Análisis y Diseño de Algoritmos

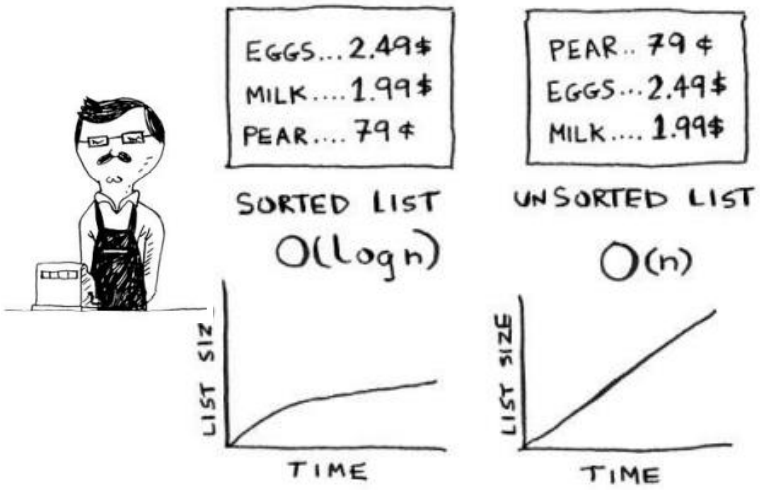
Sesión 6

Logro de la sesión

Al finalizar la sesión, el estudiante realiza un análisis y desarrollo de algoritmos de dispersión vinculados a las tablas hash, utilizando un lenguaje de programación

Tablas Hash

Supongamos que trabajas en el supermercado. Cuando un cliente compra un producto, tienes que buscar el precio en un libro. Si el libro no está ordenado alfabéticamente, puede llevarle mucho tiempo buscar "manzana" en cada línea. Estaría haciendo una búsqueda simple, donde tiene que mirar cada línea. ¿Recuerdas cuánto tiempo tomaría eso? O(n). Si el libro está ordenado alfabéticamente, puede ejecutar una búsqueda binaria para encontrar el precio de una manzana. Sólo tomaría O(log n) operaciones



# OF ITEMS IN THE BOOK	O(n)	O(log n)
100	10 sec	1 sec ← YOU NEED TO CHECK $\log_2 100 = 7$ LINES
1000	1.66 min	1 sec ← NEED TO CHECK $\log_2 1000 = 10$ LINES
10000	16.6 min	2 sec ← $\log_2 10000 = 14$ LINES = 2 sec

Como recordatorio, ¡hay una gran diferencia entre el tiempo O(n) y O(log n)! Suponga que puede mirar 10 líneas del libro por segundo. Esto es lo que le llevaría la búsqueda simple y la búsqueda binaria.

Como cajero, buscar cosas en un libro es una molestia, incluso si el libro está ordenado. Puedes notar a los clientes molestándose de a pocos a medida que buscas artículos en el libro. Lo que realmente necesitas es una asistente que tenga todos los nombres y precios memorizados. Entonces no necesitas buscar nada: le preguntas y ella te dice la respuesta al instante.

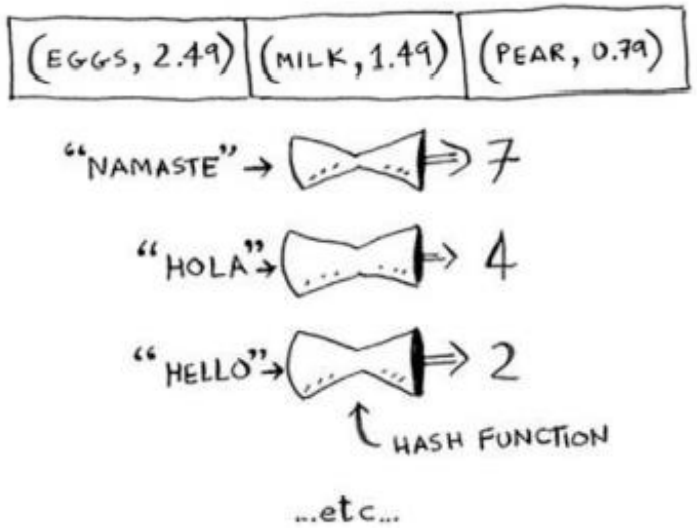
Tu amiga Maggie puede darte el precio de un producto en tiempo O(1) para cualquier artículo, sin importar qué tan grande sea el libro. Es incluso más rápida que la búsqueda binaria.

# OF ITEMS IN THE BOOK	SIMPLE SEARCH O(n)	BINARY SEARCH O(log n)	MAGGIE O(1)
100	10 sec	1 sec	INSTANT
1000	1.6 min	1 sec	INSTANT
10000	16.6 min	2 sec	INSTANT

Tablas Hash

¡Qué maravillosa persona! ¿Cómo se obtiene un "Maggie"? Podría implementar este libro como un arreglo. Cada elemento del arreglo contiene en realidad 2 elementos: uno es el nombre de un tipo de producto y el otro es el precio. Si ordena esta matriz por nombre, puede ejecutar una búsqueda binaria para encontrar el precio de un artículo. Entonces puede encontrar elementos en tiempo O (log n). Pero desea encontrar elementos en tiempo O (1). Es decir, quieres hacer una "Maggie". Ahí es donde entran las funciones hash.

Requerimientos para las funciones hash:



Una función hash es una función a la cual le pasas un tipo de dato (típicamente un string) y obtienes de vuelta un número

1. Deben ser consistentes, si le pasas manzana y obtienes 4, siempre que le pases manzana debes obtener 4

2. Debe asociar entradas diferentes a números diferentes, una función hash no nos sirve si siempre devuelve 1 por cada entrada que le pasas

- Suponga que la función hash para una cadena suma el valor Unicode de cada carácter.

```
public int hashcode(String s) {
    int result = 0;
    for (int i = 0; i < s.length(); i++)
        result += s.charAt(i);
    return result;
}
```

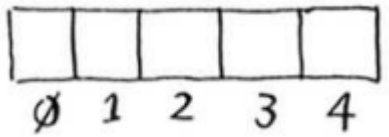
¿Código hash para "DAB" y “BAD"?

A. 301 103
B. 4 4
C. 412 214
D. 5 5
E. 199 199

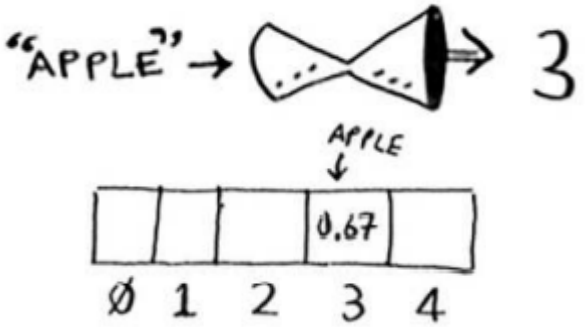
Tablas Hash

Entonces, una función hash asigna entradas a números. ¿Para qué sirve eso? Bueno, ¡puedes usarlo para hacer tu "Maggie"!

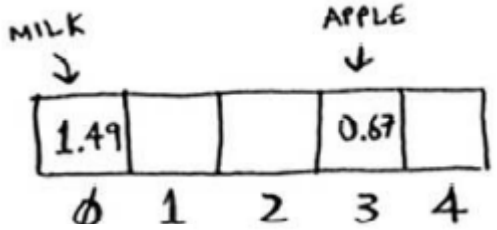
Comience con un arreglo vacío, almacenarás todos tus precios en este arreglo



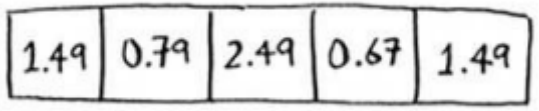
Agreguemos el precio de una manzana. Introduzca "applea" en la función hash La función hash genera "3". Así que guardemos el precio de una manzana en el índice 3 en la matriz.



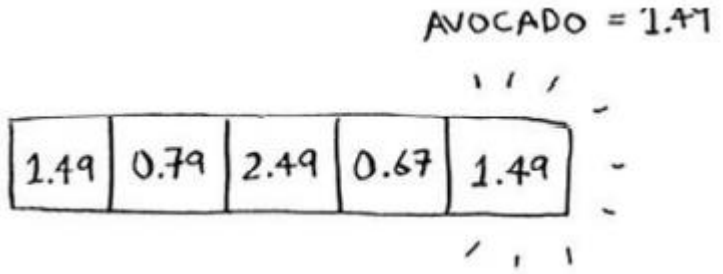
Agreguemos la leche. Introduzca "milk" en la función hash. La función hash dice "0". Guardemos el precio de la leche en el índice 0.



Continúe, y eventualmente todo el arreglo estará lleno de precios.



Ahora preguntas: "Oye, ¿cuál es el precio de una palta?" no necesitas buscarlo en la matriz. Solo introduce "avocado" en la función hash. Te dice que el precio está almacenado en el índice 4. Y efectivamente, allí está



¿Qué es una Tabla Hash?

- Estructura que contiene **valores**
- Puedo hallar un valor a partir de una **clave**

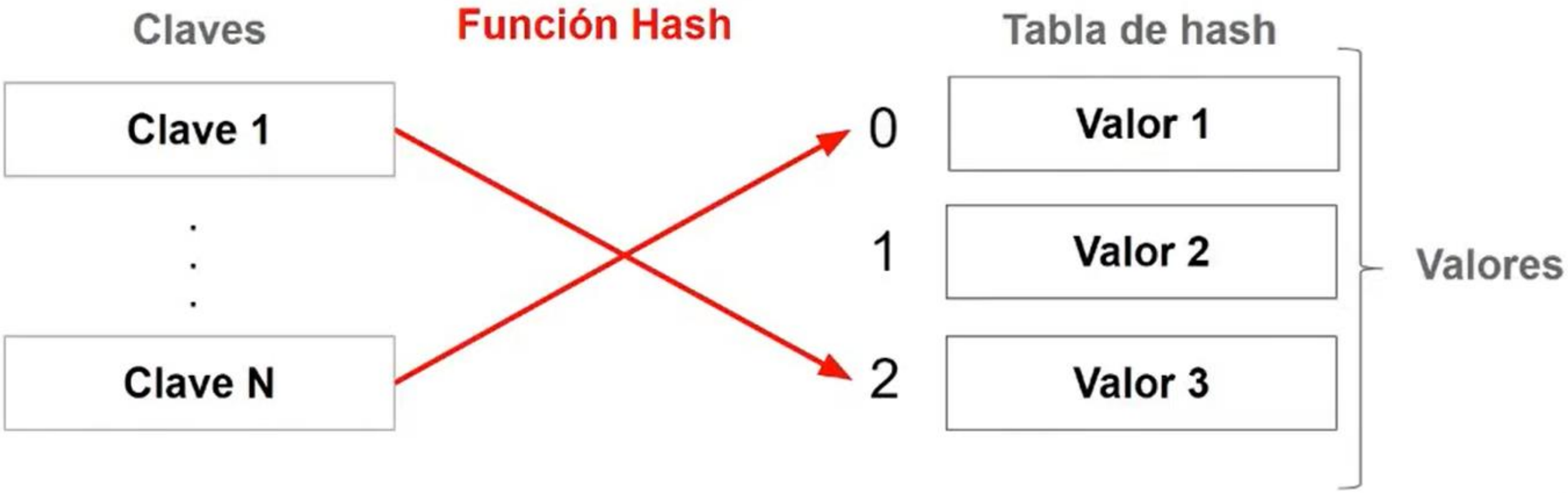


Tabla Hash: Almacenar objetos

Un proceso mas general…

El proceso de almacenar objetos usando una función hash es el siguiente:

1. Cree un arreglo de tamaño M para almacenar objetos, este arreglo se llama Hash-Table.

2. Encuentre un código hash de un objeto pasándolo a través de la función hash.

3. Tome el módulo de código hash por el tamaño de Hash-Table para obtener el índice de la tabla donde se almacenarán los objetos.

