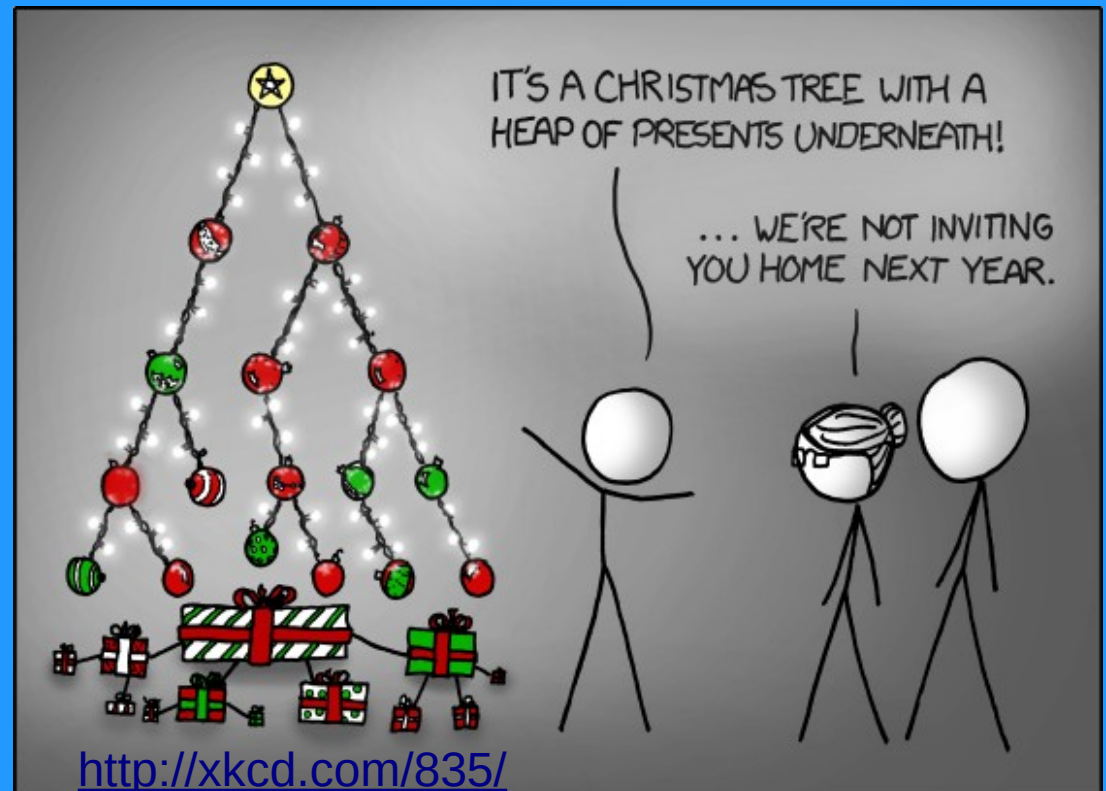


# Estructuras de Datos

## 2020



# Recursión

If pants wore pants, would it wear them  
like this?

or like this?



**“Para entender la Recursión, hay que  
entender la Recursión..Y para  
entender Recursión hay que  
entender recursión  
Y si ahora entiendes recursión  
entendiste Recursión!”  
:-)**

# Recursión

## Definición

La **Recursividad** es una de las Herramientas que permiten expresar la resolución de Problemas Evolutivos **donde es posible que un Módulo de Software se invoque a sí mismo en la evolución del Problema a la Solución.**

Cuando se obtienen soluciones a problemas en los que una función se llama a sí misma para resolver el problema, se tiene Una función recursiva

# Recursión. Un ejemplo cotidiano

## Buscar una palabra en el diccionario

Una solución posible podría ser buscarla secuencialmente...

¿Otras alternativas?

# Recursión

Si aprovechamos el orden:

```
//Suponemos que la palabra buscada se encuentra en el diccionario
```

```
Si el diccionario tiene 1 sola página
```

```
    entonces La encontramos! obtener el significado y retornarlo
```

```
    sino
```

```
        Abrir el dicc. en un punto cercano al medio
```

```
        Determinar en qué mitad del dicc. Está la palabra
```

```
        Si la palabra está en la 1era mitad entonces
```

```
            buscar la palabra en la 1era mitad
```

```
            sino
```

```
                buscar la palabra en la 2da mitad
```

```
        fin si
```

```
    fin si
```

```
fin buscar
```

En cualquier instancia de resolución, buscar la palabra en la 1era mitad del diccionario o en la 2da mitad implica VOLVER a dar los mismos pasos que en la instancia anterior...

