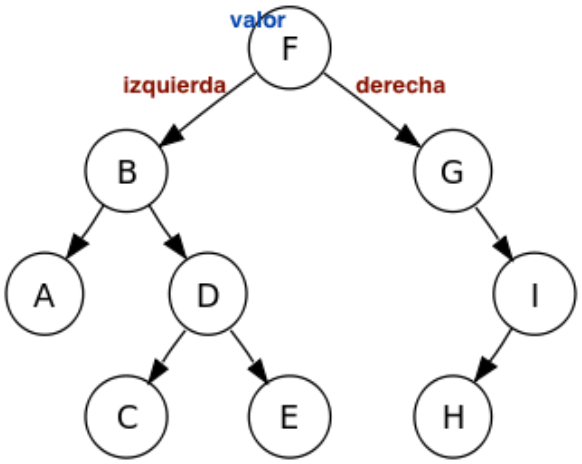
Clase 13: AB y ABB (Árbol de Búsqueda Binaria)

(Capítulo 9.9 del Apunte) Versión corregida 2020-11-13.

Recuerdo Árboles Binarios (AB)



Cumplen alguna de las siguientes condiciones:

- Un AB es vacío, o
- Tiene un valor, un AB a la izquierda, y otro AB a la derecha

Creación de un AB

Un AB es una estructura recursiva pues un **AB** contiene un valor (llamado raíz) y dos **AB**s adentro!

AB = **valor** + **AB izquierdo** + **AB derecho**

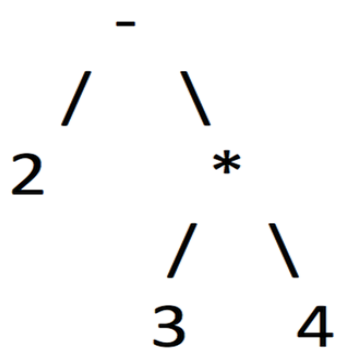
```
import estructura


#Arbol Binario
#AB: valor(any), izq(AB), der(AB)
estructura.crear("AB","valor izq der")


# creamos un AB

ab=AB("F",\
    AB("B",\
        AB("A",None,None),\
        AB("D",\
            AB("C",None,None),\
```

```
            AB("E",None,None))),\
    AB("G",\
        None,\
        AB("I",\
            AB("H", None, None),\
            None)))


print(ab)


print(ab.valor)
print(ab.izq)
print(ab.der)


print(ab.izq.valor) # dentro del sub-arbol izquierdo: la raiz
print(ab.izq.izq) # dentro del sub-arbol izquierdo: su sub-arbol izquierdo
print(ab.izq.der) # dentro del sub-arbol izquierdo: su sub-arbol izquierdo
```

Vimos tareas típicas sobre un AB

- Crear un AB
- Contar los nodos de un AB
- Calcular la altura de un AB
- Contar las hojas de un AB

Un tipo de Árbol Binario: **Un AB de expresiones aritméticas**




```
#arbol expresión
#AB con numeros en los valores de las hojas
#   y operadores en los valores de las no-hojas


ae=AB("-", \
      AB(2,None,None),\
      AB("*",\
         AB(3,None,None),\
```

```
        AB(4,None,None)))


#evaluar: AB -> num
#evalua expresión representada en AB
#ej: evaluar(ae) -> -10
def evaluar(A):

    assert type(A)==AB

    if A.izq==None and A.der==None:
        return A.valor

    a=evaluar(A.izq) #primer operando
    b=evaluar(A.der) #segundo operando
    op=A.valor       #operador


    if op=="+":
        return a+b
    if op=="-":
        return a-b
    if op=="*":
        return a*b
    if op=="/":
        return a/b

assert evaluar(ae)==-10



otroAE = AB("*",AB("+",AB(3,None,None),AB(4,None,None)),AB("/",AB(10,None,None),AB(5,N
print(otroAE)


print(evaluar(otroAE))
```

Propuesto 1:: Modificar la función anterior para que permita evaluar AB que representen expresiones aritméticas con exponenciación, como por ejemplo: `5+2**3`

En otras palabras, si

```
ae2 = AB("+",AB(5,None,None),AB("**",AB(2,None,None),AB(3,None,None)))
```

entonces

```
evaluar(ae2)
```

retorna 13.

Propuesto 2:: Modificar la función anterior para que permita evaluar AB que representen expresiones aritméticas con números negativos, como por ejemplo:

`7+2*-3`

En otras palabras, si

```
ae3 = AB("+",AB(5,None,None),AB("*",AB(2,None,None),AB("-",None,AB(3,None,None))))
```

entonces

```
evaluar(ae3)
```

retorna 1.

Pista: Fijarse que el árbol de expresión aritmético para `-3` es el árbol para `0-3` pero reemplazando el subárbol con la hoja `0` por un `None`. Entonces, para evaluar árboles con negativos, sólo tiene que modificar la función para detectar si un árbol con "-" en la raíz representa (a) una resta (caso ya listo) o (b) un número negativo.

