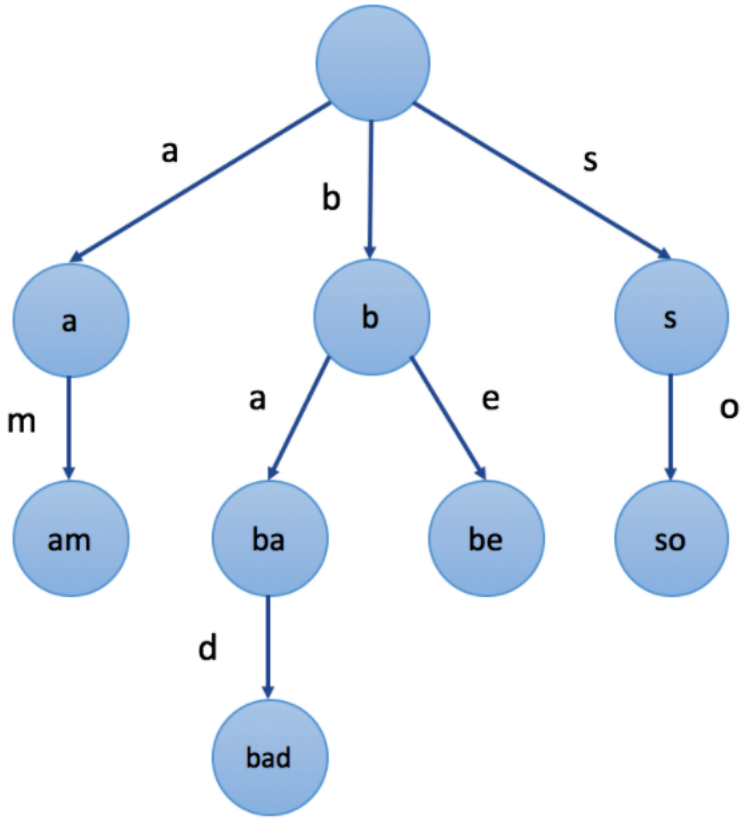
merkle-patricia-trie

August 27, 2020

1 TRIE | PREFIX TRIE | DIGITAL TRIE



^example of what a **TRIE** may look like

- notice how root is empty
- all descendants of a node have a common prefix
- widely used in autocomplete, spell checkers
- can represent a trie with an array or a hashmap

This is how you insert into trie. O(N) time

and how you search in a Trie. O(N) time

Trie is also sometimes called **prefix tree**, it is a special form **N-ary tree** (meaning each node can have no more than **N** children)

Origin of word **trie** is from re**trie**ve

You use an array or a hashmap to implement a **Trie**. For example

[4]:
```
from typing import Tuple
import json


class TrieNode:
    def __init__(self, char: str):
        self.char = char
        self.children = []          # using an array to store children
        self.word_finished = False  # Is it the last character of the word ?
        self.counter = 1            # How many times this character appeared in␣
 ↪the addition process

    # could have used __str__ as well. __repr__ conventional use is to␣
 ↪reconstruct
    # the instance of a class given a string. For example, if class Foo has a␣
 ↪single
    # attribute bar, and its value is 'bar', then __repr__ would return
    # Foo(bar='bar'). Here we use __repr__ to help us visualize the Trie
    def __repr__(self):
        return json.dumps(self._json(), indent=4)

    def _json(self):
        return {
            "char": self.char,
            "children": [child._json() for child in self.children],
            "word_finished": self.word_finished,
            "counter": self.counter,
        }


# Adds a word to a TrieNode
# 1. current node is root at start
# 2. for each character in the word
#     - look for the TrieNode whose value (self.char) == this character
#     - if can't find, create a TrieNode; else traverse down the found TrieNode
#
# Time Complexity of this is O(N * M). Meaning very inefficient
```

```
# N - length of word
# M - maximum number of children a TrieNode can have (a Trie is a special form␣
 ↪N-ary Trie)
#
# How could we improve the time complexity of this algo?
# We could store pointers to the locations of the nodes separately. For example,␣
 ↪given the root (its
# hash, or some sort of key), we can look up the pointer to it in a database␣
 ↪that would hold this info
# for us. We can now get the instance of the root TrieNode, what next? We can␣
 ↪now use the first character
# of the word as an index, to go to the db to get the pointer to the next␣
 ↪TrieNode, and so on.
# You will agree that this is much more efficient. We have paid the price of␣
 ↪extra storage in return for
# the greater performance (time complexity)
# Our TrieNode now, would at most take
# Time Complexity O(N)
# with Space Complexity O(L), where L is the total number of TrieNodes (many of␣
 ↪TrieNode)
# If you look here: https://eth.wiki/en/fundamentals/patricia-tree
# then that is exactly how it is done in Ethereum. The only caveat, Ethereum␣
 ↪doesn't use Tries, at least,
# not the basic prefix tries. Let's continue exploring to understand why
def add(root, word: str):
    node = root
    for char in word:
        found_in_child = False
        for child in node.children:
            if child.char == char:
                child.counter += 1
                node = child
                found_in_child = True
                break

        if not found_in_child:
            new_node = TrieNode(char)
            node.children.append(new_node)
            node = new_node

    node.word_finished = True


# def find_prefix(root, prefix: str) -> Tuple[bool, int]:
#     node = root
#     if not root.children:
```

```
#         return False, 0
#     for char in prefix:
#         char_not_found = True
#         for child in node.children:
#             if child.char == char:
#                 char_not_found = False
#                 node = child
#                 break
#         if char_not_found:
#             return False, 0
#     return True, node.counter


# print(find_prefix(root, 'hac'))
# print(find_prefix(root, 'hack'))
# print(find_prefix(root, 'hackathon'))
# print(find_prefix(root, 'ha'))
# print(find_prefix(root, 'hammer'))
```

