TECNICATURA UNIVERSITARIA
EN PROGRAMACIÓN
A DISTANCIA

Trabajo Práctico Integrador
Programación 1

Título: “Datos avanzados - Árboles”

Profesor:
Nico Quirós

Tutora:
Flor Gubiotti

Alumnos:
Sol Yoon - solyoon90@gmail.com
Margarita Zerpa – zerpamargarita2712@gmail.com

Comisión 23

Fecha de Entrega:
20 de junio de 2025

Contenido

Introducción 3
Marco Teórico 4
Caso práctico 13
Metodología utilizada 21
Resultados obtenidos 22
Conclusiones 23
Bibliografía 24

Introducción

El presente trabajo se enfoca en la implementación de árboles utilizando listas en el lenguaje de programación Python.

Los árboles son estructuras fundamentales en informática, aplicadas en diversos contextos como la representación de expresiones matemáticas, la organización jerárquica de archivos, la búsqueda eficiente de datos —como en los árboles binarios de búsqueda— y la toma de decisiones, como sucede con los árboles de decisión en inteligencia artificial. Comprender cómo se construyen y manipulan estas estructuras resulta esencial para el desarrollo de programas más eficientes y organizados.

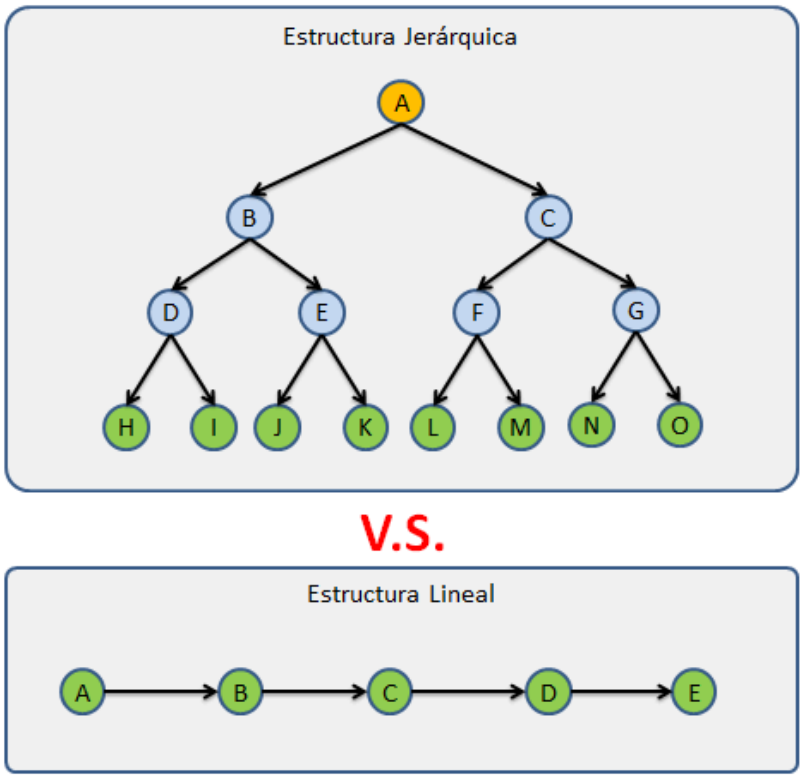
El objetivo principal del trabajo es comprender y demostrar cómo se puede construir y recorrer un árbol binario utilizando listas anidadas en Python, incluyendo operaciones básicas como la inserción, la búsqueda y los distintos tipos de recorrido.

Marco Teórico

¿Qué es un árbol en programación?

Un árbol es una estructura de datos no lineal que simula una jerarquía mediante una colección de nodos conectados. A diferencia de las estructuras lineales como listas o arreglos, los árboles permiten organizar información de forma jerárquica, partiendo de un nodo raíz y ramificándose en subnodos (llamados hijos). Cada nodo puede tener cero o más hijos, y cada nodo (excepto la raíz) tiene un único padre.

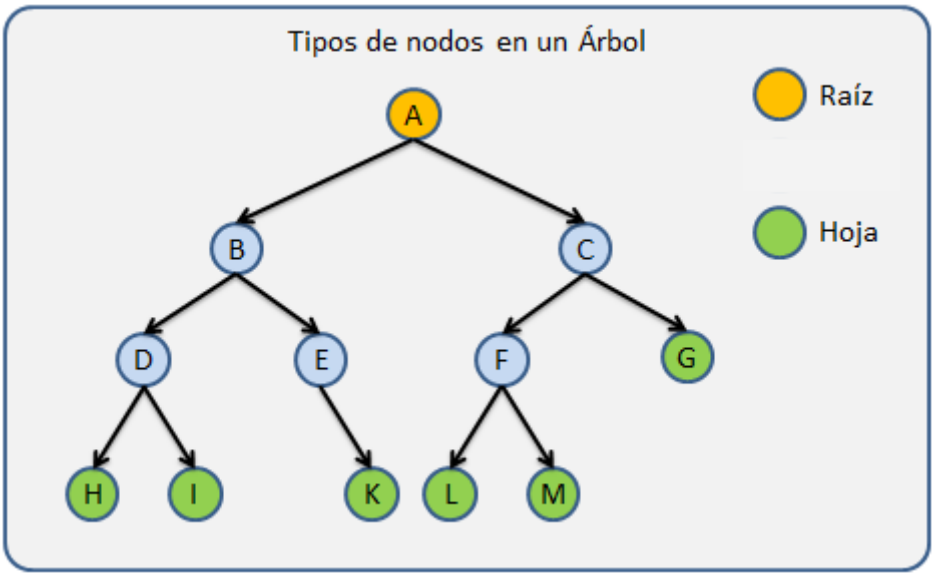
El tipo más común de árbol es el árbol binario, en el cual cada nodo tiene a lo sumo dos hijos: uno izquierdo y uno derecho. Si se imponen reglas de ordenamiento, como que los valores menores se ubican a la izquierda y los mayores a la derecha, se tiene un árbol binario de búsqueda (ABB o BST).