4. Finalmente, almacene estos objetos en el índice designado.

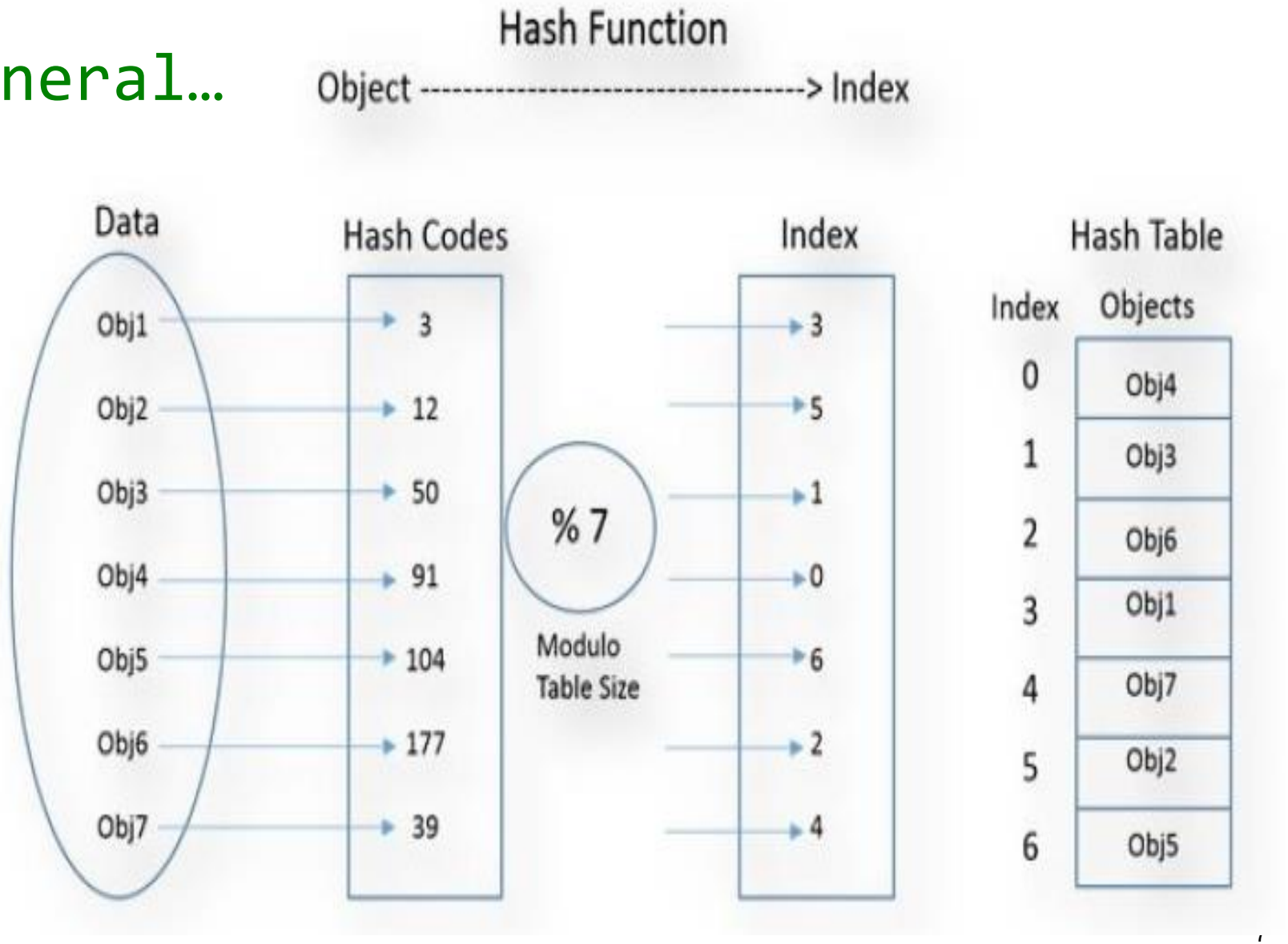


Tabla Hash: Búsqueda de objetos

Un proceso mas general…

El proceso de búsqueda de objetos en una Hash-Table usando una función hash es el siguiente:

1. Encuentre un código hash del objeto que estamos buscando pasándolo a través de la función hash.
2. Tome el módulo de código hash por el tamaño de la tabla hash para obtener el índice de la tabla donde se almacenan los objetos.
3. Finalmente, recupere el objeto del índice designado.

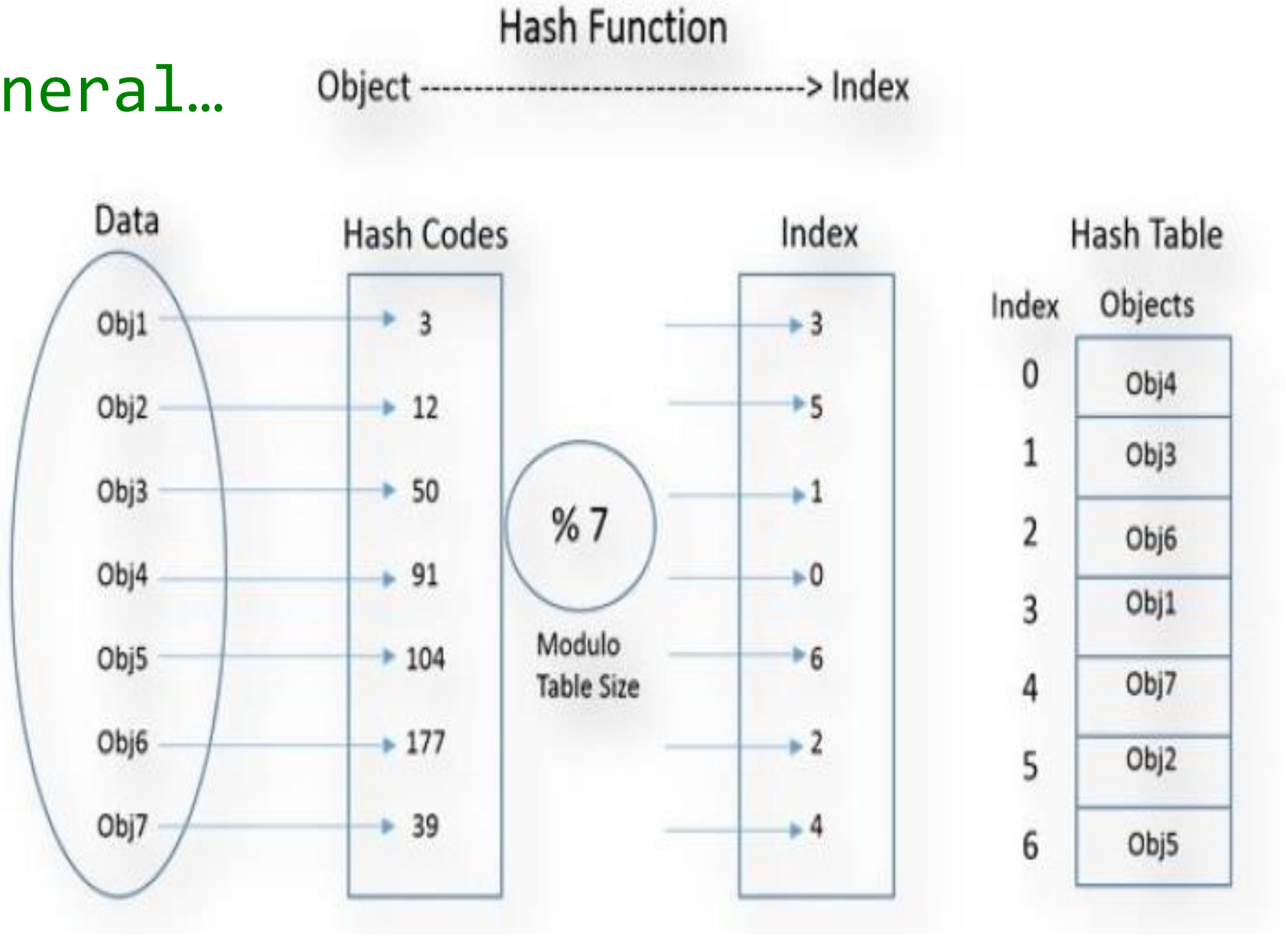
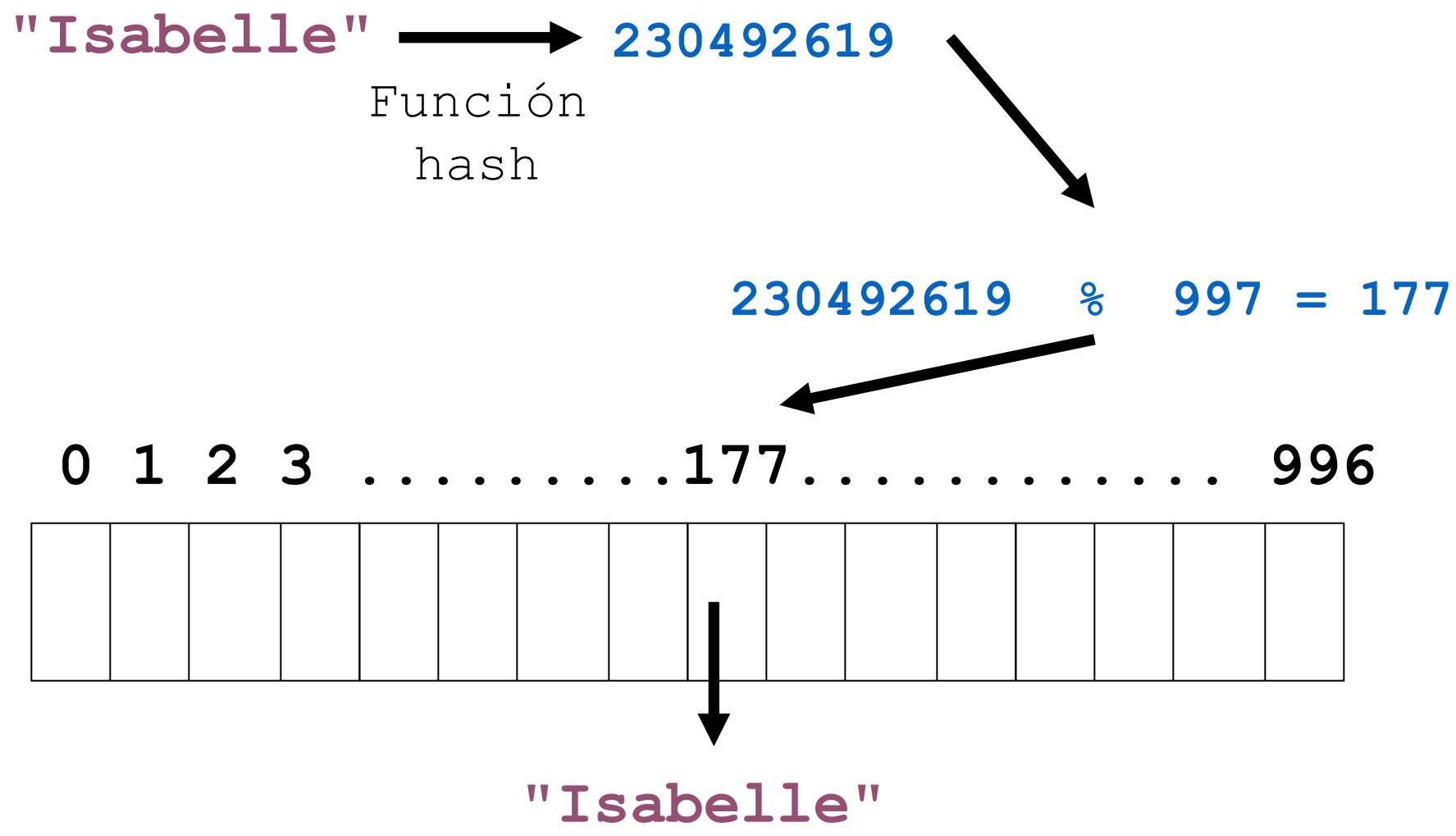
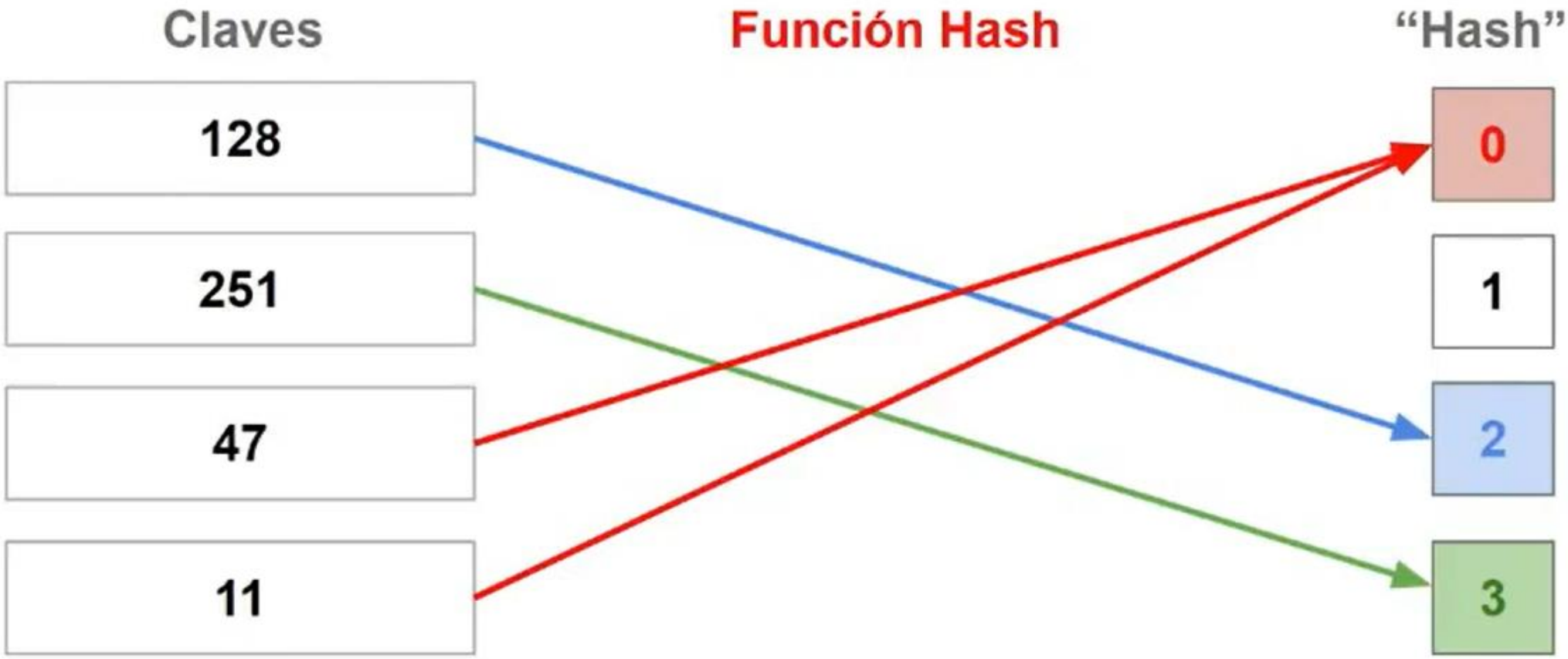


