LeanIMT: An optimized Incremental Merkle Tree

Privacy & Scaling Explorations

June 18, 2024

1 Abstract

Contents

2 Introduction

2.1 Motivation

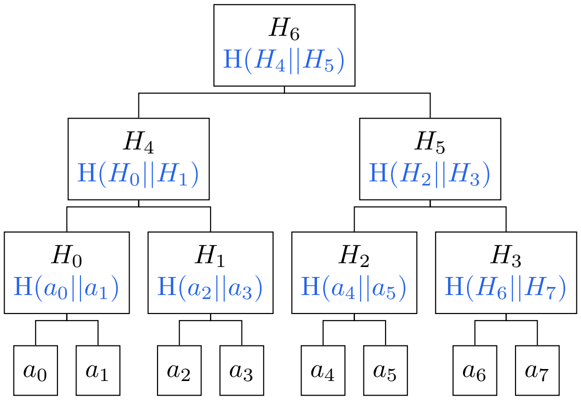
3 Merkle Tree

3.1 Incremental Merkle Tree

An Incremememental Merkle Tree (IMT) is a Merkle Tree (MT) designed to be updated efficiently.

3.2 Binary Tree

A Binary Tree is a tree data structure in which each node has at most two children, referred to as the left child and the right child.
TODO: Explain what is a Merkle tree and an Incremental Merkle Tree.



4 LeanIMT

4.1 Definition

The **LeanIMT** (Lean Incremental Merkle Tree) is a Binary IMT.

The LeanIMT has two properties:
1. Every node with two children is the hash of its two child nodes.
2. Every node with one child has the same value as its child node.

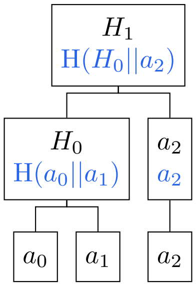
Example of a LeanIMT

T - Tree
V - Vertices (Nodes)
E - Edges (Lines connecting Nodes)

$T = (V, E)$

$V = \{a_0, a_1, a_2, H_0, H_1, H_2\}$

$E = \{(a_0, H_0), (a_1, H_0), (a_2, a_2), (H_0, H_1), (a_2, a_2)\}$



4.2 Insertion

There are two cases:
1. When the new node is a left node.
2. When the new node is a right node.

We will always see one of these cases in each level when we are inserting a node. It is like, when you insert a node, if that node is left node, the parent node which is in the next level, will be the same node. If it is a right node the parent node, will be the hash of this node with the node in its left. This algorithm will be the same in each level, not only in level 0.

Case 1: The new node is a left node

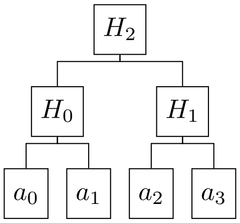
It will not be hashed, it's value will be sent to the next level.

If we add a_4.

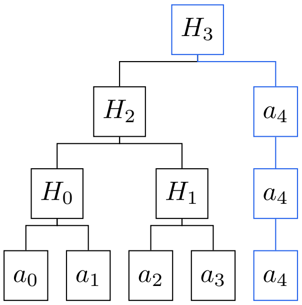
$T = (V, E)$

$V = \{a_0, a_1, a_2, a_3, H_0, H_1, H_2\}$

$E = \{(a_0, H_0), (a_1, H_0), (a_2, H_1), (a_3, H_1), (H_0, H_2), (H_1, H_2)\}$



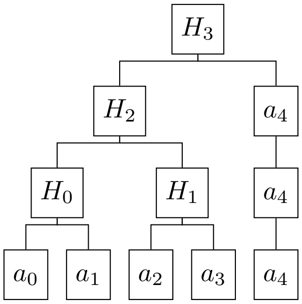
Before inserting a_4



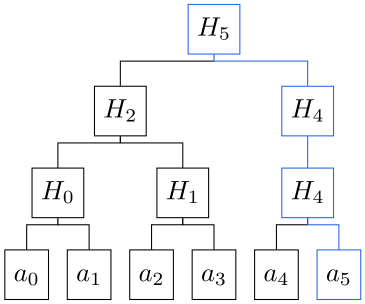
After inserting a_4

Case 2: The new node is a right node

If we add a_5.



Before inserting a_5



After inserting a_5

4.2.1 Pseudocode

Algorithm 1 LeanIMT Insert algorithm

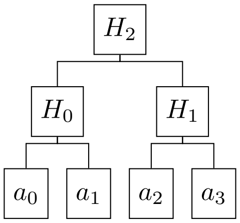
```
procedure Insert($leaf$)
    if $depth < newDepth$ then    ▷ $newDepth$ is the new depth of the tree after inserting the new node
        add a new empty array to nodes    ▷ Add a new tree level
    end if
    $node \leftarrow leaf$
    $index \leftarrow size$    ▷ The index of the new leaf equals the number of leaves in the tree.
    for level from 0 to depth - 1 do
        nodes[level][index] ← node
        if index is odd then    ▷ It's a right node
            sibling ← nodes[level][index - 1]
            node ← hash(sibling, node)
        end if
        index ← $\lfloor index/2 \rfloor$    ▷ Divides the index by 2 and discards the remainder.
    end for
    nodes[depth] ← [node]    ▷ Store the new root at the top level
end procedure
```

4.3 Batch Insertion

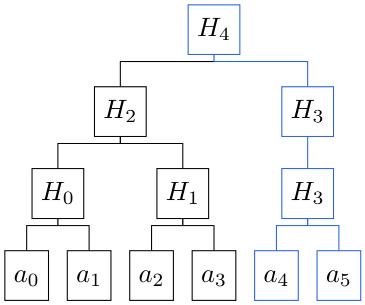
Performing the insertion in bulk rather than individually using a loop can lead to significant performance improvements. This optimization stems from the reduced number of hashing operations required. By inserting many elements at once, the algorithm can minimize redundant computations and manage memory more efficiently, resulting in faster execution and better overall performance.

When inserting n members, all levels will be updated n times if the batch insertion function is not being used.
The core idea behind the batch insertion algorithm is to update each level only once even if there are many members to be inserted.
The algorithm will go through the nodes that are necessary to update the next level of the tree. The other nodes in the tree won't be used or changed.

Insert a_4 and a_5.



Before inserting a_4 and a_5



After inserting a_4 and a_5

4.3.1 Pseudocode

Algorithm 2 LeanIMT InsertMany algorithm

```
procedure InsertMany($leaves$: List of nodes)
    $startIndex \leftarrow \lfloor size/2 \rfloor$ ▷ Divides the size of the tree by 2 and discards the remainder.
    Add $leaves$ to the tree leaves
    for level from 0 to depth - 1 do
        numberOfNodes ← $\lceil nodes[level].length/2 \rceil$ ▷ Calculate the number of nodes of the next level. numberOfNodes will be the smallest integrer which is greater than or equal to the result of dividing the number of nodes of the level by 2.
        for index from $startIndex$ to numberOfNodes - 1 do
            rightNode ← nodes[level][index * 2 + 1] ▷ Get the right node if exists.
            leftNode ← nodes[level][index * 2] ▷ Get the left node if exists.
            if $rightNode$ exists then
                parentNode ← $hash(leftNode, rightNode)$
            else
                parentNode ← $leftNode$
            end if
            nodes[level + 1][index] ← parentNode ▷ Add the parent node to the tree.
        end for
        startIndex ← $\lfloor startIndex/2 \rfloor$ ▷ Divide startIndex by 2 and discards the remainder.
    end for
end procedure
```

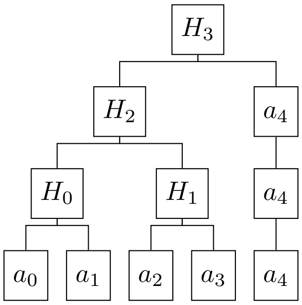
4.4 Update

There are two cases:

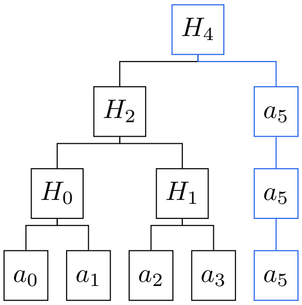
1. When there is no right sibling.
2. When there is right sibling.

Case 1: There is no right sibling

Update a_4 to a_5



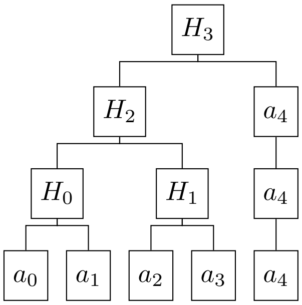
Before updating a_4



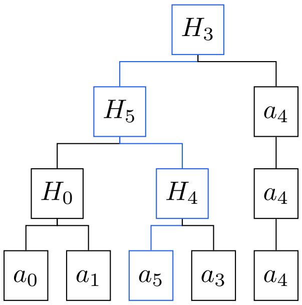
After updating a_4

Case 2: There is right sibling

Update a_2 to a_5



Before updating a_2 for a_5



After updating a_2 for a_5

4.4.1 Pseudocode

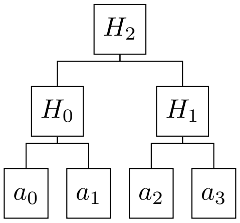
Algorithm 3 LeanIMT Update algorithm

```
procedure UPDATE($index$, $newLeaf$)
    $node \leftarrow newLeaf$
    for level from 0 to depth - 1 do
        nodes[level][index] ← node
        if index is odd then                                  ▷ It's a right node
            sibling ← nodes[level][index - 1]
            node ← hash(sibling, node)
        else                                                  ▷ It's a left node
            sibling ← nodes[level][index + 1]
            if sibling exists then               ▷ It's a left node with a right sibling
                node ← hash(node, sibling)
            end if
        end if
        index ← $\lfloor index/2 \rfloor$          ▷ Divides the index by 2 and discards the remainder.
    end for
    nodes[depth] ← [node]                    ▷ Store the new root at the top level
end procedure
```

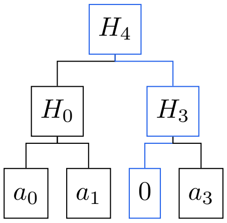
4.5 Remove

The *remove* function is the same as the *update* function but the value used to update is 0.
You can use a value other than 0, the idea is to use a value that is not a possible value for a correct member in the list.

Remove a_2



Before removing a_2



After removing a_2

4.5.1 Pseudocode

Algorithm 4 LeanIMT Remove algorithm

```
procedure REMOVE($index$)
    $update(index, 0)$
end procedure
```

4.6 Generate Merkle Proof

TODO: Add description with diagrams.

Algorithm 5 LeanIMT generateProof algorithm

```
procedure GENERATEPROOF($index$)
    siblings ← empty list
    path ← empty list
    for level from 0 to depth - 1 do
        isRightNode ← index is odd
        if isRightNode is true then                ▷ It's a right node
            siblingIndex ← index - 1
        else                                       ▷ It's a left node
            siblingIndex ← index + 1
        end if
        sibling ← nodes[level][siblingIndex]
        if sibling exists then
            add isRightNode to path
            add sibling to siblings
        end if
        index ← $\lfloor index/2 \rfloor$          ▷ Divides the index by 2 and discards the remainder.
    end for
    leaf ← leaves[index]
    index ← reverse path and use the list as a binary number and get the decimal representation
    siblings ← leaves[index]
    proof ← $\{root, leaf, index, siblings\}$
    return proof
end procedure
```

4.7 Verify Merkle Proof

Algorithm 6 LeanIMT verifyProof algorithm

```
procedure VERIFYPROOF($index$)
    { root, leaf, siblings, index } ← proof                    ▷ Deconstruct the proof
    node ← leaf
    for i from 0 to siblings.length - 1 do
        isOdd ← devide index by 2 $i$ times and check if the result is odd
        if isOdd is true then                                   ▷ $node$ is a right child
            node ← $hash(siblings[i], node)$
        else                                                    ▷ It's a left node
            node ← $hash(node, siblings[i])$
        end if
    end for
    if $root$ is equal $node$ then
        return true
    else
        return false
    end if
end procedure
```

5 Implementations

5.1 TypeScript

5.2 Solidity

6 Benchmarks

7 Conslusions

This document is based on the work of [1].

References

[1] Barry Whitehat Kobi Gurkan Koh Wei Jie. “Semaphore: Zero-Knowledge Signaling on Ethereum”. In: (2020). URL: https://semaphore.pse.dev/whitepaper-v1.pdf.