Capítulo 9

MÉTODOS DE BÚSQUEDA

9.1 INTRODUCCIÓN

Este capítulo se dedica al estudio de una de las operaciones más importantes en el procesamiento de información: la **búsqueda**. Esta operación se utiliza básicamente para recuperar datos que se habían almacenado con anticipación. El resultado puede ser de éxito si se encuentra la información deseada, o de fracaso, en caso contrario.

La búsqueda ocupa una parte importante de nuestra vida. Prácticamente todo el tiempo estamos *buscando* algo. El mundo en que se vive hoy día es desarrollado, automatizado, y la información representa un elemento de vital importancia. Es fundamental estar informados y, por lo tanto, buscar y recuperar información. Por ejemplo, se buscan números telefónicos en un directorio, ofertas laborales en un periódico, libros en una biblioteca, etcétera.

FIGURA 9.1
Ejemplo práctico de búsqueda.

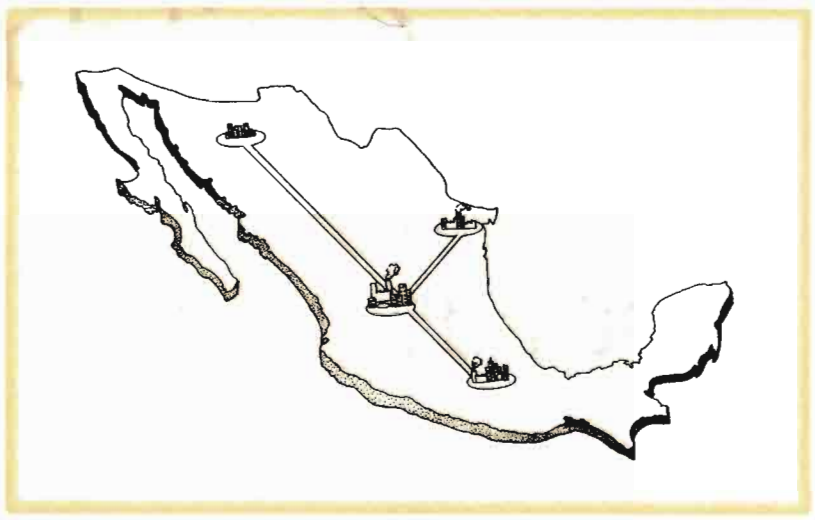


FIGURA 9.2
Ejemplo práctico de búsqueda.

En los ejemplos mencionados, la búsqueda se realiza, generalmente, sobre elementos que están ordenados. Los directorios telefónicos están organizados alfabéticamente. las ofertas laborales están ordenadas por tipo de trabajo y los libros de una biblioteca están clasificados por tema. Sin embargo, puede suceder que la búsqueda se realice sobre una colección de elementos no ordenados. Por ejemplo, cuando se busca la localización de una ciudad dentro de un mapa.

Se concluye que la operación de búsqueda se puede llevar a cabo sobre elementos ordenados o desordenados. Cabe destacar que la búsqueda es más fácil y ocupa menos tiempo cuando los datos se encuentran ordenados.

Los métodos de búsqueda se pueden clasificar en **internos** y **externos**, según la ubicación de los datos sobre los cuales se realizará la búsqueda. Se denomina **búsqueda interna** cuando todos los elementos se encuentran en la memoria principal de la computadora; por ejemplo, almacenados en arreglos, listas ligadas o árboles. Es **búsqueda externa** si los elementos están en memoria secundaria; es decir, si hubiera archivos en dispositivos como cintas y discos magnéticos.

En la siguiente sección se estudiarán los métodos internos, posteriormente en la sección 9.3 se hará lo propio con los externos.

9.2 BÚSQUEDA INTERNA

La **búsqueda interna** trabaja con elementos que se encuentran almacenados en la memoria principal de la máquina. Éstos pueden estar en estructuras estáticas —arreglos— o dinámicas —listas ligadas y árboles—. Los métodos de búsqueda interna más importantes son:

- Secuencial o lineal
- Binaria

- Por transformación de claves
- Árboles de búsqueda

9.2.1 Búsqueda secuencial

La **búsqueda secuencial** consiste en revisar elemento tras elemento hasta encontrar el dato buscado, o llegar al final del conjunto de datos disponible.

Primero se tratará sobre la búsqueda secuencial en arreglos, y luego en listas enlazadas. En el primer caso, se debe distinguir entre arreglos ordenados y desordenados.

Esta última consiste, básicamente, en recorrer el arreglo de izquierda a derecha hasta que se encuentre el elemento buscado o se termine el arreglo, lo que ocurra primero. Normalmente cuando una función de búsqueda concluye con éxito, interesa conocer en qué posición fue hallado el elemento que se estaba buscando. Esta idea se puede generalizar para todos los métodos de búsqueda.

A continuación se presenta el algoritmo de búsqueda secuencial en arreglos desordenados.

Algoritmo 9.1 Secuencial_desordenado

Secuencial_desordenado (*V*, *N*, *X*)

{Este algoritmo busca secuencialmente el elemento X en un arreglo unidimensional desordenado V, de N componentes}
{I es una variable de tipo entero}

1. Hacer $I \leftarrow 1$
2. Mientras $((I \leq N)$ y $(V[I] \neq X))$ *Repetir*
   Hacer $I \leftarrow I + 1$
3. {Fin del ciclo del paso 2}
4. *Si* $(I > N)$
   entonces
   Escribir "La información no está en el arreglo"
   si no
   Escribir "La información se encuentra en la posición", I
5. {Fin del condicional del paso 4}

Son dos los posibles resultados que se pueden obtener al aplicar este algoritmo: la posición en la que encontró el elemento, o un mensaje de fracaso si el elemento no se halla en el arreglo. Si hubiera dos o más ocurrencias del mismo valor, se encuentra la primera de ellas. Sin embargo, es posible modificar el algoritmo para obtener todas las ocurrencias del dato buscado.

A continuación se presenta una variante de este algoritmo, pero utilizando recursividad, en lugar de iteratividad.

Algoritmo 9.2 Secuencial_desordenado_recursivo

Secuencial_desordenado_recursivo (*V*, *N*, *X*, *I*)

{Este algoritmo busca secuencialmente, y de forma recursiva, al elemento *X* en el arreglo unidimensional desordenado *V*, de *N* componentes}
{*I* es un parámetro de tipo entero que inicialmente se encuentra en 1}

1. *Si* (I > *N*)
   entonces
   Escribir "La información no se encuentra en el arreglo"
   si no
   1.1 *Si* (*V*[*I*] = *X*)
   entonces
   Escribir "La información se encuentra en la posición", *I*
   si no
   Regresar a Secuencial_desordenado_recursivo con *V*, *N*, *X* e *I* + 1
   1.2 {Fin del condicional del paso 1.1}
2. {Fin del condicional del paso 1}

La búsqueda secuencial en arreglos ordenados es similar al caso anterior. Sin embargo, el orden entre los elementos del arreglo permite incluir una nueva condición que hace más eficiente al proceso. A continuación analicemos el algoritmo de búsqueda secuencial en arreglos ordenados.

Algoritmo 9.3 Secuencial_ordenado

Secuencial_ordenado (*V*, *N*, *X*)

{Este algoritmo busca secuencialmente al elemento *X* en un arreglo unidimensional ordenado *V*, de *N* componentes. *V* se encuentra ordenado crecientemente: *V*[1] ≤ *V*[2] ≤ ... ≤ *V*[*N*]}
{*I* es una variable de tipo entero}

1. Hacer *I* ← 1
2. Mientras ((*I* ≤ *N*) y (*X* > *V*[*I*])) *Repetir*
   Hacer *I* ← *I* + 1
3. {Fin del ciclo del paso 2)
4. *Si* ((*I* > *N*) o (*X* < *V*[*I*]))
   entonces
   Escribir "La información no se encuentra en el arreglo"
   si no
   Escribir "La información se encuentra en la posición", *I*
5. {Fin del condicional del paso 4}

Como el arreglo está ordenado, se establece una nueva condición: el elemento buscado tiene que ser mayor que el del arreglo. Cuando el ciclo se interrumpe, se evalúa cuál de las condiciones es falsa. Si ($I > N$) o si se comparó el elemento con un valor mayor a sí mismo ($X < V[I]$), se está ante un caso de fracaso: el elemento no está en el arreglo. Si $X = V[I]$ entonces se encontró al elemento en el arreglo.

A continuación se presenta el algoritmo de búsqueda en arreglos ordenados, pero en forma recursiva.

Algoritmo 9.4 Secuencial_ordenado_recursivo

Secuencial_ordenado_recursivo (*V, N, X, I*)

{Este algoritmo busca en forma secuencial y recursiva al elemento X en un arreglo unidimensional ordenado V, de N componentes. V se encuentra ordenado de manera creciente: $V[1] \leq V[2] \leq \ldots \leq V[N]$. I inicialmente tiene el valor de 1}

1. *Si* $((I \leq N)$ y $(X > V[I]))$
   entonces
      Llamar a Secuencial_ordenado_recursivo con V, N, X e $I + 1$
   si no
   1.1 *Si* $((I > N)$ o $(X < V[I]))$
      entonces
         Escribir "La información no se encuentra en el arreglo"
      si no
         Escribir "La información se encuentra en la posición", I
   1.2 {Fin del condicional del paso 1.1}
2. {Fin del condicional del paso 1}

El método de búsqueda secuencial también se puede aplicar a listas ligadas. Consiste en recorrer la lista nodo tras nodo, hasta encontrar al elemento buscado —éxito—, o hasta que lleguemos al final de la lista —fracaso—. La lista, como en el caso de arreglos, se puede encontrar ordenada o desordenada. El orden en el cual se puede recorrer la lista depende de sus características; puede ser simplemente ligada, circular o doblemente ligada. En este capítulo se presentará el caso de búsqueda secuencial en listas simplemente ligadas desordenadas. El lector, con los conocimientos que tiene sobre listas y búsqueda, puede implementar fácilmente los otros algoritmos.

Algoritmo 9.5 Secuencial_lista_desordenada

Secuencial_lista_desordenada (*P, X*)

{Este algoritmo busca en forma secuencial al elemento X en una lista simplemente ligada, que almacena información que está desordenada. P es un apuntador al primer nodo de la lista. INFO y LIGA son los campos de cada nodo}
{Q es una variable de tipo apuntador}

```
1. Hacer Q ← P
2. Mientras ((Q ≠ NIL) y (Q^.INFO ≠ X)) Repetir
       Hacer Q ← Q^.LIGA
3. {Fin del ciclo del paso 2}
4. Si (Q = NIL)
       entonces
           Escribir "La información no se encuentra en la lista"
       si no
           Escribir "La información se encuentra en la lista"
5. {Fin del condicional del paso 4}
```

Si la lista estuviera ordenada se modificaría este algoritmo, incluyendo una condición similar a la que se escribió en el algoritmo 9.3. Esto último con el objetivo de disminuir el número de comparaciones.

A continuación se presenta la variante recursiva de este algoritmo de búsqueda secuencial en listas simplemente ligadas desordenadas.

Algoritmo 9.6 Secuencial_lista_desordenada_recursivo

```
Secuencial_lista_desordenada_recursivo (P, X)

{Este algoritmo busca de manera secuencial y en forma recursiva al elemento X en una lista
simplemente ligada, que almacena información que está desordenada. P es un apuntador al
primer nodo de la lista. INFO y LIGA son los campos de cada nodo}

1. Si ((P ≠ NIL) y (P^.INFO ≠ X))
       entonces
           Regresar a Secuencial_lista_desordenada_recursivo con P^.LIGA y X
       si no
   1.1 Si (P = NIL)
           entonces
               Escribir "La información no se encuentra en la lista"
           si no
               Escribir "La información se encuentra en la lista"
   1.2 {Fin del condicional del paso 1.1}
2. {Fin del condicional del paso 1}
```

Análisis de la búsqueda secuencial

El número de comparaciones es uno de los factores más importantes que se utilizan para determinar la complejidad de los métodos de búsqueda. Para analizar la complejidad de la búsqueda secuencial, se deben establecer los casos más favorable o desfavorable que se presenten.

Al buscar, por ejemplo, un elemento en un arreglo unidimensional desordenado de N componentes, puede suceder que ese valor no se encuentre; por lo tanto, se harán N

comparaciones al recorrer todo el arreglo. Por otra parte, si el elemento se encuentra en el arreglo, éste puede estar en la primera posición, en la última o en alguna intermedia. Si es el primero, se hará una comparación; si se trata del último, se harán N comparaciones; y si se encuentra en la posición i ($1 < i < N$), entonces se realizarán i comparaciones.

Ahora bien, el número de comparaciones que se llevan a cabo si trabajamos con arreglos ordenados será el mismo que para desordenados, siempre y cuando el elemento se encuentre en el arreglo. Si no fuera éste el caso, entonces el número de comparaciones disminuirá sensiblemente en arreglos ordenados, siempre que el valor buscado esté comprendido entre el primero y el último elementos del arreglo.

Por otra parte, el número de comparaciones en la búsqueda secuencial en listas simplemente ligadas es el mismo que para arreglos. En la fórmula 9.1 se presentan los números mínimo, mediano y máximo de comparaciones que se ejecutan cuando se trabaja con la búsqueda secuencial.

Fórmula 9.1

$$C_{\text{mín}} = 1 \qquad C_{\text{med}} = \frac{(1+n)}{2} \qquad C_{\text{máx}} = N$$

La tabla 9.1 presenta, para distintos valores de N, los números mínimo, mediano y máximo de comparaciones que se requieren para buscar secuencialmente un elemento en un arreglo o lista ligada.

9.2.2 Búsqueda binaria

La **búsqueda binaria** consiste en dividir el intervalo de búsqueda en dos partes, comparando el elemento buscado con el que ocupa la posición central en el arreglo. Para el caso de que no fueran iguales se redefinen los extremos del intervalo, según el elemento central sea mayor o menor que el elemento buscado, disminuyendo de esta forma el espacio de búsqueda. El proceso concluye cuando el elemento es encontrado, o cuando el intervalo de búsqueda se anula, es vacío.

El método de búsqueda binaria funciona exclusivamente con arreglos ordenados. No se puede utilizar con listas simplemente ligadas —no podríamos retroceder para establecer intervalos de búsqueda— ni con arreglos desordenados. Con cada iteración del método el espacio de búsqueda se reduce a la mitad; por lo tanto, el número de com-

TABLA 9.1
Complejidad del método de búsqueda secuencial

N	$C_{\text{mín}}$	C_{med}	$C_{\text{máx}}$
10	1	5.5	10
100	1	50.5	100
500	1	250.5	500
1 000	1	500.5	1 000
10 000	1	5 000.5	10 000

paraciones a realizar disminuye notablemente. Esta disminución resulta significativa cuanto más grande sea el tamaño del arreglo. A continuación se presenta el algoritmo de búsqueda binaria.

Algoritmo 9.7 Binaria

Binaria (*V*, *N*, *X*)

{Este algoritmo busca al elemento *X* en un arreglo unidimensional ordenado *V* de *N* componentes}
{IZQ, CEN y DER son variables de tipo entero. BAN es una variable de tipo booleano}

```
1. Hacer IZQ ← 1, DER ← N y BAN ← FALSO
2. Mientras ((IZQ ≤ DER) y (BAN = FALSO)) Repetir
   2.1 Si (X = V[CEN])
            entonces
               Hacer BAN ← VERDADERO
            si no {Se redefine el intervalo de búsqueda}
         2.1.1 Si (X > V[CEN])
                  entonces
                     Hacer IZQ ← CEN + 1
                  si no
                     Hacer DER ← CEN – 1
         2.1.2 {Fin del condicional del paso 2.1.1)
   2.2 {Fin del condicional del paso 2.1)
3. {Fin del ciclo del paso 2)
4. Si (BAN = VERDADERO)
      entonces
         Escribir "La información está en la posición", CEN
      si no
         Escribir "La información no se encuentra en el arreglo"
5. {Fin del condicional del paso 4}
```

Analicemos ahora un ejemplo para ilustrar el funcionamiento de este algoritmo.

Ejemplo 9.1

Sea *V* un arreglo unidimensional de números enteros, ordenado de manera creciente, como se muestra en la figura 9.3.

En la tabla 9.2 se presenta el seguimiento del algoritmo 9.7 cuando *X* es igual a 325 (*X* = 325).

En la figura 9.4 se observa gráficamente, para este caso en particular, cómo se va reduciendo el intervalo de búsqueda.

FIGURA 9.3

V

101	215	325	410	502	507	600	610	612	670
1	2	3	4	5	6	7	8	9	10

TABLA 9.2
Búsqueda binaria

Paso	BAN	IZQ	DER	CEN	X = V[CEN]	X > V[CEN]
1	Falso	1	10	5	325 = 502 ? No	325 > 502 ? No
2	Falso	1	4	2	325 = 215 ? No	325 > 215 ? Sí
3	Falso	3	4	3	325 = 325 ? Sí	
4	Verdadero					

La tabla 9.3, por otra parte, muestra nuevamente el seguimiento del algoritmo 9.7 para $X = 615$, valor que no se encuentra en el arreglo.

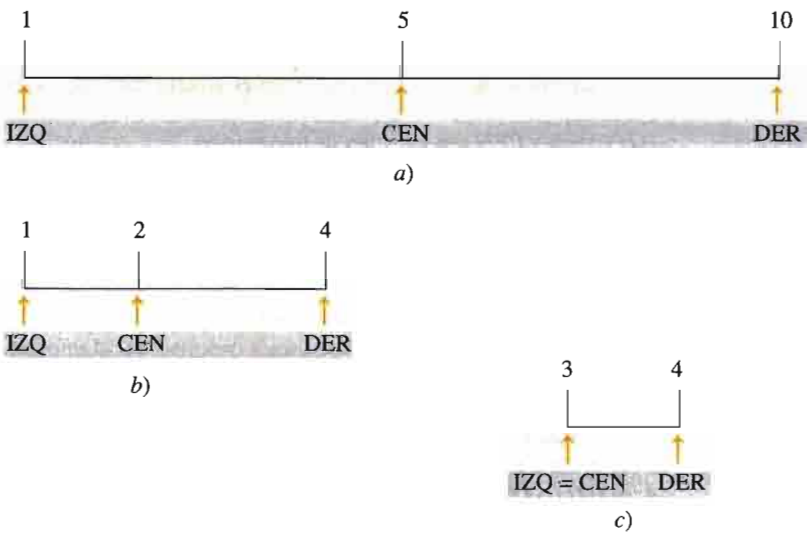
La figura 9.5 representa gráficamente cómo se va reduciendo el intervalo de búsqueda hasta anularse (DER < IZQ).

