UNIVERSIDAD TECNOLOGICA NACIONAL

Tecnicatura Universitaria en Programación

Universidad Tecnológica Nacional

Estructura de Datos Avanzados: Árboles binarios de búsqueda en Python

Alumnos – Comisión 12

Cuquejo, Mauro. Email: mauro.cuquejo@gmail.com

Díaz de Quintana, Melisa. Email: mely_ddq@hotmail.com

Trabajo práctico integrador de la asignatura PROGRAMACIÓN I

Docente Titular

Ariel Enferrel

Docente Tutor

Luciano Chiroli

09 de junio de 2025

Índice

Introducción

En el desarrollo de software y la informática en general, las estructuras de datos son fundamentales para organizar, gestionar y almacenar la información de manera eficiente. Aunque estructuras básicas como listas, pilas y colas permiten resolver problemas comunes, existen estructuras de datos avanzadas que ofrecen mayores ventajas en términos de rendimiento, búsqueda y organización jerárquica.

Estas estructuras se utilizan cuando se requiere un manejo más sofisticado de los datos, como sucede en algoritmos de búsqueda, inteligencia artificial, bases de datos, compiladores, sistemas de archivos y más. Entre las más destacadas se encuentran los árboles**,** grafos**,** tablas hash**,** heaps y tries**.**

Dentro de estas estructuras, los árboles representan una forma jerárquica de organizar la información. Un árbol está compuesto por nodos conectados por aristas, donde cada nodo puede tener hijos, y existe uno especial llamado raíz, que es el punto de partida.

Un árbol binario es un tipo particular de árbol donde cada nodo puede tener como máximo dos hijos: uno izquierdo y uno derecho. Esta simplicidad estructural lo hace ideal para múltiples aplicaciones, como la representación de expresiones algebraicas, algoritmos de búsqueda (como los árboles binarios de búsqueda) y estructuras internas de bases de datos.

El estudio y la implementación de árboles binarios en Python no solo nos ayudarán a comprender cómo se estructuran los datos jerárquicos, sino que también sienta las bases para estructuras más complejas.

Objetivos

El propósito principal de este trabajo es comprender la utilidad, eficiencia y optimización de los árboles binarios mediante el desarrollo de un caso práctico en Python. Se abordarán los conceptos teóricos a través de su implementación práctica, con el fin de consolidar los conocimientos adquiridos.

Se analizará cómo se realizan las operaciones fundamentales en un árbol binario, tales como la inserción**,** eliminación y reemplazo de nodos. Asimismo, se explorará el proceso de búsqueda de información dentro de su estructura y los tipos de recorrido que puede realizar el árbol como preorden**,** inorden y postorden**.**

A partir de lo expuesto, el objetivo no es solo comprender su uso funcional, sino también destacar la importancia de los árboles binarios como base fundamental para estructuras más complejas en informática**.**

Marco teórico

Un árbol se puede definir como una estructura jerárquica y en forma no lineal, aplicada sobre una colección de elementos u objetos llamados nodos. (Cairó & Guardati, 2006).

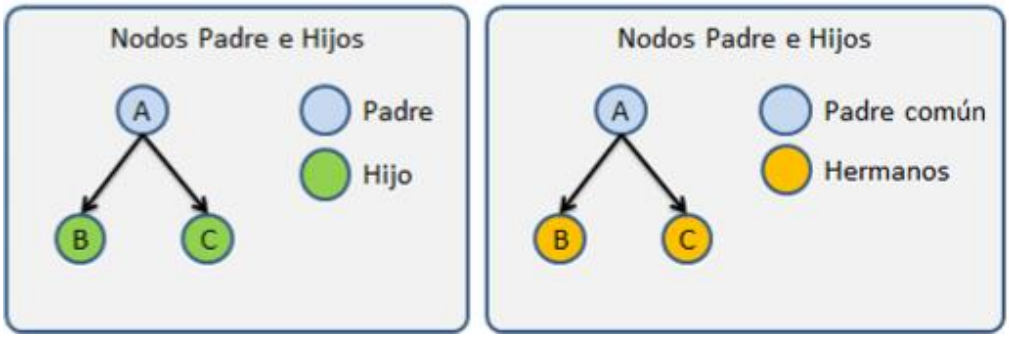
Se les llama estructuras dinámicas, porque las mismas pueden cambiar tanto de forma como de tamaño durante la ejecución del programa. Y estructuras no lineales porque cada elemento del árbol puede tener ninguno, uno, o más de un sucesor.

