TABLAS HASH

Joaquín Fernández-Valdivia
Javier Abad
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Granada

DECSAI

Hashing

- Uno de los objetivos que nos ha hecho estudiar diferentes estructuras es la eficiencia que conlleva la operación de buscar un elemento en un conjunto

- Hasta el momento la mejor eficiencia que hemos obtenido es O(log2(n)) usando dos estructuras que trabajan por comparación de valores clave:

Hashing

- Uno de los objetivos que nos ha hecho estudiar diferentes estructuras es la eficiencia que conlleva la operación de buscar un elemento en un conjunto

- Hasta el momento la mejor eficiencia que hemos obtenido es O(log2(n)) usando dos estructuras que trabajan por comparación de valores clave:

  - Vector ordenado aplicando búsqueda binaria

Hashing

- Uno de los objetivos que nos ha hecho estudiar diferentes estructuras es la eficiencia que conlleva la operación de buscar un elemento en un conjunto

- Hasta el momento la mejor eficiencia que hemos obtenido es O(log2(n)) usando dos estructuras que trabajan por comparación de valores clave:

  - Vector ordenado aplicando búsqueda binaria

  - Árbol binario de búsqueda equilibrado (p.ej. AVL)

Hashing

- Las Tablas Hash mejoran la eficiencia de la operación de búsqueda hasta O(1)
- Esta estructura se caracteriza porque intenta asignar a cada elemento del conjunto una única posición dentro de una tabla y a cada posición de la tabla un único elemento
- Al tener la tabla un espacio finito habrá elementos a los que se le asignan una misma posición en la tabla y por lo tanto se producirá una colisión
- Esta estructura prevé esta posibilidad y por lo tanto veremos distintos mecanismos para resolver estas colisiones

Hashing: idea básica

- El Hashing no opera mediante la comparación entre valores clave, sino buscando una función, h(k), que nos dé la localización exacta de la clave k en la estructura de datos en la que estén almacenadas las claves

- ¿Son fáciles de encontrar esas funciones h? **NO**

  - Si buscamos que $\forall i \neq j \Rightarrow h(i) \neq h(j)$

    - Tabla tamaño 40 y 30 claves

$40^{30} \simeq 1.15 \times 10^{48}$ posibles funciones

$40!/10! \simeq 2.25 \times 10^{41}$ no generan duplicados

¡¡¡Sólo nos servirían 2 de cada 10 millones!!!

Paradoja del Cumpleaños

- Las funciones que evitan duplicados son muy difíciles de encontrar incluso para tablas pequeñas

- Paradoja: si en una reunión están presentes 23 o más personas, hay bastante probabilidad de que dos de ellas hayan nacido el mismo día del mismo mes

- Si seleccionamos una función aleatoria que aplique 23 claves a una tabla de tamaño 365, la probabilidad de que dos claves no caigan en la misma localización es de solo 49,27%

- Para 57 claves la probabilidad es del 99,67%

Hashing: idea básica

- Los registros de datos a los que corresponden las claves suele estar almacenados en un fichero de un sistema de almacenamiento externo

- La tabla Hash actúa a modo de índice

- Nuestro objetivo será:

    - **Encontrar funciones h** (funciones hash) que generen el menor número posible de colisiones

    - **Diseñar métodos de resolución de colisiones**, cuando éstas se produzcan

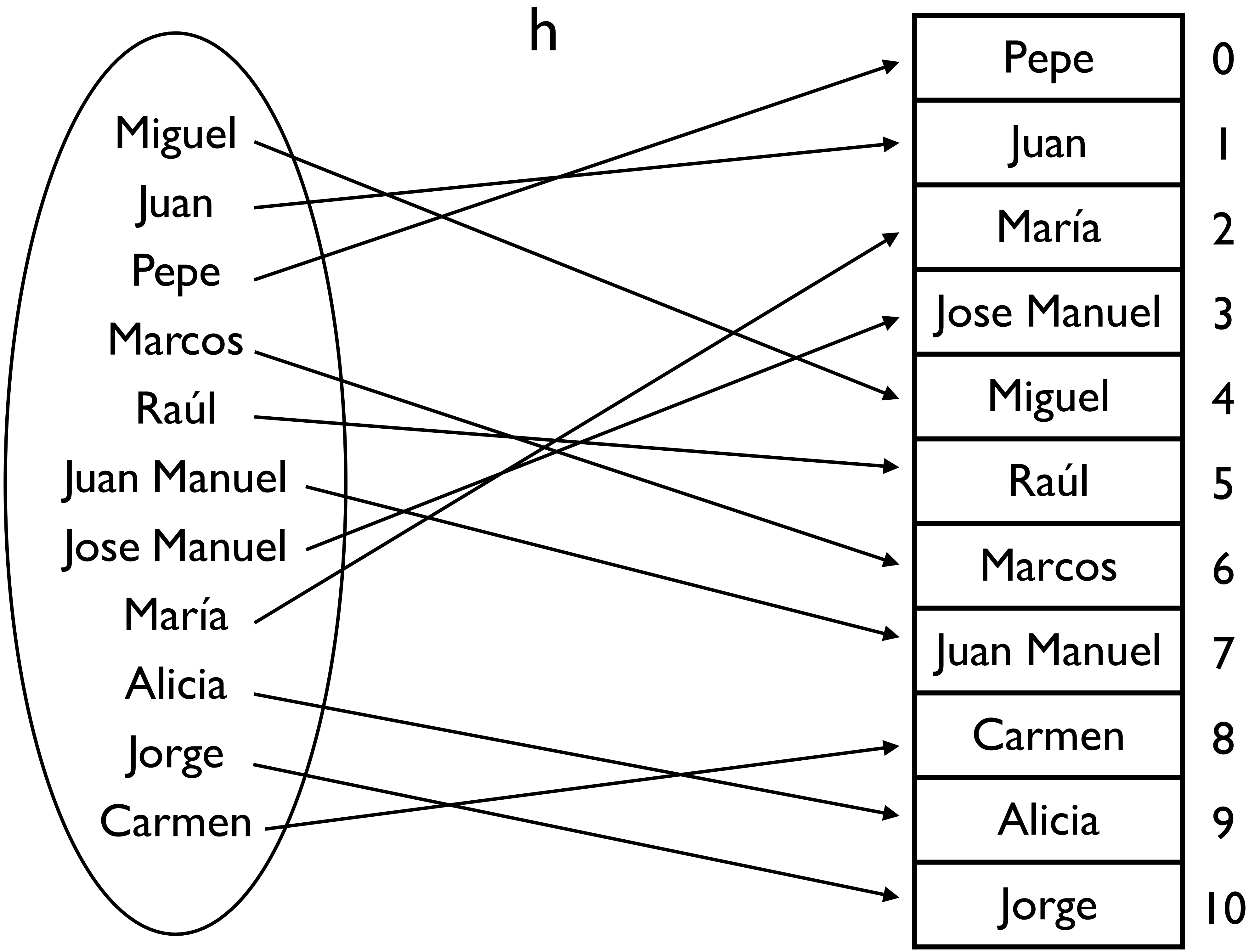
Tablas Hash

- Una **tabla Hash** es un contenedor asociativo (tipo diccionario) que permite un almacenamiento y posterior recuperación eficientes de elementos, denominados valores, a partir de otros objetos, llamados claves

- La forma ideal de realizar la búsqueda de un elemento en un contenedor sería aplicar una función matemática sobre el dato (generalmente enteros o cadenas de caracteres) y que ésta devolviera directamente el lugar en el que se encuentra. Esto sería O(1)

- A esa función se le llama **función Hash**

Tablas Hash


h
Miguel
Juan
Pepe
Marcos
Raúl
Juan Manuel
Jose Manuel
María
Alicia
Jorge
Carmen
Pepe
Juan
María
Jose Manuel
Miguel
Raúl
Marcos
Juan Manuel
Carmen
Alicia
Jorge
0
1
2
3
4
5
6
7
8
9
10

Tablas Hash

Tabla hash

h(k)	*k*	Posición dentro del fichero
0	239	*i*
1	500	*n*
	.	
	.	
	.	
n	733	1

Fichero

	k	Contenido
	.	
1	733	bla bla bla bla bla
	.	
	.	
	.	

