Hashing

- Materia Algoritmos y Estructuras de Datos – CB100
- Cátedra Schmidt
- Tablas de dispersión, tablas hash

Introducción:

Supongamos que tenemos que administrar una cochera donde entran 500 autos. En una primera reflexión pensaríamos en ir guardando en un vector o un archivo los registros de cada uno (patente, dueño del auto, teléfono, etc), en el orden que ingresen. Sin embargo, una búsqueda en este vector o archivo debería ser secuencial, por lo que tendría orden N: O(N). En este caso 500.

Para hacer más eficiente la búsqueda podríamos ordenar este vector, por ejemplo por patente, y hacer una búsqueda binaria. De esta forma bajaríamos el orden a log2(N). En este caso 9.

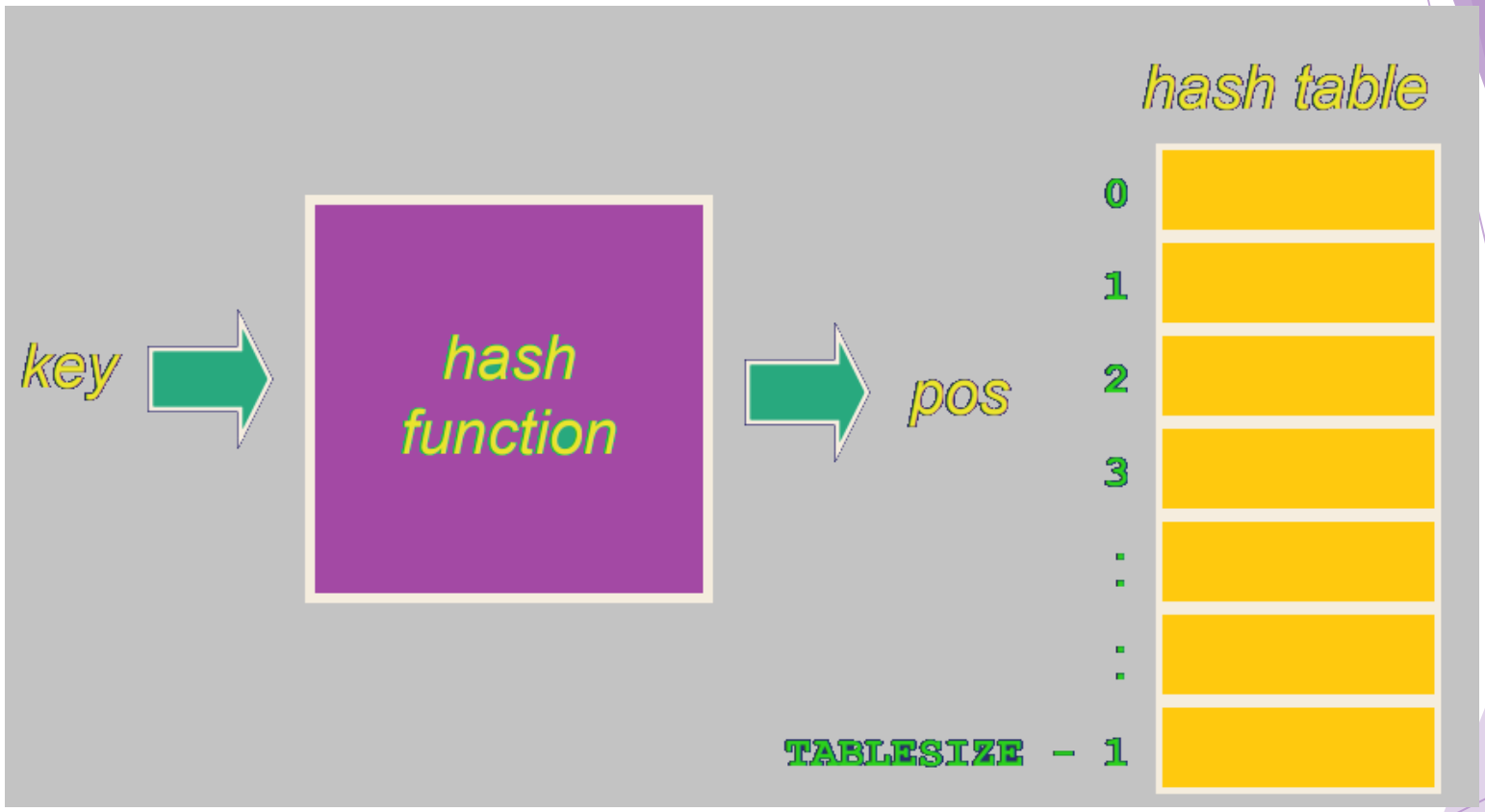
Pero si el tiempo de búsqueda es muy crítico y quisiéramos llevarlo a O(1), es decir, obtener el registro en el primer intento, lo que se nos ocurriría es hacer coincidir la posición del registro con el propio valor de la clave. Por ejemplo, la patente AAA000 iría en la posición 1, la patente AAA001 en la 2, etc. El problema de hacer esto es que necesitaríamos un vector de 26x26x26x10x10x10 = 17576000 posiciones, lo cual no tendría sentido para guardar sólo 500 datos.

