

**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
2. Marco teórico
3. Caso práctico
4. Metodología utilizada
5. Resultados obtenidos
6. Conclusiones
7. Bibliografía
8. Anexos

## 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.

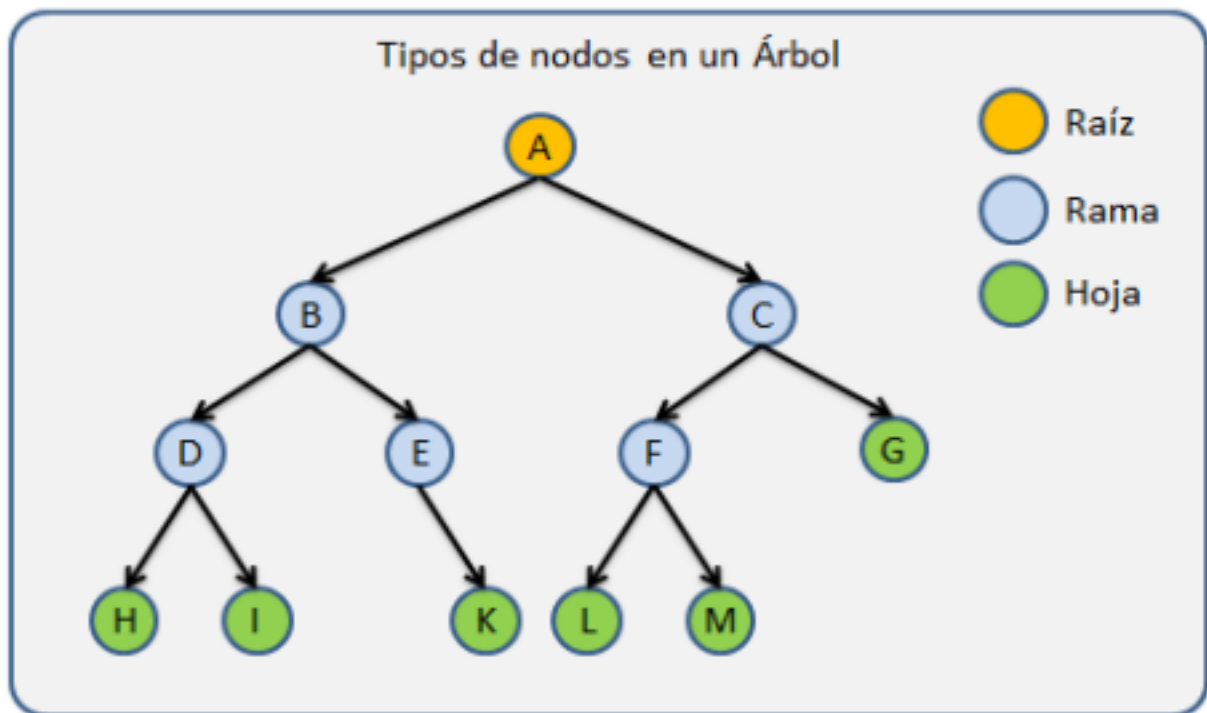
A continuación, explicaremos los conceptos básicos de los árboles, sus tipos principales y las ventajas que ofrecen frente a otras estructuras de datos.

## 2. Marco teórico

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.

### Componentes

- **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:** Nodo que tiene uno o más nodos hijos conectados directamente a él.
- **Hoja:** Nodo sin hijos.
- **Subárbol:** Porción de un árbol, compuesto por cualquier nodo del mismo junto con todos sus descendientes.
- **Arista:** Llamamos así a las conexiones entre nodos ( ej:  $A \rightarrow B$ ).



### Propiedades Clave

- **Altura:** Longitud del camino más largo desde la raíz hasta 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.

```

      A      (Profundidad: 0 / Altura: 2)
    /  \
   B    C   (Profundidad: 1 / Altura de B: 1)
  /  \  \
 D   E  F   (Profundidad: 2 / Altura: 0)

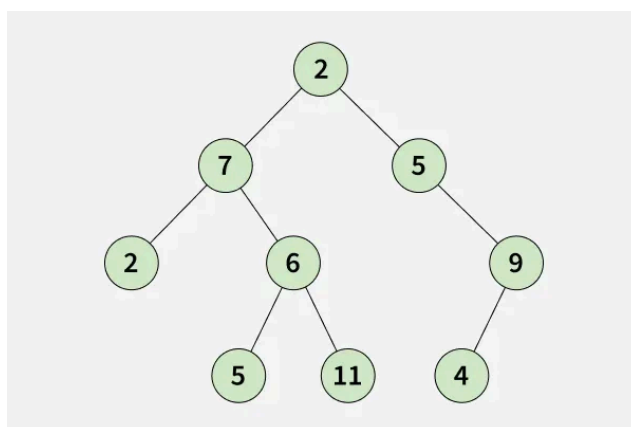
```

(Grado de A: 2)  
(Grado de B: 2, Grado de C: 1)

## 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.



### 2. 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.

### 3. Árbol Binario 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). En este tipo de árboles, la diferencia entre la altura del subárbol izquierdo y derecho, de cualquier nodo debe ser mínima (como mucho 1, o , -1). Esto mantiene el árbol "parejo", ayudando a que buscar, agregar o borrar datos sea más rápido.

#### **4. 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.

#### **5. Árbol Completo**

Todos los niveles están completamente llenos, excepto quizás el último.

En el último nivel, los nodos están lo más a la izquierda posible.

#### **6. Árbol Lleno (full)**

Tipo de árbol donde todos los nodos tienen 0 o 2 hijos (nunca 1 solo).

#### **7. Árbol perfecto**

Es completo y lleno a la vez.

Todos los niveles están completos y todos los nodos tienen 2 hijos, salvo las hojas.

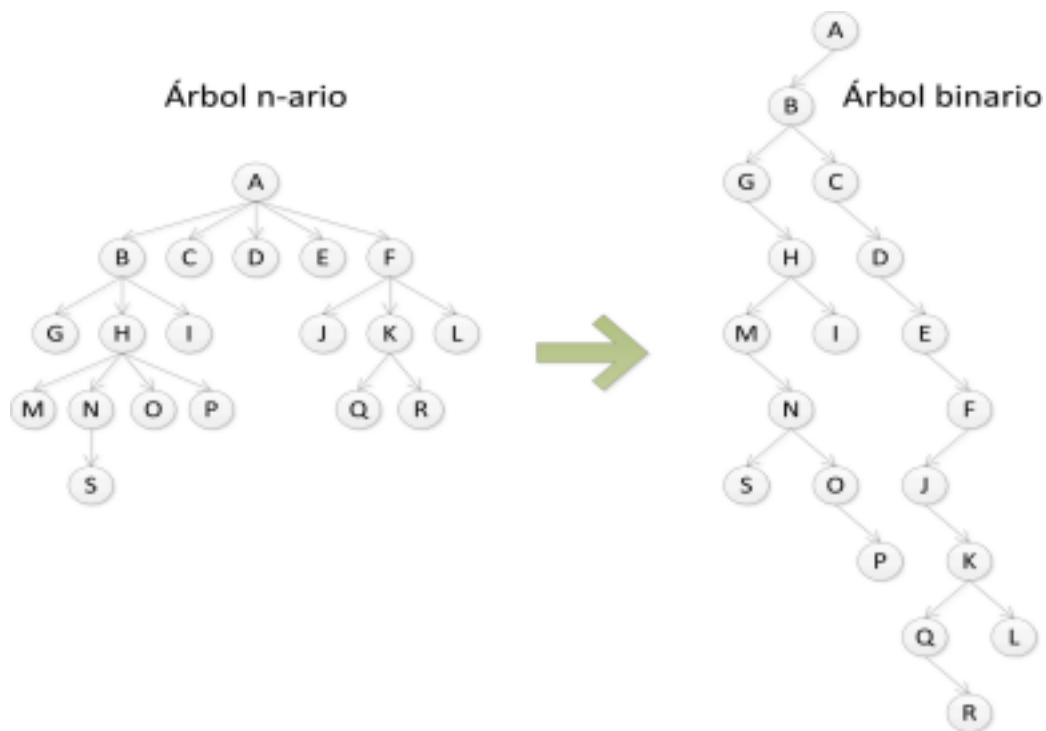
#### **Árboles especializados**

- **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 círculos y líneas
- 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

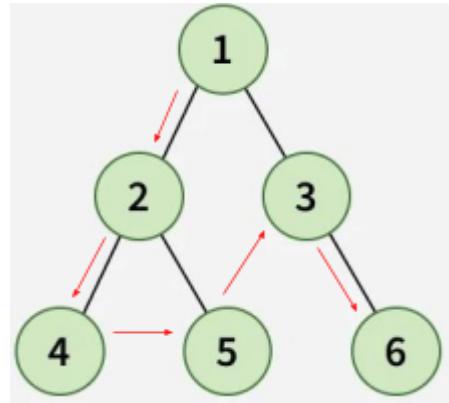
- Inserción/Búsqueda/Eliminación: según el tipo de caso que sea el árbol.  
(balanceado o desbalanceado) puede ser rápido o lento respectivamente.

## Recorridos

### - Preorden:

Raíz → Izquierda → Derecha.

Ejemplo:  $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 6$ .

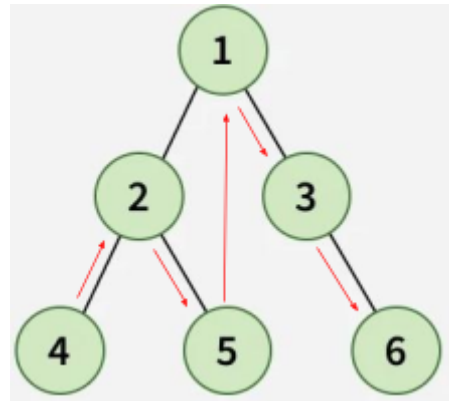


### - Inorden:

Izquierda → Raíz → Derecha.

En ABB: Ordena los datos de menor a mayor.

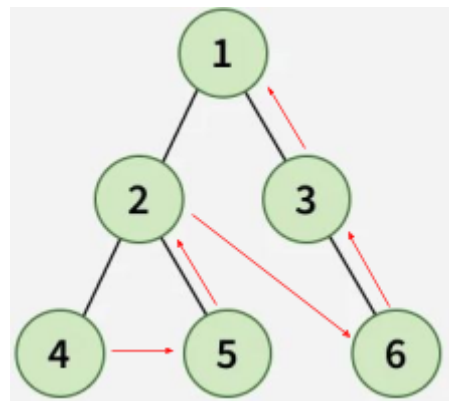
Ejemplo:  $4 \rightarrow 2 \rightarrow 5 \rightarrow 1 \rightarrow 3 \rightarrow 6$ .



### - Postorden:

Izquierda → Derecha → Raíz.

Ejemplo:  $4 \rightarrow 5 \rightarrow 2 \rightarrow 6 \rightarrow 3 \rightarrow 1$ .





---

## Ventajas y Desventajas

- Ventajas

- **Búsqueda eficiente:** especialmente los árboles binarios de búsqueda balanceados, permiten reducir el espacio de búsqueda a la mitad en cada paso, logrando búsquedas en tiempo logarítmico.
- **Modelado de jerarquías naturales:** reflejan de forma clara y estructurada relaciones de tipo padre-hijo, permitiendo representar niveles de dependencia o clasificación de manera ordenada y fácil de recorrer.
- **Flexibilidad en aplicaciones reales:** pueden adaptarse a diferentes estructuras de datos (como listas, textos, archivos o expresiones), permiten insertar y eliminar elementos dinámicamente sin necesidad de reorganizar toda la estructura, y facilitan recorridos personalizados según las necesidades del problema.