Los **árboles binarios** son una estructura de datos de tipo jerárquico en la que cada nodo puede tener, como máximo, dos hijos, denominados hijo izquierdo e hijo derecho. A pesar de su aparente simplicidad, los árboles binarios son una estructura sumamente versátil y poderosa, que sirve de base para numerosos algoritmos y aplicaciones en la informática moderna.

Características de la estructura de un árbol binario:

1. En relación a otros nodos:

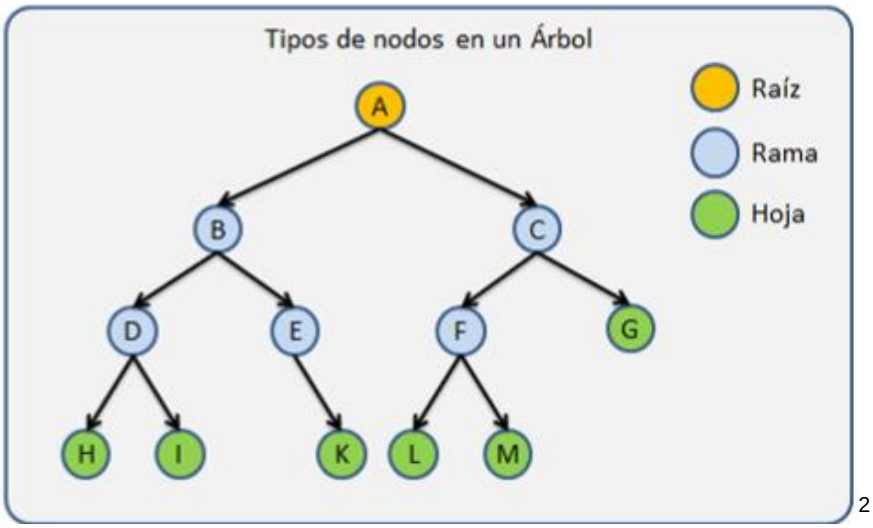
    - Nodos: Se le llama nodo a cada elemento que contiene un Árbol.
    - Nodo Padre: Se utiliza este término para llamar a todos aquellos nodos que tienen al menos un hijo.
    - Nodo Hijo: Los hijos son todos aquellos nodos que tienen un padre.
    - Nodo Hermano: Los nodos hermanos son aquellos nodos que comparten a un mismo padre en común dentro de la estructura.



1

2. <u>En relación a la posición dentro del árbol:</u>

   - <u>Nodo Raíz</u>: Se refiere al primer Nodo de un Árbol. Sólo un nodo del Árbol puede ser la Raíz.
   - <u>Nodo Hoja</u>: Son todos aquellos nodos que no tienen hijos, los cuales siempre se encuentran en los extremos de la estructura.
   - <u>Nodo Rama</u>: Aquellos nodos que no son la raíz y que además tienen al menos un hijo.

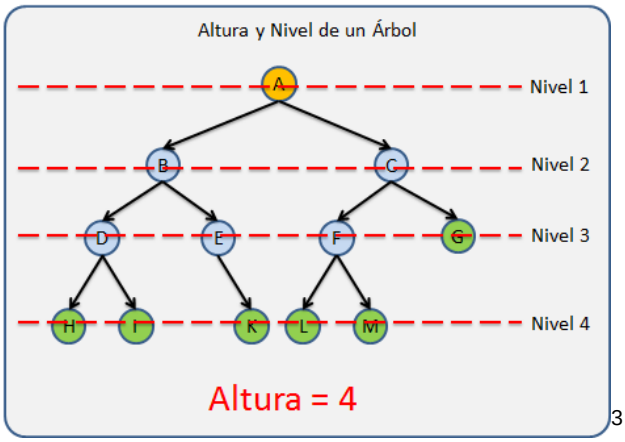


2

Figura 1 Ejemplo de árbol con nodos padres e hijos.
Fuente: Estructura de datos: Árbol. (s.f.).
Figura 2 Ejemplo tipos de nodos en un Árbol.
Fuente: Estructura de datos: Árbol. (s.f.).