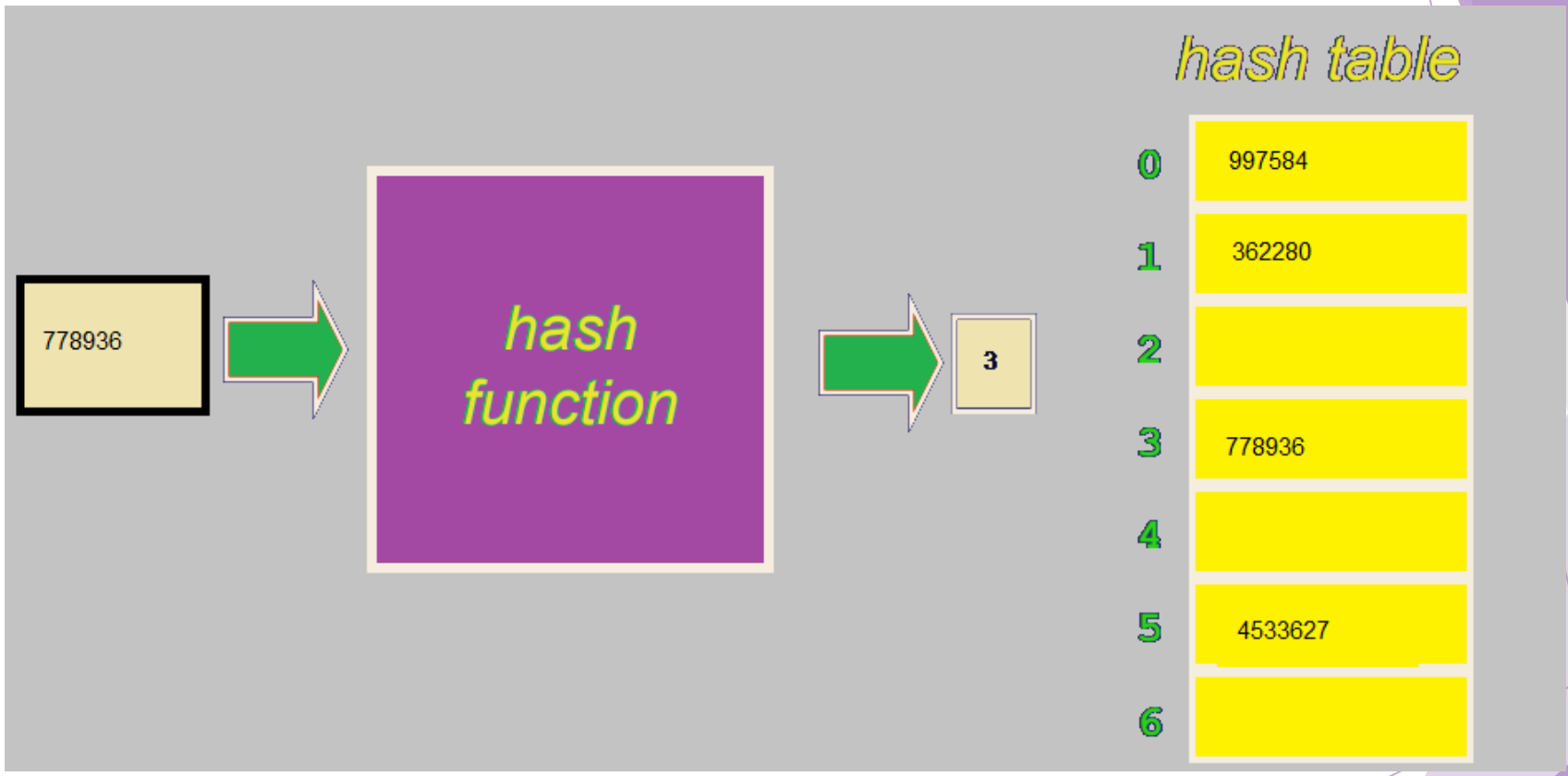
En estos casos intervienen las funciones de hashing. Lo que hacen es llevar esta clave a una nueva, más manejable. Por ejemplo, una función de hashing para la cochera podría ser quitar la parte de las letras de las patentes y quedarse sólo con la parte numérica. Entonces tendríamos un vector de 1000 posiciones (ya mucho más coherente) en donde la patente MNK025 iría en la posición 25 y la patente IXY162 iría en la 162. El problema es que si viene un auto con la patente KHH162 debería ocupar el mismo registro que el anterior. Cuando sucede esto se dice que hay una colisión.

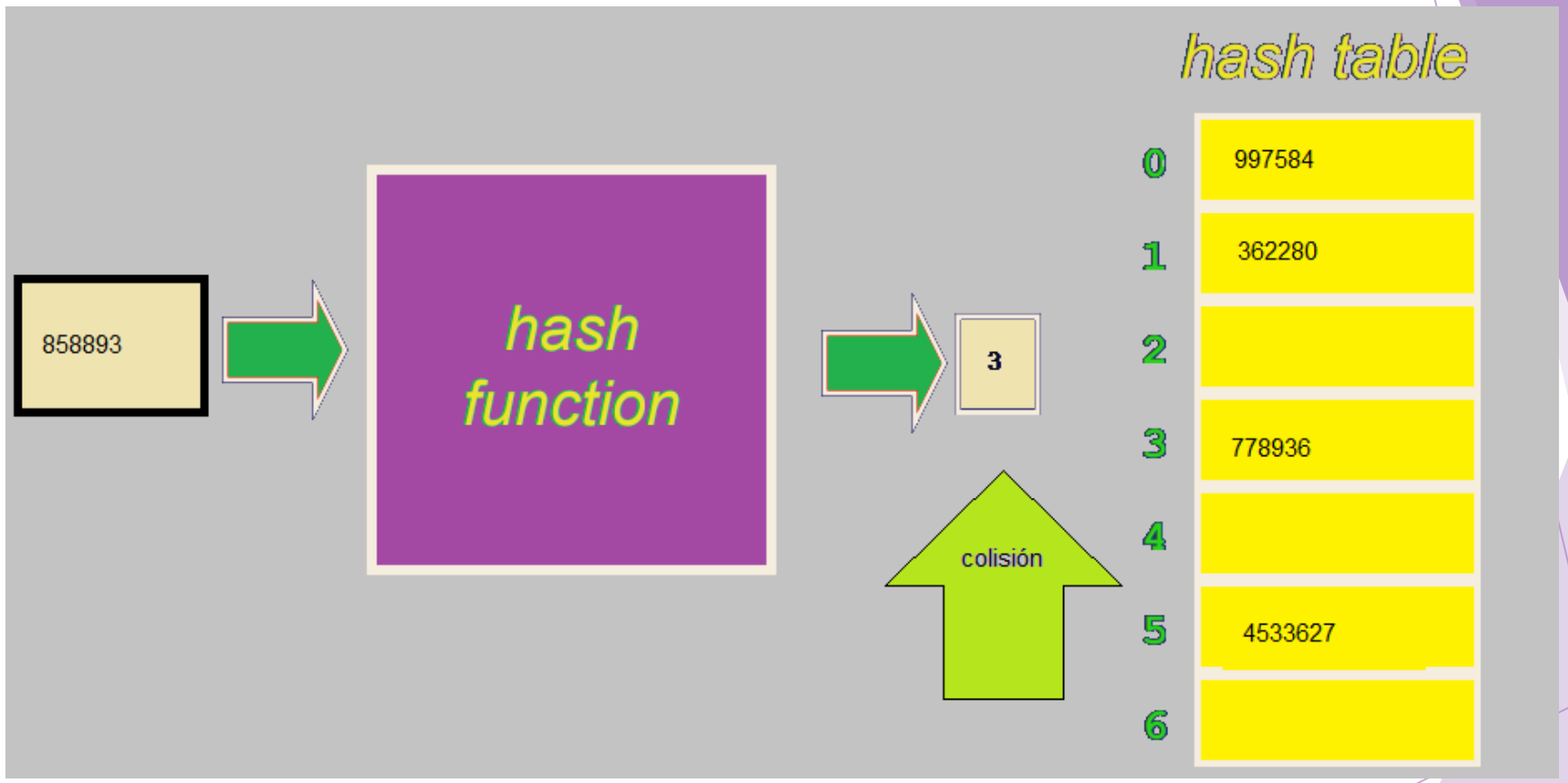
Las técnicas de hashing tratan sobre las diferentes funciones de generación de claves y cómo resolver el problema de las colisiones.

