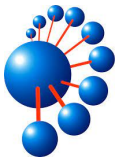
DEPARTAMENTO DE INGENIERÍA INFORMÁTICA
Y CIENCIAS DE LA COMPUTACIÓN
FACULTAD DE INGENIERÍA
UNIVERSIDAD DE CONCEPCIÓN



Implementación de un Dynamic Range Min-Max Tree

por
Matías Ignacio Mora P.

Memoria presentada para la obtención del título de
Ingeniero Civil Informático

Patrocinante: José Sebastian Fuentes S.
Comisión 1: Cecilia Hernández R.
Comisión 2: Perluigi Cerulo.

Concepción, 26 de marzo de 2023

A todas las personas que me acompañaron en este proceso, les dedico este trabajo.

Resumen

Hoy en día, la cantidad de nueva información que se produce crece sin detención, lo que supone un potencial problema a las tecnologías de almacenamientos si estos datos no son representados adecuadamente. Una solución para enfrentar estos potenciales problemas es representar los datos es espacio compacto, al tiempo que se permite consultarlos mientras están representados en esa forma compacta. Esa solución toma forma en el campo de investigación de las denominadas *estructuras de datos compactas*.

Una de las estructuras de datos compactas más conocidas son los árboles compactos, dentro de los cuales, el range min-max tree destaca por sus resultados prácticos. Actualmente, el range min-max tree cuenta con una versión estática y una versión dinámica a nivel teórico, pero a nivel práctico no existe una implementación disponible de la versión dinámica.

En el presente trabajo se propone implementar la versión dinámica del range min-max tree, basándose en su descripción teórica, para así explorar los alcances de esta versión. Implementando distintos métodos, tal como la inserción y eliminación, además de comparar con la versión estática el tiempo de ejecución, se reportan resultados empíricos y se deja disponible la primera implementación pública de esta estructura.

Índice

Índice de figuras

1. Introducción

Hoy en día, la tecnología ha crecido a un ritmo que se escapa al crecimiento humano. Tomando como ejemplo el caso del internet y la cantidad de información que esta contiene, un estudio [1] afirma que en abril del año 2022 existió un aumento del 63 % equivalente a 5 mil millones de nuevos usuarios en internet. Además de lo anterior, los grandes volúmenes de datos suponen un gran problema a las tecnologías de almacenamientos si estos no son representados adecuadamente. Soluciones que se han creado para solventar esta problemática abarcan compresión de la información, procesamiento paralelo y distribuido, y las más recientes son las estructuras de datos compactas (EDC). Las EDC son estructuras que permiten almacenar datos usando espacio cercano al mínimo teórico, al tiempo que soportan consultas eficientes en tiempo. En esta memoria de título, nos enfocaremos en una de las EDC más conocidas, los *árboles compactos [3].*

Los árboles compactos ayudan a almacenar secuencias de datos en forma compacta, las que tienen aplicaciones en bioinformática, representación de red sociales, representación de información geográfica, así como almacenar los distintos trayectos que hay entre ciudades en un país, entre otros.

Dentro de las representaciones que existen para árboles compactos el *range min max tree* [7] ha entregado mejores resultados tanto como en la teoría, como en la práctica. Si bien en teoría esta estructura permite modificaciones, en la práctica solo existen implementaciones estáticas. Lo más cercano a una implementación dinámica es el trabajo realizado por Cordova y Navarro [2], no obstante su implementación no está disponible.

En esta memoria de título se propone implementar la versión dinámica del range min-max tree, descrita de manera teórica en *Compact Data Structures [6].* Se realizaran experimentos para conocer los tiempos que demoran los distintos métodos que soporta la estructura, como también la comparación de uso de memoria entre las versiones estática y dinámica.

En la sección 2 se detallará acerca de la versión estática del range min max tree. En la sección 3 se detallará acerca de la versión dinámica del range min max tree. En la sección 4 se detallará acerca de las diferencias que existen entre las dos versiones de la estructura. En la sección 5 se detallará acerca de la implementación propuesta de esta memoria de título. En la sección 6 se detallará acerca de los experimentos realizados, los resultados y un análisis de estos. En la sección 7 se detallará las conclusiones de la memoria de título, el trabajo a futuro y complicaciones del trabajo realizado.

2. Range Min Max Tree estático

El range min max tree (rmMT)[6] es una estructura de datos compacta para representar árboles ordinales. Corresponde a un árbol binario completo[4], el cual guarda en sus hojas una representación de paréntesis balanceados del árbol que representa, mientras que en sus hojas internas almacena algunos valores estadísticos que permiten la navegación del árbol representado.

La construcción del rmMT tiene como entrada una secuencia balanceada de paréntesis del árbol a representar. Los paréntesis sirven como indicador de la cantidad de nodos y su posición en el árbol original.

2.1. Secuencia Balanceada de Paréntesis

Un árbol puede ser representado como una secuencia balanceada de paréntesis por medio de un recorrido en profundidad. Computacionalmente hablando, cada nodo de un árbol se representa con un par de paréntesis: un paréntesis de apertura cuando el nodo se visita por primera vez durante el recorrido en profundidad, y un paréntesis de cierre cuando se visita el nodo por última vez, es decir, cuando todo su subárbol ha sido visitado. A nivel de memoria, un paréntesis de apertura se representa con un bit en 1, mientras que un paréntesis de cierre se representa con un 0. La Figura 1 muestra un ejemplo de una secuencia balanceada de paréntesis.

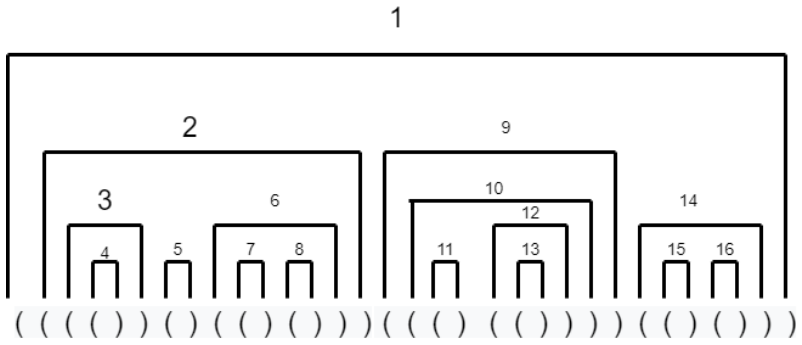


Figura 1: Secuencia balanceada de paréntesis a partir de la figura 2.

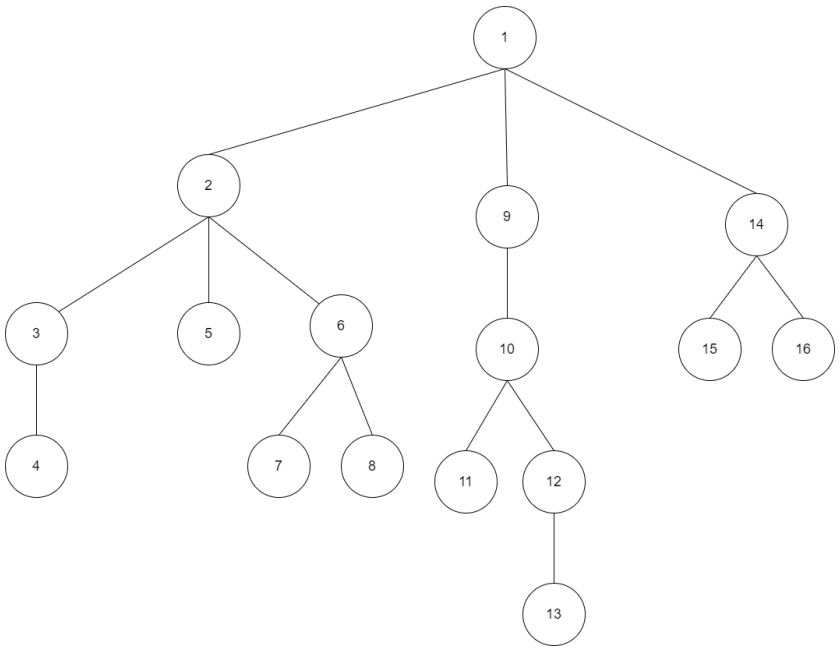


Figura 2: Ejemplo de un árbol cualquiera.

2.2. Descripción de la estructura

A continuación, se describirá la forma en la que se construye la estructura.

Terminología básica.

- **Exceso:** Dada una secuencia de paréntesis balanceada, se define exceso como el valor numérico de la diferencia entre la cantidad de paréntesis de apertura, con la cantidad de paréntesis de cierre. Sea $e(i)$ el exceso parcial hasta la posición i, tenemos

$$e(i) = e(i-1) + 1 \text{ Para un paréntesis de apertura.}$$
$$e(i) = e(i-1) - 1 \text{ Para un paréntesis de cierre.}$$

- **Valor Mínimo:** Es el valor mínimo de exceso de una secuencia de paréntesis, este valor mínimo se denota por m.

- **Valor Máximo:** Es el valor máximo de exceso de una secuencia de paréntesis, este valor máximo se denota por M.

- **Valor de Exceso:** Dada una secuencia de paréntesis, el valor del exceso corresponde al exceso al final de la secuencia. Este valor de exceso se denota por e.
- **Valor de num:** Dado un bloque de memoria que almacena paréntesis, se define como la cantidad total de paréntesis almacenados en el bloque de memoria. Este valor se denota como num.
- **Longitud de paréntesis:** Dado un bloque de memoria que almacena paréntesis, se define como la capacidad máxima del bloque. Este valor se denota como l.

Cómputo de las hojas. Con la secuencia balanceada de paréntesis creado, se dispone a crear los nodos hojas del rmMT, donde esta secuencia es dividida a través de bloques de memoria de tamaño fijo. En regla general, se asume que se dispone la memoria necesaria para que cada bloque quede con su capacidad máxima utilizada. Por cada bloque creado, se crea un nodo hoja, donde este último almacena los valores de exceso, mínimo y máximo, asociados a ese bloque.

Cómputo de nodos intermedios. Teniendo los nodos hoja con su respectiva información, estos se agrupan en pares consecutivos. Por cada par de nodos hoja, se crea un nodo padre, el cual heredará la información de sus dos nodos hijos. En particular, el valor mínimo del nodo padre será el valor mínimo entre sus hijos, el valor máximo será el valor máximo entre sus dos hijos y el valor de exceso será la suma de los excesos de sus dos hijos. Este proceso se va repitiendo hasta llegar al nodo raíz del range min-max tree.

Considerando que en ocasiones la cantidad de nodos hojas creadas puede llegar a ser distinto a una potencia de 2, esto producirá un desbalance en el árbol resultante. Para evitarlo, se crean nodos artificiales, los que poseen información "vacía"para cumplir con la regla de árbol binario completo.

La Figura 3 muestra un ejemplo del proceso de construcción.

Nótese que una secuencia balanceada de paréntesis es válida cuando el valor de exceso almacenado en la raíz del range min-max tree es cero.

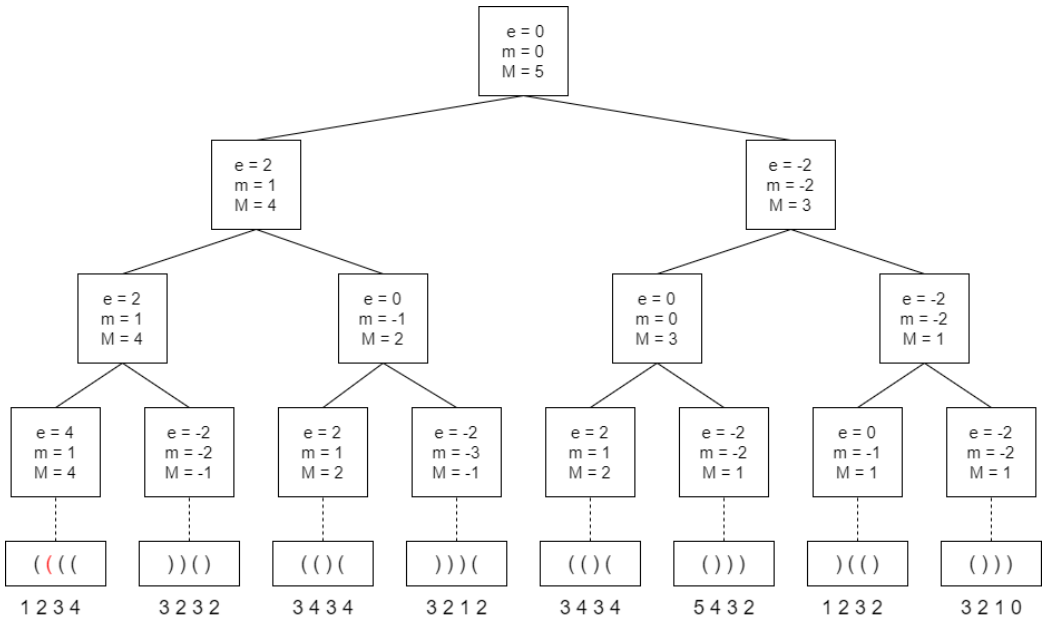


Figura 3: Ejemplo construcción del static-Range min max tree a partir del árbol de la figura 1.

