A Patricia Merkle Trie provides a cryptographically authenticated data structure that can be used to store all (key, value) bindings.

Patricia Merkle Tries are fully deterministic, meaning that a trie with the same (key, value) bindings is guaranteed to be identical—down to the last byte. This means they have the same root hash, providing the holy grail of O(log(n)) efficiency for inserts, lookups and deletes. Also, they are simpler to understand and code than more complex comparison-based alternatives, like red-black trees.

## Prerequisites

It would be helpful to have basic knowledge of Merkle trees and serialization to understand this page.

## Basic radix tries

In a basic radix trie, every node looks as follows:

[i0, i1 ... in, value]

Where i0 ... in represent the symbols of the alphabet (often binary or hex), value is the terminal value at the node, and the values in the i0 ... in slots are either NULL or pointers to (in our case, hashes of) other nodes. This forms a basic (key, value) store. For example, if you are interested in the value that is currently mapped to dog in the trie, you would first convert dog into letters of the alphabet (giving 64 6f 67), and then descend the trie following that path until you find the value. That is, you would first look up the root hash in a flat key/value DB to find the root node of the trie (which is an array of keys to other nodes), use the value at index 6 as a key (and look it up in the flat key/value DB) to get the node one level down, then pick index 4 of that to look up the next value, then pick index 6 of that, and so on, until, once you followed the path: root -> 6 -> 4 -> 6 -> 15 -> 6 -> 7, you look up the value of the node that you have and return the result.

There is a difference between looking something up in the 'trie' and the underlying flat key/value 'DB'. They both define key/values arrangements, but the underlying DB can do a traditional 1 step lookup of a key. Looking up a key in the trie requires multiple underlying DB lookups to get to the final value described above. Let's refer to the latter as a path to eliminate ambiguity.

The update and delete operations for radix tries are simple, and can be defined roughly as follows:

def update(node,path,value):

if path == '':

curnode = db.get(node) if node else [ NULL ] \* 17

newnode = curnode.copy()

newnode[-1] = value

else:

curnode = db.get(node) if node else [ NULL ] \* 17

newnode = curnode.copy()

newindex = update(curnode[path[0]],path[1:],value)

newnode[path[0]] = newindex

db.put(hash(newnode),newnode)

return hash(newnode)

def delete(node,path):

if node is NULL:

return NULL

else:

curnode = db.get(node)

newnode = curnode.copy()

if path == '':

newnode[-1] = NULL

else:

newindex = delete(curnode[path[0]],path[1:])

newnode[path[0]] = newindex

if len(filter(x -> x is not NULL, newnode)) == 0:

return NULL

else:

db.put(hash(newnode),newnode)

return hash(newnode)

Show all

The "Merkle" part of the radix trie arises in the fact that a deterministic cryptographic hash of a node is used as the pointer to the node (for every lookup in the key/value DB key == keccak256(rlp(value)), rather than some 32-bit or 64-bit memory location as might happen in a more traditional trie implemented in C. This provides a form of cryptographic authentication to the data structure; if the root hash of a given trie is publicly known, then anyone can provide a proof that the trie has a given value at a specific path by providing the hashes of each node joining a specific value to the tree root. It is impossible for an attacker to provide a proof of a (path, value) pair that does not exist since the root hash is ultimately based on all hashes below it, so any modification would change the root hash.

While traversing a path one nibble at a time, as described above, most nodes contain a 17-element array. One index for each possible value held by the next hex character (nibble) in the path, and one to hold the final target value if the path has been fully traversed. These 17-element array nodes are called branch nodes.

## Merkle Patricia Trie

However, radix tries have one major limitation: they are inefficient. If you want to store just one (path,value) binding where the path is (in the case of the ethereum state trie), 64 characters long (number of nibbles in bytes32), you will need over a kilobyte of extra space to store one level per character, and each lookup or delete will take the full 64 steps. The Patricia trie introduced here solves this issue.

