CAPÍTULO

19 Tablas hash

En el Capítulo 18 hemos discutido los árboles binarios de búsqueda, los cuales permiten varias operaciones sobre un conjunto de elementos. Este capítulo discute las *tablas hash* o *tablas de localización*, que permiten únicamente un subconjunto de las operaciones permitidas por los árboles binarios de búsqueda. La implementación de las tablas hash se conoce usualmente como *hashing* o *localización*. La *localización* es la técnica empleada para realizar inserciones, eliminaciones y búsquedas con coste medio constante.

A diferencia de los árboles binarios de búsqueda, el tiempo medio de ejecución de las operaciones de las tablas hash se basa en propiedades estadísticas en lugar de en la espera de entradas aleatorias. Esta mejora se consigue perdiendo gran cantidad de información sobre el orden de los elementos; así, ciertas operaciones, como `buscarMin` o `buscarMax` y la impresión ordenada de la tabla, no pueden realizarse en tiempo lineal. Como consecuencia, la tabla hash y el árbol binario de búsqueda tienen aplicaciones diferentes y eficiencia distinta.

En este capítulo veremos:

- Varios métodos para implementar las tablas de localización.
- Algunas comparaciones analíticas entre dichos métodos.
- Diversas aplicaciones del hashing.
- Una comparación entre las tablas hash y los árboles binarios de búsqueda.

19.1 Ideas básicas

Las *tablas de localización* permiten consultar o eliminar cualquier elemento conociendo su nombre, de modo que lo que estamos implementando es un diccionario. Desearíamos ser capaces de ejecutar las operaciones básicas en tiempo constante, tal y como se ha conseguido en los casos de las pilas y de las colas. Debido a que el tipo de accesos está ahora mucho menos restringido, esto parece un objetivo imposible. Parece razonable que cuando el diccionario aumenta de tamaño, las búsquedas tardarán más tiempo, sin embargo, esto no tiene por qué ser necesariamente así.

La *tabla hash* se emplea para implementar un diccionario en el que cada operación se ejecuta en tiempo constante.

Supongamos, por ejemplo, que todos los elementos son enteros no negativos pequeños, entre 0 y 65.535. Entonces podemos emplear un vector, indexado entre 0 y 65.535, para implementar las operaciones como sigue. En primer lugar, inicializamos todas las casillas del vector `a` con 0. Para realizar `insertar(i)` ejecutamos `a[i]++`. Observe que `a[i]` representa el número de veces que ha sido insertado `i`. Para ejecutar `buscar(i)` comprobamos que `a[i]` no sea 0. Para realizar `eliminar(i)`, nos aseguramos de haber encontrado `i`, y si es así, ejecutamos `a[i]--`. Claramente, el tiempo de ejecución de cada operación es constante. Tenemos además el coste de la inicialización del vector, pero ésta supone una cantidad de trabajo constante (65.536 asignaciones), pues sólo se ejecuta una vez.

Esta solución tiene dos problemas. En primer lugar, supongamos que manejamos enteros de 32 bits en lugar de enteros de 16 bits. En tal caso, el vector `a` debería almacenar 4 billones de elementos, lo que es impracticable. En segundo lugar, si los elementos no son enteros sino cadenas de caracteres (o cualquier otra cosa más general), no podríamos emplear directamente enteros para indexar el vector.

El segundo problema no lo es realmente. Igual que el número `1234` es una secuencia de dígitos `(1, 2, 3, 4)`, la cadena `"hola"` es la secuencia formada por los caracteres `'h'`, `'o'`, `'l'` y `'a'`. Tenemos que el número `1234` se puede recuperar mediante el valor de $1 \cdot 10^3 + 2 \cdot 10^3 + 3 \cdot 10^1 + 4 \cdot 10^0$. Recuerde, de la Sección 12.1, que un carácter puede ser representado con 7 bits como un número entre 0 y 127. Como un carácter es básicamente un entero pequeño, podemos interpretar una cadena como un entero. Una representación posible es $\texttt{'h'} \cdot 128^3 + \texttt{'o'} \cdot 128^3 + \texttt{'l'} \cdot 128^1 + \texttt{'a'} \cdot 128^0$. Esto permitiría una implementación mediante un vector básico, como la que hemos visto anteriormente.

El problema que conlleva esta estrategia es que la representación entera descrita genera enteros muy grandes: la representación de `'hola'` es 224.229.227 y cadenas más largas generan representaciones mucho mayores. Esto nos conduce de nuevo al primer problema: ¿Cómo evitamos el uso de un vector absurdamente grande?

Una *función hash* convierte un elemento en un entero adecuado para indexar el vector en el que el elemento es almacenado. Si la función hash fuese inyectiva, podríamos acceder al elemento mediante su índice.

Esto se consigue usando una función que asocia a números grandes (o cadenas interpretadas como números) otros más pequeños y manejables. La función que asocia a un elemento un índice pequeño se conoce como *función hash* o *función de localización*. Si `x` es un entero (no negativo) arbitrario entonces `x%tamanyoTabla` genera un número entre 0 y `tamanyoTabla-1`, adecuado para indexar una posición de un vector de tamaño `tamanyoTabla`. Si `s` es una cadena, podemos convertir `s` en un entero grande `x`, empleando el método sugerido anteriormente y después aplicar el operador `%` para obtener un índice adecuado. Así, si `tamanyoTabla` es 10.000, `"hola"` se indexaría por 9.227. La Sección 19.2 discute con detalle la implementación de funciones de localización para cadenas de caracteres.

Como la función hash no es inyectiva, se producirá la colisión de muchos elementos potenciales en el mismo índice.

El empleo de la función de localización introduce una complicación: es posible que a dos elementos distintos les corresponda la misma posición. Esto no podemos evitarlo, ya que existen muchos más elementos potenciales que posiciones. Cuando esto sucede, se produce una *colisión*. Existen muchos métodos para resolver con rapidez una colisión. Estudiaremos tres de los más simples: *exploración lineal*, *exploración cuadrática* y *encadenamiento separado*. Todos ellos son sencillos de implementar, pero tienen una eficiencia distinta, dependiendo del grado de ocupación del vector.

19.2 Función de localización

Calcular la función de localización para cadenas tiene una sutil complicación: la conversión de `s` a `x` genera un entero que es casi más grande de lo que puede almacenar convenientemente la máquina. Esto es debido a que $128^4 = 2^{28}$, que sólo se diferencia del mayor entero de una máquina 32-bit en un factor de 8. Como consecuencia, no podemos calcular la función de localización manejando directamente potencias de 128. En su lugar, nos basaremos en la siguiente observación. Un polinomio cualquiera

$$A_3X^3 + A_2X^2 + A_1X^1 + A_0X^0 \tag{19.1}$$

puede evaluarse como

$$(((A_3)X + A^2)X + A_1)X + A_0 \tag{19.2}$$

Usando un pequeño truco podemos evaluar eficientemente la función de localización sin producir desbordamientos.

Observe que en la Ecuación 19.2 evitamos el cálculo directo de X^i. Esto es conveniente por tres razones. En primer lugar, se evita un resultado intermedio demasiado grande, el cual, como veremos más tarde, provocaría en general, un desbordamiento. En segundo lugar, el cómputo en la ecuación sólo contiene tres productos y tres sumas; un polinomio de grado N se calcula con N productos y N sumas. Esto resulta más económico que el cálculo directo en la Ecuación 19.1. En tercer lugar, el cálculo se realiza de izquierda a derecha (A_3 corresponde a `'h'`, A_2 a `'o'` y así sucesivamente, y X es 128).

Permanece aún el problema de desbordamiento: como el resultado de los cálculos es el mismo, probablemente es demasiado grande. Sin embargo, sólo necesitamos el resultado módulo `tamanyoTabla`. Aplicando el operador `%` después de cada producto (o suma), conseguimos que los resultados intermedios sean pequeños[1]. La función de localización resultante se muestra en la Figura 19.1. Una característica algo incómoda de esta función es que el cálculo de módulo es costoso. Como el desbordamiento está permitido (y sus resultados son consistentes en una misma plataforma), podemos hacer que el cálculo de la función de localización sea algo más rápido realizando una sola operación módulo, justo antes del `return`. Desgraciadamente, la repetición del producto por 128 tiende a desplazar los caracteres del principio hacia la izquierda, no influyendo en el resultado. Para remediar esta situación, multiplicaremos por 37 en lugar de hacerlo por 128. Esto reduce el desplazamiento de los caracteres iniciales.

```
1  // Función de localización aceptable
2  public final static int hash( String clave, int tamanyoTabla )
3  {
4      int valorHash = 0;
5
6      for ( int i = 0; i < clave.length( ); i++ )
7          valorHash = valorHash * 128 + clave.charAt( i ) )
8                                                  % tamanyoTabla;
9      return valorHash;
10 }
```

Figura 19.1 Primer intento de implementación de una función de localización.

[1] En la Sección 7.4 se estudian las propiedades de la operación módulo.

El resultado se muestra en la Figura 19.2. Ésta no es necesariamente la mejor función posible. Es cierto además que en algunas aplicaciones (por ejemplo, cuando intervienen cadenas largas), podemos querer jugar con ella. Sin embargo, en la mayoría de los casos la función es bastante buena. Observe que el desbordamiento podría generar números negativos. Por ello, si el operador de módulo genera un valor negativo, lo convertimos en positivo (líneas 15 y 16). Nótese también que el resultado obtenido al permitir el desbordamiento y haciendo una sola operación de módulo final no es el mismo que el que se obtendría haciendo dicha operación tras cada paso. De modo que hemos alterado ligeramente la función original, pero esto no representa ningún problema, pues no se trataba de algo a mantener a ultranza.

