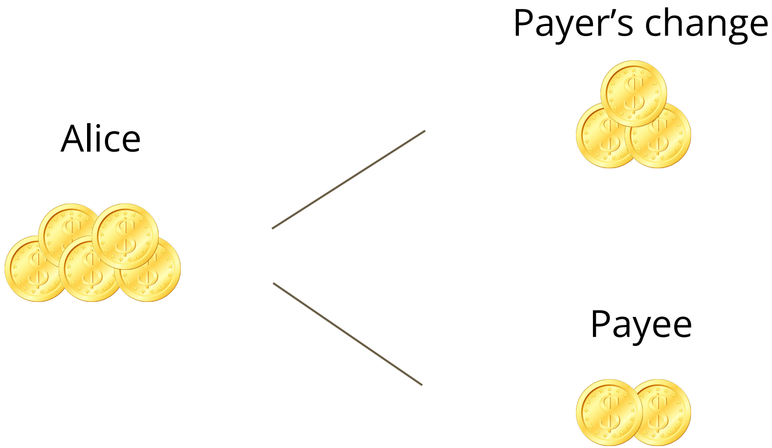
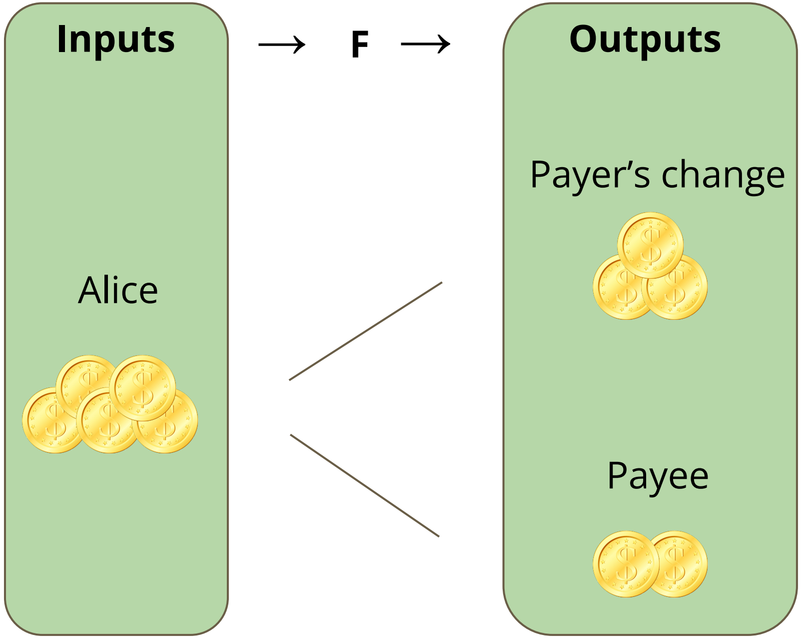
Utreexo

🌳🌳🌳🌳🌳🌳🌳🌳🌳🌳🌳🌳

Bitcoin transactions



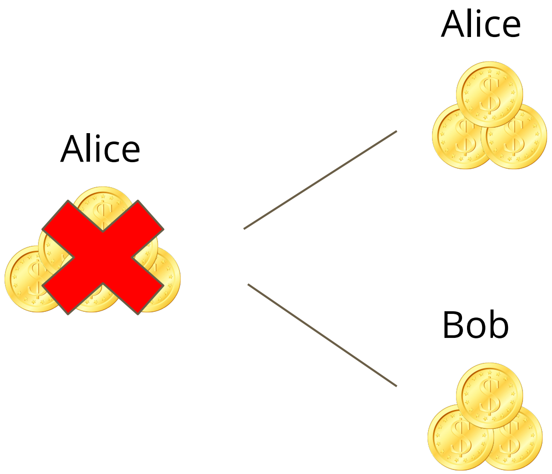
Bitcoin transactions



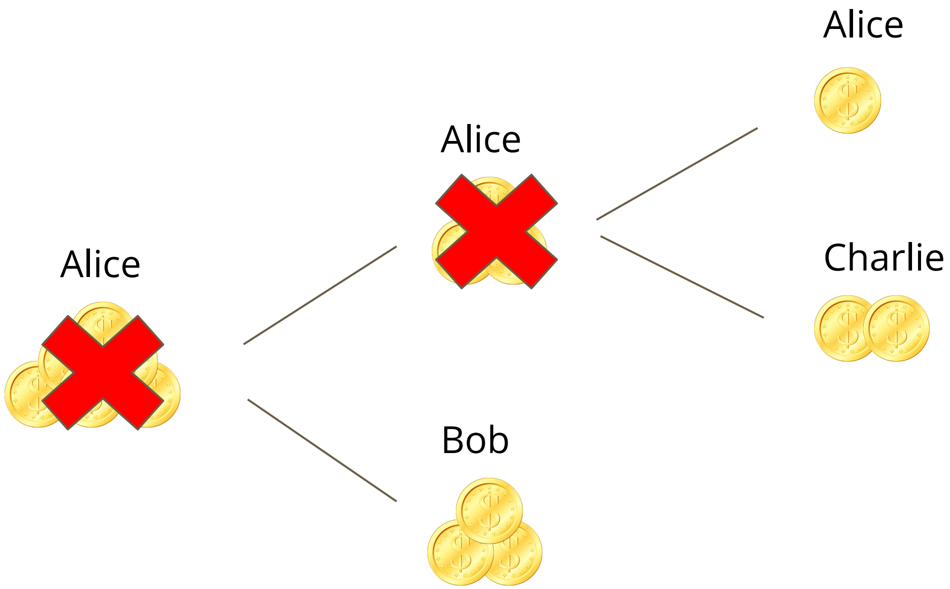
Bitcoin transactions

Alice

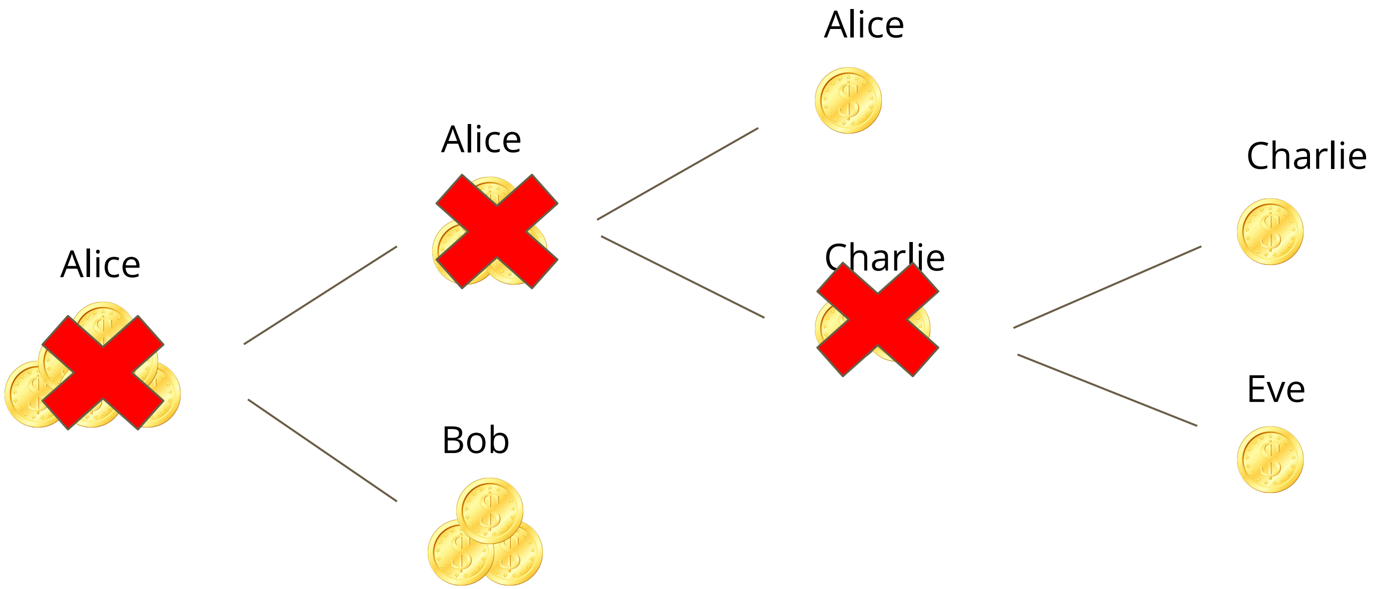
Bitcoin transactions



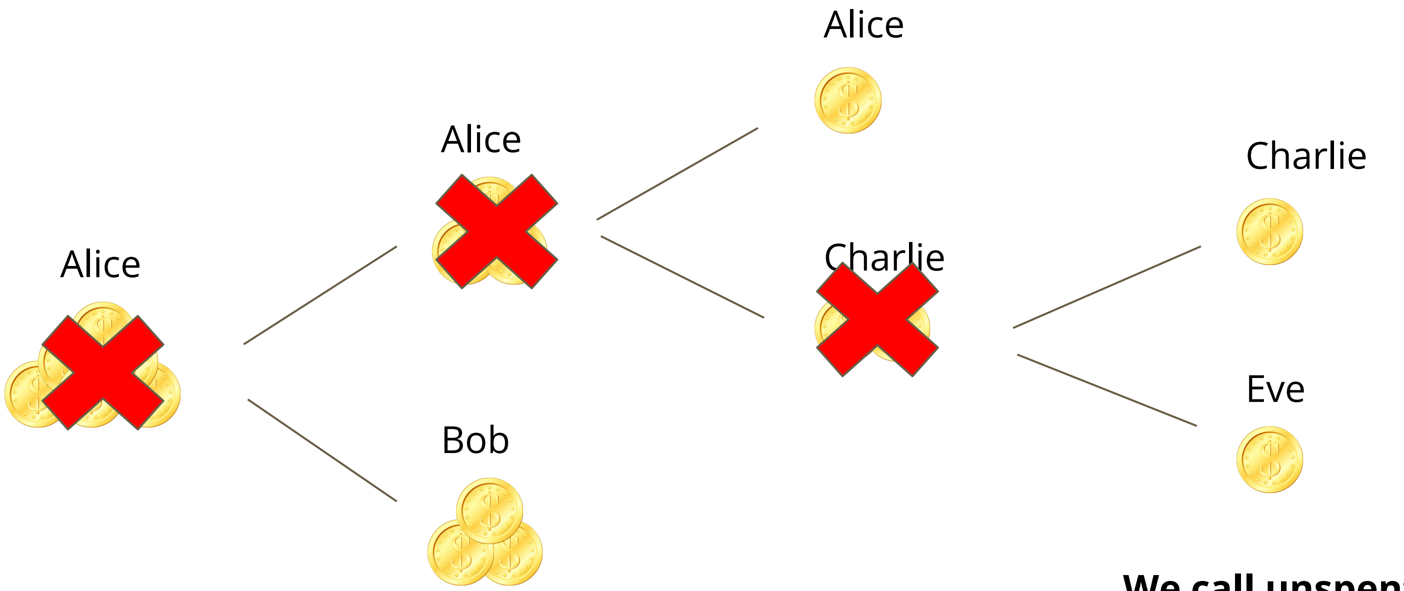
Bitcoin transactions



Bitcoin transactions



Bitcoin transactions



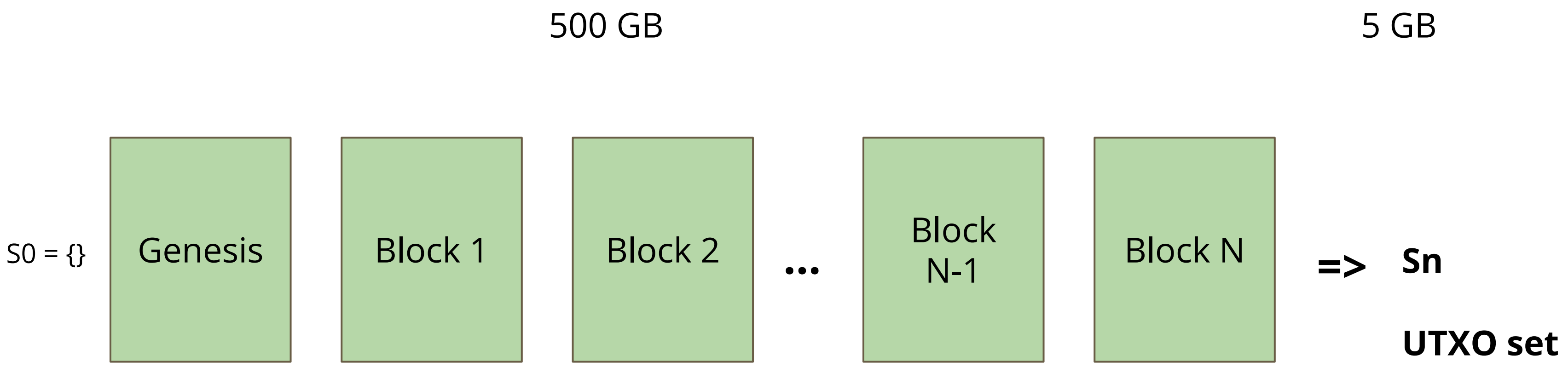
We call unspent piles
Unspent Transaction Output (UTXO)

UTXO set

A set of unspent coins

Bitcoin UTXO set
has over **130 MILLION** piles of coins (outputs) which is ~5GB of data.

Bitcoin sync

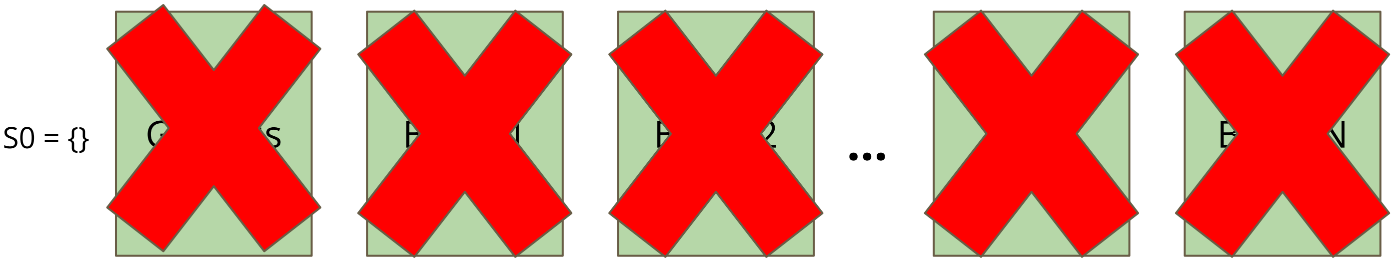


We download all the blocks and execute them. When we've executed all of the blocks, we arrive at the current state of the chain i.e. the current UTXO set.

Bitcoin sync with pruning

500 GB

5 GB



=> **Sn**

UTXO set

We download all the blocks and execute them. When we’ve executed all of the blocks, we arrive at the current state of the chain i.e. the current UTXO set.

We only keep the UTXO set!

Non-dormant state

UTXO set size: ~130M
Block inputs: ~5000

A very small set of outputs is active in a block. Do we really need to know about every output on each block?

What if instead of caring about every UTXO, we only cared about those that are actually being touched in the update/block?

Cryptographic Accumulators

Given a set of elements, we can:

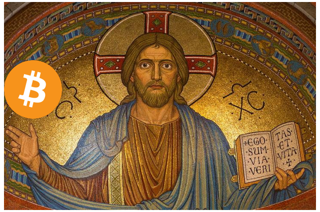
- Add element to the set
- Remove element from the set
- Prove element is in the set
- Prove element is not in the set

without revealing other elements in the set.

Based on the operations the accumulator supports, we mark it as static, dynamic or universal accumulator.

Utreexo (dynamic accumulator) 🌳

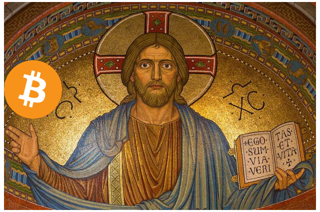
1. Supports element addition, removal and inclusion proofs
2. Accumulator size is ~1kb
3. Optimized for Bitcoin



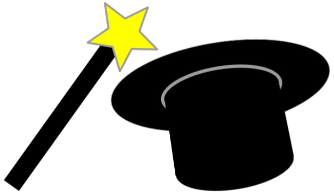
How does it work?

Utreexo (dynamic accumulator)

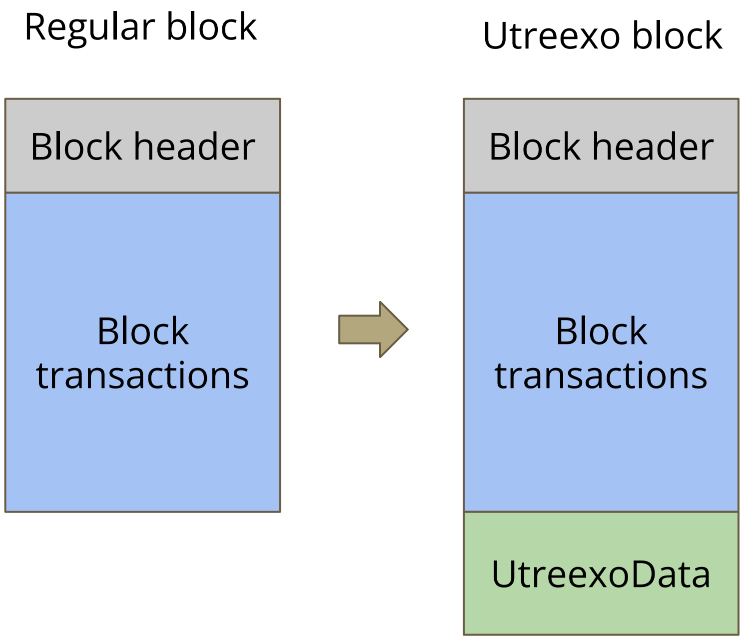
1. Supports element addition, removal and inclusion proofs
2. Accumulator size is ~1kb
3. Optimized for Bitcoin



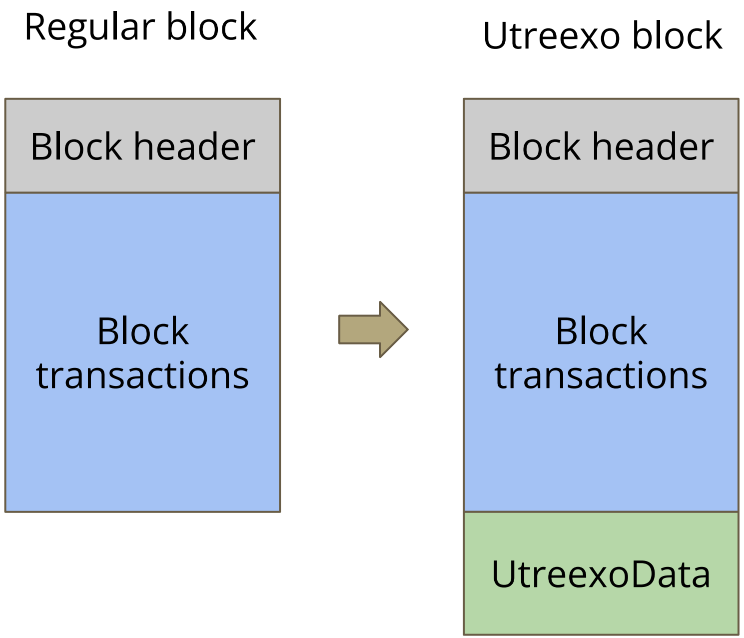
How does it work?



Utreexo block

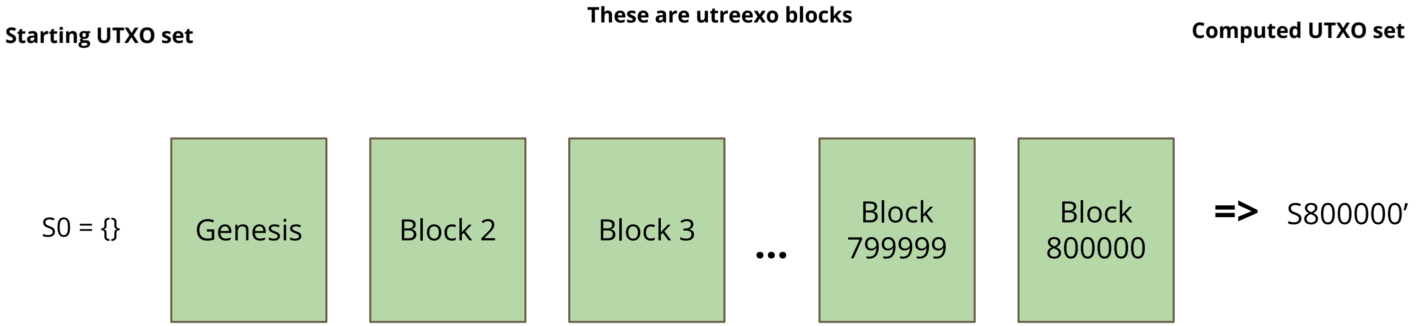


Utreexo block

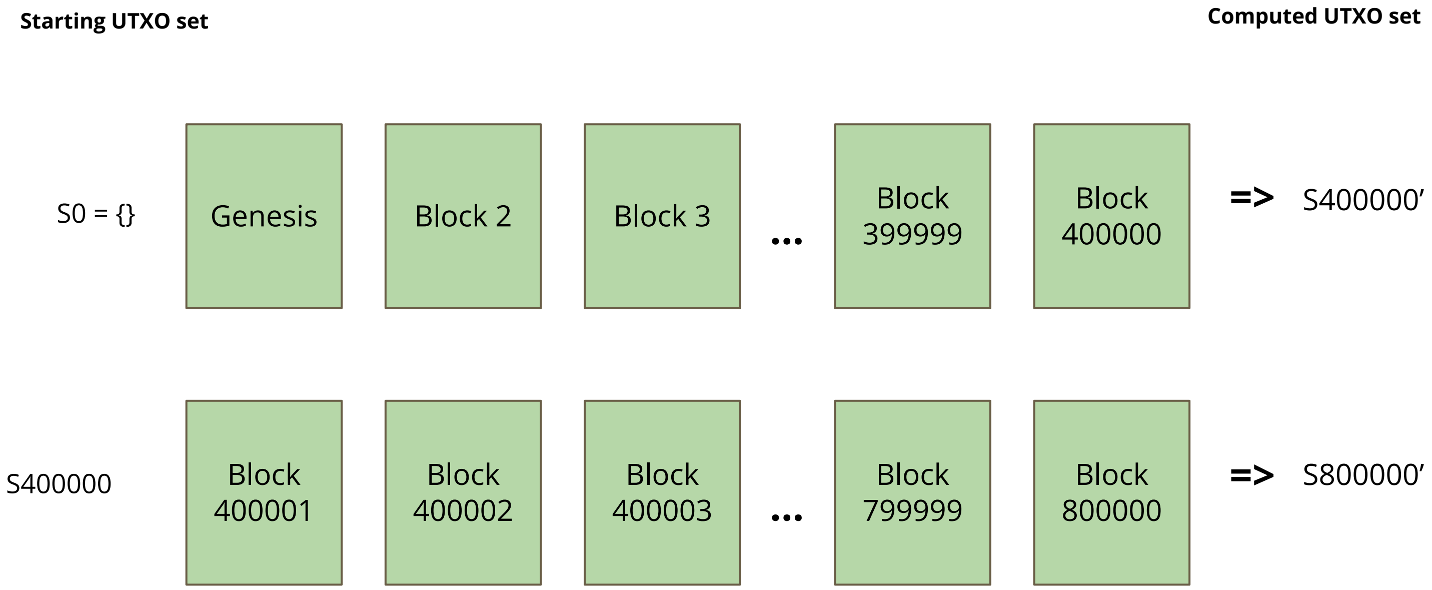


~30% more bandwidth

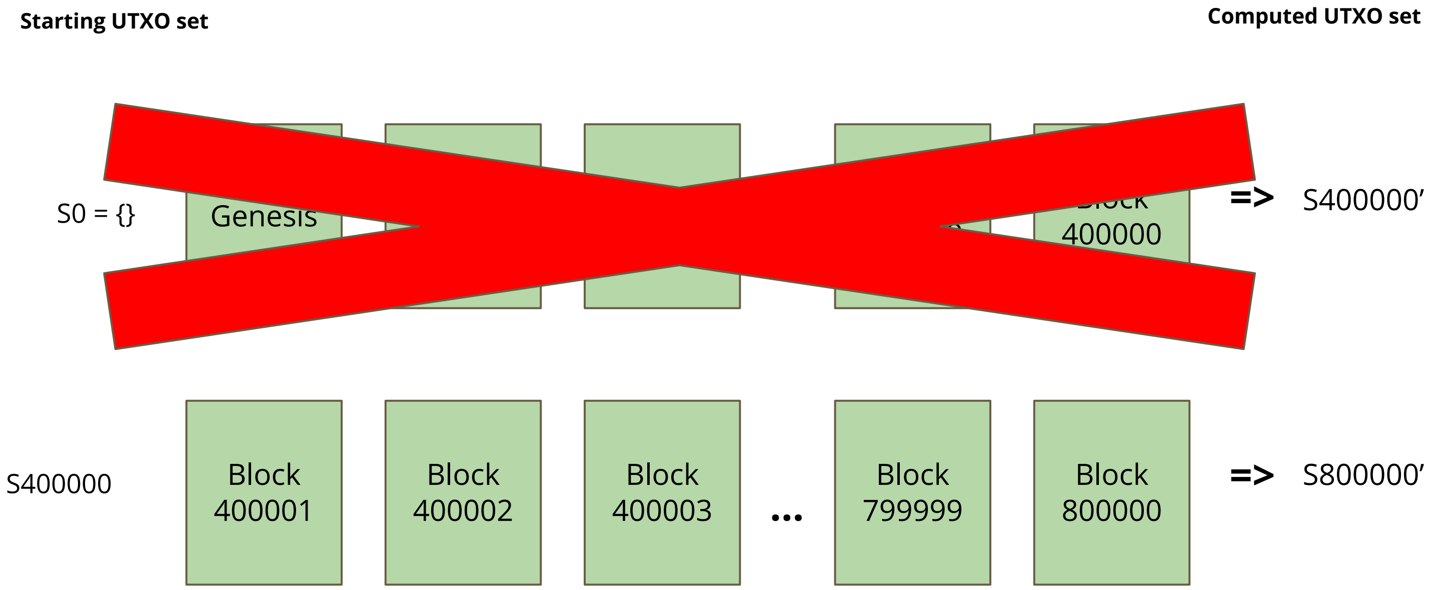
Utreexo sync



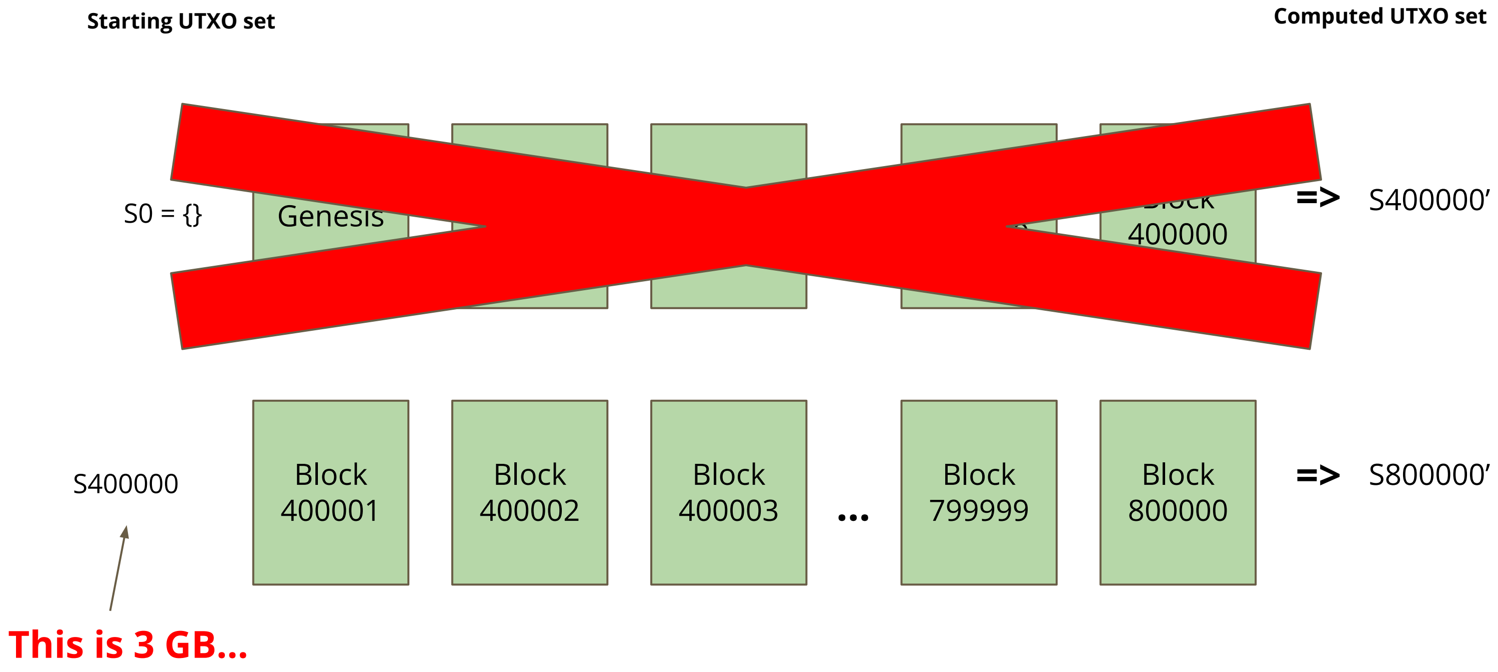
AssumeUtxo


Starting UTXO set
Computed UTXO set
S0 = {}
Genesis
Block 2
Block 3
...
Block 399999
Block 400000
=> S400000’
S400000
Block 400001
Block 400002
Block 400003
...
Block 799999
Block 800000
=> S800000’

AssumeUtxo


Starting UTXO set
Computed UTXO set
S0 = {}
Genesis
400000
=> S400000’
S400000
Block 400001
Block 400002
Block 400003
...
Block 799999
Block 800000
=> S800000’

AssumeUtxo

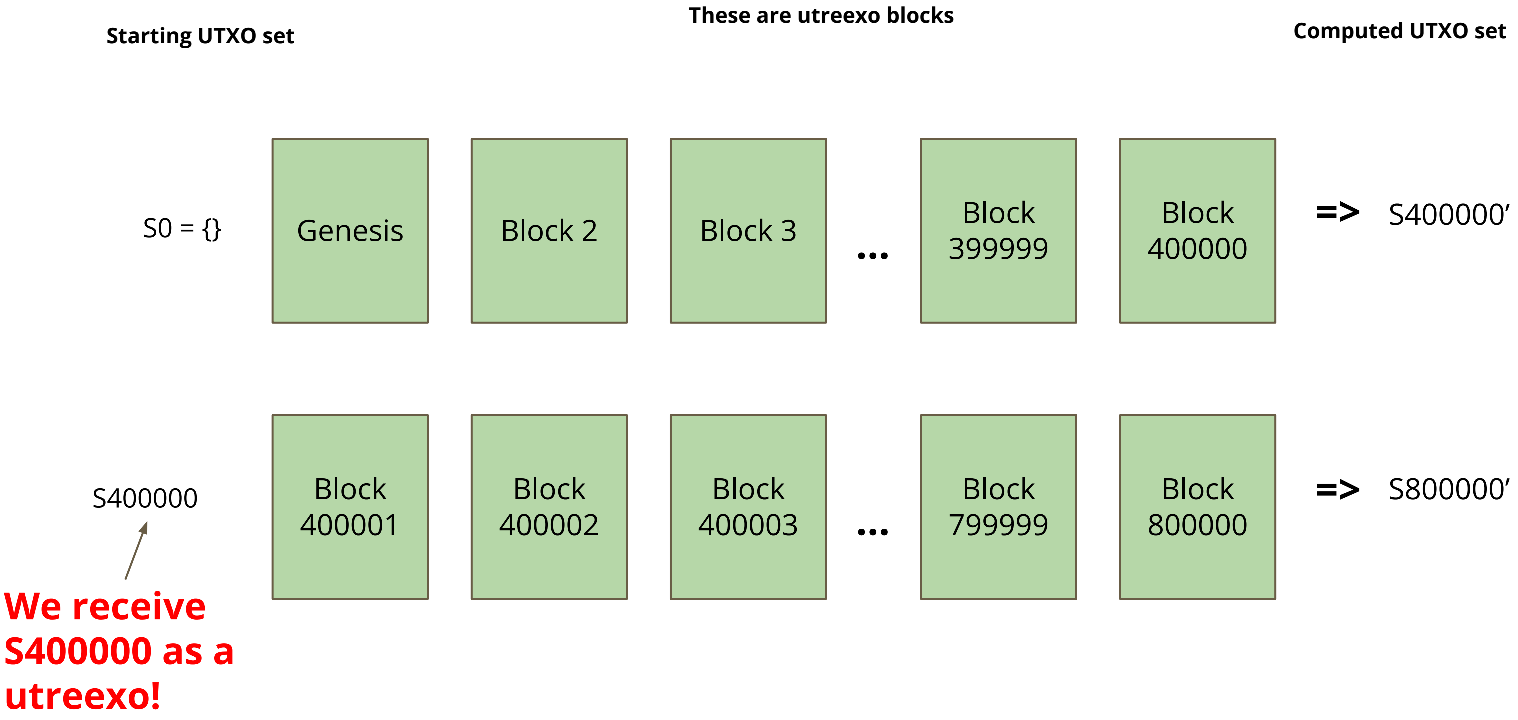


Starting UTXO set
Computed UTXO set
S0 = {}
Genesis
400000
=>
S400000’
S400000
Block 400001
Block 400002
Block 400003
...
Block 799999
Block 800000
=>
S800000’
This is 3 GB...

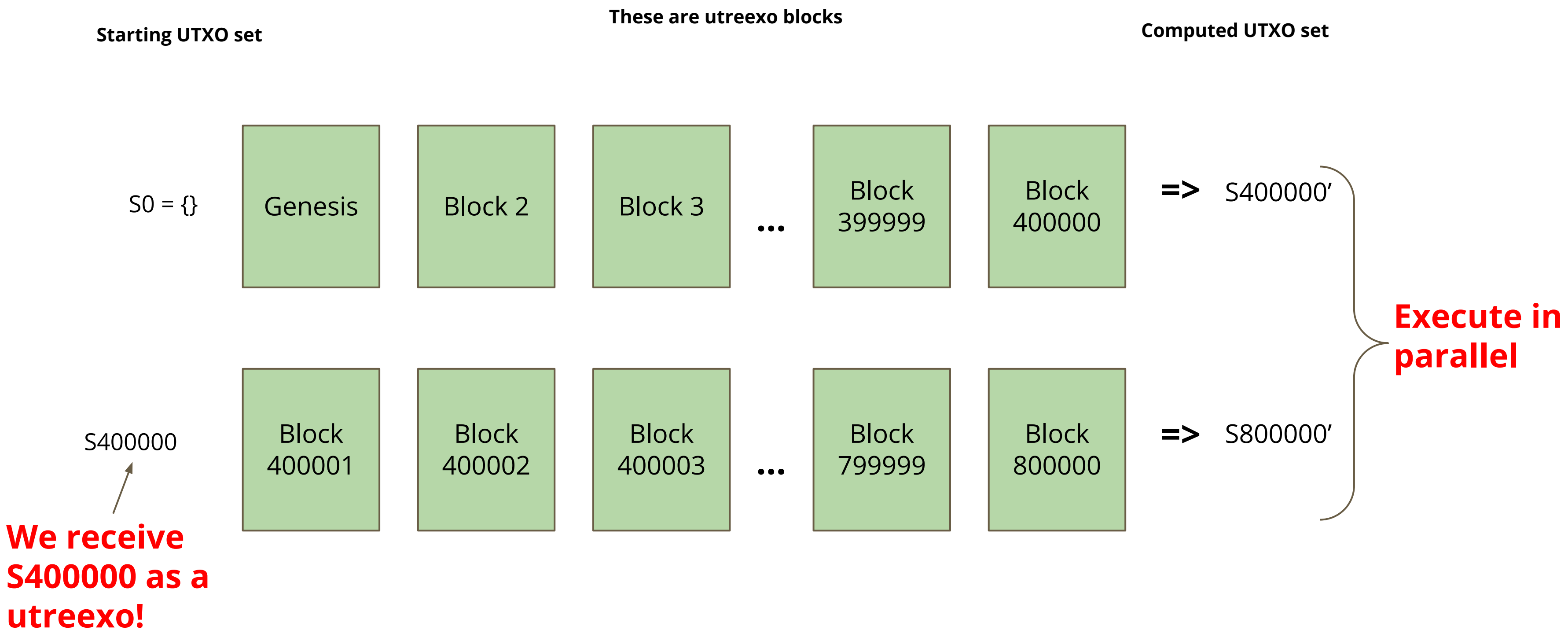
Wait a minute... S400000 can be 1kb!

We can now make the **Initial Block Download** run in **parallel**!

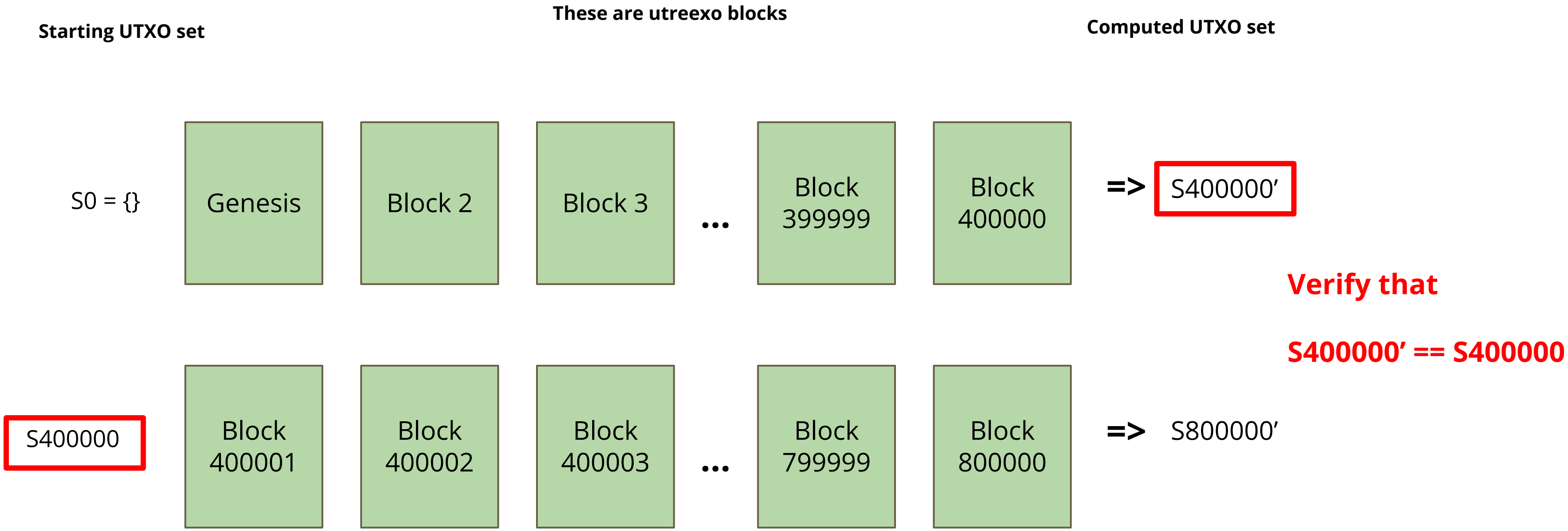
Utreexo PIBD



Utreexo PIBD



Utreexo PIBD



Utreexo PIBD

These are utreexo blocks

Starting UTXO set								Computed UTXO set
S0 = {}	Genesis	Block 2	Block 3	...	Block 99	Block 200000	=>	S200000’
S200000	Block 200001	Block 200002	Block 200003	...	Block 39999	Block 400000	=>	S400000’
S400000	Block 400001	Block 400002	Block 400003	...	Block 599999	Block 600000	=>	S600000’
S600000	Block 600001	Block 600002	Block 600003	...	Block 799999	Block 800000	=>	S800000’

Execute in parallel

Utreexo PIBD

Starting UTXO set	These are utreexo blocks							Computed UTXO set
S0 = {}	Genesis	Block 2	Block 3	...	Block 99	Block 200000	=>	S200000’
S200000	Block 200001	Block 200002	Block 200003	...	Block 39999	Block 400000	=>	S400000’
S400000	Block 400001	Block 400002	Block 400003	...	Block 599999	Block 600000	=>	S600000’
S600000	Block 600001	Block 600002	Block 600003	...	Block 799999	Block 800000	=>	S800000’

Verify that

S200000’ == S200000

Utreexo PIBD

These are utreexo blocks

Starting UTXO set								Computed UTXO set
S0 = {}	Genesis	Block 2	Block 3	...	Block 99	Block 200000	=>	S200000’
S200000	Block 200001	Block 200002	Block 200003	...	Block 39999	Block 400000	=>	S400000’
S400000	Block 400001	Block 400002	Block 400003	...	Block 599999	Block 600000	=>	S600000’
S600000	Block 600001	Block 600002	Block 600003	...	Block 799999	Block 800000	=>	S800000’