# Recursión

Si aprovechamos el orden:

```
Buscar(diccionario, palabra)
```

```
Si el diccionario tiene 1 sola página
```

```
    entonces La encontramos! obtener el significado e informarlo
```

```
    sino
```

```
        Abrir el diccionario en un punto cercano al medio
```

```
        Determinar en qué mitad del diccionario Está la palabra
```

```
        Si la palabra está en la 1era mitad entonces
```

```
            Buscar(primer mitad de diccionario, palabra)
```

```
            sino
```

```
                Buscar(segunda mitad de diccionario, palabra)
```

```
        fin si
```

```
    fin si
```

```
fin buscar
```

La estrategia es dividir el diccionario a la mitad y aplicar la misma idea sobre la mitad Seleccionada...

# Recursión

## Observaciones importantes en la solución anterior :

Una vez divido el diccionario y determinada la parte que contiene a la palabra, el método de búsqueda que usamos para esa mitad, es el mismo al usado para el diccionario completo.

La mitad del diccionario donde seguro no esta la palabra la descartamos, lo que implica una reducción en el tamaño del problema

Existe un caso distinto que se resuelve de manera particular (diferente del resto) que sucede cuando el diccionario queda reducido a una sola página.

Lo vamos a llamar  
Caso Base (puede haber  
Más de uno!)



# Recursión

Cuando se construye una solución recursiva para un Problema cualquiera se deberían hacer las siguientes preguntas:

- 1-¿Cómo definir el problema en términos de un problema más simple de la misma naturaleza?
- 2-¿Cómo será disminuido el tamaño del problema en cada llamado recursivo?
- 3-¿Qué instancia del problema servirá como **caso base**?

Cuando el Caso Base es alcanzado la Recursividad termina y el Problema se Resuelve directamente. Es importante notar que la reducción del tamaño del Problema garantiza que el Caso Base sea alcanzado

# **Ejecución de un Programa Pila de Activación**

# Recursión

Para entender mejor como funcionan los módulos recursivos es importante analizar el comportamiento de la **Pila de activación**.

Básicamente en un Procesador hay un módulo activo (instrucciones + datos) y cuando este invoca a otro módulo que toma el control del procesador, el nuevo módulo "se apila" en memoria con sus instrucciones y datos locales.

# Recursión

Cuando un programa comienza su ejecución se crea la pila de activación, desde aquí cada elemento que se agregue a la pila ocupará un nuevo segmento de memoria (*frame*)

Cuando un módulo finaliza su ejecución su frame se quita de la pila.

Lo vemos con ejemplo: Factorial

# Recursión

Factorial (n) =  $n * (n-1) * (n-2) * \dots * 1$

Factorial (0) = 1

```
def Factorial (n):  
    fact = 1 # Contiene el calculo parcial  
    for x in range(1,n+1):  
        fact = fact * x;  
    return fact  
  
print(Factorial(6))
```

# Recursión

Factorial (n) =  $n * (n-1) * (n-2) * \dots * 1$

Factorial (0) = 1

```
def Factorial(n):  
    if ( n <= 1 ):  
        return 1  
    else:  
        return n * Factorial (n-1)  
  
print(Factorial(6))
```

[Ver en Python Tutor](#)

Ver en [www.cs.usfca.edu/~galles/visualization/RecFact.html](http://www.cs.usfca.edu/~galles/visualization/RecFact.html)

# Recursión. Problemas

Cada invocación a la función hace una copia del área de datos para esa función cada llamado crea una nueva entrada en la pila de activación.

En el caso (3) se llega al caso degenerado, allí cuando la función termina su ejecución libera el espacio

ocupado en la memoria y retorna el control a Factorial con  $N=2$ ,

A partir de este punto se resuelven los cálculos y se retorna resultado.

Qué pasa si queremos calcular el factorial de 500?

Cuántos frames habrá?

Cuánta memoria se utilizará?

# Recursión

Pensemos em el método `__eq__` para una pila:

```
def __eq__(self, other):  
    x = self._top  
    y = other._top  
    while x is not None and y is not None:  
        if x.value != y.value:  
            return False  
        x = x.next  
        y = y.next  
    return x is None and y is None
```

¿ Alternativa recursiva?



