Índice

[**CLASE PILA**](#_bbjmscgiblab) **2**

[PILA CON LISTA](#_2pa35sk6odgv) 3

[PILA CON NODOS](#_pmw85282b20m) 5

[**CLASE COLA**](#_1ul7oc9oyrg0) **8**

[COLA CON LISTA](#_nmyoyojjubeb) 9

[**CLASE COLA DOBLEMENTE TERMINADA**](#_69f1jsl7lu9k) **10**

[COLA DOBLE CON NODOS](#_oleylbbaz8i6) 11

[**CLASE ARREGLO DINÁMICO**](#_qi7mopq58d5i) **14**

[**CLASE CONJUNTO**](#_oqlhk2o9jukp) **15**

[CONJUNTO](#_r56j74ld2w06) 16

[**CLASE LISTA ENLAZADA SIMPLE**](#_tmqdlmce90y) **18**

[LISTA SIMPLE LISTA](#_oy9dvejzoxpy) 19

[LISTA SIMPLE CON NODO](#_rbhklisjix) 22

[**CLASE LISTA DOBLEMENTE ENLAZADA**](#_nd3ox848j3n3) **25**

[LISTA DOBLE](#_d9m0uiyogmox) 26

[LISTA DOBLE CON NODO](#_jcoxxabuz106) 28

[LISTA CIRCULAR](#_5pp278i6nzo) 30

[LISTA CIRCULAR DOBLE](#_kc55ijfsdppb) 32

[**CLASE COORDENADA**](#_eqb6mniq2vnw) **35**

[COORDENADA](#_44awz1xc1eax) 36

[**CLASE ARBOL BINARIO DE BÚSQUEDA**](#_vtj34qdzpkjl) **37**

[ABB](#_4lzh3zu3m508) 38

[AVL](#_oryb84thm0wp) 52

[**CLASE FECHA**](#_u9jemgd31ehv) **68**

[FECHA](#_s2irenqsu16n) 70

[**CLASE GRAFO**](#_xsf32zmquj7y) **72**

[GRAFO CON MATRIZ](#_s9v4ytn22g7f) 73

# 

# 

# CLASE PILA

TAD Nombre\_del\_TAD

Pila<a>

Igualdad Observacional:

Si a y b son dos pilas

a es igual a b si se cumple que:

Las longitudes de a y b son iguales

Y

cada elemento en a es igual al correspondiente

elemento en b.

Usa

Natural ; Bool ; None ; <a> ; secuencia<a>

Parámetro Formal

<a>

Géneros

Pila<a>

observadores básicos

tamaño(Pila<a> P) → Natural

es\_vacia(Pila<a> P) → Bool

tope(Pila<a> P)→ a {pre: la pila tiene al menos un elemento}

Generadores

nueva\_pila()→ None

a\_partir(secuencia<a> P)→ a

otras operaciones

apilar(Pila<a> P, a)→ None {post:}

desapilar(Pila<a> P, a)→ None {pre: la pila tiene al menos un elemento}

Axiomas

tamaño(Pila<a> P){'devuelve un entero con la cantidad de elementos en la pila P'}

es\_vacia(Pila<a> P){'true -si la pila P esta vacia, false -si la pila P no esta vacia'}

tope(Pila<a> P){'devuelve el elemento que se encuentra en el tope de la pila P'}

nueva\_pila(){'crea una pila vacia')

a\_partir(secuencia<a> P){'crea una pila 'P' apartir de los elementos de la secuencia'}

apilar(Pila<a> P, a elemen){'agrega un elemento 'elemen' al tope de la pila 'P''}

desapilar(Pila<a> P){'elimina el elemento que se encuentra al tope de la pila 'P''}

Exporta

Pila<a> ; tamaño ; es\_vacia ; tope ; nueva\_pila ;

a\_partir() ; apilar ; desapilar

## PILA CON LISTA

class Pila() :

# todas las fuciones son de o(1)

# escepto la funcion \_\_repr\_\_

# el metodo length es orden constante tmb pero sin

# usar el metodo len de python

\_\_slots\_\_ = ['\_values','\_count']

def \_\_init\_\_(self, iterable=None):

self.\_values = []

self.\_count = 0

if iterable is not None:

for value in iterable:

self.push(value)

def empty(self):

return len(self.\_values) == 0

@property

def top(self):

assert not self.empty(), 'La pila esta vacia'

return self.\_values[-1]

def clear(self):

self.\_values.clear()

def push(self, value):

self.\_values.append(value)

self.\_count += 1

def pop(self):

assert not self.empty(), 'La pila esta vacia'

self.\_count -= 1

return self.\_values.pop()

def copy(self):

nueva\_pila = Pila()

nueva\_pila.\_values = self.\_values.copy()

return nueva\_pila

def copy\_recursivo(self):

other = Pila()

aux = Pila(self.\_values)

def copy(aux,other):

if aux.empty():

return other

else:

top = aux.pop()

copy(aux,other)

other.push(top)

copy(aux,other)

return other

def length(self):

return self.\_count

def \_\_len\_\_(self):

return len(self.\_values)

def len\_recursivo(self):

aux = Pila(self.\_values)

def lenR(aux):

if aux.empty():

return 0

else:

aux.pop()

return 1 + lenR(aux)

count = lenR(aux)

return count

def \_\_eq\_\_(self, other):

return self.\_values == other.\_values

def \_\_repr\_\_(self):

return 'Pila (['+", ".join(repr(i) for i in self.\_values) + "])"

## PILA CON NODOS

from dataclasses import dataclass

class Stack():

@dataclass

class \_node():

\_value=any

\_next:"\_node"=None

\_\_slots\_\_=["\_top","\_inicio","\_length"]

def \_\_init\_\_(self,iterable=None):

self.\_length=0

self.\_top=None

if iterable is not None:

for values in iterable:

self.push(values)

def apilar(self,values):

if self.\_top==None:

nuevo=Stack.\_node()

nuevo.\_value=values

nuevo.\_next=None

self.\_inicio=nuevo

self.\_top=nuevo

self.\_length+=1

else:

nuevo2=Stack.\_node()

nuevo2.\_value=values

nuevo2.\_next=None

self.\_top.\_next=nuevo2

self.\_top=nuevo2

self.\_length+=1

def \_\_len\_\_(self):

return self.\_length

def clear(self):

self.\_top=None

self.\_length=0

def is\_empty(self):

return self.\_top is None

def desapilar(self):

assert not self.is\_empty(),'sin elementos'

value= self.\_top.\_value

self.\_top=self.\_top.\_next

self.\_length-=1

return value

def recorrer(self):

current=self.\_inicio

while(current!=None):

print(current.\_value)

current=current.\_next

def top(self):

assert not self.is\_empty(),'Sin elementos'

return self.\_top.\_value

def copy(self):

n=Stack()

if not self.is\_empty():

node=self.\_top

new\_node=Stack.\_node(node.\_value,None)

n.\_top=new\_node

while node.\_next is not None:

node=node.\_next

new\_node.\_next=Stack.\_node(node.\_value,None)

new\_node=new\_node.\_next

return n

def lenRecursivo(self):

def lenR(nodo):

if nodo.\_next==None:

return 0

else:

return 1+ lenR(nodo.\_next)

return lenR(self.\_inicio)

def lenIterativo(sef):

n=0

node=self.\_top

while node is not None:

node=node.\_next

n+=1

return n

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

def equal\_recursivo(self,other):

def do\_equal(x,y):

if x is None and y is None:

return x is None and y is None

elif x is None or y is None:

return False

elif x.\_value !=y.\_value:

return False

elif x.\_value==y.\_value:

return do\_equal(x.\_next,y.\_next)

else:

return False

return do\_equal(self.\_top,other.\_top)

def \_\_repr\_\_(self):

values=[]

node=self.\_top

while node is not None:

values.insert(0,node.\_value)

node=node.\_next

return 'Stack([' + ', '.join(repr(x) for x in values) + '] )'

# CLASE COLA

TAD nombre\_del\_TAD

Cola<a>

igualdad observacional

Si a y b son dos colas

a es igual a b si se cumple que:

las longitudes de a y b son iguales

Y

cada elemento de a es igual al correspondiente

en b

Usa

Natural ; Bool ; String ; None ; <a> ; secuencia<a>

Parámetro Formal

<a>

Géneros

Cola<a> C

observadores básicos

tamaño(Cola<a> C) → Natural

es\_vacia(Cola<a> C) → Bool

pri\_elem(Cola<a> C) → a {pre: la cola tiene al menos un elemento}

ult\_elem(Cola<a> C)→ a {pre: la cola tiene al menos un elemento}

Generadores

nueva\_cola() → Cola<a> P {post: crea una cola vacia}

cola(secuencia<a> C) → Cola<a> P {post: crea una cola con los elementos de la secuencia}

otras operaciones

encolar(Cola<a> C, a) → None {post: se agrega el elemento a al final de la cola}

desencolar(Cola<a> C) → a {pre: la cola tiene al menos un elemento}

vaciar(Cola<a>) → None {post: la cola esta vacia}

Axiomas

tamaño(Cola<a> C) {'devuelve un entero con la cantidad de elementos en la cola'}

es\_vacia(Cola<a> C) {retorna 'true -si la cola esta vacia, false -si la cola no esta vacia'}

pri\_elem(Cola<a> C) {'devuelve el primer elemento que entro en la cola'}

ult\_elem(Cola<a> C) {'devuelve el ultimo elemento que entro en la cola'}

nueva\_cola() {'crea una cola vacia')

cola(secuencia<a> C) {'crea una cola apartir de los elementos de la secuencia'}

insertar(Cola<a> C, a) {'agrega el elemento 'a' al final de la cola C '}

sacar(Cola<a>) {'obtenemos el ultimo elemento de la cola'}

Exporta

Cola<a> ; tamaño ; es\_vacia ; prim\_elem ; ult\_elem ; nueva\_cola ; cola ;

insertar ; sacar

## COLA CON LISTA

class Cola():

\_\_slots\_\_ = ['\_values','\_count']

# todas las fuciones son de o(1)

# escepto la funcion \_\_repr\_\_

# el metodo length es orden constante tmb pero sin

# usar el metodo len de python

def \_\_init\_\_(self, iterable = None):

self.\_values = []

self.\_count = 0

if iterable is not None:

self.push(iterable)

def is\_empty(self):

return len(self.\_values) == 0

@property

def top(self):

assert not self.is\_empty(), 'La cola esta vacia'

return self.\_values[-1]

def clear(self):

self.\_values.clear()

def push(self, value):

self.\_values.insert(0,value)

self.\_count += 1

def pull(self):

assert not self.is\_empty(), 'La cola esta vacia'

return self.\_values.pop()

def copy(self):

nueva\_cola = Cola()

nueva\_cola.\_values = self.\_values.copy()

return nueva\_cola

def length(self):

return self.\_count

def \_\_len\_\_(self):

return len(self.\_values)

def \_\_eq\_\_(self, other):

return self.\_values == other.\_values

def \_\_repr\_\_(self):

return "Cola{["+", ".join(repr(i) for i in self.\_values) + "]}"

# CLASE COLA DOBLEMENTE TERMINADA

TAD Nombre\_del\_TAD:

ColaDoble<a>

Igualdad Observacional:

Sean A y B dos colas dobles, son iguales si: la longitud de A y B son

iguales, y cada elemento de A es igual al correspondiente de B.

