ESTRUCTURA DE DATOS PARA BUSCAR CONTENIDOS EN UN DIRECTORIO

Kevin Arley Parra
Universidad EAFIT
Colombia
kaparrah@eafit.edu.co

Daniel Alejandro Mesa Arango
Universidad EAFIT
Colombia
damesaa@eafit.edu.co

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

RESUMEN
El problema se resume a un sistema que pueda buscar elementos dentro de un directorio de una forma eficiente sin importar el tamaño de éste. La importancia de este problema radica en la gran cantidad de datos que se manejan en la actualidad en un mundo digitalizado y donde se busca dar la mejor atención y producto a los usuarios, algunos problemas que pueden relacionar son por ejemplo, que tan eficiente es una estructura de datos en la que se comparan datos para dar una respuesta a una solicitud como buscar una placa de automóvil en una base de datos con gran cantidad de información, donde se debe de usar una estructura de datos adecuada para que el programa no tarde demasiado en encontrar lo que se requiere, un árbol rojo negro es una de las soluciones para el problema mencionado, en el cual podemos realizar búsquedas muy rápidas y cumplir así con el propósito de encontrar un elemento en un directorio,, logrando buenos tiempos de ejecución.

PALABRAS CLAVE

Estructura de datos → árbol → árbol rojo negro → eficiencia → búsqueda rápida → búsqueda → java → directorios → archivos → búsqueda de archivos → búsqueda de directorios → árbol binario → árbol de búsqueda.

PALABRAS CLAVE DE LA CLASIFICACIÓN DE LA ACM

Funcionalidad → Técnicas de diseño de software → Teoría de la computación → Computabilidad → Funciones recursivas → Cálculo probabilístico → Cálculos de proceso → Clases de complejidad → Abstracción → Diseño y análisis de algoritmos → Análisis de algoritmos gráficos → Algoritmos de programación → Diseño y análisis de estructuras de datos → Técnicas de diseño de algoritmos → Divide y conquistaras → Programación dinámica.

1. INTRODUCCIÓN

Hoy en un mundo tan digitalizado se hace necesario tener los datos almacenados y que puedan ser consultados en cualquier momento por un usuario, sin embargo con el creciente volumen de datos muchas de las tecnologías y estructuras de datos han quedado obsoletas causando que los tiempos de espera para guardar algo en determinado subdirectorio o consultar algo sea mayor y consuma muchos recursos de la máquina, por esto han surgido nuevas soluciones que hacen mucho mejor el trabajo, varias son las soluciones que se pueden implementar las cuales traen sus ventajas y desventajas, sistemas como la búsqueda que ofrece Windows al usuario son ejemplos de implementación de una estructuras de datos, el usado por Microsoft es NTFS, implementa los árboles-B[1]. Se muestra entonces cuatro posibles soluciones y una por la cual optar considerándola la mejor para el caso.

2. PROBLEMA

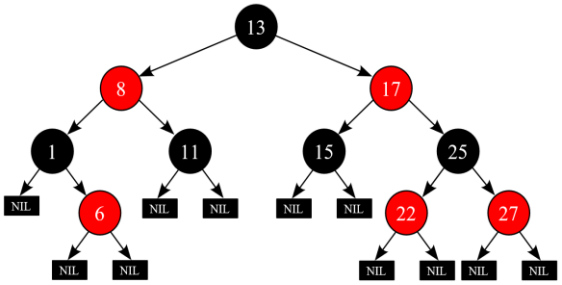
El problema consiste en buscar los archivos ó el archivo en un directorio de manera eficiente, que se puedan encontrar estos fácilmente, sin que el usuario tenga que esperar mucho para que se encuentre lo que desea.

3. TRABAJOS RELACIONADOS

Para el actual informe se realizó una investigación sobre las diferentes estructuras de datos que pueden dar una solución efectiva al problema planteado, a continuación, una descripción de cada una de ellas.

