UNIVERSIDAD NACIONAL AUTÓNOMA
DE MÉXICO

FACULTAD DE CIENCIAS

FUNCIONES HASH CRIPTOGRÁFICAS

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

P R E S E N T A :

NOMBRE DEL ALUMNO
JORGE ALBERTO MEDINA ROSAS

TUTORA
DRA. ELISA VISO GUROVICH

2009

1.	Datos del alumno Medina Rosas Jorge Alberto 56 18 16 70 Universidad Nacional Autonoma de Mexico Facultad de Ciencias Ciencias de la Computación 098159820
2.	Datos del tutor Dr Elisa Viso Gurovich
3.	Datos del sinodal 1 Dr José de Jesús Galaviz Casas
4.	Datos del sinodal 2 Dr Sergio Rajsbaum Gorodesky
5.	Datos del sinodal 3 L en C C Francisco Solsona Cruz
6.	Datos del sinodal 4 L en C C Manuel Alberto Sugawara Muro
7.	Datos del trabajo escrito Funciones Hash Criptográficas 68 p 2009

A mi familia y a mis amigos.

Índice general

Índice de figuras

Introducción

Un problema práctico o de la vida cotidiana es el que se refiere a la ubicación de elementos específicos dentro de un conjunto finito de tal manera que su localización sea precisa y veloz, como se diría popularmente, "buscar una aguja en un pajar".

Ahora bien, dentro del contexto de ciencias de la computación podemos reformular este problema de la siguiente manera: queremos localizar ubicaciones de memoria u objetos de información dentro de una estructura de datos de la manera más rápida posible y usando la menor cantidad posible de memoria. La manera de abordarlo en este trabajo será usando ciertos algoritmos llamados *tablas hash*, *tablas de dispersión* o *tablas de distribución*, los cuales implementaremos a partir de una estructura de datos muy simple llamada *diccionario*, que describiremos en el capítulo 1.

El problema de las búsquedas de información ha sido bastante estudiado e interpretado en las ciencias de la computación. El saber cuánto tiempo lleva encontrar algo en memoria es a veces un requisito, sin importar la cantidad de elementos reservados en memoria. Además, una velocidad eficiente es una condición muchas veces necesaria que obliga a utilizar una buena estructura de datos y un buen algoritmo para las búsquedas.

Una posible manera de localizar elementos es por medio de etiquetas o identificadores que sirven como medio de referencia del elemento. Mientras exista una mayor cantidad de elementos en cuestión, tanto mejor será el uso de un método de identificación eficiente sobre los elementos del conjunto. Un posible método es la asignación de una *llave* (un arreglo de carácteres) a cada elemento; la llave se toma de un conjunto de llaves y en general se puede obtener fácilmente, por ejemplo el RFC de una "persona física".

Una condición que debe cumplir esta llave es la de ser única, esto es, a cada elemento se le asigna una y sólo una llave; además podemos fijar una longitud máxima de llave, digamos de 8 letras de las 27 del alfabeto, asegurando así, para este caso, la unicidad de las llaves para conjuntos de hasta 2^{11} elementos (unos trescientos mil millones de elementos). Esta asociación de llaves da un pequeño salto en el problema ya que reduce de manera considerable la longitud de entrada para realizar su búsqueda; es más eficiente buscar a una persona por su CURP que por su cadena de ADN, ya que leer un CURP de 18 letras es mucho más rápido que leer una cadena de miles de millones de nucleótidos; sin embargo, la pregunta aún persiste: ¿de qué manera es que conociendo esta diminuta "etiqueta" podemos recuperar eficazmente el elemento asociado?

Nuestra respuesta: usando funciones de hash, lo cual explicaremos a lo largo del capítulo 2.

Las tablas hash nos dan la certeza de que las búsquedas, en promedio, se llevarán acabo con una complejidad de a lo más $O(1)$ de tiempo, es decir, el tiempo de búsqueda es constante; esto quiere decir que no dependen del número de elementos de la estructura de datos; el orden de complejidad de las búsquedas en los diccionarios puede llegar a lo más a $O(n)$ si se utilizan listas ligadas o hasta $O(\log(n))$ usando el conocido algoritmo de búsqueda binaria (en alguna estructura de datos que la permita). Ambos valores se encuentran por encima de $O(1)$.

La mayor desventaja de las tablas hash son las llamadas *colisiones*. Esto sucede ya que en la mayoría de los casos el dominio, en este caso el conjunto de llaves, es mucho mayor que su contradominio, el conjunto de direcciones de memoria, de modo que la *función hash* (la caja negra de nuestro algoritmo) no es inyectiva, provocando que existan valores $f(k_1) = f(k_2)$ para k_1 y k_2 dos diferentes llaves. Para disminuir las colisiones se buscan funciones hash que distribuyan uniformemente los valores de todas las llaves en el contradominio, intentando además que las llaves parecidas produzcan valores muy diferentes.

El objetivo del capítulo 2 será el de revisar los conceptos arriba expuestos, presentando las funciones hash que son más conocidas y usadas. Y presentaremos una serie de resultados experimentales para darnos una idea, de un modo informal, del comportamiento real de estas funciones ante diferentes conjuntos de entrada.

Ligaremos el tema de las *funciones hash criptográficas* al de distribuir uniformemente los valores hash, usando la siguiente hipótesis: mientras mejor se cifren las llaves tanto mejor se dispersarán los valores que regresará la función hash, y de esta manera reduciremos el número de colisiones en la tabla, aunque el costo será cargar con llaves más grandes y más complejas de construir por los pasos de cifrado. Esto lo explicaremos en el capítulo 3.

Si bien es cierto que el problema de encontrar una función aleatoria o aproximadamente aleatoria difiere un tanto del problema de encontrar funciones de cifrado o de firmas digitales, también es cierto que una función de encripción debería de proveer una cierta aleatoriedad en cuanto a su salida, dos valores o dos llaves de caracteres semejantes después de evaluarse en una encripción se espera que resulten ser dos resultados muy diferentes, de lo contrario su criptoanálisis evidenciaría fallos, y además esperando que el problema de invertir el cifrado sea equivalente a un problema no polinomial bajo cierta cota conocida que establece el número de elementos diferentes necesarios para que la probabilidad de encontrar una colisión sea $1/2$, "el problema del cumpleaños". Casualmente esto forma parte de la definición de funciones hash criptográficas, informalmente consideramos que la propiedad fuerte de una

función hash criptográfica es: ser muy difícil encontrar un segundo elemento x' del dominio tal que su evaluación bajo la función sea igual al de una evaluación conocida anteriormente $h(x)$, con $x \neq x'$, definiciones 3.1.2 y 3.1.3.

El objetivo del capítulo 3 será explicar exactamente la definición de funciones hash criptográficas, revisaremos 3 diferentes métodos generales que permiten construcciones "confiables" y presentaremos nuestros resultados experimentales de la misma manera que en el capítulo 2. Y además llevaremos a cabo la construcción de una función hash criptográfica basándonos en dos algoritmos muy populares de firmas digitales, MD5 y SHA1. Por último, realizaremos un simulacro de pruebas para evaluar la eficiencia de nuestra aproximación de función hash criptográfica.

Diccionarios

1.1. Ideas generales

Cuando hablamos de una estructura de datos que implementa los métodos de `agregar` un elemento, `borrar` un elemento y `contener?` un elemento nos referimos a un diccionario.

Los diccionarios son estructuras de datos que guardan entradas de parejas (`llave`, `elemento`), donde cada llave mapea a un único elemento.

Veamos el código de una interfaz en *Java* de un diccionario:

```
import java.util.Iterator;

public interface Diccionario{
  public Object agrega( Object llave,
                        Object elemento );
  public Object daElemento( Object llave );
  public Object daLlave( Object elemento );
  public boolean contieneElemento( Object llave );
  public Iterator daLlaves();
  public Iterator daElementos();
  public Object borraElemento( Object llave );
  public int longitud();
  public String toString();
}
```

Los métodos que requiere implementar la interfaz son los siguientes:

- El método agrega inserta una nueva pareja de elementos *(llave, elemento)*; si la llave había sido asignada anteriormente se reemplaza su elemento asociado y se regresa el elemento anterior, de lo contrario agrega la nueva pareja y regresa *null*.

- El método daElemento regresa el elemento usando la llave que lo identifica o *null* en el caso de no encontrarlo.

- El método daLlave regresa la llave usando el elemento asociado dentro de la estructura. Este método existe sólo por convención ya que es la llave la que caracteriza al elemento asociado y no al contrario, dentro del diccionario podrían existir entradas con elementos iguales pero llaves distintas (en la implementación del apéndice A el método regresa la primer llave encontrada).
- El método contieneElemento pregunta si se encuentra la pareja *(llave, elemento)* en el diccionario.
- Los métodos de enumeración daLlaves y daElementos devuelven una iteración de cada conjunto.
- El método longitud regresa el número de entradas del diccionario.
- El método toString regresa el diccionario en forma de cadena.

En el apéndice A tenemos una implementación del diccionario soportado por una lista; ahí podemos observar que los métodos de agregar, de búsqueda y de borrado llevan tiempo a lo más $O(n)$, ya que se va revisando la lista secuencialmente hasta encontrar el elemento requerido.

Podemos también implementar un diccionario usando un árbol binario (comparando las llaves) en vez de usar una lista y en este caso los métodos llevan tiempo a lo más $O(\log(n))$.

Tablas hash o de dispersión

2.1. Ideas generales

Podemos pensar en una tabla hash como un sistema de almacenamiento o estructura de datos, que contiene entradas en una tabla o listado y que para cada entrada almacena una llave distinta; esto implica que una llave identifica unívocamente a una entrada y la distingue del resto de las demás entradas. Una entrada además puede contener alguna información asociada a la llave. En abstracto, podemos pensar en una entrada de la tabla como una pareja ordenada (k, e) que consiste de una llave k y de una información asociada e, tal que para un mismo valor de e no puede suceder que se le asocie un valor distinto del valor de k.

Entonces tenemos que una tabla hash no es más que una tabla de entradas o elementos de la forma (k, e), tal como se muestra en la tabla 2.1.

Llave	**Entrada**
1	Dana
2	Eva
3	Greta
4	Helena
5	Marie

Tabla 2.1: *Tabla Hash que contiene entradas* (k, e).

Ahora bien, ¿cómo es que realizamos las búsquedas? O dicho de otra manera, ¿cómo es que dada la llave k, encontramos la entrada con la pareja (k, e)? La manera de hacerlo es usando una función $h : \mathcal{K} \to \mathbb{N}$ que para cada llave $k \in \mathcal{K}$, $h(k)$ evalúa la dirección de memoria de la entrada (k, e), esto se denomina *dispersión* o *hashing*. Idealmente, podríamos pensar en mapear todas las llaves posibles a direcciones de memoria, pero en la mayoría de los casos el número de posibles llaves (y a veces hasta el número de llaves usadas realmente) excede en mucho el número de posibles direcciones de memoria. Por ejemplo, podríamos tener el siguiente caso: una llave de contraseña de 8 carácteres, que permite minúsculas, mayúsculas y signos de exclamación necesitaría aproximadamente de 2^{15} (casi dos mil billones) direcciones de memoria para almacenar todas las entradas (k, e), siendo actualmente casi imposible contar con ese número de usuarios.

Para no desperdiciar tanta memoria, en la práctica se permiten *colisiones*, es decir, a diferentes llaves k_1 y k_2 el resultado de calcular $h(k_1)$ coincide con el de $h(k_2)$, lo cual significa que a dos llaves les toca la misma dirección de memoria. En estos casos, obviamente, no podemos guardar las dos entradas (k_1, e_1) y (k_2, e_2) en la misma posición de la tabla. Bajo estas circunstancias tenemos que adoptar algún método de corrección para encontrar espacio adicional donde guardar una de las dos entradas. Existen básicamente dos métodos de implementar una tabla hash para manejar colisiones: por *encadenamiento* o por *direccionamiento abierto*, los cuales veremos más adelante en la sección 2.3.

Podemos definir una tabla hash como un diccionario que contiene entradas con llaves únicas y que utiliza una función de hash h para encontrar la posición de cada entrada dentro del diccionario, entonces tenemos que la función manda el conjunto de llaves a un dominio entero más reducido de direcciones de memoria o posiciones dentro de una lista,

$$h : \mathcal{K} \to \mathbb{Z}_n. \tag{2.1}$$

Con esta definición nos damos cuenta que las tablas hash son una estructura de datos que busca elementos en tiempo constante, siempre que no existan colisiones, es decir, el tiempo de búsqueda de un elemento está determinado por el tiempo que lleva calcular la función hash y no depende del número de entradas de la tabla. Sin embargo, si el número de colisiones es considerable se presenta el peor de los casos en donde la complejidad de búsqueda es por lo menos $\theta(n)$; en la práctica se diseñan funciones que produzcan pocas colisiones (10 % del total de elementos de la tabla). En lo posible se buscan funciones fáciles de calcular y que dispersen lo mejor posible el rango de direcciones.

Dos criterios principales al seleccionar una función hash son los siguientes:

1) La función deberá producir tan pocas colisiones como sea posible. En la medida de lo posible, deberá intentar distribuir uniformemente las llaves sobre todo el conjunto $\mathbb{Z}_n$ de direcciones de memoria. Por supuesto, a menos que las llaves se conozcan con anterioridad, el determinar si una función hash dispersará sus valores se vuelve algo un poco complicado. Sin embargo, aunque es raro conocer las llaves que serán usadas antes de seleccionar la función hash es un poco más común conocer propiedades acerca de la construcción de las mismas.

2) La función debe ser fácil y rápida de calcular. Aunque la función proporcione direcciones únicas, no es buena si se consume más tiempo en su evaluación que buscar en la mitad de la tabla.

