Promethon: new way to look at the next generation of internet

Intro

Ethereum uses a world state to hold states of each account . World state is implemented by using Merkle Tries which makes finding accounts and getting the hash of the whole state faster and with much less process complexity. If we didn't use Merkle Trie and used normal arrays instead, finding and hashing would have process complexity of O(n) but now it has O(log(n)) which is much more efficient.
But the size of this world state is growing rapidly and there hasn't been any efficient solution for shrinking it down.
There has been some ideas that never came to life that best of them are

- State rent (precursor to state expiry), original 2015 proposal: https://github.com/ethereum/EIPs/issues/35
- State expiry A state expiry and statelessness roadmap - HackMD

All of these solutions are experimented in this document
https://hackmd.io/@vbuterin/state_size_management

Process complexity of ethereum transaction handling is O(log(n)) which is and storage complexity of ethereum is O(n) so if we want to make any changes to the current version of ethereum and upgrade it, we should keep in mind that not to make those complexity worse because then our upgrade wouldn't be completely an upgrade and just be trade off between features.

Our project promises to reduce unused accounts without compromising any time complexity. This project fixes the logic of ethereum in which people can have their data saved forever just by giving a saving fee and even one thousand years later, nodes should hold that data even though the person that put it there doesn't need it anymore. And in order for us to achieve this without compromising any storage or time complexity, we use red black tree and Merkle tree together.

To charge every account for staying in world state in each time period, we either have to go through every account and take some money from their balance or we can have all the accounts in order of their balance and only examine the account with the least money because if this account has enough money to stay on our database and it's balance hasn't reached zero to be erased from world state, other accounts has enough money too. The first approach takes time complexity of O(n) but second one takes time complexity of O(log(n)) but having an array which saves our accounts in order of their balance takes time complexity of O(n)! So we use the red black tree as a form of balance ordering data structure that gives us an array of accounts in order of their balances with time complexity of O(log(n)).

In this method, we have a world state database just like ethereum and we also have a red black tree where each node is mapped to an account and the value for each node is the amount of ether they have.

Related Works

There have been a lot of attempts to overcome this issue around the world state of ethereum because it has been a concern for the network from the beginning.
Some of these related works are brought here:

- **The Stateless Client Concept**, original ethresear.ch post (2017):
  https://ethresear.ch/t/the-stateless-client-concept/172
  (see also EthHub)In reality, “stateless” does not really mean “no state”! What it actually means is that you made state someone else’s problem. This proposal takes a completely different approach that is not compatible with the current version of ethereum world state.

  It suggests that we don't need to have all the world state on our full node to validate blocks but instead we can have only the Merkle trie root hash and for verifying use Merkle branches. Despite all the benefits that this approach can bring us; it makes ethereum processes very dependent on other oracles and databases which can cause the network not to work flawlessly and the current user experience will be altered and you won't be able to manage your accounts by only having their private keys.

- **State expiry**
  https://notes.ethereum.org/@vbuterin/verkle_and_state_expiry_proposa
  State expiry suggests that we remove state that has not been recently accessed from the state every once in a while like each year.
  This approach requires the core to switch to Verkle-Trie so that when we want to bring back accounts that have been deleted, we won't need to use large amounts of data to prove its previous state.
  State expiry has a time period and every once in a time period, it cleans up the accounts that haven't been used in the previous time period, the time period suggested to be 1 year.

  One of the problems of State expiry is that it has process complexity of O(n*log(n)). This is a lot on the scale of ethereum. The other problem is that state expiry only solves problem of having too many accounts in the time between two periods(in this case, 2 years) and we can have a big number of useless accounts in the start of the period and our clients should carry those accounts for nearly a complete period (which in our case is 1 year).

  The other problem that should be addressed is the fact that how can we find the best period of time; because if we set the period for too short, the process will be massive because (n*log(n)) is a big number if you consider the fact that (n) in the current version of ethereum can be up to 2^60. And if we set the period for too long, we technically haven't solved any problem and we can have those useless accounts for too long.

The last thing that should be noted is the fact that the current way that ethereum wants to store data is not perfect. When a new data is added to World state, it has a process complexity of O(log(n)) and it takes storage space of O(n) if n be the number of bits. An ethereum client receives an amount of gas to cover the work that it has done and storage that it has used up. But all this money is received by one client and the data that it wants to store should remain in the blockchain forever or at least for one period of time of State expiry and in the meantime all other nodes should contain data even if they came to the network after they were stored.