A continuación se presenta una variante del algoritmo de búsqueda binaria que no utiliza bandera —BAN—.

TABLA 9.3
Búsqueda binaria

Paso	BAN	IZQ	DER	CEN	X = V[CEN]	X > V[CEN]
1	Falso	1	10	5	615 = 502 ? No	615 > 502 ? Sí
2	Falso	6	10	8	615 = 610 ? No	615 > 610 ? Sí
3	Falso	9	10	9	615 = 612 ? No	615 > 612 ? Sí
4	Falso	10	10	10	615 = 670 ? No	615 > 670 ? No
5	Falso	10	9			

FIGURA 9.4
Reducción del intervalo de búsqueda. *a*) Paso 1. *b*) Paso 2. *c*) Paso 3 de la tabla 9.2.



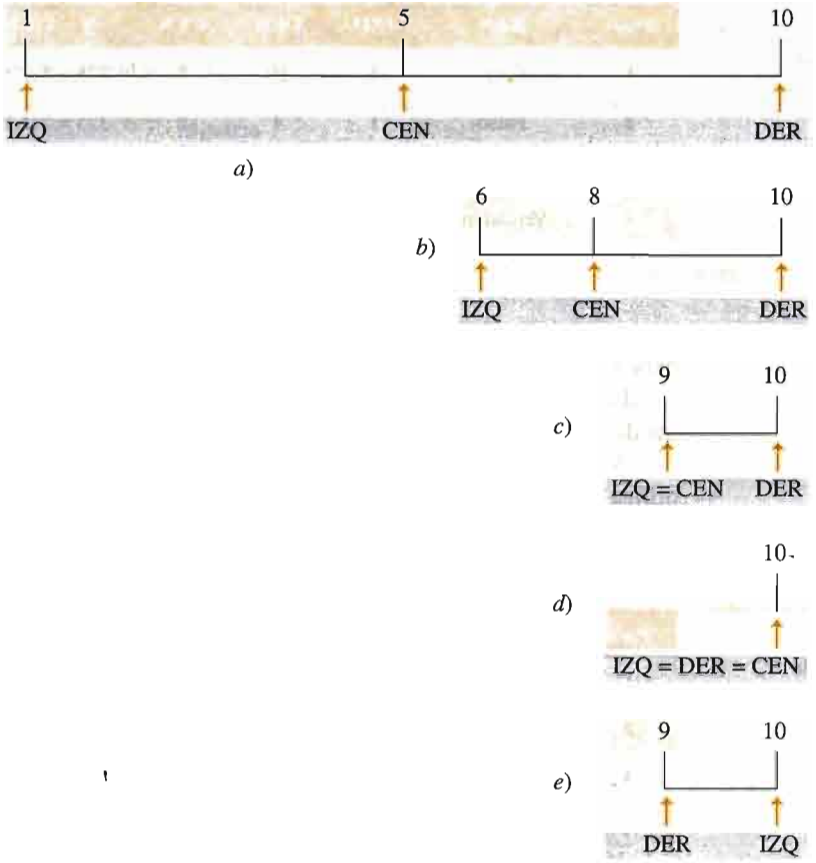


FIGURA 9.5
Reducción del intervalo de búsqueda. *a*) Paso 1. *b*) Paso 2. *c*) Paso 3. *d*) Paso 4. *e*) Paso 5 de la tabla 9.3.

Algoritmo 9.8 Binaria_sin_bandera

Binaria_sin_bandera (*V*, *N*, *X*)

{Este algoritmo busca al elemento *X* en el arreglo unidimensional ordenado *V* de *N* componentes}
{IZQ, DER y CEN son variables de tipo entero}

1. Hacer IZQ ← 1, DER ← *N* y CEN ← PARTE ENTERA ((IZQ + DER)/2)
2. Mientras ((IZQ ≤ DER) y (*X* ≠ *V*[CEN])) *Repetir*
   - 2.1 *Si* *X* > *V*[CEN]
     entonces
     Hacer IZQ ← CEN + 1
     si no
     Hacer DER ← CEN – 1
   - 2.2 {Fin del condicional del paso 2.1}
     Hacer CEN ← PARTE ENTERA ((IZQ + DER)/2)
3. {Fin del ciclo del paso 2}
4. *Si* (IZQ > DER)

```
    entonces
        Escribir "La información no se encuentra en el arreglo"
    si no
        Escribir "La información se encuentra en la posición", CEN
5. {Fin del condicional del paso 4}
```

Finalmente, se presenta una versión recursiva de este algoritmo de búsqueda binaria.

Algoritmo 9.9 Binaria_recursivo

```
Binaria_recursivo (V, IZQ, DER, X)

{Este algoritmo busca al elemento X en el arreglo unidimensional ordenado V de N componentes. IZQ ingresa inicialmente al algoritmo con el valor de 1. DER, por otra parte, ingresa con el valor de N}
{CEN es una variable de tipo entero}

1. Si (IZQ ≥ DER)
      entonces
         Escribir X, "No se encuentra en el arreglo"
      si no
         Hacer CEN ← PARTE ENTERA ((DER + IZQ)/2)
   1.1 Si (X = V[CEN])
         entonces
            Escribir "El dato buscado se encuentra en la posición", CEN
         si no
      1.1.1 Si (X > V[CEN])
               entonces
                  Regresar a Binaria_recursivo con V, CEN + 1, DER, X
               si no
                  Regresar a Binaria_recursivo con V, IZQ, CEN – 1, X
      1.1.2 {Fin del condicional del paso 1.1.1}
   1.2 {Fin del condicional del paso 1.1}
2. {Fin del condicional del paso 1}
```

Análisis de la búsqueda binaria

Para analizar la complejidad del método de búsqueda binaria es necesario establecer los casos más favorables y desfavorables que se pudieran presentar en el proceso de búsqueda. El primero sucede cuando el elemento buscado es el central, en dicho caso se hará una sola comparación; el segundo sucede cuando el elemento no se encuentra en el arreglo; entonces se harán aproximadamente $\log_2 (n)$ comparaciones, ya que con cada comparación el número de elementos en los cuales se debe buscar se reduce en un factor de 2. De esta forma, se determinan los números mínimo, mediano y máximo de comparaciones que se deben realizar cuando se utiliza este tipo de búsqueda.

$$C_{mín} = 1 \qquad C_{med} = \frac{(1+\log_2(N))}{2} \qquad C_{máx} = \log_2(N)$$

Fórmula 9.2

En la tabla 9.4 se presentan, para distintos valores de *N*, los números mínimo, mediano y máximo de comparaciones requeridas para buscar un elemento en un arreglo aplicando el método de búsqueda binaria.

Si se comparan los valores de la tabla 9.1 con los de la tabla 9.4 resulta claro que el método de búsqueda binaria es más eficiente que el método de búsqueda secuencial. Además, la diferencia se hace más significativa conforme más grande sea el tamaño del arreglo. Sin embargo, no hay que olvidar que el método de búsqueda binaria trabaja solamente con arreglos ordenados; por lo tanto, si el arreglo estuviera desordenado antes de emplear este método, aquél debería ordenarse.

Cabe destacar, sin embargo, que la ordenación de un arreglo también implica comparaciones y movimientos de elementos, así que si se va a realizar sólo una búsqueda sobre un arreglo desordenado conviene utilizar el método secuencial. En cambio, si se realizan búsquedas en forma continua, convendría ordenarlo para poder aplicar el método de búsqueda binaria.

9.2.3 Búsqueda por transformación de claves

Los dos métodos analizados anteriormente permiten encontrar un elemento en un arreglo. En ambos casos el tiempo de búsqueda es proporcional a su número de componentes. Es decir, a mayor número de elementos se debe realizar mayor número de comparaciones. Se mencionó además que si bien el método de búsqueda binaria es más eficiente que el secuencial, existe la restricción de que el arreglo se debe encontrar ordenado.

Esta sección se dedica a un nuevo método de búsqueda. Este método, conocido como **transformación de claves** o ***hash***, permite aumentar la velocidad de búsqueda sin necesidad de tener los elementos ordenados. Cuenta con la ventaja de que el tiempo de búsqueda es independiente del número de componentes del arreglo.

Supongamos que se tiene una colección de datos, cada uno de ellos identificado por una clave. Es claro que resulta atractivo tener acceso a ellos de manera directa; es decir, sin recorrer algunos datos antes de localizar al buscado. El método por transformación de claves permite realizar justamente esta actividad; es decir, localizar el dato en forma

TABLA 9.4
Complejidad del método de búsqueda binaria

N	$C_{mín}$	C_{med}	$C_{máx}$
10	1	2.5	4
100	1	4	7
500	1	5	9
1 000	1	5.5	10
10 000	1	7.5	14

directa. El método trabaja utilizando una función que convierte una clave dada en una dirección —índice— dentro del arreglo.

$$\text{dirección} \leftarrow H \text{ (clave)}$$

La función *hash* (H) aplicada a la clave genera un índice del arreglo que permite acceder directamente al elemento. El caso más trivial se presenta cuando las claves son números enteros consecutivos.

Supongamos que se desea almacenar la información relacionada con 100 alumnos cuyas matrículas son números del 1 al 100. En este caso conviene definir un arreglo de 100 elementos con índices numéricos comprendidos entre los valores 1 y 100. Los datos de cada alumno ocuparán la posición del arreglo que se corresponda con el número de la matrícula; de esta manera se podrá acceder directamente a la información de cada alumno.

Consideremos ahora que se desea almacenar la información de 100 empleados. La clave de cada empleado corresponde al número de su seguro social. Si la clave está formada por 11 dígitos, resulta por completo ineficiente definir un arreglo con 99 999 999 999 elementos para almacenar solamente los datos de los 100 empleados. Utilizar un arreglo tan grande asegura la posibilidad de acceder directamente a sus elementos; sin embargo, el costo en memoria resulta tanto ridículo como excesivo. Siempre es importante equilibrar el costo del espacio de memoria con el costo por tiempo de búsqueda.

Cuando se tienen claves que no se corresponden con índices —por ejemplo, por ser alfanuméricas—, o cuando las claves representen valores numéricos muy grandes o no se corresponden con los índices de los arreglos, se utilizará una función *hash* que permita transformar la clave para obtener una dirección apropiada. Esta función *hash* debe ser simple de calcular y asignar direcciones de la manera más uniforme posible. Es decir, debe generar posiciones diferentes dadas dos claves también diferentes. Si esto último no ocurre ($H(K_1) = d$, $H(K_2) = d$ y $K_1 \neq K_2$) hay una **colisión**, que se define como la asignación de una misma dirección a dos o más claves distintas.

En este contexto, para trabajar con este método de búsqueda se debe seleccionar previamente:

- Una función *hash* que sea fácil de calcular y distribuya uniformemente las claves.
- Un método para resolver colisiones. Si éstas se presentan, se contará con algún método que genere posiciones alternativas.

Estos dos casos se tratarán en forma separada. Como ya se mencionó, seleccionar una buena función *hash* es muy importante, pero es difícil encontrarla. Básicamente porque no existen reglas que permitan determinar cuál será la función más apropiada para un conjunto de claves que asegure la máxima uniformidad en su distribución. Realizar un análisis de las principales características de las claves siempre ayuda en la elección de una función de este tipo. A continuación se explican algunas de las funciones *hash* más utilizadas.

9.2.4 Función *hash* por módulo: división

La función ***hash* por módulo** o **división** consiste en tomar el residuo de la división de la clave entre el número de componentes del arreglo. Supongamos, por ejemplo, que se

tiene un arreglo de N elementos, y K es la clave del dato a buscar. La función *hash* queda definida por la siguiente fórmula:

$$H(K) = (K \bmod N) + 1 \qquad \textbf{Fórmula 9.3}$$

En la fórmula 9.3 se observa que al residuo de la división se le suma 1, esto último con el objetivo de obtener un valor comprendido entre 1 y N.

Para lograr mayor uniformidad en la distribución, es importante que N sea un número primo o divisible entre muy pocos números. Por lo tanto, si N no es un número primo, se debe considerar el valor primo más cercano.

En el ejemplo 9.2 se presenta un caso de función *hash* por módulo.

Ejemplo 9.2

Supongamos que $N = 100$ es el tamaño del arreglo, y las direcciones que se deben asignar a los elementos (al guardarlos o recuperarlos) son los números del 1 al 100. Consideremos además que $K_1 = 7\,259$ y $K_2 = 9\,359$ son las dos claves a las que se deben asignar posiciones en el arreglo. Si aplicamos la fórmula 9.3 con $N = 100$, para calcular las direcciones correspondientes a K_1 y K_2, obtenemos:

$$H(K_1) = (7\,259 \bmod 100) + 1 = 60$$
$$H(K_2) = (9\,359 \bmod 100) + 1 = 60$$

Como $H(K_1)$ es igual a $H(K_2)$ y K_1 es distinto de K_2, se está ante una colisión que se debe resolver porque a los dos elementos le correspondería la misma dirección.

Observemos, sin embargo, que si aplicáramos la fórmula 9.3 con un número primo cercano a N, el resultado cambiaría:

$$H(K_1) = (7\,259 \bmod 97) + 1 = 82$$
$$H(K_2) = (9\,359 \bmod 97) + 1 = 48$$

Con $N = 97$ se ha eliminado la colisión.

9.2.5 Función *hash* cuadrado

La función ***hash* cuadrado** consiste en elevar al cuadrado la clave y tomar los dígitos centrales como dirección. El número de dígitos que se debe considerar se encuentra determinado por el rango del índice. Sea K la clave del dato a buscar, la función *hash* cuadrado queda definida, entonces, por la siguiente fórmula:

$$H(K) = \text{dígitos\_centrales}\,(K^2) + 1 \qquad \textbf{Fórmula 9.4}$$

La suma de una unidad a los dígitos centrales es útil para obtener un valor comprendido entre 1 y N.

En el ejemplo 9.3 se presenta un caso de función *hash* cuadrado.

Ejemplo 9.3

Sea $N = 100$ el tamaño del arreglo, y sus direcciones los números comprendidos entre 1 y 100. Sean $K_1 = 7\,259$ y $K_2 = 9\,359$ dos claves a las que se deben asignar posiciones en el arreglo. Se aplica la fórmula 9.4 para calcular las direcciones correspondientes a K_1 y K_2:

$K_1^2 = 52\,693\,081$
$K_2^2 = 87\,590\,881$

$H(K_1)$ = dígitos_centrales (52 693 081) + 1 = 94
$H(K_2)$ = dígitos_centrales (87 590 881) + 1 = 91

Como el rango de índices en nuestro ejemplo varía de 1 a 100, se toman solamente los dos dígitos centrales del cuadrado de las claves.

9.2.6 Función *hash* por plegamiento

La **función *hash* por plegamiento** consiste en dividir la clave en partes, tomando igual número de dígitos aunque la última puede tener menos, y operar con ellas, asignando como dirección los dígitos menos significativos. La operación entre las partes se puede realizar por medio de sumas o multiplicaciones. Sea K la clave del dato a buscar. K está formada por los dígitos $d_1, d_2, ..., d_n$. La función *hash* por plegamiento queda definida por la siguiente fórmula:

$$H(K) = \text{dígmensig}((d_1 \ldots d_i) + (d_{i+1} \ldots d_j) + \ldots + (d_1 \ldots d_n)) + 1 \qquad \textbf{Fórmula 9.5}$$

El operador que aparece en la fórmula operando las partes de la clave es el de suma, pero, como ya se aclaró, puede ser el de la multiplicación. En este contexto, la suma de una unidad a los dígitos menos significativos —dígmensig— es para obtener un valor comprendido entre 1 y N.

En el ejemplo 9.4 se presenta un caso de función *hash* por plegamiento.

Ejemplo 9.4

Sea $N = 100$ el tamaño del arreglo, y las direcciones que deben tomar sus elementos los números comprendidos entre 1 y 100. Sean $K_1 = 7\,259$ y $K_2 = 9\,359$ dos claves a las que se deben asignar posiciones en el arreglo. Se aplica la fórmula 9.5 para calcular las direcciones correspondientes a K_1 y K_2.

$H(K_1)$ = dígmensig (72 + 59) + 1 = dígmensig (131) + 1 = 32
$H(K_2)$ = dígmensig (93 + 59) + 1 = dígmensig (152) + 1 = 53

De la suma de las partes se toman solamente dos dígitos porque los índices del arreglo varían de 1 a 100.

9.2.7 Función *hash* por truncamiento

La **función *hash* por truncamiento** consiste en tomar algunos dígitos de la clave y formar con ellos una dirección. Este método es de los más sencillos, pero es también de los que ofrecen menos uniformidad en la distribución de las claves.

Sea K la clave del dato a buscar. K está formada por los dígitos d_1, d_2, ..., d_n. La función *hash* por truncamiento se representa con la siguiente fórmula:

$$H(K) = \text{elegirdígitos } (d_1, d_2 \ldots d_n) + 1 \qquad \textbf{Fórmula 9.6}$$

La elección de los dígitos es arbitraria. Se podrían tomar los de las posiciones impares o de las pares. Luego se podrían unir de izquierda a derecha o de derecha a izquierda. La suma de una unidad a los dígitos seleccionados es útil para obtener un valor entre 1 y 100.

En el ejemplo 9.5 se muestra un caso de función *hash* por truncamiento.

Ejemplo 9.5

Sea $N = 100$ el tamaño del arreglo, y las direcciones de sus elementos los números entre 1 y 100. Sean $K_1 = 7\,259$ y $K_2 = 9\,359$ dos claves a las que se deben asignar posiciones en el arreglo. Se aplica la fórmula 9.6 para calcular las direcciones correspondientes a K_1 y K_2.

$H(K_1)$ = elegirdígitos (7 259) + 1 = 75 + 1 = 76
$H(K_2)$ = elegirdígitos (9 359) + 1 = 95 + 1 = 96

En este ejemplo se toman el primero y tercer números de la clave y se unen de izquierda a derecha.

Es importante destacar que en todos los casos anteriores se presentaron ejemplos de claves numéricas. Sin embargo, en la práctica las claves pueden ser alfabéticas o alfanuméricas. En general, cuando aparecen letras en las claves se suele asociar a cada una un entero con el propósito de convertirlas en numéricas.

