Trabajo de Investigación: Implementación de Árboles en Python Utilizando Listas

---

Alumnos: (grupo 8)

- Manzanelli, Marcos correo: marcos.manzanelli.l@gmail.com
- Massazza, Sergio correo: sermass@yahoo.com

Materia: Programación I

Profesor: Bruselario, Sebastián
Tutora: Cimino, Virginia

Fecha de Entrega: (08/06/2025)

1. Introducción

¿Qué son los árboles en programación?

Los Árboles son estructuras de datos, son muy utilizadas, pero también una de las más complejas. Se caracterizan por almacenar sus nodos en forma jerárquica y no en forma lineal como las Listas enlazadas, Colas, Pilas, etc... Son utilizadas en algoritmos para el guardado de información en nuestra computadora por carpetas, en la estructura de una página web, en base de datos, en algoritmos de búsqueda. En los últimos años este tipo de estructuras han sido utilizadas con mucha frecuencia en la Inteligencia artificial.

Hemos visto que en Python la lista es una estructura mutable (es decir podemos modificar sus elementos, agregar y borrar), y también que podemos definir elementos de una lista que sean de *tipo lista*, en ese caso decimos que tenemos una lista anidada.

Este trabajo explora una forma alternativa de implementar árboles **sin clases ni objetos**, utilizando únicamente **listas anidadas**, aprovechando esta capacidad de las listas de contener otras listas.

2.Objetivos

- Comprender cómo se puede representar un árbol binario utilizando listas en Python.
- Implementar operaciones básicas sobre árboles: inserción y recorridos.
- Evaluar las ventajas y desventajas de este enfoque frente a implementaciones orientadas a objetos.
- Promover el uso de representaciones simples como herramienta educativa.

---

3. Marco Teórico

Un **árbol** es una estructura de datos no lineal compuesta por nodos. El nodo superior se llama **raíz**, y cada nodo puede tener cero o más nodos hijos.

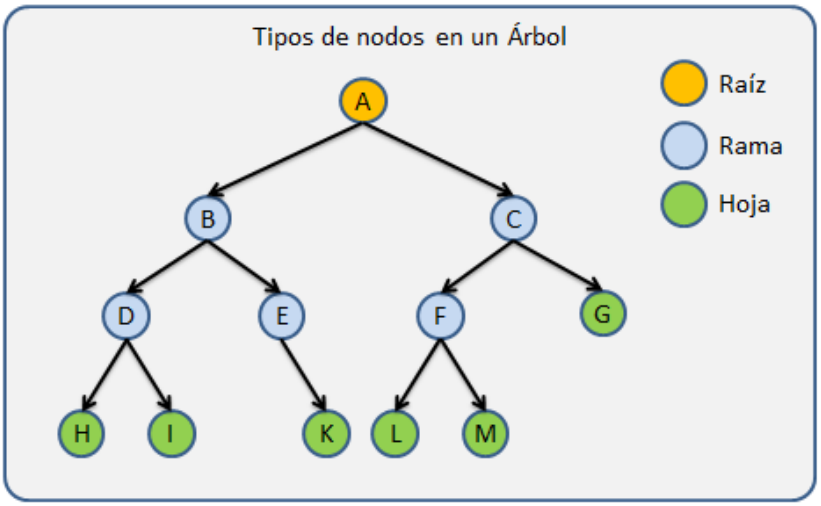

Fig 1 La imagen muestra de forma gráfica cuales son los nodos Raíz, Rama, Hoja.

Propiedades clave de los árboles binarios:

- **Raíz**: nodo inicial.
- **Hojas**: nodos sin hijos.
- **Altura**: número de niveles desde la raíz hasta la hoja más profunda. (Máx número de niveles)
- **Subárboles o ramas**: cualquier nodo junto a sus descendientes forman un subárbol.
- **Orden:** El orden de un árbol es el número máximo de hijos que puede tener un Nodo.

