

ESTRUCTURAS DE DATOS

DICCIONARIOS

El TAD Diccionario

Manuel Montenegro Montes

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

Motivación

- Leer un texto de entrada e imprimir el número de veces que aparece cada palabra.

David tomó la llave para entregársela a Laura. Esta última, no obstante, declinó hacer uso de la llave mientras que no fuera absolutamente necesario.

David 1
tomó 1
la 2
llave 2
para 1
...

- ¿Cómo almacenamos las palabras que nos encontramos?

Motivación

- Tabla en la que almacenamos el número de veces que aparece cada palabra encontrada hasta el momento.
- Con cada palabra recibida:
 - Si existe una entrada en la tabla con esa palabra, incrementamos su contador.
 - Si no, insertamos una nueva entrada con esa palabra y con su contador a 1.

Palabra	Contador
“David”	1
“tomó”	1
“la”	2
“llave”	2
...	...

¿Qué es un diccionario?

- Un tipo abstracto de datos que almacena un conjunto de pares.
- A cada par se le llama **entrada**.
- A la primera componente de cada par se le denomina **clave**.
- A la segunda componente se le denomina **valor asociado** a esa clave.
- No existen dos pares con la misma clave.

Palabra	Contador
"David"	1
"tomó"	1
"la"	2
"llave"	2
...	...

Claves

Valores

Terminología

diccionarios

tablas

arrays asociativos

maps

associative arrays

dictionaries

symbol tables

Palabra	Contador
"David"	1
"tomó"	1
"la"	2
"llave"	2
...	...

Modelo conceptual de diccionarios

- Sean:
 - K – conjunto de claves
 - V – conjunto de valores
- Un diccionario M es un conjunto de pares (k, v) , donde $k \in K$, $v \in V$.
- No existen pares $(k, v), (k, v') \in M$ tales que $v \neq v'$.

Palabra	Contador
“David”	1
“tomó”	1
“la”	2
“llave”	2
...	...

$$M = \{ (“David”, 1), (“tomó”, 1), (“la”, 2), \dots \}$$

Operaciones en el TAD Diccionario

- Constructoras:
 - Crear un diccionario vacío: ***create_empty***
- Mutadoras:
 - Añadir una entrada al diccionario: ***insert***
 - Eliminar una entrada del diccionario: ***erase***
- Observadoras:
 - Saber si existe una entrada con una clave determinada: ***contains***
 - Saber el valor asociado con una clave: ***at***
 - Saber si el diccionario está vacío: ***empty***
 - Saber el número de entradas del diccionario: ***size***

Operaciones constructoras y mutadoras

$\{ \text{true} \}$

create_empty() $\rightarrow (M: \text{Map})$

$\{ M = \emptyset \}$

$\{ \text{true} \}$

insert($k: \text{Key}, v: \text{Value}, M: \text{Map}$)

$M = \begin{cases} \text{old}(M) & \text{si } \exists v'. (k, v') \in \text{old}(M) \\ \text{old}(M) \cup \{(k, v)\} & \text{en otro caso} \end{cases}$

$\{ \text{true} \}$

erase($k: \text{Key}, M: \text{Map}$)

$M = \{ (k', v') \in \text{old}(M) \mid k' \neq k \}$

Operaciones observadoras

$\{ \text{true} \}$

contains(k : Key, M : Map) $\rightarrow (b$: bool)

$\{ b \Leftrightarrow \exists v. (k, v) \in M \}$

$\{ \exists v'. (k, v') \in M \}$

at(k : Key, M : Map) $\rightarrow (v$: value)

$\{ (k, v) \in M \}$

$\{ \text{true} \}$

empty(M : Map) $\rightarrow (b$: bool)

$\{ b \Leftrightarrow M = \emptyset \}$

$\{ \text{true} \}$

size(M : Map) $\rightarrow (n$: int)

$\{ n = |M| \}$

Interfaz en C++

```
template <typename K, typename V>
class map {
public:
    map();
    map(const map &other);
    ~map();

    void insert(const K &key, const V &value);
    void erase(const K &key);

    bool contains(const K &key) const;

    const V & at(const K &key) const;
    V & at(const K &key);

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

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



Interfaz en C++

```
template <typename K, typename V>
class map {
public:
    map();
    map(const map &other);
    ~map();

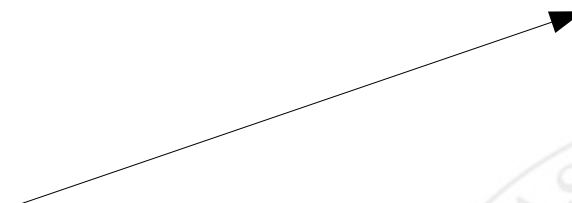
    void insert(const map_entry &entry);
    void erase(const K &key);

    bool contains(const K &key) const;

    const V & at(const K &key) const;
    V & at(const K &key);

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

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



```
struct map_entry {
    K key;
    V value;
};
```

Ejemplo

```
map<string, int> personas;
```

```
personas.insert({"Aarón", 42});  
personas.insert({"Estela", 41});
```

```
cout << personas.contains("Aarón") << endl;  
cout << personas.at("Aarón") << endl;
```

```
personas.insert({"Carlos", 31});
```

```
personas.erase("Estela");
```

```
personas.at("Aarón") = 43;
```

personas = \emptyset

personas = {("Aarón", 42), ("Estela", 41) }

true
42

personas = {("Aarón", 42), ("Carlos", 31), ("Estela", 41) }

personas = {("Aarón", 42), ("Carlos", 31) }

personas = {("Aarón", 43), ("Carlos", 31) }

Ejemplo

```
string palabra;  
map<string, int> dicc;  
  
cin >> palabra;  
  
while (!cin.eof()) {  
    if (dicc.contains(palabra)) {  
        dicc.at(palabra)++;  
    } else {  
        dicc.insert({palabra, 1});  
    }  
  
    cin >> palabra;  
}
```



Dos implementaciones

- Mediante **árboles binarios de búsqueda** (MapTree)
- Mediante **tablas *hash*** (MapHash)



ESTRUCTURAS DE DATOS

DICCIONARIOS

Diccionarios mediante árboles binarios de búsqueda

Manuel Montenegro Montes

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

Operaciones en el TAD Diccionario

- Constructoras:
 - Crear un diccionario vacío: ***create_empty***
- Mutadoras:
 - Añadir una entrada al diccionario: ***insert***
 - Eliminar una entrada del diccionario: ***erase***
- Observadoras:
 - Saber si existe una entrada con una clave determinada: ***contains***
 - Saber el valor asociado con una clave: ***at***
 - Saber si el diccionario está vacío: ***empty***
 - Saber el número de entradas del diccionario: ***size***

Dos implementaciones

- Mediante **árboles binarios de búsqueda** (MapTree) ←
- Mediante **tablas *hash*** (MapTable)

Este vídeo



Interfaz de MapTree

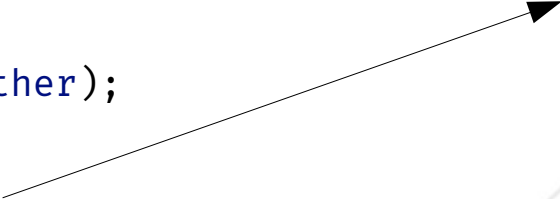
```
template <typename K, typename V>
class MapTree {
public:
    MapTree();
    MapTree(const MapTree &other);
    ~MapTree();

    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);

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

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



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

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

Representación privada de MapTree

```
template <typename K, typename V>
class MapTree {
    ...
private:
```

```
    struct Node {
        MapEntry entry;
        Node *left, *right;
```

```
        Node(Node *left, const MapEntry &entry, Node *right);
    };
```

```
    Node *root_node;
    int num_elems;
```

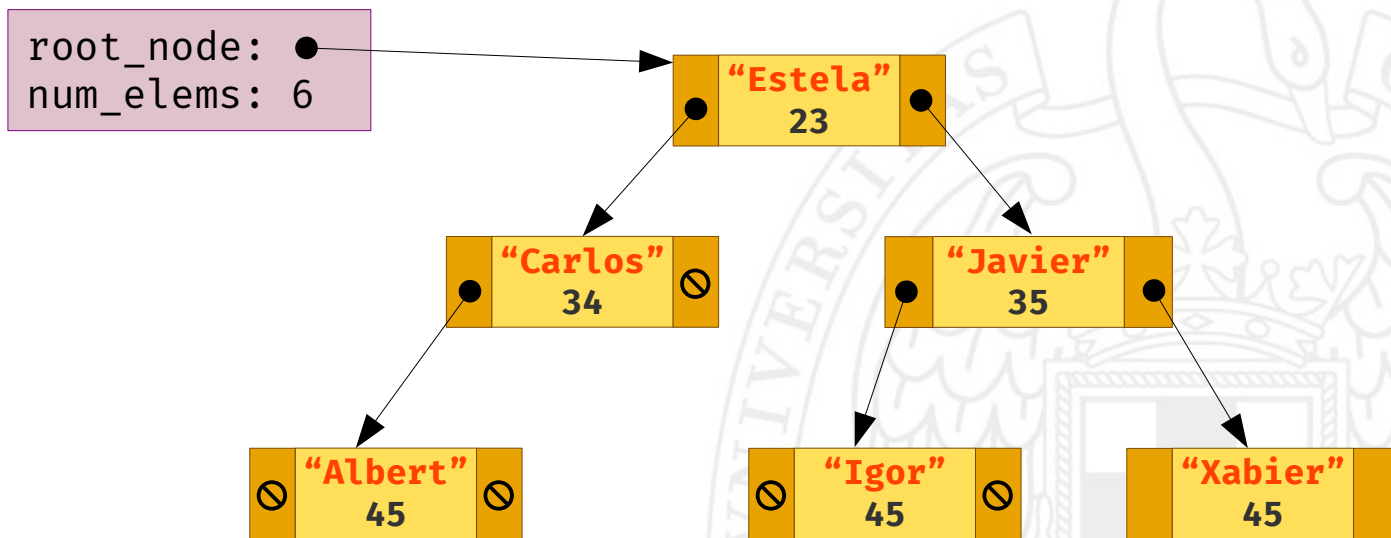
```
    // métodos auxiliares privados
    // ...
};
```

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

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

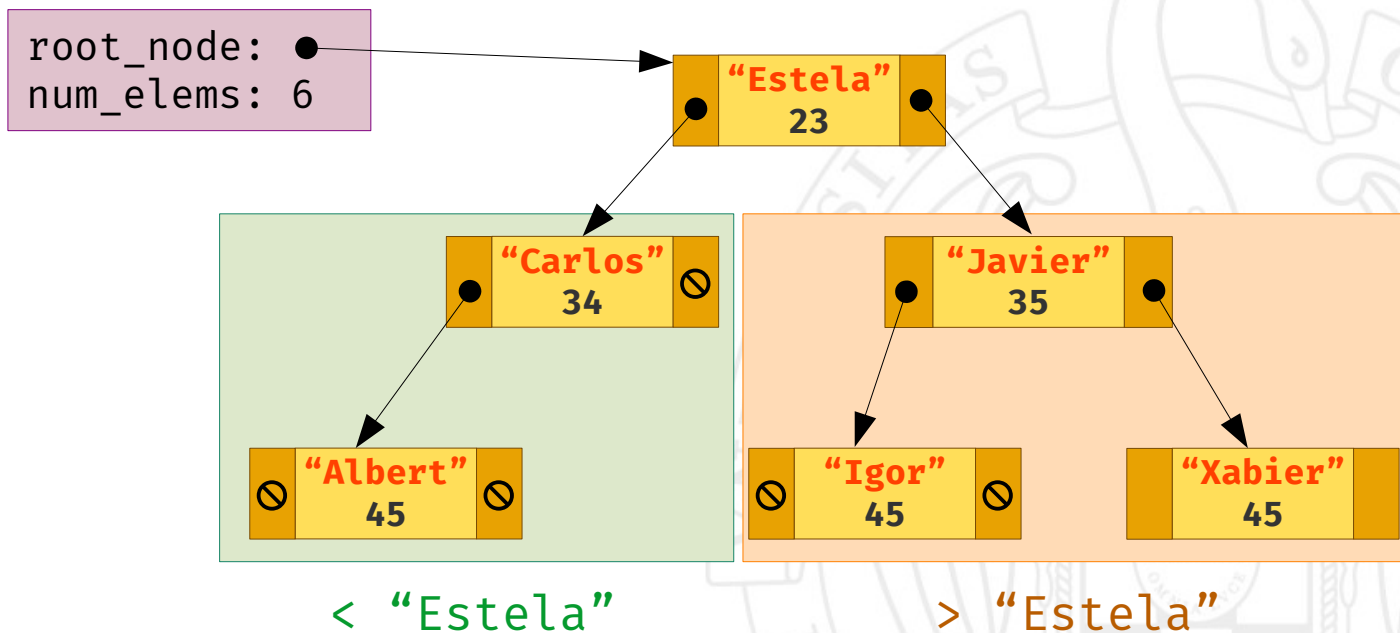
Representación de un MapTree

$\{("Carlos", 34), ("Estela", 23), ("Xabier", 45), ("Igor", 45), ("Javier", 35), ("Albert", 45)\}$



Representación de un MapTree

- El orden de los elementos en el árbol binario de búsqueda viene determinado por el orden de las claves.



Métodos auxiliares

