

PROGRAMACIÓN

**Tecnicatura Universitaria en Programación - Universidad Tecnológica
Nacional. Arquitectura y Sistemas Operativos**

Alumnos:

Galo Coria Maiorano – galocoriamai@gmail.com

Lucas Gragera – lucasgragera51@gmail.com

Materia: Programación

Profesor: Bruselario, Sebastián

Fecha de Entrega: 09 de junio de 2025

Índice

1. Introducción
2. Marco Teórico
3. Caso Práctico
4. Metodología Utilizada
5. Resultados Obtenidos
6. Conclusiones
7. Bibliografía
8. Anexos

1. Introducción

Elegimos el tema "**Implementación de Árboles en Python utilizando listas**" porque nos parece muy útil y esencial para nuestra formación como programadores. Estas estructuras jerárquicas permiten almacenar y manipular información de forma eficiente, aprovechando las capacidades de las listas en Python.

Los **árboles binarios de búsqueda (BST)** son una potente estructura de datos que facilita la organización de información, permitiendo una búsqueda y recuperación de valores de manera rápida y ordenada.

La importancia de los árboles en la programación es clave, ya que permiten:

- Organizar la información de forma jerárquica y eficiente.
- Realizar búsquedas y clasificaciones de manera rápida.
- Recorrer datos mediante diferentes métodos como: **preorden**, **inorden**, **postorden** y **por niveles**, según las necesidades del problema.

En la actualidad, con el crecimiento de la **inteligencia artificial** y la **ciencia de datos**, el uso de estructuras como los árboles es cada vez más relevante.

Algoritmos como los **árboles de decisión** se basan precisamente en estas estructuras para tomar decisiones, clasificar información y predecir resultados de forma precisa y automatizada.

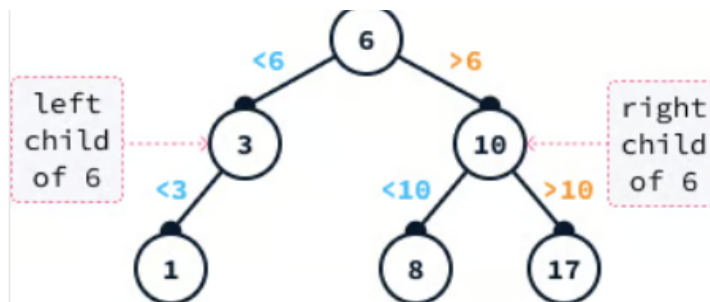
Nuestro objetivo con este trabajo integrador es demostrar la importancia de los árboles en la programación, así como también su correcto funcionamiento e implementación.

2. Marco Teórico

Los árboles binarios de búsqueda (BST) son una potente estructura de datos para organizar información, lo que permite una búsqueda y recuperación de valores eficiente.

Los árboles binarios de búsqueda organizan los datos mediante un orden específico. **Cada nodo contiene un valor y tiene enlaces a hasta dos nodos más: el hijo izquierdo y el hijo derecho .**

En un BST, la regla es que el valor del hijo izquierdo debe ser menor que el de su nodo padre, y el del hijo derecho debe ser mayor, como se ve en la foto.



¿Qué es un árbol?

Un árbol es una estructura de datos jerárquica compuesta por nodos, donde **cada nodo puede tener cero o más nodos hijos. El nodo principal se llama raíz y los nodos sin hijos se llaman hojas.**

¿Por qué usar listas?

En Python, una manera sencilla y didáctica de representar árboles es usando listas anidadas.

Representación de un árbol con listas

Un árbol binario (máximo dos hijos por nodo) puede representarse se puede representar de forma sencilla utilizando listas anidadas:

Árbol binario: [raíz, subárbol izquierdo, subárbol derecho]

arbol = [10, [5, None, None], [20, None, None]]

Esto representa:

10

**/ **

5 20

Cada nodo es una lista de tres elementos:

- El valor del nodo.
- Su subárbol izquierdo (otra lista o None).
- Su subárbol derecho (otra lista o None).

Operaciones básicas

Algunas operaciones típicas sobre árboles incluyen:

- Recorridos** (preorden, inorden, postorden).
- Búsqueda de un valor.**
- Inserción de nodos.**
- Eliminación.**

3. Caso Práctico

Este es el árbol que diseñamos para implementar árboles de lista en Python. Decidimos agregar varios nodos en la rama derecha y algunos en la rama izquierda, ya que como vimos en el teórico, la rama derecha debe ser la mayor en que la rama izquierda.

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

```
def crear_arbol(valor): #Creamos el arbol
    return [valor, [], []]
```

```
def insertar_izquierda(nodo, nuevo_valor): #Integra la rama izquierda
    subarbol_izq = nodo[1]
    if subarbol_izq:
        nodo[1] = [nuevo_valor, subarbol_izq, []]
    else:
        nodo[1] = [nuevo_valor, [], []]
```

```
def insertar_derecha(nodo, nuevo_valor): #Integra la rama derecha
    subarbol_der = nodo[2]
    if subarbol_der:
        nodo[2] = [nuevo_valor, [], subarbol_der]
    else:
        nodo[2] = [nuevo_valor, [], []]
```

