Ethereum Data Structures and Encoding

Saravanan Vijayakumaran

Associate Professor
Department of Electrical Engineering
Indian Institute of Technology Bombay

February 13, 2024

Recursive Length Prefix Encoding

RLP Encoding (1/3)

- Applications may need to store complex data structures
- RLP encoding is a method for serialization of such data
- Value to be serialized is either a byte array or a list of values
- The values in a list can be of different types and can themselves be lists
  - Examples: “abc”, [“abc”, [“def”, “ghi”], [“”]]
- The RLP encoding of an object **x**

$$\texttt{RLP}(\mathbf{x}) = \begin{cases} R_b(\mathbf{x}) & \text{if } \mathbf{x} \text{ is a byte array} \\ R_l(\mathbf{x}) & \text{otherwise} \end{cases}$$

- $\texttt{BE}$ stands for big-endian representation of a positive integer

$$\texttt{BE}(x) = (b_0, b_1, ...) : b_0 \neq 0 \wedge x = \sum_{n=0}^{n<\|\mathbf{b}\|} b_n \cdot 256^{\|\mathbf{b}\|-1-n}$$

RLP Encoding (2/3)

- Byte array encoding

$$R_b(\mathbf{x}) = \begin{cases} \mathbf{x} & \text{if} \quad \|\mathbf{x}\| = 1 \wedge \mathbf{x}[0] < 128 \\ (128 + \|\mathbf{x}\|) \cdot \mathbf{x} & \text{else if} \quad \|\mathbf{x}\| < 56 \\ \left(183 + \left\|\mathtt{BE}(\|\mathbf{x}\|)\right\|\right) \cdot \mathtt{BE}(\|\mathbf{x}\|) \cdot \mathbf{x} & \text{else if } \left\|\mathtt{BE}(\|\mathbf{x}\|)\right\| \leq 8 \end{cases}$$

- $(a) \cdot (b) \cdot c = (a, b, c)$
- Examples
  - Encoding of 0xaabbcc = 0x83aabbcc
  - Encoding of empty byte array = 0x80
  - Encoding of 0x80 = 0x8180
  - Encoding of “Lorem ipsum dolor sit amet, consectetur adipisicing elit” = 0xb8, 0x38, ’L’, ’o’, ’r’, ’e’, ’m’, ’ ’, . . . , ’e’, ’l’, ’i’, ’t’
- Length of byte array is assumed to be less than 256^8
- First byte can be at most 191

RLP Encoding (3/3)

- List encoding of $\mathbf{x} = [\mathbf{x}_0, \mathbf{x}_1, \ldots]$

$$
\begin{aligned}
R_l(\mathbf{x}) &= \begin{cases} (192 + \|s(\mathbf{x})\|) \cdot s(\mathbf{x}) & \text{if} \quad \|s(\mathbf{x})\| < 56 \\ \left(247 + \left\|\mathtt{BE}(\|s(\mathbf{x})\|)\right\|\right) \cdot \mathtt{BE}(\|s(\mathbf{x})\|) \cdot s(\mathbf{x}) & \text{otherwise} \end{cases} \\
s(\mathbf{x}) &= \mathtt{RLP}(\mathbf{x}_0) \cdot \mathtt{RLP}(\mathbf{x}_1) \ldots
\end{aligned}
$$

- Examples
  - Encoding of empty list [ ] = 0xc0
  - Encoding of list containing empty list [[ ]] = 0xc1 0xc0
  - Encoding of [[ ], [[ ]], [[ ], [[ ]]]] = 0xc7, 0xc0, 0xc1, 0xc0, 0xc3, 0xc0, 0xc1, 0xc0
- First byte of RLP encoded data specifies its type
  - 0x00, . . . , 0x7f $\implies$ byte
  - 0x80, . . . , 0xbf $\implies$ byte array
  - 0xc0, . . . , 0xff $\implies$ list

Reference: `https://ethereum.org/developers/docs/data-structures-and-encoding/rlp`

Merkle Patricia Trie

Motivation

- The Ethereum world state consists of many key-value mappings
  - Account addresses mapped to account states
  - Contract storage mapping variables to values
  - Transaction indices mapped to transaction bytes
- We need a way to condense these mappings into a hash for efficient consensus and retrieval
- Ethereum uses a Merkle Patricia trie for storing mappings
  - Trie = Tree optimized for information re**trie**val
  - Patricia = Practical Algorithm To Retrieve Information Coded in Alphanumeric
  - Merkle: Tree nodes are hashed to generate a root hash

