Published in Coinmonks

Jeremy Then

May 21, 2022 · 7 min read · Listen

Merkle Tree, a simple explanation and implementation

Merkle Tree, a simple explanation and implementation

The purpose of this article is to have a basic, but complete understanding of the Merkle Tree and its implementation.
Since I couldn't find an article covering the implementation of the Merkle tree creation, Merkle root, Merkle Proof, and validation in a simple way, I decided to implement a basic solution and create this article with my findings to share with the community.

Merkle Tree

A Merkle Tree is a tree data structure (typically a binary tree) of hashes, where each leaf node contains the hash of a block of data (eg: the transaction hash in a blockchain), and each parent node contains the hash resulting from the concatenation and hashing of its children node's hashes.

This tree is used to verify and demonstrate that the hash of a leaf node is part of the hash of the root node in an efficient way, since we only need a small set of the hashes of the tree to carry out this verification.

This data structure is often used in distributed systems, like in Bitcoin and other blockchains, to verify if a transaction hash belongs to the Merkle root of a block header, in a lightweight, efficient and fast way.

The use of Merkle Tree/Root/Proof in Bitcoin allows the implementation of Simple Payment Verification (SPV), which is a way for lightweight clients to check if a transaction is actually part of a block, without the client having to download the whole block or the whole blockchain.

By only having the block header with the Merkle root, the transaction it wants to verify, and the Merkle proof structure obtained from a trusted Bitcoin node, it can try to reconstruct the Merkle root hash and validate the transaction.

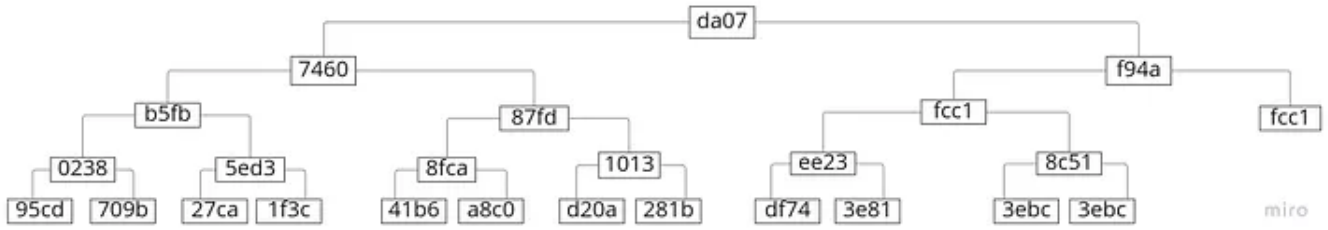
Ralph Merkle patented the hash tree in 1979, later named after him.

Example of a Merkle Tree

From a list of hashes:

```
[95cd, 709b, 27ca, 1f3c, 41b6, a8c0, d20a, 281b, df74, 3e81, 3ebc]
```

Its Merkle tree would look like:



(These 4 character hashes are short for longer, 64 characters hashes, to be shown later in the article)

Notice how the last hash, `3ebc`, was copied and added to the end of the list. This is needed to be able to concatenate it with itself and hash it, since we hash in pair and if the hash list length is odd, then we copy it and add it to the end of the list to make the hash list even. The same happens with `fcc1`.
Each of the tree levels is called a hash list.

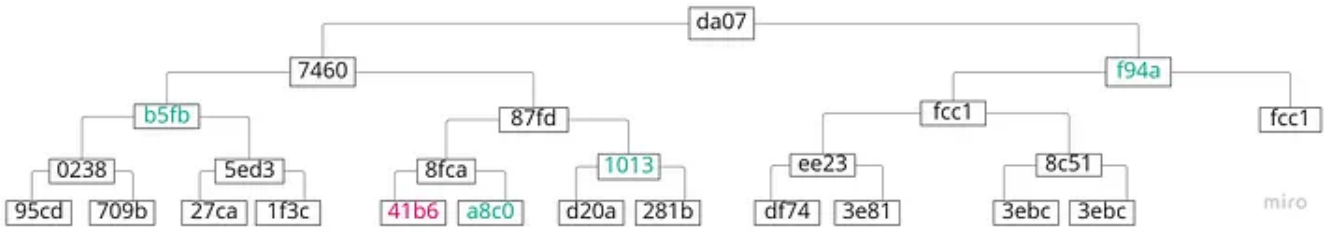
Merkle Root

The Merkle root is the root node of the tree. In our example above, it's the node with the **`da07`** hash.

Merkle Proof

A Merkle proof (also known as Merkle path) is a structure that holds the minimum needed hashes/nodes of a branch of the tree to be able to prove that a hash belongs to the Merkle root by recreating the Merkle root only with this information.

For example:



If we are interested in verifying if the hash `41b6` is actually included in the Merkle root (`da07`) then we would only need to have the hashes in color:

```
[41b6, a8c0, 1013, b5fb, f94a]
```

With `41b6`, we need `a8c0` to reconstruct `8fca`.
With `8fca`, we need `1013` to reconstruct `87fd`.
With `87fd`, we need `b5fb` to reconstruct `7460`.
With `7460` we need `f94a` to reconstruct **`da07`**, the Merkle root.