Toda función de localización debe ser sencilla de computar, pero también ha de distribuir uniformemente las claves. Si existen demasiadas colisiones, la eficiencia de la tabla se reduce drásticamente.

Vale la pena insistir en que, aunque la eficiencia es una consideración importante durante el diseño de la función, perseguimos que ésta distribuya las claves de forma equitativa, a fin de reducir al máximo las colisiones. Como consecuencia, debemos ser cuidadosos en no llevar las optimizaciones demasiado lejos. Un ejemplo de ello es la función de localización de la Figura 19.3. En ella nos limitamos a añadir los caracteres a las claves y devolvemos el resultado módulo `tamanyoTabla`. ¿Qué podría ser más sencillo? La función es fácil de implementar y calcula el valor hash rápidamente. Sin embargo, si `tamanyoTabla` es muy grande, la función no distribuye bien las claves. Por ejemplo, supongamos que `tamanyoTabla` es 10.000. Supongamos también que las claves tienen, a lo sumo, 8 caracteres. Entonces, como un elemento ASCII del tipo `char` es un entero entre 0 y 127, la función sólo podría tomar valores entre 0 y 1.016 (127*8). Indudablemente, ésta no es una distribución uniforme. Toda la eficiencia ganada por la velocidad del cálculo de la función será más que contrarrestada por el esfuerzo que requiere resolver una gran cantidad de colisiones.

La tabla se indexa desde 0 hasta `tamanyoTabla-1`.

Por último, observe que 0 es un resultado posible de la función, por lo que las tablas se indexan comenzando en el 0.

```
1  /**
2   * Rutina de localización para objetos de la clase String.
3   * @param clave la cadena a procesar.
4   * @param tamanyoTabla el tamaño de la tabla hash.
5   * @return el valor hash.
6   */
7  public final static int hash( String clave, int tamanyoTabla )
8  {
9      int valorHash = 0;
10
11     for ( int i = 0; i < clave.length( ); i++ )
12         valorHash = 37 * valorHash + clave.charAt( i );
13
14         valorHash % = tamanyoTabla;
15         if ( valorHash < 0 )
16             valorHash + = tamanyoTabla;
17
18         return valorHash;
19 }
```

Figura 19.2 Función de localización más rápida que saca provecho del desbordamiento.

```
1  // Función de localización pobre cuando tamanyoTabla es grande
2  public final static int hash( String clave, int tamanyoTabla)
3  {
4      int valorHash = 0;
5
6      for ( int i = 0; i < clave.length( ); i++ )
7          valorHash + = clave.charAt( i );
8
9      return valorHash % tamanyoTabla;
10 }
```

Figura 19.3 Función de localización inadecuada cuando `tamanyoTabla` es grande.

19.3 Exploración lineal

En la *exploración lineal* las colisiones se resuelven examinando secuencialmente el vector (con circularidad) hasta que encontremos una posición vacía.

Ahora que disponemos de una función de localización, necesitamos decidir lo que hacer cuando se produzca una colisión. Más concretamente, si a X le corresponde una posición que ya está ocupada, ¿dónde lo colocamos? La estrategia más simple posible es la *exploración lineal*, que consiste en buscar secuencialmente en el vector hasta que encontremos una posición vacía. La búsqueda rota desde la última posición hasta la primera, si ello es necesario. La Figura 19.4 muestra el resultado de insertar las claves 89, 18, 49, 58 y 9 en una tabla hash cuando se emplea exploración lineal. Suponemos que la función de localización devuelve la clave X módulo el tamaño de la tabla. La Figura 19.4 incluye los valores de la función de localización.

```
hash( 89, 10 ) = 9
hash( 18, 10 ) = 8
hash( 49, 10 ) = 9
hash( 58, 10 ) = 8
hash(  9, 10 ) = 9
```

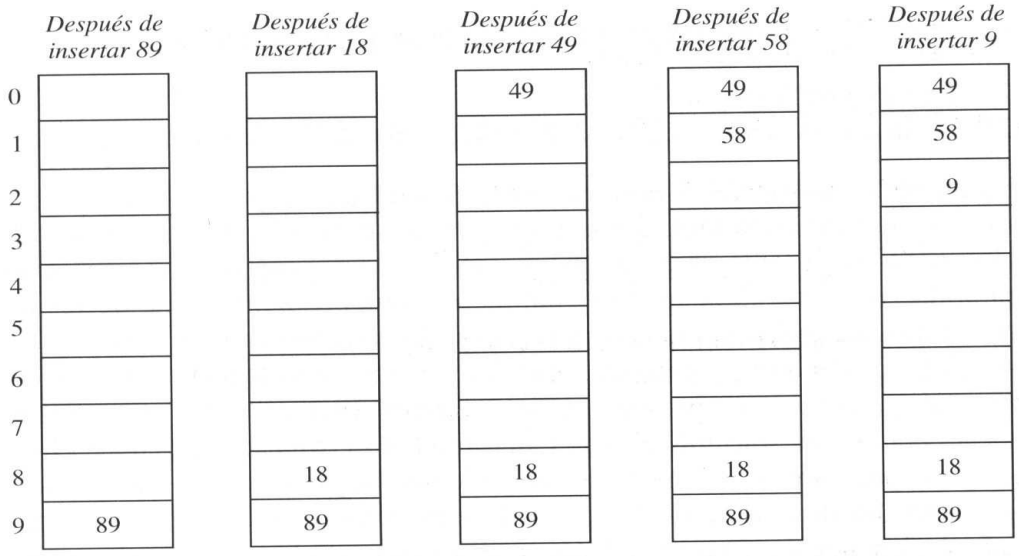

Figura 19.4 Tabla hash con exploración lineal tras cada inserción.

La primera colisión se produce cuando se inserta 49; este elemento se coloca en la siguiente posición vacía, que es la posición 0. Después, 58 colisiona con 18, 89 y 49 antes de encontrar una casilla vacía tres posiciones más allá, que es la posición 1. La colisión del elemento 9 se resuelve de modo similar. Siempre que la tabla sea suficientemente grande, podremos encontrar una casilla vacía. Sin embargo, el tiempo que se emplea en buscarla puede ser bastante elevado. Por ejemplo, si existe una sola casilla vacía en el vector, es posible que tengamos que examinar la tabla entera hasta llegar a ella. En media, podemos esperar tener que buscar en la mitad de la tabla hasta encontrar una casilla vacía. Esto está muy lejos del tiempo constante por acceso que deseamos. En contraposición, si suponemos que la tabla se va a mantener relativamente vacía, entonces las inserciones no serán tan costosas. Todo esto se discutirá en breve.

La rutina `buscar` sigue la misma secuencia de pruebas que `insertar`.

El algoritmo `buscar` sigue exactamente el mismo camino que el algoritmo `insertar`. Si llega a una casilla vacía , el elemento que estamos buscando no se encuentra en la tabla; en caso contrario, lo encontraremos. Por ejemplo, para encontrar 58, empezamos en la casilla 8 (tal y como indica la función hash). Aquí encontramos un elemento, pero como no es el buscado, pasamos a examinar la posición 0 y después la 1, hasta que lo encontramos. Si realizamos `buscar` con 19, se examinarían las casillas 9, 0, 1 y 2 antes de encontrar una celda vacía en la posición 3. Concluimos que no hemos encontrado 19.

Debemos emplear la eliminación perezosa.

La eliminación estándar no puede aplicarse, ya que al igual que sucede en los árboles binarios de búsqueda, un elemento de la tabla hash no sólo se representa a sí mismo, sino que conecta otros elementos haciendo de posición ocupada durante la resolución de conflictos. Así, si eliminamos 89 de la tabla hash, prácticamente todas las operaciones `buscar` posteriores fallarán. Como consecuencia, se implementa la eliminación perezosa, marcando los elementos como borrados de la tabla. Esta información se almacena en un atributo adicional. Cada elemento está *activo* o *borrado*.

19.3.1 Análisis de la exploración lineal

El análisis simplificado de la exploración lineal supone que las pruebas sucesivas son independientes. Esto no es cierto en la práctica, por lo que el análisis estima a la baja los costes de búsquedas e inserciones.

Para estimar la eficiencia de la exploración lineal, hacemos dos suposiciones:

1. La tabla hash es grande.
2. Cada intento en la tabla hash es independiente de los intentos anteriores.

La primera suposición es perfectamente razonable, ya que en otro caso no estaríamos manejando una tabla hash. La segunda suposición indica que si la fracción ocupada de la tabla es λ, entonces cada vez que examinamos una celda, la probabilidad de que esté ocupada es también λ, independientemente de los intentos anteriores. La independencia es una propiedad estadística importante que simplifica en un grado muy importante el análisis de sucesos aleatorios. Desgraciadamente, tal y como se discute en la Sección 19.3.2, dicha suposición de independencia no es sólo injustificada sino también errónea. A pesar de ello, nos es de ayuda pues nos indica hasta donde podemos llegar si somos más cuidadosos en la resolución de conflictos. Como ya hemos dicho antes, la eficiencia de una tabla hash depende en gran medida de lo llena que está la tabla. Su ocupación se indica mediante el factor de carga.

DEFINICIÓN: El *factor de carga* de una tabla hash es la fracción ocupada de la tabla. Denotamos el factor de carga mediante λ. Está entre 0 (vacía) y 1 (llena).

El *factor de carga* de una tabla hash es la fracción ocupada de la tabla. Está entre 0 (vacía) y 1 (llena).