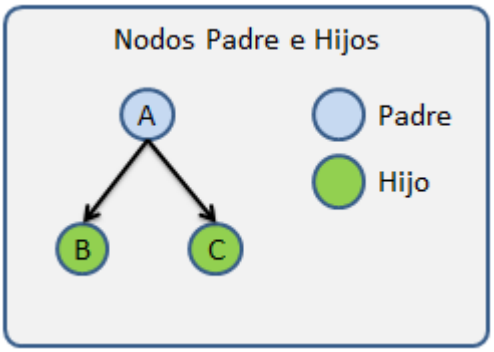


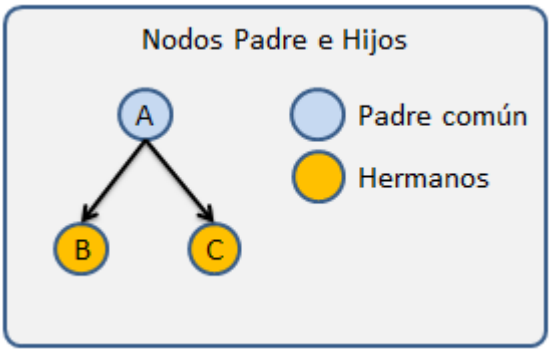
Componentes básicos de un árbol

- **Raíz (root):** Es el nodo principal del árbol, el único que no tiene padre. Desde él se derivan todos los demás nodos.

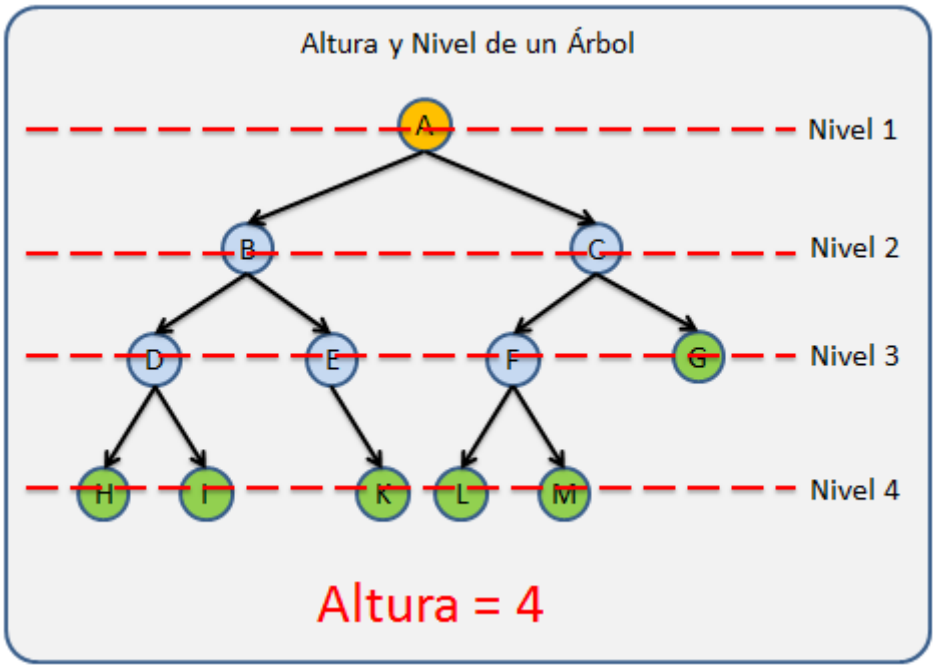
- **Padre:** Un nodo que tiene uno o más nodos hijos conectados directamente a él.
- **Hijo:** Cada uno de los nodos que dependen directamente de otro nodo (su padre).
- **Hermano:** Nodos que comparten el mismo padre.
- **Hoja (leaf):** Nodo que no tiene hijos. Representa el final de una rama.
- **Rama (branch):** Camino que va desde un nodo (usualmente la raíz) hasta una hoja.



