Capítulo 20

Tablas hash

En el Capítulo 19 hemos hablado del árbol de búsqueda binaria que nos permite realizar diversas operaciones sobre un conjunto de elementos. En este capítulo vamos a ver la tabla hash, que solo soporta un subconjunto de las operaciones permitidas por los árboles de búsqueda binaria. La implementación de las tablas hash se denomina frecuentemente *hashing*, e incluye inserciones, borrados y búsquedas en un tiempo medio constante.

A diferencia del árbol de búsqueda binaria, el tiempo de ejecución promedio de las operaciones con una tabla hash se basa en propiedades estadísticas, más que en las expectativas de disponer de entradas con apariencia aleatoria. Esta mejora se obtiene a expensas de un pérdida de la información de ordenación correspondiente a los elementos: no se soportan en un tiempo lineal operaciones tales como `findMin` y `findMax` o la impresión de una tabla completa en orden. En consecuencia, la tabla hash y el árbol de búsqueda binaria tienen aplicaciones y propiedades de rendimiento algo diferentes.

En este capítulo veremos

- Varios métodos de implementar la tabla hash.
- Comparaciones analíticas entre estos métodos.
- Algunas aplicaciones de las tablas hash.
- Comparaciones entre las tablas hash y los árboles de búsqueda binaria.

20.1 Ideas básicas

La *tabla hash* soporta la extracción o borrado de cualquier elemento nominado. Queremos podemos soportar las operaciones básicas en un tiempo constante, al igual que para la pila y la cola. Puesto que los accesos están mucho menos restringidos, este soporte parece casi un objetivo imposible de alcanzar. Es decir, con toda seguridad, cuando el tamaño del conjunto se incremente, las búsquedas en el mismo deberán requerir más tiempo. Sin embargo, veremos que esto no es necesariamente así.

La *tabla hash* se utiliza para implementar un conjunto en tiempo constante por cada operación.

Suponga que todos los elementos con los que estamos tratando son enteros pequeños no negativos, comprendidos entre 0 y 65.535. Podemos utilizar una matriz simple para implementar

cada operación de la forma siguiente. En primer lugar, inicializamos una matriz `a` indexada entre 0 y 65.535 con todo 0s. Para realizar `insert(i)`, ejecutamos `a[i]++`. Observe que `a[i]` representa el número de veces que `i` ha sido insertado. Para realizar `find(i)`, verificamos que `a[i]` no sea 0. Para realizar `remove(i)`, nos aseguramos de que `a[i]` sea positivo y luego ejecutamos `a[i]--`. El tiempo requerido por cada operación es obviamente constante; incluso el coste adicional de inicialización de la matriz requiere una cantidad constante de trabajo (65.536 asignaciones).

Hay dos problemas con esta solución. En primer lugar, suponga que tuviéramos enteros de 32 bits, en lugar de enteros de 16 bits. Entonces, la matriz `a` debería contener 4.000 millones de elementos, lo que resulta complicado desde el punto de vista práctico. En segundo lugar, si los elementos no son enteros, sino cadenas de caracteres (o incluso algo más genérico), no pueden utilizarse para indexar una matriz. El segundo problema no es, en realidad, un problema en absoluto. Al igual que un número `1234` es una colección de dígitos `1`, `2`, `3` y `4`, la cadena de caracteres `"junk"` es una colección de caracteres `'j'`, `'u'`, `'n'` y `'k'`. Observe que el número `1234` es simplemente $1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0$. Recuerde de la Sección 12.1 que un carácter ASCII se puede representar típicamente con 7 bits como un número comprendido entre 0 y 127. Puesto que un carácter es básicamente un entero de pequeño tamaño, podemos interpretar una cadena de caracteres como un entero. Una posible representación sería $\texttt{'j'} \cdot 128^3 + \texttt{'u'} \cdot 128^2 + \texttt{'n'} \cdot 128^1 + \texttt{'k'} \cdot 128^0$. Esta técnica nos permitiría emplear la implementación basada en una matriz simple que hemos analizado anteriormente.

Una *función hash* convierte el elemento en un entero que sea adecuado para indexar una matriz en la que se almacene el elemento. Si la función hash fuera biunívoca podríamos acceder al elemento empleando su índice matricial.

El problema con esta estrategia es que la representación en forma de entero que acabamos de describir nos proporciona números enteros de un tamaño enorme. La representación para `"junk"` nos da 224.229.227, y las cadenas de mayor longitud generan representaciones todavía mayores. Este resultado nos devuelve al primero de los problemas mencionados: ¿cómo evitamos utilizar una matriz de un tamaño absurdamente grande?

Para hacerlo, empleamos una función que asigna los números de gran tamaño (o las cadenas de caracteres interpretadas como números) a números más pequeños y manejables. Una función que establezcan una correspondencia entre un elemento y un índice de pequeño tamaño se conoce con el nombre de *función hash*. Si `x` es un entero arbitrario (no negativo), entonces `x%tableSize` genera un número comprendido entre 0 y `tableSize-1` que es adecuado para indexar en una matriz de tamaño `tableSize`. Si `s` es una cadena de caracteres, podemos convertir `s` en un entero de gran tamaño `x` utilizando el método sugerido anteriormente y luego aplicar el operador mod (`%`) para obtener un índice adecuado. Así, si `tableSize` es 10.000, `"junk"` se indexaría como 9.227. En la Sección 20.2 hablaremos detalladamente de la implementación de la función hash para cadenas de caracteres.

El uso de la función hash introduce una complicación: puede que haya dos o más elementos distintos a los que les corresponda la misma posición, provocando una *colisión*. Esta situación no se puede evitar nunca, porque hay muchos más elementos que posiciones disponibles. Sin embargo, existen muchos métodos para resolver rápidamente una colisión. Investigaremos tres de los métodos más sencillos: sondeo lineal, sondeo cuadrático y encadenamiento separado. Cada uno de estos métodos es simple de implementar, pero sus respectivos rendimientos son distintos, dependiendo de lo llena que esté la matriz.

Puesto que la función hash no es biunívoca, varios elementos se corresponderán con un mismo índice y provocarán una *colisión*.

20.2 Función hash

Calcular la función hash para cadenas de caracteres presenta una sutil complicación: la conversión de la `String s a x` genera un entero que casi con seguridad será mayor que los valores que la máquina permite almacenar, porque $128^4 = 2^{28}$. Este tamaño de entero es solo 8 veces menor que el valor `int` más grande permitido. En consecuencia, no podemos esperar obtener el valor de la función hash calculando una potencia de 128. En lugar de ello, utilizamos la siguiente observación. Un polinomio genérico

$$A_3X^3 + A_2X^2 + A_1X^1 + A_0X^0 \tag{20.1}$$

puede evaluarse como

$$(((A_3)X + A_2)X + A_1)X + A_0 \tag{20.2}$$

Utilizando un truco podemos evaluar la función hash de manera eficiente y sin desbordamiento.

Observe que en la Ecuación 20.2 evitamos calcular el polinomio directamente, lo cual resulta conveniente por tres razones distintas. En primer lugar, evita tener un resultado intermedio de gran tamaño, el cual, como ya hemos visto, provocaría un desbordamiento. En segundo lugar, el cálculo de esa ecuación implica únicamente tres multiplicaciones y tres sumas; un polinomio de grado N se calcula en N multiplicaciones y sumas. Estas operaciones son menos costosas que el cálculo de la Ecuación 20.1. En tercer lugar, el cálculo se hace de izquierda a derecha (A_3 se corresponde con `'j'`, A_2 se corresponde con `'u'`, etc. y X es 128).

Sin embargo, sigue existiendo un problema de desbordamiento. El resultado del cálculo sigue siendo el mismo y es probable que sea demasiado grande. Pero aquí hay que recordar que solo necesitamos el resultado mod `tableSize`. Aplicando el operador `%` después de cada multiplicación (o suma), podemos cerciorarnos de que los resultados intermedios sigan siendo pequeños.[1] La función hash resultante se muestra en la Figura 20.1. Una desventaja de esta función hash es que el cálculo del operador mod es costosa. Pero, como el desbordamiento está permitido (y sus resultados son coherentes en una plataforma determinada), podemos acelerar un poco más la función hash realizando una única operación mod inmediatamente antes de volver. Lamentablemente, la multiplicación repetida por 128 tendería a desplazar los primeros caracteres hacia la izquierda, fuera de la respuesta. Para aliviar este problema, multiplicamos por 37 en lugar de por 128, lo que ralentiza el desplazamiento de los primeros caracteres.

La función hash debe ser simple de calcular, pero también debe distribuir las claves de manera equitativa. Si hay demasiadas colisiones, el rendimiento de la tabla hash se verá enormemente afectado.

El resultado se muestra en la Figura 20.2. no se trata necesariamente de la mejor de las funciones posibles. Asimismo, en algunas aplicaciones (por ejemplo, si estamos utilizando cadenas de caracteres largas), nos gustaría mejorarla. Sin embargo, hablando en términos generales, la función es bastante buena. Observe que el desbordamiento podría introducir números negativos. Por ello, si la operación mod genera un valor negativo, lo hacemos positivo (líneas 15 y 16). Observe también que el resultado obtenido al permitir el desbordamiento y realizar una operación mod al final no coincide

[1] La Sección 7.4 habla de las propiedades de la operación mod.

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

Figura 20.1 Un primer intento de implementar una función hash.

con el que se obtiene al realizar la operación mod después de cada paso. Por tanto, hemos modificado ligeramente la función hash, lo que no es un problema.

Aunque la velocidad es un aspecto importante a la hora de diseñar una función hash, también queremos asegurarnos de que distribuye las claves de manera equitativa. En consecuencia, no debemos llevar demasiado lejos nuestras optimizaciones. Un ejemplo sería la función hash mostrada en la Figura 2.3, que simplemente suma los caracteres que forman la clave y devuelve el resultado mod `tableSize`. Difícilmente encontraríamos una función más simple. La función es fácil de implementar y calcula un valor hash muy rápidamente. Sin embargo, si `tableSize` es grande, la función no distribuye las claves demasiado bien. Por ejemplo, suponga que `tableSize` es 10.000.

```
1    /**
2     * Una rutina hash para objetos String.
3     * @param key el String al que hay que aplicar el hash.
4     * @param tableSize el tamaño de la tabla hash.
5     * @return el valor hash.
6     */
7    public static int hash( String key, int tableSize )
8    {
9       int hashVal = 0;
10
11      for( int i = 0; i < key.length( ); i++ )
12         hashVal = 37 * hashVal + key.charAt( i );
13
14      hashVal %= tableSize;
15      if( hashVal < 0 )
16         hashVal += tableSize;
17
18      return hashVal;
19   }
```

Figura 20.2 Una función hash más rápida que aprovecha el mecanismo de desbordamiento.

```
1    // Una función hash inadecuada si tableSize es grande.
2    public static int hash( String key, int tableSize )
3    {
4       int hashVal = 0;
5
6       for( int i = 0; i < key.length( ); i++ )
7          hashVal += key.charAt( i );
8
9       return hashVal % tableSize;
10   }
```

Figura 20.3 Una función hash inadecuada si `tableSize` es grande.