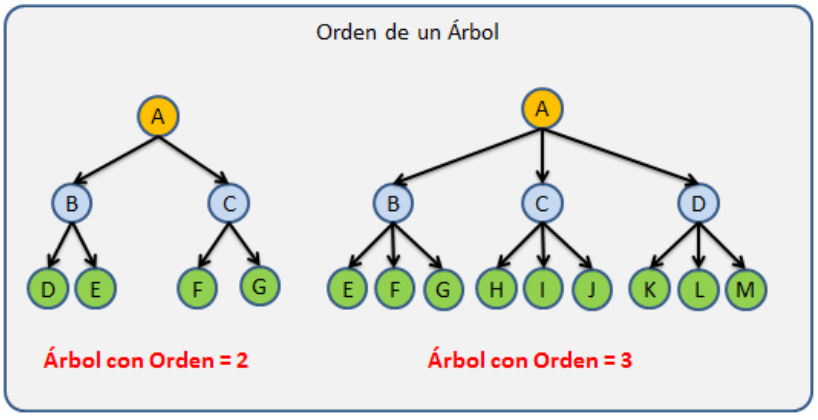

Fig 2 Árboles con Orden = 2(Izquierda) y un segundo con Orden = 3(Derecha).

- **Grado:** El grado se refiere al número mayor de hijos que tiene alguno de los nodos del Árbol y está limitado por el Orden, ya que este indica el número máximo de hijos que puede tener un nodo

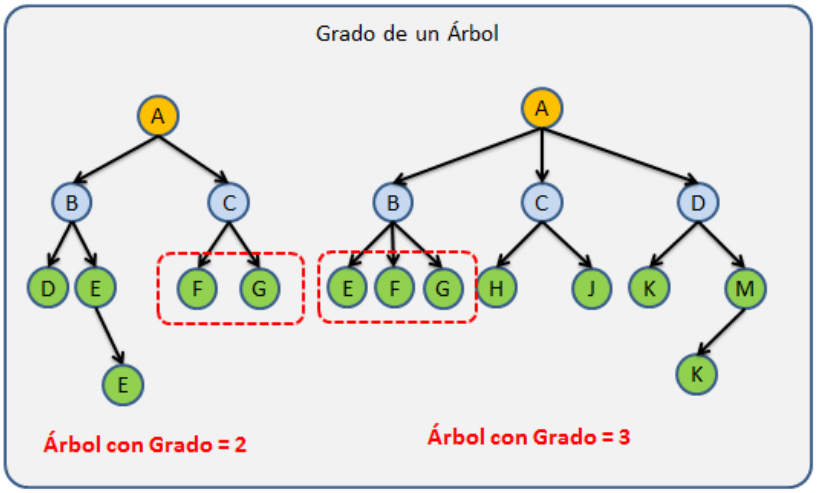

fig 3 podemos apreciar un Árbol con grado 2(Izquierda) y un otro con grado 3(Derecha).

- **Árbol n-ario:** los arboles n-arios son aquellos árboles donde el número máximo de hijos por nodo es de N, en la figura 3 podemos apreciar dos árboles con grado 2 y grado 3, estos dos árboles también los podemos definir como Árbol n-ario con n = 2 y n=3 (binario y ternario respectivamente).

  El **árbol binario**, cada nodo tiene como máximo dos hijos: izquierdo y derecho.

Las implementaciones típicas en lenguajes como Java o C++ utilizan punteros y clases. En Python, una alternativa es usar listas del tipo:

[valor, subárbol izquierdo, subárbol derecho],

donde cada subárbol también es una lista similar.

Recorrido sobre Árboles

Los recorridos son algoritmos que nos permiten recorrer un árbol en un orden específico, los recorridos nos pueden ayudar encontrar un nodo en el árbol, o buscar una posición determinada para insertar o eliminar un nodo.

Podemos catalogar las búsqueda en dos tipos, las búsqueda en profundidad y las búsquedas en amplitud.
Las tres más utilizadas son de profundidad: recorrido pre-orden, post-orden e in-orden

- **Búsqueda en profundidad**

  - Recorrido Pre-orden: El recorrido inicia en la Raíz y luego se recorre cada unos de los sub-árboles de izquierda a derecha.

