

ESTRUCTURAS DE DATOS

DICCIONARIOS

Introducción a las tablas *hash*

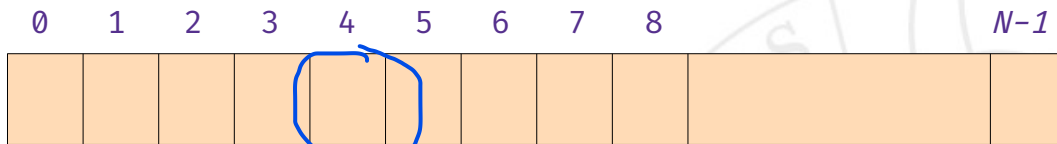
Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

¿Qué es una tabla *hash*?

Esta va a ser una nueva forma de implementar los diccionarios. Mediante las tablas hash

- Es una estructura de datos que permite implementar colecciones de datos no secuenciales: diccionarios, conjuntos, etc. para almacenar almacena datos no secuenciales

- Utiliza un vector de tamaño N (número primo). No existe la noción de posición. Preferentemente



A cada posición del vector se le llama cajón o en inglés bucket

- Se basa en una función hash h que devuelve, para una clave, un número entero. Función que transforma una clave en un número entero.

$$h: K \rightarrow \mathbb{Z}$$

Independientemente del tipo de clave que tengamos, vamos a tener que transformar esa clave en un entero.

- Este número determina la posición del vector en la que se almacena la clave.

A veces, esa función hash va a devolver un número mayor que el número de buckets que tiene el vector, no pasa nada, veremos formas de arreglar eso.

Ejemplos de funciones hash

¿Cómo transformamos las claves en enteros?-> funciones hash

- Para números enteros o naturales, la identidad es suficiente:

$$h(x) = x$$

- Para cadenas, suele utilizarse la siguiente fórmula:

$$h(s) = s[0] \cdot p^0 + s[1] \cdot p^1 + s[2] \cdot p^2 + \dots + s[n-1] \cdot p^{n-1}$$

donde:

- s es una cadena de longitud n.
- s[i] es el código asociado al carácter i-ésimo.
- p es un número primo (normalmente $p = 31$ o $p = 53$ o $p = 131$)

Minúsculas

Minúsculas y mayúsculas

Un número mayor de caracteres.

los que se suelen utilizar

¿Cómo se implementa esta función *hash*?

- Necesitamos una función `hash` que se comporte de forma diferente en función de si recibe un entero, una cadena, etc.
- También queremos poder extenderla para tratar nuevos tipos de claves.
- **Solución:** objetos función.

Hasta ahora hemos visto `hash` para enteros y strings, pero... ¿qué pasa si queremos hacer lo mismo para la clase `Fecha`?

RECORDATORIO :Instancias de clases. Sobrecargan el operador paréntesis ().

```
template<class K>
class std::hash {
public:
    int operator()(const K &key) const;
};
```

Esto está en C++. Devuelve el código asociado a esa clave.

¿Cómo se implementa esta función hash?

- Las plantillas de C++ pueden particularizarse para tipos de datos concretos.
- Por ejemplo, implementación para el caso $K = \text{int}$.

INT

```
template<
class std::hash<int> {
public:
    int operator()(const int &key) const {
        return key;
    }
};
```

Clase Hash particularizada para los números enteros.

Al ser enteros devolvemos la misma clave que recibimos.

¿Cómo se implementa esta función *hash*?

- Las plantillas de C++ pueden particularizarse para tipos de datos concretos.
- Por ejemplo, implementación para el caso $K = \text{string}$.

STRING

```
template<>
class std::hash<std::string> {
public:
    int operator()(const std::string &key) const {
        const int POWER = 37;
        int result = 0;
        for (int i = key.length() - 1; i ≥ 0; i--) {
            result = result * POWER + key[i];
        }
        return result;
    }
};
```

En teoría aplica la función para los strings que hay arriba.

Ejemplo

```
hash<int> h_int;  
hash<std::string> h_str;
```

```
std::cout << h_int(24) << std::endl;  
std::cout << h_str("Pepe") << std::endl;  
std::cout << h_str("Maria") << std::endl;
```

OBJETOS FUNCIÓN PARA EL TIPO DE DATOS DE ENTEROS Y PARA LOS STRINGS. SOBRECARGAN EL OPERADOR PARÉNTESIS.

24
5273098
187271914

CÓDIGOS HASH

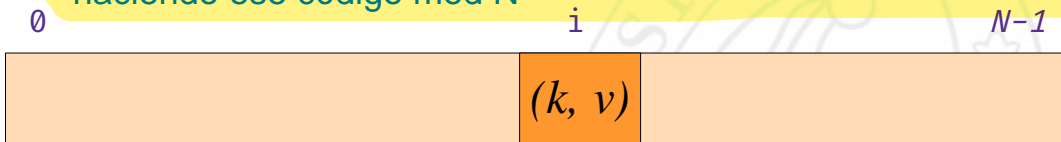
¿Cómo funcionan las tablas *hash*?

Supongamos que queremos **insertar** una entrada (k, v) en un diccionario implementado mediante una tabla *hash* con N posiciones.

- 1) Calculamos $i := h(k) \bmod N$.
- 2) Insertamos el par (k, v) en la posición i -ésima del vector.

Código hash para una clave k

Insertamos el par en la posición i que se calcula calculando el código hash para la clave k y haciendo ese código mod N

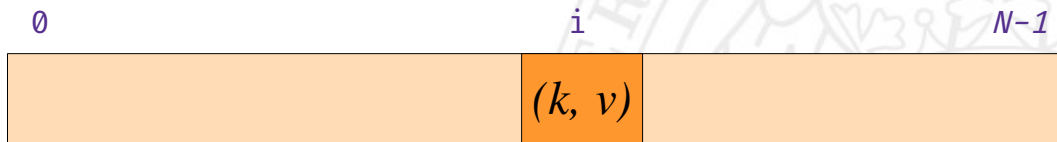


INSERCIÓN

¿Cómo funcionan las tablas *hash*?

Supongamos que queremos **buscar** la entrada con clave k en un diccionario implementado mediante una tabla *hash* con N posiciones.

- 1) Calculamos $i := h(k) \bmod N$. Código hash para esa clave, hago el módulo N y eso nos devuelve la posición i del vector.
- 2) Obtenemos el par (k, v) de la posición i -ésima del vector.
- 3) Devolvemos v . Devolvemos el valor asociado a la clave.



BÚSQUEDA

Ejemplo

- Con $N = 13$, queremos insertar la entrada $(35, v_1)$.
- Hacemos $h(35) \bmod 13 = 35 \bmod 13 = 9$. Luego insertamos la entrada en la posición 9

0	1	2	3	4	5	6	7	8	9	10	11	12
									35 v_1			

EL MÓDULO LO QUE CALCULA AQUÍ ES EL RESTO DE HACER LA DIVISIÓN $35 / 13$

Ejemplo

- Ahora insertamos la entrada $(136, v_2)$
- Hacemos $h(136) \bmod 13 = 136 \bmod 13 = 6$.

$$\begin{array}{r} 136 \overline{) 13} \\ \underline{10} \\ 66 \end{array} \rightarrow \bmod$$

0	1	2	3	4	5	6	7	8	9	10	11	12
						136 v_2			35 v_1			

Ejemplo

N es 13 porque es la capacidad del vector.

- Buscamos la entrada con clave 35.
- $h(35) \bmod 13 = 35 \bmod 13 = 9$.

0	1	2	3	4	5	6	7	8	9	10	11	12
						136 v_2			35 v_1			

Clave
encontrada

Ejemplo

- Buscamos la entrada con clave 41.
- $h(41) \bmod 13 = 41 \bmod 13 = 2$.

0	1	2	3	4	5	6	7	8	9	10	11	12
						136 v_2			35 v_1			



Clave **no**
encontrada

Ejemplo

- Buscamos la entrada con clave 149.
- $h(149) \bmod 13 = 149 \bmod 13 = 6$.

0	1	2	3	4	5	6	7	8	9	10	11	12
						135 v_2			35 v_1			

Clave **no**
encontrada

SON CLAVES DISTINTAS.

Ejemplo

- Insertamos la entrada $(61, v_3)$.
- $h(61) \bmod 13 = 61 \bmod 13 = 9$.

No podemos quitar la 35 para escribir el 61 porque en algún momento a lo mejor queremos leer la clave 35

0	1	2	3	4	5	6	7	8	9	10	11	12
						135 v_2			35 v_1			

ACABA DE OCURRIR UNA COLISIÓN

¡Posición
ocupada!