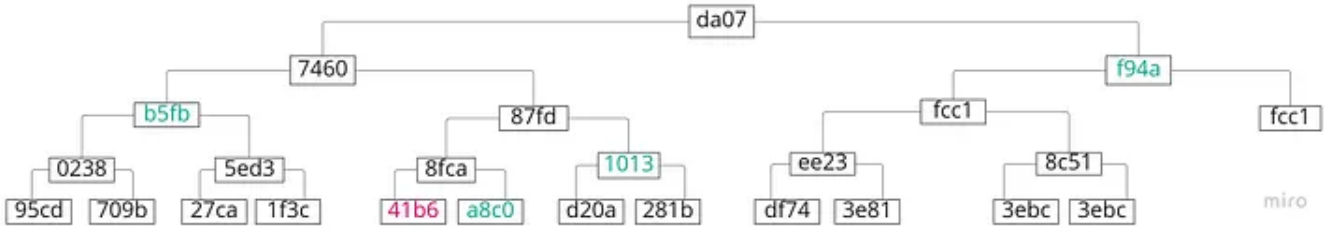
If after applying our hash (`41b6`) to the Merkle proof we get the expected **`da07`** Merkle root, then we know that our hash is part of the Merkle root.

In this case, for 11 hashes, we only needed 4 hashes, along our hash `41b6` to be able to reconstruct the Merkle root **`da07`**.

With this information, we notice a pattern and we can determine that we only need about log(n) number of hashes to be able to check if a hash belongs to a certain Merkle root, which is really efficient, instead of having to hash them all or to have all the hashes to verify that.

For example, if a block in Bitcoin has about 1000 transactions, we would only need to retrieve the Merkle proof from a trusted Bitcoin node with about 10 transaction hashes, instead of the whole block with the 1000 transactions.

Something important that we need to have in mind is the order of concatenation of the hashes when we are creating and using the Merkle proof before hashing them:



`41b6` needs to be concatenated with `a8c0`, `41b6` being on the left side and `a8c0` on the right side:

Because `41b6` is a left child, and `a8c0` is a right child.

The resulting hash `8fca` from the previous concatenation and hashing needs to be concatenated with `1013`, `8fca` being on the left side and `1013` on the right side, because `8fca` is a left child and `1013` is a right child.

The resulting hash `87fd` from the previous concatenation and hashing needs to be concatenated with `b5fb`, `87fd` being on the right side and `b5fb` on the left side, because `87fd` is a right child and `b5fb` is a left child.

The resulting hash `7460` from the previous concatenation and hashing needs to be concatenated with `f94a`, `7460` being on the left side and `f94a` on the right side, because `7460` is a left child and `f94a` is a right child.

We need to specify in the Merkle proof structure this direction that each node hash should be concatenated with:

The concatenation order would be as follows:

```
41b6 + a8c0 => 8fca
8fca + 1013 => 87fd
b5fb + 87fd => 7460
7460 + f94a => da07
```

With this, our Merkle proof is complete and ready to be transferred and used.

Basic Implementation

Blockchains like Bitcoin use Merkle trees to check if a transaction belongs to the Merkle root in a block header.
Each software using the Merkle tree structure is free to use any implementation they see fit.

Bitcoin, for example, converts the concatenated hash to binary first, before hashing it.
Bitcoin also uses a double sha256 hash, something like:

We're going to keep our implementation as simple as possible, just to try to see and understand the concept easily, just the bare minimum.

Our implementation will be in nodejs.

We are going to use `sha256` to hash the concatenation of our hashes. So, we will have 64 character hashes.

Each function will have some documentation to help clarify what it does.

You can find the complete code here.

First, we import the `crypto` module:

Define a couple of direction constants:

Define a list of random sha256 hashes to work with:

Define a utility function to help us abstract the sha256 hashing with the crypto module:

Define a utility function to calculate if a leaf node is a left or right child:

Define a utility function to check if a list of hashes has an odd length, and if so, copy the last hash and add it to the end of the hash list:

Define the function that will receive a hash list and calculate and return the Merkle root, recursively:

We can test this code by calling the `generateMerkleRoot` with the list of hashes:

This would print:

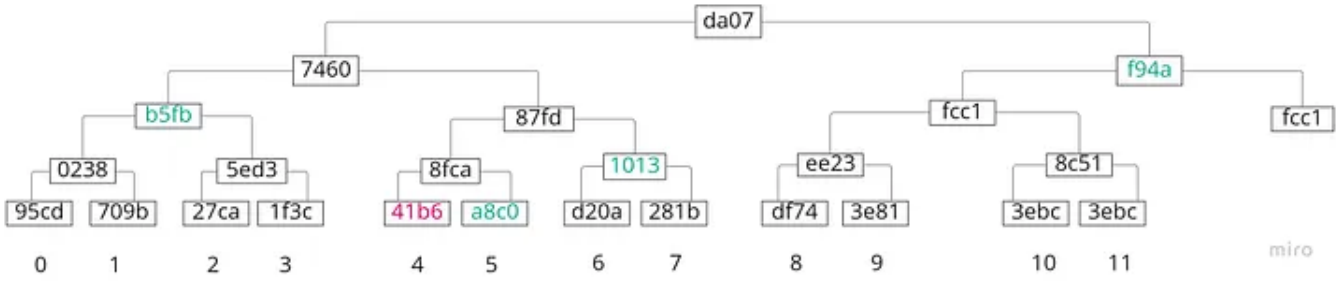
```
merkleRoot:
68e6cdf0cae7fb8eef39cc899c8882e34dd1727a2d08f2303811886949c539e6
```

Let's now generate the Merkle tree:

We can try out this function by calling it with the hashes:

This will long something like:

Let's now generate the Merkle Proof structure:



For example, if we want to find the parent node of `41b6` we need to know its index first. We know that the index of `41b6` is 4, so if we divide 4 / 2 and round the result, we can find its parent: 4 / 2 => 2. So, 2 in the index of the parent node of `41b6` in the next level.

If we want to find the parent node of `a8c0` , we do the same. Its index is 5, so 5 / 2 => 2 (rounded down with Math.floor). Then we see that `41b6` and `a8c0` share the same parent.

This works the same for finding all the parents of any nodes in the tree, except for the root node since it does not have a parent.

Let's call the function with a hash and the list of hashes:

And it logs:

Now, let's add a function to verify if a hash is part of a Merkle root, by reconstructing a Merkle root from the Merkle proof structure:

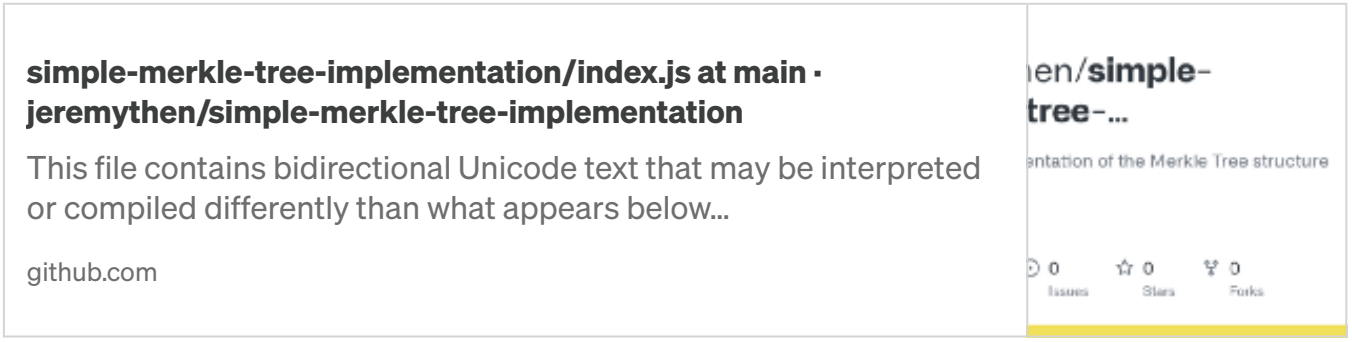
Let's call it and check its result against our expected Merkle root:

This will log:

```
merkleRootFromMerkleProof:
68e6cdf0cae7fb8eef39cc899c8882e34dd1727a2d08f2303811886949c539e6
merkleRootFromMerkleProof === merkleRoot:  true
```

Nice.

See the whole code here:



Follow me for more blockchain-related topics.

References:

https://learnmeabitcoin.com/technical/merkle-root
https://en.wikipedia.org/wiki/Merkle_tree

Merkle | Merkle Tree | Merkle Root | Merkle Proof | Bitcoin

Your email

Get this newsletter

By signing up, you will create a Medium account if you don't already have one. Review our Privacy Policy for more information about our privacy practices.

More from Coinmonks

Follow

Coinmonks (http://coinmonks.io/) is a non-profit Crypto Educational Publication. Follow us on Twitter @coinmonks and Our other project—https://coincodecap.com, Email —gaurav@coincodecap.com

Ren & Heinrich · Jan 18

AI Cryptos are the Next Big Thing

Share your ideas with millions of readers.

Write on Medium

Gaurav Agrawal · Dec 28, 2020

Best 18 FREE Crypto Trading Bots in 2023

TheLuWizz · Mar 10, 2021

12 Best FREE Crypto Telegram Channels 2023 (Trading Signals)

Gaurav Agrawal · Jun 3, 2021

4 Best Free Crypto Signals 2023 — Telegram Trading Signals

Mike Coldman · Feb 4

Cryptos to Buy and Hold Before BullRun 2023!

Read more from Coinmonks