

Estructuras de Datos.

Grado en Ingeniería Informática, Ingeniería del Software e Ingeniería de Computadores

ETSI Informática

Universidad de Málaga

5. Tablas Hash

José E. Gallardo, Francisco Gutiérrez, Pablo López
Dpto. Lenguajes y Ciencias de la Computación
Universidad de Málaga

Índice

- Hashing (dispersión)
 - Colisiones
- Funciones Hash (7-20)
 - Hash modular
 - Hash por congruencia lineal
 - Hash para flotantes
 - Hash polinómico
 - hashCode en Java
- Resolución de colisiones (21-54)
 - Encadenamiento separado (Separate Chaining)
 - Factor de carga, rendimiento y reubicación (rehashing)
 - Direccionamiento abierto (Open addressing). Prueba lineal
- Bibliografía básica: *Algorithms*, Robert Sedgewick and Kevin Wayne, Addison-Wesley (2011) (& 3.4, páginas 458-)

Hashing (Asociación)

- Permite almacenar N elementos que posteriormente serán localizados a través de un acceso rápido.

- Cada elemento se identifica con una clave (key)

- La solución ideal es encontrar una función

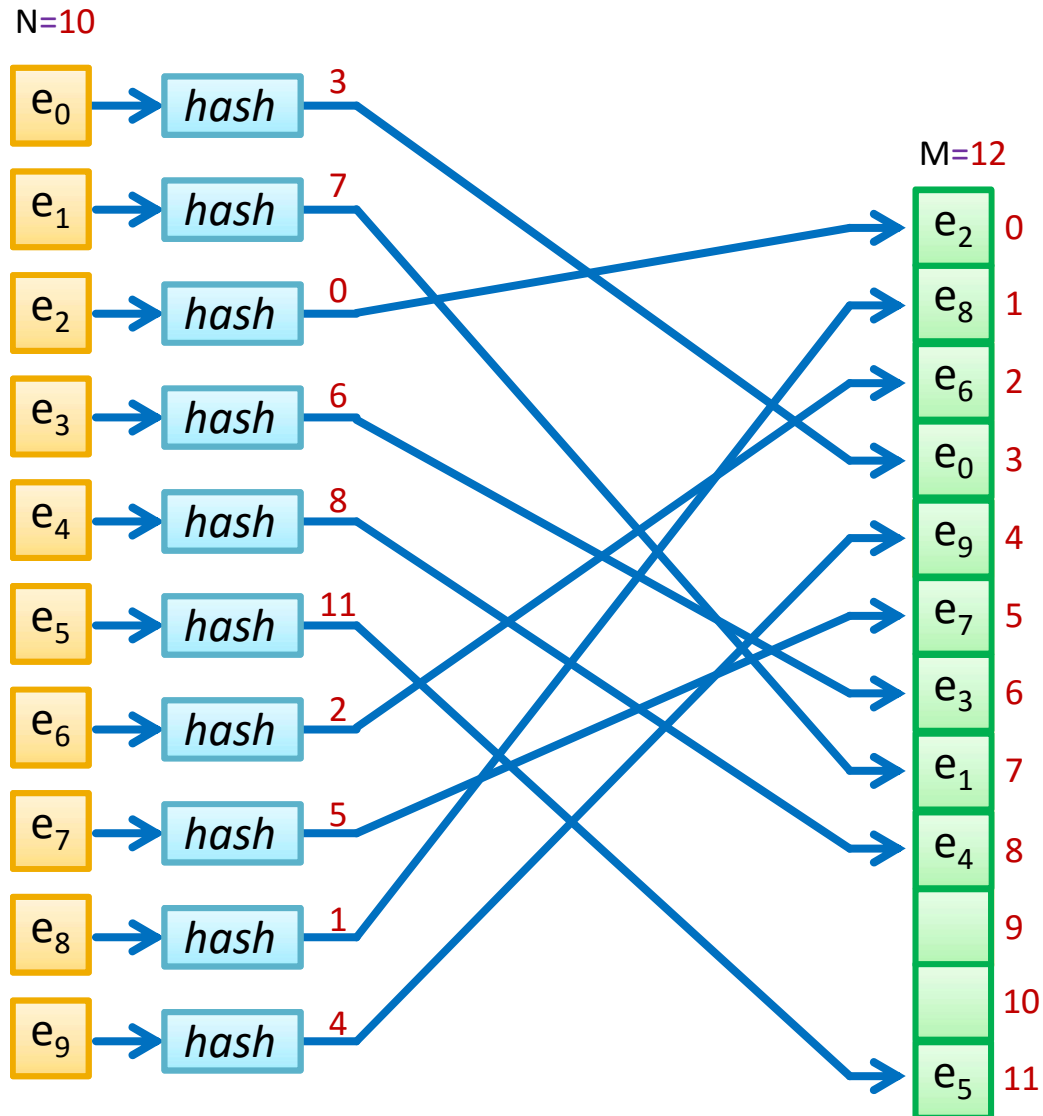
$hash :: Key \rightarrow \{0..M-1\}$

que asocia a cada clave un valor diferente del rango $[0..M-1]$.

Así, guardamos los elementos en un array de longitud M y el acceso a cada elemento estará en $O(1)$ 😊

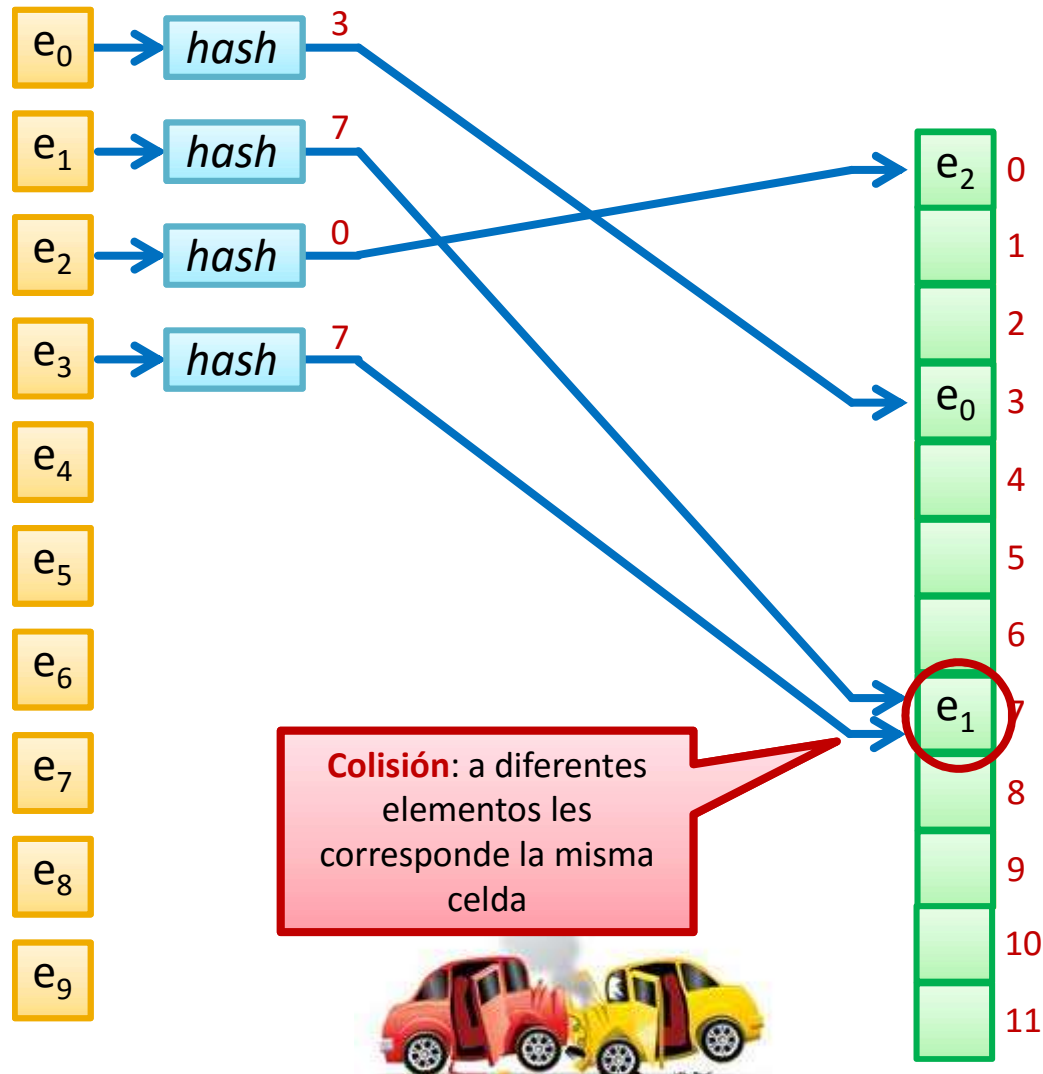
- El elemento e_i será guardado en la posición $hash(e_i)$

