UNIVERSIDAD CATÓLICA DEL MAULE

Facultad de Ciencias de la Ingeniería

Escuela de Ingeniería Civil Informática

Profesor Guía

Dr. Ricardo Barrientos

Creación de un Índice Invertido Utilizando Estructuras de Datos Concurrentes (Lista Enlazada y Mapa)

Chien-Hao, Chen

Tesis para optar al

Título Profesional de Ingeniero Civil Informático

Talca, Agosto del 2017

UNIVERSIDAD CATÓLICA DEL MAULE

FACULTAD DE CIENCIAS DE LA INGENIERÍA

ESCUELA DE INGENIERÍA CIVIL INFORMÁTICA

TESIS PARA OPTAR AL

TÍTULO PROFESIONAL DE INGENIERO CIVIL INFORMÁTICO

"Creación de un Índice Invertido Utilizando Estructuras de Datos Concurrentes (Lista Enlazada y Mapa)"

Chien-Hao, Chen

COMISIÓN EXAMINADORA	**FIRMA**
PROFESOR GUÍA Dr. Ricardo Barrientos	_____
PROFESOR COMISIÓN Dr. Paulo Gonzáles	_____
PROFESOR COMISIÓN Mg. Felipe Tirado	_____
NOTA FINAL EXAMEN DE TÍTULO	_____

TALCA, Agosto del 2017

Sumario

Actualmente para recuperar la información desde la Web, los motores de búsqueda utilizan una estructura de datos denominada índice invertido, está compuesta por un mapa de términos, y donde cada término está compuesto por una lista enlazada de documentos, cuales son los archivos que conforman la Web.

Todos los motores de búsqueda Web están forzados a utilizar una plataforma paralela para lograr una alta tasa de respuesta a las consultas. En este contexto, los algoritmos concurrentes son de gran valor, dado que son capaces de explotar eficientemente el hardware de una plataforma paralela.

El objetivo de la presente tesis es proponer y evaluar algoritmo de creación de un índice invertido, utilizando una estructura de datos concurrente sobre una plataforma paralela. Para esto, se utilizan dos enfoques de concurrencia: (1) *lock-free* y (2) *lock-based*. El primero hace la referencia del uso de algoritmos que no implican un bloqueo de hilos. El segundo implica el bloqueo de hilos para implementar la sincronización entre ellos. También, se implementa el algoritmo de búsqueda binaria de manera concurrente para la inserción de los términos en el índice invertido.

Los experimentos se llevaron acabo comparando los algoritmos implementados con métodos presentes en el estado del arte, en donde nuestra propuesta basada en el concepto de *lock-free* obtuvo el mejor rendimiento. Esto se debió al costo asociado a los bloqueos de código y de hilos, necesarios para implementar la versión *lock-based*.

Índice general

Índice de figuras

Capítulo 1

Introducción

La presente tesis implicó una investigación en las áreas de Computación Paralela y Estructuras de Datos Concurrentes. En esta tesis se proponen e implementan algoritmos basados en el concepto de *Non-Blocking* [Her11]: *wait-free*, *lock-free* y *lock-based*, junto con el método de *búsqueda binaria* [Gar15] para aumentar el rendimiento en la creación de una estructura de *índice invertido* [Man08].

Actualmente existe un gran volumen de datos y el tamaño aún está creciendo día a día. Para rescatar la información necesaria, desde una cantidad inmensa de archivos se requiere una estructura de datos eficiente como el índice invertido. Sin embargo, el algoritmo de su creación no es simple; se debe considerar el diseño fundamental de la estructura de datos, la implementación del algoritmo de búsqueda, y los problemas asociados a la concurrencia.

En el capítulo 2 se viene el Marco Teórico mostrando las teorías y dificultades asociadas al tema en cuestión, para luego, se presenta el Estado del Arte, que constituye los trabajos de investigaciones vinculadas de la presente tesis. Luego, se aborda la parte de Desarrollo donde se describen los algoritmos propuestos e implementados. Se continuará con la sección de Experimentos donde se podrá apreciar el despliegue de la medición y el análisis de los resultados, y finalmente, dar cierre a la presente tesis con el capítulo de Conclusiones donde se condensa lo principal de esta tesis.

1.1. Objetivos

El objetivo principal de la presente tesis es evaluar la eficiencia sobre una plataforma paralela de un método algorítmico para la creación de índices invertidos.

Los objetivos específicos se definen a continuación:

- Diseñar el modelo de mapa y estructuras de datos a utilizar.
- Implementar búsqueda binaria sobre las estructuras de datos seleccionadas.
- Implementar la creación de un índice invertido sobre una plataforma paralela.
- Buscar las estrategias para coordinar la comunicación entre los hilos (threads).
- Evitar los problemas que surgen del acceso a recursos compartidos.
- Medir y comparar el rendimiento de los algoritmos implementados.

Capítulo 2

Marco Teórico

2.1. Computación Paralela

Tradicionalmente un *software* está diseñado en programación secuencial, las instrucciones del programa se ejecutan una después de otra. Por otro lado, la computación paralela utiliza multi-recursos, ejecuta varias instrucciones de manera simultáneamente (ver figura 2.1).

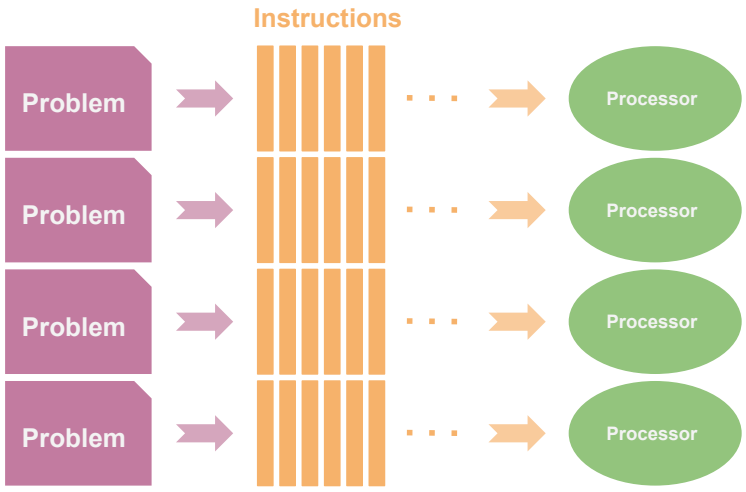

Figura 2.1: Una demostración simple de la computación paralela.

2.2. Inanición

Inanición (*starvation*) es un problema que puede ocurrir en la computación paralela, la situación de inanición ocurre cuando existe un proceso nunca logra el recurso necesario para terminar o avanzar su trabajo. Frecuentemente existe una confusión entre la definición de Inanición y Bloqueo Mutuo, siendo que el problema de inanición es un tipo de *livelock*.

Livelock es un caso especial de inanición de recursos para un *Deadlock*, generalmente ocurre en alguna condición específica en que los procesos no progresan, por ejemplo, 2 procesos están tratando de acceder a un recurso compartido en el mismo momento, lógicamente uno de ellos debe ceder su paso al otro para permitir el desarrollo del proceso completo. Pero ambos ceden el recurso en el mismo tiempo y vuelven a solicitar el recurso en el mismo momento, y así sucesivamente se repitirá la misma situación hasta que uno se salga del ciclo, tal como muestra la figura 2.2.

Figura 2.2: Un ejemplo simple de la inanición.

La utilización de prioridades (*scheduling*) es un factor que puede ocurrir en este tipo de problemas, ya que los procesos de alta prioridad se ejecutan siempre primero, restrigiendo procesos de baja prioridad. Hasta un cierto momento, un proceso de alta prioridad podría necesitar un resultado del proceso de baja prioridad lo que conducirá a que no se pueda completar.

2.3. Bloqueo Mutuo

El bloqueo mutuo (*Deadlock*) o interbloqueo es un problema común en Sistemas Distribuidos, Computación Concurrente y Computación Paralela. Los interbloqueos surgen de la competición de recursos cuando existen más de 2 procesos que están esperando mutuamente un recurso para terminar el trabajo del otro proceso y así continuar al siguiente paso, pero ninguno de los procesos puede avanzar, ya que el recurso necesario está ocupado por el proceso que también se encuentra en estado de espera.

El diagrama de figura 2.3 es un ejemplo simple de bloqueo mutuo, donde 2 procesos se quedan paralizados por los recursos que sostienen entre ellos, donde ambos no ceden el paso para liberar el recurso al otro. Esta situación puede darse con una mayor cantidad de hilos en interacción, cuyo progreso de programa requiere el avance de los hilos, produciendose bloqueo mutuo deteniendo el avance cuando se produce una intención de bloqueo, así el flujo de trabajo no puede continuar.

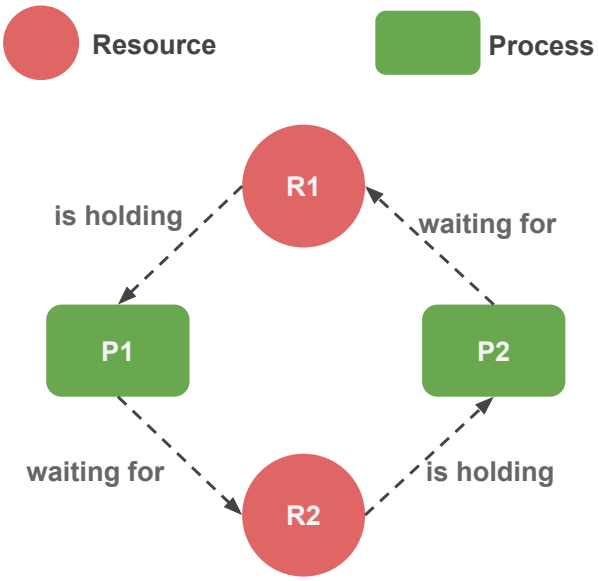

Figura 2.3: Un ejemplo simple de bloqueo mutuo.

2.4. Problema ABA

El problema ABA es un fenómeno que puede ocurrir durante un programa multi-hilo en la operación de lectura (*read*). Si ocurre cuando una variable que cambia su valor, y otra acción de *lectura* no alcanza a detectar el cambio anterior, produciendo una re-escritura cambiando el valor actual.

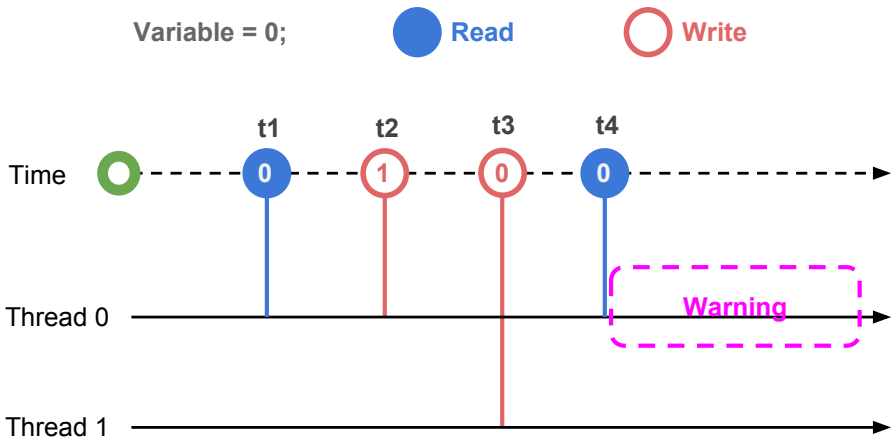

Figura 2.4: Un ejemplo simple de problema ABA.

La consecuencia del problema ABA puede manifestarse como se muestra en el esquema 2.4, donde se representa el avance paralelo de los hilos 0 y 1 los cuales acceden a una variable, en el momento “t4” el hilo 0 necesita leer el valor de la variable para determinar las acciones del proceso posterior, producto que durante t2 y t4, la variable es modificada por otro hilo. Por lo tanto, la acción posterior del hilo 0 puede ser inesperada.

2.5. Lock-Based

El bloqueo es un tipo de mecanismo para el control de acceso a los datos o a la ejecución de código. El objetivo del *lock* es impedir que otros hilos puedan realizar alguna transacción como “leer” y/o “escribir”, mientras el hilo todavía está usando el bloqueo. La adquisición de bloqueo se obtiene cuando otro hilo libera el bloqueo. Cuando hay otros hilos esperando un mismo *lock*, la adquisición de éste va a depender de la prioridad.

Definición

Lock-Based realiza la operación de bloqueo, evitando la ejecución hasta que otro hilo libera el recurso [Ram13a].

Lock-Based tiene 2 tipos de protocolos:

- **Exclusivo**: Este tipo de bloqueo sólo tiene los 2 estados (Bloqueado/Desbloqueado) para definir el uso de datos por los hilos requeridos.
- **Compartido**: El uso de los datos es más flexible por la regla de este tipo de bloqueo, los datos se pueden "leer", pero no permite realizar la actualización cuando está bloqueado.

2.6. Lock-Free

Lock-Free es un tipo del algoritmo *non-blocking* [Her11] que se caracteriza por no ocurrencia de exclusión mutua en el diseño de operaciones de recurso compartido. Aunque la operación de un hilo se haya interrumpido, las operaciones de otro hilo no se verán afectadas.

El objetivo de *lock-free* es la disminución del uso de bloqueo y la reducción del tiempo de espera para mejorar la concurrencia en el sentido de la escalabilidad. Elimina la posibilidad de que ocurra la condición de carrera, el bloqueo mutuo y la inversión de prioridades. Al tener mucho factor oculto, afecta el rendimiento óptimo total del programa.

Definición

Un método de *lock-free* garantiza que siempre hay por lo menos, un hilo avanzando su progreso de tarea, terminando en un cierto número de paso finito [Ram13a].

En el siguiente diagrama 2.5 identifica la programación es de *lock-free*:

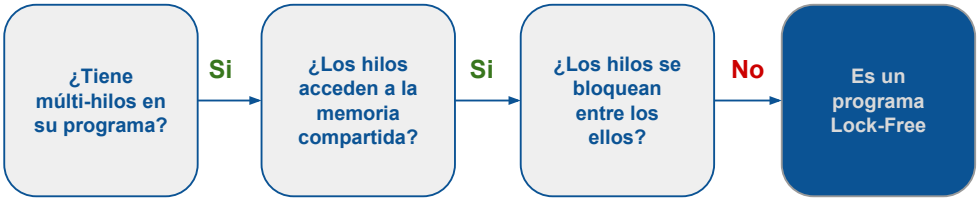

Figura 2.5: Un esquema para identificar la programación de lock-free [Pre12].

2.7. Wait-Free

Wait-Free es un tipo de algoritmo *non-blocking* como *lock-free*, pero cuyo implementación es más compleja. La propiedad de *wait-free* garantiza que todos los hilos tienen su propia duración en el tiempo para completar su progreso y terminar las tareas finalmente. Dicho método de implementación es realizado escasamente, ya que el algoritmo debe incluir las características de *Starvation-Free* (el recurso requerido siempre está disponible para un hilo deseado) y *Obstruction-Free* (cuando un hilo está en la condición de aislamiento, va a terminar su tarea durante un número finito de pasos). Muchas veces las implementaciones de la misma estructura de datos sobre *wait-free* tienen peor rendimiento que la del algoritmo *lock-free*, puesto que para cumplir las propiedades del algoritmo se requiere mucho el uso de contención y generalmente el contenido de las operaciones es más extensa [Ram13a].

Definición

Wait-Free garantiza que todos los hilos pueden finalizar su ejecución en un número de paso limitado [Ram13a].

El algoritmo *lock-free* tiene un límite de escalabilidad para el rendimiento de una estructura de datos, es decir, si se incrementa la cantidad de hilos hasta un cierto punto, se notará que no se logra acelerar o mejorar el rendimiento total del programa. Esto, debido al hecho de cumplir la propiedad de *wait-free*, se necesita mucho más tiempo en el procesamiento de menor cantidad de hilos, pero luego si se incrementa el número de hilos, se obtendrá un mejor rendimiento al realizar la comparación con *lock-free* (ver figura 2.6).

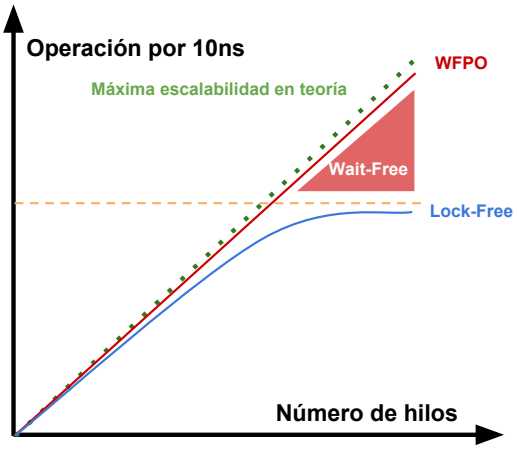

Figura 2.6: El diagrama de rendimiento entre lockfree y wait-free [Ram13b].

2.8. Índices invertidos

Un índice invertido es una estructura de datos de la búsqueda de índices, usualmente este método se utiliza para facilitar la búsqueda de información requerida o la recuperación de información en motores de búsqueda web [Man08]. Si el sistema busca información desde el contexto según las claves de las palabras ingresadas en un gran volumen de archivos o documentos, se vuelve muy ineficiente, lo cual es muy costoso por el tiempo y los recursos. Entonces, la idea de un índice invertido es utilizar la estructura de datos como un almacenamiento, donde los términos de palabras se almacenan como índices, vinculando el nombre o la ruta de documentos asociados (ver figura 2.7).

Preprocesamiento de los datos

Tal como se describió en el concepto de índices invertidos, el producto final de éstos facilita la entrega del resultado de una búsqueda, dado que se ha realizado un trabajo de revisión y recolección de términos a priori. Luego los puntos importantes, pre-procesos y las condiciones que siguen los motores de búsqueda antes de la creación del índice invertido.

- Los términos almacenados deben ser normalizados en función de la raíz de la palabra.
- Los índices no pueden ser repetidos (*Key*).
- El orden de los índices deben ser sistemático, por ejemplo, ordenan lexicográficamente, u

otra forma que implique mejorar el rendimiento de búsqueda posterior.

- Los contenidos de cada índice entre ellos pueden ser repetidos, pero en el mismo campo no pueden tener el nombre de documentación repetido (*Value*).

- Las listas de ocurrencias asociadas de índices deben ser ordenadas por ranking de uso.

Un ejemplo de índices invertidos

Se cuenta con los siguientes 3 documentos D1, D2 y D3, donde la parte derecha es el contenido de cada uno:

D1 = ”Anoche se puso a llover.”
D2 = “¿Le gusta pescado?”
D3 = “Anoche comí pescado.”

Por la regla de índices invertidos se debe aplicar un filtro de contexto de archivo para sacar las etiquetas de códigos de programación. Después aplicar algún método o algoritmo de stemming (opcional) para reducir la cantidad de palabras innecesarias y buscar la raíz de los términos.

Términos	**Raíz**
Comí	Comer
Gusta	Gustar
Puso	Poner

Tabla 2.1: Un ejemplo de los términos.

Al transferir los 3 documentos al proceso de creación de índices invertidos, y se aplica un algoritmo de ranking para cada ocurrencia de los documentos (los factores de ranking pueden ser: el tiempo en que fue generado el término, la ocurrencia por términos asociados, el nivel de importancia, etc.), el resultado de trabajo se mostrará de la siguiente manera:

Los siguientes puntos descritos son desafiantes, y a su vez, muy importantes para la creación de índices invertidos, por lo que se deben tener en cuenta [ZPP+13]:

- La optimización del tiempo de construcción, actualización y búsqueda de información.