A	B	C	D	... Z
01	02	03	04	... 27

Si, por ejemplo, la clave fuera **ADA**, su equivalente numérica sería **010401**. Si hubiera combinación de letras y números, se procedería de la misma manera. Por ejemplo. dada una clave **Z4F21**, su equivalente numérica sería **2740621**. Otra alternativa sería tomar el valor decimal asociado para cada carácter según el código ASCII. Una vez obtenida la clave en su forma numérica, se puede utilizar normalmente cualesquiera de las funciones antes mencionadas. El ejemplo 9.11 ilustra un caso de clave alfabética.

9.2.8 Solución de colisiones

La elección de un método adecuado para resolver **colisiones** es tan importante como la elección de una buena función *hash*. Cuando ésta obtiene una misma dirección para dos claves diferentes, se está ante una colisión. Normalmente, cualquiera que sea el método

elegido resulta costoso tratar las colisiones. Es por ello que se debe hacer un esfuerzo importante para encontrar la función que ofrezca la mayor uniformidad en la distribución de las claves.

La manera más natural de resolver el problema de las colisiones es reservar una casilla por clave; es decir, aquellas que se correspondan una a una con las posiciones del arreglo. Pero, como ya se mencionó, esta solución puede tener un alto costo en memoria; por lo tanto, se deben analizar otras alternativas que permitan equilibrar el uso de memoria con el tiempo de búsqueda.

En adelante se estudiarán algunos de los métodos más utilizados para resolver colisiones, que se pueden clasificar en:

- Reasignación
- Arreglos anidados
- Encadenamiento

9.2.9 Reasignación

Existen varios métodos que trabajan bajo el principio de comparación y reasignación de elementos. A continuación se analizarán tres de ellos:

- Prueba lineal
- Prueba cuadrática
- Doble dirección *hash*

Prueba lineal

El método de **prueba lineal** consiste en que una vez que se detecta la colisión, se recorre el arreglo secuencialmente a partir del punto de colisión, buscando al elemento. El proceso de búsqueda concluye cuando el elemento es hallado, o cuando se encuentra una posición vacía. El arreglo se trata como una estructura circular: *el siguiente elemento después del último es el primero*.

A continuación se expone el algoritmo de solución de colisiones por medio de la prueba lineal.

Algoritmo 9.10 Prueba_lineal

Prueba_lineal (*V*, *N*, *K*)

{Este algoritmo busca al dato con clave *K* en el arreglo unidimensional *V* de *N* elementos. Resuelve el problema de las colisiones por medio del método de prueba lineal}
{*D* y *DX* son variables de tipo entero}

1. Hacer $D \leftarrow H(K)$ {Genera dirección}
2. *Si* ((*V*[*DX*] ≠ VACÍO) y (*V*[*D*] = *K*))
   entonces

```
        Escribir "La información está en la posición", D
    si no
        Hacer DX ← D + 1
2.1 Mientras ((DX ≤ N) y (V[DX] ≠ VACÍO) y (V[DX] ≠ K) y (DX ≠ D))
    Repetir
        Hacer DX ← DX + 1
2.1.1 Si (DX = N + 1) entonces
        Hacer DX ← 1
2.1.2 {Fin del condicional del paso 2.1.1}
2.2 {Fin del ciclo del paso 2.1}
2.3 Si ((V[DX] = VACÍO) o (DX = D))
    entonces
        Escribir "La información no se encuentra en el arreglo"
    si no
        Escribir "La información está en la posición", DX
2.4 {Fin del condicional del paso 2.3}
3. {Fin del condicional del paso 2}
```

La cuarta condición del ciclo del punto 2.1, $(DX \neq X)$, es para evitar caer en un ciclo infinito si el arreglo estuviera lleno y el elemento a buscar no se encontrara en él.

La principal desventaja de este método es que puede haber un fuerte agrupamiento alrededor de ciertas claves, mientras que otras zonas del arreglo podrían permanecer vacías. Si las concentraciones de claves son muy frecuentes, la búsqueda será principalmente secuencial, perdiendo así las ventajas del método *hash*. El ejemplo 9.6 ilustra el funcionamiento del algoritmo 9.10.

Ejemplo 9.6

Sea V un arreglo unidimensional de 10 elementos. Las claves 25, 43, 56, 35, 54, 13, 80 y 104 fueron asignadas según la función *hash*:

$$H(K) = (K \bmod 10) + 1$$

En la figura 9.6 se aprecia el estado de arreglo (9.6*a*) y la tabla con $H(K)$ para cada clave (9.6*b*).

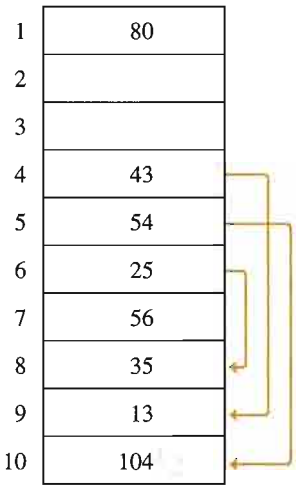
En la tabla 9.5 se presenta el seguimiento de las variables importantes del algoritmo 9.10 para el caso del ejemplo anterior. El dato a buscar es igual a 35.

Al aplicar la función hash a la clave 35, se obtiene una dirección (D) igual a 6. Sin embargo, en esa posición no se encuentra el elemento buscado, por lo que se comienza a recorrer secuencialmente el arreglo a partir de la posición (DX) igual a 7. En este caso la búsqueda concluye cuando se encuentra al valor buscado en la posición 8.

TABLA 9.5
Solución de colisiones por la prueba lineal. $K = 35$

D	DX
6	7
	8

FIGURA 9.6
Solución de colisiones por la prueba lineal. *a*) Arreglo. *b*) Tabla con *H*(*K*).



a)

K	*H*(*K*)
25	6
43	4
56	7
35	6
54	5
13	4
80	1
104	5

b)

En la tabla 9.6 se presenta ahora el seguimiento de las variables importantes del algoritmo 9.10, pero ahora para un caso más complejo del ejemplo anterior. El dato a buscar es igual a 13.

Prueba cuadrática

El método de la **prueba cuadrática** es similar al anterior. La diferencia consiste en que en el de la prueba cuadrática las direcciones alternativas se generarán como $D + 1$, $D + 4$, $D + 9$, ..., $D + i^2$ en vez de $D + 1$, $D + 2$, ..., $D + i$. Esta variación permite una mejor distribución de las claves que colisionan.

A continuación se presenta el algoritmo de solución de colisiones por medio de la prueba cuadrática.

TABLA 9.6
Solución de colisiones por la prueba lineal. $K = 13$

D	*DX*
4	5
	6
	7
	8
	9

Algoritmo 9.11 Prueba_cuadrática

Prueba_cuadrática (*V*, *N*, *K*)

{Este algoritmo busca al dato con clave *K* en el arreglo unidimensional *V* de *N* elementos. Resuelve el problema de las colisiones por medio de la prueba cuadrática}
{*D*, *DX* e *I* son variables de tipo entero}

```
1. Hacer D ← H(K) {Genera dirección}
2. Si ((V[DX] ≠ VACÍO) y (V[D] = K))
     entonces
        Escribir "La información está en la posición", D
     si no
        Hacer I ← 1 y DX ← (D + (I * I))
     2.1 Mientras ((V[DX] ≠ VACÍO) y (V[DX] ≠ K)) Repetir
            Hacer I ← I + 1 y DX ← (D + (I * I))
        2.1.1 Si (DX > N) entonces
                 Hacer I ← 0, DX ← 1 y D ← 1
        2.1.2 {Fin del condicional del paso 2.1.1}
     2.2 {Fin del ciclo del paso 2.1}
     2.3 Si (V[DX] = VACÍO)
            entonces
               Escribir "La información no está en el arreglo"
            si no
               Escribir "La información está en la posición", DX
     2.4 {Fin del condicional del paso 2.3}
3. {Fin del condicional del paso 2}
```

A continuación se presenta un ejemplo que ilustra el funcionamiento del algoritmo 9.11.

Ejemplo 9.7

Sea *V* un arreglo unidimensional de diez elementos. Las claves 25, 43, 56, 35, 54, 13, 80, 104 y 55 se asignaron según la función *hash*:

$$H(K) = (K \bmod 10) + 1$$

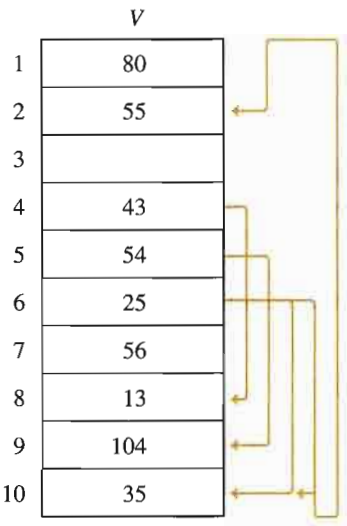
En la figura 9.7 se presenta el estado del arreglo (9.7*a*) y la tabla con $H(K)$ para cada clave (9.7*b*).

La tabla 9.7 contiene el seguimiento de las variables importantes del algoritmo 9.11 para el caso del ejemplo anterior, y el dato a buscar es igual a 35.

TABLA 9.7
Solución de colisiones por la prueba cuadrática. $K = 35$

D	*I*	*DX*
6	1	7
	2	10

FIGURA 9.7
Solución de colisiones por la prueba cuadrática. *a*) Arreglo. *b*) Tabla con *H*(*K*).



a)

K	*H*(*K*)
25	6
43	4
56	7
35	6
54	5
13	4
80	1
104	5
55	6

b)

Al aplicar la función *hash* a la clave 35, se obtiene una dirección (*D*) igual a 6; sin embargo, en esa dirección no se encuentra el elemento buscado. Se calcula posteriormente *DX*, como la suma *D* + (*I* * *I*), obteniéndose de esta forma la dirección 7. El algoritmo de búsqueda concluye cuando se encuentra el valor deseado en la décima posición del arreglo.

En la tabla 9.8 se presenta el seguimiento de las variables importantes del algoritmo 9.11 para un caso más complejo que el anterior. El dato a buscar es 55.

Doble dirección *hash*

El método de **doble dirección *hash*** consiste en que una vez que se detecta la colisión, se genera otra dirección aplicando la misma función *hash* a la dirección previamente obtenida. El proceso se detiene cuando el elemento es hallado, o cuando se encuentra una posición vacía.

TABLA 9.8
Solución de colisiones por la prueba cuadrática. *K* = 55

D	*I*	*DX*
6	1	7
	2	10
	3	15
1	0	1
	1	2

$DH(K)$
$D'(H(D))$
$D''(H(D'))$
...

La función *hash* que se aplica no necesariamente tiene que ser la misma que originalmente se aplicó a la clave; podría ser cualquier otra. Sin embargo, no existe ningún estudio que precise cuál es la mejor función que se debe utilizar en el cálculo de las direcciones sucesivas.

Analicemos ahora el algoritmo de solución de colisiones por medio del método de la doble dirección *hash*.

Algoritmo 9.12 Doble_dirección

Doble_dirección (V, N, K)

{Este algoritmo busca al dato con la clave K en el arreglo unidimensional V de N elementos. Resuelve el problema de las colisiones por medio de la doble dirección *hash*}
{D y DX son variables de tipo entero}

1. Hacer $D \leftarrow H(K)$
2. *Si* (($V[DX] \neq$ VACÍO) y ($V[D] = K$))
   entonces
      Escribir "La información se encuentra en la posición", D
   si no
      Hacer $DX \leftarrow H'(D)$
   2.1 Mientras (($DX \leq N$) y ($V[DX] \neq$ VACÍO) y ($V[DX] \neq K$) y ($DX \neq D$)) *Repetir*
         Hacer $DX \leftarrow H'(DX)$
   2.2 {Fin del ciclo del paso 2.1}
   2.3 *Si* (($V[DX]$ = VACÍO) o ($V[DX] \neq K$))
         entonces
            Escribir "La información buscada no está en el arreglo"
         si no
            Escribir "La información está en la posición", DX
   2.4 {Fin del condicional del paso 2.3}
3. {Fin del condicional del paso 2}

El siguiente ejemplo ilustra el funcionamiento de este algoritmo.

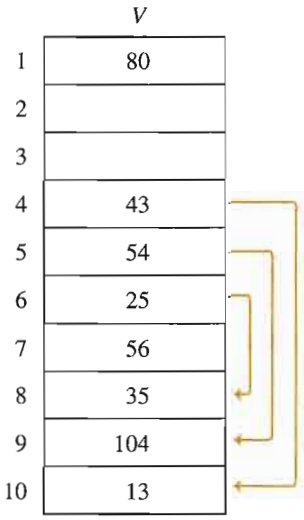
Ejemplo 9.8

Sea V un arreglo unidimensional de diez elementos. Las claves 25, 43, 56, 35, 54, 13, 80 y 104 fueron asignadas según la función *hash*:

$$H(K) = (K \bmod 10) + 1$$

Además se definió una función H' para calcular direcciones alternativas en caso de haber colisión.

FIGURA 9.8
Solución de colisiones por el método de doble dirección *hash*. *a*) Arreglo. *b*) Tabla con $H(K)$, $H'(D)$, $H'(D')$, $H'(D'')$.



a)

K	$H(K)$	$H'(D)$	$H'(D')$	$H'(D'')$
25	6	–	–	–
43	4	–	–	–
56	7	–	–	–
35	6	8	–	–
54	5	–	–	–
13	4	6	8	10
80	1	–	–	–
104	5	7	9	–

b)

$$H'(D) = ((D + 1) \bmod 10) + 1$$

En la figura 9.8 se presenta el estado del arreglo (9.8*a*) y la tabla (9.8*b*) con $H(K)$ para cada clave, y $H'(D)$ en caso de colisión.

En la tabla 9.9 se presenta el seguimiento del algoritmo 9.12 para el caso del ejemplo anterior. El dato a buscar es igual a 13.

Al aplicar la función *hash* (H) a la clave 13, se obtuvo una dirección (D) igual a 4. Como en esa posición no se encuentra el elemento buscado, se aplica reiteradamente H', generando direcciones hasta localizar el valor deseado. En este ejemplo fue preciso aplicar tres veces la función H', obteniéndose las direcciones 6, 8 y 10, en la que finalmente se encontró el dato buscado.

9.2.10 Arreglos anidados

El método de **arreglos anidados** consiste en que cada elemento del arreglo tenga otro arreglo, en el cual se almacenen los elementos que colisionan. Si bien la solución parece ser sencilla, es claro que resulta ineficiente. Al trabajar con arreglos se depende del espacio que se haya asignado a éstos, lo cual conduce a un nuevo problema difícil

TABLA 9.9
Solución de colisiones por el método de doble dirección *hash*. $K = 13$

D	DX
4	6
	8
	10

de solucionar: elegir un tamaño adecuado de arreglo que permita un equilibrio entre el costo de memoria y el número de valores —que colisionan— que pudiera almacenar. Analicemos un ejemplo.

Ejemplo 9.9

Sea V un arreglo unidimensional de diez elementos. Los elementos con claves 25, 43, 56, 35, 54, 13, 80 y 104 se almacenaron en el arreglo unidimensional V utilizando la función *hash*:

$$H(K) = (K \bmod 10) + 1$$

En la figura 9.9 se presenta el estado del arreglo anidado (9.9*a*) y la tabla con $H(K)$ para cada clave (9.9*b*).

9.2.11 Encadenamiento

El método de **encadenamiento** consiste en que cada elemento del arreglo tenga un apuntador a una lista ligada, la cual se irá generando y almacenará los valores que colisionan. Es el método más eficiente debido al dinamismo propio de las listas. Cualquiera que sea el número de colisiones que se presenten, se podrán resolver sin inconvenientes.

Como desventajas del método de encadenamiento se menciona el hecho de que ocupa espacio adicional al de la tabla y que exige el manejo de listas ligadas. Además, si las listas crecen demasiado se perderá la facilidad de acceso directo del método *hash*.

La figura 9.10 muestra la estructura de datos necesaria para resolver colisiones por medio del método de encadenamiento.

A continuación se presenta el algoritmo de solución de colisiones por encadenamiento.

FIGURA 9.9
Solución de colisiones con arreglos anidados.
a) Arreglo anidado.
b) Tabla con $H(K)$.

V

1	80			. . .	
2					
3					
4	43	13			
5	54	104			
6	25	35			
7	56				
8					
9					
10					

a)

K	$H(K)$
25	6
43	4
56	7
35	6
54	5
13	4
80	1
104	5

b)

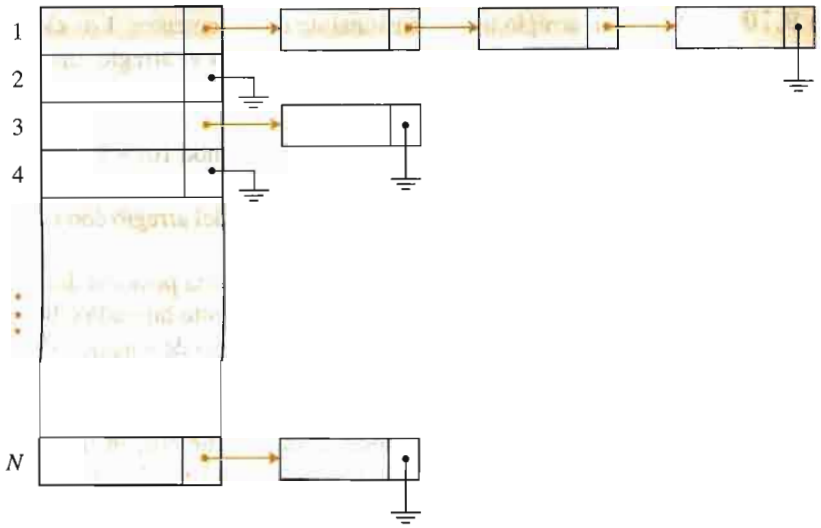


FIGURA 9.10
Solución de colisiones por encadenamiento.

Algoritmo 9.13 Encadenamiento

Encadenamiento (*V*, *N*, *K*)

{Este algoritmo busca al dato con clave *K* en el arreglo unidimensional *V* de *N* elementos. Resuelve las colisiones por medio de encadenamiento en listas ligadas. SIG e INFO son los campos de cada nodo de la lista}
{*D* es una variable de tipo entero. *Q* es una variable de tipo puntero}

```
1. Hacer D ← H(K) {Genera dirección}
2. Si ((V[D] ≠ VACÍO) y (V[D] = K))
      entonces
         Escribir "La información está en la posición", D
      si no
         Hacer Q ← V[D].SIG {Apuntador a la lista}
      2.1 Mientras ((Q ≠ VACÍO) y (Q^.INFO ≠ K))
            Hacer Q ← Q^.SIG
      2.2 {Fin del ciclo del paso 2.1}
      2.3 Si (Q = VACÍO)
            entonces
               Escribir "La información no se encuentra en la lista"
            si no
               Escribir "La información se encuentra en la lista"
      2.4 {Fin del condicional del paso 2.3}
3. {Fin del condicional del paso 2}
```

El funcionamiento de este algoritmo queda más claro con el siguiente ejemplo.