Hashing (II)



- Esta es una **función hash perfecta**: Cada elemento se localiza en una posición diferente del array 😊
- Estas funciones son difíciles de encontrar 😞

Colisiones

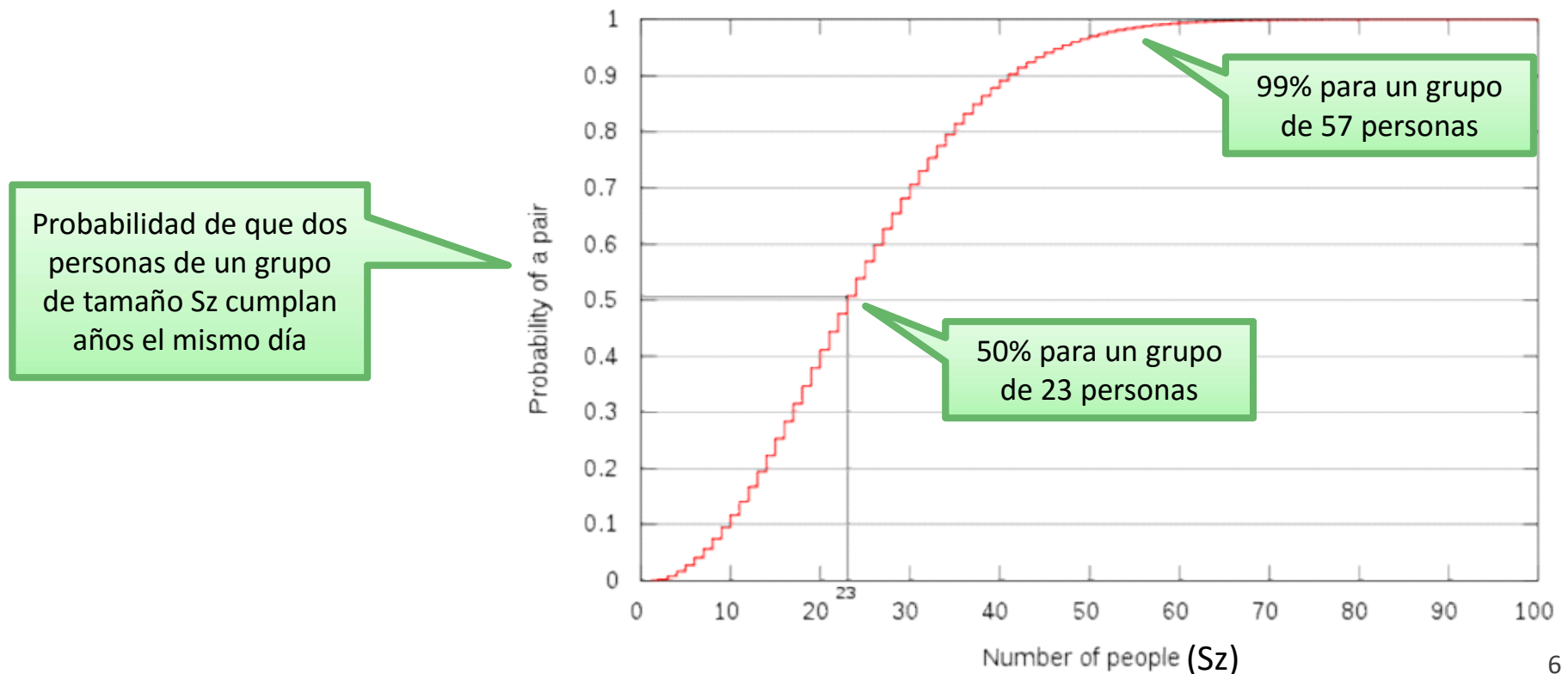


- En la práctica las funciones hash no son perfectas
- Distintos elementos tienen el mismo valor hash (**colisión**)
- Aceptamos esta posibilidad y consideraremos

Técnicas para resolver las colisiones

Colisiones (II)

- A través de un array con M celdas, siendo $M > N$ (N = número de elementos), podemos disminuir la probabilidad de colisiones.
- Aún así, será inevitable un pequeño número de colisiones, aunque M sea mucho mayor que N ([paradoja del cumpleaños](#))



Funciones Hash

- Una función *hash* transforma una clave en un índice:
 $hash :: \text{Key} \rightarrow \{0..M-1\}$
- M es el número de celdas del array
- Cada clave puede ser:
 - un entero, un flotante, un string, un objeto compuesto, etc.
- Necesitaremos distintas funciones hash para tipos de claves diferentes

Funciones Hash (II)

- Requisito principal:

- Dos claves iguales tendrán asociados el mismo valor hash:

$$k_1 = k_2 \Rightarrow hash(k_1) = hash(k_2)$$

hash es una función pura



- Propiedades adicionales:

- **Coste bajo:** *hash* debe ser computada eficientemente
- **Uniforme:** el comportamiento de las tablas hash se degrada cuando los elementos a memorizar no están distribuidos uniformemente sobre el conjunto de celdas
 - Debe asignarse un valor *hash* lo más uniformemente posible
 - los valores hash deben ser generados con la misma probabilidad

Hash Modular (I)

- Es adecuado cuando las claves son enteros

- $hash(key) = key \% M$

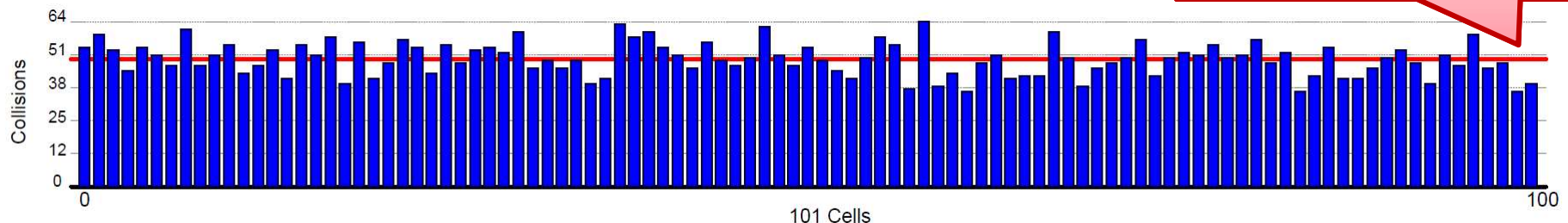
M es el tamaño
de la tabla

- Su cómputo es muy eficiente
- Si las claves no están distribuidas uniformemente los valores hash **se distribuirán mejor** si M es un número **primo**:
 - Usaremos siempre arrays cuyo tamaño sea primo 😊

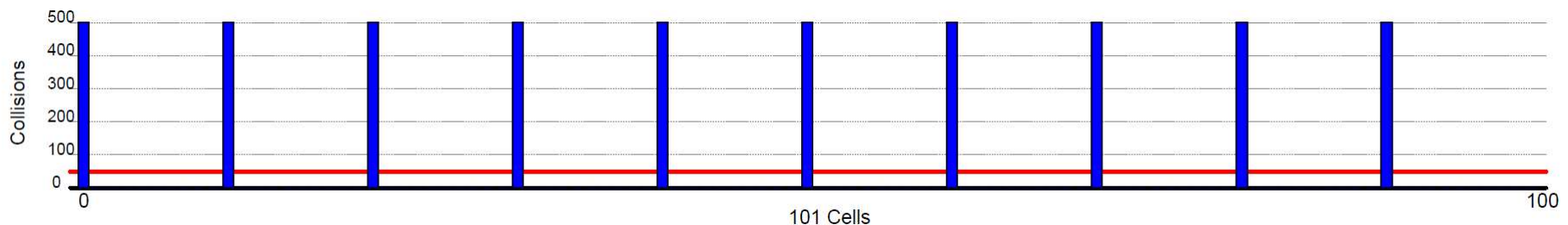
Hash Modular (II)

- Colisiones para 5000 números **aleatorios** usando hash modular en una tabla de 101 celdas (un número primo).
Los valores hash están distribuidos uniformemente 😊

La **línea roja** es la media del número de colisiones (5000/101)



- Colisiones para los siguientes 5000 enteros **no aleatorios**:
 $[i * 101 + q \mid i \leftarrow [1..500], q \leftarrow [0, 10..90]]$
Los valores hash no están distribuidos uniformemente 😞