Verify that

S400000’ == S400000

Utreexo PIBD

These are utreexo blocks

Starting UTXO set									Computed UTXO set
S0 = {}	Genesis	Block 2	Block 3	...	Block 99	Block 200000	=>	S200000’
S200000	Block 200001	Block 200002	Block 200003	...	Block 39999	Block 400000	=>	S400000’
S400000	Block 400001	Block 400002	Block 400003	...	Block 599999	Block 600000	=>	$\boxed{\text{S600000’}}$
$\boxed{\text{S600000}}$	Block 600001	Block 600002	Block 600003	...	Block 799999	Block 800000	=>	S800000’

Verify that

S600000’ == S600000

Utreexo PIBD

Starting UTXO set

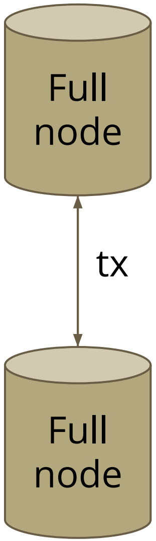
These are utreexo blocks

Computed UTXO set

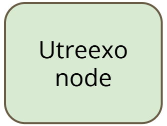
S0 = {}	Genesis	Block 2	Block 3	...	Block 99	Block 200000	=>	S200000’
S200000	Block 200001	Block 200002	Block 200003	...	Block 39999	Block 400000	=>	S400000’
S400000	Block 400001	Block 400002	Block 400003	...	Block 599999	Block 600000	=>	S600000’
S600000	Block 600001	Block 600002	Block 600003	...	Block 799999	Block 800000	=>	**S800000’**

S800000’ is the current UTXO set

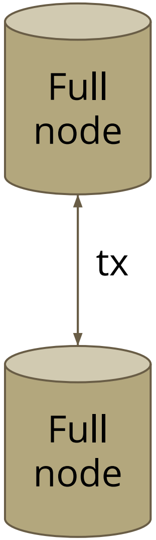
Utreexo Bridge nodes



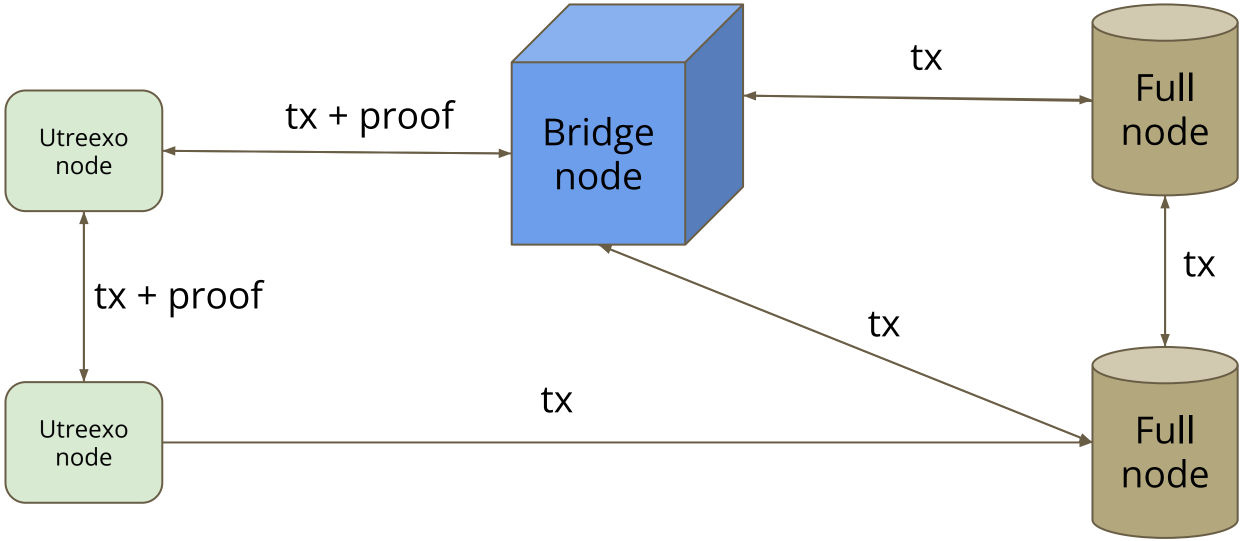
Utreexo Bridge nodes



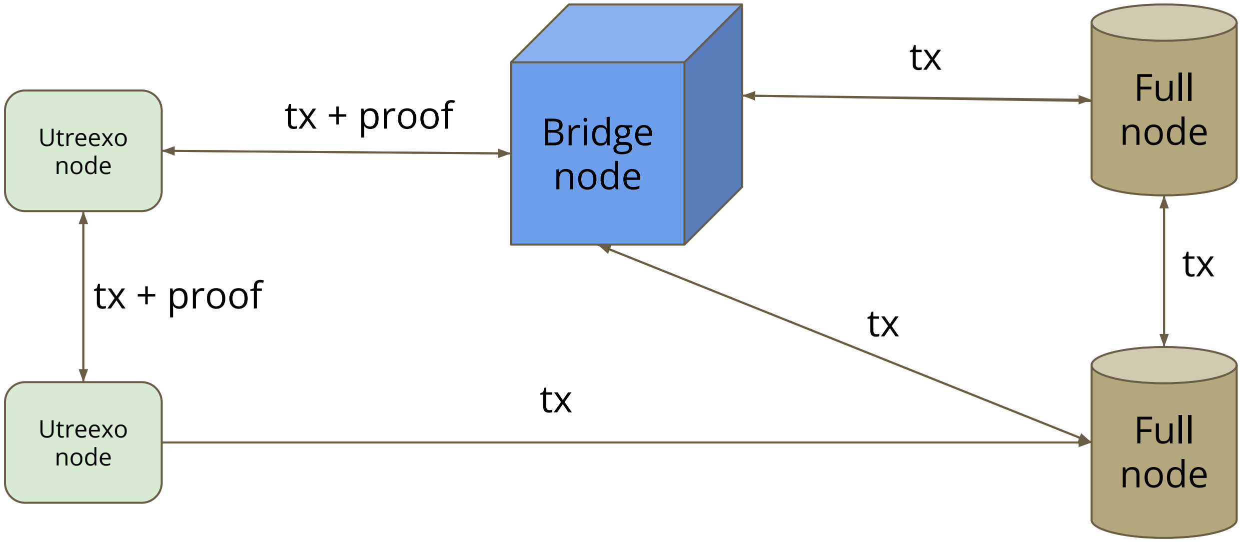
Where do I get blocks with proofs from? No way to bootstrap.



Utreexo Bridge nodes



Utreexo Bridge nodes



No soft fork needed!

Tradeoffs?

Pros:

- Compressed state from 5GB to ~1kb
- No disk IO
- No database! (levelDB)
- Parallel IBD
- No soft forks

Cons:

- More bandwidth
- Proof construction/updating

Quick summary

1. We have a large set of UTXOs (130 million)
2. We can already prune blocks, but we’re left with 5GB state (utxo set)
3. We can create a compact representation of that set (~1kb)
4. We can prove elements are in the compact set representation as well as add/delete them

We now have a node that takes very little storage, but requires a bit more data to download.

What does this enable?

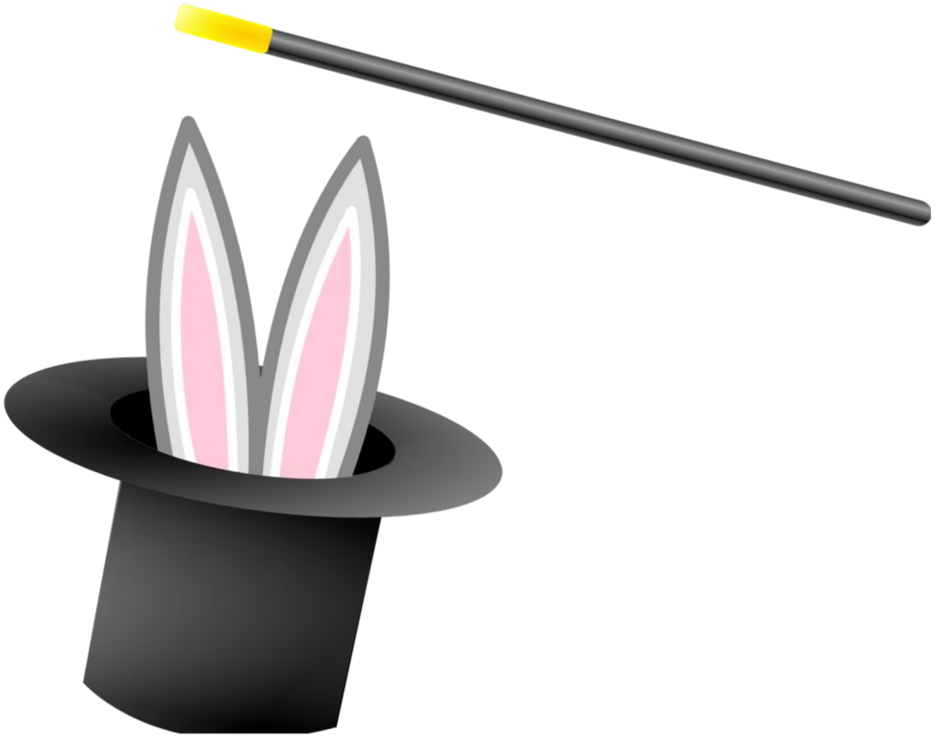
Compact full nodes.

Any device or process connected to the internet could run a compact full node without using much storage.

- Web wallet - Chrome extension node (every time we open a web browser, an extension syncs the utreexo node)

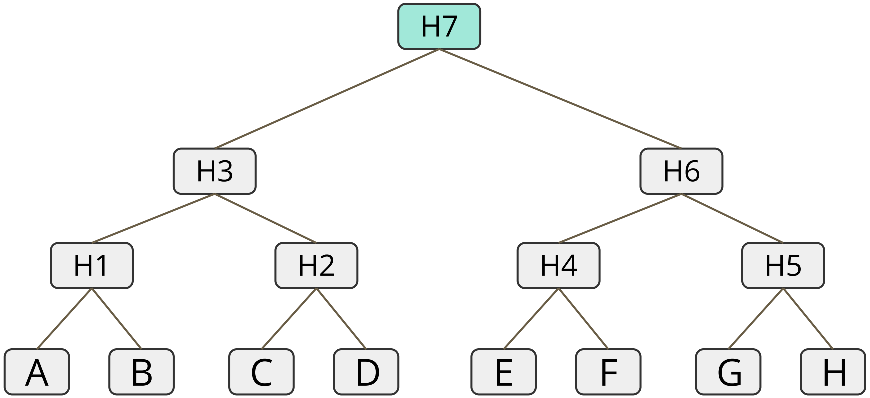
- Router node - Router could run a node in the background

Ok, but what is this magic?



Merkle trees

A merkle tree commits to a sequence of elements.

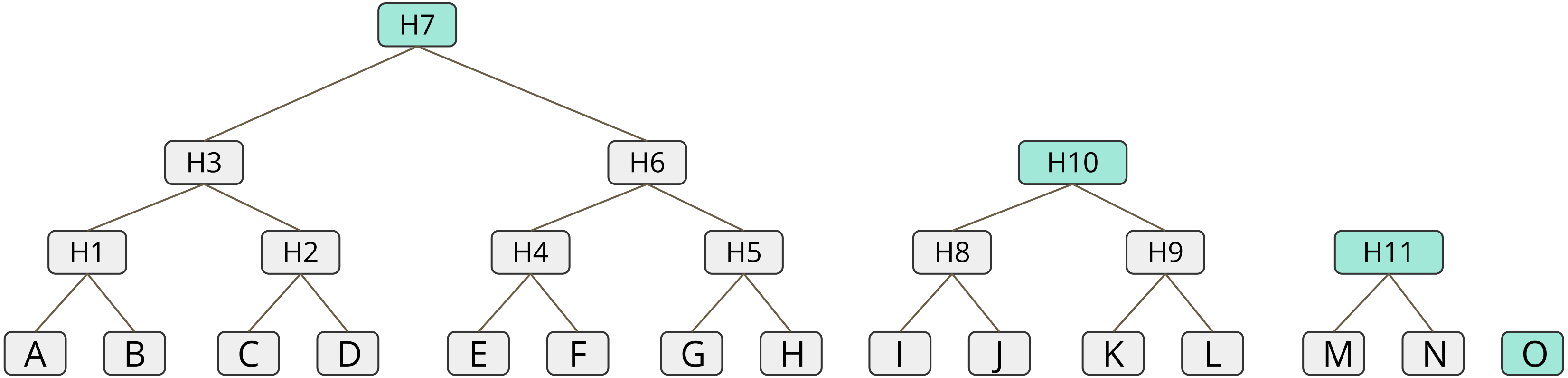


H1 commits to **A and B**.
H2 commits to **C and D**.
H3 commits to **H1 and H2** => **A, B, C and D**.

The root of this merkle tree (H7) commits to A, B, C, D, E, F, G and H which are all the 8 elements in the tree.

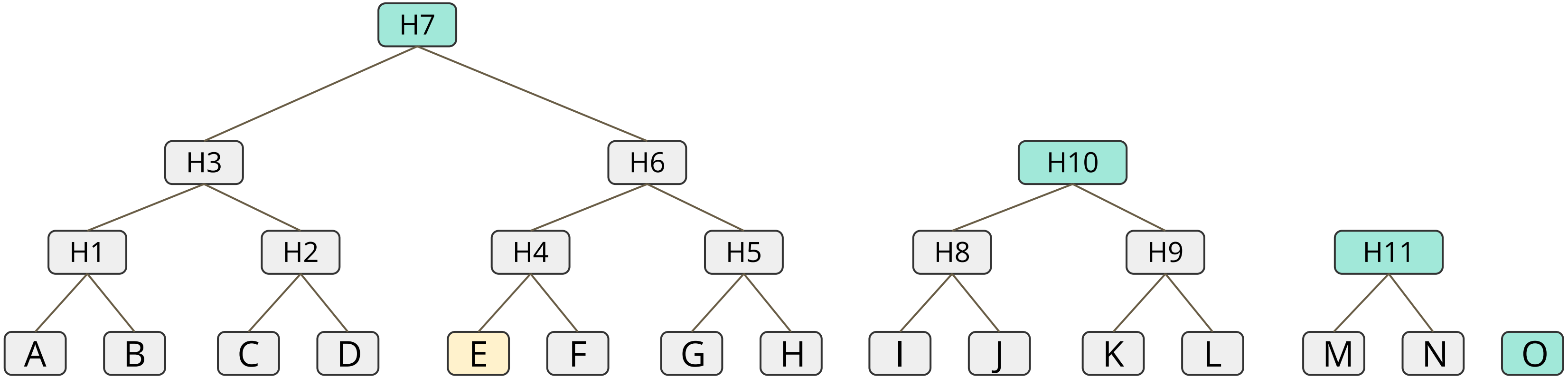
Utreexo - What does it look like?

