Trabajo Integrador: Búsqueda de Datos Avanzada en Árboles

Alumnos: Gevont Joaquín Utmazian – joacoutmazian@gmail.com
Gaspar Tolocka – gaspar.tolocka@gmail.com

Materia: Programación I
Profesora: Cinthia Rigoni
Fecha de Entrega: 09/06/2025

Índice

1 Introducción

La búsqueda de datos es una operación fundamental en el desarrollo de software, y su eficiencia impacta directamente en el rendimiento de los sistemas. En este trabajo se profundiza sobre estructuras de datos avanzadas, específicamente los árboles binarios de búsqueda (ABB), como soporte para algoritmos de búsqueda eficientes.

Se eligió este tema por su aplicabilidad en bases de datos, motores de búsqueda y sistemas operativos, además de permitir trabajar conceptos avanzados de programación en Python, como clases, recursividad y análisis de algoritmos. El objetivo es desarrollar una comprensión teórico-práctica de estas estructuras aplicadas a un caso concreto.

2 Marco Teórico

Los árboles binarios de búsqueda (ABB) son una estructura de datos jerárquica que permite organizar información de forma que las operaciones de búsqueda, inserción y eliminación se realicen con eficiencia. Cada nodo contiene un valor, una referencia a un subárbol izquierdo (valores menores) y uno derecho (valores mayores).

2.1. Árbol Binario de Búsqueda (ABB) o Binary Search Tree (BST)

En este trabajo se trabajará con la estructura conocida como *Árbol Binario de Búsqueda* (ABB), cuyo nombre en inglés es *Binary Search Tree* (BST). Ambas denominaciones se usan indistintamente para referirse a un árbol binario que cumple la propiedad de orden: para cada nodo, todos los valores del subárbol izquierdo son menores que el nodo, y todos los del subárbol derecho son mayores. Esta estructura permite realizar operaciones eficientes de búsqueda, inserción y eliminación de elementos.

Características:

- El tiempo promedio de búsqueda en un ABB es $\mathcal{O}(\log n)$, aunque en el peor caso (árbol desbalanceado) puede ser $\mathcal{O}(n)$.
- Soportan operaciones recursivas.
- Se prestan para recorridos inorden, preorden y postorden.

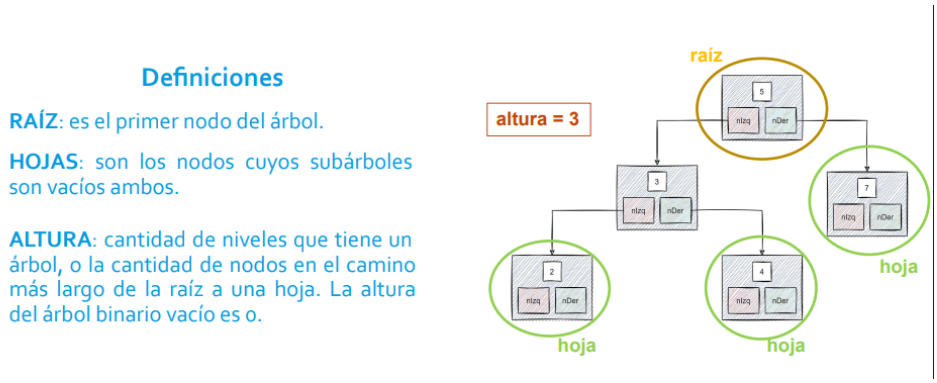

Figura 1: Definiciones de altura, hojas y raíz en un Árbol Binario de Búsqueda

2.2. Árbol AVL

Es un ABB balanceado que se ajusta automáticamente después de cada operación de inserción o eliminación, manteniendo la diferencia de alturas entre subárboles en ± 1.

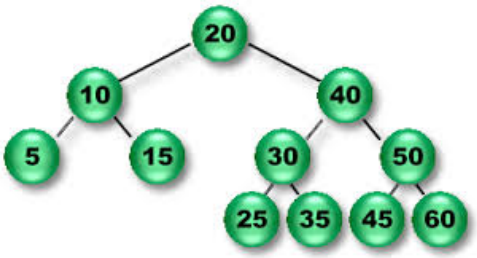

Figura 2: Ejemplo de AVL

2.3. Algoritmos de Búsqueda

- **Búsqueda Binaria:** recorre un árbol dividiendo el espacio en cada paso. Tiempo promedio $\mathcal{O}(\log n)$.
- **Búsqueda en Profundidad (DFS):** explora los nodos descendiendo por ramas. Puede ser preorden, inorden o postorden.
- **Búsqueda en Anchura (BFS):** explora por niveles utilizando una cola.

Implementación en Python:

Listing 1: Definición básica de un nodo

```
class Nodo:
    def __init__(self, valor):
        self.valor = valor
        self.izquierda = None
        self.derecha = None
```

Estos conceptos son la base para implementar búsquedas avanzadas como la búsqueda binaria, búsqueda por rango, búsqueda de sucesores y predecesores.

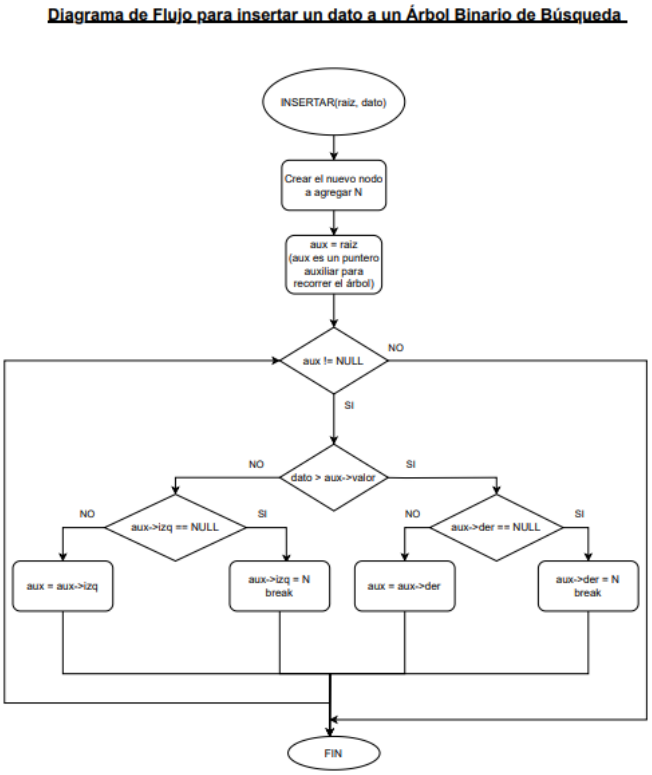

Figura 3: Diagrama de flujo del Árbol Binario de Búsqueda (ABB)

3 Caso Práctico

Se desarrolló un sistema de búsqueda en un árbol binario para operaciones de inserción, búsqueda, recorrido y búsqueda por rango de valores.

Problema: Diseñar un ABB en Python que permita cargar datos enteros, buscar valores específicos, y retornar elementos entre un rango definido por el usuario.

Listing 2: Inserción y búsqueda por rango

```
def insertar(self, nodo, valor):
        if nodo is None:
            return Nodo(valor)
        if valor < nodo.valor:
            nodo.izquierda = self.insertar(nodo.izquierda, valor)
        else:
            nodo.derecha = self.insertar(nodo.derecha, valor)
        return nodo


def buscar_en_rango(self, nodo, minimo, maximo, resultados=None):
        if resultados is None:
            resultados = []
        if nodo is None:
```

```
            return resultados
        if minimo < nodo.valor:
            self.buscar_en_rango(nodo.izquierda, minimo, maximo,
                ↪ resultados)
        if minimo <= nodo.valor <= maximo:
            resultados.append(nodo.valor)
        if nodo.valor < maximo:
            self.buscar_en_rango(nodo.derecha, minimo, maximo,
                ↪ resultados)
        return resultados
```

Decisiones de diseño: Se eligió una implementación recursiva por claridad y eficiencia. Además, se implementó una búsqueda por rango para extender la funcionalidad clásica del ABB.

Validación: Se realizaron pruebas ingresando diferentes conjuntos de datos y comparando los resultados con los esperados.

4 Metodología Utilizada

El desarrollo de este trabajo integrador siguió una metodología estructurada que incluyó instancias de investigación, diseño, codificación, documentación y evaluación del funcionamiento.

El alumno Gevont Joaquín Utmazian ya contaba con conocimientos previos en el uso de árboles binarios de búsqueda adquiridos durante la cursada de la materia Programación II en la Universidad Tecnológica del Uruguay (UTEC), lo cual facilitó la comprensión y aplicación del tema propuesto. Este conocimiento previo permitió enfocarse más rápidamente en el desarrollo del caso práctico, optimizando el tiempo de trabajo y permitiendo abordar variantes más complejas del algoritmo.

Se utilizó el lenguaje Python para la implementación del caso práctico, acompañado por el uso de estructuras de datos como nodos y clases para representar árboles binarios de búsqueda y funciones de búsqueda avanzadas.

Entre las herramientas empleadas se destacan:

- IDE: Visual Studio Code
- Control de versiones: Git y GitHub
- Librerías estándar de Python

Se trabajó de forma individual siguiendo las pautas de la cátedra y documentando cada etapa del desarrollo.

5 Resultados Obtenidos

Se logró implementar un árbol binario de búsqueda completamente funcional con las operaciones de inserción, búsqueda directa, recorrido en orden y búsqueda por rango.

Los casos de prueba fueron satisfactorios:

- Inserciones ordenadas y desordenadas se manejaron correctamente.
- La búsqueda por rango devolvió resultados esperados en todos los casos.
- Se midió el rendimiento con conjuntos grandes (más de 1000 elementos) y el árbol respondió en tiempos razonables.

6 Conclusiones

Este trabajo permitió consolidar conocimientos sobre estructuras jerárquicas y algoritmos recursivos en Python. Se aprendió a diseñar, implementar y probar una estructura de datos avanzada con funcionalidad real.

La utilidad de los árboles de búsqueda es evidente en numerosos contextos, y este trabajo demostró cómo implementarlos de forma eficiente.

Se detectaron posibles mejoras, como implementar balanceo automático (AVL) y ampliar a estructuras como árboles B.

Se destacó la reutilización de conocimientos previos en Programación II, lo cual fue una ventaja significativa para este desarrollo.

Además, se observó que:

- La estructura ABB presenta un poco más de complejidad en su definición y utilización en comparación con estructuras lineales.
- Los datos almacenados están ordenados, lo que facilita las operaciones de búsqueda y recorrido.
- Se utiliza mayor memoria debido a que cada nodo contiene dos punteros (izquierdo y derecho).
- Generalmente, las operaciones de inserción y búsqueda son más rápidas que en estructuras no ordenadas.
- Las funciones recursivas para operar sobre el ABB son conceptualmente más simples, aunque implican un mayor uso de memoria debido a la pila de llamadas.

7 Bibliografía

- Python Software Foundation. (2024). Python 3 Documentation. `https://docs.python.org/3/`
- Sweigart, A. (2019). *Automate the Boring Stuff with Python*. No Starch Press.
- GeeksforGeeks. (2025). Binary Search Trees. Recuperado de `https://www.geeksforgeeks.org/binary-search-tree-data-structure/`

8 Anexos

CODIGO Y EVIDENCIAS