Usa:

Natural ; Bool ; None ; <a> ; secuencia<a>

Parametro Formal:

<a>

Género:

ColaDoble<a> C

Observadores Básicos:

Tamaño(ColaDoble<a> C) -> Natural

Es\_vacia(ColaDoble<a> C) -> Bool

Primero(ColaDoble<a> C) -> a {pre: La cola doble tiene al menos un elemento}

Ultimo(ColaDoble<a> C) -> a {pre: La cola doble tiene al menos un elemento}

Generadores:

Vacía() -> ColaDoble<a> C {post: La cola doble retornada esta vacia}

A\_partir\_de(Secuencia<a>) -> ColaDoble<a> C {post: La cola doble retornada contiene

los elementos de la secuencia con el mismo

orden que tiene la secuencia}

Otras opecaiones:

Agregar\_frente(ColaDoble<a> C, a) -> None

Agregar\_Final(ColaDoble<a> C, a) ->None {pre: la cola doble contiene al menos un elemento}

Borrar(ColaDoble<a> C) -> None {pre: la cola doble contiene al menos un elemento}

Axiomas:

Tamaño(ColaDoble<a> C) {retorna la cantidad de elementos que tiene la cola doble D}

Es\_vacia(ColaDoble<a> C) {retorna -true si la cola doble D esta vacia o -false si la cola doble D contiene elementos}

Primero(ColaDoble<a> C) {retorna el primer elemento de la cola doble D}

Ultimo(ColaDoble<a> C) {retorna el ultimo elemento de la cola doble D}

Vacía() {crea una cola doblemente termianda vacia}

A\_partir\_de(Secuencia<a> C) {crea una cola doble D a partir de la secuencia pasada por parametro}

Agregar\_frente(ColaDoble<a> C, a elemen) {agrega el elemento elemen al principio de la cola doble D}

Agregar\_Final(ColaDoble<a> C, a elemen) {agrega el elemento elemen al final de la cola doble D}

Borrar(ColaDoble<a> C) {elimina por completo la cola doble D}

Exporta:

ColaDoble<a> , tamaño , Es\_vacia , Primero , Ultimo , Vacía , A\_partir\_de , Agregar\_frente

Agregar\_Final , Borrar

## COLA DOBLE CON NODOS

from dataclasses import dataclass

from typing import Any # Any representa "cualquier tipo"

#import pdb; pdb.set\_trace()

class Cola\_doble():

@dataclass # decorador para definir una data class

class \_Node:

\_value: Any

sig: '\_Node'

prev: '\_Node'

\_\_slots\_\_ = ['\_len','\_start','\_end','\_back']

def \_\_init\_\_ (self, iterable = None):

self.\_end = self.\_start = self.\_back = None

self.\_len = 0

if iterable is not None:

self.push\_end(iterable)

def push\_end(self, value):

if (self.\_start == None):

aux = Cola\_doble.\_Node(value, None, None)

self.\_back = aux

self.\_start = aux

self.\_end = aux

self.\_len += 1

else:

aux = Cola\_doble.\_Node(value, None, None)

aux.prev = self.\_end

self.\_end.sig = aux

self.\_len += 1

def encolar\_frente(self, value):

nodoNuevo = Cola\_doble.\_Node(value, None, None)

#self.\_head.prev = nodoNuevo

nodoNuevo.sig = self.\_head

self.\_head = nodoNuevo

self.\_len += 1

if self.\_end == None:

self.\_end = nodoNuevo

def desencolar\_frente(self):

assert not self.is\_empty(), 'La cola esta vacia'

auxiliar = self.\_head

self.\_head = auxiliar.sig

#self.\_head.prev = None

self.\_len -= 1

return auxiliar.\_value

def desencolar\_final(self):

assert not self.is\_empty(), 'La cola esta vacia'

auxiliar = self.\_end

self.\_end = auxiliar.prev

#self.\_end.sig = None

self.\_len -= 1

return auxiliar.\_value

def borrar(self):

assert not self.is\_empty(), 'La cola esta vacia'

self.\_end = self.\_head = None

self.\_len = 0

def \_\_len\_\_(self):

return self.\_len

def \_\_repr\_\_(self):

values = []

node = self.\_head

while node is not None:

values.insert(len(values), node.\_value)

node = node.sig

return '(' + ', '.join(repr(x) for x in values) + ')'

def is\_empty(self):

return self.\_len==0

def \_\_eq\_\_(self, other):

aux1 = self.\_start

aux2 = other.\_start

while aux1 == None and aux2 == None:

if aux1.\_values == other.\_values:

aux1 = aux1.sig

aux2 = aux2.sig

else:

return False

return True

"""

@property

def primero(self):

assert not self.is\_empty(), 'La cola esta vacia'

#auxiliar = self.\_head

return self.\_head.\_value

@property

def ultimo(self):

assert not self.is\_empty(), 'La cola esta vacia'

#auxiliar = self.\_end

return self.\_end.\_value

"""

# CLASE ARREGLO DINÁMICO

Molde de Especificacion:

TAD Nombre\_del\_TAD:

ArregloDinamico<a>

Igualdad Observacional:

Si a y b son dos arreglos dinamicos

a es igual a b si cumple que:

El tamaño de a y b son iguales

Cada elemento en a es igual a correspondiente a cada

elemento en b.

Usa:

Bool, Natural, a, secuencia<a>

Parametro Formal:

<a>

Género:

ArregloDinamico<a> A

Observadores Básicos:

es\_vacio(ArregloDinamico<a> A) -> Bool

tamaño(ArregloDinamico<a> A) -> Natural {pre: el arreglo tiene al menos un elemento}

buscar( ArregloDinamico<a> A, a) -> Natural {pre: el arreglo tiene al menos un elemento}

Generadores:

nuevo\_arreglo() -> None

nuevo\_arreglo(secuencia<a> A) -> None

copiar(ArregloDinamico<a> A) -> ArregloDinamico<a> B

Otras Operaciones:

agregar(ArregloDinamico<a> A, a) -> None

quitar(ArregloDinamico<a> A, a) -> None {pre: el arreglo tiene al menos un elemento}

elimiar(ArregloDinamico<a> A) -> None

accederIndice (ArregloDinamico<a> A, Indice) -> a {pre: el arreglo tiene al menos un elemento}

Axiomas:

es\_vacio(ArregloDinamico<a> A) {retorna 'True -si el arreglo esta vacio o False -si el arreglo tiene al menos un elemento'}

tamaño(ArregloDinamico<a> A) {'devuelve un entero con la cantidad de elementos en el arreglo A'}

buscar(ArregloDinamico<a> A, a) {'Devuelve el indice del elemento a en el arreglo A'}

nuevo\_arreglo() {'Crea un arreglo vacio'}

nuevo\_arreglo(secuencia<a> A) {'crea un arreglo A a partir de una secuencia'}

copiar(ArregloDinamico<a> A) { retorna un nuevo ArregloDinamico<a> B que es una copia del ArregloDinamico<a> A}

agregar(ArregloDinamico<a> A, a) {'agrega el elemento a al arreglo A'}

quitar(ArregloDinamico<a> A, a) {'elimina ul elemento a del arreglo A'}

elimiar(ArregloDinamico<a> A) {'elimina el arreglo A por completo'}

accederIndice (ArregloDinamico<a> A, i) {'accedemos a un elemento que se encuentra en el indice del arreglo A'}

Exporta:

ArregloDinamico<a> ; es\_vacia ; tamaño ; buscar ; nuevo\_arreglo ; nuevo\_arreglo()

copiar ; agregar ; quitar ; elimiar ; accederIndice.

# CLASE CONJUNTO

TAD Nombre\_del\_TAD:

Conjunto<a>

Igualdad Observacional:

Si a y b son dos conjuntos

a es igual a b si cumple que:

El tamaño de a y b son iguales

Contienen los mismos elementos

Usa:

Bool, Natural, a, secuencia<a>, Conjunto

Parametro Formal:

<a>

Género:

Conjunto<a> S

Observadores Básicos:

es\_vacio(Conjuntos<a> S) -> Bool

tamaño(Conjuntos<a> S) -> Natural

encontrar(Conjuntos<a> S, a) -> Bool

Generadores:

nuevo\_conjunto() -> None {post: 'Crea un conjunto vacio'}

nuevo\_conjunto(secuencia<a> S, a) -> a { post: 'Crea un conjunto a partir de una secuencia'}

Otras opecaiones:

agregar(Conjunto<a> S, a) -> None

quitar(Conjunto<a> S, a) -> None {pre: el conjunto tiene al menos un elemento}

vaciar(Conjunto<a> S) -> None {post: 'elimina todos los elementos del conjunto'}

union(Conjunto<a> S, Conjunto<a> R) -> Conjunto<a>

interseccion(Conjunto<a> S, Conjunto<a> R) -> Conjunto<c>

diferencia(Conjunto<a> S, Conjunto<a> R) -> Conjunto<c>

Axiomas:

es\_vacio(Conjuntos<a> S) {retorna un 'True -si el conjunto esta vacio o False -si el conjunto no esta vacio'}

tamaño(Conjuntos<a> S) {'devuelve un entero con el numero de elementos que contiene'}

encontrar(Conjuntos<a> S, a) {retorna 'True -si el elemento a pertenece al conjunto S Flase -si no pertenece'}

nuevo\_conjunto() {'Crea un conjunto vacio'}

nuevo\_conjunto(secuencia<a> S) {'Crea un conjunto S a partir de una secuencia de elementos a'}

agregar(Conjunto<a> S, a) {'agrega el elemento 'a' al conjunto S'}

quitar(Conjunto<a> S, a) {'quita el elemento 'a' del conjunto S'}

vaciar(Conjunto<a> S) {'elimina todos los elementos del conjunto S'}

union(Conjunto<a> S, Conjunto<a> R) {'une a los conjuntos a y b dentro del conjunto a, eliminando los elementos que tengan en comun'}

interseccion(Conjunto<a> S, Conjunto<a> R) {'retorna un nuevo conjunto con los elementos que se encuentran en ambos conjuntos'}

diferencia(Conjunto<a> S, Conjunto<a> R) {'retorna un nuevo conjunto que contiene los elementos del conjunto S que no están en el conjunto R'}

Exporta:

Conjunto<a> ; es\_vacio ; tamaño ; encontrar ; nuevo\_conjunto ; nuevo\_conjunto() ;

agregar ; quitar ; vaciar ; union ; interseccion ; diferencia.

## CONJUNTO

import nodosimple

class Conjunto():

\_\_slots\_\_ = ['\_first', '\_size']

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

self.\_first = None

self.\_size = 0

@property

def empty(self):

return self.\_size == 0

def \_\_len\_\_(self):

return self.\_size

def add(self, date):

if self.empty:

self.\_first = nodosimple.Nodo(date)

self.\_size += 1

else:

aux = nodosimple.Nodo(date)

aux.\_next = self.\_first

self.\_first = aux

self.\_size += 1

def \_\_eq\_\_(self,other):

if self.\_size == other.\_size:

aux1 = self.\_first

aux2 = other.\_first

else:

return False

while aux2 is not None:

while aux1 is not None:

if aux1.\_date != aux2.\_date:

aux1 = aux1.\_next

elif aux1.\_date == aux2.\_date:

break

elif aux1.\_next == None:

return False

aux2 = aux2.\_next

aux1 = self.\_first

return True

def run\_list(self):

aux = self.\_first

while aux != None:

print(aux.\_date)

aux = aux.\_next

# CLASE LISTA ENLAZADA SIMPLE

TAD Nombre\_del\_TAD

lista\_simple<a>

Igualdad Observacional:

Si a y b son dos pilas

a es igual a b si se cumple que:

Las longitudes de a y b son iguales

Y

cada elemento en a es igual al correspondiente

elemento en b.

Usa

Parámetro Formal

<a>

Géneros

lista\_simple<a> L

observadores básicos

es\_vacia(lista\_simple<a> L) → Bool

tamaño(lista\_simple<a> L) → Natural

mostrar(lista\_simple<a> L[i]) → a {pre: la lista tiene al menos un elemento}

Generadores

nueva\_lista() → None

nueva\_lista(secuencia<a> L) → a

otras operaciones

agregarFrente(lista\_simple<a> L, a) → None

agregarFinal(lista\_simple<a> L, a) → None

agregarIndice(lista\_simple<a> L, a , i) → None

quitar(lista\_simple<a> L ,a) → a {pre: la lista tiene al menos un elemento}

elimiar(lista\_simple<a> L) → None {pre: la lista tiene al menos un elemento}

accederIndice(lista\_simple<a> L ,i) → a {pre: la lista tiene al menos un elemento}

cortar(lista\_simple<a> L , i ) → lista\_simple<a> B / Bool {pre: la lista tiene al menos un elemento}

{post: si el indice especificado no se encuentra retorna false}

{post: en la lista original se mantendan los elementos que no fueron cortados}

Axiomas

es\_vacia(lista\_simple<a> L) {'retorna true -si la lista L esta vacia, false -si la lista L contiene elementos'}

tamaño(lista\_simple<a> L ) {'devuelve un entero con la cantidad de elemetos que tiene la lista L'}

mostrar(lista\_simple<a> L[i]) {'muestra el elemento del indice [i] en la lista L'}

nueva\_lista() {'crea una lista vacia'}

nueva\_lista(secuencia<a> L) {'crea la lista L a partir de una secuencia'}

agregarFrente(lista\_simple<a> L, a ) {'agrega el elemento a al principio de la lista L'}

agregarFinal(lista\_simple<a> L, a ) { agrega el elemento a en el final de la lista L }

agregarIndice(lista\_simple<a> L , a ,i) { agrega el elemento a en la posicion i de la lista L}

quitar(lista\_simple<a> L, a) {quita el elemento a si se encuentra en la lista L}

elimiar(lista\_simple<a> L) { borra toda la lista L}

accederIndice(lista\_simple<a> L,i) {obtenemos el elemento que esta en la posicion i de la lista L}

cortar(lista\_simple<a> L, i ) {retorna una nueva lista con los elementos siguiente al indice especificado,

y ademas, en la lista original borra los elementos siguiente del indice especificado. Sí el indice no es valido,

producira un Error}

Exporta

lista\_simple<a>; es\_vacia ; tamaño ; mostrar; nueva\_lista ; nueva\_lista() ; agregarFrente ;

agregarFinal ; accederIndice ; quitar ; eliminar ; accederIndice ; cortar

## LISTA SIMPLE LISTA

from Implementaciones import nodosimple

class SimpleList():

\_\_slots\_\_ = ['\_head', '\_size', '\_tail']

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

self.\_head = None

self.\_tail = None

self.\_size = 0

@property

def is\_empty(self):

return self.\_head == None

@property

def front(self):

return self.\_head.\_date

@property

def back(self):

return self.\_tail.\_date

def \_\_len\_\_(self):

return self.\_size

def len\_recursive(self):

aux = self.\_head

def lenR(aux):

if aux == None:

return 0

else:

aux = aux.\_next

return 1 + lenR(aux)

count = lenR(aux)

return count

def add\_back(self,date):

if self.is\_empty:

self.\_head = self.\_tail = nodosimple.Nodo(date)

self.\_size += 1

else:

aux = nodosimple.Nodo(date)

self.\_tail.\_next = aux

self.\_tail = aux

self.\_size += 1

def add\_front(self,date):

if self.is\_empty:

self.\_head = self.\_tail = nodosimple.Nodo(date)

self.\_size += 1

else:

aux = nodosimple.Nodo(date)

aux.\_next = self.\_head

self.\_head = aux

self.\_size += 1

def delete\_back(self):

aux = self.\_head

while aux.\_next != self.\_tail:

aux = aux.\_next

aux.\_next = None

self.\_tail = aux

self.\_size -= 1

def delete\_front(self):

self.\_head = self.\_head.\_next

self.\_size -= 1

def add\_rec(self):

aux = self.\_head

def add(aux):

if aux == None:

return 0

else:

count = aux.\_date

aux = aux.\_next

return (count + add(aux))

count = add(aux)

return count

def add\_iter(self):

aux = self.\_head

count = 0

while aux != None:

count = count + aux.\_date

aux = aux.\_next

return count

def run\_list(self):

aux = self.\_head

while aux != None:

print(aux.\_date)

aux = aux.\_next

## LISTA SIMPLE CON NODO

from dataclasses import dataclass

from tkinter.messagebox import RETRY

from typing import Any

class SinglyLinkedList():

@dataclass

class \_Head:

next: '\_Node' = None

@dataclass

class \_Node:

value: Any

next: '\_Node' = None

\_\_slots\_\_ = ['\_head','\_size']

def \_\_init\_\_(self, value = None):

if value == None:

self.\_head = SinglyLinkedList.\_Head(None)

self.\_size = 0

else:

new = SinglyLinkedList.\_Node(value, None)

self.\_head = SinglyLinkedList.\_Head(new)

self.\_size += 1

def \_\_len\_\_(self):

return self.\_size

def is\_empty(self):

return self.\_size == 0

def clean(self):

self.\_head.next = None

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

# Usando abstracciones

p = self.begin()

while p != self.end():

yield p.value

p.advance()

# Otra opción más "a mano"...

# nodo = self.\_head.next

# while nodo is not None:

# yield nodo.value

# nodo = nodo.next

def begin(self):

return SinglyLinkedList.Coordinate(self.\_head.next)

def end(self):

return SinglyLinkedList.Coordinate(None)

def before\_begin(self):

return SinglyLinkedList.Coordinate(self.\_head)

def insert\_after(self, coord, value):

current = coord.\_node

new\_node = SinglyLinkedList.\_Node(value=value, next=current.next)

current.next = new\_node

self.\_size += 1

return SinglyLinkedList.Coordinate(new\_node)

def insert\_front(self, value):

self.insert\_after(self.before\_begin(), value)

def erase\_after(self, coord):

current = coord.\_node

current.next = current.next.next

self.\_size -= 1

return coord.next()

def \_\_repr\_\_(self):

return 'SinglyLinkedList([' + ', '.join(repr(v) for v in self) + '])'

"""

def \_\_repr\_\_(self):

values = []

aux = self.\_head.next

while aux is not None:

values.insert(len(values), aux.value)

aux = aux.next

return '\n'.join(repr(x) for x in values)

"""

class Coordinate():

\_\_slots\_\_ = ['\_node']

def \_\_init\_\_ (self, coordinate\_or\_node):

if isinstance(coordinate\_or\_node,SinglyLinkedList.Coordinate):

self.\_node = coordinate\_or\_node.\_node

else:

self.\_node = coordinate\_or\_node

@property

def value(self):

return self.\_node.value

@value.setter

def value(self,value):

self.\_node.value = value

def advance (self):

self.\_node = self.\_node.next

return self

def next (self):

c = SinglyLinkedList.Coordinate(self.\_node)

return c.advance()

def \_\_eq\_\_ (self, other):

return self.\_node is other.\_node

# CLASE LISTA DOBLEMENTE ENLAZADA

TAD Nombre\_del\_TAD:

ColaDoble<a>

Igualdad Observacional:

Sean A y B dos colas dobles, son iguales si: la longitud de A y B son

iguales, y cada elemento de A es igual al correspondiente de B.

Usa:

Natural ; Bool ; None ; <a> ; secuencia<a>

Parametro Formal:

<a>

Género:

ColaDoble<a> C

Observadores Básicos:

Tamaño(ColaDoble<a> C) -> Natural

Es\_vacia(ColaDoble<a> C) -> Bool

Primero(ColaDoble<a> C) -> a {pre: La cola doble tiene al menos un elemento}

Ultimo(ColaDoble<a> C) -> a {pre: La cola doble tiene al menos un elemento}

Generadores:

Vacía() -> ColaDoble<a> C {post: La cola doble retornada esta vacia}

A\_partir\_de(Secuencia<a>) -> ColaDoble<a> C {post: La cola doble retornada contiene

los elementos de la secuencia con el mismo

orden que tiene la secuencia}

Otras opecaiones:

Agregar\_frente(ColaDoble<a> C, a) -> None

Agregar\_Final(ColaDoble<a> C, a) ->None {pre: la cola doble contiene al menos un elemento}

Borrar(ColaDoble<a> C) -> None {pre: la cola doble contiene al menos un elemento}

Axiomas:

Tamaño(ColaDoble<a> C) {retorna la cantidad de elementos que tiene la cola doble D}

Es\_vacia(ColaDoble<a> C) {retorna -true si la cola doble D esta vacia o -false si la cola doble D contiene elementos}

Primero(ColaDoble<a> C) {retorna el primer elemento de la cola doble D}

Ultimo(ColaDoble<a> C) {retorna el ultimo elemento de la cola doble D}

Vacía() {crea una cola doblemente termianda vacia}

A\_partir\_de(Secuencia<a> C) {crea una cola doble D a partir de la secuencia pasada por parametro}

Agregar\_frente(ColaDoble<a> C, a elemen) {agrega el elemento elemen al principio de la cola doble D}

Agregar\_Final(ColaDoble<a> C, a elemen) {agrega el elemento elemen al final de la cola doble D}

Borrar(ColaDoble<a> C) {elimina por completo la cola doble D}

Exporta:

ColaDoble<a> , tamaño , Es\_vacia , Primero , Ultimo , Vacía , A\_partir\_de , Agregar\_frente

Agregar\_Final , Borrar

## LISTA DOBLE

import nododoble

class DoblyList():

\_\_slots\_\_ = ['\_first', '\_last', '\_size']

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

self.\_first = self.\_last = None

self.\_size = 0

@property

def is\_empty(self):

return self.\_size == 0

@property

def front(self):

return self.\_first.\_date

@property

def back(self):

return self.\_last.\_date

@property

def len(self):

return self.\_size

def add\_back(self, date):

if self.is\_empty:

self.\_first = self.\_last = nododoble.Nodo(date)

else:

aux = self.\_last

self.\_last = aux.\_next = nododoble.Nodo(date)

self.\_last.\_prev = aux

self.\_size += 1

def add\_front(self, date):

if self.is\_empty:

self.\_first = self.\_last = nododoble.Nodo(date)

else:

aux = nododoble.Nodo(date)

aux.\_next = self.\_first

self.\_first.\_prev = aux

self.\_first = aux

self.\_size += 1

def remove\_front(self):

if self.is\_empty:

print("Esta vacia")

elif self.\_first.\_next == None:

self.\_first = self.\_last = None

self.\_size = 0

else:

self.\_first = self.\_first.\_next

self.\_first.\_prev = None

self.\_size -= 1

def remove\_back(self):

if self.is\_empty:

print("Esta vacia")

elif self.\_first.\_next == None:

self.\_first = self.\_last = None

self.\_size = 0

else:

self.\_last = self.\_last.\_prev

self.\_last.\_next = None

self.\_size -= 1

def inverso(self):

other = self.\_last

new = DoblyList()

while other != None:

new.add\_back(other.\_date)

other = other.\_prev

return new

def run\_first(self):

aux = self.\_first

while aux != None:

print(aux.\_date)

aux = aux.\_next

def run\_last(self):

aux = self.\_last

while aux != None:

print(aux.\_date)

aux = aux.\_prev

## LISTA DOBLE CON NODO

from nododoble import Head, Nodo, Coordinate, Iterator

class DoblyList():

\_\_slots\_\_ = ['\_head', '\_size']

