

ESTRUCTURAS DE DATOS

DICCIONARIOS

# Tablas *hash* cerradas

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

# Objetivo

Funcionan de la siguiente manera. Si queremos buscar una entrada en la tabla calculamos su posición ( $\text{hash}(k) \bmod N$ ). Si está ocupada deberemos de buscar otra posición alternativa (tantas veces como se necesite) dentro de la tabla hash (veremos distintas formas). La forma que nosotros insertamos ese elemento es en las posiciones siguientes (las no ocupadas)

- Implementar el TAD Diccionario mediante una tabla *hash* cerrada.

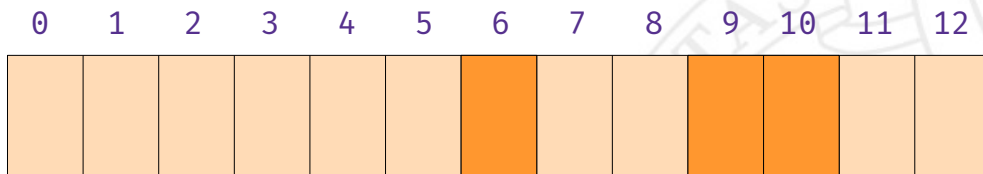
| 0 | 1 | 2 | 3 | 4 | 5 | 6     | 7 | 8 | 9     | 10    | 11 | 12 |
|---|---|---|---|---|---|-------|---|---|-------|-------|----|----|
|   |   |   |   |   |   | $k_2$ |   |   | $k_1$ | $k_3$ |    |    |
|   |   |   |   |   |   | $v_2$ |   |   | $v_1$ | $v_3$ |    |    |

En caso de colisión, no hay lista para cada bucket.

$k_1$   $k_3$

# Idea de implementación

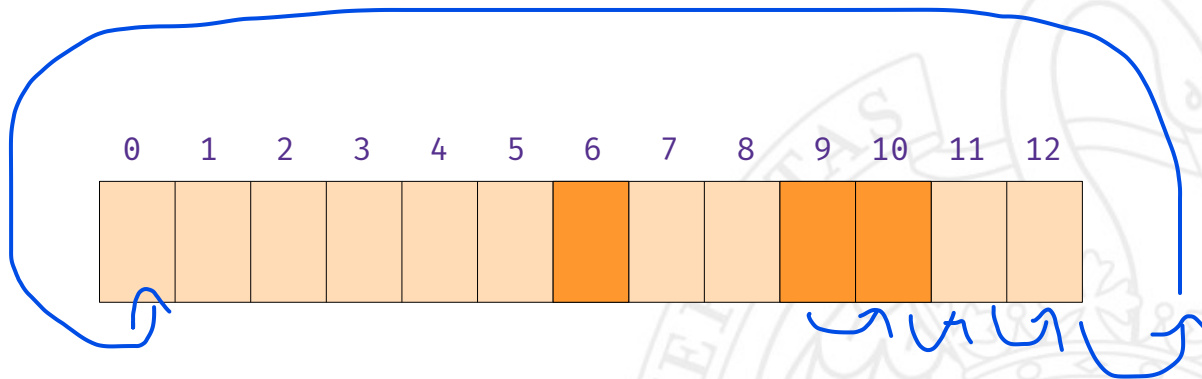
- Existen posiciones **libres** y **ocupadas**.



Necesitaremos un valor para saber si la entrada está libre u ocupada.

# Inserción

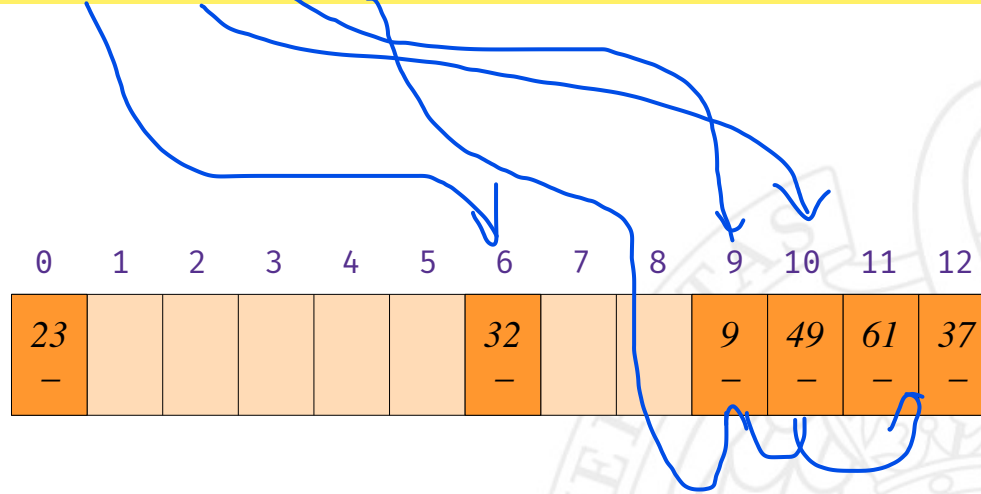
- Si la entrada está ocupada, insertamos en la siguiente. Si también está ocupada, miramos en la siguiente, etc. hasta encontrar una posición libre.



