Data Structure for efficient indexing of files and directories

Juan S. Cardenas Rodriguez
Universidad EAFIT
Medellin, Colombia
jscardenar@eafit.edu.co

David Plazas Escudero
Universidad EAFIT
Medellin, Colombia
dplazas@eafit.edu.co

Mauricio Toro Bermudez
Universidad EAFIT
Medellin, Colombia
mtorobe@eafit.edu.co

ABSTRACT

In the following article, a data structure to represent files and directories of a computer was developed, taking into account operations such as efficient search and information retrieval. Representing and indexing files efficiently is key whenever users require quick access to information within their own computers. A data structure was created based on the idea of Tries, using the characters of the name of a file/directory and indexing them on hash-based structures; a successful implementation was achieved using Python with results better than expected.

CCS CONCEPTS

• **Theory of computation** → *Data structures design and analysis*;

KEYWORDS

Data structures, efficiency, indexing, searching files, complexity, memory.

1 INTRODUCTION

Data structures are transversal to all studies related to computers; at any time, data analysis is required to predict or develop new things. In this way, understanding which cases require the implementation of an specific data structure or is fundamental, or, at least, know how they work so that an efficient one we can build our own in favor of solving our problem more efficiently.

In today's world, people want everything fast; times have changed and if the user orders the computer to do something for them, they want an immediate answer. Especially when finding a file in their computer. In the hypothetical case where the user forgot where his Power Point presentation was and he has 5 minutes before presenting it; it would be frustrating that he had to wait a considerable amount of time for the computer to retrieve the file's location. In this sense, if we make a slow algorithm for searching files we would bring an unsatisfying user experience.

Consequently, although this problem is an specific one, it may apply to many other areas of knowledge; for example, in gaming, if the user wants to access his save file or check collision between objects, it would be a disappointing experience that this processes would take a lot of time. So, finally, we see that the real world often pushes developers to find faster ways to perform one task, and learning this skill from early on is almost essential to be a better developer.

2 PROBLEM

Design and develop a data structure to efficiently represent files and directories of a computer and search/retrieve information from it.

3 RELATED WORK

3.1 Red-black trees

Before we get into red-black trees, we will talk about binary search tree (BST). A BST is a tree on which nodes satisfy:

- The left sub-tree of a node has a key less or equal to its parents node's key.
- The right sub-tree of a node has a key greater or equal to its parents node's key [4].

A red-black tree is a BST with one extra bit of storage per node: its color, which can be either black or red [2]. No leaf is more than twice as far from the root as any other [1]. A BST is a red-black tree if it satisfies the following properties:

- Every node is either red or black.
- The root is black.
- Every leaf is black.
- If a node is red, then both of its children are black.
- For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Insertion of a node into an n-node red-black tree can be accomplished in $O(log(n))$ time. We use a slightly modified version of the tree-insert procedure; and we use recoloring and rotation to re-balance the tree [2].

Figure 1 shows an example of a red-black tree.

3.2 Hash tables

A hash table is a generalization of the simpler notion of an ordinary array [2], and each data value has its own index value. It is a data structure in which insertion and search operations are very fast [4]. Whenever an element is to be inserted, we compute the hash code of the key passed and locate the index using that hash code as an index in the array. Each data value is converted into an index of the array, using a hash function. This is how searching is carried out on Hash tables. The hash functions has to do an uniform distribution of keys: each key is equally likely to hash to any of the slots of the array, independently of where any other key has hashed to [2].

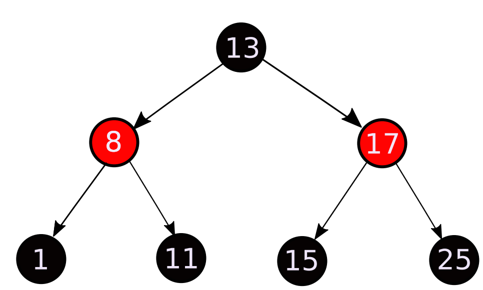

Figure 1: Example of red-black tree with integers

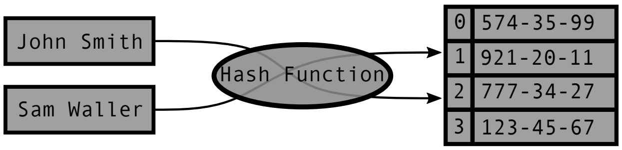

Figure 2: Example of Hash table with string keys.

Figure 2 shows an example of a Hash table.

3.3 B-Trees

The B-tree aims to solve the problem of given a large collection of objects, each having a key and an value, design a disk-based index structure which efficiently supports query and update. It is defined as a tree structure which satisfies the following properties:

- A node x has a value $x.num$ as the number of objects stored in x; they are stored in increased order.
- Every leaf node has the same depth.
- An index node x stores $x.num + 1$ child pointers.
- Every node except the root node has to be at least half full.
- If the root node is an index node, it must have at least two children.

Figure 3 shows an example of a B-Tree. The algorithm makes sure that root node is not currently full and, if it isn't, it inserts it in that node in a sorted way. If the root node is full, the algorithm will split it into two nodes and, the previous will pass to a higher level and, repeat the process until the node is available.

