PROGRAMACIÓN

Trabajo Integrador N° 1

Universidad Tecnológica Nacional

Estudiante: Tomás Yovino.

Docente: Ariel Enferrel.

Curso: Primera instancia integradora – Programación.

Año: 2025.

SELECCIÓN DEL TEMA

DESCRIPCIÓN DEL TEMA

Tema: Estructuras de datos avanzadas: árboles.

Un árbol es una estructura de datos jerárquica compuesta por nodos conectados por aristas, donde existe un nodo raíz sin padre y todos los demás nodos tienen exactamente un padre. Cada nodo puede tener cero o más hijos, formando una estructura en forma de "ramas" que refleja relaciones de dependencia y organización no lineal.

MOTIVACIÓN Y JUSTIFICACIÓN

Eficiencia y flexibilidad: Los árboles permiten búsquedas, inserciones y eliminaciones con complejidad logarítmica en estructuras balanceadas, frente a la linealidad de listar o arreglos desordenados.

Aplicaciones críticas:

- Bases de datos y sistemas de archivos emplean variantes como B-trees y B+-trees para índices y organización en disco.
- Compiladores utilizan árboles de sintaxis abstracta (AST) para el análisis y optimización de código.
- Motores de búsqueda y AI usan árboles de decisión y estructuras especiales (LD-trees, Octrees).

Variedad de variantes: Desde los árboles binarios de búsqueda (BST) hasta estructuras balanceadas (AVL, Red-Black) y árboles especializados (Segment Tree, Fenwick Tree), cada variante optimiza distintos aspectos (memoria, velocidad o uso en memoria secundaria).

Relevancia académica y profesional: Comprender árboles y sus algoritmos es fundamental en asignaturas de Algoritmos y Estructuras de Datos, así como en el diseño de arquitecturas de software de alta performance.

OBJETIVOS DE LA INVESTIGACIÓN

Objetivo general: Analizar y comparar las principales variantes de árboles como estructuras de datos avanzadas, evaluando su diseño, algoritmos fundamentales y aplicaciones prácticas.

Objetivos específicos:

- Definir y clasificar las propiedades básicas de los árboles (terminología, recorridos, medidas de altura y profundidad).
- Implementar un Árbol Binario de Búsqueda (BST) en Python, incluyendo inserción, búsqueda, eliminación y recorridos.
- Desarrollar la versión balanceada AVL, incorporando rotaciones y mantenimiento del factor de balance.
- Medir y comparar empíricamente la eficiencia de BST vs AVL en inserción y búsqueda sobre distintos tamaños de datos.
- Investigar dos variantes avanzadas (por ejemplo, Red-Black Tree y Segment Tree), describiendo su estructura, algoritmos y casos de uso.
- Aplicar una de las implementaciones en un caso práctico real (índice de archivos o simulación de base de datos pequeña) para demostrar ventajas de balanceo.
- Documentar todo el proceso y presentar los resultados en un repositorio Git con código, pruebas y un breve video tutorial.

BIBLIOGRAFÍA ANOTADA

• Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C- (2009). Introduction to Algorithms (3ra ed.) MIT Press.

• Goodriich, M. T., Tamassia, R. & Goldwasser, M. H. (2014). Dta Structures & Algorithms in Python. Wiley.

• Documento de la cátedra: “Árboles”. Universidad Tecnológica Nacional. Programación I.

• “Binary search tree.” Wikipedia.

• “AVL tree.” Wikipedia.

• GeeksforGeeks. “AVL Tree | Set 1 (Insertion)”.

• TopCoder. “A Tutorial on Segment Trees”.

MARCO TEÓRICO

INTRODUCCIÓN A LOS ÁRBOLES

Un árbol es una estructura de datos no lineal cuyos elementos, llamados nodos, están conectados mediante aristar o ramas para reflejar relaciones jerárquicas. Existe un único nodo raíz, que no tiene padre, y a partir de éste se desarrollan uno o varios nodos hijos, pudiendo extenderse hasta varios niveles de profundidad. Esta estructura es ideal para modelar jerarquías en sistemas de archivos, bases de datos, análisis sintáctico de compiladores y muchas otras aplicaciones.

Términos básicos

- Raíz: nodo inicial del árbol (nivel 1).
- Padre e hijo: si un nodo A está conectado a un nodo B directamente por una rama y A está un nivel arriba, A es padre de B y B es hijo de A.
- Hermanos: nodos que comparten el mismo padre.
- Nodo interno o rama: nodo con al menos un hijo.
- Nodo hoja: nodo sin hijos, ubicado en los extremos de las ramas.

PROPIEDADES Y MEDIDAS

- Profundidad de un nodo: número de ramas desde la raíz hasta ese nodo (la raíz tiene profundidad 0).
- Nivel de un nodo: profundidad más uno; la raíz está en el nivel 1.
- Altura de un árbol: nivel máximo que alcanza el árbol.
- Longitud de camino: cantidad de ramas en la ruta entre dos nodos.
- Grado de un nodo: número de hijos que posee; el grado del árbol es el máximo grado entre sus nodos.
- Orden de un árbol: restricción preestablecida de cuántos hijos puede tener cada nodo (por ejemplo, orden 2 para árboles binarios).
- Peso: total de nodos en el árbol; indica el tamaño y consumo de memoria de la estructura.

RECORRIDOS EN ÁRBOLES BINARIOS

En un árbol binario cada nodo posee como máximo dos hijos, denominados “izquierdo” y “derecho”. Los tres recorridos fundamentales son:

Preorden:

- Visitar la raíz.
- Recorrer recursivamente el subárbol izquierdo.
- Recorrer recursivamente el subárbol derecho.

Inorden:

- Recorrer recursivamente el subárbol izquierdo.
- Visitar la raíz.
- Recorrer recursivamente el subárbol derecho.

Postorden:

- Recorrer recursivamente el subárbol izquierdo.
- Recorrer recursivamente el subárbol derecho.
- Visitar la raíz.

ÁRBOLES DE BÚSQUEDA BINARIA (BST)

Un BST es un árbol binario que cumple:

- Todos los valores del subárbol izquierdo de un nodo son menores que el valor del nodo.
- Todos los valores del subárbol derecho son mayores.

Operaciones básicas

Inserción: Se compara el valor a insertar con la raíz y se desciende recursivamente por izquierda o derecha hasta encontrar un lugar libre.

Búsqueda: Similar a la inserción, pero se detiene al encontrar el valor o al llegar a un nodo nulo.

Eliminación:

- Nodo hoja: se elimina directamente.
- Nodo con un solo hijo: se reemplaza el nodo por su hijo.
- Nodo con dos hijos: se busca el sucesor inorden (mínimo del subárbol derecho) o predecesor inorden (máximo del subárbol izquierdo), se intercambia su valor con el nodo a eliminar y luego se elimina ese sucesor/predecesor, garantizando la propiedad del BST.

En el peor caso (BST degenerado), inserción y búsqueda son O(n), mientras que el caso promedio son O(log n) si el árbol está balanceado.

ESTRUCTURAS BALANCEADAS "AVANZADAS"

Para evitar degeneración se emplean árboles que mantienen balance dinámico:

- AVL: mantiene el factor de balanceo (diferencia de alturas de subárboles izquierdo y derecho) en -1, 0 o 1 para cada nodo. Tras una inserción o eliminación, realiza rotaciones simples o dobles para restaurar el balance (O(1) por rotación), garantizando altura O(log n) en todo momento.
- Red-Black: impone propiedades de color (rojo/negro) en los nodos para asegurar que ningún camino raíz-hoja sea más de dos veces el más corto, logrando altura O(log n) con rotaciones y recoloreos menos estrictos que AVL.
- B-trees (y variantes B+): generalizan balanceo a ordenes mayores, permitiendo múltiples hijos por nodo, optimizando acceso en memoria secundaria (disco) mediante nodos de gran orden.

