recorrerse el archivo luego de ejecutado el Ejemplo 4.3. Así, ahora se consideran válidos solo aquellos elementos que no se encuentren indicados como borrados.

EJEMPLO 4.4

```
program ejemplo_4_4;
  const valoralto="ZZZ";
  type
       empleado = record;
              Nombre        : string[50];
              Direccion     : string[50];
              Documento     : string[12];
              Edad          : integer;
              Observaciones: string[200];
      end;
      archivoemple = file of Empleado;
var
 reg: empleado;
 archivo: archivoemple;

procedure leer (var archivo: archivoemple; var dato:empleado);
begin
  if (not eof(archivo)) then
    read (archivo,dato)
  else
    dato.Nombre := valoralto;
end;

{programa principal}
begin
  assign (archivo, 'archemple');
  reset(archivo);
  leer(archivo,reg);
  {se avanza hasta Carlos Garcia}
  while (reg.Nombre<>valoralto)do
    begin
      if (reg.Nombre<>'***')then
        write('El nombre de es:'+reg.Nombre)
      leer(archivo,reg);
    end;
  close(archivo);
end.
```

Es posible, en este momento, hacer una comparación de los dos métodos de baja. Las ventajas, en cada caso, tienen que ver con la *performance* del algoritmo y el espacio utilizado en el disco rígido. Mientras que la baja lógica no recupera espacio en memoria secundaria, se comporta de forma mucho más eficiente en el tiempo de respuesta. Además, es posible combinar el proceso de baja lógica con el proceso de ingreso de nueva información al archivo de datos.

Recuperación de espacio

El proceso de baja lógica marca la información de un archivo como borrada. Ahora bien, esa información sigue ocupando espacio en el disco rígido. La pregunta a responder sería: ¿qué hacer con dicha información? Esta pregunta tiene dos respuestas posibles:

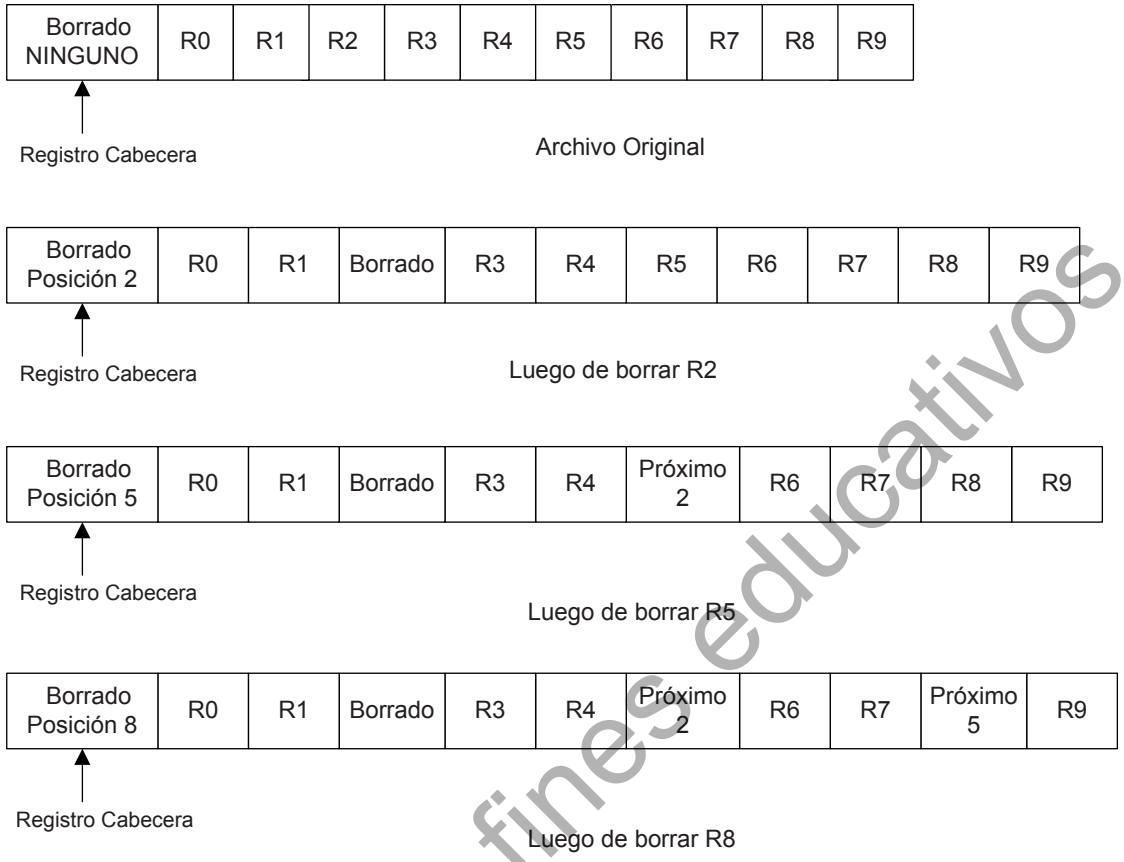
- **Recuperación de espacio:** periódicamente utilizar el proceso de baja física para realizar un proceso de compactación del archivo. El mismo consiste en quitar todos aquellos registros marcados como borrados, utilizando para ello cualquiera de los algoritmos discutidos anteriormente para borrado físico.
- **Reasignación de espacio:** otra alternativa posible consiste en recuperar el espacio, utilizando los lugares indicados como borrados para el ingreso (altas) de nuevos elementos al archivo.

Reasignación de espacio

Esta técnica consiste en reutilizar el espacio indicado como borrado para que nuevos registros se inserten en dichos lugares. Así, el proceso de alta discutido en capítulos anteriores se vería modificado; en lugar de avanzar sobre la última posición del archivo (donde se encuentra la marca de `EOF`), se debe localizar alguna posición marcada como borrada para insertar el nuevo elemento en dicho lugar.

Este proceso puede realizarse buscando los lugares libres desde el comienzo del archivo, pero se debe considerar que de esa manera sería muy lento. La alternativa consiste en recuperar el espacio de forma eficiente. Para ello, y como lo muestra la Figura 4.2, 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

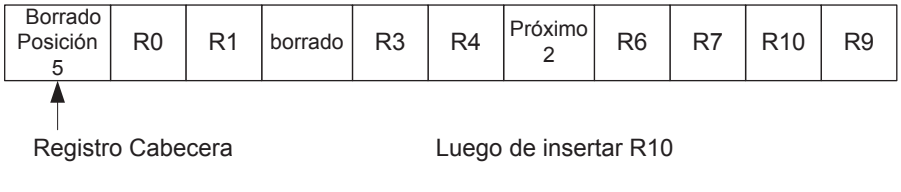
Figura 4.2



Se puede observar que en la posición donde estaba el registro 2 aparece la leyenda Borrado, en tanto que, en las demás posiciones, la leyenda indica Próximo y el número del siguiente registro borrado. Esto significa cuál es la siguiente posición libre, en tanto que para la posición 2, al no haber elemento siguiente, la leyenda Borrado indica fin de la lista.

Ahora, para recuperar el espacio, el proceso de alta debe, primeramente, verificar la lista por espacios libres. En caso de encontrar algún espacio libre, el registro se deberá insertar en esa posición, actualizando el registro cabecera, tal como lo muestra la Figura 4.3.

Figura 4.3



Luego de insertar el registro 10, en la posición donde antes se encontraba el registro 8, el registro cabecera del archivo es actualizado, apuntando ahora a la posición del registro 5.

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.

Campos y registros con longitud variable

La información en un archivo siempre es homogénea. Esto es, todos los elementos almacenados en él son del mismo tipo. De esta forma, cada uno de los datos es del mismo tamaño, generando lo que se denomina archivos con registros de longitud fija.

La longitud de cada registro está determinada por la información que se guarda.

- Si se define un archivo que contiene números enteros, cada elemento ocupa 2 bytes.
- Si se define un archivo que contiene caracteres, cada elemento ocupa 1 byte.
- Si se define un archivo que contiene String[20], cada elemento ocupa 20 bytes.
- Si se define un archivo que contiene registros con Nombre y apellido (string[40]), DNI (string[8]), Edad (integer), cada elemento ocupa 50 bytes (la suma de la longitud de cada uno de los campos).

Administrar archivos con registros de longitud fija tiene algunas importantes ventajas: el proceso de entrada y salida de información, desde y hacia los *buffers,* es responsabilidad del sistema operativo; los procesos de alta, baja y modificación de datos se corresponden con todo lo visto hasta el momento.

No obstante, hay determinados problemas donde no es posible, no es deseable trabajar con registros de longitud fija. Supóngase el siguiente ejemplo. Se define un registro empleado con la siguiente estructura:

```
Type Empleado = Record;
       Nombre       : string[50];
       Direccion    : string[50];
       Documento    : string[12];
       Edad         : integer;
       Observaciones: string[200];
     End;
```

El registro anterior ocupa 314 bytes. Cada vez que se ingresan los datos de un empleado, en promedio se ocupan 30 lugares para el nombre, otro tanto para la dirección y, en algunos casos, 50 bytes para observaciones. En promedio, cada registro utiliza 124 de los 314 bytes disponibles. Esto significa desperdicio de espacio y mayor tiempo de procesamiento. Se transfieren 314 bytes de los cuales solo 124 representan información útil; el resto son espacios de relleno.

Para evitar estas situaciones, es de interés contar con alguna organización de archivos que solo utilice el espacio necesario para almacenar la información. Este tipo de soluciones se representan con archivos donde los registros utilicen longitud variable. En estos casos, como el nombre lo indica, la cantidad de espacio utilizada por cada elemento del archivo no está determinada *a priori*.

Para poder arribar a soluciones de este tipo, es necesario que los archivos de datos tengan una estructura diferente de lo visto hasta el momento. La declaración del archivo vista hasta el momento determina el tipo de dato que contendrá y, por consiguiente, el tamaño de cada elemento. Es necesario entonces definir el archivo de otra forma. Para ello existen varias soluciones, pero por razones prácticas en este libro se tratará al archivo de datos como una secuencia de caracteres, donde `EOF` seguirá representando la marca de fin de archivo.

Entonces, cada elemento de dato debe descomponerse en cada uno de sus elementos constitutivos y así, elemento a elemento, guardarse en el archivo. En el caso de necesitar transferir un string, debe hacerse carácter a carácter; en caso de tratarse de un dato numérico, cifra a cifra.

El siguiente problema consiste en definir dónde empieza y dónde termina cada registro o cada campo del archivo. Es necesario colocar marcas que delimiten cada elemento. Estas marcas se denominan marcas de fin de campo o marcas de fin de registro, en cada caso, y pueden ser cualquier carácter que luego no deberá poder ser parte de un elemento de dato del archivo.

El Ejemplo 4.5 describe un algoritmo que crea un archivo de empleados, con nombre y apellido, dirección y documento, donde cada registro es tratado como de longitud variable.

EJEMPLO 4.5

```
program ejemplo_4_5;
Var
 empleados: file; {archivo sin tipo}
 nombre,apellido,direccion,documento: string;

Begin
  Assign(empleados,'empleados.txt');
  Rewrite(empleados,1);
  writeln('Ingrese el Apellido');
  readln(apellido);
  writeln('Ingrese el Nombre');
  readln(nombre);
  writeln('Ingrese direccion');
  readln(direccion);
  writeln('Ingrese el documento');
  readln(documento);

  while apellido<>'zzz' do
    Begin
      BlockWrite(empleados,apellido,length(apellido)+1);
      BlockWrite(empleados,'#',1);
      BlockWrite(empleados,nombre,length(nombre)+1);
      BlockWrite(empleados,'#',1);
      BlockWrite(empleados,direccion,length(direccion)+1);
      BlockWrite(empleados,'#',1);
      BlockWrite(empleados,documento,length(documento)+1);
      BlockWrite(empleados,'@',1);
      writeln('Ingrese el Apellido');
      readln(apellido);
      writeln('Ingrese el Nombre');
      readln(nombre);
      writeln('Ingrese direccion');
      readln(direccion);
      writeln('Ingrese el documento');
      readln(documento);
    end;
  close(empleados);
end.
```

Se deben notar las características diferentes que tiene este tipo de problema:

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 @.

Al utilizar registros de longitud variable, una de las principales diferencias con el uso de registros de longitud fija radica en las operaciones de lectura y escritura. 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. Queda como ejercicio intelectual desarrollar el Ejemplo 4.5, suponiendo que el registro es definido con longitud fija.

La primera conclusión que se puede obtener a partir del uso de registros de longitud variable es que 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.

El Ejemplo 4.6 muestra el algoritmo que permite recorrer el archivo anteriormente generado y presenta los datos en pantalla. Nuevamente, queda como tarea plantear la solución para archivos con registros de longitud fija.

EJEMPLO 4.6

```
program ejemplo_4_6;
Var
  empleados: file; {archivo sin tipo}
  campo, buffer :string;
Begin
  Assign(empleados,'empleados.txt');
  reset(empleados,1);
  while not eof(empleados) do
    Begin
      BlockRead(empleados,buffer,1);
      while (buffer<>'@') and not eof(empleados)do
        begin
          while ((buffer<>'@') and
                 (buffer<>'#') and
```

continúa >>>

```
                not eof(empleados))do
            begin
              campo := campo + buffer ;
              BlockRead(empleados,buffer,1);
            end;
          writeln(campo);
        end;

      if not eof(empleados) then
        BlockRead(empleados,nombre,1);
    end;
  close(empleados);
end.
```

Alternativas para registros de longitud variable

Cuando se plantea la utilización de espacio con longitud variable sobre técnicas de espacio fijo, existen algunas variantes que se pueden analizar.

En el apartado anterior, se trabajó con registros de longitud variable y campos de longitud variable. En ese caso, la solución planteada utilizó delimitadores de campo para describir el final de cada uno de ellos. Así, si un registro dispone de 4 campos, luego de obtener 4 marcas de fin se asume la finalización de un registro y el comienzo de uno nuevo. Esto implica que no es necesario indicar de forma explícita el fin del registro.

Existen otras variantes; estas tienen que ver con utilizar indicadores de longitud de campo y/o registro. De esta manera, antes de almacenar un registro, se indica su longitud; luego, los siguientes bytes corresponden a elementos de datos de dicho registro.

Eliminación con registros de longitud variable

El proceso de baja sobre un archivo con registros de longitud variable es, *a priori*, similar a lo discutido anteriormente. Un elemento puede ser eliminado de manera lógica o física. En este último caso, el modo de recuperar espacio es similar a lo planteado en los Ejemplos 4.3 o 4.4.

El proceso de baja lógica no tiene diferencias sustanciales con respecto a lo discutido anteriormente. Sin embargo, cuando se desea recuperar el espacio borrado lógicamente con nuevos elementos, deben tenerse en cuenta nuevas consideraciones. Estas tienen que ver con el espacio disponible. Mientras que con registros de longitud fija los elementos a eliminar e insertar son del mismo tamaño, utilizando

registros de longitud variable esta precondición no está presente. Para insertar un elemento no basta con disponer de lugar; es necesario, además, que el lugar sea del tamaño suficiente.

La filosofía para administración del lugar disponible se plantea, *a priori*, similar al caso discutido con registros de longitud fija. Para ello 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 quepa el registro a insertar.
- **Mejor ajuste:** consiste en seleccionar el espacio más adecuado para el registro. Se considera el espacio más adecuado como aquel de menor tamaño donde quepa el registro.
- **Peor ajuste:** consiste en seleccionar el espacio de mayor tamaño, asignando para el registro solo los bytes necesarios.

Fragmentación

Para detallar el proceso de selección de espacio, es necesario avanzar en conceptos de fragmentación. Existen dos tipos de fragmentación: interna y externa.

Se denomina **fragmentación interna** a aquella que se produce cuando a un elemento de dato se le asigna mayor espacio del necesario.

Los registros de longitud fija tienden a generar fragmentación interna, se asigna tanto espacio como lo necesario de acuerdo con la definición del tipo de dato. Pero este espacio no siempre se condice con lo que realmente utiliza el registro. Por ejemplo, si un campo del registro es `Nombre` definido como un string[50], y el nombre en cuestión es “Juan Perez”, solo se ocupan realmente 10 de los 50 lugares asignados.

Se denomina **fragmentación externa** al espacio disponible entre dos registros, pero que es demasiado pequeño para poder ser reutilizado.

Las dos fragmentaciones tienen que ver con la utilización del espacio. En el primer caso, fragmentación interna, se reserva lugar que no se utiliza; en tanto, la fragmentación externa se genera por dejar espacios tan pequeños que no pueden ser utilizados.

Fragmentación y recuperación de espacio

Como se mencionó anteriormente, el procedimiento de recuperación de espacio generado por bajas, utilizando registro de longitud variable, presenta tres alternativas.

Cada una de estas alternativas selecciona el espacio considerado más conveniente. Las técnicas de primer y mejor ajuste suelen implementar una variante que genera fragmentación interna. Así, una vez seleccionado el lugar libre, el espacio asignado corresponde a la totalidad de lo disponible. De esa forma, al nuevo registro se le puede asignar más del espacio necesario.

Por el contrario, la técnica de peor ajuste solo asigna el espacio necesario. Se recuerda que peor ajuste consiste en buscar sobre la lista de registros borrados el espacio disponible de mayor tamaño. Una vez localizado, se asignan solo los bytes necesarios y el resto del espacio libre sigue figurando en la lista de disponibles. De esta forma, es posible que la técnica de peor ajuste genere fragmentación externa dentro del archivo. Es deseable, en esos casos, disponer de un algoritmo que se ejecute periódicamente para la recuperación de estos espacios no asignados. Este tipo de algoritmo se conoce como *garbage collector*.

Conclusiones

Las tres técnicas definidas presentan ventajas y desventajas. Desde un punto de vista de *performance* para su implantación, el algoritmo que utilice primer ajuste tiende a ser el más rápido. Solo debe recorrerse la lista de registros borrados hasta encontrar el primer lugar donde quepa el nuevo elemento de datos. Dicho lugar se quita de la lista, asignando todo el espacio disponible al nuevo elemento.

Las alternativas de mejor y peor ajuste, en comparación con primer ajuste, son más ineficientes. En ambos casos, es necesario recorrer toda la lista para encontrar el lugar más adecuado según el caso. Una vez localizado dicho lugar, mejor ajuste se resuelve con mayor rapidez, se asigna dicho lugar completo y se quita de la lista. En tanto, peor ajuste debe asignar solo el espacio requerido y luego dejar en la lista el espacio restante con su respectiva capacidad.

Desde un punto de vista de la fragmentación generada, solo la técnica de peor ajuste mantiene la filosofía de trabajo de longitud variable, donde cada registro ocupa solamente el espacio que necesita.

Modificación de datos con registros de longitud variable

Hasta el momento, se discutieron los procesos de ingreso (alta) y borrado de datos sobre un archivo que contenga registros de longitud variable. Además, en el apartado anterior se presentaron alternativas de reutilización de espacio.

También es importante discutir el proceso de modificación de información de un archivo. Cuando se trabaja con archivos que soportan registros de longitud fija, una modificación consiste en sobreescribir un registro con el nuevo dato. Esto fue discutido en detalle en el Capítulo 3.

Sin embargo, cuando los archivos soportan registros de longitud variable, surge un nuevo problema. Modificar un registro existente puede significar que el nuevo registro requiera el mismo espacio en disco, que ocupe menos espacio o que requiera uno de mayor tamaño. Como es natural, el problema no se genera cuando ambos registros requieren el mismo espacio. 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.

El problema surge cuando el nuevo registro ocupa mayor espacio que el anterior. En este caso, no es posible utilizar el mismo espacio físico, y el registro necesita ser reubicado.

En general, para evitar todo este análisis y para facilitar el algoritmo de modificación sobre archivos con registros de longitud variable, 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.

Cuestionario del capítulo

1. ¿Por qué el proceso de baja de información de un archivo fue cambiando a lo largo del tiempo?
2. ¿Cuáles son las diferencias sustanciales entre el proceso de baja lógica y baja física?
3. ¿Cuáles son las ventajas que plantea la reutilización de espacio ligada con la baja de información en un archivo?
4. ¿Qué ventajas presenta la utilización de registros de longitud variable?
5. ¿Por qué el proceso de recuperación de espacio, utilizado en archivos con registros de longitud variable, difiere del mismo proceso con archivos que contengan registros de longitud fija?
6. ¿Cuáles son las principales características que llevarían a utilizar primer ajuste en vez de peor ajuste, como método de recuperación de espacios?
7. ¿Qué motiva a tener un proceso más complejo de modificación de datos, cuando se trabaja con registros de longitud variable?

Ejercitación

1. Utilizando el archivo generado en el Ejercicio 1 del Capítulo 3, genere un algoritmo que permita borrar los promotores que tengan ventas por debajo de $15000.
   a) Borrando los datos de manera física.
   b) Borrando los datos de manera lógica.
2. Realice los cambios necesarios a los algoritmos del Ejercicio 1, para poder determinar el tiempo de ejecución necesario en cada uno de ellos. Para tal fin, el archivo de datos inicial debe ser el mismo y contar con un mínimo de 50.000 registros.
3. Defina un programa que genere un archivo que contenga los datos personales de los docentes de la facultad, utilizando para ello registros de longitud fija.
4. Implante un algoritmo que, a partir del archivo de datos generado en el ejercicio anterior, elimine de forma lógica los docentes de más de 65 años.
5. Genere todas las modificaciones necesarias sobre el Ejercicio 4 de manera de poder implementar el proceso de altas reutilizando espacio. Para ello, deberá utilizar las políticas de:
   a) Primer ajuste b) Mejor ajuste c) Peor ajuste

Capítulo 5

Búsqueda de información. Manejo de índices

Objetivo

En los capítulos previos, se definieron las operaciones básicas sobre archivos. Sin embargo, dichas operaciones se analizaron desde el punto de vista algorítmico y no con una visión de BD (motivación fundamental de este libro). Con esta motivación, el propósito de este capítulo es trabajar sobre la búsqueda de información en archivos, con especial atención en la *performance*, dividiendo el problema en dos grandes grupos: el primero de ellos realiza la búsqueda sobre archivos directamente, mientras que en el segundo caso se plantea el uso de estructuras de datos auxiliares que sirven de soporte a la mejora en el acceso.

Proceso de búsqueda

Cuando se realiza la búsqueda de un dato, se debe considerar la cantidad de accesos a disco en pos de encontrar esa información, y en cada acceso, la verificación de si el dato obtenido es el buscado (comparación). Es así que surgen dos parámetros a analizar: cantidad de accesos y cantidad de comparaciones. El primero de ellos es una operación sobre memoria secundaria, lo que implica un costo relativamente alto; mientras que el segundo es sobre memoria principal, por lo que el costo es relativamente bajo. A modo de ejemplo, si un acceso a memoria principal representa 5 segundos, un acceso a memoria secundaria representaría 29 días. Por lo tanto, se tomará en cuenta solo el costo de acceso a memoria secundaria para los análisis de *performance* tanto de este capítulo como de los subsiguientes.

El proceso de búsqueda implica un análisis de situaciones en función del tipo de archivo sobre el que se quiere buscar información.

El caso más simple consiste en disponer de un archivo serie, es decir, sin ningún orden preestablecido más que el físico, donde, para acceder a un registro determinado, se deben visitar todos los registros previos en el orden en que estos fueron almacenados. Aquí, la búsqueda de un dato específico se detiene en el registro que contiene ese dato (previo acceso a todos los registros anteriores), o en el final del archivo si el resultado no es exitoso (el dato no se encuentra). Por lo tanto, el mejor caso es ubicar el dato deseado en el primer registro (1 lectura), y el peor caso, en el último registro (*N* lecturas, siendo *N* la cantidad de registros). El caso promedio consiste entonces en realizar *N*/2 lecturas. Es decir, la *performance* depende de la cantidad de registros del archivo, siendo entonces de orden *N*.

Mejor caso → 1 acceso
Peor caso → *N* accesos

Promedio → *N*/2 accesos

Si el archivo está físicamente ordenado y el argumento de búsqueda coincide con el criterio de ordenación, la variación en relación con el caso anterior se produce para el caso de que el dato buscado no se encuentre en dicho archivo. En ese caso, el proceso de búsqueda se detiene en el registro cuyo dato es "mayor" buscado. De ese modo, en ese caso no existe la necesidad de recorrer todo el archivo. No obstante, la *performance* sigue siendo de orden *N*, y a la estrategia de búsqueda utilizada en ambos casos se la denomina secuencial.

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. En este caso, la estrategia se ve alterada. 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 el ejemplo siguiente, se busca el elemento 15; como el archivo contiene 20 registros, el primer Número Relativo de Registro (NRR) accedido es el registro del décimo lugar.

1	3	4	5	8	11	15	18	21	25	29	30	40	43	45	56	67	87	88	90

Como en esa posición está el elemento 25, los registros posteriores al 10 se descartan.

1	3	4	5	8	11	15	18	21	25	29	30	40	43	45	56	67	87	88	90

Se calcula el nuevo punto medio, que será la posición 4 del archivo.

1	3	4	5	8	11	15	18	21	25	29	30	40	43	45	56	67	87	88	90

Como en esta posición tampoco está el elemento buscado, se descartan los menores al 5.

1	3	4	5	8	11	15	18	21	25	29	30	40	43	45	56	67	87	88	90

El proceso calcula nuevamente el punto medio y localiza el elemento buscado.

1	3	4	5	8	11	15	18	21	25	29	30	40	43	45	56	67	87	88	90

Este criterio de búsqueda, donde la mitad de los registros restantes se descartan en cada comparación, se denomina búsqueda binaria. El Ejemplo 5.1 muestra el seudocódigo de dicha búsqueda.

EJEMPLO 5.1

```
Function Busqueda_Binaria (archivo, clave, long_archivo)
Variables
menor, mayor, clave_encontrada, registro: integer

registro   := -1
menor      := 1
mayor      := long_archivo

mientras ( menor <= mayor )
  medio :=[(mayor+menor )/2]
  buscar reg con NRR = medio
  clave_encontrada = clave canónica correspondiente al
  reg. leído
  si clave < clave_encontrada
    entonces
      mayor := medio+1
    sino
      si clave > clave_encontrada
         entonces
             menor := medio + 1
      sino
             registro:= NRR

Busqueda_Binaria:= registro

fin
```

En general, una búsqueda binaria en un archivo con *N* registros se realiza en a lo sumo $log_2(N)+1$ comparaciones y en promedio $(log_2(N)+1)/2$ comparaciones. Por lo tanto, se concluye que la búsqueda binaria es de orden $log_2(N)$, 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.

La búsqueda de un registro particular en un archivo que contiene registros de longitud fija puede optimizarse, al extremo de lograr ubicar dicho registro con solo un acceso. Esto sucede porque dicho archivo tiene los registros numerados de 0 a *N*, es decir, 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.

Así, para lograr ubicar un registro particular, basta con disponer del NRR del registro y acceder directamente al registro deseado.

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.

Retomando la búsqueda (secuencial o binaria) en archivos físicamente ordenados, como es necesario garantizar el orden del archivo para poder llevarla a cabo, se definen a continuación distintas alternativas para ordenar físicamente un archivo.

Ordenamiento de archivos

Dada la conveniencia de que un archivo se encuentre ordenado, para lograr mejor desempeño o *performance* en la búsqueda de datos, es necesario discutir alternativas de ordenación física para los archivos de datos.

Si se aplicara cualquier algoritmo de ordenación de vectores, por ejemplo, para realizar la ordenación de un archivo, habría que tener en cuenta que se debe leer y escribir varias veces cada elemento, con el consiguiente desplazamiento desde la posición inicial hasta la posición final en que cada uno quede ubicado. Estas operaciones realizadas en memoria secundaria serían excesivamente lentas.

Supóngase que se dispone de un archivo con 1.000 elementos y que el método de ordenación utilizado es el de *N* pasadas. Este método recorre el archivo buscando el elemento menor y lo posiciona en el

primer lugar, luego repite la misma operación buscando el segundo menor y así sucesivamente hasta ordenar todo el archivo. En este caso, serán necesarias 1.000 lecturas para determinar el menor elemento, y 2 escrituras que intercambien el primer elemento menor con el elemento que está en la posición 1 del archivo (1.002 accesos). El proceso se repite, ahora con 999 lecturas, para el segundo elemento con sus 2 escrituras (1.001 accesos). Esto significa que la cantidad de operaciones requeridas será:

1.002 + 1.001 + 1.000 +(…)+ 4 operaciones

Supóngase ahora que el archivo contiene 1.000.000 de registros. El número final de accesos es excesivamente elevado.

Si el archivo a ordenar puede almacenarse en forma completa en memoria RAM, una alternativa muy atrayente es trasladar el archivo completo desde memoria secundaria hasta memoria principal, y luego ordenarlo allí. Si bien esto implica leer los datos de memoria secundaria, su acceso es secuencial sin requerir mayores desplazamientos, por lo que su costo no es excesivo. Luego, la ordenación efectuada en memoria principal será realizada con alta *performance*. La operatoria finaliza escribiendo nuevamente el archivo ordenado en memoria secundaria, otra vez en forma consecutiva, con pocos desplazamientos de la cabeza lectograbadora del disco.

Esta posibilidad constituye la mejor alternativa en cuanto a *performance*, para ordenar físicamente un archivo, pero solamente puede ser utilizada para archivos pequeños.

Si el archivo no cabe en memoria RAM, una segunda alternativa constituye transferir a memoria principal de cada registro del archivo solo la clave por la que se desea ordenar, junto con los NRRs, a los registros correspondientes en memoria secundaria. De ese modo, al transferir solo esos datos, es posible almacenar mayor cantidad de registros en RAM y, por lo tanto, ordenar un archivo de mayor cantidad de datos. Además, es una muy buena opción, ¿para qué transferir cada registro en forma completa si finalmente se considerará la clave como argumento para hacer la ordenación?

El algoritmo ordena en memoria principal solo las claves. Posteriormente, se debe leer nuevamente cada registro del archivo (de acuerdo con el orden establecido por las claves) y escribirlo sobre el archivo ordenado. De este modo, el proceso requiere leer el archivo en forma completa dos veces y reescribirse ordenado una vez. Esto implica muchos desplazamientos en memoria secundaria. Por lo tanto, es alto el costo a pagar para poder utilizar la memoria principal como medio de ordenación, además de la restricción de que todas las claves más los NRRs deben caber en dicha memoria, por lo que si el archivo es

realmente grande, esta alternativa se ve imposibilitada. No obstante, la idea de reorganizar solo las claves es interesante y será tratada nuevamente en este capítulo.

La tercera alternativa surge cuando el archivo es realmente grande, de modo tal que las claves no se pueden ordenar en memoria principal porque no caben.

Es así que esta opción implica plantear otra estrategia para ordenar el archivo, obviamente descartando la posibilidad de hacerlo directamente sobre memoria secundaria, por los costos ya discutidos. 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 (estos tres pasos anteriores se denominan *sort* interno).
4. 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. Esta operatoria ya fue anteriormente descripta en el Capítulo 3.

A modo de ejemplo, supóngase que se dispone de un archivo de 800.000 registros y 1 Mb de memoria principal (si bien es poca la capacidad de memoria definida, se deben tomar los datos a modo de ejemplo solamente). Cada registro ocupa 100 bytes, por lo que la longitud total del archivo es 80.000.000 bytes, es decir, aproximadamente 80 Mb (el archivo no se puede transferir a memoria principal).

La clave de cada registro ocupa 10 bytes, por lo que para albergar todas las claves se necesitan 8 Mb (tampoco se pueden transferir a memoria principal). Es así que la única opción posible es generar particiones del tamaño de memoria principal disponible, 1 Mb / 100 bytes por registro = 10.000 registros forman una partición, y al disponer de 800.000 registros se originarán 80 particiones.

Para estos datos, se analiza el desplazamiento necesario para realizar el *merge* o fusión, considerando solamente las operaciones de lectura (faltaría realizar los cálculos para la escritura).

Dado que cada una de las 80 particiones es del tamaño de la memoria principal disponible (puesto que así fue definido en su conformación), al realizar el *merge* simultáneo de las 80 particiones, solo es posible asignar 1/80 parte de la memoria a cada partición. En consecuencia,

es preciso desplazarse 80 veces por cada una de ellas para leerla en forma completa, y como hay 80 particiones para completar la operación de *merge*, se tienen que realizar:

80 particiones * 80 desplazamientos por porción =
6.400 desplazamientos

Ya se ha mencionado el alto costo de realizar desplazamientos en memoria secundaria, por lo que 6.400 desplazamientos resulta una cantidad que necesita ser reducida mediante algún método alternativo.

Generalizando, para realizar el *merge* de *K* formas de *K* porciones, donde cada partición es del tamaño disponible de memoria principal, se requieren *k* desplazamientos para leer todos los registros en cada partición individual. Además, dado que *k* porciones, la operación de *merge* requiere k^2 desplazamientos. Por lo tanto, la ordenación evaluada en términos de desplazamientos tiene una *performance* del orden k^2, y como *k* es directamente proporcional a la cantidad total (*N*) de registros del archivo, se puede concluir que la ordenación es una operación de $O(N^2)$ evaluada en términos de desplazamiento.

A continuación, se analizan situaciones alternativas al *sort* interno, que posibiliten incrementar el tamaño de cada partición, sin requerir memoria principal adicional. La *performance* del *merge* está ligada de manera inversa a la cantidad de particiones generadas. A mayor número de particiones, menor la *performance* final. Por consiguiente, si se logra reducir el número de particiones, la *performance* debe mejorar.

Selección por reemplazo

Si pudiera incrementarse de algún modo el tamaño de las particiones, decrecería la cantidad de trabajo requerida durante el *merge* para el proceso de ordenar un archivo. Particiones más grandes implican menor cantidad total de particiones, y al disponer de menos particiones, se necesitan menos desplazamientos en memoria secundaria para realizar las lecturas necesarias. La pregunta, en este punto, sería: ¿cómo crear particiones que sean por ejemplo dos veces más grandes que la cantidad de registros que se pueden almacenar en memoria principal? La respuesta implica suplir esta necesidad con una nueva estrategia que se utiliza con un algoritmo conocido como **selección por reemplazo**.

Este método se basa en el siguiente concepto: seleccionar siempre de memoria principal la clave menor, enviarla a memoria secundaria y reemplazarla por una nueva clave que está esperando ingresar a memoria principal.

Sintetizando, los pasos a cumplir serían:

1. Leer desde memoria secundaria tantos registros como quepan en memoria principal.
2. Iniciar una nueva partición.
3. Seleccionar, de los registros disponibles en memoria principal, el registro cuya clave es menor.
4. Transferir el registro elegido a una partición en memoria secundaria.
5. Reemplazar el registro elegido por otro leído desde memoria secundaria. Si la clave de este registro es menor que la clave del registro recientemente transferido a memoria secundaria, este nuevo registro se lo guarda como no disponible.
6. Repetir desde el Paso 3 hasta que todos los registros en memoria principal estén no disponibles.
7. Iniciar una nueva partición activando todos los registros no disponibles en memoria principal y repetir desde el Paso 3 hasta agotar los elementos del archivo a ordenar.

El Ejemplo 5.2 ilustra cómo funciona este método. Durante la primera partición, cuando las claves son muy pequeñas para incluirse allí, se las marca entre paréntesis, es decir, se las "duerme", indicando que formarán parte de la siguiente partición. Cabe destacar que el tamaño de memoria principal admite 3 claves, mientras que el tamaño de la primera partición quedó de 6 claves y el de la segunda partición, de 7 claves.

EJEMPLO 5.2

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

continúa >>>

Primera partición completa; se inicia la construcción de la siguiente:

Resto de la entrada	Mem. principal	Partición generada
34, 19, 25, 59	(15) (18) (8)	–
34, 19, 25	15 18 59	8
34, 19	25 18 59	15, 8
34	25 19 59	18, 15, 8
	25 34 59	19, 18, 15, 8
	– 34 59	25, 19, 18, 15, 8
	– – 59	34, 25, 19, 18, 15, 8
	– – –	59, 34, 25, 19, 18, 15, 8

Generalizando, se puede establecer que, en promedio, este método aumenta el tamaño de las particiones al doble de la cantidad de registros que se podrían almacenar en memoria principal (2P). Esto ha sido demostrado ya en la década de 1970. Además, si los datos de entrada a memoria principal están (aunque sea) parcialmente ordenados, el tamaño promedio de las particiones generadas es superior al doble (2P).

Selección natural

El método de selección natural en relación con selección por reemplazo es una variante 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).

Comparando los tres métodos analizados:

- **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 de registro, 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, necesariamente, de igual tamaño.

Merge en más de un paso

En pos de buscar mejoras aun mayores que las alternativas ya vistas, se plantea la posibilidad de realizar el *merge* o fusión en más de un paso. Esto requiere leer 2 veces cada registro; no obstante, se analiza a modo de ejemplo la situación planteada anteriormente (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.

El primer paso consiste en tomar 8 particiones, asignando un *buffer* a cada una. Estos 8 *buffers* de entrada pueden almacenar 1/8 de cada partición, requiriendo 8 desplazamientos cada uno para recuperar una partición, y como se dispone de 8 particiones, serán necesarios 64 desplazamientos en memoria secundaria para generar el archivo intermedio (8 desplazamientos * 8 particiones).

Este proceso debe repetirse 9 veces más para generar los 10 archivos intermedios; entonces, se deberán realizar 10 * 64 = 640 desplazamientos.

Además, cada una de las 10 particiones es 8 veces más grande; esto implica 80 desplazamientos más por partición, y como son 10 particiones, 10 * 80 = 800 desplazamientos más. En total, se requieren 640 + 800 = 1.440 desplazamientos.

Si comparamos los 6.400 desplazamientos iniciales que eran requeridos en el *merge* en un paso, la mejora es evidente.

Si bien existen otros métodos de *merge* (balanceado de *N* caminos, óptimo, *multiface*, entre otros), no serán analizados en este libro.

Indización

Se ha planteado que el propósito de ordenar un archivo radica en tratar de minimizar los accesos a memoria secundaria durante la búsqueda de información. Sin embargo, la *performance* es del orden $log_2(N)$, y puede ser mejorada.

Supóngase el problema de buscar un tema en un libro; independientemente de si este estuviera ordenado por temas, la acción natural sería buscar el material deseado en el índice temático del libro, y luego, acceder directamente a la página que se incluya en dicho índice. Es de notar que en este caso se busca la información en una fuente de datos adicional (el índice), que es de tamaño considerablemente menor, para luego acceder directamente a dicha información.

Analizando este ejemplo, puede intuirse que, con el uso de una fuente de información organizada adicional, se puede lograr mejorar la *performance* de acceso a la información deseada. Esta idea conduce a pensar en realizar algo similar con el acceso a la información almacenada en archivos, es decir, utilizar una estructura adicional que permita mejorar la *performance* de acceso al archivo.

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.

De este modo, con el razonamiento previo, el índice es un nuevo archivo ordenado, con registros de longitud fija, con la diferencia de que contiene solo un par de datos del archivo original, y, por ende, es mucho más pequeño. En el peor caso, se plantea de nuevo la situación de ordenar este nuevo archivo en memoria secundaria, con alguno de los criterios planteados previamente en este capítulo. Una vez disponible el índice, la búsqueda de un dato se realiza primero en el índice (a partir de la clave), de allí se obtiene la dirección efectiva del archivo de datos y luego se accede directamente a la información buscada.

Quedan por analizar, entonces, la generación y el mantenimiento de esta nueva estructura. En primer lugar, se tratará el caso del índice creado a partir de la clave primaria, denominado índice primario.

A modo de ejemplo, se plantea el siguiente archivo de datos, organizado con registros de longitud variable:

Dir. Reg.	Cía.	Código	Título	Grupo musical compositor	Intérprete
15	WAR	23	Early Morning	A-ha	A-ha
36	SON	13	Just a Like a Corner	Cock Robin	Cock Robin
83	BMG	11	Selva	La Portuaria	La Portuaria
118	SON	15	Take on Me	A-ha	A-ha
161	VIR	2310	Land of Confusion	Genesis	Phil Collins
209	VIR	1323	Summer Moved On	A-ha	A-ha
248	AME	2323	Africa	Toto	Toto
275	RCA	1313	Leave of New York	REM	REM
313	ARI	2313	Here Come The Rain Again	Eurythmics	Annie Lennox

Al crearse el archivo, con clave primaria formada por los atributos Cía + Código , se crea el índice primario asociado. El índice y el archivo quedarían de la siguiente manera:

Clave	Ref.	Dir. reg.	Registro de datos
AME2323	248	15	WAR | 23 | Early Morning | A-ha | A-ha
ARI2313	313	36	SON | 13 | Just a Like a Corner | Cock Robin | Cock Robin
BMG11	83	83	BMG | 11 | Selva | La Portuaria | La Portuaria
RCA1313	275	118	SON | 15 | Take on Me | A-ha | A-ha
SON13	36	161	VIR | 2310 | Land of Confusion | Genesis | Phil Collins
SON15	118	209	VIR | 1323 | Summer Moved On | A-ha | A-ha
VIR1323	209	248	AME | 2323 | Africa | Toto | Toto
VIR2310	161	275	RCA | 1313 | Leave of New York | REM | REM
WAR23	15	313	ARI | 2313 | Here Come The Rain Again | Eurythmics | Annie Lennox

La tabla de la izquierda presenta el índice generado ordenado por la clave primaria del archivo de datos, en tanto que la tabla de la derecha muestra el archivo de datos propiamente dicho, utilizando registros de longitud variable. Es de notar que el archivo de datos no está ordenado.

Para realizar cualquier búsqueda de datos, por ejemplo, si se desea buscar el compositor del tema “Here Come The Rain Again”, en primer lugar se busca la clave primaria (ARI2313) en el índice. Dado que el índice es un archivo de registros con longitud fija, es posible realizar sobre este una búsqueda binaria. Si además el índice cabe en memoria principal, puede ser transferido hacia allí y luego efectuarse la búsqueda.

Una vez hallada la referencia (313), correspondiente a la dirección del primer byte del registro buscado, se accede directamente al archivo de datos según lo indicado en dicha referencia.

Resumiendo, 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. La mejor estructura y organización para los índices será un tema analizado en los capítulos siguientes.

Retomando la generación y el mantenimiento del índice, se analizarán brevemente estas operaciones.

Creación de índice primario

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

Altas en índice primario

La operación de alta de un nuevo registro al archivo de datos consiste simplemente en agregar dicho registro al final del archivo. A partir de esta operación, con el NRR o la dirección del primer byte, según corresponda, más la clave primaria, se genera un nuevo registro de datos a insertar en forma ordenada en el índice. En caso de que el índice se encuentre en memoria principal, la inserción implica un costo relativamente bajo. Si se encuentra en memoria secundaria, el costo es mayor.

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

En este caso, 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

La principal ventaja que posee el uso de un índice primario radica en que, al ser de menor tamaño que el archivo asociado y tener registros de longitud fija, posibilita mejorar la *performance* de búsqueda. Además, se pueden realizar búsquedas binarias, mientras que en el archivo original esto quedaba condicionado a que contuviera registros de longitud fija. Más adelante, en los siguientes capítulos, se analizará la utilización de estructuras de datos para índices, más eficientes en términos de búsqueda, que permitan mejorar la *performance* de una búsqueda binaria.

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. El tratamiento de un índice que soporte una clave candidata es similar al definido anteriormente para un índice primario. En los próximos capítulos, se ejemplificarán con más detalle el uso, la organización y las ventajas de estos índices.

Índices secundarios

La pregunta que seguramente surgió al buscar el compositor de la canción “Here Come The Rain Again” por la clave primaria (ARI2313) es cómo saber este dato, la clave primaria. No es natural ni intuitivo solicitar un dato por clave primaria, sino por el nombre de la canción o eventualmente por autor, que son atributos mucho más fáciles de recordar. Estos atributos, nombre de canción o autor, podrían contener valores repetidos en el archivo original. Por este motivo, no es posible pensarlos como parte de una clave primaria. La clave que soporta valores repetidos se denomina clave secundaria.

Por lo tanto, es necesario crear otro tipo de índice mediante el cual se pueda acceder a la información de un archivo, pero con datos fáciles de recordar. De esta manera surge el uso de índices secundarios.

Un índice secundario es una estructura adicional que permite relacionar una clave secundaria con una o más claves primarias, dado que, como ya se ha mencionado previamente, varios registros pueden contener la misma clave secundaria. Luego, 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. El lector podría preguntarse por qué motivo el índice secundario no posee directamente la dirección física de elemento de dato. La respuesta es simple: 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. Suponga que para un archivo cualquiera se tienen definido un índice primario y cuatro secundarios. Si se modifica la posición física de un registro en el archivo de datos, los cinco índices deberían modificarse. Para mejorar esta situación, solo el índice primario tiene la dirección física y es el único que debe alterarse; todos los índices secundarios permanecen sin cambios. Solamente deberían ser

modificados si la clave primaria fuera alterada, situación que virtualmente no ocurre. Este tema será tratado en el Capítulo 12. Continuando con los datos del ejemplo antes presentado, en la siguiente tabla se muestra un índice secundario (índice de grupos de música).

Clave secundaria	Clave primaria
A-ha	SON15
A-ha	VIR1323
A-ha	WAR23
Cock Robin	SON13
Eurythmics	ARI2313
Genesis	VIR2310
La Portuaria	BMG11
REM	RCA1313
Toto	AME2323

En el ejemplo puede verse que, cuando la clave se repite, existe un segundo criterio de ordenación, establecido por la clave primaria.

Si se desea buscar un tema del grupo REM, primero se busca la clave REM en el índice secundario, se obtiene allí la clave primaria RCA1313 y finalmente se accede al índice primario para obtener luego la dirección física del archivo (275).

Nuevamente, el índice secundario está almacenado en un archivo con registros de longitud fija, y es de notar que es posible tener más de un índice secundario por archivo. Se deben analizar las operaciones básicas sobre el archivo que contiene al índice secundario: crear, agregar, modificar y eliminar.

Creación de índice secundario

Al implantarse el archivo de datos, se deben crear todos los índices secundarios asociados, naturalmente vacíos, solo con el registro encabezado. Esta operación es similar a la definida para índices primarios.

Altas en índice secundario

Cualquier alta en el archivo de datos genera una inserción en el índice secundario, que implica reacomodar el archivo en el cual se almacena. Esta operación es de bajo costo en términos de *performance* si el índice puede almacenarse en memoria principal, pero resulta muy

costosa si está en memoria secundaria, considerando además que potencialmente puede existir (como en el ejemplo visto) un segundo nivel de ordenación, determinado por la clave primaria.

Modificaciones en índice secundario

Aquí se deben analizar dos alternativas. La primera de ellas ocurre cuando se produce un cambio en la clave secundaria. En este caso, se debe reacomodar el índice secundario, con los costos que ello implica. El segundo caso ocurre cuando cambia el resto del registro de datos (excepto la clave primaria), no generando así ningún cambio en el índice secundario.

Bajas en índice secundario

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.

La eliminación a realizarse en el índice secundario, almacenado en un archivo con registros de longitud fija, puede ser física o lógica.

Una alternativa interesante es borrar solo la referencia en el índice primario. En este caso, el índice primario actúa como una especie de barrera de protección, que aísla a los índices secundarios de este tipo de cambios en el archivo de datos. El beneficio principal es que el índice secundario no se altera, pero existe un costo adicional representado en el espacio ocupado de datos que ya no existen. Si el archivo de datos fuese poco volátil, este no sería un gran problema, pero si fuera volátil, se deberían programar borrados físicos de los índices secundarios.

Alternativas de organización de índices secundarios

Se ha visto que el índice secundario debe reacomodarse con cada alta realizada sobre el archivo de datos, aun si se ingresa una clave secundaria ya existente. Esto se debe a que existe un segundo nivel de ordenación por la clave primaria asociada al registro ingresado.

La organización de índices presentada hasta el momento consiste en almacenar la misma clave secundaria en distintos registros, tantas como ocurrencias haya. Esto implica mayor espacio, generando una menor posibilidad de que el índice quepa en memoria.

La primera alternativa de mejora sobre el espacio ocupado por el índice secundario implica almacenar en un mismo registro todas las ocurrencias de la misma clave secundaria. Con el ejemplo presentado previamente, esta nueva organización consiste en que cada registro esté formado por la clave secundaria, más un arreglo de claves primarias correspondientes a dicha clave. En el ejemplo presentado:

A-ha	SON15	VIR1323	WAR23

En este caso, al agregar un nuevo registro al archivo de datos con la misma clave secundaria, no genera una inserción en el índice secundario. Solo se debe insertar en forma ordenada la clave primaria en el vector de claves primarias respectivo.

El problema inherente a esta alternativa es la elección del tamaño del registro, ya que conviene que este sea de longitud fija. Esto implica decidir el tamaño del vector, por lo que puede haber casos en que resulte insuficiente o que sobre espacio (provocando fragmentación). Por lo tanto, si bien es una alternativa razonable, la solución a un inconveniente produce potencialmente otro problema.

Otra alternativa es pensar en una lista de claves primarias asociada a cada clave secundaria. De esta manera, no se debe realizar reserva de espacio y puede existir cualquier número de claves primarias por cada clave secundaria.

Con el ejemplo que se ha presentado, el índice secundario quedaría organizado del siguiente modo:

NRR	Clave secundaria	Puntero	NRR	Clave primaria	Enlace
0	A-ha	2	0	VIR2310	–1
1	Cock Robin	3	1	BMG11	–1
2	Eurythmics	4	2	SON15	5
3	Genesis	0	3	SON13	–1
4	La Portuaria	1	4	ARI2313	–1
5	REM	6	5	VIR1323	8
6	Toto	7	6	RCA1313	–1
			7	AME2323	–1
			8	WAR23	–1

Analizando brevemente el ejemplo, se puede observar que, para acceder a las claves primarias asociadas a las secundarias, primeramente se accede al archivo que contiene solo las claves secundarias, y allí se obtiene la dirección (NRR) de acceso a la lista de claves primarias asociadas. Dicha lista, denominada lista invertida, se encuentra almacenada en otro archivo, el cual se recorre de acuerdo con el camino establecido por el atributo enlace, que indica cuál es el próximo registro asociado, que en caso de no existir, contendrá un puntero nulo (enlace = –1, en el ejemplo).

Ambos archivos están organizados con registros de longitud fija.

Esta opción tiene las siguientes ventajas:

- El único reacomodamiento en el índice se produce cuando se agrega una nueva clave primaria. Igualmente el índice es más pequeño, por lo que dicho reacomodamiento resulta menos costoso.
- Si se agregan o borran datos de una clave secundaria ya existente, solo se debe modificar el archivo que contiene la lista, y en cada caso solo se debe modificar una lista.
- Dado que se generan dos archivos, uno de ellos podría residir en memoria secundaria, liberando así memoria principal. No obstante, si existiesen muchos índices secundarios, el mantenimiento de dos archivos por cada índice podría resultar muy costoso.

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.

De esta forma, el índice incluye solo un subconjunto de datos asociado a los registros del archivo que interesa consultar, generando una estructura que actúa como filtro de acceso a la información de interés en el archivo.

Obviamente, en este caso, solo se deben considerar las modificaciones al archivo, vinculadas con los datos presentes en el índice.

Cuestionario del capítulo

1. ¿Por qué tiene tanta importancia la operación de búsqueda de información?
2. ¿Qué motiva a analizar solo la cantidad de accesos a disco cuando se habla de análisis de *performance* de operaciones de búsqueda?
3. ¿Cuál es el costo de tener un archivo físicamente ordenado?
4. ¿Qué es un índice? ¿Para qué sirve?
5. ¿Qué diferencias existen entre índices primarios, candidatos y secundarios?

Ejercitación

1. Suponga que tiene un archivo de datos desordenado con 1.600.000 registros de 200 bytes cada uno y una clave primaria que ocupa 20 bytes, y solo dispone de 4 Mb de memoria. Discuta cómo podría ordenar este archivo.
2. Sobre el ejercicio anterior se podrían haber generado particiones como parte del proceso de ordenación.
   a. ¿Cuántas particiones utilizando sort interno se pudieron generar?
   b. Discuta el costo de ordenar las particiones con un proceso clásico de merge que utilice todos los archivos intermedios generados simultáneamente.
   c. Discuta nuevamente el inciso anterior, proponiendo una intercalación en dos pasos.
   d. Resuelva el caso anterior con una intercalación en tres pasos.
   e. Compare los resultados obtenidos en los incisos b, c y d. ¿Qué puede concluir al respecto?

Capítulo 6

Árboles. Introducción

Objetivo

En el Capítulo 5, se discutió sobre la necesidad de localizar rápidamente alguna información particular dentro de un archivo. Se debe tener en cuenta que, sobre una BD, aproximadamente 80% de las operaciones efectuadas consisten en búsqueda de información. Por lo tanto, es de gran importancia poder recuperar un dato requerido con la menor cantidad de accesos sobre disco. Es así que la organización física que se determine para el archivo resultará de suma importancia para la *performance* final de cada una de las operaciones de consulta.

Se discute en este capítulo la utilización de árboles binarios de búsqueda, como forma de organizar los datos en un archivo sobre un dispositivo de almacenamiento secundario, omitiéndose los conceptos básicos de la estructura árbol. Se presentan las ventajas y desventajas de esta política y, asimismo, se plantean los algoritmos que permiten implantar árboles binarios sobre disco.

Árboles binarios

El principal problema para poder administrar un índice en disco rígido lo representa la cantidad de accesos necesarios para recuperar la información. Como se discutió en el Capítulo 5, el proceso de búsqueda de información es costoso y, además, el mantenimiento de la información ante operaciones de altas, bajas y modificaciones también debe ser considerado como parte del proceso de actualización del archivo de datos.

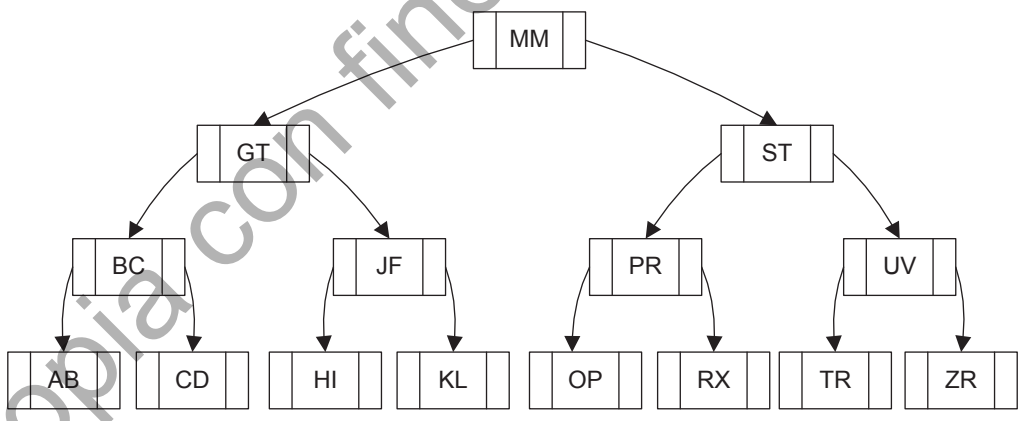
Las estructuras tipo árbol presentan algunas mejoras tanto para la búsqueda como para el mantenimiento del orden de la información. Esta aseveración será discutida a lo largo de este capítulo.

Los árboles binarios representan la alternativa más simple para la solución de este problema. Es probable que el lector haya estudiado temas relacionados con árboles binarios, e implantado soluciones recursivas o iterativas utilizando estructuras de datos dinámicas sobre memoria RAM. Se presentarán aquí las variantes que deben llevarse a cabo para que los algoritmos tradicionales puedan implantarse sobre estructuras que permitan almacenar la información en dispositivos secundarios.

> 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. En ese caso, como lo muestra la Figura 6.1, los elementos que se agregan en un árbol se insertan manteniendo un orden. Esto es, a la izquierda de un elemento se encuentran los elementos menores que él, y a la derecha, los mayores.

Figura 6.1



La búsqueda en este tipo de estructuras se realiza a partir del nodo raíz y se recorre, explorando hacia los nodos hoja. Se chequea un nodo; si es el deseado, la búsqueda finaliza o, en su defecto, se decide si la búsqueda continúa a izquierda o a derecha descartando la mitad de los elementos restantes. Por este motivo, la búsqueda de información en un árbol binario es de orden logarítmico. Encontrar un elemento es del orden log_2 (N), siendo N la cantidad de elementos distribuidos en el árbol.

Es probable que el lector haya implementado algoritmos de árboles binarios sobre memoria RAM. Sin embargo, para poder utilizar estas ideas como soporte de índices de búsqueda, es necesario que los árboles binarios se implanten sobre almacenamiento secundario.

La Figura 6.2 presenta la definición de datos necesaria para generar un archivo de índices utilizando árboles binarios sobre disco. El lector puede observar que el registro de datos contiene tres campos. El campo `elemento_de_dato` contendrá la clave del índice, los campos `hijo_izquierda e hijo_derecha` definen la dirección del hijo menor y mayor, respectivamente. Mientras que sobre memoria RAM estos hijos se representaban mediante punteros, ahora el valor de cada uno es representado mediante un valor entero. Ese valor indica el NRR del hijo dentro del archivo de datos.

Figura 6.2

```
type   registroarbolbinario = record
              elemento_de_dato:  tipo_de_dato;
              hijo_izquierda,
              hijo_derecha    :  integer;
       end;

       indicebinario = file of registroarbolbinario;
```

La Figura 6.3 representa un formato posible del archivo de datos que se corresponde con el árbol presentado en la Figura 6.1. Es posible observar que cada elemento tiene la dirección de sus respectivos hijos y, en caso de tratarse de un nodo terminal u hoja, con un valor negativo se indica que no dispone de hijos.

Figura 6.3

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

El Ejemplo 6.1 muestra el seudocódigo del algoritmo que permite la creación/inserción de nuevos elementos de datos sobre un archivo que almacena un índice estructurado en un árbol binario.

EJEMPLO 6.1

```
Type
  Nodo = record
    Elemento: tipo_dato_elemento;
    Hijo_Izquierdo: integer;
    Hijo_Derecho:integer;
End;
Archivo: file of Nodo;

Procedure Insertar (var A: Archivo, elem:
                    tipo_dato_elemento);
Var
  Raiz, nodo_nuevo: Nodo;
  Pos_nuevo_nodo:   integer;
  Encontre_Padre:   boolean;

Begin
Reset(A);
With nodo_nuevo do
  Elemento   :=elem;
  Hijo_izquierdo    := -1;
  Hijo_Derecho      := -1;
End;

If Eof(A) Then {significa que es un árbol vacío y el
elemento es insertado como raíz}
  Write(A,nodo_nuevo);
Else
  Read(A, Raiz);
  Pos_nuevo_nodo := filesize(A);
  Seek(A, pos_nuevo_nodo); {posicionarse al final del
  archivo}
  Write(A, nodo_nuevo); {escribir el nuevo nodo al final}
  Encontre_Padre := false;

  {buscar al padre para agregar la referencia al nuevo
  nodo}
  While not (Encontre_Padre) do
  begin
      If (Raiz.elemento > nodo_nuevo.elemento) Then
        If (Raiz.hijo_izquierdo <> -1) Then
           Seek(A, Raiz.hijo_izquierdo);
           Read(A, Raiz);
        Else
           Raiz.hijo_izquierdo := Pos_nuevo_nodo;
           Encontre Padre:= true;
```

continúa >>>

```
        Else
          If (Raiz.hijo_derecho <> —1) Then
             Seek(A, Raiz.hijo_derecho);
             Read(A, Raiz);
          Else
             Raiz.hijo_derecho := Pos_nuevo_nodo;
             Encontre_Padre := true;
          end;
        end;
    end;
    {raíz es el padre y ya lo leí, debo volver a
    posicionarme}
    Seek(A, Filepos(A)—1);

    {guardo al padre con la nueva referencia}
    Write (A, raiz);
end.
```

Performance de los árboles binarios

El Ejemplo 6.2 presenta el proceso que permite buscar dentro del archivo anterior (que representa un árbol binario) un elemento particular.

EJEMPLO 6.2

```
Type
  Nodo = record
      Elemento: tipo_dato_elemento;
      Hijo_Izquierdo: integer;
      Hijo_Derecho:integer;
End;
Archivo : file of Nodo;

Function Buscar (var A:archivo, elem:tipo_dato_elemento):
integer
{retorna el NRR del nodo; si el árbol es vacío, retorna
—1}

Var
  Raiz: nodo;
  Encontre:boolean;
  Pos:integer;

Begin

  Reset(A);
  Encontre := False;
  Pos      := —1;
```

continúa >>>

```
If not eof(A) Then
 begin
     Read(A, Raiz);

     While not (encontre) and not eof(A) do
           If(raiz.elemento > elem) then
              Pos := raiz.hijo_izquierdo;
              Seek(pos);
              Read(A, Raiz);
           Else
              If (Raiz.elemento < elem) then
                 Pos:= raiz.hijo_derecho;
                 Seek(pos);
                 Read(A, Raiz);
              Else
                 Encontre := true;

 End;

 {pos tiene por defecto –1 bsino el NRR del nodo donde
 está el elemento buscado}
 Buscar := pos;
End
```

Se puede observar que la búsqueda de un elemento de datos comienza siempre desde la raíz, recorriendo a izquierda o a derecha según el elemento que se desea ubicar. De esta forma, en cada revisión se descarta la mitad del archivo restante, donde el dato buscado seguro no se encuentra. Así, como se mencionó anteriormente, la *performance* para buscar un elemento es del orden $Log_2(N)$ accesos a disco, siendo N la cantidad de nodos (elementos de datos) que contiene el árbol. Esta organización es comparable a lo discutido en el Capítulo 5: el algoritmo de búsqueda sobre un archivo de índices ordenado por clave tiene el mismo orden de *performance*.

Una ventaja de la organización mediante árboles binarios está dada en la inserción de nuevos elementos. Mientras que un archivo se desordena cuando se agrega un nuevo dato, si la organización se realiza con la política de árbol binario, la operatoria resulta más sencilla en términos de complejidad computacional. La secuencia de pasos para insertar un nuevo elemento es la siguiente:

1. Agregar el nuevo elemento de datos al final del archivo.
2. Buscar al padre de dicho elemento. Para ello se recorre el archivo desde la raíz hasta llegar a un nodo terminal.
3. Actualizar el padre, haciendo referencia a la dirección del nuevo hijo.

Se puede observar la *performance* desde el punto de vista de accesos a disco: se deben realizar $Log_2(N)$ lecturas, necesarias para localizar el padre del nuevo elemento, y dos operaciones de escritura (el nuevo elemento y la actualización del padre). De este modo, si se compara este método con reordenar todo el archivo, como se presentó en el Capítulo 5, esta opción resulta mucho menos costosa en términos de *performance* final.

La operación de borrado presenta un análisis similar. Para quitar un elemento de un árbol, este debe ser necesariamente un elemento terminal. Si no lo fuera, debe intercambiarse el elemento en cuestión con el menor de sus hijos mayores. Suponga que se desea quitar el elemento ST del árbol de la Figura 6.1; al no ser un elemento terminal, debe ser reemplazado por TR, que representa al menor de sus hijos mayores.

Esta operatoria de borrado, en términos de acceso a disco, significa reemplazar la llave TR por la llave ST, en el registro 2 del archivo de datos de la Figura 6.3, y, posteriormente, borrar el registro 12. Así, deberán realizarse nuevamente $Log_2(N)$ lecturas (necesarias para localizar tanto a ST como TR) y dos escrituras. La primera de ellas, para dejar a TR en el registro 2, y la segunda, para marcar como borrado al registro 12. Nótese que sobre el archivo se pueden aplicar las técnicas de recuperación de espacio discutidas en el Capítulo 4.

Como conclusión, los árboles binarios representan una buena elección en términos de inserción y borrado de elementos, que supera ampliamente los resultados obtenidos en el Capítulo 5. En lo que se refiere a operaciones de búsqueda, los árboles binarios tienen un comportamiento similar al comportamiento obtenido al disponer de un archivo ordenado.

Archivos de datos vs. índices de datos

Suponga el lector que se dispone de un archivo de datos que contempla los clientes de cierta compañía. La estructura del archivo se presenta en la Figura 6.4. Sobre este archivo se necesita realizar búsquedas de información por código de cliente, por nombre de cliente, por tarjeta de identificación o por fecha de nacimiento, indistintamente.

La pregunta es: ¿por qué criterio se ordena el archivo? El árbol binario que se genera ¿qué tipo de orden va a contemplar? ¿Se van a generar cuatro copias del archivo de manera que cada una de ellas esté ordenada por un criterio diferente?

Figura 6.4

```
type  clientes = record
        código          :  integer;
        nombre          :  string;
        dni             :  string;
        fecha_nacimiento:  fecha;
      end;
      archivocliente = file of clientes;
```

La respuesta a las preguntas anteriores es que son necesarios los cuatro ordenamientos y, por ende, generar cuatro árboles binarios diferentes. Lo que el lector debe contemplar en este momento es que cada estructura ordenada solo necesita el atributo por el cual ordena. Así, un árbol binario que ordena por tarjeta de identidad solo debe tener esta información.

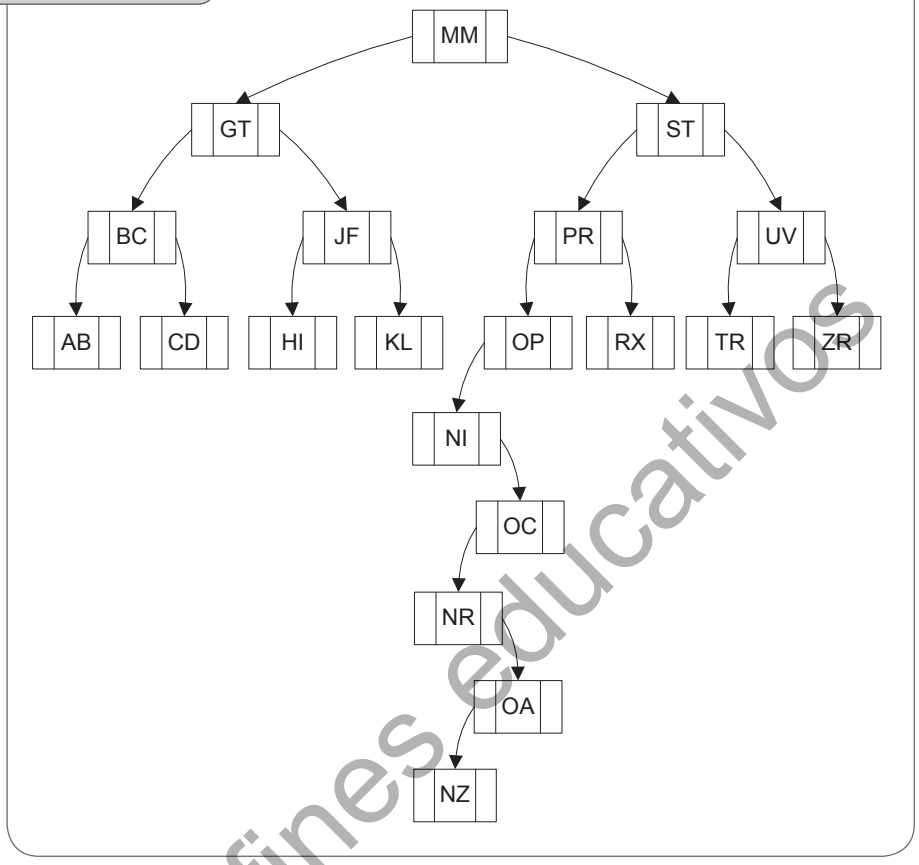
De este modo, y siguiendo lo discutido en el Capítulo 5, se debe separar el archivo de datos de los índices de dicho archivo. El archivo de datos se trata como un archivo serie, donde cada elemento es insertado al final del archivo y no hay orden físico de datos. Se genera, además, un archivo por cada árbol binario que necesite implantar un índice diferente. Cada archivo de índice tendrá la estructura presentada en la Figura 6.3, con la inclusión de la dirección (NRR) del registro completo en el archivo de datos.

Problemas con los árboles binarios

El desempeño de la búsqueda en un árbol binario –y, en particular, en el representado en la Figura 6.1– es bueno porque el árbol se encuentra 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.

¿Qué sucedería si, sobre el árbol de la Figura 6.1, se insertaran en el siguiente orden las claves `NI, OC, NR, OA, NZ`? La Figura 6.5 presenta este caso. Se puede observar que ahora el árbol se encuentra desbalanceado.

Figura 6.5



La *performance* de búsqueda ya no puede considerarse más en el orden logarítmico. El caso degenerado de un árbol binario transforma al mismo en una estructura tipo lista y, en ese caso, la *performance* de búsqueda decae, transformándose en orden lineal.

La pregunta a responder entonces es: ¿en qué situaciones un árbol binario resulta una buena organización para un índice? La respuesta es: cuando el árbol binario está balanceado. La correcta elección de la raíz del árbol determinará si el mismo permanecerá balanceado o no. No obstante, cuando se genera un archivo que implanta un índice de búsqueda, es imposible a priori determinar cuál es la mejor raíz, dado que dependerá de los elementos de datos que se inserten.

Árboles AVL

La situación planteada anteriormente está resuelta. 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 delta, y dicho delta es el nivel de balanceo en altura del árbol. Otra definición posible plantea que un árbol balanceado en altura es aquel en el que la diferencia máxima entre las alturas de cualquiera de dos subárboles que tienen raíz común no supera un delta determinado.

Así, un árbol AVL es un árbol balanceado en altura donde el delta determinado es uno, es decir, el máximo desbalanceo posible es uno. En el caso del árbol de la Figura 6.5, al insertar la clave NI, el árbol aún se mantiene balanceado como AVL(1). Sin embargo, cuando se intenta insertar OC, se viola el precepto establecido. Se debe llevar a cabo, en esta situación, un rebalanceo de este árbol. No es propósito de este libro definir con detalle este proceso. Se debe comprender que, si bien este algoritmo es relativamente sencillo de implementar, los costos computacionales de acceso a disco aumentan considerablemente, por lo que su implantación deja de ser viable.

Entonces, se dispone de una estructura capaz de mantener el balanceo acotado, pero asumiendo mayores costos en las operaciones de inserción y borrado. Por lo tanto, los árboles binarios y los AVL no representan una solución viable para los índices del archivo de datos.

Paginación de árboles binarios

En capítulos anteriores, se discutió el concepto de *buffering*. Cuando se transfiere desde o hacia el disco rígido, dicha transferencia no se limita a un registro, sino que son enviados un conjunto de registros, es decir, los que quepan en un *buffer* de memoria. Las operaciones de lectura y escritura de datos de un archivo utilizando *buffers* presentan una mejora de *performance*. Este concepto es de utilidad cuando se genera el archivo que contiene el árbol binario. Dicho árbol se divide en páginas, es decir, se pagina, y cada página contiene un conjunto de nodos, los cuales están ubicados en direcciones físicas cercanas, tal como se muestra en la Figura 6.6.

Así, cuando se transfieren datos, no se accede al disco para transferir unos pocos bytes, sino que se transmite una página completa.

Una organización de este tipo reduce el número de accesos a disco necesarios para poder recuperar la información. En el árbol de la Figura 6.6 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. Esto es considerando que para transferir una página completa, si bien se accede a direcciones físicas diferentes, tales direcciones son cercanas y los desplazamientos realizados para su acceso son despreciables.

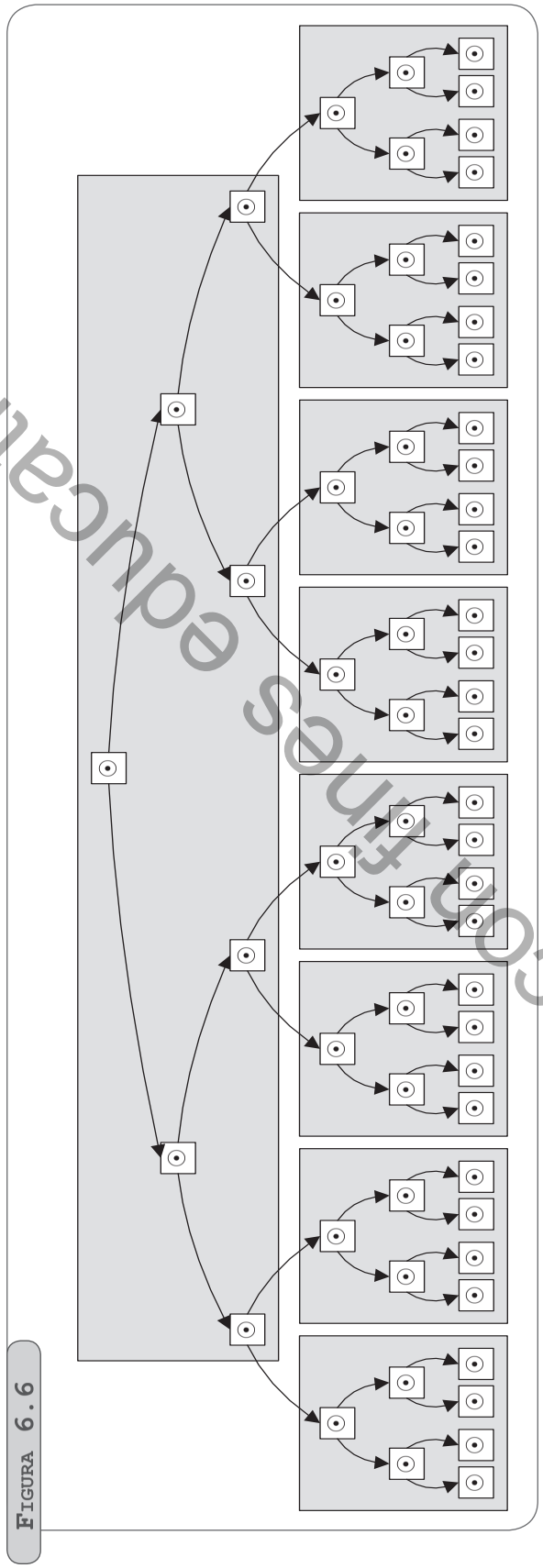


Figura 6.6

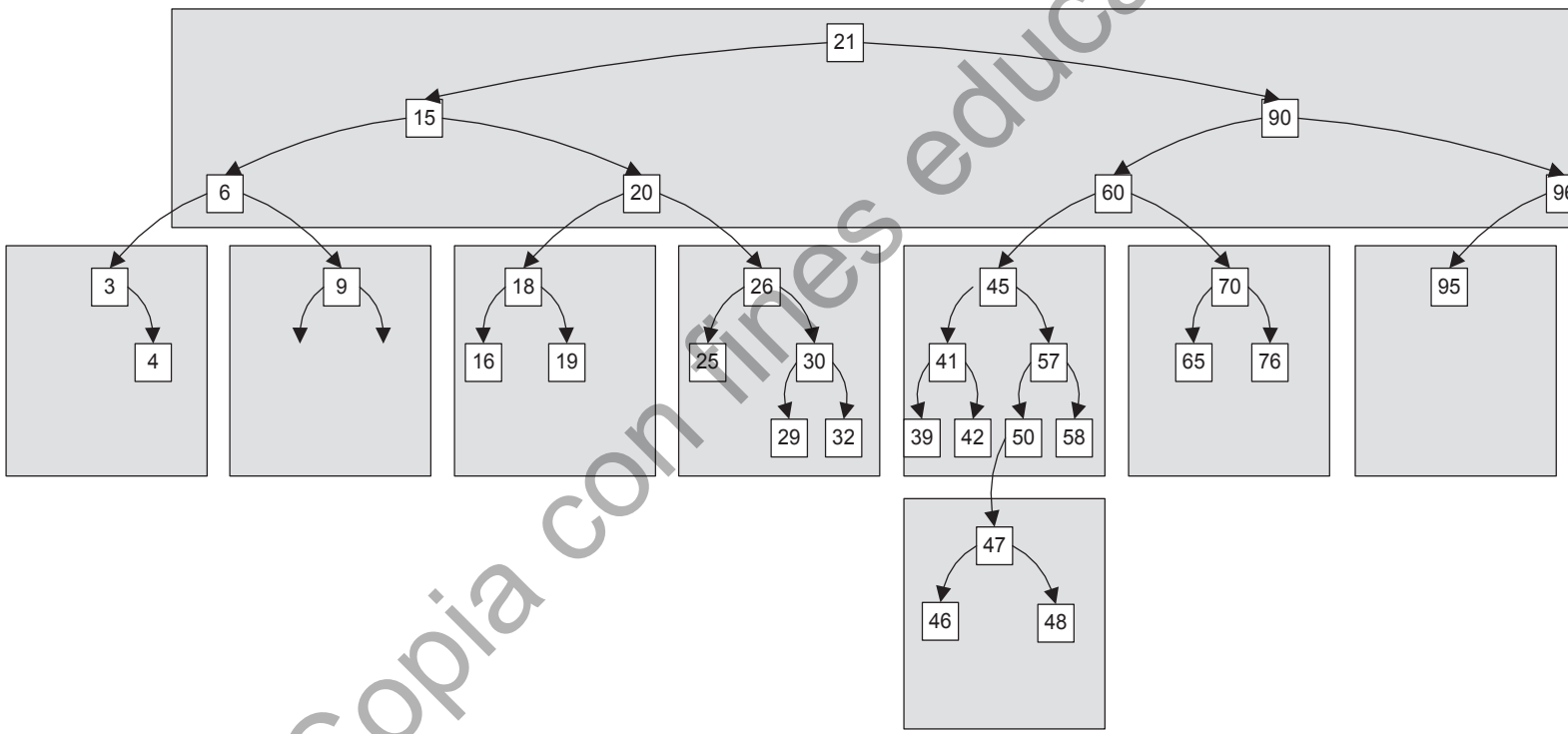
Al dividir un árbol binario en páginas, es posible realizar búsquedas más rápidas de datos almacenados en memoria secundaria. Se puede observar que, en el ejemplo expuesto, cada página contiene siete elementos, pero es solo para presentar la situación. En una situación más realista, una página puede contener 255 o 511 nodos del árbol, con lo cual la altura relativa decrece rápidamente, permitiendo recuperar un elemento del árbol en muchos menos accesos que los planteados originalmente ($log_2(N)$). Ahora, para analizar la *performance* resultante, deberían tenerse en cuenta la cantidad de nodos que caben en una página. Así, suponiendo que en un *buffer* caben 255 elementos, el tamaño de cada página sería entonces de 255 nodos, resultando la *performance* final de búsqueda del orden de $log_{256}(N)$, es decir, $log_{k+1}(N)$, siendo N la cantidad de claves del archivo y K la cantidad de nodos por página.

La cuestión a analizar, ahora, es cómo generar un árbol binario paginado. Para que el uso de las páginas tenga sentido, los siete elementos correspondientes a la misma (de acuerdo con la Figura 6.6) deberían llegar consecutivos para almacenarse en el mismo sector del disco. En su defecto, sería imposible que una página contuviese los elementos relacionados. Como esta opción es imposible, no se puede condicionar la llegada de elementos al archivo de datos, y es menester encontrar otra solución. Así, cada página deberá quedar incompleta si al llegar un elemento no estuviese relacionado con los anteriores. Es decir, un elemento deberá estar contenido en la página que le corresponde, en lugar de quedar en la primera disponible. La Figura 6.7 presenta gráficamente la forma que tendría un árbol binario paginado de acuerdo con las claves que se insertan en él.

Para lograr esta construcción, hay que pensar en páginas, los elementos que caben en cada una de ellas, la forma en que se construye un árbol binario y, además, los temas relacionados con el balanceo. Dividir el árbol en páginas implica un costo extra necesario para su reacomodamiento y para mantener su balanceo interno. Un algoritmo que soporte esta construcción será muy costoso de implementar y luego también en cuanto a *performance*.

Aquí se plantea una disyuntiva, el paginado de árboles binarios con su beneficio y el costo que ello trae aparejado. La solución para este problema consiste en adoptar la idea de manipular más de un registro (es decir, la página) y tratar de disponer de algoritmos a bajo costo para construir un árbol balanceado.

Figura 6.7



Árboles multicamino

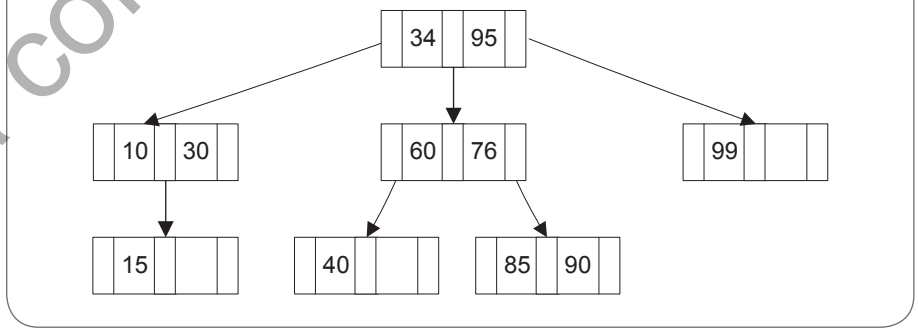
Los árboles binarios son un tipo de estructura de datos que resulta muy simple, pero luego de presentar el concepto de árboles binarios paginados se deberían replantear las siguientes ideas: ¿por qué los árboles deben ser binarios?, ¿por qué no es posible tener un árbol ternario?, ¿qué sería tener un árbol ternario? y, por último, ¿cómo generalizar estas ideas?

Anteriormente se definió a un árbol binario como una estructura de datos en la cual cada nodo puede tener cero, uno o dos hijos. Siguiendo esta línea se podría definir a un árbol ternario como aquel en el cual cada nodo puede tener cero, uno, dos o tres hijos.

La Figura 6.8 presenta un árbol ternario. En el nodo raíz hay dos elementos, el 34 y el 95; a la izquierda del 34, como primer puntero se referencia a un nodo que contiene los elementos menores que 34. Los valores comprendidos entre 34 y 95 se referencian por un puntero existente entre ambos, en tanto que a la derecha del 95 se apunta hacia el nodo que contiene los elementos mayores.

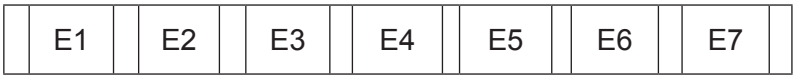
Debe notarse que no todos los nodos están “completos” en cuanto a elementos y en punteros, es decir, hay nodos con lugares libres y con la posibilidad de tener más hijos. En definitiva y en esencia, un árbol ternario es similar a uno binario, solo que cada nodo tiene la posibilidad de poseer hasta dos elementos y hasta tres hijos.

Figura 6.8



La Figura 6.7 presentó un árbol binario paginado, donde cada página contiene siete nodos. ¿Qué sucede ahora si, en lugar de pensar en un árbol binario paginado, que contiene siete nodos repartidos en tres niveles, se construye un árbol que contenga en un nodo los siete elementos? La Figura 6.9 presenta la estructura que debería tener dicho nodo. Se puede observar que la misma contendrá ocho punteros hacia los hijos de ese nodo padre.

Figura 6.9



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.

Así, el orden de un árbol binario es dos (máxima cantidad de descendientes), el orden de un árbol ternario será tres, y, en general, un árbol multicamino de orden *M* contendrá un máximo de *M* descendientes por nodo.

Un árbol multicamino representa otra forma de resolver el concepto de página vertido anteriormente. En este caso, el orden del árbol dependerá del tamaño de la página y de los elementos que se coloquen en ella.

Sin embargo, queda latente aún el problema del balanceo. Como muestra la Figura 6.8, el árbol ternario generado mantiene este problema, el cual puede generalizarse a los árboles multicamino. Se ha dado respuesta parcial entonces a los inconvenientes planteados. Es posible redefinir la estructura de un árbol binario paginado para hacerla manejable como un árbol multicamino; sin embargo, deberá plantearse alguna solución al problema del balanceo. El Capítulo 7 aborda dicho problema.

Cuestionario del capítulo

1. ¿Qué ventaja presenta una organización tipo árbol para representar los índices de un archivo de datos?
2. ¿Por qué un árbol binario resulta insuficiente para organizar el índice?
3. ¿Los árboles AVL representan una solución viable? ¿Por qué motivo?
4. ¿Qué características ponderan los árboles binarios paginados como estructura alternativa?
5. En función de sus respuestas a la Pregunta 2, ¿los árboles multicamino solucionan dichos inconvenientes?

Ejercitación

1. A partir de los ejemplos en seudocódigo del capítulo, implemente el algoritmo que permita borrar un elemento de datos de un archivo que contiene un árbol binario.
2. Discuta la estructura de datos necesaria para almacenar un árbol ternario.
3. Discuta los algoritmos de creación, búsqueda y eliminación de elementos sobre un árbol ternario.
4. Discuta la estructura de datos necesaria para almacenar un árbol multicamino.
5. Sabiendo que la *performance* de búsqueda sobre un árbol binario balanceado es de orden $Log_2(N)$, discuta cuál sería la *performance* de búsqueda sobre un árbol ternario balanceado.
6. Generalice la respuesta anterior para un árbol multicamino.
7. ¿Por qué un árbol debe permanecer balanceado para asegurar las respuestas obtenidas en los Ejercicios 5 y 6?
8. Suponga que dispone de páginas de 4 Kb y que los elementos del árbol ocupan, cada uno, 20 bytes. Si se desea generar un árbol multicamino, ¿cuál podría ser el orden del mismo?

Capítulo 7

Familia de árboles balanceados

Objetivo

En este capítulo se plantea la utilización de árboles balanceados como solución a la implantación de índices para controlar el acceso a la información contenida en archivos de datos. Los primeros tipos de árboles balanceados presentados son los árboles *B*; se discuten sus características, operaciones y *performance*. Luego, se presentan los árboles *B** como alternativa, nuevamente analizando sus características y *performance*.

Por último, se presentan y discuten los árboles *B*+, estructuras que permiten optimizar la búsqueda de información, ya que otorgan además un acceso secuencial eficiente.

Árboles *B* (balanceados)

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.

Analizando las propiedades anteriores, la primera de ellas establece que si el orden de un árbol *B* es, por ejemplo, 256, esto indica que la máxima cantidad de descendientes de cada nodo es 256, porque este valor determina la cantidad de punteros disponibles.

La segunda propiedad posibilita que la raíz no disponga de descendientes, pero cuando sea necesario deberá contar con dos, como mínimo. En ningún caso, la raíz puede tener un solo descendiente.

La tercera característica determina que cualquier nodo que posea una cantidad determinada de hijos, por ejemplo, 256, necesariamente debe contener 255 elementos, es decir, uno menos.

La cuarta propiedad establece la mínima cantidad de elementos contenidos en un nodo terminal. Al ser *M* el orden del árbol, la máxima cantidad admisible de elementos será *M*-1, pero, además, la mínima cantidad de elementos es uno menos que la parte entera de la mitad del orden. Así, si 256 fuera el orden establecido, un nodo terminal deberá poseer 127 elementos como mínimo y 255 como máximo.

La quinta propiedad implica una restricción parecida a la anterior: cualquier nodo que no sea terminal o raíz debe tener una cantidad mínima de descendientes. Siguiendo con un árbol de orden 256, cada nodo deberá tener un mínimo de 128 descendientes.

La última propiedad establece el balanceo del árbol. La distancia desde la raíz hasta cada nodo terminal debe ser la misma. Esta opción permite analizar la *performance* de un árbol *B* y saber que esta *performance* será respetada sin importar cómo se construya, a partir de la llegada de los elementos de datos.

Hasta el momento, se dispone de una definición de árbol *B* y de seis propiedades. Se deberá ahora determinar la estructura de datos que brinde soporte a estos árboles y discutir las operaciones que permitan crear, buscar, borrar y mantener la información contenida en esta estructura de datos.

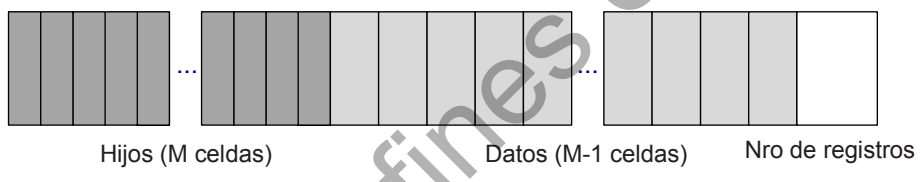
En el Capítulo 6, cuando se presentó una estructura tipo árbol como implementación posible para los índices de un archivo de datos, se discutió que dichas estructuras contendrían solamente la información para permitir el acceso eficiente a estos archivos. Así, por un lado, se manipula el archivo de datos como un archivo serie, y por otro, cada uno de los archivos que implementaban a los índices respectivos, con una referencia al registro completo en el archivo de datos. En este capítulo, se continúa en ese sentido. Los archivos que contendrán a los índices representados como estructuras de tipo árbol *B*, *B** o *B*+ solamente contendrán información que permitirá el ordenamiento y la búsqueda eficiente, quedando el registro completo en el archivo de datos.

La Figura 7.1 presenta el formato de un nodo posible para un árbol *B* de orden *M*. Dicha figura se corresponde con la definición de la estructura de datos siguiente:

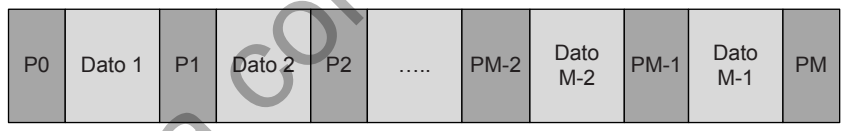
```
Const orden = 255;
Type reg_arbol_b = record;
          Hijos: array [0..orden] of integer;
          Claves: array [1..orden] of tipo_de_dato;
          Nro_registros: integer;
      End;
```

Donde `hijos` es un arreglo que contiene la dirección de los nodos que son descendientes directos, `claves` es un arreglo que contiene las claves que forman el índice del árbol y `nro_registros` indica la dimensión efectiva en uso de cada nodo, es decir, la cantidad de elementos del nodo. Nótese que el vector de `Hijos` es un elemento más grande que el vector de `Claves`. Nótese además que el orden del árbol en este caso es 256.

Figura 7.1



Formato del nodo para archivo del índice árbol b



Formato gráfico del nodo del índice árbol B

Operaciones sobre árboles *B*

En este apartado, se describen las operaciones `Crear/Insertar, Buscar, Borrar y Modificar` sobre un árbol *B*. Además, en cada caso, se analizará la *performance* de cada una de dichas operaciones. La primera operación a discutir indica la forma en que debe ser creado un árbol *B*.

Creación de árboles *B*

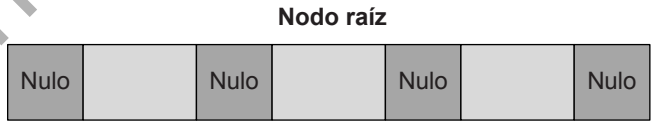
Un árbol balanceado tiene como principal característica que todos los nodos terminales se encuentran a la misma distancia del nodo raíz. Para poder lograr esta construcción, no es posible concebir un árbol *B* de la misma forma que un árbol "tradicional". Cuando no se dispone de suficiente espacio y para preservar el principio de altura constante, es la raíz la que debe alejarse de los nodos terminales.

El proceso de creación comienza con una estructura vacía. Para permitir que el lector comprenda adecuadamente el proceso de creación, se resolverá simultáneamente el problema en forma gráfica, sobre un árbol, y también se presentará la generación del archivo de índices asociado.

A fines prácticos, las claves que compondrán el árbol *B* serán pares de letras y, además, el orden del árbol se establecerá en 4. Este orden del árbol tan pequeño se determina para permitir, en pocos pasos, presentar el mecanismo de creación.

En el momento inicial, el único nodo existente en el árbol será un nodo raíz, el cual no contendrá ningún elemento. Asimismo, el archivo del índice se encontrará vacío. La Figura 7.2 y la Tabla 7.1 (representación gráfica del registro del archivo índice) presentan esta situación.

Figura 7.2



Construcción gráfica del árbol

Tabla 7.1

Nodo Raiz: Nulo		
Punteros	***Datos***	***Nro Datos***

Cuando llega el primer elemento, `FG`, este se inserta en la primera posición libre del nodo raíz; este proceso se muestra en la Figura 7.3 y en la Tabla 7.2. El proceso detecta que, en el archivo de índice, el nodo raíz es nulo; por lo tanto, se inserta el primer registro del

archivo en el primer lugar de los datos, la clave ingresada FG, el número de lugares ocupados se establece en 1 y, al no tener este nodo raíz descendientes, los punteros a los hijos se establecen en nulo. Nótese que se utiliza –1 para indicar valor nulo; cuando una dirección es válida, debe registrar el nodo siguiente. El nodo siguiente es otro registro del archivo; por ese motivo se selecciona –1 como indicador de nulo. El primer registro del archivo está ubicado en la posición cero.

Figura 7.3

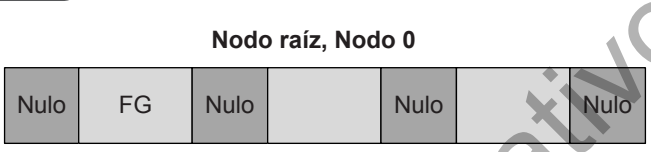


Tabla 7.2

	Nodo Raiz: 0						
	Punteros				***Datos***		***Nro.Datos***
0	-1	-1			FG		1

Los elementos siguientes comenzarán siempre el proceso de inserción a partir del nodo raíz. Si el nodo raíz tiene lugar, se procederá a insertarlos en este nodo teniendo en cuenta que los elementos de datos deben quedar ordenados dentro del nodo. Se debe notar que este proceso no resulta costoso, dado que el registro que contiene al nodo raíz está almacenado en memoria; por lo tanto, su ordenación solo debe contabilizar accesos a memoria RAM.

De esta forma, si llegan los elementos TR y AD, la Figura 7.4 y la Tabla 7.3 muestran cómo queda el índice construido.

Figura 7.4

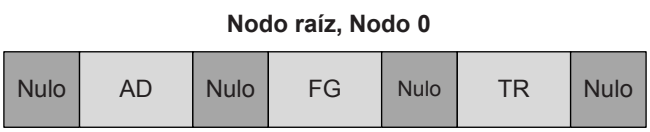


Tabla 7.3

	Nodo Raiz: 0							
	Punteros				Datos			Nro Datos
0	-1	-1	-1	-1	AD	FG	TR	3

La llegada de un nuevo elemento provocará un *overflow* en el nodo. Este *overflow* significa que en el nodo no hay capacidad disponible para almacenar un nuevo elemento de datos.

Supóngase que la siguiente clave es `ZT`; cuando se intenta insertar la clave en el registro 0, se determina el *overflow*. Ante la ocurrencia de *overflow*, el proceso es el siguiente:

1. Se crea un nodo nuevo.
2. La primera mitad de las claves se mantienen en el nodo viejo.
3. La segunda mitad de las claves se trasladan al nodo nuevo.
4. La menor de las claves de la segunda mitad se promociona al nodo padre.

En el ejemplo planteado, se manipulan cuatro claves: `AD`, `FG`, `TR`, `ZT`. En este caso, el registro 0 mantendrá a `AD` y `FG`, en el registro 1 se dejará `ZT`, y `TR` se promocionará al nodo padre. Como el nodo dividido, el nodo cero, era hasta el momento la raíz del árbol, se debe generar un nuevo registro, el 2, que será considerado el nuevo nodo raíz. (Ver Figura 7.5 y Tabla 7.4). En la Tabla 7.4, se puede observar que la dirección del nuevo nodo raíz fue modificada y ahora apunta al nodo 2.

Figura 7.5

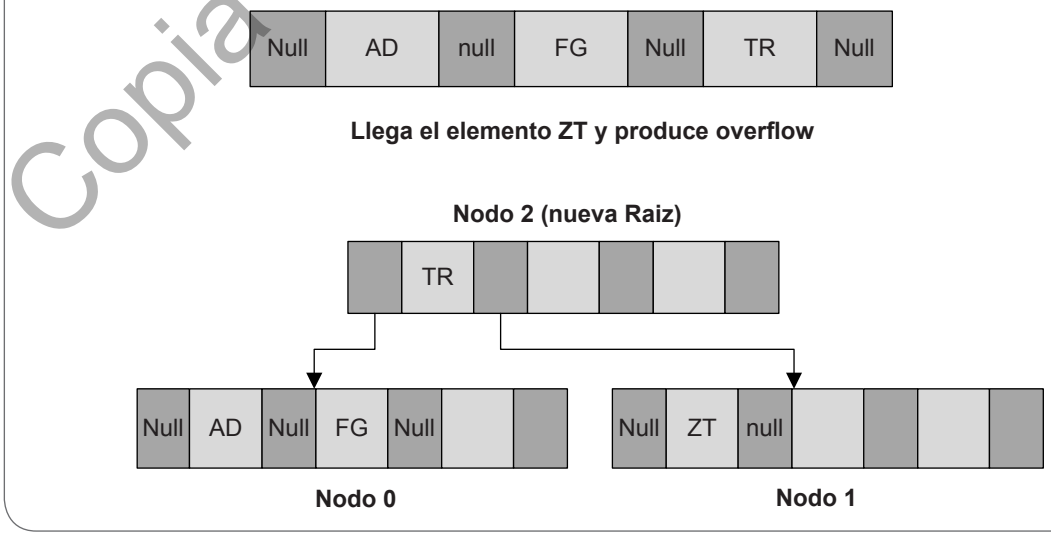


Tabla 7.4

	Nodo Raiz: 0							
	Punteros				Datos			Nro Datos
0	-1	-1	-1		AD	FG		2
1	-1	-1			ZT			1
2	0	1			TR			1

Siguiendo con el ejemplo, se intenta insertar la clave WA; algorítmicamente el proceso comienza desde el nodo raíz, el nodo 2. WA es mayor que la clave actual del nodo TR, la dirección a derecha del primer elemento del nodo es 1; esto significa que no es un nodo terminal, y como los elementos deben insertarse en un nodo terminal, se recupera desde memoria secundaria el nodo 1. Aquí se determina que la clave WA es menor que ZT, y como la dirección a izquierda de ZT es nula, se está ante un nodo terminal que tiene capacidad para almacenar WA. La clave es ubicada en dicho nodo. (Ver Figura 7.6 y Tabla 7.5).

Figura 7.6

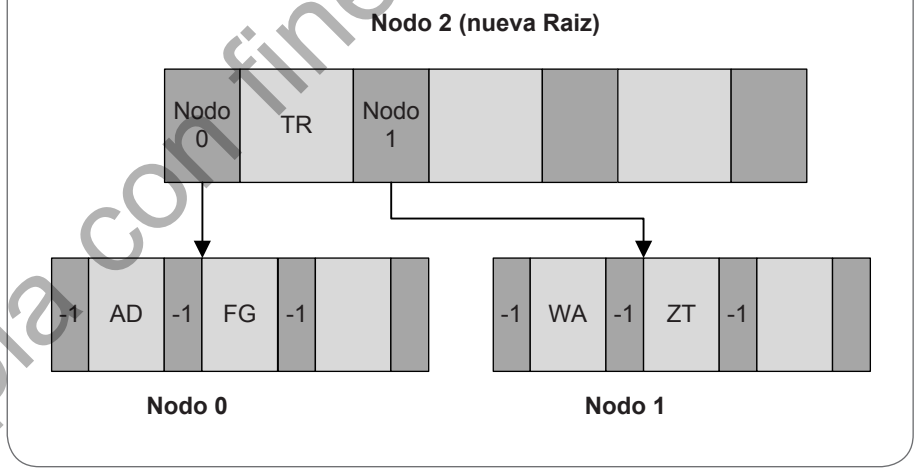


Tabla 7.5

	Nodo Raiz: 2							
	Punteros				Datos			Nro Datos
0	-1	-1	-1		AD	FG		2
1	-1	-1	-1		WA	ZT		2
2	0	1			TR			1

Supóngase que ahora se deben insertar las claves `BH` y `XF`; procediendo igualmente que el paso anterior, cada elemento es insertado en el nodo 0 y el nodo 1, respectivamente, sin generar *overflow* sobre dichos nodos. La Figura 7.7 y la Tabla 7.6 dan cuenta de dicha situación.

Figura 7.7

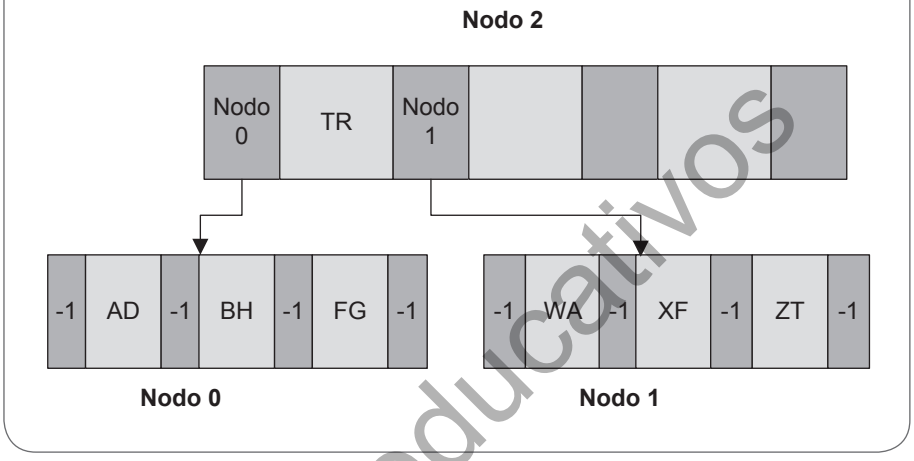


Tabla 7.6

	Nodo Raiz: 2							
	Punteros				*Datos*			*Nro Datos*
0	-1	-1	-1	-1	AD	BH	FG	3
1	-1	-1	-1	-1	WA	XF	ZT	3
2	0	1			TR			1

La siguiente clave que se debe insertar en el árbol es `MN`; el proceso nuevamente comienza desde el nodo raíz y, desde este, se accede al nodo 0. El nodo 0 es un nodo terminal pero no dispone de capacidad para almacenar más elementos. Esta situación produce un *overflow* en el nodo 0, por lo que se debe generar un nuevo nodo, el 3 en este caso. El lector debe notar que las claves que producen *overflow* son `AD`, `BH`, `FG` y `MN`. En el nodo 0 quedarán las claves `AD` y `BH`, en el nodo 3 quedará `MN`, y `FG` será promocionada al nodo raíz junto con la dirección del nuevo nodo generado. En el nodo 2 se producirá un reacomodamiento para permitir que `FG` sea insertado en el orden respectivo. (Ver Figura 7.8 y Tabla 7.7).

Figura 7.8

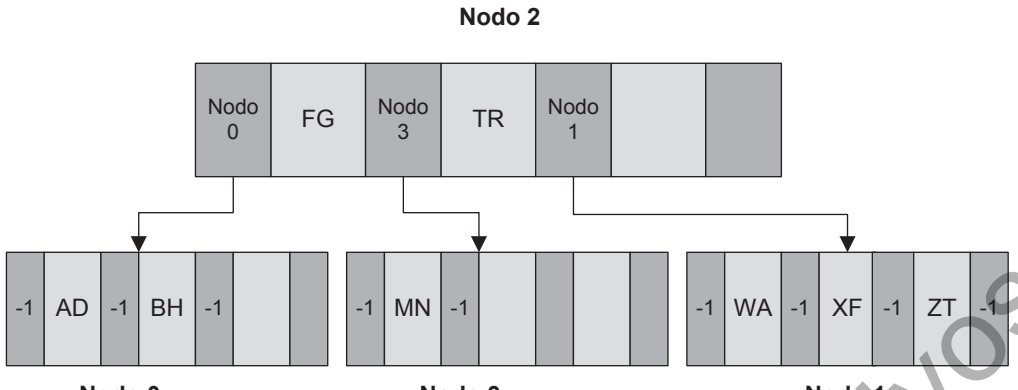


Tabla 7.7

	Nodo Raiz: 2							
	Punteros				Datos			Nro Datos
0	-1	-1	-1		AD	BH		2
1	-1	-1	-1	-1	WA	XF	ZT	3
2	0	3	1		FG	TR		2
3	-1	-1			MN			1

La clave a insertar ahora es VU. Procediendo de la misma forma que en el paso anterior, se producirá *overflow* sobre el nodo 1. Se genera un nuevo nodo, el 4, y luego de realizar las divisiones respectivas quedarán las claves VU y WA en el nodo 1, ZT en el nodo 4 y XF se promocionará al nodo raíz. Esta situación se presenta en la Figura 7.9 y en la Tabla 7.8.

Figura 7.9

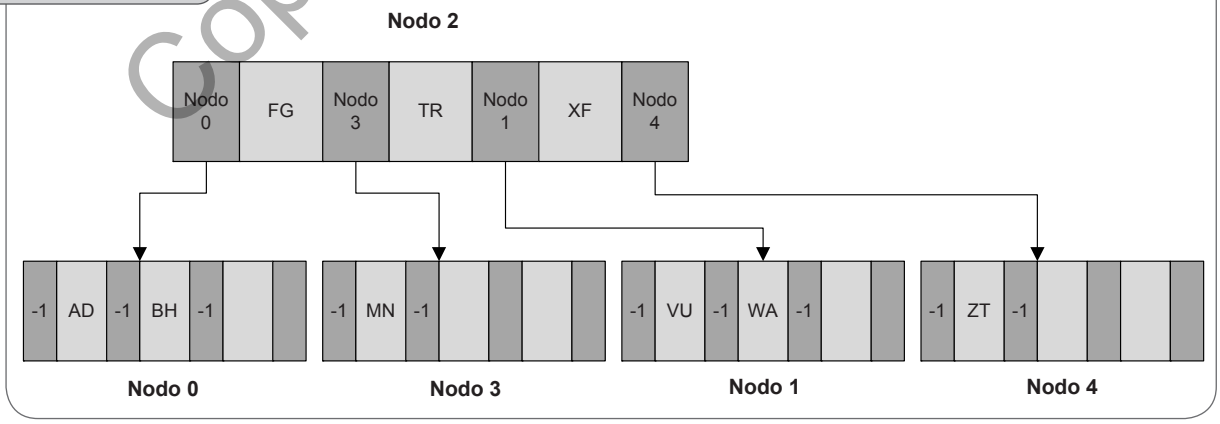


Tabla 7.8

	Nodo Raiz: 2							
	Punteros				Datos			Nro Datos
0	-1	-1	-1		AD	BH		2
1	-1	-1	-1		VU	WA		2
2	0	3	1	4	FG	TR	XF	3
3	-1	-1			MN			1
4	-1	-1			ZT			1

Se deben introducir en el árbol las claves CD y UW. Las mismas son insertadas en los nodos 0 y 1, respectivamente, sin producir *overflow*. (Ver Figura 7.10 y Tabla 7.9).

Figura 7.10

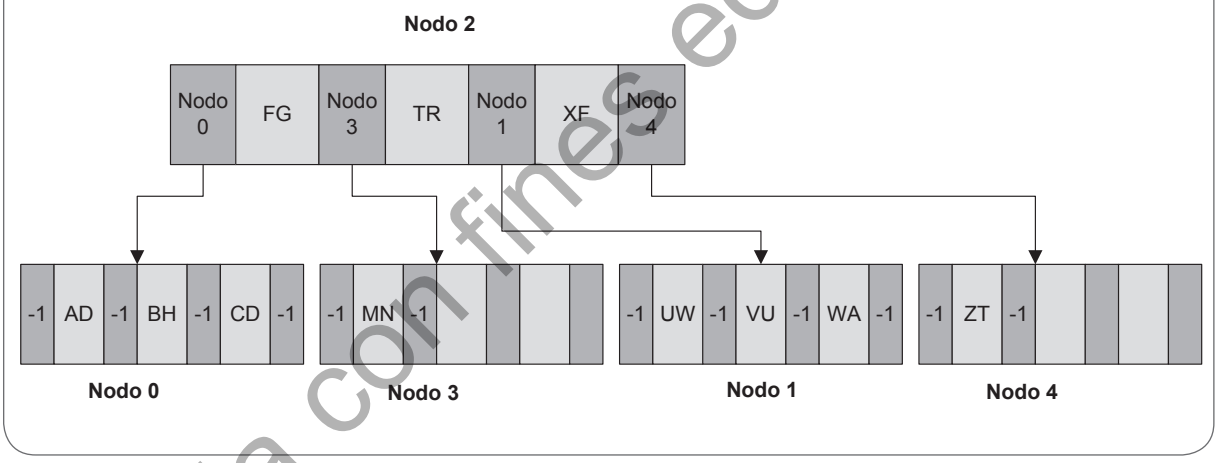


Tabla 7.9

	Nodo Raiz: 2							
	Punteros				Datos			Nro Datos
0	-1	-1	-1	-1	AD	BH	CD	3
1	-1	-1	-1	-1	UW	VU	WA	3
2	0	3	1	4	FG	TR	XF	3
3	-1	-1			MN			1
4	-1	-1			ZT			1

Por último, se inserta la clave `DI` en el árbol. Desde el nodo raíz se accede al nodo 0, donde debería insertarse la clave. Pero, en este nodo, nuevamente se produce *overflow*. Se crea, entonces, el nodo 5, las claves `AD` y `BH` quedan en el nodo 0, y `DI` pasa al nodo 5, promocionando al nodo padre la clave `CD`. Se debe notar que ahora en el nodo 2 se produce *overflow*. En este caso se procede nuevamente dividiendo dicho nodo, y se genera de esta forma el nodo 6. Las claves `CD` y `FG` quedan en el nodo 2; `XF` pasa al nodo 6; en tanto `TR` debe promocionarse al nodo raíz. Este nodo raíz debe generarse dado que hasta el momento el nodo 2 actuaba como tal. Se crea el nodo 7 (nueva raíz) que contendrá a la clave `TR` y a la dirección de sus nuevos hijos: el nodo 2 y el nodo 6. Obsérvese que, en la Tabla 7.10, se modifica la dirección del nodo raíz, siendo ahora el nodo 7.

El proceso continúa con la inserción de elementos en el árbol de la misma forma; cuando el árbol crece en altura, es la raíz la que se aleja de los nodos terminales, de modo tal que siempre el árbol crezca de manera balanceada. En todo momento, durante el proceso de creación del árbol, las propiedades descriptas anteriormente fueron respetadas.

En el presente libro no se describe el algoritmo que permite generar un árbol *B*. Este algoritmo se deja como ejercicio intelectual para el lector, considerando que se encuentra disponible en diversos sitios web, con versiones en diferentes lenguajes de programación.

Figura 7.11

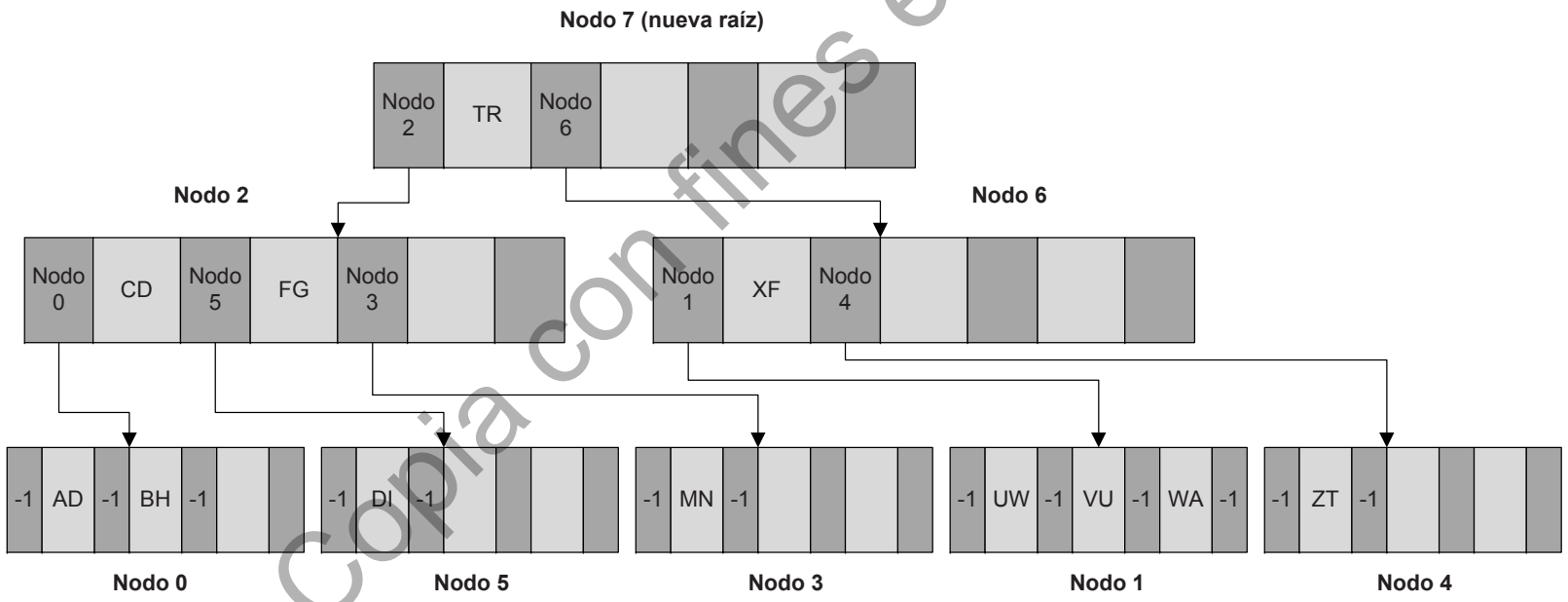


Tabla 7.10

	Nodo Raiz: 7							
	Punteros				Datos			Nro Datos
0	-1	-1	-1		AD	BH		2
1	-1	-1	-1	-1	UW	VU	WA	3
2	0	5	3		CD	FG		2
3	-1	-1			MN			1
4	-1	-1			ZT			1
5	-1	-1			DI			1
6	1	4			XF			1
7	2	6			TR			1

Búsqueda en un árbol *B*

En el apartado anterior, se describió mediante un ejemplo el mecanismo de inserción de elementos en un árbol *B*. El proceso de búsqueda coincide con la primera parte del proceso mostrado.

El primer paso de la inserción consiste en localizar el nodo que debería contener al nuevo elemento. El proceso de búsqueda realiza la misma operación; comenzando desde el nodo raíz, se procede a buscar el elemento en cuestión. En caso de encontrarlo en dicho nodo, se retorna una condición de éxito (esto implica retornar la dirección física en memoria secundaria, asociada al registro que contiene la clave encontrada). 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.

Tomando como ejemplos la Figura 7.11 y la Tabla 7.10, si se busca el elemento `DI`, comenzando desde el nodo 7 (nodo raíz), allí no se encuentra tal elemento y, siendo `DI` una clave menor que `TR`, se procede a recuperar el nodo inmediato izquierdo (el nodo 2). En dicho nodo, nuevamente no se encuentra la clave `DI`, y como esta es mayor que `CD` y menor que `FG`, se bifurca al nodo 5, donde se localiza la clave buscada, y, por lo tanto, el proceso retorna éxito.

Supóngase ahora que la clave buscada es `DR`; procediendo de la misma forma que en el párrafo anterior, se recupera el nodo 5 donde no se encuentra la clave buscada. Como `DR` es mayor que `DI`, se intenta recuperar el nodo derecho. El puntero indica que no hay más nodos disponibles. Por lo tanto, el elemento buscado no se encuentra en el árbol balanceado, y el proceso no retorna éxito dado que el elemento no está en el árbol.

Eficiencia de búsqueda en un árbol *B*

El proceso algorítmico de búsqueda de un elemento en un árbol balanceado no difiere demasiado del mismo proceso en un árbol binario común. Comienza la búsqueda en el nodo raíz y se va bifurcando hacia los nodos terminales en la medida en que el elemento no sea localizado.

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 tal como fuera definida previamente en el Capítulo 6. Si el elemento se encuentra ubicado en el nodo raíz, la cantidad de accesos requeridos es 1. En caso de localizar al elemento en un nodo terminal (o que el elemento no se encuentre), serán requeridos *H* accesos.

La cuestión por resolver, a fin de poder comparar la eficiencia de búsqueda contra los métodos discutidos en capítulos anteriores, consiste en poder comparar *H* (altura, o cantidad de niveles del árbol) con *N* (cantidad de registros en el árbol). Se debe recordar que, en casos anteriores, la eficiencia x se calculó siempre en función de los registros que contiene el archivo.

Para poder comparar resultados, es necesario medir la eficiencia en variables similares. Para ello, se debe acotar el valor de *H*. Una cota para *H* en función de los registros que el árbol contiene (*N*) permitirá realizar la comparación requerida.

Antes de comenzar con el análisis, se tendrá en cuenta, aunque no se demostrará en el presente libro, el siguiente axioma: “Un árbol *B* asociado a un archivo de datos que contiene *N* registros contendrá un total de *N*+1 punteros nulos en sus nodos terminales”. A partir de la Figura 7.11 o de la Tabla 7.10, se deduce que la cantidad de registros almacenados en el archivo es 12, y que la suma de punteros nulos (–1) en los nodos terminales es 13. Este axioma permitirá a continuación acotar *H*.

Para poder realizar una cota, se debe analizar siempre la peor situación posible; de esa forma se garantiza que dicha cota se cumplirá siempre. Para definir la cota, se deben tomar en cuenta las propiedades básicas de un árbol *B*. Una de ellas indica que, en un árbol *B*, el nodo raíz tiene dos descendientes como mínimo o, en su defecto, ninguno. Si ocurriera este último caso, el árbol tendría un solo nivel, y por ende, con un acceso se recupera el elemento buscado o se

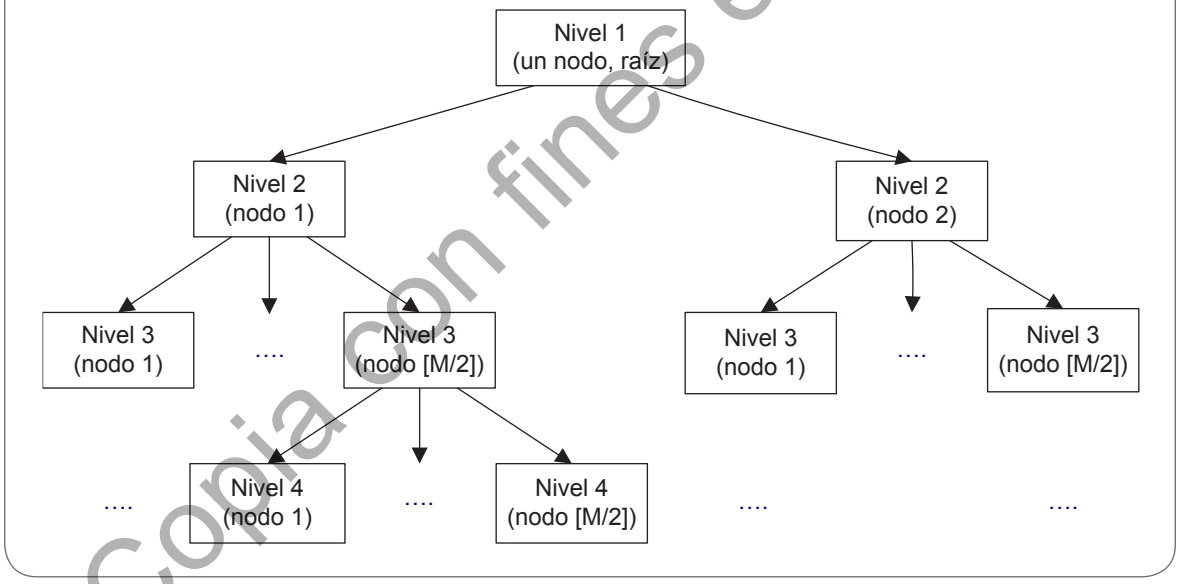
detecta su inexistencia. Ahora bien, el peor caso posible es tener un nodo raíz con dos descendientes (un solo elemento de datos en el nodo raíz). Esta es la peor situación posible, dado que con solo dos descendientes se está forzando la mayor cantidad de niveles posible, y esto permite establecer una cota.

Siguiendo este análisis y teniendo en cuenta otra de las propiedades del árbol *B*, un nodo que no es terminal ni raíz, es decir, un nodo interno, tiene un mínimo de [*M*/2] descendientes, siendo *M* el orden del árbol *B*.

Resumiendo lo anterior, la raíz tiene un mínimo de dos descendientes y cualquier nodo no terminal tiene un mínimo de [*M*/2] descendientes.

Por lo tanto, del primer nivel se dispone de dos nodos descendientes; del segundo nivel se dispone de [*M*/2] nodos descendientes por cada uno de los nodos de ese nivel (dos nodos); del tercer nivel, de [*M*/2] nodos descendientes por cada uno de los nodos de ese nivel (2 * [*M*/2] nodos), y así sucesivamente. (Ver Figura 7.12).

Figura 7.12



Generalizando (valores de cota)

Nivel 1 (raíz)	2 descendientes
Nivel 2	2 * [*M*/2] descendientes
Nivel 3	2 * [*M*/2] * [*M*/2] descendientes
…	
Nivel *H*	2 * ([*M*/2])$^{H-1}$ descendientes

Se debe tener en cuenta ahora el axioma presentado anteriormente: en el último nivel del árbol *B* (nodos terminales) existen en total *N*+1 punteros nulos, si *N* es la cantidad de registros que contiene el árbol.

Por lo tanto, la cantidad de punteros nulos es mayor que la cota de descendientes encontrada anteriormente:

$N+1 >= 2 * ([M/2])^{H-1}$

A partir de un simple análisis de la inecuación anterior, se acotará la cantidad de niveles del árbol (*H*) en función del orden de dicho árbol (*M*) y la cantidad de registros contenidos (*N*).

$(N+1) / 2 > ([M/2])^{H-1}$

Aplicando logaritmos a la fórmula anterior, se obtiene:

$Log_{[M/2]} ((N + 1) / 2) > H - 1$

Y luego:

$H < 1 + Log_{[M/2]} ((N + 1) / 2)$

Arribando así a la cota deseada.

Si se le asigna un valor posible al orden del árbol –por ejemplo, 512–, y se supone un árbol que contiene 1.000.000 de registros, en el peor caso su altura será:

$H < 3.34$

En este ejemplo, en el peor de los casos el árbol tendrá cuatro niveles, es decir que serán necesarios a lo sumo cuatro accesos para recuperar cualquier elemento. Si se compara este resultado estimativo con los presentados en capítulos anteriores, se puede observar una notable mejora en el proceso de búsqueda de información sobre un árbol *B*.

En el apartado siguiente, se analizará la eficiencia del proceso de inserción.

Eficiencia de inserción en un árbol *B*

Tomando como base el análisis previo, que permitió establecer una cota para la búsqueda de información en un árbol *B*, se procederá a analizar el caso de una inserción.

Se debe tener en cuenta que otra de las propiedades de los árboles *B* determina que todos los elementos de datos se insertan en los nodos terminales. Por ese motivo es necesario realizar *H* lecturas para poder encontrar el nodo donde el elemento será almacenado (comenzando en la raíz y avanzando hacia el nodo terminal).

Aquí pueden surgir dos alternativas. Si el nodo a realizar la operación dispone de lugar, la inserción del nuevo elemento no produce *overflow*, solo será necesaria una escritura en el nodo terminal con el nuevo elemento y de esta forma se da por finalizado el proceso de alta.

La alternativa consiste en que se produzca un *overflow* (se debe recordar cómo se trata esta situación, la cual fue explicada anteriormente). El peor caso es aquel donde el *overflow* se propaga hacia la raíz, haciendo aumentar en uno el nivel del árbol. Analizando detalladamente, si el nodo terminal se completa, entonces deberá realizarse un proceso de división, lo que motivará la escritura de dos nodos en el nivel terminal. Se propaga una clave hacia el nivel superior, la cual, como se considera la peor situación, también genera *overflow*; se debe dividir el nodo y realizar dos nuevas escrituras. Esto se repite hacia la raíz, la cual también se divide generando dos escrituras. Por último, se genera una nueva raíz. Todo este proceso genera dos escrituras por nivel y, además, aumenta en uno el nivel del árbol.

Resumiendo, la *performance* de la inserción en un árbol *B* está compuesta por:

H lecturas

1 escritura (mejor caso)

H lecturas

(2 * *H*) + 1 escrituras (peor caso)

Estudios empíricos realizados han demostrado que, generando árboles *B* de orden *M* = 10, la cantidad de *overflows* es de 25% aproximadamente. En tanto, si el orden se eleva a *M* = 100, la cantidad de *overflows* se reduce a 2%. Analizando esta última situación, 98 de cada 100 inserciones se tratan por el mejor caso, en tanto que solamente dos no están dentro de esta consideración, lo que no significa necesariamente que se trate del peor caso.

Por lo tanto, la inserción de nuevos elementos en el árbol *B* requiere un número considerablemente bajo de operaciones de entrada-salida para lograr mantener el orden en la estructura. Queda aún analizar el proceso de baja y su eficiencia.

Eliminación en árboles *B*

Se han tenido en cuenta hasta el momento las operaciones de búsqueda y creación/inserción sobre un árbol *B*. Estas dos operaciones, si bien pueden ser consideradas las más importantes desde un punto de vista de utilización (recuérdese que, en promedio, 80% de las operaciones sobre un archivo son de consultas y que, del 20% restante, la mayoría la representan operaciones de inserción), no son las únicas operaciones posibles.

Otra operación a tener presente consiste en el proceso de baja de información, tal cual como fuera discutido en capítulos anteriores. Más allá de trabajar utilizando borrados lógicos o físicos, o de las técnicas de recuperación de espacio que se podrían llevar adelante, en este apartado se discutirá en abstracto el proceso de baja sobre un árbol *B*.

Para poder avanzar al respecto, se debe tener en cuenta una consideración que se hereda desde los árboles binarios: para poder borrar un elemento, el mismo debe estar localizado en un nodo terminal. Respetando esta consideración, si en la Figura 7.11 se desea eliminar el elemento `TR`, el mismo deberá ser intercambiado por otro elemento residente en un nodo terminal, sin que esto afecte el recorrido del árbol generado. Entonces, algorítmicamente se deberá intercambiar `TR` con la mayor de sus claves menores (`MN`), o la menor de sus claves mayores (`UW`). Se ha demostrado empíricamente que la conveniencia en este reemplazo está dada por la menor de las claves mayores.

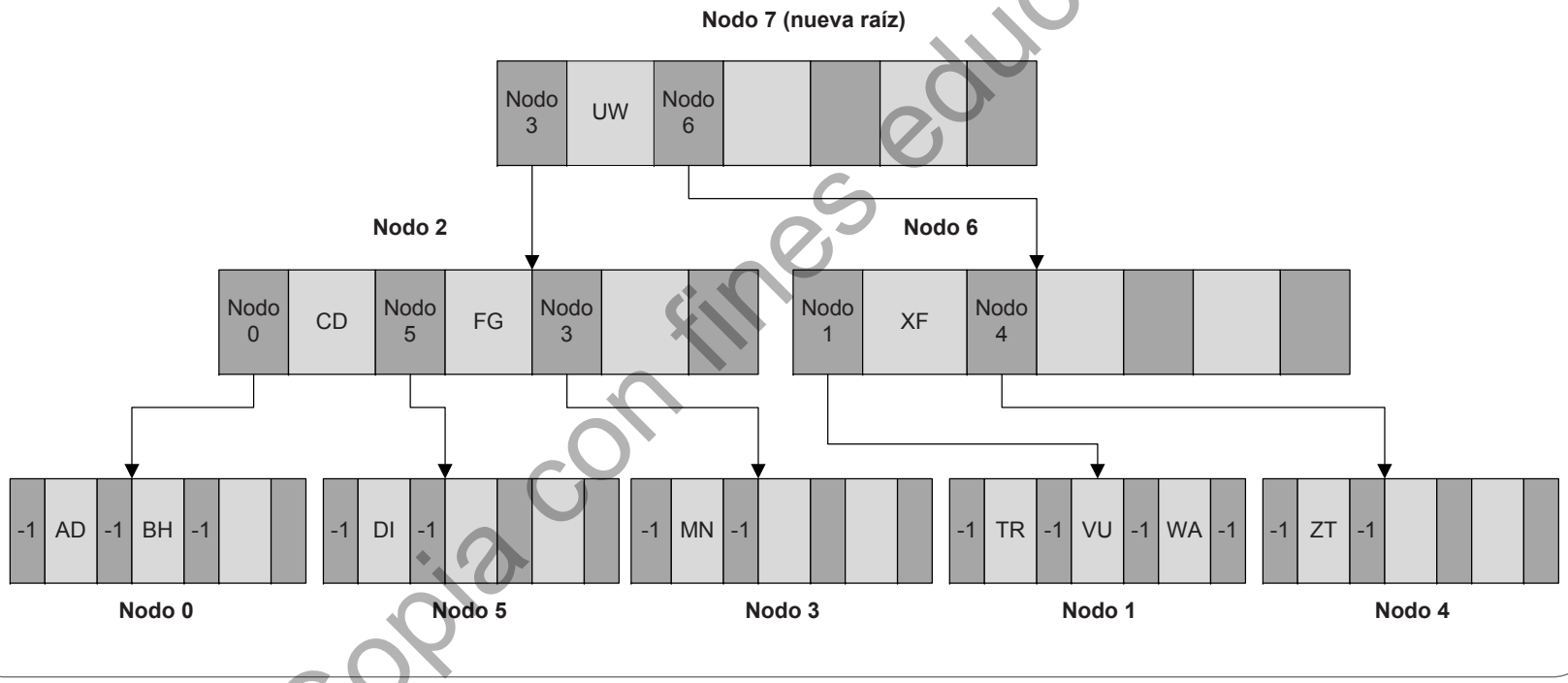
Así, el primer paso para eliminar `TR` será intercambiarlo con `UW`, quedando el árbol temporalmente como se observa en la Figura 7.13. Si bien no se muestra gráficamente, a continuación, `TR` se elimina del nodo terminal (situación a analizar en los párrafos siguientes).

Como ejercicio intelectual y tomando nuevamente la Figura 7.11, suponga ahora que desea eliminar el elemento `FG`, ¿cuál sería la opción de cambio? En este caso, se debería intercambiar `FG` con `MN`.

Como se presentó en el capítulo anterior, en un árbol binario, el proceso de baja finaliza liberando el espacio que el elemento ocupa e indicando su padre una dirección nula. La diferencia con un árbol balanceado radica en que un nodo terminal contiene un conjunto de elementos, entre ellos el que se debe borrar. Luego de quitar este elemento del nodo terminal, se deben seguir cumpliendo las propiedades antes descriptas para árboles *B*.

Existen entonces dos situaciones en el proceso de baja. La primera situación se presenta cuando, al borrar el elemento del nodo terminal,

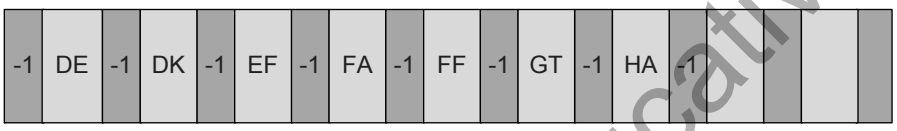
Figura 7.13



la cantidad de elementos restantes no está por debajo de la cantidad mínima ([*M*/2]–1). En este caso, no se genera un *underflow* en el nodo, y el proceso de baja finaliza.

La Figura 7.14 a) representa un nodo terminal de un árbol balanceado de orden 10; sobre ese nodo se desea eliminar el elemento `FA`, el cual una vez eliminado deja al nodo como lo muestra la Figura 7.14 b). Con el orden 10 indicado y como el nodo contenía siete elementos, luego de borrar `FA` quedan seis elementos. Como se sigue cumpliendo la propiedad indicada, el proceso de baja finaliza.

Figura 7.14



Nodo Terminal

Figura 7.14 a)

Nodo Terminal

Figura 7.14 b)

La segunda situación posible es aquella que genera una situación de *underflow*. En este caso, el nodo al que se le quita un elemento deja de cumplir la condición de contener al menos [*M*/2] –1 elementos. A fin de poder ilustrar esta situación, es necesario presentar algunas definiciones extra.

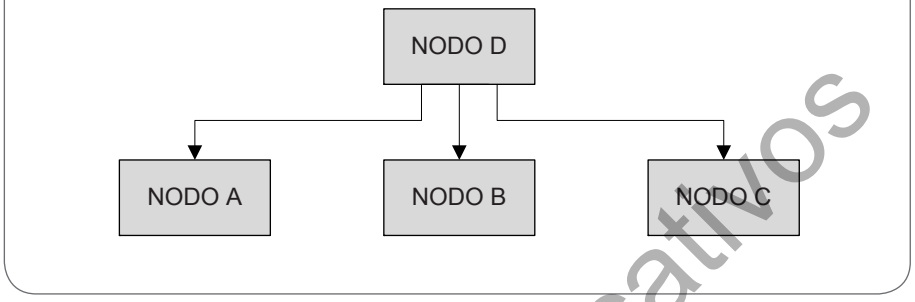
Se denomina **nodos hermanos** a aquellos nodos que tienen el mismo nodo padre.

Se denomina **nodos adyacentes hermanos o nodos hermanos adyacentes** a aquellos nodos que, siendo hermanos, son además dependientes de punteros consecutivos del padre.

La Figura 7.15 ilustra las definiciones anteriores. Los tres nodos hijos –*A*, *B* y *C*– dependen del mismo padre, *D*; por ende, son nodos hermanos. Ahora, el nodo *A* y el nodo *B* son apuntados por punteros

consecutivos, entonces son nodos adyacentes hermanos. Lo mismo ocurre entre *B* y *C*. Se debe notar que los nodos *A* y *C* solo son hermanos, puesto que, al no ser apuntados por punteros consecutivos en el padre, no pueden considerarse adyacentes.

Figura 7.15



Retomando el proceso de baja, ya se mostró que la división de nodos garantiza el cumplimiento de las propiedades de un árbol *B* cuando se insertan nuevos elementos en caso de *overflow*. Ahora, se debe garantizar el mismo cumplimiento de estas propiedades cuando se eliminen las claves y se produzca *underflow* sobre un nodo. Aquí, las acciones que es posible llevar a cabo varían de acuerdo con las situaciones que el árbol plantee.

Supóngase la siguiente situación: en la Figura 7.16 a) se muestra un árbol de orden 10, sobre el nodo *A* se elimina el elemento `FA`, y como solamente quedan tres elementos en el nodo, no se cumple la propiedad con respecto a la cantidad de elementos mínimos de un nodo, dado que en este ejemplo la mínima cantidad necesaria de elementos es cuatro. Entonces se debe llevar a cabo algún tipo de acción para corregir esta situación. Todas las acciones que sea posible realizar deben afectar a la menor cantidad posible de nodos del árbol, con la finalidad de mantener acotados los cambios y, por ende, la eficiencia del proceso. Así, se debe trabajar con los nodos adyacentes hermanos para poder subsanar el problema.

Figura 7.16

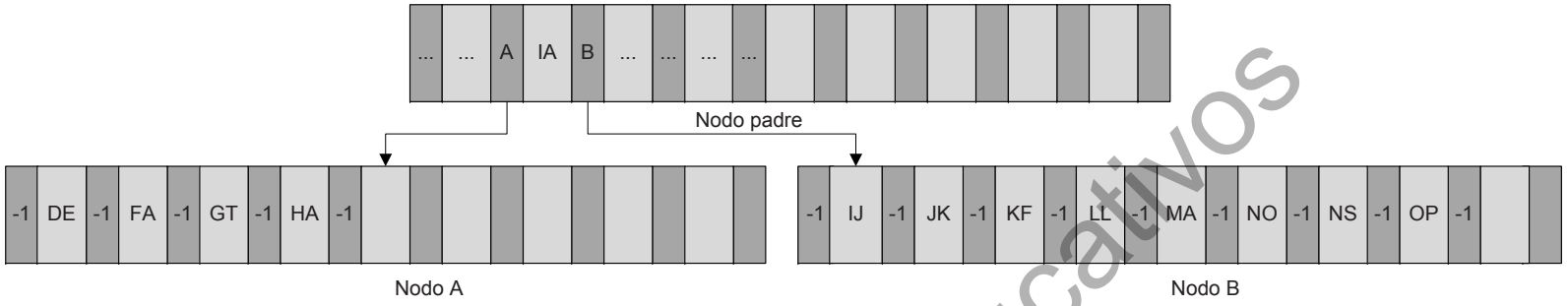


Figura 7.16 a)

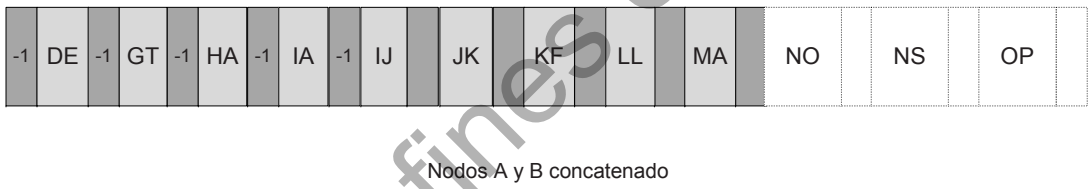


Figura 7.16 b)

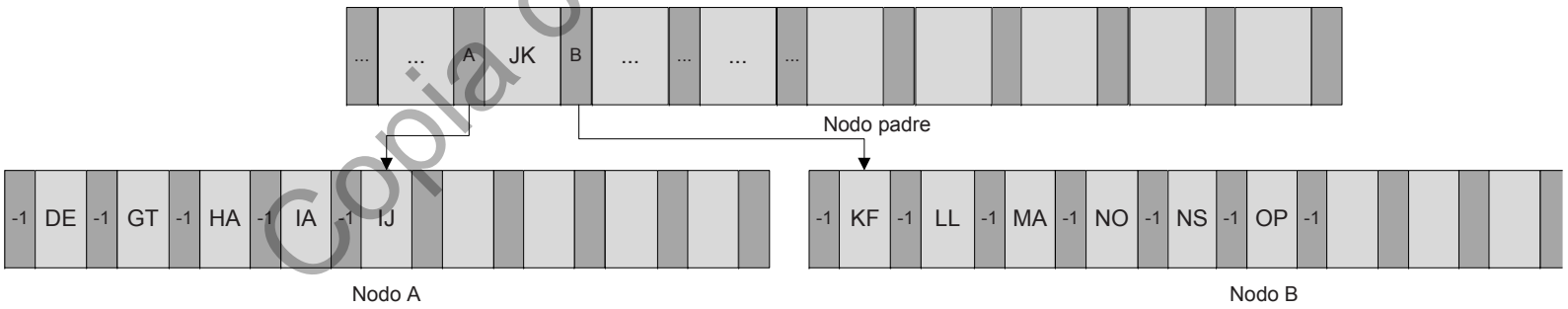


Figura 7.16 c)

La acción más inmediata que se puede plantear consiste en la opuesta a la división en un caso de *overflow*, la cual se puede denominar concatenación. Siguiendo este análisis, en la Figura 7.16 b), la opción consiste en concatenar el nodo *A* con el nodo *B*. Se debe notar que el nodo *B* contiene ocho elementos; por lo tanto, si se realiza la concatenación de ambos nodos, el resultado será un nodo que contendrá 12 elementos (tres del nodo *A*, ocho del nodo *B* y uno del nodo padre). Esta acción no es válida en este caso, ya que se superaría la máxima cantidad de elementos posibles para un nodo.

La alternativa, denominada redistribución, plantea utilizar algún nodo adyacente hermano del nodo conflictivo, permitiendo que dicho nodo adyacente hermano ceda elementos al nodo que presenta insuficiencia. Analizando la Figura 7.16 b), el nodo *B* dispone de ocho elementos; de aquí que es posible que le ceda al nodo *A* alguno de ellos. Este proceso significará que el nodo padre sea afectado. Se puede observar que si el elemento `IJ` pasara al nodo *A*, el elemento `IA`, contenido en el padre y utilizado como separador entre *A* y *B*, no sería de utilidad. Luego, como lo muestra la Figura 7.16 c), el proceso consiste en redistribuir entre los nodos *A* y *B* los elementos que conforman *A*, *B* y el padre. Así los cinco primeros (`DE, GT, HA, IA, IJ`) quedarán en el nodo *A*, los seis últimos quedarán en el nodo B (`KF, LL, MA, NO, NS, OP`) y el restante será ubicado en el padre (`JK`). Generalizando lo anterior: en una redistribución, la mitad de los elementos queda en un nodo, la otra mitad, en el segundo nodo, y se toma un elemento que actúe como separador y quede ubicado en el nodo padre.

Una ventaja que presenta la redistribución es que la cantidad de nodos no se ve afectada. Entonces, con solo cambiar de posición elementos entre tres nodos (dos hijos y el padre), el cambio sobre el árbol *B* queda acotado sin posibilidad de propagar cambios al resto de dicho árbol.

No obstante, la redistribución no siempre es posible. Para redistribuir se debe disponer de un nodo adyacente hermano con suficientes elementos para compartir. Obsérvese el caso planteado en la Figura 7.17, partiendo de la base del mismo árbol original de la Figura 7.16. Aquí se borrará nuevamente el elemento `FA` del nodo *A*, pero ahora el nodo *B* solo dispone de cuatro elementos. Al intentar redistribuir, el nodo *A* o el nodo *B* mantendrán su insuficiencia. En este caso, no es posible realizar una redistribución. La única operación viable resulta la concatenación, es decir, tomar los elementos del nodo *A*, juntarlos con los elementos del nodo *B* y traer a ese nuevo nodo el elemento del padre, dado que al juntar *A* y *B* no será necesario utilizar un separador. La Figura 7.17 b) plantea el caso descripto.

Se debe notar que la operación de concatenación puede propagar cambios a lo largo del árbol. En el ejemplo planteado, el nodo padre (que no es la raíz del árbol) deja de tener un elemento. Entonces puede ocurrir que el nodo padre genere insuficiencia. Este caso deberá ser resuelto como se planteara anteriormente. Si hubiera insuficiencia en el nodo padre, se procederá a redistribuir con algún adyacente hermano de este, y si no se pudiera realizar esta operación de redistribución, se aplicará la operación de concatenación. Así, la concatenación puede propagarse hacia arriba en el árbol *B*, obligando a modificar la raíz, disminuyendo en uno la cantidad de niveles del árbol.

Figura 7.17

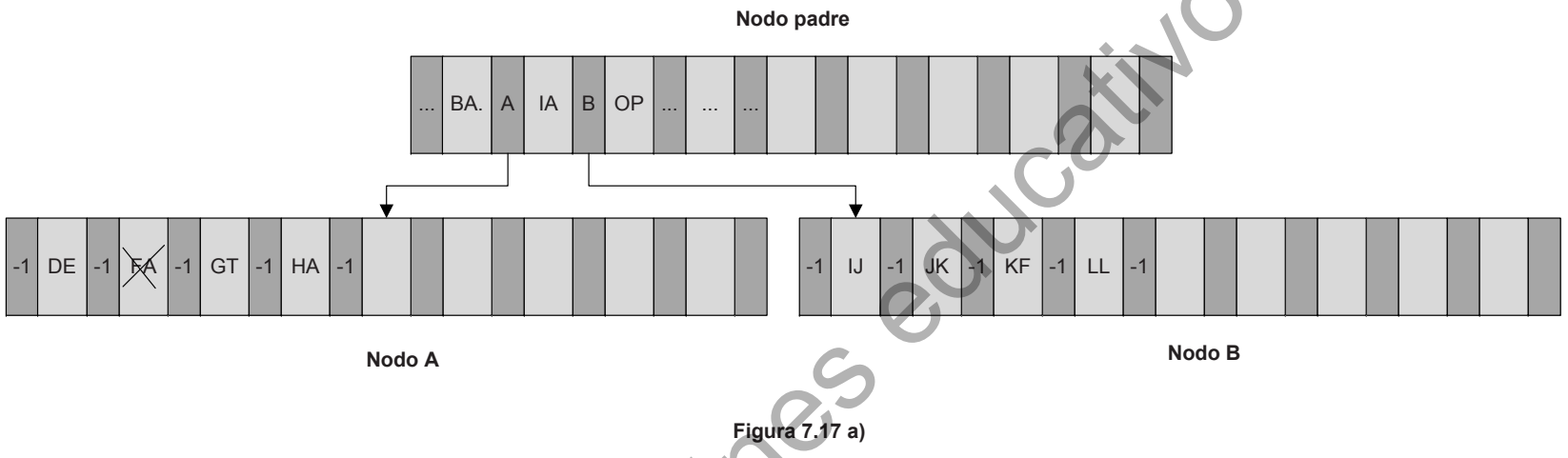


Figura 7.17 a)

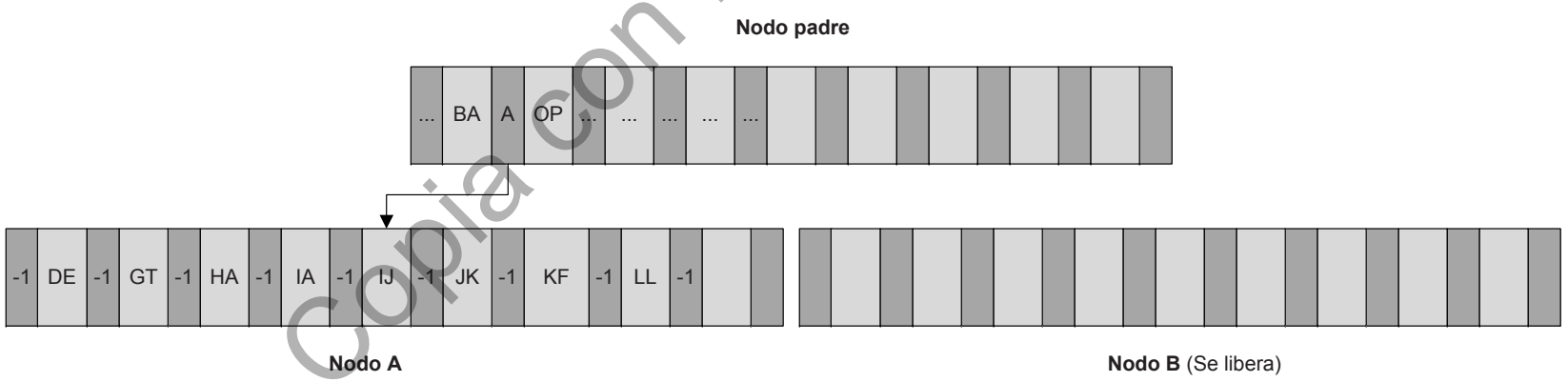


Figura 7.17 b)

Eficiencia de eliminación en un árbol *B*

De la misma forma que se realizó el estudio de eficiencia en la inserción y la búsqueda, se presenta el análisis de eficiencia en la eliminación.

Aquí se puede comenzar por el mejor caso. Así, se intenta eliminar un elemento que está en un nodo terminal y cuyo borrado no genera insuficiencia. Entonces serán necesarias tantas lecturas como niveles tenga el árbol y una sola escritura (la correspondiente al nodo terminal sin el elemento borrado).

El peor caso quedará representado cuando la operación de borrado necesite concatenar (recuerde el lector que la redistribución acota los efectos a tres nodos). La concatenación implica leer un nodo adyacente hermano. Por cada nivel que se deba concatenar habrá dos lecturas, salvo en el nodo raíz, el cual carece de hermanos. La cantidad de escrituras se limita a una por nivel.

Resumiendo (siendo *H* la cantidad de niveles):

	Mejor caso	Peor caso
Cantidad de lecturas	*H*	2 * *H* – 1
Cantidad de escrituras	1	*H* – 1

A partir del análisis de eficiencia de la inserción, y teniendo en cuenta lo discutido en el Capítulo 4 sobre la cantidad de operaciones de baja que se registran sobre archivos, el estudio de eficiencia de la eliminación plantea una situación muy beneficiosa.

Modificación en árboles *B*

La restante operación que se debe analizar es la operación de modificación de un elemento contenido en un árbol *B*. Hasta aquí, se analizó la eficiencia de operaciones de inserción y eliminación, y se determinó que los resultados obtenidos son positivos.

La opción más simple para tratar un caso de modificación es proceder tomando un cambio como una baja del elemento anterior y un alta del nuevo elemento. Esta consideración no requiere mayor análisis de operatividad y tiene como resultado, en el análisis de *performance*, el obtenido de sumar una operación de baja seguida de una de alta. En el peor caso, dicha situación se mantiene acotada en el tiempo de respuesta.

Algunas conclusiones sobre árboles *B*

De acuerdo con lo planteado hasta el momento, los árboles balanceados representan una buena solución como estructura de datos, para implementar el manejo de índices asociados a archivos de datos.

Se debe notar que cualquier operación (consulta, alta, baja o modificación) se realiza en términos aceptables de *performance*. Se puede considerar, entonces, que este tipo de estructura representa una solución viable.

Se debe tener siempre en cuenta que las estructuras de árboles *B* serán utilizadas para administrar los índices asociados a claves primarias, candidatas o secundarias. Dichos archivos de índices contendrán la estructura planteada, clave, hijos y referencia del resto del registro en el archivo de datos original. El archivo de datos original se plantea como un archivo serie donde cada elemento se inserta siempre al final.

Como se definió en el Capítulo 5 para índices secundarios, tanto estos índices como aquellos que almacenan claves candidatas referencian a la clave primaria, en lugar de referenciar directamente a la posición física respectiva en el archivo de datos. Esto se debe a cuestiones de *performance*.

En caso de modificar la posición física de un registro, solo debe modificarse un índice, el correspondiente a la clave primaria.

De modificarse la clave primaria, deben modificarse el resto de los índices, pero como se explicará con detalle en el Capítulo 12, al diseñar un modelo de datos, en general las claves primarias se definen bajo la pauta de “nunca serán modificadas”.

Árboles *B**

Los árboles *B** representan una variante sobre los árboles *B*. La consideración efectuada para avanzar sobre una estructura *B** parte del análisis realizado en el proceso de baja de información. Resumiendo este caso, el tratamiento de una saturación en un árbol *B* genera una sola operación, la división. El tratamiento de una insuficiencia genera dos operaciones, redistribución y/o concatenación. Claramente se puede observar que división y concatenación son operaciones opuestas. Entonces, el proceso de tratamiento de saturaciones en un árbol *B* no plantea una operación equivalente a la redistribución en el proceso de baja.

Precisamente, la algorítmica que planteara Knuth define una alternativa para los casos de *overflow*. Así, antes de dividir y generar nuevos nodos se dispone de una variante, redistribuir también ante una saturación. Esta acción demorará la generación de nuevos nodos y, por ende, tendrá el efecto de aumentar en forma más lenta la cantidad de niveles del árbol. Si la cantidad de niveles del árbol crece más lentamente, la *performance* final de la estructura es mejor.

Si se aplica el concepto de redistribuir, cuando un nodo se completa, reubica sus elementos utilizando un nodo adyacente hermano. Cuando no sea posible esta redistribución, se estará ante una situación donde tanto el nodo que genera *overflow* como su adyacente hermano están completos. Esto abre la posibilidad de dividir partiendo de dos nodos completos y generando tres nodos completos en dos terceras partes (2/3).

El aspecto más importante de la división planteada para un árbol *B** es que produce nodos completos en 2/3, en vez de nodos con solo la mitad de los elementos como planteaba un árbol *B*.

Un **árbol *B**** es un árbol balanceado con las siguientes propiedades especiales:

- 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, [(2*M*-1)/3] -1 elementos, y como máximo, *M*-1 elementos.
- Los nodos que no son terminales ni raíz tienen, como mínimo, [(2*M*-1) / 3] descendientes.
- Todos los nodos terminales se encuentran al mismo nivel.

Si se analizan las propiedades anteriores, se puede observar que cuando el árbol *B** tiene un solo nivel (la raíz) y este nivel se completa, no está disponible la posibilidad de redistribuir, dado que la raíz no tiene hermanos. Entonces, la propiedad que indica la mínima cantidad de elementos tiene una excepción de tratamiento, al dividir el nodo raíz y generar dos hijos. En este caso, los dos nodos hijos generados solamente completarán su espacio a la mitad. Esta situación se considera como la única excepción admitida para el tratamiento de las propiedades de árboles *B**.

La operación de búsqueda sobre un árbol *B** es similar a la presentada anteriormente para árboles *B*. La naturaleza de ambos árboles para localizar un elemento no presenta diferencias de tratamiento. La búsqueda comienza en el nodo raíz y se avanza hasta encontrar el elemento, o hasta llegar a un nodo terminal y no poder continuar con el proceso.

La operación de baja resulta nuevamente similar, tanto para los casos donde no se genera *underflow* como para aquellos donde sí se genera insuficiencia. En este último caso, la primera opción consiste en redistribuir, o en su defecto se deberá concatenar, basados en los principios definidos para árboles *B*. Si bien existen algunas variantes posibles, no es objetivo de este libro profundizar su tratamiento.

Operaciones de inserción sobre árboles *B**

La operación de inserción debe ser discutida con detalle. Las diferentes variantes de inserción tienen que ver, básicamente, con alternativas propuestas en los casos de redistribución.

El proceso de inserción en un árbol *B** puede ser regulado de acuerdo con tres políticas básicas: política de un lado, política de un lado u otro lado, política de un lado y otro lado.

Así, cada política determina, en caso de *overflow*, el nodo adyacente hermano a tener en cuenta. La política de un lado determina que el nodo adyacente hermano considerado será uno solo, definiendo la política de izquierda, o en su defecto, la política de derecha. En caso de completar un nodo, intenta redistribuir con el hermano indicado. En caso de no ser posible porque el hermano también está completo, tanto el nodo que genera *overflow* como dicho hermano son divididos de dos nodos llenos a tres nodos dos tercios llenos.

Figura 7.18

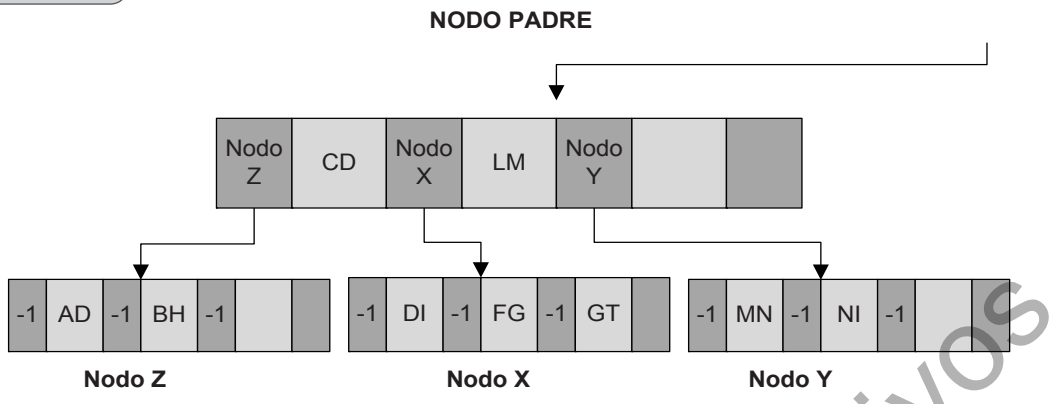


Figura 7.18 a)

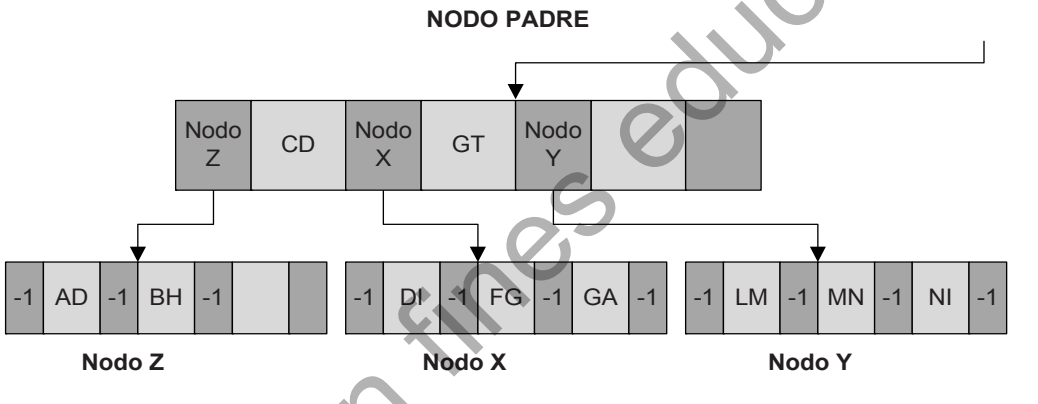


Figura 7.18 b)

La Figura 7.18 presenta un ejemplo resumido de la política de un lado, utilizando el hermano de la derecha. Aquí, teniendo un árbol *B** de orden 4, se puede observar que el nodo *X* está completo [Figura 7.18 a)] y se debe insertar el elemento `GA`; como dicho elemento no tiene lugar, se intenta redistribuir con el nodo adyacente hermano *Y*, como lo muestra la Figura 7.18 b). Se debe tener en cuenta que se han reacomodado los elementos de los nodos *X* e *Y*, así como el elemento separador en el nodo padre.

Utilizando el mismo ejemplo se pueden hacer varias consideraciones. Si llegase el elemento `ZT`, el nodo *Y* se encuentra completo, generando un *overflow*. El inconveniente que se genera está dado porque el nodo *Y* es el nodo ubicado más a la derecha del padre. Entonces, dicho nodo carece de hermano derecho. En esta situación, y solo en esta situación, es válido aplicar redistribución con el hermano izquierdo, pero no es posible aplicar dicha redistribución pues el nodo *X* está completo. En este caso, nuevamente tanto el nodo que genera *overflow* como dicho hermano son divididos de dos nodos llenos a tres nodos dos tercios llenos.

Figura 7.19

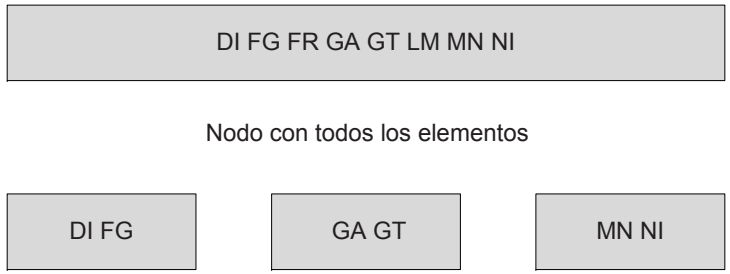


Figura 7.19 a)

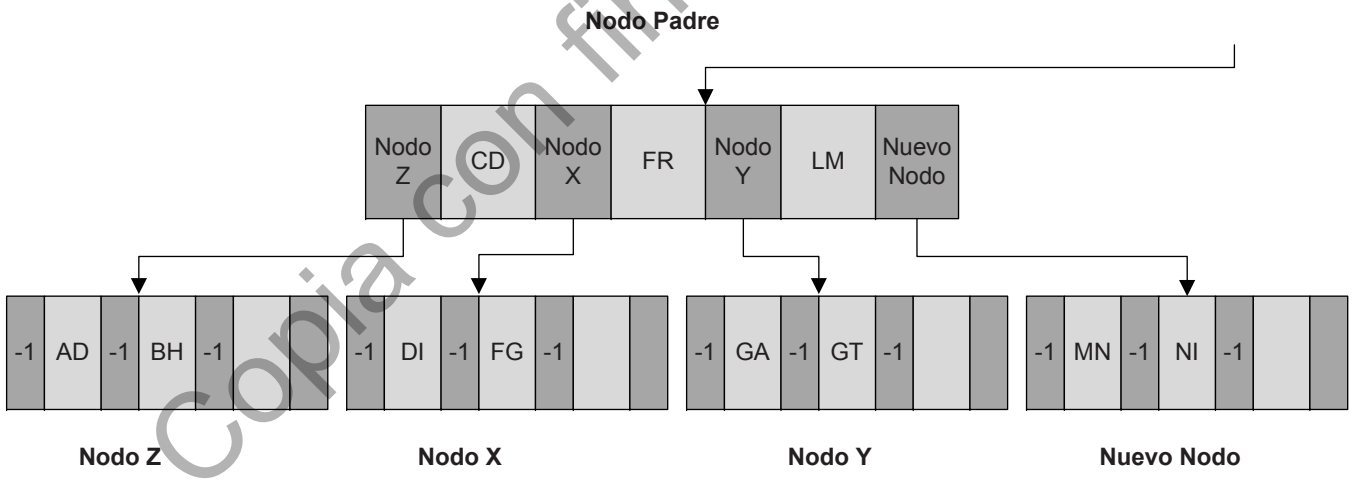


Figura 7.19 b)

Otra situación para ser considerada es la siguiente: suponga que el elemento a insertar en el árbol de la Figura 7.18 b) es `FR`. El nodo *X* está completo, no se puede redistribuir con el nodo hermano *Y* pues también está completo y, si bien el nodo *Z* dispone de lugar para redistribuir, al tratarse de la política de un lado, derecha, algorítmicamente no se puede redistribuir con *Z*. Entonces, como lo muestra la Figura 7.19 a), a partir de los nodos *X*, *Y* y de los elementos separadores ubicados en el nodo padre, se genera temporalmente un nodo con suficiente capacidad para contener a todos los elementos. Este nodo se debe dividir en tres nodos con dos tercios llenos cada uno. Por último, a partir de la división mostrada en la Figura 7.19 a), el árbol queda como el presentado en la Figura 7.19 b).

La política de un hermano adyacente o el otro hermano adyacente representa una alternativa al caso anterior. Aquí, 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 dos nodos llenos a tres nodos dos tercios llenos. La Figura 7.20 a) expone, bajo el ejemplo definido anteriormente, la situación planteada; la política a utilizar es derecha o izquierda. En este caso, el nodo *X* contiene tres elementos y se debe insertar el elemento `GA`, se genera una saturación y, como la política planteada es primero derecha luego izquierda, se intenta redistribuir con el nodo *Y* (adyacente hermano de la derecha), como lo presenta la Figura 7.20 b). Suponga que luego se inserta el elemento `TY`; como el nodo *Y* tiene capacidad suficiente, no se genera saturación.

La Figura 7.20 c) muestra cómo queda el árbol *B** luego de introducir la clave `EM`; el nodo *X* produce saturación, el nodo *Y* no acepta redistribuir y, entonces, se redistribuye con el nodo *Z* (adyacente hermano de la izquierda).

Suponga ahora que se debe insertar `DZ`; los tres nodos terminales *X*, *Y* y *Z* están completos. Por lo tanto, no es posible redistribuir. Esta política determina que se debe tomar el nodo *X* y uno de sus adyacentes hermanos y proceder con la división. Siguiendo la política de derecha o izquierda presentada en este problema, se realizaría una división como se explicó en la Figura 7.19, utilizando los nodos *X* e *Y*.

La última política alternativa plantea ambos adyacentes hermanos. Aquí, la forma de trabajo es similar al caso 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 tres cuartas partes completas cada uno. Si bien esta alternativa completa más cada nodo del

Figura 7.20

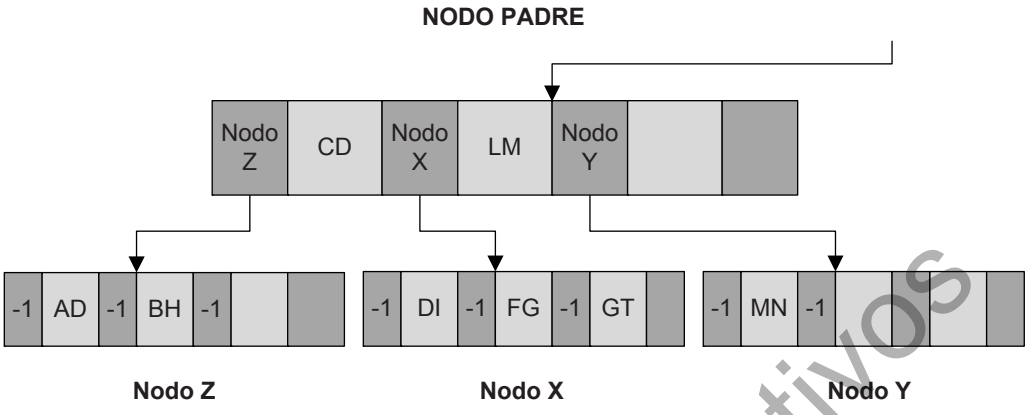


Figura 7.20 a)

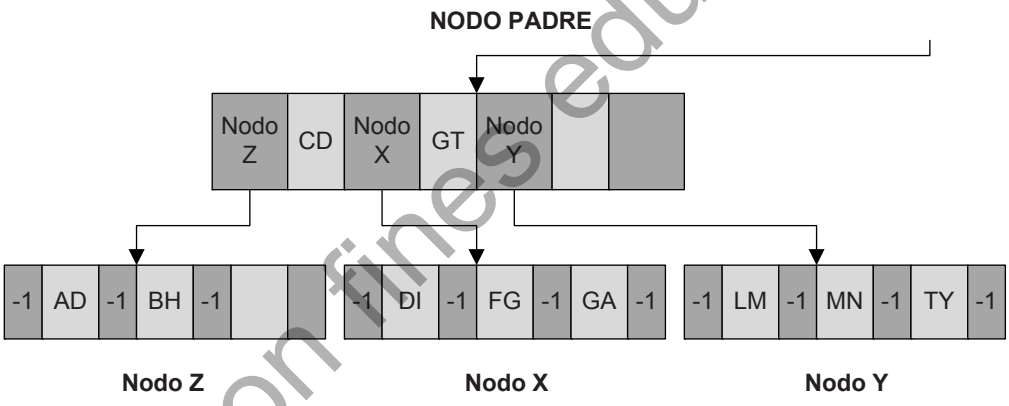


Figura 7.20 b)

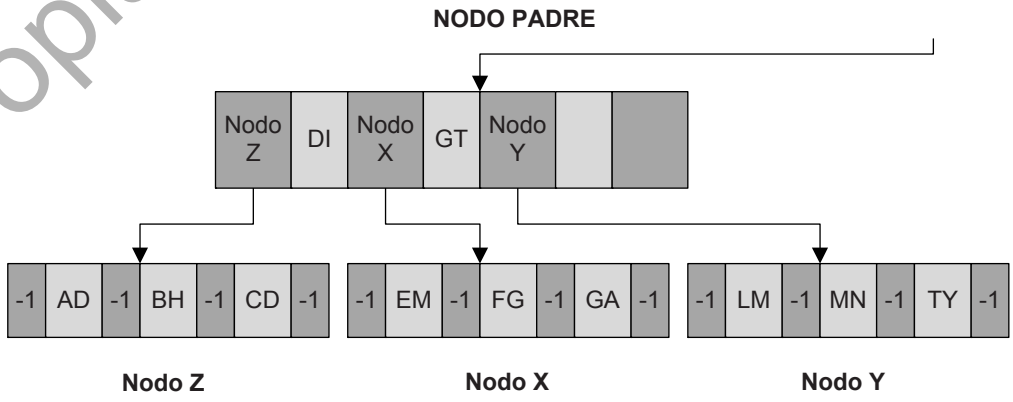


Figura 7.20 c)

árbol, y por consiguiente, genera árboles de menor altura, necesita de mayor número de operaciones de entrada-salida sobre disco para poder ser implementada. No es objetivo del presente material profundizar al respecto.

Análisis de *performance* de inserción en árboles *B**

La *performance* resultante de la inserción sobre árboles *B** dependerá de cada política. Así, ante la ocurrencia de *overflow*, como mínimo cada una de las políticas requiere dos lecturas (el nodo que se satura y un adyacente hermano) y tres escrituras (cada nodo hijo y el padre). Se debe considerar que 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, la política de un lado necesita a su vez realizar cuatro escrituras (el nodo padre, los dos nodos que estaban completos y el nuevo que se genera). Es la misma situación generada por la política de un lado o el otro lado, dado que en este caso se divide nuevamente de dos nodos completos a tres nodos.

Por último, la política de un lado y el otro lado genera cuatro escrituras, dado que, además de involucrar a los tres nodos terminales, se deberán tener en cuenta el nodo padre y el nuevo generado.

Manejo de *buffers*. Árboles *B* virtuales

Como se discutió hasta el momento, los árboles *B* representan una muy eficiente estructura de almacenamiento para índices. El análisis de *performance* de las operaciones determina que el resultado obtenido depende directamente de la altura del árbol. Se ha mostrado que es posible generar árboles que contengan un número importante de elementos manteniendo la altura acotada.

Se debe tener en cuenta, además, que si un árbol *B* tiene *H* niveles, esto no significa que se necesite esa misma cantidad de accesos para recuperar un elemento.

Cuando se decide la capacidad de cada nodo, esta capacidad está determinada básicamente por la cantidad de información que podrá manejar el SO, a través del uso de *buffers*. Así, el concepto de *buffer* y el de nodo están íntegramente relacionados.

Se debe tener en cuenta que un SO administra un número importante de *buffers* y que, de acuerdo con la política de trabajo, es posible que

aquellos *buffers* más utilizados se mantengan en memoria principal, a fin de minimizar el número de accesos a disco. Una de las políticas más utilizadas se conoce como LRU, que, por sus siglas en inglés, significa "último más recientemente utilizado". Con esta política se mantienen en memoria principal los nodos que más se han visitado en forma reciente.

Un árbol *B* que administre un índice de acceso muy frecuente significará para el SO que el *buffer* que contenga el nodo raíz sea muy utilizado. Entonces, el SO tenderá a mantener dicho *buffer* en memoria, disminuyendo en uno el número de accesos a disco, requerido para recuperar la información.

En general, empíricamente es posible afirmar que si se dispone de un árbol de tres niveles, la primera búsqueda de un elemento demandará a lo sumo tres accesos a disco; la segunda búsqueda demandará a lo sumo dos accesos a disco (en este caso, la raíz ya se mantiene en memoria principal). Ahora, buscar cinco elementos requerirá en promedio 1.71 accesos al disco, disminuyendo la cantidad de accesos requeridos a medida que se intente localizar mayor información.

Esta última apreciación, si bien es solamente un análisis empírico, muestra una característica extra para los árboles *B* que hace más eficiente su utilización.

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.

Este nuevo tratamiento propuesto para archivos intenta compatibilizar el orden físico de los elementos con un acceso indizado, de acuerdo con lo definido para árboles *B*.

Suponga el lector que se dispone de un archivo con *N* registros que fueron organizados físicamente por orden de llegada, pero que se creó una estructura de índice utilizando un árbol *B* o *B** que permite la localización rápida de la información. Si se desea un tratamiento

secuencial de los elementos del archivo utilizando el orden de la llave, la única forma de extraerlos en orden es utilizando el índice (árbol *B*). Para recuperar los *N* registros en orden, es necesario recorrer todos los nodos del árbol a través de los punteros definidos. Esto significa acceder a un nodo terminal, volver a su padre y acceder al nodo terminal siguiente, repitiendo este proceso para todo el árbol. Esto resulta en un algoritmo mucho menos eficiente que tener la misma estructura ordenada físicamente.

Si el archivo requiere mucho acceso secuencial utilizando el índice (almacenado en un árbol *B*, por ejemplo, para obtener algún tipo de listado o presentar la información en pantalla), la solución resulta muy ineficiente.

Por otro lado, si se plantea la alternativa de mantener el archivo físicamente ordenado, se soluciona el problema de acceso anterior pero resulta inaceptable, al momento de acceder para recuperar un dato, realizar una inserción o una eliminación.

Estos problemas necesitan una solución alternativa que permita compatibilizar ambos accesos.

Archivos físicamente ordenados a bajo costo

El problema definido tiene claramente dos partes establecidas: el acceso secuencial indizado (árbol) y el orden físico de los elementos. Hasta el momento, en el capítulo se han discutido opciones para resolver el acceso de manera eficiente. Este apartado presentará una alternativa de orden lógico secuencial a bajo costo.

En el Capítulo 5, se analizó el costo del ordenamiento físico de un archivo y el mismo resultó extremadamente ineficiente. Por este motivo, la solución a los archivos físicamente ordenados tiene que provenir de su creación y administración, sin forzar el reordenamiento ante cada operación de inserción.

Como se discutió anteriormente, el nodo pasa a ser la unidad de entrada-salida, es decir, todos los elementos de un nodo son leídos consecutivamente y el nodo luego es almacenado en un *buffer*; o viceversa, se toman los elementos del nodo de un *buffer* y se escriben dichos elementos consecutivamente a disco. Por este motivo, cada vez que se accede a un nodo, debe contabilizarse un acceso a disco, independientemente de la cantidad de registros contenidos en él.

La Figura 7.21 a) muestra el contenido de un nodo o bloque, con capacidad para cinco registros, donde se almacenan los nombres de cinco alumnos de una cátedra de la Facultad de Informática. Supóngase

ahora que dicho archivo tiene dos nodos; como lo muestra la Figura 7.21 b), para poder mantener el orden entre los elementos, ambos nodos deben estar enlazados.

La pregunta a realizar es: ¿cómo se llegó a tener el archivo ordenado?, o ¿qué sucede si llega una clave nueva, por ejemplo, `Estevez`, a insertar en el primer nodo y no tiene espacio?

La respuesta a la pregunta es tratar cada nodo o bloque como si fuera un nodo terminal de un árbol *B*. Así, al llegar Estevez, debería insertarse entre Detomaso y Fernandez, para mantener el orden. Al no tener capacidad suficiente, el nodo debe ser dividido, de acuerdo con lo planteado en la algorítmica de árboles *B*.

La Figura 7.21 c) presenta dicha situación; se crea un nuevo nodo, donde se redistribuyen los elementos. Para mantener el orden del archivo, se modifican los enlaces entre los nodos.

Se debe notar que la *performance* de esta operación se limita a dos escrituras, el nodo existente modificado y el nuevo nodo, similar a la *performance* de una división con árboles *B*.

Figura 7.21

Figura 7.21 a)

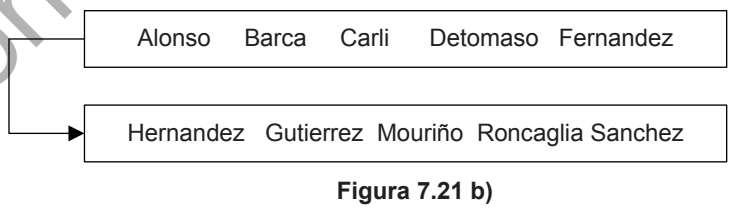


Figura 7.21 b)

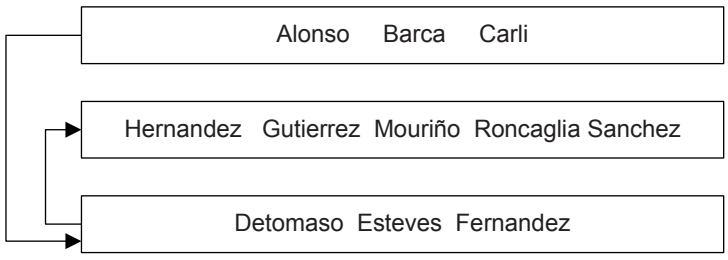


Figura 7.21 c)

La eliminación de registros dentro del archivo puede ocasionar que un nodo tenga "pocos" elementos. Nuevamente, si se piensa en la operación de baja sobre un árbol *B*, un nodo que presenta insuficiencia puede ser concatenado o fusionado con el nodo hermano adyacente.

Esta solución presenta una ventaja concreta. Si bien el archivo no se encuentra físicamente ordenado, cada bloque sí lo está. Y, además, como cada nodo se encuentra enlazado con el siguiente, es posible realizar un recorrido secuencial ordenado a muy bajo costo en términos de accesos a disco.

El último análisis a realizar tiene que ver con la elección del tamaño del nodo. Nuevamente, dicho tamaño dependerá de la capacidad del *buffer* de entrada-salida del SO, y de la información que se desea almacenar para cada registro (la llave o clave más la dirección física del resto de la información del registro en el archivo de datos).

Árboles *B*+

Se ha discutido una solución de bajo costo, para recuperar los elementos en forma ordenada de un archivo, sin necesidad de reacomodamientos físicos costosos.

Ahora se debe encontrar un mecanismo que permita localizar los datos contenidos en los nodos, a bajo costo. La solución que presentaban los árboles *B* y *B** debería poder aplicarse en este entorno.

La estructura intermedia resultante se denomina árbol *B*+ e incorpora las características discutidas para árboles *B*, además del tratamiento secuencial ordenado del archivo. Así, se podrán realizar búsquedas aleatorias rápidas de información, en conjunto con acceso secuencial eficiente.

Un **árbol *B*+** es un árbol multicamino con las siguientes propiedades:

- 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.

Si se compara la definición anterior con la resultante para árboles *B*, se encontrarán similitudes, excepto en la última propiedad.

Es esta última propiedad la que 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. Así, los árboles *B*+ diferencian los elementos que constituyen datos de aquellos que son separadores.

Al comenzar la creación de un árbol *B*+, el único nodo disponible, el nodo raíz, actúa tanto como punto de partida para búsquedas como para acceso secuencial [Figura 7.22 a)].

En el momento en que el nodo se satura (en la figura, el árbol se supone de orden 5), se produce una división. Se genera un nuevo nodo terminal donde se redistribuyen los elementos.

Como lo muestra la Figura 7.22 b), ahora se dispone de tres nodos, los nodos terminales contendrán todos los elementos y el nodo raíz contendrá el elemento que actúa como separador. Se debe notar que la clave utilizada como separador es la misma que está contenida en el nodo terminal, es decir, se utiliza una copia del elemento y no el elemento en sí.

Figura 7.22

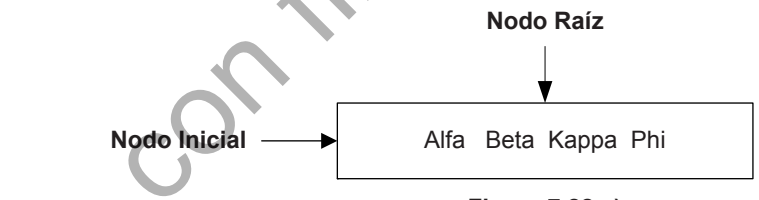


Figura 7.22 a)

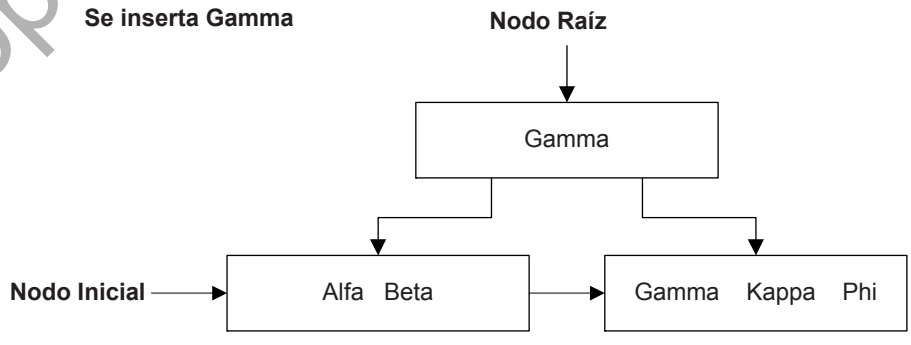


Figura 7.22 b)

El proceso de creación del árbol *B*+ sigue los lineamientos discutidos anteriormente para árboles *B*.

Los elementos siempre se insertan en nodos terminales. Si se produce una saturación, el nodo se divide y se promociona una copia (aquí está la diferencia) del menor de los elementos mayores, hacia el nodo padre. Si el padre no tiene espacio para contenerlo, se dividirá nuevamente.

Se debe notar que, en caso de dividir un nodo no terminal, se debe promocionar hacia el padre el elemento en sí y no una copia del mismo, es decir, solo ante la división de un nodo terminal se debe promocionar una copia.

Para borrar un elemento de un árbol *B*+, siempre se borra de un nodo terminal, y si hubiese una copia de ese elemento en un nodo no terminal, esta copia se mantendría porque sigue actuando como separador.

Árboles *B*+ de prefijos simples

El agregado de la opción `prefijos simples` a un árbol *B*+ intenta aprovechar mejor el uso de espacio físico.

Se puede observar en la Figura 7.22 b) que al producirse una división se ha promocionado la menor de las claves mayores, Gamma. También, puede observarse que en este caso no es necesario promocionar Gamma, ya que con la letra G alcanzaría para actuar como separador.

Un **árbol *B*+ de prefijos simples** es un árbol *B*+ donde los separadores están representados por la mínima expresión posible de la clave, que permita decidir si la búsqueda se realiza hacia la izquierda o hacia la derecha.

Las Figuras 7.23 a) y b) presentan dos ejemplos de división donde el árbol tiene la política de prefijos simples. En un caso, alcanza con la letra G como separador, pero en el otro es necesario que el separador sea Gon. El separador siempre tiene la menor cantidad de caracteres posible, de modo de poder actuar como tal entre los nodos hijos.

Figura 7.23

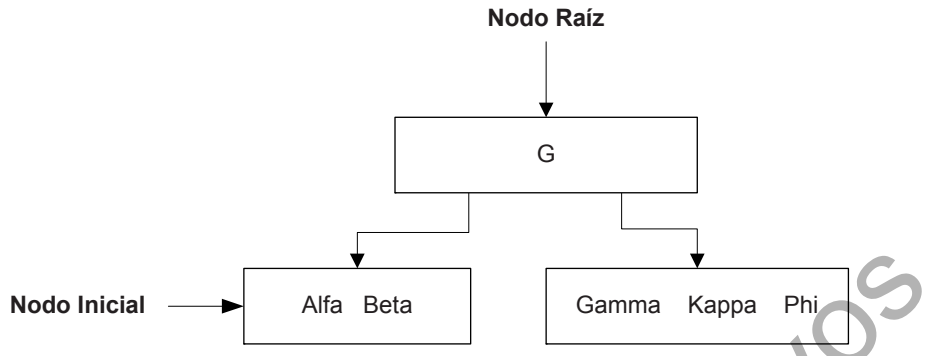


Figura 7.23 a)

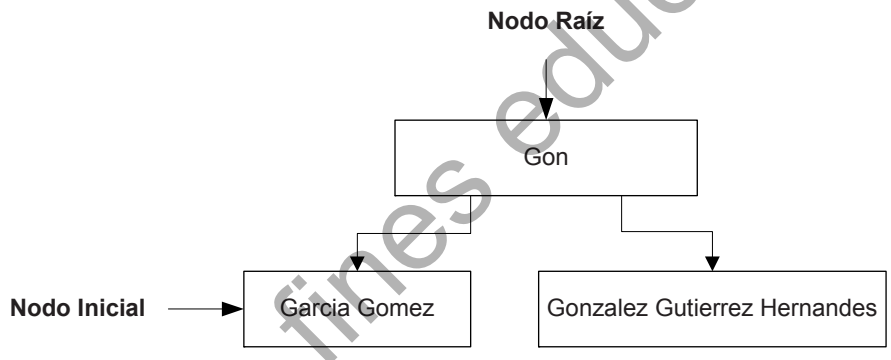


Figura 7.23 b)

Árboles balanceados. Conclusiones

La familia de los árboles balanceados son estructuras muy poderosas y flexibles para la administración de índices asociados a archivos de datos, con un nivel de *performance* muy interesante. Sin embargo, no deben considerarse como la única solución posible a todos los problemas.

La mejor estructura para un archivo va a depender del archivo en sí y del propósito de uso de dicho archivo. Es decir, si el archivo de datos se limita a unos pocos registros que pueden estar contenidos en un nodo, utilizar un árbol no es necesario, dado que agrega un nivel de indirección por cada índice generado. Tampoco es necesario si el archivo no requiere consultarse con frecuencia.

Como se discutirá más adelante, hay metodologías para organizar archivos que presentan un mejor tiempo de respuesta. Entonces, ante determinadas necesidades de *performance*, puede ocurrir que los árboles balanceados no cubran las expectativas de un problema particular.

Sin embargo, la familia de árboles *B* tiene una gran aplicabilidad en situaciones en las que resulta necesario acceder a un archivo tanto para búsquedas aleatorias eficientes como para acceso secuencial.

Las características compartidas entre los árboles de la familia *B* son las siguientes:

- Manejo de bloques o nodos de trabajo; esto facilita las operaciones de entrada-salida y mantiene acotado el número de accesos para realizar dichas operaciones.
- Todos los nodos terminales están a la misma distancia del nodo raíz, es decir, estos árboles están balanceados.
- Crecen de abajo hacia arriba, a diferencia de los árboles binarios comunes.
- Son “bajos” y “anchos”, a diferencia de los binarios, que son “altos” y “delgados”.
- Es posible implementar algoritmos que manipulen registros de longitud variable para, de esta forma, mejorar aun más la *performance* y el uso del espacio.

En particular, los árboles *B** mejoran la eficiencia de los árboles *B*, aumentando la cantidad mínima de elementos en cada nodo y, por consiguiente, ayudando a disminuir la altura final del árbol. El costo que se debe pagar en este caso es que las operaciones de inserción resultan un poco más lentas.

Por último, en los árboles *B*+, toda la información de las claves está contenida en los nodos terminales. Los nodos no terminales actúan como separadores, y poseen una copia de los elementos contenidos en los nodos terminales.

Cuestionario del capítulo

1. ¿Qué características principales presenta una estructura de árbol balanceada sobre un árbol binario?
2. ¿Por qué es importante el concepto de orden en un árbol multicamino? ¿La respuesta anterior es la misma si el árbol es multicamino balanceado?
3. ¿Por qué la cantidad de niveles de un árbol es importante para el estudio de la *performance* de las operaciones de búsqueda, inserción y borrado?
4. ¿Cómo afecta el orden del árbol a la cantidad de niveles de dicho árbol?
5. ¿Qué características diferentes presenta un árbol *B** respecto de un árbol *B*?
6. ¿Cómo mejora la *performance* de búsqueda en un árbol *B** respecto de un árbol *B*?
7. ¿Cuáles son los costos de generar una estructura *B** en lugar de una estructura *B*?
8. ¿Qué aspectos hacen necesario utilizar árboles *B*+?
9. ¿Qué se puede decir de la *performance* de búsqueda en un árbol B+ respecto de las estructuras *B** o *B* común?
10. ¿Qué significa utilizar un árbol *B*+ de prefijos simples?

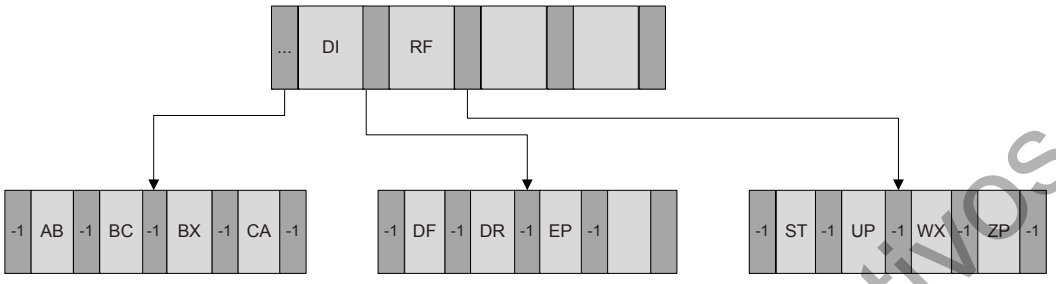
Ejercitación

1. Dado el siguiente conjunto de claves, genere un árbol *B* de orden 5 y muestre cómo se construye el archivo respectivo: `QW RT NB AS FG DF JH OP ZX BB YT LA CA WS MZ UI AW PN FR HL XC BT`.
2. Genere, a partir de las claves definidas en el ejercicio anterior, un árbol *B**:
   a. Con política de derecha.
   b. Con política de izquierda o derecha.
   c. Con política de derecha e izquierda.
3. A partir de la Figura 7.24, y sabiendo que corresponde a un árbol B de orden 5:
   a. Inserte `CV`. b. Inserte `XX`. c. Borre `HI`.

Ejercitación

4. Suponga ahora que la Figura 7.24 representa a un árbol B* de orden 5.

Figura 7.24



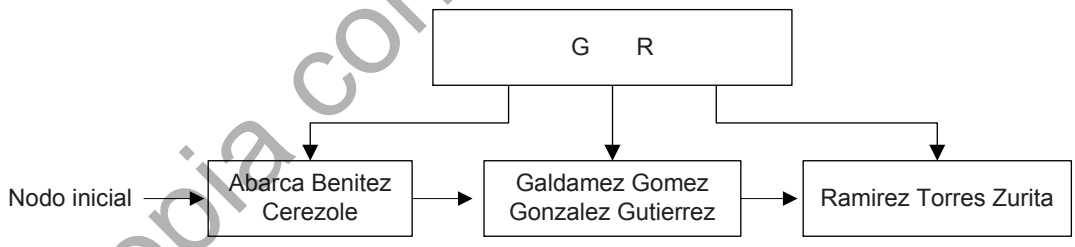
Resuelva los problemas del ejercicio anterior suponiendo que:

a. Se trabaja con la política de izquierda.

b. Se trabaja con la política de izquierda o derecha.

5. Dado el árbol de la Figura 7.25 y sabiendo que el mismo es un árbol B+ de prefijos simples de orden 4, inserte las siguientes claves: García, Hernández, Molinari, Perez.

Figura 7.25



CAPÍTULO 8

Dispersión (*hashing*)

Objetivo

Un archivo directo, o con acceso directo, es un archivo en el cual cualquier registro puede ser accedido sin acceder antes a otros registros, es decir, cualquier registro puede ser accedido directamente. En un archivo serie, en cambio, un registro está disponible solo cuando el registro predecesor fue procesado. Resulta interesante pensar a un archivo directo como un arreglo de registros, en tanto que un archivo serie puede pensarse como una lista de registros.

Los archivos directos son almacenados en disco en diferentes formas respecto de los archivos serie. Esta forma de almacenamiento debe permitir una rápida recuperación de la información contenida en el archivo, que es el objetivo principal que se persigue con un archivo directo.

El mecanismo que trata de asegurar una recuperación rápida de registros, en un solo acceso promedio, lleva el nombre de dispersión o *hashing*. En este capítulo, se presentan las alternativas más comunes de dispersión: dispersión con espacio de direccionamiento estático y dispersión con espacio de direccionamiento dinámico. Además, se analizan con detalle las propiedades fundamentales del método y se desarrolla un estudio numérico que permite asegurar la localización rápida de un registro en particular.

Conceptos de dispersión. Métodos de búsqueda más eficientes

Suponga que se necesita acceder a un archivo de datos, y que para la rápida localización de los registros allí contenidos se genera un árbol *B+* de prefijos simples, como los estudiados en el Capítulo 7. Siguiendo el análisis numérico efectuado, en promedio son necesarios entre tres y cuatro accesos para recuperar un registro. Si bien esta *performance* fue considerada muy interesante en el capítulo anterior, se podría presentar alguna situación en la práctica donde sea ineficiente consumir cuatro accesos a memoria secundaria para recuperar un registro.

Entonces, ¿cuál es la alternativa disponible? Debe existir un mecanismo que permita reducir el número de accesos a disco.

Hashing o dispersión es un método que mejora la eficiencia obtenida con árboles balanceados, asegurando en promedio un acceso para recuperar la información.

Se presentan a continuación definiciones para el método:

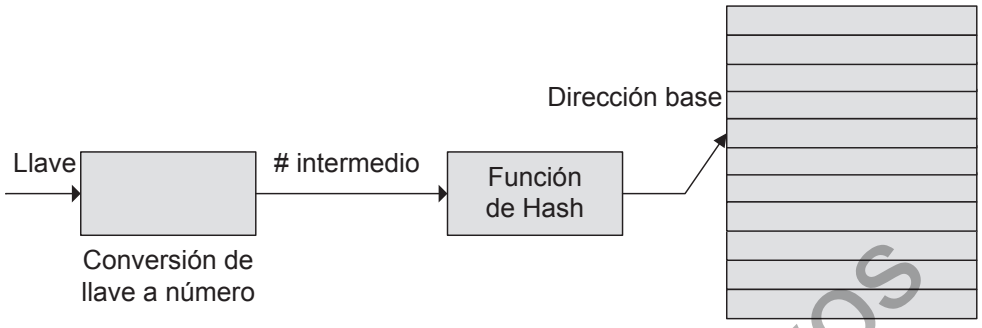
- Técnica para generar una dirección base única para una clave dada. La **dispersión** se usa cuando se requiere acceso rápido mediante una clave.
- Técnica que convierte la clave asociada a un registro de datos en un número aleatorio, el cual posteriormente es utilizado para determinar dónde se almacena dicho registro.
- Técnica de almacenamiento y recuperación que usa una función para mapear registros en direcciones de almacenamiento en memoria secundaria.

La primera definición plantea la dispersión como una técnica que permite generar la dirección donde se almacena un registro a partir de la llave o clave de dicho registro. La segunda definición agrega la conversión de la clave en un número aleatorio que será la base para determinar dónde el registro se almacenará. Ambas definiciones están en concordancia.

Por último, la tercera definición menciona que la técnica utiliza una función, que será responsable de obtener la dirección física de almacenamiento del registro.

Las tres definiciones anteriores pueden resumirse en la Figura 8.1. Aquí puede observarse cómo la llave primero se convierte en un número, sobre el cual puede aplicarse la función de *hash*, que permite obtener la dirección donde el registro se almacenará dentro del archivo.

Figura 8.1



La técnica de dispersión presenta una serie de atributos. Estos pueden resumirse en los siguientes:

- No se requiere almacenamiento adicional. Esto significa que cuando se elige la opción de dispersión como método de organización de archivos, es el archivo de datos el que resulta disperso. Esta dispersión se genera a partir de la clave o llave primaria del archivo, y el registro que contiene la información relacionada con la clave es ubicado en el espacio resultante de aplicar la función de *hash*. Así, no es necesario tener una estructura auxiliar que actúe como soporte para poder acceder rápidamente a la información.
- Facilita la inserción y eliminación rápida de registros en el archivo. Como se analizará más adelante en este capítulo, el proceso de inserción o borrado de información se realiza de una manera más eficiente en términos de accesos a disco. En general, con un solo acceso a disco, un registro puede ser insertado o eliminado del archivo. La ventaja de este método reside básicamente en este punto.
- Localiza registros dentro del archivo con un solo acceso a disco. Como corolario del punto anterior, otra ventaja del método de dispersión consiste en ubicar cada elemento de datos, en promedio, con un acceso a disco. Si bien no es posible asegurar que cualquier registro sea encontrado en un solo acceso, la gran mayoría de las búsquedas serán efectivamente satisfechas con un acceso a disco. Resta estudiar, en este capítulo, bajo qué condiciones no se respeta esta característica.

Sin embargo, el método de dispersión presenta limitaciones. Estas indican situaciones donde el método no es aplicable, o donde, a partir de su aplicación, no es posible lograr otras características que podrían ser deseadas para el archivo de datos. Estas limitaciones son las siguientes:

- No es posible aplicar la técnica de dispersión en archivos con registros de longitud variable. Esto se debe a que cada dirección física obtenida debe tener capacidad para almacenar un registro de tamaño conocido. Es decir, cuando se obtiene una dirección de almacenamiento, asociada a esta hay un espacio de almacenamiento conocido y acotado. No puede ocurrir que el registro a almacenar no quepa en dicho espacio.
- No es posible obtener un orden lógico de los datos. Utilizando índices como metodología de acceso a datos, no solo la búsqueda es eficiente, sino que además presenta la característica de mantener los registros ordenados. Bajo la técnica de *hashing* o dispersión no es posible preservar la propiedad de orden. Los registros son esparcidos en el archivo de datos, de acuerdo con la dirección que se obtiene con el uso de la función de *hash*.
- No es posible tratar con claves duplicadas. Así, no es aplicable la función de *hash* sobre una clave secundaria. Si se analiza con detalle dicha situación, se podrá observar a qué se debe este impedimento. Una clave secundaria puede repetirse; por lo tanto, dos registros diferentes con la misma clave secundaria, aplicando la función de *hash*, tendrían como resultado la misma dirección de memoria. Aquí ocurriría una contradicción con el principio básico del método de dispersión. Cada registro debe, *a priori*, residir en direcciones diferentes del archivo. Si se aceptara trabajar con claves secundarias, esta propiedad no se cumpliría.

Tipos de dispersión

El método de *hashing* presenta dos alternativas para su implantación: tratamiento de espacio en forma estática o tratamiento de espacio en forma dinámica.

Se denomina ***hashing* con espacio de direccionamiento estático** a aquella política donde el espacio disponible para dispersar los registros de un archivo de datos está fijado previamente. Así, la función de *hash* aplicada a una clave da como resultado una dirección física posible dentro del espacio disponible para el archivo.

Se denomina ***hashing* con espacio de direccionamiento dinámico** a aquella política donde el espacio disponible para dispersar los registros de un archivo de datos aumenta o disminuye en función de las necesidades de espacio que en cada momento tiene el archivo. Así, la función de *hash* aplicada a una clave da

como resultado un valor intermedio, que será utilizado para obtener una dirección física posible para el archivo. Estas direcciones físicas no están establecidas *a priori* y son generadas de manera dinámica.

En el resto del capítulo, se presentarán ambas políticas con detalle. En primer término, será abordado el *hashing* con espacio de direccionamiento estático. De esta forma se podrán analizar el funcionamiento del método, sus principales características y el tratamiento de condiciones especiales. Luego, se analizarán los casos problemáticos y cómo estos pueden ser solucionados con espacio de direccionamiento variable.

Parámetros de la dispersión

El método de dispersión, cuando utiliza espacio de direccionamiento estático, presenta cuatro parámetros esenciales que definen su comportamiento. En este apartado, se abordan estos parámetros y se analiza con detalle la influencia que ejercen sobre el método.

Los cuatro parámetros a estudiar son los siguientes:

- Función de *hash*.
- Tamaño de cada nodo de almacenamiento.
- Densidad de empaquetamiento.
- Métodos de tratamiento de desbordes (*overflow*).

Función de *hash*

El primer parámetro a analizar es la función de *hash*. Esta función puede verse como una caja negra que recibe como entrada una clave, y produce una dirección de memoria donde almacenar el registro asociado a la clave en el archivo de datos.

Una **función de *hash* o dispersión** es una función que transforma un valor, que representa una llave primaria de un registro, en otro valor dentro de un determinado rango, que se utiliza como dirección física de acceso para insertar un registro en un archivo de datos.

La definición anterior determina que la función de *hash* debe retornar como resultado una dirección que se encuentre en un determinado

rango. Sin embargo, una vez aplicada dicha función sobre una clave, el valor resultante puede ser cualquiera.

Se debe tener presente que el conjunto de direcciones físicas sobre la estructura de almacenamiento secundario (disco rígido) es un elemento finito, es decir, el método cuenta *a priori* con un conjunto fijo y finito de direcciones disponibles.

Como la función de *hash* podría retornar cualquier valor, es necesario, previo a retornar el resultado final, mapear dicho valor dentro del rango de valores posibles.

Un ejemplo sencillo para una función de *hash* puede ser aquella que sume los valores ASCII de la clave y luego mapee el resultado dentro del rango de direcciones disponibles.

La Figura 8.2 presenta el algoritmo de dicha función de *hash*. Se puede observar en la figura que la función recibe como parámetro de entrada la clave a convertir y su dimensión, y que retorna la dirección física donde el registro será almacenado. Además, en la función de *hash* se conoce la cantidad de direcciones físicas disponibles, lo que permite mapear el resultado a un rango previamente establecido. Se debe asumir que la función `ASCII` retorna el código correspondiente al carácter indicado.

Figura 8.2

```
function  hash_ascii ( llave:  string, tamaño: integer ) :  integer;
var  valor, i : integer;
begin
     valor := ASCII( llave[1] );
     for i = 2 to tamaño
          valor := valor + ASCII( llave[i] );
     hash_ascii := valor / numero_direcciones_fisicas;
end;
```

Pueden observarse de la función anterior su sencillez, su facilidad de implementación y su velocidad de ejecución. Así, la función de *hash* aplicada sobre cualquier llave resolverá rápidamente la dirección física donde almacenar el registro asociado a la clave.

Sin embargo, esta sencillez tiene asociado un inconveniente. Suponga el lector dos claves diferentes: `DACA` y `CADA`. Si se aplica la función anterior, el valor ASCII resultante será el mismo; por lo tanto, los registros asociados a ambas claves se deberían almacenar en el mismo lugar de memoria. Esta situación genera una colisión entre ambos registros. En este caso, las llaves `DACA` y `CADA` se denominan sinónimos.

Las colisiones o desbordes ocasionan problemas. No es posible almacenar dos registros en el mismo espacio físico. Así, es necesario encontrar una solución a este problema. Las alternativas en este caso son dos:

- Elegir un algoritmo de dispersión perfecto, que no genere colisiones. Este tipo de algoritmo debe asegurar que dadas dos claves diferentes siempre se obtendrán dos direcciones físicas diferentes. Se ha demostrado que obtener este tipo de funciones resulta extremadamente difícil. En general, intentar generar estos algoritmos no es una opción válida cuando se decide trabajar con el método de dispersión.
- Minimizar el número de colisiones a una cantidad aceptable, y de esta manera tratar dichas colisiones como una condición excepcional. Aquí se debe tener en cuenta que uno de los parámetros que afectan la eficiencia del *hash*, y que aún no se han discutido, es precisamente el método de tratamiento de colisiones o desbordes. Existen diferentes modos de reducir el número de colisiones; las alternativas disponibles son:
  - º Distribuir los registros de la forma más aleatoria posible. Las colisiones se presentan cuando dos o más claves compiten por la misma dirección física de memoria. Para ello, se debe buscar una función de dispersión que distribuya su resultado de la forma más aleatoria posible.
  - º Utilizar más espacio de disco. Si se intentan distribuir 10 registros en 10 lugares, significa que hay un lugar disponible para cada registro, con lo cual las posibilidades de colisión son altas. Ahora bien, si se dispone de 100 lugares para cada uno de los 10 registros, se tendría un archivo con 10 direcciones posibles para cada registro. Esto, sin duda, debería disminuir el número de colisiones que se producen. La desventaja asociada con utilizar más espacio del necesario es, básicamente, el desperdicio de espacio. No hay una única respuesta para la pregunta acerca de cuánto espacio "vacío" debe dejarse o tolerarse en un archivo para minimizar las colisiones; a lo largo del capítulo, se analizarán alternativas.
  - º Ubicar o almacenar más de un registro por cada dirección física en el archivo. Esto es, que cada dirección obtenida por la función de *hash* sea la dirección de un nodo o sector del disco donde es posible almacenar más de un registro. Así, dos claves, aun en situación de colisión, podrían ser almacenadas en la dirección física asignada por la función de *hash*, pues el nodo tiene mayor capacidad. Esto significa que las claves sinónimos para una función de *hash* determinada podrán albergarse en la misma

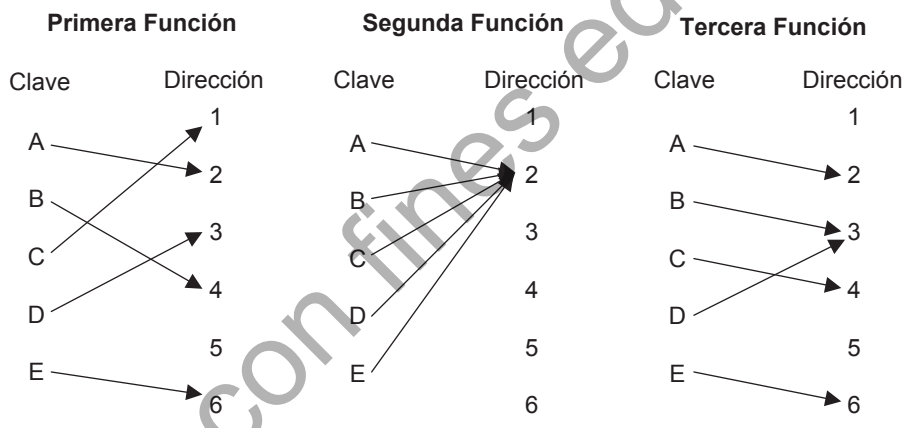
dirección física. No obstante, la capacidad de un nodo es limitada, es decir, puede albergar una cantidad (previamente fijada) de registros asociados a claves sinónimos. Si, a pesar de la mayor de capacidad del nodo, un registro ya no cabe en esa dirección, se dice que el nodo está saturado o en *overflow* (tema que será analizado posteriormente).

Uno de los objetivos fundamentales cuando se utiliza el método de dispersión o *hashing* es la selección de la función de *hash*. Esta función debe esparcir los registros de la manera más uniforme posible, es decir, que a cada clave se le asigne una dirección física distinta.

El algoritmo definido en la Figura 8.2 dista en mucho de ser un algoritmo uniforme, como ya se analizó.

La Figura 8.3 presenta el resultado de tres funciones de *hash* diferentes esparciendo un conjunto de registros.

Figura 8.3



Se puede observar que la primera función esparce los registros de manera uniforme; hay disponibles seis direcciones y los cinco registros distribuidos ocupan lugares diferentes. La segunda función es la peor posible; los cinco registros se ubican en el mismo lugar del disco. En tanto, la tercera función, si bien no es perfecta, puede considerarse una función aceptable. Allí, los registros *A* y *D* se ubican en el mismo sector, en tanto que el resto se asigna a diferentes lugares.

Aunque la distribución aleatoria de registros entre direcciones no representa la situación ideal, es una alternativa razonable, dado que es muy poco factible encontrar una función que logre una distribución realmente uniforme.

En bibliografía especializada del tema es posible analizar distintas alternativas de funciones de *hash*.

Tamaño de cada nodo de almacenamiento

Dentro de los parámetros que afectan la eficiencia del método de dispersión está presente el tamaño o capacidad de cada nodo.

En el apartado anterior, cuando se discutió la función de *hash*, fue presentada como alternativa a las colisiones la posibilidad de que un nodo tenga capacidad de albergar más de un registro.

Para determinar el tamaño ideal de un nodo, se realiza el mismo análisis que el efectuado en el Capítulo 7 para árboles balanceados. Es así que la capacidad del nodo queda determinada por la posibilidad de transferencia de información en cada operación de entrada/salida desde RAM hacia disco, y viceversa.

Posteriormente, se analizarán situaciones de poseer nodos con mayor capacidad y sus beneficios. Estos beneficios tienen que ver con la cantidad de colisiones producidas.

Densidad de empaquetamiento

El primer parámetro analizado, la función de *hash*, mostró las ventajas que se obtienen a partir de una elección que minimice las colisiones. También se analizó que con el uso de memoria adicional es posible reducirlas.

Se analizará un nuevo concepto, la densidad de empaquetamiento, como alternativa de evaluación.

> Se define la **Densidad de Empaquetamiento (DE)** como la relación entre el espacio disponible para el archivo de datos y la cantidad de registros que integran dicho archivo.

La siguiente fórmula describe la DE cómo la razón entre la cantidad de registros que componen un archivo (*r*) y el espacio disponible para almacenar ese archivo.

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)**.

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

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 posibilidad de colisiones, dado que en ese caso se dispone de menos espacio para esparcir registros. Por el contrario, si la DE se mantiene baja, se dispone de mayor espacio para esparcir registros y, por ende, disminuye la probabilidad de colisiones.

Por otra parte, cuando la DE se mantiene baja, se desperdicia espacio en el disco, dado que se utiliza menor espacio que el reservado, generando fragmentación.

La DE no es constante. Al inicio, la DE será baja. Suponga que se crea un archivo, al cual inicialmente se le agrega un registro, y que la función de *hash* puede direccionar 100 lugares de disco y cada uno tiene capacidad para 10 registros. Esto significa que la DE inicial es (1 / 100 * 10), lo que puede considerarse como un número muy bajo, con un alto desperdicio de espacio en disco. Sin embargo, a medida que el archivo es utilizado, se van incorporando nuevos registros. Luego de un tiempo de uso, el archivo puede contener 500 registros, y la DE aumenta a 50%. El archivo puede continuar "creciendo" hasta tener 990 registros almacenados, con una DE de 99%. Note el lector que, en este caso, se debería considerar aumentar el espacio disponible, dado que el archivo está próximo a su límite de espacio.

La acción de aumentar el espacio de direcciones (por ejemplo, de 100 a 200 direcciones) implica reubicar a todos los registros ya almacenados. Si la función de *hash* en uso esparce entre 100 direcciones, al aumentar el número base de direcciones, la función de *hash* debe cambiar para adaptar sus resultados al nuevo espacio disponible y, por ende, todos los registros esparcidos deben ser reubicados.

Métodos de tratamiento de desbordes (*overflow*)

Un desborde u *overflow* ocurre cuando un registro es direccionado a un nodo que no dispone de capacidad para almacenarlo. Cuando esto ocurre, deben realizarse dos acciones: encontrar lugar para el registro en otra dirección y asegurarse de que el registro posteriormente sea encontrado en esa nueva dirección. Para efectivizar estas dos acciones, existen diferentes métodos. Más adelante, en este mismo capítulo, estos métodos serán tratados junto con la *performance* de cada uno de ellos.

En el apartado siguiente, se realizará un estudio con respecto a la probabilidad de ocurrencia de *overflow*.

Estudio de la ocurrencia de *overflow*

Como ya se planteó en este capítulo, una de las características principales del método de dispersión es asegurar que los registros sean rápidamente localizados en el archivo, a tal punto que se plantea como acceso directo a la información, lo que implica encontrar los registros en un solo acceso a disco. Si bien esto es posible, no siempre se puede cumplir.

En este apartado se analizará la *performance* del método de dispersión, y se estudiará bajo qué circunstancias se puede garantizar acceso directo y cuándo esto no es posible.

Dado que generalmente no se puede lograr una distribución uniforme de los registros en el espacio de direcciones disponible, se debe estudiar la probabilidad de distribución. Esto permite analizar cuáles direcciones quedarán saturadas o completas y cuáles no.

Al intentar ubicar un registro, la función de *hash* retorna la dirección física donde el registro debe residir. Esta dirección física es la dirección de un nodo dentro del espacio de direccionamiento definido. Una vez dispersados todos los registros de un archivo, las preguntas que sería interesante responder son las siguientes:

- ¿Cuántos nodos no recibieron ningún registro?
- ¿Cuántos nodos recibieron solo un registro?
- En general, ¿cuántos nodos recibieron *i* registros?

Para poder realizar este análisis, será necesario considerar algunas nociones mínimas de probabilidades.

En el problema que se analizará a continuación, se definen una serie de variables que lo identifican:

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.

C determina la capacidad de cada nodo.

Luego, es necesario poder determinar la probabilidad de que un nodo reciba *i* registros.

Se demostrará que:

$$P(i) = \frac{K! * (1/N)^i * (1 - 1/N)^{K-i}}{i! * (K-i)!}$$

Se denominará $P(A)$ a la probabilidad de direccionar un nodo específico de los N disponibles, y $P(B)$, a la probabilidad de no direccionar ese nodo específico, es decir, direccionar cualquiera de los N-1 restantes.

Si se arroja una moneda al aire, la probabilidad de obtener cara o ceca es igual, 1/2 en cada caso.

Si se arroja un dado, la probabilidad de obtener un número específico es 1/6, debido a que hay seis resultados posibles.

Generalizando el caso anterior, $P(A)$ indica la probabilidad de que una clave sea direccionada a un nodo específico dentro de los N disponibles, entonces:

$P(A) = 1/N$

Como $P(B)$ complementa a $P(A)$,

$P(A) + P(B) = 1$

Entonces, $P(B) = 1 - P(A)$; la probabilidad de no ir a un nodo específico es direccionar cualquiera de los restantes.

Reemplazando $P(A)$:

$P(B) = 1 - P(A) = 1 - 1/N = (N - 1) / N$

¿Cuál es la probabilidad de que dos claves cualesquiera ocupen un nodo específico?

Siguiendo el razonamiento anterior, se intenta analizar la posibilidad de $P(AA)$, es decir, que dos claves se ubiquen en el mismo nodo.

Cuando los eventos controlados por las probabilidades resultan independientes, ocurre que:

$P(AA) = P(A) * P(A) = (1/N) * (1/N)$

En este caso, la dirección obtenida para una clave no condiciona a la obtenida para otra clave. Es decir, dos claves diferentes no se condicionan entre sí, y por lo tanto, se pueden considerar independientes.

Continuando con el mismo razonamiento, la probabilidad de que dos claves no ocupen un nodo específico será:

$P(BB) = P(B) * P(B) = (1 - 1/N) * (1 - 1/N)$

Por último, la probabilidad de que una clave se direccione a un nodo específico, y otra clave, a un nodo diferente, será:

$P(AB) = P(A) * P(B) = (1/N) * (1 - 1/N)$

Con el mismo criterio, con tres claves, se puede observar que:

$P(AAA) = P(A) * P(A) * P(A) = (1/N) * (1/N) * (1/N) = (1/N)^3$

A partir de tres claves se pueden analizar varias alternativas: *P*(*ABB*), *P*(*BBB*), *P*(*BBA*), etcétera.

Generalizando lo anterior, ¿cuál sería la probabilidad de que un nodo reciba *i* registros?

$P(A\ldots A) = P(A) * \ldots * P(A) = (1/N)^i$ (se define *A*…*A* como la ocurrencia de *A* *i* veces)

El problema original consistía en analizar la probabilidad de que *i* llaves, de las *K* disponibles, se direccionaran hacia un nodo específico, es decir, analizar *P*(*A*) *i* veces y *P*(*B*) *K-i* veces.

$P(A\ldots AB\ldots B) = P(A) * \ldots * P(A) * P(B) * \ldots * P(B) = P(A)^i * P(B)^{K-i} = (1/N)^i * (1 - 1/N)^{K-i}$

El resultado anterior corresponde tanto a *P*(*A*…*AB*…*B*) como a *P*(*B*…*BA*…*A*), con la ocurrencia de *A* *i* veces y *B* *K-i* veces.

Queda por analizar aún cuántas combinaciones de *A* y *B* producen el mismo resultado. Se puede observar que de las *K* llaves disponibles interesa la ubicación de *i* claves. La forma de selección de *i* llaves, a partir de las *K* disponibles, responde al número combinatorio *K* tomado de *i*. En términos matemáticos, esto se expresa como:

$K! / (i! * (K-i)!)$

Entonces, *P*(*i*), la probabilidad de que un nodo reciba *i* claves entre las *K* disponibles será:

$$P(i) = \frac{K! * (1/N)^i * (1 - 1/N)^{K-i}}{i! * (K-i)!}$$

Este resultado corresponde a la fórmula inicial planteada.

Poisson demostró (esto es obviado) que la probabilidad anterior se puede acotar a partir de la siguiente fórmula:

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

Ejemplo 1

Corresponde ahora estudiar numéricamente el problema. Suponga que se dispone de 10.000 direcciones donde se deben almacenar 10.000 llaves, y que cada dirección solo puede almacenar un registro. En este caso:

$N = 10.000$, $K = 10.000$, $C = 1$

La DE, de acuerdo con la fórmula antes definida, será:

$DE = K / (N * C) = 10.000 / 10.000 = 1$

El cálculo de la probabilidad de que un nodo no reciba ninguna clave (i = 0), utilizando la función de Poisson, será:

$$P(0) = \frac{(10.000 / 10.000)^0 * e^{-(10.000/10.000)}}{0!} = 0.3679$$

Esto significa que 36,79% de las direcciones disponibles no serán direccionadas por la función de *hash* aplicada a cualquiera de las 10.000 llaves. Es decir que 3.679 (36.79% de 10.000) nodos quedarán sin registros.

Si se calcula la probabilidad de que un nodo reciba un solo registro ($i = 1$):

$$P(1) = \frac{(10.000 / 10.000)^1 * e^{-(10.000/10.000)}}{1!} = 0.3679$$

Nuevamente, 3.679 nodos contendrán solo un registro.

$P(2) = 0.1839$, lo que significa que 1.839 nodos recibirán dos registros. Se debe notar que en este caso se tendrán 1.839 registros en saturación u *overflow*. El primer registro cabe en el nodo, y como el nodo solamente puede almacenar un solo registro, el segundo registro produce *overflow*.

$P(3) = 0.0613$ significa 613 direcciones con tres registros. En este caso, se tendrán 1.226 registros en saturación. La función de *hash* intenta asignar tres registros a 613 direcciones; el primero cabe, no así los dos restantes (613 * 2 = 1.226).

Generalizando el problema, hasta el momento se esperan 3.065 (1.839 + 1.226) registros en *overflow*, una cantidad importante. Es decir, hasta aquí, más de 30% de los registros dispersos en el archivo no serán asignados a la dirección base obtenida por la función de *hash*.

Ejemplo 2

Suponga ahora $K = 10.000$, $N = 20.000$, $C = 1$. La DE en este caso se reduce a 50%.

La siguiente tabla muestra la probabilidad de tener nodos sin registros asignados, o con uno, dos, tres o cuatro registros asignados en cada caso.

i	P(*i*)	*N* * *P*(*i*)	Saturación (*N* * *P*(*i*) * (*i*-1))
0	0.6065	12.130	0 registros
1	0.3032	6.065	0 registros
2	0.0758	1.516	1.516 registros
3	0.0126	252	504 registros
4	0.0016	32	96 registros

N * *P*(*i*) muestra el número esperable de nodos que tendrán asignado cero, uno, dos, tres o cuatro registros. En el caso de no tener registros o solamente contener uno, no se produce saturación. Los casos restantes generan *overflow*.

Se puede observar que, en este caso, la saturación es de 2.116 registros, lo que significa 21% de los registros esparcidos.

El resultado es mejor que el presentado anteriormente, ya que con una DE del 100%, la probabilidad de saturación es del orden de 36%.

Por lo tanto, si se baja la DE a 50%, el *overflow* se reduce a 21%.

De aquí se pueden obtener dos conclusiones. La positiva es que, a menor tasa de empaquetamiento, se reduce la condición de *overflow*. La conclusión negativa es que, con baja tasa de DE, aún existe un alto porcentaje de saturación.

Ejemplo 3

Bajo las mismas condiciones de DE del ejemplo anterior (DE = 50%), suponga que *K* = 10.000, *N* = 10.000, *C* = 2.

Se debe notar que la DE no varía pues el denominador sigue siendo el mismo, pero la relación entre *K* y *N* ahora es uno.

i	*P*(*i*)	*N* * *P*(*i*)	Saturación (*N* * *P*(*i*) * (*i*-1))
0	0.3678	3.678	0 registros
1	0.3678	3.678	0 registros
2	0.1839	1.839	0 registros
3	0.0613	613	613 registros
4	0.0153	153	306 registros

Se debe notar que ahora, cuando un nodo recibe dos registros, no genera saturación (*C* = 2). Por lo tanto, se tendrá un registro en *overflow* con *i* = 3, y dos registros en dicha situación con *i* = 4. En este caso, se tendrán 919 registros en saturación, menos de 10% de la cantidad total.

Se puede concluir que, si se aumenta la capacidad de cada nodo, el porcentaje de *overflow* se reduce.

La siguiente tabla resume el estudio anterior. Se presentan cálculos de saturación con diferentes DE y diferentes capacidades para los nodos (que pueden almacenar 1, 2, 5, 10 o 100 registros).

DE	1	2	5	10	100
10%	4.8	0.6	0.0	0.0	0.0
20%	9.4	2.2	0.1	0.0	0.0
30%	13.6	4.5	0.4	0.0	0.0
40%	17.6	7.3	1.1	0.1	0.0
50%	21.3	10.4	2.5	0.4	0.0
60%	24.8	13.7	4.5	1.3	0.0
70%	28.1	17.0	7.1	2.9	0.0
75%	*29.6*	*18.7*	*8.6*	*4.0*	*0.0*
80%	31.2	20.4	10.3	5.3	0.1
90%	34.1	23.8	13.8	8.9	0.8
100%	36.8	27.1	17.6	12.5	4.0

Es posible observar que, a medida que la capacidad de cada nodo aumenta, la probabilidad de saturación disminuye.

Para el caso de disponer de una DE de 75% con nodos que pueden almacenar hasta 100 registros, la probabilidad de colisiones es de menos de 0,09%.

Resolución de colisiones con *overflow*

Aunque la función de *hash* sea eficiente y aun con DE relativamente baja, es probable que las colisiones produzcan *overflow* o saturación. Por este motivo, se debe contar con algún método para reubicar a aquellos registros que no pueden ser almacenados en la dirección base obtenida a partir de la función de *hash*.

Se presentan cuatro de estos métodos: saturación progresiva, saturación progresiva encadenada, doble dispersión y área de desbordes por separado.

Saturación progresiva

El método consiste en almacenar el registro en la dirección siguiente más próxima al nodo donde se produce saturación.

Suponga que tiene un caso como la Figura 8.4 a). El problema simplificado trata con nodos de capacidad 2. En el nodo 50, se encuentran los registros correspondientes a `Alvarez y Gonzales`. 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 registro, por lo tanto, es almacenado en dicho lugar, y la situación queda como la indicada en la Figura 8.4 b).

Figura 8.4

	.....
50	Alvarez Gonzales
51	Abarca Zurita
52	Hernandez
53	.....

Figura 8.4 a)

	.....
50	Alvarez Gonzales
51	Abarca Zurita
52	Hernandez Perez
53	.....

Figura 8.4 b)

	.....
50	Alvarez Gonzales
51	Abarca
52	Hernandez Perez
53	.....

Figura 8.4 c)

El proceso de búsqueda de información debe, ahora, sufrir algún cambio. Si se busca a Perez, la dirección base que determina la función de *hash* seguirá siendo la dirección 50. Como el elemento no se encuentra en dicho lugar y el nodo está completo, se debe continuar la búsqueda en los nodos subsiguientes hasta encontrar el elemento, o hasta encontrar un nodo que no esté completo.

En el caso del ejemplo planteado, no se encuentra a Perez en el nodo 50, tampoco en el nodo 51 y sí es localizado en el nodo 52. Se debe notar la simpleza del método, pero además su limitada eficiencia. En el ejemplo planteado fueron necesarios tres accesos hasta encontrar al registro.

Suponga ahora que, con base en el ejemplo de la Figura 8.4 b), se busca al registro cuya clave es Rodríguez. El resultado de la función de *hash* retorna la dirección 50. Se busca al elemento en el nodo 50, luego en el 51 y en el 52 hasta que, cuando se busca en el 53, se encuentra un nodo no completo y sin el elemento Rodriguez. Es posible decidir, en ese momento, que el elemento no está en el archivo.

El método podría requerir chequear todas las direcciones disponibles en un caso extremo, para poder localizar un registro.

Asimismo, esta técnica necesita una condición excepcional adicional. Suponga ahora que en la Figura 8.4 b) se elimina el registro correspondiente a Zurita, y el gráfico queda como lo muestra la Figura 8.4 c). En este caso, si se intenta localizar a Perez, se presenta un inconveniente. Al chequear la dirección 51, esta no se encuentra completa, por lo que la búsqueda se detendría, sin permitir que Perez sea localizado.

El método necesita indicar que si una dirección estuvo completa anteriormente, debe ser marcada como dirección ya saturada a fin de no impedir la búsqueda potencial de registros. En este caso, el nodo 51, si bien no está saturado, al haberlo estado mantiene esa condición. Se podría utilizar # para indicar dicha situación en el nodo 51 (de ese modo, el nodo quedaría con Abarca #).

Saturación progresiva encadenada

La saturación progresiva encadenada presenta otra variante al tratamiento de la saturación. En líneas generales, el método funciona igual a su predecesor. Un elemento que se intenta ubicar en una dirección completa es direccionado a la inmediata siguiente con espacio disponible. La diferencia radica en que, una vez localizada la nueva dirección, esta se encadena o enlaza con la dirección base inicial, generando una cadena de búsqueda de elementos.

La Figura 8.5 a) presenta dicha situación. Se puede observar que ahora cada nodo tiene además la dirección siguiente. Inicialmente, las direcciones siguientes contienen –1, indicando que no direccionan a ningún nodo con saturación. Cuando se inserta Perez [Figura 8.5 b)], el registro se almacena en la dirección 52 y, por ende, se genera el enlace respectivo.

Figura 8.5

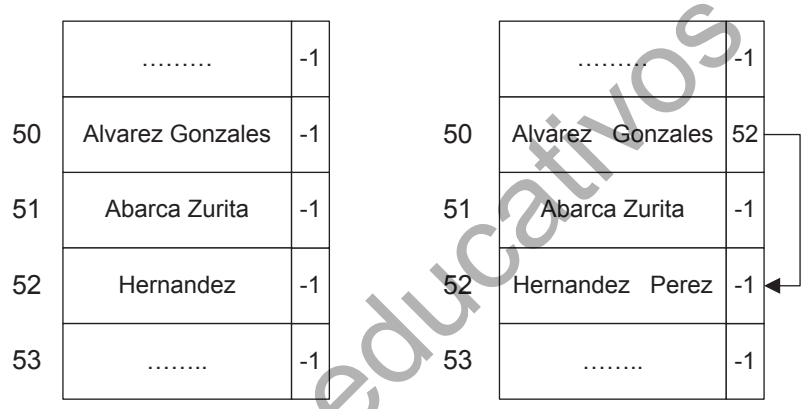


Figura 8.5 a) **Figura 8.5 b)**

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. Si bien la *performance* final de cada método dependerá del orden de llegada de las llaves, en líneas generales puede establecerse que el método de saturación progresiva encadenada presenta mejoras de *performance* respecto de su predecesor. Sin embargo, requiere que cada nodo manipule información extra: la dirección del nodo siguiente.

Cabe considerar que el enlace entre nodos sirve tanto para la búsqueda como para la inserción de nuevas claves. Si, por ejemplo, se desea insertar una clave a la cual la función de *hash* le asigna el nodo 50, al estar este nodo lleno se intenta insertar en el siguiente según el enlace (nodo 52). Nuevamente, al estar completo el nodo 52, se busca en el próximo nodo indicado en el enlace, que como en este caso tiene valor –1, hace que se deba buscar el nodo siguiente más próximo con espacio libre.

Existen algunas otras variantes y consideraciones respecto de este método de tratamiento de desbordes, que quedan por fuera del contenido del presente material.

Doble dispersión

El problema general que presenta la saturación progresiva es que, a medida que se producen saturaciones, los registros tienden a esparcirse en nodos cercanos. Esto podría provocar un exceso de saturación sectorizada. Existe un método alternativo denominado doble dispersión.

El método consiste en disponer de dos funciones de *hash*. La primera obtiene a partir de la llave la dirección de base, en la cual el registro será ubicado.

De producirse *overflow*, se utilizará la segunda función de *hash*. Esta segunda función no retorna una dirección, sino que su resultado es 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.

La doble dispersión tiende a esparcir los registros en saturación a lo largo del archivo de datos, pero con un efecto lateral importante. Los registros en *overflow* tienden a ubicarse "lejos" de sus direcciones de base, lo cual produce un mayor desplazamiento de la cabeza lectora/grabadora del disco rígido, aumentando la latencia entre pistas y, por consiguiente, el tiempo de respuesta.

Área de desbordes por separado

Ante la ocurrencia de *overflow*, los registros son dispersados en nodos que no se corresponden con su dirección base original. Así, a medida que se completa un archivo por dispersión, pueden existir muchos registros ocupando direcciones que originalmente no les correspondían, disminuyendo la *performance* final del método de *hashing* utilizado.

Para evitar estas situaciones, se sugiere como alternativa el uso del área de desbordes por separado. Aquí se distinguen dos tipos de nodos: aquellos direccionables por la función de *hash* y aquellos de reserva, que solo podrán ser utilizados en caso de saturación pero que no son alcanzables por la función de *hash*.

Figura 8.6

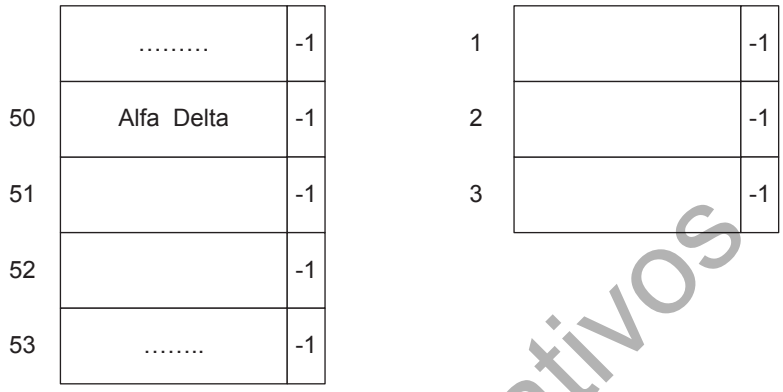


Figura 8.6 a)

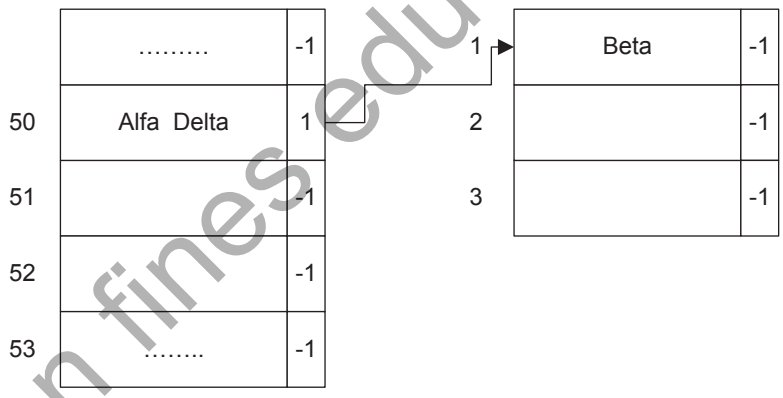


Figura 8.6 b)

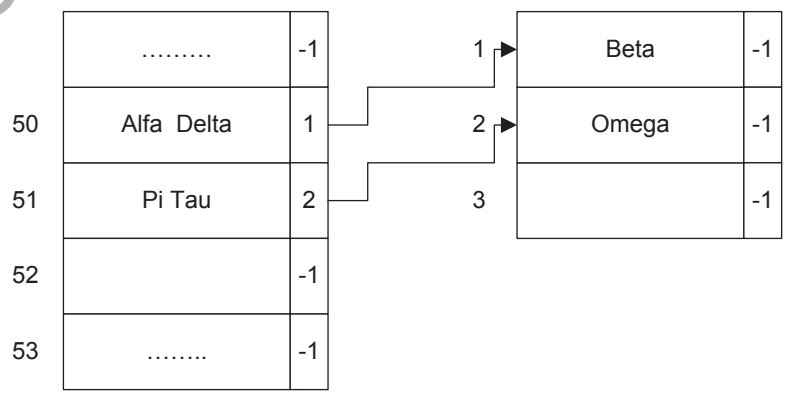


Figura 8.6 c)

continúa >>>

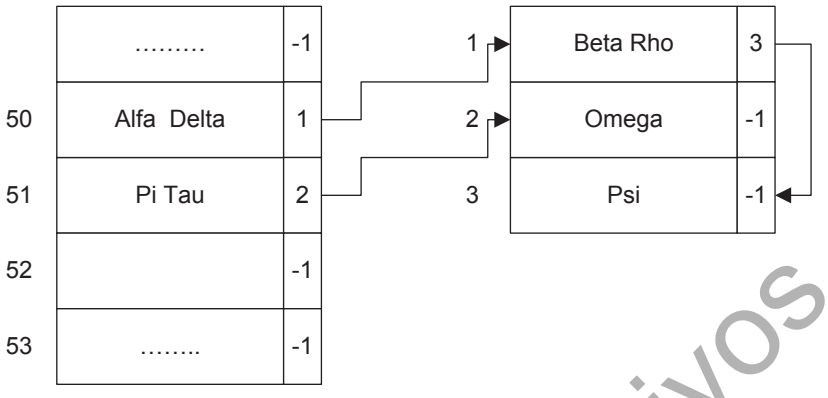


Figura 8.6 d)

La Figura 8.6 a) presenta un ejemplo del problema.

Las llaves Alfa y Delta direccionan la cubeta 50. Suponiendo que esta tiene capacidad para dos registros, se completa. Al dispersar la clave Beta, la dirección obtenida a partir de la función de *hash* es nuevamente la dirección 50. Se produce saturación, y el registro es reubicado en la primera dirección disponible dentro del área de desbordes separada [Figura 8.6 b)]. La dirección base original se encadena con la dirección de reserva.

Luego, como se muestra en la Figura 8.6 c), se dispersan las llaves Pi y Tau, con dirección base 51; si llega Omega al mismo nodo, se produce *overflow* y se utiliza otra dirección del área de desbordes, la 2 en este caso.

Por último, la Figura 8.6 d) muestra lo que ocurriría si llegaran dos claves más que obtuvieran la dirección 50. En este caso, Rho y Psi; la primera de ellas se ubica en la dirección 1 de desbordes; la segunda no cabe: por lo tanto, se redirecciona un nuevo nodo, el tercero del área de desbordes, el cual se enlaza con el primero.

Hash asistido por tabla

El método de *hash* resultó ser el más eficiente en términos de recuperación de información; permite conseguir un acceso para recuperar un dato en más de 99.9% de los casos, cuando la DE es inferior a 75%. Asimismo, las operaciones de altas y bajas se comportan con el mismo nivel de eficiencia para los mismos porcentajes.

Sin embargo, restan aún situaciones en las que puede ser necesario utilizar más de un acceso para recuperar o almacenar un registro. Si bien se discutieron anteriormente alternativas eficientes, es de notar que, a medida que se generan situaciones de saturación, el número de accesos requeridos aumenta.

Existe una alternativa que sigue utilizando espacio de direccionamiento estático y que, aun así, asegura acceder en un solo acceso a un registro de datos. Esta variante para *hash* estático se denomina *hash* asistido por tabla, y para poder ser implementada requiere que una de las propiedades (definidas al principio de este capítulo) no sea cumplida; necesita una estructura adicional.

El método utiliza tres funciones de *hash*: la primera de ellas retorna la dirección física del nodo donde el registro debería residir (F1H); la segunda retorna un desplazamiento, similar al método de doble dispersión, (F2H); y la tercera retorna una secuencia de bits que no pueden ser todos unos (F3H). El método comienza con una tabla con tantas entradas como direcciones de nodos se tengan disponibles. Cada entrada tendrá todos sus bits en uno.

Ante cada inserción, la primera función de *hash* retorna la dirección del nodo donde se debe almacenar el registro. Si el nodo tiene suficiente espacio, es decir, no se produce saturación, el registro es almacenado en ese lugar.

La Tabla 8.1 presenta el resultado de aplicar las tres funciones de *hash* a todas las claves que se utilizarán en el ejemplo. A fines prácticos, F3H retorna, en este caso, solamente 4 bits. Se debe notar que, en dicha tabla, la F1H retorna muchas direcciones similares, para poder presentar la forma que el método tiene para tratar la saturación.

Tabla 8.1

Clave	F1H(Clave)	F2H(Clave)	F3H(Clave)
Alfa	50	3	0001
Beta	51	4	0011
Gamma	52	7	0101
Delta	50	3	0110
Epsilon	52	3	1000
Rho	51	5	0100
Pi	50	2	0010
Tau	53	2	1110
Psi	50	9	1010
Omega	50	4	0000

La Figura 8.7 a) muestra la llegada de las claves Alfa, Beta, Gamma, Delta, Epsilon y Rho; suponga que cada nodo tiene capacidad para dos elementos.

Figura 8.7

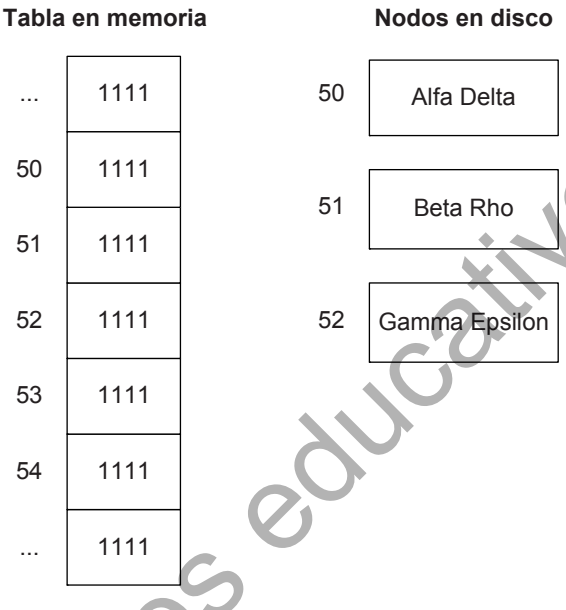


Figura 8.7 a)

Figura 8.7 b)

continúa >>>

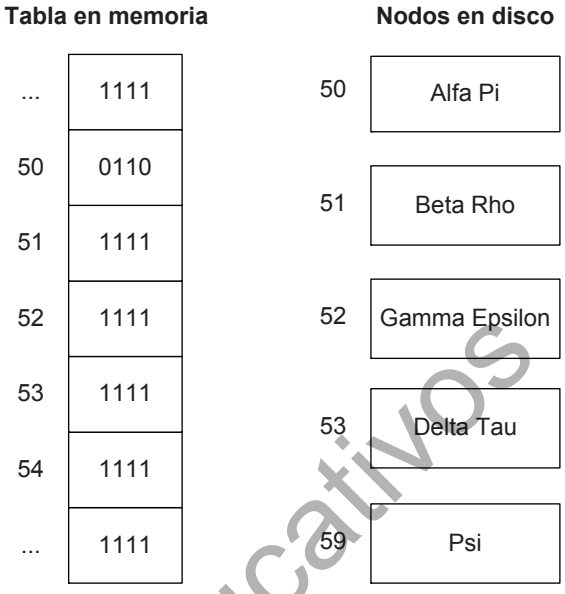


Figura 8.7 c)

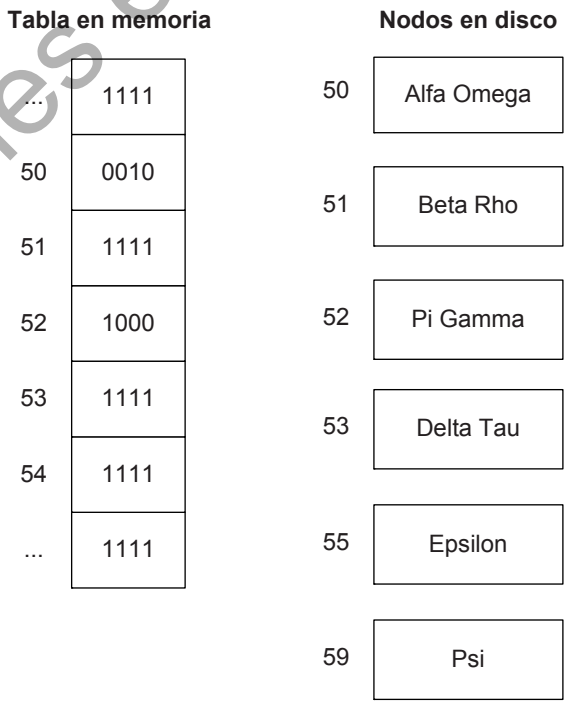


Figura 8.7 d)

Puede observarse que, en cada uno de los casos planteados, no se ha generado saturación. La siguiente clave en la Tabla 8.8 es Pi, que

debería insertarse en la dirección 50. Como esta dirección está completa, el procedimiento es el siguiente:

1. Se obtiene el valor de la F3H para todos los registros en el nodo 50.
2. Se determina la clave que genera el valor mayor. En este caso, Delta tiene el valor mayor.
3. Se escribe el nodo 50 con todos los registros menos el que posee F3H mayor.
4. En la tabla en memoria, para la posición 50 queda el valor de F3H de Delta. La Figura 8.7 b) presenta el estado de dicho nodo.
5. Para la clave seleccionada en el Paso 2, Delta para este ejemplo, se obtiene la F2H.
6. Se suma, a la F1H de Delta, el desplazamiento determinado por F2H.
7. Se intenta insertar Delta en la dirección resultante, en caso de que no haya saturación. Si la hubiera, se procederá nuevamente con los pasos indicados desde 1. Nuevamente, la Figura 8.7 b) presenta el estado final del proceso de inserción con saturación.

Para localizar la llave Delta a partir del archivo generado en la Figura 8.7 b), el proceso debe asegurar un solo acceso a disco, y el procedimiento a realizar es el siguiente:

1. Se genera la F1H para la clave a buscar.
2. Se genera la F3H para la misma clave.
3. Se chequea el contenido de la posición de la tabla indicada por F1H, comparándolo con el resultado de F3H.
   a. Si el resultado de F3H tiene valor menor que el encontrado en la tabla, se busca la clave en cuestión en la dirección del nodo indicada por F1H y el proceso termina.
   b. Si la F3H tiene valor mayor o igual que el encontrado en la tabla, el proceso continúa como se indica en el Paso 4.
4. Se obtiene el valor de F2H y se suma al valor indicado por F1H, obteniéndose una nueva posición.
5. Se comienza nuevamente desde el Paso 3, utilizando ahora como posición el resultado de F1H más el desplazamiento.

De acuerdo con los pasos indicados anteriormente, si se busca Delta, F1H(Delta) = 50, F3H(Delta) = 0110, al comparar este último valor con el almacenado en la dirección 50 de la tabla, el resultado es igual; por lo tanto, se calcula F2H(Delta) = 3 y se suma dicho valor (desplazamiento) a la dirección base previamente obtenida, y el resultado es 53. Se compara el resultado de F3H(Delta) con el contenido de la dirección 53 de la tabla. Como el valor de F3H es menor, la clave Delta debería encontrarse en la dirección 53.

Se accede al nodo 53 y se recupera Delta. Cabe destacar que hubo un solo acceso a memoria secundaria; el resto de las operaciones fueron sobre memoria principal.

Suponga que se desea acceder a la clave Psi, procediendo como en el ejemplo anterior y utilizando los valores de la Tabla 8.1, la dirección de F1H(Psi) = 50, F3H(Psi) = 1010. Como dicho valor (1010) es mayor que la entrada 50 de la tabla, la clave buscada no está en el nodo 50.

Se obtiene la F2H(Psi) = 2, se suma el desplazamiento (50 + 2) y se chequea la dirección 52 de la tabla; en este caso, el resultado de F3H es menor y se accede a la dirección 52 donde Psi no se localiza.

En este caso, se está en condiciones de responder que la clave buscada no se encuentra en el archivo.

Quedan para insertar cuatro claves de la tabla. A continuación, se muestra cómo es el proceso en cada caso. Siguiendo el procedimiento de inserción descripto anteriormente, la clave Tau no produce *overflow*, por lo que se inserta en el nodo 53.

La llave Psi se debería almacenar en el nodo 50, nuevamente produce *overflow*. Se calcula F3H para Alfa, Pi y Psi, y al ser el valor de Psi el mayor, es esta la clave que se quita del nodo 50. En este caso, como el valor de F3H(Psi) es mayor que el valor contenido en la dirección 50 de la tabla, este valor no debe ser cambiado para que la búsqueda pueda realizarse sin problemas.

El desplazamiento (FH2) indicado para Psi es 9, por lo que la dirección utilizada para almacenar Psi es 59. La Figura 8.7 c) muestra cómo queda el archivo luego de insertar Tau y Psi.

Por último, Omega también debe ser almacenada en el nodo 50.

Se produce *overflow* y, entre las tres claves, la que tiene F3H mayor es Pi. En el nodo 50 quedan Alfa y Omega, se actualiza la entrada 50 de la tabla con el valor de F3H(Pi) y se intenta ubicar a Pi en la dirección 52 (50 + el desplazamiento indicado por FH2 para Pi).

Se produce nuevamente *overflow*. Entre las tres claves (Pi, Gamma y Epsilon), la que posee F3H mayor es Epsilon. En el nodo 52 quedarán, entonces, Pi y Gamma, actualizando la entrada 52 de la tabla (con el valor de FH3 para Epsilon).

Epsilon será direccionada al nodo 55 [que se obtiene de sumar F1H(Epsilon) + F2H(Epsilon), como lo muestra la Figura 8.7 d)].

El método asegura encontrar la clave buscada en un solo acceso a disco.

El costo asociado presenta dos aristas. La primera tiene que ver con la necesidad de utilizar espacio extra para la administración de la tabla en memoria principal. La segunda, en tanto, tiene que ver con la complejidad adicional en caso de una inserción que produzca saturación. En este caso, la cantidad de accesos para terminar el proceso está vinculada con la cantidad de *overflows* sucesivos que se produzcan.

La conclusión final de esta variante para *hash* con espacio de direccionamiento estático es que este método pondera la búsqueda sobre las otras operaciones.

El proceso de eliminación

Es necesario, además, hacer mención –aunque de manera somera– del proceso de eliminación que utiliza la variante de *hash* asistido por tabla.

A fines prácticos, se puede establecer el siguiente proceso para dar de baja un registro del archivo:

1. Se localiza el registro de acuerdo con el proceso de búsqueda definido anteriormente.
2. Si el paso anterior encontró la llave buscada, se reescribe el nodo en cuestión sin el elemento a borrar.

Es de notar la simplicidad del proceso de eliminación. Sin embargo, se debe tener en cuenta un caso especial. Suponga que se borra una clave de un nodo que se encontraba completo. Cuando se intente insertar un nuevo elemento en el nodo, habrá lugar.

Si el nodo ya había producido *overflow*, la tabla en memoria contiene un valor correspondiente a la F3H de la llave que produjo saturación. En ese caso, el nuevo elemento a insertar debe cumplir la siguiente propiedad:

F3H(nuevo elemento) < Tabla(dirección del nodo)

es decir, la tercera función de *hash* debe otorgar un valor menor al dato que se encuentra en la tabla en memoria para el nodo en cuestión.

Hash con espacio de direccionamiento dinámico

Hasta el momento, se ha considerado al método de *hash* como una alternativa de almacenamiento que direcciona una clave a un nodo dentro de un conjunto de direcciones disponibles y establecidas de antemano. En todos los ejemplos planteados se determinaron el número de nodos y la capacidad de ellos, sin estudiar ese punto con detalle.

Además, 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.

Se debe notar que falta realizar un análisis mayor con respecto a la cantidad de direcciones disponibles, en el momento en que se empieza a trabajar con un archivo utilizando dispersión.

Suponga que se dispone de 100 direcciones de nodos con capacidad para 200 registros cada una de ellas. Entre todas esas direcciones es posible dispersar hasta 20.000 registros. Para un problema particular, cualquiera podría ser la cantidad de registros y, por ende, se podría utilizar una función de *hash* que dispersara las claves entre los 100 nodos disponibles. De ese modo, si la tasa de crecimiento del archivo fuera de 100 nuevos registros por día, luego de 150 días se llegaría a una DE de 75%. A partir de ese punto, la DE sería lo suficientemente elevada como para generar más saturación de la esperada. Además, luego de 200 días, el archivo alcanzaría el tope máximo posible de 20.000 registros.

La pregunta que debería realizarse es: ¿qué hacer en dicho caso? La respuesta es clara. Cuando el archivo se completa, es necesario obtener mayor cantidad de direcciones para nodos. Y también es necesario que la función de *hash* direccione más nodos de disco.

El problema tiene entonces una solución sencilla, se aumentan las direcciones de 200 a 400. Será necesario, además, modificar la función de *hash*. Ahora debe dispersar entre 400 direcciones disponibles, y eso traerá aparejado que todo el archivo deberá ser redispersado, dado que la función original debe cambiar.

El costo de redispersar un archivo es muy alto. El tiempo utilizado es importante, y mientras se realiza esta operatoria no es posible que el usuario final de la información acceda al archivo. Por este motivo, cuando se piensa en la creación de un archivo, es importante:

1. Analizar la tasa de crecimiento posible del archivo.
2. Determinar la cantidad de nodos que optimice el uso de espacio vs. la periodicidad de la redispersión.

Otra alternativa posible es trabajar con archivos que administren el espacio de direccionamiento de manera dinámica. Esto es, no establecer *a priori* la cantidad de nodos disponibles, sino que esta crezca a medida que se insertan nuevos registros.

Surge así la necesidad de utilizar *hash* con espacio de direccionamiento dinámico. Este *hash* dispersa las claves en función de las direcciones disponibles en cada momento, y la cantidad de direcciones puede crecer, *a priori* sin límites, en función de las necesidades de cada archivo particular.

Existen diferentes alternativas de implementación para *hash* con espacio dinámico: *hash* virtual, *hash* dinámico y *hash* extensible, entre otras. En este libro, será considerada solamente la última alternativa, el *hash* extensible.

Hash extensible

El método de *hash* extensible es una alternativa de implementación para *hash* con espacio de direccionamiento dinámico. 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.

Este método, al igual que el resto de los que trabajan con espacio dinámico, 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.

El principal problema que se tiene con los métodos dinámicos en general y con el *hash* extensible en particular es que las direcciones de nodos no están prefijadas *a priori*, y por lo tanto la función de *hash* no puede retornar una dirección fija. Entonces, es necesario cambiar la política de trabajo de la función de dispersión.

Para el método extensible, 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.

Suponga que la función de *hash* retorna 32 bits. En ese caso es posible direccionar 2^{32} direcciones de nodos diferentes, si fuera necesario. Si se tiene en cuenta que cada dirección podría almacenar 100 registros, por ejemplo, la cantidad de claves a dispersar es importante.

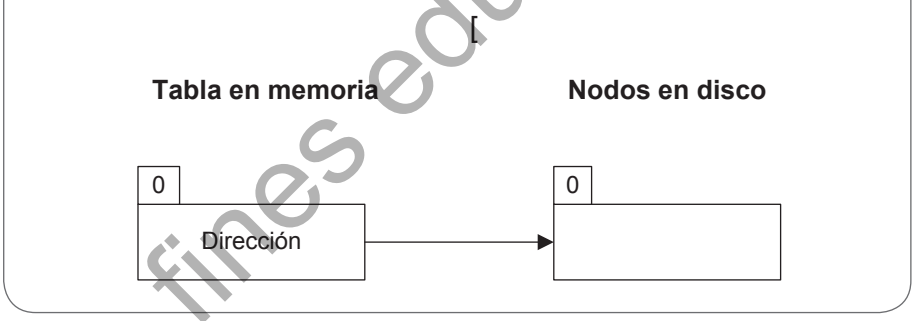
El método necesita, para su implementación, de una estructura auxiliar. Esta estructura es una tabla que se administra en memoria principal. A diferencia del *hash* asistido por tabla, esta estructura contiene la

dirección física de cada nodo. Se debe tener en cuenta que la función de *hash* no retorna una dirección física, sino una secuencia de bits. Dicha secuencia permite obtener de la tabla en memoria la dirección física del nodo para almacenar la llave. Además, la tabla será utilizada posteriormente para recuperar cada registro en un solo acceso a disco.

El tratamiento de dispersión con la política de *hash* extensible comienza con un solo nodo en disco, y una tabla que solamente contiene una dirección, la del único nodo disponible.

La Figura 8.8 presenta el estado inicial del archivo al aplicar el método. El número cero sobre la tabla indica que no es necesario ningún bit de la secuencia obtenida por la función de dispersión, para obtener la dirección física donde almacenar el registro. Considere el lector que hay una sola dirección disponible. Además, inicialmente, el nodo en disco también tiene cero.

Figura 8.8



La Tabla 8.2 muestra una secuencia de llaves a insertar en el archivo generado a partir de la Figura 8.8. Para este problema se establece que la capacidad de cada nodo es de dos registros.

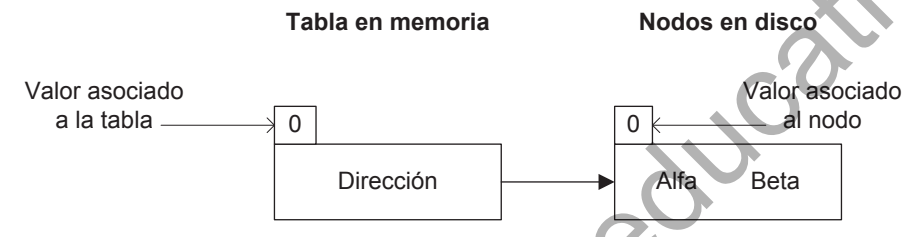
Tabla 8.2

Clave	FH(Clave)
Alfa	00……..1001
Beta	00……..0100
Gamma	00……..0010
Delta	00……..1111
Epsilon	00……..0000
Rho	00……..1011
Pi	00……..0110
Tau	00……..1101
Psi	00……..0001
Omega	00……..0111

Para realizar inserciones, el método trabaja de la siguiente forma. De acuerdo con la Tabla 8.2 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, siempre y cuando no genere saturación; como el nodo está vacío, Alfa se escribe en ese lugar.

Para insertar la clave Beta se procede de la misma forma, dado que nuevamente no se produce *overflow*. La Figura 8.9 muestra cómo quedan el archivo y la tabla en memoria luego de insertar Alfa y Beta.

Figura 8.9



La tercera llave a dispersar es, de acuerdo con la Tabla 8.2, Gamma. Se procede de forma similar a lo descripto anteriormente, pero ahora el único nodo disponible tiene su capacidad colmada. La inserción de Gamma produce *overflow*. La secuencia de acciones, en este punto, es presentada en la Figura 8.10 a); primero se aumenta en uno el valor asociado al nodo saturado. Luego, se genera un nuevo nodo con el mismo valor asociado al nodo saturado.

Cuando se compara el valor del nodo con el valor asociado a la tabla, se puede observar que el primero es mayor que el segundo. Esto significa que la tabla no dispone de entradas suficientes para direccionar al nuevo nodo. De hecho, la tabla tiene una celda única, y como se dispone ahora de dos nodos, hace falta generar más direcciones. En esta situación, la cantidad de celdas de la tabla se duplica, y el valor asociado a la tabla se incrementa en uno. La Figura 8.10 b) muestra cómo queda la tabla.

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*. Esta situación se presenta en la Figura 8.10 c).

Figura 8.10

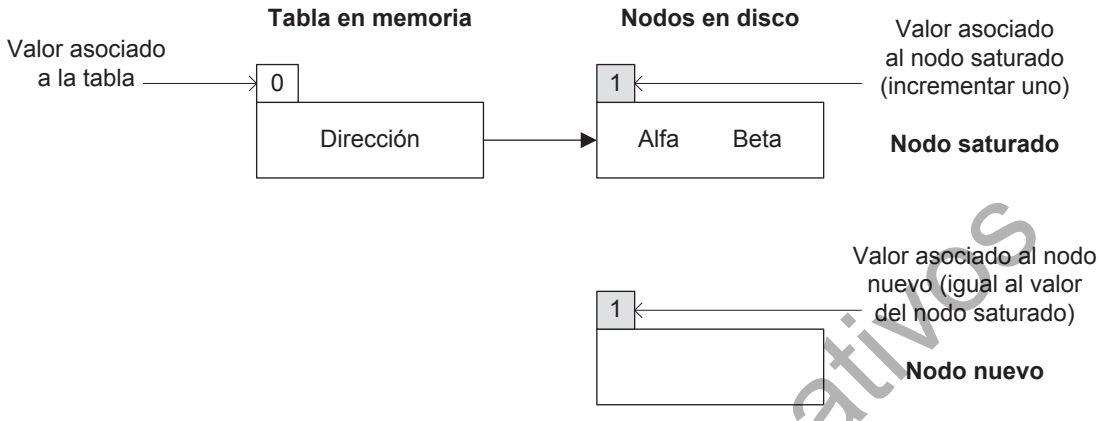


Figura 8.10 a)

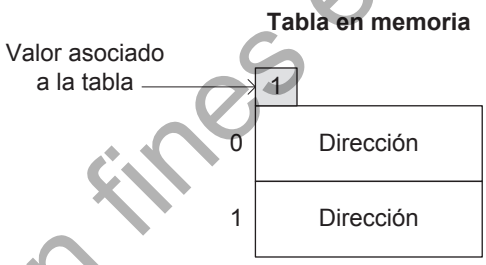


Figura 8.10 b)

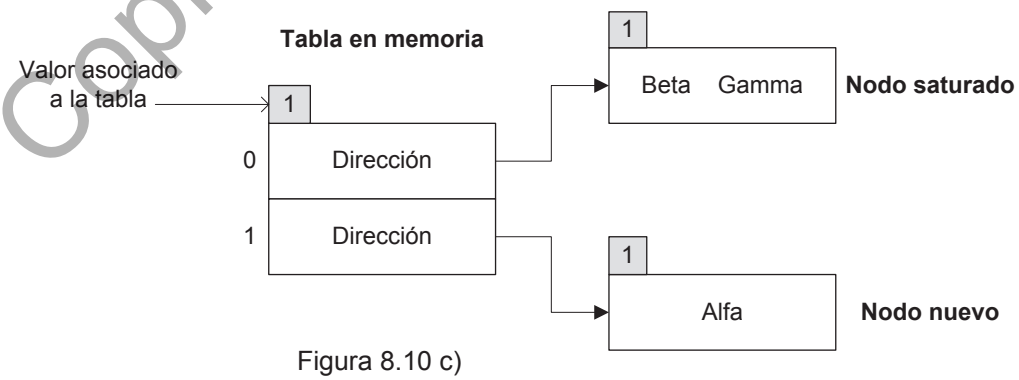


Figura 8.10 c)

continúa >>>

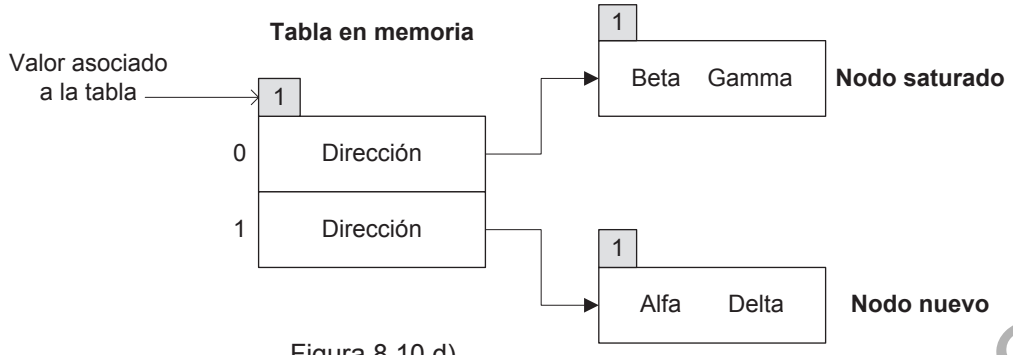


Figura 8.10 d)

La llave Delta tiene un uno en el bit menos significativo. Al direccionar el nodo correspondiente al bit en uno, no se genera *overflow*, por lo que Delta se almacena junto con Alfa [Figura 8.10 d)].

La quinta llave, Epsilon, debe ser almacenada en el nodo asociado a la celda 0 de la tabla. Dicho nodo se encuentra completo, lo que genera un nuevo *overflow*. La Figura 8.11 a) muestra el estado del nodo saturado y el nuevo nodo generado.

Como se explicó en el caso anterior, al no disponer de celdas suficientes en la tabla en memoria principal, se duplica el espacio disponible, que a partir de este momento necesita 2 bits de la función de *hash* para poder direccionar un registro [Figura 8.11 b)].

La celda de referencia 00 contiene la dirección del nodo saturado, en tanto que la celda de referencia 10 contiene la dirección del nuevo nodo.

Los registros Beta, Gamma y Epsilon son reubicados en función de sus 2 bits menos significativos [Figura 8.11 c)]. Se debe notar que el nodo asociado con las celdas de referencia 1, a partir de este momento, es referenciado por dos celdas de la tabla, las correspondientes a 01 y 11.

Figura 8.11

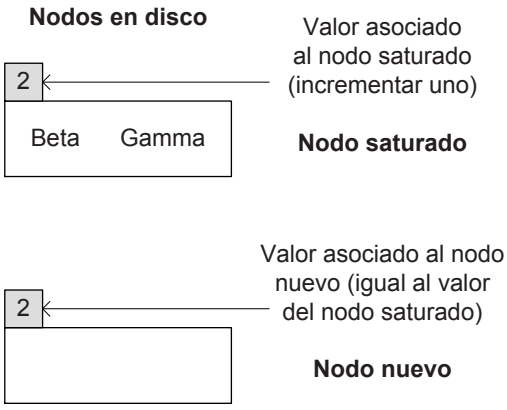


Figura 8.11 a)

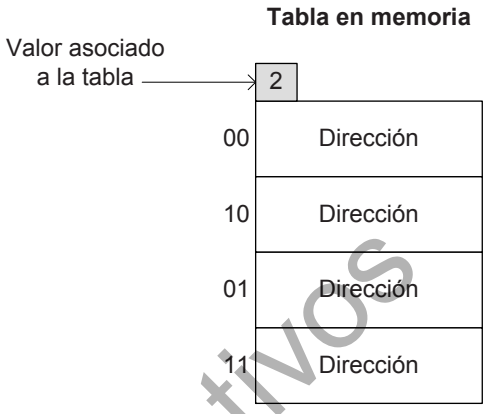


Figura 8.11 b)

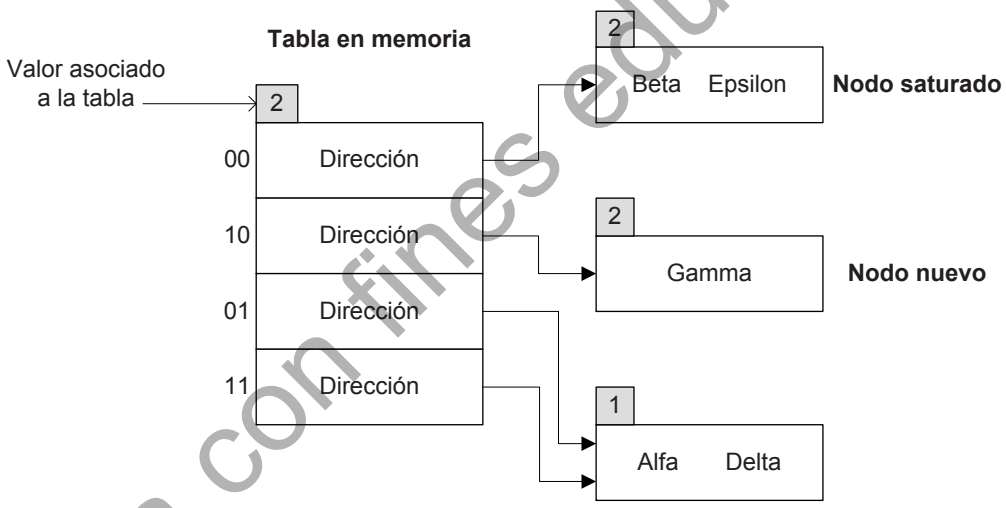
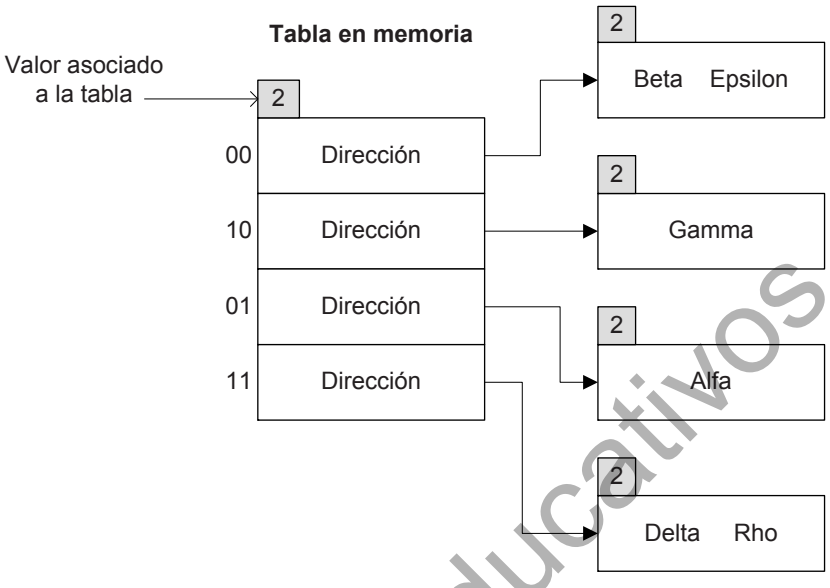


Figura 8.11 c)

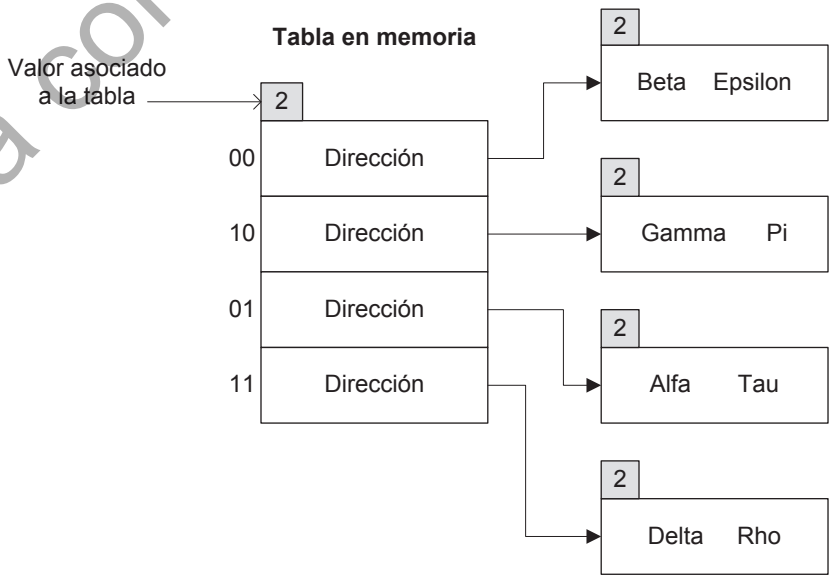
La sexta llave recibida es Rho; su dirección de almacenamiento corresponde al nodo asociado a la celda 11. Dicho nodo está completo, Alfa y Delta lo ocupan. Nuevamente, se genera una situación de saturación y el nodo es dividido. Ahora, el valor asociado a ambos nodos coincide con el valor asociado a la tabla en memoria. Este caso significa que la tabla posee direcciones suficientes para direccionar al nuevo nodo; por lo tanto, la cantidad de celdas no debe ser duplicada. Puede observarse en la Figura 8.11 c) que hay dos direcciones apuntando al mismo nodo. La Figura 8.12 muestra cómo queda el ejemplo luego de insertar Rho.

Figura 8.12



En el ejemplo planteado a partir de las llaves de la Tabla 8.2 quedan, aún, elementos para insertar. La clave siguiente es Pi, la función de *hash* retorna en sus 2 bits menos significativos 10, y el nodo tiene capacidad suficiente para almacenar la clave. Lo mismo ocurre con la llave Tau. En la Figura 8.13 se representa esta situación.

Figura 8.13



La llave Psi se direcciona al nodo correspondiente a la celda 01, la cual produce saturación. La Figura 8.14 muestra el proceso de inserción de la clave en el archivo.

Figura 8.14

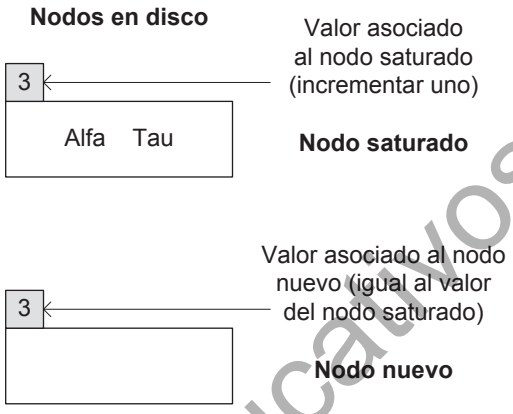


Figura 8.14 a)

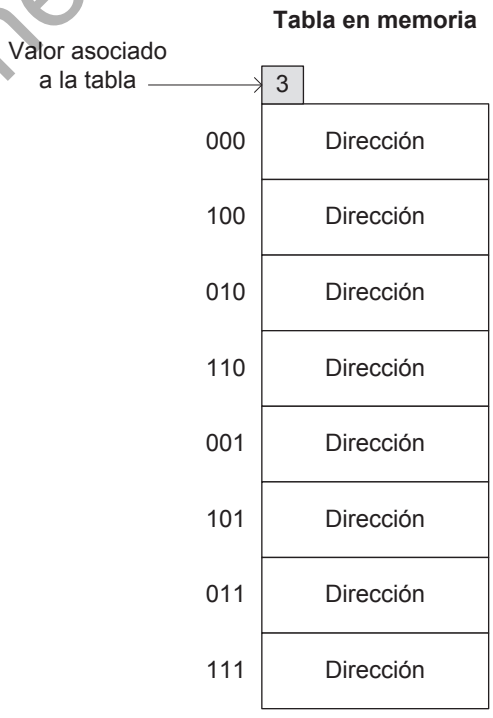


Figura 8.14 b)

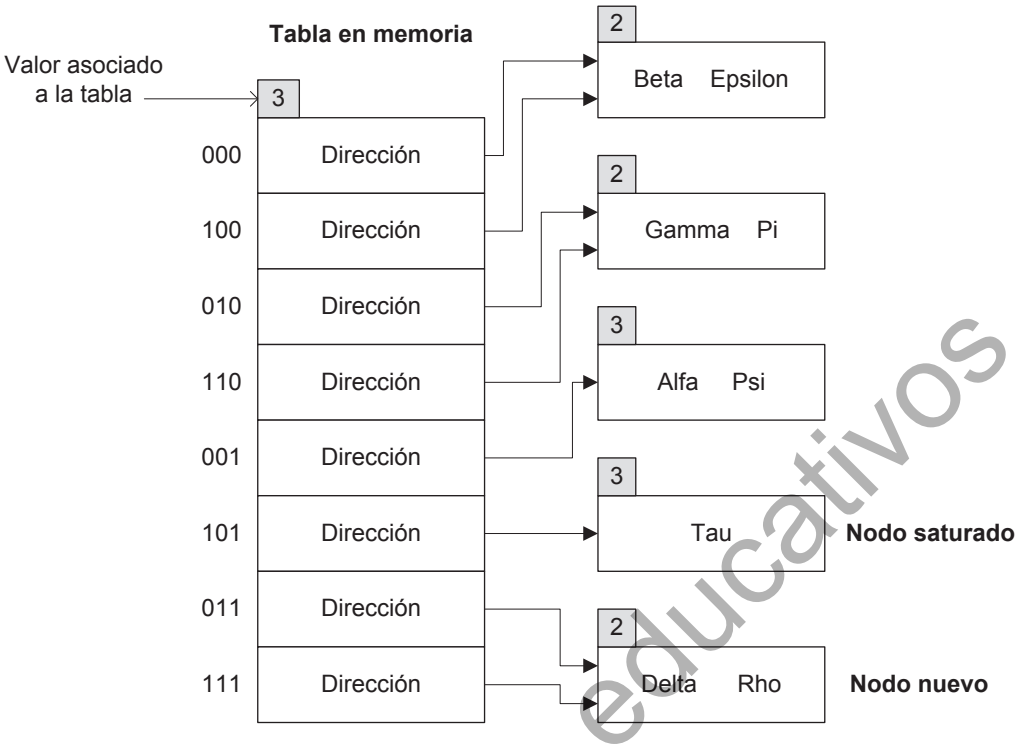
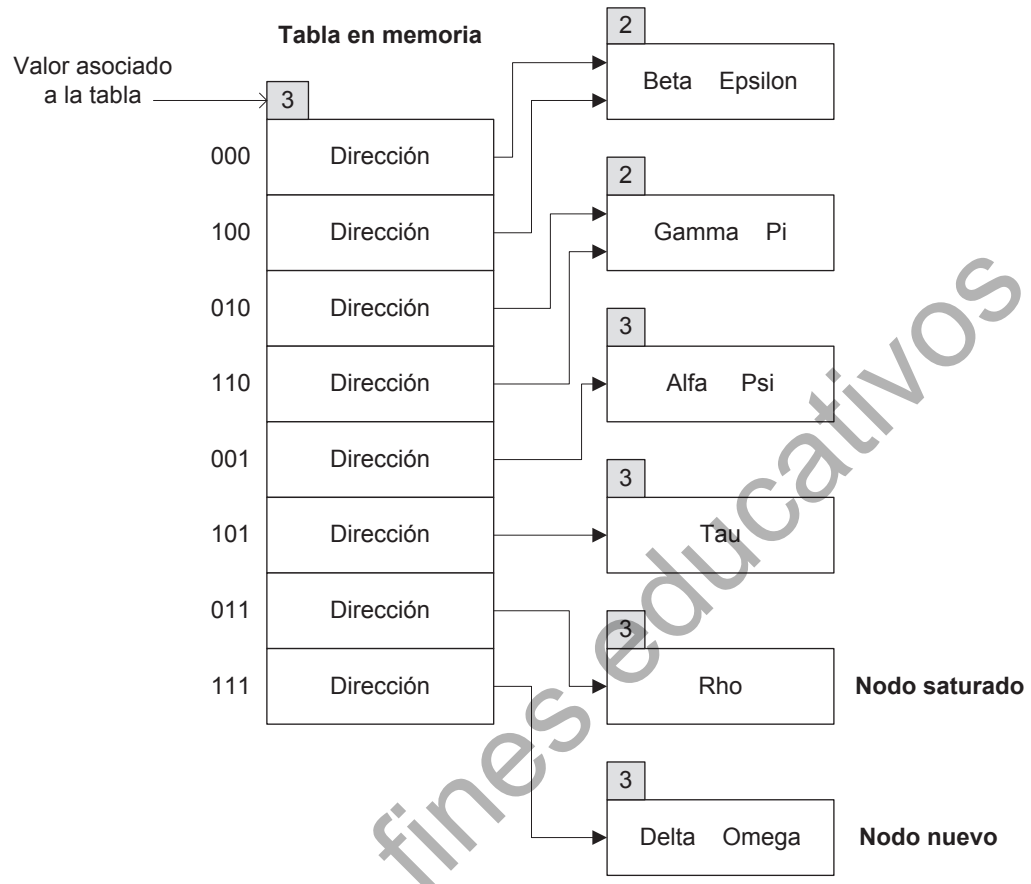


Figura 8.14 c)

La última llave, Omega, se direcciona al nodo correspondiente a la celda 111; nuevamente hay saturación, y la tabla en memoria y los nodos en disco quedan como se presenta en la Figura 8.15.

Figura 8.15



Resumiendo, el método de *hash* extensible trabaja según las siguientes pautas:

- Se utilizan solo los bits necesarios de acuerdo con cada instancia del archivo.
- Los bits tomados forman la dirección del nodo que se utilizará.
- Si se intenta insertar en un nodo lleno, deben reubicarse todos los registros allí contenidos entre el nodo viejo y el nuevo; para ello se toma 1 bit más.
- La tabla tendrá tantas entradas (direcciones de nodos) como 2^i, siendo *i* el número de bits actuales para el sistema.

El proceso de búsqueda asegura encontrar cada registro en un solo acceso. Se calcula la secuencia de bits para la llave, se toman tantos bits de esa llave como indique el valor asociado a la tabla, y la dirección del nodo contenida en la celda respectiva debería contener el registro buscado. En caso de no encontrar el registro en dicho nodo, el elemento no forma parte del archivo de datos.

Conclusiones

Los métodos de búsqueda de información en un archivo presentan ventajas y desventajas en cada caso. Como se discutió a lo largo de toda esta sección de archivos, la búsqueda secuencial de información tiene una *performance* de búsqueda muy deficiente, pero el archivo generado no necesita mayor análisis en cuanto al proceso de altas y/o bajas.

También se indicó oportunamente que el mayor porcentaje de operaciones sobre los archivos son de consultas; por lo tanto, es necesario en una primera etapa maximizar la eficiencia del proceso de búsqueda. Por este motivo, se analizaron alternativas que contemplan la generación de índices y su implantación mediante árboles.

Los árboles balanceados representaron una solución aceptable en términos de eficiencia, reduciendo el número de accesos a disco. Para árboles balanceados se discutieron tres alternativas: *B*, *B** y *B*+. Las dos primeras plantean el mismo comportamiento, pero se diferencian en que *B** completa más los nodos. La variante *B*+ presenta una ventaja adicional, ya que permite no solo la búsqueda eficiente sino también el recorrido secuencial del archivo.

No obstante, para aquellos problemas donde la eficiencia de búsqueda debe ser extrema –y, por ende, se debe realizar un solo acceso para recuperar la información–, se dispone del método de dispersión.

Como se analizó en este capítulo, la dispersión con espacio de direccionamiento estático asegura (en un gran porcentaje de los casos) encontrar los registros buscados con un solo acceso a disco.

Si se desea asegurar siempre un acceso a disco para recuperar la información, la variante de *hash* extensible, que utiliza espacio de direccionamiento dinámico, lo logra. El costo que debe asumirse con esta técnica es mayor procesamiento cuando una inserción genera *overflow*.

La siguiente tabla resume los diferentes métodos de gestión de archivos.

Organización	Acceso a un registro por clave primaria	Acceso a todos los registros por clave primaria
Ninguna	Muy ineficiente	Muy ineficiente
Secuencial	Poco eficiente	Eficiente
Secuencial indizada	Muy eficiente	El más eficiente
Hash	El más eficiente	Muy ineficiente

Cuestionario del capítulo

1. ¿Cuáles son las ventajas de utilizar una política de *hash* para organizar un archivo de datos?
2. ¿Es posible que un archivo se organice mediante *hashing* si se utiliza para dispersar una clave primaria o candidata? ¿Qué sucede si la clave es secundaria?
3. ¿Cuáles son las propiedades básicas que debe cubrir una función de *hash*?
4. ¿Qué parámetros afectan la eficiencia del método de dispersión?
5. Conceptualmente, ¿qué diferencia existe entre colisión y saturación?
6. ¿Qué mide la DE de un archivo? ¿Cómo se calcula?
7. ¿Por qué son necesarios los métodos de tratamiento de saturación? ¿En qué casos son aplicados?
8. ¿Qué aporta el *hash* asistido por tabla?
9. ¿Cuál es la necesidad de disponer políticas de *hash* con espacio de direccionamiento dinámico?
10. ¿Cuál propiedad establecida para los métodos de *hash* no es cumplida por las políticas de *hash* con espacio de direccionamiento dinámico?

Ejercitación

1. Según un resultado matemático sorprendente llamado la paradoja del cumpleaños, si hay más de 23 personas en una habitación, entonces existe más de 50% de posibilidades de que dos de ellas tengan la misma fecha de cumpleaños. ¿En qué forma la paradoja del cumpleaños ilustra un importante problema asociado con la dispersión?
2. Suponga que se tiene un archivo donde el 85% de los accesos son generados por 20% de los registros, y se genera un archivo con una DE de 0.8 y con nodos con capacidad para cinco registros. Cuando se genera el archivo, se dispersa primero 20% de los registros más utilizados. Luego de dispersar a estos registros y antes de ubicar a los demás, ¿cuál es la DE del archivo parcialmente lleno? Con esta DE, calcule el porcentaje de 20% activo que sería registro en saturación. Discuta los resultados.
3. Suponga que debe organizarse un archivo. La siguiente tabla presenta las claves a dispersar y el resultado de aplicar una

determinada función de *hash*, sabiendo que cada nodo tiene capacidad para dos registros. Muestre cómo se genera el archivo suponiendo que la saturación se trata con:

a. Política de saturación progresiva.

b. Política de saturación progresiva encadenada.

Indique en cada caso cuántos accesos son necesarios para recuperar la información.

Clave	Función de *hash*
Roberto	20
María	21
Alberto	22
Sandra	20
Victoria	22
Ignacio	20
Juan	21

4. Para las claves siguientes, realice el proceso de dispersión mediante el método de *hashing* extensible, sabiendo que cada nodo tiene capacidad para dos registros. El número natural indica el orden de llegada de las claves.

1	Nano	0000 1010	2	Mega	0101 0001
3	Micro	1010 1100	4	Giga	0001 1110
5	Mili	0110 0011	6	Tera	1100 0110
7	Kilo	1110 0101	8	Peta	0011 1001

5. Suponga que se está dispersando un archivo de *N* registros, y que se utiliza el método de área de desbordes por separado para la administración de colisiones. Además, se sabe que cada nodo tiene capacidad para dos registros. A continuación, se detalla el conjunto de elementos que llegan al archivo y se indica, además, la dirección del nodo donde debe residir.

   Muestre cómo se completa el archivo, sabiendo que el área de desbordes por separado comienza en el nodo con dirección 105.

Orden	Clave	Cubeta	Orden	Clave	Cubeta
1	Alfa	20	7	Omega	20
2	Beta	21	8	Pi	21
3	Gamma	21	9	Rho	20
4	Delta	22	10	Tita	20
5	Epsilon	20	11	Psi	20
6	Fi	21	12	Tau	21