3.1. Árbol rojo negro

Concretamente es un árbol binario de búsqueda equilibrado, la estructura original fue creada por Rudolf Bayer en 1972[2], es complejo, pero tiene un buen peor caso de tiempo de ejecución para sus operaciones y es eficiente. Puede buscar, insertar y borrar en un tiempo O(log n), donde n es el número de elementos del árbol.



Gráfica 1. Ejemplo de un árbol rojo negro.

Además de los requisitos impuestos a los árboles binarios de búsqueda convencionales, se deben satisfacer las siguientes reglas para tener un árbol rojo-negro válido:

1. Todo nodo es o bien rojo o bien negro.
2. La raíz es negra.

3. Todas las hojas (NULL) son negras.

4. Todo nodo rojo debe tener dos nodos hijos negros.

5. Cada camino desde un nodo dado a sus hojas descendientes contiene el mismo número de nodos negros.

Estas reglas producen una regla crucial para los árboles rojo-negro: el camino más largo desde la raíz hasta una hoja no es más largo que dos veces el camino más corto desde la raíz a una hoja. El resultado es que dicho árbol está aproximadamente equilibrado.

Dado que las operaciones básicas como insertar, borrar y encontrar valores tienen un peor tiempo de ejecución proporcional a la altura del árbol, esta cota superior de la altura permite a los árboles rojo-negro ser eficientes en el peor caso, a diferencia de los árboles binarios de búsqueda.

3.2. Árbol B

Los B-árboles surgieron en 1972 creados por R.Bayer y E.McCreight[3]. El problema original comienza con la necesidad de mantener índices en almacenamiento externo para acceso a bases de datos, es decir, con el grave problema de la lentitud de estos dispositivos se pretende aprovechar la gran capacidad de almacenamiento para mantener una cantidad de información muy alta organizada de forma que el acceso a una clave sea lo más rápido posible. Los B árboles son árboles cuyos nodos pueden tener un número múltiple de hijos tal como muestra el esquema de uno de ellos en la gráfica 2.

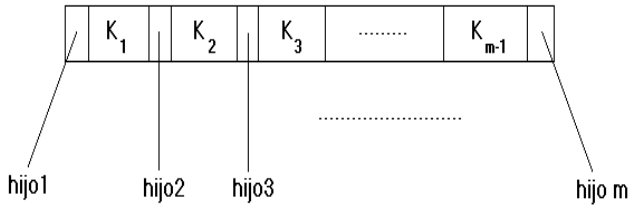


Figura 1:Esquema de un nodo de un arbol B de orden m.

Gráfica 2. Ejemplo de un árbol B

Como se puede observar en la Gráfica 2, un B árbol se dice que es de orden m si sus nodos pueden contener hasta un máximo de m hijos. En la literatura también aparece que si un árbol es de orden m significa que el mínimo número de hijos que puede tener es m+1(m claves).

1. Seleccionar como nodo actual la raíz del árbol.

2. Comprobar si la clave se encuentra en el nodo actual:
3. Si la clave está, fin.
4. Si la clave no está:

- Si estamos en una hoja, no se encuentra la clave. Fin.
- Si no estamos en una hoja, hacer nodo actual igual al hijo que corresponde según el valor de la clave a buscar y los valores de las claves del nodo actual (i buscamos la clave K en un nodo con *n* claves: el hijo izquierdo si $K<K_1$,el hijo derecho si $K>K_n$ y el hijo i-ésimo si $K_i<K<K_{i+1}$)y volver al segundo paso.

Breve explicación de la inserción

Las inserciones se hacen en los nodos hoja. [4]

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.

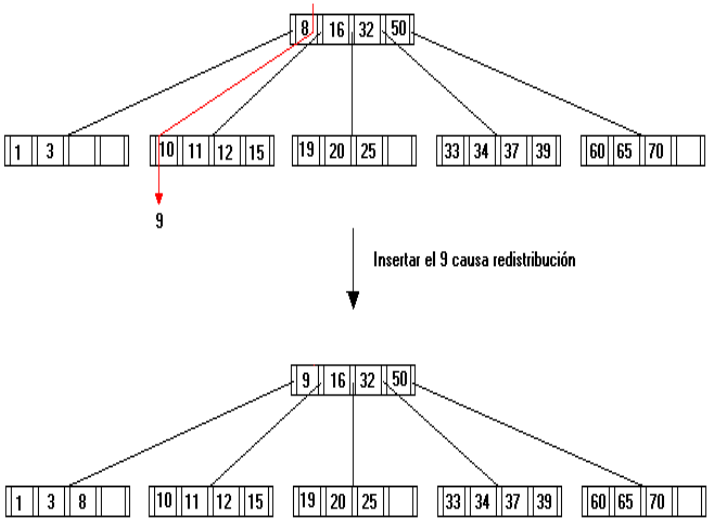
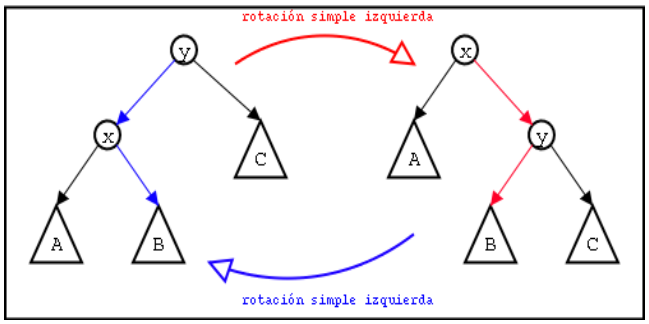


Figura 3:Inserción con redistribución.

Gráfica 3. Ejemplo de inserción en un árbol B.

3.3. Árbol AVL

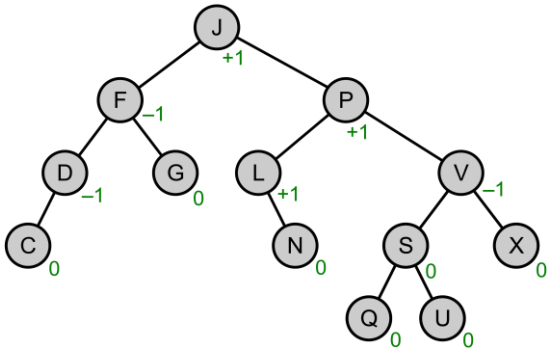
Un árbol AVL es un árbol binario de búsqueda que cumple con la condición de que la diferencia entre las alturas de los subárboles de cada uno de sus nodos es, como mucho 1[5]. La denominación de árbol AVL viene dada por los creadores de tal estructura (Adelson-Velskii y Landis). Recordamos que un árbol binario de búsqueda es un árbol binario en el cual cada nodo cumple con que todos los nodos de su subárbol izquierdo son menores que la raíz y todos los nodos del subárbol derecho son mayores que la raíz. Recordamos también que el tiempo de las operaciones sobre un árbol binario de búsqueda son O(log n) promedio, pero el peor caso es O(n), donde n es el número de elementos.



Gráfica 4. Ejemplo de rotación en un árbol AVL.

El árbol de la Gráfica 4. es un árbol de búsqueda, se debe cumplir x < y, y además todos los nodos del subárbol A deben ser menores que x y y; todos los nodos del subárbol B deben ser mayores que x pero menores que y; y todos los nodos del subárbol C deben ser mayores que y, y por lo tanto que x.

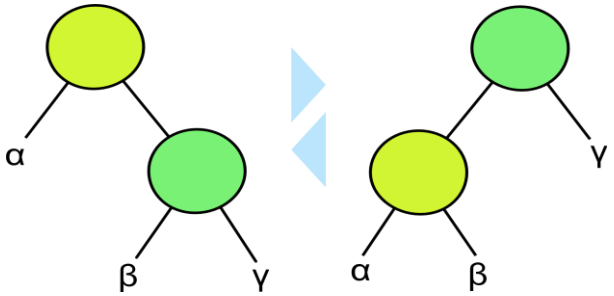
En la Gráfica 4 se ha modificado sencillamente el árbol. Como puede verificarse fácilmente por las desigualdades descritas en el párrafo anterior, el nuevo árbol sigue manteniendo el orden entre sus nodos, es decir, sigue siendo un árbol binario de búsqueda. A esta transformación se le denomina rotación simple (o sencilla). A continuación, se muestra un ejemplo sencillo de un árbol AVL:



Gráfica 5. Ejemplo de un árbol AVL.

