Universidad Nacional de San Martín

Licenciatura en Ciencia de Datos

Algoritmos II

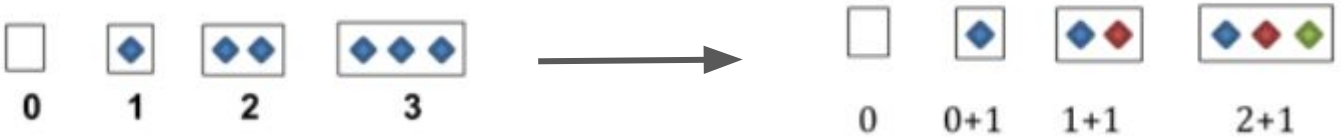
TAD recursivos

Así como vimos que hay funciones/métodos recursivos, también vimos que existen **estructuras de datos recursivas**

TAD Nat

Debemos construir un **tipo de dato** que represente los **números naturales incluyendo el 0**.

-Implica implementar un **tipo de dato infinito**, ya que no hay máximo.



-Si pensamos esta **estructura de forma recursiva** una situación especial es la **inexistencia de elementos**, el **cero (caso base)**. Los **números mayores** respetan cierto **patrón**; El número 1 se compone de un elemento más que el número 0, el número 2 se construye con un elemento más que el número 1, **su predecesor**, y así sucesivamente.

-**El próximo número natural será igual al número actual con un elemento adicional**

TAD Nat (cont)

-Si a la **operación +1** la definimos como **Sucesor**, podemos representar de forma recursiva todos los números naturales así:

Número	Representación de la estructura
0	Cero
1	Sucesor(Cero)
2	Sucesor(Sucesor(Cero))
...	...
n	Sucesor(...(Sucesor(Cero)) -> con n niveles recursivos de Sucesor

Estructura interna

-La **primera abstracción** que hay que hacer es **identificar el VALOR MÍNIMO de un número natural**, mientras que la **segunda** permite **CONSTRUIR el RESTO** de los números **RECURSIVAMENTE**

-Como en Python **NO PODEMOS OPERAR DIRECTAMENTE CON PUNTEROS** lo implementamos con **CLASS**

```
from typing import Union, TypeAlias
Nat: TypeAlias = Union["Cero", "Suc"]  # Define un alias de tipo 'Nat' que puede ser de la la clase 'Cero' o 'Suc'
class Cero:
    def __repr__(self):  # Define el método especial __repr__ para representar la clase como una cadena
        return 'Cero'  # Retorna la cadena 'Cero' cuando se imprime una instancia de 'Cero'
```

-La **clase Cero no necesita** una **estructura interna** (atributos) ya que estamos modelando el **ELEMENTO ÚNICO** que representa el cero

Estructura interna (cont)

```
class Suc:
    def __init__(self, pred: Nat):
        self.pred = pred
    def __repr__(self):
        if isinstance(self.pred, Cero):  # Si 'pred' es una instancia de 'Cero'
            return 'Suc(Cero)'  # Retorna la cadena 'Suc(Cero)' (el "1")
        else:  # Si 'pred' no es una instancia de 'Cero' (es decir, es una instancia de 'Suc')
            return f'Suc({self.pred.__repr__()})'  # Retorna la cadena 'Suc(pred)', donde 'pred' es la representación de la instancia de 'Suc' anterior
```

La clase **Suc** representa la **idea del Sucesor** y permite **construir representaciones del resto** de los números naturales

- Su **estructura** requiere **sólo saber cuál es el predecesor** que también será de tipo **Nat**.
- La estructura tiene **recursión indirecta simple** de la estructura. Es **indirecta** porque **Nat** puede ser de tipo **Suc** y **Suc** se **compone** de un **Nat**: una **recursión mutua**.

Estructura interna (cont) y operaciones

```
    def __str__(self):  # Define el método especial __str__ para convertir la instancia a una cadena
        return str(nat_to_int(self))  # Retorna la representación de la instancia convertida a un número entero mediante la función 'nat_to_int' (la vamos a ver en otra diapo!)
```

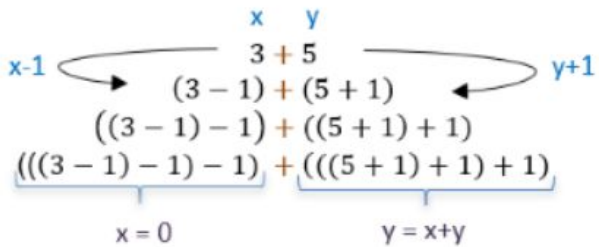
operaciones

```
 def cero() -> Nat:
    return Cero()  # Retorna una nueva instancia de 'Cero'
 def es_cero(n: Nat) -> bool:
    return isinstance(n, Cero)
 def suc(n: Nat) -> Nat:  # Define una función que retorna una instancia de
'Suc' con 'n' como su predecesor
    return Suc(n)  # Retorna una nueva instancia de 'Suc' con 'n' como
predecesor
def pred(n: Nat) -> Nat:
    if es_cero(n):
        raise ValueError('cero no tiene predecesor')
    else:
        return n.pred
```

Operaciones (cont)

```
def nat_to_int(n: Nat) -> int:
    if es_cero(n):  # Si 'n' es una instancia de 'Cero'
        return 0  # Retorna 0, ya que 'Cero' representa el número 0
    else:  # Si 'n' no es una instancia de 'Cero' (es decir, es una instancia de 'Suc')
        return 1 + nat_to_int(pred(n))  # Retorna 1 más el resultado de 'nat_to_int' aplicado al predecesor de 'n'

def suma(x: Nat, y: Nat) -> Nat:
    if es_cero(x):  # Si 'x' es una instancia de 'Cero'
        return y  # Retorna 'y', ya que sumar 0 a 'y' resulta en 'y'
    else:  # Si 'x' no es una instancia de 'Cero' (es decir, es una instancia de 'Suc')
        return suma(pred(x), suc(y))  # Llama recursivamente a 'suma' con el predecesor de 'x' y el sucesor de 'y'
```



Operaciones básicas: para trabajar ver repo!

https://github.com/mapreu/algoritmos2/blob/main/02_recursion/tads/nat.py.

La línea de código significa:

```
__all__ = ['Nat', 'cero', 'division', 'es_cero', 'igual', 'mayor', 'mayor_igual', 'menor', 'menor_igual', 'nat_to_int', 'potencia', 'pred', 'producto', 'resta', 'suc', 'suma']
```

__all__ en u**n módulo de Python** es una lista de nombres de objetos que se exportan cuando se usa la instrucción **from module import ***. Es una forma de **controlar qué se puede importar desde un módulo**

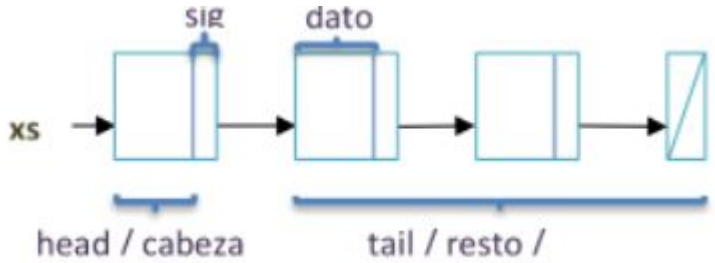
Ejercicio: desarrollar las operaciones restantes de Nat

Definir:

1) igual
2) menor
3) menor o igual
4) mayor
5) mayor o igual
6) resta
7) producto
8) división
9) potencia

TAD Lista dinámica

-**Secuencia de nodos** que se **conectan en una única dirección** y tiene la **ventaja** sobre los arreglos estáticos que puede **incrementarse de forma indefinida** (hasta lo que permita la memoria)



-Tenemos **nodos** que se componen de un **dato de cierto tipo** y un **puntero** al **siguiente nodo**. Al **dato del primer nodo** de la lista lo consumiremos con la operación **head(),** mientras que la **sublista restante** la obtendremos con la operación **tail()**.

Estructura interna: abrir repo!

-Tenemos **dos abstracciones a modelar**: la **lista vacía** y **agregar un elemento**. Solución: **caso especial** es la **lista vacía** y **otro** caso es la **construcción de una lista a partir de otra** incorporando un **nodo nuevo al inicio.**

```
from typing import Generic, TypeVar, Optional, TypeAlias

T = TypeVar('T')
ListaGenerica: TypeAlias = "Lista[T]"
class Nodo(Generic[T]):
    def __init__(self, dato: T, sig: Optional[ListaGenerica] = None):
        self.dato = dato
        if sig is None:  # Si el siguiente nodo es None inicializa 'sig' con una nueva Lista vacía
            self.sig = Lista()
        else:
            self.sig = sig
class Lista(Generic[T]):
    def __init__(self):
        self._head: Optional[Nodo[T]] = None  # Inicializa la cabeza de la lista como None (lista vacía)
```

-**TAD Lista** utiliza **recursión mutua,** (**Lista** se puede componer de un **Nodo** y **Nodo** se compone con otra **Lista**). El **constructor de Nodo** acepta dos argumentos: el **dato a almacenar** y **opcionalmente** el **objeto de la lista que le sigue**. Si **no** se recibe **sig** o,**se construye un nodo** que **apunta** su siguiente a una **lista vacía**