```
template <typename K, typename V>
class MapTree {
    ...
private:
    ...

    static std::pair<Node *, bool> insert(Node *root, const MapEntry &elem);
    static Node * search(Node *root, const K &key);
    static std::pair<Node *, bool> erase(Node *root, const K &key);
};
```

- Iguales que los utilizados en ABBs.
- Diferencia: se realizan comparaciones entre **las claves**.

Métodos auxiliares

```
template <typename K, typename V>
class MapTree {
    ...
private:
    ...

    static Node * search(Node *root, const K &key) {
        if (root == nullptr) {
            return nullptr;
        } else if (key < root->entry.key) {
            return search(root->left, key);
        } else if (root->entry.key < key) {
            return search(root->right, key);
        } else {
            return root;
        }
    }
};
```



Métodos contains() y at()

```
template <typename K, typename V>
class MapTree {
public:
    ...
    bool contains(const K &key) const {
        return search(root_node, key) != nullptr;
    }

    const V & at(const K &key) const {
        Node *result = search(root_node, key);
        assert (result != nullptr);
        return result->entry.value;
    }

    V & at(const K &key) {
        Node *result = search(root_node, key);
        assert (result != nullptr);
        return result->entry.value;
    }
};
```



Búsqueda e inserción mediante []



Motivación

- Muchas veces encontramos código como este:

```
if (!dicc.contains(k)) {  
    words.insert({k, 1});  
} else {  
    words.at(k) = ... ;  
}
```

- No es equivalente a:

```
if (!dicc.contains(k)) {  
    words.at(k) = 1;  
} else {  
    words.at(k) = ... ;  
}
```

Error: at() exige que la clave se encuentre en el diccionario

Motivación

Definimos una operación alternativa a `at()`, llamada `operator[]`.

`dicc.at(key)`

- Devuelve una referencia al valor asociado con la clave `key`.
- Si `key` no se encuentra, se produce un error.

`dicc[key]`

- Devuelve una referencia al valor asociado con la clave `key`.
- Si `key` no se encuentra, se añade una nueva entrada a `dicc` que asocia `key` con un valor por defecto.

Nueva operación auxiliar

```
template <typename K, typename V>
class MapTree {
...
private:
...
static std::tuple<bool, Node *, Node *> search_or_insert(Node *root,
                                                         const K &key) {
    if (root == nullptr) {
        Node *new_node = new Node(nullptr, {key}, nullptr);
        return {true, new_node, new_node};
    } else if (key < root->entry.key) {
        auto [inserted, new_root, found_node] = search_or_insert(root->left, key);
        root->left = new_root;
        return {inserted, root, found_node};
    } else if (root->entry.key < key) {
        auto [inserted, new_root, found_node] =
            search_or_insert(root->right, key);
        root->right = new_root;
        return {inserted, root, found_node};
    } else {
        return {false, root, root};
    }
}
```

Nueva operación auxiliar

```
template <typename K, typename V>
class MapTree {
    ...
private:
    ...
    static std::tuple<bool, Node *, Node *> search_or_insert(Node *root,
                                                             const K &key) {
        if (root == nullptr) {
            Node *new_node = new Node(nullptr, {key}, nullptr);
            return {true, new_node, new_node};
        } else if (key < root->entry.key) {
            auto [inserted, new_root, found_node] = search_or_insert(root->left, key);
            root->left = new_root;
            return {inserted, root, found_node};
        } else if (root->entry.key < key) {
            auto [inserted, new_root, found_node] =
                search_or_insert(root->right, key);
            root->right = new_root;
            return {inserted, root, found_node};
        } else {
            return {false, root, root};
        }
    }
}
```

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

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

Nueva operación auxiliar

```
template <typename K, typename V>
class MapTree {
    ...
private:
    ...
    static std::tuple<bool, Node *, Node *> search_or_insert(Node *root,
                                                             const K &key) {
        if (root == nullptr) {
            Node *new_node = new Node(nullptr, {key}, nullptr);
            return {true, new_node, new_node};
        } else if (key < root->entry.key) {
            auto [inserted, new_root, found_node] = search_or_insert(root->left, key);
            root->left = new_root;
            return {inserted, root, found_node};
        } else if (root->entry.key < key) {
            auto [inserted, new_root, found_node] = search_or_insert(root->right, key);
            root->right = new_root;
            return {inserted, root, found_node};
        } else {
            return {false, root, root};
        }
    }
}
```

Nueva operación auxiliar

```
template <typename K, typename V>
class MapTree {
...
private:
...
static std::tuple<bool, Node *, Node *> search_or_insert(Node *root,
                                                         const K &key) {
    if (root == nullptr) {
        Node *new_node = new Node(nullptr, {key}, nullptr);
        return {true, new_node, new_node};
    } else if (key < root->entry.key) {
        auto [inserted, new_root, found_node] = search_or_insert(root->left, key);
        root->left = new_root;
        return {inserted, root, found_node};
    } else if (root->entry.key < key) {
        auto [inserted, new_root, found_node] = search_or_insert(root->right, key);
        root->right = new_root;
        return {inserted, root, found_node};
    } else {
        return {false, root, root};
    }
}
```

Implementación de operator[]

```
template <typename K, typename V>
class MapTree {
public:
    ...

    V &operator[](const K &key) {
        auto [inserted, new_root, found_node] = search_or_insert(root_node, key);
        this->root_node = new_root;
        if (inserted) { num_elems++; }
        return found_node->entry.value;
    }
};
```




Resultado

```
if (!dicc.contains(k)) {  
    words.insert({k, 1});  
} else {  
    words.at(k) = ... ;  
}
```



```
if (!dicc.contains(k)) {  
    words.at(k) = 1;  
} else {  
    words.at(k) = ... ;  
}
```



Resultado

```
if (!dicc.contains(k)) {  
    words.insert({k, 1});  
} else {  
    words.at(k) = ... ;  
}
```



```
if (!dicc.contains(k)) {  
    words[k] = 1;  
} else {  
    words[k] = ... ;  
}
```



Coste de las operaciones

Operación	Árbol equilibrado	Árbol no equilibrado
<i>constructor</i>	$O(1)$	$O(1)$
<i>empty</i>	$O(1)$	$O(1)$
<i>size</i>	$O(1)$	$O(1)$
<i>contains</i>	$O(\log n)$	$O(n)$
<i>at</i>	$O(\log n)$	$O(n)$
<i>operator[]</i>	$O(\log n)$	$O(n)$
<i>insert</i>	$O(\log n)$	$O(n)$
<i>erase</i>	$O(\log n)$	$O(n)$

n = número de entradas en el diccionario

ESTRUCTURAS DE DATOS

DICCIONARIOS

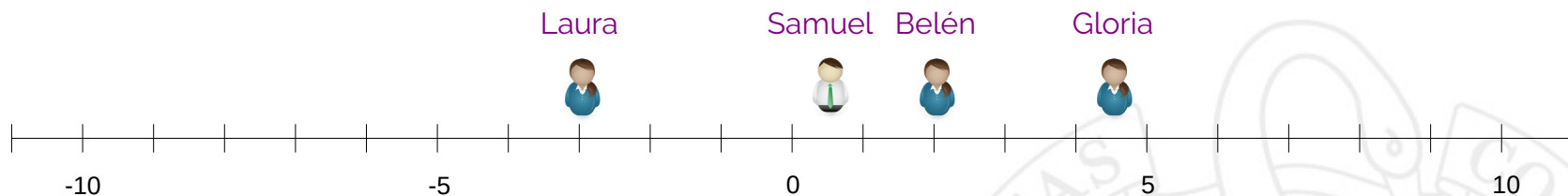
Relaciones de orden en ABBs

Manuel Montenegro Montes

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

Motivación

- Queremos simular el movimiento de varias personas en una calle recta:

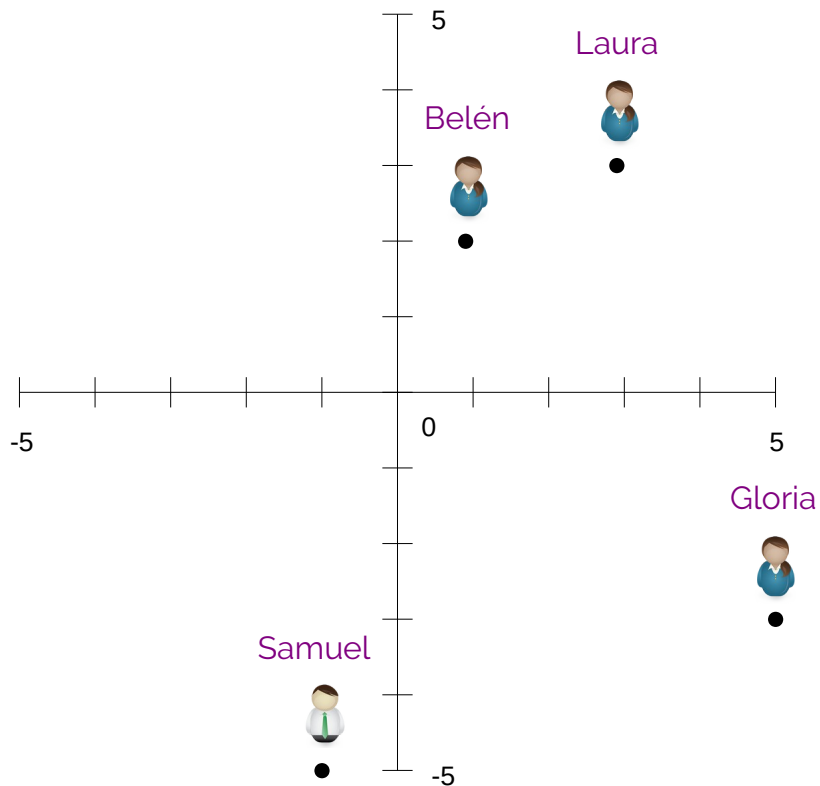


- ¿Cómo represento el estado actual?

```
MapTree<double, string> posiciones;  
posiciones.insert({-3, "Laura"});  
posiciones.insert({4.5, "Gloria"});  
posiciones.insert({2, "Belén"});  
posiciones.insert({0.5, "Samuel"});
```

Motivación

- ¿Hacemos lo mismo, pero ahora en un plano?



```
struct Coords {  
    double x;  
    double y;  
};
```

```
MapTree<Coords, string> posiciones;  
posiciones.insert({{3, 3}, "Laura"});  
posiciones.insert({{5, -3}, "Gloria"});  
posiciones.insert({{1, 2}, "Belén"});  
posiciones.insert({{-1, -5}, "Samuel"});
```



¿Qué ha pasado?

- Obtenemos el siguiente error:

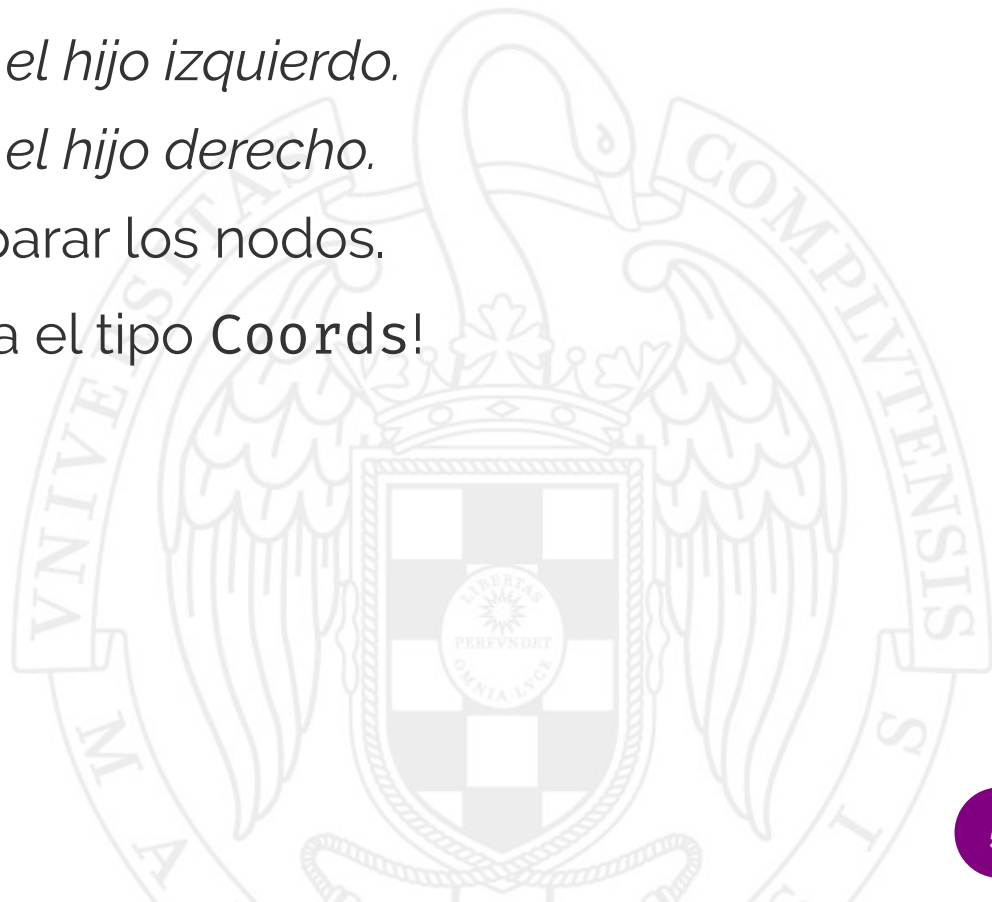
```
map_tree.h:127:30: error: no match for 'operator<' (operand types are 'const Coords'
and 'Coords')
```

