“Árboles Binarios con Listas”

Alumnos:
Paez Franco Leonel francoleonelpaez@gmail.com

Nowell Julieta julietanowell@gmail.com

Materia:
Programación I

Profesor/a:
Julieta Trapé

Fecha de Entrega:
09/06/2025

📑 Índice

1. Introducción

Las estructuras de datos son esenciales en programación, ya que permiten organizar y manipular información de manera eficiente. Entre ellas, los árboles, y en particular los árboles binarios, destacan por su capacidad para representar relaciones jerárquicas, optimizar búsquedas y facilitar la toma de decisiones en algoritmos avanzados.

En Python, los árboles binarios suelen implementarse mediante clases y objetos para representar nodos y sus conexiones. Sin embargo, existe una alternativa menos convencional pero igualmente válida: utilizar listas anidadas. Esta aproximación aprovecha la flexibilidad de las listas en Python, las cuales pueden contener otros elementos, incluyendo otras listas, para modelar la estructura jerárquica de un árbol.

En este contexto, exploraremos cómo representar árboles binarios mediante listas y casos prácticos de aplicación. Además, revisaremos los distintos tipos de recorridos (preorden, inorden y postorden) y su implementación en esta estructura simplificada.

2. Marco Teórico

Un árbol binario es una estructura de datos jerárquica en la que cada nodo tiene, como máximo, dos hijos: izquierdo y derecho.

Características principales

- Raíz: Nodo superior del que descienden todos los demás.
- Rama: Nodo con hijos
- Nodo hoja: Nodo sin hijos.
- Subárbol: Árbol que deriva de un nodo hijo.
- Profundidad: Número de aristas desde la raíz hasta un nodo.
- Altura: Máxima profundidad del árbol.

Un árbol binario puede representarse mediante listas anidadas, donde cada nodo es una lista con la estructura:

```
arbol = [valor, subarbol_izquierdo, subarbol_derecho]
```

Tipos de recorridos de un árbol binario:

- In-Orden (Izquierda - Raíz - Derecha)
- Pre-Orden (Raíz - Izquierda - Derecha)
- Post-Orden (Izquierda - Derecha - Raíz)

3. Caso Práctico

En este trabajo hemos abordado la teoría sobre los árboles binarios, desde su definición hasta sus características, propiedades y tipos de recorridos. Es momento de plasmar todo el código; en este caso, veremos cómo se construye un árbol binario. También incluiremos la construcción de un árbol general, para que puedan notarse las diferencias en el proceso de construcción.
Luego de eso, modificaremos el código para que el árbol, en cada paso que dé, nos devuelva por consola sus características. Esto lo haremos recorriéndolo de la forma que estamos acostumbrados a trabajar hasta ahora, es decir, desde arriba hacia abajo.
Finalmente, concluimos con los diferentes tipos de recorridos y sus comportamientos, los cuales también visualizamos en consola.

Empezamos con la creación de los árboles, esta es una estructura básica para la creación de un árbol binario:

arbol_binario.py U

arbol_binario.py > ...

```python
# Esta función crea un nodo para un árbol binario.
# Recibe un valor y asigna None a sus hijos izquierdo y derecho.
def crear_nodo(valor):
    return [valor, None, None]
# Estas funciones asignan un subárbol hijo al nodo padre.
# Dependiendo de la función, se agrega a la izquierda o a la derecha.
def agregar_izquierdo(padre, hijo):
    padre[1] = hijo

def agregar_derecho(padre, hijo):
    padre[2] = hijo
# Esta función recorre e imprime el árbol en preorden,
# utilizando sangría para reflejar la profundidad.
def imprimir_arbol(nodo, nivel=0):
    if nodo is not None:
        print("  " * nivel + str(nodo[0]))
        imprimir_arbol(nodo[1], nivel + 1)
        imprimir_arbol(nodo[2], nivel + 1)

# Ejemplo de construcción de un árbol binario simple y su visualización por consola.
# Crear nodos
raiz = crear_nodo("A")
nodo_b = crear_nodo("B")
nodo_c = crear_nodo("C")
nodo_d = crear_nodo("D")
nodo_e = crear_nodo("E")

# Construir el árbol
#        A
#       / \
#      B   C
#     / \
#    D   E

agregar_izquierdo(raiz, nodo_b)
agregar_derecho(raiz, nodo_c)
agregar_izquierdo(nodo_b, nodo_d)
agregar_derecho(nodo_b, nodo_e)
# Función para imprimir el árbol con sangría según nivel
def imprimir_arbol(nodo, nivel=0):
    if nodo is not None:
        print("  " * nivel + str(nodo[0]))
        imprimir_arbol(nodo[1], nivel + 1)
        imprimir_arbol(nodo[2], nivel + 1)
# Mostrar árbol estructurado
print("=== Visualización del Árbol ===\n")
imprimir_arbol(raiz)