Suponga también que todas las claves tienen una longitud de 8 caracteres o menos. Puesto que un `char` ASCII es un entero entre 0 y 127, la función hash solo puede asumir valores comprendidos entre 0 y 1.016 (127 × 8). Esta restricción no permite, obviamente, una distribución equitativa. Cualquier velocidad que hayamos ganado al acelerar el cálculo de la función hash se verá más que compensado por el esfuerzo necesario para resolver un número de colisiones mayor que el esperado. Sin embargo, en el Ejercicio 20.15 se describe una alternativa razonable.

La tabla va de 0 a `tableSize - 1`.

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

20.2.1 `hashCode` en `java.lang.String`

En Java, los tipos de la librería que se pueden insertar razonablemente en un `HashSet` o que se pueden utilizar como clave en un `HashMap` ya tienen definidos los métodos `equals` y `hashCode`. En particular, la clase `String` tiene un `hashCode` cuya implementación resulta crítica para el rendimiento de los `HashSet` y `HashMap` donde están involucrados objetos `String`.

La historia del método `hashCode` de `String` es en sí misma bastante instructiva. Las primeras versiones de Java utilizaban esencialmente la misma implementación de la Figura 20.2, incluyendo el multiplicador constante 37, pero sin las líneas 14 a 16. Pero posteriormente se "optimizó" la implementación de modo que si la cadena `String` tenía una longitud mayor de 15 caracteres, solo se utilizaban para calcular el `hashCode` 8 o 9 caracteres, espaciados de forma más o menos equitativa dentro del objeto `String`. Esta versión se utilizó en Java 1.0.2 y Java 1.1, pero resultó ser una mala idea, porque había muchas aplicaciones que contenían grandes grupos de objetos `String` de gran longitud que eran en cierta forma similares. Dos de esos ejemplos eran los mapas en los que se utilizaban como claves direcciones URL como `http://www.cnn.com/` y los mapas en los que se empleaban nombres completos de archivo como

```
/a/file.cs.fiu.edu./disk/storage137/user/weiss/public_html/dsj4/code.
```

El rendimiento en estos mapas se degradaba considerablemente, porque las claves tendían a generar un número relativamente bajo de códigos hash distintos.

En Java 1.2, se volvió al método `hashCode` de la versión más simple, empleando 31 como multiplicador constante. No hace falta decir que los programadores que diseñaron la librería Java se cuentan entre los de más talento del mundo, de modo que es fácil comprender que diseñar una función hash de primera categoría es un proceso lleno de trampas y no tan simple como puede parecer.

En Java 1.3, se probó una nueva idea con algo más de éxito. Puesto que la parte más costosa de las operaciones con tablas hash es el cálculo del código `hashCode`, el método `hashCode` de la clase `String` contiene una optimización importante: cada objeto `String` almacena internamente el valor de su `hashCode`. Inicialmente es 0, pero si se invoca `hashCode`, se recuerda posteriormente el valor obtenido. De ese modo, si tratamos de calcular una segunda vez `hashCode` con el mismo objeto `String`, podemos evitar volver a realizar los costosos cálculos. Esta técnica se denomina *almacenamiento en caché del código hash*, y representa otro de esos clásicos compromisos entre tiempo y espacio. La Figura 20.4 muestra una implementación de la clase `String` en la que se almacena en caché el código hash.

El almacenamiento en caché del código hash funciona únicamente porque los objetos `String` son inmutables: si permitiéramos que el objeto `String` cambiara, se invalidaría el valor de `hashCode` y tendríamos que volver a asignar a este un valor 0. Aunque está claro que para dos objetos `String` con el mismo estado hay que calcular de manera independiente su código hash, también hay muchas situaciones en las que se consulta una y otra vez el código hash de un mismo objeto `String`. Una situación en la que sirve de ayuda el almacenamiento en caché del código hash es durante el recálculo de la distribución hash de los elementos, porque todos los objetos `String` implicados en ese recálculo ya tendrán almacenados en caché sus códigos hash.

20.3 Sondeo lineal

En el *sondeo lineal*, las colisiones se resuelven explorando secuencialmente una matriz (con encadenamiento circular entre el final y el principio), hasta encontrar una celda vacía.

Ahora que disponemos de una función hash, tenemos que decidir qué hacer cuando se produce una colisión. Específicamente, si al aplicar la función hash a X obtenemos una posición que ya está ocupada, ¿dónde colocamos ese elemento? La estrategia más simple posible es la del *sondeo lineal*, que consiste en buscar secuencialmente en la matriz hasta encontrar una celda vacía. La búsqueda salta circularmente desde la última posición hasta la primera en caso necesario. La Figura 20.5 muestra el resultado de insertar las claves 89, 18, 49, 58 y 9 en una tabla hash, cuando se utiliza un sondeo lineal. En el ejemplo estamos asumiendo que disponemos de una función hash que devuelve la clave X mod el tamaño de la tabla. La Figura 20.5 incluye el resultado de la función hash.

La primera colisión se produce al insertar 49; el 49 se coloca en la siguiente celda disponible –en concreto, en la celda 0, que está vacía. Después, 58 colisiona con 18, 89 y 49 antes de encontrar una celda vacía, tres posiciones más allá, en la posición 1. La colisión para el elemento 9 se resuelve de forma similar. Mientras que la tabla sea lo suficientemente grande, siempre se podrá encontrar una celda vacía. Sin embargo, el tiempo necesario para encontrar una celda vacía puede llegar a ser muy largo. Por ejemplo, si solo queda una celda vacía en la tabla, puede que tengamos que realizar una búsqueda de tabla completa para encontrarlo. En promedio, cabría esperar tener que explorar la mitad de la tabla para encontrarlo, lo cual está bastante lejos del tiempo constante por acceso que estábamos intentando obtener. Pero si la tabla está relativamente vacía, las inserciones no deberían ser demasiado costosas. Enseguida analizaremos esta técnica.

```
1    public final class String
2    {
3       public int hashCode( )
4       {
5          if( hash != 0 )
6          return hash;
7
8          for( int i = 0; i < length( ); i++ )
9             hash = hash * 31 + (int) charAt( i );
10         return hash;
11      }
12
13      private int hash = 0;
14   }
```

Figura 20.4 Extracto del método `hashCode` de la clase `String`.

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

	Después de insertar 89	*Después de insertar 18*	*Después de insertar 49*	*Después de insertar 58*	*Después de insertar 9*
0			49	49	49
1				58	58
2					9
3					
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Figura 20.5 Tabla hash con sondeo lineal después de cada inserción.

El algoritmo `find` simplemente sigue la misma técnica que el algoritmo `insert`. Si llega a una celda vacía, querrá decir que el elemento que estamos buscando no se ha encontrado; en caso contrario, terminará por encontrar la correspondencia deseada. Por ejemplo, para encontrar 58, comenzamos por la celda 8 (tal como nos ha indicado la función hash). Vemos allí un elemento, pero no es el que estamos buscando, así que probamos con la celda 9. De nuevo, tenemos

El algoritmo `find` sigue la misma secuencia de sondeo que el algoritmo `insert`.

un elemento, pero no es el deseado, así que buscamos en la celda 0 y luego en la celda 1, hasta encontrar una correspondencia. Una operación `find` para el valor 19 implicaría probar las celdas 9, 0, 1 y 2 antes de encontrar una posición vacía en la celda 3. Eso querrá decir que no se ha encontrado el valor 19.

No se puede realizar un borrado estándar porque, como sucede con los árboles de búsqueda binaria, cada elemento de la tabla hash no solo se representa a sí mismo, sino que también conecta entre sí otros elementos, sirviendo como posible celda de almacenamiento durante la resolución de colisiones. Por tanto, si eliminamos 89 de la tabla hash, casi todas las restantes operaciones `find` fallarán. En consecuencia, lo que hacemos es implementar un mecanismo de *borrado perezoso*, que consiste en marcar los elementos como borrados en lugar de eliminarlos físicamente de la tabla. Esta información se almacena en un miembro de datos adicional. Cada uno de los elementos de la tabla podrá estar *activo* o *borrado*.

Se debe utilizar un *borrado perezoso*.

20.3.1 Un análisis simplista del sondeo lineal

El análisis simplista del sondeo lineal se basa en la suposición de que los sondeos sucesivos son independientes. Esta suposición no es cierta y, por tanto, el análisis subestima el coste de las búsquedas y de las inserciones.

Para estimar el rendimiento del sondeo lineal, hacemos dos suposiciones:

1. La tabla hash es de gran tamaño.
2. Cada sondeo de la tabla hash es independiente del sondeo anterior.

La primera suposición es razonable; en caso contrario, estaríamos perdiendo el tiempo con una tabla hash. La segunda suposición dice que si la fracción de la tabla que está llena es λ, cada vez que examinemos una celda la probabilidad de que esté ocupada también será λ, independientemente de las operaciones de sondeo anteriores. La independencia es una importante propiedad estadística, que simplifica enormemente el análisis de sucesos aleatorios. Lamentablemente, como se explica en la Sección 20.3.2, la suposición de independencia no solo no está justificada, sino que es totalmente errónea. Por tanto, el análisis simplista que vamos a realizar es incorrecto. Aun así es útil realizarlo porque nos dice lo que cabe esperar conseguir si tenemos más cuidado con respecto al modo de resolver las colisiones. Como hemos mencionado anteriormente en el capítulo, el rendimiento de la tabla hash dependerá de lo llena que esté. El grado en el que la tabla esté llena vendrá dado por el denominado factor de carga.

El *factor de carga* de una tabla hash con sondeo es la fracción de la tabla que está llena. Varía entre 0 (vacía) y 1 (llena).

Definición: El *factor de carga*, λ, de una tabla hash con sondeo es la fracción de la tabla que está llena. El factor de carga varía entre 0 (vacía) y 1 (completamente llena).

Con esto podemos proporcionar un análisis simple, pero incorrecto, del sondeo lineal en el Teorema 20.1.

Teorema 20.1	Si asumimos que los sondeos son independientes, el número medio de celdas examinadas durante una inserción utilizando sondeo lineal es $1/(1-\lambda)$.
Demostración	Para un tabla con un factor de carga igual a λ, la probabilidad de que cualquier celda esté vacía es $1-\lambda$. En consecuencia, el número esperado de intentos independientes requeridos para encontrar una celda vacía será $1/(1-\lambda)$.

En la demostración del Teorema 20.1 utilizamos el hecho de que, si la probabilidad de que un cierto suceso ocurra es p, entonces se requiere en promedio $1/p$ intentos hasta que se produce el suceso, suponiendo que los intentos sean independientes entre sí. Por ejemplo, el número esperado de veces que tendremos que arrojar una moneda hasta obtener cara es dos, y el número esperado de veces que tendremos que arrojar un dado de seis caras hasta obtener un 4 es seis, suponiendo que los intentos sean independientes.

20.3.2 Lo que realmente sucede: agrupamiento primario

El efecto del *agrupamiento primario* es la formación de grandes agrupamientos de celdas ocupadas que hacen que las inserciones en el agrupamiento sean costosas (y además la inserción hace que el agrupamiento sea aun mayor).

Lamentablemente, la suposición de independencia no se cumple como se muestra en la Figura 20.6. La parte (a) de la figura muestra el resultado de rellenar una tabla hash hasta un 70 por ciento de capacidad, si todos los sondeos sucesivos son independientes. La parte (b) muestra el resultado de la técnica de sondeo lineal. Observe la serie de agrupamientos. Este fenómeno se conoce con el nombre de *agrupamiento primario* o *clustering primario*.

En el fenómeno del agrupamiento primario se forman grandes bloques de celdas ocupadas. Cualquier clave cuya función hash la asigne a uno de estos agrupamientos requerirá un número excesivo de intentos para resolver la colisión, después de lo cual contribuirá a aumentar el tamaño del agrupamiento. No solo los elementos que colisionan debido a que los resultados de aplicarles las funciones hash coinciden provocan un rendimiento degenerado, sino que también un elemento que colisione con una posición alternativa para otro elemento hará que el rendimiento se reduzca. El análisis matemático requerido para tener en cuenta este fenómeno es complejo, pero ha sido resuelto, proporcionándonos el Teorema 20.2.

Teorema 20.2 El número medio de celdas examinadas en una inserción utilizando el sondeo lineal es aproximadamente $(1 + 1/(1 - \lambda)^2)/2$.

Demostración La demostración queda fuera del alcance de este texto. Consulte la referencia **[6]**.

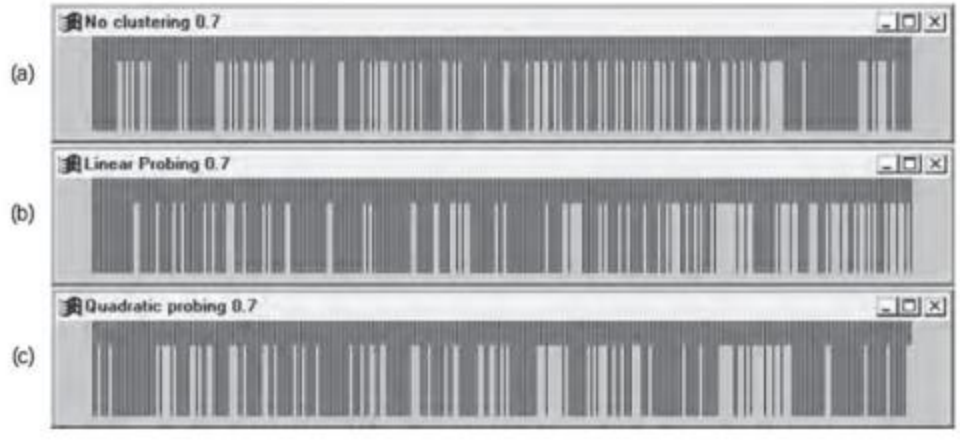

Figura 20.6 Ilustración del fenómeno del agrupamiento primario en el sondeo lineal (b), comparado con la ausencia de agrupamiento (a) y el agrupamiento secundario, menos significativo en el sondeo cuadrático (c). Las líneas representan celdas ocupadas y el factor de carga es 0,7.

La agrupación primaria es un problema para factores de carga altos. Para tablas medio llenas el efecto no es desastroso.

Para una tabla semillena, obtenemos 2,5 como número medio de celdas examinadas durante una inserción. Este resultado es casi el mismo que lo que indicaba el análisis simplista. La principal diferencia se manifiesta a medida que λ se aproxima a 1. Por ejemplo, si la tabla está llena al 90 por ciento, $\lambda = 0{,}9$. El análisis simplista sugiere que habría que examinar 10 celdas –lo que es bastante, pero tampoco excesivo. Sin embargo, según el Teorema 20.2, la respuesta real es que hará falta examinar unas 50 celdas. Eso es excesivo (especialmente porque este número es solo un promedio, con lo que algunas inserciones deben ser aun peores).

20.3.3 Análisis de la operación `find`

Una operación `find` que no tenga éxito cuesta lo mismo que una inserción.

El coste de una operación `find` que tenga éxito es un promedio de los costes de inserción para todos los factores de carga más pequeños.

El coste de una inserción se puede utilizar para obtener una cota del coste de una operación `find`. Hay dos tipos de operación `find`: las que tienen éxito y las que no. Una operación `find` que no tenga éxito es fácil de analizar. La secuencia de celdas examinadas para una búsqueda sin éxito del elemento X es la misma que la secuencia examinada para insertar X con la operación `insert`. Por tanto, tenemos una respuesta inmediata en lo que se refiere al coste de una operación `find` que no tenga éxito.

Para las operaciones `find` que sí tengan éxito, las cosas son ligeramente más complicadas. La Figura 20.5 muestra una tabla con $\lambda = 0{,}5$. Así, el coste medio de una inserción es 2,5. El coste medio para encontrar con `find` el elemento recién insertado sería entonces 2,5, independientemente de cuántas inserciones se produzcan después. El coste medio para encontrar el primer elemento insertado en la tabla será siempre 1,0. Por tanto, en una tabla con $\lambda = 0{,}5$, algunas búsquedas serán sencillas y otras más difíciles. En particular, el coste de una búsqueda del elemento X que tenga éxito es igual al coste de insertar X *en el momento en que* X *fue insertado*. Para calcular el tiempo medio requerido para realizar una búsqueda con éxito en una tabla con un factor de carga λ, debemos calcular el coste medio de inserción, promediando entre todos los factores de carga que han terminado dándonos un factor λ. Con este trabajo preliminar, podemos calcular los tiempos medios de búsqueda para el sondeo lineal, como se enuncia y se demuestra en el Teorema 20.3.

Teorema 20.3

El número medio de celdas examinadas en una búsqueda que no tenga éxito utilizando un sondeo lineal es aproximadamente $(1 + 1/(1 - \lambda)^2)/2$. El número medio de celdas examinadas en una búsqueda que tenga éxito es aproximadamente $(1 + 1/(1 - \lambda))/2$.

Demostración

El coste de una búsqueda sin éxito coincide con el coste de una inserción. Para una búsqueda con éxito, calculamos el coste medio de inserción para toda la secuencia de inserciones. Puesto que la tabla es grande, podemos calcular este promedio evaluando la integral

$$S(\lambda) = \frac{1}{\lambda} \int_{x=0}^{\lambda} I(x)\,dx$$

Continúa

Demostración (cont.)

En otras palabras, el coste medio de una búsqueda que tenga éxito para una tabla con un factor de carga λ es igual al coste de una inserción en una tabla con factor de carga x, promediado entre los factores de carga 0 a λ. A partir del Teorema 20.2, podemos obtener la siguiente ecuación:

$$\begin{aligned} S(\lambda) &= \frac{1}{\lambda}\int_{x=0}^{\lambda}\frac{1}{2}\left(1+\frac{1}{(1-x)^2}\right)dx \\ &= \frac{1}{2\lambda}\left(x+\frac{1}{(1-x)}\right)_{x=0}^{\lambda} \\ &= \frac{1}{2\lambda}\left(\left(\lambda+\frac{1}{(1-\lambda)}\right)-1\right) \\ &= \frac{1}{2}\left(\frac{2-\lambda}{1-\lambda}\right) \\ &= \frac{1}{2}\left(1+\frac{1}{(1-\lambda)}\right) \end{aligned}$$

Podemos aplicar la misma técnica para obtener el coste de una operación `find` con éxito utilizando la suposición de independencia entre sucesos (utilizando $I(x) = 1/(1 - x)$ en el Teorema 20.3). Si no se produce agrupamiento, el coste medio de una operación `find` con éxito para el caso de sondeo lineal es $-\ln(1 - \lambda)/\lambda$. **Si el factor de carga es 0,5, el número promedio de sondeos para** una búsqueda con éxito utilizando sondeo lineal es 1,5, mientras que el análisis sin agrupamiento sugiere 1,4 sondeos. Observe que este promedio no depende de la ordenación de las clases de entrada; depende únicamente de lo equitativa que sea la función hash. Observe también que, aun cuando dispongamos de buenas funciones hash, habrá secuencias de sondeo tanto cortas como largas que contribuyan al promedio. Por ejemplo, es seguro que existirán algunas secuencias de longitud 4, 5 y 6, incluso en una tabla hash que esté medio llena. (Determinar la secuencia de sondeo más larga esperada requiere cálculos enormemente complicados.) El fenómeno del agrupamiento primario no solo hace que la secuencia promedio de sondeos sea más larga, sino que también hace que una secuencia larga de sondeos sea más probable. El principal problema con el agrupamiento primario es, por tanto, que el rendimiento se degrada enormemente cuando se realizan inserciones con factores de carga altos. Asimismo, algunas de las secuencias de sondeo más largas con las que típicamente nos toparemos (las que se encuentran por encima de la media) también tienen una probabilidad mayor de presentarse.

Para reducir el número de sondeos, necesitamos un esquema de resolución de colisiones que evite la aparición del fenómeno del agrupamiento primario. Sin embargo, observe que si la tabla está medio llena, eliminar los efectos del agrupamiento primario solo permitiría ahorrar la mitad de un sondeo como promedio para una operación de inserción o una búsqueda que no tenga éxito, y que solo permitiría ahorrar un décimo de sondeo como promedio en una búsqueda que tenga éxito. Aunque cabría esperar que se reduzca la probabilidad de obtener una secuencia de sondeo algo más larga, *el sondeo lineal no es una estrategia tan mala*. Dado que es tan fácil de implementar, cualquier método que empleemos para eliminar el fenómeno del agrupamiento primario debe tener una complejidad comparable. En caso contrario, estaríamos invirtiendo demasiado tiempo en

ahorrar únicamente una fracción de sondeo. Uno de esos métodos alternativos sencillos es el *sondeo cuadrático*.

20.4 Sondeo cuadrático

El *sondeo cuadrático* examina las celdas que están situadas a una distancia de 1, 4, 9, etc. del punto de sondeo original.

El *sondeo cuadrático* es un método de resolución de colisiones que elimina el problema del agrupamiento primario que presenta el sondeo lineal. Lo consigue examinando determinadas celdas situadas a una cierta distancia del punto de sondeo original. Su nombre se deriva del uso de la fórmula $F(i) = i^2$ para resolver las colisiones. Específicamente, si la función hash se evalúa como H y una búsqueda en la celda H no permite finalizar la operación, probamos sucesivamente con las celdas $H + 1^2$, $H + 2^2$, $H + 3^2$, ..., $H + i^2$ (con encadenamiento circular entre el final y el principio de la tabla). Esta estrategia difiere de la estrategia de sondeo lineal consistente en buscar en $H + 1$, $H + 2$, $H + 3$, ..., $H + i$.

La Figura 20.7 muestra la tabla que resulta cuando se emplea el sondeo cuadrático en lugar del sondeo lineal para la secuencia de inserción mostrada en la Figura 20.5. Cuando 49 colisiona con 89, la primera alternativa que se intenta está situada a una celda de distancia. Esta celda está vacía, por lo que se coloca allí el 49. A continuación, el 58 colisiona en la posición 8. Se prueba con la celda situada en la posición 9 (que está a una celda de distancia), pero se produce otra colisión. Se encuentra una celda vacante en la siguiente celda que se prueba, que está a $2^2 = 4$ posiciones de distancia, con respecto a la posición hash original. De ese modo, el 58 se coloca en la celda 2. Lo mismo sucede con el 9. Observe que las posiciones alternativas para los elementos a los que les

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

	Después de insertar 89	Después de insertar 18	Después de insertar 49	Después de insertar 58	Después de insertar 9
0			49	49	49
1					
2				58	58
3					9
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Figura 20.7 Una tabla hash con sondeo cuadrático después de cada inserción (observe que el tamaño de la tabla se ha elegido descuidadamente, porque no es un número primo).

corresponde un resultado hash de 8 y las posiciones alternativas para los elementos a los que les corresponde un resultado hash de 9 no coincide. La larga secuencia de sondeo para insertar 58 no tuvo ningún efecto sobre la inserción subsiguiente de 9, lo que contrasta con lo que sucedía en el caso del sondeo lineal.

Recuerde que los puntos de sondeo posteriores están a un número cuadrático de posiciones de distancia con respecto al *punto de sondeo original*.

Tenemos que tomar en consideración algunos cuantos detalles antes de escribir el código.

- En el sondeo lineal, cada sondeo prueba con una celda diferente. ¿Está garantizado en el sondeo cuadrático que, cuando se pruebe celda, no la hayamos probado ya durante el curso del acceso actual? ¿Está garantizado en el sondeo cuadrático que, cuando estemos insertando X y la tabla no esté llena, terminaremos insertando X?
- El sondeo lineal se puede implementar fácilmente. El sondeo cuadrático parece requerir operaciones de multiplicación y de módulo. ¿Esta aparente complejidad añadida hace que el sondeo cuadrático resulte poco práctico?
- ¿Qué sucede (tanto en el sondeo lineal como en el cuadrático) si el factor de carga se hace demasiado alto? ¿Podemos expandir dinámicamente la tabla, como suele hacerse con otras estructuras de datos basadas en matrices?

Afortunadamente, las noticias son relativamente buenas en todos los casos citados. Si el tamaño de la tabla es primo y el factor de carga nunca es mayor de 0,5, siempre podremos insertar un nuevo elemento X y ninguna celda será sondeada dos veces durante un mismo acceso. Sin embargo, para que estas garantías se cumplan, tenemos que asegurarnos de que el tamaño de la tabla sea un número primo. Podemos demostrar este caso en el Teorema 20.4. Para ser exhaustivos, la Figura 20.8 muestra una rutina que genera números primos, utilizando el algoritmo mostrado en la Figura 9.8 (no merece la pena utilizar un algoritmo más complejo).

Si el tamaño de la tabla es primo y el factor de carga no es mayor que 0,5, todos los sondeos se realizarán en posiciones diferentes y siempre se podrá insertar un elemento.

Teorema 20.4 Si se utiliza el sondeo cuadrático y el tamaño de la tabla es un número primo, entonces siempre se podrá insertar un nuevo elemento si la tabla está como mínimo medio vacía. Además, en el curso de la inserción ninguna celda será sondeada dos veces.

Demostración

Sea M el tamaño de la tabla. Suponga que M es un primo impar mayor que 3. Vamos a demostrar que las primeras $\lceil M/2 \rceil$ posiciones alternativas (incluyendo la original) son distintas. Dos de estas posiciones son $H + i^2 \pmod M$ y $H + j^2 \pmod M$, donde $0 \leq i, j \leq \lfloor M/2 \rfloor$. Vamos a suponer que el teorema no es cierto y que esas dos ubicaciones son la misma y que $i \neq j$. Entonces

$$H + i^2 \equiv H + j^2 \pmod M$$
$$i^2 \equiv j^2 \pmod M$$
$$i^2 - j^2 \equiv 0 \pmod M$$
$$(i - j)(i + j) \equiv 0 \pmod M$$

Continúa

Demostración (cont.)

Puesto que M es primo, querrá decir que $i - j$ o $i + j$ es divisible por M. Puesto que i y j son distintos y su suma es menor que M, ninguna de esas dos posibilidades puede producirse. Con esto llegamos a una contradicción. De aquí se deduce que las primeras $\lceil M/2 \rceil$ alternativas (incluyendo la ubicación original) son todas ellas distintas, lo que garantiza que una inserción tenga éxito si la tabla está al menos medio vacía.

Si la tabla está más que medio llena, aunque solo sea por 1 posición, la inserción podría fallar (aunque el fallo es extremadamente improbable). Si hacemos que el tamaño de la tabla sea un número primo y mantenemos el factor de carga por debajo de 0,5, tenemos garantizado el éxito de la inserción. Si el tamaño de la tabla no es primo, el número de ubicaciones alternativas puede verse enormemente reducido. Por ejemplo, si el tamaño de la tabla fuera 16, las únicas ubicaciones alternativas estarían situadas a las distancias 1, 4, o 9 del punto de sondeo original. De nuevo, el tamaño no es realmente un problema: aunque no dispondríamos de $\lfloor M/2 \rfloor$ alternativas garantizadas, usualmente tendríamos más de las necesarias. Sin embargo, es mejor jugar sobre seguro y utilizar la teoría como guía durante el proceso de selección de parámetros. Además, se ha demostrado empíricamente que los números primos tienden a ser muy adecuados para las tablas hash, porque tienden a eliminar parte de la no aleatoriedad que en ocasiones introducen las funciones hash.

El sondeo cuadrático se puede implementar sin multiplicaciones ni operaciones módulo. Puesto que no se ve afectado por el fenómeno del agrupamiento primario, resulta mejor que el sondeo lineal en la práctica.

La segunda consideración importante tiene relación con la eficiencia. Recuerde que, para un factor de carga de 0,5, la eliminación del fenómeno del agrupamiento primario solo permite ahorrar 0,5 sondeos para una inserción típica y un 0,1 sondeos para una búsqueda con éxito típica. Es verdad que obtenemos algunas ventajas adicionales: encontramos con una secuencia de sondeo larga es significativamente menos probable. Sin embargo, si realizar un sondeo mediante la técnica del sondeo cuadrático requiere el doble tiempo, estará claro que no merece la pena el esfuerzo. El sondeo lineal se implementa mediante una suma simple (sumando 1), mediante una

```
1    /**
2     * Método para encontrar un número primo mayor o igual que n.
3     * @param n el número inicial (tiene que ser positivo).
4     * @return un número primo mayor o igual que n.
5     */
6    private static int nextPrime( int n )
7    {
8        if( n % 2 == 0 )
9            n++;
10
11       for( ; !isPrime( n ); n += 2 )
12           ;
13
14       return n;
15   }
```

Figura 20.8 Una rutina utilizada en el sondeo cuadrático para encontrar un número primo mayor o igual que *N*.

comprobación para determinar si hace falta volver al principio de la tabla por haber alcanzado el final y, muy raramente, mediante una resta (si tenemos que volver al principio de la tabla). La fórmula del sondeo cuadrático sugiere que tenemos que sumar 1 (para pasar de $i-1$ a i), realizar una multiplicación (para calcular i^2), hacer otra suma y luego una operación módulo. Ciertamente, este cálculo parece ser demasiado costoso como para resultar práctico. Sin embargo, podemos emplear el siguiente truco, como se explica en el Teorema 20.5.

Teorema 20.5 El sondeo cuadrático se puede implementar sin costosas multiplicaciones y divisiones.

Demostración Sea H_{i-1} el sondeo calculado más recientemente (H_0 es la posición hash original) y sea H_i el sondeo que estamos intentando calcular. Entonces tenemos

$$H_i = H_0 + i^2 \pmod{M}$$
$$H_{i-1} = H_0 + (i-1)^2 \pmod{M} \qquad \textbf{(20.3)}$$

Si restamos estas dos ecuaciones, obtenemos

$$H_i = H_{i-1} + 2i - 1 \pmod{M} \qquad \textbf{(20.4)}$$

La Ecuación 20.4 nos dice que podemos calcular el nuevo valor H_i a partir del valor anterior H_{i-1} sin necesidad de elevar i al cuadrado. Aunque seguimos teniendo una multiplicación, esa multiplicación es por 2, que se puede implementar de manera trivial mediante un desplazamiento de bits en la mayoría de las computadoras. ¿Y qué pasa con la operación módulo? Esa operación, asimismo, no es realmente necesaria, porque la expresión $2i - 1$ tiene que ser menor que M. Por tanto, si la sumamos a H_{i-1}, el resultado seguirá siendo o bien menor que M (en cuyo caso no necesitamos calcular el módulo) o solo un poco mayor que M (en cuyo caso podemos calcular el módulo simplemente restando M).

