UNIVERSIDAD AUTÓNOMA METROPOLITANA – IZTAPALAPA
DIVICIÓN DE CIENCIAS BÁSICAS E INGENIERÍA

Tablas de ruteo IP dinámicas basadas en árboles Multibit

Idónea comunicación de resultados presentada por
Israel De Olmos Ramírez
Para obtener el grado de
Maestro en Ciencias y Tecnologías de la Información

Asesores:

Dr. Miguel Ángel Ruiz Sánchez
Dr. Ricardo Marcelín Jiménez

Jurado calificador:

Presidente: Dr. Javier Gómez Castellanos
Secretario: Dr. Cesar Jalpa Villanueva
Vocal: Dr. Miguel Ángel Ruiz Sánchez

México, D.F. Abril 2015

Casa abierta al tiempo
UNIVERSIDAD AUTÓNOMA METROPOLITANA
Iztapalapa

PC y TI

UNIVERSIDAD AUTÓNOMA METROPOLITANA – IZTAPALAPA
DIVICIÓN DE CIENCIAS BÁSICAS E INGENIERÍA

Tablas de ruteo IP dinámicas basadas en árboles Multibit

Idónea comunicación de resultados presentada por
Israel De Olmos Ramírez
Para obtener el grado de
Maestro en Ciencias y Tecnologías de la Información

Asesores:

Dr. Miguel Ángel Ruiz Sánchez
Dr. Ricardo Marcelín Jiménez

Jurado calificador:

Presidente: Dr. Javier Gómez Castellanos
Secretario: Dr. Cesar Jalpa Villanueva
Vocal: Dr. Miguel Ángel Ruiz Sánchez

México, D.F. Abril 2015

Resumen

Debido al crecimiento exponencial de internet, el tamaño de las tablas de ruteo también se ha incrementado. Un enrutador típico del *backbone* de internet en el año 2014 llega a almacenar alrededor de 500 000 prefijos de red (1). Este crecimiento propicia que los enrutadores se tornen lentos al realizar la reexpedición de paquetes, ya que les toma más tiempo decidir por cuál de sus interfaces deberá ser reenviada la información.

Para reducir estos tiempos, se ha optado por almacenar los prefijos de red dentro de estructuras que permitan realizar actualizaciones y búsquedas de información. El tiempo de búsqueda dependerá de la complejidad de la estructura. La solución clásica a este problema es utilizar árboles binarios, ya que esta estructura nos permite almacenar prefijos de red de distintas longitudes, por otra parte, tanto el proceso de búsqueda como el proceso de actualización tienen una complejidad lineal (2) (3).

Sin embargo, debido a que las longitudes de los prefijos pueden ser grandes (de 32 bits en el peor de los casos para IPV4) las búsquedas pueden tornarse lentas ya que en esta técnica se compara solamente un bit del prefijo a la vez. En este trabajo se propone una estructura de almacenamiento que se basa en árboles Multibit los cuales, a diferencia de los árboles binarios, pueden comparar varios bits a la vez, mejorando los tiempos de búsqueda (al número de bits comparados se le conoce como *stride*).

Los árboles Multibit o Tries-Multibit ofrecen la ventaja de que las operaciones de búsqueda son más rápidas que en el algoritmo clásico que utiliza árboles binarios, sin embargo, para las estructuras Trie-Multibit en su forma simple no existen operaciones de actualización, es decir no existen operaciones de inserción y borrado (que no impliquen la reconstrucción de todo el árbol) que garanticen la integridad de los datos almacenados, por lo que se pueden perder elementos en el proceso de actualización del árbol, esta problemática será abordada con más detalle en el capítulo 3.1.

La estructura propuesta en este documento permite a los árboles Multibit realizar operaciones de actualización garantizando que no exista pérdida de información en el proceso, para esto, se propone y se evalúa la estructura Trie-Multibit con respaldo. Dicha estructura es implementada en lenguaje C y son evaluados los algoritmos de inserción, borrado y búsqueda.

Después de analizar los resultados experimentales, podemos concluir que el almacenamiento en la estructura Trie-Multibit con respaldo es más conveniente que el almacenamiento en árboles binarios, ya que las operaciones de búsqueda son más rápidas y que a pesar de que las operaciones de actualización son más lentas, éstas pueden ser toleradas debido a que ocurren con menor frecuencia.

Agradecimientos

Quisiera agradecer de manera muy especial a las personas que hicieron posible la culminación de este trabajo. Especialmente a mi madre Imelda Ramírez Ruiz y a mi padre Hipólito De Olmos Quiroz, quienes nunca me han dejado de apoyar y que gracias a ellos he podido completar esta tesis y esta etapa de mi vida. También quisiera agradecer a mis hermanos Alexis, Jesús, Hugo y a mi prometida Nancy quienes gracias al apoyo y comprensión de todos ellos fue posible dedicar el tiempo necesario para poder terminar el presente proyecto.

Agradezco también a la Universidad Autónoma Metropolitana por haberme formado desde la licenciatura hasta el día de hoy y por haberme aceptado en el programa de la Maestría en Ciencias y Tecnologías de la información. Quiero agradecer a mis asesores los doctores Ricardo Marcelín Jiménez y Miguel Ángel Ruiz Sánchez, no solamente por haberme aceptado como su estudiante, sino también por el gran apoyo que me brindaron en la realización y culminación de este trabajo. Muchas gracias a los doctores Cesar Jalpa Villanueva y Javier Gómez Castellanos por haberme hecho el honor de participar como parte del jurado de la presente tesis.

En general quisiera agradecer a mis profesores, compañeros y amigos del postgrado, quienes siempre han estado dispuestos a ayudarme bajo cualquier circunstancia. También quisiera agradecer al Consejo Nacional de Ciencia y Tecnología por haberme otorgado el apoyo financiero durante la realización de este postgrado ya que sin esto, la presente tesis jamás hubiera sido posible. Muchas gracias a todos ellos.

Contenido

Lista de figuras

Lista de tablas

1 Introducción

En este primer capítulo veremos un panorama general del proyecto realizado, abordando aspectos como los objetivos principales, la metodología, la problemática y algunos conceptos básicos del funcionamiento de las redes de computadoras.

Visualizaremos un breve panorama de la estructura de internet poniendo especial atención en el funcionamiento de los enrutadores ya que éstos están directamente relacionados con la propuesta de este trabajo.

1.1 Objetivos principales

El objetivo principal de este proyecto, es proponer un esquema de almacenamiento para tablas de reexpedición IP dinámicas; es decir este esquema debe cumplir con las operaciones de inserción, borrado y búsqueda de elementos (prefijos de red e interfaces de salida). El esquema propuesto debe estar basado en TDA´s (tipos de datos abstractos) llamados Trie-Multibit, los cuales en su forma simple solamente soportan la operación de búsqueda, ya que en las operaciones de inserción y borrado no garantizan la integridad de la información (esto se aborda con detalle en la sección 3.1).

La estructura propuesta debe ser implementada en lenguaje C y deberá ser evaluada para así mostrar cuál es su desempeño frente al esquema de almacenamiento clásico que utiliza árboles binarios.

1.2 Justificación

En los últimos años estar conectado a internet se ha convertido en un servicio prácticamente esencial dentro de nuestra vida cotidiana, ya que nos permite intercambiar información de todo tipo, los nuevos servicios nos permiten almacenar información dentro de servidores remotos, intercambiar audio y video en tiempo real, recibir correspondencia digital, realizar transacciones bancarias, pagos, etc. Debido a que las aplicaciones paras las que utilizamos el internet son cada vez más diversas, la gran red ha crecido de forma exponencial (1).

En la Figura 1 se muestra el crecimiento que ha tenido en los últimos años una tabla de ruteo típica del *backbone* de internet, se puede observar que en el año 2014 ésta tabla almacena más de 500 000 prefijos de red activos, lo que significa que en este enrutador se comunican más de 500 000 subredes, en esta misma figura se puede ver que el crecimiento de esta tabla ha sido exponencial por lo que podemos intuir que seguirá creciendo.

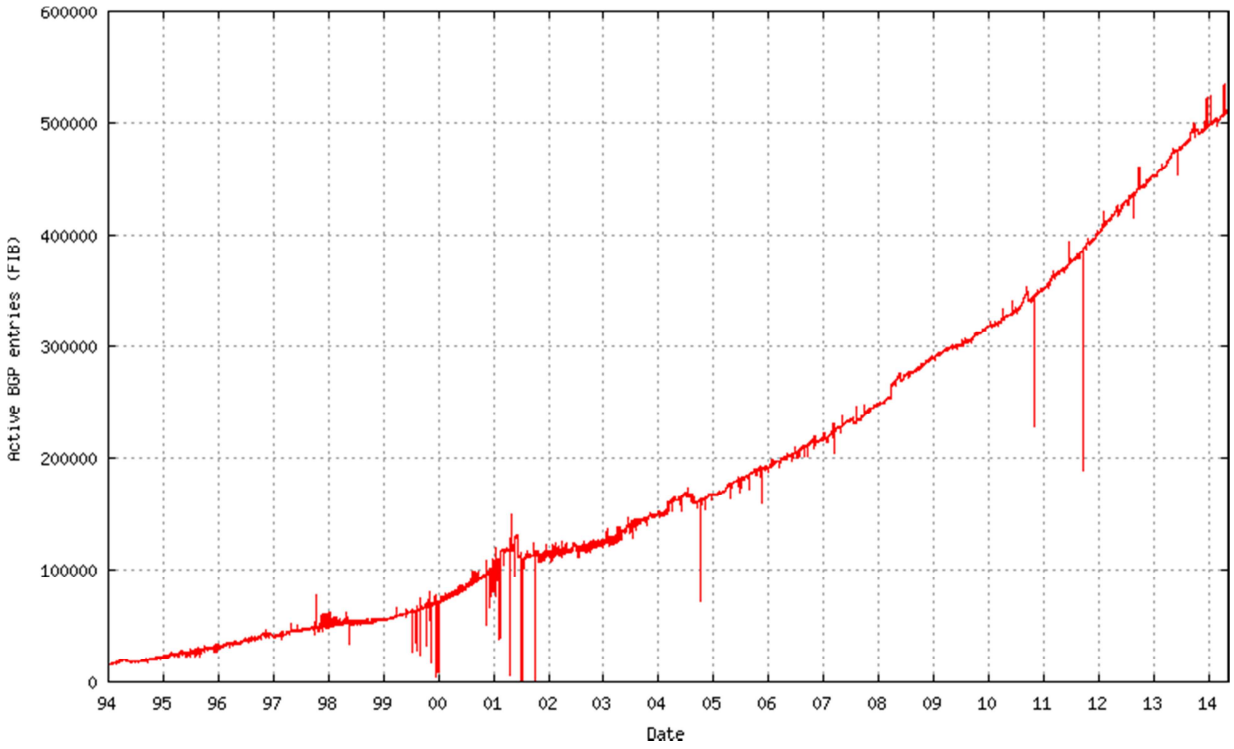


Figura 1 Datos obtenidos del Sistema Autónomo AS6447, última actualización realizada el miércoles 7 de mayo de 2014 a las 18:00:00 (UTC+1000) [1].

Debido a que los enrutadores son los encargados de comunicar cada una de las subredes, es necesario que posean los recursos suficientes para almacenar, actualizar y sobre todo para consultar la información contenida dentro de la tabla de ruteo. Dichas consultas deben ser lo

más rápidas posibles ya que hoy en día es común encontrar aplicaciones en tiempo real que utilizan el internet, como por ejemplo video-juegos, video-conferencias, video-llamadas, llamadas telefónicas sobre IP, etc. Este tipo de aplicaciones exigen que la red sea especialmente rápida ya que el retardo les afecta directamente en la calidad del servicio que ofrecen.

1.3 Metodología

La metodología que se siguió para alcanzar los objetivos planteados fue la siguiente:

- Estudio del estado del arte sobre las estructuras que han sido propuestas para almacenar prefijos de red.
- Estudio de posibles técnicas de construcción de Tries Multibit y desventajas al usarse para almacenar prefijos de red.
- Elaboración de propuesta conceptual de la estructura Trie Multibit con respaldo soportando operaciones de actualización.
- Implementación de propuesta en lenguaje C e implementación del almacenamiento clásico en árboles binarios.
- Elaboración de propuesta de experimentos para medir el desempeño de los árboles Trie-Multibit con respaldo, comparándolos con los árboles binarios.
- Evaluación de la estructura propuesta mediante los experimentos propuestos.
- Reporte de resultados.

1.4 Redes de comunicaciones

Una red de comunicaciones se define como *"un conjunto de dispositivos que brindan un servicio: la transferencia de información entre usuarios localizados en distintos puntos geográficos"* (4). Es decir, se desea que exista comunicación entre cualesquiera terminales que pertenezcan a la red, por lo que se puede ver como una abstracción que desde el punto de vista del usuario es algo desconocido pero funcional, ésta es la razón por la que una red se simboliza con una nube y solamente se indican las terminales (también llamados *hosts*) que se encuentran en esa red, de esta manera la red se encarga de establecer la comunicación entre las terminales sin importar que tan compleja sea la propia red.

Internet es en realidad una red de comunicaciones que esencialmente comunica computadoras (entre otros dispositivos) y éste engloba a su vez a otras redes. Los dispositivos que se conectan a internet deben poder comunicarse entre ellos sin importar que en realidad pertenezcan a subredes distintas (redes dentro de internet), esta idea es la que se presenta en la Figura 2.

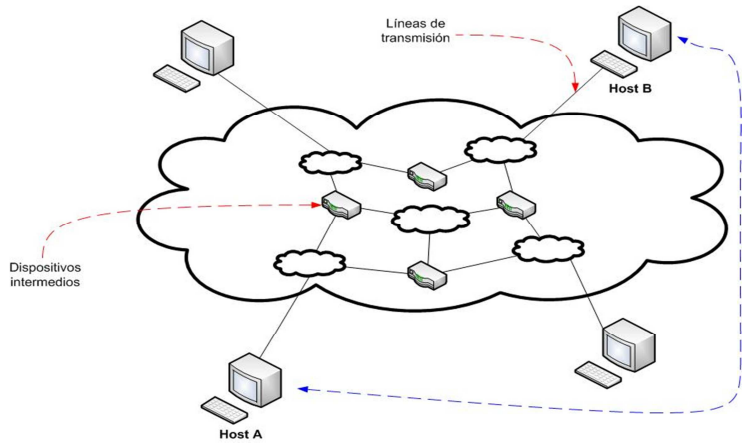


Figura 2 Esquema general del funcionamiento de internet

Como se puede observar en la Figura 2, si el *host* A trata de comunicarse con el *host* B, es necesario que los paquetes de información se desplacen hacia su destino atravesando distintas subredes. Por lo tanto deben existir líneas de transmisión, además de dispositivos intermedios que se encarguen de recibir los paquetes de información y redirigirlos hacia la subred más cercana al *host* destino, a estos dispositivos se les conoce como dispositivos de encaminamiento o enrutadores. Para que un enrutador tome la decisión apropiada al redirigir los paquetes, se debe conocer la dirección destino de ellos y con base en ésta, se debe decidir la ruta que deben seguir los paquetes de información.

1.5 Dispositivos de encaminamiento o enrutadores

En la Figura 2 podemos observar que para que los paquetes puedan viajar de una subred a otra es necesario que pasen a través de dispositivos intermedios llamados enrutadores, a continuación se proporciona una breve explicación de su funcionamiento.

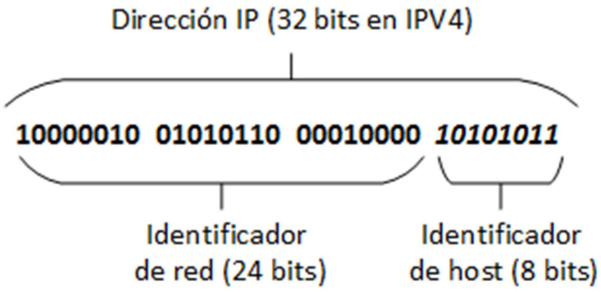
Un enrutador es un dispositivo que maneja la conexión entre dos o más redes, posee un número de interfaces de red a la entrada y a la salida, su función es la de encaminar los paquetes de información por la mejor ruta hacia su destino (2). Para poder realizar el encaminamiento de paquetes los enrutadores utilizan el protocolo IP (*Internet Protocol*) en este documento nos referiremos al protocolo IP versión 4 (IPV4).

En el protocolo IPV4 se especifica que cada dirección IP (asignada a cada *host* o terminal de trabajo) consta de 32 bits y este identificador es único en la red, un ejemplo de dirección IP es 10000010 01010110 00010000 01000010 en notación binaria o 130.86.16.66 en notación decimal. Internet es en realidad una red de redes, es decir se tienen distintas subredes interconectadas a través de enrutadores y cada *host* se encuentra conectado en realidad a una de las subredes de internet (5).

Debido a la arquitectura de internet, es necesario identificar a cada subred, por lo que cada dirección IP se divide en dos partes, el identificador de red y el identificador de *host*, donde el identificador de red indica en qué subred se encuentra el *host* y el identificador de *host* indica el *host* dentro de la subred. El identificador de red o también conocido como prefijo de red, corresponde con los primeros bits de la dirección IP (de izquierda a derecha), de tal manera que todos los *host* de la misma subred tienen los primeros bits idénticos en su dirección IP. No todos los identificadores de red tienen la misma longitud y dependiendo de ésta será el número de *hosts* que pueden albergar.

Ejemplo 1:

Podemos analizar el caso de una red con un identificador de 24 bits.



Otra forma de escribir el identificador de red:

10000010 01010110 00010000 *

Esta red puede albergar $2^{32-24} = 2^8 = 256$ *hosts*.

Otra forma de denotar el identificador de red es escribiéndolo en decimal e indicar el número de bits que posee después de una diagonal, para el ejemplo anterior el identificador de red se denota: 130.86.16.0/24.

Un enrutador debe realizar dos operaciones fundamentales para la reexpedición de paquetes, la primera ocupa los algoritmos llamados “algoritmos de ruteo”, los cuales esencialmente se

encargan de que el enrutador tenga información actual del estado de las subredes, es decir si alguna ha dejado de funcionar o si se ha creado una subred nueva, estos algoritmos se encargan de recopilar información de la topología de la red, el costo de las líneas de comunicación, etc.

Una vez que se ha recopilado la información necesaria, ésta se guarda en una tabla dentro de la memoria del enrutador, a esta tabla se le conoce como "Tabla de ruteo". De tal manera que cuando llega un paquete al enrutador, éste consulta su tabla de *ruteo* y con base en la dirección destino del paquete decide hacia donde se debe reexpedir el paquete para que llegue a su destino, lo cual nos lleva a la segunda tarea principal del enrutador que es precisamente la de realizar la búsqueda en la tabla lo más rápido posible. La propuesta de este documento se enfoca en optimizar esta tarea.

1.6 Crecimiento de internet

Internet ha tenido un gran desarrollo en los últimos años, su crecimiento ha sido de orden exponencial y por consiguiente, el número de paquetes de información que deben ser transportados a través de la red también ha aumentado, además hoy en día se han desarrollado distintas aplicaciones de tiempo real como la telefonía por internet (VoIP), video-llamadas, video-conferencias, etc. Por lo tanto, las aplicaciones del internet de hoy en día exigen que la red pueda transportar grandes cantidades de paquetes de información y además que esto se realice en el menor tiempo posible.

Para poder resolver esta problemática, investigadores de diversas áreas han desarrollado tecnologías que permiten el transporte de los paquetes a través de las líneas de comunicación en tiempos realmente pequeños utilizando los últimos avances en la tecnología de la fibra óptica, además los avances en tecnologías ópticas también han llegado a introducirse en el diseño de circuitos integrados propiciando que los procesadores de los dispositivos de encaminamiento sean cada vez más rápidos, sin embargo es necesario también avanzar en el diseño de algoritmos que utilicen el menor número de ciclos de reloj y sobre todo el menor número de accesos a memoria para poder reexpedir un paquete, debido a que importará poco el tener la mejor tecnología si no hacemos un uso eficiente de ella.

Históricamente el tráfico sobre internet se ha duplicado cada año, la velocidad de las transmisiones ópticas cada 7 meses, pero la velocidad de los enrutadores cada año y medio (2) si no se hace algo para optimizar el tiempo de reexpedición, cada enrutador terminará siendo un cuello de botella para el transporte de paquetes y esto propiciaría que la red no funcione adecuadamente, con esto surge la necesidad de estudiar el proceso de encaminamiento y así mejorar el desempeño de los enrutadores.

1.7 Búsqueda en las tablas de ruteo

Las tablas de ruteo del *backbone* de internet actualmente llegan a tener almacenados alrededor de 500 000 prefijos de red activos, lo cual dificulta la búsqueda, esto sin mencionar que la búsqueda de un prefijo de red no es tarea fácil debido a que no se trata de una búsqueda clásica como la de un elemento en un arreglo. En un principio se atacó este problema creando solamente tres clases de redes (Figura 3), cada una de ellas se caracterizaba por poseer un número distinto de bits en el identificador de red y en el identificador de host. Podían existir 2^7 redes de la **clase A**, 2^{14} redes de la **clase B** y 2^{21} redes de la **clase C**, y podían albergar 2^{24}, 2^{16} y 2^8 hosts respectivamente. Para poder diferenciar el tipo de red, se utilizaba un identificador al principio de la dirección IP y dependiendo de éste se podía decir de manera exacta a qué tipo de red pertenecía esa dirección IP (5) (6).

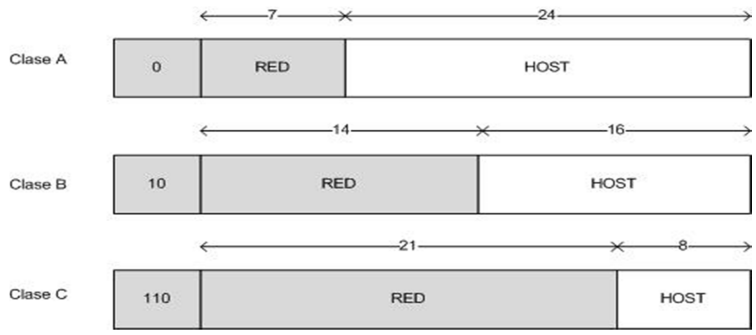


Figura 3 Clases de redes A, B y C que existían en los inicios del internet.

De tal manera que cuando se quería redireccionar un paquete, bastaba con leer los primeros bits de la dirección destino, se determinaba la clase de la red a la que pertenecía la red destino y con esto ya se sabía la longitud que debía tener el identificador de red así como el identificador de host. De esta forma se sabía exactamente el prefijo que se buscaba en la tabla de ruteo y la búsqueda se reducía a buscar un elemento perfectamente conocido dentro de un arreglo. Posteriormente cuando el internet comenzó a tener un gran crecimiento, este esquema tuvo que ser modificado ya que el número de redes que podía albergar internet era muy limitado (2^7 de la **clase A** + 2^{14} de la **clase B** + 2^{21} de la **clase C**) y por consiguiente las direcciones IP disponibles se estaban terminando.

El nuevo esquema propuesto se denominó esquema de direccionamiento CIDR (*classes inter-domain routing*). A diferencia del anterior, propone eliminar las clases y que la longitud de los prefijos de red sea de cualquier tamaño (de 1 a 32 bits), con esto se explota mejor el conjunto de direcciones IP. El esquema CIDR es el que se utiliza actualmente y presenta el inconveniente

de que la dirección IP ya no presenta información acerca de la longitud del prefijo de red, por lo tanto no se sabe exactamente que prefijo se está buscando dentro de la tabla de ruteo, sino que se busca el prefijo más largo que coincida con los primeros bits de la dirección IP destino, a este prefijo se le conoce como prefijo BMP (*best matching prefix*). (6) (7). Ahora que sabemos que la dirección IP se divide en dos partes (identificador de red e identificador de host) es conveniente mencionar que en este trabajo se denotará el prefijo de red escribiendo la cadena de bits del identificador de red con un asterisco al final para indicar que faltaría el identificador de host para tener la dirección IP completa.

Ejemplo 2. Si tenemos que redireccionar un paquete con la dirección IP destino igual a **130.86.16.66** y la tabla de reexpedición de nuestro enrutador es la que se muestra en la Tabla 1, el proceso de redireccionamiento sería el siguiente:

Tabla 1 Tabla de ruteo para ilustrar el ejemplo 2

Número de Subred	Prefijo de red	Siguiente dispositivo	Interfaz de salida
1	0*	192.41.8.6	A
2	01000*	192.41.8.30	B
3	011*	192.41.9.25	C
4	1*	192.42.15.250	D
5	100*	192.42.15.41	E
6	1100*	192.42.15.230	F
7	1101*	192.43.9.98	G
8	1110*	192.43.30.2	H
9	1111*	192.43.68.29	I

Si vemos la dirección IP destino en notación binaria:

130.86.16.66 = 10000010 01010110 00010000 01000010

Nos damos cuenta que el prefijo de red más largo que coincide con los primeros bits de la dirección IP destino es el prefijo número cinco ya que a pesar de que no es el más largo de todos, si es el prefijo más largo que coincide con la dirección destino (éste es el prefijo BMP),

por lo tanto el paquete debe ser reexpedido por la interfaz **E** hacia el siguiente enrutador que tiene la dirección 192.42.15.41. Ahora surge la siguiente interrogante ¿cómo acomodar estos prefijos para que la búsqueda pueda llevarse a cabo en el menor tiempo posible, pues una tabla de ruteo en el *backbone* de internet tiene alrededor de 500 000 elementos? A lo largo de este trabajo se presentan las principales soluciones propuestas hasta el momento, en el capítulo 2 se muestra un resumen de dichas estructuras, describiendo sus características y explicando brevemente su funcionamiento.

En la sección 2.4 se estudia con profundidad a los árboles Multibit refiriéndonos a la estructura teórica, el proceso de construcción, se analiza la expansión controlada de prefijos propuesta por Srinivasan y Varghese en (8), de igual forma en el capítulo 3.1 se estudia el algoritmo de búsqueda y las problemáticas que se presentan al tratar de realizar operaciones de actualización.

En el capítulo 3 se presentan las características de nuestra propuesta la cual hemos llamado Trie-Multibit con respaldo, se explican la estructura teórica, el proceso de construcción, cómo esta estructura puede implementar operaciones de actualización así como de búsqueda, se dan características de la implementación y de la serie de experimentos realizados para evaluar el desempeño de nuestra propuesta comparándola con el almacenamiento en árboles binarios.

En el capítulo 4 se muestran los resultados obtenidos al implementar la estructura Trie-Multibit con respaldo en lenguaje C, se hace una comparación con el esquema de almacenamiento en árboles binarios y se evalúa el desempeño de ambas propuestas utilizando una tabla de *routeo* típica del *backbone* del internet (AS6447), por último en los capítulos 5 y 6 se muestran las conclusiones a las que se llegan a partir de los resultados obtenidos y las recomendaciones para trabajos futuros.

Capítulo 2

2 Estado del arte

Para poder optimizar la búsqueda del prefijo BMP dentro de la tabla de ruteo, así como el proceso de actualización de la misma tabla, se ha propuesto almacenarla en estructuras de datos que faciliten dichas tareas, las propuestas han sido variadas y en este capítulo se mencionan las soluciones más representativas. Podemos agrupar las propuestas en esencialmente cuatro tipos, los basados en árboles o tries, basados en tablas Hash, basados en rangos y los basados en mapas de bits, esta clasificación la encontramos de forma más detallada en (9).

2.1 Basados en tries

2.1.1 Trie Binario

El primero de los algoritmos que mencionaremos es el basado en un Trie binario, en la Figura 4 se muestra un esquema de dicha estructura, se puede observar que cada uno de los nodos puede tener como máximo dos nodos hijos, sabemos que es de profundidad tres debido a que si nos colocamos en el nodo más alto (nodo raíz) y trazamos un camino hasta el nodo más lejano, dicho camino contendrá a lo más tres nodos más el nodo raíz (10) (3).

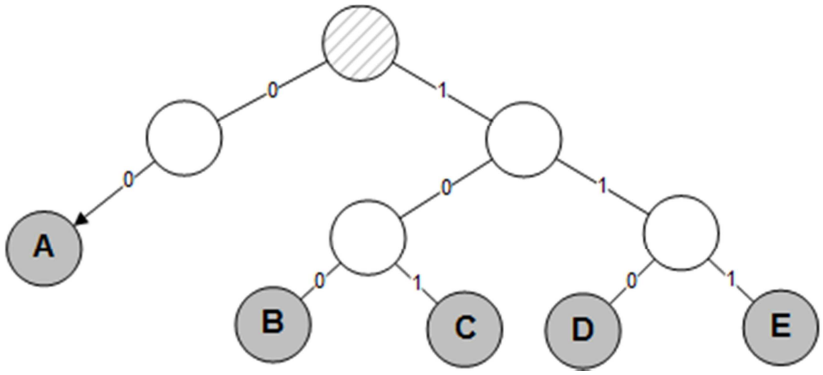


Figura 4 Trie binario de profundidad tres incompleto

En la Figura 4 también se puede apreciar que pueden existir nodos con menos de dos hijos, si este es el caso, se dice que el árbol está incompleto.

Estas estructuras de datos pueden ser utilizadas para almacenar tablas de *ruteo*, el almacenamiento se realiza en cada uno de los nodos de la siguiente forma. Se toma el prefijo de red y se lee en notación binaria de izquierda a derecha y dependiendo del valor de los bits del prefijo se van creando los nodos a la izquierda o a la derecha (si el valor es 0 a la izquierda y si es 1 a la derecha) hasta alcanzar la longitud del prefijo de red y en este último nodo se almacena la interfaz por donde deben reexpedirse los paquetes de información.

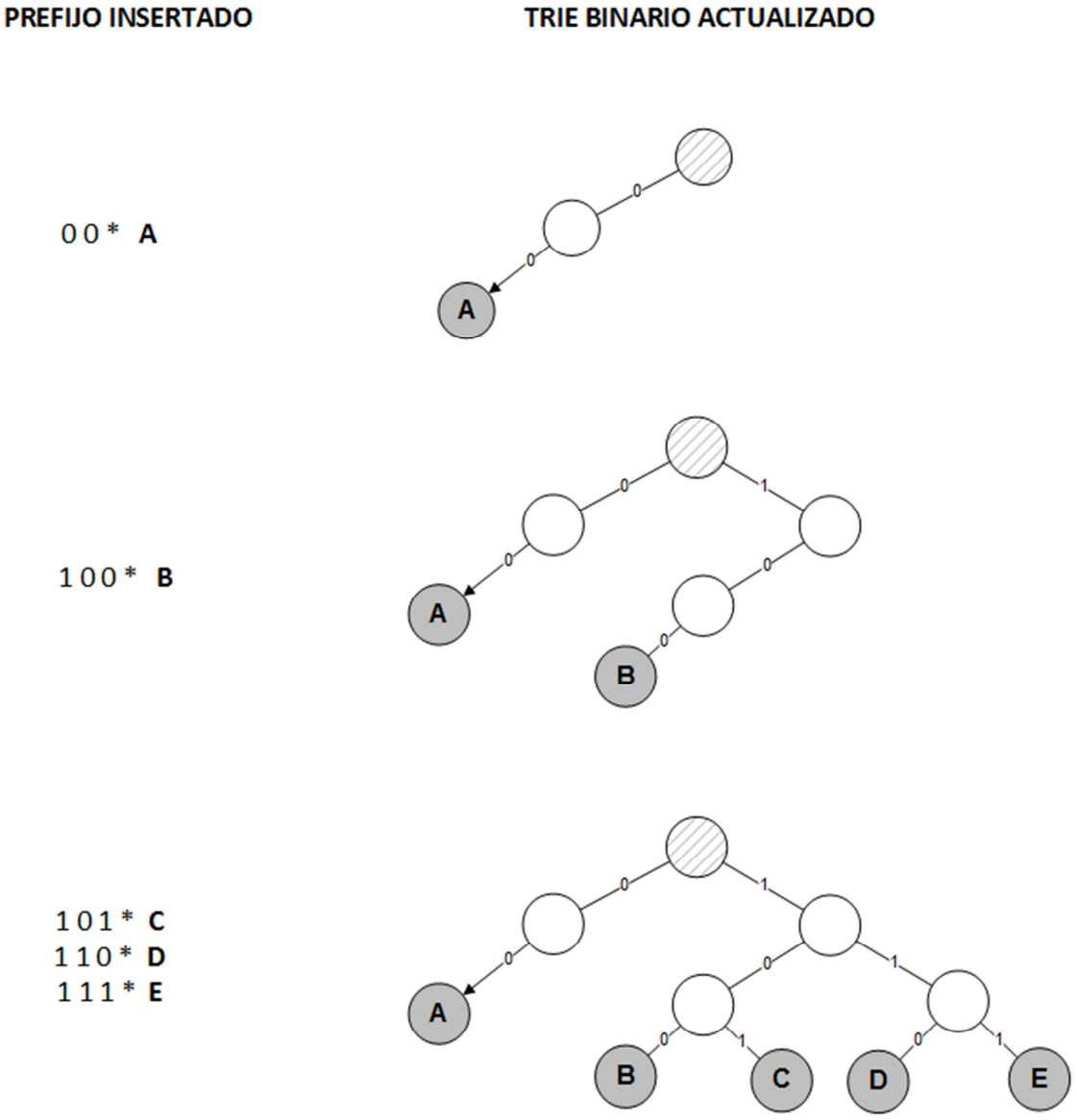


Figura 5 Esquema de inserción de prefijos dentro de un Trie Binario

En la Figura 5 se ilustra el proceso que se sigue al insertar un nuevo prefijo de red dentro de esta estructura, este mismo proceso puede ser utilizado para realizar el borrado de alguno de los elementos. Al utilizar esta estructura para el almacenamiento de prefijos e interfaces nos

damos cuenta que el árbol puede crecer hasta 32 niveles debido a que ésta es la longitud máxima de los prefijos que especifica el protocolo IPV4.

Para realizar la búsqueda del prefijo BMP en esta estructura, primero nos colocamos en el nodo raíz y leemos la dirección IP destino de izquierda a derecha en notación binaria, comparamos uno por uno de los bits y dependiendo de su valor se pasará a la rama izquierda o a la derecha, si en algún momento se han comparado 32 bits o no existe el nodo al que se debe pasar, se asumirá que la interfaz por la que se debe reexpedir el paquete será la última encontrada y el prefijo BMP será la secuencia de bits que se siguió para llegar a ella. Para ilustrar este proceso de búsqueda tomemos la dirección IP destino del ejemplo 2 y el árbol de almacenamiento que se muestra en la Figura 4.

130.86.16.66 = 10000010 01010110 00010000 01000010

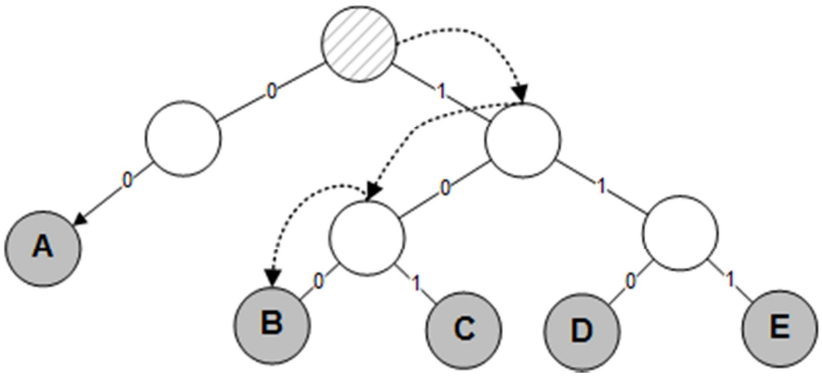


Figura 6 Esquema del proceso de búsqueda seguido en un Trie binario

Al ver la dirección IP en notación binaria vemos que el recorrido de búsqueda en el árbol será derecha, izquierda, izquierda y en este punto se tratará de pasar a la rama izquierda nuevamente pero ésta no existe en el árbol por lo que el prefijo BMP es el prefijo 100* y la interfaz de salida correspondiente es la interfaz B. En general si buscamos un prefijo de longitud *W* dentro de esta estructura, se harán un total de *W* comparaciones antes de encontrarlo y si queremos insertar o borrar algún prefijo, el número de operaciones que realicemos dependerá de su longitud, por tanto la complejidad tanto para la búsqueda como para la actualización de esta estructura es del orden de *W* (se denota como $O(W)$).

2.1.2 Tries de ramas comprimidas (path-compressed Tries)

Esta es una técnica que propone reducir la profundidad de los árboles binarios y con esto también el tiempo de búsqueda de alguno de los elementos almacenados. Debido a que un árbol binario puede albergar prefijos de distintas longitudes, se suelen tener nodos que solamente tienen un hijo. En estos nodos no es necesario comparar algún bit ya que solo existe

un camino, además de que representan memoria extra no necesaria para la búsqueda de elementos. Se propone eliminar dicho nodos del árbol, sin embargo, para que la operación de búsqueda pueda funcionar apropiadamente es necesario agregar información a los nodos que permanecen en el árbol (11) (12).

En la Figura 7 se muestra el Trie binario construido a partir de la tabla de ruteo del ejemplo 2, como se puede observar existen cuatro nodos que solamente tienen un hijo, por lo que deben ser removidos, podemos notar que uno de los nodos a eliminar almacena la interfaz A, por lo que esta interfaz será movida a su próximo descendiente con más de un hijo.

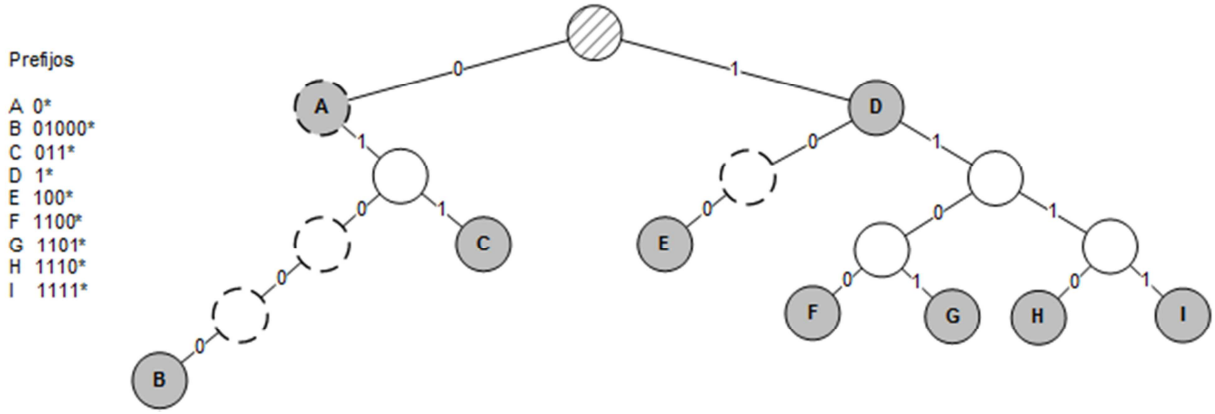


Figura 7 Árbol binario construido a partir de la tabla de ruteo del ejemplo 2.

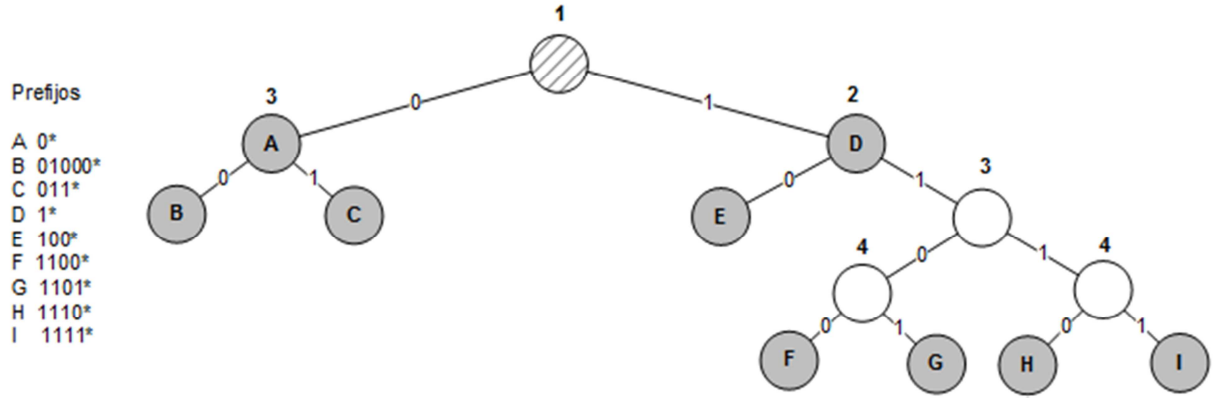


Figura 8 Árbol de rutas reducidas.

En la Figura 8 podemos observar que los nodos que solamente tenían un hijo, han sido removidos del árbol, por lo que algunas ramas han sido comprimidas. Sin embargo, debido a que algunos nodos han desaparecido, debemos indicar cuál es el bit que debe ser comparado para pasar al siguiente nivel, por lo que en la parte superior de los nodos ahora aparece un número que nos da esta información. En el ejemplo 3 se muestra el funcionamiento de este algoritmo.

Ejemplo 3.

Si suponemos que tenemos el árbol de rutas reducidas que se muestra en la Figura 8 y recibimos un paquete con la dirección IP destino igual a:

98.86.16.66 = 011 00010 01010110 00010000 01000010

Para realizar la búsqueda partimos de la raíz, ésta nos indica que hay que comparar el bit número uno, como éste es igual a cero, avanzamos a la rama izquierda y verificamos que el prefijo de la interfaz A sea prefijo de la dirección IP destino. Como el prefijo de la interfaz A si es un prefijo de la dirección IP destino, guardamos el prefijo de la interfaz A como candidato para ser el prefijo BMP, ahora podemos proseguir.

El nodo en el que ahora nos encontramos nos indica que hay que comparar el bit número tres, el cual es igual a uno, por lo que avanzamos a la rama derecha y verificamos que el prefijo de la interfaz C sea prefijo de la dirección IP destino. Como el prefijo de la interfaz C si es un prefijo válido para la dirección IP destino lo guardamos sustituyendo al candidato anterior, ahora como ya no existen ramas hacia donde avanzar podemos decir que el prefijo BMP es el almacenado más recientemente, es decir el prefijo de la interfaz C.

Como podemos ver, para poder localizar el prefijo BMP solamente nos tomó dos comparaciones, por lo que hemos comprobado que la técnica de almacenamiento en árboles con ramas reducidas disminuye el consumo de memoria y el número de comparaciones en las búsquedas. Sin embargo las optimizaciones se pueden hacer siempre y cuando existan nodos con un solo hijo, esto hace que la técnica pierda impacto cuando se tienen en el árbol pocos nodos de este tipo, ya que su complejidad tiende a ser la misma que la de los árboles binarios sin reducción $O(W)$.

2.1.3 Tries de niveles comprimidos (LC Tries)

Esta es una técnica que ha contribuido a las estructuras basadas en Tries, es propuesta por Nilsson y Karlsson en (13) el proceso para construir el Trie es el siguiente. Primero se debe construir el Trie binario, se debe aplicar la técnica de compresión de ramas revisada en la sección 2.1.2 con una ligera modificación, para poder ver las particularidades nos guiaremos en el Ejemplo 4.

Ejemplo 4.

Supongamos que deseamos almacenar la siguiente lista de prefijos.

Tabla 2 Prefijos que deben almacenarse para ejemplo 4

Número de prefijo	Prefijo en notación binaria
0	0000
1	0001
2	00101
3	010
4	0110
5	0111
6	100
7	101000
8	101001
9	10101
10	10110
11	10111
12	110
13	11101000
14	11101001

Como primer paso debemos construir el Trie binario, este se muestra en la Figura 9.

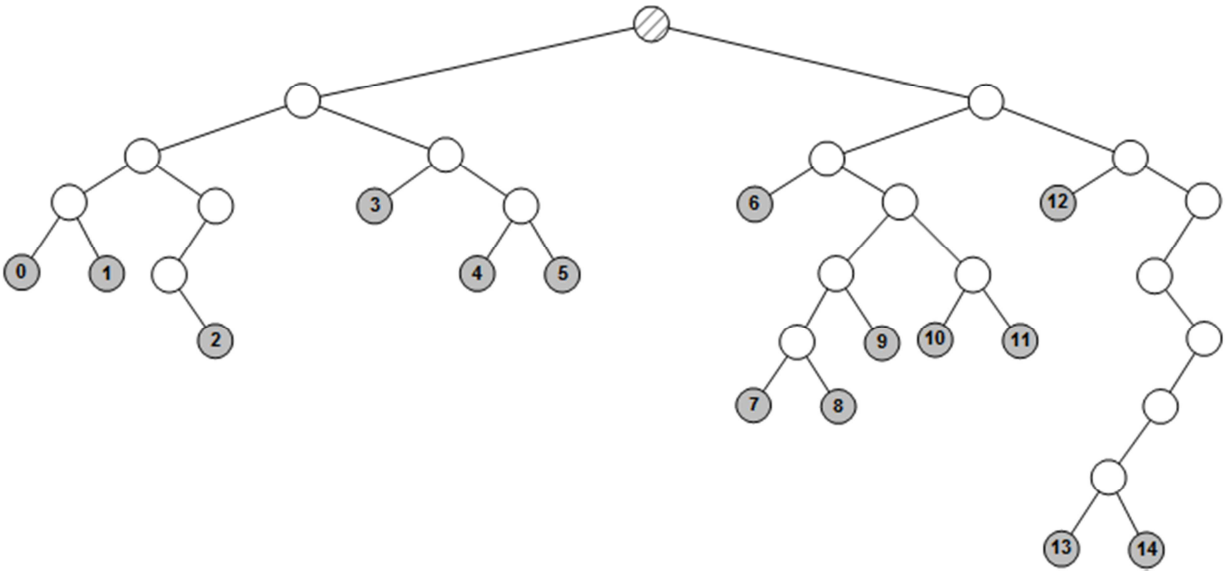


Figura 9 Trie binario construido a partir de la tabla 2.

Una vez que hemos construido el árbol binario, aplicamos la técnica *path-compressed Tries* revisada en el punto 2.1.2, con la diferencia de que en vez de anotar el bit que debe ser comparado para realizar la búsqueda, anotamos el número de nodos que fueron eliminados en la compresión, a este número lo llamaremos **Skip** y podemos ver cómo fue aplicado en nuestro ejemplo en la Figura 10**.**

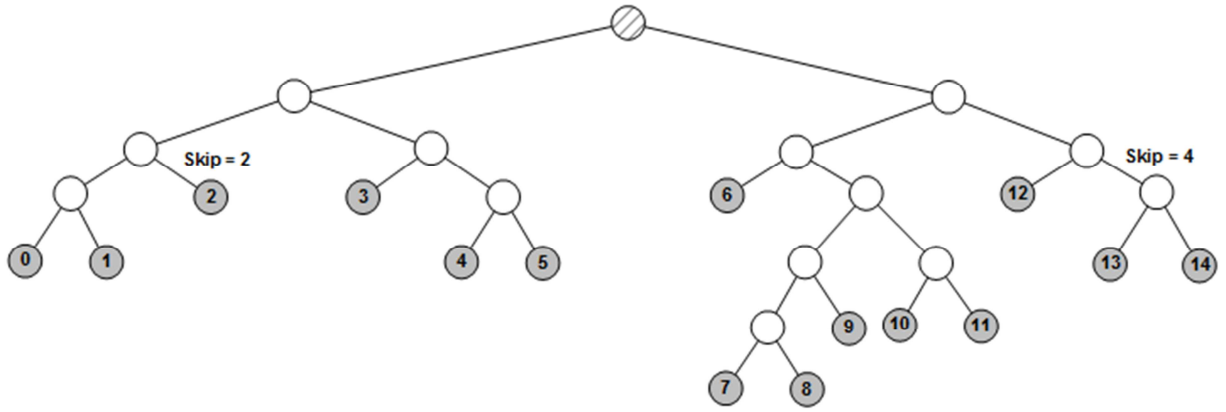


Figura 10 Trie de ramas comprimidas a partir de la tabla 2.

En la Figura 10 podemos ver que nuestro árbol ya no presenta nodos con un solo hijo, por lo que ahora podemos observar que tenemos niveles completos del árbol en los que no tenemos interfaces guardadas, si comprimimos estos niveles, reducimos la profundidad del árbol y con esto el tiempo de búsqueda. Al realizar la compresión de niveles, llegamos al árbol que se muestra en la Figura 11.

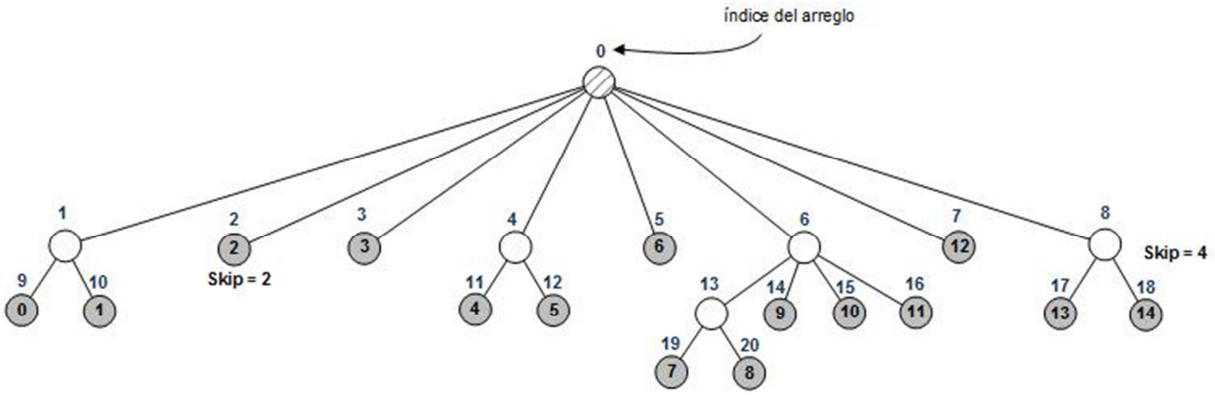


Figura 11 Trie de niveles comprimidos a partir de la tabla 2.

Los autores proponen implementar el trie de niveles comprimidos utilizando un arreglo en el que se almacenen cada uno de los nodos de la siguiente forma.

Tabla 3 Representación del arreglo propuesto por Nilsson y Karlsson para representar el LC Trie.

	Branch	Skip	Pointer
0	3	0	1
1	1	0	9
2	0	2	2
3	0	0	3
4	1	0	11
5	0	0	6
6	2	0	13
7	0	0	12
8	1	4	17
9	0	0	0
10	0	0	1
11	0	0	4
12	0	0	5
13	1	0	19
14	0	0	9
15	0	0	10
16	0	0	11
17	0	0	13
18	0	0	14
19	0	0	7
20	0	0	8

Nilsson y Karlsson proponen que el LC Trie sea almacenado dentro de un arreglo en donde cada elemento representa uno de los nodos del árbol y este a su vez se encuentra compuesto de la siguiente forma.

Los nodos son numerados en forma ascendente comenzando por el nodo raíz (color azul en la Figura 11). El elemento **branch** representa el número de bits necesarios para direccionar el número de hijos del nodo, es decir si, éste número *k* es mayor o igual a 1 el nodo deberá tener 2^k hijos y si *k* es igual a 0 significa que el nodo es una hoja. La siguiente columna contiene el valor de **skip** que indica el número de bits que pueden ser saltados al realizar una operación de búsqueda, por último, la columna **pointer** tiene dos diferentes interpretaciones, para un nodo interno este valor apunta al hijo que se encuentra más a la izquierda y para un nodo hoja, el valor apunta a el prefijo BMP correspondiente. De esta forma se puede implementar un algoritmo de búsqueda como el siguiente.

```
node = trie[0];
pos = node.skip;
branch = node.branch;
adr = node.adr;
while (branch != 0) {
   node = trie[adr + EXTRACT(pos, branch, s)];
   pos = pos + branch + node.skip;
   branch = node.branch;
   adr = node.adr;
}
return adr;
```

Figura 12 Pseudocódigo del algoritmo de búsqueda propuesto.

Donde el arreglo **trie** es el mostrado en la Tabla 3, para mayores detalles podemos consultar la referencia (13).

2.2 Basados en tablas hash

En (14) y (9) se describen propuestas para el almacenamiento de prefijos dentro de las estructuras conocidas como tablas hash. En este esquema se tiene un número fijo de tablas de almacenamiento, los elementos son clasificados y agrupados en dichas tablas. Se calcula una clave a cada elemento y por medio de ella se sabe exactamente en cual tabla debe ser guardado. Por lo que al momento de realizar la búsqueda de algún elemento, basta con calcular la clave que le corresponde y con esto se sabe exactamente en cuál de las tablas se encuentra y la búsqueda se torna muy ágil.

En (14) y (9) podemos ver que para el caso en que deseamos almacenar prefijos de red, la búsqueda no es tan simple ya que como se mencionó en la sección 1.7 no se sabe exactamente cuál elemento se está buscando, si no que se busca el prefijo BMP. Para poder llevar a cabo el almacenamiento de prefijos de red se propone guardarlos dependiendo de su longitud. Se plantea crear un arreglo el cual almacene apuntadores hacia tablas de almacenamiento, el índice del arreglo indicará la longitud de cada uno de los prefijos almacenados en la tabla, por lo que este arreglo a lo más tendrá 32 elementos. El elemento uno apuntará a una tabla que almacenará todos los prefijos de longitud 1, el elemento 2 apuntará a una tabla que almacenará todos los prefijos de longitud dos y así sucesivamente, la idea principal se muestra en la Figura 13.

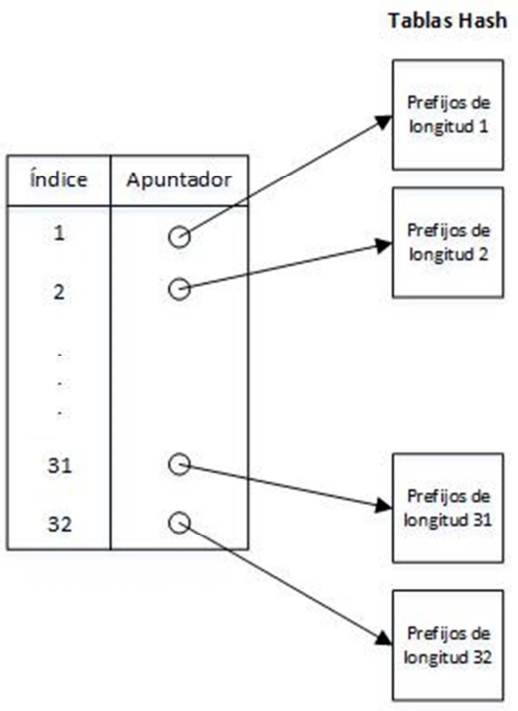


Figura 13 Esquema general del almacenamiento de prefijos en tablas hash.

Si se sigue este esquema, el tiempo de búsqueda dependerá directamente del mecanismo que se tenga para escoger en cuál de las tablas se buscará primero. A continuación se presentan dos mecanismos que se proponen para realizar la búsqueda.

2.2.1 Búsqueda lineal en tablas hash

La búsqueda lineal es el mecanismo más natural para realizar la búsqueda dentro de las tablas hash, esta consiste en buscar primero una coincidencia de prefijo dentro de la tabla que almacena prefijos de mayor longitud, si se encuentra un prefijo que coincida, se toma como el prefijo BMP y si no se encuentra, se continua la búsqueda en la tabla de menor longitud siguiente.

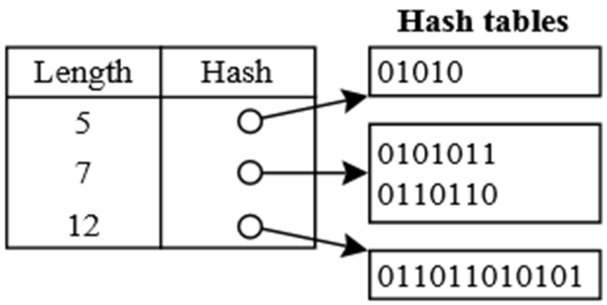


Figura 14 Estructura de almacenamiento en tablas hash con 5 prefijos.

Tomaremos como ejemplo la estructura que se muestra en la Figura 14. Como podemos ver tenemos un prefijo de longitud 5, dos de longitud 7 y solo uno de longitud 12. Cada uno de los prefijos se encuentra almacenado dentro de la tabla que les corresponde de acuerdo a su longitud.

Es posible plantear el pseudocódigo del proceso de búsqueda. Para esto definiremos algunos aspectos.

- ***L*** será el arreglo que contiene las longitudes y los apuntadores (**len** y **hash** respectivamente).
- ***D*** será la dirección IP a la que le estamos buscando el prefijo BMP.

A continuación se muestra el algoritmo que realiza las búsqueda lineal propuesta en (14).

```
////////////////////////////////////////////////////////////////////////

Function  Busqueda_Lineal(D)   /* Se busca el prefijo BMP de la dirección D */
Inicializar BMP a una cadena vacía;
i = Índice más grande en el arreglo L;
While ( BMP == NULL ) and ( i ≥ 0 )
      Extrae los primeros L[i].len bits de D y crea D´;
      BMP  =  Busca(D´,L[i].hash);  /*Busca  el  prefijo  D´  dentro  de  la  tabla
                                          apuntada*/
      i=i-1;
End While

////////////////////////////////////////////////////////////////////////
```

2.2.2 Búsqueda Binaria

Hemos revisado como la búsqueda lineal nos ofrece una solución inmediata para localizar un prefijo BMP dentro de tablas hash, sin embargo cuando tenemos una gran cantidad de prefijos almacenados, este algoritmo puede tornarse lento ya que en el peor de los casos si el prefijo BMP se encuentra en la tabla 1, el algoritmo habrá recorrido las otras 31 tablas antes de poder localizarlo. Por lo que se propone otro mecanismo para escoger las tablas en las que se busca el prefijo.

La propuesta plantea realizar una búsqueda binaria dentro de las tablas siguiendo el recorrido mostrado en la Figura 15, asumiendo que se tienen longitudes desde 1 hasta 32 para IPV4.

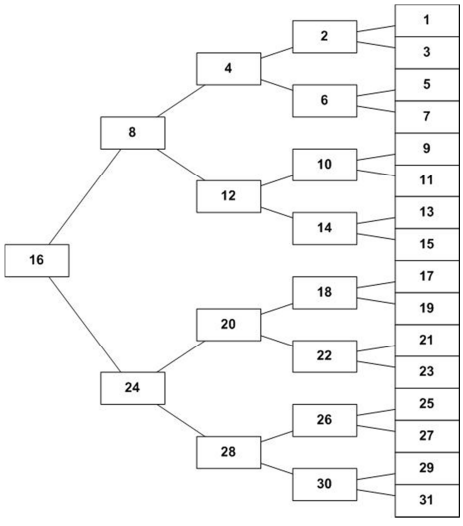


Figura 15 Recorrido binario de las tablas hash.

Supongamos que tenemos almacenados los prefijos P1= 0, P2=00 y P3=111. En la Figura 16 se muestra el diagrama de este ejemplo.

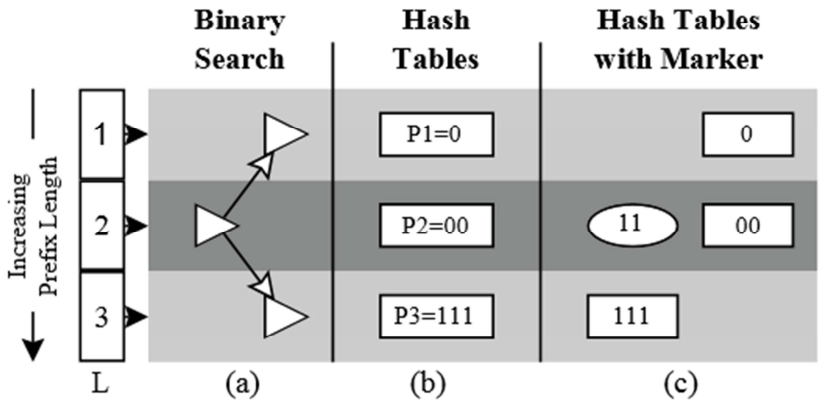


Figura 16 Ejemplo de búsqueda binaria.

Dentro de nuestra estructura tendríamos un total de 3 tablas ya que almacenamos prefijos de 3 longitudes distintas. Ahora realicemos la búsqueda del prefijo P3, al obedecer la técnica de la búsqueda binaria (Figura 16a) debemos buscar primero en la tabla de longitudes 2, sin embargo al no encontrarse el prefijo buscado debemos tomar la decisión de seguir buscando en las tablas de longitudes mayores o menores, por lo que surge la necesidad de introducir una "marca" que nos indique si el prefijo se encuentra en las tablas de longitudes mayores o no. En la Figura 16c se muestra la marca que debe ser agregada simbolizándola con un ovalo. Queda claro que al agregar esta serie de marcas se ve incrementado el número de prefijos almacenados, lo que nos lleva a la pregunta de ¿cuantas marcas son necesarias?

Una vista rápida a este esquema nos podría llevar a la idea de que si el prefijo a almacenar es de longitud ***L***, se debe dejar una marca en todas las tablas que contengan prefijos con longitudes menores a ***L***. Si hacemos un análisis más minucioso, nos daremos cuenta que solamente es necesario colocar marcas en las tablas desde las cuales se pueda llegar a ese prefijo, ya que se sigue el recorrido mostrado en la Figura 15.

En la Figura 17 se muestra en color rojo las tablas que contienen las marcas que deben ser dejadas por un prefijo de 21 o de 23 bits, como se puede ver, la posición de las marcas dependen directamente del mecanismo que se siga para elegir las tablas en que se buscará primero.

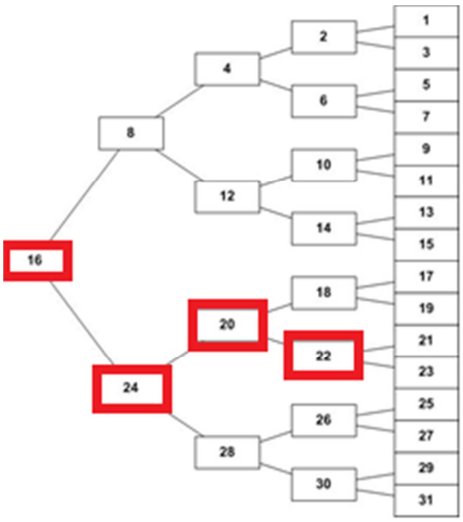


Figura 17 Marcas dejadas por un prefijo de 21 o 23 bits

Para más detalles sobre este mecanismo de almacenamiento podemos consultar la referencia (14).

2.3 Búsqueda por intervalos de prefijos

En (15) podemos encontrar una propuesta en la que se sugiere almacenar los prefijos de red dentro de una tabla, para después poder realizar una búsqueda binaria. Dentro de la búsqueda binaria se buscan coincidencias exactas por lo que al utilizarla para buscar prefijos de longitud variable, es necesario hacer algunas modificaciones para que este esquema funcione.

Para poder ilustrar de mejor forma este mecanismo, utilizaremos un ejemplo muy simple en el que tenemos direcciones de máximo 6 bits y solamente tendremos tres prefijos de red: 1*, 101* y 10101*. Como se mencionó anteriormente, la primera cuestión que debemos notar es que el algoritmo de búsqueda binaria no trabaja con cadenas de longitudes variables, por lo que mapeamos cada prefijo a direcciones de 6 bits completando los bits faltantes con ceros, quedando la tabla como se muestra en la Figura 18.

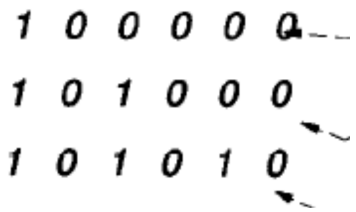


Figura 18 Prefijos adaptados a longitudes de 6 bits.

Una vez que hemos almacenado los prefijos de red, supongamos que tenemos que buscar los prefijos *BMP* de las tres direcciones siguientes: 101011, 101110 y 111110. Si se realiza la búsqueda binaria, esta fallaría debido a que ninguna de las direcciones buscadas se encuentra en la tabla. El apuntador de búsqueda terminaría al final de la tabla debido a que todas las direcciones buscadas son mayores que 101010, notemos que realmente a cada dirección buscada le corresponde un prefijo de la tabla, esto se muestra en la Figura 19.

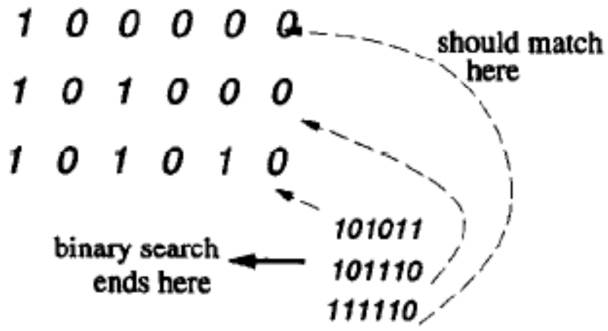


Figura 19 Posición donde termina la busqueda binaria y posición dónde deberia terminar.

Al analizar el ejemplo anterior nos damos cuenta que esencialmente existen dos problemas al realizar la búsqueda del prefijo BMP. El primer problema es que cuando finaliza la búsqueda, el apuntador termina muy lejos del prefijo correcto y el segundo problema es que al realizar búsquedas de direcciones a los que les corresponden prefijos distintos, el apuntador de todas ellas termina en la misma región de la tabla.

Para resolver la segunda problemática tomemos en cuenta el significado de un prefijo de red. Al escribir el prefijo 1* en realidad estamos escribiendo el rango de direcciones desde **1**00000 hasta **1**11111 por lo que cada prefijo puede ser escrito por dos direcciones completas (el inicio y el fin de su rango). Al incluir ambas direcciones de cada prefijo dentro de la tabla y ordenándolas, llegamos a la tabla que se muestra en la Figura 20, en la que se muestra cada inicio de rango conectado con su fin por medio de una línea.

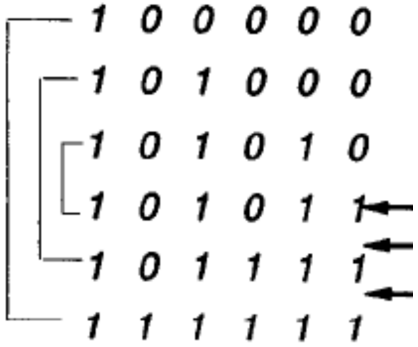


Figura 20 Tabla que incluye las direcciones de inicio y fin de rango de cada prefijo de red.

Si realizamos la búsqueda binaria dentro de este conjunto de direcciones, veremos que los apuntadores terminan en regiones distintas, de hecho, la búsqueda de la dirección 101011 termina en una coincidencia exacta, la búsqueda de la dirección **101110** termina en una falla en la región entre 101011 y 101111, y la búsqueda de la dirección **111110** termina en una falla en la región entre 101111 y 111111. (Figura 21)

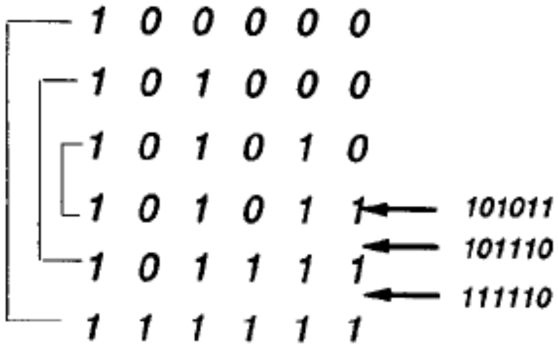


Figura 21 Las búsquedas de direcciones con prefijos BMP distintos, terminan en regiones distintas.

Para poder visualizar de mejor forma que la problemática 2 ha quedado resuelta, es decir, las búsquedas de direcciones con prefijos BMP distintos, terminan en regiones distintas, recurriremos a la siguiente figura.