Ahora podemos realizar, en el Teorema 19.1, un análisis simple de la exploración lineal, aunque parcialmente incorrecto como consecuencia de nuestras observaciones sobre la independencia.

Teorema 19.1

Si se supone independencia entre intentos, el número medio de celdas que se examinan en una inserción con exploración lineal es $1/(1 - \lambda)$.

Demostración

En una tabla con factor de carga λ*, la probabilidad de que una celda esté vacía es* $1 - \lambda$*. Como consecuencia, el número esperado de intentos independientes hasta encontrar una celda vacía es* $1/(1 - \lambda)$.

La demostración del Teorema 19.1 usa el hecho de que, si la probabilidad de que un suceso se produzca es p, se necesitan en media $1/p$ intentos para que dicho suceso se produzca, suponiendo que los intentos son independientes. Por ejemplo, el número esperado de lanzamientos de una moneda hasta que salga cara es dos, y la cantidad de tiros de un dado hasta que salga un 4 es seis. En ambos casos se asume independencia.

19.3.2 Lo que sucede realmente: la agrupación primaria

Desgraciadamente, la independencia asumida no se cumple. Esto se muestra en la Figura 19.5. En la parte superior tenemos el resultado de llenar la tabla hasta el 70 por ciento de su capacidad, cuando los intentos sucesivos son independientes. En el centro el correspondiente al uso de la exploración lineal. Observe los conjuntos de agrupaciones; éste es el fenómeno conocido como *agrupación primaria*.

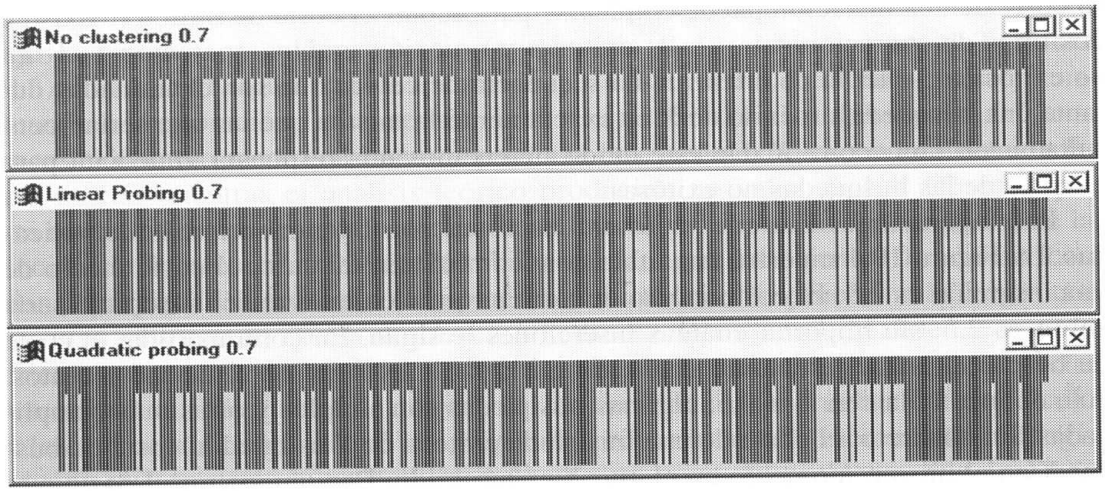

Figura 19.5 Ilustración de la agrupación primaria en la exploración lineal (en el centro) respecto a la no agrupación (arriba) y a la menos importante agrupación secundaria en la exploración cuadrática (abajo); las líneas más largas representan celdas ocupadas; el factor de carga es 0.7.

El efecto de la *agrupación primaria* es la formación de grandes grupos de celdas ocupadas, haciendo que las inserciones dentro las agrupaciones sean más costosas. Además, las inserciones hacen que dichos grupos crezcan.

En la agrupación primaria se tienen bloques de celdas ocupadas. Como consecuencia, muchas claves requerirán un número excesivo de intentos para resolver el conflicto, para terminar añadiéndose a uno de los bloques. En la agrupación primaria, la falta de eficiencia no está sólo provocada por los elementos que colisionan por tener el mismo valor hash, sino también por aquéllos que colisionan en las posiciones alternativas de otros. El análisis matemático que se precisa para tener todo esto correctamente en cuenta es bastante complicado, pero ha sido realizado ya, obteniéndose el resultado del Teorema 19.2.

Teorema 19.2 *El número medio de celdas examinadas en una inserción con exploración lineal es cercano a* $(1 + 1/(1 - \lambda)^2)/2$.

Demostración *La demostración queda fuera de los objetivos del texto. Véase [6].*

La agrupación primaria presenta problemas con factores de carga elevados, aunque para tablas medio vacías su efecto no es desastroso.

Para una tabla medio llena, tenemos que 2,5 es la cantidad media de celdas examinadas durante una inserción. Esto es muy similar a lo que el análisis anterior ha indicado. Las diferencias se hacen mayores cuando λ se acerca a 1. Por ejemplo, si la tabla está ocupada en un 90 por ciento, entonces $\lambda = 0{,}9$. El análisis sugiere que deberían ser examinadas diez celdas. Esto es bastante, aunque podría resultar asumible. Sin embargo, según el Teorema 19.2, la respuesta real es que necesitaríamos examinar unas 50 celdas. Esto es ciertamente excesivo, particularmente porque se trata sólo de un valor medio, por lo que pueden existir inserciones todavía peores.

19.3.3 Análisis de la operación `buscar`

Una operación `buscar` no exitosa tiene el mismo coste que una inserción.

El coste de una inserción puede emplearse para acotar el coste de `buscar`. Existen dos tipos de operaciones `buscar`: exitosas y no exitosas. Una operación `buscar` no exitosa es sencilla de analizar. La secuencia de celdas que son examinadas durante una búsqueda sin éxito de X es exactamente la misma que las que serían consultadas en `insertar` X. De este modo obtenemos una respuesta inmediata para el coste de las búsquedas no exitosas.

El coste de una operación `buscar` exitosa es la media de los costes de las inserciones en tablas con factores de carga más pequeños.

Para operaciones `buscar` exitosas, las cosas son ligeramente más complicadas. La Figura 19.4 muestra una tabla con $\lambda = 0{,}5$. En tal caso, el coste medio de una inserción es 2,5. El coste medio de `buscar` un elemento recién insertado sería entonces 2,5, sin importar cuántas inserciones le sigan. En contrapartida, el coste de buscar el primer elemento insertado en la tabla es siempre el de 1,0 intentos. Así, en una tabla con $\lambda = 0{,}5$, algunas búsquedas son simples y otras más complicadas. En concreto, el coste de una búsqueda exitosa de X es igual al coste de buscar X en el mismo momento en el que X es insertado. Para encontrar el coste medio de una búsqueda exitosa en una tabla con factor de carga λ, debemos calcular el coste medio de una inserción haciendo la media sobre todos los factores de carga que nos llevan a λ. Con esta base, podemos calcular el tiempo medio de una búsqueda con exploración lineal.

Para reducir el número de intentos, necesitamos un esquema de resolución de conflictos que evite la agrupación primaria. Observe que si la tabla está medio vacía, eliminar los efectos de la agrupación primaria podría evitar, en media, la mitad de los intentos en el caso de una inserción o de una búsqueda sin éxito y una décima parte en el caso de una búsqueda exitosa. A pesar de que resulte razonable intentar reducir la probabilidad de las secuencias más largas, *la exploración lineal es una estrategia bastante útil*. Como es tan fácil de implementar, cualquier otro método que se utilice para eliminar la agrupación primaria debería tener una complejidad similar. En caso contrario, invertiríamos demasiado tiempo intentando evitar una pequeña fracción del número de pruebas. Una alternativa razonable es la *exploración cuadrática*.

19.4 Exploración cuadrática

La *exploración cuadrática* examina las celdas 1, 4, 9 y sucesivas a partir de la posición inicial.

La *exploración cuadrática* es un método de resolución de conflictos que elimina el problema de la agrupación primaria provocado por la exploración lineal. Su nombre se debe al uso de la fórmula $F(i) = i^2$ en la eliminación de los conflictos. Más concretamente, si la función de localización devuelve como resultado H y la búsqueda en la celda indexada por H no nos conduce a ninguna conclusión, entonces se consultan sucesivamente las celdas $H + 1^2$, $H + 2^2$, $H + 3^2$, ..., $H + i^2$, (empleando circularidad). Esto es claramente diferente a la exploración lineal que consultaría las celdas $H + 1$, $H + 2$, $H + 3$, ..., $H + i$.

Recordemos que los puntos siguientes de prueba distan de la posición original un número cuadrático de posiciones.

La Figura 19.6 muestra la tabla que se obtiene al emplear exploración cuadrática en lugar de exploración lineal, durante la inserción de la secuencia de la Figura 19.4. Cuando 49 colisiona con 89, la primera alternativa es visitar la celda siguiente. Dicha casilla está vacía, así que podemos colocar en ella el 49. A continuación, 58 colisiona con 8. Se examina la celda en la novena posición (la siguiente), pero se produce otra colisión. En la siguiente casilla examinada, que está $2^2 = 4$ posiciones más allá *de la posición hash original*, se encuentra una posición vacante. Así colocamos, en el segundo intento, el elemento 58. Lo mismo sucede para el 9. Observe que los lugares alternativos para los elementos cuyo valor hash es 8 y los lugares alternativos para los elementos cuyo valor hash es 9 no son los mismos. La larga secuencia de intentos realizada para insertar el elemento 58 no afecta a la inserción posterior de 9. Esto contrasta con lo que sucedía cuando empleábamos exploración lineal.

