LeanIMT: An optimized Incremental Merkle Tree

Privacy & Scaling Explorations

July 13, 2024

1 Abstract

Created: July 13, 2024
Updated: July 13, 2024

Contents

1 Abstract **1**

2 Introduction **4**
2.1 Motivation 4

3 Merkle Tree **4**
3.1 Incremental Merkle Tree 4
3.2 Binary Tree 4

4 LeanIMT **4**
4.1 Definition 4
4.2 Insertion 5
4.2.1 Pseudocode 7
4.2.2 Correctness 7
4.2.3 Time complexity 8
4.3 Batch Insertion 8
4.3.1 Pseudocode 10
4.3.2 Correctness 10
4.3.3 Time complexity 10
4.4 Update 12
4.4.1 Pseudocode 13
4.4.2 Correctness 13
4.4.3 Time complexity 13
4.5 Remove 14
4.5.1 Pseudocode 14
4.5.2 Correctness 14
4.5.3 Time complexity 14
4.6 Generate Merkle Proof 15
4.6.1 Correctness 17
4.6.2 Time complexity 17
4.7 Verify Merkle Proof 18
4.7.1 Correctness 20
4.7.2 Time complexity 20

5 Implementations **20**
5.1 TypeScript 20
5.2 Solidity 20

6 Benchmarks **20**

7 Conclusions **20**

2 Introduction

2.1 Motivation

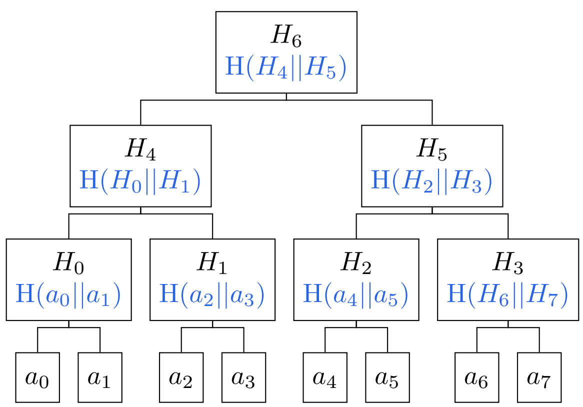
3 Merkle Tree

3.1 Incremental Merkle Tree

An Incremental 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.
The tree is always built from the leaves to the root.
The tree will always be balanced by construction.
In a LeanIMT a node is either a leaf or a parent.

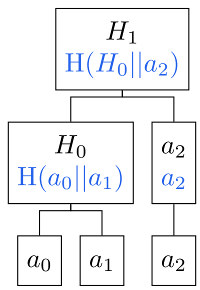
Example of a LeanIMT

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

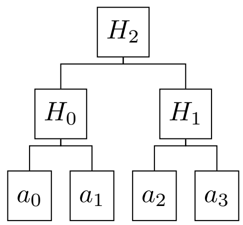
It will not be hashed, it's value will be sent to the next level.

If we add a_4.