- Desventajas

- **Complejidad en balanceo (AVL, etc.):** Si un árbol no está balanceado, puede generar tiempos de búsqueda desiguales. Esto puede ser un problema en aplicaciones donde la velocidad es crítica.
- **Mayor consumo de memoria que listas:** Los árboles pueden requerir una cantidad significativa de memoria para almacenarse, especialmente si son muy grandes. Esto puede ser un problema para aplicaciones que tienen recursos de memoria limitados.
- **Operaciones de eliminación complejas:** Si un problema requiere únicamente búsqueda, inserción y eliminación, y no otras operaciones como recorrido de datos ordenados, mínimo o máximo inmediato (floor y ceiling), los diccionarios siempre superan en rendimiento a los árboles binarios de búsqueda auto-balanceados.

### 3. Caso práctico

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

En este caso práctico se implementa un Árbol Binario de Búsqueda (ABB) utilizando listas anidadas en Python. Cada nodo del árbol está representado como una lista de tres elementos: el valor, el subárbol izquierdo y el subárbol derecho.

El programa permite insertar valores numéricos en el árbol, buscar elementos y visualizar la estructura del árbol de forma jerárquica. También se incluyen los tres recorridos clásicos del árbol (inorden, preorden y postorden), lo que permite observar el comportamiento del ABB según distintas formas de recorrido.

Todo esto se gestiona mediante un menú interactivo que guía al usuario para realizar las operaciones básicas sobre el árbol binario.

#### Inicio del script

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

# Insertar valor en ABB
def insertar(arbol, valor):
    if arbol is None:
        return crear_nodo(valor)
    if valor < arbol[0]:
        arbol[1] = insertar(arbol[1], valor)
    else:
        arbol[2] = insertar(arbol[2], valor)
    return arbol

# Buscar valor en el árbol (retorna True o False)
def buscar(arbol, valor):
    if arbol is None:
        return False
    if valor == arbol[0]:
        return True
    elif valor < arbol[0]:
        return buscar(arbol[1], valor)
    else:
        return buscar(arbol[2], valor)

# 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, prefijo='', es_izquierdo=True):
    if arbol is not None:
        mostrar_arbol(arbol[2], prefijo + ('| ' if es_izquierdo else ' '), False)

        print(prefijo + ('└─ ' if es_izquierdo else '─ ') +
              str(arbol[0]))

        mostrar_arbol(arbol[1], prefijo + (' ' if es_izquierdo else '| '), True)

# Jerarquía en el árbol
def imprimir_jerarquia_nivel(arbol):
    from collections import deque
    if not arbol:
        return
    cola = deque()
    cola.append(arbol)
    while cola:
        nivel = len(cola)
        linea = []
        for _ in range(nivel):
            nodo = cola.popleft()
            if nodo:
                linea.append(str(nodo[0]))
                cola.append(nodo[1])
                cola.append(nodo[2])
        if linea:
            print(" ".join(linea))

# Menú interactivo
def menu():
```

```
print("\nÁrbol Binario de Búsqueda (ABB) usando listas")
print("1. Insertar un valor")
print("2. Mostrar recorrido Inorden")
print("3. Mostrar recorrido Preorden")
print("4. Mostrar recorrido Postorden")
print("5. Buscar un valor")
print("6. Salir")

def main():
    arbol = None # árbol vacío
    while True:
        menu()
        opcion = input("Ingrese una opción: ")

        if opcion == '1':
            try:
                val = int(input("Ingrese un número para insertar: "))
                arbol = insertar(arbol, val)
                print(f"Valor {val} insertado.")
            except ValueError:
                print("Por favor, ingrese un número entero válido.")