[5]:
```
root = TrieNode('*')
add(root, "dog")
add(root, "done")
add(root, "dope")
```

[6]:
```
root
```

[6]:

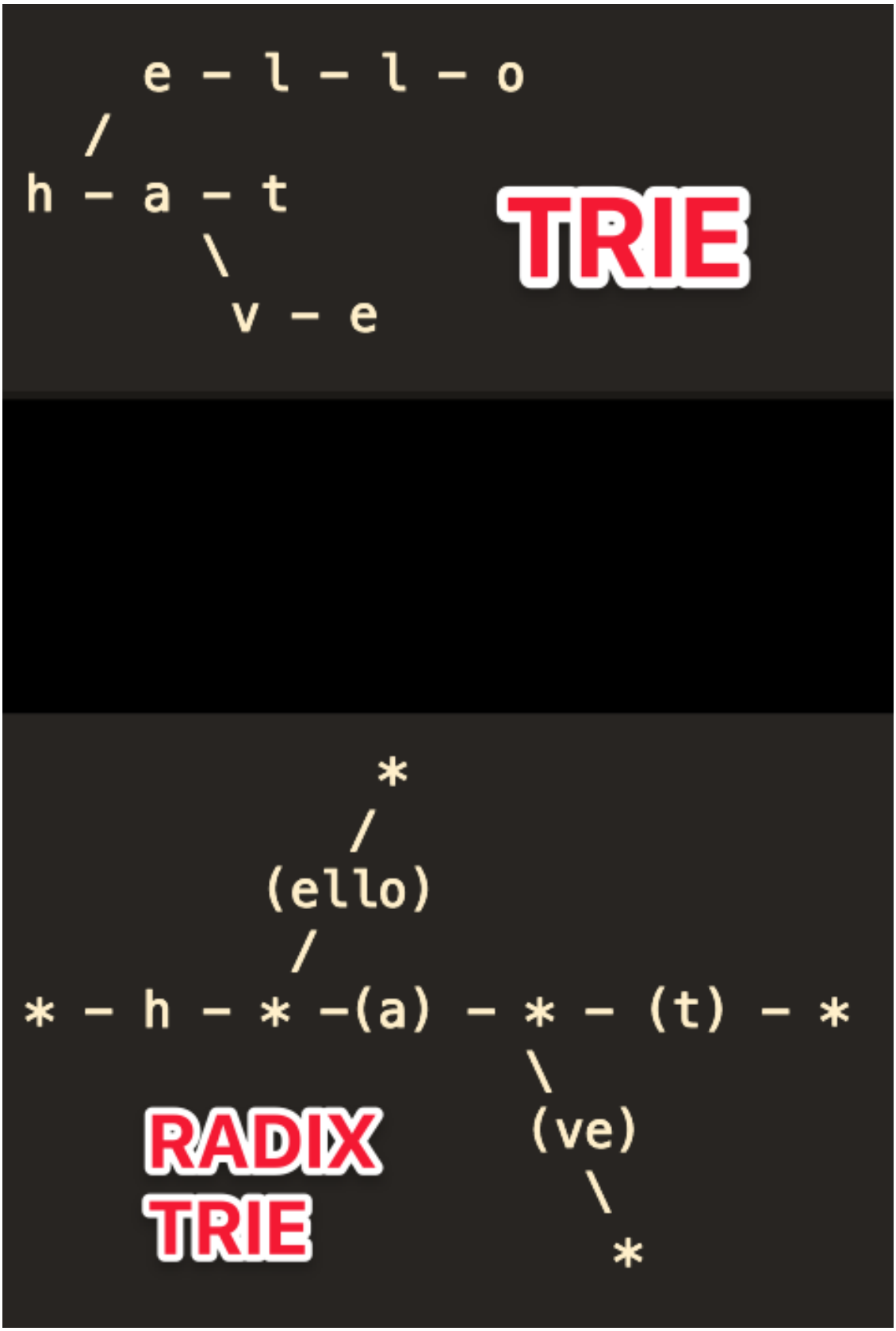
```
{
    "char": "*",
    "children": [
        {
            "char": "d",
            "children": [
                {
                    "char": "o",
                    "children": [
                        {
                            "char": "g",
                            "children": [],
                            "word_finished": true,
                            "counter": 1
                        },
                        {
                            "char": "n",
                            "children": [
                                {
                                    "char": "e",
                                    "children": [],
```

