Licenciatura en Ciencia de Datos

Algoritmos II

Definiciones

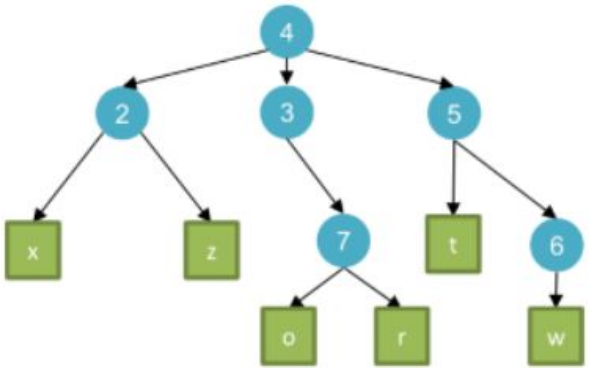
-**No** existe el concepto de **árbol vacío**.

-Elemento mínimo: **hoja**.

-**Se diferencian los nodos que son hojas**: el contenido de nodos intermedios (incluida la raíz) puede ser de un **tipo de dato distinto** al de las hojas.

Tipo de dato de las **hojas**: **T**

Tipo de dato de **nodos no hojas**: **S**



Consideraciones de nuestro TAD

-El **constructor** debería generar una **hoja**, recibiendo un dato de tipo **T**.

- Método constructor **crear_nodo_y_hojas()** para **generar árboles** con al menos **dos nodos**.

-**Insertar_subárbol()** debería **validar** que **no** se aplique sobre una **hoja**.

Implementación

- En **Python NO** tenemos forma de hacer una **IMPLEMENTACIÓN SEGURA** porque la **estrategia de anotaciones de tipos NO ES SUFICIENTE**

```
from typing import Generic, TypeVar

T = TypeVar('T')  # Tipo genérico para las hojas del árbol
S = TypeVar('S')  # Tipo genérico para los nodos del árbol

class ArbolH(Generic[T, S]):
    def __init__(self, dato: T | S):
        # Inicializa el árbol con un dato que puede ser de tipo T o S.
        self._dato: T | S = dato  # El dato del nodo o de la hoja.
        self._subarboles: list[ArbolH[T, S]] = []  # Lista para almacenar los subárboles.
        self._tipo_hoja = type(dato)  # Almacena el tipo del dato inicial como tipo de hoja.
        self._tipo_nodo = None  # Inicialmente no hay un tipo de nodo definido.
```

-**S** no debería aceptarse en el constructor porque **al crear un solo nodo** es **hoja,** pero es "**chanchada**" por el lenguaje

-La estructura es similar al **árbol n-ario**, pero agregamos **dos atributos para guardar los tipos de datos**, porque es útil para **validarlos**.

Creación de nodo intermedio y una o más hojas: con método estático

```
    @staticmethod
    def crear_nodo_y_hojas(dato_raiz: S, *datos_hojas: T) -> "ArbolH[T, S]": #operador * para permitir que  reciba un número variable de argumentos posicionales, separados por ,
        # Crea un nodo raíz con hojas. Verifica que haya al menos un dato para las hojas.
        if not datos_hojas:
            raise ValueError("Se requiere al menos un dato para las hojas")
        # Verifica que todos los datos de las hojas sean del mismo tipo.
        if (not all([isinstance(dato, type(datos_hojas[0])) for dato in datos_hojas])):
            raise ValueError("Todos los datos de las hojas deben ser del mismo tipo")
        nuevo = ArbolH(dato_raiz) # Crea un nuevo árbol con el dato raíz.
        for dato in datos_hojas:
            # Crea un subárbol para cada hoja y lo añade al árbol raíz.
            subarbol = ArbolH(dato)
            subarbol._tipo_nodo = type(dato_raiz)  # Define el tipo del nodo raíz.
            nuevo._subarboles.append(subarbol)  # Agrega el subárbol.
        nuevo._tipo_nodo = type(dato_raiz)  # Almacena el tipo del nodo raíz.
        nuevo._tipo_hoja = type(datos_hojas[0])  # Define el tipo de las hojas.
        return nuevo
```

Funciones importantes

```
    def es_hoja(self) -> bool:
        return self._subarboles == []

    def dato_hoja(self) -> T:
        if self.es_hoja():
            return self._dato
        raise ValueError("El nodo actual no es una hoja")

    def dato_nodo(self) -> S:
        if not self.es_hoja():
            return self._dato
        raise ValueError("El nodo actual es una hoja")

    @property
    def subarboles(self) -> "list[ArbolH[T,S]]":
        return self._subarboles
```

Funciones importantes

```
    def _son_mismos_tipos(self, otro: "ArbolH[T,S]") -> bool:
            return (
            isinstance(otro, ArbolH)
         and ( self._tipo_nodo == otro._tipo_nodo
                  or  # Verifica si los tipos de nodos son iguales.
                  self.es_hoja()
                  or  # Si el árbol actual u otro es una hoja, no compara tipos de
            nodo
                 otro.es_hoja())
         ) and self._tipo_hoja == otro._tipo_hoja  # Verifica que los tipos de hoja
sean iguales.
        )
```

Funciones importantes

```
def _insertar_subarbol_nocheck(self, subarbol: "ArbolH[T,S]") -> None:
    # Esta función inserta un subárbol en el árbol actual sin realizar verificaciones previas.
    # Asignamos el tipo de nodo del árbol actual al tipo de nodo del subárbol
    subarbol._tipo_nodo = self._tipo_nodo
    # Agregamos el subárbol a la lista de subárboles del árbol actual
    self._subarboles.append(subarbol)
```

¿Por qué hacemos este código?: veamos la siguiente diapo!

Funciones importantes

```
def insertar_subarbol(self, subarbol: "ArbolH[T,S]")-> None:
    if self.es_hoja(): # Si el árbol actual es una hoja, no se pueden insertar subárboles
        raise ValueError("No se pueden insertar subárboles en un nodo hoja")
    # Verificamos si el tipo de datos del subárbol es consistente con el tipo de datos del árbol actual
    if not self._son_mismos_tipos(subarbol): # Si los tipos de datos no son consistentes, no podemos insertar el subárbol
        raise ValueError("El árbol a insertar no es consistente con los tipos de datos del árbol actual")
    # Insertamos el subárbol en el árbol actual sin realizar ninguna verificación previa porque ya las hicimos
     self._insertar_subarbol_nocheck(subarbol)
```

Funciones importantes

```
    def __str__(self) -> str:
        # Retorna una representación en string del árbol.
        def mostrar(t: ArbolH[T,S], nivel: int):
            tab = '.' * 4  # Identación para mostrar el nivel del nodo.
            indent = tab * nivel  # Calcula la indentación de acuerdo al nivel.
            if t.es_hoja():
                dato = f'[{t.dato_hoja()}]' # Si es hoja, muestra el dato entre corchetes.
            else:
                dato = str(t.dato_nodo())  # Si es nodo, muestra el dato normalmente.
            out = f'{indent} {dato} \n'  # Formatea la salida.
            for subarbol in t.subarboles:
                out += mostrar(subarbol, nivel + 1)  # Recorre recursivamente los
subárboles.
            return out

        return mostrar(self, 0)  # Empieza desde el nivel 0.
```

Ejercicio

Desarrollar la función con recursividad del TAD:

def es_valido(self) -> bool: