**Título del trabajo:** Estructuras de datos avanzadas - Árboles en Python

**Alumnos:** Manuel Galarza - manu.galarza@gmail.com
Gabriel Etchegoyen - gabrieletchegoyen@gmail.com

**Materia:** Programación I

**Profesor/a:** Julieta Trapé

**Fecha de Entrega:** 09/06/2025

---

## Índice:



---

## 1. Introducción

Los árboles son una de las estructuras de datos más versátiles y esenciales en la programación y la informática. A diferencia de las estructuras lineales como arrays o listas enlazadas, los árboles organizan los datos de forma jerárquica, lo que los hace ideales para representar relaciones con múltiples niveles de dependencia.

Desde sistemas de archivos y bases de datos hasta algoritmos de inteligencia artificial y redes jerárquicas, los árboles permiten operaciones eficientes como búsqueda, inserción y eliminación en tiempo logarítmico en casos balanceados. Su capacidad para modelar escenarios del mundo real, como organigramas empresariales o torneos deportivos, los convierte en una herramienta fundamental para cualquier desarrollador.

A continuación, explicaremos los conceptos básicos de los árboles, sus tipos principales y las ventajas que ofrecen frente a otras estructuras de datos.

---

## 2. Marco teórico

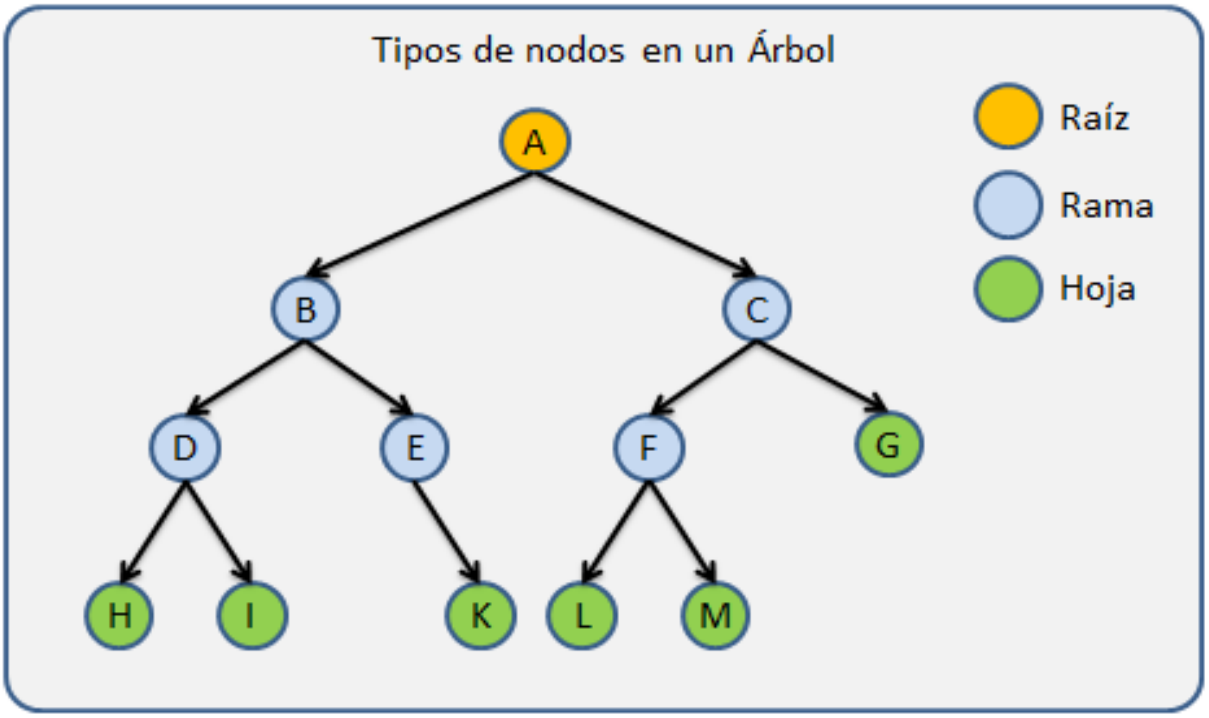
**Introducción**

Los árboles son estructuras de datos no lineales y jerárquicas que organizan elementos (nodos) en relaciones de padre-hijo. A diferencia de las estructuras lineales (listas, arrays), los árboles permiten modelar relaciones con múltiples niveles de dependencia, lo que los hace esenciales en áreas como bases de datos, sistemas de archivos, inteligencia artificial y más.

**Definición y Componentes**

Un árbol se compone de:

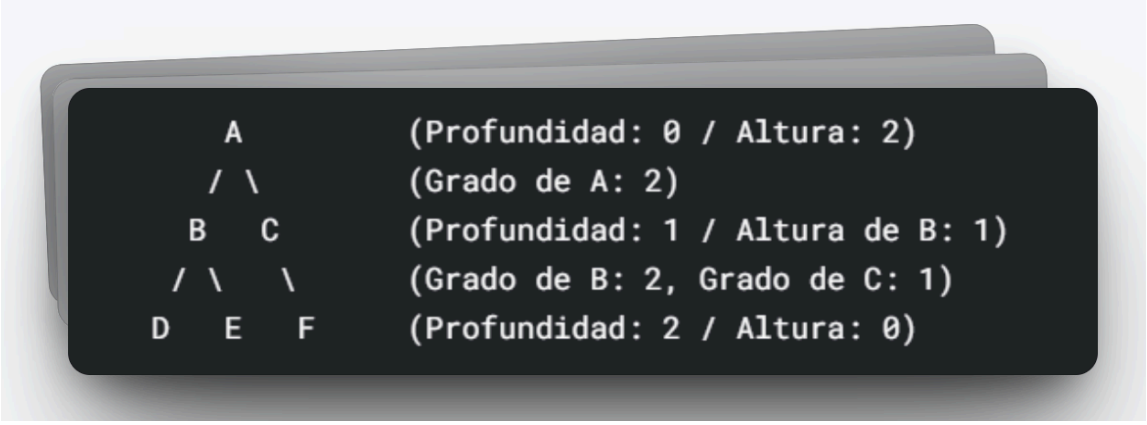
-Nodo: Unidad fundamental que contiene datos y referencias a otros nodos
-Raíz: Nodo superior del que derivan todos los demás
-Hijos: Nodos que descienden directamente de otro nodo (padre)
-Hoja: Nodo sin hijos
-Altura: Longitud del camino más largo desde la raíz hasta una hoja
-Arista: Llamamos así a las conexiones entre nodos ( ej: A⟶B ).




**Propiedades Clave**

- Altura: Longitud del camino más largo desde la raíz a una hoja.
- Profundidad: Longitud del camino desde la raíz a un nodo específico.
- Grado: Número máximo de hijos por nodo (ej: árbol binario ⟶ grado 2).

- Peso: Número total de nodos en el árbol.




**Tipos principales de Árboles**

1. Árbol Binario

Es un tipo de árbol donde cada "nodo" (como una cajita con datos) puede tener máximo 2 hijos: uno a la izquierda y otro a la derecha.

-Árbol Binario de Búsqueda (ABB)

Está organizado de forma que:

- Los valores menores van a la izquierda.
- Los valores mayores van a la derecha. Esto hace que buscar datos sea más rápido.

-Árbol Balanceado (como AVL o Rojo-Negro)

Son árboles binarios que se ajustan solos para que no queden muy desbalanceados (es decir, con un lado mucho más largo que el otro).

Esto mantiene el árbol "parejo", ayudando a que buscar, agregar o borrar datos sea más rápido.

2. Árboles n-arios

Son árboles donde cada nodo puede tener más de 2 hijos.

Se usan cuando se necesita manejar mucha información organizada, como en bases de datos.

Ejemplo: los árboles B (B-trees), muy usados en sistemas de archivos y bases de datos relacionales principalmente.

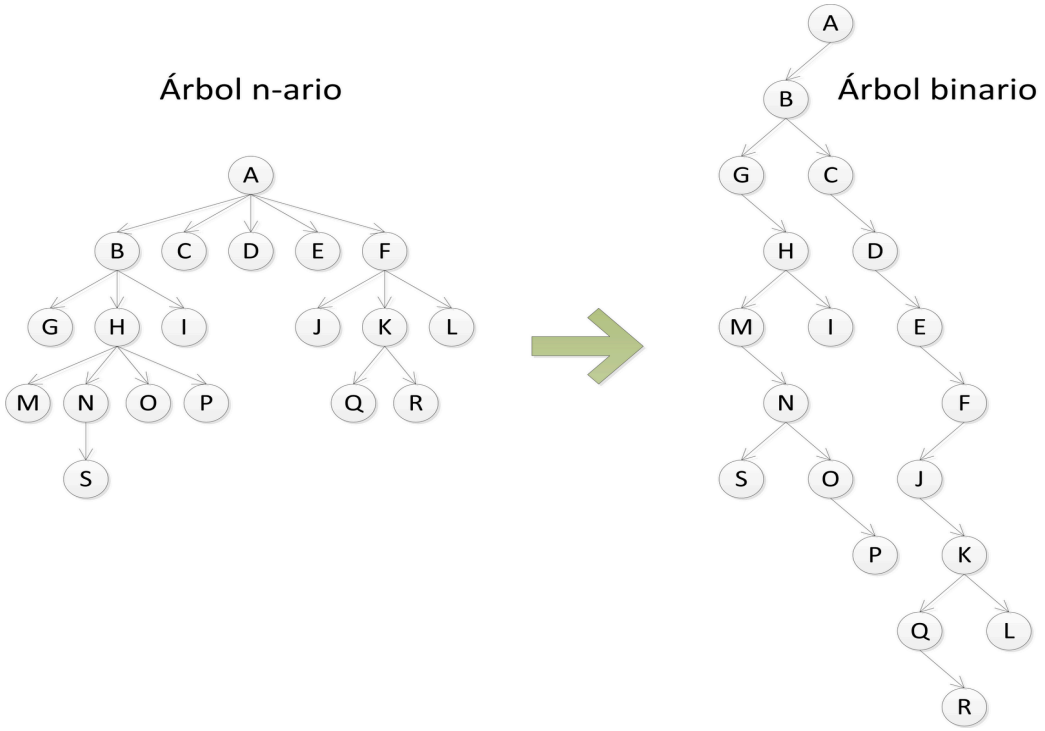
3. Árboles especializados

-Árboles de Segmentos (Segment Trees)

Son estructuras optimizadas para consultas de rangos y actualizaciones eficientes en arreglos. Organizan los datos en un árbol binario donde cada nodo almacena información agregada (como sumas, mínimos o máximos) de un segmento del arreglo. Son fundamentales en algoritmos avanzados, sistemas de bases de datos y aplicaciones que requieren procesamiento rápido de intervalos, como sistemas geográficos o análisis de series temporales.

-Árboles de decisión

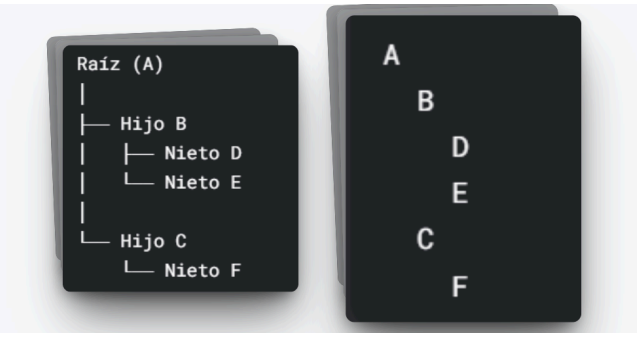
Estos árboles, utilizados en machine learning, modelan decisiones mediante reglas simples en forma de estructura jerárquica. Cada nodo interno representa una pregunta sobre los datos (como "¿edad > 30?"), las ramas son posibles respuestas y las hojas contienen los resultados finales (clases o valores predictivos).




**Representaciones**

- Gráfica: Nodos y aristas (visualización estándar), por lo general son dibujos con circulos y lineas
- Conjuntos anidados: Usando teoría de conjuntos se utilizan llaves para anidar nodos hijos dentro de padres. Ejemplo: {A, {B, {D}, {E}}, {C, {F}}}.
- Paréntesis anidados: Utilizan una jerarquía con paréntesis. Ejemplo: `(A (B (D)(E))(C (F)))`.

- Indentación: Sangrías para mostrar jerarquía. Uso de sangrías (tab-espaciado) para mostrar los niveles del árbol. Ejemplo:




**Aplicaciones**

- Sistemas de archivos: Directorios y subdirectorios.
- Bases de datos: Índices para búsquedas rápidas (ABB).
- Expresiones matemáticas: Árboles de sintaxis (ej: `(3+5)*(2-1)`).
- Machine Learning: Árboles de decisión para clasificación.
- Redes: Enrutamiento jerárquico.

**Operaciones Básicas**

- Inserción/Búsqueda/Eliminación: según el tipo de caso que sea el árbol (balanceado o desbalanceado) puede ser rápido o lento respectivamente.
- Recorridos:

Preorden: Raíz → Izquierda → Derecha. Ejemplo: A → B → D → E → C → F.

Inorden: Izquierda → Raíz → Derecha. En ABB: Ordena los datos de menor a mayor.

Ejemplo: D → B → E → A → F → C.

Postorden: Izquierda → Derecha → Raíz. Ejemplo: D → E → B → F → C → A.

**Ventajas y Desventajas**

-Ventajas

Búsqueda eficiente según el caso.

Modelado de jerarquías naturales.

Flexibilidad en aplicaciones reales.

-Desventajas

Complejidad en balanceo (AVL, etc.).

Mayor consumo de memoria que listas.

Operaciones de eliminación complejas.

---

## 3. Caso práctico

**Representación y visualización de un Árbol Binario en Python**

En este caso práctico se implementa un árbol binario utilizando listas anidadas en Python.
Se incluyen los tres recorridos clásicos (inorden, preorden y postorden)

Inicio del script

```
# Crear nodo del árbol
def crear_nodo(valor, izquierdo=None, derecho=None):
    return [valor, izquierdo, derecho]

# Recorridos
def inorden(arbol):
    if arbol is not None:
        inorden(arbol[1])
        print(arbol[0], end=' ')
        inorden(arbol[2])
```

```
def preorden(arbol):
    if arbol is not None:
        print(arbol[0], end=' ')
        preorden(arbol[1])
        preorden(arbol[2])

def postorden(arbol):
    if arbol is not None:
        postorden(arbol[1])
        postorden(arbol[2])
        print(arbol[0], end=' ')

# Mostrar árbol visualmente
def mostrar_arbol(arbol, nivel=0):
    if arbol is not None:
        mostrar_arbol(arbol[2], nivel + 1)
        print('    ' * nivel + str(arbol[0]))
        mostrar_arbol(arbol[1], nivel + 1)



# Crear un árbol binario de ejemplo

arbol = crear_nodo('A', crear_nodo('B', crear_nodo('D'), crear_nodo('E')), crear_nodo('C',
None, crear_nodo('F')))

# Mostrar visualización del árbol
print("Árbol binario:")
mostrar_arbol(arbol)

# Recorridos
print("\nRecorrido inorden:")
inorden(arbol)

print("\nRecorrido preorden:")
preorden(arbol)

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

fin del script
```

## 4. Metodología utilizada

- Estudio del tema elegido y sus características “ Arboles Binarios”
- Consulta de documentación oficial de Python en https://docs.python.org/3
- Implementación del código en lenguaje de programación Python, utilización de editor de código Visual Studio Code en versión 3.11 de python
- Ensayo y realización de codigo para generar el resultado de un árbol binario con los tipos de recorridos

---

## 5. Resultados obtenidos

- Construcción de un árbol binario utilizando listas anidadas, representando cada nodo junto a sus respectivos hijos izquierdo y derecho.
- Implementación correcta de los tres recorridos clásicos:
    - Inorden (izquierda - raíz - derecha)
    - Preorden (raíz - izquierda - derecha)
    - Postorden (izquierda - derecha - raíz)
- Visualización jerárquica simple del árbol en consola, representando la estructura del árbol de forma vertical con indentación según el nivel de profundidad.

