Making Utreexo

Journey to the current Utreexo design

https://github.com/kcalvinalvin/2024-praha-devhackday

Utreexo: A dynamic hash-based accumulator optimized for the Bitcoin UTXO set

Thaddeus Dryja
tdryja@media.mit.edu

MIT Digital Currency Initiative

Abstract

In the Bitcoin consensus network, all nodes come to agreement on the set of Unspent Transaction Outputs (The "UTXO" set). The size of this shared state is a scalability constraint for the network, as the size of the set expands as more users join the system, increasing resource requirements of all nodes. Decoupling the network's state size from the storage requirements of individual machines would reduce hardware requirements of validating nodes. We introduce a hash based accumulator to locally represent the UTXO set, which is logarithmic in the size of the full set. Nodes attach and propagate inclusion proofs to the inputs of transactions, which along with the accumulator state, give all the information needed to validate a transaction. While the size of the inclusion proofs results in an increase in network traffic, these proofs can be discarded after verification, and aggregation methods can reduce their size to a manageable level of overhead. In our simulations of downloading Bitcoin's blockchain up to early 2019 with 500MB of RAM allocated for caching, the proofs only add approximately 25% to the amount otherwise downloaded.

https://eprint.iacr.org/2019/611.pdf

Utreexod beta release

494 views

Calvin Kim

to Bitcoin Development Mailing List

May 7, 2024, 12:40:46 PM

Hi all,

Utreexod, a full node implementation with Utreexo accumulator support, is now available for general testing by the public. Notable features are:

- Implementation of efficient Utreexo accumulators with an improved deletion algorithm from the Utreexo paper.
- Efficient P2P transaction relay with support for caching utreexo proofs for mempool transactions.
- Quick sync to the tip of the blockchain with AssumeUtreexo.
- Built in wallet support (with BDK wallet).
- Electrum personal server support for usage with other wallets.

It's important to note that the code and protocol is not peer reviewed. Using the node for production usage or for non-negligible amounts of Bitcoin is not recommended. There will also be breaking changes requiring frequent updates as well. Utreexod is also based off of btcd (https://github.com/btcsuite/btcd), a bitcoin node implementation in Go. Because of this, there may be consensus incompatibilities either as a result of a btcd bug or a Bitcoin Core bug.

Also note that there are not many Utreexo nodes on the network. It may take a while for the node to find a Utreexo node it can sync off of. To help bootstrap the network, consider running an archival node or a bridge node. Refer to the readme on the github (https://github.com/utreexo/utreexod) page for instructions on how to run a bridge node or an archival utreexo node.

Prebuilt static binaries for Linux, MacOS, and Windows are available at (https://github.com/utreexo/utreexod/releases/)

Huge thanks to HRF for funding my work for this release. Without the support I wouldn't have been able to focus on Utreexo.

Also thanks to all the folks that directly contributed to the release:

What's an accumulator?

- Compactly represent a set (UTXO set)

What's an accumulator?

- Compactly represent a set (UTXO set)
- Have proof of existence for each UTXO

Algorithm 1 AddOne

```
function AddOne(acc, leaf)                ▷ add a leaf to the accumulator
    n ← leaf                              ▷ n is initially the leaf to add
    h ← 0                                 ▷ height is initially 0
    r ← acc[h]                            ▷ r is the lowest root
    while r ≠ ∅ do                        ▷ loop until an empty root
        n ← Parent(r, n)
        acc[h] ← ∅
        h ← h + 1
        r ← acc[h]
    acc[h] ← n
    return acc
```

Efficient Asynchronous Accumulators for Distributed PKI

Leonid Reyzin and Sophia Yakoubov

Boston University
{reyzin, sonka}@bu.edu

Abstract. Cryptographic accumulators are a tool for compact set representation and secure set membership proofs. When an element is added to a set by means of an accumulator, a membership witness is generated. This witness can later be used to prove the membership of the element. Typically, the membership witness has to be synchronized with the accumulator value: it has to be updated every time another element is added to the accumulator, and it cannot be used with outdated accumulator values. However, in many distributed applications (such as blockchain-based public key infrastructures), requiring strict synchronization is prohibitive. We define *low update frequency*, which means that a witness only needs to be updated a small number of times, and *old-accumulator compatibility*, which means that a witness can be used with outdated accumulator values. Finally, we propose an accumulator that achieves both of those properties.

Keywords: cryptographic accumulators

https://eprint.iacr.org/2015/718.pdf

5.1.2 Chaining Value Stack

To help picture the role of the CV stack, Figure 6 shows a growing tree as chunk CVs are added incrementally. As just discussed above, chunk CVs are added to this tree only after the caller has supplied at least 1 byte for the following chunk, so we know that none of these chunks or parent nodes is the root of the tree, and we do not need to worry about the `ROOT` flag yet.

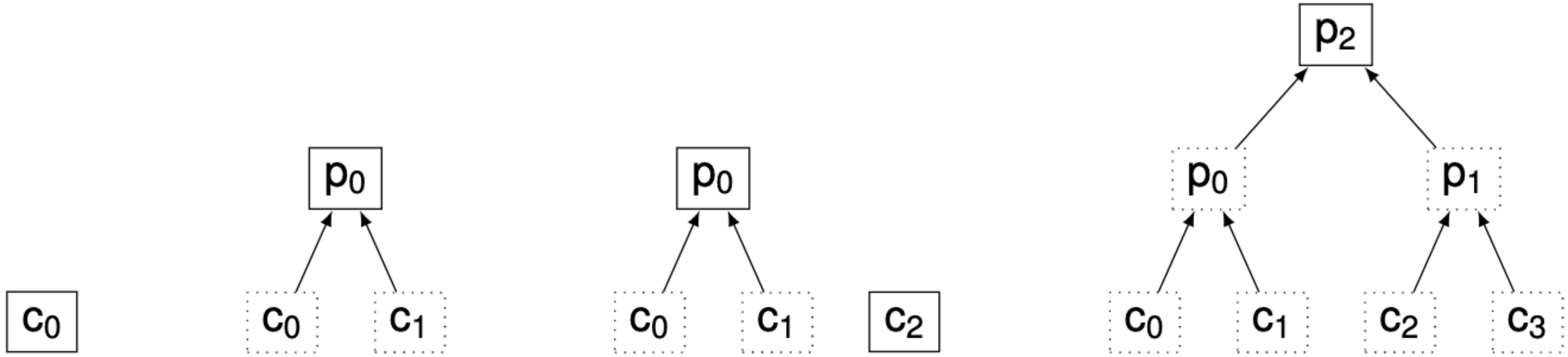
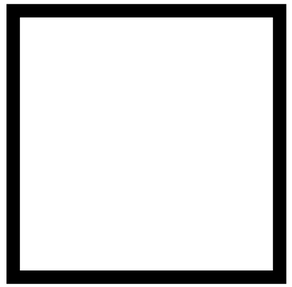


Figure 6: An incomplete tree growing incrementally from 1 to 4 chunks. Dotted boxes represent CVs that no longer need to be stored.

https://raw.githubusercontent.com/BLAKE3-team/BLAKE3-specs/master/blake3.pdf

1 UTXO

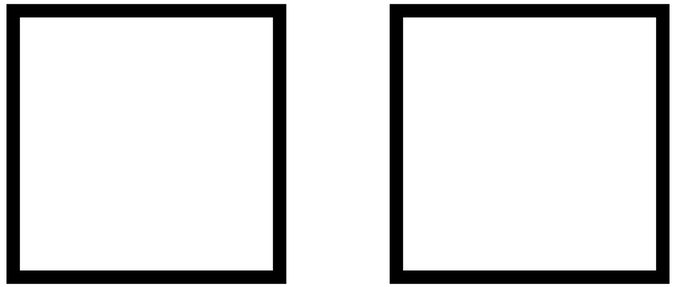
Single root



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

2 UTXOs

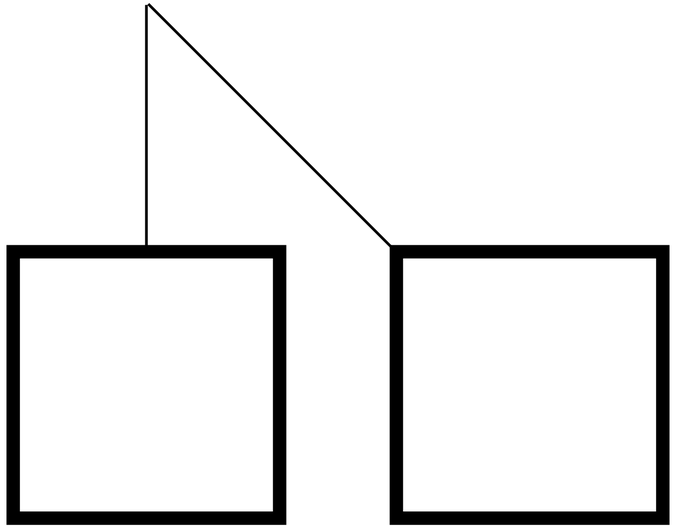
Add 1 more



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

2 UTXOs

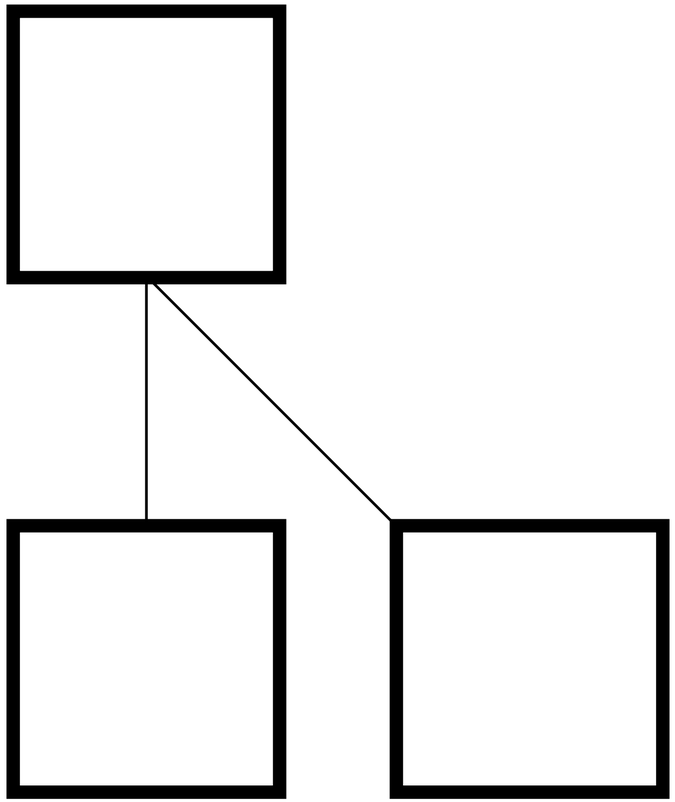
Concatenate



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

2 UTXOs

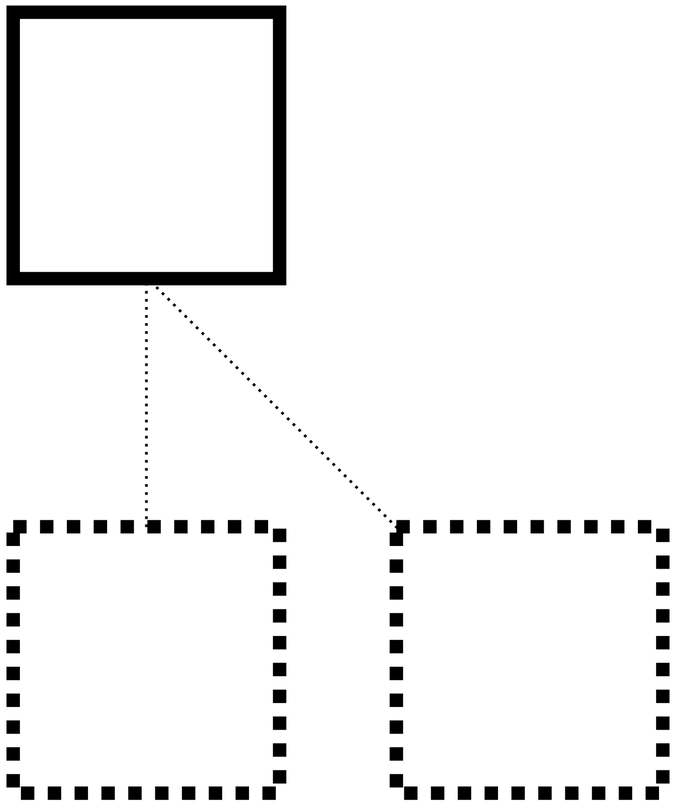
Hash to create a new root



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

2 UTXOs

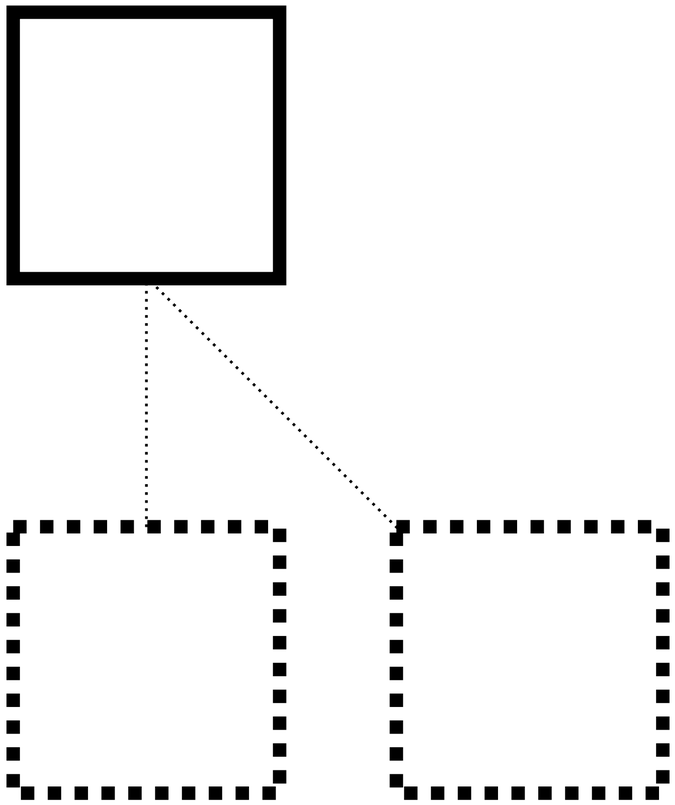
Can now delete the leaves



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

3 UTXOs

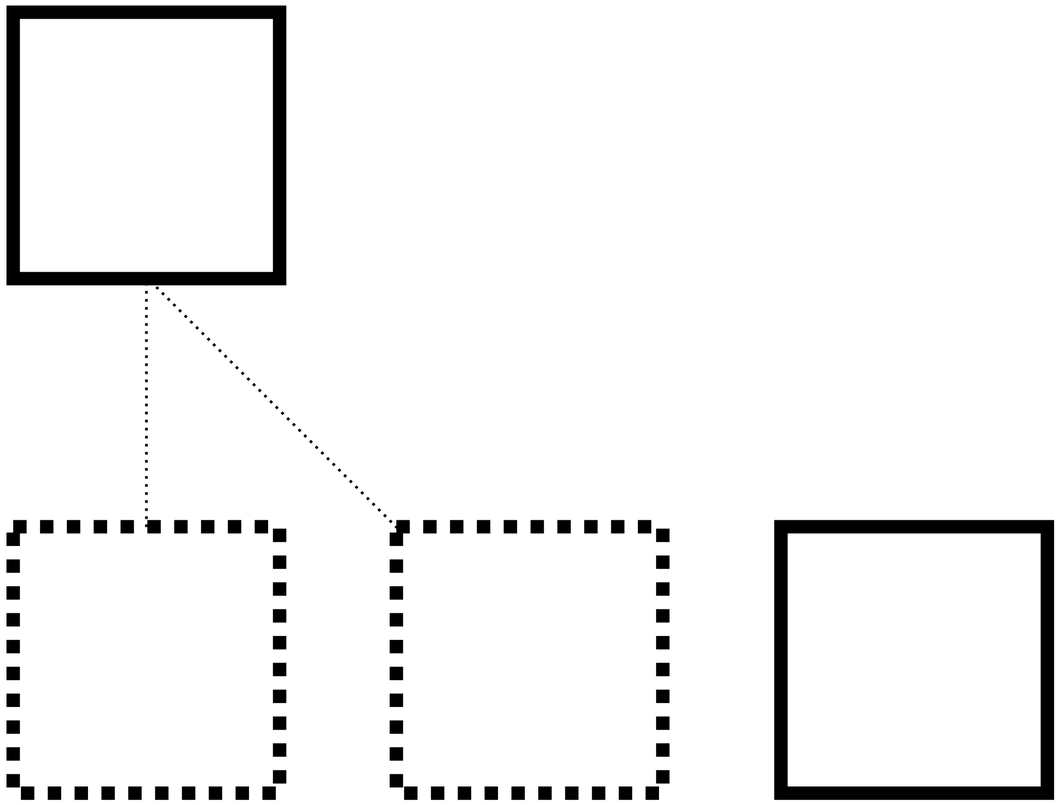
Add 1 more



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

3 UTXOs

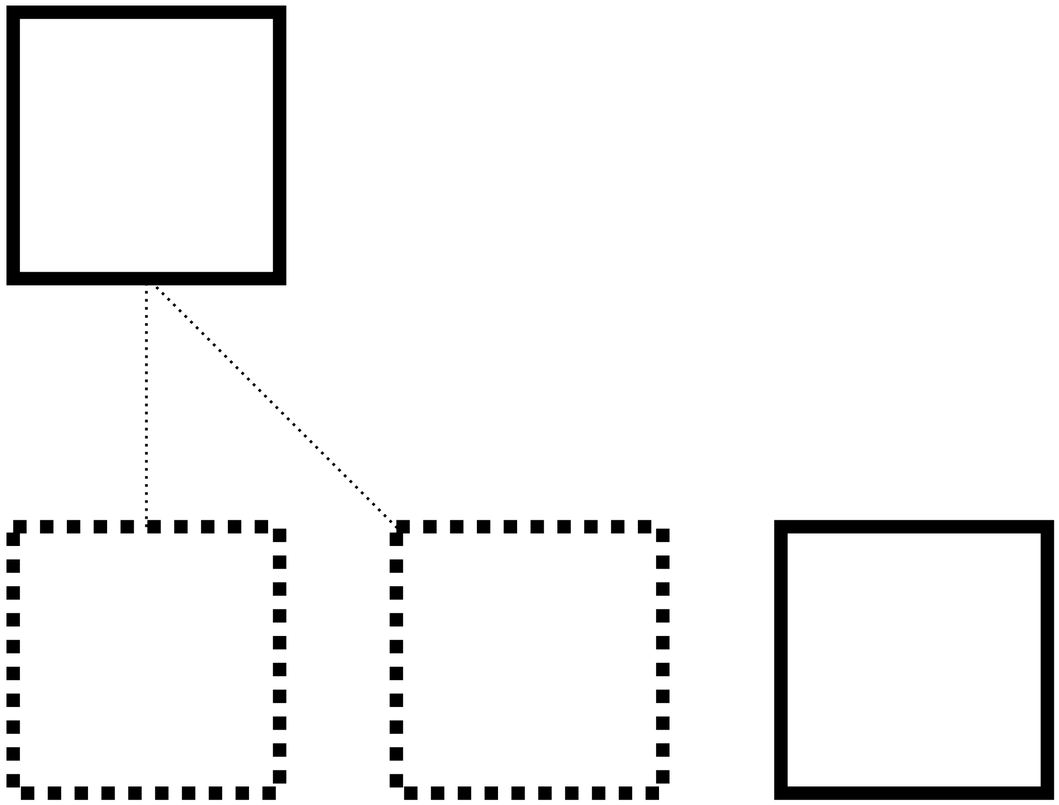
Add 1 more



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

3 UTXOs

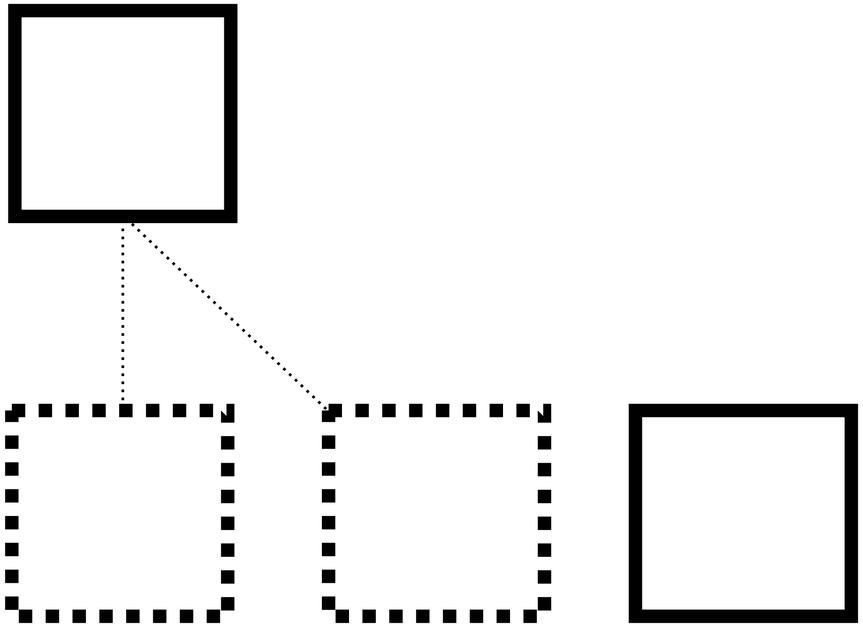
Nothing to hash with. Becomes a root



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

4 UTXOs

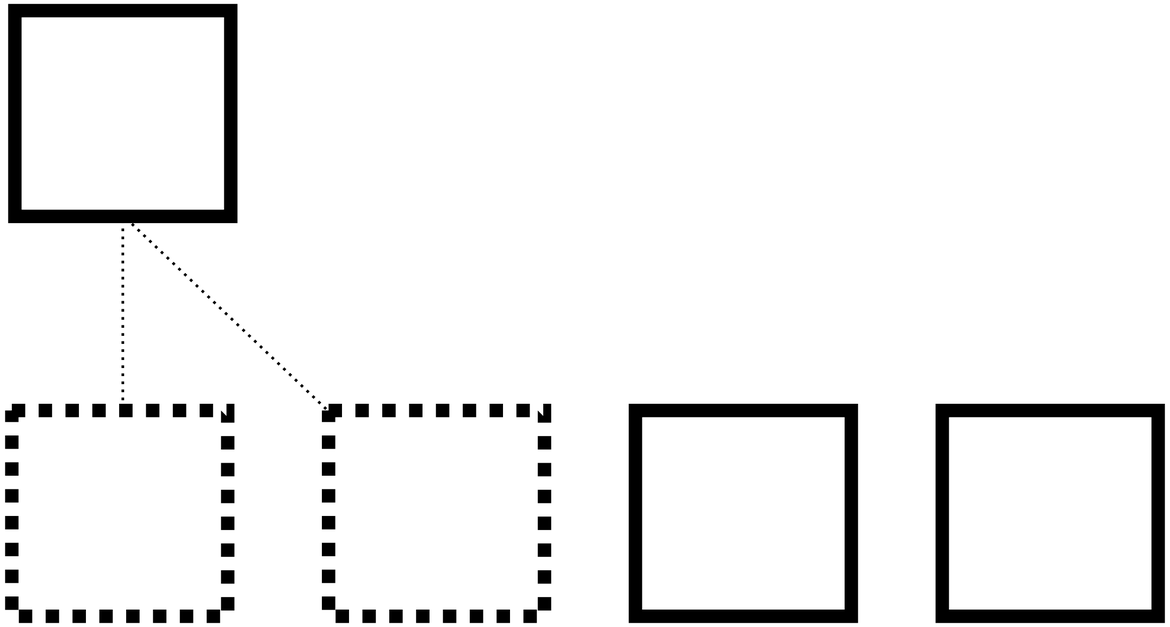
Another one



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

4 UTXOs

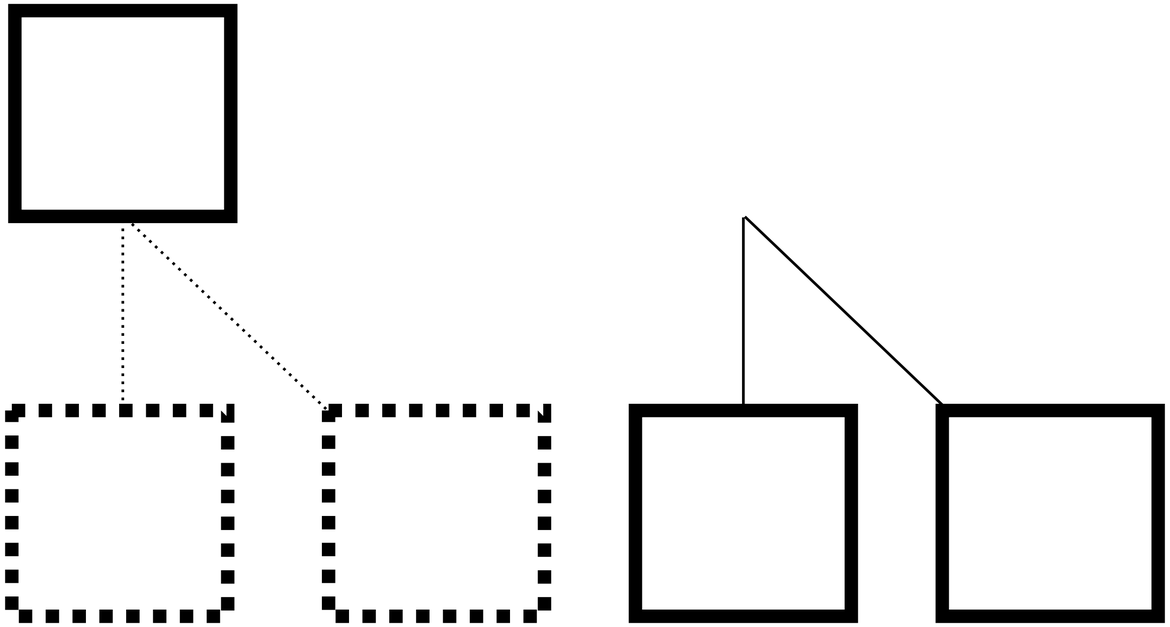
Another one



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

4 UTXOs

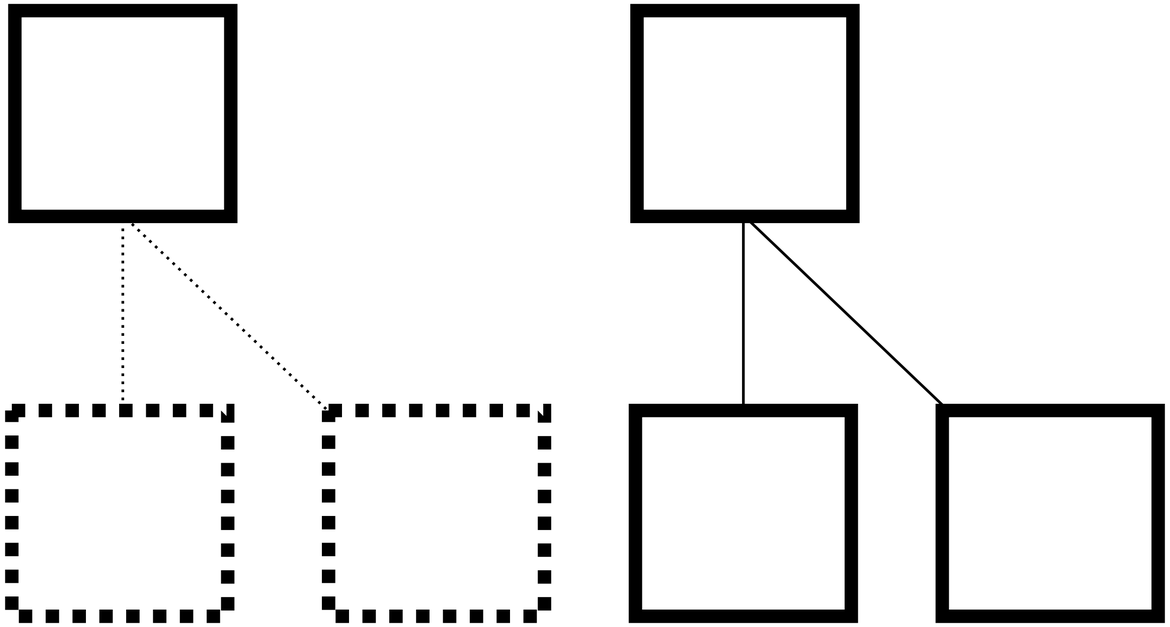
Concatenate



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

4 UTXOs

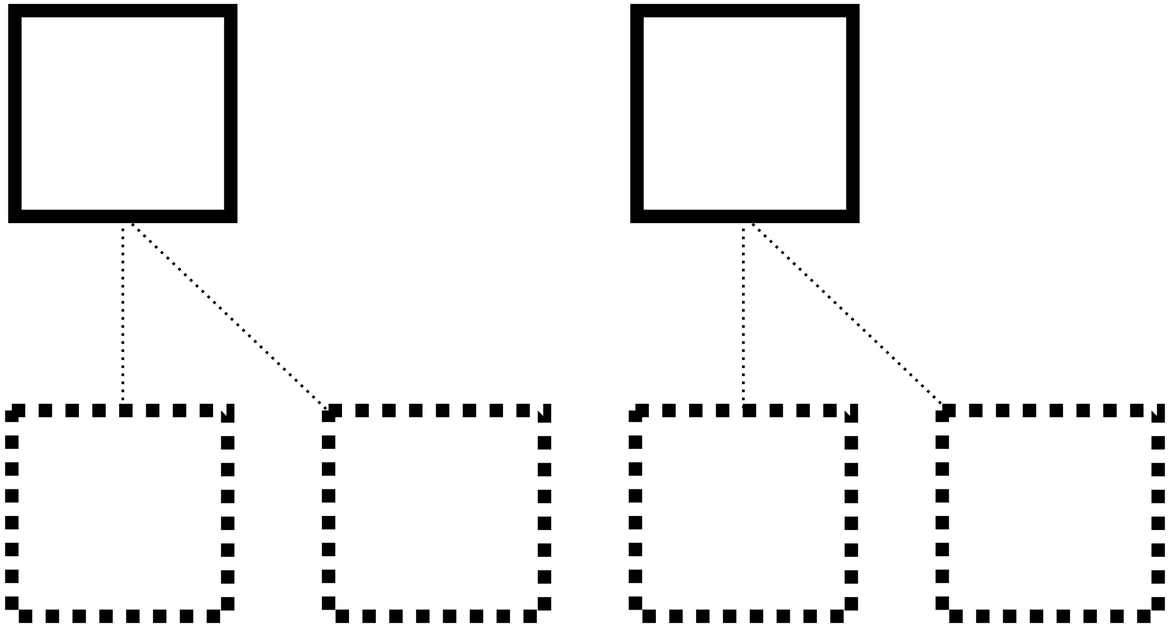
Hash



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

4 UTXOs

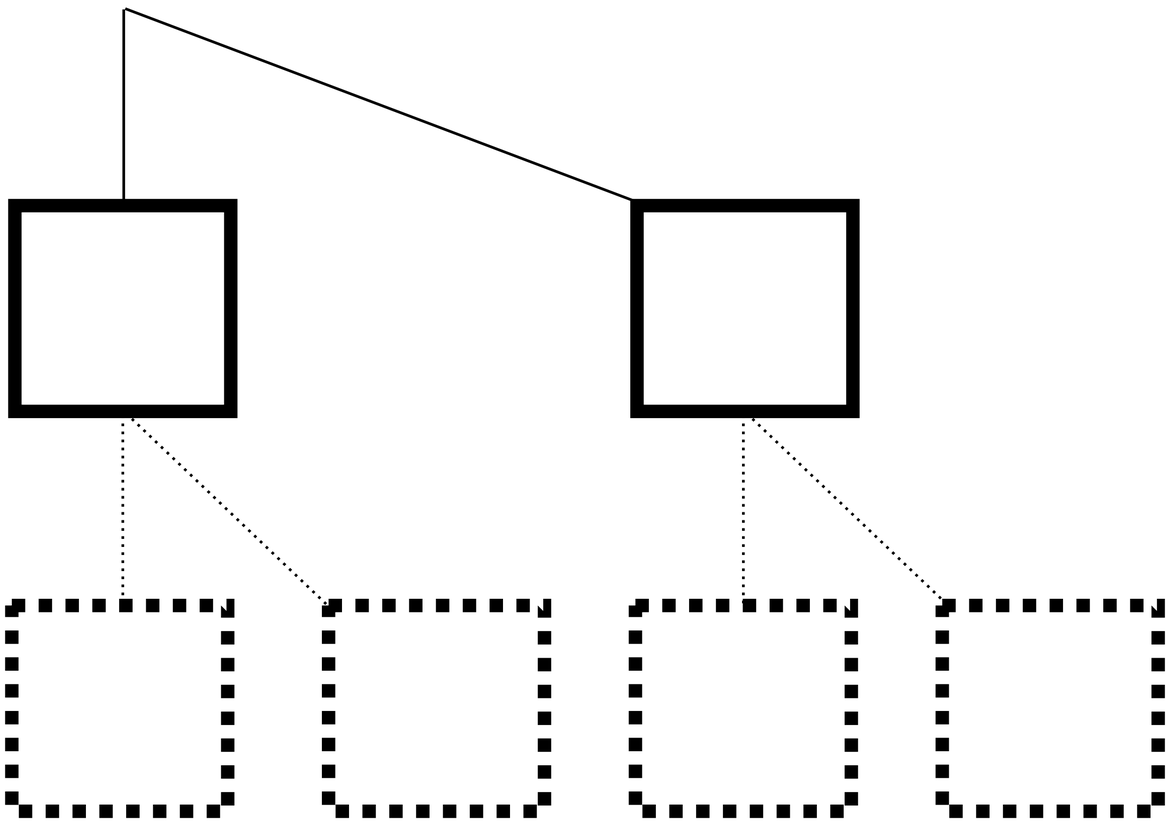
Can now throw away the leaves



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

4 UTXOs

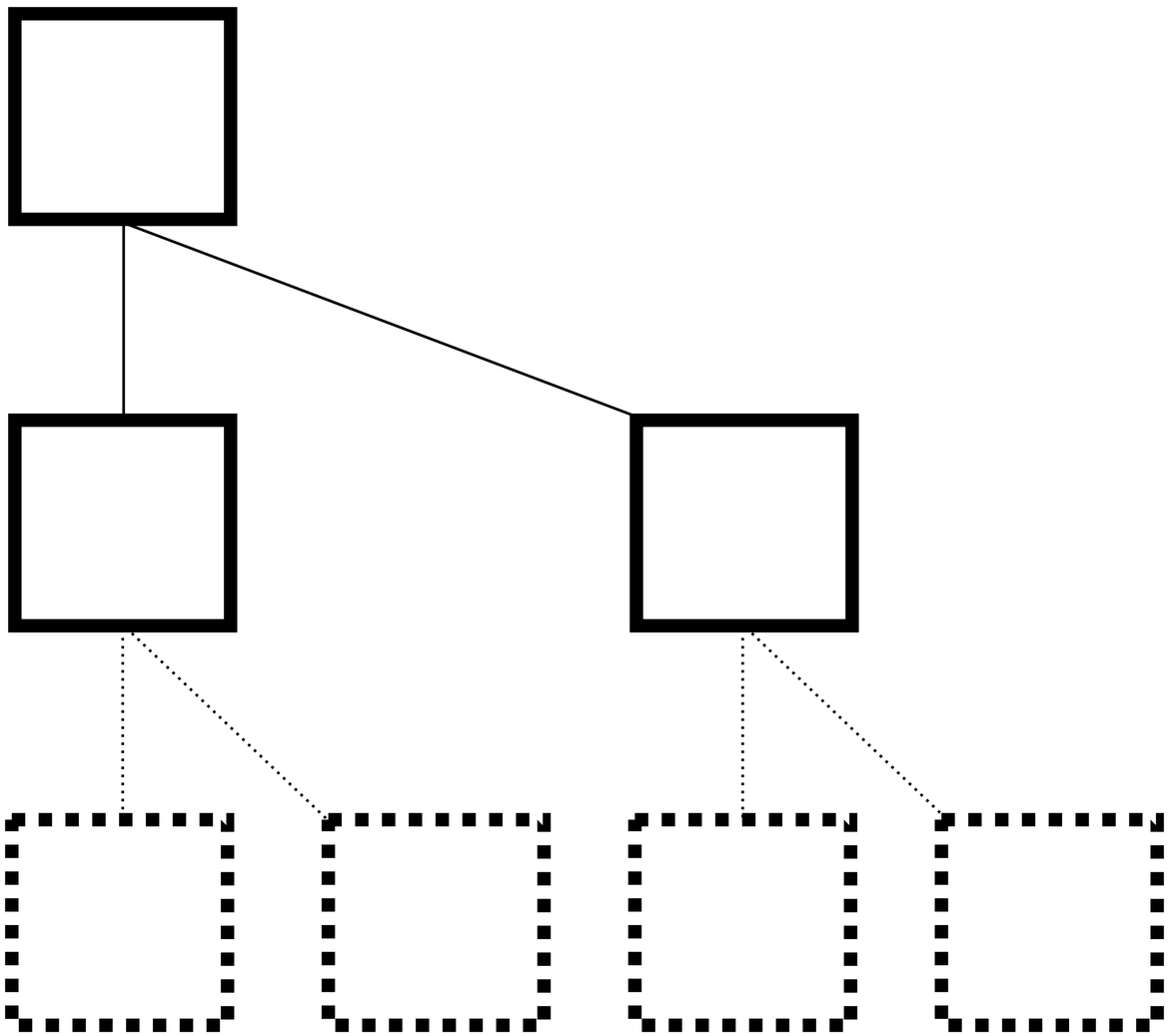
Concatenate



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

4 UTXOs

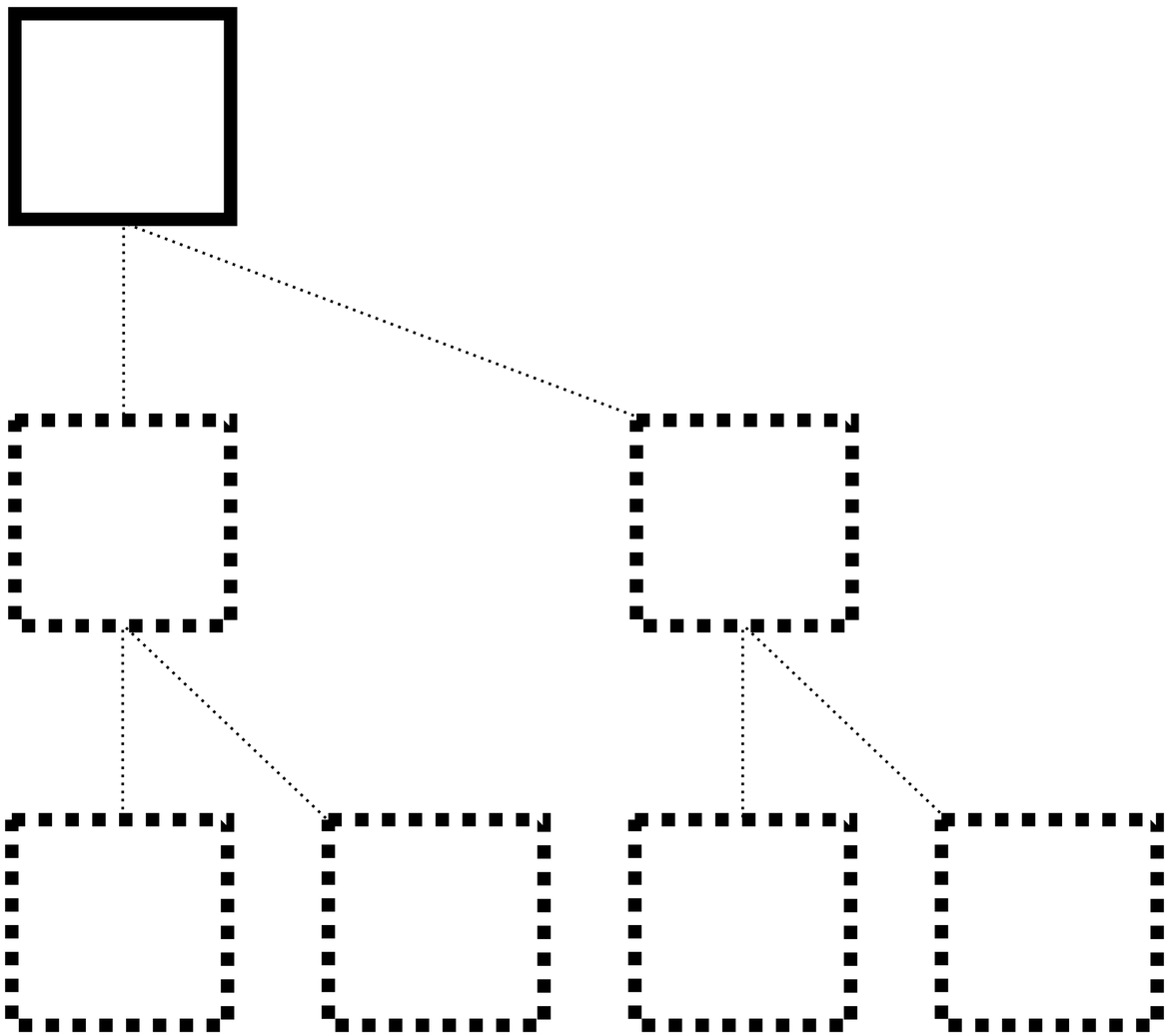
Hash



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

4 UTXOs

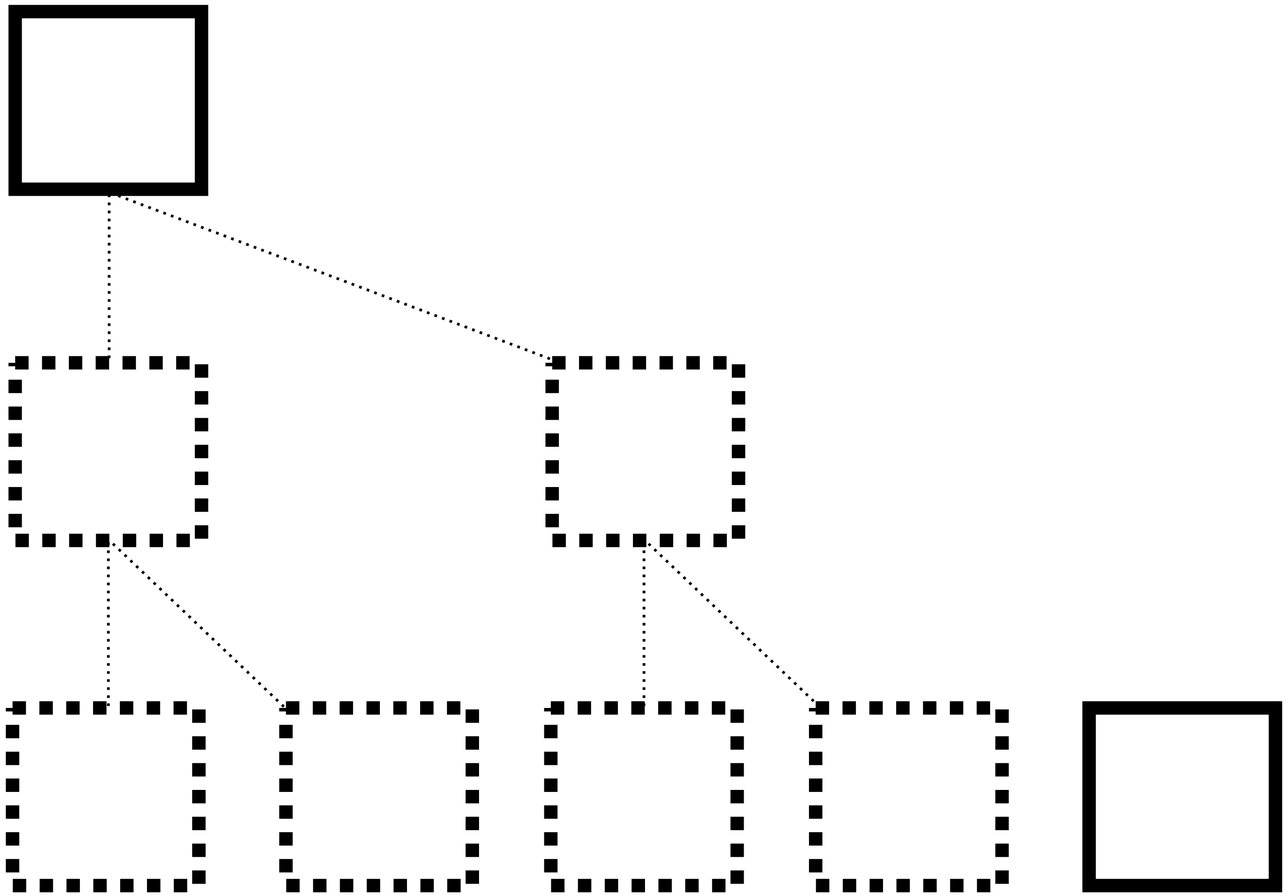
Throw away



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

5 UTXOs

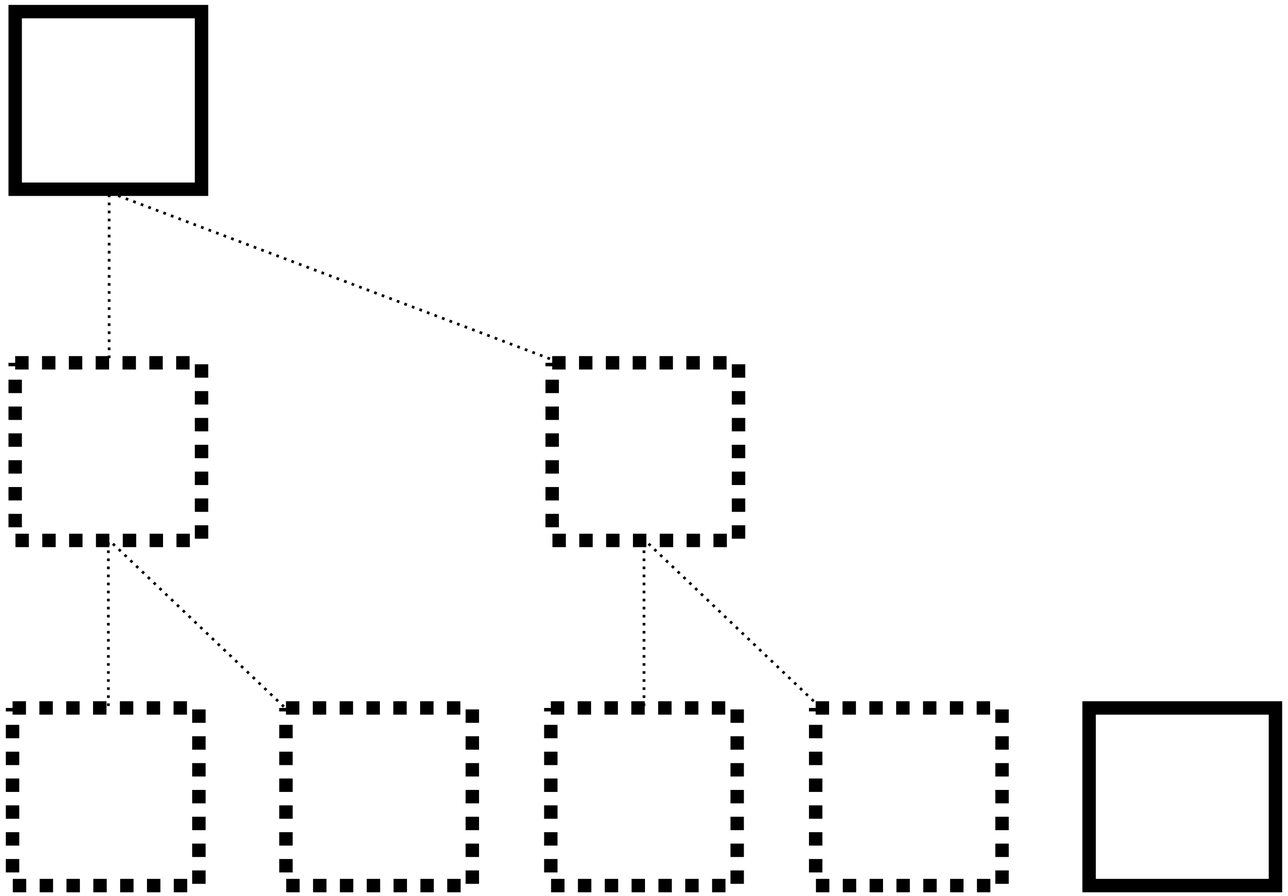
Add one more



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

5 UTXOs

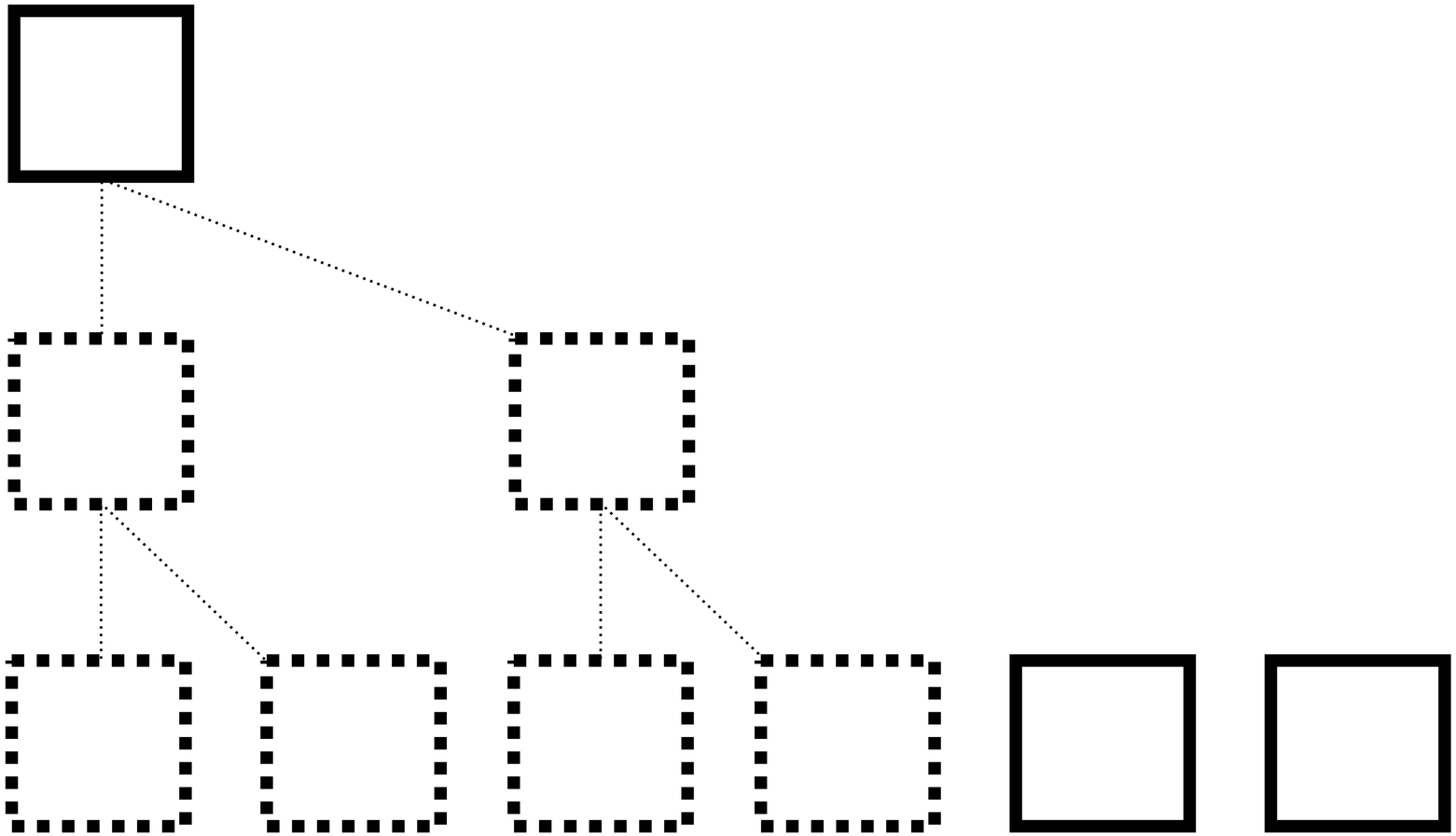
Nothing to hash with so it becomes a root



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

6 UTXOs

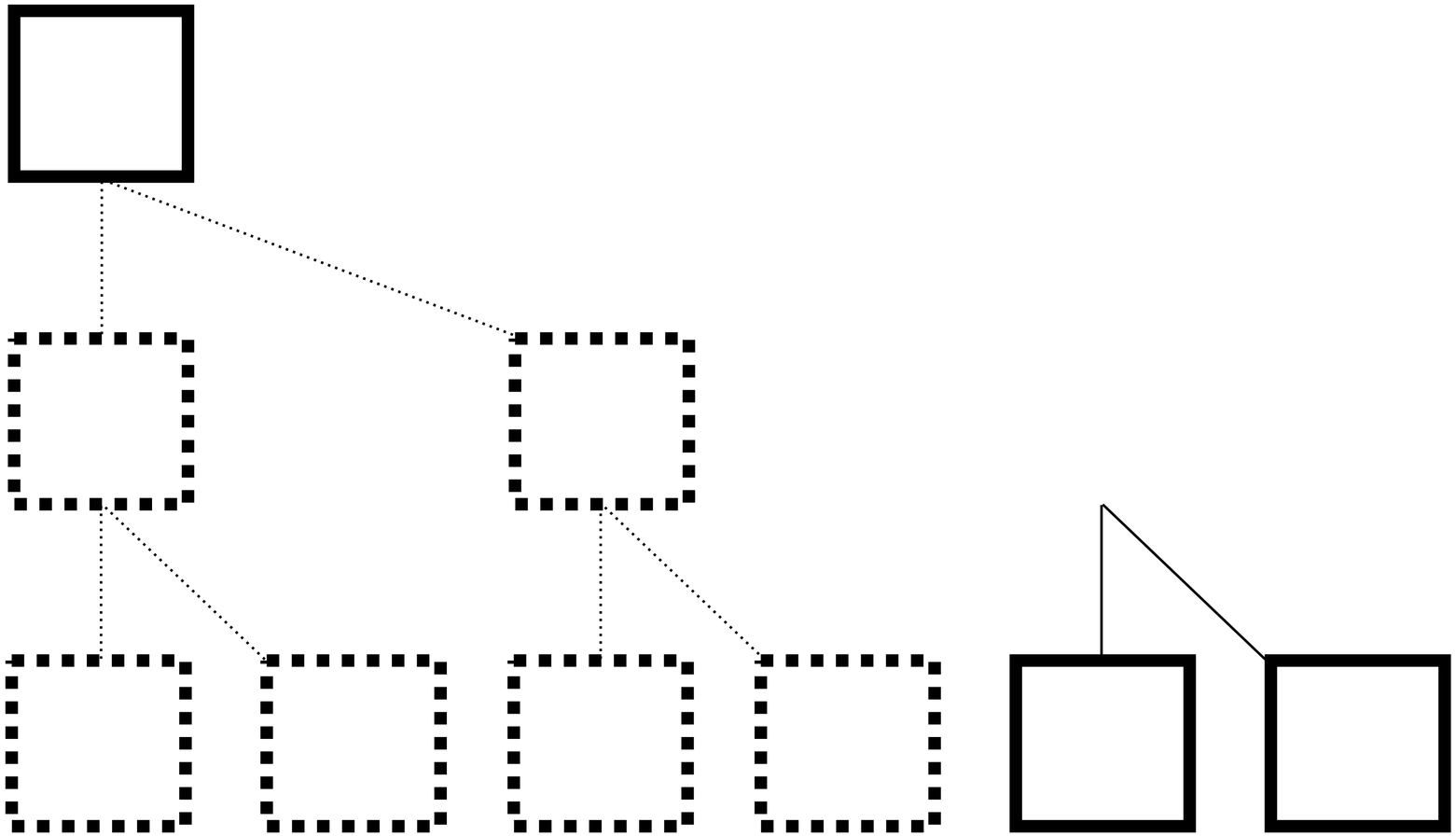
1 more



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

6 UTXOs

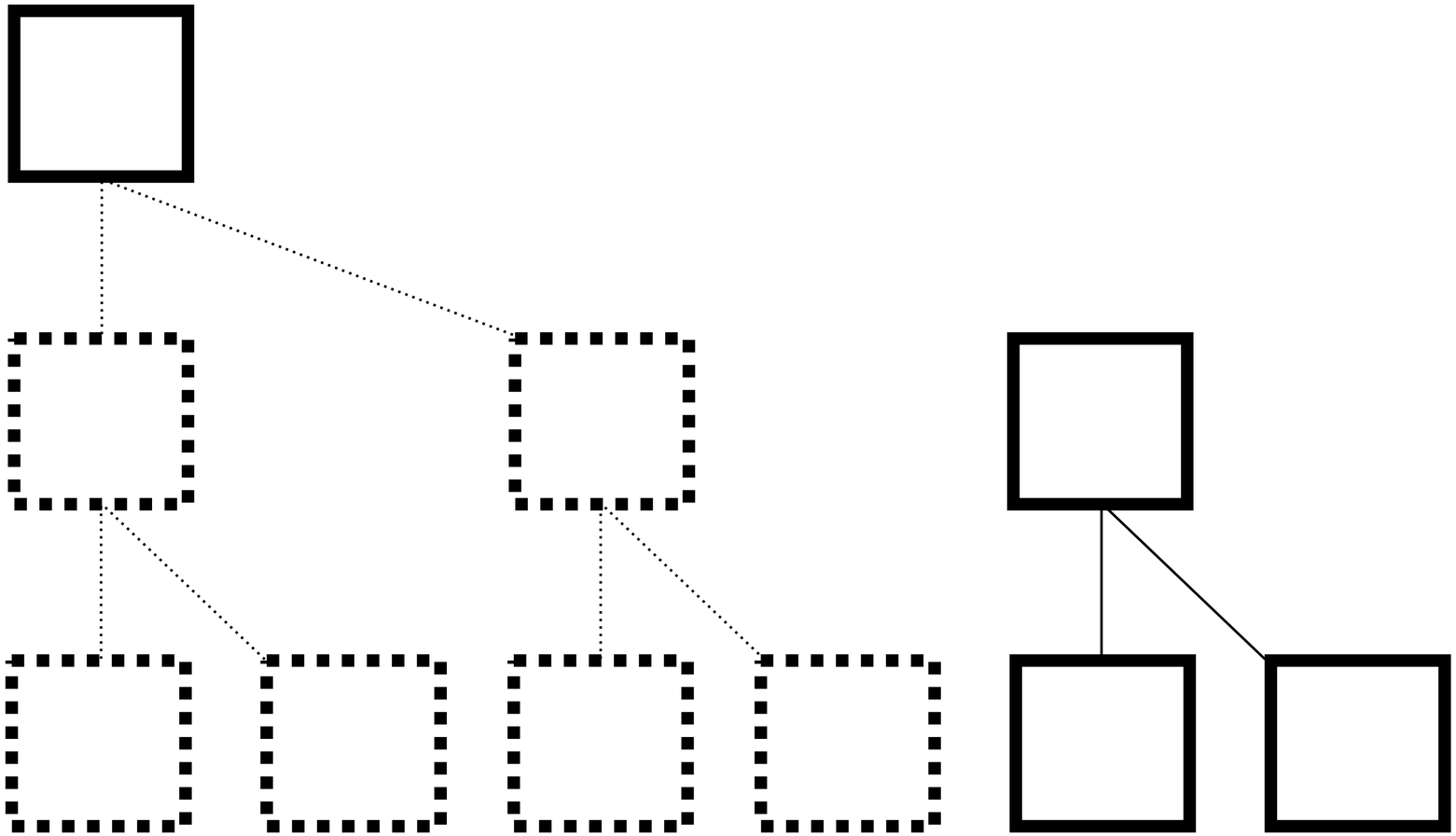
Concatenate



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

6 UTXOs

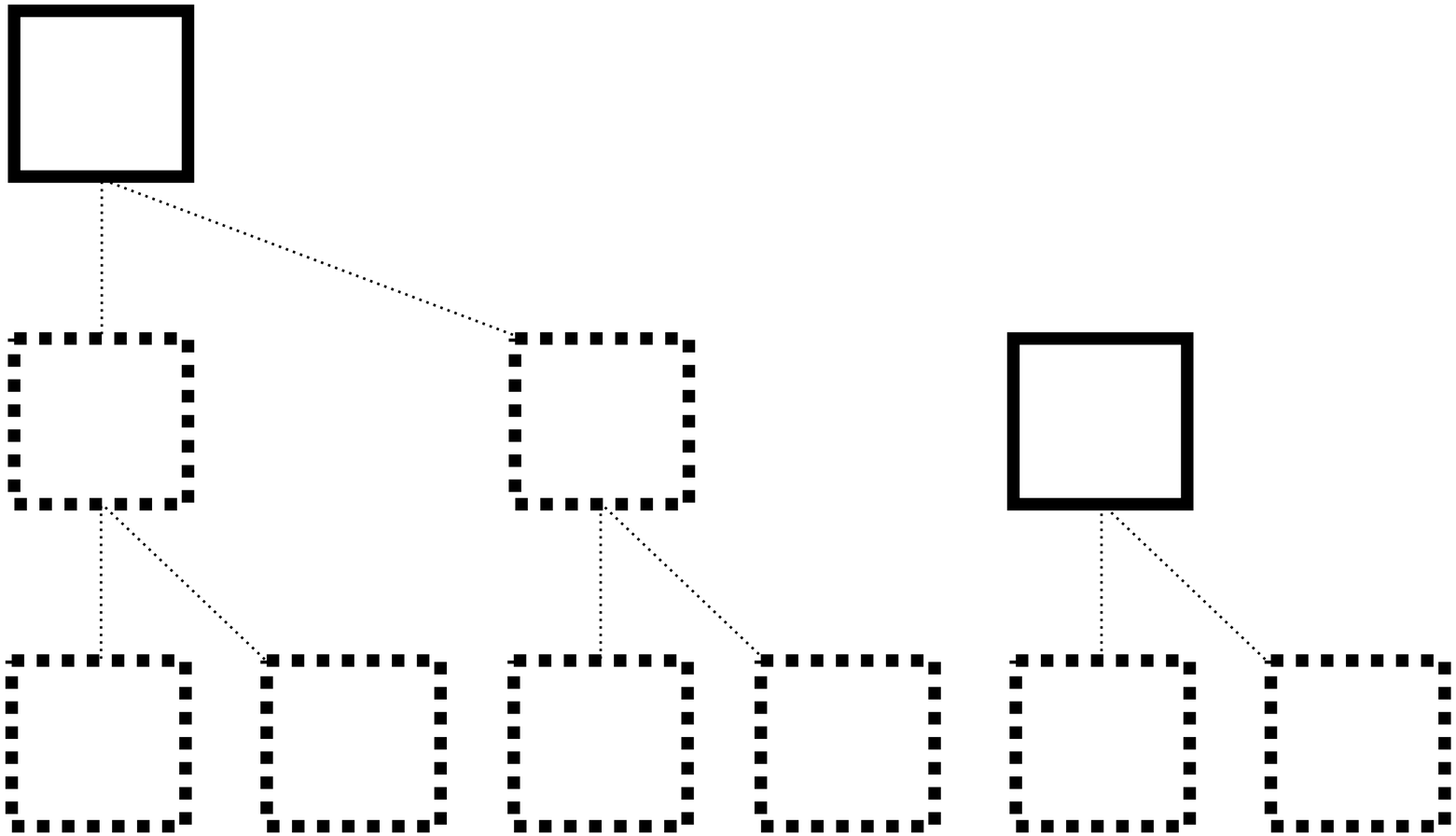
Hash



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

6 UTXOs

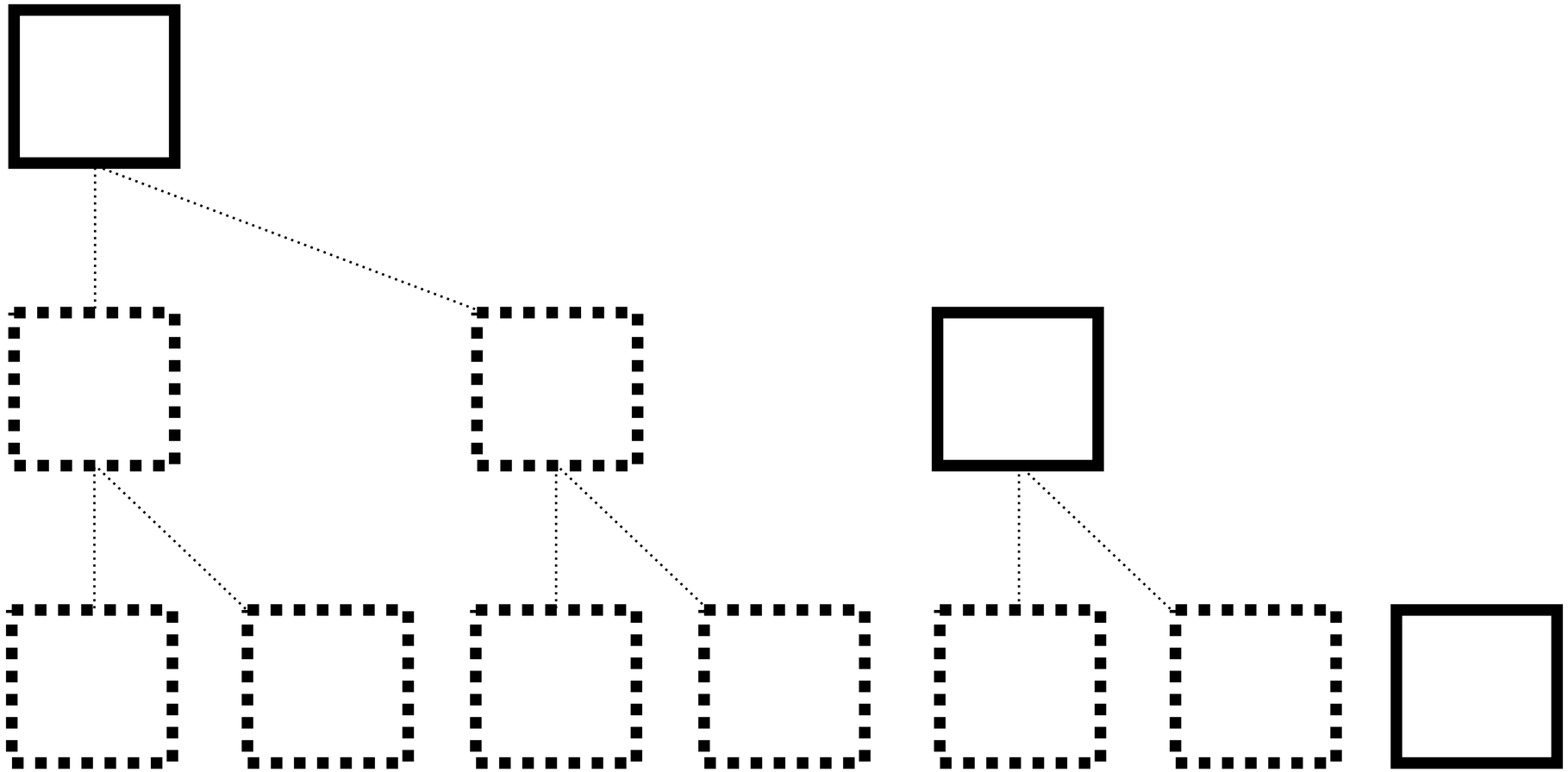
Throw away



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

7 UTXOs

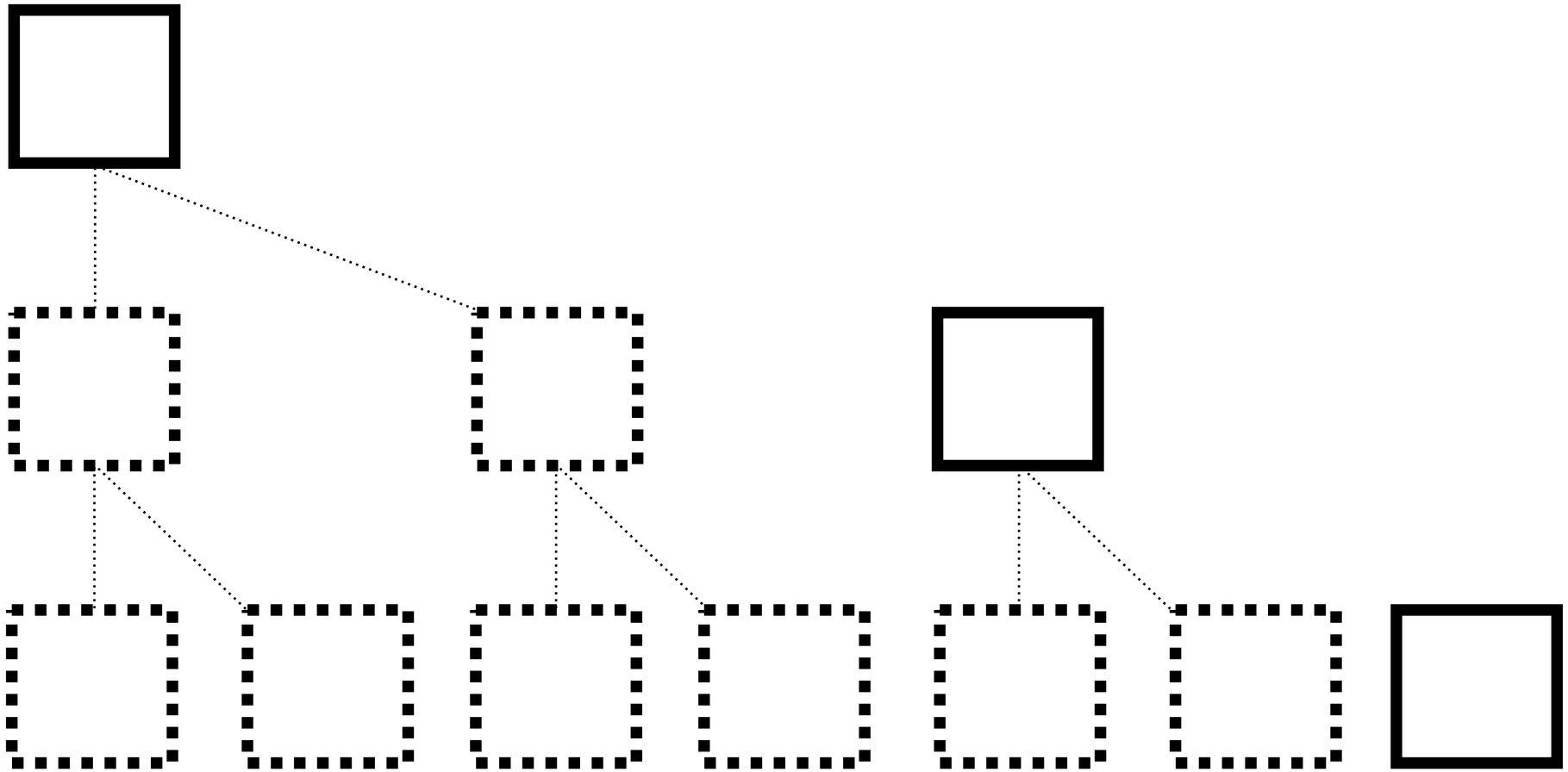
Another one



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

7 UTXOs

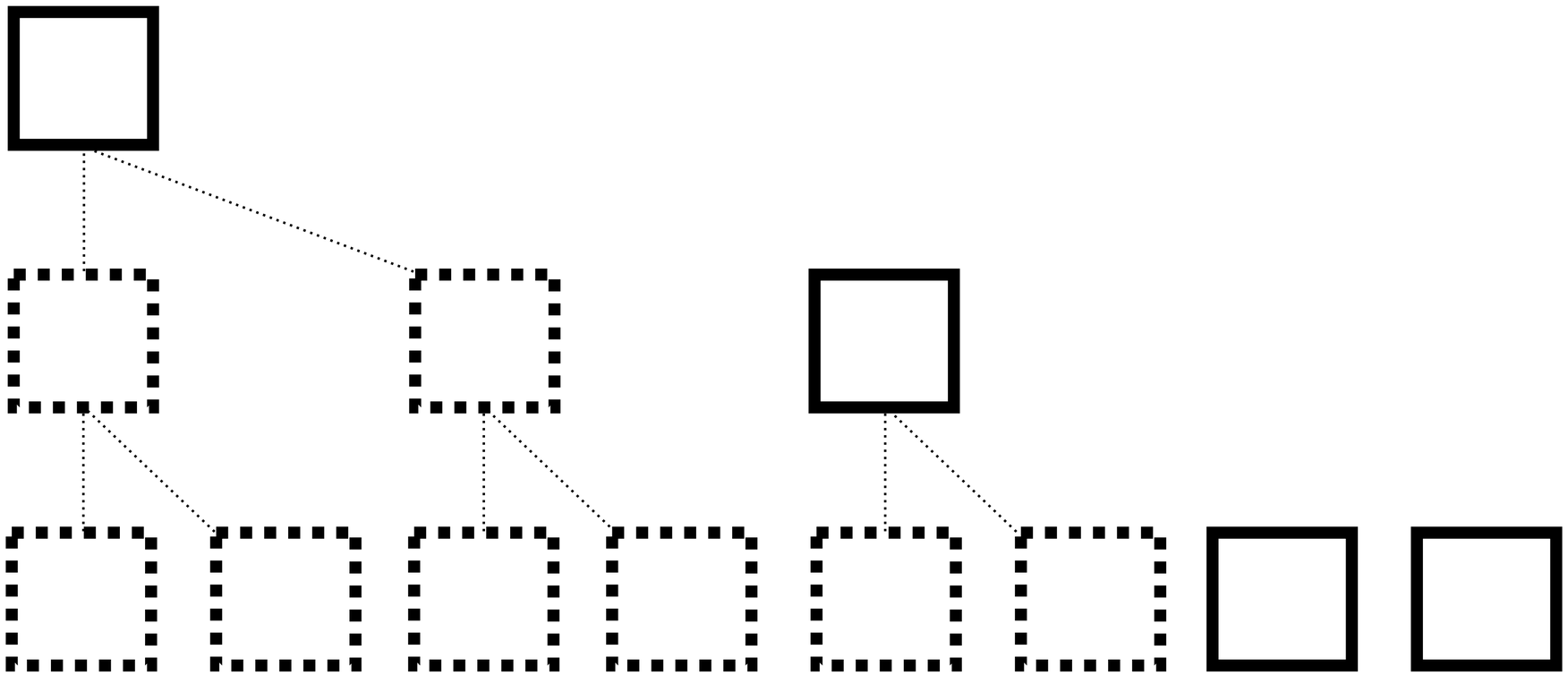
Nothing to hash with so it becomes a root