It's a forest of perfect merkle trees. A utreexo node only keeps the roots of the trees.



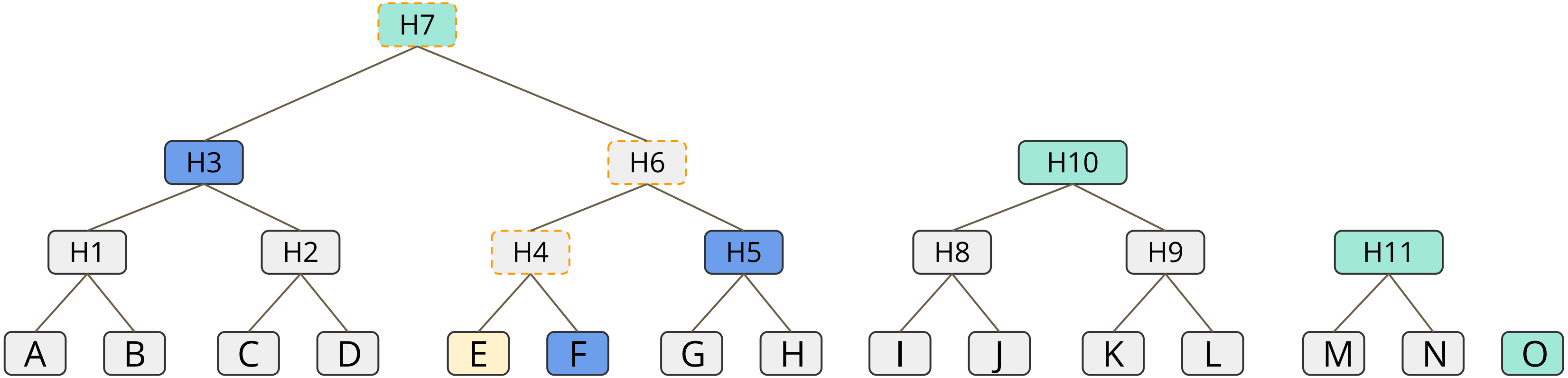
Utreexo - prove inclusion 🌳

How do we prove E is in the set?



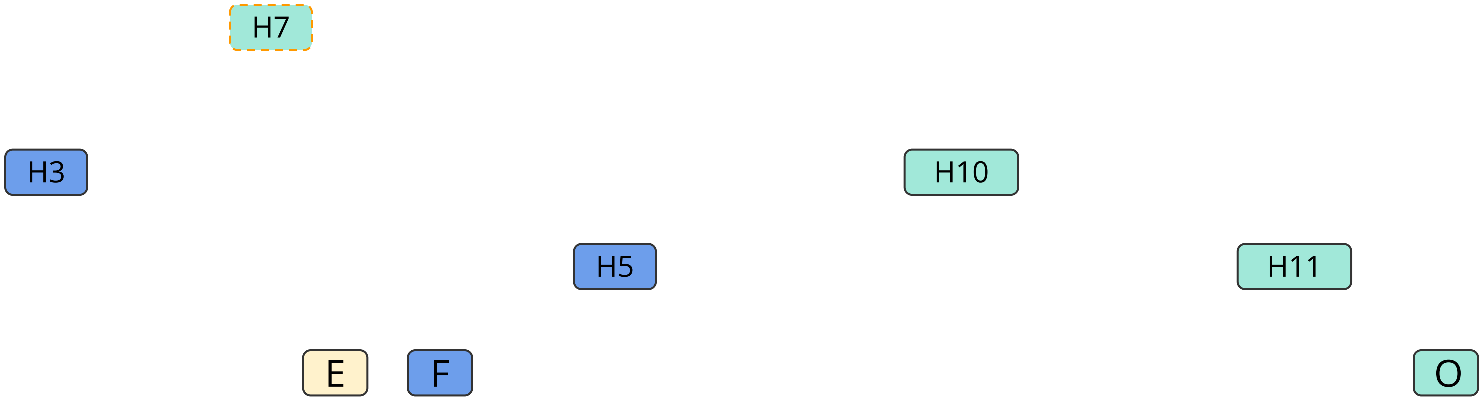
Utreexo - prove inclusion 🌳

We can prove an element E is in the tree by providing the blue parts. We can compute the path to H7 and compare if it's the same H7 we have.



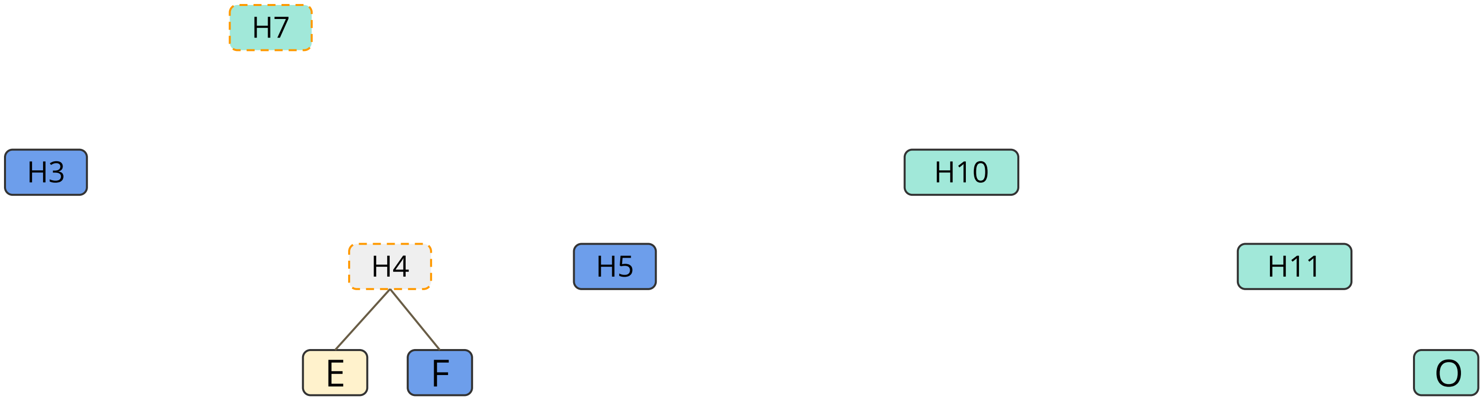
Utreexo - prove inclusion 🌳

We can prove an element E is in the tree by providing the blue parts. We can compute the path to H7 and compare if it's the same H7 we have.



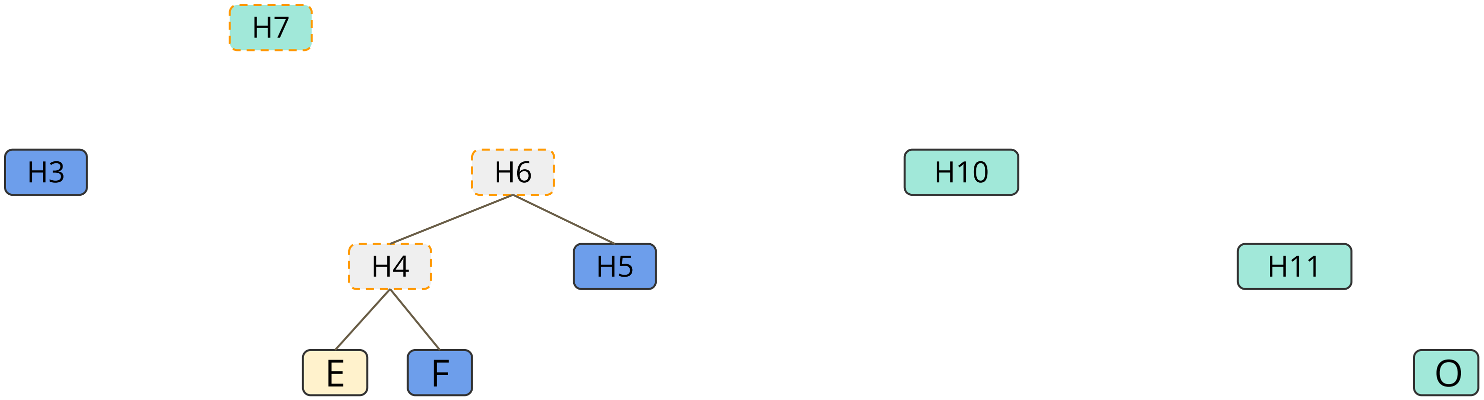
Utreexo - prove inclusion 🌳

We can prove an element E is in the tree by providing the blue parts. We can compute the path to H7 and compare if it's the same H7 we have.



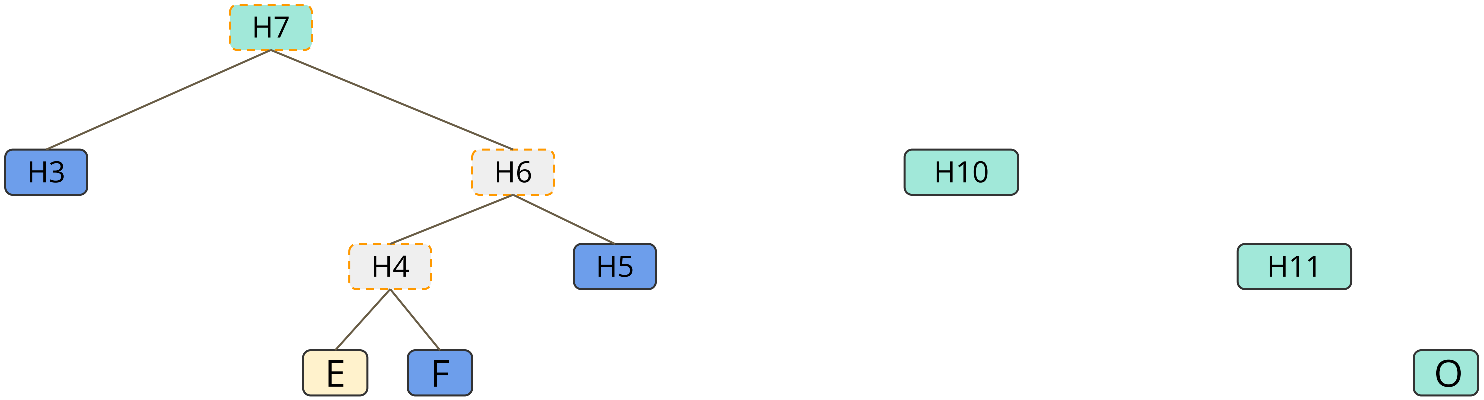
Utreexo - prove inclusion 🌳

We can prove an element E is in the tree by providing the blue parts. We can compute the path to H7 and compare if it's the same H7 we have.



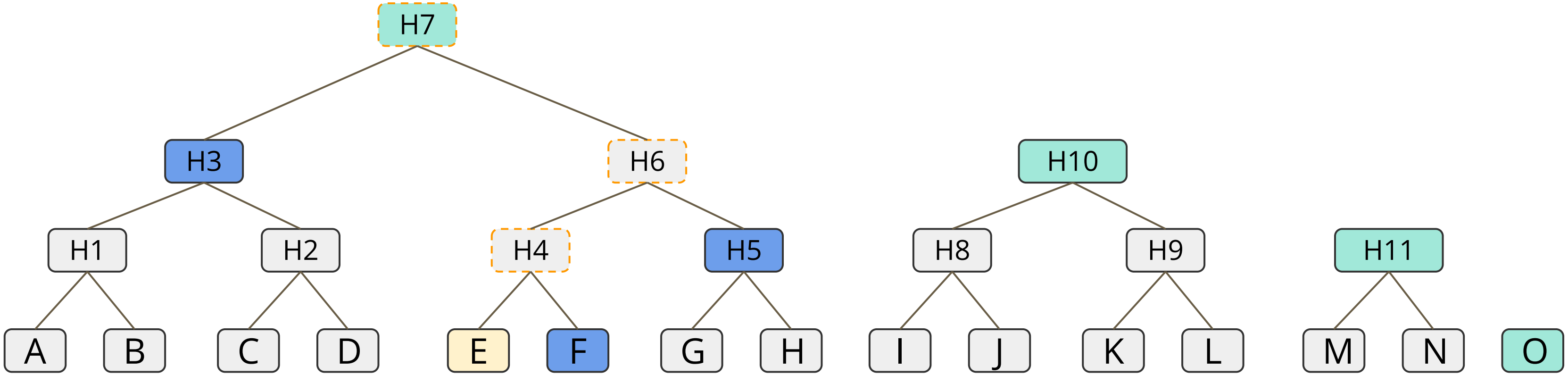
Utreexo - prove inclusion 🌳

We can prove an element E is in the tree by providing the blue parts. We can compute the path to H7 and compare if it's the same H7 we have.



