A sparse Merkle tree (SMT) is a data structure useful for storing a key/value map which works as follows. An empty SMT is simply a Merkle tree with 2^{256}

leaves, where every leaf is a zero value. Because every element at the second level of the tree is the same ($z2 = hash(0, 0)$

), and every element at the third level is the same ($z3 = hash(z2, z2)$

) and so forth this can be trivially computed in 256 hashes. From there, we can add or remove values by modifying values in place in the sparse Merkle tree, eg. to add the value 42 at position 3, we modify the second value the second level to $v2 = hash(0, 42)$

, the first value at the third level to $v3 = hash(0, v2)$

, the first value at the fourth level to $v2 = hash(v3, z3)$

(since at this point, the left subtree represents keys 0…3 and the right subtree represents keys 4…7 which are still all empty), and so forth up to the top.

We don’t need to store any level in a literal array; we can simply store a hash map of $parent \rightarrow (leftchild, rightchild)$

, and in general the map will grow by at most 256 elements for each item we add into the tree. Adding, updating or removing any element takes 256 steps, and a Merkle proof of an element takes 256 hashes, if done naively.

Sparse Merkle trees are conceptually elegant in their extreme simplicity; from a mathematical perspective, they really are just Merkle trees. In simplest form, the get

function is literally ten lines of code:

```
def get(db, root, key): v = root path = key_to_path(key) for i in range(256): if (path >> 255) & 1: v = db.get(v)[32:] else: v = db.get(v)[:32] path <<= 1 return v
```

But they are seemingly less efficient: for a tree with N non-empty elements, a more specialized key/value tree, eg. a Patricia tree or Merklix tree or AVL tree, requires $\log(N)$

operations to read or write a value, and a Merkle branch has length $\approx 32 * \log_2(N)$

optimally, whereas in a sparse Merkle tree, it seems like we require 256 operations to read or write, and a Merkle branch has length $32 * 256 = 8192$

bytes.

However, these inefficiencies are largely illusory

; it turns out that it is possible to optimize away most of the inefficiencies by building more complex client-side algorithms on top of sparse Merkle trees. First, we can cut down the Merkle branch size to $32 + 32 * \log_2(N)$

(actually lower overhead than my earlier implementation of a binary Patricia tree), by using a simple trick. If a sparse Merkle tree has N

nonzero nodes, then once the branch goes deeper than $\log(N)$

levels into the tree, it is likely that there is only one nonzero node in the remaining subtree. At this point, every time you go one level further down, the subtree that does not contain the node will be the zero subtree for that given level. Since there are only 256 possible values for a zero subtree, we can calculate and store these once, and simply omit them from the proof. We can prepend 32 bytes to the proof to show the client at what indices we omitted a zero subtree value. Here’s the algorithm:

```
def compress_proof(proof): bits = bytearray(32) oproof = b'' for i, p in enumerate(proof): if p == zerohashes[i+1]: bits[i // 8] ^= 1 << i % 8 else: oproof += p return bytes(bits) + oproof

def decompress_proof(oproof): proof = [] bits = bytearray(oproof[:32]) pos = 32 for i in range(256): if bits[i // 8] & (1 << (i % 8)): proof.append(zerohashes[i+1]) else: proof.append(oproof[pos: pos + 32]) pos += 32 return proof
```

But we can go further. Although we will not be able to reduce the number of hashes

required to update a value to less than 256, we can reduce the number of disk reads

required to $\log_2(N)$

, or even $\log_{16}(N)$

(ie. same as the current Patricia tree). The algorithm is to simply change the way that we store Patricia tree nodes, in some cases grouping multiple nodes together in a DB entry. If there is a subtree with only one element, we can simply store a record saying what the value is, what the key is, and what the hash is (to avoid having to recompute it). I do this in this sample implementation:

https://github.com/ethereum/research/blob/master/trie_research/bintrie2/new_bintrie_optimized.py

Notice that this ends up looking somewhat similar to an implementation of a Patricia tree, except the Patricia-tree-like structure exists only at the database level, not at the hash computation level, where it is still a sparse Merkle tree. The same keys and values put into the tree will still give the same hash.

And here is an implementation that pretends that the binary tree is a hexary tree, using H(H(H(H(i0, i1), H(i2, i3)), H(H(i4, i5), H(i6, i7))), H(H(H(i8, i9), H(i10, i11)), H(H(i12, i13), H(i14, i15))))

as the hash function. This once again does not affect hash computation, so the same keys and values put into the tree will continue to give the same hash, but it reduces the number of database reads/writes required by a factor of ~3-4 relative to the previous binary tree implementation.

https://github.com/ethereum/research/blob/master/trie_research/bintrie2/new_bintrie_hex.py

The decoupling between hash computation format and database storage format allows different clients to have different implementations, making it much easier to optimize implementations over time.