polygon zkEVM

Knowledge Layer

Architecture

An Efficient Append-only SMT

v.1.0

May 30, 2024

1 Introduction and Motivation

This section entails the design of an **append-only sparse Merkle tree** which is specifically designed to optimize space usage thanks to its append-only characteristic. This kind of trees are designed for scenarios where the tree represents an ordered list of data elements which can only be modified by appending new ones. With each new element appended, the root changes and successive roots can be computed with minimal persistent data. This structure will be managed by a smart contract on-chain, responsible for updating the roots of this tree. Rather than storing the whole the tree, only two elements are stored:

- An array of the size of the tree depth, denoted to as `branch`.
- The last appended element's index, denoted to as `lastElementIndex`.

Importantly, the depth of the tree, representing its maximum capacity, is known beforehand. In our case, we are using binary trees with 32 levels, enabling to accommodate 2^{32} elements. This drops the necessity to use markers to distinguish between branches and leaves. Furthermore, note that the tree is inherently balanced.

By convention, for our append-only tree we are going to use 0s as default value for empty leaves. Hence, when the list represented using the incremental Merkle tree is empty, the following holds:

$$\begin{aligned} h_i &= 0, \\ h_{j,k} &= h(0,0) =: h^{(00)}, \\ h_{m,n,l} &= h(h^{(00)}, h^{(00)}) =: h^{(0000)}, \\ &\dots \end{aligned}$$

Here, h denotes the employed hash function, and the comma notation $(,)$ signifies juxtaposition. Observe that, in this scenario, each of the hashes stored **do only depend on the level being stored**. Hence, we just need to compute a different hash value per level.

2 A Toy Example

For sake of simplicity, lets consider as a toy example a small incremental Merkle tree of a maximum capacity of 8 leaves (hence, the having maximum depth of $d = \log_2(8) = 3$). Before adding elements into the list, we are in the 0-tree situation described before (Figure 1).

In this case, `lastElemIndex` is set to 0 and the 3-element array `branch` is also set to zero `branch` = (0, 0, 0). Note that we can compute the root of the empty tree from the zero hash values. Note also that the hash of zero nodes is uniquely determined by the height of the subtree under the node.

$$\begin{aligned} \texttt{root} &= h^{(00000000)}, \\ h^{(00000000)} &:= h(h^{(0000)}, h^{(0000)}), \\ h^{(0000)} &:= h(h^{(00)}, h^{(00)}), \\ h^{(00)} &:= h(0,0). \end{aligned}$$

Suppose we want to append an element x_0 into the list, positioning it in the first position, as shown in Figure 2.

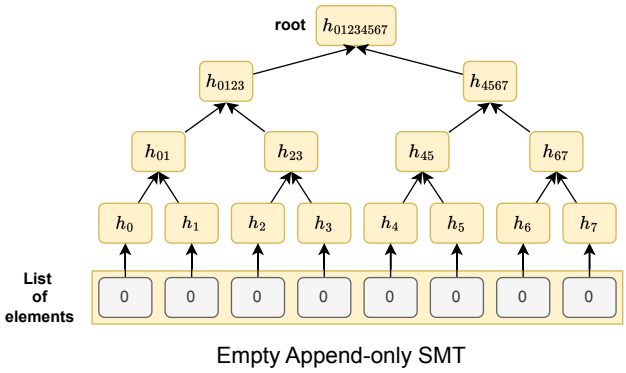

Figure 1: Illustration of the initial state (0-tree) representing a 8-sized list of uninitialized values, i.e., each containing a 0 by default.

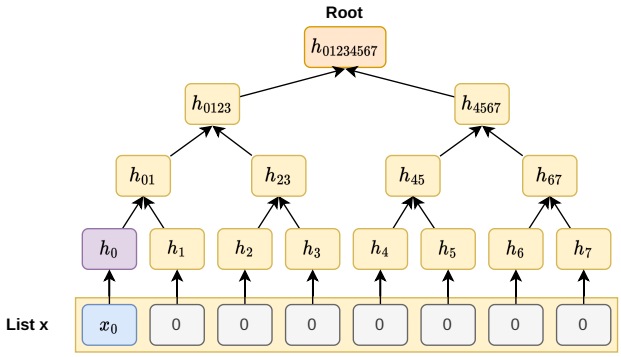

Figure 2: Illustration of the tree state after appending x_0 to the list.