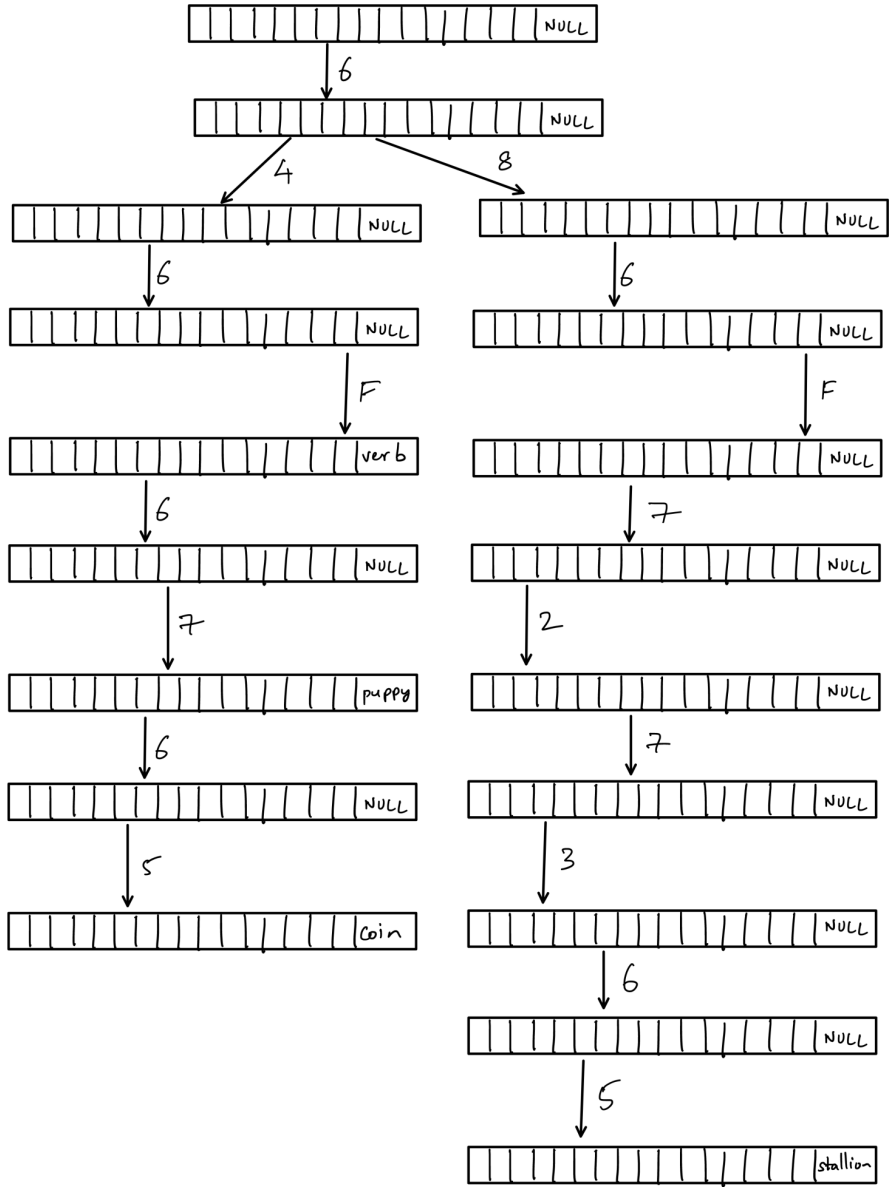
Trie

- A trie is a search tree with k-ary keys
- Example: Trie with hexadecimal string keys
  - Every node is of the form $[i_0, i_1, \ldots, i_{15}, \text{value}]$
  - The i_j entries are pointers to other nodes or NULL
  - Consider key-value pairs: (‘do’, ‘verb’), (‘dog’, ‘puppy’), (‘doge’, ‘coin’), (‘horse’, ‘stallion’)
  - d = 0x64, o = 0x6F, g = 0x67, r = 0x72
  - The mapping with hexadecimal keys

Key	Value
0x64 6F	verb
0x64 6F 67	puppy
0x64 6F 67 65	coin
0x68 6F 72 73 65	stallion

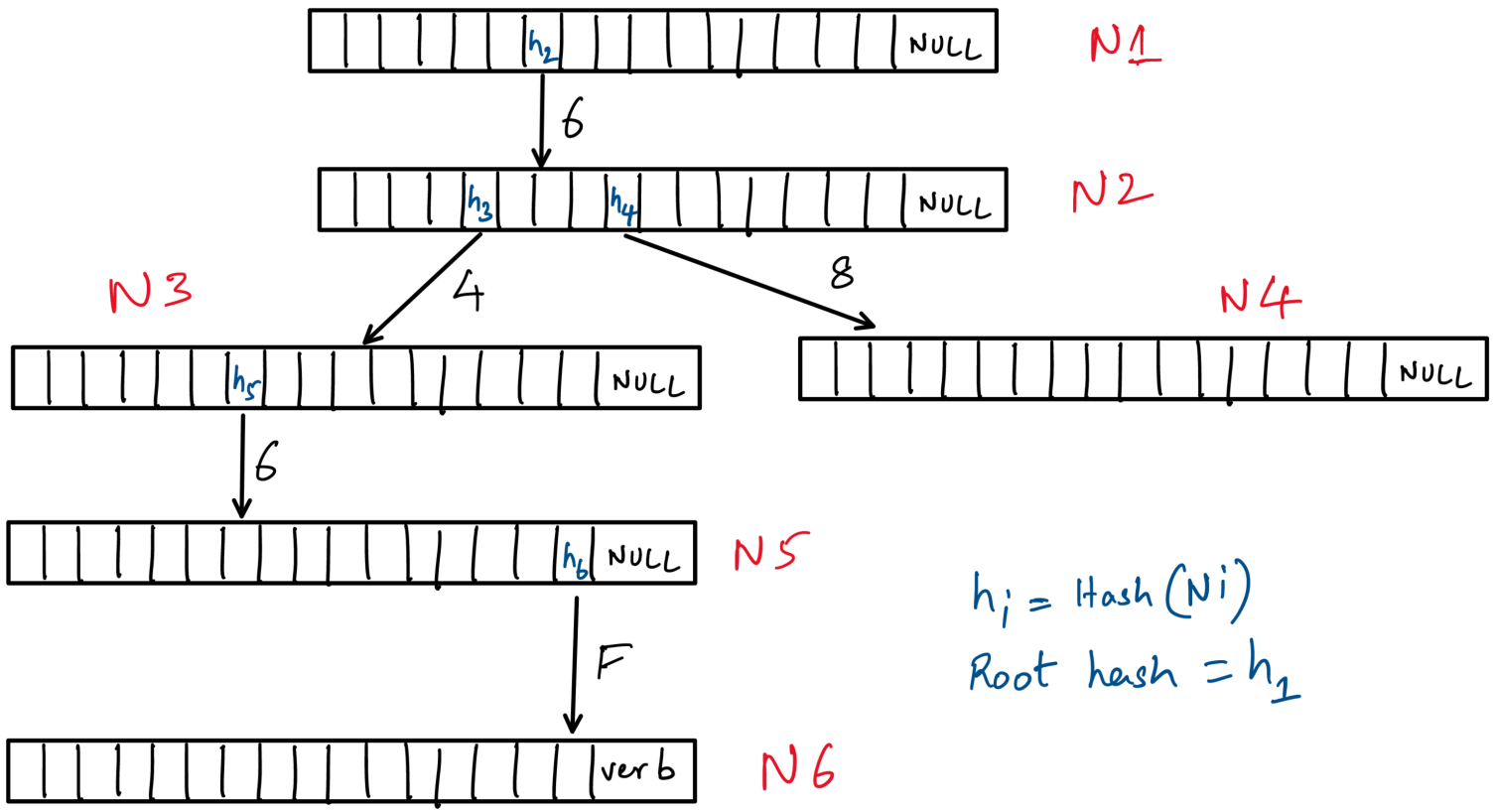
  - What is the corresponding trie?

