

ESTRUCTURAS DE DATOS

DICCIONARIOS

Relaciones de orden en ABBs

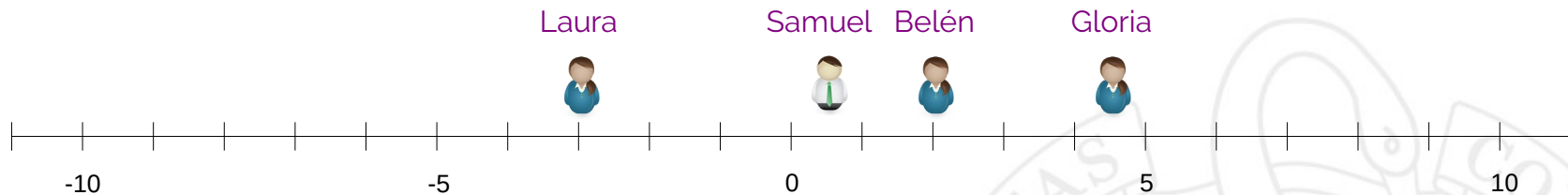
Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

Motivación

Aquí vamos a hablar de las comparaciones

Cómo se parametrizan los diccionarios.

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



- ¿Cómo represento el estado actual?

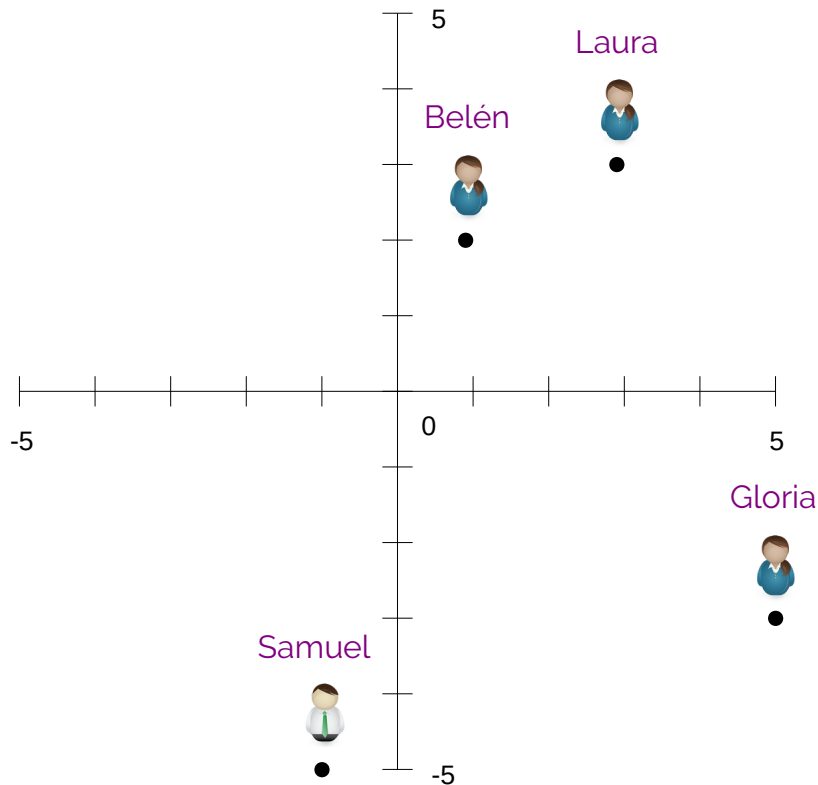
```
MapTree<double, string> posiciones;
```

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

diccionarios que asocia posiciones double a personas o nombres.

Motivación

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



Eje de coordenadas.

```
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"});
```



Pues esto tal cual está ahora NO funciona. El compilador dice que da error.