2.2. Hashing

En algunos casos podemos tener un conjunto reducido de llaves que pueden ser usadas como entradas en nuestra tabla; en estos casos es posible reservar una entrada en la tabla para cada llave posible. Por ejemplo, supongamos que tenemos un juego de cartas en el cual hay 52 cartas en juego; supongamos además que queremos guardar alguna información adicional a cada carta, tal como saber si ha sido vista en el transcurso del juego. Bajo estas circunstancias puede ser posible tener una tabla T con 52 entradas, reservando a cada entrada una carta en particular. Entonces necesitamos una función para mapear una llave k, que identifica una carta en particular, a una dirección de la tabla. Estas direcciones podrían ser enteros en el rango de $[0, \ldots, 51]$.

En otros casos, a pesar de tener un número muy grande de posibles llaves sólo una fracción de ellas aparecerá en la tabla. Por ejemplo, supongamos que queremos guardar una tabla con la información de empleados para una compañía que tiene contratadas a 5000 personas; entonces podríamos usar el RFC para identificar unívocamente a cada entrada de la tabla. En el caso del RFC, éste se construye de la siguiente manera: 4 letras que identifican el nombre de la persona, 6 dígitos para la fecha de nacimiento y 3 carácteres alfanuméricos que forman la homoclave; en este caso tendríamos aproximadamente 10^{16} posibles llaves de RFC; si tenemos 5000 empleados aproximadamente sólo el 5×10^{-11} por ciento de todas las posibles llaves son empleadas para tener un conjunto de llaves únicas, de tal manera que a cada entrada de empleado le corresponda una única llave en la tabla.

Estos últimos casos son los que nos interesan; por tanto necesitamos una función que cumpla con las siguientes características:

- El resultado de la función deberá estar completamente determinado por la entrada, es decir, solamente la llave determina el resultado. Si ocupamos algo además de la llave esto podría influir en una mala distribución de los valores de la función.

- Se usa toda la entrada para calcular el resultado, lo que se traduce en usar todos los bits de la llave. Si la función no utiliza todos los bits, entonces pequeñas variaciones en las entradas provocan un número inapropiado de valores hash similares, resultando en colisiones.

- La función distribuye uniformemente las entradas a través de todo el rango de la función. Entre más uniforme el rango de la función más se distribuyen los valores provocando menos colisiones.

- La función genera resultados completamente diferentes para entradas muy parecidas. En la práctica muchos conjuntos de llaves contienen elementos muy parecidos y queremos que estos elementos también se dispersen en nuestro contradominio.

Veamos un ejemplo en lenguaje C de una función hash:

```
int hash( char* cad, int table_size ){
    int sum;
    for( ; *cad; cad++ )
        sum += *cad;
    return sum % table_size;
}
```

Analicemos qué reglas satisface la función hash anterior. La regla 1 se cumple, pues el resultado está totalmente determinado por la llave ya que el valor hash es sólo la suma de sus carácteres. La regla 2 se cumple pues cada carácter es sumado. La regla 3 no se cumple: aun cuando a simple vista no se aprecie que la función no distribuya uniformemente las llaves, al analizarla más rigurosamente encontramos que para muchas entradas, estadísticamente, no se distribuyen bien. La regla 4 no se cumple ya que la cadena "bog" y la cadena "gob" se mapean a lo mismo: pequeñas variaciones en las llaves deberían producir diferentes valores hash pero en este caso no es así. En resumen esta función hash no es tan buena; sólo lo es para ilustrar un ejemplo.

Dirección	Elemento	Elemento	Elemento	Elemento
0	A_0	H_7	$\widetilde{N}_{14}$	U_{21}
1	B_1	I_8	O_{15}	V_{22}
2	C_2	J_9	P_{16}	W_{23}
3	D_3	K_{10}	Q_{17}	X_{24}
4	E_4	L_{11}	R_{18}	Y_{25}
5	F_5	M_{12}	S_{19}	Z_{26}
6	G_6	N_{13}	T_{20}	

Tabla 2.2: *Tabla Hash que usa la función módulo* 7.

A continuación mencionaremos otro ejemplo de una función hash muy sencilla. En este ejemplo utilizaremos como llaves letras del alfabeto con subíndices

$A_1, B_2, \ldots, Z_{26}$. Cada subíndice es un entero dando la posición de las letras en orden alfabético. Tendremos una tabla hash T de tamaño 7, desde la posición 0 hasta la posición 6. Nuestra función hash será dividir el subíndice por 7 y luego devolver el residuo resultante de la división, que es equivalente a la función $h(L_n) = (n \text{ mód } 7)$; entonces tendremos la tabla 2.2 llena con el alfabeto.

Una buena función hash $h(L_n)$ mapearía llaves de una manera uniforme al conjunto total de posibles direcciones de 0 a 6 en la tabla; en el ejemplo se ve una buena dispersión por parte del módulo primo 7, en cuanto a que hay el mismo número de elementos por dirección.

2.2.1. Método de la división

El método de la división es particularmente sencillo, ya que simplemente usamos el residuo módulo m

$$h(k) = k \pmod{m}. \tag{2.2}$$

Una buena idea es que el tamaño de la tabla sea algo más grande que el número de elementos que se desea insertar. Cuanto mayor sea el tamaño de la tabla, mayor será el rango de la imagen de la función hash, y por tanto se disminuye la probabilidad de que se produzcan colisiones. Por supuesto, esto conlleva un compromiso entre espacio y tiempo: dejar posiciones en blanco es ineficiente en términos de espacio, pero reduce el número de colisiones y, por lo tanto, es más eficiente en términos de tiempo.

Los mejores resultados con el método de la división se producen cuando el tamaño de la tabla corresponde a un número primo, ya que así, en general, la función dispersa de manera más uniforme sobre todas las posibles posiciones de la tabla y el número de colisiones se minimiza.

¿Qué condiciones deberá cumplir m para ser suficientemente bueno? No debería ser par ya que mapearía las llaves de tamaño par a entradas pares, y llaves de tamaño impar a entradas impares, rompiendo con la uniformidad. Aún peor sería definir m como una potencia del tamaño de una "palabra", ya que k mód m sería simplemente calcular los bits menos significativos de k. [Knuth] refiere buenos candidatos para m a primos tales que $r^m \neq \pm a \pmod{m}$, donde r es el tamaño del alfabeto de la llave y m, a son enteros pequeños.

2.2.2. Método de la multiplicación

El método de hashing multiplicativo es igual de fácil de hacer, pero un poco más complicado de describir ya que tenemos que imaginarnos trabajando con fracciones

en vez de con enteros. Primero multiplicamos la llave k (carácter por carácter) por una constante A en el rango $0 < A < 1$, extraemos la parte fraccionaria de kA, multiplicamos este valor por m y tomamos el piso del resultado; en otras palabras la función hash es la siguiente:

$$h(K) = \left\lfloor m\big(kA - \lfloor kA \rfloor\big) \right\rfloor \tag{2.3}$$

donde $(kA - \lfloor kA \rfloor)$ significa tomar la parte fraccionaria.

Una ventaja del método de la multiplicación es que el valor de m no es crítico, sino que se utiliza para ampliar el rango hasta m. Comúnmente se escoge una potencia de 2, es decir, $m = 2^p$ para algún entero p, ya que así la implementación resulta ser más fácil. Supongamos que la "palabra" de la máquina es de w bits y que k cabe en una sola "palabra". Entonces primero multiplicamos k por el entero de w bits $\lfloor A \cdot 2^w \rfloor$, para convertir A en un entero. El resultado es un valor de $2w$ bits $r_1 2^w + r_0$ donde r_0 es la parte fraccionaria de la multiplicación y el valor deseado de p bits consiste en los p bits más significativos de r_0.

Este método trabaja con cualquier constante A, aunque trabaja mejor con ciertos valores. La opción óptima depende de las características de las llaves a mapear. [Knuth] discute las opciones de A y sugiere que si

$$A \simeq (\sqrt{5} - 1)/2 = 0.6180339887,$$

es muy probable que trabaje razonablemente bien.

2.3. Manejo de colisiones

2.3.1. Encadenamiento

En el método de encadenamiento, para la resolución de colisiones, insertamos a todos los elementos que se mapean en la misma posición en una lista ligada. La posición j contiene una referencia a la cabeza de la lista de todos los elementos que se mapean en j; si no hay elementos la referencia contiene *null*. La figura 2.1 muestra como se vería una tabla hash con encadenamiento.

Los métodos con encadenamiento quedarían implementados de la siguiente manera:

- Agregar (T, x): Agrega x en la cabeza de la lista $T[h(k)]$.
- Búsqueda (T, x): Busca el elemento con la llave k en la lista $T[h(k)]$.

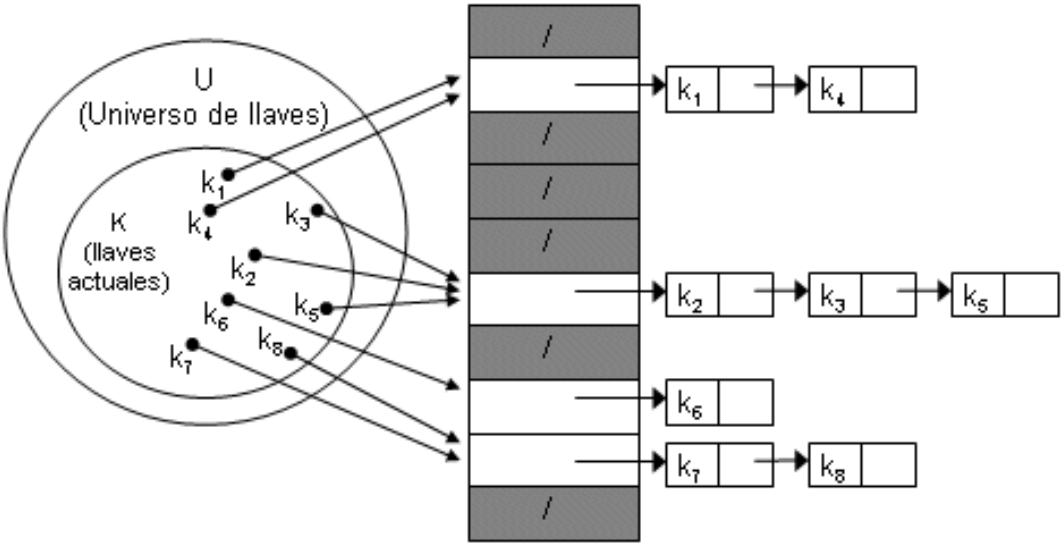

Figura 2.1: *Método por encadenamiento.*

- Borrar (T, x): Borra x de la lista $T[h(k)]$.

El tiempo que se lleva en agregar a un elemento es a lo más $O(1)$. Para las búsquedas el peor caso es proporcional al tamaño de la lista, al igual que en el borrado de elementos. Ahora bien, ¿cómo se comporta la dispersión en el encadenamiento? y en particular, ¿cuánto tiempo toma buscar un elemento dada una llave?

Dada una tabla T con m lugares que guarda n elementos, definimos el *factor de carga* α para T como $\frac{n}{m}$, que será el promedio de elementos en el encadenamiento.

El comportamiento del peor caso de hashing con encadenamiento es muy malo ya que todas las n llaves se mapean a la misma entrada de la tabla, creando una lista de longitud n. El tiempo del peor caso para las búsquedas es al menos $\theta(n)$, además del tiempo que se consume en calcular la función hash, lo que no resulta mejor que usar una lista ligada para todos los elementos. Claramente la decisión de usar tablas hash no se basa en su desempeño en el peor caso.

El desempeño promedio del hashing por encadenamiento depende de que tan bien la función hash distribuya el conjunto de llaves almacenadas de entre las m entradas de la tabla. Podemos suponer que cualquier elemento es igualmente probable de mapear a alguna de las m entradas, independientemente de los elementos que ya han sido mapeados. Llamamos a esta suposición *hashing simple uniforme*.

Podemos suponer también que el valor hash $h(k)$ se calcula en a lo más $O(1)$ de

tiempo, de tal manera que el tiempo requerido para buscar un elemento con llave k depende linealmente de la longitud de la lista $T[h(k)]$. Haciendo a un lado el tiempo constante que lleva calcular la función hash y el tiempo usado en acceder a la entrada $h(k)$, consideremos el número esperado de elementos revisados por el algoritmo de búsqueda, esto es, el número de elementos en la lista $T[h(k)]$ que se revisan para comparar si las llaves son iguales a k. Podemos considerar dos casos: la búsqueda es exitosa, cuando se encuentra al elemento con la llave k o la búsqueda es infructuosa cuando ningún elemento en la tabla tiene la llave k; este último sería el peor caso de los dos.

