# TDA TABLA HASH

# BUSQUEDA POR HASHING

## INTRODUCCION

* No opera por comparaciones entre valores clave, sino encontrando alguna función $h(K)$ que nos de la localización exacta directamente de la clave $K$ en la estructura de datos donde estén almacenadas las claves

. Son faciles de encontrar esas funciones $h$?

NO Si buscamos que para $\forall i \neq j \Rightarrow h(i) \neq h(j)$

• Tabla tamaño 40 y 30 claves

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

$\frac{40!}{10!} \approx 2.25 \times 10^{41}$ no generan duplicados

→ Solo 2 de cada 10 millones nos valdrían

. Paradoja del cumpleaños

Objetivo por tanto:

* Encontrar funciones $h$ (funciones hash) que generen el menor número posible de colisiones
* Diseñar métodos de resolución de colisiones cuando éstas se produzcan

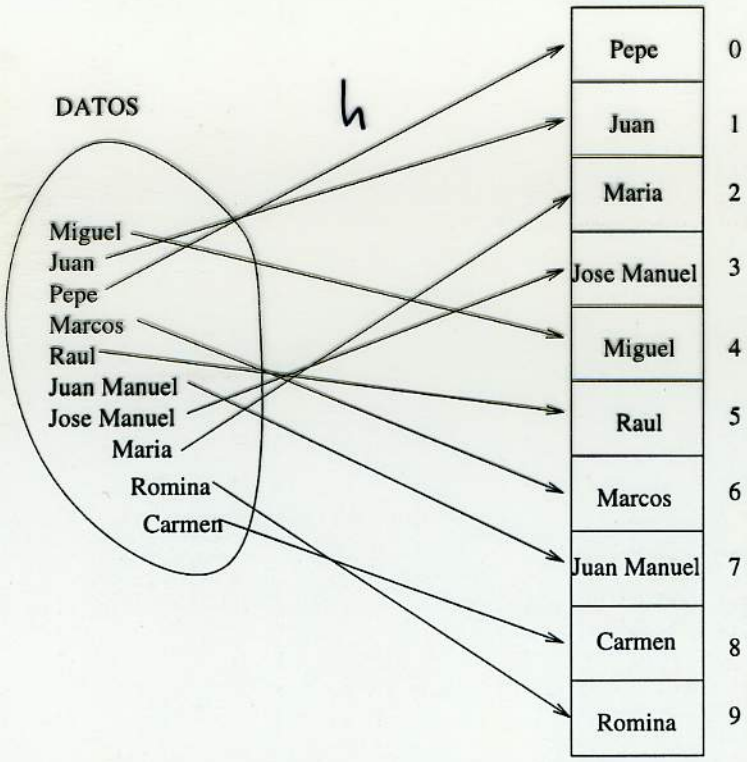
# Introducción a las tablas Hash (I)

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 una búsqueda de un elemento en un contenedor sería aplicar una función matemática sobre el dato y que me devolviera directamente el lugar en el que se encuentra. Esto sería $O(1)$.

# Introducción a las tablas Hash (II)

A esa función se le llama función hash.



# Introducción a las tablas Hash

- Esa función tendría que ser inyectiva. El problema es que encontrar una función de ese tipo no es nada sencillo.

- Cuando existe una función hash biyectiva decimos que tenemos un *función hash perfecta*. El conjunto de datos debe ser fijo y predeterminado.

- Para el resto de los casos tendremos funciones sobreyectivas, es decir, para dos claves distintas podremos obtener un mismo valor.

  En este caso se producen *colisiones* en el valor de la función hash.

- Dependiendo de la forma en la que se resuelvan esas colisiones tendremos *encadenamiento separado* o *direccionamiento abierto*.

# ESQUEMA

- FUNCIONES HASH
  - TRUNCAMIENTO
  - PLEGADO
  - MULTIPLICATIVO
  - **DIVISION**
- RESOLUCION DE COLISIONES
  - HASHING ABIERTO: TABLA HASH ABIERTA
  - HASHING CERRADO: TABLA HASH CERRADA
    - Rehashing lineal
    - **Rehashing doble**
    - otros

# Funciones hash

Se deben definir de manera que:

- Tome como valor todos y cada uno de los posibles valores.
- distribuyan de forma lo más aleatoria posible las claves, y
- minimice el número de colisiones.

Partiendo de que el lugar disponible para colocar las claves es un vector dinámico de tamaño $M$, existen distintos tipos de métodos para determinar una función hash.

## Funciones hash

$$h : C \longrightarrow Z$$

- el dominio, $C$, corresponde al conjunto de las posibles claves
- el rango, $Z$, es el conjunto de enteros positivos (que puede contener el cero) y corresponde al conjunto de los índices sobre la tabla Hash.

En la práctica se deben utilizar funciones:

1. Que sean rápidas de calcular.
2. Que produzcan una dispersión aleatoria pero uniforme en la tabla.

1. **Truncamiento**

   Consiste en eliminar algunos dígitos de la clave.