Como buscar aparcamiento.

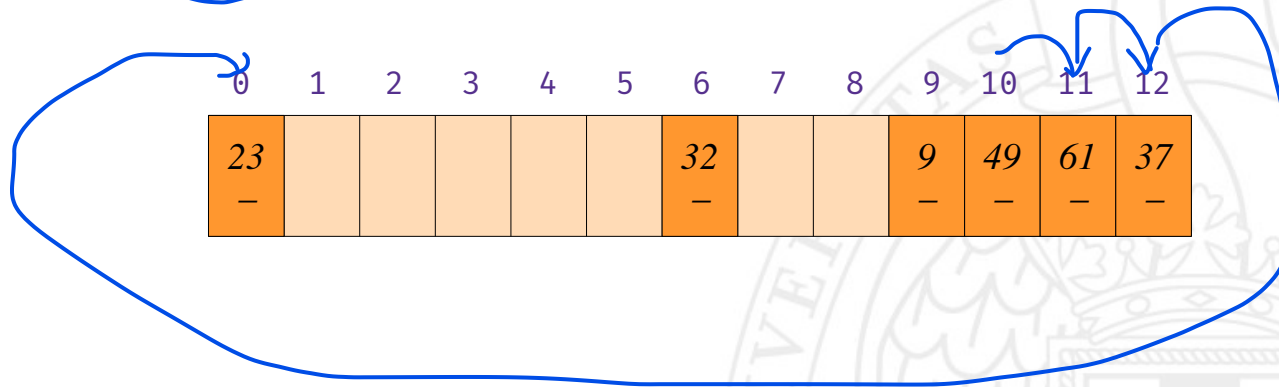
# Ejemplo

- Insertamos claves 32, 49, 9, 61, 37, 23 (en este orden).



# Búsqueda

- Buscamos en el cajón  $h(k) \bmod \text{CAPACITY}$ .
- A partir de ahí, buscamos en entradas sucesivas hasta encontrar la clave o llegar a una posición vacía.



- Ejemplo: buscamos 23 y 63.  
10

# ¡Cuidado con el borrado!

- Si eliminamos la entrada sin más, podemos imposibilitar la búsqueda de claves que vienen después.
- Ejemplo: borramos 61. *mod 13 = 0*

| 0  | 1 | 2 | 3 | 4 | 5 | 6  | 7 | 8 | 9 | 10 | 11 | 12 |
|----|---|---|---|---|---|----|---|---|---|----|----|----|
| 23 |   |   |   |   |   | 32 |   |   | 9 | 49 | 61 | 37 |
| -  |   |   |   |   |   | -  |   |   | - | -  | -  | -  |

Eliminaríamos esta entrada.

- ¿Y si ahora buscamos 23?

Al eliminar la entrada con el 61, si hiciéramos la búsqueda del 23,  $23 \bmod 13 = 10$ . Como la siguiente posición es vacía pararíamos de buscar ahí. PERO esto NO ES CORRECTO, porque el 23 está en la posición 1.

# Solución

- Distinguir entre entradas libres y entradas eliminadas.
- La búsqueda se detiene cuando llegamos a una posición libre.
- ...pero podemos escribir en entradas eliminadas.

Las eliminadas no son libres, pero OJO también se puede escribir en las entradas eliminadas.

| 0  | 1 | 2 | 3 | 4 | 5 | 6  | 7 | 8 | 9 | 10 | 11 | 12 |
|----|---|---|---|---|---|----|---|---|---|----|----|----|
| 23 |   |   |   |   |   | 32 |   |   | 9 | 49 |    | 37 |
| -  |   |   |   |   |   | -  |   |   | - | -  |    | -  |

- ¿Y si ahora buscamos 23?
- ¿Y si ahora insertamos la clave 36?