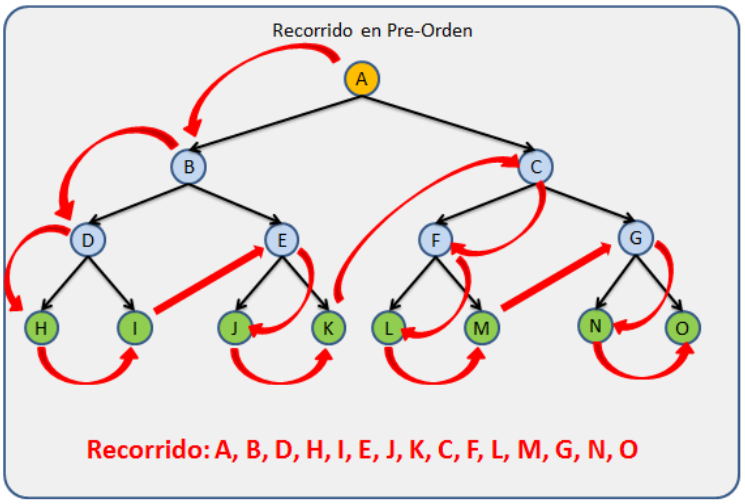

fig 4 podemos ver el orden en que es recorrido el árbol iniciando desde la Raíz.

  - **Recorrido Post-orden:** Se recorren cada uno de los sub-árboles y al final se recorre la raíz.

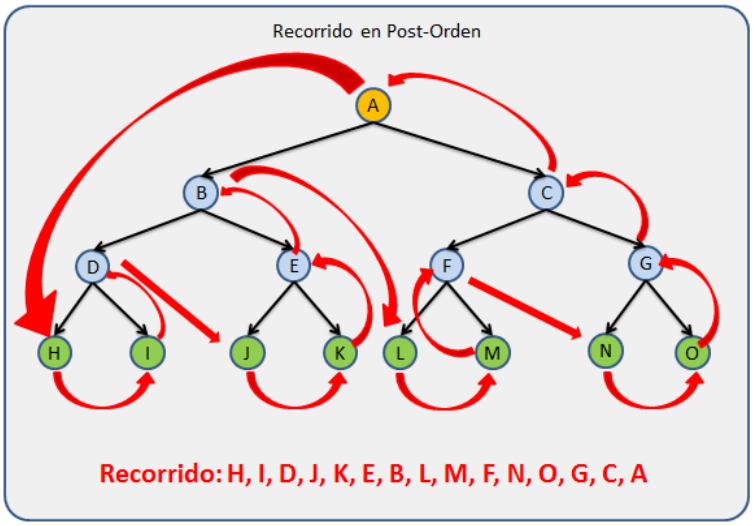

fig 5 podemos observar como se realiza el recorrido en Post -Orden. Es importante notar que el primer nodo que se imprime no es la Raíz pues en este recorrido la Raíz de cada Sub-Árbol es procesado al final, ya que toda su descendencia ha sido procesada.

- **Recorrido in-orden:** Se recorre el primer sub-árbol, luego se recorre la raíz y al final se recorren los demás sub-árboles.

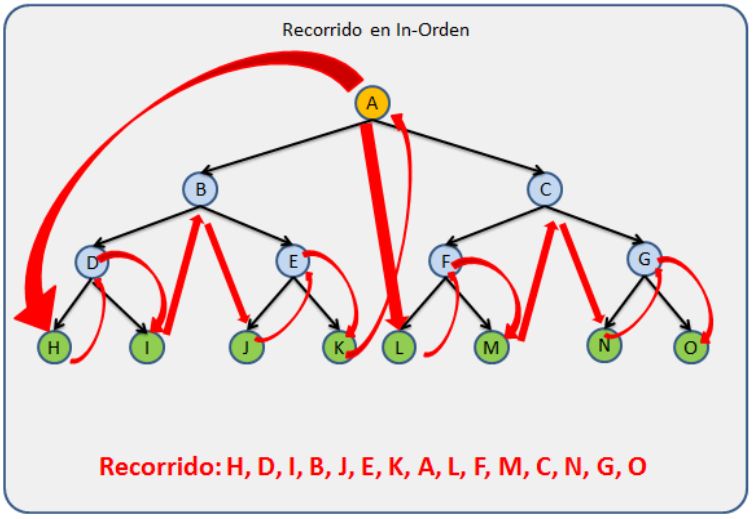

Fig. 6: En la imagen se muestra como es el recorrido In-Orden, Podemos apreciar que la Raíz no es el primero elemento en ser impreso pues este recorrido recorre su rama izquierda, luego la raíz del sub-árbol y luego la rama derecha.