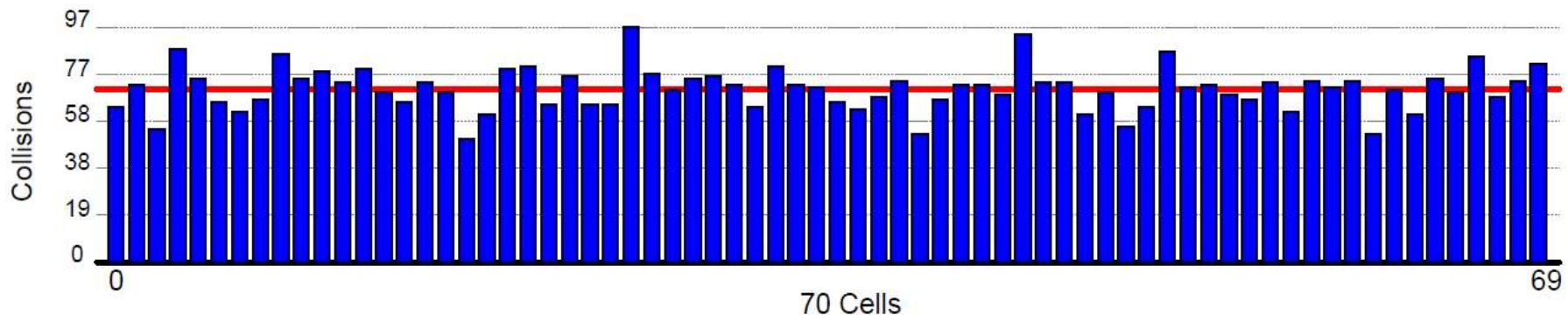


Hash Modular (III)

- Sea $M = 2 \times 5 \times 7 = 70$

no es un número primo

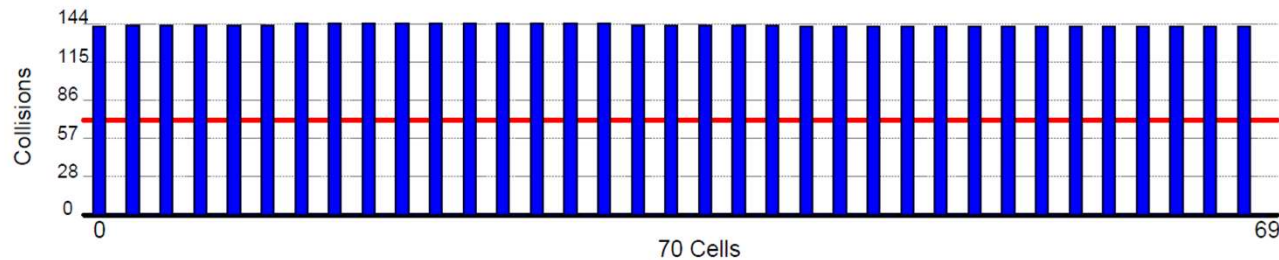
- Para una tabla de $M=70$ elementos, usando hash modular las colisiones para 5000 aleatorios se distribuyen uniformemente. Los valores hash también se distribuyen uniformemente. 😊



Hash Modular (IV)

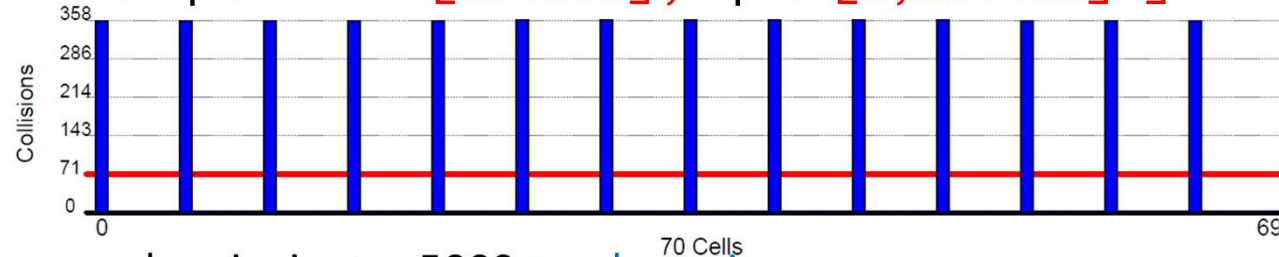
- Para una tabla de 70 celdas:
- Colisiones para los siguientes 5000 no aleatorios:

$$[i*2 + q \mid i \leftarrow [1..500], q \leftarrow [0,10..90]]$$



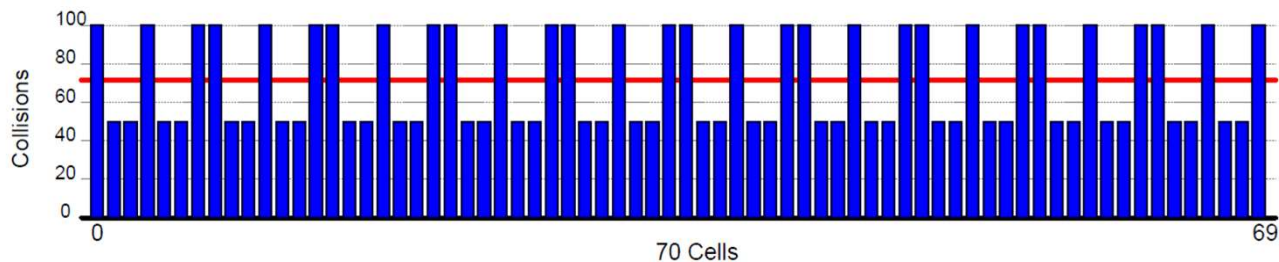
- Colisiones para los siguientes 5000 no aleatorios:

$$[i*5 + q \mid i \leftarrow [1..500], q \leftarrow [0,10..90]]$$



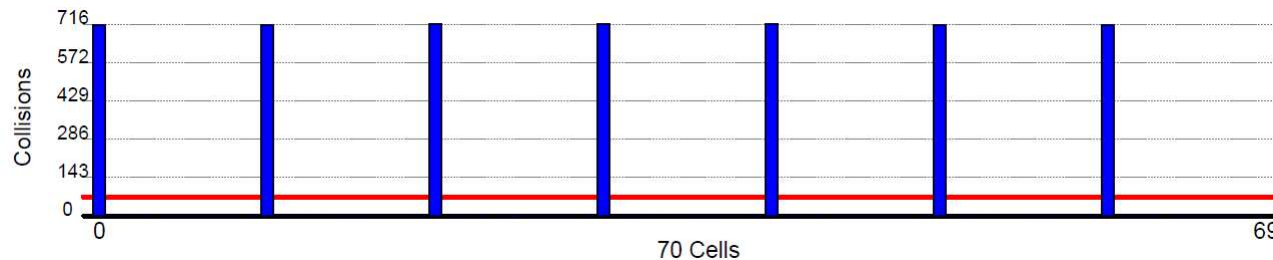
- Colisiones para los siguientes 5000 no aleatorios:

$$[i*7 + q \mid i \leftarrow [1..500], q \leftarrow [0,10..90]]$$



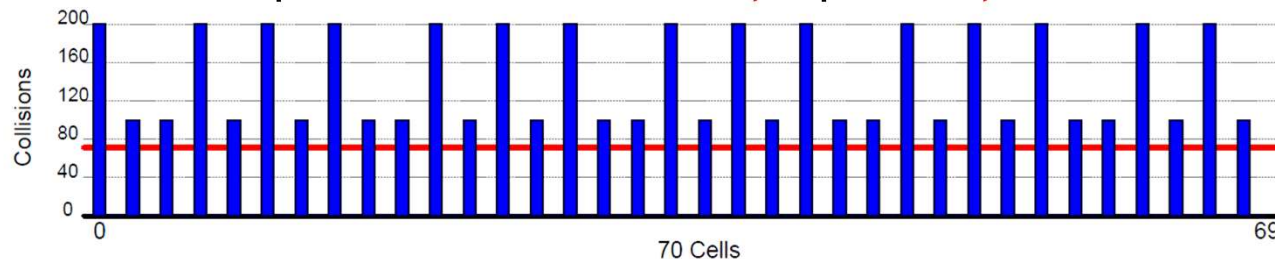
Hash Modular (V)

- Para una tabla de 70 celdas:
- Colisiones para los siguientes 5000 **no aleatorios**:
 $[i*10 + q \mid i \leftarrow [1..500], q \leftarrow [0,10..90]]$

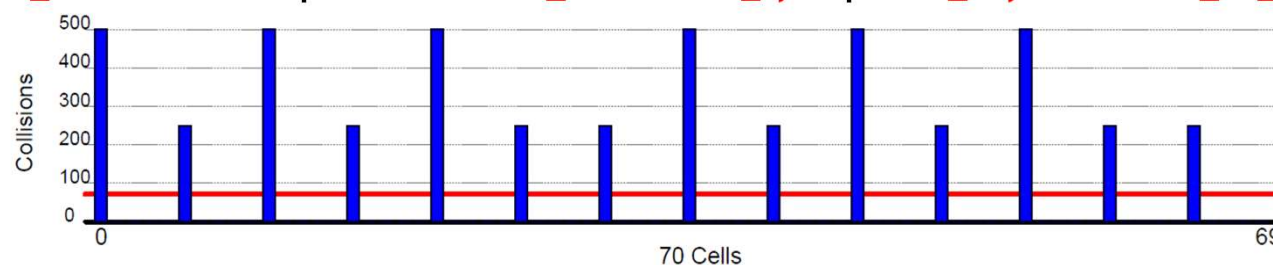


Si el tamaño de la tabla no es un número primo, hay muchas más posibilidades de que resulte una distribución no uniforme 😞

