Cartesian Merkle Tree

Artem Chystiakov, Oleh Komendant, Kyrylo Riabov

April 2025

Abstract

This paper introduces the Cartesian Merkle Tree, a deterministic data structure that combines the properties of a Binary Search Tree, a Heap, and a Merkle tree. The Cartesian Merkle Tree supports insertions, updates, and removals of elements in $O(\log n)$ time, requires n space, and enables membership and non-membership proofs via Merkle-based authentication paths. This structure is particularly suitable for zero-knowledge applications, blockchain systems, and other protocols that require efficient and verifiable data structures.

1 Introduction

Cartesian Merkle Tree (CMT) research is motivated by the increasing demand for efficient and secure data structures in cryptographic, blockchain, and zero-knowledge (ZK) systems. Traditional Merkle trees, such as Sparse Merkle Trees (SMT)[ide24], are widely used but have limitations in terms of balance and efficiency.

CMTs integrate binary search tree properties for key-based organization and heap properties for priority balancing, data storage optimization, elements retrieval, and proof generation. To ensure determinism, the priority value for an element is derived from its key using a predefined algorithm, such as a hash function.

CMTs offer an efficient alternative to SMTs, reducing memory usage while preserving the key properties of the latter. One of the key features of CMT is that it stores useful data in every node, unlike SMTs which only store it in the leaves. The time complexity of the operations is still $O(\log n)$, with the only trade-off being a Merkle proof size at worst two times larger than SMTs.

2 Background

Merkle trees are fundamental to many cryptographic protocols, enabling efficient and secure proofs of data inclusion. They are widely used across various systems, ranging from maintaining validator sets in Ethereum to storing the state of zero-knowledge layer-2 rollups. The primary strength of Merkle trees lies in their ability to prove the inclusion of data in a highly compact and efficient manner.

Although the standard Merkle tree structure is binary, there are variations, such as the Merkle-Patricia tree, which utilizes sixteen child nodes per branch, optimizing performance in specific contexts. Despite these advances, Merkle trees still face challenges related to storage and operational complexity. Typically, they require $O(\log n)$ operations and $2n$ storage for binary trees.

While vanilla Merkle trees are often impractical for on-chain usage, they are still frequently utilized in off-chain whitelists and similar applications. However, two significant Merkle tree modifications have emerged that unlock the full potential of the data structure: Incremental Merkle Trees (IMT) and Sparse Merkle Trees (SMT).

2.1 Incremental Merkle Tree

IMT is a push-only data structure that can be used on-chain to build the tree, but requires an off-chain service to generate inclusion proofs. IMT is currently used by TornadoCash for anonymizing depositors, by Semaphore to store group membership commitments, and by the BeaconChain deposit smart contract to manage the list of Ethereum validators.

Unlike standard Merkle trees, IMTs do not store individual elements. Instead, they are merely used to "build" the root. IMTs require $\log n$ storage, with inclusion proofs also having $O(\log n)$ complexity. Importantly, IMTs cannot be used independently without an off-chain service, as the entire tree must be reconstructed to generate an inclusion proof.

2.2 Sparse Merkle Tree

Until recently, SMTs were regarded as one of the most efficient data structures, particularly in the context of blockchain technologies and ZK systems. For instance, SMT is used by Scroll to maintain its ZK rollup state, by Rarimo to store ZK-provable national passport-based identity data, and by iden3 to manage custom-issued on-chain identities.

SMT is a particularly fascinating data structure. Unlike IMT, SMT does not require any off-chain services. It is deterministic, meaning the structure of the tree remains identical for a given set of elements, regardless of their insertion order. The position of an element in the tree is determined by its bitwise prefix: if a 0 is encountered, the left child is selected; if a 1 is encountered, the right child is chosen. This results in the tree size being constrained by the bitwise length of this prefix. However, reaching depths of 97 or more is currently infeasible due to limitations within the EVM stack. In practice, the size of the tree *without collisions* cannot exceed 2^{50}, or approximately 1.12×10^{15} (one quadrillion) elements, according to the Birthday Problem[Wik25a].