2.3. Operaciones que soporta la estructura

Las operaciones permitidas en la estructura rmMT son de navegación, las que son reducidas a la operación de encontrar el paréntesis de cierre de un paréntesis de apertura y viceversa. Adicionalmente, la estructura permite computar información topológica del árbol, como la profundidad o la cantidad de hijos de un nodo. Dichas operaciones básicas utilizan las primitivas forward search y backward search, las que se explican a continuación.

2.3.1. Forward Search

Dada una secuencia balanceada de paréntesis esta búsqueda tiene como finalidad encontrar el paréntesis de cierre a partir de uno de apertura. Esto se hace mediante cuatro pasos, partiendo con el ingreso de la posición del paréntesis de apertura y el nodo a cuál está asociado, luego se realiza un recorrido de la sub secuencia de paréntesis del bloque asociado al nodo para verificar sí el paréntesis de cierre se encuentra en la sub secuencia, en el caso de encontrar el paréntesis se da por finalizada la búsqueda, o por el contrario,

se deberá recorrer los demás nodos en orden de izquierda a derecha de la estructura, hasta encontrar el nodo que contiene el paréntesis de cierre. Una vez detectado el nodo se realiza un recorrido hasta encontrar el nodo hoja que contiene dicho paréntesis, finalmente se realiza un recorrido de la sub secuencia de paréntesis.

Profundizando lo anteriormente mencionado, esta búsqueda tiene como objetivo encontrar la posición j del paréntesis de cierre dada la posición i de uno de apertura. Para realizar la búsqueda y que sea válida se debe cumplir la desigualdad tal que, la posición i tiene que ser menor que la posición j, por lo cual la posición de j siempre tiene que ubicar hacia la derecha con respecto a la posición de i, y además debe cumplir que el exceso hasta la posición i menos 1 sea igual al exceso hasta la posición j. Esta operación se puede generalizar de la siguiente forma, donde B es una secuencia de paréntesis, i es la posición del paréntesis de apertura, j es la posición del paréntesis de cierre y d es el exceso parcial de la posición i.

$$Forward\ Search(B, i, d) = min(\ j > i,\ excess(B, j) = excess(B, i) + d)$$

La forma de abarcar la búsqueda que se realiza en el range min Max-Tree, es hacia el sentido derecho de este mismo, y comienza por la posición siguiente de la posición inicial de búsqueda, vale decir $i + 1$. En la búsqueda se revisa los valores de mínimo y exceso que contiene el nodo, debido a la utilización de estos para realizar la navegación. Donde sí en el nodo hoja recorrido no se encuentra el paréntesis de cierre, se considera un valor de exceso parcial de recorrido descrito por d' y además, se debe realizar la pregunta, sí el nodo recorrido es un hijo izquierdo se debe verificar que; el exceso d del nodo recorrido sea menor o igual al exceso parcial más el valor mínimo del hijo derecho, en el caso de cumplir dicha condición se verifica que en ese nodo contiene el paréntesis de cierre del nodo a consultar. En caso contrario se revisa el nodo padre con el nuevo valor d', así sucesivamente se sube por el árbol hasta encontrar un nodo padre que contenga un hijo, tal que ese hijo tiene en su información la posición del paréntesis de cierre del nodo.

El siguiente pseudocódigo extraído del libro *Compact Data Structures [6]* describe el funcionamiento de la búsqueda.

Algorithm 1: Forward Search.

Result: Obtener la posición del paréntesis de cierre.

```
1  Begin
2      f ← ⌈i/c⌉ ;
3      t ← ⌈(i + 1)/b⌉ b/c;
4      d′←0;
5      foreach j←i + 1 to fc do
6          if bitread(b, j)←1 then
7              d′←d′ + 1
8          d′←d′ − 1 ;
9          if d′←d then
10             Return (d,j)
11     foreach p ←f + 1 to fc do
12         x← bitsread(B, (p − 1)c + 1, pc);
13         if d′ + C[x].m ≤ d then
14             Break
15         d′←d′ + C[x].e
16     if t ≤ p then
17         return (d′,tc + 1)
18     foreach j←(p − 1)c to pc do
19         if bitread(B, j)←1 then
20             d′←d′ + 1;
21         d′←d′ − 1;
22         if d′←d then
23             return (d,j)
```

2.3.2. Backward Search

Esta búsqueda es homóloga respecto a la búsqueda Forward Search, la diferencia es la dirección en la cual se realiza la búsqueda, debido a que esta va revisando hacia la izquierda del nodo a buscar. A diferencia de Forward Search se comienza por un paréntesis de cierre, por lo cual la desigualdad para validar y realizar la búsqueda se cumple tal que, la posición de i es mayor a la posición de j.

Esta operación se generaliza de la siguiente manera:
$Backward\ Search(B, i, d) = max(\ j < i,\ excess(B, j) = excess(B, i) + d).$

El siguiente pseudocódigo extraído del libro *Compact Data Structures* [6] describe el funcionamiento de la búsqueda.

Algorithm 2: Backward Search.

Result: Obtener la posición del paréntesis de apertura.

```
1  Begin
2      k ← ⌈i/b⌉;
3      d' ← 0;
4      foreach j ← i down to (k − 1)b + 1 do
5          if bitread(b, j) ← 1 then
6              d' ← d' − 1;
7          d' ← d' + 1 ;
8          if d' ← d then
9              Return j − 1
10     v ← leafnum(k);
11     while v < 2^[log(v)] and d' − R[v − 1].e + R[v − 1].m > d do
12         if v mod 2 ← 1 then
13             d' ← d' − R[v − 1].e;
14         v ← ⌈v/2⌉;
15     if v ← 2^logv then
16         return 0;
17     v ← v − 1;
18     while v ≤ r do
19         if d' − R[2v + 1].e + R[2v + 1].m ≤ d then
20             v ← 2v + 1;
21         d' ← d' − R[2v + 1].e;
22         v ← 2v;
23     k = numleaf(v);
24     foreach j ← kb down to (k − 1)b do
25         if d' ← d then
26             Return j
27         if bitread(B, j) ← 1 then
28             d' ← d' − 1;
29         d' ← d' + 1;
```

2.4. Limitaciones de la estructura

Al ser una estructura de datos estática la información está sujeta a su espacio de memoria previamente definida, por lo cual no puede haber variaciones de esta, ya sea añadiendo o eliminando los paréntesis. Por lo cual una vez ya asignada la memoria esta nunca cambia a lo largo de la creación del árbol.

3. Dynamic Range Min Max Tree

La estructura range min max tree dinámico o en su abreviatura rmMT-dyn, busca subsanar las falencias que posee la versión estática del árbol, la que es muy limitada al momento de realizar distintas operaciones, dado que solo permite realizar búsquedas. Esta estructura posee las características de AVL o red black tree, por lo cual la estructura cumple con la condición de árbol binario balanceado. El dinamismo de la estructura facilita la manipulación de la información contenida, dado que se permiten operaciones, tales como, inserción y eliminación, las cuales serán profundizadas en la sección **3.4**.

La construcción de la estructura en su versión dinámica varía de su versión estática en las condiciones de inserción y eliminación de paréntesis en los bloques de memoria, además de agregar dos valores en los nodos internos que servirán como navegación de la estructura al realizar las distintas operaciones.

Terminología básica.

- **Valor de ones:** Se define como la cantidad de paréntesis de apertura que contiene un nodo, este valor se describe como *ones*.
- **Factor de carga:** Se define como la cantidad de paréntesis que posee el bloque en la construcción inicial, este factor sirve para conocer cuántos paréntesis tendrá inicialmente cada bloque de memoria. Este se define como k.
- **Longitud de paréntesis:** Se define como la cantidad de paréntesis actual que posee un determinado bloque de memoria. Este valor se describe como l.

3.1. Creación de nodos hoja

Una vez computada la secuencia balanceada de paréntesis $P[0, 2n-1]$, donde n es la cantidad de nodos del árbol a representar, se crean bloques de memoria para almacenar la secuencia. Estos bloques utilizan la misma cantidad de memoria reservada, además de poseer un factor de carga k, el cual se define al inicio de la construcción de la estructura. Estos bloques son almacenados en una lista enlazada, la cual permite manipular los bloques, ya sea agregando o bien eliminando nodos del rmMT-dyn.

Dada una serie de inserciones y eliminaciones en la estructura, estas operaciones producen variaciones en la cantidad de paréntesis por bloque. Para mantener un buen rendimiento del rmMT-dyn, se deben mantener la siguiente heurística para la cantidad de paréntesis en cada bloque: Dado un parámetro $L > 0$ cada bloque de paréntesis cumple con $L \leq k \leq 2L$. Los bloques creados a partir de la secuencia de paréntesis son insertados en una lista doblemente enlazada.

En esta variación en el manejo de memoria en los bloques, se debe almacenar un puntero en los nodos hojas del rmMT-dyn que referencie el nodo de la lista enlazada que contiene al respectivo bloque. Cabe destacar que si ocurre algún cambio de posición en los bloques de memoria, el puntero de referencia del nodo hoja debe ser actualizado. En otras palabras, sí se mueve un segmento de la secuencia solo cambia el puntero de referencia del nodo hoja. Una vez creado los nodos hoja, se computan y almacenan los valores de mínimo, máximo, último exceso, además de *num* y *ones*.

3.2. Computación de nodos internos

Los nodos hojas se relacionan en pares. Por cada par de nodos relacionado hay un nodo padre, el cual contiene la información de sus dos nodos hijos, esta información la obtiene a partir de las siguientes ecuaciones:

Dado un nodo v, con hijo izquierdo $v.izq$ e hijo derecho $v.der$, definimos las siguientes ecuaciones:

$$
\begin{aligned}
v.e &= v.izq.e + v.der.e \\
v.m &= min\{v.izq.m, v.der.min + v.izq.e\} \\
v.M &= max\{v.izq.M, v.der.M + v.izq.e\} \\
v.num &= v.izq.num + v.der.num \\
v.ones &= v.izq.ones + v.der.ones
\end{aligned}
$$

Una vez computado el último nivel de los nodos internos, estos vuelven a enlazarse de par en par hacia un nodo padre, hasta así completar el recorrido hacia el nodo raíz de la estructura. Cada nodo interno de la estructura contiene solo la información de su propio subárbol. Cuando se termina el cómputo de valores para los nodos internos se debe verificar si la secuencia balanceada de paréntesis no fue alterada en algún punto de la construcción, por lo cual se debe verificar en el nodo raíz los siguientes valores:

- El valor de e de toda la secuencia balanceada de paréntesis debe tener un valor de 0.
- El valor m describe el valor mínimo global de la secuencia balanceada de paréntesis y debe tener un valor de 0.
- El valor de máximo describe el valor máximo global de la secuencia balanceada. Este valor debe ser igual a la altura del árbol representado por la secuencia balanceada de paréntesis.
- El valor de *ones* debe ser exactamente la mitad a la cantidad total de paréntesis que hay en la secuencia.
- El valor de *num* del nodo raíz describe la cantidad total de paréntesis que hay en la secuencia de paréntesis.

Porlo anterior, la estructura dinámica se representa de la forma que se puede apreciar en la figura 4

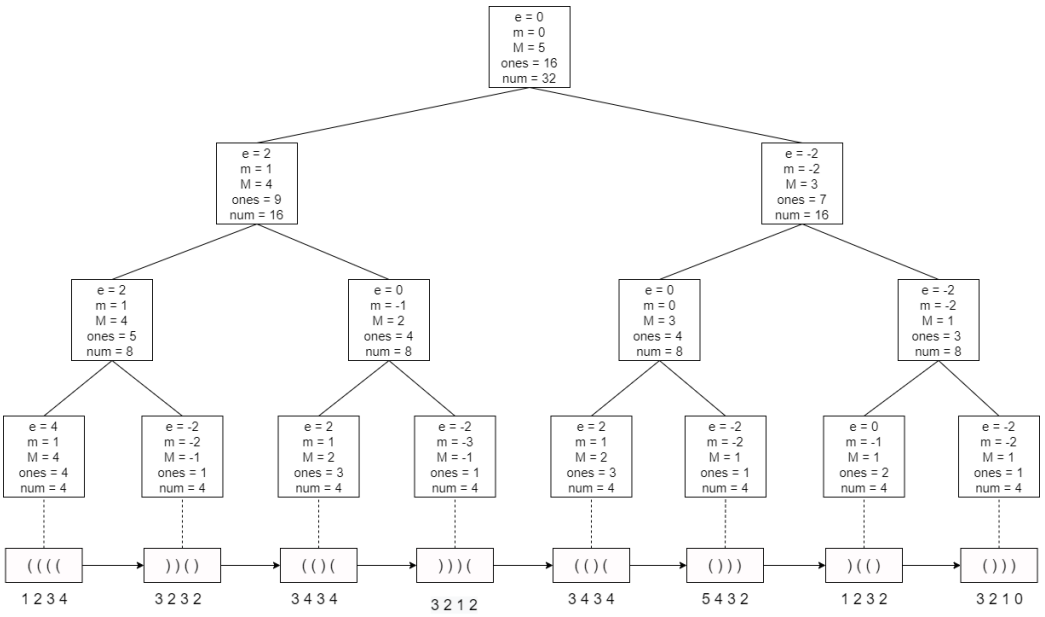


Figura 4: Ejemplo construcción del dynamic-Range min max tree para la secuencia de paréntesis de la figura 1.

3.3. Operaciones

El rmMT-dyn al igual que en su versión estática se pueden realizar operaciones de búsqueda para conocer la posición de los paréntesis de apertura, como también los de cierre de algún nodo. A diferencia de la estructura estática, la versión dinámica permite realizar operaciones que la anterior no tiene soporte, tal como insertar un par de paréntesis y en su contraparte eliminarlos.

A continuación se explicará en profundidad la forma de realizar estas funciones en la estructura, además de los cambios que pueden provocar en esta.

3.3.1. Inserción de nodos

Esta operación tiene como objetivo añadir información a la estructura por medio de la inserción de paréntesis, donde los nuevos paréntesis son añadidos en una posición determinada, destacando que todas las inserciones se realizan en los nodos hojas de la estructura. Para conocer la posición del paréntesis a insertar se debe realizar un recorrido en DFS, donde el valor de

ones determinará la posición, dado que este contiene la información de la cantidad de paréntesis de apertura que posee un nodo. Una vez identificada la posición del paréntesis y el nodo que lo contiene, se debe verificar si el nodo cumple con la invariante. En caso de que sea una inserción de un par de paréntesis que referencia a un nodo hijo izquierdo, el método concluye. Por otra parte, de ser una inserción de un par de paréntesis que referencia a un nodo derecho o un nodo padre, se debe realizar forward search para encontrar la posición del paréntesis de cierre y luego insertar. Si el nodo no cumple con la invariante se debe realizar la división del nodo y su bloque correspondiente, por lo cual el nodo hoja se convierte en un nodo interno y se crean nuevos nodos hojas, restableciendo la invariante. Luego se realiza la actualización de los valores que fueron modificados en el nodo inicial. Posteriormente se actualizan los valores de la rama en la cual se encuentra el nodo al cual se le realizó una inserción, cuyos cálculos de dichos valores son los descritos en las fórmulas de la sección **3.3** Finalmente, si la creación de nuevos nodos internos deja al rmMT-dyn no balanceado, se deben re-localizar los nodos según corresponda, para que esta quede balanceado nuevamente.

En la figura 5 se ve el caso de una inserción de un nuevo nodo en el árbol representado por un par de paréntesis (y). Las consecuencias de dicha inserción producen cambios en el rmMT-dyn, donde los nodos de color verde indican los nuevos nodos hoja creados y los nodos rojos indican todos los nodos donde existieron actualizaciones de sus valores. Cabe destacar que cada nuevo nodo hoja creado se encuentra al 50 por ciento de su capacidad máxima de almacenamiento.

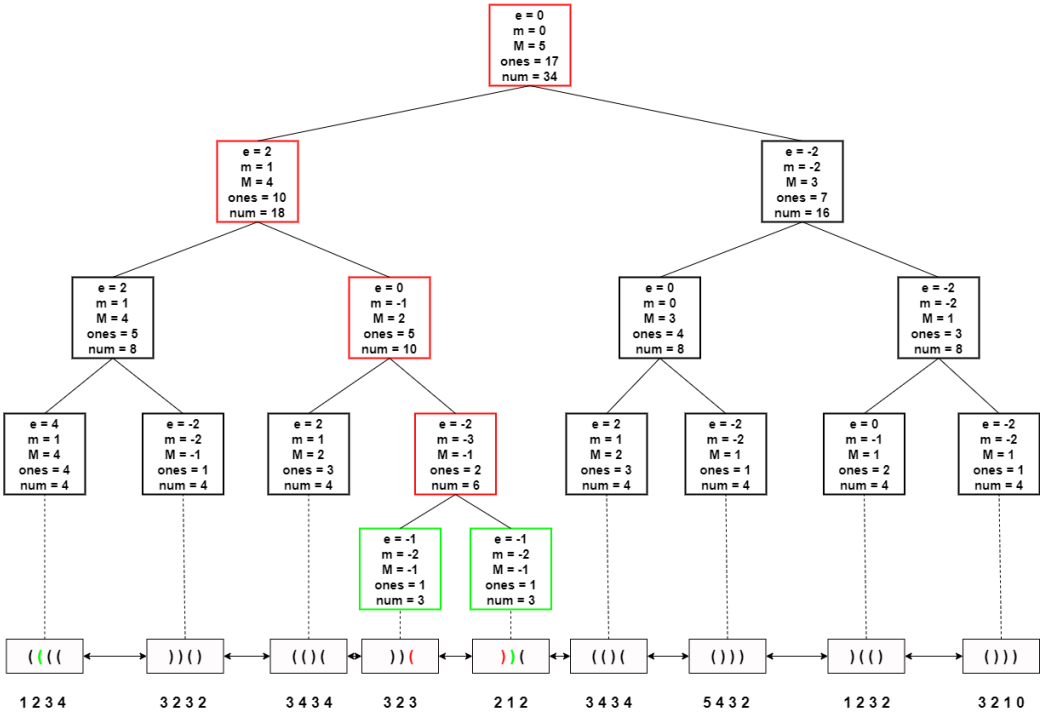


Figura 5: Modificación del rmMT-dyn la figura 4, luego de insertar un nuevo hijo derecho al nodo 2 del árbol representado de la figura 2. Los paréntesis que representan al nodo insertado, están destacados en rojo.

3.3.2. Eliminación de nodos

La operación de eliminar es análoga a la de inserción, pero con la diferencia que una vez identificados los paréntesis estos se eliminan y actualizan los valores del nodo hoja, para luego actualizar los niveles superiores. En el caso que el bloque de memoria contenga una cantidad menor a la cantidad mínima de paréntesis permitida por un bloque, este nodo se fusiona con su bloque hermano. En caso de que el bloque hermano no pueda ingresar más paréntesis debido a insuficiencia de memoria disponible, este le cede unos de sus paréntesis para que cada nodo hoja asociado a su bloque cumpla con la invariante.

4. Diferencia entre versiones de la estructura

Esta sección compara, de manera resumida, las versiones estática y dinámica del range min-max tree. Considerando su construcción, ambas son similares, puesto que la computación de los valores es la misma, y la forma en que son asociados los nodos hojas hasta llegar al nodo raíz del árbol. Como la versión dinámica busca dar solución a las limitantes que posee la versión estática, la primera en la creación de los bloques de memoria se denomina un factor de carga para determinar la cantidad de paréntesis que tendra inicialmente el bloque, además que el conjunto de bloques se representa como una estructura de lista doblemente enlazada, por lo cual un bloque puede conocer a sus bloques vecinos, mientras que en la versión estática esto no ocurre, dado que el bloque de memoria solo se referencia al nodo hoja correspondiente del árbol. En la creación de los nodos hojas en la versión estática solo contempla tres valores a almacenar, los cuales son; el valor mínimo, máximo y el valor de exceso final, mientras que, la versión dinámica también considera los valores mencionados, además de la cantidad de paréntesis de apertura y la carga que posee.

Las características por las cuales se rigen las dos versiones son similares con la diferencia en que, la versión dinámica supone las características de un AVL o un Red Black Tree, mientras que la versión estática solo cumple las condiciones de un árbol binario completo. Considerando el caso que la cantidad de bloques sea distinta a una potencia de dos, ocurre la creación de un bloque de memoria sin contenido que referencia a un nodo hoja sin información, mientras que en la versión dinámica no se crea un bloque dado que la condición de existencia de este último es de poseer una cantidad mínima de paréntesis contenidos.

Considerando el caso de las operaciones soportadas entre las dos versiones de la estructura, la versión estática solo soporta métodos de navegación de información, mientras que la dinámica además de soportar los métodos de la primera, se agregan las operaciones de inserción y eliminación.

5. Implementación propuesta

En este capítulo se detallará la implementación del código que describe el rmMT-dynamic, el cual se divide en las secciones de construcción de la estructura y de los métodos implementados para validar su correcto funcionamiento.

Como punto de partida, se tomó la decisión de implementar el código en el lenguaje de programación C++, debido a su manejo a bajo nivel con respecto a las asignaciones de memoria para cada objeto creado y a su vez al manejo de procesamiento que soporta la estructura en discusión.

5.1. Construcción de la estructura

Se contempla en la construcción de la estructura la creación de tres clases que describen el funcionamiento de esta misma: clase nodo, rmMT y la clase bloque. Cada una de ellas contiene información requerida para validar la estructura. En el caso de clase nodo tiene la información de mínimo, máximo, exceso, cantidad de paréntesis, cantidad de paréntesis de apertura, cantidad de paréntesis de cierre, la altura del nodo y sus enlaces hacia sus nodos padres e hijos. La clase rmMT contiene la información global de la estructura. Dado que la estructura es dinámica esta clase almacena la información del nodo raíz de la estructura. En el caso de la clase bloque, esta contiene la información de la sub secuencia de paréntesis, como también los enlaces de los bloques vecinos y del nodo hoja al cual está asociada.

Dada la entrada de secuencia balanceada de paréntesis se define de forma local el tamaño máximo que tendrá un bloque de memoria, la carga inicial que tendrá el nodo hoja en la construcción, la cantidad de paréntesis total de la secuencia balanceada de paréntesis y por último la cantidad de nodos hojas que serán creados, este cálculo se realiza con la división entre la cantidad total de paréntesis de la secuencia balanceada de paréntesis y la carga inicial que tendrá cada nodo hoja.

Para cada iteración del ciclo que recorre la cantidad de nodos hojas, se asigna memoria a un nodo hoja y a su respectivo bloque para almacenar la sub secuencia de paréntesis, donde además se obtienen los valores de máximo, mínimo, exceso, cantidad de paréntesis de apertura, cantidad de paréntesis de cierre, cantidad de paréntesis total del nodo.

Finalizada la creación del nodo hoja y del bloque se realiza la conexión entre ellos. Al mismo tiempo, se almacena en un vector, para posteriormente

ser enlazados con sus respectivos vecinos.

A continuación se visualiza el pseudocódigo de la construcción de la construcción de los nodos hoja.

Algorithm 3: Creación de los nodos hojas y de su vector.

```
Result: Obtener el vector de los nodos hojas.
1  Begin
2      n ← Número de paréntesis;
3      s ← 2L ; // Invariante
4      c ← porcentaje de llenado de los nodos hoja (s*0.75);
5      numleaf ← n/c;
6      vector b; // Vector de bloques
7      vector vl; // Vector de hojas
8      for i ← 0 to numleaf do
9          create Leaf;
10         create Block;
11         Llimit ← c * i;
12         Rlimit ← Llimit + c;
13         for symbol ← Llimit to Rlimit do
14             Block.vect ← get_bit (parenthesis, symbol);// Lectura del
                  archivo de paréntesis
15             Leaf ← input (value);
16         Block.leaf ← Leaf; // enlaza el bloque con el nodo hoja
17         Leaf.block ← Block; // enlaza el nodo hoja con el bloque
18         b ← push_back(block);
19         vl ← push_back(leaf);
```

Una vez creado todos los nodos hojas, es posible que el último nodo hoja de la estructura rmMT-dyn no cumpla con la invariante. Si esto ocurre, se le quita al vecino anterior los paréntesis necesarios para cumplir con la invariante, esto puede hacer que el vecino no cumpla con la invariante, por lo cual se debe repetir el proceso hasta que todos los nodos hoja cumplan con la invariante. Luego de verificada, se realiza un recorrido en el vector de bloques para enlazar cada bloque con sus respectivos vecinos que para efectos de esta memoria de título serán vecino anterior y vecino posterior.

A continuación, se puede ver el pseudocódigo utilizado para los enlaces de los bloques.

Algorithm 4: Enlazamiento de los bloques.

```
Result: Enlazar los bloques con sus vecinos
1  Begin
2      for i ← 0 to vector_size do
3          b[i].prev ← b[i − 1];
4          b[i].next ← b[i + 1];
5          if i = 0 then
6              b[i].prev ← NULL;
7              b[i].next ← b[i + 1];
8          if i + 1 = vector_size then
9              b[i].prev ← b[i − 1];
10             b[i].next ← NULL;
```

Una vez creados todos los nodos hojas con sus respectivos bloques de sub-secuencia de paréntesis, se realiza el cálculo de cuantos nodos internos que puede llegar a tener la estructura, este cálculo se realiza por una de las características fundamentales de un árbol binario, la cual es: la cantidad de nodos internos es igual a la cantidad de nodos hojas menos uno. Para cada iteración sobre la cantidad de nodos internos se reserva memoria a un nodo, donde el caso inicial es la raíz de toda la estructura, en las iteraciones siguientes son los demás nodos internos de la estructura. Una vez asignada la memoria para un nodo este se almacena en un vector de nodos internos, puesto que para un nodo conozca a su respectivo padre y a sus hijos, para realizar los enlaces de los nodos internos se aplicó el siguiente criterio donde, dado un nodo i su padre se conoce por $(i-1)/2$. Para luego conocer que ese nodo i es un hijo izquierdo o derecho se realiza la división simple por dos. En caso de ser resultado impar el nodo i es un hijo izquierdo, en caso contrario es un hijo derecho.

Una vez creado todos los nodos internos y a su vez almacenados en un vector, se realiza un recorrido por niveles (recorrido BFS) para obtener el último nivel de nodos internos de la estructura. Una vez obtenido los nodos estos son almacenados provisoriamente en un vector, para realizar los enlaces con sus respectivos nodos hojas.

Realizando un recorrido en el vector de nodos que representa el último nivel de los nodos internos, se enlaza por cada nodo interno dos nodos hojas. Vale decir, dos nodos hojas consecutivos se enlazan a un nuevo nodo interno, el cual hereda su información siguiendo las fórmulas descritas en la **sección**

3.2.

