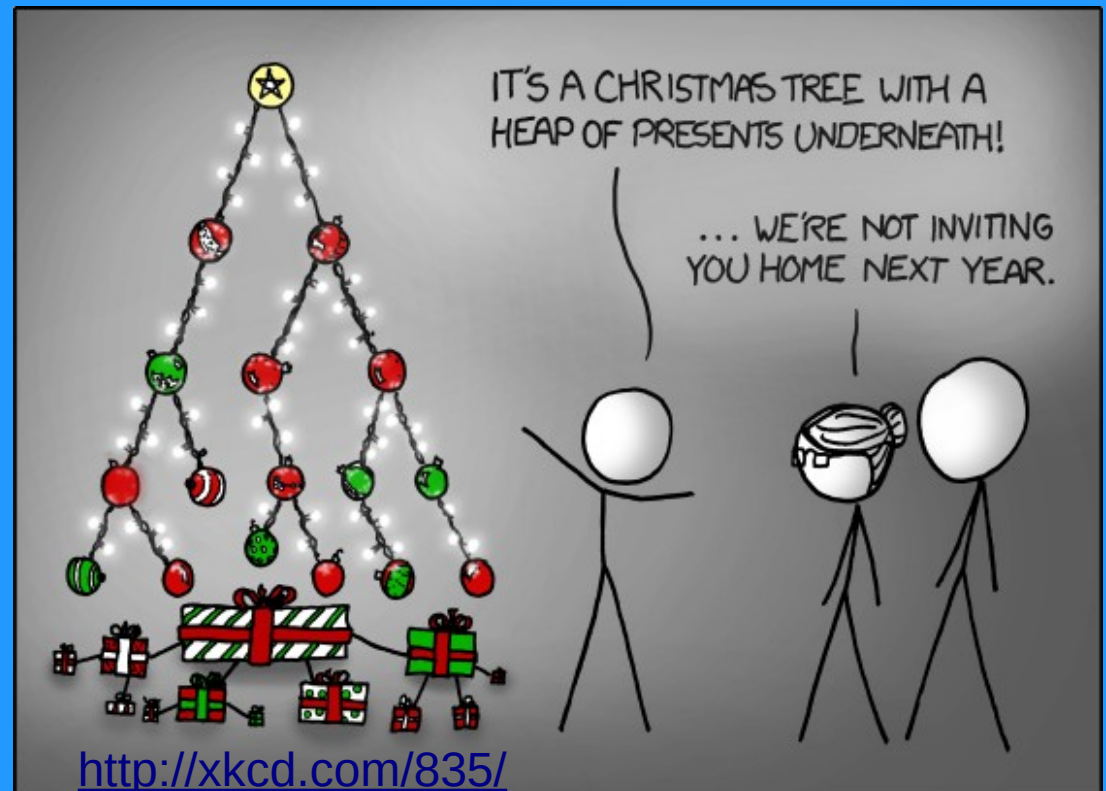


Estructuras de Datos

2020



TAD:

Implementación

Otra Implementación para la Pila...

**Estructura Interna
que define a la Pila**

TAD Pila dinámica

Vamos a implementar la Pila o Stack usando nodos enlazados. Cada nodo lo implementaremos como una instancia de un dataclass.

Un nodo “va a tener” un valor y su siguiente.
Vamos a **usar dataclass de Python**

Data classes **(dataclass)**

dataclass

Es una clase pensada para almacenar datos, similar a un registro (como los **structs** vistos en C++)

Posee campos que representan los datos lógicos que queremos asociar.

dataclass

Los tipos que se definen con **dataclass** permiten agrupar información relacionada en una sola entidad u objeto.

A esta entidad se la suele denominar **registro** y permite definir nuevos tipos de datos que están compuestos por valores de tipos más simples.

dataclass

Es necesario importar el decorador **dataclass** desde el módulo **dataclasses** para poder crear registros:

```
>>> from dataclasses import dataclass
```

Una vez importada este decorador, podemos definir **registros**.

dataclass

Para **definir** un nuevo tipo de registro, usamos el decorador **dataclass** sobre una clase definida de la siguiente manera:

```
from dataclasses import dataclass
from typing import Any # Any representa "cualquier tipo"

@dataclass # decorador para definir una data class
class Persona:
    nombre: str
    apellido: str
    dni: int
    sexo: Any
    madre: 'Persona' = None
    padre: 'Persona' = None
```