Ponemos como ejemplo otro tipo de búsqueda la de amplitud:

- **Búsqueda en amplitud.**

  Se recorre primero la raíz, luego se recorren los demás nodos ordenados por el nivel al que pertenecen en orden de Izquierda a derecha.

  Este tipo de búsqueda se caracteriza por que la búsqueda se hace nivel por nivel y de izquierda a derecha.

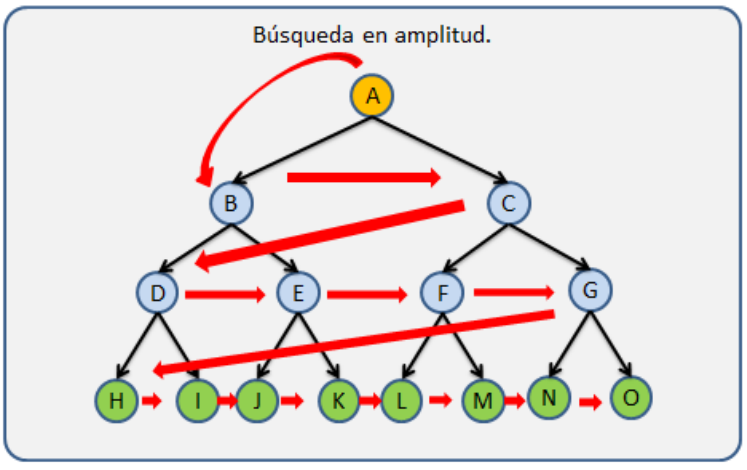

Fig. 7: En la imagen se observa como es que un nodo es buscado mediante la búsqueda en amplitud.

4. Metodología

- Se utilizó el lenguaje **Python 3.x**.
- Cada nodo del árbol se representa como una lista de tres elementos:
    1. Valor del nodo.
    2. Subárbol izquierdo.
    3. Subárbol derecho.
- Se desarrollaron funciones para crear árboles, insertar nodos y recorrerlos.
- Se incluyó una función de impresión rotada del árbol para facilitar la visualización.

---

5. Desarrollo / Implementación

A continuación, se presenta el código completo para implementar un árbol binario utilizando únicamente listas:

```
# Programa que le pide al usuario que ingrese primero la raiz del arbol
#  y luego los hijos que desee. Para terminar de ingresar hijos pide que se ingrese un 0
# Una vez corrido el programa muestra la forma del arbol a 90° y los recorridos:
# preorden,inorden y posorden

# Cada nodo se representa como: [valor, hijo_izquierdo, hijo_derecho]
def crear_arbol(valor):
    return [valor, [], []]

# La función insertar selecciona de que lado del árbol coloca el
# nuevo valor según sea mayor o menor que el nodo en que está parado.
# Toma dos parámetros: nodo que es el sub-arbol, y nuevo_valor que es
# el hijo ingresado

def insertar(nodo, nuevo_valor):
    if nuevo_valor < nodo[0]:
        insertar_izquierda(nodo, nuevo_valor)
    else:
        insertar_derecha(nodo, nuevo_valor)
```

```python
# La función insertar_izquierda toma los mismos parámetros que insertar
# y se fija si hay un valor en el nodo siguiente. Si no hay, crea el nuevo nodo.
# Si ya hay un valor vuelve a consultar (por recursividad con la función insertar)
# como es el nuevo_valor con el actual nodo. Esto lo repite hasta encontrar un
# nodo vacío para insertar

def insertar_izquierda(nodo, nuevo_valor):
    subarbol_izq = nodo[1]
    if subarbol_izq:
        insertar(subarbol_izq, nuevo_valor)
    else:
        nodo[1] = [nuevo_valor, [], []]



# La función insertar_derecha es similar a la izquierda
def insertar_derecha(nodo, nuevo_valor):
    subarbol_der = nodo[2]
    if subarbol_der:
        insertar(subarbol_der, nuevo_valor)
    else:
        nodo[2] = [nuevo_valor, [], []]




# Las funciones recorrido son recursivas y cambian entre
#  una y otra en la posición en que se encuentra la ejecución del print

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

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

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

def imprimir_arbol(arbol, nivel=0):
    if arbol:
        imprimir_arbol(arbol[2], nivel + 1)
        print('    ' * nivel + str(arbol[0]))
        imprimir_arbol(arbol[1], nivel + 1)

#-------------------------------------------------
# probar ingresando en este orden:
# 6 4 5 8 3 9 7
# para obtener un arbol con todas las ramas

raiz = int(input(" Ingrese el valor raíz "))
arbol = crear_arbol(raiz)

print ("Ahora inserte tantos hijos como quiera. Para terminar tecla '0'")
hijo = int(input("Ingrese un hijo "))
insertar(arbol, hijo)
while hijo != 0:
    hijo = int(input("Ingrese un hijo "))
    if hijo != 0:
        insertar(arbol, hijo)
```

```
print("Árbol visualizado (rotado 90°):")
imprimir_arbol(arbol)

# Recorridos
print("\nRecorrido Preorden:")
preorden(arbol)

print("\nRecorrido Inorden:")
inorden(arbol)

print("\nRecorrido Postorden:")
postorden(arbol)
```

