DATA STRUCTURE FOR EFFICIENT INDEXING OF FILES

Mateo Restrepo Sierra
Universidad Eafit
Colombia
mrestrepos@eafit.edu.co

Nicolás Restrepo López
Universidad Eafit
Colombia
nrestrepol@eafit.edu.co

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

ABSTRACT

Every day, technology changes and develops, so most companies are getting into it to offer a better and more comfortable experience to costumers. However, here lies a huge problem, which is that, most of the times, it is difficult to handle great amounts of information on the companies' databases easily. It becomes imperative to develop more efficient ways to get access to that information, so it is simple, intuitive, and requires fewer resources -in other words, it is cheaper-.

To establish an optimal solution, we must look for different types of data structures, so we can find one that fits our requirements, which are mainly based on searching specific elements within a great amount of data. That is when we realized that the implementation of a tree would be efficient, because complexities for a worst-case scenario is always O(Log n). However, as we will discuss later, it is more suitable to use a Self-Balanced Tree to avoid cases in which our search would become O(n). Finally, we decided to implement an AVL Tree, which might be the best Self-Balanced Tree when talking about searching, because that is our main concern. Insertion and deletion are not as important.

Keywords

Data Structure, Abstract Data Type (ADT), Binary Search Tree, Self-Balanced Tree, Complexity.

ACM CLASSIFICATION KEYWORDS

Data → Data Structures → Trees

Data → Files → Sorting/Searching

1. INTRODUCTION

People are using operating systems daily; some of them prefer Windows, some others prefer Mac OS, Android, or Linux, but most of them do not know about the *File System*, which plays an important role on the success of an operating system. The idea of a *File System* is to manage all the free space and allow to access saved files on a disk or a partition [1].

In this project, we are implementing and designing a Data Structure in order to search for files efficiently in a directory. We are also making comparisons between different Data Structures to determine which one is the best for the given task. We are not limiting ourselves on the implementation of only one single structure.

2. PROBLEM

Our main problem is the need of listing information contained on a directory, to search files and subdirectories efficiently. To do so, we need to implement an adequate Data Structure that would store that information, allowing us to look it up.

Searching can be made by name or the size of files. When looking for a directory we also must display all subdirectories stored inside of it. This very is important because it makes us know all the information contained and the way it is classified.

3. RELATED WORK

3.1 Array

An Array is a Data Structure determined by a collection of elements of the same type, and each one is store in a specific index. The main operations are insertion, traverse, deletion, search and update. All of them seem to be good when implementing, but their complexity is far from efficient. This happens because those complexities are the same for average and worst cases, which is O(n), where n represents the size of the array. If the array is big, it would take a great amount of time to complete different tasks. This structure is more likely to be used as a Hash Table, as we will see in the following index.

3.2 Hash Table

A Hash Table works like sets, in which elements cannot be repeated. Its creation is divided in two parts: an array and a mapping function. The hash function provides a way to assign the numbers to different data types, to store it in a specific index of the array. This structure has a good time of complexity in the average time, that is O(1), but in the worsts of the cases it behaves like a normal array, so it is O(n) [2]. Also, this data structure might cause problems derived from conflicts between two elements for which their hash function returns the same value. In this case, it is recommended to implement a multidimensional data structure, being a Hash Table made of LinkedLists, for example.

3.3 Red-Black Tree

A Red-Black Tree is an evolution of the Binary Search Tree that gives an extra special feature to each node, which is a color that could be red or black. This property makes possible for the tree to have a size equal or less than Log n, being n the number of nodes. Therefore, all operations in this Data Structure have the time of complexity of O(Log n), among

which we can highlight insertion, deletion and searching [3][4]. We call this kind of tree a Self-Balanced Tree, which are more effective than common Binary Search Trees in the case of inserting elements in order, that in the second structure will result in the creation of a *pseudo* LinkedList.

3.4 B-Tree

A B-Tree is a Self-Balancing Tree that, differently from other trees of its kind, does not store information in the main memory, but in the disk. B-Trees were designed to handle great amounts of data, that cannot be stored in the main memory. In this tree, data is read like blocks, which takes a lot of time that main memories would not support. The idea is to access the disk as least as possible [5]. Therefore, the main operations of B-Trees have all complexities of O(n), being n the number of nodes.

4. AVL TREE

At first, we need to know that an AVL Tree is a data structure, more specifically a self-balanced Binary Search Tree (BST). Being self-balanced means that, for all nodes, the difference between the height of the left and right subtrees is at least one.