Ejemplo 9.10

Sea V un arreglo unidimensional de diez elementos. Los elementos con claves 25, 43, 56, 35, 54, 13, 80 y 104 se almacenaron en el arreglo unidimensional V utilizando la siguiente función *hash*:

$$H(K) = (K \bmod 10) + 1$$

En la figura 9.11 se presenta el estado del arreglo con encadenamiento (9.11*a*) y la tabla con $H(K)$ para cada clave (9.11*b*).

Una vez detectada la colisión en una cierta posición del arreglo, se debe recorrer la lista asociada a ella hasta encontrar el elemento buscado o llegar a su final.

En el ejemplo 9.11 se presenta otro caso de solución de colisiones por encadenamiento donde las claves son alfabéticas.

Ejemplo 9.11

Sea P un arreglo unidimensional de diez elementos, en el cual se almacenan los datos de algunos pinos mexicanos. Se utiliza como clave el nombre de los pinos para asignar a cada uno de ellos una dirección en el arreglo P. Para ello primero se obtendrá un número que resultará de sustituir cada letra por un dígito (del 01 al 27), y a este número se le aplicará la función *hash* (H) definida de la siguiente manera:

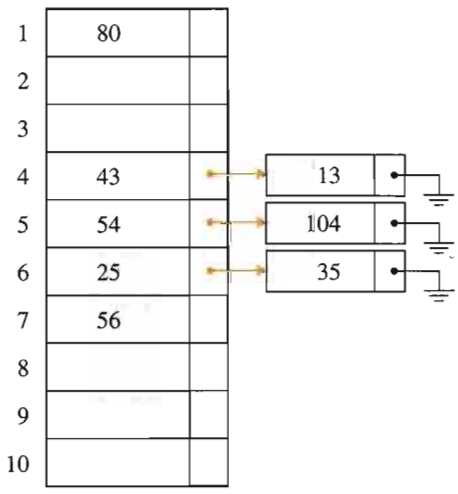
$$H\ (\text{clave}) = (\text{clave} \bmod 10) + 1$$

La tabla 9.10 contiene los nombres de los pinos, el valor numérico asociado (clave) y la dirección en P que le corresponde.

Como se puede apreciar en la tabla, ha habido colisiones. Para resolverlas, se aplicará el método de encadenamiento. La estructura resultante se muestra en la figura 9.12.

Cabe destacar que cualquiera que sea el método seleccionado para resolver las colisiones, se debe tener en cuenta en qué estado queda la estructura al insertar y, sobre todo, al eliminar elementos. La eliminación es la operación que más afecta cuando se

FIGURA 9.11
Solución de colisiones por encadenamiento.



a)

K	$H(K)$
25	6
43	4
56	7
35	6
54	5
13	4
80	1
104	5

b)

TABLA 9.10

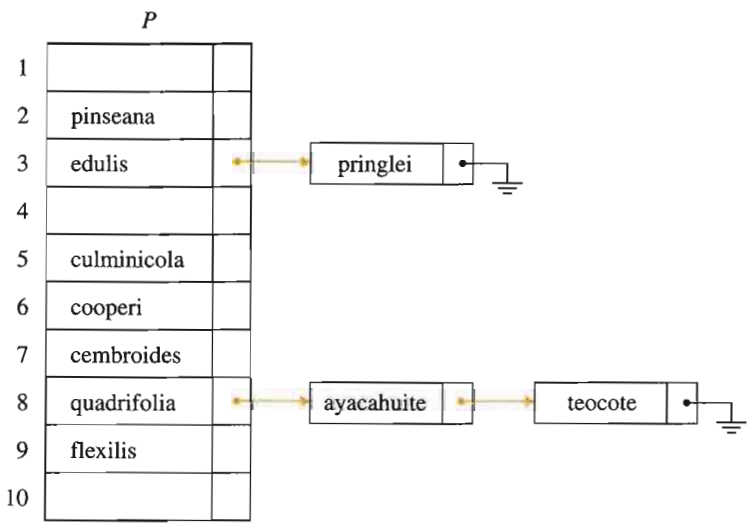
Nombre	Valor numérico	Dirección
Cembroides	96	7
Edulis	72	3
Culminicola	114	5
Quadrifolia	117	8
Pinseana	81	2
Flexilis	98	9
Ayacahuite	97	8
Teocote	87	8
Cooperi	85	6
Pringlei	92	3

tienen colisiones, por lo que se le debe dedicar especial atención para no perder eficiencia en la búsqueda.

Análisis del método por transformación de claves

Para analizar la complejidad de este método es necesario realizar varios cálculos probabilísticos, que no se estudiarán en esta obra. La dificultad del análisis se debe principalmente a que no sólo interviene la función *hash* sino también el método utilizado para resolver las colisiones. Por lo tanto, se debería analizar cada una de las posibles combinaciones que se pudieran presentar.

FIGURA 9.12
Solución de colisiones por encadenamiento.



Sea λ el factor de ocupación de un arreglo, definido como *M*/*N*, donde *M* es el número de elementos en el arreglo y *N* es su tamaño. Según Lipschutz, la probabilidad de llevar a cabo una búsqueda con éxito (*S*) y otra sin éxito (*Z*), quedan determinadas por las siguientes fórmulas:

Fórmulas 9.7

***a*) Búsqueda con éxito**

$$S(\lambda) = \frac{(1 + 1/(1-\lambda))}{2}$$

***b*) Búsqueda sin éxito**

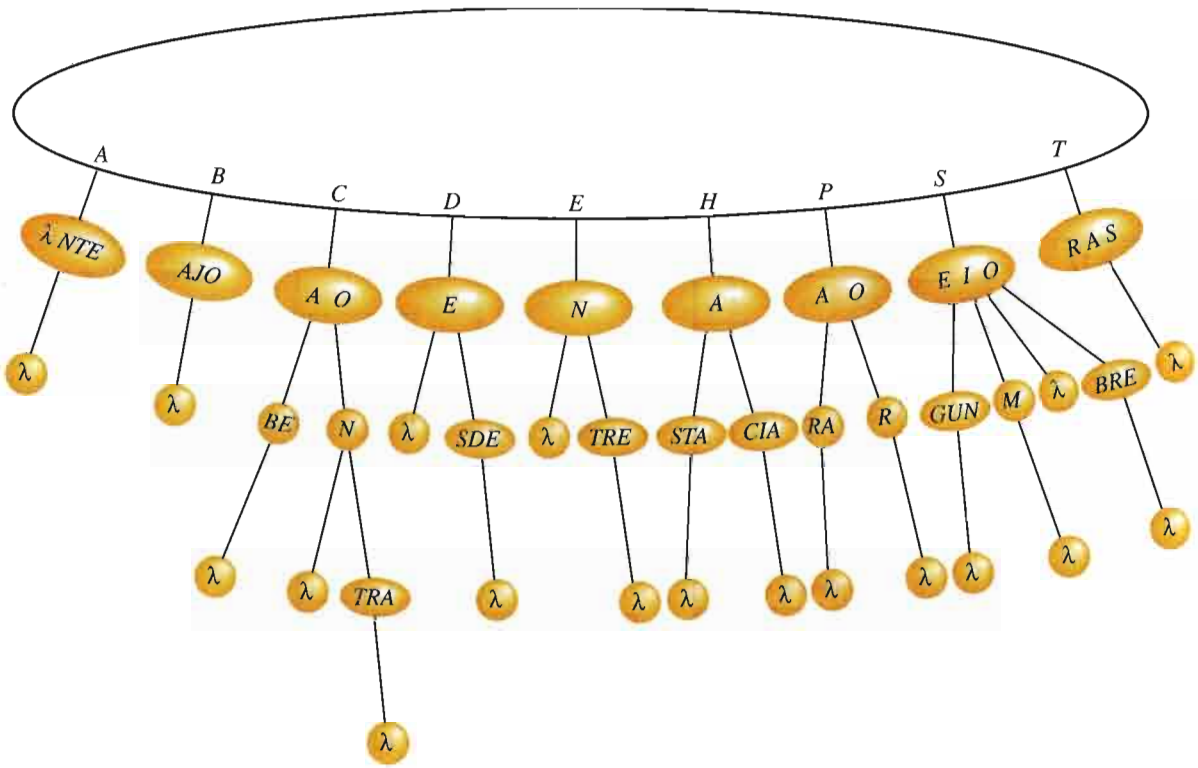
$$Z(\lambda) = \frac{(1 + 1/(1-\lambda)^2)}{2}$$

Cabe aclarar que estas fórmulas son válidas solamente en caso de funciones *hash* con el método lineal de solución de colisiones.

9.2.12 Árboles de búsqueda

En el capítulo 6 se presentaron los árboles como una estructura poderosa y eficiente para almacenar y recuperar información. Debido al dinamismo que caracteriza a los árboles, el beneficio de utilizarlos es mayor cuanto más variable sea el número de datos a tratar.

FIGURA 9.13
Representación de tries.



En esta sección sólo se hablará de la estructura **trie**, que es una variante de la estructura tipo árbol.

Un trie es una estructura similar a un árbol con *N* raíces, con la particularidad de que cada nodo del árbol puede ser nuevamente un trie. En la figura 9.13 se presenta un diagrama correspondiente a un trie que contiene las proposiciones del castellano.

Un trie puede representar una estructura sumamente útil para búsqueda. Las raíces del árbol tienen como objetivo dirigir el camino de búsqueda hacia la meta. La profundidad de una estructura de este tipo depende de la discriminación en la clave de búsqueda que realice el usuario. En la figura 9.13 se puede observar un trie cuya profundidad es variable para cada raíz. De esta forma se localiza la información buscada directamente en el nodo terminal, sin tener que realizar búsqueda secuencial. En la figura 9.14 el lector puede observar un trie con profundidad tres; la discriminación en la clave de búsqueda es igual a 2.

Con el propósito de instrumentar esta estructura en un lenguaje de alto nivel, podemos representar un trie como un bosque. Posteriormente, aplicando las reglas necesarias —analizadas en el capítulo 6—, se debe convertir esta estructura en árbol binario. En la figura 9.15 se muestra el bosque que representa al trie de la figura 9.13.

Finalmente, en la figura 9.16 se muestra al árbol binario que representa al bosque de la figura 9.15.

FIGURA 9.14

Representación de un trie con discriminación 2.

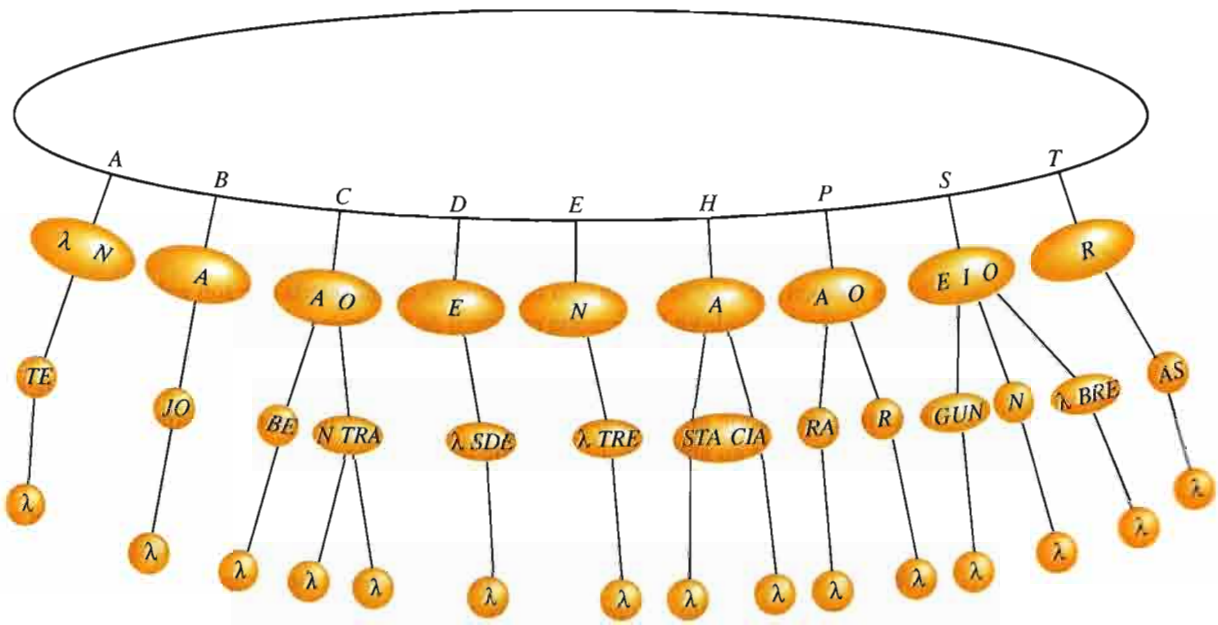
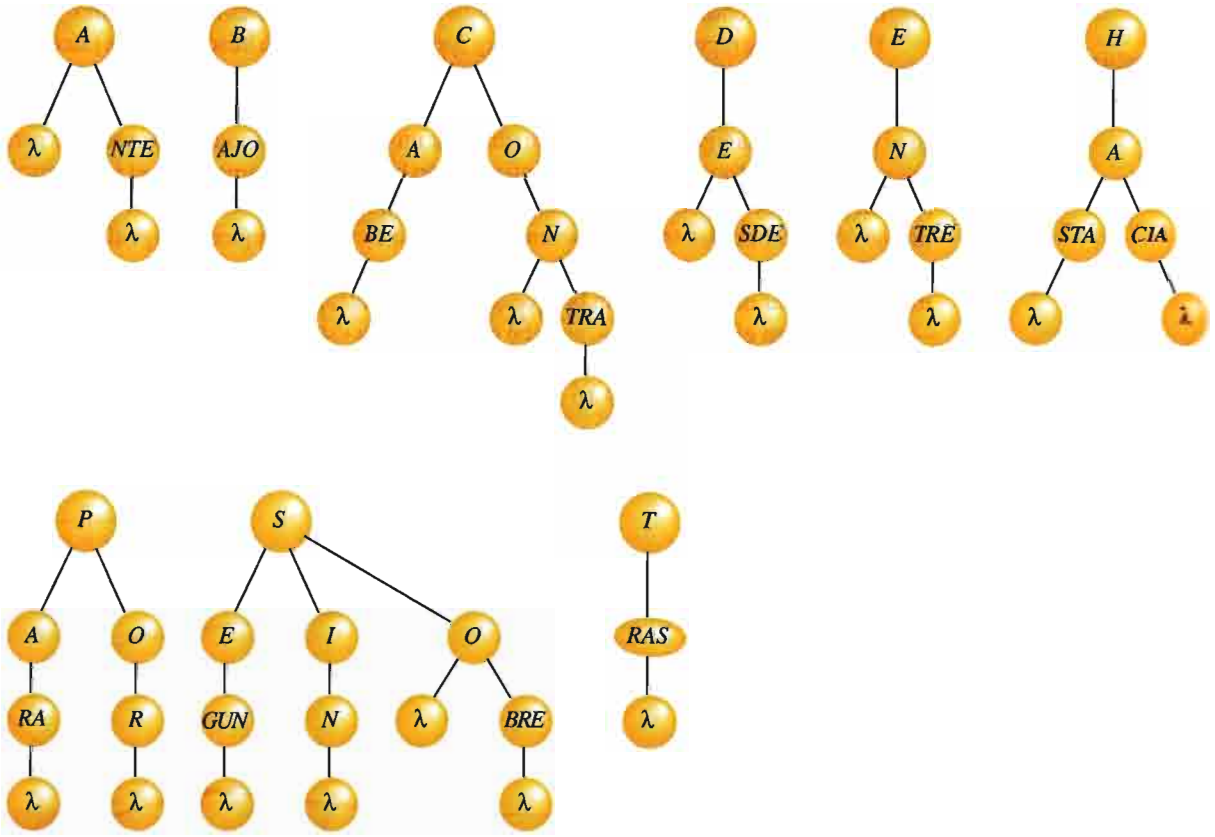


FIGURA 9.15
Representación del trie de la figura 9.13 como bosque.



9.3 BÚSQUEDA EXTERNA

En la sección anterior se estudiaron las técnicas de búsqueda que son aplicables cuando la información reside en la memoria principal de la computadora. En particular, se analizó la operación de búsqueda en estructuras estáticas —arreglos— y dinámicas —listas y árboles— de información. Sin embargo, existen casos en los cuales no se puede manejar toda la información en memoria principal, sino que es necesario trabajar con información almacenada en archivos. Este tipo de búsqueda se denomina **búsqueda externa**.

