Datos Avanzados: Arboles binarios en Python

Materia: Programación I

Alumnos:
David Lazarte

Profesores:
Cinthia Rigoni
Ana Mutti

INDICE

Introducción

Los árboles, como estructuras de datos avanzadas, cumplen un rol fundamental en la programación ya que permiten manipular y organizar la información de forma eficiente. En este caso particular, los árboles binarios se destacan por su utilidad en diversas aplicaciones, como la búsqueda binaria, el ordenamiento de datos, la representación jerárquica de información y la toma de decisiones.
Gracias a su estructura, en la que cada nodo puede tener hasta dos hijos (uno izquierdo y otro derecho), los árboles binarios permiten recorrer y procesar conjuntos de datos de manera sistemática, reduciendo tiempos de ejecución y mejorando el rendimiento de los algoritmos.
En este pequeño trabajo se abordará el estudio de los árboles binarios desde una perspectiva teórico-práctica, utilizando el lenguaje de programación Python y se empleará una representación simple basada en listas anidadas o arrays, lo cual facilita la comprensión del funcionamiento interno del árbol y sus operaciones básicas.

Marco Teórico

Un árbol binario es una estructura de datos jerárquica donde cada nodo tiene como máximo dos hijos: uno izquierdo y uno derecho.
Es una de las estructuras más usadas en informática por su eficiencia para representar relaciones jerárquicas, realizar búsquedas, ordenar datos, entre otros.

Un árbol binario está compuesto por un nodo raíz que es el principal desde nacen los demás o en otras palabras, se ramifican los demás nodos, los nodos hijos que los descendiente directos de su nodo padre, hojas que también son nodos pero estos no tienen hijos. Como dato adicional, podemos decir que un subárbol es un nodo con sus hijos y por último, la altura de un árbol indica la cantidad de niveles desde el primer nodo (raíz) hasta las hojas (nodos sin hijos) que se encuentran al final.

Si quisiéramos representar un árbol binario simple a partir de listas tendríamos lo siguiente:

arbol_binario = ['NODO_A', ['SUB_NODO_B', [],[]], ['SUB_NODO_C', [],[]]]

A partir de ahora, podemos realizar operaciones como agregar nuevos nodos, ya sea a la izquierda o derecha o recorrer los mismos.

Caso Practico

A continuación, se realizara la implementación de un arbol binario utilizando listas, paso este caso práctico se utiliza Python en su versión 3.13.5.

Comenzamos con una función para crear un arbol:

```python
def crearArbol(nombreNodoRaiz):
    return [nombreNodoRaiz, [], []]
```

Ahora para poder manipular el arbol creado, agregamos dos funciones más para agregar nodos hijos a la izquierda o a la derecha del mismo:

```python
def insertarIzquierda(arbol, nuevoValor):
    subarbolIzq = arbol[1]
    if subarbolIzq:
        arbol[1] = [nuevoValor, subarbolIzq, []]
    else:
        arbol[1] = [nuevoValor, [], []]

def insertarDerecha(arbol, nuevoValor):
    subarbolDer = arbol[2]
    if subarbolDer:
        arbol[2] = [nuevoValor, [], subarbolDer]
    else:
        arbol[2] = [nuevoValor, [], []]
```

Para realizar recorridos de modo PREORDER, INORDER y POSTORDER vamos a definir tres funciones más:

```python
def recPreorden(arbol):
    if arbol:
        print(arbol[0], end=' ')
        recPreorden(arbol[1])
        recPreorden(arbol[2])

def recInorden(arbol):
    if arbol:
        recInorden(arbol[1])
        print(arbol[0], end=' ')
        recInorden(arbol[2])

def recPostorden(arbol):
    if arbol:
        recPostorden(arbol[1])
        recPostorden(arbol[2])
        print(arbol[0], end=' ')
```

Para el recorrido Preorden: desde la raíz, pasando por izquierda y luego derecha.
Inorden: de izquierda a la raiz y luego a la derecha.
Postorden: de izquierda a derecha y raíz.

```python
import data

##Implementacion de las funciones:

#creamos el arbol de prueba (Nodo raiz A)
arbol_prueba = data.crearArbol('A')

#agregamos 2 hijos, nodos B y C
data.insertarIzquierda(arbol_prueba, "B")
data.insertarDerecha(arbol_prueba, "C")

#agregamos 2 hijos pero al nodo B (izquierdo) en este casos los D y E
data.insertarIzquierda(arbol_prueba[1], "D")
data.insertarDerecha(arbol_prueba[1], "E")


##Recorremos:
print("Recorrido Preorden:")
data.recPreorden(arbol_prueba)
#la salida deberia ser: A B D E C

print("\nRecorrido Inorden:")
data.recInorden(arbol_prueba)
#la salida deberia ser: D B E A C

print("\nRecorrido Postorden:")
data.recPostorden(arbol_prueba)
#la salida deberia ser: D E B C A
```

Conclusiones

Trabajar con árboles binarios nos permite organizar datos de forma jerárquica y eficiente, lo que es útil en muchos problemas reales, como por ejemplo: búsquedas rápidas en grandes volúmenes de datos (como en bases de datos o buscadores), organización de decisiones en inteligencia artificial o juegos (árboles de decisión). Procesamiento de expresiones matemáticas y estructuras anidadas (como compiladores).
El uso de listas para representar árboles también nos permite comprender la lógica de forma sencilla, sin necesidad de clases o estructuras complejas, lo cual es ideal para comenzar con este tipo de estructuras.

En resumen, los árboles binarios son herramientas potentes y versátiles que, una vez comprendidas, nos abren la puerta a resolver problemas más complejos de manera ordenada y eficiente.

Bibliografía

- Documentación Tecnicatura universitaria en programación.
- Web oficial Python – Documentación referida a estructuras de datos: https://docs.python.org/es/3.13/tutorial/datastructures.html
- Google Site – Programación 2 https://sites.google.com/site/programacioniiuno/temario/unidad-5---grafos/rboles-binarios