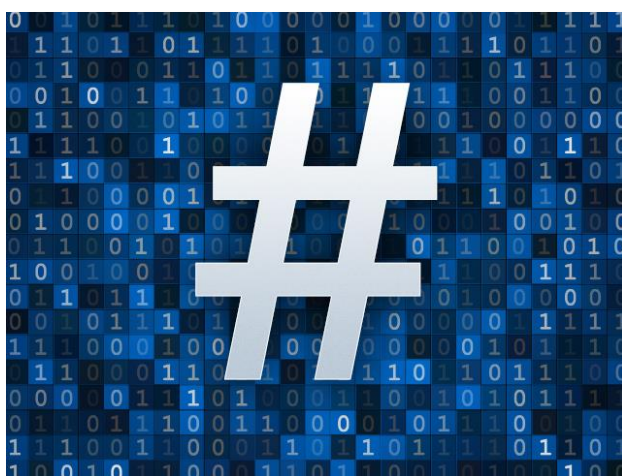




UCAM
UNIVERSIDAD
CATÓLICA DE MURCIA

GRUPO  **TUDIUM**
FORMACIÓN



Tema 4

HASHING

STUDIUM

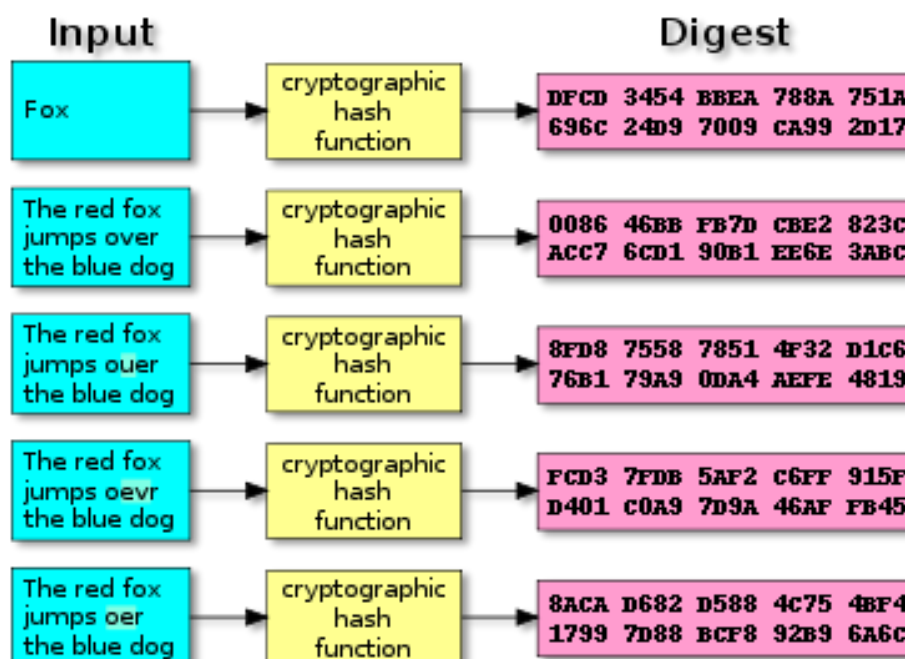
www.grupostudium.com
informacion@grupostudium.com
954 539 952

Introducción

En temas anteriores hemos estudiado tanto los algoritmos más eficientes y populares tanto de **ordenación** como de **búsqueda** de elementos en diferentes estructuras de datos, como **tablas**, **listas**, **árboles** o **grafos**.

En ocasiones se nos presentaban casos en los que tanto la búsqueda como la ordenación se hacían tediosas y poco eficientes. En el presente tema, estudiaremos métodos de **almacenamiento de información** y la posterior **búsqueda** mediante cálculo de dirección basado en clave, también conocido como **Hashing**.

Por ejemplo, si sabemos que se ordenó una lista, podríamos buscar, en tiempo logarítmico usando una **búsqueda binaria**. En este tema intentaremos ir un paso más allá construyendo una estructura de datos en la que se pueda buscar en tiempo **O(1)**. Este concepto se conoce como **búsqueda por transformación de claves** (o hashing en inglés).



Para hacer esto, necesitaremos saber aún más sobre dónde podrían estar los elementos cuando vamos a buscarlos en la colección. Si cada elemento está donde debe estar, entonces la búsqueda puede usar una sola comparación para descubrir la presencia de un elemento. Veremos, sin embargo, que éste no suele ser el caso.