¿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) {  
             ~~~~~^~~~~~
```

No es capaz de aplicar el operador "<" a los argumentos. Que son de tipos Coord. No tenemos definido el operador para Coord.

¿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`!

De esta forma no tenemos que sobrescribir tanto el operando ">" como el "<"

Vamos a explorar distintas soluciones para que podamos arreglar lo anterior. Es la solución más obvia.

Solución 1: implementar <

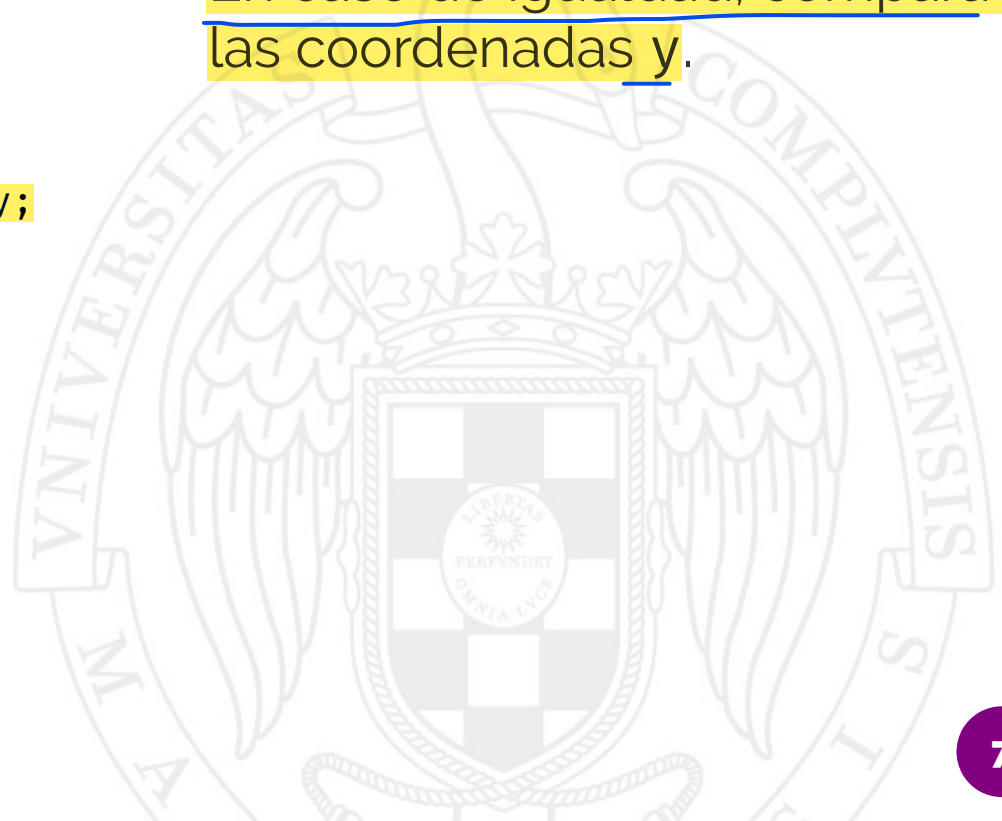
Solución 1

```
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 .

Si lo comparas consigo mismo debe devolver false.

- **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 operador $<$ que definamos cumpla estas tres propiedades, pero si no las cumple, el ABB puede comportarse de manera inconsistente.

Nosotros nos tenemos que encargar de que se cumplan estas propiedades.

- 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?

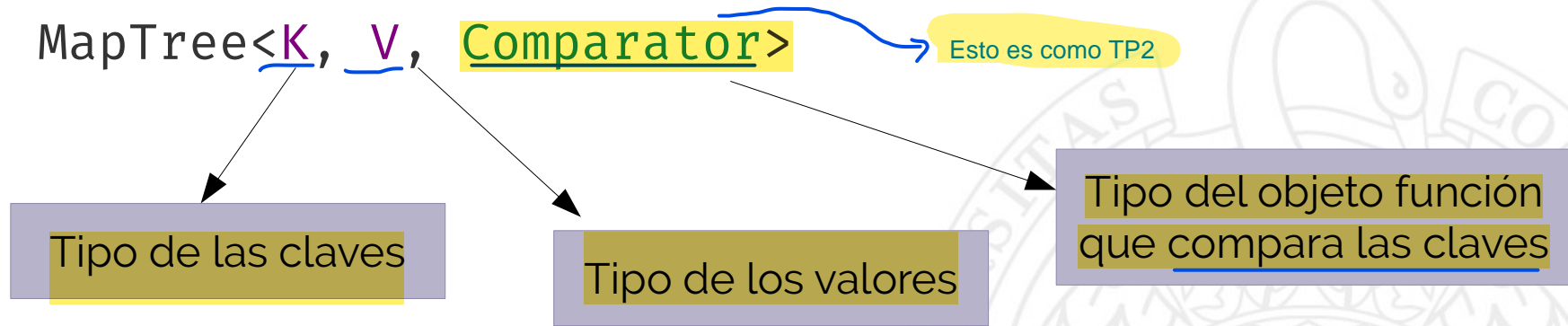
Esto con el operador $<$ no lo vamos a poder conseguir.