        elif opcion == '2':
            print("Árbol binario:")
            mostrar_arbol(arbol)
            print("Recorrido Inorden: ", end='')
            inorden(arbol)
            print()

        elif opcion == '3':
            print("Árbol binario:")
            mostrar_arbol(arbol)
            print("Recorrido Preorden: ", end='')
            preorden(arbol)
            print()

        elif opcion == '4':
            print("Árbol binario:")
            mostrar_arbol(arbol)
            print("Recorrido Postorden: ", end='')
            postorden(arbol)
            print()

        elif opcion == '5':
            try:
                val = int(input("Ingrese un número para buscar: "))
                encontrado = buscar(arbol, val)
                if encontrado:
                    print(f"El valor {val} está en el árbol.")
                else:
                    print(f"El valor {val} NO está en el árbol.")
```

```
except ValueError:
    print("Por favor, ingrese un número entero válido.")

elif opcion == '6':
    print(";Hasta luego!")
    break
else:
    print("Opción no válida, intente de nuevo.")

if __name__ == "__main__":
    main()
```

#### 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 código 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.
- <https://www.geeksforgeeks.org/types-of-trees-in-data-structures/>
- <https://runestone.academy/ns/books/published/pythonds/Trees/ExamplesofTrees.html>

## 8. Anexos

Enlace a vídeo explicativo :

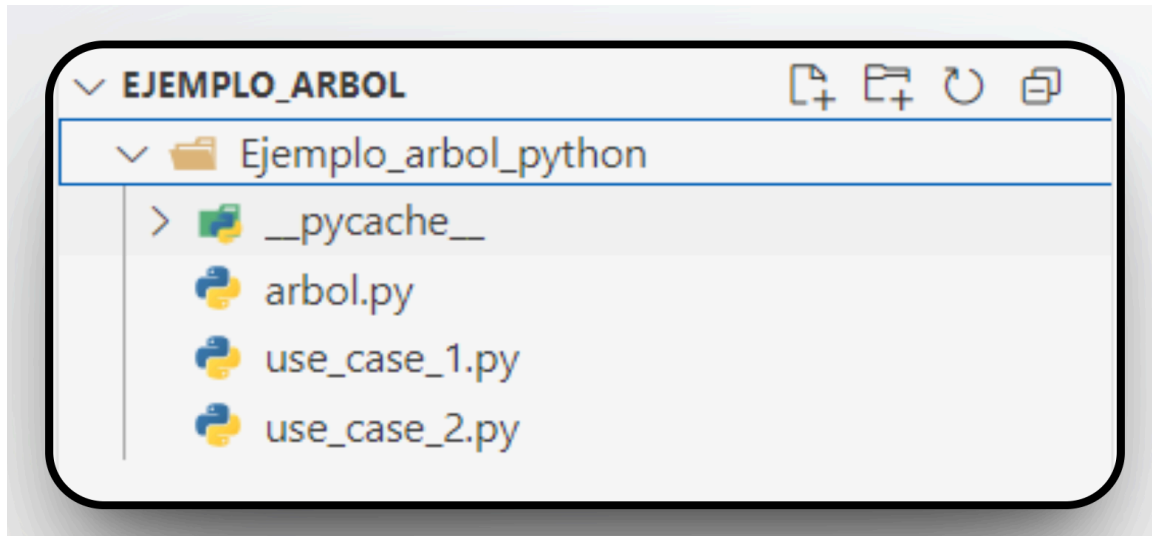
<https://youtu.be/wA4dhWkQnwU>

Enlace a repositorio en Github:

[https://github.com/Gabriel071185/trabajo\\_integrador\\_programacion1\\_utn.git](https://github.com/Gabriel071185/trabajo_integrador_programacion1_utn.git)

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

**Primera parte:** Generamos un código modularizado en tres diferentes archivos , el primero de ellos contiene las funciones dedicadas a la creación de los nodos y sus respectivos recorridos incluida una interacción con el usuario para generar un arbol , ver los distintos tipos de recorridos , inserción y búsquedas de los nodos. Luego completamos con dos módulos que implementan dos casos de uso, uno con letras del alfabeto y otro simulando un organigrama jerárquico de una empresa (Puestos).



**Segunda 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. Se agregan las funciones de búsqueda e inserción para interacción con un usuario a través de la consola.

