

## Trabajo práctico integrador.

### Datos avanzados, Árboles Binarios de Búsqueda.

#### Alumnos:

Reynoso Franco - [franreynoso12@gmail.com](mailto:franreynoso12@gmail.com)

Rojas Maximiliano - [maximilianoyamil1@gmail.com](mailto:maximilianoyamil1@gmail.com)

**Materia:** Programación 1.

**Profesor:** Prof. Nicolás Quirós

**Tutor:** Francisco Quarño

**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.

#### Introducción.

Un Árbol es un tipo de grafo ¿Qué es un grafo? Es una especie de mapa que conecta y relaciona cosas: objetos, lugares, ideas, datos varios etc. Claro es más abstracto, y eso lo hace aplicable a muchas cosas. Los grafos se utilizan por ejemplo en redes sociales donde cada persona es un nodo (un vértice) y se conecta a través de enlaces (una línea imaginaria que flota por el aire) con otras personas, o como el ejemplo del mapa, las ciudades o puntos importantes como estaciones de tren o la plaza principal pueden comprenderse como nodos y las calles o avenidas son las aristas (los enlaces) que las unen entre sí.

#### Marco Teórico.

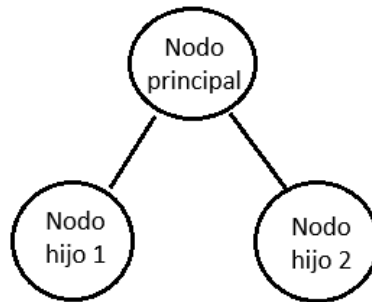
Pero volvamos al árbol, (que es un tipo especial de grafo).

Un árbol de búsqueda binario es una estructura de datos jerárquica, es decir una manera de ordenar y organizar datos para que realizar operaciones sobre ellos sea más fácil y eficiente. Como se mencionó anteriormente, los árboles tienen nodos, cada uno de esos nodos (dato, o con junto de datos) pueden tener (o no) además dos hijos.

La estructura es jerárquica porque:

- Tiene una raíz, el nodo principal.
- De su raíz se desprenden otros dos nodos (o uno o ninguno).

- Los nodos se disponen: los menores a la izquierda y los mayores a la derecha.



Este tipo de árbol permite realizar operaciones como **búsqueda, inserción y eliminación** de manera eficiente con una complejidad promedio de  $O(\log n)$ , lo que lo convierte en una herramienta clave en programación y optimización de datos.

### Caso Práctico.

Se desarrollo un código para crear un árbol y almacenar los datos de una lista determinada.

A continuación, se detalla el código donde:

- Utilizamos una lista para representar nodos: [valor, izq. Der.]
- Implementamos funciones para **insertar**, **buscar**, e **imprimir** el árbol.
- Aplicamos **recursividad** para recorrer y modificar el árbol de manera dinámica.

```

• # Lista de 3 elementos: [nodo, subárbol_izquierdo,
• subárbol_derecho]
•
• [7, [3, [1, None, None], [5, None, None]], [9, [8, None, None],
• [10, None, None]]
•
• # Insertar un valor en el árbol
• def insertar(arbol, valor):
•     if arbol is None:
•         return [valor, None, None] # Crea un nodo nuevo si el
arbol está vacío
•     if valor < arbol[0]:
•         arbol[1] = insertar(arbol[1], valor) # Inserta en el
subarbol izquierdo
•     elif valor > arbol[0]:

```

```

•         arbol[2] = insertar(arbol[2], valor) # Inserta en el
subarbol derecho
•         return arbol
•
•
• # Busca un valor en el árbol
• def buscar(arbol, valor):
•     if arbol is None:
•         return False # Si no se encontró devuelve false
•     if valor == arbol[0]: # Si se encontró devuelve true
•         return True
•     elif valor < arbol[0]:
•         return buscar(arbol[1], valor) # Si en no se encontró busca
en subárbol izquierdo
•     else:
•         return buscar(arbol[2], valor) # Si en no se encontró
busca en subárbol derecho
•
• # Imprime el árbol en orden
• def imprimir_inorden(arbol):
•     if arbol is not None:
•         imprimir_inorden(arbol[1]) # hacia la izquierda
•         print(arbol[0], end=' ') # raíz
•         imprimir_inorden(arbol[2]) # hacia la derecha
•
• # Empieza el programa.
•
• arbol = None
•
• # Inserta los valores
• for numero in [7, 3, 9, 1, 5, 8, 10]:
•     arbol = insertar(arbol, numero)
•
• # Imprime el árbol en orden
• print("Valores en orden:")
• imprimir_inorden(arbol)
• print()
•
• # Buscar un valor determinado
• valor = 5
• if buscar(arbol, valor):
•     print(f"El valor {valor} está en el árbol.")
• else:
•     print(f"El valor {valor} NO está en el árbol.")
•
• # Resultado esperado
•
• # Valores en orden:
• # 1 3 5 7 8 9 10
• # El valor 5 está en el árbol.

```

Este ejemplo demuestra cómo un árbol binario permite organizar datos de forma eficiente para búsquedas y ordenamientos.

### Metodología utilizada.

- Se desarrolló en el Lenguaje Python, versión Python 3.11.9
- Entorno de desarrollo: Visual Studio Code.
- **Investigación previa:** Se analizó el concepto de árboles binarios, su implementación tradicional con clases y las alternativas sin orientación a objetos.
- **Diseño de la representación:** Se definió una estructura basada en listas, donde cada nodo es una lista de tres elementos: [valor, subárbol izquierdo, subárbol derecho].
- **Implementación del código:** Se desarrollaron funciones para inserción, búsqueda y recorrido del árbol, asegurando claridad y eficiencia.
- **Pruebas y validación:** Se ejecutaron casos de prueba con distintos conjuntos de datos, verificando la correcta inserción y búsqueda de elementos.

### Resultados obtenidos.

- El código funcionó correctamente.
- La inserción y búsqueda de elementos se ejecutaron correctamente, respetando la estructura jerárquica.
- Se comprobó que la eficiencia es aceptable para conjuntos de datos pequeños y medianos

### Conclusión:

Este trabajo permitió comprender la estructura y funcionamiento de los árboles de búsqueda binaria en Python sin utilizar clases, aplicando un enfoque basado en listas. A lo largo del proceso, se evidenció que esta representación por listas, si bien más sencilla e intuitiva, puede presentar desafíos en términos de manipulación y eficiencia.

Para futuras mejoras, se podrían explorar algoritmos de balanceo como **AVL o rojo-negro**, así como integrar mecanismos para **optimizar búsquedas y recorridos**.