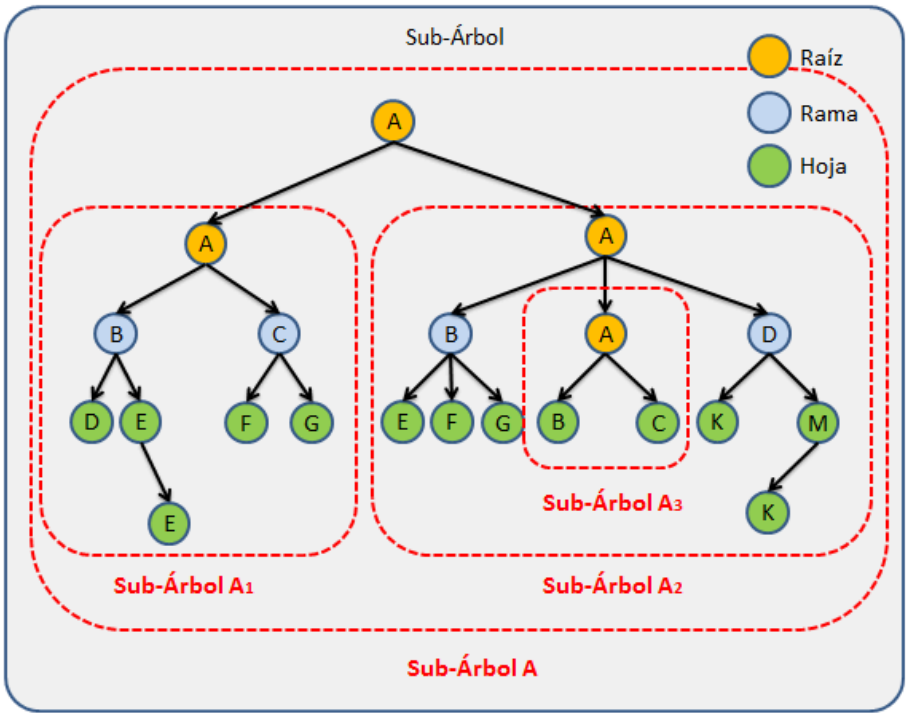




- **Altura:** Es la longitud del camino más largo desde la raíz hasta una hoja.
- **Nivel:** Es la distancia o profundidad de un nodo con respecto a la raíz (la raíz está en el nivel 0, sus hijos en el nivel 1, etc.).

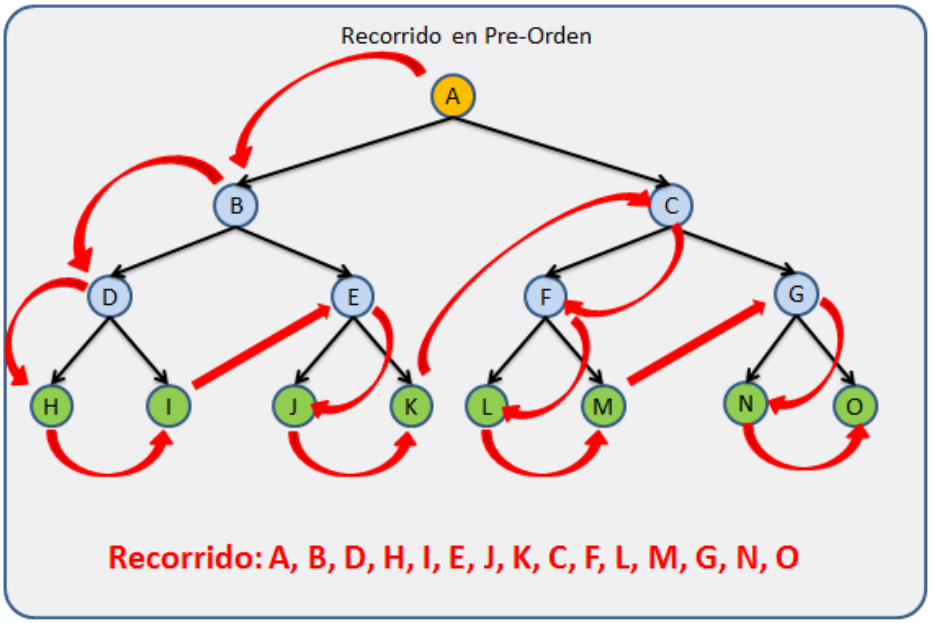


- **Subárbol:** Cualquier nodo con sus respectivos descendientes puede ser considerado un subárbol independiente dentro del árbol general.

