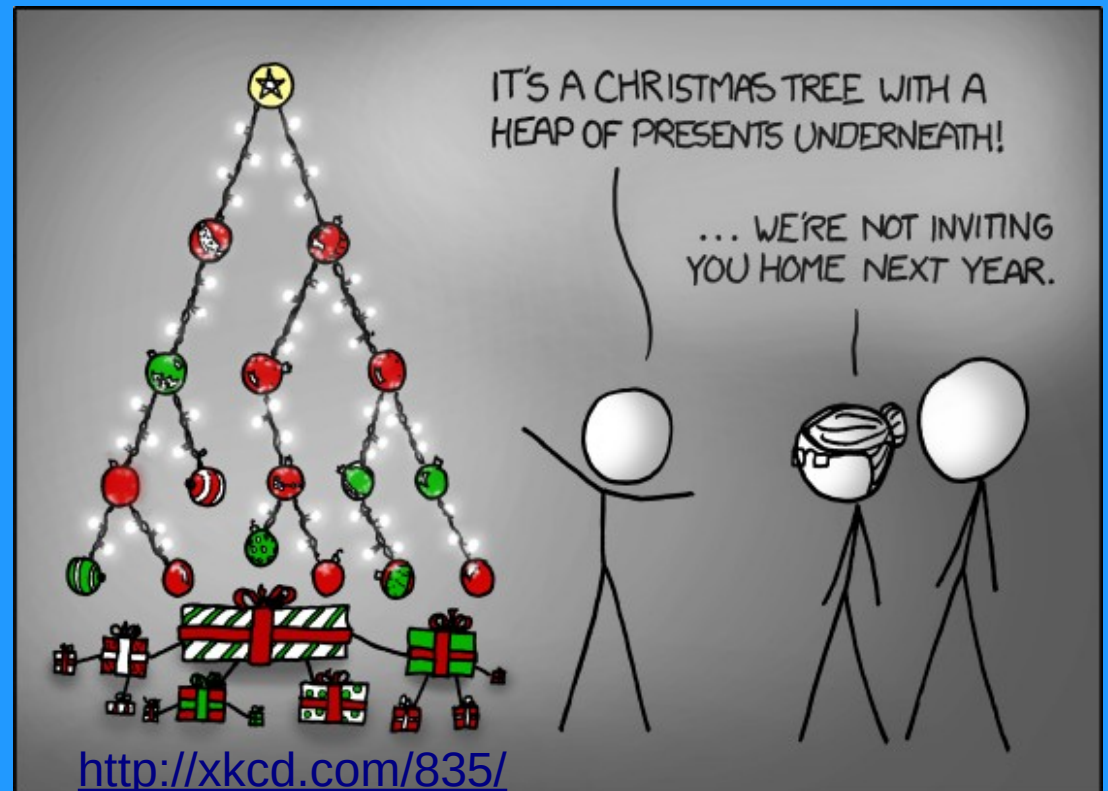


Estructuras de Datos

2020



ABB

**Implementamos
más Operaciones...**

**Recordemos la
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...

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>>

¿Debería pasar por
Parámetro
Una posición o
coordenada?

**Insertar un
elemento**

Tipos de parámetros en Python..

Al implementar debemos tener en cuenta que Python soporta un único tipo de pasaje de parámetro: “Copia de referencia a objeto”.

No disponemos del pasaje “por referencia”.

Tipos de parámetros en Python..

Es por lo anterior que en las implementaciones donde necesitamos **cambiar en un objeto la referencia a otro objeto** retornamos lo necesario para modificarla “a la vuelta de la invocación”...

ABB. Insertar un elemento

```
def insert(self, key, value=None):
    def do_insert(node, parent):
        if node is None:
            node = TreeDict._Node(key, value, parent)
            coord = TreeDict._Coordinate(node)
            self._len += 1
        elif key < node.key:
            node.left, coord = do_insert(node.left, node)
        elif key > node.key:
            node.right, coord = do_insert(node.right, node)
        else: # key == node.key
            node.value = value
            coord = TreeDict._Coordinate(node)
        return node, coord
    self._root.left, coord = do_insert(self._root.left, self._root)
    return coord
```

La función do_insert
Es interna a insert

**Eliminar un
elemento**

ABB. Borrar un elemento

Tenemos 3 casos para considerar:

1-No hay ningún nodo con llave igual a la que se desea eliminar: No hay que hacer ninguna tarea en particular
Retornamos False

2-El nodo a eliminar tiene como máximo un subárbol descendiente: Dos posibilidades:

- Si no tiene subárbol izquierdo, cambiar el puntero al nodo a eliminar para que apunte al subárbol derecho del nodo a eliminar.
- Si no tiene subárbol derecho, cambiar el puntero al nodo a eliminar para que apunte al subárbol izquierdo del nodo a eliminar.