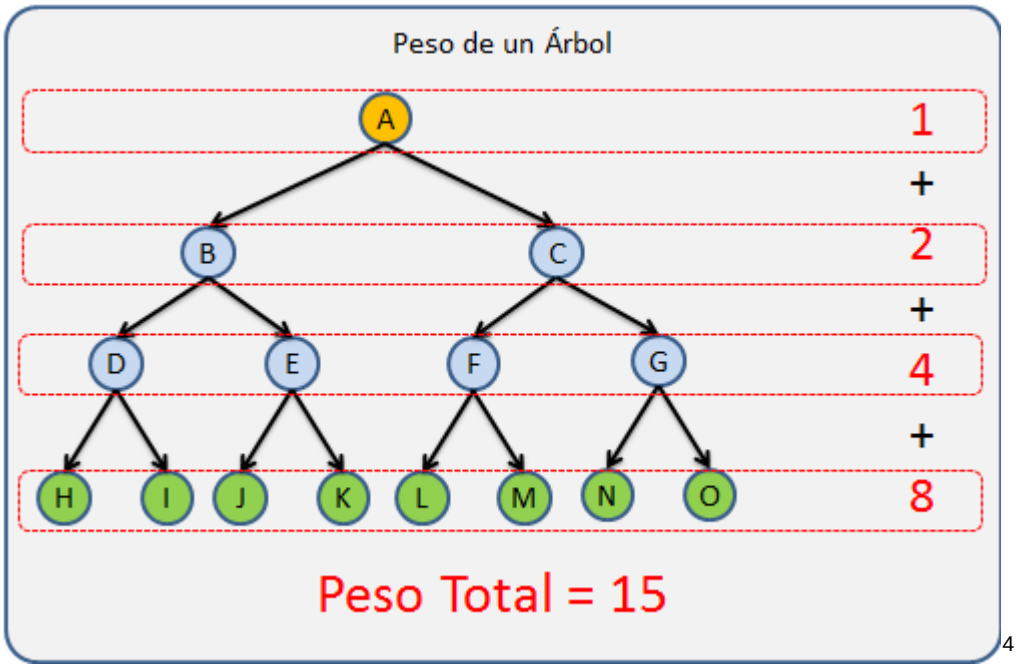
3. En relación al tamaño del árbol:

- Nivel: El nivel de un nodo es su distancia a la raíz.
- Altura: Se le llama altura al número máximo de niveles de un árbol.
- Peso: Es el número de nodos que tiene un árbol.
- Orden: El Orden de un árbol es el número máximo de hijos que puede tener un Nodo. Es una constante que se define antes de crear el árbol.
- Grado: Número de hijos de un nodo y está limitado por el Orden, ya que este indica el número máximo de hijos que puede tener un nodo. El grado de un árbol se define como el máximo grado de todos sus nodos.
- Camino: Secuencia nodos de conectados dentro de un árbol.
- Longitud del camino: Cantidad de nodos que se deben recorrer para llegar desde la raíz a un nodo determinado.
- Sub-Árbol: Conocemos como Sub-Árbol a todo Árbol generado a partir de una sección determinada del Árbol, por lo que podemos decir que un Árbol es un nodo Raíz con N Sub-Árboles.

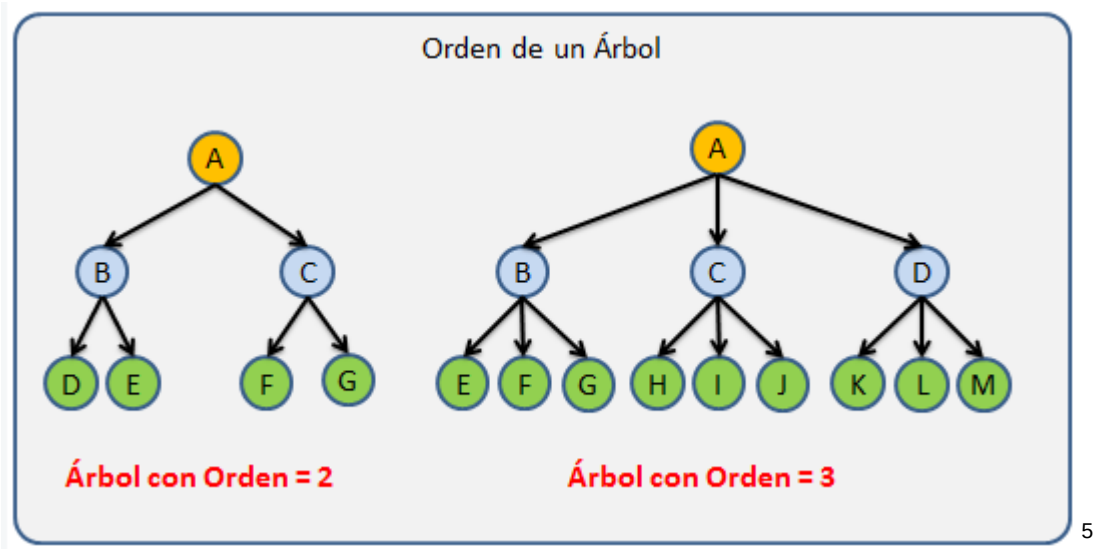


[3]

Figura 3. Ejemplo altura y nivel de un árbol.
Fuente: Estructura de datos: Árbol. (s.f.).