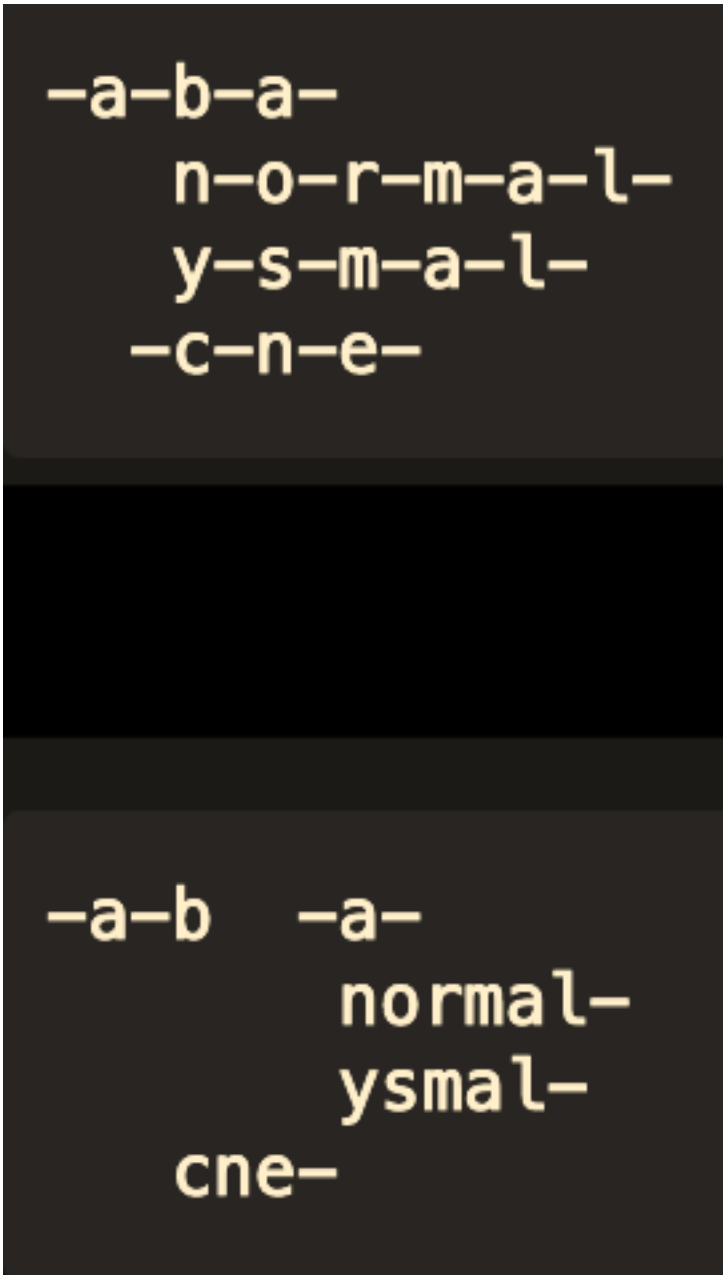
```
                                "word_finished": true,
                                "counter": 1
                            }
                        ],
                        "word_finished": false,
                        "counter": 1
                    },
                    {
                        "char": "p",
                        "children": [
                            {
                                "char": "e",
                                "children": [],
                                "word_finished": true,
                                "counter": 1
                            }
                        ],
                        "word_finished": false,
                        "counter": 1
                    }
                ],
                "word_finished": false,
                "counter": 3
            }
        ],
        "word_finished": false,
        "counter": 3
    }
],
"word_finished": false,
"counter": 1
}
```