```

```
PROBLEMAS   SALIDA   CONSOLA DE DEPURACIÓN   TERMINAL   PUERTOS

PS C:\Users\frank\Integrador_Programacion> & C:/Users/frank/AppData/Local/Programs/Python/Python313/python.exe c:/Users/frank/Integrador_Programacion/arbol_binario.py
=== Visualización del Árbol ===

A
  B
    D
    E
  C
PS C:\Users\frank\Integrador_Programacion>
```

Esta es la estructura básica de un árbol general. La diferencia clave está en que no hay restricción de ramas, pueden ser dos o más a diferencia de binario.

arbol_generico.py > imprimir_arbol

```python
# Crear un nodo para un árbol genérico.
# A diferencia de un árbol binario, cada nodo puede tener múltiples hijos.
def crear_nodo(valor, ramas=None):
    return [valor, ramas if ramas is not None else []]

# Agrega una rama (hijo) a un nodo existente.
def agregar_rama(nodo_raiz, valor_rama):
    rama = crear_nodo(valor_rama)
    nodo_raiz[1].append(rama)
    return rama

# Imprime el árbol con indentación para reflejar la jerarquía.
def imprimir_arbol(nodo, nivel=0):
    print("  " * nivel + nodo[0])
    for rama in nodo[1]:
        imprimir_arbol(rama, nivel + 1)
# Construcción del árbol:
# Raíz (100)
# ├── Rama 1 (50)
# │    ├── Rama 1.1 (25)
# │    │    ├── Hoja 1.1.1 (5)
# │    │    └── Hoja 1.1.2 (3)
# │    └── Hoja 1.2 (15)
# └── Rama 2 (30)
#      ├── Hoja 2.1 (12)
#      └── Hoja 2.2 (8)
# Construye un árbol general con valores asignados a cada nodo.
```

```python
def construir_arbol_con_valores():
    raiz = crear_nodo("Raíz (Valor: 100)")

    rama1 = agregar_rama(raiz, "Rama 1 (Valor: 50)")
    rama2 = agregar_rama(raiz, "Rama 2 (Valor: 30)")

    rama11 = agregar_rama(rama1, "Rama 1.1 (Valor: 25)")
    agregar_rama(rama1, "Hoja 1.2 (Valor: 15)")

    agregar_rama(rama2, "Hoja 2.1 (Valor: 12)")
    agregar_rama(rama2, "Hoja 2.2 (Valor: 8)")

    agregar_rama(rama11, "Hoja 1.1.1 (Valor: 5)")
    agregar_rama(rama11, "Hoja 1.1.2 (Valor: 3)")

    return raiz

# Mostrar el árbol por consola
print("=== Árbol General con Valores ===\n")
arbol = construir_arbol_con_valores()
imprimir_arbol(arbol)

```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS

PS C:\Users\frank\Integrador_Programacion> & C:/Users/frank/AppData/Local/Programs/Python/Python313/python.exe c:/Users/frank/Integrador_Programacion/arbol_generico.py
=== Árbol General con Valores ===

Raíz (Valor: 100)
  Rama 1 (Valor: 50)
    Rama 1.1 (Valor: 25)
      Hoja 1.1.1 (Valor: 5)
      Hoja 1.1.2 (Valor: 3)
    Hoja 1.2 (Valor: 15)
  Rama 2 (Valor: 30)
    Hoja 2.1 (Valor: 12)
    Hoja 2.2 (Valor: 8)
PS C:\Users\frank\Integrador_Programacion>
```

Los árboles binarios contienen diferentes propiedades, longitud de camino, profundidad, nivel, altura, grado, y peso. Vamos a ver cómo visualizar por consola sus respectivas propiedades. También existe la propiedad de orden, la cual indica el número máximo de hijos que puede tener un nodo, en este caso, al ser árboles binarios el número máximo siempre será 2, entonces hacer una función para calcular su orden sería innecesario.

arbol_binario_propiedades.py > ...

