

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

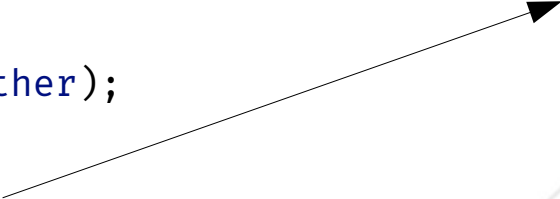
```
template <typename K, typename V>
class MapTree {
public:
    MapTree();
    MapTree(const MapTree &other);
    ~MapTree();

    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;
```

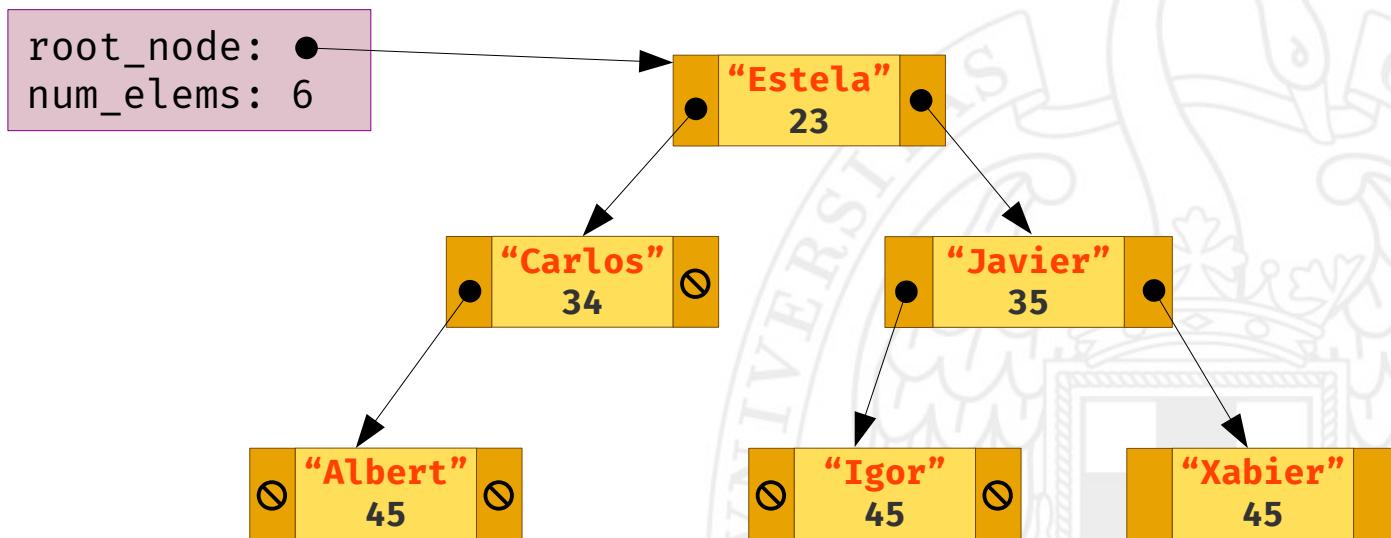
```
    // métodos auxiliares privados
    // ...
};
```

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

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

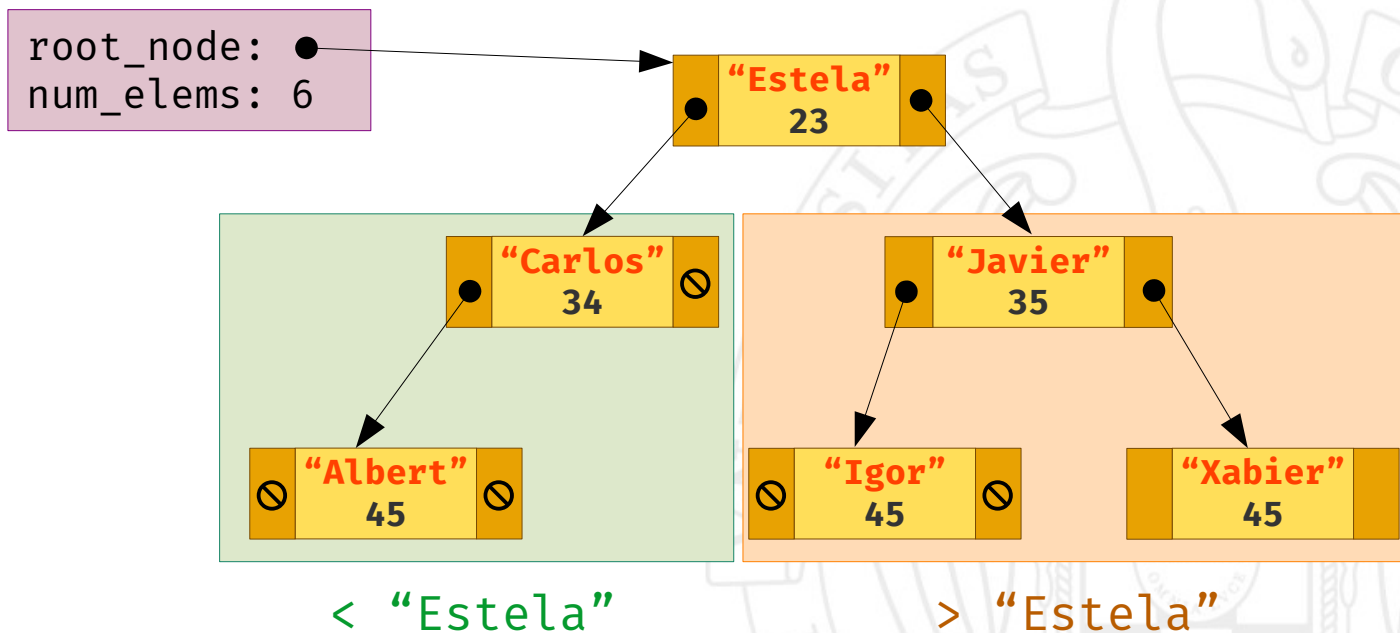
Representación de un MapTree

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



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