- Para obtener la información asociada a la clave k = 733 sólo tendríamos que aplicar la función hash para obtener la dirección del fichero en la que se guarda su información, es decir h(733)=n
- Vamos a la tabla a la posición n y en esta posición consultamos la dirección en el fichero donde tenemos el resto de la información

Tablas Hash

Tabla hash

h(k)	*k*	Posición dentro del fichero
0	239	*i*
1	500	*n*
	.	
	.	
	.	
n	733	1

Fichero

	k	Contenido
	.	
1	733	bla bla bla bla bla
	.	
	.	
	.	

- La función hash cuesta calcularla O(1), lo que hace que nuestra operación de búsqueda sea O(1)
- El problema de las tablas hash es que a veces diferentes claves tiene el mismo valor hash y por lo tanto se produce colisiones
- Y en segundo lugar se debe activar mecanismos para no generar tablas muy descompensadas con respecto al número de datos que queremos almacenar

Tablas Hash

Clave

Tabla Hash

0	23121	i
1	24576	n
	. . .	
i	23396	n-1
	. . .	
n-1	22563	1
n	21456	0

Fichero

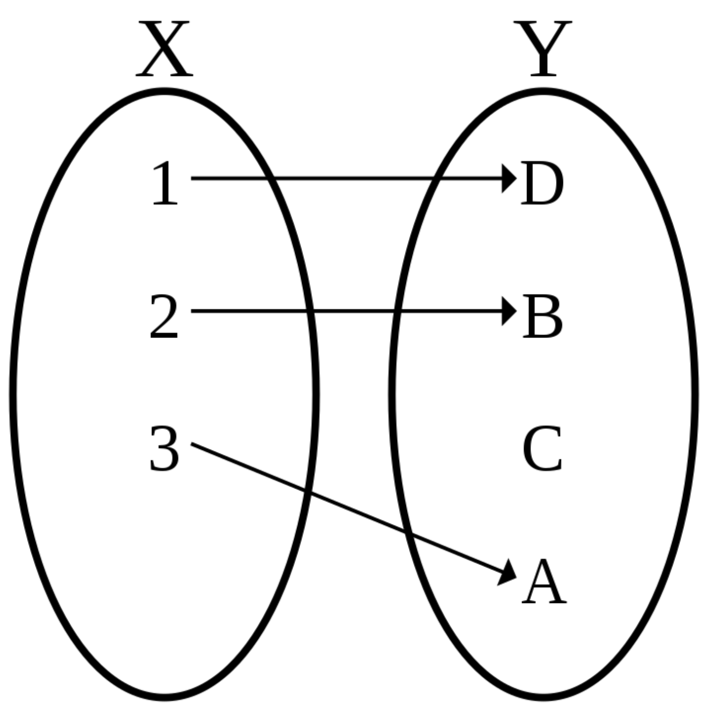
0	21456	. . . . . .
1	22563	. . . . . .
	. . .	
i	23121	. . . . . .
	. . .	
n-1	23396	. . . . . .
n	24576	. . . . . .

Clave 23396
h(23396) = i

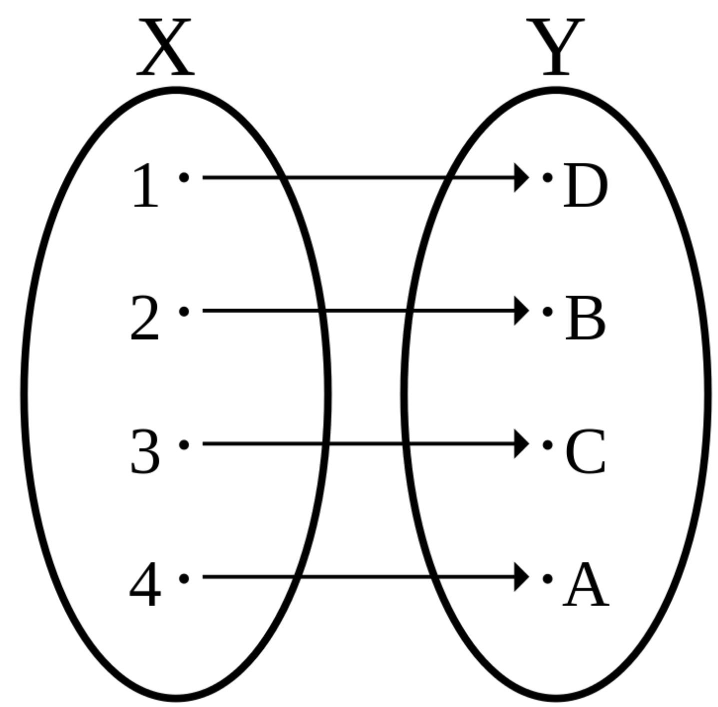
Posición en el fichero

Tablas Hash

- La función hash debería ser inyectiva. El problema es que encontrar una función así no es nada sencillo
- Cuando tenemos una función hash biyectiva decimos que tenemos una función hash perfecta. El conjunto de datos debe ser fijo y predeterminado



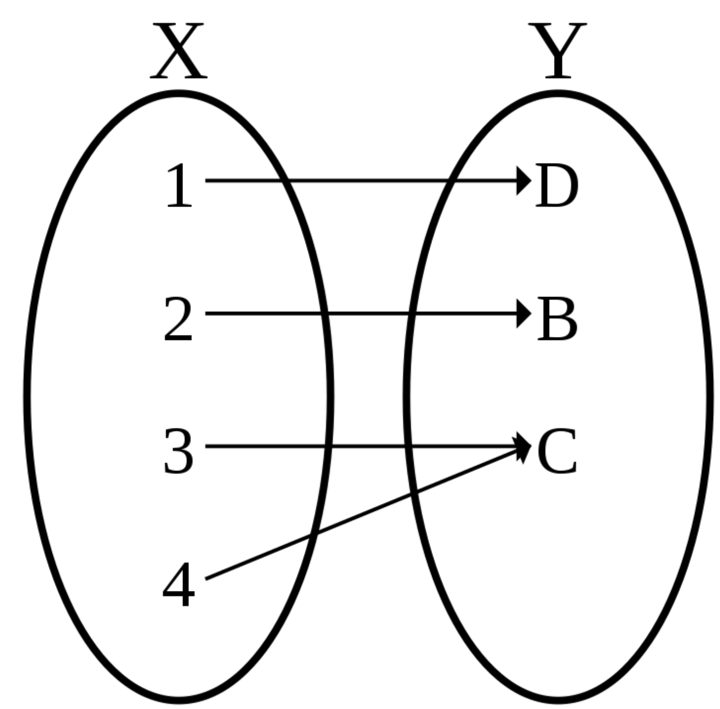
Función inyectiva



Función biyectiva

Tablas Hash

- Para el resto de casos tendremos funciones sobreyectivas, esto es, para algunas parejas de claves diferentes obtendremos el mismo valor. En este caso se producen colisiones en el valor de la función Hash
- **Colisión**: Dadas dos claves distintas, k_1 y k_2, si $h(k_1)=h(k_2)$ se produce una colisión
- Dependiendo de cómo resolvamos esas colisiones tendremos hashing abierto o cerrado



Función sobreyectiva

Esquema

- Funciones Hash
  - Truncamiento
  - Plegado
  - Multiplicación
  - Resto de la división
- Resolución de colisiones
  - Hashing abierto: Tabla Hash abierta
  - Hashing cerrado: Tabla Hash cerrada
    - Rehashing lineal
    - Rehashing doble
    - Otros

Funciones Hash

$$h: C \rightarrow Z$$

- El dominio, C, corresponde al conjunto de posibles claves
- El rango, Z, es el conjunto de enteros positivos (puede contener el 0), y corresponde al conjunto de índices sobre la tabla Hash
- La función Hash se debe definir de forma que
  - Sea rápida de calcular
  - Tome todos y cada uno de los posibles valores
  - Distribuya de forma lo más aleatoria posible las claves
  - Minimice el número de colisiones

Funciones Hash

1. **Truncamiento**: Consiste en eliminar algunos dígitos de la clave

h(123456789) = h(123<u>456789</u>) = 123

h(121567890) = h(121<u>567890</u>) = 121

- Inconveniente: la tabla Hash deberá tener un tamaño potencia de 10
- Alternativa: truncamiento a nivel interno (a nivel de bits). La tabla debe tener un tamaño potencia de 2

Funciones Hash

2. **Plegado**: Consiste en dividir una clave numérica en dos o más partes y sumarlas