Si la podemos insertar aquí.



# Clase MapHash: interfaz pública

```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
public:
    MapHash();
    MapHash(const MapHash &other);
    ~MapHash();

    void insert(const MapEntry &entry);
    void erase(const K &key);

    bool contains(const K &key) const;
    const V & at(const K &key) const;
    V & at(const K &key);
    V & operator[](const K &key);

    int size() const;
    bool empty() const;

    MapHash & operator=(const MapHash &other);
    void display(std::ostream &out) const;

private:
    // ...
};
```

```
struct MapEntry {
    K key;
    V value;

    MapEntry(K key, V value);
    MapEntry(K key);
};
```

IGUAL QUE EN LAS TABLAS HASH ABIERTAS

# Clase MapHash: representación privada

```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
```

```
private:
```

```
enum class State { empty, occupied, deleted };
```

```
struct Bucket {
```

```
State state;
```

```
MapEntry entry; KEY+VALUE
```

```
Bucket(): state(State::empty) { }
};
```

```
Bucket *buckets;
```

```
Hash hash;
```

```
int num_elems;
```

```
};
```

Constructor predeterminado inicializa todos los buckets al estado de empty.

| 0  | 1 | 2 | 3 | 4 | 5 | 6  | 7 | 8 | 9 | 10 | 11 | 12 |
|----|---|---|---|---|---|----|---|---|---|----|----|----|
| 23 |   |   |   |   |   | 32 |   |   | 9 | 49 |    | 37 |
| -  |   |   |   |   |   | -  |   |   | - | -  |    | -  |

# Función auxiliar de búsqueda de clave

```
std::pair<int, int> search_pos(const K &key)
```

Búsqueda de posiciones

pos\_to\_insert

pos\_found

- Busca una clave **key** recibida como parámetro en el vector **buckets**.
- Componentes del objeto **pair** de salida:
  - pos\_found  
Posición del vector en la que se ha encontrado la clave. Si no se encuentra, es -1.
  - pos\_to\_insert  
Posición del vector en el que se debería insertar la clave, en caso de ser necesario (o -1 si el vector está lleno)

# Función auxiliar de búsqueda de clave

```
std::pair<int, int> search_pos(const K &key) const {  
    int h = hash(key) % CAPACITY;  
  
    int pos_to_insert = -1;  
    int pos_found = -1; Inicializamos variables que vamos a devolver a -1.  
  
    while (pos_found == -1 && buckets[h].state != State::empty) {  
        if (pos_to_insert == -1 && buckets[h].state == State::deleted) {  
            pos_to_insert = h; si su estado es deleted entonces sería la posición en la que insertar.  
        }  
  
        if (buckets[h].state == State::occupied && buckets[h].entry.key == key) {  
            pos_found = h; entonces hemos encontrado la clave que buscábamos  
        }  
  
        h = (h + 1) % CAPACITY;  
    }  
  
    if (pos_found == -1 && pos_to_insert == -1) {  
        pos_to_insert = h;  
    }  
    return {pos_to_insert, pos_found};  
}
```

# Función auxiliar de búsqueda de clave

```
std::pair<int, int> search_pos(const K &key) const {  
    int h = hash(key) % CAPACITY;  
  
    int pos_to_insert = -1;  
    int pos_found = -1;  
  
    while (pos_found == -1 && buckets[h].state != State::empty) {  
        if (pos_to_insert == -1 && buckets[h].state == State::deleted) {  
            pos_to_insert = h;  
        }  
  
        if (buckets[h].state == State::occupied && buckets[h].entry.key == key) {  
            pos_found = h;  
        }  
  
        h = (h + 1) % CAPACITY;  
    }  
  
    if (pos_found == -1 && pos_to_insert == -1) {  
        pos_to_insert = h;  
    }  
    return {pos_to_insert, pos_found};  
}
```

Repetimos mientras no  
hayamos encontrado  
la clave, o hayamos  
llegado a una posición  
vacía