Note that, in order to compute the root of the current tree, we only need the value x_0.

$$\begin{aligned}
\texttt{root} &= h_{01234567}, \\
h_{01234567} &= h(h_{0123}, h^{(0000)}), \\
h_{0123} &= h(h_{01}, h^{(00)}), \\
h_{01} &= h(h_0, 0), \\
h_0 &= h(x_0).
\end{aligned}$$

However, when appending another element x_1 we will not have x_0 in possession, but we will need it in order to get h_0 and use it to compute the tree's root (See Figure 3). Henceforth, we will use the first element of the branch array in order to store h_0. We then set the branch to $\texttt{branch} = (h_0, 0, 0)$.

Let us now add another element x_1 into the tree (See Figure 3):

In this case, we update `lastElemIndex` to 2. As said before, in order to compute the root of the current tree, we only need h_0, which is already stored in `branch`, since we can

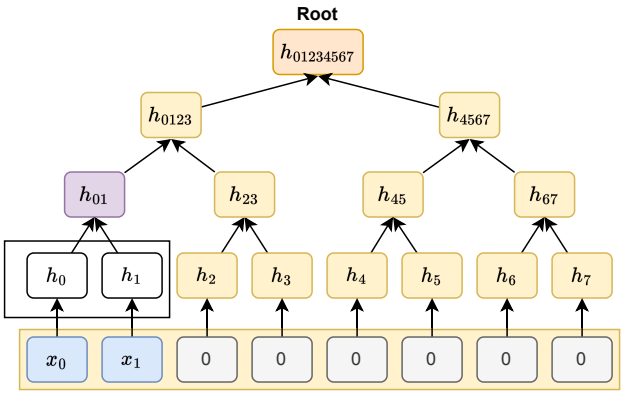

Figure 3: Illustration of the tree state after appending x_1 to the list. From now on h_0 and h_1 will not be needed anymore individually since h_{01} actually integrate both.

compute h_1 directly hashing x_1.

$$\begin{aligned} \texttt{root} &= h_{01234567}, \\ h_{01234567} &= h(h_{0123}, h^{(0000)}), \\ h_{0123} &= h(h_{01}, h^{(00)}), \\ h_{01} &= h(h_0, h(x_1)). \end{aligned}$$

Observe that, as before, we need to update the branch array in order to prepare it for the next addition to the list. In this case, we will need both h_0 and h_{01} (See Figure 4) in order to compute the next tree's root. Hence, we update branch to $\texttt{branch} = (h_0, h_{01}, 0)$. Its crucial to note that, from now on, h_0 and h_1 are not needed any more for updating the root since they are integrated in h_{01}, as we can see in Figure 3.

Let us now add another element x_2 into the tree (See Figure 4).

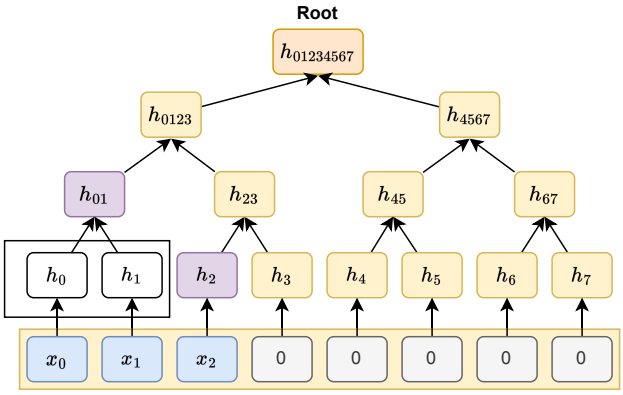

Figure 4: Illustration of the tree state after appending x_2 to the list.

As before, we update `lastElemIndex` to 3. Note that, as said before, in order to

compute the root of the current tree, we only need both h_2 and h_{01}.

$$\begin{aligned}
\texttt{root} &= h_{01234567},\\
h_{01234567} &= h(h_{0123}, h^{(0000)}),\\
h_{0123} &= h(h_{01}, h_{23}),\\
h_{23} &= h(h_2, 0),\\
h_2 &= h(x_2).
\end{aligned}$$

At this point, we need to update the `branch` array in order to prepare it for the next addition. In this case, we will need h_2 (since h_3 can be computed from x_3) and h_{01}. Hence we update the first element of the array from h_0 to h_2, obtaining $\texttt{branch} = (h_2, h_{01}, 0)$.