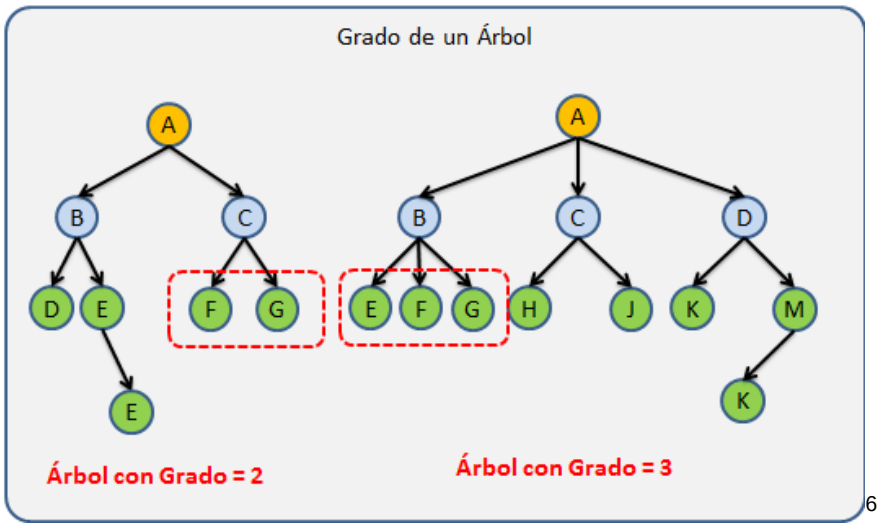
[4]



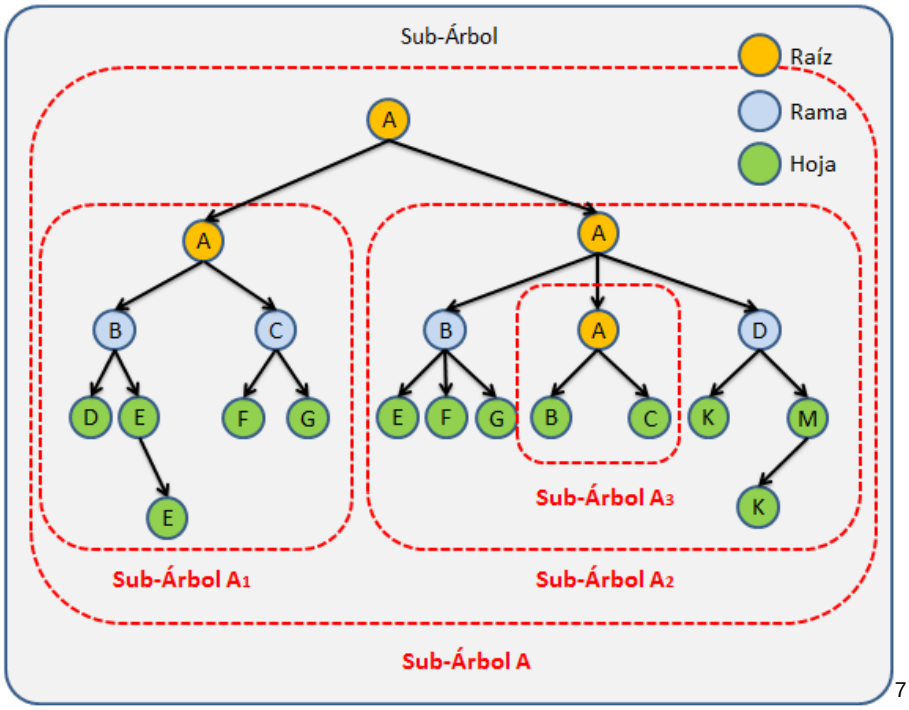
[5]

[4] **Figura 4.** Ejemplo peso de un árbol.
Fuente: Estructura de datos: Árbol. (s.f.).

[5] **Figura 5.** Ejemplo de orden de un árbol.
Fuente: Estructura de datos: Árbol. (s.f.).