Resultados obtenidos en la consola:

Ingrese el valor raíz 6

Ahora inserte tantos hijos como quiera. Para terminar tecla '0'

Ingrese un hijo 4

Ingrese un hijo 5

Ingrese un hijo 8

Ingrese un hijo 3

Ingrese un hijo 9

Ingrese un hijo 7

Ingrese un hijo 0

Árbol visualizado (rotado 90°):

```
    9
  8
    7
6
    5
  4
    3
```

Recorrido Preorden:

6 4 3 5 8 7 9

Recorrido Inorden:

3 4 5 6 7 8 9

Recorrido Postorden:

3 5 4 7 9 8 6

PS C:\Users>

6. Resultados

Con esta implementación pudimos crear y recorrer un árbol binario usando listas en Python. Usamos listas anidadas para representar cada nodo, con tres posiciones: el valor del nodo, el subárbol izquierdo y el subárbol derecho. Esto nos permitió ir armando el árbol de forma sencilla y visual.

Probamos el programa insertando varios números y pudimos ver cómo se organizaban en el árbol según si eran mayores o menores que el nodo actual. También funcionaron bien los tres tipos de recorridos (preorden, inorden y postorden), mostrando los valores en distintos órdenes según la estrategia usada.

Algo muy útil fue la función que imprime el árbol rotado 90 grados, ya que eso nos ayudó a ver cómo se armaba el árbol realmente. Al hacerlo así, pudimos entender mejor la estructura jerárquica.

Ventajas:

- Es una forma simple de entender cómo funciona un árbol binario.
- No hace falta saber programación orientada a objetos para poder implementarlo.
- El código es corto y fácil de seguir.
- Sirve mucho para aprender y practicar la recursividad.

Desventajas:

- Si el árbol es muy grande, las listas se vuelven difíciles de manejar.
- No se puede aprovechar bien la organización del código como con las clases.
- Es más complicado hacer cosas avanzadas como árboles balanceados o genéricos.

---

7. Conclusión

Este trabajo nos ayudó a entender cómo funcionan los árboles binarios y cómo podemos representarlos de una forma sencilla usando listas en Python. Aunque este método no es el más profesional ni el más eficiente para proyectos grandes, sí es muy útil para aprender.

Nos permitió enfocarnos en la lógica de cómo se arma y se recorre un árbol, sin tener que usar conceptos más avanzados como objetos, clases o punteros. Además, pudimos practicar el uso de funciones recursivas, que son muy importantes cuando se trabaja con estructuras como los árboles.

---

8. Bibliografía

- Python Software Foundation. https://docs.python.org
- https://www.oscarblancarteblog.com (extraccion de imagenes)