```
1 # Crear nodo del árbol
2 def crear_nodo(valor, izquierdo=None, derecho=None):
3     return [valor, izquierdo, derecho]
4
5 # Insertar valor en ABB
6 def insertar(arbol, valor):
7     if arbol is None:
8         return crear_nodo(valor)
9     if valor < arbol[0]:
10         arbol[1] = insertar(arbol[1], valor)
11     else:
12         arbol[2] = insertar(arbol[2], valor)
13     return arbol
14
15 # Buscar valor en el árbol (retorna True o False)
16 def buscar(arbol, valor):
17     if arbol is None:
18         return False
19     if valor == arbol[0]:
20         return True
21     elif valor < arbol[0]:
22         return buscar(arbol[1], valor)
23     else:
24         return buscar(arbol[2], valor)
25
26 # Recorridos
27 def inorden(arbol):
28     if arbol is not None:
29         inorden(arbol[1])
30         print(arbol[0], end=' ')
31         inorden(arbol[2])
32
33 def preorden(arbol):
34     if arbol is not None:
35         print(arbol[0], end=' ')
36         preorden(arbol[1])
37         preorden(arbol[2])
38
39 def postorden(arbol):
40     if arbol is not None:
41         postorden(arbol[1])
42         postorden(arbol[2])
43         print(arbol[0], end=' ')
```

**Tercera parte:** Continuación del primer módulo, en donde utilizamos dos funciones



para visualizar el árbol generado.

En primer lugar, la función “mostrar\_arbol” imprime en consola la jerarquía del árbol en distribución horizontal, lo que significa que los niveles se representan como tabulaciones respecto del margen izquierdo. De esta manera, aquellos nodos que estén a la misma distancia de este margen pertenecen al mismo nivel del árbol.

En segundo lugar, la función imprimir\_jerarquia\_nivel imprime en cada línea de la consola los nodos que se encuentran en el mismo nivel dentro del árbol.


```

1  # Mostrar árbol visualmente
2  def mostrar_arbol(arbol, prefijo='', es_izquierdo=True):
3      if arbol is not None:
4          mostrar_arbol(arbol[2], prefijo + ('| ' if es_izquierdo else ' '),
5                          False)
6          print(prefijo + ('└─ ' if es_izquierdo else '┌─ ') + str(arbol[0]))
7          mostrar_arbol(arbol[1], prefijo + (' ' if es_izquierdo else '| '),
8                          True)
9
10 # Jerarquia en el árbol
11 def imprimir_jerarquia_nivel(arbol):
12     from collections import deque
13     if not arbol:
14         return
15     cola = deque()
16     cola.append(arbol)
17     while cola:
18         nivel = len(cola)
19         linea = []
20         for _ in range(nivel):
21             nodo = cola.popleft()
22             if nodo:
23                 linea.append(str(nodo[0]))
24                 cola.append(nodo[1])
25                 cola.append(nodo[2])
26         if linea:
27             print(" ".join(linea))

```

**Cuarta parte:** La función menú() muestra en pantalla un menú de opciones para que el usuario interactúe con el programa del árbol binario. Imprime una lista numerada con las acciones disponibles: insertar un valor, mostrar los recorridos (inorden, preorden, postorden), buscar un valor y salir del programa. No recibe ni retorna ningún valor; solo imprime el menú para guiar al

usuario.