Hasta ahora las técnicas de localización de registros vistas, emplean un proceso de búsqueda que implica cierto tiempo y esfuerzo. El método de transformación de claves nos permite encontrar directamente el registro buscado en tablas o archivos que no se encuentran necesariamente ordenados, en un tiempo **independiente de la cantidad de datos**.

En los apartados siguientes se hará referencia a tablas como las estructuras de almacenamiento de la información, pues en general el método de Hashing se aplica en situaciones que implican el manejo de una considerable cantidad de información organizada.

A diferencia de una búsqueda indexada por claves ordinaria, donde usamos el valor de las claves como índices de una tabla y necesitamos indispensablemente que los mismos sean enteros distintos dentro de un rango equivalente al rango de la tabla, utilizar el método de Hashing nos permite manejar aplicaciones de búsqueda donde no tenemos claves con características tan limitadas. El resultado es un método de búsqueda completamente diferente a los métodos basados en comparaciones; ahora en vez de navegar por las estructuras comparando palabras clave con las claves en los elementos, tratamos de referenciar los elementos en una tabla directamente haciendo operaciones aritméticas para transformar claves en direcciones de la tabla. Esto último se logra implementando una **Función Hash**, que se va a encargar de dicha transformación.

Colisiones

Idealmente, todas las claves deberían corresponder con direcciones diferentes, pero es muy frecuente que dos o más claves diferentes sean transformadas a la misma dirección, cuando esto pasa, se dice que se presenta una **Colisión**. Es por eso por lo que también se debe implementar algún proceso de **resolución de Colisiones**, que se va a encargar de tratar tales situaciones. Uno de los métodos de resolución de colisiones que existen usa **Listas enlazadas** y se lo denomina “encadenamiento directo” o “Hashing abierto” el cual es muy útil en situaciones dinámicas, donde el número de elementos es difícil de predecir por adelantado.

Eficiencia

El Hashing es un buen ejemplo de balance entre tiempo y espacio. Si no hubiera limitaciones de memoria, entonces podríamos hacer cualquier búsqueda en un solo acceso simplemente utilizando la clave como una dirección de memoria. Este ideal no puede ser llevado a cabo, porque la cantidad de memoria requerida es prohibitiva cuando la cantidad de registros es considerable. De la misma manera, si no hubiese limitación de tiempo, podríamos usar un menor espacio en memoria usando un método secuencial. Hashing provee una manera de usar una razonable cantidad de memoria y tiempo y lograr un balance entre los dos extremos mencionados. En particular, podemos lograr el balance deseado simplemente ajustando el tamaño de la tabla, sin necesidad de re-escribir código o cambiar algoritmos.

En líneas generales podemos decir, que es razonable esperar búsquedas (Search), borrados (Delete) e inserciones (Insert) en tiempo constante, independientemente

del tamaño de la tabla. O sea que es ideal para aplicaciones que realicen mayoritariamente este tipo de operaciones; por el otro lado, las técnicas de Hashing no proveen implementaciones eficientes para otras operaciones como por ejemplo Ordenamiento (Sort) o Selección (Select).

Cuando hablemos de **funciones Hash** estaremos haciendo referencia a una función para resumir o identificar probabilísticamente un gran conjunto de información, dando como resultado un conjunto imagen finito generalmente menor (un subconjunto de los números naturales)

Si nos referimos a un **algoritmo de Hashing** aludimos a un algoritmo que se utiliza para generar un valor de Hash para algún **dato**, como por ejemplo **claves**. Un algoritmo de Hash hace que los cambios que se produzcan en los datos de entrada provoquen cambios en los bits del Hash. Gracias a esto, los Hash permiten detectar si un dato ha sido modificado.

Ventajas

- Se pueden usar los valores naturales de la llave, puesto que se traducen internamente a direcciones fáciles de localizar.
- Se logra independencia lógica y física, debido a que los valores de las llaves son independientes del espacio de direcciones.
- No se requiere almacenamiento adicional para los índices.

Desventajas

- No pueden usarse registros de longitud variable.
- El archivo no está clasificado.
- No permite claves repetidas.
- Solo permite acceso por una sola llave.

Las funciones hash son muy usadas; una de las utilidades que tiene es proteger la **confidencialidad** de una contraseña, ya que podría estar en texto plano y ser accesible por cualquiera y aun así no poder ser capaces de deducirla. En este caso, para saber si una contraseña que está guardada, por ejemplo, en una base de datos es igual a la que hemos introducido no se descifra el hash (ya que debería de ser imposible hacerlo) sino que se aplicará la misma función de resumen a la contraseña que especificamos y se comparará el resultado con el que tenemos guardado (como se hace con las contraseñas de los sistemas Linux).