Every node in the tree has a number assigned to it, which represents its balanced factor. These are calculated with the formula *Bf = Hl - Hr*, where *Bf* is the Balanced Factor, *Hl* is the height of the left subtree and *Hr* is the height of the right subtree.

The next diagram in Figure 1 shows a simple example of an AVL Tree with balanced factors shown in green above the nodes.

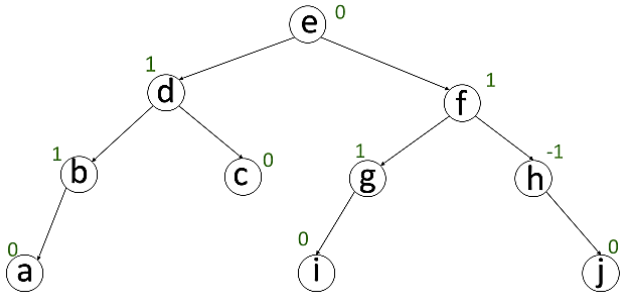

Figure 1: This is a simple example of an AVL Tree.

4.1 Operations of the data structure

The operations of the AVL are the basic ones: Access, Search, Insertion and Deletion. As we know, these are the main operations for any data structure. It is important to be careful with the way these operations could affect the efficiency of the structure. That is because, when inserting and deleting, it is necessary to check whether the tree becomes unbalanced or not. If so, the nodes must rotate to reorganize the structure in a balanced form [3]. This provides a solution for problems in Binary Search Trees mentioned in index 3.3.

The diagram in Figure 2 shows the traverse in the searching operation for AVL Trees, represented in orange and discontinued arrows.

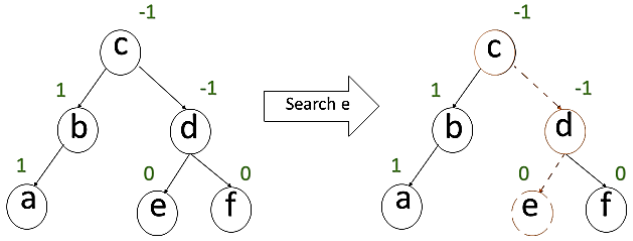

Figure 2: Searching operation. As you can see, it is not necessary to traverse the entire tree.

4.2 Design criteria of the Data Structure

For our project, searching is the most important operation, so it needs to be the most efficient of them all. When looking at the complexity tables for different data structures, the best ones for the worst cases are trees. There are many kinds of trees, but we chose two specific ones: Red Black Tree and AVL Tree. When comparing them, we see that both have the same complexity for searching, which is O(Log n), and that is very efficient. However, which one is the best? For insertion, Red Black Tree can be a better than the AVL, but as we said earlier, for this project searching is the most important item. The AVL Tree is better in searching than the Red Black Tree because the height of its subtrees is shorter, and this provides faster look-ups for the information, by the cost of insertion and delete [6].

4.3 Complexity Analysis

The following table in Figure 3 shows the complexity analysis for every standard method in AVL Trees.

Method	Complexity
Access Height of the Node*	O(1)
Rotate Node to the Left*	O(1)
Rotate Node to the Right*	O(1)
Find the Balance Factor*	O(1)
Insert a Node	O(Log n)
Delete a Node	O(Log n)
Find the Leftmost Node*	O(Log n)
Search an Element	O(Log n)
Print *Pre-Order*	O(n)
Print *In Order*	O(n)
Print *Post Order*	O(n)

Figure 3: Complexity table for AVL Tree standard methods. Those signed with a (*) are auxiliary for main methods to work.

4.4 Execution Time

Table in Figure 4 shows running times in milliseconds for five different collections of randomly created non-negative integer. Each collection contains 30 nodes.

Running Times (Milliseconds)					
	Tree 1	**Tree 2**	**Tree 3**	**Tree 4**	**Tree 5**
Insert	5030	5010	5010	4010	5010
Delete	5030	6020	6020	6020	6020
Search	16030	16040	16140	32120	16050
Pre-Order	61600	61800	59590	59700	59310
In Order	61300	57290	61430	61370	61400

Figure 4: Execution time for public standard methods in AVL Trees. Post Order method was not considered, as well as private methods labeled as auxiliary in Figure 3.

4.5 Memory Used

Table in Figure 5 shows memory used in megabytes for five different collections of randomly created non-negative integer. Each collection contains 30 nodes.

