

# [ Segunda entrega ]

## Trabajo Práctico Integrador: Programación I

### Datos Generales

- Título del trabajo: Estructuras de datos avanzadas: Árboles
- Alumnos: Paschetta Gastón      Mail: [gpaschetta17@gmail.com](mailto:gpaschetta17@gmail.com)  
Maza Bruno                      Mail: [bruno.maza@tupad.utn.edu.ar](mailto:bruno.maza@tupad.utn.edu.ar)
- Carrera: Tecnicatura Universitaria en Programación
- 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 representan un concepto fundamental en el estudio de las estructuras de datos. Se organizan jerárquicamente y permiten modelar relaciones padre-hijo, facilitando la resolución de ciertos problemas informáticos, almacenando y moviendo datos de forma eficiente y ordenada. En este trabajo práctico, se explora la estructura de datos de tipo árbol desde una perspectiva teórica y se implementa un caso práctico en Python que refleja su funcionamiento.

## 2. Marco Teórico

Un árbol es una estructura de datos no lineal compuesta por nodos conectados mediante aristas. Se caracteriza por:

Un nodo raíz (root), que es el punto de partida.

Cada nodo puede tener cero o más hijos (children).

Un nodo sin hijos se denomina hoja (leaf).

No existen ciclos: no se puede volver al mismo nodo siguiendo las conexiones.

Tipos de árboles comunes:

Árbol binario: cada nodo tiene como máximo dos hijos.

Árbol binario de búsqueda (BST): los nodos están organizados de tal manera que los valores menores están en el subárbol izquierdo y los mayores en el derecho.

Árbol balanceado: mantiene el equilibrio de altura entre subárboles para optimizar el rendimiento.

Operaciones comunes:

- Inserción de nodos
- Borrado de nodos

Recorridos:

- Preorden: Raíz, rama izquierda, rama derecha
- Inorden: Izquierda, raíz, derecha
- Postorden: Izquierda, derecha, raíz.

Búsqueda de elementos

Los árboles se utilizan en sistemas de archivos, índices de bases de datos, inteligencia artificial, y más.

### 3. Caso Práctico

Exploramos una forma distinta a las propuestas de los TP modelo para evitar semejanzas en la solución y no caer en copiar en vez de investigar. A continuación, este código en Python implementa un árbol usando diccionarios con funciones para insertar nodos y recorrerlo en orden:

**def crear\_nodo(value):**

```
    return {'valor': value, 'izq': None, 'der': None}
```

**def insertar(nodo, value):**

```
    if nodo is None:
```

```
        return crear_nodo(value)
```

```
    if value < nodo['valor']:
```

```
        nodo['izq'] = insertar(nodo['izq'], value)
```

```
    else:
```

```
        nodo['der'] = insertar(nodo['der'], value)
```

```
    return nodo
```

**def inorden(nodo):**

```
    if nodo is not None:
```

```
        inorden(nodo['izq'])
```

```
        print(nodo['valor'], end=' ')
```

```
        inorden(nodo['der'])
```

**def preorden(nodo):**

```
    if nodo is not None:
```

```
print(nodo['valor'], end=' ')
preorden(nodo['izq'])
preorden(nodo['der'])
```

```
def postorden(nodo):
```

```
    if nodo is not None:
        postorden(nodo['izq'])
        postorden(nodo['der'])
        print(nodo['valor'], end=' ')
```

## **# MAIN PROGRAM**

```
arbol = None
```

```
values = []
```

```
while True:
```

```
    entry = input("Ingrese un valor para insertar al árbol (o 'Q' para terminar): ")
```

```
    # Si el usuario ingresa -1, se termina la entrada de datos
```

```
    if entry.lower() == 'q':
```

```
        break
```

```
    try:
```

```
        value = int(entry)
```

```
        values.append(value)
```

```
    except ValueError:
```

```
        print("Por favor, ingrese un número válido.")
```

```
for element in values:
```

```
    arbol = insertar(arbol, element)
```

```
print(f"Árbol creado con los valores: {values}\n")
```

```
while True:
```

```
    entrada = input("Seleccione el tipo de recorrido:\n1. Inorden\n2. Preorden\n3.
```

```
Postorden\n4. Salir\nOpción: ")
```

```
    if entrada == '1':
```

```
        print("Recorrido inorden:")
```

```
        inorden(arbol)
```

```
    elif entrada == '2':
```

```
        print("Recorrido preorden:")
```

```
        preorden(arbol)
```

```
elif entrada == '3':
    print("Recorrido postorden:")
    postorden(arbol)
elif entrada == '4':
    print("Saliendo del programa.")
    break
else:
    print("Opción no válida. Ingrese numeros del 1 al 4.\n")
```

Cabe aclarar que algunos comentarios que originalmente están en el archivo .py han sido quitados para incluir aquí en el documento, exclusivamente la estructura de código.

## 4. Metodología Utilizada

La implementación se realizó en Python utilizando diccionarios para representar los nodos y el árbol. Se optó por un enfoque recursivo para la inserción y el recorrido, lo que refleja naturalmente como se usan los árboles en casos reales y la estructura jerárquica del árbol.

### [ ACTUALIZACION ]

#### Segunda entrega

El desarrollo de las funciones al comienzo del código son las que van a construir y manipular un árbol binario de búsqueda.

La función **crear\_nodo(value)** se creó para generar un nuevo nodo del árbol, que buscamos representar como un diccionario que guarda el valor del nodo y referencias a sus hijos izquierdo y derecho, inicializados en None (null en otros lenguajes). Esta idea la tomamos analizando como funcionaba la versión con clases, siendo los atributos representados como **keys** del diccionario y lo que contendría la variable se refiere al **value**.

Luego, la función **insertar(nodo, value)** cumple la tarea de construir el árbol. Compara el valor a insertar con el valor del nodo actual, usando condicionales para ver si el valor se añade a la izquierda o a la derecha, con la lógica que hace que un árbol binario de búsqueda se ordena. Basicamente lo que se evalúa es el valor del nuevo nodo y si donde se lo está intentado insertar hay un lugar vacío o ya hay otro valor en dicho lugar.

Las funciones `inorden(nodo)`, `preorden(nodo)` y `postorden(nodo)` se encargan de recorrer el árbol de distintas maneras:

- `Inorden` recorre primero el hijo izquierdo, luego el nodo actual (key "valor") y después el hijo derecho;
- `Preorden` comienza por el nodo actual y luego visita ambos hijos;
- `Postorden` recorre primero los hijos y después el nodo.

Por último, `imprimir_arbol(nodo, nivel=0)` es una función que visualiza mediante `print` con espacios vacíos, los valores que necesitábamos imprimir pasados a **string**. Esto último es fundamental ya que cuando tuvimos la función en mente nos imaginamos algo parecido al resultado final pero nos daba error, hasta que entendimos que pasando los datos a `string` podíamos concatenarlos usando recursividad. De nuevo, haciendo algo de prueba y error, lo cual no es una buena práctica, pero nos sirvió para entender y resolver el problema. Imprime el árbol "acostado", como si lo que está al lado izquierdo, fuese el nodo raíz.

- Estas funciones usan recursión, lo que nos complicó en el desarrollo y usamos `prints` y modo `debug` para revisar la ejecución del programa.  
Las funciones se llaman a sí mismas para repetir el proceso en cada nodo existente para construir el árbol o simplemente imprimir sus valores en el orden elegido.

En el programa principal, estas funciones se utilizan para construir y mostrar el árbol con los datos que el usuario va ingresando. Primero, se solicita un valor para la raíz y luego se siguen pidiendo más valores para insertar. Cada valor válido se guarda en una lista llamada `values`, y al finalizar la carga, todos esos valores se insertan uno por uno en el árbol con `insertar()`. Finalmente, mediante un menú, el usuario puede seleccionar cómo recorrer el árbol (funciones "**orden**") o visualizar su estructura (`imprimir_arbol`).

[ - ]

## 5. Resultados Obtenidos

La ejecución del código muestra el árbol de manera ordenada según el recorrido elegido. La inserción de elementos se realiza sin errores, con validación y el límite lo define el usuario. Hicimos distintas pruebas ingresando varios conjuntos de datos para verificar el correcto funcionamiento del código.

## 6. Conclusiones

El presente trabajo, nos hizo comprender en profundidad la estructura de los árboles y su aplicación mediante un ejemplo práctico en Python. Se demostró la utilidad de los árboles para organizar información y consultarla cuando esta tiene una jerarquía que obedecer. A través del código se logró implementar una versión básica, pero funcional, que sienta las bases para estructuras más complejas en futuros trabajos.

## 7. Bibliografía

- Python Software Foundation. (2023). Documentación oficial de Python.  
<https://docs.python.org/3/>
- Apuntes .pdf de la materia