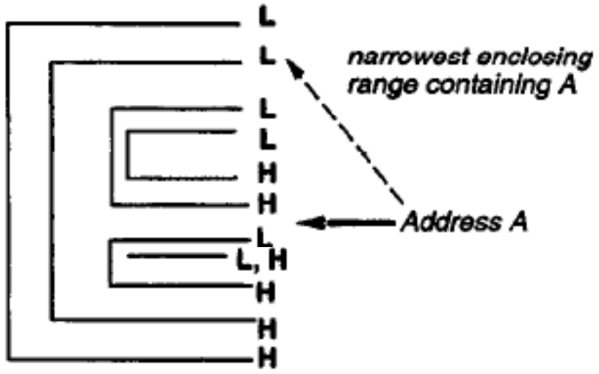


Figura 22 conjunto arbitrario de rangos de direcciones.

Imaginemos que tenemos un conjunto arbitrario de rangos de direcciones marcando sus correspondientes inicios (simbolizados con la letra **L**) y sus correspondientes fines (sombolizados con la letra **H**). Sin importar que dirección escojamos dentro de la tabla, siempre al realizar una busqueda lineal hacia la parte superior, la primer **L** que encontremos sin su correspondiente **H** nos indicará el prefijo BMP correcto. Con estas adaptaciones la problemática dos ha quedado resuelta, sin embargo desafortunadamente la problemática 1 aún no se ha solucionado.Hemos mencionado que la problemática 1 tiene que ver con que el apuntador de búsqueda termina muy lejos del prefijo BMP por lo que es necesario una búsqueda lineal de la primer **L** que no tiene su correspondiente **H**, sin embargo esto disminuiria la eficiencia del algoritmo de búsqueda ya terminaría siendo una búsqueda lineal, para subsanar esto, los autores proponen realizar un pre-computo la tabla de ruteo agregando algunos campos como se muestra en la Figura 23.

							>	=
P1)	1	0	0	0	0	0	P1	P1
P2)	1	0	1	0	0	0	P2	P2
P3)	1	0	1	0	1	0	P3	P3
	1	0	1	0	1	1	P2	P3
	1	0	1	1	1	1	P1	P2
	1	1	1	1	1	1	–	P1

Figura 23 Tabla de ruteo pre-computada agregando apontadores inmediatos.

Al tener la tabla pre-computada vemos como los prefijos de red (El inicio de los rangos) son marcados en la parte izquierda por las letras **P1, P2** y **P3**. En la parte derecha se agregan dos columnas, la columna marcada con el símbolo **>** indicará el prefijo que le corresponde si la

búsqueda termina y la dirección buscada es mayor a la señalada por el apuntador de búsqueda, y la columna **=** indicará el prefijo que le corresponde si la dirección buscada es igual a la señalada por el apuntador de búsqueda. En general este algoritmo termina teniendo en la búsqueda un orden de complejidad logarítmica ya que emplea la búsqueda binaria, sin embargo al realizar el pre-cómputo de la tabla de ruteo, se ve afectada la complejidad de inserción y de borrado, para mayores detalles de este algoritmo, consultar la referencia (15).

2.4 Almacenamiento en árboles Multibit

Al revisar el funcionamiento de los árboles binarios en el punto 2.1.1 vimos que para poder pasar de un nivel a otro solamente se necesita preguntar por el valor de un bit. Ahora para ver de mejor forma el funcionamiento de los árboles Multibit, definimos al *stride* como el número de bits que se tienen que comparar para pasar de un nivel a otro, podemos decir que los árboles binarios tienen un *stride* de un bit en todos sus niveles. Ahora si seguimos esta misma idea podríamos construir un árbol que posea *strides* de más de un bit. En la Figura 24 se muestra un árbol cuyos *strides* dependen del nivel.

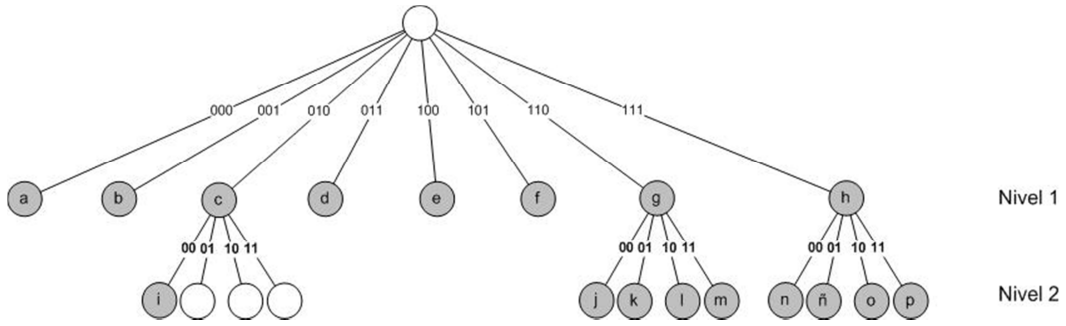


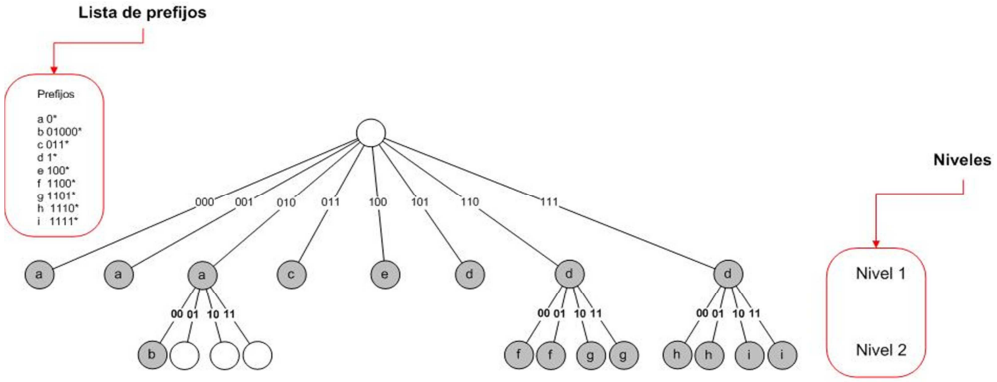
Figura 24 Árbol Trie-Multibit con *strides* variables.

En el árbol de la Figura 24 se puede apreciar que para poder llegar al nivel uno se tienen que comparar tres bits a la vez y dependiendo de los valores de cada uno de ellos se decide a cuál de las ramas se avanza, sin embargo para poder llegar del nivel uno al nivel dos se tienen que comparar solamente dos bits, a este tipo de árbol se le conoce como Trie-Multibit ya que no necesariamente poseen *strides* de un solo bit (8) (11). A los árboles Trie-Multibit se les pueden dar aplicaciones como el almacenamiento de prefijos de red ya que en cada nodo puede albergarse la interfaz de un prefijo, el prefijo de la interfaz almacenada es el camino de bits comparados para llegar al nodo. Es decir, si utilizamos el ejemplo de la Figura 24, el prefijo de la interfaz almacenada en el nodo **a** será 000, el prefijo de **b** será 001 y así sucesivamente, en el

segundo nivel el prefijo de la interfaz almacenada en el nodo **i** será 010 00, el de **j** será 110 00 y de igual forma sería si existieran más niveles. Si quisiéramos realizar una búsqueda de un prefijo almacenado dentro de esta estructura el número de comparaciones que tendríamos que hacer para poder encontrar la interfaz correspondiente dependerá de la longitud del *stride* del árbol, si suponemos que el árbol tiene un *stride* de k bits en cada nivel, la complejidad de la búsqueda será del orden de $O\left(\left\lceil\frac{w}{k}\right\rceil\right)$ donde w es la longitud del prefijo, esto se debe a que se comparan k bits a la vez para pasar de un nivel a otro. Al realizar el proceso de inserción o borrado de prefijos nos damos cuenta que tenemos que realizar $\left\lceil\frac{w}{k}\right\rceil$ comparaciones para ubicarnos en una posición adecuada dentro del árbol y una vez allí a lo más modificaremos 2^K elementos por lo que la complejidad de actualización es del orden de $O\left(\left\lceil\frac{w}{k}\right\rceil + 2^k\right)$. No obstante, existe un problema con la actualización de este tipo de estructura en el contexto de almacenamiento de prefijos de red, ya que en ocasiones pueden existir pérdidas de información, para poder comprender porque ocurre este problema es necesario conocer de manera más profunda la construcción de estos árboles.

2.4.1 Construcción de los árboles Trie-Multibit

Como hemos visto, un árbol Trie-Multibit se ajusta en gran medida para el almacenamiento de prefijos de red ya que con esta estructura se pueden implementar búsquedas de prefijos solamente preguntando por el valor de grupos de bits de la dirección IP destino. En esta sección se explica cómo se construye el árbol Trie-Multibit a partir de una tabla de ruteo dada. Para que se pueda construir el árbol primero debemos de contar con lo siguiente: una tabla de ruteo (es la que será almacenada en el árbol), definir cuántos niveles tendrá nuestro árbol y el número de bits que tendrán los *strides* de cada nivel. Las características principales de un árbol Trie-Multibit se muestran en la Figura 25.



El stride en el nivel 1 es igual a 3 debido a que se comparan 3 bits y en el nivel 2 el stride es de 2 bits.

Figura 25 Características principales de un árbol Trie-Multibit.

La longitud máxima de los prefijos que se pueden almacenar en un árbol Trie-Multibit es igual a la suma de los strides de todos los niveles, en el ejemplo de la Figura 25 las interfaces que se pueden almacenar son solamente las correspondientes a prefijos que tengan longitudes menores o iguales a 5 bits = (3 bits del nivel 1 + 2 bits del nivel 2), dentro de este conjunto de prefijos se pueden insertar directamente las interfaces de aquellos que tengan longitudes de tres y de cinco bits, el resto tendrán que ser modificados para que se ajusten a las dos longitudes mencionadas, a este proceso de ajuste se le conoce como expansión de prefijos.

2.4.2 Expansión de prefijos

Si vemos la lista de prefijos del ejemplo de la Figura 25 podemos ver que varios de los prefijos no coinciden con las longitudes de tres y cinco bits, estos prefijos tuvieron que ser ajustados a dichas longitudes. Para ilustrar este proceso tomemos como ejemplo al prefijo de la interfaz **a** el cual es 0*, este prefijo tiene una longitud de 1 bit y debe ser ajustado a 3 o 5 bits, para no invadir las secciones de almacenamiento de otros niveles, elegimos el valor más cercano (tres bits), al llevar a cabo este ajuste se dice que el prefijo es "expandido" a tres bits. Para esto se aumentan los bits faltantes y se toman todas las posibles combinaciones entre los bits aumentados, dándonos como resultado en este caso cuatro prefijos distintos **0**00, **0**01, **0**10 y **0**11, por lo que estos cuatro nodos en un principio tienen guardada la interfaz **a**, a medida que el resto de los prefijos se insertan en el árbol vemos que el prefijo correspondiente a la interfaz **c** (**011**) tiene 3 bits y coincide con uno de los prefijos expandidos de **a**, en este caso se dice que el nodo ha sido capturado por un prefijo original y este tiene prioridad sobre uno expandido, por lo que la interfaz que almacenará el nodo es la **c** y no **a**. En la Tabla 4 se muestra cada uno de los prefijos que se expandieron y la asignación de los nodos para el ejemplo de la Figura 25. (8)

Tabla 4 Tabla de expansión de prefijos a 3 o a 5 bits.

Prefijo de red	**Prefijos expandidos**
0*	**0**00 (a) expandido a **0**01 (a) 3 bits **0**10 (a)
01000* (capturado)	**01000** (b) nodo capturado
011* (capturado)	**011** (c) nodo capturado
1*	**1**01 (d) expandido a

	110 (d) **1**11 (d)	3 bits
100* (capturado)	**100** (e)	nodo capturado
1100*	**1100**0 (f) **1100**1 (f)	expandido a 5 bits
1101*	**1101**0 (g) **1101**1 (g)	expandido a 5 bits
1110*	**1110**0 (h) **1110**1 (h)	expandido a 5 bits
1111*	**1111**0 (i) **1111**1 (i)	expandido a 5 bits

Una vez que los prefijos han sido expandidos a las longitudes deseadas, se procede a insertarlos de manera directa en el nodo que les corresponda dentro del árbol tomando en cuenta que los prefijos capturados tienen prioridad sobre los prefijos expandidos.

En la Figura 26 se muestra el árbol Trie-Multibit de dos niveles (el primer nivel con un *stride* de 3 bits y el segundo con 2 bits) después de insertar todos los prefijos de la Tabla 4.

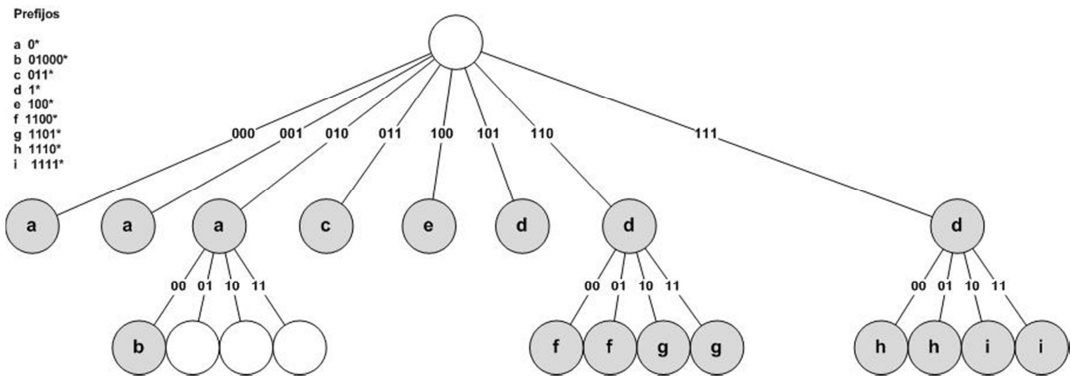


Figura 26 Árbol Trie-Multibit después de la inserción de todos los prefijos.

Ahora, si suponemos que nos llega un paquete de información con la dirección destino del ejemplo 2.

130.86.16.66 = 10000010 01010110 00010000 01000010

Al realizar la búsqueda dentro del árbol de la Figura 26 primero comparamos el número de bits equivalente al *stride* del nivel uno el cual es igual a tres, al comparar los primeros tres bits de la dirección destino avanzamos a la rama identificada con los bits 100 y debido a que este nodo ya no tiene hijos se dice que el prefijo BMP es 100 y que la interfaz de salida correspondiente a esta dirección destino es la interfaz **e**.

2.5 Algoritmo de la universidad de Lulea

En (16) y (17) sedetalla el algoritmo de la universidad de Lulea, esta toma como base a los árboles Trie-Multibit pero con algunas particularidades:

- Un nodo con hijos NO puede almacenar una interfaz de red.
- Todos los nodos sin hijos deben tener interfaz de red.

Una vez dadas estas especificaciones podemos tomar al árbol de la Figura 26 y ajustarlo para que cumpla con lo pedido. En la Figura 27 se puede observar que las interfaces que se encontraban en los nodos con hijos han sido retiradas y los nodos sin hijos que se encontraban vacíos han sido llenados con la interfaz que poseía el padre.

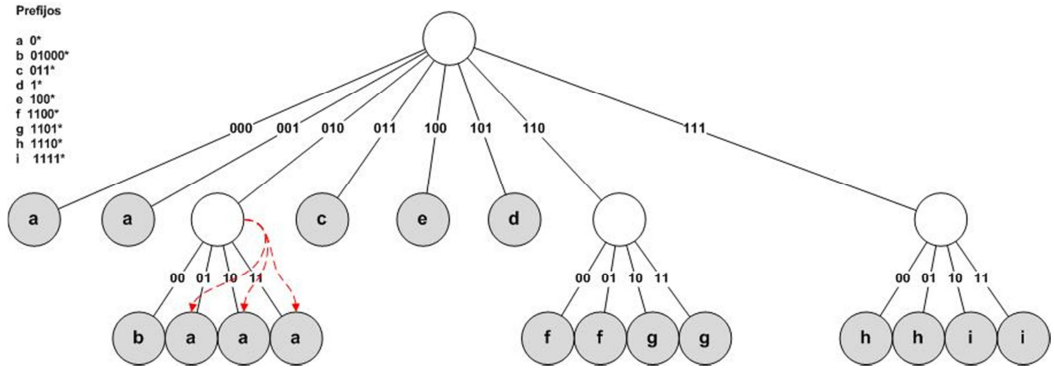


Figura 27 Árbol Trie-Multibit modificado para cumplir las especificaciones de la construcción del algoritmo de lulea.

Ahora que tenemos el árbol que cumple con las especificaciones pedidas, necesitamos ver el árbol como un conjunto de arreglos, en nuestro ejemplo el árbol se verá como un arreglo de 2^3elementos y tres arreglos de 2^2elementos, esto se ve de una manera más clara en la Figura 28.

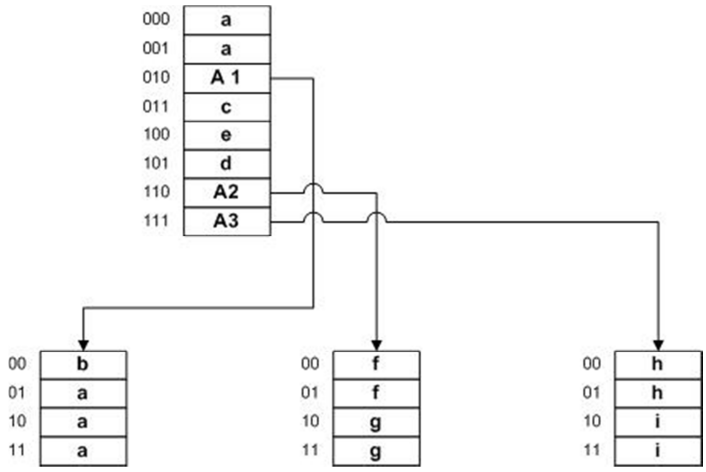


Figura 28 Árbol Trie-Multibit visto como arreglos.

En la Figura 28 se puede observar al árbol de la Figura 27 visto como un conjunto de arreglos interconectados entre ellos, en esta figura los elementos A1, A2 y A3 son apuntadores hacia arreglos. El siguiente paso será tomar cada uno de los arreglos y aplicar un proceso de compresión, para mostrar este proceso tomaremos únicamente el arreglo más grande, cabe mencionar que este proceso es el mismo para todos los arreglos.

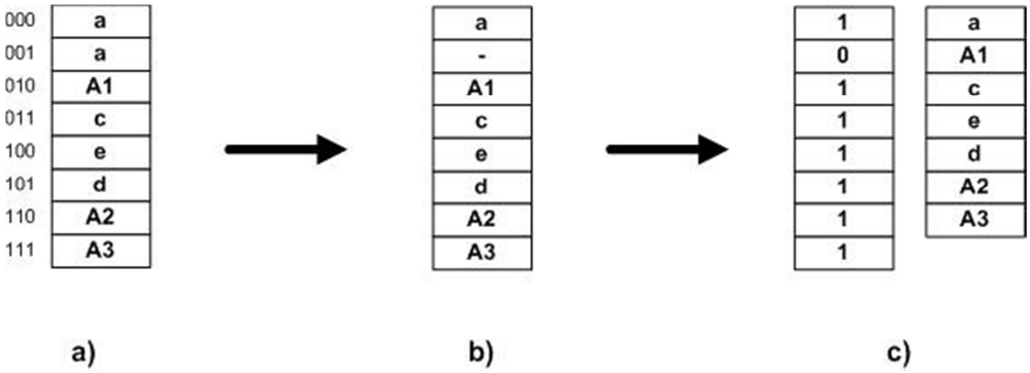


Figura 29 a) Arreglo raíz del árbol mostrado en la figura 14. b) Arreglo sin los elementos repetidos c) Arreglo sin repeticiones y mapa de bits generado.

En la Figura 29-a vemos el arreglo que comprimiremos, para esto el primer paso es detectar las interfaces que se repiten. Como se puede ver, solo se repite la interfaz '**a'** en las localidades **000** y **001**, por lo que a partir de esta información construiremos un mapa de bits que nos diga cuando se repiten los elementos del arreglo. El mapa de bits tendrá tantos bits como elementos tenga el arreglo original (con repeticiones), lo primero es recorrer el arreglo original de arriba hacia abajo y cuando encontremos una interfaz distinta a la anterior pondremos un '1' en el mapa de bits y cuando encontremos un elemento repetido pondremos un '0'. Una vez que

tenemos el mapa de bits conocemos cuando se repiten las interfaces, por lo que ya no necesitamos el arreglo con repeticiones, así que las eliminamos del arreglo original (ver Figura 29-b), después de este proceso tendremos como resultado dos arreglos, el mapa de bits y el arreglo de elementos sin repeticiones, estos son los que se muestran en la Figura 29-c.

En la Figura 30 podemos ver el resultado después de haber aplicado el proceso de compresión sobre todos los arreglos del árbol.

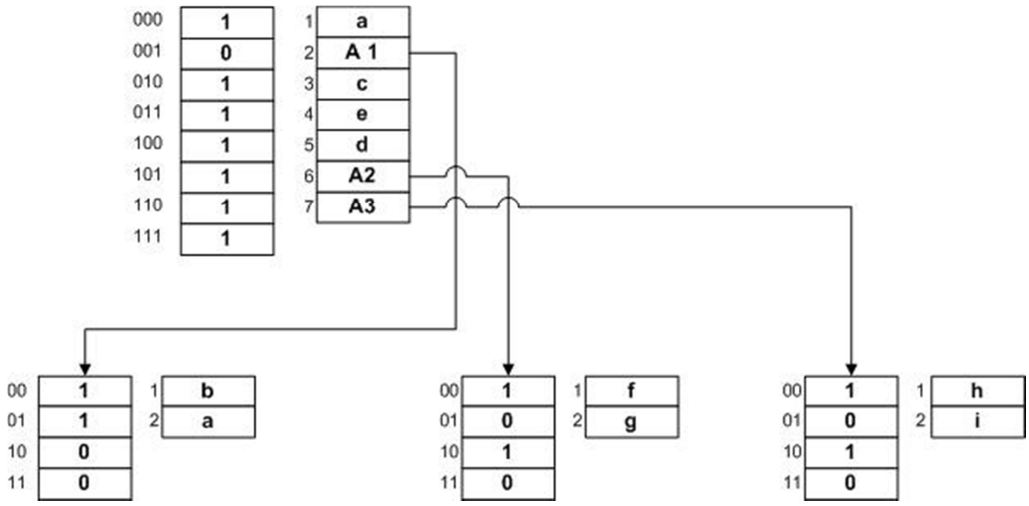


Figura 30 Árbol de la Figura 28 después haberle aplicado el proceso de compresión.

Una vez que tenemos construida esta estructura sólo resta mostrar el procedimiento de búsqueda de un elemento a partir de una dirección IP destino. Supongamos que nos llega un paquete con una dirección destino como la que se muestra:

218.86.16.66 = 11011010 01010110 00010000 01000010

Debido a que el *stride* del primer nivel es igual a tres, primero comparamos los primeros tres bits los cuales son **110**, ahora buscamos en el mapa de bits del primer nivel y contamos el número de 'unos' almacenados en el arreglo desde la localidad **000** hasta la localidad **110**, incluyendo ambas. Como se puede observar entre las localidades **000** y **110** existen seis 'unos' almacenados, por lo que ahora buscamos el elemento de la localidad seis del arreglo de elementos sin repeticiones, en esta localidad encontramos el elemento **A2** el cual es un apuntador a otro mapa de bits, por lo que repetimos el procedimiento con los siguientes bits y el siguiente mapa de bits.

Al repetir el procedimiento tomamos los siguientes dos bits de la dirección IP destino ya que el *stride* del siguiente nivel es igual dos, los siguientes bits son **11** así que contamos el número de 'unos' que se encuentran almacenados desde la localidad **00** hasta la **11** en arreglo apuntado por **A2,** debido a que se encuentran dos 'unos', nos fijamos en el contenido de la localidad dos del arreglo de elementos correspondiente y debido a que el elemento almacenado en la localidad dos ya no es un apuntador sino la interfaz **g,** podemos decir que la interfaz de salida para un paquete con la dirección destino 218.86.16.66 es la interfaz **g.**

El algoritmo de la universidad de Lulea propone construir las tablas a partir de un árbol Trie-Multibit de tres niveles con *strides* de 16 para el primer nivel, 8 para el segundo y 8 para el tercero, es decir en el primer nivel nuestro mapa de bits tendrá $2^{16} = 65536$ elementos en el segundo nivel tendrá $2^8 = 256$ y en el tercero $2^8 = 256$.

2.6 Desempeño de los esquemas presentados

Como se ha visto, a lo largo de este capítulo se han planteado distintas estructuras de datos para el almacenamiento de la tabla de ruteo, cada una de las propuestas ofrece distintos órdenes de complejidad en sus operaciones básicas. En la Tabla 5 se muestran las complejidades de los algoritmos de búsqueda y actualización de cada uno los esquemas presentados, así como la cota superior de memoria, donde *w* representa la longitud del prefijo, N es el número de prefijos almacenados y *k* es el *stride* del árbol utilizado.

Tabla 5 Complejidad de los algoritmos de actualización y búsqueda de los esquemas presentados.

	Esquema de almacenamiento	Complejidades		Cota superior de Memoria
		Actualización	Búsqueda	
1	Árboles binarios	$O(w)$	$O(w)$	$O(Nw)$
2	Arboles de ruta reducida	$O(w)$	$O(w)$	$O(N)$
3	Tablas Hash	$O(\mathrm{N}\log_2 \mathrm{w})$	$O(\log_2 \mathrm{w})$	$O(\mathrm{N}\log_2 \mathrm{w})$
4	Intervalos de prefijos	$O(N)$	$O(\log_2 \mathrm{N})$	$O(N)$
5	Arboles Trie-Multibit	$O\left(\left\lceil\frac{W}{k}\right\rceil + 2^k\right)$	$O\left(\left\lceil\frac{W}{k}\right\rceil\right)$	$O\left(N\left\lceil\frac{w}{k}\right\rceil 2^k\right)$
6	Algoritmo de la universidad de Lulea	-	$O\left(\left\lceil\frac{W}{k}\right\rceil\right)$	$O\left(N\left\lceil\frac{w}{k}\right\rceil 2^k\right)$

Como se puede ver en la Tabla 5, los esquemas que utilizan el almacenamiento en arboles binarios y árboles de ruta reducida presentan una complejidad, en sus algoritmos de búsqueda y actualización, que dependen exclusivamente de la longitud del prefijo, por lo que mientras

mayor sea la longitud del prefijo, mayor será el número de accesos a memoria que se realizarán para poder completar la tarea del algoritmo. En el esquema de almacenamiento que utiliza tablas hash y el denominado "intervalo de prefijos" se puede ver que la complejidad de los algoritmos tienen una dependencia de la variable N, la cual suele ser del orden de cientos de miles, por lo que las operaciones de actualización y búsqueda realizarán más accesos a memoria que los esquemas uno y dos. Los esquemas cinco y seis, están basados en árboles Trie-Multibit, por lo que dependen directamente de la longitud del prefijo y del *stride* del árbol, estos algoritmos son los que presentan las búsquedas más rápidas. Sin embargo, el algoritmo de actualización que se utiliza en el esquema cinco, presenta algunos inconvenientes ya que no garantiza la integridad de los prefijos que se almacenan, este punto será abordado de manera más detallada en el capítulo 3 ya que este trabajo está fuertemente ligado a este punto.

Capítulo 3

3 Trie Multibit con respaldo

En este capítulo analizaremos la estructura que se propone en este trabajo, al revisar el estado del arte nos encontramos con los árboles Multibit, que se caracterizan por poder controlar la velocidad del algoritmo búsqueda, variando el *stride* del árbol. Sin embargo esta estructura por si sola, en incapaz de realizar actualizaciones de elementos (borrado e inserción) garantizando la integridad de la información, esto será revisado en la sección siguiente.

3.1 Problemas en la actualización de los árboles Multibit

En la sección 2.4 se describen las técnicas que se siguen para poder construir un árbol Trie-Multibit, sin embargo una vez construido se presenta una dificultad al actualizar el árbol, es decir al insertar o borrar prefijos. La problemática esencialmente radica en que existen casos en los que se puede perder información almacenada en el árbol. Para ilustrar esto podemos utilizar el caso que hemos venido manejando (ejemplo de la Figura 25). Supongamos que se quieren insertar al árbol de la Figura 25 los prefijos e interfaces siguientes **j 101***, **k 110*** y **l 111***, aplicando el procedimiento de inserción nos damos cuenta que los nodos correspondientes a estos prefijos ya se encuentran ocupados por expansiones del prefijo correspondiente a la interfaz **d**, pero como ya habíamos dicho, un prefijo original tiene prioridad sobre uno expandido por lo que éstas expansiones son sustituidas, después de haber insertado las interfaces **j**, **k** y **l** el árbol quedaría como se muestra en la Figura 31.

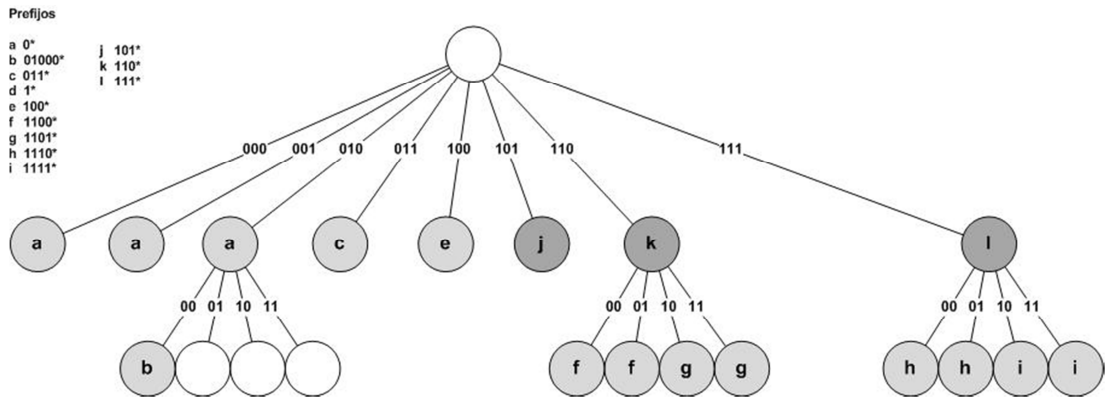


Figura 31 Árbol Trie-Multibit de la Figura 25 después de agregar las interfaces j, k y l.

Como se puede apreciar en el árbol de la Figura 31 la información correspondiente a la interfaz **d** ha desaparecido por completo, hasta este punto no habría problema alguno, sin embargo ¿Qué pasa si después de realizar las inserciones extras decidimos borrar la interfaz **j 101**? .Aquí es en donde notamos que éste nodo quedaría vacio y no podríamos recuperar la interfaz **d** debido a que ya no se encuentra en el árbol. Al observar el caso anterior podemos darnos cuenta que las interfaces de los prefijos que tienen que ser expandidos para insertarse dentro del árbol, son propensas a ser eliminadas al insertar y borrar otras interfaces. Para que los árboles Trie-Multibit puedan ser utilizados en el almacenamiento de prefijos es necesario que sean capaces de actualizarse sin que existan pérdidas de información. En este capítulo se plantea una posible solución a esta problemática detectada.

La solución que proponemos para que los árboles Trie-Multibit puedan implementar operaciones de inserción y de borrado sin pérdida de información, consiste en agregar una sección de respaldo de prefijos, con esto la estructura será capaz de recordar los prefijos que fueron sobre escritos, aunque se traten de prefijos con baja prioridad (expandidos).

3.2 Recordar prefijos originales

La solución que proponemos en esta tesis es la de no olvidar de manera total los prefijos e interfaces que han sido insertados en el árbol, aún si han sido reescritos dentro de los nodos, es decir, dejarlos almacenados por si se presenta el caso en el que se necesite recuperar alguno. Para esto se propone que el árbol cuente con una estructura de almacenamiento en cada uno de los nodos, esta estructura contendrá cada prefijo (sin expandir) que existe en los nodos hijos correspondientes, esto se ilustra de una mejor manera en la Figura 32.

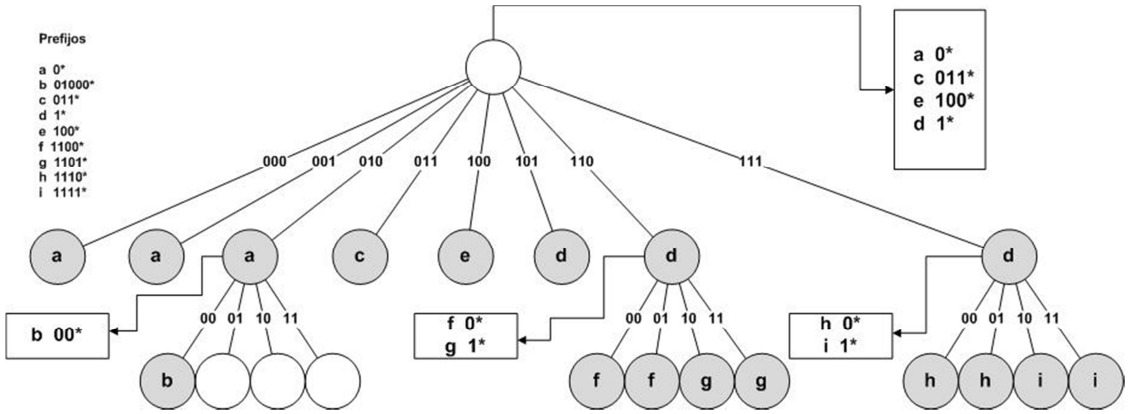


Figura 32 Modificación propuesta a la estructura de datos Trie-Multibit para respaldo de prefijos.

Con esta estructura auxiliar es posible recuperar las interfaces si es que se llegan a perder en el proceso de actualización. De esta manera el procedimiento de borrado tiene que incluir una parte en la que se consulte la estructura correspondiente y se determine cuál es la interfaz que debe almacenarse en lugar de la que ha sido borrada. El proceso de inserción por otra parte, tiene que incluir una sección en la que además de insertar la interfaz dentro del árbol, también debe insertarla en la estructura de respaldo correspondiente. Para poder hacer un análisis más riguroso de cómo esta modificación afecta las complejidades de búsqueda y actualización del árbol Trie-Multibit, es necesario definir qué tipo de estructura de datos se usará en la sección de respaldo.

3.3 Respaldo de prefijos en árboles binarios

Debido a que la estructura de respaldo debe ser accesada constantemente, debemos cuidar que la estructura de datos que nos va a servir de respaldo pueda consultarse rápidamente además de que también pueda actualizarse de igual forma, la estructura que se propone para la tabla de respaldo es la de árboles binarios ya que su complejidad de búsqueda y actualización son del orden de $O(w)$ donde w es la longitud del prefijo que se busque, almacene o borre del árbol binario. En la Figura 33 se muestra cómo se realizaría el respaldo en árboles binarios para el ejemplo de la Figura 32.

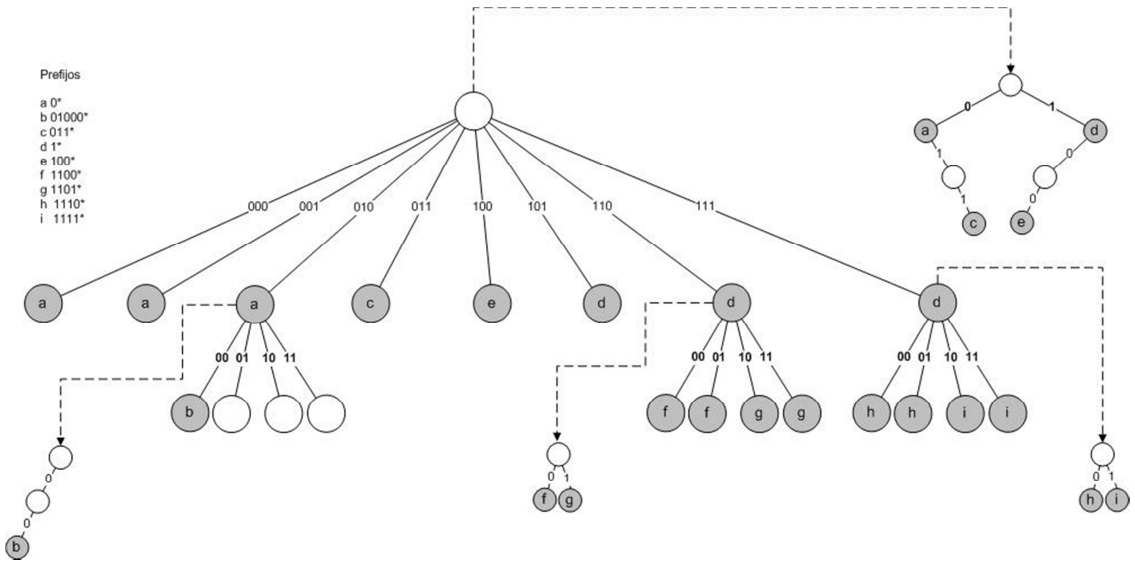


Figura 33 Árbol Trie-Multibit con respaldo de interfaces en árboles binarios.

Una vez que hemos decidido la estructura que utilizaremos en el respaldo de prefijos debemos observar cómo son afectadas las complejidades de búsqueda y actualización del árbol Trie-Multibit. En este análisis tomaremos un árbol Trie-Multibit con un *stride* fijo de *k* bits en todo el árbol. En el caso de la búsqueda, la complejidad no se ve afectada ya que si buscamos un prefijo de longitud *w* nos tomará $\left\lceil \frac{w}{k} \right\rceil$ comparaciones para localizarlo, ya que en este proceso no interviene la estructura de respaldo, a continuación se explica el proceso de borrado e inserción en la estructura Trie-Multibit con respaldo.

3.4 Borrado e inserción

En el proceso de borrado de un prefijo en el árbol Trie-Multibit con respaldo sigue los siguientes pasos:

1. Se busca la interfaz a borrar *X* dentro del árbol Trie-Multibit.
2. Se borra de la estructura de respaldo correspondiente la interfaz *X* y se obtiene la interfaz sustituta *Y*.
3. Los nodos correspondientes a las expansiones del prefijo de la interfaz *X* son reescritos con información del sustituto *Y*.

Por lo que podemos deducir la complejidad del algoritmo de borrado donde la operación fundamental es el acceso al árbol ya sea para lectura o escritura. El punto uno del algoritmo se refiere a una búsqueda la cual tiene una complejidad de $\left\lceil \frac{w}{k} \right\rceil$ como ya habíamos visto. El número de operaciones que se harán en el punto número dos dependerá de la longitud del prefijo a

borrar. Debido a que para pasar de un nivel a otro es necesario ir comparando k bits, las estructuras de respaldo podrán almacenar prefijos de a lo más **k** bits, es decir los árboles binarios de respaldo podrán crecer tanto como el *stride* del nivel lo permita. El punto número tres toma en cuenta la modificación de los nodos correspondientes a las expansiones del prefijo de la interfaz a borrar, en el peor de los casos tendremos 2^k nodos que modificar. Por lo que el orden de complejidad del algoritmo de borrado es de $O\left(\left\lceil\frac{w}{k}\right\rceil + k + 2^k\right)$.

Una vez hecho el análisis del algoritmo de borrado podemos seguir con el análisis del algoritmo de inserción. En el proceso de inserción de un prefijo en el árbol Trie-Multibit modificado se desarrolla con los siguientes pasos:

1. Se busca la posición dentro del árbol donde se insertará la nueva interfaz *X*.
2. Se inserta la interfaz *X* en la estructura de respaldo.
3. Se insertan las expansiones del prefijo de la interfaz X.

Ahora podemos deducir la complejidad del algoritmo de inserción, donde nuestra operación fundamental al igual que en el algoritmo de borrado es el acceso al árbol ya sea para lectura o escritura. El punto uno del algoritmo se refiere a una búsqueda la cual tiene una complejidad de $\left\lceil\frac{w}{k}\right\rceil$, como ya habíamos visto. El número de operaciones que se harán en el punto número dos dependerá de la longitud del prefijo a insertar, debido a que para pasar de un nivel a otro es necesario ir comparando k bits y considerando que en cada nivel encontraremos estructuras de respaldo en cada nodo, podemos decir que el elemento a insertar en la estructura de respaldo será de a lo más **k** bits. El punto número tres toma en cuenta la modificación de los nodos correspondientes a las expansiones del prefijo a insertar, en el peor de los casos tendremos 2^k nodos que modificar. Por lo que el orden de complejidad del algoritmo de inserción es de $O\left(\left\lceil\frac{w}{k}\right\rceil + k + 2^k\right)$. Como podemos ver los órdenes de complejidad de inserción y de borrado coinciden por lo que podemos hablar de un orden de complejidad de actualización.

3.5 Consumo de memoria

Es necesario hacer un análisis de la cantidad de memoria que se utiliza en la estructura que estamos proponiendo. Podemos iniciar observando la cantidad de memoria que utiliza la estructura Trie-Multibit simple, el número de nodos que utiliza esta estructura dependerá de la cantidad de interfaces almacenadas, de la longitud de los prefijos de cada interfaz y de la longitud del *stride* del árbol ocupado. Ahora, si suponemos que utilizamos un árbol Trie-Multibit con un *stride* de *k* bits y queremos almacenar N interfaces tomando en cuenta que sus prefijos correspondientes tienen una longitud máxima de w, con esto podemos deducir la cantidad de nodos que serán ocupados para construir este árbol. Es necesario recordar que para poder

localizar cada una de las interfaces en el árbol debemos seguir un camino de $\left\lceil \frac{w}{k} \right\rceil$ nodos y que cada nodo potencialmente puede tener hasta 2^k hijos, esto nos lleva a deducir que la cota superior de memoria es del orden de $O\left(N\left\lceil \frac{w}{k} \right\rceil 2^k\right)$. Debido a que nuestra propuesta básicamente aumenta una sección de respaldo al árbol Trie-Multibit clásico es lógico que observemos un aumento en el consumo de memoria, el número de localidades necesarias para el respaldo de las interfaces dependerá de cuántas interfaces hay que respaldar y de la longitud máxima de los árboles de respaldo, ya que tenemos que respaldar N interfaces y los árboles binarios de respaldo pueden tener hasta k niveles, podemos decir que por cada interfaz respaldada, sin importar el árbol de respaldo en el que se encuentre, necesitará como máximo un camino de k nodos por lo que necesitaremos Nk localidades extras. Este análisis nos lleva a concluir que la cota superior de memoria para nuestra propuesta es de $O\left(N\left\lceil \frac{w}{k} \right\rceil 2^k + Nk\right)$. Para fines de comparación es conveniente hacer este mismo análisis para el almacenamiento de prefijos con el esquema clásico de árboles binarios. En el caso de los árboles binarios solo hace falta recordar que si se tienen que almacenar N interfaces y sus prefijos tienen como longitud máxima w bits; Por cada interfaz almacenada necesitaremos, para llegar a ellas, un camino con una longitud máxima de w bits, esto nos lleva a que la cota máxima de memoria para el almacenamiento en árboles binarios es del orden de $O(Nw)$.

3.6 Ordenes de complejidad y cotas superiores

En la

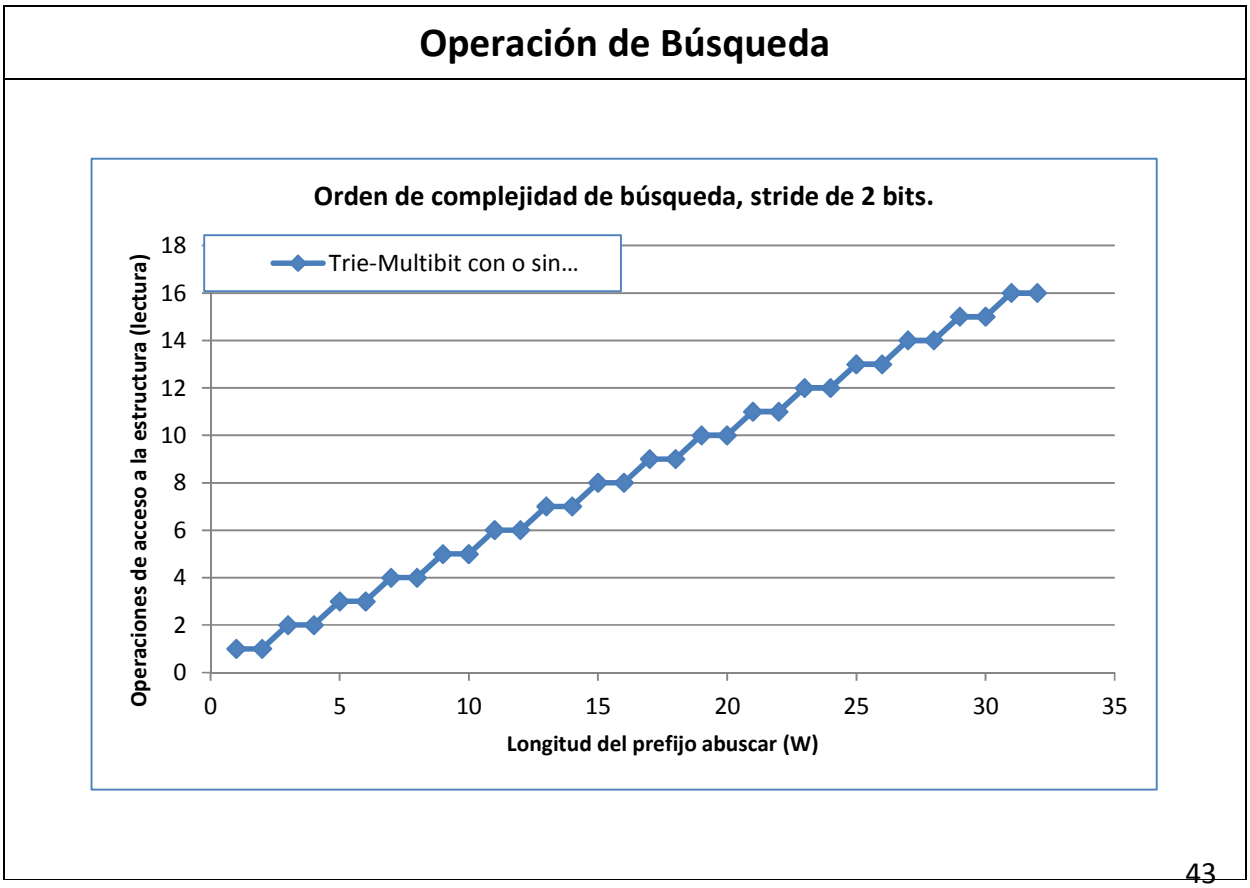
Tabla 6 se muestra un resumen de los órdenes de complejidad de búsqueda para las estructuras de almacenamiento en árboles Trie-Multibit clásicos, en árboles Trie-Multibit con respaldo en árboles binarios y los árboles binarios simples. En la operación de actualización dinámica solamente se muestran los órdenes de complejidad de las estructuras de almacenamiento en árboles Trie-Multibit con respaldo en árboles binarios y el almacenamiento en árboles binarios simples, esto se debe a que estas dos estructuras garantizan que al realizar una inserción o borrado de alguna interfaz no se pierda información vital de la tabla de ruteo. En la misma tabla se muestran las cotas superiores de consumo de memoria para las tres estructuras, para poder hacer un análisis de las tendencias de los órdenes, es conveniente graficar los resultados presentados en la tabla.

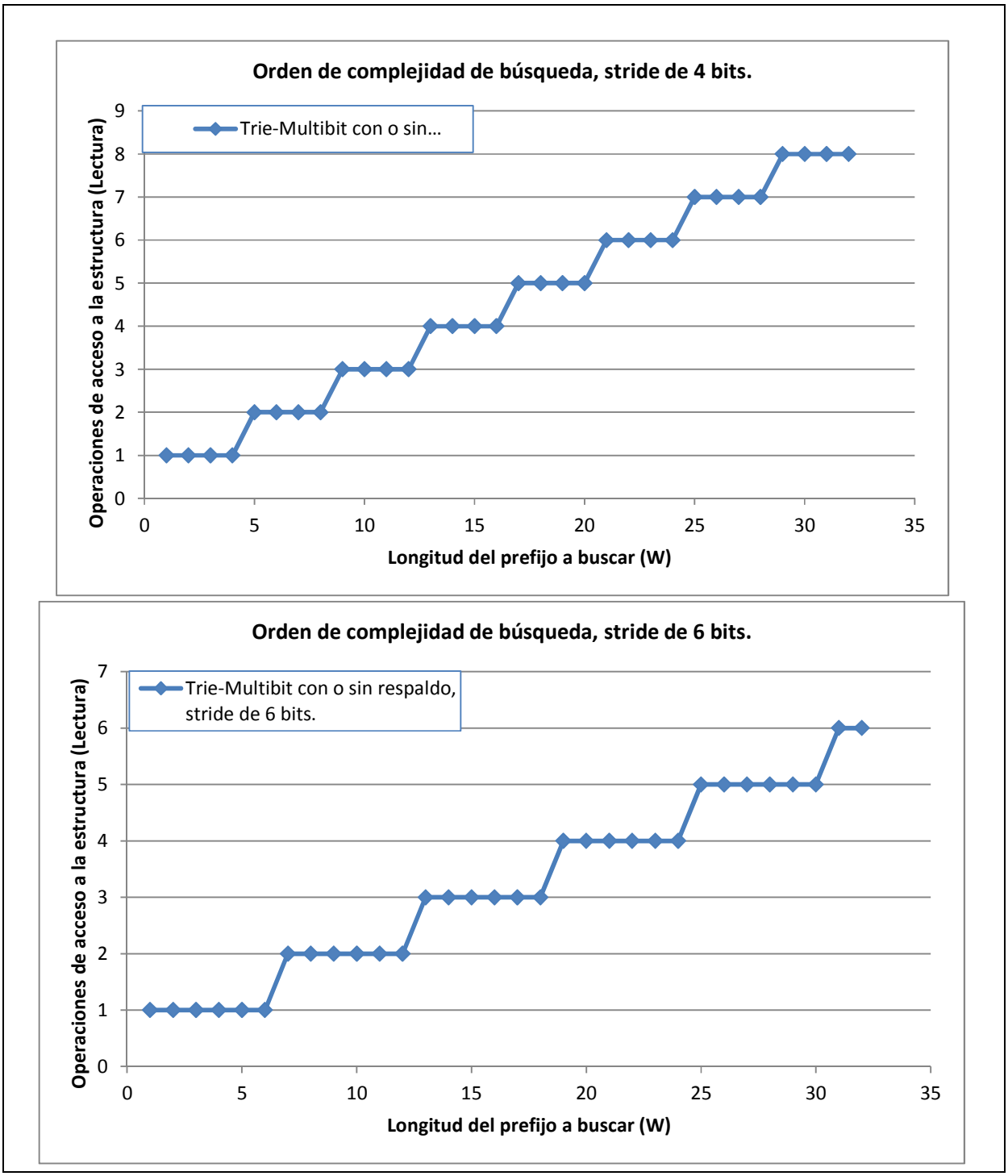
Tabla 6 Comparación de los órdenes de complejidad y cotas superiores de memoria para el árbol Trie-Multibit, el árbol Trie-Multibit con respaldo en árboles binarios y el esquema clásico de almacenamiento en árboles binarios.

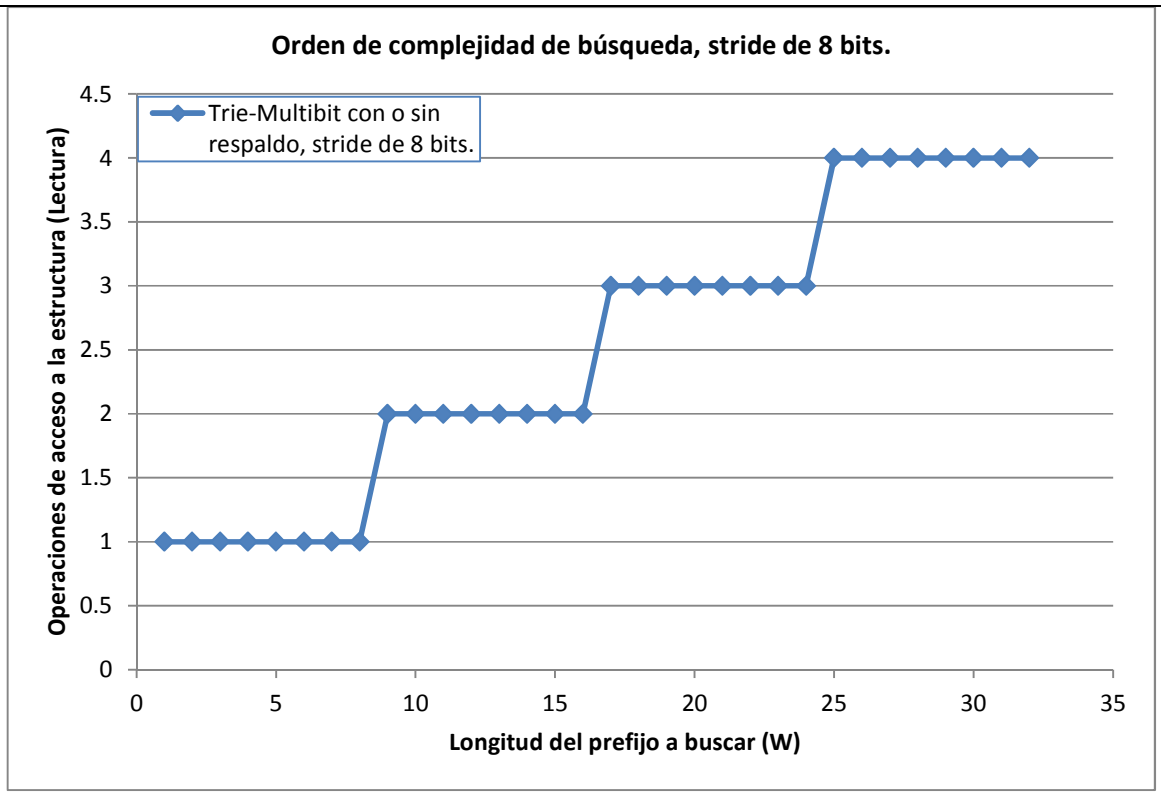
	Complejidades		Cota superior de memoria
	Búsqueda	**Actualización dinámica**	
Trie-Multibit	$O\left(\left\lceil\frac{W}{k}\right\rceil\right)$	No existe	$O\left(N\left\lceil\frac{W}{k}\right\rceil 2^k\right)$
Trie-Multibit con respaldo en árboles binarios	$O\left(\left\lceil\frac{W}{k}\right\rceil\right)$	$O\left(\left\lceil\frac{W}{k}\right\rceil + 2^k + k\right)$	$O\left(N\left\lceil\frac{W}{k}\right\rceil 2^k + Nk\right)$
Árboles binarios	$O(w)$	$O(w)$	$O(Nw)$

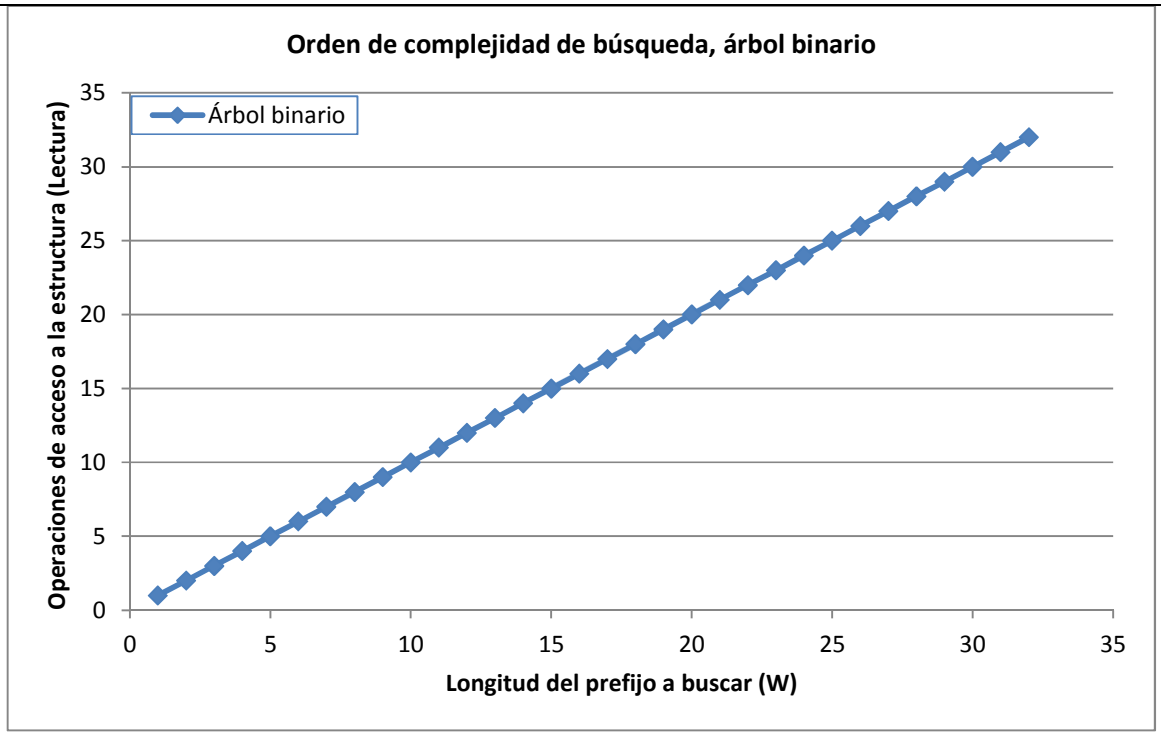
En la Tabla 7 se muestran las gráficas de los órdenes de complejidad para la operación de búsqueda utilizando las estructuras de almacenamiento de árboles Trie-Multibit simples, árboles Trie-Multibit con respaldo en árboles binarios y el almacenamiento clásico en árboles binarios.

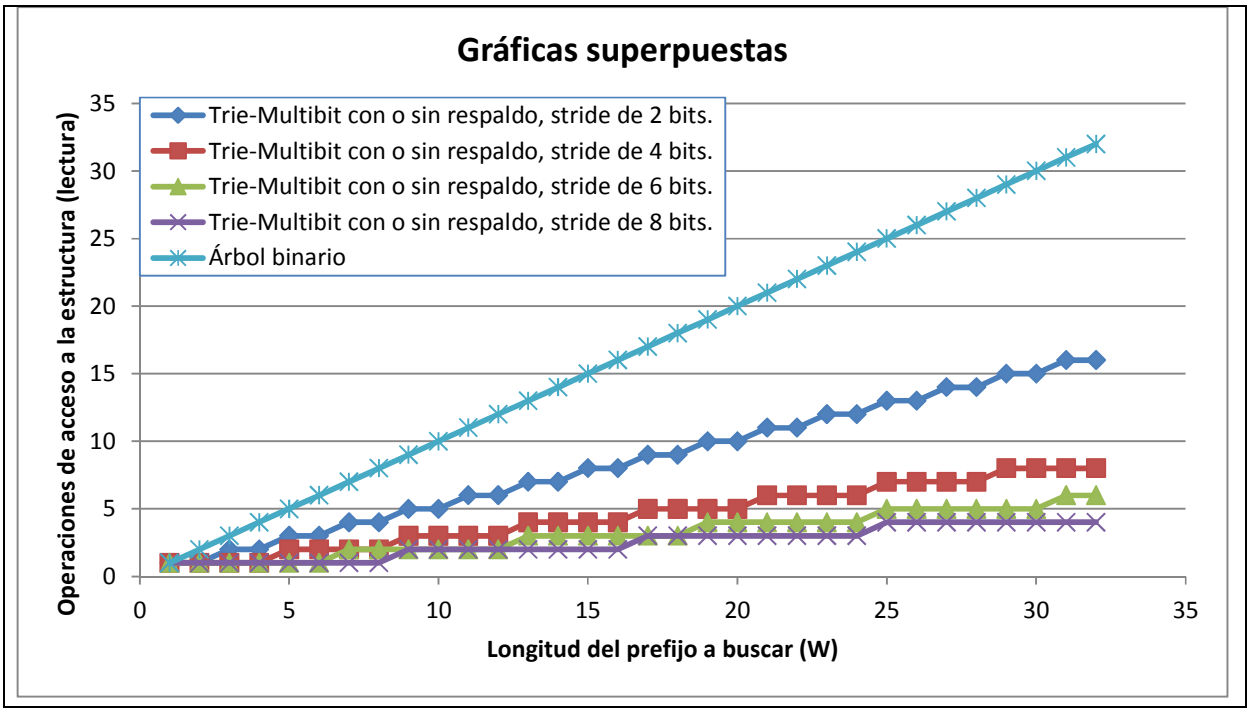
Tabla 7 Órdenes de complejidad para operación de búsqueda.




Orden de complejidad de búsqueda, stride de 4 bits.
Trie-Multibit con o sin...
Operaciones de acceso a la estructura (Lectura)
Longitud del prefijo a buscar (W)
0
1
2
3
4
5
6
7
8
9
0
5
10
15
20
25
30
35
Orden de complejidad de búsqueda, stride de 6 bits.
Trie-Multibit con o sin respaldo, stride de 6 bits.
Operaciones de acceso a la estructura (Lectura)
Longitud del prefijo a buscar (W)
0
1
2
3
4
5
6
7
0
5
10
15
20
25
30
35


Orden de complejidad de búsqueda, stride de 8 bits.
Trie-Multibit con o sin respaldo, stride de 8 bits.
Operaciones de acceso a la estructura (Lectura)
Longitud del prefijo a buscar (W)
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
0
5
10
15
20
25
30
35

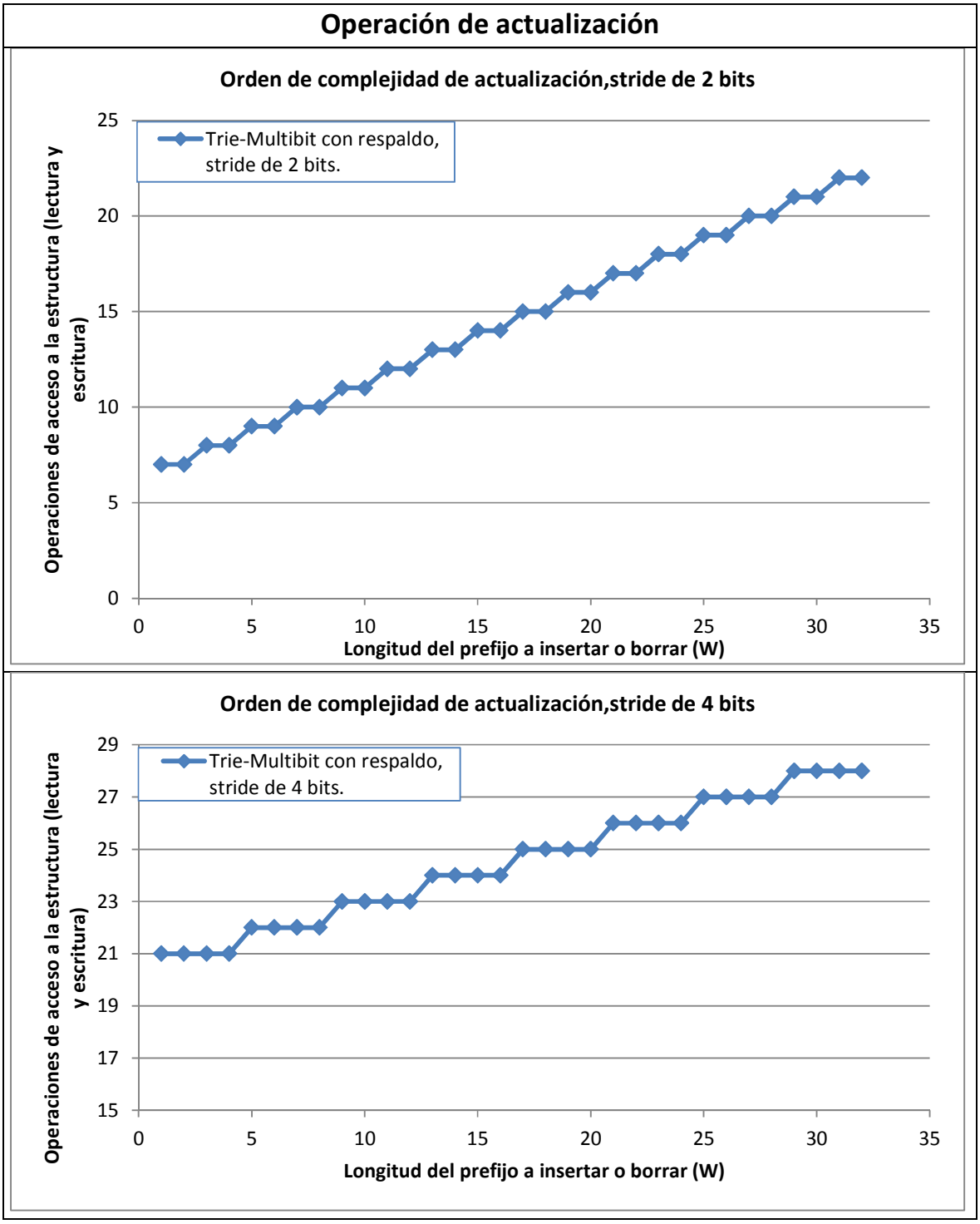

Orden de complejidad de búsqueda, árbol binario
Árbol binario
Operaciones de acceso a la estructura (Lectura)
Longitud del prefijo a buscar (W)
0
5
10
15
20
25
30
35

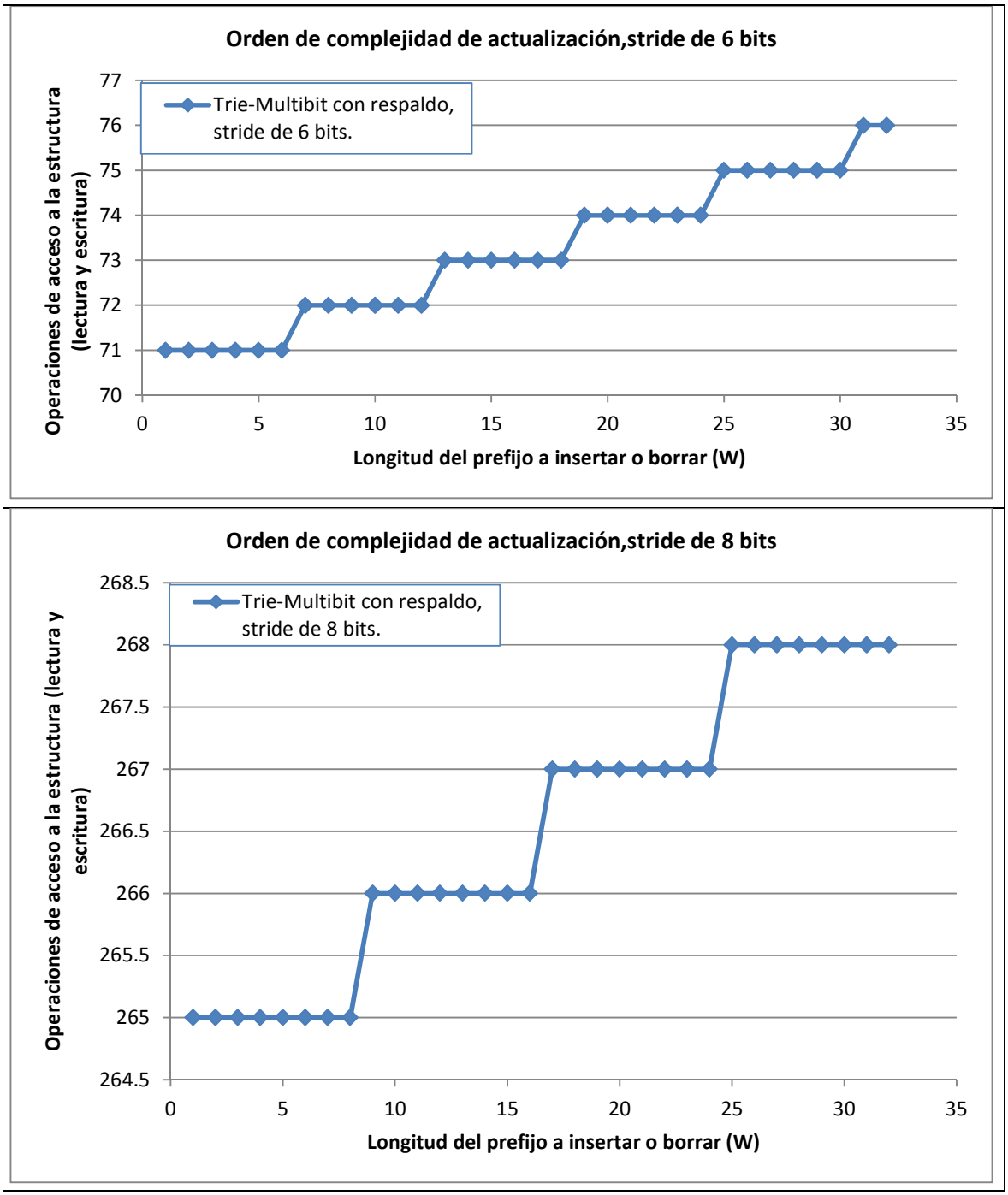


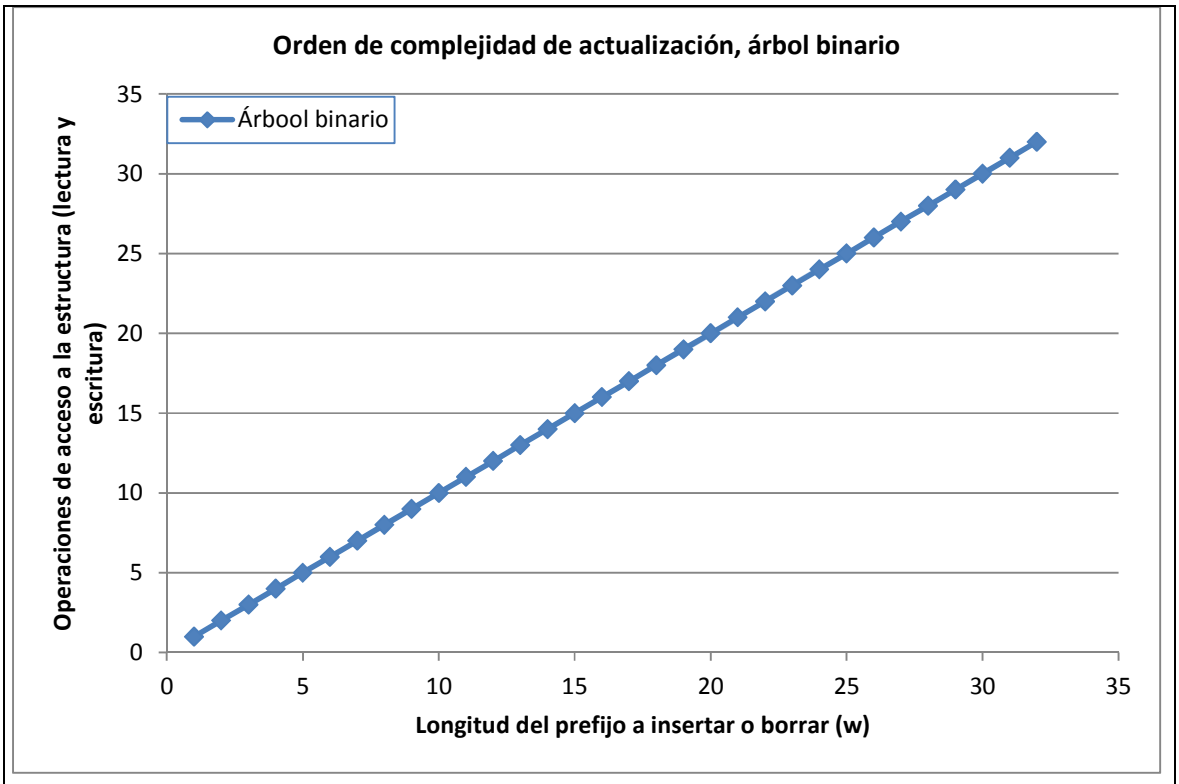
En la Tabla 7 se muestra como para las distintas longitudes del prefijo buscado, la operación de búsqueda en los árboles Trie-Multibit, realiza menor número de accesos a memoria que la operación de búsqueda que utiliza el esquema de almacenamiento en árboles binarios. A medida que el *stride* del árbol va aumentando de dos hasta ocho bits, el número de accesos a memoria cada vez es menor y si asumimos que el acceso a la estructura ya sea para lectura o escritura es la operación que consume más tiempo, podemos decir que la búsqueda es cada vez más rápida a medida que aumenta el *stride* del árbol.