[6]



[7]

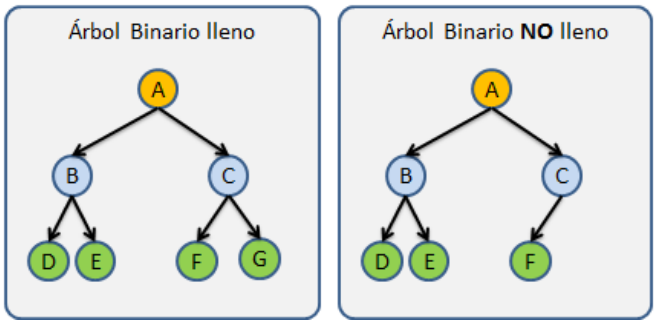
[6] **Figura 6** Ejemplo grados de un árbol.
Fuente: Estructura de datos: Árbol. (s.f.).

[7] **Figura 7** Ejemplo de Sub-árbol.
Fuente: Estructura de datos: Árbol. (s.f.).

Tipos de árboles binarios

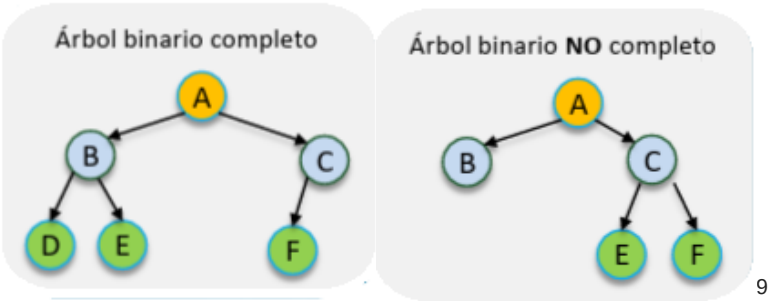
Existen muchos tipos de árboles binarios. Nos centraremos en los principales.

- **Árbol binario lleno**: Un árbol binario lleno es un árbol en el que cada nodo tiene cero o dos hijos. Ningún nodo en un árbol binario lleno tiene solo un hijo.



[8]

- **Árbol binario completo**: Un árbol binario completo es aquel en el que todos los niveles están completamente llenos, excepto posiblemente el último, que se llena de izquierda a derecha. Esta estructura garantiza que el árbol sea lo más compacto posible.

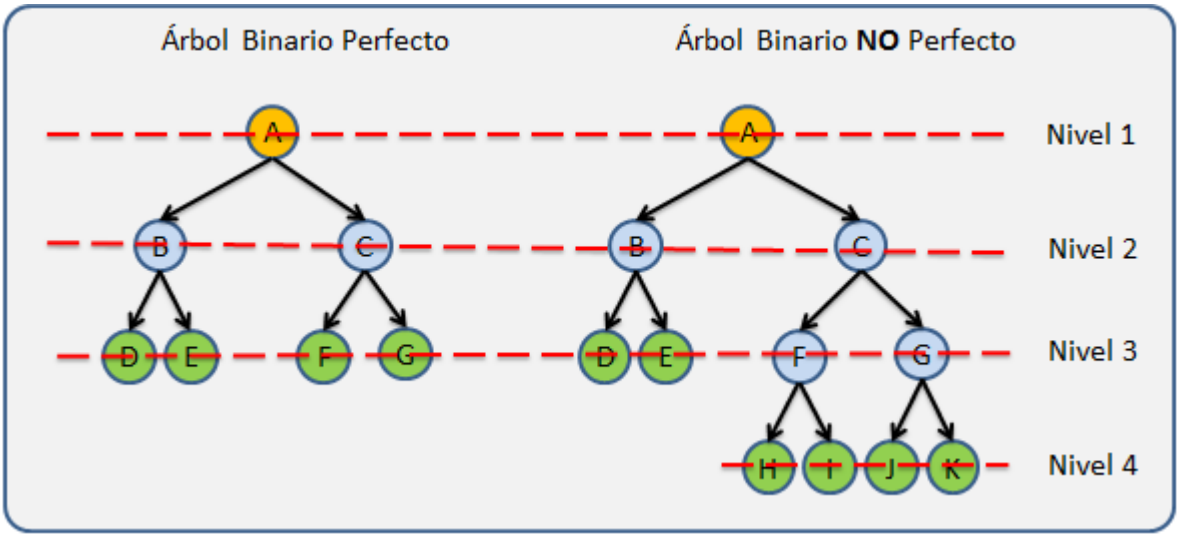


[9]

[8] **Figura 8.** Ejemplo de árbol lleno y no lleno.
Fuente: Estructura de datos: Árbol. (s.f.).