Imaginemos que tenemos la siguiente estructura para guardar información:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|------|------|------|------|------|------|------|------|------|------|------|
| None | None | None | None | None | None | None | None | None | None | None |

Y ahora queremos guardar los siguientes datos: 54, 26, 93, 17, 77 y 31. Para hacerlo, usaremos la función de hash **módulo 11** (tamaño de la tabla), de manera que cada valor se transforma de la siguiente forma:

54 → 10

26 → 4

93 → 5

17 → 6

77 → 0

31 → 9

Como podemos ver, mediante la función de hash, “transformamos” cada número en una posición de la tabla donde guardar los datos, de manera que tendríamos lo siguiente:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|----|------|------|------|----|----|----|------|------|----|----|
| 77 | None | None | None | 26 | 93 | 17 | None | None | 31 | 54 |

Luego, para rescatar cualquiera de estos valores, simplemente aplicamos de nuevo la función de hash y ya sabremos en qué posición buscar. Búsqueda de **O(1)**. Supereficiente, y la tabla ¡ni siquiera está ordenada!

Pero hay un problema: si ahora intentamos meter el valor 32, al aplicar la función de hash nos devuelve el valor 10, que ya está ocupada. Se ha producido una **colisión**, que, como veremos en el apartado siguiente, debemos solventar de alguna forma.

Funciones de hash hay muchas; unas mejores, otras peores. En el ámbito de la **seguridad informática** y la **criptografía** existen miles. Aquí estudiaremos las más populares, bien por sencillas, bien por eficientes.

Usos

Las aplicaciones de estas funciones hash son variadas, pero vamos a revisar algunas de ellas:

- **Gestión de identificadores y contraseñas:** son ampliamente utilizadas, los proveedores de servicios en la nube o plataformas donde necesitemos datos de accesos como usuario y contraseña. Estos servicios deben comprobar si los datos introducidos son correctos, pero no almacenan estos datos, sino hashes de estos. Por ello, las plataformas te piden reiniciar la contraseña cuando solicitas recuperarla.
- **Prueba de la integridad de contenidos:** basándonos en el efecto avalancha¹ de ciertas funciones hash, podemos comprobar si un fichero ha sido manipulado, ya sea en su contenido, nombre, extensión o cualquier otro aspecto que se nos ocurra. A esto se le suele llamar **checksum criptográfico**.
- **Identificación de contenido:** en algunas aplicaciones se usa el valor hash de un fichero para identificar su contenido, independientemente de otros parámetros. Dropbox utiliza funciones hash en este sentido, un ejemplo de ellos podemos verlo en el informe revista TechCrunch. En él se nos cuenta que Dropbox bloqueó una cuenta a un usuario por compartir contenido protegido por las leyes de EEUU de derechos de autor. ¿Cómo sabía Dropbox del contenido del fichero? Sencillo, los autores legítimos del contenido habrían generado un hash del mismo, y lo habrían agregado a un listado de contenidos protegidos. Por tanto, en cuanto el usuario intentó compartir el fichero, Dropbox comparó el hash del fichero con el listado, al ver que estaba incluido en dicha lista bloqueó la operación.
- **Detección de virus:** Muchos antivirus definen funciones hash que capturan la esencia del virus generando lo que se llama la **firma del virus**. Esto permite detectarlos y distinguirlos de otros programas o virus.

Colisiones y alternativas para su solución

Si dos llaves generan un hash apuntando al mismo índice, los registros correspondientes no pueden ser almacenados en la misma posición. En estos casos, cuando una casilla ya está ocupada, debemos encontrar otra ubicación donde almacenar el nuevo registro, y hacerlo de tal manera que podamos encontrarlo cuando se requiera.

Para dar una idea de la importancia de una buena estrategia de resolución de colisiones, considérese el siguiente resultado, derivado de la **paradoja de las fechas de nacimiento**. Aun cuando supongamos que el resultado de nuestra función hash