Tabla Hash: Almacenar y buscar objetos



¿Qué es una función Hash?

- Es una **función** que transforma claves en un número asociado



Método hashCode de la clase Java String

```
public int hashCode() {
    int h = hash;
    if (h == 0 && value.length > 0) {
        char[] val = value;
        for (int i = 0; i < value.length; i++) {
            h = 31 * h + val[i];
        }
        hash = h;
    }
    return h;
}
```

key	hashCode()
"Aa"	2112
"BB"	2112

key	hashCode()
"AaAaAaAa"	-540425984
"AaAaAaBB"	-540425984
"AaAaBBAa"	-540425984
"AaAaBBBB"	-540425984
"AaBBAaAa"	-540425984
"AaBBAaBB"	-540425984
"AaBBBBAa"	-540425984
"AaBBBBBB"	-540425984

key	hashCode()
"BBAaAaAa"	-540425984
"BBAaAaBB"	-540425984
"BBAaBBAa"	-540425984
"BBAaBBBB"	-540425984
"BBBBAaAa"	-540425984
"BBBBAaBB"	-540425984
"BBBBBBAa"	-540425984
"BBBBBBBB"	-540425984

2^N strings of length 2N that hash to same value!

Tabla Hash: Colisiones

Ejemplo simple

Supongamos que estamos usando nombres como nuestra clave y nuestra función hash es la siguiente:

- Toma la tercera letra del nombre, toma el valor int de la letra (a = 0, b = 1, ...), divide entre 6 y toma el resto
- ¿Qué numero esconde el nombre “Bellers”?
  - ✓ l -> 11 -> 11% 6 = 5

- Mike = (10% 6) = 4
- Kelly = (11% 6) = 5
- Olivia = (8% 6) = 2
- Isabelle = (0% 6) = 0
- David = (21% 6) = 3
- Margaret = (17% 6) = 5 (uh oh)
- Wendy = (13% 6) = 1

- Esta es una función hash imperfecta. Una función hash perfecta produce una asignación uno a uno de las claves a los valores hash. En este caso para Kelly y Margaret se genera una “colisión”
- ¿Cuál es el número máximo de valores que esta función puede combinar perfectamente?