3 Data Structure Description

A Cartesian Merkle Tree can be seen as a standard Cartesian tree or Treap with the additional data Merkleization property. Each element in CMT corresponds to a point on a two-dimensional plane, with the key `k` representing the X-coordinate and the priority `p` representing the Y-coordinate. In traditional Cartesian trees [Wik25b], the value of `p` is typically chosen at random, which contributes to a more balanced tree structure. However, if `p` is deterministically derived from `k`, then the same key will always produce the same point on the plane. As a result, the structure of the CMT becomes deterministic.

Let $\texttt{e} = \texttt{k}$ be a new entry in the tree `T`, where `k` is the key of the entry. The entry `e` may also optionally have a field `v`, the value. The node where this data element `e` is stored is determined based on the information contained in `e`. Let `H` be a cryptographically secure hash function that returns the hash result for an arbitrary number of values. Let $\texttt{p} = P_H(\texttt{e})$ be the priority of the element `e`, where P_H can be any deterministic algorithm that transforms the values of `e` into a number.

Let $\texttt{node} = (\texttt{k}, \texttt{p}, \texttt{mh})$ be any node in the tree `T`, where `mh` is the Merkle Hash value of the node, calculated as follows:

$$mh = H(\texttt{entry} \parallel \texttt{leftChildMH} \parallel \texttt{rightChildMH})$$

Here, `leftChildMH` and `rightChildMH` are the Merkle hash values of the node's children, sorted in ascending order, and `entry` is the node's useful payload. If a child is absent, its hash value is considered to be 0.

The node where `e` should be stored is determined by the following rules:

$$\texttt{leftChild.k} \leq \texttt{e.k} \leq \texttt{rightChild.k}$$

$$\texttt{leftChild.p} \leq \texttt{e.p} \quad \text{and} \quad \texttt{rightChild.p} \leq \texttt{e.p}$$

The first rule ensures the binary search tree property, while the second rule maintains the min-heap property.

