ESTRUCTURAS DE DATOS

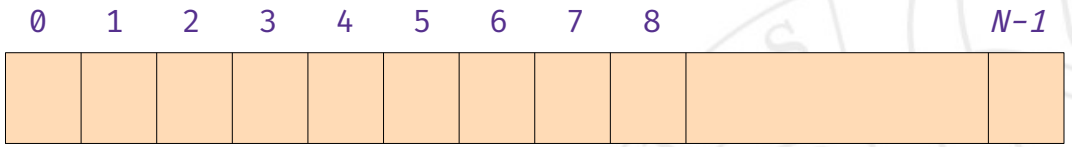
DICCIONARIOS

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*?

- Es una estructura de datos que permite implementar colecciones de datos no secuenciales: diccionarios, conjuntos, etc.
- Utiliza un vector de tamaño N (número primo).



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

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

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

Ejemplos de funciones *hash*

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

$$h(x) = x$$

- Para cadenas, suele utilizarse la siguiente fórmula:

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

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 $p = 131$)

¿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.

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

¿Cómo se implementa esta función *hash*?

- Las plantillas de C++ pueden particularizarse para tipos de datos concretos.
- Por ejemplo, implementación para el caso `K = int`.

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

¿Cómo se implementa esta función *hash*?

- Las plantillas de C++ pueden particularizarse para tipos de datos concretos.
- Por ejemplo, implementación para el caso `K = 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;
  }
};
```

Ejemplo

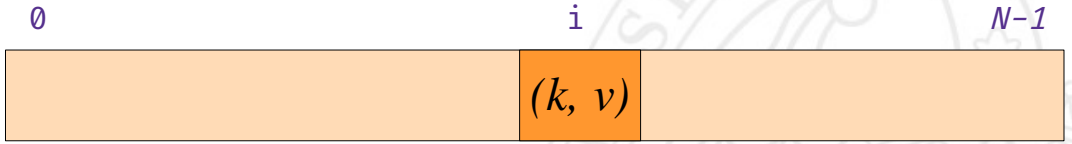
```
hash<int> h_int;
hash<std::string> h_str;

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

```
24
5273098
187271914
```

¿Cómo funcionan las tablas *hash*?

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

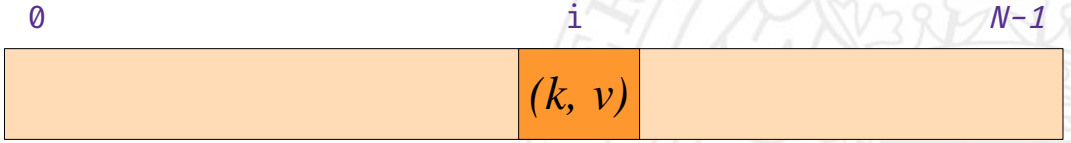
1) Calculamos $i := h(k) \bmod N$.
2) Insertamos el par (k, v) en la posición i-ésima del vector.



¿Cómo funcionan las tablas *hash*?

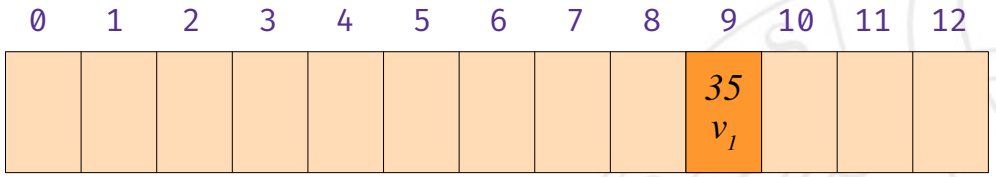
Supongamos que queremos **buscar** la entrada con clave k en un diccionario implementado mediante una tabla *hash* con N posiciones.

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



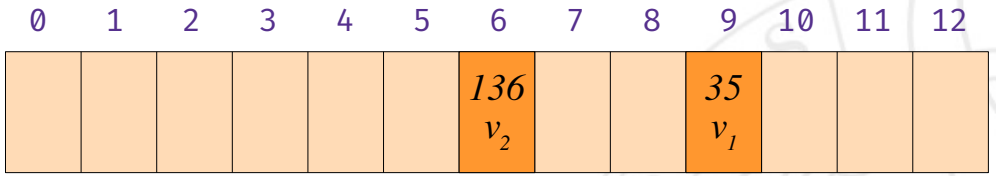
Ejemplo

- Con $N = 13$, queremos insertar la entrada $(35, v_1)$.
- Hacemos $h(35)$ **mod** $13 = 35$ **mod** $13 = 9$.