- Colisiones para los siguientes 5000 **no aleatorios**:
 $[i*14 + q \mid i \leftarrow [1..500], q \leftarrow [0,10..90]]$



- Colisiones para los siguientes 5000 **no aleatorios**:
 $[i*35 + q \mid i \leftarrow [1..500], q \leftarrow [0,10..90]]$



Métodos Hash basados en Congruencias Lineales (MAD Hashing)

- Para claves enteras son interesantes códigos hash generados por congruencias lineales MAD (Multiply, Add and Divide)

- $hash(key) = ((a * key + b) \% p) \% M$

- donde:

- p es un número primo,
 - $p > M$ (M = tamaño de la tabla)
 - $0 < a \leq p - 1$
 - $0 \leq b \leq p - 1$

Se generan igual que los pseudo-aleatorios
http://en.wikipedia.org/wiki/Linear_congruential_generator

- Se distribuyen pseudo-aleatoriamente 😊

Hashing para Números Flotantes

- Podemos tratar un flotante como un entero a través de su representación binaria
- En Java:

```
public static int floatToIntBits(float value)
```



En java.lang.Float

Hashing Polinomial (I)

- Son útiles si la clave es un string.
- Tomamos como código hash el valor de un polinomio donde los coeficientes $(x_0 x_1 \dots x_{n-1})$, son los códigos de los caracteres del string:

$$\begin{aligned} \text{hash}(x_0 x_1 \dots x_{n-1}) &= x_0 a^{n-1} + x_1 a^{n-2} + \dots + x_{n-2} a + x_{n-1} \\ &= x_{n-1} + a (x_{n-2} + a (x_{n-3} + \dots + a (x_2 + a (x_1 + a x_0) \dots))) \end{aligned}$$

Algoritmo de Horner

- Distribuimos estos valores en el rango $[0..M-1]$ vía módulo M :

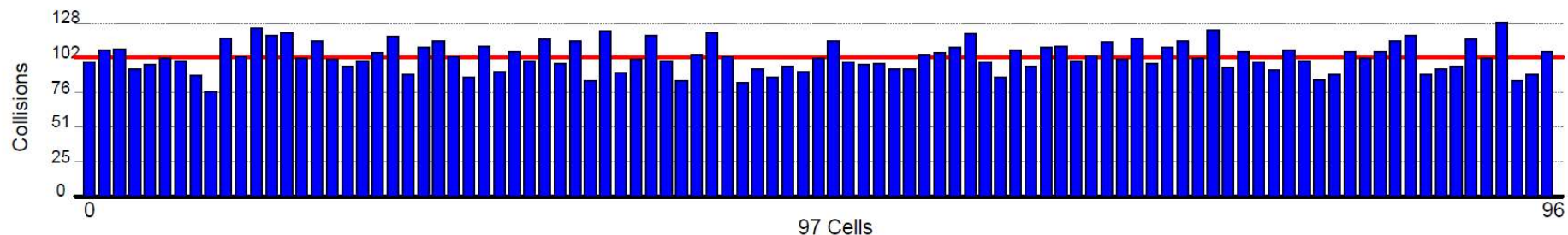
```
int hash = 0;
for (int i = 0; i < s.length(); i++)
    hash = (a * hash + s.charAt(i)) % M;
```

M es el tamaño de la tabla

- Para un string de n caracteres, el código hash realiza n sumas, n productos y n módulos
- Para una buena distribución de los códigos hash, a debe ser un número **primo**

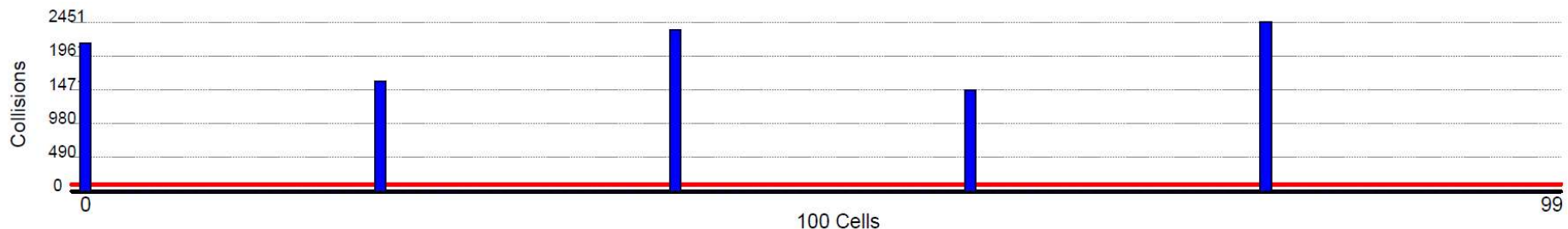
Hashing Polinomial (II)

- Distribución de las primeras 10.000 palabras de *El Quijote* sobre 97 celdas con hash polinómico ($a = 31$). 😊



- Distribución de las primeras 10.000 palabras de *El Quijote* sobre 100 celdas con hash polinómico ($a = 20$).

El tamaño de la tabla y el multiplicador a no son coprimos 😞



hashCode en Java

- La clase `Object` de Java define el método `hashCode`
- Devuelve un entero (int) de 32-bit (puede ser negativo)
- La implementación por defecto (heredada de `Object`) usa la dirección en memoria del objeto (funciona mal 😞)
- La mayoría de las clases de las librerías de Java (`Integer`, `Double`, `String`, ...) redefinen el método `hashCode` de forma más apropiada 😊
- Debemos redefinir `hashCode` en nuestras propias clases combinando los valores hash de los componentes del objeto
- `hashCode` debe conservar la igualdad:

$$e_1.\text{equals}(e_2) \Rightarrow e_1.\text{hashCode}() == e_2.\text{hashCode}()$$

Redefinición de hashCode. Ejemplo

- Podemos usar los valores hashCode de las componentes y combinarlos vía hashing polinomial:

```
public class Transaction {
```

```
    private final String who;  
    private final Date when;  
    private final double amount;
```

El valor hash de la clave es combinación de los de sus variables de instancia

```
    public int hashCode() {
```

Comenzamos con un valor no nulo

```
        int hash = 17;
```

```
        hash = 31 * hash + who.hashCode();
```

```
        hash = 31 * hash + when.hashCode();
```

```
        hash = 31 * hash + ((Double)amount).hashCode();
```

```
        return hash;
```

```
    }
```

Los tipos primitivos no son objetos: hay que hacer el *boxing*

```
    ...  
}
```

Calculando el Código hashCode de una Celda

- hashCode devuelve un entero de 32 bits (puede ser negativo 😞)

```
int hash(Key x) {  
    return (x.hashCode() & 0x7fffffff) % M;  
}
```

M es el tamaño de la tabla

Elimina el signo negativo
del valor hash

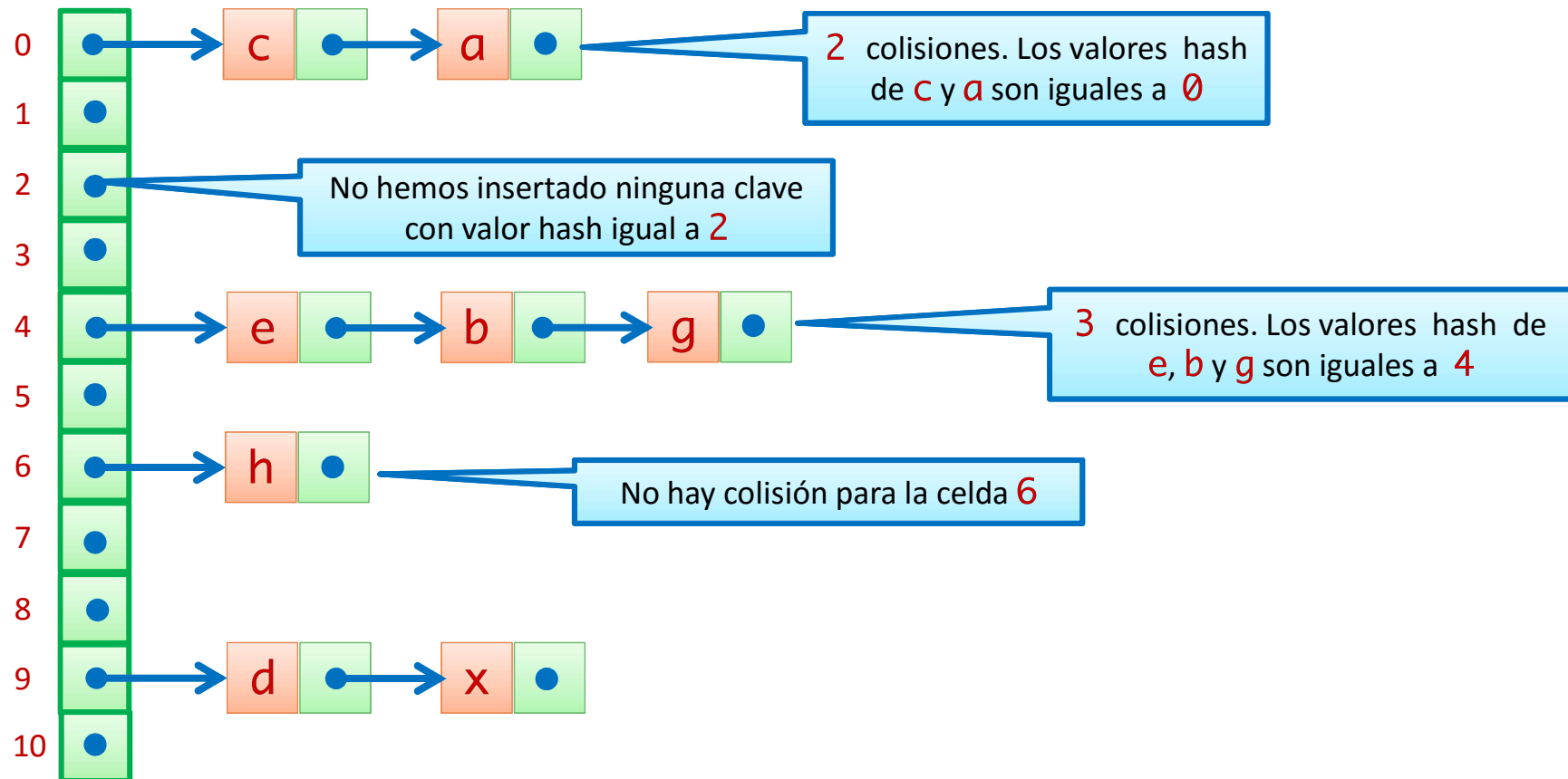
- Math.abs() no debe utilizarse ya que puede devolver un número negativo (para el mayor negativo admisible 😞)

Resolución de Colisiones

- En la práctica, coincidirán los valores hash de algunas claves: **colisiones**
- Existen distintas formas de **resolver colisiones**:
 - **Encadenado (Separate Chaining)**: una lista enlazada contiene los elementos que colisionan en la misma celda.
 - **Direccionamiento abierto (Open addressing)**: cada elemento que colisiona es reubicado en otra celda del array
 - **Prueba lineal**: asignar la siguiente celda libre
 - **Hashing doble**: con una función hash adicional, calcularemos la distancia a la siguiente celda (esta alternativa no la estudiaremos)

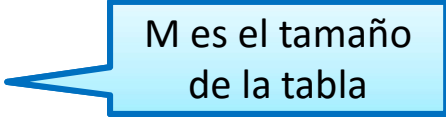
Encadenado Separado (I)

- Memorizamos en una lista enlazada los elementos que colisionan



Encadenado Separado (II).


Factor de Carga, Rendimiento y Rehashing

- Si los valores hash están distribuidos uniformemente, después de n inserciones:
la longitud de cada lista enlazada es $\cong n / M$ 
- n / M se conoce como **Factor de Carga** de la tabla hash
- Cuando el factor de carga aumenta, el rendimiento de la tabla hash disminuye 😞
- Para recuperarlo, debemos realizar una reubicación (**Rehashing**) periódica:
 - Ampliamos el array aproximadamente al doble de tamaño (intentamos usar como nuevo tamaño un número **primo**)
 - Los valores hash deben recalcularse y las claves deben reinsertarse en la tabla nueva.

Encadenado Separado (III)

```
public class SeparateChainingHashTable<K,V> implement HashTable<K,V> {  
    static private class Node<C,D> {  
        C key;  
        D value;  
        Node<C,D> next;  
        public Node(C k, D v, Node<C,D> n) {  
            key = k;  
            value = v;  
            next = n;  
        }  
    }  
    private Node<K,V> table[];  
    private int size; //number of currently inserted elements  
    private double maxLoadFactor; //maximum load factor allowed  
    public SeparateChainingHashTable(int numCells, double loadFactor) {  
        table = (Node<K,V>[]) new Node[numCells];  
        size = 0;  
        maxLoadFactor = loadFactor;  
    }  
}
```

Un nodo:



Representaremos asociaciones *key -> value* con técnicas hash

Número inicial de celdas en la tabla

Factor de carga máximo permitido. Si se sobrepasa, realizaremos un *rehashing*

Encadenado Separado (IV)

```
public boolean isEmpty() {  
    return size==0;  
}
```

```
public int size() {  
    return size;  
}
```

```
private int hash(K key) {  
    return (key.hashCode() & 0x7fffffff) % table.length;  
}
```

```
private double loadFactor() {  
    return (double)size / (double) table.length;  
}
```

```
// Precondition: idx is hash value for key  
// returns reference to linked node with key or null (if not found)
```

```
private Node<K,V> searchNode(K key, int idx) {  
    Node<K,V> current = table[idx];  
    while((current != null) && (!current.key.equals(key)))  
        current = current.next;  
  
    return current;  
}
```

Encadenado Separado (V)

```
public void insert(K key, V value) {
    if(loadFactor() > maxLoadFactor)
        rehashing(); // avoid degradation of performance

    int idx = hash(key);
    Node<K,V> node = searchNode(key, idx);
    if (node == null) { // key was not already in table
        table[idx] = new Node<>(key,value,table[idx]);
        size++;
    }
    else // key was already in table: update value
        node.value = value;
}

// returns value associated with key in table
public V search(K key) {
    int idx = hash(key);
    Node <K,V> node = searchNode(key, idx);
    return node==null ? null : node.value;
}
```

Encadenado Separado (VI)

```
// deletes node from
// corresponding linked list
public void delete(K key) {
    int idx = hash(key);
    Node<K,V> prev = null,
               current = table[idx];
```

```
while((current != null) && (!current.key.equals(key))) {
    prev = current;
    current = current.next;
}
```

```
if(current != null) { // node was found: delete it
    if(prev==null) // node was first in list
        table[idx] = current.next;
    else
        prev.next = current.next;
    size--;
}
```

Cada alumno debe revisar **pormenorizadamente** la implementación completa en el Campus Virtual, sobre todo la reasignación (**rehashing**) y el iterador (**iterator**) sobre las claves de la tabla

Encadenado Separado (VII) vs Árboles Binarios de Búsqueda

- Test Experimental
 - Hemos medido el tiempo de ejecución para 1 millón de operaciones aleatorias (inserción, búsqueda y eliminación) en una tabla inicialmente vacía
 - Usando una CPU Intel i7 860
 - La tabla hash (SeparateChainingHashTable) fue 3.6 veces más rápida que un árbol AVL 😊
 - y 3.3 veces más rápida que un árbol BST 😊
- Principales desventajas de la tabla hash vs AVL y BST:
 - Las búsquedas vía una relación de orden (menor, mayor, predecesor, etc.) no puede implementarse eficientemente.
 - No podemos garantizar una complejidad logarítmica de las operaciones y éstas pueden llegar a ser lineales en el peor caso (si muchas claves tienen el mismo valor hash).
 - Cuando se produce el rehashing el tiempo de esa operación es bastante elevado (no apta para tiempo real).

Direccionamiento Abierto. Prueba Lineal

■ Direccionamiento abierto:

- No usa listas enlazadas para resolver las colisiones.
- Resuelve cada colisión colocando el elemento en otra celda libre (el factor de carga debe ser ≤ 1).
- La búsqueda y la eliminación de claves se complican.

