Trabajo Práctico Integrador (Programación I)

Árboles de operaciones matemáticas

Martínez Luciano Joaquín – lucianomartinez27@gmail.com
Rodriguez Santiago Gabriel – santiii.ssg@gmail.com

Materia: Programación I
Profesor: Julieta Trapé
Fecha de Entrega: 09/6/2025

Índice

Introducción

En informática, las estructuras de datos son herramientas que nos permiten organizar y almacenar información de manera eficiente. Una de las estructuras que se destacan son los árboles, esta estructura es fundamental para operaciones como búsqueda, toma de decisiones, pero resulta particularmente útil para representar estructuras jerárquicas.

En este trabajo abordaremos la aplicación de **árboles binarios de operaciones matemáticas**. Se explorará cómo se pueden representar las distintas expresiones algebráicas a través de distintos tipos de nodos, con ejemplos prácticos, que faciliten la comprensión y visualización de las estructuras.

En este contexto, nos proponemos demostrar mediante un caso práctico en Python, la implementación de un árbol binario de operaciones matemáticas, incluyendo la construcción de expresiones complejas, su evaluación y visualización.

A través de este estudio se busca consolidar el conocimiento sobre esta estructura de datos e ilustrar su aplicación práctica. Haciendo hincapié en cómo estas estructuras nos permiten modelar y resolver problemas en programación.

Marco Teórico

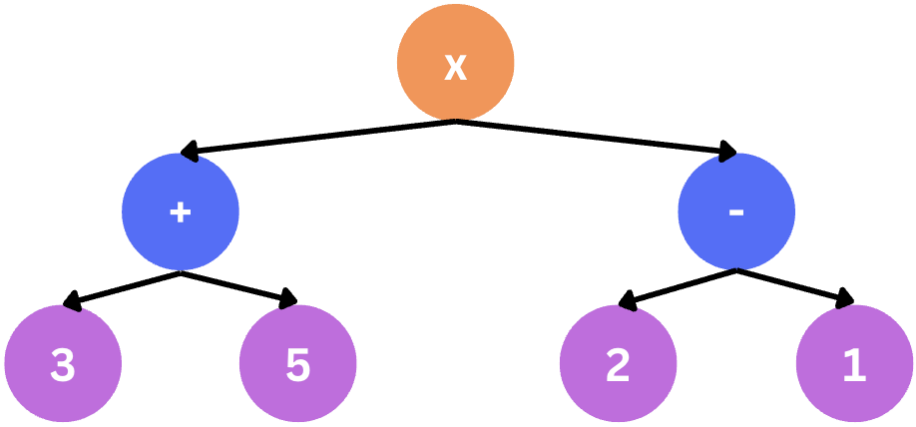
¿Qué es un Árbol?
Un árbol es una estructura de datos jerárquica que está compuesta de nodos, cada uno de los cuales puede apuntar a uno o varios nodos.

Árbol binario:
En un **árbol binario** cada nodo tiene como máximo dos hijos, el hijo izquierdo y el hijo derecho.
Árbol binario de Operaciones Matemáticas:
Particularmente, un **árbol binario de operaciones matemáticas**, representa operaciones algebraicas y aritméticas. En esta estructuras los nodos internos pueden representar **operadores matemáticos** (como suma, resta, multiplicación y división) o valores (el 1, 2, 3, etc).
Elementos:

- **Nodo raíz:** Es el punto de entrada del árbol y representa la operación final o con menos prioridad de la expresión.

- **Nodos internos:** Son los nodos que contienen la expresiones izquierda y derecha.
- **Nodo hoja:** Son los nodos donde finaliza el árbol, es decir no tienen hijos y representan un **valor.**

Representación gráfica:



Caso Práctico

El objetivo es desarrollar un programa que represente operaciones matemáticas básicas a través de **árboles binarios**, utilizando clases en Python. La implementación permite construir expresiones matemáticas anidando operaciones y valores, para luego poder evaluar la expresión y obtener el resultado o mostrarlo de forma gráfica.

Se definió la clase **Expresión**, que representa la base del resto de las expresiones, la misma define los métodos que las subclases deben implementar. Por su parte, **ExpresionDoble**, representa las operaciones, que contienen la **expresión izquierda** y **expresión derecha** y define la lógica de cómo se debe imprimir por pantalla el árbol. Por su parte, **Suma, Resta, Multiplicación, y División** representan las operaciones matemáticas básicas. Por último, **Valor** es el **Nodo hoja**, que simplemente representa un valor numérico.