ABB. Borrar un elemento

Tenemos 3 casos para considerar:

3-El nodo a eliminar tiene dos subárboles descendientes:

Caso más complejo, no se puede solucionar con “enganches” sencillos y hay dos posibilidades:

- Cambiar el puntero al nodo a eliminar para que apunte al nodo que contiene **la mayor clave del subárbol izquierdo del nodo a eliminar**.
- Cambiar el puntero al nodo a eliminar para que apunte al nodo que contiene **la menor clave del subárbol derecho del nodo a eliminar**.

En ambos casos se debe tener en cuenta que el nodo usado para reemplazar al que se está eliminando puede tener un subárbol izquierdo o derecho (según cada caso) propio.

ABB. Borrar un elemento

```
def erase(self, key):  
    #Funciones internas: do_erase, erase_node, extract_maximum,  
    # y assign_parent  
    ...  
  
    result, self._root.left, coord=do_erase(self._root.left)  
    return result, coord
```

ABB. Borrar un elemento

```
def do_erase(node):
```

```
    if node is None:
```

```
        result = False
```

```
        coord = self.end()
```

```
    elif key < node.key:
```

```
        result, node.left, coord = do_erase(node.left)
```

```
    elif key > node.key:
```

```
        result, node.right, coord = do_erase(node.right)
```

```
    else: # key == node.key
```

```
        result = True
```

```
        coord = TreeDict._Coordinate(node).advance()
```

```
        node = erase_node(node)
```

```
    return result, node, coord
```

Buscamos
La clave pedida...

ABB. Borrar un elemento

```
def erase_node(node):
```

```
    parent = node.parent
```

```
    # caso 1 o caso 2
```

```
    if node.left is None:
```

```
        node = node.right
```

```
    elif node.right is None:
```

```
        node = node.left
```

```
    else:
```

```
    #caso 3
```

```
        node = extract_maximum_from(node)
```

```
    assign_parent(node, parent)
```

```
    self._len -= 1
```

```
    return node
```

Si encontramos la clave
Vemos los casos
Posibles:

ABB. Borrar un elemento

```
def extract_maximum_from(node):  
  
    prev = None  
    maximum = node.left  
    while maximum.right is not None:  
        prev = maximum #padre del max  
        maximum = maximum.right  
    assign_parent(maximum, node.parent)  
    maximum.right = node.right  
    assign_parent(maximum.right, maximum)  
    if prev is not None:  
        prev.right = maximum.left  
        assign_parent(prev.right, prev)  
        maximum.left = node.left  
        assign_parent(maximum.left, maximum)  
  
    return maximum
```

Buscamos el
Máximo del
Subárbol izquierdo

ABB. Borrar un elemento

```
def assign_parent(node, parent):  
    if node is not None:  
        node.parent = parent
```

Más operaciones...

ABB. Clear y repr

```
def clear(self):
```

```
    self._root.left = None  
    self._len = 0
```

```
def __repr__(self):
```

```
    return 'TreeDict([' + ', '.join(repr(x) for x in self.items()) + '])'
```

ABB. Copy

```
def copy(self):
```

```
    def do_copy(node, parent):
```

```
        if node is None:
```

```
            new_node = None
```

```
        else:
```

```
            new_node = TreeDict._Node(node.key, node.value, parent)
```

```
            new_node.left = do_copy(node.left, new_node)
```

```
            new_node.right = do_copy(node.right, new_node)
```

```
        return new_node
```

```
new_tree = TreeDict()
```

```
new_tree._root.left = do_copy(self._root.left, new_tree._root)
```

```
new_tree._len = self._len
```

```
return new_tree
```

¿Ayuda la recursión?

¿Cómo lo harían iterativo?