■ Prueba lineal:

- Trata el array como una estructura circular y se colocan los elementos que colisionan en la siguiente celda libre.
- El rendimiento se degrada en exceso cuando el factor de carga es > 0.5 . **Solución:** aumentar el tamaño de la tabla y realizar una reubicación (*rehashing*)

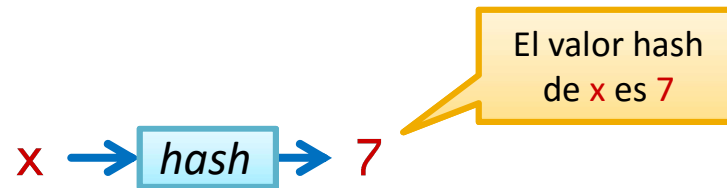
Prueba Lineal. Inserción (I)

- Inserción:
 - Sea `idx` el valor hash de la clave a insertar.
 - Si la celda `cell[idx]` está libre, insertarla aquí.
 - En otro caso, colocarla en la siguiente libre
 - *siguiente* debe interpretarse en forma circular



Prueba Lineal. Inserción (II)

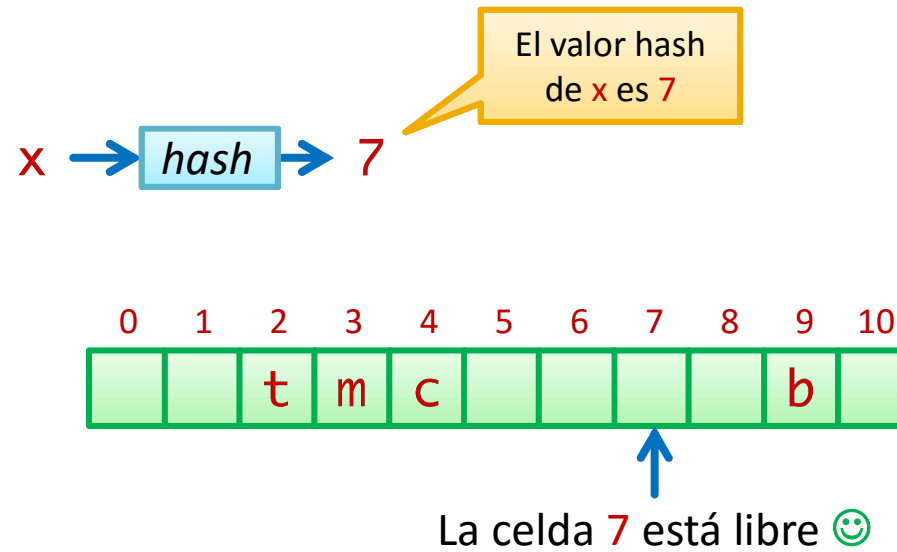
- insert **x**



0	1	2	3	4	5	6	7	8	9	10
		t	m	c					b	

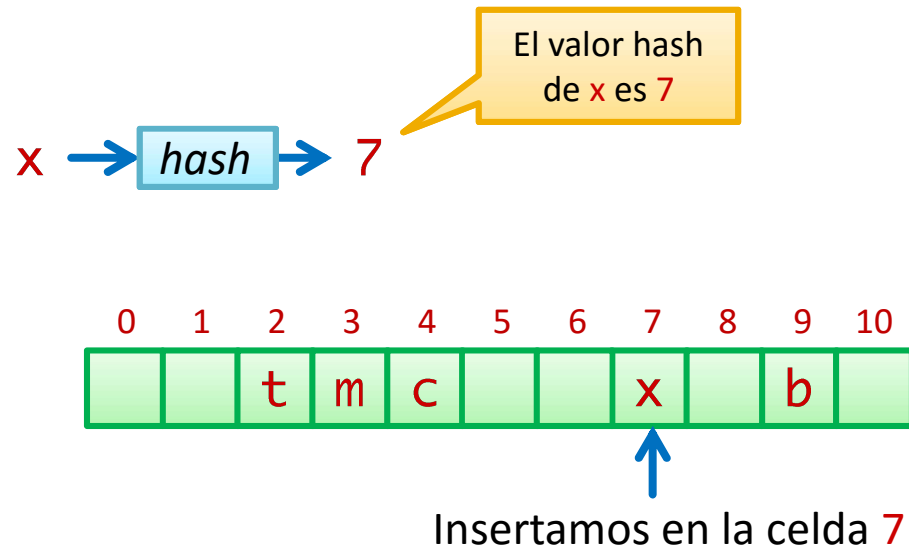
Prueba Lineal. Inserción (III)

■ insert x



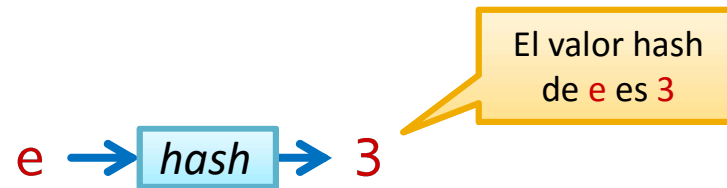
Prueba Lineal. Inserción (IV)

■ insert x



Prueba Lineal. Inserción (V)

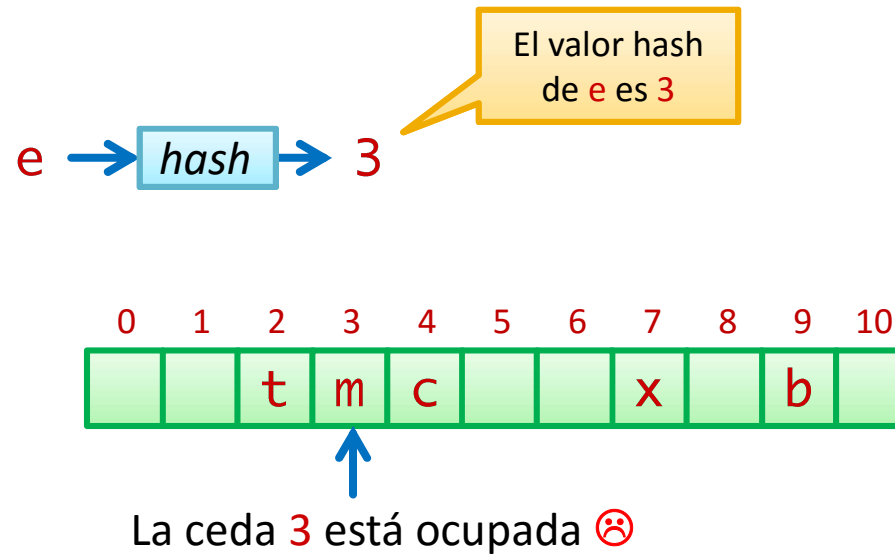
- insert **e**



0	1	2	3	4	5	6	7	8	9	10
		t	m	c			x		b	

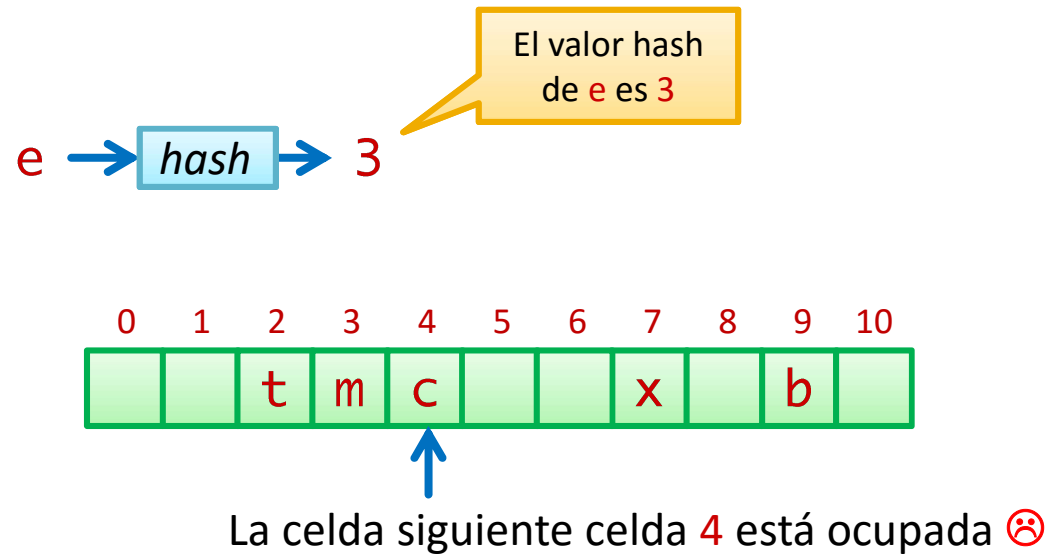
Prueba Lineal. Inserción (VI)

■ insert *e*



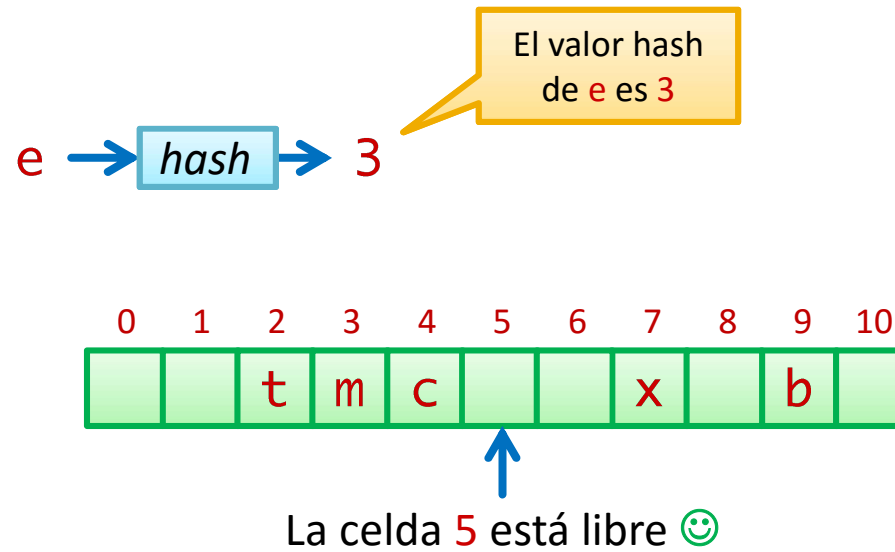
Prueba Lineal. Inserción (VII)

