Técnicas de búsqueda

HORADIO RODRIGUEZ HONTORIA

1. INTRODUCCION

La búsqueda en tablas en memoria es uno de los procesos más frecuentes en cualquier programa, especialmente si se trata de una aplicación de gestión. Raro es el programa que no incluya el manejo (creación, consulta, manipulación, actualización, etc...) de una o varias tablas. Se trata, por otra parte, en la mayor parte de los casos del proceso que más tiempo CPU consume. El hecho de que normalmente este tiempo sea pequeño en relación con el tiempo total de ejecución del programa no es razón suficiente para que no intentemos minimizarlo o por lo menos reducirlo apreciablemente, máxime si, como veremos, la diferencia de esfuerzo de programación para hacer una búsqueda eficiente y otra que no lo sea es pequeña.

El planteo del problema en su forma más general es el siguiente:

Dado un conjunto de *N* registros R_1, R_2, ... R_N identificables por sus claves K_1, K_2, ... K_N (que supondremos distintas) se trata de ver si dada una clave K podemos encontrar un registro del conjunto que tenga a K por clave asociada. En caso afirmativo decimos que la búsqueda ha tenido éxito y nos interesará saber la posición del registro encontrado para hacer uso de él. En caso negativo decimos que la búsqueda ha fracasado y en ciertos casos nos interesará incorporar el nuevo registro a nuestro conjunto.

No presumiremos a priori una organización determinada de la tabla ni una ordenación de algún tipo en sus elementos. El estudio de los diversos métodos de búsqueda nos impondrá en cada caso las limitaciones necesarias.

Por razones de simplicidad ignoraremos, en lo que sigue, en cada registro todo lo que no sea la clave. Supondremos que el resto de la información está aparte (en memoria o en un almacenamiento externo, nos es indiferente). Supondremos también que las claves son de longitud constante y alfanuméricas.

2. METODOS DE BUSQUEDA

Para empezar a estudiar el problema vamos a intentar una sistematización de los métodos de búsqueda más usuales.

Podemos clasificar los métodos de búsqueda según varios criterios.

Atendiendo a la localización de la tabla podemos hablar de búsquedas internas o externas (según la localización en memoria o en un almacenamiento exterior de las claves).

Hablaremos de búsqueda estática si la tabla está ya construida y es inamovible y de búsqueda dinámica si la tabla es susceptible de actualizaciones (altas y/o bajas).

Si nos fijamos en la organización de la tabla podemos encontrar tablas lineales (ordenadas o no) y tablas encadenadas.

Todos estos criterios de clasificación son de alguna forma complementarios y atienden a los diversos aspectos que inciden en el proceso de la búsqueda. Sin embargo para nuestra exposición nos guiaremos por una última clasificación que atiende al propio proceso de la búsqueda y que condiciona todos los aspectos que hemos visto:

Si examinamos cualquier proceso de búsqueda veremos que hay un elemento común a todos ellos: de alguna manera una comparación entre la clave que buscamos y alguna de las de la tabla debe gobernar la búsqueda.

Esta comparación la podemos llevar a cabo de diversas maneras: usando la clave misma, en forma global, como elemento comparador; transformando la clave de manera que nos venga facilitada la comparación o bien efectuando una partición de la misma (a nivel de octetos o de bits) y usando las partes como elementos en la comparación.

Siguiendo este criterio hemos establecido la clasificación de la figura 1.

En la exposición que sigue usaremos en lo posible un método deductivo justificando la utilización de las diferentes variantes de los algoritmos como solución a los problemas que detectemos en las anteriores. En algunos casos detallaremos el algoritmo, en forma lo más estructurada posible, en otros casos acompañamos el programa en COBOL correspondiente. La tabla viene definida en la figura 2 (lógicamente no en todos los métodos usaremos todos los campos). La clave a buscar estará contenida en el campo *CLAVE*. El campo *ENCONTRADO* contendrá la posición de la clave encontrada en caso de búsqueda con éxito y cero en caso contrario.

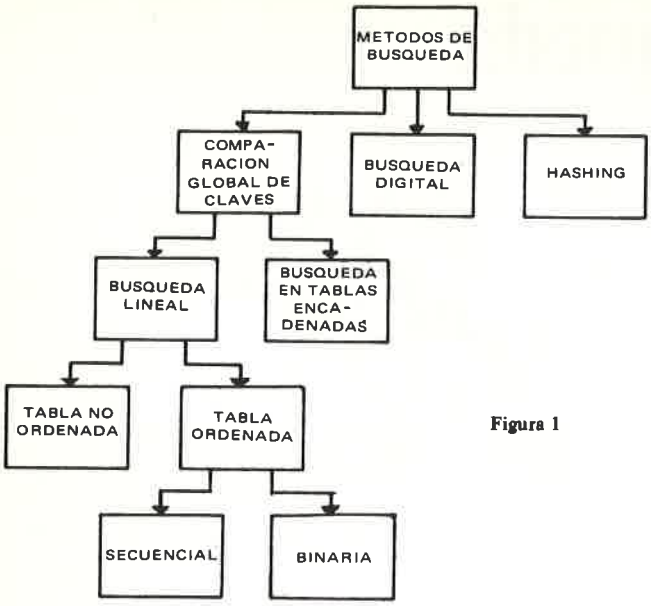


Figura 1

```
01  TABLA-OBJETO.
    02      FILLER      PIC  X(17)  VALUE LOW-VALUE.
    02      T-OBJE OCCURS 1000
                   DEPENDING ON  K-MAX
                   ASCENDING KEY IS TO-CLA
                   INDEXED BY  K.
       03   TO-CLA      PIC X(10).
       03   TO-POIN     PIC 999.
       03   TO-CAB      PIC 999.
       03   TO-BAL      PIC S9.
```

FIGURA 2

2.1. Métodos de comparación global de claves

Englobamos en este apartado a todos los métodos en los que la búsqueda es guiada por comparaciones de la clave a buscar con diferentes claves de la tabla, realizándose todas las comparaciones a nivel global (es decir, consideramos a la propia clave como elemento de comparación).

Dividiremos los métodos de este apartado en dos grandes grupos atendiendo a la organización de la tabla: tablas de organización lineal y tablas de organización encadenada.

2.1.1. Tablas organizadas linealmente

En ellas los distintos elementos adoptan una estructura secuencial. La clave K-ésima está en la posición K de la tabla y la siguiente en la posición K + 1. En caso de ampliación de la tabla los nuevos elementos irían al final (si la tabla no está ordenada) u ocuparían su puesto obligando a un corrimiento a todos los elementos siguientes. En caso de querer borrar un elemento deberíamos eliminarlo desplazando una posición los siguientes o marcarlo de alguna manera, lo que obligaría seguramente a aumentar la longitud de la clave.

Saltan a la vista las ventajas de una organización de este tipo (en relación con el problema concreto de la búsqueda):

- ocupación mínima de memoria
- facilidad de acceso
- métodos de búsqueda más sencillos.

Y los inconvenientes:

- dificultad de actualización (tanto ampliación como eliminación de claves);
- rigidez de organización que impide (si la tabla ha de estar ordenada) aprovechar información relativa a los elementos a buscar, de la que a lo mejor disponemos, y que permitiría acortar el tiempo de búsqueda (por ejemplo un dato tan importante como la frecuencia de aparición de las claves, que en muchos casos puede ser estimada, no podría utilizarse en un método en que la tabla fuera lineal y ordenada).

Parece pues lógico dividir este apartado en otros dos según que la tabla esté o no clasificada (la clave de clasificación sería naturalmente la propia clave de búsqueda y en lo que sigue, y sin que ello suponga ninguna restricción, supondremos que la ordenación es ascendente).

2.1.1.1. Tablas no ordenadas

Una tabla de estas características deja pocas posibilidades a la imaginación a la hora de pensar en métodos de búsqueda.

La única posibilidad que tenemos es la de recorrer la tabla desde el primer elemento, secuencialmente, hasta que se produzca una cualquiera de las siguientes condiciones:

1 – encontrar la clave
2 – final de tabla.

En el 1er. caso la búsqueda habrá tenido éxito y en el 2.º no.

El programa ALGϕ31 se adapta a este procedimiento.

Naturalmente el algoritmo es mejorable: basta introducir la clave que buscamos en la posició $K_{máx} + 1$ (siendo $K_{máx}$ el n.º de elementos de la tabla) para que nos evitemos una de las dos comparaciones ya que en este caso siempre se producirá la condición 1 antes de llegar al final de la tabla.

El programa ALGϕ41 hace uso de esta mejora (en dicho programa el campo GUAR-CLA nos sirve para conservar el contenido de la posición $K_{máx} + 1$).

Aún podemos mejorar el sistema aprovechándonos de las posibilidades de indexación. En ALGϕ51 establecemos comparaciones con los elementos K y K + 1 y por lo tanto podemos incrementar el índice K de 2 en 2 lo cual nos reduce en un 50 % el tiempo empleado en la operación de incrementación.

Lógicamente si se colocan las claves por orden decreciente de frecuencia de aparición (caso de conocer este dato) disminuiría notablemente el tiempo medio de acceso en caso de éxito en la búsqueda. En el caso de que el elemento no estuviera en la tabla no habría forma de evitar el recorrerla en su totalidad. Hay que señalar que una ordenación de este tipo no conllevaría una rigidez semejante a la de lo que hemos llamado tablas ordenadas ya que en nuestro caso la ordenación sería un simple factor de aceleración de la búsqueda sin que sea un factor intrínseco al método de búsqueda.

2.1.1.2. Búsqueda en tablas ordenadas

El hecho de que las claves aparezcan en la tabla ordenadas en orden ascendente nos va a permitir utilizar una serie de algoritmos que tengan en cuenta este factor.

El primer sistema que se nos puede ocurrir es simplemente adaptar los métodos del apartado anterior estableciendo una condición de finalización anterior a las que allí veíamos: es lo que hemos llamado búsqueda secuencial en la tabla ordenada. Englobaremos en el otro apartado: búsquedas binarias, todos los métodos que aprovechan la ordenación de la tabla para crear implícitamente una estructura de árbol binario que guíe la búsqueda.

2.1.1.2.1. Búsqueda secuencial

Se trata de sustituir la condición de final de tabla del ALGϕ31 por otra que concluya la búsqueda en cuanto la clave investigada sea superior a la que buscamos. Debemos suponer que en Kmáx + 1 hemos almacenado una clave superior a la máxima que podamos alcanzar.

El algoritmo sería

1 i = 1
2 hacer 4 hasta que $K \leqslant K_i$
3 Si $K = K_i$ éxito, si no fracaso
4 i = i + 1

De forma similar podríamos adaptar el ALGϕ51.

Desde luego sólo se produce un ahorro de tiempo, respecto a los métodos de tabla no ordenada, usando este procedimiento, cuando la búsqueda es infructuosa.

El algoritmo que hemos expuesto es precisamente el que utiliza la instrucción SEARCH del COBOL.

Una última observación respecto a estos métodos secuenciales: en caso de búsqueda con éxito el n.º de comparaciones es en media $K_{máx}/2$ (ya hemos visto que en algún caso las comparaciones son dobles). En caso de búsqueda infructuosa el n.º de comparaciones es $K_{máx}$ si la tabla no está ordenada y $K_{máx}/2$ en los casos de tabla ordenada.

2.1.1.2.2. Búsquedas binarias

La idea en que se basa una búsqueda binaria es la siguiente:

Si tenemos la tabla formada por las claves

$K_1 \quad K_2 \quad \ldots \quad K_N$

y en determinado momento comparamos K con Ki se pueden producir las siguientes situaciones

K < Ki (podemos eliminar de la búsqueda las claves Ki, Ki + 1 ... K_N)

K = Ki éxito en la búsqueda

K > Ki (podemos eliminar de la búsqueda las claves K1, K2, ... Ki)

El empleo de comparaciones dobles en vez de las sencillas que hasta ahora hemos tratado es lo que caracteriza a los métodos binarios.