ABB. Operadores de un dict

```
# key in dict --> dict.__contains__(key)
```

```
def __contains__(self, key):  
    return self.find(key) != self.end()
```

```
# dict[key] --> dict.__getitem__(key)
```

```
def __getitem__(self, key):  
    p = self.find(key)  
    if p == self.end():  
        raise KeyError(key)  
    return p.value
```

ABB. Operadores de un dict

```
# dict[key] = value --> dict.__setitem__(key, value)
def __setitem__(self, key, value):

    self.insert(key, value)

# del dict[key] --> dict.__delitem__(key)
def __delitem__(self, key):

    success, _ = self.erase(key)
    if not success:
        raise KeyError(key)
```

Especificación Igualdad Observacional

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

ABB. Comparación

```
def __eq__(self, other):
```

```
    p = self.begin()  
    q = other.begin()
```

```
    while p != self.end() and q != other.end():  
        if p.key != q.key or p.value != q.value:  
            return False  
        p.advance()  
        q.advance()
```

```
    return p == self.end() and q == other.end()
```

Con qué criterio lo vamos
A recorrer??

Especificación. ABB

Otras Operaciones

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

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

Las implementamos
Luego de implementar
La coordenada

Recorridos

ABB. Recorridos

En-orden: (izquierdo, raíz, derecho).

Para recorrer un árbol binario no vacío en inorden (simétrico), hay que realizar las siguientes operaciones recursivamente en cada nodo:

“me invoco” el sub-árbol izquierdo

“Proceso” la raíz

“me invoco” el sub-árbol derecho

Coordenada

ABB. Coordenada

Vamos a pensar en las operaciones del TAD Coordenada para un ABB

¿Qué operaciones debe proveer la coordenada?

value acceder al elemento al que hace referencia la coordenada

Advance retorna la coordenada que referencia al siguiente elemento del árbol (en un recorrido in-orden)

Retreat retorna la coordenada que referencia al anterior elemento del árbol (en un recorrido in-orden)

== para poder comparar coordenadas.

ABB. Coordenada

```
class _Coordinate():
    __slots__ = ['_node']

def __init__(self, coordinate_or_node=None):
    if isinstance(coordinate_or_node, Treedict._Coordinate):
        self._node = coordinate_or_node._node
    else:
        self._node = coordinate_or_node
```

A igual que en la lista doble
Vamos a guardar en esta
Alternativa solo el
“nodo actual”

ABB. Coordenada

```
@property  
def key(self):  
    return self._node.key
```

```
@property  
def value(self):  
    return self._node.value
```

```
@value.setter  
def value(self, value):  
    self._node.value = value
```

No vamos a permitir
Que se modifiquen la claves
¿Estaría bien permitirlo?

ABB. Coordenada

```
def advance(self):
    node = self._node
    if node.right is not None:
        node = _minimum_node(node.right)
    else:
        while node.parent is not None:
            prev = node
            node = node.parent
            if node.right is not prev:
                break
    self._node = node
    return self
```

Busco “un padre” mayor
que él
Para posicionarse...

ABB. Coordenada

```
def retreat(self):
    node = self._node
    if node.left is not None:
        node = _maximum_node(node.left)
    else:
        while node.parent is not None:
            prev = node
            node = node.parent
            if node.left is not prev:
                break
    self._node = node
    return self
```

ABB. Coordenada

```
def next(self):  
    return TreeDict._Coordinate(self._node).advance()  
  
def prev(self):  
    return TreeDict._Coordinate(self._node).retreat()  
  
def __eq__(self, other):  
    return self._node is other._node
```

Especificación. ABB

Otras Operaciones

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

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

Ahora que ya tenemos la coordenada
las podemos implementar

ABB. Begin y end

```
def begin(self):  
    return TreeDict._Coordinate(_minimum_node(self._root))  
  
def end(self):  
    return TreeDict._Coordinate(self._root)
```

Coordenada Liviana y Pesada

ABB. Coordenadas

Vamos a llamar a la coordenada anterior **Liviana**.

-En el árbol cada nodo conoce a su padre

-De esta manera la coordenada no necesita guardar información para retornar.

Otra alternativa sería:

La coordenada "tiene la información necesaria para volver a la raíz desde el nodo donde está parado. Para ello podemos usar una pila.

A esta coordenada la vamos a llamar **Pesada**



¿Qué pasaría en ese caso si el árbol cambia?

ABB. Coordenada Pesada

La especificación de las operaciones no cambia!

Y en la implementación de la coordenada...

¿Qué deberíamos cambiar???

**Y si queremos iterar
nuestro ABB...**

ABB. Iterable

Un **iterable** es un objeto para el cual la **función `iter()`** devuelve un **iterador** que permite **recorrerlo secuencialmente**, pasando **una sola vez** por cada uno de los elementos sobre los que itera.