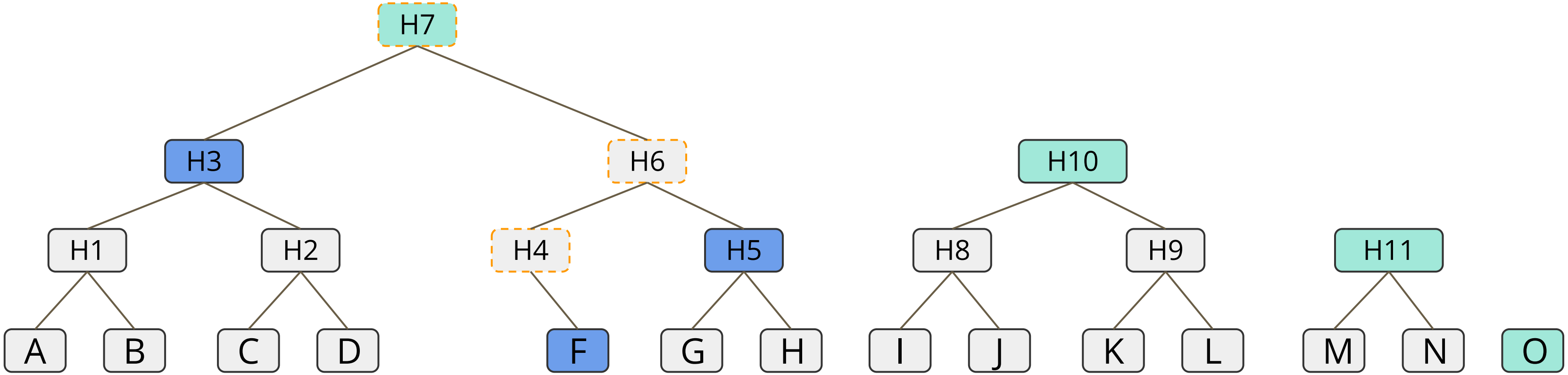
Utreexo - deletion 🌳

We use inclusion proofs to delete E and find another element to move to the position of E.



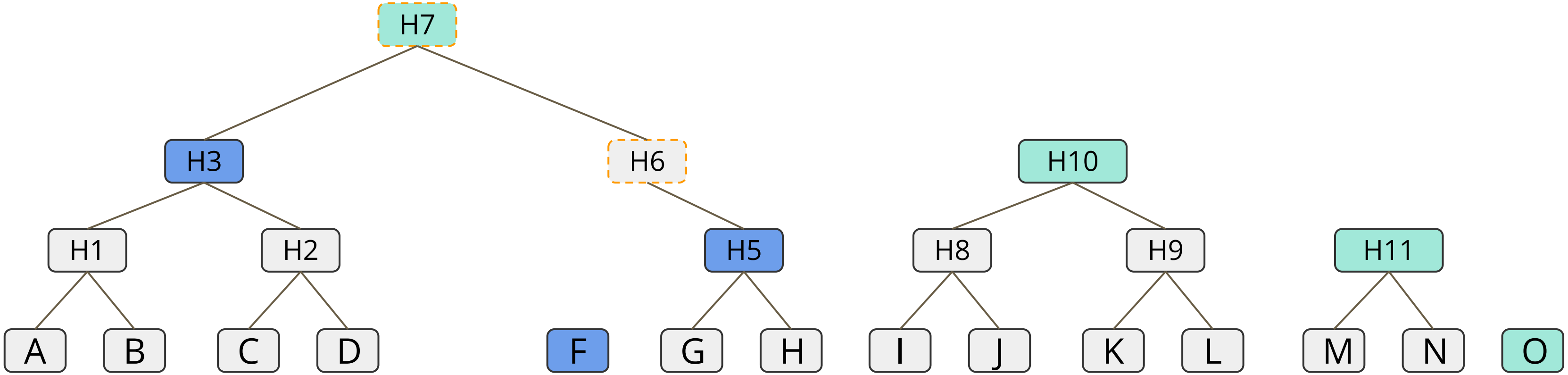
Utreexo - deletion 🌳

We use inclusion proofs to delete E and find another element to move to the position of E.



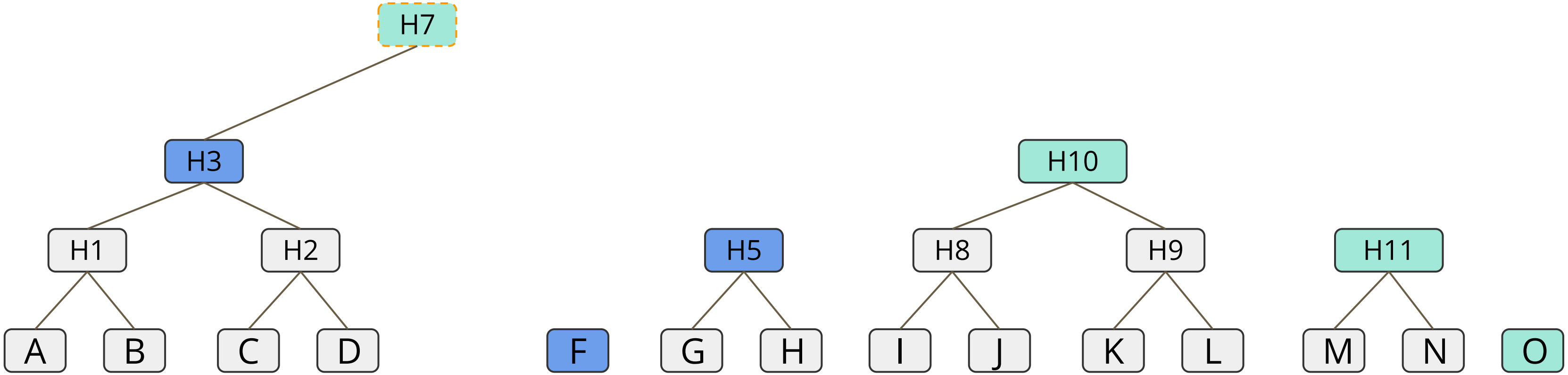
Utreexo - deletion 🌳

We use inclusion proofs to delete E and find another element to move to the position of E.



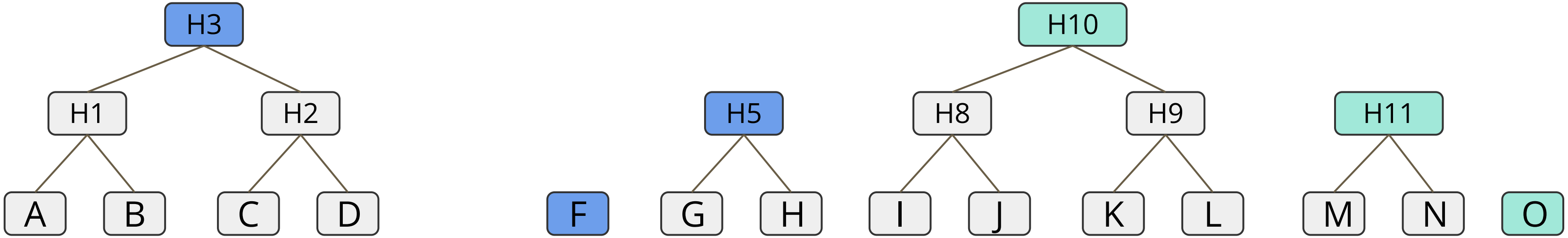
Utreexo - deletion 🌳

We use inclusion proofs to delete E and find another element to move to the position of E.



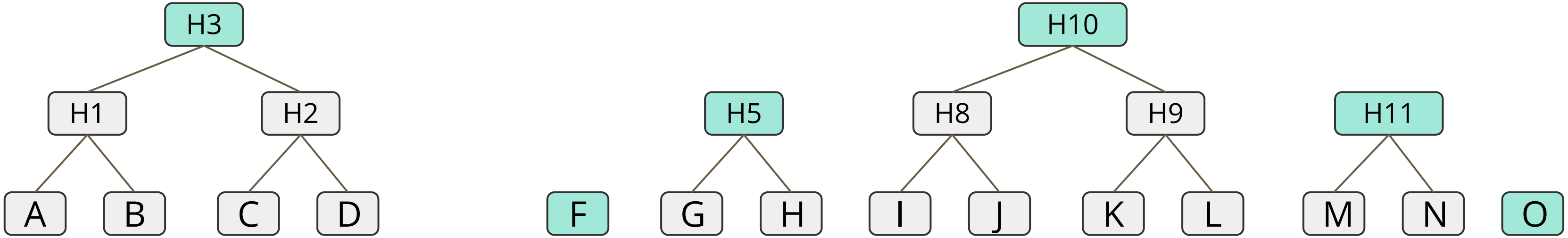
Utreexo - deletion 🌳

We use inclusion proofs to delete E and find another element to move to the position of E.



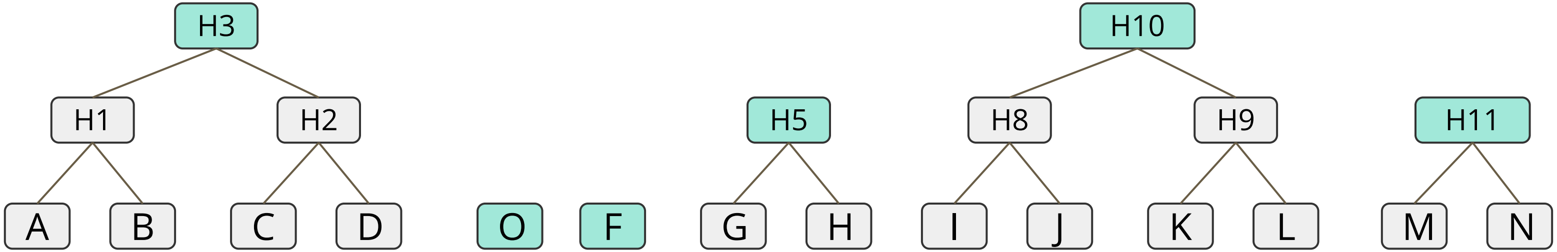
Utreexo - deletion 🌳

We use inclusion proofs to delete E and find another element to move to the position of E.



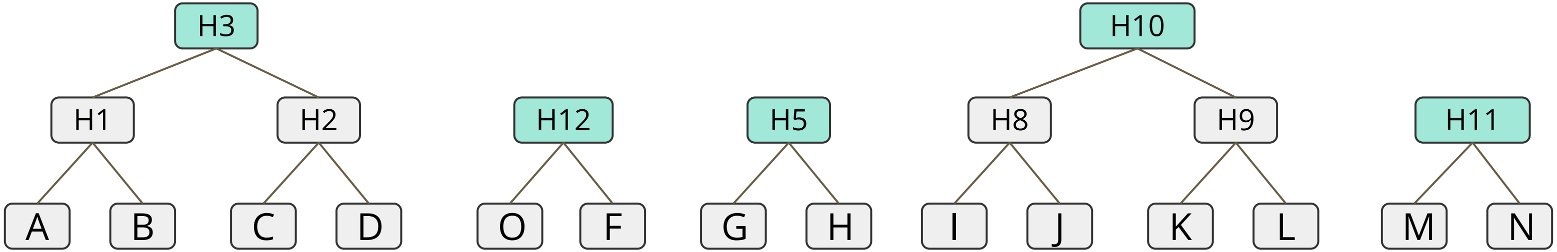
Utreexo - deletion 🌳

We use inclusion proofs to delete E and find another element to move to the position of E.



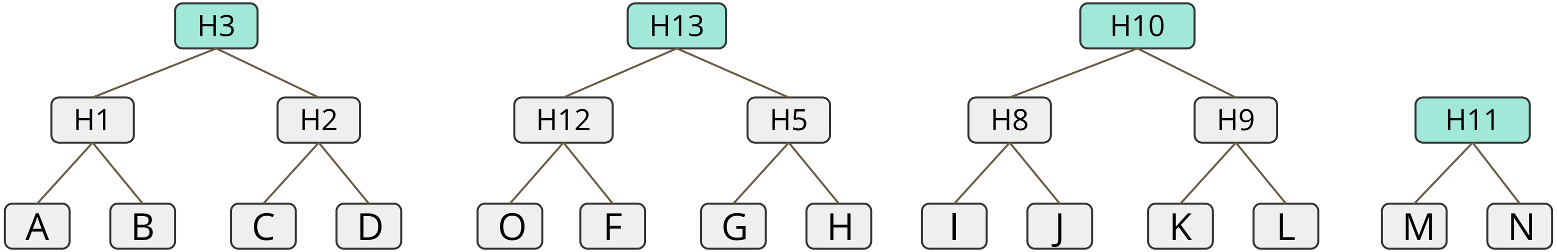
Utreexo - deletion 🌳

We use inclusion proofs to delete E and find another element to move to the position of E.



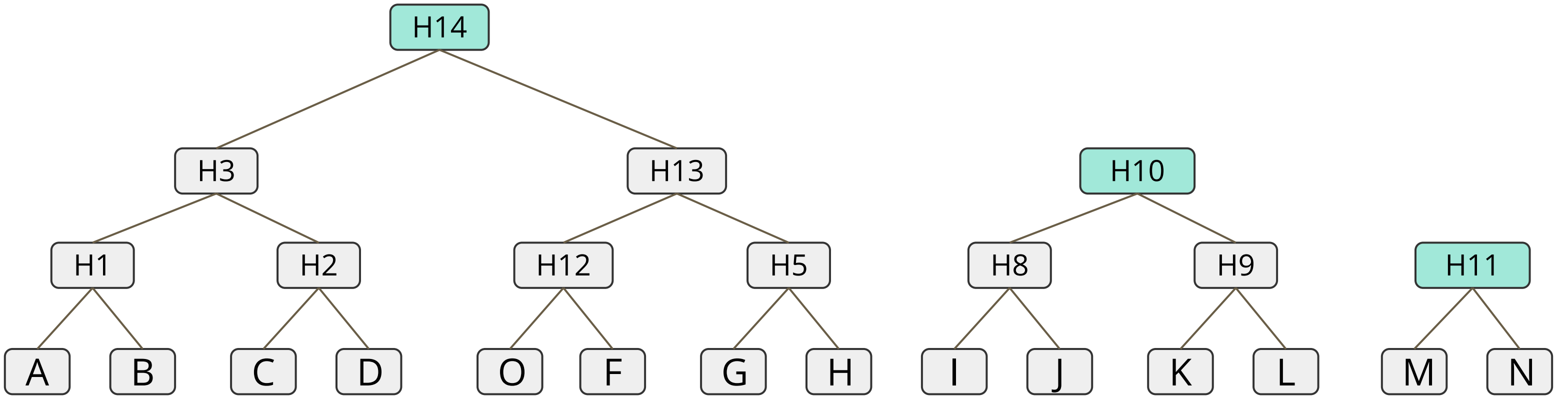
Utreexo - deletion 🌳

We use inclusion proofs to delete E and find another element to move to the position of E.