Evaluación y Visualización:

Cada tipo de nodo define el método **evaluar()**, quien se encarga de retornar el resultado, que en el caso de las **operaciones** es el resultado de evaluar el **nodo**

izquierdo aplicando la operación sobre el **nodo derecho**, mientras que en los **valores**, retorna el número que tenga asignado.
En cuanto a la visualización, se define el método __str__(), qué es un método especial de python para representar una clase como string. En las operaciones imprime el resultado, junto con el operador y el resto de las expresiones. Mientras que los valores imprimen el número que tengan asignado.

```
El resultado de la multiplicacion es 8.0
      (8.0)
       (*)
      /   \
  (8)       (1.0)
  (+)        (-)
 /   \      /   \
 3   5     (2.0)    1
           (/)
           /   \
          10   5
```

```python
class Expresion:
  def evaluar(self):
    raise Exception("Tiene que implementarlo la subclase")

  def calcular_ancho(self):
    raise Exception("Tiene que implementarlo la subclase")

class Valor(Expresion):
  def __init__(self, valor):
    self.valor = valor

  def calcular_ancho(self):
    return 1

  def evaluar(self):
    return self.valor
```

```
  def __str__(self):
    return str(self.valor)

class ExpresionDoble(Expresion):
  def __init__(self, exp1, exp2):
    self.exp_izquierda = exp1
    self.exp_derecha = exp2

  def operador(self):
    raise Exception("Tiene que implementarlo la subclase")

  def calcular_ancho(self):
    return self.exp_izquierda.calcular_ancho() + self.exp_derecha.calcular_ancho()

  def __str__(self):
    expresion_1 = str(self.exp_izquierda)
    expresion_2 = str(self.exp_derecha)
    lineas_expresion_1 = expresion_1.splitlines()
    lineas_expresion_2 = expresion_2.splitlines()
    ancho_del_arbol = self.calcular_ancho()
    resultado = str(self.evaluar())
    ancho_resultado = len(resultado) // 2
    padding = ancho_del_arbol * 2
    linea_resultado = f"({resultado}) ".rjust(padding + ancho_resultado)
    linea_operador = f"({self.operador()}) ".rjust(padding)
    linea_de_ramas = "/   \\".rjust(padding)

    cantidad_de_lineas_de_la_expresion_mas_larga = max(len(lineas_expresion_1),
len(lineas_expresion_2))
    todas_las_lineas = []
    for linea in range(cantidad_de_lineas_de_la_expresion_mas_larga):
      linea_exp_1 = lineas_expresion_1[linea] if linea < len(lineas_expresion_1) else ""
      linea_exp_2 = lineas_expresion_2[linea] if linea < len(lineas_expresion_2) else ""
      padding_centro = 3 if linea_exp_2.isnumeric()  else ancho_del_arbol
      padding_izquierda = 0 if linea_exp_2.isnumeric()  else ancho_del_arbol
      padding_derecha = 0 if linea_exp_1.isnumeric() else ancho_del_arbol
      linea_exp_1 = linea_exp_1.rjust(padding_izquierda)
      linea_exp_2 = linea_exp_2.ljust(padding_derecha)
      todas_las_lineas.append(f"{linea_exp_1}{" " * padding_centro}{linea_exp_2}")
```

```
        return (f"{linea_resultado}\n"+
                f"{linea_operador}\n"+
                f"{linea_de_ramas}\n"+
                '\n'.join(todas_las_lineas))

class Suma(ExpresionDoble):

    def operador(self):
        return "+"

    def evaluar(self):
        # evaluamos ambas expresiones y sumamos
        return self.exp_izquierda.evaluar() + self.exp_derecha.evaluar()

class Resta(ExpresionDoble):

    def operador(self):
        return "-"

    def evaluar(self):
        # evaluamos ambas expresiones y restamos
        return self.exp_izquierda.evaluar() - self.exp_derecha.evaluar()

class Multiplicacion(ExpresionDoble):

    def operador(self):
        return "*"

    def evaluar(self):
        # evaluamos ambas expresiones y multiplocamos
        return self.exp_izquierda.evaluar() * self.exp_derecha.evaluar()

class Division(ExpresionDoble):

    def operador(self):
        return "/"

    def evaluar(self):
        # evaluamos ambas expresiones y dividimos
        return self.exp_izquierda.evaluar() / self.exp_derecha.evaluar()
```

```
multiplicacion = Multiplicacion(
  Suma(
    Valor(3),
    Valor(5)
  ),
  Resta(
    Division(
      Valor(10),
      Valor(5)
    ),
    Valor(1)
  )
)

print(f"El resultado de la multiplicacion es {multiplicacion.evaluar()}")
print(multiplicacion)

suma = Suma(
  Division(
    Valor(-6),
    Valor(-3)
  ),
  Multiplicacion(
    Resta(
      Valor(96),
      Valor(12)
    ),
    Valor(23)
  )
)
print(f"El resultado de la suma es {suma.evaluar()}")
print(suma)
```

Metodología Utilizada

• **Análisis del problema**: Se identificó la necesidad de representar y evaluar expresiones matemáticas de forma estructurada utilizando árboles binarios. Se decidió emplear nodos para representar valores numéricos y operaciones básicas como suma, resta, multiplicación y división.