Trie Example



Merkle Trie

- Merkle tries are a cryptographically secure data structure used to store key-value bindings
- Instead of pointers, the hash of a node is used for lookup in a **key-value database** (like LevelDB)
- In Ethereum, a `node` is stored at the key `Keccak(RLP(node))` in the database



Merkle Trie Update

```
1  # Update value at path in a trie with root hash equal to
       node_hash
2  def update(node_hash, path, value):
3      # Get the node with key node_hash from database
4      # If it does not exist, create a new NULL node
5      curnode = db.get(node_hash) if node else [NULL]*17
6      newnode = curnode.copy()
7
8      if path == '':
9          # If end of path is reached, insert value in current
               node
10         newnode[-1] = value
11     else:
12         # Update node indexed by first path nibble and proceed
13         newindex = update(curnode[path[0]], path[1:], value)
14         # Update hash value of node indexed by first path
               nibble
15         newnode[path[0]] = newindex
16
17     # Insert database entry with hash-node key-value pair
18     db.put(hash(newnode), newnode)
19     return hash(newnode)
```

Source: Ethereum Docs

Merkle Patricia Trie

- Merkle tries are inefficient due to large number of empty nodes
- PATRICIA = Practical Algorithm To Retrieve Information Coded in Alphanumeric
- Node which is an only child is merged with its parent
- A node in a Merkle Patricia trie is either
  - **NULL**
  - **Branch**: A 17-item node $[i_0, i_1, \ldots, i_{15}, \texttt{value}]$
  - **Leaf**: A 2-item node $[\texttt{encodedPath}, \texttt{value}]$
  - **Extension**: A 2-item node $[\texttt{encodedPath}, \texttt{key}]$
- In leaf nodes, `encodedPath` completes the remainder of a path to the target `value`
- In extension nodes
  - `encodedPath` specifies partial path to skip
  - `key` specifies location of next node in database
- Two requirements
  - Need some way to distinguish between leaf and extension nodes
  - `encodedPath` is a nibble array which needs to be byte array

Hex-Prefix Encoding

- Efficient method to encode nibbles into a byte array
- Also stores an additional flag t
- Let $\mathbf{x} = [\mathbf{x}[0], \mathbf{x}[1], \ldots,]$ be a sequence of nibbles

$$\texttt{HP}(\mathbf{x}, t) = \begin{cases} (16f(t), 16\mathbf{x}[0] + \mathbf{x}[1], 16\mathbf{x}[2] + \mathbf{x}[3], ...) & \text{if } \|\mathbf{x}\| \text{ is even} \\ (16(f(t) + 1) + \mathbf{x}[0], 16\mathbf{x}[1] + \mathbf{x}[2], 16\mathbf{x}[3] + \mathbf{x}[4], ...) & \text{o.w.} \end{cases}$$

$$f(t) = \begin{cases} 2 & \text{if } \quad t \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

- High nibble of first byte has two bits of information
  - Lowest bit encodes oddness of length
  - Second-lowest bit encodes the flag
- Low nibble of first byte is zero if length is even and equal to first nibble otherwise

Hex-Prefix Encoding of Trie Paths

- First nibble of `encodedPath`

Hex	Bits	Node Type	Path Length
0	0000	extension	even
1	0001	extension	odd
2	0010	leaf	even
3	0011	leaf	odd

