Licenciatura en Ciencia de Datos

Algoritmos II

¿Por qué vimos árboles con hojas?

Para poder modelar (hacer un TAD) **Árboles de Expresión Aritmética.**

-Caso particular de **Árbol Binario con Hojas**

- Es una estructura de datos en forma de árbol que se utiliza para **representar expresiones aritméticas de manera jerárquica.**

-Una expresión aritmética se compone de **términos** y **operadores** (importantes para **cómputo**)

-Cada **nodo** representa un **operador o un operando** de la expresión

-Los **nodos hoja** representan **operandos** (expresiones base, términos mínimos)

- Los **nodos internos** representan **operadores**, como suma (+), resta (-), multiplicación (*), división (/).
- Los **hijos** de un nodo interno son **operandos u otros operadores.**

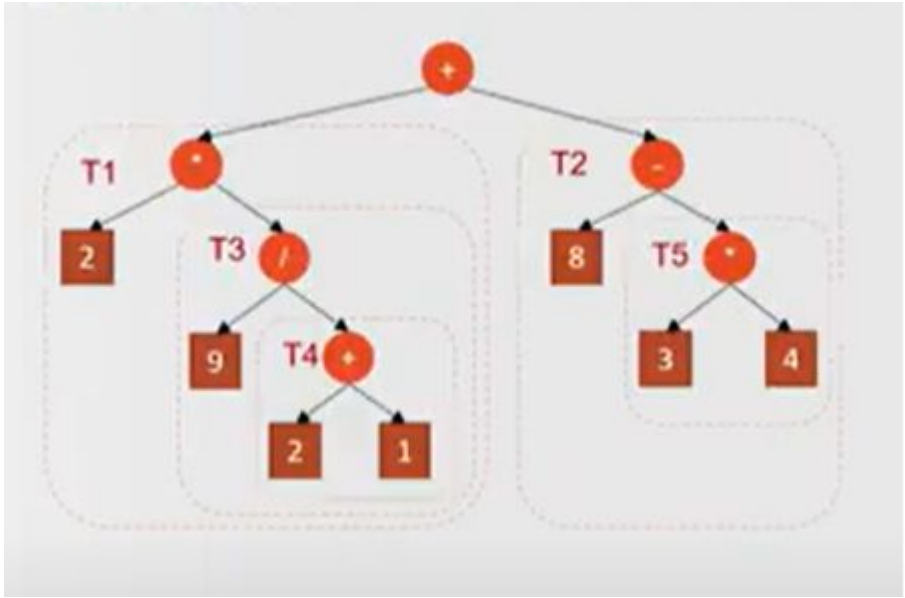
-La idea es evaluarlas para dar un **resultado**

Ejemplo de evaluación de la expresión

EXPRESIÓN = 2 ∗ 9 / (2 + 1) + 8 − 3 ∗ 4

DESCOMPOSICIÓN EN TÉRMINOS HASTA LLEGAR AL RESULTADO:

T1 + T2	6+8-T5
2* T3 + T2	6+8-(3*4)
2* (9/T4) + T2	6+8-12
2*(9/(2+1)) +T2	6 + (-4)
2*(9/3)+T2	**2**
2*3 +T2	
6+T2	



Implementación: abrir repo!

```
from abc import ABC, abstractmethod  # Importa la clase base abstracta y el decorador para métodos abstractos
from typing import TypeAlias  # Importa TypeAlias para definir alias de tipos
from arbol_hojas import ArbolH  # Importa la clase ArbolH que se usa como base para el árbol de expresión aritmética

# Define un alias de tipo Number, que puede ser int o float
Number: TypeAlias = int | float
```

Implementación

```
# Define una clase abstracta Operador, base para los operadores matemáticos
class Operador(ABC):
    simbolo: str  # Atributo que representa el símbolo del operador (por ejemplo, '+', '-', '*', '/')

    @staticmethod
    @abstractmethod  # Método abstracto que obliga a las subclases a implementar su propia versión de 'operar'
    def operar(a: Number, b: Number) -> Number:  # Toma dos números y retorna un número (resultado de la operación)
        ...

    # Método que convierte la instancia de Operador en su representación en cadena, devolviendo el símbolo
    def __str__(self) -> str:
        return self.simbolo  # Retorna el símbolo del operador como cadena
```

Implementación

```
# Clase Suma que hereda de Operador, representa la operación de suma
class Suma(Operador):
    simbolo: str = '+'  # Define el símbolo '+' como símbolo de la operación

    @staticmethod
    def operar(a: Number, b: Number) -> Number:  # Implementa la operación suma
        return a + b  # Retorna la suma de 'a' y 'b'

# Clase Producto que hereda de Operador, representa la operación de multiplicación
class Producto(Operador):
    simbolo: str = '*'  # Define el símbolo '*' para la multiplicación

    @staticmethod
    def operar(a: Number, b: Number) -> Number:  # Implementa la operación de
multiplicación
        return a * b  # Retorna el producto de 'a' y 'b'
```