- La forma de tratamiento a los vocabularios, cómo se almacenan y se relacionan entre ellos, debido a la gran cantidad de espacio que utiliza.

- La tolerancia a los fallos.

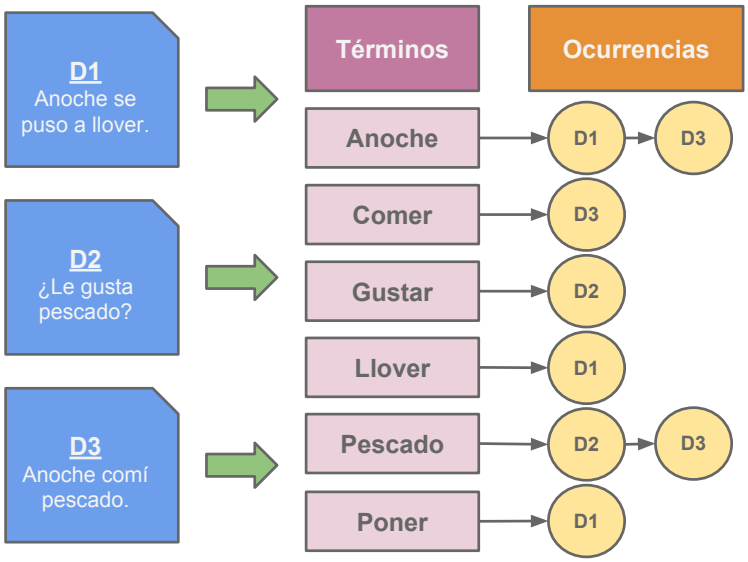

Figura 2.7: Ejemplo de un índice invertido.

2.9. Estructura de datos: Lista enlazada

La lista enlazada [Par01] es un tipo de estructura de datos básica que se usa comunmente, ya que su modelo de diseño es simple, pudiéndose personalizar la estructura de datos según la necesidad del usuario. Básicamente la lista enlazada se construye con el grupo de nodos y utiliza los punteros para asociarse entre ellos. La forma de vincular a los nodos es secuencial, es decir, que un nodo puede referenciarse solo a otro nodo. La lista enlazada no tiene restricción para el orden de los nodos, la cual no tiene que ser ordenada según el peso de cada nodo o depende de alguna condición que si esté definida. Por lo tanto la complejidad de tiempo para una lista enlazada se da de la siguiente forma: la inserción es *O(1)*, la búsqueda es *O(n)*.

Ventajas

- Fácil de implementar la inserción y eliminación de nodos. Por la característica de puntero, el manejo de memoria será dinámico.
- Fácil de personalizar el diseño de lista enlazada. El usuario puede modificar la estructura de lista enlazada fácilmente por el modelo de nodo.
- Lista enlazada es un tipo de estructura de datos dinámicos. Ya que la eliminación o inserción de nodos no afecta el espacio de los otros nodos.

Desventajas

- Ocupa más espacio de memoria que una estructura de datos de arreglo, ya que usa puntero para asociar los nodos.
- No se puede acceder a cualquier nodo directamente, hay que realizar la búsqueda desde el nodo inicial por diseño de lista simple enlazada.
- Por la limitación de lista simple enlazada, la búsqueda de un nodo no se puede retroceder al nodo anterior. Si quiere implementar lista doblemente enlazada, se gasta más espacio de memoria.

2.9.1. Tipo de lista enlazada

La variedad de diseño de la lista enlazada es mucha, básicamente la estructura se construye por los nodos (elemento) los cuales se conocen como los datos, y en el enlace de los nodos se ocupa el puntero para vincularlos. La cabecera y cola de lista enlazada también son importantes para definir un diseño de modelo.

Lista simple enlazada

La cabecera siempre va a ser el último nodo insertado y la cola será un nodo vacío. Cada nodo contiene un puntero para apuntar al otro nodo, construyéndose en forma secuencial como una lista.

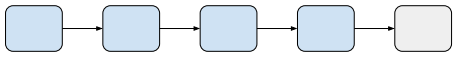

Figura 2.8: Lista simple enlazada

Lista doblemente enlazada

La lista doblemente enlazada tiene mayor flexibilidad de retroceder al nodo anterior que el diseño de lista simple enlazada. Por lo tanto, ahora los nodos contienen 2 punteros en cual uno apunta al siguiente nodo, y otro apunta al nodo anterior. Además, el uso de puntero ocupa un espacio de extra de memoria, así como también las operaciones (Inserción, Eliminación) de nodo va a ser más complejos. Por la razón anterior, la lista doblemente enlazada tiene sus desventajas si se realiza una comparación con el modelo de lista simple enlazada.

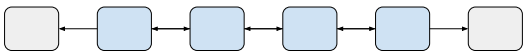

Figura 2.9: Lista doblemente enlazada

Lista enlazada circular

En este caso, el último nodo vacío se eliminó y se hizo un enlace con el primer nodo. Esta forma de lista enlazada es como una cola circular, donde el cuerpo principal aún es una lista simple enlazada.

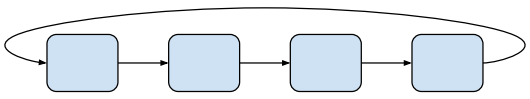

Figura 2.10: Lista enlazada circular

Lista múltiple enlazada

Generalmente la lista múltiple enlazada se utiliza en casos más especiales, ya que en su diseño, como se muestra en la figura 2.11, cada nodo contiene su otra propia lista enlazada a parte. La lista enlazada principal se usa como el campo de jerarquía mayor (por ejemplo: País, Años, Encargo, etc.) y las sub-listas enlazadas contienen información de jerarquía menor.

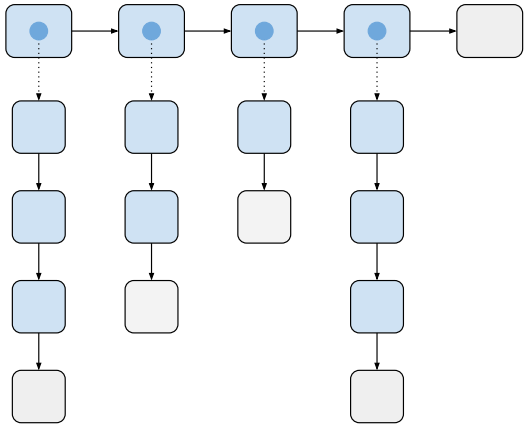

Figura 2.11: Lista múltiple enlazada

Skip List

Skip list [Pug90] (lista por saltos) es un tipo de estructura de datos abstracto, su diseño de modelo está basado en la lista enlazada. *Skip list* usa su característica de la probabilidad para mejorar el rendimiento de las operaciones como la consulta, inserción o eliminación de elementos. Este tipo de estructura de datos genera los niveles por cierta probabilidad y se vinculan entre los nodos si tienen un mismo nivel asociado, tal como muestra la siguiente figura 2.12:

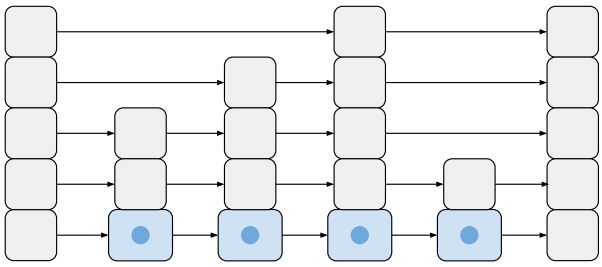

Figura 2.12: Lista por saltos con 5 niveles

Una *skip list* tiene que tener un nodo de cabecera para definir el nivel máximo que se puede agregar en cada nodo, y la probabilidad (p) de generar más niveles, el cual se puede personalizar dependiendo de su uso.

Entonces, para iniciar una *skip list* es necesario dar un valor de nivel máximo y la probabilidad (p) de agregar una nueva capa. Una restricción de *skip list* exige que los elementos insertados deben ser ordenados de alguna forma en la lista almacenada.

El rendimiento de esta estructura de datos tiene relación con el valor de máximo nivel (L) que se define para la lista. Según *William Pugh*, autor de *skip list*, hay que saber la cantidad total de los elementos(n) que se insertarán y la probabilidad (p) de generar nueva capa en cada nodo. Con estos datos, se puede definir un valor óptimo (L*) para obtener el mejor rendimiento en la ejecución de programa.

$$L* = \log_{1/p} n \tag{2.1}$$

n: *Cantidad total de elemento insertado*

p: *Probabilidad de nivel*

$$Complejidad: \ Promedio\ O(log\, n);\ Peor\ caso\ O(n);$$

2.10. Estructura de datos: Mapa

Este tipo de estructura de datos también se conoce como: vector asociativo, diccionario, tabla de consulta, hash o mapeador. La funcionalidad del mapa es asociar las claves (único) con los valores correspondientes para facilitar las tareas posteriores que se necesitan realizar a la búsqueda de datos e informaciones. La estructura de mapa no tiene la condición de que las claves tienen que ser ordenadas de manera lexicográfica, ya que el objetivo principal del mapa es hacer la asociación entre las claves y el conjunto de los valores. Pero si se considera el rendimiento de la búsqueda posterior, el mapa se puede dejar en la condición de ordenamiento que se desee por algún algoritmo de búsqueda que se vaya a utilizar.

Las operaciones de mapa que se pueden realizar son: inserción, eliminación, modificación y búsqueda. Usualmente el diseño de esta estructura de datos es como un matriz de tamaño *N* filas y dos columnas, cada espacio de la fila se ocupa por cada clave, en la primera columna por la misma razón se guarda el valor de clave, la segunda columna se almacenan todos los valores asociados (datos) de la clave. Usualmente un modelo de vector asociativo es como la siguiente figura 2.13:

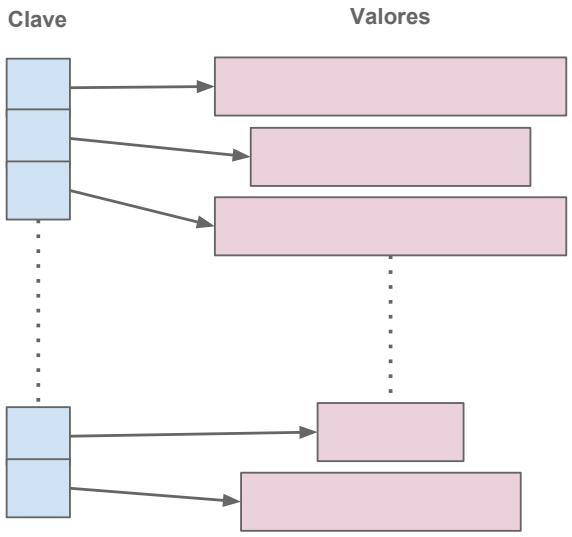

Figura 2.13: La estructura de datos - Mapa

El diseño de mapa es muy importante, tiene su punto relevante en que puede influir mucho con el rendimiento de las operaciones del mapa. Los factores pueden manifestarse en cómo se ordena el almacenamiento de claves y valores, el modelo de la estructura de mapa, el algoritmo de búsqueda implementada, la coordinación de las tareas de hilos, etc.

2.11. Búsqueda Binaria

La búsqueda binaria es un tipo del algoritmo de búsqueda, que se puede realizar en un arreglo ordenado. El algoritmo hace la comparación del valor entre el número y el elemento de la mitad en el arreglo correspondiente en un instante del tiempo, puesto que el rango de objetivo debe cumplir la condición que se estima. Si el valor esperado no coincide a la condición requerida, el

algoritmo se aplica recursivamente sobre la otra mitad inferior o superior, y así sucesivamente hasta que el rango de búsqueda sea suficientemente pequeño.

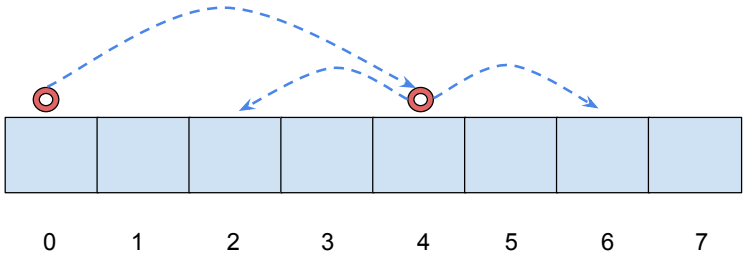

Figura 2.14: Ejemplo de búsqueda binaria

Si nuestro arreglo tiene n elementos, el mejor caso de búsqueda con la complejidad de tiempo va a ser *O(1)*, el peor caso va a ser *O(log n)*.

Algoritmo 1 *BS*: La búsqueda binaria [Gar15].

BúsquedaBinaria(Arreglo *A*,Valor *v*, Bajo *b*, Alto *a*)

1: *//Inicializa con b = 0, a = N-1*
2: *//Invariantes: v >A[i] para todo i <b && v <A[i] para todo i >a*
3: **if** $(a < b)$ **then**
4: **return** not_found
5: **end if**
6: $m = (b + a)/2$
7: **if** $(A[m] > v$ **then**
8: **return** $BúsquedaBinaria(A, v, b, m - 1)$
9: **else if** $(A[m] < v)$ **then**
10: **return** $BúsquedaBinaria(A, v, m + 1, a)$
11: **else**
12: **return** m
13: **end if**

Capítulo 3

Estado del Arte

A continuación, se realiza el análisis y la recolección de información sobre los investigadores que hicieron estudios semejantes y relevantes, relacionados a la presente tesis. La siguiente tabla 3.1 muestra el resultado de la búsqueda de las estructuras de datos concurrentes, que son diseñadas con el algoritmo *Non-Blocking* y son códigos libres.

Las estructuras de datos *Non-Blocking* en la tabla 3.1, no aplican bloqueo ni la exclusión mutua para garantizar la seguridad en hilos, éstos utilizan la operación atómica que no interrumpe la ejecución de los procesos. Se realiza la sincronización de hilos en el espacio de usuario sin que interactue con el sistema operativo directamente (no todos los hardware o arquitecuras soportan la misma instrucción atómica, para más detalles se requiere revisar las documentaciones oficiales de cada uno de ellos indicados en las referencias de la tabla 3.1).

Oracle Java *ConcurrentHashMap* es una estructura de datos utilizada para la programación en muchas áreas, ya que su característica de gran flexibilidad, facilita el uso de la indexación de los elementos; si se utiliza el principio de la operación atómica, no se aplica el bloqueo en el diseño de código.

La librería Boost C++ posee estructuras de datos de tipo *lock-free* (cola, cola circular y pila), consideran el uso de las operaciones atómicas, la asignación de memoria y el manejo de excepción; también se puede definir el tamaño fijo o la capacidad de la estructura de datos; más descripción y detalle de ello se encuentra en la documentación oficial de éstos.

Categoría	**Nombre**	**Lenguaje**	**Fuente**
Stack	Stack	C	Liblfds [Hen04]
Stack	Stack	C++	Boost [Her11]
Stack	Stack	Ada	NBAda [Gid08]
Stack	Trieber Stack	C	ASCYLIB [Tre86]
Stack	Trieber Stack	C++	CDS [Tre86]
Stack	FCStack	C++	CDS [HT10]
Queue	Queue	C	Liblfds [MS96]
Queue	Queue	C++	Boost [Her11]
Queue	Queue	Ada	NBAda [Gid08]
Queue	Deques	Ada	NBAda [Gid08]
Queue	Bounded Queue	C	Liblfds [MS96]
Queue	Priority Queue	Ada	NBAda [Gid08]
Queue	Lotan and Shavit priority queue	C	ASCYLIB [SL00]
Queue	MSQueue	C	ASCYLIB [MS96]
Queue	Basket Queue	C++	CDS [MH07b]
Queue	MSQueue	C++	CDS [MS96]
Queue	RWQueue	C++	CDS [MS96]
Queue	MoirQueue	C++	CDS [MS96]
Queue	Optimistic Queue	C++	CDS [ELM08]
Queue	Segmented Queue	C++	CDS [Afe10]
Queue	FCQueue	C++	CDS [HT10]
Queue	Tsigas Cycle Queue	C++	CDS
Queue	Vyukov MPMC Cycle Queue	C++	CDS
Queue	Ringbuffer (circular queue)	C	Liblfds [MS96]
Queue	Ringbuffer (circular queue)	C++	Boost [Her11]
List	LinkedList	C++	Lawrence Bush [Bus02]
List	Harris linked list	C	ASCYLIB [Har01]
List	Michael linked list	C	ASCYLIB [Mic02]
List	Harris linked list with ASCY	C	ASCYLIB [DT15]
List	Fraser skip list	C	ASCYLIB [Fra04]
List	Fraser skip list with Herlihy's optimization	C	ASCYLIB [MHS11]

List	Free List	C	Liblfds
List	Single LinkedList (ordered)	C	Liblfds
List	Lazy List, Single LinkedList (ordered)	C++	CDS [SHS05]
List	Michael List, Single LinkedList (ordered)	C++	CDS [Mic02]
List	Single LinkedList (unordered)	C	Liblfds
Map	HashMap	C	Liblfds [Mic02]
Map	Linear map	C++	Junction [Pre16]
Map	Leapfrog map	C++	Junction [Pre16]
Map	Grampa map	C++	Junction [Pre16]
Map	Spllited Ordered List Map	C++	CDS [OS03]
Map	Stripped Map	C++	CDS [Her11]
Map	Michael HashMap	C++	CDS [Mic02]
Map	Skip List Map	C++	CDS [Pug90]
Map	Feldman Hashmap	C++	CDS [SF13]
Map	HashMap	Java	Oracle [Ora16]
Map	Map	Scala	Root package
Map	Cuckoo Map	C++	CDS [M.H07a]
Tree	Binary Tree	C	Liblfds [Bro10]
Tree	Ellen et al. binary search tree	C	ASCYLIB[Ell10]
Tree	Howley and Jones binary search tree	C	ASCYLIB[How12]
Tree	Natarajan and Mittal binary search tree	C	ASCYLIB[Nat14]
Tree	Ellen's Binary Tree	C++	CDS
Tree	Ramachandran's Binary Search Tree		[Ram15]
Tree	Suffix Tree	Java	Concurrent Suffix Trees [Gal12]
Tree	Radix Tree	Java	Concurrent Suffix Trees [Gal12]
Tree	B Tree	C	B-tree project
Tree	Red Black Tree	C	University of Cambridge [FH07]
Tree	Red Black Tree		[Nat13]
Tree	Bronson AVL Tree	C++	CDS
Tree	Trie	Scala	Root package
Tree	Heap	C++	CDS [G.H96]
Set	Cuckoo Set	C++	CDS [M.H07a]

Set	Stripped Set	C++	CDS [Her11]
Set	Feldman Hash Set	C++	CDS [SF13]
Set	Skip List Set	C++	CDS [Pug90]
Set	Sets	Ada	NBAda [Gid08]

Tabla 3.1: Estructuras de datos concurrentes con la propiedad del algoritmo *Non-Blocking*.

La Tabla 3.2 contiene los nombres de instituciones o proyectos que crearon la biblioteca y el código abierto para la estructura de datos concurrentes, además incluyen el diseño de mapa.

Nombre	**Última actualización**
ASCYLIB	10-2016
CDS	01-2017
Folly	03-2017
Intel TBB	03-2017
Junction	01-2017
Liblfds	02-2017
Libcuckoo	03-2017
Nbds	05-2009
Oracle	07-2016
TERVEL	01-2016

Tabla 3.2: Estructura de datos concurrente - Mapa

3.1. Investigación de ASCYLIB

(Optimistic Concurrency with OPTIK, 2016) [Gue16]

ASCYLIB es una biblioteca de estructuras de datos concurrente, y tiene más de 40 tipos de implementaciones realizadas: Lista enlazada, Tabla hash, Lista por saltos, Árbol binario, Cola, Cola de prioridades, Pila, entre otros. La mayoría de ellas, tienen la implementación para la versión secuencial, *lock-based* y *lock-free*.

Lo que es más importante de ASCYLIB que se asocia a nuestra tesis, es su diseño de mapa y uso del bloqueo en el paralelismo. Tiene su propio concepto de patrón *OPTIK*, el cual puede observarse en el diagrama de figura 3.1.

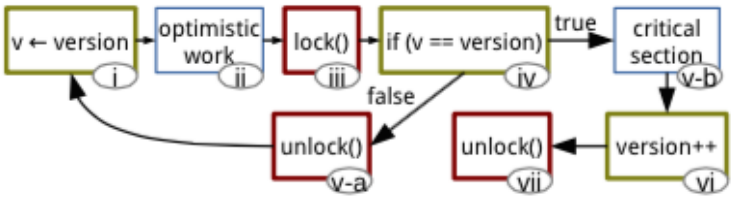

Figura 3.1: El patrón de OPTIK (vista de alto nivel) [Gue16].

OPTIK (*Optimistic concurrency with ticket locks*) es una implementación de bloqueo, la cual utiliza el control de versión para manejar la sección crítica. El número de versión tiene la misma granularidad que un *lock* normal, que es conocido como contador, el que valida los procesos de la operación concurrente de manera progresiva: OPTIK *lock* tiene 8-byte unsigned (*uint64_t* in *C*). Si *lock* es un valor impar, significa que el recurso o la operación está desbloqueada, en cambio, si el valor es un número par, la operación o el recurso compartido estará bloqueado para evitar el uso de los datos inválidos.

Con relación al punto anterior, se debe mencionar que ASCYLIB contiene la estructura de datos de tipo mapa, la cual está asociada al tema principal de la presente tesis y es precisamente en este donde se realizó un análisis del algoritmo, dentro del cual se observó que su implementación del bloqueo cubre una gran región en el código general de la inserción, aunque el patrón del OPTIK (ver figura 3.1) permite realizar un ciclo de intento para ejecutar de nuevo la inserción; es decir, la implementación de ASCYLIB posee una sección crítica de tamaño amplio.

Los experimentos comparamos nuestros resultados con el método *OPTIK global-lock array map* que corresponde a una estructura de dato de mapa concurrente.

Capítulo 4

Desarrollo

4.1. Introducción

En este capítulo, el método que plantea la creación de índices invertidos es desarrollados. ¿Cómo es nuestro diseño del mapa?, ¿Cuál es el objetivo?, ¿Cómo se implementa?, entre otros. También se detalla el paralelismo involucrado en la solución propuesta.

Para tener mayor claridad de nuestro trabajo, se divide el capítulo en 2 secciones: (1) *Secuencial* y (2) *Paralelismo*. La primera explica las lógicas básicas y la programación realizada sobre el diseño de la estructura de datos, el uso de punteros y la implementación de algoritmos de búsqueda. En la segunda se usa OpenMP [Cha01] para implementar los algoritmos multi-hilo. También se detallan el bloqueo y la solución para evitar la inanición, el *deadlock* y la condición de carrera.

4.2. Secuencial

En esta sección se desarrolla dos métodos de manera secuencial: (1) *LinkedListMap* y (2) *BinaryListMap*.

Los diseños de la estructura de mapa en ambos métodos tienen nodo cabecera en cada lista,

a su vez, un indicador del inicio y final de la lista. Por defecto al inicio del programa, este valor es vacío, y el número de *ranking* es 1 (para todas las listas de documentos).

4.2.1. LinkedListMap

En este método, los datos están almacenados en una lista principal previamente enlazada. La lista de documentos se vinculan como sub-lista por cada término, y los nodos de la lista de documentos estan ordenados de acuerdo al número de *ranking*.

4.2.1.1. Diseño de modelo

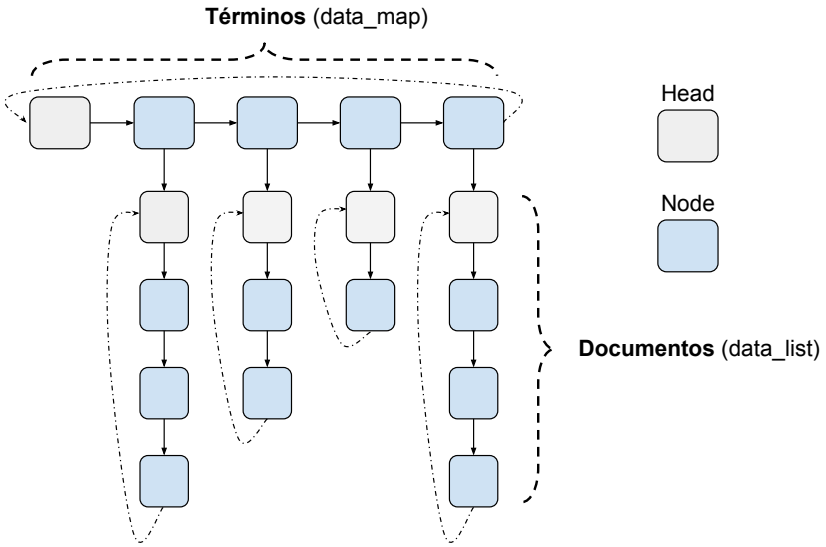

Figura 4.1: El diseño de *LinkedListMap*.

En este modelo se usan 4 estructuras para definir los datos y el cómo se relacionan. Dos de ellos se utilizan para clasificar diferentes tipos de uso (*Términos* y *Documentos*). Además se utilizará una estructura de tipo *union*, que organiza de mejor forma los miembros de este modelo, ya que las estructuras de términos y documentos tienen campos de datos que pueden ser reutilizados entre ellos. Así, con el uso de una estructura de tipo *union*, se puede optimizar y eliminar redundancia en el código general del programa.

A continuación se mostrarán las estructuras utilizada en el algoritmo *LinkedListMap*:

Struct map_t representa la estructura de un mapa. Un campo es *head* (cabeza) de tipo *node_t*, y se define como un nodo normal. Otro tipo es *size*, de tipo *int*, para registrar el tamaño actual del mapa.

```
typedef struct _map{
        node_t head;
        int size;
}map_t;
```

Estructura 4.1: Estructura de mapa *LinkedListMap*

Struct node_t es una estructura de tipo nodo, siendo la parte esencial para generar un mapa o una lista enlazada. Éste cuenta con 3 miembros: *key* (tipo *char**, se utiliza para almacenar los términos o nombres de documentos), *data* (tipo *private_data*, donde depende del tipo de información que se está almacenando y se clasifica por uso del término o el documento), **next* (tipo *node_t*, que en el caso de *LinkedListMap*, *next* sólo se utiliza para asociar el siguiente nodo).

```
typedef struct _node{
        char *key;
        private_data data;
        struct _node *next;
}node_t;
```

Estructura 4.2: Estructura de nodo *LinkedListMap*

Union private_data es un coordinador para separar los 2 tipos de uso de nodo (término y documento).

```
typedef union _private_data {
        struct _data_map data_map;
        struct _data_list data_list ;
} private_data ;
```

Estructura 4.3: Estructura de coordinador *private_data*

Esta unión cuenta con 2 miembros: *data_map* (tipo *_data_map*, se ocupa si el nodo es utilizado para términos) y *data_list* (tipo *_data_list*, se ocupa si el nodo es utilizado para documentos).

Struct data_list es un tipo de estructura de nodo, en el cual se almacena la información del documento para un término determinado.

```
typedef struct _data_list {
        float ranking;
        int frequency;
} _data_list ;
```

Estructura 4.4: Estructura de documento *data_list*

Consiste con *ranking* (*float*, el cual depende del tipo de algoritmo para almacenar el resultado del *ranking*), y *frequency* (*int*, indicando la cantidad de veces que aparece un término en algún nodo de documentos).

Struct data_map es un tipo de estructura de nodo para el uso de términos. Se utiliza en la lista principal del mapa; no es como el nodo de documento.

```
typedef struct _data_map{
        struct _node *value;
        int size ;
}_data_map;
```

Estructura 4.5: Estructura de término *data_map*

Está compuesto por **value* (*node_t*, este nodo es la cabeza de la lista de documentos para el mismo término) y *size* (*int*, es el tamaño actual de la lista de documento generada en cada instante).

4.2.1.2. Función predefinida - Mapa de Lista Simple Enlazada

Inicializar el mapa

La función *Inicializar_Mapa* es el primer paso que se debe realizar para crear un nuevo mapa. La idea de esta función es aprovechar el recurso total de la máquina, pudiéndose iniciar y usar varios mapas en un solo programa.

Insertar un nodo

La función *Insertar_Mapa* hace la inserción de un nodo en un mapa determinado. Tal función ya tiene considerado resolver los casos del término que ya existían en el mapa o en la lista de documentos y en cuál posición de ésta debe ubicarse el término.

Algoritmo 2 *INLLM*: Insertar un nodo de término o documento en LinkedListMap.

Insertar_Mapa(Mapa *M*, Término *T*, Documento *D*, Ranking *R*, Frecuencia *F*)

1: $Nodo\_Repetido \leftarrow \emptyset$
2: $Posición \leftarrow \emptyset$
3: $Repetido \leftarrow RevisiónNodo\_BúsquedaPosición\_Término(M, T, \&Nodo\_Repetido, \&Posición)$
4: **if** $(Repetido == TRUE)$ **then**
5: *//Nodo_Repetido es un nodo del mapa que tenga el mismo término lo que va a insertar.*
6: $Repetido \leftarrow RevisiónNodo\_BúsquedaPosición\_Documento(Nodo\_Repetido, D, R, \&Posición)$
7: **if** $(Repetido == FALSE)$ **then**
8: $Insertar\_Documento(\&Nodo\_Repetido, D, R, F, Posición)$
9: **end if**
10: **else**
11: $Insertar\_Término(\&M, T, D, R, F, Posición)$
12: **end if**

Para insertar un nuevo nodo en un mapa, se debe verificar si este proceso ya se había llevado a cabo en otra ocasión, después realizar la búsqueda de posición adecuada para este nuevo nodo en el mapa. Si los dos trabajos se hacen por separados, el cómputo total del programa será muy costoso en cuanto al tiempo necesario por recorrer todo el mapa. Aprovechando las características de este mapa, los nodos insertados estarán ordenados de forma lexicográfica. Así en la búsqueda de la posición de un nuevo nodo y la verificación del término repetido se pueden utilizar el mismo ciclo de la búsqueda para realizar los dos trabajos a la vez (línea 3 en algoritmo 2 INLLM).

Algoritmo 3 *RNBPT*: La revisión de nodo repetido y la búsqueda de posición para el término.

RevisiónNodo_BúsquedaPosición_Término
(Mapa *M*, Término *T*, Nodo_Repetido **N_R*, Posición **P*)

```
 1: Repetido ← FALSE
 2: *Nodo = M → Cabeza
 3: while (Comparar_Término(Nodo → Siguiente → Valor, T) <= 0) do
 4:     if (Es_Mismo_Término(Nodo → Siguiente → Valor, T)) then
 5:         *N_R ← (Nodo → Siguiente)
 6:         Repetido ← TRUE
 7:         Break
 8:     end if
 9:     if (Es_Mismo_Término(Nodo → Siguiente → Valor, ∅)) then
10:         Break
11:     end if
12:     Nodo = Nodo → Siguiente
13: end while
14: *P = Nodo
15:
16: return Repetido
```

El algoritmo 3 RNBPT muestra el proceso de revisión del término repetido y la búsqueda de posición adecuada. La línea 3 implica que si el término del nuevo nodo es mayor que el término del siguiente nodo, se sigue el ciclo, hasta encontrar la posición adecuada para el nuevo nodo.

Principalmente la función utiliza dos condiciones para obtener el resultado. La primera condición (línea 4 en el algoritmo 3 RNBPT) está definida para verificar si el término del siguiente nodo es el mismo que el que va a insertar. La variable *Repetido* tiene valor *FALSE* como está predefinido. En el caso de aprobar la condición, *Repetido* va a ser de valor *TRUE*, o sea la clave es repetida, ya que el término fue insertado en el mapa anteriormente. Por último, un uso de *break* se lleva a cabo en el ciclo de búsqueda por si fuese un término repetido. La segunda condición (línea 8 en el algoritmo 3 RNBPT) utiliza el estado de la condición por el diseño de nuestro mapa, el inicio de la lista principal es un nodo vacío y el último nodo tiene un punto de salto que conecta con el punto de partida, formando una especie de *anillo*. Si se quiere impedir que una situación de búsqueda que resulte infinita, se utiliza esta condición para salir del ciclo, así se establece que si el ciclo sale de este estado, el nodo va a estar en la última posición en la cola de la lista.

Después de la búsqueda, la función se dividirá en 2 secciones diferentes por la verificación de la clave repetida. La variable *Repetido* (línea 3 en algoritmo 2 INLLM) va a dirigir el camino del código junto con el resultado de la búsqueda. Si no se comprueba la primera condición (línea 4 en algoritmo 2 INLLM), se entra a la sección de la inserción de un nuevo nodo, en la cual se inicializan variables nuevas y se pasan los datos de inicialización de parámetros a las variables de este nodo. La parte del enlace de los punteros, se obtiene en el resultado anterior de la búsqueda con la variable *Posición* (la variable había almacenado un nodo, que pertenece a la posición inferior respecto al nuevo nodo), así simplemente se puede realizar la conexión de punteros (ver figura 4.2):

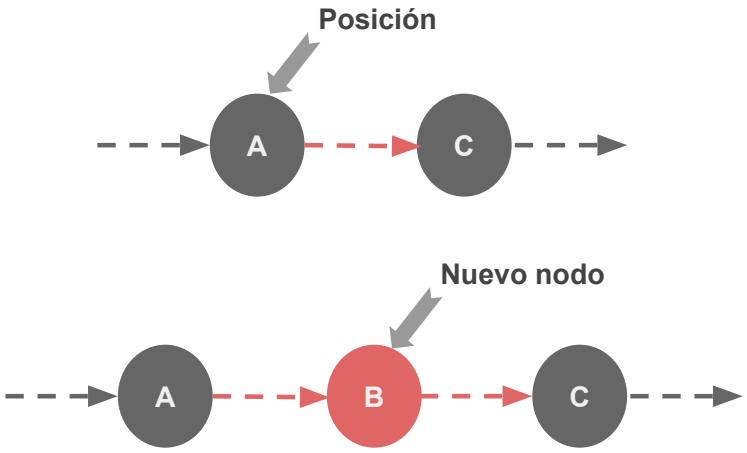

Figura 4.2: La inserción de un nodo nuevo.

La sección de clave repetida (línea 4 en algoritmo 2 INLLM) requiere realizar otra fase de verificación, porque cada lista de documento es dependiente de un término, y el contenido de la lista no debe tener una identificación de un documento repetido.

Para verificar si esta identificación de documento ya había aparecido en la lista, se utiliza el mismo concepto del algoritmo 3 RNBPT. El punto más importante en esta lista es el orden de los nodos, vale decir, la lista de términos ordenados por lexicografía. Sin embargo, la lista de documento es ordenada de acuerdo al el número de ranking. Por lo tanto, no puede salir del ciclo de verificación hasta que el último nodo esté revisado o se encuentre repetido con antelación.

4.2.2. BinaryListMap

BinaryListMap está compuesto por un mapa con la implementación de la búsqueda binaria, y utiliza la propiedad de la lista doblemente enlazada como base de la estructura para vincular los nodos.

4.2.2.1. Diseño del modelo

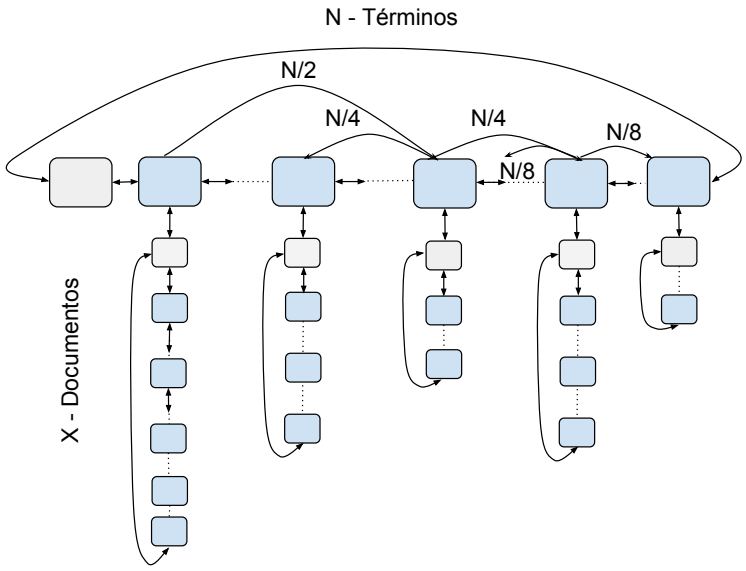

Figura 4.3: El diseño de BinaryListMap.

La idea es usar los punteros para apuntar a la posición donde reubicar, entonces la cantidad de punteros por cada nodo va a ser 4, en este caso. Y para realizar una búsqueda binaria, es necesario realizar la actualización de los saltos de los punteros.

Dicha actualización es un trabajo costoso, ya que es necesario realizar el conteo de avances para llegar a la posición que corresponde a un puntero, en la forma secuencial. El mapa del presente diseño no es como un arreglo que puede saltar directamente al índice requerido por medio del cálculo *"N/2"* (ver figura 4.3).

El modelo de la estructura de datos que se usa, es el mismo que *LinkedListMap*, utilizando también 4 estructuras para definir los campos de los datos. La diferencia entre los 2 modelos de diseño es la cantidad de punteros de saltos por nodo, donde *BinaryListMap* usa cuatro

punteros de salto (*struct _node *next[4]*). La siguiente estructura corresponde a un nodo de *BinaryListMap*. Note la diferencia con el nodo de *LinkedListMap*(Estructura 4.2):

```
typedef  struct  _node{
            char  *key;
             private_data  data;
             struct  _node *next[4];
}node_t;
```

Estructura 4.6: Estructura de nodo *BinaryListMap*

Dos punteros son para nodos laterales de ambos lados (la dirección de superior e inferior), y otros dos son para un salto mayor, e igualmente tienen dirección de superior e inferior (ver figura 4.4).

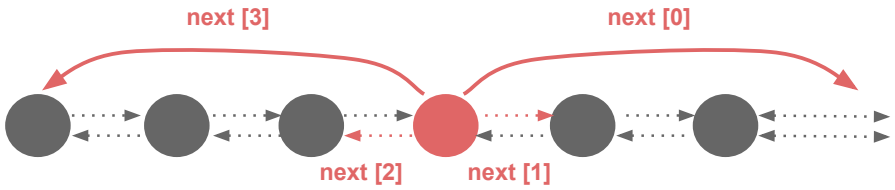

Figura 4.4: Los punteros de saltos en *BinaryListMap*.