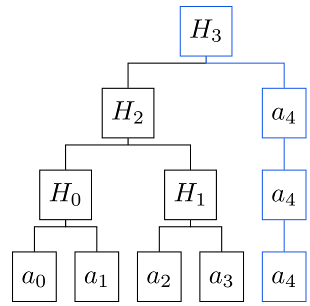
$T = (V, E)$

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

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



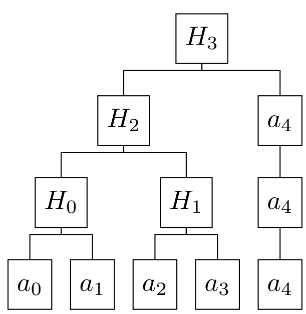
Before inserting a_4



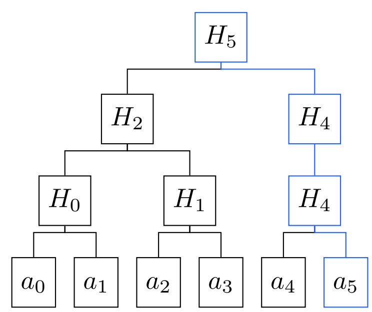
After inserting a_4

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 ← leaf
    index ← 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 ← ⌊index/2⌋    ▷ Divides the index by 2 and discards the remainder.
    end for
    nodes[depth] ← [node]    ▷ Store the new root at the top level
end procedure
```

4.2.2 Correctness

The idea is to prove that after inserting a new node to the LeanIMT, the tree keeps all the properties.
The Mathematical Induction method will be used to prove the correctness of the algorithm.
n: Number of leaves
T: LeanIMT Tree
$n = 0$
If the tree is empty, the Insert function returns a new node with the value of the leaf. This satisfies the LeanIMT rpoperties because the node does not have children. (Trivial case)
$n \Rightarrow n + 1$
Let's assume that with n nodes, the tree T is a correct LeanIMT and prove that with $n + 10$ nodes, T is still a correct LeanIMT. (Inductive step)
The new node is always added at the end of the list of nodes.
To insert a new node, there are two possible cases:
1. When the new node is a left node
2. When the new node is a right node
Case 1
If the new node is a left node, it means that there were an even number of nodes.

Then, sice it's a left node, the parents has only one child and the parent has the same value as the child which is is the new node. Then all the ancestors will be constructed following the two properties of the LeanIMT.
Since T was a correct LeanIMT with n nodes and inserting a new node that is a left node follows the properties of the LeanIMT $\Rightarrow$ the entire tree T is still a correct LeanIMT.
Case 2
If the new node is a right node, it means that there were an odd number of nodes.
Since it is a right node, the value of the parent will be the hash of the left child with the right child which is the new node. Then all the ancestors will be constructed following the two properties of the LeanIMT.
Since T was a correct LeanIMT with n nodes and inserting a new node that is a right node follows the properties of the LeanIMT $\Rightarrow$ the entire tree T is still a correct LeanIMT.
Then for the two cases 1 and 2 the Insert algorithm follows the LeanIMT properties $\Rightarrow$ the Insert algorithm is correct.

4.2.3 Time complexity

n: Number of leaves in the tree.
d: Tree depth.

Every time a new node is added, it is necessary to update or add the ancestors up to the root of the tree.

Number of operations when adding a leaf: $\lceil d \rceil$

$$
\begin{aligned}
\lceil d \rceil &\leq d + 1 \\
&\leq O(\log n) + 1 \\
&\Rightarrow \boxed{O(\log n)}
\end{aligned}
$$

$$
\begin{aligned}
d = \lceil \log(n) \rceil &\leq \log(n) + 1 \\
&\Rightarrow O(\log n)
\end{aligned}
$$

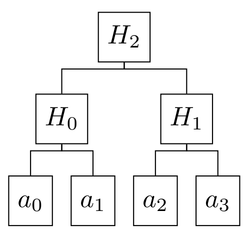
The time complexity of the *Insert* function is $\boldsymbol{O(\log n)}$.

4.3 Batch Insertion

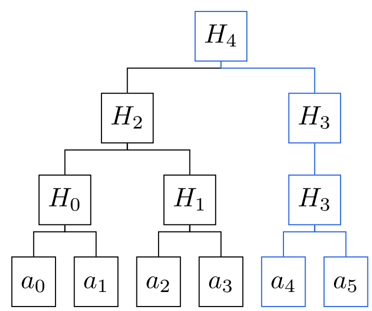
Performing the insertion in bulk rather than individually using a loop can lead to significant performance improvements because the number of hashing operations is reduced.

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 will not change.

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 $\leftarrow \lceil nodes[level].length/2 \rceil$ ▷ 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 $\leftarrow hash(leftNode, rightNode)$
            else
                parentNode $\leftarrow leftNode$
            end if
            nodes[level + 1][index] ← parentNode ▷ Add the parent node to the tree.
        end for
        startIndex $\leftarrow \lfloor startIndex/2 \rfloor$ ▷ Divide startIndex by 2 and discards the remainder.
    end for
end procedure

4.3.2 Correctness

4.3.3 Time complexity

n: Number of leaves in the tree.
d: Tree depth.
m: Number of leaves to insert.

Number of operations when inserting elements in batch:

$\lceil m \rceil + \lceil \frac{m}{2} \rceil + \lceil \frac{m}{4} \rceil + ... + \lceil \frac{m}{2^d} \rceil$

That is the same as $\sum_{k=0}^{d} \lceil \frac{m}{2^k} \rceil$

$\lceil m \rceil \leq m + 1$ then $\lceil \frac{m}{2^k} \rceil \leq \frac{m}{2^k} + 1$

$\sum_{k=0}^{d}\lceil \frac{m}{2^k}\rceil \leq \sum_{k=0}^{d}(\frac{m}{2^k}+1)$

$\leq \sum_{k=0}^{d} \frac{m}{2^k} + \sum_{k=0}^{d} 1$

$\leq 2m + O(\log(n+m))$

$\leq O(m) + O(\log(n+m))$

$\Rightarrow \boxed{O(m)}$

$\sum_{k=0}^{d} \frac{m}{2^k} = m\sum_{k=0}^{d} \frac{1}{2^k} \approx m * 2 \Rightarrow 2m$

$\sum_{k=0}^{d} \frac{1}{2^k}$ (Geometric series)

$|r| < 1$; $r = \frac{1}{2}$

$\sum_{k=0}^{\infty} a * r^n = \frac{a}{1-r} = \frac{1}{1-\frac{1}{2}}$

$= \frac{1}{2}$

$= 2$

$\sum_{k=0}^{d} 1 = \frac{a}{1-r} = d + 1$

$= O(\log(n+m)) + 1$

$\Rightarrow O(\log(n+m))$

$d = \lceil \log(n+m) \rceil \leq \log(n+m) + 1$

$\Rightarrow O(\log(n+m))$

Then the time complexity of the *InsertMany* function is $\boldsymbol{O(m)}$.

Loop Insertion vs Batch Insertion

The time complexity of the Insertion function using a loop is $O(\log(n+m))$.

Going to the root to update or add nodes requires $\log(n+m)$ number of operations.
If we go to the root to update or add nodes m times (one time per leaf to add) then we will have:

$m * \log(n+m) \Rightarrow \boxed{O(m\log(n+m))}$

⇒ Time complexity Loop Insertion is superlinear: $\boldsymbol{O(m \log(n+m))}$

⇒ Time complexity Batch Insertion (*InsertMany* function) is linear: $\boldsymbol{O(m)}$ (linear)

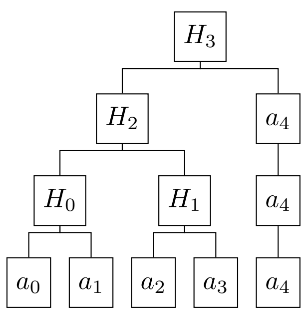
⇒ In terms of time complexity, it is more efficient to use the *InsertMany* function than the *Insert* function in a loop.

4.4 Update

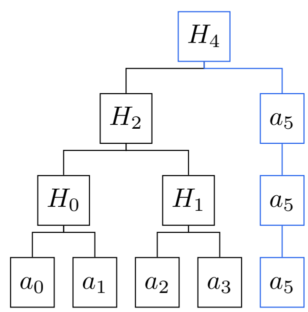
There are two cases:
1. When there is no right sibling.
2. When there is right sibling.

Case 1: There is no right sibling

Update a_4 to a_5



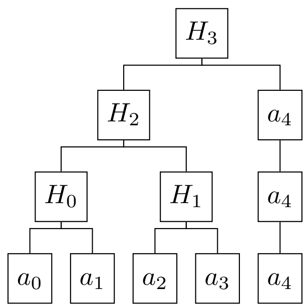
Before updating a_4



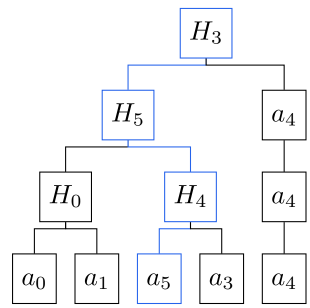
After updating a_4

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 ← 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 ← ⌊index/2⌋          ▷ Divides the index by 2 and discards the
    remainder.
    end for
    nodes[depth] ← [node]                   ▷ Store the new root at the top level
end procedure
```

4.4.2 Correctness

4.4.3 Time complexity

n: Number of leaves in the tree.
d: Tree depth.

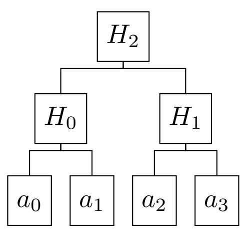
Every time a leaf is updated, it is necessary to update all the ancestors up to the root of the tree.

Number of operations when updating a leaf: $\lceil d \rceil$
This proof is the same as the proof of the time complexity of the Insert function.
The time complexity of the *Update* function is $\boldsymbol{O(\log n)}$.

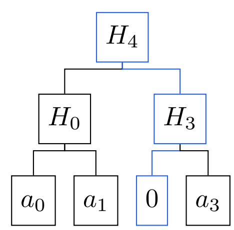
4.5 Remove

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

4.5.2 Correctness

4.5.3 Time complexity

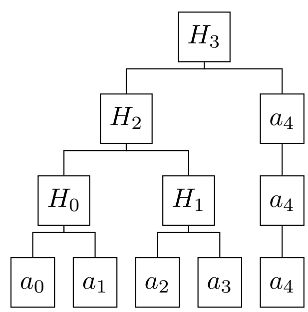
4.6 Generate Merkle Proof

There are two cases:
1. When the leaf is the last leaf and a left node.
2. Other cases

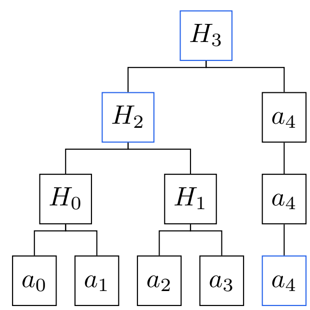
d: Depth of the tree
We will always see one of these two cases. When you want to generate the proof for the case 1, there will always be one node in the siblings list, for the case 2 there will always be $\lceil d \rceil$ number of nodes.

Case 1: The leaf is the last leaf and a left node

If we want to generate a proof for the node a_4.



LeanIMT to generate a proof for a_4

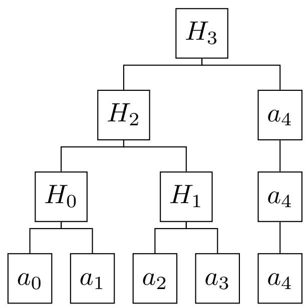


Nodes used to generate a proof for a_4

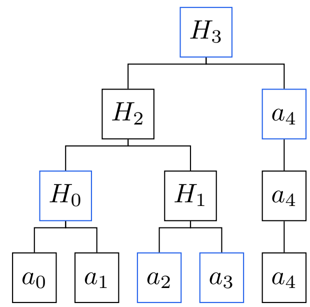
path: [1]
Merkle Proof: {
root: H_3
leaf: a_4
index: 1
siblings: [H_2]
}

Case 2: Other cases

If we want to generate a proof for the node a_3.



LeanIMT to generate a proof for a_3



Nodes used to generate a proof for a_3

path: [1, 1, 0]
Merkle Proof: {
root: H_3
leaf: a_3
index: 3
siblings: [a_2, H_0, a_2]
}

Algorithm 5 LeanIMT generateProof algorithm

```
procedure GENERATEPROOF(index)
    siblings ← empty list ▷ List to store the nodes necessary to rebuild the root.
    path ← empty list ▷ List of 0s or 1s to help rebuild the root. 0 if the current node is a left node and the sibling is a right node and 1 otherwise.
    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 ← ⌊index/2⌋ ▷ 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
```

4.6.1 Correctness

4.6.2 Time complexity

n: Number of leaves in the tree.
d: Tree depth.

To generate a Merkle Proof it is necessary to visit all the ancestors of the leaf up to the root of the tree.

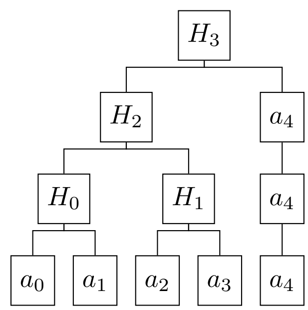
Number of operations to generate a Merkle Proof: $\lceil d \rceil$
This proof is the same as the proof of the time complexity of the Insert function.
The time complexity of the *generateProof* function is $\boldsymbol{O(\log n)}$.

4.7 Verify Merkle Proof

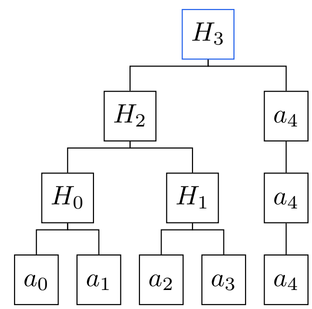
There are two cases:
1. When the leaf is the last leaf and a left node.
2. Other cases

Case 1: The leaf is the last leaf and a left node

If we want to verify a proof for the node a_4.



LeanIMT to verify a proof for a_4

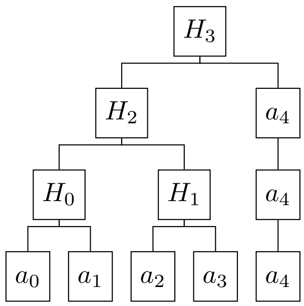


Nodes rebuilt to verify a proof for a_4

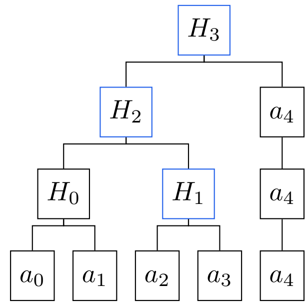
path: [1]
Merkle Proof: {
root: H_3
leaf: a_4
index: 1
siblings: [H_2]
}

Case 2: Other cases

If we want to verify a proof for the node a_3.



LeanIMT to verify a proof for a_3



Nodes rebuilt to verify a proof for a_3

path: [1, 1, 0]
Merkle Proof: {
root: H_3
leaf: a_3
index: 3
siblings: [a_2, H_0, a_2]
}

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

4.7.1 Correctness

4.7.2 Time complexity

n: Number of leaves in the tree.
d: Tree depth.

To verify a Merkle Proof it is necessary to visit (rebuild) all the ancestors of the leaf up to the root of the tree.

Number of operations to verify a Merkle Proof: $\lceil d \rceil$
This proof is the same as the proof of the time complexity of the Insert function.
The time complexity of the *verifyProof* function is $\boldsymbol{O(\log n)}$.

5 Implementations

TODO: Explain TypeScript and Solidity implementations.

5.1 TypeScript

5.2 Solidity

6 Benchmarks

7 Conclusions

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.

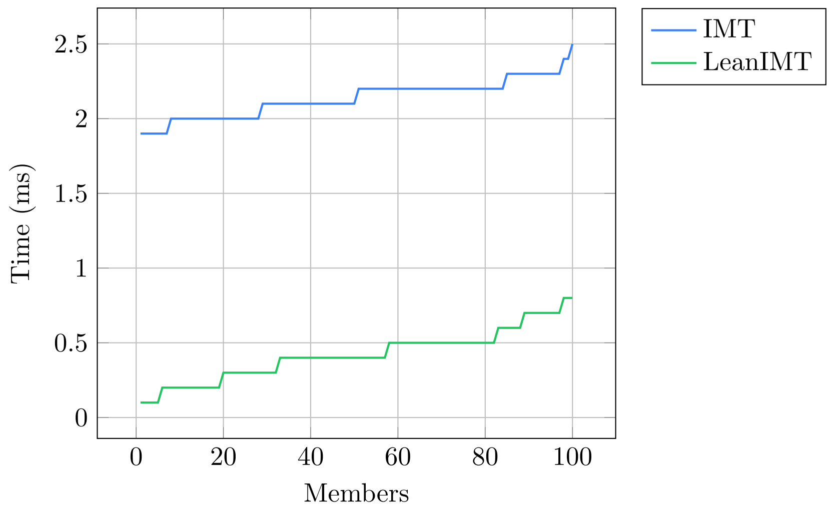


Figure 1: Insert function IMT vs LeanIMT

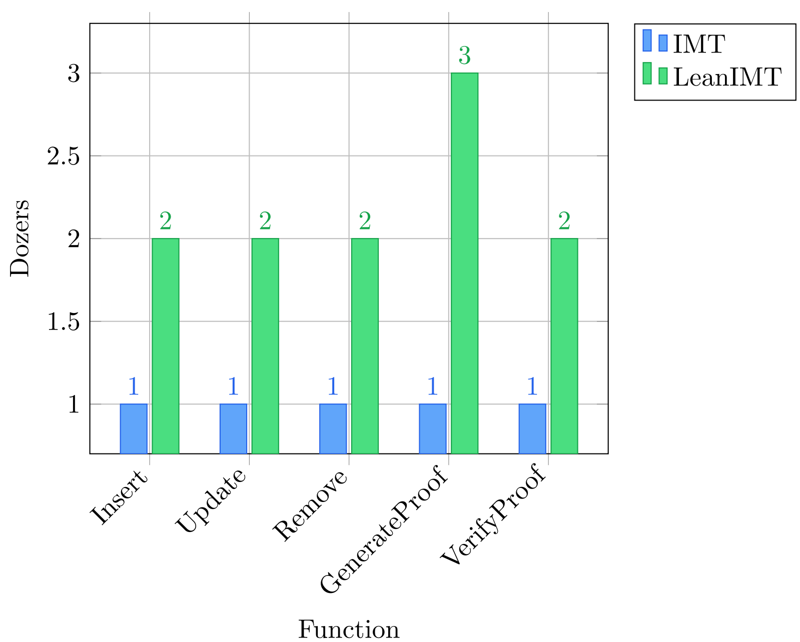


Figure 2: Insert function IMT vs LeanIMT