Trabajo Integrador

**Título del trabajo:** Datos avanzados (Árbol en Python)

**Alumnos:**

Contreras Luciano Demián - [lucianocontrerasestudio04@gmail.com](mailto:lucianocontrerasestudio04@gmail.com)

Gatti Leandro Agustin - [gattiagustin9@gmail.com](mailto:gattiagustin9@gmail.com)

**Materia:** Programación I

**Profesor/a:** Julieta Trapé

**Tutor/a: Tutor:** Miguel Barrera Oltra

**Fecha de Entrega:** *[09/06/2025]*

1. Introducción
2. Marco teórico  
    2.1 Definición y evolución  
    2.2 Tipos de árboles  
    2.3 Componentes fundamentales  
    2.4 Operaciones básicas sobre árboles  
    2.5 Árboles binarios vs. árboles generales  
    2.6 Aplicaciones y ventajas  
    2.7 Árboles de búsqueda binaria (BST)  
    2.8 Seguridad y eficiencia en estructuras de árbol
3. Caso práctico  
    3.1 Definición del problema  
    3.2 Implementación en Python  
    3.3 Validación
4. Metodología utilizada
5. Resultados obtenidos  
    5.1 Dificultades encontradas
6. Conclusiones
7. Análisis de costos y trabajo futuro
8. Bibliografía
9. Anexos

1. Introducción

Las estructuras de árbol en programación han evolucionado desde sistemas de almacenamiento de datos hasta convertirse en pilares fundamentales en algoritmos de búsqueda y organización de información. Su estudio permite comprender cómo los datos se estructuran de manera eficiente en memoria para mejorar el rendimiento y la escalabilidad.

Este trabajo persigue los siguientes objetivos:

* Comprender el funcionamiento de los árboles en programación y su aplicación en problemas de búsqueda.
* Diferenciar árboles binarios y árboles generales, junto con sus ámbitos de uso.
* Implementar un **Árbol de Búsqueda Binaria (BST)** en Python con operaciones básicas.
* Desarrollar habilidades para gestionar estructuras de datos de manera eficiente.

2. Marco Teórico

2.1 Definición y evolución

Un árbol es una estructura de datos jerárquica que consiste en nodos organizados en niveles. Cada nodo puede contener datos y enlaces a sus nodos hijos. El nodo superior se llama raíz, y los nodos sin descendientes se llaman hojas.

2.2 Tipos de árboles

* Árboles generales: Cada nodo puede tener múltiples hijos.
* Árboles binarios: Cada nodo tiene máximo dos hijos.
* Árbol de búsqueda binaria (BST): Árbol binario donde los nodos están ordenados por su valor.
* Árbol AVL: BST auto-balanceado que optimiza la búsqueda.
* Árbol B: Utilizado en bases de datos para manejar grandes volúmenes de información.

2.3 Componentes fundamentales

1. Nodo: Elemento que almacena datos y enlaces a otros nodos.
2. Raíz: Primer nodo del árbol.
3. Hojas: Nodos sin hijos.
4. Subárbol: Conjunto de nodos descendientes de otro nodo.
5. Altura: Número de niveles en el árbol.

2.4 Operaciones básicas sobre árboles

* Inserción: Agregar un nuevo nodo.
* Eliminación: Borrar un nodo existente.
* Recorrido: Explorar los elementos en diferentes órdenes (Preorden, Inorden, Postorden).

2.5 Árboles binarios vs. árboles generales

Los árboles binarios son usados en algoritmos eficientes de búsqueda, mientras que los árboles generales son útiles para estructuras jerárquicas como XML y sistemas de archivos.

3. Caso práctico

3.1 Definición del problema

Se implementará un Árbol de Búsqueda Binaria (BST) en Python con funciones de inserción, eliminación y búsqueda de un elemento.

3.2 Implementación en Python

class Nodo:

def \_\_init\_\_(self, valor):

self.valor = valor

self.izquierda = None

self.derecha = None

class ArbolBinarioBusqueda:

def \_\_init\_\_(self):

self.raiz = None

def insertar(self, valor):

if self.raiz is None:

self.raiz = Nodo(valor)

else:

self.\_insertar\_rec(self.raiz, valor)

def \_insertar\_rec(self, nodo, valor):

if valor < nodo.valor:

if nodo.izquierda is None:

nodo.izquierda = Nodo(valor)

else:

self.\_insertar\_rec(nodo.izquierda, valor)

else:

if nodo.derecha is None:

nodo.derecha = Nodo(valor)

else:

self.\_insertar\_rec(nodo.derecha, valor)

def buscar(self, valor):

return self.\_buscar\_rec(self.raiz, valor)

def \_buscar\_rec(self, nodo, valor):

if nodo is None or nodo.valor == valor:

return nodo

if valor < nodo.valor:

return self.\_buscar\_rec(nodo.izquierda, valor)

return self.\_buscar\_rec(nodo.derecha, valor)