Operaciones básicas

Constructoras

-Se apoya en **dos abstracciones**: **construir una lista vacía** y construir una **lista a partir de otra (__init__())** con un **elemento agregado al inicio**: **operación modificadora** para asemejarla a cómo se hace con las **listas nativas**.

```
    def insertar(self, dato: T):
        actual = copy(self)  # Crea una copia superficial de la lista actual
        self._head = Nodo(dato, actual)  # Inserta el nuevo nodo en la cabeza
```

-insertar() recibe un **dato** para ser almacenado en un **nuevo nodo a la cabeza** de la lista. **Antes** de generar el nuevo nodo, **debemos copiar superficialmente la lista actual** lo cual nos **genera** un **nuevo objeto de tipo Lista pero mantiene las mismas referencias** de objetos que lo componen; el **actual._head sigue apuntando** al **mismo objeto que self._head** (necesario para **evitar recursión infinita)**

Proyectoras

```
    def es_vacia(self) -> bool:
        return self._head is None  # Verifica si la lista está vacía
    def head(self) -> T:
        if self.es_vacia():
            raise IndexError('lista vacia')  # Lanza un error si la lista está vacía
        else:
            return self._head.dato  # Retorna el dato de la cabeza de la lista
```

Proyectoras (cont)

```
    def tail(self) -> ListaGenerica:
        if self.es_vacia():
            raise IndexError('lista vacia')  # Lanza un error si la lista está vacía
        else:
            return self._head.sig.copy()  # Retorna una copia de la lista sin el primer elemento
```

-Problema:Copiamos superficialmente la **sublista** para **evitar** que **modifiquen** el **original** por error y **se destruya la estructura** si decidimos **eliminar el último nodo de ys**, **también** estaremos eliminando el último nodo de xs **sin saberlo**.
-**Posible SOLUCIÓN: replicar la estructura completa reemplazando la copia superficial por profunda por mediante la implementación de un método propio copy()**.

Operaciones (cont)

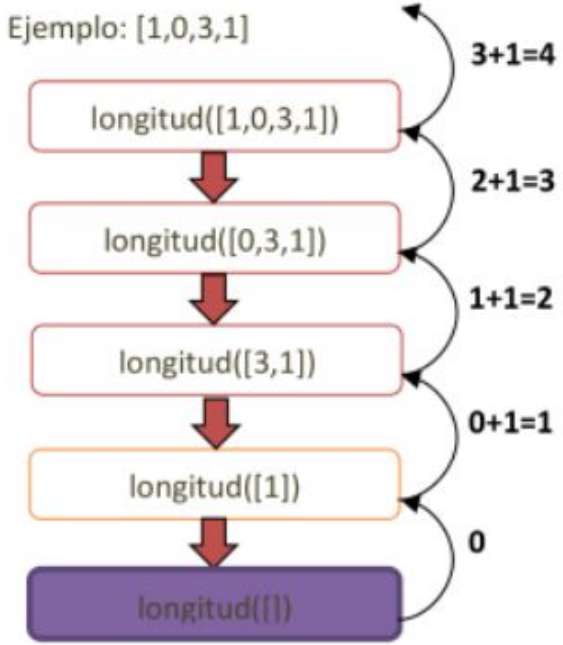
```
    def copy(self) -> ListaGenerica:
        if self.es_vacia():
            return Lista()  # Retorna una lista vacía si la original está vacía
        else:
            parcial = self._head.sig.copy()  # Copia  deep recursivamente la lista siguiente
            actual = Lista()  # Crea una nueva lista
            actual._head = Nodo(copy(self._head.dato), parcial)  # Copia el nodo actual y lo
asigna
            return actual
```

- **Caso base**: generamos una **nueva lista vacía**

-**Caso recursivo:** primero **copiamos también en profundidad la cola de la lista actual**; dará una **nueva lista independiente.**Sólo debemos **agregarle** en la **cabeza** un **nodo nuevo** con el **dato actual**

Operaciones (cont)

```
def __len__(self):
    if self.es_vacia():
        return 0
    else:
        return 1 + self.tail().__len__()
```



- Podríamos haber definir un método **longitud()**, pero sobreescribi**r __len__() permite utilizar len()**.

- **Caso base:** longitud es 0

-**Caso recursivo: primero** computamos la **longitud de la cola de la lista actual** y finalmente la d**evolvemos incrementada en 1 para contabilizar el primer nodo.**

Ejercicio:

completar las funciones del TAD del repo para que corra main del template del repo