Data structure for efficient indexing of files and directories

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

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

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

ABSTRACT

In this article we will develop a data structure to represent files and directories of a computer, taking into account operations such as efficient search and information retrieval. Representing and indexing files efficiently is key whenever users need quick access to information within their own computers.

CCS Concepts

•**Information systems** → Record storage systems;

Keywords

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

1. INTRODUCTION

Data structures are transversal to all studies related to computers; at each moment, we need to handle data and manipulate it so we can use them to do predictions or develop things with it. In this manner, is fundamental that we fully understand what specific cases requiere the implementation of an specific data structure or, at least know how they work so we can build our own in favour of solving our problem more efficently.

First of all, we live in a world where everyone needs everything quick; times have changed and if the user orders the computer to do something for them, they want a quick answer. Especially when finding an archive in their computer. Imagine that the user forgot where his Powe r Point presentation was and he had 5 minutes before presenting it; it would be infuriating 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 archives we would bring an unsatisfying experience to the user.

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 an awful 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.

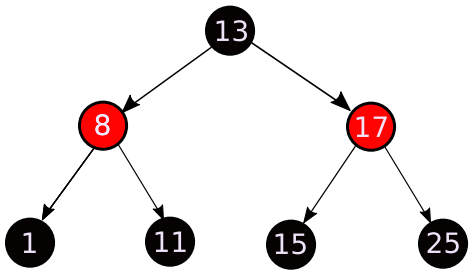

Figure 1: Example of red-black tree with integers

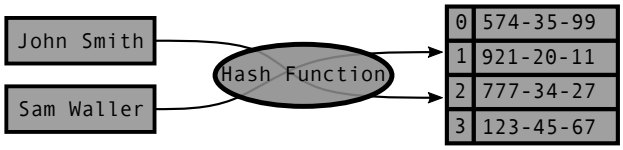

Figure 2: Example of Hash table with string keys.

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 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 efficently 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 currenty 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 avaible.

3.4 Skip list

Let S_0, S_1... S_r be a collection of sets that satifies 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

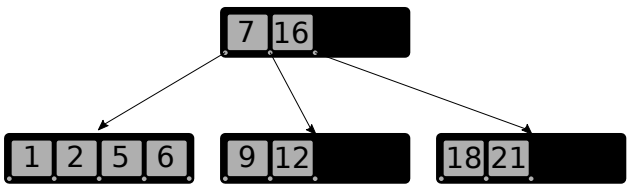

Figure 3: Example of B-Tree with integers

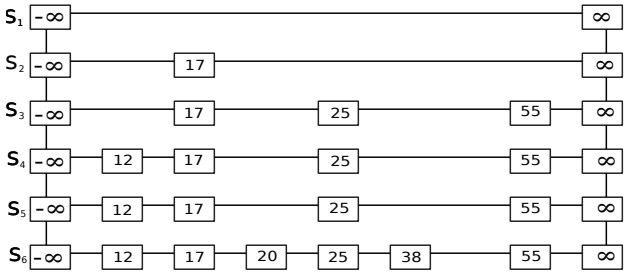

Figure 4: Example of a SkipList

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 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 adresses in sorted way and so fourth [3].

4. NASH TABLE

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".

4.1.2 Deletion

Figure 7 shows an example when the program is asked to remove the file named "a.docx".

4.1.3 Search

Figure 8 shows an example of search for "a.jar".

4.2 Complexity of operations

Table 1 shows the complexity of the operations of the data structure designed.

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

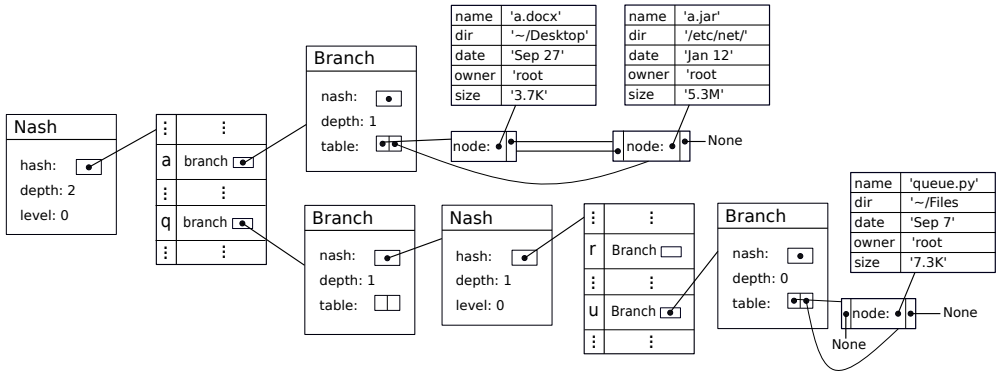

Figure 5: Example of NashTable

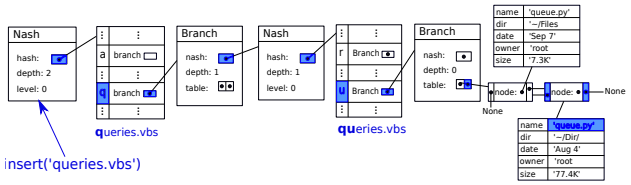

Figure 6: Insertion for NashTables.

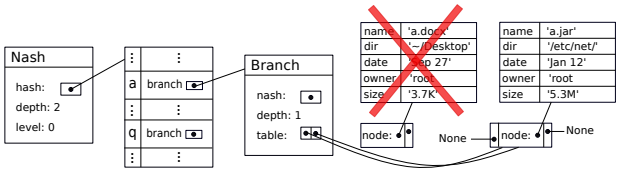

Figure 7: Deletion for NashTables.

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. For example, insertion has an average time of $5.78\mu s$, which is really fast. Although the complexity of the operations seems slightly inefficent, when the algorithms are applied, different results are obtained;

Table 1: Complexity of operations of NashTables

Operation	Complexity
Read data	$O(tl)$
Insertion	$O(n)$
Search	$O(s^2)$
Remove	$O(s)$

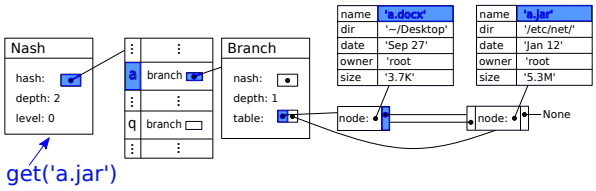

Figure 8: Search for NashTables.

Table 2: Execution time for each operation.

Operation	WorstTime	BestTime	AverageTime
Read data	$114478\mu s$	$53179\mu s$	$69442\mu s$
Insertion	$27\mu s$	$5\mu s$	$5.78\mu s$
Search	$6\mu s$	$1\mu s$	$1.61\mu s$
Remove	$20\mu s$	$5\mu s$	$6.25\mu s$

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 computer are called the same. All operations - read data, insertion, search, remove - were executed fast enough.

5. REFERENCES

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

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. The MIT Press, Cambrigde, Massachusetts, 2001.

[3] D. P. Mehta and S. Sahni. *Handbook of data structures and applications*. CRC Press, 2004.

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

Table 3: Memory consumption.

MemoryConsumption	DataSet1
Memory	40.96MB