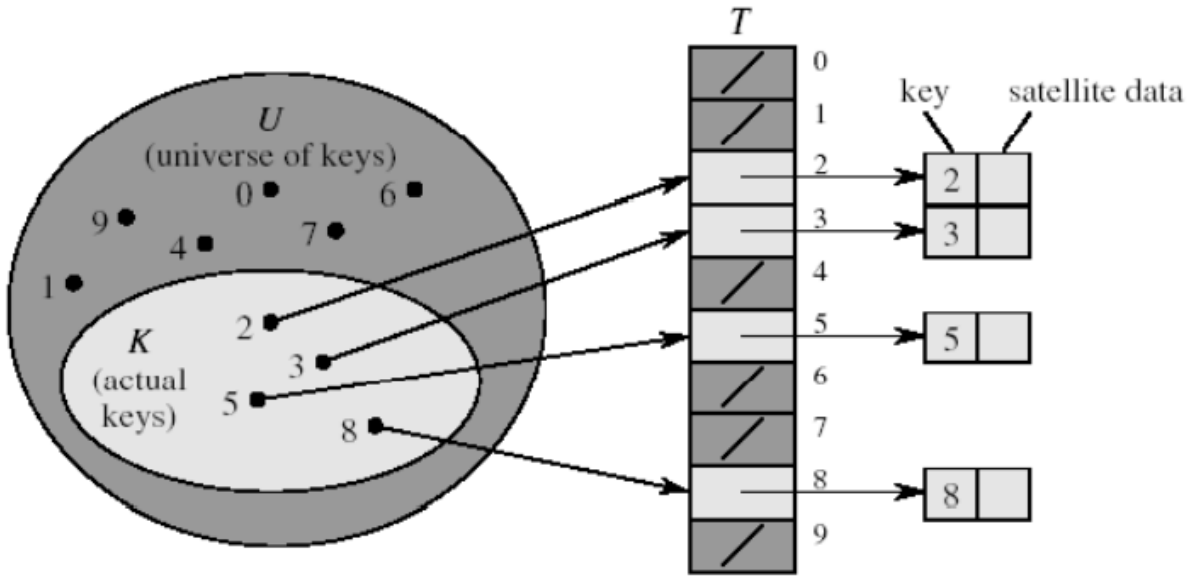
Tipos de Hash: Gestión de Colisiones

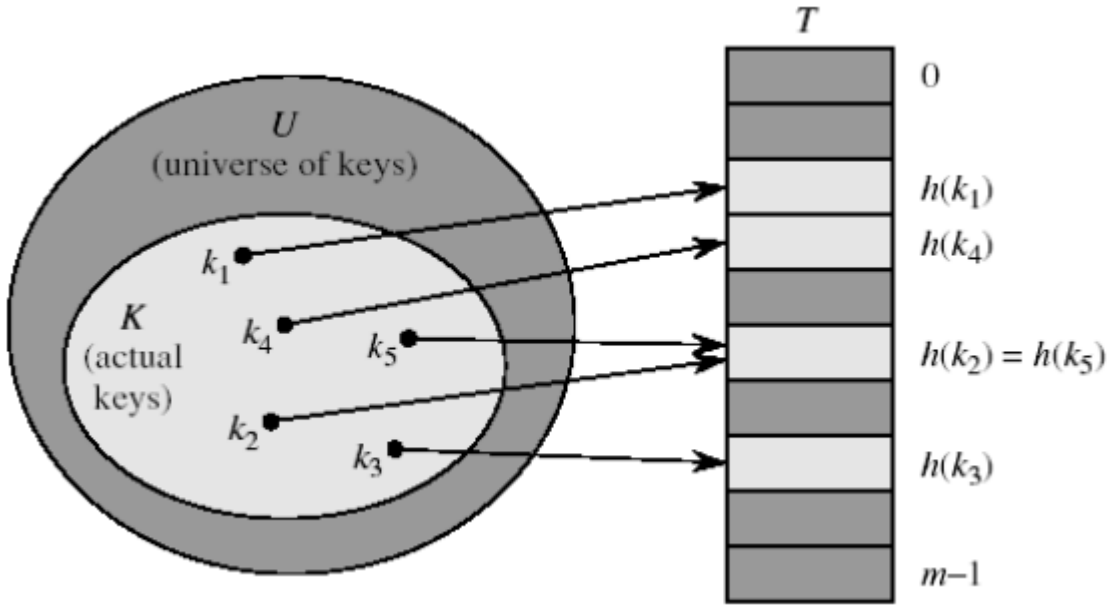
Direccionamiento Cerrado

Hash Abierto

Direccionamiento Abierto

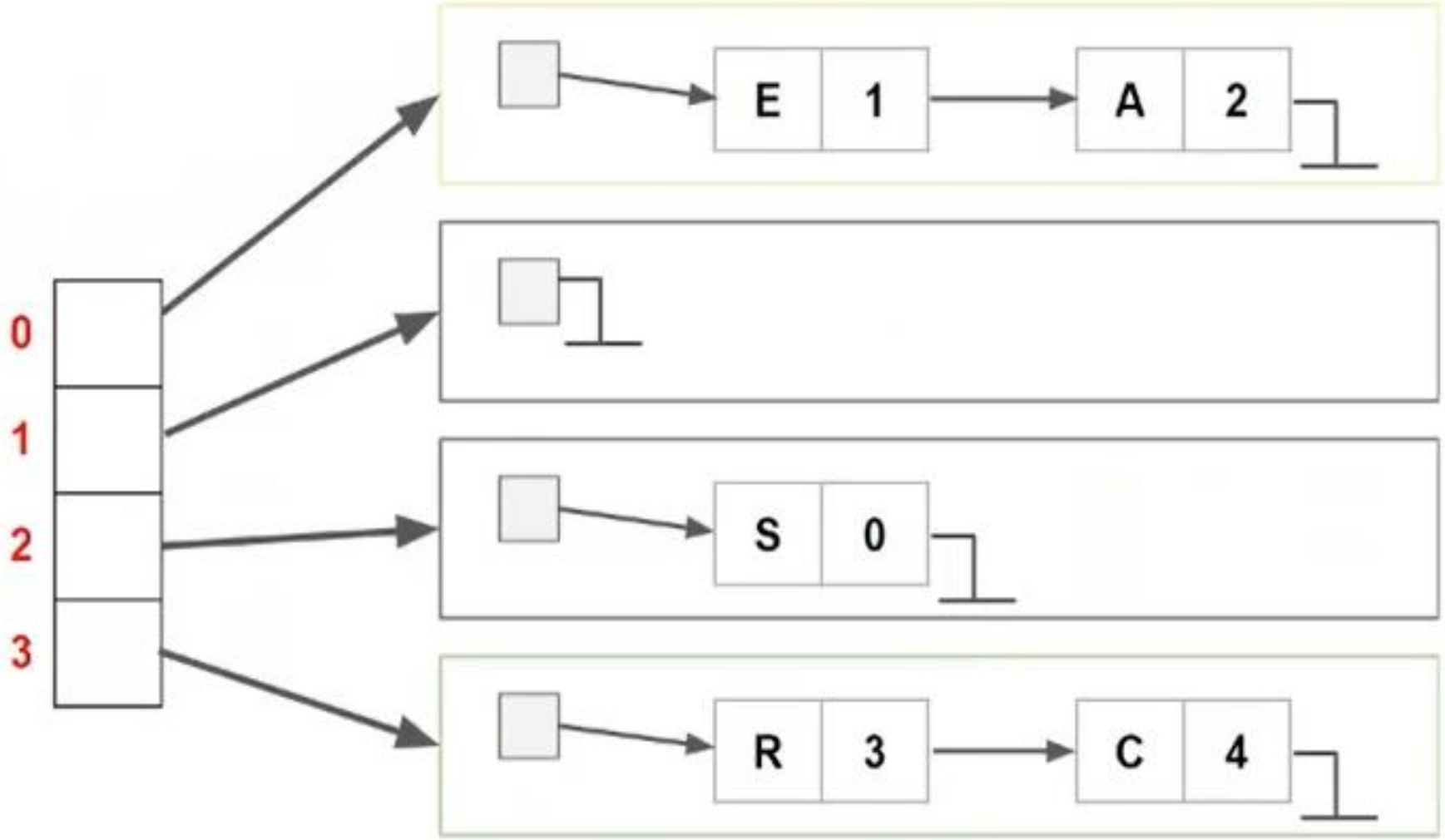
Hash Cerrado





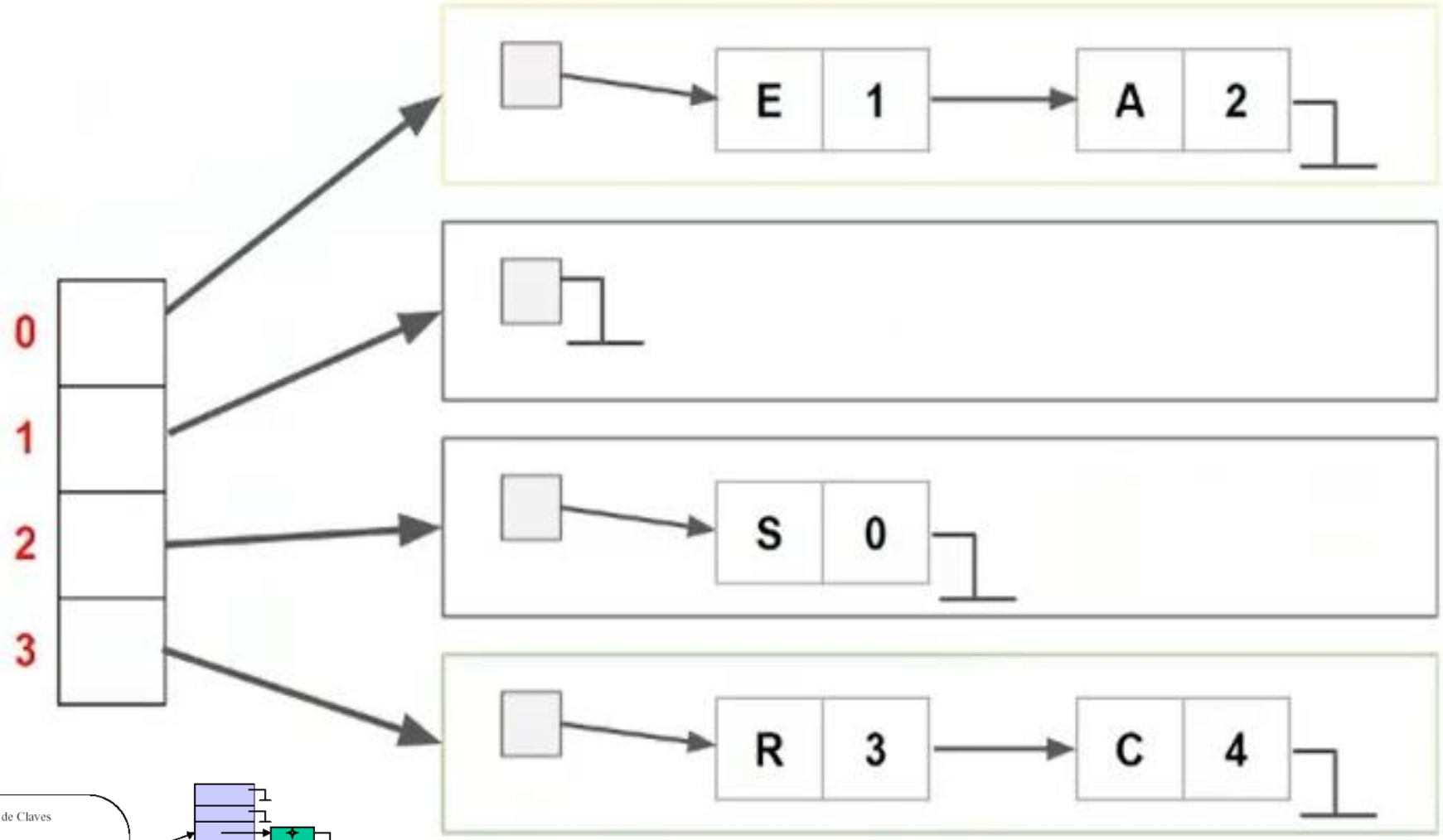
Tipos de Hash: Abierto con Encadenamiento o Chaining

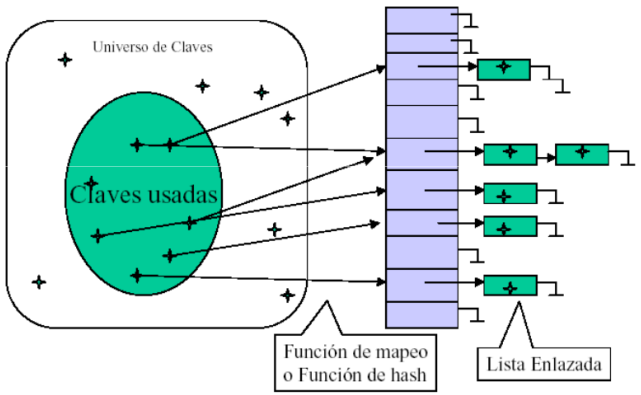
Clave	Hash	Valor
S	2	0
E	0	1
A	0	2
R	3	3
C	3	4



Tipos de Hash: Abierto con Encadenamiento o Chaining

Clave	Hash	Valor
S	2	0
E	0	1
A	0	2
R	3	3
C	3	4





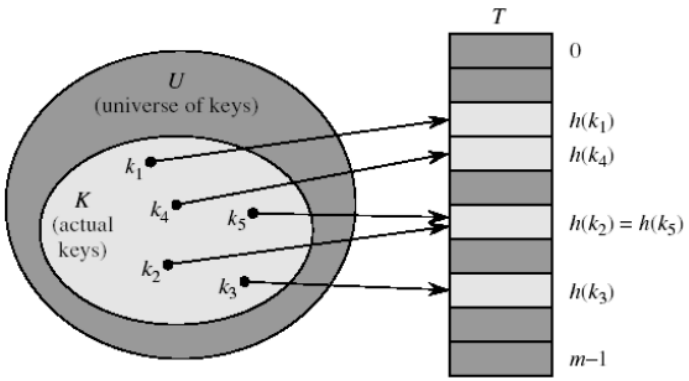
- Para encontrar una clave con su correspondiente valor en caso de colisión:
  - O(n)
    donde n es la cantidad de elementos que colisionaron
  - Es decir, voy a tener que recorrer la lista enlazada

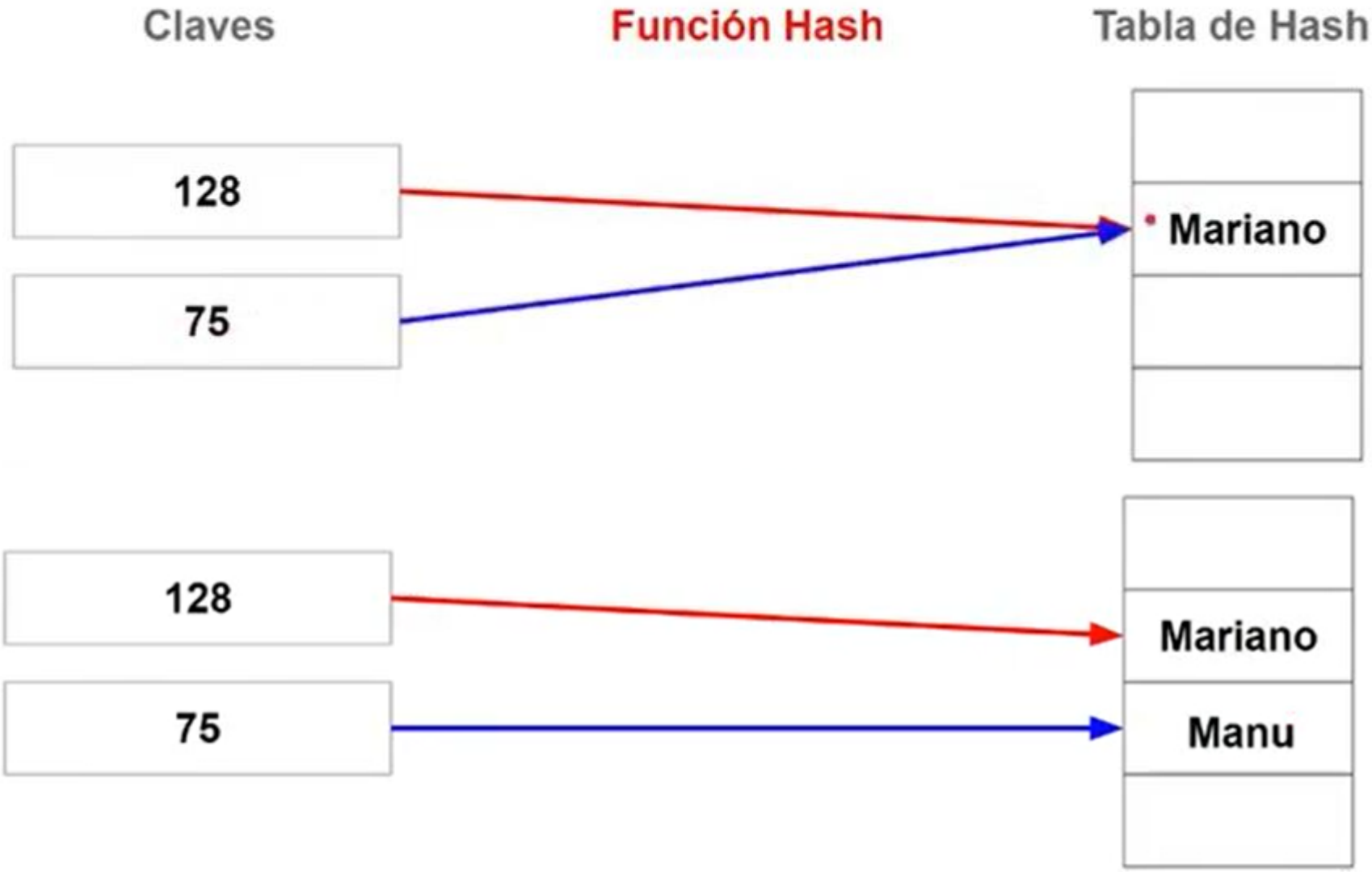
Tipos de Hash: Abierto con Encadenamiento

```java
public class HashTableSC {
private int tableSize;
Node[] listArray;
private class Node {
private int value;
private Node next;
public Node (int v, Node n) {
value = v;
next = n;}
};
public HashTableSC () {
tableSize = 512;
listArray = new Node[tableSize];
for (int i = 0; i < tableSize; i++) {
listArray[i] = null; }
}
private int computeHash (int key) // division method
{
int hashValue = key;
return hashValue % tableSize;
}
public void add (int value) {
int index = computeHash (value);
listArray[index] = new Node (value, listArray[index]);
}
public boolean remove (int value) {
int index = computeHash (value);
Node nextNode, head = listArray[index];
if (head != null && head.value == value) {
listArray[index] = head.next;
return true;
}
while (head != null) {
nextNode = head.next;
if (nextNode != null && nextNode.value == value) {
head.next = nextNode.next;
return true;
} else {
head = nextNode; }
}
return false;
}
public void print () {
for (int i = 0; i < tableSize; i++) {
System.out. println ("printing for index value :: " + i + "List of value printing :: ");
Node head = listArray[i];
while (head != null) {
System.out. println (head.value);
head = head.next; }
}
}
public boolean find (int value) {
int index = computeHash (value);
Node head = listArray[index];
while (head != null) {
if (head.value == value) {
return true; }
head = head.next;
}
return false;
}
public static void main (String[] args) {
HashTableSC ht = new HashTableSC ();
for (int i = 100; i < 110; i++) {
ht. add (i);
}
System.out. println ("search 100 :: " + ht. find (100));
System.out. println ("remove 100 :: " + ht. remove (100));
System.out. println ("search 100 :: " + ht. find (100));
System.out. println ("remove 100 :: " + ht. remove (100));
}
}
```

Tipos de Hash: Cerrado

- Todos los valores se guardan dentro de la misma tabla
- Tamaño de tabla >= nro. de claves





- Si hay colisión, sigo recorriendo el array hasta encontrar el próximo espacio libre
- Por esto se conoce como “direccionamiento abierto”

Tipos de Hash
Cerrado - Tipos de métodos de búsqueda

Sondeo lineal

En el sondeo lineal (Linear Probing), tratamos de resolver la colisión de un índice de una tabla hash buscando secuencialmente la ubicación libre de la tabla hash. Supongamos, si k es el índice recuperado de la función hash. Si el índice k-ésimo ya está lleno, buscaremos (k+1) %M, luego (k+2) %M y así sucesivamente. Cuando obtengamos un espacio libre, insertaremos los datos en ese espacio libre.

Sondeo cuadrático

En el sondeo cuadrático (Quadratic Probing), tratamos de resolver la colisión del índice de una tabla Hash aumentando cuadráticamente el índice de búsqueda de ubicación libre. Supongamos, si k es el índice recuperado de la función hash. Si el k-ésimo índice ya está lleno, buscaremos $(k+1)^2$ %M, luego $(k+2)^2$ %M y así sucesivamente. Cuando obtengamos un espacio libre, insertaremos los datos en ese espacio libre.

Hash doble

Aplicar una segunda función de hash a la clave cuando hay colisión

Factor de carga

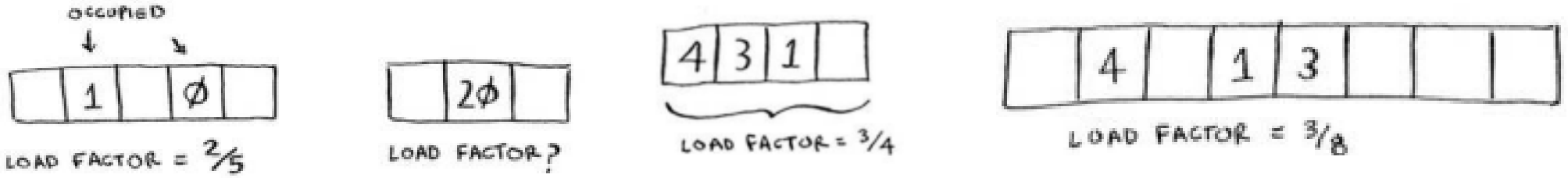
El factor de carga de una tabla hash es fácil de calcular

$$\alpha = n / m$$

n: número de claves almacenadas actualmente
m: capacidad de la tabla de hash

Indica grado de ocupación de la tabla hash y que tan probable es que haya colisión

Entre 0 y 1



Factor de carga

Factor de carga = (Número máximo permitido de elementos en la tabla hash) / Tamaño de la tabla hash

Según la definición anterior, el factor de carga es una medida de cuántos elementos puede contener la tabla hash antes de aumentar su capacidad. Cuando el número de elementos excede el factor de carga y la capacidad actual, la tabla hash se vuelve a procesar (rehashed). Normalmente, el tamaño de la tabla hash se duplica en este caso.

Por ejemplo, si el factor de carga es 0,8 y la capacidad de la tabla hash es 1000, entonces podemos agregar 800 elementos a la tabla Hash sin volver a hacer el hash (rehashing).

Rehash

- Cuando α >= 0.75, es hora de rehashear
- La capacidad de la tabla de hash debería aumentar

Ejemplo: m = 20 (capacidad)

$$\alpha = n / m$$
$$0.75 = n / 20$$
$$n = 15$$

Cuando almacene la clave 15, voy a tener que aumentar la capacidad de la tabla de hash (podría duplicarla, por ejemplo)

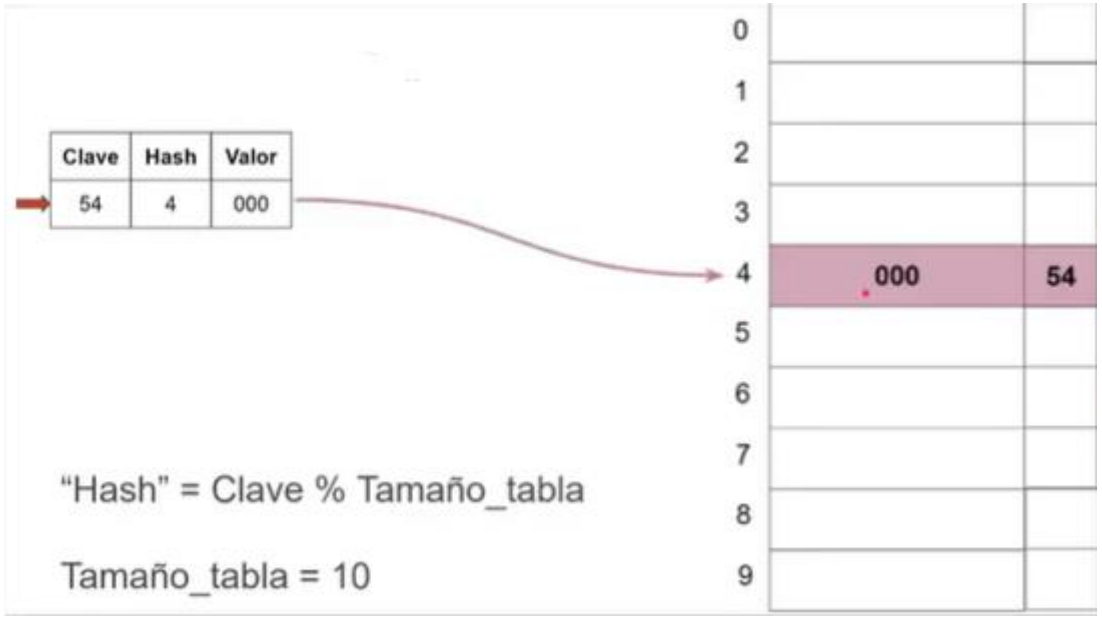
Operaciones Tabla Hash: Insertar

Clave	Hash	Valor
54	4	000

0		
1		
2		
3		
4		
5		
6		
7		
8		
9		

"Hash" = Clave % Tamaño_tabla

Tamaño_tabla = 10



Clave	Hash	Valor
54	4	000
26	6	111
70	0	222
31	1	333
93	3	444

0	**222**	**70**
1	**333**	**31**
2		
3	**444**	**93**
4	**000**	**54**
5		
6	**111**	**26**
7		
8		
9		

"Hash" = Clave % Tamaño_tabla

Tamaño_tabla = 10

Operaciones Tabla Hash: Insertar con colisión

Clave	Hash	Valor
54	4	000
26	6	111
70	0	222
31	1	333
93	3	444
40	**0**	**555**

Método lineal

P = (P + 1) % Tamaño_tabla
P = (0 + 1) % 10 = 1

0	**222**	**70**
1	**333**	**31**
2		
3	**444**	**93**
4	**000**	**54**
5		
6	**111**	**26**
7		
8		
9		

Clave	Hash	Valor
54	4	000
26	6	111
70	0	222
31	1	333
93	3	444
40	0	555

P = (P + 1) % Tamaño_tabla
P = (1 + 1) % 10 = 2

0	**222**	**70**
1	**333**	**31**
2		
3	**444**	**93**
4	**000**	**54**
5		
6	**111**	**26**
7		
8		
9		23

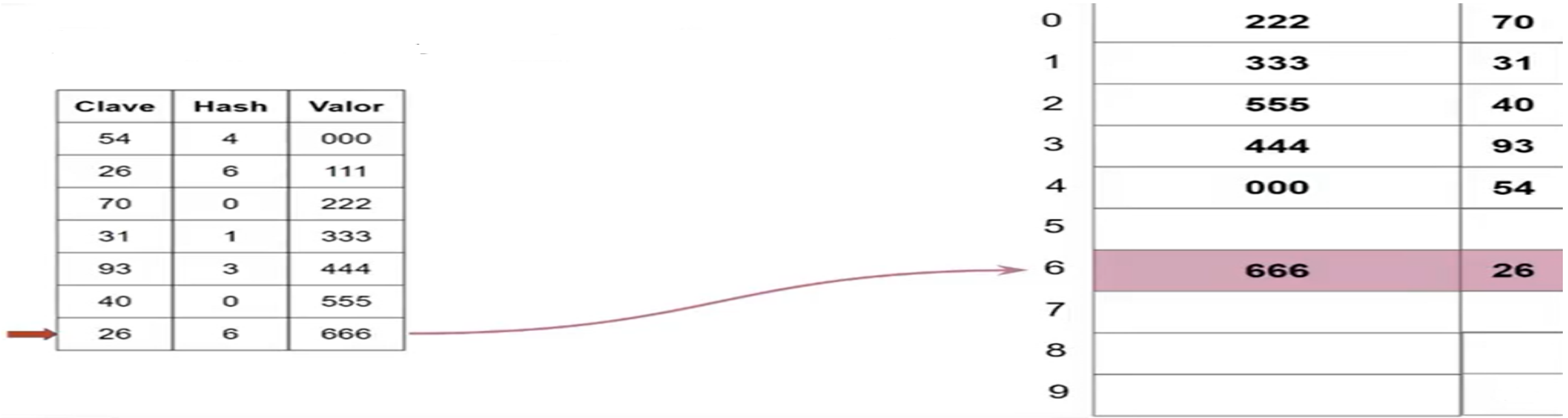
Clave	Hash	Valor
54	4	000
26	6	111
70	0	222
31	1	333
93	3	444
40	0	555

0	**222**	**70**
1	**333**	**31**
2	**555**	**40**
3	**444**	**93**
4	**000**	**54**
5		
6	**111**	**26**
7		
8		
9		

Operaciones Tabla Hash: Insertar existente

Clave	Hash	Valor
54	4	000
26	6	111
70	0	222
31	1	333
93	3	444
40	0	555
26	**6**	**666**

0	222	70
1	333	31
2	555	40
3	444	93
4	000	54
5		
6	111	26
7		
8		
9		



Tipos de Hash: Cerrado

Índice	Valor
0	
1	
2	
3	
4	
5	
6	
7	

20	33	7	10	12	14	56	100

Operaciones Tabla Hash: Rehash

Clave	Hash	Valor
54	4	000
26	6	111
70	0	222
31	1	333
93	3	444
40	0	555
26	6	666
7	7	777
29	**9**	**888**

¿Cuál es el **factor de carga** si inserto la nueva clave?

$$\alpha = n / m$$

n: 8

m: 10

¡Aumento el tamaño de la tabla de hash!