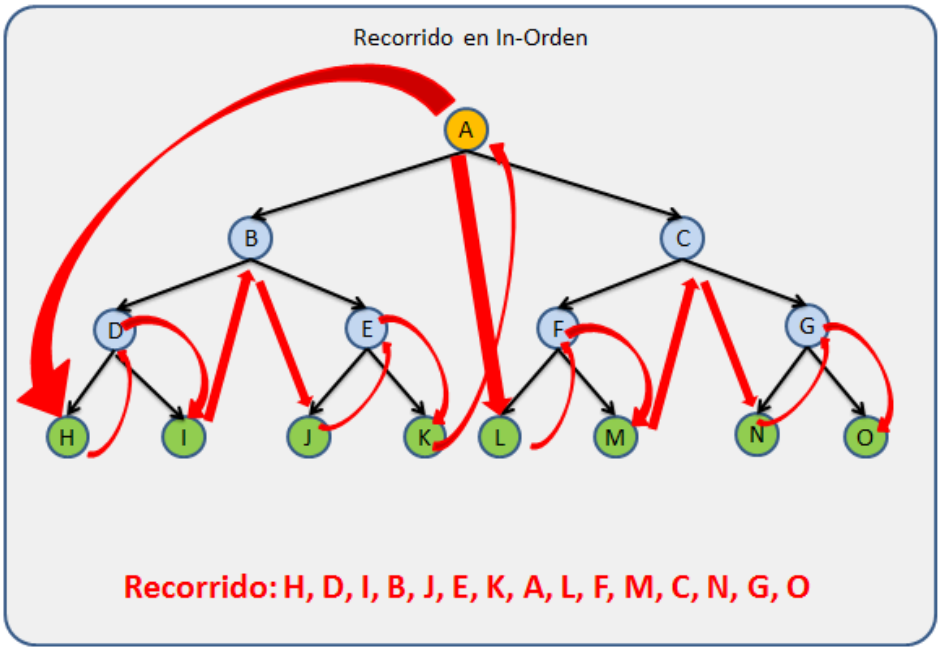


- **Recorridos:** Son las formas de visitar los nodos del árbol. Los principales son:

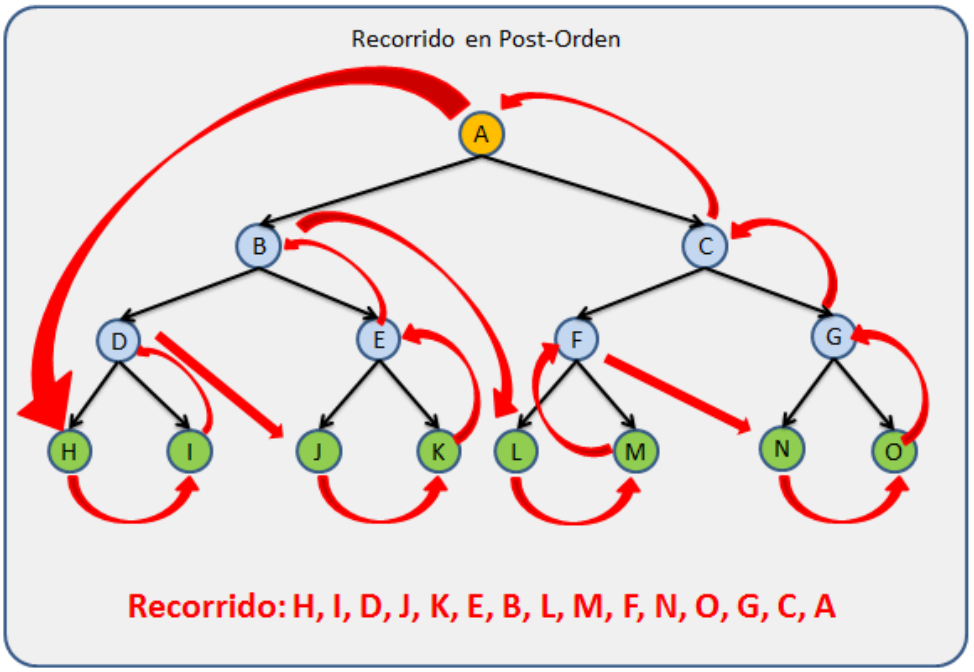
- **Preorden:** Se visita primero el nodo actual, luego el subárbol izquierdo y después el derecho.



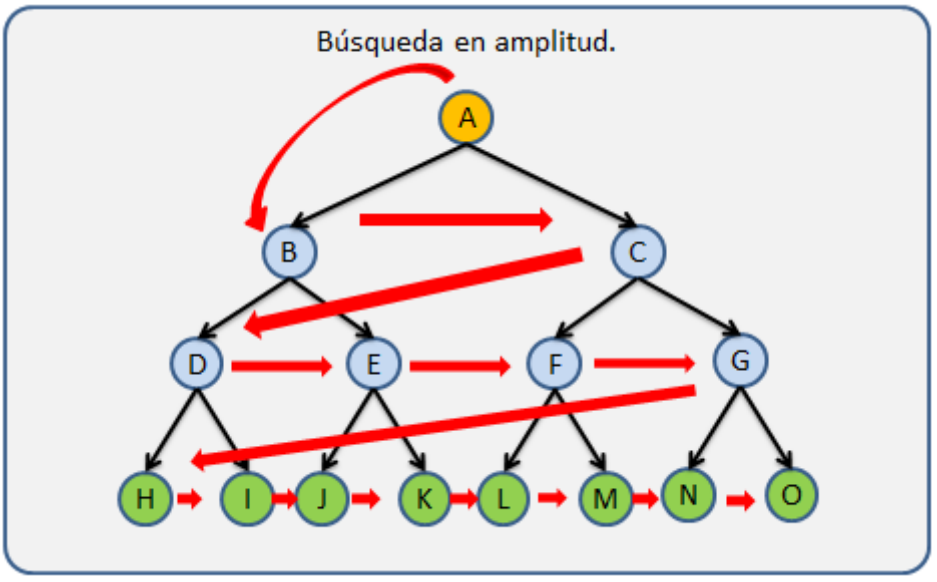
- **Inorden:** Se recorre primero el subárbol izquierdo, luego el nodo actual y finalmente el derecho.