   - $h(123456789) = h(123\underline{456789}) \equiv 123$
   - $h(121567890) = h(121\underline{567890}) \equiv 121$

   Inconveniente: las tablas hash deben ser de un tamaño potencia de 10.

   Alternativa: truncamiento a nivel interno (a nivel de bits). La tabla debe ser de un tamaño potencia de 2.

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 la función hash para que rote alguno de los sumandos:

   $$h(\overline{123}\underline{954}) = 123 + 654 = 777$$

   Puede combinarse con la de truncamiento:

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

   Puede involucrar a más de dos sumandos.

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

   Inconveniente: las tablas hash deben ser de un tamaño potencia de 10.

# MULTIPLICACION

Parecido al plegado, solo que en lugar de sumas involucran productos. Puede haber truncamiento antes o después del producto.

Ejemplo: Tabla de tamaño 10.000 y claves de 9 dígitos

$h(123456789) = 123 * 789 = 97047 \equiv 7047$

- Requiere tablas de tamaño potencia de 10
- Tiende a esparcir claves → menos colisiones

Variantes

1) Cuadrado del centro

$h(123456789) = 7936$

$456^2 = 207936 \rightarrow 7936$

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

2) Centro del cuadrado

$h(1234) = 1234^2 = 1522756 = 1522756 = 2275$

(calcular el cuadrado de la clave y seleccionar un cierto número de cifras del centro)

3) $h(K) = Int(M \times Frac(C \times K))$ → M tamaño tabla

→ $0 \leq C \leq 1$ $\left(C = \frac{2}{1+\sqrt{5}}\right)$

# DIVISION - RESTO

Consiste en tomar el resto de la división de la clave entre el tamaño de la tabla (M)

$h(K) = K \bmod M$ $\quad (h(K) = K \,\%\, M)$

- Método muy simple que no requiere truncamiento
- Las tablas pueden ser de tamaño arbitrario

Ejemplo

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 |

$(h(52) = 52 \,\%\, 11 = 8)$

$(h(44) = 44 \,\%\, 11 = 0)$

Consideraciones

- El tamaño de la tabla ha de ser por lo menos igual al número de claves
- La mejor elección es no tomar M como par o impar simplemente sino como primo: M un número primo mayor que el número de claves.

# Diseño de una función Hash

(Resumen)

- Método de multiplicación: Consiste en multiplicar por un valor y luego quedarnos con algunos de los bits de la expresión binaria del dato. Tiene el inconveniente de que $M$ sólo puede ser una potencia de 2.

- Método de división: Se calcula el resto módulo $M$ a la clave, con $M$ primo

- Método de multiplicación y división: $h(x) \equiv (ax + b)\,\%M$ con $a\,\%M \neq 0$

Dependiendo del tipo de dato que sea la clave tenemos distintas posibilidades:

- Si es un entero podemos utilizar como función $h(x) = x\,\%M$

- Si es un string $h(x) = (x[0] + ... + x[n-1])\,\%M$ con $n$ la longitud de x

## Tratamiento de colisiones

- **Motivación**: En la práctica totalidad de los casos, las funciones hash producen colisiones.

- **Objetivo**: Encontrar un mecanismo para la ubicación de la clave que provoca la colisión de forma que después, en una operación de consulta, la búsqueda se realice de forma eficiente.

- **Cómo**. Depende de la estructura de datos elegida.

  - Estructuras de datos dinámicas (*listas*).
    **Hashing abierto**.

  - Estructuras de datos estáticas (*vectores*).
    **Hashing cerrado**.

En última instancia, depende de si se conoce de antemano el número -o una estimación del mismo- de elementos a ubicar en la tabla hash.

- **No** se conoce: hashing abierto,

- **Si** se conoce: hashing cerrado.

## 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* que se aloja en el Heap.

- La tabla hash puede implementarse:

  1. Como un vector estático de punteros a estas listas.
  2. Como una lista dinámica de punteros.

- En cualquier caso, se fija el tamaño a priori.

- **Búsqueda**. Se reduce a recorrer la lista oportuna.

- Criterio de inserción: LIFO, FIFO u orden.

- **Ventaja**. La tabla puede tener un tamaño inferior al número de registros ya que "crece" mediante asignación dinámica de memoria.

- **Desventaja**. El espacio adicional ocupado por los punteros requeridos para mantener las estructuras de tipo lista.