- **State rent (precursor to state expiry)**, original 2015 proposal: (https://github.com/ethereum/EIPs/issues/35)

  This article proposed the closest approach to our solution but it is not complete. It suggests that we should stop charging for **fee = bytes** and use **fee = bytes * time** but the way it wants to do it is highly dependent on some unknown client that finds accounts that should be deleted and send delete transactions to the network and in the network, nodes calculate the total balance of the account by subtracting the rent from it's balance and see weather it reaches zero or less, if it reached, they delete it from the world state.

  In our proposal, we came up with a solution that gives all the nodes the ability to find out which accounts should be deleted without compromising any time or storage complexity.

  The complete algorithm for this method is to use the current version of ethereum merkle-patricia-trie. And have a function in every node that takes an account and check if it should still stay in our storge by subtracting its **value** by **(bytes * (block passed since the last access) * (fee for each byte per block)** and if this equation equals to zero or less, then the account should be deleted.

  But all the nodes shouldn't check every account to see whether or not they should stay in the storage every block that passes because the time complexity would be massive and unreasonable. But for the network to work correctly, we must ensure that there will always be at least one node that searches for accounts that should be deleted and reports them to the network.

  This article didn't propose any solution for ensuring the existence of that node and didn't propose any solution to reduce the time complexity that checks every account to find the zero balance.

Priminaly Construction

Two of the most important data structures used in our project are red-black tree and Merkle Patricie Trie:

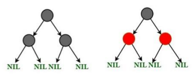
Red-Black tree: When it comes to searching and sorting data, one of the most fundamental data structures is the binary search tree. However, the performance of a binary search tree is highly dependent on its shape, and in the worst case, it can degenerate into a linear structure with a time complexity of O(n).

red black tree is a data structure that is like a binary search tree but in which every node is colored with either red or black, finds elements with time complexity of O(log(n)) but key difference between red black tree and binary tree is that it guarantees after any insertion or deletion, finding elements will still be of time complexity of O(log(n)). But a binary tree in its worst shape can take up to process complexity of O(n). Both insertion and deletion of Red black tree takes process complexity of O(log(n)).

This tree is mostly used when we want to save data in a specific order(ascending or descending) and want to ensure that search of those data won't take time complexity of more than O(log(n))

Red black tree ensures time complexity of O(log(n)) by maintaining the balance of the binary tree. Balance is always ensured because it is impossible to have chain of three nodes:

But it is possible to have these structures for three nodes:

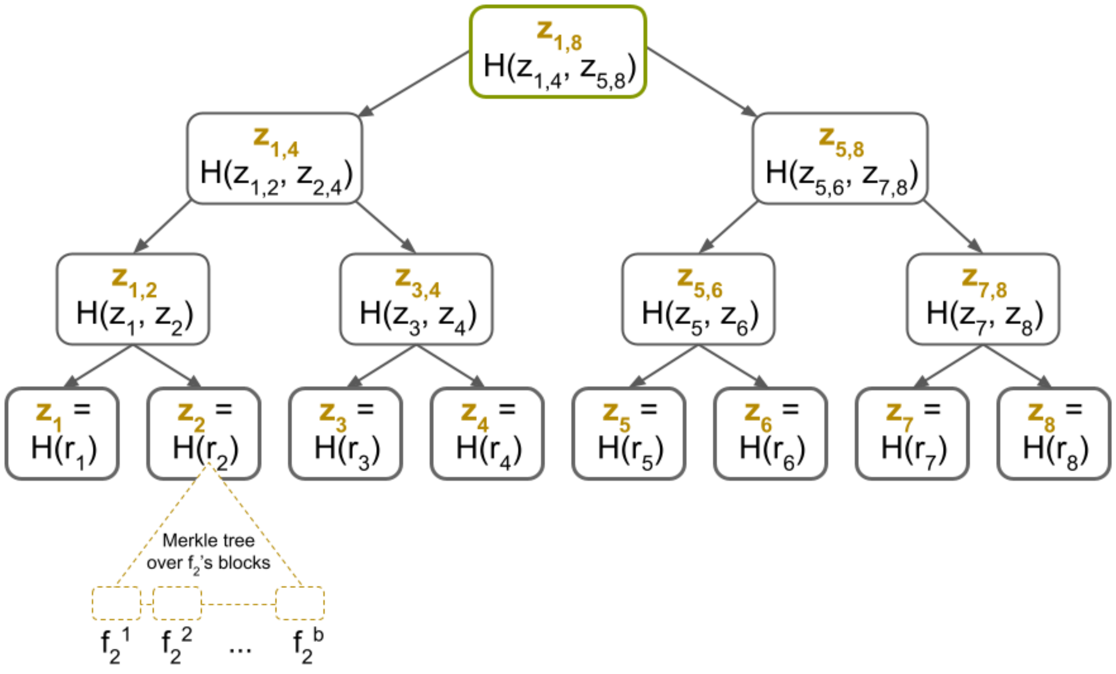


More info: https://www.geeksforgeeks.org/introduction-to-red-black-tree/amp/

In 1972, Rudolf Bayer[5] invented a data structure that was a special order-4 case of a B-tree. In a 1978 paper, "A Dichromatic Framework for Balanced Trees",[7] Leonidas J. Guibas and Robert Sedgewick derived the red–black tree from the symmetric binary B-tree.[8] after that In 1993, Arne Andersson introduced the idea of a right leaning tree to simplify insert and delete operations.[11]and finally In 1999, Chris Okasaki showed how to make the insert operation purely functional. Its balance function needed to take care of only 4 unbalanced cases and one default balanced case.[12]

Merkle Patricia Tree: Merkle tree or hash tree is a tree that is used for data verification. In most data structures when you have the hash of the whole database, if you change just one element in the database, in order to get the new hash of the whole database, you have to do the hash function for all the elements in the database which takes time complexity of O(n) but in Merkle tree, it takes only time complexity of O(log(n)). It is a tree data structure where each non-leaf node is a hash of its child nodes. All the leaf nodes are at the same depth and are as far left as possible.

This tree is usually used in systems that require to compare the whole state of more than one storage and check whether they are similar or not.



More info: https://www.geeksforgeeks.org/introduction-to-merkle-tree/

Merkle trees were invented by Ralph Merkle in 1988 in an attempt to construct better digital signatures.
Original paper: http://people.eecs.berkeley.edu/~raluca/cs261-f15/readings/merkle.pdf

Solution

This project overcomes the issue of having too many unused accounts on any account-state-based blockchains like ethereum.

How do we do it?

We made a rule for paying rent for each account to save their data on the world state. The current method for saving data only costs one time on the execution of the transaction that saves the data and after that, the data will always stay on the blockchain and this causes the problem of having too many unused accounts and contract data because there is no cost for keeping the data.

We accomplish this by reducing value from every account each block that passes and if any balance of account reaches zero, it will be erased.

Problem

Reducing amount from each account requires iteration over the whole data and in ethereum case which is more than 100GB, is sth that surely can not be done in every block creation which is only around 12s.

Solution

If we arrange all the accounts in ascending order and put them in an array, then if we only control the account with minimum balance, we can tell whether other accounts have zero balance or not.

The problem with this approach is that if any account does a transaction and its position changes in the array, rearranging the accounts would require an execution with time complexity of O(n).

Executing any transaction in the current implementation of ethereum has the time complexity of O(log(n)) and changing it to O(n) would not be an improvement and execution of this many transactions with O(n) with the current hardware is impossible.

Solution for this problem is to use red-black-tree instead of an array to hold the accounts with ascending order. Using red-black-tree allows us to rearrange the accounts with time complexity of O(log(n))

We also can achieve this by other trees of sorting with low time complexity.