$h(\overline{123}\underline{456}) = 123 + 456 = 579$

Puede modificarse para que rote algún sumando

$h(\overline{123}\underline{456}) = 123 + 654 = 777$

Puede combinarse con el truncamiento

$h(\overline{456}\underline{882}) = 456 + 882 = 1338 \Rightarrow \underline{1}338 = 338$

Puede involucrar más de 2 sumandos

$h(\overline{123}\underline{456}\overline{789}) = 123 + 456 + 789 = 1368$

- Inconveniente: El tamaño de la tabla Hash debe ser potencia de 10

Funciones Hash

3. **Multiplicación**: Similar al plegado, pero en lugar de sumas, involucra productos. Puede haber plegado antes o después del producto

Ejemplo: Tabla de tamaño 10000 y claves de 9 dígitos

$h(\underline{123}456\underline{789}) = 123 \times 789 = \underline{9}7047 \Rightarrow 7047$

- Requiere tablas de tamaño potencia de 10
- Tiende a esparcir claves ⇒ menos colisiones
- Variantes:
  - Cuadrado del centro
  - Centro del cuadrado

Funciones Hash

- Cuadrado del centro:

  Seleccionar un cierto número de cifras del centro de la clave y calcular su cuadrado [+truncamiento]

  $h(123\underline{456}789) = 7936$

  $456^2 = \underline{20}7936 \Longrightarrow 7936$

- Centro del cuadrado:

  Calcular el cuadrado de la clave y seleccionar un cierto número de cifras del centro

  $h(1234) = 1234^2 = 15\underline{2275}6 = 2275$

Funciones Hash

4. **Resto de la división**: Consiste en tomar el resto de la división de la clave entre el tamaño de la tabla (M)

$$h(k) = k \bmod M \qquad (h(k) = k\%M)$$

- Método muy simple que no requiere truncamiento

Ejemplo: h(k) = k%11

Claves: 12, 21, 68, 38, 52, 70, 44, 18

Rango: 0..10 (11 casillas)

12	21	68	38	52	70	44	18
↓	↓	↓	↓	↓	↓	↓	↓
1	10	2	5	8	4	0	7

Funciones Hash

- Consideraciones sobre el método del resto:
  - El tamaño de la tabla Hash debe ser, al menos, igual al número de claves posibles
  - La mejor elección es no tomar M simplemente par o impar, sino primo $\Longrightarrow$ M un número primo mayor que el número de claves

Funciones Hash

Cadena de caracteres

- Código ASCII

$$h(k) = (k[0] + k[1] + \ldots + k[n-1])\%M$$

- Número de caracteres (p.ej. los t primeros)

$$h(k) = (T^0 * k[0] + T^1 * k[1] + \ldots + T^t * k[t-1])\%M$$

Funciones Hash

Cadena de caracteres

Ejemplo: Alfabeto que tiene $L = 28$ caracteres, una tabla hash con 5 posiciones y la cadena *Hola*

- Vamos a usar un prefijo de 3 caracteres para la función hash y teniendo en cuenta que los códigos ASCII son:
  - H: 72
  - o: 111
  - l: 108

$$h(k) = (28^0 * 72 + 28^1 * 111 + 28^2 * 108)\%5 = 2$$

Ejemplo (sin colisiones)

M = 11

(primo más cercano a 8)

	Código	Pos
0		X
1		X
2		X
3		X
4		X
5		X
6		X
7		X
8		X
9		X
10		X

Tabla Hash

	Código	Apellidos
0	12	Abadía Ruiz
1	21	Bernabé Pérez
2	68	Carrasco Ruiz
3	38	Domingo Lucas
4	52	Fernández Sánchez
5	70	Jiménez Ruiz
6	44	Martín Pérez
7	18	Rodriguez Gómez

Fichero

Tamaño:8 registros

Ejemplo (sin colisiones)

- Funcionamiento:

Registro 0 $\equiv$ (12, Abadía Ruiz)

h(12) = 12%11 = 1

	Código	Pos
0		X
1	12	0
:	.	.

	Código	Apellidos
0	12	Abadía Ruiz
1	21	Bernabé Pérez
:	.	.

k	12	21	68	38	52	70	44	18
h(k)	1	10	2	5	8	4	0	7

Ejemplo (sin colisiones)

	Código	Pos
0	44	6
1	12	0
2	68	2
3		X
4	70	5
5	38	3
6		X
7	18	7
8	52	4
9		X
10	21	1

Tabla Hash

	Código	Apellidos
0	12	Abadía Ruiz
1	21	Bernabé Pérez
2	68	Carrasco Ruiz
3	38	Domingo Lucas
4	52	Fernández Sánchez
5	70	Jiménez Ruiz
6	44	Martín Pérez
7	18	Rodriguez Gómez

Fichero

Ejemplo: consultas

- Si queremos obtener los datos del registro con código k = 52

  a) h(52) = 52%11 = 8

  b) Accedemos a la casilla 8 de la tabla Hash

  c) Consultamos la posición del registro:4

  d) Accedemos a la posición en el fichero, recuperando la información: (52,Fernández Sánchez)

- Datos del registro con código k = 14

  a) h(14) = 14%11 = 3

  b) Casilla 3 vacía $\Longrightarrow$ registro inexistente

	Código	Pos
0	44	6
1	12	0
2	68	2
3		X
4	70	5
5	38	3
6		X
7	18	7
8	52	4
9		X
10	21	1

Tabla Hash

Colisiones

- Una colisión se da cuando dos claves diferentes tienen el mismo valor hash

- Es decir, dadas dos claves k_i y k_j siendo $k_i \mathrel{!=} k_j$ ocurre una colisión si $h(k_i) = h(k_j)$

- Otro requisito es que la tabla tenga un tamaño idóneo para los datos a almacenar
  - Una tabla muy grande con respecto al número de datos evitaría colisiones, pero desperdiciaría mucho espacio
  - Una tabla muy pequeña aprovecharía bien el espacio, pero tendrá muchas colisiones

Tratamiento de colisiones

- **Motivación:** en la práctica totalidad de los casos, las funciones Hash provocan colisiones
- **Objetivo:** encontrar un mecanismo para la clave que provoca la colisión de forma que más tarde, en una operación de consulta, la búsqueda sea eficiente
- Alternativas para resolver las colisiones: dependen de la estructura de datos elegida
- En última instancia, depende de si conocemos de antemano o no el número de elementos a ubicar en la tabla Hash (o, al menos, una estimación)

