

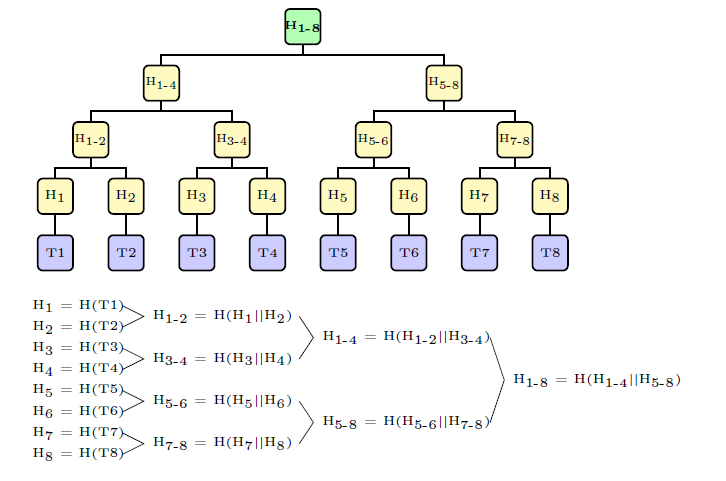
Programming III

Semester Project – Merkle Trees

* Students:
* Gelsi Federico
* Huaman Melgarejo Cecilia
* Santos Nicolás
* Professor:
* Wehbe Ricardo
* Class:
* UADE – Team 2 – Merkle Tree

# Task

A Merkle tree is a binary tree in which every leaf node is labelled with the cryptographic hash of a transaction. They provide a kind of summing up of the transactions contained in a block. For instance:



We assume a blockchain structure in which each full node has a copy of all transactions and a copy of the root of the corresponding Merkle tree (the blue and green nodes) and each thin node has but a copy of the root of the Merkle tree (the green node.) A thin node may want to know whether a transaction, identified by a given number, is in the block or not. Since the root of the tree is not falsifiable because of the properties of the hash functions, it does not want the full node to merely answer “yes” of “no”, but it wants to verify it by recalculating the path up toward the root of the tree. For that, it needs that the full node sends it the missing elements. Of course, the full node may send it the whole bunch of hashes. But this would be a lot of traffic. There are more efficient ways to solve this problem.

Your task is:

1. Design and implement a full node. You have to choose the data structures you consider more appropriate to solving this problem. Notice that the full node is only asked to hold the transactions and the root of the tree; you may choose whether to calculate the intermediate nodes or to have then pre-computed in memory.
2. Design an efficient algorithm so that the full node answers a hypothetical request of a thin node by sending the hashes that are necessary to compute the path to the root and only these hashes. The full node receives the number of the transaction.
3. Implement your algorithm in Java. You are expected to deliver the source files and a short report detailing the strategy of the algorithm and explaining the problem you are solving in detail. Since this is just a prototype, you may use the Java hashCode function as a hash function. This is not what you would do in real life, though.

# Submission and Acceptance

The Semester Project must be submitted through the WebCampus groups. No other delivery will be accepted.

* The final implementation version together with the report must be delivered by 6th November. No delivery will be accepted after this date.
* If corrections are required, the new delivery date is 13th November. This is the definitive date.
* Defense of the Semester Project will take place on 13th November and 20th November. No defense is possible if the Semester Project has not been accepted.

Please notice that after the first delivery only one additional delivery can be made if the work was not approved, assuming the first one was submitted in due time. If the work was not delivered on the first date, there is no possibility of from another submission. The approval of the Semester Project implies the approval of both the delivery and the oral exposition.

**Explanation of the problem**

Given a full node, that we call 'Block', a transaction and the position of the transaction, we have to verify that the transaction that the input give us it is in the data structure called Merkle Tree.

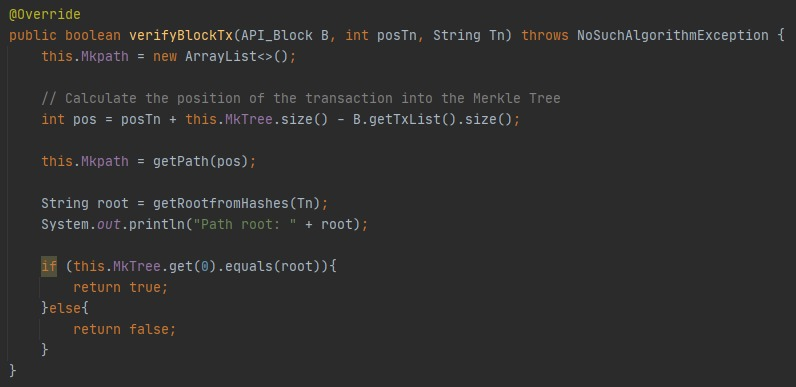
To do this, we could recreate the whole Merkle Tree with our given transaction instead of the transaction that was in the original Merkel Tree and hashing all transaction and travelling the tree form bottom to top. However, this has very low performance because we will need to travel the whole tree given us a O(n) complexity, in our case at least because we implemented as an array of strings, so we need to find a better way to check if the transaction belongs to the tree or not.

**Proposed solution**

The better way to do this, it is by taking only the hashes that are necessary to recreate the root of the Merkel tree, having log2(Tx) hashes instead of 2\*Tx-1 hashes, being Tx the amount of transactions that compose the tree.