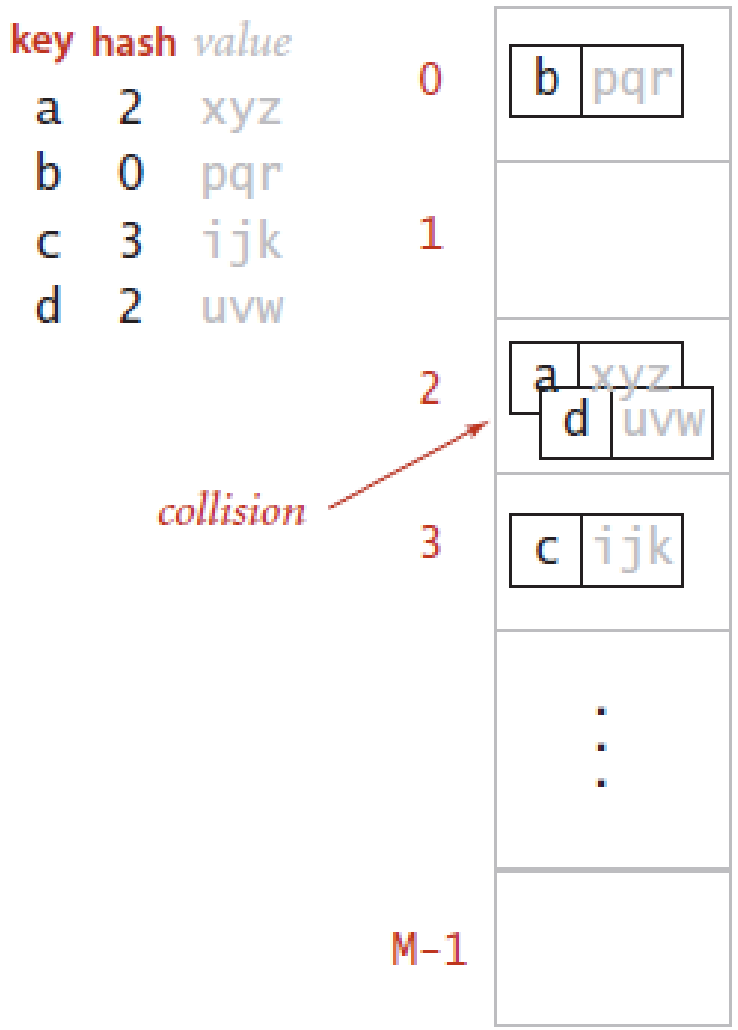
Las tablas hash son estructuras de datos que se utilizan para almacenar un número elevado de datos sobre los que se necesitan operaciones de búsqueda e inserción muy eficientes. Una tabla hash almacena un conjunto de pares “(clave, valor)”. La clave es única para cada elemento de la tabla y es el dato que se utiliza para buscar un determinado valor.
Un diccionario es un ejemplo de estructura que se puede implementar mediante una tabla hash. Para cada par, la clave es la palabra a buscar, y el valor contiene su significado. El uso de esta estructura de datos es tan común en el desarrollo de aplicaciones que algunos lenguajes las incluyen como tipos básicos.

Una alternativa a los algoritmos de búsqueda basados en comparaciones entre las claves son aquellos que se basan en la transformación de la clave.
Aplican una operación aritmética sobre la clave para obtener su localización en la estructura de datos.


hash table
key
hash function
pos
0
1
2
3
:
:
TABLESIZE - 1


hash table
778936
hash function
3
0
997584
1
362280
2
3
778936
4
5
4533627
6


858893
hash function
3
colisión
hash table
0
997584
1
362280
2
3
778936
4
5
4533627
6

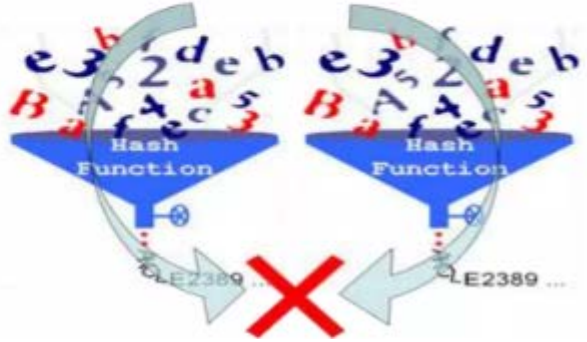


Hashing: the crux of the problem

Colisión:

Un problema potencial encontrado en el proceso hash, es que tal función no puede ser uno a uno; las direcciones calculadas pueden no ser todas únicas, cuando H(K1) = H(K2).

Si K1 es diferente de K2 decimos que hay una colisión. A dos claves diferentes que les corresponda la misma dirección relativa se les llama sinónimos.



Funciones Hash

- Asigna claves a posiciones en la tabla Hash.
- Es facil de calcular.
- Usa todas las claves.
- Distribuya las claves uniformemente

Otro ejemplo:

Un ejemplo práctico para ilustrar qué es una tabla *hash* es el siguiente: Se necesita organizar los periódicos que llegan diariamente de tal forma que se puedan ubicar de forma rápida, entonces se hace lo siguiente: se hace una gran caja para guardar todos los periódicos (una tabla), y se divide en 31 contenedores (ahora es una *hash table* o tabla fragmentada), y la clave para guardar los periódicos es el día de publicación (índice). Cuando se requiere buscar un periódico se busca por el día que fue publicado y así se sabe en que zócalo (*bucket*) está. Varios periódicos quedarán guardados en el mismo zócalo (es decir, colisionan al ser almacenados), lo que implica buscar en la sub-lista que se guarda en cada zócalo. De esta forma se reduce el tamaño de las búsquedas de O(n) a O(log(n)).

Una buena función hash es esencial para el buen rendimiento de una tabla hash. Las colisiones son generalmente resueltas por algún tipo de búsqueda lineal, así que si la función tiende a generar valores similares, las búsquedas resultantes se vuelven lentas.
En una función hash ideal, el cambio de un simple bit en la llave (incluyendo el hacer la llave más larga o más corta) debería cambiar la mitad de los bits del hash, y este cambio debería ser independiente de los cambios provocados por otros bits de la llave. Como una función hash puede ser difícil de diseñar, o computacionalmente cara de ejecución, se han invertido muchos esfuerzos en el desarrollo de estrategias para la resolución de colisiones que mitiguen el mal rendimiento del *hasheo*. Sin embargo, ninguna de estas estrategias es tan efectiva como el desarrollo de una buena función hash de principio.
Es deseable utilizar la misma función hash para arrays de cualquier tamaño concebible. Para esto, el índice de su ubicación en el array de la tabla hash se calcula generalmente en dos pasos:
1.Un valor hash genérico es calculado, llenando un entero natural de máquina.
2.Este valor es reducido a un índice válido en el vector encontrando su módulo con respecto al tamaño del array.
El tamaño del vector de las tablas hash es con frecuencia un número primo. Esto se hace con el objetivo de evitar la tendencia de que los hash de enteros grandes tengan divisores comunes con el tamaño de la tabla hash, lo que provocaría colisiones tras el cálculo del módulo. Sin embargo, el uso de una tabla de tamaño primo no es un sustituto a una buena función hash.
Un problema bastante común que ocurre con las funciones hash es el *aglomeramiento*. El *aglomeramiento* ocurre cuando la estructura de la función hash provoca que llaves usadas comúnmente tiendan a caer muy cerca unas de otras o incluso consecutivamente en la tabla hash. Esto puede degradar el rendimiento de manera significativa, cuando la tabla se llena usando ciertas estrategias de resolución de colisiones, como el sondeo lineal.
Cuando se depura el manejo de las colisiones en una tabla hash, suele ser útil usar una función hash que devuelva siempre un valor constante, como 1, que cause colisión en cada inserción.

key	hash ($M = 100$)	hash ($M = 97$)
212	12	18
618	18	36
302	2	11
940	40	67
702	2	23
704	4	25
612	12	30
606	6	24
772	72	93
510	10	25
423	23	35
650	50	68
317	17	26
907	7	34
507	7	22
304	4	13
714	14	35
857	57	81
801	1	25
900	0	27
413	13	25
701	1	22
418	18	30
601	1	19

Modular hashing

```
int hash = 0;
for (int i = 0; i < s.length(); i++)
   hash = (R * hash + s.charAt(i)) % M;
```

Hashing a string key

Hash:

Una función hash H es una función computable mediante un algoritmo,

$$H: U \rightarrow M$$

$$x \rightarrow h(x),$$

que tiene como entrada un conjunto de elementos, que suelen ser cadenas, y los convierte (mapea) en un rango de salida finito, normalmente cadenas de longitud fija.