3.4 Skip list

Let S_0, S_1... S_r be a collection of sets that satisfies that for the set $S_r = 0$ and for each $i < j$, $0 \leq i \leq r$ and $0 \leq j \leq r$ then S_j is a subset of S_i. The index of each set is said to represent the level of each element in the set. Then, a skip list is the linked lists L_i each of them containing the respective S_i; in every linked list we attach the special keys negative and positive infinity at the start and end of the list respectively. Every list will have:

- Horizontal pointers: pointers that connect items in the same list.
- Descent pointers: is the pointer that every key k in L_i will have directed to the key k in L_{i-1}.

Figure 4 shows an example of a skip list. To insert an element to a skip list we first begin by marking the skip list (to know how to mark it check the chapter on the bibliography) with respect to the x to be inserted. It pushes the marked boxes to a stack when they are marked in the procedure and then pop marked boxes from the

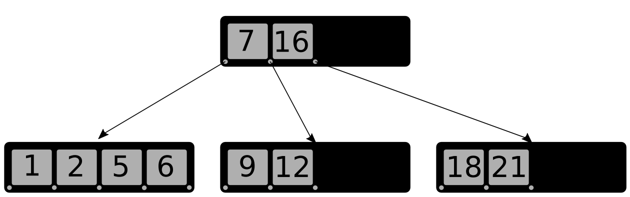

Figure 3: Example of B-Tree with integers

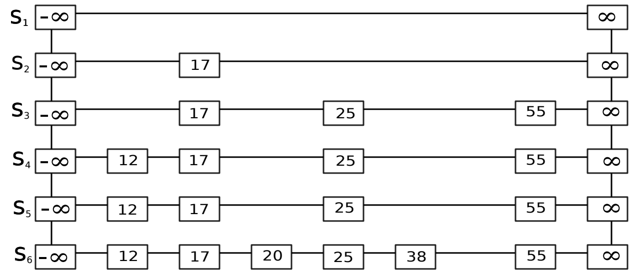

Figure 4: Example of a SkipList

stack as needed and then insert x next to each of the popped boxes. The idea of adding the negative and positive infinity is just to put some fixed values of the least and greatest value. So, in this way, this concept could apply in other areas as addresses in sorted way and so fourth [3].

4 NASH TABLE

The first approach proposed is NashTables. This is a data structure based on hash tables, double linked lists and Tries. It takes each letter of each file and it indexes it accordingly; this allows us to build a structure where the user can find files just by typing the first letters of the name of the file. Figure 5 shows the structure of a NashTable with some examples.

4.1 Operations of the data structure

4.1.1 Insertion. Figure 6 shows the insertion of a file named "queries.vbs". The program will first enter the "q" entry in the hash table because files are indexed based on the characters their name, from there it will enter the branch; since the depth of the NashTable is two, it will enter once again on the nash attribute of the branch, which is as well another NashTable; again, it will enter in the "u" entry in the hash; since the program has reached the maximum depth, it will now enter the dictionary of information of the file into the table attribute, which is a double linked-list.

4.1.2 Deletion. Figure 7 shows an example when the program is asked to remove the file named "a.docx". The proram will follow the same process as it were searching for a file. Whenever it is found, it is just deleted from the doubly linked list.

4.1.3 Search. Figure 8 shows an example of search for "a.jar". In first place, the program will search whether there is or not a key associated with the first letter in our hash table; if it does not exist, the program will know that the file is not in our data structure. Otherwise, we call recursively the Nash Table at that key, now using the second letter of the string. Lastly, if we reach the last letter of the string passed we proceed to return all the files that are in the table of the branch associated with that last key and, the files of the further Nash Tables after that key.

4.2 Complexity of operations

Table 1 shows the complexity of the operations of the data structure designed. Here, m is the number of lines in the file being read (usually an output of `tree` command in Unix), p is the maximum number of characters in said file, n is the length of the name of the

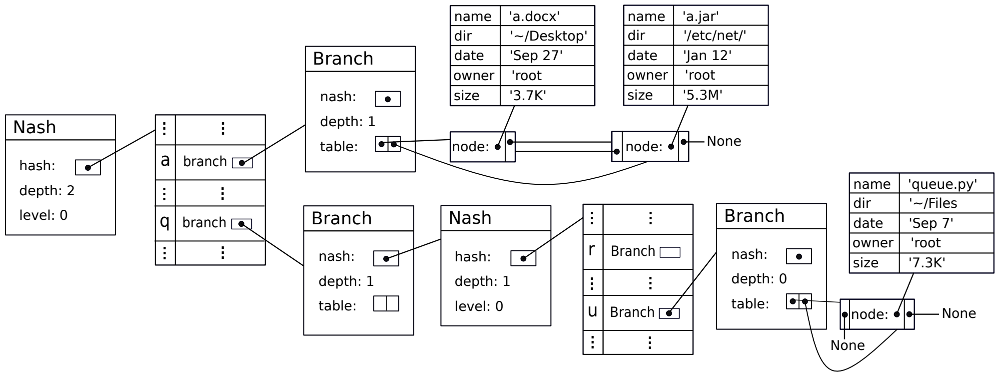

Figure 5: Example of NashTable

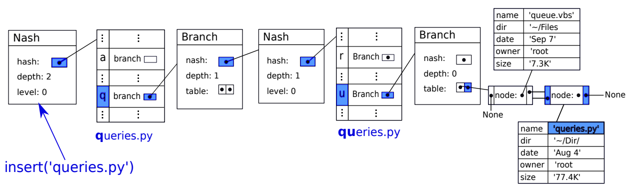

Figure 6: Insertion for NashTables.