Teorema 2.3.1. *En una tabla hash en donde las colisiones son resueltas por encadenamiento y suponiendo hashing simple uniforme, una búsqueda infructuosa en promedio consume un tiempo de $\theta(1+\alpha)$, donde α es el factor de carga.*

Demostración. Usando la suposición del hashing uniforme tenemos que cualquier llave es igualmente probable de mapear hacia alguna entrada m. Entonces el tiempo promedio que se necesita para buscar elementos que no radican en la tabla será igual al de comparar cada nodo de la lista que existe en la entrada m de la tabla. Y en promedio la longitud de cualquier lista es igual al factor de carga $\alpha = \frac{n}{m}$. Entonces el número esperado de elementos a comparar en esta búsqueda es α. Entonces tenemos que el tiempo total requerido para realizar una búsqueda sin éxito incluyendo el tiempo que lleva calcular el valor de $f(x)$ es igual a $\theta(1+\alpha)$. □

Teorema 2.3.2. *En una tabla hash en donde las colisiones son resueltas por encadenamiento y suponiendo hashing simple uniforme, una búsqueda exitosa en promedio consume un tiempo de $\theta(1+\alpha)$, donde α es el factor de carga.*

Demostración. Podemos suponer que la llave a buscar es igualmente probable de ser cualquiera que exista en la tabla. Para encontrar el número esperado de elementos examinados podemos utilizar el promedio de todas las longitudes que puede tener una lista que exista en alguna entrada de la tabla cuando el total de elementos que existen es menor o igual a n. Entonces la longitud esperada de cada lista es $\frac{i}{m}$ donde i va desde 1 hasta n y esto implica que el número esperado de elementos examinados en una búsqueda exitosa es igual a calcular el promedio de estas longitudes

$$\begin{aligned}\frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{i-1}{m}\right) &= 1+\frac{1}{nm}\sum_{i=1}^{n}(i-1)\\ &= 1+\left(\frac{1}{nm}\right)\left(\frac{(n-1)n}{2}\right)\\ &= 1+\frac{\alpha}{2}-\frac{1}{2m}\end{aligned}$$

Entonces el tiempo total requerido para una búsqueda exitosa, incluyendo el tiempo que lleva calcular la función hash, es $\theta\left(2 + \frac{\alpha}{2} - \frac{1}{2m}\right) = \theta(1 + \alpha)$. □

¿Qué significa este análisis? Quiere decir que si el número de entradas de la tabla hash es al menos proporcional al número de elementos en la tabla, tenemos que $n = O(m)$ y consecuentemente, $\alpha = \frac{n}{m} = \frac{O(m)}{m} = O(1)$. Entonces las búsquedas, en promedio, consumen un tiempo constante. Y con esto tenemos que en promedio todas las operaciones que implementa un diccionario son soportadas en una Tabla Hash en un tiempo de a lo más $O(1)$.

2.3.2. Direccionamiento abierto

En el método por direccionamiento abierto todos los elementos se guardan en la tabla misma. Entonces cada entrada de la tabla consiste de un elemento o de *null* en el conjunto dinámico. Para buscar un elemento examinamos sistemáticamente las entradas de la tabla hasta encontrar el elemento deseado, o hasta que sea claro que el elemento no existe dentro de la tabla. Aquí no existen las listas ni los elementos fuera de la tabla como pasa en el método por encadenamiento, lo que implica que en el direccionamiento abierto la tabla hash puede ser llenada de tal manera que no sea posible hacer más inserciones; además el factor de carga α nunca puede exceder 1.

La ventaja del direccionamiento abierto es que se elimina el uso de referencias de memoria; en vez de usar referencias se calcula la secuencia de entradas a examinar para buscar la llave. La memoria liberada por no guardar referencias provee a la tabla hash con un número mayor de entradas por la misma cantidad de memoria, aportando potencialmente un número menor de colisiones y una recuperación más rápida.

Para realizar inserciones usando direccionamiento abierto examinamos secuencialmente la tabla hash hasta que encontramos un lugar vacío donde poner la llave. En vez de usar un orden fijo $[0, 1, \ldots, m-1]$ (el cual requiere $\theta(n)$ de tiempo), la secuencia de posiciones depende de la llave que se quiere insertar. Para determinar la entrada a probar extendemos la función hash para incluir la posición de prueba como un segundo argumento. Entonces la función hash h se convierte en:

$$h' : \mathcal{K} \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\} \tag{2.4}$$

Y necesitamos que para cada llave k de la secuencia de prueba $\langle h'(k, 0), h'(k, 1), \ldots, h'(k, m-1)\rangle$ sea una permutación de $\langle 0, 1, \ldots, m-1\rangle$ de tal manera que cada posición de la tabla sea considerada eventualmente como una posible entrada de la nueva llave.

En la tabla 2.3 podemos ver el pseudo-código del método de insertar.

En el algoritmo de búsqueda, para localizar una llave k se prueba la misma secuencia de entradas que en el algoritmo de inserción. Entonces la búsqueda puede terminarse cuando se encuentra una entrada vacía, ya que k habría sido insertada ahí.

```
i=0
repeat j = h(k,i)
       if T[j] = null
          then T[j] = k
               return j
          else i++
until i = m
```

Tabla 2.3: *Pseudo-código del método insertar.*

En la tabla 2.4 mostramos el pseudo-código del método de buscar.

```
i=0
repeat j = h(k,i)
       if T[j] = k
          then return j
       i++
until T[j] = null or i = m
```

Tabla 2.4: *Pseudo-código del método buscar.*

El borrado de elementos en una tabla hash con direccionamiento abierto es un poco más difícil. Cuando borramos una llave de una entrada i no podemos simplemente marcar la entrada como vacía guardando un *null*. Haciendo esto haría imposible encontrar llaves que hayan pasado por i y la hayan encontrado ocupada. Una solución es marcar la entrada guardando un valor especial de borrado en lugar de *null*. Entonces tenemos que modificar el método de búsqueda para que siga buscando cuando encuentre el valor de borrado, mientras que el método de inserción trataría a esa entrada como vacía. Haciendo esto el tiempo de búsqueda no depende del factor de carga α y por esta razón el método de encadenamiento es más usado como un método de resolución de colisiones cuando las llaves van a ser borradas constantemente.

Tres métodos son comúnmente usados para calcular la secuencia de entradas en el encadenamiento abierto: lineal, cuadrático y doble hashing. Estos métodos aseguran que la secuencia $\langle h'(k,0), h'(k,1), \ldots, h'(k,m-1)\rangle$ sea una permutación de $\langle 0, 1, \ldots, m-1\rangle$ para cada llave k.

Método lineal

En el método lineal tenemos una función hash $h' : \mathcal{K} \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}$ de la forma:

$$h'(k,i) = (h(k) + i) \pmod{m} \tag{2.5}$$

para $i \in [0, 1, \ldots, m-1]$. Dada una llave k, la sucesión a probar es una sucesión consecutiva $[h(k), h(k)+1, \ldots, 0, \ldots, h(k)-1]$ que recorre todas las posiciones de la tabla.

Una desventaja de este método es que a la larga se forman puntos de acumulación, es decir, sucesiones muy grandes de entradas consecutivas que se forman a lo largo de la tabla.

Por ejemplo, consideremos una tabla con 19 posiciones y 9 elementos en ella, tal como se muestra en la figura 2.2.

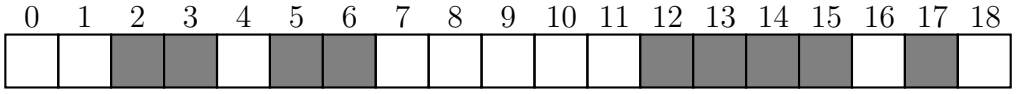

Figura 2.2: *Acumulación primaria.*

Las casillas obscuras representan los espacios ocupados. La siguiente llave podrá ser insertada con el método lineal en cualquiera de las 10 casillas restantes desocupadas, pero éstas no tienen la misma probabilidad entre sí, ya que todas las casillas cuentan y se van recorriendo a la derecha hasta encontrar una entrada vacía; de hecho, k sería insertada en la posición 16 si $12 \leq h(k) \leq 16$, mientras que para 8 sería sólo si $h(k) = 8$. Entonces la posición 16 es cinco veces más probable que la posición 8.

Este fenómeno hace que las listas de los elementos ocupados crezcan mucho más rápido, rompiendo con la uniformidad de la función hash, lo cual es una desventaja para el método lineal. A esto se le llama *acumulación primaria*.

Se puede probar que el número promedio de iteraciones en una búsqueda es

aproximadamente:

$$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right), \qquad \text{para una búsqueda exitosa.}$$
$$\frac{1}{2}\left(1+\left(\frac{1}{1-\alpha}\right)^2\right), \qquad \text{para una búsqueda fallida.}$$

Método de hashing doble

El hashing doble es uno de los mejores métodos que existen para direccionamiento abierto, porque las permutaciones que produce tienen muchas de las características de las permutaciones aleatorias. El hashing doble usa una función hash de la siguiente forma:

$$h(k,i) = (h_1(k) + ih_2(k)) \pmod{m} \tag{2.6}$$

donde h_1 y h_2 son dos funciones hash diferentes. La posición inicial es $T[h_1(k)]$ y las demás posiciones son desplazamientos de $h_2(k)$ de posiciones anteriores, módulo m. En este caso la secuencia de prueba depende de dos maneras diferentes de la llave k, ya que la posición inicial y el desplazamiento pueden variar independientemente.

Método uniforme

Este método se explica en términos del factor de carga $\alpha = \frac{n}{m}$ de la tabla hash, mientras el número de elementos n y el tamaño de la tabla m se acercan a ∞. Con direccionamiento abierto tenemos que $\alpha \leq 1$.

Si suponemos el hashing uniforme, entonces queremos decir que la sucesión de prueba $\langle h'(k,0), h'(k,1), \ldots, h'(k,m-1)\rangle$ para alguna k es igualmente probable de ser alguna permutación de $\langle 0, 1, \ldots, m-1\rangle$, asegurando la uniformidad de la función hash.

En un análisis más detenido, el número esperado de comparaciones para una llave, bajo la suposición de hashing uniforme, está dado por el siguiente teorema.

Teorema 2.3.1. *En una tabla hash donde las colisiones son resueltas con direccionamiento abierto y con factor de carga $\alpha \leq 1$, el número esperado de comparaciones en una búsqueda fallida para una llave k es a lo más*

$$\frac{1}{1-\alpha} \tag{2.7}$$

bajo la suposición de hashing uniforme.

Teorema 2.3.2. *En una tabla hash donde las colisiones son resultas con direccionamiento abierto y con factor de carga $\alpha \leq 1$, el número esperado de comparaciones en una búsqueda exitosa para una llave k es a lo más*

$$\frac{1}{a}\ln\left(\frac{1}{1-\alpha}\right)+\frac{1}{\alpha} \tag{2.8}$$

bajo la suposición de hashing uniforme.

Las demostraciones de estos dos teoremas las podemos encontrar en [Cormen] páginas 237-239.

De los teoremas anteriores se deduce que si α es constante, una búsqueda llevará a lo más $O(1)$ de tiempo. Por ejemplo, si la tabla está medio llena el promedio del número de intentos que se llevan a cabo en una búsqueda infructuosa es $\frac{1}{1-.5}=2$. Y si está un 90 % llena, el número promedio de intentos es $\frac{1}{1-.9}=10$.

2.4. Rehash

Algo que merece mencionarse en cualquier implementación de alguna tabla hash es la parte de rehash. ¿Qué es el rehash?, el rehash es simplemente el mecanismo que se usa cuando todas las casillas de la tabla hash están llenas y se requiere agregar una nueva pareja de llave y entrada.

Lo que se usa es crear una nueva tabla de más capacidad, esto implica que todos los valores de la vieja tabla tienen que ser actualizados con una nueva llave, ya que la función hash depende del número de elementos que puede contener la tabla.

Esto consume un tiempo de orden lineal, entonces desde un principio se puede especificar un posible máximo número de casillas si es que se conoce información acerca de las entradas antes de agregarlas. Además se recomienda dar un tamaño de proporcional de 1.5 sobre el tamaño de la vieja tabla para evitar que el rehash afecte el tiempo constante de inserción de elementos.

2.5. Ejemplos de funciones hash

A continuación veremos algunas de las funciones hash más utilizadas:

- Método de la división

  Una de las transformaciones más antiguas y sencillas consiste en dividir por el número de direcciones posibles. La función hash h se define como:

  $$h(k) = k \text{ mód } m. \tag{2.9}$$

- Pseudo-aleatorio

  El algoritmo de la división es usado en los generadores de números aleatorios y puede ser utilizado como función hash. Este generador tiene típicamente la forma:

  $$h(k) = (ak + c) \text{ mód } p. \tag{2.10}$$

  donde p, normalmente, es la potencia de 2 más cercana (por encima de n), n es el número de elementos de la tabla y $c = -1$. El hashing pseudo-aleatorio se usa con frecuencia como patrón de medida frente al efecto de azar de otros algoritmos de hashing.

- Método de la mitad del cuadrado

  Se calcula el cuadrado de la llave k. Entonces la función hash se define como:

  $$h(k) = L \tag{2.11}$$

  donde L se obtiene eliminando dígitos a ambos extremos de k^2 para todas las llaves.

- Método de plegado

  La llave k se divide en varias partes $k_1, k_2, \ldots, k_r$, donde cada parte, excepto posiblemente la última, tiene el mismo número de dígitos que la dirección requerida. Entonces, a cada una de las partes se le aplica el operador OR exclusivo para formar la dirección. Equivale a la suma binaria de bits si se ignora el acarreo,

  $$h(k) = k_1 \oplus k_2 \oplus \ldots \oplus k_r \tag{2.12}$$

  A veces a las partes pares $k_2, k_4, \ldots$ se las invierte antes de sumarlas, con el fin de conseguir valores mejor distribuidos y únicos.

  El problema del método de plegado de bits es cuando las llaves contienen los mismos grupos de bits pero en orden diferente. Estas llaves se dispersan con el mismo valor. Este método se combina frecuentemente con otros métodos de hash.

- Método de análisis de dígitos

  Sólo se seleccionan los dígitos que tienen la distribución más uniforme. Los que no son tan uniformes se borran de la llave, hasta que se obtiene el número deseado de dígitos. Este método implica un conocimiento de las llaves con anterioridad. Un cambio en el conjunto de llaves implica un nuevo análisis de

los dígitos, por lo que este método no se ajusta a las exigencias de rapidez de inserción y posibilidad de borrado.