1.1 WHY DON'T WE USE TRIE IN ETHEREUM?

- awful time complexity
- even if improved (like above), wastes space (see radix trie below for comparison)
- not useful, because we can't verify the integrity and validity of the data

2 RADIX TRIE





i.e. the RADIX TRIE uses the space better. So RADIX TRIE is just like TRIE, but with better space

3 Merkle Trie | Patricia Trie

Just like Radix trie, with an addition of being cryptographically verifiable

> INCOSISTENCY in eth.wiki (I think):
> From: https://eth.wiki/en/fundamentals/patricia-tree

""" 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 """
^ I believe this explanation is incorrect. It is not radix tries that have this issue, but tries. Radix tries fix this problem, as we have seen from the screenshots above. We do not need to create a 17 item array for each nibble. If we have a single (path, value) binding, then we can just store path and the value in the very first node after root

---

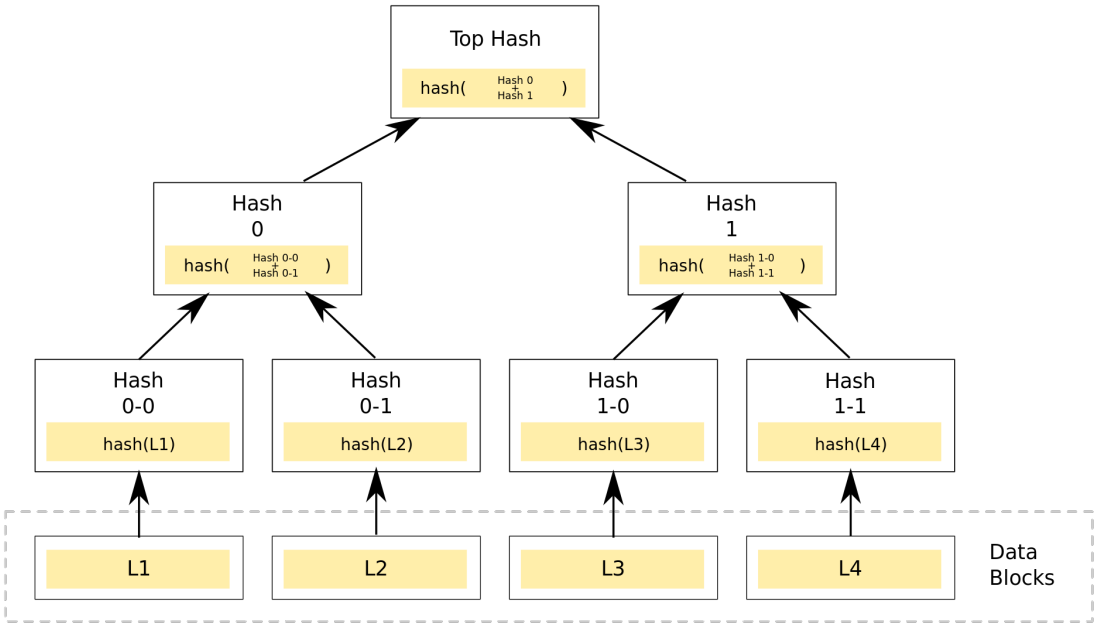
LEAF NODE - a node that doesn't have children

Merkle Tries are usually implemented as binary tries

Merkle Tries are most useful in:

(i) distributed systems, for efficient data verification

e.g. in Git, Tor, Bitcoin ANNNND in Ethereum



Modified Merkle Trie is Ethereum's optimized Merkle Trie

3.1 Modified Merkle Patricia Trie | Merkle Patricia Trie | Modified Merkle Trie

Ethereum's data structure is often called Merkle Patricia Trie, without the Modified prefix

It is "Modified" because it has been optimised for Ethereum's needs

For example, in Modified Merkle Patricia Trie, there are three types of nodes:

(i) **branch node**

(ii) **extension node**

(iii) **leaf node**

We need these, to primarily make better use of space

MPT is

- CRYPTOGRAPHICALLY AUTHENTICATED
- can store all (key, value) bindings (we RLP encode the key)
- O(log N) INSERT, LOOKUP, DELETE

Note

Due to the introduction of extension and leaf nodes, we may end up having to traverse an odd-length remaining path. This introduces a challenge

All paths are stored as `bytes` type, and a single byte is 2 nibbles (2 hex chars). In this setting, how do you distinguish nibble '1' from a nibble '01'? You can't. Both are represented as `<01> bytes` (you cannot create a byte from odd-length nibbles)

1 byte = 2 hex

So we must do something about this. We can trivially solve this issue with flags. We can prefix all the 2-item nodes (leaf and extension) with 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
```

we do not care about the branch node because it does not contain the nibble path, and so does not suffer from this problem

I find it very difficult to understand the Merkle Patricia Trie on eth.wiki, their `compact_encode` function is a bit strange too. So we will be implementing our own hex prefix algorithm from the yellow paper. Here is what it should do

Appendix C. Hex-Prefix Encoding

Hex-prefix encoding is an efficient method of encoding an arbitrary number of nibbles as a byte array. It is able to store an additional flag which, when used in the context of the trie (the only context in which it is used), disambiguates between node types.

It is defined as the function HP which maps from a sequence of nibbles (represented by the set $\mathbb{Y}$) together with a boolean value to a sequence of bytes (represented by the set $\mathbb{B}$):

$$\text{HP}(\mathbf{x}, t) : \mathbf{x} \in \mathbb{Y} \equiv \begin{cases} (16f(t), 16\mathbf{x}[0] + \mathbf{x}[1], 16\mathbf{x}[2] + \mathbf{x}[3], ...) & \text{if } \|\mathbf{x}\| \text{ is even} \\ (16(f(t) + 1) + \mathbf{x}[0], 16\mathbf{x}[1] + \mathbf{x}[2], 16\mathbf{x}[3] + \mathbf{x}[4], ...) & \text{otherwise} \end{cases} \tag{171}$$

$$f(t) \equiv \begin{cases} 2 & \text{if } t \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{172}$$

Thus the high nibble of the first byte contains two flags; the lowest bit encoding the oddness of the length and the second-lowest encoding the flag t. The low nibble of the first byte is zero in the case of an even number of nibbles and the first nibble in the case of an odd number. All remaining nibbles (now an even number) fit properly into the remaining bytes.

To be honest with you, even this is a bit confusing. First of all, there is no definition of what t is, except for the fact that it is boolean. We can probably, assume, that t is the node type (extension or leaf). Though, it remains to be determined whether extension === True or leaf === True

Then, $||x||$ is the notation that is generally used to denote an L_2 norm (i.e. the length of the line /vector in Euclidean space), although, in this particular case, it appears that it is used as the cardinality of the set, where each x[i] is the hex character. It is not clear why we need to do those mathematical operations in any case

Given the above, let's just look at the most popular client out there: geth. And see how they do it

the comment section before the hexToCompact implementation: Trie keys are dealt with in three distinct encodings:

KEYBYTES encoding contains the actual key and nothing else. This encoding is the input to most API functions.

HEX encoding contains one byte for each nibble of the key and an optional trailing 'terminator' byte of value 0x10 which indicates whether or not the node at the key contains a value. Hex key encoding is used for nodes loaded in memory because it's convenient to access.

COMPACT encoding is defined by the Ethereum Yellow Paper (it's called "hex prefix encoding" there) and contains the bytes of the key and a flag. The high nibble of the first byte contains the flag; the lowest bit encoding the oddness of the length and the second-lowest encoding whether the node at the key is a value node. The low nibble of the first byte is zero in the case of an even number of nibbles and the first nibble in the case of an odd number. All remaining nibbles (now an even number) fit properly into the remaining bytes. Compact encoding is used for nodes stored on disk.

```
func hexToCompact(hex []byte) []byte {
        terminator := byte(0)
        if hasTerm(hex) {
                terminator = 1
                hex = hex[:len(hex)-1]
        }
        buf := make([]byte, len(hex)/2+1)
        buf[0] = terminator << 5 // the flag byte
        if len(hex)&1 == 1 {
                buf[0] |= 1 << 4 // odd flag
                buf[0] |= hex[0] // first nibble is contained in the first byte
                hex = hex[1:]
        }
        decodeNibbles(hex, buf[1:])
        return buf
}
```

and here is the test for the above function