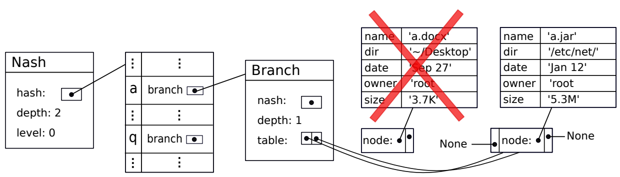

Figure 7: Deletion for NashTables.

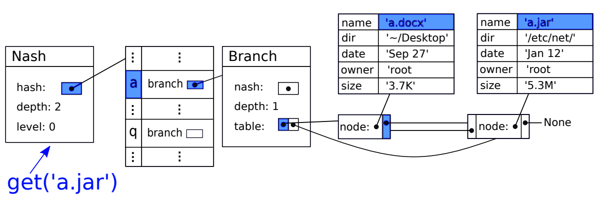

Figure 8: Search for NashTables.

Table 1: Time complexity for each operation

Operation	Complexity
Creation	$O(mp)$
Search	$O(n+k)$
Insert	$O(n)$
Remove	$O(n)$

file/directory and k is the number of files/directories that start with the same characters.

4.3 Design criteria of the data structure

The NashTable is based on hash tables and double linked lists; searching files is the priority. Each NashTable has a hash table, its hash function is associated with the nth letter of the name of the file, and n depends on the depth of the structure; the hash table has a reference to a Branch with has another NashTable and so fourth. Although this first approach to solve the problem is not optimized for low memory consumption, it has been developed for optimize searching with different options. Hash tables were used since they are able to search for files in constant time, and double linked lists because insertion in the last position is achieved in constant time and also because they don't have a fixed size (unlike arrays), allowing the NashTable to add objects depending on the files indexed.

4.4 Time and memory consumption

Table 2 shows the execution time for each operation of NashTables; and Table 3 shows memory consumption for each operation as well. The DataSet 1 has 425 folders and 3225 files, the DataSet 2 has 1937 folders and 23934 files, and DataSet3 has 56549 folders and 470070 files. For example, search has an average time of 0.024ms, which is really fast. Although the complexity of the operations seems slightly inefficient, when the algorithms are applied, different results are obtained; this is due to the fact that the worst-case scenario is highly unlikely, starting with the fact that not all the files indexed in a

Table 2: Execution time for each operation

Operation	Average Time		
	DataSet 1	DataSet 2	DataSet 3
Create	213.355ms	5453.520ms	29177.054ms
Insert	0.623ms	0.046ms	0.050ms
Search	0.026ms	0.022ms	0.023ms
Remove	0.018ms	0.020ms	0.020ms

Table 3: Memory consumption.

Memory Consumption	DataSet1	DataSet2	DataSet3
Memory	96.72MB	475.136MB	1638.4MB

computer are called the same. It is worth highlighting that, even though the size of the data set is not the same, the execution time for each operation, different from the creation of the structures itself, does not change much; therefore, the execution time for said operations does not depend on the size of the data, which is a huge advantage. All operations - insertion, search, remove - were executed fast enough; the creation of the data structure is slightly slow.

5 CONCLUSIONS

Based on the presented results, a successful implementation of the data structure was achieved. For example, files can be searched in less than one second in data sets that have more that 400k files, due to the fact that searching is independent from the amount of files indexed. It is clear that the data structure proposed is not efficient in terms of memory. This can be done for further implementations. Additionally, some useful functionalities like searching by size, owner or date created can be added for further work as well; making a better guided user interface (GUI) is another objective to achieve, even though it is not completely necessary.

6 ACKNOWLEDGEMENTS

This research was partially supported by GitHub, JetBrains' Pycharm and Pyhton Software Foundation.

REFERENCES

[1] Paul E. Black. 2015. Dictionary of algorithms and data structures: red-black tree. (2015). Retrieved August 7, 2017 from https://xlinux.nist.gov/dads/HTML/redblack.html.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. *Introduction to algorithms.* The MIT Press, Cambrigde, Massachusetts.

[3] Dinesh P Mehta and Sartaj Sahni. 2004. *Handbook of data structures and applications.* CRC Press.

[4] TutorialsPoint. 2016. Data structures and algorithms. (2016). Retrieved August 7, 2017 from http://www.tutorialspoint.com/data_structures_algorithms/data_structures_algorithms_tutorial.pdf.