

Estructuras de Datos. Diccionarios

Lic. Santiago Zanella

2008

1 Introducción

Para muchas aplicaciones se requiere mantener un conjunto dinámico que permita las operaciones INSERT, SEARCH y DELETE, llamado también diccionario. Por ejemplo, un compilador necesita mantener una tabla de símbolos para buscar información sobre cada identificador en un programa, insertando identificadores para las variables locales de un procedimiento, buscando información sobre los identificadores en el contexto actual y eliminándolos cuando salen de alcance.

Si no existiesen limitaciones de espacio, se podría implementar un diccionario simplemente utilizando la *clave* de un elemento como índice en una tabla de búsqueda directa –un arreglo–. Si no existiesen limitaciones de tiempo, se podría implementar utilizando una lista enlazada con un requerimiento de espacio mínimo. Una *tabla hash* es una estructura de datos que intenta encontrar un balance entre estos dos extremos.

Cuando el número de claves que se almacenan efectivamente es pequeño comparado con el número total de claves posibles, una tabla hash ofrece una alternativa eficiente a una tabla de búsqueda directa, utilizando solamente un arreglo de tamaño proporcional a la cantidad de claves almacenadas. En lugar de utilizar la clave como un índice directamente, el índice se calcula a partir de la clave utilizando una *función hash*.

Aunque el costo de una búsqueda en una tabla hash de n elementos en el peor caso es el mismo que el costo de una búsqueda en una lista enlazada – $\Theta(n)$ –, en la práctica resulta mucho más eficiente. Bajo ciertas hipótesis, el tiempo esperado de una búsqueda de un elemento en una tabla hash es $O(1)$.

2 Tablas de Direccionamiento Directo

El direccionamiento directo es una técnica simple que funciona bien cuando el universo U de claves es razonablemente pequeño. Supongamos que una aplicación necesita un conjunto dinámico en el cual cada elemento posee una clave dentro de un universo

$$U = \{0, 1, \dots, m - 1\}$$

donde m no es demasiado grande. Supongamos además que no existen dos elementos con la misma clave.

Para representar el conjunto dinámico utilizamos un arreglo como una *tabla de direccionamiento directo*

$$T[0 \dots m - 1]$$

en la cual cada posición, o *slot*, corresponde a una clave en el universo U . La Fig. 1 muestra la idea: un slot k apunta a un elemento en el conjunto con clave k . Si no existe un elemento con clave k en el conjunto entonces $T[k] = \perp$.

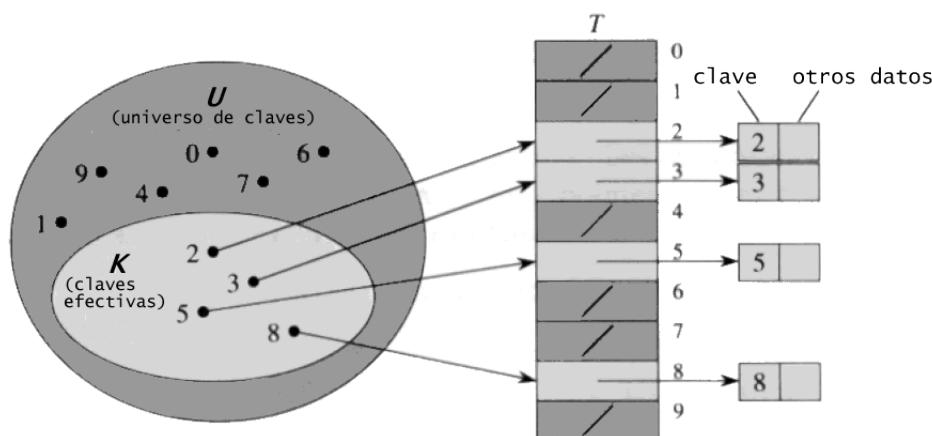


Figura 1: Implementación de un diccionario mediante una tabla de direccionamiento directo T . Cada clave en el universo $U = \{0, 1, \dots, 9\}$ corresponde a un índice en la tabla. El conjunto $K = \{2, 3, 5, 8\}$ de las claves efectivas determina los slots en la tabla que contienen elementos, los otros slots sombreados contienen el valor \perp .

```

1 procedure Initialize(T)
2   for i := 0 to m - 1 do
3     T[i] :=  $\perp$ ;
4   end
5 end

```

```
1 procedure Insert(T, x)
2     T[key(x)] := x;
3 end
```

```
1 function Search(T, k)
2     return T[k];
3 end
```

```
1 procedure Delete(T, x)
2     T[key(x)] :=  $\perp$ ;
3 end
```

Cada una de las operaciones requiere tiempo $\Theta(1)$, que es la cota inferior para cualquier implementación. El orden de espacio en cambio es $\Theta(|U|)$.