ABB. Iterable

A un **iterador** se le puede hacer dos cosas:

- Aplicarle la función **next()** para obtener el próximo elemento de la secuencia. Si no hubiera más elementos, levanta una excepción de tipo **StopIteration**.
- Aplicarle la función **iter()**, que en este caso devuelve el mismo iterador dado.

Lo último implica que un **iterador** es a la vez un objeto **iterable**!

ABB. Iterable

Un **iterable** debe implementar el método `__iter__()`, que es llamado por `iter()`, para que devuelva un iterador sobre él.

Un **iterador** debe implementar el método `__iter__()` para que se retorne a sí mismo y el método `__next__()` para que retorne el siguiente valor o levante **StopIteration**.

Generadores en Python

Un **generador** es un mecanismo que permite escribir **una función** que se comporte como **un iterable**.

Un generador **permite hacer más cosas que las que vamos a ver ahora**.

Veremos sólo el aspecto de los generadores **relacionado con la iteración**.

Generadores para iterar ABB

No vamos a implementar en este caso una clase para el iterador.

Un generador es un iterable y a su vez es un iterador sobre él mismo. Es un objeto que produce valores por demanda al iterarlo.

No va a hacer falta que implementemos **next** ya que **el generador lo tiene implementado internamente.**

Generadores. yield

Un generador se define como una función, excepto que contiene la sentencia yield dentro del bloque de sentencias.

Cuando **se invoca al generador**, éste no comienza a ejecutar sus sentencias sino que “automáticamente” se retorna un **objeto generador iterable** listo para ser usado.

Cuando se le haga un **next()**, comenzará a ejecutarse el cuerpo del generador hasta encontrar la sentencia **yield**.

Generadores. yield

yield permite interrumpir la ejecución del generador y retornar un valor como si se tratara de un iterador.

Si se le pide el próximo valor con **next()**, el generador reanudará su ejecución luego del **yield** como si no se hubiera interrumpido.

Cuando el generador termina, se producirá un **StopIteration** con el siguiente **next()**.

Generadores

Visualizando la ejecución del generador:

```
>>> c = cuenta_regresiva(5)
>>> next(c)
5
>>> next(c)
4
>>>
```

Sólo se crea el objeto generador, aún no se ejecuta el código del generador.

Esqueleto general:

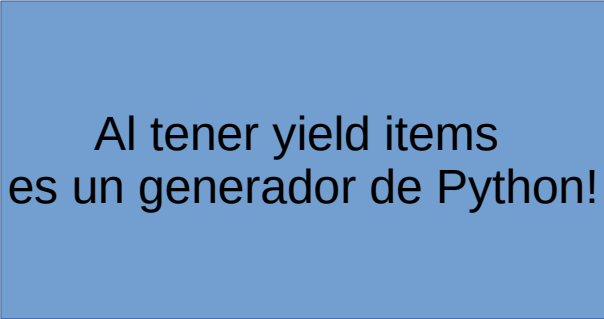
Devuelvo el siguiente
Me duermo...

```
def cuenta_regresiva(n):
    while n >= 0:
        yield n
        n -= 1
```

ABB iterable

```
def items(self):  
    pos = self.begin()  
    end = self.end()  
    while pos != end:  
        yield pos.key, pos.value  
        pos.advance()
```

```
def values(self):  
    for _, value in self.items():  
        yield value
```



Al tener yield items
es un generador de Python!

AVL

Árboles

Balanceados

AVL

Para que los árboles binarios puedan ser considerados de búsqueda deberían ser “balanceados”

Se entiende por **árbol balanceado (AVL)** aquel donde la diferencia entre el camino más largo y el más corto desde la raíz a las hojas es “EL MISMO” o al menos un valor acotado.

Los árboles binarios de búsqueda no tienen por que estar balanceados

AVL

Se debe analizar algún tipo de procedimiento que me asegure que los árboles se mantienen balanceados

Elegir “bien” la raíz. Como hacerlo?

Poner todo en “otra estructura” y luego pasarlo a un árbol? ????

Construir el árbol y cuando se desbalancea reacomodar?

Construir el árbol de otra forma de manera que quede siempre balanceado?

AVL

Al tener un AVL tenemos máxima eficiencia en la búsqueda :-)

¿Cuál es la desventaja? :-)

La propiedad de equilibrio de los árboles AVL implica una *dificultad a la hora de insertar o eliminar* elementos:

estas operaciones pueden no conservar dicha propiedad.

AVL

En el siguiente capítulo
AVL'S...