Antes de codificar los algoritmos debemos considerar algunos detalles:

- En la exploración lineal, cada intento examina una celda diferente. ¿Garantiza la exploración cuadrática que cuando se examina una celda, ésta no ha sido consultada previamente en el proceso actual? ¿Garantiza la exploración cuadrática que durante la inserción de X, X es siempre insertado si la tabla no está aún completa?
- La exploración lineal se implementa fácilmente. La exploración cuadrática parece precisar de las operaciones multiplicación y módulo. ¿Esta aparente complejidad adicional hace que la exploración cuadrática sea ineficiente?
- ¿Qué sucede (en ambas exploraciones, lineal y cuadrática) si el factor de carga es demasiado elevado? ¿Podemos expandir dinámicamente la tabla, de modo similar a cómo se hace con las estructuras de datos basadas en vectores?

```
hash( 89, 10 ) = 9
hash( 18, 10 ) = 8
hash( 49, 10 ) = 9
hash( 58, 10 ) = 8
hash(  9, 10 ) = 9
```

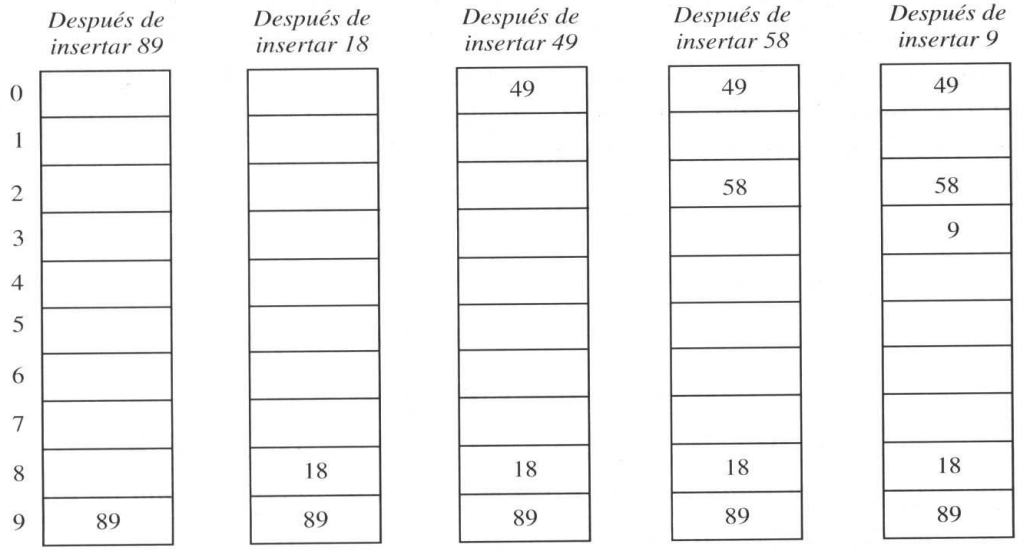

Figura 19.6 Tabla hash con exploración cuadrática tras cada inserción (nótese que el tamaño de la tabla ha sido elegido inadecuadamente ya que no es un número primo).

Afortunadamente, las noticias son relativamente buenas en todos los puntos. Si el tamaño de la tabla es primo y el factor de carga no excede nunca 0,5, sabemos que siempre insertaremos el nuevo elemento X y que ninguna celda será consultada dos veces durante un acceso. Sin embargo, para garantizar que se cumplen estas propiedades, debemos asegurar que el tamaño de la tabla es un número primo, tal y como muestra el Teorema 19.4. Por completitud, la Figura 19.7 muestra una rutina que genera números primos. Dicha rutina emplea el algoritmo de la Figura 9.8.

Si el tamaño de la tabla es primo y el factor de carga no excede nunca 0,5, entonces todos los intentos se realizarán sobre celdas distintas y siempre podremos insertar un elemento.

```
1  /**
2   * Método interno para generar un número primo
3   * mayor o igual que n. Véase la Figura 9.8 para encontrar esPrimo.
4   */
5  private final static int siguientePrimo( int n )
6  {
7      if( n % 2 == 0 )
8          n++;
9
10     for ( ; esPrimo( n ); n += 2 )
11         ;
12
13     return n;
14 }
```

Figura 19.7 Rutina empleada en la exploración cuadrática para encontrar un primo mayor o igual que N.

Teorema 19.4 *Si se emplea exploración cuadrática y el tamaño de la tabla es un número primo, siempre podemos insertar un nuevo elemento en la tabla si ésta se encuentra medio vacía. Además, durante una operación de inserción no se examina ninguna celda dos veces.*

Demostración *Sea M el tamaño de la tabla. Supongamos que M es un número primo mayor que 3. Mostraremos en primer lugar, que las primeras* $\lfloor M/2 \rfloor$ *posiciones alternativas son distintas. Dos de estas posiciones son* $H + i^2 \pmod{M}$ *y* $H + j^2 \pmod{M}$, *donde* $0 < i, j \leq \lfloor M/2 \rfloor$. *Supongamos que ambas son iguales, pero que* $i \neq j$. *Entonces*

$$H + i^2 \equiv H + j^2 \pmod{M}$$

$$i^2 \equiv j^2 \pmod{M}$$

$$i^2 - j^2 \equiv 0 \pmod{M}$$

$$(i - j)(i + j) \equiv 0 \pmod{M}.$$

Como M es primo, se sigue que $i + j$ *o* $i - j$ *es divisible por M. Como i y j son distintos y su suma es menor que M, no puede darse ninguna de estas alternativas, de modo que se llega a una contradicción. Como consecuencia, las primeras* $\lfloor M/2 \rfloor$ *posiciones alternativas son todas distintas, de modo que una inserción siempre tendrá éxito si la tabla está al menos medio vacía.*

Si la tabla está algo más que medio llena, la inserción puede fallar (aunque esto es extremadamente improbable). En la práctica, no nos planteamos la posibilidad de errores durante las inserciones, ya que ello implicaría que hemos realizado demasiados intentos. Recordemos que en cada inserción esperamos realizar cerca de 2 o 2,5 intentos, mientras que para fallar en una de ellas con una tabla de tamaño 100.000 se necesitarían 50.000 intentos. Además, si el tamaño de la tabla es siempre un número primo y el factor de carga permanece por debajo de 0,5, hemos garantizado absolutamente el éxito de la inserción. Si el tamaño de la tabla no es primo, el número de posiciones alternativas puede verse reducido dramáticamente. Por ejemplo, si el tamaño de la tabla es 16, entonces las únicas posiciones alternativas están a 1, 4 y 9 casillas de la inicial. Una vez más esto no es considerable: aunque no hemos garantizado $\lfloor M/2 \rfloor$ intentos, en la mayoría de las ocasiones tendremos muchos más de las que esperábamos necesitar. A pesar de todo, es mejor jugar sobre seguro y emplear la teoría como guía para escoger los parámetros. Más aún, se ha demostrado empíricamente que los tamaños primos son los ideales para las tablas hash, ya que eliminan algo de la no aleatoriedad ocasionalmente introducida por las funciones de localización.

La exploración cuadrática puede implementarse sin multiplicaciones ni operaciones módulo. Debido a que no sufre agrupación primaria, en la práctica es más adecuada.

El segundo aspecto importante es la eficiencia. Recordemos que para un factor de carga de 0,5, la eliminación de la agrupación primaria evita en media sólo 0,5 de las pruebas en una inserción y 0,1 en una búsqueda con éxito. Tenemos otro beneficio adicional: encontrar una secuencia de pruebas larga es significativamente menos probable. Sin embargo, si realizar una prueba en la exploración cuadrática es dos veces más costoso, el esfuerzo no valdrá la pena. La exploración lineal se implementa mediante una sencilla suma (de 1), un test para comprobar si necesitamos circularidad, y una sustracción (en el caso de que se precise circularidad). La fórmula

para la exploración cuadrática indica que se necesita sumar 1 (para pasar de $i - 1$ a i), una multiplicación (para calcular i^2), otra suma y una operación módulo. Ciertamente, estos cálculos parecen demasiado costosos para ser empleados en la práctica. Sin embargo, disponemos del truco siguiente, explicado en el Teorema 19.5.

Teorema 19.5

La exploración cuadrática puede implementarse sin costosas multiplicaciones ni divisiones.

Demostración

Sea H_{i-1} la posición computada más reciente (H_0 es la posición hash inicial), y H_i la posición que estamos intentado calcular. Entonces se tiene

$$\begin{aligned} H_i &= H_0 + i^2 \,(\text{mod}\, M) \\ H_{i-1} &= H_0 + (i-1)^2 \,(\text{mod}\, M). \end{aligned} \tag{19.3}$$

Si restamos ambas ecuaciones en la Ecuación 19.3, obtenemos

$$\begin{aligned} H_i - H_{i-1} &= i^2 - (i-1)^2 \,(\text{mod}\, M) \\ H_i &= H_{i-1} + 2i - 1 \,(\text{mod}\, M). \end{aligned} \tag{19.4}$$

La Ecuación 19.4 nos indica que podemos calcular el nuevo valor H_i a partir del valor previo H_{i-1} sin elevar i al cuadrado. Aunque aún tenemos una multiplicación, ésta es por 2, implementada trivialmente en la mayoría de los computadores mediante el intercambio de bits. La operación módulo no es realmente necesaria, ya que el valor de $2i - 1$ debe ser menor que M. Así, si la añadimos a H_{i-1}, el resultado seguirá siendo menor que M (en cuyo caso no necesitamos el módulo) o ligeramente mayor (en cuyo caso, podemos calcular el módulo restándole M).