### Optimization

Merkle Patricia tries solve the inefficiency issue by adding some extra complexity to the data structure. A node in a Merkle Patricia trie is one of the following:

1. NULL (represented as the empty string)
2. branch A 17-item node [ v0 ... v15, vt ]
3. leaf A 2-item node [ encodedPath, value ]
4. extension A 2-item node [ encodedPath, key ]

With 64 character paths it is inevitable that after traversing the first few layers of the trie, you will reach a node where no divergent path exists for at least part of the way down. It would be naive to require such a node to have empty values in every index (one for each of the 16 hex characters) besides the target index (next nibble in the path). Instead we shortcut the descent by setting up an extension node of the form [ encodedPath, key ], where encodedPath contains the "partial path" to skip ahead (using compact encoding described below), and the key is for the next db lookup.

In the case of a leaf node, which can be determined by a flag in the first nibble of encodedPath, the situation above occurs and also the "partial path" to skip ahead completes the full remainder of a path. In this case value is the target value itself.

The optimization above however introduces some ambiguity.

When traversing paths in nibbles, we may end up with an odd number of nibbles to traverse, but because all data is stored in bytes format, it is not possible to differentiate between, for instance, the nibble 1, and the nibbles 01 (both must be stored as <01>). To specify odd length, the partial path is prefixed with a flag.

### Specification: Compact encoding of hex sequence with optional terminator

The flagging of both odd vs. even remaining partial path length and leaf vs. extension node as described above reside in the first nibble of the partial path of any 2-item node. They result in the following:

hex char bits | node type partial path length

----------------------------------------------------------

0 0000 | extension even

1 0001 | extension odd

2 0010 | terminating (leaf) even

3 0011 | terminating (leaf) odd

For even remaining path length (0 or 2), another 0 "padding" nibble will always follow.

def compact\_encode(hexarray):

term = 1 if hexarray[-1] == 16 else 0

if term: hexarray = hexarray[:-1]

oddlen = len(hexarray) % 2

flags = 2 \* term + oddlen

if oddlen:

hexarray = [flags] + hexarray

else:

hexarray = [flags] + [0] + hexarray

// hexarray now has an even length whose first nibble is the flags.

o = ''

for i in range(0,len(hexarray),2):

o += chr(16 \* hexarray[i] + hexarray[i+1])

return o

Show all

Examples:

> [ 1, 2, 3, 4, 5, ...]

'11 23 45'

> [ 0, 1, 2, 3, 4, 5, ...]

'00 01 23 45'

> [ 0, f, 1, c, b, 8, 10]

'20 0f 1c b8'

> [ f, 1, c, b, 8, 10]

'3f 1c b8'

Here is the extended code for getting a node in the Merkle Patricia trie:

def get\_helper(node,path):

if path == []: return node

if node = '': return ''

curnode = rlp.decode(node if len(node) < 32 else db.get(node))

if len(curnode) == 2:

(k2, v2) = curnode

k2 = compact\_decode(k2)

if k2 == path[:len(k2)]:

return get(v2, path[len(k2):])

else:

return ''

elif len(curnode) == 17:

return get\_helper(curnode[path[0]],path[1:])

def get(node,path):

path2 = []

for i in range(len(path)):

path2.push(int(ord(path[i]) / 16))

path2.push(ord(path[i]) % 16)

path2.push(16)

return get\_helper(node,path2)

Show all

### Example Trie

Suppose we want a trie containing four path/value pairs ('do', 'verb'), ('dog', 'puppy'), ('doge', 'coin'), ('horse', 'stallion').

First, we convert both paths and values to bytes. Below, actual byte representations for paths are denoted by <>, although values are still shown as strings, denoted by '', for easier comprehension (they, too, would actually be bytes):

<64 6f> : 'verb'

<64 6f 67> : 'puppy'

<64 6f 67 65> : 'coin'

<68 6f 72 73 65> : 'stallion'

Now, we build such a trie with the following key/value pairs in the underlying DB:

rootHash: [ <16>, hashA ]

hashA: [ <>, <>, <>, <>, hashB, <>, <>, <>, [ <20 6f 72 73 65>, 'stallion' ], <>, <>, <>, <>, <>, <>, <>, <> ]

hashB: [ <00 6f>, hashD ]

hashD: [ <>, <>, <>, <>, <>, <>, hashE, <>, <>, <>, <>, <>, <>, <>, <>, <>, 'verb' ]

hashE: [ <17>, [ <>, <>, <>, <>, <>, <>, [ <35>, 'coin' ], <>, <>, <>, <>, <>, <>, <>, <>, <>, 'puppy' ] ]

When one node is referenced inside another node, what is included is H(rlp.encode(x)), where H(x) = keccak256(x) if len(x) >= 32 else x and rlp.encode is the [RLP](https://ethereum.org/en/fundamentals/rlp) encoding function.