3.4. Árbol biselado

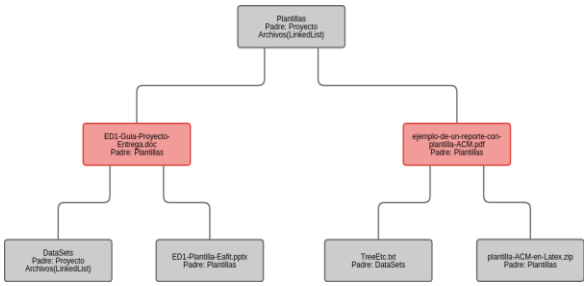
Un Árbol biselado o Árbol Splay es un Árbol binario de búsqueda auto balanceable[6], con la propiedad adicional de que a los elementos accedidos recientemente se accede más rápidamente en accesos posteriores. Realiza operaciones básicas como pueden ser la inserción, la búsqueda y el borrado en un tiempo del orden de O(log n). Para muchas secuencias no uniformes de operaciones, el árbol biselado se comporta mejor que otros árboles de búsqueda, incluso cuando el patrón específico de la secuencia es desconocido. Esta estructura de datos fue inventada por Robert Tarjan y Daniel Sleator. Todas las operaciones normales de un árbol binario de búsqueda son combinadas con una operación básica, llamada biselación. Esta operación consiste en reorganizar el árbol para un cierto elemento, colocando éste en la raíz. Una manera de hacerlo es realizando primero una búsqueda binaria en el árbol para encontrar el elemento en cuestión y, a continuación, usar rotaciones de árboles de una manera específica para traer el elemento a la cima. Alternativamente, un algoritmo "de arriba abajo" puede combinar la búsqueda y la reorganización del árbol en una sola fase.



Gráfica 6. Ejemplo de rotación en un árbol biselado.

4. ESTRUCTURA DE DATOS PARA BÚSQUEDAS BASADA EN UN ÁRBOL ROJO NEGRO

La estructura de datos que hemos diseñado podría tratarse como dos estructuras, por una parte, se tiene un TreeMap el cual se usa para la búsqueda de los archivos, y por otra se tiene una implementación en la cual hay dos clases, “Archivo” y “Carpeta”, que heredan a su vez de una clase llamada “AbstractClass”. Se usa un TreeMap para la búsqueda ya que de esta manera podemos asegurar una mejor complejidad en la búsqueda. Mientras que la forma de construir las carpetas y archivos en el sistema que implementamos permite saber la carpeta contenedora de un archivo y tener los archivos que tiene cada carpeta.



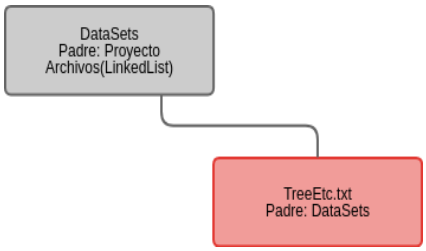
Gráfica 7: Árbol rojo negro de archivos y carpetas, un archivo y carpeta son clases que contienen un padre y un nombre o si hay iguales será una LinkedList con todos los iguales.

4.1 Operaciones de la estructura de datos

Creación: Para generar la estructura en el TreeMap se usa el nombre del archivo. Por ejemplo, si tenemos esta estructura en un archivo txt que luego leemos con el sistema:

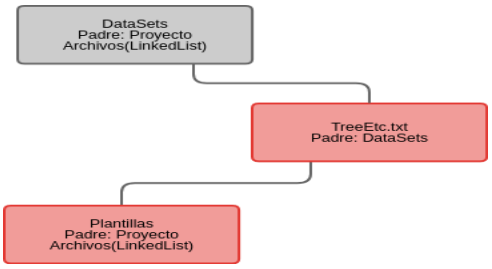
```
Proyecto/
├── [mauriciotoro 4.0K]  DataSets
│   └── [mauriciotoro 252K]  TreeEtc.txt
└── [mauriciotoro 4.0K]  Plantillas
```

El sistema empezará a agregar primero DataSets y luego agrega TreeEtc.txt, como lo muestra la gráfica 8:



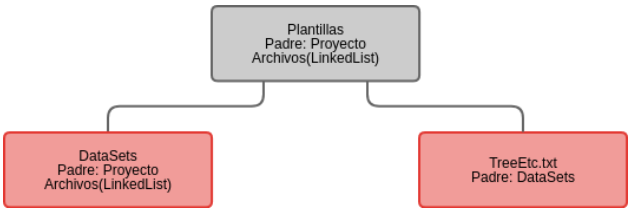
Gráfica 8: Imagen de la creación del árbol rojo negro.

Luego agrega Plantillas, la cual es mayor, por así decirlo, según el orden lexicográfico a DataSets, entonces va al lado derecho, pero Plantillas es menor que TreeEtc.txt, entonces lo inserta al lado izquierdo de TreeEtc.txt, como lo muestra la gráfica 9:



Grafica 9: Imagen de la inserción de un elemento en el árbol.

Sin embargo, vemos que el árbol está desbalanceado según las reglas del árbol rojo negro, por lo que él se balancea, como se muestra la gráfica 10:



Grafica 10: Balanceo del árbol rojo negro en la implementación realizada.

Para buscar un archivo, por ejemplo TreeEtc.txt, el sistema toma el nombre del archivo y lo busca en el TreeMap. Primero se para en el primer nodo, que sería la raíz, como TreeEtc.txt está después de Plantillas, en orden lexicográfico, entonces busca a la izquierda, donde efectivamente está el archivo buscado, como lo muestra la Grafica anterior.

Otras consideraciones a tener en cuenta sobre el diseño de la estructura, es que, en el TreeMap, no se presenta la misma jerarquía de directorios que se ingresa mediante la lectura del archivo, por lo que si en todo el directorio, hay varios archivos con el mismo nombre en diferentes carpetas, al ingresar la primera instancia de uno de estos, lo guardara sin problema, pero luego, al ingresar la segunda instancia tendremos un problema, esto sería una colisión. La solución que planteamos fue, en pocas palabras, la siguiente, si al ingresar un valor, encuentra que el nodo en el que debería ir está ocupado, entonces crea una lista enlazada, y en vez de guardar el archivo, guarda la lista enlazada que contendrá todos los archivos que tengan ese mismo nombre. De esta manera, al buscar, un nombre de archivo que tenga varias instancias, se encontrará una lista enlazada con todas las instancias con ese nombre.

4.2 Criterios de diseño de la estructura de datos

La estructura de datos fue diseñada de manera que podamos obtener buenos tiempos de ejecución, un árbol binario de búsqueda y balanceado nos permite tener complejidades sobre todas sus operaciones O(log n) por lo que al tener gran cantidad de datos no sería una tarea ardua hacer operaciones como búsqueda, inserción o borrado, otra pregunta que nos podríamos hacer es ¿Por qué simplemente no guardar las cadenas de caracteres? y esta tiene una respuesta, al guardar solo cadenas de caracteres podríamos solo saber si está en el directorio padre de todos los otros, mientras que si diseñábamos un objeto para carpetas y archivos podríamos conocer en todo momento la jerarquía sin sacrificar las complejidades, dentro de la estructura de datos también se hace uso de LinkedList ¿Por qué esto? Como se mencionó en el numeral anterior el árbol rojo negro no permite almacenar elementos iguales y ya que en java el TreeMap que es la implementación del árbol rojo negro necesita una llave y un valor, la llave es el nombre (cadena de caracteres del .txt) y el valor es el objeto es decir