3 Lista Enlazada

Se puede implementar un diccionario sobre una lista enlazada, utilizando las operaciones de inserción, búsqueda y borrado sobre listas vistas anteriormente.

La implementación de diccionarios mediante listas enlazadas es eficiente en el uso del espacio: para almacenar n elementos utiliza espacio $\Theta(n)$, la cota inferior para cualquier implementación. Sin embargo, cada operación toma tiempo $O(n)$.

4 Tablas Hash

La dificultad del direccionamiento directo es obvia: si el universo U es grande, resulta impráctico sino imposible mantener una tabla T de tamaño $|U|$. Además, cuando el tamaño del conjunto K de claves efectivamente almacenadas es relativamente pequeño comparado con el tamaño de U , la mayoría del espacio de la tabla de direccionamiento directo se desperdicia.

Una tabla hash requiere tan solo $\Theta(|K|)$ espacio para almacenar un conjunto K de claves y permite buscar, eliminar e insertar un elemento en tiempo $O(1)$ (la trampa está en que para tablas hash este tiempo se cumple en el caso promedio mientras que para tablas de direccionamiento directo se cumple también en el peor caso).

Con direccionamiento directo, un elemento con clave k se almacena en el slot k . Con hashing, el mismo elemento se almacena en el slot $h(k)$, donde h es una *función hash* usada para computar el slot para una clave k :

$$h : U \longrightarrow \{0, 1, \dots, m - 1\}$$

La Fig. 2 muestra la idea. El punto es que la función hash reduce el rango de índices en el arreglo que se manipula. En lugar de $|U|$ valores, solamente se manipulan m índices, con el correspondiente ahorro de espacio.

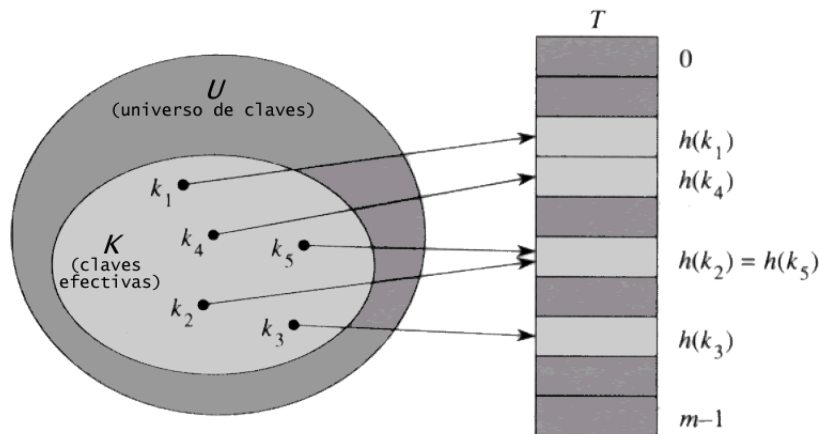


Figura 2: Uso de una función hash h para obtener slots en la tabla a partir de claves. Las claves k_2 y k_5 tienen el mismo valor hash, colisionan.

El inconveniente de esta solución tan elegante es que como $m < |U|$, existen necesariamente claves en el universo que tienen el mismo valor de hash –produciendo colisiones–. Una función de hash adecuada puede minimizar el número de colisiones, pero no evitarlas completamente y por lo tanto es necesario un método para tratar las colisiones.

4.1 Resolución de Colisiones por Encadenamiento

La técnica de encadenamiento soluciona las colisiones insertando todos los elementos con el mismo valor hash en el mismo slot usando una lista enlazada. El slot j contiene solo un puntero a la cabeza de la lista que contiene todos los elementos almacenados con valor hash j (Fig. 3).

```

1 procedure Initialize(T)
2   for i := 0 to m - 1
3     T[i] :=  $\perp$ ;
4   end
5 end