---

## 6. Conclusiones

Los árboles son estructuras fundamentales en la informática por su capacidad para representar datos jerárquicos y facilitar operaciones eficientes como búsquedas, inserciones y recorridos. Su elección y correcta implementación dependen del tipo de problema a resolver.
Dominar su uso permite diseñar algoritmos más eficientes, escalables y adecuados para resolver desafíos complejos en diversas áreas de la programación y la ciencia de datos.

---

## 7. Bibliografía


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

  Documentación oficial de Python para comprensión de manejos de listas, estructuras de datos
- Apuntes y videos de la materia Programación I (UTN) sobre Árboles en Python


---

## 8. Anexos

Enlace a vídeo explicativo :

https://youtu.be/9MW5o3x5MTY

Enlace a repositorio en Github:

https://github.com/Gabriel071185/trabajo_integrador_programacion1_utn.git

**Capturas del programa en funcionamiento en tres partes con sus respectivas capturas.**

**Primera parte:** Funciones dedicadas a crear los nodos (valor , una rama izquierda inicializada en valor None y una derecha también en valor None) , los tres tipos de recorridos mencionados y una función que visualiza el árbol generado.

```python
# Crear nodo del árbol
def crear_nodo(valor, izquierdo=None, derecho=None):
    return [valor, izquierdo, derecho]

# Recorridos
def inorden(arbol):
    if arbol is not None:
        inorden(arbol[1])
        print(arbol[0], end=' ')
        inorden(arbol[2])

def preorden(arbol):
    if arbol is not None:
        print(arbol[0], end=' ')
        preorden(arbol[1])
        preorden(arbol[2])

def postorden(arbol):
    if arbol is not None:
        postorden(arbol[1])
        postorden(arbol[2])
        print(arbol[0], end=' ')

# Mostrar árbol visualmente
def mostrar_arbol(arbol, nivel=0):
    if arbol is not None:
        mostrar_arbol(arbol[2], nivel + 1)
        print('    ' * nivel + str(arbol[0]))
        mostrar_arbol(arbol[1], nivel + 1)
```

**Segunda parte:** creación de variable “árbol” para llamar a la función “crear_nodo” , en este caso utilizamos un caso simple de asignación de valores con letras ( “a”, ”b”, “c”, etc).
Se procede a imprimir en consola el árbol generado ( def mostrar_arbol ) y los tres tipos de recorridos.

```python
# Crear un árbol binario de ejemplo

arbol = crear_nodo('A', crear_nodo('B', crear_nodo('D'), crear_nodo('E')), crear_nodo('C', None, crear_nodo('F')))

# Mostrar visualización del árbol
print("Árbol binario:")
mostrar_arbol(arbol)

# Recorridos
print("\nRecorrido inorden:")
inorden(arbol)

print("\nRecorrido preorden:")
preorden(arbol)

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

```

**Tercera parte:** mostramos el resultado por consola al usuario (representación del árbol y sus recorridos )

```
PROBLEMAS   SALIDA   CONSOLA DE DEPURACIÓN   PUERTOS   TERMINAL

PS C:\Users\gabri> & C:/Users/gabri/AppData/Local/Programs/Python/Python311/python.exe
Árbol binario:
        F
    C
A
        E
    B
        D

Recorrido inorden:
D B E A C F
Recorrido preorden:
A B D E C F
Recorrido postorden:
D E B F C A
PS C:\Users\gabri>
```