Estructura de datos para organización y búsqueda de archivos

Isabela Muriel Roldan
Universidad Eafit
Colombia
imurielr@eafit.edu.co

Mateo Flórez Restrepo
Universidad Eafit
Colombia
mflorezr@eafit.edu.co

Juan Pablo Castaño Duque
Universidad Eafit
Colombia
jpcastanod@eafit.edu.co

Mauricio Toro
Universidad Eafit
Colombia
mtorobe@eafit.edu.co

RESUMEN

El objetivo de este informe es analizar y solucionar el problema de la búsqueda eficiente de archivos y subdirectorios almacenados en un directorio. Lo que se busca es encontrar y utilizar una estructura de datos que permita guardar los archivos de manera organizada con el fin de que, a la hora de consultarlos el sistema los encuentre en el menor tiempo posible.

La solución a este problema es de gran importancia debido a que a medida que la tecnología avanza se necesita almacenar cada vez mayor número de información y si esta no es guardada de manera organizada se pueden llegar a generar grandes molestias debido a la poca capacidad de memoria y la ineficiencia a la hora de consultar los archivos. Existen varios problemas similares al que se plantea en este informe y algunos de estos serán analizados con el propósito de hallar una solución más efectiva.

Finalmente se decide solucionar el problema por medio de una estructura de datos llamada árbol AVL, la cual permitió realizar la inserción y búsqueda de información de manera casi instantánea, con una complejidad muy baja, lo que favorece los objetivos mencionados anteriormente. Al final se logró concluir que existen estructuras de datos que no son tan eficientes para la solución de problemas de este tipo, por ejemplo, las tablas de hash, debido a la gran cantidad de colisiones que se pueden causar por el desconocimiento de la cantidad exacta de información que se desea analizar.

Palabras clave

Estructura de datos; directorio; fichero; tabla de hash; complejidad; tiempos de ejecución; memoria; operaciones; notación O; árbol AVL; colisión

Palabras clave de la clasificación de la ACM

Software and its engineering → Software organization and properties → Operating systems → File systems management

Theory of computation → Data structures design and analysis → Sorting and searching

1. INTRODUCCIÓN

En el día a día, en cuanto a la computación y tecnología respecta, se busca un progreso constante para que cada vez las cosas sean más útiles, más optimas y de un uso mucho más sencillo para el usuario, lo que trae como consecuencia que los programas se vayan haciendo obsoletos a medida que las tecnologías avanzan.

Un gran ejemplo de lo mencionado anteriormente son los sistemas de archivos, este es el encargado de almacenar los archivos de un sistema operativo, pero como todos los programas tiene sus limitaciones, una de ellas es la cantidad de archivos que puede almacenar ya que lo que en un tiempo pudo ser una cifra exorbitante hoy es una cifra insuficiente para la cantidad de datos que se manejan, por lo tanto se necesita encontrar una manera eficiente de almacenar estos archivos e incluir otras funciones que sean útiles para el mismo, tales como la búsqueda. Este es el objetivo que tiene este proyecto.

2. PROBLEMA

El problema al cual nos enfrentamos se basa en crear a través de una estructura de datos de archivos eficiente en cuanto a las especificaciones del cliente para encontrar fácilmente todos aquellos archivos, ficheros y subdirectorios que se encuentren listados dentro de un directorio.

Resolver este problema sería un avance en cuanto a los sistemas de archivos y almacenamiento que maneja un sistema operativo, sobre todo para aquellas grandes empresas que trabajan una gran cantidad de información importante.

3. TRABAJOS RELACIONADOS

3.1 Tablas de hash

Las tablas de hash son estructuras de datos utilizadas para almacenar gran cantidad de datos que luego requieran ser buscados e insertados (permite almacenar y recuperar) en un sistema de archivos muy eficiente. Una tabla de hash almacena un conjunto de pares "(clave, valor)". La clave es única para cada elemento de la tabla y es el dato que se utiliza para buscar un determinado valor. Un contenedor asociativo.

La implementación de una tabla de hash está basada en los siguientes elementos:

- Una tabla de un tamaño razonable para almacenar los pares (clava, valor)
- Una función "hash" que recibe la clave y devuelve un índice para acceder a una posición de la tabla
- Un procedimiento para tratar los casos en los que la función anterior devuelve el mismo índice para dos

claves distintas. Esta situación se conoce con el nombre de colisión.

Las posibles implementaciones de cada uno de estos tres elementos se traducen en una infinidad de formas de implementar una tabla de hash.

Se debe tener cuidado con el tamaño que requiere una tabla de hash de acuerdo al tipo de aplicación y su implementación, ya que si es muy pequeña puede llegar a convertirse en una lista encadenada, en cuyo caso la búsqueda puede ser muy ineficiente ya que se debe pasar por cada valor, y si por el contrario la tabla es enorme, debido a las posiciones vacías que quedaran la cantidad de memoria se malgasta innecesariamente.

3.2 Árbol rojo negro

El problema con algunas estructuras de datos binarias es que en la mayoría de los casos un lado del árbol queda más sobrecargado que el otro, lo que hace que, a la hora de realizar el proceso de búsqueda, se alargue el camino a recorrer para hallar los datos y el tiempo de espera es mucho mayor. El árbol rojo negro consiste en un árbol de búsqueda binaria, esto significa que agrupa los valores en dos grupos, a la izquierda los menores y a la derecha los mayores, pero la diferencia entre este y otros árboles binarios es que a este se le incorpora un bit extra de almacenamiento por nodo, lo que permite determinar si un nodo es rojo o negro según las propiedades de esta estructura. Con ayuda de dichas propiedades y el color de cada nodo, se determina un orden de los datos que va cambiando a media que se agrega nueva información, de manera que el árbol queda equilibrado y acorta el camino a la hora de buscar un dato determinado.

3.3 Árbol B

Es un árbol de búsqueda que puede estar vacío o aquel cuyos nodos pueden tener varios hijos, existiendo una relación de orden entre ellos, tal como muestra la Figura 1.

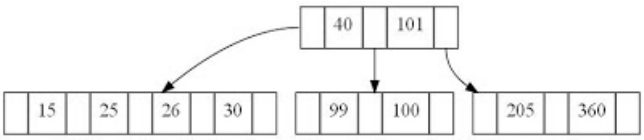


Figura 1: Grafica Árbol B.

Este árbol tiene una organización ascendente, siempre el número a la izquierda es menor que el de la derecha.

Propiedades:

- Es un árbol multicaminos.
- Todas sus hojas están en el mismo nivel.
- Los datos se llaman claves y se almacenan en páginas
- "M" es el grado del árbol y es el máximo número de hijos que puede tener cualquier página.
- "M-l" es el número de claves que puede tener cualquier página.
- "M-1/2" es el número mínimo de claves que puede tener cualquier página.

Procesos que puede realizar el árbol:

Búsqueda:

1. Situarse en el nodo raíz.
2. (*) Comprobar si contiene la clave a buscar.
    1. Encontrada fin de procedimiento.
    2. No encontrada:
        1. Si es hoja no existe la clave.
        2. En otro caso el nodo actual es el hijo que corresponde:
            1. La clave a buscar k < k1: hijo izquierdo.
            2. La clave a buscar k > ki y k < ki+1 hijo iésimo.
            3.Volver a paso 2(*).

Inserción:

1. Realizando una búsqueda en el árbol, se halla el nodo hoja en el cual debería ubicarse el nuevo elemento.

2. Si el nodo hoja tiene menos elementos que el máximo número de elementos legales, entonces hay lugar para uno más. Inserte el nuevo elemento en el nodo, respetando el orden de los elementos.