```

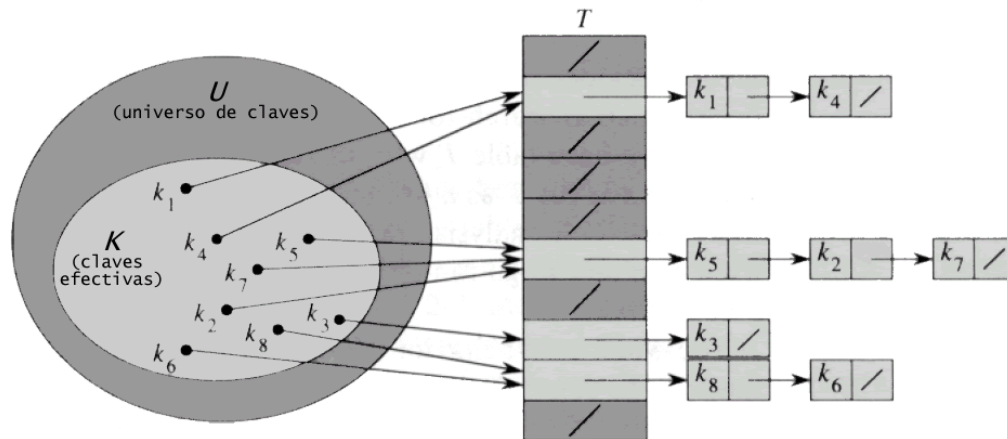


Figura 3: Manejo de colisiones por encadenamiento. Cada slot $T[j]$ en la tabla hash contiene una lista enlazada de todas las claves cuyo valor hash es j . Por ejemplo, $h(k_1) = h(k_4)$ y $h(k_2) = h(k_5) = h(k_7)$

```

1 procedure Insert( $T, x$ )
2   insertar  $x$  al principio de la lista en  $T[h(\text{key}(x))]$ ;
3 end

```

```

1 function Search( $T, k$ )
2   buscar el un elemento con clave  $k$  en la lista  $T[h(k)]$ ;
3 end

```

```

1 procedure Delete( $T, x$ )
2   borrar  $x$  de la lista  $T[h(\text{key}(x))]$ ;
3 end

```

Una inserción toma siempre tiempo $O(1)$. La búsqueda y el borrado en el peor caso toman tiempo proporcional al tamaño de la lista.

4.1.1 Análisis de la técnica de encadenamiento

Definición Factor de carga

Dada una tabla hash T con m slots que almacena n elementos, se define el factor de carga α de T como

$$\alpha = \frac{n}{m}$$

Observar que el factor de carga de una tabla puede ser inferior, igual o superior a uno.

El factor de carga de una tabla se puede interpretar como el número promedio de elementos almacenados en cada *cadena*. Analizaremos la complejidad de las operaciones sobre la

estructura en términos de α .

El peor caso para una tabla hash que use la técnica de encadenamiento es que todas las claves almacenadas tengan el mismo valor hash, en cuyo caso se termina con una única lista enlazada que contiene todos los elementos almacenados y el costo de la búsqueda y el borrado es $O(n)$.

El costo promedio depende de cuán uniformemente la función hash distribuya el conjunto de claves entre los m slots de la tabla. Por ahora, asumiremos que es equiprobable que una clave cualquiera dada del universo caiga en cualquiera de los m slots, esta hipótesis se denomina hipótesis de *hashing uniforme*. Se asume también que el valor hash $h(k)$ puede computarse en tiempo $O(1)$.

El tiempo requerido para buscar un elemento con clave k depende linealmente de la longitud de la lista $T[h(k)]$ y de si existe o no un elemento con esa clave en la lista.

Teorema 4.1 *En una tabla hash en la cual las colisiones se manejan por encadenamiento, una búsqueda no exitosa toma tiempo $\Theta(1 + \alpha)$ en el caso promedio, bajo la hipótesis de hashing uniforme.*

Demostración Para determinar que no se encuentra un elemento con clave k en la lista $T[h(k)]$ es necesario recorrer toda la lista. El tiempo de búsqueda está dado por la longitud de la lista, que en promedio por la hipótesis de hashing uniforme es igual al factor de carga $\alpha = n/m$ de la tabla. Por lo tanto, el número de comparaciones efectuadas en una búsqueda no exitosa es en promedio α y el tiempo total requerido es $\Theta(1 + \alpha)$ (contando el tiempo necesario para calcular $h(k)$)

Teorema 4.2 *En una tabla hash donde las colisiones se resuelven por encadenamiento, una búsqueda exitosa toma tiempo $\Theta(1 + \alpha)$ en el caso promedio, bajo la hipótesis de hashing uniforme.*

Demostración Se asume que la clave que se busca puede ser cualquiera de las n claves almacenadas en la lista con igual probabilidad. Como al insertar un elemento nuevo se lo agrega al principio de la lista del slot que le corresponde, el número promedio de comparaciones efectuadas durante una búsqueda de ese elemento es igual a 1 más el número de elementos subsiguientes que se insertan en ese slot.