El Teorema 19.5 muestra que podemos calcular la siguiente posición que debemos examinar empleando una suma (para incrementar i), un cambio de bits (para evaluar $2i$), una sustracción por 1 (para evaluar $2i - 1$), otra suma (para incrementar la posición inicial con $2i - 1$), un test para comprobar si es necesaria la circularidad, y una poco probable sustracción para implementar la operación módulo. Como consecuencia, la diferencia entre la exploración lineal y la exploración cuadrática es, por cada prueba, un cambio de bits, una sustracción por 1 y una suma. Esto parece ser menor que el coste de realizar una prueba adicional cuando se manejan claves complicadas (como cadenas de caracteres).

El último detalle a considerar es la expansión dinámica. Si el factor de carga es superior a 0,5, duplicaremos el tamaño de la tabla hash. Esto nos lleva a considerar varios puntos. En primer lugar, ¿cuánto nos costará encontrar otro número primo? La respuesta es que es fácil encontrarlos. Sólo tenemos que examinar $O(\log N)$ números antes de encontrar uno que sea primo. Como consecuencia, la rutina de la Figura 19.7 es muy eficiente. Como el test de primalidad tiene un coste máximo de $O(N^{1/2})$, la búsqueda de un número primo tiene un coste máximo de $O(N^{1/2} \log N)$ [2]. Esto es mucho menos que el coste $O(N)$ de copiar los contenidos de la tabla vieja en la nueva.

Debe expandirse la tabla siempre que el factor de carga sea superior a 0,5. Este proceso se conoce como *rehashing*. Siempre debe incrementarse el tamaño de la tabla hasta llegar a un número primo. Dichos números primos son fáciles de encontrar. Cuando expandimos la tabla hash, reinsertamos los elementos de la tabla antigua en la nueva empleando una nueva función hash.

[2] Esta rutina también es necesaria si incluimos un constructor que permita al usuario especificar el tamaño inicial aproximado de la tabla hash. La implementación de la tabla hash es la responsable de asegurar que se emplea un número primo.

Una vez que se ha asignado memoria al vector de mayor tamaño, ¿debemos copiar en él toda la información? La respuesta es no. Un nuevo vector implica una nueva función hash, de modo que no podemos emplear las posiciones antiguas. Debemos buscar cada elemento en la tabla vieja, calcular su nuevo valor hash e insertarlo en la nueva tabla. Este proceso recibe el nombre de *rehashing*. La sección siguiente muestra que es sencillo implementar el rehashing en Java.

```
1  package EstructurasDatos;
2
3  import Soporte.*;
4  import Excepciones.*;
5
6  // Clase abstracta ExploracionTablaHash
7  //
8  // CONSTRUCCIÓN: sin ninguna inicialización
9  //
10 // ***********************OPERACIONES PÚBLICAS************************
11 // void insertar( x )    --> Inserta x
12 // void eliminar( x )    --> Elimina x
13 // Hashable buscar( x ) --> Devuelve el elemento que ajusta con x
14 // void vaciar( )        --> Elimina todos los elementos
15 // int hash( String str, int tamanyoTabla )
16 //                       --> Método para indexar cadenas en una tabla hash
17 // ******************************ERRORES*******************************
18 // buscar y eliminar lanzan ElementoNoEncontrado
19 // insertar sobreescribe el valor anterior en caso de repetición; no error
20
21 /**
22  * Implementación de las tablas hash mediante tablas con exploración.
23  * Es una clase abstracta que debe ser extendida para
24  * implementar un algoritmo de exploración concreto, como
25  * la exploración cuadrática.
26  * Nótese que todos los "ajustes" se basan en el método equals.
27  */
28 public abstract class ExploracionTablaHash implements TablaHash
29 {
30     /**
31      * Método abstracto para la resolución de conflictos.
32      * Es el único método que debe ser rescrito en cada clase.
33      * @param x el elemento a insertar.
34      * @return la posición donde finaliza la búsqueda.
35      */
36     protected abstract int buscarPos( Hashable x );
37
38     public final static int hash( String clave, int tamanyoTabla )
39       { /* Figura 19.2 */ }
40     public ExploracionTablaHash( )
41       { /* Figura 19.10 */ }
42     public final vaciar( )
43       { /* Figura 19.10 */ }
44     public final Hashable buscar( Hashable x ) throws ElementoNoEncontrado
45       { /* Figura 19.11 */ }
46     public final void eliminar( Hashable x ) throws ElementoNoEncontrado
47       { /* Figura 19.12 */ }
48     public final void insertar( Hashable x )
49       { /* Figura 19.13 */ }
```

Figura 19.8 Esqueleto de la clase de las tablas hash con exploración (parte 1).

19.4.1 Implementación en Java

El usuario debe implementar una función de localización adecuada y un método `equals` para los objetos almacenados en la tabla hash.

Ahora ya estamos preparados para presentar una implementación completa en Java de las tablas hash con exploración cuadrática. Emplearemos herencia y definiremos en primer lugar una clase abstracta que implemente las tablas hash. Después extenderemos dicha clase codificando un método de resolución de conflictos mediante exploración cuadrática. Recordemos que el interfaz `Hashable` necesita la implementación de una función hash adecuada. El esqueleto de la clase se muestra en las Figuras 19.8 y 19.9. A continuación se describe detalladamente cada rutina de la misma.

La tabla hash consiste en un vector de referencias `EntradaHash`; cada referencia es `null` o apunta a un objeto que almacena un elemento y un atributo que indica si esa entrada está activa o se ha eliminado. El vector se declara en la línea 53 como `protected`, de modo que el método de resolución de conflictos, implementado en las clases derivadas, pueda acceder a él. Necesitamos conocer el número de elementos insertados en la tabla hash (incluyendo los elementos marcados como eliminados). Dicho número se almacena en `tamanyoActual`, declarado en la línea 54.

El resto de la clase contiene las declaraciones de las rutinas de la tabla hash. La operación `buscar` devuelve el elemento encontrado en la búsqueda de `x`. Si `x` no se encuentra en la tabla, se lanza una excepción. El método de inserción añade un nuevo elemento a la tabla; si dicho elemento ya está en ella, el valor anterior se sobrescribe con el nuevo.

Las rutinas de inicialización de la tabla hash se muestran en la Figura 19.10. En ellas no hay ningún detalle notable. Observe que inicialmente todas las referencias de `vector` son `null`. Las rutinas de búsqueda se muestran en la Figura 19.11.

```
50      private static final int TAMANYO_TABLA_POR_DEFECTO = 11;
51
52          /** El vector de elementos. */
53      protected EntradaHash [ ] vector; // El vector de elementos
54      private int tamanyoActual;  // El número de celdas ocupadas
55
56      private final void crearVector( int tamanyoVector )
57        { /* Figura 19.10 */ }
58      private final void confirmaBuscar( int posicionActual,
59                        String mensaje ) throws ElementoNoEncontrado
60        { /* Figura 19.11 */ }
61  }
62
63  // La entrada básica almacenada en ExploracionTablaHash
64  class EntradaHash
65  {
66      Hashable elemento;    // El elemento
67      boolean estaActivo;   // false si se elimina
68
69      public EntradaHash( Hashable e )
70        { this( e, true ); }
71
72      public EntradaHash( Hashable e, boolean i )
73        { elemento = e; estaActivo = i; }
74  }
```

Figura 19.9 Esqueleto de la clase de las tablas hash con exploración (parte 2).

```
1  /**
2   * Método interno para asignar memoria a vector.
3   * @param tamanyoVector el tamaño del vector.
4   */
5  private final void crearVector( int tamanyoVector )
6  {
7      vector = new EntradaHash[ tamanyoVector ];
8  }
9
10 /**
11  * Construye la tabla hash.
12  */
13 public ExploracionTablaHash( )
14 {
15     crearVector( TAMANYO_TABLA_POR_DEFECTO );
16     vaciar( );
17 }
18
19 /**
20  * Vacía de forma lógica la tabla hash.
21  */
22 public final void vaciar( )
23 {
24     tamanyoActual = 0;
25     for( int i = 0; i < vector.length; i++ )
26         vector[ i ] = null;
27 }
```

Figura 19.10 Inicialización de la tabla hash.

```
1  /**
2   * Busca un elemento en una tabla hash.
3   * @param x el elemento a buscar.
4   * @return el elemento que ajusta con el buscado.
5   * @exception ElementoNoEncontrado si no hay en la tabla
6   *      ningún elemento que ajuste con x.
7   */
8  public final Hashable
9  buscar( Hashable x ) throws ElementoNoEncontrado
10 {
11     int posicionActual = buscarPos( x );
12     confirmaBuscar( posicionActual, "buscar de ExploracionTablaHash" );
13     return vector[ posicionActual ].elemento;
14 }
15
16 /**
17  * No hace nada si posicionActual existe y está activa.
18  * En caso contrario, lanza una excepción.
19  * @param posicionActual el resultado de la llamada a buscarPos.
20  * @param mensaje la cadena para construir la excepción.
21  * @exception ElementoNoEncontrado si la celda está inactiva.
22  */
23 private final void confirmaBuscar( int posicionActual,
24                                    String mensaje ) throws ElementoNoEncontrado
25 {
26     if( vector[ posicionActual ] == null ||
27         vector[ posicionActual ].estaActivo == false )
28         throw new ElementoNoEncontrado( mensaje );
29 }
```

Figura 19.11 Método `buscar` de la clase de las tablas hash con exploración.