Si las llaves no son enteros, se tienen que convertir a enteros antes de aplicar una de las funciones hash anteriores. Para el caso de cadena de carácteres, hay varias maneras de hacerlo:

- Se puede utilizar la representación interna con bits de cada carácter interpretada como número binario. Una desventaja de esto es que las representaciones con bits de las letras o dígitos tienden a ser muy similares entre sí en la mayoría de las computadoras.
- Otra función de dispersión es una función polinómica con base en la regla de Horner:
$$A_0 + X(A_1 + X(A_2 + \ldots + X(A_r)\ldots) \tag{2.13}$$
Esta función hash no es necesariamente la mejor con respecto a la distribución de la tabla, pero es muy simple y rápida de calcular. Si las llaves son muy largas el cálculo de la función hash será muy largo. En este caso, no se utilizan todos los carácteres. La longitud y las propiedades de las llaves influirán en los carácteres a elegir.

Una desventaja de todas estas funciones de dispersión es que no preservan el orden, es decir, no se cumple que

$$h(k_1) > h(k_2), \quad \text{siempre que,} \quad k_1 > k_2$$

Esto nos impide recorrer la tabla hash en orden secuencial con respecto a la llave. Sin embargo, las funciones que preservan el orden no son en general uniformes, es decir, no minimizan el número de colisiones.

2.6. Funciones hash perfectas

Dado un conjunto de llaves $k_1, k_2, \ldots, k_n$, una *función hash perfecta* es aquella función h, tal que $h(k_i) \neq h(k_j)$ para toda $i \neq j$. Es decir, no se producen colisiones.

En general es difícil encontrar una función hash perfecta para un conjunto particular de llaves. Además, una vez que se añaden unas cuantas llaves más al conjunto para el cual se había encontrado una función hash perfecta, ésta deja de ser perfecta en general para el nuevo conjunto. Así, aunque es deseable encontrar una función de

dispersión de este tipo para asegurar la recuperación inmediata, esto no es práctico a no ser que el conjunto de llaves sea estático y se busque en él con frecuencia. El ejemplo más claro de esto es un compilador. El conjunto de palabras reservadas del lenguaje de programación que se está compilando no cambia y se tiene que acceder muchas veces a este conjunto. Por lo tanto, una función de dispersión perfecta permitiría al compilador determinar rápidamente si una palabra es reservada o no.

Mientras más grande sea el conjunto de llaves más difícil será encontrar una función de dispersión perfecta.

En general es deseable tener una función de dispersión perfecta para un conjunto de n llaves en una tabla de sólo n posiciones. A este tipo de función de dispersión perfecta se la llama "mínima".

A continuación veremos algunos ejemplos de funciones de dispersión perfectas que dependen de un conjunto de llaves conocido.

- Funciones de dispersión perfectas por reducción al cociente. Son de la forma:

$$h(k) = \left\lfloor \frac{k+s}{d} \right\rfloor \tag{2.14}$$

  donde s y d son enteros.

- Funciones de dispersión perfecta por reducción de residuo. Son de la forma:

$$h(k) = \left\lfloor \frac{(k + s * r) \text{ mód } x}{d} \right\rfloor \tag{2.15}$$

  y un algoritmo para calcular los valores enteros r, s, x y d. Estas dos funciones fueron desarrolladas por Sprugnoli.

- Otro método para funciones hash perfectas nos lo ofrece Cichelli. Su fórmula es:

$$\begin{aligned} h(k) = {} & \text{longitud de } k + \\ & \text{valor asociado del primer carácter de } k + \\ & \text{valor asociado del último carácter de } k \end{aligned} \tag{2.16}$$

  donde k es la clave. Para un conjunto de llaves en particular, se deben calcular los valores asociados para los carácteres.

Funciones hash criptográficas

3.1. Ideas generales

Las *funciones hash criptográficas* son aquellas que encriptan una entrada y actúan de forma parecida a las funciones hash, ya que comprimen la entrada a una salida de longitud menor y son fáciles de calcular.

La razón por la cual nos interesan las funciones hash criptográficas es la de profundizar un poco más en el problema de las colisiones tanto matemática como conceptualmente.

El razonamiento es el siguiente, entre más difícil de invertir sea una función en algún punto arbitrario (con una imagen que tienda a ser uniforme en el contradominio) entonces más próxima será a asemejar una función aleatoria, con lo cual más difícil será predecir una evaluación en algún punto y por ende más difícil encontrar dos valores para los cuales la evaluación de la función coincida (colisión).

Dos preguntas son fundamentales para las funciones hash criptográficas: ¿qué significa formalmente que ocurra una colisión? y ¿cuál será la probabilidad de que ocurra una colisión en un hash ideal?, estas dos cuestiones las resolveremos más adelante.

Bajo ciertas hipótesis con respecto a la longitud de las cadenas de entrada, las funciones hash criptográficas se pueden utilizar como medios de autenticación bastante eficientes. El valor hash en estos casos recibe varios nombres: *imprint*, *digital fingerprint*, *message digest*.

Las funciones hash criptográficas o no invertibles (*one-way functions*) son aquellas que son difíciles de invertir, pero fáciles de calcular. Una función no invertible f es una función con las siguientes propiedades:

1. Calcular la función $f : \mathcal{K} \to \mathbb{Z}_n$ lleva tiempo polinomial, fácil de calcular.
2. Calcular una preimagen x para una $y = h(x)$ arbitraria lleva más que tiempo polinomial, difícil de invertir.

Hasta el momento no se ha encontrado una función que pueda ser demostrada como una función one-way, como tampoco se ha podido demostrar que no existen este tipo de funciones.

Fundamentalmente queremos pedir las siguientes propiedades a estas funciones: que dada una x sea muy fácil calcular $h(x)$, pero que, contrariamente, x sea difícil

de calcular dada una $y = h(x)$. Además pedimos que sea difícil encontrar una pareja (x, y) con $x \neq y$ tal que $h(x) = h(y)$, es decir, que sea difícil producir colisiones. Al decir difícil lo que queremos decir es que el tiempo que se necesita para encontrar una colisión consume por lo menos tiempo exponencial y por fácil nos referimos a que el tiempo que se necesita para calcular una imagen consume a lo más tiempo polinomial.

A continuación damos las siguientes definiciones que deberán cumplir las funciones hash criptográficas:

Definición 3.1.1. Resistencia a preimágenes (no invertibles): *es imposible en la práctica calcular una entrada que sea mapeada a una salida dada, es decir, encontrar una preimagen x tal que $h(x) = y$ dada una y.*

Definición 3.1.2. Segunda resistencia a preimágenes (resistencia débil a colisiones): *es computacionalmente difícil encontrar una segunda entrada $x' \neq x$ tal que $h(x') = h(x)$.*

Definición 3.1.3. Resistencia a colisiones (resistencia fuerte a colisiones): *es computacionalmente difícil encontrar dos diferentes entradas x, x' que sean mapeadas a la misma salida, es decir, que $h(x) = h(x')$.*

Nos damos cuenta que estas definiciones están relacionadas entre sí; por ejemplo, encontrar preimágenes diferentes dada una x equivale a la segunda definición y si en ésta a su vez hacemos variar la preimagen tendremos la tercera definición. Vamos a analizar un poco más detenidamente las definiciones anteriores, pero para esto necesitaremos introducir un modelo "ideal" de hash, el llamado modelo aleatorio de oráculo.

3.2. Modelo aleatorio de oráculo

El modelo aleatorio de oráculo fue introducido por primera vez por Bellare y Rogaway proveyéndonos de un modelo matemático de una función hash "ideal". En este modelo una función hash $h : \mathcal{X} \to \mathcal{Y}$ es tomada aleatoriamente del conjunto $\mathcal{F}^{\mathcal{X},\mathcal{Y}}$ de todas las funciones que tienen como dominio $\mathcal{X}$ y como contradominio $\mathcal{Y}$ y en donde estamos limitados (o afortunados) en preguntar a un oráculo los valores de h. Esto significa que no tenemos una fórmula o un algoritmo para calcular los valores de la función h; la única manera de calcular un valor $h(x)$ es preguntar a este oráculo su valor. Esto también puede ser pensado como si estuviéramos buscando el valor de $h(x)$ en un libro de números aleatorios muy grande, es decir, para cada x existe un

valor de $h(x)$ aleatorio. Este modelo simplifica el uso de la función hash, ya que la construye aleatoriamente y el cálculo de la misma es trivial.

Como consecuencia de este modelo se puede observar que aunque se sepan valores de $h(x_i)$ la probabilidad de que $h(x)$ sea igual a alguna y es independiente de los valores anteriores calculados; además $h(x)$ puede ser cualquier valor de y, por ser una función aleatoria. Entonces la probabilidad de que $h(x)$ sea igual a alguna y es igual a $\frac{1}{|\mathcal{Y}|}$; esto nos da la siguiente proposición.

Proposición 3.2.1. *Supóngase que f es tomada aleatoriamente del conjunto $\mathcal{F}^{\mathcal{X},\mathcal{Y}}$ y sea $\mathcal{X}_0 = \{x_1, x_2, \ldots, x_t\} \subseteq \mathcal{X}$. Supóngase que los valores $f(x_i) = y_i$ han sido determinados (preguntando al oráculo) para $1 \leq i \leq t$. Entonces la probabilidad $\forall x \in \mathcal{X} \backslash \mathcal{X}_0$*

$$P[f(x) = y \mid f(x_1) = y_1, \ldots, f(x_t) = y_t] = \frac{1}{|\mathcal{Y}|}.$$

¿Cuál será la probabilidad de éxito de los algoritmos de nuestras definiciones pasadas de encontrar preimágenes, encontrar segundas preimágenes y encontrar colisiones, con este modelo? A continuación probamos tres teoremas, con ayuda de la proposición anterior, que explican la complejidad de los algoritmos de las definiciones.

Teorema 3.2.1. *$\forall\, \mathcal{X}_0 \subseteq \mathcal{X}$ con $q = |\mathcal{X}_0|$ y $M = |\mathcal{Y}|$, la probabilidad del caso promedio de éxito para encontrar preimágenes en $\mathcal{X}_0$ es:*

$$\epsilon = 1 - \left(1 - \frac{1}{M}\right)^q. \tag{3.1}$$

Demostración. Sea $y \in \mathcal{Y}$ y $\mathcal{X}_0 = \{x_1, \ldots, x_q\}$. Para $1 \leq i \leq q$, sea E_i el evento $h(x_i) = y$. Se sigue de la proposición 3.2.1 que los E_i son eventos independientes y que la probabilidad $P[E_i] = \frac{1}{M} \quad \forall\, i, 1 \leq i \leq q$. Además encontrar una preimagen debería ser tan probable como que algún evento E_i fuera cierto; entonces se sigue que la probabilidad de encontrar una preimagen es:

$$\epsilon = P[E_1 \vee E_2 \vee \ldots \vee E_q] = 1 - \left(1 - \frac{1}{M}\right)^q.$$

□

Si tomamos a M muy grande tenemos que $1 - \frac{1}{M}$ se aproxima asintóticamente a

$e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} \cdots$, entonces tenemos que $1 - \left(1 - \frac{1}{M}\right)^q \simeq 1 - e^{-q/M}$, y luego:

$$\begin{aligned} \epsilon &\simeq 1 - e^{-q/M} \\ -q/M &\simeq \ln(1-\epsilon) \\ q &\simeq M \ln\left(\frac{1}{1-\epsilon}\right) \end{aligned}$$

y para una probabilidad de $\epsilon = 1/2$ tenemos que el número de elementos a probar para encontrar una preimagen es justamente $|\mathcal{Y}|$, lo cual es bastante aceptable ya que en general $\mathcal{Y}$ se hace de un tamaño relativamente pequeño.

El siguiente teorema encuentra segundas preimágenes:

Teorema 3.2.2. *$\forall\ \mathcal{X}_0 \subseteq \mathcal{X}$ con $q = |\mathcal{X}_0|$ y $M = |\mathcal{Y}|$, la probabilidad del caso promedio de éxito para encontrar una segunda preimagen en $\mathcal{X}_0$ es:*

$$\epsilon = 1 - \left(1 - \frac{1}{M}\right)^{q-1}. \tag{3.2}$$

Demostración. El algoritmo de segunda preimagen es muy parecido al de preimagen, cambia solamente en que hay que guardar el valor de $h(x)$ para el valor de entrada x. □

Como en el análisis del teorema anterior, q es proporcional a $|\mathcal{Y}|$ cuando la probabilidad es de un medio.

