

Proyecto Integrador: Árboles Binarios en Python Utilizando Listas

Título del proyecto: Árboles Binarios en Python con Listas

Alumno: Salomon Marcos Farid

Materia: Programación I

Profesora: Cinthia Rigoni

Tutor: Walter Pintos

Fecha de entrega: 09 de junio de 2025

Link Youtube: <https://youtu.be/vj-7blXKu6k>

Link Github: <https://github.com/turkaym/UTN-TUPaD-P1/tree/main/12%20Datos%20Avanzados>

1. Introducción

El árbol binario es una estructura de datos jerárquica en la que cada nodo puede tener como máximo dos hijos (izquierdo y derecho). Se aplica en sistemas de archivos, algoritmos de ordenamiento y búsqueda eficiente. En este trabajo se empleará una representación con listas anidadas en Python para facilitar la comprensión de la mecánica de inserción, los recorridos y la visualización.

2. Marco Teórico

- **Definición:** Un árbol binario es un grafo acíclico con un nodo raíz y, en cada nodo, un máximo de dos subárboles.
- **Elementos:**
 - **Nodo:** [valor, subárbol_izquierdo, subárbol_derecho].
 - **Raíz:** nodo principal sin padre.
 - **Hojas:** nodos sin hijos (None).
- **Propiedades:**
 - Altura, profundidad, grado y peso.
- **Recorridos:**
 - **Inorden:** izq → raíz → der.
 - **Preorden:** raíz → izq → der.
 - **Postorden:** izq → der → raíz.
- **Representación con Listas:**
 - Cada nodo es una lista de 3 elementos; los subárboles pueden ser listas o None.
- **Complejidad:**
 - Inserción y búsqueda en árboles no balanceados: $O(n)$ en el peor caso.

3. Metodología Utilizada

1. Estudio de características de árboles binarios.
2. Implementación manual de funciones sobre listas.
3. Programación recursiva de recorridos (inorden, preorden, postorden).
4. Testeo con valores predefinidos.
5. Extensión con búsqueda, cálculo de altura y recorrido por niveles.

4. Desarrollo: Script Inicial

#modulo árbol_lista.py

"""

Módulo principal: define la estructura básica del árbol y sus operaciones.

"""

def crear_arbol(valor):

"""

Crea un nuevo árbol con el valor dado y subárboles vacíos.

:param valor: dato a almacenar en la raíz del árbol

:return: lista [valor, None, None]

"""

return [valor, None, None]

def insertar_izquierda(arbol, valor):

"""

Inserta un nuevo nodo con 'valor' como hijo izquierdo.

Si ya había, ese subárbol pasa a ser hijo izquierdo del nuevo nodo.

"""

if arbol[1] is None:

arbol[1] = crear_arbol(valor)

else:

```
nuevo = crear_arbol(valor)
nuevo[1] = arbol[1]
arbol[1] = nuevo
```

```
def insertar_derecha(arbol, valor):
```

```
    """
```

Inserta un nuevo nodo con 'valor' como hijo derecho.

Si ya había, ese subárbol pasa a ser hijo derecho del nuevo nodo.

```
    """
```

```
    if arbol[2] is None:
```

```
        arbol[2] = crear_arbol(valor)
```

```
    else:
```

```
        nuevo = crear_arbol(valor)
```

```
        nuevo[2] = arbol[2]
```

```
        arbol[2] = nuevo
```

```
def imprimir_arbol(arbol, nivel=0):
```

```
    """
```

Imprime el árbol rotado 90° en sentido horario.

:param nivel: para controlar sangrías según profundidad.

```
    """
```

```
    if arbol is None:
```

```
        return
```

```
    imprimir_arbol(arbol[2], nivel+1)
```

```
    print(' ' * nivel + str(arbol[0]))
```

```
    imprimir_arbol(arbol[1], nivel+1)
```

```

2 Datos Avanzados > arbol_listas.py > imprimir_arbol
1  """
2  Módulo principal: define la estructura básica del árbol y sus operaciones.
3  """
4
5
6  def crear_arbol(valor):
7      """
8      Crea un nuevo árbol con el valor dado y subárboles vacíos.
9      :param valor: dato a almacenar en la raíz del árbol
10     :return: lista [valor, None, None]
11     """
12     return [valor, None, None]
13
14
15  def insertar_izquierda(arbol, valor):
16      """
17      Inserta un nuevo nodo con 'valor' como hijo izquierdo.
18      Si ya había, ese subárbol pasa a ser hijo izquierdo del nuevo nodo.
19      """
20      if arbol[1] is None:
21          arbol[1] = crear_arbol(valor)
22      else:
23          nuevo = crear_arbol(valor)
24          nuevo[1] = arbol[1]
25          arbol[1] = nuevo
26
27
28  def insertar_derecha(arbol, valor):
29      """
30      Inserta un nuevo nodo con 'valor' como hijo derecho.
31      Si ya había, ese subárbol pasa a ser hijo derecho del nuevo nodo.
32      """
33      if arbol[2] is None:
34          arbol[2] = crear_arbol(valor)
35      else:
36          nuevo = crear_arbol(valor)
37          nuevo[2] = arbol[2]
38          arbol[2] = nuevo
39
40
41  def imprimir_arbol(arbol, nivel=0):
42      """
43      Imprime el árbol rotado 90° en sentido horario.
44      :param nivel: para controlar sangrías según profundidad.
45      """
46      if arbol is None:
47          return
48      imprimir_arbol(arbol[2], nivel+1)
49      print('    ' * nivel + str(arbol[0]))
50      imprimir_arbol(arbol[1], nivel+1)
51