Hashing abierto

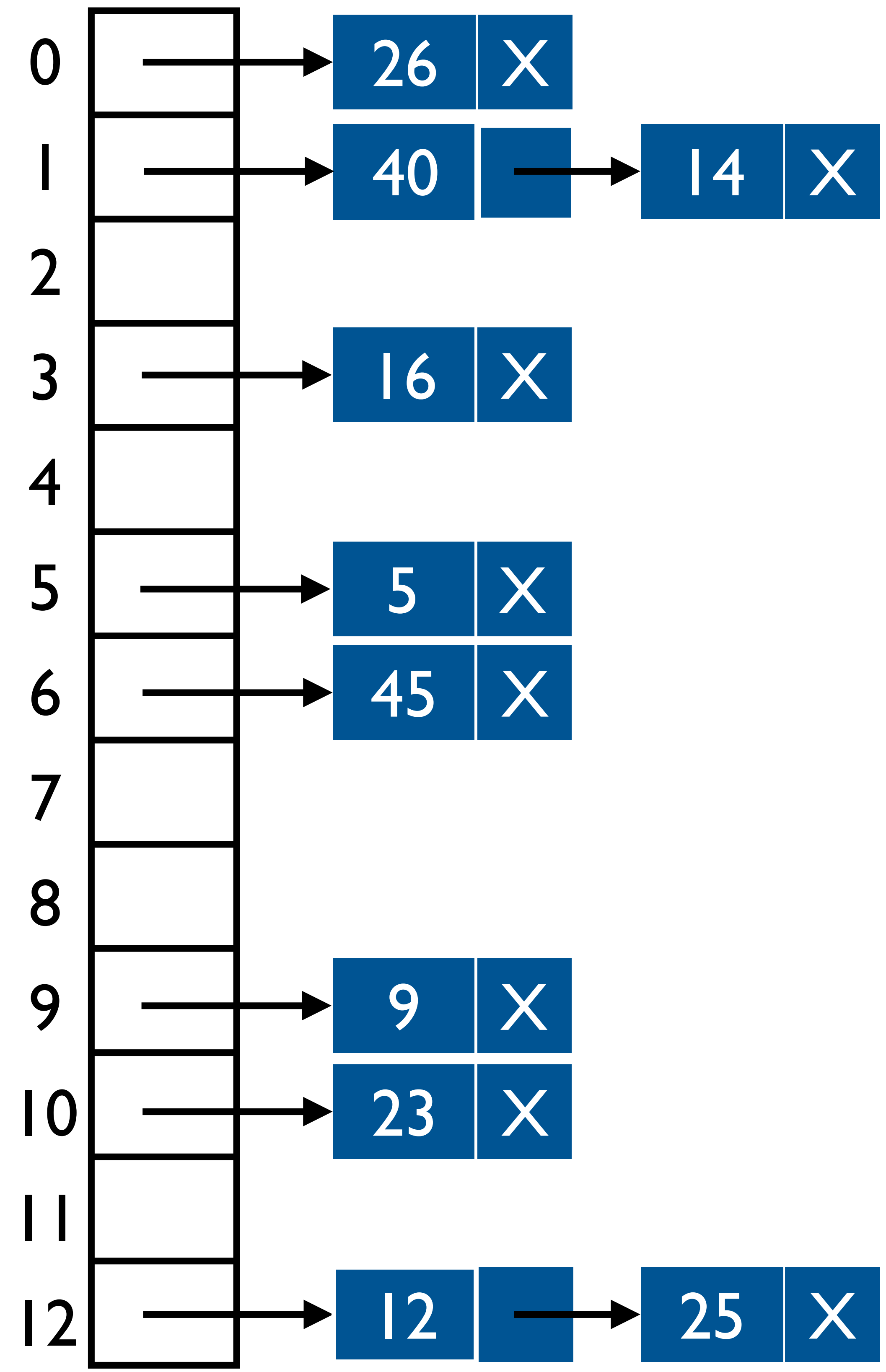
- Consiste en construir para cada índice de la tabla una **lista de claves sinónimas**
  - Cada una de estas listas puede implementarse como una lista dinámica
- El tamaño de la tabla Hash se fija a priori y suele implementarse como un vector estático de punteros a estas listas
- Ventaja: La tabla puede tener un tamaño inferior al número de claves, ya que "crece" con memoria dinámica
- Desventaja: El espacio adicional requerido por los punteros necesarios para mantener las listas y la eficiencia de las operaciones sobre las listas

Hashing abierto

- Búsqueda: calculamos el valor hash de la clave y buscamos en la lista enlazada correspondiente
  - Si la inserción es LIFO o FIFO, se debe recorrer la lista completa
  - Si se inserta de forma ordenada, se reduce, en media, el tiempo de búsqueda (aunque la inserción es más costosa)
  - Búsqueda de clave inexistente: si se llega al final de la lista correspondiente y no se encuentra un nodo con la clave buscada

Hashing abierto

- Las colisiones se resuelven insertándolas en una lista
- La ED resultante es un vector de listas
- Factor de carga: número medio de claves por lista
- Objetivo: que el factor de carga esté próximo a 1
- Ejemplo: 23,45,16,26,40,14,5,12,9,25 con h(x) = x%13