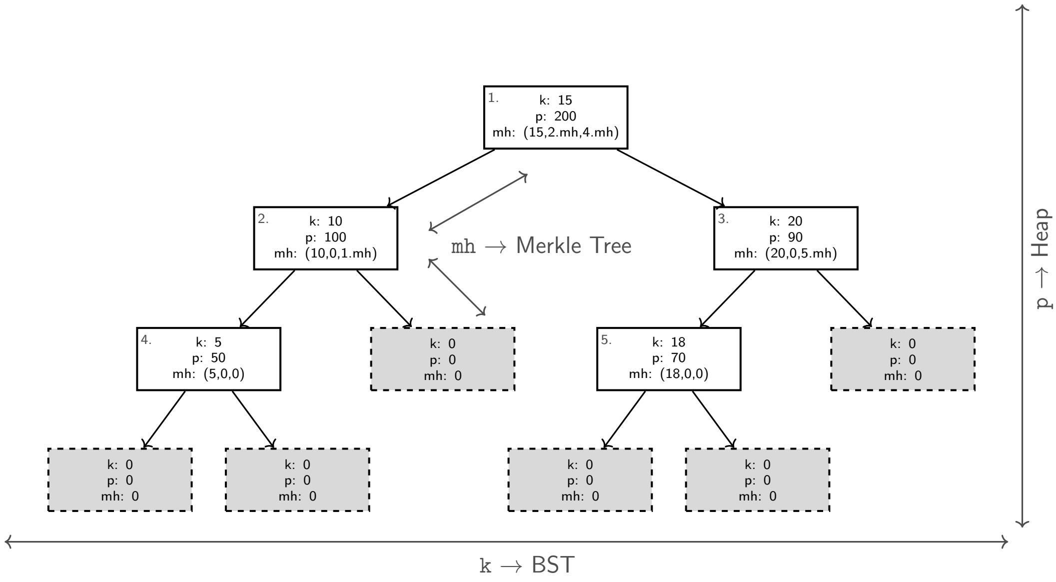

Figure 1: Example of CMT

Algorithm 1: CMT Utils Functions

```
Function CalculateMH(key, leftChildMH, rightChildMH):
    /* Sort leftChildMH, rightChildMH in ascending order */
    if leftChildMH < rightChildMH then
        return H(key||leftChildMH||rightChildMH);
    else
        return H(key||rightChildMH||leftChildMH);

Function RightRotate(node):
    /* Save pointers to the left child and its right child */
    currentLeftChild ← node.leftChild;
    newLeftChild ← currentLeftChild.rightChild;
    /* Perform the right rotation by updating the pointers */
    currentLeftChild.rightChild ← node;
    node.leftChild ← newLeftChild;
    /* Update the mh value of the node after rotation */
    node.mh ← CalculateMH(node.e.k, leftChildMH, rightChildMH);

Function LeftRotate(node):
    /* Save pointers to the right child and its left child */
    currentRightChild ← node.rightChild;
    newRightChild ← currentRightChild.leftChild;
    /* Perform the left rotation by updating the pointers */
    currentRightChild.leftChild ← node;
    node.rightChild ← newRightChild;
    /* Update the mh value of the node after rotation */
    node.mh ← CalculateMH(node.e.k, leftChildMH, rightChildMH);
```

3.1 Insertion

When inserting an entry `e`, the corresponding node `n` in the tree `T` is determined by traversing downward from the root while maintaining the BST property. Once inserted, the structure is recursively adjusted upward to the root to restore the min-heap property using left or right rotations. At each modification of any node, the `mh` value is also recomputed.

Algorithm 2: Insertion of an Element into the CMT

```
Input: Element e = k to be inserted
Output: Updated tree T
Function Insert(T, e):
    Find the appropriate position for e in T based on the BST property;
    Create a new node n to insert e, and set:
    begin
        Set n.e ← e;
        Set n.p ← P_H(e);
        Set n.mh ← CalculateMH(e.k, leftChildMH, rightChildMH);
    /* Restore min-heap property by rotating upwards */
    Node ← n;
    while Node is not root and Parent.Priority < Node.Priority do
        /* Determine which child the node is to its parent and perform needed
           rotation */
        if Parent.leftChild = Node then
            RightRotate(Parent);
        else if Parent.rightChild = Node then
            LeftRotate(Parent);
        Node ← Parent(Node);
        /* Update mh value of the node after rotation */
        Node.mh ← CalculateMH(Node.e.k, leftChildMH, rightChildMH);
```

Example

In this example, consider the insertion of the element `e` where `e.k` $= 13$ and $P_H(\mathtt{e}) = 250$ into the tree shown in Figure 1. After the element insertion, the tree will have the following structure:

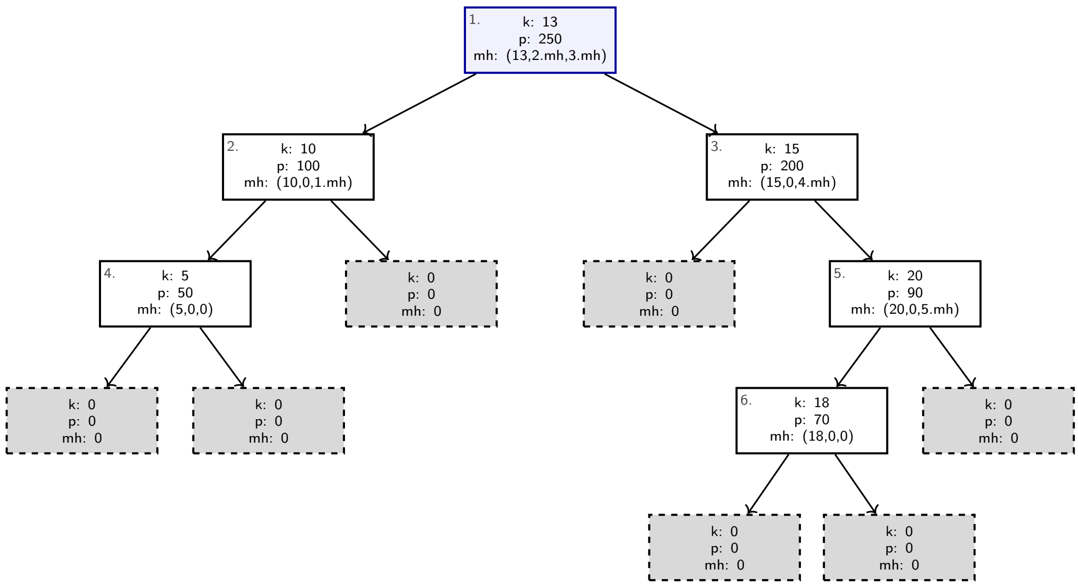

Figure 2: CMT after insertion

3.2 Removal

To remove an entry `e`, first determine whether there exists a node `n` in the tree `T` that corresponds to `e`. If such a node is found, assign $\texttt{n.p} = -\infty$, ensuring that during the heap property restoration via rotations, `n` will be moved to a leaf position. Once the node becomes a leaf, it can be easily removed. After removal, recursively traverse upward and update the `mh` values.

Algorithm 3: Removal of an Element from the CMT

```
Input: Element e = k to be removed
Output: Updated tree T
Function Remove(T, e):
    Find the node n corresponding to e in T;
    if n does not exist then
        Throw error: "Element not found";
    /* Mark node for removal by setting its priority to −∞ */
    n.p ← −∞;
    /* Restore min-heap property via rotations */
    Node ← n;
    while Node is not a leaf do
        /* Determine which child has higher priority and perform rotation */
        if Node.leftChild.p > Node.rightChild.p then
            RightRotate(Node);
        else
            LeftRotate(Node);
        Node ← new position after rotation;
    /* Remove the node */
    Remove Node from T;
    /* Update mh values while traversing upwards */
    Parent ← Parent(Node);
    while Parent is not null do
        Parent.mh ←
         CalculateMH(Parent.e.k, Parent.leftChildMH, Parent.rightChildMH);
        Parent ← Parent(Parent);
```

Example

Consider removing of an element `e` with a key $\texttt{e.k} = 15$ from the tree shown in Figure 2. After removing, the tree structure will be modified as follows:

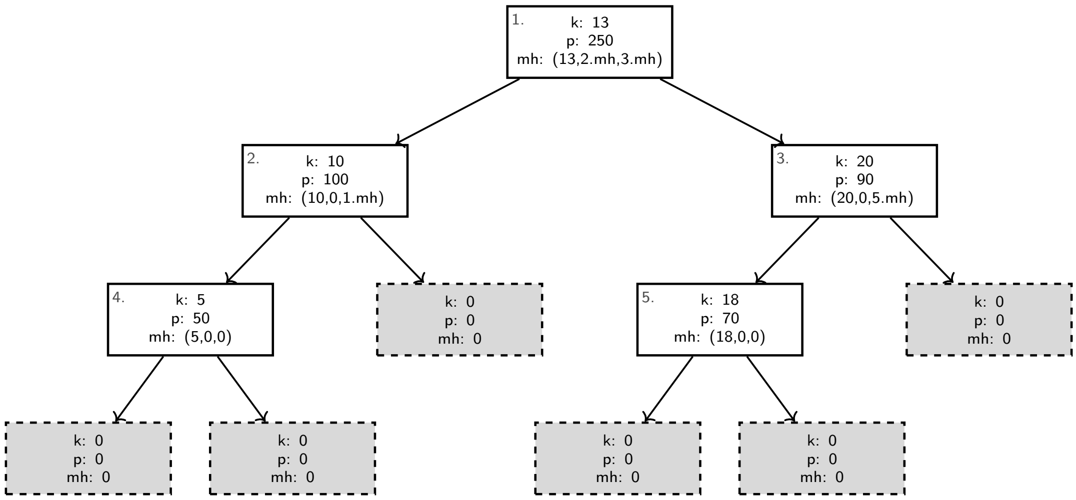

Figure 3: CMT after removal

3.3 CMT Proof

Let `CMT proof` be a proof of membership of an element `e` in a tree `T`, represented as $\texttt{proof} = [\texttt{prefix}, \texttt{suffix}]$, where:

- `prefix` is an ordered list of Merkle path nodes, each containing pairs of values $(\texttt{n.e.k}, \texttt{n.mh})$ for each node `n`;
- `suffix` consists of `e.leftChildMH` and `e.rightChildMH`, representing the subtree structure of `e`;
- `existence` is a boolean flag indicating whether `e` exists in the tree;
- `nonExistenceKey` is used when `e` does not exist in the tree, and helps verify that `e` is absent.

The initial value of `acc` is computed as:

$$\texttt{acc} = H((\texttt{existence}?\texttt{e.k} : \texttt{nonExistenceKey}) \parallel \texttt{proof.suffix[0]} \parallel \texttt{proof.suffix[1]})$$

ensuring that $\texttt{proof.suffix[0]} < \texttt{proof.suffix[1]}$.

Then, `acc` is iteratively updated using values from `prefix`:

$$\begin{cases} \texttt{acc} = H(\texttt{n.e.k} \parallel \texttt{n.mh} \parallel \texttt{acc}), & \text{if } \texttt{n.mh} < \texttt{acc}, \\ \texttt{acc} = H(\texttt{n.e.k} \parallel \texttt{acc} \parallel \texttt{n.mh}), & \text{otherwise.} \end{cases}$$

The proof is considered valid if the final value of `acc` matches the root of `T`.

Algorithm 4: CMT Proof Generation

```
Input: Element e = k to be proven in tree T
Output: Proof proof = [prefix, suffix, existence, nonExistenceKey]
Function GenerateProof(T, e):
    Initialize empty lists: prefix ← [], suffix ← [];
    Initialize bool variable existence ← true;
    Initialize variable currentNode ← null;
    /* Get the appropriate node for the entry e */
    if e does not exist in the tree T then
        currentNode ← node with appropriate key for non-existence proof;
        existence ← false;
        nonExistenceKey ← currentNode.e.k;
    else
        currentNode ← node in T where n.e = e;
    /* Set suffix as the hash values of currentNode's children */
    suffix ← [n.leftChildMH, n.rightChildMH];
    /* Construct prefix by traversing the path to the root */
    while currentNode is not root do
        parent ← Parent(currentNode);
        Append (parent.e.k, parent.mh) to prefix;
        currentNode ← parent;
    return [prefix, suffix, existence, nonExistenceKey];
```

Algorithm 5: Verification of CMT Proof

```
Input: Proof proof = [prefix, suffix, existence, nonExistenceKey], Element e, Root hash
       root
Output: true if e is in the tree or e is not in the tree and existence is false, false otherwise
Function VerifyProof(proof, e, root):
    /* Initialize acc with the element's Merkle hash */
    if proof.existence then
        acc ← CalculateMH(e.k, proof.suffix[0], proof.suffix[1]);
    else
        acc ← CalculateMH(nonExistenceKey, proof.suffix[0], proof.suffix[1]);
    /* Iteratively compute the hash up the Merkle path */
    foreach (k, mh) in proof.prefix do
        acc ← CalculateMH(k, acc, mh);
    /* Check if computed hash matches the root */
    return acc = root;
```

Example

Consider the generation and verification of a CMT proof for an entry e, where e.k $= 18$, in a tree depicted in Figure 4. To make the example clearer, we replace the mh in all nodes with particular numbers.

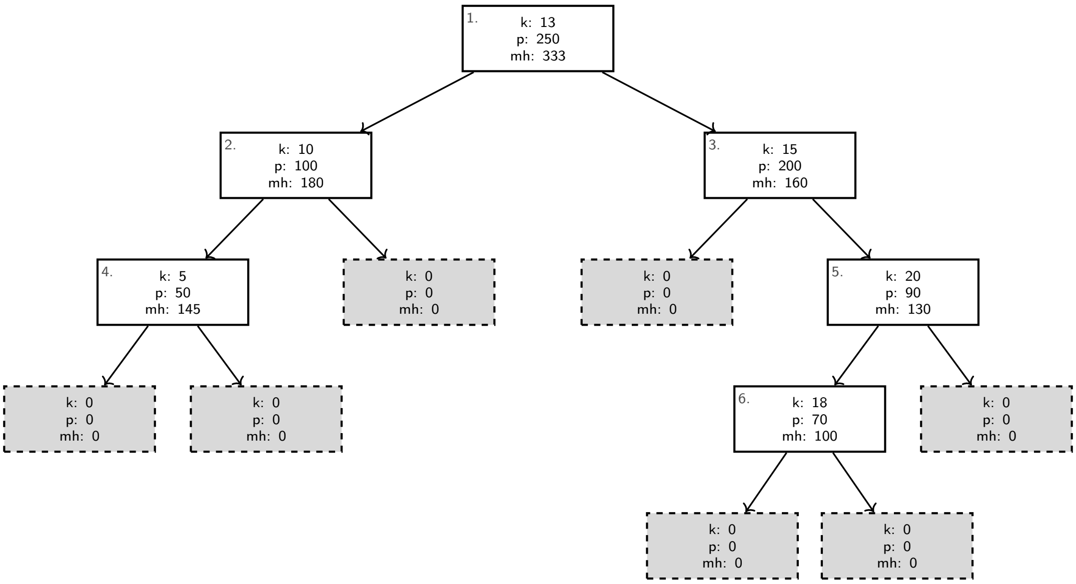

Figure 4: CMT proof example

Algorithm 6: CMT Inclusion Proof Example

Input:

1. `proof`:
   - `prefix`: $[13, 180, 15, 0, 20, 0]$
   - `suffix`: $[0, 0]$
   - `existence`: true
   - `nonExistenceKey`: 0
2. `rootNodeMH`: 333

```
/* Initialize acc with the element's Merkle hash                              */
acc = H(e.k || proof.suffix[0] || proof.suffix[1]); /* H(18 || 0 || 0) = 100  */

/* Compute the final acc value using the entries from the prefix.             */
acc = H(proof.prefix[4] || proof.prefix[5] || acc); /* H(20 || 0 || 100) = 130 */
acc = H(proof.prefix[2] || proof.prefix[3] || acc); /* H(15 || 0 || 130) = 160 */
acc = H(proof.prefix[0] || acc || proof.prefix[1]); /* H(13 || 160 || 180) = 333 */

/* Compare acc with rootNodeMH                                                */
acc == root.mh; /* Result is true                                             */
```

Consider the case where an entry `e` with key `e.k` $= 25$ is not present in the tree depicted in Figure 4. In this scenario, the proof will be structured as follows:

Algorithm 7: CMT Exclusion Proof Example

Input:

1. `proof`:
   - `prefix`: $[13, 180, 15, 0]$
   - `suffix`: $[100, 0]$
   - `existence`: false
   - `nonExistenceKey`: 20
2. `rootNodeMH`: 333

```
/* Initialize acc with the element's Merkle hash */
acc = H(proof.nonExistenceKey || proof.suffix[0] || proof.suffix[1]);
    /* H(20 || 0 || 100) = 130 */

/* Compute the final acc value using the entries from the prefix. */
acc = H(proof.prefix[2] || proof.prefix[3] || acc); /* H(20 || 0 || 130) = 160 */
acc = H(proof.prefix[0] || acc || proof.prefix[1]); /* H(13 || 160 || 180) = 333 */

/* Compare acc with rootNodeMH */
acc == root.mh; /* Result is true */
```

4 Reference Implementation

To provide a practical implementation of the Cartesian Merkle Tree and its proof verification, we reference two existing implementations in Solidity and Circom:

- The Solidity implementation, available in the `solidity-lib` repository[Sol25b], provides smart contract functionalities for CMT construction and proof generation.
- The Circom implementation, available in the `circom-lib` repository[Sol25a], offers a zk-SNARK-friendly circuit for verifying CMT proofs within zero-knowledge proofs.

These implementations are fully compatible: proofs generated by the Solidity implementation can be used to generate and verify zero-knowledge proofs in the Circom implementation.

5 Benchmarks

This section presents the benchmarking results for the `Insert` and `Remove` functions in the Solidity CMT implementation [Sol25b] using the `Keccak256` and `Poseidon` hash functions. A comparative analysis of EVM gas costs was performed for each function using different datasets. The tests were conducted with 100, 1000, 5000, and 10000 elements to show how data size impacts performance. The EVM gas costs can be considered as normalized computation units, therefore, if the operation takes more gas, it is more computationally intensive.

In order to ensure a fair comparison of the CMT and SMT structures benchmarks, the Solidity version of the SMT[Sol25c] was tested using the same methods as the CMT. To maintain consistency, the `value` field was removed from the SMT `Node` structure so that they occupy the same number of storage slots.

5.1 Insert Operation

The `Insert` function gas benchmarks were obtained by inserting 100, 1000, 5000, and 10000 random elements into the trees.