Como no podemos usar el tipo `Persona` (porque aún no se terminó de definir) usamos un string con el nombre del tipo

dataclass

Usamos el **nuevo tipo** definido para **crear** objetos que contendrán **valores** en cada uno de los **campos especificados**:

```
>>> p = Persona('Benjamín', 'López', 30474187, 'M')
>>> p
Persona(nombre='Benjamín', apellido='López', dni=30474187,
sexo='M', madre=None, padre=None)
>>> p.nombre
'Benjamín'
>>> p.apellido
'López'
>>> p.dni
30474187
```

Es importante tener en cuenta que: los tipos son solo una documentación...Python no hace chequeos referidos a los tipos

dataclass

Ahora vamos a definir la estructura de nuestra pila usando dataclass:

```
from dataclasses import dataclass
from typing import Any # Any representa "cualquier tipo"

class Stack():
    @dataclass # decorador para definir una data class
    class _Node:
        value: Any
        prev: '_Node'
        __slots__ = ['_top']
```

El atributo `_top` es el puntero inicial de la lista que va a contener los elementos de la pila

Generadores...

Especificación. Pila Dinámica

Generadores

`vacía() → Pila<a>`

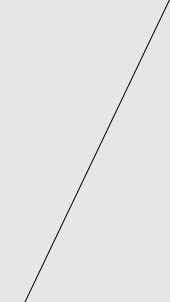
`{Post: La pila retornada esta vacía}`

`a_partir_de(Secuencia<a>)→ Pila<a>`

`{Post: La pila contiene encolados los elementos de la secuencia recibida}`

Implementación. Generadores

```
def __init__(self, iterable=None):  
    self._top = None  
    if iterable is not None:  
        for value in iterable:  
            self.push(value)
```



Para crear una lista vacía
ponemos el puntero nulo.

Observadores

Especificación. Pila Dinámica

observadores básicos

`tamaño(Pila<a>) → Natural`

`es_vacia(Pila<a>) → Bool`

`tope(Pila<a>) → a`

`{Pre: la pila tiene al menos un elemento}`

Implementación. Tamaño?

```
class Stack():
```

```
    ...
```

```
    def __len__(self):  
        n = 0  
        node = self._top  
        while node is not None:  
            node = node.prev  
            n += 1  
        return n
```

¿Qué Orden tiene esta operación?

¿Podemos mejorarlo?

Implementación. es_vacíá?

```
class Stack:
```

```
    ...
```

```
    def is_empty(self):
```

```
        return self._top is None
```

¿Qué Orden tiene esta operación?

¿Podemos mejorarlo?

Implementación. Tope?

```
class Stack():  
    ...  
    @property  
    def top(self):  
        assert not self.is_empty(), 'Pila vacía'  
        return self._top.value
```

Especificación. Pila Dinámica

Otras Operaciones

`apilar(Pila<a>, a) → None`

`{Pos: la pila no esta vacía}`

`desapilar(Pila<a>) → a`

`{Pre: la pila tiene al menos un elemento}`

Implementación. Apilar Desapilar

```
class Stack():  
    ...  
    def push(self, value):  
        self._top = Stack._Node(value, self._top)  
  
    def pop(self):  
        assert not self.is_empty(), 'Pila vacía'  
        value = self._top.value  
        self._top = self._top.prev  
        return value
```

Implementación. copy()

```
def copy(self):  
    new_stack = Stack()  
    if not self.is_empty():  
        node = self._top  
        new_node = Stack._Node(node.value, None)  
        new_stack._top = new_node  
        while node.prev is not None:  
            node = node.prev  
            new_node.prev = Stack._Node(node.value, None)  
            new_node = new_node.prev  
    return new_stack
```

Implementación. del()

```
class Stack():  
    ...  
    def clear(self):  
        self._top = None
```

Todos los objetos que dejamos sin referenciar serán automáticamente liberados de la memoria por Python.

Redefinimos operadores..

Especificación. Pila Dinámica

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**.

Implementación. Comparación

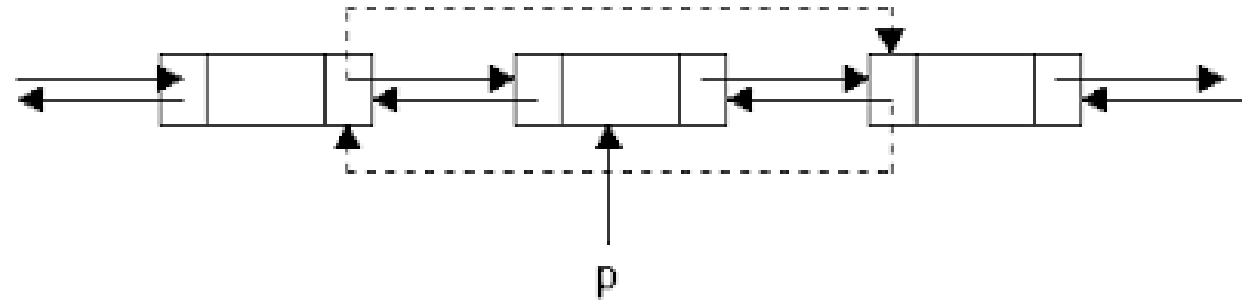
```
def __eq__(self, other):  
    x = self._top  
    y = other._top  
    while x is not None and y is not None:  
        if x.value != y.value:  
            return False  
        x = x.prev  
        y = y.prev  
    return x is None and y is None
```