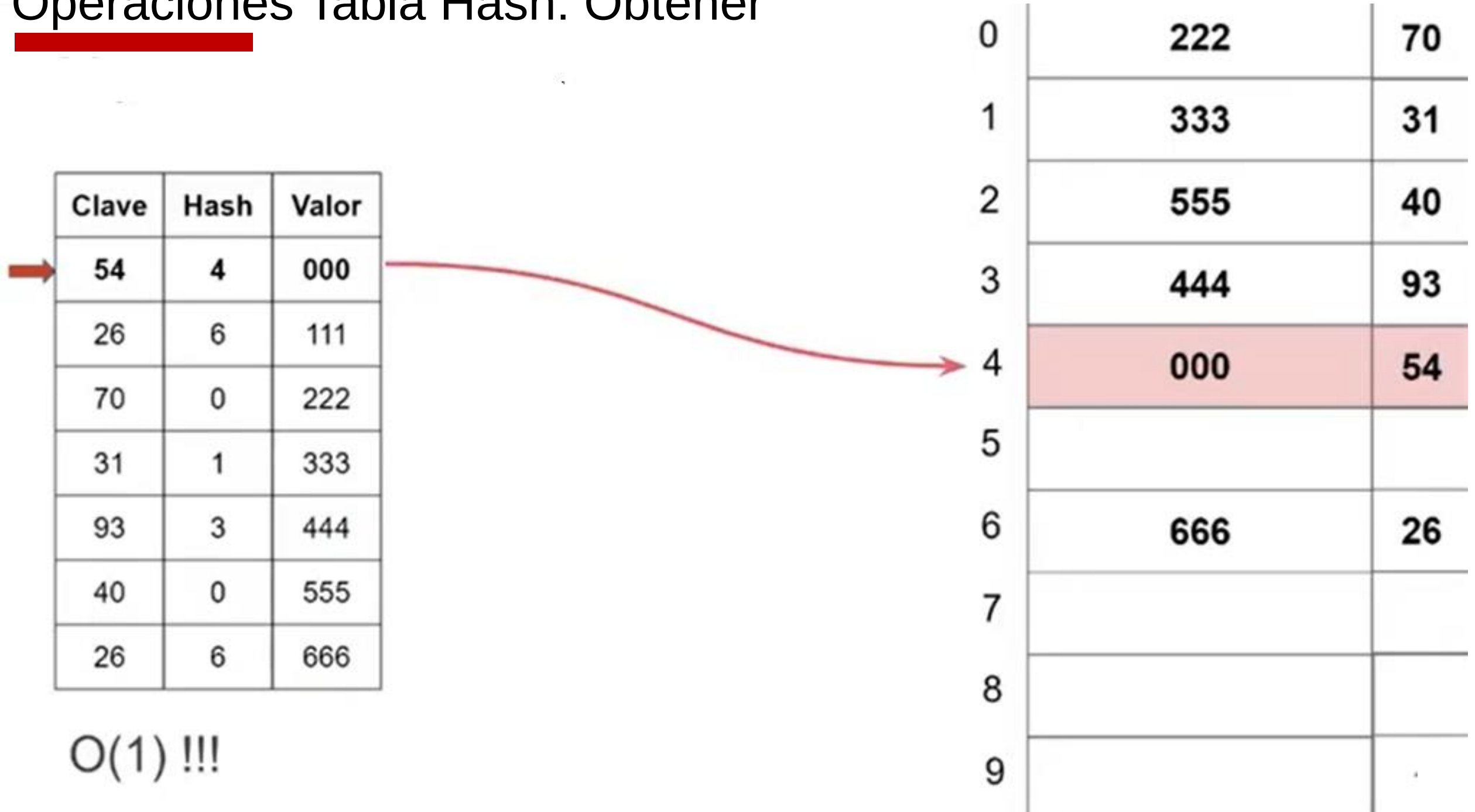
0	**222**	**70**
1	**333**	**31**
2	**555**	**40**
3	**444**	**93**
4	**000**	**54**
5		
6	**666**	**26**
7	**777**	**7**
8		
9		

Demo Tabla Hash

Operaciones Tabla Hash: Obtener

Clave	Hash	Valor
54	4	000
26	6	111
70	0	222
31	1	333
93	3	444
40	0	555
26	6	666

O(1) !!!



Operaciones Tabla Hash: Obtener

Clave	Hash	Valor
54	4	000
26	6	111
70	0	222
31	1	333
93	3	444
40	**0**	**555**
26	6	666

0	**222**	**70**
1	**333**	**31**
2	**555**	**40**
3	**444**	**93**
4	**000**	**54**
5		
6	**666**	**26**
7		
8		
9		

Clave	Hash	Valor
54	4	000
26	6	111
70	0	222
31	1	333
93	3	444
40	**0**	**555**
26	6	666

No coincide la clave, no es el valor que busco

Sigo buscando hasta que:
- Lo encuentre
- Encuentre espacio vacío

P = (P + 1) % Tamaño_tabla
P = (0 + 1) % 10 = 1

0	**222**	**70**
1	**333**	**31**
2	**555**	**40**
3	**444**	**93**
4	**000**	**54**
5		
6	**666**	**26**
7		
8		
9		

Clave	Hash	Valor
54	4	000
26	6	111
70	0	222
31	1	333
93	3	444
40	**0**	**555**
26	6	666

No coincide la clave, no es el valor que busco

Sigo buscando hasta que:
- Lo encuentre
- Encuentre espacio vacío

P = (P + 1) % Tamaño_tabla
P = (1 + 1) % 10 = 2

0	**222**	**70**
1	**333**	**31**
2	**555**	**40**
3	**444**	**93**
4	**000**	**54**
5		
6	**666**	**26**
7		
8		
9		

Clave	Hash	Valor
54	4	000
26	6	111
70	0	222
31	1	333
93	3	444
40	**0**	**555**
26	6	666

0	**222**	**70**
1	**333**	**31**
2	**555**	**40**
3	**444**	**93**
4	**000**	**54**
5		
6	**666**	**26**
7		
8		
9		

Operaciones Tabla Hash: Obtener inexistente

Clave	Hash	Valor
54	4	000
26	6	111
70	0	222
31	1	333
93	3	444
40	0	555
26	6	666
3	**3**	**777**

No coincide la clave, no es el valor que busco

Sigo buscando hasta que:
- Lo encuentre
- Encuentre espacio vacío

0	222	70
1	333	31
2	555	40
3	444	93
4	000	54
5		
6	666	26
7		
8		
9		

Clave	Hash	Valor
54	4	000
26	6	111
70	0	222
31	1	333
93	3	444
40	0	555
26	6	666
3	**3**	**777**

No coincide la clave, no es el valor que busco

Sigo buscando hasta que:
- Lo encuentre
- Encuentre espacio vacío

0	222	70
1	333	31
2	555	40
3	444	93
4	000	54
5		
6	666	26
7		
8		
9		

Clave	Hash	Valor
54	4	000
26	6	111
70	0	222
31	1	333
93	3	444
40	0	555
26	6	666
3	**3**	**777**

El espacio se encuentra vacío

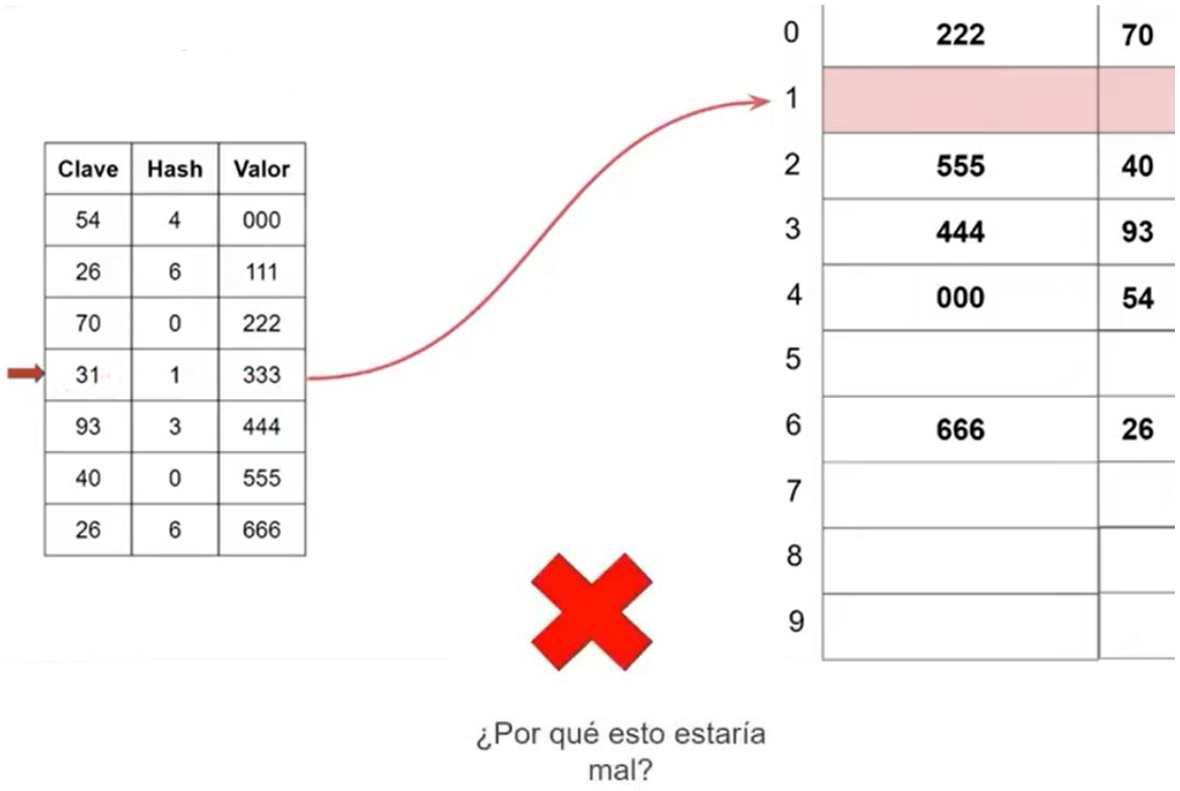
La clave no puede estar en la tabla, porque se hubiese posicionado en ese espacio

0	222	70
1	333	31
2	555	40
3	444	93
4	000	54
5		
6	666	26
7		
8		
9		

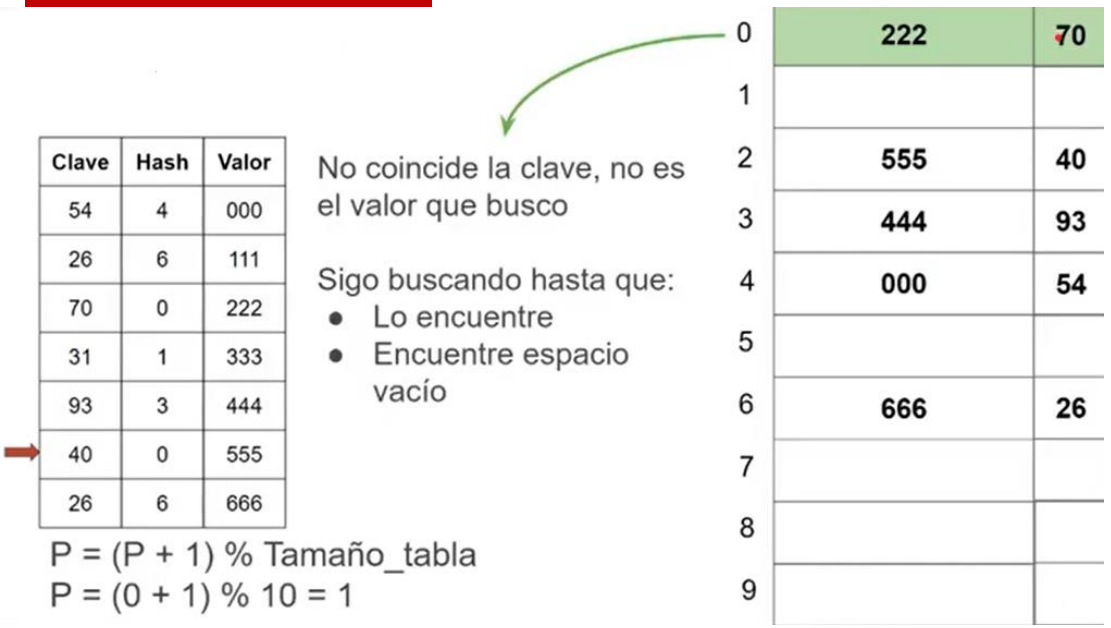
Operaciones Tabla Hash: Eliminar

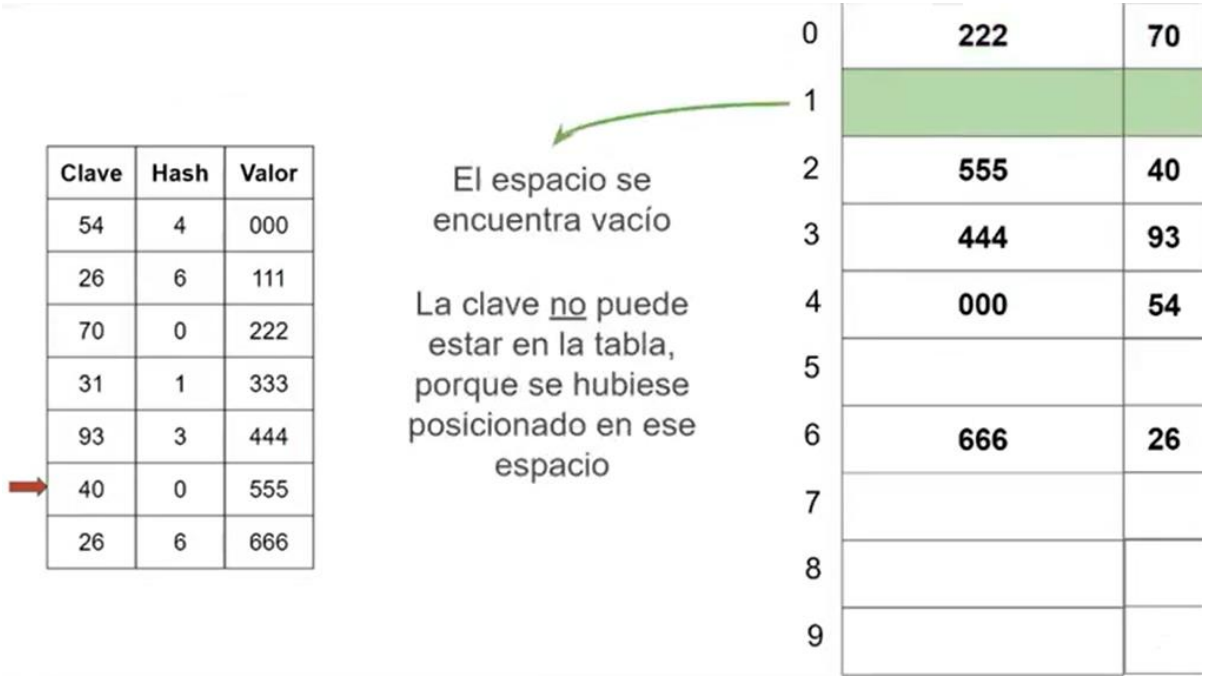
Clave	Hash	Valor
54	4	000
26	6	111
70	0	222
31	**1**	**333**
93	3	444
40	0	555
26	6	666

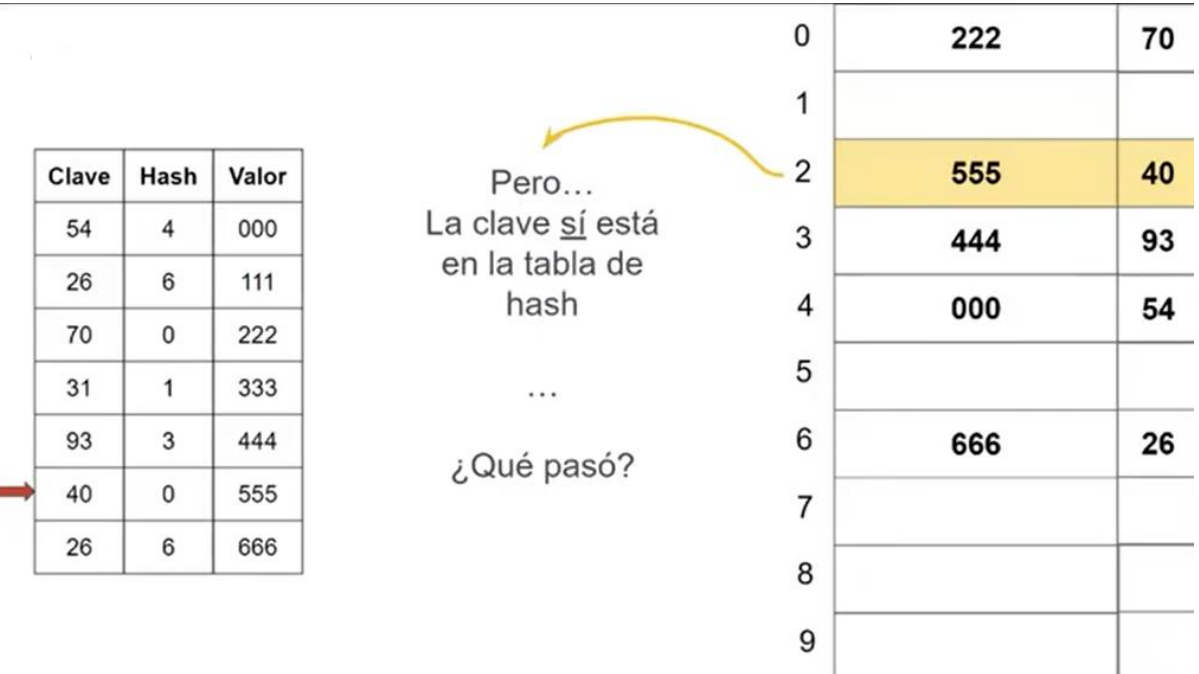
0	**222**	**70**
1	**333**	**31**
2	**555**	**40**
3	**444**	**93**
4	**000**	**54**
5		
6	**666**	**26**
7		
8		
9		



Operaciones Tabla Hash: Eliminar







¿Cómo eliminar?

- Reemplazar el espacio que acabamos de vaciar
- Utilizar un flag para indicar que se borró algo

Operaciones Tabla Hash: Eliminar con reemplazo

Clave	Hash	Valor
54	4	000
26	6	111
70	0	222
31	1	333
93	3	444
40	0	555
26	6	666

0	222	70
1	333	31
2	555	40
3	444	93
4	000	54
5		
6	666	26
7		
8		
9		

Clave	Hash	Valor
54	4	000
26	6	111
70	0	222
31	1	333
93	3	444
40	0	555
26	6	666

Avanzo hasta encontrar:

- el próximo espacio vacío
- una clave que pueda ser movida a ese nuevo espacio que vaciamos

0	222	70
1		
2	555	40
3	444	93
4	000	54
5		
6	666	26
7		
8		
9		

Clave	Hash	Valor
54	4	000
26	6	111
70	0	222
31	1	333
93	3	444
40	0	555
26	6	666

Avanzo hasta encontrar:

- el próximo espacio vacío
- una clave que pueda ser movida a ese nuevo espacio que vaciamos

→ Me encuentro con la clave 40 con “hash” 0

0	222	70
1		
2	555	40
3	444	93
4	000	54
5		
6	666	26
7		
8		
9		

Clave	Hash	Valor
54	4	000
26	6	111
70	0	222
31	1	333
93	3	444
40	0	555
26	6	666

El espacio está ocupado, sigo recorriendo

P = (P + 1) % Tamaño_tabla
P = (0 + 1) % 10 = 1

0	222	70
1		
2	555	40
3	444	93
4	000	54
5		
6	666	26
7		
8		
9		

Operaciones Tabla Hash: Eliminar con reemplazo

Clave	Hash	Valor
54	4	000
26	6	111
70	0	222
31	1	333
93	3	444
40	0	555
26	6	666

El espacio está libre, puedo guardarlo acá

0	222	70
1		
2	555	40
3	444	93
4	000	54
5		
6	666	26
7		
8		
9		

Clave	Hash	Valor
54	4	000
26	6	111
70	0	222
31	1	333
93	3	444
40	0	555
26	6	666

El espacio está libre, puedo guardarlo acá

Pero lo saco de acá…

Volvió a quedar un espacio libre → aplico el mismo procedimiento

0	222	70
1	555	40
2		
3	444	93
4	000	54
5		
6	666	26
7		
8		
9		

Clave	Hash	Valor
54	4	000
26	6	111
70	0	222
31	1	333
93	3	444
40	0	555
26	6	666

En este caso, el siguiente no es vacío y la clave 93 con "hash" 3 está bien posicionada

0	222	70
1	555	40
2		
3	444	93
4	000	54
5		
6	666	26
7		
8		
9		

Clave	Hash	Valor
54	4	000
26	6	111
70	0	222
31	1	333
93	3	444
40	0	555
26	6	666

En este caso, el siguiente no es vacío y la clave 54 con "hash" 4 está bien posicionada

