Practica 3

Árboles Binarios

Ejercicio 1: Utilizando la clase Tree presentada a continuación

```
   class Tree:
     def __init__(self, cargo, left = None, right = None):
       self.cargo = cargo
       self.left = left
       self.right = right
```

Implemente los siguiente métodos:

Ayuda: pensar que cada árbol tiene a su izquierda y derecha objetos árboles como sus hijos **.**

- **nodos: devuelve la cantidad de nodos del árbol**
- **menor_mayor: devuelve el menor y el mayor elemento del árbol en una tupla**
- **buscar: busca si un elemento está o no en el árbol**
- **altura: calcula la altura del árbol, la distancia desde la raíz hasta la hoja más lejana**

In [1]:

```
class Tree:
  def __init__(self, cargo, left=None, right=None):
    self.cargo = cargo
    self.left = left
    self.right = right

  def nodos(self):
    ...

  def menor_mayor(self):
    ...

  def buscar(self, element):
    ...

  def altura(self):
    ...



#          ___8___
#        /        \
#       2          15
#     /  \        /  \
#    4   -3     17    20
#  / \               /  \
# 7   3            22    0
tree = Tree(8, Tree(2, Tree(4,Tree(7), Tree(3)), Tree(-3)), Tree(15, Tree(17), Tree(20,T
ree(22),Tree(0))))
```

Ejercicio 2:

a. Implementar los recorridos PreOrder, InOrder y PostOrder como funciones recursivas

b. Implementar los recorridos PreOrder, InOrder y PostOrder como funciones iterativas.

***Ayuda*: Para las versiones iterativas, necesitará utilizar una Pila como estructura de datos adicional. Puede importar una implementacion cualquiera de Pila que haya realizado en la Practica anterior.**

In [3]:

```
# Version recursiva

def treePreOrder(tree):
  ...
def treeInOrder(tree):
  ...
def treePostOrder(tree):
  ...

tree = Tree(8, Tree(4, Tree(2,Tree(1), Tree(3)), Tree(7)), Tree(15, Tree(17), Tree(20,Tr
ee(19),Tree(22))))
#          ___8___
#        /        \
#       4          15
#     /  \        /  \
#    2    7     17    20
#   / \              /  \
#  1   3           19    22
```

In [4]:

```
# Version iterativa
def treePreOrder(tree):
  ...


def treeInOrder(tree):
  ...

def treePostOrder(tree):
  ...

tree = Tree(8, Tree(4, Tree(2,Tree(1), Tree(3)), Tree(7)), Tree(15, Tree(17), Tree(20,Tr
ee(19),Tree(22))))
#          ___8___
#        /        \
#       4          15
#     /  \        /  \
#    2    7     17    20
#   / \              /  \
#  1   3           19    22

treePreOrder(tree)
print("")

treeInOrder(tree)
print("")

treePostOrder(tree)
print("")
```

Ejercicio 3: En la práctica anterior mencionamos cómo utilizamos notación postfija de expresiones para evaluar en un Stack y sin usar paréntesis. Con árboles podemos representar expresiones infijas sin paréntesis. Cada nodo interno del árbol representa un operador, izquierda y derecha son subexpresiones, y las hojas son números. Implementar una clase Expression que herede de Tree, un árbol de expresiones infijas, con dos métodos.

- **imprimir: que imprime la expresión de forma infija con paréntesis.**
- **evaluar: evalúa todo el árbol y lo reduce a un número.**

Ejemplo

```
        *
      /   \
     +     -      ==>    ((1 + 2) * (3 - 4))    ==>    -2
```

```
    / \   / \
   1   2 3   4
```

In [5]:

```
class Expression(Tree):
  def imprimir(self):
    ...

  def evaluar(self):
    ...

node1 = Expression(1)
node2 = Expression(2)
node3 = Expression(3)
node4 = Expression(4)

suma = Expression("+", node1, node2)
resta = Expression("-", node3, node4)
mult = Expression("*", suma, resta)

mult.imprimir()
print("")
mult.evaluar()
```

Árboles Binarios de Búsqueda

Ejercicio 4: Utilizando la misma clase Tree de la sección anterior, implemente otra clase llamada BSTree que herede de esta, reimplemente los métodos menor_mayor, buscar e implemente un nuevo método llamado insertar que inserte un elemento.