Todas ellas emplean el método abstracto `buscarPos`. Debemos extender la clase abstracta `ExploracionTablaHash` implementando dicho método. Las rutinas `buscar` son sencillas de codificar. Un elemento es encontrado si el resultado de `buscarPos` es una celda activa (esto es, no es `null` y tampoco está marcada como eliminada). De este modo, si `buscarPos` se detiene en una celda activa debe existir un elemento que ajuste con el buscado. Esta comprobación se hace en el método privado `confirmaBuscar`. De modo similar, la rutina `eliminar` de la Figura 19.12 es sencilla. Comprobamos si `buscarPos` nos conduce a una celda activa. Si es así, la celda se marca como eliminada; en caso contrario, hemos acabado.

La mayoría de las rutinas se reducen a unas pocas líneas de código, ya que para realizar la exploración cuadrática invocan a `buscarPos`.

La rutina de inserción se muestra en la Figura 19.13. En la línea 9 invocamos al método `buscarPos`. `buscarPos` nos devuelve la posición en la que insertar `x`.

La mayor parte de `insertar` se refiere al rehashing, que se realiza cuando la tabla está (medio) llena.

```
1  /**
2   * Elimina elementos de la tabla hash.
3   * @param x el elemento a eliminar.
4   * @exception ElementoNoEncontrado si no se encuentra
5   *      en la tabla hash ningún elemento que ajuste con x.
6   */
7  public final void eliminar( Hashable x ) throws ElementoNoEncontrado
8  {
9      int posicionActual = buscarPos( x );
10     confirmaBuscar( posicionActual, "eliminar de
                                        ExploracionTablaHash" );
11     vector[ posicionActual ].estaActivo = false;
12 }
```

Figura 19.12 Rutina `eliminar` para las tablas hash con exploración.

```
1  /**
2   * Inserción en la tabla hash. Si el elemento ya está en
3   * en la tabla, lo reemplaza por el nuevo elemento.
4   * @param x el elemento a insertar.
5   */
6  public final void insertar( Hashable x )
7  {
8          // Inserta x como activo
9      int posicionActual = buscarPos( x );
10     vector[ posicionActual ] = new EntradaHash( x, true );
11     if( ++tamanyoActual < vector.length / 2 )
12         return;
13
14         // CÓDIGO DE REHASHING
15     EntradaHash [ ] vectorAntiguo = vector;
16
17         // Crea una nueva tabla vacía con tamaño doble
18     crearVector( siguientePrimo( 2 * vectorAntiguo.length ) );
19     tamanyoActual = 0;
20
21         // Copia la tabla
22     for( int i = 0; i < vectorAntiguo.length; i++ )
23         if( vectorAntiguo[ i ] != null && vectorAntiguo[ i ].estaActivo )
24             insertar( vectorAntiguo[ i ].elemento );
25 }
```

Figura 19.13 Rutina de inserción de la clase de las tablas hash con exploración, que incluye el código de rehashing.

```
1  // Implementación de las tablas hash con exploración cuadrática.
2  public class TablaExploracionCuadratica extends ExploracionTablaHash
3  {
4      protected final int buscarPos( Hashable x )
5      {
6          int colision = 0;
7          int posicionActual = x.hash( vector.length );
8
9          while( vector[ posicionActual ] != null &&
10                !vector[ posicionActual ].elemento.equals( x ) )
11             {
12             posicionActual += 2 * ++colision - 1;
13             if( posicionActual >= vector.length )
14                 posicionActual -= vector.length;
15             }
16
17             return posicionActual;
18     }
19 }
```

Figura 19.14 Clase que implementa la exploración cuadrática.

La inserción se realiza en la línea 10, en la que construimos una nueva `EntradaHash` y la añadimos al vector. Observe que esto sobrescribe cualquier valor que ajuste con `x` almacenado previamente. En la línea 11 ajustamos `tamanyoActual` y terminamos, a menos que sea necesario realizar una reevaluación de la función de localización (*rehashing*). Así, el código restante implementa el rehashing.

En la línea 15 se guarda una referencia a la tabla original. Después creamos una nueva tabla vacía de tamaño doble, en las líneas 18 y 19. Por último, recorremos el vector antiguo insertando sus elementos activos en la nueva tabla. La rutina `insertar` emplea una nueva función de localización (ya que el vector tiene un tamaño diferente) y resuelve de forma automática todas las colisiones. Podemos estar seguros de que la llamada recursiva a `insertar` (en la línea 24) no fuerza un nuevo rehash. Como alternativa, podemos reemplazar la línea 24 con dos líneas de código entre sus correspondientes llaves (*véase* el Ejercicio 19.14).

La exploración cuadrática se implementa en `buscarPos`. Esta rutina emplea el truco descrito anteriormente para evitar el cálculo de las multiplicaciones y de las operaciones de módulo.

De momento, nada de lo que hemos hecho depende de la exploración cuadrática. La Figura 19.14 implementa dicha estrategia de resolución de conflictos extendiendo `ExploracionTablaHash` y codificando el método `buscarPos` mediante el algoritmo de exploración cuadrática. La búsqueda en la tabla continúa hasta que se encuentra una celda vacía o un elemento que ajuste. La metodología descrita en el Teorema 19.5 se implementa en las líneas 12 a 14. Observe que si `x` está marcado como eliminado, su posición será devuelta por `buscarPos`. `confirmaBuscar` lanzará una excepción siempre que determine que `x` no es un elemento activo.

19.4.2 Análisis de la exploración cuadrática

El análisis matemático riguroso de la exploración cuadrática todavía no ha podido ser realizado, debido a su complejidad. Aunque esta estrategia elimina la agrupación primaria, los elementos indexados en la misma posición probarán las mismas celdas alternativas. Esto se conoce con el nombre de *agrupación secundaria*. Debido a ello, de nuevo, no podemos suponer la independencia de intentos sucesivos.

Sin embargo, la agrupación secundaria produce una separación mucho menor de los resultados teóricos. Los resultados de las simulaciones sugieren que, por lo general, sólo se incrementa en 1/2 el número de intentos por búsqueda, y esto sólo llega a suceder en tablas con factores de carga elevados. La Figura 19.5 muestra la diferencia entre la exploración lineal y la cuadrática, poniendo de manifiesto que el grado de agrupación es menor en la exploración cuadrática que en la lineal.

En la *agrupación secundaria*, los elementos indexados en la misma posición examinarán las mismas celdas alternativas. La agrupación secundaria supone una separación menor de los resultados teóricos.

Existen varias técnicas para eliminar la agrupación secundaria. La más conocida es el *doble hashing*, que emplea una segunda función de localización para la resolución de conflictos. Más concretamente, realizamos intentos a distancia $Hash_2(X)$, $2Hash_2(X)$ y así sucesivamente. Esta segunda función de localización debe reunir ciertas propiedades (por ejemplo, no puede devolver 0), y debemos asegurar que puede llegar a examinar todas las celdas. Una función como $Hash_2(X) = R - (X \bmod R)$, donde R es un primo menor que M, generalmente funcionará bien. El doble hashing es muy interesante teóricamente hablando, ya que puede demostrarse que realiza el mismo número de intentos que se obtienen en el análisis aleatorio teórico de la exploración lineal. Sin embargo, es algo más complicado de implementar que la exploración cuadrática, y como veremos, algunos de sus detalles requieren especial atención.

El *hashing doble* es una técnica de hashing que elimina la agrupación secundaria. Dicha técnica emplea una segunda función de localización para la resolución de conflictos.

Aparentemente no existe ningún argumento de peso para no emplear en la práctica la estrategia de exploración cuadrática, salvo el gasto que supone mantener la tabla medio vacía. Además, éste sólo sería relevante en otros lenguajes de programación, distintos a Java, en los que es posible que los elementos que almacenemos sean de gran tamaño.

19.5 Hashing enlazado

Una alternativa a la exploración cuadrática muy conocida es el *hashing enlazado* o *hashing abierto*. En el hashing enlazado, se maneja un vector de listas enlazadas: $L_0, L_1, \ldots, L_{M-1}$. La función de localización indica en qué lista debemos insertar el elemento X y después, durante la ejecución de una operación `buscar`, nos dice en qué lista se encuentra el elemento X. La idea subyacente es que, aunque la búsqueda en una lista enlazada es una operación lineal, si las listas son lo suficientemente cortas, el tiempo de búsqueda será reducido. En particular, supongamos que el factor de carga, que ahora se define como N/M, es λ. Nótese que en el hashing enlazado, el factor de carga no está acotado por 1,0. La longitud media de cada lista es λ, lo que hace que el número medio de intentos durante una inserción o una búsqueda sin éxito sea λ. El número de intentos correspondiente a una búsqueda exitosa es $1 + \lambda/2$. Esto es debido a que una búsqueda exitosa siempre se realizará en una lista no vacía, y es de esperar que tengamos que recorrer la mitad de sus elementos. El coste relativo de una búsqueda exitosa respecto a una no exitosa puede parecer anómalo cuando $\lambda < 2$, ya que la primera de ellas es más costosa que la segunda. Sin embargo, esto tiene sentido ya que muchas de las búsquedas sin éxito toparían con una lista vacía.

El *hashing enlazado* es una alternativa eficiente, hablando en términos de espacio, a la exploración cuadrática, en la que se maneja un vector de listas enlazadas. Es menos sensible a factores de carga elevados.

Un factor de carga muy corriente es 1,0; un factor de carga menor no aumenta de modo significativo la eficiencia, y además necesita de un espacio extra. El atractivo del hashing enlazado es que la eficiencia no se ve afectada por un moderado incremento del factor de carga, además de que podemos evitar el rehashing.