```
127 |           } else if (entry.key < root->entry.key) {
    |                               ~~~~~^~~~~~
```



¿Qué ha pasado?

- A la hora de insertar nodos de un árbol binario de búsqueda, comparamos la clave que queremos insertar con algunos de los nodos del árbol:
 - *Si $clave < nodo.clave$, insertar en el hijo izquierdo.*
 - *Si $nodo.clave < clave$, insertar en el hijo derecho.*
- Utilizamos el operador $<$ para comparar los nodos.
- ¡Este operador no está definido para el tipo `Coords`!



Solución 1: implementar <



Solución 1

```
struct Coords {  
    double x;  
    double y;  
};  
  
bool operator<(const Coords &p1,  
               const Coords &p2) {  
    return p1.x < p2.x  
        || p1.x == p2.x && p1.y < p2.y;  
}
```

- Orden **lexicográfico**:

Compara las coordenadas x.
En caso de igualdad, compara
las coordenadas y.

¿Sirve cualquier definición de $<$?

- Tiene que cumplir las siguientes propiedades:
 - **Antirreflexiva:** Nunca se cumple $a < a$ para ningún a .
 - **Asimétrica:** Si $a < b$, entonces no se cumple $b < a$.
 - **Transitiva:** Si $a < b$ y $b < c$, entonces $a < c$.

El compilador no comprueba que el `operator<` que definamos cumpla estas tres propiedades, pero si no las cumple, el ABB puede comportarse de manera inconsistente.

- La definición que escojamos determina el orden en el que iteremos sobre las entradas de un árbol.

Problemas

- Si definimos el operador $<$ para un tipo de datos, este se aplica a todos los `MapTree` que utilicen ese tipo como clave.
- ¿Y si quiero utilizar una relación de orden para un `MapTree`, y otra relación distinta para otro `MapTree`?



Solución 2: parametrizar MapTree



Parametrizar MapTree

- Indicamos cómo comparar las claves mediante un **objeto función**.
- Consiste en añadir un tercer parámetro de tipo a MapTree:

MapTree<K, V, Comparator>

Tipo de las claves

Tipo de los valores

Tipo del objeto función
que compara las claves

Parametrizar MapTree

- Indicamos cómo comparar las claves mediante un **objeto función**.
- Consiste en añadir un tercer parámetro de tipo a MapTree:

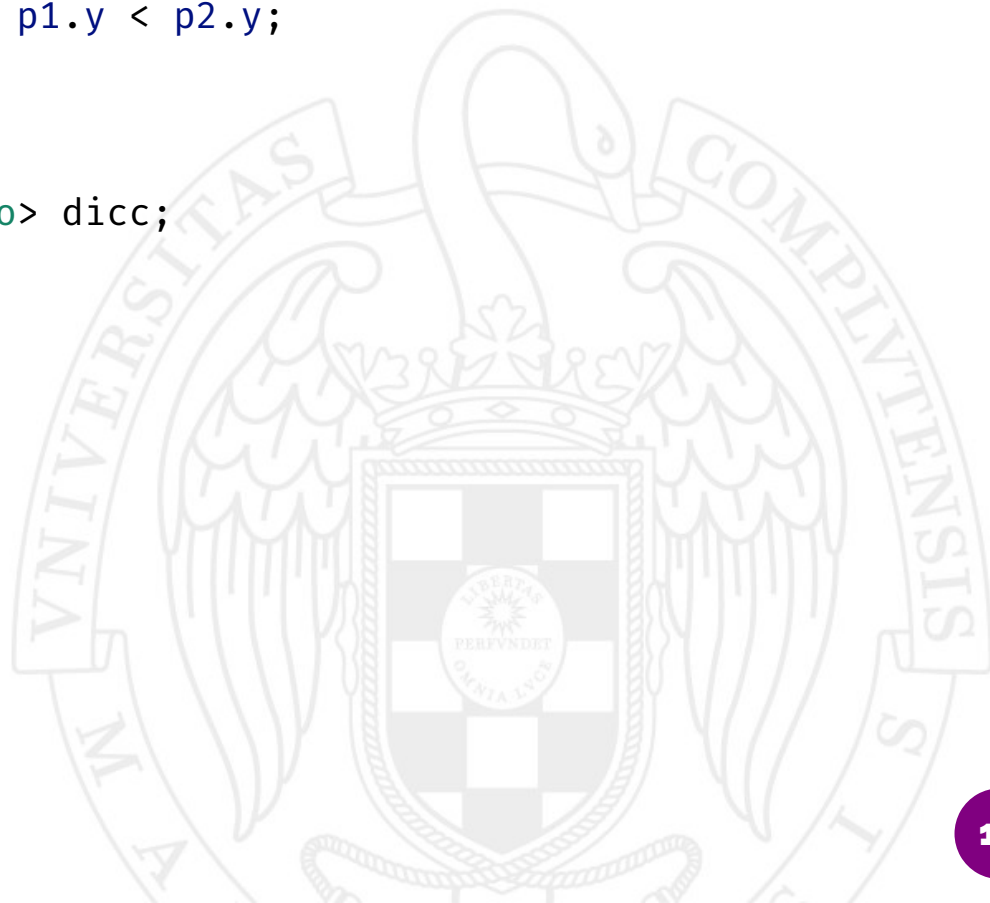
`MapTree<K, V, Comparator>`

- La clase `Comparator` debe sobrecargar el operador (`()`).
 - La sobrecarga recibe dos parámetros.
 - Devuelve `true` si el primero es estrictamente menor que el segundo.

Objetos función

```
struct OrdenLexicografico {  
    bool operator()(const Coords &p1, const Coords &p2) const {  
        return p1.x < p2.x || p1.x == p2.x && p1.y < p2.y;  
    }  
};
```

```
MapTree<Coords, string, OrdenLexicografico> dicc;  
dicc.insert({{3, 3}, "Laura"});  
dicc.insert({{5, -3}, "Gloria"});  
dicc.insert({{1, 2}, "Belén"});  
dicc.insert({{-1, -5}, "Samuel"});
```



¿Y si quiero utilizar <?

- Utilizar solamente dos parámetros: tipo de las claves y valores.

```
MapTree<Coords, string> dicc;  
dicc.insert({{3, 3}, "Laura"});  
dicc.insert({{5, -3}, "Gloria"});  
dicc.insert({{1, 2}, "Belén"});  
dicc.insert({{-1, -5}, "Samuel"});
```



Implementación

```
template <typename K, typename V, typename ComparatorFunction = std::less<K>>
class MapTree {
    ...
private:
    struct Node { ... };

    Node *root_node;
    int num_elems;
    ComparatorFunction less_than;

    ...
}
```



Implementación: antes

```
template <typename K, typename V, typename ComparatorFunction = std::less<K>>
class MapTree {
    ...
private:
    ...
    ComparatorFunction less_than;

    static Node * search(Node *root, const K &key) {
        if (root == nullptr) {
            return nullptr;
        } else if (key < root->entry.key) {
            return search(root->left, key);
        } else if (root->entry.key < key) {
            return search(root->right, key);
        } else {
            return root;
        }
    }
};
```



Implementación: después

```
template <typename K, typename V, typename ComparatorFunction = std::less<K>>
class MapTree {
    ...
private:
    ...
    ComparatorFunction less_than;

    static Node * search(Node *root, const K &key) {
        if (root == nullptr) {
            return nullptr;
        } else if (less_than(key, root->entry.key)) {
            return search(root->left, key);
        } else if (less_than(root->entry.key, key)) {
            return search(root->right, key);
        } else {
            return root;
        }
    }
};
```

ESTRUCTURAS DE DATOS

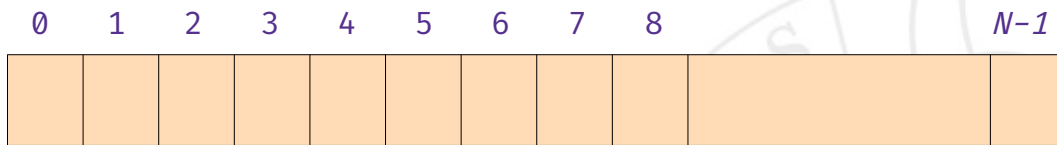
DICCIONARIOS

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:

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

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 $p = 131$)

¿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 $K = \text{int}$.

```
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 $K = \text{string}$.

```
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;
```