# Recursión

Pensemos em el método `__eq__` para una pila:

```
def __eq__(self, other):  
    def are_equal(x, y):  
        if x is None or y is None:  
            return x is None and y is None  
  
        if x.value != y.value:  
            return False  
  
        return are_equal(x.next, y.next)  
    return are_equal(self._top, other._top)
```

Si x e y son None  
retorna True y si  
una de ellas  
es None retorna False

# Recursión

Pensemos em otra implementación posible:

```
def _eq_(self, other):  
    def are_equal(x, y):  
        if x is None and y is None:  
            return True  
        elif x is None or y is None:  
            return False  
        else:  
            if x.value != y.value:  
                return False  
            return are_equal(x.next, y.next)  
    return are_equal(self._top, other._top)
```

¿Ventajas/Desventajas?

# Pila de Activación y Tad Pila

La pila de activación tiene el comportamiento del TAD pila visto en clases anteriores.

En algunos casos nos va convenir usar una estructura Auxiliar (pila) en otros podemos aprovechar **Pila de Ejecución y la Recursión.**

# Recursión

## **Algunas Conclusiones**

La recursión es un arma poderosa para expresar en forma sintética y clara problemas donde un módulo debe invocarse a sí mismo.

Algunos problemas pueden resolverse con la misma facilidad usando tanto recursión como iteración

En general para un mismo algoritmo la recursión permite una expresión más clara y sintética lo que simplifica la comprensión y el mantenimiento.

A su vez las soluciones recursivas son normalmente más costosas en tiempo y memoria (menos eficientes).

# Árboles

# Árboles

Si Necesitamos representar una Jerarquía...

- Con listas simples:

Tenemos un sucesor para cada elemento

- Con listas dobles:

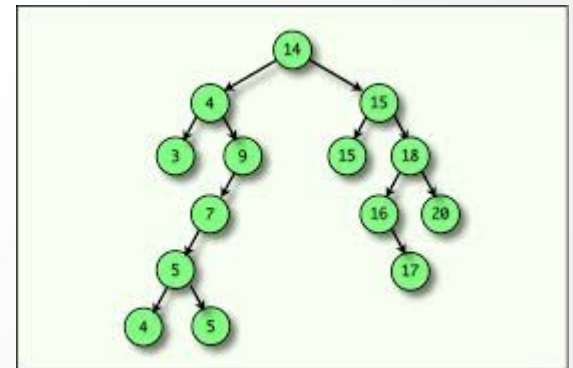
Un sucesor y un predecesor.

Dos sucesores (para dos criterios diferentes)

# Árboles

**Hasta ahora no vimos una estructura que permita representar “jerarquías”**

Ej: relaciones familiares, orden de elementos de manera más eficiente...



Usamos  
Árboles!

# Árbol General



# Árboles. Definición

Un árbol es una estructura de datos que satisface tres propiedades:

- 1- cada elemento del árbol (nodo) se puede relacionar con cero o más elementos a quienes llama “hijos”;
- 2-si el árbol no está vacío, hay un único elemento al cual se llama raíz y que no tiene padre (predecesor), es decir, no es hijo de ningún otro;
- 3-todo otro elemento del árbol posee un único padre y es un descendiente (hijo del hijo del hijo, etc.) de la raíz.

Árbol es un conjunto de nodos que:  
Es vacío, o bien,  
Tiene un nodo raíz del que descienden 0 o más subárboles.