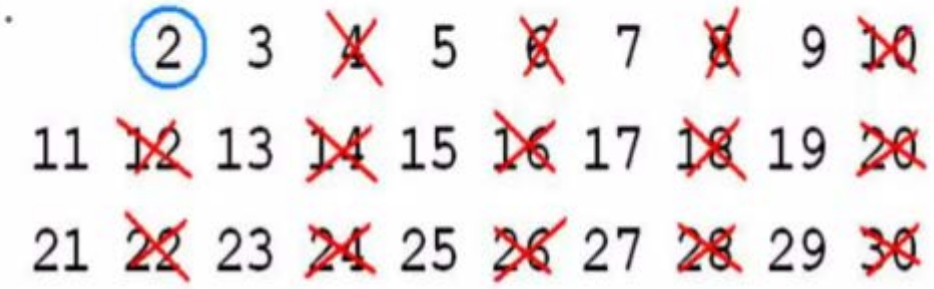
Método de División

La función de este método es dividir el valor de la llave entre un numero apropiado, y después utilizar el residuo de la división como dirección relativa para el registro.

$$F(hash) = llave \% divisor.$$

Existen varios factores que deben considerarse para seleccionar el divisor:

- divisor > n : suponiendo que solamente un registro puede ser almacenado en una dirección relativa dada.
- Seleccionarse el divisor de tal forma que la probabilidad de colisión sea minimizada.



Uso:

Cuando la distribución de los valores de llaves no es conocida.

Assume a table with 8 slots:

Hash key = key % table size

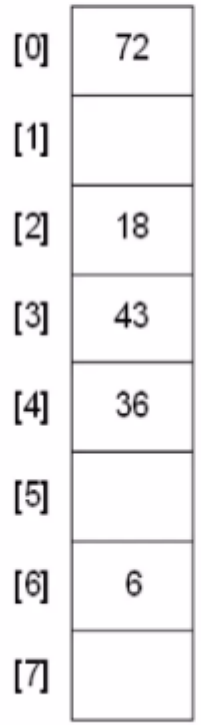
4 = 36 % 8

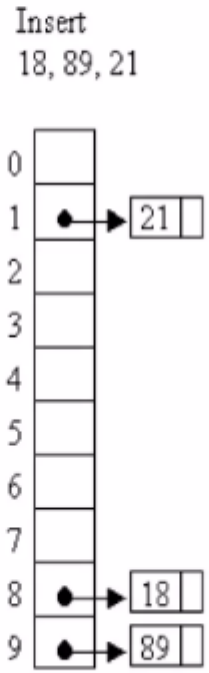
2 = 18 % 8

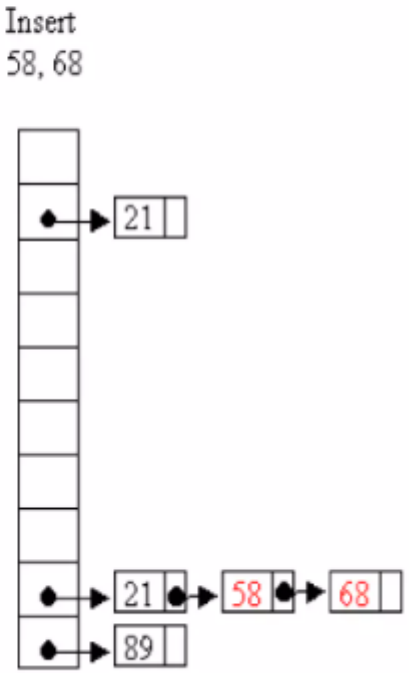
0 = 72 % 8

3 = 43 % 8

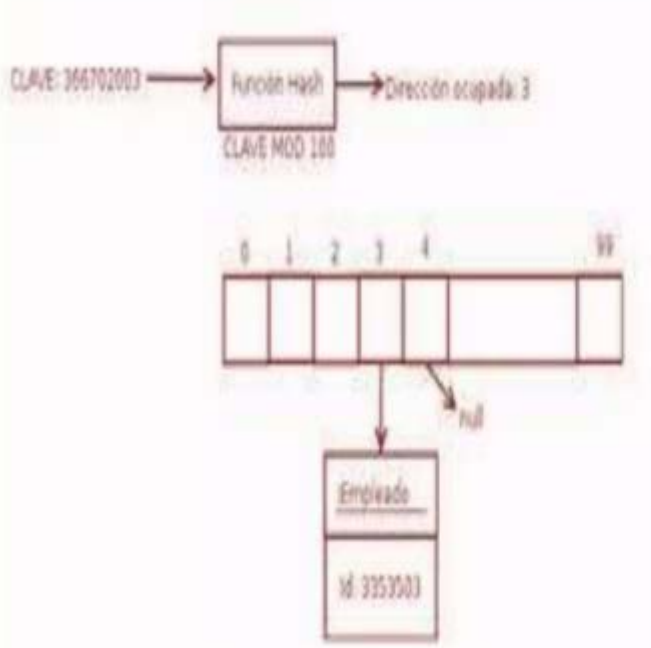
6 = 6 % 8







Ejemplo:



Si la tabla hash tiene tamaño m = 12 y la llave es

k = 100, entonces

h(k) = 100 mod 12 = 4.

Método de medio cuadrado

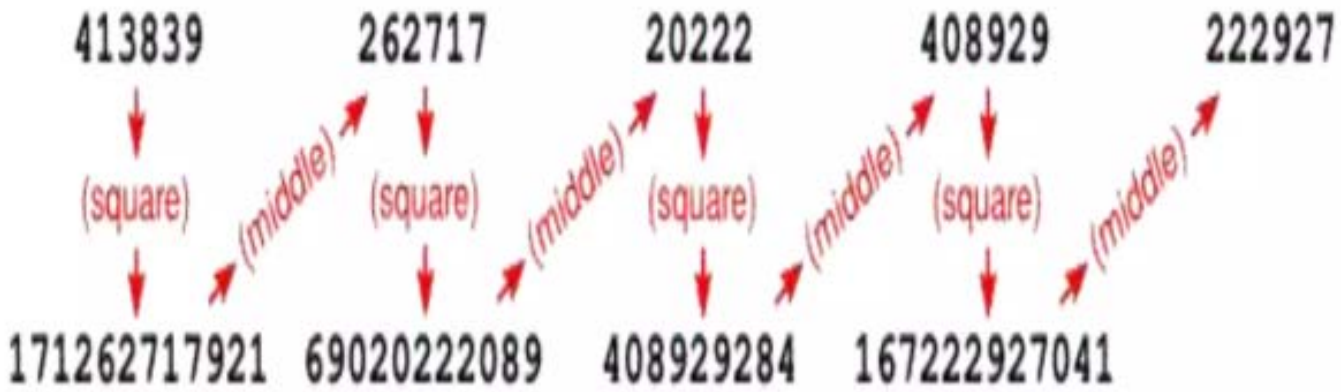
Consiste en elevar al cuadrado la clave y tomar los dígitos centrales como dirección. El número de dígitos a tomar queda determinado por el rango del índice. Sea K la clave del dato a buscar, la función hash queda definida por la siguiente formula:

$$H(K) = digitos\_centrales\ (K\text{^}2) + 1$$

Las suma de los dos dígitos centrales de la clave K (elevada al cuadrado) más 1, debe obtener un valor entre 1 y N (N, tamaño del arreglo).