```

```
# utilidades_arbol.py
```

```
"""
```

Módulo de utilidades: recorridos, búsqueda y análisis.

```
"""
```

```
def inorden(arbol, resultado=None):
```

```
    """Recorrido inorden: izq → raíz → der."""
```

```
    if resultado is None:
```

```
        resultado = []
```

```
    if arbol is not None:
```

```
        inorden(arbol[1], resultado)
```

```
        resultado.append(arbol[0])
```

```
        inorden(arbol[2], resultado)
```

```
    return resultado
```

```
def preorden(arbol, resultado=None):
```

```
    """Recorrido preorden: raíz → izq → der."""
```

```
    if resultado is None:
```

```
        resultado = []
```

```
    if arbol is not None:
```

```
        resultado.append(arbol[0])
```

```
        preorden(arbol[1], resultado)
```

```
        preorden(arbol[2], resultado)
```

```
    return resultado
```

```
def postorden(arbol, resultado=None):
```

```
    """Recorrido postorden: izq → der → raíz."""
```

```
    if resultado is None:
```

```
        resultado = []
```

```

if arbol is not None:
    postorden(arbol[1], resultado)
    postorden(arbol[2], resultado)
    resultado.append(arbol[0])
return resultado

```

```

def buscar(arbol, valor):
    """Retorna True si 'valor' existe en el árbol."""
    if arbol is None:
        return False
    if arbol[0] == valor:
        return True
    return buscar(arbol[1], valor) or buscar(arbol[2], valor)

```

```

def altura(arbol):
    """Retorna la altura máxima del árbol."""
    if arbol is None:
        return 0
    return 1 + max(altura(arbol[1]), altura(arbol[2]))

```

```

def nivel_por_nivel(arbol):
    """Recorrido por niveles (BFS)."""
    if arbol is None:
        return []
    cola, resultado = [arbol], []
    while cola:
        nodo = cola.pop(0)
        resultado.append(nodo[0])
        if nodo[1]:

```

```

cola.append(nodo[1])

if nodo[2]:
    cola.append(nodo[2])

return resultado.

```

```

2 Datos Avanzados > utilidades_arbol.py > ...
1  """
2  Módulo de utilidades: recorridos, búsqueda y análisis.
3  """
4
5
6  def inorden(arbol, resultado=None):
7      """Recorrido inorden: izq → raíz → der."""
8      if resultado is None:
9          resultado = []
10     if arbol is not None:
11         inorden(arbol[1], resultado)
12         resultado.append(arbol[0])
13         inorden(arbol[2], resultado)
14     return resultado
15
16
17  def preorden(arbol, resultado=None):
18      """Recorrido preorden: raíz → izq → der."""
19      if resultado is None:
20          resultado = []
21      if arbol is not None:
22          resultado.append(arbol[0])
23          preorden(arbol[1], resultado)
24          preorden(arbol[2], resultado)
25      return resultado
26
27
28  def postorden(arbol, resultado=None):
29      """Recorrido postorden: izq → der → raíz."""
30      if resultado is None:
31          resultado = []
32      if arbol is not None:
33          postorden(arbol[1], resultado)
34          postorden(arbol[2], resultado)
35          resultado.append(arbol[0])
36      return resultado
37
38