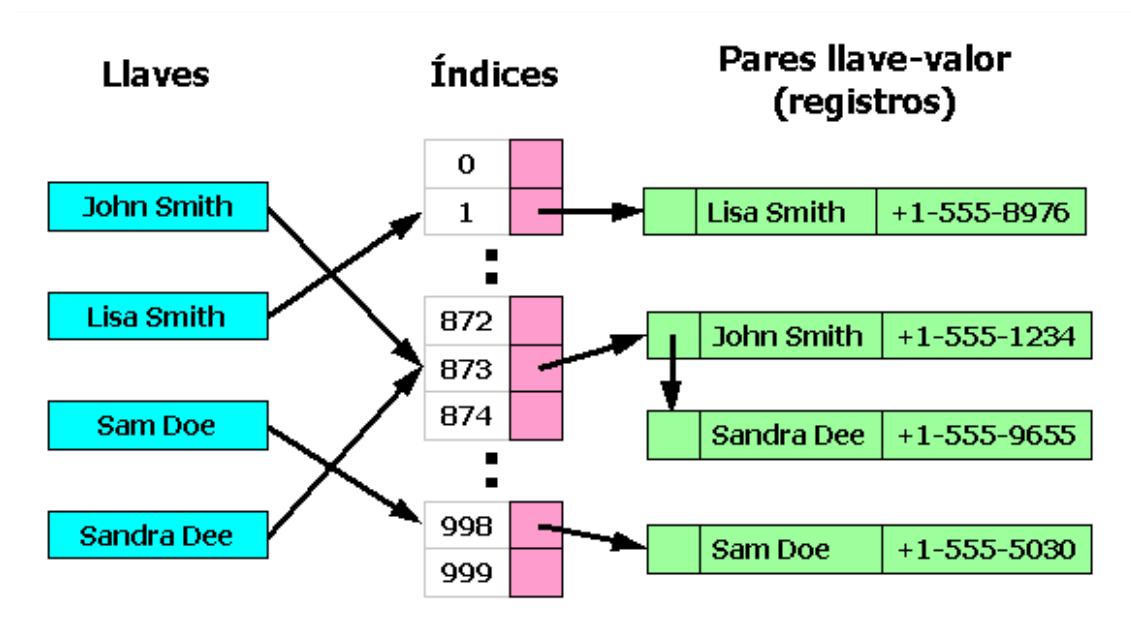
¹ El efecto avalancha se da cuando cualquier cambio en el input, sin importar si es pequeño (un bit) o grande (todo el texto) produzca que el hash varíe totalmente.

genera índices aleatorios distribuidos uniformemente en todo el vector, e incluso para vectores de 1 millón de entradas, hay un 95% de posibilidades de que al menos una colisión ocurra antes de alcanzar los 2.500 registros.

Hay varias técnicas de resolución de colisiones, pero las más populares son el **encadenamiento** y el **direccionamiento abierto**.

Encadenamiento

También conocido como **encadenamiento separado**, **Hashing abierto** o **Direccionamiento cerrado**. En la técnica más simple de encadenamiento, cada casilla en el array referencia una lista de los registros insertados que colisionan en la misma casilla. La inserción consiste en encontrar la casilla correcta y agregar al final de la lista correspondiente. El borrado consiste en buscar y quitar de la lista el elemento en cuestión.



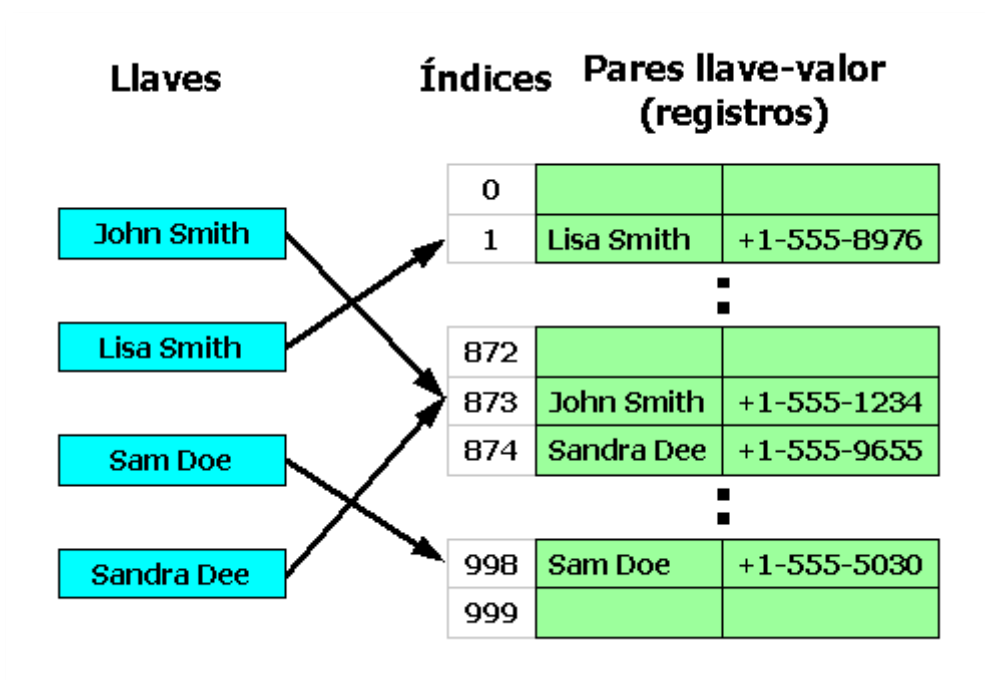
La técnica de encadenamiento tiene ventajas sobre el direccionamiento abierto. Primero el borrado es simple y segundo el crecimiento de la tabla puede ser pospuesto durante mucho más tiempo dado que el rendimiento disminuye mucho más lentamente incluso cuando todas las casillas ya están ocupadas. De hecho, muchas tablas hash encadenadas pueden no requerir crecimiento nunca, dado que la degradación de rendimiento es lineal en la medida que se va llenando la tabla. Por ejemplo, una tabla hash encadenada con dos veces el número de elementos recomendados, será dos veces más lenta en promedio que la misma tabla a su capacidad recomendada.