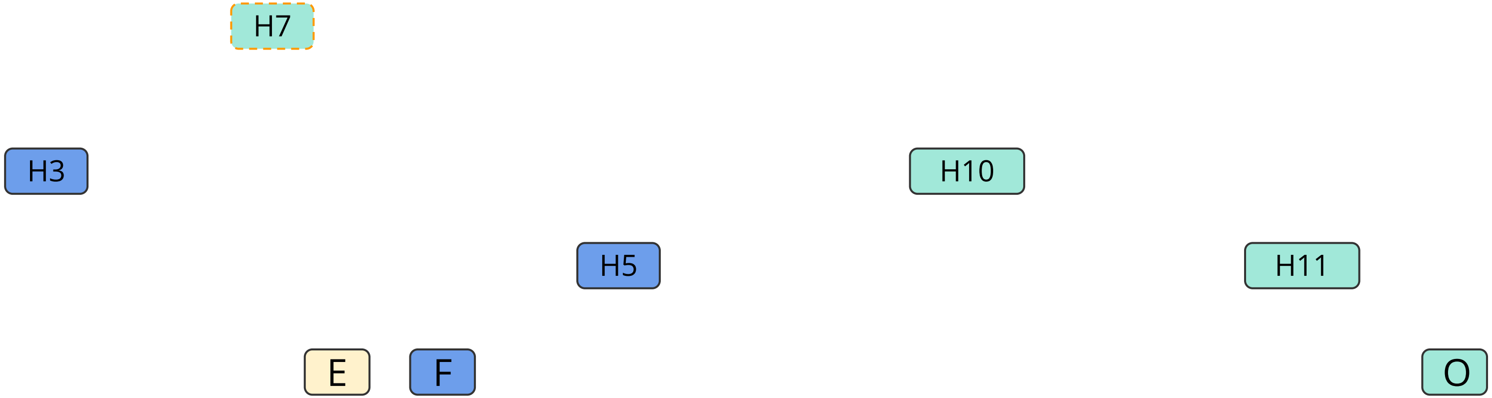
Utreexo - deletion 🌳

We use inclusion proofs to delete E and find another element to move to the position of E.



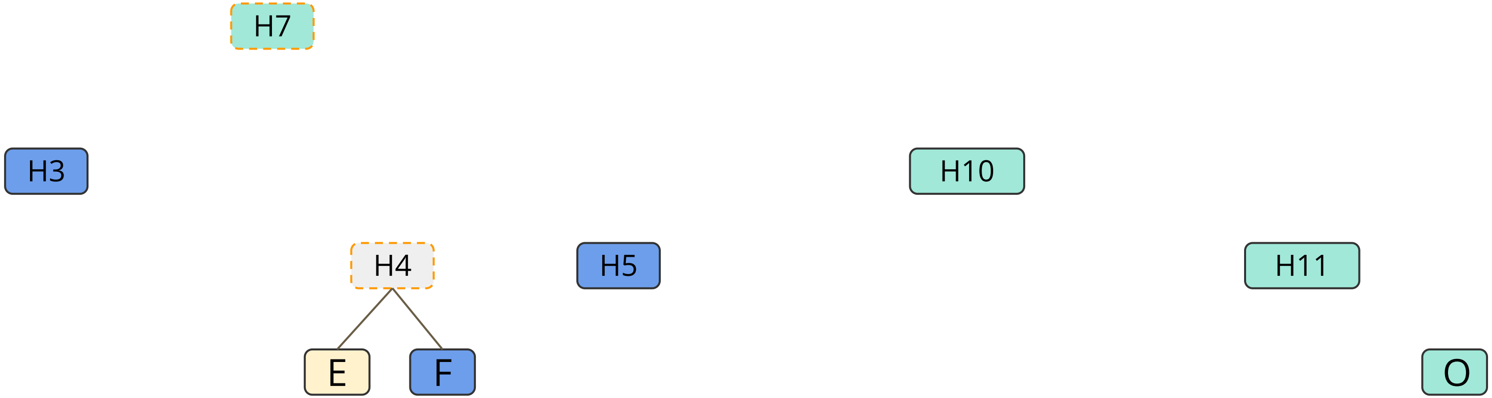
Utreexo - deletion 🌳

We use inclusion proofs to delete E and find another element to move to the position of E.



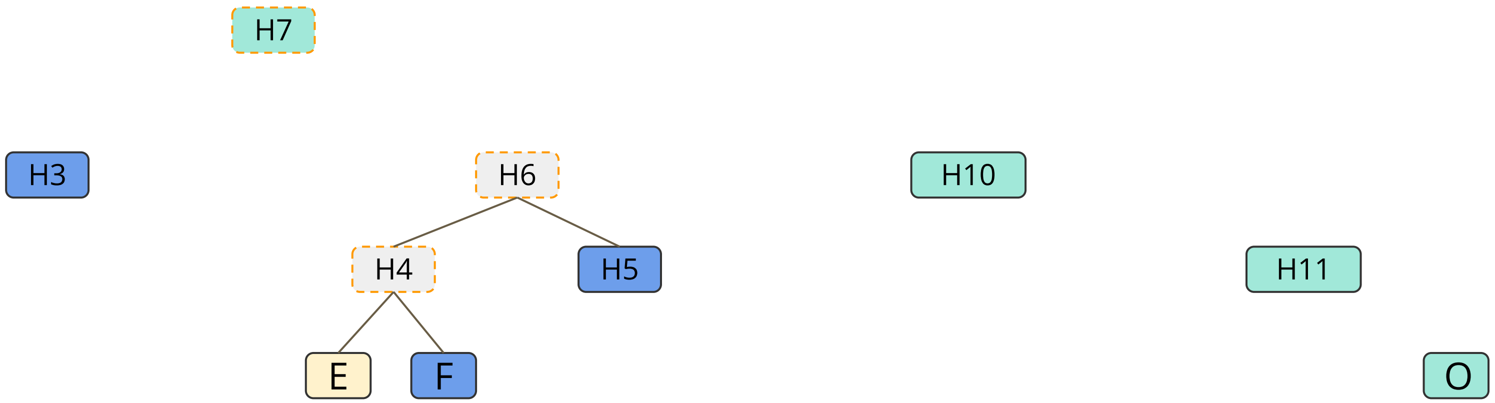
Utreexo - deletion 🌳

We use inclusion proofs to delete E and find another element to move to the position of E.



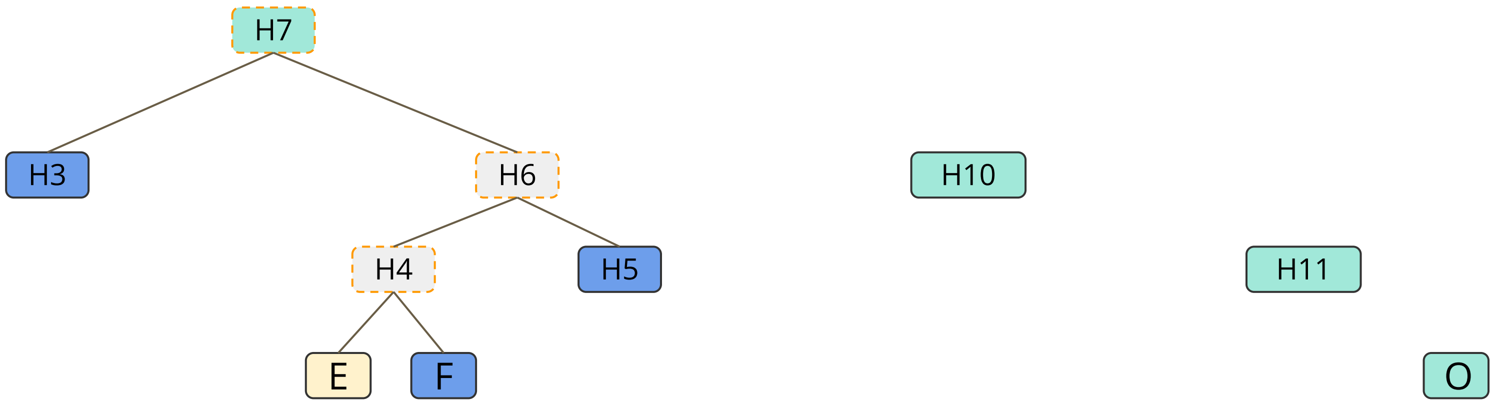
Utreexo - deletion 🌳

We use inclusion proofs to delete E and find another element to move to the position of E.



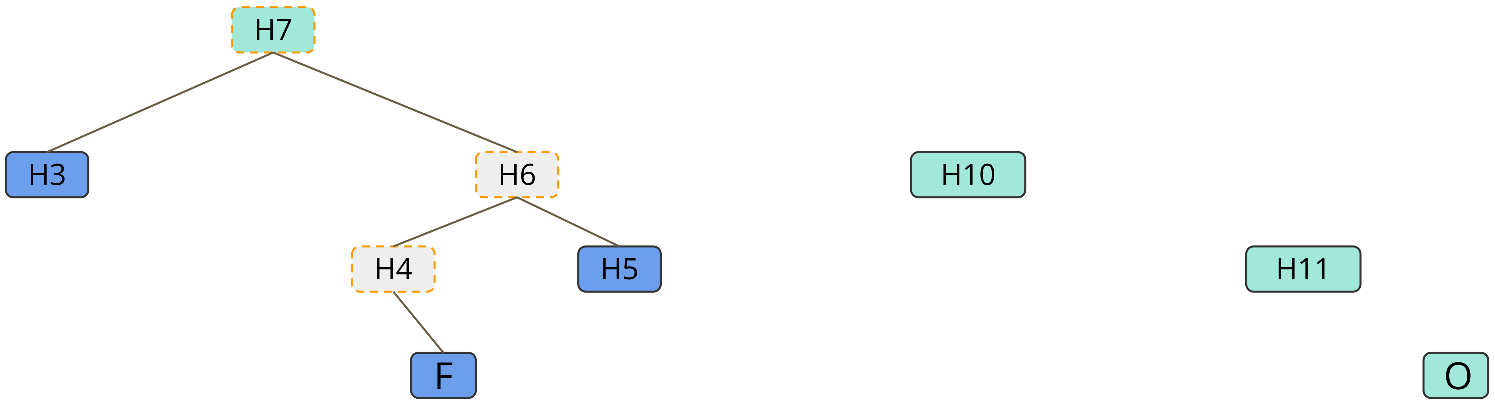
Utreexo - deletion 🌳

We use inclusion proofs to delete E and find another element to move to the position of E.



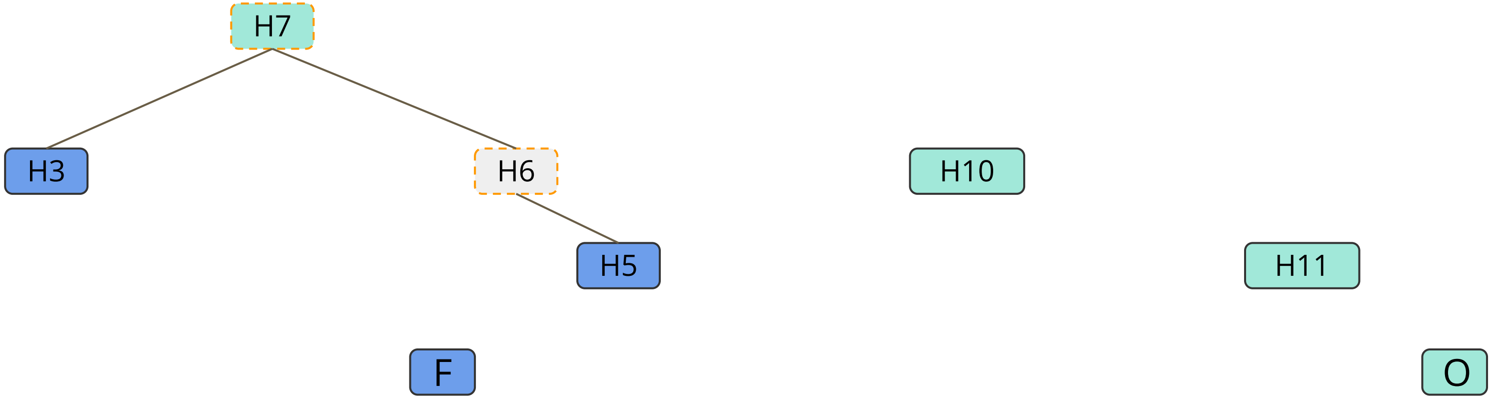
Utreexo - deletion 🌳

We use inclusion proofs to delete E and find another element to move to the position of E.



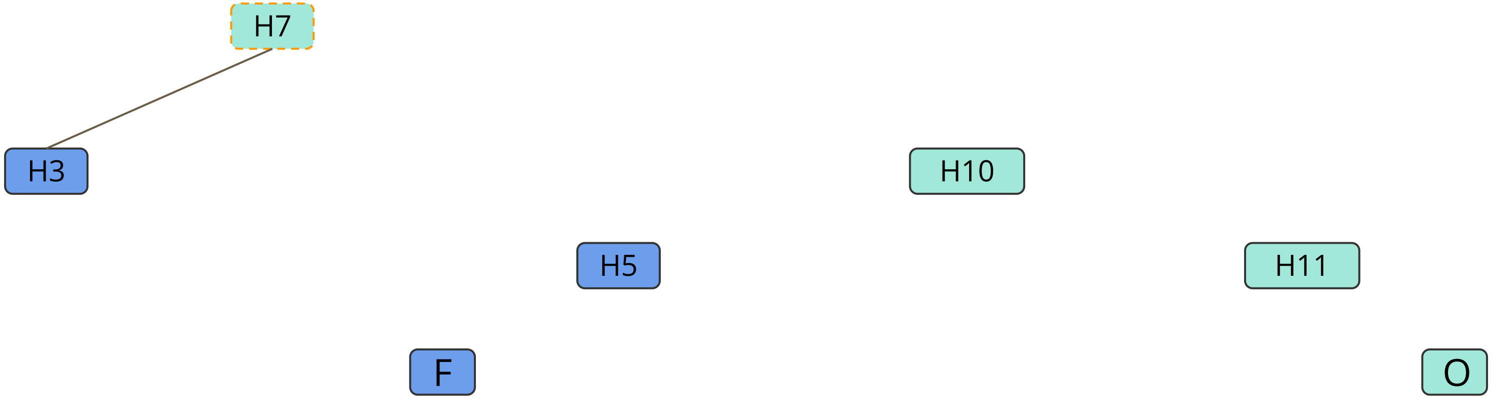
Utreexo - deletion 🌳

We use inclusion proofs to delete E and find another element to move to the position of E.