La otra diferencia del modelo es en *struct map_t* (entre Estructura 4.1 y 4.7), éste tiene un nuevo campo de datos tipo *int umbral*. Se usa para definir cuándo se realiza la actualización de punteros para los saltos mayores (*next[0]* y *next[3]*). Si no se realiza la actualización, los punteros no van a servir, como un campo de redundancia.

```
typedef  struct  _map{
            node_t  head;
             int  size;
             int  umbral;
}map_t;
```

Estructura 4.7: Estructura de mapa *BinaryListMap*

La condición para la actualización de puntero es dependiente del resultado de la operación módulo entre el tamaño de lista actual y un umbral de salto que es indicado por el usuario, es decir, la función se efectúa cuando se cumple la iteración de *umbral* (línea 12 en algoritmo 4 INBLM), actualizándose los punteros de saltos. Dado que dicha actualización es realizada por la condición del tamaño de la lista actual, ésta será la lista principal, y las sub-listas estarán implementadas como el diseño de *LinkedListMap*.

Hasta este punto, aún se mantiene una dificultad a resolver ¿Qué momento es el óptimo para realizar la actualización de los punteros de saltos? Para responder esta pregunta hay que hacer experimentos con diferentes tipos de entrada ordenada por lexicografía, o al revés a través de un orden arbitrario, ya que esto puede afectar el rendimiento total del programa. El factor más importante es la cantidad total de elementos que se insertarán, porque la actualización de punteros se realiza cuando se llega a un cierto número de términos insertados. Lamentablemente como no se puede saber el número exacto de los nodos que se agregarán en el mapa, es posible que los datos de entrada puedan estar repetidos en otro documento. Idealmente el umbral de actualización es un valor que no sobrepasa la mitad de la cantidad total de nodos que se insertará. Si se usa un umbral con un valor muy elevado, el rendimiento de la búsqueda binaria no será efectivo.

4.2.2.2. Función predefinida - Mapa de Búsqueda Binaria

Inicializar el mapa

A diferencia de la presente función en la versión *BinaryListMap*, aquí se inicializa los 4 punteros de salto, y se asigna el valor para el campo *umbral* (estructura 4.7).

Insertar un nodo

El esquema general de la función *Insertar_Mapa* del algoritmo 4 INBLM es similar al algoritmo 2 INLLM, pero para tener el efecto de la búsqueda binaria, el método principal de la búsqueda (línea 3 en algoritmo 4 INBLM) requiere de un cambio considerable.

Algoritmo 4 *INBLM*: Insertar un nodo de término o documento en BinaryListMap.

Insertar_Mapa(Mapa *M*, Término *T*, Documento *D*, Ranking *R*, Frecuencia *F*)

```
 1: Nodo_Repetido ← ∅
 2: Posición ← ∅
 3: Repetido ← RevisiónNodo_BúsquedaPosición_Término(M, T, &Nodo_Repetido, &Posición)
 4: if (Repetido == TRUE) then
 5:     //Nodo_Repetido es un nodo del mapa que tenga el mismo término lo que va a insertar.
 6:     Repetido ← RevisiónNodo_BúsquedaPosición_Documento(Nodo_Repetido, D, R, &Posición)
 7:     if (Repetido == FALSE) then
 8:         Insertar_Documento(&Nodo_Repetido, D, R, F, Posición)
 9:     end if
10: else
11:     Insertar_Término(&M, T, D, R, F, Posición)
12:     if (M → Tamaño%M → Umbral == 0) then
13:         Simplificar_Punteros(M → Cabeza.Siguiente[1], M → Tamaño)
14:         Actualizar_Punteros(M → Cabeza.Siguiente[1], NULL, M → Tamaño, −1)
15:     end if
16: end if
```

Si revisa el código de la versión *LinkedListMap* (algoritmo 3 RNBPT), la búsqueda siempre hace un previo chequeo del término en el siguiente nodo, para hacer la comparación por lexicografía y el avance del paso por nodo. Ahora, en la versión *BinaryListMap* (algoritmo 5 RNBPT), la comparación de término que se necesita realizar está en la posición actual. En algunas ocasiones, necesitará hacer el chequeo de los nodos cercanos para evitar el caso de *loop* infinito. Se explicará el método de la búsqueda por cada condición *if* en el código (línea 4 - 22 en el algoritmo 5 RNBPT).

En el caso de aprobar la verificación de clave repetida, se empieza a ver las condiciones de la búsqueda de posición adecuada para el nuevo nodo, lo que implica un primer *if* con dos condiciones (línea 8 en el algoritmo 5 RNBPT). La primera condición de este *if* es verificar si el nuevo término tiene un orden mayor de lexicografía y la segunda es para saber si el siguiente nodo tiene un término vacío (apunta al nodo cabecera). Si cumple las 2 condiciones, puede ser que el mapa este vacío, así se podrá insertar el nuevo nodo y salir del *while* directamente. Si no cumple, significa que este nodo tiene mayor orden que los ya insertados.

La segunda condición *if* (línea 11 en el algoritmo 5 RNBPT) verifica los términos más cercanos, donde el nodo predecesor debe tener un mayor orden lexicográfico y el nodo de sucesor debe tener un menor orden lexicográfico. Si el nuevo término no cumple las 2

condiciones de arriba, seguirá el ciclo de salto por la búsqueda binaria.

Algoritmo 5 *RNBPT*: La revisión del nodo repetido y la búsqueda de posición para el término.

RevisiónNodo_BúsquedaPosición_Término
(Mapa *M*, Término *T*, Nodo_Repetido **N_R*, Posición **P*)

```
 1: Repetido ← FALSE
 2: *Nodo = M → Cabeza
 3: while (TRUE) do
 4:     if (Es_Mismo_Término(Nodo → Valor, T)) then
 5:         Repetido ← TRUE
 6:         Break
 7:     end if
 8:     if (Es_Menor_Léxicográfico(Nodo → Valor, T)
        &&Es_Mismo_Término(Nodo → Siguiente[1] → Valor, ∅)) then
 9:         Break
10:     end if
11:     if (Es_Menor_Léxicográfico(Nodo → Valor, T)
        &&Es_Mayor_Léxicográfico(Nodo → Siguiente[1] → Valor, T) then
12:         Break
13:     end if
14:     if (Es_MenorIgual_Léxicográfico(Nodo → Valor, T)
        &&!Es_Mismo_Término(Nodo → Siguiente[0] → Valor, ∅)) then
15:         Node = Node → Siguiente[0]
16:     else if (Es_MayorIgual_Léxicográfico(Nodo → Valor, T)
        &&!Es_Mismo_Término(Nodo → Siguiente[3] → Valor, ∅)) then
17:         Node = Node → Siguiente[3]
18:     else if (Es_MenorIgual_Léxicográfico(Nodo → Valor, T) then
19:         Node = Node → Siguiente[1]
20:     else
21:         Node = Node → Siguiente[2]
22:     end if
23: end while
24: *P = Nodo
25:
26: return Repetido
```

Según la regla de búsqueda binaria, los saltos mayores tienen prioridad al realizar el chequeo. La primera condición *if* (línea 14 en el algoritmo 5 RNBPT) está comparando si el nuevo término tiene un mayor léxico que el nodo actual, además que el destino de salto superior no sea en el valor vacío. Luego, *else if* es el caso contrario, el nuevo término tiene un menor léxico que el nodo actual, pero el destino de salto inferior no debe ser vacío. La última condición de *if* se ocupa cuando se acerca al último nodo del mapa, por lo tanto, se avanza con saltos menores.

En el caso de la inserción de la sub-lista (lista de documento), no se implementa la búsqueda binaria, ya que la sub-lista está ordenada por el número de ranking. Si quiere verificar que un documento es repetido o no, se debe recorrer todos los nodos en la lista, uno por uno. Por

lo tanto, será mejor utilizar búsqueda binaria en la lista principal para que un nuevo término busque su posición adecuada.

Restablecer los punteros

En este diseño se necesita una función para simplificar los punteros de saltos mayores que apuntan a los nodos cercanos, ya que los punteros de salto mayor pueden provocar un loop de la búsqueda infinita. Si no realiza la simplificación de los punteros de salto mayor y se los actualizan directamente, es muy probable generar el siguiente problema:

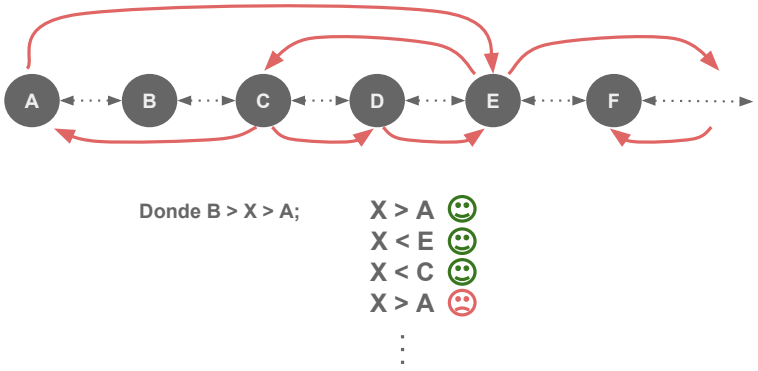

Figura 4.5: Problema que implica un bucle infinito.

La condición para salir el ciclo de búsqueda, nos impide detectar el nodo B, puesto que siempre la prioridad de los saltos mayores tiene mayor preferencia que los saltos cercanos. Entonces si quiere mantener la característica de los punteros de la búsqueda binaria y que no presente el problema mencionado con anterioridad, se tiene que efectuar la función de simplificación antes que la actualización de punteros esté realizada.

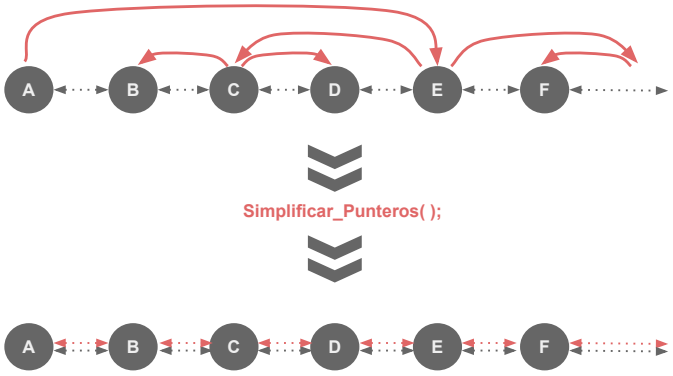

Figura 4.6: La función *Simplificar_Punteros()*.

Como se muestra en el diagrama de arriba, la función *Simplificar_Puntero* anula los punteros de salto mayor, y deja cada enlace conectado con los nodos cercanos. El trabajo de la función es muy costoso, ya que no se sabe exactamente, cuáles son los nodos que tienen asignados los saltos mayores; por lo anterior, se tiene que realizar una búsqueda de chequeo y modificar el puntero de nodo uno por uno hasta el final.

Actualizar los punteros

En las secciones anteriores, se hablado del funcionamiento y la ventaja de búsqueda binaria. Ahora, se explicará cómo se actualizan los punteros de saltos mayores desde la lógica fundamental de la implementación.

Para realizar la búsqueda binaria con los punteros, el primer paso es calcular la distancia entre el punto inicial y los puntos más lejanos del salto, y el cómo calcular la distancia entre los nodos. La figura 4.7 muestra que si se avanzan más puntos asociados hacia el nivel más profundo, las distancias de cada nivel, entre padre e hijos, serán *1/2* de diferencia, y van disminuyendo hacia el nivel más profundo. El tema de dividir con un número impar tiene un margen de error, ya que se está usando la forma de redondeo para obtener la distancia; la condición de búsqueda para los hijos es *N/2 >= 1*, significa que la distancia debe ser mayor o igual al 1.

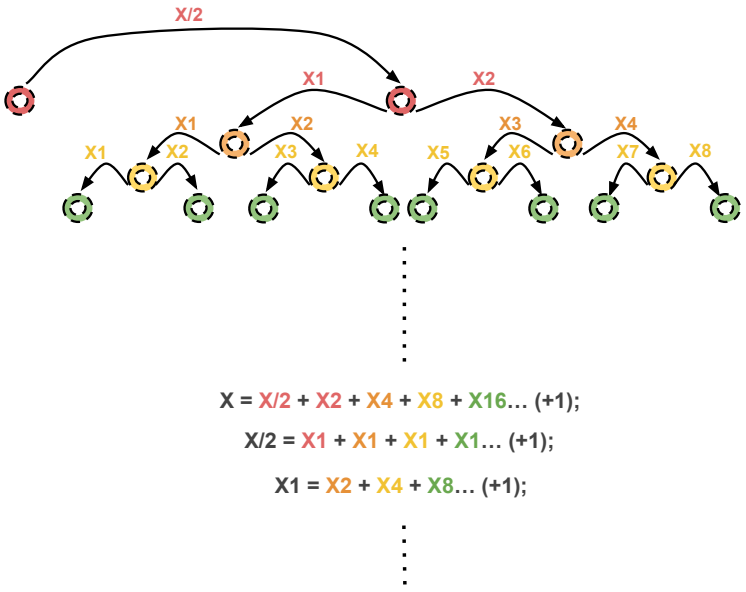

Figura 4.7: El proceso de la función recursiva *Actualizar_Punteros()*.

Para minimizar el error de la división, se usa la resta como se muestra en el dibujo anterior. *X* es el tamaño actual del mapa y se usa *X/2* para sacar el primer salto superior. Pero en los saltos posteriores hay que considerar el número de pasos que había avanzado. Así, los resultados de saltos deben cumplir la condición que se mostró en la imagen *X = X/2 + X2 + X4... (+1);*, en donde el primer nodo del mapa se conoce como un avance del salto, entonces tiene un error *+1*.

El diseño de *BinaryListMap* tiene 4 punteros de saltos, donde dos de ellos son para el salto mayor. Para realizar la actualización de los saltos mayores, se tiene una función recursiva *Actualizar_Punteros* (algoritmo 6 AP), la cual necesita 4 parámetros principales que son *Nodo* (nodo que se requiere a actualizar), *Nodo_Previo* (nodo auxiliar para sacar el salto inferior), *N_SaltosSuperior* (número de salto superior para tal nodo actual) y *N_SaltosInferior* (número de salto inferior para el nodo actual).

Algoritmo 6 *AP*: Actualizar los saltos de punteros para la búsqueda binaria.

Actualizar_Punteros
(Nodo **N*, Nodo_Previo **N_P*, N_SaltosSuperior *N_SS*, N_SaltosInferior *N_SI*)

1: **if** $(N\_SS/2 > 0)$ **then**
2: $\quad {}^{*}Nodo\_Adelantar = N$
3: $\quad {}^{*}Nodo\_Retroceder = N$
4: $\quad$ **for** $i = 0, i < N\_SS/2, i++$ **do**
5: $\quad\quad Nodo_A delantar = Nodo_A delantar \rightarrow Siguiente[1]$
6: $\quad$ **end for**
7: $\quad N \rightarrow Siguiente[0] = Nodo_A delantar$
8: $\quad Actualizar\_Punteros(Nodo\_Adelantar, N, N\_SS - N\_SS/2, N\_SS/2)$
9: $\quad$ **if** $(N\_P! = NULL \&\& N\_SI/2 > 0)$ **then**
10: $\quad\quad$ **for** $i = 0, i < N\_SI/2, i++$ **do**
11: $\quad\quad\quad Nodo\_Retroceder = Nodo_R etroceder \rightarrow Siguiente[2]$
12: $\quad\quad$ **end for**
13: $\quad\quad N \rightarrow Siguiente[3] = Nodo\_Retroceder$
14: $\quad\quad Actualizar\_Punteros(Nodo\_Retroceder, N, N\_SI/2, N\_SI - N\_SI/2)$
15: $\quad$ **end if**
16: **end if**

Si se quiere efectuar la actualización de punteros, donde el parámetro *Nodo* será el primer nodo del mapa, *Nodo_Previo* tendrá un valor inicial *NULL*, por el primer nodo que no se relaciona con ningún nodo predecesor aún. Otro parámetro importante *N_SaltosSuperior* tendrá el valor del tamaño actual del mapa para empezar. *N_SaltosInferior* asignará un valor *-1* inicialmente, ya que el primer nodo no tiene su nodo predecesor.

Las variables *Nodo_Adelantar* y *Nodo_Retroceder* son nodos de conteo para obtener los saltos mayores. La variable *Nodo_Adelantar* es utilizada para obtener el nodo de salto superior, por otro lado, *Nodo_Retroceder* es encargado de capturar el nodo de salto inferior. Se puede apreciar en la función recursiva anterior, que el salto superior presenta una mayor prioridad con respecto al salto inferior dado el diseño del algoritmo. La otra dificultad es el cálculo para los saltos inferiores, se utilizar *N_SaltosInferior*, el resultado del predecesor (*N/4*) y el número de paso reciente (*N/2*) en la operación matemática (la división o la resta). Por ejemplo la siguiente figura 4.8:

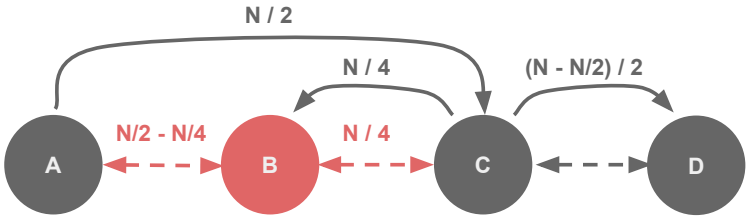

Figura 4.8: Un ejemplo de la actualización de los punteros de saltos.

El cálculo entre nodo B y C es simple, ya que con la división *N_SaltosInferior/2* se obtiene el valor. ¿Pero qué pasaría si el nodo B también tiene un salto inferior ahora?, tal como se aprecia en la siguiente figura 4.9:

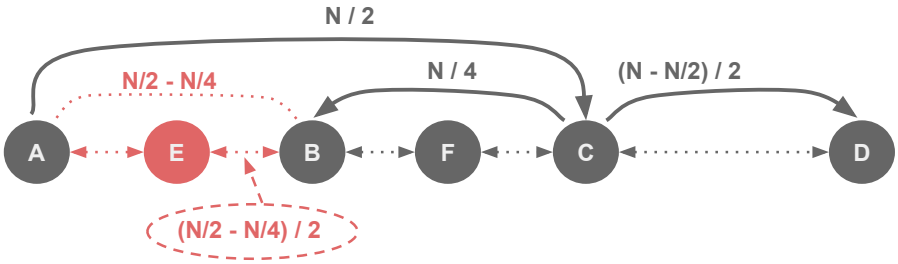

Figura 4.9: Extensión del ejemplo anterior.

Para resolver este caso, no se puede realizar la división *N_SaltosInferior/2* directamente. Según la lógica del algoritmo, se debe hacer el cálculo de la resta primero *N/2 - N/4* (*N_SaltosInferior - N_SaltosInferior/2*) y luego se implementa la división *(N/2 - N/4)/2*.

4.3. Paralelismo

Este capítulo se muestran las problemáticas y las soluciones desarrolladas en el área de Computación Paralela. Las secciones del capítulo serán similar al capítulo anterior: *LinkedListMap* y *BinaryListMap*.

Se usará a *OpenMP* [Cha01] para implementar el manejo de los hilos. Antes todo, el primer paso a resolver, es la entrada de datos: ¿en qué forma se podría transferir datos a nuestro mapa multi-hilo? Los datos deben estar distribuidos en la forma correcta, para que cada hilo pueda trabajar sin retrasar o afectar a los demás procesos. Es posible que pueda ocurrir que exista más de un hilo leyendo la misma entrada. Esto implicaría que la misma operación se ejecutaría más de una vez por hilos distintos.

4.3.1. Distribución de Datos

Los datos primarios del archivo se guardarán en la variable del arreglo *Entradas[i]* (algoritmo 7 DE) directamente. Se asigna *Entradas* como el tipo privado (*firstprivate*) en la sección de multi-hilos. *firstprivate* indica que cada hilo tendrá su propia variable con el valor que había definido inicialmente. Si un hilo está realizando una modificación con la misma variable, otros hilos no recibirán ningún cambio sobre esta acción. Pero la variable del tipo compartido es diferente, si realiza un cambio en el valor desde cualquier hilo, provoca que los demás hilos tendrán error por el dato obsoleto o la información incompleta.

Num_Nodos_Insertados (algoritmo 7 DE) es un tipo de variable auxiliar para el conteo de los nodos insertados. Cada hilo tendrá esta misma variable como su propio elemento (*Num_Nodos_Insertados[omp_get_thread_num()]*), y la suma total de ellas corresponde al tamaño actual del mapa.

En el tema de la medición del rendimiento, es importante la escalabilidad entre el número de hilos y la cantidad de datos. Por la afinidad de la CPU, se utiliza la biblioteca *sched.h* que permite asignar de manera exclusiva la ejecución de un hilo en un único núcleo.

Algoritmo 7 *DE*: La distribución de las entradas.

```
#pragma omp parallel
firstprivate(Entradas) shared(Mapa, Num_Nodos_Insertados){
 1: for i = omp_get_thread_num(), i < Num_Entradas, i+ = omp_get_num_threads() do
 2:     Insertar_Mapa(Mapa,
                      Entradas[i].Término,
                      Entradas[i].Documento,
                      Entradas[i].Ranking,
                      Entradas[i].Frecuencia,
                      Num_Nodos_Insertados)
 3: end for
}
```

Para evitar que los hilos no lean la misma entrada repetida, se hace la distribución de datos por cada múltiplo del índice que corresponde. Los hilos tienen su propio número de identidad, éste se obtiene de la función de *OpenMP omp_get_thread_num()*. Se utiliza dicha identidad como el índice inicial de los primeros datos, los restantes serán asignados por la suma del múltiplo *omp_get_num_threads()* (la cantidad de hilo) por cada iteración del ciclo.

4.3.2. LinkedListMap

Para mostrar el método de la ditribución de datos por cada hilo, se presenta el esquema 4.10.

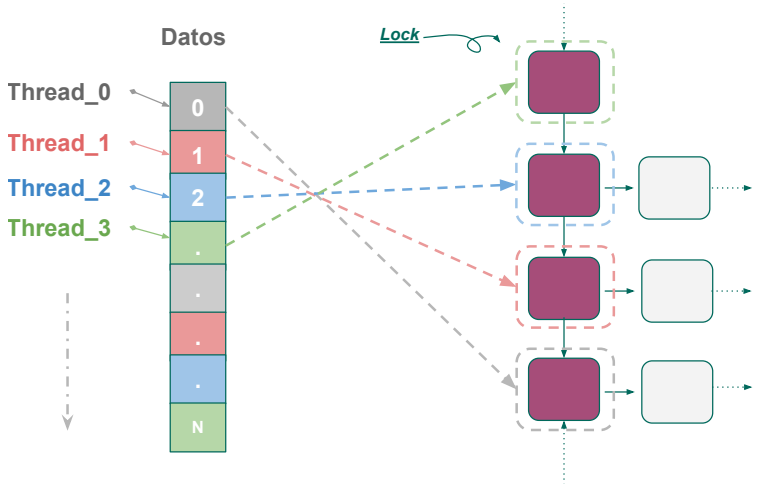

Figura 4.10: El esquema de la distribución de datos por cada hilo (*LinkedListMap lock-based*).

4.3.2.1. Diseño de modelo

La diferencia del diseño entre la versión secuencial (la sección 4.2.1.1.) y concurrente (la sección 4.3.2.1) es la presencia de una nueva variable en la estructura *node_t*. Dicha variable es un tipo de *omp_lock_t*, en cual se puede controlar el estado de bloqueo del mismo nodo. El uso del bloqueo es dependiente de la disponibilidad de ello, siempre un recurso compartido permite a un solo hilo tener acceso a él. Los demás hilos esperan hasta el bloqueo se libera.

```
typedef struct _node{
            omp_lock_t lock;
            char *key;
            private_data data;
            struct _node *next;
}node_t;
```

Estructura 4.8: Estructura de nodo *LinkedListMap lock-based*

4.3.2.2. Función predefinida - Mapa de Lista Simple Enlazada Concurrente

Inicializar el mapa

Por la nueva agregación de *omp_lock_t*, la presente función se inicia el bloqueo (*omp_init_lock*) para el nodo inicial.

Insertar un nodo

El nuevo cambio de la función son la entrada *Num_Nodos_Insertados* (una variable de conteo) y el método de la búsqueda. Para adaptar el uso de *lock*, se necesita unos cambios del algoritmo, pero la lógica principal no cambia. La demostración de los procesos *Insertar_Mapa*, tiene el siguiente diagrama que se divide en varios sectores:

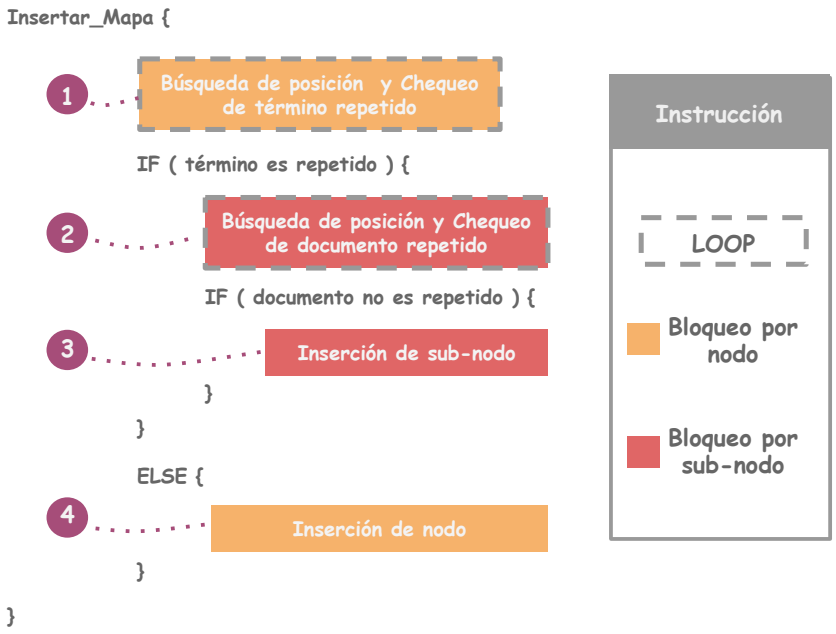

Figura 4.11: El diagrama de procesos de la función *Insertar_Mapa* (*LinkedListMap lock-based*).

En la primera sección del diagrama, se realiza una serie de búsquedas para el nuevo término. Dentro del ciclo se debe encontrar una posición adecuada y revisar si el término ya existió en el mapa. Con respecto a la revisión del nuevo término, es necesario realizar el bloqueo previamente, ya que si los otros subprocesos están tratando de ejecutar la acción "*Escribir*", obtendrá un resultado inválido porque el valor no completa su actualización o es un dato obsoleto.

Algoritmo 8 *RNBPTCLM*: Revisión del nodo repetido y Búsqueda de posición para el término.

RevisiónNodo_BúsquedaPosición_Término
(Mapa *M*, Término *T*, Nodo_Repetido **N_R*, Posición **P*)

```
 1: Repetido ← FALSE
 2: *Nodo = M → Cabeza
 3: while (TRUE) do
 4:     Nodo_Bloqueado = Nodo
 5:     while(!omp_test_lock(&Nodo_Bloqueado → lock))
 6:     if (Es_MenorIgual_Léxicográfico(Nodo → Siguiente → Valor, T)) then
 7:         if (Es_Mismo_Término(Nodo → Siguiente → Valor, T)) then
 8:             *N_R ← (Nodo → Siguiente)
 9:             Repetido ← TRUE
10:             Break
11:         end if
12:         if (Es_Mismo_Término(Nodo → Siguiente → Valor, ∅)) then
13:             Break
14:         end if
15:         Nodo = Nodo → Siguiente
16:     else
17:         Break
18:     end if
19:     omp_unset_lock(&Nodo_Bloqueado → lock)
20: end while
21: *P = Nodo
22:
23: return Repetido
```

Después de la finalización de la búsqueda, el hilo sostendrá su nodo encontrado con el estado de bloqueo, lo cual fue asignado por él. Así, los otros subprocesos no podrán acceder al mismo recurso compartido, ni avanzarán a los nodos después de él. Si los nodos del predecesor están en estado libre, el subproceso puede seguir su trabajo, si no, hay que esperar hasta que el sostenedor libera su bloqueo.

Ahora bien, si un hilo ya ha obtenido su resultado desde la búsqueda y además el caso no es un término repetido, se inserta el nuevo nodo al mapa y se libera el *lock* asociado después del proceso (sección 4). La variable de conteo *Num_Nodos_Insertados* nos ayuda a contar los términos insertados y sin usar el bloqueo, ya que cada subproceso tiene su propia variable auxiliar. En cambio, si usa una sola variable compartida y realiza el conteo con ella, se necesita el bloqueo, la sección crítica y la operación atómica para evitar un valor inválido por la condición de carrera.

```
Num_Nodos_Insertados[omp_get_thread_num()]++;
```

Después que los hilos terminan su inserción, se suma el valor del arreglo y se almacena en la variable compartida. Esta manera nos ahorra el tiempo de espera por cada vez que la inserción es realizada.

```
for(i=0;i<num_threads;i++)
  Mapa->Tamaño+=Num_Nodos_Insertados[i];
```

Si el término es repetido, el hilo liberará el bloqueo del nodo asociado y pedirá otro nuevo lock por la búsqueda de sub-lista (sección 2). El caso de la sublista es parecido como la primera sección, mientras se está chequeando el documento, si no está repetido, se bloquean los nodos de la comparación uno por uno, y se libera después del uso. Si justamente se encuentra una posición adecuada durante el camino de la búsqueda, no se libera tal bloqueo hasta finaliza el proceso completo.

Un punto importante de la sub-lista es que el orden de los nodos depende del número ranking, no es como la lista principal que usa el orden de la lexicografía. Esto significa que ante la necesaria revisión de documento repetido, se recorre la sub-lista completa, no teniendo otra excepción. Aquí se tiene un ejemplo para explicar el problema mencionado arriba (figura 4.12):

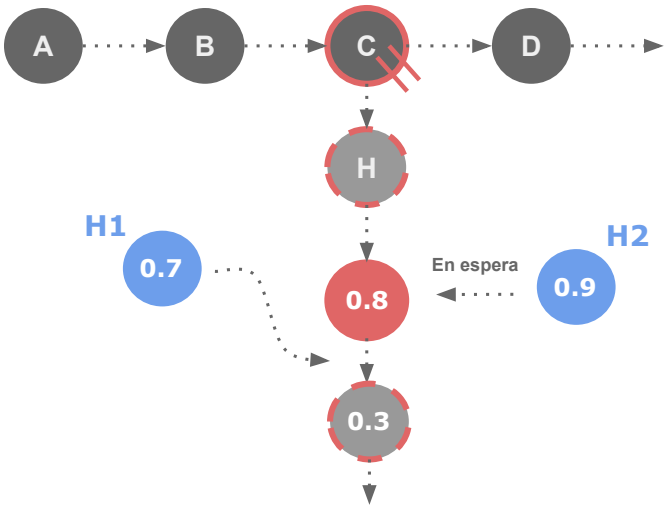

Figura 4.12: Un ejemplo del problema simple sobre la inserción de un nuevo nodo en la lista de documento.

Suponga que los subprocesos *H1* y *H2* están intentando a insertar el mismo documento en la misma sub-lista. *H1* encontró su posición adecuada y ganó la prioridad de obtener el bloqueo del

nodo (*ranking 0.8* en figura 4.12), no liberará el nodo rojo hasta terminar la inserción completa. Puesto que si con *ranking 0.8* tiene disponible el acceso antes, la revisión del subproceso *H2* podría ser más rápida que la inserción de *H1*. Aquí el problema será que el *H2* no alcanza a detectar un documento repetido el cual ya está insertado por el otro subproceso *H1*.

Para finalizar la búsqueda, generará un bloqueo en el comienzo de la sub-lista, como una ficha de inserción para evitar el problema mencionado de la arriba. Por lo tanto, una inserción de documento podría tener uno o dos bloqueos y se liberan juntos en el final del proceso. Aunque la inserción de la sub-lista no se puede realizar por multi-subprocesos simultáneamente, la búsqueda de la posición y verificación de documento se avanzan en forma paralela.

Algoritmo 9 *RNBPD*: La revisión del nodo repetido y búsqueda de posición para el documento.

RevisiónNodo_BúsquedaPosición_Documento
(Nodo_Repetido *N_R*, Documento *D*, Ranking *R*, Posición **P*)

```
 1: Repetido ← FALSE
 2: *Nodo = N_R
 3: *Sub_Nodo = Nodo → Documento.Cabeza
 4: *Ranking_Nodo = Sub_Nodo
 5: Posición_Bloqueada = 0
 6: while (TRUE) do
 7:     Nodo_Bloqueado = Sub_Nodo
 8:     while(!omp_test_lock(&Nodo_Bloqueado → lock))
 9:     if (Sub_Nodo → Ranking >= R
        &&(R >= Sub_Nodo → Siguiente → Ranking
        ||Sub_Nodo → Siguiente → Ranking == 1)) then
10:         Ranking_Nodo = Sub_Nodo
11:         Posición_Bloqueada = 1
12:     end if
13:     if (Es_Mismo_Documento(Sub_Nodo → Valor, D)) then
14:         Repetido ← TRUE
15:         Break
16:     end if
17:     if (Es_Mismo_Documento(Sub_Nodo → Siguiente → Valor, ∅)) then
18:         Break
19:     end if
20:     Sub_Nodo = Sub_Nodo → Siguiente
21:     if (Posición_Bloqueada! = 1) then
22:         omp_unset_lock(&Nodo_Bloqueado → lock)
23:     end if
24:     if (Posición_Bloqueada == 1) then
25:         Posición_Bloqueada = −1
26:     end if
27: end while
28:
29: return Repetido
```

La variable auxiliar *Posición_Bloqueada* (línea 5 en el algoritmo 9 RNBPD) nos facilita sostener el bloqueo del nodo predecesor para la posición adecuada del documento que se insertará. El valor por defecto de *Posición_Bloqueada* es 0, significa que no se encuentra una posición adecuada aún, o sino el valor *Posición_Bloqueada* cambiará en 1. Se usa la expresión *Posición_Bloqueada = -1* (línea 25 en el algoritmo 9 RNBPD) para mantener dicho bloqueo, y luego en el resto de la búsqueda se puede liberar *lock* del otro nodo fácilmente.

4.3.3. BinaryListMap

Se presentan dos tipos de implementación paralela: *lock-based* y *lock-free*, particularmente la versión *lock-free* tiene dos diferentes de métodos, uno utiliza la función recursiva *Actualizar_Punteros*, otro es la implementación del *arreglo auxiliar* para realizar la búsqueda binaria. Se explicarán las ventajas o las desventajas entre los dos métodos de algoritmos non-blocking en la siguiente sección.

4.3.3.1. Lock Based

Para mostrar el método de la ditribución de datos por cada hilo, se presenta el esquema 4.13.

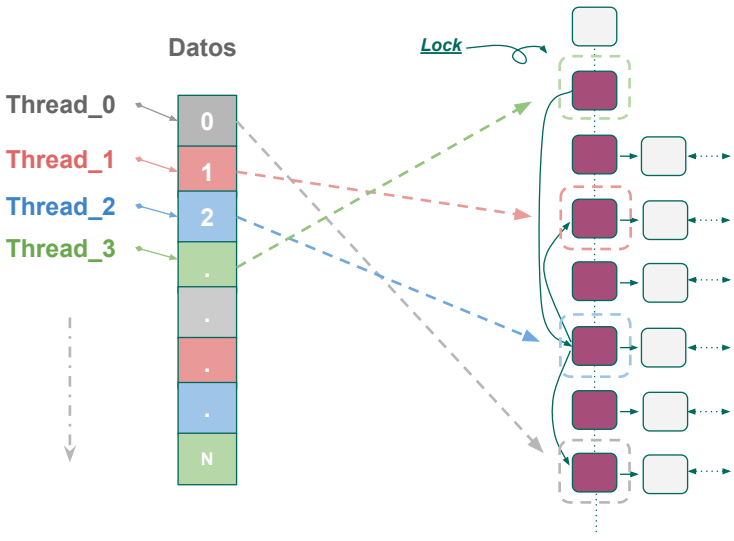

Figura 4.13: El esquema de la distribución de datos por cada hilo (*BinaryListMap lock-based*).

lock-based utiliza instrucciones de bloqueo para resolver el problema de los recursos compartidos. La dificultad del uso *lock* es la minimización de las segmentaciones bloqueadas. Si dicho área es más pequeño, la influencia en el rendimiento total será menos afectada. La sección anterior *LinkedListMap* también fue implementado por el mismo método *lock-based*.

Para adaptar el diseño de búsqueda binaria, se necesitan las nuevas variables globales como: *Tamaño_Previo* y *Hilo_Finalizado*. La variable *Num_Nodos_Insertados* no se ocupa en esta versión, porque se utiliza la sección crítica y los detalles se explicarán en las siguientes secciones.

```
shared(Mapa, Tamaño_Previo, Hilo_Finalizado)
```

Diseño de modelo

La estructura principal no tiene cambio relevante, la única diferencia entre la versión *lock-based* y secuencial es *omp_lock_t lock*, establece el estado de *lock* en *struct node_t* para poder realizar las operaciones del bloqueo por todo recurso compartido (nodo).

```
typedef struct _node{
            omp_lock_t lock;
            char *key;
             private_data data;
             struct _node *next[4];
}node_t;
```

Estructura 4.9: Estructura de nodo *BinaryListMap lock-based*

Función predefinida - Mapa de Búsqueda Binaria Lock-Based

Inicializar el mapa

La función es como el algoritmo de la verión Secuencial (sección 4.2.2.2 *Inicializar el mapa*), pero se agrega *omp_init_lock* para iniciar el uso de la variable *lock*.

Insertar un nodo

La función *Insertar_Mapa* de esta versión de código, utiliza algunos parámetros importantes como *Num_Entradas* (la cantidad total de la entrada), *Indices_Entrada* (el índice de dato actual), *Hilo_Finalizado* (el indicador de la finalización para los subprocesos), *Tamaño_Previo* (el tamaño de mapa generado por la última actualización de puntero que había realizado). El diseño del algoritmo contiene un *loop global* para controlar los movimientos de los subprocesos; el cual siempre permite que los hilos lleguen a la barrera, sin uno de ellos termina su propio trabajoy sale antes del *loop* antes (la variable *Hilo_Finalizado[Hilo_ID()]* se encarga la identificación de este proceso).

El siguiente diagrama es un marco global de la función *Insertar_Mapa*, se muestran los usos de bloqueos, la sección crítica y la instrucción de barrera:

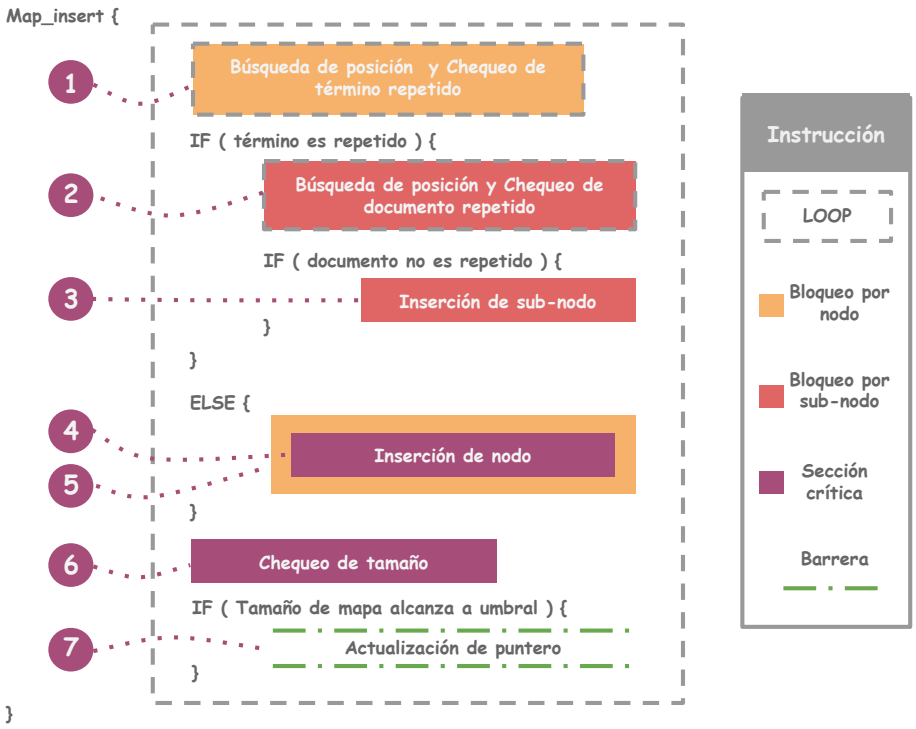

Figura 4.14: El diagrama de procesos de la función *Insertar_Mapa* (*BinaryListMap Lock-Based*).

En las secciones críticas (sección 4 y 6) se utiliza para que no interfiera en el resultado de la revisión de la variable global (*M->Tamaño*). Aunque en el diagrama se muestra que

dicha variable está protegida por el bloqueo de algún nodo (Sección 5), igualmente necesita realizar el chequeo de la condición (Sección 6) entre el tamaño de mapa y el número de umbral establecido; además, si quiere "*leer*" una variable global, hay que evitar que otro subproceso tenga posibilidad de hacer la modificación en ella.

La sección 4 y 6 pertenecen a la misma sección crítica *#pragma omp critical (Verificar_Umbral)*. La sección 4 se encarga de la inserción e incrementar el conteo *M ->Tamaño*. La sección 6 hace la verificación entre la cantidad de términos insertados y el número de umbral con las condiciones de la línea 3 del algoritmo 10 SCSS; la cual permite a todos subprocesos que puedan aprobar y asegura que ningún hilo pueda entrar otra vez dado que la última actualización fue muy cerca a la vez anterior.

Algoritmo 10 *SCSS*: Verificacón de umbral.

```
1:  Actualizar_Punteros ← False
2:  #pragma omp critical (Verificar_Umbral){
3:  if (M → Tamaño <= Tamaño_Previo + M → Umbral + Num_Hilos
    &&M → Tamaño == Tamaño_Previo + M → Umbral
    &&M → Tamaño >= Tamaño_Previo + M → Umbral − Num_Hilos) then
4:      Actualizar_Punteros ← TRUE
5:  end if
6:  }
7:  if (Actualizar_Punteros == TRUE) then
8:      #pragma omp barrier
9:      if (Hilo_ID() == 0) then
10:         Tamaño_Previo ← (M → Tamaño)
11:         Simplificar_Punteros(M → Cabeza.Siguiente[1], M → Tamaño)
12:         Actualizar_Punteros(M → Cabeza.Siguiente[1], NULL, M → Tamaño, −1)
13:     end if
14:     #pragma omp barrier
15: end if
```

Se utilizan dos barreras en donde la sección 7 del diagrama (línea 7-15 en el algoritmo 10 SCSS), nos asegura que todos los subprocesos llegan al mismo punto del código y salgan juntos sin ningún otro hilo que esté procesando todavía. En el sector de la condición *if* (línea 9 en el algoritmo 10 SCSS) solo un único *hilo[0]* puede realizar la simplificación (*Simplificar_Punteros*) y la actualización (*Actualizar_Punteros*) de los punteros. Y los otros hilos esperarán en la última instrucción de barrera hasta que el *hilo[0]* termine su trabajo completamente.

4.3.3.2. Lock Free

Para mostrar el método de la ditribución de datos por cada hilo, se presenta el esquema 4.15.

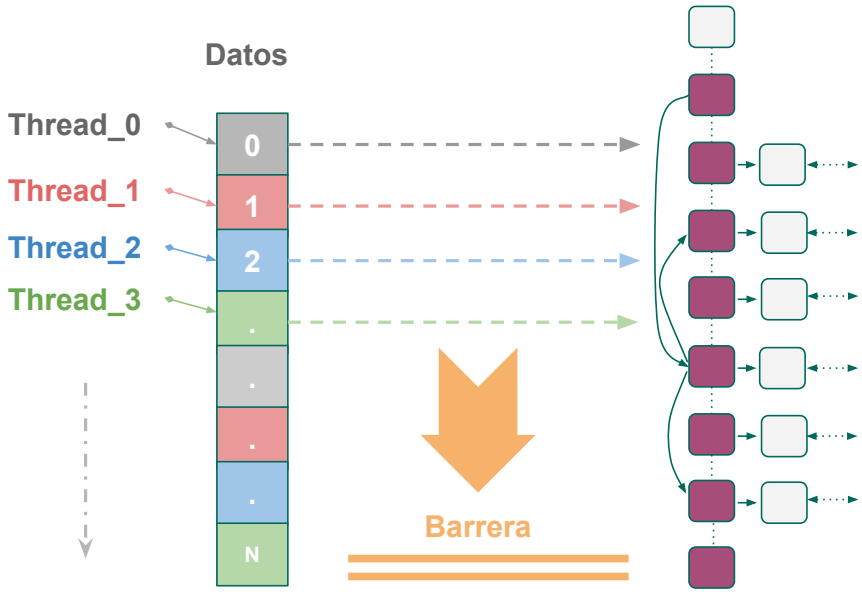

Figura 4.15: El esquema de la distribución de datos por cada hilo (*BinaryListMap lock-free*).

En esta sección se implementará la creación de índice invertido sin el uso de lock para las tareas de multihilos. Se utilizarán las variables globales de la versión anterior *lock-based* y se agregarán unas nuevas variables locales para manejar el flujo de trabajo y evitar la condición de carrera, el problema ABA, entre otros.

LockFreeBinaryListMap

LockFreeBinaryListMap usa el mismo método de la distribución de datos (algoritmo 7 DE), asigna los datos por las iteraciones de los índices. Las variables compartidas que se utilizarán son: *Tamaño_Previo*, *Hilo_Finalizado*; *Hilo_Usando* (un arreglo auxiliar del tamaño de *Num_Hilos*, temporalmente almacena el término de nodo que va a necesitar por cada subproceso), *Num_Nodos_Insertados* (un conteo de arreglo global, contiene la cantidad de nodo insertado por cada subproceso).

```
shared(Mapa, Tamaño_Previo, Hilo_Finalizado , Hilo_Usando, Num_Nodos_Insertados)
```

Diseño de modelo

Ya que lock-free es un tipo de algoritmo de non-blocking, el único cambio de la estructura será en *node_t*, donde se elimina *omp_lock_t lock*.

```
typedef struct _node{
        char *key;
        private_data data;
        struct _node *next[4];
}node_t;
```

Estructura 4.10: Estructura de nodo *BinaryListMap lock-free*

Función predefinida - Mapa de Búsqueda Binaria Lock-Free

Insertar un nodo

En comparación a *lock-based*, *lock-free* posee 2 entradas nuevas (*Hilo_Usando* y *Num_Nodos_Insertados*). Para tener la mayor claridad de implementación, se muestra el diagrama de marco general (figura 4.16), en el cual explica la lógica sobre el uso de la instrucción de barrera, el cómo se reemplaza el bloqueo *lock*:

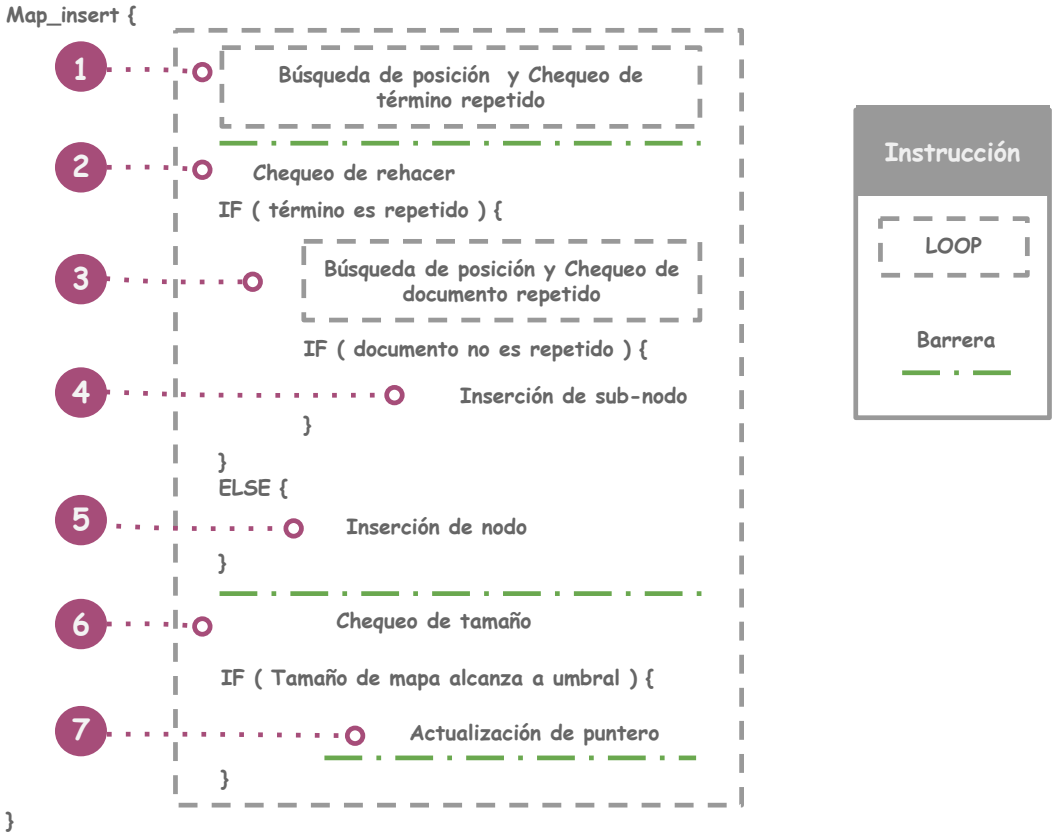

Figura 4.16: El diagrama de procesos de la función *Insertar_Mapa* (*BinaryListMap Lock-Free*).

Loop global utiliza la variable *Hilo_Finalizado* y *Denuevo* (una variable local) que puede controlar el flujo de trabajo total del programa. La variable *Denuevo* es la clave, determina la acción de "*rehacer*" de cada hilo. En un instante, si existen más de dos hilos que están compitiendo por el mismo nodo, la variable indicará cuál nodo no tiene que *rehacer* la búsqueda en el otro ciclo; así, los subprocesos no bloquean el nodo predecesor y evita el conflicto de recurso compartido.

La sección 2 averigua el resultado de la búsqueda anterior, determina si un subproceso es necesario para realizar la acción de rehacer. Para determinar la prioridad de un subproceso, éste es dependiente de la identidad. Mientras tenga un menor número *Hilo_ID()*, este tiene más preferencia de lograr el uso del nodo que los otros subprocesos.

La instrucción de barrera en la sección 6, garantiza que la inserción de los subprocesos terminen juntos en el mismo punto de avance. Así, puede realizar el chequeo de la variable necesaria con la información válida.

La última instrucción de barrera se ubica en la sección 7, que permite la espera de los subprocesos. Dicho esto, podría generar una interrogante: ¿Por qué se utiliza solo una barrera aquí y *lock-based* se usan dos instrucciones de barrera en el mismo sector? Este diseño de código *lock-based* no tiene la segunda barrera de *lock-free* (sección 6 de figura 4.16), pero *lock-based* necesita una barrera extra (sección 7 de figura 4.14) para asegurar que todos los hilos llegan al mismo punto, y no se interfieren por el acceso a un recurso compartido. El siguiente algoritmo corresponde a la sección 7 del diagrama:

Algoritmo 11 *SCS*: La verificacón de umbral.

1: **if** $(M \rightarrow Tamaño <= Tamaño\_Previo + M \rightarrow Umbral + Num\_Hilos$
$\&\& M \rightarrow Tamaño == Tamaño\_Previo + M \rightarrow Umbral$
$\&\& M \rightarrow Tamaño >= Tamaño\_Previo + M \rightarrow Umbral - Num\_Hilos)$ **then**
2: **if** $(Hilo\_ID() == 0)$ **then**
3: $Simplificar\_Punteros(M \rightarrow Cabeza.Siguiente[1], M \rightarrow Tamaño)$
4: $Actualizar\_Punteros(M \rightarrow Cabeza.Siguiente[1], NULL, M \rightarrow Tamaño, -1)$
5: **end if**
6: $\#pragma\ omp\ barrier$
7: **if** $(Hilo\_ID() == 0)$ **then**
8: $Tamaño\_Previo \leftarrow (M \rightarrow Tamaño)$
9: $(M \rightarrow Umbral) \leftarrow (M \rightarrow Umbral) * Reducción$
10: **end if**
11: **end if**

Para acomodar el uso de la barrera en la sección 7, la variable *Tamaño_Previo* es una clave importante. Si la actualización de la variable (línea 8 en el algoritmo 11 SCS) se coloca en antes de la barrera como la de versión *lock-based* (línea 10 en el algoritmo 10 SCSS), podría ocurrir que el subproceso *Hilo[0]* haga la modificación de valor tan rápido que otros subprocesos no alcancen a pasar la primera condición *if* del algoritmo 11 SCS.

La línea 9 del algoritmo 11 SCS fue un logro del resultado de los experimentos para mejorar el rendimiento total del algoritmo *Insertar Un Nodo*. Los detalles se van a explicar en la sección 5.2 Experimento Concurrente del capítulo 5 Experimentos.

LockFreeBinaryArrayMap

La diferencia de las entradas de variables entre *LockFreeBinaryArrayMap* y *LockFreeBinaryListMap* es utilizar *Arreglo_Punteros* para realizar el trabajo de búsqueda binaria . Tal arreglo tiene un tamaño de espacio considerable, y esto es dependiente del nuevo umbral (*Arreglo_Punteros->Umbral*). Se explicarán los detalles en el capítulo *Experimentos*, el cómo seleccionar un buen umbral.

```
shared(Mapa, Tamaño_Previo, Hilo_Finalizado , Hilo_Usando, Num_Nodos_Insertados, Arreglo_Punteros)
```

Diseño de modelo

En el diseño de estructura lock-free anterior, se utiliza la lista doble enlazada como una base del diseño. Ahora, en *LockFreeBinaryArrayMap* no es necesario ocupar tantos punteros en *struct node_t* para realizar la operación de búsqueda binaria, puesto que tiene un arreglo auxiliar aparte, se reemplaza el funcionamiento de los punteros de saltos. Por lo tanto, el diseño del modelo para este mapa será basado en la lista simple enlazada como *LinkedListMap*.

LockFreeBinaryArrayMap tiene una estructura aparte, ya que *Arreglo_Punteros* es un tipo de arreglo que almacena los punteros de saltos necesarios, se requiere tener el tipo de variable en forma del doble puntero *node_t*, para retener los multi-enlaces de nodos. La estructura contiene una variable *size_instant*, dado que no se ocupa el mismo tamaño de arreglo siempre, dependiente del umbral instantáneo y el número de términos insertados, va a estar creciendo el uso de espacio que corresponde para el tamaño de *Arreglo_Punteros* (*Array_Pointers*).

```
typedef   struct   _Array_Pointers{
            node_t** node;
            int   size_instant ;
            int   size ;
            int   umbral;
} Array_Pointers_t ;
```

Estructura 4.11: Estructura de arreglo de puntero *Array_Pointers_t*

Función predefinida - Mapa de Arreglo Binario Lock-Free

Actualización de arreglo global

El objetivo de esta función es igual a la función *Actualizar_Punteros* de *LockFreeBinaryListMap* (el algoritmo 6 AP). El arreglo de punteros no se actualiza después de cada inserción nueva, ya que dicho arreglo tendrá una gran cantidad de elementos en ello y es muy costoso si se actualiza frecuentemente.

Básicamente la lógica de llenar nuestro arreglo *Arreglo_Punteros->nodo[i]* se muestra de la siguiente manera:

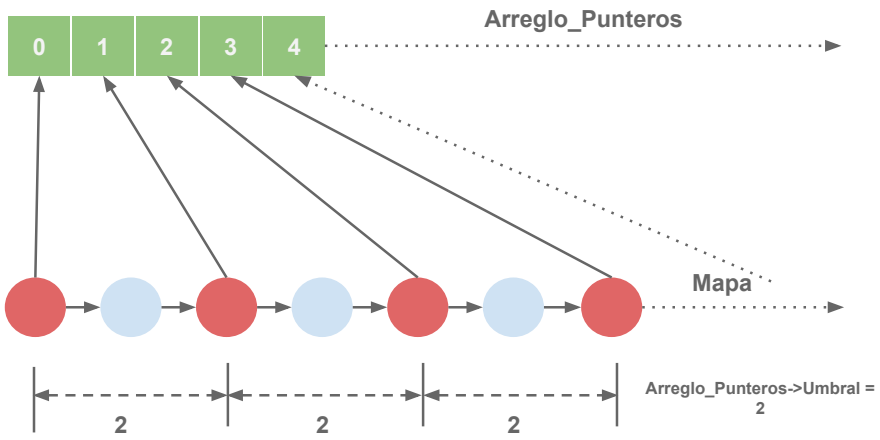

Figura 4.17: Un ejemplo de la función *Actualizar_Arreglo_Punteros*.

La figura 4.17 es un ejemplo del *Arreglo_Punteros->Umbral = 2*, la idea es que cada hilo hace la asignación para su elemento propio. El método es similar a lo que se ha aplicado en la distribución de datos primarios (el algoritm 7 DE). El algoritmo utiliza la identificación de hilos y calcula múltiplos de ellas, los elementos de arreglo se consideran como varias partes encargadas por el diferente subproceso, lo que corresponde a lo siguiente:

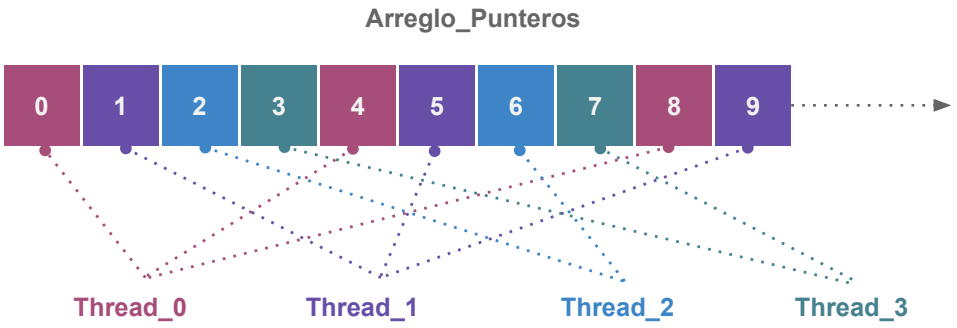

Figura 4.18: La distribución de los trabajos por hilos.