[9] **Figura 9** Ejemplo de árbol completo y no completo.
Fuente: Estructura de datos: Árbol. (s.f.).

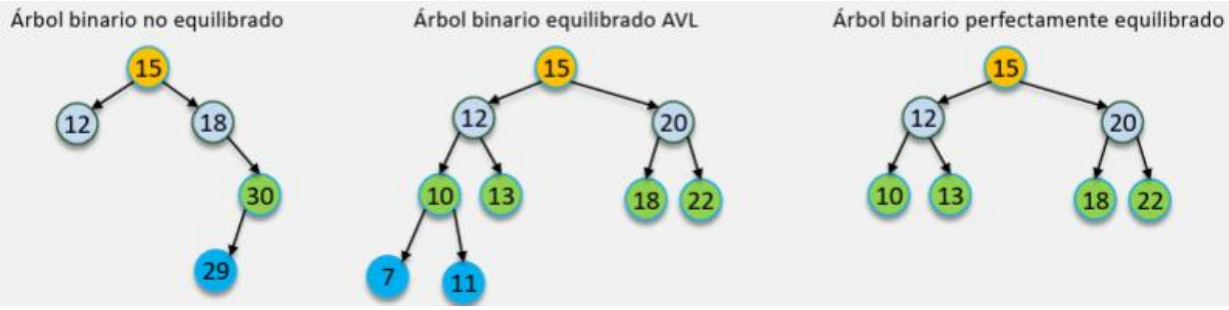
- **Árbol binario perfecto**: Un árbol binario perfecto es un árbol binario completo en el que todos los nodos internos tienen exactamente dos hijos y todos los nodos hoja se encuentran al mismo nivel. Este tipo de árbol binario es completo, lo que garantiza que todos los nodos hoja se encuentren en la profundidad máxima y que cada padre tenga dos hijos.



[10]

- **Árbol binario balanceado**: Un árbol binario balanceado es aquel en el que la altura de los dos subárboles de cualquier nodo difiere como máximo en uno. Este balanceo garantiza que el árbol se mantenga lo más plano posible, lo que promueve operaciones eficientes. Ejemplos de árboles binarios balanceados incluyen los árboles AVL y los árboles Rojo-Negro. El balanceo es fundamental para mantener una complejidad de **O(log n)** en operaciones de búsqueda, inserción y eliminación en grandes conjuntos de datos.

[10] **Figura 10** Ejemplo de árbol perfecto y no perfecto
Fuente: Estructura de datos: Árbol. (s.f.).



[11]

Tras comprender la estructura general y los distintos tipos de árboles binarios (como los árboles completos, llenos y balanceados) nos adentraremos en una de sus variantes más utilizadas en la práctica: el árbol binario de búsqueda (BST, por sus siglas en inglés *Binary Search Tree*). Esta clase particular de árbol binario no solo mantiene la estructura jerárquica ya conocida, sino que incorpora una propiedad esencial: el orden. Esta característica permite realizar operaciones de búsqueda, inserción y eliminación de manera mucho más eficiente que en un árbol binario convencional, convirtiendo al BST en una herramienta clave en el diseño de algoritmos y estructuras de datos optimizadas.

Un árbol binario de búsqueda es un caso especializado de árbol binario que impone un orden estricto para habilitar operaciones eficientes**.** Las propiedades claves son que para todo nodo, los valores en su subárbol izquierdo son menores que su valor, y los valores en su subárbol derecho son mayores que su valor.

Este es el tipo de árbol es el que desarrollaremos y utilizaremos para nuestro caso práctico.

[11] **Figura 11** Ejemplo de árbol balanceado y no balanceado.
Fuente: Estructura de datos: Árbol. (s.f.).

Caso Práctico

En el presente trabajo práctico nos centramos en el estudio y desarrollo de un Árbol Binario de Búsqueda (BST), una estructura de datos fundamental que organiza la información de forma jerárquica siguiendo un orden estricto. Esta propiedad de ordenamiento, donde los valores menores se ubican en el subárbol izquierdo y los mayores en el derecho, permite realizar operaciones de manera eficiente, con una complejidad promedio de O(log n) para las operaciones básicas.