Memory Used (Megabytes)					
	Tree 1	**Tree 2**	**Tree 3**	**Tree 4**	**Tree 5**
Insert	1,1	8,4	5,9	2,5	1,3
Delete	5,4	1,7	1,6	1,3	7,1
Search	1,7	1,3	2,7	1,7	1,2
Pre-Order	4,9	1,8	1,4	3,5	6,6
In Order	4,4	2,7	1,7	8,1	1,6

Figure 5: Memory used by public standard methods in AVL Trees. Post Order method was not considered, as well as private methods labeled as auxiliary in Figure 3.

4.6 Result Analysis

Table in Figure 5 shows minimum, average and maximum running times in milliseconds for previously used collections, as well as memory used in Megabytes.

	Running Time (Milliseconds)			Memory Used (Megabytes)		
	Worst	**Average**	**Best**	**Worst**	**Average**	**Best**
Insert	5030	4814	4010	8,4	3,84	1,1
Delete	6020	5822	5030	7,1	3,42	1,3
Search	32120	19276	16030	2,7	1,72	1,2
Pre-Order	61800	60398	59310	6,6	3,64	1,4
In Order	61430	60558	57290	1,6	3,7	8,1

Figure 6: Table showing minimum, average and maximum running times in milliseconds for every operation, as well as memory used in Megabytes.

As we can see, printing trees and printing sorted trees are the most complex operations, with an estimated complexity of O(n), being n the number of nodes in the tree. That is because, for printing it is necessary to traverse the whole tree. On the other side, insertion, deletion and searching have less complexity, estimated on O(Log n). In these cases, the algorithm splits on halves, representing the two children —left and right— of every node. Here, deletion is not relevant for our purposes, so we focus mainly on searching, which has an acceptable average running time of 19276 milliseconds for a 30-node tree when establishing the one second stopover.

5. AVL TREE USING HASH FUNCTIONS AND STACKS

After analyzing the advantages of the implementation of an AVL Tree to solve our problem, we decided to use it [7], but with slightly differences so it would fit our requirements. We created a new class called *Thingy*, which would represent an object that could be either a directory or a file. This *Thingy* class has seven fields, consisting on one integer value, four Strings, one boolean and one LinkedList. These are, in order: data, name, size, user, directory, isFolder and subdirectories. Respectively, they represent an integer value to be inserted in the tree, the name of the object, its size, its user or domain, its path, if it is a directory or not and a list of all its direct subdirectories. This will all be explained in index 5.2.

5.1 Operations of the Data Structure

As our data structure is an AVL Tree, it has the main functions insert, delete and search. We would not worry that much about deletion because it is not relevant for our implementation. We would focus on the other two.

5.1.1 Insertion

This is something that will only be done once when reading the .txt file in which the elements to be inserted are to be received. To make things simple, we create two AVL Trees: one that stores the names of the objects and the other one stores their sizes. We first create the trees, and insertion would proceed as following.

Every line of the .txt file must be read, and while that happens, the name, user and size are taken from it. With those values, a new *Thingy* object is created. Depending on the indentation of the current line and its following, the object would be added to auxiliary Stacks that store the sub directories of the current directory in which we are located. These Stacks would continue storing newly created *Thingys* until we exit the current directory. That is when all its content is to be inserted in the tree by name, using a Hash function to convert the String to an integer value that would be designated as the *data* field of the *Thingy*. This is the function:

$$\sum_{i=0}^{n-1} a_i * 31^{n-1-i}$$

Here, n represents the length of the name and a_i the ASCI value of the character in the i position in the name. This function allows for negative values to be created.

When emptying the Stack, the names of those inserted *Thingys* are to be added to the LinkedList *subdirectories* field of the directory that we just exited.

But we cannot forget about the *sizes* tree. Objects are inserted in it right after the insertion is done in the *names* tree. We must create a new *Thingy* value that gets the same parameters as the previous object that was inserted in *names*. Then, we check the letter in the String that represents the size. Depending on that letter (K, M or G), we multiply the decimal part of the String to have integer values in Bytes unit as standard. If the String has no letter, it is assumed that the size is in Bytes. That resulting integer value is assigned to the *data* file, and then new object is inserted in *sizes*.

The insertion process of a single element has a logarithmical complexity, following an order in which higher or equal numbers are inserted on the right and lower are inserted on the left. This process also follows the balancing property of AVL Tree as we can see on Figure 8.