AQUÍ QUEREMOS HACER UNA INSERCIÓN

Colisiones

Cuando lo de arriba ocurre decimos que hay una colisión

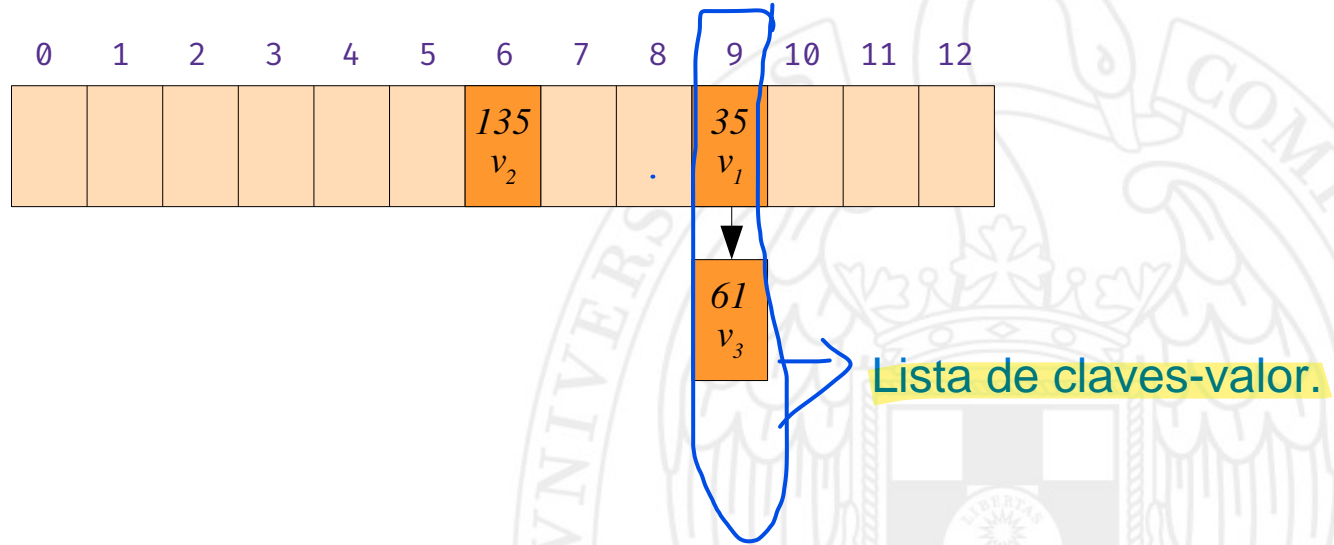
- Cuando la función *hash* envía dos claves k_1 y k_2 a la misma posición del vector se produce una **colisión**.
- Esto sucede cuando $h(k_1) \bmod N = h(k_2) \bmod N$.
- Una buena función *hash* debe distribuir de la manera más uniformemente posible las claves entre las distintas posiciones del vector, para que la probabilidad de colisiones sea baja.
- Pero, tarde o temprano, tendremos colisiones.

Deberíamos a toda costa evitar tener estas colisiones. esto se consigue construyendo una BUENA FUNCIÓN HASH que nos evite en la mayoría de ocasiones estas colisiones.

1-0

¿Cómo solucionamos las colisiones?

- **Tablas *hash* abiertas:** Cada posición del vector contiene una **lista** de todas las **claves** destinadas ahí.

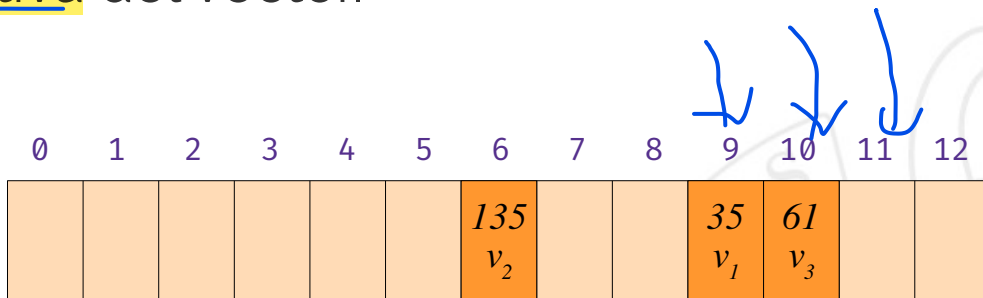


Buscaríamos en todos los elementos asignados a un cajón.

¿Cómo solucionamos las colisiones?

2^o

- **Tablas *hash* cerradas:** reubican el par que queremos insertar en una posición alternativa del vector.



0	1	2	3	4	5	6	7	8	9	10	11	12
						135 v_2			35 v_1	61 v_3		

Si queremos buscar la clave 61, la función hash te enviará a la posición 9, entonces vas a tener que ir revisando las posiciones siguientes hasta encontrar una posición vacía o bien la clave.

Implementaciones

una lista asociada a cada bucket/cajón

- TAD Diccionario utilizando tablas *hash* abiertas.
 - Tablas de tamaño fijo.
 - Tablas redimensionables dinámicamente.
- TAD Diccionario utilizando tablas *hash* cerradas.

buscamos posiciones alternativas en presencia de una colisión.