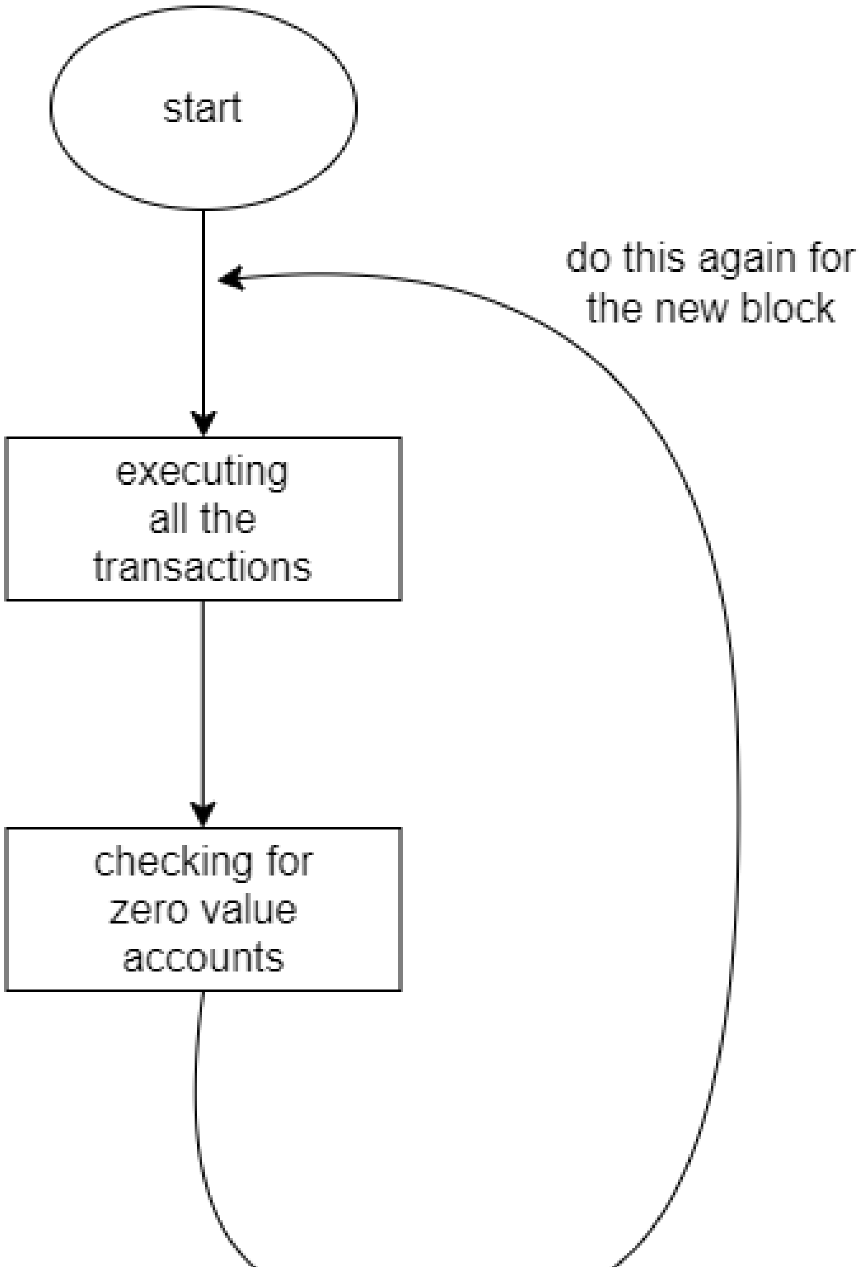
components

- First we should have all the accounts in the merkle patricia trie and our leaf nodes are also nodes for our red black tree
- Key for merkle patricia trie is addresses of accounts.

- Value for each markle patricia trie entry is a pointer to a red black tree node that holds the values of the account it is being mapped to by.
- We also need a red black tree that each of its nodes holds value for one of the accounts in the merkle patricia trie
- Key for each red black tree node: 1- balance of the account 2- last block that this value was modified

Algorithm of executing a block

1. Execute all the transactions
2. Find and delete the accounts that should be deleted


start
do this again for the new block
executing all the transactions
checking for zero value accounts

Algorithm of executing a transaction

Every transaction has a **sender**, **receiver** and a **value**.

1. Follow the red black tree node that is mapped by the Merkle patricia trie address of the sender.
2. Check the **value of the sender's account** by following formula:

$$Value\ of\ account\ to\ check\ = Value\ of\ account\ - (current\ block\ number\ - \ last\ modified\ block\ number)\ *\ (\ bytes\ of\ storage\ that\ the\ account\ takes\ \ *\ \ price\ per\ byte)$$

3. Go ahead if the **value of the sender's account >= transaction value** is correct. if not, revert the transaction.
4. If yes, change the **value of the sender's account to check** by following formula:

$$Value\ of\ account\ \ = Value\ of\ account\ to\ check\ +\ value\ of\ the\ transaction$$

5. And also update the **last modified** value of the **sender**'s account:

$$last\ modified\ =\ block.blocknumber$$

6. Do a fix-up for this node in the red black tree
7. Update the **value of the receiver's account** by following formula:

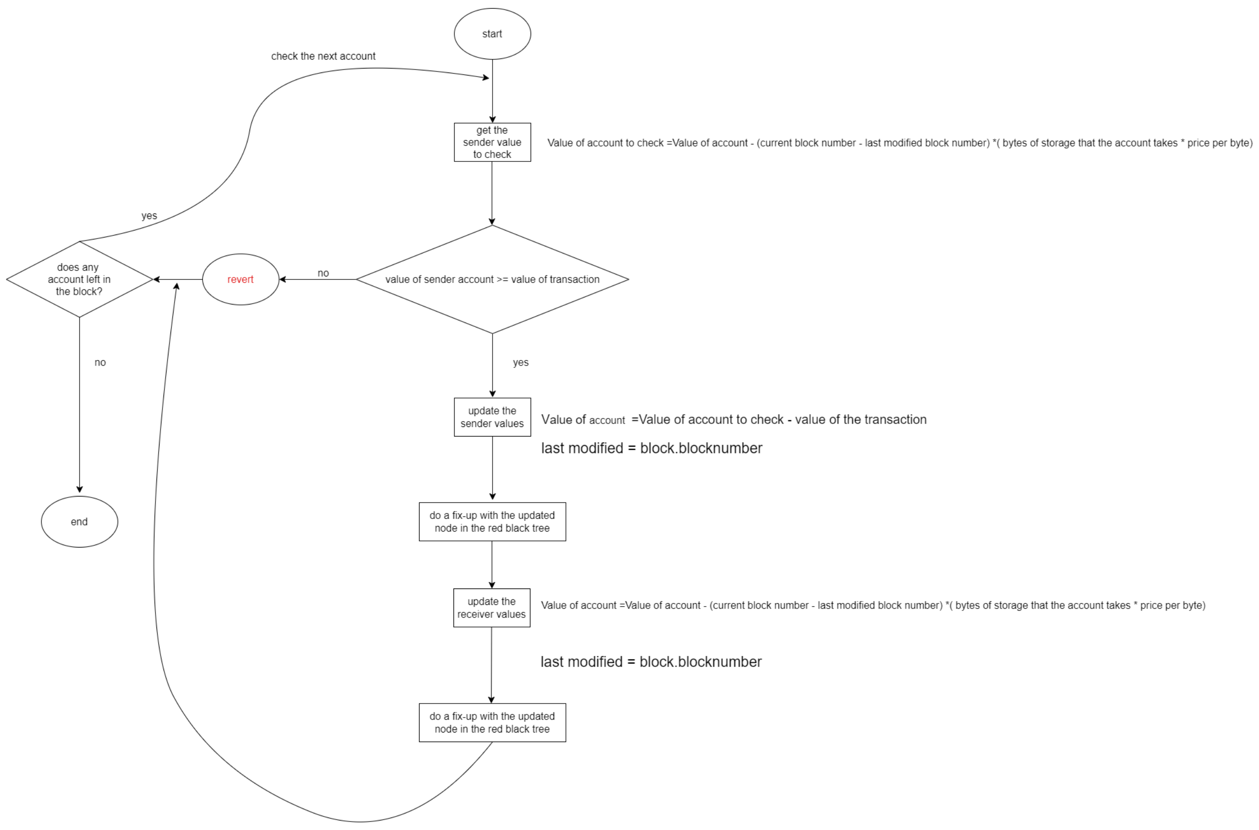
$$Value\ of\ account\ \ = Value\ of\ account\ +\ value\ of\ the\ transaction\ \ - (current\ block\ number\ - \ last\ modified\ block\ number)\ *\ (\ bytes\ of\ storage\ that\ the\ account\ takes\ \ *\ \ price\ per\ byte)$$

8. And also update the **last modified** value of the **receiver**'s account:

$$last\ modified\ =\ block.blocknumber$$

9. Do a fix-up for this node in the red black tree

the value of the sender's account to check is stored in the memory and not in storage, it will be deleted after the execution.


start
check the next account
get the sender value to check
Value of account to check =Value of account - (current block number - last modified block number) *(bytes of storage that the account takes * price per byte)
yes
does any account left in the block?
revert
no
value of sender account >= value of transaction
no
yes
update the sender values
Value of account =Value of account to check - value of the transaction
last modified = block.blocknumber
end
do a fix-up with the updated node in the red black tree
update the receiver values
Value of account =Value of account - (current block number - last modified block number) *(bytes of storage that the account takes * price per byte)
last modified = block.blocknumber
do a fix-up with the updated node in the red black tree

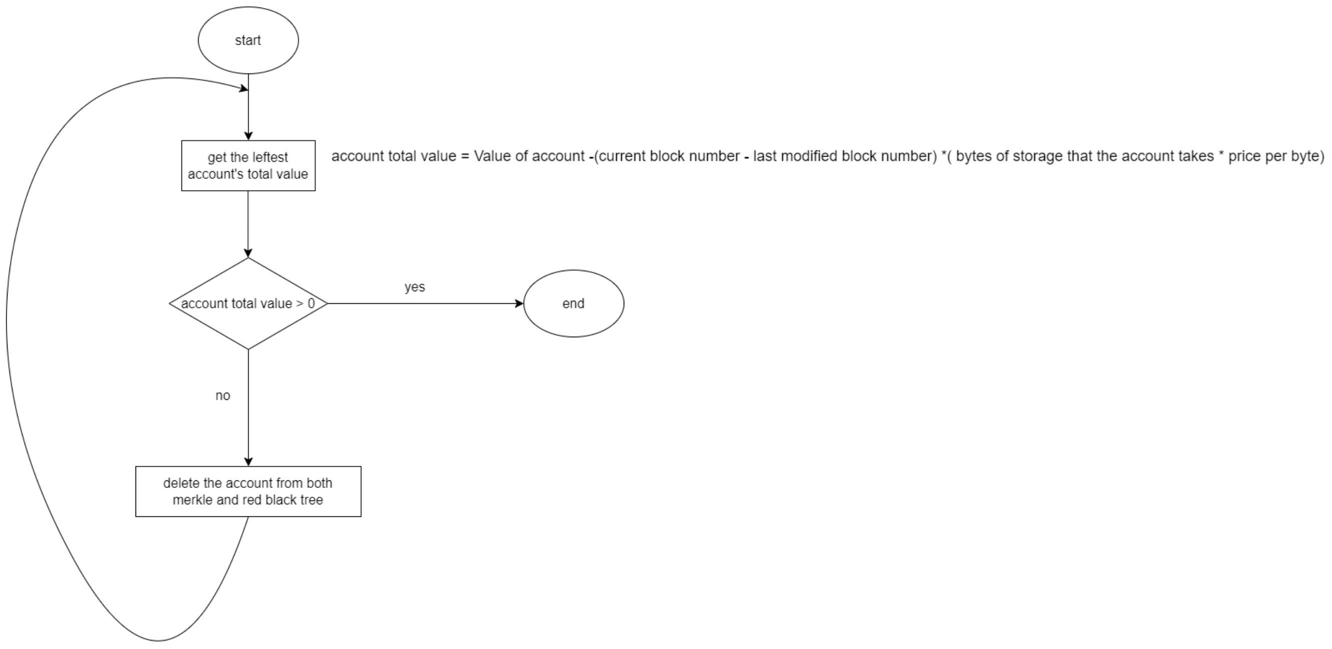
Algorithm of finding and deleting the accounts that should be deleted

1. Go to the left of the red black tree as much as possible and find the node with the least value possible
2. Get the **account total value** by the following formula:

   $$account\ total\ value\ =\ Value\ of\ account\ -\ (current\ block\ number\ -\ last\ modified\ block\ number)\ *\ (\ bytes\ of\ storage\ that\ the\ account\ takes\ *\ price\ per\ byte)$$

3. Check the following statement:
   $account\ total\ value\ >\ 0$
4. If it was greater than zero, there is no account to modify because all the accounts have value more than zero
5. If it was not greater than zero, this account should be deleted so we delete it from both Merkle patricia trie and the red black tree
6. We do this algorithm again from the first step until we reach to an account with total value more than zero

account total value is stored in the memory and not in storage, it will be deleted after the execution.


start
get the leftest account's total value
account total value = Value of account -(current block number - last modified block number) *(bytes of storage that the account takes * price per byte)
account total value > 0
yes
end
no
delete the account from both merkle and red black tree

Evaluation

Executing transactions and blocks and storing data in the storage are some of the most important operations for each full node in the ethereum network.

the current version of ethereum does each one these operations with the following complexity:

- Executing blocks: execution of each block that has m transactions in it, takes time complexity of O(m).
- Executing transactions: execution of transaction takes time complexity of O(log(n)) where n is the height of the Merkle patricia trie which won't exceed more than 40.
- Storing data in the storage: storing data for Merkle trie of the n accounts, has storage complexity of O(n).