Para encontrar el número promedio de elementos examinados al buscar un elemento se calcula el promedio de elementos que se insertaron posteriormente en la lista donde se insertó. Para encontrar el número promedio de comparaciones de cualquier búsqueda se promedia luego sobre los n elementos. Como la probabilidad de que un elemento se inserte en un slot cualquiera es $1/m$, el costo $C(n, m)$ de una búsqueda en la tabla es

$$C(n, m) = \frac{1}{n} \sum_{i=1}^n \left(1 + \overbrace{\sum_{j=i+1}^n \frac{1}{m}}^{\text{elementos insertados posteriormente}} \right) \quad (1)$$

$$= 1 + \frac{1}{n} \sum_{i=1}^n \left(\sum_{j=i+1}^n \frac{1}{m} \right) \quad (2)$$

$$= 1 + \frac{1}{nm} \sum_{i=1}^n (n - i) \quad (3)$$

$$= 1 + \frac{1}{nm} \left(\sum_{i=1}^n n - \sum_{i=1}^n i \right) \quad (4)$$

$$= 1 + \frac{1}{nm} \left(n^2 - \frac{n(n+1)}{2} \right) \quad (5)$$

$$= 1 + \frac{n}{m} - \frac{n+1}{2m} \quad (6)$$

$$= 1 + \frac{n-1}{2m} \quad (7)$$

$$= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n} \quad (8)$$

$$= \Theta(1 + \alpha) \quad (9)$$

¿Qué conclusión puede sacarse de este análisis? Si el número de slots en la tabla es al menos proporcional al número de elementos, entonces $n = O(m)$ y

$$\alpha = n/m \leq cm/m = c = O(1)$$

Como una operación INSERT toma tiempo constante, entonces el costo de cualquier operación sobre un diccionario implementado con la técnica de encadenamiento es de orden $O(1)$.

5 Funciones Hash

Una buena función hash debe poder calcularse en tiempo constante $O(1)$ y satisfacer (al menos aproximadamente) la hipótesis de hashing uniforme: es equiprobable que una clave dada tenga cualquier valor hash entre 0 y $m - 1$.

En la práctica no es posible elegir una función que satisfaga esta hipótesis porque no se conoce de antemano la verdadera distribución de probabilidad con la que se insertan claves en la tabla hash. Se utilizan algunas heurísticas para elegir una función adecuada. Por

ejemplo, en el caso de la tabla de símbolos de un compilador, en la cual las claves son cadenas de caracteres que representan identificadores de un programa es común que dos identificadores muy similares, como *tmp* y *temp* aparezcan en el mismo programa. Una buena función hash en este caso minimizaría la chance de que dos claves muy similares colisionen.

5.1 Claves como Números Naturales

La mayoría de las funciones hash asumen que el universo de claves es el conjunto \mathbb{N} de números naturales. Por lo tanto, si las claves no son números naturales se debe encontrar una forma de interpretarlas como si lo fueran. Por ejemplo, una clave que sea una cadena de caracteres puede interpretarse como un número entero en una determinada base, considerando cada caracter como un dígito. Por ejemplo, para cadenas de caracteres del alfabeto $\{a, b, \dots, z\}$ podemos asignarle a cada letra un valor del 0 al 25:

$$\text{tmp} = 19 \cdot 26^2 + 12 \cdot 26^1 + 15 \cdot 26^0 \quad (10)$$

$$\text{temp} = 19 \cdot 26^3 + 4 \cdot 26^2 + 12 \cdot 26^1 + 15 \cdot 26^0 \quad (11)$$

Se puede encontrar para cualquier tipo de clave una interpretación similar como número natural considerando su representación binaria.