■ insert *e*



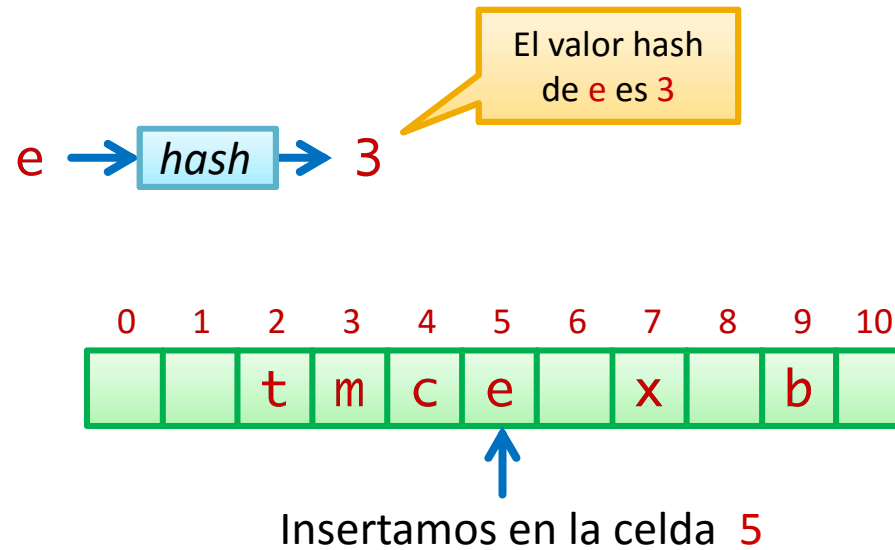
Prueba Lineal. Inserción (VIII)

■ insert e



Prueba Lineal. Inserción (IX)

■ insert e

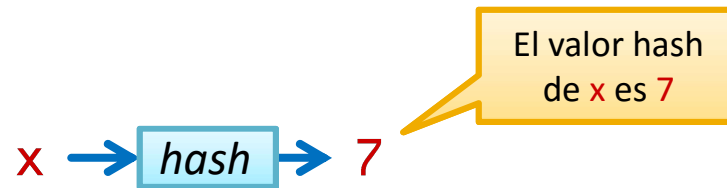


Prueba Lineal. Búsqueda (I)

- Búsqueda:
 - Comenzando en la celda correspondiente al valor hash, inspeccionamos secuencialmente hasta que:
 - o bien encontramos la clave (**éxito**),
 - o bien encontramos en su lugar una celda libre (**fracaso**)

Prueba Lineal. Búsqueda con éxito

■ search **x**



0	1	2	3	4	5	6	7	8	9	10
		t	m	c	e		x		b	

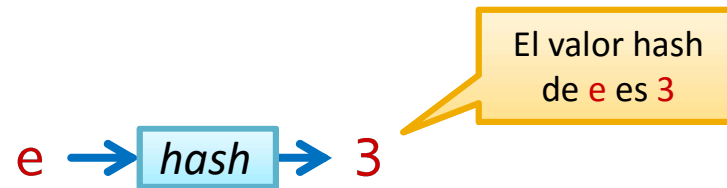
Prueba Lineal. Búsqueda con éxito (II)

■ search x



Prueba Lineal. Búsqueda con éxito (III)

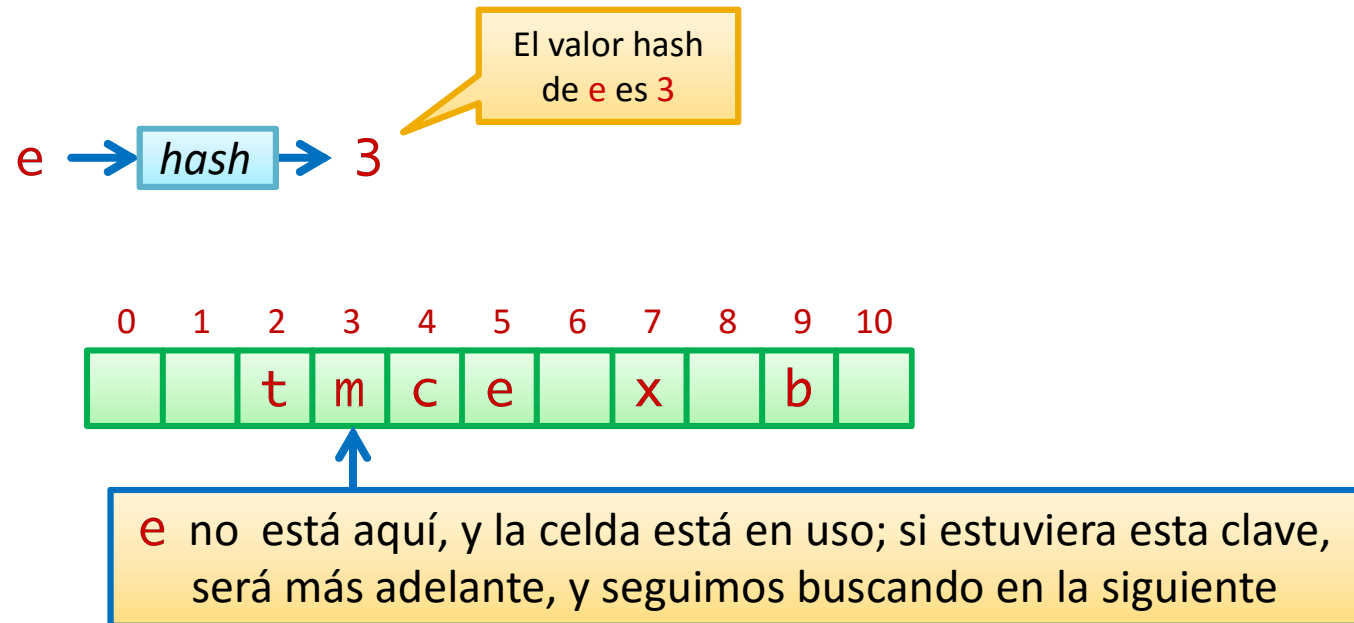
- search e



0	1	2	3	4	5	6	7	8	9	10
		t	m	c	e		x		b	

Prueba Lineal. Búsqueda con éxito (IV)

■ search e



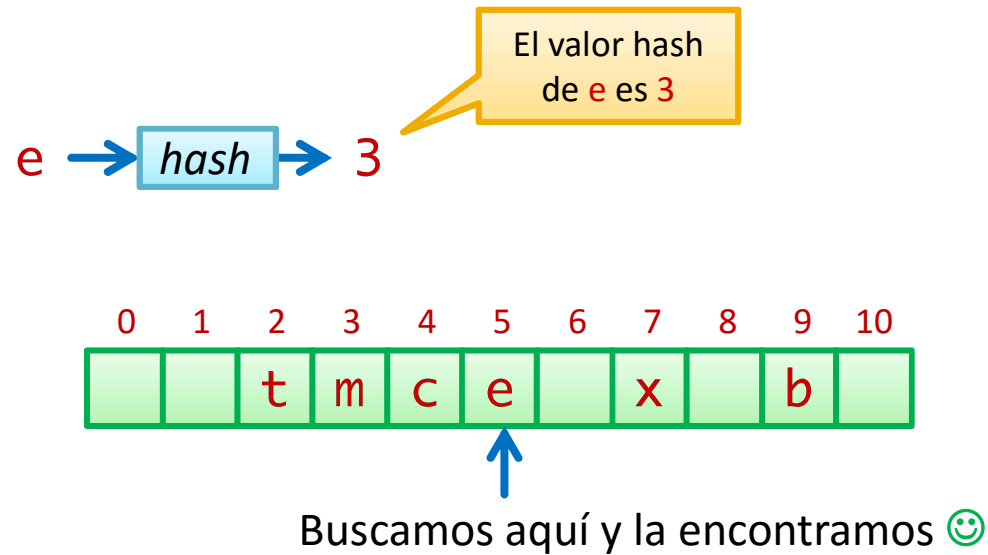
Prueba Lineal. Búsqueda con éxito (V)

