# DESIGN OF A DATA STRUCTURE THAT OPTIMIZES SEARCH EFFICIENCY IN A DIRECTORY (SEARCH SPACE).

Alejandro Murillo González
Universidad Eafit
Colombia
amurillog@eafit.edu.co

Juan Pablo Vidal Correa
Universidad Eafit
Colombia
jpvidalc@eafit.edu.co

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

## ABSTRACT

Every day massive amounts of digital information must be handled through technological devices. This brings challenges regarding the manipulation of large amounts of data, such as search time complexity and information retrieval. Therefore, the development of an effective data structure that optimizes storage and search becomes a priority to support new technological innovations. Current approaches have problems optimizing memory usage, even though they are -in most cases- kept within the limits of high-performance systems, which are greater than the computation power of mobile devices; but also emphasize search speed. Then, approaches that tackle memory overuse and search time turn critical to be able to work with exponentially growing search spaces and to bring the technological advances to devices with limited memory.[3]

This work proposes a data structure that optimizes search efficiency within a search space, specifically a Directory, and intends to keep memory usage as low as possible. Search complexity in this data structure is O(1), and memory consumption is fair given the response times, meaning a very good approximation to the solution of the problem, since it is acceptable to use large amounts of memory, if by doing so we can speed up access times.[1]

**Keywords**

Data structures, efficiency, complexity, memory, Search Space.

**<u>Computing classification system concepts</u>**

• Theory of computation → Data structures design and analysis → Shortest paths →Record storage systems

## 1. INTRODUCTION

During the 20th century, the technological advances accelerated exponentially thanks to the momentum produced by two world wars and the subsequent cold war, since these conflicts allowed our mastery of science and technology to increase. This evolution allowed today's people to enter in “the information age”, where humanity is daily besieged by massive amounts of information. The above has generated that our technological devices have to face the challenge of controlling large amounts of data, product of the information that is constantly received. This in a short time due to the great information traffic that is handled.

This information is manipulated with file systems, which are responsible for assigning space to the files, managing the free space and allowing access to the saved data. This generates the problem that some file systems are overwhelmed by the immense flow of information, since they do not support the amount of information or their response time is very limited for what is currently needed.

Due to the above, the current file systems must be replaced to guarantee efficiency, this is the case of ext2, a file system, which by maintaining a linked list allocation system, that needs to review each block to find the data, shows problems when handling large amounts of information, and that is why it is replaced by ext4, which can support information volumes of up to 1024 PiB (Pebibyte).

## 2. PROBLEM

Design a file system that optimizes the search and information retrieval processes, to improve the efficiency to find files and subdirectories in a directory with a large amount of information. This would allow the file system to be updated to face new challenges in the future as regards of the handling and collection of information.

## 3. RELATED WORK

### 3.1 B-Trees:

The B-tree aims to solve the problem of given a large collection of objects, each having a key and a 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.

<u>Search:</u>

Finding a key in a B-tree is a simple operation because it consists of locating in the root node of the tree, if the key is there we have finished and if not we select from among the children the one that is between two key values which are smaller and larger than the required one respectively, and we repeat the process until we find it. In case a sheet is reached and we cannot continue the search, the key is not in the tree.

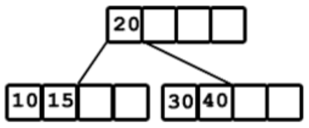

**Figure 1:** Example of B-Tree

### 3.2 B+ Tree

The B + trees are an improvement over the B trees, because they preserve the property of rapid random access and also allow a fast sequential route. In these all the keys are in sheets, duplicating in the root and internal nodes those that are necessary to define the search paths. The problem with these trees is that they need more memory space than the B trees, since there is duplication in some keys, so that the B tree has fewer nodes. In addition, in B + trees the keys of the root and interior pages are used only as indexes.

Search:

The search should not stop when the key is found on the root page or on an inside page, but should continue on the page pointed to by the right branch of that key.

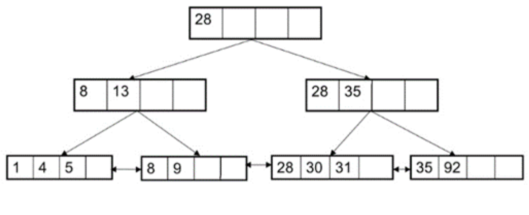

**Figure 2**: Example of B+ Tree

### 3.3 Hash tables

Hash tables are data structures that are used to store a large number of data on which very efficient search and insertion operations are needed. A hash table stores a set of pairs "(key, value)". The key is unique for each element of the table and is the data that is used to find a certain value.

Search:

A table of a reasonable size is created to store the pairs (key, value) and a hash function that receives the key and returns an index to access a position in the table. There are cases in which the function returns the same index for two different keys, this is known as a **collision**. There are multiple libraries in almost all programming languages that provide very efficient implementations of these tables (python have a O(1) hash table and have a robust algorithm that prevents many collisions).

Solution for the collision:

It can combine the table structure with the linked list structure, which allows each position of the table to store in the head of a linked list that contains all those elements whose hash function returned the same result. If there is no element in the position, then it contains a pointer to NULL.

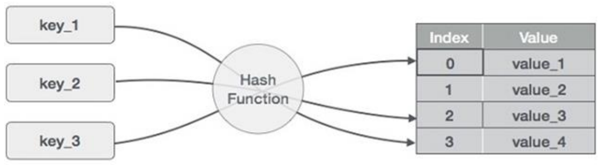

**Figure 3:** Example of Hash table

### 3.4 Red-black trees

Before defining a red-black trees, it must be explained what it is a 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 node0s key.
- The right sub-tree of a node has a key greater or equal to its parent's nodes key.

A red-black trees is a binary tree in which each node has a color as an extra attribute, either red or black. The color of the nodes ensures that the longest path of the root to a leaf is no longer than twice the length of the shortest. This means that the tree is strongly balanced.

Search:

It consists in accessing the root of the tree and comparing its value with the desired value. If the element to be localized coincides with that of the root, the search has concluded successfully. If the element is smaller, it looks in the left subtree; if it is greater, in the right. If a leaf node is reached and the element has not been found it is assumed that it does not exist in the tree. It should be noted that the search in this type of trees is very efficient and represents a logarithmic function.

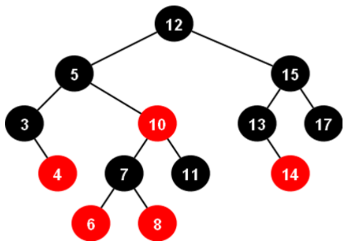

**Figure 4**: Example of Red-black tree

## 4. TREEHASH:

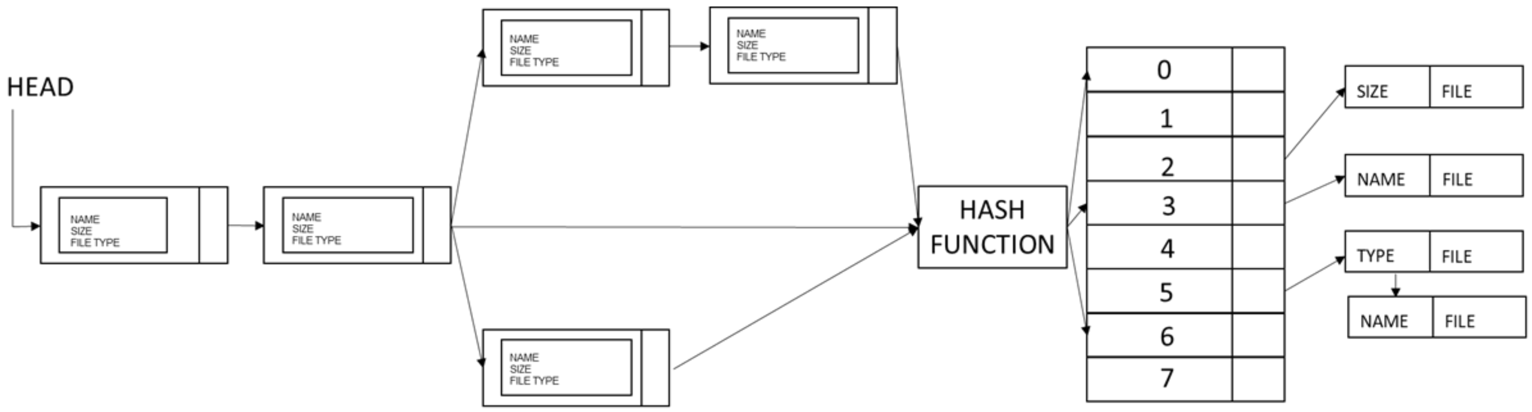

**Figure 5**: Example of TreeHash

### 4.1 Operations of the data structure

4.1.1 Search

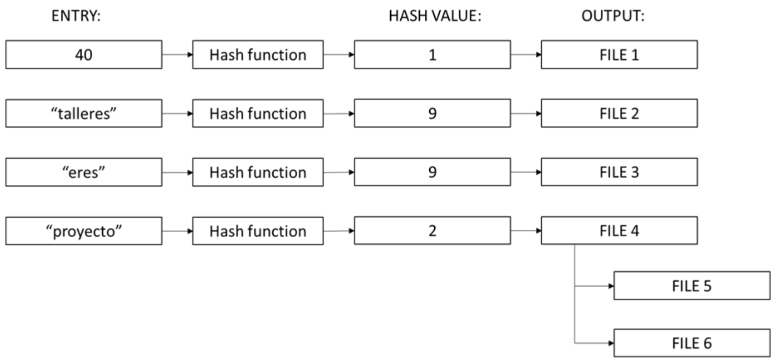

**Figure 6**: Search for TreeHash

### 4.2 Design criteria of the data structure

To design the data structure we took into account the efficiency in time, files that contain other files and the comparison between words, that is to say words or names that contain similar characters, but that correspond to different files, such as searching for the "w "And that the result shows both" workshop "and" work "because both start with the same letter. Due to the above, a simply linked list was implemented whose final node starts other lists simply linked (see graph 5), which collect all the files in the directory and hierarchize them, taking into account inheritance relationships, where a file contains other files In addition, each node contains the user, the size and the name of the file; this allows us to deal with the problem of inheritance between files, and also optimizes the time spent collecting and traversing the entire data set, this allows to pass the data already organized to a hash function, which gives a hash value to each file according to its size, file name and / or file type (in the case of Root files), which allows to cover the problem of files with similar names, because it will assign them the same hash value, which will allow displaying the files with the same name, allowing to cover a search more similar to the algorithms used by search engines, which show similar searches. The previous combination between a linked list and a hash table allows to optimize the time, since it organizes the data, cataloging them for a faster search, and also preventing collision.

### 4.3 Complexity analysis

| **Operation** | **Complexity** |
|---|---|
| Read data | *O(1)* |
| Insertion (hash table) | *O(1)* |
| Search | *O(1)* |
| Print search | *O(n)* |

**Table 1**: Complexity of operations

### 4.4 Execution time

| | **DataSet 1** | **DataSet 2** |
|---|---|---|
| Creation | 0.0013179243639414946 *ms* | 0.1978782839961042 *ms* |
| Search | 2.7441e-11 *ms* | 5.63e-10 *ms* |

**Table 2**: Execution time of the operations of the data structure for each data set.

### 4.5 Memory used

| | **DataSet 1** | **DataSet 2** |
|---|---|---|
| Memory consumption | 23,5 *MB* | 44,7 *MB* |

**Table 3**: Memory used for each operation of the data structure and for each data set data sets.

The DataSet 1 has 3 directories, 18 files files and the DataSet 2 has 5 directories and 25 files.