- **Postorden:** Se recorren primero los subárboles izquierdo y derecho, y al final el nodo actual.



- **Por niveles (o amplitud):** Se recorren los nodos nivel por nivel, de arriba hacia abajo y de izquierda a derecha.



Listas en Python

En Python, una lista es una estructura de datos que permite almacenar múltiples elementos ordenados, que pueden ser de distintos tipos. Se define utilizando corchetes `[ ]`, y sus elementos se pueden acceder mediante índices.

```
mi_lista = [10, 20, 30]
print(mi_lista[0])  # Imprime 10
```

Las listas son **dinámicas** (pueden crecer o reducirse) y **mutables** (sus valores pueden modificarse). Además, pueden contener **otras listas como elementos**, lo que permite representar estructuras jerárquicas o anidadas, como es el caso de los árboles.

Esta característica es la que se aprovecha en este trabajo para representar nodos y subnodos del árbol mediante listas anidadas.

Árboles en Python usando listas

En Python, aunque es más habitual implementar árboles con clases (por ejemplo, usando objetos Nodo), también se pueden representar usando **listas anidadas**. Esta representación es útil para visualizar y manipular árboles de forma sencilla y didáctica.

Representación:

Una lista de tres elementos puede representar un nodo:

```
[valor, subárbol_izquierdo, subárbol_derecho]
```

Por ejemplo:

```
[10, [5, [], []], [15, [], []]]
```

Esto representa un árbol con raíz `10`, hijo izquierdo 5 y derecho `15`.

Ventajas y desventajas del uso de listas

Ventajas	Desventajas
Simple de implementar y entender	Menos flexible que clases
Visualmente clara para árboles pequeños	Difícil de escalar a árboles más complejos
Ideal para fines educativos	Mayor probabilidad de errores al acceder a índices

Funciones y operaciones comunes de listas usadas en árboles

Para implementar árboles con listas, se utilizan operaciones básicas que permiten manipular los nodos:

- **Indexación (`[ ]`)**: acceder a los elementos del nodo (valor, izquierdo, derecho).

```
nodo[0]  # valor
nodo[1]  # subárbol izquierdo
nodo[2]  # subárbol derecho
```

- **Asignación por índice**: modificar un subárbol o valor del nodo.

```
nodo[1] = [7, [], []]  # asignar nuevo subárbol izquierdo
```

- **Comparación con listas vacías**: para verificar si un subárbol está vacío (es decir, si es una hoja).

```
if nodo[1] == []:
    print("No tiene hijo izquierdo")
```

- `.append()`: no es esencial en este tipo de estructura fija (de tres elementos), pero puede usarse si se construyen árboles de forma dinámica.

- **Recursión y condicionales**: se usan para recorrer el árbol, realizar búsquedas o insertar nuevos valores.

```
def inorden(nodo):
    if nodo != []:
        inorden(nodo[1])
        print(nodo[0])
        inorden(nodo[2])
```

Estas operaciones permiten construir, recorrer y modificar árboles binarios de manera sencilla sin necesidad de clases.

Tipos de recorridos en árboles binarios

Los recorridos de un árbol binario son métodos sistemáticos para visitar todos los nodos. Existen varios tipos, cada uno con un propósito distinto:

- **Recorrido Inorden (In-order)**

Visita el subárbol izquierdo, luego la raíz, y finalmente el subárbol derecho.
En árboles binarios de búsqueda, este recorrido devuelve los elementos **ordenados de menor a mayor**.

```
def inorden(nodo):
    if nodo != []:
        inorden(nodo[1])
        print(nodo[0])
        inorden(nodo[2])
```

- **Recorrido Preorden (Pre-order)**

Visita primero la raíz, luego el subárbol izquierdo y por último el derecho.
Es útil para **clonar árboles** o para guardar su estructura.

```
def preorden(nodo):
    if nodo != []:
        print(nodo[0])
        preorden(nodo[1])
        preorden(nodo[2])
```

- **Recorrido Postorden (Post-order)**

Visita primero el subárbol izquierdo, luego el derecho y finalmente la raíz.
Es común en evaluaciones de expresiones y borrado de árboles.

```
def postorden(nodo):
    if nodo != []:
        postorden(nodo[1])
        postorden(nodo[2])
        print(nodo[0])
```