Now that we know the complexity of each operations in the ethereum, we should note that if we want to propose a new solution for unused accounts, we shouldn't worsen the complexity of any of these algorithms because the ethereum network highly depend on them and big complexity like O(n) for executing transaction can "mess up"(find better word) the network and nodes won't operate like now because n is a very large number.

Evaluation of the new proposed solution is as follow:

- Executing transaction: executing transaction on the current ethereum network requires finding and modifying data on the Merkle patricia trie and the new proposed method only changes the leaf nodes of the Merkle trie to nodes of a red black tree so finding data stays the same and is O(n)

  Modifying data requires modifying the key value of the red black tree node that the Merkle patricia trie is pointing to. Modifying the data has the time complexity of O(1) with low constant, and after the alteration, we should do a fix-up on the changed value of the red black tree which has time complexity of O(log (n)).

  So the algorithm goes like this:

  1. Find the value of the sender and do the operation mentioned in the "solution" part: O(log (n))
  2. Check whether the sender has the money they want to send with a simple if: O(1)
  3. Update the values of the sender's account: O(1)
  4. Do a fix-up on the red black tree: O(log(n))
  5. Find receiver's account or create if it doesn't exist: O(log(n))
  6. Update the values of the receiver's account: O(1)

7. Do a fix-up on the red black tree: O(log(n))

So to sum upm, it is: O(log(n)) + O(1) + O(1) + O(log(n)) + O(log(n)) + O(1) + O(log(n)) = O(log(n))

- Executing blocks: for execution of blocks, first ethereum executes all the transactions inside it and after that, nodes do the consensus algorithm to keep the network safe. In our approach, everything stays the same as ethereum but one part is included and that is deleting the accounts with zero or less ether.
  Thanks to the red-black tree, we know that the account with the least ether is in the leftist part of the tree and we only need to check the balance of one node. If it had zero or less ether, it should be deleted from both Merkle and red black trees and there should be a fix-up in the red-black tree.
  There should be a loop that iterates over the leftist account of the red-black tree and deletes the accounts with less or zero balance and stops the loop if it reaches to an account more than zero ether.

  The new algorithm only has the mentioned loop in addition to the original ethereum algorithm. If in the current state we have m accounts that need to be deleted, the loop iterates m times and in every iteration, it deletes an account which has time complexity of O(log(n)) that n is the number of total accounts. So the time complexity is O(m * log(n)).
  M can be as big as n but this happens when the network is going to be abandoned forever. So m is approximately n/total number of blocks until the blockchain ends.

- Storing data in the storage: for storing n accounts in the Merkle patricia trie we have storage complexity of O(n) and storing n account details on red-black trie has storage complexity of O(n). So in total, storage has the complexity of O(n) + O(n) = O(n) and is the same as the current version of ethereum.

Evaluation in comparison to state expiry:

N = number of accounts

M = number of accounts that should be deleted in each block ≃ n/ total blocks in the blockchain in all time(m is very low)

T = number of transactions in each block

	Our approach	State expiry
Executing transaction	O(log(n))	O(log(n))
Executing block	O(t) + O(m)	O(t)
Storage complexity	O(n)	O(n)
Additional complexity on each state(one year)	none	O(n)

Benefits

According to the leader of the go-ethereum project (the official node client for ethereum network), at the time of writing this article, the Ethereum world state is near 185 GB reference. By the data gathered from a big query, this world state holds for over 261615030 accounts.

Three out of every five of these accounts do not have any ether(158195507 accounts) and all of the EOA(externally owned account) addresses between these, have useless data and are abandoned. But because of the current rules in the ethereum network, they are not deleted and their data should stay in the network for eternity.

So when 60% of ethereum world state consists of useless data and this useless data is growing rapidly. Our solution is to use a rent mechanism to ensure that everything that states in the network is useful and valuable for its owner. And when the owner doesn't need the data any more, they can stop paying for the rent and the data will be erased automatically.

This method will instantly erase near 112 GB of ethereum world state and will erase ….. More in ….. Days if the owners don't charge their accounts.