El Teorema 20.5 muestra que podemos calcular la siguiente posición que hay que sondear utilizando una suma (para incrementar i), un desplazamiento de bits (para evaluar $2i$), una resta de 1 unidad (para calcular $2i - 1$), otra suma (para incrementar la posición anterior en $2i - 1$), una comprobación para determinar si hay que volver al principio de la tabla y, muy raramente, una resta para implementar la operación módulo. La diferencia es por tanto, un desplazamiento de bits, una resta de 1 unidad y una suma por cada sondeo. El coste de esta operación es probablemente menor que el coste de realizar un sondeo adicional, en el caso de que estén involucradas claves complejas (como cadenas de caracteres).

El detalle final que tenemos que analizar es el de la expansión dinámica. Si el factor de carga es mayor que 0,5, lo que querríamos es duplicar el tamaño de la tabla hash. Pero esta solución plantea unas cuantas cuestiones. En primer lugar, ¿qué dificultad tendrá el encontrar otro número primo? La respuesta es que los números primos son fáciles de encontrar. Esperamos tener que probar solamente $O(\log N)$ números hasta encontrar un número que sea primo. En consecuencia, la rutina mostrada en la Figura 20.8 es muy rápida. La comprobación de primalidad tarda como mucho un tiempo $O(N^{1/2})$,

Expanda la tabla en cuanto el factor de carga alcance el valor 0,5; a dicho proceso se le denomina *rehashing*. Efectúe siempre la duplicación seleccionando un número primo. Los números primos son fáciles de encontrar.

por lo que la búsqueda de un número primo tarda como máximo un tiempo $O(N^{1/2} \log N)$.[2] Este coste es mucho menor que el coste $O(N)$ de transferir el contenido de la tabla antigua a la nueva.

Una vez que hemos asignado una matriz de mayor tamaño, ¿nos limitamos a copiar todos los elementos? La respuesta es por supuesto que no. La nueva matriz implica una nueva función hash, por lo que no podemos emplear las posiciones de la matriz antigua. Por tanto, tenemos que examinar cada elemento de la tabla anterior, calcular su nuevo valor hash e insertarlo en la nueva tabla hash. Este proceso se denomina *rehashing* y se implementa fácilmente en Java.

Al expandir una tabla hash reinserte los elementos en la nueva tabla utilizando la nueva función hash.

20.4.1 Implementación Java

El usuario debe proporcionar un método `hashCode` adecuado para los objetos.

Ahora estamos listos para proporcionar una implementación Java completa de una tabla hash con sondeo cuadrático. Lo haremos implementando la mayor parte de las estructuras `HashSet` y `HashMap` de la API de Colecciones. Recuerde que tanto `HashSet` como `HashMap` requieren un método `hashCode`. El método `hashCode` no tiene parámetro `tableSize`; los algoritmos para tablas hash realizan una operación módulo al final internamente, después de utilizar la función hash suministrada por el usuario. El esqueleto de clase para `HashSet` se muestra en la Figura 20.9. Para que los algoritmos funcionen correctamente, `equals` y `hashCode` deben ser coherentes. Es decir, si dos objetos son iguales, sus valores hash también deben ser iguales.

La tabla hash está compuesta por una matriz de referencias `HashEntry`. Cada referencia `HashEntry` puede ser o `null` o un objeto que almacene un elemento y un miembro de datos que nos diga si la entrada está activa o borrada. Puesto que las matrices de tipo genérico son ilegales, `HashEntry` no es genérica. La clase anidada `HashEntry` se muestra en la Figura 20.10. La matriz se declara en la línea 49. Necesitamos llevar la cuenta tanto del tamaño lógico del `HashSet` como del número de elementos de la tabla hash (incluyendo los elementos como borrados); estos valores se almacenan en `currentSize` y `occupied`, respectivamente, que se declaran en las líneas 46 y 47.

```
1  package weiss.util;
2
3  public class HashSet<AnyType> extends AbstractCollection<AnyType>
4                                 implements Set<AnyType>
5  {
6     private class HashSetIterator implements Iterator<AnyType>
7        { /* Figura 20.18 */ }
8     private static class HashEntry implements java.io.Serializable
9        { /* Figura 20.10 */ }
10
```

Continúa

Figura 20.9 El esqueleto de clase para una tabla hash con sondeo cuadrático.

[2] Esta rutina también es necesaria si añadimos un constructor que permita al usuario especificar un tamaño inicial aproximado para la tabla hash. La implementación de la tabla hash debe garantizar que se utilice un número primo.

```
11    public HashSet( )
12       { /* Figura 20.11 */ }
13    public HashSet( Collection<? extends AnyType> other )
14       { /* Figura 20.11 */ }
15
16    public int size( )
17       { return currentSize; }
18    public Iterator<AnyType> iterator( )
19       { return new HashSetIterator( ); }
20
21    public boolean contains( Object x )
22       { /* Figura 20.12 */ }
23    private static boolean isActive( HashEntry [ ] arr, int pos )
24       { /* Figura 20.13 */ }
25    public AnyType getMatch( AnyType x )
26       { /* Figura 20.12 */ }
27
28    public boolean remove( Object x )
29       { /* Figura 20.14 */ }
30    public void clear( )
31       { /* Figura 20.14 */ }
32    public boolean add( AnyType x )
33       { /* Figura 20.15 */ }
34    private void rehash( )
35       { /* Figura 20.16 */ }
36    private int findPos( Object x )
37       { /* Figura 20.17 */ }
38
39    private void allocateArray( int arraySize )
40       { array = new HashEntry[ arraySize ]; }
41    private static int nextPrime( int n )
42       { /* Figura 20.8 */ }
43    private static boolean isPrime( int n )
44       { /* Véase el código en línea */ }
45
46    private int currentSize = 0;
47    private int occupied = 0;
48    private int modCount = 0;
49    private HashEntry [ ] array;
50 }
```

Figura 20.9 (*Continuación*).

```
1    private static class HashEntry implements java.io.Serializable
2    {
3       public Object element;    // el elemento
4       public boolean isActive;  // false si está marcado como borrado
5
6       public HashEntry( Object e )
7       {
8          this( e, true );
9       }
10
11      public HashEntry( Object e, boolean i )
12      {
13         element = e;
14         isActive = i;
15      }
16   }
```

Figura 20.10 La clase anidada `HashEntry`.

La disposición general es similar a la de `TreeSet`.

El resto de la clase contiene declaraciones para las rutinas y el iterador de la tabla hash. La disposición general es similar a la de `TreeSet`.

Se declaran tres métodos privados; los describiremos cuando se utilicen en la implementación de la clase. Ahora podemos analizar la implementación de la clase `HashSet`.

La mayoría de las rutinas tienen solo unas cuantas líneas de código porque llaman a `findPos` para realizar el sondeo cuadrático.

Los constructores de la tabla hash se muestran en la Figura 20.11; no tienen nada de especial. La rutina de búsqueda, `contains`, y el método no estándar `getMatch` se muestran en la Figura 20.12. `contains` utiliza el método privado `isActive`, mostrado en la Figura 20.13. Tanto `contains` como `getMatch` llaman también a `findPos`, que se muestra posteriormente, para implementar el sondeo cuadrático. El método `findPos` es el único lugar de todo el código que depende del mecanismo de sondeo cuadrático. Por tanto, `contains` y `getMatch` son fáciles de implementar: un elemento se habrá encontrado si el resultado de `findPos` es una celda activa (si `findPos` se detiene en una celda activa, debe existir una correspondencia). De forma similar, la rutina `remove` mostrada en la Figura 20.14 es corta. Comprobamos si `findPos` nos lleva hasta una celda activa; en caso afirmativo, la celda se marca como borrada. En caso contrario, se devuelve `false` inmediatamente. Observe que esto hace que se reduzca `currentSize`, pero no `occupied`. Asimismo, si hay muchos elementos borrados, se modifica el tamaño de la tabla hash, en las líneas 16 y 17. El mantenimiento de `modCount` es idéntico al de otros componentes de la API de Colecciones que hemos implementado previamente. `clear` elimina todos los elementos del `HashSet`.

La rutina `add` se muestra en la Figura 20.15. En la línea 8 llamamos a `findPos`. Si se consigue encontrar `x`, devolvemos `false` en la línea 10 porque no están permitidos los elementos duplicados. En caso contrario, `findPos` proporciona el lugar en el que insertar `x`. La inserción se realiza en la línea 12. Ajustamos `currentSize`, `occupied` y `modCount` en las líneas 13 a 15 y volvemos, a menos que haga falta un rehash; si es así, invocamos el método privado `rehash`.

```
1     private static final int DEFAULT_TABLE_SIZE = 101;
2
3     /**
4      * Construir un HashSet vacio.
5      */
6     public HashSet( )
7     {
8        allocateArray( DEFAULT_TABLE_SIZE );
9        clear( );
10    }
11
12    /**
13     * Construir un HashSet a partir de cualquier colección.
14     */
15    public HashSet( Collection<? extends AnyType> other )
16    {
17       allocateArray( nextPrime( other.size( ) * 2 ) );
18       clear( );
19
20       for( AnyType val : other )
21          add( val );
22    }
```

Figura 20.11 Inicialización de la tabla hash.

La rutina `add` realiza rehashing si la tabla está (medio) llena.

El código que implementa el *rehashing* se muestra en la Figura 20.16. La línea 7 guarda una referencia a la tabla original. En las líneas 10 a 12 creamos una nueva tabla hash vacía, que tendrá un factor de carga de 0,25 cuando `rehash` termine. Después recorremos la matriz original y añadimos con `add` los elementos activos a la nueva tabla. La rutina `add` utiliza la nueva función hash (ya que está basada lógicamente en el tamaño de `array`, que ha variado) y resuelve automáticamente todas las colisiones. Podemos estar seguros de que la llamada recursiva a `add` (en la línea 17) no provoca otro rehash. Alternativamente, podríamos sustituir la línea 17 por dos líneas de código encerradas entre llaves (véase el Ejercicio 20.14).

Hasta ahora, nada de lo que hemos hecho depende del sondeo cuadrático. La Figura 20.17 implementa `findPos`, que se encarga finalmente de tratar con el algoritmo de sondeo cuadrático. Continuamos explorando la tabla hasta encontrar una celda vacía o una correspondencia. Las líneas 22 a 25 implementan directamente la metodología descrita en el Teorema 20.5, utilizando dos sumas. Existen complicaciones adicionales, porque `null` es un elemento válido en el `HashSet`; el código ilustra por qué es preferible asumir que `null` no es válido.

La Figura 20.18 proporciona la implementación de la clase interna iteradora. Es código relativamente estándar, aunque contiene algunas complicaciones. `visited` representa el número de llamadas a `next`, mientras que `currentPos` representa el índice del último objeto devuelto por `next`.

```
1   /**
2    * Este método no es estándar Java.
3    * Al igual que contains, comprueba si x se encuentra en el conjunto.
4    * Si está, devuelve la referencia al objeto correspondiente;
5    * en caso contrario, devuelve null.
6   * @param x el objeto que hay que buscar.
7    * @return si contains(x) es false, el valor de retorno es null;
8    * en caso contrario, el valor de retorno es el objeto que hace que
9    * contains(x) devuelva true.
10   */
11  public AnyType getMatch( AnyType x )
12  {
13     int currentPos = findPos( x );
14
15     if( isActive( array, currentPos ) )
16        return (AnyType) array[ currentPos ].element;
17     return null;
18  }
19
20     /**
21   * Comprueba si algún elemento se encuentra en esta colección.
22   * @param x cualquier objeto.
23   * @return true si esta colección contiene un elemento igual a x.
24   */
25  public boolean contains( Object x )
26  {
27     return isActive( array, findPos( x ) );
28  }
```

Figura 20.12 Las rutinas de búsqueda para una tabla hash con sondeo cuadrático.

```
1   /**
2    * Comprueba si el elemento de pos está activo.
3    * @param pos una posición de la tabla hash.
4    * @param arr matriz de obj. HashEntry (puede ser oldArray durante el rehash).
5    * @return true si esta posición está activa.
6    */
7   private static boolean isActive( HashEntry [ ] arr, int pos )
8   {
9      return arr[ pos ] != null && arr[ pos ].isActive;
10  }
```

Figura 20.13 El método `isActive` para una tabla hash con sondeo cuadrático.

```
1    /**
2     * Elimina un elemento de esta colección.
3     * @param x cualquier objeto.
4     * @return true si este elemento ha sido eliminado de la colección.
5     */
6    public boolean remove( Object x )
7    {
8       int currentPos = findPos( x );
9       if( !isActive( array, currentPos ) )
10         return false;
11
12      array[ currentPos ].isActive = false;
13      currentSize--;
14      modCount++;
15
16      if( currentSize < array.length / 8 )
17         rehash( );
18
19      return true;
20   }
21
22   /**
23    * Cambia el tamaño de esta colección a cero.
24    */
25   public void clear( )
26   {
27      currentSize = occupied = 0;
28      modCount++;
29      for( int i = 0; i < array.length; i++ )
30         array[ i ] = null;
31   }
```

Figura 20.14 Las rutinas `remove` y `clear` para una tabla hash con sondeo cuadrático.

Finalmente, la Figura 20.19 implementa `HashMap`. Se parece bastante a `TreeMap`, salvo porque `Pair` es una clase anidada en lugar de una clase interna (no necesita acceder a un objeto externo), y porque implementa tanto `equals` como `hashCode` en lugar de la interfaz `Comparable`.

20.4.2 Análisis del sondeo cuadrático

El sondeo cuadrático no ha sido todavía analizado matemáticamente, aunque sabemos que elimina el fenómeno del agrupamiento primario. En el sondeo cuadrático, los elementos cuyo valor hash corresponde a una misma posición provocan un sondeo de las mismas celdas alternativas, fenómeno

```
1   /**
2    * Añade un elemento a esta colección.
3    * @param x cualquier objeto.
4    * @return true si este elemento ha sido añadido a la colección.
5    */
6   public boolean add( AnyType x )
7   {
8      int currentPos = findPos( x );
9      if( isActive( array, currentPos ) )
10        return false;
11
12     if( array[ currentPos ] == null )
13        occupied++;
14     array[ currentPos ] = new HashEntry( x, true );
15     currentSize++;
16     modCount++;
17
18     if( occupied > array.length / 2 )
19        rehash( );
20
21     return true;
22  }
```

Figura 20.15 La rutina `add` para una tabla hash con sondeo cuadrático.

```
1   /**
2    * Rutina privada para realizar el rehashing.
3    * Puede ser invocada tanto por add como por remove.
4    */
5   private void rehash( )
6   {
7      HashEntry [ ] oldArray = array;
8
9        // Crear una nueva tabla vacía
10     allocateArray( nextPrime( 4 * size( ) ) );
11     currentSize = 0;
12     occupied = 0;
13
14       // Copiar la tabla
15     for( int i = 0; i < oldArray.length; i++ )
16        if( isActive( oldArray, i ) )
17           add( (AnyType) oldArray[ i ].element );
18  }
```

Figura 20.16 El método `rehash` para una tabla hash con sondeo cuadrático.

```
1    /**
2     * Método que realiza la resolución del sondeo cuadrático.
3     * @param x el elemento que hay que buscar.
4     * @return la posición donde termina la búsqueda.
5     */
6    private int findPos( Object x )
7    {
8       int offset = 1;
9       int currentPos = ( x == null ) ?
10                          0 : Math.abs( x.hashCode( ) % array.length );
11
12      while( array[ currentPos ] != null )
13      {
14         if( x == null )
15         {
16            if( array[ currentPos ].element == null )
17               break;
18         }
19         else if( x.equals( array[ currentPos ].element ) )
20            break;
21
22         currentPos += offset;            // Calcular i-ésimo sondeo
23         offset += 2;
24         if( currentPos >= array.length ) // Implementar el módulo
25            currentPos -= array.length;
26      }
27
28      return currentPos;
29   }
```

Figura 20.17 La rutina que se encarga finalmente de tratar con el sondeo cuadrático.

que se conoce con el nombre de *agrupamiento secundario*. De nuevo, no podemos asumir que los sondeos sucesivos sean independientes entre sí. El agrupamiento secundario es difícil de analizar desde el punto de vista teórico. Los resultados de simulación sugieren que, con carácter general, este fenómeno provoca menos de medio sondeo adicional por cada búsqueda, y que este incremento solo se produce para factores de carga altos. La Figura 20.6 ilustra la diferencia entre el sondeo lineal y el sondeo cuadrático y en ella podemos ver que el sondeo cuadrático no se ve tan afectado por el fenómeno del agrupamiento como el sondeo lineal.

El sondeo cuadrático se implementa en `findPos`. Utiliza el truco que hemos mencionado antes para evitar las multiplicaciones y las operaciones módulo.

```
1    /**
2     * Esta es la implementación del HashSetIterator.
3     * Mantiene una noción de una posición actual y, por supuesto,
4     * la referencia implícita al HashSet.
5     */
6    private class HashSetIterator implements Iterator<AnyType>
7    {
8       private int expectedModCount = modCount;
9       private int currentPos = -1;
10      private int visited = 0;
11
12      public boolean hasNext( )
13      {
14         if( expectedModCount != modCount )
15            throw new ConcurrentModificationException( );
16
17         return visited != size( );
18      }
19
20      public AnyType next( )
21      {
22         if( !hasNext( ) )
23            throw new NoSuchElementException( );
24
25         do
26         {
27            currentPos++;
28         } while( currentPos < array.length &&
29                                 !isActive( array, currentPos ) );
30
31         visited++;
32         return (AnyType) array[ currentPos ].element;
33      }
34
35      public void remove( )
36      {
37         if( expectedModCount != modCount )
38            throw new ConcurrentModificationException( );
39         if( currentPos == -1 || !isActive( array, currentPos ) )
40            throw new IllegalStateException( );
41
42         array[ currentPos ].isActive = false;
43         currentSize--;
44         visited--;
45         modCount++;
46         expectedModCount++;
47      }
48   }
```

Figura 20.18 La clase interna `HashSetIterator`.

```
1  package weiss.util;
2
3  public class HashMap<KeyType,ValueType> extends MapImpl<KeyType,ValueType>
4  {
5     public HashMap( )
6        { super( new HashSet<Map.Entry<KeyType,ValueType>>( ) ); }
7
8     public HashMap( Map<KeyType,ValueType> other )
9        { super( other ); }
10
11    protected Map.Entry<KeyType,ValueType> makePair( KeyType key, ValueType value)
12       { return new Pair<KeyType,ValueType>( key, value ); }
13
14    protected Set<KeyType> makeEmptyKeySet( )
15       { return new HashSet<KeyType>( ); }
16
17    protected Set<Map.Entry<KeyType,ValueType>>
18    clonePairSet( Set<Map.Entry<KeyType,ValueType>> pairSet )
19    {
20       return new HashSet<Map.Entry<KeyType,ValueType>>( pairSet );
21    }
22
23    private static final class Pair<KeyType,ValueType>
24    extends MapImpl.Pair<KeyType,ValueType>
25    {
26       public Pair( KeyType k, ValueType v )
27          { super( k, v ); }
28
29       public int hashCode( )
30       {
31          KeyType k = getKey( );
32          return k == null ? 0 : k.hashCode( );
33       }
34
35    public boolean equals( Object other )
36    {
37       if( other instanceof Map.Entry )
38       {
39          KeyType thisKey = getKey( );
40          KeyType otherKey = ((Map.Entry<KeyType,ValueType>) other).getKey( );
41
```

Continúa

Figura 20.19 La clase `HashMap`.

```
42          if( thisKey == null )
43              return thisKey == otherKey;
44          return thisKey.equals( otherKey );
45      }
46      else
47          return false;
48      }
49    }
50 }
```

Figura 20.19 (*Continuación*).

En el fenómeno del *agrupamiento secundario*, los elementos cuyo valor hash se corresponde con una misma posición hacen que se sondeen las mismas celdas alternativas. El agrupamiento secundario constituye todavía una incógnita menor, desde el punto de vista del análisis teórico.