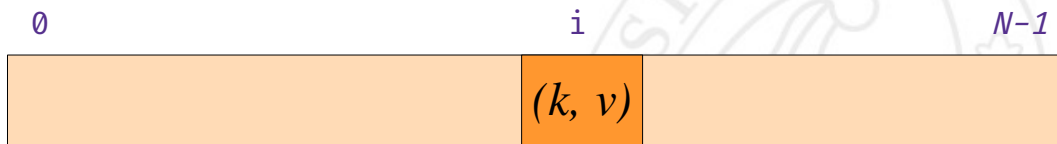
```
std::cout << h_int(24) << std::endl;  
std::cout << h_str("Pepe") << std::endl;  
std::cout << h_str("Maria") << std::endl;
```

```
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.

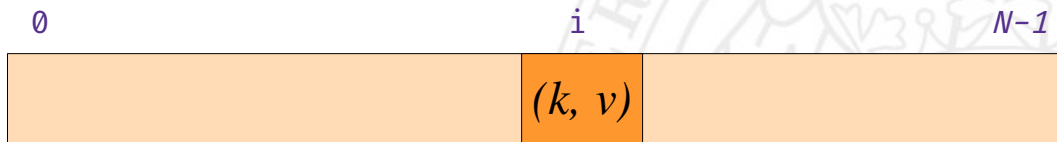
- 1) Calculamos $i := h(k) \bmod N$.
- 2) Insertamos el par (k, v) en la posición i -ésima del vector.



¿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 $h(35) \bmod 13 = 35 \bmod 13 = 9$.

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

Ejemplo

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

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

Ejemplo

- 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

Ejemplo

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

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

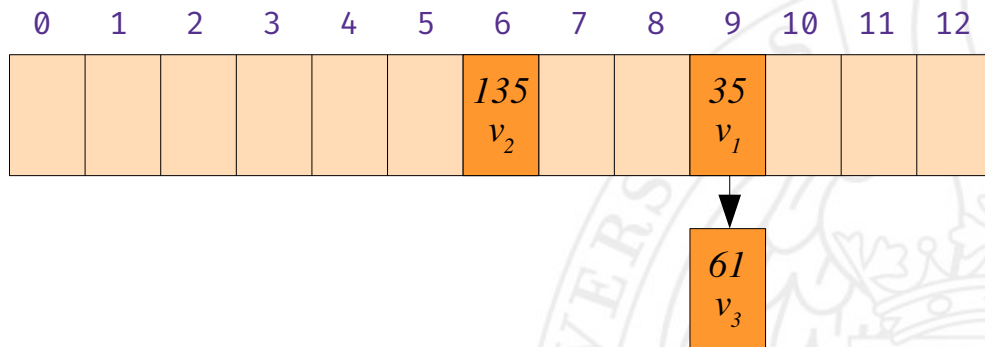
↑
*¡Posición
ocupada!*

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.



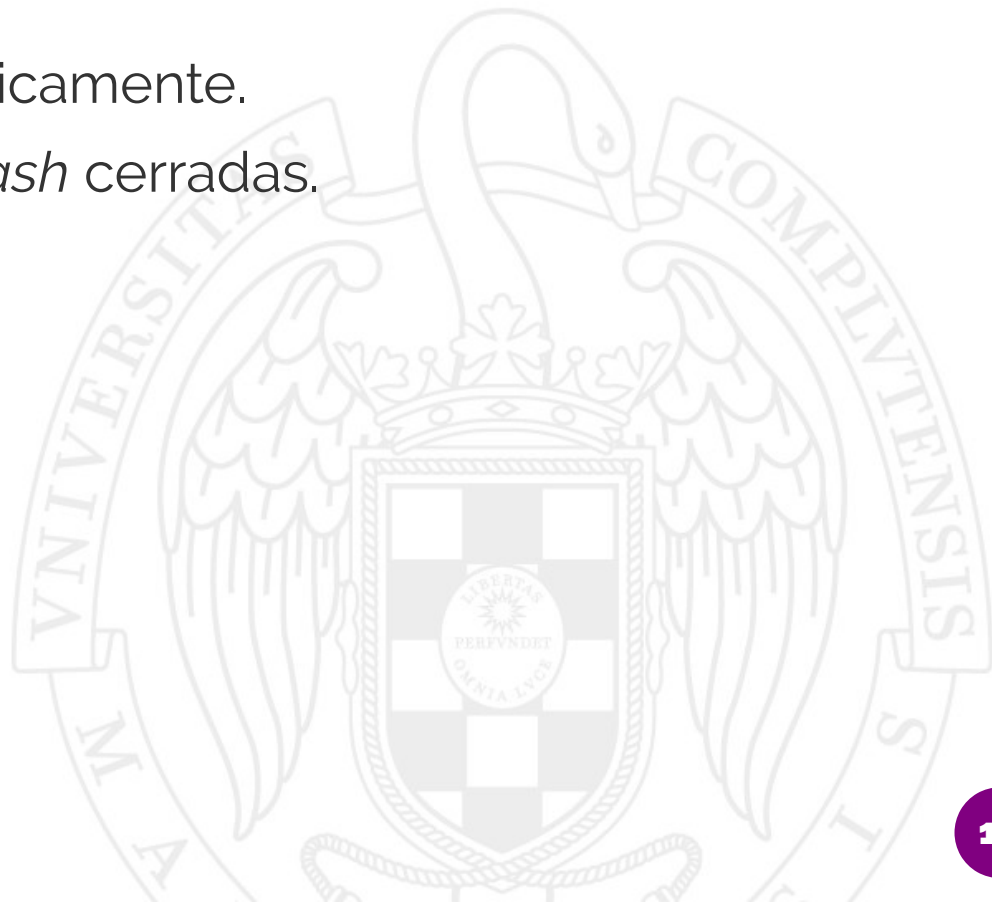
¿Cómo solucionamos las colisiones?

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

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

Implementaciones

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



ESTRUCTURAS DE DATOS

DICCIONARIOS

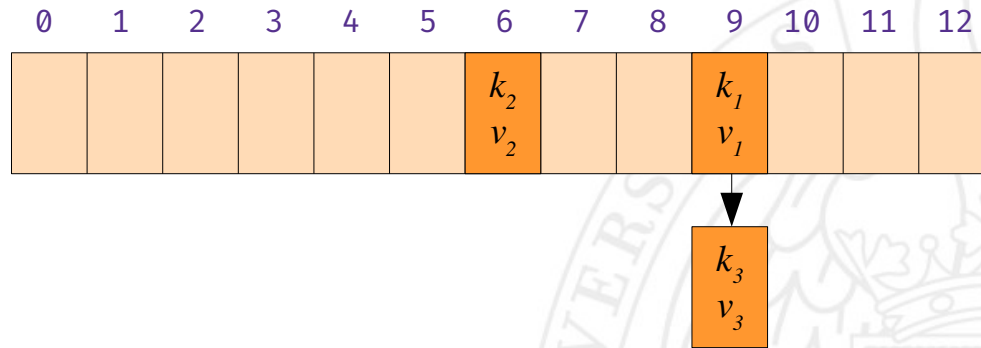
Tablas *hash* abiertas

Manuel Montenegro Montes

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

Objetivo

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



Recordatorio: TAD Diccionario

- Constructoras:
 - Crear un diccionario vacío: ***create_empty***
- Mutadoras:
 - Añadir una entrada al diccionario: ***insert***
 - Eliminar una entrada del diccionario: ***erase***
- Observadoras:
 - Saber si existe una entrada con una clave determinada: ***contains***
 - Saber el valor asociado con una clave: ***at***
 - Saber si el diccionario está vacío: ***empty***
 - Saber el número de entradas del diccionario: ***size***

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);
};
```

Clase MapHash: representación privada

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

```
private:
```

```
    using List = std::forward_list<MapEntry>;
```

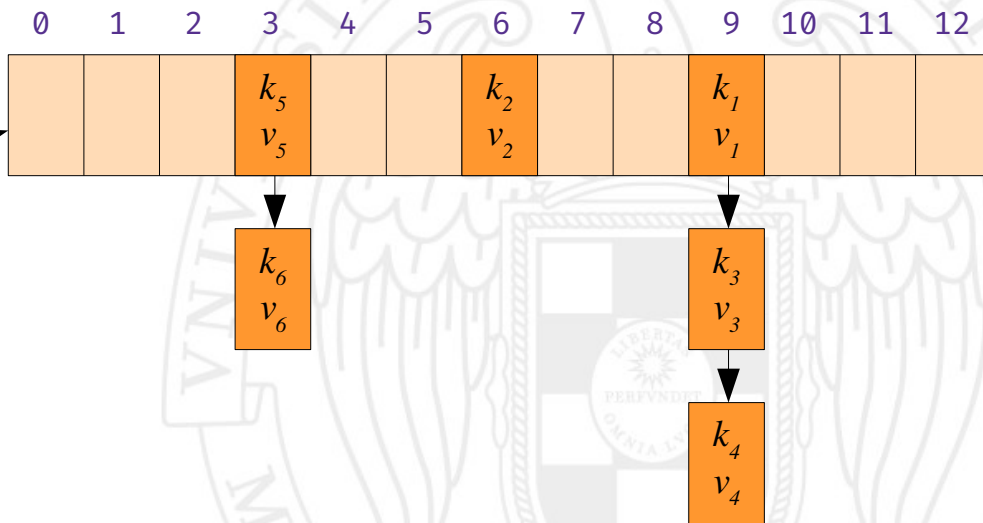
```
    List *buckets;
```

```
    int num_elems;
```

```
    Hash hash;
```

```
};
```

buckets: •
num_elems: 6
hash: ...



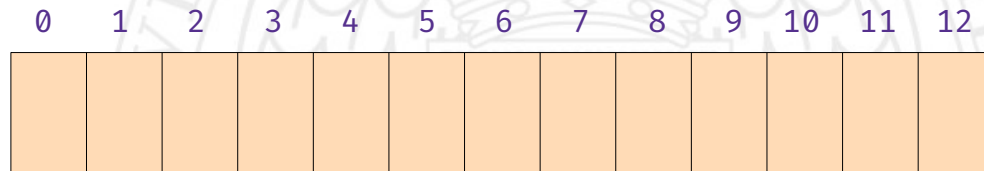
Clase MapHash: constructores

```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
public:
    MapHash(): num_elems(0), buckets(new List[CAPACITY]) { };

    MapHash(const MapHash &other): num_elems(other.num_elems),
                                   hash(other.hash),
                                   buckets(new List[CAPACITY]) {
        std::copy(other.buckets, other.buckets + CAPACITY, buckets);
    };

    ~MapHash() {
        delete[] buckets;
    }

private:
    List *buckets;
    int num_elems;
    Hash hash;
    // ...
};
```



Clase MapHash: búsqueda

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

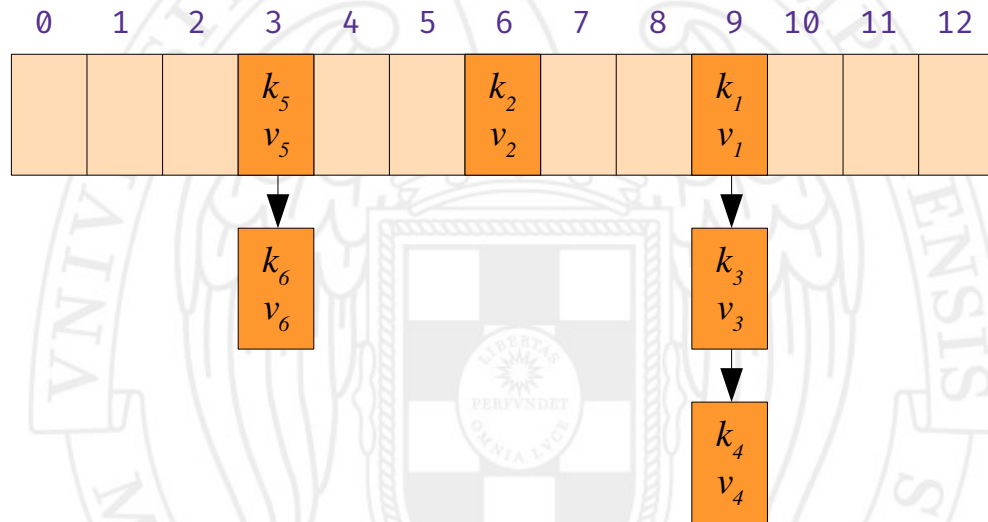
```
class MapHash {
```

```
public:
```

```
    const V & at(const K &key) const {  
        int h = hash(key) % CAPACITY;  
        const List &list = buckets[h];  
  
        auto it = find_in_list(list, key);  
  
        assert (it != list.end());  
        return it->value;  
    }
```

```
private:
```

```
    List *buckets;  
    int num_elems;  
    Hash hash;  
    // ...  
};
```



Clase MapHash: búsqueda

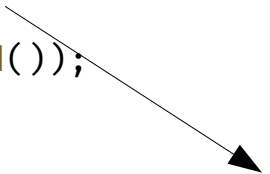
```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
public:
```

```
    const V & at(const K &key) const {
        int h = hash(key) % CAPACITY;
        const List &list = buckets[h];

        auto it = find_in_list(list, key);

        assert (it != list.end());
        return it->value;
    }
```

```
private:
    List *buckets;
    int num_elems;
    Hash hash;
    // ...
};
```



```
List::const_iterator find_in_list(const List &list, const K &key) {
    auto it = list.begin();
    while (it != list.end() && it->key != key) {
        ++it;
    }
    return it;
}
```

Clase MapHash: inserción

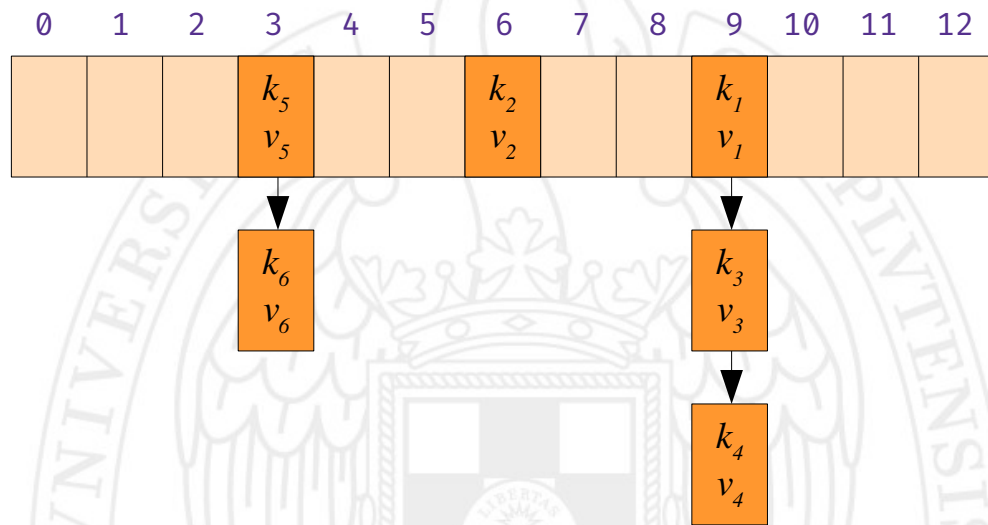
```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
public:
```

```
    void insert(const MapEntry &entry) {
        int h = hash(entry.key) % CAPACITY;
        List &list = buckets[h];

        auto it = find_in_list(list, entry.key);

        if (it == list.end()) {
            list.push_front(entry);
            num_elems++;
        }
    }
};
```

```
private:
    List *buckets;
    int num_elems;
    Hash hash;
    // ...
};
```