```

```

8
9 def buscar(arbol, valor):
10     """Retorna True si 'valor' existe en el árbol, False en caso contrario."""
11     if arbol is None:
12         return False
13     if arbol[0] == valor:
14         return True
15     return buscar(arbol[1], valor) or buscar(arbol[2], valor)
16
17
18 def altura(arbol):
19     """Retorna la altura máxima del árbol."""
20     if arbol is None:
21         return 0
22     return 1 + max(altura(arbol[1]), altura(arbol[2]))
23
24
25 def nivel_por_nivel(arbol):
26     """Recorrido por niveles (BFS)."""
27     if arbol is None:
28         return []
29     cola, resultado = [arbol], []
30     while cola:
31         nodo = cola.pop(0)
32         resultado.append(nodo[0])
33         if nodo[1]:
34             cola.append(nodo[1])
35         if nodo[2]:
36             cola.append(nodo[2])
37     return resultado
38

```

main.py

```

from arbol_listas import crear_arbol, insertar_izquierda, insertar_derecha, imprimir_arbol
from utilidades_arbol import inorden, preorden, postorden, buscar, altura, nivel_por_nivel

```

```

if __name__ == "__main__":

```

```

    # 1. Creamos la raíz

```

```

    raiz = crear_arbol(10)

```

```

    # 2. Hijos de la raíz

```



```
insertar_izquierda(raiz, 5)
```

```
insertar_derecha(raiz, 15)
```

```
# 3. Nietos de la raíz (hijos de 5)
```

```
insertar_izquierda(raiz[1], 3)
```

```
insertar_derecha(raiz[1], 7)
```

```
# 4. Hijos de 15
```

```
insertar_izquierda(raiz[2], 17)
```

```
insertar_derecha(raiz[2], 20)
```

```
# 5. Visualizamos el árbol y usamos utilidades
```

```
print("Impresión del árbol:")
```

```
imprimir_arbol(raiz)
```

```
print("Inorden:", inorden(raiz))
```

```
print("Preorden:", preorden(raiz))
```

```
print("Postorden:", postorden(raiz))
```

```
print("Buscar 7:", buscar(raiz, 7))
```

```
print("Altura:", altura(raiz))
```

```
print("Por niveles:", nivel_por_nivel(raiz))
```

```

12 Datos Avanzados > main.py > ...
1  from arbol_listas import crear_arbol, insertar_izquierda, insertar_derecha, imprimir_arbol
2  from utilidades_arbol import inorden, preorden, postorden, buscar, altura, nivel_por_nivel
3
4  if __name__ == "__main__":
5      # 1. Creamos la raíz
6      raiz = crear_arbol(10)
7
8      # 2. Hijos de la raíz
9      insertar_izquierda(raiz, 5)
10     insertar_derecha(raiz, 15)
11
12     # 3. Nietos de la raíz (hijos de 5)
13     insertar_izquierda(raiz[1], 3)
14     insertar_derecha(raiz[1], 7)
15
16     # 4. Hijos de 15
17     insertar_izquierda(raiz[2], 17)
18     insertar_derecha(raiz[2], 20)
19
20     # 5. Visualizamos el árbol y usamos utilidades
21     print("Impresión del árbol:")
22     imprimir_arbol(raiz)
23
24     print("Inorden:", inorden(raiz))
25     print("Preorden:", preorden(raiz))
26     print("Postorden:", postorden(raiz))
27     print("Buscar 7:", buscar(raiz, 7))
28     print("Altura:", altura(raiz))
29     print("Por niveles:", nivel_por_nivel(raiz))
30

```

5. Resultados Obtenidos

- El árbol binario se construyó correctamente con nodos insertados según su valor.
- Los recorridos Inorden, Preorden y Postorden se realizaron de manera correcta.
- Búsqueda y cálculo de altura funcionaron adecuadamente.
- La visualización con `imprimir_arbol` muestra la estructura jerárquica rotada 90°.

6. Conclusiones

El uso de árboles binarios en Python facilita la organización de datos de manera eficiente. Los algoritmos de recorrido permiten explorar la estructura según las necesidades de cada problema.

Reflexión: Aprender estructuras de datos como árboles mejora las habilidades de programación y fomenta el pensamiento estructurado para resolver problemas complejos.

7. Bibliografía

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.
- Documentación de Python: <https://docs.python.org/3/tutorial/>

- Visualgo: <https://visualgo.net/en/bst>

8. Anexos

- Capturas de pantalla del código funcionando en consola.
- Repositorio en GitHub: <https://github.com/grupo-arboles/estructura-arbol-python>
- Video explicativo mostrando la estructura del árbol, los recorridos y reflexión final.

9. Entregables

- Informe (PDF) con todas las secciones anteriores.
- Script Python: `arbol_listas.py` / `utilidades.py` / `main.py`.
- Capturas de ejecución y presentación en PowerPoint.