Recorridos con impresión de nodos vacíos explícitos (None)

```
def preorden(arbol):
    if arbol:
        print(arbol[0], end=' ')
        preorden(arbol[1])
        preorden(arbol[2])
```

```
def inorden(arbol):
    if arbol:
        inorden(arbol[1])
        print(arbol[0], end=' ')
        inorden(arbol[2])
```

```
def postorden(arbol):
    if arbol:
        postorden(arbol[1])
```

```
    postorden(arbol[2])
    print(arbol[0], end=' ')

# Visualización del árbol, rotado 90° hacia la izquierda
def imprimir_arbol(arbol, nivel=0):
    if arbol:
        imprimir_arbol(arbol[2], nivel + 1) # Subárbol derecho
        print(' ' * 4 * nivel + str(arbol[0])) # Nodo actual, con sangría
        imprimir_arbol(arbol[1], nivel + 1) # Subárbol izquierdo

# -----
# Ejemplo de uso
# -----

# Crear árbol binario con números
arbol = crear_arbol(7)
insertar_izquierda(arbol, 10)
insertar_derecha(arbol, 9)
insertar_izquierda(arbol[1], 1)
insertar_izquierda(arbol[1][1], 20)
insertar_derecha(arbol[1], 5)
insertar_izquierda(arbol[2], 40)
insertar_derecha(arbol[2], 55)
insertar_derecha(arbol[2][2], 60)
insertar_derecha(arbol[2][2][2], 65)

# Mostrar el 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)
```

Resultado:

Árbol visualizado (rotado 90°):

```
      65
     60
    55
   9
  40
 7
   5
  10
   1
    20
```

Recorrido Preorden:

7 10 1 20 5 9 40 55 60 65

Recorrido Inorden:

20 1 10 5 7 40 9 55 60 65

Recorrido Postorden:

20 1 5 10 40 65 60 55 9 7

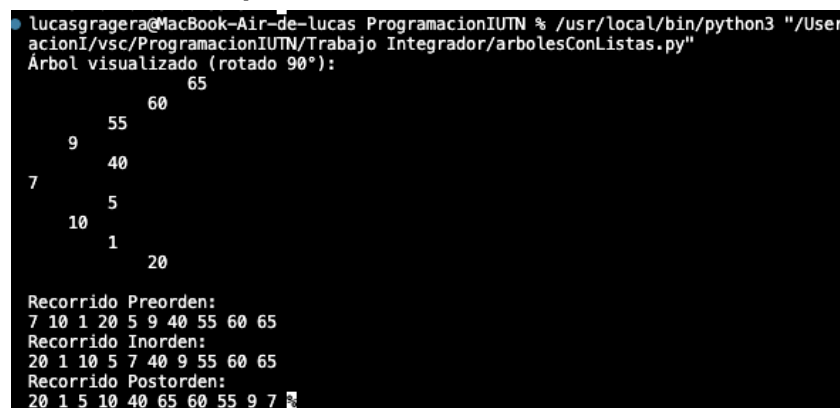
En la creación del árbol binario con números, para agregar ramas, se utiliza el [1] para las ramas izquierdas y el [2] para las ramas derechas.

Ejemplos:

```
insertar_derecha(arbol[2][2][2], 65)
```

```
insertar_izquierda(arbol[1][1], 20)
```

Captura del resultado el VSC:



```
lucasgragera@MacBook-Air-de-lucas ProgramacionIUTN % /usr/local/bin/python3 "/User
acionI/vsc/ProgramacionIUTN/Trabajo Integrador/arbolesConListas.py"
Árbol visualizado (rotado 90°):
      65
     60
    55
   9
  40
 7
   5
  10
   1
    20

Recorrido Preorden:
7 10 1 20 5 9 40 55 60 65
Recorrido Inorden:
20 1 10 5 7 40 9 55 60 65
Recorrido Postorden:
20 1 5 10 40 65 60 55 9 7
```

4. Metodología Utilizada

Nuestro trabajo comenzó antes de elegir este tema, hicimos una investigación breve sobre los temas a elegir, y esta nos pareció la más interesante.

Para el marco teórico hicimos una investigación más profunda en distintas páginas hasta que nos encontramos con “datacamp” que fue de gran ayuda.

En cuanto al desarrollo, utilizamos el modelo como base, y desde ahí comenzamos a editarlo y agregarle líneas de código como ramas y hojas.

Utilizamos la página mencionada y además ayuda de la IA para ciertos casos.

El reparto de tareas en si no se llevó a cabo, se realizó de manera sincrónica mediante videollamada entre ambos integrantes, generando todas las actividades en conjunto, sin necesidad de división de las mismas.

5. Resultados Obtenidos

La implementación se llevó a cabo de manera exitosa, desarrollando el marco teórico y alcanzando los objetivos previamente propuestos.

Cabe destacar la simplicidad y eficiencia de esta implementación mediante el árbol binario, el código es bastante legible y fácil de entender. Pero no se deja de lado la gran limitación de no poder agregar más de dos ramas a un nodo.

Afortunadamente no se nos presentaron grandes dificultades, y el código pudo ser llevado a cabo sin inconvenientes mayores.

Tuvimos algunos inconvenientes con el agregado de ramas debido a los índices de izquierda [1] y derecha [2], pero lo logramos entender y pudimos llevarlo a cabo.

Link al repositorio de GitHub:

<https://github.com/lucasgragera/ProgramacionIUTN/blob/main/Trabajo%20Integrador/arbolesConListas.py>

6. Conclusiones

El tema que elegimos para este trabajo integrador representa un pilar fundamental en la formación de cualquier programador, ya que incluye listas anidadas en conjunto con teorías importantes.

Es una metodología muy fácil de implementar a Python debido a su simplicidad y facilidad de crear y editar lo cual es una muy buena ventaja. Su gran desventaja y limitante es la imposibilidad de agregar más de dos ramas a un nodo, aunque en lo personal y llevando a cabo este trabajo integrador, no nos presentó ninguna dificultad.

7. Bibliografía

- <https://www.datacamp.com/tutorial/avl-tree>
- Ayudas de la IA en algunos puntos del trabajo integrador