Y finalmente para encontrar colisiones tenemos el siguiente teorema:

Teorema 3.2.3. *$\forall\ \mathcal{X}_0 \subseteq \mathcal{X}$ con $q = |\mathcal{X}_0|$ y $M = |\mathcal{Y}|$, la probabilidad del caso promedio de éxito para encontrar una colisión en $\mathcal{X}_0$ es:*

$$\epsilon = 1 - \left(\frac{M-1}{M}\right)\left(\frac{M-2}{M}\right)\cdots\left(\frac{M-q+1}{M}\right). \tag{3.3}$$

Demostración. Sea $\mathcal{X}_0 = \{x_1, \ldots, x_n\}$. Y para cada $1 \leq i \leq q$, sea E_i el evento $h(x_i) \notin \{x_1, \ldots x_{i-1}\}$. Usando inducción se sigue de la proposición 4.1 que la probabilidad de $E_1 = 1$ y

$$P[E_i \mid E_1 \wedge E_2 \wedge \ldots \wedge E_{i-1}] = \left(\frac{M-q+1}{M}\right), \qquad \text{para } 2 \leq i \leq q.$$

Entonces tenemos que:

$$P[E_1 \wedge E_2 \wedge \ldots \wedge E_{i-1}] = \left(\frac{M-1}{M}\right)\left(\frac{M-2}{M}\right)\cdots\left(\frac{M-q+1}{M}\right).$$

□

El teorema anterior nos dice que la probabilidad de no encontrar colisiones es:

$$\left(1-\frac{1}{M}\right)\left(1-\frac{2}{M}\right)\cdots\left(1-\frac{q-1}{M}\right)=\prod_{i=1}^{q-1}\left(1-\frac{i}{M}\right).$$

Usando el argumento de $1-x\simeq e^{-x}$ para x grande tenemos que:

$$\begin{aligned}\prod_{i=1}^{q-1}\left(1-\frac{i}{M}\right) &\simeq \prod_{i=1}^{q-1} e^{-i/M}\\ &\simeq e^{\sum_{i=1}^{q-1} i/M}\\ &\simeq e^{-q(q-1)/2M}.\end{aligned}$$

De esta manera podemos estimar la probabilidad de encontrar por lo menos una colisión:

$$\begin{aligned}\epsilon &\simeq e^{\frac{-q(q-1)}{2M}}\\ \ln(1-\epsilon) &\simeq \frac{-q(q-1)}{2M}\\ q &\simeq \sqrt{2M\ln\left(\frac{1}{1-\epsilon}\right)}\end{aligned}$$

para $\epsilon = 0.5$ tenemos que

$$q \simeq 1.17\sqrt{M},$$

esto significa que con solo $\sqrt{M}$ elementos aleatorios de $\mathcal{X}$ se produce una colisión con $\frac{1}{2}$ de probabilidad; a este resultado también se le conoce como la "paradoja del cumpleaños"; si tomamos a $M = 365$ como los días de un año, la probabilidad de que dos personas celebren su cumpleaños el mismo día del año será de $\frac{1}{2}$ cuando $q = 22.3$, es decir, tomando sólo 23 personas al azar.

La paradoja del cumpleaños nos impone una cota inferior en la longitud de la salida de nuestra función hash. Por ejemplo una firma digital de 40-bits sería bastante insegura, ya que una colisión puede ser encontrada con probabilidad de $\frac{1}{2}$ con solo 2^{20} valores hash aleatorios. En la práctica se suele sugerir una longitud de 128-bits, ya que 2^{64} es una cota difícil de alcanzar. De hecho 160-bits de longitud es usualmente recomendada.

3.3. Clasificación

Las funciones hash criptográficas pueden ser divididas en dos clases: funciones hash sin llave, las cuales en su especificación dictan un solo parámetro de entrada (el mensaje); y las funciones hash con llave las cuales en su especificación dictan dos diferentes entradas, el mensaje y una llave secreta.

Además podemos tener otra clasificación basada en ciertas propiedades que proveen y repercuten en aplicaciones específicas.

1. *Códigos de detección de modificaciones (MDC)*

   También conocidos como "códigos de detección de manipulaciones", el propósito de un MDC es (informalmente) proveer una imagen representativa o hash de un mensaje, satisfaciendo las siguientes propiedades:

   - *funciones hash no invertibles (OWHF)*: para éstas, encontrar una entrada que sea mapeada a una salida dada es difícil, es decir, cumplen con las propiedades de resistencia a preimágenes y segunda resistencia a preimágenes.
   - *funciones hash resistentes a colisiones (CRHF)*: para éstas, encontrar dos entradas que tengan el mismo valor hash es difícil, es decir, cumplen con la propiedad de resistencia a colisiones.

   La meta final es facilitar la integridad de la información requerida por aplicaciones específicas. MDC son una subclase de las funciones hash sin llave.

2. *Códigos de autenticación de mensajes (MAC)*

   Una MAC o código de autenticación de mensajes es una familia de funciones h_k parametrizadas por una llave secreta k, donde las funciones cumplen con las siguientes características:

   a) Fácil de calcular.- para una función h_k y dado un valor k y una entrada x, $h_k(x)$ es fácil de calcular.

   b) Compresión.- h_k mapea una entrada x de tamaño arbitrario a una salida $h_k(x)$ de longitud fija n.

   c) Resistencia a calcularse.- dados cero o más pares MAC $(x_i, h_k(x_i))$, es computacionalmente difícil calcular una nueva pareja $(x, h_k(x))$ tal que $h_k(x) = h_k(x_i)$ para alguna x diferente de x_i.

> El propósito de una MAC es, informalmente, el de aumentar, sin el uso de mecanismos adicionales, la confianza respecto a la fuente del mensaje y a su integridad. Las MAC tienen dos distintos parámetros, el mensaje de entrada y una llave secreta incluida como parte del mensaje de entrada. El tamaño de la llave determinará la seguridad del algoritmo.

Generalmente se supone que la especificación del algoritmo de la función hash es de dominio público. De esta manera en el caso de los MDC, dado un mensaje como entrada cualquiera puede calcular el resultado del hash; y en el caso de las MAC, dado un mensaje de entrada, cualquiera con el conocimiento de la llave puede calcular el resultado del hash.

3.4. Construcciones generales

Qué tan sencillo será construir una buena función hash criptográfica, no lo sabemos con seguridad, pero la experiencia nos dicta que en general es difícil.

La idea más simple es usar algo ya hecho y esto es precisamente lo que se hace en la práctica: se usan cifrados de bloques como pieza primaria. Intentemos usar un cifrado de bloque ya hecho, como DES. DES es una función de 64-bits a 64-bits con llave de 56-bits. Por desgracia esta función no comprime y nuestras entradas suelen tener entre 64 y 128 bits. ¿Qué es lo que hacemos con nuestra entrada más grande? Tomamos los primeros 8 bytes, los encriptamos y usamos el resultado como llave para encriptar los siguientes 8 bytes y repetimos hasta terminar con nuestra entrada; la primera llave la podemos tomar como la cadena "10101010" (nuestra implementación de DES toma 64-bits como tamaño de la llave).

Al final, a nuestra salida $E_k(x)$ de 64-bytes la transformamos a un entero como si fuera su representación en binario. Y tenemos una función hash implementada a base de DES, para usarse en una tabla hash; al entero resultante se le reduce módulo el tamaño de la tabla. Pero en la práctica, ¿qué pasa? Estadísticamente con llaves tomadas de un diccionario al azar, las colisiones en una tabla hash implementada con encadenamiento y usando una función hash construida con DES, se pueden ver en la figura 3.1.

Tenemos que la máxima colisión fue de 5 elementos en una tabla que contiene 500 elementos, una muy buena distribución (el 1 % de elementos). Ahora, experimentaremos con un conjunto de llaves muy similares entre sí para ver qué sucede. En realidad las llaves son la cadena "Llave" concatenada a un número consecutivo concatenado con "Llave " otra vez; las llaves están en el intervalo "Llave000Llave"-"Llave500Llave". La gráfica de colisiones se muestra en la figura 3.2.

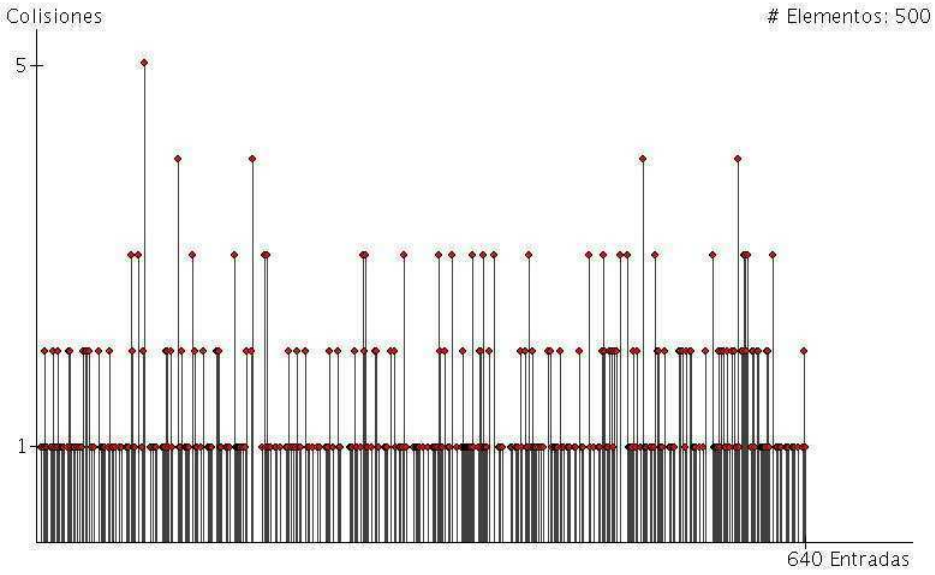

Figura 3.1: *Colisiones usando DES.*

Ahora nuestra máxima colisión fue de 4, lo que hace de nuestra función hash una función hash criptográfica buena. Veamos qué sucede con el mismo conjunto de llaves pero con el método de la división para darnos una idea comparativa. La gráfica se muestra en la figura 3.3.

La diferencia es radical: la máxima colisión es de 100 elementos y no existe la uniformidad que obtuvimos con DES. DES como función hash funciona excelente; por ejemplo, para 10,000 llaves consecutivas tiene una máxima colisión de 6. Sin embargo DES es algo lento para calcularse comparado con otros algoritmos de hashing que veremos más adelante.

A continuación veremos un modelo más general en donde iteramos varias veces el bloque de cifrado y combinamos los bloques encriptados con la entrada original. En la siguiente sección damos un método general para construir una función hash criptográfica iterada.

3.4.1. Modelo general para una función hash iterada

La mayoría de las funciones hash sin llave secreta se diseñan como procesos iterativos, de manera que mapean entradas de longitud arbitraria procesando sucesivamente bloques de n-bits. El esquema general de la función se puede ver en la figura 3.4 y más a detalle mostramos su funcionamiento en la figura 3.5.

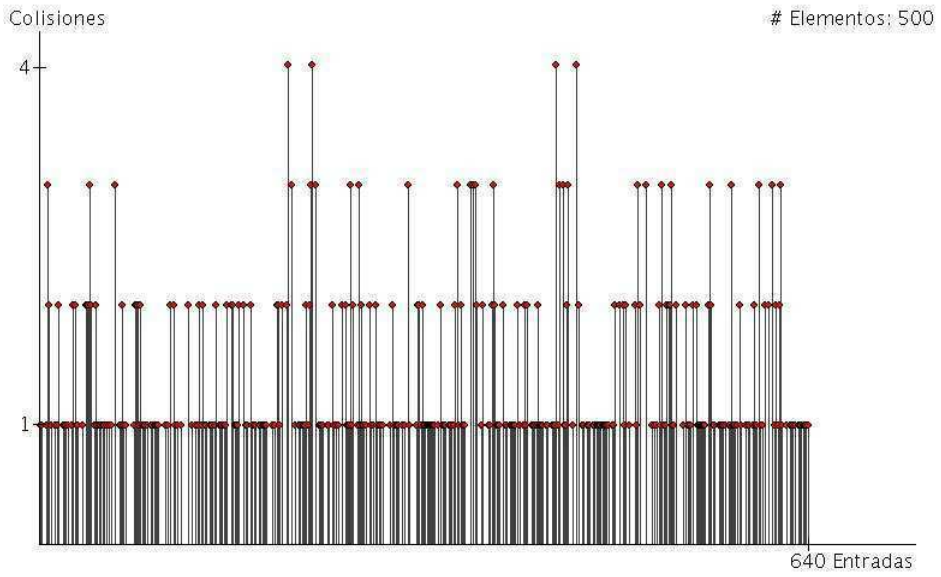

Figura 3.2: *Colisiones usando un conjunto de llaves semejantes.*

Siguiendo el esquema tenemos que primero la entrada x se divide en bloques de r-bits, $x_1, x_2, \ldots, x_t$. Además se extiende con bits cero (*padding*) de tal manera que la longitud total sea múltiplo de r y frecuentemente se agrega un bloque con la longitud original de la entrada. Después cada bloque sirve como entrada a una función hash interna con entrada de longitud de r-bits, que calcula un resultado intermedio de tamaño de n-bits, para una n fija, y lo hace combinando el valor hash intermedio de n bits anterior y el siguiente bloque de entrada x_i; esta función es la función de compresión. En otras palabras, lo que hace la función h es procesar los bloques H_i que denotan el resultado intermedio en el paso i de la iteración. El proceso general para una función iterada con entrada $x = x_1 x_2 \ldots x_t$ se modela de la siguiente manera, con H_i como una variable de encadenamiento entre el paso $i-1$ y el paso i y H_0 una constante predefinida o valor de inicialización IV:

$$H_0 = IV; \qquad H_i = f(H_{i-1}, x_i),\ 1 \leq i \leq t; \qquad h(x) = g(H_t).$$

Se usa una transformación opcional g como paso final para mapear la variable de encadenamiento H_t de n-bits a un resultado de m-bits $g(H_t)$; muchas veces g es la identidad.

Las funciones hash iteradas se distinguen entre sí por sus métodos de preprocesamiento, función de compresión y transformación final.

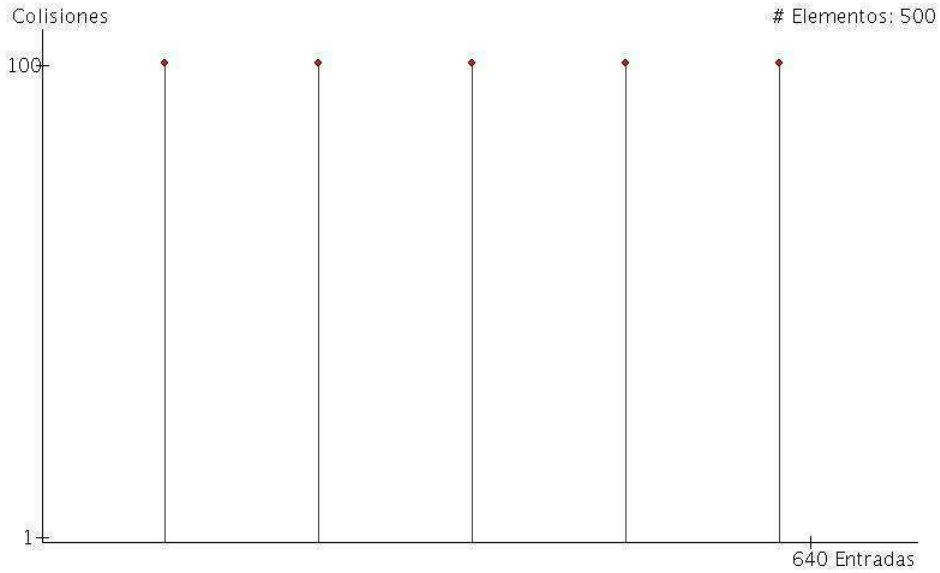

Figura 3.3: *Colisiones usando el método de la división.*

3.5. MDC

Desde un punto de vista estructural los códigos de detección de modificaciones o MDC se pueden clasificar por la naturaleza de sus funciones de compresión. Desde este punto de vista existen tres categorías de funciones hash iteradas: basadas en cifrados de bloques, funciones hash hechas a la medida y funciones hash basadas en aritmética modular. Las funciones hash hechas a la medida están específicamente diseñadas para ser muy veloces.

3.5.1. Funciones hash basadas en cifrados de bloques

Una motivación práctica para construir funciones hash a partir de cifrados de bloques es que si existe una implementación eficiente de un cifrado de bloques dentro de un sistema, entonces usándolo como el componente central de la función hash puede proveer la funcionalidad anterior a un bajo costo. La esperanza es que un buen cifrado de bloques pueda servir como un bloque de construcción para la creación de una función hash con propiedades adecuadas para múltiples aplicaciones.

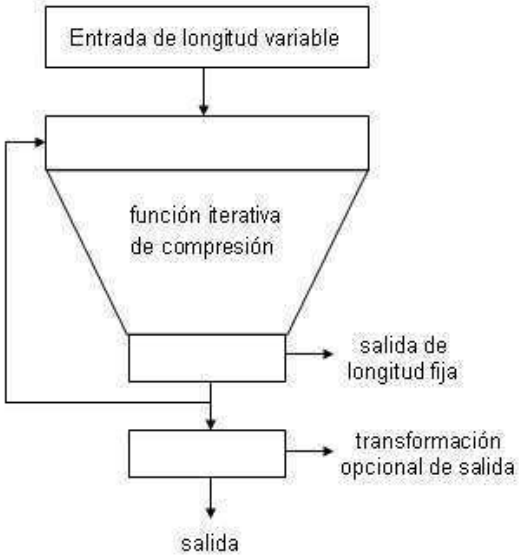

Figura 3.4: *Una función hash iterada.*

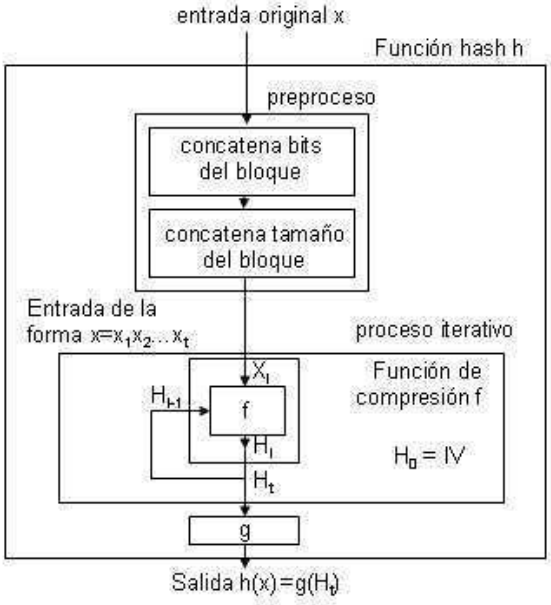

Figura 3.5: *Funcionamiento de una función hash iterada.*

MDC de longitud simple

Los siguientes tres esquemas están íntimamente relacionados. Éstos usan los siguientes componentes predefinidos:

1. Un cifrado de bloques E_K con entrada de n-bits parametrizado por una llave simétrica K;
2. una función g que mapea entradas de n-bits a llaves K apropiadas para E; y
3. un valor inicial fijo IV, apropiado para usarse con E.

Algoritmo 3.5.1.1. *Matyas-Meyer-Oseas*
Entrada: una cadena x.
Salida: un código hash de n-bits.

1. *La entrada x se divide en bloques de tamaño de n-bits y se rellena de ceros, si es necesario, para completar el último bloque. Se denota la entrada rellenada consistiendo de t bloques de n-bits como: $x_1x_2 \ldots x_t$. Se debe especificar una constante inicial de n-bits IV.*
2. *La salida H_t está definida por: $H_0 = IV$; $H_i = E_{g(H_{i-1})}(x_i) \oplus x_i$, $1 \leq i \leq t$.*

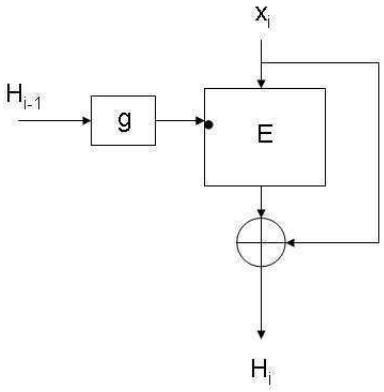

Figura 3.6: *Algoritmo de Matyas-Meyer-Oseas.*

Algoritmo 3.5.1.2. *Davies-Meyer*
Entrada: una cadena x.
Salida: un código hash de n-bits.

1. *La entrada x se divide en bloques de tamaño de k-bits donde k es el tamaño de la llave y se rellena de ceros, si es necesario, para completar el último bloque. Se denota la entrada rellenada que consiste de t bloques de k-bits como: $x_1 x_2 \dots x_t$. Se debe especificar una constante inicial de n-bits IV.*

2. *La salida H_t está definida por: $H_0 = IV$; $H_i = E_{x_i}(H_{i-1}) \oplus H_{i-1}$, $1 \leq i \leq t$.*

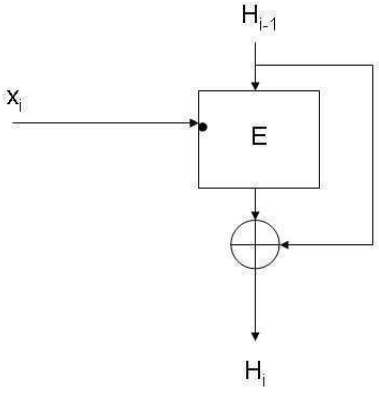

Figura 3.7: *Algoritmo de Davies-Meyer.*

Algoritmo 3.5.1.3. *Miyaguchi-Preneel*

Entrada: una cadena x.

Salida: un código hash de n-bits.

El algoritmo es idéntico al algoritmo 3.5.1.1. excepto que a la salida H_{i-1} del paso anterior se le aplica un XOR con el paso actual de la iteración. Más precisamente H_i está definida como: $H_0 = IV$; $H_i = E_{g(H_{i-1})}(x_i) \oplus x_i \oplus H_{i-1}$, $1 \leq i \leq t$.

MDC de longitud doble: MDC-2 y MDC-4

Los algoritmos de MDC-2 y MDC-4 son MDC que requieren 2 y 4 operaciones de cifrado de bloques por bloque respectivamente. Estos algoritmos usan una combinación de 2 o 4 iteraciones del algoritmo de Matyas-Meyer-Oseas para producir un hash de longitud doble. Cuando se usan siguiendo la especificación original, utilizan DES como el bloque de cifrado; éstos producen códigos hash de 128-bits de longitud. La construcción general puede usar otros bloques de cifrados. MDC-2 y MDC-4 usan los siguientes componentes preespecificados:

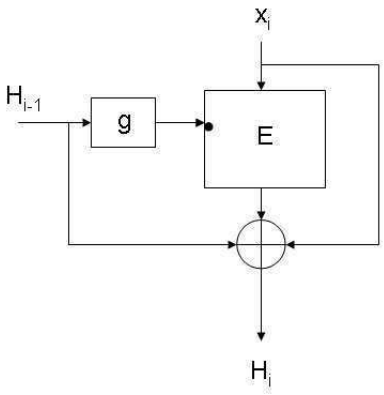

Figura 3.8: *Algoritmo de Miyaguchi-Preneel.*

1. DES como bloque de cifrado E_K de entrada $n = 64$-bits parametrizado con una llave de 56-bits.

2. Dos funciones g y $\overline{g}$ que mapean entradas U de 64-bits a llaves de 56-bits de la siguiente manera. Para $U = u_1u_2 \dots u_{64}$, se borra cada octavo bit empezando con u_8 y se reemplaza el segundo y tercer bit por '10' para g, y '01' para $\overline{g}$:

$$g(U) = u_1 10 u_4u_5u_6u_7u_9u_{10} \dots u_{63} \qquad \text{y}$$
$$\overline{g(U)} = u_1 01 u_4u_5u_6u_7u_9u_{10} \dots u_{63}.$$

Algoritmo 3.5.1.4. *MDC-2*
Entrada: una cadena x de tamaño $r = 64t$ para $t \geq 2$.
Salida: un código hash de 128-bits.

1. *La entrada x se divide en bloques de longitud de 64-bits: $x = x_1x_2 \dots x_t$.*

2. *Se escogen dos constantes de 64-bits: IV y $\overline{IV}$ de un conjunto de valores recomendados. Dos valores recomendados son (en hexadecimal):*

$$IV = 0x5252525252525252 \qquad y \qquad \overline{IV} = 0x2525252525252525.$$

3. *Sea $||$ la concatenación y C_i^L, C_i^R la parte izquierda y derecha de 32-bits de C_i. La salida $h(x) = H_t||\overline{H}_t$ está definida por (para $1 \leq i \leq t$):*

$$H_0 = IV; \qquad k_i = g(H_{i-1}); \qquad C_i = E_{k_i}(x_i) \oplus x_i; \qquad H_i = C_i^L||\overline{C}_i^R \qquad y$$
$$\overline{H_0} = \overline{IV}; \qquad \overline{k_i} = \overline{g(H_{i-1})}; \qquad \overline{C_i} = E_{\overline{k_i}}(x_i) \oplus x_i; \qquad \overline{H_i} = \overline{C}_i^L||C_i^R.$$

Algoritmo 3.5.1.5. *MDC-4*

Entrada: una cadena x de tamaño $r = 64t$ para $t \geq 2$.
Salida: un código hash de 128*-bits.*

1. *Como en el paso 1 de MDC-2.*
2. *Como en el paso 2 de MDC-2.*
3. *Con la notación de MDC-2, la salida es $h(x) = G_t||\overline{G_t}$ definida de la siguiente manera (para $1 \leq i \leq t$):*

$$
\begin{array}{llll}
G_0 = IV; & \overline{G_0} = \overline{IV}, & & \\
k_i = g(G_{i-1}); & C_i = E_{k_i}(x_i) \oplus x_i; & H_i = C_i^L||\overline{C}_i^R, & \\
\overline{k_i} = \overline{g}(\overline{G_{i-1}}); & \overline{C_i} = E_{\overline{k_i}}(x_i) \oplus x_i; & \overline{H_i} = \overline{C}_i^L||C_i^R, & \\
j_i = g(H_i); & D_i = E_{j_i}(\overline{G_{i-1}}) \oplus \overline{G_{i-1}}; & G_i = D_i^L||\overline{D}_i^R, & \\
\overline{j_i} = \overline{g}(\overline{H_i}); & \overline{D_i} = E_{\overline{j_i}}(G_{i-1}) \oplus G_{i-1}; & \overline{G_i} = \overline{D}_i^L||D_i^R. &
\end{array}
$$

3.5.2. Funciones hash hechas a la medida

Estas funciones hash son especialmente diseñadas desde cero para el propósito de hacer hashing, con una ejecución optimizada en mente, y sin usar componentes existentes como cifrados de bloques o aritmética modular. Las funciones hash que reciben mayor atención son aquellas basadas en la función hash MD4. MD4 fue diseñada específicamente para implementaciones de software en máquinas de 32-bits. Motivados por razones de seguridad se diseñó MD5 casi inmediatamente, como una variación de MD4. Otra importante variante es el algoritmo de SHA-1, el más reciente algoritmo basado en MD4.

MD4

MD4 es una función hash con salida de 128-bits. Las metas del diseño original de MD4 eran que romperlo requiriera estrictamente un esfuerzo de fuerza bruta, es decir, encontrar dos colisiones con el mismo valor hash debería tomar cerca de 2^{64} operaciones y encontrar una entrada que devolviera el mismo valor hash para una entrada fija debería tomar aproximadamente 2^{128} operaciones. Ahora se sabe que MD4 falla en estas metas. Se han encontrado colisiones para MD4 en 2^{20} operaciones en su función de compresión. Sin embargo incluiremos el algoritmo de MD4 como referencia histórica y criptoanalítica; además sirve como una referencia conveniente para describir otras funciones hash dentro de esta familia.

Algoritmo 3.5.2.1. *MD4*
Entrada: una cadena x de tamaño arbitrario.
Salida: un código hash de 128-bits.

1. *Se definen cuatro constantes de 32-bits (IVs):*

$$h_1 = \texttt{0x67452301}, \qquad h_3 = \texttt{0x98badcfe},$$
$$h_2 = \texttt{0xefcdab89}, \qquad h_4 = \texttt{0x10325476}.$$

2. *Se definen las siguientes constantes de 32-bits:*
   - $y[j] = 0, 0 \leq j \leq 15$*;*
   - $y[j] =$ `0x5a827999`*,* $16 \leq j \leq 31$ *(raíz cuadrada de 2);*
   - $y[j] =$ `0x6ed9eba1`*,* $32 \leq j \leq 47$ *(raíz cuadrada de 3).*

3. *Se definen los órdenes para acceder a las palabras de entrada:*
   - $z[0 \ldots 15] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]$*;*
   - $z[16 \ldots 31] = [0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15]$*;*
   - $z[32 \ldots 47] = [0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15]$.

4. *Finalmente se definen el número de bits en las rotaciones a la izquierda:*
   - $s[0 \ldots 15] = [3, 7, 11, 19, 3, 7, 11, 19, 3, 7, 11, 19, 3, 7, 11, 19]$*;*
   - $s[16 \ldots 31] = [3, 5, 9, 13, 3, 5, 9, 13, 3, 5, 9, 13, 3, 5, 9, 13]$*;*
   - $s[32 \ldots 47] = [3, 9, 11, 15, 3, 9, 11, 15, 3, 9, 11, 15, 3, 9, 11, 15]$.

5. *Se extiende x de tal manera que su longitud en bits sea un múltiplo de 512, como sigue: se agrega un bit 1, después se agregan $r - 1$ bits cero para la r más pequeña tal que la cadena resultante sea un múltiplo de 512. Finalmente se agrega la representación de 64-bits de la longitud de x como dos palabras de 32-bits, con la palabra menos significativa primero. Sea m el número de bloques de 512-bits en la cadena resultante. La entrada extendida consiste de $16m$ palabras de 32-bits: $x_0 x_1 \ldots x_{16m-1}$.*

   Se inicializa: $(H_1, H_2, H_3, H_4) \leftarrow (h_1, h_2, h_3, h_4)$.

6. *Para cada i de 0 a $m - 1$, se copia el i-ésimo bloque de 16 palabras de 32-bits a las variables temporales $X[j] \leftarrow x_{16i+j}, 0 \leq j \leq 15$; entonces se procesan en 3 rondas de 16 pasos cada una:*

- *Para j de 0 a 15:*
  $t \leftarrow (A + f(B,C,D) + X[z[j] + y[j]])$.
  $(A,B,C,D) \leftarrow (D, t \hookleftarrow s[j], B, C)$.
- *Para j de 16 a 31:*
  $t \leftarrow (A + g(B,C,D) + X[z[j] + y[j]])$.
  $(A,B,C,D) \leftarrow (D, t \hookleftarrow s[j], B, C)$.
- *Para j de 32 a 47:*
  $t \leftarrow (A + h(B,C,D) + X[z[j] + y[j]])$.
  $(A,B,C,D) \leftarrow (D, t \hookleftarrow s[j], B, C)$.
- *(H_1, H_2, H_3, H_4)* $\leftarrow$ *($H_1 + A$, $H_2 + B$, $H_3 + C$, $H_4 + D$).*

donde f, g, h están definidas por: $f(u,v,w) = u\&v \vee \overline{u}\&w$, $g(u,v,w) = u\&v \vee u\&w \vee v\&w$, $h(u,v,w) = u \oplus v \oplus w$.

7. *El valor del hash es la concatenación: $H_1||H_2||H_3||H_4$.*

3.6. Aproximación a una función hash criptográfica

Intentaremos construir una función hash a partir de los algoritmos basados en MD4. Nuestra primera idea sería la de dispersar los valores de los bits por medio de operaciones booleanas a través de todos los demás bits, es decir, que cada bit de entrada afecte a todos los bits de salida. Una manera de hacerlo es dividir la entrada en bloques del mismo tamaño y a los n bloques aplicarles alguna función booleana que reduzca al mínimo el número de colisiones; el problema de esto es que podemos tener un número reducido de bloques y una cantidad muy grande de funciones booleanas para escoger la óptima; por ejemplo, para 16 bloques de 32 bits el número posible de funciones booleanas considerando las operaciones de negación, AND, OR, XOR y la suma sería de $2^{16} * 4^{15}$, un número muy grande de posibilidades a probar. Una posible solución sería la de dividir el número de bloques a diferentes salidas y después unir las salidas a una sola. Por ejemplo, podemos usar 5 bloques de entrada para 16 salidas.

$$S_i = B_{i-2} \oplus B_{i-1} \oplus B_i \oplus B_{i+1} \oplus B_{i+2}.$$

y unir las salidas de la siguiente forma:

$$
\begin{aligned}
a_1 &= S_1 + S_2 + S_3;\\
a_2 &= S_4 + S_5 + S_6;\\
a_3 &= S_7 + S_8 + S_9;\\
a_4 &= S_{10} + S_{11} + S_{12};\\
a_5 &= S_{13} + S_{14} + S_{15} + S_{16}.
\end{aligned}
$$

de tal manera que la concatenación de las a_i sea una salida de 160 bits. Ahora bien, ¿qué función booleana será la óptima a utilizar? Lo que podemos hacer es probar estadísticamente todas las posibles combinaciones de operaciones y utilizar la que tenga la más baja colisión máxima. En nuestra simulación la función booleana con más baja colisión resultó ser la siguiente:

$$S_i = \neg B_{i-2} + B_{i-1} + B_i \oplus B_{i+1} \oplus \neg B_{i+2} + \neg A_i.$$

en donde $\oplus$ simboliza al operador XOR y las A_i son valores aleatorios para dispersar mejor los bloques de entrada. Sin embargo esto no es suficiente para dispersar los bits de entrada; lo que tenemos que hacer es iterar la fórmula de tal manera que con pocas iteraciones los bits de entrada afecten a todos los bits de salida, tal como lo hace la iteración que se muestra en la tabla 3.1.

```
for( i=0; i<7; i++ ){
  S[i] = ~A[i] + ~B[i-2] + B[i-1] + B[i] ^ B[i+1]
         ^ ~B[i+2];
  B = S;
}
```

Tabla 3.1: *Código que muestra un ciclo para dispersar las entradas de una función booleana.*

De esta manera los bits de entrada son dispersados a todos los demás bits de salida. Cabe destacar que en esta construcción no hemos hecho ninguna rotación o *shift*, algo que los algoritmos anteriores hacían. Usando la construcción anterior y utilizando como valores de entrada cadenas que difieren en un bit, las colisiones que devuelve la función hash se pueden apreciar en la figura 3.9.

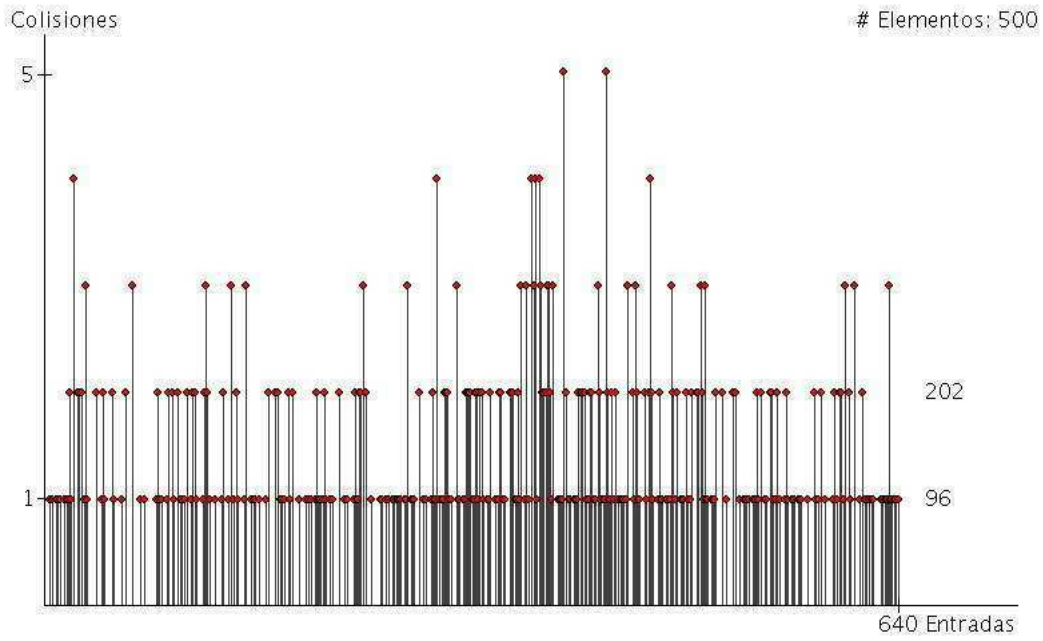

Figura 3.9: *Colisiones en una función hash construida a la medida.*

Conclusiones

A través de este trabajo hemos querido mostrar los beneficios que se obtienen de utilizar algoritmos de búsqueda usando funciones de hash; las búsquedas que se llevan a cabo por medio de estas funciones son realmente eficaces. Estamos hablando de búsquedas que en promedio consumen un tiempo a lo más constante, y esto quiere decir que el número de elementos que existen en nuestra estructura de datos no influye en el tiempo de las búsquedas, una ventaja realmente asombrosa comparada con otros métodos de búsqueda.

¿Qué obstáculos encontramos al implementar este algoritmo? En primer lugar la inserción de muchos elementos puede ser un problema ya que la longitud de la tabla se utiliza al final del cálculo del hash para poder obtener la posición de cada elemento, entonces para poder aumentar el tamaño de la tabla se necesita hacer un rehash a todos los elementos de la tabla. La situación es la siguiente, la tabla tiene un número fijo de casillas vacías para agregar elementos, entonces cuando se han ocupado todas las casillas (o alcanzado un cierto porcentaje del número de casillas) se tiene que aumentar el número de éstas; pero como el resultado de la función hash depende del número de casillas de la tabla, hay que asignar nuevos valores de hash a todos los elementos de tal manera que se dispersen en las nuevas casillas; esta nueva asignación es el rehash y consume un cierto tiempo. Cuando la tabla está llena, la inserción consume αn de tiempo, donde α es el tiempo que lleva calcular la función hash. De la misma manera el rehash también sucede en la eliminación de muchos elementos.

Otro problema fue el de encontrar un conjunto de entrada que nos permitiera obtener el peor caso de una tabla hash, equivalente a realizar búsquedas en una lista; sin embargo con la ayuda de los algoritmos de encripción este conjunto resulta muy difícil de construir, lo cual nos asegura una buena dispersión de los elementos en la tabla.

El problema de encontrar colisiones está íntimamente relacionado con la propiedad que tiene una función de no ser invertible (*one-way functions*), lo cual nos llevó al estudio de las funciones hash basadas en algoritmos criptográficos; y es que pareciera ser que mientras mejor se cifra nuestra entrada mejor se dispersa dentro de un conjunto de entradas similares entre sí. Y aunque por el momento no existen en la literatura construcciones válidas de funciones no invertibles, es fácil probar que si $P = NP$, entonces cualquier función entera que tenga asociada una máquina de Turing para calcular $f(x)$ puede ser invertida con una máquina de Turing no deterministica, lo cual implicaría que la clase de funciones no invertibles es vacía. Sin

embargo carecemos de una prueba formal para la implicación de regreso: si la clase de funciones no invertibles no es vacía entonces $P \neq NP$.

Aunque formalmente no podemos probar que determinada función hash disperse efectivamente todos los posibles conjuntos de entrada, podemos probar estadísticamente diferentes conjuntos de entrada y observar la máxima colisión para darnos una idea de que tan bien se dispersan los valores hash; esto nos ayuda, en la práctica, a elegir una buena construcción de función hash.

¿De qué forma podemos construir una buena función hash? Existen varias maneras, en particular, las funciones iterativas son las que mejor se comportan en la práctica; su mejor ejemplo son las basadas en la familia de algoritmos MD4, que aunque originalmente fueron creadas para autenticación de datos pueden ser perfectamente adaptadas para fungir como la caja negra de una función hash. En el capitulo 3 se realizo una construcción aproximada de función hash criptográfica, simplificando MD4 obtuvimos un algoritmo veloz con un porcentaje aceptable de colisiones con diferentes conjuntos de entrada.

Quedaría por demostrar la existencia de algoritmos perfectos, es decir, funciones hash perfectas para diferentes conjuntos de entrada o cuando menos cotas en las máximas colisiones. Esto implicaría saber formalmente qué tan distintos son los valores que regresa determinada función hash. La dificultad para lograr esto radica en que existe una infinidad de posibles funciones hash y una infinidad de posibles conjuntos de entrada a utilizar.

De todos modos la imposibilidad de poder probar la uniformidad de una función hash no nos impide utilizar una basándonos en su desempeño experimental. Con esto en mente podemos con cierta seguridad construir tablas hash que cumplan nuestras expectativas: en promedio, una velocidad de búsqueda de tiempo constante.

Al recorrer y estudiar las diferentes partes del tema de funciones de hash nos sentimos en la necesidad de recomendar su utilización en los casos aplicables por lo fácil que resulta su implementación en software. Los dos apéndices incluidos contienen el código fuente de la implementación que se utilizo para realizar las pruebas experimentales.

Finalmente, sabemos que la velocidad de cómputo y la capacidad de almacenamiento aumentan conforme pasa el tiempo, también sabemos que cada vez se manejan mayores volúmenes de información y mayores niveles de confidencialidad; ante las necesidades básicas que tiene el ser humano por conocer y por aprender es que se construyen las herramientas que tiene él para poder buscar y poder recordar. Esto nos motiva a pensar que la necesidad de mejores algoritmos de búsqueda y mejores estructuras de datos seguirán siendo aún una necesidad primordial en la investigación futura de la computación teórica.