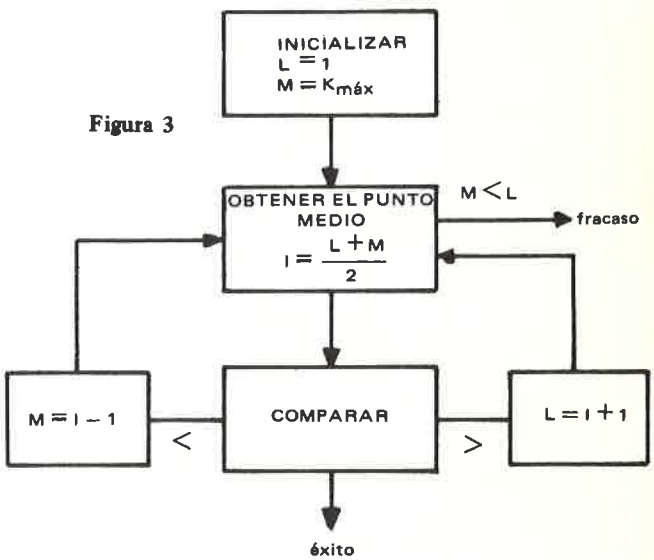
Para mejor comprender los algoritmos que a continuación expondremos nos ayudaremos con árboles de decisión binarios que asociaremos a cada uno de aquellos.

Los nodos no terminales j corresponderán a las diversas claves Kj contenidas en la tabla. Existirán por lo tanto Kmáx nodos no terminales. Los nodos terminales i corresponden a claves no presentes en la tabla comprendidas entre i e i + 1. La raíz del árbol indica el elemento de la tabla por el cual empezamos la búsqueda. La fila en que se encuentra un nodo nos indica el n.º de comparaciones necesario para llegar a él. De cada nodo no terminal salen 2 ramas (hijos derecho e izquierdo). El algoritmo seguirá la rama derecha si K > Ki y la izquierda si K < Ki (si K = Ki la búsqueda concluye con éxito).

La primera forma que se nos ocurre para adaptar las ideas anteriores consiste en comenzar por la mitad de la tabla, efectuar las comparaciones efectuadas arriba limitando nuestra búsqueda a una subtabla de tamaño mitad a la anterior y repetir el proceso hasta encontrar la clave o reducir la tabla a cero elementos.

El algoritmo de búsqueda sería el de la figura 3.

Figura 3



L y M representan en cada momento los elementos inicial y final de la subtabla considerada. I es el punto medio de dicha subtabla.

Este algoritmo (ALGϕ71) corresponde a la sentencia *SEARCH ALL* del COBOL.

La figura 4 nos muestra el árbol binario asociado para el caso $K_{máx} = 16$.

Así como en la búsqueda lineal el n.º medio de comparaciones era $K_{máx}$ en la búsqueda binaria es del orden de $\log_2 K_{máx}$ (se puede demostrar que el n.º máximo es $\log_2 K_{máx} + 1$ y el medio $\log_2 K_{máx} - 1$). Ello, como veremos, nos permitirá decantarnos hacia el uso de búsquedas binarias a partir de un n.º determinado de elementos.

Un nuevo método, derivado de éste es el de la búsqueda binaria uniforme. La idea es sustituir los 3 índices l, m, i que señalan el centro y los límites de

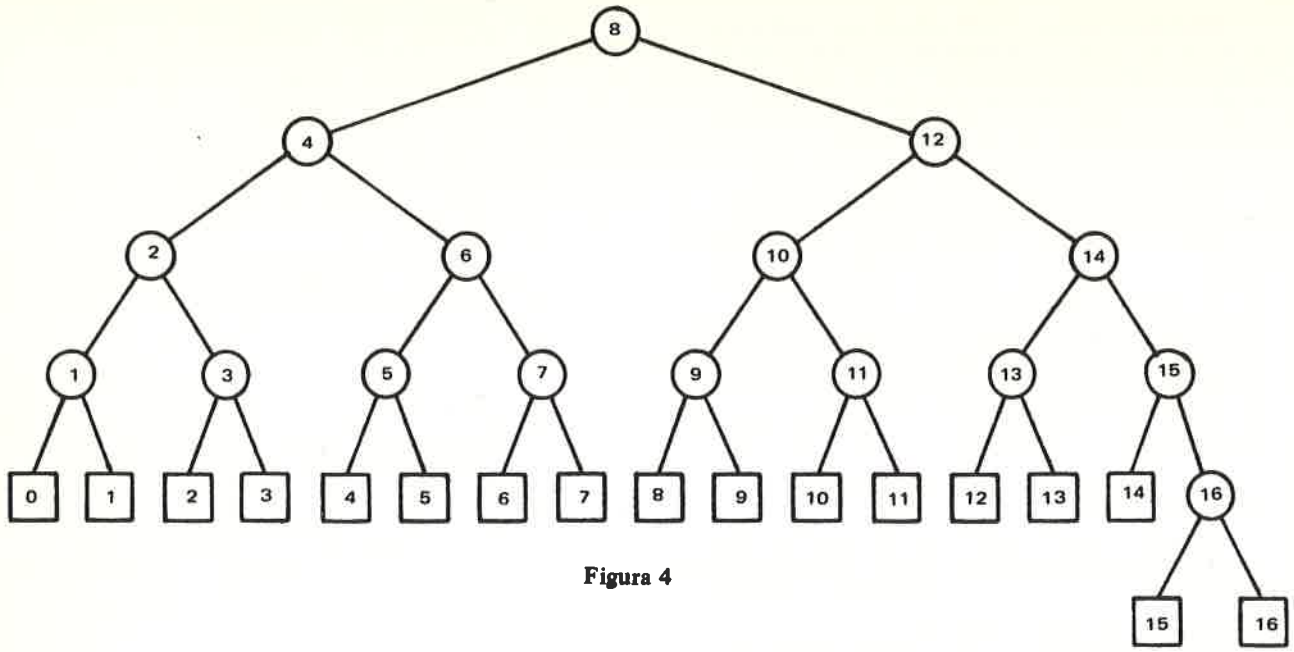


Figura 4

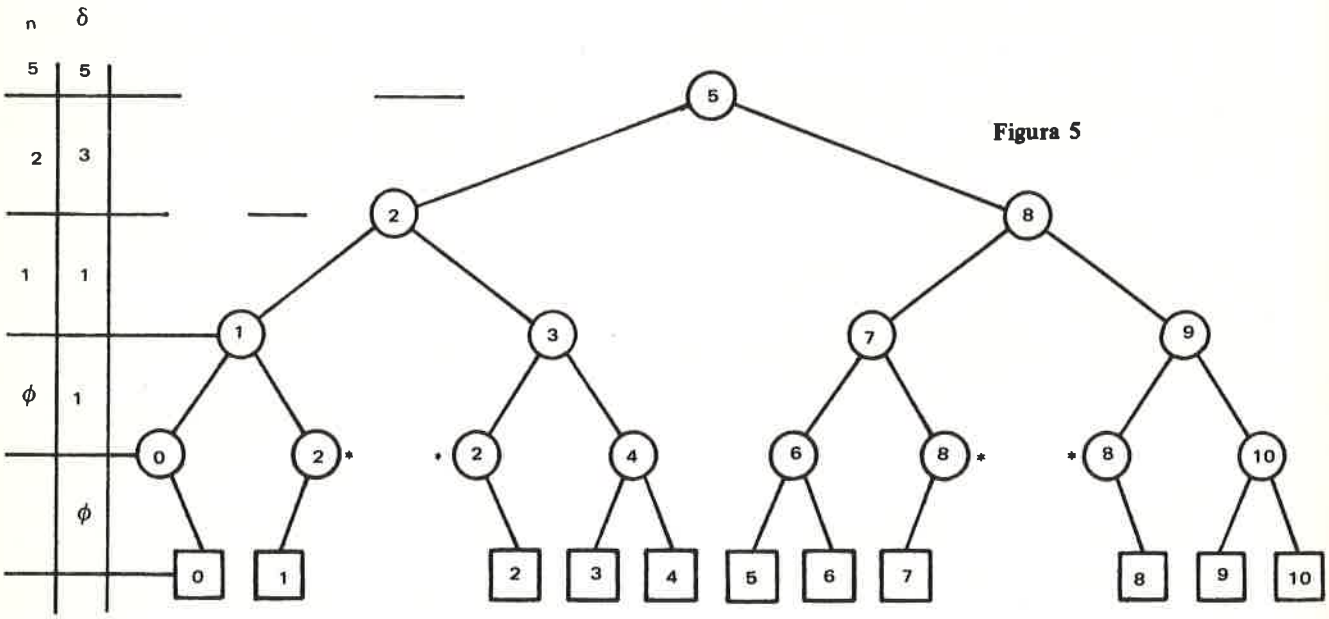


Figura 5

la subtabla por sólo dos que señalen el centro y la semiamplitud de dicha subtabla.

El algoritmo sería

1 $I = \frac{K_{máx}}{2}$

$M = \frac{K_{máx}}{2}$

2 Hacer 4, 5 hasta que K = Ki o M = O

3 Si K = Ki éxito, si no, fracaso

4 Si K < Ki $I = I - \frac{m}{2}$ si no

$I = I + \frac{m}{2}$

5 $m = \frac{m}{2}$

El programa correspondiente es el ALGϕ81 y el árbol correspondiente está en la figura 5 (para el caso $K_{máx} = 10$).

Un simple examen del algoritmo anterior nos indica que es más costoso que una búsqueda binaria normal. Un examen del árbol asociado nos señala la presencia de elementos redundantes que darán lugar a comparaciones extras y que incrementarán el número medio de comparaciones por búsqueda. ¿Cuál es pues la ventaja o la justificación de este algoritmo? Tal como lo hemos expuesto no hay ninguna; ahora bien podemos efectuar alguna modificación que lo haga competitivo.

El método se llama búsqueda binaria uniforme debido a que la diferencia entre el n.º de un nodo del nivel *m* y su antecesor en el nivel *m-1* es constante (= δ en la figura 5) para todos los nodos no terminales del nivel. Ahora bien estos valores de δ podemos tenerlos calculados y guardados en una tabla de manera que los pasos 4 y 5 del anterior algoritmo se reduzcan a consultas en una tabla auxiliar. Por otra parte como habrá tantos δ como niveles y el n.º de niveles es del orden de $\log_2 K_{máx}$ resulta que la tabla auxiliar será relativamente corta.