Los archivos se usan normalmente cuando el volumen de datos es significativo, o cuando la aplicación exige la permanencia de los datos, aun después de que ésta se termine de ejecutar. Como los archivos se encuentran almacenados en dispositivos periféricos —cintas, discos, etc.—, las operaciones de escritura y lectura de datos tienen un alto costo en cuanto a tiempo, por los accesos a estos periféricos. Para disminuir el

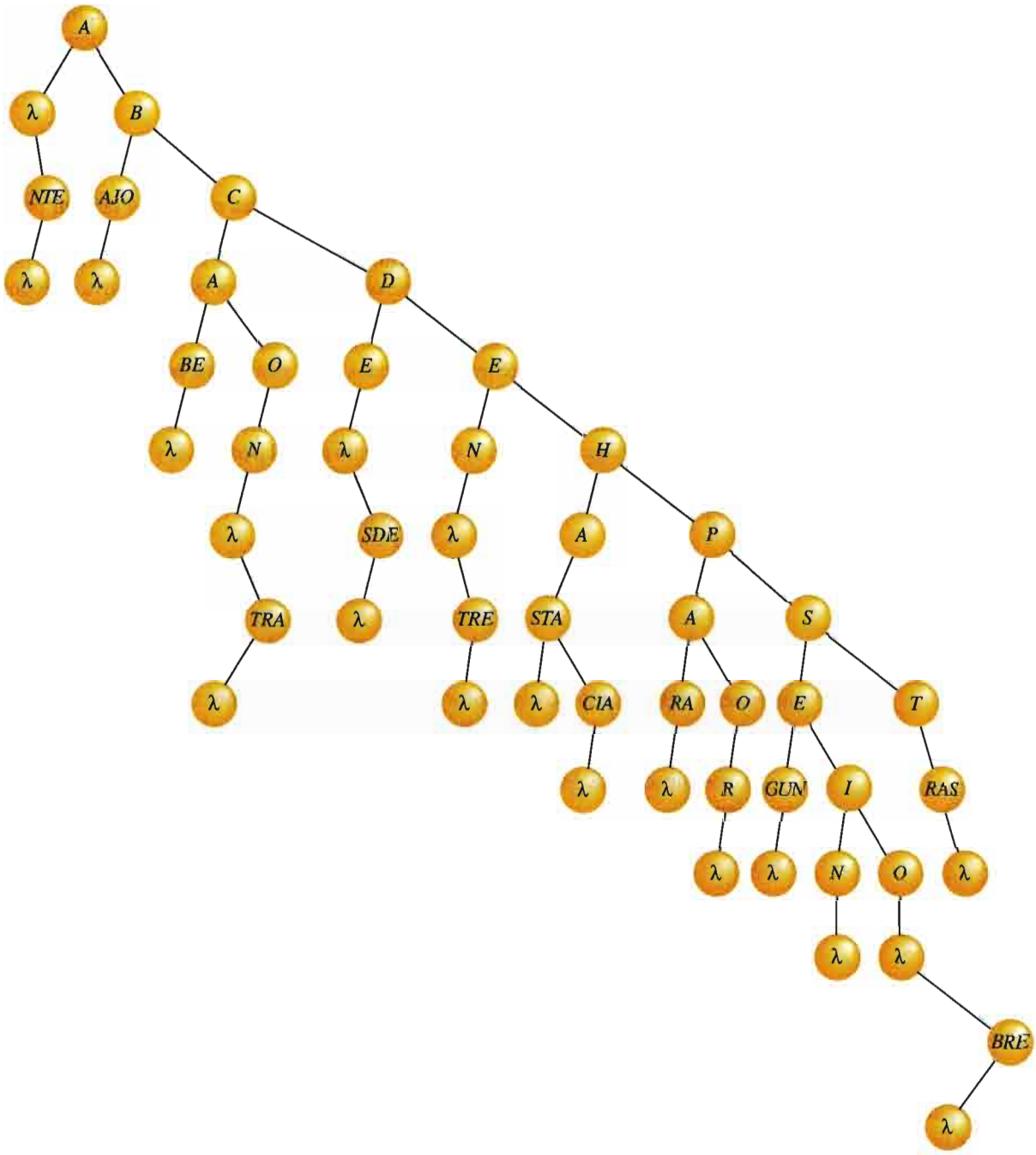


FIGURA 9.16
Representación del bosque de la figura 9.15 como árbol binario.

tiempo de acceso es muy importante optimizar las operaciones de búsqueda, inserción y eliminación en archivos. Una forma de hacerlo es trabajar con archivos ordenados. A continuación se describen algunos de los métodos más utilizados en búsqueda externa.

9.3.1 Búsqueda en archivos secuenciales

Los **archivos secuenciales** son aquellos cuyos componentes o registros ocupan posiciones relativas consecutivas. Todo componente o registro de un archivo tiene generalmente un campo que lo identifica, llamado **campo clave**. Éste se encuentra formado por un conjunto de caracteres o dígitos. Además, ocupa la misma posición relativa en todos los registros de un mismo archivo. Algunos ejemplos de campos clave son el número de cliente —archivo de clientes—, el número de contribuyente —archivo de hacienda—, la matrícula de un alumno —archivo de alumnos—, el número de empleado —archivo de empleados—, etc. Puede suceder que la clave de un registro esté formada por más de un campo. Por ejemplo, en un sistema de inventarios cada pieza se podría identificar por un campo que haga referencia al departamento al cual pertenece, y otro campo para la pieza en sí.

Enseguida se describen algunos métodos de búsqueda en archivos secuenciales.

9.3.2 Búsqueda secuencial

El método de **búsqueda secuencial** consiste en recorrer el archivo comparando la clave buscada con la clave del registro en curso. El recorrido lineal del archivo termina cuando se encuentra el elemento, o cuando se alcanza el final del archivo. Se pueden presentar algunas variantes dentro de este método, dependiendo sobre todo de si el archivo está ordenado o desordenado.

A continuación se detalla el algoritmo de búsqueda lineal en un archivo secuencial desordenado.

Algoritmo 9.14 Archivo_secuencial_desordenado

Archivo_secuencial_desordenado (*FA*, *K*)

{Este algoritmo busca secuencialmente en un archivo desordenado *FA*, un registro con clave *K*}
{BAN es una variable de tipo booleano. *R* es una variable de tipo registro. CLAVE es un campo del registro}

1. Abrir el archivo *FA* para lectura
2. Hacer BAN ← FALSO
3. Mientras ((no sea el fin de archivo de *FA*) y (BAN = FALSO)) *Repetir*
   Leer *R* de *FA*
   3.1 *Si* (*R*.CLAVE = *K*) *entonces*
   Escribir "La información se encuentra en el archivo"

```
        Hacer BAN ← VERDADERO
   3.2 {Fin del condicional del paso 3.1}
4. {Fin del ciclo del paso 3}
5. Si (BAN = FALSO) entonces
        Escribir "La información no se encuentra en el archivo"
6. {Fin del condicional del paso 5}
```

Este algoritmo es similar al 9.1. En general, tiene las mismas características que el método secuencial en arreglos desordenados.

El algoritmo de búsqueda en archivos ordenados se estudiará considerando, en particular, archivos ordenados en forma creciente.

Algoritmo 9.15 Archivo_secuencial_ordenado

```
Archivo_secuencial_ordenado (FA, K)
{Este algoritmo busca secuencialmente en un archivo FA ordenado en forma creciente, un
registro con clave K}
{BAN es una variable de tipo booleano. R es una variable de tipo registro. CLAVE es un
campo del registro}
1. Abrir el archivo FA para lectura
2. Hacer BAN ← FALSO
3. Mientras ((no sea el fin de archivo de FA) y (BAN = FALSO)) Repetir
        Leer R de FA
   3.1 Si (R.CLAVE ≥ K) entonces
            Hacer BAN ← VERDADERO
   3.2 {Fin del condicional del paso 3.1}
4. {Fin del ciclo del paso 3}
5. Si (R.CLAVE = K)
   entonces
        Escribir "La información se encuentra en el archivo"
   si no
        Escribir "La información no se encuentra en el archivo"
6. {Fin del condicional del paso 5}
```

La diferencia entre este algoritmo y el anterior consiste en que la búsqueda también se detiene cuando la clave de R es mayor que K. Esto último se debe a que si el archivo está ordenado, ya no se encontrará el registro con clave K entre los registros aún no visitados.

9.3.3 Búsqueda secuencial mediante bloques

La **búsqueda secuencial mediante bloques** consiste en tomar bloques de registros en vez de registros aislados. Un **bloque** es un conjunto de registros. Su tamaño es arbitrario y depende del número de elementos del archivo. Generalmente se define el tamaño del bloque igual a $\sqrt{N}$, donde $\sqrt{N}$ es el número de registros del archivo —la demostración de por qué es $\sqrt{N}$ se presenta más adelante—. El archivo debe estar ordenado. La búsqueda se realiza al comparar la clave en cuestión con el último registro de cada bloque. Si la clave resulta menor, entonces se busca en forma secuencial a través de los registros salteados en el bloque. En caso contrario se continúa con el siguiente bloque. En promedio, el número de comparaciones requeridas para encontrar un valor dado será igual a $\sqrt{N}$.

A continuación se presenta un algoritmo de búsqueda secuencial usando bloques.

Algoritmo 9.16 Archivo_secuencial_bloques

Archivo_secuencial_bloques (*FA*, *N*, *K*)

{Este algoritmo busca secuencialmente en un archivo ordenado *FA* de *N* elementos, un registro con clave *K*}
{*I* y *TB* son variables de tipo entero. BAN es una variable de tipo booleano}

1. Abrir el archivo *FA* para lectura
2. Hacer BAN ← FALSO, *I* ← 1 y *TB* ← Parte Entera (sqrt (*N*)) {Calcula el tamaño del bloque como la raíz cuadrada de *N*}
3. Mientras ((*TB* * *I* ≤ *N*) y (BAN = FALSO)) *Repetir*
   Leer *R* de *FA* en la posición *TB* * *I*
   3.1 *Si* (*R*.CLAVE ≥ *K*)
   entonces
   Hacer BAN ← VERDADERO
   si no
   Hacer *I* ← *I* + 1
   3.2 {Fin del condicional del paso 3.1}
4. {Fin del ciclo del paso 3}
5. *Si* (BAN = VERDADERO)
   entonces
   5.1 *Si* (*R*.CLAVE = *K*)
   entonces
   Escribir "La información se encuentra en el archivo"
   si no
   Realizar búsqueda secuencial en los registros salteados: del registro (*TB* * (*I* – 1) + 1) al registro (*TB* * *I* – 1)
   Reposicionar el puntero del archivo, y aplicar el algoritmo 9.15 para ejecutar la búsqueda elemento por elemento
   5.2 {Fin del condicional del paso 5.1}
   si no {Si *TB* no es múltiplo de *N*, quedaron elementos sin revisar}
   Realizar búsqueda secuencial en los registros comprendidos entre (*TB* * (*I* – 1) + 1) y *N*
6. {Fin del condicional del paso 5}

En este algoritmo se lee el último registro de cada bloque, y de la comparación del elemento buscado con él se decide cómo continuar con la búsqueda. El siguiente ejemplo ilustra mejor el funcionamiento de este algoritmo.

Ejemplo 9.12

Sea *FA* un archivo ordenado de 20 registros. Los registros ocupan posiciones consecutivas con direcciones relativas del 1 al 20. Las claves de los registros almacenados en *FA* son:

204, 311, 409, 415, 439, 450, 502, 507, 600, 623, 679, 680, 691, 692, 695, 698, 730, 850, 870, 889.

Dado que se conoce *N*, se calcula el tamaño del bloque de la siguiente manera:

$$TB = \sqrt{20} \cong 4$$

204, 311, 409, **415**, 439, 450, 502, **507**, 600, 623, 679, **680**, 691, 692, 695, **698**, 730, 850, 870, **889**.

La tabla 9.11 presenta el seguimiento del algoritmo 9.16 para $K = 623$.

En la columna *Registro leído* aparece el último registro del bloque, pasos 1, 2 y 3. En el paso 3, cuando se cumple la condición de que *R*.CLAVE ≥ *K*, entonces se comienza la búsqueda secuencial a partir del elemento ($TB * (I - 1) + 1$) —elemento 9—, en este caso el 600, hasta que se encuentra el valor deseado —éxito— o hasta el elemento ($TB * I - 1$) —elemento 11—. Observe que en el paso 5 se encuentra el registro buscado.

9.3.4 Búsqueda secuencial con índices

El método de **búsqueda secuencial con índices** trabaja con bloques y con archivos de índices. En el archivo de índices se almacenan las claves que hacen referencia a cada bloque y la dirección de los bloques en el archivo. La búsqueda de un elemento comienza recorriendo el archivo de índices, comparando las claves allí almacenadas con la clave del elemento en cuestión. Una vez que se determina el bloque en el cual se puede encontrar el registro buscado, se continúa la búsqueda ahora recorriendo secuencialmente dicho bloque.

TABLA 9.11
Búsqueda secuencial con bloque

Paso	*I*	Registro leído	Compara	Bandera
1	1	415	415 ≥ 623 ?	*F*
2	2	507	507 ≥ 623 ?	*F*
3	3	680	680 ≥ 623 ?	*V*
4		600	600 = 623 ?	
5		623	623 = 623 ?	

La desventaja de este método es que requiere más espacio de memoria, ya que se trabaja con dos archivos: el principal, en el cual se almacenan los registros, y el de índices. Una forma de acelerar el proceso de búsqueda consiste en mantener en memoria principal el archivo de índices.

En la figura 9.17 se presenta un esquema de un archivo con su correspondiente archivo de índices.

El archivo de índices se recorre secuencialmente hasta encontrar la clave que sea mayor o igual a la clave buscada. Cuando esto último suceda, se tomará la dirección del bloque apuntado por dicha clave y se aplicará búsqueda secuencial (algoritmo 9.15) en dicho bloque.

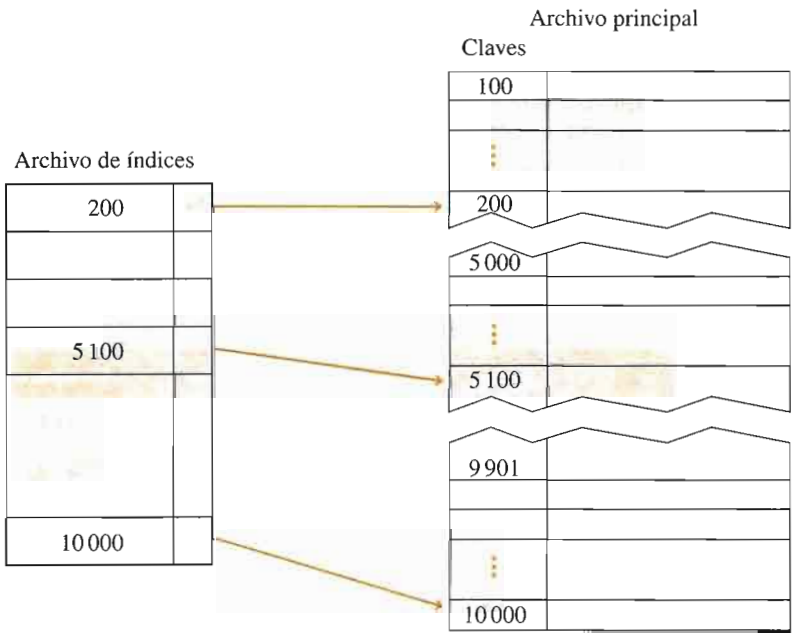
Determinación del tamaño del bloque

El tamaño del bloque se debe elegir de tal forma que permita reducir el número de comparaciones. Sea N el número de registros en el archivo y TB el tamaño del bloque. La probabilidad de encontrar un registro en un bloque es igual para todos los bloques; por lo tanto, el número medio de bloques examinados será:

$$\sum_{i=1}^{N/TB}\left(i * \frac{1}{(N/TB)}\right) = \frac{N/TB+1}{2} \qquad (1)$$

Donde $1/(N/TB)$ representa la probabilidad de encontrar un registro en un bloque. Considere, además, que todos los registros tienen la misma probabilidad de ser el buscado; por lo tanto, el número medio de registros examinados será:

FIGURA 9.17
Búsqueda secuencial con índices.



$$\sum_{i=0}^{TB-1}\left(i * \frac{1}{TB}\right) = \frac{TB-1}{2} \tag{2}$$

Donde $1/TB$ es la probabilidad de que el registro examinado sea el buscado.

Se suman las expresiones 1 y 2 para obtener el número total medio de comparaciones (TC) que se deben hacer para encontrar un elemento en el archivo.

$$TC = \frac{(N/TB)+1}{2} + \frac{TB-1}{2}$$

Operando se obtiene:

$$TC = \frac{N}{2*TB} + \frac{TB}{2} \tag{3}$$

Al minimizar TC se podrá determinar cuál es el tamaño adecuado para definir los bloques; es decir, el problema se reduce a encontrar un valor tal para TB que minimice el valor de TC.

$$\frac{d(TC)}{d(TB)} = \frac{-N}{2(TB)^2} + \frac{1}{2} \tag{4}$$

Se iguala a cero la expresión 4 y se hacen las operaciones:

$$\frac{-N}{2(TB)^2} + \frac{1}{2} = 0 \qquad \frac{N}{2(TB)^2} = \frac{1}{2} \tag{5}$$

De la expresión 5 se puede afirmar que el valor de TB que minimiza a TC es:

$$TB = \sqrt{N} \qquad \textbf{Fórmula 9.8}$$

Los archivos de índices, por otra parte, se pueden definir a distintos niveles; es decir, se pueden definir índices de índices. Si bien este tipo de organización optimiza el tiempo de búsqueda, tiene el inconveniente de que ocupa mucho espacio de almacenamiento.

9.3.5 Búsqueda binaria

El principio que rige el método de **búsqueda binaria** en la búsqueda externa es el mismo que se explicó en búsqueda binaria interna, sección 9.2.2 de este capítulo. El archivo debe estar ordenado y se debe conocer su número de elementos (N) para aplicar este método. El lector puede desarrollarlo fácilmente, ya que conoce el método de búsqueda binaria en memoria principal —interna—.

Cabe destacar que un gran inconveniente de la búsqueda binaria externa es que requiere accesos a diferentes posiciones del dispositivo periférico en el cual está almacenado el archivo; ello produce un alto costo en tiempo de acceso, que hace muy impráctica esta búsqueda.

9.3.6 Búsqueda por transformación de claves (*hash*)

El método de búsqueda externa por **transformación de claves** tiene básicamente las mismas características que el presentado en la sección 9.2.3. Los archivos normalmente se encuentran organizados en áreas llamadas **cubetas**. Éstas se encuentran formadas por cero, uno o más bloques de registros. Por lo tanto, la función *hash*, aplicada a una clave, dará como resultado un valor que hace referencia a una cubeta en la cual se puede encontrar el registro buscado.

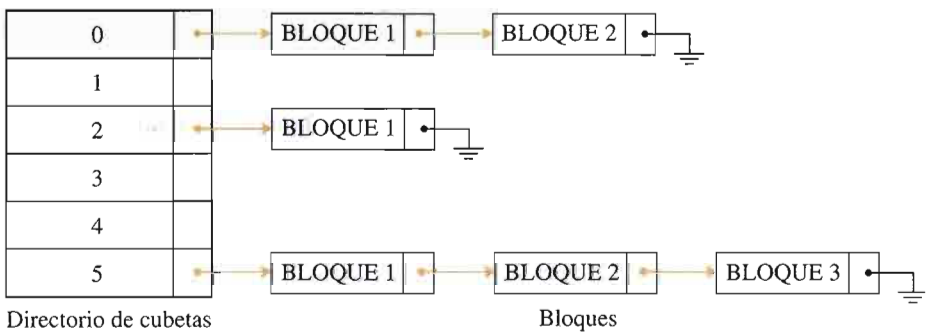
Tal como se mencionó en búsqueda interna, la elección de una adecuada función *hash* y de un método para resolver colisiones es fundamental para lograr mayor eficiencia en la búsqueda.