```python
def crear_nodo(valor): ···

def agregar_izquierdo(padre, hijo): ···

def agregar_derecho(padre, hijo): ···

def longitud_de_camino(nodo, valor_buscado, longitud_actual):
    if nodo is None:
        return -1
    if nodo[0] == valor_buscado:
        return longitud_actual
    resultado_izquierdo = longitud_de_camino(nodo[1], valor_buscado, longitud_actual + 1)
    if resultado_izquierdo != -1:
        return resultado_izquierdo
    return longitud_de_camino(nodo[2], valor_buscado, longitud_actual + 1)

def profundidad(nodo, valor_buscado, profundidad_actual):
    if nodo is None:
        return -1
    if nodo[0] == valor_buscado:
        return profundidad_actual
    resultado_izquierdo = profundidad(nodo[1], valor_buscado, profundidad_actual + 1)
    if resultado_izquierdo != -1:
        return resultado_izquierdo
    return profundidad(nodo[2], valor_buscado, profundidad_actual + 1)

def nivel(nodo, valor_buscado, nivel_actual):
    if nodo is None:
        return -1
    if nodo[0] == valor_buscado:
        return nivel_actual
    resultado_izquierdo = nivel(nodo[1], valor_buscado, nivel_actual + 1)
    if resultado_izquierdo != -1:
        return resultado_izquierdo
    return nivel(nodo[2], valor_buscado, nivel_actual + 1)

```

```python
def altura(nodo):
    if nodo is None:
        return -1
    if nodo[1] is None and nodo[2] is None:
        return 0
    altura_izquierda = altura(nodo[1])
    altura_derecha = altura(nodo[2])
    return 1 + max(altura_izquierda, altura_derecha)

def grado(nodo, valor_buscado):
    if nodo is None:
        return -1
    if nodo[0] == valor_buscado:
        grado_actual = 0
        if nodo[1] is not None:
            grado_actual = grado_actual + 1
        if nodo[2] is not None:
            grado_actual = grado_actual + 1
        return grado_actual

    resultado_izquierdo = grado(nodo[1], valor_buscado)
    if resultado_izquierdo != -1:
        return resultado_izquierdo

    resultado_derecho = grado(nodo[2], valor_buscado)
    return resultado_derecho

def peso(nodo):
    if nodo is None:
        return 0
    peso_izquierdo = peso(nodo[1])
    peso_derecho = peso(nodo[2])
    return 1 + peso_izquierdo + peso_derecho

```

```python
# Crear nodos
raiz = crear_nodo(1)
nodo2 = crear_nodo(2)
nodo3 = crear_nodo(3)
nodo4 = crear_nodo(4)
nodo5 = crear_nodo(5)
nodo6 = crear_nodo(6)
nodo7 = crear_nodo(7)

# Construir el árbol
#        1
#       / \
#      2   3
#     / \   \
#    4   5   6
#             \
#              7

agregar_izquierdo(raiz, nodo2)
agregar_derecho(raiz, nodo3)
agregar_izquierdo(nodo2, nodo4)
agregar_derecho(nodo2, nodo5)
agregar_derecho(nodo3, nodo6)
agregar_derecho(nodo6, nodo7)

