

Trabajo Práctico – Árboles y Árboles Binarios

ALUMNOS:	David Gutierrez	davidgutierrez02027@gmail.com	COM15
	Mauro Valitutto	Maurovalitutto@hotmail.com	COM22
MATERIA:	Programación I		
PROFESOR:			
FECHA DE ENTREGA:	09/06/2025		

Introducción General

En informática, los árboles son estructuras de datos fundamentales que permiten representar relaciones jerárquicas de forma eficiente. A diferencia de otras estructuras lineales como listas o arreglos, los árboles reflejan relaciones del tipo padre-hijo, siendo ideales para modelar sistemas de organización jerárquica, toma de decisiones y búsqueda estructurada.

Un árbol se representa como una estructura invertida que parte de un nodo raíz y se ramifica en nodos hijos, facilitando operaciones como búsquedas rápidas, ordenamientos eficientes y análisis jerárquico de información.

¿Por qué son importantes los árboles?

- **Versatilidad:** Se utilizan en múltiples áreas, desde sistemas de archivos hasta inteligencia artificial.
 - **Eficiencia:** Permiten reducir la complejidad de operaciones como búsqueda, inserción y eliminación. Mejor uso de memoria.
 - **Aplicaciones prácticas:**
 - Árboles de decisión en IA.
 - Representación jerárquica de carpetas.
 - Compresión de datos y parsers en compiladores.
 - Índices en bases de datos (como los árboles B y B+).
-

Componentes Fundamentales

- **Nodo:** Contiene un dato y conexiones a otros nodos.
 - **Raíz** Nodo principal del árbol.
 - **Nodo** Elemento del árbol con un valor.
 - **Padre** Nodo que tiene hijos.
 - **Hijo** Nodo descendiente de otro nodo.
 - **Hoja** Nodo sin hijos.
 - **Subárbol** Árbol que parte desde un nodo cualquiera.
 - **Nivel** Profundidad del nodo desde la raíz.
 - **Altura** Longitud máxima desde la raíz hasta una hoja.
 - **Grado** Cantidad de hijos de un nodo.
-

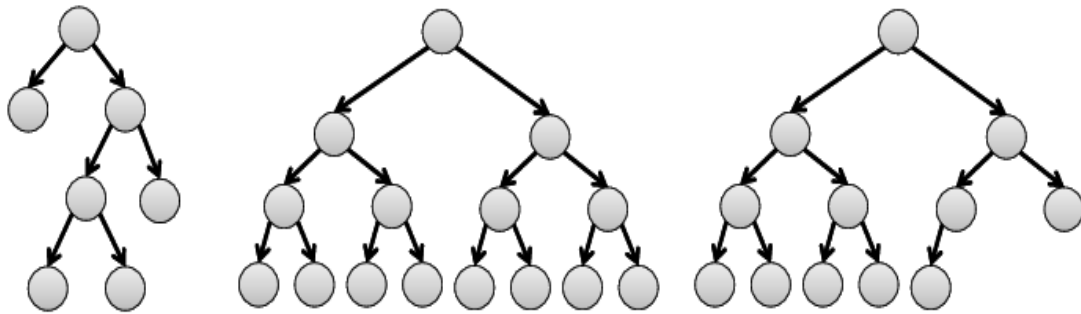
Clasificación de Árboles

Tipo de árbol	Descripción
Árbol Binario	Cada nodo tiene como máximo dos hijos.
Árbol Binario de Búsqueda (ABB)	Árbol binario con criterio de orden: $izq < nodo < der$.
Árbol AVL / Balanceado	ABB con balance automático para mantener eficiencia.
Árbol B / B+	Utilizados en bases de datos por su capacidad de múltiples hijos.

Propiedades de los Árboles

- Un árbol con n nodos tiene exactamente $n-1$ aristas.
- La altura de un árbol con un solo nodo es 0.
- **Árbol completo:** Todos los niveles están llenos, salvo posiblemente el último.
- **Árbol lleno:** Todos los nodos tienen 0 o 2 hijos.

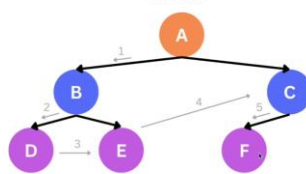
- **Árbol balanceado:** La diferencia de altura entre subárboles es de a lo sumo 1.



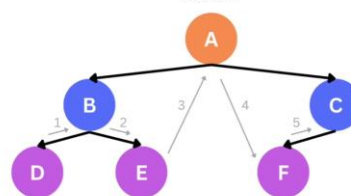
Recorridos de Árboles (Traversals)

Tipo de recorrido	Orden de visita
Preorden	Nodo → Izquierda → Derecha
Inorden	Izquierda → Nodo → Derecha
Postorden	Izquierda → Derecha → Nodo

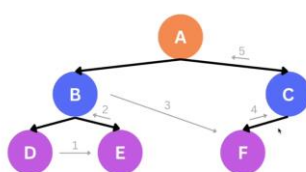
Recorrido de un árbol binario
Preorden



Recorrido de un árbol binario
Inorden



Recorrido de un árbol binario
Postorden



Estos recorridos permiten procesar los nodos en diferentes contextos, como copiar, evaluar o visualizar un árbol.

Árboles Binarios y Árboles Binarios de Búsqueda (ABB)

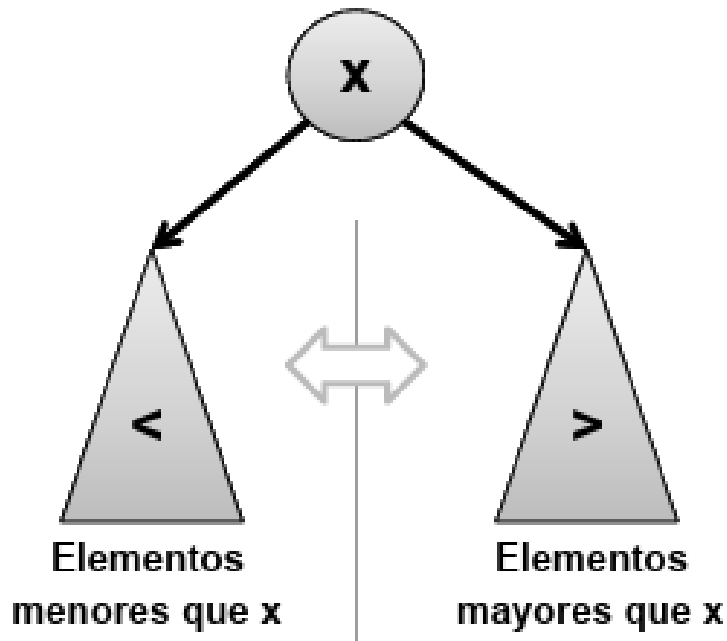
Un **árbol binario** restringe cada nodo a tener como máximo dos hijos: izquierdo y derecho.

El **Árbol Binario de Búsqueda (ABB):** es un árbol binario cuyos nodos almacenan elementos comparables mediante \leq y donde todo nodo cumple la propiedad de ordenación:

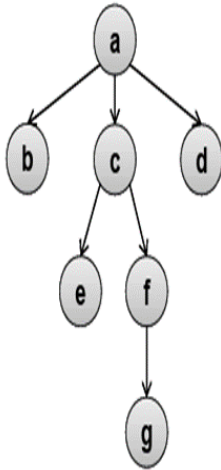
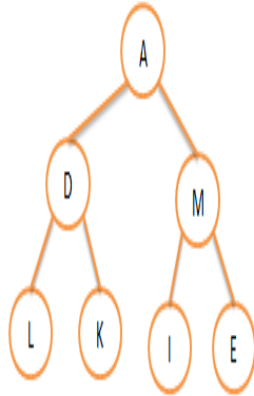
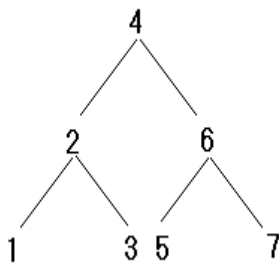
Propiedad de ordenación: Todo nodo es mayor que los nodos de su subárbol izquierdo, y menor que los nodos de su subárbol derecho.

- Subárbol izquierdo: valores menores que el nodo.
- Subárbol derecho: valores mayores que el nodo.

Esto permite búsquedas eficientes en promedio en tiempo logarítmico ($O(\log n)$), si el árbol está balanceado.



Cuadro Comparativo

Característica	Árbol General	Árbol Binario	ABB (Búsqueda)
Hijos por nodo	Ilimitados	Máximo 2	Máximo 2
Orden entre nodos	Arbitrario	Arbitrario	Ordenado
Eficiencia de búsqueda	Baja	Media	Alta
			

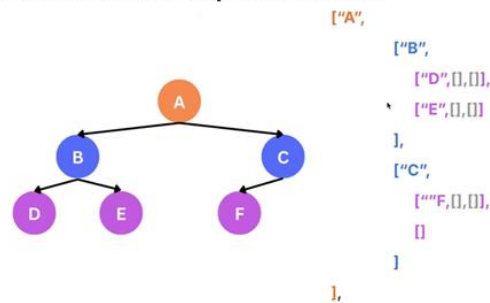
Representación de Árboles en Python

En Python, los árboles pueden representarse de varias formas:

- **Clases (más estructurado y flexible):**
Crear una clase Nodo o TreeNode que tenga atributos para almacenar el valor y enlaces a hijos (en árboles binarios, por ejemplo, izquierda y derecha).
- **Diccionarios:**
Guardar los nodos y sus enlaces en diccionarios, útil en ciertos tipos de árboles como árboles de búsqueda.
- **Listas (menos formal, más simple):**
Usar listas anidadas para representar árboles, especialmente árboles binarios.
Ejemplo: [valor, hijo_izquierdo, hijo_derecho].
- **Otros enfoques:**
 - Árboles con bibliotecas específicas (como anytree, binarytree para visualización, etc.)
 - Estructuras personalizadas combinando listas, clases y diccionarios.

Implementaciones Típicas	Nuestro Enfoque con Listas
Utilizan punteros y clases para crear estructuras de árbol, lo que requiere definir objetos y métodos específicos. Aunque son potentes, pueden ser complejas para quienes aprenden.	Usamos listas en Python que representan subárboles, simplificando la lógica del árbol y eliminando la complejidad sintáctica, permitiendo enfocarse en la estructura sin interacción con programación orientada a objetos.

Estructura de Representación



Cada nodo es una lista con tres elementos: [valor, subárbol izquierdo, subárbol derecho]. Esta estructura, aunque menos robusta que las clases, es didáctica y clara para representar árboles binarios.

Características ideales de la lista de entrada para construir un buen ABB

Característica

Desordenada o semi-aleatoria

Sin duplicados

Tipo de datos consistente

Cantidad razonable

Con un valor "central" al principio (opcional)

¿Por qué es importante?

Evita que el árbol se degrade en una lista lineal (desbalanceado). Mejora el rendimiento de búsquedas e inserciones.

Un ABB clásico no maneja valores duplicados (cada valor debe ser único). Esto simplifica la lógica y mantiene la propiedad del ABB.

Todos los elementos deben ser del mismo tipo (por ejemplo, solo números). Comparaciones entre tipos diferentes pueden fallar.

Demasiados valores sin un algoritmo de balanceo pueden generar un árbol muy profundo y desequilibrado.

Si el primer valor insertado es cercano al promedio, puede ayudar a generar un árbol más equilibrado naturalmente. Ej: [50, 30, 70, 20, 40, 60, 80].

Caso Práctico: Implementación de un Árbol Binario de Búsqueda (ABB)

Objetivo

Implementar en Python un árbol binario de búsqueda utilizando **listas anidadas** que permita:

- Insertar valores respetando el orden del ABB.
- Realizar un recorrido **inorden** para mostrar los valores ordenados.
- Buscar un valor y mostrar los pasos realizados.

Metodología

1. **Análisis del problema:** Se desea ordenar y buscar números de forma eficiente.
2. **Estructura seleccionada:** Árbol binario de búsqueda representado con listas anidadas.
3. **Diseño e implementación:** Funciones para insertar, buscar, recorrer e imprimir el árbol.
4. **Pruebas:** Ingreso de valores, búsqueda interactiva y visualización jerárquica.