# Árboles

## Algunas aplicaciones para los Árboles:

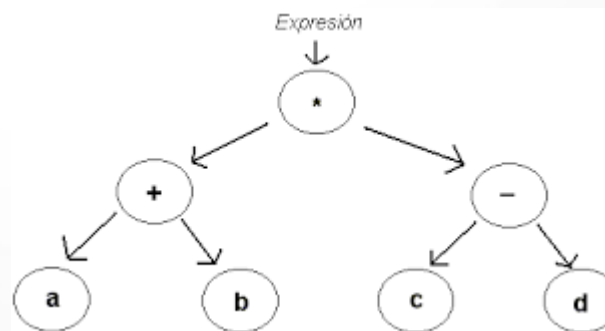
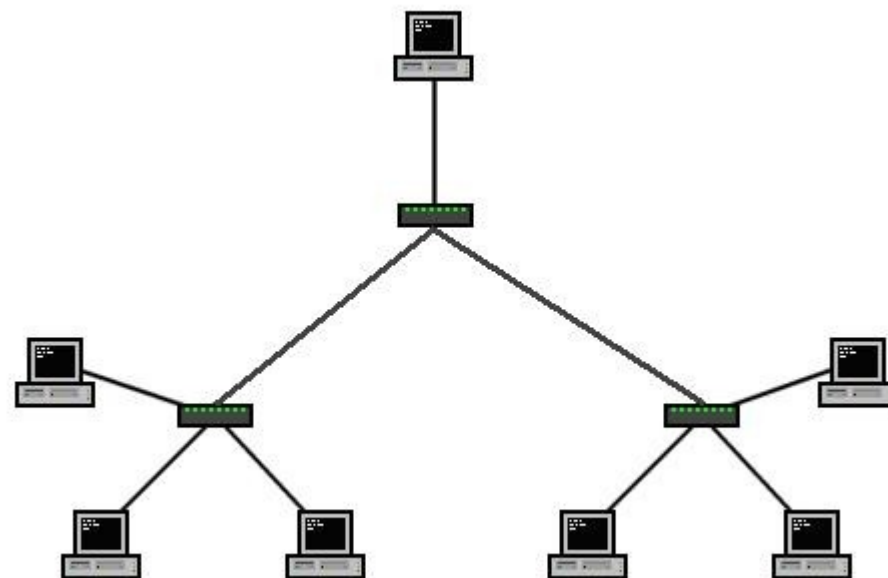
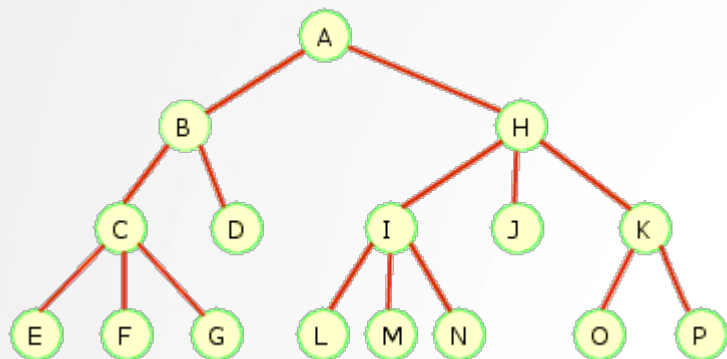
Es posible desarrollar algoritmos de búsqueda eficientes representando la secuencia de **pruebas de una búsqueda binaria** como un árbol.

Pueden organizarse archivos en una computadora utilizando un **árbol de directorios**.

Es posible evaluar expresiones aritméticas y ejecutar programas representando la sintaxis de estas expresiones y **programas como un árbol**.

Es posible determinar quien es el ganador de un juego de estrategia **representando las posiciones permitidas en el juego mediante un árbol**

# Árboles



# Árboles. Más Definiciones

- Se dice que un nodo a es descendiente directo (**hijo**) de un nodo b, si a es apuntado por b, y es ascendiente directo (padre) si a apunta a b.
- La **raíz de un árbol** se define como el nodo que no tiene ascendientes.
- Si un nodo no tiene descendientes se lo denomina **hoja o nodo terminal**.
- El enlace entre dos nodos se conoce como **arista**.

# Árboles. Más Definiciones

- Se denomina ***grado de un nodo*** al número de descendientes directos del nodo.
- El **grado de un árbol** queda determinado por el mayor de los grados de los nodos.
- Se denomina ***nivel de un nodo*** al número de descendientes que deben recorrerse desde la raíz hasta dicho nodo. **El nivel de la raíz es 0.**
- El nivel máximo en que se encuentran los elementos de un árbol se denomina ***profundidad o altura***.

# **Árboles Binarios**

**ABB**

# Árboles Binarios Y ABB

Un árbol donde cada nodo no tiene más de dos hijos se denomina **árbol binario** (el grado es a lo sumo dos)

Un árbol binario se dice que está ordenado cuando cada nodo del árbol es mayor (según algún criterio preestablecido) que los elementos de su subárbol izquierdo (el que tiene como raíz a su hijo izquierdo) y menor o igual que los de su subárbol derecho (el que tiene como raíz a su hijo derecho). **A este árbol lo llamaremos ABB (árbol binario de búsqueda)**