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

self.\_head = Head()

self.\_head.\_prev = self.\_head.\_next = self.\_head

self.\_size = 0

@property

def is\_empty(self):

return self.\_head.\_next is self.\_head

@property

def front(self):

assert not self.is\_empty, 'Empty list'

return self.\_head.\_next.\_date

@property

def back(self):

assert not self.is\_empty, 'Empty list'

return self.\_head.\_prev.\_date

def append\_front(self, date):

self.insert(self.begin(), date)

def append\_back(self, date):

self.insert(self.end(), date)

def clear(self):

self.\_head.\_prev = self.\_head.\_next = self.\_head

def begin(self):

return Coordinate(self.\_head.\_next)

def end(self):

return Coordinate(self.\_head.\_prev)

def \_\_len\_\_(self):

n = 0

nodo = self.\_head.\_next

while nodo is not self.\_head:

n += 1

nodo = nodo.\_next

return n

def insert(self, coord, date):

aux = coord.\_node

new = Nodo(date)

new.\_prev = aux.\_prev

new.\_next = aux

new.\_prev.\_next = new

new.\_next.\_prev = new

self.\_size += 1

return Coordinate(aux)

def \_\_eq\_\_(self, other):

p = self.begin()

q = other.begin()

while p != self.end() and q != other.end():

if p.value != q.value:

return False

p.advance()

q.advance()

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

def \_\_repr\_\_(self):

return 'DoublyLinkedList([' + ', '.join(repr(v) for v in self) + '])'

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

return Iterator(self.\_head.\_next, self.\_head)

def CoordinateMaker(self, index):

assert not self.is\_empty, 'Empty list'

aux = self.\_head

count = 0

def valid\_index(indx, max):

return (indx >= 0 and indx < max)

if (valid\_index(index, len(self))):

while aux is not None and count <= index:

aux = aux.\_next

count += 1

return Coordinate(aux)

## LISTA CIRCULAR

from Implementaciones import nodosimple

class CircularList():

\_\_slots\_\_ = ['\_first', '\_last', '\_size']

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

self.\_first = self.\_last = None

self.\_size = 0

@property

def is\_empty(self):

return self.\_size == 0

@property

def front(self):

return self.\_first.\_date

@property

def back(self):

return self.\_last.\_date

@property

def len(self):

return self.\_size

def add\_back(self, date):

if self.is\_empty:

self.\_first = self.\_last = nodosimple.Nodo(date)

self.\_last.\_next = self.\_first

else:

aux = nodosimple.Nodo(date)

self.\_last.\_next = aux

aux.\_next = self.\_first

self.\_last = aux

self.\_size += 1

def add\_front(self, date):

if self.is\_empty:

self.\_first = self.\_last = nodosimple.Nodo(date)

self.\_last.\_next = self.\_first

else:

aux = nodosimple.Nodo(date)

self.\_last.\_next = aux

aux.\_next = self.\_first

self.\_first = aux

self.\_size += 1

def remove\_front(self):

if self.is\_empty:

print("Is empty")

elif self.\_first == self.\_last:

self.\_first = self.\_last = None

self.\_size -= 1

else:

self.\_first = self.\_first.\_next

self.\_last.\_next = self.\_first

self.\_size -= 1

def remove\_back(self):

if self.is\_empty:

print("Is empty")

elif self.\_first == self.\_last:

self.\_first = self.\_last = None

self.\_size -= 1

else:

aux = self.\_first

while aux.\_next != self.\_last:

aux = aux.\_next

self.\_last = aux

self.\_last.\_next = self.\_first

self.\_size -= 1

def run\_list(self):

aux = self.\_first

while aux:

print(aux.\_date)

aux = aux.\_next

if aux == self.\_first:

break

## LISTA CIRCULAR DOBLE

from Implementaciones import nododoble

class DobleCircularList():

\_\_slots\_\_ = ['\_size','\_first','\_last',]

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

self.\_size = 0

self.\_first = self.\_last = None

@property

def empty(self):

return self.\_size == 0

@property

def front(self):

return self.\_first.\_date

@property

def back(self):

return self.\_last.\_date

@property

def len(self):

return self.\_size

def find(self, find):

aux = self.\_first

count = 1

while aux.\_next != self.\_first:

if aux.\_date == find:

return 'El elemento ', find , 'se encontro en la posicion ', count

else:

aux = aux.\_next

count += 1

if aux.\_date == find:

return 'El elemento ', find , 'se encontro en la posicion ', count

return 'Elemento no encontrado'

def add\_front(self, date):

if self.empty:

self.\_first = self.\_last = nododoble.Nodo(date)

self.\_first.\_next = self.\_last

self.\_first.\_prev = self.\_last

self.\_last.\_next = self.\_first

self.\_last.\_prev = self.\_first

else:

aux = nododoble.Nodo(date)

aux.\_next = self.\_first

aux.\_prev = self.\_last

self.\_first.\_prev = aux

self.\_last.\_next = aux

self.\_first = aux

self.\_size += 1

def add\_back(self, date):

if self.empty:

self.\_first = self.\_last = nododoble.Nodo(date)

self.\_first.\_next = self.\_last

self.\_first.\_prev = self.\_last

self.\_last.\_next = self.\_first

self.\_last.\_prev = self.\_first

else:

aux = nododoble.Nodo(date)

aux.\_next = self.\_first

aux.\_prev = self.\_last

self.\_first.\_prev = aux

self.\_last.\_next = aux

self.\_last = aux

self.\_size += 1

def remove\_front(self):

if self.empty:

print("List empty")

elif self.\_first == self.\_last:

self.\_first = self.\_last = None

self.\_size -= 1

else:

self.\_first = self.\_first.\_next

self.\_last.\_next = self.\_first

self.\_first.\_prev = self.\_last

self.\_size -= 1

def remove\_back(self):

if self.empty:

print("List empty")

elif self.\_first == self.\_last:

self.\_first = self.\_last = None

self.\_size -= 1

else:

self.\_last = self.\_last.\_prev

self.\_last.\_next = self.\_first

self.\_first.\_prev = self.\_last

self.\_size -= 1

def run\_list(self):

aux = self.\_first

while aux:

print(aux.\_date)

aux = aux.\_next

if aux == self.\_first:

break

# CLASE COORDENADA

TAD Nombre\_del\_TAD:

Coordenada <b<a>>

Igualdad Observacional:

Si a y b son dos coordenadas

a es igual a b si se cumple que: hacen referencia a la misma

posición dentro del mismo contenedor

Usa:

Parámetro Formal:

<b, a>

Géneros:

Coordenada <b<a>>

Observadores Básicos:

valor(Coordenada<b<a>>) → a {Pre: La coordenada es válida}

Generadores:

a\_partir\_de(Coordenada <b<a>>) → Coordenada <b<a>>

Otras Operaciones:

avanzar(Coordenada<b<a>>) → Coordenada<b<a>>

siguiente(Coordenada<b<a>>) → Coordenada<b<a>>

retroceder(Coordenada<b<a>>) → Coordenada<b<a>>

previo(Coordenada<b<a>>) → Coordenada<b<a>>

Axiomas:

a\_partir\_de(Coordenada <b<a>> c): crea una coordenada a partir de la coordenada c

valor(Coordenada<b<a>> c): retorna el valor al que hace referencia la coordenada c

siguiente(Coordenada<b<a>> c): retorna una nueva coordenada que hace referencia a la posición siguiente de c.

previo(Coordenada<b<a>> c): retorna una nueva coordenada que hace referencia a la posición anterior de c.

avanzar(Coordenada<b<a>> c) avanza a la siguiente a c

retroceder(Coordenada<b<a>> c) retrocede a la anterior a c

Exporta:

Coordenada<b<a>>,a\_partir\_de, siguiente, previo,

valor, avanzar, retroceder

## COORDENADA

import nododoble

class Coordinate():

\_\_slots\_\_ = ['\_node']

def \_\_init\_\_(self, cordinate\_node):

if isinstance(cordinate\_node, Coordinate):

self.\_node = cordinate\_node.\_node

else:

self.\_node = cordinate\_node

@property

def value(self):

return self.\_node.\_date

@value.setter

def value(self,date):

self.\_node.\_date = date

def advance(self):

self.\_node = self.\_node.\_next

return self

def next(self):

return Coordinate(self.\_node).advance()

def retreat(self):

self.\_node = self.\_node.\_prev

return self

def prev(self):

return Coordinate(self.\_node).retreat()

def \_\_eq\_\_(self,other):

return self.\_node is other.\_node

# CLASE ARBOL BINARIO DE BÚSQUEDA

TAD Nombre TAD:

BinarySerchTree<a,b>

Igualdad Observacional:

si c y d son dos arboles binario de busqueda

c es igual a d si se cumple que:

c contiene los mismos elementos que d

Usa:

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

Coordenada<BinarySerchTree<a,b>>

Parámetro Formal:

a, b

Géneros:

BinarySerchTree<a,b>

observadores básicos:

cantidad\_de\_nodos(BinarySerchTree<a, b>) → Natural

es\_vacío(BinarySerchTree<a, b>) → Bool

máximo(BinarySerchTree<a, b>) → Coordenada<BinarySerchTree<a,b>> {Pre: el árbol tiene al menos un elemento}

mínimo(BinarySerchTree<a, b>) → Coordenada<BinarySerchTree<a,b>> {Pre: el árbol tiene al menos un elemento}

buscar\_clave(BinarySerchTree<a, b>, a) → Coordenada<BinarySerchTree<a,b>>

Generadores:

vacío() → BinarySerchTree<a, b> {Post: El Árbol retornado esta vacío}

a\_partir\_de(Secuencia<Tupla<a, b>>) → BinarySerchTree<a,b> {Post: El árbol contiene los elementos de la secuencia recibida}

otras operaciones:

insertar(BinarySerchTree<a, b>, a, b) →Coordenada<BinarySerchTree<a, b>> {Pos: El árbol no esta vacío}

borrar(BinarySerchTree<a, b>, a) → Bool, Coordenada<BinarySerchTree<a, b>>

inicio(BinarySerchTree<a, b>) → coordenada<BinarySerchTree<a, b>>

fin(BinarySerchTree<a, b>) → coordenada<BinarySerchTree<a, b>>

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

es\_vacío(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

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:

BinarySerchTree<a, b>, vacío, a\_partir\_de,

insertar, borrar, tamaño, es\_vacío, mínimo, máximo,

buscar\_clave, inicio, fin

## ABB

from dataclasses import dataclass

from typing import Any, Union

##implementacion arbol binario de busqueda

## basicamente iteramos el arbol hasta encontrar el nodo que tiene el valor menor

def \_minimum\_node(node): ##la funcion que retorna el minimo nodo

if node is not None: ##si el arbol no esta vacio

while (node.left is not None): ##mientras el nodo.left no sea un nodo terminal

node = node.left ##iteramos

return node

## basicamente iteramos el arbol hasta encontrar el nodo que tiene el mayor valor

def \_maximum\_node(node): ##la funcion que retorna el nodo mayor

if node is not None: ##si el arbol no esta vacio

while (node.right is not None): ##mientras el nodo.right no sea un nodo terminal

node = node.right ##iteramos

return node

class Abb():

@dataclass

class \_Node:

key: Any ##la clave 'se puede comparar'

value: Any ##el valor

parent: Union['\_Node','\_Root'] = None ## el union es porque el padre puede ser o un nodo o la raiz

left: '\_Node' = None ##inicializo los nodos

right: '\_Node' = None

@dataclass

class \_Root:

left: '\_Node' = None ##inicializo los nodos

right: '\_Node' = None

parent: '\_Node' = None ##el nodo que simboliza el padre, si el padre es None significa que ese nodo es Root

\_\_slots\_\_ = ['\_root','\_len'] ##atributos

def \_\_init\_\_(self, iterable=None):

self.\_root = self.\_Root()

self.\_len = 0

if iterable is not None:

for key, value in iterable:

self.insert(key, value)

def is\_empty(self):

return self.\_root.left is None ## aca se usa el nodo izquierdo como nodo base? es como el prev de la pila que implemente