# Mostrar propiedades
print("=== Propiedades del Árbol ===\n")
print("Altura del árbol:", altura(raiz))
print("Peso del árbol:", peso(raiz))
for valor in [1, 4, 5, 6, 7]:
    print(f"Longitud del camino hasta el nodo {valor}:", longitud_de_camino(raiz, valor, 0))
for valor in [1, 4, 5, 6, 7]:
    print(f"Profundidad del nodo {valor}:", profundidad(raiz, valor, 0))
for valor in [1, 4, 5, 6, 7]:
    print(f"Nivel del nodo {valor}:", nivel(raiz, valor, 0))
for valor in [1, 2, 3, 6, 7]:
    print(f"Grado del nodo {valor}:", grado(raiz, valor))

```

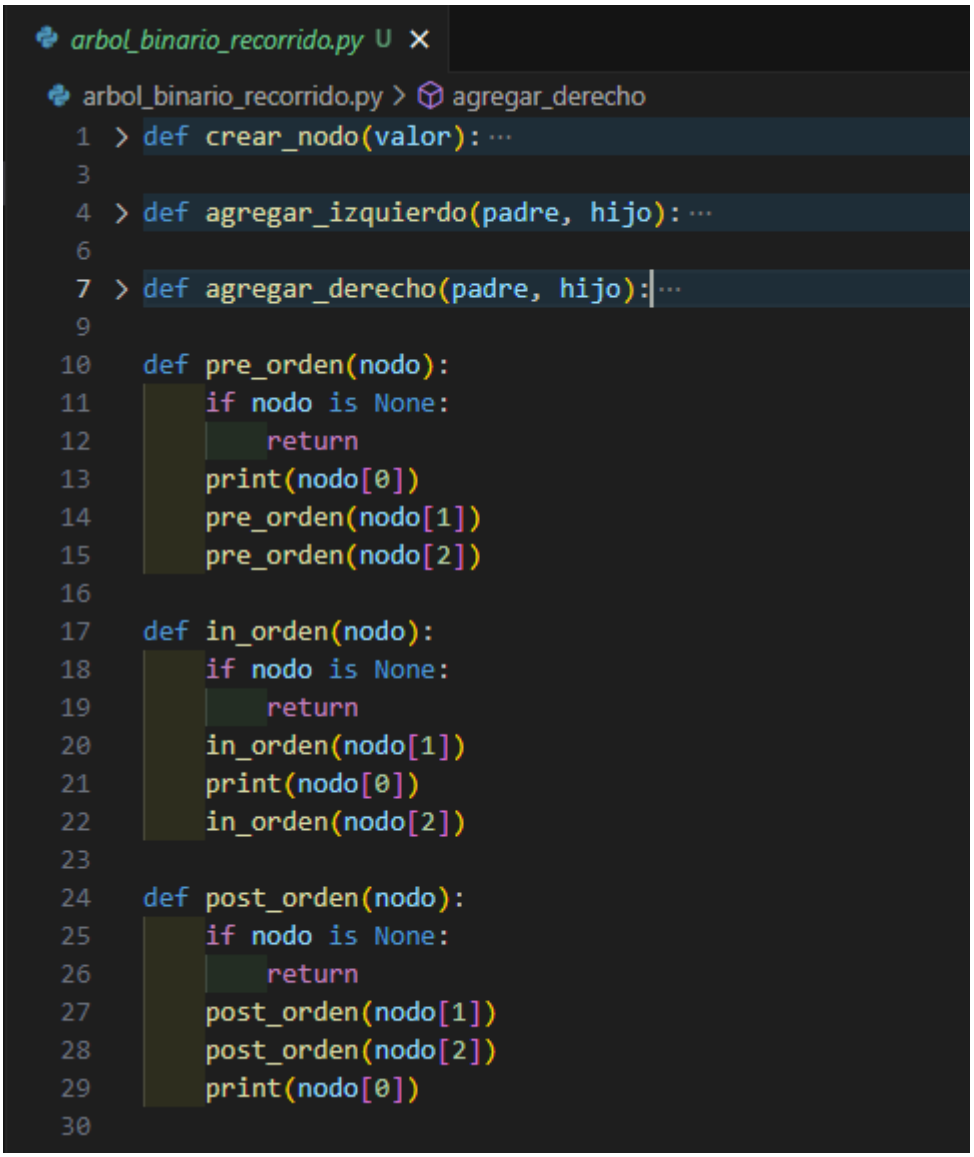
```
PS C:\Users\frank\Integrador_Programacion> & C:/Users/frank/AppData/Local/Programs/Python/Python313/python.exe c:/Users/frank/Integrador_Programacion/arbol_binario_propiedades.py
=== Propiedades del Árbol ===

Altura del árbol: 3
Peso del árbol: 7
Longitud del camino hasta el nodo 1: 0
Longitud del camino hasta el nodo 4: 2
Longitud del camino hasta el nodo 5: 2
Longitud del camino hasta el nodo 6: 2
Longitud del camino hasta el nodo 7: 3
Profundidad del nodo 1: 0
Profundidad del nodo 4: 2
Profundidad del nodo 5: 2
Profundidad del nodo 6: 2
Profundidad del nodo 7: 3
Nivel del nodo 1: 0
Nivel del nodo 4: 2
Nivel del nodo 5: -1
Nivel del nodo 6: -1
Nivel del nodo 7: -1
Grado del nodo 1: 2
Grado del nodo 2: 2
Grado del nodo 3: 1
Grado del nodo 6: 1
Grado del nodo 7: 0
PS C:\Users\frank\Integrador_Programacion>
```

Los árboles binarios cuentan con tres tipos diferentes de recorridos:

- In-Orden (Izquierda - Raíz - Derecha)
- Pre-Orden (Raíz - Izquierda - Derecha)
- Post-Orden (Izquierda - Derecha - Raíz)

Aca visualizamos cómo se comporta cada uno de esos nodos en consola



```python
# Crear nodos
raiz = crear_nodo(1)
nodo2 = crear_nodo(2)
nodo3 = crear_nodo(3)
nodo4 = crear_nodo(4)
nodo5 = crear_nodo(5)
nodo6 = crear_nodo(6)
nodo7 = crear_nodo(7)

# Construir el árbol
#        1
#       / \
#      2   3
#     / \   \
#    4   5   6
#             \
#              7

agregar_izquierdo(raiz, nodo2)
agregar_derecho(raiz, nodo3)
agregar_izquierdo(nodo2, nodo4)
agregar_derecho(nodo2, nodo5)
agregar_derecho(nodo3, nodo6)
agregar_derecho(nodo6, nodo7)

# Recorridos

print("=== Recorridos del Árbol ===\n")
print("Recorrido en Pre-orden:")
pre_orden(raiz)
print("Recorrido en In-orden:")
in_orden(raiz)
print("Recorrido en Post-orden:")
post_orden(raiz)

```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS

PS C:\Users\frank\Integrador_Programacion> & C:/Users/frank/AppData/Local/Programs/Python/Python313/python.exe c:/Users/frank/Integrador_Programacion/arbol_binario_recorrido.py
=== Recorridos del Árbol ===

Recorrido en Pre-orden:
1
2
4
5
3
6
7
Recorrido en In-orden:
4
2
5
1
3
6
7
Recorrido en Post-orden:
4
5
2
7
6
3
1
PS C:\Users\frank\Integrador_Programacion>
```

4. Metodología Utilizada

Para el diseño del código se elaboraron cuatro archivos diferentes con el objetivo de obtener una mejor perspectiva de las salidas en consola. Los dos primeros puntos a resolver fueron la creación de los árboles, tanto el binario como el general. Posteriormente, se dividieron en dos archivos distintos sus propiedades y su recorrido, de manera que se pudieran analizar con mayor detalle una vez impresos en consola.

Las principales dificultades no se presentaron durante la elaboración del código, sino en cómo mostrar los resultados de forma clara y explicativa para el usuario al momento de presentar las propiedades y el recorrido del árbol binario. En este paso, con ayuda de inteligencia artificial, se concluyó que lo más conveniente era, primero, visualizar la estructura del árbol mediante comentarios antes de construirlo y, luego, ejecutar las llamadas correspondientes. Esto se debe a que, si se imprimiera toda la información de forma simultánea (incluido el árbol y sus propiedades), la salida podría resultar visualmente sobrecargada, dificultando la comprensión y provocando la pérdida de detalles importantes.

Como recurso principal se utilizó Visual Studio Code para la implementación del código en el lenguaje Python, sin recurrir a librerías externas.

El trabajo fue dividido en dos facetas: teoría y puesta en práctica. Cada uno de los integrantes se encargó de una faceta, mientras que el otro realizó la revisión correspondiente, garantizando así una revisión cruzada y un aporte mutuo al desarrollo del proyecto.

5. Resultados Obtenidos

En el caso práctico, se logró visualizar de forma ideal la creación de los árboles en Python, tanto binarios como normales. También se logró mostrar con claridad sus propiedades y tipos de recorridos, para facilitar la visualización y complementar las explicaciones brindadas sobre los mismos.

Se presentaron dificultades en la etapa de visualización en consola, ya que la devolución hacía que se sobrecargue la pantalla de información, lo que generaba pérdida de claridad en conceptos importantes. Por este motivo, se tomó la decisión de dividir el código en cuatro archivos.

6. Conclusiones

Durante el transcurso del trabajo se aprendió cómo crear árboles en Python, así como sus propiedades y recorridos. Este trabajo representó un reto y una oportunidad para poner en práctica el uso de listas, lo que permitió profundizar y mejorar la comprensión de las mismas y de su potencial.

Dentro de las posibles mejoras, una podría ser centralizar todo en un solo archivo, para evitar tener que cargar cuatro diferentes, especialmente considerando que tres de ellos comparten funciones. Otra mejora podría ser evitar reescribir funciones que comparten el mismo código (como es el caso de longitud de camino, profundidad y nivel), ya que estas utilizan la misma lógica en su ejecución. Si bien en este caso se optó por incluirlas para mostrar cómo se construyen, su reutilización sería más eficiente.

La única dificultad, y razón por la cual se crearon los cuatro archivos, fue la visualización en consola. Al tener todo en un solo archivo, la salida resultaba sobrecargada y visualmente pesada, lo que motivó la decisión de dividir el código.

7. Bibliografía

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

https://docs.python.org/3/

8. Anexos

Repositorio en Github: https://github.com/FranPaez/Integrador_Programacion
Video: https://www.youtube.com/watch?v=i_LSfikKqmE