Las tablas hash encadenadas heredan las desventajas de las listas enlazadas. Cuando se almacenan cantidades de información pequeñas, el gasto extra de las listas puede ser significativo. También los viajes a través de las listas tienen un rendimiento de caché muy pobre.

Otras estructuras de datos pueden ser utilizadas para el encadenamiento en lugar de las listas enlazadas. Al usar árboles auto-balanceables, por ejemplo, el tiempo teórico del peor de los casos disminuye de **$O(n)$** a **$O(\log n)$** . Sin embargo, dado que se supone que cada lista debe ser pequeña, esta estrategia es normalmente ineficiente a menos que la tabla hash sea diseñada para correr a máxima capacidad o existan índices de colisión particularmente grandes. También se pueden utilizar vectores dinámicos para disminuir el espacio extra requerido y mejorar el rendimiento del caché cuando los registros son pequeños.

Direccionamiento abierto

También llamado **Hashing cerrado**. Las tablas hash de direccionamiento abierto pueden almacenar los registros directamente en el array. Las colisiones se resuelven mediante un **sondeo** del array, en el que se buscan diferentes localidades del array (secuencia de sondeo) hasta que el registro es encontrado o se llega a una casilla vacía, indicando que no existe esa llave en la tabla.



Las secuencias de sondeo más socorridas incluyen:

- **Sondeo lineal**, en el que el intervalo entre cada intento es constante (frecuentemente 1).
- **Sondeo cuadrático**, en el que el intervalo entre los intentos aumenta linealmente (por lo que los índices son descritos por una función cuadrática), y
- **Doble hasheo** en el que el intervalo entre intentos es constante para cada registro, pero es calculado por otra función hash.

El sondeo lineal ofrece el mejor rendimiento del caché, pero es más sensible al aglomeramiento, en tanto que el doble hasheo tiene pobre rendimiento en la caché, pero elimina el problema de aglomeramiento. El sondeo cuadrático se sitúa en medio. El doble hasheo también puede requerir más cálculos que las otras formas de sondeo.

Una influencia crítica en el rendimiento de una tabla hash de direccionamiento abierto es el porcentaje de casillas usadas en el array. Conforme el array se acerca al 100% de su capacidad, el número de saltos requeridos por el sondeo puede aumentar considerablemente. Una vez que se llena la tabla, los algoritmos de sondeo pueden incluso caer en un círculo sin fin. Incluso utilizando buenas funciones hash, el límite aceptable de capacidad es normalmente 80%. Con funciones hash pobremente diseñadas el rendimiento puede degradarse incluso con poca información, al provocar aglomeramiento significativo. No se sabe a ciencia cierta qué provoca que las funciones hash generen aglomeramiento, y es muy fácil escribir una función hash que, sin querer, provoque un nivel muy elevado de aglomeramiento.

Desbordamiento

Se dice que se ha producido un **desbordamiento** cuando queremos insertar una nueva clave y se aplica a una dirección de memoria completamente ocupada.

Borrado de elementos en tablas hash

Para borrar un elemento simplemente debemos tener en cuenta el tipo de resolución de colisiones que hayamos adoptado.

En el caso del **Encadenamiento**, es bien sencillo: simplemente eliminamos de la lista el elemento en cuestión y listo.

En el caso del **Direccionamiento abierto** también es muy sencillo, pues al encontrar el elemento en cuestión, simplemente liberamos la posición ocupada borrando su contenido.