5.2 Método del resto

El *método del resto* la asigna a una clave como valor hash el resto de su división por m , la cantidad de slots en la tabla:

$$h(k) = k \bmod m$$

Cuando se utiliza este método se debe tener cuidado del valor de m que se elige. Por ejemplo, si m es una potencia de 2, 2^p , $h(k)$ está dado por los primeros p bits de k , sin tener en cuenta los bits de orden superior. Al menos que se sepa de antemano que la distribución de los primeros p bits de las claves es uniforme, conviene buscar una función que tenga en cuenta todos los bits de las claves. Otro ejemplo, si $m = 2^p - 1$ y k es una cadena de caracteres representada en base 2^p , dos cadenas idénticas excepto por una transposición de dos caracteres adyacentes tienen el mismo valor hash. En general, los valores primos son buenos candidatos para m .

5.3 Método de Horner

Cuando se usan cadenas de caracteres como claves su interpretación como números naturales puede fácilmente sobrepasar la capacidad de representación de números enteros de la

máquina.

El método del resto se adapta particularmente bien a este caso. Utilizando una técnica de evaluación de polinomios conocida como método de Horner junto con propiedades de la función mod se puede calcular el valor hash de una clave sin calcular su representación como número natural:

Lema 5.1 Sean $m, k, l \in \mathbb{N}$, entonces

$$(k + l) \bmod m = (k \bmod m + l) \bmod m \quad (12)$$

$$(k l) \bmod m = ((k \bmod m) l) \bmod m \quad (13)$$

Lema 5.2 Método de Horner

$$\sum_{i=0}^n a_i b^i = (\dots (((a_n b + a_{n-1}) b) + \dots + a_1) b + a_0$$

Usando las propiedades de la función mod enunciadas anteriormente y el método de Horner, puede construirse un procedimiento que calcule la representación numérica de una clave módulo m sin calcular explícitamente esta representación:

```
1 function h(string key)
2   r := 0;
3   for i := length(key) to 0 do
4     r := (r * b + key[i]) mod m;
5   end
6   return r;
7 end
```

6 Direccionamiento Abierto (Closed Hashing)

En la técnica de *direccionamiento abierto* para manejo de colisiones todos los elementos se almacenan en la tabla hash misma. Es decir, cada entrada en la tabla contiene un elemento del diccionario o está vacía. Cuando se busca un elemento, se examinan sistemáticamente todos los slots hasta que se encuentra el elemento deseado o se determina que no se encuentra en la tabla. No hay listas de elementos almacenadas fuera de la tabla como con la técnica de encadenamiento y por lo tanto, en el direccionamiento abierto, es posible que la tabla se complete en su totalidad y que no se puedan realizar más inserciones. El factor de carga α de una tabla hash con direccionamiento abierto nunca puede exceder 1.

Para realizar una inserción usando el método de direccionamiento abierto se examinan sistemáticamente los slots de la tabla hasta que se encuentra una posición libre. En lugar

de usar un orden prefijado como $0, 1, \dots, m - 1$ (que requiere un tiempo de búsqueda $\Theta(n)$), la secuencia de posiciones que se examinan *depende de la clave que se inserta*. Para determinar los slots a examinar, se extiende la función hash para incluir el número de *salto* como un segundo argumento:

$$h : U \times \{0, 1, \dots, m - 1\} \longrightarrow \{0, 1, \dots, m - 1\}$$

Con el direccionamiento abierto, para cada clave k se examina la secuencia de slots:

$$h(k, 0), h(k, 1), \dots, h(k, m - 1)$$

y se requiere que dicha secuencia de slots sea exhaustiva, es decir que sea una permutación de $0, 1, \dots, m - 1$, de forma tal que todos los slots en la tabla se examinen.

```

1 function Insert (T, x)
2   k := key(x);
3   i := 0;
4   do j := h(k, i)
5     if T[j] =  $\perp$  then
6       T[j] := x;
7       return j;
8     else
9       i := i + 1;
10    end
11    until i = m
12    return OVERFLOW;
13 end

```

El algoritmo de búsqueda examina la misma secuencia de slots que en la inserción y termina con éxito al encontrar la clave buscada o sin éxito al encontrar una posición vacía, porque de haberse insertado el elemento debería haber ocupado esa posición –consideraremos que pasa con los borrados más tarde–.

```

1 function Search (T, k)
2   i := 0;
3   do j := h(k, i)
4     if key(T[j]) = k then
5       return T[j];
6     else
7       i := i + 1;
8     end
9     until i = m or T[j] =  $\perp$ 
10    return  $\perp$ ;
11 end

```

La operación DELETE debe implementarse con cuidado. Cuando se borra un slot i , no se puede simplemente marcar la posición como vacía porque afectaría a las operaciones SEARCH subsiguientes. Una solución es utilizar otro valor para indicar una posición que contenía un elemento que fue borrado y modificar la operación de búsqueda para que continúe si encuentra ese tipo de posiciones y la operación de inserción para que considere vacías las posiciones de ese tipo. Cuando se hace esto sin embargo, el tiempo de búsqueda deja de ser dependiente del factor de carga α . Por esta razón, cuando se requiere la operación DELETE, se prefiere la técnica de encadenamiento para manejo de colisiones.