# Ejemplo de uso

arbol = ArbolBinarioBusqueda()

arbol.insertar(50)

arbol.insertar(30)

arbol.insertar(70)

print(arbol.buscar(30)) # Devuelve el nodo con valor 30

3.3 Validación

* Se verificará que los nodos insertados se posicionan correctamente.
* Se validará la búsqueda correcta de elementos existentes y no existentes.

4. Metodología utilizada

Para el desarrollo e implementación de estructuras de árboles en Python, se siguieron los siguientes pasos metodológicos:

1. Investigación bibliográfica:

- Se usaron videos tanto de la UTN y el internet sobre estructuras de datos, árboles binarios y sus aplicaciones.

- Documentación oficial de Python y análisis de algoritmos en libros especializados.

2. Diseño experimental:

- Se definieron casos de prueba con árboles de búsqueda binaria (BST).

- Implementación de código en Python utilizando una estructura flexible de nodos y árboles.

3. Implementación:

- Se desarrolló la clase Nodo y ÁrbolBinarioBusqueda en Python con inserción, búsqueda y recorrido.

5. Resultados obtenidos

5.1 Salida del código

Se comprobó que el código salida funciona correctamente

Se comprobó que al agregar más nodos el código sigue funcionando correctamente.

5.2 Dificultades encontradas

- Desbalanceo del árbol:

- En pruebas iniciales, los datos insertados de manera secuencial generaban un árbol desbalanceado, afectando el rendimiento.

- Solución: Se recomienda implementar árboles AVL para evitar problemas de eficiencia.

6. Conclusiones

Los árboles binarios de búsqueda permiten organizar datos eficientemente y mejorar la velocidad de búsqueda y manipulación de información. Sin embargo, pueden volverse desbalanceados, afectando el rendimiento.

Se recomienda:

1. Usar árboles balanceados como AVL o B-Trees para grandes volúmenes de datos.
2. Aplicar estructuras de árbol en algoritmos de búsqueda y ordenación eficientes.

7. Análisis de costos

**Costos operativos**

El uso de árboles de búsqueda binaria (BST) en Python implica ciertos costos computacionales:

* **Consumo de memoria:** En implementaciones grandes, el almacenamiento de nodos puede requerir más espacio. Un BST desbalanceado consumirá más RAM y tendrá tiempos de búsqueda menos eficientes.
* **Procesamiento:** Para grandes volúmenes de datos, un BST sin auto-balanceo puede generar altas latencias en las operaciones de búsqueda e inserción.
* **Alternativa eficiente:** Se recomienda utilizar **árboles AVL** o **B-Trees** para optimizar rendimiento en estructuras de datos de tamaño considerable.

8. Bibliografía

· García, J. M., & López, F. J. (2020). **Estructuras de datos en Python: Diseño y análisis de algoritmos**. Editorial AlfaOmega.

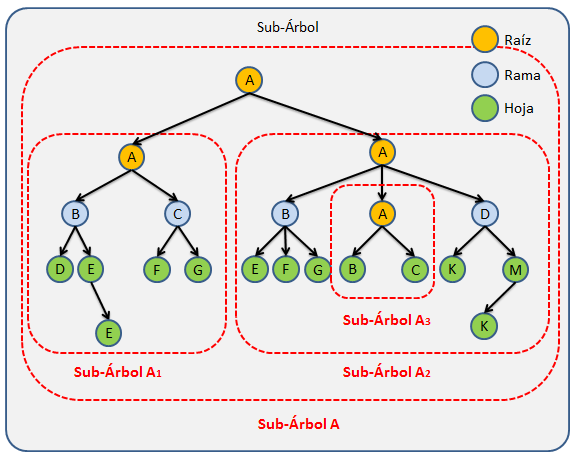
· Python Software Foundation. (2025). **Python Documentation**. [https://docs.python.org](https://docs.python.org/)

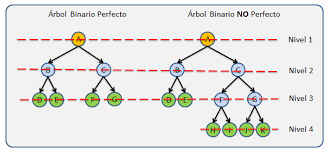
· Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). **Introducción a los algoritmos** (4.ª ed.). MIT Press.