## 4.6 Result analysis

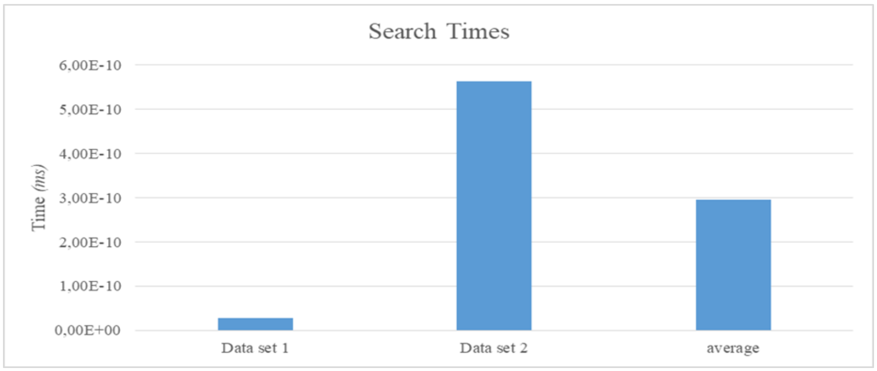

**Figure 7**: Comparison of search time between sets and their average.

The generated data structure fulfilled the proposed criteria, since as it could be observed with the data obtained about the execution time for the test data sets, a minimum time for the search can be evidenced (see figure 7), which is the main objective for the problem that we are solving, so it can be said that the algorithm optimizes the search times. Otherwise memory consumption shows a problem, since the algorithm consumes a lot of resources for its implementation, this due to the use of linked lists, which in turn generate other linked lists which generates a very large memory expense, despite speed, so it is necessary to consider the memory expense for future implementations.