Para los casos de imparidad, vale decir, cuando el vector de nodos hojas tiene un tamaño impar de elementos, el último nodo interno solo tendrá un hijo izquierdo.

Una vez enlazados los nodos hojas con sus respectivos nodos internos, se procede a realizar un recorrido con respecto al vector de los nodos internos para ir escalando hacia el nodo raíz, para obtener la información global de la secuencia balanceada de paréntesis que represa un árbol cualquiera.

A continuación, se puede visualizar el pseudocódigo de la creación de los nodos internos y su computación.

Algorithm 5: Creación de los nodos internos y enlazar con los nodos hojas.

Result: creación de los nodos internos y computación final de la estructura.

```
1  Begin
2      Internal_node_vector I;
3      amount_internal ← length_vector(vl - 1);
4      for i ← 0 to amount_internal do
5          create internal;
6          I ← push_back(internal);
           // Los nodos internos creados se ingresan al vector I
7          if i = 0 then
8              root ← I[i] // Nodos interno que es raíz
9              root.left ← NULL;
10             root.right ← NULL;
11             root.parent ← NULL;
12             root.level ← i
13         else
14             if I[i − 1]/2 ≠ NULL then // Si existe el padre, se
                enlaza
15                 I[i].parent ← I[(i − 1)/2];
16                 I[i].level ← I[(i − 1)/2].level + 1;
17                 I[i].left ← NULL;
18                 I[i].right ← NULL;
19                 if i mod 2 ← 1 then
20                     I[(i − 1)/2].left ← I[i];
21                 else
22                     I[(i − 1)/2].right ← I[i];
23     vector last_internal;
24     last_internal ← get_internal_last(); // Obtener último nivel
           nodos internos
25     update_internal(vl,last_internal);
26     for i ← arr.size() − 1 to 1 do
27         update_value();
28         i ← i − 2;
```

Finalmente, como la construcción fue realizada mediante el uso de vectores, para que un nodo conozca sus nodos hijos y padre, se contempla un análi-

sis asintótico de tiempo $O(n)$, donde n es el tiempo que se demora crear los nodos. En memoria esta limitada por la cantidad de nodos internos del árbol.

5.2. Métodos implementados en la estructura

En esta sección se explicará la implementación de los distintos métodos base y su función en la estructura.

Detección de un bloque en el rmMT-dyn. Para recorrer la estructura se tomó la decisión de ir realizando el recorrido en profundidad con respecto a la cantidad de *ones* que contiene cada nodo. Ya que esa cantidad describe el número de paréntesis de apertura que contiene un nodo, permite navegar el rmMT-dyn hasta la hoja que contiene el paréntesis de apertura que representa a un nodo del árbol que es representado por la secuencia balanceada de paréntesis.

Para saber si moverse hacia un nodo izquierdo o derecho se realiza la siguiente pregunta: dado un valor a, si el hijo izquierdo tiene un valor de *ones* mayor a a, se continúa el recorrido hacia la izquierda, en caso, contrario se mueve hacia el hijo derecho actualizando el valor de a donde:

$$a = a - v.izq.ones$$

Esta búsqueda finaliza cuando encuentra un nodo hoja que contenga a.

Búsqueda de un paréntesis de apertura dentro de un bloque. Dado un nodo hoja y un valor a, se realiza un recorrido al bloque de paréntesis del nodo hoja, donde se cuentan los paréntesis de apertura. El recorrido termina cuando la cantidad de paréntesis contados es igual a a.

Forward Search. Dado un nodo hoja y un valor a se realiza la búsqueda forward search para encontrar el nodo hoja que contenga el paréntesis de cierre de a.

Búsqueda del paréntesis de cierre dentro de un bloque. Dado un nodo hoja y un d que describe el exceso parcial entre un paréntesis de apertura con su respectivo paréntesis de cierre, se realiza un recorrido al bloque de paréntesis del nodo hoja, donde por cada paréntesis de apertura visitado,

suma en uno el valor de d, y en caso contrario resta uno el valor de d. La búsqueda termina cuando el valor de d más uno es igual a cero.

Actualización de los valores un nodo hoja. Dado un nodo hoja se procede a contabilizar todos los paréntesis que contiene el bloque asociado al nodo hoja, donde se calcula los valores de m, M, e, $ones$ y num del bloque. Una vez realizado se actualiza la información del nodo hoja.

Actualización de una rama del árbol. Dado un nodo hoja, al cual se realizaron modificaciones en sus valores almacenados, se procede a actualizar los valores de la rama del árbol.

Reestructuración. En ocasiones es necesario reconstruir toda la estructura rmMT-dyn. A partir de la raíz, se obtienen los nodos hojas de la estructura, los cuales son almacenados en un vector. Luego de eso se procede a realizar la misma lógica que la construcción inicial de la estructura, a partir del punto cuando ya se crearon los nodos hojas con sus respectivos bloques de paréntesis.

Para los métodos de inserción se realiza la reestructuración en el momento cuando la diferencia entre alturas de los nodos hojas hijos de un nodo interno sea mayor a 2.

Para el método de eliminación se realiza la reestructuración cuando el nodo interno vecino no posee nodos hojas hijos.

Función Match. Esta función realiza las búsquedas de forward y backward search, los cuales servirá para realizar las comparaciones de tiempo con la versión estática de la estructura.

Para la implementación del la función de inserción, se decidió tener tres variantes de la función, ya que es de nuestro interés qué sucede cuando se inserta lo más a la izquierda posible, lo más derecha posible y qué sucede cuando se insertan en dos bloques distintos del rmMT-dyn.

En el siguiente pseudocódigo se puede apreciar la forma general de inserción para cada variante.

Algorithm 6: Inserción

Result: Insertar un nuevo nodo (par de paréntesis) a la secuencia de paréntesis.

```
1  Begin
2      Node leaf ← navigate(root, a);
3      pos ← open(leaf, a);
4      Block aux ← leaf.bloc;
5      if aux.y.size() < 2L then
           // Condición de cumplimiento de la invariante
6          insert_parenthesis(aux,pos);
7          input_value(leaf) // Actualiza los valores del nodo
               hoja
8          update_value(leaf) // Actualiza los valores de la rama
               del árbol
9      else
10         Node child_left ← New Node();
11         Node child_right ← New Node();
12         Block block_left ← New Block();
13         Block block_right ← New Block();
14         Link_block(block_left)← Link_leaf(child_left);
15         Link_block(block_right)← Link_leaf(child_right);
16         Divide_block(block_left, block_right, aux)// Divide el
               bloque aux en dos bloques
17         Delete aux;
18         if pos < block_left.size() then
19             insert_parenthesis(block_left,pos);
20             input_value(child_left);
21             input_value(child_right);
22             update_value(child_left);
23             update_value(child_right);
24             check_balanced(child_left);
               // Función que verifica el balance del árbol
25         else
26             insert_parenthesis(block_right,pos);
27             input_value(child_left);
28             input_value(child_right);
29             update_value(child_left);
30             update_value(child_right);
31             check_balanced(chil_dright);
```

5.2.1. Inserción de un nodo hijo izquierdo en la sub secuencia de paréntesis

Para realizar la inserción hacia la izquierda se debe conocer en cuál nodo se desea insertar un nodo hijo izquierdo, por lo cual dado un valor a se conoce la posición y el nodo hoja donde se realiza dicha inserción.

Se consulta si el bloque donde será insertado el nuevo par de paréntesis cumple con la condición de invariante. De ser así, se realiza la inserción en la posición siguiente para el paréntesis de apertura y en la posición más dos para el paréntesis de cierre, luego se realiza la actualización del nodo y su respectiva rama hasta llegar hacia el nodo raíz de la estructura.

En caso de no cumplir la invariante se divide el bloque en dos partes iguales desencadenando que el nodo hoja deba tener un hijo izquierdo y derecho, para que la estructura siga cumpliendo la invariante. Para ello se procede a asignar memoria para dos nuevos nodos hojas y sus respectivos bloques de paréntesis, luego se procede a dividir el bloque de paréntesis en su mitad, donde para el bloque del hijo izquierdo se consideran todos los paréntesis desde el inicio del bloque hasta la mitad de este mismo, para el bloque del nuevo hijo derecho se consideran todos los paréntesis desde la mitad hasta el final del bloque.

Una vez realizada la división del bloque de paréntesis se actualizan los enlaces de los dos nuevos nodos hojas con su respectivo nodo padre interno y sus respectivos bloques. Luego de actualizar los enlaces se procede a eliminar la memoria del bloque antiguo.

Finalmente se verifica si el valor de a es menor a la cantidad de paréntesis del hijo izquierdo se realiza la inserción en el hijo izquierdo, luego de insertar el par de paréntesis se actualizan los valores de los dos nodos hojas y su respectiva sub rama del árbol, en caso contrario que el valor de a sea mayor a la cantidad de paréntesis abiertos del hijo izquierdo se inserta en la posición menos el tamaño del hijo izquierdo.

La diferencia entre alturas de los nodos hojas de un nodo interno no puede ser mayor a 2, por lo cual se verifica desde el nodo al cual fue insertado el nuevo par de paréntesis si hasta tres de sus ancestros cumplen con que la diferencia entre sus nodos hijos sea menor a 2. En caso de no cumplir, se realiza una reestructuración de la estructura.

Finalmente, como se deben recorrer los bloques en el peor caso el análisis asintótico en tiempo para este método es de $O(n + m)$, donde n son las cantidades de operaciones y m son las cantidades de reestructuraciones.

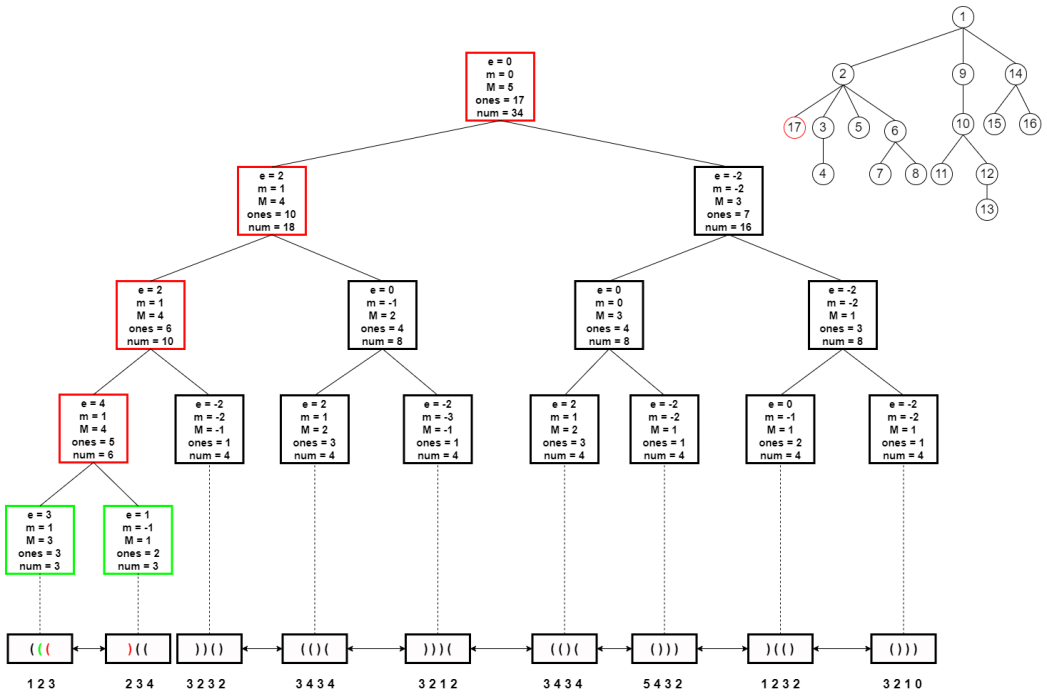


Figura 6: Ejemplo del método de inserción de un nodo hijo izquierdo.

A partir del ejemplo de la figura 4 se realizó la implementación del método de inserción de un nodo hijo izquierdo al nodo con id 2, dando como resultado la figura 6, donde los nodos en color rojo actualizaron sus valores y los nodos de color verde son los nuevos nodos hojas.

5.2.2. Inserción de un nodo hijo derecho en la sub-secuencia de paréntesis

Para realizar una inserción de un nodo hijo derecho a un nodo v, se debe conocer el paréntesis de cierre del nodo v, por lo cual se utiliza el algoritmo de forward search para encontrar el paréntesis de cierre con su respectivo nodo hoja al cual se realizará la inserción.

Se verifica que el nodo donde se realizará la inserción, nodo v, cumple con la invariante de los nodos hoja. En caso de cumplirla, se debe verificar si la posición en el bloque del nodo hoja v del cual se insertará el par de paréntesis corresponde al inicio del bloque, lo que provoca la inserción del par de paréntesis en la posición 0 del bloque, insertando primero el paréntesis de

cierre y luego el de apertura respectivamente. Posteriormente, se actualiza la rama correspondiente del árbol. En el caso de tener que insertar en una posición distinta al inicio del bloque, se inserta en primer lugar el paréntesis de cierre en la posición menos uno para luego insertar el paréntesis de apertura en la posición menos dos.

En caso de no cumplir con la invariante se realiza el mismo cálculo que insertar a la izquierda.

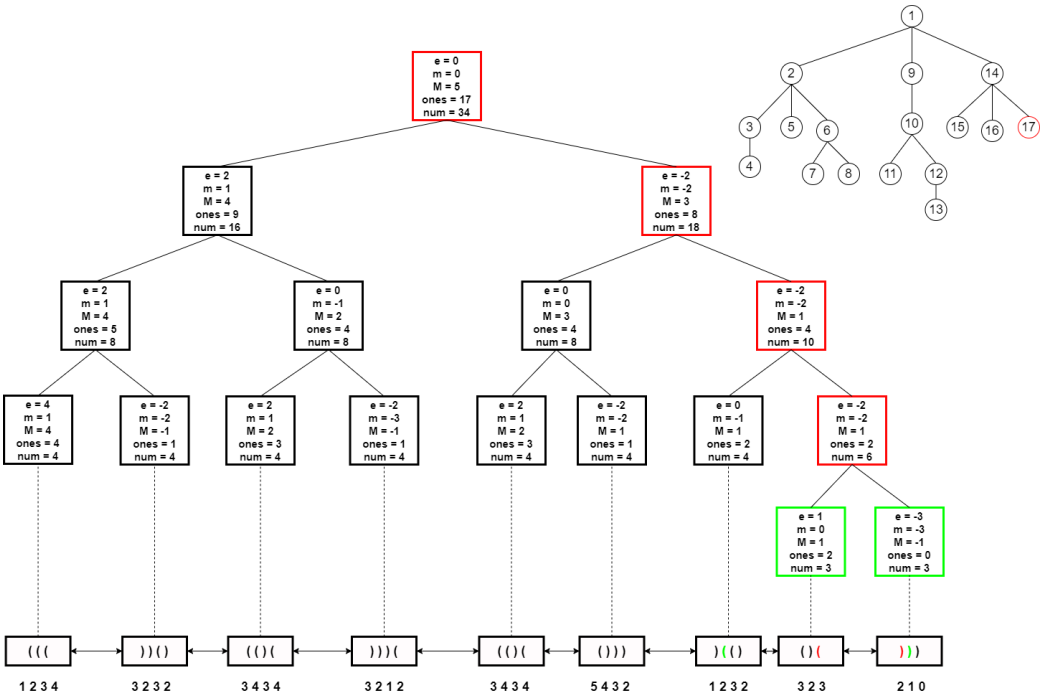


Figura 7: Ejemplo del método de inserción de un nodo hijo derecho.

A partir del ejemplo de la figura 4 se realizó la implementación del método de inserción de un nodo hijo derecho al nodo con id 14, dando como resultado la figura 7, donde los nodos en color rojo actualizaron sus valores y los nodos de color verde son los nuevos nodos hojas.

Finalmente, como se deben recorrer los bloques en el peor caso el análisis asintótico en tiempo para este método es de $O(n + m)$, donde n son las cantidades de operaciones y m son las cantidades de reestructuraciones.

5.2.3. Inserción de un nodo padre en la sub-secuencia balanceada de paréntesis

Este método de inserción contempla el suceso de insertar un par de paréntesis no necesariamente en un mismo nodo del rmMT-dyn, por lo cual se contempla las búsquedas de find open y find close por separado, es decir, que la inserción de paréntesis se contempla por unidad y no como par de paréntesis.

Lo cual se contempla para el paréntesis de apertura tiene la restricción de cuando se considera la posición en el bloque del paréntesis de apertura sea igual a cero, este es insertado en la posición 0, en el caso del paréntesis de cierre no se considera ninguna restricción extra, puesto que se inserta en la posición siguiente al paréntesis de cierre.

Para mantener el orden, primero se inserta el nuevo paréntesis de apertura y luego el de cierre. Los procesos de inserción son los mismos descritos en las secciones anteriores.

Finalmente, como se deben recorrer los bloques en el peor caso el análisis asintótico en tiempo para este método es de $O(n + m)$, donde n son las cantidades de operaciones y m son las cantidades de reestructuraciones.

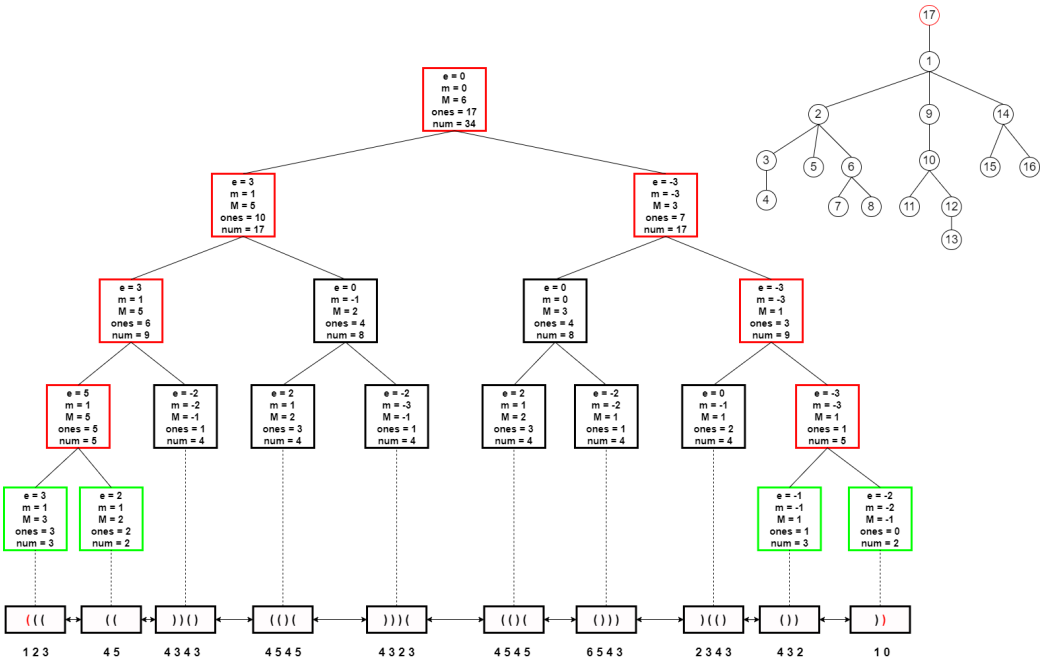


Figura 8: Ejemplo del método de inserción de un nodo padre.

A partir del ejemplo de la figura 4 se realizó la implementación del método de inserción de un nodo padre al nodo con id 1, dando como resultado la figura 8, donde los nodos en color rojo actualizaron sus valores y los nodos de color verde son los nuevos nodos hojas.

5.2.4. Eliminación de un nodo en la sub secuencia de paréntesis

Para eliminar un par de paréntesis, vale decir, un paréntesis de apertura con su respectivo paréntesis de cierre, se realizan las búsquedas de posición en los bloques de paréntesis como también los nodos que contiene estos paréntesis. Para ello se verifica primero si los nodos encontrados para el paréntesis de apertura y cierre son los mismos. En ese caso se realiza la eliminación en conjunto. Para ello se dispone a eliminar en las posiciones correspondientes los paréntesis de cierre y de apertura, luego se verifica si el nodo hoja luego de la eliminación cumple con la invariante. En caso de cumplir con la invariante se realiza la actualización de la rama del árbol. En el caso de no cumplir con la invariante de que la cantidad de paréntesis que contiene el bloque asociado al nodo hoja es menor a L, se verifican tres casos:

Caso 1. Verificar si el bloque que no cumple la invariante posee un vecino izquierdo con el que se pueda fusionar, lo que se hace con la ayuda de un vector auxiliar se juntan las dos secuencias de paréntesis en una en orden de izquierda a derecha. Una vez realizada la fusión de bloques se actualizan las ramas de la estructura y finalmente se elimina el nodo que contiene el bloque vacío y el mismo bloque vacío, para finalmente verificar si la estructura requiere una reestructuración.

Caso 2. En el caso de no ser necesaria la fusión, se realiza un intercambio de paréntesis, donde al vecino anterior se le solicitan los suficientes paréntesis para que el nodo cumpla con la invariante, para luego actualizar la rama hasta la raíz de la estructura.

Caso 3. En el caso de no existir un vecino anterior se realizan las mismas consultas descritas, pero para el vecino siguiente.

En el caso donde los paréntesis de cierre y apertura se encuentran en distintos nodos hoja cada eliminación, así como también el intercambio de paréntesis entre los bloques vecinos, se realiza de forma separada.

Finalmente, como se deben recorrer los bloques en el peor caso el análisis asintótico en tiempo para este método es de $O(n + m)$, donde n son las cantidades de operaciones y m son las cantidades de reestructuraciones.

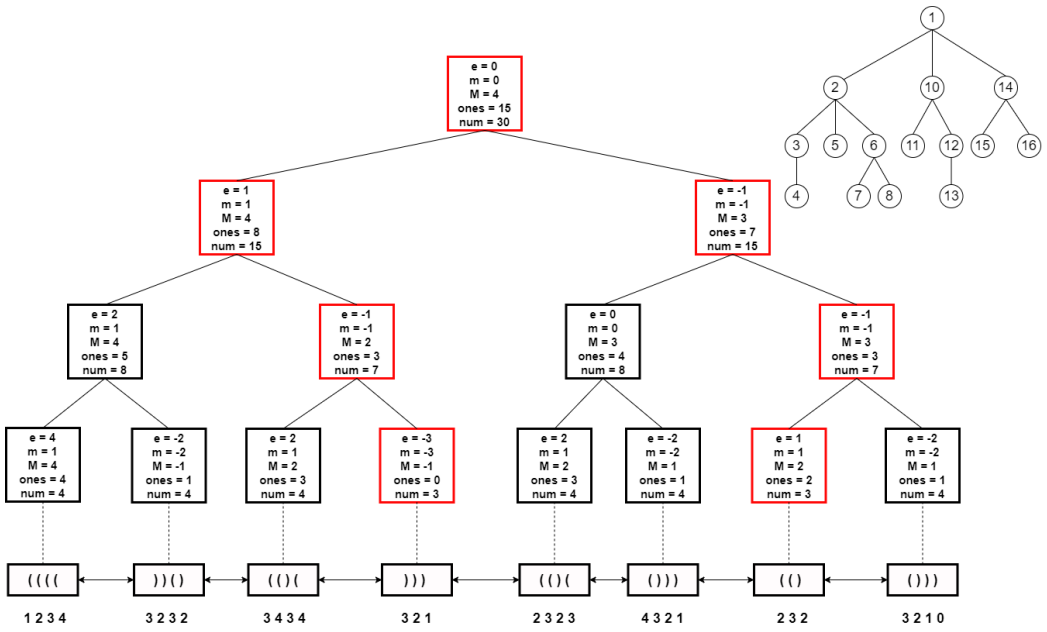


Figura 9: Ejemplo del método de eliminación de un nodo.

A partir del ejemplo de la figura 4 se realizó la implementación del método de eliminación del nodo con id 9, dando como resultado la figura 9, donde los nodos en color rojo actualizaron sus valores.

5.3. Validación de la estructura

Para realizar la validación de la estructura se compiló con la extensión `-g`, dicha extensión sirve para debuggear el código de la estructura, además se utilizó la herramienta Valgrind para el correcto uso de memoria. Además, se utilizó la herramienta GDB como complemento al uso de la IDE Visual Studio Code, donde se verificó la información que almacenaba el nodo raíz, la cual tiene correlación con la entrada de la secuencia balanceada de paréntesis. Para testear durante el desarrollo se utilizaron dos input de testing, uno con 4.096 paréntesis y la creación de 16 nuevos nodos hojas; mientras que el otro es una secuencia balanceada de paréntesis para verificar las búsquedas y corregir la imparidad de la estructura.

6. Estudio experimental

En este capítulo se abordará todo lo relacionado con los experimentos realizados como también el análisis de los resultados obtenidos. La implementación de la estructura de datos descrita en las secciones anteriores se encuentra disponible en el siguiente link `https://github.com/matimora/Memoria-Final.git`

6.1. Entorno de trabajo

El entorno de ejecución de los experimentos se realizó en un servidor privado del *Departamento de Ingeniería Informática y Ciencias de la Computación de la Universidad de Concepción*. Este servidor tiene las siguientes características: Procesador Intel(R) Xeon(R) Gold 5320T CPU @ 2.30GHz, memoria RAM de 256 GB, memorias caché L1i de 32KB, L1d de 48KB, L2 de 1.25MB, L3 de 30MB, y Sistema operativo debian GNU/Linux 11 (bullseye).

6.2. Descripción de los experimentos

Se contemplaron dos grandes áreas a analizar, las cuales son tiempo de ejecución y espacio utilizado de la estructura. Para ello se compiló el código C++ con la extensión de optimización `-O3` para ambas versiones de la estructura.

Se realizaron 3 tipos de experimentos para medir tiempo: *1)* el primero contempla el tiempo de búsqueda de paréntesis entre la versión estática y la versión dinámica de la estructura; *2)* el segundo tipo contempla los tiempos de ejecución de los distintos métodos de inserción y eliminación de la versión dinámica de la estructura; Finalmente, *3)* se miden los tiempos de construcción de la estructura en su versión estática y dinámica.

Con respecto al espacio utilizado por la estructura dinámica, se midió cómo cambia la memoria reservada y utilizada al realizar una serie de operaciones de inserción y eliminación. Para la versión estática, se midió el tamaño de la estructura luego de su construcción.