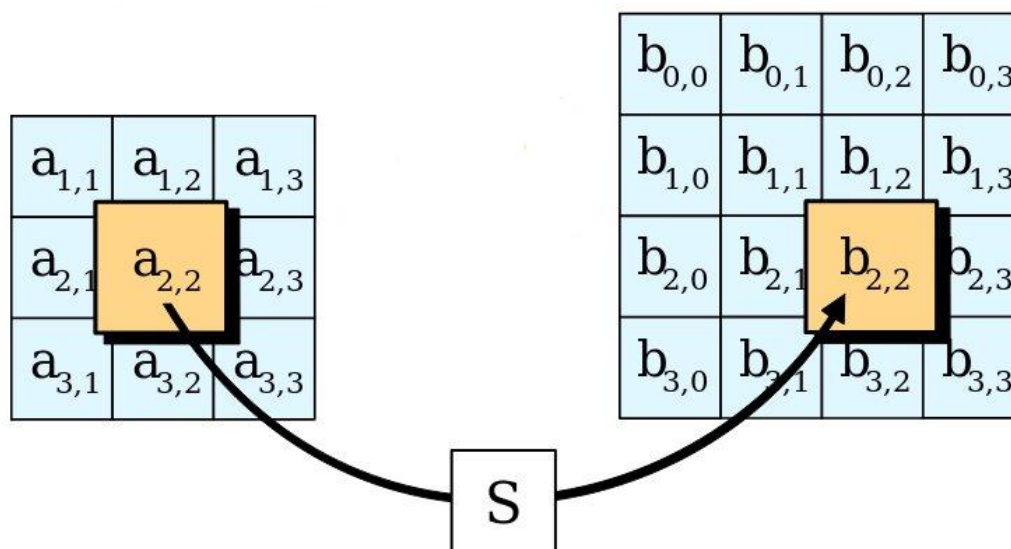
Reordenamiento en tablas hash

Cuando hablamos de **reordenamiento**, nos estamos refiriendo a que en las ocasiones en las que hemos llenado por completo (o estamos a punto de hacerlo) nuestra estructura de datos, debemos dentro de lo posible ampliar dicha estructura. Pero esto implica dos tareas:

1. Ampliar el espacio de memoria destinado a la estructura, y
2. Modificar la función de Hash para que se refiera a un espacio mayor y volver a cargar y reorganizar de nuevo los datos.

La primera tarea depende de nuestro sistema, sobre todo del hardware. Estamos hablando tanto de memoria RAM, como de disco duro o almacenamiento en general si la información será descargada o respaldada en disco.

La segunda tarea, si hemos conseguido la primera, pues no tiene sentido si no hemos obtenido más espacio para la nueva estructura, es más sencilla ya que implica un cambio de función hash y la ejecución de esta para redistribuir los datos que estaban en la antigua estructura en la nueva ubicación.



El tamaño de una tabla hash es el otro parámetro que debe ser ajustado con cierto cuidado. Al igual que en el caso de la función de hash, se puede analizar cuáles son las consecuencias extremas de la elección de un tamaño de tabla inadecuado. Por ejemplo, si el tamaño es demasiado pequeño, digamos 1, de nuevo la tabla se comporta como una lista encadenada. En cambio, si el tamaño es enorme, el malgasto de memoria es evidente, puesto que habrá un elevado número de posiciones cuya lista de colisiones está vacía.

Cuando se utilizan listas de colisión se suele utilizar una técnica basada en una “densidad”. Por densidad se entiende el valor al dividir el número de elementos en la tabla y su número de entradas. Si la función de hash produce una distribución uniforme de los elementos, un valor elevado de la densidad denota una longitud excesiva de las listas de colisiones. Un valor de densidad muy reducido denota una longitud muy corta de las cadenas de colisión y por tanto una infrautilización de memoria.

Las implementaciones más complejas de tablas hash incluyen dos umbrales para el valor de la densidad. Cada operación de inserción o borrado de un elemento consulta estos valores. Si la densidad supera el valor máximo, se reserva espacio para una tabla mayor, se borran todos los elementos de la tabla antigua y se insertan en la nueva (con densidad obviamente menor). Análogamente, si al borrar un elemento la densidad es menor que el umbral inferior, se realiza una operación similar, pero con una nueva tabla más pequeña. De esta forma se consigue mantener de una forma transparente al exterior, el valor de la densidad en límites razonables. La operación de redimensionado de la tabla es cara en tiempo de CPU, pero se supone que las operaciones futuras de búsqueda, al encontrar listas de colisión de la dimensión adecuada, rentabilizan su ejecución.

La elección de un **tamaño de tabla** adecuado es fundamental para obtener el mejor rendimiento de una tabla hash. Si seleccionamos un tamaño muy pequeño, obtenemos tiempos que pueden resultar muy malos cuando el factor de carga se acerca a 1, o incluso podemos llenar la tabla sin posibilidad de seguir insertando elementos.

