W-tree - Una nueva estructura de datos dinámica para la administración de claves ordenadas en memoria principal.

Sebastián Pacheco Cáceres. Sep 2022

Universidad Austral de Chile, Valdivia - Chile

Resumen Las tendencias actuales de investigación científica computarizada se han encaminado en desarrollar su experimentación utilizando un gran volumen de datos y todas las capacidades computacionales disponibles en las nuevas tecnologías. Las Estructuras de Datos para Ordenamiento de Claves, como el transversalmente conocido Binary Search Tree (BST), han tomado un rol protagonista en la administración de estos datos. Se ha mantenido la atención en la investigación de estas estructuras por más de 50 años, resultando en al menos 20 variaciones distintas que son útiles en diferentes contextos.
En busca de facilitar los entornos de experimentación computacional y conseguir un rendimiento mejor que estas estructuras tradicionales, tanto en memoria como tiempo de ejecución, ofrecemos una nueva variación denominada **W-tree**. Esta estructura considera un parámetro k que establece la cantidad máxima de claves ordenadas que puede almacenar un nodo, y a la vez restringe a $k-1$ su capacidad de enlaces hacia nodos descendientes, lo que se conoce como k-aridad. Gracias al ordenamiento interno de los nodos, las claves mínima y máxima del árbol estarán en los extremos izquierdo y derecho del arreglo de claves, lo que podría aumentar drásticamente la búsqueda de claves inexistentes. Esto también impide realizar rotaciones para balancear la estructura, por lo que la heurística considera mecanismos para amortizar horizontalmente el crecimiento de altura. Esto mantiene el comportamiento asintótico de un BST, pero obtiene un mucho mejor rendimiento empírico.
Para poner a prueba el rendimiento del W-tree, la estructura fue implementada como clases de C++ y pretenden ser "plug & play". Fue comparado contra un BST y un RB-tree (set de la stdlib de C++) en un experimento que prueba las operaciones por etapas. El W-tree demostró siempre utilizar menos memoria que estas estructuras, factor que mejora al aumentar k. Las operaciones mostraron resultados siempre más eficientes que ambas estructuras, pero el comportamiento destaca en un k específico a partir del cual empeora sus resultados.
En base a ambos resultados, se propone utilizar $k = 2^{11} = 2048$. Con este valor y con $n = 2^{28}$ se consigue utilizar, en promedio, $\sim 40\,\%$ de la memoria que ocupa un BST, y en paralelo conseguir un speedup $\sim x2{,}4$ tanto en búsqueda, inserción como eliminación de claves en comparación al RB-tree.

Keywords: W-tree · Data Structure · BST Variation · Efficient

1. Introducción

Un Binary Search Tree es una estructura de datos básica utilizada para la administración de Claves Ordenadas, y partir de ella se han propuesto un sinfín de variaciones que pretenden mejorar su rendimiento bajo diferentes contextos. Todas ellas deben hacerse cargo de las operaciones básicas de búsqueda, inserción y eliminación de claves. Adicionalmente se puede hablar de balanceo, rotaciones, divisiones, entre otro tipo de operaciones que cumplen distintos propósitos.

El uso de Estructuras de Datos para el Ordenamiento de Claves, de la familia de los árboles binarios como BST o AVL, es transversal y llega al uso cotidiano de diversas maneras, como en los sistemas de ficheros de ext3 o ext4, en bases de datos, y en particular, también se utilizan en la administración de datos en problemas de computación geométrica. Dentro de este campo de investigación existe un continuo interés por la mejora, pues cada vez se tiende a utilizar más datos para llegar a una mejor solución[1], por lo que se suele sacrificar precisión en pos de utilizar mejor los recursos.

Dado que construir un BST óptimo que minimiza la cantidad de comparaciones necesarias para encontrar una clave es un problema NP-completo [8], existe una gran cantidad de variaciones del BST tradicional, de los cuales se diferencian dos enfoques: uno orientado a las operaciones con bloques de información (información en discos o memoria secundaria) y el otro orientado a información en memoria principal (memoria primaria o temporal), donde correspondientemente se destacan los árboles B+ y árboles Rojo-Negro (RB-tree)[2].

Para la utilización en memoria principal generalmente se recurre a la familia de árboles AVL[7], principalmente al RB-tree, que pagan un coste computacional relativamente pequeño a favor de mantener el árbol relativamente equilibrado, para alguna definición particular de equilibrio. Si bien esto es una solución bastante eficiente frente a los peores casos de uso, no lo es para el caso promedio, fenómeno que ocurre de diversas maneras dependiendo de la heurística de la variación en particular, y la naturaleza de la distribución de la cual provengan los datos. Considerando su rendimiento empírico, las estructuras que se utilizan en memoria principal suelen ser las más clásicas, como árboles RB, AVL e inclusive los simples BST[14].

El presente proyecto se enfoca en el diseño e implementación de una nueva variación de un BST orientada a memoria principal, denominado **W-tree**. Su propósito es lograr mejorar la eficiencia empírica, tanto de las operaciones utilizadas como de la memoria requerida, con relación a otras estructuras como los árboles AVL, árboles RB o árboles de búsquedas en k-dimensiones —las cuales son muy utilizadas en computación geométrica—.

El W-tree considera una nueva heurística de ordenamiento que toma ventaja de la k-aridad, donde k es un parámetro que determina la cantidad máxima de claves por nodo y también restringe a $k-1$ la cantidad máxima de descendientes. A cada par de claves dentro del nodo le puede corresponder un nodo descendente, evocando una forma de V a la relación (que refleja el caso de $k=2$), que por la k-aridad lo convierte similar a una W (que corresponde al caso de $k=3$).

Esto lo vuelve incompatible con la operación de rotación de los AVL (también ocupado en RB-trees y T-trees[9]), lo que es sustituido por una preferencia a amortiguar horizontalmente el crecimiento frente a una inserción. Un nodo descendiente solo se creará si es que no es posible amortiguar la inserción hacia nodos laterales, que existan o puedan ser creados. Los nodos que tienen descendencia se denominan "internos" y los que no tienen se denominan "hojas". Todo nodo interno siempre contiene k claves.

La mejora de eficiencia lograda en el W-tree, tanto de tiempo de ejecución de las operaciones básicas como de memoria utilizada, se logra gracias a la forma de la estructura, un posible recorrido en caché de las claves de un nodo, y otras optimizaciones orientadas al uso de memoria. La experimentación indica que los mejores resultados varían según la operación observada, pero generalmente se consiguen utilizando un parámetro k particular, que tiene relación con el tamaño de la memoria caché y su comunicación con la memoria principal (RAM).

Teniendo de base los pseudocódigos diseñados para las diversas operaciones a utilizar, se realizó una implementación en C++ del W-tree, en formato de librería, que logra ser un sustituto equi-

valente a las estructuras tradicionales y que no requiere de mayores esfuerzos para integrarse en proyectos existentes, siendo un equivalente de un *Ordered Set*.

Utilizando esta implementación, se muestran los resultados de pruebas comparativas tanto propias (respecto a su k-aridad) como con estas otras estructuras. Actualmente están disponibles las comparaciones con un BST y un Árbol RB, donde este último corresponde a la implementación del contenedor Set de la librería estándar de C++ al compilar con gcc [1].

2. Estado del arte

Un BST es una estructura de datos orientada al Ordenamiento de Claves, y surge basándose en el funcionamiento del algoritmo de Binary Search[5], lo que en esencia explica que el dominio de búsqueda busca reducirse solo a los menores o mayores valores respecto de algún valor pivote en particular. Sin embargo, la búsqueda binaria trabaja con un arreglo ya ordenado, lo que requiere invertir recursos computacionales previamente.

Es aquí donde el BST se presenta como una solución dinámica para mantener el orden de claves de acuerdo a inserciones y/o eliminaciones progresivas. La forma del BST resultante es entonces sensible a, por ejemplo, el tipo de distribución que caracterice a los elementos insertados y de los elementos buscados[16], que no necesariamente son similares.

Construir un BST "óptimo", considerando como óptimo que minimice la cantidad de comparaciones necesarias para encontrar una clave cualquiera (o más de una), es un problema **NP-completo tratable** según fue demostrado en 1976[8].

- Óptimo: Minimiza la cantidad de comparaciones necesarias buscar algún conjunto de elementos.
- Tratable: El problema utiliza operaciones de coste polinomial en tiempo.
- NP-completo: La minimización involucrada en crear un BST óptimo es un problema reducible al NP-completo de Exact-Cover. Esto también implica que existe un algoritmo exponencial que lo soluciona (que en este caso es mucho mejor), y que es verificable en tiempo polinomial.

Suponiendo la naturaleza de $P \neq NP$, la misma publicación argumenta que no existe un único algoritmo eficiente que se haga cargo del problema, sino que las soluciones deben optar por plantear heurísticas que mejoren los resultados, construyendo *árboles de decisión semi-óptimos*. Para más detalles sobre esta NP-completitud se puede consultar el Anexo 10.1.

Entre las herramientas disponibles en los lenguajes de programación, y en específico los de bajo nivel como C++, se encuentra la posibilidad de referenciar la posición de variables o estructuras, abriendo la posibilidad de moverse a través de estas posiciones bajo ciertas condiciones.

Es entonces posible representar, computacionalmente, grafos con nodos cuya complejidad depende de la implementación, donde en general **la operación más costosa suele ser moverse a través de direcciones**. Esto ha mantenido la atención en su investigación (principalmente en la eficiencia con la arquitectura hardware), con una orientación hacia el rendimiento empírico, por más de 50 años, resultado en un gran número de estructuras que son variaciones de un BST.

Conociendo entonces la naturaleza del problema, el estudio del estado del arte se enfocó en conocer las variaciones existentes del BST, sus posibles aplicaciones prácticas, y también optimizaciones aplicables a un nuevo diseño.

[1] Implementación de Set en compilador gcc: `https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.1/stl__tree_8h-source.html`

Las primeras indagaciones indicaron que, dadas las limitaciones tipo Von-Neumann (cuellos de botella en el transporte de información) del hardware computacional, existen variaciones orientadas a funcionar en memoria principal (caché, RAM, entre otras) y en memoria secundaria (HDDs, SSDs, entre otras).

También surgen un par de conceptos interesantes de tener en cuenta en las descripciones presentadas en este escrito:

- k-aridad: Este término en particular se refiere a la cantidad máxima de descendientes de un nodo. En otros casos la k-aridad es considerada con la cantidad máxima de valores, como en los T-trees [9] donde, a su vez, se restringe a un máximo de dos descendientes.
- balanceados: A un árbol BST (o similar) se le considera balanceado cuando la diferencia de altura entre sus sub-árboles izquierdo y derecho es relativamente pequeña, de acuerdo al estándar definido para la estructura en particular. Para lograr este balance se ocupan operaciones como rotaciones, splits o joins, además de utilizar algunas métricas particulares para dirigir los ajustes. Un claro ejemplo de esto son los árboles AVL [7] y los T-trees.

Revisión Bibliográfica Dado el uso de estructuras de ordenamiento como una de las herramientas básicas para la experimentación computarizada, la naturaleza NP del problema, y el conocimiento generalizado que se tiene de estas estructuras, fue planteada una primera pregunta de investigación para impulsar la revisión bibliográfica:

¿Existen variaciones de BST con k-aridad que sean más eficientes
que un BST tradicional en programas que corren en memoria principal?

Una primera búsqueda resultó en 125 artículos, por lo que fue necesario incluir términos excluyentes que acotaran los resultados que principalmente corresponden a resultados de investigación en biología, como proteínas BST, entre otras, y otros términos incluyentes que ampliaran la cantidad de artículos con acceso público.

Tras una serie de iteraciones, la cadena de búsqueda final resultó en la siguiente:

("BST" OR "Binary Search Tree" OR "k-ary tree" OR "k-ary" OR "k-tree" OR "multiway-tree" OR "multiway") AND ("Cache" OR "Main Memory" OR "RAM" OR "Empiric" OR "Empirical") NOT ("BST-2" OR "BST-236" OR "Tetherin" OR "Aspacytarabine" OR "Ferroelectric" OR "CO2")

La base de consulta fue únicamente una en vista de la gran cantidad de resultados:

- `https://www.webofscience.com`: Iniciativa del grupo Clarivate que busca apoyar la investigación y emprendimiento, ofreciendo una base de datos que indexa un gran número de artículos, revistas, actas de lectura y otro tipo de material de investigación científica.

Esta cadena de búsqueda fue revisada en la web por última vez el 12 de octubre de 2021, y resultó en **56 artículos** en total. El único filtro aplicado es su disponibilidad de acceso libre, además de la posterior clasificación por criterios incluyentes/excluyentes de acuerdo a si tratan sobre memoria

caché, alguna aplicación práctica o es una variación de un BST. Esto también implica que no es de nuestro interés que las publicaciones sean necesariamente recientes.

