Trees in Ethereum

Trees in Ethereum

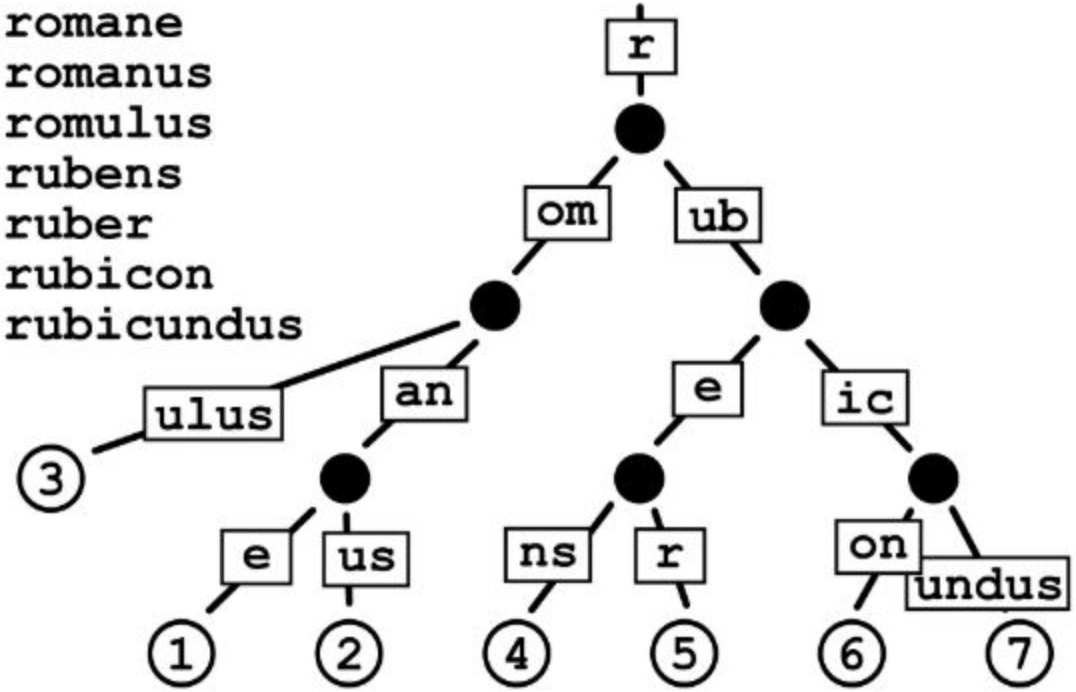
- Ethereum makes use of a data structure called a radix trie, also referred to as a Patricia trie or a radix tree and combines this data structure with a Merkle tree to create a **Patricia Merkle Trie**.

Radix Trie

- “Trie” comes from the word “retrieval”, to give you a hint as to what Patricia Merkle Tries (also referred to as Patricia Merkle Trees) optimize for.
- A radix trie is a tree-like data structure that is used to retrieve a string value by traversing down a branch of nodes that store associated references (keys) that together lead to the end value that can be returned:

Radix Trie

1 romane
2 romanus
3 romulus
4 rubens
5 ruber
6 rubicon
7 rubicundus



Merkle Patricia trie

- A **Merkle Patricia trie** is a data structure that stores key-value pairs, just like a hash table.
- In addition to that, it also allows us to verify data integrity and the inclusion of a key-value pair.
- Patricia Merkle Trees are basically Merkle trees.
- Efficient for data verification needs, but also efficient for editing that data.
- **Patricia??**
    - P = Practical
    - A = Algorithm
    - T = To
    - R = Retrieve
    - I = Information
    - C = Coded
    - I = In
    - A = Alphanumeric

Why Does Ethereum Use a Merkle Patricia Trie

There are typically two different types of data:

- **Permanent**
  - Once a transaction occurs, that record is sealed forever
    - This means that once you locate a transaction in a block’s transaction trie, you can return to the same path over and over to retrieve the same result
- **Ephemeral**
  - In the case of Ethereum, account states change all the time! (ie. A user receives some ether, interacts with a contract, etc)
  - **nonce**, **balance**, **storageRoot**, **codeHash**

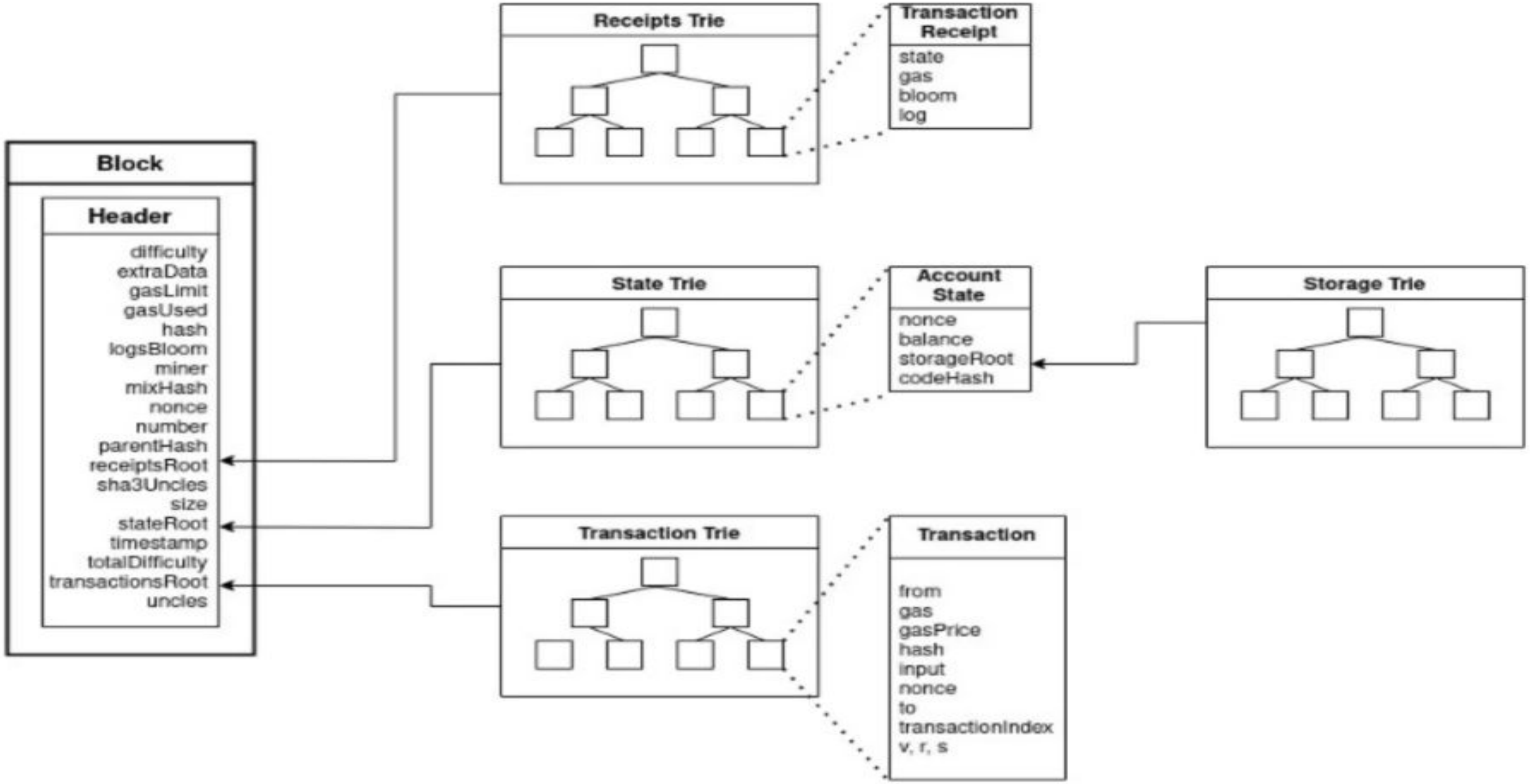
- Permanent data, like mined transactions, and ephemeral data, like Ethereum accounts (balance, nonce, etc), should be stored *separately*.
- Merkle trees, again, are perfect for permanent data. PMTs are perfect for ephemeral data, which Ethereum is in plenty supply of.
- Unlike transaction history, Ethereum account state needs to be frequently updated.
- The balance and nonce of accounts is often changed, and new accounts are frequently inserted.
- Keys in storage are frequently inserted and deleted.

Ethereum Block Header

- The block header is the hash result of all of the data elements contained in a block. It’s kind of like the gift-wrap of all the block data.
- It also includes:
  - **State Root**: the root hash of the state trie
    - The state trie acts as a mapping between addresses and account states.
  - **Transactions Root**: the root hash of the block’s transactions
    - **The transaction trie records transactions in Ethereum**. Once the block is mined, the transaction trie is *never* updated.
  - **Receipts Root**: the root hash of the receipts trie
    - The transaction receipt trie records receipts (outcomes) of transactions.

Note: Refer https://www.alchemy.com/docs/patricia-merkle-tries

Block Header



ECDSA

<u>Key Generation</u>

The ECDSA key-pair (pricey, pubKey) consists of

- *priKey:* private key is a random integer in the range [0...*n*-1]
- *pubKey* = *priKey* * G (the private key, multiplied by the generator point G)

<u>Signature Generation</u>

1. Calculate the message hash h, using a cryptographic hash function h = hash(*message*)
2. Generate a random number k in the range [1..*n*-1]
3. Calculate the random point $R = k * G$ and take its x-coordinate: $r = R.x$
4. Calculate the signature $s = k^{-1} * (h + r * priKey) (\mathrm{mod}\ n)$
5. Return the signature $\{r, s\}$.

The calculated signature $\{r, s\}$ is a pair of integers, each in the range [1...*n*-1].

<u>Signature Verification</u>

To verify a signature takes as input the signed message *msg*, the signature $\{r, s\}$ and the public key *pubKey*, corresponding to the signer's private key. The output is boolean value: *valid* or *invalid* signature. The ECDSA signature verification algorithm works as follows.

1. Calculate the message hash, using the same cryptographic hash function used during the signing process: h = hash(*msg*)
2. Calculate the modular inverse of the signature: $s1 = S^{-1} (\mathrm{mod}\ n)$
3. Recover the random point used during the signing: $R' = (h * s1) * G + (r * s1) * pubKey$
4. Obtain from R' its x-coordinate: $r' = R'.x$
5. Calculate the signature validation result by comparing whether $r' == r$

The common idea of the signature verification is to recover the point R' using the public key and check whether it is same point R, generated randomly during the signing process.