Por otro lado, si escogemos un valor muy alto los tiempos serán pequeños, pero el espacio de memoria desperdiciado será mayor.

Una forma de aliviar este problema es permitir el **redimensionamiento** o **rehashing** de la tabla, es decir, modificar el tamaño de la tabla de forma dinámica. Básicamente, la idea consiste en seleccionar un tamaño relativamente pequeño, y comprobar el factor de carga cada vez que se modifica la tabla. Así, cuando el factor de carga es demasiado elevado, podemos optar por crear una nueva tabla de mayor tamaño, y volver a insertar todos los elementos que existían hasta el momento.

Lógicamente, esta operación es **muy costosa**, ya que hay que volver a realizar tantas inserciones como elementos hay en la tabla. Por tanto, habrá que tener cuidado con la estrategia de crecimiento que se establece.

Eficiencia de algoritmos hash

El proceso de creación de un valor de hash debe ser **rápido** y no debe exigir mucho poder de cálculo.

Se denomina **factor de carga**, (o densidad de carga), α , al cociente entre el número de claves en uso **m** y el número total de registros almacenables en la tabla de dispersión. Así:

$$\alpha = \frac{m}{s \cdot b}$$

donde **s** es el número de registros por bloque y **b** es el número de bloques que hay en la tabla de dispersión.

El factor de carga de la tabla es m/N , donde **m** es el número de índices distintos que se han almacenado en la tabla y **N** es el tamaño de la matriz que se emplea para implementarla.

Si suponemos que todos los índices, todos los punteros y todos los valores almacenados en la tabla ocupan una cantidad de espacio constante, entonces la tabla ocupa un espacio que está en **O (N + m)** y la longitud media de la lista es igual al factor de carga. Así un incremento de N reduce la longitud, pero incrementa el espacio.

Si se mantiene el factor de carga entre $\frac{1}{2}$ y 1, la tabla ocupa un espacio que está en **O (m)**, lo cual es óptimo salvo un pequeño factor constante, y la longitud media de la lista es menor que 1, lo cual implica un acceso eficiente a la tabla.

Resulta tentador mejorar este esquema sustituyendo las N listas de colisión por árboles equilibrados, pero no merece la pena si el factor de carga se mantiene en valores pequeños.

El factor de carga se mantiene en valores pequeños mediante una redistribución. Cuando el factor de carga supera el valor uno, se dobla el tamaño de la matriz empleada para implementar la tabla de dispersión.

La función de dispersión cambia para doblar su alcance, y todas las entradas que están en la tabla se redistribuyen a sus nuevas posiciones en alguna lista de la matriz mayor. La redistribución es costosa pero infrecuente así, no incrementa mucho el tiempo de acceso a la tabla. Cada vez que el factor de carga sobrepasa el valor uno, se reorganiza la tabla y el factor de carga vuelve a valer $\frac{1}{2}$.

Implementación

En este apartado vamos a realizar una implementación simple del **hashing cerrado**. Para ello supondremos un tipo de dato `char *` para el cual diseñaremos una función hash simple consistente en la suma de los códigos ASCII que componen dicha cadena.

Una posible implementación de la estructura a conseguir es la siguiente:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #define NCASILLAS 100
6  #define VACIO NULL
7
8  static char* BORRADO="";
9
10 typedef char **TablaHash;
11
12 void error(char*);
13 TablaHash CrearTablaHash();
14 void DestruirTablaHash(TablaHash);
15 int Hash(char*);
16 int Localizar(char*, TablaHash);
17 int Localizarl(char*, TablaHash);
18 int MiembroHash(char*, TablaHash);
19 void InsertarHash(char*, TablaHash);
20 void BorrarHash(char*, TablaHash);
21
22 int main()
23 {
24     TablaHash t;
25     t = CrearTablaHash();
26     // Obtener Hash de cadena Hola
27     printf("Hola -->%d\n", Hash("Hola"));
28     // Obtener Hash de cadena hola
29     printf("hola -->%d\n", Hash("hola"));
30     // Insertar cadena Hola en la tabla hash
31     InsertarHash("Hola", t);
32     // Mostrar contenido ubicación donde debería estar Hola
33     printf("%s\n", t[Localizar("Hola", t)]);
34     // Mostrar contenido ubicación donde debería estar hola
35     printf("%s\n", t[Localizar("hola", t)]);
36     return 0;
37 }
38
```