Solución 2: parametrizar MapTree

con respecto a la relación de orden.

Parametrizar MapTree

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

Esto lo hacen todos los objetos función.

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

Objetos función

Este es nuestro objeto 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;  
    }  
};
```

Es nuestra clase Comparator.

```
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"});
```

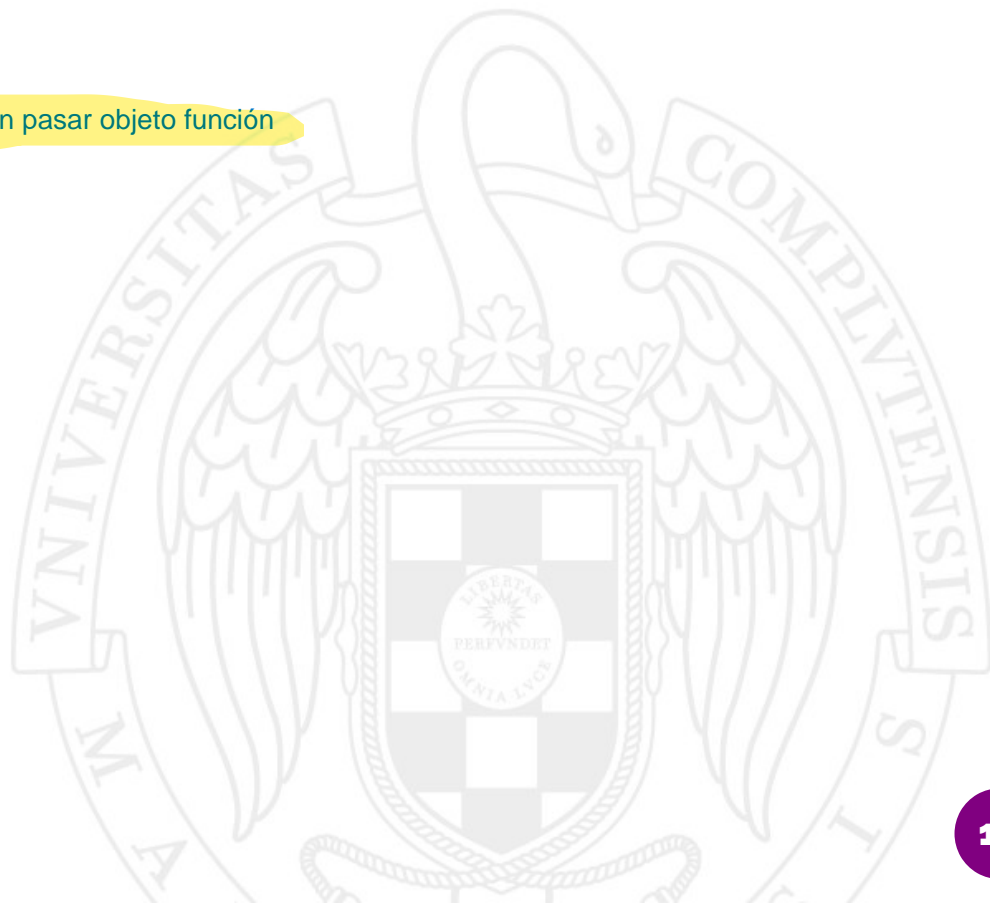
Objetos se insertarían en el diccionario de manera ordenada según el orden definido

¿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"});
```

Sin pasar objeto función



Implementación

$\lambda(x,y) \rightarrow x < y$

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

operador parentesis

```
    ...  
private:  
    struct Node { ... };
```

```
    Node *root_node;
```

```
    int num_elems;
```

```
    ComparatorFunction less_than;
```

Instancia de la clase que recibe como parámetro

```
    ...  
}
```

Objeto función.

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;
        }
    }
};
```

En vez de utilizar el operador < utilizaremos nuestro objeto función.

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;
        }
    }
};
```

compara las claves, si tenemos (X,Y) y $x < y$ entonces es menor.