6.1 Examen Lineal

Sea h' una función hash ordinaria, el método de examen lineal utiliza la técnica de direccionamiento abierto con la función hash

$$h(k, i) = (h'(k) + i) \bmod m$$

La secuencia de slots que examina el método es:

$$h'(k), h'(k) + 1, \dots, m - 1, 1, \dots, h'(k) - 1$$

Como la posición inicial determina la secuencia, solo existen m secuencias diferentes que pueden usarse.

El examen lineal es fácil de implementar pero sufre de un problema conocido como el *problema de aglomeración -clustering-*. Luego de pocas inserciones, tienden a formarse largas aglomeraciones de posiciones consecutivas ocupadas. Los clusters tienden a formarse porque si existe un slot vacío precedido de i slots ocupados, la probabilidad de que sea el próximo slot que se use es $(i + 1)/m$ comparado con la probabilidad $1/m$ en el caso de que el slot precedente también esté vacío. Por esto, los clusters de slots ocupados tienden a crecer con el tiempo y no se cumple la hipótesis de hashing uniforme.

6.2 Hashing Doble

La técnica de *Hashing Doble* es una de las mejores opciones para direccionamiento abierto. Utiliza una función hash de la forma

$$h(k, i) = (h_1(k) + i h_2(k)) \bmod m$$

donde h_1 y h_2 son funciones hash auxiliares. Al contrario que en la técnica de examen lineal, los slots examinados no solo dependen del slot inicial sino también del valor de la clave k .

El valor de $h_2(k)$ debe ser primo relativo del tamaño de la tabla para asegurar que se recorran todos los slots. El uso de una segunda función hash para calcular el *salto* en cada examen incrementa el número de secuencias posibles, siendo $\Theta(m^2)$ en lugar de $\Theta(m)$ como en el caso del examen lineal.

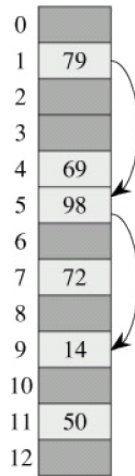


Figura 4: Inserción por doble hashing de la clave 14 en una tabla de tamaño 13 con $h_1(k) = k \bmod 13$ y $h_2(k) = 1 + (k \bmod 11)$

7 Análisis del Direccionamiento Abierto

El análisis del direccionamiento abierto como en el caso del análisis del encadenamiento, se realiza en términos del factor de carga α de la tabla. Por supuesto en este caso el factor de carga nunca puede ser mayor que 1 porque todos los elementos se almacenan en la misma tabla y por lo tanto $n \leq m$.

Solamente enunciaremos los resultados principales sin demostrarlos.

Teorema 7.1 *En una tabla hash con manejo de colisiones por direccionamiento abierto y factor de carga $\alpha < 1$, el número promedio de comparaciones en una búsqueda no exitosa es a lo sumo $1/(1 - \alpha)$, asumiendo la hipótesis de hashing uniforme.*

Corolario 7.2 *La inserción de un elemento en una tabla hash con manejo de colisiones por direccionamiento abierto y factor de carga $\alpha < 1$ requiere a lo sumo $1/(1 - \alpha)$ comparaciones en promedio, asumiendo la hipótesis de hashing uniforme.*

Teorema 7.3 *Dada una tabla hash con manejo de colisiones por direccionamiento abierto y factor de carga $\alpha < 1$, el número promedio de comparaciones en una búsqueda exitosa es a lo sumo*

$$\frac{1}{\alpha} \ln \frac{1}{1 - \alpha}$$

asumiendo la hipótesis de hashing uniforme y que cada clave puede buscarse con igual probabilidad.

Referencias

- [AHU74] Alfred V. Aho, J. E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
- [CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.
- [Sed92] Robert Sedgewick. *Algorithms in C++*. Addison-Wesley, 1992. ISBN: 0-201-51059-6.