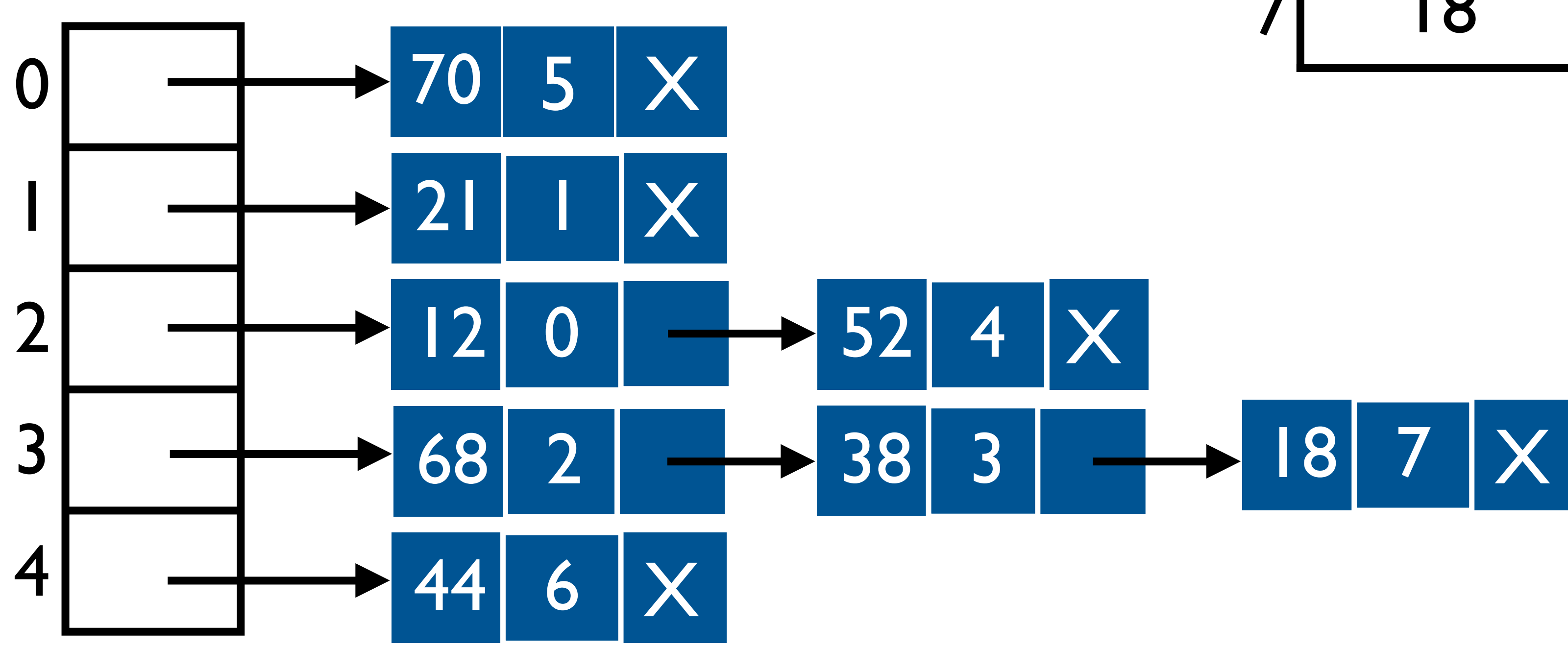
Hashing abierto

k	12	21	68	38	52	70	44	18
h(k)	2	1	3	3	2	0	4	3

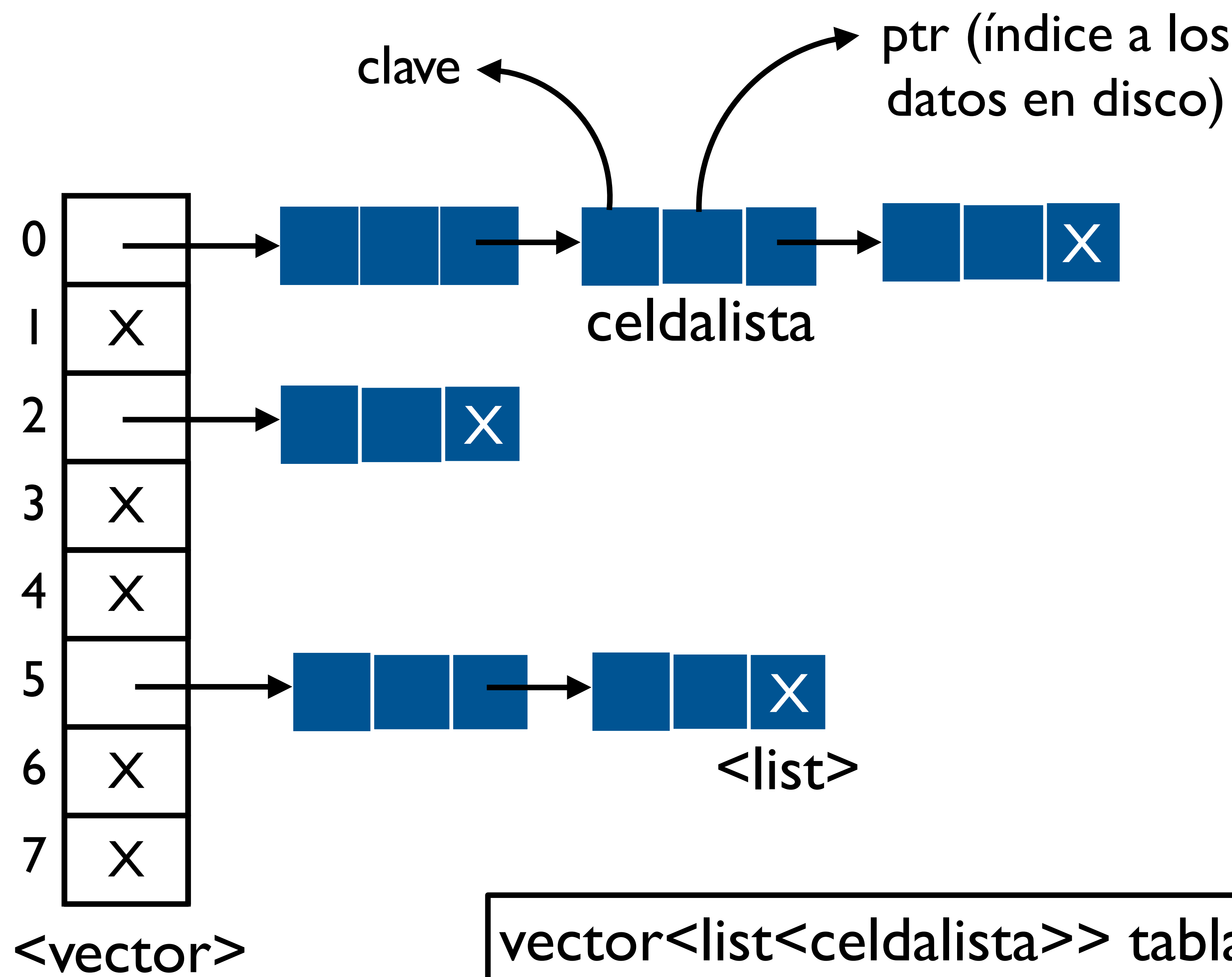
h(k) = k%M, con M = 5

	Código	Apellidos
0	12	Abadía Ruiz
1	21	Bernabé Pérez
2	68	Carrasco Ruiz
3	38	Domingo Lucas
4	52	Fernández Sánchez
5	70	Jiménez Ruiz
6	44	Martín Pérez
7	18	Rodriguez Gómez

Fichero



Clase Tabla Hash abierta



STL

```
vector<list<celdalista>> tablaHash;
```

Hashing cerrado

- Usamos un vector para alojar la tabla Hash
- **Rehashing**: Cuando se produzca colisión, la resolvemos usando una función adicional, asignándole otro valor hash a la clave hasta encontrar un hueco
- Estrategias:
  - Rehashing lineal
  - Sondeo aleatorio
  - Hashing doble

Hashing cerrado

- Las búsquedas se hacen siguiendo la misma secuencia de la función hash usada para la inserción
- ¡¡Cuidado con los borrados!! La casilla puede formar parte de una cadena de búsqueda
  - La casilla debe marcarse como **borrada**, un estado diferente al de **libre** u **ocupada**
- Diferencia entre casilla libre y borrada:
  - Inserción: borrada y libre son equivalentes (disponemos de un hueco)
  - Búsqueda: borrada y ocupada son equivalentes (seguimos el proceso de búsqueda)

Hashing cerrado. Redimensionamiento

- Redimensionamiento de la tabla Hash
  - Consiste en volver a construir la tabla Hash con un nuevo tamaño, y volver a hacer hashing (y, eventualmente, rehashing) de todas las claves de la tabla antigua (la función Hash cambia al cambiar M)
  - Debe realizarse cuando la tabla hash se desborda (se llena) o cuando su eficiencia decaiga demasiado debido a inserciones y borrados

Hashing cerrado. Rehashing lineal

- **Rehashing lineal**: $h_i(k) = [h(k) + (i-1)] \% M, \quad i=2,3...$
- Estrategia:
  - Si se evalúa h(k) para una clave k y hay colisión
  - Generamos la secuencia de valores $h_2(k), h_3(k)$... mientras se mantenga el estado de colisión
  - Cuando para un t, $h_t(k)$ no se produzca colisión, se termina la secuencia de rehashing y ubicamos la clave en $h_t(k)$
- Podemos reescribir la función de rehashing lineal como

$$\begin{cases} h_0(k) = h(k) \\ h_i(k) = [h_{i-1}(k) + 1] \% M, \quad i=2,3\ldots \end{cases}$$

Hashing cerrado. Rehashing lineal

- Ejemplo: 23,45,16,26,40,14,5,12,9,25 con h(x) = x%13

k	23	45	16	26	40	14	5	12	9	25
h(k)	10	6	3	0	1	1	5	12	9	12

	Clave	Posición	Status
0	26	pos	O
1	40	pos	O
2	14	pos	O
3	16	pos	O
4	25	pos	O
5	5	pos	O
6	45	pos	O
7			L
8			L
9	9	pos	O
10	23	pos	O
11			L
12	12	pos	O

O: Casilla ocupada
L: Casilla libre
B: Casilla borrada

Hashing cerrado. Rehashing lineal

- Ejemplo: $h(x) = x\%13$

k	Registro	h(k)
119	0	2
85	1	7
43	2	4
141	3	11
72	4	8
91	5	0
109	6	5
147	7	6
38	8	12
137	9	9
148	10	
101	11	

$h(72) = 7$

$h_2(72) = (7+(2-1))\%13 = 8$

$h(147) = 4$

$h_2(147) = (4+(2-1))\%13 = 5$

$h_3(147) = (4+(3-1))\%13 = 6$

$h(137) = 7$

$h_2(137) = (7+(2-1))\%13 = 8$

$h_3(147) = (7+(3-1))\%13 = 9$

	Clave	Posición	Status
0	91	5	O
1			L
2	119	0	O
3			L
4	43	2	O
5	109	6	O
6	147	7	O
7	85	1	O
8	72	4	O
9	137	9	O
10			L
11	141	3	O
12	38	8	O

O: Casilla ocupada
L: Casilla libre
B: Casilla borrada

Hashing cerrado. Rehashing lineal

- Ejemplo: $h(x) = x\%13$

k	Registro	h(k)
119	0	2
85	1	7
43	2	4
141	3	11
72	4	8
91	5	0
109	6	5
147	7	6
38	8	12
137	9	9
148	10	10
101	11	1

$h(148) = 5$
$h_2(148) = (5+(2-1))\%13 = 6$
$h_3(148) = (5+(3-1))\%13 = 7$
$h_4(148) = (5+(4-1))\%13 = 8$
$h_5(148) = (5+(5-1))\%13 = 9$
$h_6(148) = (5+(6-1))\%13 = 10$

$h(101) = 10$
$h_2(101) = (10+(2-1))\%13 = 11$
$h_3(101) = (10+(3-1))\%13 = 12$
$h_4(101) = (10+(4-1))\%13 = 0$
$h_5(101) = (10+(5-1))\%13 = 1$

	Clave	Posición	Status
0	91	5	O
1	101	11	O
2	119	0	O
3			L
4	43	2	O
5	109	6	O
6	147	7	O
7	85	1	O
8	72	4	O
9	137	9	O
10	148	10	O
11	141	3	O
12	38	8	O

O: Casilla ocupada
L: Casilla libre
B: Casilla borrada

Hashing cerrado. Rehashing lineal

- Ejemplo: h(x) = x%13

k	Registro	h(k)	Rendimiento
119	0	2	1
85	1	7	1
43	2	4	1
141	3	11	1
72	4	8	2
91	5	0	1
109	6	5	1
147	7	6	3
38	8	12	1
137	9	9	3
148	10	10	6
101	11	1	5
			26

	Clave	Posición	Status
0	91	5	O
1	101	11	O
2	119	0	O
3			L
4	43	2	O
5	109	6	O
6	147	7	O
7	85	1	O
8	72	4	O
9	137	9	O
10	148	11	O
11	141	3	O
12	38	8	O

O: Casilla ocupada
L: Casilla libre
B: Casilla borrada

Hashing cerrado. Rehashing lineal

- El rehashing lineal tiende a crear agrupaciones primarias
- Una **agrupación primaria** es una sucesión de casillas ocupadas en una tabla Hash a distancia 1 (contiguas)
- Las agrupaciones primarias conllevan largas series de búsqueda que degradan la eficiencia de las inserciones y los borrados