## 5. SEARCH SPACE:

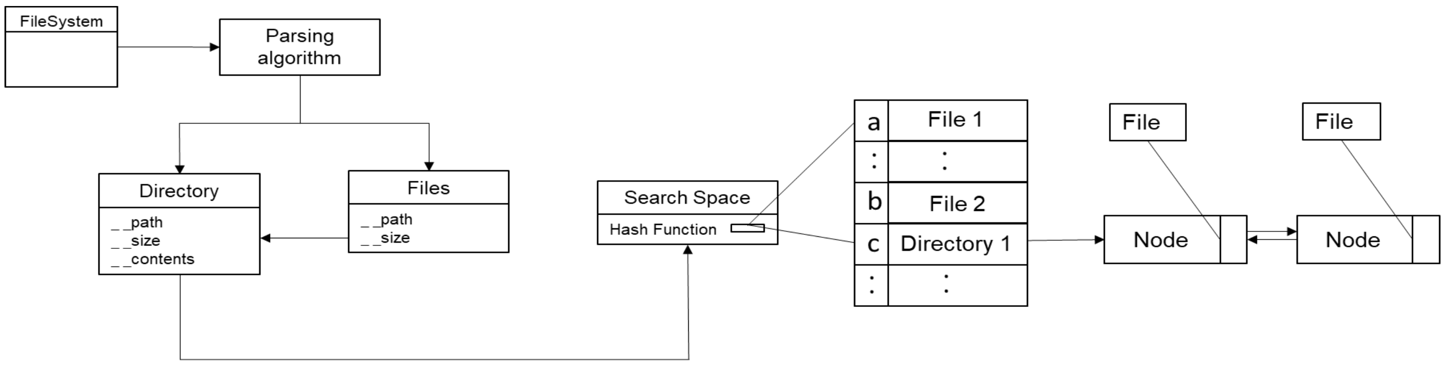

**Figure 7:** Search space structure.

## 5.1 Operations of the data structure

### 5.1.1 Search

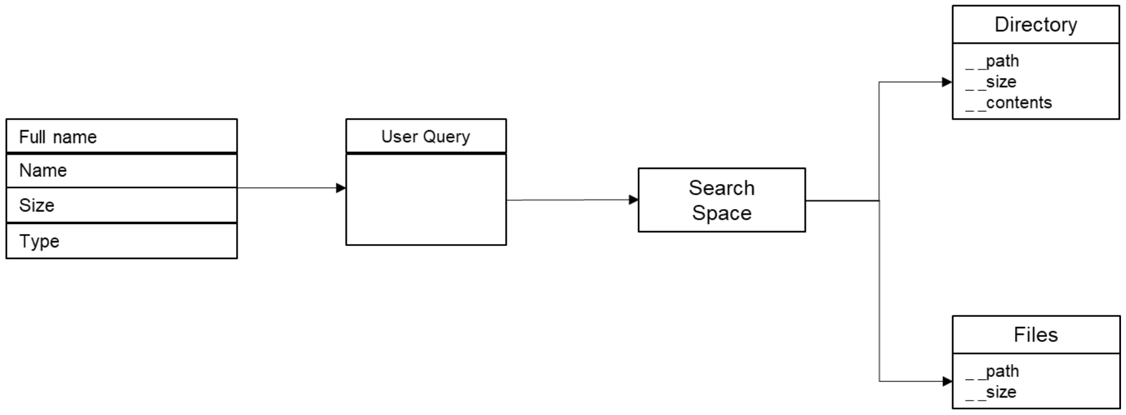

**Figure 7:** Search for space structure.

### 5.1.2 Insertion

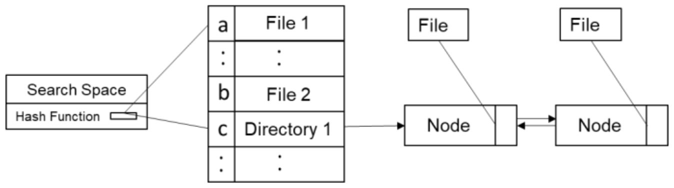

**Figure 7:** Insertion for space structure

## 5.2 Design criteria of the data structure:

The design of the data structure considered access time and memory consumption combined with the search parameters that can be used to find a certain file; these might be: name, full name, type, size and path. The latter takes into account that the search space features a hierarchical arrangement of files. This enables users to retrieve the same file independently of the parameters used in the query. Thereby we see hash tables as an appropriate alternative, because it allows to give a hash value to each search parameter (of each file), and retrieve the associated file's path fast. The downside is that the same file will be referenced multiple times resulting in increased (but no considerably so) memory usage; we try to address this by making a tradeoff between what is stored in each File/Directory object and the retrieval time of some of the properties of each of these objects, what we do is compute from each instance of these objects' path property some of the parameters that identify it (mentioned previously).

To use hash tables involves dealing with collisions, a known drawback of this data structure. The data structure designed is programmed using Python, which uses a robust hashing algorithm for its dictionary implementation and that works well with its String implementation. In case of collisions, due to the hashing algorithm -which are uncommon- or because is another reference to the same parameter, the references are added in a doubly Linked List. By storing references in a Linked List, we can maintain the hierarchical order of the directories/files and access directories contents fast. Other advantage of Linked List is insertion, deletion complexity of O(1) and traversal of O(n).

Since hash tables' access time compares favorably against other data structures -and considering industry success stories such as Amazon Web Services' use of NoSQL databases in its widely used product DynamoDB, which makes use of a partition key for retrieval of information, similar to a hash table-, the multiple references to a single file concern is considered disregarded.

## 5.3 Complexity analysis

| **Operation** | **Complexity** |
|---|---|
| Read data | *O(1)* |
| Insertion (hash table) | *O(1)* |
| Search | *O(1)* |
| Print search | *O(n)* |

The hash tables implemented in python are O(1).[2]

**Table 5:** Table to report complexity analysis of the data structure.

## 5.4 Execution time

| | DataSet 1 | DataSet 2 |
|---|---|---|
| Create | 0.0028356753674414946 *s* | 0.07583519934008941 s |
| Search | 1.9451e-11 *s* | 5.688884393961750e-06 *s* |

**Table 6:** Execution time of the data structure's operation for each data set.

## 5.5 Memory Usage

| | *DataSet 1* | *DataSet 2* |
|---|---|---|
| *Memory consumption* | 13,5 *MB* | 28 *MB* |

**Table 7:** Memory used by each operation of the data structure in different data sets.

The DataSet 1 has 3 directories, 18 files files and the DataSet 2 has 425 directories and 3225 files. [5]

## 5.6 Result analysis

This data structure enables search in constant time. Thus, search time complexity is lower than other approaches when compared with the same data sets (see Figure 8 and table 8); which was one of the fundamental design criteria. On the other hand, the memory usage is not as optimal as the access time complexity, because multiple references -even though they, by themselves, are small- were stored in the hash table for the same file; but is a tradeoff supported by other authors such as Lafore, who says that it is acceptable to use large amounts of memory, if by doing so we can speed up access times.[1]

## 6. CONCLUSIONS

While discussing the best approach to achieve the objective of generating a data structure appropriate to handle -efficiently- large amounts of data and search through it, the core question was how to optimize time complexity and memory usage. The different approximations led us to conclude that constant search times made greater memory usage worth the shot.

The data structure designed is able to meet the criteria: time optimization and efficient memory usage, because it was possible to see minimum search times (see Table 6), and keep memory usage within appropriate limits.

When the data obtained in the first approximation is compared with the data from the final implementation, it is possible to notice significant improvements in memory usage. With respect to search times, the change is subtle, given that search times in the first approximation were already good. Therefore, it is correct to say that there is a gain in general performance and that the tradeoff between memory and search complexity is better balanced.

Storage, due to the need to point multiple times to the same file from different identification parameters, makes the data structure not completely optimal, but this is countered by making instances of different objects smaller and relying on computing some of their properties as needed.

### 6.1 Future work

Memory usage, by avoiding multiple references from the hash table to the same object, is a place to further improve the proposed data structure. This could be by optimizing the hash table or taking a different approach in the way objects are referenced. Being able to narrow, by putting in place filters, the search space should also be take into consideration.

Insertion after the file containing the initial data is parsed should not be a problem given the characteristics of the data structure. Implementing deletion and edition capabilities, which could have complexity O(n), should be possible. Hierarchy changes is also a problem worth working on.

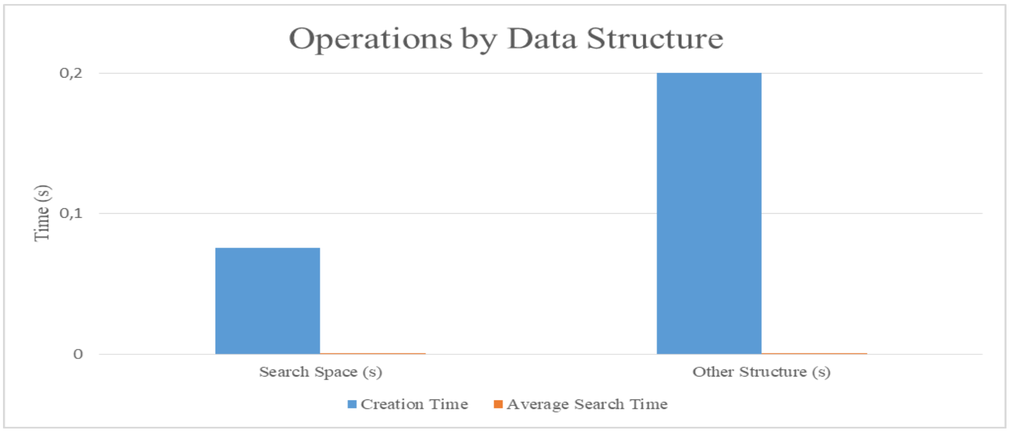

**Figure 8:** Comparison between access times with other types of structures (Nash table). [6]

| Data Set 1 | Search Space (s) | Other Structure (s) |
|---|---|---|
| Creation Time | 0,075835199 | 0,213355 |
| Average Search Time | 5,69E-06 | 2,60E-05 |

**Table 8:** Comparison between access times with other types of structures (Nash table). [6]

## REFERENCES

1. Lafore, R. *Data structures and algorithms in java*. Sams Publishing, Indianapolis, 2003.
2. Hartley, J. TimeComplexity, *Python*. Retrieved October 11, 2017 from Python Wiki: https://wiki.python.org/moin/TimeComplexity.
3. Purdy,M anb Daugherty P. Why Artificial Intelligence is the Future of Growth, Accenture. Retrieved September 11, 2017 from: https://www.accenture.com/us-en/_acnmedia/PDF-33/Accenture-Why-AI-is-the-Future-of-Growth.pdf
4. DataStructures Tool. Available from: http://www.hci.uniovi.es/Products/DSTool/b/b-queSon.html (2012); accessed 10 August 2017.

**Github of the DateSet:**

5. Toro,M.MauricioToro.Github. Retrieved October 29, 2017 from Github: https://github.com/mauriciotoro/ST0245-Eafit

**Github of the other structure:**

6. Cardenas,J.S. and Plazas,D.Daplas,Github.Retrieved October 29, 2017 from Github: https://github.com/Daples/ST0245-032