**Especificamos el**  
**ABB**



# Especificación. “Molde General”

TAD Nombre TAD

Igualdad Observacional

Usa

Parámetro Formal

Géneros

observadores básicos

Generadores

otras operaciones

Axiomas

Exporta

# Especificación. ABB

**TAD** ArbolBinarioDeBusqueda<a, b>

## Igualdad Observacional

Si **a** y **b** son dos árboles binarios de búsqueda

**a** es igual a **b** si se cumple que: a y b tienen los mismos elementos

## Usa

Natural, Bool, Secuencia<Tupla <a, b>>, Tupla<a, b>, None,  
coordenada<ArbolBinarioDeBusqueda<a, b>>

## Parámetro Formal

a, b

## Géneros

ArbolBinarioDeBusqueda <a, b>

# Especificación. ABB

## observadores básicos

`cantidad_de_nodos(ArbolBinarioDeBusqueda<a, b>) → Natural`

`es_vacío(ArbolBinarioDeBusqueda<a, b>) → Bool`

`máximo(ArbolBinarioDeBusqueda<a, b>) →  
Coordenada<ArbolBinarioDeBusqueda<a, b>>`

`{Pre: el árbol tiene al menos un elemento}`

`mínimo(ArbolBinarioDeBusqueda<a, b>) →  
Coordenada<ArbolBinarioDeBusqueda<a, b>>`

`{Pre: el árbol tiene al menos un elemento}`

`buscar_clave(ArbolBinarioDeBusqueda<a, b>, a) →  
Coordenada<ArbolBinarioDeBusqueda<a, b>>`

# Especificación. ABB

## Generadores

`vacío() → ArbolBinarioDeBusqueda<a, b>`

`{Post: El Árbol retornado esta vacío}`

`a_partir_de(Secuencia<Tupla<a, b>>) →  
ArbolBinarioDeBusqueda<a,b>`

`{Post: El árbol contiene los elementos de la secuencia  
recibida}`

# Especificación. ABB

## Otras Operaciones

`insertar(ArbolBinarioDeBusqueda<a, b>, a,  
b)→Coordenada<ArbolBinarioDeBusqueda<a, b>>`

`{Pos: El árbol no esta vacío}`

`borrar(ArbolBinarioDeBusqueda<a, b>, a)→ Bool,  
Coordenada<ArbolBinarioDeBusqueda<a, b>>`

# Especificación. ABB

## Otras Operaciones

**inicio(ArbolBinarioDeBusqueda<a, b>)→  
coordenada<ArbolBinarioDeBusqueda<a, b>>**

**fin(ArbolBinarioDeBusqueda<a, b>)→  
coordenada<ArbolBinarioDeBusqueda<a, b>>**

# Especificación. ABB

## Axiomas

**vacío():** Crea un árbol vacío (sin elementos)

**a\_partir\_de(Secuencia<Tupla<a,b> s):** crea un árbol que contiene las claves de tipo a asociadas a los valores de tipo b dados en la secuencia s

**insertar(ArbolBinarioDeBusqueda<a, b> t, a clave b valor):** agrega la clave en el árbol t con valor asociado

**borrar(ArbolBinarioDeBusqueda<a, b> t, a clave):** borra del árbol t la clave y su valor asociado.

**tamaño(ArbolBinarioDeBusqueda<a, b> t):** Retorna/ devuelve la cantidad de elementos del árbol t

# Especificación. ABB

`es_vacíó(ArbolBinarioDeBusqueda<a, b> t):`  
Retorna/devuelve verdadero si el árbol t esta vacío y falso en caso contrario

`máximo(ArbolBinarioDeBusqueda<a, b> t):` retorna/  
devuelve una coordenada que hace referencia al mayor elemento del árbol t.

`mínimo(ArbolBinarioDeBusqueda<a, b> t):` retorna/  
devuelve una coordenada que hace referencia al menor elemento del árbol t.

`buscar_clave(ArbolBinarioDeBusqueda<a, b> t, a clave):` si clave pertenece al árbol t retorna una coordenada que hace referencia al elem con dicha clave. En caso contrario retorna una coordenada que hace referencia al siguiente del último elemento del árbol



# Especificación. Lista Doble

`inicio(ArbolBinarioDeBusqueda<a, b> t):` devuelve una coordenada que hace referencia al primer elemento del árbol `t`

`fin(ArbolBinarioDeBusqueda<a, b> t):` devuelve una coordenada que hace referencia al siguiente del último elemento del árbol `t`

## Exporta

`ArbolBinarioDeBusqueda<a, b>, vacío, a_partir_de, insertar, borrar, tamaño, es_vacío, mínimo, máximo, buscar_clave, inicio, fin`