Los elementos de dicha tabla auxiliar *DELTA* pueden calcularse con arreglo a la siguiente fórmula:

$$DELTA\ (m) = \left[\frac{K_{máx} + 2^{m-1}}{2^m}\right]$$

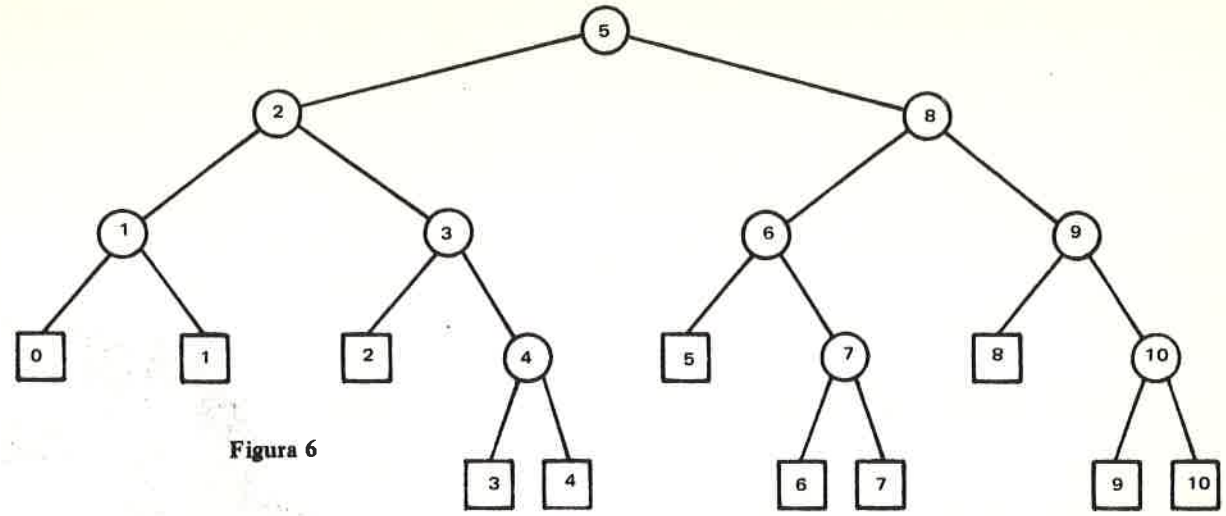


Figura 6

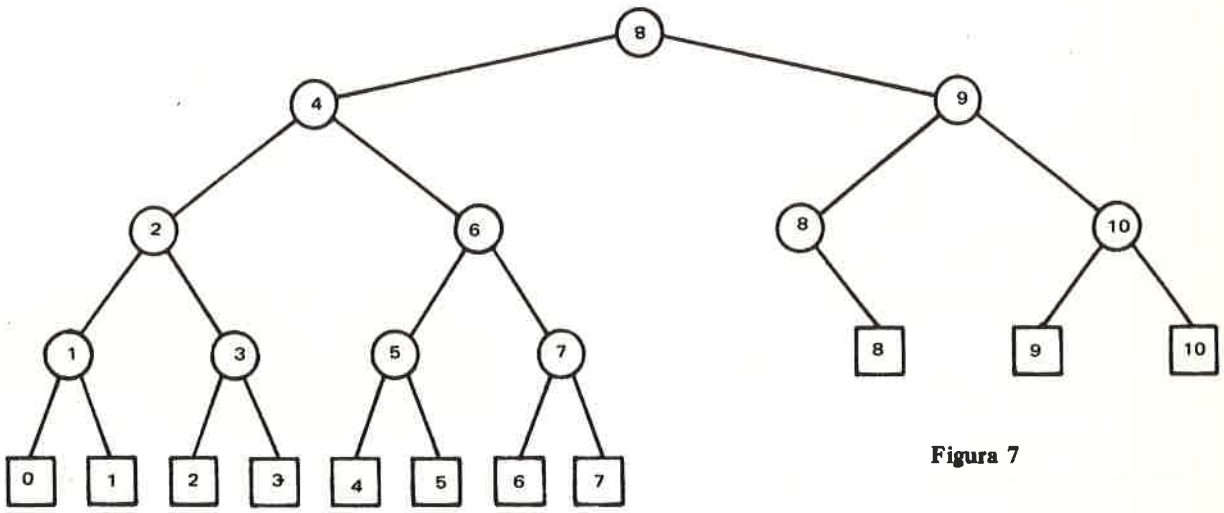


Figura 7

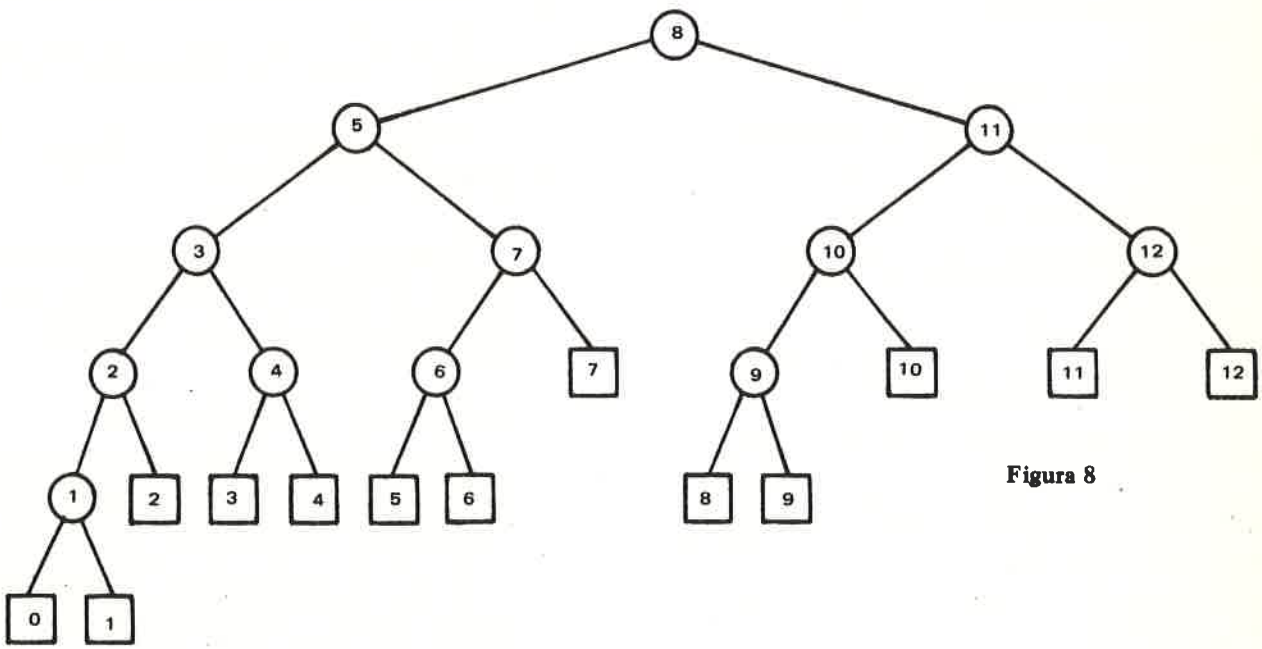


Figura 8

para

$$1 \leqslant m \leqslant \left[\log_2 K_{máx} + 2 \right]$$

El algoritmo sería:

1 i = DELTA (ł)
m = 2

2 hacer 4 y 5 hasta que K = Ki
o bien DELTA (m) = 0

3 si K = Ki éxito, si no fracaso

4 si K < Ki i = i – DELTA (m)
si no i = i + DELTA (m)

5 m = m + 1

El programa correspondiente es el ALGϕ91.

Una nueva modificación del algoritmo de búsqueda binaria es el método de Shar.

Si comparamos los árboles de las figuras 4 y 6 correspondientes a búsquedas binarias en tablas de 16 y 10 elementos respectivamente vemos que la primera es un árbol prácticamente simétrico (como lo sería siempre que $K_{máx}$ fuera una potencia exata de 2) mientras que el otro no.

El algoritmo de Shar se propone eliminar esta distorsión.

Previamente hemos de calcular el valor K_{SHAR} que es la máxima potencia entera de 2 inferior a $K_{máx}$.

Si K_{SHAR} fuera igual a $K_{máx}$ o bien $K \leqslant K_{SHAR}$ efectuaríamos una búsqueda binaria uniforme normal, utilizando K_{SHAR} en lugar de $K_{máx}$; en caso contrario haríamos una búsqueda uniforme en la subtabla comprendida entre K_{SHAR} y $K_{máx}$.

El programa es el ALG1ϕ1 y el árbol el de la figura 7.

El último método que veremos en este apartado es el de Fibonacci.

La idea de utilizar los números de Fibonacci puede parecer peregrina si nos fijamos únicamente en el árbol que da origen al algoritmo que veremos: en efecto, se trata (figura 8) de un árbol desequilibrado y desde luego distante del óptimo que es el árbol binario simétrico. Sin embargo la ventaja del método está no en disminuir el "camino medio" sino en el coste de dicho camino, cosa que se logra sustituyendo las divisiones que van asociadas a cada paso en los algoritmos anteriores por restas.

Será por lo tanto especialmente útil este método en los ordenadores en los que la división está fuertemente penalizada en cuanto a tiempo de CPU.

La sucesión de los números de Fibonacci es la siguiente:

0, 1, 1, 2, 3, 5, 8, 13, 21, etc...

Los números están sujetos a la relación de recurrencia

$$F_k = F_{k-1} + F_{k-2}$$

A los números de Fibonacci les podemos asociar los árboles de Fibonacci definidos de forma recurrente así:

Un árbol de Fibonacci de orden K tiene $F_{k+1}-1$ nodos internos y F_{k+1} nodos terminales y está construido de la siguiente manera:

Si K = 0 ó K = 1 el árbol es 0

Si $K \geqslant 2$ la raíz es F_k, el subárbol izquierdo es el árbol de Fibonacci de orden K-1 y el subárbol de la derecha el árbol de orden K-2 con todos sus números incrementados en F_k.

Dado pues el árbol hemos de encontrar el algoritmo que lo desarrolle.

Fijémonos en la figura 8. Para cualquier nodo no terminal se cumple que los números de los 2 hijos difieren del n.º del padre en la misma cantidad, que es un n.º de Fibonacci.

Por ejemplo: $5 = 8 - F_4$

$$11 = 8 = F_4$$

Además si la diferencia es Fj, la diferencia en el nivel siguiente será F_{j-1} a la izquierda y F_{j-2} a la derecha.

Como el campo cubierto por un árbol de Fibonacci de orden K llega hasta $F_{k+1} - 1$ supondremos que el n.º de elementos de la tabla es inferior en 1 a un n.º de Fibonacci.

El algoritmo será el siguiente (suponemos conocidos los números F_{k+1}, F_{k-1}, F_{k-2}).

1. Inicializar $i = F_k$ (raíz)
   $l = F_{k-1}$
   $m = F_{k-2}$
2. Comparar
   si K > Ki ir a 3
   si K < Ki ir a 4
   si K = Ki éxito
3. si m = O fracaso
   si no i = i − m
   (l, m) = (m, l-m)
   ir a 2
4. si l = 1 fracaso
   si no i = i + m
   l = l − m
   m = m − l

Nótese que así como en 3 (camino de la izquierda) nos limitamos a rebajar un grado todos los ns. de Fibonacci en 4 el proceso es más complicado ya que los números de los nodos en los subárboles de la derecha vienen aumentados por el valor de i.

l y *m* representan a lo largo de todo el algoritmo números de Fibonacci consecutivos.

El programa ALG111 desarrolla este algoritmo.

F_{k+1}, F_k, F_{k-1}, F_{k-2} son 4 ns de Fibonacci consecutivos, estando $K_{máx}$ comprendido entre los 2 primeros.

Para eliminar la restricción de ser $K_{máx} + 1 = F_{k+1}$ hemos empleado un sistema similar al del método de Shar.

Previamente al algoritmo indicado preguntaríamos si $K > K_{Fk}$; en caso negativo seguiríamos normalmente el algoritmo si no restaríamos de la diferencia entre $K_{máx}$ y $K_{Fk+1} + 1$ e iríamos directamente al punto 4.

2.1.2. Búsqueda en tablas encadenadas

En los apartados anteriores hemos visto el uso de las tablas lineales. Vimos en los últimos métodos cómo podíamos imaginar la existencia de una estructura implícita de árbol binario que sirviera de soporte a nuestra búsqueda. Vimos también que esa estructura de árbol nos proporcionaba el número mínimo de comparaciones para acceder a cualquier elemento de la tabla. Sin embargo todo ello es apropiado únicamente en tablas de longitud fija ya que tanto la inclusión como la exclusión de registros en la tabla son procesos largos que harían inoperante el método.