Inserción de las cuatro primeras claves:

0	1	$\boxed{2}$	3	$\boxed{4}$	5	6	$\boxed{7}$	8	9	10	$\boxed{11}$	12
		↑		↑			↑				↑	
		119		43			85				141	

Hashing cerrado. Rehashing lineal

Inserción de la clave 72:

0 1 $\boxed{2}$ 3 $\boxed{4}$ 5 6 $\boxed{7}$ 8 9 10 $\boxed{11}$ 12

72

0 1 $\boxed{2}$ 3 $\boxed{4}$ 5 6 $\boxed{7 \quad 8}$ 9 10 $\boxed{11}$ 12

Inserción de las claves 91 y 109:

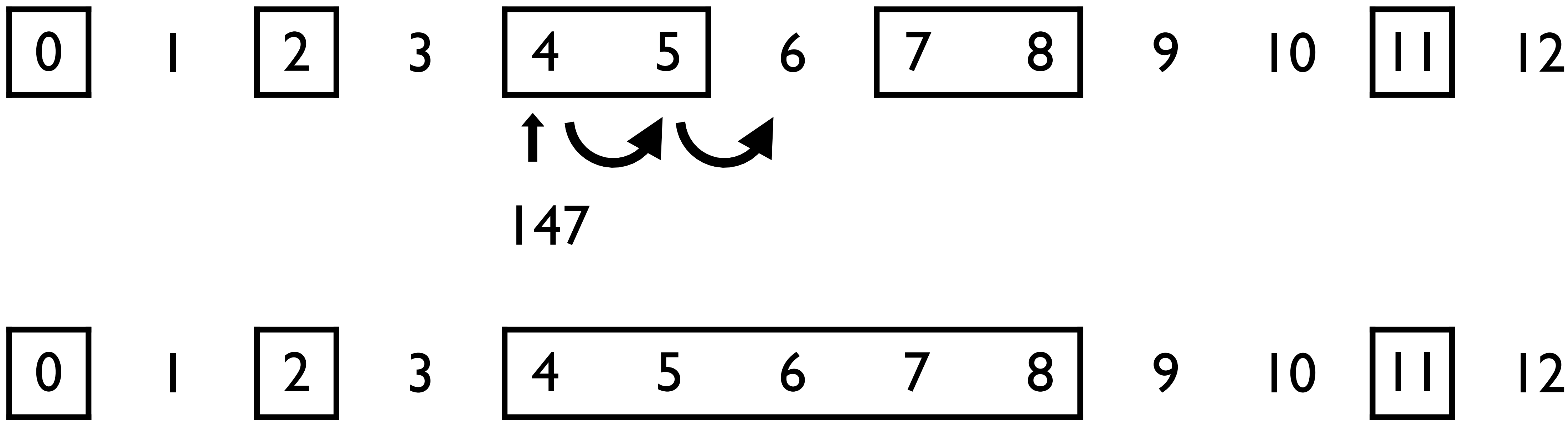
0 1 $\boxed{2}$ 3 $\boxed{4}$ 5 6 $\boxed{7 \quad 8}$ 9 10 $\boxed{11}$ 12

91 109

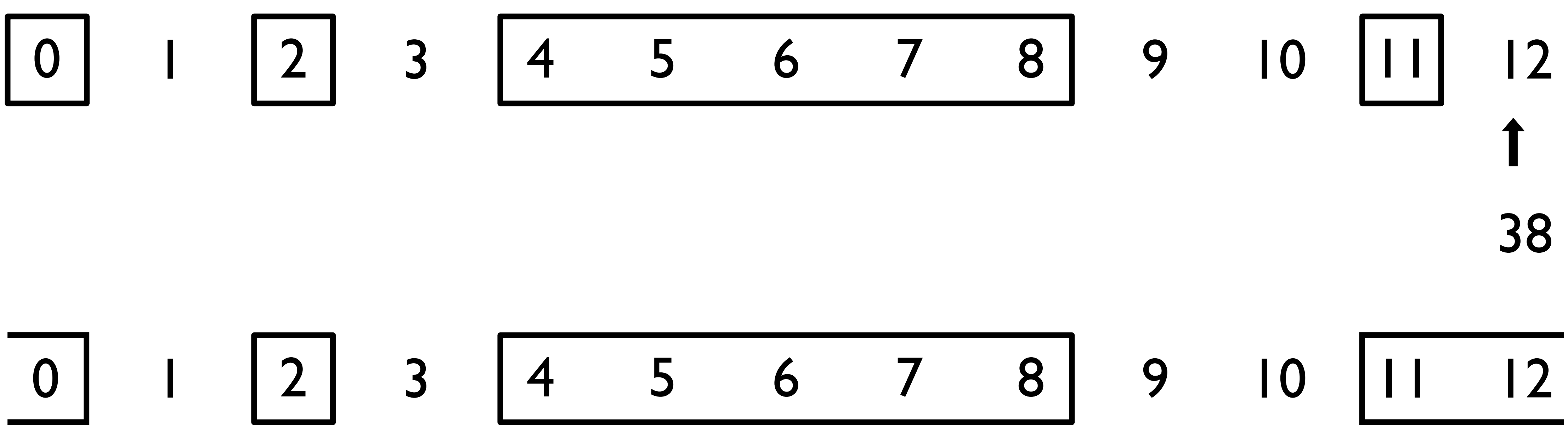
$\boxed{0}$ 1 $\boxed{2}$ 3 $\boxed{4 \quad 5}$ 6 $\boxed{7 \quad 8}$ 9 10 $\boxed{11}$ 12

Hashing cerrado. Rehashing lineal

Inserción de la clave 147:

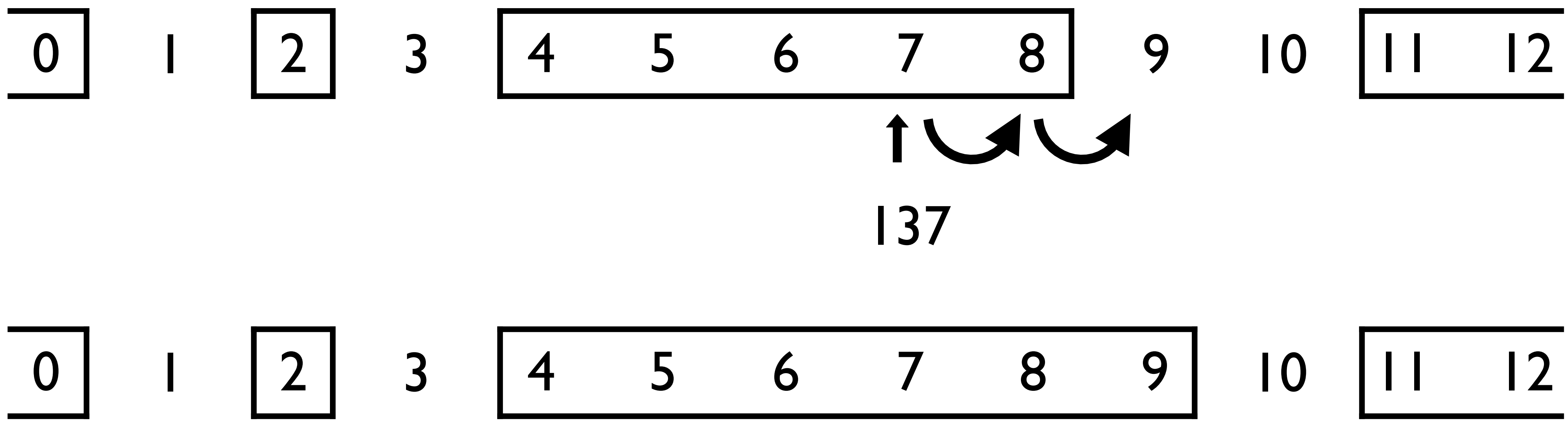


Inserción de la clave 38:

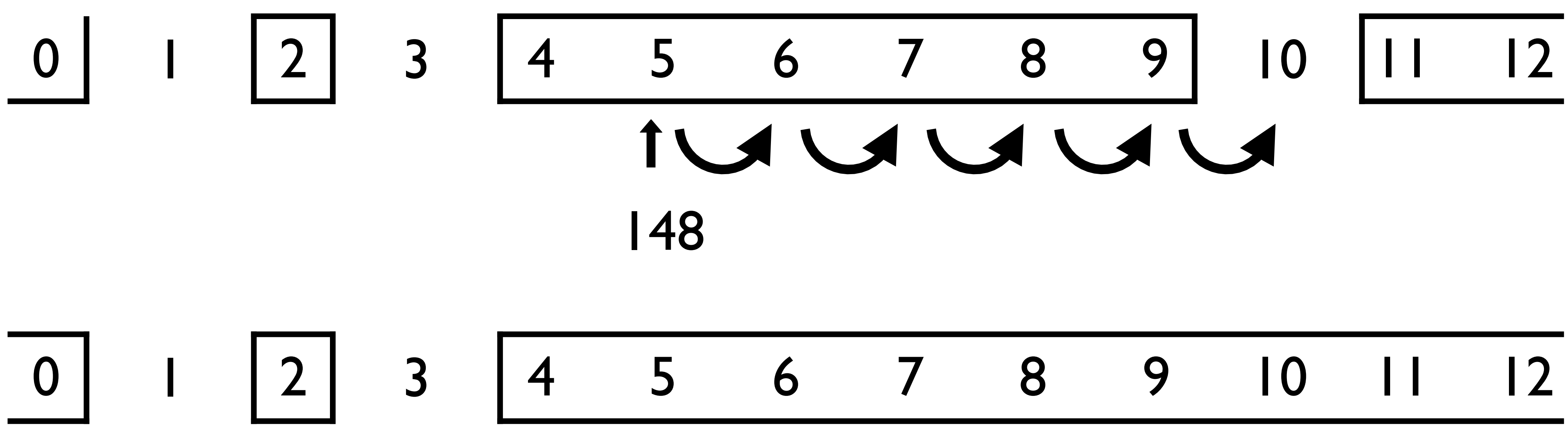


Hashing cerrado. Rehashing lineal

Inserción de la clave 137:

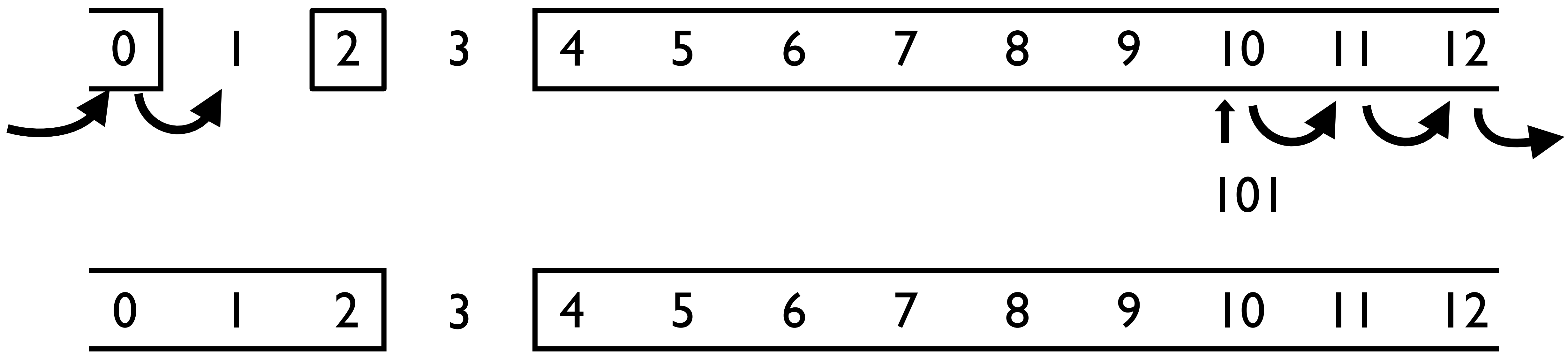


Inserción de la clave 148:



Hashing cerrado. Rehashing lineal

Inserción de la clave 101:



- Soluciones ante la aparición de agrupaciones primarias:
  - Mantener estructuras de datos auxiliares que mantengan información (inicio y fin) de las agrupaciones primarias, de forma que se pueda acceder directamente a los "huecos"
  - Buscar otros métodos que distribuyan las casillas vacías de forma más aleatoria (al fin y al cabo, la idea del Hashing es la distribución "aleatoria" de claves)

Rehashing lineal. Algoritmo de búsqueda

1. Calcular h(k)
2. Si (no borrada(h(k)) && clave(h(k)) == k)

   posicion = registro(h(k))

   Si no,

     Repetir

       $h_i(k)$ = rehashing($h_{i-1}(k)$)

     hasta que (no borrada(h(k)) && (clave($h_i(k)$)=k || vacia($h_i(k)$)))

   Si (clave($h_i(k)$) == k)

     posicion = registro($h_i(k)$)

   Si no, posicion = -1

3. Devolver (posicion)

Hashing cerrado. Sondeo aleatorio

$$h_i(k) = (h(k) + (i-1) * C) \% M, \quad i=2,3,...$$

$$h_i(k) = [h_{i-1}(k) + C] \% M, \quad i=2,3,...$$

donde

- h(k) es el valor de la función Hash
- M es el tamaño de la tabla
- C>1 y es primo relativo (no tener factores en común) con M

Hashing cerrado. Sondeo aleatorio

- Ejemplo: $h(x) = x\%13$

k	Registro	h(k)
119	0	2
85	1	7
43	2	4
141	3	11
72	4	12
91	5	0
109	6	5
147	7	9
38	8	1
137	9	
148	10	
101	11	

$h(72) = 7$

$h_2(72) = (7+(2-1) * 5)\%13 = 12$

$h(147) = 4$

$h_2(147) = (4+(2-1) * 5)\%13 = 9$

$h(38) = 12$

$h_2(38) = (12+(2-1) * 5)\%13 = 4$

$h_3(38) = (12+(3-1) * 5)\%13 = 9$

$h_4(38) = (12+(4-1) * 5)\%13 = 1$

	Clave	Posición	Status
0	91	5	O
1	38	8	O
2	119	0	O
3			L
4	43	2	O
5	109	6	O
6			L
7	85	1	O
8			L
9	147	7	O
10			L
11	141	3	O
12	72	4	O

O: Casilla ocupada
L: Casilla libre
B: Casilla borrada

Hashing cerrado. Sondeo aleatorio

- Ejemplo: $h(x) = x\%13$

k	Registro	h(k)
119	0	2
85	1	7
43	2	4
141	3	11
72	4	12
91	5	0
109	6	5
147	7	9
38	8	1
137	9	6
148	10	10
101	11	

$h(137) = 7$

$h_2(137) = (7+(2-1) * 5)\%13 = 12$

$h_3(137) = (7+(3-1) * 5)\%13 = 4$

$h_4(137) = (7+(4-1) * 5)\%13 = 9$

$h_5(137) = (7+(5-1) * 5)\%13 = 1$

$h_6(137) = (7+(6-1) * 5)\%13 = 6$

$h(148) = 5$

$h_2(148) = (5+(2-1) * 5)\%13 = 10$

	Clave	Posición	Status
0	91	5	O
1	38	8	O
2	119	0	O
3			L
4	43	2	O
5	109	6	O
6	137	9	O
7	85	1	O
8			L
9	147	7	O
10	137	9	O
11	141	3	O
12	72	4	O

O: Casilla ocupada
L: Casilla libre
B: Casilla borrada

Hashing cerrado. Sondeo aleatorio

- Ejemplo: h(x) = x%13

k	Registro	h(k)
119	0	2
85	1	7
43	2	4
141	3	11
72	4	12
91	5	0
109	6	5
147	7	9
38	8	1
137	9	6
148	10	10
101	11	3

h(101) = 10
h2(101) = (10+(2-1) * 5)%13 = 2
h3(101) = (10+(3-1) * 5)%13 = 7
h4(101) = (10+(4-1) * 5)%13 = 12
h5(101) = (10+(5-1) * 5)%13 = 4
h5(101) = (10+(5-1) * 5)%13 = 9
h5(101) = (10+(5-1) * 5)%13 = 1
h5(101) = (10+(5-1) * 5)%13 = 6
h5(101) = (10+(5-1) * 5)%13 = 11
h6(101) = (10+(6-1) * 5)%13 = 3

	Clave	Posición	Status
0	91	5	O
1	38	8	O
2	119	0	O
3	101	11	O
4	43	2	O
5	109	6	O
6	137	9	O
7	85	1	O
8			L
9	147	7	O
10	137	9	O
11	141	3	O
12	72	4	O

O: Casilla ocupada
L: Casilla libre
B: Casilla borrada

Hashing cerrado. Sondeo aleatorio

- Ejemplo: h(x) = x%13

k	Registro	h(k)	Rendimiento	
119	0	2	1	
85	1	7	1	
43	2	4	1	
141	3	11	1	
72	4	12	2	
91	5	0	1	
109	6	5	1	
147	7	9	2	
38	8	1	4	
137	9	6	6	
148	10	10	2	
101	11	3	10	33

Problema:
Agrupaciones secundarias de orden C

	Clave	Posición	Status
0	91	5	O
1	38	8	O
2	119	0	O
3	101	11	O
4	43	2	O
5	109	6	O
6	137	9	O
7	85	1	O
8			L
9	147	7	O
10	137	9	O
11	141	3	O
12	72	4	O

Hashing cerrado. Rehashing doble

$h_i(k) = (h_{i-1}(k) + h_0(k))$ % $M \quad i = 2, 3, \ldots$

$h_0(k) = 1 + (k\%(M-2))$

$h_1(k) = h(k)$

- Puede haber otras elecciones de $h_0(k)$, siempre que no sea constante y distinta de 0
- Buena cuando M y M-2 son primos relativos

Hashing cerrado. Rehashing doble

- Ejemplo: $h(x) = x\%13$

k	Reg	$h_1(k)$	$h_0(k)$
119	0	2	10
85	1	7	9
43	2	4	11
141	3	11	10
72	4	7	7
91	5	0	4
109	6	5	11
147	7	4	5
38	8	12	6
137	9	7	6
148	10	5	6
101	11	10	3

$h(119) = 2$
$h(85) = 7$
$h(43) = 4$
$h(141) = 11$
$h(72) = 7$
$h_2(72) = (h_1(72)+h_0(72))\%13=$
$= (7+7)\%13 = 1$
$h(91) = 0$
$h(109) = 5$
$h(147) = 4$
$h_2(147) = (h_1(147)+h_0(147))\%13 =$
$= (4+5)\%13 = 9$
$h(38) = 12$

O: Casilla ocupada
L: Casilla libre
B: Casilla borrada

	Clave	Posición	Status
0	91	5	O
1	72	4	O
2	119	0	O
3			L
4	43	2	O
5	109	6	O
6			L
7	85	1	O
8			L
9	147	7	O
10			L
11	141	3	O
12	38	8	O

Hashing cerrado. Rehashing doble

- Ejemplo: $h(x) = x\%13$

k	Reg	$h_1(k)$	$h_0(k)$
119	0	2	10
85	1	7	9
43	2	4	11
141	3	11	10
72	4	7	7
91	5	0	4
109	6	5	11
147	7	4	5
38	8	12	6
137	9	7	6
148	10	5	6
101	11	10	3

$h(137) = 7$

$h_2(137) = (h_1(137)+h_0(137))\%13 = (7+6)\%13 = 0$

$h_3(137) = (h_2(137)+h_0(137))\%13 = (0+6)\%13 = 6$

$h(148) = 5$

$h_2(148) = (h_1(148)+h_0(148))\%13 = (5+6)\%13 = 11$

$h_3(148) = (h_2(148)+h_0(148))\%13 = (11+6)\%13 = 4$

$h_4(148) = (h_3(148)+h_0(148))\%13 = (4+6)\%13 = 10$

$h(101) = 10$

$h_2(101) = (h_1(101)+h_0(101))\%13 = (10+3)\%13 = 0$

$h_3(101) = (h_2(101)+h_0(101))\%13 = (0+3)\%13 = 3$

	Clave	Posición	Status
0	91	5	O
1	72	4	O
2	119	0	O
3	101	11	O
4	43	2	O
5	109	6	O
6	137	9	O
7	85	1	O
8			L
9	147	7	O
10	148	10	O
11	141	3	O
12	38	8	O

Hashing cerrado. Rehashing doble

- Ejemplo: $h(x) = x\%13$

k	Reg	$h_1(k)$	$h_0(k)$	Rendimiento
119	0	2	10	1
85	1	7	9	1
43	2	4	11	1
141	3	11	10	1
72	4	7	7	2
91	5	0	4	1
109	6	5	11	1
147	7	4	5	2
38	8	12	6	1
137	9	7	6	3
148	10	5	6	4
101	11	10	3	3
				21

	Clave	Posición	Status
0	91	5	O
1	72	4	O
2	119	0	O
3	101	11	O
4	43	2	O
5	109	6	O
6	137	9	O
7	85	1	O
8			L
9	147	7	O
10	148	10	O
11	141	3	O
12	38	8	O

Tablas Hash STL

La STL define las tablas hash como las clases:

- *unordered_set, unordered_multiset*:

  Se usan cuando se quiere almacenar un conjunto de claves

  Los accesos por clave se hacen muy rápidos

  Si se admiten claves repetidas se usará un *unordered_multiset*, en caso de que no se admita claves repetidas se usará un *unordered_set*

- *unordered_map, unordered_multimap*

  Se usan para almacenar de nuevo un conjunto de claves que tiene una información asociada a la clave

  Si se admite claves repetidas se debe usar *unordered_multimap*, en otros casos *unordered_map*

Ejemplo

Algoritmo Karp-Rabin

Este algoritmo pretende encontrar si un texto contiene una cadena

Ejemplo

Algoritmo Karp-Rabin

Este algoritmo pretende encontrar si un texto contiene una cadena

```
int Fuerza_Bruta(const string & texto, const string &cadena){
   int n = texto.size();
   int m = cadena.size();
   for (int i=0;i<n-m+1;i++){
    bool seguir=true;
    for (int j=0;j<m && !seguir; j++){
      if (texto[i+j]!=cadena[j])
        seguir =false;
     }
     if (seguir)
     return i;
   }
   return -1;
}
```

Ejemplo

Algoritmo Karp-Rabin

Se basa en el hecho de comparar el trozo del texto correspondiente y la cadena

Si el trozo del texto que se analiza y la cadena tienen la misma función hash se pasa a analizar sin son iguales carácter a carácter

En caso de que no sea así no se pierde tiempo haciendo el for que recorre para j en el algoritmo de la fuerza bruta

Ejemplo

Algoritmo Karp-Rabin

```
#include <iostream>
#include <fstream>
#include <string>
#include <unordered_set>
using namespace std;
typedef unordered_set<string> stringset;

int Karp_Rabin(const string & texto, const string & cadena){
   stringset myconj;
   //obtenemos la funcion hash para string
   stringset::hasher fn = myconj.hash_function();
   int n= texto.size();
   int m =cadena.size();
   int hp = fn(cadena); //obtenemos el valor hash de cadena
   int hs= fn(texto.substr(0,m)); //funcion hash del trozo de texto
   for (int i=0;i<n-m+1;++i){
       hs =fn(texto.substr(i,m));
      if (hp==hs){ //ahora comparamos
         if (texto.substr(i,m)==cadena)
           return i;
      }
   }
   return -1;
}
```

Ejemplo

Definir una función hash para un TDA Diccionario de forma que se aplique la función hash sobre los primeros *len* elementos de la clave

Ejemplo

Definir una función hash para un TDA Diccionario de forma que se aplique la función hash sobre los primeros *len* elementos de la clave

$$fh(clave) = factor^0 * clave[0] + factor^1 * clave[1] + \ldots + factor^{len-1} * clave[len-1]$$

Ejemplo

Definir una función hash para un TDA Diccionario de forma que se aplique la función hash sobre los primeros *len* elementos de la clave

```
template <class T,class U>
class Diccionario{
  private:
    //funcion hash
    class my_hash{
      private:
          //numero de elementos sobre los que calcular la funcion hash
          unsigned int len;
          //razon para pasar de un elemento a otro
          unsigned int factor;
      public:
        //Constructor
```

Ejemplo

```
        my_hash(unsigned int l=3,unsigned int f=28):len(l),factor(f){}

        //modifica len y factor
        void set(int l,int f){
            len=l; factor=f;

        }
        //devuelve el valor hash de la clave usando solamente los len primero
        //valores de clave
        size_t operator()(const T & clave) const{
          size_t s=0;
          int ff=1;
          for (int i=0;i<len;i++){
            s=(int)(s+ff*clave[i]);
            ff*=factor;
          }
          return s;
        }

    };

    unordered_map<T,U,my_hash> datos;
    ....
};
```