Uso:

puede aplicarse en archivos con factores de cargas bastantes bajas



Ejemplo:

Sea N=100 el tamaño del arreglo.

Sea su dirección los números entre 1 y 100.

Sea K1 = 7259 una clave a la que se le debe asignar una posición en el arreglo.

Valor de la llave	Llave al cuadrado	Dirección relativa
123456789	15241578750190521	8750
987654321	975461055789971041	5789
123456790	15241578997104100	8997
555555555	308641974691358025	4691
000000472	00000000000222784	0000
100064183	10012860719457489	0719
200120472	40048203313502784	3313
200120473	40048203713743729	3713
117400000	13782760000000000	0000
027000400	02430021600160000	1600

"Colision"

K1^2 = 52 693 081

H(K1) = (52 693 081) + 1 = 94

FUNCIONES HASH: MÉTODO PLEGADO

- Para las funciones de plegado: Dividir la clave en partes iguales y sumarlas. La suma de las partes puede realizarse de dos formas

  a) Plegado por desplazamiento.

  b) Plegado por las fronteras.

- El plegado por desplazamiento consiste en sumar las partes directamente.

- El plegado por las fronteras consiste en plegar el identificador por las fronteras de las partes y sumar los dígitos coincidentes.

- Si la claves es una cadena de caracteres, los dígitos de cada carácter vienen determinados por el valor decimal de la secuencia de cotejo correspondiente (ASCII, EBCDIC, etc.)

FUNCIONES HASH: MÉTODO PLEGADO

a)

68	75	82	71	63	93	102	75

	b)	c)	d)
	75	75	75
	102	201	102
	93	93	93
	63	36	252
+	71	+ 71	+ 71
	82	28	74
	75	75	75
	68	86	34
Dirección =	629	665	776

e)

```
102 => 01100110                01100110 => 102
 63 => 00111111   ------->     11111100 => 252
 82 => 01010010                01001010 =>  74
 68 => 01000100                00100010 =>  34
```

a) Cadena de ocho caracteres representada por los números de orden dentro de la secuencia de cotejo correspondiente.

b) Plegado por desplazamiento.

c) Plegado en las fronteras en base decimal.

d) Plegado en las fronteras en base binaria.

e) Detalle del plegado en base binaria.

FUNCIONES HASH: MÉTODO COMPRESIÓN

Dividir la clave en componentes, traducir su número de cotejo a binario y aplicar la operación xor y al resultado la operación resto de división entera.

Ejemplo: C = Cadena de caracteres.

C = 'David'

'D' = 01100100
'A' = 01100001
'V' = 01110010
'I' = 01101001
'D' = 01100100

XOR

01111010 = 122 mod 26 = 18

FUNCIONES HASH: MÉTODO EXTRACCIÓN

Si tenemos claves numéricas (si no las transformamos), calculamos su cuadrado y nos quedamos con algún número de la zona central del resultado. Nos quedamos con tantos dígitos como necesitemos para mapear el array. Ejemplo: Array con N = 0..99 C = 340 i2 = 115600 H (C) = 56 C = 521 i 2 = 271441 H (C) = 14

Doble hash

- Lo que se hace es aplicar una segunda función de dispersión a la clave, y luego se prueba a distancias h2(x), 2h2(x), ...
- Usar una segunda función de hash, diferente de la primera,
- para determinar el incremento que usar para repartir las llaves

- Es muy importante la buena elección de h2(x) y, además, nunca debe ser cero. Si se elige la función:
- h2(x) = R - (x MOD R)
- con R un número primo menor que MAX_T, funcionará bien.

- Utilizando como segunda función hash 7-(clave%7) el ejemplo quedaría de la siguiente manera:

1	35
2	104
3	13
4	43
5	54
6	25
7	56
8	80
9	55
10	

K	H(k)	H'(k)
25	6	
43	4	
56	7	
35	6	7
54	5	
13	4	3
80	1	4
104	5	1
55	6	1

- Si observan, a partir de la segunda función hash, se realizan los incrementos dados de acuerdo a la dirección obtenida, es decir si la dirección obtenida en la segunda función es 4 y esa posición esta ocupada, se realizan incrementos de 4 en 4 hasta encontrar una celda vacía .

Transformación Radix

Usando esta transformación se cambia la base numérica del numero en cuestión, por ejemplo el numero 345 en base decimal se puede representar como el 423 en base 9. Luego este valor es dividido por el tamaño de la tabla quedándonos con el modulo

Resolución de colisiones:

- Mediante direccionamiento abierto (open adressing o closed hashing)
- Mediante Encadenamiento (chaining o open hashing)

Direccionamiento abierto o hashing cerrado [editar]

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

Las secuencias de sondeo más utilizadas son:

1. **Sondeo lineal**

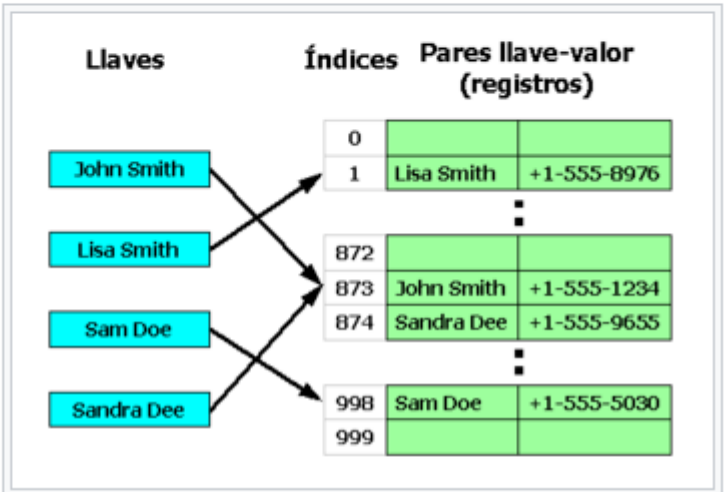
   En el que el intervalo entre cada intento es constante (frecuentemente 1). El sondeo lineal ofrece el mejor rendimiento del caché, pero es más sensible al aglomeramiento.

2. **Sondeo cuadrático**

   En el que el intervalo entre los intentos aumenta linealmente (por lo que los índices son descritos por una función cuadrática). El sondeo cuadrático se sitúa entre el sondeo lineal y el doble hasheo.

