100_011 MerkleTree.Blockchain

Kursinių darbų gynimas: Gruodžio 13, 20 d., 17:30 per Zoom.

ledger

Bookkeeping --> accounting --> balance --> state

Bookkeeping is the recording of financial transactions, and is part of the process of accounting in business.[1] Transactions include purchases, sales, receipts and payments by an individual person or an organization/corporation. There are several standard methods of bookkeeping, including the single-entry and double-entry bookkeeping systems.

From <https://en.wikipedia.org/wiki/Bookkeeping>

https://www.dreamstime.com/stock-image-d-life-cycle-accounting-process-illustration-circular-flow-chart-image30625511

Ethereum

IBM Hyperledger Fabric - IBM HF

Authorized capital
Credit
Fixed Assets
Costs
Incomes

Op.No.	Input	Output	RemainingAmount
1	123	0	123
2	5	11	117

Compare with UTXO system
https://medium.com/@olxc/ethereum-and-smart-contracts-basics-e5c84838b19

State 1

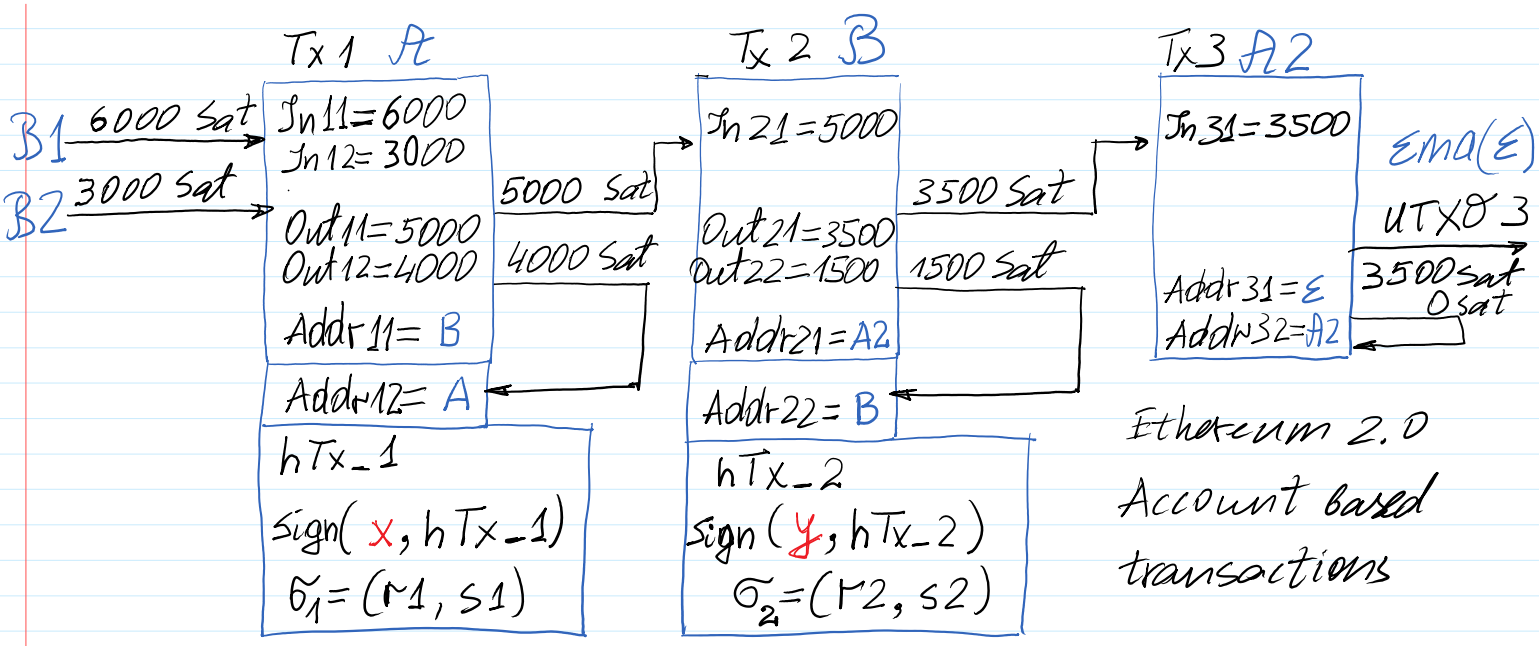
Authorized Capital	Credit	Fixed Assets	Electricity Cost	Mining	Percent for Credit	Balance

State 2

Authorized Capital	Credit	Fixed Assets	Electricity Cost	Mining	Percent for Credit	Balance

$1\,BTC = 10^{8}\,sat$; $1\,sat = 10^{-8}\,BTC$

Unspent Transactions Output – UTxO



$$\Sigma_{In} = \Sigma_{out} \; : \quad Tx\,1 \; : \quad In11 + In12 = Out11 + Out12$$
$$6000 + 3000 = 5000 + 4000 = 9000$$

'Tx1 : In11 = 6000 || In12 = 3000 || Out11 = 5000 || Out12 = 4000 || Rec1=B || Rec2=A'

$$hTx\_1 = h28(\downarrow)$$

Transaction template:
Tx_N = 'TxN:In11=...||In12=...||Out11=...||Out12=...||Rec1=...||Rec2=...'

Transactions:
Tx_1 = 'Tx1:In11=6000||In12=3000||Out11=5000||Out12=4000||Rec1=B||Rec2=A'
Tx_2='Tx2:In21=5000||Out21=3500||Out22=1500||Rec1=A2||Rec2=B'
Tx_3='Tx3:In31=3500||Out31=3500||Out32=0||Rec1=E||Rec2=A2'

>> hTx_1=h28('Tx1:In11=6000||In12=3000||Out11=5000||Out12=4000||Rec1=B||Rec2=A')
hTx_1 = 5B5412B
>> hTx_1=h28(Tx_1)
hTx_1 = 5B5412B

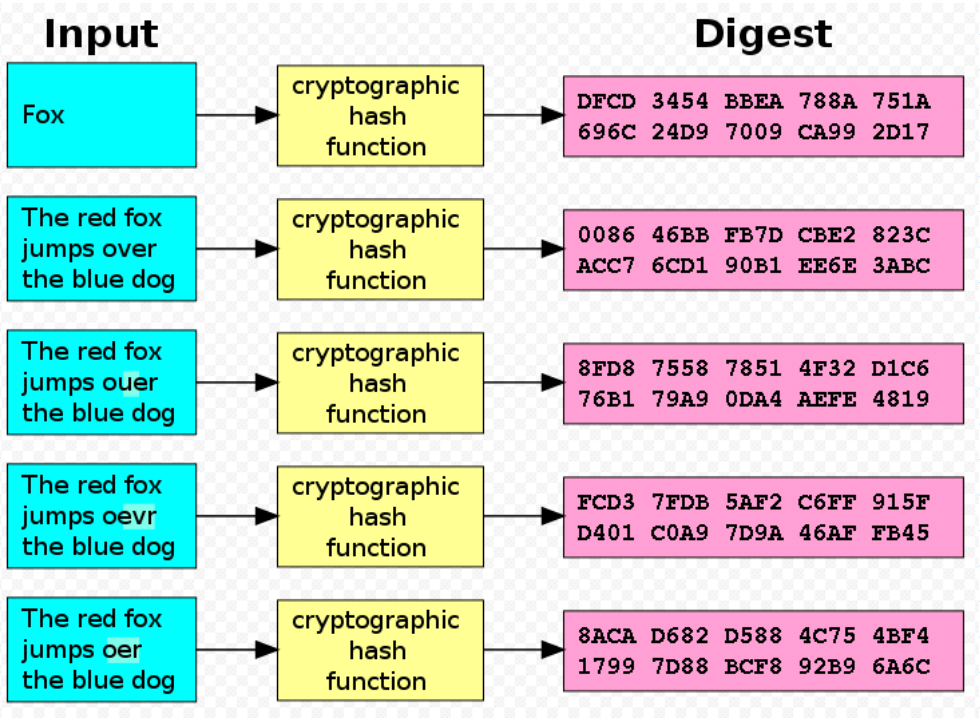
>> hTx_2=h28('Tx2:In21=5000||Out21=3500||Out22=1500||Rec1=A2||Rec2=B')
>> hTx_2=h28(Tx_2)
hTx_2 = D5C895A

>> hTx_3=h28('Tx3:In31=3500||Out31=3500||Out32=0||Rec1=E||Rec2=A2')
>> hTx_3=h28(Tx_3)

hTx_3 = FEC59B7

State tranasition diagramm

H-Functions. Merkle authentication tree



SHA-1 : 160 bits
SHA-256 bits
64 hex
Octave 6.3.0

h28('...') - 7 hex
hd28('...') - dec

h24
hd24 hd26 hd28
sha256 AES128

Merkle_Tree

Handbook of Applied Cryptography by A. Menezes, P. van Oorschot and S. Vanstone

Binary trees

A *binary tree* is a structure consisting of vertices and directed edges. The vertices are divided into three types:

1. a *root vertex*. The root has two edges directed towards it, a left and a right edge.
2. *internal vertices*. Each internal vertex has three edges incident to it – an upper edge directed away from it, and left and right edges directed towards it.
3. *leaves*. Each leaf vertex has one edge incident to it, and directed away from it.

The vertices incident with the left and right edges of an internal vertex (or the root) are called the *children* of the internal vertex. The internal (or root) vertex is called the *parent* of the associated children. Figure 13.5 illustrates a binary tree with 7 vertices and 6 edges.

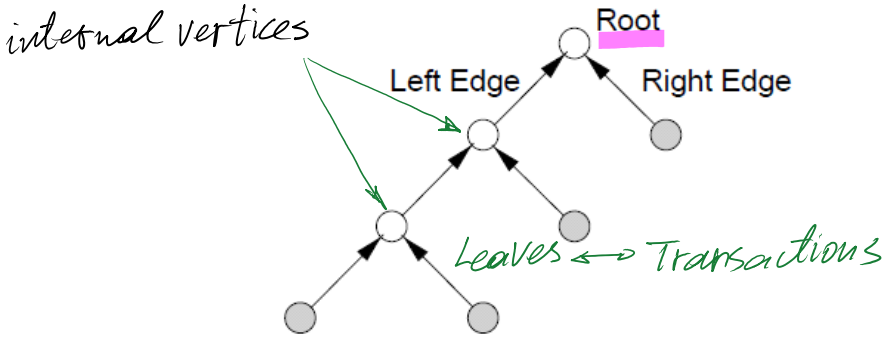


Figure 13.5: *A binary tree (with 4 shaded leaves and 3 internal vertices).*

Constructing and using authentication trees

Consider a binary tree T which has t leaves. Let h be a collision-resistant hash function. T can be used to authenticate t public values, $Y_1, Y_2, \ldots, Y_t$, by constructing an *authentication tree* T^* as follows.

1. Label each of the t leaves by a unique public value Y_i.
2. On the edge directed away from the leaf labeled Y_i, put the label $h(Y_i)$.
3. If the left and right edge of an internal vertex are labeled h_1 and h_2, respectively, label the upper edge of the vertex $h(h_1 \| h_2)$.
4. If the edges directed toward the root vertex are labeled u_1 and u_2, label the root vertex $h(u_1 \| u_2)$.

Once the public values are assigned to leaves of the binary tree, such a labeling is well-defined. Figure 13.6 illustrates an authentication tree with 4 leaves. Assuming some means to authenticate the label on the root vertex, an authentication tree provides a means to authenticate any of the t public leaf values Y_i, as follows. For each public value Y_i, there is a unique path (the *authentication path*) from Y_i to the root. Each edge on the path is a left or right edge of an internal vertex or the root. If e is such an edge directed towards vertex x, record the label on the other edge (not e) directed toward x. This sequence of labels (the *authentication path values*) used in the correct order provides the authentication of Y_i, as illustrated by Example 13.17. Note that if a single leaf value (e.g., Y_1) is altered, maliciously or otherwise, then authentication of that value will fail.

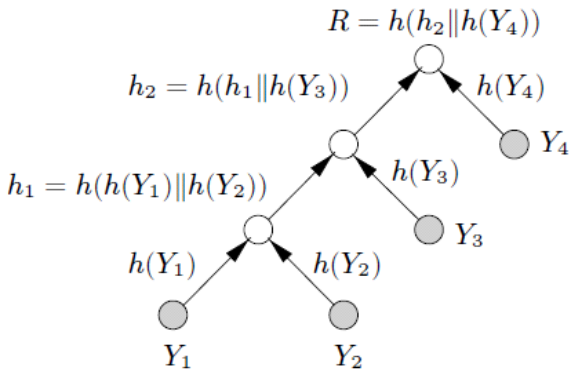


Figure 13.6: *An authentication tree.*

13.17 Example (*key verification using authentication trees*) Refer to Figure 13.6. The public value Y_1 can be authenticated by providing the sequence of labels $h(Y_2)$, $h(Y_3)$, $h(Y_4)$. The authentication proceeds as follows: compute $h(Y_1)$; next compute $h_1 = h(h(Y_1)) \| h(Y_2))$; then compute $h_2 = h(h_1 \| h(Y_3))$; finally, accept Y_1 as authentic if $h(h_2 \| h(Y_4)) = R$, where the root value R is known to be authentic. □

The advantage of authentication trees is evident by considering the storage required to allow authentication of t public values using the following (very simple) alternate approach: an entity A authenticates t public values $Y_1, Y_2, \ldots, Y_t$ by registering each with a trusted third party. This approach requires registration of t public values, which may raise storage issues at the third party when t is large. In contrast, an authentication tree requires only a single value be registered with the third party.

If a public key Y_i of an entity A is the value corresponding to a leaf in an authentication tree, and A wishes to provide B with information allowing B to verify the authenticity of Y_i, then A must (store and) provide to B both Y_i and all hash values associated with the authentication path from Y_i to the root; in addition, B must have prior knowledge and trust in the authenticity of the root value R. These values collectively guarantee authenticity, analogous to the signature on a public-key certificate. The number of values each party must store (and provide to others to allow verification of its public key) is $\lg(t)$, as per Fact 13.19.

13.18 Fact (*depth of a binary tree*) Consider the length of (or number of edges in) the path from each leaf to the root in a binary tree. The length of the longest such path is minimized when the tree is *balanced*, i.e., when the tree is constructed such that all such paths differ in length by at most one. The length of the path from a leaf to the root in a balanced binary tree containing t leaves is about $\lg(t)$.

13.19 Fact (*length of authentication paths*) Using a balanced binary tree (Fact 13.18) as an authentication tree with t public values as leaves, authenticating a public value therein may be achieved by hashing $\lg(t)$ values along the path to the root.

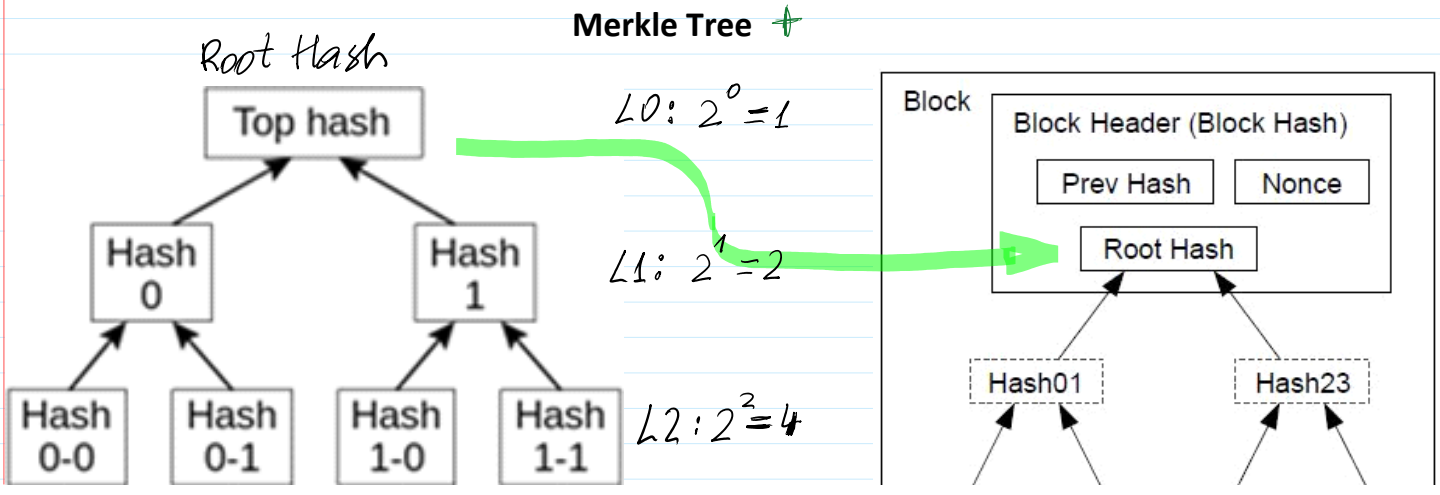
13.20 Remark (*time-space tradeoff*) Authentication trees require only a single value (the root value) in a tree be registered as authentic, but verification of the authenticity of any particular leaf value requires access to and hashing of all values along the authentication path from leaf to root.

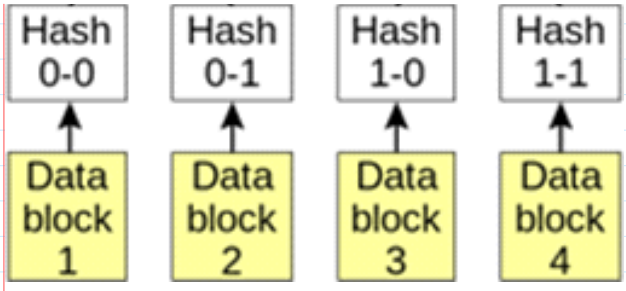
13.21 Remark (*changing leaf values*) To change a public (leaf) value or add more values to an authentication tree requires recomputation of the label on the root vertex. For large balanced trees, this may involve a substantial computation. In all cases, re-establishing trust of all users in this new root value (i.e., its authenticity) is necessary.

The computational cost involved in adding more values to a tree (Remark 13.21) may motivate constructing the new tree as an unbalanced tree with the new leaf value (or a subtree of such values) being the right child of the root, and the old tree, the left. Another motivation for allowing unbalanced trees arises when some leaf values are referenced far more frequently than others.

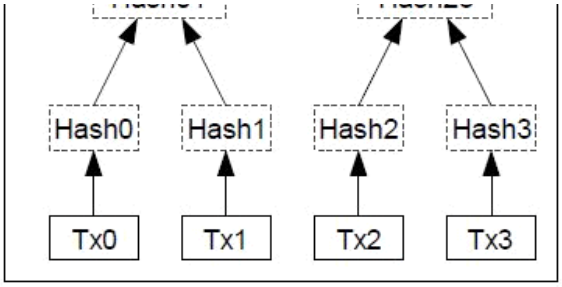
Bitcoin transactions are permanently recorded in the network through files called blocks. Maximum size of the block is currently limited to 1 MB but it may be increased in the future. Each block contains a UNIX time timestamp, which is used in block validity checks to make it more difficult for adversary to manipulate the block chain. New blocks are added to the end of the record (block chain) by referencing the hash of the previous block and once added are never changed. A variable number of transactions is included into a block through the merkle tree (fig 3.). Transactions in the Merkle tree are hashed using double SHA256 (hash of the hash of the transaction message).

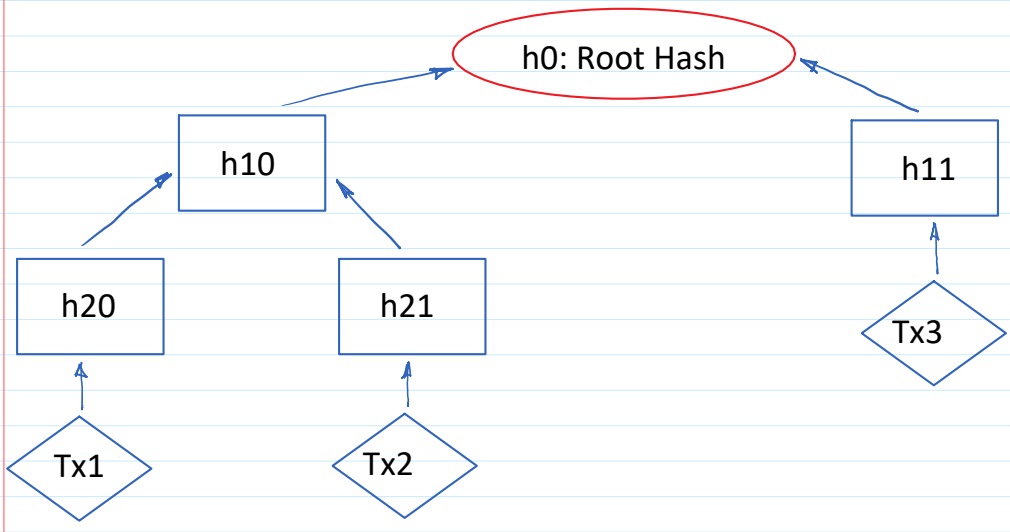
Transactions are included into the block's hash indirectly through the merkle root (top hash of a merkle tree). This allows removing old transactions (fig. 4) without modifying the hash of the block. Once the latest transaction is buried under enough blocks, previous transactions serve only as a history of the ownership and can be discarded to save space.





$L2 : 2^2 = 4$





L0

L1

L2

$2^{12} = 4096$

>> h20=h28(hTx_1)
h20 = 5B5412B
>> h21=h28(hTx_2)
h21 = D5C895A
>> h11=h28(hTx_3)
h11 = FEC59B7

>> h10=h28('5B5412B||D5C895A')
h10 = B6BD3A4

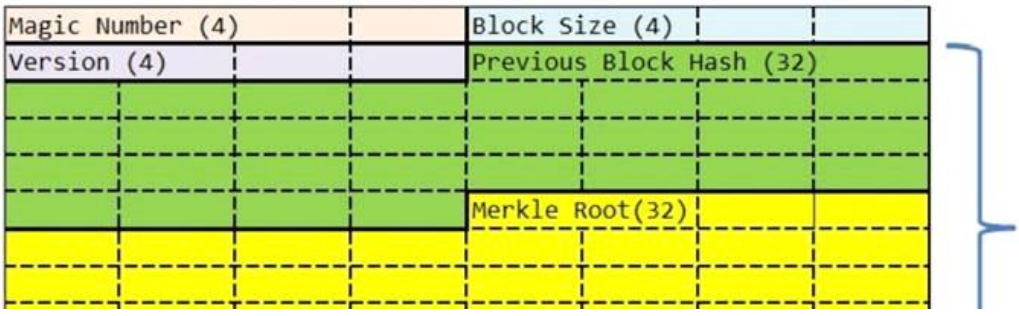
Root Hash: h0

>> h0=h28('B6BD3A4||FEC59B7')
h0 = **60BA3B5**

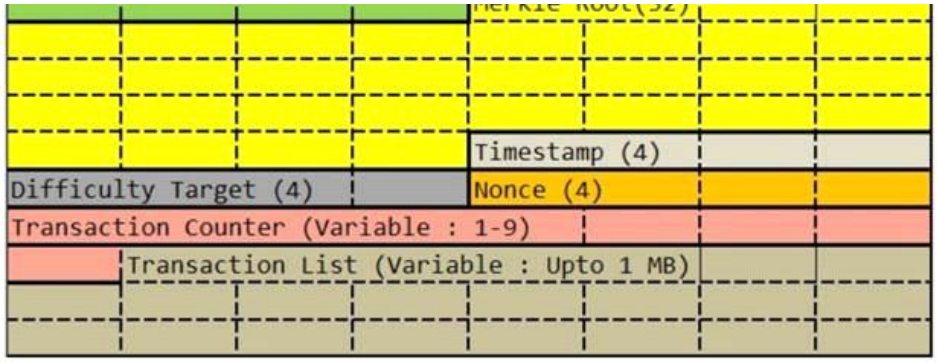
Python : sha256

h20: 5B5412B
h21: D5C895A
h11: FEC59B7

h10: 625A41F
h0: **60BA3B5**



Block size = 4 Bytes
4 Bytes × 8 bits = 32 bits
Block can have
2^{32} bits 4294967296
In ASCII encoding



$2^{32} = 4294967296$

In ASCII encoding 8 bits represents 1 symbol a, b, c, ... #_

Block represents 536 870 912 symbols

Difficulty Target (DT): defines the complexity of block mining. In our simulation DT we will choose to find h-value of mining (mined block) having only 1 leading hexadecimal digit.

Transactions:
Tx_1 = 'Tx1:In11=6000||In12=3000||Out11=5000||Out12=4000||Rec1=B||Rec2=A'
Tx_2='Tx2:In21=5000||Out21=3500||Out22=1500||Rec1=A2||Rec2=B'
Tx_3='Tx3:In31=3500||Out31=3500||Out32=0||Rec1=E||Rec2=A2'

>> hTx_1=h28(Tx_1)
hTx_1 = 5B5412B

>> hTx_2=h28(Tx_2)
hTx_2 = D5C895A

>> hTx_3=h28(Tx_3)
hTx_3 = FEC59B7

hPrBl	hRoot	Bl_N:hPrBl=0CAF06F||hRoot=2CC219F||hTx_N1=AFC73D8||hTx_N2=13251F8||hTx_N3=5B5412B||Nonce=1000	hBl_N	Nonce	hBl_N_Mined
0CAF06F	2CC219F	**Bl_1:hPrBl=0CAF06F||hRoot=2CC219F||hTx_1=AFC73D8||hTx_2=13251F8||hTx_3=5B5412B||Nonce=1000**		1021	**06F61B0**

$$Pr\{M: DT = 0xxxxxx\}, \quad x \in \{0_h, 1_h, \ldots, 15_h\} = \frac{\text{Adequate number of variants}}{\text{Total number of variants}} = \frac{ANV}{TNV}$$

$\Pr\{M: DT = 0xxxxxx\} = \frac{\text{Total number of variants}}{TNV}$

$x \in \{0_h, 1_h, \ldots, 15_h\}$

$\{0000_b, 0001_b, \ldots, 1111_b\} \Rightarrow [0, 2^4 - 1]$

6 hex numbers $xxxxxx \in [0, 2^{24} - 1] \Rightarrow ANV = 2^{24}$

7 hex number $yxxxxxx \in [0, 2^{28} - 1] \Rightarrow TNV = 2^{28}$

$\Pr\{M: DT = 0xxxxxx\} = \frac{2^{24}}{2^{28}} = \frac{1}{2^4} = \frac{1}{16}$

Average probability to mine a block is 1/16 $\Rightarrow$ In average we must perform 16 computations to mine a block.

Till this place

Tx1:In11=6000||In12=3000||Out11=5000||Out12=4000||Rec1=2||Rec2=1

	N	Id	PuK_N	InN1	InN2	OutN1	OutN2	RecN1	RecN2
	123	S.Eligijus		6000	3000	5000	4000	2	1
	234	M.Gabriele							
	345	Ausrys							

>> Tx_1='Tx1:In11=6000||In12=3000||Out11=5000||Out12=4000||Rec1=B||Rec2=A'
Tx_1 = Tx1:In11=6000||In12=3000||Out11=5000||Out12=4000||Rec1=B||Rec2=A
>> hTx_1=h28(Tx_1)
hTx_1 = AFC73D8

TxN:InN1=…||InN2=…||OutN1=…||OutN2=…||RecN1=…||RecN2=…	hTx_N	N1	N2	N3	H(TxN1) hTx_N1	H(TxN2) hTx_N2	H(TxN3) hTx_N3
Tx1:In11=6000||In12=3000||Out11=5000||Out12=4000||Rec1=B||Rec2=A	AFC73D8	1	2	3	AFC73D8	13251F8	5B5412B
Tx2:In21=5000||Out21=3500||Out22=1500||Rec1=A2||Rec2=B	13251F8						
Tx3:In31=3500||Out31=3500||Out32=0||Rec1=E||Rec2=A2	5B5412B						

changed

hPrBl	hRoot	Bl_N:hPrBl=0CAF06F||hRoot=2CC219F||hTx_N1=AFC73D8||hTx_N2=13251F8||hTx_N3=5B5412B||Nonce=1000	hBl_N	Nonce	hBl_N_Mined
0CAF06F	2CC219F	**Bl_1:hPrBl=0CAF06F||hRoot=2CC219F||hTx_1=AFC73D8||hTx_2=13251F8||hTx_3=5B5412B||Nonce=1000**		1021	**06F61B0**

Bl_1:hPrBl=0CAF06F||hRoot=2CC219F||hTx_1=AFC73D8||hTx_2=13251F8||hTx_3=5B5412B||Nonce=1000

>> Bl_1Mng='Bl_1:hPrBl=0CAF06F||hRoot=2CC219F||hTx_1=AFC73D8||hTx_2=13251F8||hTx_3=5B5412B||Nonce=1000'
Bl_1Mng = Bl_1:hPrBl=0CAF06F||hRoot=2CC219F||hTx_1=AFC73D8||hTx_2=13251F8||hTx_3=5B541
2B||Nonce=1000

For mining h – value of Bl_1Mng must be computed

For mining h - value of Bl_1Mng must be computed according to Difficulty Target (DT)

>> hBl_1Mining=h28(Bl_1Mining)
hBl_1Mining = 2520EB3
>> hBl_1Mining=h28(Bl_1Mining)
hBl_1Mining = D4FB37A

>> hBl_1Mng=h28('Bl_1:hPrBl=0CAF06F||hRoot=2CC219F||hTx_1=AFC73D8||hTx_2=13251F8||hTx_3=5B5412B||Nonce=1000')
hBl_1Mng = 2520EB3

>> hBl_1Mng=h28('Bl_1:hPrBl=0CAF06F||hRoot=2CC219F||hTx_1=AFC73D8||hTx_2=13251F8||hTx_3=5B5412B||Nonce=1001')
hBl_1Mng = D4FB37A

>> hBl_1Mng=h28('Bl_1:hPrBl=0CAF06F||hRoot=2CC219F||hTx_1=AFC73D8||hTx_2=13251F8||hTx_3=5B5412B||Nonce=1002')
hBl_1Mng = 6EC93FC

>> hBl_1Mng=h28('Bl_1:hPrBl=0CAF06F||hRoot=2CC219F||hTx_1=AFC73D8||hTx_2=13251F8||hTx_3=5B5412B||Nonce=1003')
hBl_1Mng = 98A6656

After 22 trials when Nonce=1021 the block is mined:
>> hBl_1Mng=h28('Bl_1:hPrBl=0CAF06F||hRoot=2CC219F||hTx_1=AFC73D8||hTx_2=13251F8||hTx_3=5B5412B||Nonce=1021')
hBl_1Mng = 06F61B0
After 22 trials when Nonce=1021 the block is mined.

Till this place

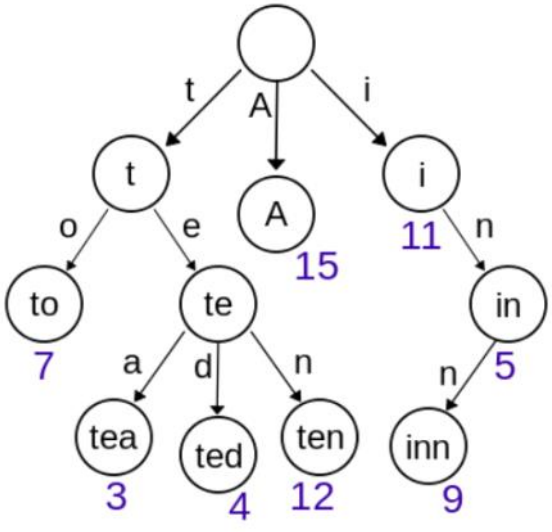
https://medium.com/codechain/modified-merkle-patricia-trie-how-ethereum-saves-a-state-e6d7555078dd

Trie ← Retrieve

Modified Merkle Patricia Trie (a.k.a MPT) as the method to save Ethereum state.
Basically, MPT is a combination of Patricia trie and Merkle tree, with few additional optimizations that fit the characteristics of Ethereum. Thus, an understanding of the Patricia trie and Merkle tree should precede the understanding of MPT.

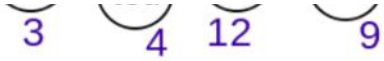
Patricia Trie

Patricia trie is a data structure which is also called Prefix tree, radix tree or trie. Trie uses a key as a path so the nodes that share the same prefix can also share the same path. This structure is fastest at finding common prefixes, simple to implement, and requires small memory. Thereby, it is commonly used for implementing routing tables, systems that



Example of Patricia Trie

implement, and requires small memory. Thereby, it is commonly used for implementing routing tables, systems that are used in low specification machines like the router.



Example of Patricia Trie