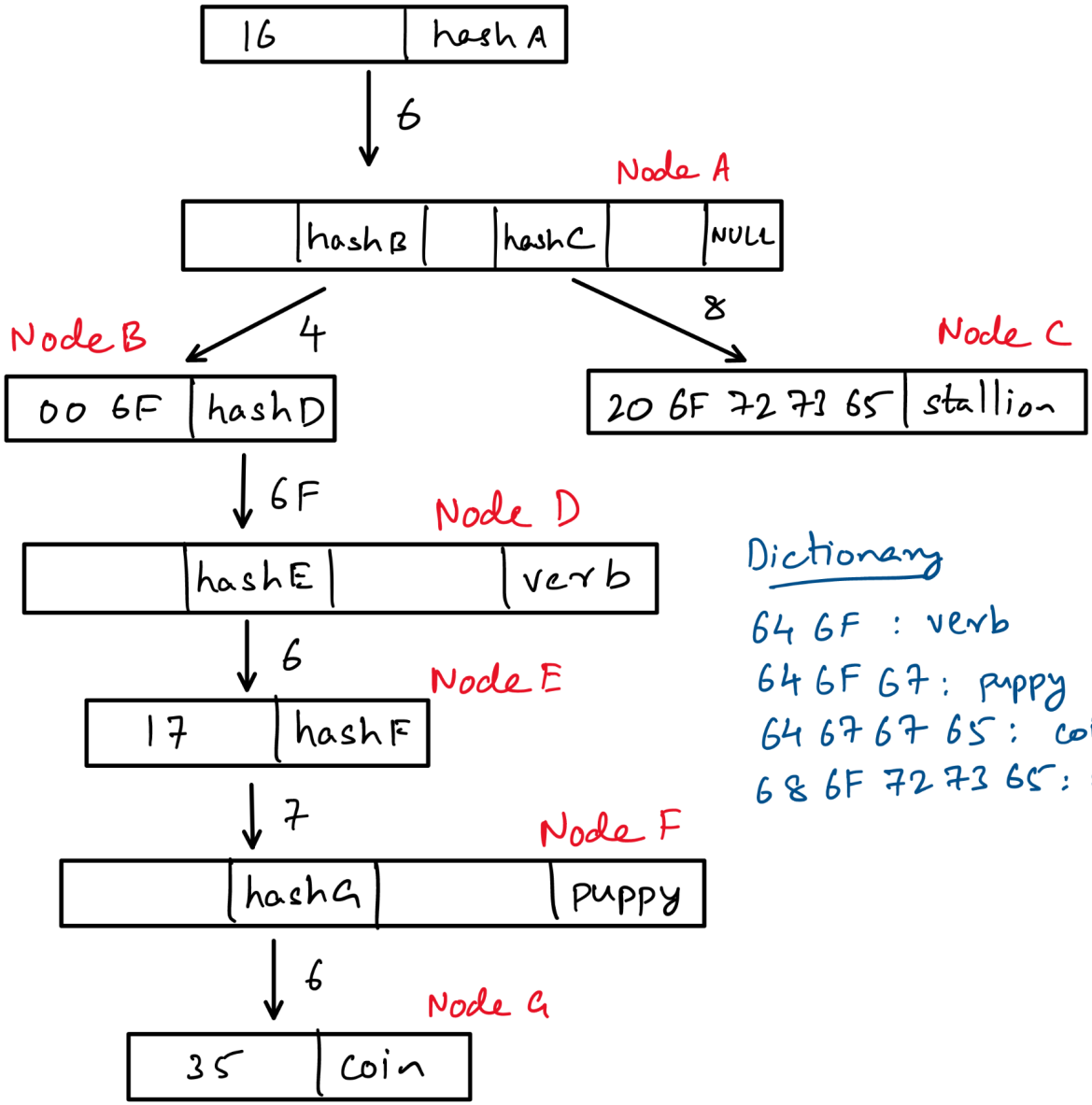
- Examples
  - $[0, f, 1, c, b, 8, \text{value}] \rightarrow$ '20 0f 1c b8'
  - $[f, 1, c, b, 8, \text{value}] \rightarrow$ '3f 1c b8'
  - $[1, 2, 3, 4, 5, \text{key}] \rightarrow$ '11 23 45'
  - $[0, 1, 2, 3, 4, 5, \text{key}] \rightarrow$ '00 01 23 45'

Example Merkle Patricia Trie

- Hex keys and their values
  - 64 6f : ‘verb’
  - 64 6f 67 : ‘puppy’
  - 64 6f 67 65 : ‘coin’
  - 68 6f 72 73 65 : ‘stallion’
- Database view of the Merkle Patricia Trie

rootHash	[<16>, hashA]
hashA	[<>, <>, <>, <>, hashB, <>, <>, <>, hashC, <>, <>, <>, <>, <>, <>, <>, <>]
hashC	[<20 6f 72 73 65>, ’stallion’]
hashB	[<00 6f>, hashD]
hashD	[<>, <>, <>, <>, <>, <>, hashE, <>, <>, <>, <>, <>, <>, <>, <>, <>, ’verb’]
hashE	[<17>, hashF]
hashF	[<>, <>, <>, <>, <>, <>, hashG, <>, <>, <>, <>, <>, <>, <>, <>, <>, ’puppy’]
hashG	[<35>, ’coin’]

Merkle Patricia Trie Example



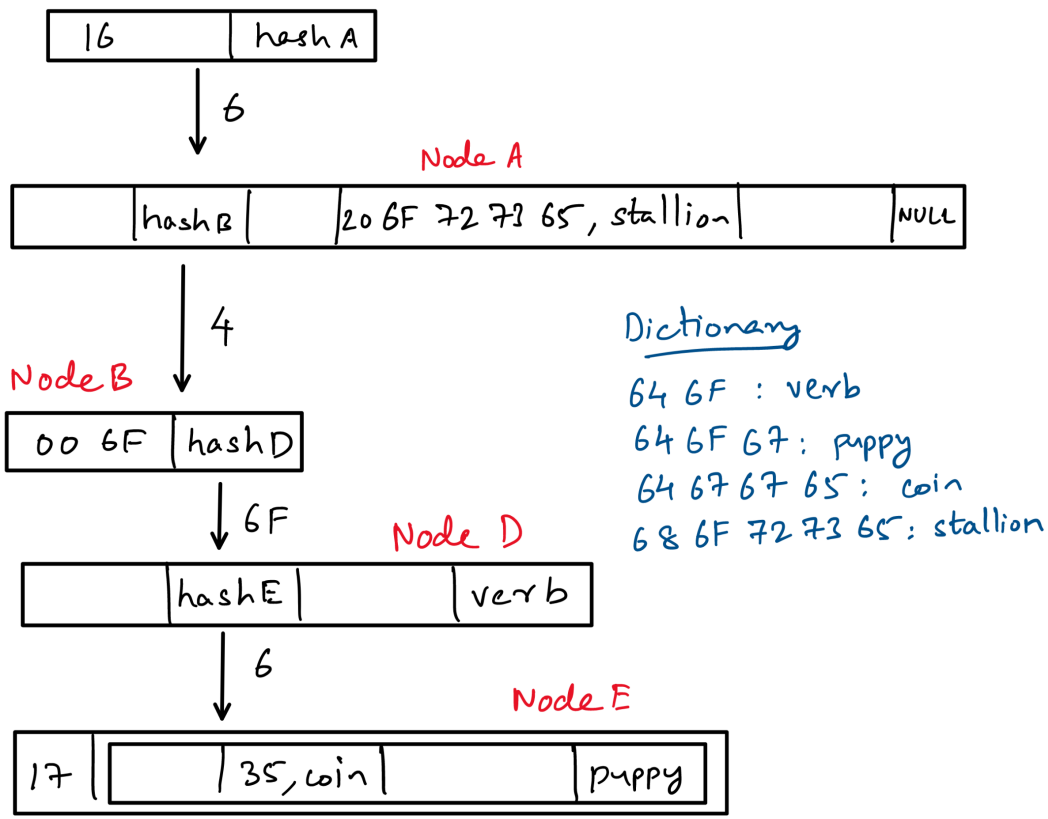
Dictionary

64 6F : verb
64 6F 67 : puppy
64 67 67 65 : coin
68 6F 72 73 65 : stallion

Merkle Patricia Trie with Embedded Nodes

- If its RLP encoding fits in 32 bytes, the node is inserted into its parent instead of the hash

rootHash: [<16>, hashA]
hashA: [<>, <>, <>, <>, hashB, <>, <>, <>, [<20 6f 72 73 65>, 'stallion'], <>, <>, <>, <>, <>, <>, <>, <>]
hashB: [<00 6f>, hashD]
hashD: [<>, <>, <>, <>, <>, <>, hashE, <>, <>, <>, <>, <>, <>, <>, <>, <>, 'verb']
hashE: [<17>, [<>, <>, <>, <>, <>, <>, <>, [<35>, 'coin'], <>, <>, <>, <>, <>, <>, <>, <>, <>, 'puppy']]



References

- Yellow paper https://ethereum.github.io/yellowpaper/paper.pdf
- RLP https://ethereum.org/developers/docs/data-structures-and-encoding/rlp
- Merkle Patricia Tree https://ethereum.org/developers/docs/data-structures-and-encoding/patricia-merkle-trie
- Flow Blog Post https://flow.com/engineering-blogs/ethereum-merkle-patricia-trie-explained