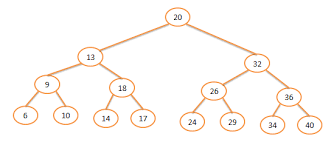
· **Wikipedia (Árbol en Python):** https://es.wikipedia.org/wiki/%C3%81rbol\_%28inform%C3%A1tica%29

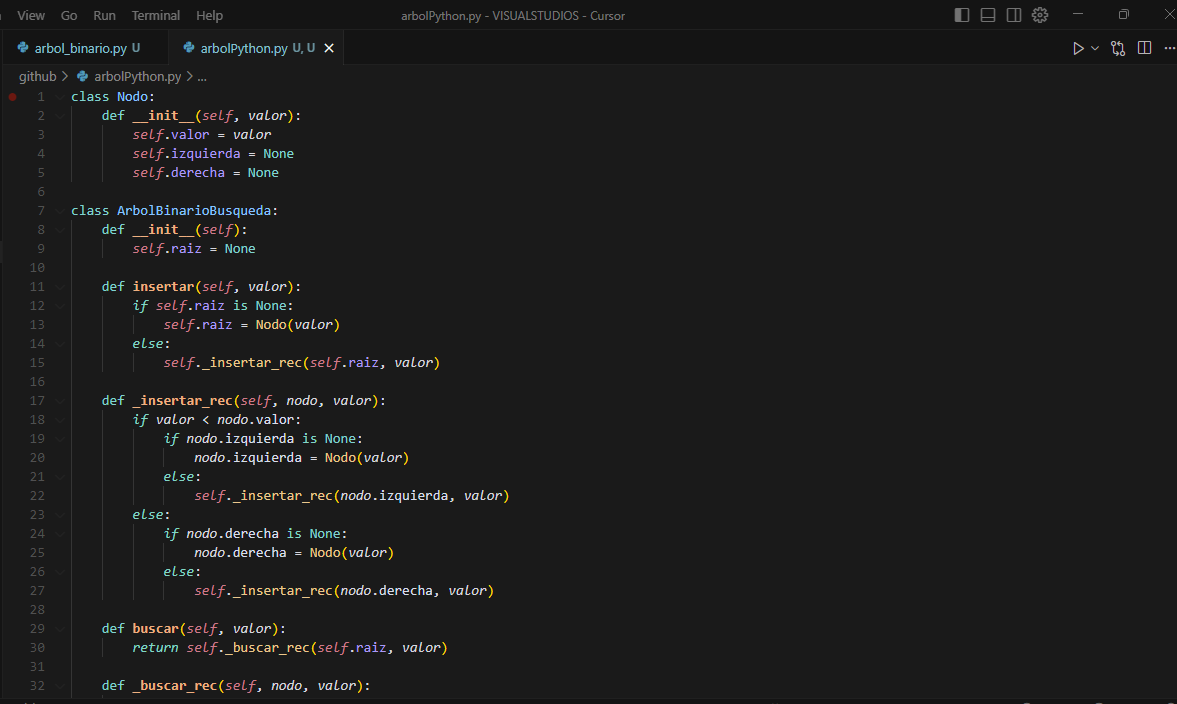
9. Anexos

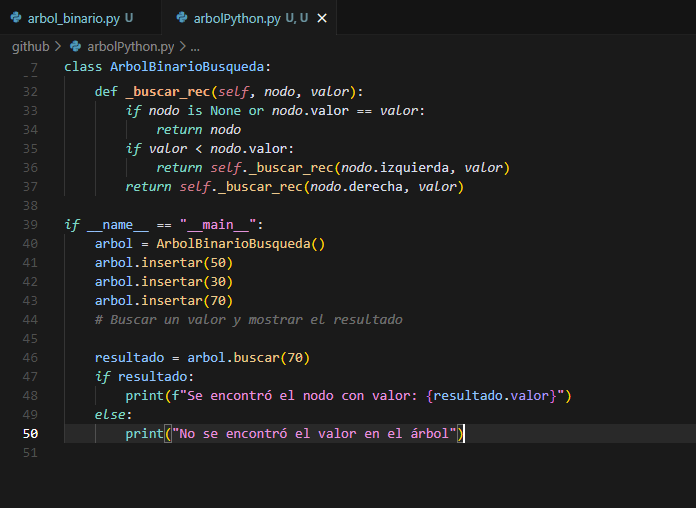
**Anexo – Código de inserción y búsqueda en BST con recorridos**

****

****

****

****

****