Note that when updating a trie, one needs to store the key/value pair (keccak256(x), x) in a persistent lookup table if the newly-created node has length >= 32. However, if the node is shorter than that, one does not need to store anything, since the function f(x) = x is reversible.

## Tries in Ethereum

All of the merkle tries in Ethereum's execution layer use a Merkle Patricia Trie.

From a block header there are 3 roots from 3 of these tries.

1. stateRoot
2. transactionsRoot
3. receiptsRoot

### State Trie

There is one global state trie, and it updates over time. In it, a path is always: keccak256(ethereumAddress) and a value is always: rlp(ethereumAccount). More specifically an ethereum account is a 4 item array of [nonce,balance,storageRoot,codeHash]. At this point it's worth noting that this storageRoot is the root of another patricia trie:

### Storage Trie

Storage trie is where all contract data lives. There is a separate storage trie for each account. To retrieve values at specific storage positions at a given address the storage address, integer position of the stored data in the storage, and the block ID are required. These can then be pased as arguments to the eth\_getStorageAt defined in the JSON-RPC API, e.g. to retrieve the data in storage slot 0 for address 0x295a70b2de5e3953354a6a8344e616ed314d7251:

curl -X POST --data '{"jsonrpc":"2.0", "method": "eth\_getStorageAt", "params": ["0x295a70b2de5e3953354a6a8344e616ed314d7251", "0x0", "latest"], "id": 1}' localhost:8545

{"jsonrpc":"2.0","id":1,"result":"0x00000000000000000000000000000000000000000000000000000000000004d2"}

Retrieving other elements in storage is slightly more involved because the position in the storage trie must first be calculated. The position is calculated as the keccak256 hash of the address and the storage position, both left-padded with zeros to a length of 32 bytes. For example, the position for the data in storage slot 1 for address 0x391694e7e0b0cce554cb130d723a9d27458f9298 is:

In a Geth console, this can be calculated as follows:

> var key = "000000000000000000000000391694e7e0b0cce554cb130d723a9d27458f9298" + "0000000000000000000000000000000000000000000000000000000000000001"

undefined

> web3.sha3(key, {"encoding": "hex"})

"0x6661e9d6d8b923d5bbaab1b96e1dd51ff6ea2a93520fdc9eb75d059238b8c5e9"

The path is therefore keccak256(<6661e9d6d8b923d5bbaab1b96e1dd51ff6ea2a93520fdc9eb75d059238b8c5e9>). This can now be used to retrieve the data from the storage trie as before:

curl -X POST --data '{"jsonrpc":"2.0", "method": "eth\_getStorageAt", "params": ["0x295a70b2de5e3953354a6a8344e616ed314d7251", "0x6661e9d6d8b923d5bbaab1b96e1dd51ff6ea2a93520fdc9eb75d059238b8c5e9", "latest"], "id": 1}' localhost:8545

{"jsonrpc":"2.0","id":1,"result":"0x000000000000000000000000000000000000000000000000000000000000162e"}

### Transactions Trie

There is a separate transactions trie for every block, again storing (key, value) pairs. A path here is: rlp(transactionIndex) which represents the key that corresponds to a value determined by:

if legacyTx:

value = rlp(tx)

else:

value = TxType | encode(tx)

More information on this can be found in the [EIP 2718](https://eips.ethereum.org/EIPS/eip-2718" \t "https://ethereum.org/en/developers/docs/data-structures-and-encoding/patricia-merkle-trie/_blank) documentation.

### Receipts Trie

Every block has its own Receipts trie. A path here is: rlp(transactionIndex). transactionIndex is its index within the block it's mined. The receipts trie never updates. Similarly to the Transactions trie, there are current and legacy receipts. To query a specific receipt in the Receipts trie the index of the transaction in its block, the receipt payload and the transaction type are required. The Returned receipt can be of type Receipt which is defined as the concentenation of transaction type and transaction payload or it can be of type LegacyReceipt which is defined as rlp([status, cumulativeGasUsed, logsBloom, logs]).