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

8 UTXOs

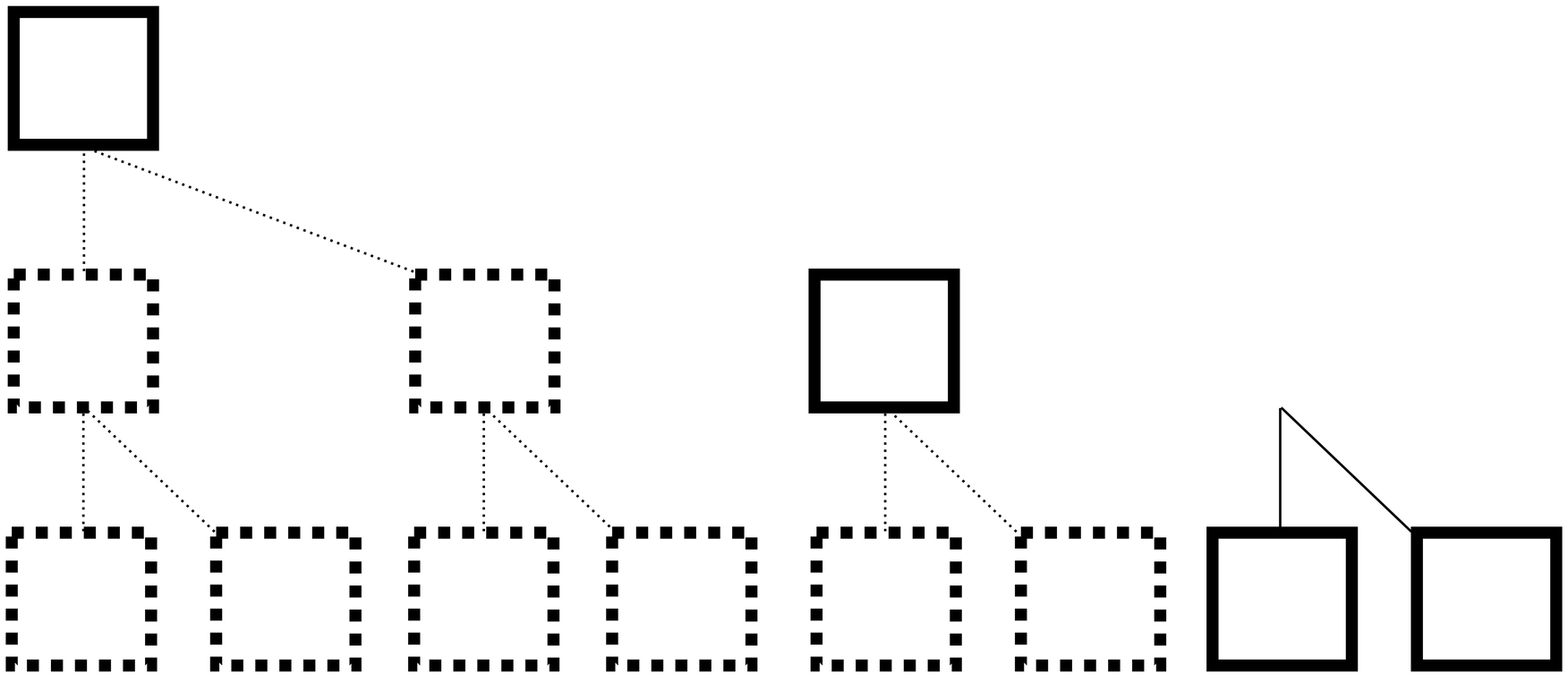
Another one



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

8 UTXOs

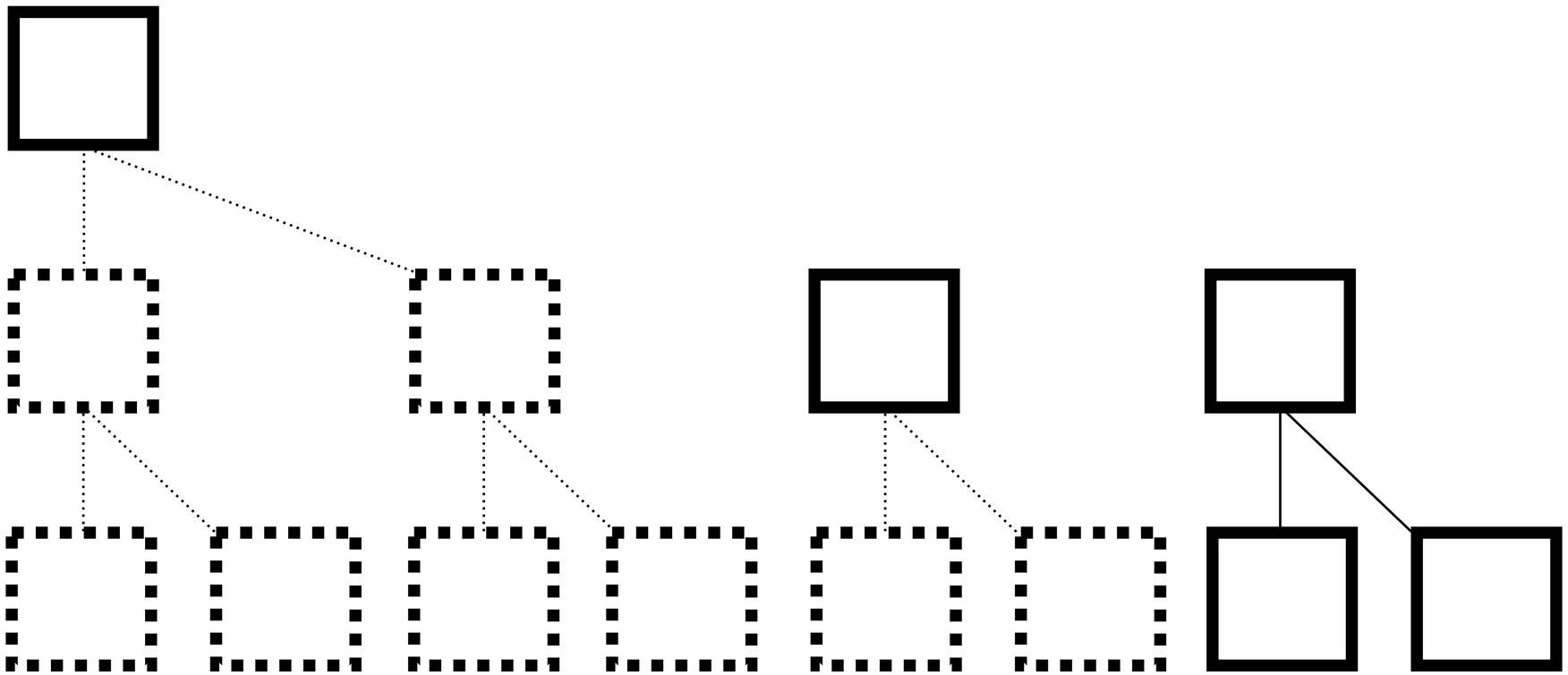
Concatenate



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

8 UTXOs

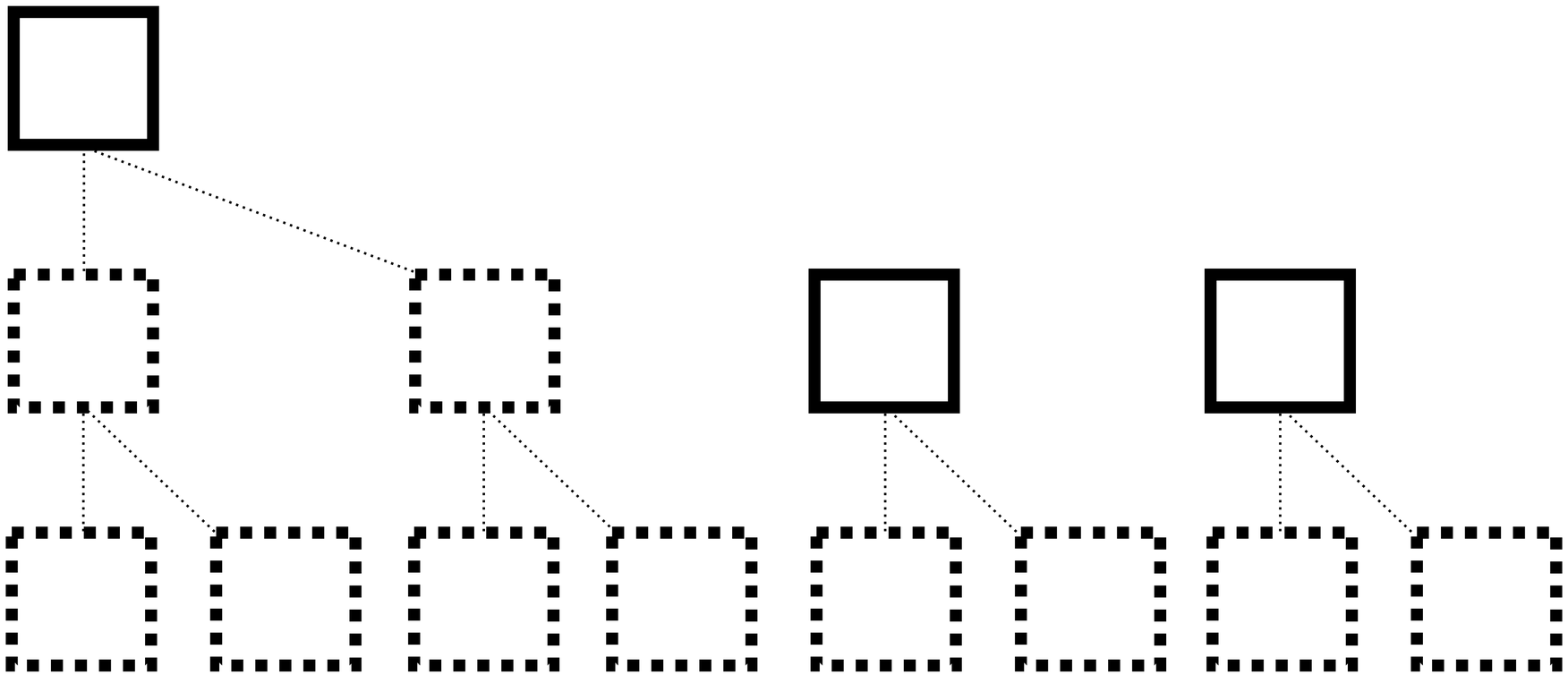
Hash



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

8 UTXOs

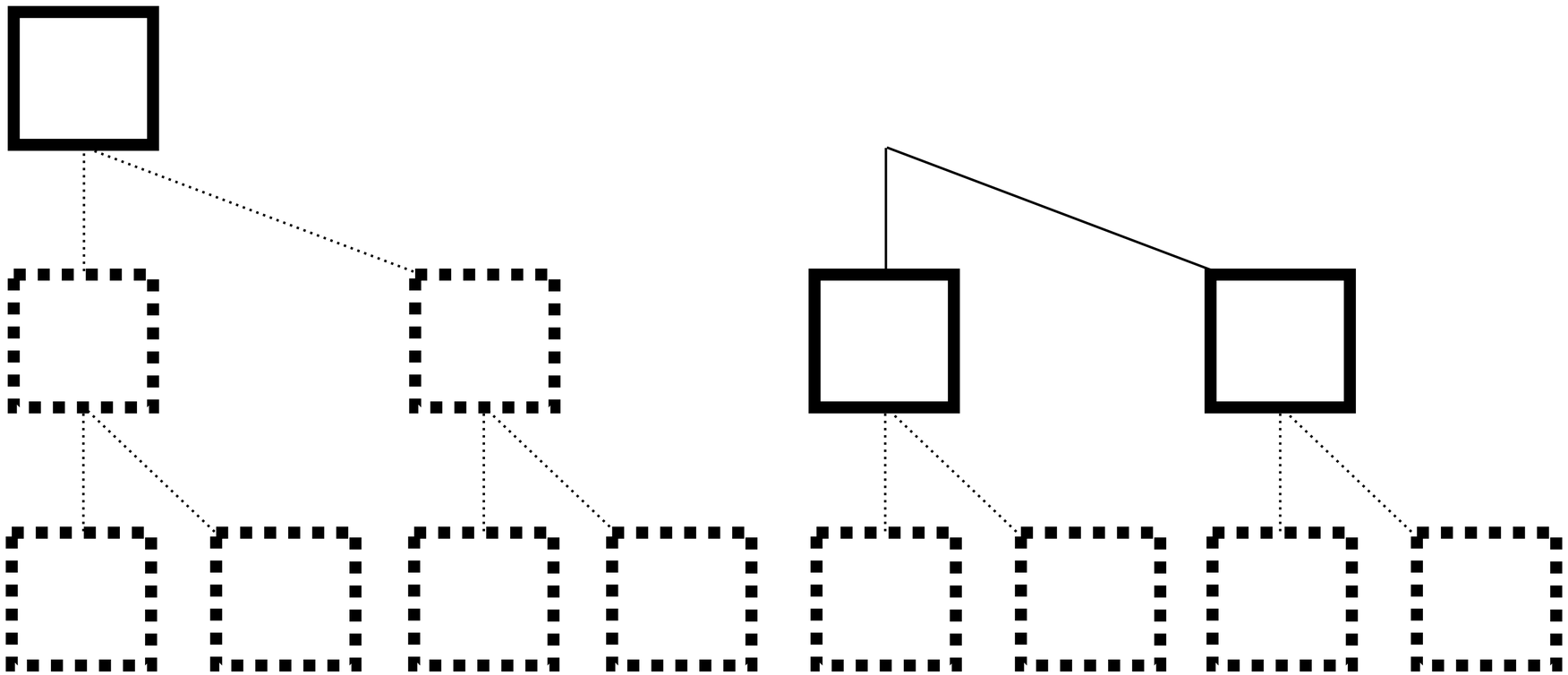
Throw away



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

8 UTXOs

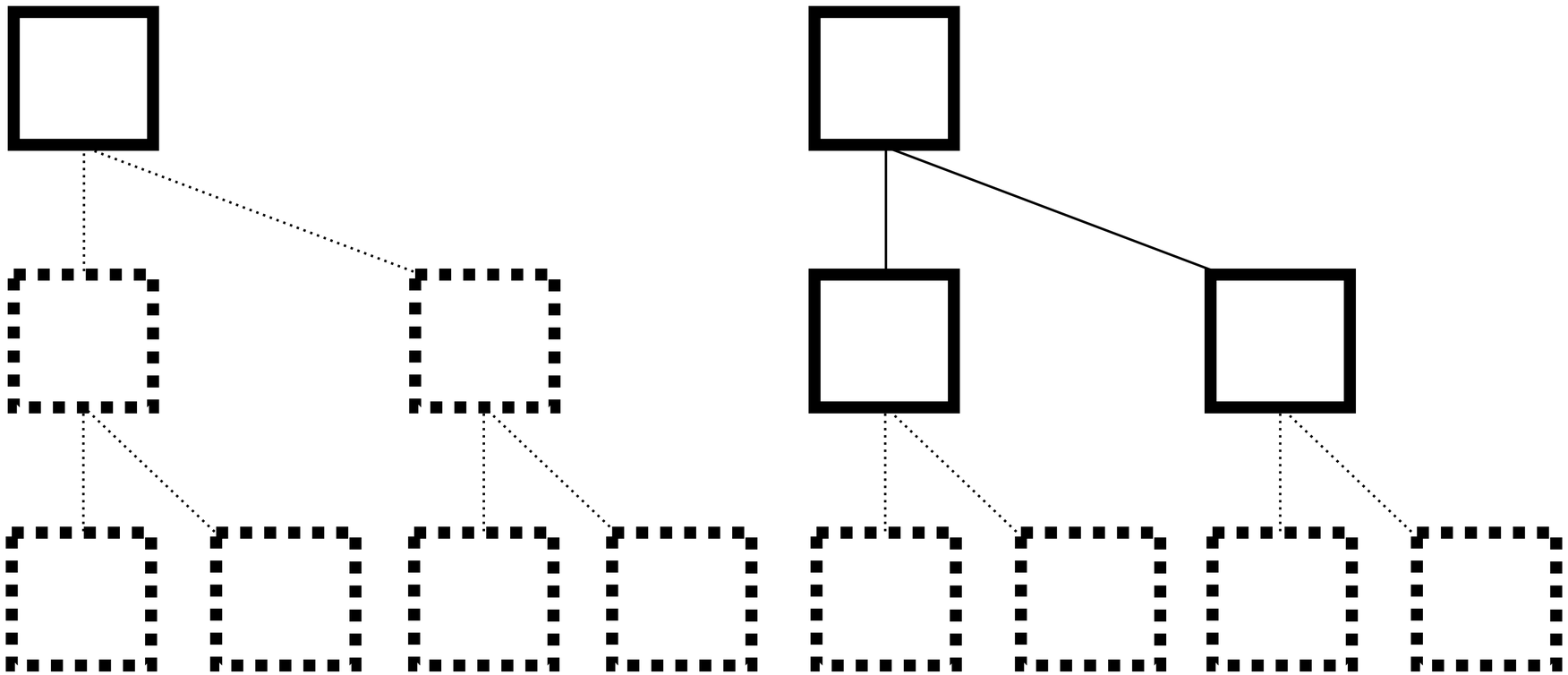
Concatenate



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

8 UTXOs

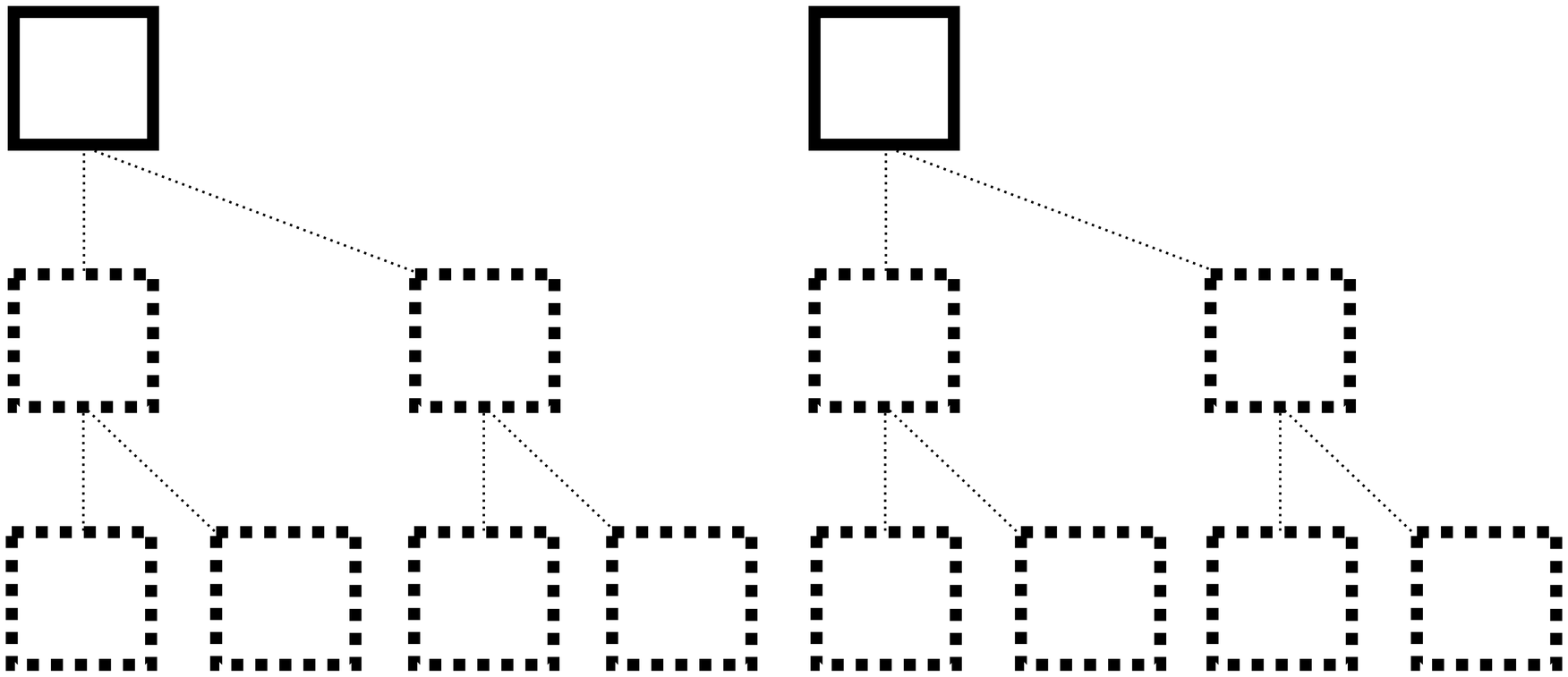
Hash



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

8 UTXOs

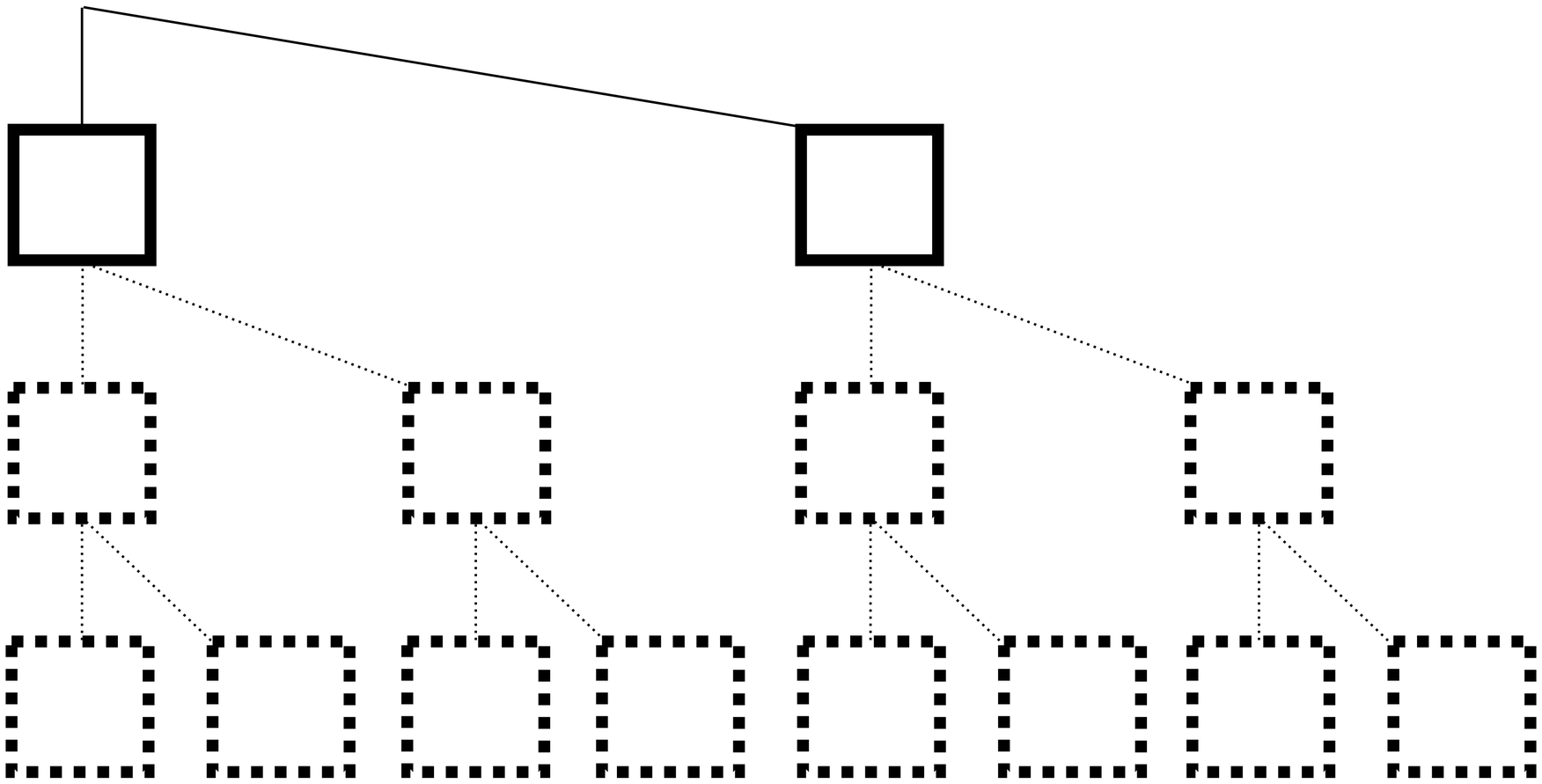
Throw away



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

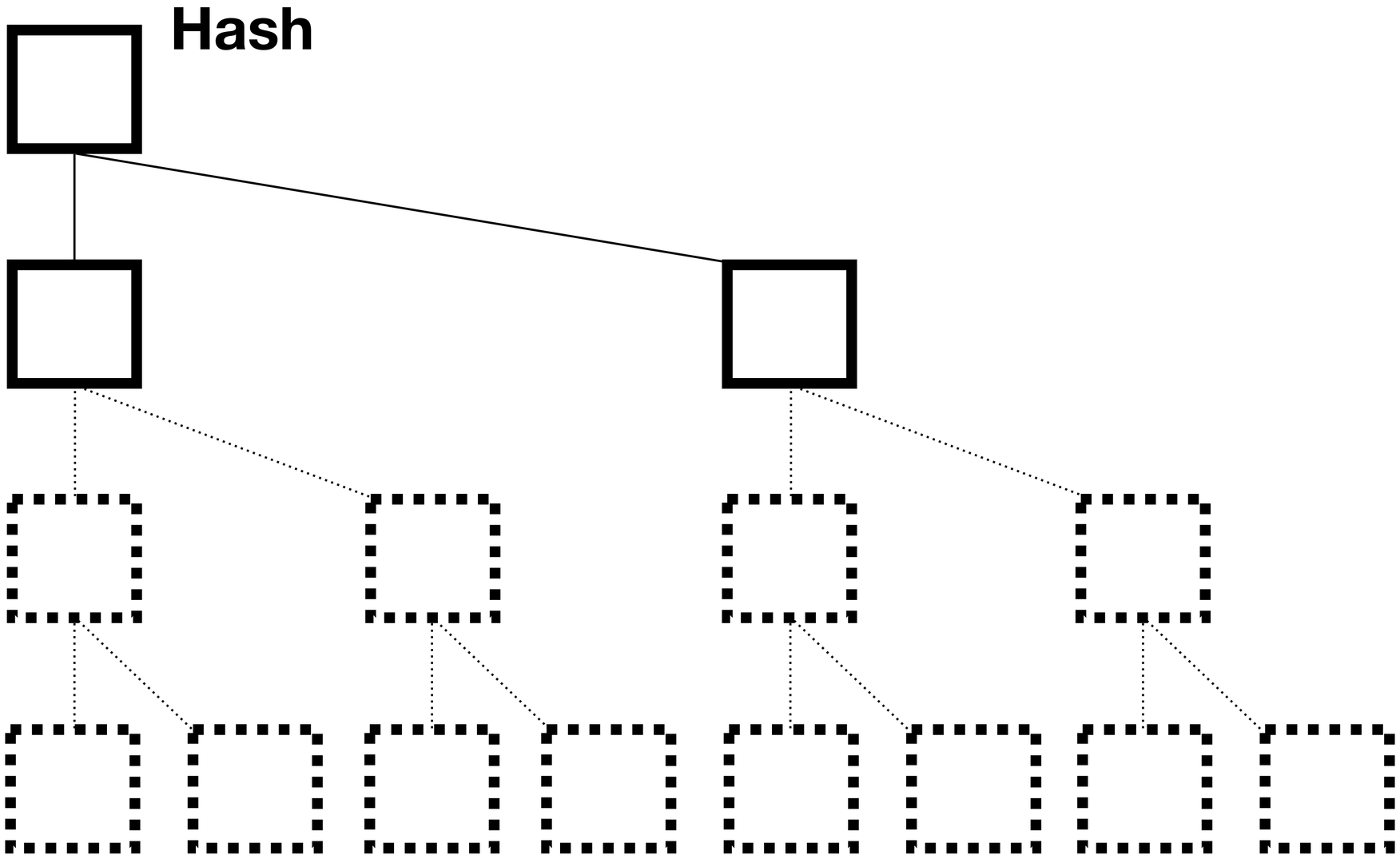
8 UTXOs

Concatenate



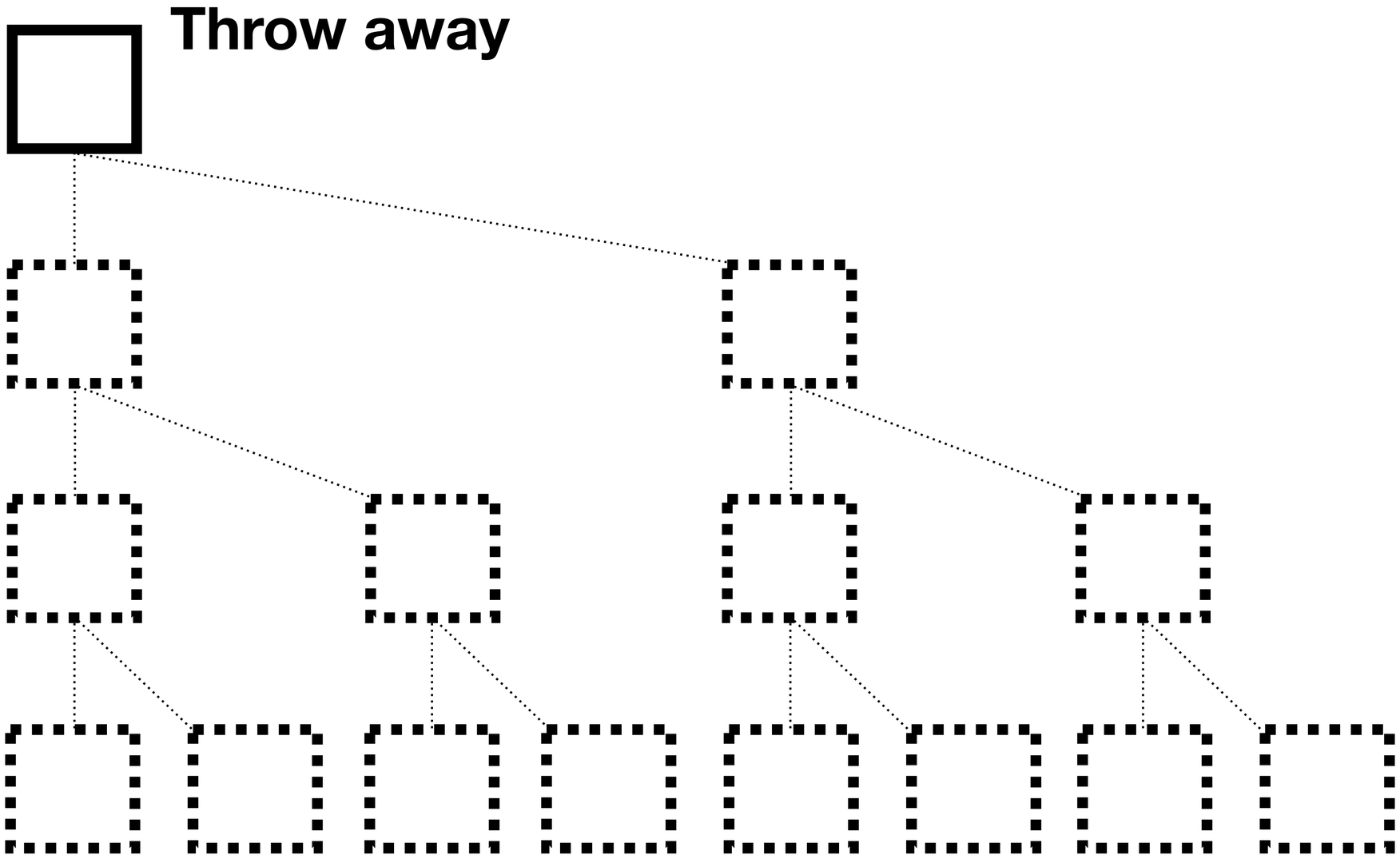
https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

8 UTXOs



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

8 UTXOs



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

Algorithm 2 DeleteOne

```
function DeleteOne(acc, proof)   ▷ Delete leaf from the accumulator
    n ← ∅
    h ← 0
    while h < len(proof) do
        p ← proof[h]                    ▷ Iterate over each proof element
        if n ≠ ∅ then
            n ← Parent(p, n)
        else if acc[h] = ∅ then
            acc[h] ← p
        else
            n ← Parent(p, acc[h])
            acc[h] ← ∅
        h ← h + 1
    acc[h] ← n
    return acc
```

Algorithm 2 De[illegible]One

```
function D[illegible]E(acc, proof)   ▷ Delete[illegible]the accumulator
    n ← ∅[illegible]
    h ← 0
    while h < [illegible]
        p ← proof[illegible]ver each proof element
        if n ≠ ∅ then[illegible]
            n ← PARENT[illegible]
        else if acc[h] = ∅[illegible]
            acc[h] ← p[illegible]
        else
            n ← P[illegible]])
            acc[illegible]
        h ←[illegible]
    acc[h][illegible]
    return ac[illegible]
```

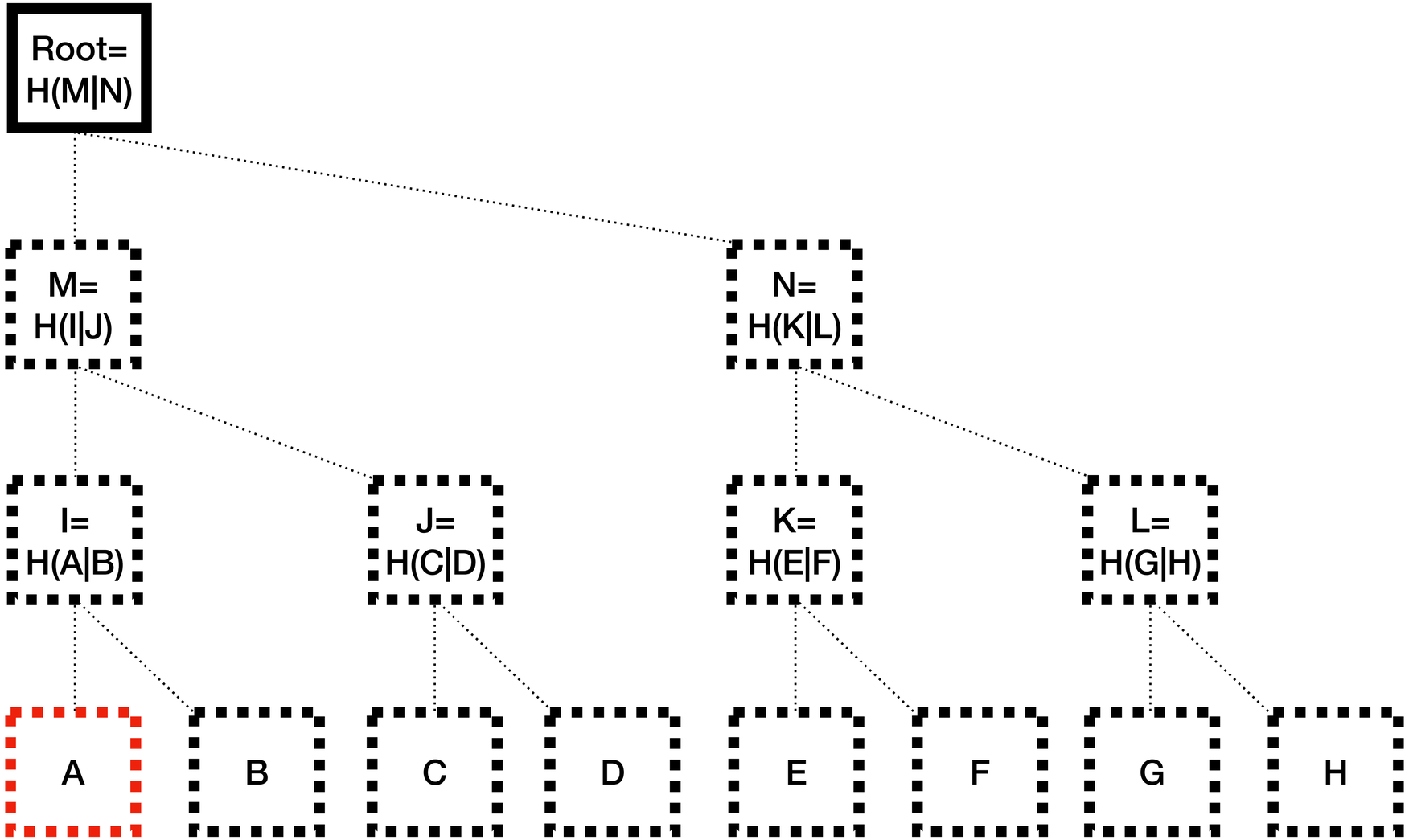
7 Appendix

7.1 Batch deletion process

The following is a description of the batch deletion process. While the single element deletion process is sufficient and can be invoked repeatedly, it is inefficient to do so. When many deletions occur simultaneously we can significantly reduce the number of hash operations needed to remove elements from the accumulator by batching the deletions.

Similar to the $DeleteOne()$ function, the process for deleting many elements operates a row at a time, climbing from the bottom to the top of the forest. At each row, the same phases are followed. The data operated on at each row, includes the known hashes and their positions, as well a list of which locations to delete. The phases are, in order: "Twin", or twin extraction, "Swap", or sibling swapping, "Root", promoting a node to or demoting a node from a root position, and "Climb", ascending to the next row up. These 4 phases are described with accompanying diagrams. In the following diagrams, green nodes are nodes which are always known, as they are or were tree roots. Pink nodes are nodes which are being deleted.

Deleting A



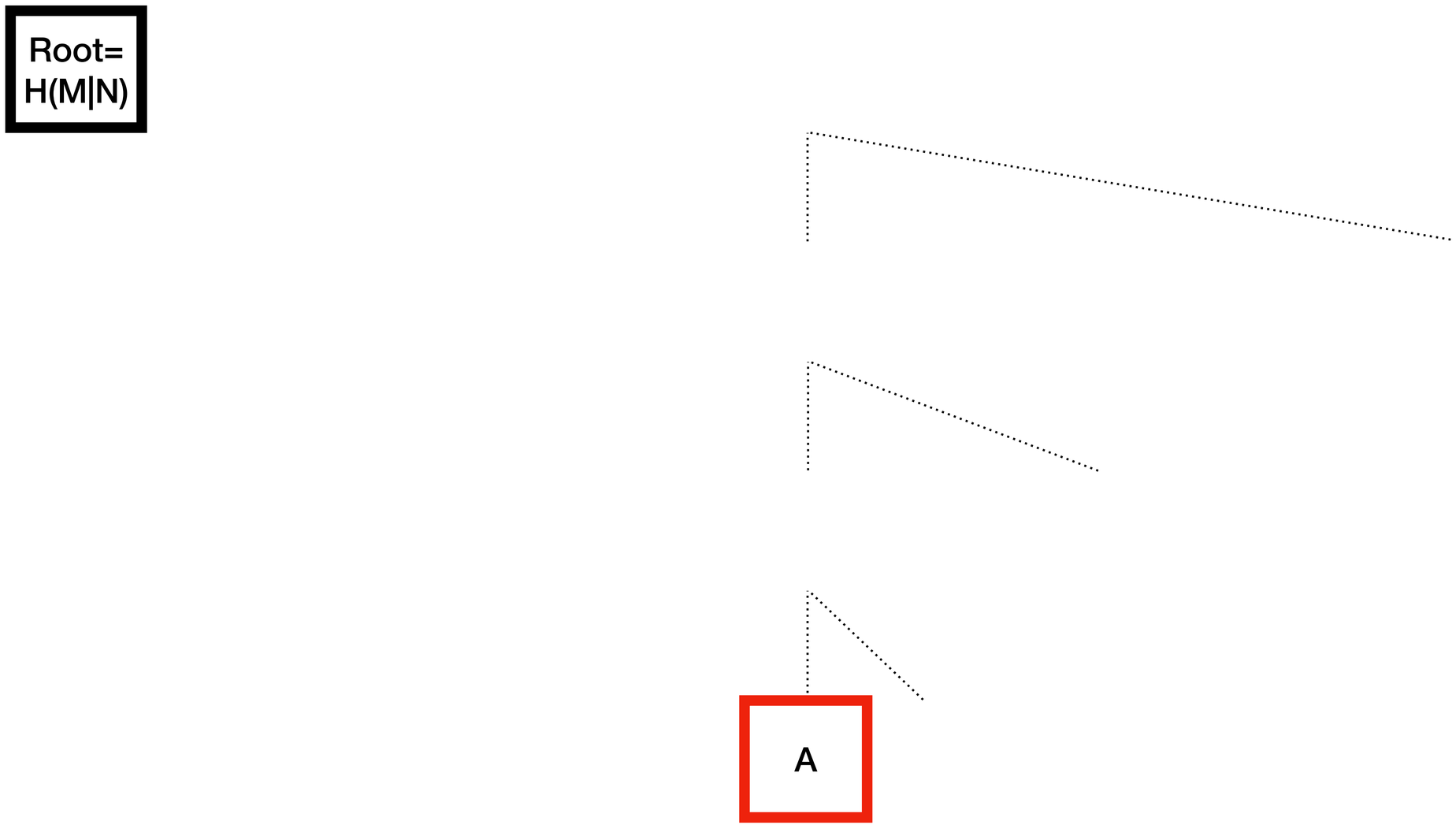
https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

The only requirement is the roots

Root=
H(M|N)

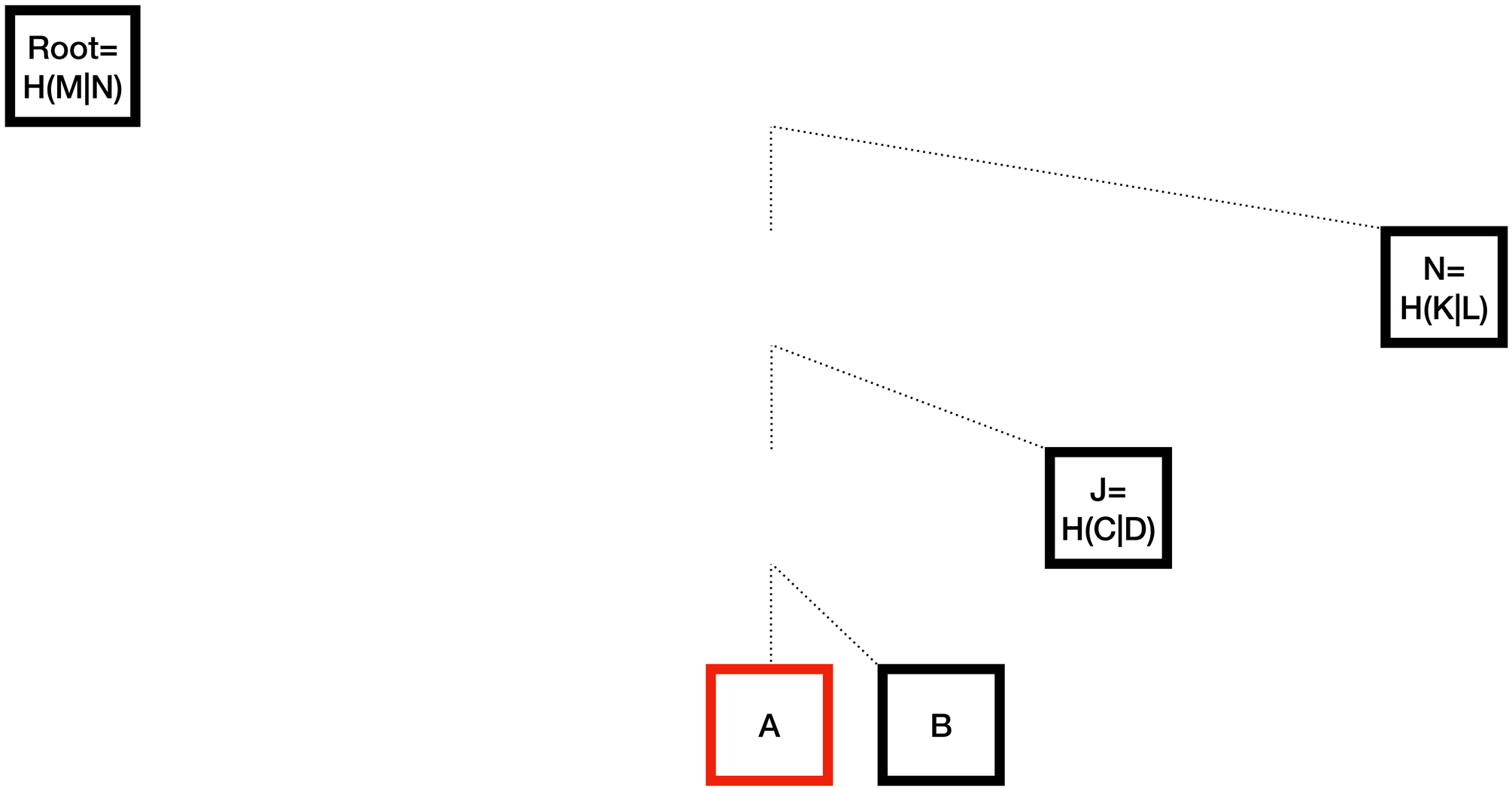
https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

Calculate A from the Bitcoin Block



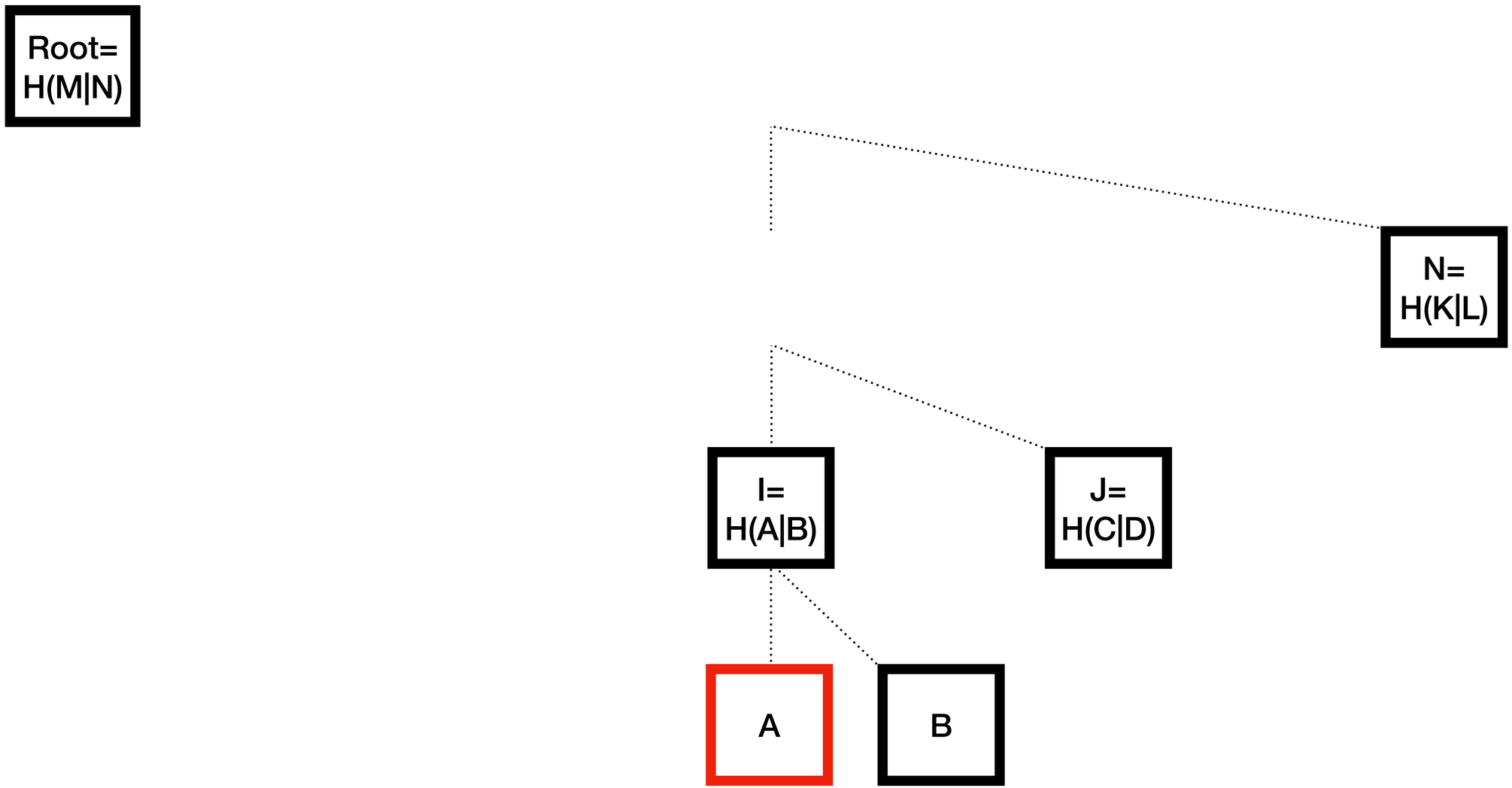
https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

Receive the proof: B, J, N



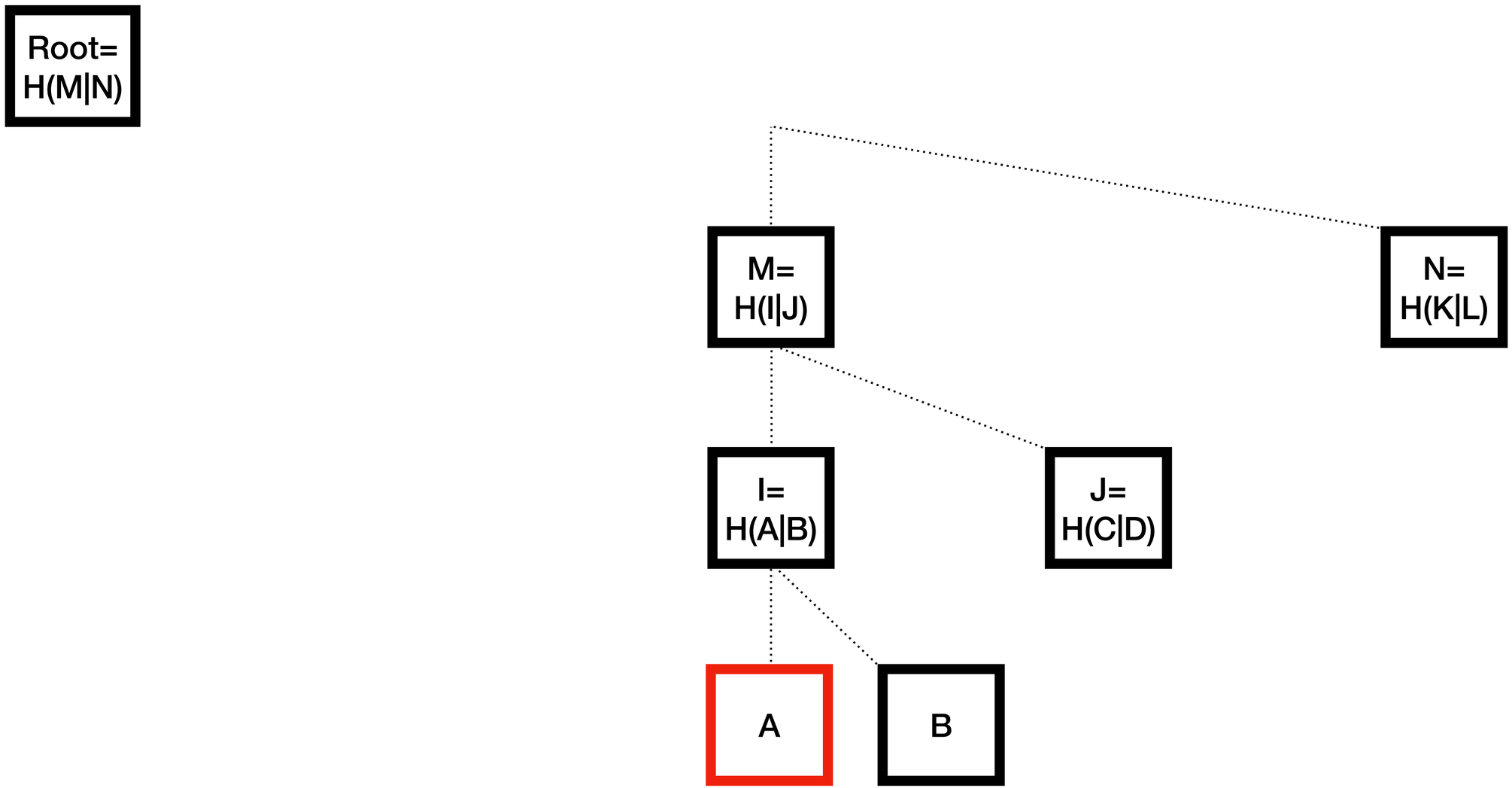
https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

Calculate the hash for I



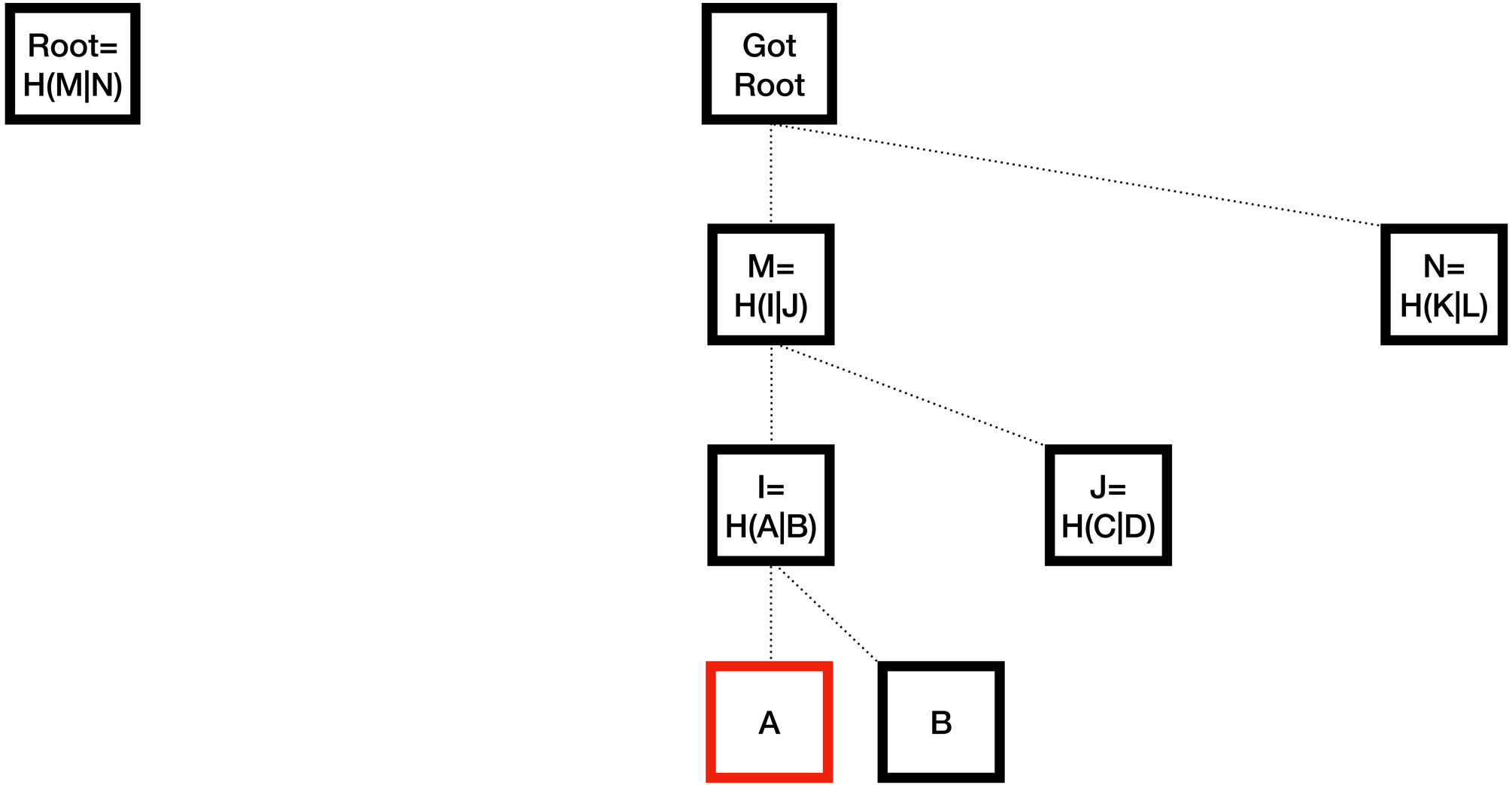
https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

Calculate the hash for M



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

Calculate Root



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

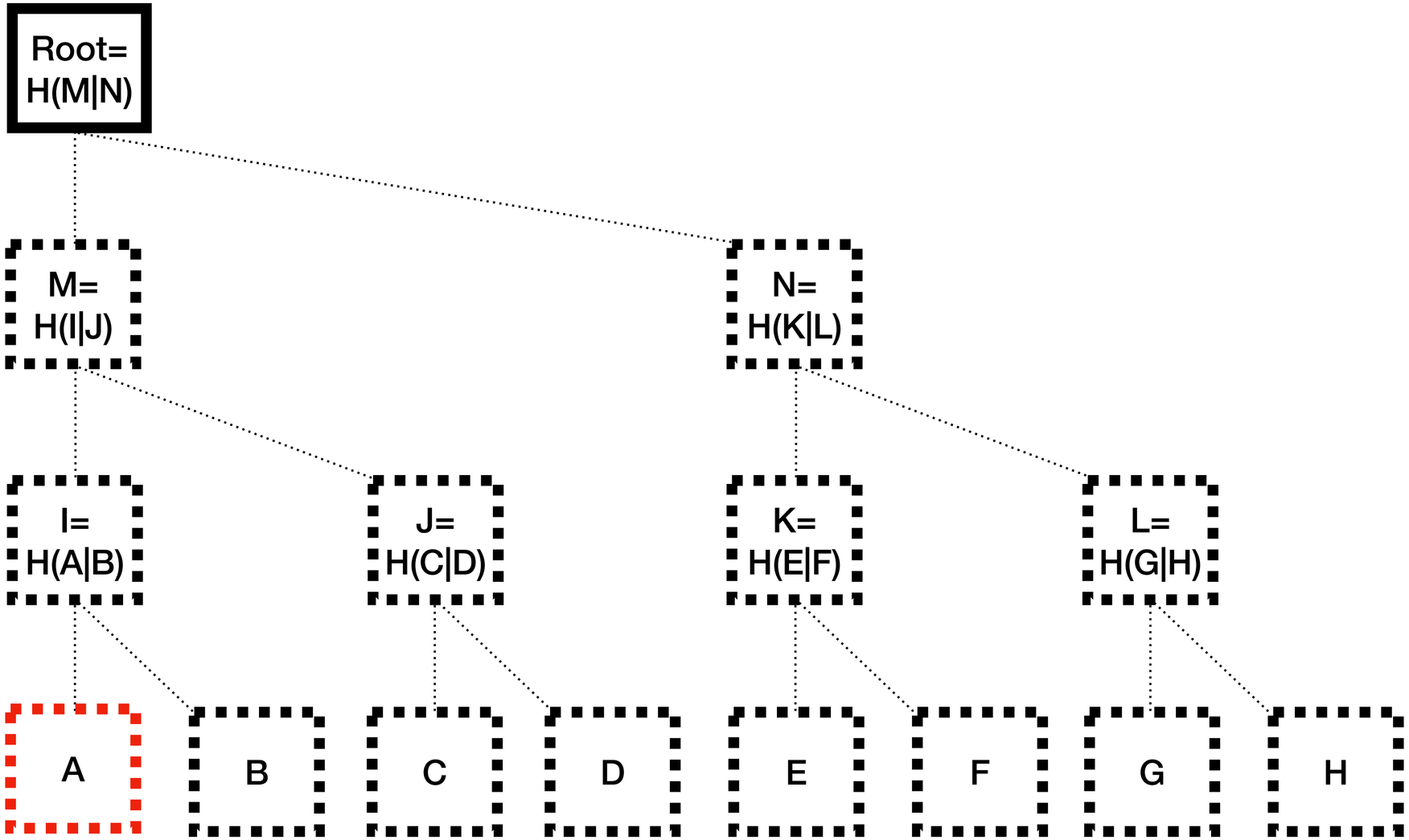
Compare roots

Continue if the roots are equal. Ban peer if not equal

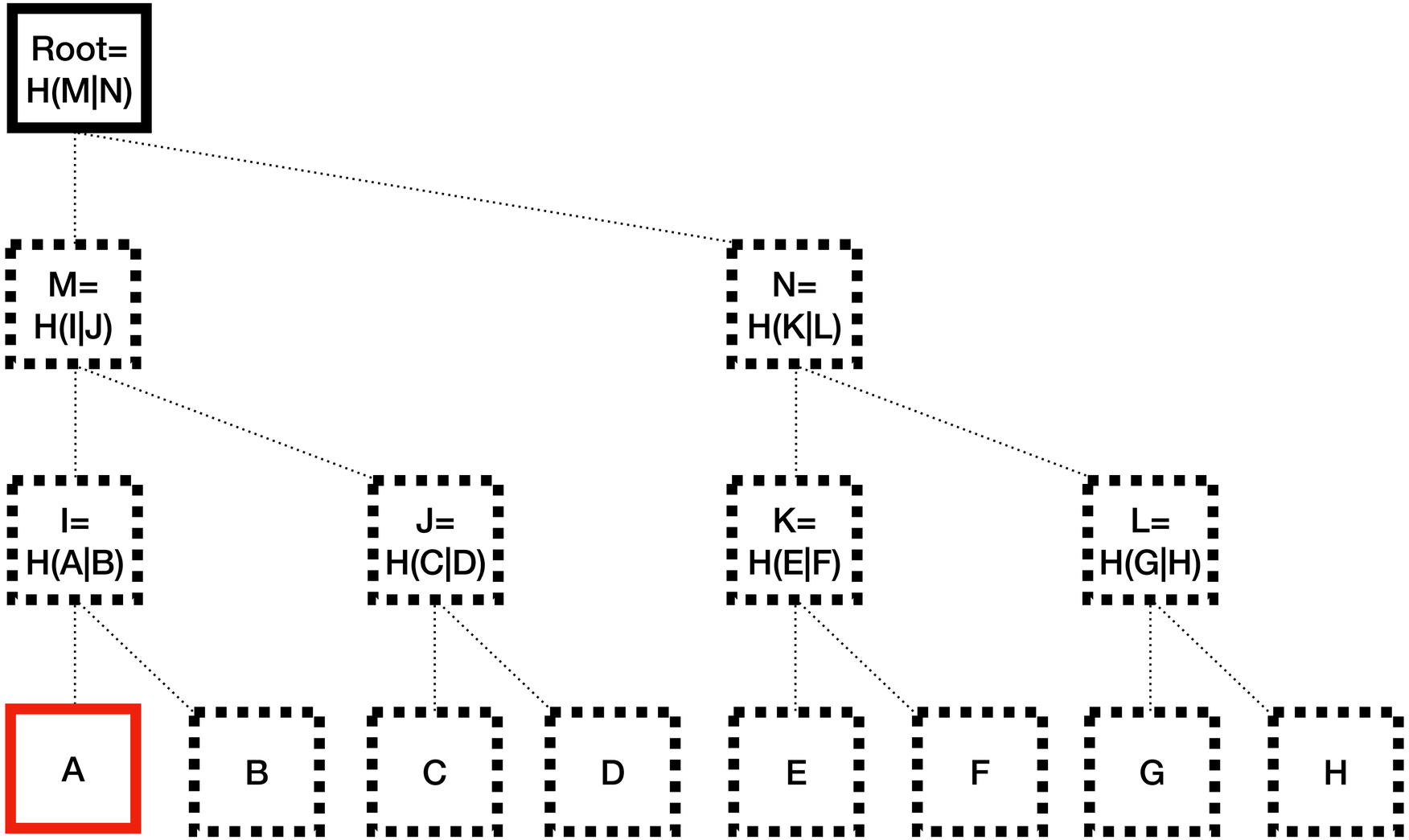
Root=H(M|N) == Got Root

https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

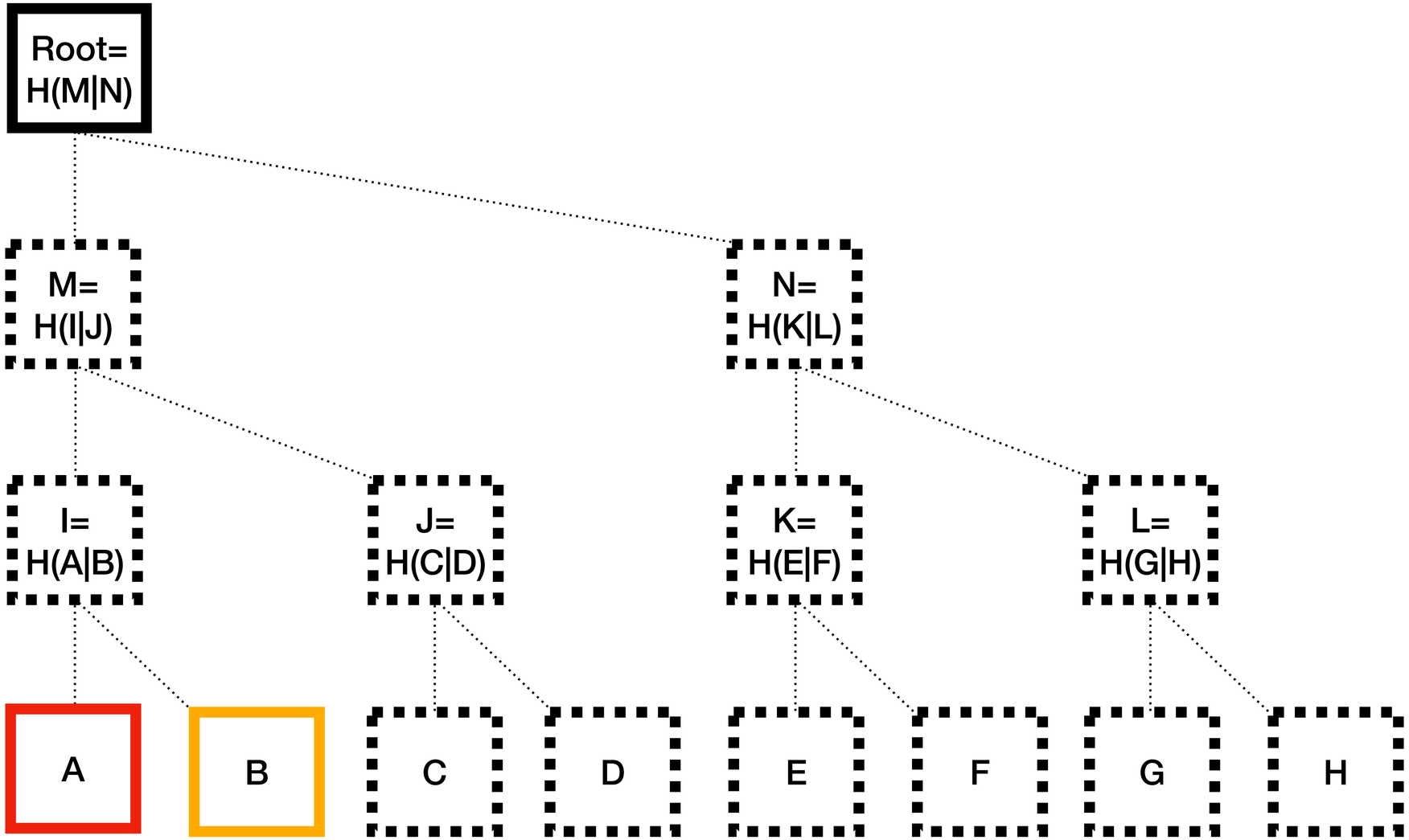
Deleting A



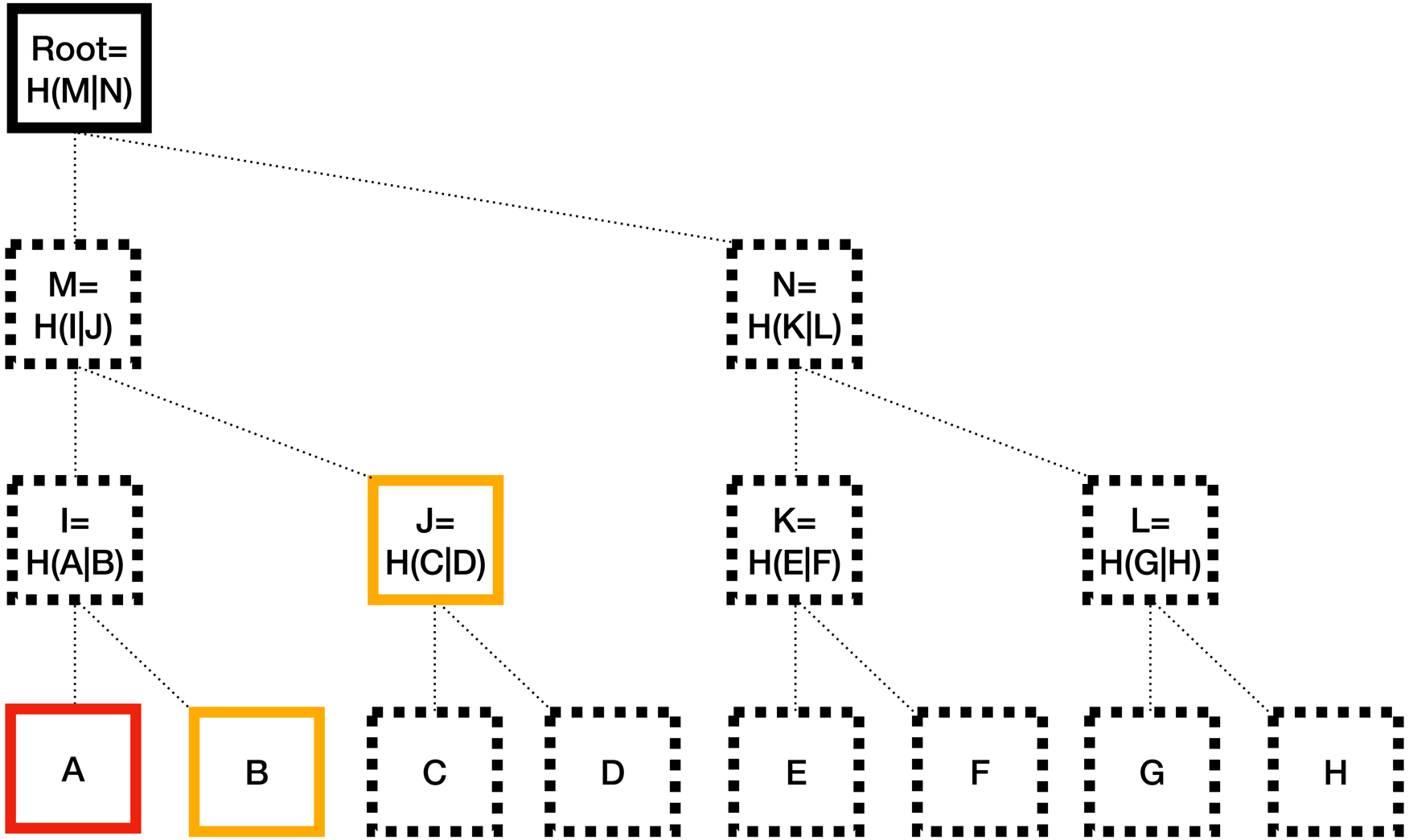
Populate old nodes



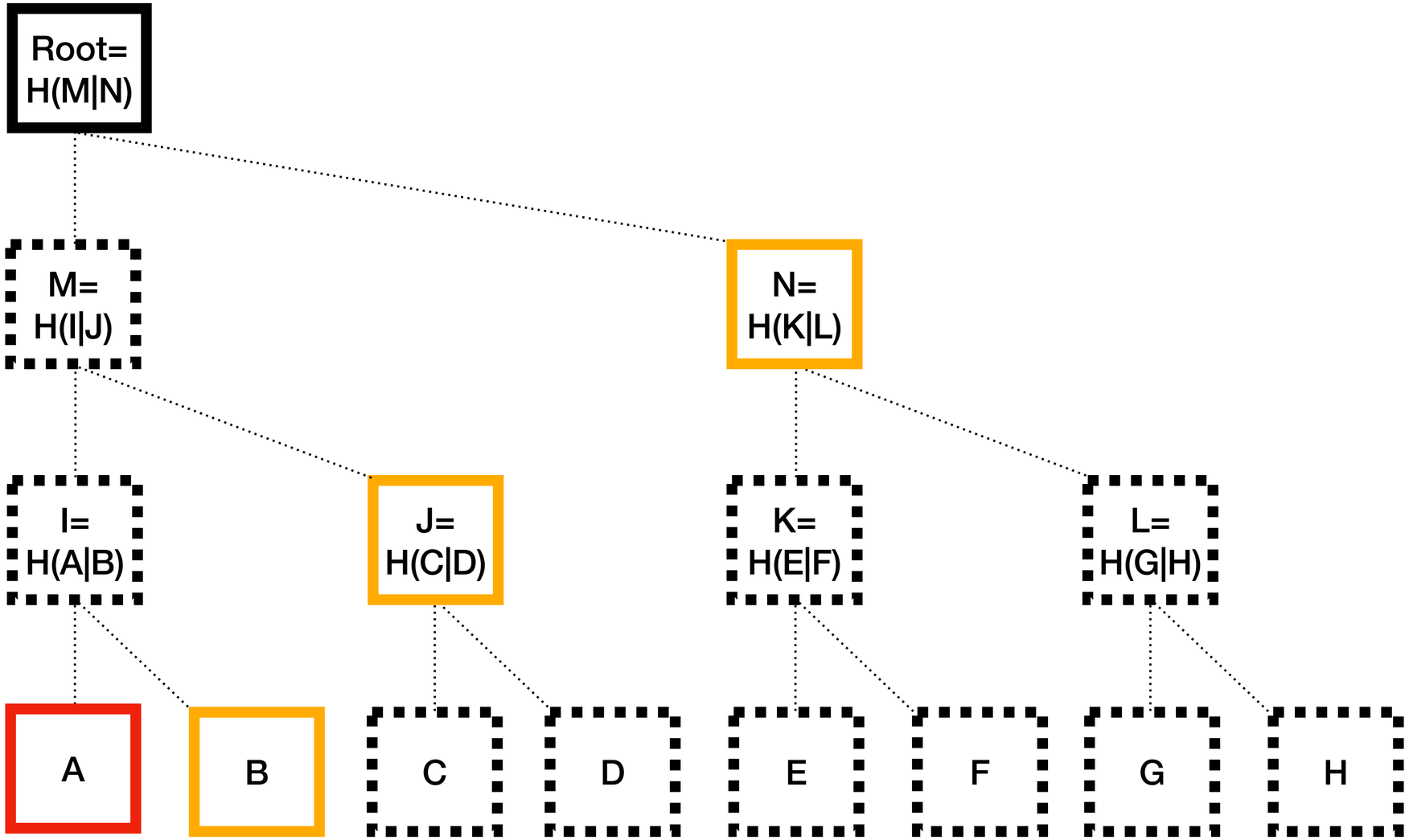
Populate old nodes



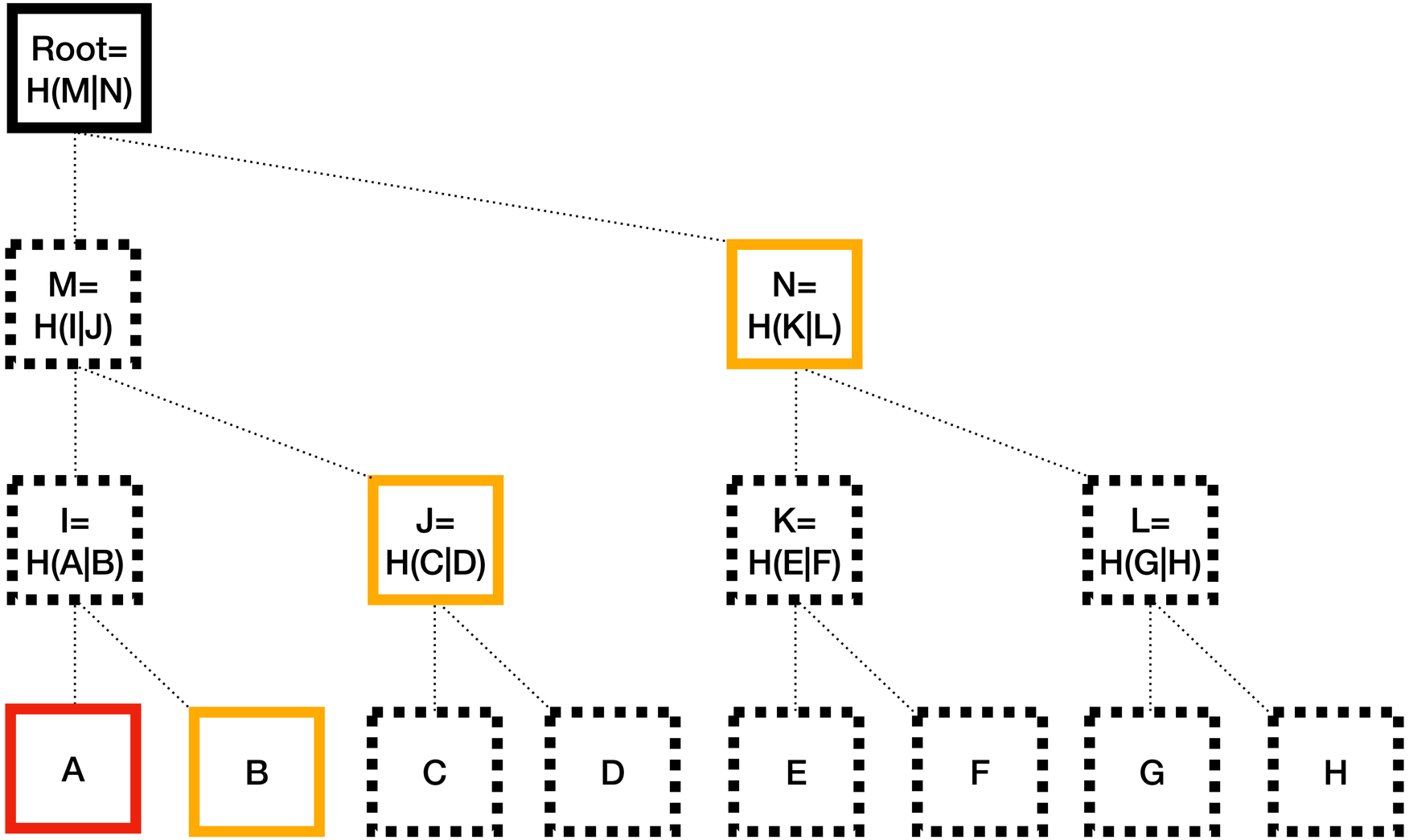
Populate old nodes



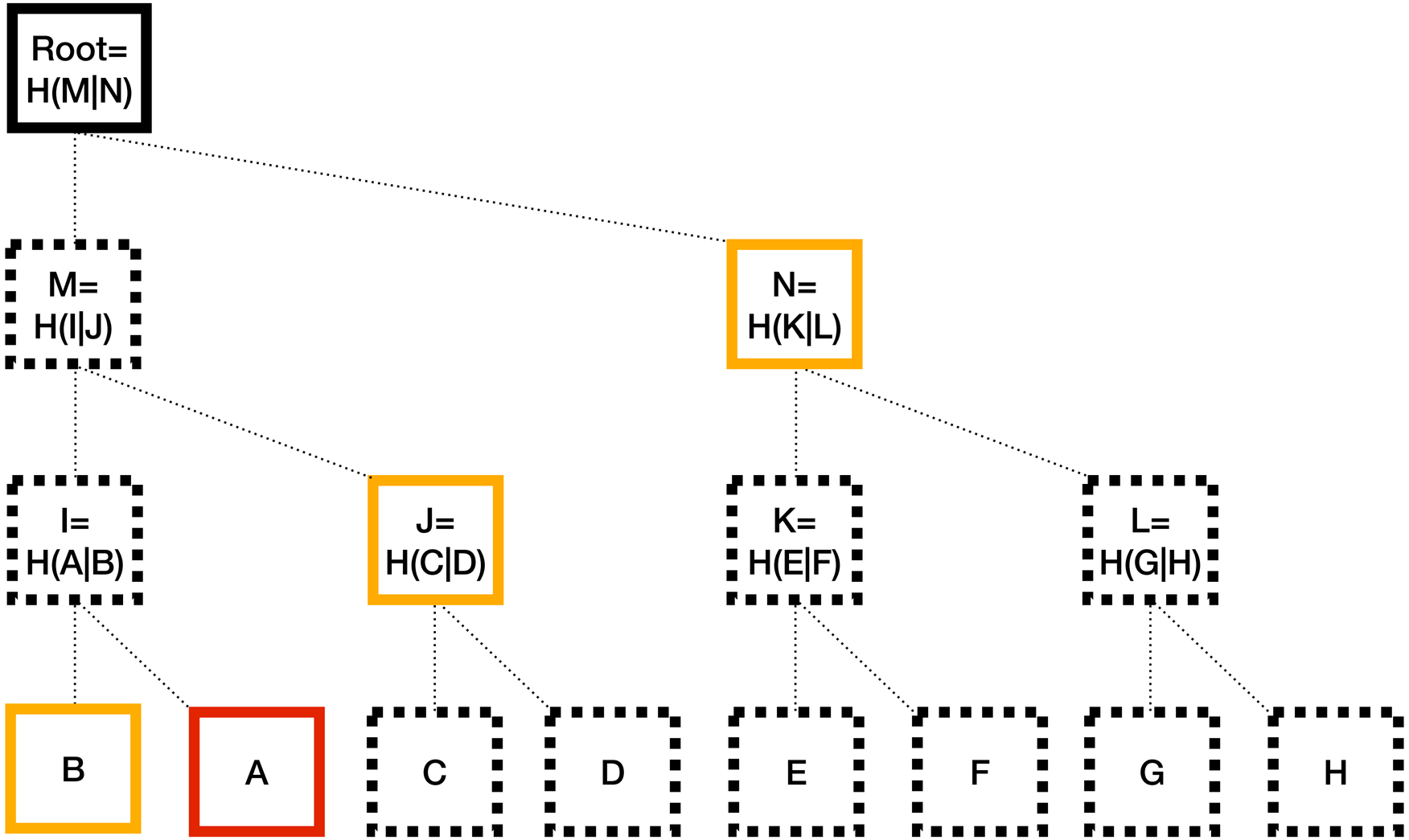
Populate old nodes



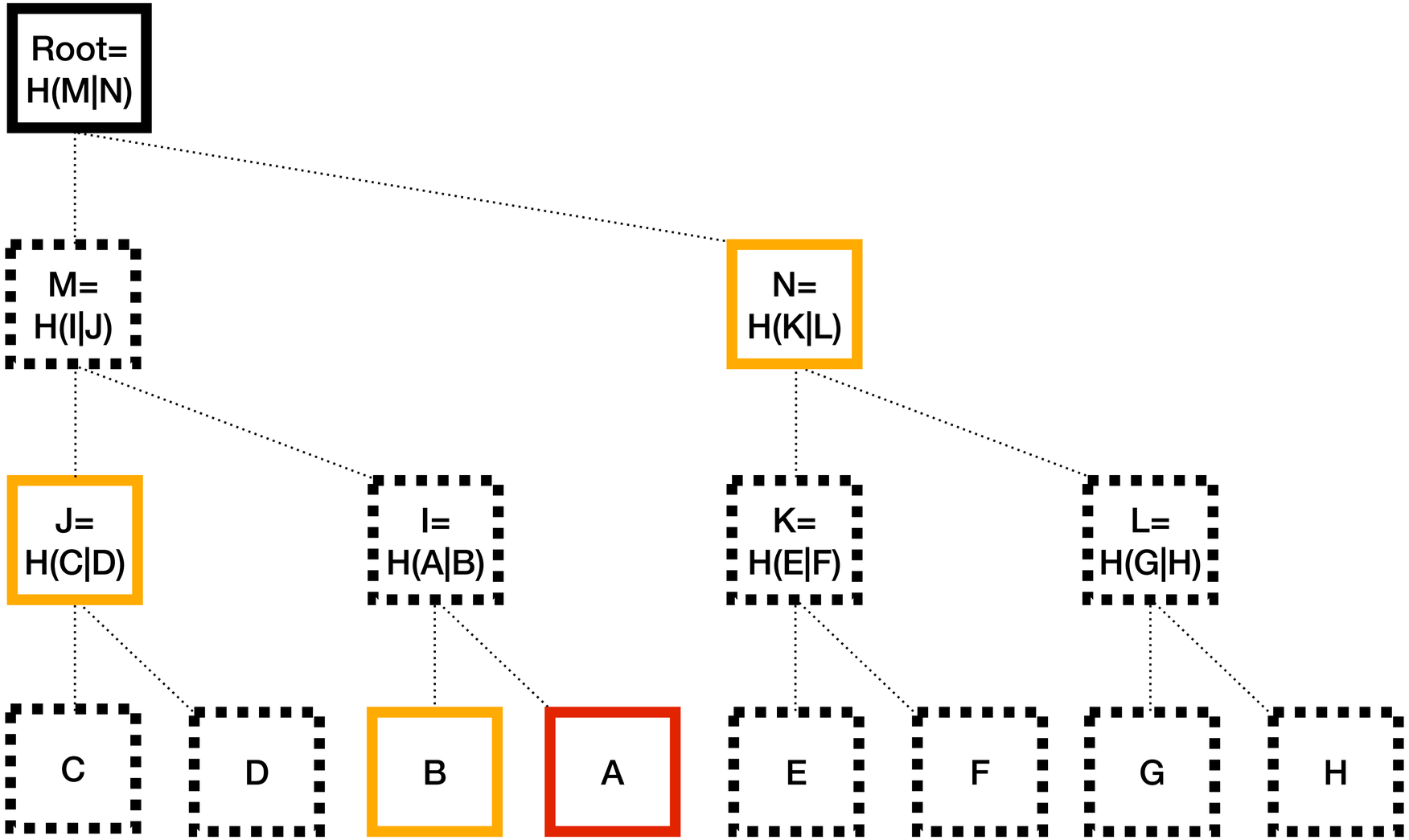
Swap from row 0



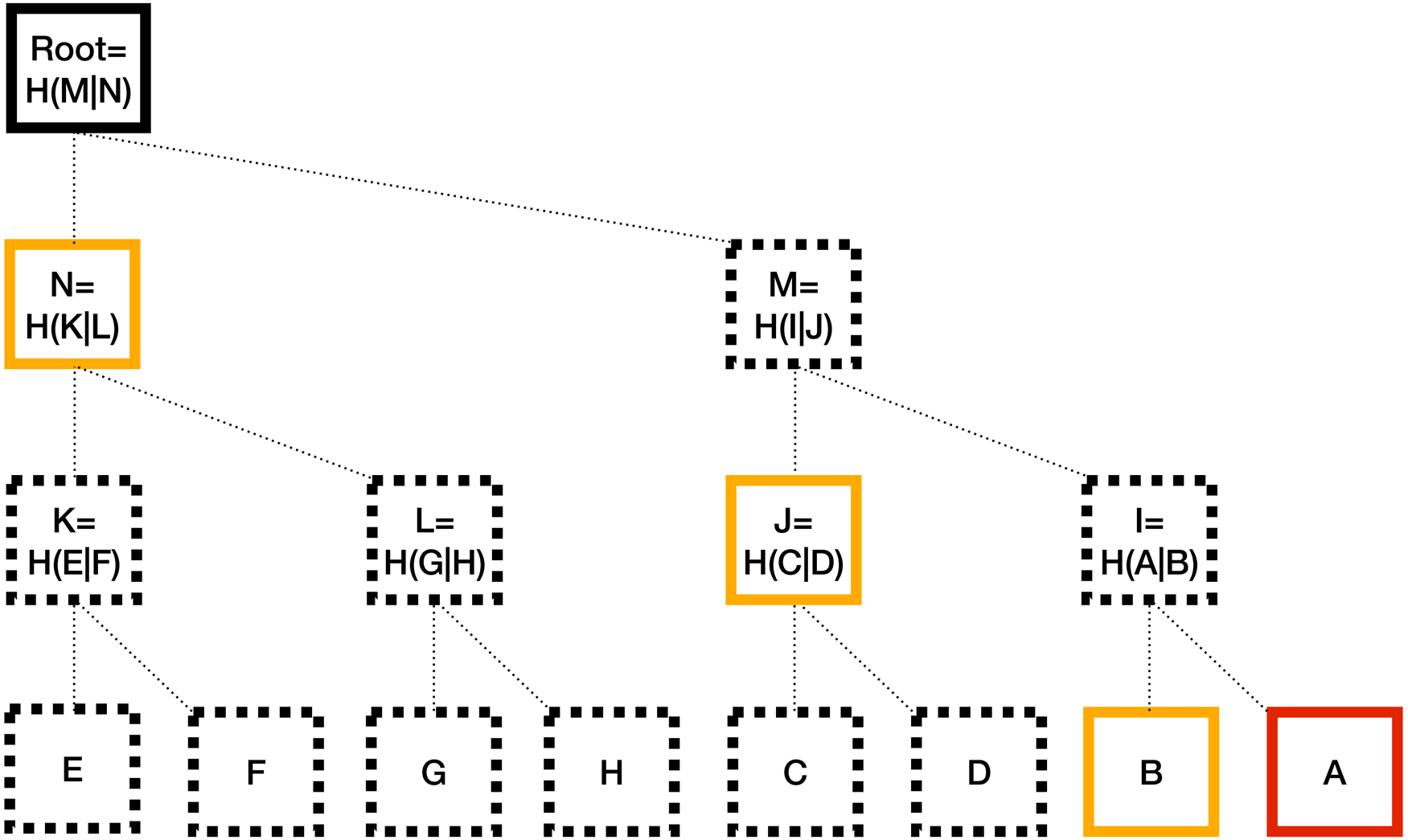
Swap from row 0



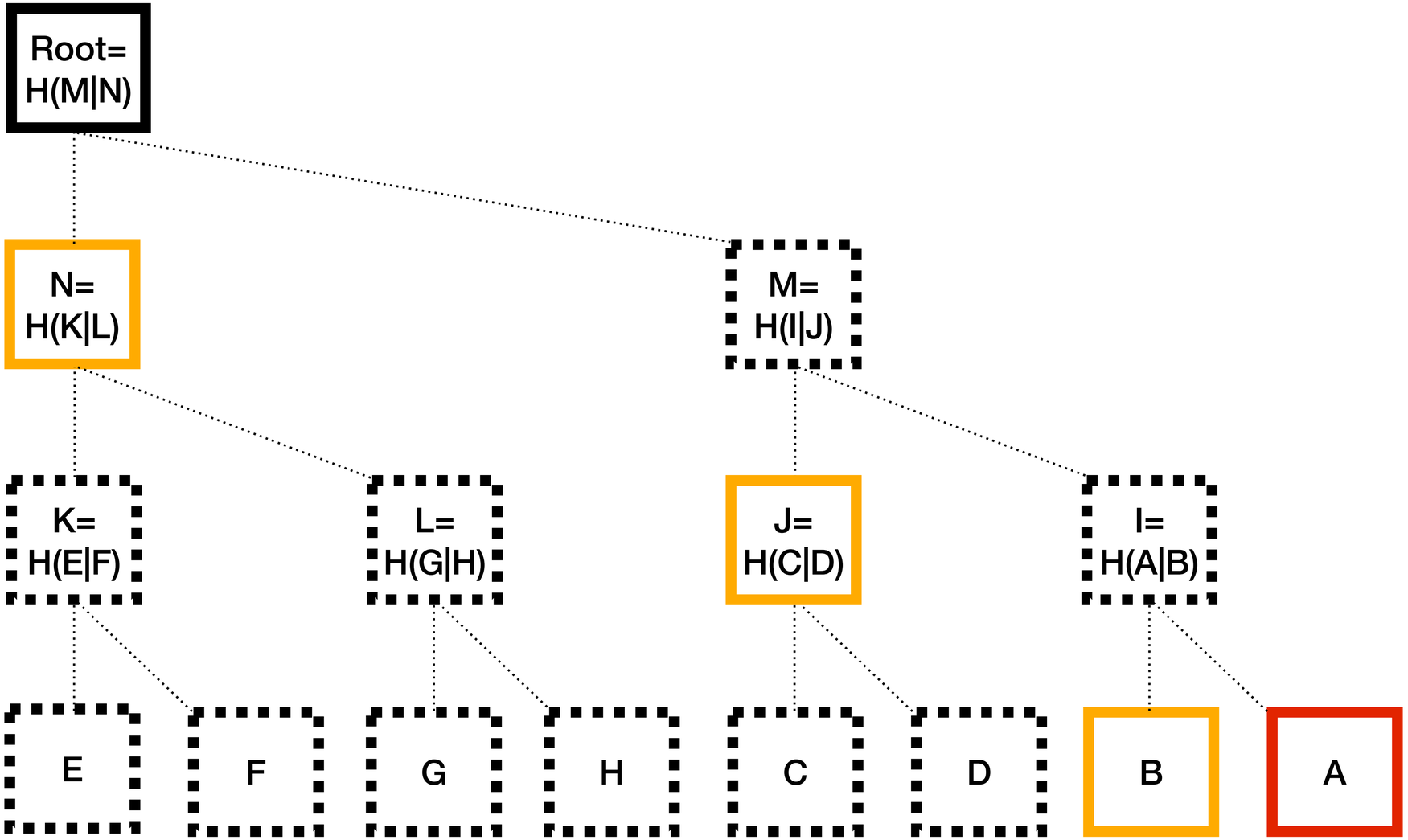
Swap nodes on row 1



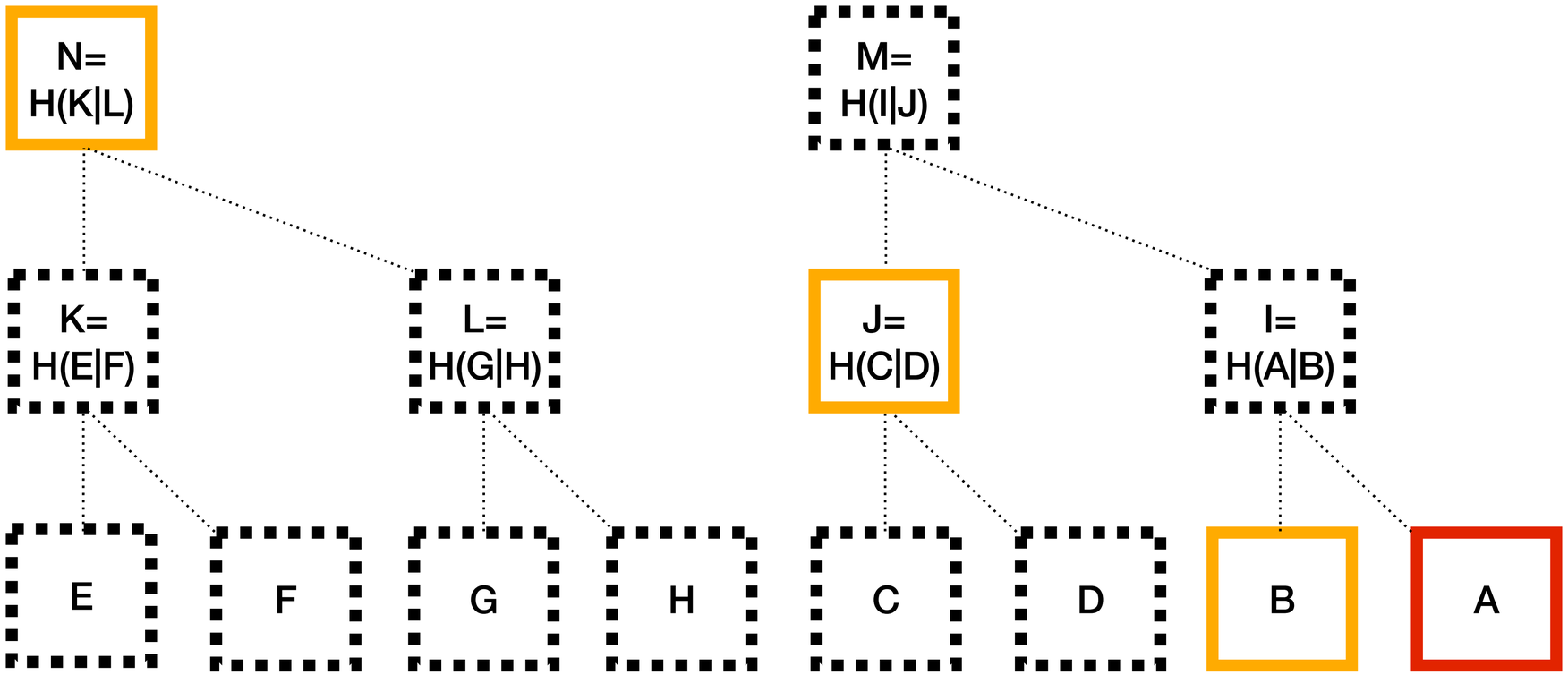
Swap nodes on row 2



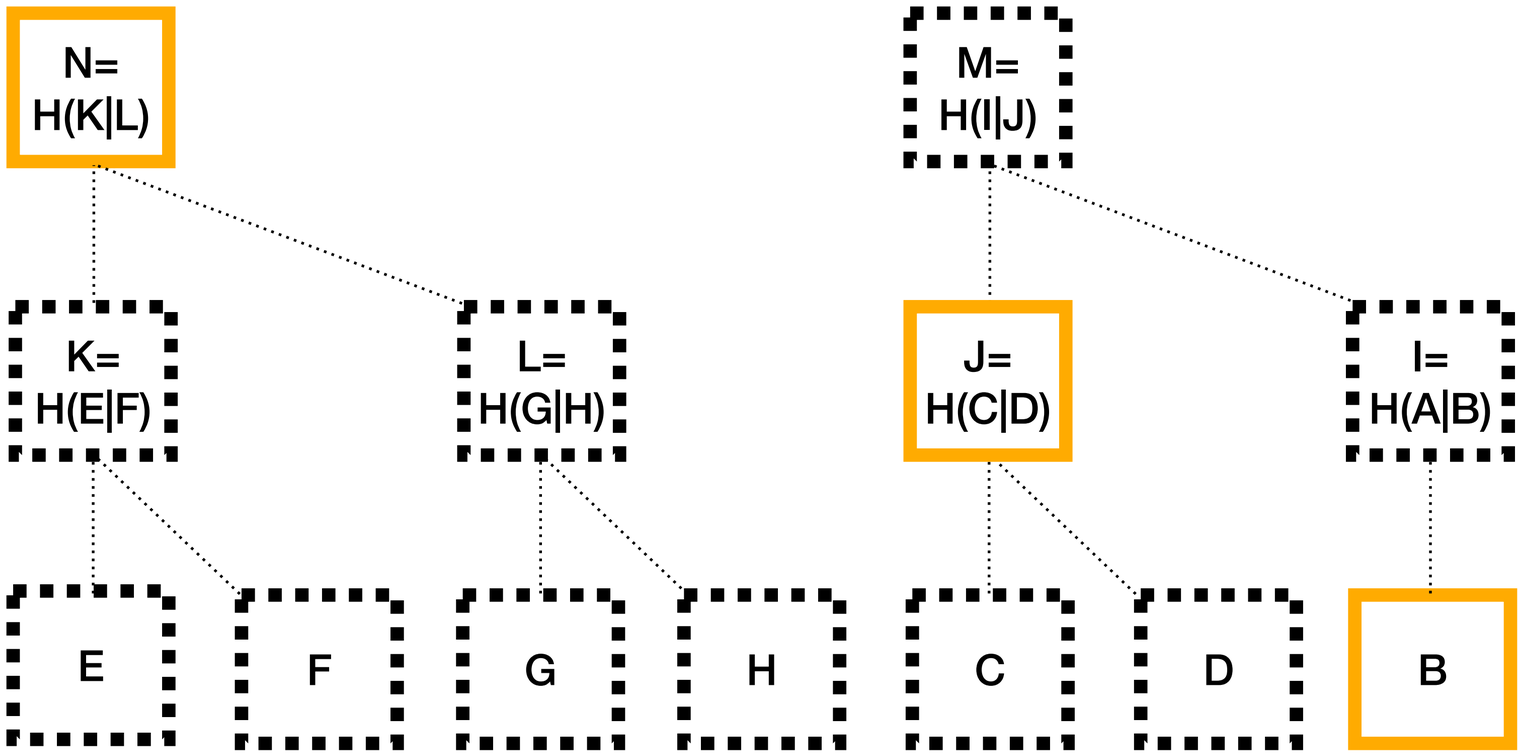
Remove unnecessary nodes



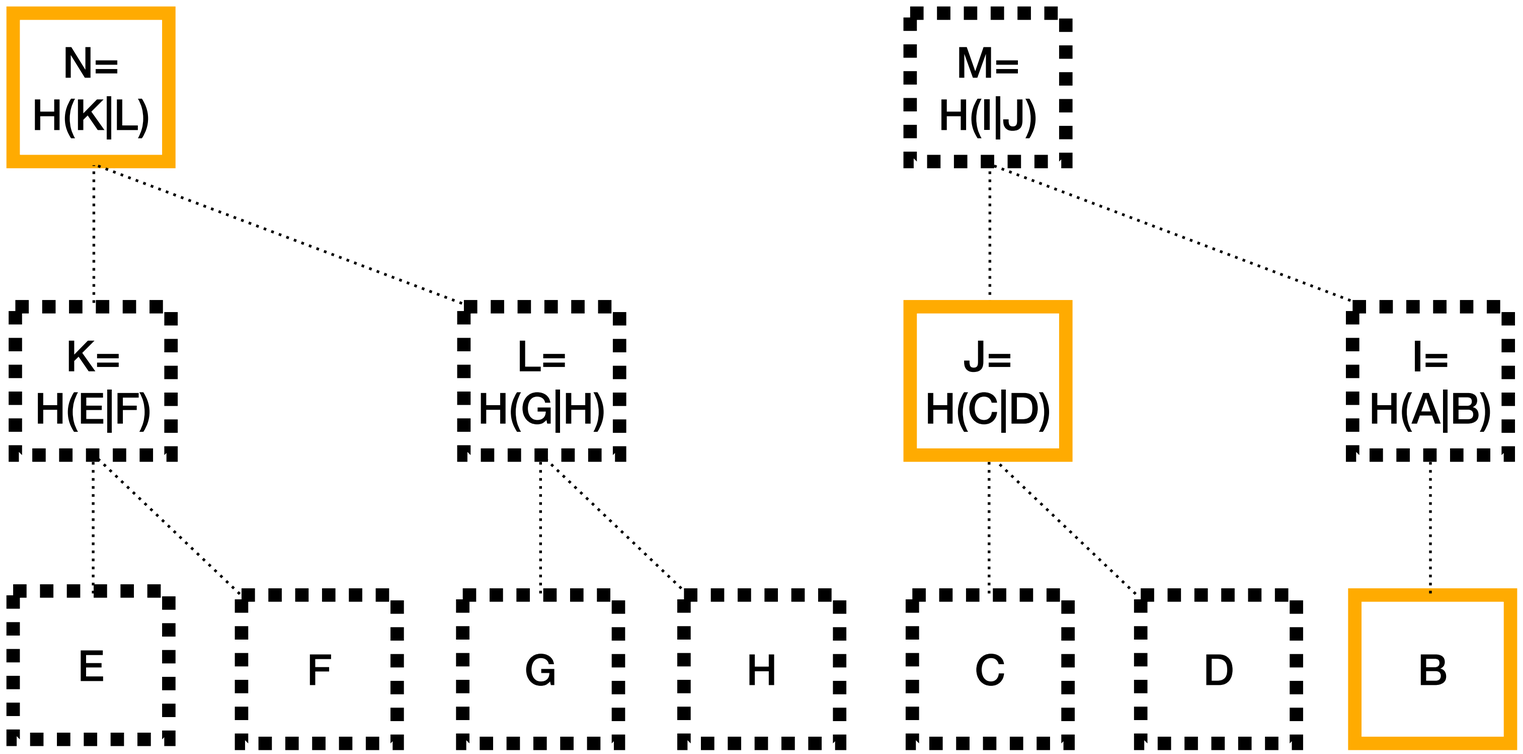
Remove unnecessary nodes



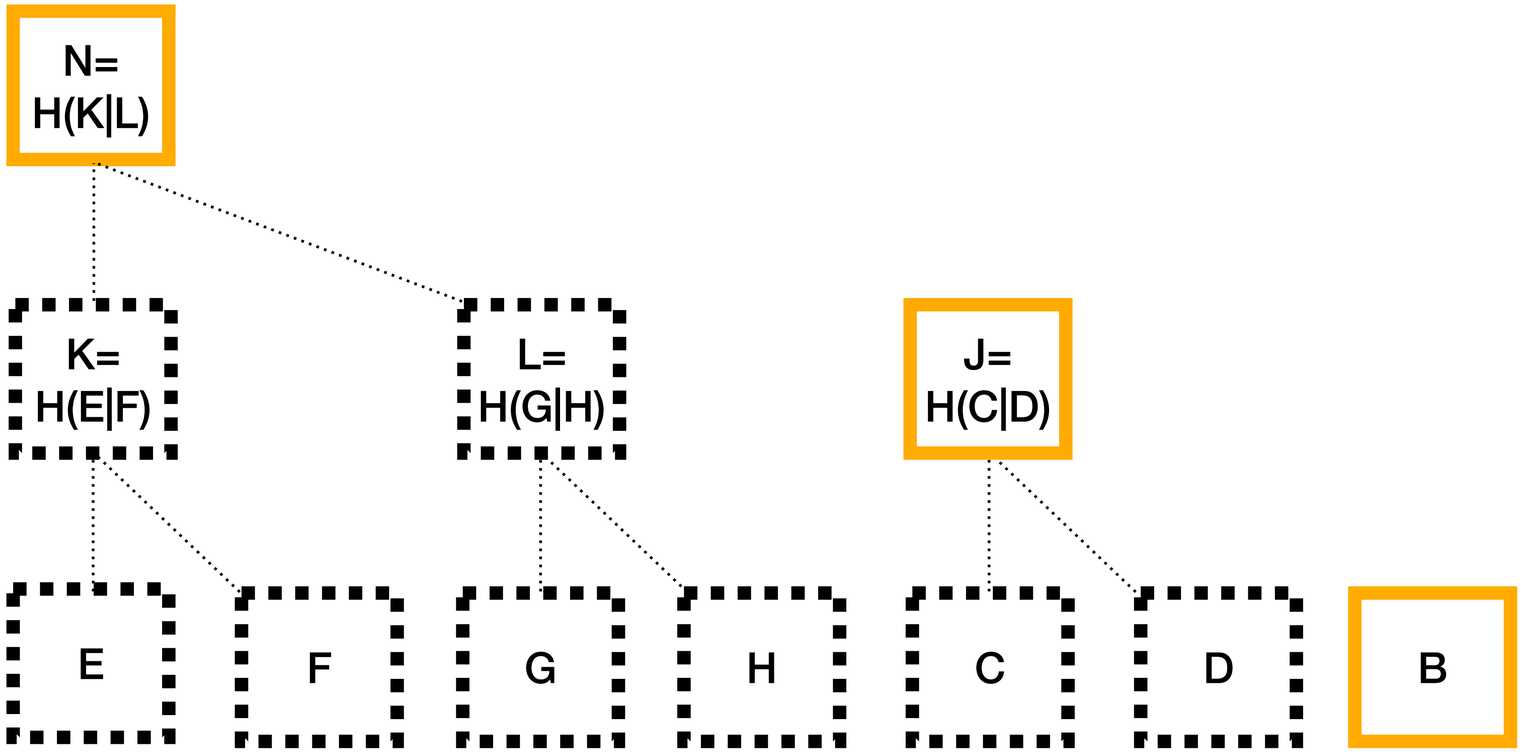
Remove unnecessary nodes



We don't have M&I



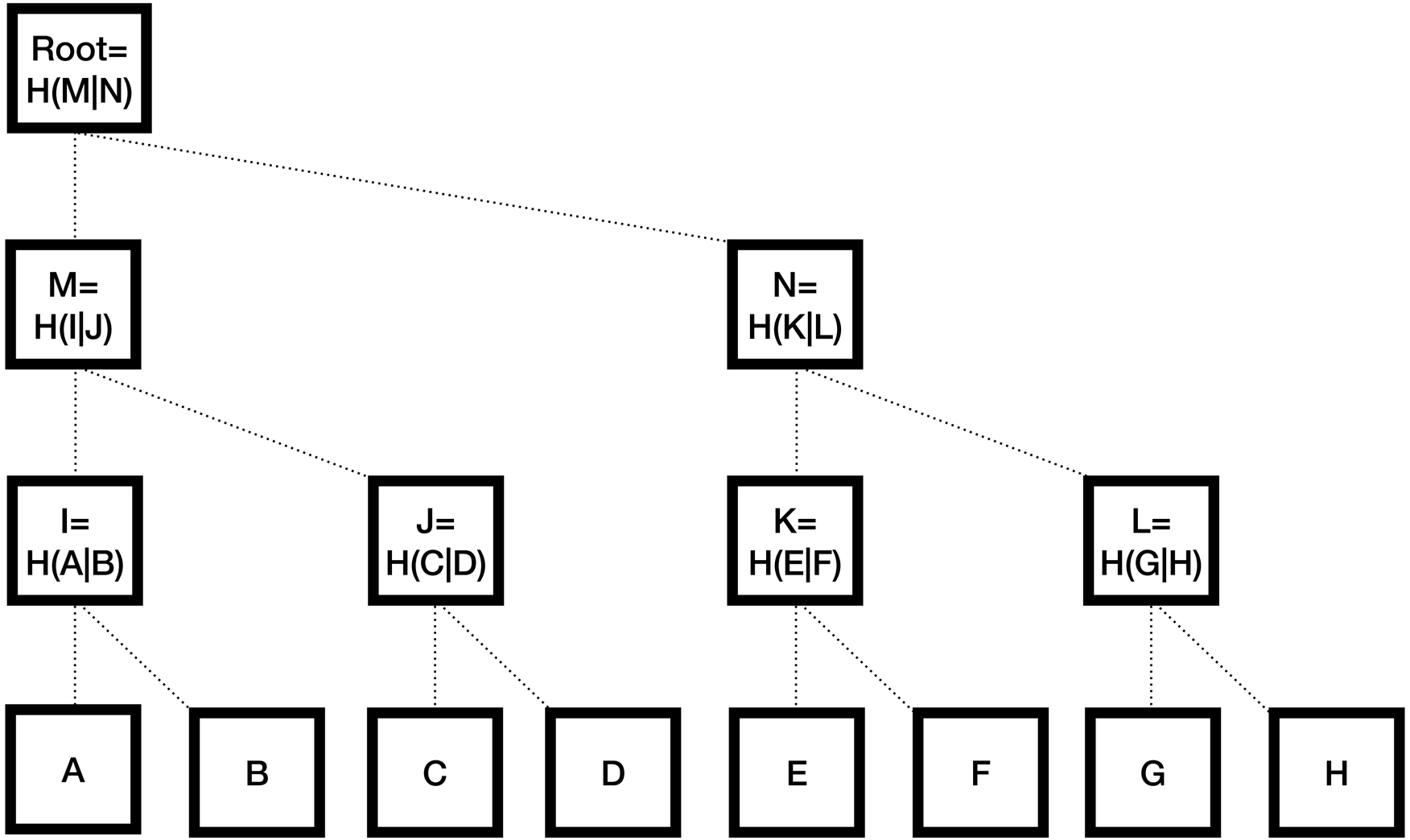
Finished tree



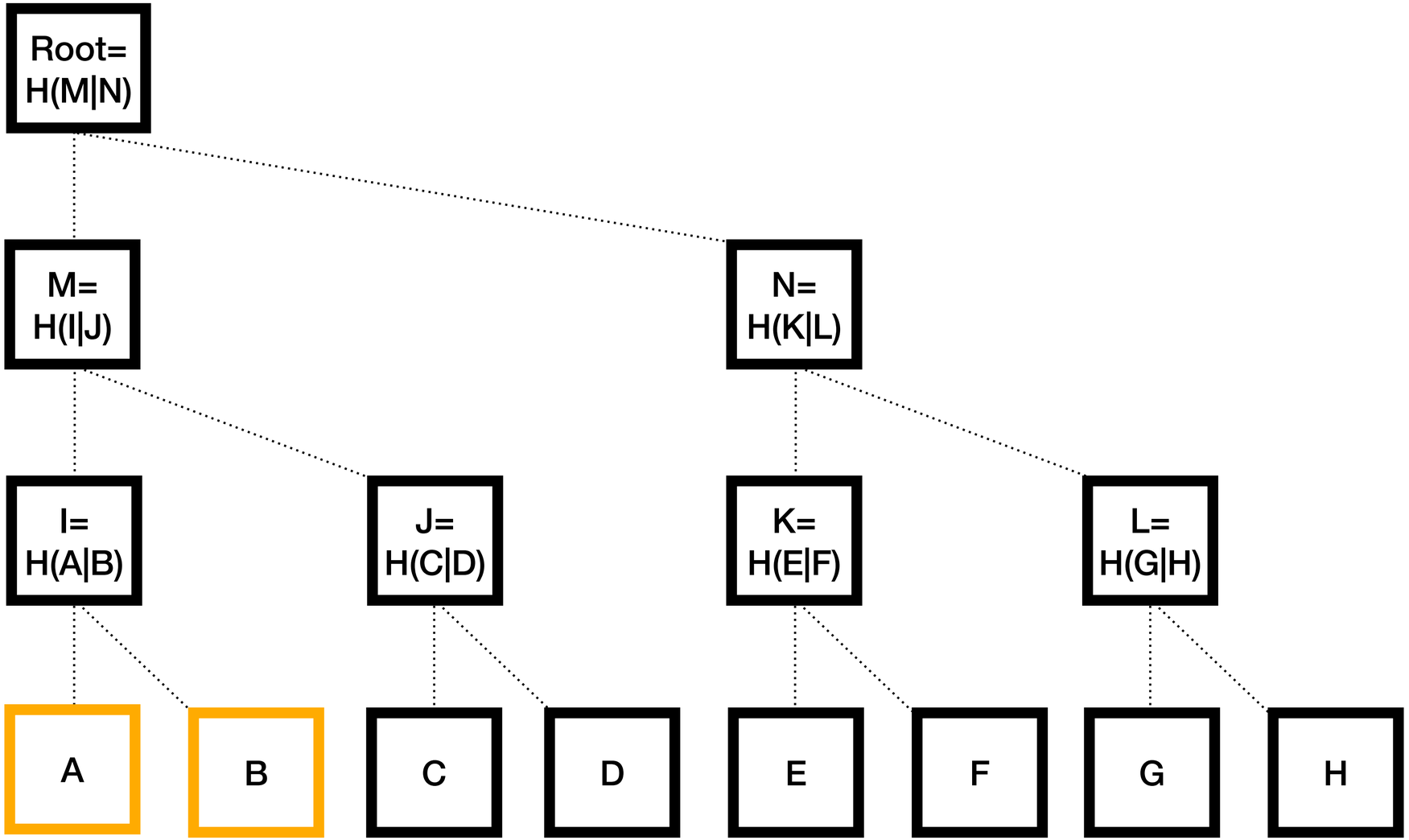
Bigger proof sizes

Problem with the approach used in the Utreexo paper

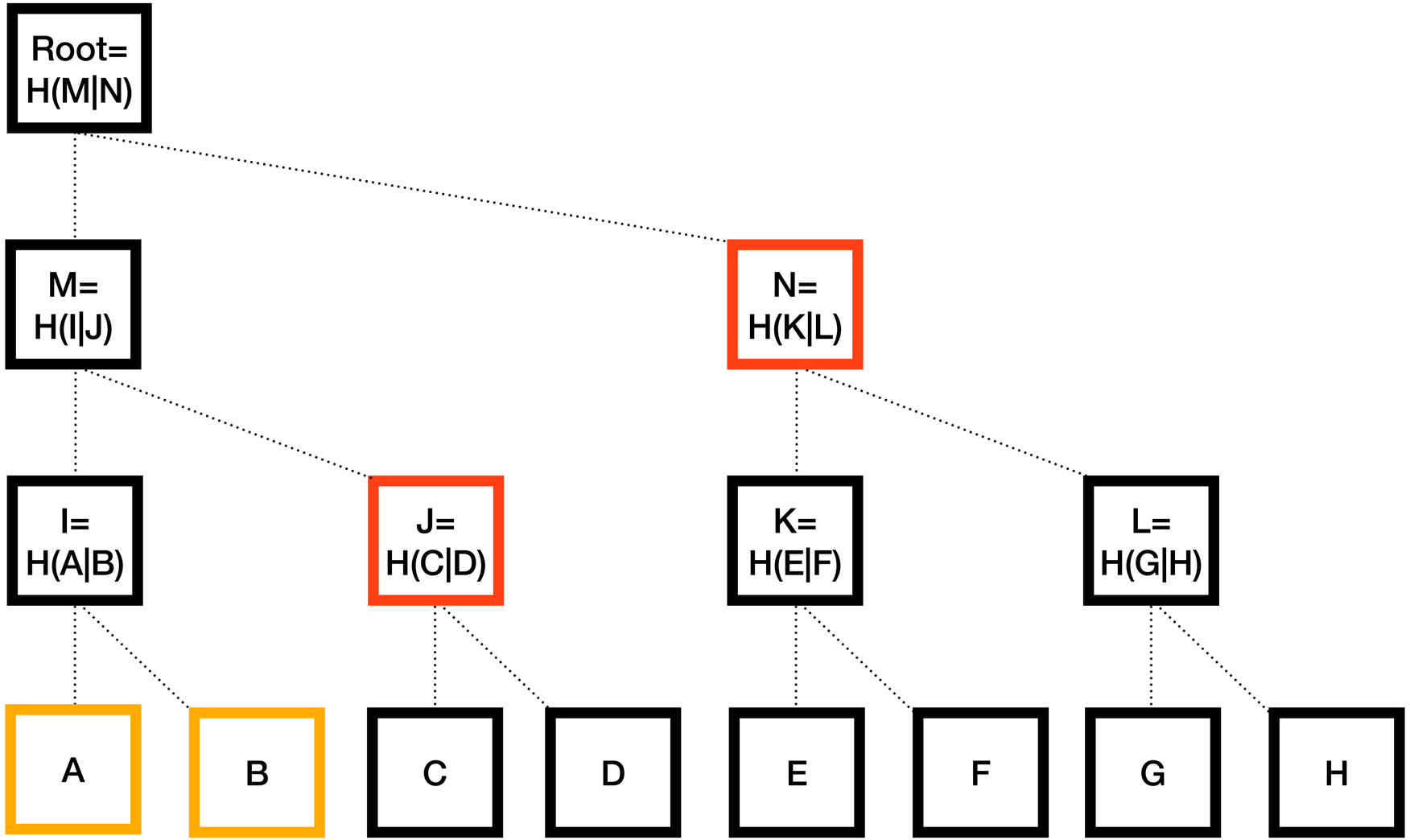
Proving elements



A&B



J&N needed as proof



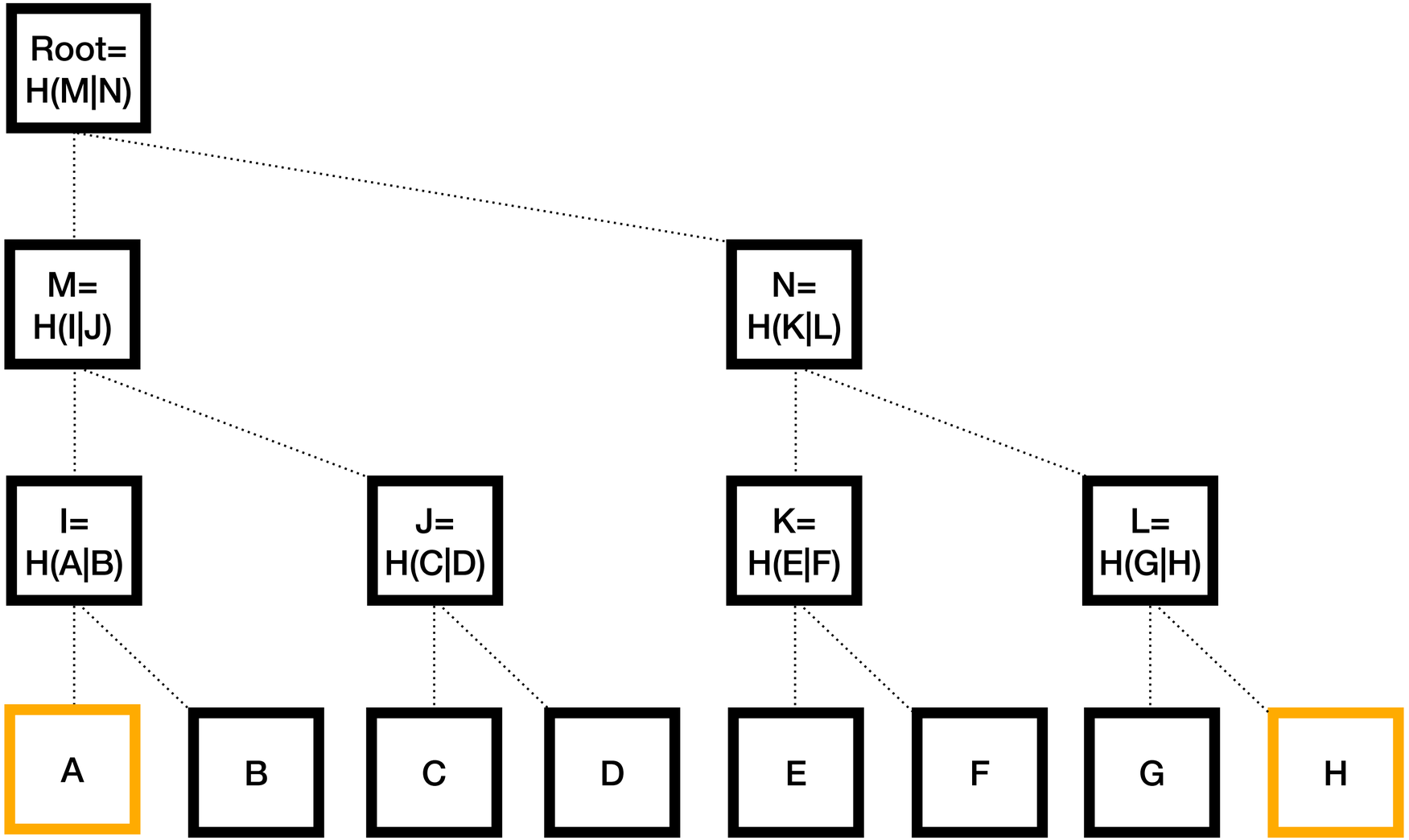
Total size

- A&B are calculable from the Bitcoin Block
- J&N need to be provided as extra data

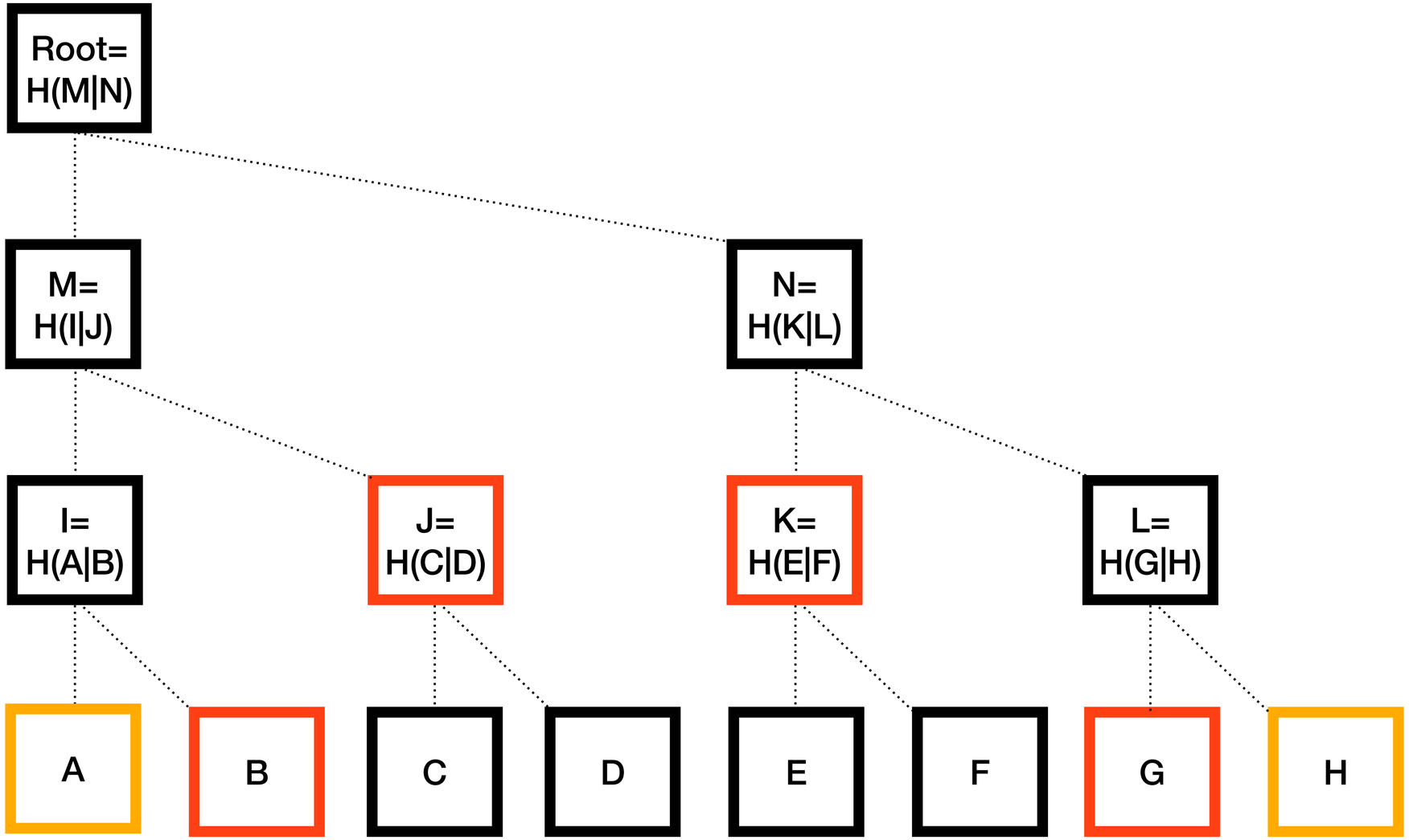
Total size

- Proving A&B are 64 bytes of proof data

A&H

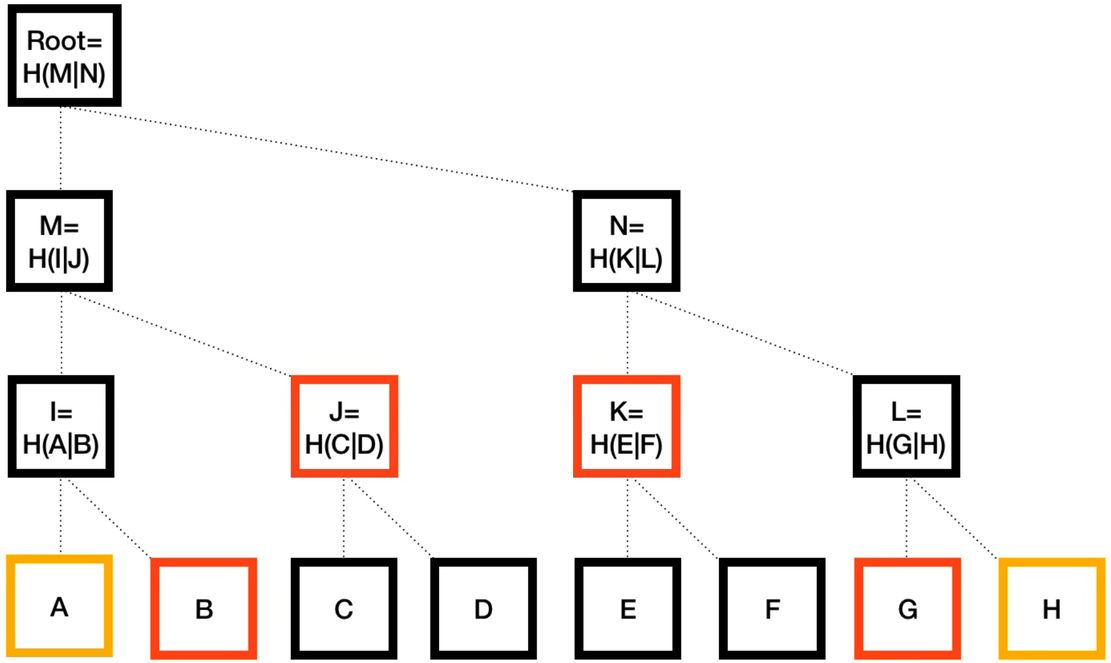


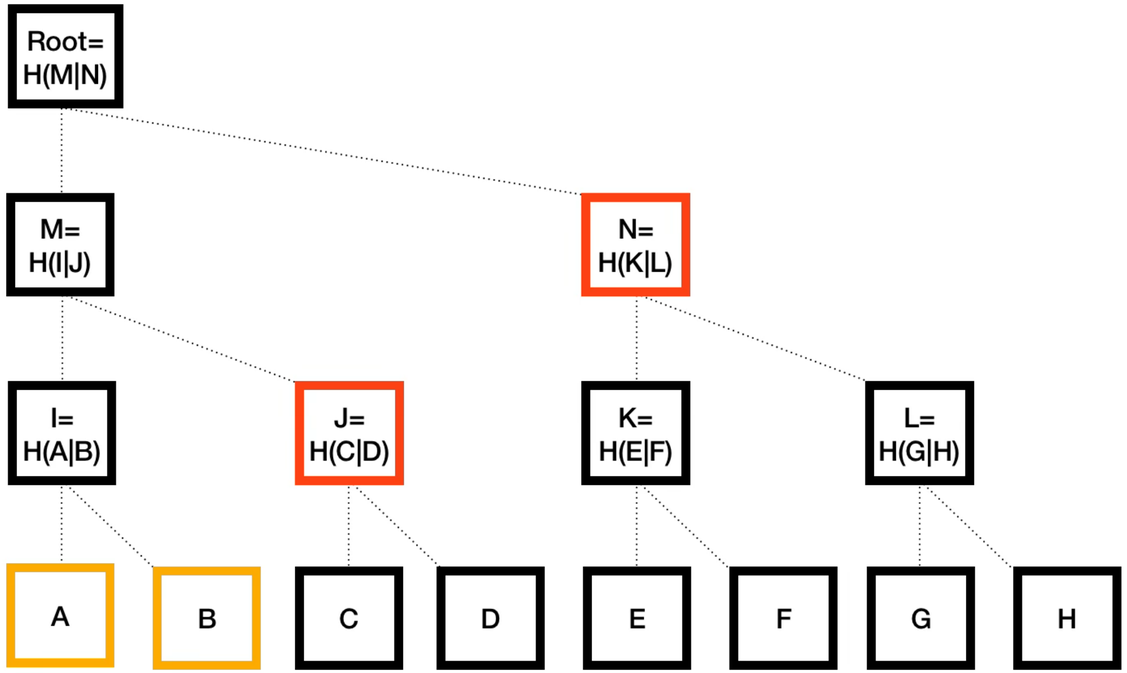
B, G, J, K needed as proof



Total size

- Proving A&H are 128 bytes of proof data

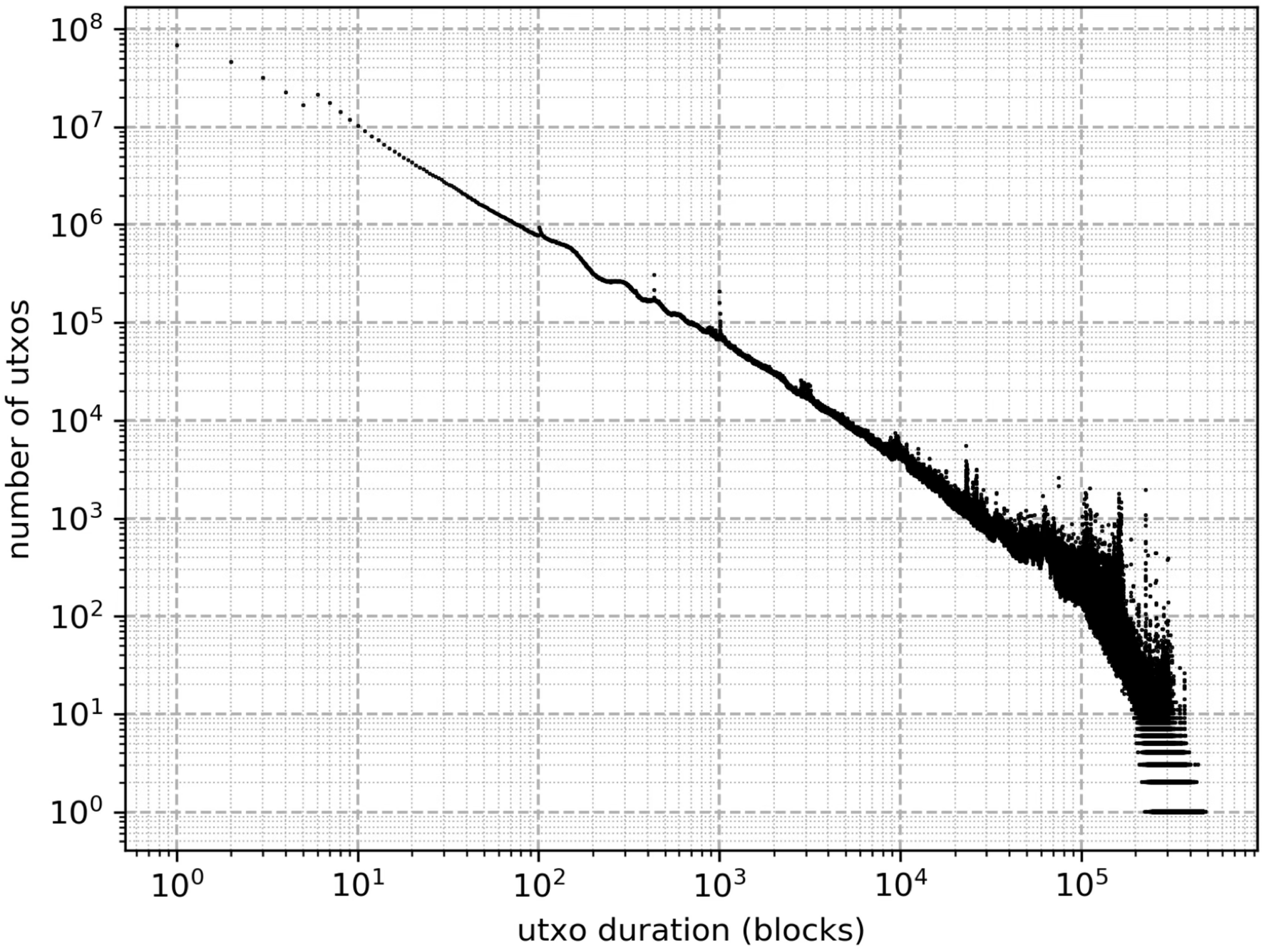

Root=
H(M|N)
M=
H(I|J)
N=
H(K|L)
I=
H(A|B)
J=
H(C|D)
K=
H(E|F)
L=
H(G|H)
A
B
C
D
E
F
G
H


Root=
H(M|N)
M=
H(I|J)
N=
H(K|L)
I=
H(A|B)
J=
H(C|D)
K=
H(E|F)
L=
H(G|H)
A
B
C
D
E
F
G
H

locality matters

For reducing proof sizes

Newer UTXOs are more likely to be spent



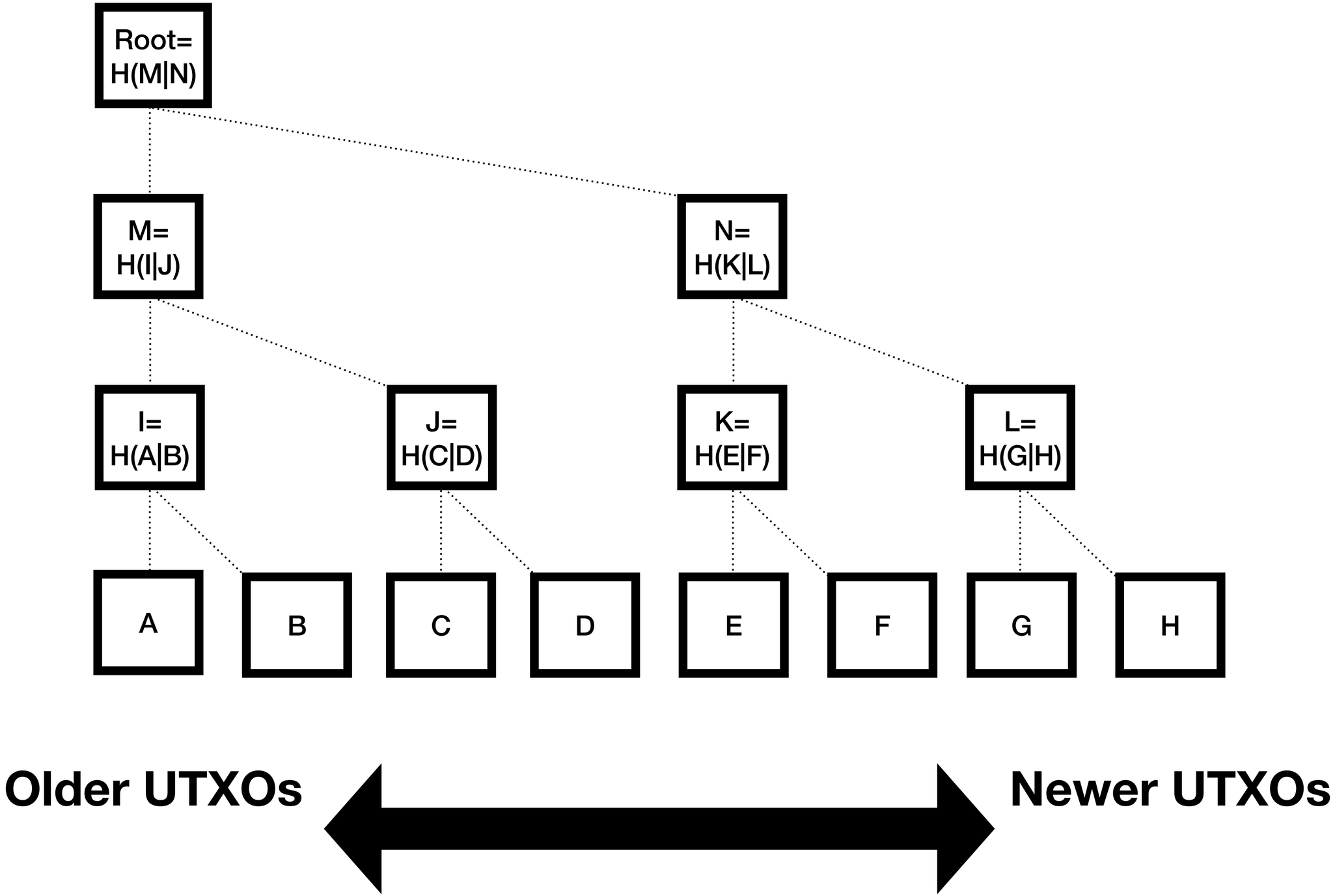
https://eprint.iacr.org/2019/611.pdf

Swapping is bad

For reducing proof sizes

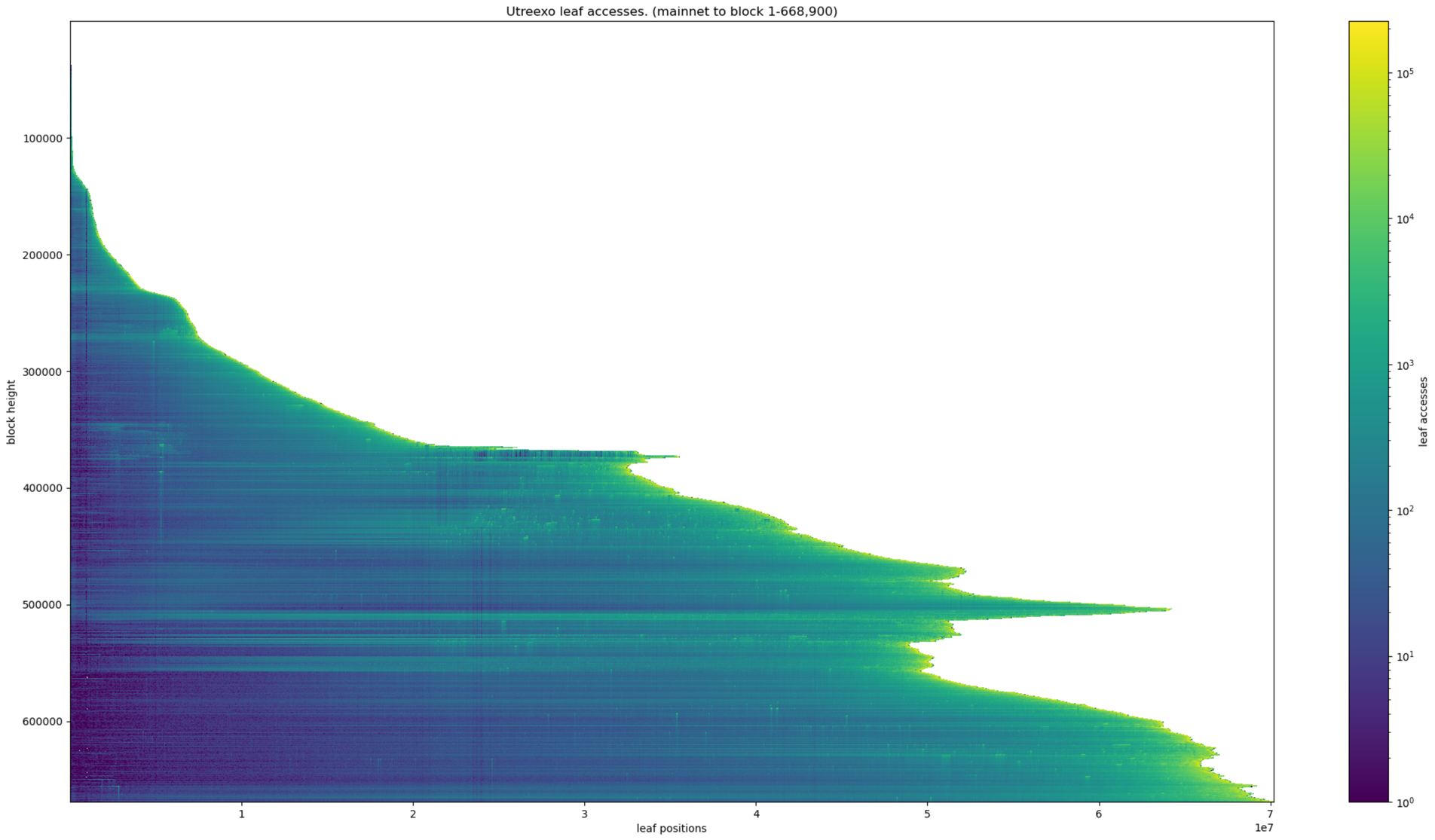
Utreexo additions

- Since additions are appended, newer UTXOs stay to the right

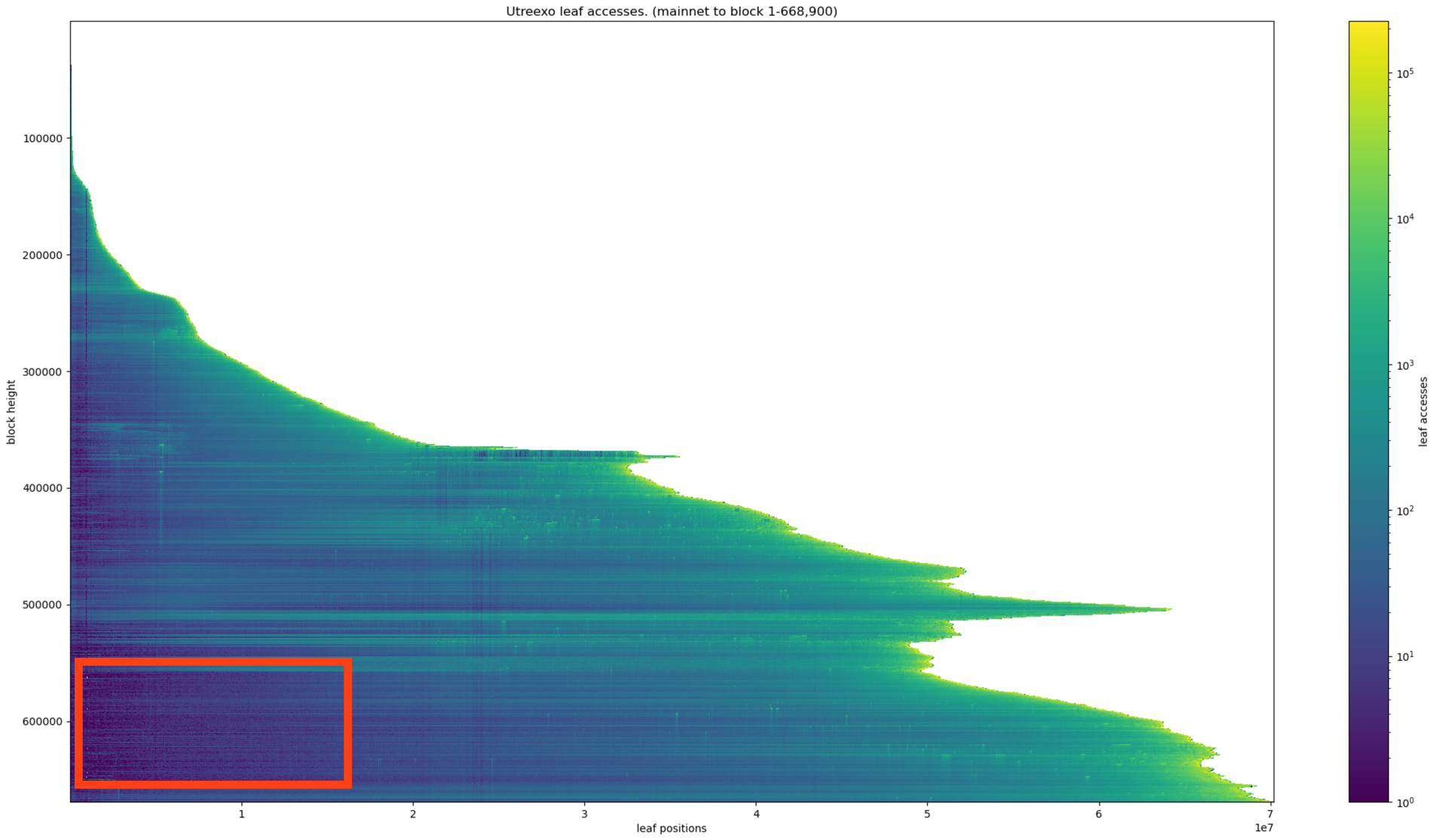

Root=
H(M|N)
M=
H(I|J)
N=
H(K|L)
I=
H(A|B)
J=
H(C|D)
K=
H(E|F)
L=
H(G|H)
A
B
C
D
E
F
G
H
Older UTXOs
Newer UTXOs

Utreexo additions

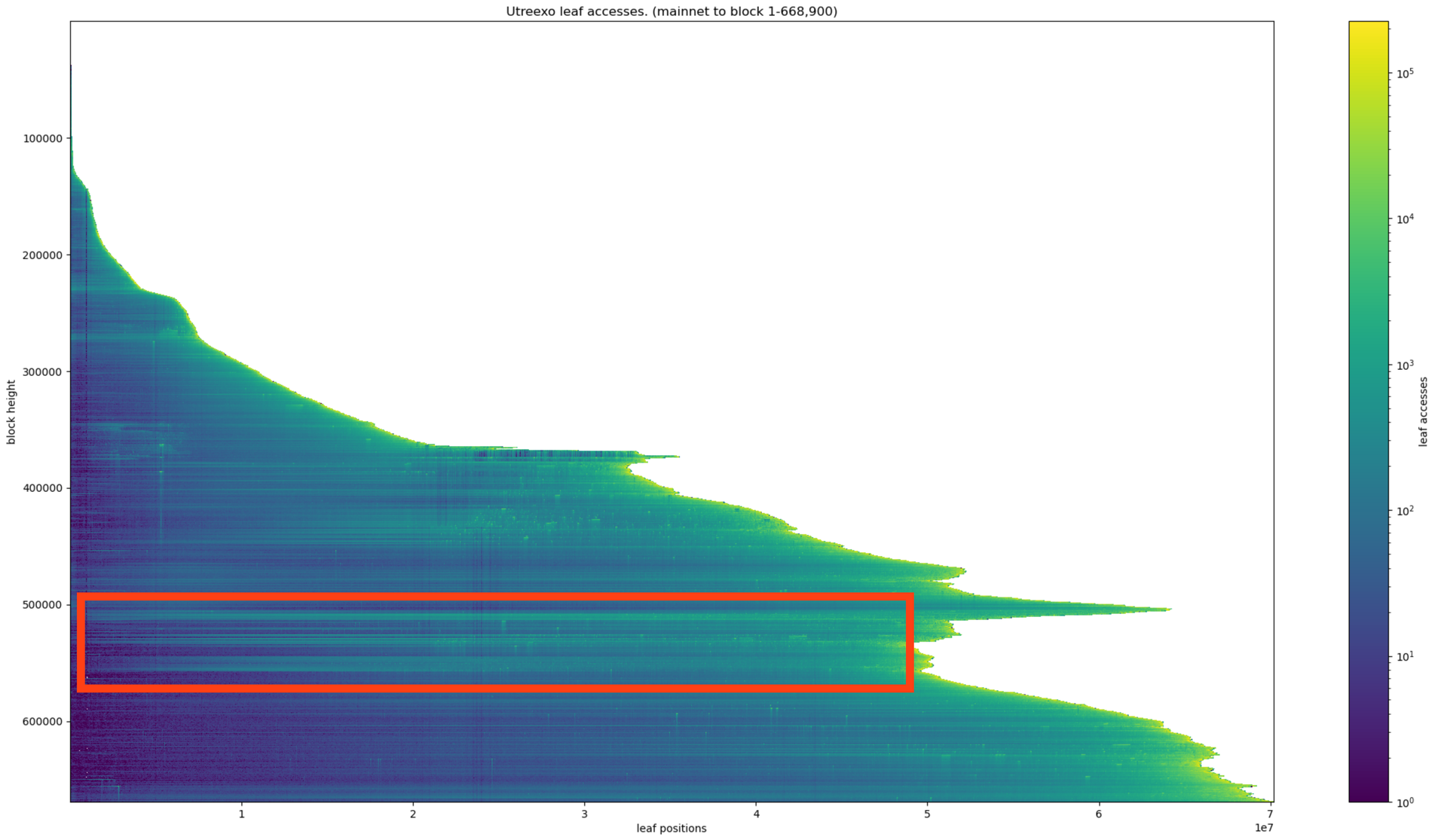
- Since additions are appended, newer UTXOs stay to the right
- Swapping mixes up newer UTXOs with older UTXOs



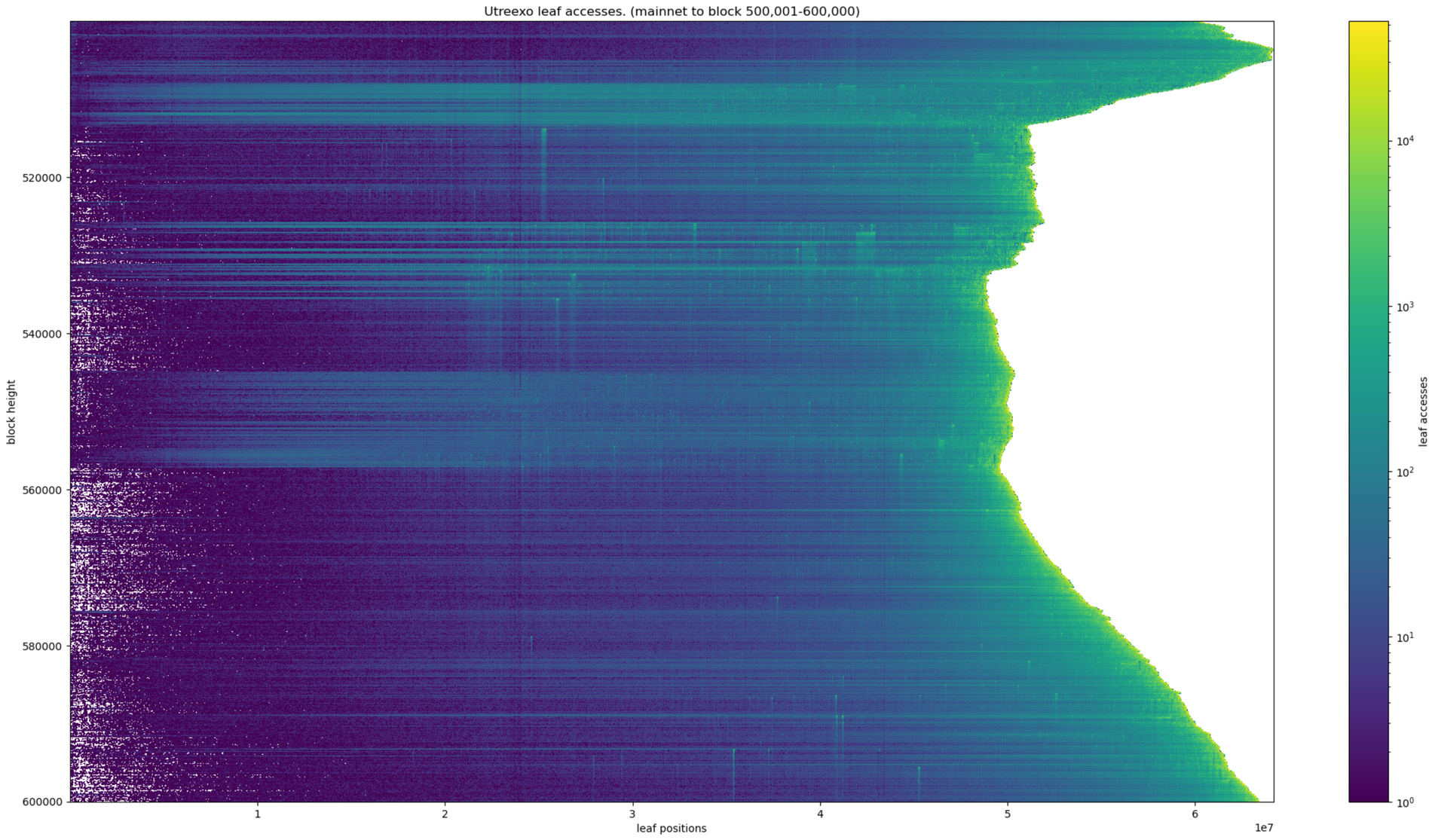
https://github.com/mit-dci/utreexo/issues/257



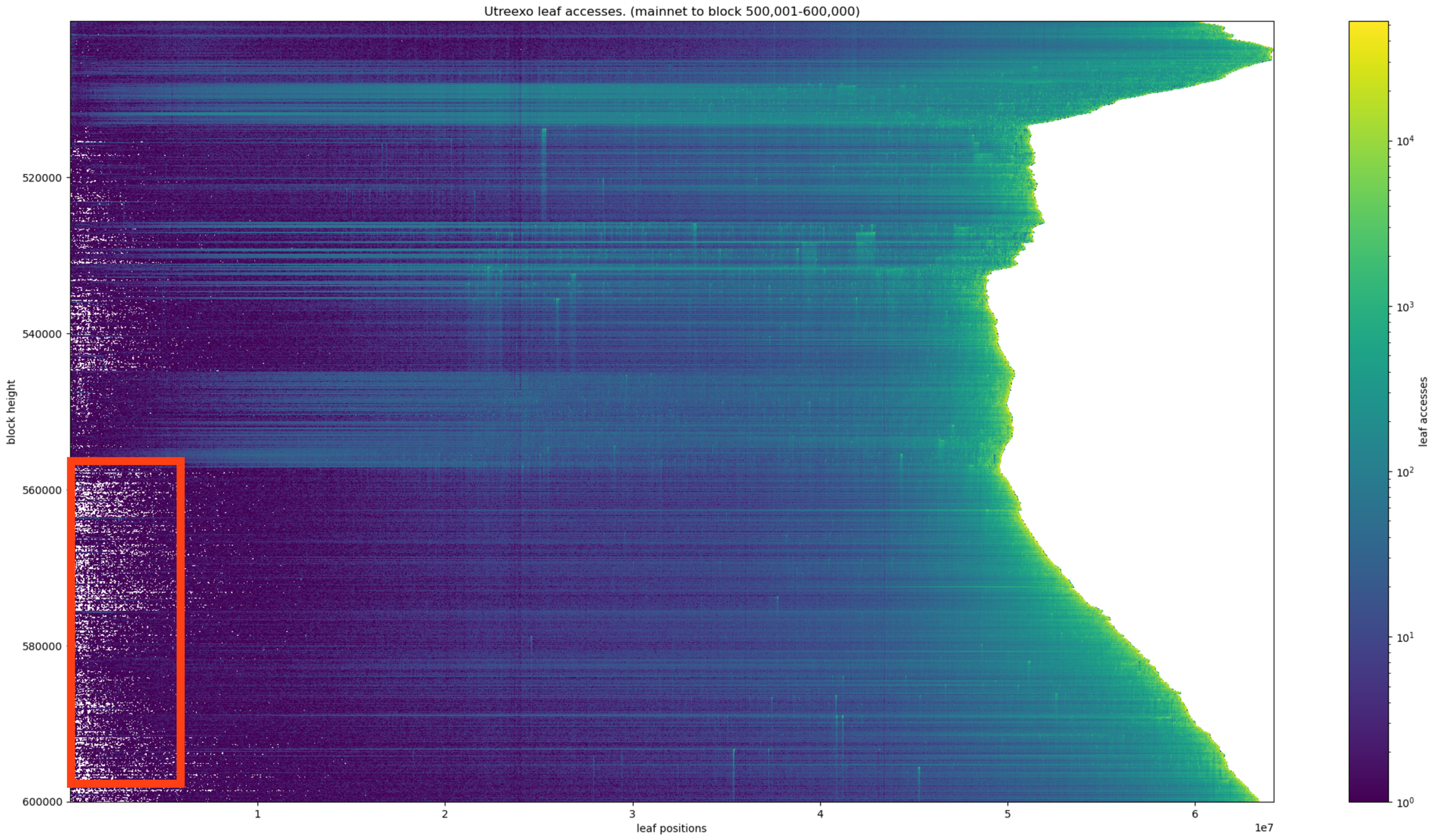
https://github.com/mit-dci/utreexo/issues/257



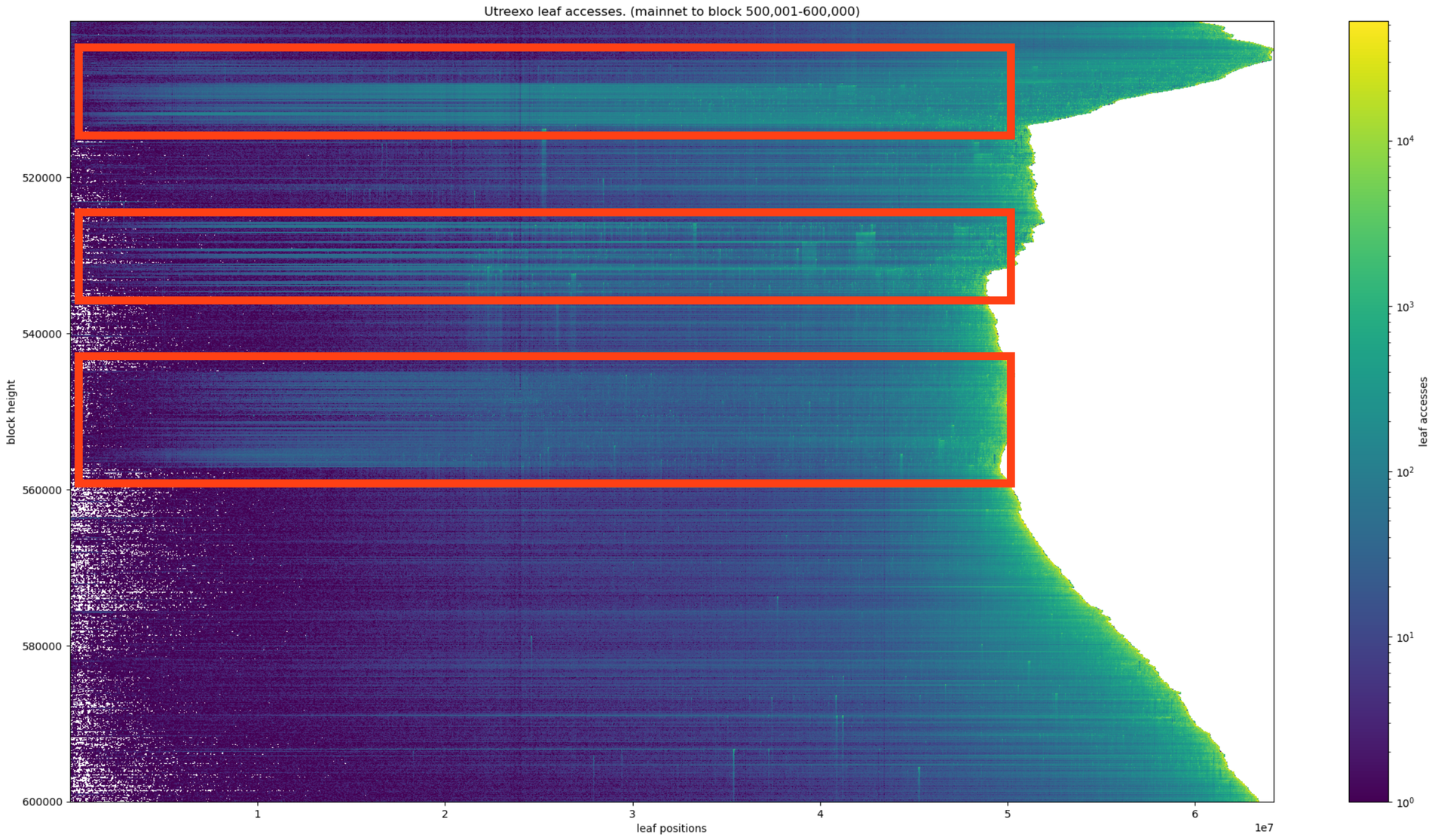
https://github.com/mit-dci/utreexo/issues/257



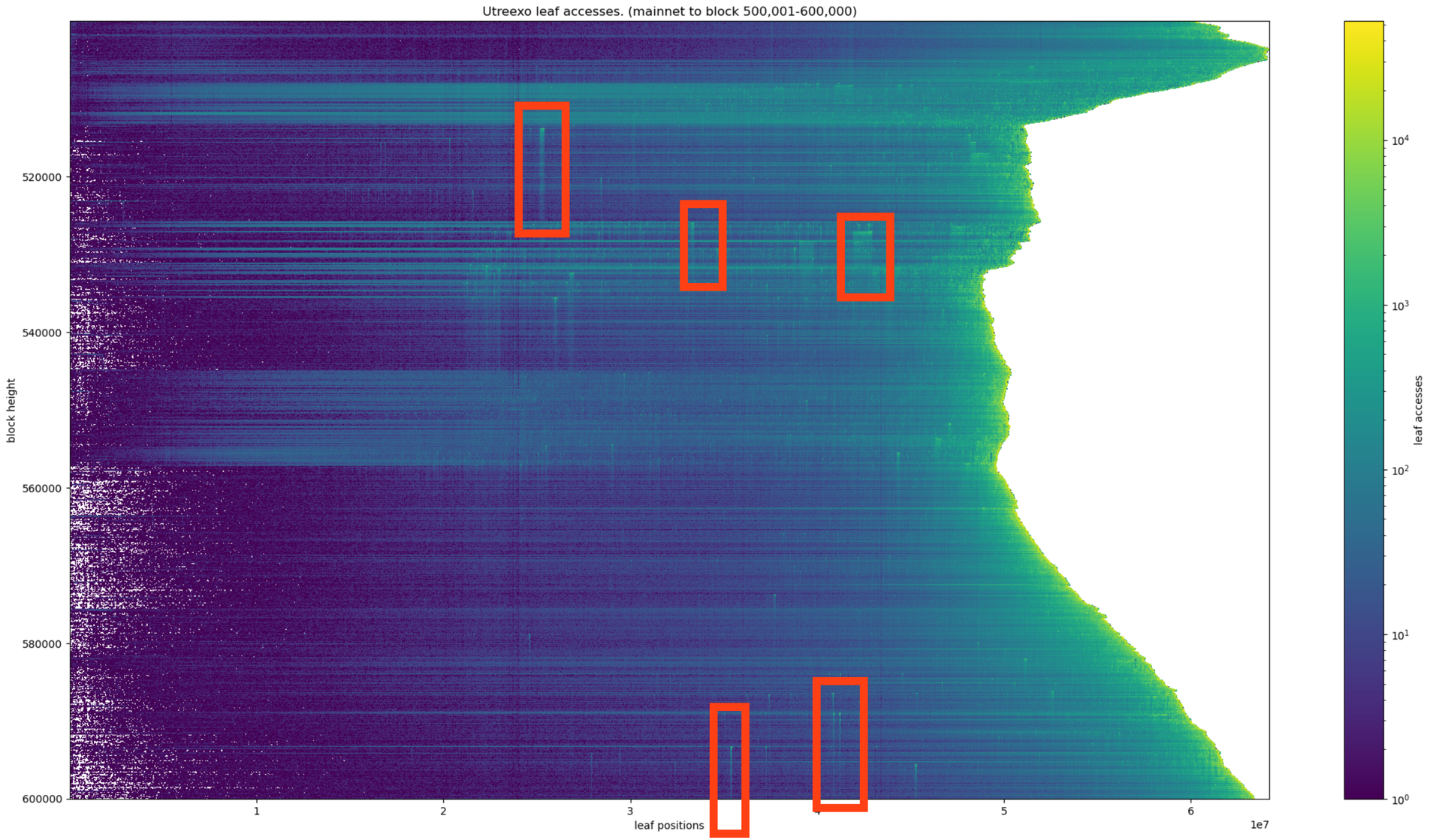
https://github.com/mit-dci/utreexo/issues/257



https://github.com/mit-dci/utreexo/issues/257



https://github.com/mit-dci/utreexo/issues/257

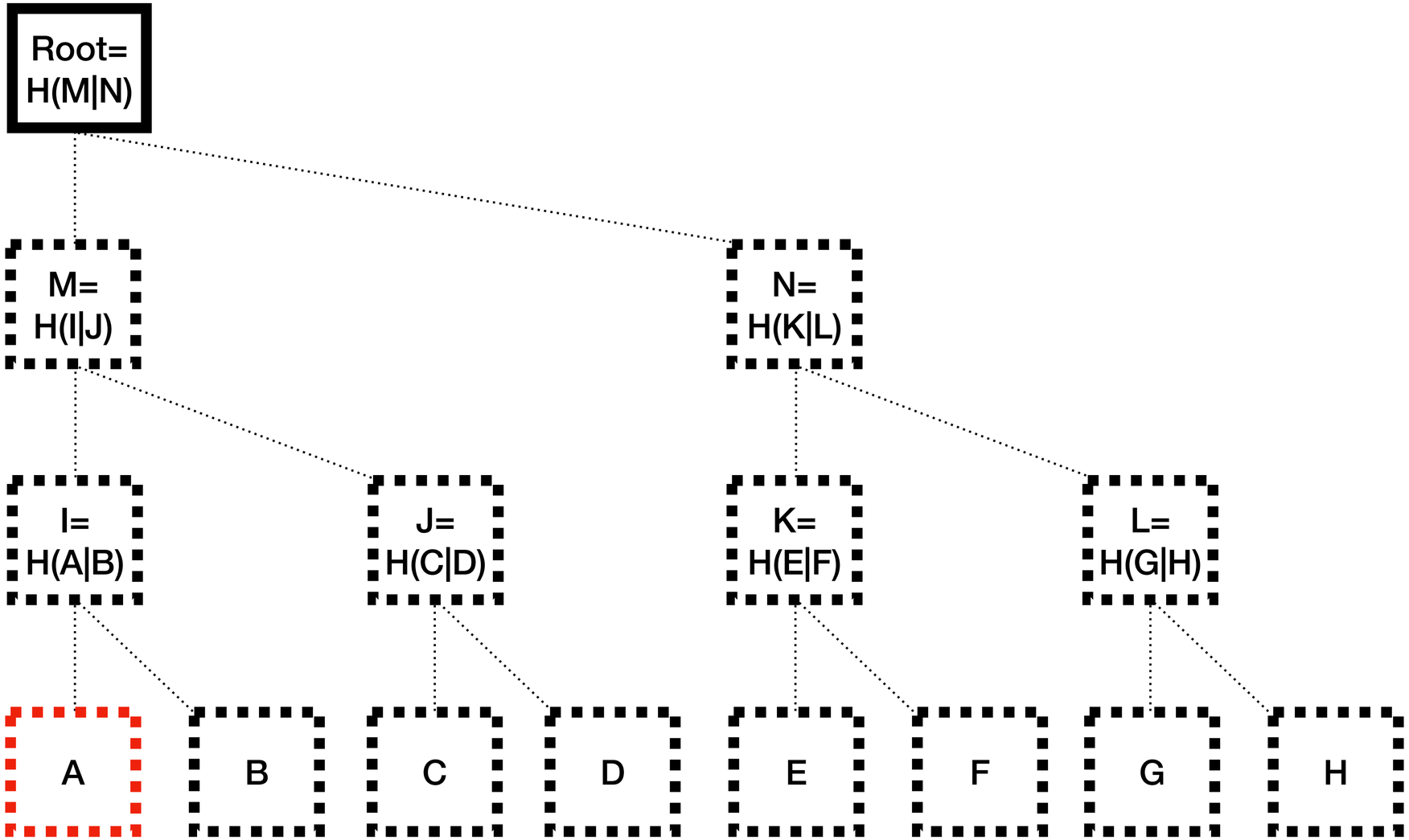


https://github.com/mit-dci/utreexo/issues/257

New algorithm

- Moves elements up

Deleting A



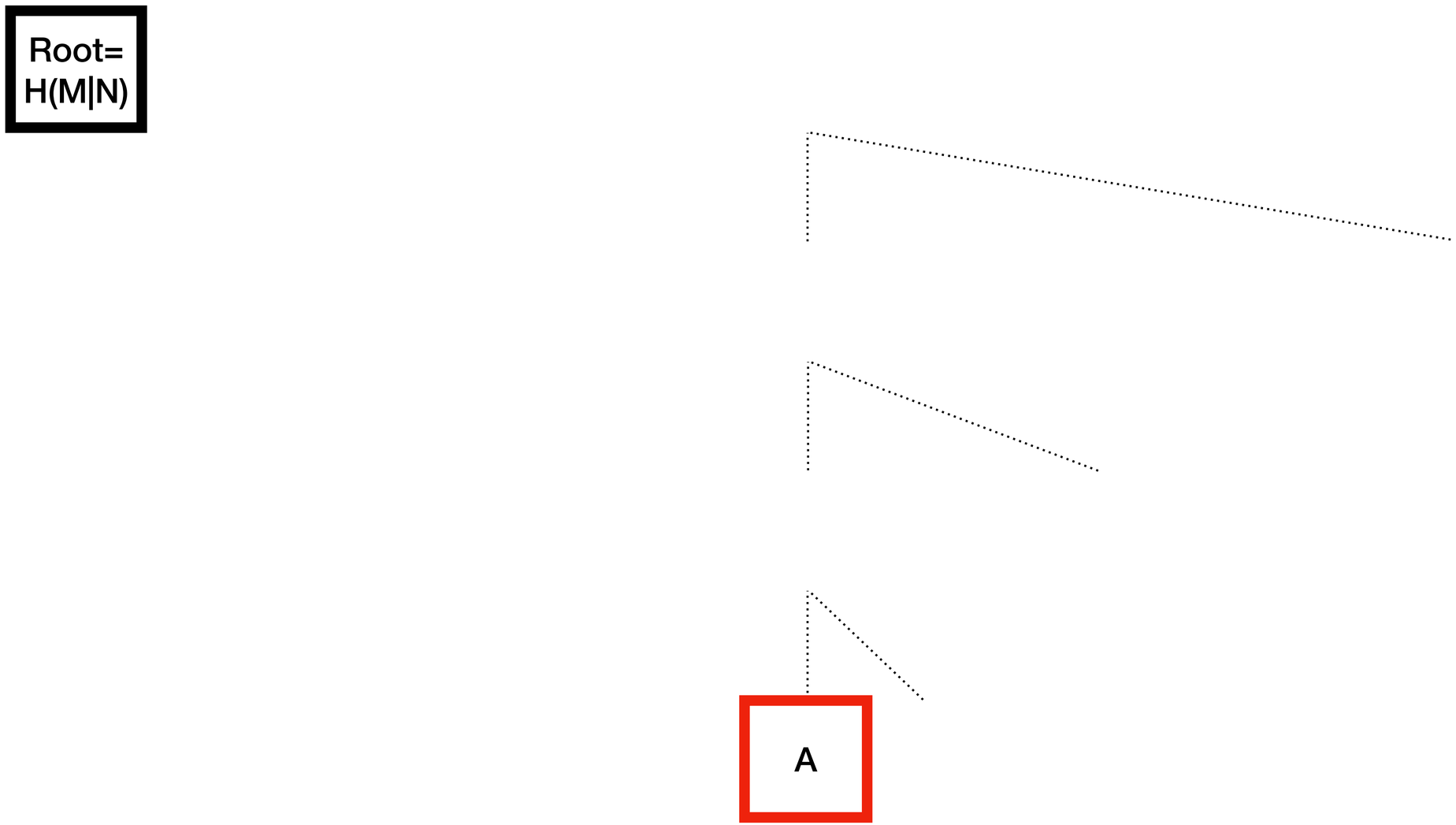
https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

The only requirement is the roots

Root=
H(M|N)

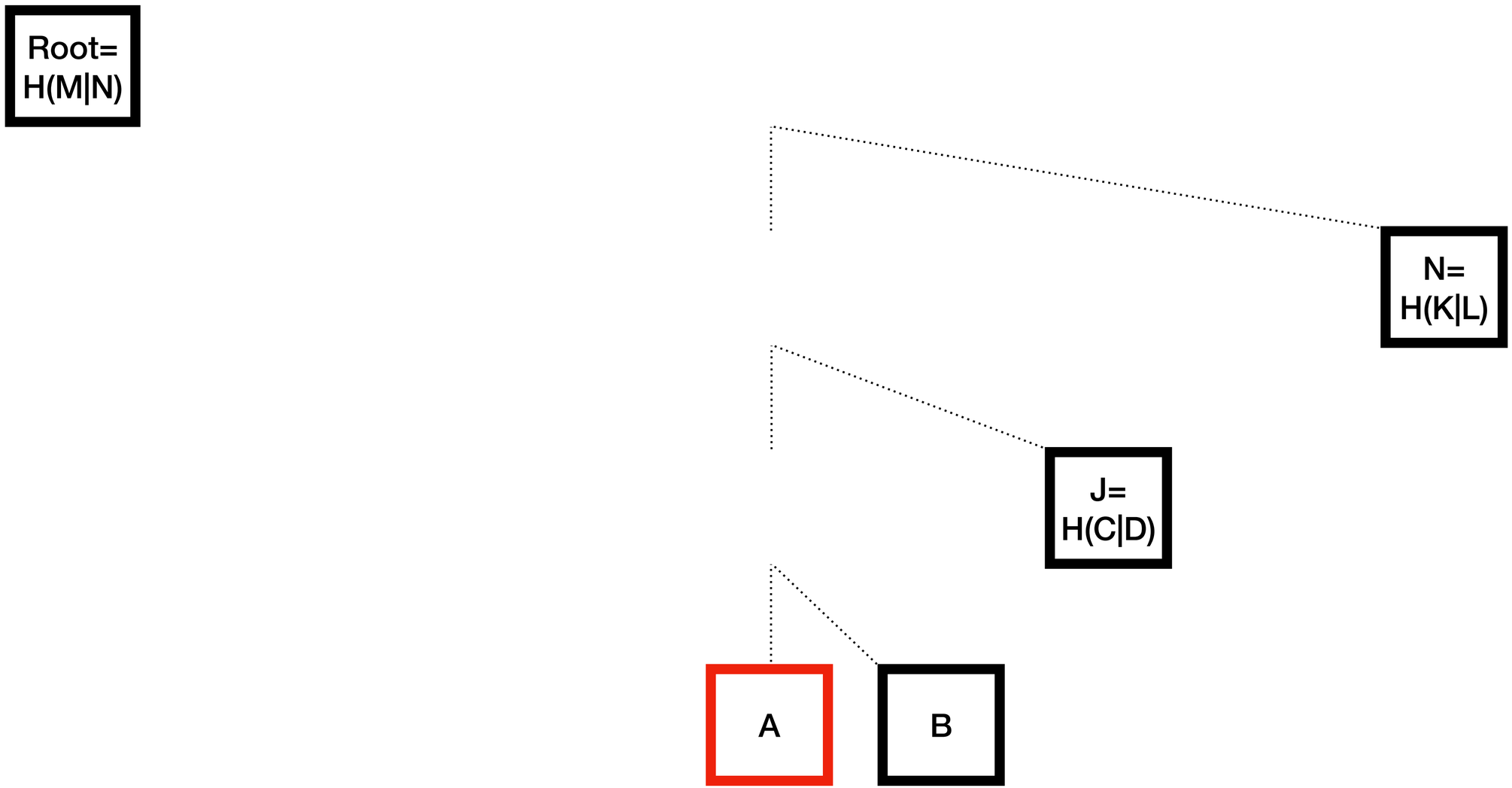
https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

Calculate A from the Bitcoin Block



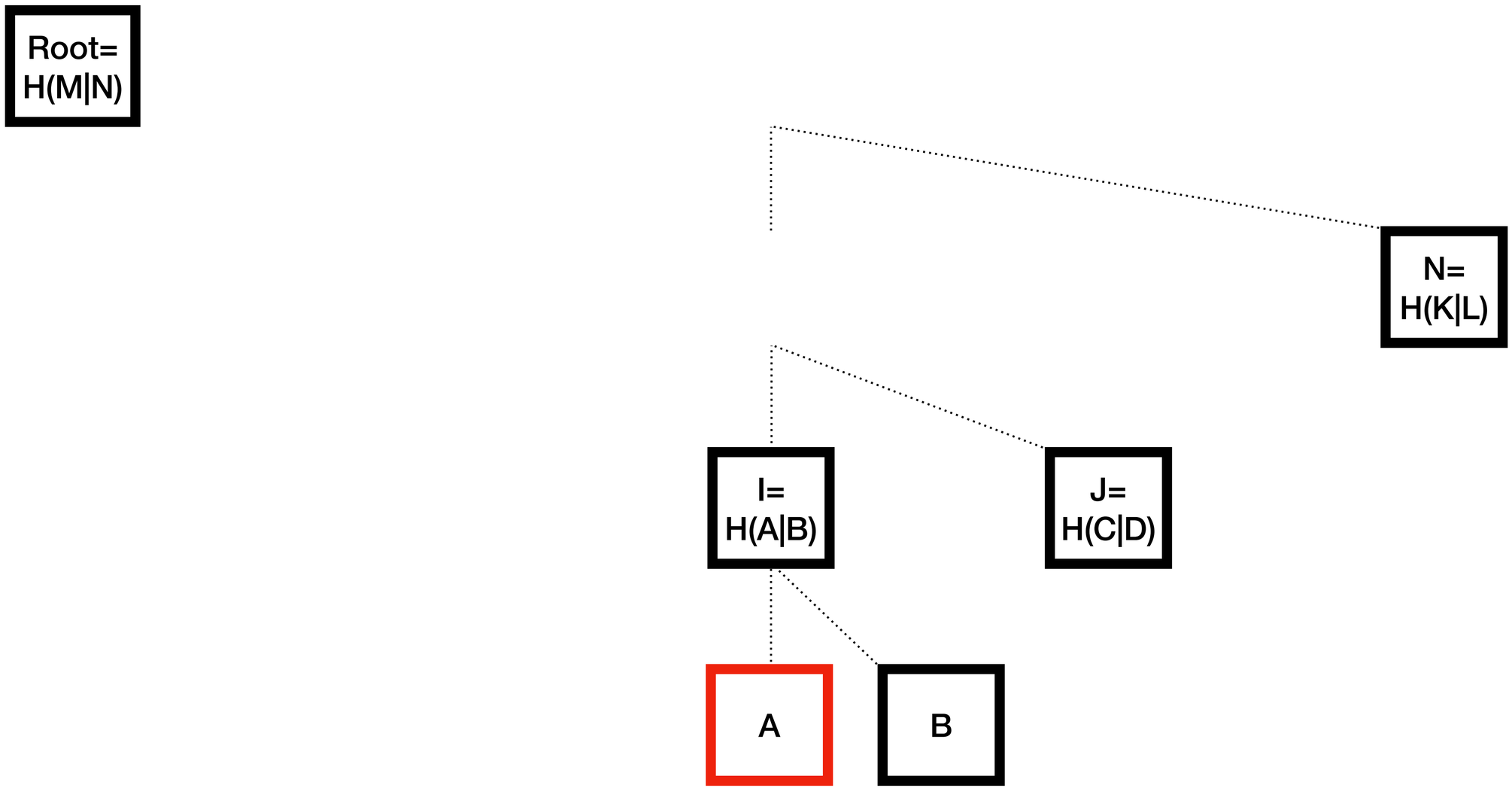
https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

Receive the proof: B, J, N



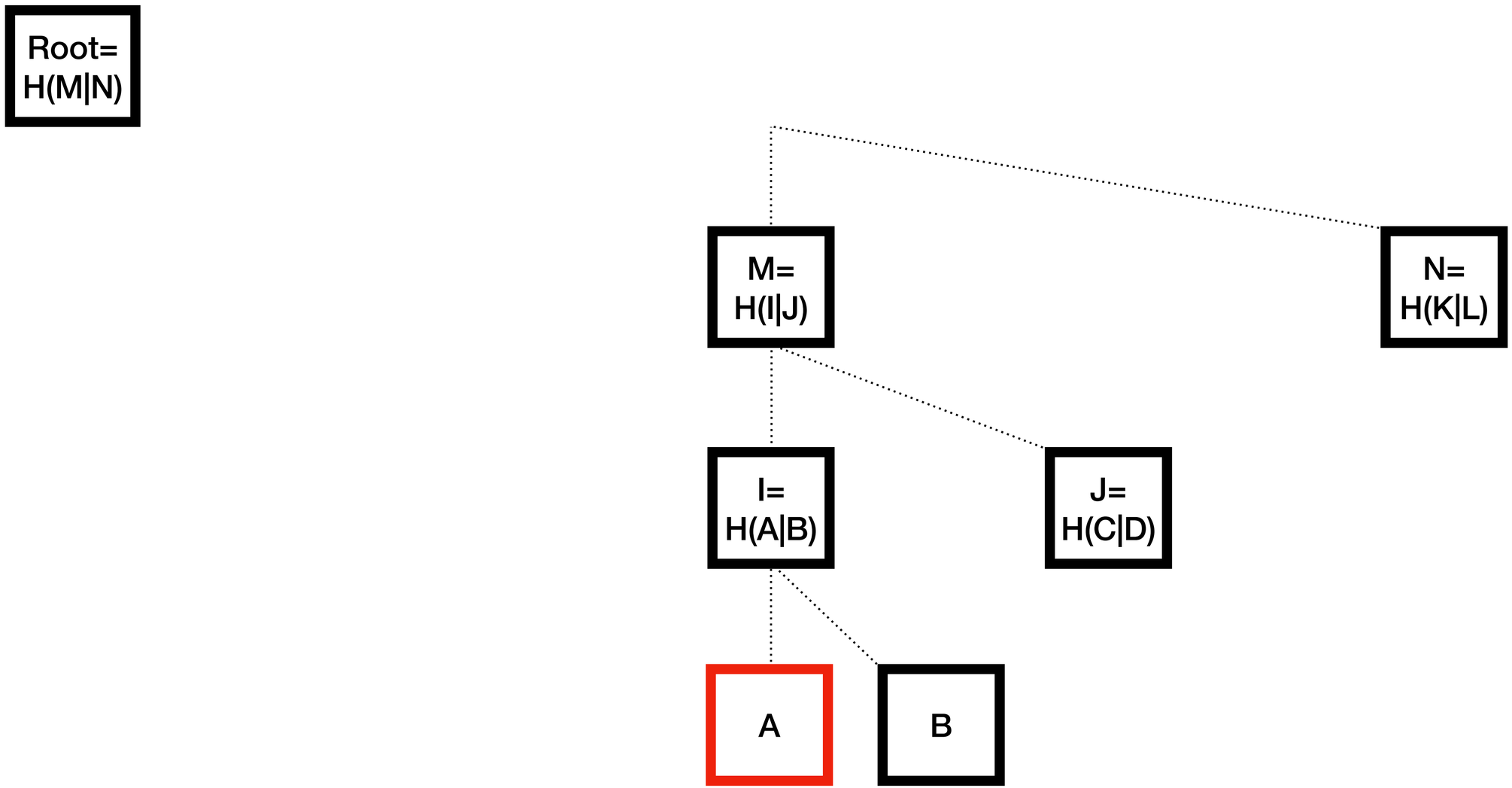
https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

Calculate the hash for I



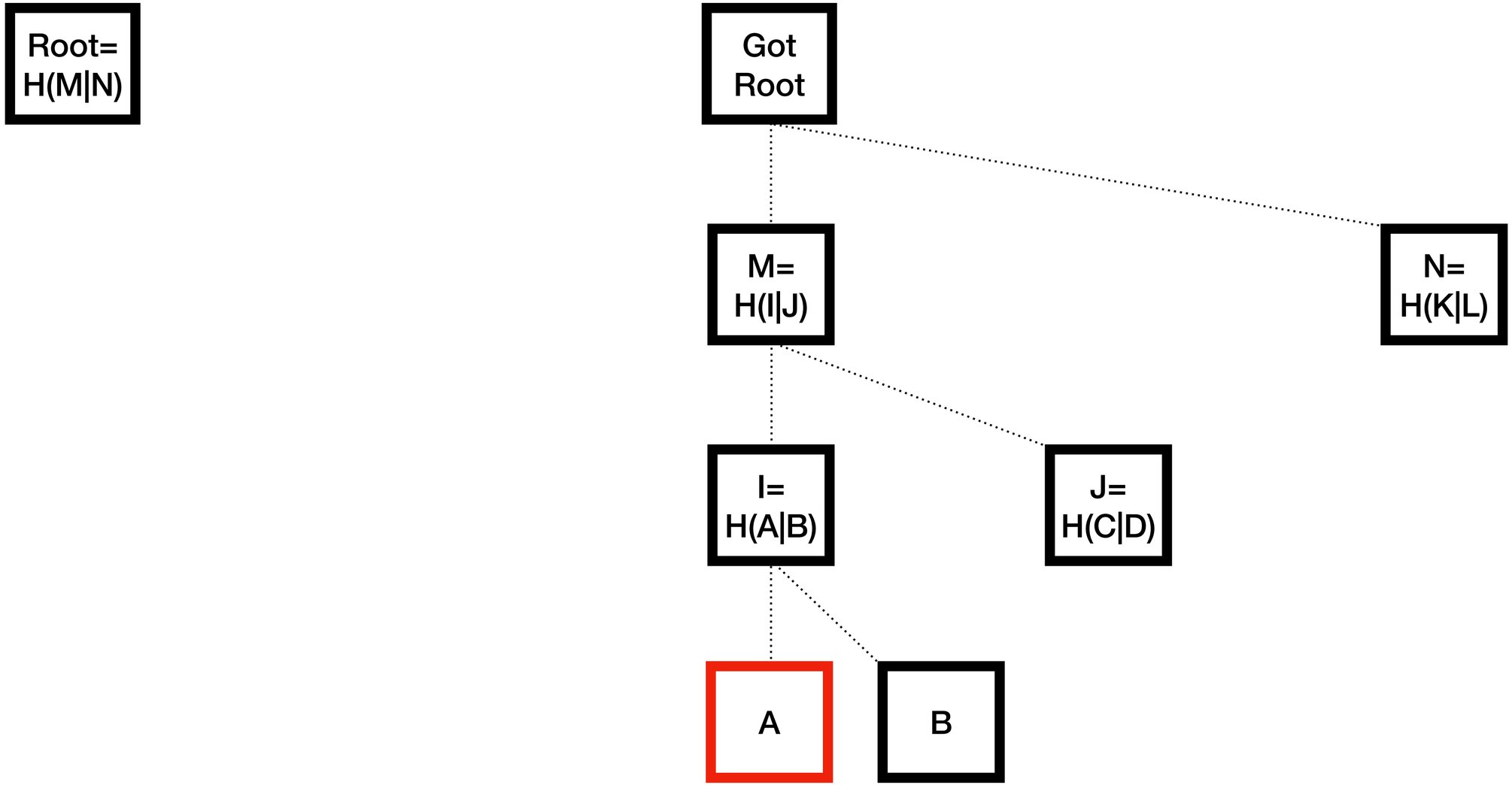
https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

Calculate the hash for M



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

Calculate Root



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

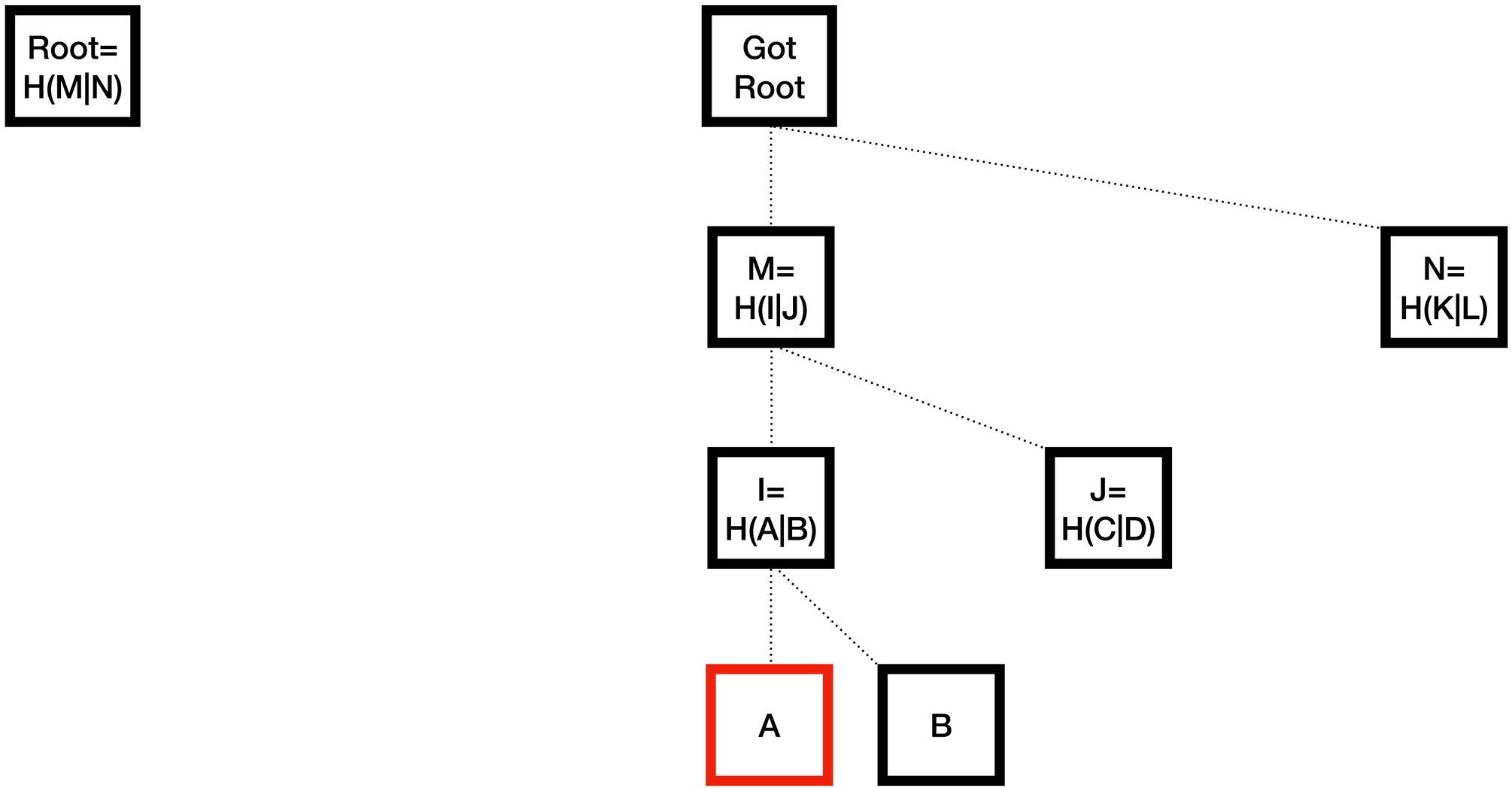
Compare roots

Continue if the roots are equal. Ban peer if not equal

Root=H(M|N) == Got Root

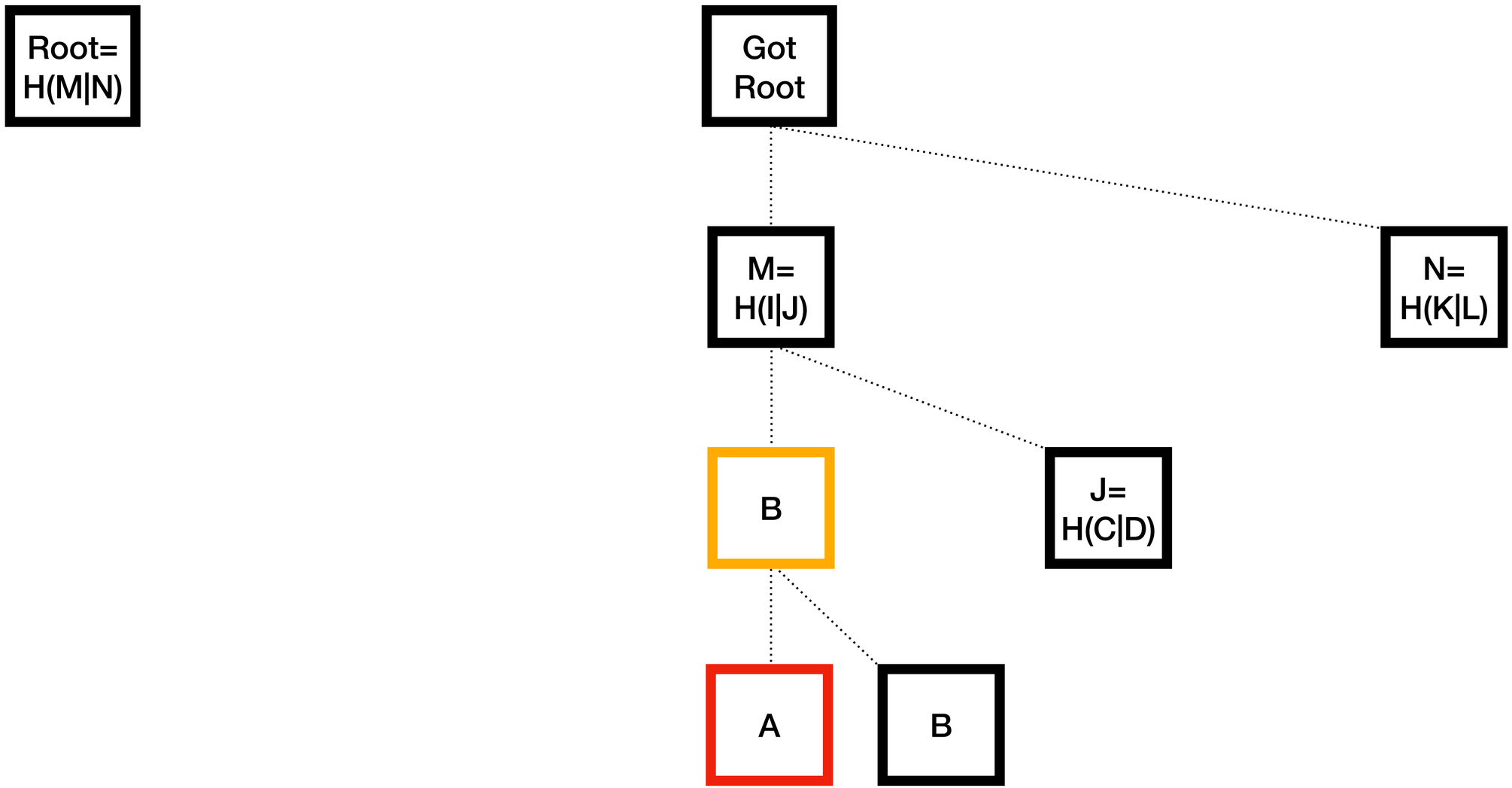
https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

Calculate root after deletion



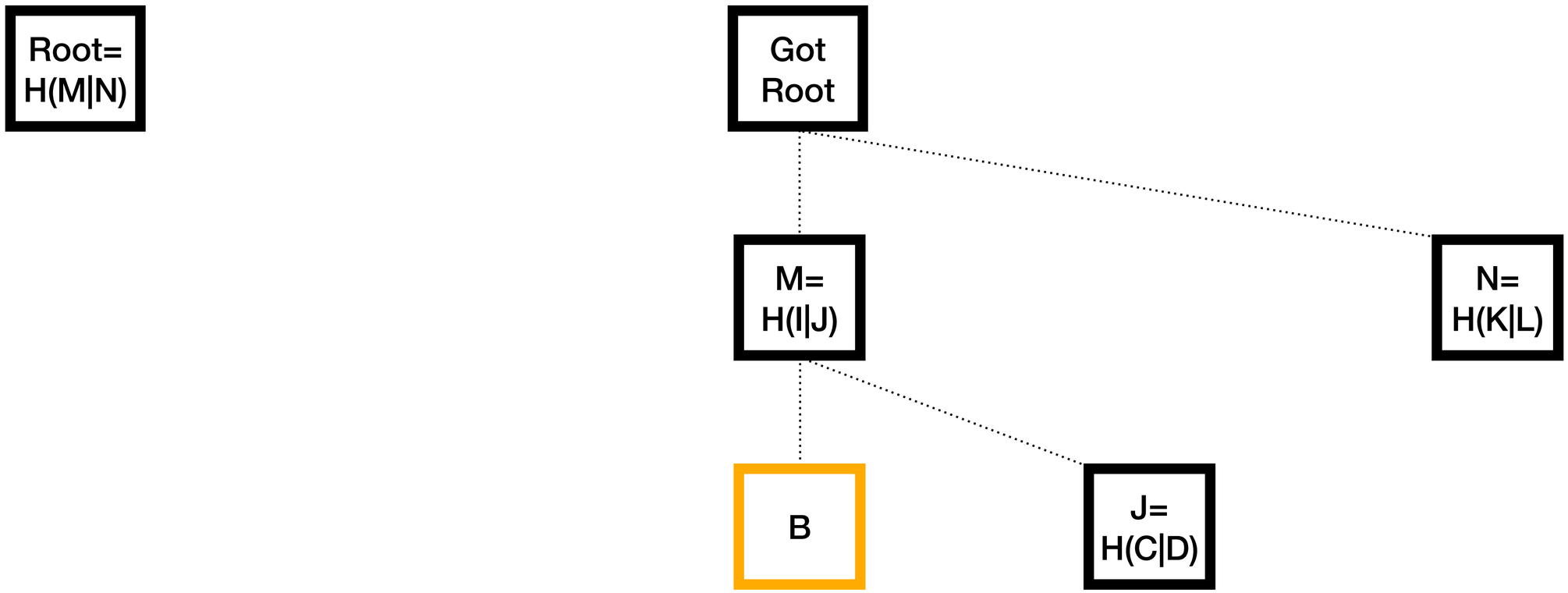
https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

Move up B to I



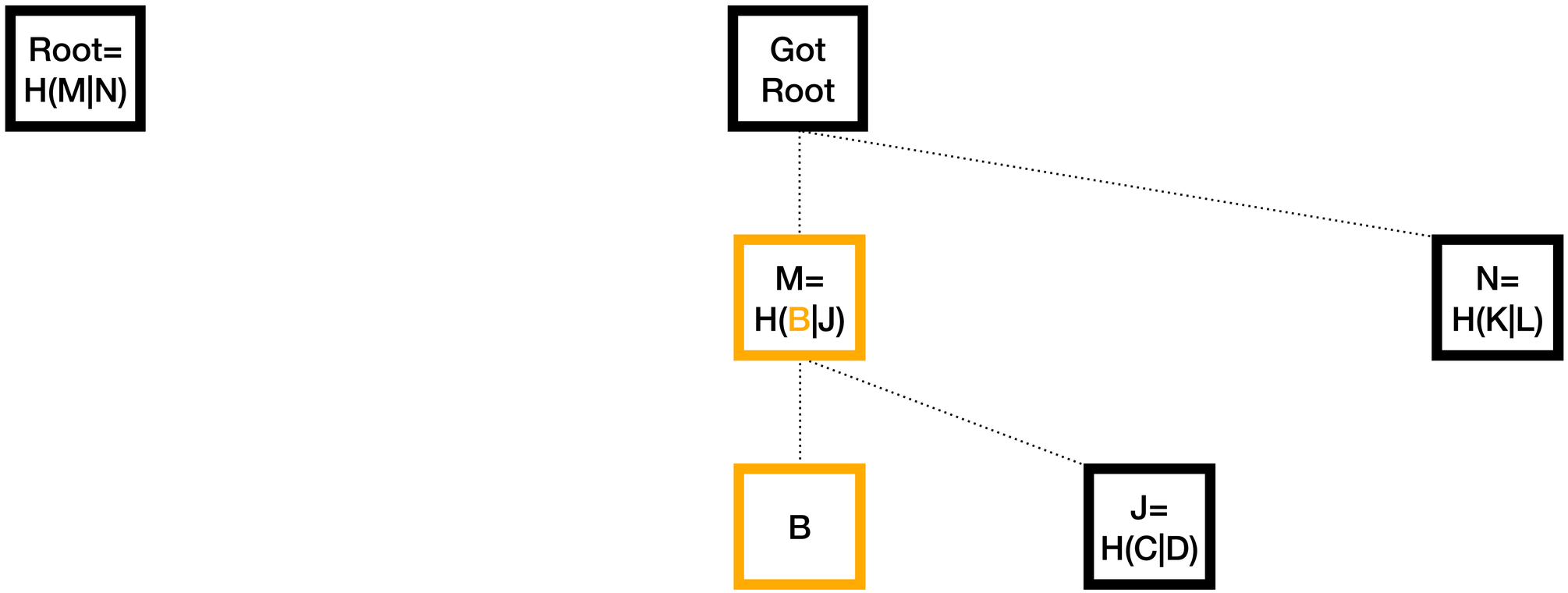
https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

Remove old nodes for A&B



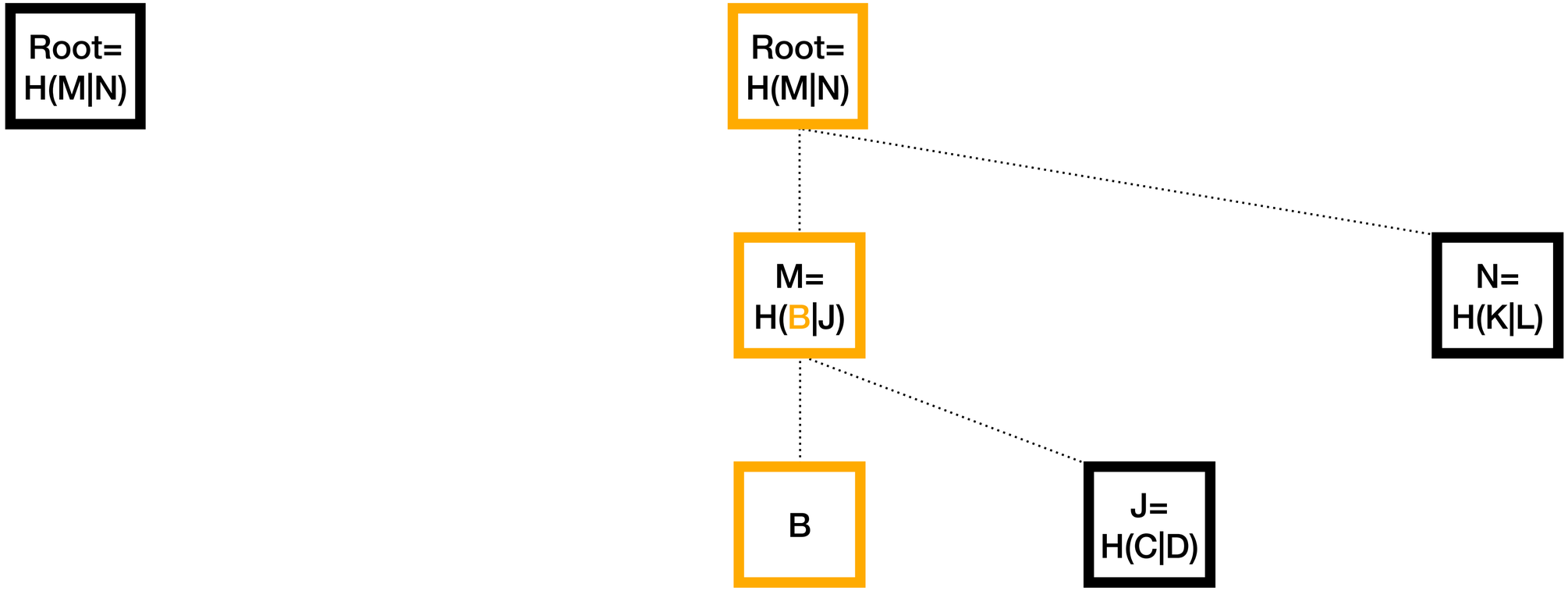
https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

Calculate H(B|J)



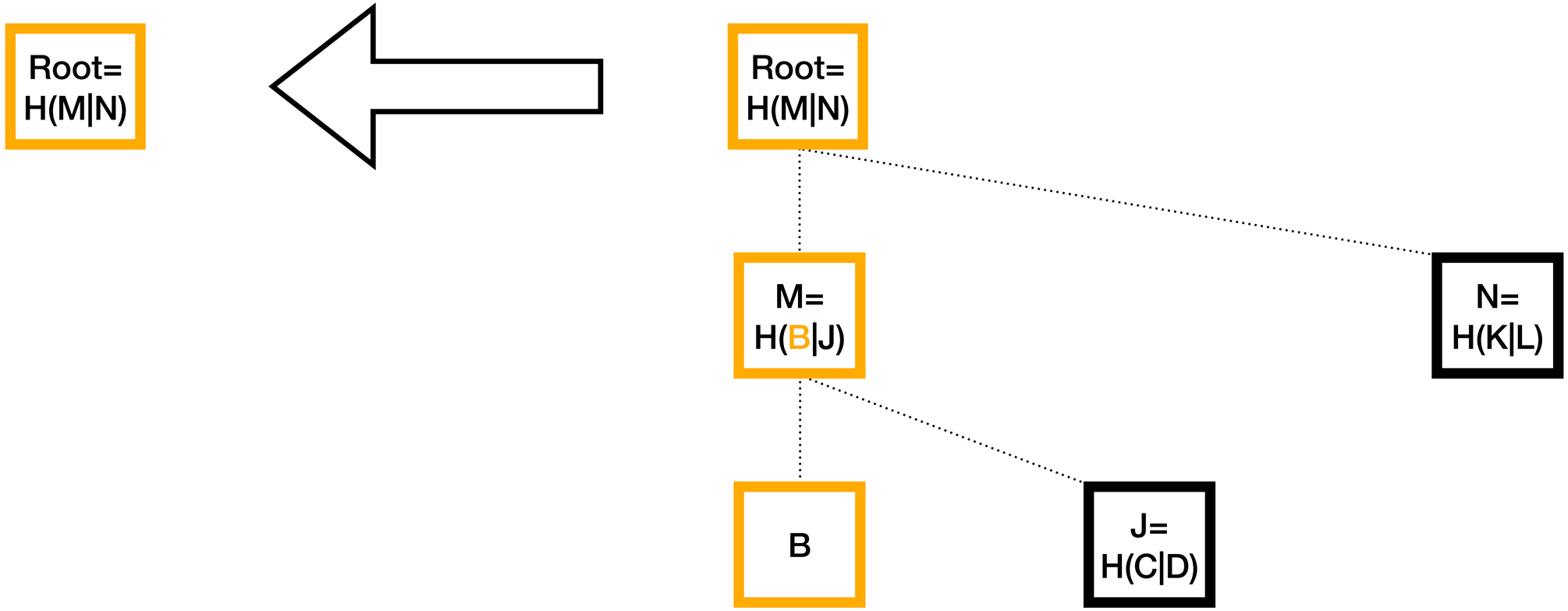
https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

Calculate new root



https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

Copy over the new root to be saved



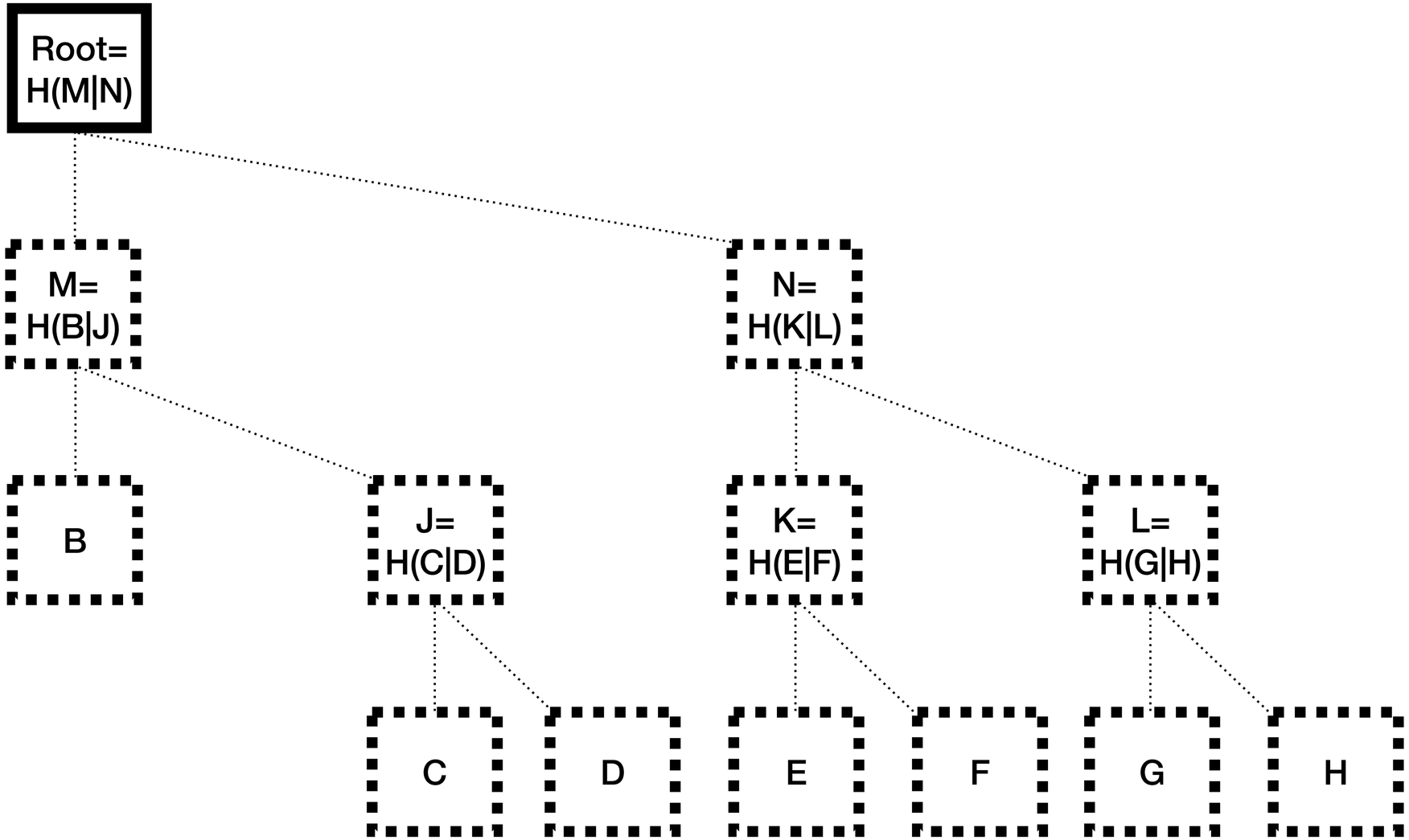
https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

