I took a shot at implementing my own Merkle Tree class and proofs, with the goal of enabling a single compact multi-proof to prove the existence of an arbitrary array of elements, as well append another arbitrary set, with various options (sorting pairs at hash-time, etc).

I got some ideas from various gists and posts I read, so this is not all entirely novel.

Finally published today, and it includes both the javascript class, as well as the compatible smart contracts. It also has many test for each.

I wanted to get some feedback, because in working on this, I eventually ran into Merkle Mountain Ranges, and noticed that, while generally similar, my algorithms don't "bag the peaks", but rather, just bubbles up at most one single child at each level. It's a Merkle tree where we don't bother with anything to the right of an "append index", which is the element count, and that element count is also part of the proof, since the actual root is H(element_count, unbalanced_tree_root)

.

I'm not mathematically rigorous, so my best brief explanation is that the pair hashing algorithm is H(i, j) = i

, where j

is "to the right" of the "append index" (i.e. it doesn't exist).

So, doesn't end up look quite like "mountain ranges", since the peaks don't all hash to the far left, but I do get a nice property that the algorithms are more generic, in that peaks aren't actually special, or even handled uniquely at all.

Further, while I have not implemented it yet, I have a clear outline for how I can infer the indices of elements of a proof, without needing to provide them. Effectively, the compact proof (given a Merkle Tree where pairs aren't sorted at hash-time) is an array of bytes32, where:

- proof[0] is a set of up to 255 bits indicating whether each hash-step will involve 2 known nodes (a provided leaf or a pre-computed hash) or 1 known node and one decommitment
- proof[1] is a set of up to 255 bits indicating whether each hash-step should just be bubbled up the child (skip hashing altogether)
- hashing stops (whe should be at the root) when (proof[0][i] && proof[1][i]), so we can have proofs that are up to 255 hash-steps (although it is rather trivial to increase this)
- proof[2] is a set of up to 255 bits indicating the order of the hash-step (H(a, b) or H(b, a))
- proof[3, n] is a set of decommitments

Without proof[2], the algorithm can perform proofs just fine, if pairs are sorted at hash-time.

Given proof[2] though, I can build up and array of indices (same length as array of elements) by setting bit 2^level to 0 or 1, based on if an element is on the left or right, as a hash is being computed as part of the proof, going up level-by-level, and eventually end up with an array of corresponding indices where the proved/provided elements can be found.

With respect to appending, the decommitments of a single-proof of the non-existent element at the append-index, are sufficient decommitments to append an arbitrary number of elements, and retrieve the new root. Further, given a non-indexed multi-proof (and without inferring indices from it), so long as one of the element is "right enough" in the tree, the decommitments needed for the append-proof can be inferred during the multi-proof steps.

So, if all you're doing is proving many elements, the "current" append-proof decommitments are inferred, allowing appending to the current root. if you're updating many elements as part of the multi-proof, then the "new" append-proof decommitments are inferred, allowing appending to newly updated root. This allows for the "single step and single proof to update many and append many". This currently works, both in the js library, and the smart contracts.

I realize this isn't well-explained, but any feedback is appreciated. Or if I should do a better job at explaining, let me know. I'd be happy to.