Clase MapHash: inserción

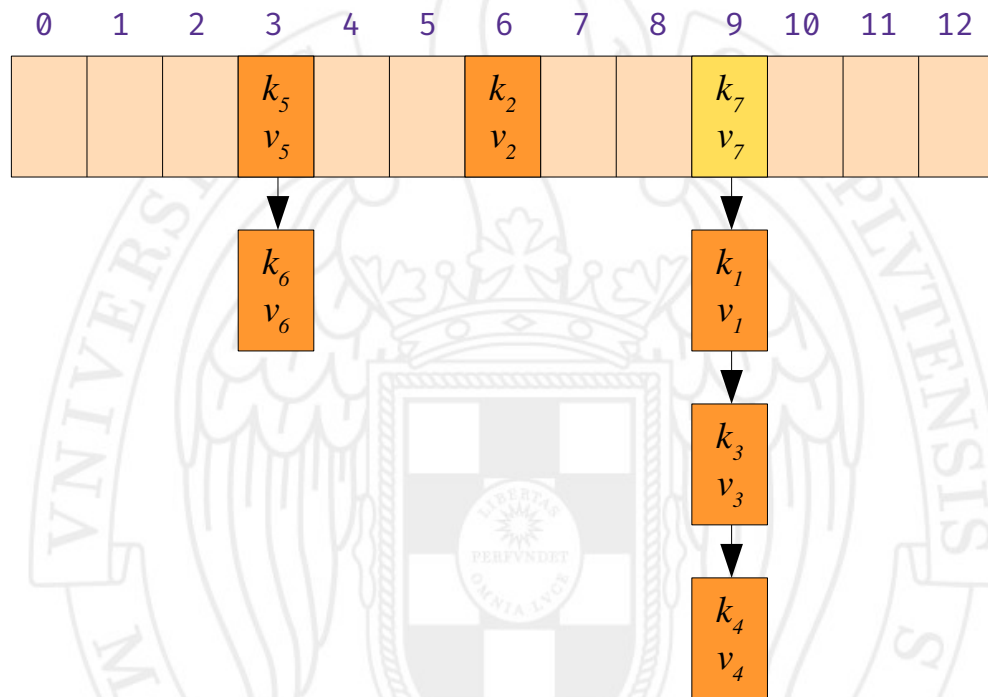
```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
public:
```

```
    void insert(const MapEntry &entry) {
        int h = hash(entry.key) % CAPACITY;
        List &list = buckets[h];

        auto it = find_in_list(list, entry.key);

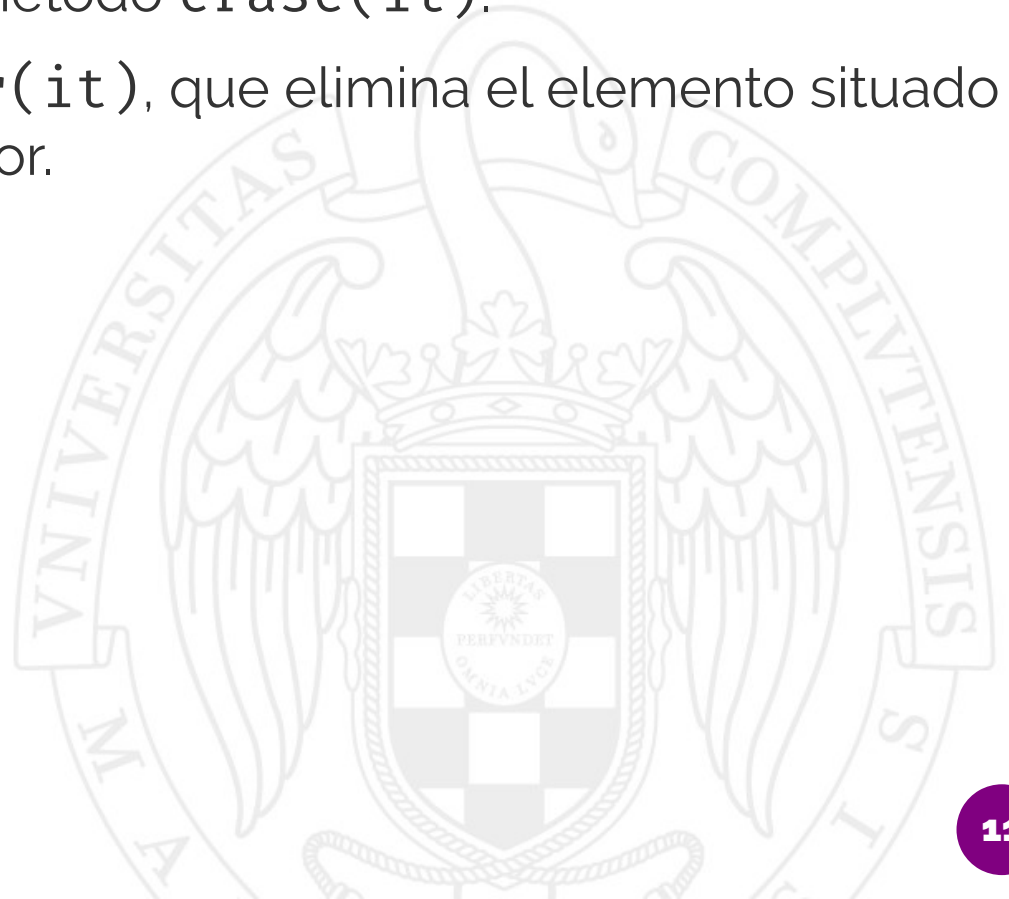
        if (it == list.end()) {
            list.push_front(entry);
            num_elems++;
        }
    }
};
```

```
private:
    List *buckets;
    int num_elems;
    Hash hash;
    // ...
};
```



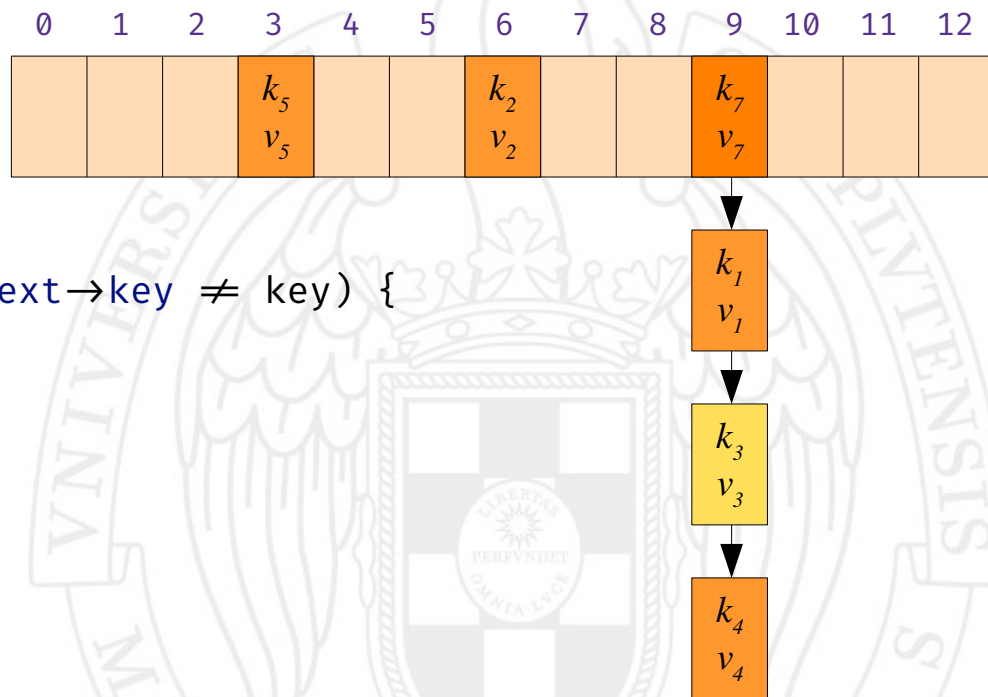
Clase MapHash: borrado

- Similar a la inserción.
- La clase `forward_list` no tiene método `erase(it)`.
- Pero sí tiene método `erase_after(it)`, que elimina el elemento situado después del apuntado por el iterador.



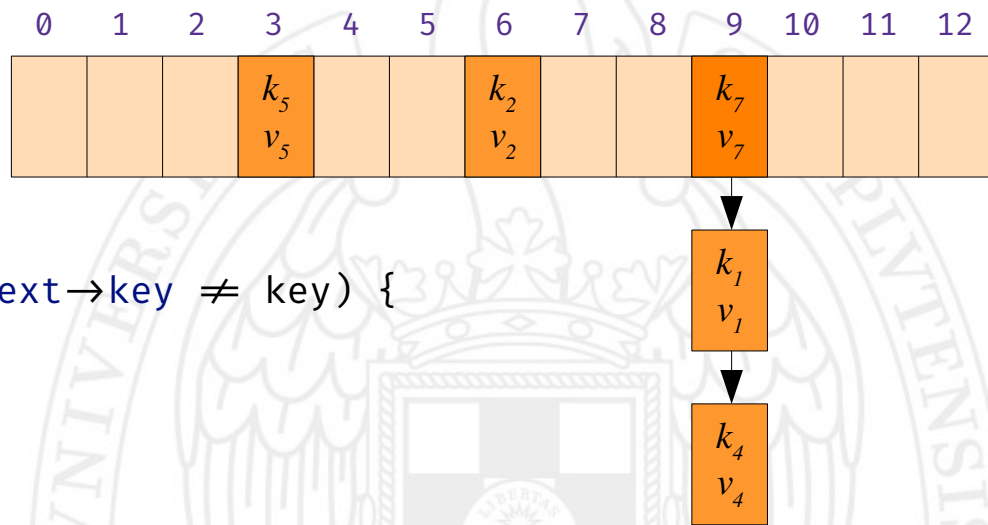
Clase MapHash: borrado

```
void erase(const K &key) {  
    int h = hash(key) % CAPACITY;  
    List &list = buckets[h];  
    if (!list.empty()) {  
        if (list.front().key == key) {  
            list.pop_front();  
            num_elems--;  
        } else {  
            auto it_prev = list.begin();  
            auto it_next = ++list.begin();  
  
            while (it_next != list.end() && it_next->key != key) {  
                it_prev++;  
                it_next++;  
            }  
            if (it_next != list.end()) {  
                list.erase_after(it_prev);  
                num_elems--;  
            }  
        }  
    }  
}
```



Clase MapHash: borrado

```
void erase(const K &key) {  
    int h = hash(key) % CAPACITY;  
    List &list = buckets[h];  
    if (!list.empty()) {  
        if (list.front().key == key) {  
            list.pop_front();  
            num_elems--;  
        } else {  
            auto it_prev = list.begin();  
            auto it_next = ++list.begin();  
  
            while (it_next != list.end() && it_next->key != key) {  
                it_prev++;  
                it_next++;  
            }  
            if (it_next != list.end()) {  
                list.erase_after(it_prev);  
                num_elems--;  
            }  
        }  
    }  
}
```



ESTRUCTURAS DE DATOS

DICCIONARIOS

Tablas *hash* cerradas

Manuel Montenegro Montes

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

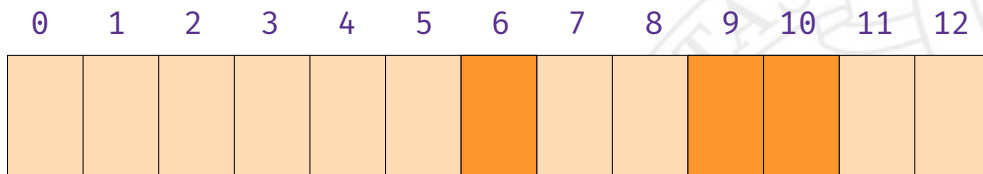
Objetivo

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

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

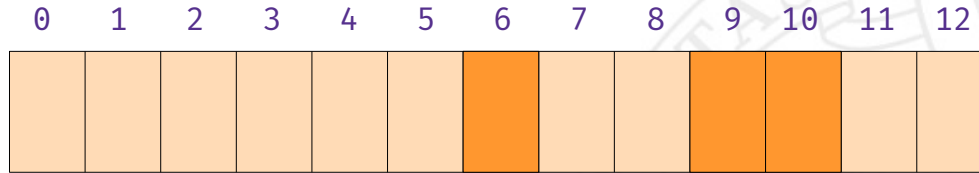
Idea de implementación

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



Inserción

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



Ejemplo

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

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

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.

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

- Ejemplo: buscamos 23 y 63.

¡Cuidado con el borrado!

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

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

- ¿Y si ahora buscamos 23?

Solución

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

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);
```

```
};
```

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;
```

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

```
    Bucket *buckets;
```

```
    Hash hash;
```

```
    int num_elems;
```

```
};
```

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)
```

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;

    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};
}
```


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};  
}
```

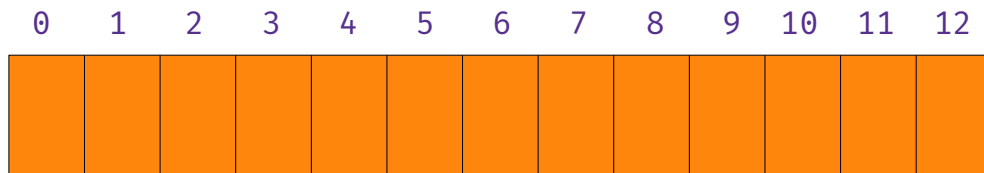
Pasamos a la posición siguiente. Si llegamos al final del vector, volvemos a la posición 0.

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?



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

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};
}
```

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:
    // ...
};
```



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:
    // ...
};
```



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



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:

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

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

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

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:

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

Sondeo lineal

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

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

- 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**.