Done

Root=
H(M|N)

https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

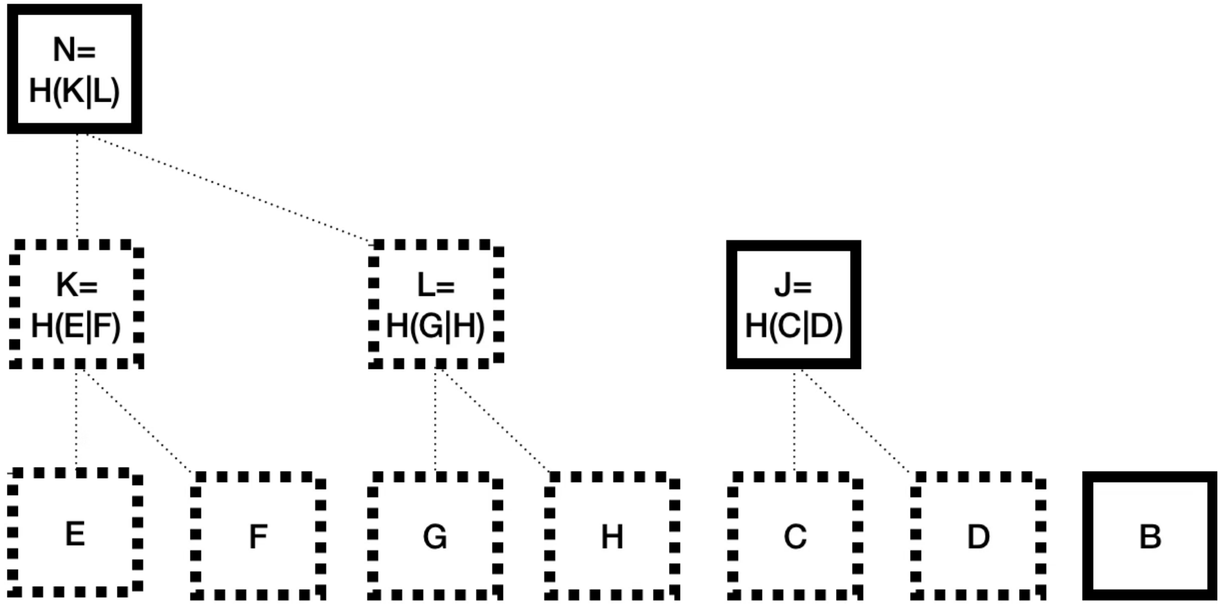
After deleting A



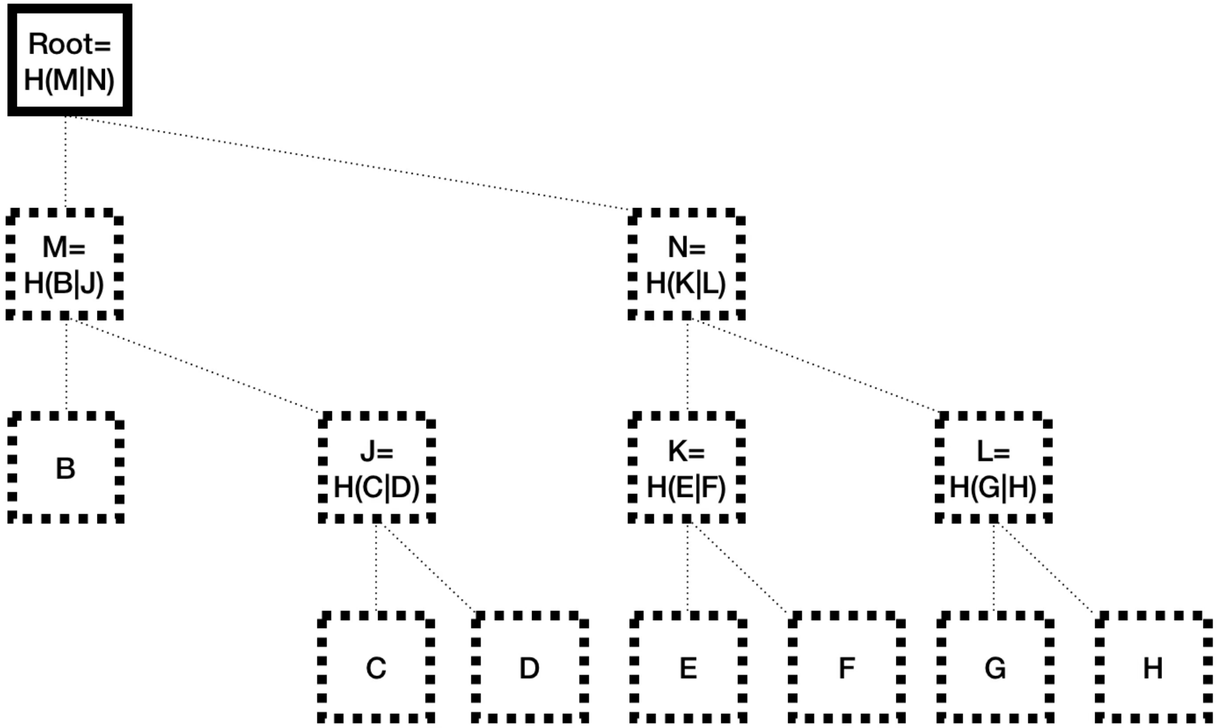
https://gist.github.com/kcalvinalvin/a790d524832e1b7f96a70c642315fffc

Comparison

Old



New



~20% proof size savings

When measured up to height 700,000

Implementing the accumulator

- Bridge node implementation
- CSN implementation
- Consensus critical code

Implementing the accumulator

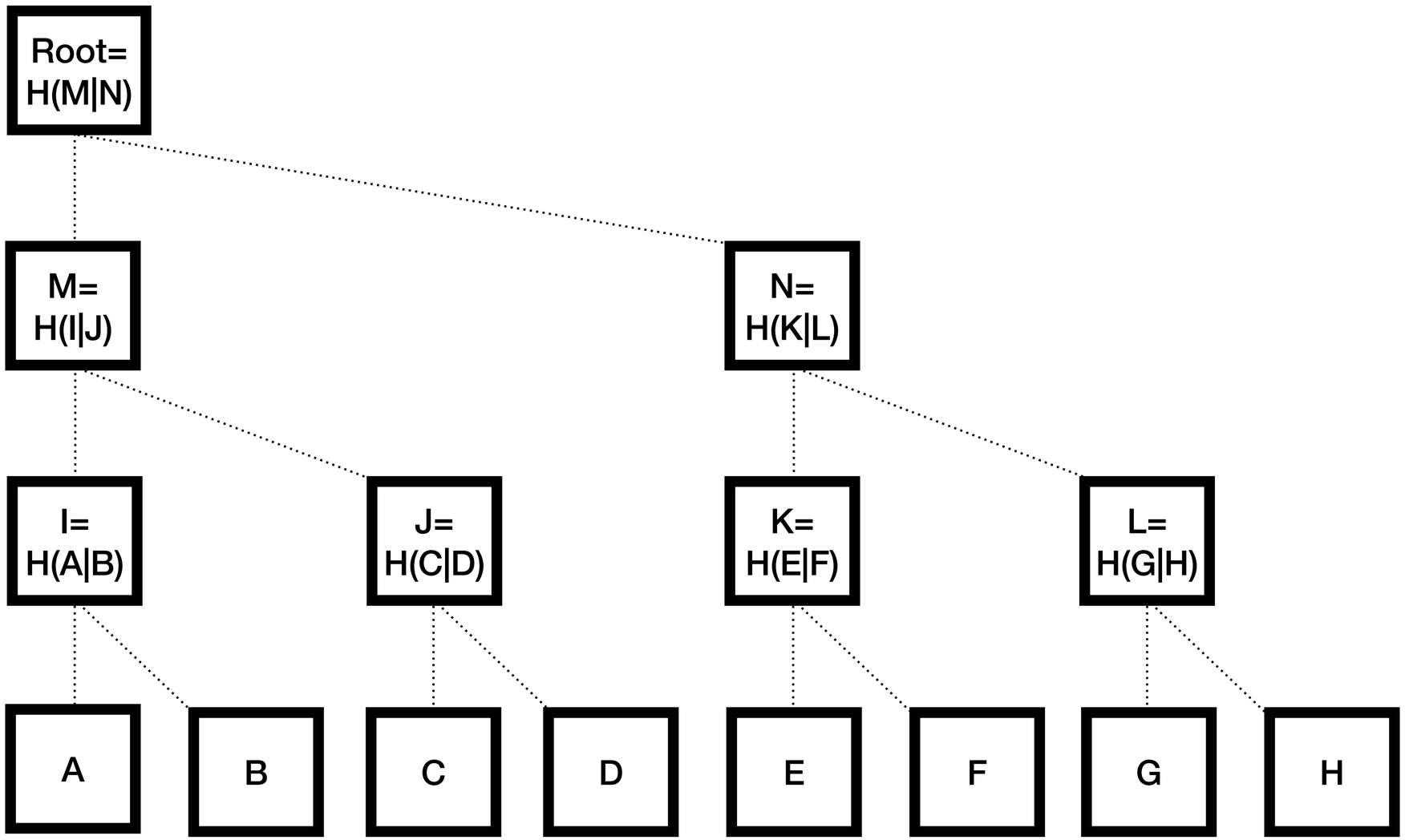
- Bridge node implementation
- CSN implementation
- Consensus critical code

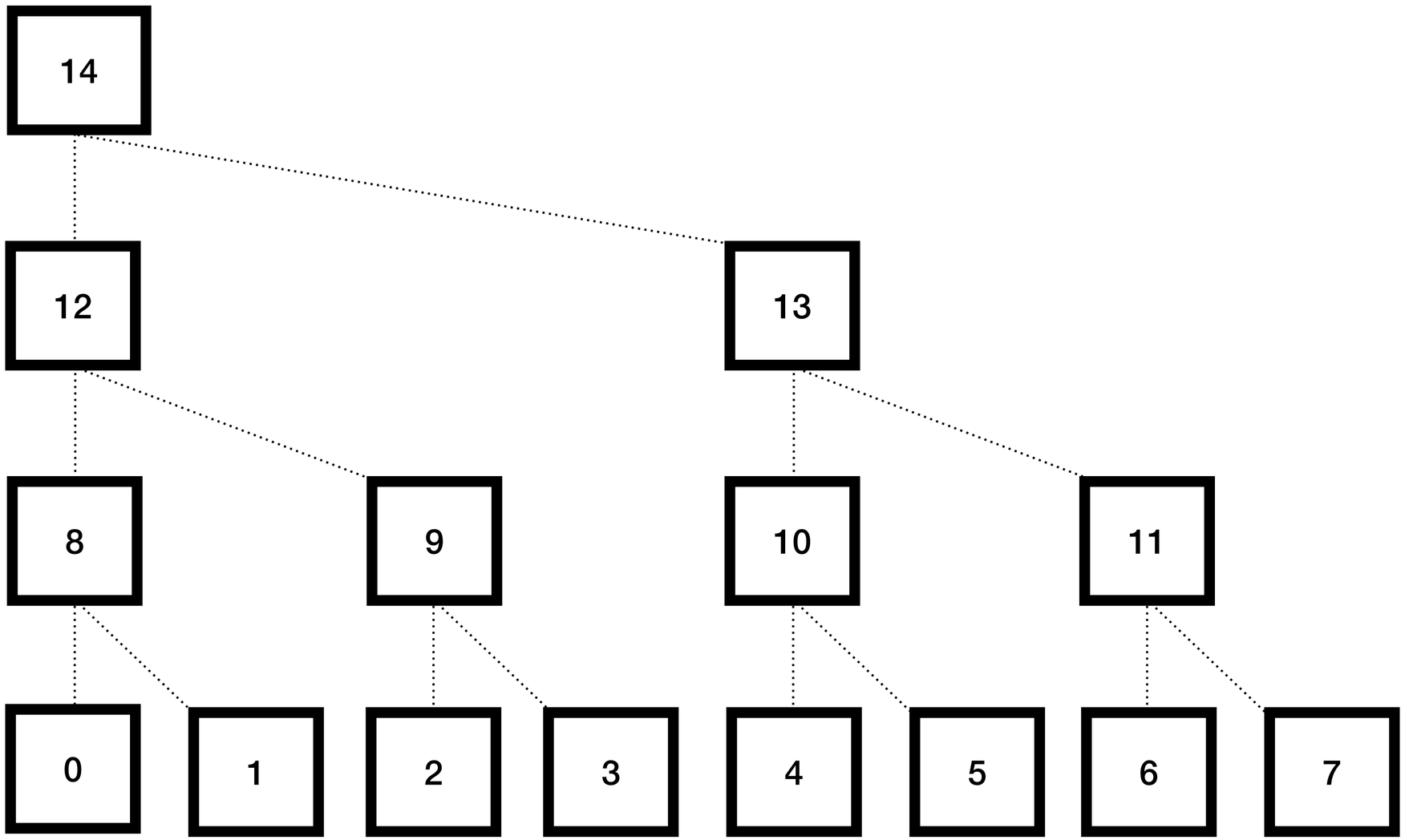
Bridge node implementation

- Must keep the entire merkle forest

Bridge node implementation

- Must keep the entire merkle forest
- Generating proofs should be quick


Root=
H(M|N)
M=
H(I|J)
N=
H(K|L)
I=
H(A|B)
J=
H(C|D)
K=
H(E|F)
L=
H(G|H)
A
B
C
D
E
F
G
H


14
12
13
8
9
10
11
0
1
2
3
4
5
6
7

On disk representation

A	B	C	D	E	F	G	H	I= H(A|B)	J= H(C|D)	K= H(E|F)	L= H(G|H)	M= H(I|J)	N= H(K|L)	Root= H(M|N)

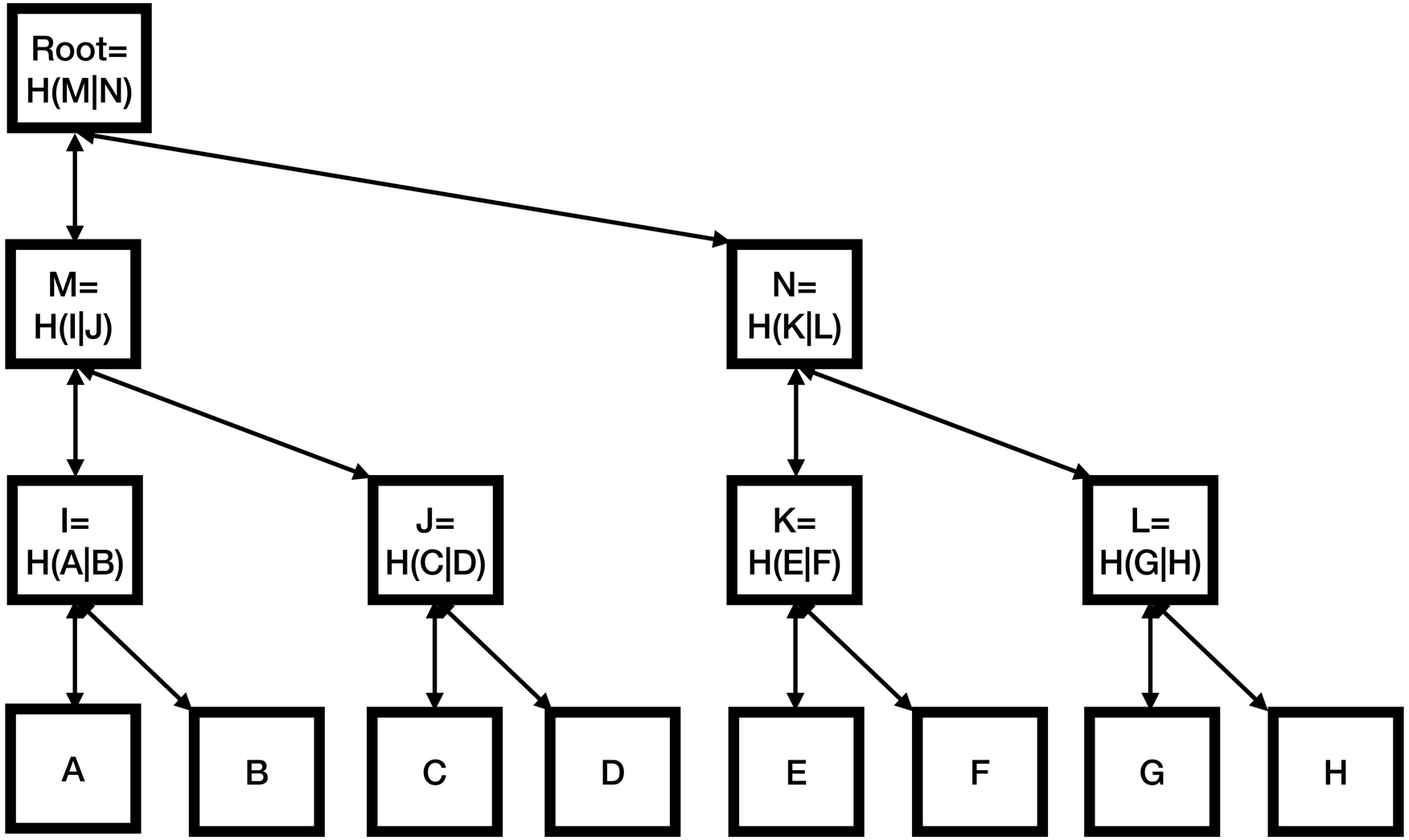
Original on-disk representation

- One big file on disk
- Position * 32 = file offset
- Key-value store

Original on-disk representation

- Simple
- Slow

Pointer representation



Pointer based implementation

- All in memory and pointer based
- Extremely performant

Pointer based implementation

- Required 64GB of memory

High memory usage #63

Closed **Davidson-Souza** opened this issue on Apr 15, 2023 · 3 comments · Fixed by #147

Davidson-Souza commented on Apr 15, 2023 Member

While synchronizing with testnet or mainnet I got a peak memory usage of 8GB, using the default config. I'm not familiar with Go profiling utilities, but I can try getting more data if needed. Here's a print of an utreexod running on testnet, taking 8GB + some swap.
My computer is an AMD Ryzen 5 2500U, 12GB of RAM. go version go1.20.3 linux/amd64

```
MEM%▽   TIME+  Command
58.1  8h18:23 ./utreexod --testnet --utreexoproofindex
58.1   4:14.09 ./utreexod --testnet --utreexoproofindex
58.1  44:05.68 ./utreexod --testnet --utreexoproofindex
58.1  39:24.78 ./utreexod --testnet --utreexoproofindex
58.1   0:00.00 ./utreexod --testnet --utreexoproofindex
58.1  44:59.94 ./utreexod --testnet --utreexoproofindex
58.1  40:24.05 ./utreexod --testnet --utreexoproofindex
58.1  41:10.91 ./utreexod --testnet --utreexoproofindex
58.1   0:00.00 ./utreexod --testnet --utreexoproofindex
58.1  45:02.80 ./utreexod --testnet --utreexoproofindex
58.1  37:02.51 ./utreexod --testnet --utreexoproofindex
58.1  40:30.08 ./utreexod --testnet --utreexoproofindex
58.1  43:17.24 ./utreexod --testnet --utreexoproofindex
58.1  42:31.75 ./utreexod --testnet --utreexoproofindex
58.1  38:28.72 ./utreexod --testnet --utreexoproofindex
58.1  37:10.64 ./utreexod --testnet --utreexoproofindex
```

👍 1

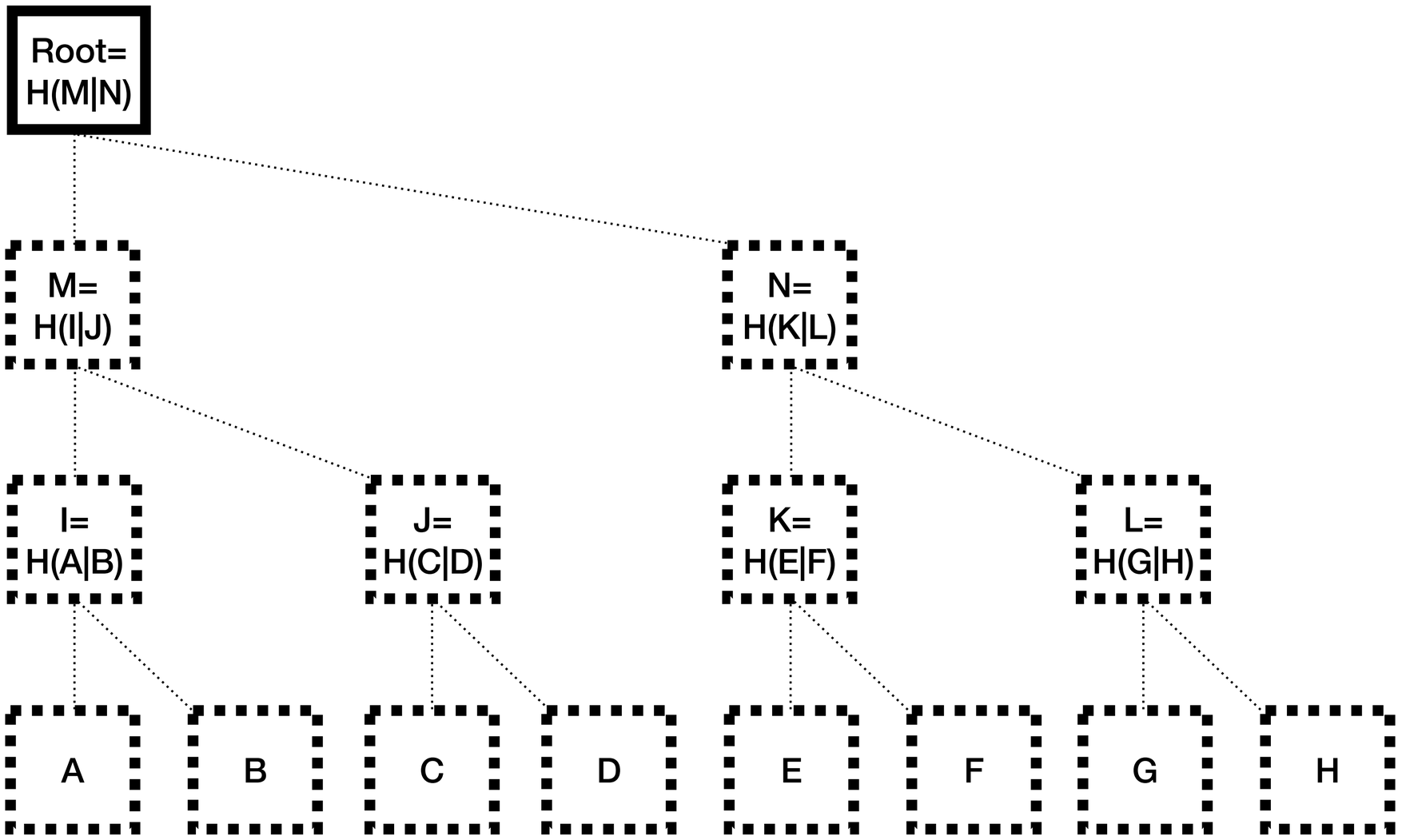
jaonoctus commented on Apr 16, 2023 Contributor

same here

```
[  108.983398] Out of memory: Killed process 1212 (utreexod) total-vm:6759384kB,
```

Compact State Node implementation

- Only keeps the roots
- Must be able to cache proofs for mempool txs

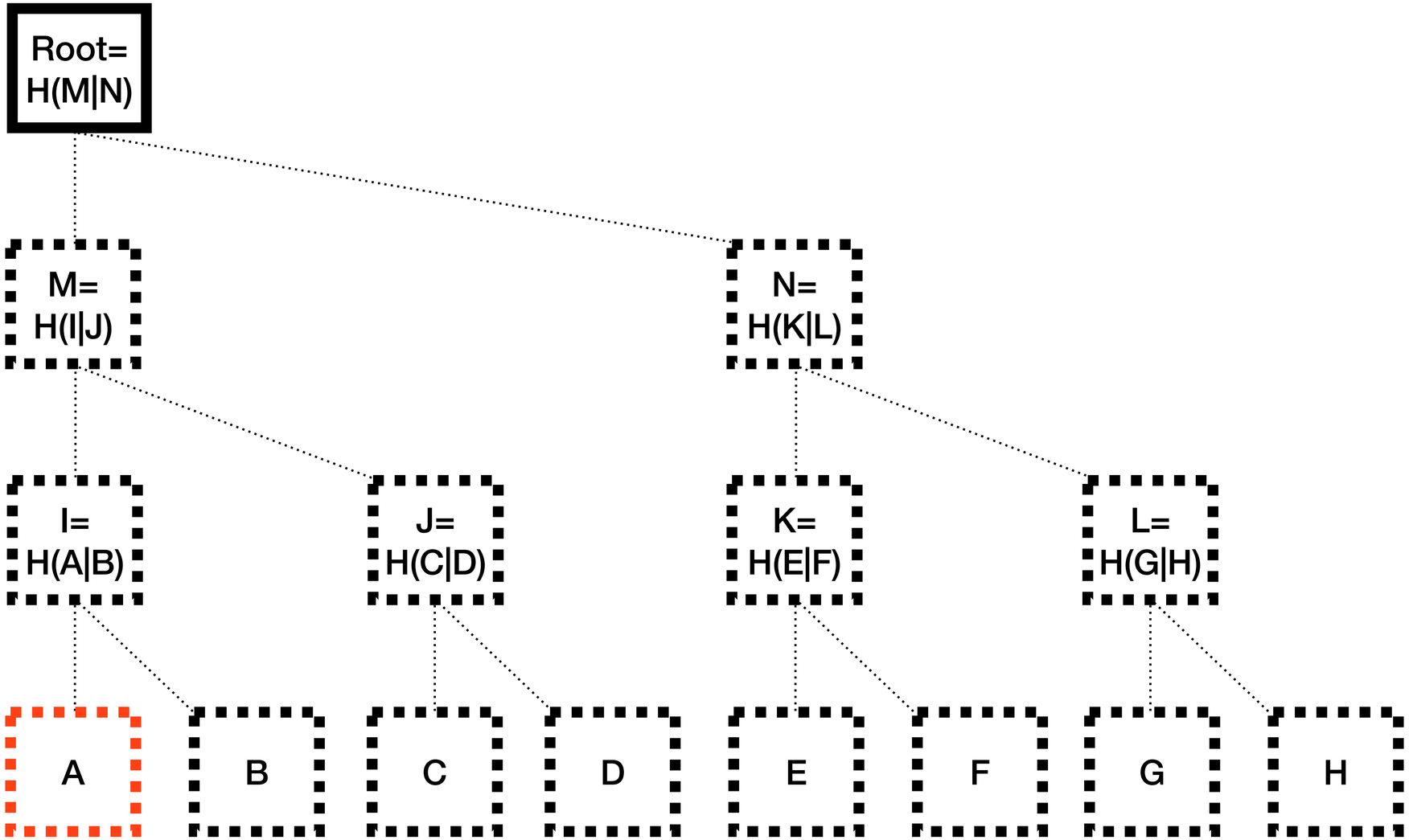

Root=
H(M|N)
M=
H(I|J)
N=
H(K|L)
I=
H(A|B)
J=
H(C|D)
K=
H(E|F)
L=
H(G|H)
A
B
C
D
E
F
G
H

Root=
H(M|N)

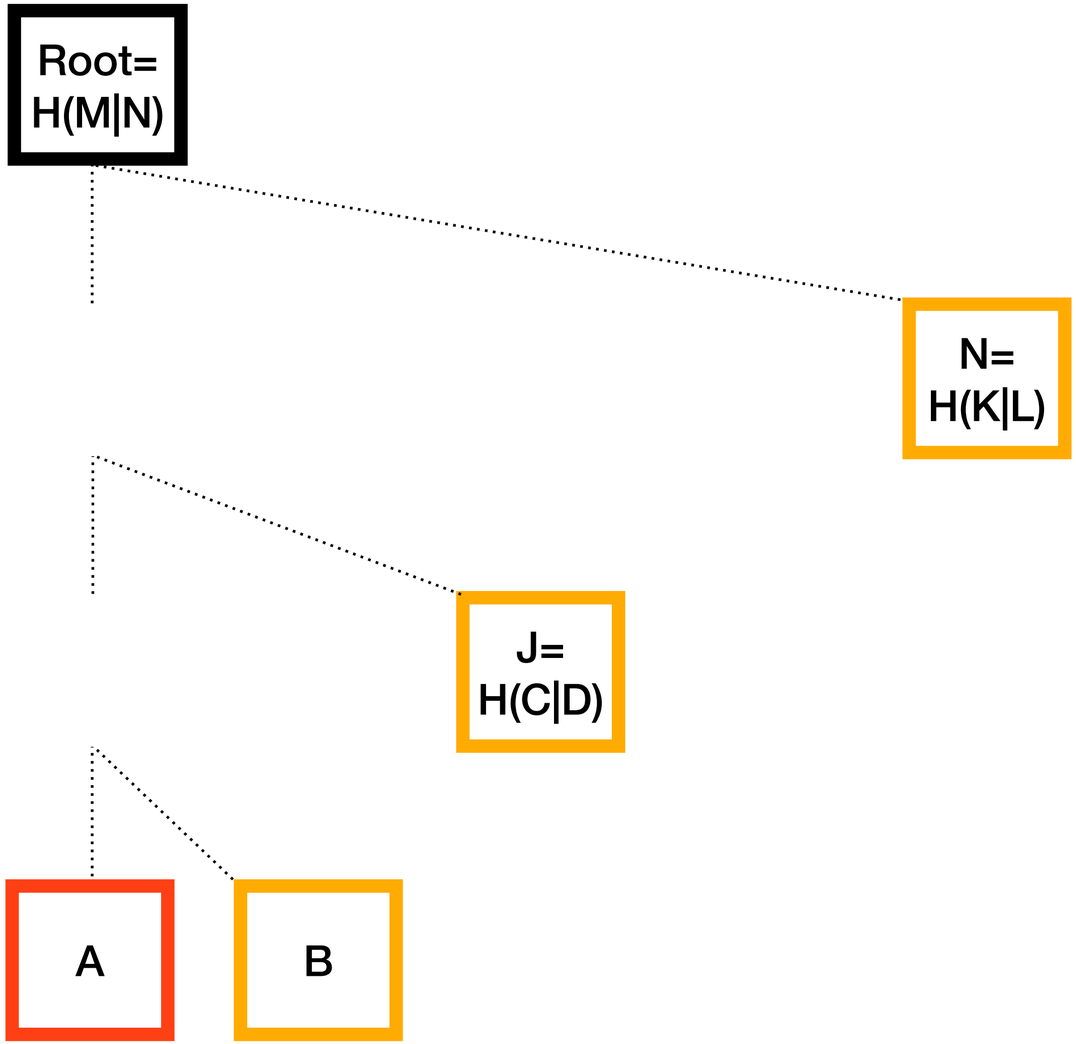
Utreexo nodes are able to propagate transactions