# **Implementación del TAD**

## **ABB**

**Estructura Interna  
que define el ABB**

# TAD ABB

```
class TreeDict():
```

```
@dataclass
```

```
class _Node:
```

```
    key: Any # Comparable
```

```
    value: Any
```

```
    parent: Union['_Node', '_Root'] = None
```

```
    left: '_Node' = None
```

```
    right: '_Node' = None
```

```
@dataclass
```

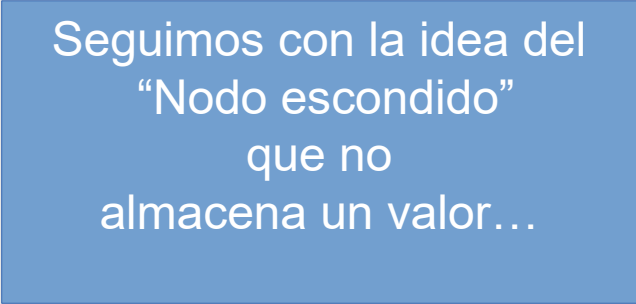
```
class _Root:
```

```
    left: '_Node' = None
```

```
    right: '_Node' = None
```

```
    parent: '_Node' = None
```

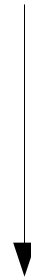
```
__slots__ = ['_root', '_len']
```



Seguimos con la idea del  
“Nodo escondido”  
que no  
almacena un valor...

# AB y ABB

la representación interna para un árbol binario  
Y un ABB podría ser la misma?



La estructura interna puede ser la misma  
Lo que cambian son las  
Implementaciones de las  
Operaciones

**Generadores...**

# Especificación. ABB

## Generadores

`vacío() → ArbolBinarioDeBusqueda<a>`

`{Post: El Árbol retornado esta vacío}`

`a_partir_de(Secuencia<a>) → ArbolBinarioDeBusqueda<a>`

`{Post: El árbol contiene los elementos de la secuencia recibida}`

# Implementación. Lista Doble

```
def __init__(self, iterable=None):  
  
    self._root = TreeDict._Root()  
    self._len = 0  
    if iterable is not None:  
        for key, value in iterable:  
            self.insert(key, value)
```



**Implementamos las  
operaciones..**

# Especificación. ABB

## observadores básicos

`cantidad_de_nodos(ArbolBinarioDeBusqueda<a>) → Natural`

`es_vacío(ÁrbolBinarioDeBusqueda<a>) → Bool`

`máximo(ArbolBinarioDeBusqueda<a>) →  
Coordenada<ArbolBinarioDeBusqueda<a>>`

`{Pre: el árbol tiene al menos un elemento}`

`mínimo(ArbolBinarioDeBusqueda<a>) →  
Coordenada<ArbolBinarioDeBusqueda<a>>`

`{Pre: el árbol tiene al menos un elemento}`

`buscar_clave(ÁrbolBinarioDeBusqueda<a>, a elem) →  
Coordenada<ArbolBinarioDeBusqueda<a>>`

# Implementación. Tamaño y es\_vacío?

```
def is_empty(self):  
    return self._root.left is None
```

```
def __len__(self):  
    return self._len
```

# Implementación. Máximo y Mínimo

```
def minimum(self):  
    return TreeDict._Coordinate(_minimum_node(self._root))  
  
def maximum(self):  
    return TreeDict._Coordinate(_maximum_node(self._root.left))
```

# Implementación. Máximo y Mínimo

```
def _minimum_node(node):  
    If node is not None:  
        while node.left is not None:  
            node = node.left  
    return node  
  
def _maximum_node(node):  
    If node is not None:  
        while node.right is not None:  
            node = node.right  
    return node
```

# Implementación. Buscar

```
def find(self, key):  
    def do_find(node):  
        if node is None:  
            return self.end()  
        elif key < node.key:  
            return do_find(node.left)  
        elif key > node.key:  
            return do_find(node.right)  
        else: # key == node.key  
            return TreeDict._Coordinate(node)  
    return do_find(self._root.left)
```

¿Lo podemos implementar iterativo?

# Implementación. Buscar

```
def find(self, key):  
    node= self._root.left  
    while True:  
        if node is None:  
            return self.end()  
        elif key < node.key:  
            node=node.left  
        elif key > node.key:  
            node=node.right  
        else: # key == node.key  
            return TreeDict._Coordinate(node)
```

¿Ventajas?

¿Desventajas?

# ABB

**En la próxima clase implementamos  
Más operaciones...**