6.2.1. Datasets y especificaciones generales

Para llevar a cabo los experimentos de tiempo se utilizaron los siguientes cuatro datasets, los que se encuentran disponibles en `http://www.inf.udec.`

`cl/~jfuentess/datasets/sequences.php`:

- `wiki`: Contiene 498.753.914 paréntesis
- `dna`: Contiene 1.154482.174 paréntesis
- `worldleaders`: Contiene 179.236.696 paréntesis
- `coreutils`: Contiene 774.329.686 paréntesis

La cantidad de inserciones y eliminaciones fueron de uno hasta 10 millones en intervalos de un millón. Cabe destacar que por cada iteración se insertan o eliminan 2 paréntesis.

La cantidad de repeticiones de experimentos fueron de 30 veces, reportando el promedio de tiempo de ejecución del experimento.

Para los experimentos de espacio se seleccionó un solo dataset, ya que el comportamiento de la memoria es similar para todos ellos. En particular, se utilizó el dataset `wiki`.

La cantidad de inserciones y eliminaciones para medir el consumo de memoria fueron de 10 millones hasta 50 millones en intervalos de 10 millones, puesto que en estos intervalos se pueden apreciar cambios en la estructura.

Para la ejecución de los experimentos se creó un script para iterar entre las funciones y los datasets a experimentar. Para cada uno de estos últimos se realizaron los siguientes experimentos:

- Ejecutar 30 veces el método de inserción de un nodo hijo izquierdo, para análisis de tiempo de ejecución.
- Ejecutar 30 veces el método de inserción de un nodo hijo derecho, para análisis de tiempo de ejecución.
- Ejecutar 30 veces el método de inserción de un nodo padre, para análisis de tiempo de ejecución.
- Ejecutar 30 veces el método de eliminación de un nodo, para análisis de tiempo de ejecución.
- Ejecutar 30 veces la construcción de la estructura para la versión dinámica, para análisis de tiempo de ejecución.
- Ejecutar 30 veces la construcción de la estructura para la versión estática, para análisis de tiempo de ejecución.

- Ejecutar 30 veces el método match de la estructura para la versión dinámica, para análisis de tiempo de ejecución.
- Ejecutar 30 veces el método match de la estructura para la versión dinámica, para análisis de tiempo de ejecución.
- Ejecutar 30 veces el método match de la estructura para la versión estática, para análisis de tiempo de ejecución.
- Ejecutar 3 veces los métodos de inserción y eliminación para determinar el uso de memoria.

6.3. Análisis de resultados

Como se puede apreciar en el gráfico de eliminación de la figura 10, el tiempo basándose en la cantidad de operaciones realizadas tiene una tendencia de crecimiento lineal pero variable a partir desde el punto 4 millones, puesto que antes de este existe un crecimiento lineal para todos los dataset, mientras que después de este también hay un crecimiento lineal para cada dataset con una diferencia de tiempo debido a que, los tamaños de estos son distintos. Esto se debe al constante análisis del bloque luego de realizar una eliminación, ya que este puede fusionarse con un bloque vecino o bien puede pedir paréntesis a estos mismos, para seguir cumpliendo con la invariante.

Análizando más a fondo los resultados obtenidos, se puede apreciar que los dataset de wiki y world leader tienen tiempos superiores con respecto a los demás, puesto que wiki y world pese a ser de menor tamaño en comparación al resto, tiene mayor probabilidad de realizar reestructuraciones,dado que tiene menor cantidad de nodos hoja. Otro punto a destacar es la estabilización final del último tramo en la mayoría de los dataset, esto se debe a que los paréntesis a eliminar se encuentran en el mismo bloque, considerando el dataset world leader en el último tramo es más abrupto el cambio, ya que es el dataset más pequeño.

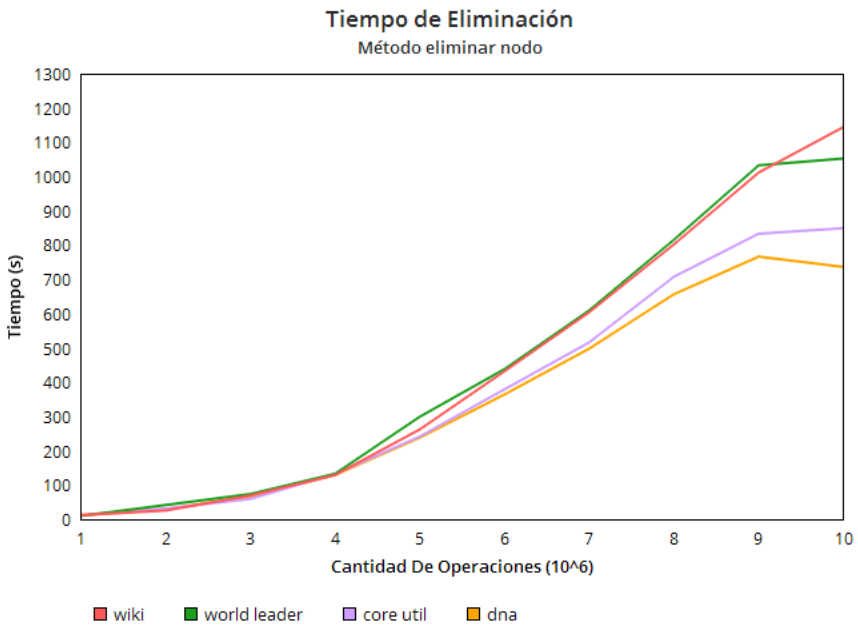


Figura 10: Tiempo de eliminación caso especifico para el dataset `wiki`.

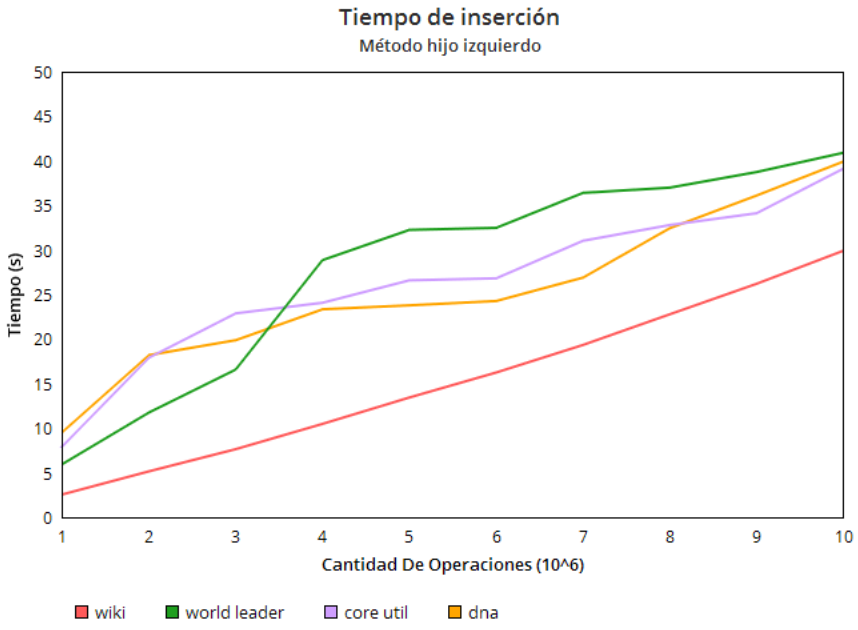


Figura 11: Tiempo del método inserción de un nodo hijo izquierdo.

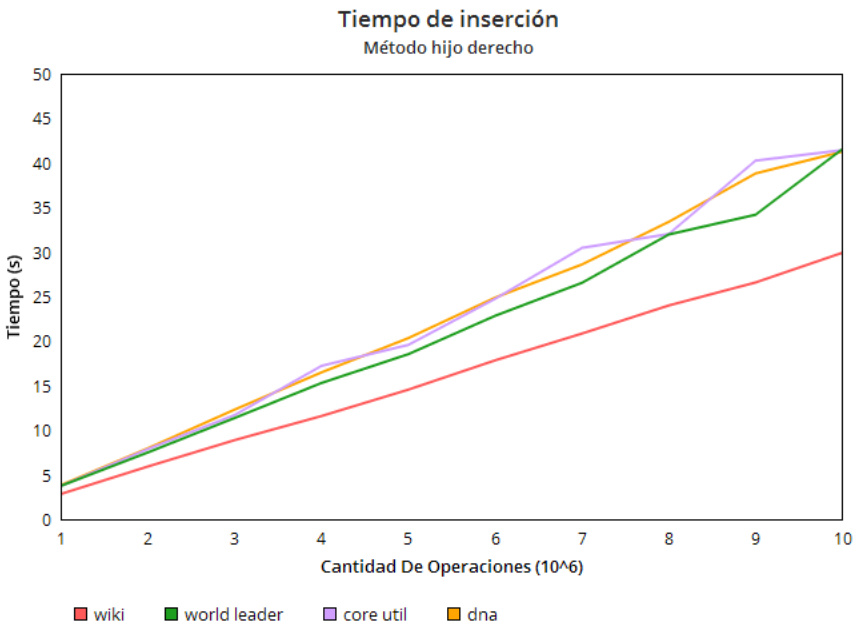


Figura 12: Tiempo del método inserción de un nodo hijo derecho.

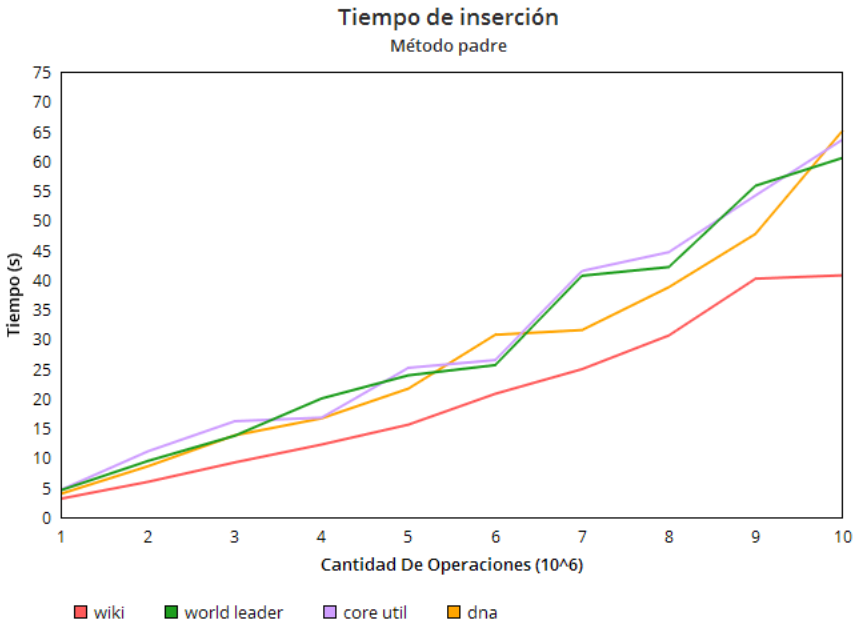


Figura 13: Tiempo del método inserción de un nodo padre

Para el experimento de tiempo en las distintas inserciones, en las figuras 11,12 y 13 se puede apreciar un aumento lineal de tiempo con distinción en los tiempos entre cada experimento, analizando los distintos métodos, se puede apreciar que insertar un nodo hijo izquierdo es más lento que insertar un nodo hijo derecho, puesto que la estructura tiende a utilizar la mayor cantidad de memoria disponible a lo más izquierda posible de esta, desencadenando posibles reestructuraciones, mientras que insertar un nodo hijo derecho es menos probable realizar reestructuraciones, ya que a la derecha de la estructura tiende haber más espacio disponible para futuras inserciones. Con respecto al insertar un nodo padre implica más condiciones, dado que los paréntesis a insertar no siempre serán ubicados en el mismo nodo, por lo cual se produce esta pequeña curvatura en la pendiente de la figura 13. En relación con los resultados obtenidos, se concluye que el método de inserción de un hijo derecho es 1.4 veces más rápido que el método de inserción de un hijo izquierdo, y 1.7 más rápido que el método de inserción de un nodo padre.

Por otro lado, para el análisis de los tiempos de construcción de las dos versiones de la estructura (ver figura 14), se puede apreciar que existe una diferencia de aproximadamente cuatro veces el tiempo de construcción de la estructura dinámica en comparación a la estática. Esta diferencia se debe a que la forma de construcción de las dos versiones, puesto que la versión estática realiza una metodología de construcción desde los nodos hojas hasta llegar el nodo raíz, mientras que la versión dinámica se construyen los nodos hoja, luego desde el nodo raíz se crean y enlazan los nodos interno hasta llegar a los nodos hoja.

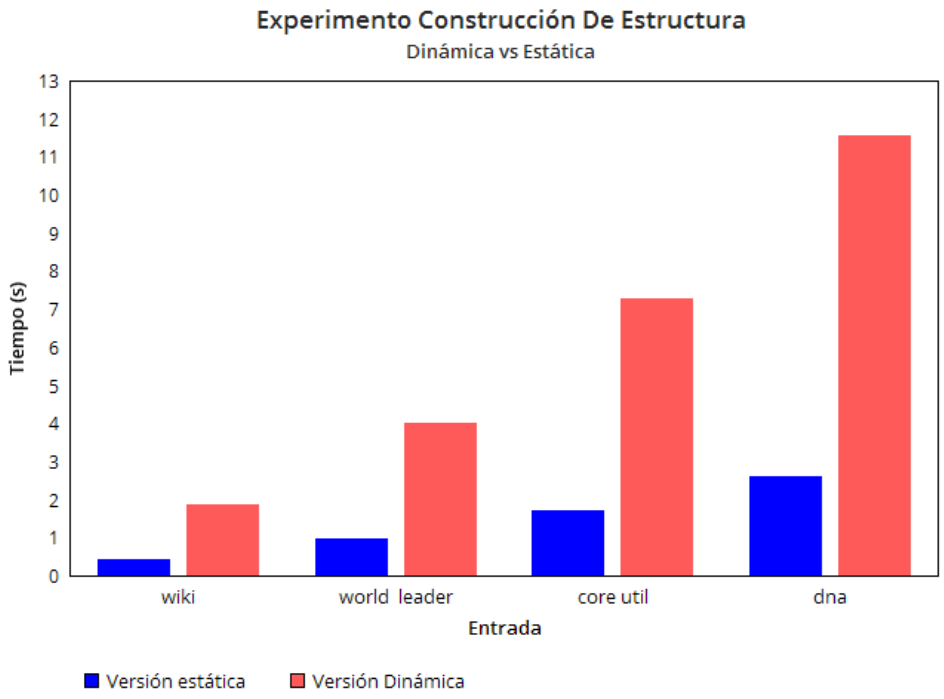


Figura 14: Tiempo Construcción en las distintas versiones de la estructura.

Como se puede apreciar en las figuras 15 y 16, el tiempo de búsqueda, tanto para forward search como para backward search, en ambas versiones es lineal, donde los tiempos para la versión estática es 2 veces más rápida que la versión dinámica para la mayoría de los datasets, exceptuando por `wiki`, donde la versión estática es 1.7 veces más rápida.

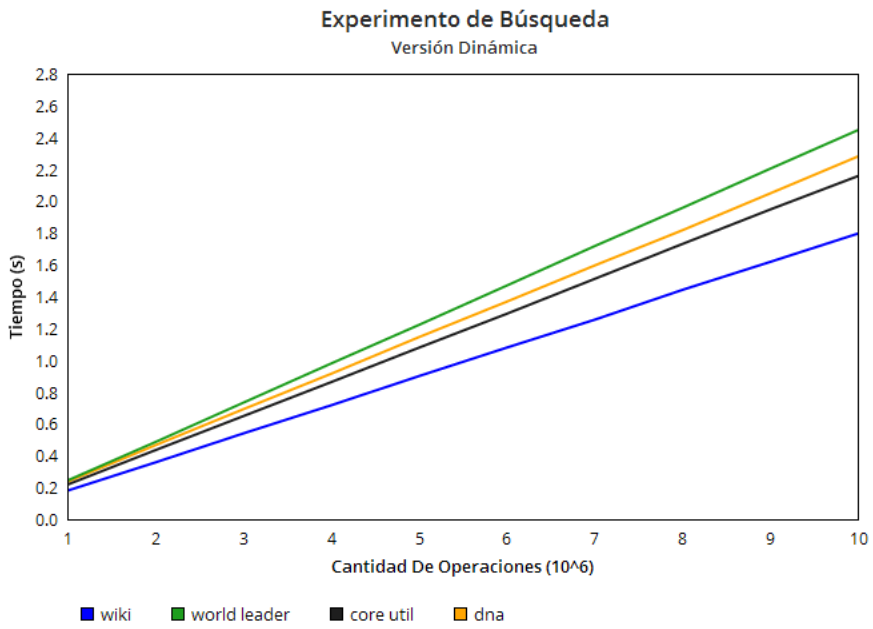


Figura 15: Tiempo de búsqueda para la versión dinámica

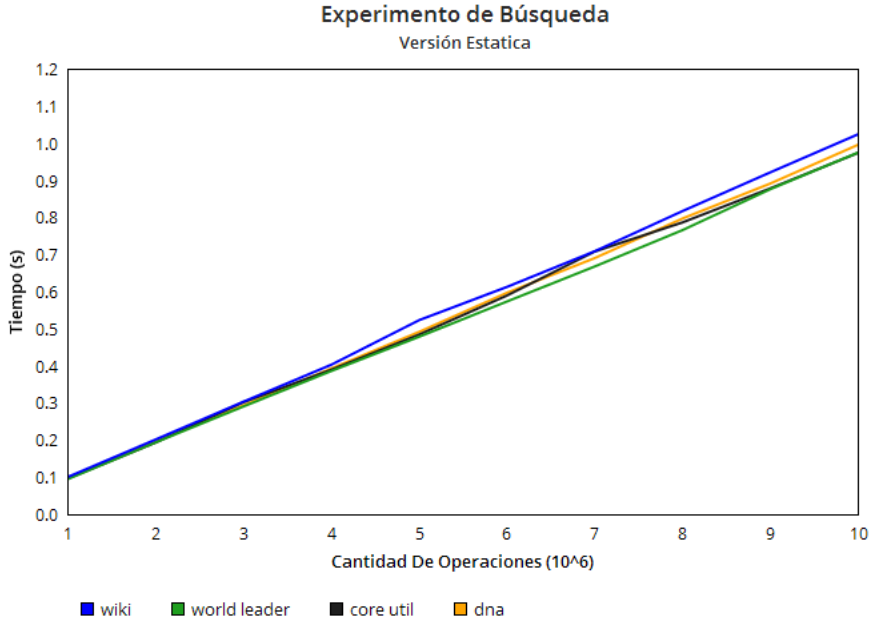


Figura 16: Tiempo de búsqueda para la versión estática

En cuanto a los resultados arrojados en el experimento de memoria utilizada y disponible, las figuras 17,18 y 19 muestan que la memoria utilizada entre las versiones es distinta, puesto que la versión dinámica utiliza más espacio, ya que esta utiliza punteros de memoria, en su contraparte la versión estática no los utiliza. La versión dinámica se pueden realizar modificaciones tanto como inserción y eliminación de paréntesis, se asigna más memoria para evitar que en la primera inserción o eliminación se realice una división de bloques. Se desglosa además que la memoria fluctúa entre el 50 % y un valor cercano al 100 % disponible, pero nunca es 100 %, dado que al utilizar toda la memoria disponible el nodo hoja al cual se le realizan inserciones o eliminaciones deja de cumplir la invariante, vale decir que el factor de carga para cada nodo influye en la memoria final utilizada por la estructura.

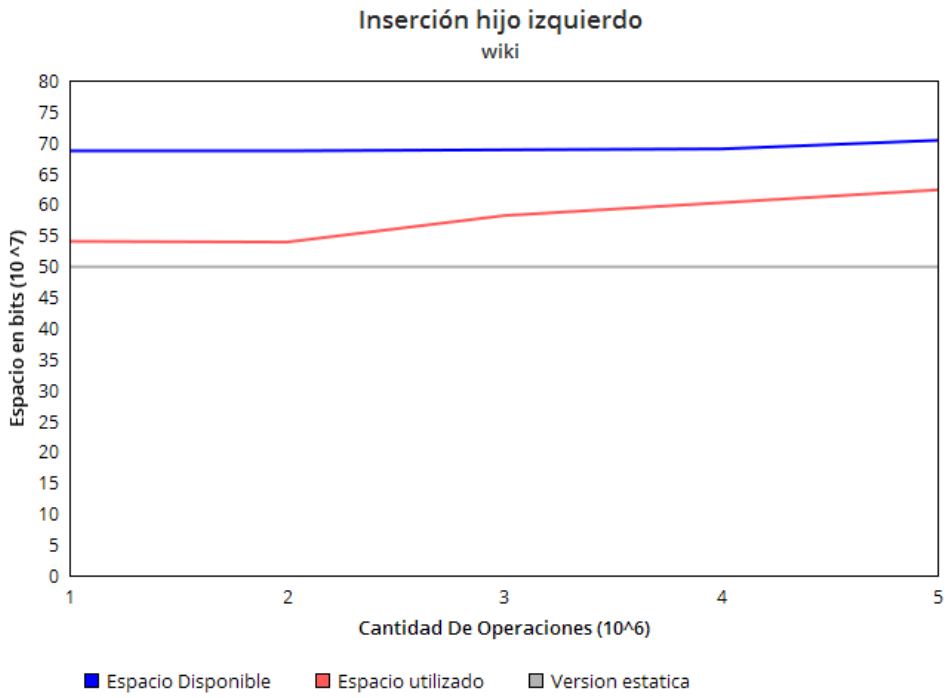


Figura 17: Gráfico de memoria del método inserción de un nodo hijo izquierdo del dataset `wiki`.

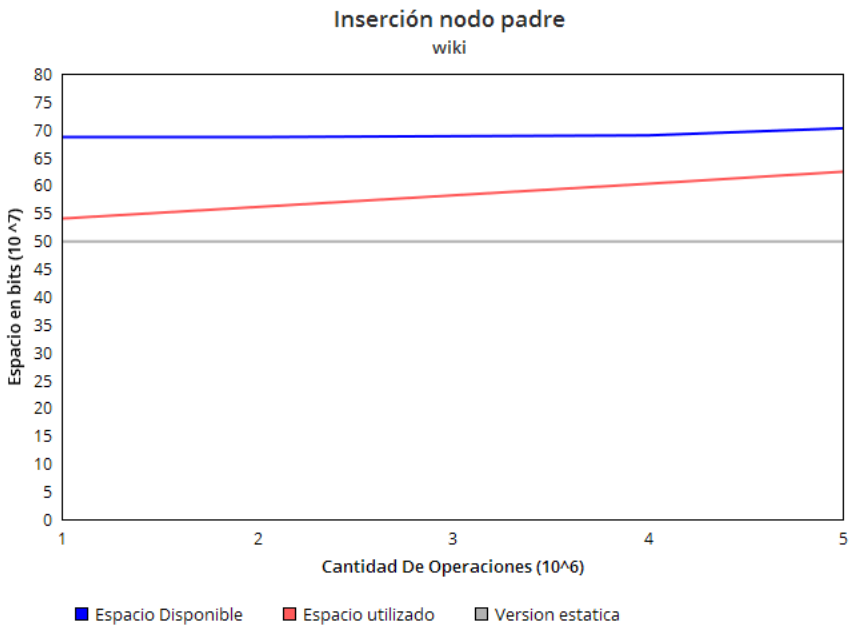


Figura 18: Espacio del método inserción de un nodo padre del dataset `wiki`.

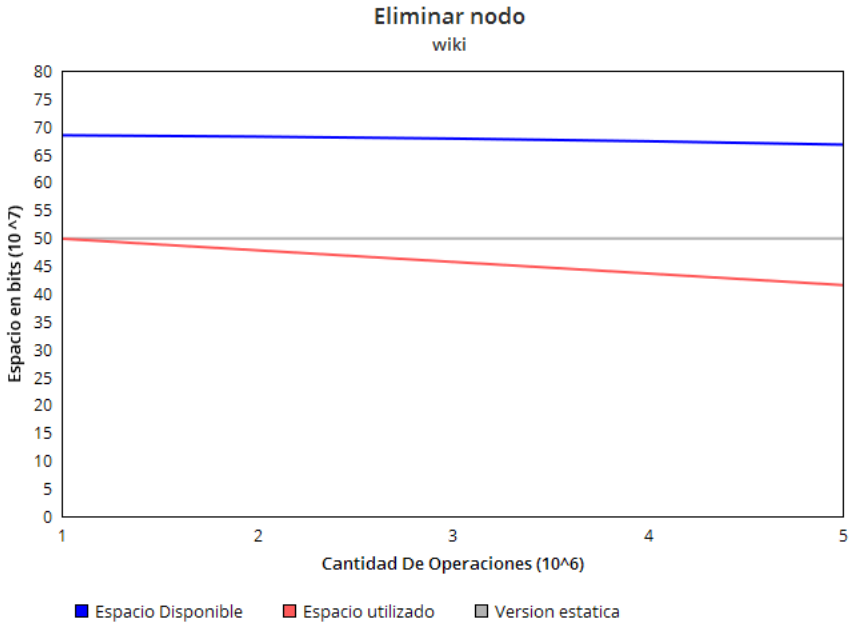


Figura 19: Espacio del método eliminación de un nodo del dataset `wiki`.

7. Conclusiones y trabajo futuro

La problemática abordada al inicio de este informe contempló que no existe una implementación de la versión dinámica del rmMT, por lo cual solo se teoriza su comportamiento. A partir de lo anterior, se estableció como trabajo la implementación práctica y comparación de la versión dinámica del rmMT. De esta manera, a lo largo del desarrollo del trabajo diseñaron e implementaron operaciones de inserción y eliminación, las que manipulan la información almacenada en la estructura. Por medio de la comparación experimental de ambas versiones, se puede concluir que considerando los tiempos de construcción de ambas versiones, la versión estática es 4 veces más rápida que la versión dinámica. Esta diferencia puede estar relacionada con la forma de construir la estructura, puesto que en la versión estática se asigna la memoria antes de construir, mientras que en la versión dinámica se asigna memoria en el momento de creación de los nodos.

Otro factor a considerar son la forma de búsqueda entre versiones, donde a partir de los resultados obtenidos de los experimentos realizados la versión estática es 1.8 veces más rápida que la versión dinámica, dado que la versión estática trabaja más directamente con la secuencia balanceada de paréntesis, mientras que la versión dinámica cada nodo solo conoce su propia sub secuencia de paréntesis.

En cuanto a los resultados reflejados en los experimentos de tiempo de inserción y eliminación, en la versión dinámica se concluye que es una mejora de la versión estática, puesto que se demuestra de forma experimental que la versión dinámica soporta métodos de inserción y eliminación.

Como es sabido, la versión dinámica soporta cambios en comparación a su contraparte que solo permite métodos de búsqueda, mientras que la primera solo debe revisar los tiempos de insertar o eliminar, la segunda al momento de añadir o eliminar paréntesis se debe reconstruir por completo la estructura, es por ello que el tiempo que conlleva realizar cambios en la estática se da por el cálculo de tiempo de construcción por la cantidad de operaciones a realizar. Por lo que se concluye que la versión dinámica es el valor obtenido de lo último mencionado veces más rápida que la estática.

Si bien, en este trabajo se cumplieron los objetivos, no estuvo excepto de problemáticas durante su desarrollo, tal como el diseño preliminar de la creación de la estructura, dado que se tenía contemplada crear la estructura a partir de los nodos hojas hasta llegar al nodo raíz. No obstante, al realizar varios experimentos se desechó la idea, ya que se llegaba a una redundancia

de información entre los nodos, vale decir que un nodo hijo posee la misma información que un nodo padre, por lo cual se rediseñó y se realizó la forma de, a partir de la cantidad de los nodos hojas, se crean los nodos internos, para luego enlazar dichos nodos.

Por otro lado, en el ámbito técnico, estuvo la problemática de los limitantes de la máquina de trabajo, puesto que se utilizó una máquina virtual con limitados recursos, provocando un desarrollo poco fluido, problema que fue solucionado gracias a que se otorgó el acceso a un servidor del *Departamento de Ingeniería Informática y Ciencias de la Computación de la Universidad de Concepción*.

Finalizando este capítulo, si se desea trabajar a futuro con el tema de la versión dinámica del range min max tree, se puede abordar la utilización de programación paralela en la construcción de la estructura, donde por cada par de nodos del range min max tree se genere una hebra, construyendo las ramas de la estructura de forma paralela. Esto se puede profundizar revisando el paper denominado "Parallel construction of succinct trees" [5], el cual presenta algoritmos paralelos para la construcción de la versión estática del range min-max tree. Otro trabajo futuro es la comparación del rendimiento con diferentes entradas, variando la frecuencia de las actualizaciones, además de testear nuestras soluciones en otros ambientes con recursos más limitados, como sería el caso de dispositivos presentes en edge computing.

Referencias

[1] Data never sleeps10.0. https://www.domo.com/data-never-sleeps. Accessed: 2023-03-17.

[2] Practical dynamic entropy-compressed bitvectors with applications. https://link.springer.com/chapter/10.1007/978-3-319-38851-9_8. Accessed: 2023-03-17.

[3] Joshimar Cordova and Gonzalo Navarro. Practical dynamic entropy-compressed bitvectors with applications. In Andrew V. Goldberg and Alexander S. Kulikov, editors, *Experimental Algorithms*, pages 105–117, Cham, 2016. Springer International Publishing.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[5] José Fuentes-Sepúlveda, Leo Ferres, Meng He, and Norbert Zeh. Parallel construction of succinct trees. *Theoretical Computer Science*, 700:1–22, 2017.

[6] Gonzalo Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016.

[7] Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. 10(3), may 2014.

8. Anexo

En esta sección se encuentran todas las tablas con los datos utilizados en la elaboración de cada gráfico incluído en las secciones previas.

Entrada	**Tiempo (s)**
1.000.000	2.5583
2.000.000	5.1560
3.000.000	7.6448
4.000.000	10.4832
5.000.000	13.4390
6.000.000	16.2439
7.000.000	19.3358
8.000.000	22.7766
9.000.000	26.1981
10.000.000	29.9107

Cuadro 1: Tabla de tiempo de ejecución del método inserción nodo hijo izquierdo, del dataset wiki

Entrada	**Tiempo (s)**
1.000.000	2.8498
2.000.000	5.9252
3.000.000	8.8866
4.000.000	11.5887
5.000.000	14.5473
6.000.000	17.8353
7.000.000	20.8400
8.000.000	23.9905
9.000.000	26.5722
10.000.000	29.9052

Cuadro 2: Tabla de tiempo de ejecución del método inserción nodo hijo derecho, del dataset wiki

Entrada	Tiempo (s)
1.000.000	3.1340
2.000.000	5.9654
3.000.000	9.2323
4.000.000	12.2431
5.000.000	15.5901
6.000.000	20.7769
7.000.000	24.9280
8.000.000	30.5935
9.000.000	40.1600
10.000.000	40.722

Cuadro 3: Tabla de tiempo de ejecución del método inserción nodo padre, del dataset wiki

Entrada	Tiempo (s)
1.000.000	11.8180
2.000.000	25.5095
3.000.000	69.2585
4.000.000	128.9650
5.000.000	262.0484
6.000.000	431.8270
7.000.000	604.0303
8.000.000	802.0324
9.000.000	1011.4500
10.000.000	1144.1110

Cuadro 4: Tabla de tiempo de ejecución del método eliminación, del dataset wiki

Entrada	Tiempo (s)
1.000.000	5.9622
2.000.000	11.7602
3.000.000	16.5811
4.000.000	28.8605
5.000.000	32.2487
6.000.000	32.4749
7.000.000	36.4003
8.000.000	36.9887
9.000.000	38.7397
10.000.000	40.9035

Cuadro 5: Tabla de tiempo de ejecución del método inserción nodo hijo izquierdo, del dataset world leader

Entrada	Tiempo (s)
1.000.000	3.7607
2.000.000	7.5059
3.000.000	11.3739
4.000.000	15.2985
5.000.000	18.5300
6.000.000	22.8347
7.000.000	26.5341
8.000.000	31.9571
9.000.000	34.1757
10.000.000	41.5341

Cuadro 6: Tabla de tiempo de ejecución del método inserción nodo hijo derecho, del dataset world leader

Entrada	Tiempo (s)
1.000.000	4.5708
2.000.000	9.4805
3.000.000	13.7283
4.000.000	19.9947
5.000.000	23.8894
6.000.000	25.6100
7.000.000	40.6530
8.000.000	42.1154
9.000.000	55.8018
10.000.000	60.4809

Cuadro 7: Tabla de tiempo de ejecución del método inserción nodo padre, del dataset world leader

Entrada	Tiempo (s)
1.000.000	9.7142
2.000.000	41.6098
3.000.000	73.5483
4.000.000	132.6516
5.000.000	299.1749
6.000.000	438.1606
7.000.000	609.4396
8.000.000	815.5602
9.000.000	1032.6398
10.000.000	1052.5225

Cuadro 8: Tabla de tiempo de ejecución del método eliminación, del dataset world leader

Entrada	Tiempo (s)
1.000.000	7.9472
2.000.000	17.9112
3.000.000	22.8835
4.000.000	24.0735
5.000.000	26.5872
6.000.000	26.8120
7.000.000	31.0251
8.000.000	32.7936
9.000.000	34.1155
10.000.000	39.1194

Cuadro 9: Tabla de tiempo de ejecución del método inserción nodo hijo izquierdo, del dataset core util

Entrada	Tiempo (s)
1.000.000	3.7608
2.000.000	7.8836
3.000.000	11.7097
4.000.000	17.2191
5.000.000	19.5708
6.000.000	24.7444
7.000.000	30.4524
8.000.000	32.0105
9.000.000	40.2443
10.000.000	41.4213

Cuadro 10: Tabla de tiempo de ejecución del método inserción nodo hijo derecho, del dataset core util

Entrada	Tiempo (s)
1.000.000	4.6431
2.000.000	11.1053
3.000.000	16.1671
4.000.000	16.7664
5.000.000	25.1534
6.000.000	26.4519
7.000.000	41.4588
8.000.000	44.6190
9.000.000	54.1735
10.000.000	63.5499

Cuadro 11: Tabla de tiempo de ejecución del método inserción nodo padre, del dataset core util

Entrada	Tiempo (s)
1.000.000	11.4794
2.000.000	31.1796
3.000.000	59.4180
4.000.000	132.3773
5.000.000	241.3162
6.000.000	379.0937
7.000.000	516.1272
8.000.000	707.6080
9.000.000	833.0433
10.000.000	849.2021

Cuadro 12: Tabla de tiempo de ejecución del método eliminación, del dataset core util

Entrada	Tiempo (s)
1.000.000	9.5513
2.000.000	18.1983
3.000.000	19.8732
4.000.000	23.3389
5.000.000	23.7821
6.000.000	24.2652
7.000.000	26.8907
8.000.000	32.4374
9.000.000	36.1020
10.000.000	39.9165

Cuadro 13: Tabla de tiempo de ejecución del método inserción nodo hijo izquierdo, del dataset dna

Entrada	Tiempo (s)
1.000.000	3.8548
2.000.000	7.9745
3.000.000	12.3160
4.000.000	16.4877
5.000.000	20.3334
6.000.000	24.8781
7.000.000	28.5947
8.000.000	33.3780
9.000.000	38.8041
10.000.000	41.2530

Cuadro 14: Tabla de tiempo de ejecución del método inserción nodo hijo derecho, del dataset dna

Entrada	Tiempo (s)
1.000.000	3.9647
2.000.000	8.5927
3.000.000	13.7911
4.000.000	16.6534
5.000.000	21.6551
6.000.000	30.7141
7.000.000	31.5090
8.000.000	38.7196
9.000.000	47.6987
10.000.000	65.0146

Cuadro 15: Tabla de tiempo de ejecución del método inserción nodo padre, del dataset dna

Entrada	Tiempo (s)
1.000.000	12.6769
2.000.000	29.2129
3.000.000	66.2278
4.000.000	129.4082
5.000.000	238.0849
6.000.000	364.0743
7.000.000	497.7934
8.000.000	656.5349
9.000.000	766.2018
10.000.000	736.2805

Cuadro 16: Tabla de tiempo de ejecución del método eliminación de un nodo, del dataset dna

Entrada	**Tiempo (s)** `wiki`	**Tiempo (s)** `worldleaders`	**Tiempo (s)** `coreutils`	**Tiempo (s)** `dna`
1.000.000	0.1006	0.0949	0.0963	0.0976
2.000.000	0.2015	0.1938	0.1934	0.1954
3.000.000	0.3033	0.2904	0.3018	0.2933
4.000.000	0.4029	0.3859	0.3894	0.3927
5.000.000	0.5229	0.4781	0.4841	0.4914
6.000.000	0.6124	0.5731	0.5892	0.5967
7.000.000	0.7085	0.6675	0.7078	0.6901
8.000.000	0.8168	0.7655	0.7862	0.7967
9.000.000	0.9211	0.8757	0.8779	0.8910
10.000.000	1.0237	0.9745	0.9736	0.9955

Cuadro 17: Tabla de tiempo de ejecución de los métodos de búsqueda de la versión estática

Entrada	**Tiempo (s)** `wiki`	**Tiempo (s)** `worldleaders`	**Tiempo (s)** `coreutils`	**Tiempo (s)** `dna`
1.000.000	0.1806	0.2448	0.2187	0.2364
2.000.000	0.3580	0.4864	0.4352	0.4662
3.000.000	0.5400	0.7334	0.6488	0.6922
4.000.000	0.7167	0.9796	0.8624	0.9149
5.000.000	0.9002	1.2222	1.0789	1.1453
6.000.000	1.0792	1.4693	1.2919	1.3682
7.000.000	1.2544	1.7172	1.5111	1.5953
8.000.000	1.4421	1.9576	1.7307	1.8169
9.000.000	1.6180	2.2038	1.9474	2.0487
10.000.000	1.7961	2.4466	2.1576	2.2811

Cuadro 18: Tabla de tiempo de ejecución de los métodos de búsqueda de la versión dinámica

Entrada	Espacio Disponible (bits)	Espacio Utilizado (bits)
10.000.000	686.178.941	539.927.543
20.000.000	686.178.941	538.753.914
30.000.000	687.817.226	581.560.196
40.000.000	689.304.272	602.376.522
50.000.000	703.201.943	623.192.849

Cuadro 19: Tabla de espacio de la estructura para el dataset `wiki` con la función inserción nodo hijo izquierdo

Entrada	Espacio Disponible (bits)	Espacio Utilizado (bits)
10.000.000	686.178.941	539.927.543
20.000.000	686.178.941	560.743.869
30.000.000	687.815.954	581.560.419
40.000.000	689.232.649	602.423.677
50.000.000	702.024.474	624.110.105

Cuadro 20: Tabla de espacio de la estructura para el dataset `wiki` con la función inserción nodo padre

Entrada	Espacio Disponible (bits)	Espacio Utilizado (bits)
10.000.000	684.036.848	498.294.894
20.000.000	681.640.233	477.478.567
30.000.000	678.095.971	456.662.241
40.000.000	673.314.716	435.845.914
50.000.000	667.211.478	415.029.588

Cuadro 21: Tabla de espacio de la estructura para el dataset `wiki` con la función eliminación de un nodo

Entrada	Tiempo versión estática	Tiempo versión dinámica
`wiki`	0.4144	1.8728
`worldleader`	0.9777	4.0164
`coreutils`	1.7192	7.2658
`dna`	2.6036	11.5552

Cuadro 22: Tabla de tiempo de construcción de la estructura en sus dos versiones