Assumption we're working with

Transaction is spending A



What a CSN keeps



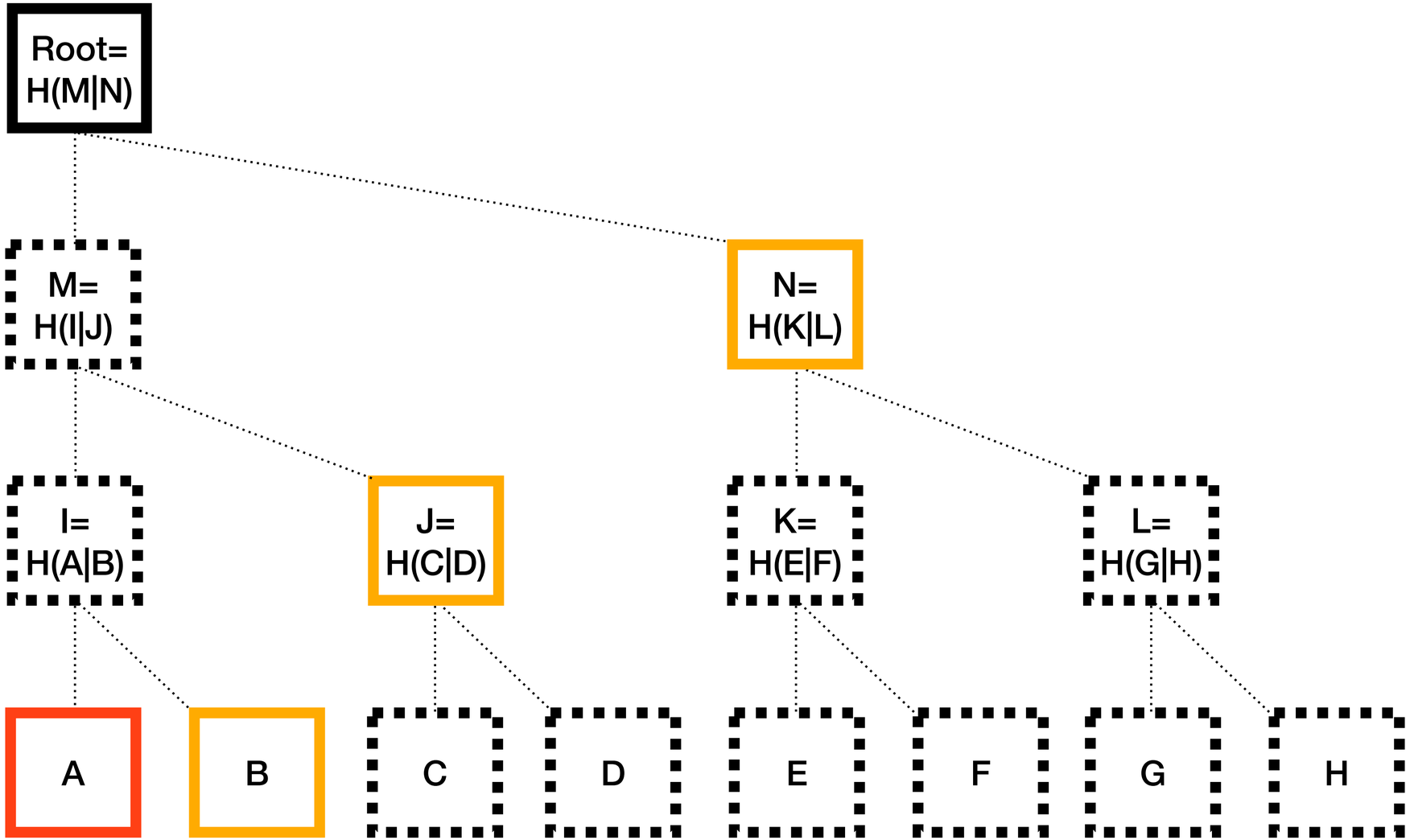
Pointer based implementation

- Fast operations

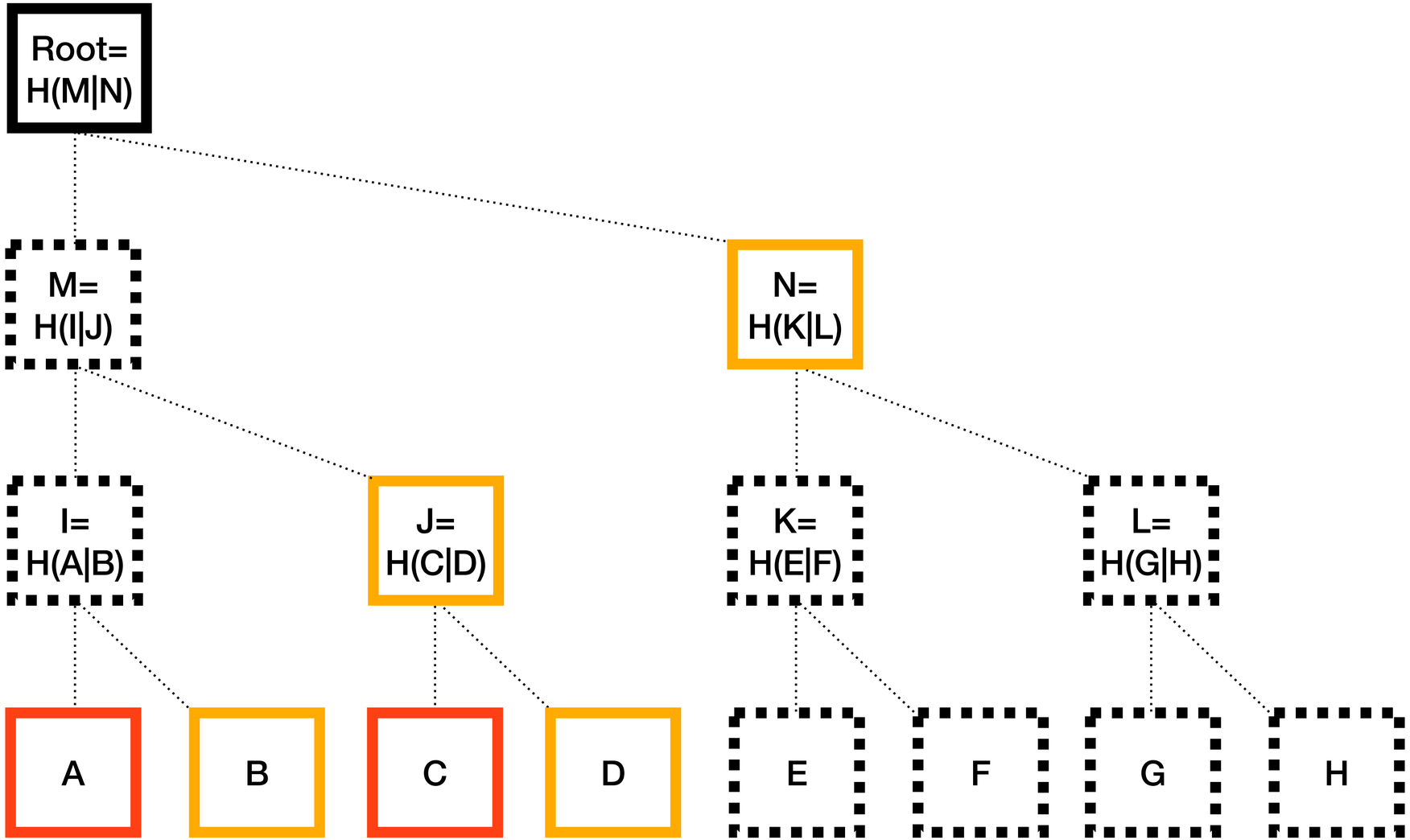
Pointer based implementation

- Fast operations
- Fairly complicated code to support caching

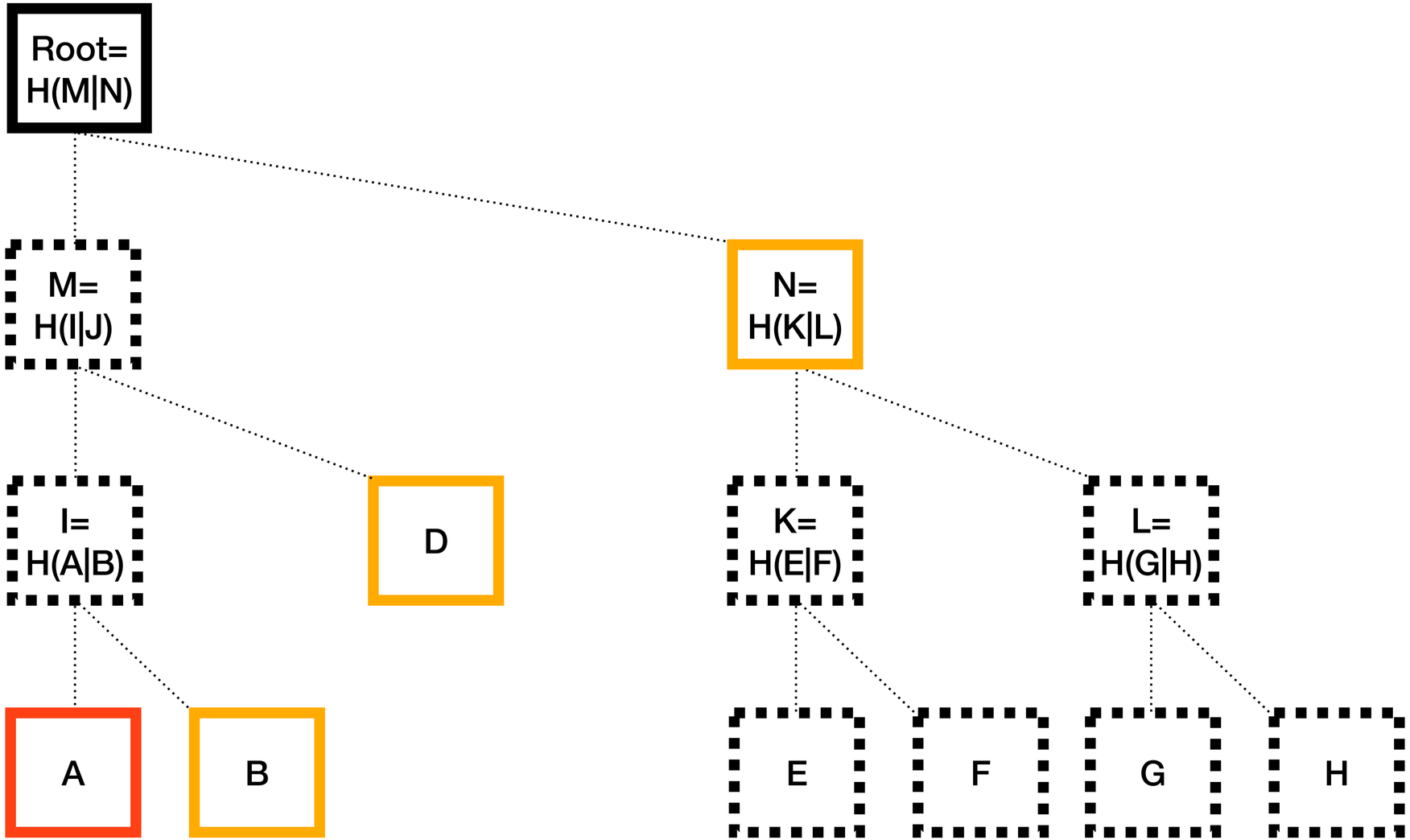
If C gets spent before A



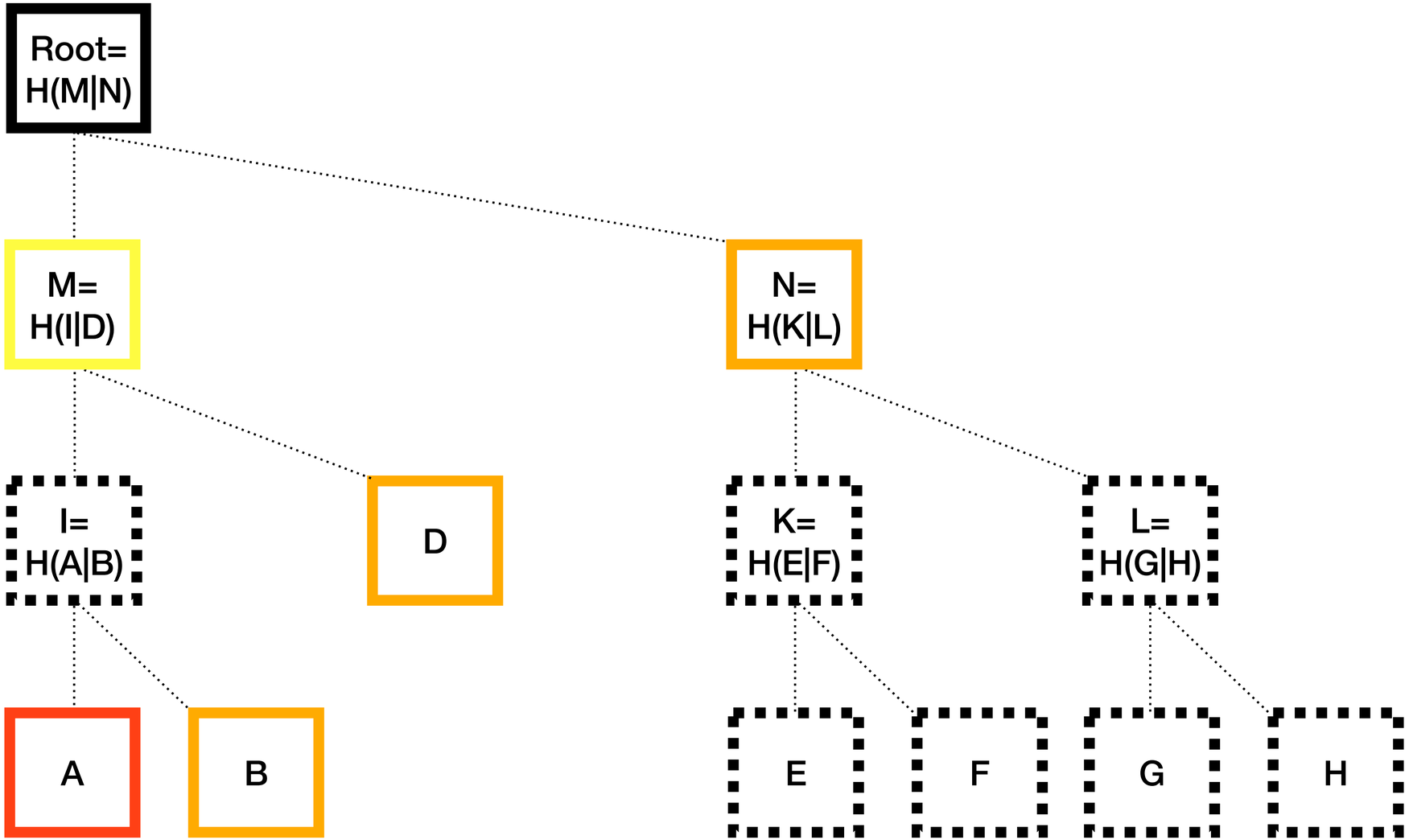
If C gets spent before A



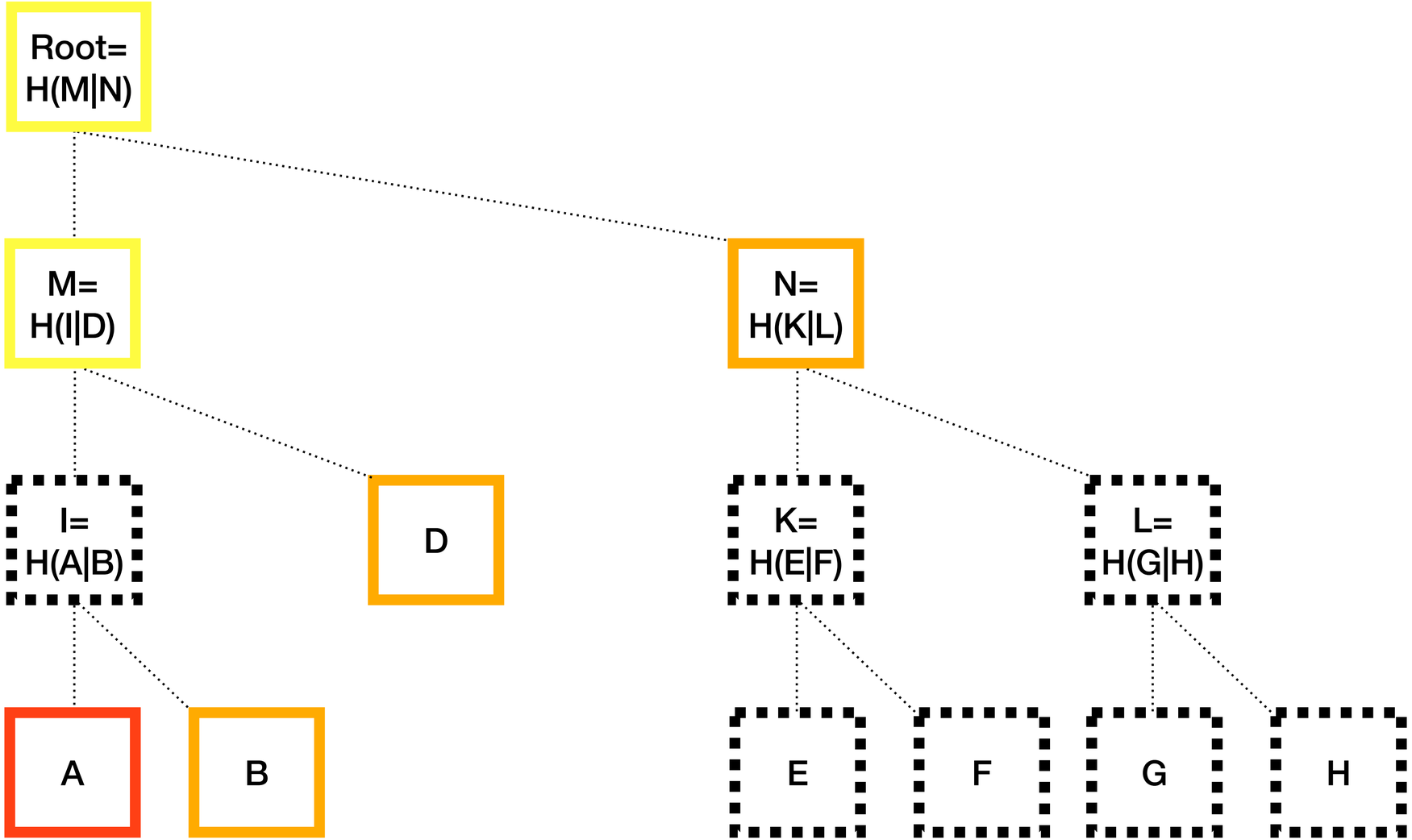
If C gets spent before A



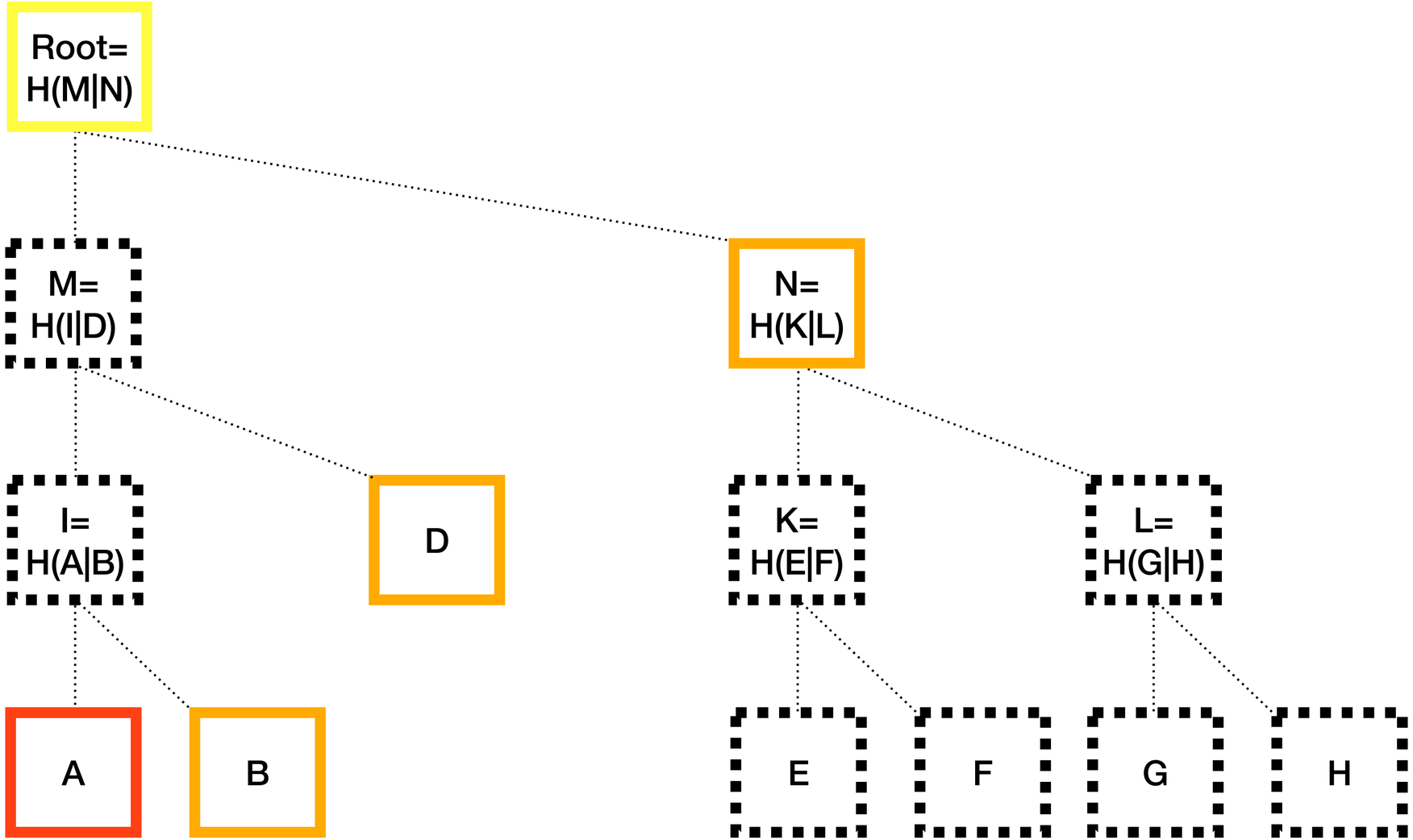
If C gets spent before A



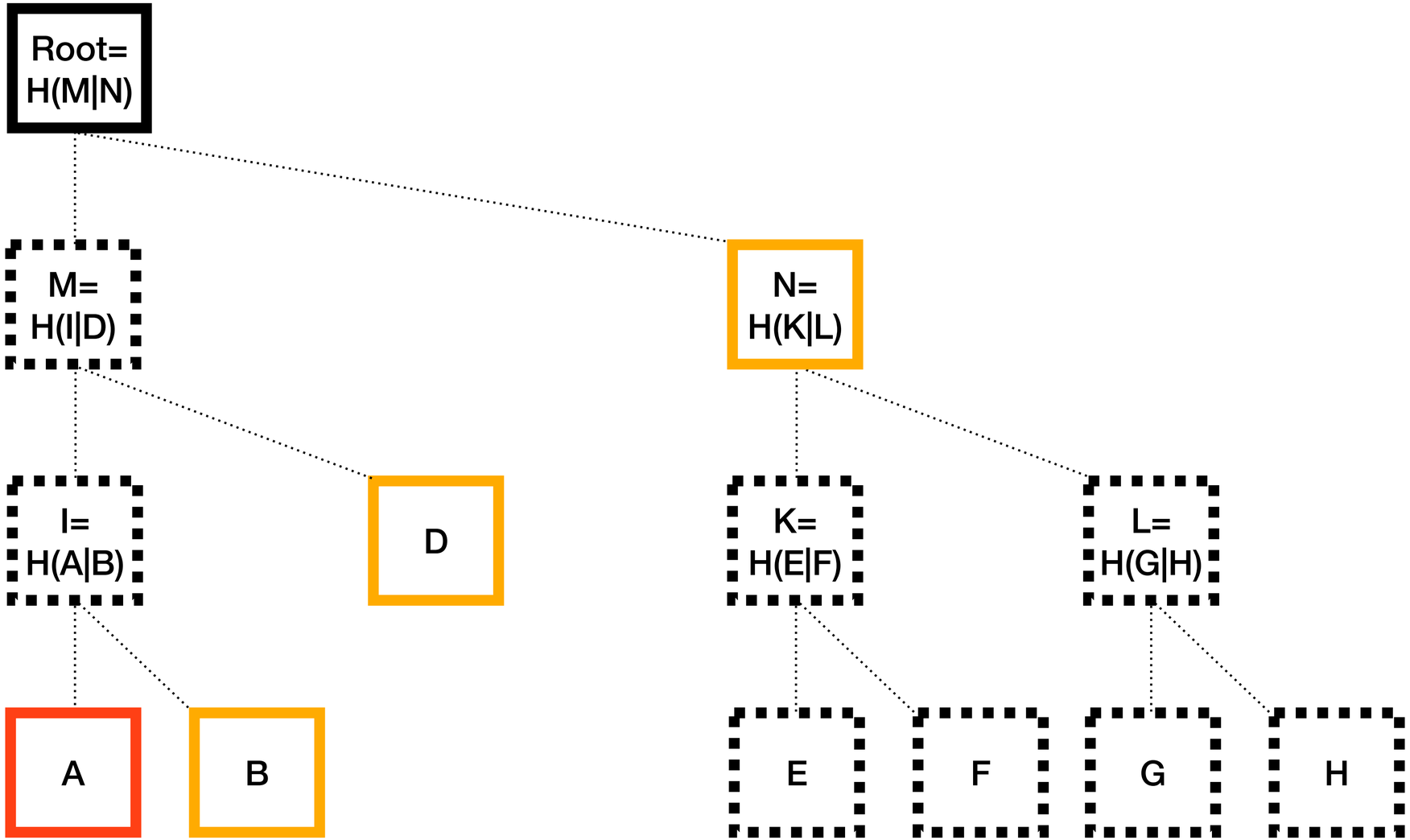
If C gets spent before A



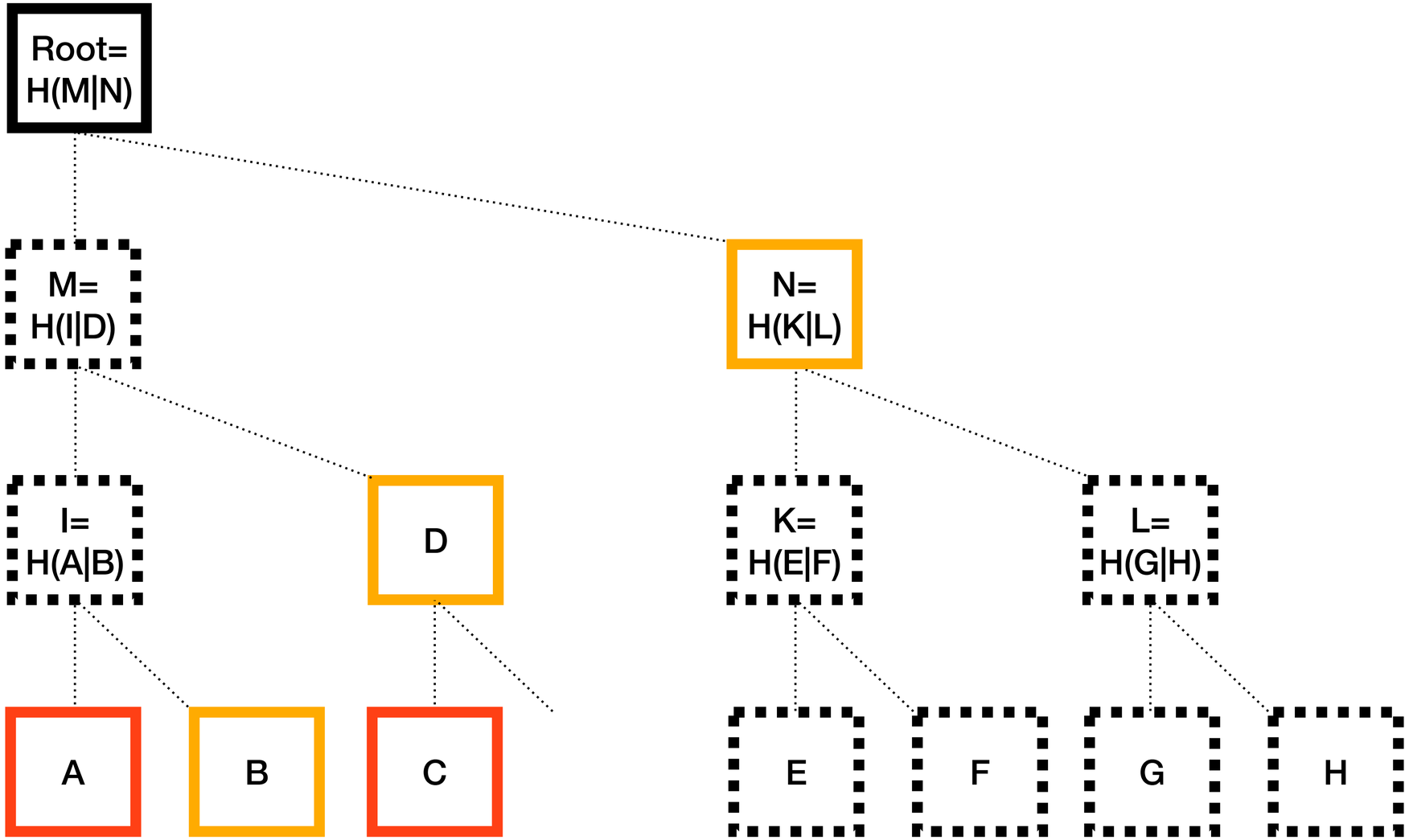
If C gets spent before A



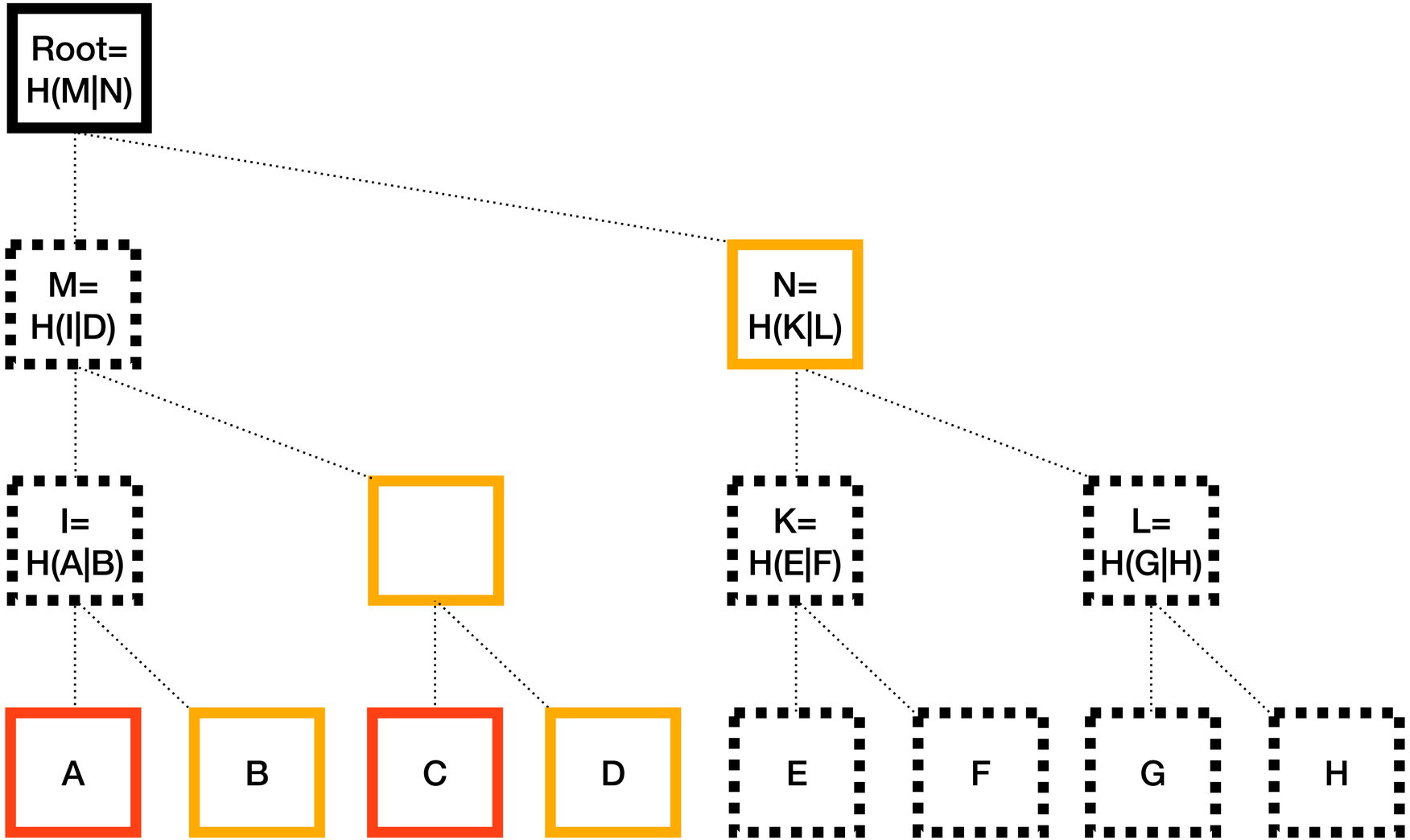
If block with transaction C gets reorged



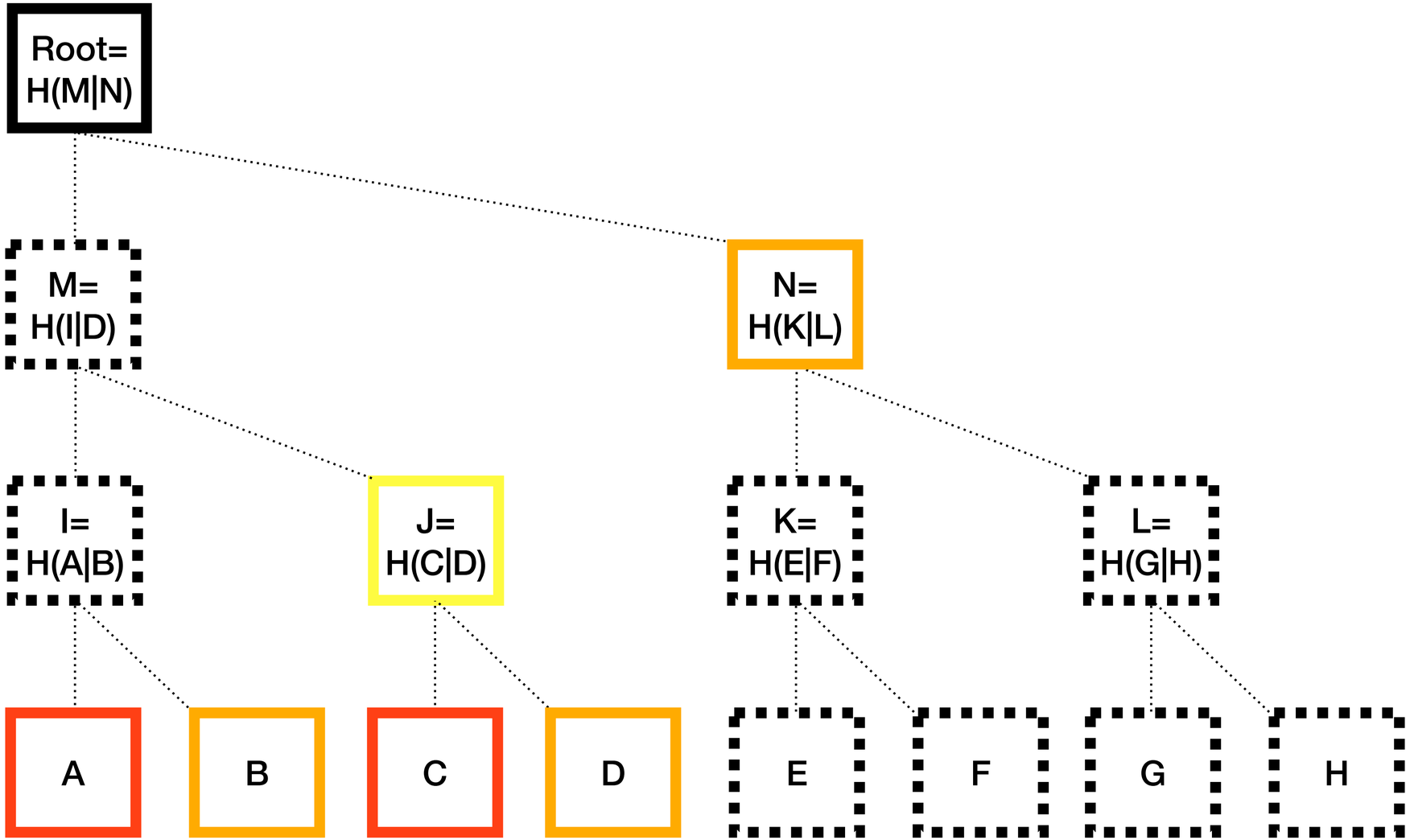
If block with transaction C gets reorged



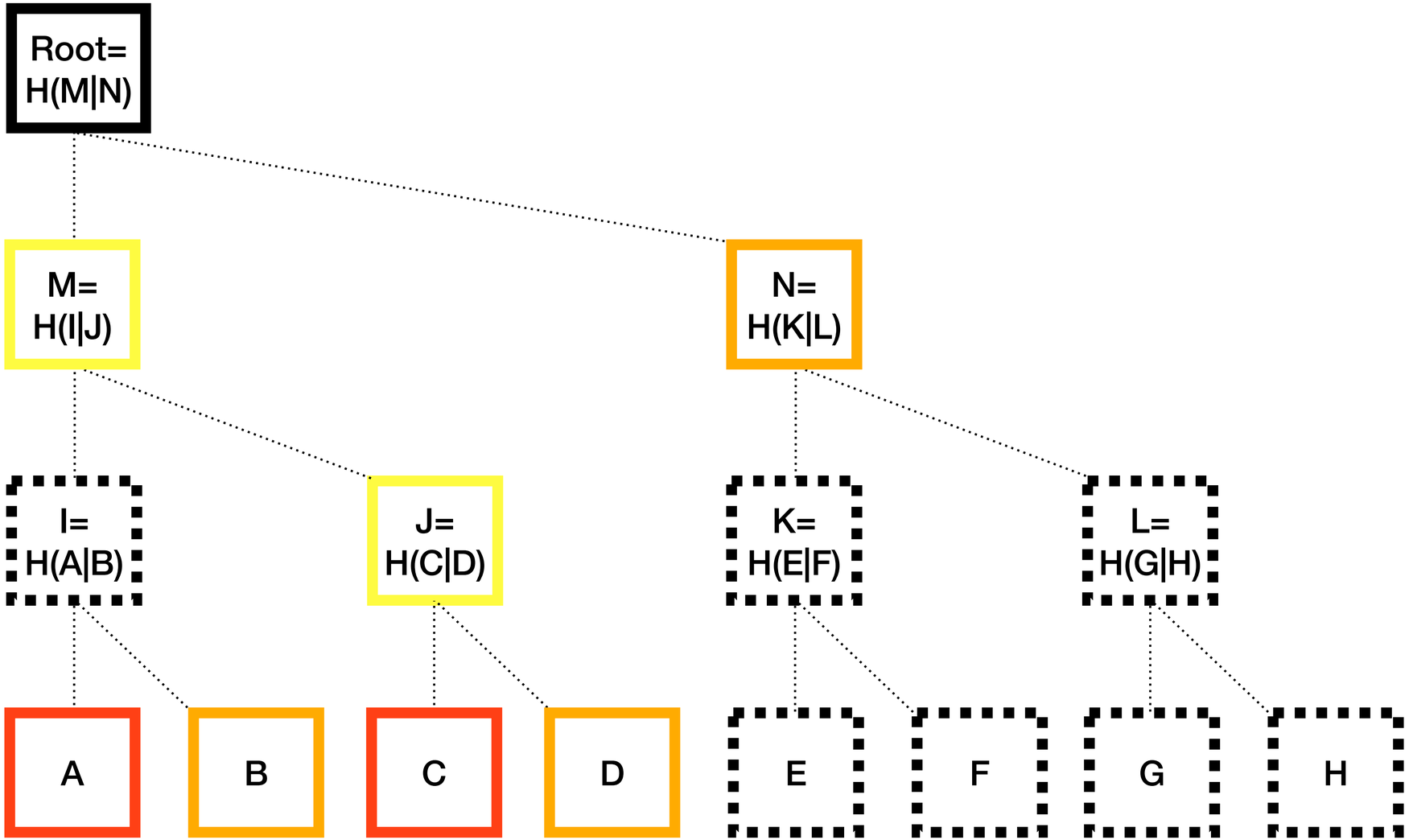
If block with transaction C gets reorged



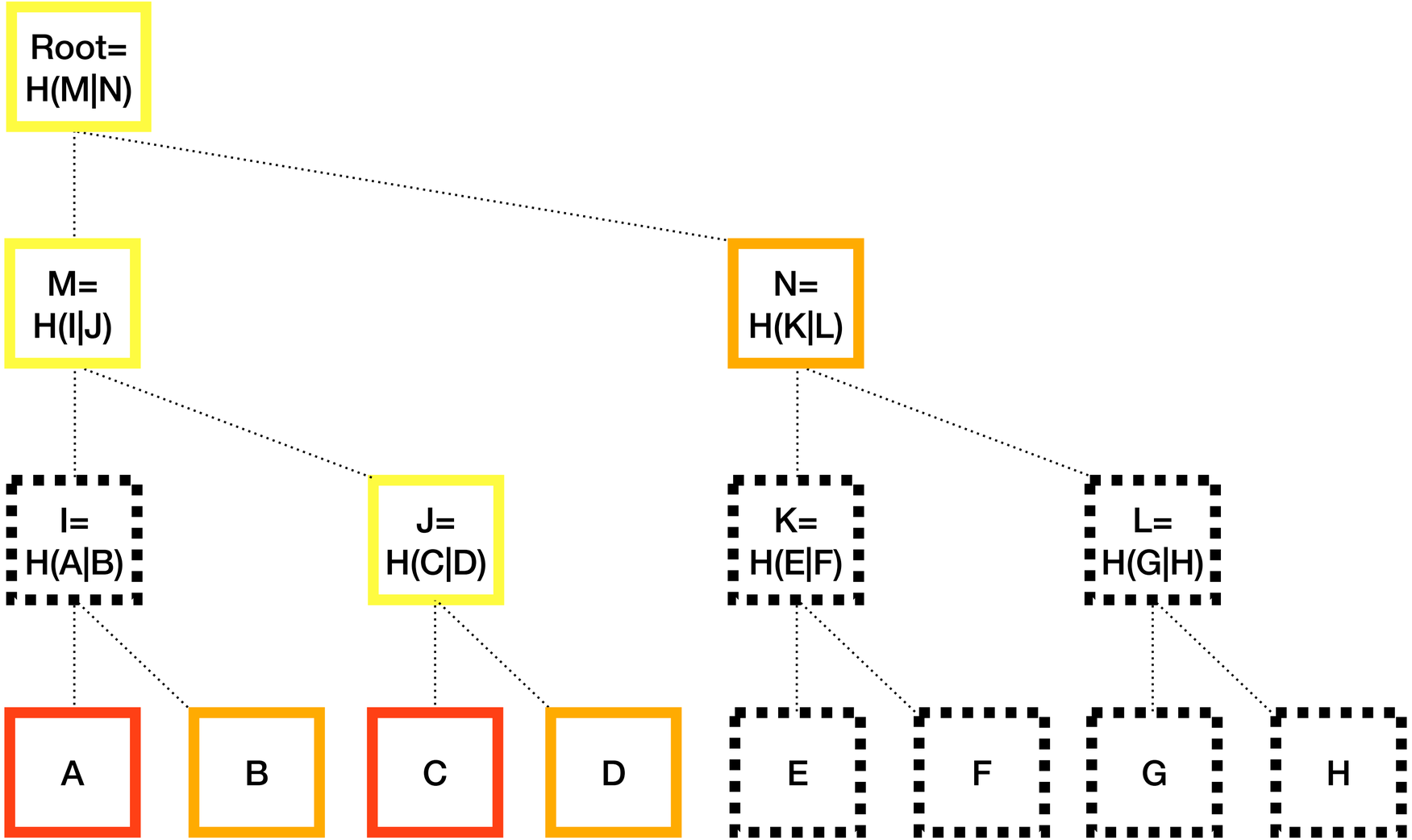
If block with transaction C gets reorged



If block with transaction C gets reorged



If block with transaction C gets reorged



Hashmap based implementation

- key-value (position-hash) is stored in a hashmap

Hashmap based implementation

- key-value (position-hash) is stored in a hashmap
- Allowed for simpler code vs pointer based

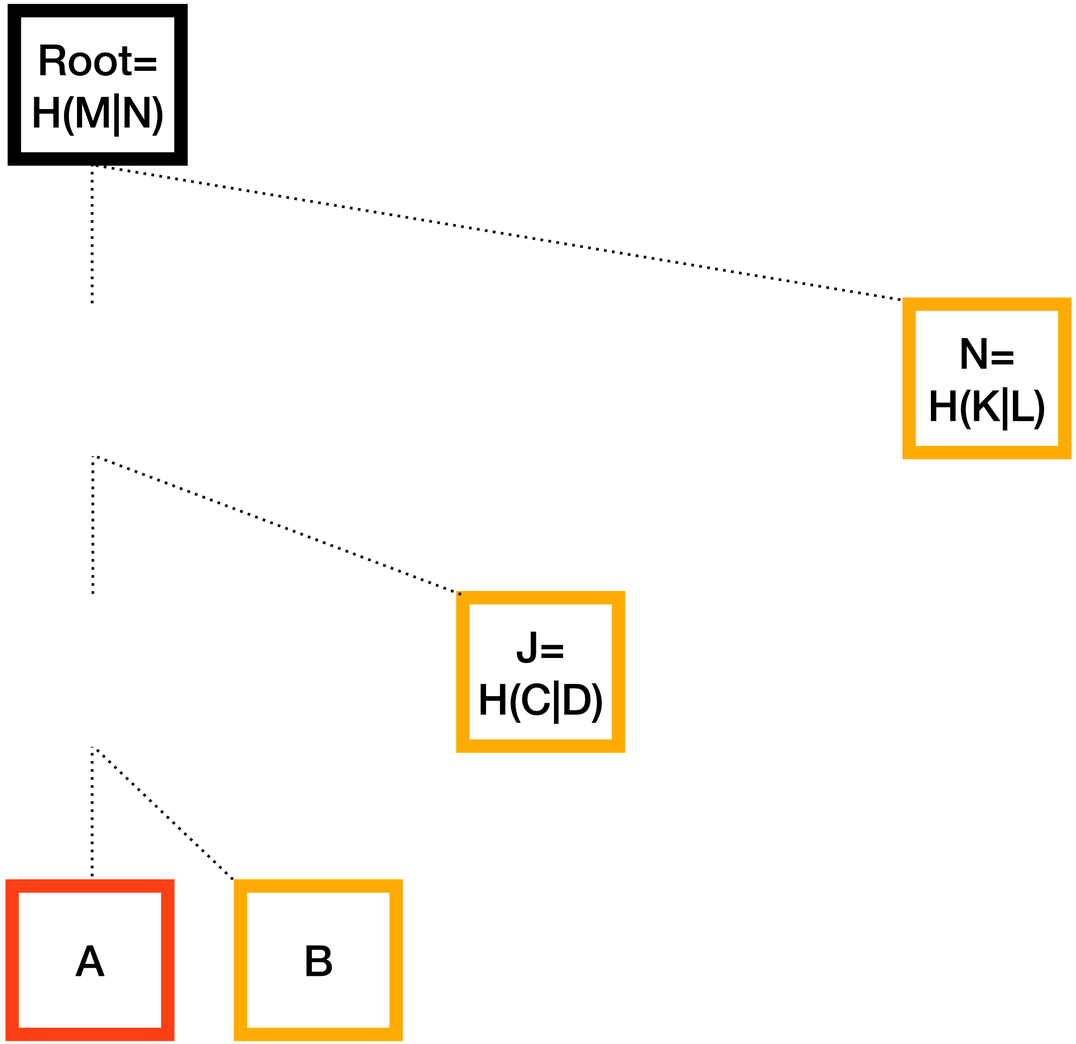
Hashmap based implementation

- key-value (position-hash) is stored in a hashmap
- Allowed for simpler code
- Can also keep everything

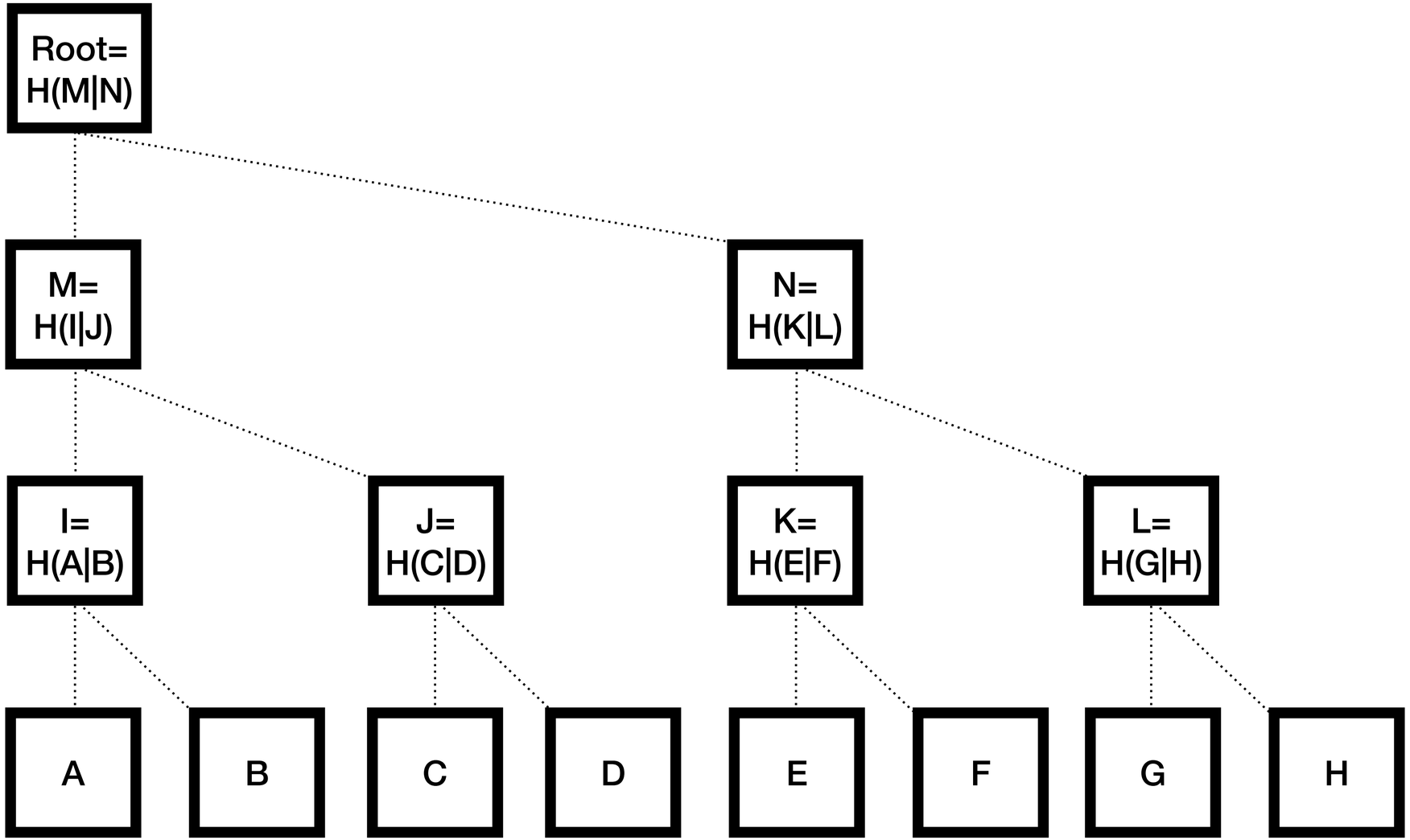
```
  --blockminsize=                    Mininum block size in bytes to be used when creating a block
  --blockmaxweight=                  Maximum block weight to be used when creating a block (default: 3000000)
  --blockminweight=                  Mininum block weight to be used when creating a block
  --blockpriritysize=                Size in bytes for high-priority/low-fee transactions when creating a block (default: 50000)
  --addrindex                        Maintain a full address-based transaction index which makes the searchrawtransactions RPC available
  --txindex                          Maintain a full hash-based transaction index which makes all transactions available via the getrawtransaction RPC
  --ttlindex
  --utreexoproofindex
  --flatutreexoproofindex            Maintain a utreexo proof for all blocks in flat files
  --utreexoproofindexmaxmemory=      The maxmimum memory in mebibytes (MiB) that the utreexo proof indexes will use up. Passing in 0 will make the entire proof index stay on disk.
                                     Passing in a negative value will make the entire proof index stay in memory. Default of 250MiB. (default: 250)

  --cfilters                         Enable committed filtering (CF) support
  --nopeerbloomfilters               Disable bloom filtering support
  --dropaddrindex                    Deletes the address-based transaction index from the database on start up and then exits.
  --dropcfindex                      Deletes the index used for committed filtering (CF) support from the database on start up and then exits.
  --droptxindex                      Deletes the hash-based transaction index from the database on start up and then exits.
  --dropttlindex                     Deletes the time to live index from the database on start up and then exits.
```

Cut | Copy | Paste | AutoFill | Translate | Share... | Paste selection | Search | Share

What a CSN keeps



What a bridge node keeps



Hashmap based implementation for bridge nodes

- Leveldb based implementation
- Cache in memory

Hashmap based implementation

- Worse performance vs the pointer based implementation

Implementing the accumulator

- Bridge node implementation
- CSN implementation
- Consensus critical code

Consensus critical code

- Add, delete, verify

Consensus critical code

- Add, delete, verify
- No caching (cannot support tx propagation)

Consensus critical code

- Add, delete, verify
- No caching (cannot support tx propagation)
- Very simple

pytreexo.py buffers

```python
import copy
import hashlib


class Proof:
    def __init__(self, targets: [int], proof: [bytes]):
        self.targets = targets
        self.proof = proof

    def __str__(self):
        proof_str = [x.hex() for x in self.proof]
        s = "targets {}, proof {}".format(self.targets, proof_str)
        return s

    def __copy__(self):
        return Proof(self.targets.copy(), self.proof.copy())


class Stump:
    def __init__(self):
        self.numleaves = 0
        self.roots = []

    def __str__(self):
        roots_str = [x.hex() if x is not None else "None" for x in self.roots]
        s = "numleaves {}, roots {}".format(self.numleaves, roots_str)
        return s

    def __eq__(self, other):
        return self.roots == other.roots and self.numleaves == other.numleaves

    def add(self, adds: [bytes]):
        for add in adds:
            newroot = add
            row = 0
            while (self.numleaves >> row) & 1 == 1:
                root = self.roots.pop()
                if root is None:
                    continue
                else:
                    newroot = parent_hash(root, newroot)
                row += 1

            self.roots.append(newroot)
            self.numleaves += 1

    def verify(self, dels: [bytes], proof: Proof) -> [int]:
        if len(dels) != len(proof.targets):
            raise("len of dels and proof.targets differ")

        root_candidates = calculate_roots(self.numleaves, dels, proof)

        root_idxs = []
        for i in range(len(self.roots)):
            j = len(self.roots) - (i+1)
            if len(root_candidates) > len(root_idxs):
                if self.roots[j] == root_candidates[len(root_idxs)]:
                    root_idxs.append(j)

        if len(root_idxs) != len(root_candidates):
            raise("calculated roots from the proof and matched roots differ")

        return root_idxs

    def delete(self, dels: [bytes], proof: Proof):
        dels_copy = dels.copy()
        proof_copy = copy.copy(proof)
        root_idxs = self.verify(dels_copy, proof_copy)

        modified_roots = calculate_roots(self.numleaves, None, proof)

        for i, idx in enumerate(root_idxs):
            self.roots[idx] = modified_roots[i]


def parent_hash(left, right):
    if left is None: return right
    if right is None: return left
    return hashlib.new('sha512_256', left+right).digest()


def parent(pos: int, total_rows: int) -> int:
    return (pos >> 1) | (1 << total_rows)


def tree_rows(n: int) -> int:
    return 0 if n == 0 else (n - 1).bit_length()


def row_maxpos(row: int, total_row: int) -> int:
    mask = (2 << total_row) - 1
    return ((mask << int(total_row-row)) & mask) - 1


def parent(pos: int, total_rows: int) -> int:
    return (pos >> 1) | (1 << total_rows)


def tree_rows(n: int) -> int:
    return 0 if n == 0 else (n - 1).bit_length()


def row_maxpos(row: int, total_row: int) -> int:
    mask = (2 << total_row) - 1
    return ((mask << int(total_row-row)) & mask) - 1


def root_position(leaves: int, row: int, total_rows: int) -> int:
    mask = (2 << total_rows) - 1
    before = leaves & (mask << (row + 1))
    shifted = (before >> row) | (mask << (total_rows + 1 - row))
    return shifted & mask


def isroot(position: int, numleaves: int, row: int, total_rows: int) -> bool:
    root_present = numleaves & (1 << row) != 0
    rootpos = root_position(numleaves, row, total_rows)
    return root_present and rootpos == position


def next_least_list(list0, list1):
    if list0 and list1:
        if list0[0] < list1[0]:
            return 0
        else:
            return 1
    elif list0 and not list1:
        return 0
    elif not list0 and list1:
        return 1
    else:
        return None


def calculate_roots(numleaves: int, dels: [bytes], proof: Proof) -> [bytes]:
    total_rows = tree_rows(numleaves)

    if not proof.targets:
        return []

    dels = dels if dels is not None else [None] * len(proof.targets)
    proof.targets, dels = (list(t) for t in zip(*sorted(zip(proof.targets, dels))))

    next_hashes, next_positions, calculated_roots = [], [], []

    row = 0
    while row <= total_rows:
        pos, cur_hash = -1, bytes
        sib_present, sib_pos, sib_hash = False, -1, bytes

        index = next_least_list(proof.targets, next_positions)
        if index is None:
            break

        pos = proof.targets.pop(0) if index == 0 else next_positions.pop(0)
        cur_hash = dels.pop(0) if index == 0 else next_hashes.pop(0)

        while pos > row_maxpos(row, total_rows):
            row += 1

        if isroot(pos, numleaves, row, total_rows):
            calculated_roots.append(cur_hash)
            continue

        index = next_least_list(proof.targets, next_positions)
        if index is not None:
            sib_pos = proof.targets[0] if index == 0 else next_positions[0]
            if pos | 1 == sib_pos:
                sib_present = True
                sib_pos = proof.targets.pop(0) if index == 0 else next_positions.pop(0)
                sib_hash = dels.pop(0) if index == 0 else next_hashes.pop(0)

        next_hash = bytes
        if sib_present:
            next_hash = parent_hash(cur_hash, sib_hash)
        else:
            proofhash = proof.proof.pop(0)

            if pos & 1 == 0:
                next_hash = parent_hash(cur_hash, proofhash)
            else:
                next_hash = parent_hash(proofhash, cur_hash)

        next_hashes.append(next_hash)
        next_positions.append(parent(pos, total_rows))

    return calculated_roots
```

NORMAL ⑂ main pytreexo.py python utf-8[unix] 0% ln:1/174≡%:1

pytreexo.py python utf-8[unix] 99% ln:173/174≡%:1

[0] 1:fish 2:fish- 3:fish 4:fish 5:fish 6:fish 7:fish 8:fish 9:fish 10:nvim*

CPU:32.4% | MEM:38% | SW:4% | 0.23 0.21

Consensus critical code

- Add, delete, verify
- No caching (cannot support tx propagation)
- Very simple
- Can only support blocks-only

Maybe a separate client?

Progress towards end user readiness

Things done & things left to do

- ✓ Accumulator design
- ✓ Working full node
- ✓ Wallet support
- P2P protocol that supports caching
- Efficient mempool with Utreexo

Progress towards end user readiness

Things done & things left to do

- ✓ Accumulator design
- ✓ Working full node
- ✓ Wallet support
- ✓ P2P protocol that supports caching
- ✓ Efficient mempool with Utreexo

Progress towards Utreexo adoption

Things left to do

- BIPs

Post

Chris Guida | Run ₿ Knots | LNHANCE | ⚡ @cguida6 · 5/17/24

Is there a draft BIP for utreexo anywhere?

@kcalvinalvinn
@tdryja

2 1 467

Calvin @kcalvinalvinn · 5/17/24

@Erik17192799 and I am writing them up now.

2 2 10 307

Anthony Towns @ajtowns · 1h

WHERE IS IT???

1 29

Calvin @kcalvinalvinn · 1h

Ok I've been doing too many conferences so haven't made much progress.

There's like drafts that we're still working on that needs work.

1 35

Anthony Towns
@ajtowns

ENACT THE DRAFT!!! NATIONAL SERVICE IS MANDATORY!!!

10:20 AM · 6/12/24 From Earth · **25** Views

Related posts

Calvin @kcalvinalvinn · 1s

YES SIR

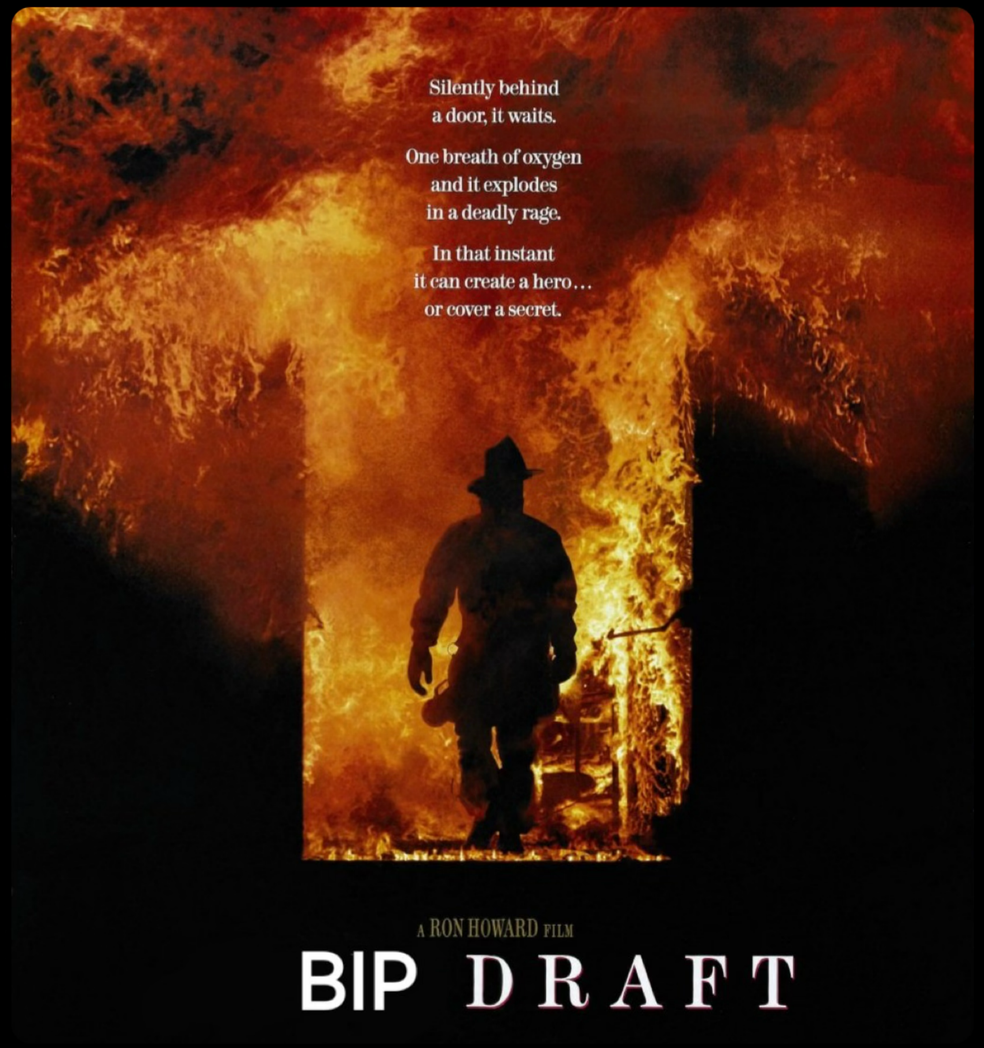
https://x.com/ajtowns/status/1800802883413442639

Anthony Towns
@ajtowns

BIP DRAFT IS BEST DRAFT!

11:26 AM · 6/12/24 From Earth

Anthony Towns
@ajtowns



11:36 AM · 6/12/24 · **37** Views

1 Like

Progress towards Utreexo adoption

Things left to do

- BIPs
- Performance improvements

Progress towards Utreexo adoption

Things left to do

- BIPs
- Performance improvements
- Robust parallel block sync

Faster Blockchain Validation with Utreexo Accumulators

BitMEX Research 19 May 2021

Abstract: In this piece, 100x Group grantee Calvin Kim announces the success in speeding up Bitcoin's Initial Block Download (IBD) using the Utreexo client. While the speed improvement can vary depending on one's local hardware and bottlenecks, the initial download and verification can be up to 62% faster compared to Bitcoin Core. Since many optimisations are yet to be implemented, the speedup is expected to increase. Calvin goes on to talk about how the IBD can be split up into multiple tasks, and therefore conducted by multiple computers. Eventually the plan is to implement Utreexo in C++ and get it merged into Bitcoin Core. While this may be a long way off, the C++ implementation has already been started.

https://blog.bitmex.com/faster-blockchain-validation-with-utreexo-accumulators/

Progress towards Utreexo adoption

Things left to do

- BIPs
- Performance improvements
- Robust parallel block sync
- Bitcoin Core implementation(?)

"I think I'll only work on this for one more year"

Calvin Kim

Progress towards Utreexo adoption

Things left to do

- BIPs
- Performance improvements
- Robust parallel block sync
- Bitcoin Core implementation(?)
- Wallet adoption

utreexod

Build and Test passing | license ISC | godoc reference

utreexod is a full node bitcoin implementation with support for utreexo accumulators. Utreexo accumulator is an append only merkle forest data structure with support for deleting elements from the set. More information at Utreexo library github repository

The main features over a traditional node are:

- Immediate node bootstrap by having the UTXO state hardcoded into the codebase.
- Uses a tiny amount of memory.
- Extremely low disk i/o needed compared to a traditional node. Will not wear out micro sd cards.

The catch is that it uses more bandwidth compared to a normal node. For block downloads it's around 1.7 times more data. For transactions the absolute worst case is 4 times more but transaction relay supports caching so it'll be a lot better.

This project is currently under active development and is in a beta state. Using it on mainnet for anything other than negligible amounts of bitcoin is not recommended.

Requirements

Go 1.18 or newer.

Rust 1.73.0 or newer (To compile the built in BDK wallet support).

Floresta

Welcome to Floresta, a lightweight Bitcoin full node implementation written in Rust, powered by Utreexo a novel dynamic accumulator designed for the Bitcoin UTXO set.

This project is composed of two parts, `libfloresta` and `florestad`. `libfloresta` is a set of reusable components that can be used to build Bitcoin applications. `florestad` is built on top of `libfloresta` to provide a full node implementation, including a watch-only wallet and an Electrum server. If you just want to run a full node, you can use `florestad` directly, either by building it from source or by downloading a pre-built binary from the releases.

If you want to use `libfloresta` to build your own Bitcoin application, you can find the documentation here.

Comunity

If you want to discuss this project, you can join our Discord server here.

Building

You'll need Rust and Cargo, refer to this for more details. Minimum support version is rustc 1.74 and newer.

Once you have Cargo, clone the repository with:

```
git clone https://github.com/Davidson-Souza/Floresta.git
```

go to the Floresta directory

```
cd Floresta/
```

https://github.com/kcalvinalvin/2024-praha-devhackday