Let us now add another element x_3 into the tree (See Figure 5):

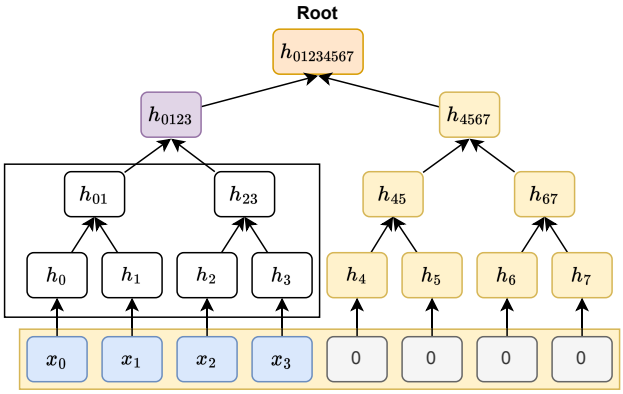

Figure 5: Illustration of the tree state after appending x_3 to the list.

As before, we update `lastElemIndex` to 4. Note that, from the elements in `branch` we can compute the `root`:

$$\begin{aligned}
\texttt{root} &= h_{01234567},\\
h_{01234567} &= h(h_{0123}, h^{(0000)}),\\
h_{0123} &= h(h_{01}, h_{23}),\\
h_{23} &= h(h_2, h_3),\\
h_3 &= h(x_3).
\end{aligned}$$

At this point and similarly as before, $h_0, h_1, h_2, h_3, h_{01}$ and h_{23} are not needed any more for updating the root since they are integrated in h_{0123} and this will not change anymore, as shown in Figure 5. Hence, we should store it in `branch`. We will use the third element in order to do it: $\texttt{branch} = (h_2, h_{01}, h_{0123})$.

Let us now add another element x_4 into the tree (Figure 6):

As before, we update `lastElemIndex` to 5. Note that, from the elements in $\texttt{branch} = (h_2, h_{01}, h_{0123})$ we can compute the `root`:

$$\begin{aligned}
\texttt{root} &= h_{01234567},\\
h_{01234567} &= h(h_{0123}, h_{4567}),\\
h_{4567} &= h(h_{45}, h^{(00)}),\\
h_{45} &= h(h_4, 0),\\
h_4 &= h(x_4).
\end{aligned}$$

And the same repeats, we update `branch` array to (h_4, h_{01}, h_{0123}) in order to be able to compute the following root when adding a sixth element.

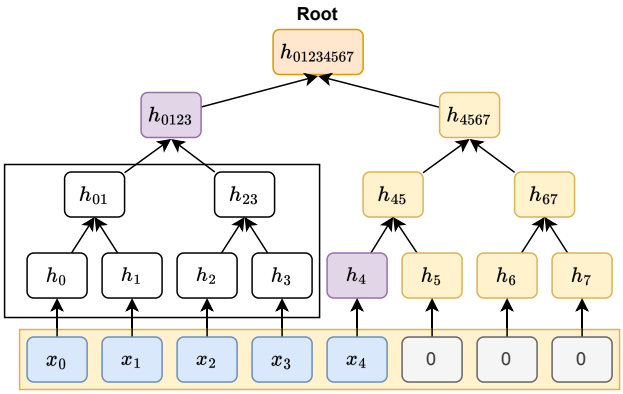

Figure 6: Illustration of the tree state after appending x_4 to the list.

3 Observations and Generalities. Updating the `branch` array and computing the `root`.

Let us provide some observations that we can extract from the previous toy example. Recall that the `branch` array has length d since we do not need more than d elements at the same time in order to compute the `root`. In fact, we usually need less than d elements, leaving the unused elements without update, as we have seen before. Moreover, observe that we are storing an element present in the i level of the tree in the $d - i$ index of the `branch` array. In other words, in the toy example proposed above, elements from the third level ($i = 3$) occupy index 0, elements from the second level ($i = 2$) reside at index 1, and elements from the first level ($i = 1$) are found at index 2 in the `branch` array.

Algorithm for Updating the `branch` array. In important thing to observe is that, the position of the branch that we have to update corresponds to the position of the less significant bit to 1 of the binary representation of `lastElemIndex`.

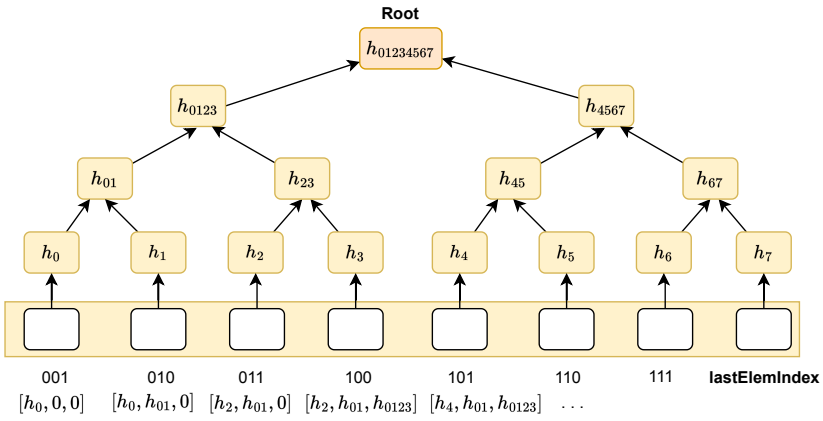

Figure 7: Relationship with `lastElemIndex` and index of `branch` array to be updated.

For example, when `lastElemIndex` is 1, whose binary representation is `0b001`, we need to update the **first** element of the `branch` array

$$(0, 0, 0) \mapsto (h_0, 0, 0).$$

Alternatively, when `lastElemIndex` is 2, whose binary representation is `0b010`, we need to update the **second** element of the `branch` array

$$(h_0, 0, 0) \mapsto (h_0, h_{01}, 0).$$

Finally, when `lastElemIndex` is 4, whose binary representation is `0b100`, we need to update the **third** element of the `branch` array

$$(h_2, h_{01}, 0) \mapsto (h_2, h_{01}, h_{0123}).$$

This fact provides a computable way of updating `branch`, providing us with a systematic algorithm for this task. The following pseudo-code computes the value and index to be written in the branch array:

```
currentHash = dataHash
index = 0

while read bit of lastElemIndex next lsb {
        if(bit == 0) {
                currentHash = h(branch[index]|currentHash)
        } else { // bit == 1
                branch[index]= currentHash
                return
        }
        index++
}
```

Figure 8: Pseudo-code for Updating the `branch` array.

Algorithm for Computing the `root`. Similarly, the presence of a 1 in the binary decomposition of `lastElemIndex` signals the need for zero hashes in the computation of the root. Take for example `lastElemIndex` equal 1, whose binary representation is `0b001`. Recall that we can compute `root` as follows:

$$\begin{aligned} h_{01} &= h(h_0, 0) && \textbf{step 0} \\ h_{0123} &= h(h_{01}, h^{(00)}) && \textbf{step 1} \\ \texttt{root} &= h(h_{0123}, h^{(0000)}) && \textbf{step 2} \end{aligned}$$

Notice that zero hashes come into play precisely when a 0 appears in the binary representation of `lastElemIndex`, written in little endian. In this specific example, zero hashes are utilized in Steps 1 and 2, while in Step 0, they are not required. This fact is completely general.

Consider another case where `lastElemIndex` is equal to 2 with a binary representation of `0b010`. The computation of `root` is as follows:

$$\begin{aligned} h_{01} &= h(h_0, h(x_1)) && \textbf{step 0} \\ h_{0123} &= h(h_{01}, h^{(00)}) && \textbf{step 1} \\ \texttt{root} &= h(h_{0123}, h^{(0000)}) && \textbf{step 2} \end{aligned}$$

In this case, zero hashes are introduced in Step 1 while being absent in Steps 0 and 2, aligning with the expected behavior. Following the previous observations, the following pseudo-code computes the root:

```
currentZeroHash = 0
currentHash = currentZeroHash
index = 0

while read bit of lastElemIndex next lsb {
        if(bit == 0) {
                currentHash = h(currentHash | currentZeroHash)
        } else { // bit == 1
                currentHash = h(branch[index] | currentHash)
        }
        currentZeroHash = h(currentZeroHash | currentZeroHash)
        index++
}
```

Figure 9: Pseudo-code for Computing the `root`.