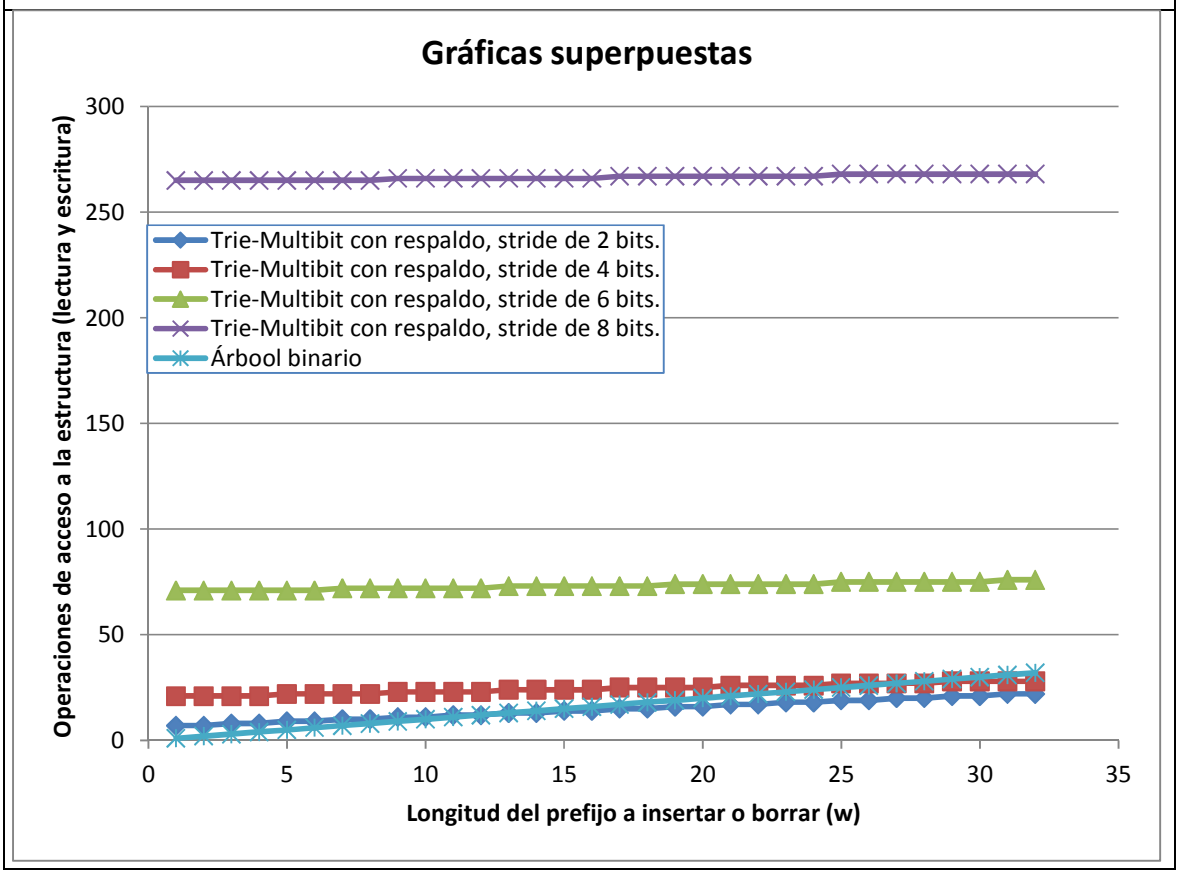
En la Tabla 8 se comparan las operaciones de actualización dinámica bajo el esquema de almacenamiento en árboles Trie-Multibit con respaldo y el almacenamiento en árboles binarios simples. Se puede ver que el algoritmo de actualización del árbol binario realiza menos accesos a la estructura, que el mismo algoritmo basado en Trie-Multibit con respaldo, a medida que el *stride* del árbol aumenta, la operación de actualización realiza más accesos a la estructura, esto se debe a que cada nodo puede tener hasta 2^k hijos, así que mientras más grande sea el *stride* (k) será necesario actualizar más nodos, por lo que la operación de actualización es más rápida cuando el *stride* es menor.

Tabla 8 Complejidad del algoritmo de actualización .




Orden de complejidad de actualización,stride de 6 bits
Trie-Multibit con respaldo, stride de 6 bits.
Operaciones de acceso a la estructura (lectura y escritura)
77
76
75
74
73
72
71
70
0
5
10
15
20
25
30
35
Longitud del prefijo a insertar o borrar (W)
Orden de complejidad de actualización,stride de 8 bits
Trie-Multibit con respaldo, stride de 8 bits.
Operaciones de acceso a la estructura (lectura y escritura)
268.5
268
267.5
267
266.5
266
265.5
265
264.5
0
5
10
15
20
25
30
35
Longitud del prefijo a insertar o borrar (W)

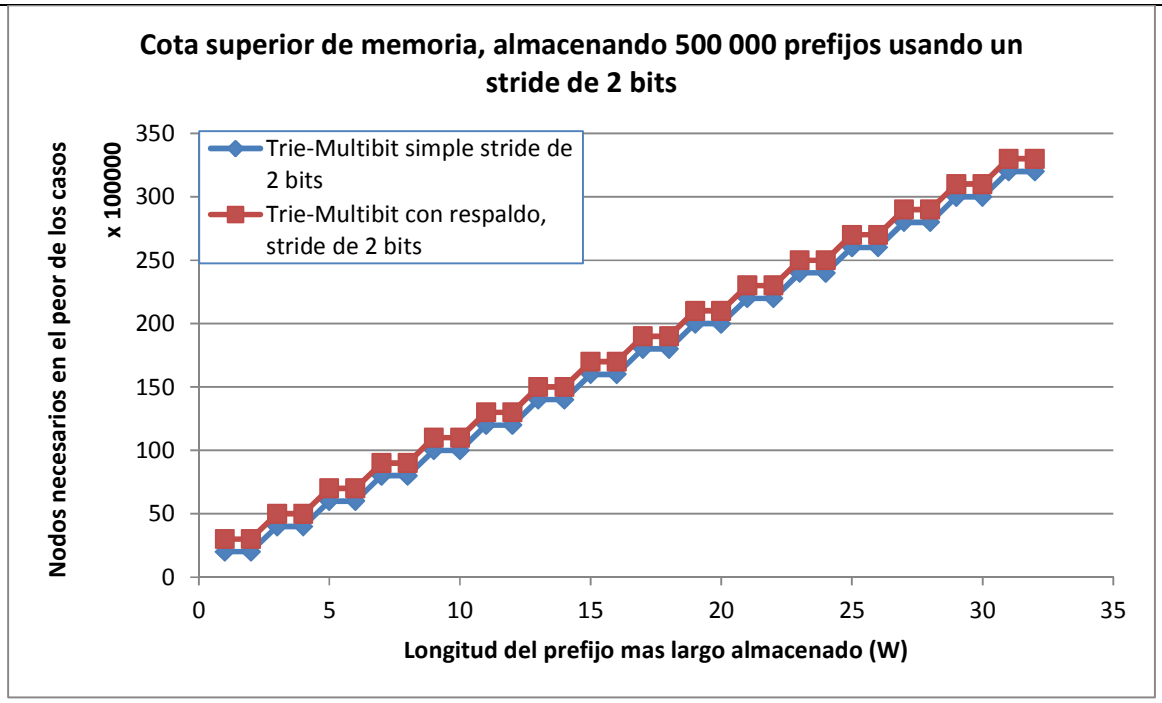

Orden de complejidad de actualización, árbol binario
Árbool binario
Operaciones de acceso a la estructura (lectura y escritura)
Longitud del prefijo a insertar o borrar (w)
0
5
10
15
20
25
30
35

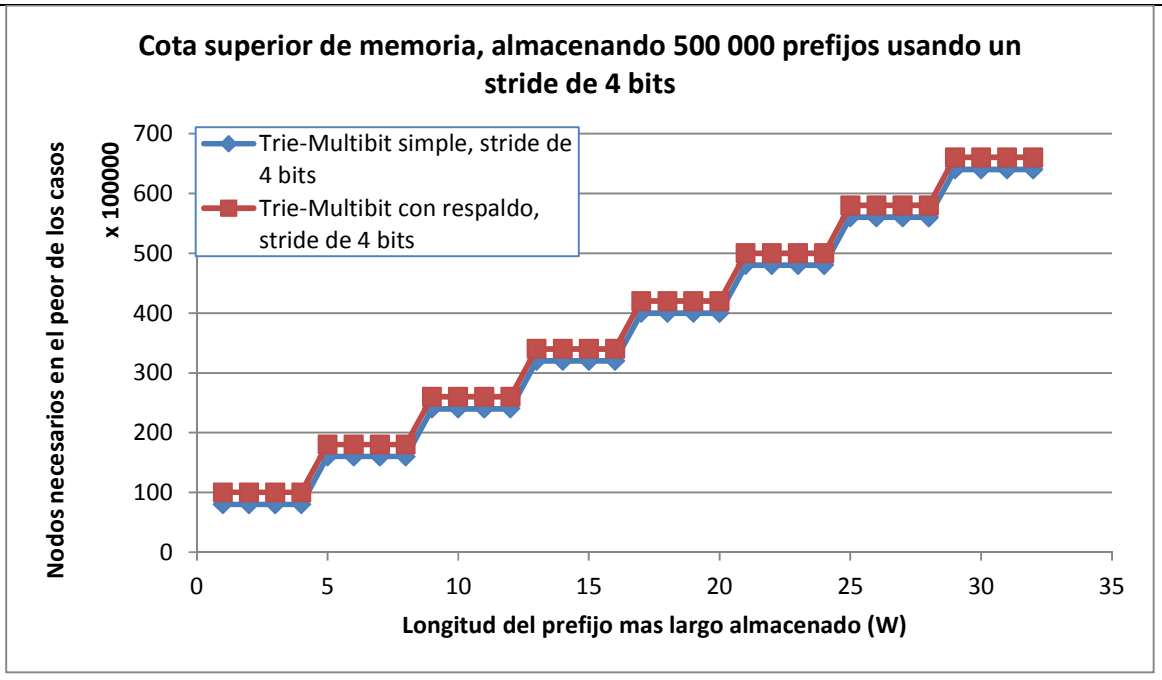

Gráficas superpuestas
Trie-Multibit con respaldo, stride de 2 bits.
Trie-Multibit con respaldo, stride de 4 bits.
Trie-Multibit con respaldo, stride de 6 bits.
Trie-Multibit con respaldo, stride de 8 bits.
Árbool binario
Operaciones de acceso a la estructura (lectura y escritura)
Longitud del prefijo a insertar o borrar (w)
0
50
100
150
200
250
300
0
5
10
15
20
25
30
35

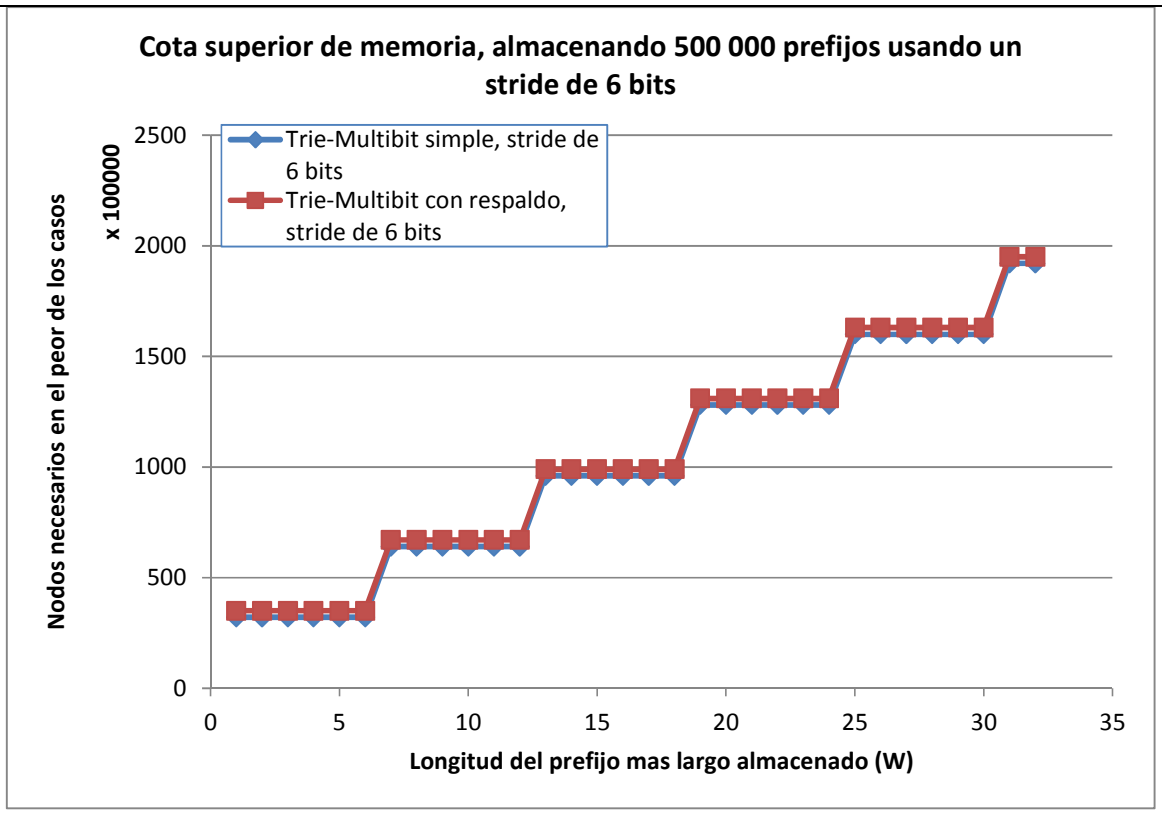
En la Tabla 9 se muestran las cotas superiores de consumo de memoria para los esquemas de almacenamiento en árboles Trie-Multibit simples, almacenamiento en árboles Trie-Multibit con respaldo en árboles binarios y almacenamiento en árboles binarios simples.

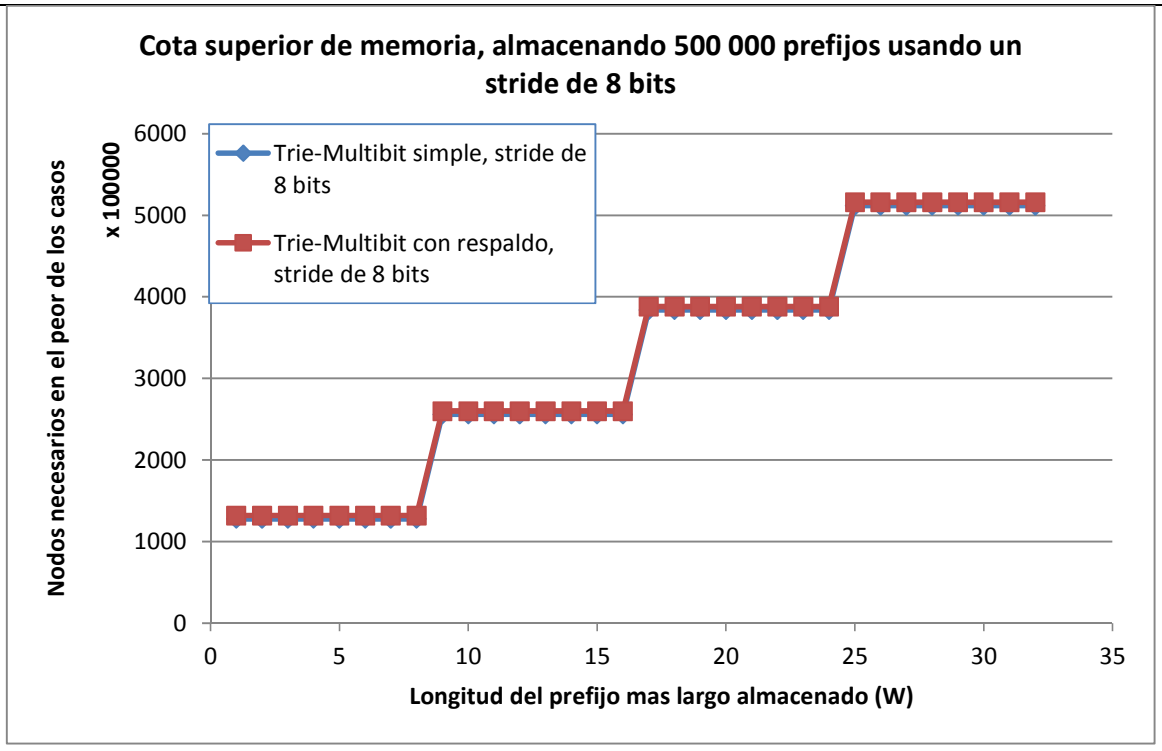
Tabla 9 Cotas de memoria superiores para distintos esquemas.

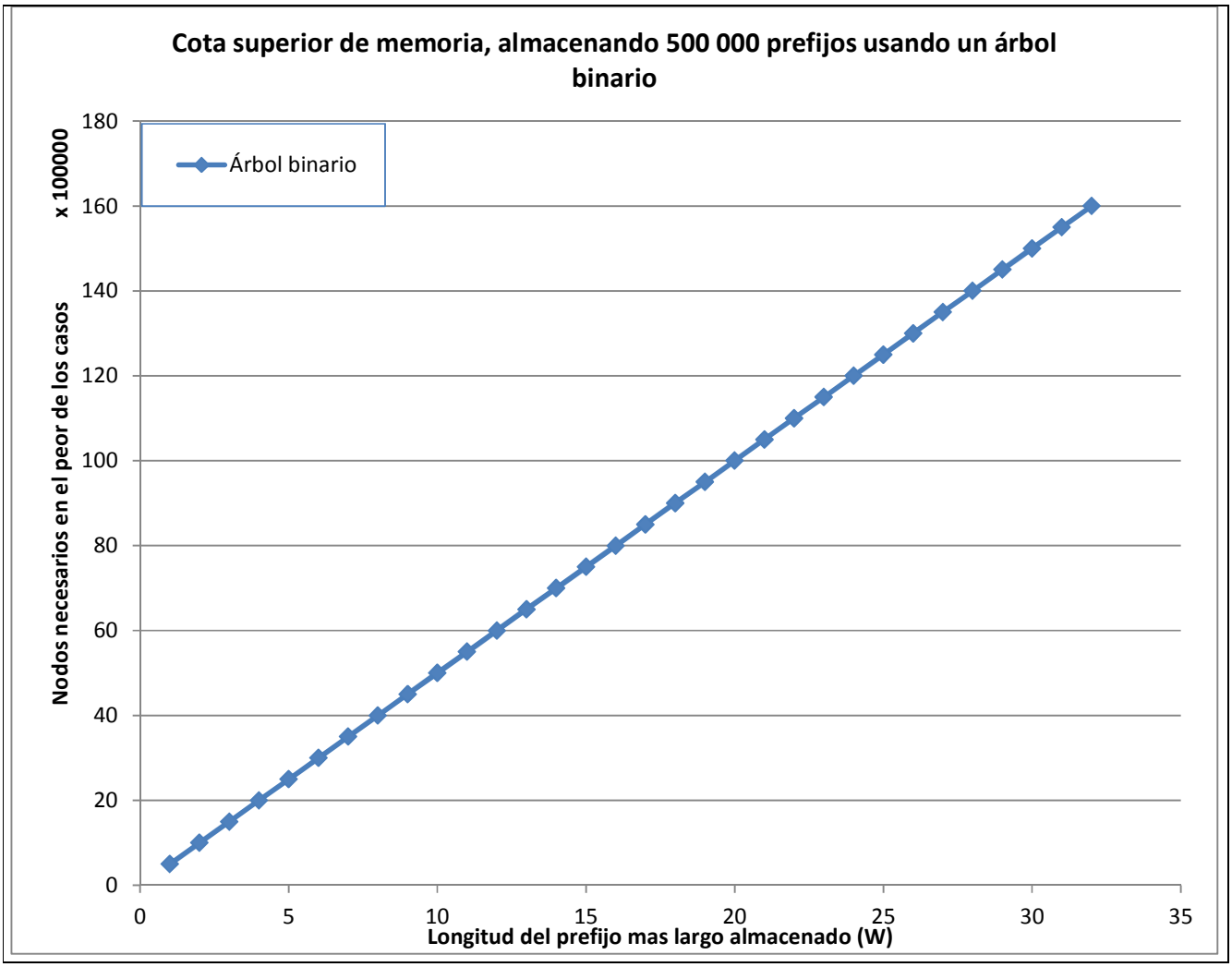
COTAS DE MEMORIA

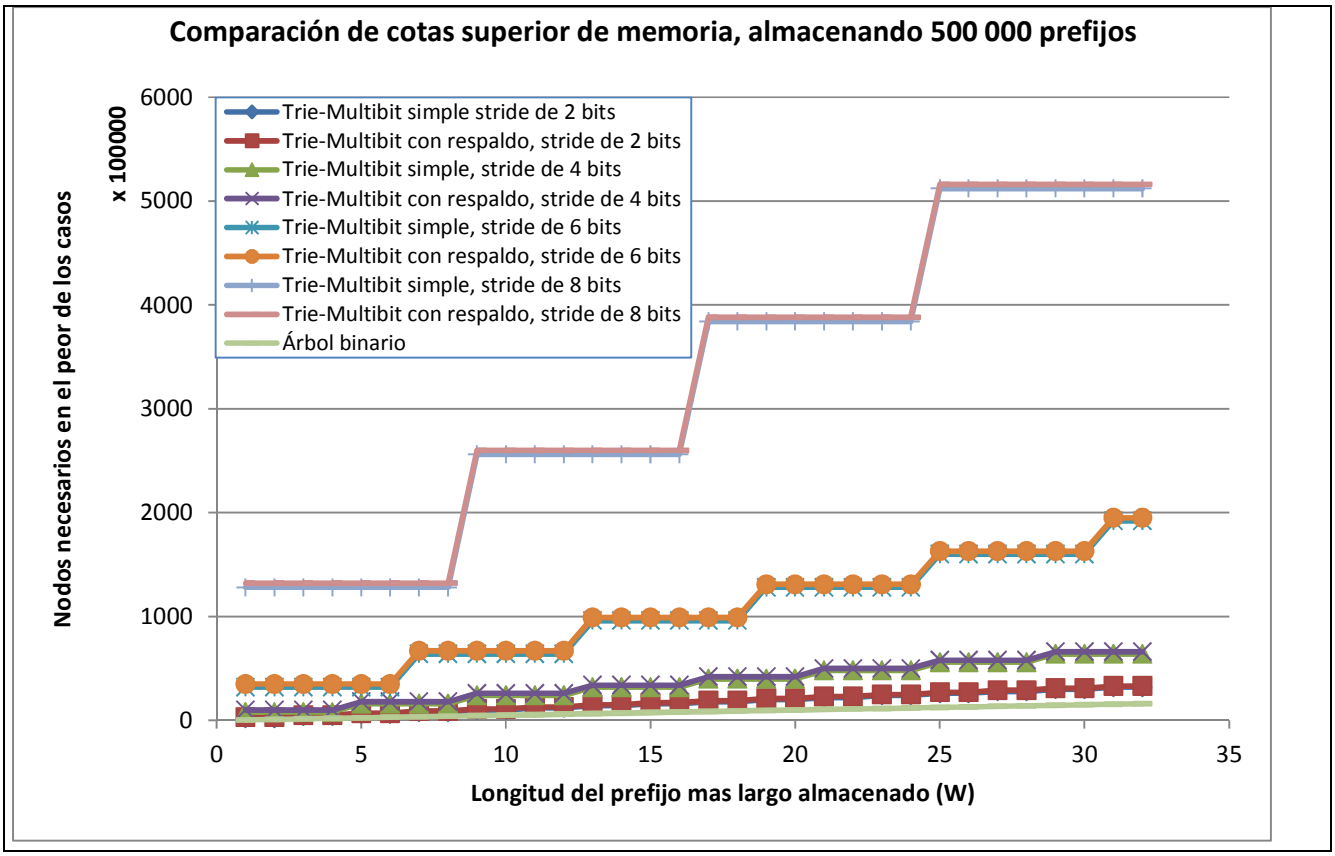





Cota superior de memoria, almacenando 500 000 prefijos usando un stride de 6 bits
Nodos necesarios en el peor de los casos x 100000
Trie-Multibit simple, stride de 6 bits
Trie-Multibit con respaldo, stride de 6 bits
2500
2000
1500
1000
500
0
0
5
10
15
20
25
30
35
Longitud del prefijo mas largo almacenado (W)


Cota superior de memoria, almacenando 500 000 prefijos usando un stride de 8 bits
Nodos necesarios en el peor de los casos x 100000
Trie-Multibit simple, stride de 8 bits
Trie-Multibit con respaldo, stride de 8 bits
6000
5000
4000
3000
2000
1000
0
0
5
10
15
20
25
30
35
Longitud del prefijo mas largo almacenado (W)


Cota superior de memoria, almacenando 500 000 prefijos usando un árbol binario
Árbol binario
x 100000
Nodos necesarios en el peor de los casos
180
160
140
120
100
80
60
40
20
0
0
5
10
15
20
25
30
35
Longitud del prefijo mas largo almacenado (W)



Para realizar las gráficas de la Tabla 9 se asume que la tabla almacenada consta de 500 000 prefijos, esto se debe a que actualmente esta es la longitud promedio de las tablas de ruteo en el núcleo de internet (Figura 1)también se puede ver claramente que los esquemas que utilizan árboles Trie-Multibit pueden llegar a consumir más memoria a medida que el *stride* del árbol aumenta y que el esquema de almacenamiento en árboles binarios tiene el menor consumo. Podemos notar que existe una diferencia relativamente pequeña, en cuanto a consumo de memoria, entre los esquemas de almacenamiento en árboles Trie-Multibit simples y Trie-multibit con respaldo, esto se debe a que los árboles Trie-Multibit con respaldo necesitan más memoria precisamente para guardar el respaldo de los prefijos e interfaces.

Para poder obtener una expresión matemática que nos indique la cantidad de memoria extra necesaria, es necesario analizar la relación que existe entre la memoria necesaria para implementar un árbol Trie-Multibit sin respaldo $\left(N\left\lceil\frac{w}{k}\right\rceil 2^k\right)$ y la memoria necesaria para almacenar el dicho respaldo (Nk).

A continuación se muestra la expresión que describe el porcentaje de memoria extra que se necesita para implementar un almacenamiento en arboles Trie-Multibit con respaldo respecto a

uno sin respaldo, donde *N* es el número de prefijos almacenados, *k* es el *stride* del árbol y *w* es la longitud del prefijo más largo almacenado.

$$\frac{Memoria\ necesaria\ para\ respaldo\ de\ la\ tabla}{Memoria\ necesaria\ para\ implementación\ \text{sin}\ respaldo} * 100 = \frac{Nk}{N\left\lceil\frac{w}{k}\right\rceil 2^k} * 100 = \frac{k}{\left\lceil\frac{w}{k}\right\rceil 2^k} * 100$$

En la Tabla 10 se muestran los porcentajes de memoria extra que se requiere para el almacenamiento del respaldo, como ejemplo podemos leer el primer renglón de la tabla como: "los árboles Trie-Multibit de ocho bits con respaldo pueden llegar a necesitar hasta un 0.78% más de memoria que los árboles Trie-Multibit de ocho bits sin respaldo". Como se puede observar el peor de los casos analizados se da en los árboles de dos y cuatro bits ya que se puede llegar a ocupar hasta un 3.12% de memoria extra. Otro punto importante que debemos mencionar es que debido a los tamaños de memoria RAM que existen actualmente (alrededor de 4096 Mbytes) los árboles Trie-Multibit con respaldo pueden ser implementados sin problemas para el protocolo IPV4, para realizar esta tabla se asume que el prefijo más largo que se puede almacenar es de 32 bits.

Tabla 10 Porcentaje de consumo de memoria extra de los árboles Trie-Multibit con respaldo respecto a los árboles Trie-Multibit simples.

Árboles Trie-Multibit con respaldo y un *stride* de:	**Porcentaje de consumo de memoria extra respecto a los árboles Trie-Multibit simples**
8 bits	0.78%
6 bits	1.56%
4 bits	3.12%
2 bits	3.12%

Capítulo 4

4 Evaluación experimental de la propuesta

4.1 Validación de algoritmos

Una vez realizado el análisis teórico de los algoritmos que se encargan del borrado, inserción y búsqueda para la estructura de almacenamiento en árboles Mulibit con respaldo, realizamos la implementación de dichos algoritmos en lenguaje C.

Antes de llevar a cabo la evaluación experimental de los algoritmos, fue necesario validarlos, es decir fue necesario corroborar que cada algoritmo siguiera los órdenes de complejidad calculados teóricamente por lo que se procedió de la siguiente manera. En una computadora personal con un procesador Intel Pentium 4 a 3Ghz, con una memoria RAM de 3 Gbytes, se generó un prefijo de longitud aleatoria (distribución uniforme) de 1 a 32 bits, se utilizó el algoritmo de inserción para incluir el prefijo dentro de la estructura propuesta, posteriormente se realizó la búsqueda del mismo prefijo utilizando el algoritmo de búsqueda y finalmente se realizó la eliminación de dicho prefijo utilizando el algoritmo de borrado.

El procedimiento descrito anteriormente se realizó para cada una de las 32 longitudes posibles de los prefijos, se contó el número de ciclos de reloj que le tomó a cada algoritmo realizar su tarea, se calculó el promedio y la varianza de forma iterativa siguiendo las ecuaciones 1 y 2 respectivamente.

$$\bar{x}_{i+1} = \bar{x}_i + \frac{x_{i+1} - \bar{x}_i}{i+1} \quad para \quad i > 2 \qquad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (1)$$

$$s^2{}_{i+1} = \left(1 - \frac{1}{i}\right) s^2{}_i + (i+1)(\bar{x}_{i+1} - \bar{x}_i)^2 \quad para \; i > 2 \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (2)$$

Siguiendo el teorema del límite central (18) se utilizó la siguiente condición para saber si la cantidad de experimentos era suficiente.

$$C\left(\frac{s}{\sqrt{n}}\right) < error$$

Dónde:

$s = \sqrt{s^2} =$ Desviación estándar.
$n =$ Número de experimentos realizados.
$error =$ Error propuesto, para estos experimentos se propuso un error de 20 ciclos de reloj.
$C = 2.33 =$ Esto se debe a que deseamos que los experimentos tengan una confiabilidad del 98% y este valor de C satisface la siguiente ecuación.

$$0.98 = 1 - 2(1 - Q(C))$$

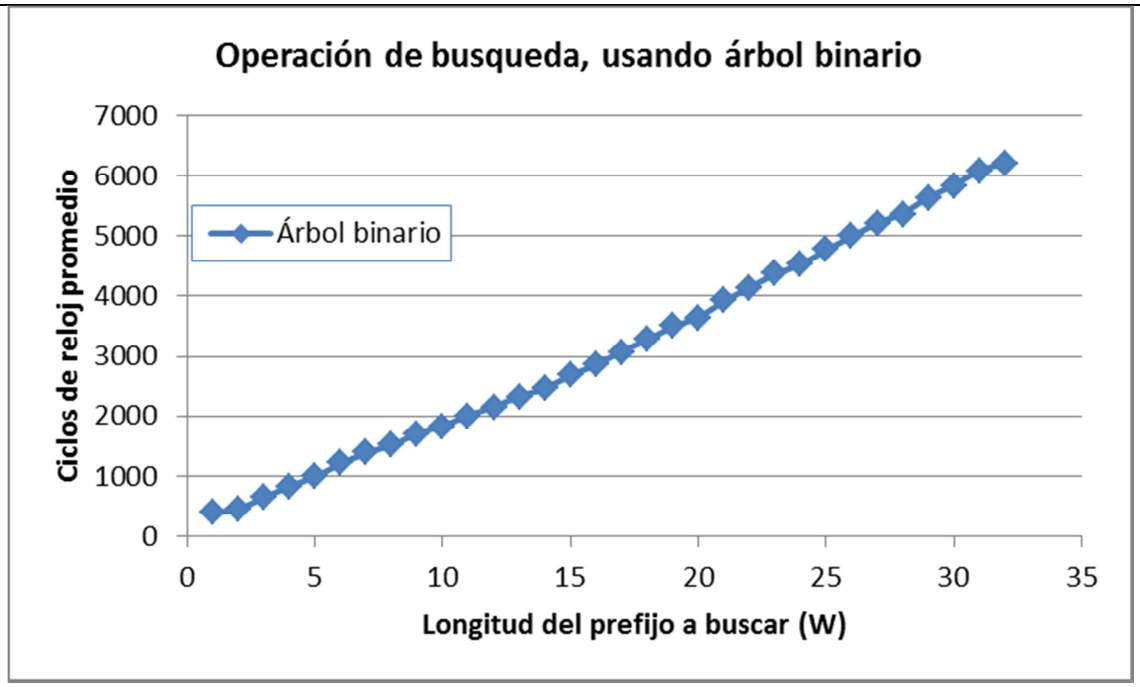
Donde $Q(x)$ es la función que denota el área bajo la curva de la distribución de probabilidad estándar. Los árboles utilizados para realizar la validación y la evaluación experimental se describen a continuación.

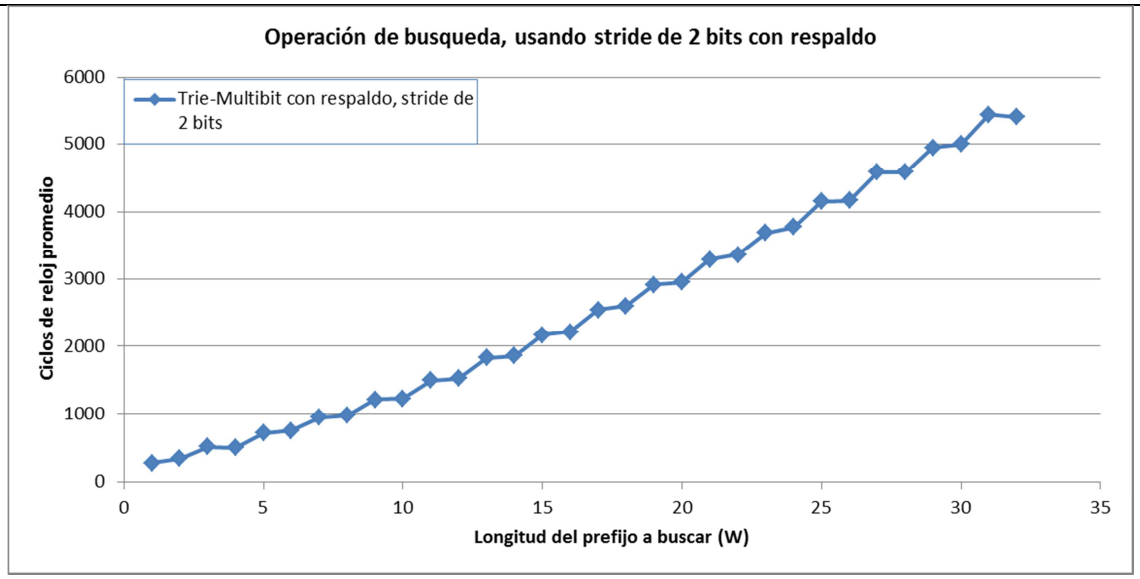
Se implementó un árbol binario de 32 niveles y cuatro árboles Trie-Multibit con respaldo, el primero posee 16 niveles y el *stride* que se utiliza para pasar de un nivel a otro es de dos bits, el segundo árbol contiene ocho niveles y su *stride* es de cuatro bits, el tercer árbol consta de seis niveles, para poder pasar a los primeros cinco niveles se tienen *strides* de seis bits y para pasar al sexto se tiene un *stride* de dos bits, por último el cuarto árbol consta de cuatro niveles y el *stride* de cada nivel es de ocho bits.

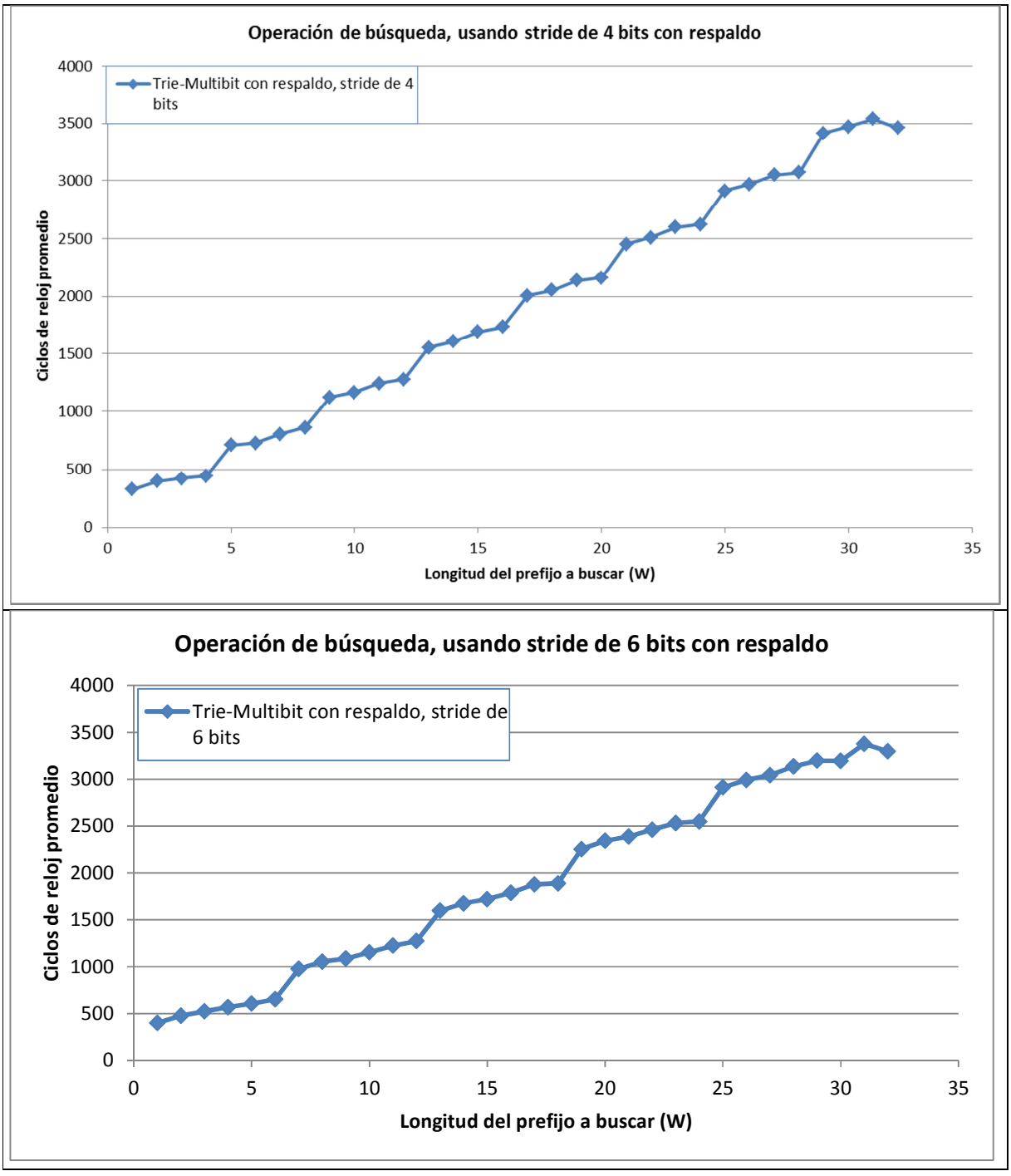
En la Tabla 11 se observan los resultados obtenidos en la validación de la operación de búsqueda, se puede observar que al almacenar los prefijos en los árboles Trie-Multibit con respaldo, la operación de búsqueda es más rápida que al almacenar los prefijos en árboles binarios simples, además la búsqueda es más rápida cuando el *stride* del árbol es más grande, esto sigue la tendencia de los órdenes de complejidad que se calcularon y que se muestran en la Tabla 11, por lo que podemos decir que los algoritmos de búsqueda fueron implementados apropiadamente y son válidos para realizar la evaluación de su desempeño con una tabla de ruteo real.

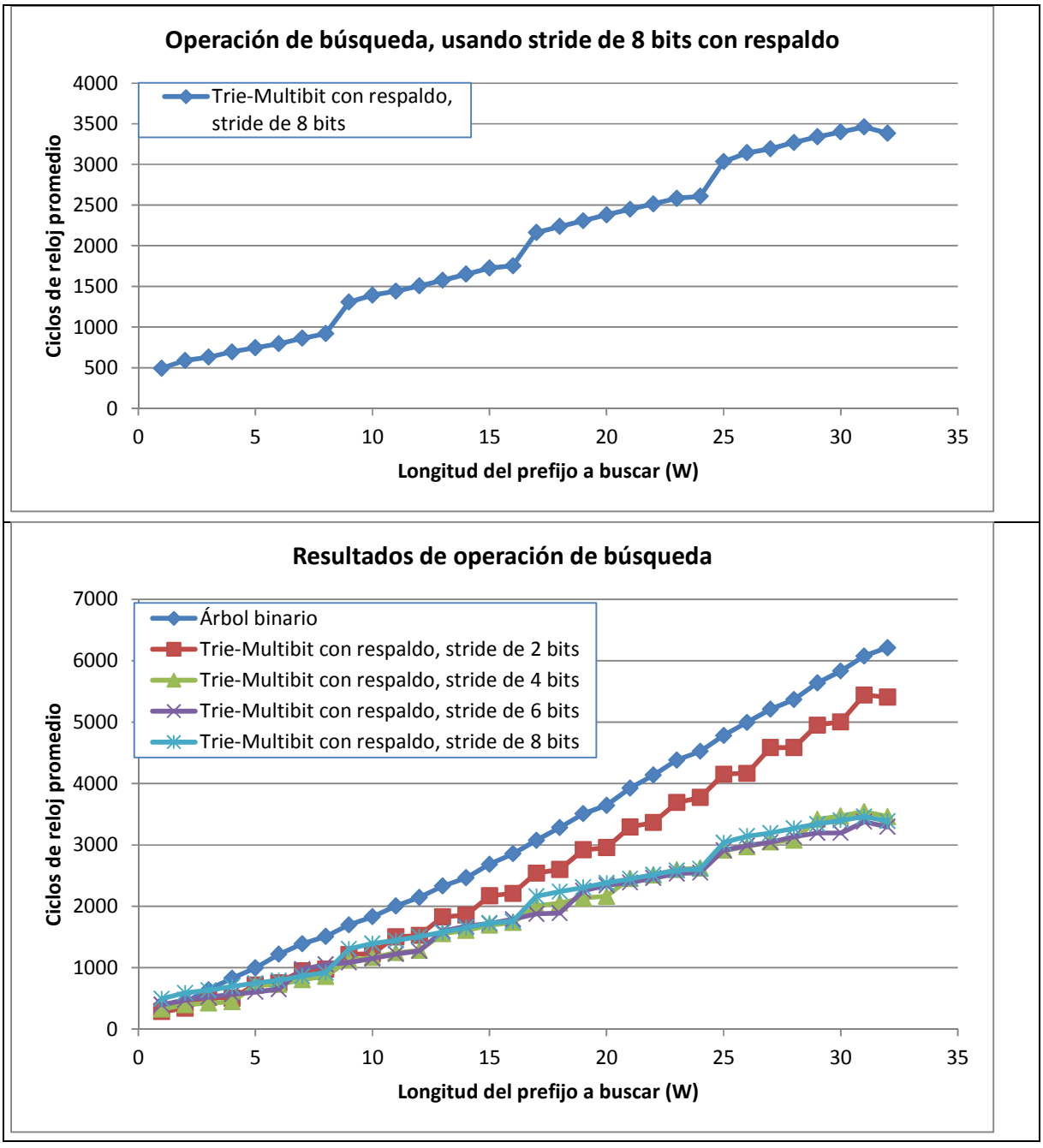
Tabla 11 Resultados experimentales obtenidos para validación de algoritmo de búsqueda.

RESULTADOS EXPERIMENTALES DE BÚSQUEDA



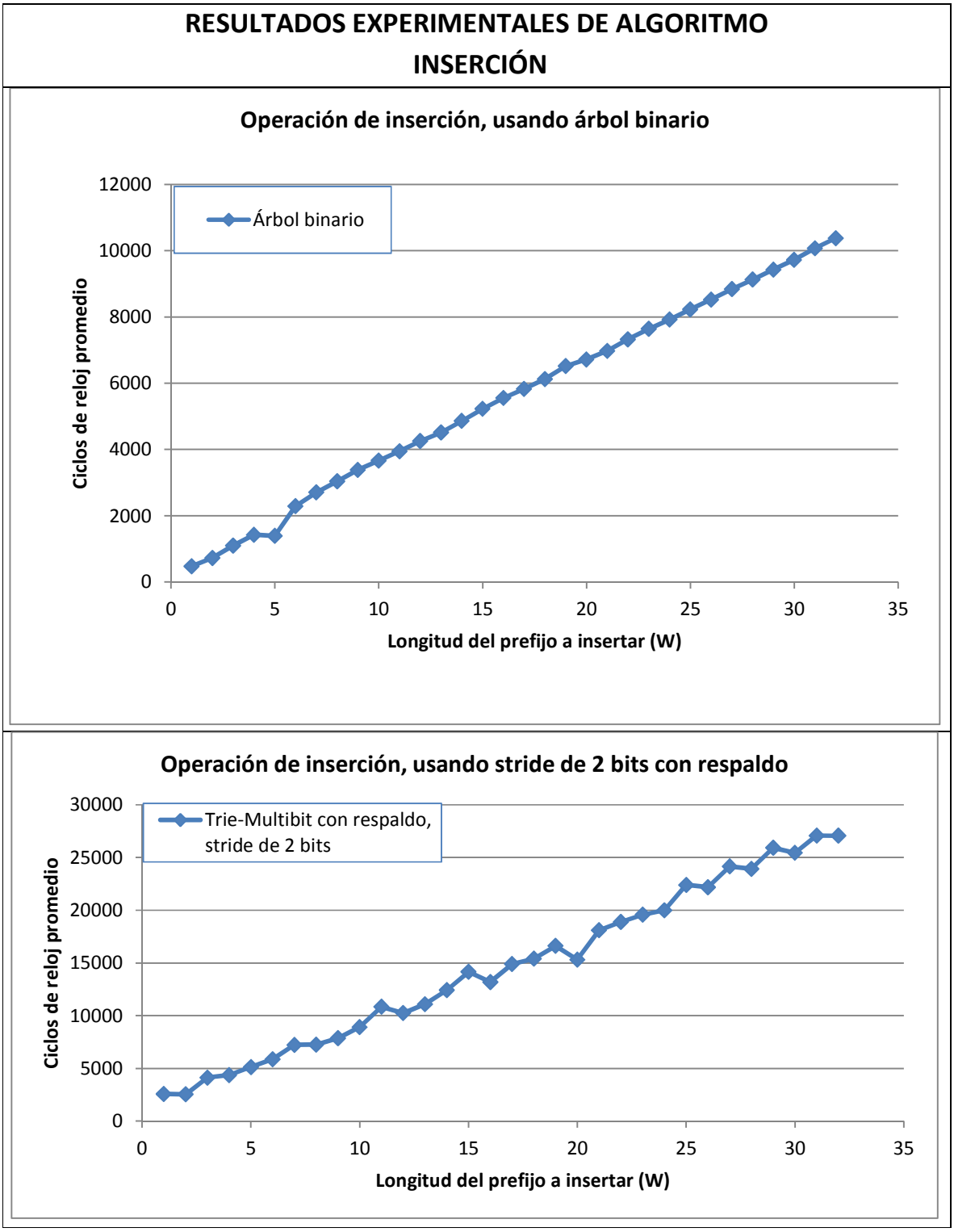


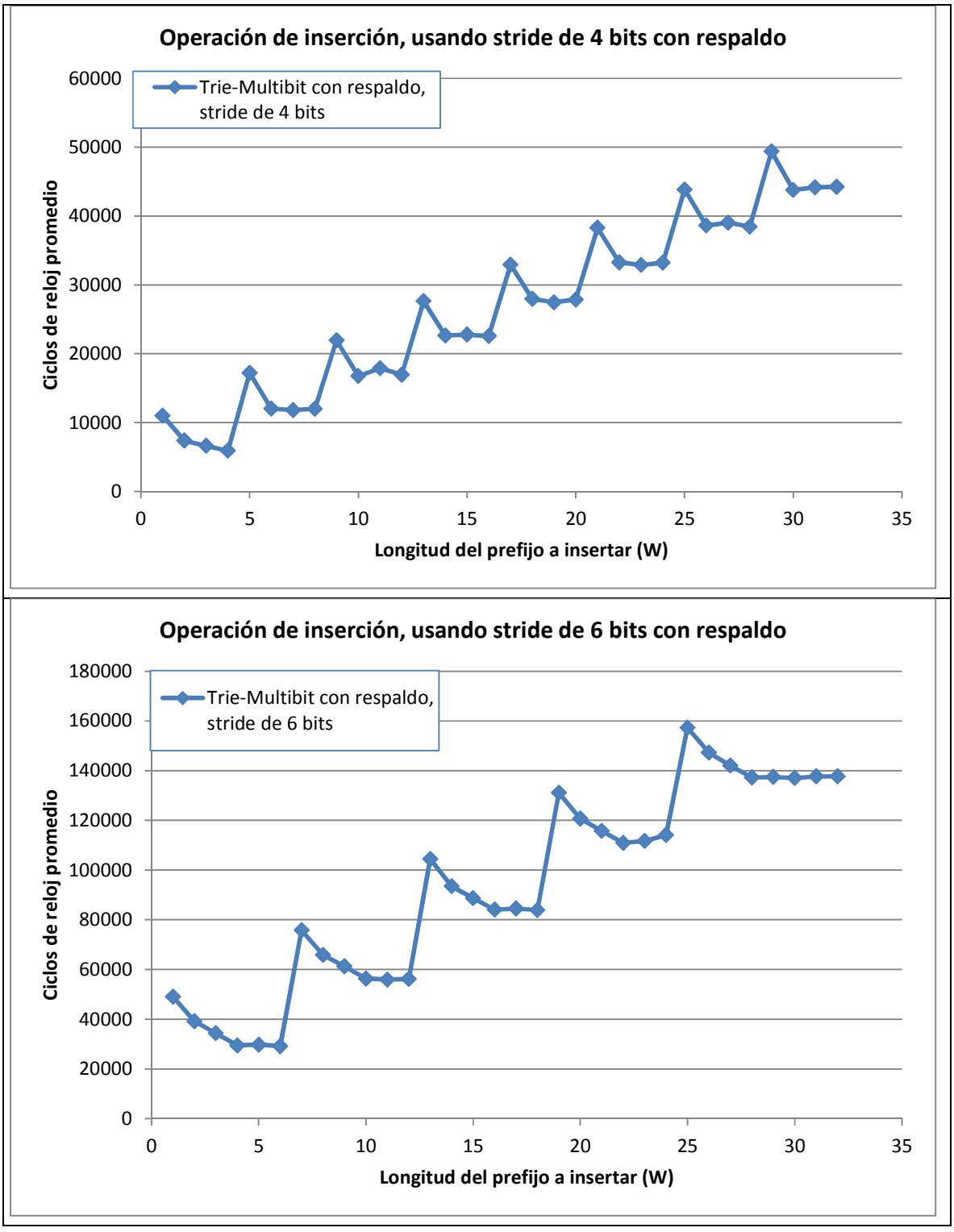

Operación de búsqueda, usando stride de 4 bits con respaldo
Trie-Multibit con respaldo, stride de 4 bits
Ciclos de reloj promedio
Longitud del prefijo a buscar (W)
0
500
1000
1500
2000
2500
3000
3500
4000
0
5
10
15
20
25
30
35
Operación de búsqueda, usando stride de 6 bits con respaldo
Trie-Multibit con respaldo, stride de 6 bits
Ciclos de reloj promedio
Longitud del prefijo a buscar (W)

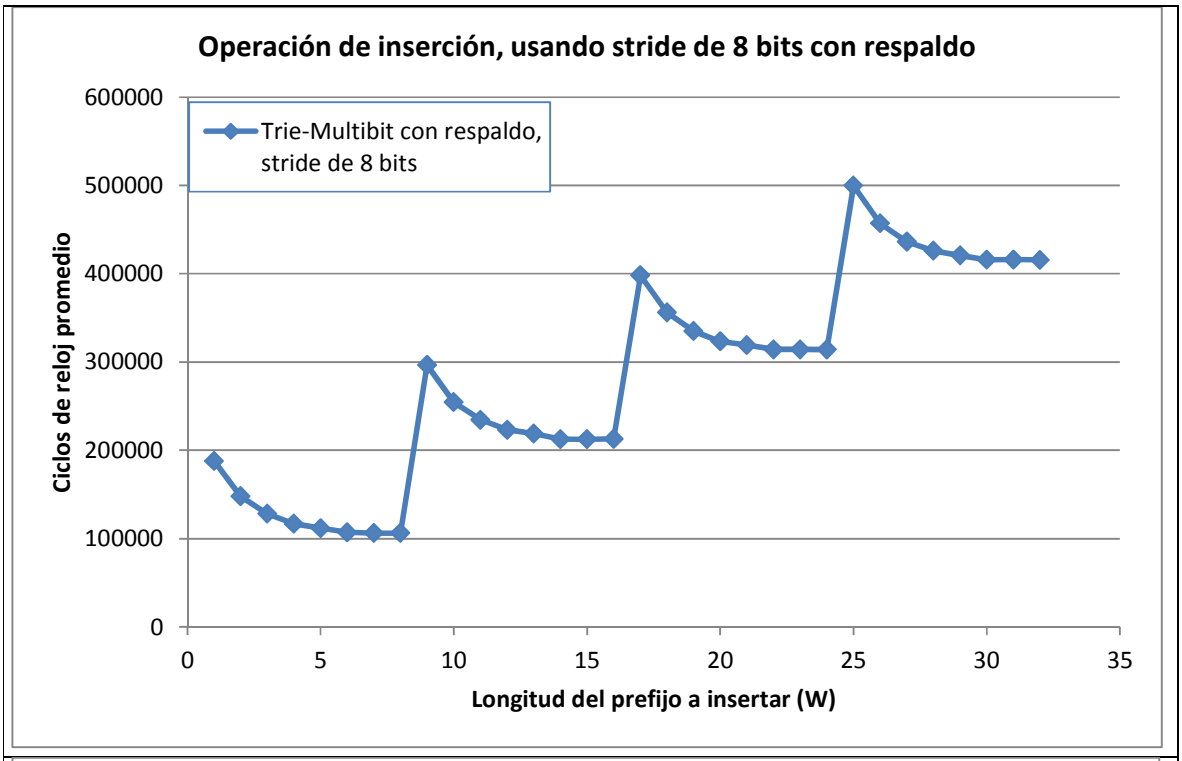


En la Tabla 12 y Tabla 13 se muestran los resultados de la evaluación de los algoritmos de inserción y borrado para el almacenamiento en árboles Trie-Multibit con respaldo, como se puede observar ambas figuras son muy parecidas, esto era de esperarse ya que como se mencionó en la sección 3.6 el orden de complejidad de la actualización aplica tanto para inserción como para borrado.

Tabla 12 Resultados experimentales obtenidos para la validación del algoritmo de inserción.




Operación de inserción, usando stride de 4 bits con respaldo
Trie-Multibit con respaldo, stride de 4 bits
Ciclos de reloj promedio
Longitud del prefijo a insertar (W)
0
10000
20000
30000
40000
50000
60000
0
5
10
15
20
25
30
35
Operación de inserción, usando stride de 6 bits con respaldo
Trie-Multibit con respaldo, stride de 6 bits
Ciclos de reloj promedio
Longitud del prefijo a insertar (W)
0
20000
40000
60000
80000
100000
120000
140000
160000
180000
0
5
10
15
20
25
30
35


Operación de inserción, usando stride de 8 bits con respaldo
Trie-Multibit con respaldo, stride de 8 bits
Ciclos de reloj promedio
600000
500000
400000
300000
200000
100000
0
0
5
10
15
20
25
30
35
Longitud del prefijo a insertar (W)

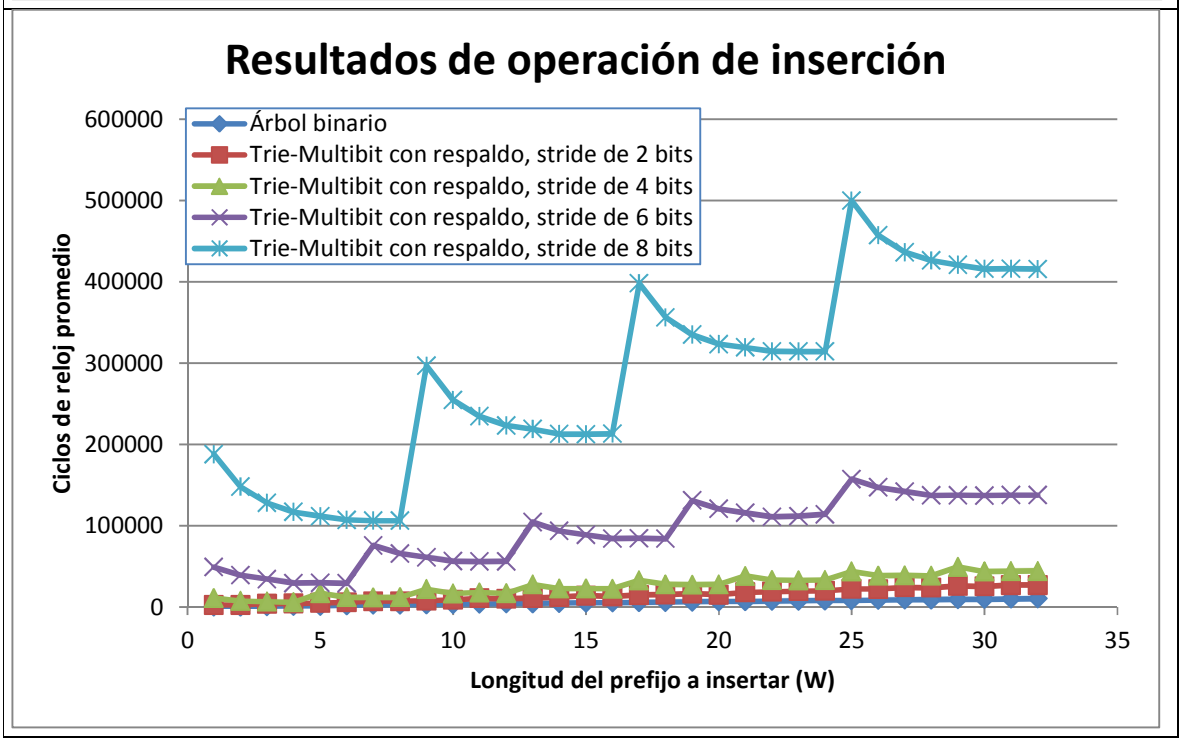
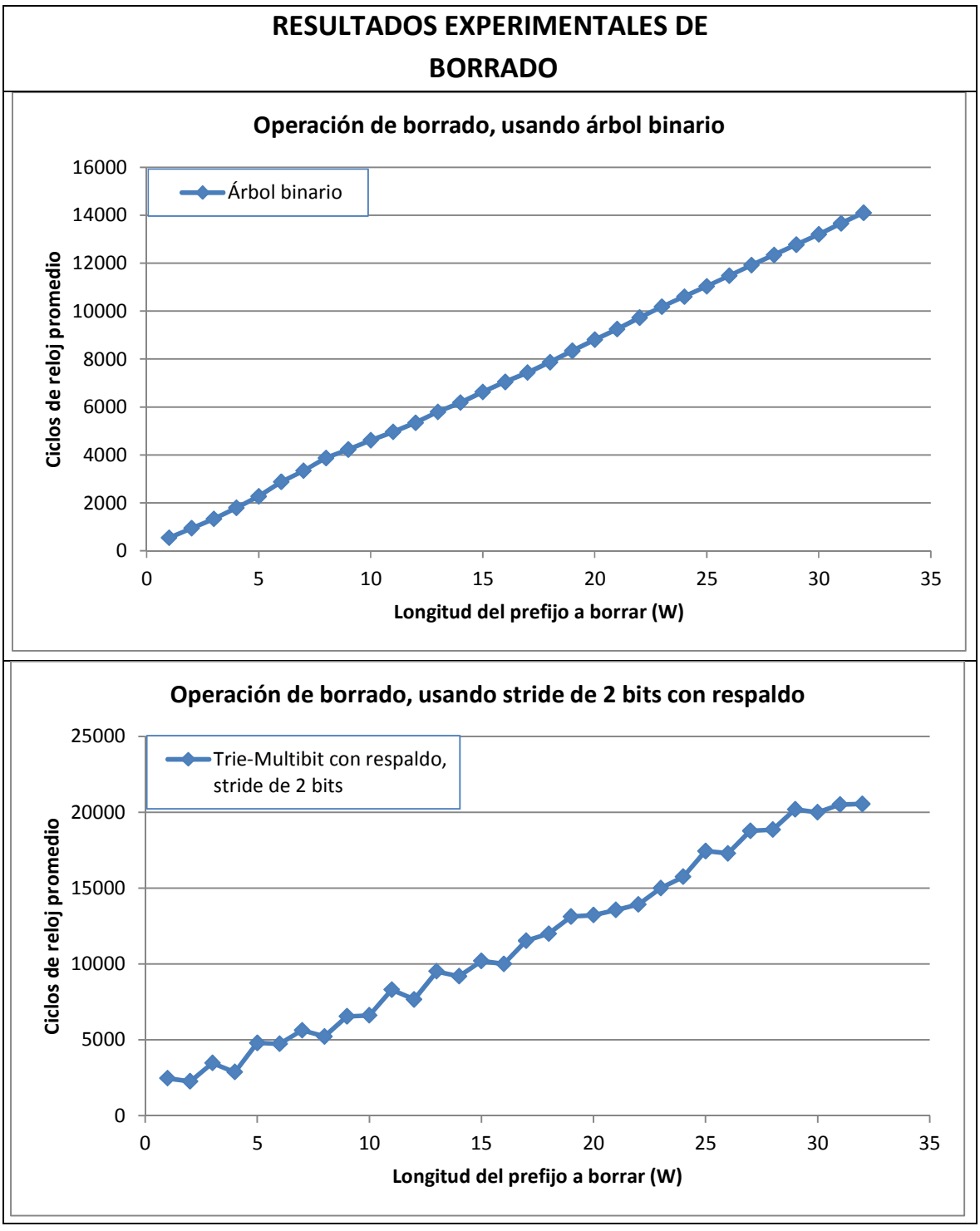
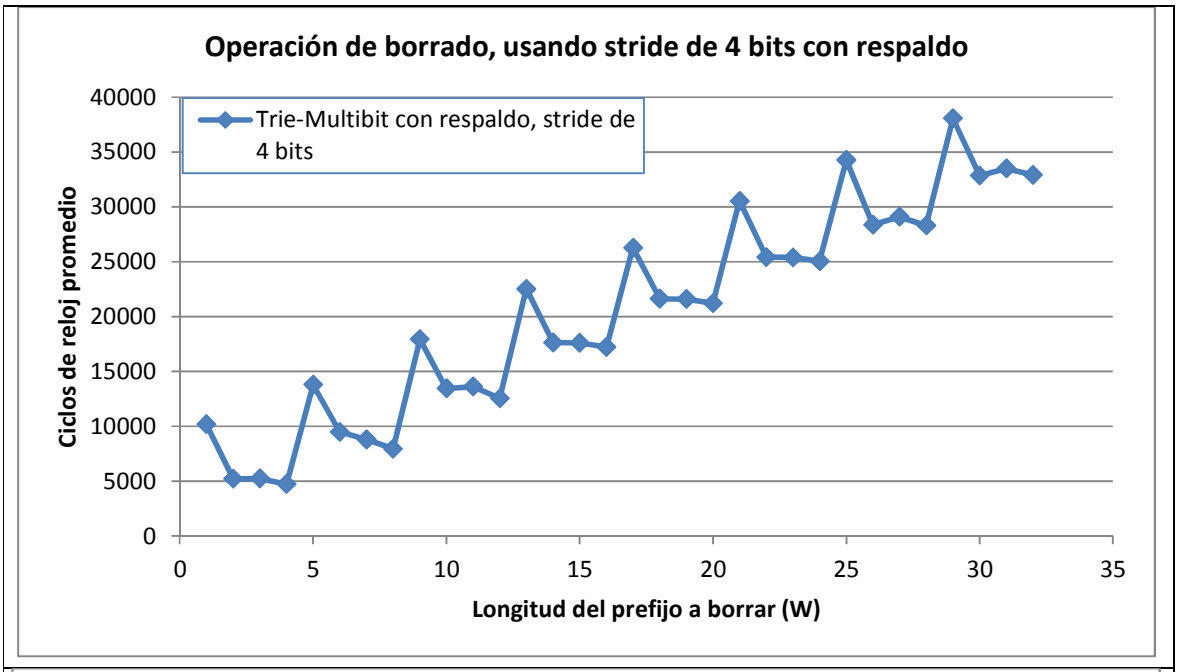
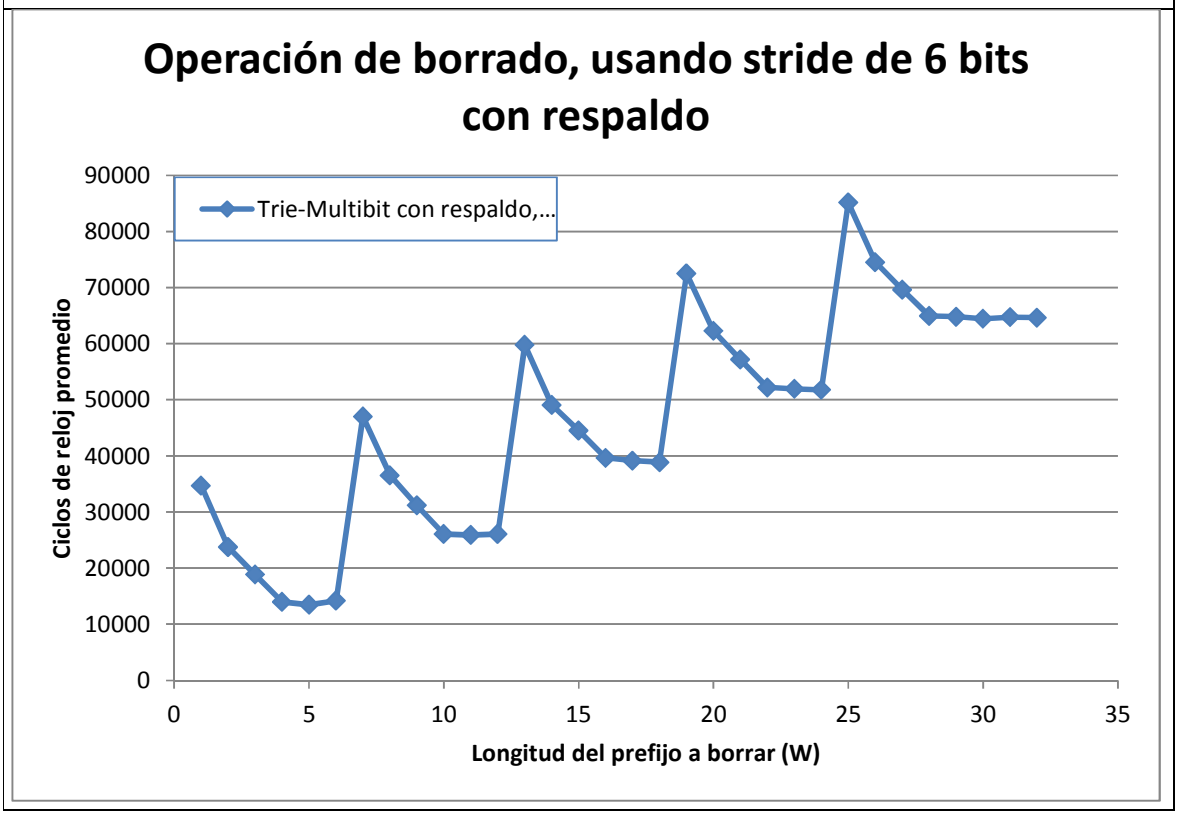
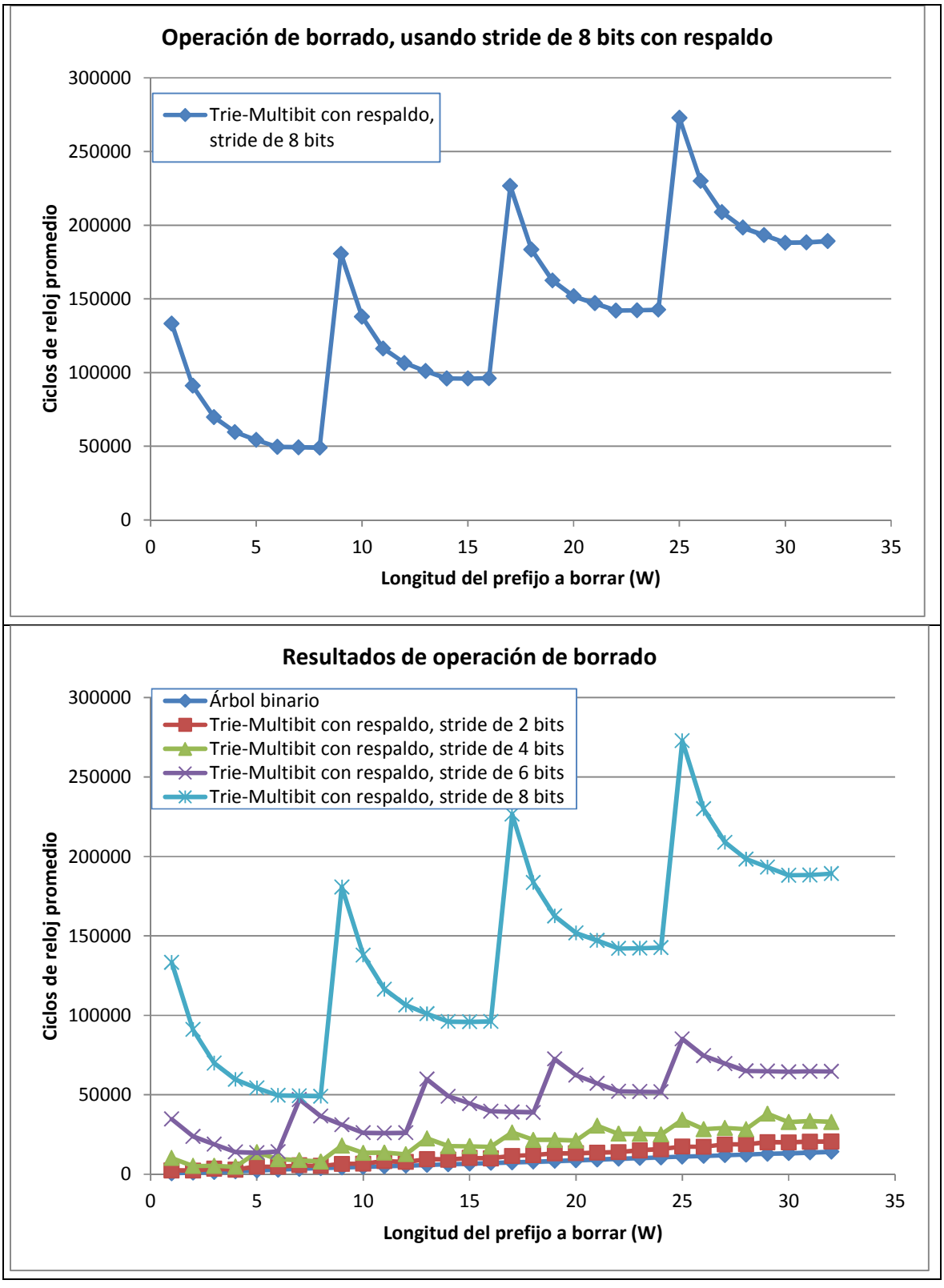

Resultados de operación de inserción
Árbol binario
Trie-Multibit con respaldo, stride de 2 bits
Trie-Multibit con respaldo, stride de 4 bits
Trie-Multibit con respaldo, stride de 6 bits
Trie-Multibit con respaldo, stride de 8 bits
Ciclos de reloj promedio
600000
500000
400000
300000
200000
100000
0
0
5
10
15
20
25
30
35
Longitud del prefijo a insertar (W)

Tabla 13 Resultados experimentales obtenidos para la validación del algoritmo de borrado.




Operación de borrado, usando stride de 4 bits con respaldo
Trie-Multibit con respaldo, stride de 4 bits
Ciclos de reloj promedio
Longitud del prefijo a borrar (W)
0
5000
10000
15000
20000
25000
30000
35000
40000
0
5
10
15
20
25
30
35


Operación de borrado, usando stride de 6 bits con respaldo
Trie-Multibit con respaldo,...
Ciclos de reloj promedio
Longitud del prefijo a borrar (W)
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
0
5
10
15
20
25
30
35


Operación de borrado, usando stride de 8 bits con respaldo
Trie-Multibit con respaldo, stride de 8 bits
Ciclos de reloj promedio
Longitud del prefijo a borrar (W)
300000
250000
200000
150000
100000
50000
0
0
5
10
15
20
25
30
35
Resultados de operación de borrado
Árbol binario
Trie-Multibit con respaldo, stride de 2 bits
Trie-Multibit con respaldo, stride de 4 bits
Trie-Multibit con respaldo, stride de 6 bits
Trie-Multibit con respaldo, stride de 8 bits
Ciclos de reloj promedio
Longitud del prefijo a borrar (W)

Como se puede observar ambas tablas muestran como las operaciones de actualización sobre los árboles Trie-Multibit con respaldo son más lentas a medida que el *stride* del árbol aumenta, además se puede ver cómo estas mismas operaciones sobre árboles binarios simples consumen menos tiempo que sobre los árboles Trie-Multibit con respaldo. En los resultados mostrados en la Tabla 12 y Tabla 13 se muestra que siguen las tendencias de los órdenes de complejidad mostrados en la Tabla 8, sin embargo, en los resultados experimentales se pueden observar varias crestas decrecientes en el cambio de nivel, esto se debe a lo siguiente:

Tomaremos como ejemplo el Trie-Multibit con *strides* de 4 bits, ahora intentaremos insertar un prefijo de un bit, nos damos cuenta que tenemos que expandir dicho prefijo ya que no coincide con los bits del nivel, así que necesitamos crear $2^{(4-1)} = 2^3$ nodos y escribirlos con la información correspondiente (ver procedimiento en la página 29). Lo mismo pasa si deseamos insertar un prefijo de 2 bits o 3, por lo que necesitamos crear los nodos faltantes en cada caso, lo mismo pasa para los prefijos que serán insertados en los siguientes niveles, esto se ve más claramente en la Tabla 14.

Tabla 14 listado que muestra el número de nodos creados de expansión por prefijo.

Longitud del prefijo que insertamos	Nodos de expansión que debemos crear y escribir
1	$2^{(4-1)} = 2^3 = 8$
2	$2^{(4-2)} = 2^2 = 4$
3	$2^{(4-3)} = 2^1 = 2$
4	$2^{(4-4)} = 2^0 = 1$
5	$2^{(4-1)} = 2^3 = 8$
6	$2^{(4-2)} = 2^2 = 4$
7	$2^{(4-3)} = 2^1 = 2$
8	$2^{(4-4)} = 2^0 = 1$
…	…

Como se puede observar en cada inicio de nivel, tenemos que crear y escribir más nodos que al final del mismo nivel, este comportamiento del algoritmo da lugar a las crestas observadas en las gráficas experimentales, cabe mencionar que estas no aparecen en los órdenes teóricos mostrados en la Tabla 8, esto se debe a que en los órdenes de complejidad son holgados y hemos tomado el peor de los casos es decir, se toma la creación del mayor número de nodos para todo el nivel. Con esto hemos corroborado que los algoritmos de inserción y borrado también son válidos para ser evaluados con una tabla de ruteo real.

4.2 Implementación con una tabla de ruteo típica del backbone

Una vez que hemos corroborado que los algoritmos implementados siguen los órdenes de complejidad calculados en la sección 3.6 podemos evaluar el desempeño de dichos algoritmos utilizando una tabla de ruteo típica del núcleo del internet, las características de la tabla que se utilizó para su evaluación se mencionan a continuación.

Fue tomada del sistema autónomo AS6447 el 7 de mayo de 2014 (1), cuenta con 445017 prefijos de red distribuidos de la siguiente forma:

Tabla 15 Distribución de prefijos de tabla de ruteo utilizada en evaluación experimental.

Longitud	***Frecuencia***	***Longitud***	***Frecuencia***
1	1	17	6786
2	0	18	11131
3	0	19	21999
4	0	20	32291
5	0	21	34036
6	0	22	47537
7	0	23	41959
8	18	24	230636
9	14	25	610
10	29	26	640
11	88	27	337
12	243	28	382
13	479	29	332
14	863	30	331
15	1570	31	26
16	12599	32	80
		TOTAL	**445017**

Dando lugar al siguiente histograma de frecuencias.

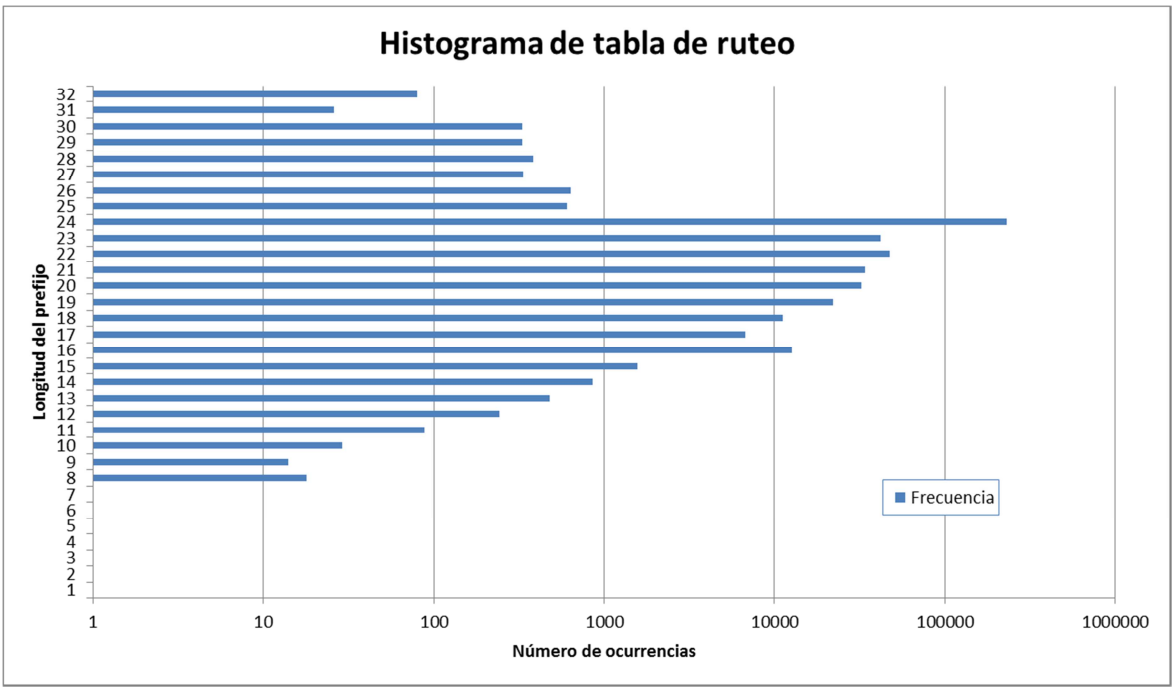


Figura 34 Histograma de frecuencias de la tabla de ruteo tomada del sistema autónomo AS6447

Para realizar la evaluación experimental, se procedió a crear un árbol binario y cuatro árboles Multibit con *strides* de 2, 4, 6 y 8 bits respectivamente, y se insertaron todos los prefijos de la tabla de ruteo dentro de dichos árboles.

Posterior a esto se generaron direcciones IP de 32 bits de forma aleatoria, siguiendo el protocolo IPV4, se midió el número de ciclos de reloj que le tomó al algoritmo de búsqueda encontrar el prefijo BMP (con una confianza del 95%) y la memoria consumida por las distintas estructuras al almacenar la tabla de ruteo, dando lugar a los siguientes resultados.

Tabla 16 Resultados experimentales almacenando una tabla de ruteo típica.

Estructura	Ciclos de reloj promedio en búsquedas	Memoria necesaria para almacenar la tabla de ruteo (Mbytes)
Árbol binario	2966.5	50.3
Trie-multibit con stride de 2 bits	2085.6	130.8
Trie-multibit con stride de 4 bits	1541.2	184.8
Trie-multibit con stride de 6 bits	1608.7	314.2
Trie-multibit con stride de 8 bits	1614.7	495.0

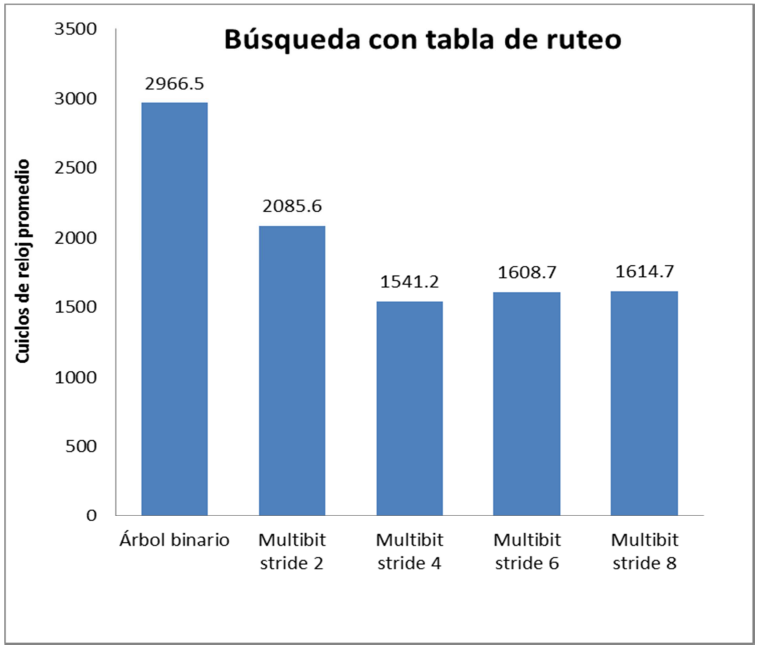


Figura 35 Ciclos de reloj al buscar el prefijo BMP utilizando la tabla de ruteo AS6447

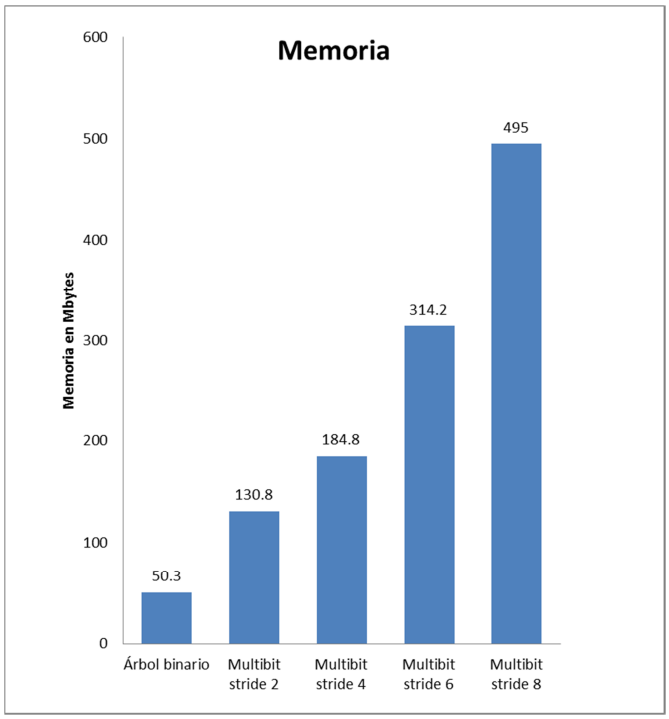


Figura 36 Memoria que utiliza cada esquema para almacenar la tabla AS6447

Capítulo 6

5 Conclusiones y análisis de resultados

De acuerdo a los resultados obtenidos podemos concluir que la búsqueda utilizando los árboles Trie-Multibit con respaldo es cada vez más rápida y las operaciones de actualización son cada vez más lentas cuando el *stride* del árbol crece. Si comparamos el desempeño de los algoritmos de los árboles Trie-Multibit con respaldo contra el desempeño de los algoritmos de los árboles binarios simples, nos damos cuenta que es conveniente utilizar el almacenamiento en árboles Trie-Multibit con respaldo ya que las operaciones de búsqueda son más rápidas y a pesar de que las operaciones de actualización son más lentas también son menos frecuentes.

Para poder darnos una idea de las proporciones entre las operaciones de búsqueda y las de actualización consideremos que la cantidad máxima de bits que puede procesar un *router* depende del tipo de línea de entrada que tenga, si suponemos que el *router* cuenta con una línea de entrada de tipo T1 su velocidad será de 1.5Mbps y considerando que un paquete en el internet tiene una longitud promedio de 354 bytes aproximadamente (2) el *router* estará procesando 530 búsquedas de prefijos en un segundo, además de acuerdo a estadísticas recolectadas de los 50 *routers* del núcleo del internet más dinámicos, se produce en promedio la actualización de un prefijo de la tabla de ruteo cada segundo (1), esto representa apenas el 0.1886 % de las operaciones de búsqueda que realiza el *router*, por lo que si el *router* está realizando búsquedas de prefijos la mayor parte del tiempo conviene que las búsquedas sean lo más rápidas posibles, esto hace que el almacenamiento en los árboles Trie-Multibit con respaldo sean mejor opción que el almacenamiento en árboles binarios. En la Tabla 17 se muestra como el porcentaje de actualizaciones es muy pequeño comparado con el número de búsquedas que realiza el *router* con distintos tipos de línea de entrada.

Tabla 17 Porcentaje de las operaciones de actualización para distintos tipos de líneas de entrada del *router.*

Tipo de línea en el *router*	Velocidad de la línea (Gbps)	Velocidad de la línea con paquetes de 354 bytes. (Mpps)	Porcentaje de operaciones de actualización
T1	0.0015	0.00053	0.1886 %
OC3c	0.155	0.054	$1.85 * 10^{-3}$ %
OC12c	0.622	0.22	$4.54*10^{-4}$ %
OC48c	2.50	0.88	$1.13*10^{-4}$ %
OC192c	10.0	3.53	$2.83*10^{-5}$ %
OC768c	40.0	14.1	$7.09*10^{-8}$ %

Por otra parte al analizar los resultados obtenidos procesando una tabla típica del internet vemos como al procesar direcciones IP aleatorias, cada uno de los esquemas implementados se comporta de manera distinta, el que presenta mayor retardo es el esquema basado en árboles binarios simples, mejorando significativamente al utilizar los árboles Trie-Multibit con respaldo, son embargo en la Figura 35 vemos como a medida que vamos aumentando el *stride* del árbol después de los 4 bits el algoritmo ya no mejora la velocidad de búsqueda, de hecho esta comienza a empeorar a partir del *stride* igual a 6. Este comportamiento se puede apreciar incluso al final de la Tabla 11 donde vemos la misma situación, esto nos lleva a concluir que el *stride* óptimo dependerá de la tabla de ruteo con la que trabaje. Para poder calcular el *stride* y el número de niveles óptimos Srinivasan y Varghese proponen en (8) hacer un pre-procesamiento de la tabla ruteo y por medio de programación dinámica calcular los valores ideales, y con esto construir el árbol adecuado, sin embargo habría que analizar las repercusiones en las operaciones de inserción y borrado.

Dentro de este trabajo se ha descrito un esquema de almacenamiento para prefijos de red IPV4 que se basa en árboles trie-Multibit y en árboles binarios. Se ha proporcionado la parte que le hacía falta a los árboles Trie-Multibit para ser utilizados en el almacenamiento de tablas de ruteo, ya que con la sección de respaldo esta estructura ya es capaz de garantizar la integridad de la información al realizar inserciones y borrados de prefijos de red.

Al analizar los resultados observamos que este esquema puede competir y superar al esquema clásico de almacenamiento de prefijos en árboles binarios simples. Sin embargo aún hay trabajo por hacer, la comparación del desempeño se ha realizado solamente con árboles binarios,

quedando pendiente realizarla con esquemas adicionales como los revisados en la sección 2 *estado del arte.*

Además sería interesante tratar de conseguir trazas reales de tráfico de internet que pudieran arrojarnos resultados más apegados a la realidad al momento de procesar la tabla típica del backbone. Como hemos mencionado también puede implementarse la idea de analizar estadísticamente la tabla de ruteo y utilizar cómputo dinámico para calcular los niveles y los *strides* óptimos, para así construir un árbol Trie-Multibit que se adapte a cada tabla de ruteo específica. Y por supuesto también analizar las repercusiones que tendrían todos estos cambios en las operaciones de búsqueda y actualización. Por último revisar la escalabilidad a IPV6 sería un buen complemento a este trabajo.

6 Referencias

1. **Smith, Philip.** BGP Routing Table Analysis Reports. [En línea] [Citado el: 7 de Mayo de 2014.] http://bgp.potaroo.net/.

2. **Wu, Weigong.** *packet forwarding technologies.* United States of America : Auerbach Publications, 2008.

3. **Kirschenhofer, Peter y Prodinger, Helmut.** *Some Further Results on Digital Search Trees.* Austria : Institut fur Algebra and Diskrete Mathematik, 1986.

4. **Leon Garcia, Alberto y Widjaja, Indra.** *Communication Networks.* s.l. : Mc Graw Hill, 2004.

5. **Semeria, Chuck.** *Understanding IP Addressing:Everything You Ever Wanted To Know.* s.l. : 3Com, 1996.

6. *Routing lookups in hardware at memory access speeds.* **Gupta, Pankaj, Lin, Steven y McKeown, Nick.** s.l. : IEEE INFOCOM, 1998, IEEE.

7. *Memory-efficient state lookups with fast updates.* **Sandeep, Sikka y Varghese, George.** Stockholm : SIGCOMM, 2000, ACM.

8. *Fast address lookups using controlled prefix expansion.* **Srinivasan, V. y Varghese, G.** s.l. : ACM Transaction on Computer Systems, 1999, ACM.

9. *A Dynamic Binary Hash Scheme for IPv6 Lookup.* **Qiong, Sun, y otros, y otros.** 2008, IEEE.

10. *Trie Memory.* **Fredkin, Edward, Beranek, Bolt y Newman.** 1960, ACM.

11. *Survey and Taxonomy of IP Address Lookup Algorithms.* **Ruiz Sánchez, Miguel Ángel, W. Biersack, Ernst y Dabbous, Walid.** 2, s.l. : IEEE Network, 2001, IEEE/ACM, Vol. 15.

12. *PATRICIA - Practical Algorithm To Retrieve Information.* **R. Morrison, Donald.** 1968, ACM.

13. *IP-Address Lookup Using LC-Tries.* **Nilsson, Stefan y Karlsson, Gunnar.** 1999, IEEE.

14. *Scalable High Speed IP Routing Lookups.* **Waldvogel, Marcel, y otros, y otros.** Cannes Francia : SIGCOMM, 1997, ACM.

15. *IP Loookups Using Multiway and Multicolumn Search.* **Lampson, Butler, Srinivasan, Venkatachary y Varghese, George.** 3, s.l. : IEE/ACM, 1999, IEEE/ACM, Vol. 7.

16. *Small Forwarding Tables for Fast Routing Lookups.* **Degermark, Mikael, y otros, y otros.** 1997, ACM SIGCOMM.

17. **Comer, Douglas E.** *Redes globales de informacion con internet y TCP/IP.* s.l. : Prentice Hall, 1996.

18. **M. Ross, Sheldon.** *Simulation.* California : Elsevier, 2006.

19. **Sklower, Keith.** *A Tree-Based Packet Routing Table for Berkeley Unix.* Berkeley, California : Computer Science Division University of California, 1991.

20. *An Efficient Compression Method for Patricia Tries.* **Shishibori, Masami, y otros, y otros.** 1997, IEEE.

21. *An Evaluation of IP-Address Lookup.* **Haider, Aun, Sirisena, Harsha y Mortensen, Brian B.** 2006, IEEE/ICIIS.

22. *Efficient Caching for IP Lookups.* **Ioannidis, Ioannis y Grama, Ananth.** 2004.

23. *Efficient Construction of Multibit Tries for IP Lookup.* **Sahni, Sartaj y Kim, Kun Suk.** 2003, ACM/IEEE.

24. *Efficient Construction of Pipelined Multibit-Trie Router-Tables.* **Kun Suk, Kim y Sartaj, Sahni.** 2007, IEEE.

25. *Fast and Scalable schemes for the IP address Lookup Problem.* **Yazdani, Nasser y Min, Paul S.** 2000, IEEE.

26. *Modified LC-Trie Based Efficient Routing Lookup.* **Liu, Ravikumar V.C Rabi Mahapatra J.C.** 2002, IEEE.

27. *Origins of Internet Routing Instability.* **Labovitz, Craig, Malan, G. Robert y Jahanian, Farnam.** 1999, IEEE.

28. *Parallel Searching Techniques for Routing Table Lookup.* **Knox, D. y Panchanathan, S.** 1993, IEEE.

29. *Range Tries for Scalable Address Lookup.* **Sourdis, Ioannis, y otros, y otros.** 2009, ACM.

30. *Routing on Longest-Matching Prefixes.* **Doeringer, Willibald.** 1996, IEEE/ACM.

31. *Fast Filter Updates for Packet Classification using TCAM.* **Song, Haoyu y Turner, Jonathan.** 2006, IEEE.

32. **Sánchez Jiménez, Fidel Ulises.** *Mecanismos de Codificación de Vector de Bits para Búsquedas en Tablas de Ruteo IP.* s.l. : Universidad Autónoma Metropolitana, 2012.

33. **Gómez Buendía, Joel Yazbek.** *Algoritmos de Búsqueda y Actualización de la informacion para ruteadores IP.* s.l. : Universidad Autónoma Metropolitana, 2008.

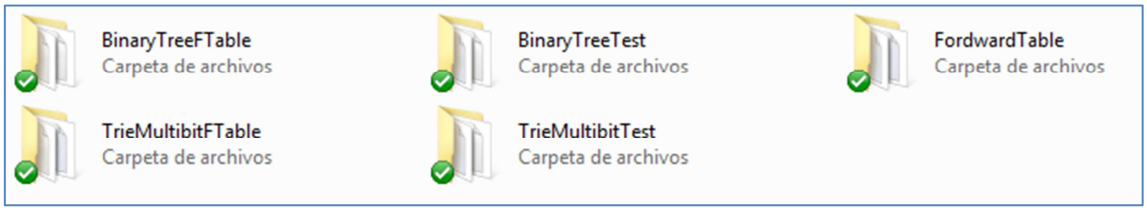
34. **Smith, Philip.** BGP Routing Table Analysis Reports. [En línea] [Citado el: 15 de febrero de 2010.] http://bgp.potaroo.net/.

7 ANEXO

PROGRAMAS IMPLEMENTADOS

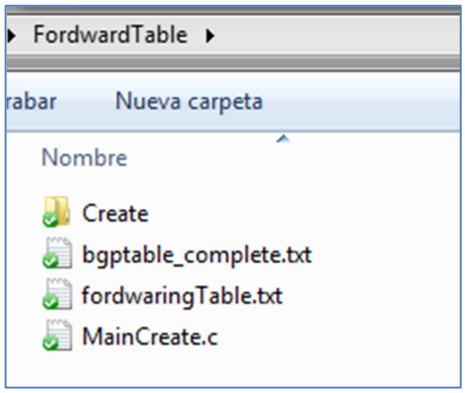
A continuación se describen algunas características de los programas elaborados al implementar este proyecto. Para realizar la implementación se utilizó el ambiente de desarrollo **Eclipse** en la versión Luna 4.4.1 y los programas fueron escritos en lenguaje C y compilados en un SO Linux en su distribución Ubuntu 14.04 LTS de 64 bits, utilizando el compilador gcc.

Se crearon 5 proyectos los cuales se muestran a continuación.



7.1.1 Construcción de tabla de reexpedición a partir de la tabla de ruteo completa

El primer proyecto que fue necesario crear fue el proyecto **FordwardTable** el cual a su vez contiene los siguientes elementos.



El archivo **MainCreate.c** es el archivo principal de este proyecto, este programa recibe como entrada una tabla de ruteo completa y crea una tabla de reexpedición la cual solamente incluye los prefijos de red actuales en notación binaria y con un número de interfaz. Este programa es necesario debido a que la tabla de ruteo completa mide alrededor de 650 Mbytes ya que incluye mucha más información que no nos es útil. El resto de los programas se encuentran en la carpeta **Create**.

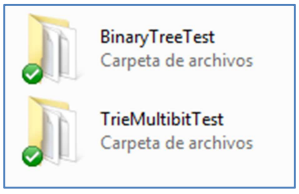
A continuación se muestra el código fuente de los archivos **MainCreate.c** y **Create.c**

```
 /*
//////////////////////////////MainCreate.C/////////////////////////////////////////////

   OCTUBRE-2014
            AUTOR
            Israel De Olmos Ramírez
            UNIVERSIDAD AUTÓNOMA METROPOLITANA - IZTAPALAPA, MEXICO, DF.
            Maestría en Ciencias y Tecnologías  de la Información
 */
#include<stdio.h>
#include"./Create/Create.h"
#define bgpTable  "./bgptable_complete.txt"
#define fordwaringTable "./fordwaringTable.txt"

int main(){
            Create(fordwaringTable,bgpTable);  //Crea la table de reexpedición a partir de una table de ruteo completa
return 0;
}

////////////////////////////////////////////////////////////////////////////////////////////////////
/* Create.c
   OCTUBRE-2014
            AUTOR
            Israel De Olmos Ramírez
            UNIVERSIDAD AUTÓNOMA METROPOLITANA - IZTAPALAPA, MEXICO, DF.
            Maestría en Ciencias y Tecnologías  de la Información
 */
#include<stdio.h>
#include<string.h>
#include"Create.h"

void Create(char* dest,char *route){
            char buffer[100];    //buffer de lectura de linea.
            char interfaceBuffer[20]="";
            char binaryPrefix[40];
            char outInterface[100][20]; //almacena hasta 100 interfaces de salida
            int j,i=0;
            for(j=0;j<100;j++)
            {
                        strcpy(outInterface[j],"            ");
            }
            ////////////////////////////////////////////////////////////////////
            // Apertura de archivo que contiene la tabla bgp y el archivo que contendrá la tabla de reexpedición(fordwaringTable)
            //
            FILE *p = fopen(route,"r+");// Archivo de lectura
            FILE *q = fopen(dest,"w");  //Archivo de escritura

            if(p!=NULL){
                        printf("Creando la tabla de reexpedición(fordwaringTable)...\n");
            }else{
                        printf("El archivo  %s no ha sido localizado.\n",route);
                        return;}
            ////////////////////////////////////////////////////////////////////
            fgets(buffer,100,p);
            getInterface(interfaceBuffer,buffer);
            strcpy(outInterface[i],interfaceBuffer);
```

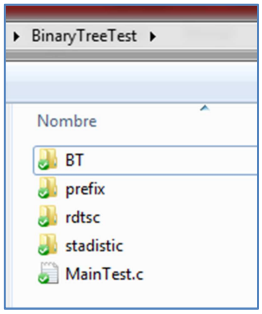
```
		prefixToBinary(binaryPrefix,buffer);
		fprintf(q,"%s %d\n",binaryPrefix,i);
		//printf("%s %d\n",binaryPrefix,i);
		i++;
		while(!feof(p)){
			fgets(buffer,100,p);
			if(findChr(buffer,'>')!=-1 && findChr(buffer,'/')!=-1){
				prefixToBinary(binaryPrefix,buffer);
				getInterface(interfaceBuffer,buffer);
				if(findSimilar(interfaceBuffer,outInterface)==-1)
				{
					strcpy(outInterface[i],interfaceBuffer);
					fprintf(q,"%s %d\n",binaryPrefix,i);
					//printf("%s %d\n",binaryPrefix,i);
					i++;
				}else{
					fprintf(q,"%s %d\n",binaryPrefix,findSimilar(interfaceBuffer,outInterface));
				}
			}
			strcpy(buffer,"");
		}
		printf("\nSe localizaron %d interfaces de salida distintas\n",i);
		printf("\nSe generó el archivo:%s \n",dest);
		fclose(p);
		fclose(q);
  return;
}
```

7.1.2 Validación de algoritmos de búsqueda y actualización

Los proyectos que fueron elaborados para validar las operaciones de búsqueda y de actualización de los árboles binarios y los árboles Trie-Multibit con respaldo fueron los siguientes.



El proyecto **BinaryTreeTest** fue realizado para validar las operaciones de búsqueda y actualización del esquema basado en árboles binarios, el contenido de este proyecto se muestra a continuación.



Cada elemento del proyecto se describe en la tabla siguiente.

Elemento del proyecto	Descripción
Carpeta **BT**	Contiene los algoritmos que se encargan de la inserción, borrado y búsqueda dentro de los árboles binarios.
Carpeta **Prefix**	Contiene los algoritmos necesarios para el manejo de los prefijos en formato binario.
Carpeta **rdtsc**	Contiene los algoritmos necesarios para realizar la medición de los ciclos de reloj de cada operación.
Carpeta **stadistics**	Contiene los algoritmos necesarios para llevar el manejo de la estadística de cada una de las muestras tomadas.
Archivo **MainTest.c**	Contiene el programa principal encargado de realizar la validación de las operaciones de búsqueda y actualización de árboles binarios.

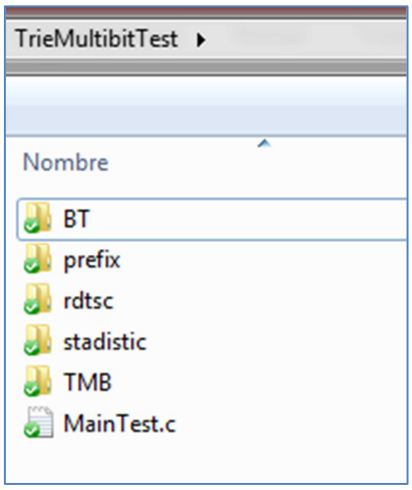
A continuación se muestra el código fuente del archivo **Maintest.c**

```
/*
//////////////////////////////MainTest.C/////////////////////////////////////////////

   OCTUBRE-2014
          AUTOR
          Israel De Olmos Ramírez
          UNIVERSIDAD AUTÓNOMA METROPOLITANA - IZTAPALAPA, MEXICO, DF.
          Maestría en Ciencias y Tecnologías  de la Información
 */

#include "./stadistic/stadistic.h"
#include "./prefix/prefix.h"
#include "./rdtsc/rdtsc.h"
#include "./BT/BinaryTree.h"
#include <stdlib.h>
#include <time.h>
#include <math.h>
#define C 2.33    // C = 1.96 para tener una confianza del 95% ;   0.95 = 1 - 2(1-Q(C))
                  // C = 2.33 para tener una confianza del 98% ;   0.95 = 1 - 2(1-Q(C))

   int error=1;   // Error que se propone, se debe cumplir que   C*desv_est/pow(n,0.5) <= error
                  //por el teorema del límite central

   long double initTime,endTime,duration;

   int main(){
   int i;
   long double random,j=1;
   long double average,previousAverage=0;
   long double variance,previousVariance=0;
   long double estandarDesviation,errorFlag;
   char binaryPrefix[40],ipAdress[40];
   unsigned int seed = time(NULL);  //Se inicializa la semilla de la función aleatoria
   srand(seed);
   BTNode *root;

   root=createBTNode(NULL);
```

```
//---*****************************************************************************
// calcula estadisticas de inserción
//-----------------------------------------------------------------------------

for(i=1;i<=32;i++){              //Se realiza una evaluación a prefijos de longitudes desde 1 hasta 32 bits
        j=1;
        do{
        random = randomPrefix(pow(2,i-1));  //Se genera un prefijo aleatorio en forma de un número decimal
                                            //respetando el rango válido de acuerdo a la longitud del prefijo.
        printbin(random,i,binaryPrefix);     //El prefijo es escrito a su notación binaria asignando una interfaz ficticia.

          ////////////////////////////////////////////
          // Ventana de medición de tiempo
          //----------------------------------------
                initTime =rdtsc();
                 insertPrefixBT(binaryPrefix,root);   //Se inserta el prefijo dentro del árbol binario
                endTime = rdtsc();
          ////////////////////////////////////////////////

                deletePrefixBT(binaryPrefix,root);   //Se borra el prefijo del árbol binario
                duration =endTime-initTime;
                average = Average(previousAverage,duration,j);
                variance = Variance(previousVariance,previousAverage,average,duration,j);
                estandarDesviation = sqrt(variance);
                errorFlag = C*estandarDesviation/sqrt(j);
          //----------------------------------------
                previousAverage = average;
                previousVariance = variance;
                j++;

                }while( errorFlag > error || j<100);
                 printf("\nINSERCIÓN   longitud %d -- %LF - muestra %LF  promedio %LF  varianza %LF desviación
estandar %lf criterio de paro %LF",i,j,duration,average,variance,sqrt(variance),errorFlag);
}// for

//---*****************************************************************************
// calcula estadisticas de búsqueda
//-----------------------------------------------------------------------------
error=1;
 for(i=1;i<=32;i++){              //Se realiza una evaluación a prefijos de longitudes desde 1 hasta 32 bits
        j=1;
        do{
        random = randomPrefix(pow(2,i-1));  //Se genera un prefijo aleatorio en forma de un número decimal
                                            //respetando el rango válido de acuerdo a la longitud del prefijo

          printbin(random,i,binaryPrefix);     //El prefijo es escrito a su notación decimal asignando una interfaz ficticia
          generateIpAdress(binaryPrefix,ipAdress);   //Genera una IP válida para ser buscada dentro del árbol Binario
        insertPrefixBT(binaryPrefix,root);          //Se inserta el prefijo dentro del árbol binario
          ////////////////////////////////////////////
         // Ventana de medición de tiempo
          //----------------------------------------
                initTime =rdtsc();
                findBT(ipAdress,root);              //Se busca la dirección Ip dentro del árbol binario
                endTime = rdtsc();
          ////////////////////////////////////////////////
                deletePrefixBT(binaryPrefix,root);       //Se borra el prtefijo dentro del árbol binario
```

```
                                duration =endTime-initTime;
                                average = Average(previousAverage,duration,j);
                                variance = Variance(previousVariance,previousAverage,average,duration,j);
                                estandarDesviation = sqrt(variance);
                                errorFlag = C*estandarDesviation/sqrt(j);
                                //----------------------------------------
                                previousAverage = average;
                                previousVariance = variance;
                                j++;
                                }while( errorFlag > error || j<100);
                                 printf("\nBUSQUEDA   longitud %d -- %LF - muestra %LF  promedio %LF  varianza %LF desviación
estandar %lf criterio de paro %LF",i,j,duration,average,variance,sqrt(variance),errorFlag);
        }// for

        //---*************************************************************************
        // calcula estadisticas de borrado
        //-------------------------------------------------------------------------------
  error = 1;
        for(i=1;i<=32;i++){             //Se realiza una evaluación a prefijos de longitudes desde 1 hasta 32 bits
                j=1;
                do{
                random = randomPrefix(pow(2,i-1));  //Se genera un prefijo aleatorio en forma de un número decimal
                                                    //respetando el rango válido de acuerdo a la longitud del prefijo

                printbin(random,i,binaryPrefix);     //El prefijo es escrito a su notación decimal asignando una interfaz ficticia
               insertPrefixBT(binaryPrefix,root);  //Se inserta el prefijo dentro del árbol binario
                 ////////////////////////////////////////////
                // Ventana de medición de tiempo
                //----------------------------------------
                        initTime =rdtsc();

                        deletePrefixBT(binaryPrefix,root);    //Se borra el prefijo del árbol binario

                        endTime = rdtsc();
                 ////////////////////////////////////////////

                        duration =endTime-initTime;
                        average = Average(previousAverage,duration,j);
                        variance = Variance(previousVariance,previousAverage,average,duration,j);
                        estandarDesviation = sqrt(variance);
                        errorFlag = C*estandarDesviation/sqrt(j);

                //----------------------------------------
                        previousAverage = average;
                        previousVariance = variance;
                        j++;

                        }while( errorFlag > error || j<100);
                         printf("\nBORRADO   longitud %d -- %LF - muestra %LF  promedio %LF  varianza %LF desviación
estandar %lf criterio de paro %LF",i,j,duration,average,variance,sqrt(variance),errorFlag);
        }// for

  return 0;
}
```

El proyecto **TrieMultibitTest** fue realizado para validar las operaciones de búsqueda y actualización del esquema basado en árboles Multibit, el contenido de este proyecto se muestra a continuación.



Cada elemento del proyecto se describe en la tabla siguiente.

Elemento del proyecto	Descripción
Carpeta **BT**	Contiene los algoritmos que se encargan de la inserción, borrado y búsqueda dentro de los árboles binarios de respaldo.
Carpeta **Prefix**	Contiene los algoritmos necesarios para el manejo de los prefijos en formato binario.
Carpeta **rdtsc**	Contiene los algoritmos necesarios para realizar la medición de los ciclos de reloj de cada operación.
Carpeta **stadistics**	Contiene los algoritmos necesarios para llevar el manejo de la estadística de cada una de las muestras tomadas.
Carpeta **TMB**	Contiene los algoritmos que se encargan de la inserción, borrado y búsqueda dentro de los árboles Trie-Multibit.
Archivo **MainTest.c**	Contiene el programa principal encargado de realizar la validación de las operaciones de búsqueda y actualización de árboles Trie-Multibit con respaldo.

A continuación se muestra el código fuente del archivo **MainTest.c**

```
/*
//////////////////////////////MainTest.C/////////////////////////////////////////////
   OCTUBRE-2014
            AUTOR
            Israel De Olmos Ramírez
            UNIVERSIDAD AUTÓNOMA METROPOLITANA - IZTAPALAPA, MEXICO, DF.
            Maestría en Ciencias y Tecnologías  de la Información
 */

#include<stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>
#include "./rdtsc/rdtsc.h"
#include "./TMB/MultibitTrie.h"
#include "./prefix/prefix.h"
#include "./stadistic/stadistic.h"

#define C 2.33     // C = 1.96 para tener una confianza del 95% ;    0.95 = 1 - 2(1-Q(C))
                   // C = 2.33 para tener una confianza del 98% ;    0.95 = 1 - 2(1-Q(C))

int error;    // Error que se propone, se debe cumplir que   C*desv_est/pow(n,0.5) <= error
            //por el teorema del límite central

char binaryPrefix[40],ipAdress[40];
int strides[35];
long long int nPrefix[35], nIpAdress[35];
long double initTime,endTime,duration;
int i=1,interface=9;

long double Average(long double previousAverage,long double sample, long double i);
long double Variance(long double previousVariance,long double previousAverage,long double average, long double sample, long
double i);

int main(){
            long double random,j=1;
            long double average,previousAverage=0;
            long double variance,previousVariance=0;
            long double estandarDesviation,errorFlag;

            TMBNode *multibitTrie;
           ////////////////////////////////////////////////////////////
            // Se indican los strides del árbol
            strides[0]=2;
            strides[1]=2;
            strides[2]=2;
            strides[3]=2;
            strides[4]=2;
            strides[5]=2;
            strides[6]=2;
            strides[7]=2;
            strides[8]=2;
            strides[9]=2;
            strides[10]=2;
            strides[11]=2;
```

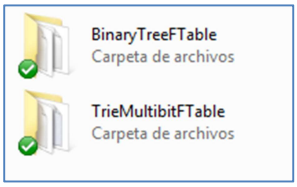
```
        strides[12]=2;
        strides[13]=2;
        strides[14]=2;
 strides[15]=2;
 strides[16]=-1;//bandera que indica el fin de los strides
////////////////////////////////////////////////////////////


        unsigned int seed = time(NULL);  //Se inicializa la semilla de la función aleatoria
        srand(seed);                //utilizando la estampa de tiempo actual
        multibitTrie = createTMBNodes(1);   //Crea en nodo raiz del árbol TrieMultibit



        //---*************************************************************************
        // calcula estadisticas de inserción
        //-------------------------------------------------------------------------
        error=20;
        for(i=1;i<=32;i++){             //Se realiza una evaluación a prefijos de longitudes desde 1 hasta 32 bits
                j=1;
                do{
                random = randomPrefix(pow(2,i-1));  //Se genera un prefijo aleatorio en forma de un número decimal
                                                    //respetando el rango válido de acuerdo a la longitud del prefijo

                printbin(random,i,binaryPrefix);     //El prefijo es escrito a su notación decimal asignando una interfaz ficticia
                getNPrefix(strides,binaryPrefix,nPrefix); //Se calcula la posición del prefijo en cada nivel del árbol.

                ////////////////////////////////////////////
                // Ventana de medición de tiempo
                //----------------------------------------
                initTime =rdtsc();
                //.......................................
                insertPrefixTMB(0,strides,strides[0],i,nPrefix,interface,binaryPrefix,multibitTrie);
                //.......................................
                endTime = rdtsc();
                ////////////////////////////////////////////

                deletePrefixTMB(0,strides,strides[0],i,nPrefix,interface,binaryPrefix,multibitTrie);
                 duration =endTime-initTime;
                average = Average(previousAverage,duration,j);
                variance = Variance(previousVariance,previousAverage,average,duration,j);
                estandarDesviation = sqrt(variance);
                errorFlag = C*estandarDesviation/sqrt(j);
                //----------------------------------------
      previousAverage = average;
      previousVariance = variance;
      j++;

                }while( errorFlag > error || j<100);
    printf("\nINSERTCIÓN   longitud %d -- %LF - muestra %LF  promedio %LF  varianza %LF desviación estandar %lf criterio de paro %LF",i,j,duration,average,variance,sqrt(variance),errorFlag);
        }// for
```

```
        //---*****************************************************************************
        // calcula estadisticas de búsqueda
        //-----------------------------------------------------------------------------
        error=1;
        for(i=1;i<=32;i++){          //Se realiza una evaluación a prefijos de longitudes desde 1 hasta 32 bits

        j=1;
        do{
        random = randomPrefix(pow(2,i));  //Se genera un prefijo aleatorio en forma de un número decimal
                                          //respetando el rango válido de acuerdo a la longitud del prefijo

        printbin(random,i,binaryPrefix);     //El prefijo es escrito a su notación decimal asignando una interfaz ficticia
        getNPrefix(strides,binaryPrefix,nPrefix); //Se calcula la posición del prefijo en cada nivel del árbol.
        generateIpAdress(binaryPrefix,ipAdress); //Genera una IP válida para ser buscada dentro del árbol Trie-Multibit
        insertPrefixTMB(0,strides,strides[0],i,nPrefix,interface,binaryPrefix,multibitTrie);
       ////////////////////////////////////////////
        // Ventana de medición de tiempo
        //----------------------------------------
        initTime =rdtsc();
        //.......................................
        findTMB(0,strides,ipAdress,multibitTrie);
        //.......................................
        endTime = rdtsc();
        ////////////////////////////////////////////

      deletePrefixTMB(0,strides,strides[0],i,nPrefix,interface,binaryPrefix,multibitTrie);
      duration =endTime-initTime;
      average = Average(previousAverage,duration,j);
      variance = Variance(previousVariance,previousAverage,average,duration,j);
      estandarDesviation = sqrt(variance);
      errorFlag = C*estandarDesviation/sqrt(j);
      //----------------------------------------
      previousAverage = average;
      previousVariance = variance;
      j++;
      }while( errorFlag > error || j<100);
    printf("\nBUSQUEDA   longitud %d -- %LF - muestra %LF  promedio %LF  varianza %LF desviación estandar %lf criterio de
paro %LF",i,j,duration,average,variance,sqrt(variance),errorFlag);
        }// for

        //---*****************************************************************************
        // calcula estadisticas de borrado
        //-----------------------------------------------------------------------------
        error=20;
        for(i=1;i<=32;i++){              //Se realiza una evaluación a prefijos de longitudes desde 1 hasta 32 bits
                j=1;
                do{
                random = randomPrefix(pow(2,i-1));  //Se genera un prefijo aleatorio en forma de un número decimal
                                                    //respetando el rango válido de acuerdo a la longitud del prefijo
                 printbin(random,i,binaryPrefix);       //El prefijo es escrito a su notación decimal asignando una interfaz ficticia
               getNPrefix(strides,binaryPrefix,nPrefix); //Se calcula la posición del prefijo en cada nivel del árbol.
               insertPrefixTMB(0,strides,strides[0],i,nPrefix,interface,binaryPrefix,multibitTrie);
               ////////////////////////////////////////////
              // Ventana de medición de tiempo
              //----------------------------------------
             initTime =rdtsc();
```

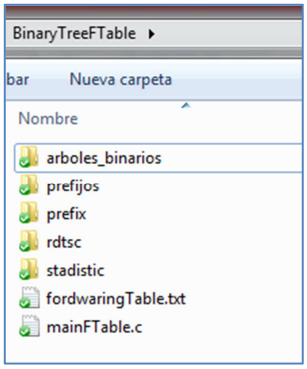
```
                //.....
                deletePrefixTMB(0,strides,strides[0],i,nPrefix,interface,binaryPrefix,multibitTrie);
                //.....
                                endTime = rdtsc();
                                ////////////////////////////////////////////////

        duration =endTime-initTime;
        average = Average(previousAverage,duration,j);
        variance = Variance(previousVariance,previousAverage,average,duration,j);
        estandarDesviation = sqrt(variance);
        errorFlag = C*estandarDesviation/sqrt(j);

                                //-----------------------------------------
        previousAverage = average;
        previousVariance = variance;
        j++;

                        }while( errorFlag > error || j<100);
     printf("\nBORRADO   longitud %d -- %LF - muestra %LF  promedio %LF  varianza %LF desviación estandar %lf criterio de paro
%LF",i,j,duration,average,variance,sqrt(variance),errorFlag);
             }// for

  return 0;
}
```

7.1.3 Implementación de búsquedas almacenando una tabla de ruteo real

Los proyectos que fueron elaborados para implementar las operaciones de búsqueda en los árboles binarios y los árboles Trie-Multibit con respaldo utilizando una tabla de ruteo real del sistema autónomo AS6447 son los siguientes.



El proyecto **BinaryTreeFTable** fue elaborado para poder cargar una tabla de ruteo real dentro de un árbol binario, también se encarga de generar direcciones IP aleatorias de 32 bits y realiza las búsquedas del prefijo BMP tomando las medidas del retardo en términos de ciclos reloj. El contenido de este proyecto es el siguiente.



Cada elemento del proyecto se describe en la tabla siguiente.

Elemento del proyecto	Descripción
Carpeta **arboles_binarios**	Contiene los algoritmos que se encargan de la inserción, borrado y búsqueda dentro de los árboles binarios.
Carpeta **prefix** y **prefijos**	Contiene los algoritmos necesarios para el manejo de los prefijos en formato binario.
Carpeta **rdtsc**	Contiene los algoritmos necesarios para realizar la medición de los ciclos de reloj de cada operación.
Carpeta **stadistic**	Contiene los algoritmos necesarios para llevar el manejo de la estadística de cada una de las muestras tomadas.
Archivo **fordwardingTable.txt**	Este archivo es generado por el proyecto **FordwardTable** y contiene la tabla de reexpedición ya en formato binario y con las inerfaces de red ya numeradas.
Archivo **MainFTable.c**	Contiene el programa principal encargado de realizar el almacenamiento de la tabla de reexpedición dentro de la estructura basada en árboles binarios, genera las direcciones IP de 32 bits y realiza las búsquedas de los prefijos BMP midiendo el tiempo de respuesta en términos de ciclos de reloj.

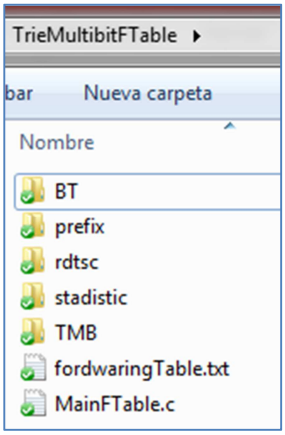
A continuación se muestra el código fuente del archivo **MainFTable.c**

```
/*
   OCTUBRE-2014
          AUTOR
          Israel De Olmos Ramírez
          UNIVERSIDAD AUTÓNOMA METROPOLITANA - IZTAPALAPA, MEXICO, DF.
          Maestría en Ciencias y Tecnologías  de la Información
 */

#include "./arboles_binarios/proto_AB.h"
#include "./prefijos/prefijo.h"
#include "./stadistic/stadistic.h"
#include "./prefix/prefix.h"
#include "./rdtsc/rdtsc.h"
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include<stdio.h>
```

```
#define path "./fordwaringTable.txt"

//////////////////////////////////////////
/// crea el árbol binario y almacena la tabla de reexpedición
/// indicada en la constante 'path'
void createFromTable(AB tree);
#define C 2.33      // C = 1.96 para tener una confianza del 95%  ;   0.95 = 1 - 2(1-Q(C))
                    // C = 2.33 para tener una confianza del 98%  ;   0.95 = 1 - 2(1-Q(C))

int error=1;    // Error que se propone, se debe cumplir que   C*desv_est/pow(n,0.5) <= error
                //por el teorema del límite central


long double initTime,endTime,duration; //variables necesarias para recolectar las estadisticas de ciclos de reloj

int main(){
        int i;
        long double random,j=1;
        long double average,previousAverage=0;
        long double variance,previousVariance=0;
        long double estandarDesviation,errorFlag;
        char binaryPrefix[40],ipAdress[40];
        long long int accesos_busca;
        long long int bandera;

        unsigned int seed = time(NULL);  //Se inicializa la semilla de la función aleatoria
        srand(seed);
        AB root;
        InfoAB raiz;
        raiz.interfaz=0;
        root=crea_nodoAB(raiz);

  createFromTable(root);
  printf("\nTabla %s cargada correctamente, presione enter para continuar...",path);
  getc(stdin);

  //.....................................................................................

  //---*************************************************************************************
        // calcula estadisticas de búsqueda
        //-------------------------------------------------------------------------------
                j=1;
                do{
                i=randomLength();           //Se selecciona la longitud del prefijo de forma aleatoria
                random = randomPrefix(pow(2,i-1));  //Se genera un prefijo aleatorio en forma de un número decimal
                                                    //respetando el rango válido de acuerdo a la longitud del prefijo
                printbin(random,i,binaryPrefix);     //El prefijo es escrito a su notación decimal asignando una interfaz ficticia
                generateIpAdress(binaryPrefix,ipAdress);   //Genera una IP válida para ser buscada dentro del árbol Binario

                ////////////////////////////////////////////
                // Ventana de medición de tiempo
                //----------------------------------------
                initTime =rdtsc();
                busca_b(ipAdress,root,0,&bandera,&accesos_busca);
                endTime = rdtsc();
                ////////////////////////////////////////////
                duration =endTime-initTime;
                average = Average(previousAverage,duration,j);
                variance = Variance(previousVariance,previousAverage,average,duration,j);
```

```
                    estandarDesviation = sqrt(variance);
                    errorFlag = C*estandarDesviation/sqrt(j);
                    //----------------------------------------
                    previousAverage = average;
                    previousVariance = variance;
                    j++;

                    }while( errorFlag > error || j<100);
                     printf("\nBUSQUEDA Arbol Binario con tabla  Ultima_longitud %d -- Experimentos %LF - Ultima_muestra
%LF          promedio     %LF          varianza     %LF          desviación estandar     %lf          criterio de paro
%LF\n",i,j,duration,average,variance,sqrt(variance),errorFlag);
  //.....................................................................................
  return 0;
}



////////////////////////////////////////////
/// crea el árbol binario y almacena la tabla de reexpedición
/// indicada en la constante 'path'
void createFromTable(AB tree){
        FILE *p = fopen(path,"r");// Abre la tabla de reexpedión
        char buffer[100];
        long long int accesos_inserta;

        ////////////////////////////////////////////////////////
        // Se valida la existencia de la tabla de reexpedición
        if(p!=NULL){
                printf("Creando arbol binario...\n");
        }else{
                printf("El archivo  %s no ha sido localizado.\n",path);
                return;
        }
        ////////////////////////////////////////////////////////////
        fgets(buffer,100,p); //se omite interfaz por defecto ya que esta se especifica en la construccion
                        // de cada nodo 'findBT' en BinaryTree.c
        ////////////////////////////////////////////////////////
        // Se llena el árbol con la tabla de reexpedición
        while(!feof(p)){
                fgets(buffer,100,p);
                tree = inserta_b(buffer,tree,0,&accesos_inserta,0);
        }
        printf("Árbol creado correctamente");
        return;
}
```

El proyecto **TrieMultibitFTable** fue elaborado para poder cargar una tabla de ruteo real dentro de un árbol Multibit con respaldo, también se encarga de generar direcciones IP aleatorias de 32 bits y realiza las búsquedas del prefijo BMP tomando las medidas del retardo en términos de ciclos reloj. El contenido de este proyecto es el siguiente.



Cada elemento del proyecto se describe en la tabla siguiente.

Elemento del proyecto	Descripción
Carpeta **BT**	Contiene los algoritmos que se encargan de la inserción, borrado y búsqueda dentro de los árboles binarios de respaldo.
Carpeta **prefix**	Contiene los algoritmos necesarios para el manejo de los prefijos en formato binario.
Carpeta **rdtsc**	Contiene los algoritmos necesarios para realizar la medición de los ciclos de reloj de cada operación.
Carpeta **stadistic**	Contiene los algoritmos necesarios para llevar el manejo de la estadística de cada una de las muestras tomadas.
Carpeta **TMB**	Contiene los algoritmos que se encargan de la inserción, borrado y búsqueda dentro de los árboles Trie-Multibit.
Archivo **fordwardingTable.txt**	Este archivo es generado por el proyecto **FordwardTable** y contiene la tabla de reexpedición ya en formato binario y con las inerfaces de red ya numeradas.
Archivo **MainFTable.c**	Contiene el programa principal encargado de realizar el almacenamiento de la tabla de reexpedición dentro de la estructura basada en árboles TrieMultibit con respaldo, genera las direcciones IP de 32 bits y realiza las búsquedas de los prefijos BMP midiendo el tiempo de respuesta en términos de ciclos de reloj.

A continuación se muestra el código fuente del archivo **MainFTable.c**

```
/*
   OCTUBRE-2014
            AUTOR
            Israel De Olmos Ramírez
            UNIVERSIDAD AUTÓNOMA METROPOLITANA - IZTAPALAPA, MEXICO, DF.
            Maestría en Ciencias y Tecnologías  de la Información
 */

#include "./stadistic/stadistic.h"
#include "./prefix/prefix.h"
#include "./rdtsc/rdtsc.h"
#include "./TMB/MultibitTrie.h"
#include <stdlib.h>
```

```
#include <time.h>
#include <math.h>
#include <stdio.h>

#define path "./fordwaringTable.txt"
void createFromTable(TMBNode *tree,int *strides);
#define C 2.33    // C = 1.96 para tener una confianza del 95% ;   0.95 = 1 - 2(1-Q(C))
                  // C = 2.33 para tener una confianza del 98% ;   0.95 = 1 - 2(1-Q(C))

int error=1;   // Error que se propone, se debe cumplir que   C*desv_est/pow(n,0.5) <= error
               //por el teorema del límite central

long double initTime,endTime,duration;
int strides[35];



int main(){
        int i;
        long double random,j=1;
        long double average,previousAverage=0;
        long double variance,previousVariance=0;
        long double estandarDesviation,errorFlag;
        char binaryPrefix[40],ipAdress[40];

        unsigned int seed = time(NULL);  //Se inicializa la semilla de la función aleatoria
        srand(seed);
        TMBNode *root;

        ////////////////////////////////////////////////////////
        // Se indican los strides del árbol
        strides[0]=8;
        strides[1]=8;
        strides[2]=8;
        strides[3]=8;
        strides[4]=-1;
        strides[5]=-1;
        strides[6]=-1;
        strides[7]=-1;
        strides[8]=-1;
        strides[9]=-1;
        strides[10]=-1;
        strides[11]=-1;
        strides[12]=-1;
        strides[13]=-1;
        strides[14]=-1;
        strides[15]=-1;
        strides[16]=-1;//bandera que indica el fin de los strides
        ////////////////////////////////////////////////////////
        root = createTMBNodes(1);   //Crea en nodo raiz del árbol TrieMultibit
  createFromTable(root,strides);
  printf("\nTabla %s cargada correctamente, presione enter para continuar...",path);
  getc(stdin);

  //.....................................................................................
  //---*************************************************************************************
  // calcula estadisticas de búsqueda
  //-------------------------------------------------------------------------------
                j=1;
```

```
                do{
                i=randomLength();           //Se selecciona la longitud del prefijo de forma aleatoria
                random = randomPrefix(pow(2,i-1));  //Se genera un prefijo aleatorio en forma de un número decimal
                                                    //respetando el rango válido de acuerdo a la longitud del prefijo
                //printf("\n-------------------------------------------------------------");
               //printf("\n longitud aleatoria: %d prefijo a buscar decimal: %LF",i,random);
                printbin(random,i,binaryPrefix);     //El prefijo es escrito a su notación decimal asignando una interfaz ficticia
                //printf("\n prefijo generado: %s",binaryPrefix);
                generateIpAdress(binaryPrefix,ipAdress);   //Genera una IP válida para ser buscada dentro del árbol Binario
                //printf("\n dirección generada: %s", ipAdress);
                ////////////////////////////////////////////
                // Ventana de medición de tiempo
                //----------------------------------------
                        initTime =rdtsc();
                        findTMB(0,strides,ipAdress,root);       //Se busca la dirección Ip dentro del árbol binario
                        endTime = rdtsc();
                 ////////////////////////////////////////////

                        duration =endTime-initTime;
                        average = Average(previousAverage,duration,j);
                        variance = Variance(previousVariance,previousAverage,average,duration,j);
                        estandarDesviation = sqrt(variance);
                        errorFlag = C*estandarDesviation/sqrt(j);
                        //----------------------------------------
                        previousAverage = average;
                        previousVariance = variance;
                        j++;

                        }while( errorFlag > error || j<100);
                         printf("\nBUSQUEDA Árbol MULTIBIT con tabla  Ultima_longitud %d -- Experimentos %LF - Ultima_muestra %LF  promedio %LF  varianza %LF  desviación estandar %lf  criterio de paro %LF\n",i,j,duration,average,variance,sqrt(variance),errorFlag);
   //.....................................................................................
   return 0;
}


////////////////////////////////////////////
/// crea el árbol binario y almacena la tabla de reexpedición
/// indicada en la constante 'path'
void createFromTable(TMBNode *tree,int* strides){
        FILE *p = fopen(path,"r");// Archivo de lectura
        char buffer[100];
        int length,interface;
        long long int nPrefix[35]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

        ////////////////////////////////////////////////////////
        // Se valida la existencia de la tabla de reexpedición
        if(p!=NULL){
                printf("Creando árbol TrieMultibit...\n");
         }else{
                printf("El archivo  %s no ha sido localizado.\n",path);
                return;
        }
        ////////////////////////////////////////////////////////////
        fgets(buffer,100,p); //se omite interfaz por defecto ya que esta se especifica en la construccion
                        // de cada nodo 'findBT' en BinaryTree.c
        ////////////////////////////////////////////////////////////
```

```
        // Se llena el árbol con la tabla de reexpedición
        while(!feof(p)){
                fgets(buffer,100,p);
                getLenInterface(&length,&interface,buffer);
                getNPrefix(strides,buffer,nPrefix);
                insertPrefixTMB(0,strides,strides[0],length,nPrefix,interface,buffer,tree);
        }

        printf("Árbol TrieMultibit creado correctamente");
        return;
}
```

Estos son a grandes rasgos los proyectos implementados al realizar la parte experimental de esta tesis, si se desea mayor detalle respecto a los códigos fuentes puede contactarse al autor mediante el e-mail israel.dor@hotmail.com.