The CMT results are presented in Table 1 for `Keccak256` and Table 2 for `Poseidon`. The SMT results are presented in Table 3 and Table 4, respectively.

Iterations	Min Gas	Avg Gas	Max Gas
100	97,593	187,682	301,113
1,000	97,605	254,332	421,359
5,000	97,605	286,195	502,851
10,000	97,593	303,871	552,723

Table 1: CMT gas usage with **Keccak256**

Iterations	Min Gas	Avg Gas	Max Gas
100	148,017	599,943	1,246,860
1,000	148,017	883,129	2,068,385
5,000	148,017	1,090,143	2,924,415
10,000	148,017	1,140,019	2,773,845

Table 2: CMT gas usage with **Poseidon**

Comparing with SMT:

Iterations	Min Gas	Avg Gas	Max Gas
100	102,125	248,063	667,958
1,000	102,137	294,404	871,190
5,000	102,125	325,785	1,306,160
10,000	102,125	339,509	877,732

Table 3: SMT gas usage with **Keccak256**

Iterations	Min Gas	Avg Gas	Max Gas
100	136,354	471,704	903,027
1,000	136,366	620,547	1,522,135
5,000	136,366	723,077	2,005,911
10,000	136,366	765,205	2,155,222

Table 4: SMT gas usage with **Poseidon**

5.2 Remove Operation

The `Remove` function gas usage was calculated by removing all inserted elements from the trees of sizes 100, 1000, 5000, and 10000 nodes.

The CMT results are presented in Table 5 for `Keccak256` and Table 6 for `Poseidon`. The SMT results are presented in Table 7 and Table 8, respectively.

Iterations	Min Gas	Avg Gas	Max Gas
100	41,917	129,109	259,680
1,000	41,917	178,084	363,816
5,000	41,917	226,253	475,639
10,000	41,917	244,545	522,075

Table 5: CMT gas usage with **Keccak256**

Iterations	Min Gas	Avg Gas	Max Gas
100	92,329	430,727	1,071,617
1,000	92,329	757,013	2,208,417
5,000	92,341	889,249	2,143,368
10,000	92,341	982,465	2,437,679

Table 6: CMT gas usage with **Poseidon**

Comparing with SMT:

Iterations	Min Gas	Avg Gas	Max Gas
100	35,029	131,847	224,916
1,000	35,039	184,186	310,235
5,000	35,039	221,294	443,089
10,000	35,029	237,191	374,284

Table 7: SMT gas usage with **Keccak256**

Iterations	Min Gas	Avg Gas	Max Gas
100	35,029	284,457	466,242
1,000	35,039	436,451	693,632
5,000	35,029	544,353	902,871
10,000	35,029	590,051	864,656

Table 8: SMT gas usage with **Poseidon**

References

[ide24] iden3. *Sparse Merkle Tree*. https://docs.iden3.io/publications/pdfs/Merkle-Tree.pdf. Accessed: 2025-01-08. 2024.

[Sol25a] DL Solarity. *Cartesian Merkle Tree Circom Circuit*. https://github.com/dl-solarity/circom-lib/blob/8f828692cb8a5211abefc2d82326b8c68fce4e1f/circuits/data-structures/CartesianMerkleTree.circom. Accessed: 2025-04-01. 2025.

[Sol25b] DL Solarity. *Cartesian Merkle Tree Solidity Implementation*. https://github.com/dl-solarity/solidity-lib/blob/bf9c4778f442810323d7038ade30acb5d273bd0d/contracts/libs/data-structures/CartesianMerkleTree.sol. Accessed: 2025-04-01. 2025.

[Sol25c] DL Solarity. *Sparse Merkle Tree Solidity Implementation*. https://github.com/dl-solarity/solidity-lib/blob/ee0ab79b70967ac6841aff9f71d2701e3f31c91c/contracts/libs/data-structures/SparseMerkleTree.sol. Accessed: 2025-04-01. 2025.

[Wik25a] Wikipedia contributors. *Birthday problem — Wikipedia, The Free Encyclopedia*. Accessed: 2025-04-01. 2025. URL: https://en.wikipedia.org/wiki/Birthday_problem.

[Wik25b] Wikipedia contributors. *Treap — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Treap. Accessed: 2025-04-14. 2025.