##si el nodo izquierdo de la raiz es none, es porque esta vacio

def \_\_len\_\_(self):

return self.\_len ##devuelve la cantidad de nodos

def len\_recursivo(self):

def do\_len(node):

if(node == None):

return 0

return 1 + do\_len(node.right) + do\_len(node.left)

return do\_len(self.\_root.left)

def begin(self):#devuelve la coord que hace referencia al nodo mas chiquito (o mas izqu del arbol)

return Abb.\_Coordinate(\_minimum\_node(self.\_root))

def end(self):#devuelve la coord que hace referencia al ultimo nodo (siguiente del ultimo)

return Abb.\_Coordinate(self.\_root)##devuelve solo root, porque no se pone maximum?-----------------------------------------------

def minimum(self):

return Abb.\_Coordinate(\_minimum\_node(self.\_root)) ##retorna una coord que hace referecia al nodo minmo

def maximum(self):

return Abb.\_Coordinate(\_maximum\_node(self.\_root.left)) ##retorna una coord que hace referecia al nodo maximo

##find recursivo, aporta mas legibilidad al codigo

def find(self, key): ##le paso la clave

def do\_find(node): ## funcion interna

if (node is None): ##si node es none

return self.end() ##retorna la coord que hace referencia a root indicando que no lo encontro

elif (key < node.key): ##si la clave es mas chica que la clave del nodo actual

return do\_find(node.left) ##me voy a buscar la clave por el lado izquierdo es decir, lo que busco es mas chiquito que el actual

elif (key > node.key): ##si la clave es mas grande que la clave del nodo actual

return do\_find(node.right)##me voy a la parte derecha del arbol, es decir, la clave es mas grande que la actual

else: ##hasta que eventualmente key==node.key, por lo tanto

return Abb.\_Coordinate(node) ##retorno la Coord que hace referencia al nodo

return do\_find(self.\_root.left) #caso recursivo

##cuales son las desventajas y ventajas de ambos ------------------------------------------------------------------------------------------------------------------

##el find es mejor iterativo que recursivo, porque el recursivo consume mucho mas recursos

##ademas te ahorras las pilas de activacion

def find\_iterativo(self, key):#hace lo mismo de la funcion de arriba

node=self.\_root.left#en logica es parecida

while True:#lo que cambia es que hace un bucle

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 Abb.\_Coordinate(node)

##este find lo hice con el proposito de saber si esta o no la clave a buscar

def find\_boolean(self, key): ##le paso la clave

def do\_find(node): ## funcion interna

if (node is None): ##si node es none

return False ##retorna False indicando que no lo encontro

elif (key < node.key): ##si la clave es mas chica que la clave del nodo actual

return do\_find(node.left) ##me voy a buscar la clave por el lado izquierdo es decir, lo que busco es mas chiquito que el actual

elif (key > node.key): ##si la clave es mas grande que la clave del nodo actual

return do\_find(node.right)##me voy a la parte derecha del arbol, es decir, la clave es mas grande que la actual

else: ##hasta que eventualmente key==node.key, por lo tanto

return True ##retorna True indicando que si la encontro

return do\_find(self.\_root.left) #caso recursivo

##lo bueno de hacerlo recursivo es porque hace el enganche automatico

##a traves de las pilas de activacion

def insert(self, key, value=None): ##funcion que inserta

def do\_insert(node, parent): ##funcion interna

if (node is None): ##

node = Abb.\_Node(key, value, parent)##aca se genera el nodo con la clave, el valor y el padre

coord=Abb.\_Coordinate(node)## coord es la coord que hace referencia al nodo de arriba

self.\_len+=1 ##le sumo 1 al len porque el len implementado es de O(1)

elif(key < node.key):## aca hace lo mismo que en el find pero con la diferencia

node.left, coord = do\_insert(node.left, node)## que en vez de pasarle solo el nodo le paso el padre tambien

elif(key > node.key):

node.right, coord= do\_insert(node.right,node)

else: #key == node.key

node.value = value ##le asigno el valor None

coord = Abb.\_Coordinate(node) ##ya lo explique un par de comentarios arriba

return node, coord ##el return se ejecuta al terminar siempre

#lo que se logra con esto es hacer el enganche automatico

self.\_root.left, coord=do\_insert(self.\_root.left, self.\_root)##primero se ejecuta esto, le paso por parametro el primer nodo y el padre del nodo actual que es la raiz

## para luego empezar a recorrer el arbol

return coord ## por ultimo retorno la coord del nodo que acabo de insertar

def insert\_iterativo(self, key, value=None):##-----------------------------------------------------------------------------------------------------------------------------

node=self.\_root.left #creo una variable auxiliar que tendra primer nodo del arbol, el hijo izquierdo de la raiz.

parent=self.\_root #creo una variable auxiliar padre para poder ir guardando el padre durante el recorrido del arbol para poder luego asignarselo al nuevo nodo insertado

while node is not None: #recorro el arbol siempre que el nodo en la ubicacion del recorrido no sea None

if key<node.key: #si la clave (key) del nodo que se quiere insertar es menor a la del nodo en la ubicacion del recorrido del arbol

parent=node #entonces el padre del nodo a insertar pasara a ser el nodo en la ubicacion del recorrido que se esta comparando

node=node.left #le reasigno el valor al nodo auxiliar para mi recorrido asignandole el valor de su hijo a la izquierda, de esta manera, voy bajando hacia la izquierda en el arbol

elif key>node.key: #si la clave del nodo que se quiere insertar es mayor que la del nodo en la ubicacion del recorrido en el arbol

parent=node #entonces el padre del nodo a instertar pasara a ser el nodo en la ubicacion del recorrido que se esta comparando

node=node.right #le reasigno el valor al nodo auxiliar para mi recorrido asignandole el valor de su hijo a la derecha, de esta manera, voy bajando hacia la derecha en el arbol

else: #si la clave no es menor ni mayor, entonces sera igual, por lo tanto el nodo que se quiere insertar ya existe en el arbol

node.value=value #cambio el valor interno que tiene el nodo en el arbol por le valor del nodo que se quiere insertar

return Abb.\_Coordinate(node) #retorno una coordenada con el nodo que se inserto, que en realidad, en este caso, solo se cambio su valor interno

node=self.\_Node(key,value,parent) #si el nodo en el recorrido es none, entonces se saldra del while,por lo tanto debere insertar el nodo en ese lugar, por lo tanto creo mi nuevo nodo con su clave, su valor y su padre que sera el anterior a el mismo

if key>parent.key: #si la clave del nodo a insertar es mayor que la del padre

parent.right=node #entonces insertare el nuevo nodo a la derecha del padre,ya que es mayor a el

else: #en cambio, si la clave del nodo a insertar es menor que la del padre

parent.left=node #entonces insertare el nuevo nodo a la izquierda del padre, ya que es menor a el

coord=Abb.\_Coordinate(node) #creo una nueva coordenada con el nuevo nodo insertado

self.\_len+=1 #aumento en uno la cantidad de nodos que tiene el arbol

return coord #retorno la coordenada

def insertar\_iterativo(self, key, value=None):

node=self.\_root.left

parent=None

while node is not None:

if key<node.key:

parent=node

node=node.left

elif key>node.key:

parent=node

node=node.right

else:

return False

neWnode=self.\_Node(key,value,parent)

if key>parent.key:

parent.right=neWnode

else:

parent.left=neWnode

return True

def erase(self, key):

def do\_erase(node):

if(node is None): ##si en nodo es terminal

result = False ##significa que no lo encontro, retorna false

coord = self.end() ##retorna la coord que hace referencia a root

elif(key < node.key):##en los dos elif busco el elemento a borrar

result, node.left,coord = do\_erase(node.left)##si la clave es menor a la clave del ndo me voy a izq

elif(key > node.key):

result, node.right,coord =do\_erase(node.right)##si la clave es mayor al la clave del nodo me voy a der

else: #key ==node.key

result = True ##si lo encontro, devuelve true, indicando que lo borro

coord = Abb.\_Coordinate(node).advance() ##avanzo por la parte ver -------------------------------------------------------------------------------------------------------------------

node = erase\_node(node)##llamo a la funcion borrar

return result,node,coord

##caso 1 no hay ninguno con la key igual a la que queremos eliminar

##caso 2 el nodo a eliminar tiene un subarbol descendiente

##caso 3 el nodo a eliminar tiene 2 sub arboles descendientes

def erase\_node(node):##esta funcion es para buscar la clave

parent = node.parent

##caso1 o caso 2

if (node.left is None): ##si el hijo izq es none

node=node.right ##me voy por la derecha

elif (node.right is None):##viseversa de lo de arriba

node=node.left

else:##cuando tanto el nodo izq como der son terminales, tengo qur extraer

#caso 3

node = extract\_maximum\_from(node) ##esta funcion se queda con el mayor de la rama izquierda

##que sera el que reemplazara el nodo borrado

assign\_parent(node, parent)

self.\_len -=1

return node

def extract\_maximum\_from(node):##-------------------------------------------------------------------------------------no entiendo que hace

prev = None ##me guardo el anterior

maximum= node.left ##me voy a la izquierda del nodo anterior

##y despues me voy todo a la derecha, para buscar el mas grande del sub arbol izquierdo

while maximum.right is not None: #me voy a la derecha

prev=maximum ##me guardo el padre de max

maximum=maximum.right ##"iteramos" sobre el arbol

assign\_parent(maximum,node.parent) ##enganches

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

def assign\_parent(node, parent):##funcion que asigna el padre a un nodo

if node is not None: ##mientras no sea un nodo terminal

node.parent = parent ##el padre del nodo es el padre pasado por parametro

result, self.\_root.left, coord=do\_erase(self.\_root.left)

## el result es un bool, dice si lo encontro o no

##el resto es lo mismo que explique arriba

##esto se hace para emular el pasaje por referencia ya que python lo hace de forma copia 'de' referencia

##yo quiero MODIFICAR el parametro que me pasan

return result,coord

def copy(self):##el nombre lo dice todo, es mejor siendo recursivo, porque te ayudan las pilas de activacion, sino tendriamos que guaradar

def do\_copy(node, parent):

if (node is None):##si es un nodo terminal

new\_node=None ## hace el arbol

else:

new\_node=Abb.\_Node(node.key, node.value,parent) ##aca creo un nuevo nodo del arbol pasandole la clave del nodo, el valor y su padre

new\_node.left = do\_copy(node.left,new\_node)##aca hago un doble llamado recursivo, esto es porque tengo que copiar tanto la rama izq como der

new\_node.right = do\_copy(node.right,new\_node)

return new\_node

##esto se ejecuta primero

new\_tree=Abb()##creo un arbol

new\_tree.\_root.left= do\_copy(self.\_root.left, new\_tree.\_root)

##el nodo siguiente de root de newTree es do\_copy(le paso el nodo siguiente de root de mi arbol que quiero copiar, la raiz del arbol que quiero copiar)

new\_tree.\_len = self.\_len ##copio el len

return new\_tree##retorno el nuevo arbol

def clear(self):

self.\_root.left = None

self.\_len = 0

#estas dos estan de mas, no es necesario en el parcial

##se usa: if (CLAVE in DICCIONARIO)

##con diccionario siendo el nombre del arbol

def \_\_contains\_\_(self,key):##retorna si contiene la clave o no

return self.find(key) != self.end()

##se usa: arbol[clave]

def \_\_getitem\_\_(self,key):

p=self.find(key)

if (p == self.end()):

raise KeyError(key)

return p.value

##CLASE COORDENADA----------------------------------------------------------------------------------------------------------------

class \_Coordinate():