# Encadenamiento separado

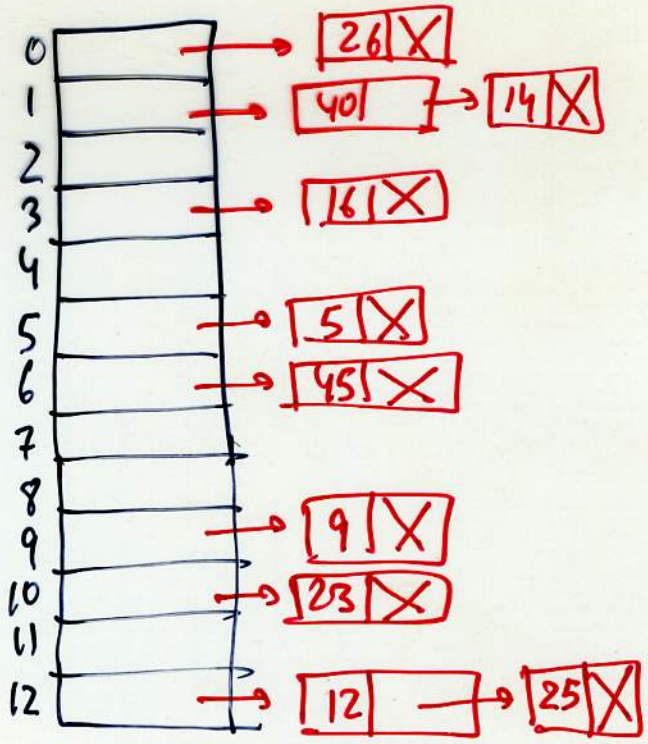
(Hashing Abierto)

Las coliciones se resuelven insertándolas en una lista. De esa forma tendríamos como estructura un vector de listas. Al número medio de claves por lista se le llama *factor de carga* y habrá que intentar que esté próximo a 1.

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

| | |
|---|---|
| 0 | <26> |
| 1 | <40 14> |
| 2 | |
| 3 | <16> |
| 4 | |
| 5 | <5> |
| 6 | <45> |
| 7 | |
| 8 | |
| 9 | <9> |
| 10 | <23> |
| 11 | |
| 12 | <12 25> |

ó



# Direccionamiento abierto

(Hashing cerrado)

Utilizamos un vector como representación y cuando se produzca una colisión la resolvemos reasignándole otro valor hash a la clave hasta que encontremos un hueco.

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

| | | |
|---|---|---|
| 0 | 26 | O |
| 1 | 40 | O |
| 2 | 14 | O |
| 3 | 16 | O |
| 4 | 25 | O |
| 5 | 5 | O |
| 6 | 45 | O |
| 7 | | L |
| 8 | | L |
| 9 | 9 | O |
| 10 | 23 | O |
| 11 | | L |
| 12 | 12 | O |

O: casilla ocupada
L: casilla libre
B: casilla borrada

# Reasignación de casillas

Para esta reasignación tenemos varias posibilidades:

- prueba lineal: $h_i(x) = (h(x) + i)\,\%M$ donde $i$ es el número de intento. Se producen agrupamientos primarios.

- prueba cuadrática: $h_i(x) \equiv h(x) + i^2$ donde $i$ es el número de intento. Se producen agrupamientos dobles.

- hashing doble: Usamos una segunda función hash $h'(x)$ con lo que tendriamos $h_i(x) = (h(x) + i * h'(x))\,\%M$. Esta función $h'(x)$ nunca podrá tomar el valor 0.
  Un ejemplo podría ser $h'(x) = q - (x\,\%q)$ con $q$ siendo un número primo menor que $M$.

# Direccionamiento abierto

(Hashing cerrado)

- Las búsquedas se hacen siguiente la misma secuencia de la función hash que se utilizó para la inserción.

- Los borrados dejan las casillas en un estado distinto al de libre u ocupado. Este estado permite que sean cosideradas como casillas libre para una inserción y como ocupadas para las búsquedas.

- Una vez que la tabla se llena se debe seleccionar un tamaño de tabla mayor que también sea primo y volver a insertar todas las claves (como cambia $M$ también cambia la función hash). A este proceso se le llama *rehashing*

## Hashing cerrado
## Borrados y redimensionamiento

- Alteración de la tabla hash: **inserción (adición)** de registros y **borrado.**

- **Algoritmo de búqueda sobre una tabla hash.**

  1. Calcular $h(c)$
  2. Si `clave(`$h(c)$`) = c` ,
     `posicion <-- registro(`$h(c)$`)`
     si no,
     Repetir
     $h_i(c)$ `<-- rehashing(`$h_{i-1}(c)$`)`
     Hasta que `((clave(`$h_i(c)$`) = c) O ( vacia (clave(`$h_i(c)$`))))`
     Si `(clave(`$h_i(c)$`) = c)`
     `posicion <-- registro(`$h_i(c)$`)`
     si no,
     `posicion <-- -1`
  3. Devolver (`posicion`)

- **Problema**: ¿Qué ocurre si se borra un registro en el fichero de datos? Puede ser parte de una cadena de búsqueda.

  **Solución**: Buscar la casilla que lo referencia en la tabla hash. Depués, debe marcarse como **borrada.**

- Diferencia entre una casilla *borrada* y una casilla *vacía*:
  - **Inserción.** *borrada* y *vacía* son equivalentes (hay un "hueco").
  - **Búsqueda.** *borrada* y *ocupada* son equivalentes en la búsqueda.

- **Redimensionamiento de la tabla hash.**

  Consiste en volver a construir la tabla hash y volver a hacer hashing (y rehashing, si aparecen colisiones) con todas las claves activas en la tabla antigua.

  **Cuándo**: cuando una tabla hash se desborda (se llena) o cuando su eficiencia baja demasiado debido a los borrados.