El *doble hash* es una técnica de hashing que no sufre el fenómeno del agrupamiento secundario. Utiliza una segunda función hash para dirigir la resolución de colisiones.

Hay disponibles técnicas que eliminan el fenómeno del agrupamiento secundario. La más popular es el *doble hash*, en el que se utiliza una segunda función hash para dirigir la resolución de colisiones. Específicamente, sondeamos a una distancia $Hash_2(X)$, $2Hash_2(X)$, etc. La segunda función hash debe ser elegida con cuidado (por ejemplo, *nunca* puede ser 0 el resultado de su evaluación) y todas las celdas deben poder ser sondeadas. Una función como $Hash_2(X) = R - (X \bmod R)$, donde R es un número primo menor que M, suele funcionar bien. El doble hash es interesante desde el punto de vista teórico, porque se puede demostrar que utiliza esencialmente el mismo número de sondeos que el análisis puramente aleatorio del sondeo lineal parece sugerir. Sin embargo, es algo más complicado de implementar que el sondeo cuadrático y requiere que se preste una atención cuidadosa a algunos detalles.

No parece que exista ninguna buena razón para no utilizar la estrategia del sondeo cuadrático, a menos que el gasto adicional derivado de la necesidad de mantener una tabla medio vacía sea inaceptable. Eso podría suceder en otros lenguajes de programación, por ejemplo, si los elementos que se estuvieran almacenando fueran de muy gran tamaño.

20.5 Hash con encadenamiento separado

El *hash con encadenamiento separado* es una alternativa muy eficiente en términos de espacio al sondeo cuadrático; en esta técnica se mantiene una matriz de listas enlazadas. Es menos sensible a los factores de carga altos.

Una alternativa bastante popular y muy eficiente en términos de espacio al sondeo cuadrático es el *hash con encadenamiento separado*, en el que se mantiene una matriz de listas enlazadas. Para una matriz de listas enlazadas, L_0, L_1, ..., $L_M - 1$, la función hash nos dice en qué lista hay que insertar un elemento X y luego, durante una operación `find`, qué lista contiene a X. La idea es que, aunque la búsqueda en una lista enlazada es una operación lineal, si las listas son suficientemente cortas, el tiempo de búsqueda será muy rápido. En particular, suponga que el factor de carga, N/M, es λ y que no está acotado por 1,0. En ese caso, la lista típica tendrá una longitud λ, haciendo que el número esperado de sondeos para una inserción o una búsqueda que no tenga éxito sea λ, y que el número esperado de sondeos para una búsqueda que tenga éxito sea $1 + \lambda/2$. La razón

es que una búsqueda con éxito deberá producirse en una lista no vacía y que, en dicha lista, cabe esperar que tengamos que recorrer la mitad de la misma. El coste relativo de una búsqueda con éxito frente a una búsqueda que no lo tenga es inusual, en el sentido de que, si $\lambda < 2$, la búsqueda con éxito es más costosa que la que no lo tiene. Sin embargo, esta condición tiene bastante sentido, porque muchas búsquedas sin éxito van a encontrarse con una lista enlazada vacía.

Un factor de carga típico es 1,0; los factores de carga menores no mejoran el rendimiento de manera significativa, y además requieren malgastar espacio adicional. El atractivo del mecanismo de encadenamiento separado es que el rendimiento no se ve afectado por un incremento moderado del factor de carga; por tanto, puede evitarse el rehashing. Para aquellos lenguajes que no permiten la expansión dinámica de matrices, este aspecto resulta importante. Además, el número esperado de sondeos en una búsqueda es inferior que en el sondeo cuadrático, particularmente para las búsquedas que no tenga éxito.

Para las tablas hash con encadenamiento separado, un factor de carga razonable es 1,0. Un factor de carga más pequeño no mejora significativamente el rendimiento; un factor de carga moderadamente más grande resulta aceptable y permite ahorrar espacio.

Podemos implementar el mecanismo de encadenamiento separado utilizando nuestras clases existentes de listas enlazadas. Sin embargo, como el nodo de cabecera representa un gasto de espacio y no es realmente necesario, si el espacio fuera un problema podríamos elegir no reutilizar componentes, implementado en su lugar una lista simple de tipo pila. El esfuerzo de codificación es bastante pequeño. Además, el coste en términos de espacio es esencialmente de una referencia por nodo, más una referencia adicional por cada lista; por ejemplo, cuando el factor de carga es 1,0, tendremos dos referencias por cada elemento. Esta característica podría tener su importancia en otros lenguajes de programación si el tamaño de un elemento fuera grande. En ese caso, tendríamos los mismos compromisos que en el caso de la implementación de las pilas con matrices o con listas enlazadas. La API de Colecciones de Java utiliza el mecanismo de encadenamiento separado para tablas hash con un factor de carga predeterminado de 0,75.

Para ilustrar la complejidad (o más bien, la relativa falta de complejidad) de la tabla hash con encadenamiento separado, la Figura 20.20 proporciona un pequeño esbozo de la implementación básica de este mecanismo. En esa implementación se evitan temas como el rehashing, no se implementa el método `remove` y ni siquiera se lleva la cuenta del tamaño actual. De todos modos, muestra cuál es la lógica básica de los métodos `add` y `contains`, ambos de los cuales utilizan el código hash para seleccionar la apropiada lista simplemente enlazada.

20.6 Comparación entre las tablas hash y los árboles de búsqueda binaria

Utilice la tabla hash en lugar de un árbol de búsqueda binaria si no necesita estadísticas de orden y le preocupa la posible existencia de entradas no aleatorias.

También podemos utilizar árboles de búsqueda binaria para implementar las operaciones `insert` y `find`. Aunque las cotas correspondientes de tiempo promedio son $O(\log N)$, los árboles de búsqueda binaria también soportan rutinas que exigen un orden y que son por tanto más potentes. Utilizando una tabla hash no podemos encontrar de manera eficiente el elemento mínimo o ampliar la tabla para permitir el cálculo de una estadística de orden. No podemos buscar de manera eficiente una cadena a menos que conozcamos la cadena de caracteres exacta. Un árbol de búsqueda binaria podría encontrar

```
1  class MyHashSet<AnyType>
2  {
3     public MyHashSet( )
4        { this( 101 ); }
5
6     public MyHashSet( int numLists )
7        { lists = new Node[ numLists ]; }
8
9     public boolean contains( AnyType x )
10    {
11       for( Node<AnyType> p = lists[ myHashCode( x ) ]; p != null; p = p.next)
12          if( p.data.equals( x ) )
13             return true;
14
15       return false;
16    }
17
18    public boolean add( AnyType x )
19    {
20       int whichList = myHashCode( x );
21
22       for( Node<AnyType> p = lists[ whichList ]; p != null; p = p.next )
23          if( p.data.equals( x ) )
24             return false;
25
26       lists[ whichList ] = new Node<AnyType>( x, lists[ whichList ] );
27       return true;
28    }
29
30    private int myHashCode( AnyType x )
31       { return Math.abs( x.hashCode( ) % lists.length ); }
32
33    private Node<AnyType> [ ] lists;
34
35    private static class Node<AnyType>
36    {
37       Node( AnyType d, Node<AnyType> n )
38       {
39          data = d;
40          next = n;
41       }
42
43       AnyType data;
44       Node<AnyType> next;
45    }
46 }
```

Figura 20.20 Implementación simplificada de una tabla hash con encadenamiento separado.

rápidamente todos los elementos en un cierto rango, mientras que esta capacidad no está soportada por una tabla hash. Además, la cota $O(\log N)$ no es necesariamente mucho peor que $O(1)$, especialmente porque en los árboles de búsqueda no hacen falta multiplicaciones o divisiones.

El caso peor para las tablas hash suele ser el resultado de un error de implementación, mientras que las entradas ordenadas pueden hacer que los árboles de búsqueda binaria tengan un rendimiento bastante inadecuado. Los árboles de búsqueda equilibrados son bastante costosos de implementar. Por tanto, si no se requiere información de orden y existe la más mínima sospecha de que la entrada pudiera estar ordenada, es preferible emplear como estructura de datos una tabla hash.

20.7 Aplicaciones de las tablas hash

Las aplicaciones de las tablas hash son muy numerosas.

Las aplicaciones de las tablas hash son muy numerosas. Los compiladores utilizan tablas hash para llevar la cuenta de las variables declaradas en el código fuente. La estructura de datos correspondiente se denomina *tabla de símbolos*. Las tablas hash son la aplicación ideal de este problema específico, porque solo se realizan operaciones `insert` y `find`. Los identificadores son normalmente cortos, por lo que la función hash se puede calcular rápidamente. En esta aplicación, la mayoría de las búsquedas tienen éxito.

Otro uso común de las tablas hash es en los programas de juegos. A medida que el programa explora a través de las diferentes líneas de juego, lleva la cuenta de las posiciones con las que ya se ha encontrado, calculando una función hash basada en la posición (y almacenando su movimiento a partir de esa posición). Si vuelve a aparecer la misma posición, usualmente debido a una simple transposición de movimientos, el programa puede ahorrarse una costosa serie de recálculos. Esta característica general de todos los programas de juegos se denomina *tabla de transposición*. Hemos hablado de esta característica en la Sección 10.2, a la hora de implementar el algoritmo del juego de las tres en raya.

Un tercer uso de las tablas hash es en los comprobadores ortográficos en línea. Si es importante la detección de errores ortográficos (en lugar de la corrección de esos errores), se puede calcular el valor hash de cada una de las palabras de un diccionario completo y comprobar las palabras en un tiempo constante. Las tablas hash están muy adaptadas a esta aplicación, porque las palabras no tienen por qué estar alfabetizadas. El imprimir los errores ortográficos en el orden en el que aparecen en el documento es perfectamente aceptable.

Resumen

Las tablas hash se pueden utilizar para implementar las operaciones `insert` y `find` en un tiempo promedio constante. Es especialmente importante prestar atención a detalles tales como el factor de carga a la hora de utilizar las tablas hash; en caso contrario, las cotas de tiempo constante dejan de tener sentido. También es importante seleccionar la función hash con cuidado cuando la clave no sea un valor entero o una cadena de caracteres corta. Hay que elegir una función fácilmente calculable y que distribuya equitativamente los valores.