- **Recorrido por niveles (Level-order o por amplitud)**

Recorre los nodos nivel por nivel, de izquierda a derecha. Requiere una estructura adicional, como una cola (`queue`), y es útil para analizar árboles amplios o balanceados.

```
from collections import deque

def recorrido_por_niveles(arbol):
    if arbol == []:
        return
    cola = deque([arbol])
    while cola:
        nodo = cola.popleft()
        print(nodo[0], end=" ")
        if nodo[1] != []:
            cola.append(nodo[1])
        if nodo[2] != []:
            cola.append(nodo[2])
```

Uso de collections.deque en Python

Para implementar el recorrido por niveles se utilizó la estructura *deque* del módulo collections. Esta estructura permite trabajar con colas de manera eficiente, ya que permite insertar y quitar elementos de ambos extremos con mejor rendimiento que una lista común.
En este trabajo, *deque* se utilizó como una cola para mantener el orden en que deben visitarse los nodos, lo cual es esencial para recorrer el árbol nivel por nivel.

Caso práctico

Descripción del problema

Se plantea la construcción y manipulación de un **árbol binario** utilizando únicamente **listas anidadas** en Python. El objetivo es:

1. Crear un árbol vacío y nodos nuevos.
2. Insertar valores como hijos izquierdos o derechos.
3. Recorrer el árbol en **preorden**, **inorden** y **postorden**.
4. Visualizar su estructura en consola “girada” 90° para facilitar la lectura.

Este caso práctico permite aplicar las funciones teóricas vistas y validar su correcto funcionamiento en un ejemplo concreto.

Código fuente

```
from collections import deque

# 1. Crear un árbol vacío
def crear_arbol():
    # Devuelve un árbol vacío representado por una lista vacía.
    return []


# 2. Crear un nodo con valor n
def crear_nodo(valor):
    """
    Crea un nodo como lista de tres elementos:
    [valor, subárbol_izquierdo, subárbol_derecho]
    Inicialmente ambos subárboles están vacíos.
    """
    return [valor, [], []]

# 3. Insertar hijo izquierdo
def insertar_izquierda(nodo, nuevo_valor):
    """
    Si no existe hijo izquierdo, lo crea.
    Si existe, empuja el subárbol existente como hijo izquierdo
del nuevo nodo.
```

```
    """
    if nodo[1] == []:
        nodo[1] = crear_nodo(nuevo_valor)
    else:
        nodo[1] = [nuevo_valor, nodo[1], []]

# 4. Insertar hijo derecho
def insertar_derecha(nodo, nuevo_valor):
    """
    Igual que insertar_izquierda, pero para el subárbol derecho.
    """
    if nodo[2] == []:
        nodo[2] = crear_nodo(nuevo_valor)
    else:
        nodo[2] = [nuevo_valor, [], nodo[2]]

# 5. Recorridos clásicos
def preorden(nodo):
    """Visita: raíz → izquierdo → derecho."""
    if nodo != []:
        print(nodo[0], end=" ")
        preorden(nodo[1])
        preorden(nodo[2])

def inorden(nodo):
    """Visita: izquierdo → raíz → derecho."""
    if nodo != []:
        inorden(nodo[1])
        print(nodo[0], end=" ")
        inorden(nodo[2])

def postorden(nodo):
    """Visita: izquierdo → derecho → raíz."""
    if nodo != []:
        postorden(nodo[1])
        postorden(nodo[2])
        print(nodo[0], end=" ")

# 6. Recorrido por niveles
def recorrido_por_niveles(arbol):
    """
    Visita nivel por nivel de izquierda a derecha
```

```
    usando una cola auxiliar.
    """
    if arbol == []:
        return
    cola = deque([arbol])
    while cola:
        nodo = cola.popleft()
        print(nodo[0], end=" ")
        if nodo[1] != []:
            cola.append(nodo[1])
        if nodo[2] != []:
            cola.append(nodo[2])

# 7. Visualizar la estructura
def visualizar_arbol(nodo, nivel=0):
    """
    Imprime el árbol "girado" 90°.
    El hijo derecho aparece arriba y el izquierdo abajo.
    """
    if nodo != []:
        visualizar_arbol(nodo[2], nivel + 1)
        print("    " * nivel + str(nodo[0]))
        visualizar_arbol(nodo[1], nivel + 1)


# --- Ejecución de código
if __name__ == "__main__":
    # Crear raíz
    arbol = crear_nodo(10)

    # Agregar varios nodos
    insertar_izquierda(arbol, 5)
    insertar_derecha(arbol, 15)
    insertar_izquierda(arbol[1], 3)
    insertar_derecha(arbol[1], 7)
    insertar_derecha(arbol[2], 20)

    # 1) Visualización
    print("Visualización del árbol:")
    visualizar_arbol(arbol)

    # 2) Recorridos
```

```
    print("\nRecorrido Preorden: ", end="")
    preorden(arbol)
    print("\nRecorrido Inorden:  ", end="")
    inorden(arbol)
    print("\nRecorrido Postorden:", end=" ")
    postorden(arbol)
    print("\nRecorrido por niveles:", end=" ")
    recorrido_por_niveles(arbol)
    print()
```

Salida de consola

```
from collections import deque

# 1. Crear un árbol vacío
def crear_arbol():
    # Devuelve un árbol vacío representado por una lista vacía.
    return []
```

```
# 2. Crear un nodo con valor n
def crear_nodo(valor):
    """
    Crea un nodo como lista de tres elementos:
    [valor, subárbol_izquierdo, subárbol_derecho]
    Inicialmente ambos subárboles están vacíos.
    """
    return [valor, [], []]
```

```
# 3. Insertar hijo izquierdo
def insertar_izquierda(nodo, nuevo_valor):
    """
    Si no existe hijo izquierdo, lo crea.
    Si existe, empuja el subárbol existente como hijo izquierdo del nuevo nodo.
    """
    if nodo[1] == []:
        nodo[1] = crear_nodo(nuevo_valor)
    else:
        nodo[1] = [nuevo_valor, nodo[1], []]
```

```python
# 4. Insertar hijo derecho
def insertar_derecha(nodo, nuevo_valor):
    """
    Igual que insertar_izquierda, pero para el subárbol derecho.
    """
    if nodo[2] == []:
        nodo[2] = crear_nodo(nuevo_valor)
    else:
        nodo[2] = [nuevo_valor, [], nodo[2]]
```

```python
# 5. Recorridos clásicos
def preorden(nodo):
    """Visita: raíz → izquierdo → derecho."""
    if nodo != []:
        print(nodo[0], end=" ")
        preorden(nodo[1])
        preorden(nodo[2])

def inorden(nodo):
    """Visita: izquierdo → raíz → derecho."""
    if nodo != []:
        inorden(nodo[1])
        print(nodo[0], end=" ")
        inorden(nodo[2])

def postorden(nodo):
    """Visita: izquierdo → derecho → raíz."""
    if nodo != []:
        postorden(nodo[1])
        postorden(nodo[2])
        print(nodo[0], end=" ")
```

```python
# 6. Recorrido por niveles
def recorrido_por_niveles(arbol):
    """
    Visita nivel por nivel de izquierda a derecha
    usando una cola auxiliar.
    """
    if arbol == []:
        return
    cola = deque([arbol])
    while cola:
        nodo = cola.popleft()
        print(nodo[0], end=" ")
        if nodo[1] != []:
            cola.append(nodo[1])
        if nodo[2] != []:
            cola.append(nodo[2])
```

```python
# 7. Visualizar la estructura
def visualizar_arbol(nodo, nivel=0):
    """
    Imprime el árbol "girado" 90°.
    El hijo derecho aparece arriba y el izquierdo abajo.
    """
    if nodo != []:
        visualizar_arbol(nodo[2], nivel + 1)
        print("    " * nivel + str(nodo[0]))
        visualizar_arbol(nodo[1], nivel + 1)
```

Ejecución de Código

```python
# --- Ejecución de código
if __name__ == "__main__":
    # Crear raíz
    arbol = crear_nodo(10)

    # Agregar varios nodos
    insertar_izquierda(arbol, 5)
    insertar_derecha(arbol, 15)
    insertar_izquierda(arbol[1], 3)
    insertar_derecha(arbol[1], 7)
    insertar_derecha(arbol[2], 20)
    # 1) Visualización
    print("Visualización del árbol:")
    visualizar_arbol(arbol)

    # 2) Recorridos
    print("\nRecorrido Preorden: ", end="")
    preorden(arbol)
    print("\nRecorrido Inorden:  ", end="")
    inorden(arbol)
    print("\nRecorrido Postorden:", end=" ")
    postorden(arbol)
    print("\nRecorrido por niveles:", end=" ")
    recorrido_por_niveles(arbol)
    print()
```

```
Visualización del árbol:
      20
   15
10
      7
   5
      3

Recorrido Preorden: 10 5 3 7 15 20
Recorrido Inorden:  3 5 7 10 15 20
Recorrido Postorden: 3 7 5 20 15 10
Recorrido por niveles: 10 5 15 3 7 20
```

Decisiones de diseño

Para la implementación del árbol binario se optó por utilizar **listas anidadas** en lugar de clases u otras estructuras más complejas. Esta elección responde a la necesidad de representar un árbol de forma simple y compatible con el lenguaje Python, cumpliendo además con el enfoque didáctico solicitado en el trabajo.

Las funciones de inserción fueron diseñadas de modo que, si un nodo ya tiene un hijo izquierdo o derecho, el nuevo valor no sobrescriba el anterior, sino que lo desplace como subárbol del nuevo nodo insertado. Esto permite mantener la información ya almacenada y construir árboles más dinámicos.

La visualización del árbol se implementó con una función recursiva que imprime los nodos "girando" el árbol 90 grados: el subárbol derecho se muestra arriba y el izquierdo debajo. Este formato facilita la comprensión de la estructura jerárquica desde la raíz hacia las hojas.

En cuanto al recorrido por niveles, se optó por utilizar una cola (`deque`) que permite recorrer el árbol por anchura, nivel a nivel, algo que no es tan sencillo con recursión. Este recorrido es útil para observar cómo crece el árbol en términos de profundidad y orden de inserción.