```
39 void error(char *mensaje)
40 {
41     printf("%c\n", *mensaje);
42 }
43
44 TablaHash CrearTablaHash()
45 {
46     TablaHash t;
47     register int i;
48
49     t=(TablaHash)malloc(NCASILLAS*sizeof(char *));
50     if(t==NULL)
51     {
52         error("Memoria Insuficiente");
53     }
54     for(i=0;i<NCASILLAS;i++)
55     {
56         t[i]=VACIO;
57     }
58     return t;
59 }
60
61 void DestruirTablaHash(TablaHash t)
62 {
63     register int i;
64
65     for(i=0;i<NCASILLAS;i++)
66     {
67         if (t[i]!=VACIO && t[i]!=BORRADO)
68         {
69             free(t[i]);
70         }
71     }
72     free(t);
73 }
74
```

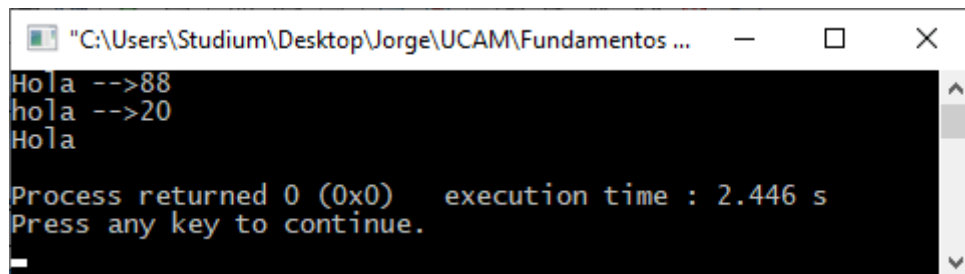
```
75 int Hash(char *cad)
76 {
77     int valor;
78     char *c;
79
80     for(c=cad, valor=0; *c; c++)
81     {
82         valor += (int)*c;
83     }
84     return(valor%NCASILLAS);
85 }
86
87 int Localizar(char *x, TablaHash t)
88 // Devuelve el sitio donde esta x o donde deberia de estar
89 // No tiene en cuenta los borrados
90 {
91     int ini, i, aux;
92
93     ini=Hash(x);
94     for(i=0; i<NCASILLAS; i++)
95     {
96         aux=(ini+i)%NCASILLAS;
97         if(t[aux]==VACIO)
98         {
99             return aux;
100         }
101         if(!strcmp(t[aux],x))
102         {
103             return aux;
104         }
105     }
106     return ini;
107 }
108
```



```
109 int Localizar1(char *x, TablaHash t)
110 // Devuelve el sitio donde podríamos poner x
111 {
112     int ini,i,aux;
113
114     ini=Hash(x);
115     for(i=0;i<NCASILLAS;i++)
116     {
117         aux=(ini+i)%NCASILLAS;
118         if(t[aux]==VACIO || t[aux]==BORRADO)
119         {
120             return aux;
121         }
122         if(!strcmp(t[aux],x))
123         {
124             return aux;
125         }
126     }
127     return ini;
128 }
129
130 int MiembroHash(char *cad, TablaHash t)
131 {
132     int pos=Localizar(cad,t);
133
134     if(t[pos]==VACIO)
135     {
136         return 0;
137     }
138     else
139     {
140         return(!strcmp(t[pos],cad));
141     }
142 }
143
```

```
144 void InsertarHash(char *cad, TablaHash t)
145 {
146     int pos;
147
148     if(!cad)
149     {
150         error("Cadena inexistente");
151     }
152
153     if(!MiembroHash(cad,t))
154     {
155         pos=Localizar1(cad,t);
156         if (t[pos]==VACIO || t[pos]==BORRADO)
157         {
158             t[pos]=(char *)malloc((strlen(cad)+1)*sizeof(char));
159             strcpy(t[pos],cad);
160         }
161         else
162         {
163             error("Tabla Llena");
164         }
165     }
166 }
167
168 void BorrarHash(char *cad, TablaHash t)
169 {
170     int pos = Localizar(cad,t);
171
172     if(t[pos]!=VACIO && t[pos]!=BORRADO)
173     {
174         if(!strcmp(t[pos],cad))
175         {
176             free(t[pos]);
177             t[pos]=BORRADO;
178         }
179     }
180 }
```

Si ejecutamos el siguiente código...



```
"C:\Users\Stodium\Desktop\Jorge\UCAM\Fundamentos ..."
Hola -->88
hola -->20
Hola
Process returned 0 (0x0)   execution time : 2.446 s
Press any key to continue.
```

Referencias

Departamento de Informática Universidad de Valladolid, en [enlace](#).

03/09/2020