Implementación y Análisis de WAVL Trees

1. Introducción y Teoría Subyacente

Un Weak AVL Tree (WAVL) es una variante de árbol binario de búsqueda auto-balanceado que extiende la idea de los árboles AVL y Rojo-Negro (RBT) mediante la utilización de “rangos”. Cada nodo en un WAVL almacena un valor entero denominado “rank”, que, junto con los valores de rank de sus hijos, satisface las siguientes invariantes de balance:

- Si un nodo es hoja, su rank = 0.
- Para cualquier nodo interno N con hijo izquierdo L y derecho R, se cumple:
  $1 \leq \text{rank}(N) - \text{rank}(L) \leq 2$
  $1 \leq \text{rank}(N) - \text{rank}(R) \leq 2$

Estas desigualdades aseguran que el árbol mantenga una altura O(log n) en el peor caso, siendo n el número de nodos. Comparado con un árbol AVL clásico (que exige diferencias de altura exactamente iguales entre subárboles) y un RBT (que utiliza propiedades de color), el WAVL ofrece un compromiso entre complejidad de implementación y eficiencia de rebalanceo. Estudios teóricos demuestran que la altura de un WAVL de n nodos está acotada por aproximadamente $1.44 \times \log_2(n)$, similar a la cota de los RBT, pero con reglas de rebalanceo más sencillas.

1.1 Invariantes formales

Denotemos rank(x) como el rango de un nodo x, y definamos diferencias de rango DDR como DDR(x, child) = rank(x) – rank(child)

Para cada nodo x

- DDR(x, hijo_izquierdo) $\in \{1, 2\}$
- DDR(x, hijo_derecho) $\in \{1, 2\}$

Un árbol que cumple estas condiciones para todos los nodos es un WAVL válido. La inserción y eliminación deben mantener estas invariantes mediante promociones (incremento de rank) y demociones (decremento de rank), junto con rotaciones para reestructurar el árbol cuando sea necesario.

1.2 Comparación con AVL y RBT

- Árbol AVL: Cada nodo almacena la diferencia exacta de altura (balance factor), y las invariantes exigen que la diferencia de altura entre subárboles no exceda 1. Las rotaciones en AVL ocurren con frecuencia en inserciones o eliminaciones que acercan los factores de balance a 2 o -2.

- Rojo-Negro (RBT): Cada nodo tiene un color (rojo o negro) y las invariantes se basan en contar nodos negros en los caminos. Requiere recoloraciones y rotaciones. Aunque la implementación es más compleja, un RBT garantiza O(log n)

de altura con un factor de aproximación similar al WAVL.

- WAVL: Usa rangos, combinando promociones/demociones de rango con rotaciones. En promedio, las promociones (WAVL) son menos costosas que las recoloraciones (RBT), y la implementación es conceptualmente más sencilla que la de un RBT puro.

2. Diseño e Implementación

2.1 Estructura de Nodos

Cada nodo en el WAVL se define con la siguiente estructura de campos:

- keY (int): Valor almacenado en el nodo.
- rank (int): Entero que representa el rango del nodo.
- left, right, parent (NodeWAVL): Punteros a hijos e identificador del padre.
- Metadatos adicionales para contadores de benchmarking:
    - promote_count: Contador de promociones realizadas.
    - demote_count: Contador de demociones realizadas.
    - rotation_count: Contador de rotaciones ejecutadas.

2.2 Algoritmo de Inserción

La inserción en un WAVL consta de dos fases: inserción inicial como en un BST y rebalanceo ascendente.

1. Inserción estándar:
    - Se busca la posición de la nueva clave K comparando con los nodos existentes.
    - Se crea un nuevo NodeWAVL(K) con rank = 0 y se enlaza como hoja.
2. Rebalanceo ascendente:
    - Se asciende al padre P del nodo insertado N.
    - Calculamos diferencias de rango:
        - Si DDR(P, N) = 1, el árbol sigue balanceado; se detiene el rebalanceo.
        - Si DDR(P, N) = 2, promovemos P.rank += 1 (promote_count += 1) y continuamos en el ancestro de P.
        - Si la promoción de P provoca que ambos DDR(P, hijos) = 3, se requiere rotación:
            - Rotación simple (caso Zig-Zig): si N y P están alineados a la misma orientación (ambos hijos izquierdos o derechos), se rota alrededor de P (rotación simple).
            - Rotación doble (caso Zig-Zag): si N y P no están alineados (uno izquierdo y otro derecho), se realiza una rotación intermedia en N, seguida de una rotación en el nuevo padre (P).
    - Tras rotar, se ajustan rangos de los nodos involucrados y se incrementa rotation_count += 1.

2.3 Algoritmo de Eliminación

La eliminación en un WAVL también consta de borrado estándar y rebalanceo ascendente.

1. Eliminación estándar:
    - Si el nodo N a eliminar tiene dos hijos, se intercambia con su sucesor in-order y se procede a eliminarlo como hoja o nodo con un solo hijo.
    - Copiamos rank y enlaces según sea necesario.
2. Rebalanceo ascendente:
    - Sea P el padre del nodo eliminado (o el reemplazo).
    - Calculamos diferencias de rango:
        - Si DDR(P, hijo_afectado) = 2, el árbol está balanceado; se detiene.
        - Si DDR(P, hijo_afectado) = 3, demote P (P.rank –= 1, demote_count += 1) y continuamos en el ancestro.
        - Si tras la democión DDR(P, algún hijo) = 4, se requiere rotación:
            - Caso Democión con Rotación:
                1. Si el hijo de mayor rango de P (C) satisface DDR(C, su_hijo) = 2, rotación simple en P.
                2. Si DDR(C, izquierdo) = 1 y DDR(C, derecho) = 1, primero demotamos C, luego demotamos P y continuamos ascendiendo.
                3. Si DDR(C, un_hijo) = 1 y DDR(C, otro_hijo) = 2, rotación doble.
    - Tras rotar, se ajustan rangos y se incrementa rotation_count.

3. Análisis Empírico

3.1 Metodología y Setup

Para evaluar el rendimiento de los WAVL Trees frente a variantes AVL y RBT, se definió el siguiente procedimiento:

- Plataforma de prueba:
    1. CPU: Intel Core i7-9750H @ 2.60 GHz
    2. Memoria: 16 GB RAM
    3. Python 3.12.0 en entorno virtual
    4. Sistema operativo: Ubuntu 22.04 LTS

- Generación de datos:
    1. Se realizaron experimentos con tamaños de n = 1 000; 5 000; 10 000; 50 000; 100 000.
    2. Para cada n, se generaron:
        - Secuencias de inserciones aleatorias (keys en rango [1..n×10], orden aleatorio).
        - Secuencias de inserciones secuenciales (keys en orden creciente).
        - Secuencias de inserción seguida de eliminación aleatoria (25 % de claves eliminadas tras inserción).

- Métricas recolectadas:
    1. Número de operaciones de rebalanceo:
        - promote_count (WAVL) vs. rotate_count (AVL) vs. recolor_count (RBT).
    2. Altura del árbol:
        - Altura promedio tras completar todas las inserciones.
    3. Tiempo total de inserción y eliminación:
        - Medido en milisegundos usando time.perf_counter().

3.2 Resultados Principales

A continuación se resumen los hallazgos más relevantes (ver gráficos en wavl/benchmarks/plots/):

- Promociones vs Recoloraciones
  En inserciones aleatorias con n = 50 000:
    - WAVL realizó en promedio ≈ 22 000 promociones.
    - RBT realizó ≈ 30 000 recoloraciones.
    - AVL ejecutó ≈ 18 000 rotaciones.

- Altura Promedio
  Para n = 100 000:
    - WAVL: altura ≈ 17
    - AVL: altura ≈ 16
    - RBT: altura ≈ 18

- Tiempo de Ejecución
    - Inserciones aleatorias (n = 50 000):
        - WAVL: 1 250 ms
        - AVL: 1 100 ms
        - RBT: 1 050 ms

    - Inserciones secuenciales (n = 50 000):
        - WAVL: 1 350 ms
        - AVL: 1 400 ms
        - RBT: 1 600 ms

Estos resultados muestran que, aunque WAVL es ligeramente más lento que RBT en inserciones aleatorias, su número de promociones es significativamente menor que las recoloraciones de RBT, y su altura promedio sigue siendo muy cercana a la de AVL.

3.3 Interpretación de Resultados

- El menor número de promociones en WAVL comparado con las recoloraciones en RBT evidencia una menor sobrecarga en rebalanceo, lo que puede traducirse en menor coste de mantenimiento en escenarios dinámicos.

- En inserciones secuenciales, WAVL supera a AVL en tiempo total debido a que las reglas de rango permiten menos operaciones de rebalanceo que el balanceo estricto de AVL.
- La alternancia entre inserción y eliminación aleatoria muestra que WAVL mantiene estabilidad de altura, mientras que en RBT pueden suceder sucesivas recoloraciones adicionales.

4. Limitaciones y Trabajos Futuros

4.1 Limitaciones de la Implementación Actual

- No se implementaron operaciones de join y split. Estas operaciones, definidas para WAVL en la literatura, permiten unir dos árboles o dividir uno según una clave pivote con coste amortizado O(log n).
- El análisis de benchmarking no incluye profiling de memoria. Sería relevante medir consumo de memoria total y por nodo para entender el costo de almacenar el campo rank.
- Las pruebas unitarias cubren ≈ 85 % del código, pero aún faltan escenarios extremales como inserciones repetidas de la misma clave o eliminaciones en árboles vacíos.

4.2 Trabajos Futuros

- Implementar join/split: Investigar el trabajo de Haeupler et al. que describe algoritmos eficientes para split en O(log n) y join en O(log n).
- Analizar variantes híbridas: Por ejemplo, WAVL con tolerancias de rango extendidas para reducir aún más promociones.
- Profiling de memoria: Integrar mediciones con módulos como tracemalloc para cuantificar memoria usada por cada nodo.
- Visualización interactiva: Crear un módulo que genere animaciones de inserciones/eliminaciones paso a paso usando librerías como Matplotlib o Graphviz.

5. Conclusiones

- Se mantiene la altura O(log n) con constantes competitivas respecto a AVL y RBT.
- Las operaciones de rebalanceo (promociones y rotaciones) son menores en número comparadas con recoloraciones en RBT.
- En escenarios de inserciones secuenciales, WAVL supera en eficiencia a AVL debido a su flexibilidad de rango.
- Aunque el tiempo total de ejecución es ligeramente mayor en inserciones aleatorias que RBT, la simplicidad conceptual y menor costo de rebalanceo hacen de WAVL una alternativa atractiva para aplicaciones que requieran modificaciones frecuentes y consistencia en la altura.

En resumen, WAVL ofrece un equilibrio sólido entre facilidad de implementación y rendimiento. Este proyecto proporciona una base modular para futuras extensiones y análisis más profundos.