El desarrollo de este caso práctico abarca:

1. La creación del árbol mediante la implementación de nodos enlazados
2. La inserción de valores manteniendo el orden estricto del BST
3. La realización de recorridos (inorden, preorden y postorden)
4. La implementación de operaciones avanzadas como búsqueda y eliminación de nodos
5. Cálculo de propiedades grado y peso del árbol

Este ejercicio no solo nos permite comprender los fundamentos teóricos de los BST, sino también apreciar su importancia práctica en aplicaciones reales. El análisis de su comportamiento nos ayudará a entender tanto sus ventajas en términos de eficiencia como las consideraciones necesarias para mantener su equilibrio. En la parte de anexos se compartirá un link con una breve explicación del código aplicado y su ejecución. Se muestran las capturas:

1. La creación del árbol mediante la implementación de nodos enlazados

```
# ------------------------------------------------------------------------
# 1) CREAR NODO
# ------------------------------------------------------------------------
def crear_nodo(dato=None, nodo_izq=None, nodo_der=None):
    return {
        "dato": dato,
        "nodo_izq": nodo_izq,
        "nodo_der": nodo_der
    }
```

2. La inserción de valores manteniendo el orden estricto del BST

```
# ------------------------------------------------------------------------
# 2) AGREGAR UN NODO AL ARBOL BINARIO
# ------------------------------------------------------------------------
def agregar_nodo(nodo, nodo_nuevo):
    if nodo["dato"] is None:
        nodo["dato"] = nodo_nuevo["dato"]
        nodo["nodo_izq"] = nodo_nuevo["nodo_izq"]
        nodo["nodo_der"] = nodo_nuevo["nodo_der"]
    else:
        if nodo_nuevo["dato"] < nodo["dato"]:
            if nodo["nodo_izq"] is None:
                nodo["nodo_izq"] = nodo_nuevo
            else:
                agregar_nodo(nodo["nodo_izq"], nodo_nuevo)
        elif nodo_nuevo["dato"] > nodo["dato"]:
            if nodo["nodo_der"] is None:
                nodo["nodo_der"] = nodo_nuevo
            else:
                agregar_nodo(nodo["nodo_der"], nodo_nuevo)
        else:
            print(f"Se descarta dato duplicado {nodo_nuevo['dato']}")
```

3. La realización de recorridos (inorden, preorden y postorden)

```
# ------------------------------------------------------
# 3.1) IN ORDER
# ------------------------------------------------------
def leer_arbol_in_orden(nodo):
    if (nodo is None):
        return []
    izq = leer_arbol_pre_orden(nodo["nodo_izq"])
    der = leer_arbol_pre_orden(nodo["nodo_der"])
    return izq + [nodo["dato"]] + der


# ------------------------------------------------------
# 3.2) PRE ORDER
# ------------------------------------------------------
def leer_arbol_pre_orden(nodo):
    if (nodo is None):
        return []
    izq = leer_arbol_pre_orden(nodo["nodo_izq"])
    der = leer_arbol_pre_orden(nodo["nodo_der"])
    return [nodo["dato"]] + izq + der


# ------------------------------------------------------
# 3.3) POST ORDER
# ------------------------------------------------------
def leer_arbol_post_orden(nodo):
    if (nodo is None):
        return []
    izq = leer_arbol_pre_orden(nodo["nodo_izq"])
    der = leer_arbol_pre_orden(nodo["nodo_der"])
    return izq + der + [nodo["dato"]]
```

4. La implementación de operaciones avanzadas como búsqueda y eliminación de nodos.

```python
# ------------------------------------------------------
# 4) BUSCAR UN NODO EN EL ARBOL
# ------------------------------------------------------
def buscar_nodo(nodo, dato):
    if nodo["dato"] is None:
        return None
    else:
        if dato < nodo["dato"]:
            return buscar_nodo(nodo["nodo_izq"], dato)
        elif dato > nodo["dato"]:
            return buscar_nodo(nodo["nodo_der"], dato)
        else:
            return nodo
```

Al momento de realizar una eliminación de un nodo con dos hijos, tenemos dos alternativas:

Sucesor inorden: el menor valor del subárbol derecho (lo más común).

Predecesor inorden: el mayor valor del subárbol izquierdo.