Para la técnica de encadenamiento separado, el factor de carga es normalmente próximo a 1, aunque el rendimiento no se degrada significativamente a menos que el factor de carga sea muy

grande. En el caso del sondeo cuadrático, el tamaño de la tabla debe ser un número primo y el factor de carga no debe sobrepasar el valor 0,5. Debe utilizarse el *rehashing* para el sondeo cuadrático con el fin de permitir que la tabla crezca y mantener así el factor de carga correcto. Esta técnica cobra su importancia en caso de que el espacio escasee y no sea posible declarar simplemente una tabla hash de tamaño enorme.

Con esto completamos nuestro análisis de los algoritmos básicos de búsqueda. En el Capítulo 21 examinaremos el montículo binario, que implementa la cola con prioridad y soporta, por tanto, un acceso eficiente al elemento mínimo de una colección de elementos.

Conceptos clave

agrupamiento primario Durante el sondeo lineal se forman grandes agrupamientos de celdas ocupadas, haciendo que las inserciones en esos agrupamientos sean muy costosas (además de que cada inserción hace que crezca aun más el agrupamiento) y afectando así al rendimiento. (771)

agrupamiento secundario Agrupamiento que se produce cuando los elementos a los que la función hash hace corresponder con una misma posición de la tabla, sondean las mismas celdas alternativas. Constituye todavía un problema menor que no se ha resuelto teóricamente. (788)

borrado perezoso La técnica consistente en marcar los componentes como borrados en lugar de eliminarlos físicamente de una tabla hash. Esta técnica es necesaria a la hora de sondear tablas hash. (718, 770)

colisión El resultado cuando dos o más elementos de una tabla hash se corresponden con una misma posición. Este problema es inevitable, porque hay más elementos que posiciones. (764)

doble hash Una técnica de hash que no sufre el problema del agrupamiento secundario. Se emplea una segunda función hash para dirigir el mecanismo de resolución de colisiones. (788)

encadenamiento separado Una alternativa al sondeo cuadrático que es bastante eficiente en términos de espacio y en la que se mantiene una matriz de listas enlazadas. Es menos sensible a los factores de carga altos y exhibe algunos de los compromisos que ya hemos tomado en consideración al analizar las implementaciones de pilas basadas en matriz y en lista enlazada. (788)

factor de carga El número de elementos de una tabla hash dividido entre el tamaño de la matriz de la tabla. Representa la fracción de la tabla que está llena. En una tabla hash con sondeo lineal, el factor de carga va de 0 (vacía) a 1 (llena). Con la técnica de encadenamiento separado, puede ser mayor que 1. (770)

función hash Una función que convierte el elemento en un entero adecuado para indexar la matriz en la que el elemento se va a almacenar. Si la función hash fuera biunívoca, podríamos acceder al elemento utilizando su índice matricial pero, como la función hash no es biunívoca, varios elementos colisionarán en un mismo índice. (788)

hashing La implementación de tablas hash para realizar inserciones, borrados y búsquedas. (788)

sondeo cuadrático Un método de resolución de colisiones que examina las celdas situadas a una distancia de 1, 4, 9 etc. del punto de sondeo original. (774)

sondeo lineal Una forma de evitar colisiones explorando secuencialmente una matriz hasta encontrar una celda vacía. (768)

tabla hash Una tabla utilizada para implementar un diccionario en un tiempo constante por cada operación. (763)

Errores comunes

1. La función hash devuelve un valor `int`. Puesto que los cálculos intermedios permiten el desbordamiento, la variable lógica debería comprobar que el resultado de la operación módulo no sea negativo, para evitar correr el riesgo de que el valor de retorno esté fuera de límites.
2. El rendimiento de una tabla de sondeo se degrada enormemente a medida que el factor de carga se aproxima a 1,0. No deje que esto suceda. Efectúe un rehash cuando el factor de carga alcance 0,5.
3. El rendimiento de todos los métodos hash depende de la utilización de una buena función de hash. Un error bastante común es el de proporcionar una función inadecuada.

Internet

Tiene a su disposición la tabla hash con sondeo cuadrático.

HashSet.java	Contiene la implementación de la clase `HashSet`.
HashMap.java	Contiene la implementación de la clase `HashMap`.

Ejercicios

EN RESUMEN

20.1 Dada la entrada {437, 123, 617, 419, 456, 674, 199}, un tamaño de tabla fijo de 10 y una función hash $H(X) = X \bmod 10$, muestre las tablas hash resultantes con

a. Sondeo lineal.

b. Sondeo cuadrático.

20.2 Muestre el resultado de aplicar un rehash a las tablas de sondeo del Ejercicio 20.1. Efectúe el rehash con un tamaño de tabla que sea un número primo.

20.3 ¿Cuál es el tamaño de tabla de sondeo apropiado si el número de elementos de la tabla hash es 8?

20.4 Explique cómo se realiza el borrado en las tablas hash tanto de sondeo como con encadenamiento separado.

20.5 ¿Cuáles son los índices de la matriz para una tabla hash de tamaño 5?

20.6 ¿Cuál es el número esperado de sondeos tanto para búsquedas con éxito como sin éxito, en una tabla de sondeo lineal con factor de carga igual a 0,25?

EN TEORÍA

20.7 Bajo ciertas suposiciones, el coste esperado de una inserción en una tabla hash con agrupamiento secundario está dado por $1/(1 - \lambda) - \lambda - \ln(1 - \lambda)$. Lamentablemente, esta fórmula no es precisa para el sondeo cuadrático. Sin embargo, suponiendo que lo fuera,

a. ¿Cuál sería el coste esperado de una búsqueda sin éxito?

b. ¿Cuál sería el coste esperado de una búsqueda con éxito?

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

a. Demuestre que, con esta estrategia, si la tabla no está llena, la colisión siempre se puede resolver.

b. ¿Cabría esperar que esta estrategia eliminara el fenómeno del agrupamiento primario?

c. ¿Cabría esperar que esta estrategia eliminara el fenómeno del agrupamiento secundario?

20.9 Si se implementa el rehashing en cuanto el factor de carga alcanza el valor 0,5, cuando se inserte el último elemento el factor de carga será como mínimo 0,25 y como máximo 0,5. ¿Cuál será el factor de carga esperado? En otras palabras, ¿es cierto o falso que el factor de carga será 0,375 como media?

20.10 Cuando se implementa el paso de rehashing hay que utilizar $O(N)$ sondeos para reinsertar los N elementos. Proporcione una estimación del número de sondeos (es decir, N o $2N$ o alguna otra cosa). *Pista*: calcule el coste medio de cada operación de inserción en la nueva tabla. Estas inserciones varían entre el factor de carga 0 y el factor de carga 0,25.

20.11 Se utiliza una tabla hash con sondeo cuadrático para almacenar 7.000 objetos `String`. Suponga que el factor de carga es 0,4 y que la longitud media de las cadenas de caracteres es 6. Determine

a. El tamaño de la tabla hash.

b. La cantidad de memoria utilizada para almacenar los 7.000 objetos `String`.

c. La cantidad de memoria adicional utilizada por la tabla hash.

EN LA PRÁCTICA

20.12 Proporcione una implementación completa de `HashSet` empleando el mecanismo de encadenamiento separado.

20.13 Implemente el sondeo lineal.

20.14 Para la tabla hash con sondeo, implemente el código de rehashing sin realizar una llamada recursiva a `add`.

20.15 Experimente con una función hash que examine cada uno de los caracteres de una cadena. ¿Es esta una mejor elección que la del texto? Explique su respuesta.

PROYECTOS DE PROGRAMACIÓN

20.16 Busque en Internet un buen diccionario en línea. Seleccione un tamaño de tabla que sea al menos el doble que el del diccionario. Aplique la función hash descrita en el texto y almacene un recuento del número de palabras que han correspondido a cada posición. Obtendrá con ello una distribución: a un cierto porcentaje de las posiciones no le habrá correspondido ninguna palabra, a otras les habrá correspondido solo una, a otras dos, etc. Compare esta distribución con lo que ocurriría en el caso de utilizar números teóricamente aleatorios (que hemos visto en la Sección 9.3).

20.17 Realice simulaciones para comparar el rendimiento observado del mecanismo de hash con los resultados teóricos. Declare una tabla hash con sondeo, inserte en la tabla 10.000 enteros generados aleatoriamente y cuente el número medio de sondeos utilizados. Este número es el coste medio de una búsqueda que tenga éxito. Repita la prueba varias veces para obtener un buen promedio. Ejecútela tanto para sondeo lineal como para sondeo cuadrático, y hágalo para factores de carga finales iguales a 0,1, 0,2, ..., 0,9. Declare siempre la tabla de modo que no haga falta rehashing. Es decir, la prueba para un factor de carga de 0,4 declararía una tabla de tamaño aproximadamente igual a 25.000 (ajustado para que sea un número primo).

Referencias

A pesar de la aparente simplicidad del algoritmo de hash, buena parte del análisis es bastante difícil y hay muchas cuestiones que todavía no han sido resueltas. Asimismo, existen muchas ideas interesantes que intentan, en general, que sea poco probable que se den las posibilidades de caso peor durante el manejo de las tablas hash.

Uno de los primeros artículos sobre hashing es [11]. En [6] se proporciona una gran cantidad de información sobre el tema, incluyendo un análisis de la técnica de hash con sondeo lineal. El doble hash se analiza en [5] y [7]. En [12] se describe otro esquema más de resolución de colisiones, denominado *hash con coalescencia*. [8] proporciona una excelente panorámica de la materia y [9] contiene sugerencias y consejos a la hora de elegir funciones hash. En [4] podrá encontrar resultados de simulación y analíticos precisos para todos los métodos descritos en este capítulo. El hash uniforme, en el que no existe agrupamiento, es óptimo en lo que respecta al coste de una búsqueda con éxito [13].

Si las claves de entrada se conocen de antemano, se pueden definir funciones hash perfectas que no permiten colisiones [1]. En [2] y [3] se proporcionan algunos esquemas hash más complicados, para los que el caso peor no depende de la entrada concreta, sino de los números aleatorios seleccionados por el algoritmo. Estos esquemas garantizan que solo

se produzca un número constante de colisiones en el caso peor (aunque la construcción de una función hash puede requerir un tiempo muy largo en el caso, improbable, de que los números aleatorios obtenidos sean inadecuados). Son útiles para implementar tablas en hardware.

En [10] se describe un método para implementar el Ejercicio 20.8.

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*, 2ª ed., Addison-Wesley, Reading, MA, 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*, 2ª ed., Addison-Wesley, Reading, MA, 1998.
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.