In [7]:

```
class BSTree(Tree):

  def menor_mayor(self):
    ...

  def buscar(self, element):
    ...

  def insert(self, element):
    ...

import random
btree = BSTree(10)
for i in range(10):
  btree.insert(random.randint(0, 20))
```

Ejercicio 5:

La magia de IPython es un sistema de comandos mágicos que sirven para realizar diversas tareas del sistema operativo directamente en un entorno que use a iPython, como Google Colab, IPython, Jupyter Notebooks, etc.

Uno de estos comandos magicos es `%%timeit`, el cual sirve para medir cuando tiempo tarda en ejecutarse un bloque de codigo. Este comando corre las celdas muchas veces y calcula el tiempo promedio de ejecucion, para asegurarse de que la medicion sea confiable.

En la siguientes dos celdas de código se realizan dos experimentos:

1. **Por un lado, se insertan los numeros del 1 al 100 en un BSTree en orden creciente, y luego, se chequea si cada uno esta en el arbol.**
2. **En otra celda, se insertan los numeros del 1 al 100 en un BSTree en orden aleatorio, y luego, se chequea si cada uno esta en el arbo.**

Corra los experimentos utilizando Google Colab o IPython y explique porque uno de los dos tarda menos que el otro.

In [ ]:

```
%%timeit
# Insertamos los numeros del 1 al 100000 en un BSTree en orden creciente
t = BSTree(1)
for i in range(100):
  t.insert(i)

# Y luego chequeamos si cada numero fue insertado
for i in range(100):
  t.buscar(i)
```

In [ ]:

```
%%timeit
# Insertamos los numeros del 1 al 100000 en un BSTree en orden aleatorio
import random
t = BSTree(1)
L = list(range(100))
random.shuffle(L)
for i in L:
  t.insert(i)

# Y luego chequeamos si cada numero fue insertado
for i in range(100):
  t.buscar(i)
```

Ejercicio 6 Escriba una funcion `copy` que reciba un BSTree y devuelva un *nuevo* BSTree identico al original.

In [ ]:

```
def copiar(arbol):
  ...

import random
btree = BSTree(10)
for i in range(10):
  btree.insert(random.randint(0, 20))

btree2 = copiar(btree)
treeInOrder(btree)
print("")
treeInOrder(btree2)
```

```
0 8 10 11 13 14 15 16 18
0 8 10 11 13 14 15 16 18
```

Ejercicio 7: Escriba una funcion `combine` que combine dos arboles binarios de búsqueda en uno solo. El resultado tambien debe ser un arbol binario de busqueda.

In [ ]:

```
def combinar(arbol1, arbol2):
  ...

import random
btree = BSTree(10)
for i in range(10):
  btree.insert(random.randint(0, 40))

btree2 = BSTree(30)
for i in range(10):
  btree2.insert(random.randint(0, 40))


treeInOrder(btree)
```

```
print("")
treeInOrder(btree2)
print("")
treeInOrder(combinar(btree, btree2))
```

```
5 9 10 16 20 21 22 23 34 35
3 4 5 20 23 26 30 37
3 4 5 9 10 16 20 21 22 23 26 30 34 35 37
```

Ejercicio 8 Escriba una función `borrar_raiz`. Dado un arbol binario de busqueda, esta funcion deberia devolver un nuevo arbol binario de busqueda que contenga los mismos datos, a excepcion de la raiz.

In [ ]:

```
def borrar_raiz(arbol):
  ...

import random
btree = BSTree(20)
for i in range(10):
  btree.insert(random.randint(0, 40))

treeInOrder(borrar_raiz(btree))
```

```
2 8 24 25 30 32 35 36 38
```

Ejercicio 9 Escriba una funcion `borrar_valor` que dado un arbol binario de busqueda y un valor, devuelva un arbol binario de busqueda sin ese valor.

In [ ]:

```
def borrar_valor(arbol, valor):
  ...



import random
btree = BSTree(20)
btree.insert(10)


treeInOrder(btree)
print("")
btree = borrar_valor(btree, 10)
treeInOrder(btree)
```

```
10 20
20
```

Ejercicio 10: Herede de BSTree una clase Diccionario, cuyos valores sean tuplas (clave, valor). Reimplemente los métodos buscar e insertar para que acepten estas tuplas, guarden de acuerdo a sus claves que son strings, y busquen el valor asociado a la clave pasada como argumento, como un diccionario.

In [9]:

```
class Diccionario(BSTree):
  def buscar(self, clave):
    ...

  def insert(self, clave, valor):
    ...

btree = Diccionario(("Nicolas", 26))
btree.insert("Hernan", 18)
btree.insert("Brisa", 18)
btree.insert("Sofia", 40)

btree.buscar("Sofia")
```