Antes de presentar algunas funciones *hash* se hará un comentario sobre las colisiones. Los bloques contienen un número fijo de registros. Con respecto a las cubetas, no se establece un límite en cuanto al número de bloques que pueden almacenar. Esta característica permite solucionar, al menos parcialmente, el problema de las colisiones. Sin embargo, si el tamaño de las cubetas crece considerablemente, se perderán las ventajas propias de este método. Es decir, si el número de bloques que se deben recorrer en una cubeta es grande, el tiempo necesario para ello será significativo; por lo tanto, ya no se contará con la ventaja del acceso directo que caracteriza al método por transformación de claves. En la figura 9.18 se presenta una estructura de archivo organizado en cubetas, las que a su vez están formadas por bloques.

Como se muestra en la figura 9.17, cada cubeta puede tener un apuntador a un bloque. Si una cubeta tiene dos o más bloques se establecen ligas entre ellos. Dada la clave de un registro buscado, se aplicará una función *hash*, la cual dará como resultado un número de cubeta. Una vez localizada ésta, habrá que recorrer sus bloques hasta encontrar el registro, o llegar a un bloque con puntero nulo, lo cual indicará que no existen otros bloques.

Es importante elegir una función *hash* que distribuya las claves en forma homogénea a través de las cubetas, de manera que se evite la concentración de numerosas claves

FIGURA 9.18
Archivo organizado con cubetas de bloques.



en una cubeta mientras otras permanecen vacías. A continuación se presentan algunas de las funciones *hash* más comunes.

Funciones *hash*

Una función *hash* se puede definir como una transformación de clave a una dirección. Al aplicar una función *hash* a una clave se obtiene el número de cubeta en la cual se puede encontrar el registro con dicha clave.

La función debe transformar las claves para que la dirección resultante sea un número comprendido entre los posibles valores de las cubetas. Por ejemplo, si se tienen 10 000 cubetas numeradas de 0 a 9 999, las direcciones producidas por la función deben ser valores comprendidos entre 0 y 9 999. Si las claves fueran alfabéticas o alfanuméricas, primero deberán convertirse en numéricas, tratando de no perder información, para luego ser transformadas en una dirección. Es importante que la función distribuya homogéneamente las claves entre los números de cubetas disponibles.

Las funciones módulo, cuadrado, plegamiento y truncamiento presentadas anteriormente para búsqueda interna son válidas también para búsqueda externa. Otra función que se puede utilizar para el cálculo de direcciones es la de conversión de bases, aunque no proporciona mayor homogeneidad en la distribución. De todas, la función módulo es, sin embargo, la que ofrece mayor uniformidad.

Conversiones de bases

La **conversión de bases** consiste en modificar de manera arbitraria la base de la clave obteniendo un número que corresponda a una cubeta. Si el número de dígitos del valor resultante excede el orden de las direcciones, entonces se suprimirán los dígitos más significativos.

Ejemplo 9.13

Supongamos que se tienen 100 cubetas, cada una de ellas referenciada por un número entero comprendido entre 1 y 100. Sea $K = 7\ 259$ la clave del registro que se busca. Se elige el 9 como base a la cual se convierte la clave.

$$H(7\ 259) = \text{dígmensig}\ (7 * 9^3 + 2 * 9^2 + 5 * 9^1 + 9 * 9^0)$$
$$H(7\ 259) = \text{dígmensig}(5\ 319) = 19$$

Se toma entonces como dirección el 19 y los dígitos más significativos, 5 y 3, se desprecian.

9.3.7 Solución de colisiones

Como se mencionó anteriormente cuando se trató búsqueda interna, uno de los aspectos que siempre se deben de considerar en el método por transformación de claves es la solución de colisiones. Cuando dos o más elementos con distintas claves tienen una misma dirección, se origina una colisión.

Para evitar las colisiones se debe elegir un tamaño adecuado de cubetas y de bloques. Con respecto a las cubetas, si se definen muy pequeñas el número de colisiones

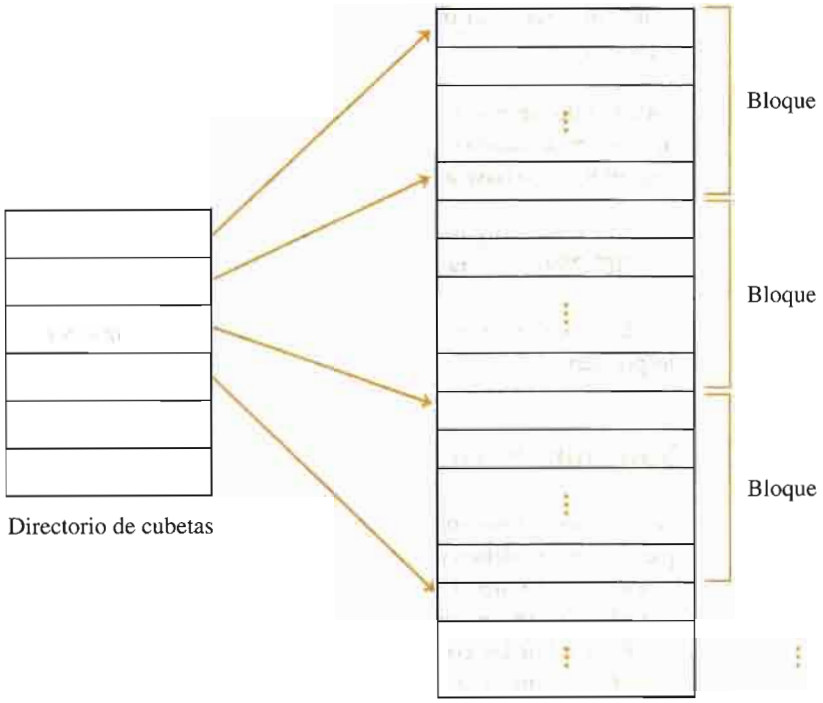
aumenta, mientras que si se definen muy grandes se pierde eficiencia en cuanto a espacio de almacenamiento. Además, si se necesitara copiar una cubeta en memoria principal y ésta fuera muy grande, ocasionaría problemas por falta de espacio. Otro inconveniente que se presenta en el caso de cubetas muy grandes es que se requiere mucho tiempo para recorrerlas.

Con respecto al tamaño de los bloques, es importante considerar la capacidad de éstos para almacenar registros. Un bloque puede almacenar uno, dos o más registros. Normalmente los tamaños de las cubetas y los bloques dependen de las capacidades del equipo con el que se esté trabajando.

Cabe destacar que utilizando una estructura como la de la figura 9.18 no se tendría problemas de colisiones, debido principalmente a que por más que la cubeta esté ocupada, es posible seguir enlazando tantos bloques como fueran necesarios. Este esquema de solución se corresponde con el presentado en búsqueda interna, bajo el nombre de encadenamiento. Sin embargo, no siempre es posible definir una estructura de este tipo. Considere, por ejemplo, un archivo organizado en cubetas como el que se muestra en la figura 9.19.

En este archivo cada cubeta tiene un bloque y, por lo tanto, una capacidad máxima determinada por el tamaño del bloque asociado con ella. Una vez que se satura la capacidad de la cubeta, cualquier registro asignado a ella producirá una colisión. A continuación se analizarán dos maneras diferentes de enfrentar esta situación.

FIGURA 9.19
Solución de colisiones.

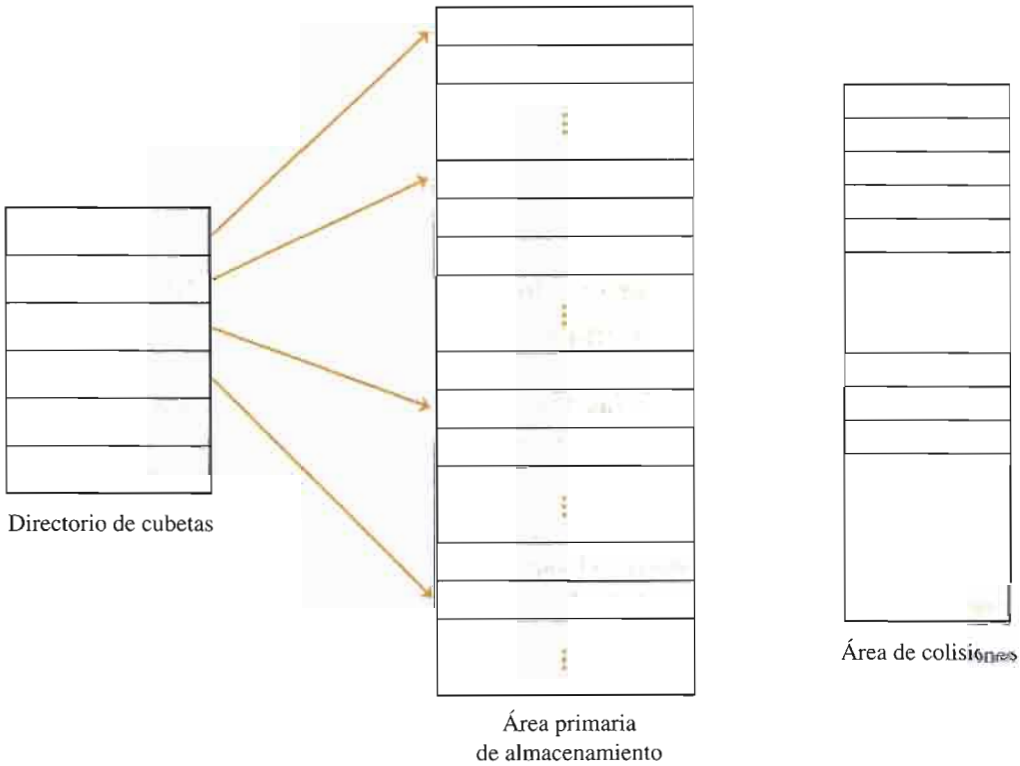


Uso de áreas independientes para colisiones

El **uso de áreas independientes para colisiones** consiste en definir áreas separadas —secundarias— de las áreas primarias de almacenamiento, en las que se almacenarán todos los registros que hayan colisionado. El área de colisiones puede estar organizada de diferentes maneras. Una alternativa consiste en tener el área común a todas las cubetas. En consecuencia, si se produce una colisión habrá que buscar a lo largo del área secundaria hasta encontrar el elemento deseado, según la figura 9.20.

Otra forma de organizar el área de colisiones consiste en dividirla en bloques, asociando cada uno de ellos a uno del área primaria. Esta alternativa optimiza el tiempo de búsqueda en el área de colisiones, pero tiene el inconveniente de que estos bloques podrían, a su vez, saturarse, ocasionando nuevamente colisiones. El esquema correspondiente a esta estructura se muestra en la figura 9.21.

FIGURA 9.20
Solución de colisiones mediante un área común de colisiones.



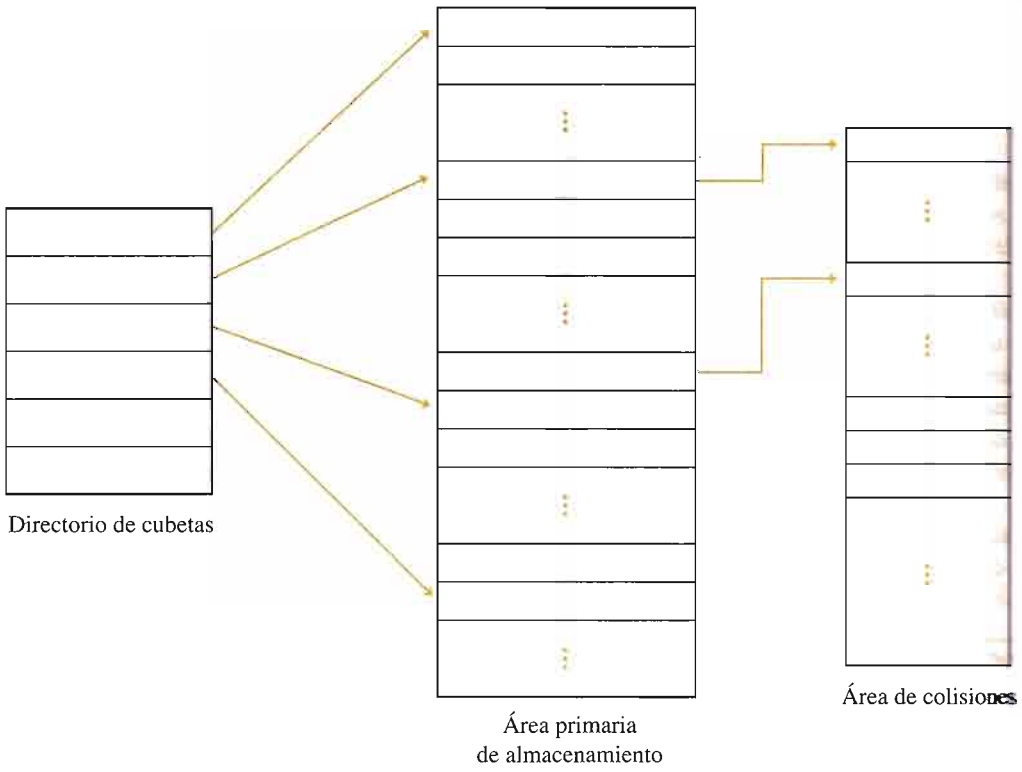


FIGURA 9.21
Solución de colisiones mediante un área de colisiones organizada en bloques.

Uso de áreas de colisiones entre los bloques de almacenamiento primario

El **uso de áreas de colisiones entre los bloques de almacenamiento primario** consiste en definir áreas de colisiones entre los bloques de almacenamiento primario. Este método es similar al presentado en búsqueda interna bajo el nombre de reasignación. Una vez detectada una colisión en un bloque se debe buscar en el área de colisiones inmediata a dicho bloque. Si el elemento no se encuentra y el área de colisiones está llena, se continuará la búsqueda a través de las otras áreas de colisiones. El proceso termina cuando el elemento se encuentra o bien cuando existen espacios vacíos en un bloque —el elemento buscado no se encuentra en el archivo—. El esquema correspondiente a este esquema se muestra en la figura 9.22.

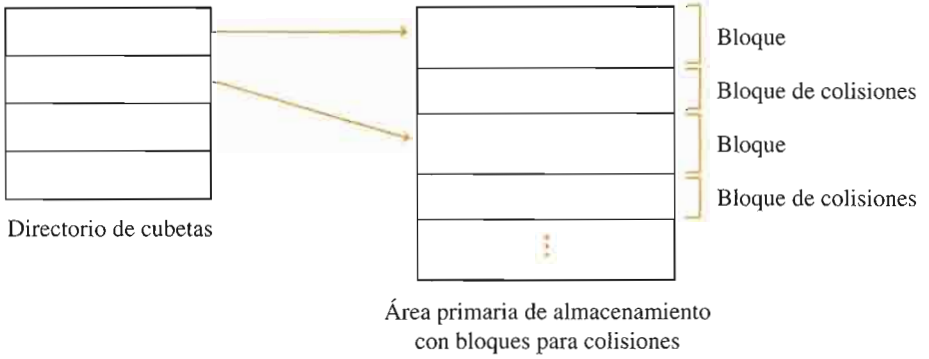


FIGURA 9.22
Solución de colisiones mediante bloques para colisiones entre los bloques primarios.

9.3.8 *Hashing* dinámico: búsqueda dinámica por transformación de claves

La principal característica del ***hashing* dinámico** es su dinamismo para variar el número de cubetas en función de su densidad de ocupación. Se comienza a trabajar con un número determinado de cubetas, y a medida que éstas se van llenando se asignan nuevas cubetas al archivo. Existen básicamente dos formas de trabajar con el *hashing* dinámico:

- Por medio de expansiones totales
- Por medio de expansiones parciales

9.3.9 Método de las expansiones totales

El método de **expansiones totales** es probablemente el más utilizado. Consiste en duplicar el número de cubetas en la medida en que éstas superan la densidad de ocupación previamente establecida. Así, por ejemplo, si el número inicial de cubetas es N y se hace una expansión total, el valor resultante —nuevo número de cubetas— será $2N$. Si se hace una segunda expansión total, se tendrá $4N$, y así sucesivamente.

El dinamismo de este método también se da en sentido contrario; es decir, que a medida que la densidad de ocupación de las cubetas disminuye, se reduce el número de éstas. Así, se gana flexibilidad en cuanto a que se pueden incrementar los espacios de almacenamiento, pero también se pueden reducir si la demanda de espacio así lo indica.

Ejemplo 9.14

Supongamos que se tiene un archivo organizado en dos cubetas ($N = 2$), y se ha fijado una densidad de ocupación de 80%. La **densidad de ocupación** se calcula como el cociente entre el número de registros ocupados y el de registros disponibles. Cada cubeta tiene dos registros, y la función *hash* que transforma claves en direcciones se define de la siguiente manera:

$$H\,(\text{clave}) = \text{clave MOD } \textit{Número de cubetas}$$

FIGURA 9.23
Hash dinámico ($N = 2$): expansión total.

Cubetas 0 1

Porcentaje de ocupación para expansión: 75%

42	15
24	

Clave	H(Clave)
42	0
24	1
15	1

Los valores 42, 24, 15 y 53 son las claves de los registros que se desea almacenar. Inicialmente el archivo está vacío. En la figura 9.23 se presenta un esquema de cómo quedan las cubetas, después de insertar las tres primeras claves.

Cuando se quiere insertar la clave 53, se supera la densidad de ocupación establecida, ya que se alcanzaría 100% de llenado. Por lo tanto, se deben expandir y reasignar los registros considerando ahora que el número de cubetas es igual a $2*N$, figura 9.24.

Supongamos ahora que se desea incorporar los registros con claves 21, 12, 14, 18, 49, 128, 22, 23 y 67 en este orden. El resultado, después de insertar las dos primeras claves, se puede observar en la figura 9.25.

Cuando se inserta el registro con clave 14, la densidad de ocupación supera el 80% fijado. Se vuelven, entonces, a expandir y a reasignar los registros almacenados (figura 9.26*a*), y luego se continúa con la inserción del resto de los elementos. La figura 9.26*b* presenta el estado de las cubetas luego de realizar todas las inserciones, excepto la última.

Cuando se inserta la última clave, 67, se supera nuevamente la densidad de ocupación y hay que volver a expandir las cubetas. Por lo tanto, ahora N será igual a 16 (figura 9.27).

Es importante señalar que en este método también se pueden producir colisiones, las cuales podrían tratarse según alguno de los esquemas propuestos anteriormente. Por ejemplo, si en el caso anterior (figura 9.26) luego de insertar los registros con claves 24 y 128 se tratara de agregar el registro con clave 192, se produciría una colisión, ya que la cubeta 0 está llena.

Ejemplo 9.15

Dado un archivo organizado en dos cubetas ($N = 2$), donde cada una de ellas tiene tres registros, se quiere almacenar las siguientes claves:

115, 96, 48, 79, 35, 26, 57, 81, 70, 64, 107, 45, 62, 98, 33, 28 y 38.

Se ha establecido una densidad de ocupación mayor a 82% para expansión y menor a 125% para reducción. Es importante remarcar que el porcentaje de ocupación, para el

FIGURA 9.24
Hash dinámico ($N = 4$): expansión total.

Cubetas 0 1 2 3

Porcentaje de ocupación para expansión: 50%

24	53	42	15

Clave	H(Clave)
42	2
24	0
15	3
53	1

FIGURA 9.25
Hash dinámico ($N = 4$): expansión total. *a)* Luego de insertar 21. *b)* Luego de insertar 12.

Cubetas 0 1 2 3

24	53	42	15
	21		

Porcentaje de ocupación para expansión: 62.50%

a)

Clave	*H*(Clave)
21	1
12	2

Cubetas 0 1 2 3

24	53	42	15
12	21		

Porcentaje de ocupación para expansión: 75%

b)

caso de reducción, se calcula como el cociente entre el número de registros ocupados y el número de cubetas.

A continuación se presenta la función *hash* que se utiliza:

$$H \text{ (clave)} = \text{clave MOD } \textit{Número de cubetas}$$

Las claves se almacenan en el orden en que se dan. La representación final se puede observar en la figura 9.28.

En el siguiente ejemplo se aclara el concepto de reducción del número de cubetas en el método dinámico por transformación de claves, con expansiones totales.

FIGURA 9.26
Hash dinámico ($N = 4$): expansión total. *a)* Luego de insertar 21. *b)* Luego de insertar 12.

Cubetas 0 1 2 3 4 5 6 7

24		42		12	53	14	15
					21		

Porcentaje de ocupación para expansión: 43.75%

a)

Cubetas 0 1 2 3 4 5 6 7

24	49	42		12	53	14	15
128		18			21	22	23

Porcentaje de ocupación para expansión: 68.75%

b)

Clave	*H*(Clave)
42	2
24	0
15	7
53	5
21	5
12	4
14	6
18	2
49	1
128	0
22	6
23	7

Cubetas	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	128	49	18	67		53	22	23	24		42		12		14	15
						21										

Porcentaje de ocupación para reducción: 81.25%

Porcentaje de ocupación para expansión: 40.62%

Clave	*H*(Clave)
42	10
24	8
15	15
53	5
21	5
12	12
14	14
18	2
49	1
128	0
22	6
23	7
67	3

FIGURA 9.27
Hash dinámico ($N = 16$): expansión total.

Ejemplo 9.16

Supongamos que se tiene el archivo en el estado que muestra la figura 9.27. Se desean eliminar ahora los registros con claves:

53, 18, 128, 23, 14, 49 y 22.

Al eliminar el registro con clave 53, la densidad de ocupación disminuye de tal manera que permite reducir el número de cubetas ($N/2$). Luego de la reducción y de la reasignación de registros, las cubetas quedan como se muestra en la figura 9.29.

Una vez eliminados los otros registros, la densidad de ocupación permite reducir nuevamente el número de cubetas. En la figura 9.30 se presenta su estado luego de la reducción de N y de la reasignación de los registros.

Ejemplo 9.17

Dado el archivo de la figura 9.28 y las especificaciones dadas en el ejemplo 9.15, elimine las siguientes claves:

Cubetas	0	1	2	3	4	5	6	7
Porcentaje de ocupación para expansión: 70.83%	96	57	26	115	28	45	70	79
	48	81	98	35			62	
Porcentaje de ocupación para reducción: 212.50%	64	33		107			38	

FIGURA 9.28
Hash dinámico ($N = 8$): expansión total.

Clave	H(Clave)
42	2
24	0
15	7
21	5
12	4
14	6
18	2
49	1
128	0
22	6
23	7
67	3

Cubetas

0	1	2	3	4	5	6	7
24	49	42	67	12	21	14	15
128		18				22	23

Porcentaje de ocupación para expansión: 150%

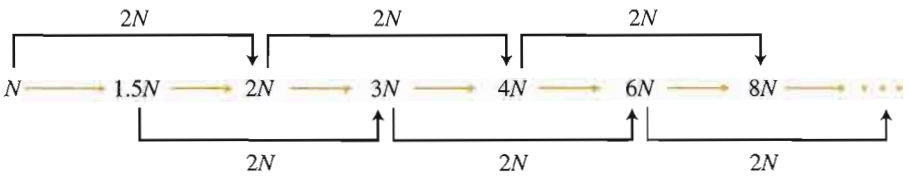
FIGURA 9.29
Hash dinámico ($N = 8$): reducción.

48, 35, 81, 70, 45, 33, 38 y 115

y verifique que el esquema final, luego de realizar las eliminaciones, quede igual al de la figura 9.31.

9.3.10 Método de las expansiones parciales

El método de las **expansiones parciales** consiste en incrementar en 50% el número de cubetas, haciendo de esta forma que dos expansiones parciales equivalgan a una total. Así, por ejemplo, si el número inicial de cubetas es N, y se hace una expansión parcial, el valor resultante será $1.5N$. Si se hacen otras expansiones parciales se tendrá $2N$, luego $3N$, y así sucesivamente.



A continuación se presenta un ejemplo de *hash* dinámico con expansiones parciales.

Clave	H(Clave)
42	2
24	0
15	3
21	1
12	0
67	3

Cubetas

0	1	2	3
24	21	42	15
12			67

Porcentaje de ocupación para expansión: 150%

FIGURA 9.30
Hash dinámico ($N = 4$): reducción.

FIGURA 9.31
Hash dinámico ($N = 4$): reducción.

Cubetas 0 1 2 3

Porcentaje de ocupación para expansión: 75%

96	57	26	
64		62	79
28		98	107

Porcentaje de ocupación para reducción: 225%

Ejemplo 9.18

Retome el ejemplo 9.14. Supongamos que hasta el momento se han almacenado los registros con claves 42, 24 y 15. Cuando se quiere insertar el registro con clave 53, el número de registros supera el máximo permitido ya que la densidad de ocupación supera 80%; por tal razón se realiza una expansión parcial. La figura 9.32 muestra el estado de las cubetas luego de expandir y reasignar los registros.

Observe que en este caso el valor de N no fue muy adecuado para distribuir uniformemente los registros a través de las cubetas. En la cubeta 0 se tiene una colisión, mientras que la cubeta 1 permanece vacía.

Supongamos ahora que se desea incorporar los registros con claves

21, 12, 14, 18, 49, 128, 22 y 23.

Al insertar el registro con clave 21 se supera la densidad de ocupación, por lo que se deben expandir nuevamente las cubetas y reasignar los registros. Se inserta a continuación el registro con clave 12, como se ve en la figura 9.33.

Al insertar el registro con clave 14, otra vez se supera el porcentaje de ocupación permitido. Se vuelven a expandir las cubetas y a reasignar los registros. El resultado final, luego de insertar todas las claves, se muestra en la figura 9.34.

Ejemplo 9.19

Dado un archivo organizado en dos cubetas ($N = 2$), donde cada cubeta tiene tres registros, se quiere almacenar las siguientes claves:

115, 96, 49, 79, 35, 27, 57, 89, 70, 64, 107, 45, 67, 98, 33, 28, 38, 104, 42 y 15.

Para este ejemplo se ha establecido una densidad de ocupación mayor a 82% para expansión y menor a 125% para reducción. A continuación se presenta la función *hash* que se utiliza:

$$H\ (\text{clave}) = \text{clave MOD } \textit{Número de cubetas}$$

FIGURA 9.32
Hash dinámico ($N = 4$): reducción.

Cubetas 0 1 2

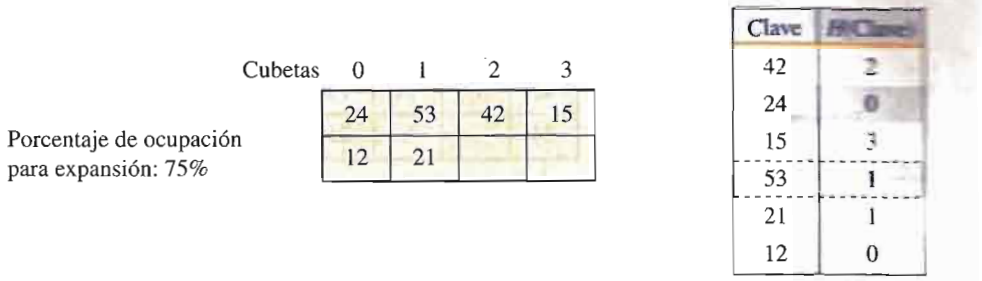
Porcentaje de ocupación para expansión: 66.66%

42		53
24		

15

Clave	*H*(Clave)
42	0
24	0
15	0
53	2

FIGURA 9.33
Hash dinámico ($N = 4$): expansión parcial.



Observe si la estructura que obtiene es igual a la que se presenta en la figura 9.35.

A continuación se presenta un ejemplo para ilustrar la reducción del número de cubetas en el método dinámico por transformación de claves, con expansiones parciales.

Ejemplo 9.20

Supongamos que se tiene un archivo en el estado que muestra la figura 9.34*b*. Elimine los registros con claves:

53, 18 y 128

y verifique si las cubetas y registros quedan igual a la gráfica que se muestra en la figura 9.36.

Ejemplo 9.21

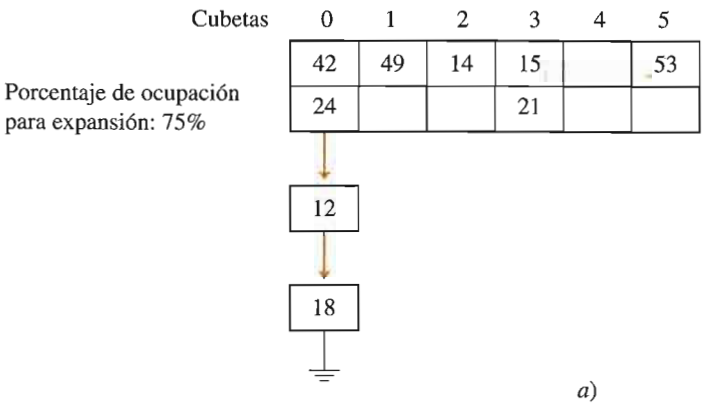
Dado el archivo de la figura 9.35 y las especificaciones dadas en el ejemplo 9.19, elimine las siguientes claves:

67, 104, 15, 45, 33, 79, 70 y 107.

Verifique si el esquema final que obtiene es igual al de la figura 9.37.

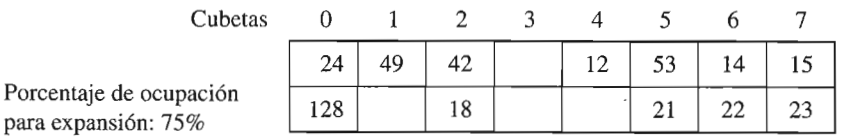
Finalmente, es importante señalar que el tamaño de las cubetas se debe establecer de acuerdo con el problema que esté intentando resolver. En los ejemplos presentados se han considerado inicialmente dos registros por cubeta. Sin embargo, este número es para que el lector observe el funcionamiento de los métodos al realizar expansiones y reducciones. Si el número de registros que utilizáramos fuera grande, entonces habría que ingresar gran cantidad de números para observar la expansión de cubetas.

Indudablemente, en la práctica se debe considerar un número mucho más grande de registros por cubeta. El número dependerá principalmente del tamaño de cada registro, de tal forma que una cubeta se pueda cargar en la memoria principal. En aplicaciones grandes, el número de registros por cubeta podría variar de 250 a 500. Si el número de registros por cubeta es pequeño y en forma continua se realizan inserciones y eliminaciones, entonces podría ocurrir que frecuentemente se deban realizar expansiones o reducciones, con la consabida pérdida de tiempo y alto costo, por la reasignación de los registros. Es el usuario quien debe definir entonces el número de registros por cubeta dependiendo del problema y de las actualizaciones que se realicen.



a)

Clave	*H*(Clave)
42	0
24	0
15	3
53	5
21	3
12	0
14	2
18	0
49	1



b)

Clave	*H*(Clave)
42	2
24	0
15	7
53	5
21	5
12	4
14	6
18	2
49	1
128	0
22	6
23	7

FIGURA 9.34
Hash dinámico: expansión parcial. *a)* Luego de expandir e insertar los registros con claves 14, 18 y 49. *b)* Luego de expandir e insertar los registros con claves 128, 22 y 23.

9.3.11 Listas invertidas

Las **listas invertidas** trabajan sobre algunos de los atributos —campos— de los registros. Los atributos pueden estar o no invertidos; es decir, pueden ser o no campos clave. Los atributos invertidos generan listas ordenadas de registros, lo cual facilita las búsquedas que se hagan en ellas. Los atributos no invertidos generan el universo, o sea para encontrar un determinado elemento —registro—, se deberá realizar una búsqueda secuencial.

Cubetas	0	1	2	3	4	5	6	7	8	9	10	11
Porcentaje de ocupación para expansión: 55.55%	96	49	98	27	64	89	42	115	104	57	70	35
			38	15	28			79		45		107
Porcentaje de ocupación para reducción: 166.66%								67		33		

FIGURA 9.35
Hash dinámico ($N = 2$): expansión parcial.

Las listas invertidas son muy recomendables cuando se trabaja sobre combinaciones de campos clave. Cuando se requiere una combinación de atributos en la búsqueda, este método resulta muy conveniente, ya que con una secuencia óptima de operadores AND y OR la búsqueda se puede llevar a cabo de forma eficiente.

La desventaja del método es que requiere de una estructura muy complicada para operar. Básicamente trabaja sobre árboles B^+ con prefijo. Analicemos a continuación un ejemplo.

Ejemplo 9.22

Supongamos que se tiene un archivo en el cual cada registro almacena la siguiente información:

Nombre	Profesión	Edad

Se tienen los datos de seis personas:

Juan	matemático	32
Daniel	físico	40
José	matemático	25

FIGURA 9.36
Hash dinámico ($N = 6$): reducción.

Cubetas	0	1	2	3	4	5
Porcentaje de ocupación para expansión: 150%	42	49	14	15	22	53
	24			21		
	12					

Clave	H(Clave)
42	0
24	0
15	3
21	3
12	0
14	2
49	1
22	4
23	5

Porcentaje de ocupación para reducción: 150%

Cubetas	0	1	2	3	4	5	6	7
	96	49	42	115	28			
	64	57	98	38				
		89		27				

35

FIGURA 9.37
Hash dinámico ($N = 8$): reducción.

Pascual	ingeniero	38
Miguel	ingeniero	43
Felipe	abogado	35

Considerando que los atributos *profesión* y *edad* están invertidos, a continuación se presentan algunas operaciones de búsqueda con sus correspondientes resultados, para que el lector observe el funcionamiento del método.

a) Lista de personas por *profesión*.

matemáticos	{Juan, José}
físicos	{Daniel}
ingenieros	{Pascual, Miguel}
abogados	{Felipe}

b) Lista de todas las personas con *profesión* matemático o físico, y con más de 25 años de *edad*.

(((profesión = matemático) OR (profesión = físico)) AND (edad > 25))

La lista formada según el atributo *profesión* es:

{Juan, José, Daniel}

Sobre esta lista se aplicará la segunda condición planteada en la búsqueda, de lo que resulta:

{Juan, Daniel}

c) Lista de todos los ingenieros menores de 50 años y mayores de 40.

(profesión = ingeniero) AND ((edad < 50) AND (edad > 40))

La lista formada según el atributo *profesión* es:

{Pascual, Miguel}

A partir de esta lista, se buscarán los registros que cumplan con las condiciones impuestas sobre el atributo *edad*. La lista resultante será:

{Miguel}

Considerando que solamente el atributo *profesión* está invertido, se presentan algunas operaciones de búsqueda con sus correspondientes resultados.

a) Lista de todas las personas con *profesión* matemático o físico, y con más de 25 años de edad.

((profesión = matemático) OR (profesión = físico)) y búsqueda secuencial en la lista de los registros marcados para localizar aquellos con edad > 25.

{Juan, José, Daniel} y sobre esta lista una búsqueda secuencial para encontrar a los individuos mayores de 25 años.

b) Lista de todos los ingenieros menores de 50 años y mayores de 40.

(profesión = ingeniero) y búsqueda secuencial en la lista de los registros marcados para localizar aquellos con edad > 40 y edad < 50.

{Pascual, Miguel} y búsqueda secuencial sobre esta lista para encontrar a los individuos menores de 50 y mayores de 40.

c) Lista de todos los abogados mayores de 40 años.

(profesión = abogado) y búsqueda secuencial en la lista de los registros marcados para localizar a aquellos con edad > 40.

{Felipe} y búsqueda secuencial sobre esta lista para encontrar a los individuos mayores de 40 años.

En este caso la solución es la lista vacía. No hay ningún registro que tenga los atributos pedidos.

Ejemplo 9.23

La Dirección General de Reclusorios ha decidido crear una base de datos con información sobre sus presos. El esquema que se considera es el siguiente:

nombre_reo	clave_reo	edad	escolaridad	cod_delito	nacionalidad

La escolaridad está codificada como:

1. Analfabeto
2. Primaria
3. Secundaria

4. Preparatoria
5. Universidad
6. Posgrado

Los códigos de delito (cod_delito) están codificados como:

1. Delito contra la salud
2. Robo con arma de fuego
3. Acoso sexual
4. Otros

Considerando que los atributos *escolaridad* y *cod_delito* están invertidos, se presentan algunas operaciones de búsqueda con sus correspondientes resultados:

a) Los reclusos analfabetos con menos de 20 años de edad.

(escolaridad = 1) y búsqueda secuencial en la lista de los registros marcados para localizar a aquellos con edad < 20.

b) Los reclusos con posgrado, cuya edad está comprendida entre 20 y 50 años, y que cometieron el delito calificado como acoso sexual.

((escolaridad = 6) AND (cod_delito = 3)) y búsqueda secuencial en la lista de los registros marcados para localizar a aquellos cuya edad está comprendida entre 20 y 50 años de edad.

c) Los reos estadounidenses.