\_\_slots\_\_ = ['\_node']## atributos de la cordenada

def \_\_init\_\_(self, node=None): ##inicializamos la coord

if isinstance(node,Abb.\_Coordinate):##si es una cordenada la que recibo por parametro en el init

self.\_node = node.\_node ##obtengo el nodo de esa supuesta coord

else:

self.\_node = node ##si solo recibo un nodo, lo asigno directamente

@property

def key(self): ##me permite ver la clave

return self.\_node.key

@property

def value(self): ##me permite ver el valor

return self.\_node.value

@value.setter

def value(self, value):##redefinir la asignacion del valor

self.\_node.value = value##yo puedo modificar el Valor pero NUNCA la clave

def advance(self): ##funcion que sirve para que la coord apunte a otro dato a medida que avanza

node = self.\_node

if node.right is not None: ##si el nodo que tengo a la derecha no es terminal

node = \_minimum\_node(node.right) ##me voy al mas chico de la rama derecha

else: ##sino

while node.parent is not None: #mientras el padre del nodo no sea none

prev = node ##aca lo que hago es ir subiendo padre por padre

node = node.parent ##hasta saber de donde vengo

if node.right is not prev: ##si el nodo de donde vengo no es prev, significa que estoy en la izquierda

break ##corta

self.\_node = node

return self

##si vuelvo por la izquierda, debo retornarme a mi y despues siempre hacia la derecha

##si vuelvo por la derecha, el padre ya fue procesado, por lo que hay que seguir el "linaje" de padres

def next(self):

return Abb.\_Coordinate(self.\_node).advance()

def retreat(self): #y el retreat hace lo mismo que el advance pero al reves

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

def prev(self):

return Abb.\_Coordinate(self.\_node).retreat()

def \_\_eq\_\_(self, other):##pregunta si dos coordenadas son las mismas

return self.\_node is other.\_node

def \_\_repr\_\_(self):

if hasattr(self.\_node, 'key'):

return 'Coordinate({{key: {}, value: {}}})'.format(

self.\_node.key, self.\_node.value)

else:

return 'end()'

def \_\_eq\_\_(self, other):

p=self.begin()##primero creo las variables auxiliares

q=other.begin()

while(p!= self.end() and q!=other.end()): ##mientras los dos no lleguen al final del arbol

if(p.key != q.key or p.value != q.value):##si la clave de uno es diferente del otro o si los valores tambien son diferentes

return False##retorna falso

p.advance()##sino sigo iterando por cada uno

q.advance()

return p==self.end() and q==other.end()##y al final comparo el final de cada arbol y retorno si son iguales o no

def eq\_recursivo(self, other):

def do\_eq(p , q):

if(p is None or q is None):

if(p is None and q is None):

return True

else:

return False

if(p.key != q.key or p.value != q.value):

return False

if p.right is not None and q.right is not None:

do\_eq(p.right,q.right)

if p.left is not None and q.left is not None:

do\_eq(p.left,q.left)

return do\_eq(self.\_root.left,other.\_root.left)

##implementar otra forma para practicar

#@ver esto !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

def advance(self): ##funcion que sirve para que la coord apunte a otro dato a medida que avanza

node = self.\_node

if node.right is not None: ##si el nodo que tengo a la derecha no es terminal

node = \_minimum\_node(node.right) ##me voy al mas chico de la rama derecha

else: ##sino

while node.parent is not None: #mientras el padre del nodo no sea none

prev = node ##aca lo que hago es ir subiendo padre por padre

node = node.parent ##hasta saber de donde vengo

if node.right is not prev: ##si el nodo de donde vengo no es prev, significa que estoy en la izquierda

break ##corta

self.\_node = node

return self

##si vuelvo por la izquierda, debo retornarme a mi y despues siempre hacia la derecha

##si vuelvo por la derecha, el padre ya fue procesado, por lo que hay que seguir el "linaje" de padres

##para que el arbol sea iterable-------------------------------------------------------------------------------------

def items(self):

pos = self.begin()

end = self.end()

while pos != end:

yield pos.key, pos.value

pos.advance()

def keys(self):

for key, \_ in self.items():

yield key

def values(self):

for \_, value in self.items():

yield value

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

return self.keys()

#--------------------------------------------------------------------------------------------------------------------

def \_\_repr\_\_(self):

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

def \_\_str\_\_(self):

#Funcion que imprime el arbol como se debe

def calculate\_placement(node, level):

if node is None:

return 0

nonlocal count

m1 = calculate\_placement(node.left, level + 1)

placements.append((level, count, node))

count += 1

m3 = calculate\_placement(node.right, level + 1)

return max(m1, len(str(node.key)), m3)

count = 0

placements = []

key\_len = calculate\_placement(self.\_root.left, 0) + 2

lines = []

prev\_level = -1

for level, pos, node in placements:

i = 2 \* level

while len(lines) <= i:

lines.append('')

skip = ' ' \* (pos \* key\_len - len(lines[i]))

lines[i] += skip + '[{:^{}}]'.format(node.key, key\_len - 2)

if prev\_level != -1:

if prev\_level < level:

i = 2 \* prev\_level + 1

skip = ' ' \* (pos \* key\_len - len(lines[i]))

c = '\\'

else:

i = 2 \* level + 1

skip = ' ' \* (pos \* key\_len - len(lines[i]) - 1)

c = '/'

lines[i] += skip + '{:>{}}'.format(c, key\_len // 2)

prev\_level = level

return '\n'.join(lines)

## AVL

from dataclasses import dataclass

from typing import Any, Union

##implementacion AVL

## basicamente iteramos el arbol hasta encontrar el nodo que tiene el valor menor

def \_minimum\_node(node): ##la funcion que retorna el minimo nodo

if node is not None: ##si el arbol no esta vacio

while (node.left is not None): ##mientras el nodo.left no sea un nodo terminal

node = node.left ##iteramos

return node

## basicamente iteramos el arbol hasta encontrar el nodo que tiene el mayor valor

def \_maximum\_node(node): ##la funcion que retorna el nodo mayor

if node is not None: ##si el arbol no esta vacio

while (node.right is not None): ##mientras el nodo.right no sea un nodo terminal

node = node.right ##iteramos

return node

class Avl():

@dataclass

class \_Node:

key: Any ##la clave 'se puede comparar'

value: Any ##el valor

height: int ##este representara la altura de cada nodo

parent: Union['\_Node','\_Root'] = None ## el union es porque el padre puede ser o un nodo o la raiz

left: '\_Node' = None ##inicializo los nodos

right: '\_Node' = None

@dataclass

class \_Root:

left: '\_Node' = None ##inicializo los nodos

right: '\_Node' = None

parent: '\_Node' = None ##el nodo que simboliza el padre, si el padre es None significa que ese nodo es Root

\_\_slots\_\_ = ['\_root','\_len'] ##atributos

def \_\_init\_\_(self, iterable=None):

self.\_root = self.\_Root()

self.\_len = 0

if iterable is not None:

for key, value in iterable:

self.insert(key, value)

def is\_empty(self):

return self.\_root.left is None ## aca se usa el nodo izquierdo como nodo base? es como el prev de la pila que implemente

##si el nodo izquierdo de la raiz es none, es porque esta vacio

def \_\_len\_\_(self):

return self.\_len ##devuelve la cantidad de nodos

def begin(self): #devuelve la coord que hace referencia al nodo mas chiquito (o mas izqu del arbol)

return Avl.\_Coordinate(\_minimum\_node(self.\_root))

def begin\_preorden(self): #devuelve la coord que hace referencia al nodo mas chiquito (o mas izqu del arbol)

return Avl.\_Coordinate(self.\_root.left)

def begin\_postorden(self): #devuelve la coord que hace referencia al nodo mas chiquito (o mas izqu del arbol)

return Avl.\_Coordinate(\_minimum\_node(self.\_root))

def end(self): #devuelve la coord que hace referencia al ultimo nodo

return Avl.\_Coordinate(self.\_root) ##devuelve solo root, porque no se pone maximum?-----------------------------------------------

def minimum(self):

return Avl.\_Coordinate(\_minimum\_node(self.\_root)) ##retorna una coord que hace referecia al nodo minmo

def maximum(self):

return Avl.\_Coordinate(\_maximum\_node(self.\_root.left)) ##retorna una coord que hace referecia al nodo maximo

def altura(self):

def do\_altura(node):

if node is None: #si el nodo es None la altura de este es -1 (se sigue explicando debajo del "return 1 + max...")

return 0

else: # la altura será la altura máxima entre el subarbol izquierdo y el derecho... + 1 ya que tengo que considerar la altura del propio nodo

return 1 + max(do\_altura(node.left), do\_altura(node.right))

# en el caso de que un nodo tenga None tanto en la hoja hizquierda como en la derecha, al retornar -1, la altura de ese nodo queda igual a cero

altura = do\_altura(self.\_root.left) #inicio a contar la altura desde la raíz del arbol

return altura #retorno la altura

##find recursivo, aporta mas legibilidad al codigo

def find(self, key): ##le paso la clave

def do\_find(node): ## funcion interna

if (node is None): ##si node es none

return self.end() ##retorna la coord que hace referencia a root indicando que no lo encontro

elif (key < node.key): ##si la clave es mas chica que la clave del nodo actual

return do\_find(node.left) ##me voy a buscar la clave por el lado izquierdo es decir, lo que busco es mas chiquito que el actual

elif (key > node.key): ##si la clave es mas grande que la clave del nodo actual

return do\_find(node.right) ##me voy a la parte derecha del arbol, es decir, la clave es mas grande que la actual

else: ##hasta que eventualmente key==node.key, por lo tanto

return Avl.\_Coordinate(node) ##retorno la Coord que hace referencia al nodo

return do\_find(self.\_root.left) #caso recursivo

##con cual nos quedamos?------------------------------------------------------------------------------------------------------------------

##el find es mejor iterativo que recursivo, porque el recursivo consume mucho mas recursos

##ademas te ahorras las pilas de activacion

def find\_iterativo(self, key): #hace lo mismo de la funcion de arriba

node=self.\_root.left #en logica es parecida

while True: #lo que cambia es que hace un bucle

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 Avl.\_Coordinate(node)

##este find lo hice con el proposito de saber si esta o no la clave a buscar

def find\_boolean(self, key): ##le paso la clave

def do\_find(node): ## funcion interna

if (node is None): ##si node es none

return False ##retorna False indicando que no lo encontro

elif (key < node.key): ##si la clave es mas chica que la clave del nodo actual

return do\_find(node.left) ##me voy a buscar la clave por el lado izquierdo es decir, lo que busco es mas chiquito que el actual

elif (key > node.key): ##si la clave es mas grande que la clave del nodo actual

return do\_find(node.right) ##me voy a la parte derecha del arbol, es decir, la clave es mas grande que la actual

else: ##hasta que eventualmente key==node.key, por lo tanto

return True ##retorna True indicando que si la encontro

return do\_find(self.\_root.left) #caso recursivo

def len\_recursivo(self):

contador=0

def do\_len(node):

if(node == None):

return 0

return 1 + do\_len(node.right) + do\_len(node.left)

contador=do\_len(self.\_root.left)

return contador

##no es el mismo del erase, pero hace lo mismo (xq el del erase es interno)

def \_assign\_parent(self, node, parent):

if node is not None: ##mientras no sea un nodo terminal

node.parent = parent ##el padre del nodo es el padre pasado por parametro

def \_update\_height(self, node): ##funcion que actualiza la altura del nodo

def max(h\_left, h\_right): # Retorna la rama mas larga (izq o der)

if h\_left > h\_right:

return h\_left

else:

return h\_right

left\_height = 0

if node.left is not None:

left\_height = node.left.height

right\_height = 0

if node.right is not None:

right\_height = node.right.height

node.height = 1 + max(left\_height, right\_height)

def \_balance\_tree(self, root):

def rotate\_right(root): ##root seria el nodo que pasan por parametro

left\_tree = root.left ##left\_tree es la variable auxiliar que apunta al hijo izq de root

root.left = left\_tree.right ##ahora pongo al hijo derecho del left de root como hijo iquierdo de root

self.\_assign\_parent(root.left, root) ##le asigno el padre

left\_tree.right = root ##ahora el hijo derecho del hijo izquirdo es root, por ende ahora root seria el que fue alguna vez el hijo izq de root

left\_tree.parent = root.parent ##le asigno el padre

root.parent = left\_tree

root = left\_tree ##ahora root es el hijo izquierdo del root original

self.\_update\_height(root.right) ##actualizo las alturas

self.\_update\_height(root)

return root ##retorno root

def rotate\_left(root): ##esto hace lo mismo que arriba pero con roles invertidos

right\_tree = root.right ##una variable auxiliar que guarda el hijo derecho de root

root.right = right\_tree.left ##pongo al hijo izquierdo de root como el hijo derecho

self.\_assign\_parent(root.right, root) ##le asigno el padre

right\_tree.left = root ##ahora el hijo izquierdo de auxiliar es la root original

right\_tree.parent = root.parent ##ahora el padre de mi root original es el auxiliar

root.parent = right\_tree

root = right\_tree ##por ultimo root ahora es auxiliar

self.\_update\_height(root.left) ##actualizo las alturas

self.\_update\_height(root)

return root

##saca la cuanta del FE entre el hijo izquierdo y dercho

def balance\_factor(node): ##FE = height(left) - height(right)