• **Diseño de clases**: Se utilizó el paradigma de **Programación Orientada a Objetos** para modelar las entidades del sistema.

• **Construcción de árboles binarios**: Se desarrolló una estructura recursiva donde cada operación binaria contiene dos subexpresiones como hijos izquierdo y derecho, formando así un árbol binario que puede evaluarse en profundidad.

• **Implementación de evaluación recursiva**: Cada clase implementó el método **evaluar()**, permitiendo calcular el resultado de la expresión completa recorriendo el árbol desde las hojas hasta la raíz.

• **Visualización del árbol**: Se sobreescribió el método **__str__()** para representar gráficamente el árbol de forma textual, facilitando así la comprensión visual de su estructura.

• **Pruebas y validación**: Se construyeron árboles de prueba con combinaciones de operaciones para validar que la lógica de evaluación y la representación visual sean correctas.

Resultados Obtenidos

Como resultado del desarrollo del trabajo práctico, se logró implementar un sistema funcional capaz de representar y evaluar expresiones aritméticas mediante árboles binarios. Entre los principales logros se destacan:

- **Evaluación correcta de expresiones complejas**: El sistema permite construir árboles que representan operaciones combinadas, respetando el orden jerárquico de las operaciones matemáticas.
- **Visualización estructurada en forma de árbol**: El método **__str__()** implementado en las clases permite mostrar gráficamente la expresión en forma jerárquica, lo cual facilita la interpretación del cálculo y demuestra el recorrido del árbol.
- **Flexibilidad y escalabilidad**: Se pueden crear nuevas expresiones fácilmente combinando objetos, gracias a la reutilización de clases. Además, el diseño modular permite agregar nuevas operaciones (como potencia o módulo) sin alterar la estructura base.

- **Aplicación correcta de principios de POO**: Se aplicaron con éxito los conceptos de herencia, encapsulamiento y polimorfismo, logrando un código limpio, mantenible y reutilizable.
- **Robustez ante combinaciones variadas**: El sistema fue probado con diferentes operaciones (positivos, negativos, divisiones) y los resultados fueron correctos en todos los casos, demostrando la confiabilidad del enfoque implementado.

Conclusiones

El desarrollo de este trabajo permitió comprender y aplicar de manera práctica el uso de **árboles binarios** como estructura de datos útil para representar y evaluar expresiones matemáticas. A través de la implementación de un sistema basado en **Programación Orientada a Objetos (POO)**, se logró construir una solución modular, clara y escalable.

Dificultades que surgieron durante el proyecto y cómo se resolvieron

Si bien la implementación de las operaciones básicas del árbol resultó simple, se presentaron desafíos significativos al intentar generar una representación gráfica del árbol binario en la consola. La dificultad principal radicó en mostrar las expresiones de la izquierda y de la derecha en una misma línea de la consola. Esto se debía a que cada para cada expresión se obtenía su representación como string multilínea, lo que generaba saltos de línea y forzaba la impresión en filas separadas.

Para resolverlo,se implementó una estrategia donde cada representación de string de la expresión se dividía en líneas individuales (método **splitlines()**), para posteriormente iterar e interpolar las líneas. Se manejó además el caso de expresiones de diferente "altura" asegurando que las líneas faltantes se traten como strings vacíos.

Un segundo obstáculo, fue mantener la alineación y **espaciado** en la representación en árboles que estaban desbalanceados, ya que se perdía el órden visual y la relación de los hijos con los padres dentro de la representación.

Si bien no se encontró una solución que abarcara todos los casos, se pudo encontrar una solución satisfactoria en una gran cantidad de escenarios prácticos, para esto se calculó el ***extensión horizontal*** *(método* ***calcular_ancho()****)* de el árbol, y se agregó ese ancho como relleno (**padding**) a cada lado de la representación como string.

Reflexión: En definitiva, el trabajo no solo permitió cumplir con los objetivos funcionales propuestos, sino que también sirvió como una experiencia enriquecedora

para fortalecer habilidades de diseño, análisis y resolución de problemas utilizando Python.

Bibliografía

- https://docs.google.com/document/d/10k16oL15EeyOaq92aoi4qwK3t_22X29-FSV2iV-8N1U/edit?tab=t.0#heading=h.8x34qta6sgy9
- GeeksforGeeks. (2025, 25 Marzo). *Level Order Tree Traversal*. Recuperado de https://www.geeksforgeeks.org/level-order-tree-traversal/

Anexos

• Capturas de pantalla del código funcionando en consola.
• Repositorio en GitHub: https://github.com/Santi-R97/integradorprogramacion
• Video explicativo: https://www.youtube.com/watch?v=hD85q91jeEc