carpeta o archivo ya que las dos heredan de AbstractClass se guarda un AbstractClass, entonces cuando sean iguales no se almacena lo anterior si no una LinkedList con todos los iguales y la llave sigue siendo la misma que sería el nombre, su complejidad aumenta un poco en una sola operación y esto es a la hora de imprimir un nodo especifico pero es para casos que ocurren poco, con lo anterior se obtiene eficiencia y sin perder en ningún momento información.

4.3 Análisis de la Complejidad

La complejidad de creación del árbol con los datos del archivo .txt se convierte en O(n log n), ya que el árbol inserta en tiempo (log n) pero al tener que insertar todos los elementos del .txt desde la primera vez su complejidad es O(n log n). Para buscar la complejidad es logarítmica porque el árbol esta ordenado. Listar todos los elementos es O(n). Obtener la ruta es O(n*m) donde n es el número de elementos en el caso de que haya una LinkeList en el nodo, m es el número de carpetas que contienen a el elemento para el cual se requiere la ruta. Imprimir el número de elementos iguales es O(m log n) primero busca si esta, y si esta y es una LinkedList m seria el tamaño de esta.

Método	Complejidad
Creación	O(n log n)
Búsqueda	O(log n)
Obtener ruta	O(n*m)
Listar todos los elementos	O(n)
Imprimir elementos repetidos	O(m log n)

Tabla 1: Tabla para reportar la complejidad

4.4 Tiempos de Ejecución

	juegos.txt	treeEtc.txt	ejemplito.txt
Creación	155.22ms	44.02ms	35.12ms
Búsqueda	0.0168ms	0.0051ms	0.0122ms
Obtener ruta	0.8ms	0.06ms	0.40ms
Listar todos los elementos	12582ms	483.81ms	0.86ms
Imprimir elementos repetidos	0.466ms	0.061ms	0.616ms

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

4.5 Memoria

	juegos.txt	treeEtc.txt	ejemplito.txt
Consumo de memoria	28,8MB	14,61MB	10,72MB

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

Como se puede observar en las tablas anteriores los tiempos son excelentes, siendo eficientes para cada uno de los casos y dependiendo de si existen o no elementos iguales dentro del .txt claro que esto no afecta mucho la complejidad y solo aumenta en una mínima parte el tiempo de ejecución, para cada una de las operaciones la eficiencia es muy buena, solo se puede observar un poco de demora al listar todos los elementos pero es algo que no se puede reducir más pues es O(n) porque tiene que imprimir cada uno de los elementos que tiene sin obviar ninguno ya que perderíamos información.

4.6 Análisis de los resultados

Estructura de datos para búsqueda en un directorio	Árbol rojo negro o TreeMap en java
Espacio en el Heap	20,78MB
Búsqueda de “content”	0.027ms
Búsqueda de “rules.txt”	0.016ms
Búsqueda de “title”	0.014ms
Búsqueda general	0.0168ms

Tabla 4: Tabla de valores durante la ejecución

5. CONCLUSIONES

Se pudo implementar una estructura de datos para lo que se solicitaba en el proyecto y pudimos manejar un problema que no habíamos contemplado en un principio, el de las colisiones que se podían generar en nuestra estructura. Además, el resultado final del proyecto ofrece algunas cosas extras que queríamos ponerle, hemos logrado un sistema eficiente que en su ejecución no contempla amplios tiempos y ofrece en caso de que fuera para un usuario un sistema muy completo el cual puede contemplar aún más mejoras en cuanto a funcionalidades, si comparamos algunos de los trabajos relacionados obtenemos algo al alcance y sólido.

5.1 Trabajos futuros

Una posible continuación para este proyecto seria implementar la búsqueda por tamaño, usuarios y por extensión de los archivos y mejorar la complejidad y gasto de memoria de muchas de las operaciones que tenemos en el código.