# Función auxiliar de búsqueda de clave

```
std::pair<int, int> search_pos(const K &key) const {  
    int h = hash(key) % CAPACITY;  
  
    int pos_to_insert = -1;  
    int pos_found = -1;  
  
    while (pos_found == -1 && buckets[h].state != State::empty) {  
        if (pos_to_insert == -1 && buckets[h].state == State::deleted) {  
            pos_to_insert = h;  
        }  
  
        if (buckets[h].state == State::occupied && buckets[h].entry.key == key) {  
            pos_found = h;  
        }  
  
        h = (h + 1) % CAPACITY;  
    }  
  
    if (pos_found == -1 && pos_to_insert == -1) {  
        pos_to_insert = h;  
    }  
    return {pos_to_insert, pos_found};  
}
```

Cambiamos pos\_insert  
a la primera posición  
eliminada que  
encontremos

# Función auxiliar de búsqueda de clave

```
std::pair<int, int> search_pos(const K &key) const {  
    int h = hash(key) % CAPACITY;  
  
    int pos_to_insert = -1;  
    int pos_found = -1;  
  
    while (pos_found == -1 && buckets[h].state != State::empty) {  
        if (pos_to_insert == -1 && buckets[h].state == State::deleted) {  
            pos_to_insert = h;  
        }  
  
        if (buckets[h].state == State::occupied && buckets[h].entry.key == key) {  
            pos_found = h;  
        }  
  
        h = (h + 1) % CAPACITY;  
    }  
  
    if (pos_found == -1 && pos_to_insert == -1) {  
        pos_to_insert = h;  
    }  
    return {pos_to_insert, pos_found};  
}
```

Si encontramos la clave key en la posición actual, establecemos pos\_found

# Función auxiliar de búsqueda de clave

```
std::pair<int, int> search_pos(const K &key) const {
    int h = hash(key) % CAPACITY;

    int pos_to_insert = -1;
    int pos_found = -1;

    while (pos_found == -1 && buckets[h].state != State::empty) {
        if (pos_to_insert == -1 && buckets[h].state == State::deleted) {
            pos_to_insert = h;
        }

        if (buckets[h].state == State::occupied && buckets[h].entry.key == key) {
            pos_found = h;
        }

        h = (h + 1) % CAPACITY;
    }

    if (pos_found == -1 && pos_to_insert == -1) {
        pos_to_insert = h;
    }

    return {pos_to_insert, pos_found};
}
```

Pasamos a la siguiente posición del vector. Así hasta que nos salgamos

Pasamos a la posición siguiente. Si llegamos al final del vector, volvemos a la posición 0.



# Función auxiliar de búsqueda de clave

```
std::pair<int, int> search_pos(const K &key) const {
    int h = hash(key) % CAPACITY;

    int pos_to_insert = -1;
    int pos_found = -1;

    while (pos_found == -1 && buckets[h].state != State::empty) {
        if (pos_to_insert == -1 && buckets[h].state == State::deleted) {
            pos_to_insert = h;
        }

        if (buckets[h].state == State::occupied && buckets[h].entry.key == key) {
            pos_found = h;
        }

        h = (h + 1) % CAPACITY;
    }

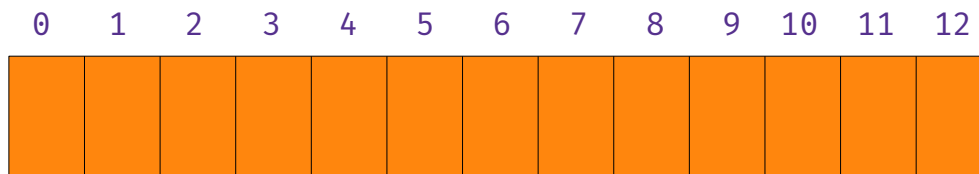
    if (pos_found == -1 && pos_to_insert == -1) {
        pos_to_insert = h;
    }

    return {pos_to_insert, pos_found};
}
```

Si al final hemos llegado a una posición libre, y no hemos pasado por ninguna borrada, establecemos `pos_to_insert`

# ¿Y si el vector está lleno?

Bueno pues el algoritmo anterior ESTÁ MAL, FALTAL, HORRIBLE.



Todos los buckets están ocupados

- Se van buscando posiciones de manera circular.
- ¡La función no termina!
- Debemos acotar el número de iteraciones.

Para ello haremos que como mucho, ese array se ejecute N veces, que es la capacidad del array.

# Función auxiliar de búsqueda de clave

```
std::pair<int, int> search_pos(const K &key) const {  
    int h = hash(key) % CAPACITY;  
    int cont = 0;  
  
    int pos_to_insert = -1;  
    int pos_found = -1;  
  
    while (cont < CAPACITY && pos_found == -1 && buckets[h].state != State::empty) {  
        if (pos_to_insert == -1 && buckets[h].state == State::deleted) {  
            pos_to_insert = h;  
        }  
  
        if (buckets[h].state == State::occupied && buckets[h].entry.key == key) {  
            pos_found = h;  
        }  
  
        h = (h + 1) % CAPACITY;  
        cont++;  
    }  
  
    if (cont < CAPACITY && pos_to_insert == -1) {  
        pos_to_insert = h;  
    }  
    return {pos_to_insert, pos_found};  
}
```

Si es igual sale del bucle

Si esto ocurre entonces HAY alguna posición libre. Si fuera cont = CAPACITY no la habría.

# Método insert

```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
public:
```

```
void insert(const MapEntry &entry) {
    auto [pos_to_insert, pos_found] = search_pos(entry.key);
    if (pos_found == -1) {
        assert (pos_to_insert != -1);
        buckets[pos_to_insert].state = State::occupied;
        buckets[pos_to_insert].entry = entry;
        num_elems++;
    }
}
```

```
private:
    // ...
};
```

Si `pos_fund != -1` implica que se ha encontrado la clave en el mapa y que por tanto no podríamos insertarlo. Si `pos_fund == 1` implica que no lo hemos encontrado y devuelve la primera posición libre. Cambiamos su estado y metemos el mapentry completo (clave-valor).

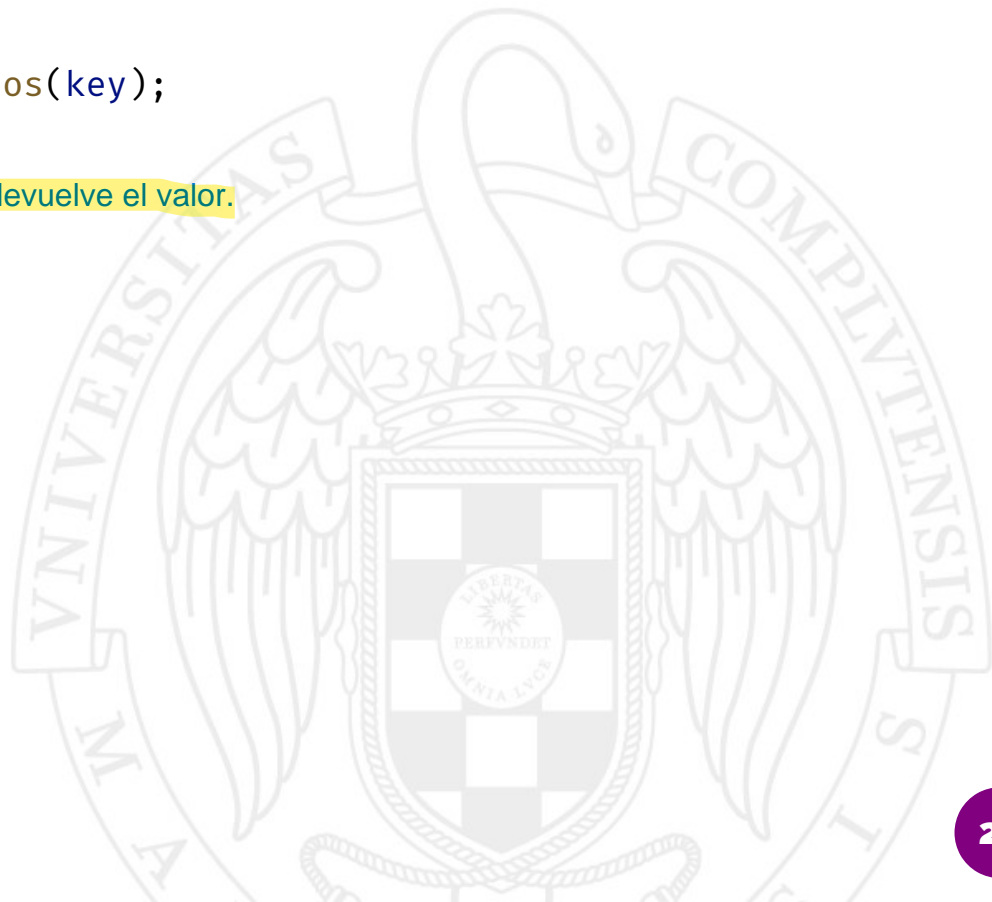
# Método at

```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
public:

    const V & at(const K &key) const {
        auto [pos_to_insert, pos_found] = search_pos(key);
        assert (pos_found != -1);
        return buckets[pos_found].entry.value;
    }

private:
    // ...
};
```

devuelve el valor.