Para el caso de búsqueda dinámica es más indicado el uso de tablas encadenadas.

En una tabla encadenada cada elemento debe contener, además de la clave, un pointer al siguiente elemento de la tabla. Lógicamente todas las ventajas e inconvenientes que señalamos al hablar de las tablas lineales se invierten en este caso.

Centrándonos en nuestro problema, parece una buena táctica intentar adaptar el método binario, teóricamente el mejor, a la nueva estructura. Se trata en suma de explicitar la estructura de árbol binario que en forma implícita habíamos descubierto. Al querer explicitar la estructura de árbol binario, deberemos asociar a cada elemento de la tabla, además de su clave, dos elementos de encadenamiento que apunten a sus hijos (derecho e izquierdo).

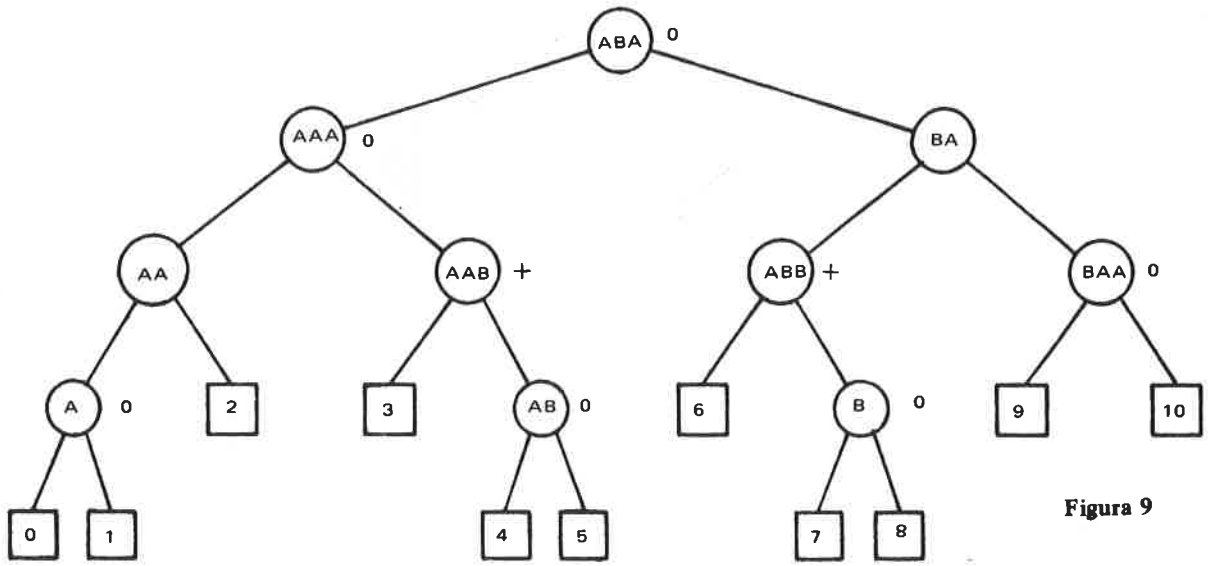


Figura 9

Veamos un ejemplo. Supongamos que las claves de que disponemos son A, AA, AAA, AAB, AB, ABA, ABB, B, BA, BAA, BAB, BB, BBA y BBB. Una posible estructura de árbol binario que contuviera las 10 primeras sería la de la figura 9. La figura 10 nos indica una posible estructura de la tabla asociada a dicho árbol.

	Elemento	Clave	Hijos der.	Hijos izq.
RAIZ →	1	ABA	3	2
	2	AAA	5	4
	3	BA	7	6
	4	AA	0	8
	5	AAB	9	0
	6	ABB	10	0
	7	BAA	0	0
	8	A	0	0
	9	AB	0	0
	10	B	0	0
SIGUIENTE ELEMENTO LIBRE →	11			

Figura 10

Si quisiéramos buscar la clave B el camino sería:

B > ABA derecha
B < BA izquierda
B > ABB derecha
B = B éxito

Si quisiéramos introducir la clave BB el camino sería:

BB > ABA derecha
BB > BA "
BB > BAA "

Debiendo la clave BB sustituir al nodo terminal 10

El algoritmo sería el siguiente

1. Colocarse en la raíz del árbol

2. Seguir el árbol (yendo por la rama derecha o izquierda según que la clave sea > ó < que el nodo) hasta que encontremos la clave o bien el pointer al hijo correspondiente sea nulo.

3. En el primer caso la búsqueda sería positiva y en el 2.º no y deberíamos insertar la nueva clave.

El programa ALG171 realiza este algoritmo

(TO-POIN es el pointer al hijo derecho
TO-CAB es " " izquierdo
K-LIBR nos da el siguiente elemento libre)

Naturalmente todas las ventajas de este método desaparecen si no es preciso hacer ampliaciones de la tabla ya que en ese caso cualquiera de los métodos binarios es más eficiente.

, Fijémonos con un poco de detalle en la estructura del árbol de la figura 9. La primera cosa que aparece a la vista es que para un conjunto cualquiera de claves el árbol no es único. Depende del orden en que hayan entrado las claves. Esto puede ser un grave problema ya que si las claves entraran ordenadas se llegaría a una estructura de árbol completamente lineal (árbol degenerado). Ello no conviene en absoluto ya que, como hemos visto, el n.º de comparaciones en dicho caso sería $\frac{K_{máx}}{2}$ frente a $\log_2 K_{máx}$ del árbol binario simétrico.

Afortunadamente resulta que si las claves entran al azar, el árbol es casi simétrico y el n.º de comparaciones es $1{,}386 \log_2 K_{máx}$.

Sin embargo no siempre es posible disponer de las claves a nuestra conveniencia y, en cualquier caso, sería deseable algún mecanismo de control que nos mantuviese el árbol dentro de unos límites de degeneración fijados por nosotros.

Existen, desde luego, algoritmos que partiendo de un árbol dado nos proporcionan el árbol equivalente óptimo. Existen también algoritmos que crean árboles óptimos teniendo en cuenta frecuencias estimadas de aparición de las diversas claves. No nos detendremos en estas sofistificaciones que se apartan un poco de la línea de este trabajo. Abordaremos en cambio un tema paralelo a éstos que es el de los árboles equilibrados.

Decimos que un árbol es equilibrado si, para cada nodo, la longitud de su subárbol derecho difiere de la de su subárbol izquierdo en un máximo de una unidad. Podemos asociar a cada nodo un nuevo campo (balance factor – BF) que valga 0 ó ± 1 de acuerdo con la diferencia de longitudes de sus 2 subárboles. En la figura 9 están indicados los BF de todos los nodos. El árbol en dicha figura es equilibrado.

La introducción de una nueva clase BAB (en lugar de 10 cambiaría el BF de BBA de 0 a +1 y el de BA de –1 a 0 pero no modificaría la situación de equilibrio del árbol. En cambio si introducimos después la clave BB el árbol se desequilibraría y sería necesario un nuevo reajuste. La figura 11 muestra la parte derecha del árbol 9 tras el proceso de reajuste (la parte izquierda no cambiaría).

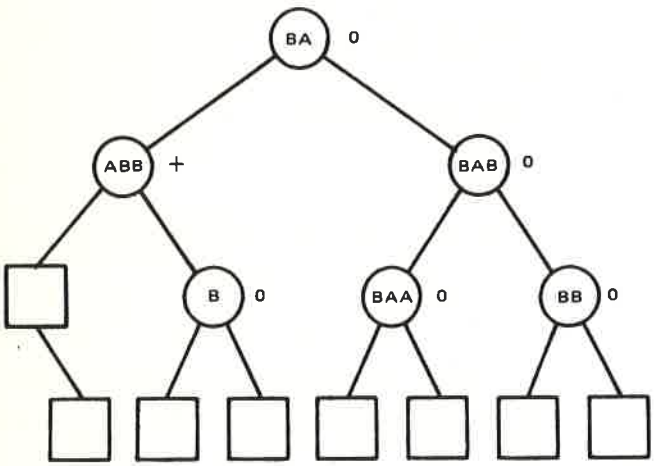


Figura 11

El problema, en general, se produce cuando tenemos un nodo con BF + 1 (–1) cuyo subárbol derecho (izquierdo) se hace más largo tras la inserción. Podemos reducir el problema a 2 casos (4 si contamos los simétricos). La figura 12 nos muestra los 2 casos posibles (a) y (b) así como el resultado del nuevo equilibrio (c) y (d). El primer caso (rotación simple) se soluciona con una rotación a la izquierda de los nodos A, B. El segundo (doble rotación) exige una rotación a la derecha de X, B y luego una a la izquierda de A y X.

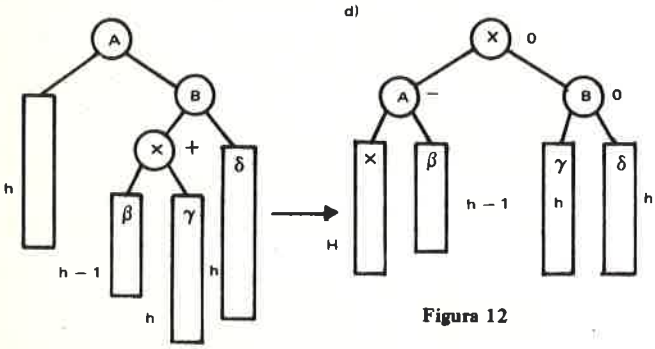
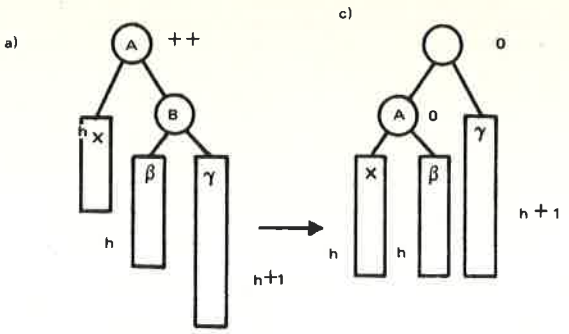


Figura 12



El algoritmo es complejo y en la figura 13 damos las líneas generales.

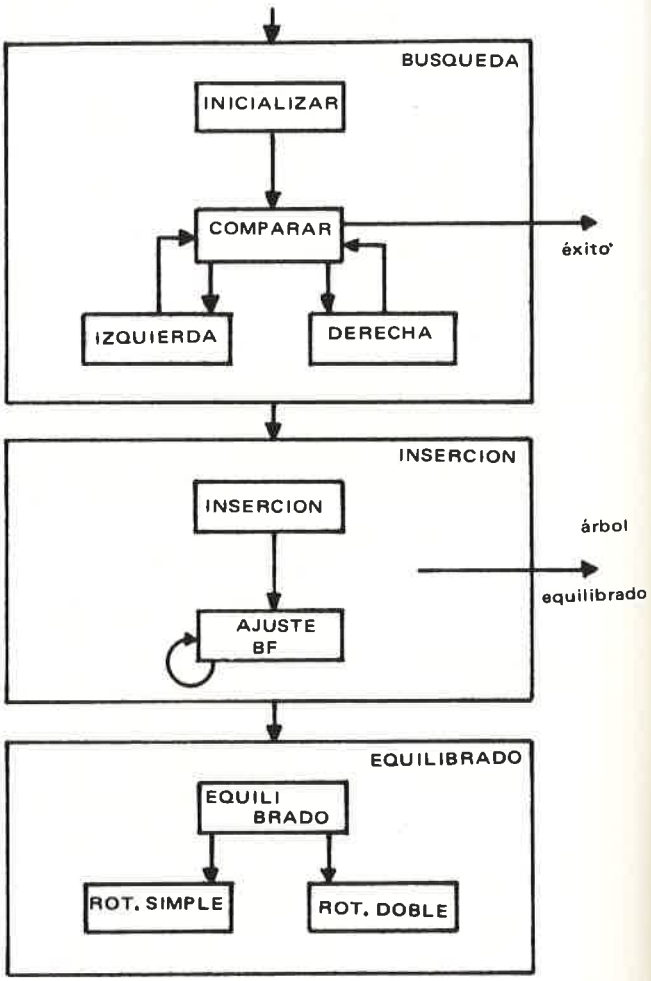


Figura 13

El programa ALG181 lo desarrolla.