Implementación

```
# Clase Resta que hereda de Operador, representa la operación de resta
class Resta(Operador):
    simbolo: str = '-'  # Define el símbolo '-' para la resta

    @staticmethod
    def operar(a: Number, b: Number) -> Number:  # Implementa la operación de resta
        return a - b  # Retorna la resta de 'a' menos 'b'

# Clase División que hereda de Operador, representa la operación de división
class Division(Operador):
    simbolo: str = '/'  # Define el símbolo '/' para la división

    @staticmethod
    def operar(a: Number, b: Number) -> Number:  # Implementa la operación de división
        return a / b  # Retorna el cociente de 'a' entre 'b'
```

Implementación

```
#Define una nueva clase llamada ExpresionAritmetica que hereda de la clase genérica ArbolH.
    # ArbolH es una estructura de árbol genérica que maneja dos tipos de datos:
    # - Number (int o float) para los nodos hoja del árbol, que almacenan valores numéricos.
    # - Operador, una clase abstracta que representa un operador matemático, para los nodos internos.
    # Esto permite que la ExpresionAritmetica maneje expresiones matemáticas en forma de árbol, donde
    # las hojas son números y los nodos internos son operadores aritméticos.

class ExpresionAritmetica(ArbolH[Number, Operador]):
    def __init__(self, dato: Number):
        # Llama al constructor de la clase base ArbolH con el dato proporcionado
        super().__init__(dato)

    @staticmethod
    def valor(valor: Number) -> "ExpresionAritmetica":
        # Método estático para crear una nueva instancia de ExpresionAritmetica con un valor dado
        return ExpresionAritmetica(valor)
```

Una expresión aritmética **puede ser int o float**.

Se representa como una **hoja del árbol.**

Implementación

```
    @staticmethod
    def valor(valor: Number) -> "ExpresionAritmetica":  # Método para crear una instancia de ExpresionAritmetica con un valor numérico
        return ExpresionAritmetica(valor)  # Retorna una nueva instancia de ExpresionAritmetica con el valor proporcionado

    # Método privado para crear una operación aritmética con dos operandos y un operador
    @staticmethod
    def _crear_operacion(operador: Operador, operando_1: "ExpresionAritmetica", operando_2: "ExpresionAritmetica") -> "ExpresionAritmetica":
        nuevo = ExpresionAritmetica(operador)  # Crea un nuevo nodo de ExpresionAritmetica con el operador
        nuevo._insertar_subarbol_nocheck(operando_1)  # Inserta el primer operando en el subárbol izquierdo
        nuevo._insertar_subarbol_nocheck(operando_2)  # Inserta el segundo operando en el subárbol derecho
        return nuevo
```

Implementación: Métodos para crear expresiones aritméticas

```
    @staticmethod
    def suma(operando_1: "ExpresionAritmetica", operando_2: "ExpresionAritmetica") -> "ExpresionAritmetica":
        return ExpresionAritmetica._crear_operacion(Suma(), operando_1, operando_2)

    @staticmethod
    def resta(operando_1: "ExpresionAritmetica", operando_2: "ExpresionAritmetica") -> "ExpresionAritmetica":
        return ExpresionAritmetica._crear_operacion(Resta(), operando_1, operando_2)

    @staticmethod
    def producto(operando_1: "ExpresionAritmetica", operando_2: "ExpresionAritmetica") ->
"ExpresionAritmetica":
        return ExpresionAritmetica._crear_operacion(Producto(), operando_1, operando_2)  # Crea y retorna una
expresión de multiplicación

    @staticmethod
    def division(operando_1: "ExpresionAritmetica", operando_2: "ExpresionAritmetica") ->
"ExpresionAritmetica":
        return ExpresionAritmetica._crear_operacion(Division(), operando_1, operando_2)
```

Implementación

```
# Método para verificar si la expresión es un valor (si es una hoja en el árbol)
    def es_valor(self) -> bool:
        return self.es_hoja()

    # Método que evalúa recursivamente la expresión aritmética
    def evaluar(self) -> Number:
        if self.es_valor(): #chequea que sea hoja
           return self.dato_hoja()  #Si el nodo es una hoja, retorna su valor numérico
        operador = self.dato_nodo()  # Si es un nodo interno, obtiene el operador
almacenado
        operando_1, operando_2 = self.subarboles  # Obtiene los subárboles (operando
izquierdo y derecho)
        return operador.operar(operando_1.evaluar(), operando_2.evaluar())  # Evalúa
recursivamente los operandos y aplica el operador
```

Implementación

```
    def __str__(self) -> str:
        return super().__str__()
```