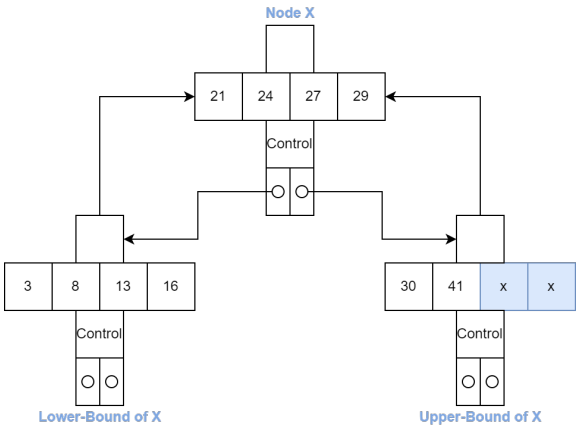
De acuerdo con los hallazgos, se podría considerar que el enfoque de la investigación se ha mantenido relativamente homogéneo entre dos perspectivas; una de incluir tanto aspectos teóricos como el análisis de comportamiento asintótico, y otra sobre aplicaciones u optimizaciones de bajo nivel, como el uso de memoria principal, eficiencia de buses o caché, e inclusive el consumo energético de los componentes electrónicos que realizan los cálculos.

Es decir, el comportamiento del hardware se destaca como un factor de los más relevantes en producir resultados contra-intuitivos respecto a lo que indican los análisis teóricos.

El análisis asintótico no necesariamente garantiza optimalidad, sino que también hay decisiones en tiempo de ejecución que no pueden ser tomadas por el usuario ni el programador, que son de autonomía del sistema operativo o parte de un funcionamiento de bajo nivel.

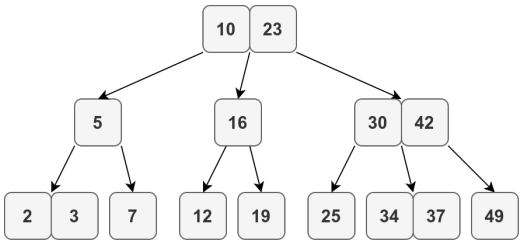
Esto ocurre en la Branch-Prediction, un fenómeno que ocurre a nivel CPU y que es aprovechado por el Skewed-tree[3] utilizando ramas desbalanceadas.

Figura 1: T-tree



Existe un estudio comparativo[9] que puso a pruebas diversas estructuras orientadas a memoria principal, y a la vez propuso la heurística de los **T-trees**, que es bastante similar a un BST. Ambas optan por ordenar sus claves ascendentemente (o descendente equivalentemente), pero un T-tree toma ventaja de la k-aridad y solo puede tener hasta dos descendientes. Para balancearse utiliza las mismas rotaciones que un AVL (RR, RL, LL y LR), pero por su k-aridad esto ocurre con mucho menor frecuencia. Uno de los principales problemas de su utilización es la cantidad de *memoria de control* de los nodos, pues cada uno deberá guardar 3 punteros (2 descendientes + 1 ascendente).

Figura 2: B-tree



Otro caso interesante de analizar es el del **B-tree**, una especie de árbol m-ario. Sus nodos crecen gradualmente teniendo hasta $(m-1)$ claves para m descendientes, donde cada nodo necesita al menos $\frac{m}{2}$ claves para crear un nodo descendiente.

Existe la variación denominada árbol B+ que solo almacena la información en nodos hojas, de manera que sus nodos internos almacenan claves artificiales que solo sirven para escoger un camino hacia las hojas. Sus nodos también incluyen un puntero hacia nodos adyacentes en el mismo nivel, facilitando el recorrido ordenado de las claves. Su principal utilización es en contextos de memoria secundaria y bases de datos, pero también existen implementaciones para memoria principal.

Posteriormente con los años se propone al denominado árbol B*, cuya heurística plantea llenar los nodos al menos a 3/4 de su capacidad. Recientemente se han combinado ambas propuestas en una sola, presentando así al árbol B*+[15]).

En un contexto más técnico, según las capacidades del hardware que se utilice, en el ámbito de la programación existen 4 modelos de datos[2] considerados estándares, que establecen el tamaño que utilizan los tipos de datos primitivos, tal como se muestra en el Cuadro 1.

En C++ la utilización de estructuras o clases suele requerir el encapsulamiento de una serie de atributos, los cuales deben ser *alineados* al largo de palabra de la arquitectura utilizada, lo que provoca la adición de **bytes de padding** suficientes para llenar la palabra.

Cuadro 1: Tamaño en bytes según diversos modelos de datos

Modelo	System	bytes int	long	pointer
LP32	Win16 x32	2	4	4
ILP32	Win32 & Unix x32	4	4	4
LLP64	Win64 x64	4	4	8
LP64	Unix x64	4	8	8

Hoy en día los computadores de uso común suelen traer sistemas de 64 bits (x86), es decir, cargan una "palabra" de 8 bytes en caché en solo un ciclo. Si bien puede haber casos donde esto tiene en el uso de memoria, ha sido un costo que se ha estudiado y asumido en pos de operar, bajo usos generales, a una mayor velocidad[3]. Aun así, los programadores también cuentan con herramientas para redefinir esta configuración, buscando compresión de datos antes que velocidad de acceso.

Cuadro 2: Tabla de memoria utilizada por el nodo de un BST tipo int en C++.

Atributo	Bytes
value	4
left_pointer	8
right_pointer	8
Total	**20**
+ padding	**24**

En el Cuadro 2 se puede observar la memoria utilizada por un nodo simple de BST implementado en C++ para un tipo de dato entero (int), donde evidentemente se ocupa más memoria en almacenar los punteros y además existen bytes de padding.

Si consideramos como *memoria de control* a la memoria utilizada por todos los atributos que no corresponden a la clave de los nodos, se cumplen las equivalencias:

$$\begin{aligned} total &= bytes\_value + (bytes\_pointers + bytes\_padding) \\ &= bytes\_value + control\_memory \end{aligned}$$

Con estos parámetros se puede calcular la relación entre la *memoria de control* y la memoria utilizada por el tipo de dato almacenado. En otras palabras, se tiene que los **Bytes Adicionales por Dato (BAPD)** para n elementos son:

$$BAPD = \frac{control\_memory}{n} = \frac{20}{1} = 20\ bytes \tag{1}$$

Está claro entonces que los punteros son los atributos más influyentes en el uso de memoria, por lo que en busca de optimizaciones es un factor interesante. En esto también se debe considerar que los nodos hojas no tienen descendencia, y por ende cuentan con espacio asignado que no necesitan.

[2] CPP Reference - Tipos de modelos de datos: `https://en.cppreference.com/w/cpp/language/types`
[3] Wikipedia - Data structure alignment: `https://en.wikipedia.org/wiki/Data_structure_alignment`

En aplicaciones prácticas la cantidad de claves que se insertan en estas estructuras es de gran volumen, por lo que lograr pequeñas diferencias en esto puede tener un gran impacto. Para más información sobre este efecto a nivel de programación, consultar este sitio web [4].

En paralelo podemos definir una métrica que no depende de la implementación, sino de la proporción entre esta cantidad de claves y punteros:

$$\text{tasa de k-aridad} = \frac{n\_punteros}{n\_valores} \tag{2}$$

En en Cuadro 3 se presenta esta tasa para el tradicional BST y el B-tree. El caso más básico dentro de estas estructuras es el BST cuyo tasa siempre resultará 2,0. Ninguna otra variación de BST considera este valor mayor a 2,0. Un contraejemplo de árbol que sí lo cumple es el Suffix-tree, pero este es utilizado con un propósito distinto y no cuenta con las propiedades de un BST.

Para el caso del B-tree, al ser un árbol m-ario requiere de utilizar $m \geq 3$, por lo que en su peor caso esta tasa nos dará 1,5. Es decir, esta estructura se enfoca en disminuir este tasa en busca de ventajas empíricas en el rendimiento.

Cuadro 3: Tasa de k-aridad para un BST y un B-tree m-ario.

Estructura	Tasa k-aridad
BST	$\frac{2}{1} = 2{,}0$
B-tree	$\frac{m}{m-1} <= 1{,}5$

Por otro lado, también se ha estudiado la cantidad posibles configuraciones que se forman al insertar una cantidad n de claves, pero variando su orden. De acuerdo con aproximaciones del Número de Catalán, esta cantidad está acotada por:

$$\Omega(4^n * n^{-3/2})$$

En este gran número de posibilidades es posible tomar partido de la información que se conoce respecto al estado del árbol, como guardar su altura, su balance, la cantidad de consultas recibidas, entre otros parámetros. Si es que el comportamiento del sistema se conoce hasta el punto de poder predecir las probabilidades de aparición, existe una variedad de algoritmos o mejoras que se encargan de comprimir y optimizar los recursos necesarios para funcionar. Cuando el funcionamiento es estático (cuando no se agregan ni eliminan elementos), existe un algoritmo de programación dinámica[17] que puede crear un BST óptimo.

En el caso dinámico, existe la variación denominada Splay-tree [16] que se encarga de acercar las claves más buscadas a la raíz, pero es claramente más efectivo bajo contextos donde se busca muy frecuentemente un pequeño conjunto de claves.

Estas estructuras son a su vez analizadas desde la perspectiva de otros paradigmas de la ciencia computacional, incentivando el interés por aplicar compresión a los datos[18] (en busca de aprovechar mejor el ancho de banda del hardware), y/o del funcionamiento en paralelo con multiprocesadores[6].

A partir del ordenamiento generado por esta diversidad de estructuras surge una gran variedad de problemas computacionales a solucionar, donde se destaca la búsqueda del k-menor de un conjunto, y/o sus k-vecinos más cercanos. Este último problema es escalable a múltiples dimensiones, y forma parte de uno de los más importantes problemas geométricos pues no existe un algoritmo determinista que resuelva esto en menos que tiempo lineal.

[4] Geeks for Geeks - Structure member alignment, padding and data-packing: `https://www.geeksforgeeks.org/structure-member-alignment-padding-and-data-packing/`

Una de las formas de abordar este problema es con aproximaciones "tolerables" calculadas con puntos semi-ordenados. Se ha demostrado que la estructura que mejor se desenvuelve en este problema es un algoritmo reciente que propone una búsqueda priorizada dentro de árboles k-means jerárquicos [10], el cual rápidamente evolucionó en la propuesta de priority search k-means tree[11].

Este tipo de soluciones también son aplicadas a problemas del mundo real, como la simulación de fenómenos físicos, el diseño espacial de redes 5G[19] y la orientación espacial para robots, los cuales se contextualizan como aplicaciones de computación geométrica.

Por otro lado, se conoce que las operaciones realizadas sobre arreglos son capaces de ser paralelizadas la mayoría de las veces. Entre sus aplicaciones se encuentra la reducción aritmética, que es la aplicación de una operación en particular a algún rango de elementos, que gracias a nuestra estructura también están ordenados.

Respecto a la reducción se encuentran diversos aportes, como por ejemplo su cálculo en GPU Tensor-cores[12], o incluso el modelo de programación MapReduce[4] de Google, que procesa y genera grandes cantidades de información. En particular, este modelo transforma una combinación (key, value) a un valor intermedio y realiza una reducción en paralelo que logra identificar todas las claves coincidentes, obviamente con datos de escala masiva.

Es necesario destacar también que muchas aplicaciones alimentan estas estructuras con los datos para luego realizar mayoritariamente operaciones de búsquedas y muy posiblemente sin realizar eliminaciones, aspecto del cual también tomará ventaja la propuesta.

2.1. Preguntas e hipótesis de investigación

En vista de la inquebrantable relación entre toda clase de aplicación computacional y la necesaria utilización de estructuras de datos ordenadas en la administración de su información, surge la idea de diseñar, implementar y poner a prueba una nueva variación de BST, orientada al ordenamiento de claves en memoria principal, en busca de mejorar la eficiencia empírica respecto a estructuras tradicionales.

Desde una perspectiva más teórica de los algoritmos, los hallazgos motivan a tomar ventaja de la k-aridad (uso de arreglos) para disminuir el uso y movimiento entre punteros. Teniendo en cuenta la tasa de k-aridad (según la Ecuación 2) del B-tree en su caso más básico, según se muestra en el Cuadro 3, la primera pregunta de investigación planteada es si es que:

¿Es posible diseñar una estructura de datos para ordenamiento de claves en memoria principal que tenga una tasa de k-aridad menor a 1,5?

Generalmente la experimentación científica computarizada es llevada a cabo por personas no necesariamente informáticas, por lo que es prudente suponer que la mayoría utiliza las estructuras (o contenedores en C++) disponibles por defecto en cada lenguaje de programación. En busca de reemplazar a la estructura tradicionalmente utilizada en C++ como contenedor **set**, es decir un árbol rojo-negro, se estipula en una segunda pregunta de investigación si es que:

¿Es posible diseñar una estructura de datos para ordenamiento de claves en memoria principal que utilice menos memoria que un árbol rojo-negro y a su vez consiga mejorar su rendimiento empírico?

La significancia de esta diferencia, a favor de nuestra propuesta o de las estructuras tradicionales, es enunciado de forma estadística a través de las siguientes hipótesis de investigación:

Hipótesis 1 Los tiempos de ejecución de la estructura propuesta son estocásticamente menores que los del set de C++

- H_0: son estocásticamente mayores o muy similares.
- H_a: son estocásticamente menores.

Hipótesis 2 La memoria asignada para la estructura propuesta son estocásticamente menores que los del set de C++

- H_0: son estocásticamente mayores o muy similares.
- H_a: son estocásticamente menores.

Para validar o rechazar estas hipótesis se utilizará el test no-paramétrico de **Mann-Whitney U**[5], útil para comparar la localización de dos distribuciones similares con pocas muestras, y también para distribuciones que no necesariamente son gaussianas. A su vez permite especificar el tipo de hipótesis alternativa, facilitando más aún la comparación.

3. Propuesta

En vista de las limitaciones que han sido estudiadas y aquí expuestas, además de las diferentes estructuras que se han propuesto como solución en la literatura, tendremos las siguientes consideraciones para diseñar una nueva solución, que posteriormente será implementada en C++:

- Arreglos: Ya que la operación más costosa al operar en estas estructuras es el movimiento a través de punteros, utilizar arreglos disminuye la cantidad de nodos necesarios para almacenar los datos, y por ende, los movimientos entre punteros.
- Poca relevancia de eliminación: La mayoría de aplicaciones prácticas crean la estructura desde 0, operan y luego limpian la memoria. Es decir, la eliminación es una operación poco prioritaria, lo que permite enfocarse en el rendimiento de las otras operaciones, principalmente de la búsqueda de elementos.
- Baja eficiencia de rutinas de balanceo: Algunas estructuras, como el árbol AVL, consideran la operación de rotación para mantener un balance en su altura. Bajo distribuciones normales de datos esto no parece tener un beneficio de mayor impacto, por lo que también puede ser un aspecto a considerar de baja relevancia.

Considerando todo lo expuesto, y principalmente la utilización de arreglos, se pensó en el diseño de una estructura que mejorase significativamente los BAPD de los nodos, lo que consecuentemente podrá disminuir los tiempos de ejecución. Es decir, se busca cumplir que la relación de k-aridad:

$$\text{tasa de k-aridad} = \frac{\text{n° punteros}}{\text{n° claves}} \leq 1{,}0$$

[5] Test Mann-Whitney U en Scipy: `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.mannwhitneyu.html`

Para el caso de un BST, la ecuación anterior siempre resultará en dos, y ninguna otra variación de BST considera este valor mayor a dos. Un ejemplo de árbol que sí lo cumple es el Suffix-tree, pero este es utilizado con un propósito distinto y no cuenta con las propiedades de un BST.

En busca de acercarnos a esta cota de la relación de k-aridad y así poder responder a la primera pregunta de investigación, surge la idea de invertir la relación clásica de k-aridad, ahora considerando k-claves y $(k-1)$ descendientes. Esto forma una especie de "W", que para ser consistente requiere de una pequeña modificación respecto a la heurística clásica de ordenamiento de claves en estas estructuras, por lo que todas las operaciones involucradas en su inserción, búsqueda y ordenamiento deberán ser diseñadas desde cero.

Para el caso de un W-tree (con $k \geq 3$), su ratio de k-aridad es:

$$\frac{2}{3} \leq \frac{k-1}{k} < 1{,}0 \tag{3}$$

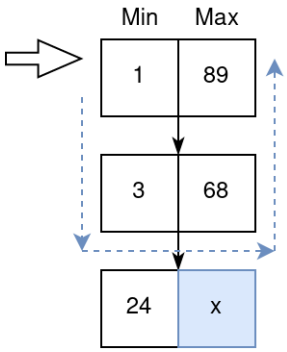
La estructura de los nodos se basa en utilizar arreglos cuyo largo es parametrizable con la variable k, y además guarda un registro booleano que indica si es que es un nodo interno u hoja. La particularidad de estos tipos de nodos será discutida posteriormente.

El W-tree y sus nodos siguen el siguiente esquema para su estructura algorítmica:

```
struct Nodo{
    n                   cantidad de elementos almacenados
    isInternal          booleano que representa si el nodo puede tener descendencia
    values[1...k]       arreglo de claves de capacidad k
    pointers[1...k-1]   arreglo de punteros de capacidad k-1
}

struct WTree{
    k       cantidad máxima de claves a almacenar en los nodos
    root    puntero que apunta al nodo raíz (de existir)
}
```

Figura 3: W-tree, k=2



La nueva forma de organizar a los valores y los punteros descendientes de cada nodo requiere de establecer criterios que indiquen cuándo crear descendencia. Ya que los nodos deben crear arreglos internos, siempre que tengan $< k$ valores existirán casillas de valores sin utilizar. También, cuando se crea el primer descendiente todo el resto de punteros son nulos.

Para que esto ocurra la menor cantidad de veces posible, la heurística considera solo crear descendencia cuando $n = k$ y además no se pueda amortiguar el crecimiento horizontalmente con nodos adyacentes existentes o que puedan existir. Todo nodo con $n \leq k$ sin descendencia se denomina **nodo hoja**, y de lo contrario es un **nodo interno**.

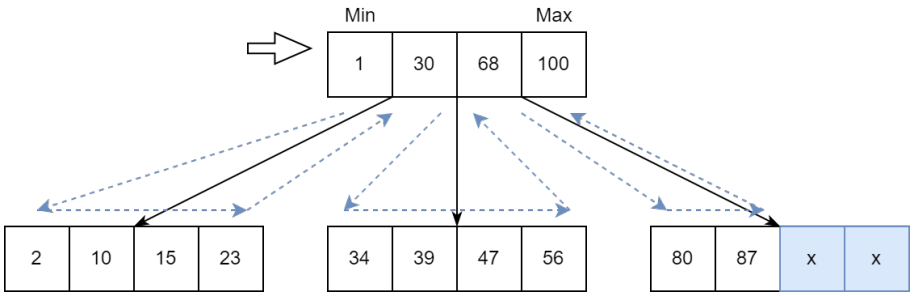
El caso más básico de la estructura es cuando $k = 2$ y se observa en la Figura 3, donde el orden de la lectura de las claves se indica con la flecha azul punteada. Este caso no parece tener

mayor utilidad, pues no es más que un arreglo secuencial pero ordenado verticalmente con el uso de punteros.

Para ahorrar espacio en el vector de valores utilizamos la técnica de **doubling array**, lo que mantiene al menos $\frac{1}{2}$ de las casillas ocupadas.

La forma útil más básica en la experimentación es para el caso de $k = 4$, que es una potencia de 2. Un ejemplo de esto se presenta en la Figura 4, donde la raíz es el único nodo interno y existen dos nodos hoja completos, cuyo recorrido ha sido marcado con las flechas punteadas azules:

Figura 4: W-tree, k=4

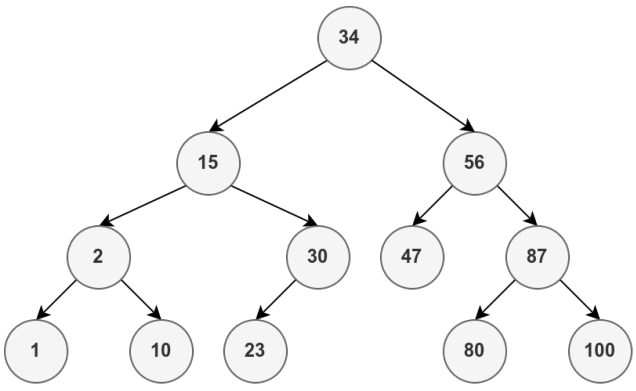


Esta nueva estructura acota de una forma distinta el dominio de los valores buscados. Un BST pretende ir dividiendo el dominio por la mitad, pero este no es acotado lateralmente (su límite es el infinito) a menos que esta sea una limitación impuesta, como es nuestro caso al trabajar con tipos de datos finitos.

Consideremos un tipo de dato integer (4 bytes) y un arreglo de las claves (que por legibilidad han sido ordenados ascendentemente) que fueron insertadas en el W-tree de la Figura 4:

$$\text{Valores} = [1, 2, 10, 15, 23, 30, 34, 39, 47, 56, 68, 80, 87, 100]$$

Figura 5: BST equivalente a W-tree de Fig. 4



Para los BST, la forma más básica de operar no considera mantener a la mano los valores del mínimo y máximo elemento. A priori el rango de búsqueda es incierto y corresponderá a todo el rango de valores que pueden tomar los integers (o eventualmente, infinito). Además, toda clave inexistente solo es identificable como tal si es que el recorrido de búsqueda llega a algún nodo hoja.

Un BST en este caso demora bastante en "darse cuenta"que la clave 60 no fue insertada, lo que puede ocurrir tras muchas más comparaciones si es que el BST no está balanceado. Por otro lado, la cota derecha fue actualizada recién en la tercera iteración, por lo que si la clave estuviera muy fuera de rango (ej.

la clave 1000) hubiera sido mucho más útil conocer desde un principio la mínima y máxima clave del árbol.

Cuadro 4: Tabla de búsqueda de la clave inexistente 60 en el BST de la Fig. 5.

	Dominio de búsqueda		
N° puntero	Izquierda	Punto Medio	Derecha
1	-2147483648	34	2147483647
2	34	56	2147483647
3	56	87	2147483647
4	56	80	87
-	**Nodo hoja alcanzado**		

En la Fig. 5 se observa un BST equivalente al de la Fig. 4. Al considerar la búsqueda de la clave inexistente 60 en el BST, en la Tabla 4 se puede observar el progreso de la cota del dominio de búsqueda.

Análogamente, esta misma búsqueda se realiza de una manera distinta en el W-tree, permitiendo identificar la inexistencia de una clave con muchas menos comparaciones y movimientos entre punteros. El progreso de este proceso se observa en la Tabla 5, donde cada iteración realiza dos comparaciones para actualizar estos rangos sin tener que almacenar las claves mínima y máxima del árbol.

Cuadro 5: Tabla de búsqueda de la clave 60 en el W-tree de la Figura 4.

	Dominio de búsqueda			
N° puntero	Izquierda	Mínimo	Máximo	Derecha
1	-2147483648	1	100	2147483647
	1	1	100	100
2	34	34	56	56
-	**Clave fuera de rango**			

En este W-tree basta con visitar dos nodos para identificar que la clave está fuera del dominio de búsqueda, lo que por cierto puede suceder tanto en nodos hojas como internos. Aquí también se observa que el dominio no está sujeto a los valores mínimos y máximos que son almacenables en el tipo de dato (integer en este caso), sino a los valores mínimos y máximos realmente insertados en la estructura. Como será explicado más adelante, esto trae importantes ganancias en los tiempos de búsqueda en general, y más específicamente en la búsqueda de claves inexistentes.

3.1. Heurística de construcción

Para la construcción de la estructura se considera una serie de necesidades que surgen al invertir la relación entre la cantidad de claves almacenadas y el número máximo de descendientes por nodo.

Las siguientes consideraciones permiten la creación coherente y funcional de la estructura:

- **Relación entre valores y descendientes:** Las claves se ordenan en forma ascendente, y a cada par consecutivo de claves, a y b, le puede corresponder un nodo descendiente si es que se inserta alguna clave dentro del rango $]a, b[$. Esto implica que la cantidad de descendientes en un nodo siempre es menor a su cantidad de claves. En un BST la cantidad de punteros corresponde al doble que su cantidad de valores, como se explica en la ecuación 1.
- **Mínimos y máximos:** Gracias a la estructura de los nodos y su manera de crear descendencia, cada nodo u contiene a las claves mínimas y máximas del sub-árbol que forma junto a su descendencia, ubicadas en los extremos inicial y final del arreglo de claves.
- **Sin claves repetidas:** No se aceptan claves repetidas, pero sí es adaptable para tal fin.

- **Nodos internos y hojas:** Todo nodo hoja puede contener $n \leq k$ valores, pero no tienen descendencia. Si es que la tienen, se denominan nodos internos.

  En consecuencia, la heurística buscará crear al menos dos (o tres) nodos por nivel de altura, postergando así la creación de nuevos niveles. Si la eliminación sucede en algún nodo interno, se busca entre su descendencia algún camino descendiente hacia una hoja y se obtiene la mínima o máxima clave (dependiendo del camino seguido) para ascender como sustituta.

3.2. Características principales

Gracias a la heurística planteada, el W-tree se caracteriza principalmente por:

- **Utilización de arreglos (*k-aridad*):** La k-aridad de la estructura es uno de los atributos más relevantes en su rendimiento, y toma principal ventaja de la diferencia entre el costo computacional requerido para recorrer un arreglo (que ocurre a nivel caché), comparado al de moverse entre punteros (o nodos correspondientemente).
- **Doubling array para las claves:** Los vectores de C++ suelen asignar más memoria que la realmente utilizada. Esto sucede duplicando su tamaño cada vez que se necesite, partiendo de algún valor base. Esto implica que siempre $n \geq \frac{capacity}{2}$.

  En el W-tree asumimos un tamaño mínimo de 4 elementos, y a partir de él suceden duplicaciones hasta llegar a k, por lo que el mejor rendimiento se conseguirá al utilizar valores potencias de 2 para este parámetro.
- **Hojas sin descendencia:** La heurística prioriza llenar la mayor cantidad de nodos hojas antes de crear descendencia, y por lo mismo **no asigna memoria** para estos punteros, lo que para datos enteros (int) permite ahorrar cerca del $\sim 80\,\%$ del espacio total utilizado por los nodos. Esto es comentado con detalles al término de este listado.
- **Inserción determinista y amortiguada horizontalmente:** Dada la naturaleza NP, existen numerosas formas coherentes de insertar una clave, por lo que es necesario establecer prioridades que permitan al algoritmo decidir qué hacer frente a situaciones ambiguas.

  El algoritmo presentado sigue una serie de reglas, descritas en la sección 4.2, diseñadas para priorizar un crecimiento horizontal. Al insertar una clave dentro de un nodo hoja que está completo ($n = k$) se intenta insertar en algún posible nodo hermano adyacente antes de crear una nueva hoja. Esta metodología no solo busca equilibrar el árbol con un menor esfuerzo que las rotaciones típicas de los AVL, sino que también ahorrar un espacio considerable al evitar crear nodos internos (y sus correspondientes punteros descendientes).
- **Acelerada búsqueda de claves ausentes:** Tomando ventaja de la forma de los nodos, la búsqueda de claves ausentes se acelera cuando, en algún nodo u, se identifica que la clave está fuera del rango formado entre el mínimo y máximo del nodo, que a su vez corresponde al del subárbol formado por u y su descendencia.
- **Eliminación compacta en nodos internos:** La eliminación en nodos internos siempre sustituye la clave removida, manteniendo $n = k$ con alguna clave sustituta ascendida desde algún nodo hoja, el cual busca priorizando la búsqueda por la izquierda.

Cuadro 6: Tabla de memoria utilizada por el nodo de un W-tree tipo int en C++.

Atributo	Bytes
k	2
n	2
isInternal	1
values	$24 + k * 4$
pointers	$24 + (k-1) * 8$
Total estático	53
+ padding	56
Máximo real	$48 + 12 * k$
Postergable	$(k-1) * 8$

En la tabla 6 se observa la estructura de la memoria utilizada por los nodos de un W-tree. Es necesario acotar que la memoria utilizada por C++ se compone de memoria "estática" y dinámica.

En el caso de estos nodos, los atributos son estáticos, pero las claves son almacenadas dinámicamente por vectores, los cuales contemplan una parte estática de 24 bytes (con atributos que le permiten la consistencia de memoria correspondiente). A esta parte estática es a la que se le aplican bytes de padding, resultando en 56 bytes para un nodo vacío. Para la parte dinámica, y como se ha nombrado anteriormente, la asignación de espacio para punteros es pospuesta lo más posible.

Para notar la relevancia de esto, considerar un W-tree con $k = 1024$:

$$\begin{aligned} \text{postergable} &= \quad 8{,}184 \text{ KB } (\sim 66{,}34\,\%) \\ \text{maximo real} &= 12{,}336 \text{ KB} \end{aligned}$$

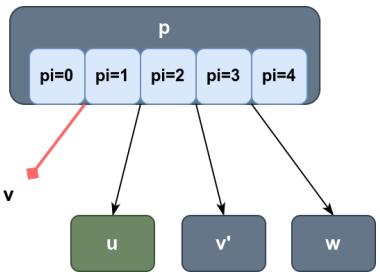
4. Diseño Algorítmico

El funcionamiento de las estructuras de datos deben hacerse cargo de al menos las tres operaciones básicas para administración de información, que corresponde a **insertar, buscar y eliminar elementos**.

En el planteamiento de los algoritmos que se hacen cargo de estas operaciones en un W-tree es útil diferenciar a los tipos de nodos involucrados. En la Figura 6 se observa la relación que se forma entre los nodos al evaluar algún nodo u particular (como nodo principal), donde:

- el nodo p siempre es el nodo ascendente directo (y nulo cuando $u = root$).
- el nodo v es un nodo nulo que puede ser creado.
- el nodo v' es un nodo existente (que puede tener o no espacio disponible para insertar).
- el nodo w es un nodo no-adyascente a u y que no es necesario de considerar para ningún algoritmo.
- dentro del nodo p a cada clave, en la posición p_i, le corresponde un puntero descendente. Es decir, en la Figura 6 se puede crear un nodo en la posición $p_i = 0$, el nodo u está en $p_i = 1$, y así consecutivamente.

Figura 6: Relación entre nodos para un W-tree (k=5).



Los algoritmos que serán descritos a continuación también contarán con un breve análisis asintótico que ayuda a identificar ventajas teóricas en la construcción de la estructura. Para esto es necesario hablar primero del peor caso de construcción de un W-tree.

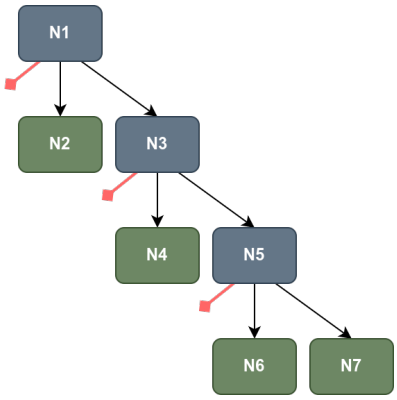
Luego, y a pesar de que para buscar claves se deben haber insertado con anterioridad, primero se describirá el funcionamiento del algoritmo de búsqueda, ya que también es utilizado por la inserción y eliminación.

Peor caso de construcción Dejando de lado la manera en la cual se ordena la estructura (gracias a sus algoritmos de inserción), las restricciones establecidas en la heurística propuesta permiten identificar claramente al peor caso de construcción, que corresponde a un patrón repetitivo.

Este peor caso ocurre cuando se ingresan las claves de forma ordenada, al igual que ocurre en un BST. Sin embargo, gracias a la amortiguación horizontal, bajo cualquier parámetro k se crearán al menos dos nodos por nivel antes de crear descendencia.

En la fig. 7 se observa una representación de un W-tree con parámetro $k = 4$ creado insertando claves en orden ascendente. Los nodos internos están coloreados de gris (N1, N3, N5) y los nodos hojas son verdes (N2, N4, N6, N7). Los punteros nulos que no son ocupados se representan con las flechas rojas que no conectan con ningún nodo.

Figura 7: Peor caso para un W-tree con $k = 4$.



Siguiendo la lógica de este peor caso, su altura está acotada por una complejidad de:

$$W_{altura} = O(\frac{nodos - 1}{2} + 1) = O(\frac{\lceil \frac{n}{k} \rceil - 1}{2} + 1) = O(\lceil \frac{n}{2 * k} \rceil) = O(\frac{n}{k}) \tag{4}$$

Como esta cota es lineal y se aleja del comportamiento logarítmico que se consigue en un BST, surge la idea de una modificación a la heurística de inserción en nodos internos, donde la posición de continuación se calcule respecto a la posición proporcional a los mínimos y máximos del nodo (utilizando una simple regla de tres). Sin embargo, esto no fue incluido en esta investigación.

Para cualquier otro caso, la altura

$$h(n) = O(\log_k n) \tag{5}$$

4.1. Búsqueda

La búsqueda de una clave es la operación más simple del W-tree, pues además es facilitada por la heurística de ordenamiento que establece que *todo nodo contiene, en los extremos de su arreglo de claves, la mínima y máxima clave del subárbol que conforma siendo raíz.*

El algoritmo entonces se reduce a una búsqueda simple, sin recursión, que parte desde la raíz del árbol, moviendo un único puntero que ha sido determinado como nodo u. Primero se pregunta si la clave buscada está dentro del rango formado por los extremos del arreglo de claves de u, resultando en una importante optimización orientada a identificación (o rechazo) prematuro de claves ausentes en la estructura. En la misma línea, gracias a la administración del caché a nivel CPU la revisión secuencial de los arreglos es más rápida que lo intuitivo.

Algorithm search(key: number): Busca existencia de *key* sin utilizar recursividad.

Input: key: clave a buscar
Output: Boolean: $True$ si es que la *key* fue encontrada.

```
1  u = root
2  while u ≠ nullpointers do
3  |  if key < u.values[1] or key > u.values[u.n] then
4  |  |  return False
5  |  i = 1
6  |  while i < u.n and key > u.values[i] do
7  |  |  i = i + 1
8  |  if u.values[i] == val then
9  |  |  return True
10 |  u = u.pointers[i − 1]
11 return False
```

Adicionalmente se puede extender este algoritmo para hacer seguimiento de las variables resultantes involucradas en la búsqueda, como p, pi (la posición de u en p), u e i. Esto es especialmente útil para la eliminación de una clave.

Para realizar una lectura ordenada de los valores en una estructura de datos se suele utilizar el patrón de iterador, que corresponde a una sub-clase que contiene la coherencia necesaria para moverse a través de la estructura. Algunos de ellos incluso mantienen su coherencia frente a nuevas inserciones o eliminaciones.

La forma de recorrer un W-tree es comenzando por el índice (i) más-izquierdo del arreglo de claves del nodo raíz. Para avanzar primero se revisa si es que se puede continuar con el decendente de la misma posición i, y en caso de no haber descendencia se aumenta la posición i. En la Fig. 4 se puede observar un recorrido ascendente de los valores, comenzando desde la raíz y siguiendo la línea punteada azul.

Cuando se llega al final del arreglo se debe ascender nuevamente, por lo que antes de descender se debe guardar hacer seguimiento del nodo por el cual se descendió, además de las correspondientes posiciones utilizadas en el proceso. Esto basta con ser implementado como stacks y ser almacenados por el iterador. Gracias a la k-aridad estos stacks debieran ser relativamente pequeños.

Es por esto que también se implementaron las funciones que guardan los nodos y posiciones que correspondan. La función **find(p, pi, u, ux)** es utilizada por la eliminación pues se deben realizar operaciones sobre el nodo encontrado. La función **findWithPath(stack_p, stack_pi, u)** es similar, pero se encarga de "recordar" el camino hacia el nodo, lo que a su vez permite la creación de un iterador válido.

4.2. Inserción

Dado que la estructura plantea una nueva heurística de ordenamiento, es necesario definir una metodología que, de manera determinista, identifique un **nodo u válido** (al cual se hará referencia a lo largo de las próximas secciones), donde debe ser insertada la clave deseada.

Los algoritmos consideran una **Inserción "Amortiguada"**, donde "amortiguada" se refiere a que evitan el crecimiento de altura y la creación de nodos internos (que contienen punteros hacia su

descendencia) lo más posible. Esto implica que un nodo hoja u, distinto a la raíz, se volverá interno solo cuando no sea posible amortiguar la inserción con algún posible nodo adyacente dentro de p, como lo serían los nodos v o v' en la Figura 6.

Reglas de inserción Con estos antecedentes, ya es posible definir una serie de reglas y/o prioridades cuya consistencia se hagan cargo de todas las posibles necesidades del algoritmo al momento de insertar.

Esto se ha logrado sintetizar en una jerarquía de 5 reglas priorizadas, que permiten la construcción de un algoritmo recursivo estilo **insert-key**(p, u, key), donde u es el nodo a ser analizado, p es el nodo ascendente que existe cuando u no es la raíz, y key es la clave a ser insertada en la estructura.

Cuadro 7: Resumen de Reglas de inserción

N°	Resumen	Definición
R1	Reemplaza extremos de algún nodo interno lleno	Sea $u.isInternal = True$ (y $u.n = k$), y se agrega una clave tal que ($key < values[1]$ o $key > values[n]$), se reemplaza tal extremo, y la clave reemplazada es reinsertada en el mismo nodo, de manera que el algoritmo busque algún descendiente válido.
R2	El nodo tiene espacio	Sea $u.n < k$, se agrega y ordena la clave en el nodo.
R3	SplitSide: Divide hacia el lado	Sea $u.n = k$, u descendente de p, y $\exists v = nullpointer$ adyacente a u en p, se creará v con la mitad de los valores de u, y se realizarán los corrimientos necesarios en p para mantener coherencia.
R4	SlideSide: Corre de u a v	Sea $u.n = k$, u descendente de p, y $\exists v \neq nullpointer$ adyacente a u en p, tal que $v.n < k$, y de acuerdo a su posición (izquierda o derecha) moverá el menor o mayor valor de u resultante tras agregar la nueva clave, manteniendo a u lleno y como nodo hoja.
R5	El nodo debe crearse	Sea $u = nullpointers$, se crea u, se inserta la clave y se vincula a p en caso de existir.

Más en específico, **las 5 reglas de inserción son:**

R1 Se necesita actualizar la primera clave cuando u está lleno y $key < u.values[1]$ (o análogamente para $key > u.values[k]$). Entonces, hacemos $x = u.values[1]$, $u.values[1] = key$ y continuamos insertando x en el primer descendiente de u (que puede o no ser nulo), lo que es equivalente a un llamado a **insert-key**($p = u$, $u = u.pointers[1]$, $key = x$).

Cuadro 8: Complejidad Algorítmica R1

Descripción	Complejidad Algorítmica
Buscar u	O($k * h(n)$)
Reemplazar en u	O(1)
Insertar x	O(insertar(x, from=u)) = O($k * h(n)$)
Total:	O($2 * k * h(n) + 1$) = O($k * h(n)$)

Las restantes 4 reglas corresponden a la inserción de key en una hoja $u = p.pointers[i]$ con su padre p lleno, cuya posición es tal que se cumple: ($p.values[i] < key$ AND $key < p.values[i+1]$).

R2 Si $u \neq NULL$ y tiene espacio ($n < k$), entonces se inserta actualizando $n = n + 1$. Esto provocará que $u.values$ tenga que actualizar su capacidad ($minK \leftarrow 2 * minK$)hasta haber alcanzado su límite, pero no es considerado como parte del algoritmo.

Cuadro 9: Complejidad Algorítmica R2

Descripción	Complejidad Algorítmica
Buscar u	O($k * h(n)$)
Insertar en u	O(k)
Total:	O($k * h(n) + k$) = O($k * h(n)$)

Si la hoja u está llena ($n = k$), entonces tenemos los siguientes dos casos (R3 y R4) que se chequean respetando la prioridad:

R3 **Split-Side:** Si existe una hoja $v = NULL$ y directamente adyacente a u (ya sea a la izquierda o la derecha), entonces se aplica la operación **Split-side**(p, $u = p.pointers[i]$, $pi = i$, key), la cual necesita conocer la posición (i) de u dentro de p, en caso de existir.
Para el caso de la derecha, se sube la clave del centro del conjunto $\{u.values \cup key\}$ tras ser ordenado, y se crea una hoja a la derecha de u que guarda las $k/2$ claves mayores. Esto se define de forma análoga para el sentido izquierdo.

Cuadro 10: Complejidad Algorítmica R3

Descripción	Complejidad hacia la izquierda	Complejidad hacia la derecha
Buscar u	O($k * h(n)$)	O($k * h(n)$)
Ordenar u	O(k)	O(k)
Insertar en v	O($\frac{k}{2} + \frac{k}{2}$)[1]	O($\frac{k}{2}$)
Total:	O($2 * k + k * h(n)$) = O($k * h(n)$)	O(1,5 $* k + k * h(n)$) = O($k * h(n)$)

[1] Para el caso de la izquierda, el tomar los $\frac{k}{2}$ menores valores implica mover el resto hacia las primeras posiciones del arreglo.

Se evidencia entonces que realizar la división hacia la derecha es más efectivo que realizarlo hacia la izquierda.

R4 **Slide-Side:** Hay espacio en una hoja v no nula y que es directamente adyacente a u (ya sea a la izquierda o la derecha). Para el caso en que $v = p.pointers[i-1]$ está a la izquierda de u, primero se ordena $\{u.values \cup key\}$. Luego la menor clave de ese conjunto es comparada con $p.values[i]$, para así insertar a la menor entre ambas a v y la otra dejarla en p. Llamaremos a esta operación **Desplazamiento a la Izquierda** en u, y equivalentemente, **Desplazamiento a la derecha**.
Es importante notar que, por ejemplo para el caso de la izquierda, la implementación no realiza esta primer ordenamiento de u con key, sino que la compara con la mínima clave de u para saber si la clave terminará o no dentro de u.

Cuadro 11: Complejidad Algorítmica R4

Descripción	Complejidad hacia la izquierda	Complejidad hacia la derecha
Buscar u	$O(k * h(n))$	$O(k * h(n))$
Insertar en u	O(k)	O(k)
Insertar $p.values[x]$ en v	O(1)	O(k)
Total:	$O(k + k * h(n))$	$O(2k + k * h(n))$

Se observa entonces que, si bien ambos tienen una complejidad lineal, realizar esta operación hacia la izquierda reduce la cota al menos la mitad de las operaciones.

R5: De no caer en ninguno de los casos anteriores se crea un nuevo nodo v y se enlaza con p en caso de existir, lo que corresponde a una operación de coste constante. Es decir, aquí lo único influyente es la búsqueda del nodo u válido.

Cuadro 12: Complejidad Algorítmica R5

Descripción	Complejidad Algorítmica
Buscar u	$O(k * h(n))$
Crear y enlazar u	O(1)
Total:	$O(k * h(n))$

De acuerdo con lo expuesto e identificado del análisis de complejidad asintótico, se ha establecido que el algoritmo de inserción siempre:

- Mantiene los nodos internos con $n = k$ claves.
- Prioriza el **split-side** a la derecha, así no realiza corrimientos en u.
- Prioriza el **slide-side** a la izquierda, ya que hacia la derecha siempre se inserta una clave en el borde izquierdo del vector de v, lo que provoca un corrimiento de todas las claves.
- Crea al menos dos nodos descendientes por nivel para su peor caso que sucede cuando se insertan los elementos en orden. Esto es porque el primer puntero y el último solo tienen un posible nodo adyacente. Para el resto de casos, los niveles suelen crear al menos 3 nodos antes de descender.

Junto a toda esta información se definen los siguientes pseudocódigos de todas las operaciones involucradas en la inserción de claves en la estructura.

Algorithm Insert(p: nodo, u: nodo, pi: index, key: number): Inserta key dentro del árbol.

Input: p: nodo ancestro, u: nodo actual, pi: índice de u dentro de p, key: clave a insertar
Output: Boolean: $True$ si key fue insertado

```
1  if u == nullpointers then
2  |  u = newNodo(key)                          // R5: Create new node
3  |  if p == nullpointers then
4  |  |  u.isInternal = True
5  |  |  root = u
6  |  else
7  |  |  p.pointers[pi] = u
8  |  return True
9  if u.n < u.k                                 // R2: Leaf node with space
10 then
11 |  return insertOnNode(u, val)
12 if u.isInternal == True                      // R1: Swap laterals
13 then
14 |  if key ≤ u.values[1] then
15 |  |  if key == u.values[1] then
16 |  |  |  return False
17 |  |  swap(key, u.values[1])
18 |  if key ≥ u.values[u.n] then
19 |  |  if key == u.values[u.n] then
20 |  |  |  return False
21 |  |  swap(key, u.values[u.n])
22 |  pi = getPosition(u, key)                  // Keep looking for u
23 |  if pi < u.n and key == u.values[pi + 1] then
24 |  |  return False
25 |  return insert(u, u.pointers[pi], pi, key)
26 if contains(u, val) == True then
27 |  return False
28 if trySplit(p, u, pi, key) == False          // R3: Split to side
29 then
30 |  if trySlide(p, u, pi, key) == False      // R4: Slide to side
31 |  then
32 |  |  u.isInternal = True
33 |  |  return insert(p, u, pi, key)          // Continue as internal node
34 return True
```

Algorithm trySplit(p: nodo, u: nodo, pi: index, key: number): Intenta dividir valores de u con algún vecino nulo.

Input: p: nodo ancestro, u: nodo actual, pi: índice de u dentro de p, key: clave a insertar

Output: Boolean: $True$ si es que pudo dividir el nodo.

```
1  m = ⌊u.k/2⌋
2  rightIsValid = (pi < u.k − 1) and (p.pointers[pi + 1] == nullpointers)
3  if rightIsValid == True then
4  |  temptop = p.values[pi + 1]
5  |  moveRightmost = bool(key < u.values[u.n])
6  |  if moveRightmost == True then
7  |  |  insertOnNode(v, u.values[u.n])
8  |  |  u.n = u.n − 1
9  |  |  insertOnNode(u, key)
10 |  p.values[pi + 1] = u.values[m]
11 |  for i = m + 1 to u.n do
12 |  |  insertOnNode(v, u.values[i])
13 |  if moveRightmost == False then
14 |  |  insertOnNode(v, key)
15 |  insertOnNode(v, temptop)
16 |  p.pointers[pi + 1] = v
17 |  return True

18 leftIsValid = (pi > 1) and (p[pi − 1] == nullpointer)
19 if leftIsValid == True then
20 |  insertOnNode(v, p.values[pi])
21 |  if key < u.values[1] then
22 |  |  insertOnNode(v, key)
23 |  else
24 |  |  insertOnNode(v, u.values[1])
25 |  |  u.values[1] = key
26 |  |  sort(u)
27 |  for i = 1 to m − 1 do
28 |  |  insertOnNode(v, u.values[i])
29 |  p.values[pi] = u.values[m − 1]
30 |  sortAndResize(u)
31 |  p.pointers[pi − 1] = v
32 |  return True
33 return False
```

Algorithm trySlide(p: nodo, u: nodo, pi: index, key: number): Intenta correr uno de los valores de u hacia algún vecino con creado y con espacio.

Input: p: nodo ancestro, u: nodo actual, pi: índice de u dentro de p, key: clave a insertar
Output: Boolean: $True$ si es que pudo realizar el corrimiento.

```
1  leftIsValid = (pi > 1) and
     (p.pointers[pi − 1].n < p.pointers[pi − 1].k)
2  if leftIsValid == True then
3  |  v = p.pointers[pi − 1]
4  |  insertOnNode(v, p.values[pi])
5  |  if key > u.values[1] then
6  |  |  p.values[pi] = u.values[1]
7  |  |  u.values[1] = key
8  |  |  sort(u)
9  |  else
10 |  |  p.values[pi] = val
11 |  return True

12 rightIsValid = (pi < u.k − 1) and
     (p.pointers[pi + 1].n < p.pointers[pi + 1].k)
13 if rightIsValid == True then
14 |  v = p.pointers[pi + 1]
15 |  insertOnNode(v, p.values[pi + 1])
16 |  if key < u.values[u.n] then
17 |  |  p.values[pi + 1] = u.values[u.n]
18 |  |  u.n = u.n − 1
19 |  |  insertOnNode(u, key)
20 |  else
21 |  |  p.values[pi + 1] = val
22 |  return True
23 return False
```

4.3. Eliminación

Al igual que la inserción, la eliminación es un proceso que puede realizarse de muchas maneras, y la solución óptima también depende de la granularidad con la que se defina.

Cuando se elimina alguna clave en un nodo hoja u, basta una pequeña corrección de ordenamiento en u de coste O(k). En el caso de eliminar una clave de un nodo interno, el problema a resolver es la búsqueda de algún nodo hoja que puede ascender un sustituto.

Si bien la optimización de esto implica la *búsqueda del camino más corto hacia una hoja*, se optó por una solución más simple considerando que esta operación será la menos utilizada. Bastará con buscar primero por los punteros a la izquierda de la posición de eliminación (ya que las operaciones de sustitución son más baratas computacionalmente), y de no encontrar nada se continuará hacia la derecha.

Para el desarrollo de la función de eliminación es útil definir dos funciones con anticipación, que se encargan de obtener la máxima y mínima clave de un sub-árbol, utilizando popMax() y popMin() correspondientemente.

Algorithm PopMax(u: nodo): Obtiene la clave máxima del nodo u, haciéndose cargo de su reemplazo cuando u es interno.

Input: u: nodo actual
Output: maxval: Valor máximo de u, nulo cuando no hay claves.

```
1  if u == null then
2  |  return NULL
3  maxval = u.values[u.n]
4  if u.isInternal == False then
5  |  u.n = u.n − 1
6  |  return maxval

7  i = u.n
8  while i > 1 and u.pointers[i − 1] == null do
9  |  i = i − 1
10 if i == 1 then
11 |  u.isInternal = false
12 |  u.n = u.n − 1
13 |  return maxval
14 v = u.pointers[i − 1]

15 for j = u.n to i + 1 by -1 do
16 |  u.values[j] = u.values[j − 1]
17 u.values[i] = PopMax(v)
18 if v.n == 0 then
19 |  u.pointers[i] = null
20 return maxval
```

Para esto deben realizar la búsqueda de los nodos hojas más-izquierdo o más-derecho, y los corrimientos necesarios en cada nodo del camino de regreso. La cantidad de corrimientos será de al menos 1 clave cada vez que no se baje por el primer o último puntero del nodo.

Cuadro 13: Complejidad Algorítmica del peor caso para PopMax().

Descripción	Complejidad
Descender hasta hoja v	O($k * h(n)$)
Obtener máximo de v	O(1)
Ascender corrigiendo máximo	O($k * h(n)$)
Tiempo algorítmico total	O($2k * h(n)$)
	O($k * h(n)$)

De acuerdo con lo presentado en la tabla 13, está claro que la mayor parte del procesamiento de popMax() se invierte en encontrar el nodo hoja correspondiente, pues obtener el máximo de un nodo hoja requiere tiempo constante O(1).

Algorithm PopMin(u: nodo): Obtiene la clave mínima del nodo u, haciéndose cargo de su reemplazo cuando u es interno.

Input: u: nodo actual
Output: minval: Valor mínimo de u, nulo cuando no hay claves.

```
1  if u == null then
2  |  return NULL
3  minval = u.values[1]
4  if u.isInternal == False then
5  |  for i = 1 to u.n − 1 do
6  |  |  u.values[i] = u.values[i + 1]
7  |  u.n = u.n − 1
8  |  return minval

9  i = 1
10 while i < u.n and u.pointers[i] == null
   do
11 |  i = i + 1
12 if i == u.n then
13 |  u.isInternal = false
14 |  for i = 1 to u.n do
15 |  |  u.values[i] = u.values[i + 1]
16 |  u.n = u.n − 1
17 |  return minval
18 for j = 1 to i − 1 do
19 |  u.values[j] = u.values[j + 1]
20 u.values[i] = PopMin(u.pointers[i])
21 if u.pointers[i].n == 0 then
22 |  u.pointers[i] = null
23 return minval
```

El objetivo de este algoritmo es similar al algoritmo anterior, pero sus operaciones son realizadas hacia el otro sentido.

Cuadro 14: Complejidad Algorítmica del peor caso para PopMin():

Descripción	Complejidad
Descender hasta hoja v	O($k * h(n)$)
Obtener mínimo de v	O(k)
Ascender corrigiendo mínimo	O($k * h(n)$)
Tiempo algorítmico total	O($k + 2k * h(n)$) O($k * h(n)$)

De acuerdo con la tabla 14, se observa que, si bien ambos algoritmos mantienen una complejidad algorítmica bastante similar, $PopMin$ agrega una pequeña complejidad de O(k) corrimientos. Esto se debe a que computacionalmente se suele trabajar con las direcciones que apuntan al comienzo de los arreglos de claves y punteros, y por ende remover el mínimo requiere mover todo el resto.

A priori, esto indica que **una preferencia por utilizar $PopMax$ sería beneficiosa**.

Es necesario acotar que cuando estas funciones dejan un nodo vacío no lo desconectan de su ancestro, sino que le dejan esa responsabilidad a la función de eliminación.

Por otro lado, la cantidad de corrimientos que deben hacerse durante el camino sugieren que, si se busca mejorar la optimalidad de la eliminación, podrían invertirse recursos computacionales a la hora de insertar de manera que los nodos descendientes creados sean ubicados en los extremos izquierdo o derecho, de manera que los caminos a las hojas generados por estas operaciones tiendan a solo modificar sus mínimas o máximas claves.

Algorithm Remove(root: nodo, val: clave): Busca la *clave* dentro del W-tree partiendo desde *root*, y en caso de encontrarla, la elimina.

Input: root: nodo raíz del W-tree
Output: bool: True si es que la clave fue eliminada correctamente.

```
1  p, pi, ix = null                          // Create variables
2  u = WTree.root
3  if find(val, p, pi, u, ix) == False then
4  |  return False
5  if u.n = 1 then
6  |  if p ≠ null then
7  |  |  p.pointers[pi] = null
8  |  else
9  |  |  root = nullpointers
10 |  return True
11 if u.isInternal == True  // First look to the left
12  then
13 |  keypos = ix
14 |  while ix > 1 and
   |   u.pointers[ix − 1] == nullpointers do
15 |  |  i = i − 1
16 |  if ix ≠ 1 then
17 |  |  v = u.pointers[ix − 1]
18 |  |  for jx = keypos to ix by -1 do
19 |  |  |  u.values[jx] = u.values[jx − 1]
20 |  |  u.values[ix] = PopMax(v)
21 |  |  if v.n = 0 then
22 |  |  |  u.pointers[ix] = null
23 |  |  return true
   |                                  // Then look to the right
24 |  ix = keypos
25 |  while ix < u.n and u.pointers[ix] == nullpointers
   |   do
26 |  |  i = i + 1
27 |  if ix ≠ u.n then
28 |  |  v = u.pointers[ix]
29 |  |  for jx = keypos to ix − 1 do
30 |  |  |  u.values[jx] = u.values[jx + 1]
31 |  |  u.values[ix] = PopMin(v)
32 |  |  if v.n = 0 then
33 |  |  |  u.pointers[ix] = null
34 |  |  return true
   |                        // Then the node must be corrected
35 |  u.isInternal = false
36 |  ix = keypos
37 while ix < u.n do
38 |  u.values[ix] = u.values[ix + 1]
39 |  i = i + 1
40 u.n = u.n − 1
41 return true
```

El algoritmo de eliminación utiliza las funciones $find()$, $popMax()$ y $popMin()$, y en conjunto logran hacerse cargo de esta operación sin demasiada complejidad.

La función $find()$ siempre retornará la posición correctamente ordenada de inserción, y su adaptación se encarga de guardar los nodos e índices pertinentes en variables que pueden ser utilizadas posteriormente.

La única decisión que toma este algoritmo es desde qué nodo descendiente se realiza el $popMax()$ o $popMin()$, y como se ha explicado, no es más que una búsqueda secuencial hacia los lados, priorizando el lado izquierdo de acuerdo a la ventaja de eficiencia de $popMax()$.

Los resultados empíricos de esta decisión deben ser analizadas con mayor detalle y de una forma más estadística, pues puede ocurrir que, dada alguna distribución en particular de los datos, los algoritmos utilizados para su creación provoquen que los caminos hacia la derecha (en busca de utilizar $popMin()$) sean más cortos que los que llevan a la izquierda.

La simplicidad de este algoritmo provoca a veces que un nodo mantenga su atributo de interno a pesar de no tener descendencia. Esto es considerado en todos los algoritmos presentados en este trabajo y basta con realizar una corrección de este atributo. La memoria asignada al vector de punteros no es reducida bajo la suposición de que el nodo se puede volver a llenar con facilidad (solo ha sido removido un elemento), y también considerando que los punteros son las variables que representan el mayor peso de un nodo.

5. Implementación

La implementación se desarrolló en C++17 en busca de tener un control de bajo nivel del funcionamiento de los algoritmos, lo que repercute directamente en su rendimiento empírico y consecuentemente en los resultados comparativos respecto a otras estructuras que también han sido implementadas en este lenguaje.

En una primera etapa se priorizó el correcto funcionamiento de los algoritmos, para posteriormente ser optimizados de acuerdo a los hallazgos de los análisis teóricos y empíricos.

La versión final integra los contenedores *vector* de la librería estándar de C++, además de los algoritmos de $find\_if\_not()$ y $move()$ de la librería *algorithm*, que además pueden trabajar con iteradores o punteros nativos.

Dentro del Proyecto de Implementación también se incluyen:

- programas C++ de testeo que ponen a prueba y validan el correcto funcionamiento de todas las operaciones bajo todas sus restricciones.
- programas C++ parametrizables de monitoreo automático de memoria y tiempos de ejecución de todas las operaciones, cuyos datos son registrados en archivos CSV para su posterior análisis. Esto también incluye scripts para su ejecución remota utilizando Slurm.
- programas Python (Jupyter) que permiten la lectura y análisis de los datos monitorizados, y así crear automáticamente gráficos comparativos.

A nivel de programación fue necesario investigar sobre las diversas abstracciones que son parte de un diseño ordenado, modular y consistente de algoritmos en este lenguaje. La solución final cuenta con la definición de clases, punteros, plantillas (templates), destructores y operadores. Aún así, las librerías de uso público integran la definición de iteradores y otro tipo de operadores que siguen un estándar (como para copias e inicializaciones), lo que no fue necesariamente abordado en este trabajo.

Para los experimentos se implementó una clase generadora de valores pseudo-aleatorios con distintas distribuciones, que guarda registro booleano de las claves ya existentes, lo que permite generar valores presentes o ausentes a voluntad.

6. Experimentación

Para poner a prueba el rendimiento empírico del W-tree bajo diversos parámetros k, se comparará con otras estructuras en una prueba separada en etapas secuenciales que utilizan las distintas funciones y miden su desempeño.

Las estructuras utilizadas para la comparación son:

- **BST:** Una implementación básica hecha desde cero para las comparaciones.
- **RB-tree:** Un árbol rojo-negro, utilizado por el contenedor *set* de C++ al compilar con gcc.

El entorno de desarrollo se contó con el siguiente hardware para la experimentación:

CPU: 1x AMD Ryzen 5 2600 Six-core 3.4 GHz
RAM: 2x Kingston KHX3200C16D4 8GX 2400 MHz
SSD: 1x Kingston SKC2500M8 1000G - Read: 3500 MB/s - Write: 2900 MB/s

Para pruebas de mayor tamaño se utilizó el supercomputador Patagon[13] de la Universidad Austral de Chile:

CPU: 2x AMD EPYC 7742 CPU 2.6GHz, 64-cores, 256MB L3 cache
RAM: 1x DDR4 - 1TB 3200 MHz
SSD: 1x NVMe PCIe 4.0 (NFS Cache) 15TB
SSD: 1x NVMe PCIe 4.0 (System Storage) 2TB

Etapas de experimentación Corresponden a 5 etapas, y el resumen de sus funciones se puede observar en la tabla 15:

Cuadro 15: Tabla síntesis de las etapas del experimento.

Etapa	**Operación**	**Tamaño**
1. Inserción desde cero	Inserción	n
2. Inserción poblada	Inserción	$\frac{n}{4}$
3. Búsqueda de claves existentes	Búsqueda	30000
4. Búsqueda de claves inexistentes	Búsqueda	30000
5. Eliminación	Eliminación	$\frac{n}{4}$

Las primeras dos etapas corresponden a las inserciones de claves, donde la segunda permite medir los tiempos una vez la estructura está poblada.

La búsqueda de claves existentes es la operación más parecida al resto de estructuras. Fue separada de la búsqueda de claves inexistentes para así tener mediciones *aisladas* del rendimiento.

Como la eliminación es una operación de baja prioridad dentro de esta investigación, nos basta con medir su rendimiento en comparación al resto.

Parámetros de experimentación Para la experimentación se cuenta con una serie de parámetros a configurar. Los valores utilizados para los parámetros se observan en la tabla 16.

Si bien el tamaño n de valores que son insertados a la estructura puede variar, nos interesa estudiar su comportamiento asintótico en comparación a las otras estructuras.

Las ejecuciones de los resultados fueron realizadas en el Patagón[13] utilizando $n = 2^{28}$, lo que ocupa $\sim$ 18GB de memoria RAM.

La distribución de las claves ingresadas de input a las diversas funciones corresponde a una distribución normal que genera valores en todo el dominio de claves positivas de tipo int (considerando entonces la macro RAND_MAX de librería estándar).

El parámetro k es el único que varía en la experimentación. Comienza en 8 y toma hasta la máxima potencia de 2 que es almacenable en un tipo usigned short (2 bytes). Es por esto que, para cada parámetro k, fue repetida la experimentación con 10 semillas distintas.

Cuadro 16: Tabla de valores de parámetros.

Parámetro	**Valores**
k	$2^i, i = 3, 4, ..., 15$
n	2^{28}
seeds	10
mean	0,5$*$ RANDMAX
std	0,075$*$ RANDMAX

6.1. Comparación de Tiempo

En las siguientes figuras se comparan los tiempos de ejecución (eje de las ordenadas) en función del parámetro k (eje de las abscisas). Para cada estructura se presenta el valor promedio y su correspondiente desviación estándar.

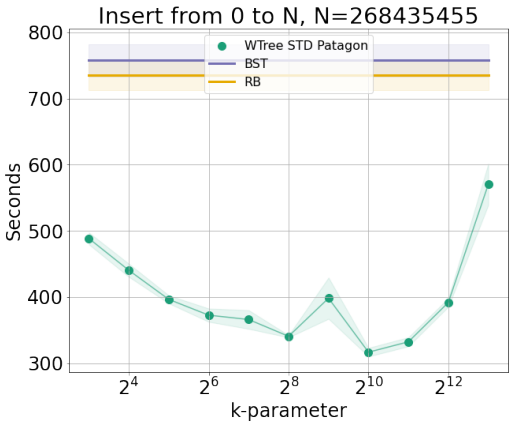
En el caso del W-tree, las mediciones (los puntos verdes) han sido conectados para mejorar la legibilidad. Las otras estructuras se observan en forma recta porque no dependen del parámetro k, es decir, corresponden a una única media y desviación estándar. A priori se esperaría que el RB siempre tenga mejores resultados que el BST, pero como se comentó en la sección 2 esto no necesariamente es cierto empíricamente.

Junto al análisis de cada etapa, se indicará el parámetro k de mejor resultado acompañado del speedup[6] respecto al desempeño del RB-tree.

Figura 8: Resultados de Etapa 1

<u>Etapa 1</u>: En esta primera etapa se insertan claves partiendo con la estructura vacía, y se observa que los resultados del BST y el RB se traslapan en pequeña proporción, lo que posiblemente implica que hay casos donde uno u otro es el de mejor rendimiento.

Los resultados del W-tree son siempre mejores. El mejor se obtiene con $k = 1024$, consiguiendo un speedup de x2,32. Le sigue $k = 2048$ con un x2,21.

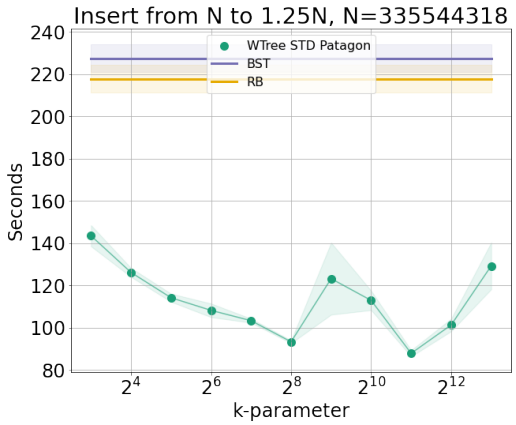


[6] El speedup es la relación entre el tiempo tomado por la estructura a comparar, divido el tiempo tomado por nuestra estructura

Etapa 2: En esta segunda etapa también se insertan elementos pero con la estructura ya poblada. Aquí se observan resultados bastante similares a los anteriores, lo que refleja que el W-tree es eficiente tanto para su creación desde cero como a medio poblar.

Los resultados del W-tree son siempre mejores, e inclusive más que los de la etapa anterior. El mejor se obtiene con $k = 2048$, consiguiendo un speedup de $x2{,}48$. Le sigue $k = 256$ con un $x2{,}33$.

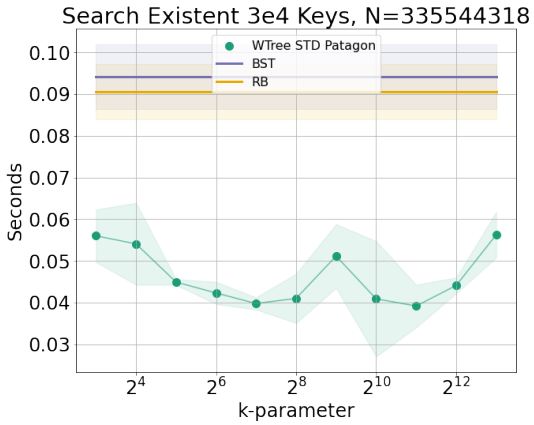
Figura 9: Resultados de Etapa 2



Etapa 3: La búsqueda de claves existentes muestra que los resultados obtenidos por el W-tree se alejan menos de las estructuras de comparación. A su vez, su desviación estándar parece comportarse más erráticamente, pero es algo que también se observa en las otras estructuras. Se asumirá como un efecto propio de la función búsqueda.

Los resultados del W-tree son siempre mejores. El mejor se obtiene con $k = 2048$, consiguiendo un speedup de $x2{,}31$, y al parecer el comportamiento es tan errático que se le asemejan bastante los casos de varios parámetros k, pero posiblemente se debe al reducido número de búsquedas.

Figura 10: Resultados de Etapa 3

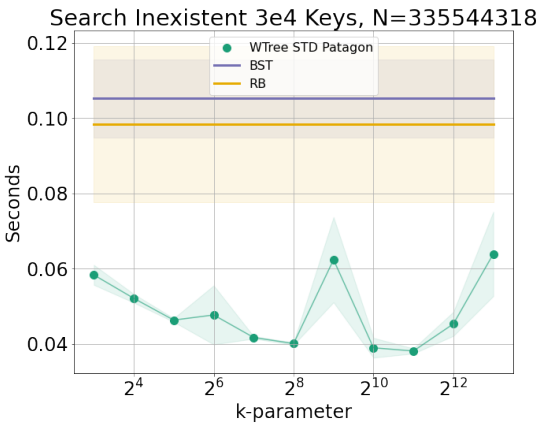


Es importante acotar que el comportamiento de estas curvas es bastante curioso, en específico para $k = 2^9$ pues es un fenómeno presente en todas las gráficas, pero será comentado en las secciones posteriores.

Etapa 4: En la búsqueda de claves inexistentes, las curvas parecen mucho más cercanas a comparación de la figura anterior, pero en realidad se debe a la gran variación en las estructuras tradicionales. Se observa que los tiempos del W-tree son similares a los anteriores pero con mínimos y máximos un poco más pronunciados, pero mucho más estables que las otras estructuras.

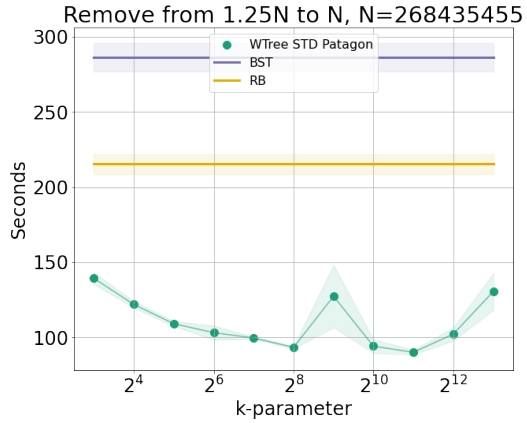
Los resultados del W-tree son siempre mejores. El mejor se obtiene con $k = 2048$, consiguiendo un speedup de $x2{,}58$. Le sigue $k = 1024$ con un $x2{,}52$.

Figura 11: Resultados de Etapa 4



Etapa 5: Dado que el BST es una implementación propia, su eliminación sigue una heurística simple de, bajo una eliminación en nodo interno, unir su descendencia priorizando de raíz al nodo derecho. Esto también valida el buen desempeño de la heurística utilizada por el RB-tree.
En la misma línea, a pesar de que la eliminación no es una operación relevante para esta investigación, esta operación consiguió bastante buenos resultados. Los resultados del W-tree son siempre mejores. El mejor se obtiene con $k = 2048$, consiguiendo un speedup de $x2{,}39$. Le sigue $k = 1024$ con un $x2{,}28$, por lo que podría decirse que el W-tree se mantiene a una relación de ventaja bastante homogénea respecto al RB-tree.

Figura 12: Resultados de Etapa 5

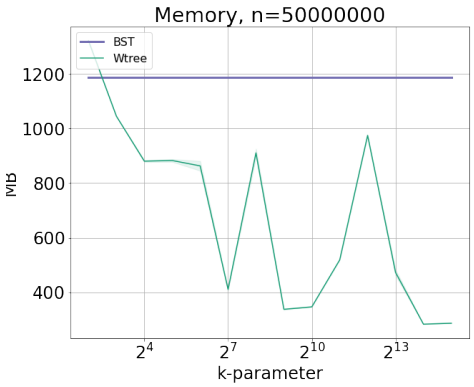


Validando así la Hipótesis 1, se puede afirmar con un 99,99 % de confianza que los tiempos empíricos tomados por el W-tree, comparado al RB-tree, son siempre estocásticamente menores (para cualquier parámetro k).

6.2. Comparación de Memoria

Respecto a la asignación de memoria, el W-tree parte ocupando hasta un 45 % más de memoria con $k = 4$ pero disminuye rápidamente, y a partir de $k = 2^3$ siempre utiliza menos memoria que un BST. Esto se observa en la Figura 13, donde también se identifica el mínimo uso en $k = 2^9$. Cabe destacar que la variación entre las muestras no se alcanza a percibir pues ronda los 5KB.

Figura 13: Uso de memoria para $5*10^7$ claves.



El parámetro k de la estructura, para problemas lo suficientemente grandes, no se comporta como un valor restrictivo. Siempre que sea mayor, mejor será el beneficio en memoria, por lo que deberá analizarse el resto de resultados para llegar a proponer algún parámetro k en especial.

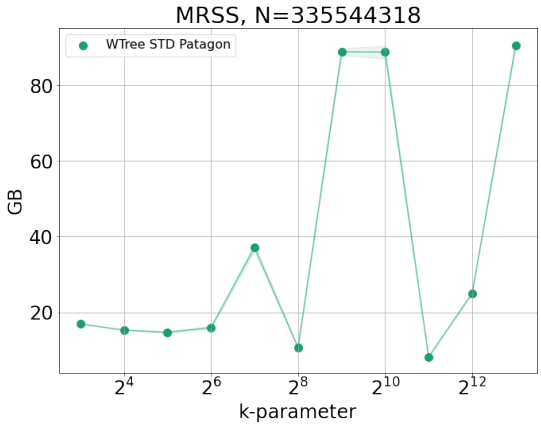
Por otro lado, la memoria requerida tanto por el BST como por el RB-tree es una cifra pseudo-determinista, pues ninguna de aquellas heurísticas considera mecanismos para ahorrar la utilización de punteros. En otras palabras, habiendo demostrado la relación de k-aridad en la Ecuación 3, está garantizado que la estructura utiliza menos memoria que estas estructuras para k pequeños, pero su desempeño para k grandes dependerá de la cantidad de elementos insertados, y por ende, de la ocupación efectiva de punteros descendientes para los nodos internos.

Validando esto con la Hipótesis 2, se puede afirmar con un 99,99 % de confianza que la memoria asignada para el W-tree, comparado con ambas estructuras, son siempre estocásticamente menores dentro de las configuraciones establecidas.

Por otro lado, el monitorio del Maximum Resident Set Size, que es una métrica que indica cuál fue el tamaño máximo de memoria principal utilizado por el proceso, indica que para k entre $2^8 y 2^{11}$ ocurre un fenómeno bastante parecido al de las comparaciones de tiempo, lo que indicaría una clara correlación. Esto se podría relacionar al comportamiento del sistema operativo y/o las CPU's respecto al uso de memoria, que refleja una baja en el rendimiento en $k = 2^9$ porque posiblemente el uso de memoria provoca movimientos hacia la memoria secundaria.

Esto también permite concluir que, mientras más se ocupe de la memoria principal (que es el caso para $k = 2^1 1$), es cuando mejores resultados se consiguen, claramente porque nos ahorramos este movimiento de información.

Figura 14: Comportamiento de Maximum Resident Set Size



7. Aplicación Práctica - Reducción Paralela de m-menores con offset

Dado que la estructura trabaja con arreglos en los nodos, surge la motivación de analizar su rendimiento al aplicar paralelismo a alguna operación sobre estos arreglos. Como generalmente las claves son valores numéricos, una de las principales operaciones que se utiliza es la suma de un rango de elementos, lo que se le denomina *reducción*. Para que la reducción sea aplicable, la operación debe ser conmutativa, es decir, no debe importar el orden en que se realizan las operaciones.

La eficiencia de los algoritmos que resuelven la reducción aritmética tiene directa relación con la arquitectura hardware. Hoy en día se destacan principalmente las soluciones paralelas, las cuales están a disposición de los programadores a través de diversas librerías y lenguajes. En GPU existen un gran número de soluciones a este problema que varían en dificultad, utilizando diversas herramientas de bajo nivel enfocadas en el uso de memoria y barreras (o la familia de conceptos similares a los semáforos).

Para intentar resolver este problema con un W-tree se debe diseñar una heurística que tome ventaja del paralelismo al utilizar arreglos utilizando la librería OpenMP [7]. En esta línea, la premisa de investigación planteada extipula que:

Es posible aplicar una reducción aritmética para un intervalo de $m - menores$ elementos dentro de un universo de n valores a partir de algún offset*.

***:** El offset se refiere a comenzar no necesariamente desde el primer valor de la estructura (que por consecuencia es el menor), sino que se puede comenzar desde la posición i de alguna clave cualquiera, donde $i \leq n$.

Como el W-tree es una estructura de datos ordenada, el recorrido ascendente de sus claves comienza por el extremo izquierdo (posición 1) del arreglo de valores de la raíz. Para avanzar primero intenta descender por el nodo descendiente que se encuentra en la misma posición de la clave (pero ahora en el vector de punteros), y en caso contrario avanza una posición en el arreglo de claves. Si se llega al extremo derecho de este arreglo, se intentará ascender al último nodo por el cual se descendió, hasta eventualmente volver a la raíz. Cuando se llegue al extremo derecho y no existan descendientes, la posición será nula. Este recorrido también ha sido descrito en la Figura 4.

Si consideramos un iterador que se puede mover de forma correcta ascendentemente entre los elementos del W-tree, la solución secuencial al problema anteriormente planteado es bastante simple. Sin embargo, el movimiento de este iterador es imposible de realizar de forma paralela (ya que necesita mantener un contador de los elementos ya reducidos), y tampoco se puede conocer a priori si un árbol contiene *menos que infinitos* elementos (y por ende menos o la misma cantidad de menores que se necesita reducir).

Por otro lado, gracias a la coherencia de este iterador es posible requerir la reducción de los m valores a partir de una ubicación inicial arbitraria, que en general será distinta a la raíz. Para ésto se integró la clase iterador además de la función $findWithPath(key)$, que encuentra alguna key válida y almacena el camino desde la raíz hasta su lugar. Es decir, el iterador integra stacks que guardan los nodos que fueron sus ascendentes (denominado p_stack en la implementación) y las correspondientes posiciones por donde se descendió (pi_stack).

[7] OpenMP: `https://www.openmp.org/`

Heurística de reducción paralela Dada la estructura del W-tree, es posible identificar que, si se conoce la cantidad de elementos (c) que están almacenados en un árbol, y es $c \leq m$ (la cantidad total de sumandos), entonces a cada nodo del árbol se le debe aplicar una reducción de sus valores. De esta manera, el iterador secuencial no realiza las c reducciones sobre este árbol, sino que añade las raíces de estos árboles a una cola que es posteriormente procesada paralelamente.

Si bien la suma de los valores dentro de un nodo es una reducción simple de paralelizar, la revisión del vector de punteros en busca de descendientes válidos no lo es, ya que el iterador debe agregar el puntero descendente a una cola compartida de manera atómica (al menos en la implementación propuesta). También es muy probable que el iterador revise un nodo y encuentre árboles en posiciones adyacentes, por lo que la cantidad de los valores entre su posición anterior y la de siguiente descendencia tenderá a ser un arreglo diminuto que no vale la pena reducir paralelamente.

De acuerdo con lo mencionado, existe una clara intuición a que la implementación paralela se enfoque en organizar múltiples hilos de ejecución donde cada uno revise un nodo en particular. Otra forma de resolver esto sería revisar un único nodo a la vez pero reducir el arreglo paralelamente, sin embargo el parámetro k no es lo suficientemente grande como para conseguir mejores significativas. De todas maneras, ambas variaciones serán puestas frente a comparación. La clase iterador y los algoritmos desarrollados pueden ser revisados en la sección de Anexos 10.2.

Experimentación El speedup se comprende como la relación entre el tiempo tomado por la versión secuencial y la versión paralela. Si es superior a 1.0 entonces existe ventaja de la versión paralela y es mejor mientras más grande sea.

La experimentación considera distintos parámetros de K y de M (tamaño del segmento), con $n = 2{*}10^7$ fijo y utilizando 6 hilos de ejecución, por lo que si nuestro speedup es superior a 6.0 indica que los núcleos se aprovechan más en conjunto que trabajando secuencialmente (o por separado).

A continuación se presentan dos tablas que muestran los mejores speedups respecto a la versión secuencial, conseguidos para alguna combinación de $log_{10}(M)$ y $log_2(K)$ en particular, que corresponden a columnas creadas al aplicar tales funciones a los parámetros M y K.

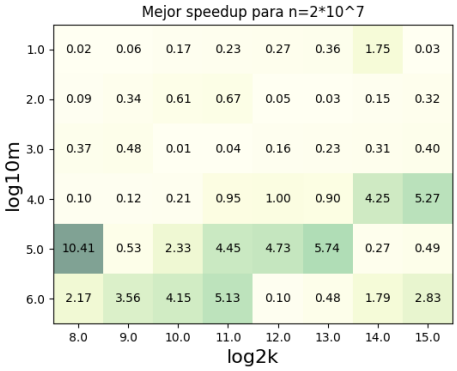


Figura 15: Mejor speedup en versión principal ($n = 2 * 10^7$)

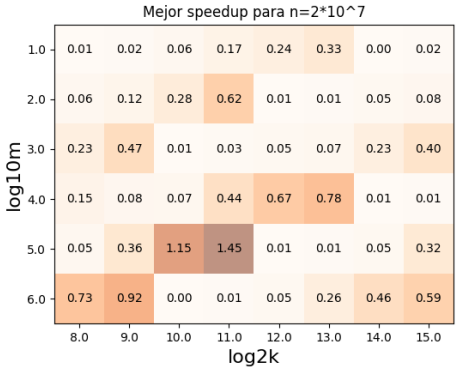


Figura 16: Mejor speedup en versión con omp en Iterator.moveNext() ($n = 2 * 10^7$)

En la Fig. 7 se destaca un speedup positivo de hasta 10.41x comparado al secuencial, y en general estos buenos resultados se consiguen para un $M \geq 100000$ y un $K \geq 2^{11}$. Para el resto, esta heurística suele conseguir un tiempo de ejecución peor que la versión secuencial. Esto también parece indicar que a medida que crezca K se conseguirán mejores resultados.

Los resultados del mismo experimento con la segunda versión paralela, que contemplaba una reducción en la sección no-iterable del movimiento del puntero, se observan en la Fig. 7 y en general son bastante malos. Esto valida la intuición de que los segmentos demasiado pequeños del movimiento secuencial del iterador no valen la pena de ser paralelizados.

Estos resultados validan las suposiciones planteadas antes de la experimentación, indicando que la heurística de recorrido se comporta de manera predecible y analizable. Esto también abre la posibilidad de profundizar en esta implementación, en busca de identificar relevancia del uso de memoria y de las decisiones tomadas por el algoritmo, el cual bajo su versión principal demostró utilizar mejor los recursos que los núcleos por separado (el speedup fue mayor a 6.0).

8. Conclusiones

La revisión del estado del arte de las distintas variaciones del Binary Search Tree inspiró en gran medida la propuesta del W-tree. Para esto se consideró principalmente el concepto de k-aridad y una utilización eficiente de memoria, lo que eventualmente también repercute en los tiempos de ejecución.

En la misma línea, esta relación de aridad nos permite responder nuestra primera pregunta de investigación, frente a la cual podemos afirmar que incluso es posible invertir esta relación, obteniendo valores siempre menores a 1,0, como se mostró en la Ecuación 3.

Para responder la segunda pregunta de investigación, respecto a la eficiencia de una nueva propuesta, es necesario validar las dos hipótesis de investigación planteadas. Tras la experimentación y al utilizar sus registros correspondientes, se validó con un 99.99 % de confianza que los recursos de tiempo y memoria utilizados por el W-tree son estocásticamente menores que los del RB-tree de C++, y por lo tanto, la segunda pregunta de investigación también puede ser respondida afirmativamente con el desempeño del W-tree.

Siempre habrá un beneficio respecto al uso de memoria mientras se aumente k y el tamaño del problema sea lo suficientemente grande, y se consigue lo contrario cuando la cantidad de claves es pequeña.

El fenómeno que ocurre para 2^8 y 2^{10} se repite en todos los gráficos, donde ambos valores corresponden a mínimos bastante cercanos. Los valores adyacentes (hacia el exterior) suelen ser los mínimos globales del experimento. Es aquí donde se evidencia el efecto del tamaño de la memoria caché de las CPU en rendimiento, pues estos valles representan los valores de k que alinean de mejor manera sus *palabras* en la caché, y posiblemente 2^9 es donde todo se desincroniza.

Nuestra experiencia empírica apunta a que el mejor k corresponde a $k = 2^{11} = 2048$, consiguiendo homogéneamente un speedup cercano al x2,4 en comparación al árbol RN.

Respecto a la implementación paralela aplicada, esta es mejorable en varios aspectos, e inclusive motiva a la implementación de otro tipo de heurísticas, como hilos que intenten *predecir* trabajo, o nuevos atributos para los nodos, como lo sería una reducción *lazy* que vaya operando a medida que se insertan valores en el árbol. Aún así, estos resultados son destacables pues el mejor speedup conseguido (10.41x) supera a la cantidad de hilos utilizados (6 hilos).

Otra opción sería realizar la reducción de los valores en GPU mientras hay hilos en CPU recorriendo el vector de punteros, lo cual parece la utilización más óptima de recursos computacionales

para este problema, considerando que además está preparado para la inserción o remoción dinámica de elementos.

Si bien el alcance de esta investigación es acotado, quedan claras oportunidades de variar y mejorar los distintos algoritmos presentados para esta nueva estructura de datos, tanto desde aspectos de programación paralela como de aplicaciones estadísticas.

En el trabajo futuro se pretende adaptar el funcionamiento de esta estructura a la de un "map", una estructura ordenada que permite crear relaciones entre tuplas. Por otro lado, en las últimas etapas de esta investigación se encontró la implementación de un B-tree en memoria principal, cuyo rendimiento empírico supera drásticamente a todas las estructuras analizadas, incluso la propuesta. Un análisis de esta implementación podría aportar seriamente en el rendimiento del W-tree, pero es necesario analizar la compatibilidad entre las técnicas de bajo nivel utilizadas por estos algoritmos.

En conclusión, el W-tree es una buena y nueva alternativa a los contenedores clásicos utilizados en la experimentación científica, los cuales en soluciones existentes pueden ser sustituidos con muy poco esfuerzo.

9. Referencias

1. James D. Anderson, Ryan M. Raettig, Josh Larson, Scott L. Nykl, Clark N. Taylor, and Thomas Wischgoll. Delaunay walk for fast nearest neighbor: accelerating correspondence matching for icp. *MACHINE VISION AND APPLICATIONS*, 33(2), FEB 2022.
2. Rudolf Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 12 1972.
3. Gerth Stølting Brodal and Gabriel Moruz. Skewed binary search trees. In Yossi Azar and Thomas Erlebach, editors, *Algorithms – ESA 2006*, pages 708–719, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
4. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, jan 2008.
5. A. S. Douglas. Techniques for the Recording of, and Reference to data in a Computer. *The Computer Journal*, 2(1):1–9, 01 1959.
6. Jian Feng, Daniel Naiman, and Bret Cooper. A parallelized binary search tree. *Journal of Information Technology & Software Engineering*, 01, 01 2011.
7. G M Adelson-Velskii and E M Landis. An algorithm for the organization of information. 1962.
8. Laurent Hyafil and Ronald L. Rivest. Constructing optimal binary decision trees is np-complete. *Information Processing Letters*, 5(1):15–17, 1976.
9. Tobin J. Lehman and Michael J. Carey. A study of index structures for main memory database management systems. In *Proceedings of the 12th International Conference on Very Large Data Bases*, VLDB '86, page 294–303, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.
10. Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *VISAPP*, 2009.
11. Marius Muja and David G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE*, 36(11):2227–2240, NOV 2014.
12. Navarro, C. A., Carrasco, R., Barrientos, R. J., Riquelme, J. A., Vega, R. Gpu tensor cores for fast arithmetic reductions. *IEEE Transactions on Parallel and Distributed Systems, 32(1)*, page 72–84, 2021.
13. Austral University of Chile. Patagón supercomputer, 2021.
14. Ben Pfaff. Performance analysis of bsts in system software. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, page 410–411, New York, NY, USA, 2004. Association for Computing Machinery.
15. Anton Rigin and Sergey Shershakov. Sqlite rdbms extension for data indexing using b-tree modifications. 31:203–216, 09 2019.
16. Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, jul 1985.
17. David Witmer, Ellango Jothimurugesan, and Ziquiang Feng. Lecture 7: Dynamic programming i: Optimal bsts, Apr 2022.
18. Yuni Xia, Yi-Cheng Tu, Mikhail Atallah, and Sunil Prabhakar. Short paper efficient data compression in location based services. 04 2022.
19. S. A. R. Zaidi. Nearest neighbour methods and their applications in design of 5g & beyond wireless networks. *ICT EXPRESS*, 7(4):414–420, DEC 2021.

10. Anexos

10.1. Sobre NP-completitud

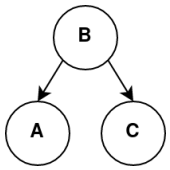
Para explicar esto, supongamos que se tienen las claves A, B y C tales que $A < B < C$. Suponiendo que a priori se logra estimar las probabilidades con las cuales se puede buscar una clave, la forma del árbol influye notablemente en el número total de comparaciones que se deben realizar para buscar todas las claves.

Un ejemplo de esto se observa en la Fig. 17a, que corresponde a un BST donde la clave B fue ingresada primero, y luego ambas claves restantes.

En el cuadro de Fig. 17b se encuentra cada clave junto a una estimación de la probabilidad de consulta (variable p), la cantidad de comparaciones que se realizan en la estructura para encontrar la clave (variable c), y su multiplicación ($p * c$). La última fila corresponde a una sumatoria de estos valores para luego compararlo con otras condiciones de experimentación.

Figura 17: Primer ejemplo de NP-completitud

(a) BST al insertar en orden las claves B, A y C.



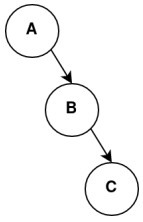
(b) Número de comparaciones ponderadas para buscar A, B y C dentro del BST de la figura 17a.

Clave	% (p)	— (c) —	(p*c)
A	0,6	2	1,2
B	0,3	1	0,3
C	0,1	2	0,2
$\sum_{i=1}^{n}$	1,0	5	1,7

A continuación se observa otro posible BST con la misma asignación de probabilidades pero generada ingresando las claves ascendentemente. La tabla a su lado muestra que, si bien la suma de las comparaciones (c) es mayor al caso anterior, la suma ponderada resulta menor.

Figura 18: Segundo ejemplo de NP-completitud

(a) BST al insertar en orden las claves A, B, C.



(b) Número de comparaciones promedios para buscar A, B y C dentro del BST de la figura 18.

Clave	% (p)	— (c) —	(p*c)
A	0,6	1	0,6
B	0,3	2	0,6
C	0,1	3	0,3
$\sum_{i=1}^{n}$	1,0	6	1,5

Es evidente que, considerando como óptimo el árbol que minimice la cantidad de comparaciones para la búsqueda de una cantidad de elementos asintóticamente grande, el árbol de la Figura 18 es mucho más eficiente que el de la Figura 17, a pesar de que el primer BST se considere óptimo en balance.

10.2. Algoritmos de reducción paralela con offset

Para el movimiento coherente a través de la estructura se utilizó un iterador con los siguientes atributos:

struct WTIterator{

p_stack	stack de nodos ascendentes al actual
pi_stack	stack de posiciones por las cuales se descendió
u	puntero al nodo actual
ux	posición del puntero dentro del nodo

}

La versión secuencial de la reducción es la siguiente:

Algorithm SequentialRedxKMinors(m: number, iterator: WTIterator): Realiza la reducción de los m menores desde la posición actual del iterador.

Input: m: largo de segmento, o cantidad de menores, iterator: un iterador de W-tree, o sus atributos correspondientemente.

Output: total: Acumulación de la reducción de los m menores claves a partir del iterador de input.

```
1  total = 0
2  while m > 0 and u ≠ NULL do
3  |  total = total + value()
4  |  if ux = u.n then
5  |  |  if p_stack.size > 0 then
6  |  |  |  u = p_stack.pop()
7  |  |  |  ux = pi_stack.pop() + 1
8  |  |  else
9  |  |  |  u = NULL
10 |  |  |  ux = ux + 1
11 |  else if u.isInternal = True and u.pointers[ux] ≠ NULL then
12 |  |  p_stack.push(u)
13 |  |  pi_stack.push(ux)
14 |  |  u = u.pointers[ux]
15 |  |  ux = 1
16 |  else
17 |  |  ux = ux + 1
18 |  m = m − 1
19 return total
```

Su versión paralela corresponde a:

Algorithm ParallelRedxKMinors(m: number, iterator: WTIterator, nt: number): Realiza la reducción en paralelo de los m menores desde la posición actual del iterador.

Input: m: largo de segmento, o cantidad de menores, iterator: un iterador de W-tree, o sus atributos correspondientemente, nt: número de threads

Output: total: Acumulación de la reducción de los m menores claves a partir del iterador de input.

```
1  total = 0
2  full_trees                                                  // stack<WTreeNode>
3  visited = 0                                  // contador de elementos ya considerados
   // 1: Recorrer buscando árboles completos
4  while visited < m and u ≠ NULL do
5  |  visited = visited + it.moveNextNode(m − visited, full_trees)
   // 2: Recorrer cada nodo raíz
6  size = full_trees.size
7  while size > 0 do
   |  // 1) pragma omp parallel shared
8  |  th_positions                                  // vector<unsigned short>
9  |  th_total = 0
10 |  th_id = omp_get_thread_num()
11 |  if th_id <= size then
12 |  |  th_node = full_trees[max(size − nt, 0) + th_id]
13 |  |  th_total = sum(th_node.values)                          // suma secuencial
14 |  |  if th_node.isInternal = True then
15 |  |  |  for i = 1 to k − 1 do
16 |  |  |  |  if th_node.pointers[i] ≠ NULL then
17 |  |  |  |  |  th_positions.push(i)
   |  |  // 2) pragma omp critical
18 |  |  total = total + th_total
19 |  |  for pos ∈ th_positions do
20 |  |  |  full_trees.push(th_node.pointers[pos])
   |  // end 2) and 1)
21 |  full_trees.erase(max(size − nt, 0), size)
22 |  size = full_trees.size
23 return total
```