Como en este caso la raíz del árbol no es fija supondremos que el primer elemento de la tabla contiene un pointer a dicha raíz. *P* se mueve a lo largo del árbol, *Q* es su hijo correspondiente a la rama adecuada. *S* apunta al nodo donde puede ser necesario empezar a reequilibrar. *T* apunta al padre de *S*. TO-BAL es la BF de cada nodo.

ALG182 realiza la búsqueda a lo largo del árbol.
ALG183 realiza la inserción de un nuevo elemento.
ALG184 realiza el reajuste de BF.
ALG185 rotación simple.
ALG186 rotación doble.
ALG187 completa en ambos casos el reequilibrado.

Vemos que se trata de un algoritmo largo y complicado y hemos de ponderar cuidadosamente la conveniencia de utilizarlo ya que como antes indicamos en la mayoría

de los casos los árboles que se crean difieren poco del simétrico.

2.2. Métodos de búsqueda digital

Un segundo grupo en la clasificación que hemos hecho de los métodos de búsqueda lo constituyen los métodos digitales.

La idea en que se basan estos métodos es sencilla y similar a la que emplearíamos manualmente en la consulta de un diccionario o de una guía de teléfonos: dividiríamos la clave en dígitos, consultaríamos el primero, lo que nos seleccionaría una subtabla, luego el 2.º, etc...

Métodos de este estilo se usan abundantemente en búsquedas en dispositivos externos (sobre todo combinados con alguno de los que hemos visto) pero se usan rara vez en búsquedas en memoria. Nos ocuparemos, por lo tanto, poco de ellos, limitándonos a exponer los 2 principales: el TRIE (de reTRIEval) y el método del árbol binario digital.

El método TRIE parte la clave a nivel de octetos. La tabla es ahora un árbol N-ario (siendo N el n.º de caracteres que pueden formar parte de la clave: 10 si son números, 26 si letras, etc..., además si las claves fueran de diferente longitud necesitaríamos una nueva dimensión que sería el espacio o cualquier otro elemento delimitador). Los elementos del árbol serían claves o bien pointers a otros elementos (quizás fuera adecuado hablar de árbol vectorial de N dimensiones). La figura 14 nos muestra un árbol de este tipo de dimensión 25. La tabla contiene 24 claves correspondientes a 24 palabras. Si quisiéramos localizar la palabra QUE seleccionaríamos la 1a. letra *Q*, buscaríamos la q-ésima componente de la raíz del árbol (1) encontraríamos QUE y daríamos por buena la búsqueda. La localización de POR sería así: (P, 1) nos enviaría al nodo 11; (0, 11) encontraría POR, si buscáramos la palabra UNO la secuencia sería (N, 1) → (N, 9) → (0, 10) y la búsqueda habría fracasado.

El algoritmo de búsqueda es sencillo:

Nos situamos en la raíz del árbol (nodo 1) en la componente definida por la 1a. letra de la clave. Vamos recorriendo el árbol situándonos cada vez en el nodo definido por el pointer encontrado en el nodo anterior, en la componente determinada por la letra correspondiente. El algoritmo acaba cuando en un nodo encontramos una clave o un espacio.

Hagamos unas cuantas consideraciones:

1 – El espacio de memoria es muy superior en este método al de cualquier otro.

2 – El acceso a una fila determinada (excepto en el caso de clave numérica) no es inmediato. En principio deberíamos pensar en algún algoritmo

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
espacio			DE		LA	LO		A		UN					
A	8			5						UNA	PARA	MAS			
B															
C															
D															
E	7											ME			
F															
G															
H															
I															
J															
K															
L	4		DEL				EL	AL							
M	12													COMO	
N	NO						EN		10					CON	
O	0			6							POR		14		
P	11														
Q	QUE														
R															
S	SE				LAS	LOS	ES								
T															
U	9														
V															
W															
X															
Y	Y														
Z															

Figura 14

de HASHING para realizarlo (lo que aumentaría el n.º de filas).

3 – Si comparamos la eficiencia del método en cuanto a tiempo medio de búsqueda con el método del árbol vemos que en caso de búsqueda con éxito los tiempos son similares y en cambio es menor en caso de búsqueda infructuosa.

4 – Las primeras columnas son las que presentan menos huecos por lo que cabe pensar en diseñar métodos de búsqueda en los que empleemos TRIE para los 2 ó 3 primeros dígitos y luego pasáramos a otros métodos.

El 2.º método digital que estudiaremos es una variante del método del árbol binario. En este caso la partición de la clave se hace a nivel de bits. La estructura que resulta es similar a la del árbol binario y en este caso la elección del camino se hace a través de un bit (p.e. 1 a la derecha y 0 a la izquierda). A medida que descendemos en el árbol vamos desplazando un pointer a través de la clave para señalar el bit que debemos tomar en consideración.

Es difícil, a priori, señalar cuál de los dos métodos, el del árbol binario o el digital es mejor. Desde luego el evitar las comparaciones hace que en este caso el camino a través del árbol sea más rápido. Sin embargo el método produce árboles más degenerados que en el caso anterior ya que allí sólo jugaban las similitudes o diferencias entre claves y aquí juegan también las estructuras en bits de los diversos caracteres.

Un problema adicional es que la programación a nivel de bits es difícil y costosa de realizar en muchos lenguajes (entre ellos el COBOL).

2.3. Hashing

Bajo este nombre englobamos todos los métodos basados en una transformación, normalmente de tipo algebraico, de la clave. Decimos basados porque la transformación es un elemento esencial al método. En cualquiera de los métodos vistos hasta ahora se pueden incluir transformaciones auxiliares en las claves (de cualquier tipo: compactaciones, empaquetado, desplazamientos, eliminación de dígitos no significativos) sin que podamos hablar de Hashing.

El Hashing consiste en aplicar al conjunto de claves una función, la función de hash, dando por resultado una secuencia de números que constituyan un conjunto más pequeño y manejable que el original.

Lo ideal sería que, si tenemos N claves, la función de hash nos produjera N números naturales del 1 al N. Como este ideal es muy difícil de conseguir, se producirán normalmente una serie de duplicidades en las claves transformadas. De la elección de la función de hash depende que estas duplicidades sean pocas. Desde luego si el porcentaje de duplicidad fuera grande deberíamos desechar el método ya que difícilmente podría competir con los hasta ahora vistos.

En un problema de hashing existen dos partes completamente independientes:

1. Elección de la función de Hash
2. Métodos de resolución de las duplicidades.

Veámoslas con cierto detalle:

La elección de una función de hash correcta depende de varios factores, principalmente de tres:

1. estructura de los datos
2. restricciones de memoria
3. características del equipo.

Respecto al primer factor hay que decir que un análisis de la estructura de las claves es fundamental. Pensemos que una clave alfanumérica de 10 posiciones (= 80 bits) puede adquirir 280-1 formas diferentes. Actuar sin más equivale a suponer que todas ellas son igualmente probables y va a dar lugar a una función de hash seguramente inadecuada. Cualquier restricción, de cualquier tipo, que impongamos nos va a simplificar el problema.

El 2.º factor es también fundamental. Cuanto mayores sean nuestros problemas de memoria más habremos de afinar en la elección de la función. Es evidente que al ampliar el intervalo de variación de las claves transformadas disminuiremos la posibilidad de duplicidades.

En cuanto al 3er. factor baste recordar que la función de hash suele ser una función complicada o por lo menos costosa de calcular. Adoptar funciones que se avengan con los tiempos de CPU de la máquina nos reducirá indudablemente el tiempo de búsqueda.

Parece pues evidente que es imposible dar criterios de uso general en la adopción de funciones de hash. La fórmula más extendida parte de considerar igualmente probables todas las combinaciones de bits que configuran las claves y calcular para cada una de estas (consideradas como número binario) el resto módulo M. Ello, si las claves están elegidas al azar, nos aseguraría una partición del conjunto de claves en M subconjuntos. Si M es grande (en cualquier caso > N, número de claves), tenemos bastantes probabilidades de que las duplicidades sean escasas.

Vamos ahora a introducirnos en el 2.º punto: resolución de duplicidades. Conviene insistir en un detalle para tener centrado el problema: las duplicidades han de ser relativamente pocas, si no es así el hashing pierde su sentido.

En lo que sigue vamos a suponer que tenemos *N* claves y que la función de hash las convierte en números comprendidos entre 1 y M.

El primer método, el más intuitivo, es el de encadenamiento. Son necesarias 2 tablas: una lineal que contiene pointers a una encadenada que contiene las claves (figura 15).

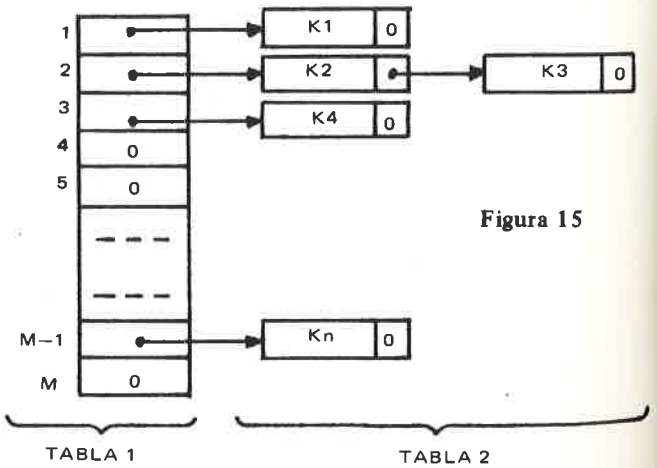


Figura 15

El algoritmo sería el siguiente:

1. Dada K calcular la función de hash h (k)
2. Consultar el elemento h (k) de la tabla 1; si es cero insertar la clave; si no, hacer el paso 3.
3. Recorrer la tabla 2 hasta encontrar la clave (éxito en la búsqueda) o encontrar un pointer nulo (fracaso, inserción de la clave).

Lógicamente el método usa *N* claves y *M* + *N* pointers.

El programa es el ALG121.

Una variación de este método nos permite operar con sólo *M* claves y *M* pointers (figura 16).

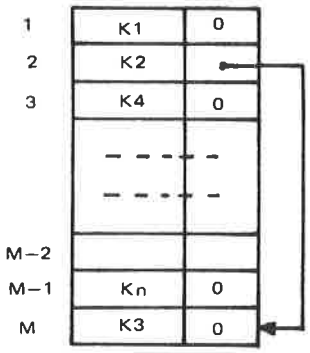


Figura 16

En este caso disponemos de una sola tabla de longitud M. Cada elemento de la tabla contiene una clave y un pointer.

El algoritmo sería similar al anterior:

1. Dada K calcular la función de hash h (k)
2. Consultar la clave correspondiente al elemento h (k) de la tabla. Si es una clave vacía insertaremos la clave K; en caso contrario haremos el paso 3.
3. Recorrer la tabla hasta encontrar la clave (éxito) o bien un pointer nulo. En este último caso deberemos introducir la clave en la 1a. posición libre que encontremos desde el final de la tabla hacia atrás.

El programa es el ALG131.