3. De otra forma, el nodo debe ser dividido en dos nodos. La división se realiza de la siguiente manera:

    1. Se escoge el valor medio entre los elementos del nodo y el nuevo elemento.

    2. Los valores menores que el valor medio se colocan en el nuevo nodo izquierdo, y los valores mayores que el valor medio se colocan en el nuevo nodo derecho; el valor medio actúa como valor separador.

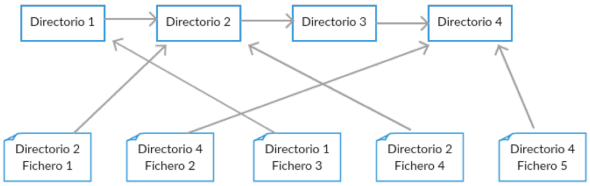
    3. El valor separador se debe colocar en el nodo padre, lo que puede provocar que el padre sea dividido en dos, y así sucesivamente.

3.4 Árbol B+

Es una estructura de datos tipo árbol que representa una colección de datos ordenados y que además permite la inserción y eliminación de manera eficiente de los elementos que se encuentran en la colección. A diferencia del Árbol B ya mencionado anteriormente y siendo una variante del mismo en un árbol B+, toda la información se guarda en las hojas. Los nodos internos sólo contienen claves y punteros.

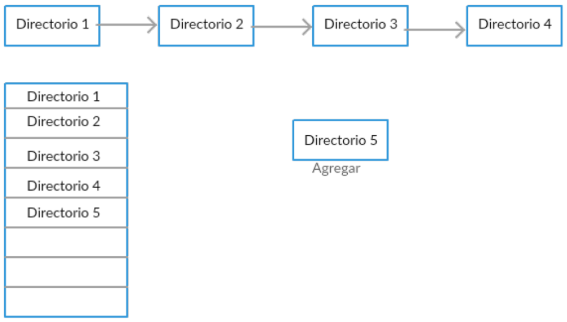
Todas las hojas se encuentran en el mismo nivel, que corresponde al más bajo. Los nodos hoja se encuentran unidos entre sí como una lista enlazada para permitir principalmente recuperación en rango mediante búsqueda secuencias.

4. ESTRUCTURA DE DATOS

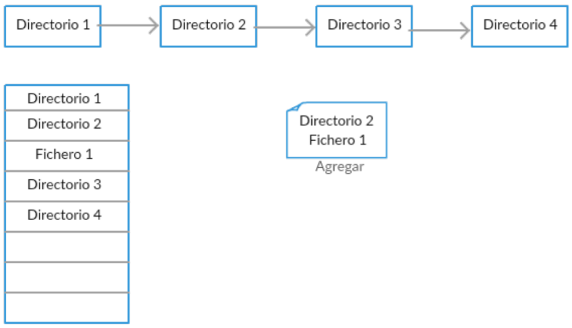


Gráfica 1: Lista de directorios, al agregar un fichero nuevo se recorre el archivo hasta encontrar el directorio correspondiente y se agrega.

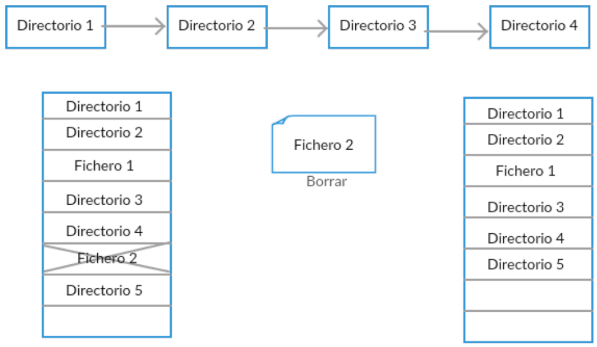
4.1 Operaciones de la estructura de datos



Gráfica 2: Representación de agregado de un directorio



Gráfica 3: Representación de agregado de un fichero



Gráfica 4: Representación de borrado de un fichero

4.2 Criterios de diseño de la estructura de datos

Elegimos esta estructura de datos debido a que es una que puede satisfacer de una manera correcta todos los objetivos específicos que se proponen en el trabajo.

