# Classe Fila, necessária para a busca em largura

class Fila:

#Construtor da classe Fila

def \_\_init\_\_(self):

self.ítens = []

self.inícioFila = 0

#Método que volta o início da fila para 0, quando ela se afastar

#muito do início

def \_\_comprime(self):

novalista = []

for i in range(self.inícioFila,len(self.ítens)):

novalista.append(self.ítens[i])

self.ítens = novalista

self.inícioFila = 0

#Método de remoção de um elemento da fila

def desenfileira(self):

if self.éVazia():

raise RuntimeError("Tentativa de desenfileirar de uma fila vazia")

if self.inícioFila \* 2 > len(self.ítens):

self.\_\_comprime()

item = self.ítens[self.inícioFila]

self.inícioFila += 1

return item

#Método de inserção de um elemento na fila

def enfileira(self, item):

self.ítens.append(item)

#Método que apenas olha o início da fila sem remover

def front(self):

if self.éVazia():

raise RuntimeError("Tentativa de acessar uma fila vazia")

return self.ítens[self.inícioFila]

#Método que verifica se uma fila está vazia

def éVazia(self):

return self.inícioFila == len(self.ítens)

#Classe que implementa uma árvore binária de busca

class ÁrvoreBináriadeBusca:

#Classe interna que representa um nó de uma árvore binária de busca

class \_\_Nó:

#Construtor para o nó, criando um nó com filhos vazios por padrão

def \_\_init\_\_(self,val,esq=None,dir=None):

self.val = val

self.esq = esq

self.dir = dir

#Método getter para valor do nó

def getVal(self):

return self.val

#Método setter para valor do nó

def setVal(self, novoval):

self.val = novoval

#Método getter para filho esquerdo do nó

def getEsq(self):

return self.esq

#Método getter para filho direito do nó

def getDir(self):

return self.dir

#Método setter para filho esquerdo do nó

def setEsq(self, novoesq):

self.esq = novoesq

#Método getter parra filho direito do nó

def setDir(self, novodir):

self.dir = novodir

#Método que define um iterador em percurso simétrico para o nó e seus filhos

def \_\_iter\_\_(self):

if self.esq != None:

for elem in self.esq:

yield elem

yield self.val

if self.dir != None:

for elem in self.dir:

yield elem

#Construtor para o objeto ÁrvoreBináriadeBusca, definindo a raiz como vazia

def \_\_init\_\_(self):

self.raiz = None

#Método de inserção de um elemento na árvore

def insere(self,val):

#Função recursica interna que de fato insere um elemento na árvore

def \_\_insere(raiz,val):

if raiz == None:

return ÁrvoreBináriadeBusca.\_\_Nó(val)

if val < raiz.getVal():

raiz.setEsq(\_\_insere(raiz.getEsq(), val))

else:

raiz.setDir(\_\_insere(raiz.getDir(), val))

return raiz

#Chamada ao método interno começando pela raiz

self.raiz = \_\_insere(self.raiz, val)

#Método que usa o iterador do nó para criar um iterador para a classe árvore

def \_\_iter\_\_(self):

if self.raiz != None:

return self.raiz.\_\_iter\_\_()

else:

return [].\_\_iter\_\_()

#Método que retorna a quantidade de elementos em uma árvore

#Usada pela questão 3

def elementos(self):

#Função recursiva que conta os nós de uma subárvore dada

def \_\_elementos(raiz):

qtd = 1 #O nó raiz conta como um elemento

if raiz.getEsq() != None:

qtd += \_\_elementos(raiz.getEsq()) #A quantidade de elementos da subárvore esquerda

if raiz.getDir() != None:

qtd += \_\_elementos(raiz.getDir()) #A quantidade de elementos da subárvore direita

return qtd

if self.raiz == None:

return 0 #Árvore vazia não possui elementos

else:

return \_\_elementos(self.raiz) #Retorna os elementos começando pela raiz

#Método semelhante ao anterior que computa a altura de uma árvore

#Usada pela questão 3

def altura(self):

#Função interna que dado um nó computa a altura desse nó

def \_\_altura(raiz):

alt\_esq = 0

alt\_dir = 0

if raiz.getEsq() != None:

#Calcula a altura do filho esquerdo

alt\_esq = \_\_altura(raiz.getEsq())

if raiz.getDir() != None:

#Calcula a altura do filho direito

alt\_dir = \_\_altura(raiz.getDir())

return max(alt\_esq, alt\_dir) + 1

if self.raiz == None:

return 0 #Árvore vazia tem altura 0

else:

return \_\_altura(self.raiz) #Retorna a altura da raiz da árvore

#Método que responde à questão 3, retorna True se a árvore for cheia

#E false caso contrário

def éCheia(self):

alt = self.altura() #Altura da árvore

elems = self.elementos() #Quantidade de elementos

#Uma árvore é cheia se todo nó com algum filho vazio está no último nível

#Isso significa que apenas árvores com 2^k - 1 nós podem ser cheias, para k > 1

#Se a altura da árvore for k temos então que a árvore é cheia

if elems == 2 \*\* alt - 1:

return True

return False

#Método que faz uma busca em largura, respondendo à questão 1

def buscaLargura(self):

#Cria uma fila necessária para a busca

fila = Fila()

#O nível é 1 para a raiz

nível = 1

if self.raiz != None:

#Se a raiz não for vazia, enfileira uma tupla com a raiz e seu nível

fila.enfileira((self.raiz, nível))

#Enquando a fila não estiver vazia

while not fila.éVazia():

#Retira o elemento da fila, junto com seu nível

(nó, nível) = fila.desenfileira()

#Enfileira o filho esquerdo, caso exista

#O nível desse filho é um a mais que seu pai

if nó.getEsq() != None:

fila.enfileira((nó.getEsq(), nível+1))

#Mesma coisa com o filho direito

if nó.getDir() != None:

fila.enfileira((nó.getDir(), nível+1))

#Imprime o valor e o nível do nó

#Como os filhos são inseridos em bloco, são retirados nessa mesma ordem

#Garantindo que os níveis serão impressos sempre em ordem crescente

#Da esquerda para a direita

print(f"Nível: {nível} Valor: {nó.val}" )

#Método que retorna uma árvore espelhada, trocando esquerda com direita

#Respondendo a questão 4

def espelho(self):

#Função recursiva interna que clona o nó e clona os filhos invertendo a ordem

def \_\_cloneEspelho(raiz):

#Quando se cria uma nó, os filhos começam por default vazios

novo = ÁrvoreBináriadeBusca.\_\_Nó(raiz.val)

#Aqui se clona o filho esquerdo, jogando para o filho direito do nó novo

if raiz.getEsq() != None:

novo.setDir(\_\_cloneEspelho(raiz.getEsq()))

#Aqui se faz o mesmo com o filho direito

if raiz.getDir() != None:

novo.setEsq(\_\_cloneEspelho(raiz.getDir()))

return novo #Retorna um nó clonado e invertido

#Cria a nova Árvore

nova = ÁrvoreBináriadeBusca()

nova.raiz = \_\_cloneEspelho(self.raiz) #Clona espelhado a raiz da árvore antiga para a nova

return nova

def main():

#Entrada para uma árvore zigzag degenerada

s = "1 7 2 6 3 5 4"

lst = s.split()

árvore = ÁrvoreBináriadeBusca()

for x in lst:

árvore.insere(float(x))

print("Uma árvore zigzag")

for x in árvore:

print(x)

print("Número de elementos: ", árvore.elementos())

print("Altura", árvore.altura())

if árvore.éCheia():

print("A árvore é cheia")

else:

print("A árvore não é cheia")

árvore.buscaLargura()

print("Espelhando a árvore")

esp = árvore.espelho()

esp.buscaLargura()

#Entrada para uma árvore cheia

s = "4 2 6 7 5 3 1"

lst = s.split()

árvore = ÁrvoreBináriadeBusca()

for x in lst:

árvore.insere(float(x))

print("Uma árvore cheia")

for x in árvore:

print(x)

print("Número de elementos: ", árvore.elementos())

print("Altura", árvore.altura())

if árvore.éCheia():

print("A árvore é cheia")

else:

print("A árvore não é cheia")

árvore.buscaLargura()

print("Espelhando a árvore")

esp = árvore.espelho()

esp.buscaLargura()

#Entrada para uma árvore completa mas não cheia

s = "4 2 6 7 5 3 1 8"

lst = s.split()

árvore = ÁrvoreBináriadeBusca()

for x in lst:

árvore.insere(float(x))

print("Uma árvore completa")

for x in árvore:

print(x)

print("Número de elementos: ", árvore.elementos())

print("Altura", árvore.altura())

if árvore.éCheia():

print("A árvore é cheia")

else:

print("A árvore não é cheia")

árvore.buscaLargura()

print("Espelhando a árvore")

esp = árvore.espelho()

esp.buscaLargura()

if \_\_name\_\_ == "\_\_main\_\_":

main()