Un nuevo paso adelante son los métodos de direccionamiento abierto. En ellos ya no necesitamos pointers. Si la clave no está en la posición h (k) buscamos secuencialmente hacia atrás, a partir de la posición anterior a h (k) hasta encontrar la clave o bien un elemento vacío (suponemos que la tabla es circular y que el anterior al primer elemento es el último). Este método supone un ahorro considerable de memoria respecto a los anteriores, pero empieza a ser ineficaz a medida que la tabla se llena (ver ALG 141).

Para solucionar este problema podemos recurrir (ALG151) al método de doble hashing:

El método es similar al anterior excepto en que la búsqueda secuencial hacia atrás en vez de hacerla de 1 en 1 se hace de h'(k) en h'(k), siendo h'(k) una nueva función de hash.

La única condición que imponemos a h'(k) es que nos cubra todo el campo de valores de la tabla 1 a M sin volver a apuntarnos a h (k) (lo que produciría un loop). Si utilizamos el método de los restos basta tomar M y M' primos entre sí para que la condición se satisfaga.

El método distribuye de una forma más eficaz las claves duplicadas, a costa de un tiempo extra en el cálculo de h'(k).

El último método que expondremos es el de Brent (ALG161).

La idea en que se basa es que normalmente es más común la búsqueda con éxito que la inserción y que vale la pena gastar un tiempo extra en el caso de inserción a fin de reducir el tiempo medio de búsqueda.

Supongamos que se han probado sin éxito las posiciones $P_0, P_1, \ldots P_{t-1}, P_t$ (el elemento P_t estaba vacío) donde

$$P_j = h(k) - j\,h'(k)$$

El siguiente paso sería incluir la clave k en la posición P_t. Ahora bien ¿qué ocurriría si la pusiéramos en la posición P_0? Pues que entonces la clave que había en P_0 (K_O) deberíamos trasladarla a la 1a. posición libre p_0^j tal que $p_0^j = p_o - jh'(K_O)$. Esto será rentable evidentemente en el caso de que la longitud a recorrer en este caso (j) fuera inferior a la primitiva (t).

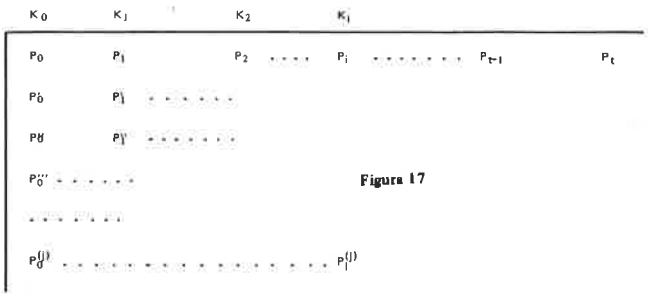


Figura 17

Igualmente podríamos hacernos la pregunta respecto a $P_1, P_2, \ldots$, etc. y finalmente podríamos enunciar la condición en forma general: es rentable colocar la clave K en la posición P_i (en vez de en P_t) y subsiguientemente la K_i en la posición P_i^j cuando $P_i^j = 0$ siendo $i + j < t$.

Con este método de Brent concluimos la exposición de los métodos de hashing, y con ellos los métodos de búsqueda. En el siguiente apartado haremos una somera valoración de los resultados obtenidos.

3. EVALUACION DE LOS METODOS

Todos los algoritmos que hemos expuesto fueron programados en COBOL para comparar su eficiencia. Algunos de los programas han sido comentados ya al exponer el método correspondiente. Los resultados se han sumarizado en la figura 18. En los algoritmos que admitían inserción se han insertado todas las claves no encontradas. Las búsquedas se han efectuado en tablas de 10, 50, 100 y 500 elementos. En cada caso se han buscado todos los elementos de la tabla. Los tiempos indicados (en mseg) son medios. Las claves se han elegido al azar de un conjunto formado por todas las palabras (truncadas a 10 letras) de un artículo de Novática (no pretendemos darle mayor generalidad de la que tiene). Los resultados son indicativos.

En la última columna indicamos las necesidades de memoria de cada método. Nos limitamos a dar las necesidades en cuanto a tablas. No tenemos en cuenta las diferencias de tamaño de los programas ni el uso de variables auxiliares, índices, etc. cuyo número varía de un método a otro. Los datos que damos son pues sólo aproximados:

C = longitud de la clave
P = longitud de los pointers
N = número de elementos de la tabla
M = amplitud de hashing.

Los tiempos de búsqueda que indicamos en la figura nos pueden servir para comparar entre sí los diferentes métodos comprendidos en cada tipo de búsqueda pero deben utilizarse con sumo cuidado al comparar los tipos entre sí. Debemos tener en cuenta que a partir del algoritmo 12 los tiempos indican no sólo consultas sino actualizaciones, que en los algoritmos de hashing se ha tomado una función de hash especialmente mala a fin de que, produciéndose muchas duplicidades, la comparación entre los métodos fuera más efectiva.

Hemos incluido también las 2 formas de búsqueda incorporadas al COBOL para que nos sirvan de orientación.

De entrada podemos establecer 2 grandes campos de aplicación: búsquedas estáticas en las que aplicaríamos los métodos lineales y binarios y búsquedas dinámicas en las que emplearíamos métodos del árbol. Los métodos de Hashing podrían seguramente competir en los 2 campos.

Los métodos lineales presentan ventajas para un n.º de elementos pequeño, a partir de 50 ya son preferibles los binarios.

De los 3 métodos de búsqueda lineal en tabla no ordenada podemos rechazar inmediatamente los 2 primeros ya que el tercero es mejor en todos los casos.

En líneas generales es siempre más efectiva la búsqueda en tablas ordenadas (excepto para un n.º pequeño de elementos en cuyo caso quizás no se compense el tiempo destinado a ordenar la tabla). De todas maneras si en una tabla no ordenada pusiéramos en primer lugar las claves más frecuentes nos mejoraría algo el tiempo medio de búsqueda.

En el campo de las búsquedas binarias podemos rechazar de entrada los métodos 8 y 10. El método 11 (Fibonacci) es el mejor en todos los casos. El 9 es ligeramente superior al 7 a expensas de un pequeño aumento en la ocupación de memoria.

Los métodos del árbol son competitivos en un espectro grande de dimensiones de la tabla. El caso de tabla no ordenada el método 18 tarda un 30 % más que el 17. Sin embargo en caso de tabla ordenada, mientras el 18 (árbol equilibrado) mantiene sus promedios el 17 cae rápidamente a niveles inaceptables (árbol degenerado).

En cuanto a métodos de Hashing podemos rechazar el de direccionamiento abierto y el de Brent. El mejor para tablas grandes es el de doble hashing mientras que para tablas pequeñas van bien los 2 de encadenamiento.

De todas maneras los datos en cuanto a hashing son sólo indicativos ya que una elección de M diferente los puede hacer variar considerablemente.

En cualquier caso lo anterior es sólo un esbozo. Un estudio previo de la naturaleza de los datos, frecuencias, etc... nos debe permitir en cada caso la elección del método más adecuado.

Horacio Rodriguez Hontoria

algoritmo	descripción	tabla ordenada				tabla no ordenada				ocupación de memoria
		10	50	100	500	10	50	100	500	
1	SEARCH	2.2	6	10.6	47.8					N X C
2	SEARCH ALL	4	5	5.8	7.4					N X C
3	búsqueda lineal 1					2.4	6.2	11.2	50.2	N X C
4	búsqueda lineal 2					2.4	4.6	7.6	31.6	(N + 1) X C
5	búsqueda lineal 3					2.4	4.4	6.8	27	(N + 1) X C
6	lineal ordenada	2.4	4.8	7.8	31.6					(N + 1) X C
7	binaria	2.8	4.4	5	6.2					N X C
8	binaria uniforme	4.2	6.4	7.4	9.8					N X C
9	b. unif. con tabla	2.8	4.2	4.8	6.4					N X C + ($\log_2$ N + 2) X P
10	Shar	4	5.8	6.8	9					N X C
11	Fibonacci	2.6	3.2	4	4.8					N X C
12	Hashing encadenamiento	32.8	11.8	8.6	10.6	40	11.6	8.6	10.4	N X C + (N + M) X P
13	H. encaden. 2	31.6	11.6	8.6	10.6	39.6	11.4	8.6	10.6	M X (C + P)
14	H. direccionam. abierto	31.8	14.4	17	64	39.6	14.8	17.4	64.2	M X C
15	Doble hashing	33.6	12.4	9.2	7.4	39.6	12	9	7	M X C
16	Brent	32.8	12.4	10.2	11.8	39.6	12.8	10	10	M X C
17	Arbol	34	23.4	33.2	139	39.8	12.4	9.8	8.2	N X (C + 2P)
18	Arbol equilibrado	35	16	13	12	42.6	15.4	12.4	11.2	N X (C + 2P + 1)

