Trabajo integrador - Programación I
Estructuras de datos avanzados: árboles

- *Datos generales.*

Alumnos:

➢ García Ramiro - ramigarcia2476@gmail.com

➢ Godoy Tomás - godoytomás96@gmail.com

Materia: Programación I

Profesor: Ariel Enferrel // Ramiro Hualpa

Fecha de entrega: 20/06/2025

- *Índice.*

- *Objetivos*

➢ Comprender cómo se puede representar un árbol binario utilizando listas en Python.

➢ Implementar operaciones básicas sobre árboles: inserción y recorridos.

➢ Evaluar las ventajas y desventajas de este enfoque frente a implementaciones orientadas a objetos.

➢ Promover el uso de representaciones simples como herramienta educativa.

- *Introducción*

En el desarrollo de software, las estructuras de datos son fundamentales para organizar y manipular información de forma eficiente. A medida que se abordan problemas más complejos, se vuelve necesario recurrir a estructuras de datos avanzadas como lo son los árboles, los grafos, las listas, entre otras. Estas estructuras permiten representar jerarquías, relaciones entre elementos y operaciones más eficientes que las ofrecidas por listas o diccionarios simples.

Los árboles son una de las estructuras de datos más fundamentales y poderosas en informática. Son utilizados en una amplia variedad de aplicaciones del mundo real, como la organización de archivos en sistemas operativos, la gestión de bases de datos (por ejemplo, en árboles de búsqueda binaria), y en algoritmos de optimización.

Un grafo es una estructura matemática utilizada para representar relaciones entre objetos. Se compone de un conjunto de nodos (también llamados vértices) y un conjunto de aristas (también conocidas como enlaces o ramas). Los grafos se utilizan para modelar una amplia variedad de problemas en los que los objetos están conectados de alguna forma, como redes sociales, sistemas de navegación, redes de comunicación, entre otros.

En Python, aunque existen bibliotecas específicas para trabajar con árboles, también es posible implementarlos de manera sencilla utilizando listas. Esta técnica consiste en representar cada nodo del árbol como una lista que contiene:

- El valor del nodo.
- Una sublista con sus nodos subyacentes (o referencias a ellos).

Este enfoque, aunque no tan eficiente ni escalable como las implementaciones basadas en clases o nodos enlazados, es útil para entender la lógica interna de los árboles y practicar su manipulación en Python de forma didáctica.

• Marco teórico

Las **estructuras de datos avanzadas** son componentes esenciales en el diseño de algoritmos eficientes y en la resolución de problemas complejos. Estas estructuras permiten representar datos y sus relaciones de maneras que optimizan el uso de memoria, el tiempo de procesamiento y la claridad del código. Dentro de estas estructuras, los árboles ocupan un lugar central por su capacidad de modelar relaciones jerárquicas y por sus múltiples aplicaciones prácticas, como en motores de búsqueda, compiladores, sistemas de archivos y algoritmos de inteligencia artificial .

En computación un ***árbol*** es una estructura de datos no lineal que se compone de nodos conectados entre sí mediante aristas (enlaces). Sus principales características son:

- Tener un nodo especial denominado **raíz**, que es el punto de partida.
- Cada nodo (excepto la raíz) tiene exactamente un **padre**.
- Los nodos pueden tener cero o más **hijos**.
- Son estructuras acíclicas

A continuación, en la *figura 1,* se muestra un ejemplo sencillo de un diagrama de árbol.

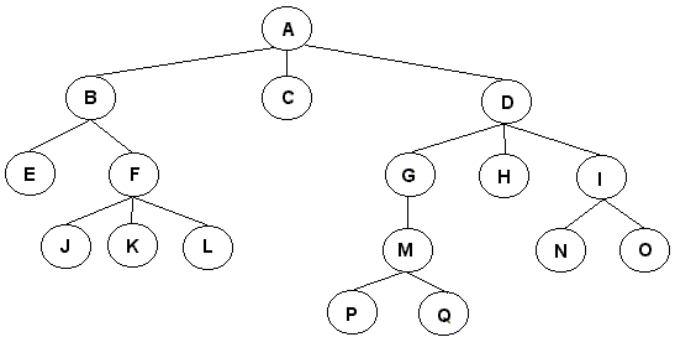

Figura 1: Representación de grafos de un árbol.

El caso más simplificado es el del **árbol binario.** Éste es un tipo especial en el

que cada nodo puede tener, como máximo, dos hijos: un hijo izquierdo y un hijo derecho (*figura 2*). Esto lo convierte en una estructura particularmente versátil y ampliamente estudiada, dado que muchos algoritmos se basan en árboles binarios para lograr un buen rendimiento (por ejemplo, en búsquedas y ordenamientos).

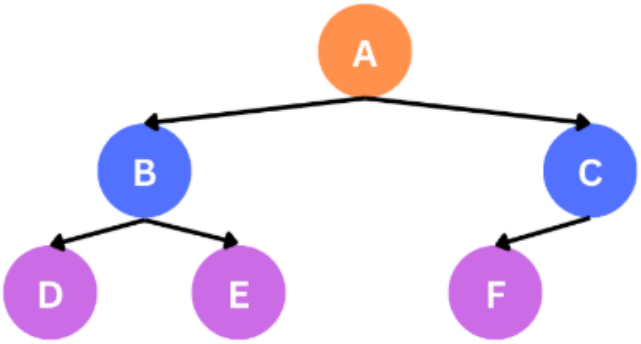

Figura 2: Diagrama de árbol binario.

Si bien Python no incluye en su biblioteca estándar un tipo de datos nativo para árboles binarios, existen múltiples formas de representarlos. Las más comunes son a través del uso de **listas**, **clases** y **objetos.**

- Metodología utilizada

Para el correcto desarrollo del trabajo práctico integrador se comenzó buscando información y ejemplos de aplicación de los diagramas de árbol. Se consultaron los materiales teóricos brindados por la cátedra.

Una vez elegido el tema se desarrolló el código para poner en práctica cada uno de los temas vistos. Ésta parte del trabajo se realizó de manera conjunta entre los autores del texto y en simultáneo. Se considera imprescindible el trabajo en equipo para poder cumplir con los objetivos propuestos previamente.

Se utilizó el lenguaje Python 3.13. Para una óptima explicación del trabajo realizado, en la siguiente sección se muestra el código completo desarrollado. Además, para que el lector pueda probarlo se adjunta un enlace de referencia con el archivo .py en el anexo del presente informe.

- Desarrollo e implementación

El programa ordena y crea un árbol genealógico a partir de distintos personajes ficticios.

En la *figura 3* se muestra como mediante listas anidadas se definen a “los nietos” (las listas vacías se agregan para indicar que cada nieto no tiene hijos, es decir, donde el árbol termina).

```
# Crear nietos
nietos_hijo1 = [["Gabriel", []], ["Jose", []], ["Ana", []]]
nietos_hijo2 = [["Pedro", []], ["Silvia", []]]
nietos_hijo3 = [["Oscar", []], ["Manuel", []], ["Carlos", []]]
```

Figura 3: Creación de listas anidadas definiendo a cada uno de los nietos.

De la misma manera, como se muestra en la *figura 4*, con una lista anidada se crearon las listas para cada hijo con su respectivo nombre en la posición lista[0], dejando los demás elementos de la lista para los nietos.

```
# Crear hijos con sus respectivos nietos
hijo1 = ["Nora", nietos_hijo1]
hijo2 = ["Sonia", nietos_hijo2]
hijo3 = ["Hugo", nietos_hijo3]
```

Figura 4: Creación de listas anidadas para cada uno de los hijos.

Luego, en la *figura 5*, se completa el árbol en sí, con su padre (nodo raíz) que sería en este caso “Salvador”.

```
# Crear el nodo padre con los hijos
arbol = ["Salvador", [hijo1, hijo2, hijo3]]
```

Figura 5: Se establece el nodo raíz.

De esta manera nuestro árbol estaría conformado por el nodo raíz, que corresponde a “Salvador”, los nodos ramas que serían “Nora”, “Sonia” y “Hugo” y, por último, los nodos hojas como “Gabriel”, “José”, “Ana”, “Pedro”, “Silvia”, “Oscar”, “Manuel” y “Carlos”. El diagrama de árbol correspondiente se muestra en la *figura 6*.

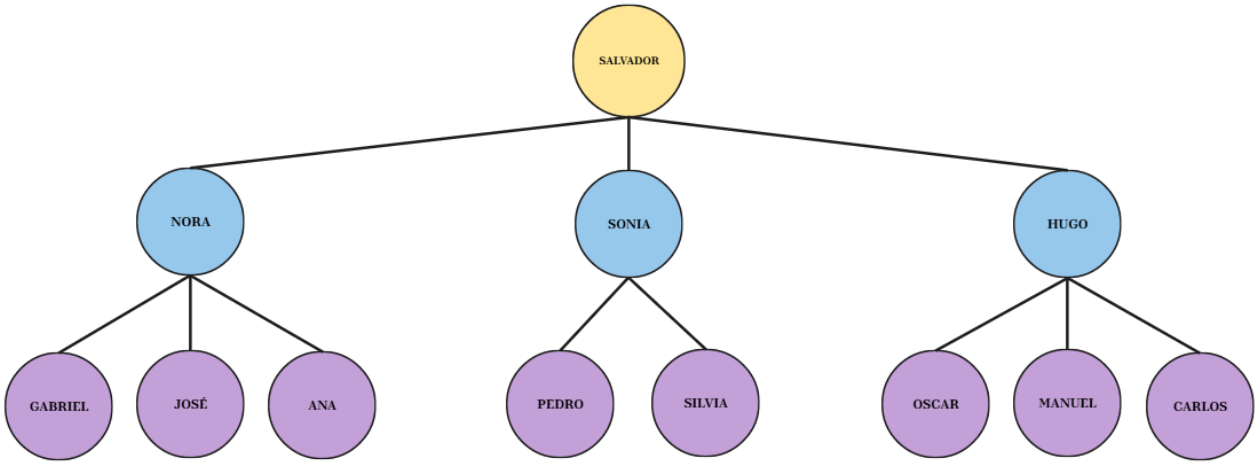

Figura 6: Diagrama de árbol correspondiente al caso práctico.

A modo ilustrativo, en la *figura 7* se muestra las listas anidadas correspondientes al diagrama de árbol citado en la *figura 6*

```
[“Salvador,
    [“Nora”,
        ["Gabriel",[]],
        ["Jose",[]],
        ["Ana",[]]
],
    [“Sonia”,
        ["Pedro",[]],
        ["Silvia",[]]
],
    [“Hugo”,
        ["Oscar",[]],
        ["Manuel",[]],
        ["Carlos",[]]
]
]
```

Figura 7: listas anidadas.

Luego, en el bloque de código que se muestra en la *figura 8*, se define a la función creada para imprimir el árbol. Al parámetro nodo se le envía la lista “árbol” para que luego mediante el bucle *for* y por recursividad se vaya recorriendo toda la lista.

```
# Función para imprimir el árbol
def imprimir_arbol(nodo, nivel=0):
    print("  " * nivel + "- " + nodo[0]) # viñetas
    for hijo in nodo[1]:
        imprimir_arbol(hijo, nivel + 1) #recursividad
```

Figura 8: Función estableciendo el árbol.

La función mostrada en la *figura 9* define el grado del nodo que se desee. Requiere recibir por parámetro la lista del árbol completa (nodo) y el nombre del nodo que deseemos saber su grado. Si pasamos como nombre el nodo raíz, va a retornar la longitud de la lista del árbol, que en este caso sería 3 (hijo1, hijo2, hijo2)(Nora, Sonia, Hugo). De no ser así, mediante recursividad irá recorriendo la lista de árbol hasta que coincida el nombre que se ingresó con el primer elemento de la lista que se está analizando y, así, entregará su longitud.

En caso de ser ingresado un nombre que no corresponde al árbol, el programa retornará “None”.

```
# Función para obtener el grado de un nodo por nombre
def grado_nodo(nodo, nombre):
    if nodo[0] == nombre:
        return len(nodo[1])
    for hijo in nodo[1]:
        grado = grado_nodo(hijo, nombre)
        if grado is not None:
            return grado
    return None  # Si no se encuentra el nodo
```

Figura 9: función que define el grado del nodo que se desee.

Por último, la función “grado_arbol” (*figura 10*) brinda el grado máximo que existe en el árbol. Guarda el grado del nodo actual, es decir la cantidad de hijos, luego en el bucle *for* va a recorrer cada hijo del nodo actual y mediante recursividad va a calcular el grado de cada sub árbol. Finalmente, va a retornar el grado máximo encontrado.

```python
# Función para obtener el grado del árbol (máximo grado entre todos los nodos)
def grado_arbol(nodo):
    grados = [len(nodo[1])]
    for hijo in nodo[1]:
        grados.append(grado_arbol(hijo))
    return max(grados)
```

Figura 10: función que determina el grado del árbol.

El programa principal quedaría tal como se muestra en la *figura 11*.

```python
# Imprimir el árbol
print("Árbol:")
imprimir_arbol(arbol)

# Consultar el grado de un nodo específico
nombre_nodo = "Sonia"
grado = grado_nodo(arbol, nombre_nodo)
if grado is not None:
    print(f"\nGrado del nodo '{nombre_nodo}': {grado}")
else:
    print(f"\nNodo '{nombre_nodo}' no encontrado.")

# Consultar el grado del árbol
print(f"Grado del árbol: {grado_arbol(arbol)}")
```

Figura 11: programa final.

● Resultados

Se logró construir el código necesario para poder implementar listas anidadas y así usar diagramas de árbol.

Durante la realización del trabajo se ha concluído que estas herramientas tienen varios puntos a favor y en contra a la hora de ser consideradas para usar. Entre las **ventajas** se puede destacar:

- ❖ Es simple y no requiere definir clases o estructuras complejas.
- ❖ Permite construir árboles rápidamente en programas pequeños o con fines educativos.
- ❖ Ocupa menos líneas de código para árboles pequeños.

Y, como **desventajas**, se concluye que:

- ❖ La manipulación del árbol (inserción, eliminación, búsqueda) se vuelve más engorrosa y menos eficiente a medida que el árbol crece.
- ❖ Es más difícil de leer y mantener, especialmente si el árbol es grande o profundo.
- ❖ Carece de la flexibilidad que ofrecen los objetos al agregar métodos y atributos personalizados.

● Conclusiones

El estudio de los árboles binarios y su implementación, incluso con una estructura tan básica como las listas de Python, es clave para comprender cómo se organizan y procesan datos jerárquicos. Aunque las listas no son la opción más eficiente para aplicaciones reales que requieren árboles binarios, permiten entender los conceptos fundamentales de forma clara. Cuando se necesiten soluciones más robustas y escalables, es recomendable migrar a implementaciones basadas en clases o utilizar bibliotecas especializadas.

- Bibliografía

- Cormen, T., Leiserson, C., Rivest, R., Stein, C. (2009). Introduction to Algorithms.
- Python Software Foundation. https://docs.python.org
- Miller, B. & Ranum, D. Problem Solving with Algorithms and Data Structures using Python.

- Anexo

Link con el archivo .py para ejecutar el código: Código integrador