We accomplish this by using the divide and conquer technique that we will explain later.

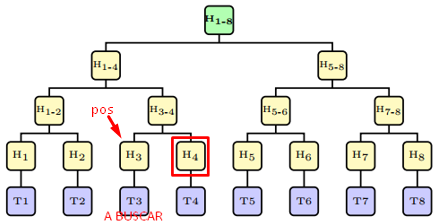
Here it is our proposed algorithm:



Below there is an example of how this algorithm works.

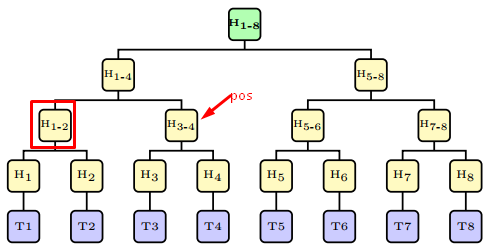
**Explanation of the above algorithm**

First, the algorithm starts at the transaction position that we want to check if is it in the Merkle Tree and add the hash of the brother into the array ‘Arr’, that will be returned once that we have reach the root, being this the base case.

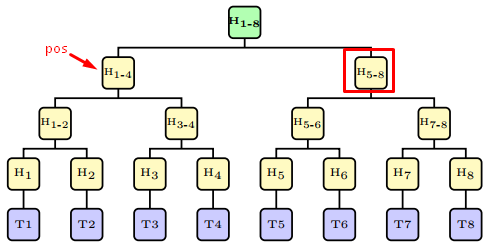


In this case we want to check T3. We add H4 into Arr.

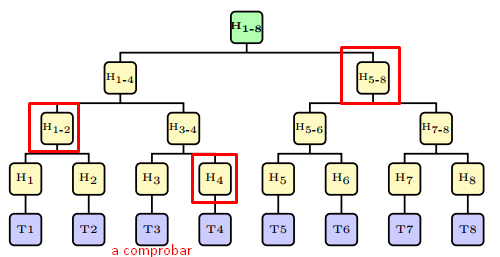
Next, the actual position ‘pos’ that was in H3 goes to the father of H3, in essence, H3-4 and add the hash H1-2 into Arr.



And then ‘pos’ goes to his father that is H1-4 and add the H5-8 hash to Arr.



Finally ‘pos’ goes to his father that is the root of the tree, so in the next call the algorithm will reach the base case finishing the recursion and returning the array ‘Arr’ with the hashes that are necessary to check if the transaction, in this case T3, it is in the Merkle tree.

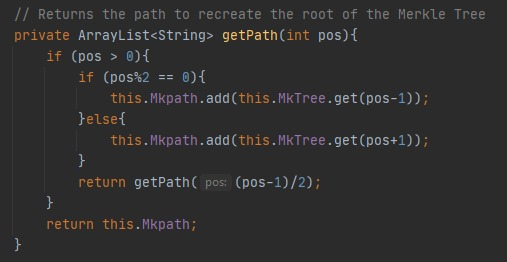


It shows the hashes returned by the algorithm “getPath (..)”

Following the pictures above, one can notice that the algorithm does not travel all the nodes of the tree, instead it goes dividing the tree in half getting subtrees until we reach the root.

**Functions that we use in the algorithm.**

getPath(..) function



Here we use the divide and conquer technique.

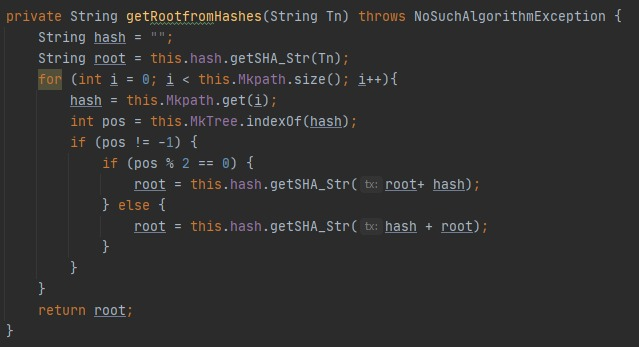
As we say before, this receives the position of the transaction that we want to check and travels up in the tree until the ‘pos’ reach the root and therefor the base case.

**Complexity:**

a = 1 ; b = 2 ; k = 0

So, a = bk 🡪 1 = 20, so the complexity is O(nk \* log n) = O(n0 log n) = **O(log n)**.

getRootfromHashes(..) function



Here we calculate the hash of the new root with the transaction to check and the hashes that are necessary to calculate this new root, that are returned by the ‘getPath(…)’ algorithm.

**Conclusion**

After doing all the research and thinking about how to implement an algorithm with a high performance we ended up with a really good application of the divide and conquer technique and we learnt a lot of things of the blockchain field

We also think that it is very useful to verify and check de consistency of the data. As we saw it has a great value in blockchain due to the sensible data such as the transactions and it is good to see that the Merkle Tree has a lot applications like in data bases and version control such as Git.

**Appendix**

*Source code:* <https://github.com/FedericoGelsi/Programming-III_Merkle-Tree>

*Bibliography:*

* <https://brilliant.org/wiki/merkle-tree/#complexity>
* <https://adr-rod87.medium.com/qu%C3%A9-es-un-merkle-tree-5f7c4f3c116a>