Figura 18.

```
*****************************************************************
ALG011.
    SET K TO 1.
    SEARCH T-OBJE VARYING K
           AT END MOVE ZERO TO ENCONTRADO
           WHEN  TO-CLA (K) = CLAVE
           SET ENCONTRADO TO K.
*****************************************************************
ALG021.
    SEARCH ALL T-OBJE AT END MOVE ZERO TO ENCONTRADO
                      WHEN TO-CLA (K) = CLAVE
                      SET ENCONTRADO  TO K.
*****************************************************************
ALG031.
    PERFORM ALG032 VARYING K FROM 1 BY 1 UNTIL
            K   K-MAX OR  TO-CLA (K) = CLAVE.
    IF K   K-MAX
           MOVE ZERO TO ENCONTRADO
       ELSE  SET  ENCONTRADO  TO K.
ALG032.
    EXIT.
*****************************************************************
ALG041.
    SET K TO K-MAX.
    MOVE TO-CLA (K + 1) TO GUAR-CLA.
    MOVE CLAVE TO TO-CLA (K + 1).
    PERFORM ALG042 VARYING K FROM 1 BY 1 UNTIL
            TO-CLA (K) = CLAVE.
    IF K   K-MAX
           MOVE ZERO TO ENCONTRADO
       ELSE  SET ENCONTRADO  TO K.
    SET  K  TO K-MAX.
    MOVE GUAR-CLA  TO  TO-CLA (K + 1).
ALG042.
    EXIT.
*****************************************************************
```

```
*****************************************************************
ALG051.
    SET K TO K-MAX.
    MOVE TO-CLA (K + 1) TO GUAR-CLA.
    MOVE CLAVE TO TO-CLA (K + 1).
    PERFORM ALG052 VARYING K FROM 1 BY 2 UNTIL
            CLAVE = TO-CLA (K) OR TO-CLA (K + 1).
    IF  TO-CLA (K) = CLAVE
            NEXT SENTENCE
        ELSE SET  K  UP BY 1.
    IF  K      K-MAX
            MOVE 0 TO ENCONTRADO
        ELSE SET ENCONTRADO TO K.
    SET K TO K-MAX.
    MOVE GUAR-CLA TO TO-CLA (K + 1).
ALG052.
    EXIT.
*****************************************************************
ALG061.
    SET K TO K-MAX.
    MOVE TO-CLA (K + 1) TO GUAR-CLA.
    MOVE HIGH-VALUE TO TO-CLA (K + 1).
    PERFORM  ALG062 VARYING K  FROM 1 BY 1 UNTIL
            CLAVE NOT   TO-CLA (K).
    IF TO-CLA (K) = CLAVE
            SET ENCONTRADO  TO K
        ELSE MOVE ZERO TO ENCONTRADO.
    SET  K  TO K-MAX.
    MOVE GUAR-CLA  TO  TO-CLA  (K + 1).
ALG062.
    EXIT.
*****************************************************************
ALG071.
    MOVE 1 TO L.
    MOVE K-MAX TO M.
    MOVE 1 TO I.
```

```
     PERFORM ALG072 UNTIL TO-CLA (I) = CLAVE OR M ) L.
     IF TO-CLA (I) = CLAVE
          MOVE I TO ENCONTRADO
        ELSE MOVE 0 TO ENCONTRADO.
 ALG072.
     COMPUTE I =  (L + M) R 2.
     IF CLAVE  ) TO-CLA (I)
               SUBTRACT 1 FROM I GIVING M
          ELSE ADD  1 TO I GIVING L.


*****************************************************************
 ALG081.
     COMPUTE  I =  K-MAX R 2 +  0,5.
     COMPUTE  M =  K-MAX R 2.
     PERFORM  ALG082 UNTIL  CLAVE = TO-CLA (I)  OR  M =  0.
     IF  TO-CLA (I)  = CLAVE
               MOVE I TO ENCONTRADO
          ELSE MOVE 0 TO ENCONTRADO.
 ALG082.
     IF TO-CLA (I)  ) CLAVE
               COMPUTE   I =  I + (M R 2 + 0,5)
          ELSE COMPUTE   I =  I - (M R 2 + 0,5).
     DIVIDE 2 INTO M.
*****************************************************************
 ALG091.
     MOVE DELTA (I)  TO  I.
     MOVE 2  TO M.
     PERFORM ALG092 UNTIL TO-CLA (I) = CLAVE
                       OR  DELTA (M)  =  0.
     IF TO-CLA (I) = CLAVE
               MOVE I TO ENCONTRADO
0         ELSE MOVE 0 TO ENCONTRADO.
 ALG092.
     IF CLAVE     TO-CLA (I)
               ADD DELTA (M) TO I
          ELSE SUBTRACT DELTA (M) FROM I.
     ADD 1 TO M.
*****************************************************************
 ALG101.
     MOVE K-SHAF TO I.
     IF CLAVE NOT   TO-CLA (I) OF K-SHAF = K-MAX
               COMPUTE I = K-SHAF R 2 + 0,5
               DIVIDE 2  INTO K-SHAF GIVING M
          ELSE COMPUTE I = (K-MAX + K-SHAF) R 2 + 0,5
               COMPUTE M = (K-MAX - K-SHAF) R 2.
     PERFORM ALG102 UNTIL TO-CLA (I) = CLAVE OR M = 0.
     IF TO-CLA (I) = CLAVE
               MOVE I TO ENCONTRADO
          ELSE MOVE 0 TO ENCONTRADO.
 ALG102.
     IF CLAVE    TO-CLA (I)
               COMPUTE  I =  (I + M R 2 + 0,5)
          ELSE COMPUTE  I =  (I - (M R 2 + 0,5)).
     DIVIDE 2 INTO M.
*****************************************************************
 ALG111.
     MOVE FK TO I.
     MOVE FK-1 TO L.
     MOVE FK-2 TO M.
     IF CLAVE   TO-CLA (I)
               COMPUTE I = I - FK1 + K-MAX + 1
               PERFORM ALG112.
     MOVE SPACE TO A.
     PERFORM ALG113 UNTIL CLAVE = TO-CLA (I) OR A = 'F'.
     IF CLAVE = TO-CLA (I)
               MOVE I TO ENCONTRADO
          ELSE MOVE 0 TO ENCONTRADO.
 ALG113.
     IF CLAVE   TO-CLA (I) PERFORM ALG112 ELSE
     IF M = 0 MOVE 'F' TO A
          ELSE SUBTRACT M FROM I
               MOVE M TO N
               SUBTRACT M FROM L GIVING M
               MOVE N TO L
 ALG112.
     IF L = 1 MOVE 'F' TO A
          ELSE ADD M TO I
               SUBTRACT M FROM L
               SUBTRACT L FROM M.
*****************************************************************


 ALG121.
     DIVIDE TA-10-NUM-HASH INTO CLAVE-HASH
               GIVING COCIENTE
               REMAINDER I.
     ADD 1 TO I.
     IF TO-CAB (I) = ZERO
               MOVE ZERO TO ENCONTRADO
                    PERFORM ALG122
                    MOVE K-HASH TO TO-CAB (I)
          ELSE MOVE TO-CAB (I) TO I
               PERFORM ALG123 UNTIL TO-POIN (I) = ZERO
                                 OR TO-CLA (I)=CLAVE
               IF TO-CLA (I) = CLAVE
                    MOVE I TO ENCONTRADO
             ELSE  MOVE  ZERO TO ENCONTRADO
                   PERFORM ALG122
                   MOVE K-HASH  TO  TO-POIN (I).
 ALG123.
     MOVE TO-POIN (I) TO I.
 ALG122.
     ADD 1 TO K-HASH.
     MOVE CLAVE TO TO-CLA (K-HASH).
     MOVE ZERO  TO TO-POIN (K-HASH).
*****************************************************************
 ALG131.
     DIVIDE TA-10-NUM-HASH INTO CLAVE-HASH
               GIVING COCIENTE
               REMAINDER  I.
     ADD 1 TO I.
     IF TO-CLA (I) = SPACE
               MOVE ZERO TO ENCONTRADO
                    MOVE CLAVE TO TO-CLA (I)
          ELSE PERFORM ALG132 UNTIL TO-CLA (I) = CLAVE
                               OR TO-POIN (I) = ZERO
               IF TO-CLA (I) = CLAVE
                    MOVE I TO ENCONTRADO
                  ELSE MOVE ZERO TO ENCONTRADO
                            PERFORM  ALG133 UNTIL TO-CLA (K-HASH)
                                                  = SPACE
                    MOVE K-HASH TO TO-POIN (I)
                    MOVE CLAVE TO  TO-CLA (K-HASH).
 ALG132.
     MOVE TO-POIN (I) TO I.
 ALG133.
     SUBTRACT 1 FROM K-HASH.
*****************************************************************
 ALG141.
     DIVIDE TA-10-NUM-HASH INTO CLAVE-HASH
               GIVING COCIENTE
               REMAINDER I.
     ADD 1 TO I.
     IF TO-CLA (I) = SPACE
               PERFORM ALG142
          ELSE PERFORM ALG143 UNTIL CLAVE  = TO-CLA (I)
                                 OR  TO-CLA (I) = SPACE
     IF TO-CLA (I) = CLAVE
               MOVE I  TO ENCONTRADO
                ELSE PERFORM  ALG142.
 ALG142.
     MOVE ZERO TO ENCONTRADO.
               MOVE CLAVE  TO TO-CLA (I).
 ALG143.
     SUBTRACT 1 FROM I.
     IF I = ZERO  MOVE K-HASH TO I.
*****************************************************************
 ALG151.
     DIVIDE TA-10-NUM-HASH INTO CLAVE-HASH
               GIVING COCIENTE
               REMAINDER I.
               ADD 1 TO I.
               IF TO-CLA (I) = SPACE
               PERFORM ALG152
          ELSE SUBTRACT  1 FROM K-HASH GIVING N
               DIVIDE N INTO CLAVE-HASH
                        GIVING COCIENTE
               REMAINDER X
               ADD 1 TO X
               PERFORM  ALG153  UNTIL TO-CLA (I) = CLAVE
                                  OR TO-CLA (I) = SPACE
               IF TO-CLA (I) = CLAVE
                    MOVE I TO ENCONTRADO
                    ELSE PERFORM ALG152.
 ALG152.
     MOVE  ZERO TO ENCONTRADO.
               MOVE CLAVE TO TO-CLA (I).
 ALG153.
     IF I   X SUBTRACT X FROM I
       ELSE  COMPUTE I = I + K-HASH - X.
*****************************************************************
 ALG161.
     DIVIDE TA-10-NUM-HASH INTO CLAVE-HASH
               GIVING COCIENTE
               REMAINDER  I.
     ADD 1 TO I.
     IF TO-CLA (I) = SPACE
               PERFORM ALG162
          ELSE  SUBTRACT 1 FROM K-HASH GIVING N
               DIVIDE N INTO CLAVE-HASH
                        GIVING COCIENTE
                        REMAINDER X
               ADD 1 TO X
               MOVE I TO Z
               MOVE ZERO TO P
               PERFORM ALG163 UNTIL TO-CLA (I) = CLAVE
                                 OR              SPACE
               IF TO-CLA (I) = CLAVE
                        MOVE I TO ENCONTRADO
                             ELSE PERFORM ALG164.
 ALG162.
     MOVE ZERO TO ENCONTRADO.
               MOVE CLAVE TO TO-CLA (I).
 ALG163.
     IF I    X SUBTRACT X FROM I
       ELSE  COMPUTE I = I + K-HASH - X.
     ADD 1 TO P.
 ALG164.
     MOVE ZERO TO S.
     MOVE ZERO TO Q.
     MOVE Z TO T.
     SET K TO Q.
     PERFORM ALG165 UNTIL TO-CLA (K) = SPACE OR S NOT ) (P - 1).
     IF TO-CLA (K) = SPACE
               MOVE TO-CLA (Z) TO TO-CLA (K)
               MOVE CLAVE  TO  TO-CLA (Z)
               MOVE ZERO TO ENCONTRADO
          ELSE  PERFORM  ALG162.
 ALG165.
     MOVE T  TO Z.
     SET K TO Z.
     MOVE TO-CLA (T) TO GUAR-CLA.
     DIVIDE N  INTO GUAR-CLA-HASH
               GIVING COCIENTE
               REMAINDER V.
     ADD 1 TO V.
     PERFORM ALG166 UNTIL TO-CLA (K) = SPACE OR S NOT ) (P - 1).
     ADD 1 TO Q.
     MOVE Q TO S.
     IF Z   X SUBTRACT X FROM Z GIVING T
       ELSE COMPUTE T = Z + K-HASH - X.
 ALG166.
     ADD 1 TO S.
     IF T   V SUBTRACT V FROM T