Estas variantes reducen la complejidad de búsqueda, inserción y eliminación a O(log n) garantizando, siendo esenciales en bases de datos y sistemas de archivos.

OTRAS APLICACIONES Y VARIANTES

- Segment Tree y Fenwick Tree (BIT): árboles implícitos diseñados para consultas y actualizaciones de rangos en arreglos con complejidad O(log n).
- Árboles de decisión y KD-trees en aprendizaje automático y procesamiento espacial.
- AST (Abstract Syntax Tree) en compiladores para representar y optimizar expresiones de código).
- Estrucutas indexadas en sistemas de archivos y motores de bases de datos (e.g. B+-trees).

ANÁLISIS DE RESULTADOS DEL BENCHMARK

A continuación, se presentan los resultados obtenidos al ejecutar el módulo *benchmark.py* para comparar el rendimiento de inserción y búsqueda en un BST sin balancear frente a un AVL balanceado:

n	BST insert (s)	BST search (s)	AVL insert (n)	AVL search (s)
1000	0.000508	0.000444	0.001884	0.000358
2000	0.001202	0.001022	0.004139	0.000786
5000	0.003126	0.002432	0.010281	0.001972
10000	0.006336	0.005408	0.022555	0.004320

(ANEXO: A.)

INTERPRETACIÓN DE LOS DATOS

Inserción:

- El BST sin balancear presenta tiempos de inserción muy bajos para estos tamaños, ya que no realiza operaciones de balanceo.
- El AVL incurre en un sobrecosto de inserción (aproximadamente 3-4x mayor) debido a las rotaciones y recálculo de alturas.

Búsqueda:

- El AVL muestra tiempos de búsqueda consistentemente menores que el BST, gracias a la altura balanceada (casi constante O(log n)).
- La ventaja es más notable a mayor n: para n=10000, la búsqueda en AVL (0.00432s) es casi la mitad que en BST (0.00541s).

CONCLUSIÓN

El BST es más eficiente en inserciones para escenarios con datos aleatorios, pero su rendimiento de búsqueda puede degradarse con distribuciones adversas.

El AVL penaliza la inserción, pero garantiza búsquedas más rápidas y uniformes, lo cual es crítico en sistemas donde las consultas de lectura predominen o la estructura deba mantenerse balanceada para evitar casos degenerados.

ANEXO

A.

integradores > integrador_1 > benchmark.py > ...

```python
from .BinarySearchTree import BinarySearchTree
from .AVLTree import AVLTree
import time
import random

"""
Benchmark de rendimiento para comparar BST sin balancear y AVL.
Imprime tiempos promedio de inserción y búsqueda para distintos tamaños.
"""

def benchmark(tree_cls, n, trials=3):
    insert_times = []
    search_times = []
    for _ in range(trials):
        data = random.sample(range(n * 10), n)
        tree = tree_cls()
        t0 = time.perf_counter()
        for x in data:
            tree.insert(x)
        insert_times.append(time.perf_counter() - t0)
        t0 = time.perf_counter()
        for x in data:
            tree.search(x)
        search_times.append(time.perf_counter() - t0)
    return sum(insert_times) / trials, sum(search_times) / trials
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
¿Existe el 60 en BST? True
BST tras eliminar 30 (Inorden): [20, 40, 50, 60, 70, 80]

--- Demo de AVL ---
AVL Inorden (de menor a mayor): [20, 30, 40, 50, 60, 70, 80]
¿Existe el 60 en AVL? True
PS C:\Users\tomas\OneDrive\Escritorio\Textos\Facultad\UTN\Programación 1\repository\integradores> python -m integrador_1.benchmark
n,BST_insert_s,BST_search_s,AVL_insert_s,AVL_search_s
1000,0.000508,0.000444,0.001884,0.000358
2000,0.001202,0.001022,0.004139,0.000786
5000,0.003126,0.002432,0.010281,0.001972
10000,0.006336,0.005408,0.022555,0.004320
PS C:\Users\tomas\OneDrive\Escritorio\Textos\Facultad\UTN\Programación 1\repository\integradores>
```