En el hashing enlazado, un factor razonable de carga es 1,0. Un factor de carga menor no mejora su eficiencia de forma sensible, mientras que otro moderadamente elevado es aceptable y puede suponer un ahorro de espacio.

Éste es un detalle importante en los lenguajes en los que no se puede realizar la expansión dinámica de los vectores. Además, el número de pruebas de una búsqueda es menor que en la exploración cuadrática, en particular en las búsquedas no exitosas.

Podemos implementar el hashing enlazado empleando las clases de listas enlazadas existentes. Sin embargo, como en esta ocasión el nodo cabecera representa un gasto inútil de espacio, pues no se necesita para nada, podemos optar por no reutilizar componentes, implementando las listas como si fuesen pilas. El esfuerzo realizado en la codificación resulta gratamente clarificador. Además, el gasto de memoria se reduce a una referencia por nodo, además de una referencia adicional por lista; por ejemplo, cuando el factor de carga es 1,0 tenemos dos referencias por elemento. Este hecho puede ser importante en otros lenguajes de programación si los elementos a guardar son de gran tamaño. En nuestro caso, tenemos las mismas dificultades que en las implementaciones de las pilas basadas en vectores y en listas enlazadas.

Resumen

Las tablas hash pueden emplearse para que el coste medio de las operaciones `insertar` y `buscar` sea constante. Cuando utilizamos tablas hash, es especialmente importante prestar atención a detalles como el factor de carga; en caso contrario, las cotas constantes de tiempo no son de aplicación. También es importante elegir cuidadosamente la función de localización, sobre todo cuando las claves que se manejan no son enteros ni cadenas cortas. Debe elegirse una función fácilmente calculable con una buena distribución de sus valores.

Normalmente, en el hashing enlazado el factor de carga está muy cercano a 1, aunque la eficiencia no empeora sensiblemente cuando este factor crece mucho. En la exploración cuadrática, el tamaño de la tabla debe ser un número primo y el factor de carga no debería exceder 0,5. El rehashing debe emplearse para permitir que la tabla crezca manteniendo el factor de carga adecuadamente. Esto es importante si hay escasez de espacio y no es posible declarar inicialmente una tabla hash de gran tamaño.

Debe usarse una tabla hash en lugar de un árbol binario de búsqueda siempre que no se necesite acceso por posición y puedan aparecer entradas ordenadas.

A pesar de los resultados obtenidos para las tablas hash, podemos seguir empleando árboles binarios de búsqueda para implementar las operaciones `insertar` y `buscar`. Aunque las cotas del tiempo medio obtenidas son $O(\log N)$, los árboles binarios de búsqueda permiten disponer adicionalmente de diversas rutinas que precisan de un cierto orden y son, por tanto, más potentes. Si empleamos una tabla hash no podemos buscar de forma eficiente el menor de los elementos almacenados ni extender la tabla para permitir computaciones sobre el i-ésimo elemento de la tabla respecto del orden. Además, no podemos buscar eficientemente una cadena a menos que la conozcamos de forma exacta. Un árbol binario de búsqueda puede encontrar rápidamente todos los elementos que se encuentran en un rango determinado, mientras que esto no lo puede hacer una tabla hash. Por otra parte, en la práctica la cota $O(\log N)$ no resulta mucho mayor que $O(1)$, especialmente porque los árboles binarios no necesitan ni de multiplicaciones ni de divisiones.

Por otra parte, el caso peor para el hashing suele provenir de una incorrecta elección de los mecanismos que intervienen, mientras que una entrada ordenada

puede provocar el mal comportamiento de los árboles binarios. Los árboles binarios de búsqueda son algo costosos de implementar. Así que, si no necesitamos ninguna información de orden y existe alguna posibilidad de que aparezcan entradas ordenadas, la tabla hash es la estructura de datos que debemos emplear.

Las aplicaciones del hashing son muy numerosas.

Las aplicaciones del hashing son muy numerosas. Los compiladores emplean tablas hash para tratar las variables declaradas en el código. Dicha estructura de datos se llama *tabla de símbolos*. Las tablas hash son las más adecuadas en este caso ya que sólo se necesitan operaciones de `insertar` y `buscar`. Normalmente, los identificadores son cortos, de modo que la función hash puede computarse rápidamente. Observe que en esta aplicación, la mayoría de las búsquedas tienen éxito.

Otro uso muy común de las tablas hash son los juegos de computador. A medida que el programa busca entre las distintas líneas de juego, lleva la cuenta de las posiciones que ha visto mediante la computación de una función hash basada en la posición (y almacenando el movimiento hecho para esa posición). Si una de dichas posiciones se repite, el programa puede evitar la repetición de (costosos) cálculos haciendo una transposición de movimientos. Esta estructura común a todos los juegos de computador se llama *tabla de transposición*. Esto ya se vio en la Sección 10.2, cuando se implementó el juego de las tres en raya.

Un tercer uso del hashing son los correctores ortográficos. Si la detección de los errores prima sobre su corrección, se puede hacer un hashing previo sobre un diccionario completo, pudiéndose comprobar la corrección de las palabras en tiempo constante. Las tablas hash son adecuadas para ello ya que no es importante mantener ordenadas las palabras. Basta con imprimir los errores en el mismo orden en el que se producen.

Esto completa la discusión de los mecanismos básicos de búsqueda. El siguiente capítulo estudia los montículos binarios, los cuales implementan las colas de prioridad, permitiendo así el acceso eficiente a los elementos más pequeños de una colección.

Elementos del juego

agrupación primaria Problema de la exploración lineal que afecta a la eficiencia. Aparecen grandes grupos de celdas ocupadas, haciendo que las inserciones en ellos sean más costosas. Además, dichas inserciones provocan que los grupos de celdas ocupadas crezcan aún más.

agrupación secundaria Agrupación que se produce cuando los elementos indexados en la misma posición examinan las mismas celdas alternativas. Supone una desviación menor de los resultados teóricos.

colisión Se produce cuando a dos o más elementos de una tabla hash les corresponde en ella la misma posición. Este problema es inevitable ya que habitualmente hay más elementos que posiciones.

eliminación perezosa Técnica de marcar los elementos como eliminados en lugar de borrarlos físicamente. Es necesaria en las tablas hash con exploración.

exploración cuadrática Resolución de conflictos que examina las celdas 1, 4, 9 y así sucesivamente, a partir de la posición inicial.

exploración lineal Técnica para evitar los conflictos, que examina secuencialmente el vector hasta que se encuentra una celda vacía.

factor de carga Número de elementos de la tabla hash dividido por el tamaño del vector de la tabla hash. En una tabla hash con exploración, el factor de carga toma valores entre 0 (vacía) y 1 (llena). En el hashing enlazado puede ser mayor que 1.

función hash Función que convierte un elemento en un entero adecuado para indexar la posición del vector en el que se almacena dicho elemento. Si la función hash es inyectiva, podemos acceder al elemento a través de su índice en el vector. Normalmente no son inyectivas, por lo que varios elementos pueden colisionar en el mismo índice.

hashing Implementación de las tablas hash para realizar inserciones, eliminaciones y búsquedas.

hashing doble Técnica de hashing que no sufre de agrupación secundaria. Para la resolución de conflictos emplea una segunda función hash.

hashing enlazado Alternativa eficiente, hablando en términos de espacio, a la exploración cuadrática, en la que se mantiene un vector de listas enlazadas. Es menos sensible a factores de carga elevados. Presenta algunos de los inconvenientes de las implementaciones de las pilas basadas en vectores y en listas enlazadas.

tabla hash Tabla empleada para implementar un diccionario cuyas operaciones se realizan en tiempo constante.

Errores comunes

1. La función de localización devuelve un elemento de tipo `int`. Debido a que los cálculos intermedios pueden provocar un desbordamiento, la variable local debe comprobar que el resultado de la operación de módulo no es negativo, evitando así la devolución de un valor fuera de rango.
2. La eficiencia de una tabla hash con exploración empeora sensiblemente a medida que el factor de carga se aproxima a 1,0. No debe permitirse que esto suceda, haciendo un rehashing cuando el factor de carga se aproxime a 0,5.
3. La eficiencia de todos los métodos de hashing depende del uso de una buena función de localización. Un error muy usual es emplear una función inadecuada.

En Internet

La tabla hash con exploración cuadrática se encuentra en el directorio **DataStructures**. En el directorio **Chapter19** podemos encontrar una colección de ficheros para generar la Figura 19.5.

HashEntry.java	Contiene la implementación de `EntradaHash`.
ProbingHashTable.java	Contiene la implementación de la clase abstracta de las tablas hash con exploración. Es la versión inglesa de la clase `ExploracionTablaHash`.

QuadraticProbingTable.java	Contiene la implementación de las tablas hash con exploración cuadrática. Es la versión inglesa de la clase `TablaExploracionCuadratica`.

Ejercicios

Cuestiones breves

19.1. ¿Cuáles son los índices del vector de una tabla hash de tamaño 11?

19.2. ¿Cuál es el tamaño apropiado de una tabla hash si el número de elementos que hay en ella es 10?

19.3. Explique cómo se realizaría la eliminación en las tablas con exploración y con hashing enlazado.

19.4. ¿Cuál es el número esperado de pruebas en una búsqueda con y sin éxito en una tabla hash con exploración lineal y factor de carga 0,25?