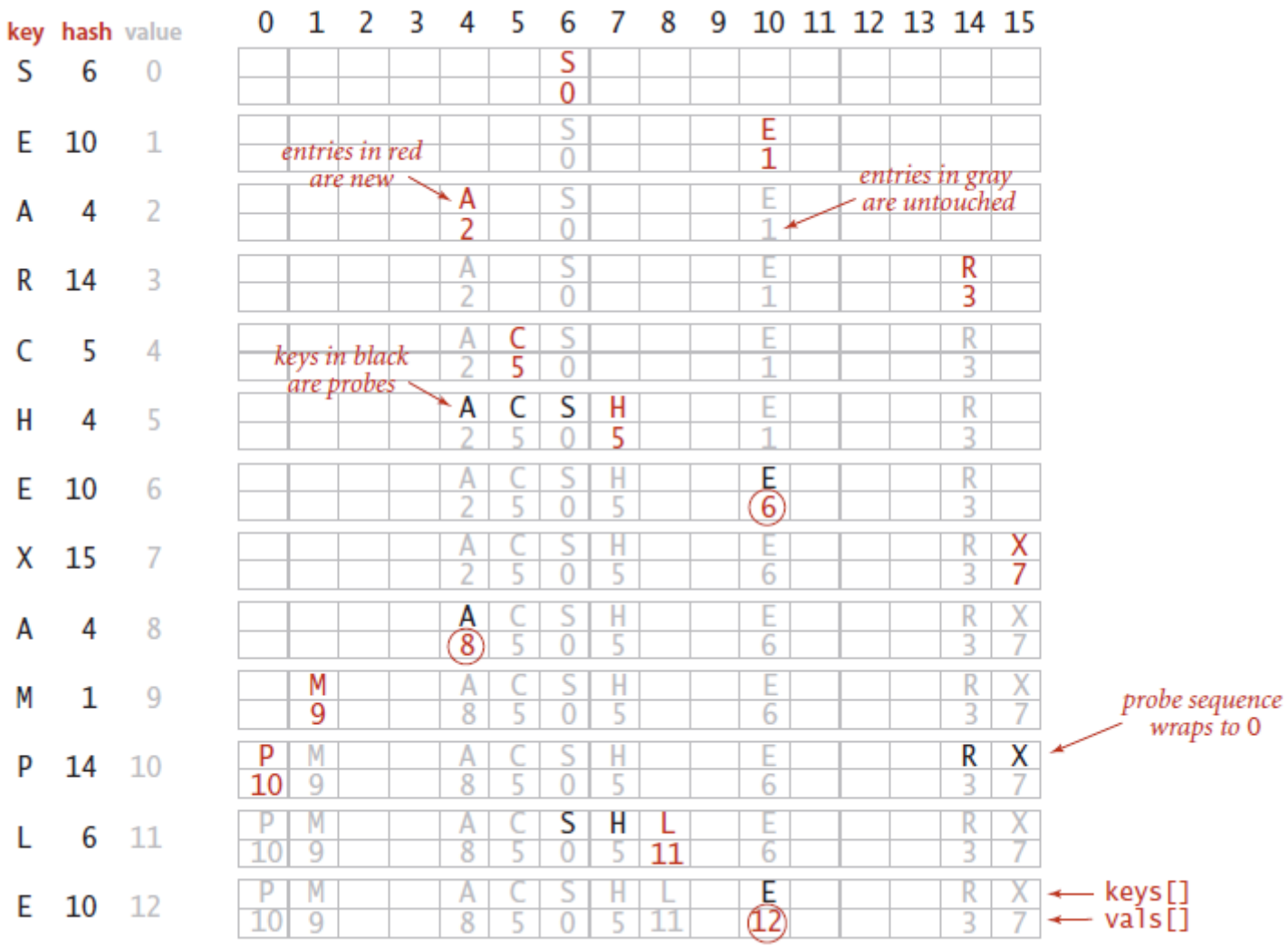
3. **Doble *hasheo***

   En el que el intervalo entre intentos es constante para cada registro pero es calculado por otra función hash. El doble hasheo tiene pobre rendimiento en el caché pero elimina el problema de aglomeramiento. Este puede requerir más cálculos que las otras formas de sondeo.



Ejemplo de direccionamiento abierto.