# Método erase

```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
public:

    void erase(const K &key) {
        auto [pos_to_insert, pos_found] = search_pos(key);
        if (pos_found != -1) {
            buckets[pos_found].state = State::deleted;
            num_elems--;
        }
    }

private:
    // ...
};
```



# Búsqueda de posiciones alternativas

VEREMOS OTRAS FORMAS DE BUSCAR POSICIONES ALTERNATIVAS.

# Recordatorio

- Las tablas hash cerradas se basan en calcular una **posición inicial**  $p_0$  en el vector y buscar una clave allí.

$$p_0 = h(k) \bmod \text{CAPACITY}$$

- Si la clave no se encuentra, se busca en **posiciones alternativas**  $p_1, p_2, \text{etc.}$

En nuestro caso:

$$p_1 = (\underline{h(k)} + 1) \bmod \text{CAPACITY}$$

$$p_2 = (\underline{h(k)} + 2) \bmod \text{CAPACITY}$$

$$p_3 = (\underline{h(k)} + 3) \bmod \text{CAPACITY}$$

etc.



# Recordatorio

- Las tablas hash cerradas se basan en calcular una **posición inicial**  $p_0$  en el vector y **buscar una clave allí**.

$$p_0 = h(k) \bmod \text{CAPACITY}$$

- Si la clave no se encuentra, se busca en **posiciones alternativas**  $p_1, p_2, \text{etc.}$

En general:

$$p_i = (h(k) + i) \bmod \text{CAPACITY}$$

Sumarle  $i$

# Sondeo lineal

- Existen otras posibilidades para la búsqueda de posiciones alternativas.
- La que hemos utilizado es un caso particular del **sondeo lineal**.

$$p_i = (h(k) + ci) \bmod \text{CAPACITY}$$

$$c = 1$$

- Por ejemplo, si  $c = 1$ ,  $h(k) = 10$ ,  $\text{CAPACITY} = 13$ .

10 11 12 0 1 2 3 ...

# Sondeo lineal

- Existen otras alternativas para la búsqueda de posiciones alternativas.
- La que hemos utilizado es un caso particular del **sondeo lineal**.

$$p_i = (h(k) + ci) \bmod \text{CAPACITY}$$

Hasta ahora hemos utilizado que  $c = 1$ , pero... ¿y si  $c$  tuviera otros valores?

- Por ejemplo, si  $c = 2$ ,  $h(k) = 10$ ,  $\text{CAPACITY} = 13$ .

10    12    1    3    5    7    9    11    ...

Es como si fuéramos de 2 en 2 en el vector.

# Sondeo lineal

- Existen otras alternativas para la búsqueda de posiciones alternativas.
- La que hemos utilizado es un caso particular del **sondeo lineal**.

$$p_i = (h(k) + ci) \bmod \text{CAPACITY}$$

- Por ejemplo, si  $c = 5$ ,  $h(k) = 10$ ,  $\text{CAPACITY} = 13$ .

10   2   7   12   4   9   ...

Saltamos 5 posiciones a la derecha.

En general con el 5 acabaríamos recorriendo todo el vector porque son PRIMOS.

# Sondeo lineal

- Existen otras alternativas para la búsqueda de posiciones alternativas.
- La que hemos utilizado es un caso particular del **sondeo lineal**.

$$p_i = (h(k) + ci) \bmod CAPACITY$$

- En general, si  $\text{mcd}(c, CAPACITY) = 1$ , el sondeo lineal garantiza el recorrido de todas las posiciones del array, en caso de ser necesario.
- En general, esto ocurre cuando CAPACITY es primo.

Vamos, que acabarías recorriendo todo el vector si ocurre que el mcd de c y capacity es 1.

# Sondeo cuadrático

- Utiliza la siguiente fórmula:

$$p_i = (h(k) + i^2) \bmod \text{CAPACITY}$$

Le suma  $i$  cuadrado al código hash que hemos calculado

- Por ejemplo, si  $h(k) = 10$ ,  $\text{CAPACITY} = 13$ .

