

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

Ideas muy parecidas a cuando trabajábamos con conjuntos y no lo va a repetir aquí todo.

- Constructoras:

- Crear un diccionario vacío: **create\_empty**

- Mutadoras:

- Añadir una entrada al diccionario: **insert**
- Eliminar una entrada del diccionario: **erase**

Entrada = Clave (K) + Valor (V).

Árboles ordenados que en sus componentes tiene diccionarios.

Lo inserta de la misma forma que vimos en el tema 8, pero hay que tener cuidado con cual es la clave.

- Observadoras:

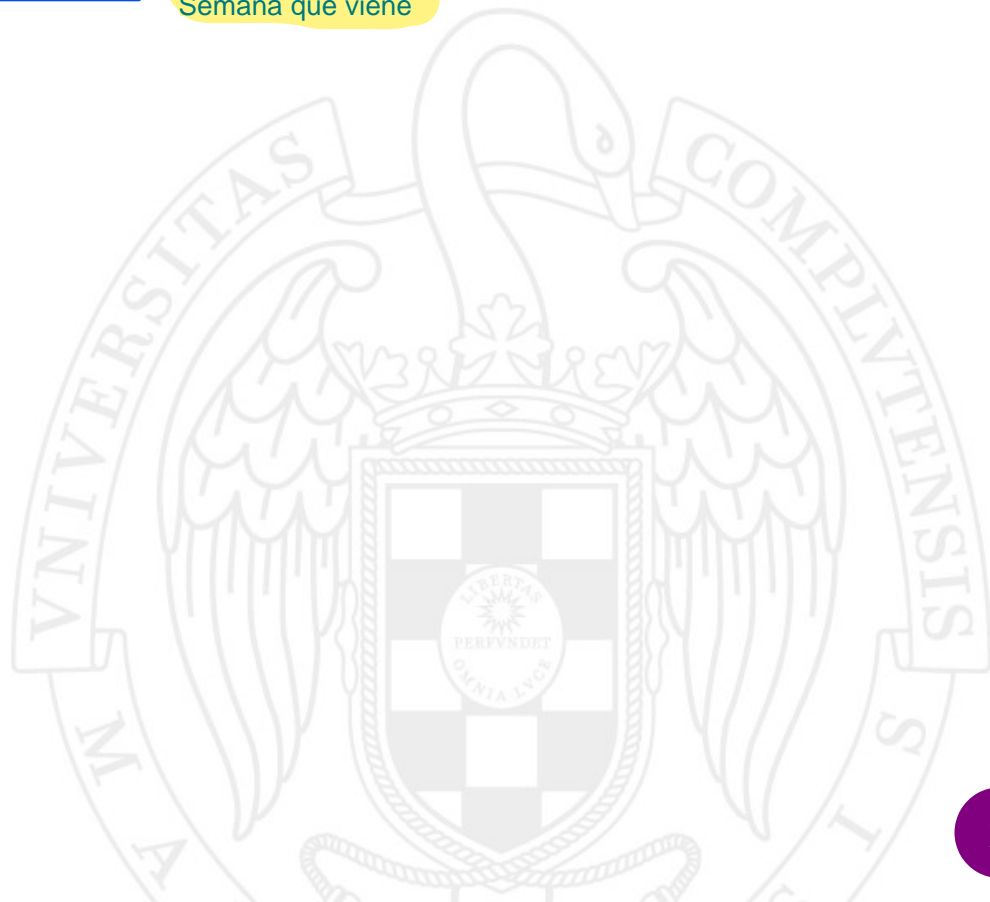
- Saber si existe una entrada con una clave determinada: **contains**
- Saber el valor asociado con una clave: **at**
- Saber si el diccionario está vacío: **empty**
- Saber el número de entradas del diccionario: **size**

# Dos implementaciones

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

**Este vídeo**

Semana que viene



# Interfaz de MapTree

Utilizamos la clase MapTree

```
template <typename K, typename V>
class MapTree { Diccionario en ABBs
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:
    // ...
};
```

esto es una clase paramétrica respecto de las claves y los valores.

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

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

Respecto del vídeo anterior ha decidido añadir dos constructoras: una que inicializa únicamente la clave y otra que inicializa tanto la clave como su valor asociado.

El resto de métodos son iguales que los del vídeo anterior.

# Representación privada de MapTree

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

```
    struct Node {
```

Esto es debido a que vamos a utilizar árboles binarios de búsqueda.

```
        MapEntry entry;
```

Nodo que apunta al hijo izquierdo y derecho

```
        Node *left, *right;
```

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

```
    Node *root_node;
```

apunta al nodo raíz del árbol binario de búsqueda.

```
    int num_elems;
```

número de nodos que tengo en el ABB por motivos de eficiencia para que haya coste constante. En este caso la operación size() devuelve ese número de elementos. Coste constante. De otra forma el coste sería lineal ya que tendríamos que recorrer el árbol binario.

```
    // métodos auxiliares privados
```

```
    // ...
```

```
};
```

Los métodos que utilizábamos en los conjuntos.

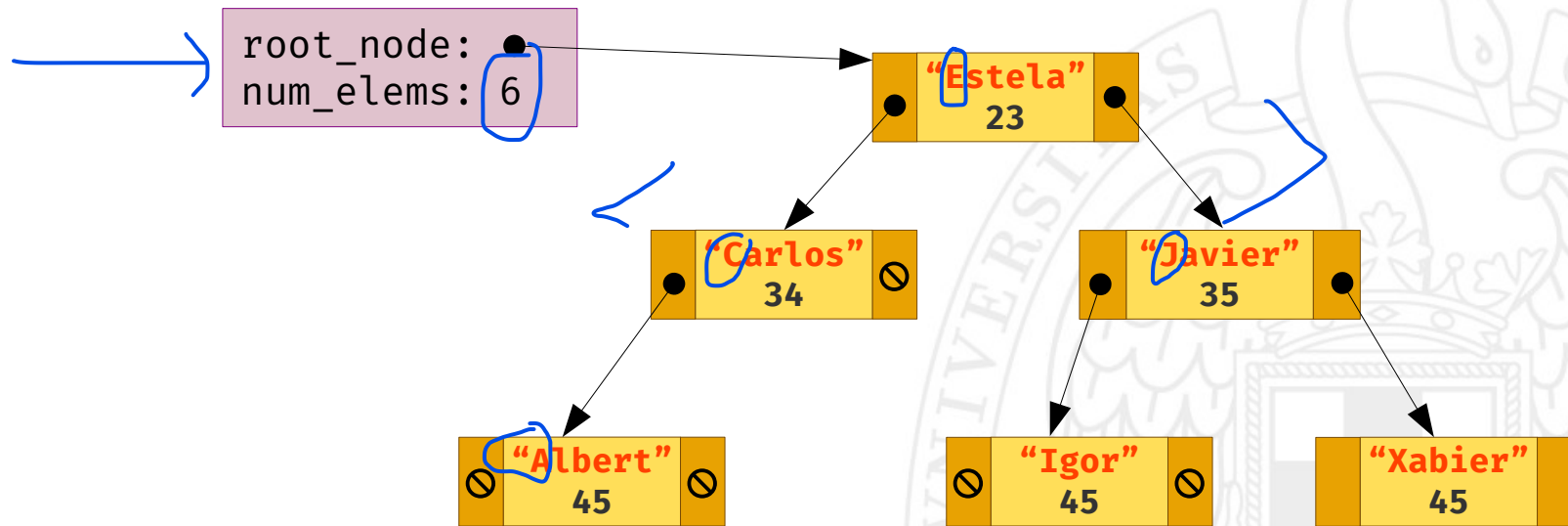
```
struct MapEntry {
    K key;
    V value;

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

Mapa con su clave y su valor asociado y los respectivos constructores que inicializan ambas o únicamente la clave.

# Representación de un MapTree

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

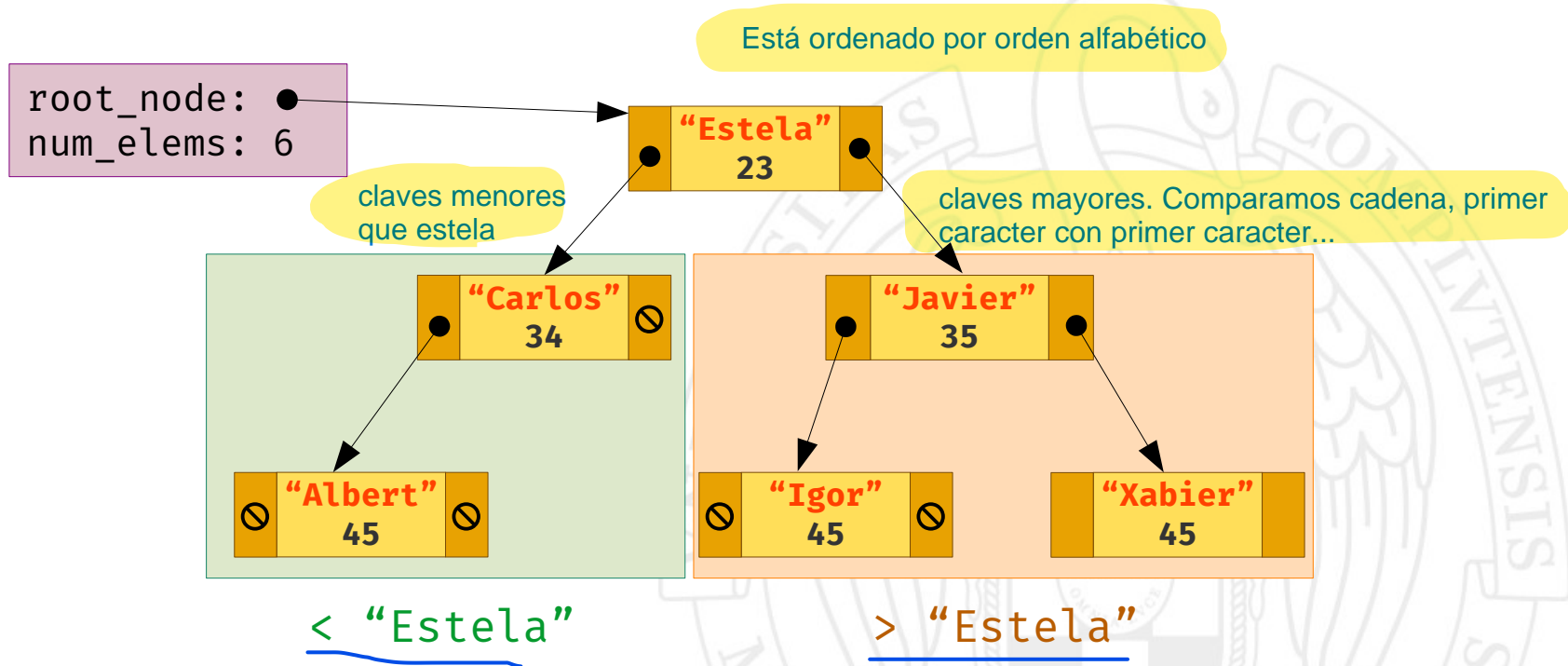


CREO que está ordenado en orden alfabético.

Claro, ordena por la clave.

# 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

Los métodos que habíamos utilizado para los árboles binarios de búsqueda.

```
template <typename K, typename V>
class MapTree {
```

```
    ...
private:
    ...
```

estos tres son los que más utilizamos, pero hay más si no me equivoco.