```
func TestHexCompact(t *testing.T) {
       tests := []struct{ hex, compact []byte }{
              // empty keys, with and without terminator.
              {hex: []byte{}, compact: []byte{0x00}},
              {hex: []byte{16}, compact: []byte{0x20}},
              // odd length, no terminator
              {hex: []byte{1, 2, 3, 4, 5}, compact: []byte{0x11, 0x23, 0x45}},
              // even length, no terminator
              {hex: []byte{0, 1, 2, 3, 4, 5}, compact: []byte{0x00, 0x01, 0x23, 0x45}},
              // odd length, terminator
              {hex: []byte{15, 1, 12, 11, 8, 16 /*term*/}, compact: []byte{0x3f, 0x1c, 0xb8}},
              // even length, terminator
              {hex: []byte{0, 15, 1, 12, 11, 8, 16 /*term*/}, compact: []byte{0x20, 0x0f, 0x1c, 0xb8}},
       }
       for _, test := range tests {
              if c := hexToCompact(test.hex); !bytes.Equal(c, test.compact) {
                     t.Errorf("hexToCompact(%x) -> %x, want %x", test.hex, c, test.compact)
              }
              if h := compactToHex(test.compact); !bytes.Equal(h, test.hex) {
                     t.Errorf("compactToHex(%x) -> %x, want %x", test.compact, h, test.hex)
              }
       }
}
```

so, as you can see the distinction between the leaf and extension node is made via an optional terminator flag that is placed at the end of the bytearray. From now on, we shall use geth as our source of truth. It appears to me, that the reason for popularity of geth is perhaps, partly due to the clarity of the code

Let's conclude on this note. Perhaps we will dive deeper in the near future. For now, this knowledge will suffice

If you are interested, there is a nice utility in ethereum/hexbytes repo on GitHub. It will nicely format the Python's ugly bytes to hexadecimal for you, here:

```
[87]: HexBytes("\x03\x08wf\xbfh\xe7\x86q\xd1\xeaCj\xe0\x87\xdat\xa1'a\xda\xc0\x01\x1a\x9e\xdd\xc4\x90
      ↪encode("iso-8859-1"))

[87]: HexBytes('0x03087766bf68e78671d1ea436ae087da74a12761dac0011a9eddc4900bf1')
```

---

4 To sum up

Trie - not very efficient in terms of space and time complexity. Mostly used in spell checkers and auto-complete

Radix Trie - mostly used in IP routing and associative arrays implementations

Merkle Trie - cryptographically authenticated data structure, lets you easily and efficiently check if portions or all of the data has been tampered with

Modified Merkle Trie - a Merkle Trie modification that is optimised for Ethereum. Main difference is in the introduction of three new types of nodes: extension, branch and leaf

---

4.1 Correct synnonyms

Trie, prefix trie, digital trie

Radix Trie (sometimes branching factor is specified, for example Radix 2 Trie. Meaning each node has up to 2 child nodes)

Merkle Trie, Patricia Trie, Hash Trie

Modified Merkle Trie, Modified Merkle Patricia Trie, Merkle Patricia Trie <- ETHEREUM's data structure to store world state

---

> Also: *"What is the meaning of radix trie (AKA Patricia trie)?"* this assumes radix trees and PATRICIA trees are one and the same thing, but they are not (e.g. see this answer). PATRICIA trees are trees that you get from running the PATRICIA **algorithm** (also FYI PATRICIA is an acronym, which stands for "Practical Algorithm To Retrieve Information Coded in Alphanumeric"). The resulting trees can be understood as radix trees with `radix = 2`, meaning that you **traverse the tree** by looking up `log2(radix)=1` bits of the input string at a time. – Amelio Vazquez-Reina Mar 5 at 23:33

source: https://stackoverflow.com/questions/14708134/what-is-the-difference-between-trie-and-radix-trie-data-structures

4.2 Word origins

Trie - from re**trie**ve

Patricia - Practical Algorithm To Retrieve Information Coded In Alphanumeric

Radix - "In a positional numeral system, the radix or base is the number of unique digits, including the digit zero, used to represent numbers" <- from Wikipedia

Merkle - Ralph Merkle patented the Merkle tree in 1979

5 Why Modified Merkle Patricia Trie and not XYZ (e.g. $PASTA or $YAM)?

1. Cryptographically secure and efficiently verifiable, especially useful in the distributed setting
2. Optimal insert / lookup / delete time complexities: O(log N)
3. Reasonable-ish space complexity

6 Better data structure for Ethereum?

I have come across https://arxiv.org/pdf/1909.11590.pdf during my research, that may present itself to be a worthy substitute

Authors claim 20k TPS...

[ ]: