TECNICATURA UNIVERSITARIA
EN PROGRAMACIÓN

# Trabajo de Investigación: Implementación de Árboles en Python Utilizando Listas

Alumnos:

Rocío Infante – infanterocio01@gmai.com

German Horvath – german.horvath@tupad.utn.edu.ar

Materia: Programación 1

Profesora: Julieta Agustina Trapé

Fecha de entrega: 20/6/2025

**Índice:**



## 1. Introducción

El presente trabajo se centra en el estudio e implementación de árboles binarios como estructura de datos, utilizando listas anidadas en el lenguaje de programación Python. La elección de este tema se debe a la relevancia que tienen los árboles en la informática, tanto por su aplicabilidad en problemas complejos como por su utilidad en optimizar procesos de búsqueda, organización jerárquica de información y construcción de algoritmos eficientes.

En el campo de la programación, las estructuras de datos permiten representar y manipular información de manera organizada. Dentro de ellas, los árboles destacan por su versatilidad y por estar presentes en una amplia variedad de aplicaciones: desde compiladores y bases de datos hasta algoritmos de inteligencia artificial. Comprender su funcionamiento y formas de implementación es fundamental para todo programador.

El objetivo principal de este trabajo es analizar la lógica estructural de los árboles binarios y demostrar que es posible representarlos y operar sobre ellos sin recurrir a clases u objetos,

utilizando únicamente listas. Esto permite reforzar la comprensión conceptual del árbol como estructura recursiva y fomenta el desarrollo de habilidades algorítmicas en un entorno más accesible como lo es Python.

## 2. Objetivos

- Analizar la estructura lógica de un árbol binario y su representación mediante listas anidadas.
- Implementar funciones básicas sobre árboles binarios, como la inserción de nodos y los recorridos en preorden, inorden y postorden.
- Comprender la estructura de árbol orientada a listas con un ejemplo en phyton.

## 3. Marco Teórico

Marco Teórico: Árboles Binarios en Ciencias de la Computación

Un árbol es una estructura de datos jerárquica, no lineal, ampliamente utilizada en informática para modelar relaciones padre-hijo, organizar datos jerárquicamente o representar decisiones. A diferencia de estructuras como listas o arreglos, que almacenan datos en forma secuencial, los árboles permiten una representación más flexible y eficiente de información con relaciones complejas.

### Estructura de un árbol

Un árbol está compuesto por nodos, donde cada nodo contiene un valor o dato, y referencias (o enlaces) a otros nodos llamados hijos. El nodo superior del árbol es denominado raíz, y desde este se derivan los demás nodos. Un nodo que no tiene hijos se llama hoja, y aquellos con al menos un hijo son nodos internos.

### Árbol binario

Un caso particular y muy estudiado es el árbol binario, en el cual cada nodo puede tener como máximo dos hijos, comúnmente denominados hijo izquierdo e hijo derecho. Esta estructura simplificada es la base de algoritmos fundamentales y de otras estructuras como árboles de búsqueda binaria, árboles balanceados, heaps, etc.

Propiedades fundamentales de los árboles binarios:

- Raíz: Nodo inicial del árbol, sin padre.
- Hojas: Nodos que no poseen hijos.
- Altura del árbol: Es el número máximo de niveles desde la raíz hasta la hoja más lejana. Sirve para medir la eficiencia de búsquedas y recorridos.
- Nivel de un nodo: Cantidad de enlaces (o pasos) desde la raíz hasta ese nodo.

- Subárbol: Cualquier nodo junto a todos sus descendientes constituye un subárbol, permitiendo definir un árbol de forma recursiva.
- Grado: Número de hijos de un nodo. En un árbol binario, el grado máximo es 2.

**Recorridos de un árbol binario**

Una de las operaciones más comunes sobre árboles binarios es el recorrido, es decir, visitar todos los nodos en cierto orden. Los tres recorridos más utilizados son:

- Inorden (izquierda, raíz, derecha): útil para árboles de búsqueda binaria, ya que devuelve los datos en orden creciente.
- Preorden (raíz, izquierda, derecha): adecuado para copiar árboles o evaluar expresiones en notación prefija.
- Postorden (izquierda, derecha, raíz): útil para liberar memoria o evaluar expresiones en notación posfija.

**Representación de árboles en distintos lenguajes**

En lenguajes como Java, C++ o C, los árboles suelen implementarse usando clases o estructuras con campos para el valor y punteros hacia los hijos. Esta representación hace uso intensivo de memoria dinámica y referencias.

En Python, debido a su flexibilidad, una opción común es usar listas anidadas, donde cada nodo se representa como una lista de tres elementos:

[valor, subárbol_izquierdo, subárbol_derecho].

Por ejemplo, un árbol binario con raíz 10, hijo izquierdo 5 y derecho 15 se representa como:

[10, [5, None, None], [15, None, None]].

Este enfoque recursivo se alinea con la estructura del árbol, facilita su visualización y permite implementar recorridos y operaciones de forma natural.

**Aplicaciones de los árboles binarios**

Los árboles binarios tienen múltiples aplicaciones en informática y otras ciencias:

- Representación de expresiones matemáticas (árboles de expresión)
- Indexación de bases de datos (árboles B y B+)
- Compiladores (análisis sintáctico)
- Algoritmos de inteligencia artificial (árboles de decisión)
- Motores de búsqueda y compresión de datos (como Huffman)

**4. Metodología**

- Se empleó el lenguaje de programación Python
- La estructura de cada nodo se definió como una lista compuesta por tres elementos: el valor del nodo, el subárbol izquierdo y el subárbol derecho.
- Se implementaron funciones específicas para la creación del árbol, la inserción de nodos tanto en la izquierda como en la derecha, y para realizar los recorridos clásicos: preorden, inorden y postorden.
- Para facilitar la comprensión de la estructura generada, se desarrolló una función que permite visualizar el árbol de forma rotada, representando los nodos de derecha a izquierda en distintos niveles.
- El diseño del código evitó el uso de clases u objetos, con el fin de explorar un enfoque alternativo basado en listas anidadas y promover la comprensión recursiva de la estructura.

**5. Desarrollo / Implementación**

```
# Cada nodo se representa como: [valor, hijo_izquierdo, hijo_derecho]

def crear_arbol(valor):
    return [valor, [], []]  # Nodo con valor, hijo izquierdo vacío, hijo derecho vacío


def insertar_izquierda(nodo, nuevo_valor):
    subarbol_izq = nodo[1]
    if subarbol_izq:  # Si ya hay algo a la izquierda
        nodo[1] = [nuevo_valor, subarbol_izq, []]  # Se inserta como nueva raíz, el árbol viejo queda colgado
    else:
        nodo[1] = [nuevo_valor, [], []]  # Inserta directamente si no hay hijo izquierdo


def insertar_derecha(nodo, nuevo_valor):
    subarbol_der = nodo[2]
    if subarbol_der:  # Si ya hay algo a la derecha
        nodo[2] = [nuevo_valor, [], subarbol_der]  # Se inserta como nueva raíz, el árbol viejo queda colgado
    else:
        nodo[2] = [nuevo_valor, [], []]  # Inserta directamente si no hay hijo derecho
```

```
# Recorrido Preorden: raíz - izquierda - derecha
def preorden(arbol):
    if arbol:
        print(arbol[0], end=' ')
        preorden(arbol[1])
        preorden(arbol[2])

# Recorrido Inorden: izquierda - raíz - derecha
def inorden(arbol):
    if arbol:
        inorden(arbol[1])
        print(arbol[0], end=' ')
        inorden(arbol[2])

# Recorrido Postorden: izquierda - derecha - raíz
def postorden(arbol):
    if arbol:
        postorden(arbol[1])
        postorden(arbol[2])
        print(arbol[0], end=' ')

# Impresión visual del árbol (rotado 90°)
def imprimir_arbol(arbol, nivel=0):
    if arbol:
        imprimir_arbol(arbol[2], nivel + 1)  # Hijo derecho
        print('    ' * nivel + str(arbol[0]))   # Nodo actual
        imprimir_arbol(arbol[1], nivel + 1)  # Hijo izquierdo

# ----------------------

# Ejemplo de uso
```

```
# Crear nodo raíz
arbol = crear_arbol('A')

# Insertar hijos
insertar_izquierda(arbol, 'B')
insertar_derecha(arbol, 'C')
insertar_izquierda(arbol[1], 'D')
insertar_derecha(arbol[1], 'E')
insertar_izquierda(arbol[2], 'F')
insertar_derecha(arbol[2], 'G')

# Visualización del árbol
print("Árbol visualizado (rotado 90°):")
imprimir_arbol(arbol)

# Recorridos
print("\nRecorrido Preorden:")
preorden(arbol)

print("\nRecorrido Inorden:")
inorden(arbol)

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

**6. conclusiones**

El desarrollo de este trabajo permitió afianzar conceptos fundamentales sobre estructuras de datos, en particular sobre árboles binarios. Se comprendió su funcionamiento teórico y también su implementación práctica utilizando listas anidadas en Python, lo cual reforzó lo aprendido como la lógica recursiva y el pensamiento algorítmico.

Este enfoque resultó especialmente útil para visualizar de forma clara cómo se relacionan los nodos y cómo se construyen las distintas ramas del árbol.

Durante el proceso surgieron algunas dificultades, especialmente relacionadas con la correcta manipulación de listas anidadas y la visualización del árbol. Estos problemas se resolvieron mediante el análisis en equipo y pruebas del código.

Como posibles mejoras podríamos aplicar esto a alguna situación real.

**7.Bibliografia**

- Python Software Foundation. (2024). Python 3 Documentation. https://docs.python.org/3/