DATA STRUCTURE IMPLEMENTATION 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-.

Keywords

Data structure, TAD, Binary Search Tree, self-balanced, complexity.

ACM CLASSIFICATION KEYWORDS

Hardware → Performance and Reliability → Performance Analysis and Design Aids

Software → Programming Languages → Processors → Run-time Environments

Software → Operating Systems → File System Management → Access Methods

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.

When they were created, FS supplied the necessities, but they were also disk layouts and had very little memory (512-1024 bytes). Then it came the FFS, in which dates are organized in cylinder groups. This makes some improvements, increases the performance, and files can now be named, renamed or locked. Those were two examples, but there are constantly new FS being created or improved in many ways[1]. And as they become more complex, searching them is also becoming more difficult. Nowadays, files are bigger and can be sorted by lots of conditions, so they must be organized in such a structure that optimizes basic operations as much as possible, using little memory and also being easy for the user.

2. PROBLEM

The problem is that, just like technology develops, costumers and people are becoming more demanding. They need faster Operation Systems that can supply all their necessities. So, it is necessary to have a nice File System that can handle lots of information on databases in just few seconds. For example, if a person needs to search some files by name or weight on his laptop, the idea is that this action could be done correctly in little time.

3. RELATED WORK

3.1 Complexity and Time:

As we all know, every Data Structure has the same basic operations: access, search, insert and delete. The are many kinds of Data Structures, such as arrays, stacks, queues, lists, trees and Hash tables, among others. However, which one is the best? That is a good question, but it cannot be completely answered. Depending on the Data Structure that is being implemented, the operations would have different complexities, so some structures would be efficient in a specific operation while other would not. When we talk about efficiency we are talking about the time taken by an algorithm to run, and that is something that must be considered when choosing a Data Structure to work with.

3.2 Searching:

As we will be using a Data Structure specifically to examine directories in a computer, the most important operation would be searching. Being said that, we are in need of finding and efficient structure when implementing that. There are lots of options. For example, lists, stacks and queues all have a complexity of O(n) for searching, while trees have $O(\text{Log } n)^2$. For a start, trees might be our best option, but we would also have to examine the exact conditions of every single type of tree.

3.3 Stack Overflow:

The problem here is very simple. Every computer has a maximum stack size. When it is full, the running process crashes, or at least it becomes slower. So, the use of memory is also something to take into account. Fortunately, most Data Structures have a space complexity of O(n); not the best but it does work[2]. However, when implementing the structure, we must attempt not to modify it in a way that could affect the memory it would normally need to work.

3.4 Sorting

One of the objectives of this project is to be able to sort different types of files according to several conditions: size,

name, date of creation and many others. There is a great amount of ways in which this could be done, but, such as the selection of a data structure, efficiency is also relevant in here. With most of the sorting methods available, their complexities are not good, being $O(n^2)$ or O(n Log n). However, methods such as Radix Sort and Counting Sort offer an interesting linear complexity of O(nw) and O(n+k), respectively[2], being w the size of words or the number of digits in Radix and k the maximum key on the collection in Counting[3].

4. AVL TREE

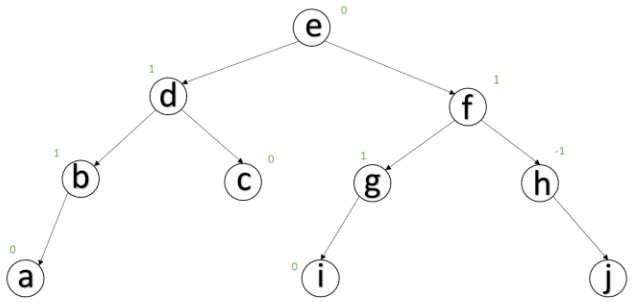

Figure 1: This is a simple example of an AVL Tree.

Note: The numbers above the tree are the balanced factor and they 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.

4.1 Operations of the data structure:

The operations that the data structure will have 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 unbalance the structure. That is why, 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[4].

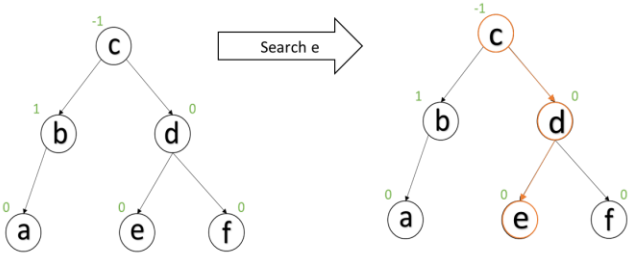

Figure 2: Searching operation. As you can see, it is not necessary to traverse the entire tree.

4.2 Design criteria of the data structure:

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. For our project, searching is the most important operation, as we won't be inserting or deleting data. This is why it needs to be efficient. When looking at 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[5].

4.3 Complexity Analysis:

Tabla de complejidades

Mètodo	Complejidad
Bùsqueda Fonètica	O(1)
Imprimir bùsqueda fonètica	O(m)
Insertar palabra busqueda fonètica	O(1)
Búsqueda autocompletado	O(s + t)
Insertar palabra en TrieHash	O(s)
Añadir bùsqueda	O(s)

Figure 3: Complexity table for standard Data Structures operations.

4.4 Execution Time:

Running Times (Milliseconds)					
	Collection 1	Collection 2	Collection 3	Collection 4	Collection 5
Insertion	50300	50100	50100	40100	50100
Deletion	50300	60200	60200	60200	60200
Searching	160300	160400	161400	321200	160500
Printing Tree	616000	618000	595900	597000	593100
Printing Sorted Tree	613000	572900	614300	613700	614000

Figure 4: Table showing running times in milliseconds for five different collections of randomly created non-negative

integers with five different operations. Each collection contains 30 nodes.

4.5 Memory Used:

Memory Used (Megabytes)					
	Collection 1	Collection 2	Collection 3	Collection 4	Collection 5
Insertion	1,1	8,4	5,9	2,5	1,3
Deletion	5,4	1,7	1,6	1,3	7,1
Searching	1,7	1,3	2,7	1,7	1,2
Printing Tree	4,9	1,8	1,4	3,5	6,6
Printing Sorted Tree	4,4	2,7	1,7	8,1	1,6

Figure 5: Table showing memory used in Megabytes for five different collections of randomly created non-negative integers with five different operations. Each collection contains 30 nodes.

4.6 Result Analysis:

	Running Time (Milliseconds)			Memory Used (Megabytes)		
	Worst	Average	Best	Worst	Average	Best
Insertion	50300	48140	40100	8,4	3,84	1,1
Deletion	60200	58220	50300	7,1	3,42	1,3
Searching	321200	192760	160300	2,7	1,72	1,2
Printing Tree	618000	603980	593100	6,6	3,64	1,4
Printing Sorted Tree	614300	605580	572900	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, insertion and deletion are not relevant for our purposes, so we focus on searching, which has an acceptable average running time of _____ for a 30-node tree.

Note 2: The algorithm for AVL Trees was taken from three different websites, appearing on the references[6 7 8].

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. Big O Cheat Sheet, retrieved September 28, 2017: http://bigocheatsheet.com/

3. Radix Sort, 2017. Retrieved October 1, 2017. https://www.hackerearth.com/es/practice/algorithms/sorting/radix-sort/tutorial/

4. Data Structures. BTech Smart Class, retrieved September 30, 2017.

5. Red Black Tree Over AVL Tree, 2012. StackOverflow, retrieved September 29, 2017. https://stackoverflow.com/questions/13852870/red-black-tree-over-avl-tree

6. AVL Tree Set 1 (Insertion), 2011. Retrieved October 1, 2017, from Geeks for Geeks: http://www.geeksforgeeks.org/avl-tree-set-1-insertion/

7. AVL Tree Set 2 (Deletion), 2011. Retrieved October 1, 2017, form Geeks for Geeks: http://www.geeksforgeeks.org/avl-tree-set-2-deletion/

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