Además, este es un método eficiente para la organización de ficheros, comparando con el tiempo de proceso que tardan otras estructuras debido a que su complejidad es menor en varios métodos a comparación.

Otra de las ventajas que presenta esta estructura de datos es que puede guardar los cambios de manera permanente en un archivo de texto, es decir, lo que el usuario haga en el transcurso del programa puede revisarlo cuando desee así el programa no esté en ejecución y de la misma manera puede usar el programa para modificar ese mismo directorio que creó hace X tiempo.

También esta estructura de datos tiene un orden establecido de entrada, (se organizan en el instante que entran), lo que hace que al momento que el usuario quiera entrar al primer directorio que ingresó o al último, tenga una complejidad de O(1) en el caso promedio, lo que significa que es muy eficiente, al igual que vaciar, ya que no tiene que buscar donde están los datos, si no que borra el archivo sin importar los datos que posea y crea uno nuevo para permitir que el usuario pueda seguir trabajando con el programa sin necesidad de hacer nada por fuera de este.

4.3 Análisis de Complejidad

Método	Complejidad
Vaciar	O(1)
AgregarDirectorio	O(1)
AgregarFichero	O(n*m)
BuscarFichero	$O(n^2)$
BorrarFichero	O(n*m)

Tabla 1: Tabla para reportar la complejidad en el peor caso

4.4 Tiempos de Ejecución

Conjunto de datos	1	2	3
Agregar directorios	5,250 ms	13,578 ms	12,173ms
Agregar fichero	4,186 ms	17,085 ms	13,862 ms
Buscar fichero	5,602 ms	8,953 ms	5,774 ms
Borrar fichero	6,731 ms	7,624 ms	6,945 ms
Vaciar proyecto	6,970 ms	6,465 ms	4,792 ms

Tabla 2: Tiempos de ejecución de las operaciones de la estructura de datos con diferentes conjuntos de datos

4.5 Memoria

Conjunto de datos	1	2	3
Consumo de memoria	3,8212 MB	6,845264 MB	7,607904 MB

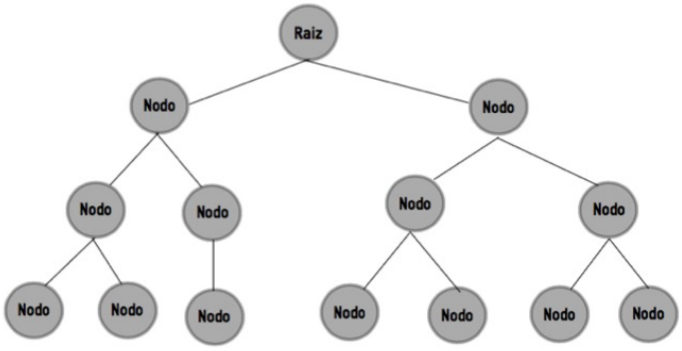
Tabla 3: Consumo de memoria de la estructura de datos con diferentes conjuntos de datos

4.6 Análisis de los resultados

Conjunto de datos	Mejor Tiempo	Peor Tiempo	Tiempo promedio
Agregar directorios	4,677 ms	25,038 ms	10,244 ms
Agregar fichero	3,098 ms	26,654 ms	12,249 ms
Buscar fichero	3,126 ms	11,817 ms	6,776 ms
Borrar fichero	4,896 ms	9,518 ms	7,100 ms
Vaciar proyecto	4,219 ms	8,934 ms	6,079 ms