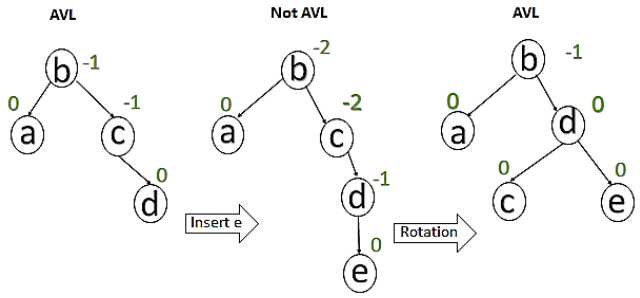

Figure 8: Insertion process on AVL Trees.

5.1.2 Searching

Our implementation has three different types of searching. The first one checks if the object does exist in the tree, and prints its *directory* field, that means, its path. If the object does not exist, the user receives a warning. The second searching method is for subdirectories. This method checks if the object does exist in the tree, and prints all elements contained in the LinkedList *subdirectories* field. The user receives a warning if the object does not exist or if it is a file instead of a directory. The last case is searching by sizes, for which we would use the *sizes* tree.

Searching by sizes is also divided into three cases. The user can search by range, which prints all objects whose sizes are in that inclusive range, *low* which means that objects with an equal or smaller size are printed and high which prints all objects with an equal or bigger size.

The diagram on Figure 9 shows how searching operation works. Each letter represents an object inserted in the tree, with all its fields already defined. The traversal done for searching the element is shown in orange and discontinued arrows.

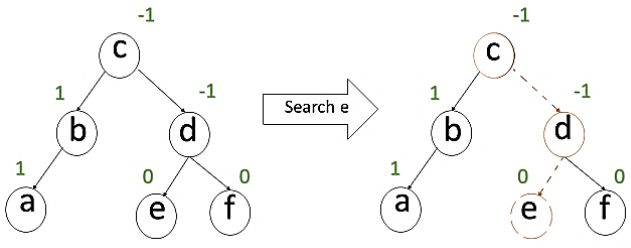

Figure 9: Searching operation for object e.

5.2 Design Criteria of the Data Structure

The constructor method would get as parameters the name, size and user. This method also creates the LinkedList *subdirectories*, that would remain empty if the object in question is a file. If it is a directory, this list would contain its direct subdirectories, meaning that files contained in a folder inside this directory are not considered, for the sake of efficiency. We determine if an object is a file or a directory by simply analyzing the indentation on the next line. If it is higher, it means that the current object is a directory. If it is equal or less, it is a file. This information would be stored in the *Thingy* class as *isFolder*. Notice that these criteria give the same treatment for files and empty directories. Finally, the *directory* field stores the path of the object.

The diagram on Figure 7 shows an early implementation of the AVL Tree meeting our criteria.

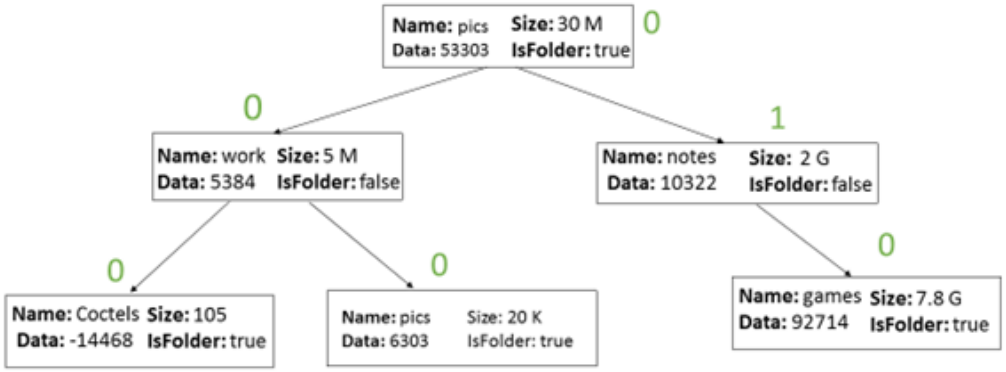

Figure 7: AVL Tree containing *Thingy* objects organized by sizes. See that directory, subdirectories and user fields are missing. Also, *work* and *pics* objects are in the wrong position.

An important change that was made to the original AVL Tree structure was to modify the fields of a node, that would not receive an integer value as parameter, but a *Thingy* object so it would be stored in the tree. This means that the insertion method in the tree was also modified to add the data of the *Thingy* field instead of the original *data* field.

5.3 Complexity Analysis

Our implementation has two basic operations: insertion and searching. The way objects are inserted in the tree determines its complexity, because even if it is an AVL Tree, the modifications done do not assure that the operation complexity is logarithmical. The same goes to searching operation. The table contained in Figure 10 shows the complexity analysis for our insertion and searching methods.