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

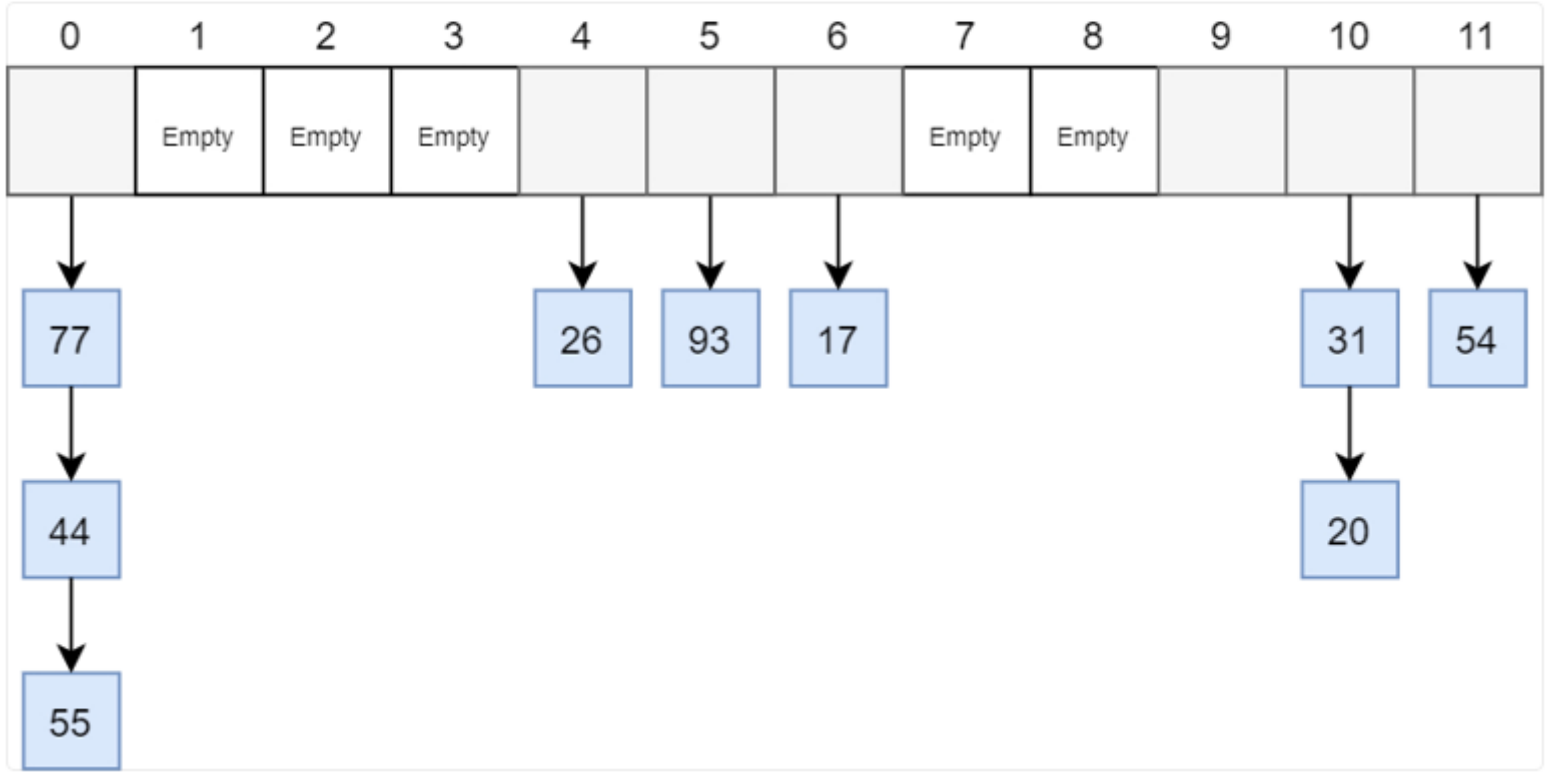


Resolución de colisiones por sondeo o prueba lineal

Encadenamiento (Técnica Hashing Abierta)

Los elementos que colisionan se van añadiendo a una lista asociada a la posición que colisiona. No se añaden registros, sino listas.

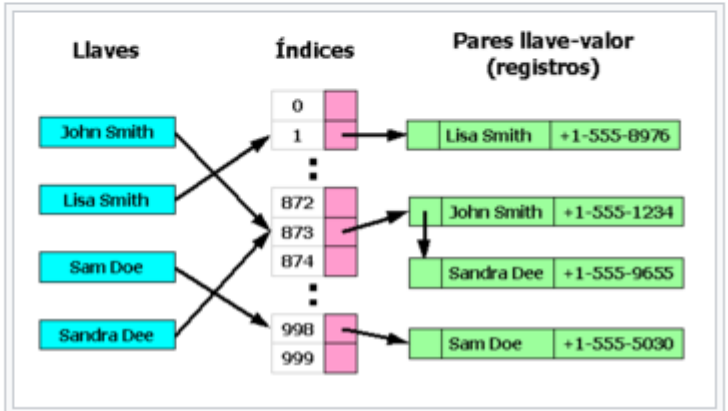
Cada posición de la tabla se mantiene con una lista enlazada en la que se van insertando elementos cuyo valor Hash les asigna la misma posición.



Direccionamiento cerrado, encadenamiento separado o hashing abierto [editar]

En la técnica más simple de encadenamiento, cada casilla en el array referencia a una lista con los registros insertados que colisionan en dicha casilla. La inserción consiste en encontrar la casilla correcta y agregar al final de la lista correspondiente. El borrado consiste en buscar y quitar de la lista.

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



Ejemplo de encadenamiento.

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

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

Deletion. How do we delete a key-value pair from a linear-probing table? If you think about the situation for a moment, you will see that setting the key's table position to `null` will not work, because that might prematurely terminate the search for a key that was inserted into the table later. As an example, suppose that we try to delete `C` in this way in our trace example, then search for `H`. The hash value for `H` is 4, but it sits at the end of the cluster, in position 7. If we set position 5 to `null`, then `get()` will not find `H`. As a consequence, we need to reinsert into the table all of the keys in the cluster to the right of the deleted key

Resolución de colisiones por sondeo o prueba cuadrática

If there is a collision at hash address h, ***quadratic probing*** probes the table at locations $h + 1$, $h + 4$, $h + 9$, ...; that is, at locations $h + i^2$. (In other words, the increment function is i^2.)

Quadratic probing substantially reduces clustering, but it is not obvious that it will probe all locations in the table, and in fact it does not. For some values of hash_size the function will probe relatively few positions in the array. For example, when hash_size is a large power of 2, approximately one sixth of the positions are probed. When hash_size is a prime number, however, quadratic probing reaches half the locations in the array.

To prove this observation, suppose that hash_size is a prime number. Also suppose that we reach the same location at probe i and at some later probe that we can take as $i + j$ for some integer $j > 0$. Suppose that j is the smallest such integer. Then the values calculated by the function at i and at $i + j$ differ by a multiple of hash_size. In other words,

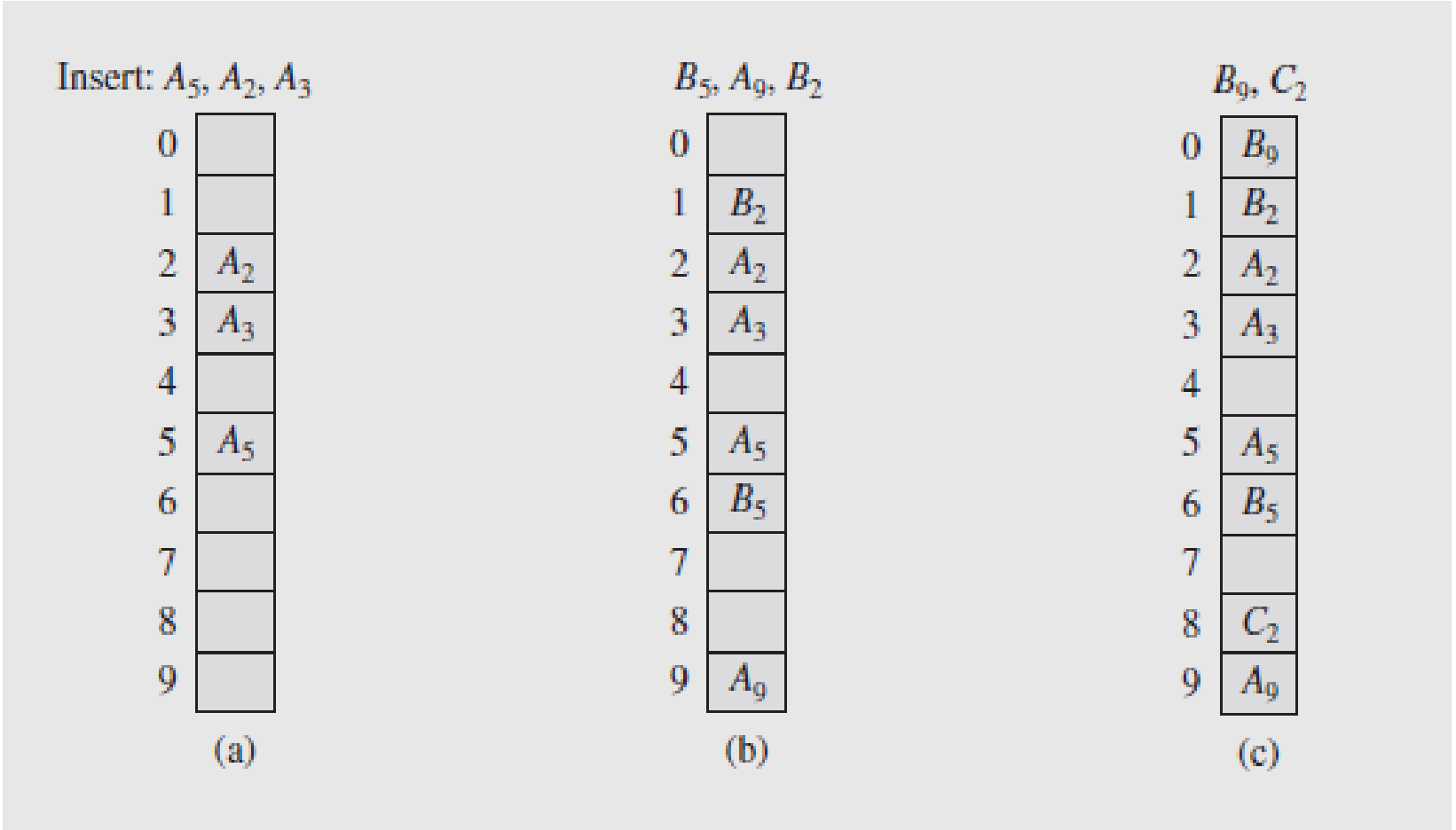
$$h + i^2 \equiv h + (i + j)^2 \text{ (mod hash\_size)}.$$