10 11 1 6 0 9 7 7 ...

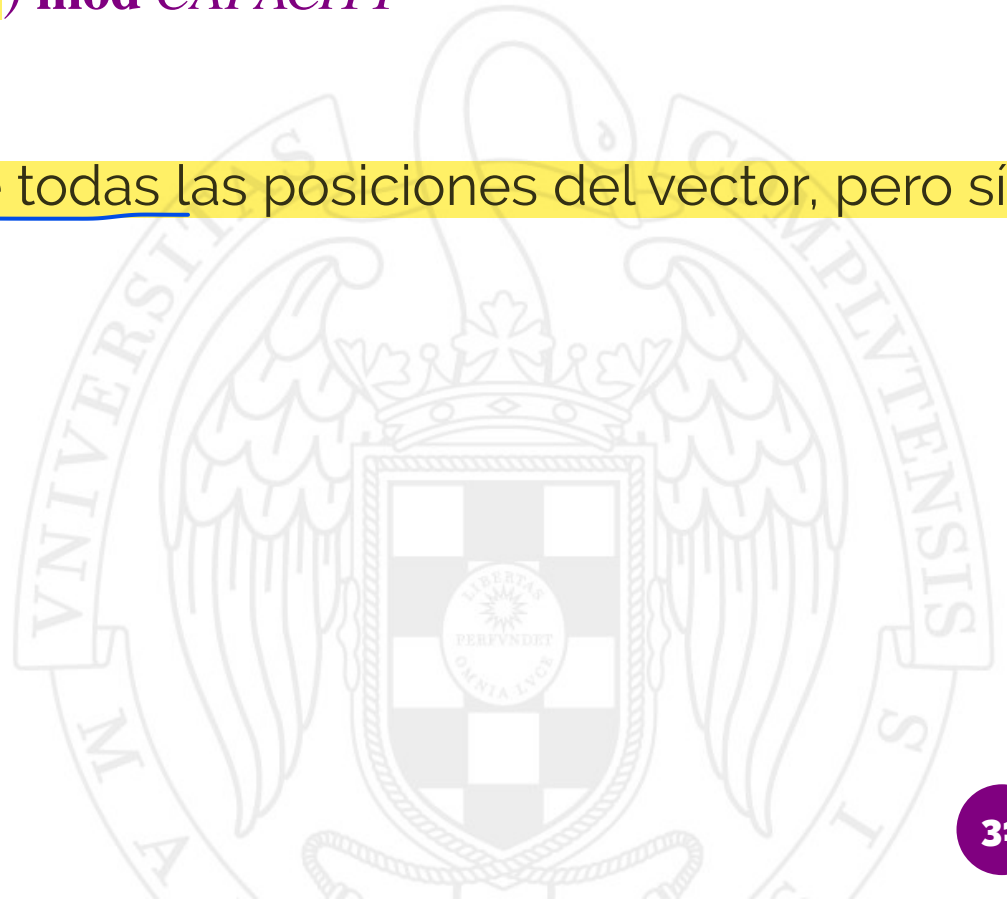
posicion 10 luego la  $1^2$  luego la  $2^2$  luego  $3^2$ ...

# Sondeo cuadrático

- Utiliza la siguiente fórmula:

$$p_i = (h(k) + i^2) \bmod \text{CAPACITY}$$

- Aquí no se garantiza el recorrido de todas las posiciones del vector, pero sí al menos la mitad de ellas.



# Doble redispersión

- Utiliza una segunda función hash  $h'$  para la búsqueda de posiciones alternativas.

$$p_i = (h(k) + i h'(k)) \bmod \text{CAPACITY}$$

- Garantiza el recorrido de todas las posiciones si  $\text{CAPACITY}$  es primo y  $h'(k)$  nunca devuelve 0.



# Más información

- R. Peña

*Diseño de Programas. Formalismo y Abstracción (3ª edición)*

Pearson Educación (2005)

Sección 8.1.3

- [https://en.wikipedia.org/wiki/Linear\\_probing](https://en.wikipedia.org/wiki/Linear_probing)
- [https://en.wikipedia.org/wiki/Quadratic\\_probing](https://en.wikipedia.org/wiki/Quadratic_probing)
- [https://en.wikipedia.org/wiki/Double\\_hashing](https://en.wikipedia.org/wiki/Double_hashing)

Más información del sondeo