Implementación. Representación

```
def __repr__(self):  
    values = []  
    node = self._top  
    while node is not None:  
        values.insert(0, node.value)  
        node = node.prev  
    return 'Stack([' + ', '.join(repr(x) for x in values) + '])'
```

**Y si queremos
implementar ahora una
lista doblemente
enlazada...**

TAD. Lista doble



Especificación



Implementación

Listas enlazadas...

Especificación. Lista

El concepto de lista es bastante intuitivo.
Encontramos varios ejemplos en la vida cotidiana:



	A	B
1	Lista de Alumnos	
2		
3	Usuario	Nombre del alumno
4	mes alu aba mar	Abarca Murga Marcelino
5	mes alu agu mig	Aguayo Bedolla Miguel Angel
6	mes alu ale rod	Alejandre Martinez Rodrigo
7	CTC_mipequeñoalu	alumno mi pequeño
8	mes alu alv ang	Alvarez Acosta Angel
9	mes alu are san	Arevalo Hernandez Sandra Patricia
10	mes alu arg min	Argon Herrera Minerva Alejandra
11	mes alu arr mar	Arreola Silva Maria Josefina
12	mes alu art jai	Arteaga Marquez Jaime
13	mes alu bec lui	Becerri Reynoso Luis Carlos
14	mes alu can luz	Cano Pardiñas Luz Del Carmen
15	mes alu car fab	Cardenas Espinosa Fabiola Vhyleika



Especificación. Lista

Lista enlazada → Estructura de datos dinámica que se construye mediante **nodos enlazados** entre sí.

Nodo → Estructura con dos campos: *dato* y **enlace**

Una lista enlazada es una colección de nodos que se ordena según su posición, tal que cada uno de ellos es accedido mediante el campo de enlace del nodo anterior.

Los elementos que la componen **no ocupan necesariamente posiciones secuenciales o contiguas en la memoria.**

Aunque pueden aparecer dispersos en la memoria, se mantiene un orden lógico interno dado por los enlaces entre ellos.

Especificación. Lista Doble

En algunas aplicaciones podemos querer recorrer la lista hacia adelante y hacia atrás, o dado un elemento, podemos querer conocer rápidamente los elementos anterior y siguiente.

En tales situaciones podemos tener en cada nodo sobre una lista un puntero a las celdas siguiente y anterior.

Especificación. Lista Doble

Características:

Lista doblemente enlazada → Estructura de datos dinámica que se construye mediante **nodos enlazados** entre sí.

Nodo → Estructura con los campos: *dato*, **enlace siguiente y enlace anterior**.

Una lista doblemente enlazada es una colección de nodos que se ordena según su posición, tal que cada uno de ellos es accedido mediante el campo de enlace del nodo anterior o siguiente.

La lista doble es homogénea, dinámica, lineal y tiene acceso secuencial en ambos sentidos.

**Especificamos la Lista
Doble**

Especificación. “Molde General”

TAD Nombre TAD

Igualdad Observacional

Usa

Parámetro Formal

Géneros

observadores básicos

Generadores

otras operaciones

Axiomas

Exporta

Especificación. Lista Doble

TAD ListaDoble <a>

Igualdad Observacional

Si **a** y **b** son dos listas dobles

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, Secuencia<a>, None

Parámetro Formal

a

Géneros

ListaDoble<a>

Especificación. Lista Doble

observadores básicos

`tamaño(ListaDoble<a>) → Natural`

`es_vacia(ListaDoble<a>) → Bool`

`primero(ListaDoble<a>) → a`

`{Pre: la lista doble tiene al menos un elemento}`

`último(ListaDoble<a>) → a`

`{Pre: la lista doble tiene al menos un elemento}`

Especificación. Lista Doble

Generadores

`vacía() → ListaDoble<a>`

`{Post: La lista doble retornada esta vacía}`

`a_partir_de(Secuencia<a>)→ ListaDoble<a>`

`{Post: La lista doble contiene los elementos de la secuencia recibida con el mismo orden que tienen en la secuencia}`