## los valores que puede retornar son: -2 -1 0 1 2

## si retorna -2 o 2 es porque esta desequilibrado

factor = 0

if (node is not None): ##si no es una hoja

if(node.left is not None): ##si el hijo izquierdo no es teminal

factor += node.left.height ##le sumo a factor su altura

if(node.right is not None): ##si el hijo derecho no es terminal

factor -= node.right.height ##le RESTO su altura

return factor ##en caso de que node sea None return 0

bf = balance\_factor(root) ##root seria el nodo desequilibrado

if (bf == 2): ##si el factor de balance es 2 positivo es porque tengo que hacer una rotacion simple derecha

if (balance\_factor(root.left) == -1): ##pero si el bf es -1 tengo que hacer una rotacion simple a izquierda primero

root.left = rotate\_left(root.left) ##rotacion a izquierda

root = rotate\_right(root) ##rotacion a derecha

elif (bf == -2): ##si el bf es 2 negativo , debo hacer una rotacion simple a izquierda

if (balance\_factor(root.right) == 1): ##pero si el bf es +1 tengo que hacer una rotacion simple a derecha

root.right = rotate\_right(root.right) ##rotacion a derecha

root = rotate\_left(root) ##rotacion a izquierda

else:

self.\_update\_height(root) ##actualizo las alturas

return root

##lo bueno de hacerlo recursivo es porque hace el enganche automatico

##a traves de las pilas de activacion

def insert(self, key, value=None): ##funcion que inserta

def do\_insert(node, parent): ##funcion interna

if (node is None): ##

node = self.\_Node(key, value, 1,parent) ##aca se genera el nodo con la clave, el valor y el padre

coord=Avl.\_Coordinate(node) ## coord es la coord que hace referencia al nodo de arriba

self.\_len+=1 ##le sumo 1 al len porque el len implementado es de O(1)

elif(key < node.key): ## aca hace lo mismo que en el find pero con la diferencia

node.left, coord = do\_insert(node.left, node) ## que en vez de pasarle solo el nodo le paso el padre tambien

elif(key > node.key):

node.right, coord= do\_insert(node.right,node)

else: #key == node.key

node.value = value ##le asigno el valor None

coord = Avl.\_Coordinate(node) ##ya lo explique un par de comentarios arriba

node=self.\_balance\_tree(node)

return node, coord ##el return se ejecuta al terminar siempre

#lo que se logra con esto es hacer el enganche automatico

##self.\_root.left, coord=do\_insert(self.\_root.left, self.\_root)##primero se ejecuta esto, le paso por parametro el primer nodo y el padre del nodo actual que es la raiz

self.\_root.left, coord = do\_insert(self.\_root.left, self.\_root)

## para luego empezar a recorrer el arbol

return coord ## por ultimo retorno la coord del nodo que acabo de insertar

def erase(self, key):

def do\_erase(node):

if(node is None): ##si en nodo es terminal

result = False ##significa que no lo encontro, retorna false

coord = self.end() ##retorna la coord que hace referencia a root

elif(key < node.key): ##en los dos elif busco el elemento a borrar

result, node.left,coord = do\_erase(node.left) ##si la clave es menor a la clave del ndo me voy a izq

elif(key > node.key):

result, node.right,coord =do\_erase(node.right) ##si la clave es mayor al la clave del nodo me voy a derecha

else: #key ==node.key

result = True ##si lo encontro, devuelve true, indicando que lo borro

coord = Avl.\_Coordinate(node).advance() ##avanzo

node = erase\_node(node) ##llamo a la funcion borrar

return result,node,coord

##existen 3 casos

##caso 1 no hay ninguno con la key igual a la que queremos eliminar

##caso 2 el nodo a eliminar tiene un subarbol descendiente

##caso 3 el nodo a eliminar tiene 2 sub arboles descendientes

def erase\_node(node): ##esta funcion es para buscar la clave

parent = node.parent

##caso1 o caso 2

if (node.left is None): ##si el hijo izq es none

node=node.right ##me voy por la derecha

elif (node.right is None): ##viceversa de lo de arriba

node=node.left

else: ##cuando tanto el nodo izq como der son terminales, tengo que extraer el maximo

#caso 3

node = extract\_maximum\_from(node) ##esta funcion se queda con el mayor de la rama izquierda

##que sera el que reemplazara el nodo borrado

assign\_parent(node, parent)

self.\_len -=1

return node

def extract\_maximum\_from(node):

prev = None ##me guardo el anterior

maximum= node.left ##me voy a la izquierda del nodo anterior

##y despues me voy todo a la derecha, para buscar el mas grande del sub arbol izquierdo

while maximum.right is not None: #me voy a la derecha

prev=maximum ##me guardo el padre de max

maximum=maximum.right ##"iteramos" sobre el arbol

assign\_parent(maximum,node.parent) ##enganches

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

def assign\_parent(node, parent): ##funcion que asigna el padre a un nodo

if node is not None: ##mientras no sea un nodo terminal

node.parent = parent ##el padre del nodo es el padre pasado por parametro

result, self.\_root.left, coord=do\_erase(self.\_root.left)

self.\_root.left=self.\_balance\_tree(self.\_root.left)

## el result es un bool, dice si lo encontro o no

##el resto es lo mismo que explique arriba

##esto se hace para emular el pasaje por referencia ya que python lo hace de forma copia 'de' referencia

##yo quiero MODIFICAR el parametro que me pasan

return result,coord

##el mejor es iterativo, pero no lo implemente...

##lo bueno del copy recursivo es la simpleza del codigo

def copy(self): ##el nombre lo dice todo,en el recursivo te ayudan las pilas de activacion, sino tendriamos que guaradar cada uno

def do\_copy(node, parent):

if (node is None): ##si es un nodo terminal

new\_node=None ## hace el arbol

else:

new\_node=Avl.\_Node(node.key, node.value,node.height,parent) ##aca creo un nuevo nodo del arbol pasandole la clave del nodo, el valor y su padre

new\_node.left = do\_copy(node.left,new\_node) ##aca hago un doble llamado recursivo, esto es porque tengo que copiar tanto la rama izq como der

new\_node.right = do\_copy(node.right,new\_node)

return new\_node

##esto se ejecuta primero

new\_tree=Avl() ##creo un arbol

new\_tree.\_root.left= do\_copy(self.\_root.left, new\_tree.\_root)

##el nodo siguiente de root de newTree es do\_copy(le paso el nodo siguiente de root de mi arbol que quiero copiar, la raiz del arbol que quiero copiar)

new\_tree.\_len = self.\_len ##copio el len

return new\_tree##retorno el nuevo arbol

def clear(self):

self.\_root.left = None

self.\_len = 0

#estas dos estan de mas, no es necesario en el parcial

##se usa: if (CLAVE in DICCIONARIO)

##con diccionario siendo el nombre del arbol

def \_\_contains\_\_(self,key):##retorna si contiene la clave o no

return self.find(key) != self.end()

##se usa: arbol[clave]

def \_\_getitem\_\_(self,key):

p=self.find(key)

if (p == self.end()):

raise KeyError(key)

return p.value

##CLASE COORDENADA----------------------------------------------------------------------------------------------------------------

class \_Coordinate():

\_\_slots\_\_ = ['\_node'] ## atributos de la cordenada

def \_\_init\_\_(self, node=None): ##inicializamos la coord

if isinstance(node,Avl.\_Coordinate): ##si es una cordenada la que recibo por parametro en el init

self.\_node = node.\_node ##obtengo el nodo de esa supuesta coord

else:

self.\_node = node ##si solo recibo un nodo, lo asigno directamente

@property

def key(self): ##me permite ver la clave

return self.\_node.key

@property

def value(self): ##me permite ver el valor

return self.\_node.value

@value.setter

def value(self, value):##redefinir la asignacion del valor

self.\_node.value = value##yo puedo modificar el Valor pero NUNCA la clave

def advance(self): ##funcion que sirve para que la coord apunte a otro dato a medida que avanza

node = self.\_node

if node.right is not None: ##si el nodo que tengo a la derecha no es terminal

node = \_minimum\_node(node.right) ##me voy al mas chico de la rama derecha

else: ##sino

while node.parent is not None: #mientras el padre del nodo no sea none

prev = node ##aca lo que hago es ir subiendo padre por padre

node = node.parent ##hasta saber de donde vengo

if node.right is not prev: ##si el nodo de donde vengo no es prev, significa que estoy en la izquierda

break ##corta

self.\_node = node

return self

##si vuelvo por la izquierda, debo retornarme a mi y despues siempre hacia la derecha

##si vuelvo por la derecha, el padre ya fue procesado, por lo que hay que seguir el "linaje" de padres

def advance\_preorden(self): #preorden ya probado ---- si lo vas a probar recorda cambiar begin() que retorne self.\_root.left

node = self.\_node

prev = node.parent

if node.left is not None:

node = node.left

elif node.right is not None:

node = node.right

else:

while node.parent is not None:

prev = node

node = node.parent

if node.right is not prev and node.right is not None:

node = node.right

break

self.\_node = node

return self

def advance\_postorden(self):

def buscar(node):

while node.right is not None:

node = node.right

if node.left is not None:

node = \_minimum\_node(node)

break

return node

node = self.\_node

if node.parent is not None:

prev = node

node = node.parent

if node.right is not prev and node.right is not None:

node = buscar(node)

self.\_node = node

return self

def next(self):

return Avl.\_Coordinate(self.\_node).advance()

def retreat(self): #y el retreat hace lo mismo que el advance pero al reves

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

def prev(self):

return Avl.\_Coordinate(self.\_node).retreat()

def \_\_eq\_\_(self, other): ##pregunta si dos coordenadas son las mismas

return self.\_node is other.\_node

def \_\_repr\_\_(self):

if hasattr(self.\_node, 'key'):

return 'Coordinate({{key: {}, value: {}}})'.format(

self.\_node.key, self.\_node.value)

else:

return 'end()'

def \_\_eq\_\_(self, other):

p=self.begin() ##primero creo las variables auxiliares

q=other.begin()

while(p!= self.end() and q!=other.end()): ##mientras los dos no lleguen al final del arbol

if(p.key != q.key or p.value != q.value): ##si la clave de uno es diferente del otro o si los valores tambien son diferentes

return False ##retorna falso

p.advance() ##sino sigo iterando por cada uno

q.advance()

return p==self.end() and q==other.end() ##y al final comparo el final de cada arbol y retorno si son iguales o no

##para que el arbol sea iterable-------------------------------------------------------------------------------------

def items(self):

pos = self.begin()

end = self.end()

while pos != end:

yield pos.key, pos.value

pos.advance()

def keys(self):

for key, \_ in self.items():

yield key

def values(self):

for \_, value in self.items():

yield value

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

return self.keys()

#--------------------------------------------------------------------------------------------------------------------

def \_\_repr\_\_(self):

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

def \_\_str\_\_(self):

#Funcion que imprime el arbol como se debe

def calculate\_placement(node, level):

if node is None:

return 0

nonlocal count

m1 = calculate\_placement(node.left, level + 1)

placements.append((level, count, node))

count += 1

m3 = calculate\_placement(node.right, level + 1)

return max(m1, len(str(node.key)), m3)

count = 0

placements = []

key\_len = calculate\_placement(self.\_root.left, 0) + 2

lines = []

prev\_level = -1

for level, pos, node in placements:

i = 2 \* level

while len(lines) <= i:

lines.append('')

skip = ' ' \* (pos \* key\_len - len(lines[i]))

lines[i] += skip + '[{:^{}}]'.format(node.key, key\_len - 2)

if prev\_level != -1:

if prev\_level < level:

i = 2 \* prev\_level + 1

skip = ' ' \* (pos \* key\_len - len(lines[i]))

c = '\\'

else:

i = 2 \* level + 1

skip = ' ' \* (pos \* key\_len - len(lines[i]) - 1)

c = '/'

lines[i] += skip + '{:>{}}'.format(c, key\_len // 2)

prev\_level = level

return '\n'.join(lines)

# CLASE FECHA

TAD Nombre\_del\_TAD:

Fecha<a>

Igualdad Observacional:

Si a y b son dos fechas

a es igual a b si se cumple que:

contienen el mismo dia, mes, año

con el mismo formato DDMMAAAA

Usa:

Natural, Bool, <a>

Parametro Formal:

<a>

Género:

Fecha<a>

Observadores Básicos:

fechaActual() -> Fecha<a> F

getDia(Fecha<a> F) -> Natural

getMes(Fecha<a> F) -> Natural

getAño(Fecha<a> F) -> Natural

es\_bisiesto(getAño()) -> Bool

Generadores:

nueva\_fecha() -> None

nueva\_fecha(Fecha<a> F) -> None

Otras opecaiones:

setFecha(Fecha<a> F) -> None

setDia(Fecha<a> F, DD) -> None

setMes(Fecha<a> F, MM) -> None

setaño(Fecha<a> F, AAAA) -> None

eliminar(Fecha<a> F) -> None

validarFecha(Fecha<a> F) -> Bool

actualizarFecha(Fecha<a> F) -> None

Axiomas:

fechaActual(Fecha<a> F) {retorna la fecha actual F en formato DDMMAAAA}

getDia(Fecha<a> F) {retorna el dia actual de la fecha F en formato DD}

getMes(Fecha<a> F) {retorna el mes actual de la fecha F en formato MM}

getAño(Fecha<a> F) {retorna el año actual de la fecha F en formato AAAA}

es\_bisiesto(getAño()) {retorna -true si el año de la fecha pasada por parametro es bisiesto sino retorna false}

nueva\_fecha() {crea una nueva fecha sin datos}

nueva\_fecha(Fecha<a> F) {crea una nueva fecha F con los datos DDMMAAAA pasados por parametro}

setFecha(Fecha<a> F) {setea la fecha actual con los datos de F}

setDia(Fecha<a> F, DD) {setea los dias DD a la fecha F}

setMes(Fecha<a> F, MM) {setea el mes MM a la fecha F}

setaño(Fecha<a> F, AAAA) {setea el año AAAA a la fecha F}

eliminar(Fecha<a> F) {elimina la fecha F,dejandola vacia}

validarFecha(Fecha<a> F) {comprueba que la fecha F pasada por parametros sea valida}

actualizarFecha(Fecha<a> F) {actualiza la fecha guardada por la pasada por parametro}

Exporta:

Fecha<a>, fechaActual, getDia, getMes, getAño, es\_bisiesto

nueva\_fecha, nueva\_fecha, setDia, setMes, setaño, eliminar

## FECHA

class Fecha():

\_\_slots\_\_ = ['\_meses', '\_values', '\_dia', '\_mes', '\_anio']

def \_\_init\_\_(self, value = 0):

self.\_meses = [None, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]

self.\_values = []

if value != 0:

self.setFecha(value)

def eliminar(self):

self.\_values = []

def esBisiesto(self):

return not self.\_anio % 4 and (self.anio % 100 or not self.\_anio % 400)

def actualizarFecha(self):

self.\_values = []

self.\_values.append(self.\_dia)

self.\_values.append(self.\_mes)

self.\_values.append(self.\_anio)

def validarFecha(self):

if (self.\_mes >= 1 and self.\_mes <= 12):

if self.esBisiesto():

self.\_meses[2] = 29

else:

self.\_meses[2] = 28

assert (self.\_dia <= self.\_meses[self.\_mes] and self.\_dia >= 1), 'Fecha invalida: El dia '+str(self.\_dia)+' se excede del permitido en el mes '+str(self.\_mes)+'(hasta '+str(self.\_meses[self.\_mes])+')'

assert not (self.\_mes > 13 or self.\_mes < 0), 'Fecha invalida: El mes no es valido'

def setFecha(self, value):

assert (len(str(value)) == 8), 'Fecha Invalida, el formato debe ser DDMMAAAA'

self.\_dia=(int(value)//1000000)%100

self.\_mes=(int(value)//10000)%100

self.\_anio=int(value)%10000

self.validarFecha()

self.actualizarFecha()

def setDia(self,dd):

self.\_dia = dd

self.validarFecha()

self.actualizarFecha()

def setMes(self,mm):

self.\_mes = mm

self.validarFecha()

self.actualizarFecha()

def setaño(self,aaaa):

self.\_anio = aaaa

self.validarFecha()

self.actualizarFecha()

def getDia(self):

return self.\_dia()

def getMes(self):

return self.\_mes()

def getAño(self):

return self.\_anio()

def \_\_repr\_\_(self):

return 'La fecha es: ('+ '/'.join(repr(x) for x in self.\_values)+ ')'

def \_\_eq\_\_(self, other):

return self.\_values==other.\_values

# CLASE GRAFO

TAD Grafo<a>

* Igualdad Observacional: Sean A y B dos grafos, serán iguales si: tienen los mismos vértices y las mismas conexiones entre ellos.
* Usa: Natural, None, Bool, Lista<a>.
* Parámetro Formal: a.
* Género: Grafo<a>.
* Observadores Básicos:
  + - * Tamaño(Grafo<a>)🡪Natural.
      * Es\_vacio(Grafo<a>)🡪Bool.
      * Adyacentes(Grafo<a>, a)🡪Lista<a>. Pre{El vértice “a” pertenece al grafo}
      * Es\_adyacente(Grafo<a>, a, a)🡪Bool. Pre{Los vértices “a” pertenecen al grafo}
      * Vértices(Grafo<a>)🡪Lista<a>.
* Generador:
  + - * Vacío()🡪Grafo<a>. Post{El grafo retornado está vacío}
* Otras Operaciones:
  + - * Agregar\_vertice(Grafo<a>,a )🡪None. Post{El grafo no está vacío}
      * Borrar\_vertice(Grafo<a>, a)🡪None. Pre{El vértice pertenece al grafo}
      * Agregar\_arista(Grafo<a>, a, a)🡪None.
      * Borrar\_arista(Grafo<a>, a, a)🡪Bool.
* Axiomas:
  + - * Vacío()🡪Crea un grafos sin elementos.
      * Tamaño(Grafo<a> T)🡪Devuelve la cantidad de vértices del grafo T.
      * Es\_vacio(Grafo<a> T)🡪Retorna Verdadero si T esta vacío, o Falso en caso contrario.
      * Adyacentes(Grafo<a> T, a)🡪Retorna una lista con los adyacentes del vértice “a”.
      * Es\_adyacente(Grafo<a> T, a v, a g)🡪Retorna Verdadero si en T, los vértices “v” y “g” son adyacentes, o Falso en caso contrario.
      * Vértices(Grafo<a> T)🡪Retorna una lista con todos los vértices del grafo T.
      * Agregar\_vertice(Grafo<a> T, a )🡪Agrega el vértice “a” al grafo T.
      * Borrar\_vertice(Grafo<a> T, a)🡪Borra el vértice “a” del grafo T.
      * Agregar\_arista(Grafo<a> T, a v, a g)🡪Agrega una arista entre “v” y “g” en el grafo T.
      * Borrar\_arista(Grafo<a> T, a v, a g)🡪Retorna Verdadero si pudo borrar la arista entre el vértice “v” y “g” en el grafo T, o Falso en caso contrario.
* Exporta:
  + - * Grafo<a>, Vacío, Tamaño, Es\_vacio, Adyacentes, Es\_adyacente, Vértices, Agregar\_vertice, Borrar\_vertice, Agregar\_arista, Borrar\_arista.

## GRAFO CON MATRIZ

class Grafos():

\_\_slots\_\_=["\_vertices","\_aristas"]

def \_\_init\_\_(self,iterable=None):

self.\_vertices=[]

self.\_aristas=[]

def empty(self):

return len(self.\_vertices)==0

def clear(self):

self.\_vertices=[]

self.\_aristas=[]

def \_\_len\_\_(self):

return len(self.\_vertices)

def vertices(self):

return list(range(len(self.\_vertices)))

def get\_vertice(self,indice):

return self.\_vertices[indice]

def set\_vertices(self,indice,valor):

self.\_vertices[indice]=valor

def erase\_item(self,key):

if isinstance(key,tuple):

self.erase\_arista(key[0],key[1])

else:

self.erase\_vertice(key)

def get\_edge(self,from\_,to):

return self.\_aristas[from\_][to]

def erase\_vertice(self,indice):

if 0<=indice<len(self.\_vertices):

del self.\_vertices[indice]

del self.\_aristas[indice]

for edge in self.\_aristas:

del edge[indie]

def erase\_arista(self,\_from,to):

if self.\_aristas[\_from][to] is not None or self.\_aristas[\_from][to]==True:

self.\_aristas[\_from][to]=None

return True

return False

def add\_vertice(self,value):

indice=len(self.\_vertices)

self.\_vertices.append(value)

for edges in self.\_aristas:

edges.append(None)

self.\_aristas.append([None]\*(indice+1))

return indice #Retorna donde guardo el elemento

def add\_arista(self,\_from,to,peso=None):

self.\_aristas[\_from][to]=peso

def \_\_eq\_\_(self,other):

return self.\_vertices==other.\_vertices and self.\_aristas==other.\_aristas

def es\_adyacente(self,\_from,to):

return self.\_aristas[\_from][to] is not None

def adyacentes(self,\_from):

resultados=[]

for i in range(len(self.\_vertices)):

if self.\_aristas[\_from][i] is not None:#Se fija si el camino desde donde le indique hasta cada uno de los vertices es != de None

resultados.append(i)

return resultados

def copy(self):

result=Grafos()

result.\_vertices=self.\_vertices.copy()

result.\_aristas=[]

for aristas in self.\_aristas:

result.\_aristas.append(aristas.copy())

return result

def \_\_repr\_\_(self):

result='( '

for v in self.vertices():

for w in self.adyacentes(v):

result+='{0}-({2})->{1}, '.format(v, w,self.get\_edge(v,w))

return result + ') :Grafos '