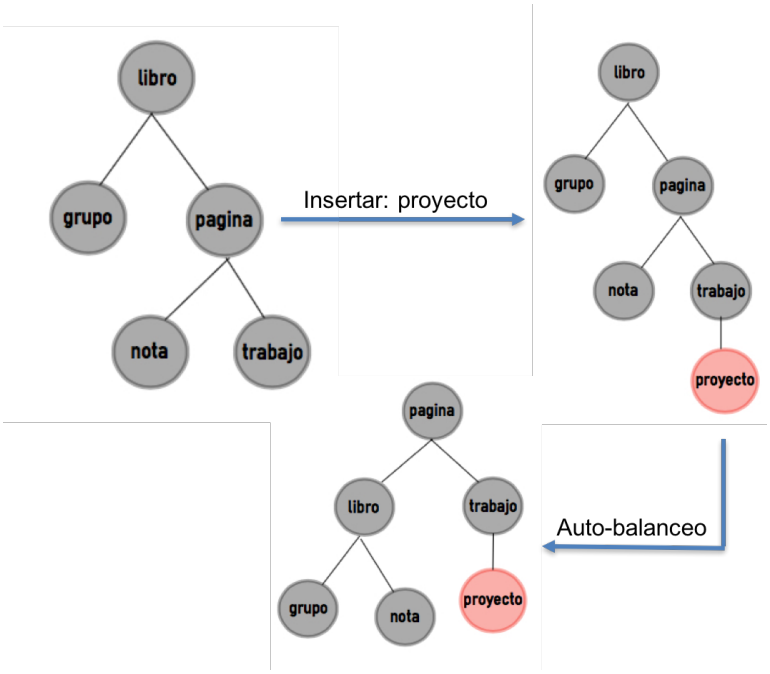
Tabla 4: Análisis de los resultados obtenidos con la implementación de la estructura de datos

5. ÁRBOL AVL

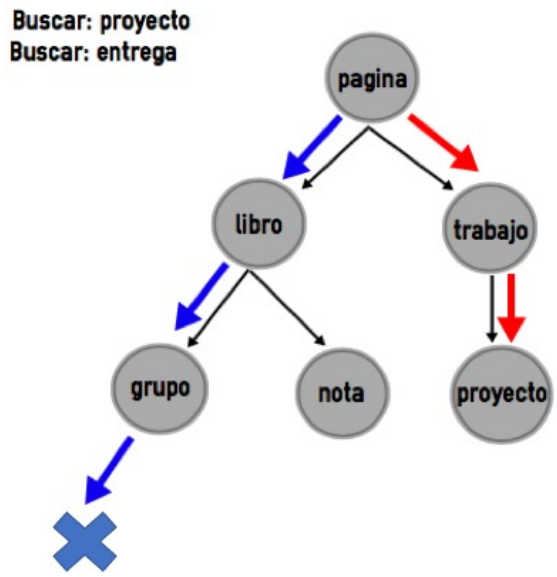


Gráfica 5: Árbol AVL que almacena directorios y ficheros alfabéticamente

5.1 Operaciones de la estructura de datos



Gráfica 6: Imagen de una operación de inserción al árbol, en la que se auto balancea si es necesario y organiza las palabras alfabéticamente, donde las letras menores van a la izquierda y las mayores a la derecha



Gráfica 7: Imagen de una operación de búsqueda en el árbol, que evalúa las palabras alfabéticamente y a partir de esto las busca rápidamente

5.2 Criterios de diseño de la estructura de datos
Esta estructura de datos organiza de una manera eficiente los datos ingresados y permite su manipulación, como el ingreso o la búsqueda de datos de una manera bastante rápida y eficaz.

La estructura de datos tiene un sistema de autobalance, lo que significa que sus nodos siempre estarán a la misma altura "n" o "n+1" lo que disminuye de una manera muy notable la complejidad de los algoritmos más importantes tales como la búsqueda y por ende el tiempo de ejecución y el gasto en el espacio de memoria de los mismos.

Esta estructura de datos si en el momento de usar el método de inserción se ve afectado el estado de equilibrio realiza rotaciones a la derecha y a la izquierda en los nodos para reorganizar el árbol, recuperar su balanceo y así no afectar la complejidad de los métodos de manera negativa.

La estructura de datos mantiene las operaciones del ABB tales como buscar un elemento, insertar un elemento, movimientos a través del árbol, etc. Pero su nueva operación es el autobalance, la cual es un gran diferencial con otras estructuras, debido a que reduce a O(log n) los métodos más importantes tales como la búsqueda o la inserción.

5.3 Análisis de la Complejidad

Método	Complejidad
Buscar	O (log n)
Insertar	O (log n)
Leer archivo	O (n log n))

Tabla 5: Tabla para reportar la complejidad en el peor de los casos

5.4 Tiempos de Ejecución

	ejemplito.txt	treeEtc.txt	juegos.txt
Lectura e inserción	1,15 ms	17,2 ms	345,14 ms
Búsqueda por nombre	0,81 ms	0,02 ms	0,47 ms

Tabla 6: Tiempos de ejecución de las operaciones de la estructura de datos con diferentes conjuntos de datos

5.5 Memoria

	ejemplito.txt	treeEtc.txt	juegos.txt
Consumo de memoria	5 MB	7MB	14 MB

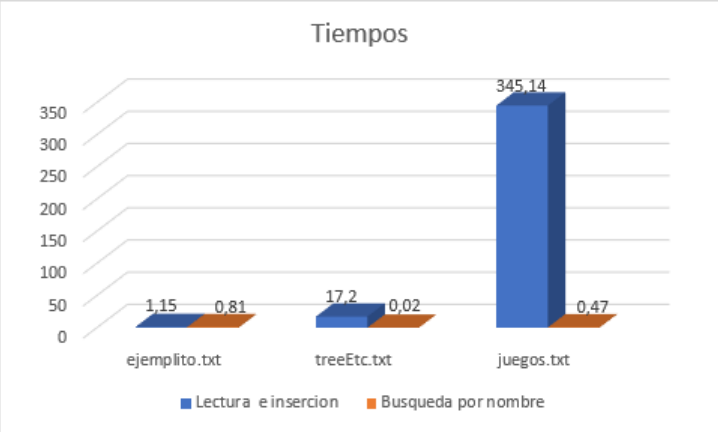
Tabla 7: Consumo de memoria de la estructura de datos con diferentes conjuntos de datos

5.6 Análisis de los resultados

	Mejor Tiempo	Peor Tiempo	Tiempo Promedio
Lectura e insercion	1,15 ms	345,14 ms	121,16 ms
Busqueda por nombre	0,02 ms	0,81 ms	0,43 ms

	Mejor Memoria	Peor Memoria	Memoria Promedio
Lectura e insercion	5 MB	14 MB	8,6 MB
Busqueda por nombre			

Tabla 8: Tabla de valores durante la ejecución



Gráfica 8: Grafica de análisis de tiempos según conjuntos de datos.

Para la estructura de datos que diseñamos (árbol AVL), existen dos algoritmos que corresponden a la función del programa:

Lectura e inserción: Para este algoritmo aparte de leer los datos de un archivo de texto, se utiliza el método insertar del árbol. Según el análisis en el peor de los casos este algoritmo se demora 345,14 ms en leer el archivo, línea por línea e insertar los ficheros y/o directorios al árbol, debido a que el conjunto de datos que utilizamos en este caso "juegos.txt" contiene más de 64.000 archivos y gasta más memoria. Y en el mejor de los casos fue de 1,15 ms para el conjunto de datos "ejemplito.txt", ya que gasta menos memoria. Sabemos por teoría que la inserción y rebalanceo del árbol contiene una complejidad de O(log n) y en el método de lectura debido a que inserta n archivos que se encuentran en el conjunto de archivos puede llegar a ser en el peor de los casos O(n log n).

Búsqueda: Luego de estar insertados en el árbol la búsqueda se vuelve más fácil. La complejidad en el peor de los casos fue 0,81 ms, y por teoría sabemos que la función búsqueda en el peor de los casos es O(log n).