Method	Complexity
Insertion	$O(m * n * t * s)$
Search by Name	$O(\log m)$
Search Subdirectories	$O(s + \log m)$
Search Lower Sizes	$O(m)$
Search Higher Sizes	$O(m)$
Search Range of Sizes	$O(m)$

Figure 10: Complexity Analysis for insertion and searching operations in our implementation, with *m* being the number of lines of the .txt file -number of objects to be inserted in each tree-, n being the length of the name of the object, t being the number of directories in the tree and s the number of files in the tree.

5.4 Execution Time

The following table in Figure 11 shows the execution times in milliseconds by our implementation using three different .txt files.

Running Times (Milliseconds)			
	ejemplito.txt	**treeEtc.txt**	**juegos.txt**
Insert	15	184	3236
Search Directory	~0	~0	~0
Search Sub directories	~0	~0	335,7
Search Low	1215	4343	15440
Search High	~0	~0	~0
Search Range	~0	~0	1117

Figure 11: Execution time in milliseconds by three different .txt files -ejemplito.txt has 4 directories and 17 files, treeEtc.txt has 425 directories and 3225 files, juegos.txt has 2609 directories and 62901 files. Searching directories were made with the last element in the .txt file, searching under 1 Megabyte, searching over 1 Gigabyte and searching between 1 Megabyte and 1 Gigabyte. Notice that some execution times were very close to zero.

5.5 Memory Used

The following table in Figure 12 shows the memory used in Megabytes by our implantation using three different .txt files in three tries.

Memory Used (Megabytes)			
	ejemplito.txt	**treeEtc.txt**	**juegos.txt**
First Try	9,566	13,750	48,253
Second Try	9,565	13,749	47,595
Third Try	9,565	13,750	52,018

Figure 12: Memory used in Megabytes by three different .txt files -ejemplito.txt has 4 directories and 17 files, treeEtc.txt has 425 directories and 3225 files, juegos.txt has 2609 directories and 62901 files.

6. CONCLUSIONS

We looked for different kinds of data structures, which are described in index 3. This helps determine which structure is more suitable to solve different problems.

The development of this project made us realize of some daily uses of data structures, and how the use of them can determine the complexity of an algorithm. In addition, data structures can also be used in combination, such as we did with an AVL Tree using Hash Functions for insertion, which was helped by Stacks. This mixture can provide organized and stronger results.

Something that we realize is that, when making our first implementation, we chose the AVL Tree simply because it was said that it provides a good complexity and execution time for searching, but this turned out to be just a myth. An AVL Tree is not efficient by itself: if insertion is made correctly, searching will be better.

6.1 Future Work

This was something we considered at first, but because of time issues, it was not possible to implement. We plan to modify our implementation, so it would be possible to execute the file or directory that is being searched. This would be very practical for the user, instead of only having the path of the file or directory.

We also think on implementing something similar in different data structures to compare and find out which one gets better results, and possible failures we might be committing and not realizing.

ACKNOLEDGEMENTS

We thank for the help provided by Juan Sebastián Cárdenas Rodriguez, who helped us in the implementation of Stacks for insertion.

REFERENCES

[1] Henson, V. A Brief History of UNIX File Systems, IBM Inc, retrieved August 13, 2017. https://www.lugod.org/presentations/filesystems.pdf

[2] Flatt, M. (Unknown). The Racket Reference, retrieved October 25, 2017. https://docs.racket-lang.org/reference/hashtables.html

[3] Morris, J. (1998). Red-Black Trees, retrieved October 25, 2017. https://www.cs.auckland.ac.nz/software/AlgAnim/red_black.html

[4] An Introduction to Binary Search and Red-Black Trees, 2016. Top Coder, retrieved October 25, 2017. https://www.topcoder.com/community/data-science/data-science-tutorials/an-introduction-to-binary-search-and-red-black-trees/

[5] B-Tree Set 1 (Introduction), 2011. Geeks for Geeks, retrieved October 25, 2017. http://www.geeksforgeeks.org/b-tree-set-1-introduction-2/

[6] Red Black Tree Over AVL Tree, 2012. Stack Overflow, retrieved September 29, 2017. https://stackoverflow.com/questions/13852870/red-black-tree-over-avl-tree

[7] Bhojasia, M. Java Program to Implement AVL, Sanfoundry, retrieved September 30, 2017. http://www.sanfoundry.com/java-program-implement-avl-tree/