```
    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;
    }
};
```

Qué deberíamos de cambiar en el método de búsqueda en el caso de la implementación con diccionarios.

comparo la clave con la clave del nodo en el que estoy yo ahora.

Esto es lo mismo para los métodos de inserción y borrado

ESTO LO EXPLICO EN UNA DE LAS DIAPOSITIVAS SIGUIENTES (EN LA 11)

# Métodos contains() y at()

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

```
    ...
    bool contains(const K &key) const {
        return search(root_node, key) != nullptr;
    }
```

LLAMA A SEARCH Y BUSCA EL NODO QUE TENGA ESA CLAVE.

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

BUSCA EL NODO Y EXIGIMOS QUE EL NODO SE HAYA ENCONTRADO.

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

```
};
```

ACCESO AL VALOR.

```
Node * insert(Node *root, const MapEntry &elem) {  
    if (root == nullptr) {  
        return new Node(nullptr, elem, nullptr);  
    } else if (elem < root->entry.key) {  
        Node *new_root_left = insert(root->left, root->entry.key);  
        root->left = new_root_left;  
        return root;  
    } else if (root->entry.key < elem) {  
        Node *new_root_right = insert(root->right, root->entry.key);  
        root->right = new_root_right;  
        return root;  
    } else {  
        return root;  
    }  
}
```

SI NO ME EQUIVOCO, LA OPERACIÓN INSERT SERÍA ALGO ASÍ.

## Búsqueda e inserción mediante [ ]

Nueva operación.

No es esencial pero nos facilita nuestra tarea con diccionarios.

# Motivación

- Muchas veces encontramos código como este:

```
if (!dicc.contains(k)) { Vemos si contiene cierta clave k.  
    words.insert({k, 1}); si no la inserta.  
} else {  
    words.at(k) = ... ; si si le asigna un valor.  
}
```

- No es equivalente a:

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

Si no se encuentra inicializarlo con el valor de 1. Pero esto no se puede hacer con la operación at.

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

# Motivación

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

`dicc.at(key)`

Vemos cual es la principal diferencia

`dicc[key]`

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

=

≠

- 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.

Ambas devuelven el valor asociado a `key` pero la diferencia está en que si la clave no se encuentra.

# Nueva operación auxiliar

Mezcla entre búsqueda o inserción.

```
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};
```

```
    }
```

```
}
```

devuelve una tupla de 3 componentes:

1ª: Indica si ha habido una inserción en el árbol (true si ha habido inserción).

2ª: Puntero a la nueva raíz tras la posible inserción.

3ª: Puntero al nodo del árbol que contenga la clave key pasada como parámetro.

Puntero a la raíz del ABB

Esta función busca un nodo con la clave key en el ABB

Si ese nodo no existe crea uno nuevo con esa 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) { clave no se encuentra. Creamos nodo con esa clave.
```

```
        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};
```

```
    }
```

```
}
```

constructor de un solo  
parámetro

```
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) {
```

Si árbol no es vacío.

```
        auto [inserted, new_root, found_node] = search_or_insert(root->left, key);
```

```
        root->left = new_root;
```

buscamos en el hijo izquierdo.

```
        return {inserted, root, found_node};
```

```
    } else if (root->entry.key < key) {
```

caso contrario buscamos en el hijo derecho.

```
        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};
```

```
    }
```

```
}
```

Clave que buscamos es menor que la clave del nodo visitado.



# 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 { árbol NO es vacío y la clave que buscamos esta y es la raíz del árbol. La hemos encontrado.
        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;
}
};
```

Raíz del árbol binario de búsqueda.

Si hay una inserción debemos de incrementar el número de elementos (obvio)


devuelve referencia al valor contenido dentro de ese nodo.

# 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) = ... ;  
}
```



Esto era incorrecto porque la operación at exige que esté la clave dentro del diccionario.

# 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>	$O(\log n)$	$O(n)$

·  $O$  implementación que reequilibra el árbol

$n$  = número de entradas en el diccionario