Validación del funcionamiento

El funcionamiento de las funciones implementadas fue validado a través de diferentes pruebas, comenzando por la creación de un árbol con una sola raíz, luego agregando nodos a izquierda y derecha, y finalmente construyendo un árbol más complejo con múltiples niveles.

Se verificó que los recorridos en **preorden**, **inorden** y **postorden** imprimen los nodos en el orden correcto según la teoría. Además, el **recorrido por niveles** confirmó que el uso de una estructura de cola permite explorar el árbol horizontalmente, como era esperado.

La función de **visualización** también demostró ser útil, ya que al ejecutar el programa con diferentes configuraciones del árbol, se pudo observar de manera clara cómo se distribuyen los nodos y su jerarquía.

En conclusión, todas las funciones trabajaron de acuerdo a lo previsto y permitieron representar, recorrer y visualizar un árbol binario utilizando exclusivamente listas en Python.

Metodología utilizada

El desarrollo del trabajo se llevó a cabo siguiendo los siguientes pasos:

- **Investigación teórica previa:**
  - Se consultaron materiales introductorios sobre estructuras de datos, enfocándose en árboles binarios.
  - Se revisaron implementaciones con listas en Python y documentación oficial del lenguaje.
- **Diseño del modelo de árbol:**
  - Se definió la representación del nodo como una lista con tres elementos: [valor, subárbol_izquierdo, subárbol_derecho].
  - Se planificó el diseño de funciones para inserción, visualización y recorridos.
- **Implementación del código:**
  - Se utilizaron funciones recursivas para los distintos tipos de recorridos (preorden, inorden, postorden).
  - Se usó collections.deque para el recorrido por niveles, aprovechando su eficiencia como estructura de cola.
  - El entorno de desarrollo fue **Visual Studio Code** con Python 3.
- **Pruebas y validación:**
  - Se crearon distintos árboles de prueba para comprobar el correcto funcionamiento de las funciones.
  - Se probaron casos especiales, como árboles vacíos o con un solo nodo.
  - Se corrigieron errores detectados en la lógica de inserción o en el orden de impresión.
- **Documentación del trabajo:**
  - Se comentaron todas las funciones del código.
  - Se redactó el presente informe siguiendo la estructura solicitada por la consigna.

Resultados obtenidos

Como resultado del trabajo se logró construir un árbol binario funcional utilizando listas, sin recurrir a clases ni librerías externas. Las funciones desarrolladas permiten:

- Crear nodos y árboles vacíos.
- Insertar nodos tanto a la izquierda como a la derecha, preservando los subárboles ya existentes.
- Visualizar el árbol con una presentación clara, que facilita el análisis de su estructura.
- Recorrer el árbol en **preorden**, **inorden**, **postorden** y **por niveles**, obteniendo en cada caso el orden correcto de los nodos.

Los casos de prueba mostraron salidas acordes a lo esperado en cada recorrido y demostraron que la lógica implementada es consistente. Durante el proceso se detectaron algunos errores menores, como problemas en la indentación o confusión en la posición de los elementos en la lista (por ejemplo, confundir la izquierda con la derecha), los cuales fueron corregidos tras revisar el diseño y realizar pruebas controladas.

El programa también fue validado con árboles vacíos y árboles con un solo nodo, lo que permitió confirmar que las funciones responden correctamente ante diferentes situaciones.

Conclusiones

La realización de este trabajo permitió entender mejor cómo funciona una estructura de datos como el árbol binario, y cómo puede representarse usando listas en Python.

Durante el desarrollo se practicaron conceptos como la recursividad, el uso de listas anidadas y los distintos tipos de recorridos de un árbol. También se trabajó en la creación de funciones reutilizables y en la forma de organizar datos de manera jerárquica.

Una posible mejora para más adelante sería agregar validaciones al momento de insertar nodos, o adaptar la estructura para permitir árboles con más de dos ramas. También sería interesante implementar funciones para buscar y eliminar nodos.

En resumen, este trabajo fue una buena oportunidad para aplicar ideas importantes de programación y estructuras de datos a través de un ejercicio concreto y útil.

Bibliografía

- Python Software Foundation. (2025). *Documentación oficial de Python 3*. https://docs.python.org/3/ (Accedido el 8 de junio de 2025)
- Geeks for Geeks. (s.f.). *Binary Tree Data Structure*. https://www.geeksforgeeks.org/binary-tree-data-structure/ (Accedido el 8 de junio de 2025)
- keepcoding.io. (2024). *¿Qué son los árboles binarios en programación?*. https://keepcoding.io/blog/arboles-binarios-en-programacion/ (Accedido el 8 de junio de 2025)
- oscarblancarteblog.com (2014) Estructura de datos – Árboles https://www.oscarblancarteblog.com/2014/08/22/estructura-de-datos-arboles/ (Accedido el 9 de junio de 2025)
- Árbol General - Recorrido Por Nivel https://www.youtube.com/watch?v=UCumF3yXYoU (Accedido el 8 de junio de 2025)