El espacio está libre, terminé con el proceso de quitar claves

0	222	70
1	555	40
2		
3	444	93
4	000	54
5		
6	666	26
7		
8		
9		

Tipos de Hash: Cerrado con sondeo lineal

```
1  public class HashTableLP {
2   private static int EMPTY_VALUE = 0;
3   private static int DELETED_VALUE = 1;
4   private static int FILLED_VALUE = 2;
5   private int tableSize;
6    int[ ] Arr;
7    int[ ] Flag;
8  public HashTableLP(int tSize ) {
9    tableSize = tSize;
10   Arr = new int[ tSize + 1];
11  Flag = new int[ tSize + 1]; // el valor 0 en el Flag se considera vacío.
12  }
13 int computeHash (int key) {
14  return key % tableSize ;}
15 int resolverFun (int index) {
16  return index;}
17 boolean add (int value) {
18   int hashValue = computeHash (value);
19   for (int i = 0; i < tableSize; i++) {
20   if (Flag[hashValue]==EMPTY_VALUE || Flag[hashValue]==DELETED_VALUE) {
21   Arr[hashValue] = value;
22   Flag[hashValue] = FILLED_VALUE;
23   return true; }
24   hashValue += resolverFun (i);
25   hashValue %= tableSize; }
26   return false; }
```

```
27 boolean find (int value) {
28   int hashValue = computeHash (value);
29   for (int i = 0; i < tableSize; i++) {
30   if (Flag[hashValue] == EMPTY_VALUE) {
31   return false; }
32   if (Flag[hashValue] == FILLED_VALUE && Arr[hashValue] == value) {
33   return true; }
34   hashValue += resolverFun (i);
35   hashValue %= tableSize;}
36   return false;}
37 boolean remove (int value) {
38   int hashValue = computeHash (value);
39   for (int i = 0; i < tableSize; i++) {
40   if (Flag[hashValue] == EMPTY_VALUE) {
41   return false; }
42   if (Flag[hashValue] == FILLED_VALUE && Arr[hashValue] == value) {
43   Flag[hashValue] = DELETED_VALUE;
44   return true; }
45   hashValue += resolverFun (i);
46   hashValue %= tableSize; }
47   return false;}
48 void print () {
49   for (int i = 0; i < tableSize; i++) {
50 if (Flag[i] == FILLED_VALUE) {
51   System.out. println ("Node at index [" + i + "] :: " + Arr[i]);} }
52   }
```

```
53 public static void main (String[] args) {
54   HashTableLP ht = new HashTableLP (1000);
55   ht. add (1);
56   ht. add (2);
57   ht. add (3);
58   ht. print ();
59   System.out. println (ht. remove (1));
60   System.out. println (ht. remove (4));
61   ht. print ();
62     }
63   }
```

Ejercicio

Implementar el siguiente Pseudocódigo con una función hash por módulo o división:

- *La función hash por módulo o división consiste en tomar el residuo de la división de la clave entre el número de componentes del arreglo donde se almacenarán las claves.*
- *Supongamos que tenemos un arreglo de N elementos y K es la clave del dato a buscar. Las función hash queda definida por la siguiente fórmula: H(K)=(KmodN)+1*
- *Para lograr mayor uniformidad es importante que N sea un número primo o divisible entre muy pocos números*

```
Prueba_lineal (V, N, K)

{Este algoritmo busca al dato con clave K en el arreglo unidimensional V de N elementos.
Resuelve el problema de las colisiones por medio del método de prueba lineal}
{D y DX son variables de tipo entero}

1. Hacer D ← H(K) {Genera dirección}
2. Si ((V[DX] ≠ VACÍO) y (V[D] = K))
       entonces
           Escribir "La información está en la posición", D
       si no
           Hacer DX ← D + 1
   2.1 Mientras ((DX ≤ N) y (V[DX] ≠ VACÍO) y (V[DX] ≠ K) y (DX ≠ D))
           Repetir
           Hacer DX ← DX + 1
       2.1.1 Si (DX = N + 1) entonces
                 Hacer DX ← 1
       2.1.2 {Fin del condicional del paso 2.1.1}
   2.2 {Fin del ciclo del paso 2.1}
   2.3 Si ((V[DX] = VACÍO) o (DX = D))
       entonces
           Escribir "La información no se encuentra en el arreglo"
       si no
           Escribir "La información está en la posición", DX
   2.4 {Fin del condicional del paso 2.3}
3. {Fin del condicional del paso 2}
```

1	80
2	
3	
4	43
5	54
6	25
7	56
8	35
9	13
10	104

K	*H(K)*
25	6
43	4
56	7
35	6
54	5
13	4
80	1
104	5

D	*DX*
4	5
	6
	7
	8
	9

Buscando el 13

¿Preguntas?

FIN

Codificación de la Búsqueda por Transformación de claves o hash

Clase HashTable

```java
package Clases;
public class HashTable {
    private static int EMPTY_NODE = -1;
    private static int LAZY_DELETED = -2;
    private static int FILLED_NODE = 0;
    private int tableSize;
    int[] Arr;
    int[] Flag;
    public HashTable(int tSize) {
        tableSize = tSize;
        Arr = new int[tSize + 1];
        Flag = new int[tSize + 1];
        for (int i = 0; i <= tSize; i++) {
            Flag[i] = EMPTY_NODE;
        }
    }
    int ComputeHash(int key) {
        return key % tableSize;
    }
    int resolverFun(int index) {
        return index;
    }
```

```java
    boolean InsertNode(int value) {
        int hashValue = ComputeHash(value);
        for (int i = 0; i < tableSize; i++) {
            if (Flag[hashValue] == EMPTY_NODE || Flag[hashValue] == LAZY_DELETED)
                Arr[hashValue] = value;
                Flag[hashValue] = FILLED_NODE;
                return true;
            }
            hashValue += resolverFun(i);
            hashValue %= tableSize;
        }
        return false;
    }
```

Codificación de la Búsqueda por Transformación de claves o hash

Clase HashTable

```
36      boolean FindNode(int value) {
37          int hashValue = ComputeHash(value);
38          for (int i = 0; i < tableSize; i++) {
39              if (Flag[hashValue] == EMPTY_NODE) {
40                  return false;
41              }
42              if (Flag[hashValue] == FILLED_NODE && Arr[hashValue] == value) {
43                  return true;
44              }
45              hashValue += resolverFun(i);
46              hashValue %= tableSize;
47          }
48          return false;
49      }
50      void Print() {
51          for (int i = 0; i < tableSize; i++) {
52              if (Flag[i] == FILLED_NODE) {
53                  System.out.println("Node at index [" + i + " ] :: " + Arr[i]);
54              }
55          }
56      }
```

Codificación de la Búsqueda por Transformación de claves o hash

Clase UsoHash

```
1  package Clases;
2  import java.util.Scanner;
3  public class Usohash {
4      public static void main(String[] args) {
5          Scanner entrada=new Scanner(System.in);
6          System.out.println("Ingrese el tamaño de la tabla");
7          int tamanio=entrada.nextInt();
8          int[] listaoriginal=new int[tamanio];
9          for(int i=0;i<tamanio;i++){
10             listaoriginal[i]=(int)(Math.random()*1000);
11         }
12         HashTable ht=new HashTable(tamanio);
13         System.out.println("\nTabla original\n");
14         for(int i=0;i<tamanio;i++){
15             System.out.println("Original at index[" +i+ "]::"+listaoriginal[i]);
16         }
17         System.out.println("\nTabla Hash con función módulo\n");
18         for(int i=0;i<tamanio;i++){
19             ht.InsertNode(listaoriginal[i]);
20         }
21         ht.Print();
22         System.out.println("\nSearch 243 ::"+ht.FindNode(243));
23     }
24 }
```

Búsqueda por Transformación de claves o hash

Tabla Hash

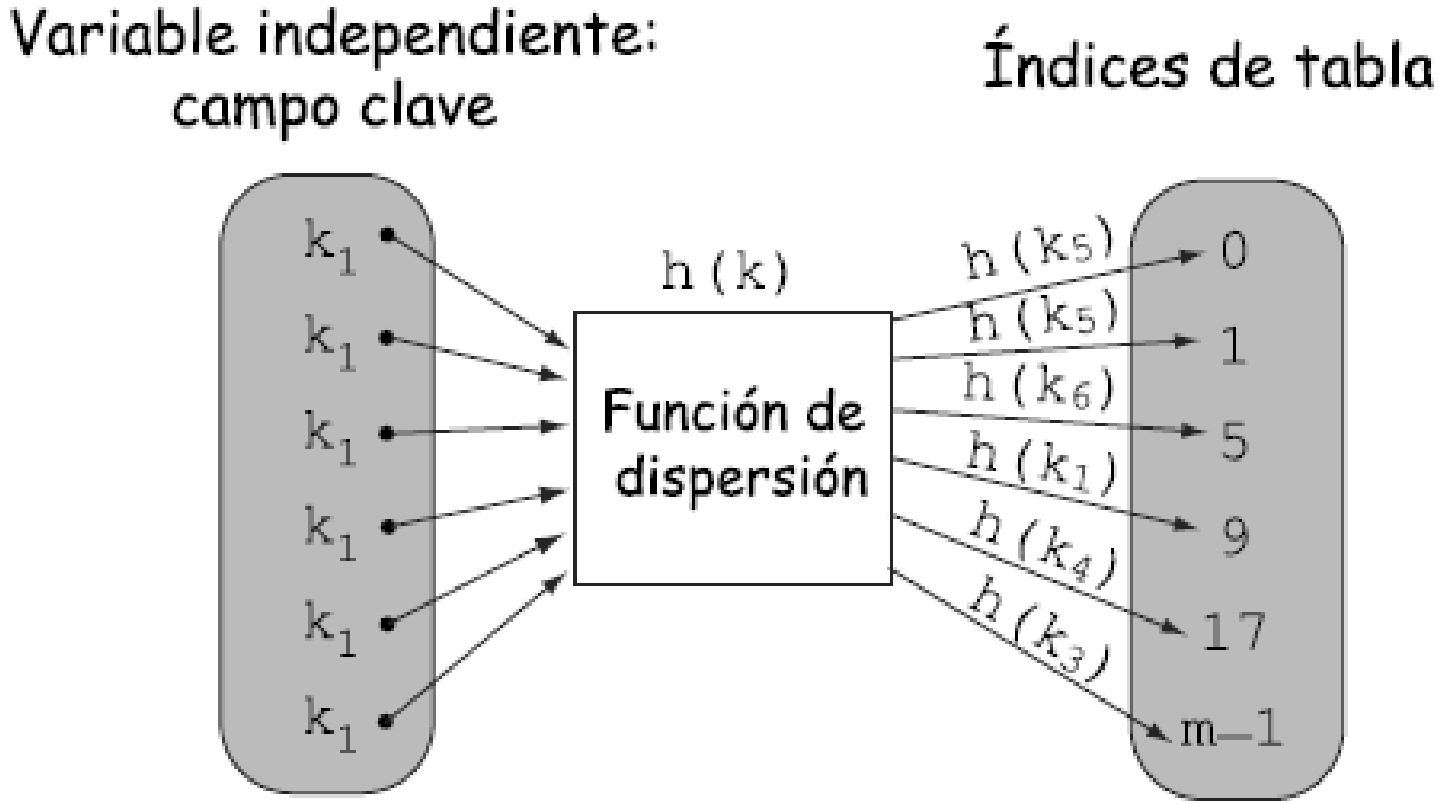
- Es una Estructura de datos.
- Se basan en la asignación de una clave a cada elemento.
- Inserción y búsqueda rápida.
- Limitadas en tamaño ya que están basados en arreglos.
- Su tamaño – 1 es recomendable que sea un numero primo

- Las claves son asignadas a elementos en una Tabla Hash usando una Función Hash.
- Una Función Hash ayuda a calcular el índice optimo en el cual un elemento debería ubicarse.
- El índice debe se menor que el tamaño de la tabla y mayor o igual a cero.
- No debe haber datos repetidos en una Tabla Hash.

Más sobre funciones hash o de dispersión

Normalmente una función hash o de dispersión es un proceso de dos pasos:

- *Transforma la clave (que puede no ser un número entero) en un valor entero*
- *Asigna el entero resultante en un índice válido para la tabla hash o tabla de dispersión (donde se almacenan todos los elementos)*



Para facilitar la búsqueda, una función hash debe generar posiciones diferentes dadas dos claves diferentes. Si esto último no ocurre (H(K1)=d,H(K2)=d y K1!=K2) se produce una colisión.

Entonces una función hash debe ser fácil de calcular y distribuir uniformemente las claves y debe evitar las colisiones. Si éstas se presentan se contará con algún método que genere posiciones alternativas.

- Las funciones hash pueden basarse en una serie de técnicas, entre las más utilizadas tenemos: función hash por módulo, función hash cuadrado, función hash por plegamiento, función hash por truncamiento.