19.5. Dada la entrada (4.371, 1.323, 6.173, 4.199, 4.344, 9.679, 1.989), una tabla de tamaño fijo 10 y la función hash $H(X) = X \bmod 10$, muestre

a) la tabla hash con exploración lineal resultante.
b) la tabla hash con exploración cuadrática resultante.
c) la tabla con hashing enlazado resultante.

19.6. Muestre el resultado de hacer un rehashing en cada una de las tablas del Ejercicio 19.5. El tamaño de la nueva tabla debe ser un número primo.

19.7. La rutina `esVacia` aún no ha sido implementada. Impleméntela de modo que devuelva la expresión `tamanyoActual == 0`.

Problemas teóricos

19.8. Una estrategia alternativa de resolución de conflictos consiste en definir una secuencia, $F(i) = R_i$, donde $R_0 = 0$ y $R_1, R_2, \ldots, R_{M-1}$ es una secuencia aleatoria de los $M - 1$ primeros enteros (recuerde que el tamaño de la tabla es M).

a) Demuestre que bajo esta estrategia, las colisiones pueden resolverse siempre que la tabla no esté llena.
b) ¿Podría eliminar esta estrategia la agrupación primaria?
c) ¿Podría eliminar esta estrategia la agrupación secundaria?
d) Si el factor de carga de la tabla es λ, ¿cuál es el coste en tiempo de una inserción?
e) La generación de una permutación aleatoria mediante el algoritmo de la Sección 9.4 implica un gran número de (costosas) llamadas al generador de números aleatorios. Dé un algoritmo eficiente para la generación de permutaciones aleatorias que evite las llamadas al generador de números aleatorios.

19.9. Si el rehashing se produce tan pronto como el factor de carga se aproxima a 0,5, en adelante cuando se inserte un nuevo elemento, el factor de carga estará entre 0,25 y 0,5. ¿Cuál es el factor de carga esperado en esta situación? En concreto, ¿es o no cierto que el factor de carga medio es 0,375?

19.10. Cuando se ejecuta el proceso del rehashing, se hacen $O(N)$ intentos para insertar N elementos. Dé una estimación del número de intentos (es decir, N o $2N$ o algo similar) necesario. *Indicación:* Calcule el coste medio de una inserción en la nueva tabla. En dichas inserciones el factor de carga varía desde 0 a 0,25.

19.11. Bajo ciertas hipótesis, el coste esperado de una inserción en una tabla hash con agrupación secundaria viene dado por $1/(1 - \lambda) - \lambda - \ln(1 - \lambda)$. Desgraciadamente, esta fórmula no es ajustada para la exploración cuadrática. Pese a ello, asumiendo que lo fuera, determine lo siguiente:

a) El coste esperado de una búsqueda sin éxito.
b) El coste esperado de una búsqueda exitosa.

19.12. Empleamos una tabla hash con exploración cuadrática para almacenar 10.000 objetos de tipo `String`. Supongamos que el factor de carga es 0,4 y que la longitud media de las cadenas es 8. Determine lo siguiente:

a) El tamaño de la tabla hash.
b) La cantidad de memoria empleada para almacenar los 10.000 objetos de tipo `String`.
c) La cantidad de memoria adicional empleada por la tabla hash.
d) La memoria total que necesita la tabla hash.
e) El coste en espacio.

Problemas prácticos

19.13. Implemente la exploración lineal.

19.14. Implemente en la clase de las tablas hash con exploración, el código del rehashing sin realizar ninguna llamada recursiva a `insertar`.

19.15. Experimente con la tabla hash que examina un carácter aleatorio en una cadena. ¿Es ésta una opción mejor que la adoptada en el texto?

19.16. Modifique la clase de las tablas hash de modo que la operación `esVacia` se ejecute en tiempo constante.

19.17. Modifique el algoritmo de eliminación de modo que el factor de carga siempre se encuentre por debajo de 1/8, y se ejecute el rehashing para mantener la tabla medio llena. Debe mantenerse un campo de datos adicional. ¿Por qué?

Prácticas de programación

19.18. Elija un diccionario de gran tamaño. Cree una tabla hash dos veces más grande que el diccionario. Aplique a cada palabra la función de localización descrita en el texto, almacénela y cuente el número de veces que se indexa cada posición. De este modo se obtendrá una distribución. Cierto porcentaje de las posiciones no se indexará nunca, algunas lo serán una vez, otras dos veces y así sucesivamente. Compare esta distribución con la que se obtiene con valores aleatorios, uniformemente distribuidos, utilizando la discusión sobre ellos de la Sección 9.3.

19.19. Realice simulaciones para comparar la eficiencia del hashing con los resultados teóricos. Declare una tabla hash con exploración. Inserte en ella 10.000 enteros generados aleatoriamente y calcule el número medio de pruebas realizadas. Éste es el coste medio de una búsqueda exitosa. Repita el test varias veces para obtener una media ajustada y realice las pruebas para las exploraciones lineal y cuadrática, con factores de carga 0,1, 0,2, ..., 0,9. Debe declararse la tabla de modo que nunca sea necesario el rehashing. Por ejemplo, en el test con factor de carga 0,4 debería declararse una tabla de tamaño aproximado 25.000 (ajústese para que sea un número primo).

19.20. Compare el tiempo necesario para las búsquedas exitosas y las inserciones en una tabla hash con hashing enlazado y factor de carga 1 y una tabla hash con exploración cuadrática y factor de carga 0,5. Realice las pruebas con enteros, cadenas y registros complicados en los que la clave sea una cadena.

19.21. Un programa en BASIC consiste en una serie de instrucciones, numeradas todas ellas en orden ascendente. El control se transmite empleando una instrucción *goto* o *gosub* y un número de instrucción. Escriba un programa que lea un programa correcto en BASIC y numere de nuevo sus instrucciones de modo que la primera empiece en el número F y cada instrucción tenga un número D mayor que la anterior. Debe asumirse un límite máximo de N instrucciones, pero los números de las mismas en la entrada podrían ser tan grandes como un entero de 32 bits. Su programa debe ejecutarse en tiempo lineal.

19.22. Diseñe un applet que ilustre el funcionamiento de la exploración cuadrática.

Bibliografía

A pesar de la aparente simplicidad del hashing, gran parte de su análisis es muy complicado y aún hay muchas cuestiones sin resolver. Existen diversos temas teóricos muy interesantes en relación con el intento de disminuir la probabilidad del mal funcionamiento del método de hashing.

Un artículo pionero sobre rehashing es [11]. En [6] puede encontrarse valiosa información acerca de este tema, incluyendo un análisis del hashing cerrado con exploración lineal. El hashing doble se analiza en [5] y [7]. Otro esquema de resolución de conflictos es el *hashing coalescente*, descrito en [12]. Un excelente resumen sobre el tema puede encontrarse en [8]; [9] realiza sugerencias y advertencias sobre la elección de las funciones de localización. En [4] pueden encontrarse precisos resultados analíticos y de simulación acerca de todos los métodos descritos en este capítulo. Yao [13] muestra que el hashing uniforme, en el que no existe ningún tipo de agrupación, es óptimo respecto al coste de una búsqueda exitosa.

Si se conocen con antelación las claves de entrada, entonces existe una función hash perfecta que evita cualquier colisión [1]. Otros esquemas de hashing algo más complicados, en los que el caso peor no sólo depende de la entrada sino también de ciertos números aleatorios elegidos, aparecen en [2] y [3]. Estos esquemas garantizan que en el caso peor se produce un número constante de colisiones

(aunque la elaboración de la función hash puede necesitar gran cantidad de tiempo en el improbable caso de que se utilicen números aleatorios inadecuados). Son muy útiles en la implementación en hardware de las tablas hash.

En [10] puede encontrarse otro método para resolver el Ejercicio 19.8.

1. J. L. Carter y M. N. Wegman, «Universal Classes of Hash Functions», *Journal of Computer and System Sciences* **18** (1979), 143-154.
2. M. Dietzfelbinger, A. R. Karlin, K. Melhorn, F. Meyer auf def Heide, H. Rohnert, y R. E. Tarjan, «Dynamic Perfect Hashing: Upper and Lower Bounds», *SIAM Journal on Computing* **23** (1994), 738-761.
3. R. J. Enbody y H. C. Du, «Dynamic Hashing Schemes», *Computing Surveys* **20** (1988), 85-113.
4. G. H. Gonnet y R. Baeza-Yates, *Handbook of Algorithms and Data Structures*, 2.ª ed., Addison-Wesley, Reading, Mass. (1991).
5. L. J. Guibas y E. Szemeredi, «The Analysis of Double Hashing», *Journal of Computer and System Sciences* **16** (1978), 226-274.
6. D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching,* 2.ª ed., Addison-Wesley, Reading, Mass. (1997).
7. G. Lueker y M. Molodowitch, «More Analysis of Double Hashing», *Combinatorica* **13** (1993), 83-96.
8. W. D. Maurer y T. G. Lewis, «Hash Table Methods», *Computing Surveys* **7** (1975), 5-20.
9. B. J. McKenzie, R. Harries, y T. Bell, «Selecting a Hashing Algorithm», *Software-Practice and Experience* **20** (1990), 209-224.
10. R. Morris, «Scatter Storage Techniques», *Communications of the ACM* **11** (1968), 38-44.
11. W. W. Peterson, «Addressing for Random Access Storage», *IBM Journal of Research and Development* **1** (1957), 130-146.
12. J. S. Vitter, «Implementations for Coalesced Hashing», *Information Processing Letters* **11** (1980), 84-86.
13. A. C. Yao, «Uniform Hashing Is Optimal», *Journal of the ACM* **32** (1985), 687-693.