Variante propuesta por Drozdek: $p(i) = h(K) + (-1)^{i-1}((i + 1)/2)^2$ for $i = 1, 2, \ldots, TSize - 1$

Open adressing

Closed hashing

Using quadratic probing for collision resolution.



Variante propuesta por Drozdek: $p(i) = h(K) + (-1)^{i-1}((i+1)/2)^2$ for $i = 1, 2, \ldots, TSize - 1$

Open adressing

Closed hashing

Resolución de colisiones aplicando doble hashing

The problem of secondary clustering is best addressed with *double hashing*. This method utilizes two hash functions, one for accessing the primary position of a key, h, and a second function, h_p, for resolving conflicts. The probing sequence becomes

$$h(K), h(K) + h_p(K), \ldots, h(K) + i \cdot h_p(K), \ldots$$

(all divided modulo *TSize*). The table size should be a prime number so that each position in the table can be included in the sequence. Experiments indicate that secondary clustering is generally eliminated because the sequence depends on the values of h_p, which, in turn, depend on the key. Therefore, if the key K_1 is hashed to the position j, the probing sequence is

$$j, j + h_p(K_1), j + 2 \cdot h_p(K_1), \ldots$$

(all divided modulo *TSize*).

Open adressing

Closed hashing

FIGURE 10.3 Formulas approximating, for different hashing methods, the average numbers of trials for successful and unsuccessful searches (Knuth 1998).

	linear probing	quadratic probing[a]	double hashing
successful search	$\frac{1}{2}\left(1 + \frac{1}{1 - \text{LF}}\right)$	$1 - \ln(1 - \text{LF}) - \frac{\text{LF}}{2}$	$\frac{1}{\text{LF}} \ln \frac{1}{1 - \text{LF}}$
unsuccessful search	$\frac{1}{2}\left(1 + \frac{1}{(1 - \text{LF})^2}\right)$	$\frac{1}{1 - \text{LF}} - \text{LF} - \ln(1 - \text{LF})$	$\frac{1}{1 - \text{LF}}$

$$\text{Loading factor} \quad \text{LF} = \frac{\text{number of elements in the table}}{\text{table size}}$$

[a] The formulas given in this column approximate any open addressing method that causes secondary clusters to arise, and quadratic probing is only one of them.

Chaining

Open hashing

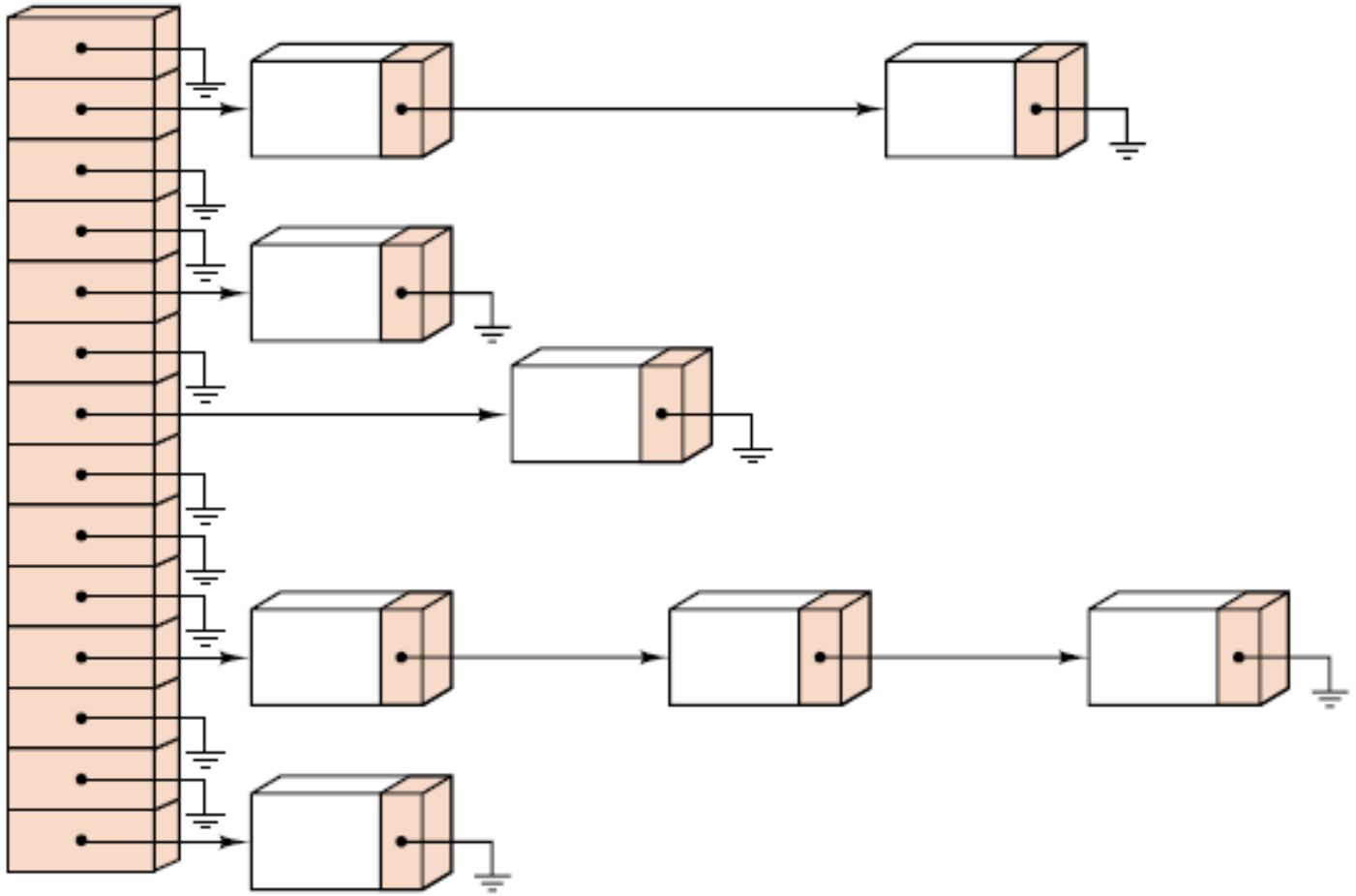


Figure 9.14. A chained hash table

Fin