       ELSE  COMPUTE  T = T + K-HASH - V.
     SET K TO T.
               IF TO-CLA (I) = CLAVE
                        MOVE I TO ENCONTRADO
                             ELSE PERFORM ALG164.
 ALG162.
     MOVE ZERO TO ENCONTRADO.
               MOVE CLAVE TO TO-CLA (I).
 ALG163.
     IF I    X SUBTRACT X FROM I
       ELSE  COMPUTE I = I + K-HASH - X.
     ADD 1 TO P.
 ALG164.
     MOVE ZERO TO S.
     MOVE ZERO TO Q.
     MOVE Z TO T.
```

```
    SET K TO 0.
    PERFORM ALG165 UNTIL TO-CLA (K) = SPACE OR S NOT ) (P - 1).
    IF TO-CLA (K) = SPACE
            MOVE TO-CLA (Z) TO TO-CLA (K)
            MOVE CLAVE  TO  TO-CLA (Z)
            MOVE ZERO TO ENCONTRADO
       ELSE  PERFORM  ALG162.
ALG165.
    MOVE T  TO Z.
    SET K TO Z.
    MOVE TO-CLA (T) TO GUAR-CLA.
    DIVIDE N  INTO GUAR-CLA-HASH
            GIVING COCIENTE
            REMAINDER V.
    ADD 1 TO V.
    PERFORM ALG166 UNTIL TO-CLA (K) = SPACE OR S NOT ) (P - 1).
    ADD 1 TO Q.
    MOVE Q TO S.
    IF Z   X SUBTRACT X FROM Z GIVING T
       ELSE COMPUTE T = Z + K-HASH - X.
ALG166.
    ADD 1 TO S.
    IF T   V SUBTRACT V FROM T
       ELSE  COMPUTE  T = T + K-HASH - V.
    SET K TO T.
*****************************************************************
 ALG171.
     SET K TO 1.
     MOVE ZERO TO Y.
     PERFORM ALG172 UNTIL TO-CLA (K) = CLAVE.
     IF Y = 0  SET ENCONTRADO TO K
          ELSE  MOVE 0 TO ENCONTRADO.
 ALG172.
     IF CLAVE ) TO-CLA (K) AND TO-POIN (K) NOT = 0
              SET K TO TO-POIN (K)
     ELSE IF CLAVE   TO-CLA (K) AND TO-CAB (K) NOT = 0
              SET K TO TO-CAB (K)
          ELSE MOVE 1  TO Y
                  ADD 1 TO K-LIBR
                  MOVE CLAVE TO TO-CLA (K-LIBR)
                  MOVE 0 TO TO-CAB (K-LIBR)
                            TO-POIN (K-LIBR)
                  IF CLAVE ) TO-CLA (K)
                        MOVE K-LIBR TO TO-POIN (K)
                        SET K TO  K-LIBR
                   ELSE MOVE K-LIBR TO TO-CAB  (K)
                        SET K TO  K-LIBR.
*****************************************************************
 ALG181.
     MOVE ZERO TO N.
     MOVE 1 TO T.
     MOVE TO-POIN (1) TO S P.
     PERFORM ALG182 UNTIL CLAVE = TO-CLA (P).
     IF N = 0 MOVE P TO ENCONTRADO
        ELSE  MOVE 0 TO ENCONTRADO.
 ALG182.
     IF CLAVE ) TO-CLA (P)
           MOVE TO-CAB (P) TO Q
           IF Q = 0 ADD 1 TO K-LIBR
                    MOVE K-LIBR TO Q
                    MOVE Q TO TO-CAB (P)
                    PERFORM ALG183
              ELSE  IF TO-BAL (Q) NOT = 0
                              MOVE P TO T
                              MOVE Q TO S
                              MOVE Q TO P
                         ELSE MOVE Q TO P
       ELSE  MOVE TO-POIN (P) TO Q
             IF Q = 0 ADD 1 TO K-LIBR
                      MOVE K-LIBR TO Q
                      MOVE Q TO TO-POIN (P)
                      PERFORM ALG183
                ELSE  IF TO-BAL (Q) NOT = 0
                              MOVE P TO T
                              MOVE Q TO S
                              MOVE Q TO P
                         ELSE  MOVE Q TO P.
ALG183.
    MOVE 1 TO N.
    MOVE Q TO P.
    MOVE CLAVE TO TO-CLA (Q).
    MOVE ZERO  TO TO-CAB (Q)
                  TO-POIN (Q)
                  TO-BAL  (Q).
    IF CLAVE ) TO-CLA (S)
             MOVE  TO-CAB (S) TO F Y
        ELSE MOVE TO-POIN (S) TO F Y.
    PERFORM ALG184 UNTIL Y = 0.
    IF CLAVE )  TO-CLA (S)
             MOVE  -1  TO U
        ELSE MOVE  +1  TO U.
    IF TO-BAL (S) = 0
             MOVE U TO TO-BAL (S)
    ELSE IF TO-BAL (S)  NOT = U
             MOVE 0 TO TO-BAL (S)
    ELSE IF TO-BAL (F) = U
             PERFORM ALG185
        ELSE PERFORM ALG186.
ALG184.
    IF CLAVE ) TO-CLA (Y)
             MOVE -1 TO TO-BAL (Y)
             MOVE TO-CAB (Y) TO Y
        ELSE MOVE +1 TO TO-BAL (Y)
             MOVE TO-POIN (Y) TO Y.
ALG185.
    MOVE F TO Y.
    IF U = +1  MOVE TO-CAB (F) TO TO-POIN (S)
               MOVE S TO TO-CAB (F)
               MOVE 0  TO TO-BAL (S) TO-BAL (F)
          ELSE MOVE TO-POIN (F) TO TO-CAB (S)
               MOVE S  TO TO-POIN (F)
               MOVE 0  TO  TO-BAL (S) TO-BAL (F).
    PERFORM ALG187.
ALG186.
    IF U = +1 MOVE  TO-CAB (F) TO Y
              MOVE  TO-POIN (Y) TO TO-CAB (F)
              MOVE  F  TO  TO-POIN (Y)
              MOVE  TO-CAB (Y) TO  TO-POIN (S)
              MOVE  S          TO  TO-CAB (Y)
         ELSE MOVE  TO-POIN  (F) TO Y
              MOVE  TO-CAB (Y)  TO  TO-POIN (F)
              MOVE  F   TO TO-CAB (Y)
              MOVE  TO-POIN (Y)  TO TO-CAB (S)
              MOVE  S  TO TO-POIN (Y).
    IF TO-BAL (Y) = U
             MOVE 0 TO TO-BAL (F)
             COMPUTE TO-BAL (S) = 0 - U
        ELSE MOVE 0 TO TO-BAL (S)
             IF TO-BAL (Y) = 0
                MOVE  0 TO TO-BAL (F)
           ELSE  MOVE  U TO TO-BAL (F).
    MOVE 0 TO  TO-BAL  (Y).
    PERFORM  ALG187.
ALG187.
    IF S = TO-POIN (T)
             MOVE Y  TO TO-POIN (T)
        ELSE MOVE Y  TO TO-CAB  (T).
*****************************************************************
```

```
    SET K TO 0.
    PERFORM ALG165 UNTIL TO-CLA (K) = SPACE OR S NOT ) (P - 1).
    IF TO-CLA (K) = SPACE
            MOVE TO-CLA (Z) TO TO-CLA (K)
            MOVE CLAVE  TO  TO-CLA (Z)
            MOVE ZERO TO ENCONTRADO
       ELSE  PERFORM  ALG162.
ALG165.
    MOVE T  TO Z.
    SET K TO Z.
    MOVE TO-CLA (T) TO GUAR-CLA.
    DIVIDE N  INTO GUAR-CLA-HASH
            GIVING COCIENTE
            REMAINDER V.
    ADD 1 TO V.
    PERFORM ALG166 UNTIL TO-CLA (K) = SPACE OR S NOT ) (P - 1).
    ADD 1 TO Q.
    MOVE Q TO S.
    IF Z   X SUBTRACT X FROM Z GIVING T
      ELSE COMPUTE T = Z + K-HASH - X.
ALG166.
    ADD 1 TO S.
    IF T   V SUBTRACT V FROM T
      ELSE  COMPUTE  T = T + K-HASH - V.
    SET K TO T.
*****************************************************************
 ALG171.
     SET K TO 1.
     MOVE ZERO TO Y.
     PERFORM ALG172 UNTIL TO-CLA (K) = CLAVE.
     IF Y = 0  SET ENCONTRADO TO K
         ELSE  MOVE 0 TO ENCONTRADO.
 ALG172.
     IF CLAVE ) TO-CLA (K) AND TO-POIN (K) NOT = 0
             SET K TO TO-POIN (K)
     ELSE IF CLAVE   TO-CLA (K) AND TO-CAB (K) NOT = 0
             SET K TO TO-CAB (K)
          ELSE MOVE 1  TO Y
                  ADD 1 TO K-LIBR
                  MOVE CLAVE TO TO-CLA (K-LIBR)
                  MOVE 0 TO TO-CAB (K-LIBR)
                            TO-POIN (K-LIBR)
                  IF CLAVE ) TO-CLA (K)
                        MOVE K-LIBR TO TO-POIN (K)
                        SET K TO  K-LIBR
                   ELSE MOVE K-LIBR TO TO-CAB  (K)
                        SET K TO  K-LIBR.
*****************************************************************
 ALG181.
     MOVE ZERO TO N.
     MOVE 1 TO T.
     MOVE TO-POIN (1) TO S P.
     PERFORM ALG182 UNTIL CLAVE = TO-CLA (P).
     IF N = 0 MOVE P TO ENCONTRADO
        ELSE  MOVE 0 TO ENCONTRADO.
 ALG182.
     IF CLAVE ) TO-CLA (P)
           MOVE TO-CAB (P) TO Q
           IF Q = 0 ADD 1 TO K-LIBR
                    MOVE K-LIBR TO Q
                    MOVE Q TO TO-CAB (P)
                    PERFORM ALG183
              ELSE  IF TO-BAL (Q) NOT = 0
                           MOVE P TO T
                           MOVE Q TO S
                           MOVE Q TO P
                      ELSE MOVE Q TO P
       ELSE  MOVE TO-POIN (P) TO Q
             IF Q = 0 ADD 1 TO K-LIBR
                      MOVE K-LIBR TO Q
                      MOVE Q TO TO-POIN (P)
                      PERFORM ALG183
                ELSE  IF TO-BAL (Q) NOT = 0
                             MOVE P TO T
                             MOVE Q TO S
                             MOVE Q TO P
                        ELSE  MOVE Q TO P.
 ALG183.
     MOVE 1 TO N.
     MOVE Q TO P.
     MOVE CLAVE TO TO-CLA (Q).
     MOVE ZERO  TO TO-CAB (Q)
                   TO-POIN (Q)
                   TO-BAL  (Q).
     IF CLAVE ) TO-CLA (S)
              MOVE  TO-CAB (S) TO P Y
         ELSE MOVE TO-POIN (S) TO P Y.
     PERFORM ALG184 UNTIL Y = Q.
     IF CLAVE )  TO-CLA (S)
              MOVE  -1  TO U
         ELSE MOVE  +1  TO U.
     IF TO-BAL (S) = 0
              MOVE U TO TO-BAL (S)
     ELSE IF TO-BAL (S)  NOT = U
              MOVE 0 TO TO-BAL (S)
     ELSE IF TO-BAL (P) = U
              PERFORM ALG185
          ELSE PERFORM ALG186.
 ALG184.
     IF CLAVE ) TO-CLA (Y)
              MOVE -1 TO TO-BAL (Y)
              MOVE TO-CAB (Y) TO Y
          ELSE MOVE +1 TO TO-BAL (Y)
              MOVE TO-POIN (Y) TO Y.
 ALG185.
     MOVE P TO Y.
     IF U = +1  MOVE TO-CAB (P) TO TO-POIN (S)
                MOVE S TO TO-CAB (P)
                MOVE 0  TO TO-BAL (S) TO-BAL (P)
           ELSE MOVE TO-POIN (P) TO TO-CAB (S)
                MOVE S  TO TO-POIN (P)
                MOVE 0  TO  TO-BAL (S) TO-BAL (P).
     PERFORM ALG187.
 ALG186.
     IF U = +1 MOVE  TO-CAB (P) TO Y
               MOVE  TO-POIN (Y) TO TO-CAB (P)
               MOVE  P  TO  TO-POIN (Y)
               MOVE  TO-CAB (Y) TO  TO-POIN (S)
               MOVE  S          TO  TO-CAB (Y)
          ELSE MOVE  TO-POIN  (P) TO Y
               MOVE  TO-CAB (Y)  TO  TO-POIN (P)
               MOVE  P    TO TO-CAB (Y)
               MOVE  TO-POIN (Y)  TO TO-CAB (S)
               MOVE  S  TO TO-POIN (Y).
     IF TO-BAL (Y) = U
              MOVE 0 TO TO-BAL (P)
              COMPUTE TO-BAL (S) = 0 - U
         ELSE MOVE 0 TO TO-BAL (S)
              IF TO-BAL (Y) = 0
                    MOVE  U TO TO-BAL (P)
               ELSE  MOVE  U TO TO-BAL (P).
     MOVE 0 TO  TO-BAL  (Y).
     PERFORM  ALG187.
 ALG187.
     IF S = TO-POIN (T)
              MOVE Y  TO TO-POIN (T)
          ELSE MOVE Y  TO TO-CAB  (T).
*****************************************************************
```