Insertar un nodo

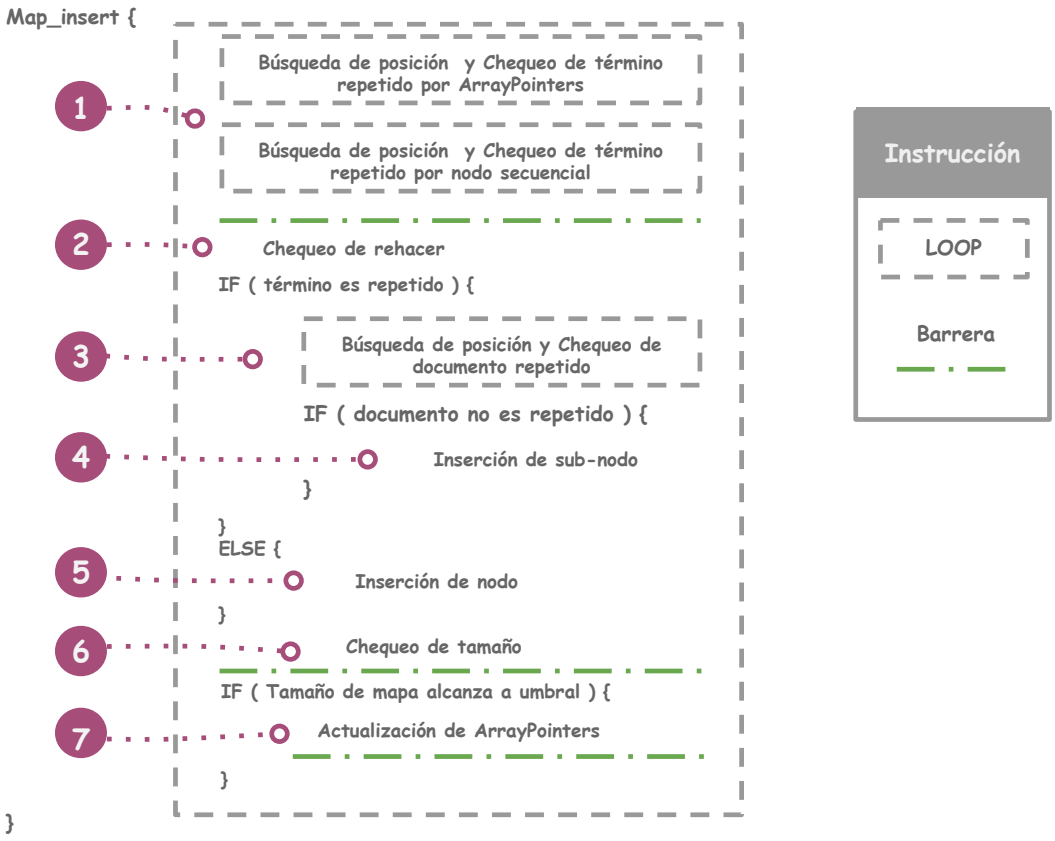

Figura 4.19: El diagrama de procesos de la función *Insertar_Mapa* (*BinaryListMap Lock-Free*).

Se puede observar que no tiene mucho cambio de lógica para el esquema general del programa (figura 4.19). Este diseño de *lock-free* tiene un umbral extra para *Arreglo_Punteros*, la sección 1 necesitará dos fases separadas, una de búsqueda y la otra de verificación del término repetido. La primera fase utiliza *Arreglo_pointers* para realizar la búsqueda binaria, pero se necesita la segunda fase que ayuda a completar el ciclo de la búsqueda, ya que el umbral del arreglo nos impide encontrar una posición exacta. Así, la segunda fase utiliza el resultado anterior y luego revisa los nodos uno por uno secuencialmente hasta que encuentra el nodo de la inserción.

La idea de implementación *LockFreeBinaryArrayMap* es mejorar el rendimiento de la sección 7. *LockFreeBinaryListMap* actualiza los punteros de salto mayor con un solo subproceso para realizar el trabajo completo. Ahora esta nueva versión actualiza el arreglo auxiliar *Arreglo_Punteros* con todos los hilos existentes, se distribuyendo la cantidad de operaciones por cada hilo separado.

4.4. Resumen del Capítulo

Las implementaciones del *Paralelismo* de la presente tesis provienen de la generación de los métodos *Secuenciales*, los cuales fueron utilizados como base del diseño, añadiendo la funcionalidad de multi-hilos sobre él; especialmente al momento de observar el diseño *LinkedListMap*, se trata de no modificar la estructura total del código Secuencial y aplicar las operaciones del *omp_lock_t*.

El tema más complejo dentro de las dificultades para realizar la búsqueda binaria, es cómo mantener el efecto de los puntos de saltos y no perder el rendimiento total del programa; la versión Paralelismo de *BinaryListMap* (*lock-free* y *lock-based*) presenta este problema de forma más relevante que otros, ya que la función de la actualización de punteros no se realiza en el proceso en forma paralela. Por lo tanto, la versión *LockFreeBinaryArrayMap* es una opción alternativa para resolver el problema mencionado anteriormente, utilizando el arreglo de saltos y distribuyendo el trabajo de la asignación por cada hilo (ver detalle en figura 4.18).

Capítulo 5

Experimentos

En el presente capítulo se realiza el experimento y la medición del tiempo para diferentes versiones de código. Se usaron las siguientes cantidades de datos: 100.000, 500.000, 1.000.000, 2.000.000; considerando el tema de cómo saturar las pruebas de plataforma paralela, se decidió utilizar tal cantidad de datos anteriormente mencionados para comprobar el comportamiento del experimento. Todos los datos son términos sin repetición y donde cada término se representa como clave única, y los datos de prueba (los caracteres de las palabras) se generan aleatoriamente, así como también la longitud de los términos.

Tabla 5.1: Propiedades del Servidor utilizado

S.O.	Procesador	CPU(s)	Memoria	Compilador
GNU Debian System Linux, kernel 3.10.0-327.36.3.el7.x86_64	**Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz**	**24**	**31907MB**	**GCC version 4.8.5, flags: -03 -fopenmp**

5.1. Experimento secuencial

La siguiente tabla muestra el resultado de rendimiento de *LinkedListMap*. Aunque 500.000 datos es equivalente a 5 veces 100.000 datos, el tiempo de medición obtenido es 65 veces más lento. Esto último es porque la cantidad de términos insertados crece y alarga la distancia entre el nodo inicial y el nodo objetivo. Mientras más grande sea la lista, se necesitará más tiempo para encontrar el objetivo.

Tabla 5.2: Tiempo de ejecución (seg.) LinkedListMap

Datos	**100.000**	**500.000**	**1.000.000**
Tiempo(seg)	74.294227	4873.39956	21963.16607

La versión *BinaryListMap* posee la variable *umbral*, la cual es importante para determinar el tiempo óptimo y realizar la actualización de puntero. Se utilizó una muestra de la base de datos (1 %, 2 %, 5 %, 10 % de la cantidad total de entrada) para encontrar un umbral óptimo.

Tabla 5.3: Tiempo de ejecución (seg.) BinaryListMap

Umbral	**100.000**	**500.000**	**1.000.000**
1 %	0.937649	10.28877	25.22488
2 %	0.595306	7.505986	20.583445
5 %	0.83609	13.720898	49.727609
10 %	2.421041	41.520832	145.771666

Como se muestra en la tabla de arriba, un umbral de 2 % tuvo mejor rendimiento que otra prueba. La relación entre umbral y tiempo de ejecución, se puede estimar considerando el comportamiento del resultado de medición tal como se observa en la siguiente gráfica:

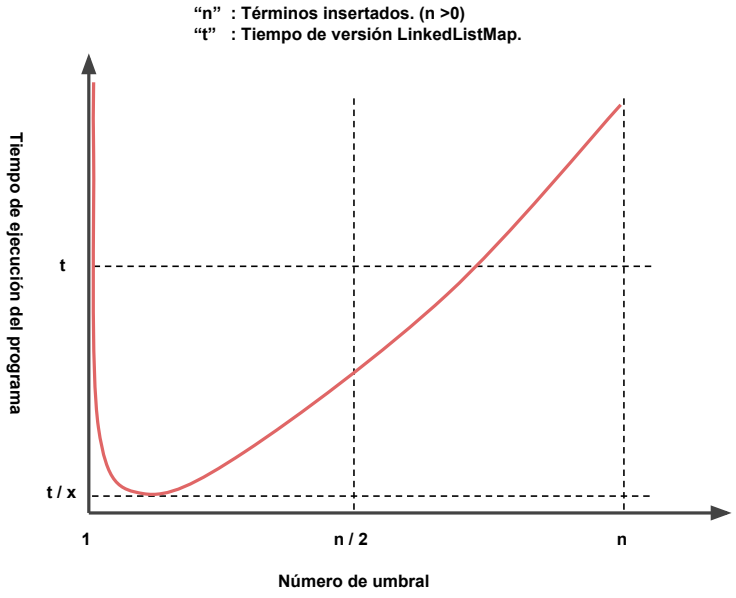

Figura 5.1: Gráfica de la estimación sobre umbral y el rendimiento del nuestro sistema.

Se puede observar que en el método *BinaryListMap* (versión secuencial) no es conveniente usar un umbral demasiado pequeño o muy grande. Un umbral pequeño significa que se necesita hacer una actualización de puntero varias veces, y un umbral grande no alcanza para lograr el

rendimiento lo que trae el algoritmo de búsqueda binaria. Se hace necesario un balance que compense el tiempo de actualización y mejore el rendimiento de la búsqueda por los saltos mayores.

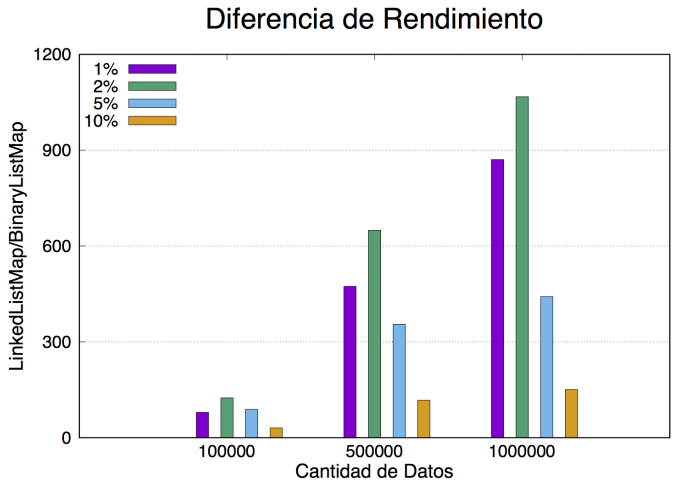

Figura 5.2: Diferencia de rendimiento, el eje Y muestra el valor de la división entre *LinkedListMap* y *BinaryListMap*.

El gráfico (figura 5.2) consta de dos resultados de medición (*LinkedListMap* y *BinaryListMap*), en donde el eje *Y* es el tiempo de *LinkedListMap* dividido por el tiempo de *BinaryListMap*, así se puede apreciar las diferencias entre las dos versiones de diseño. Desde el gráfico se puede considerar que si se tienen más datos, el rendimiento de la búsqueda binaria será más relevante, donde 1.000.000 de datos logró ser más eficiente alrededor de 1000 veces ya que ocupó un umbral conveniente (*2 %*).

5.2. Experimento concurrente en algoritmos concurrentes

LockBasedBinaryListMap usa el bloqueo para coordinar el uso de los recursos compartidos. Las siguientes tablas (tablas 5.4, 5.5 y 5.6) muestran el resultado de rendimiento en tiempo de ejecución sobre 2, 4, 8, 12 y 24 hilos (HT, *Hyperthreading*), donde cada uno utiliza los umbrales de 1 %, 2 %, 5 % y 10 %.

Tabla 5.4: Tiempo de ejecución (seg.) LockBasedBinaryListMap - 100.000 datos

Umbral \hilos (HT)	**2**	**4**	**8**	**12**	**24**
1 %	0.996511	0.927745	1.055472	1.295578	1.588627
2 %	0.645207	0.57657	0.631704	0.849548	1.102022
5 %	0.764275	0.576161	0.523996	0.595496	0.840109
10 %	1.726177	1.263926	0.933967	0.867923	0.870577

Tabla 5.5: Tiempo de ejecución (seg.) LockBasedBinaryListMap - 500.000 datos

Umbral \hilos (HT)	**2**	**4**	**8**	**12**	**24**
1 %	11.368276	10.650008	11.31254	13.231755	14.758374
2 %	7.611044	6.636709	6.475079	7.794133	8.757184
5 %	10.184373	6.557412	5.983035	5.755537	6.02197
10 %	25.273341	14.375666	12.466171	9.67271	8.351885

Tabla 5.6: Tiempo de ejecución (seg.) LockBasedBinaryListMap - 1.000.000 datos

Umbral \hilos (HT)	**2**	**4**	**8**	**12**	**24**
1 %	29.305096	25.442446	28.793891	30.937367	34.121142
2 %	22.387195	16.220925	15.994957	18.099841	20.064425
5 %	38.02428	21.793837	16.448466	15.592058	15.173217
10 %	112.529888	62.42935	40.133153	33.422582	26.762777

A continuación se busca el umbral óptimo de *LockFreeBinaryListMap* de manera similar a la aplicada anteriormente a *LockBasedBinaryListMap*:

Tabla 5.7: Tiempo de ejecución (seg.) LockFreeBinaryListMap - 100.000 datos

Umbral \hilos (HT)	**2**	**4**	**8**	**12**	**24**
1 %	0.931169	0.895387	0.976929	1.010795	1.043981
2 %	0.58915	0.550141	0.592251	0.586223	0.594018
5 %	0.707095	0.543679	0.474925	0.41422	0.401386
10 %	1.60775	1.165033	0.816787	0.655414	0.594249

Tabla 5.8: Tiempo de ejecución (seg.) LockFreeBinaryListMap - 500.000 datos

Umbral \Hilos (HT)	**2**	**4**	**8**	**12**	**24**
1 %	10.395515	9.400957	10.741938	11.322222	11.206722
2 %	6.947075	6.074419	6.383782	6.332146	6.218915
5 %	10.120642	7.368586	5.900052	4.787729	4.671584
10 %	27.091767	19.539279	12.865558	8.993257	7.811046

Tabla 5.9: Tiempo de ejecución (seg.) LockFreeBinaryListMap - 1.000.000 datos

Umbral \Hilos (HT)	**2**	**4**	**8**	**12**	**24**
1 %	29.428493	24.06014	25.455128	25.610629	26.982961
2 %	21.167599	16.443861	14.919616	14.658834	15.401467
5 %	33.239125	22.970946	17.053344	13.941459	12.460841
10 %	102.629863	72.70117	44.565068	35.463603	28.042731

Cuando se compra la versión *LockBased* y *LockFree*, el comportamiento de *LockFree* escala de mejor manera que *LockBased*. En la sección de la tabla 5.4 *LockBased* se aprecia que el resultado mostró una diferencia significativa; especialmente para 24 hios *LockFree* presentó mucho mejor rendimiento que el caso de *LockBased*.

El valor 1 (*LockBased/LockFree*) de los diagramas de barras (figura 5.3, 5.4 y 5.5) representa las 2 versiones de código que generaron el mismo resultado de medición. Generalmente los resultados de 24 hilos tienen mejor rendimiento en *LockFree*, ya que si se quiere hacer la inserción de un nuevo término en un mapa de tamaño grande, *LockFree* tiene una probabilidad más baja de realizar la acción "*rehacer*" (la sección 2 de figura 4.16 y 4.19). En cambio, en la versión *LockBased* se usan los bloqueos de recurso compartido para la búsqueda e inserción, mientras un hilo de ejecución está accediendo al nodo, los demás deben esperar hasta que el hilo desbloquee el acceso a dicho nodo.

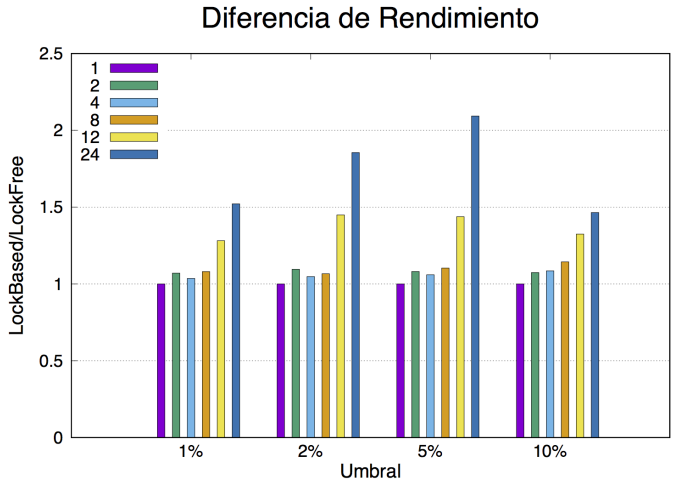

Figura 5.3: Diferencia de rendimiento entre *LockBasedBinaryListMap* y *LockFreeBinaryListMap* para 100.000 datos. El eje Y muestra el valor de la división entre *LockBased* y *LockFree*.

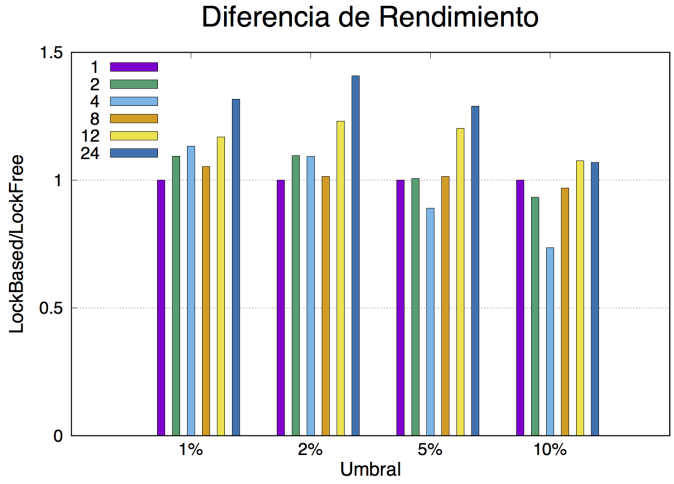

Figura 5.4: Diferencia de rendimiento entre *LockBasedBinaryListMap* y *LockFreeBinaryListMap* para 500.000 datos. El eje Y muestra el valor de la división entre *LockBased* y *LockFree*.

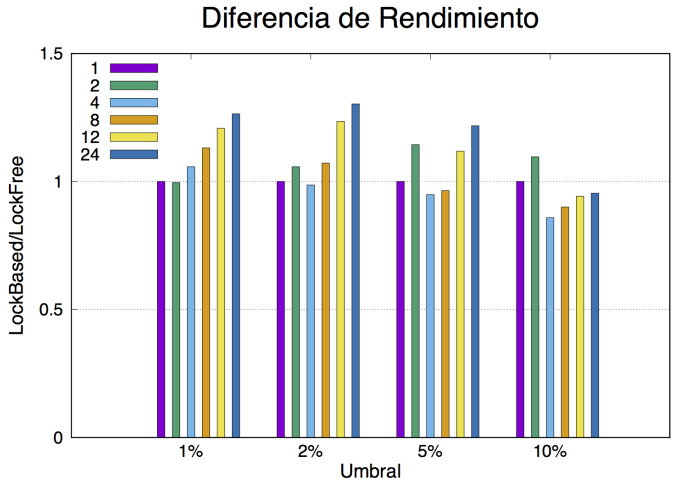

Figura 5.5: Diferencia de rendimiento entre *LockBasedBinaryListMap* y *LockFreeBinaryListMap* para 1.000.000 datos. El eje Y muestra el valor de la división entre *LockBased* y *LockFree*.

La otra observación es sobre la estimación del umbral para el caso de las versiones de paralelismo. Las dos versiones de código *LockFree* y *LockBased* tienen el siguiente problema: no siempre el mejor umbral es 2 %, mientras que si se ejecutan más hilos en proceso, el porcentaje de umbral necesita aumentar de valor, puesto que un buen umbral con poca cantidad de hilos cuesta "***x***" tiempo de espera para llegar al mismo punto de la barrera, no es el mismo tiempo de consumo que si se ocupan más hilos y esperan en conjunto para alcanzar la instrucción de barrera. Es decir, con la misma cantidad de datos y mayor cantidad de hilos en ejecución, se necesitará un umbral mayor. Por lo tanto, es ineficiente implementar la misma cantidad de veces de espera por un nuevo número de hilos, siendo una mejor opción usar un umbral de porcentaje mayor, y así disminuir la frecuencia de actualización del puntero. De esta manera se mejora el rendimiento y no se pierde el tiempo de espera por la actualización del hilo 0, permitiendo que los hilos puedan hacer más cómputo en forma paralela.

En base a la observación anterior, se aplicó una nueva forma de cálculo en donde se obtiene el número de umbral:

$$Umbral\_Next = Umbral\_Now * Reducción \tag{5.1}$$

La idea de esta fórmula es minimizar la cantidad de veces de la actualización de punteros. El nuevo elemento "*Reducción*" es un número flotante, que debe ser mayor que 1, para que el umbral se incremente en cada instante posterior a la función *Arreglo_Punteros* que fue ejecutada.

En la tabla 5.10, se muestra un nuevo resultado del experimento anterior, donde la cantidad de datos es 2.000.000. Se presenta el tiempo de ejecución respecto a los nuevos umbrales y *Reducción*.

Tabla 5.10: Tiempo de ejecución (seg.)
LockFreeBinaryListMap - 2.000.000 datos con
Reducción = 1.2

Umbral \Hilos (HT)	**2**	**4**	**8**	**12**	**24**
0.01 %	28.821838	21.890595	16.040863	13.62654	13.30732
0.02 %	32.664653	21.981678	16.844158	13.850335	14.622987
0.10 %	23.565918	19.122265	16.459138	16.519384	13.090969
0.20 %	25.298161	21.265188	14.920627	15.212422	13.897345
1 %	33.8404	26.873221	17.552204	15.647962	15.823568
2 %	56.31518	40.662986	29.029744	22.127577	20.202953

El número de umbral óptimo ya no es 2 %, puesto que se tiene el nuevo elemento influyente llamado *Reducción*, el cual afecta el rendimiento total del algoritmo por la actualización implicada. Se hicieron varias pruebas con diferentes números de umbral y distinto valor de *Reducción*, el resultado final determinó que el umbral de 0.1 % obtuvo el mejor rendimiento.
Posteriormente se realiza otro análisis de tiempo, que se puede apreciar en la tabla 5.11, esta contiene la medición de costos en segundos para la función *Actualizar los punteros*:

Tabla 5.11: Tiempo de ejecución (seg.) para Actualización de Puntero

Umbral \Hilos (HT)	**2**	**4**	**8**	**12**	**24**
0.01 %	4.413729	4.199736	4.433536	4.720754	5.456043
0.02 %	4.2231	3.945611	4.157906	4.514667	5.271428
0.10 %	4.589111	4.419445	4.684655	5.545724	5.867289
0.20 %	4.239723	3.941788	4.313533	5.336412	5.535763

Tabla 5.12: Porcentaje % entre Actualización de Puntero y Inserción Total

Umbral \Hilos (HT)	**2**	**4**	**8**	**12**	**24**
0.01 %	15 %	19 %	28 %	35 %	41 %
0.02 %	13 %	18 %	25 %	33 %	36 %
0.10 %	19 %	23 %	28 %	34 %	45 %
0.20 %	17 %	19 %	29 %	35 %	40 %

Se puede observar que tener un mejor resultado, implica un mayor costo de tiempo de ejecución para realizar la actualización de puntero. Es comprensible que los subprocesos esperen en la última instrucción de barrera, hasta que el hilo 0 termina dicha actualización. Sin embargo, para mejorar el problema mencionado arriba, se cuenta con una nueva versión para *lock-free* (*LockFreeBinaryArrayMap*). En las siguientes tablas se mostrará el resultado de la medición sobre el rendimiento de la nueva implementación de *lock-free*. Las tablas 5.13 y 5.14 muestran valores utilizando 2.000.000 de datos sin términos repetidos, para experimentos independientes, los que usaron diferentes números de umbral de mapa, *Reducción* y umbral de *Arreglo_Punteros*.

Tabla 5.13: Tiempo de ejecución (seg.)
LockFreeBinaryArrayMap 2.000.000 datos con
Reducción = 1 y Arreglo->Umbral = 5

Umbral \Hilos (HT)	**2**	**4**	**8**	**12**	**24**
0.01 %	883.429406	833.164218	852.351223	1089.615821	1109.465351
0.02 %	441.496948	411.219544	413.285174	508.909385	531.167973
0.10 %	90.846065	84.402408	83.29771	100.497385	104.824635
0.20 %	47.50997	43.644259	42.498705	50.416556	52.634451
1 %	14.643617	12.520592	10.929563	11.825239	12.549606
2 %	17.370017	13.217594	10.158434	8.960895	8.539085

En la tabla 5.13 se mostró que el nuevo método de *lock-free* logra una mayor velocidad, con la variable *Reducción* igual a uno, esto significa que para este valor de la variable el resultado final del experimento no se ve afectado. Para llegar a esto, todos los hilos trabajan de forma concurrente, actualizando en conjunto el arreglo *Arreglo_Punteros*. Esto no es como las implementaciones anteriores que utilizan solo un hilo para completar el proceso de actualización de punteros.

Tabla 5.14: Tiempo de ejecución (seg.)
LockFreeBinaryArrayMap 2.000.000 datos con
Reducción = 1.2 y Arreglo->Umbral = 5

Umbral \Hilos (HT)	**2**	**4**	**8**	**12**	**24**
0.01 %	3.872663	2.627116	2.545516	2.460768	2.177844
0.02 %	3.75437	2.558594	2.492886	2.432985	2.069822
0.10 %	3.762213	2.657021	2.670733	2.604719	2.319958
0.20 %	3.732929	2.61037	2.631713	2.561395	2.164979
AVG	3.78054375	2.61327525	2.585212	2.51496675	2.18315075

En la tabla 5.14, el experimento cuenta con un valor para la variable *Reducción* igual a 1.2. Los resultados obtenidos presentan una mejora sustancial con respecto al experimento anterior. En basa a la experimentación realizada existen 3 factores que impactan en el rendimiento final del programa, estas son *Mapa->Umbral*, *Arreglo->Umbral* y *Reducción*. Los valores de las variables influyen entre ellas, y según las pruebas realizadas por un cierto rango de número predecible, el mejor resultado se logró con:

$$Mapa->Umbral = 0{,}02\,\%; \qquad Arreglo->Umbral = 1; \qquad Reducción = 10;$$

Tabla 5.15: Tiempo de ejecución (seg.)
LockFreeBinaryArrayMap 2.000.000 datos con
Reducción = 10 y Arreglo->Umbral = 1

Umbral \Hilos (HT)	**2**	**4**	**8**	**12**	**24**
0.01 %	3.285249	2.019746	1.614023	1.34432	1.089368
0.02 %	3.078175	1.88038	1.583791	1.292575	1.0376
0.10 %	3.110892	1.952382	1.600854	1.359417	1.052426
0.20 %	3.031234	1.920441	1.588235	1.332662	1.031342
AVG	3.1263875	1.94323725	1.59672575	1.3322435	1.052684

Tabla 5.16: Tiempo de ejecución (seg.) para Actualización de Puntero

Umbral \Hilos (HT)	**2**	**4**	**8**	**12**	**24**
0.01 %	0.01843	0.018994	0.022054	0.024473	0.025494
0.02 %	0.042402	0.042649	0.054983	0.05775	0.060919
0.10 %	0.017579	0.018478	0.021207	0.024466	0.026168
0.20 %	0.039048	0.040709	0.052193	0.057542	0.059535
AVG	0.02936475	0.0302075	0.03760925	0.04105775	0.043029
%	1 %	2 %	2 %	3 %	4 %

Es muy interesante que el mejor resultado de medición requiere un *Arreglo->Umbral* igual a "*1*", esto significa que va a producir un arreglo de punteros, el que va a requerir el mismo tamaño de un mapa completo. Esta acción necesitará ocupar mucho más espacio de memoria.

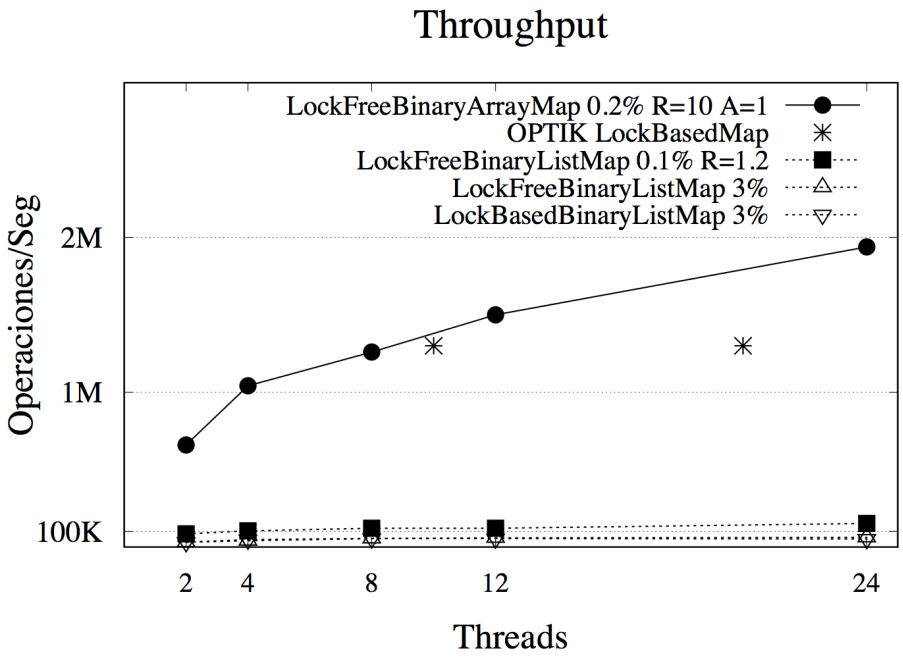

Figura 5.6: Comparación de *Throughput* (operaciones por segundo)

En relación con lo anterior, se puede observar una comparación de *Throughputs* (figura 5.6), lo que indica que para los valores mencionados anteriormente *LockFreeBinaryArrayMap* (*Umbral = 0.2 %, Reducción = 10 y Arreglo->Umbral = 1*) logra un mejor rendimiento en comparación a los otros algoritmos para una cantidad mayor de hilos, lo que en consecuencia genera una mayor diferencia de *throughput* entre ellos.

¿Costará más tiempo si se tiene un *Arreglo->Umbral* menor en la actualización de arreglo? Sí, pero el tiempo no es tan significativo como se pudiera esperar. Esto, se debe el diseño del algoritmo utiliza todos los hilos que trabajan en la función, y donde cada uno realiza un recorrido completo revisando de un nodo a otro nodo secuencialmente, no importando que el número de *Arreglo->Umbral* sea grande o pequeño. Además, la cantidad de operaciones que realiza la asignación de arreglo, se divide por el número de hilos existentes. Por lo tanto, se compensa el generar un arreglo largo (la cantidad de términos en el mapa) de punteros, debido a que la búsqueda binaria logrará un rendimiento superior.

Capítulo 6

Conclusiones

A lo largo de la presente tesis, se ha mostrado que implementar un algoritmo de creación de un índice invertido no es simple si se quiere obtener un resultado correcto, que además, sea eficiente en tiempo de ejecución. Se desarrollaron métodos alternativos en los conceptos *lock-based* y *lock-free*, donde el primero bloquea el acceso a ciertos datos usando las instrucciones de bloqueo que entrega *OpenMP*, mientras que el segundo utiliza barreras de sincronización. En especial, la implementación de la versión *lock-free*, requiere la coordinación de los recursos compartidos sin uso de *lock* para los subprocesos que no tengan el problema ABA o la condición de carrera, situacion que puede ocurrir entre ellos. El diseño de la estructura de datos también es importante, los nodos extras para el salto mayor no son recursos livianos, puesto que generalmente un mapa tiene una cantidad enorme de términos almacenados; las dos primeras versiones de *BinaryListMap* tienen dos punteros extras más que *LinkedListMap* por cada nodo generado. Se necesita una solución como *LockFreeBinaryArrayMap*, que compensa el costo complementario balanceando el tiempo de la inserción y el espacio de memoria a utilizar, siendo este método el que obtuvo mayor rendimiento.

Los resultados experimentales comprobaron que la implementación de *búsqueda binaria* mejora el rendimiento del mapa en forma significativa, y si se suma al método *Arreglo_Punteros*, ahorra los dos punteros extras de cada nodo. A través de los experimentos, se muestra que la selección de umbral es un valor cuantitativo que determina el rendimiento total del programa. La importancia entre umbral y el tiempo de ejecución se puede observar en los resultados de

mediciones, donde se mostró su influencia respecto al rendimiento de los algoritmos propuestos.

6.1. Trabajos Futuros

Entre los trabajos futuros de la presente tesis que se destacan, se presenta los siguientes:

- Buscar una solución para implementar la búsqueda binaria u otro algoritmo de búsqueda alternativa en la lista de documentos. Puesto que es necesario ajustar el problema entre el número de *ranking* y la revisión de documentos repetidos para reducir el ciclo de la búsqueda redundante.
- Considerar la utilización del menor número de punteros posibles como *LinkedListMap* y adaptar las operaciones sobre él, acondicionando el algoritmo de búsqueda con el diseño de la estructura mapa.
- Implementar la versión del algoritmo para el ambiente de los coprocesadores.

Bibliografía

[Afe10] Yanovsky Afek, Korland. Quasi-linearizability: relaxed consistency for improved concurrency. *Data structures*, 2010.

[Bro10] Casper J. Chafi H. Olukotun K. Bronson, N. G. A practical concurrent binary search tree. *Data structures*, 2010.

[Bus02] Lawrence Bush. Lock free linked list using compare & swap. *Data structures*, 2002.

[Cha01] Rohit Chandra. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[DT15] Guerraoui R. David, T. and V. Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. *Concurrency*, 2015.

[Ell10] Fatourou P. Ruppert E. & van Breugel F Ellen, F. *Non-blocking binary search trees*. Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing (pp. 131-140). ACM, 2010.

[ELM08] Nir Shavit Edya Ladan-Mozes. An optimistic approach to lock-free fifo queues. *Data structures*, 2008.

[FH07] Keir Fraser and Tim Harris. Concurrent programming without locks. *Non-Blocking*, 2007.

[Fra04] K. Fraser. Practical lock-freedom. *Non-Blocking*, 2004.

[Gal12] Niall Gallagher. Concurrent-trees. *Data structures*, 2012.

[Gar15] Hitesh Garg. Binary search algorithm - fundamentals, implementation and analysis. 2015.

[G.H96] S. Parthasarathy M.Scott G.Hunt, M.Michael. An efficient algorithm for concurrent priority queue heaps. *Data structures*, 1996.

[Gid08] Anders Gidenstam. *NbAda Non-blocking Algorithms and Data Structures Library*. 2008.

[Gue16] V Guerraoui, R. & Trigonakis. Optimistic concurrency with optik. *Optimistic concurrency*, 2016.

[Har01] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. *Non-Blocking*, 2001.

[Hen04] Shavit N. Yerushalmi L. Hendler, D. A scalable lock-free stack algorithm. *Parallelism*, 2004.

[Her11] N. Herlihy, M. & Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2011.

[How12] J Howley, S. V. & Jones. *A non-blocking internal binary search tree*. Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures (pp. 161-171). ACM, 2012.

[HT10] Shavit Hendler, Incze and Tzafrir. Flat combining and the synchronization-parallelism tradeoff. *Synchronization Parallelism*, 2010.

[Man08] Raghavan P. & Schütze H. Manning, C. D. *Introduction to information retrieval*. Cambridge: Cambridge university press, 2008.

[M.H07a] M.Tzafrir M.Herlihy, N.Shavit. Concurrent cuckoo hashing. technical report. *Data structures*, 2007.

[MH07b] Nir Shavit Moshe Hoffman, Ori Shalev. The baskets queue. *Data structures*, 2007.

[MHS11] Y. Lev M. Herlihy and N. Shavit. Concurrent lock-free skiplist with wait-free contains operator. *Non-Blocking*, 2011.

[Mic02] Maged Michael. High performance dynamic lock-free hash tables and list-based sets. *Data structures*, 2002.

[MS96] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. *Non-Blocking*, 1996.

[Nat13] Savoie L. H. & Mittal N Natarajan, A. *Concurrent Wait-Free Red Black Trees*. Symposium on Self-Stabilizing Systems (pp. 45-60). Springer International Publishing, 2013.

[Nat14] N Natarajan, A. & Mittal. *Fast concurrent lock-free binary search trees*. ACM SIGPLAN Notices (Vol. 49, No. 8, pp. 317-328). ACM, 2014.

[Ora16] Oracle. Concurrenthashmap. *Data structures*, 2016.

[OS03] Nir Shavit Ori Shalev. Split-ordered lists - lock-free resizable hash tables. *Data structures*, 2003.

[Par01] Nick Parlante. Linked list basics. *Data structures*, 2001.

[Pre12] Jeff Preshing. An introduction to lock-free programming. *Concurrency*, 2012.

[Pre16] Jeff Preshing. New concurrent hash maps for c++. *Concurrency*, 2016.

[Pug90] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Data structures*, 1990.

[Ram13a] Pedro Ramalhete. Lock-free and wait-free, definition and examples. *Non-Blocking*, 2013.

[Ram13b] Pedro Ramalhete. Why is wait-free so important? *Concurrency*, 2013.

[Ram15] N Ramachandran, A. & Mittal. *A fast lock-free internal binary search tree*. Proceedings of the 2015 International Conference on Distributed Computing and Networking (p. 37). ACM, 2015.

[SF13] Damian Dechev Steven Feldman, Pierre LaBorde. Concurrent multi-level arrays: Wait-free extensible hash maps. *Data structures*, 2013.

[SHS05] Victor Luchangco Mark Moir William N. Scherer III Steve Heller, Maurice Herlihy and Nir Shavit. A lazy concurrent list-based set algorithm. *Data structures*, 2005.

[SL00] N. Shavit and I. Lotan. Skiplist-based concurrent priority queues. *Data structures*, 2000.

[Tre86] R. Kent Treiber. Systems programming: Coping with parallelism. *Optimistic concurrency*, 1986.

[ZPP+13] Miguel Ángel Niño Zambrano, Dignory Jimena Pérez, Diana Maribel Pezo, Carlos Alberto Cobos Lozada, and Gustavo Adolfo Ramírez González. Procedure for building semantic indexes based on domain-specific ontologies. 2013.