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

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

10 12 1 3 5 7 9 11 ...

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 ...

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.

Sondeo cuadrático

- Utiliza la siguiente fórmula:

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

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

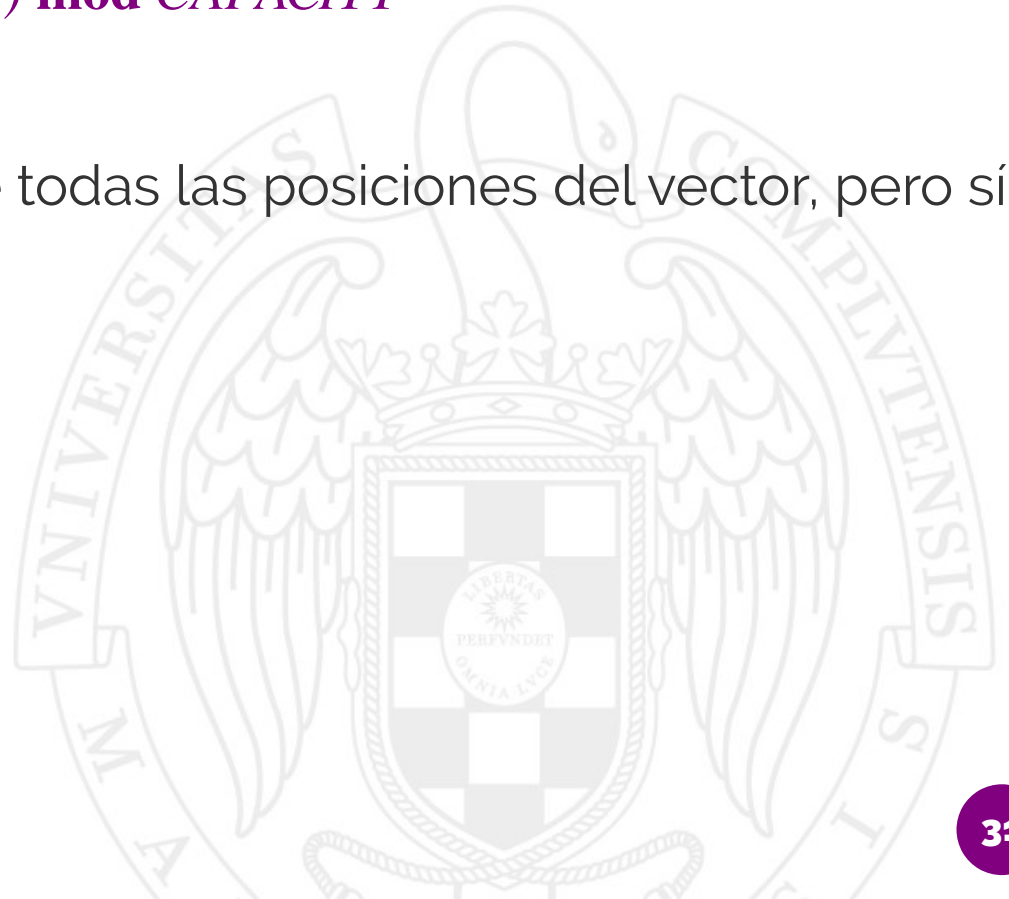
10 11 1 6 0 9 7 7 ...

Sondeo cuadrático

- Utiliza la siguiente fórmula:

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

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

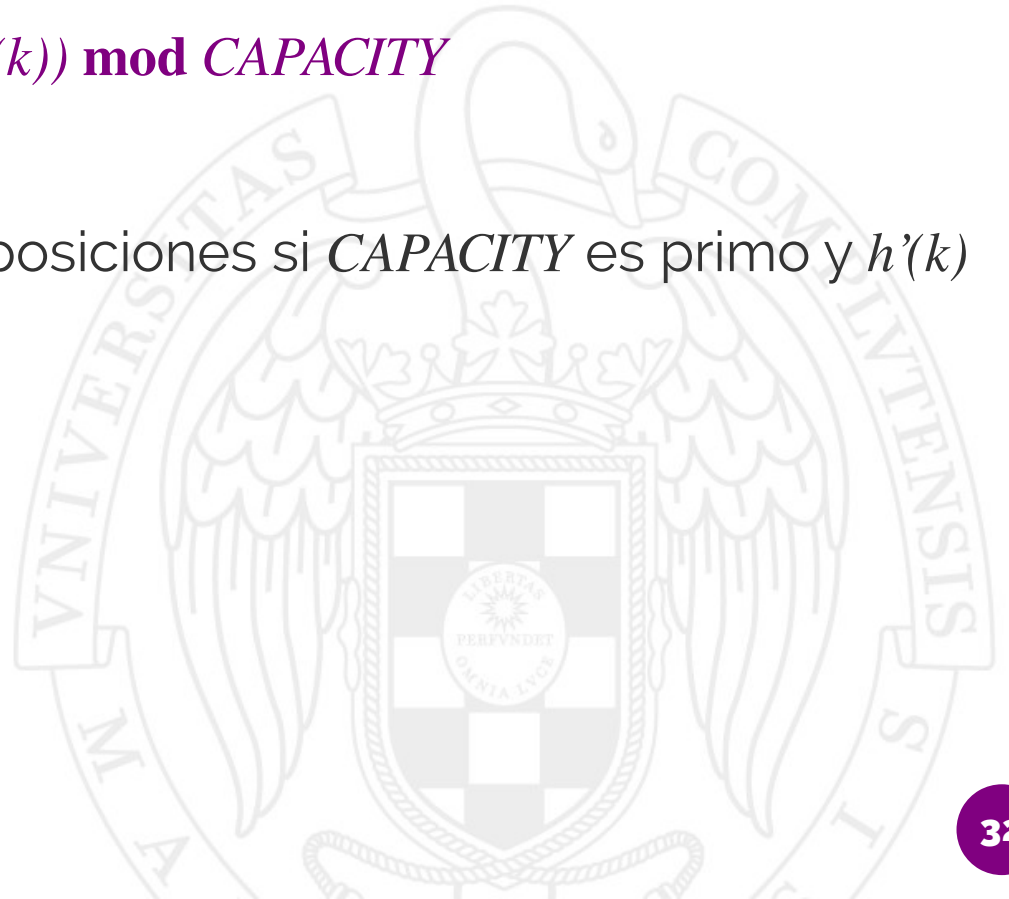


Doble redispersión

- 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 *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/Quadratic_probing
- https://en.wikipedia.org/wiki/Double_hashing



ESTRUCTURAS DE DATOS

DICCIONARIOS

Análisis de coste en tablas *hash*

Manuel Montenegro Montes

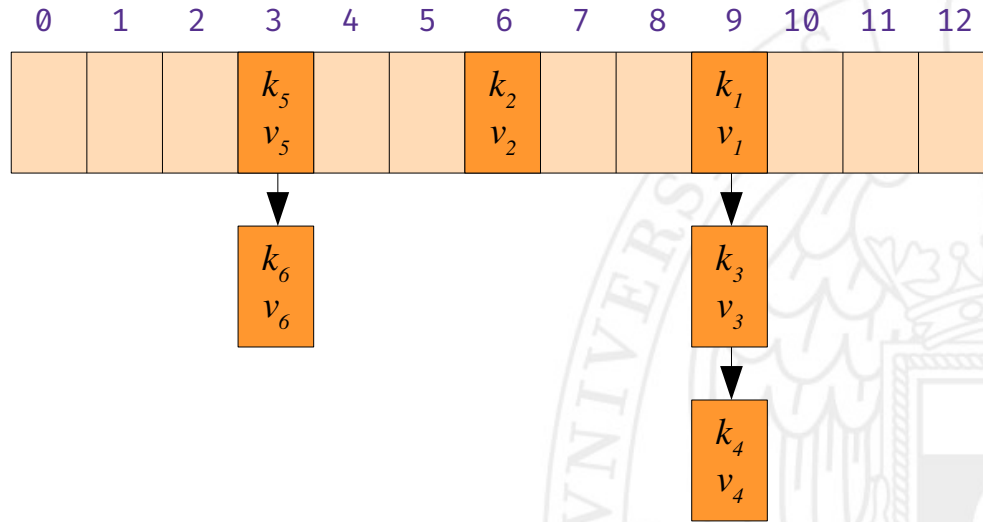
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

Tablas *hash* abiertas



Recordatorio

- Una tabla *hash* abierta asocia cada cajón con una **lista de entradas**.
- En caso de colisión entre claves, las entradas acaban en la misma lista.



Factor de carga

- El **factor de carga** α de una tabla *hash* es el cociente entre el número de entradas en la tabla y el número de cajones.

- Sean:

n - número de entradas en la tabla

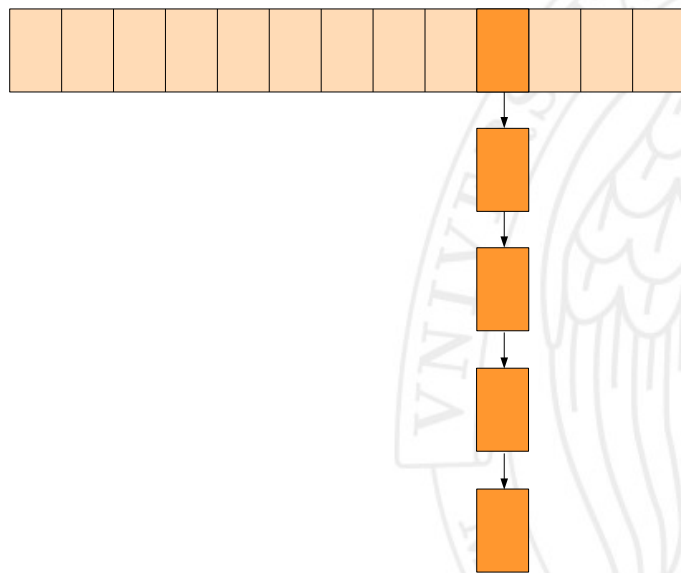
m - número de cajones

$$\alpha = \frac{n}{m}$$

- Expresaremos el coste de los algoritmos en función del factor de carga.

Dispersión uniforme

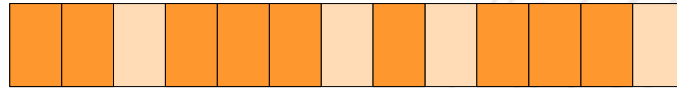
- La eficiencia de una tabla *hash* viene determinada por las propiedades de la función *hash* utilizada.
- Una función *hash* nefasta enviaría todas las claves al mismo cajón.



Dispersión uniforme

Suposición de dispersión uniforme:

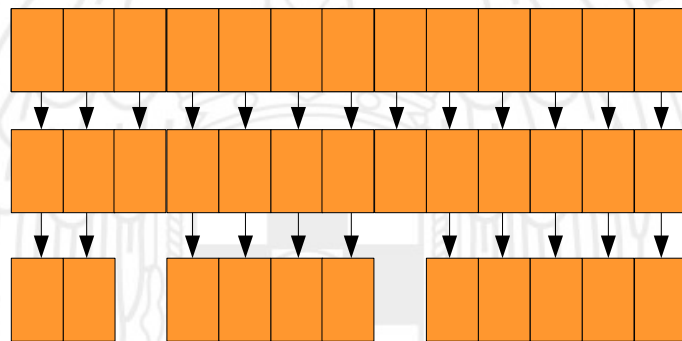
La función *hash* distribuye uniformemente todas las claves a lo largo de los cajones de la tabla.



Longitud media de las listas

- Supongamos que tenemos n elementos y m cajones, y que la propiedad de distribución uniforme se cumple.
- Sea N_i la longitud de la lista del cajón i -ésimo.
- ¿Cuál es el valor promedio de N_i ?

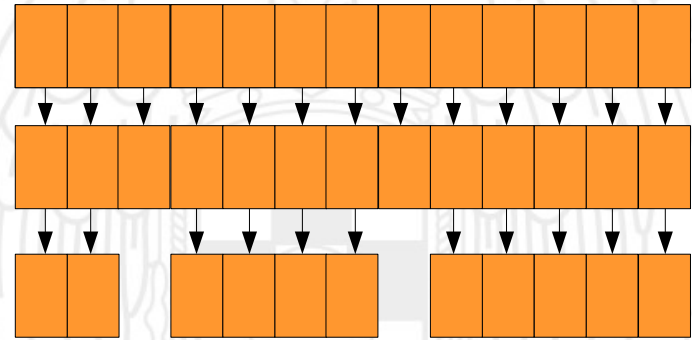
$$E[N_i] = \frac{n}{m} = \alpha$$



Búsqueda de una clave (fallo)

- Supongamos que realizamos una búsqueda de una clave que no se encuentra en la tabla.
- El coste debe recorrer una de las listas en su totalidad.
- Por tanto, el coste medio es proporcional a α .
- Similarmente para la inserción y borrado.

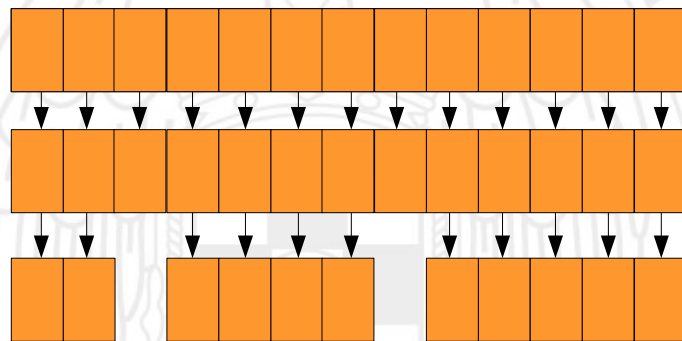
$$O(1 + \alpha)$$



Búsqueda de una clave (éxito)

- Supongamos que realizamos una búsqueda de una clave que sí se encuentra en la tabla.
- El número medio de elementos recorridos es $1 + \frac{\alpha}{2} - \frac{\alpha}{2n}$
- Similarmente para la inserción y borrado.

$$1 + \frac{\alpha}{2} - \frac{\alpha}{2n} \in O(1 + \alpha)$$



Conclusión

- El coste de todas las operaciones está acotado por α .
- **Si conseguimos mantener α acotado, el coste de las operaciones será constante.**
- ¿Cómo conseguimos mantener α acotado?

Haciendo que el número de cajones aumente proporcionalmente con el número de entradas → *Tabla dinámicamente redimensionable*.

Tablas *hash* cerradas



Recordatorio

- Una tabla *hash* cerrada contiene, en cada cajón, una única entrada.
- En caso de colisión entre claves, las entradas acaban en cajones distintos según la estrategia de redistribución utilizada.

0	1	2	3	4	5	6	7	8	9	10	11	12
			k_5 v_5	k_6 v_6		k_2 v_2			k_1 v_1	k_3 v_3	k_4 v_4	

Factor de carga

- Sean:

n - número de entradas en la tabla

m - número de cajones

$$\alpha = \frac{n}{m}$$

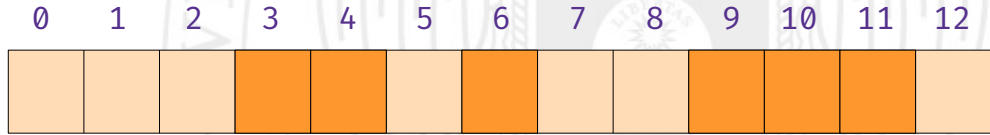
- En una *tabla hash* cerrada, siempre se cumple que $\alpha \leq 1$.



Búsqueda de una clave (fallo)

- Sea N una variable aleatoria que denota el número de intentos infructuosos hasta que llegamos a un cajón vacío.

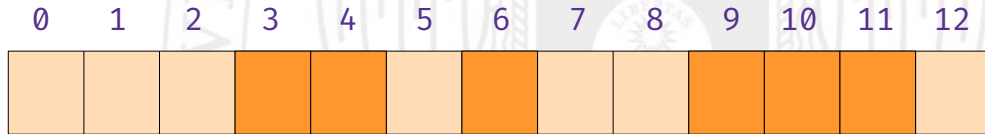
$$P\{N \geq 1\} = \frac{n}{m}$$



Búsqueda de una clave (fallo)

- Sea N una variable aleatoria que denota el número de intentos infructuosos hasta que llegamos a un cajón vacío.

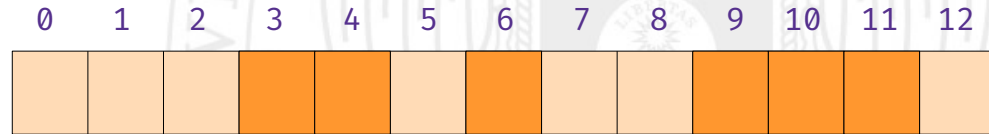
$$P\{N \geq 2\} = \frac{n}{m} * \frac{n-1}{m-1}$$



Búsqueda de una clave (fallo)

- Sea N una variable aleatoria que denota el número de intentos infructuosos hasta que llegamos a un cajón vacío.

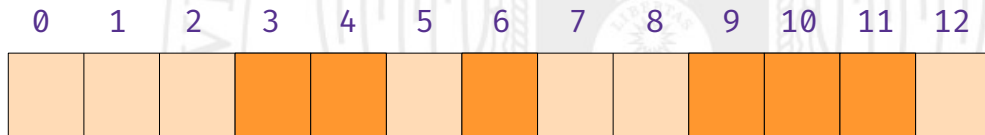
$$P\{N \geq 3\} = \frac{n}{m} * \frac{n-1}{m-1} * \frac{n-2}{m-2}$$



Búsqueda de una clave (fallo)

- Sea N una variable aleatoria que denota el número de intentos infructuosos hasta que llegamos a un cajón vacío.

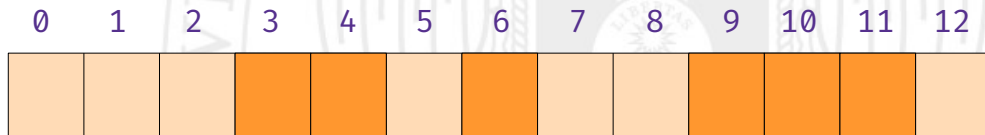
$$P\{N \geq i\} = \frac{n}{m} * \frac{n-1}{m-1} * \frac{n-2}{m-2} * \dots * \frac{n-i+1}{m-i+1}$$



Búsqueda de una clave (fallo)

- Sea N una variable aleatoria que denota el número de intentos infructuosos hasta que llegamos a un cajón vacío.

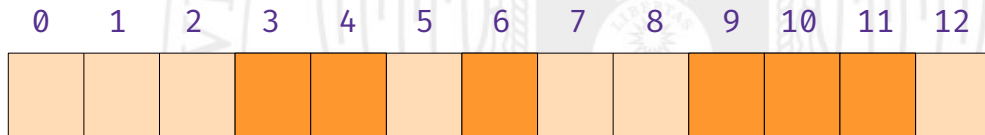
$$P\{N \geq i\} \leq \alpha^i$$



Búsqueda de una clave (fallo)

- Promedio de cajones visitados para buscar una clave:

$$1 + E[N] = 1 + \sum_{i=1}^{\infty} P\{N \geq i\} \leq 1 + \sum_{i=1}^{\infty} \alpha^i = \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1 - \alpha}$$



Búsqueda de una clave (fallo)

- Promedio de cajones visitados para buscar una clave:

$$1 + E[N] = 1 + \sum_{i=1}^{\infty} P\{N \geq i\} \leq 1 + \sum_{i=1}^{\infty} \alpha^i = \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1 - \alpha}$$

- Si $\alpha = 0.9$, entonces se visitan 10 cajones en el caso medio.
- Si $\alpha = 0.8$, entonces se visitan 5 cajones en el caso medio.
- Si $\alpha = 0.5$, entonces se visitan 2 cajones en el caso medio.

Búsqueda de una clave (éxito)

- Promedio de cajones visitados para buscar una clave:

$$\frac{1}{\alpha} * \ln \frac{1}{1 - \alpha}$$

- Si $\alpha = 0.9$, entonces se visitan 2.56 cajones en el caso medio.
- Si $\alpha = 0.8$, entonces se visitan 2 cajones en el caso medio.
- Si $\alpha = 0.5$, entonces se visitan 1.38 cajones en el caso medio.

Conclusión

- Si conseguimos mantener α inferior a 1, el coste de las operaciones será constante.
- Con $\alpha \leq 0.8$ se obtienen constantes razonables.
- ¿Cómo conseguimos mantener α constante?
 - Haciendo que el número de cajones aumente proporcionalmente con el número de entradas → *Tabla dinámicamente redimensionable*.

Bibliografía

- T. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein
Introduction to Algorithms (3ª edición)
The MIT Press (2009)
Capítulo 11
- R. Peña
Diseño de Programas. Formalismo y Abstracción (3ª edición)
Pearson Educación (2005)
Sección 8.1.3



ESTRUCTURAS DE DATOS

DICCIONARIOS

Tablas *hash* redimensionables

Manuel Montenegro Montes

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

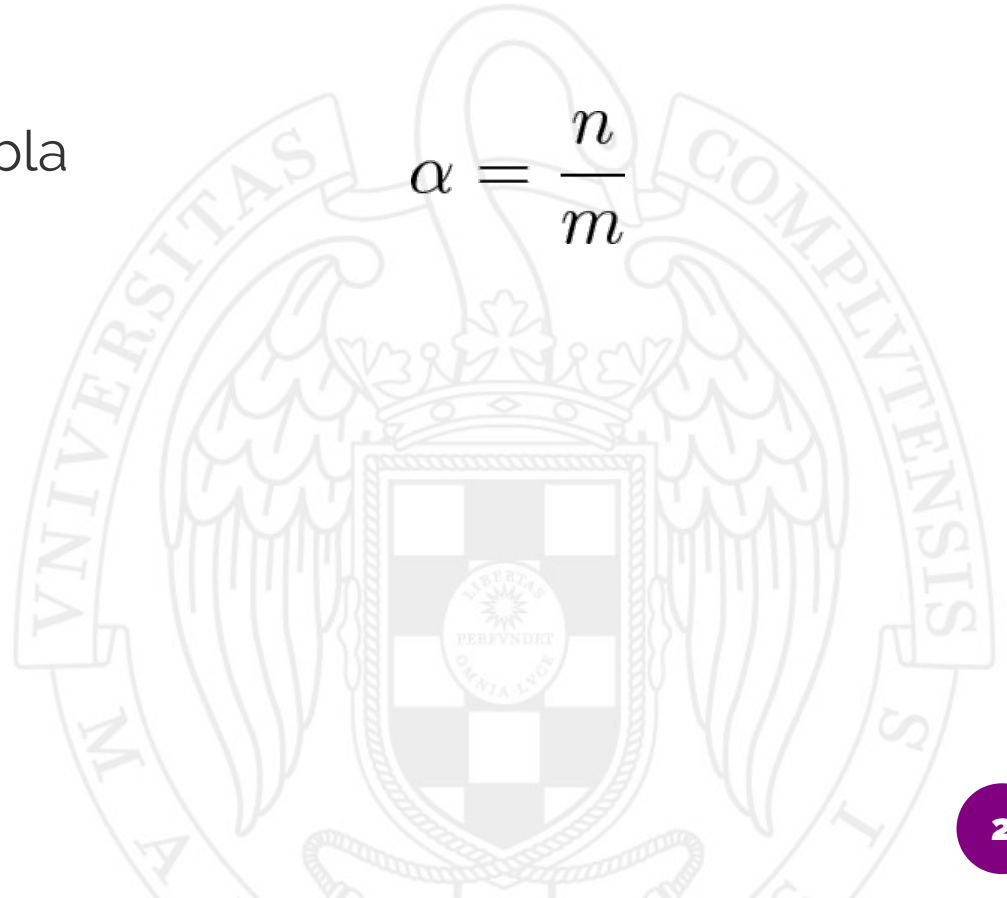
Recordatorio: factor de carga

- El **factor de carga** α de una tabla *hash* es el cociente entre el número de entradas en la tabla y el número de cajones.
- Sean:

n - número de entradas en la tabla

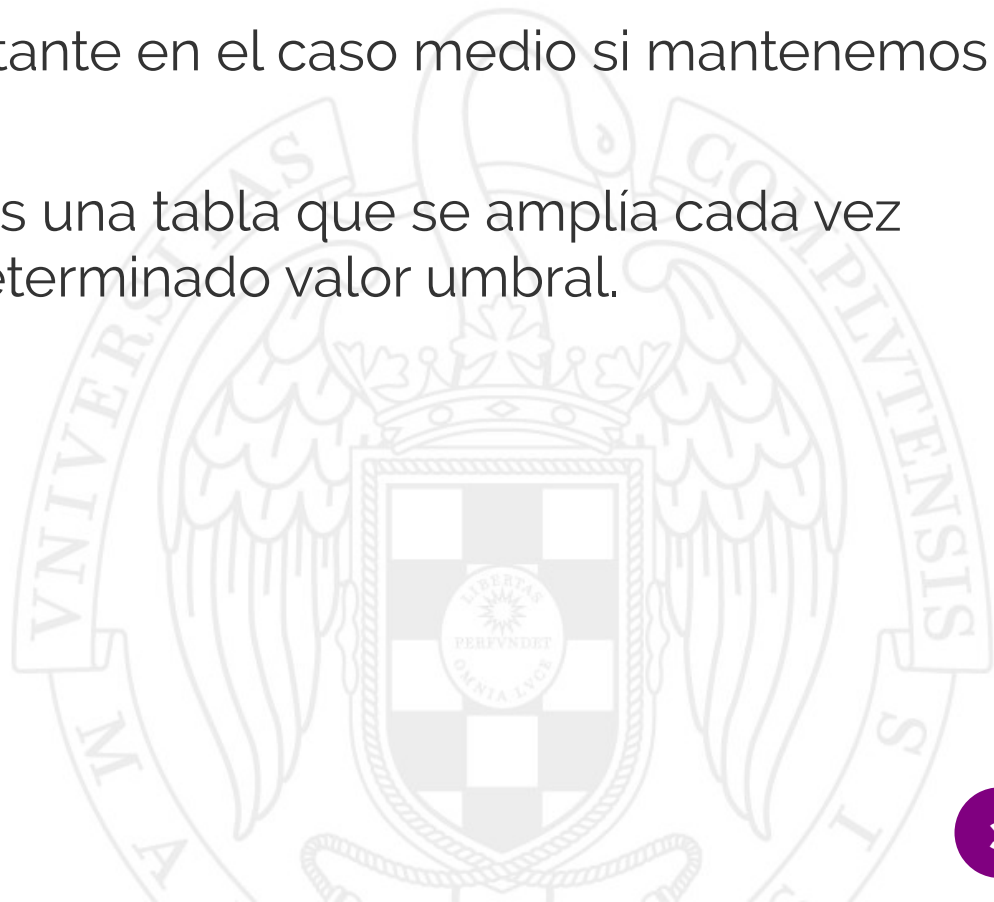
m - número de cajones

$$\alpha = \frac{n}{m}$$



Tablas redimensionables

- En el caso medio, las operaciones en una tabla hash abierta tienen coste $O(1 + \alpha)$.
- Por tanto, conseguimos coste constante en el caso medio si mantenemos el factor de carga acotado.
- Una **tabla hash redimensionable** es una tabla que se amplía cada vez que el factor de carga supera un determinado valor umbral.



Implementación

```
const int INITIAL_CAPACITY = 31;  
const double MAX_LOAD_FACTOR = 0.8;
```

Factor de carga máximo permitido

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

```
private:  
    using List = std::forward_list<MapEntry>;
```

```
    Hash hash;
```

```
    List *buckets;
```

```
    int num_elems;
```

```
    int capacity;
```

Tamaño del vector

```
    ...  
};
```


Implementación de insert (antes)

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

    void insert(const MapEntry &entry) {
        int h = hash(entry.key) % capacity;

        auto it = find_in_list(buckets[h], entry.key);

        if (it == buckets[h].end()) {
            buckets[h].push_front(entry);
            num_elems++;
        }
    }

private:
    ...
};
```



Implementación de insert (después)

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

```
    ...
```

```
    void insert(const MapEntry &entry) {
        int h = hash(entry.key) % capacity;

        auto it = find_in_list(buckets[h], entry.key);

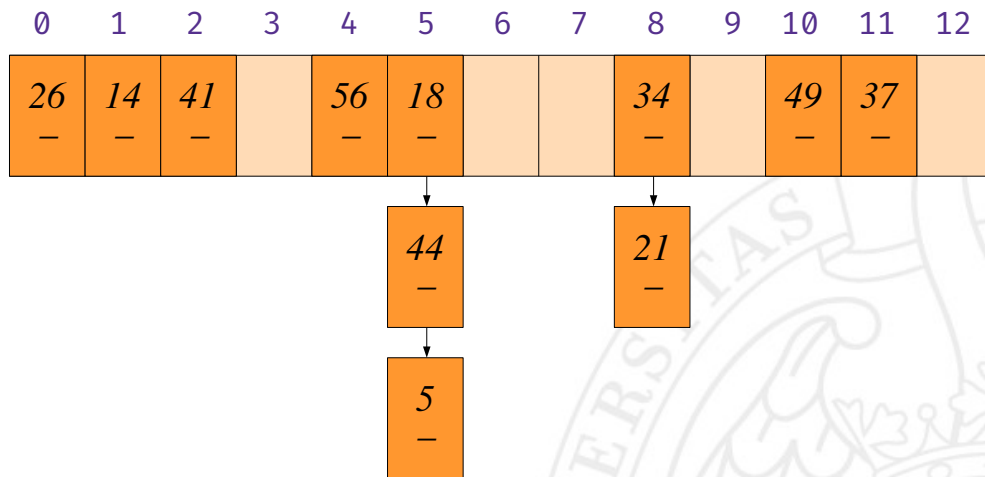
        if (it == buckets[h].end()) {
            num_elems++;
            resize_if_necessary();
            h = hash(entry.key) % capacity;
            buckets[h].push_front(entry);
        }
    }
}
```

```
private:
```

```
    ...
};
```



Ejemplo de redimensionamiento



Ejemplo de redimensionamiento

0	1	2	3	4	5	6	7	8	9	10	11	12
26	14	41		56	18			34		49	37	
-	-	-		-	-			-		-	-	

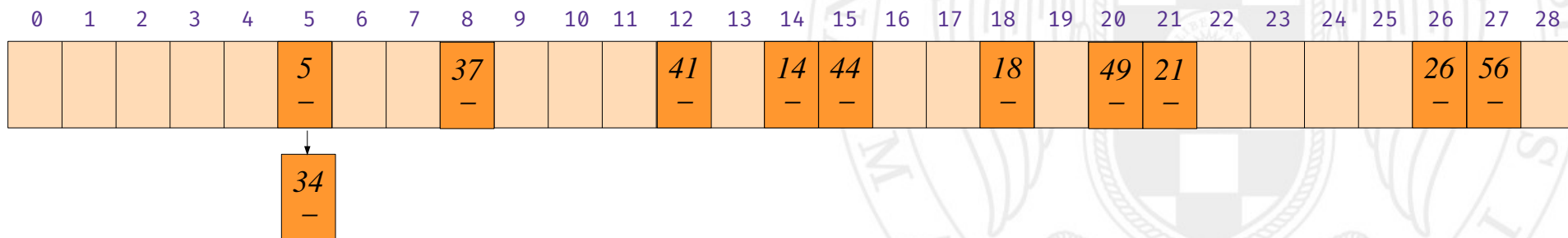
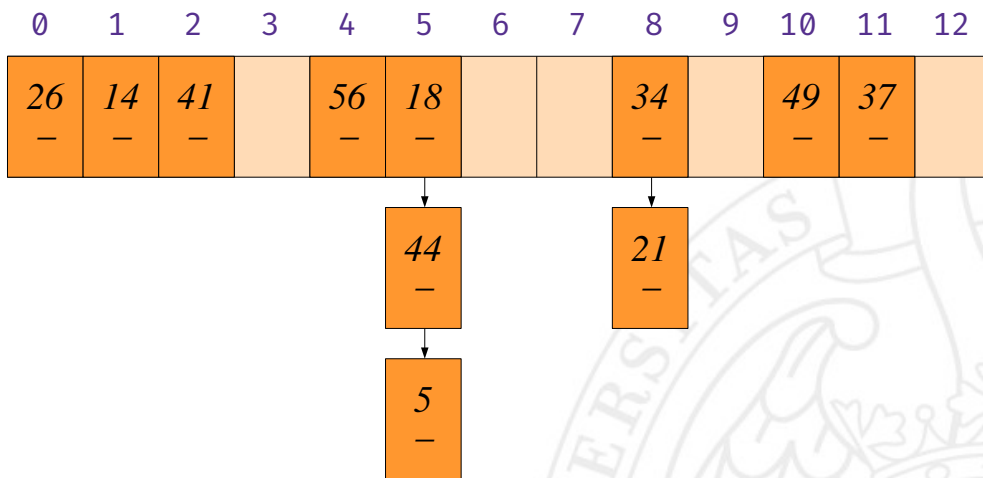
44
-

21
-

5
-

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28

Ejemplo de redimensionamiento



Método auxiliar de redimensionamiento

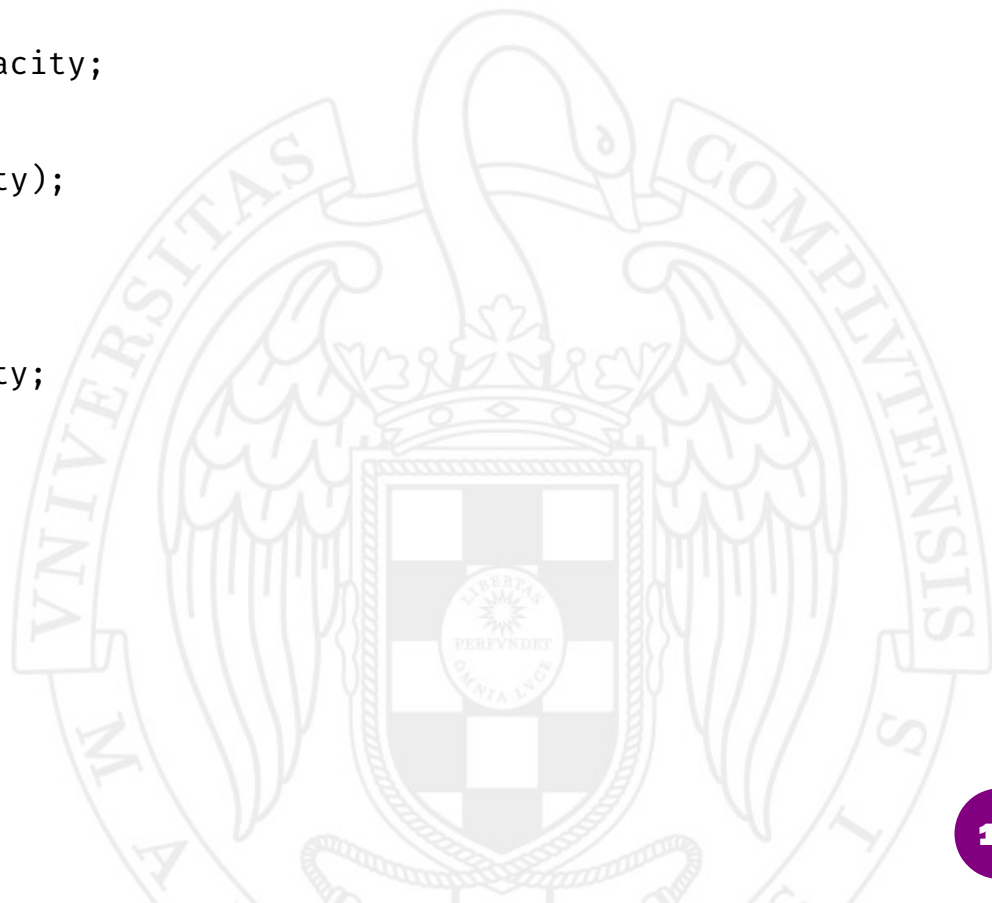
```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
private:

    void resize_if_necessary() {
        double load_factor = ((double)num_elems) / capacity;
        if (load_factor < MAX_LOAD_FACTOR) return;

        int new_capacity = next_prime_after(2 * capacity);
        List *new_array = new List[new_capacity];

        for (int i = 0; i < capacity; i++) {
            for (const MapEntry &entry : buckets[i]) {
                int new_pos = hash(entry.key) % new_capacity;
                new_array[new_pos].push_front(entry);
            }
        }

        capacity = new_capacity;
        delete[] buckets;
        buckets = new_array;
    }
};
```



Costes en tiempo

- Suponiendo **dispersión uniforme**

Operación	Tabla <i>hash</i>
<i>constructor</i>	$O(1)$
<i>empty</i>	$O(1)$
<i>size</i>	$O(1)$
<i>contains</i>	$O(1)$
<i>at</i>	$O(1)$
<i>operator[]</i>	$O(1) / O(n)$
<i>insert</i>	$O(1) / O(n)$
<i>erase</i>	$O(1)$

n = número de entradas en la tabla

Costes amortizados en tiempo

- Suponiendo **dispersión uniforme**.

Operación	Tabla <i>hash</i>
<i>constructor</i>	$O(1)$
<i>empty</i>	$O(1)$
<i>size</i>	$O(1)$
<i>contains</i>	$O(1)$
<i>at</i>	$O(1)$
<i>operator[]</i>	$O(1)$
<i>insert</i>	$O(1)$
<i>erase</i>	$O(1)$

n = número de entradas en la tabla