(búsqueda secuencial en todo el archivo para localizar a los reos de nacionalidad estadounidense.)

d) Los reos que cometieron robo con arma de fuego, menores de 22 años, o los que cometieron delito contra la salud, menores de 30 años.

((cod_delito = 2) y (búsqueda secuencial en la lista de los registros marcados para localizar edad < 22)) OR ((cod_delito = 1) y (búsqueda secuencial en la lista de los registros marcados para localizar edad < 30)).

Se ha mencionado que las listas generadas por atributos invertidos están ordenadas; por lo tanto, el tiempo de procesamiento está determinado por la lista de mayor tamaño. Una secuencia adecuada de operadores AND y OR puede ayudar a disminuir el tiempo de procesamiento. Analicemos el siguiente ejemplo.

Ejemplo 9.24

Supongamos que se tienen las listas *L*1, *L*2 y *L*3 de 1 000, 5 y 100 elementos, respectivamente. Si se necesitara unir las tres listas, el orden en el cual se hiciera la unión sería determinante en cuanto al número total de elementos con los cuales se trabaja.

1. $(L1 \cup L2) \cup L3 = (1\,000 + 5) \cup L3$
   $= 1\,005 + (1\,005 + 100) = 2\,110$

2. $(L1 \cup L3) \cup L2 = (1\,000 + 100) \cup L2$
   $= 1\,100 + (1\,100 + 5) = 2\,205$

3. $(L2 \cup L3) \cup L1 = (5 + 100) \cup L1$
   $= 105 + (105 + 1\,000) = 1\,210$

Es fácil observar que la mejor secuencia es la tercera y el resultado es 1 210; y que la peor secuencia es la segunda y el resultado es 2 205.

Ejemplo 9.25

Sean A, B, C y D listas de 100, 300, 250 y 80 elementos, respectivamente. Si se necesitara su unión, algunas de las distintas secuencias que se tendrían son:

1. $((A \cup B) \cup C) \cup D = ((100 + 300) \cup C) \cup D$
   $= (400 + (400 + 250)) \cup D$
   $= (1\,050 + (650 + 80))$
   $= 1\,780$

2. $((B \cup C) \cup A) \cup D = ((300 + 250) \cup A) \cup B$
   $= (550 + (550 + 100)) \cup B$
   $= (1\,200 + (650 + 80))$
   $= 1\,930$

3. $((A \cup D) \cup B) \cup C = ((100 + 80) \cup B) \cup C$
   $= (180 + (180 + 300)) \cup C$
   $= (660 + (480 + 250))$
   $= 1\,390$

4. $((A \cup D) \cup C) \cup B) = ((100 + 80) \cup C) \cup B$
   $= (180 + (180 + 250)) \cup B$
   $= (610 + 730)$
   $= 1\,340$

Con los ejemplos queda demostrado cómo influye el tamaño de las listas en el número total de elementos a procesar. Es posible concluir, entonces, que resulta mucho más eficiente dejar las listas de mayor tamaño para unirlas al final.

9.3.12 Multilistas

El método de búsqueda **multilistas** permite acceder a la información que se encuentra ordenada utilizando campos clave. A un registro se puede llegar por diferentes caminos. Cada camino se establece en función del campo clave sobre el cual se haga la búsqueda.

La forma más eficiente de representar multilistas es utilizando listas. A continuación se presenta un ejemplo de este método.

Ejemplo 9.26

Supongamos que se tiene un archivo en el cual cada registro almacena la siguiente información:

Nombre	Profesión	Categoría
Juan	matemático	1
Daniel	físico	2
José	matemático	2
Pascual	ingeniero	3
Miguel	ingeniero	1
Felipe	abogado	2

La figura 9.38 representa las multilistas correspondientes a los datos dados. En este caso, la información de cada individuo puede ser accesada por medio de su profesión y de su categoría, que son justamente los atributos que permiten realizar búsqueda directa en el archivo. Como se puede observar en la siguiente figura, se tiene una lista por profesión y otra por categoría.

En general, las multilistas son recomendables cuando la búsqueda se hace sobre un solo atributo. En caso de necesitarse una combinación de atributos es preferible usar listas invertidas.

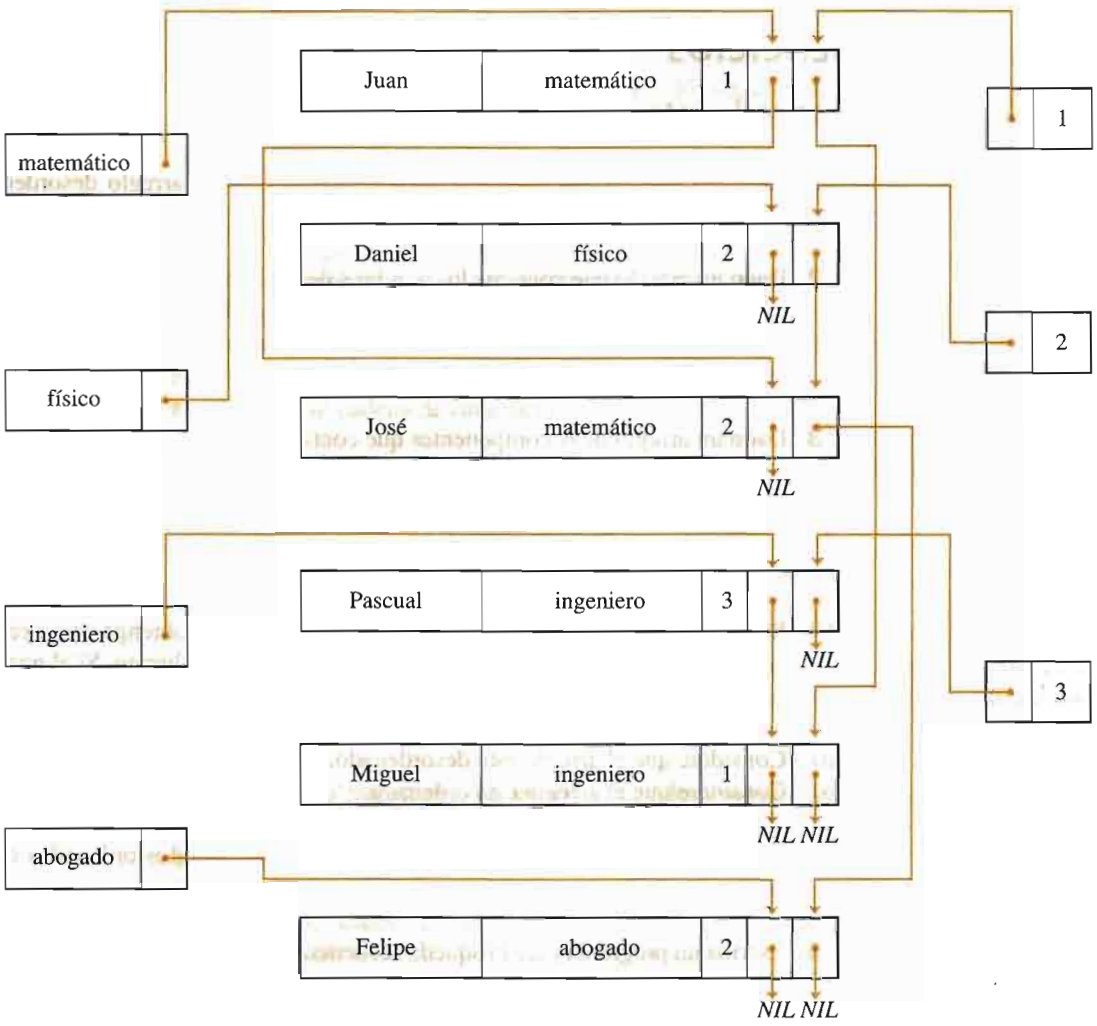


FIGURA 9.38
Multilistas.

EJERCICIOS

Búsqueda interna

1. Escriba un programa para búsqueda secuencial en un arreglo desordenado, que obtenga todas las ocurrencias de un dato dado.

2. Dado un arreglo que contiene los nombres de N alumnos ordenados alfabéticamente, escriba un programa que encuentre un nombre dado en el arreglo. Si lo encuentra debe dar como resultado la posición en la que lo encontró. En caso contrario, debe enviar un mensaje adecuado.

3. Dado un arreglo de N componentes que contienen la siguiente información:

- Nombre del alumno
- Promedio
- Número de materias aprobadas

Escriba un programa que lea el nombre de un alumno y obtenga como resultado el promedio y el número de materias aprobadas por dicho alumno. Si el nombre dado no está en el arreglo, envíe un mensaje adecuado.

a) Considere que el arreglo está desordenado.
b) Considere que el arreglo está ordenado.

4. Escriba un programa para búsqueda secuencial en arreglos ordenados de manera descendente.

5. Escriba un programa para búsqueda secuencial en listas simplemente ligadas que se encuentran desordenadas. Si el elemento se encuentra en la lista, indique el número de nodo en el cual se encontró. En caso contrario, emita un mensaje adecuado.

6. Escriba un programa para búsqueda secuencial en listas simplemente ligadas, ordenadas de manera descendente.

7. Escriba un programa de búsqueda binaria en arreglos ordenados.

a) De manera ascendente.
b) De manera descendente.

8. Resuelva el inciso *b* del problema 3 utilizando el algoritmo de búsqueda binaria.

9. Defina una clase *Arreglo*, según lo visto en el capítulo 1. En la clase debe incluir por lo menos dos métodos —de los estudiados en este capítulo— para buscar un elemento almacenado en el arreglo.

10. Dado que se requiere almacenar los registros con clave

23, 42, 5, 66, 14, 43, 59, 81, 37, 49, 28, 55, 94, 80 y 64

en un arreglo de 20 elementos, defina una función *hash* que distribuya los registros en el arreglo. Si hubiera colisiones, resuélvalas aplicando el método de reasignación lineal.

11. De un grupo de N alumnos se tienen los siguientes datos:

- Matrícula: valor entero comprendido entre 1 000 y 4 999
- Nombre: cadena de caracteres
- Dirección: cadena de caracteres

El campo clave es matrícula. Los N registros han sido almacenados en un arreglo, aplicando la siguiente función *hash*:

$$H\,(\text{clave}) = \text{dígitos\_centrales}(\text{clave}^2) + 1$$

Las colisiones han sido tratadas con el método de doble dirección *hash*.

Escriba un subprograma que lea la matrícula de un alumno y regrese como resultado su nombre y dirección. En caso de no encontrarlo, emita un mensaje adecuado.

12. Se quiere almacenar en un arreglo los siguientes datos de N personas:

- Clave de contribuyente: alfanumérico, de longitud 6.
- Nombre: cadena de caracteres
- Dirección: cadena de caracteres
- Saldo: real

Defina una función *hash* que permita almacenar en un arreglo los datos mencionados. Utilice el método de encadenamiento para resolver las colisiones.

13. Presente y explique una función *hash* que permita almacenar en un arreglo los elementos de la tabla periódica de los elementos de química y sus propiedades, de manera uniforme. La clave está dada por el nombre de los elementos.

14. Utilice la función definida en el ejercicio anterior para insertar y eliminar los elementos que se presentan a continuación:

Insertar: sodio, oro, osmio, litio, boro, cobre, plata, radio.
Eliminar: oro, osmio, boro, cobre, plata.

15. Dados los 12 signos del zodiaco (capricornio, acuario, piscis, aries, tauro, géminis, cáncer, leo, virgo, libra, escorpión, sagitario).

a) Escriba un subprograma para almacenarlos en una estructura de tries.
b) Escriba un subprograma de búsqueda para los signos, almacenados según lo especificado en el inciso anterior.

Búsqueda externa

16. Se han almacenado en un archivo secuencial los datos de los empleados de un supermercado:

- Nombre
- Registro Federal de Contribuyentes
- Fecha de ingreso
- Sueldo

Escriba un programa para buscar secuencialmente los datos de un empleado, dado su nombre como entrada.

a) Considere que el archivo está desordenado.
b) Considere que el archivo está ordenado.

17. Escriba un programa de búsqueda binaria en archivos secuenciales ordenados.

18. Defina una función *hash* que permita almacenar y posteriormente recuperar los elementos de la tabla periódica de los elementos de química en un archivo. La clave está dada por el nombre de los elementos. Resuelva las colisiones utilizando un área independiente para almacenar los elementos colisionados.

19. Se desea crear un archivo con información sobre pinos mexicanos. Cada registro contiene los siguientes datos:

- Nombre del pino
- Tipo de hojas
- Tipo de cono

El campo clave es *Nombre del pino*. Defina una función *hash* para almacenar, y posteriormente buscar, los siguientes pinos: *Cembroides, Monophylla, Nelsonii, Flexilis, Lumholtzii, Leiophylla, Douglasiana, Teocote, Herrerai, Montezumae, Cooperi, Contorta, Pondarosa, Arizonica, Caribaea, Patula, Radiata, Muricata, Remorata.*

Resuelva las colisiones utilizando un área común para almacenar los elementos colisionados.

20. Utilice la función definida en el ejercicio 13 para insertar y eliminar los elementos que se indican a continuación:

Insertar: sodio, oro, osmio, litio, boro, cobre, plata, radio.
Eliminar: oro, osmio, boro, cobre, plata.

El número de cubetas es dos ($N = 2$) y cada cubeta tiene dos registros. La densidad de ocupación permitida es 80%; en caso de superar este porcentaje se aplicarán expansiones totales.

a) Dibuje un esquema de la organización después de insertar los elementos osmio y plata; y luego de eliminar oro, boro y plata.
b) Diga qué claves originaron que el número de cubetas se expandiera o redujera.

21. Resuelva el problema anterior, pero ahora aplicando expansiones parciales, en caso de tener un porcentaje de ocupación mayor al permitido.

22. Sea $N = 2$ el número de cubetas. Cada cubeta tiene dos registros y se establece una densidad de ocupación permitida de 85%. Una vez superada esta densidad, se aplicarán expansiones parciales.

H (clave) = clave MOD N
Claves a insertar: 36, 11, 48, 06, 75, 65, 38, 88, 23, 14, 12

a) Dibuje un esquema de la organización después de insertar los elementos 06, 38, 23, 12.
b) Diga qué claves originaron que el número de cubetas se expandiera.

23. Considere el archivo del problema anterior. Elimine los registros con claves 75, 06, 65, 14, 12, 36, 23.

a) Dibuje un esquema de la organización después de eliminar los elementos 75, 14, 12.
b) Diga qué claves originaron que el número de cubetas se redujera.

24. Sea $N = 4$ el número de cubetas. Cada cubeta tiene dos registros, y se establece una densidad de ocupación permitida de 80%. Defina una función *hash* para:

Insertar las claves 77, 34, 23, 26, 39, 60, 19, 43, 70, 51, 17, 28
Eliminar las claves 23, 39, 60, 43, 17

a) Aplique expansiones totales.
b) Aplique expansiones parciales.

25. Determine cuál es el número de cubetas necesario para almacenar en un archivo nombre, apellido, edad, escolaridad y delito cometido por reos del Reclusorio Norte. El reclusorio tiene 2 700 presos.

Nota: Utilice el método de las expansiones totales. Cada cubeta tiene 50 registros. Al tener 80% de llenado se expande.

26. Determine cuál es el número de cubetas necesario para almacenar en un archivo los registros de los 5 000 000 de clientes que maneja una empresa de tarjetas de crédito.

Nota: Utilice el método de las expansiones parciales. Cada cubeta tiene 500 registros. Al tener 85% de llenado se expande.

27. Se tiene un archivo con registros que almacenan información sobre clientes de distintas sucursales bancarias. Los datos que se manejan por cada cliente son:

- Clave de la sucursal
- Nombre del titular
- Número de cuenta
- Saldo
- Número de préstamo
- Importe

Se tiene inversión sobre el campo clave *sucursal*.

a) Obtenga los registros de los clientes que tengan un préstamo mayor a $5 000 en la sucursal *Lima*.
b) Obtenga los registros de los clientes que tengan un préstamo mayor a $5 000 en la sucursal *Lima* y un saldo en su cuenta mayor a $3 000 en la sucursal *Río*.
c) Obtenga los registros de los clientes de la sucursal *Río* que tengan en su cuenta un saldo mayor a $6 000, o los registros de los clientes de la sucursal *Quito* que tengan un préstamo menor a $1 000 y un saldo en su cuenta mayor a $2 000.
d) Obtenga los registros de los clientes de la sucursal *Córdoba* que tengan un saldo mayor a $5 000 o un préstamo menor a $1 000.
e) Si se quiere determinar:

(sucursal = "Lima") OR (sucursal = "Quito") OR
(sucursal = "Río") OR (sucursal = "Córdoba")

y las correspondientes listas son de 100, 50, 120 y 200 elementos, respectivamente, ¿cuál será la secuencia óptima para alcanzar un costo mínimo?

28. En un archivo se ha almacenado la tabla periódica de los elementos químicos, junto con sus propiedades:

- Nombre
- Número atómico (*NA*)
- Peso atómico (*PA*)
- Punto de ebullición (*PE*)
- Punto de fusión (*PF*)
- Densidad (*DEN*)
- Electronegatividad (*EO*)
- Conductancia eléctrica (*CE*)
- Conductancia térmica (*CT*)

Se tiene inversión sobre los campos *punto de ebullición* y *punto de fusión*.

a) Obtenga los registros de los elementos alcalinotérreos. Éstos se determinan por las siguientes características: $EO = 2$; $1.54 < DEN < 5.01$ y su *PF* está comprendido entre los valores 922 y 1 560.
b) Obtenga los registros de los elementos del grupo 6B. Éstos se determinan por las siguientes características: $EO = -2, 4$ o 6; su *PE* está comprendido entre los valores 90.18 y 12.61.

29. En un archivo se han almacenado los datos de *N* profesionales.

- Clave de contribuyente
- Nombre
- Profesión
- Nacionalidad

Se tiene inversión sobre los campos *profesión* y *nacionalidad*:

a) Obtenga los registros de todos los ingenieros mexicanos.
b) Obtenga los registros de todos los ingenieros mexicanos de más de 60 años de edad.
c) Obtenga los registros de todos los ingenieros mexicanos de más de 60 años de edad o los pintores uruguayos.
d) Obtenga los registros de todos los abogados peruanos o los médicos chilenos de menos de 30 años.
e) Si se quiere determinar:

(profesión = ingeniero) OR (profesión = pintor) OR (profesión = médico) y las listas son de 100, 200 y 300 claves, respectivamente, ¿cuál será la secuencia óptima para alcanzar un costo mínimo?