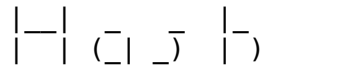
"Análisis de Algoritmos"
José Helo Guzmán



Contenido

▬▬▬ 1. Introducción.

En este capítulo aprenderemos sobre las “hash tables”, una de las estructuras más básicas y que tienen múltiples usos en los sistemas actuales.

Hablaremos sobre las “hash tables” internas, su implementación y cómo resolver los problemas de las colisiones. De esta forma, seremos capaces de analizar el desempeño de esta herramienta.

▬▬▬ 2. Linear Search y Binary Search.

En una sección anterior se mencionó el algoritmo de “Linear Search”. Se debe buscar la posición del índice en una lista cuyos elementos no tienen ningún orden.

Por ejemplo para encontrar un elemento en esta lista:

lista: [8, 5,12, 6,15, 9, 4, 3, 7,10]
1 2 3 4 5 6 7 8 9 10

En la discusión anterior habíamos encontrado que el tiempo de búsqueda está dado por:

O(n)

En el caso del algoritmo del “Binary Search” requiere que la lista esté ordenada. Por ejemplo:

lista: [3, 4, 5, 6, 7, 8, 9,10,12,15]
1 2 3 4 5 6 7 8 9 10

Se había determinado que en este caso el tiempo de búsqueda, sin incluir el tiempo en que se incurre al ordenar, o al mantener la lista ordenada, estaba dado por:

$O(\log_2(n))$

Como un recordatorio. Hay una gran diferencia entre O(n) y $O(\log_2(n))$. Por ejemplo supongamos que se está buscando un login en una red social, y que el tiempo de búsqueda está dado por 10 nombres por segundo. Entonces se necesitaría de estos tiempos para hacer un login:

Número de nombres	O(n)	$O(\log_2(n))$
100	10.00 seg	1 seg pues $\log_2(100) = 7$
1000	1.66 min	1 seg pues $\log_2(1000) = 10$
10000	16.66 min	2 seg pues $\log_2(10000) = 14$

Lo que se desea es construir un proceso de búsqueda que sea capaz de bajar el tiempo de búsqueda, lo más cercano posible a O(1)!!!

▬▬▬ 3. Conversión de Strings a Números.

En un sentido estricto, uno de los principales usos de los “hash tables” es para lo que se conoce como diccionarios en lenguajes de programación.

Uno de las primeras cosas que se necesita es poder convertir “strings” a números. Por ejemplo, supongamos que tenemos el nombre de “Ana”.

Cada letra del nombre se convierte a su valor de ascii.

“A” : 65

“n” : 110
“a” : 97

La suma da un total de: 272

“Ana” = 272

Lo mismo ocurre con el nombre de “Pedro”

“P” : 80
“e” : 101
“d” : 100
“r” : 114
“o” : 111

La suma da un total de: 506

“Pedro” = 506

▬▬▬ 4. Una Primera Aproximación.

Supongamos que tenemos una lista de 11 nombres con las notas de su última evaluación, y deseamos guardar estos valores en un vector:

Tenemos los datos:

"Mia", nota 80
"Tim", nota 100
"Bea", nota 70
"Zoe", nota 70
"Jan", nota 80
"Ada", nota 70
"Leo", nota 90
"Sam", nota 90
"Lou", nota 70
"Max", nota 95
"Ted", nota 85

Los cuales representaremos en la siguiente lista:

```
datos:
[
["Mia", 80],
["Tim",100],
["Bea", 70],
["Zoe", 70],
["Jan", 80],
["Ada", 70],
["Leo", 90],
["Sam", 90],
["Lou", 70],
["Max", 95],
["Ted", 85]
];
```

Como en este caso tenemos 11 nombres, se podría iniciar con un vector de 11 posiciones. Para colocar adecuadamente cada uno de los datos se suma su ascii. Por ejemplo

"Mia" = 77 + 105 + 97 = 279,

Para colocar a “Mia” en su posición usamos el siguiente algoritmo. Nuestros vectores inician en 1, y la función mod(x,11) produce valores en [0,10].

```
hash(largo,str):=block
(  [tot,index],
   tot:sumarStr(str),
   index: mod(tot,largo),
   if index=0 then index:largo,
   index
);
```

Por lo tanto, la posición de “Mia” estará dada por:

"Mia" = 77 + 105 + 97 = 279,

mod(279,11)=4

Por lo tanto colocamos a Mia en la posición 4 del arreglo.

```
01[]
02[]
03[]
04["Mia",80]
05[]
06[]
07[]
08[]
09[]
10[]
11[]
```

Aquí están los demás valores:

```
"Mia" =  77 + 105 +  97 = 279, mod(279,11)= 4
"Tim" =  84 + 105 + 109 = 298, mod(298,11)= 1
"Bea" =  66 + 101 +  97 = 264, mod(264,11)= 0 (Posición 11)
"Zoe" =  90 + 111 + 101 = 302, mod(302,11)= 5
"Jan" =  74 +  97 + 110 = 281, mod(281,11)= 6
"Ada" =  65 + 100 +  97 = 262, mod(262,11)= 9
"Leo" =  76 + 101 + 111 = 288, mod(288,11)= 2
"Sam" =  83 +  97 + 109 = 289, mod(289,11)= 3
"Lou" =  76 + 111 + 117 = 304, mod(304,11)= 7
"Max" =  77 +  97 + 120 = 294, mod(294,11)= 8
"Ted" =  84 + 101 + 100 = 285, mod(285,11)= 10
```

Por lo que se produce el vector:

```
01["Tim",100]
02["Leo", 90]
```

03["Sam", 90]
04["Mia", 80]
05["Zoe", 70]
06["Jan", 80]
07["Lou", 70]
08["Max", 95]
09["Ada", 70]
10["Ted", 85]
11["Bea", 70]

Como podemos ver el arreglo está completamente lleno y cada uno de los nombres tiene una casilla diferente.

De esta forma, si queremos saber la nota de “Ada”, debemos hacer los siguiente:

```
>>> find("Ada")
```

"Ada" = 65 + 100 + 97 = 262, mod(262,11)= 9

Buscamos el contenido de vector[9] = 70

En este caso, en que se cada elemento produce un índice diferente, el tiempo de búsqueda estará dado por O(1).

▬▬▬ 5. Algoritmo de Hashing.

El algoritmo de hashing es entonces un proceso donde:

- Se aplica una función a una llave para convertirla en una dirección.
- Para llaves numéricas, se pueden usar diferentes funciones.

Existe otro método conocido como la función de folding. Por ejemplo, si se tiene un número telefónico, se puede hacer la suma de sus pares.

506 8842 1923

= 05 + 06 + 88 + 42 + 19 + 23

= 183

Y luego, dependiendo del tamaño del vector, se puede dividir por una constante aplicar la función módulo.

La función correcta de “hashing” depende en gran medida del tipo de datos que se tenga.

▬▬▬ 6. Colisiones con Open Adressing.

En algunos casos puede ocurrir que al aplicar la función de hashing a dos valores diferentes se produzca el mismo valor del índice. Esto se conoce como una colisión. Veamos este caso con un conjunto de datos diferente.

●● Primer Ejemplo de Colisiones.

Se tiene los datos:

```
datos:
[
["Mia", 80],
["Tim",100],
["Bea", 70],
["Zoe", 70],
["Sue", 70],

["Len", 85],
["Moe", 90],
["Lou", 70],
["Rae", 80],
["Max", 95],
["Tod", 95]
];
```

>>> Iniciemos el proceso:

Iniciamos con un vector vacío.

```
01[]
02[]
03[]
04[]
05[]
06[]
07[]
08[]
09[]
10[]
11[]
```

```
"Mia" = 279, mod(279,11)= 4
"Tim" = 298, mod(298,11)= 1
"Bea" = 264, mod(264,11)= 0, Posición 11!
"Zoe" = 302, mod(302,11)= 5
"Sue" = 301, mod(301,11)= 4  Esta posición ya está ocupada!!!
                             por lo tanto buscamos la siguiente
                             posición disponible.
```

Por lo tanto nuestro vector queda de la siguiente manera.

```
01["Tim",100]
02[]
03[]
04["Mia", 80]
05["Zoe", 70]
```

06["Sue", 70]
07[]
08[]
09[]
10[]
11["Bea", 70]

>>> Seguimos con:

"Len" = 287, mod(287,11)= 1
Ya está ocupada!
Procedemos a la siguiente casilla libre

01["Tim",100]
02["Len", 85]
03[]
04["Mia", 80]
05["Zoe", 70]
06["Sue", 70]
07[]
08[]
09[]
10[]
11["Bea", 70]

>>> Seguimos con:

"Moe" = 289, mod(289,11)= 3

01["Tim",100]
02["Len", 85]
03["Moe", 90]
04["Mia", 80]
05["Zoe", 70]
06["Sue", 70]
07[]
08[]
09[]
10[]
11["Bea", 70]

>>> Seguimos con:

"Lou" = 304, mod(304,11)= 7

01["Tim",100]
02["Len", 85]
03["Moe", 90]
04["Mia", 80]
05["Zoe", 70]
06["Sue", 70]
07["Lou", 70]
08[]
09[]
10[]
11["Bea", 70]

>>> Seguimos con:

"Rae" = 280, mod(280,11)= 5

01["Tim",100]
02["Len", 85]
03["Moe", 90]
04["Mia", 80]
05["Zoe", 70]
06["Sue", 70]
07["Lou", 70]
08["Rae", 80]
09[]
10[]
11["Bea", 70]

>>> Seguimos con:

"Max" = 294, mod(294,11)= 8

01["Tim",100]
02["Len", 85]
03["Moe", 90]
04["Mia", 80]
05["Zoe", 70]
06["Sue", 70]
07["Lou", 70]
08["Rae", 80]
09["Max", 95]
10[]
11["Bea", 70]

>>> Seguimos con:

"Tod" = 295, mod(295,11)= 9

01["Tim",100]
02["Len", 85]
03["Moe", 90]
04["Mia", 80]
05["Zoe", 70]
06["Sue", 70]
07["Lou", 70]
08["Rae", 80]
09["Max", 95]
10["Tod", 95]
11["Bea", 70]

Si se llega al final del arreglo y no se encuentra un espacio, se debe ir al inicio del arreglo y encontrar ahí un espacio.

Para localizar a alguien en esta tabla, se utiliza el nombre y si no se encuentra en la posición buscada, se iniciará una búsqueda lineal. Por ejemplo, si queremos encontrar a Rae.

find("Rae");

"Rae" = 280, mod(280,11)= 5

01["Tim",100]
02["Len", 85]
03["Moe", 90]
04["Mia", 80]
05["Zoe", 70]◄
06["Sue", 70]◄
07["Lou", 70]◄
08["Rae", 80]◄
09["Max", 95]
10["Tod", 95]
11["Bea", 70]

Bajo estas condiciones, en el peor de los casos el tiempo de búsqueda podría tardar O(n).

Entre mayor es el número de datos, mayor serán el número de colisiones. Otra forma de evitar esto es hacer el vector más grande de lo que se ocupa. En general, se utiliza la regla que solo esté lleno un 70% del vector.

●● Segundo Ejemplo de Colisiones.

Si se tienen los siguientes datos:

```
datos:[
["Max", 85],
["Tod", 95],
["Bev",100],
["Bea",100],
["Jon", 92]
];
```

Iniciamos con un vector vacío.

01[]
02[]
03[]
04[]
05[]
06[]
07[]
08[]
09[]
10[]
11[]

"Max" = 294, mod(294,11)= 8,
"Tod" = 295, mod(295,11)= 9,
"Bev" = 285, mod(285,11)=10,
"Bea" = 264, mod(264,11)= 0, posición 11
"Jon" = 295, mod(295,11)= 9, Colisión.

Que se organizan como:

01["Jon", 92]

02[]
03[]
04[]
05[]
06[]
07[]
08["Max", 85]
09["Tod", 95]
10["Bev",100]
11["Bea",100]

Se debe buscar en la lista hasta que se encuentre un espacio.

▬▬▬ 7. Colisiones con Closed Adressing.

De nuevo se tienen los siguientes datos:

datos:[
["Mia", 80],
["Tim",100],
["Bea", 70],
["Zoe", 70],
["Sue", 70],

["Len", 85],
["Moe", 90],
["Lou", 70],
["Rae", 80],
["Max", 95],
["Tod", 95]
];

>>> Iniciemos el proceso:

Iniciamos con un vector vacío.

01[]
02[]
03[]
04[]
05[]
06[]
07[]
08[]
09[]
10[]
11[]

"Mia" = 279, mod(279,11)= 4
"Tim" = 298, mod(298,11)= 1
"Bea" = 264, mod(264,11)= 0 (posición 11!)
"Zoe" = 302, mod(302,11)= 5
"Sue" = 301, mod(301,11)= 4 Colisión!!!

```
01[["Tim",100]]
02[]
03[]
04[["Mia", 80],["Sue",70]]
05[["Zoe", 70]]
06[]
07[]
08[]
09[]
10[]
11[["Bea", 70]]
```

```
>>> Seguimos con:

"Len" = 287, mod(287,11)= 1

01[["Tim",100],["Len", 85]]
02[]
03[]
04[["Mia", 80],["Sue", 70]]
05[["Zoe", 70]]
06[]
07[]
08[]
09[]
10[]
11[["Bea", 70]]
```

```
>>> Seguimos con:

"Moe" = 289, mod(289,11) = 3

01[["Tim",100],["Len", 85]]
02[]
03[[Moe”, 90]]
04[[“Mia”, 80],[“Sue”, 70]]
05[[“Zoe”, 70]]
06[]
07[]
08[]
09[]
10[]
11[[“Bea”, 70]]
```

```
>>> Seguimos con:

“Lou” = 304, mod(304,11)= 7

01[[“Tim”,100],[“Len”, 85]]
02[]
03[["Moe", 90]]
04[["Mia", 80],[“Sue”, 70]]
05[["Zoe", 70]]
06[]
07[["Lou", 70]]
08[]
09[]
10[]
11[["Bea", 70]]
```

>>> Seguimos con:

"Rae" = 280, mod(280,11)= 5

```
01[["Tim",100],["Len", 85]]
02[]
03[["Moe", 90]]
04[["Mia", 80],["Sue", 70]]
05[["Zoe", 70],["Rae", 80]]
06[]
07[["Lou", 70]]
08[]
09[]
10[]
11[["Bea", 70]]
```

>>> Seguimos con:

"Max" = 294, mod(294,11)= 8

```
01[["Tim",100].["Len", 85]]
02[]
03[["Moe", 90]]
04[["Mia", 80],["Sue", 70]]
05[["Zoe", 70],["Rae", 80]]
06[]
07[["Lou", 70]]
08[["Max", 95]]
09[]
10[]
11[["Bea", 70]]
```

>>> Seguimos con:

"Tod" = 295, mod(295,11)= 9

```
01[["Tim",100],["Len", 85]]
02[]
03[["Moe", 90]]
04[["Mia", 80],["Sue", 70]]
05[["Zoe", 70],["Rae", 80]]
06[]
07[["Lou", 70]]
08[["Max", 95]]
09[["Tod", 95]]
10[]
11[["Bea", 70]]
```

Y así terminamos de construir la estructura.

Si deseamos encontrar a alguien, por ejemplo a Rae:

```
find("Rae");
```

Rae = 280, mod(280,11)= 5

Y buscamos linealmente en la posición 5.

```
  01[["Tim",100],["Len", 85]]
  02[]
  03[["Moe", 90]]
  04[["Mia", 80],["Sue", 70]]
▶ 05[["Zoe", 70],["Rae", 80]]
  06[]
  07[["Lou", 70]]
  08[["Max", 95]]
  09[["Tod", 95]]
  10[]
  11[["Bea", 70]]
```

Podemos ver que con este método hay un mayor número de elementos que se encuentran en la posición correcta. Así que el tiempo de búsqueda es mejor que con el método anterior.

El peor tiempo de este método es O(n), sin embargo su tiempo promedio es mucho menor.

▬▬▬ 8. Características de un Hash Table.

Cuando se construye una función de “hashing” es deseable que tenga las siguientes características:

- Mínimo número de colisiones.
- Distribuir los índices de la manera más uniforme posible.
- Sencilla de calcular.

▬▬▬ 9. El Factor de Carga.

Hemos visto que en el peor de los casos las tablas de hash tienen un tiempo de O(n). Sin embargo en general su tiempo promedio se puede acercar mucho a O(1) si se evitan las colisiones. Para evitar las colisiones se necesitan dos cosas.

- Una buena función de hash.
  Ya hemos hablado de este tema.
- Un factor de carga adecuado.
  Vamos a referirnos a este tema.

●● Factor de Carga.

El factor de carga se puede usar tanto con el “Open Hash” como con el “Close Hash”. El factor de carga de una tabla de hash se calcula de la siguiente manera:

$$\text{factor\_de\_carga} = \frac{\text{Espacios Ocupados}}{\text{}}$$

Espacios Disponibles

Por ejemplo si se tiene el siguiente arreglo:

01["Tim",100]
02[]
03[]
04["Mia", 80]
05[]

Se tiene un factor de carga de 2/5.

2 elementos de 5 campos disponibles.

El factor de carga mide la proporción de espacios ocupados con respecto al total de espacios.

En general cuando el factor de carga empieza a crecer, se necesitan agregar más espacios a la tabla de hash. Se debe cambiar tamaño del arreglo.

Se tienen los siguientes elementos:

"Mia" = 279
"Tim" = 298
"Zoe" = 302
"Sue" = 301

Se colocan los elementos se ha utilizado una función de hash con módulo 5.

01["Sue", 70]
02["Zoe", 70]
03["Tim",100]
04["Mia", 80]
05[]

Su factor de carga es de 4/5 = 0.80

Es usual que cuando el factor de carga supera el valor 0.70
entonces se duplica el tamaño del arreglo.

Primero se cambia el tamaño del arreglo, como una regla usual se duplica el tamaño.

01[]
02[]
03[]
04[]
05[]
06[]
07[]
08[]
09[]
10[]

Se vuelven a ubicar los elementos que se tienen, pues ahora se utiliza un función de hash con un módulo de 10.

"Mia" = 279
"Tim" = 298
"Zoe" = 302
“Sue” = 301

01["Sue", 70]
02["Zoe", 70]
03[]
04[]
05[]
06[]
07[]
08["Tim",100]
09["Mia", 80]
10[]

Y se tiene un nuevo factor de carga de 4/10 = 0.40.

Al tener un factor de carga menor, se tendrán menos colisiones y el tiempo que se utiliza será menor.

Por supuesto que el proceso de cambiar el tamaño de un vector también toma tiempo. No vamos a analizar esta situación, pero aún con este proceso, el tiempo promedio de las tablas de hash es bastante cercano a O(1).

▬▬▬ 10. Tablas de Hash en Algunos Lenguajes.

Muchos lenguajes de programación tienen una implementación con hash tables. A continuación se muestra un ejemplo en el lenguaje Python. Esta proceso lleva el nombre de diccionario.

```
>>> tabla = dict()

>>> tabla["Mia"] = 80

>>> tabla["Tim"] = 100

>>> tabla["Bea"] = 70

>>> print tabla
{"Mia":80, "Tim":100, "Bea":70}

>>> print tabla[“Tim”]
100
```

En el caso de Wxmaxima, se pueden usar hash tables de la siguiente forma:

```
(%i) tabla["Mia"]:80;
(%o) 80

(%i) tabla["Tim"]:100;
(%o) 100

(%i) tabla["Bea"]:70;
```

```
(%o) 70

(%i) arrayinfo(tabla);
(%o) [hashed,1,["Bea"],["Mia"],["Tim"]]

(%i) tabla["Tim"];
(%o) 100
```

▬▬▬ 11. Resumen.

• Las hash tables se utilizan para indexar grandes cantidades de datos.

• Se construye el valor del subíndice utilizando la llave misma.

• Las colisiones se resuelven mediante open adressing o close adressing.

• Estas técnicas de hashing se utilizan mucho en bases de datos, compiladores, caching de memoria en redes, autenticación de passwords y mucho más.

• La meta principal es que la insercción, borrado y recuperación de la información ocurra en un tiempo lo más cercano posible a O(1).

• Si desea seguir investigando de este tema, puede buscar información sobre la función SHA.

▬▬▬ 12. Ejercicios.

●● Ejercicio 1.

Es importante que existe consistencia en una función de hash. Esto significa que para una entrada siempre debe devolver el mismo valor. Cuáles de estas funciones son consistentes?

f(x) = 1

f(x) = rand()

f(x) = next_empty_slot()

f(x) = length(x)

●● Ejercicio 2.

Utilice los siguientes nombres:

“Tim”
“Bob”
“Ana”
“Yud”
“Bab”

Si se utiliza un arreglo con 5 campos, cuáles de las siguientes funciones de hash funcionan mejor.

- f(x) = 1

- f(x) = length(x), largo del string.

- Se utiliza el primer caracter del string como él indice.
  Se toma el valor de ascii del primer caracter del string.

- Se asocia cada letra del alfabeto sin diferenciar mayúsculas y minúsculas.
  a=2, b=3, c=5, d=7
  Se toma el string y se convierte a estos valores.
  “Bag” = 3 + 2 + 17 = 22
   Y con el valor 22 se coloca en el arreglo.