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

```python
"""
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)

```

```
# 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 > ...

```python
"""
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


```

```python

def buscar(arbol, valor):
    """Retorna True si 'valor' existe en el árbol, False en caso contrario."""
    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

```

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 > ...

```python
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))

```

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.