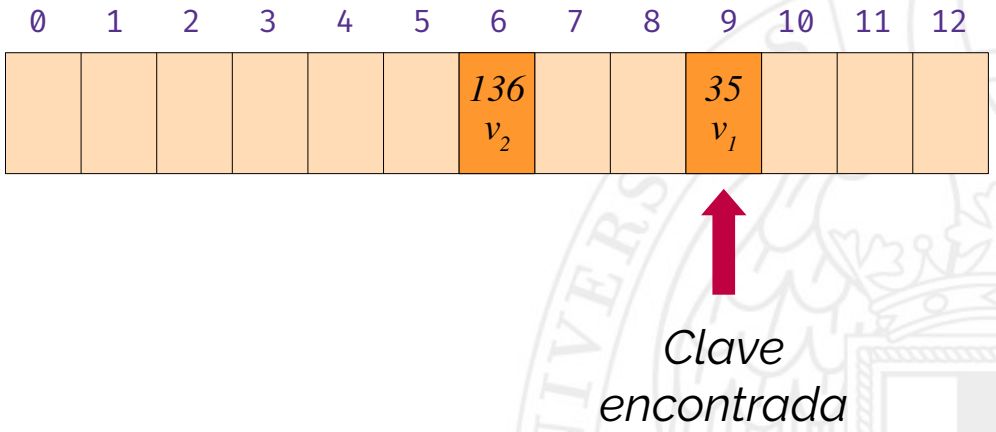
Ejemplo

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



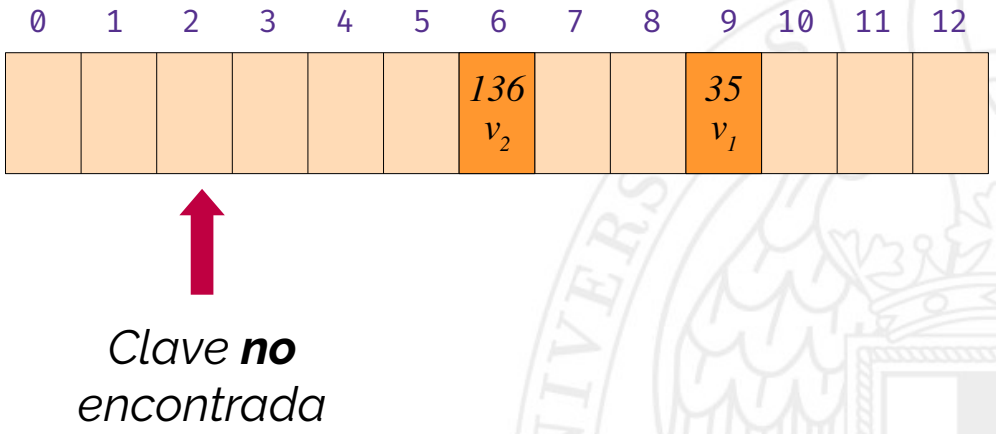
Ejemplo

- Buscamos la entrada con clave *35*.
- $h(35)$ **mod** $13 = 35$ **mod** $13 = 9$.



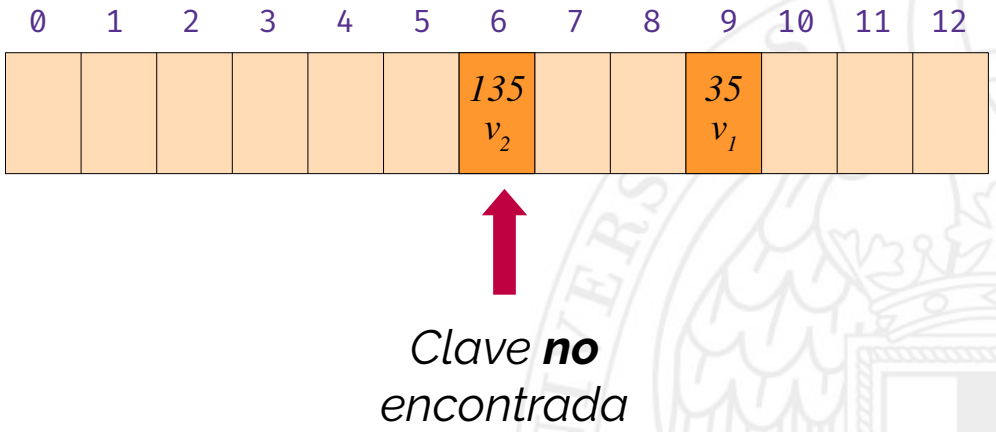
Ejemplo

- Buscamos la entrada con clave *41*.
- $h(41)$ **mod** $13 = 41$ **mod** $13 = 2$.



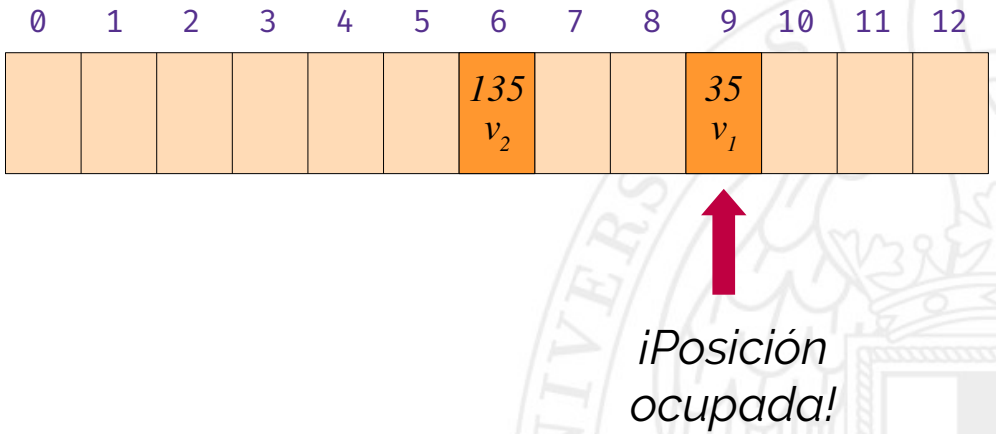
Ejemplo

- Buscamos la entrada con clave *149*.
- $h(149)$ **mod** $13 = 149$ **mod** $13 = 6$.



Ejemplo

- Insertamos la entrada $(61, v_3)$.
- $h(61)$ **mod** $13 = 61$ **mod** $13 = 9$.

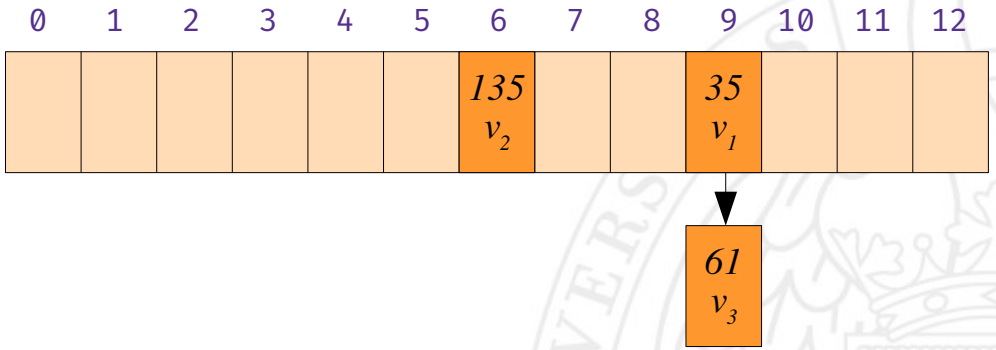


Colisiones

- 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.

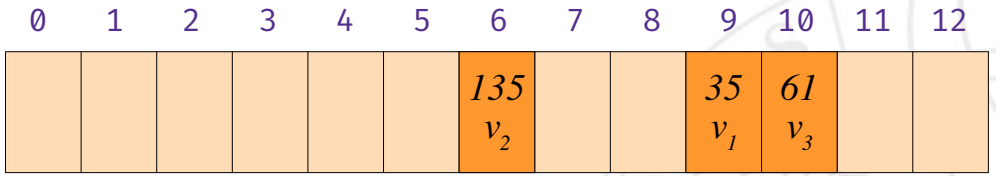
¿Cómo solucionamos las colisiones?

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



¿Cómo solucionamos las colisiones?

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



Implementaciones

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