Implementación en Python con listas

```
def insertar(arbol, valor):  
    if not arbol:  
        return [valor, [], []]  
    if valor < arbol[0]:  
        arbol[1] = insertar(arbol[1], valor)  
    else:  
        arbol[2] = insertar(arbol[2], valor)  
    return arbol  
  
def buscar_con_pasos(arbol, valor, pasos=0):  
    if not arbol:  
        return False, pasos  
    pasos += 1  
    if valor == arbol[0]:  
        return True, pasos  
    elif valor < arbol[0]:  
        return buscar_con_pasos(arbol[1], valor, pasos)  
    else:  
        return buscar_con_pasos(arbol[2], valor, pasos)
```



```
        return buscar_con_pasos(arbol[2], valor, pasos)

def mostrar_arbol(arbol, prefijo="", es_izq=True):
    if arbol:
        print(prefijo + ("├─ " if es_izq else "└─ ") +
str(arbol[0]))
        if arbol[1] or arbol[2]:
            if arbol[1]:
                mostrar_arbol(arbol[1], prefijo + ("├─ " if es_izq
else "   "), True)
            else:
                print(prefijo + ("├─ " if es_izq else "   ") + "├─
(vacio)")
            if arbol[2]:
                mostrar_arbol(arbol[2], prefijo + ("├─ " if es_izq
else "   "), False)
            else:
                print(prefijo + ("├─ " if es_izq else "   ") + "└─
(vacio)")

# ---- PROGRAMA PRINCIPAL ----
print("Ingrese números enteros para construir el árbol.")
print("Ingrese 0 para finalizar la carga.\n")
arbol = []

while True:
    try:
        numero = int(input("Ingrese un número (0 para terminar): "))
        if numero == 0:
            break
        arbol = insertar(arbol, numero)
    except ValueError:
        print("Por favor, ingrese un número entero válido.")

print("\n Ahora puede buscar un número en el árbol.")
try:
    valor_búsqueda = int(input("Ingrese un número a buscar: "))
    encontrado, pasos = buscar_con_pasos(arbol, valor_búsqueda)
    if encontrado:
        print(f"El número {valor_búsqueda} existe en el árbol. Pasos
dados: {pasos}")
```

```

    else:
        print(f"El número {valor_busqueda} NO existe en el árbol.
Pasos dados: {pasos}")
except ValueError:
    print("Entrada inválida.")

print("\n\n Representación visual del árbol:")
if arbol:
    print(arbol[0])
    if arbol[1] or arbol[2]:
        if arbol[1]:
            mostrar_arbol(arbol[1], "", True)
        else:
            print("— (vacio)")
        if arbol[2]:
            mostrar_arbol(arbol[2], "", False)
        else:
            print("— (vacio)")

```

En esta tabla se muestra cuántos pasos se requieren para encontrar ciertos valores, comparando una búsqueda en un Árbol Binario de Búsqueda (ABB) frente a una búsqueda secuencial (lineal (for , while)):



Valor buscado	Pasos en ABB	Pasos en lista desordenada (peor caso)
3	4 pasos	15 pasos
7	4 pasos	15 pasos
10	2 pasos	15 pasos
13	4 pasos	15 pasos
20	1 paso	15 pasos
25	3 pasos	15 pasos
37	4 pasos	15 pasos

A medida que aumenta la cantidad de datos, el ABB permite encontrar elementos con menos comparaciones que una lista común, lo que mejora la eficiencia significativamente.

Conclusiones

Los árboles son estructuras de datos esenciales que se adaptan a una gran variedad de problemas computacionales. En particular, los árboles binarios de búsqueda ofrecen una solución elegante para organizar y recuperar datos de manera eficiente. Comparativa de eficiencia: ABB vs Lista Desordenada

A través de este trabajo se integró la teoría con la práctica, utilizando listas anidadas en Python para construir un ABB funcional. Este enfoque no solo permitió comprender los conceptos fundamentales, sino también experimentar con su implementación real.

Bibliografía y Material de Referencia

- Material del Campus Virtual UTN ([Apunte teórico sobre árboles](#))
- [Documentación oficial de Python](#)
- Estructuras de Datos y Algoritmos Tema 4: Árboles. César Vaca Rodríguez, Dpto. de Informática, UVA