ABB: Árbol de Búsqueda Binaria

Un ABB es un árbol binario tal que:

- es un árbol vacío (None)

o sino,

- valores en el AB **izquierdo** son **menores** que el valor
- valores en el AB **derecho** son **mayores** que el valor
- AB izquierdo y AB derecho son también ABB


```
import estructura
#Arbol Binario
#AB: valor(any), izq(AB), der(AB)
estructura.crear("AB","valor izq der")

abb=AB("F",\
    AB("B",\
        AB("A",None,None),\
        AB("D",\
```

```
            AB("C",None,None),\
            AB("E",None,None))),\
    AB("G",\
        None,\
        AB("I",\
            AB("H", None, None),\
            None)))
```

Cómo encontrar un valor en un ABB

```
#buscaValor: any AB -> bool
#True si x está en arbol
#ej: buscaValor("A",abb)->True

def buscaValor(x,arbol):

    assert arbol==None or type(arbol)==AB
    if arbol==None:
        return False

    elif arbol.valor==x:
        return True
    elif x < arbol.valor:
        return buscaValor(x,arbol.izq)
    elif x > arbol.valor:
        return buscaValor(x,arbol.der)

assert buscaValor("A",abb)
```

Encontrar el mayor y el menor elemento en un ABB

```
#mayorABB: AB(any) -> any
#mayor valor de ABB A
#ej: mayorABB(abb)->"C“

def mayorABB(A):
    assert type(A)==AB #por lo menos un valor
    if A.der==None: return A.valor
    return mayorABB(A.der)

#Tests
abb2=AB("B",AB("A",None,None),AB("C",None,None))
assert mayorABB(abb2)=="C"
```

```
#menorABB: AB(any) -> any
#mayor valor de ABB A
#ej: menorABB(abb)->"C“

def menorABB(A):
    assert type(A)==AB #por lo menos un valor
    if A.izq==None: return A.valor
    return menorABB(A.izq)

#Tests
abb2=AB("B",AB("A",None,None),AB("C",None,None))
assert menorABB(abb2)=="A"
```

Ejemplo: Cómo saber si un AB es un ABB?

Es decir, cómo sabemos si un árbol binario cumple con las condiciones para ser un árbol de búsqueda binario?

```
#esABB: AB -> bool
#True si AB es un ABB
#ej: esABB(abb)->True

def esABB(arbol):

    assert arbol==None or type(arbol)==AB
    if arbol==None:
        return True

    # debemos revisar varias condiciones:
    # (1) que el AB izquierdo sea ABB,
    # (2) que el AB derecho sea ABB,
    # (3) si AB izquierdo es vacio, entonces está ok.
    #     Si no es vacio, que el mayor elemento del AB izquierdo sea < valor en la rai
    # (4) si AB derecho es vacio, entonces está ok.
    #     Si no es vacio, que el menor elemento del AB derecho sea > valor en la raiz.
    # Para hacerlo más simple, usaremos variables booleanas
    condicion1 = esABB(arbol.izq) # retorna un booleano
    condicion2 = esABB(arbol.der) # retorna un booleano
    condicion3 = (arbol.izq == None) or (mayorABB(arbol.izq) < arbol.valor) # retorna
    condicion4 = (arbol.der == None) or (menorABB(arbol.der) > arbol.valor) # retorna

    if  condicion1 and condicion2 and condicion3 and condicion4:
        return True
    else:
        return False

assert esABB(abb)
assert not esABB(AB("B",AB("C",None,AB("A",None,None)),None))
```

```
noesabb=AB("F",\
    AB("B",\
        AB("A",None,None),\
        AB("D",\
            AB("C",None,None),\
            AB("E",None,None))),\
    AB("I",\
        None,\
        AB("K",\
            AB("G", None, None),\
            None)))
# Ojo, en arbol noesabb: G es menor que I, pero está en el subárbol derecho de I, por
assert not esABB(noesabb)
siesabb = AB("F",\
    AB("B",\
        AB("A",None,None),\
        AB("D",\
            AB("C",None,None),\
            AB("E",None,None))),\
    AB("I",\
        None,\
        AB("K",\
            AB("J", None, None),\
            None)))
assert esABB(siesabb)



#escribir: AB -> None
#escribir valores de ABB A en orden ...
#ej: escribir(abb) -> ...

def escribir(arbol):
    assert arbol==None or type(arbol)==AB
    if arbol==None:
        return

    escribir(arbol.izq)
    print(arbol.valor)
    escribir(arbol.der)




escribir(abb)


#insertar: any, AB -> AB
#nuevo ABB insertando x en ABB A
#ej: insertar("A",AB("B",None,None))->
#                 AB("B",AB("A",None,None),None)
def insertar(x,arbol):
```

```
def insertar(x,arbol):

    assert arbol==None or type(arbol)==AB
    if arbol==None:
        return AB(x,None,None)

    assert x!=arbol.valor
    if x<arbol.valor:
        return AB(arbol.valor, insertar(x,arbol.izq), arbol.der)
    if x>arbol.valor:
        return AB(arbol.valor, arbol.izq, insertar(x,arbol.der) )

assert insertar("A",AB("B",None,None))== \
              AB("B",AB("A",None,None),None)
```