■ search e



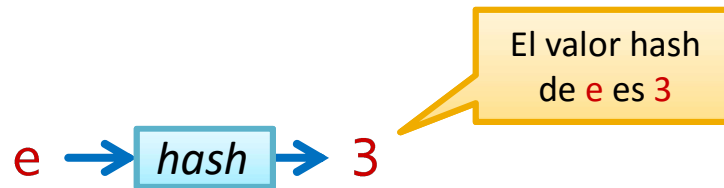
Prueba Lineal. Búsqueda con éxito (y VI)

■ search e



Prueba Lineal. Búsqueda sin éxito (I)

- search **e**



0	1	2	3	4	5	6	7	8	9	10
		t	m	c			x		b	

Prueba Lineal. Búsqueda sin éxito (II)

- search **e**



e no está aquí, y
probamos en la siguiente

Prueba Lineal. Búsqueda sin éxito (III)

- search **e**

e → hash → 3

0	1	2	3	4	5	6	7	8	9	10
		t	m	c			x		b	

e no está aquí, y
probamos en la siguiente

Prueba Lineal. Búsqueda sin éxito (IV)

- search **e**

e → hash → 3

0	1	2	3	4	5	6	7	8	9	10
		t	m	c			x		b	

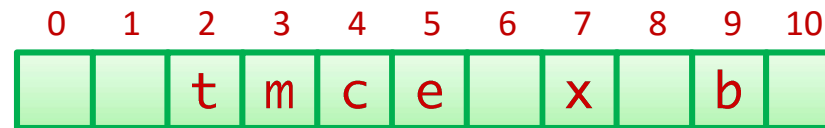
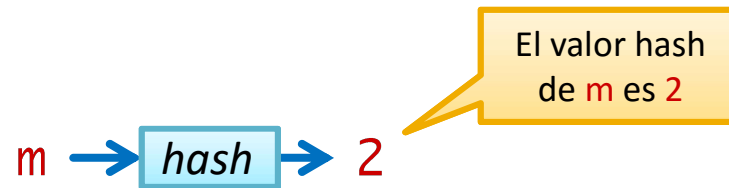
Esta está libre, luego **e** no
está en la tabla

Prueba Lineal. Eliminación (I)

- Eliminación:
 - Buscamos el elemento a eliminar.
 - Si está, lo eliminamos liberando la celda.
 - Reubicamos los elementos de la celdas siguientes a la clave eliminada hasta encontrar un hueco (entre claves que colisionan no puede haber huecos).

Prueba Lineal. Eliminación (II)

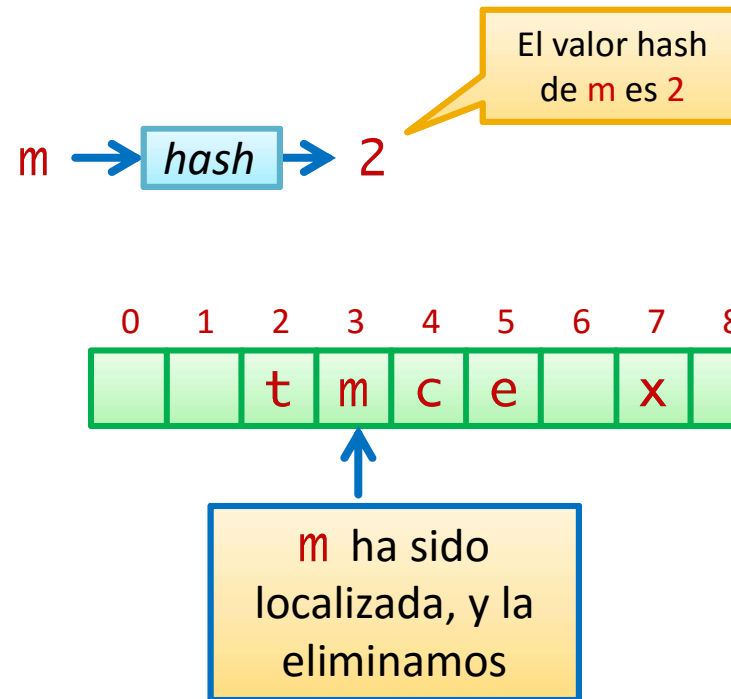
- delete **m**



m no está aquí, pero
al estar la celda
ocupada buscamos en
la siguiente

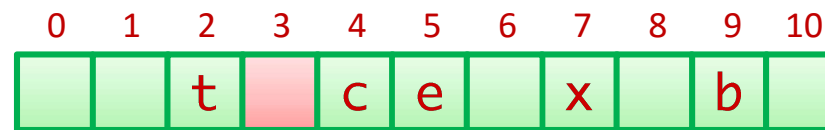
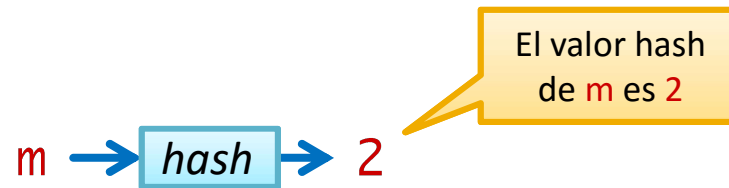
Prueba Lineal. Eliminación (III)

- delete **m**



Prueba Lineal. Eliminación (IV)

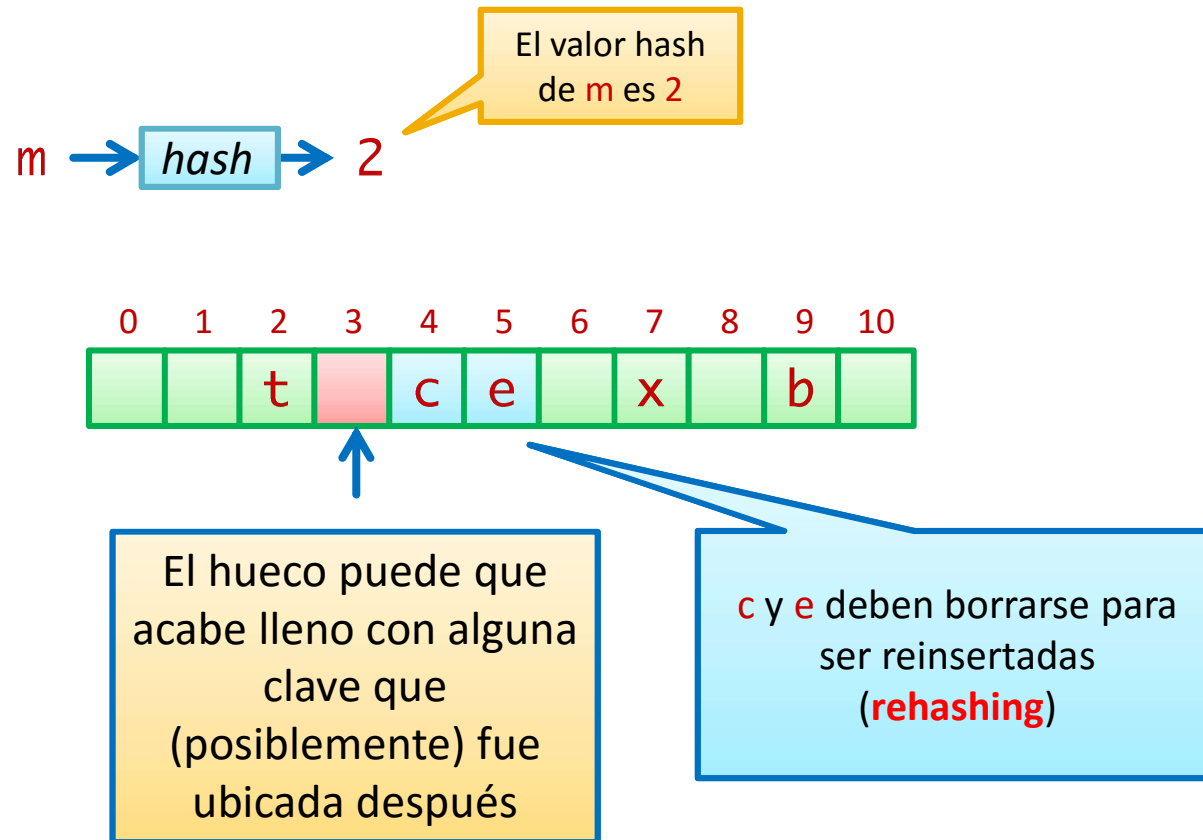
- delete m



No debemos dejar un hueco! ya que la siguiente búsqueda de un elemento con valor hash 3 será errónea

Prueba Lineal. Eliminación (V)

- delete m



Prueba Lineal. Eliminación (y VI)

Situación tras insertar sucesivamente las claves a_3 , b_3 , c_4 , d_7 (el subíndice indica el valor hash)

0	1	2	3	4	5	6	7	8	9	10
			a_3	b_3	c_4		d_7			

Situación tras eliminar la clave a_3

0	1	2	3	4	5	6	7	8	9	10
			b_3	c_4			d_7			