Decidimos optar por implementar la primera opción

```python
# ------------------------------------------------------------------------
# 5) ELIMINAR UN NODO EN EL ARBOL
# ------------------------------------------------------------------------
def eliminar_nodo(nodo, dato):
    if nodo is None:
        return None

    if dato < nodo["dato"]:
        nodo["nodo_izq"] = eliminar_nodo(nodo["nodo_izq"], dato)
    elif dato > nodo["dato"]:
        nodo["nodo_der"] = eliminar_nodo(nodo["nodo_der"], dato)
    else:
        # Caso 1: sin hijos
        if nodo["nodo_izq"] is None and nodo["nodo_der"] is None:
            return None
        # Caso 2: un hijo
        elif nodo["nodo_izq"] is None:
            return nodo["nodo_der"]
        elif nodo["nodo_der"] is None:
            return nodo["nodo_izq"]
        # Caso 3: dos hijos
        sucesor = nodo_minimo(nodo["nodo_der"])
        nodo["dato"] = sucesor["dato"]
        nodo["nodo_der"] = eliminar_nodo(nodo["nodo_der"], sucesor["dato"])

    return nodo


def nodo_minimo(nodo):
    actual = nodo
    while actual["nodo_izq"] is not None:
        actual = actual["nodo_izq"]
    return actual
```

5. El cálculo de propiedades, grado y peso del árbol

```
# ------------------------------------------------------------
# 6) CALCULAR GRADO ARBOL
# ------------------------------------------------------------
def calcular_grado_nodo(nodo):
    grado_nodo = 0
    if nodo["nodo_izq"] is not None:
        grado_nodo += 1
    if nodo["nodo_der"] is not None:
        grado_nodo += 1
    return grado_nodo


def calcular_grado_arbol_post_orden(nodo, grado_max=0):
    if (nodo is None):
        return 0
    calcular_grado_arbol_post_orden(nodo["nodo_izq"], grado_max)
    calcular_grado_arbol_post_orden(nodo["nodo_der"], grado_max)
    return max(grado_max, calcular_grado_nodo(nodo))


# ------------------------------------------------------------
# 6) CALCULAR PESO ARBOL
# ------------------------------------------------------------
def calcular_peso_post_orden(nodo):
    if (nodo is None):
        return 0
    izq = calcular_peso_post_orden(nodo["nodo_izq"])
    der = calcular_peso_post_orden(nodo["nodo_der"])
    return izq + der + 1
```

Conclusión

Los árboles binarios son una estructura de datos fundamental en informática, ya que ofrecen una forma versátil y eficiente de almacenar y gestionar datos jerárquicos. A lo largo de este trabajo, hemos profundizado en la estructura e implementación de los árboles binarios en Python, abarcando conceptos esenciales, métodos de recorrido y algunas funcionalidades.

Al comprender la estructura básica de los árboles binarios, incluyendo nodos, aristas, raíces y hojas, los desarrolladores pueden apreciar la simplicidad y la potencia de esta estructura de datos. La diferenciación entre árboles binarios generales y árboles binarios de búsqueda (BST) permite comprender mejor cómo las propiedades específicas pueden optimizar operaciones como la búsqueda, la inserción y la eliminación, que en un BST balanceado tienen una complejidad promedio de $O(\log n)$, a diferencia de un árbol binario general, será de $O(n)$.

Dominar los árboles binarios es un paso esencial para cualquier desarrollador que busque optimizar el manejo de datos y, al mismo tiempo, resolver problemas complejos con elegancia algorítmica.

Referencias

- Estructura de datos: Árbol. (s.f.). Recuperado el 6 de junio de 2025, de https://gftwiwhpzc5vpftgntujea.on.drv.tw/clase%20de%20%C3%A1rbol%20con%20exelearning/arbol/estructura_de_datos_rbol.html
- Cairó, O. y Guardati, S. (2002). Estructuras de Datos, 2da. Edición. McGraw-Hill.
- Universidad Veracruzana. (2021). *Clase 8: Árboles* [Archivo PDF]. Recuperado de https://www.uv.mx/personal/ermeneses/files/2021/08/Clase8-Arboles.pdf
- Universidad Tecnológica Nacional, Asignatura programación I (2025). *Apunte teórico sobre árboles* [archivo en línea]. Plataforma tup.sied.utn.edu.ar/. https://docs.google.com/document/d/10k16oL15EeyOaq92aoi4qwK3t_22X29-FSV2iV-8N1U/edit?tab=t.0#heading=h.d6ixjk6fxwp5

Anexos

- Link repositorio GitHub: https://github.com/mauro-cuquejo/UTN-TUPaD-P1/tree/main/12%20Datos%20Avanzados
- Link video de YouTube: https://youtu.be/P-RrdD-gPvU