```
1 # Menú interactivo
2 def menu():
3     print("\nÁrbol Binario de Búsqueda (ABB) usando listas")
4     print("1. Insertar un valor")
5     print("2. Mostrar recorrido Inorden")
6     print("3. Mostrar recorrido Preorden")
7     print("4. Mostrar recorrido Postorden")
8     print("5. Buscar un valor")
9     print("6. Salir")
```

**Quinta parte:** La función `main()` es el ciclo principal del programa del primer módulo. Inicializa el árbol vacío y muestra repetidamente el menú, permitiendo al usuario elegir opciones para insertar, mostrar recorridos, buscar valores o salir. Según la opción elegida, ejecuta la acción correspondiente hasta que el usuario decide salir.

```

1  def main():
2      arbol = None # árbol vacío
3      while True:
4          menu()
5          opcion = input("Ingrese una opción: ")
6
7          if opcion == '1':
8              try:
9                  val = int(input("Ingrese un número para insertar: "))
10                 arbol = insertar(arbol, val)
11                 print(f"Valor {val} insertado.")
12             except ValueError:
13                 print("Por favor, ingrese un número entero válido.")
14
15             elif opcion == '2':
16                 print("Árbol binario:")
17                 mostrar_arbol(arbol)
18                 print("Recorrido Inorden: ", end='')
19                 inorden(arbol)
20                 print()
21
22             elif opcion == '3':
23                 print("Árbol binario:")
24                 mostrar_arbol(arbol)
25                 print("Recorrido Preorden: ", end='')
26                 preorden(arbol)
27                 print()
28
29             elif opcion == '4':
30                 print("Árbol binario:")
31                 mostrar_arbol(arbol)
32                 print("Recorrido Postorden: ", end='')
33                 postorden(arbol)
34                 print()
35
36             elif opcion == '5':
37                 try:
38                     val = int(input("Ingrese un número para buscar: "))
39                     encontrado = buscar(arbol, val)
40                     if encontrado:
41                         print(f"El valor {val} está en el árbol.")
42                     else:
43                         print(f"El valor {val} NO está en el árbol.")
44                 except ValueError:
45                     print("Por favor, ingrese un número entero válido.")
46
47             elif opcion == '6':
48                 print("¡Hasta luego!")
49                 break
50             else:
51                 print("Opción no válida, intente de nuevo.")
52
53 if __name__ == "__main__":
54     main()

```

**Sexta parte:** Resultados por consola simulando el caso del ejemplo anterior simulando ingresos numéricos en forma jerárquica con valores ingresados por consola a través del menú.

```
PROBLEMAS    SALIDA    CONSOLA DE DEPURACIÓN    PUERTOS    TERMINAL

PS C:\Users\gabri\Downloads\Ejemplo_arbol> & C:/Users/gabri/AppData/Local/Programs/Python/Python39-64/Python.exe c:/Users/gabri/Downloads/Ejemplo_arbol/Ejemplo_arbol_python/arbol.py

Árbol Binario de Búsqueda (ABB) usando listas
1. Insertar un valor
2. Mostrar recorrido Inorden
3. Mostrar recorrido Preorden
4. Mostrar recorrido Postorden
5. Buscar un valor
6. Salir
Ingrese una opción: 1
Ingrese un número para insertar: 100
Valor 100 insertado.
```

Salida luego de varios ingresos, mostrando el arbol generado y el recorrido Preorden.

```

Árbol Binario de Búsqueda (ABB) usando listas
1. Insertar un valor
2. Mostrar recorrido Inorden
3. Mostrar recorrido Preorden
4. Mostrar recorrido Postorden
5. Buscar un valor
6. Salir
Ingrese una opción: 3
Árbol binario:
      200
     /  \
    125   125
   /  \
  100  70
 /  \  /  \
50  60 45  1
Recorrido Preorden: 100 50 45 1 70 60 125 200 125

```