AGRADECIMIENTOS

Esta investigación fue soportada parcialmente por el programa de becas con aportes de empleados EAFIT.

Esta investigación fue soportada parcialmente por el programa de créditos condonables, Alianza MINTIC - MEN - ICETEX (ACCES)

Nosotros agradecemos por la ayuda con la metodología a Agustín Nieto, estudiante de EAFIT, por los comentarios que nos hizo para mejorar el código de la lectura de archivo y el manejo de colisiones.

REFERENCIAS

[1]NTFS, 2017. Consultado el 12 de agosto de 2017, Wikipedia, La enciclopedia libre. Recuperado de: https://es.wikipedia.org/wiki/NTFS.

[2]Árbol rojo-negro, 2017. Consultado el 12 de agosto de 2017, Wikipedia, La enciclopedia libre. Recuperado de: https://es.wikipedia.org/w/index.php?title=%C3%81rbol_rojo-negro&oldid=99055928.

[3]B-Árbol, Tutor de Estructuras de Datos Interactivo: sección: inserción en un b-árbol, Exposito Lopez Daniel, Abraham García Soto, Martin Gomez Antonio Jose, director proyecto: Joaquín Fernández Valdivia, Universidad de granada: consultado el 12 de agosto de 2017, recuperado de: http://decsai.ugr.es/~jfv/ed1/tedi/cdrom/docs/arb_B.htm.

[4]Árbol-B, 2017. Consultado el 12 de agosto de 2017, Wikipedia, La enciclopedia libre. Recuperado de: https://es.wikipedia.org/w/index.php?title=%C3%81rbol-B&oldid=100262033.

[5]Gurin, S. Árboles AVL. Consultado el 11 de agosto de 2017, recuperado: http://es.tldp.org/Tutoriales/doc-programacion-arboles-avl/avl-trees.pdf.

[6]Árbol biselado, 2017. Consultado el 11 de agosto de 2017, Wikipedia, La enciclopedia libre. Recuperado de: https://es.wikipedia.org/w/index.php?title=%C3%81rbol_biselado&oldid=98936910.

Adobe Illustrator, utilización el 10 de agosto de 2017. Descarga hecha de: http://www.adobe.com/Illustrator.

Grafica 1: Árbol rojo negro example.svg. (2016, 18 de septiembre). Wikipedia, La enciclopedia libre. Obtenido el: 10 de agosto de 2017 desde https://commons.wikimedia.org/w/index.php?title=File:Red-black_tree_example.svg&oldid=206955896.

Grafica 2 y 3: ejemplo de un árbol B y ejemplo de inserción en un árbol B, Tutor de Estructuras de Datos Interactivo: sección: inserción en un b-árbol, Exposito Lopez Daniel, Abraham García Soto, Martin Gomez Antonio Jose, director proyecto: Joaquín Fernández Valdivia, Universidad de granada: consultado el 12 de agosto de 2017, recuperado de: http://decsai.ugr.es/~jfv/ed1/tedi/cdrom/docs/arb_B.htm.

Grafica 4: rotación árbol AVL tree10.png, www.ibiblio.org, consultado el 11 de agosto de 2017, obtenido de: https://www.ibiblio.org/pub/linux/docs/LuCaS/Tutoriales/doc-programacion-arboles-avl/htmls/rotacion_simple.html.

Grafica 5: AVL-tree-wBalance K.svg. (2016, 1 de junio). Wikipedia, La enciclopedia libre. Obtenido 10 de agosto de 2017 Desde: https://commons.wikimedia.org/w/index.php?title=File:AVL-tree-wBalance_K.svg&oldid=197882389.

Grafica 6: BinaryTreeRotations.svg. (2016, 23 de noviembre). Wikipedia, La enciclopedia libre. Obtenido 10 de agosto de 2017 de: https://commons.wikimedia.org/w/index.php?title=File:BinaryTreeRotations.svg&oldid=218298521.