Apéndice A
Implementación de un Diccionario

```
1  import java.util.ArrayList;
2  import java.util.Iterator;

4  /**
5   * Implementación de un diccionario usando una lista.
6   */
7  public class DiccionarioLista
8      implements Diccionario{

10     // La lista en donde se guardan las entradas.
11     private ArrayList _lista;

13     // El numero de elementos del diccionario.
14     private int _numElementos;

16     /**
17      * Constructor de un diccionario implementado con una lista.
18      */
19     public DiccionarioLista(){
20         this._lista = new ArrayList();
21     }

23     /**
24      * Método que agrega una pareja (llave,elemento).
25      * @param llave un Object con la llave de la entrada.
26      * @param elemento un Object con el elemento de la entrada.
27      * @return un Object si la llave había sido usada anteriormente regresa el
28      *         elemento anterior, de lo contrario regresa null.
29      */
30     public Object agrega( Object llave,
31                           Object elemento ){
```

```
32          Object elementoAnt = null;
33          Object[] entrada;
34          for( int i=0; i<this._numElementos; i++ ){
35              entrada = (Object[]) this._lista.get(i);
36              if( entrada[0].equals( llave ) ){
37                  this._lista.remove(i);
38                  elementoAnt = entrada[1];
39                  this._numElementos--;
40                  break;
41              }
42          }
43          this._lista.add( new Object[]{llave,
44                                        elemento} );
45          this._numElementos++;
46          return elementoAnt;
47      }
48      /**
49       * Método que regresa el elemento de la entrada dada su llave.
50       * @param llave un Object con la llave de la entrada.
51       * @return un Object con el elemento que corresponde a la llave dada.
52       */
53      public Object daElemento( Object llave ){
54          Object[] entrada;
55          for( int i=0; i<this._numElementos; i++ ){
56              entrada = (Object[]) this._lista.get(i);
57              if( entrada[0].equals( llave ) ){
58                  return entrada[1];
59              }
60          }
61          return null;
62      }
63      /**
64       * Método que regresa la llave de la entrada dado su elemento.
65       * @param elemento un Object con el elemento de la entrada.
66       * @return un Object con la llave que corresponde al elemento dado.
67       */
68      public Object daLlave( Object elemento ){
```

```
69          Object[] entrada;
70          for( int i=0; i<this._numElementos; i++ ){
71              entrada = (Object[])this._lista.get(i);
72              if( entrada[1].equals( elemento ) ){
73                  return entrada[0];
74              }
75          }
76          return null;
77      }

79      /**
80       * Método que pregunta si el elemento, dada su llave, existe en el diccionario.
81       * @param llave un Object con la llave de la entrada.
82       * @return un boolean: true si la entrada existe, false de lo contrario.
83       */
84      public boolean contieneElemento( Object llave ){
85          Object[] entrada;
86          for( int i=0; i<this._numElementos; i++ ){
87              entrada = (Object[])this._lista.get(i);
88              if( entrada[0].equals( llave ) ){
89                  return true;
90              }
91          }
92          return false;
93      }

95      /**
96       * Método que regresa un Iterator de las llaves de este diccionario.
97       * @return un Iterator con la enumeración de las llaves.
98       */
99      public Iterator daLlaves(){
100         return new Iterator(){
101                 int index = 0;
102                 public void remove();
103                 public boolean hasNext(){
104                     return index < this._numElementos;
105                 }
```

```
106                 public Object next(){
107                     Object[] entrada = (Object[])
108                             _lista.get(index++);
109                     return entrada[0];
110                 }
111             };
112     }
113     /**
114      * Método que regresa un Iterator de los elementos de este diccionario.
115      * @return un Iterator con la enumeración de los elementos.
116      */
117     public Iterator daElementos(){
118         return new Iterator(){
119                 int index = 0;
120                 public boolean hasNext(){
121                     return index < _numElementos;
122                 }
123                 public Object next(){
124                     Object[] entrada = (Object[])
125                             _lista.get(index++);
126                     return entrada[1];
127                 }
128                 public void remove();
129             };
130     }
131     /**
132      * Método que borra la entrada dada su llave.
133      * @param llave un Object con la llave de la entrada.
134      * @return un Object con el elemento que corresponde a la llave dada.
135      */
136     public Object borraElemento( Object llave ){
137         Object[] entrada;
138         for( int i=0; i<this._numElementos; i++ ){
139             entrada = (Object[])this._lista.get(i);
140             if( entrada[0].equals( llave ) ){
141                 this._lista.remove(i);
142                 this._numElementos--;
```

```
143                 return entrada[1];
144             }
145         }
146         return null;
147     }

149     /**
150      * Regresa el número de entradas de este diccionario.
151      * @return un int con la longitud del diccionario.
152      */
153     public int longitud(){
154         return this._numElementos;
155     }

157     /**
158      * Regresa una representación en forma de cadena de este diccionario.
159      * @return un String con el diccionario.
160      */
161     public String toString(){
162         StringBuffer sb = new StringBuffer()
163             .append( "[" );
164         Object[] entrada;
165         for( int i=0; i<this._numElementos; i++ ){
166             entrada = (Object[])this._lista.get(i);
167             sb.append( entrada[0] )
168                 .append( "=" )
169                 .append( entrada[1] )
170                 .append( "," );
171         }

173         sb.delete( sb.length()-1,sb.length());
174         sb.append( "]" );
175         return sb.toString();
176     }

178 }//Fin de DiccionarioLista.
```

Apéndice B
Implementación de una Tabla Hash

```
1  public abstract class TablaHash
2      implements Diccionario{

4      public abstract void cambiaFuncionHash
5                  ( FuncionHash funcionHash );

7  }//Fin de TablaHash.

9  public interface FuncionHash{

11     public int calculaLlave( String llave,
12                              int longitudTabla );
13 }//Fin de FuncionHash.

15 import java.util.ArrayList;
16 import java.util.Iterator;

18 public class TablaHashLista
19     extends TablaHash{

21     public ArrayList _lista;

23     public int _capacidad;

25     private int _numElementos;

27     private FuncionHash _funcion;

29     public TablaHashLista(){
30         this( new HashJava() );
31     }
```

```
32      public TablaHashLista( FuncionHash funcionHash ){
33          this._lista = new ArrayList();
34          this._capacidad = 20;
35          for( int i=0; i<this._capacidad; i++ )
36              this._lista.add( new ArrayList() );
37          this._funcion = funcionHash;
38      }

40      public void cambiaFuncionHash
41                  ( FuncionHash funcionHash ){
42          this._funcion = funcionHash;
43          rehash( this._capacidad );
44      }

46      private void rehash( int capacidad ){
47          ArrayList listaNueva = new ArrayList();
48          for( int i=0; i<capacidad; i++ )
49              listaNueva.add( new ArrayList() );

51          this._capacidad = capacidad;
52          int tam = this._lista.size();
53          ArrayList lista;
54          int tamLista;
55          for( int i=0; i<tam; i++ ){
56              lista = (ArrayList)this._lista.get(i);
57              tamLista = lista.size();
58              Object[] entrada;
59              int index;
60              for( int j=0; j<tamLista; j++ ){
61                  entrada = (Object[])lista.get(j);
62                  index = calculaHash( entrada[0] );
63                  ((ArrayList)listaNueva.get(index))
64                                  .add(entrada);
65              }
66          }
67          this._lista = listaNueva;
68      }
```

```
69      public Object agrega( Object llave,
70                            Object elemento ){
71          Object anterior = null;
72
73          if( _numElementos == _capacidad )
74              rehash( (int)(_capacidad*2.0) );
75
76          int index = calculaHash(llave);
77          ArrayList lista = (ArrayList)this._lista
78                            .get( index );
79
80          int tam = lista.size();
81          Object[] entrada;
82          for( int i=0; i<tam; i++ ){
83              entrada = (Object[])lista.get(i);
84              if( entrada[0].equals(llave) ){
85                  anterior = entrada[1];
86                  entrada[1] = elemento;
87                  return anterior;
88              }
89          }
90
91          lista.add( new Object[]{llave,elemento} );
92          this._numElementos++;
93
94          return anterior;
95      }
96
97      private int calculaHash( Object llave ){
98          return this._funcion
99                  .calculaLlave( llave.toString(),
100                                 this._capacidad );
101     }
```

```
102     public Object daElemento( Object llave ){
103         int index = calculaHash(llave);
104         ArrayList lista = (ArrayList)this._lista
105                           .get( index );
107         int tam = lista.size();
108         Object[] entrada;
109         for( int i=0; i<tam; i++ ){
110             entrada = (Object[])lista.get(i);
111             if( entrada[0].equals(llave) )
112                 return entrada[1];
113         }
115         return null;
116     }
118     public Object daLlave( Object elemento ){
119         int tam = this._lista.size();
121         ArrayList lista;
122         int tamLista;
123         for( int i=0; i<tam; i++ ){
124             lista = (ArrayList)this._lista.get(i);
125             tamLista = lista.size();
126             Object[] entrada;
127             int index;
128             for( int j=0; j<tamLista; j++ ){
129                 entrada = (Object[])lista.get(j);
130                 if( entrada[1].equals(elemento) )
131                     return entrada[0];
132             }
133         }
135         return null;
136     }
```

```
137     public boolean contieneElemento( Object llave ){
138         int index = calculaHash(llave);
139         ArrayList lista = (ArrayList)this._lista
140                             .get( index );

142         int tam = lista.size();
143         Object[] entrada;
144         for( int i=0; i<tam; i++ ){
145             entrada = (Object[])lista.get(i);
146             if( entrada[0].equals(llave) )
147                 return true;
148         }

150         return false;
151     }

153     public Iterator daLlaves(){
154         return new Iterator(){
155                 ArrayList _listaNueva;
156                 int index;
157                 public boolean hasNext(){
158                     if( index == 0 )
159                         this._listaNueva =
160                               daArregloLlaves();
161                     return index != _numElementos;
162                 }
163                 public Object next(){
164                     return this._listaNueva
165                             .get(index++);
166                 }
167                 public void remove(){
168                 }
169             };
170     }
```

```
171     private ArrayList daArregloLlaves(){
172         ArrayList listaNueva = new ArrayList();

174         ArrayList lista;
175         int tamLista;
176         Object[] entrada;
177         for( int i=0; i<_capacidad; i++ ){
178             lista = (ArrayList)_lista.get(i);
179             tamLista = lista.size();
180             for( int j=0; j<tamLista; j++ ){
181                 entrada = (Object[])lista.get(j);
182                 listaNueva.add( entrada[0] );
183             }
184         }

186         return listaNueva;
187     }

189     public Iterator daElementos(){
190         return new Iterator(){
191                 ArrayList _listaNueva;
192                 int index;
193                 public boolean hasNext(){
194                     if( index == 0 )
195                         this._listaNueva =
196                                 daArregloElementos();
197                     return index != _numElementos;
198                 }
199                 public Object next(){
200                     return this._listaNueva
201                             .get(index++);
202                 }
203                 public void remove(){
204                 }
205             };
206     }
```

```
207     private ArrayList daArregloElementos(){
208         ArrayList listaNueva = new ArrayList();

210         ArrayList lista;
211         int tamLista;
212         Object[] entrada;
213         for( int i=0; i<_capacidad; i++ ){
214             lista = (ArrayList)_lista.get(i);
215             tamLista = lista.size();
216             for( int j=0; j<tamLista; j++ ){
217                 entrada = (Object[])lista.get(j);
218                 listaNueva.add( entrada[1] );
219             }
220         }

222         return listaNueva;
223     }

225     public Object borraElemento( Object llave ){
226         int index = calculaHash(llave);
227         ArrayList lista = (ArrayList)this._lista
228                                         .get( index );

230         int tam = lista.size();
231         Object[] entrada;
232         for( int i=0; i<tam; i++ ){
233             entrada = (Object[])lista.get(i);
234             if( entrada[0].equals(llave) ){
235                 lista.remove(i);
236                 this._numElementos--;
237                 return entrada[1];
238             }
239         }

241         return null;
242     }
```

```
243     public int longitud(){
244         return this._numElementos;
245     }

247     public String toString(){
248         StringBuffer sb = new StringBuffer();
249         sb.append( "[" );

251         ArrayList lista;
252         int tamLista;
253         Object[] entrada;
254         for( int i=0; i<this._capacidad; i++ ){
255             lista = (ArrayList)this._lista.get(i);
256             tamLista = lista.size();
257             if( tamLista == 0 )
258                 continue;

260             sb.append( "(" );
261             for( int j=0; j<tamLista; j++ ){
262                 entrada = (Object[])lista.get(j);
263                 sb.append( entrada[0] )
264                     .append( "=" )
265                     .append( entrada[1] );
266             }
267             sb.append( ")" );
268         }
269         sb.append( "]" );
270         return sb.toString();
271     }

273 }//Fin de TablaHashLista.java
```

Bibliografía

- [Knuth]

  Knuth, Donald E.: *The Art of Computer Programming*. Addison-Wesley Publishing Company. USA, 1973. Págs. 506-549.

- [Cormen]

  Cormen, Thomas H.: *Introduction to Algorithms*. MIT Press. USA, 1990. Págs. 219-243.

- [Menezes]

  Menezes, Alfred J.: *Handbook of Applied Criptography*. CRC Press LLC. USA, 1997. Págs. 321-383.

- [Standish]

  Standish, Thomas A.: *Data Structures, Algorithms, and Software Principles*. Addison-Wesley Publishing Company. USA, 1994. Págs. 450-496.

- [Damgård]

  Damgård, Ivan.: *A design principle for hash functions*. In G. Brassard, editor, Advances in Cryptology-CRYPTO' 89, volume 435 of Lecture Notes in Computer Science. Springer-Verlag, 1990.

- [Rivest]

  Rivest, R.: *The MD4 Message Digest Algorithm*. RFC 1186, MIT, 1990.

- [Merkle]

  Merkle, R. C.: *One Way Hash Functions and DES*. Advances in Cryptology - Crypto'89, LNCS 435, Springer 1989.