Especificación. Lista Doble

Otras Operaciones

`agregar_frente(ListaDoble<a>, a) → None`

`{Post: la Lista Doble tiene al menos un elemento}`

`agregar_final(ListaDoble<a>, a) → None`

`{Post: la Lista Doble tiene al menos un elemento}`

Especificación. Lista Doble

Otras Operaciones

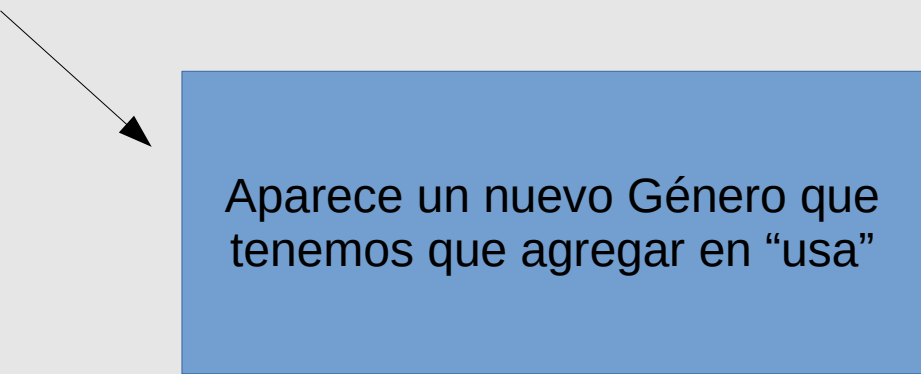
`insertar(ListaDoble<a>, coordenada<ListaDoble<a>>, a)→None`

`{Pre: la coordenada es válida}`

`{Pos: la Lista Doble no esta vacía}`

`borrar(ListaDoble<a>, coordenada<listaDoble<a>>)→
Coordenada<listaDoble<a>>`

`{Pre: la coordenada es válida}`



Aparece un nuevo Género que
tenemos que agregar en "usa"

Especificación. Lista Doble

TAD ListaDoble <a>

Igualdad Observacional

Si **a** y **b** son dos listas dobles

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, Secuencia<a>, None, **coordenada<ListaDoble<a>>**

Parámetro Formal a

Géneros ListaDoble<a>

Especificación. Lista Doble

Otras Operaciones

inicio(ListaDoble<a>) → coordenada<ListaDoble<a>>

fin(ListaDoble<a>) → coordenada<ListaDoble<a>>

El ¿por qué?
Lo vemos la clase que viene...

Especificación. Lista Doble

Axiomas

`vacía()`: Crea una lista doble (sin elementos)

`a_partir_de(Secuencia<a> s)`: crea una lista doble que contiene los elementos de la secuencia recibida con el mismo orden que tienen en la secuencia

`insertar(ListaDoble<a> l, coordenada<ListaDoble<a>> c, a elem)`: agrega elem en l en la posición que hace referencia la coordenada c

`borrar(ListaDoble<a> l, coordenada<ListaDoble<a>> c)`: borra de la lista l el elemento que se encuentra en la posición que hace referencia c. Retorna la coordenada siguiente a c.

`tamaño(listaDoble<a> l)`: Retorna/devuelve la cantidad de elementos de la lista doble l

Especificación. Lista Doble

`es_vacia(ListaDoble<a> l):` Retorna/devuelve verdadero si la lista doble `l` esta vacía y falso en caso contrario

`primero(ListaDoble<a> l):` retorna/devuelve el primer elemento de la cola `l`

`ultimo(ListaDoble<a> l):` retorna/devuelve el último elemento de la lista doble `l`.

`agregar_frente(ListaDoble<a> l, a elem):` agrega `elem` al frente de `l`

`agregar_final(ListaDoble<a> l, a elem):` agrega `elem` al final de `l`

Especificación. Lista Doble

`inicio(ListaDoble<a> l):` retorna una coordenada que hace referencia al primer elemento de `l`

`fin(ListaDoble<a> l):` Retorna una coordenada que hace referencia al siguiente del último elemento de `l`

Exporta

`ListaDoble<a>`, `vacía`, `a_partir_de`, `insertar`, `borrar`, `tamaño`, `es_vacía`, `primero`, `último`, `agregar_frente`, `agregar_final`, `inicio`, `fin`

En la siguiente clase:

- Implementamos una lista doble.
- Vemos Posiciones o Coordenadas e iteradores.