Concluimos que la estructura de datos que diseñamos es muy rápida para insertar, leer y buscar, las cuales son las funciones más importantes para el problema que estamos resolviendo, por tanto, es eficiente para ser una alternativa de solución.

6. CONCLUSIONES

En este reporte se trataron varias estructuras de datos con el fin de encontrar la más viable para cumplir con los objetivos del proyecto, al comienzo se consideraron las tablas de hash, los arboles rojo negro y los B+, sin embargo, a lo largo del informe se descubrió que estos, en especial la tabla de hash no era la mejor solución debido a las colisiones que se presentan y al problema de no saber exactamente cuántos ficheros y directorios se desean almacenar.

Al momento de realizar la solución final descubrimos que los árboles AVL son una estructura de datos muy útil debido a la

poca complejidad a la hora de insertar y buscar, además, a diferencia de un árbol binario, los AVL tienen la función de auto-balancearse y de esta forma evita la sobrecarga en ambos lados del árbol lo cual sería un gran problema puesto que el tiempo de búsqueda e inserción sería mayor.

En la primera solución presentada se realizó una estructura de datos que no era muy eficiente debido a que cada vez que deseaba insertar o buscar, el programa debía leer todo el archivo de texto, luego con la solución final logramos que solo sea necesario leer el archivo de texto una vez y almacenar los datos en el árbol AVL, de esta forma cuando se desee ingresar o buscar solo se requiere recorrer el árbol, no el archivo completo.

En trabajos futuros, si se deseara mejorar este proyecto, se agregarían nuevas funciones a la hora de buscar como, por ejemplo, que al buscar un directorio aparezca que ficheros están en el interior; esta era una función que deseábamos incluir en el proyecto actual pero que debido a las complicaciones en la diferenciación de ficheros y directorios se nos hizo muy difícil lograrlo. Además, nos gustaría hacer un buscador que sea más funcional y parecido al de la plataforma de Windows o macOS, los cuales son muy rápidos y tienen la posibilidad de, no solo buscar ficheros y directorios, si no también aplicaciones o búsquedas recientes de internet.

6.1 Trabajos futuros

En el futuro nos gustaría poder agregar nuevas funcionalidades al programa, tales como la búsqueda independiente de ficheros y directorios, para lo cual habría que crear dos árboles independientes en los que se almacenen, en el primero ficheros, y en el segundo directorios. Además, nos gustaría que al buscar un directorio el programa muestre los ficheros que están en el interior de este y que al buscar ficheros diga en que directorio se encuentra.
Otro objetivo más a futuro, sería poder buscar por memoria que ocupa un fichero o por la fecha en la que se agregó. Por otra parte, nos gustaría lograr que al realizar una búsqueda no sea necesario escribir la palabra completa, sino que con solo buscar una parte de la palabra aparezcan todos los ficheros y directorios que la contengan.

AGRADECIMIENTOS
Esta investigación fue soportada parcialmente por el icetex.

REFERENCIAS

1. Stalin, F. Árbol rojo negro la evolución de los árboles binarios de búsqueda. Recuperado agosto 10, 2017, de Computer science in esmeraldas. http://computerscienceesmeraldas.blogspot.com.co/2016/03/arbol-rojo-negro-la-evolucion-de-los.html.

2. Universidad Carlos III de Madrid. Tablas hash. Recuperado agosto 12, 2017, del Departamento de ingeniería telemática. http://www.it.uc3m.es/abel/as/MMC/M2/HashTable_es.html.

3. Wikipedia. Árbol-B. Recuperado agosto 11, 2017. https://es.wikipedia.org/wiki/Árbol-B.

4. Wikipedia. Árbol B+. Recuperado agosto 11, 2017. https://es.wikipedia.org/wiki/Árbol_B%2B.

5. Solis, S. Árboles B. Recuperado agosto 12, 2017, de youtube https://www.youtube.com/watch?v=EbiFlTGh0rI