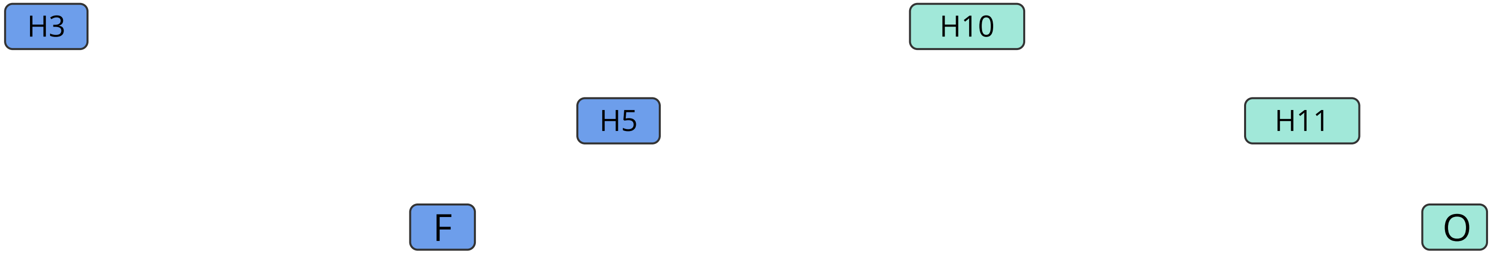
Utreexo - deletion 🌳

We use inclusion proofs to delete E and find another element to move to the position of E.



Utreexo - deletion 🌳

We use inclusion proofs to delete E and find another element to move to the position of E.

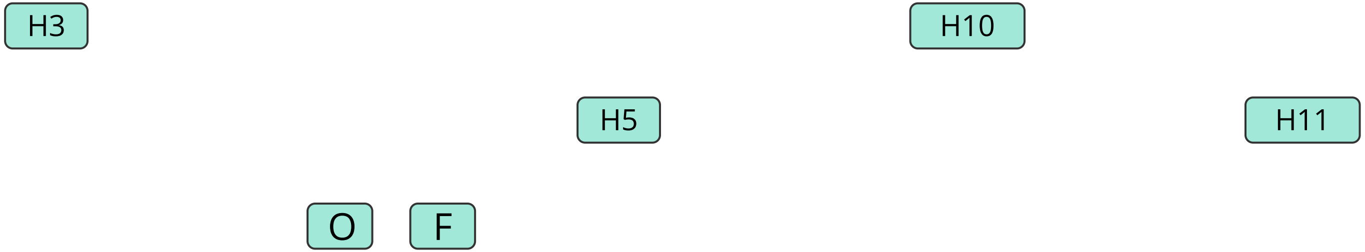


Utreexo - deletion 🌳

We use inclusion proofs to delete E and find another element to move to the position of E.

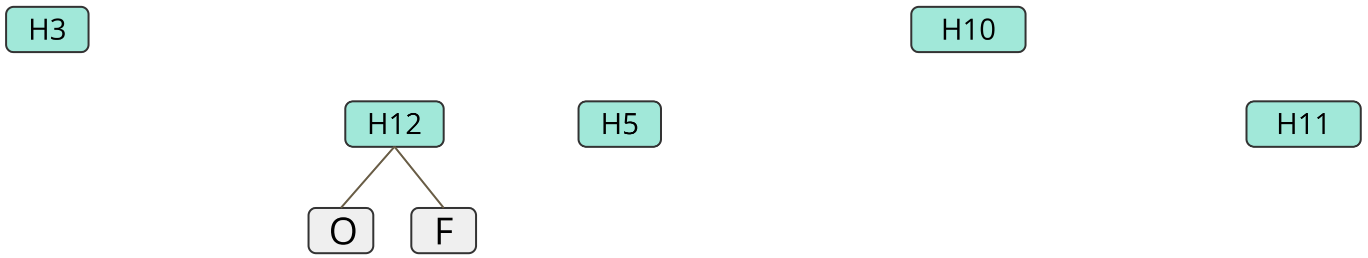
Utreexo - deletion 🌳

We use inclusion proofs to delete E and find another element to move to the position of E.



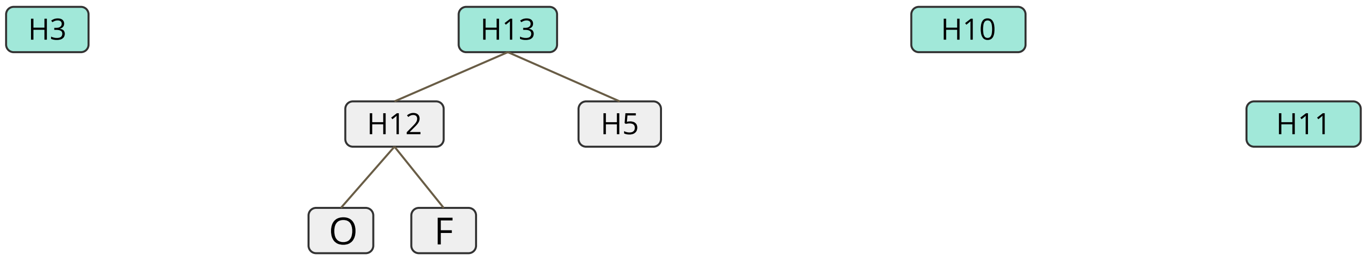
Utreexo - deletion 🌳

We use inclusion proofs to delete E and find another element to move to the position of E.



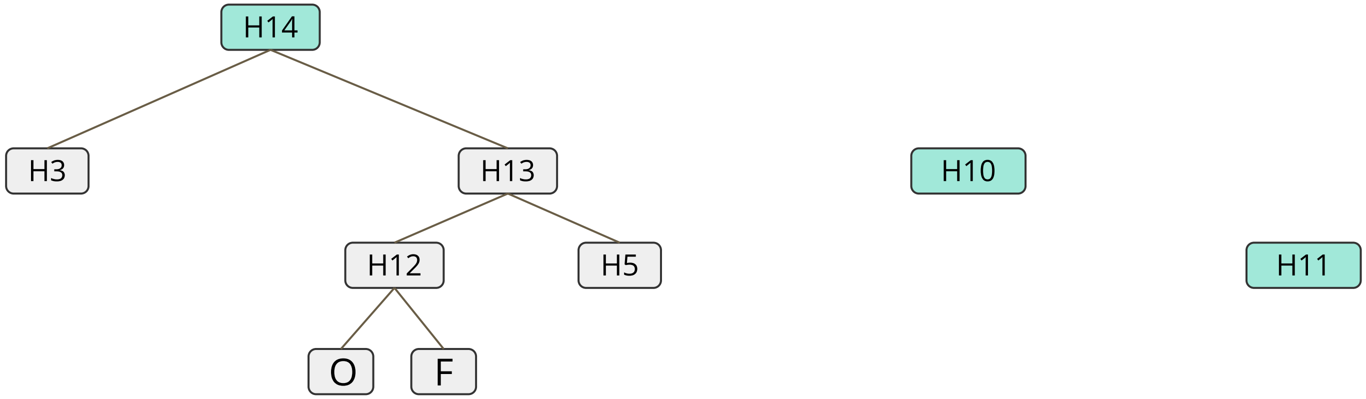
Utreexo - deletion 🌳

We use inclusion proofs to delete E and find another element to move to the position of E.



Utreexo - deletion 🌳

We use inclusion proofs to delete E and find another element to move to the position of E.

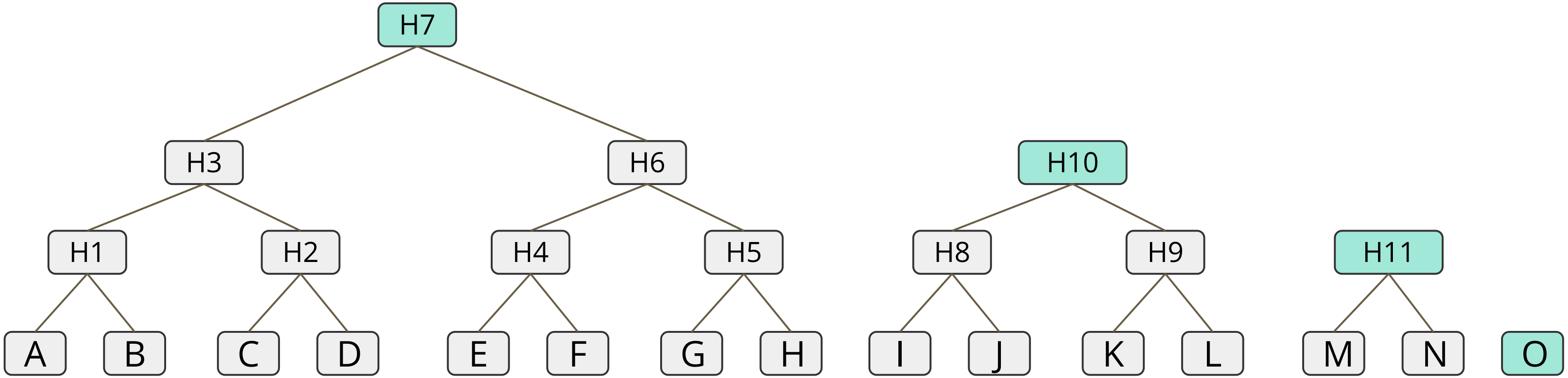


Utreexo - deletion 🌳

Delete doesn't work exactly like this in the implementation. There are several possible ways of implementing it…

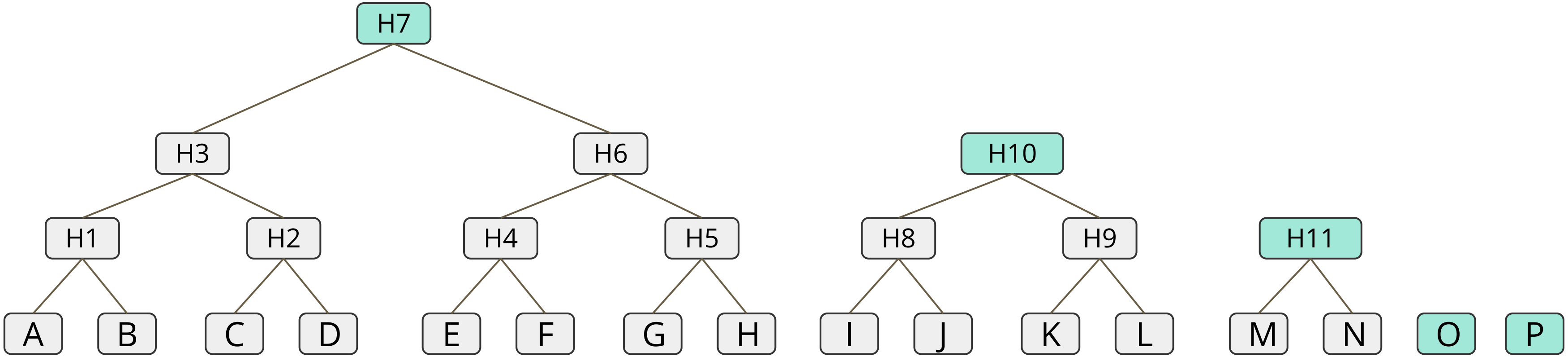
Utreexo - add 🌳

We just add an element to the right and compute the hashes that we can.



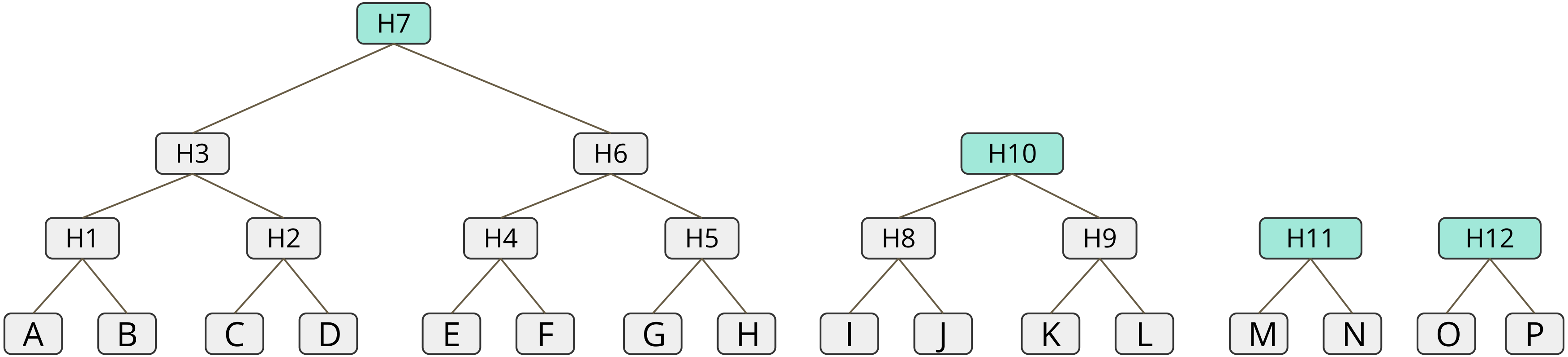
Utreexo - add 🌳

We just add an element to the right and compute the hashes that we can.



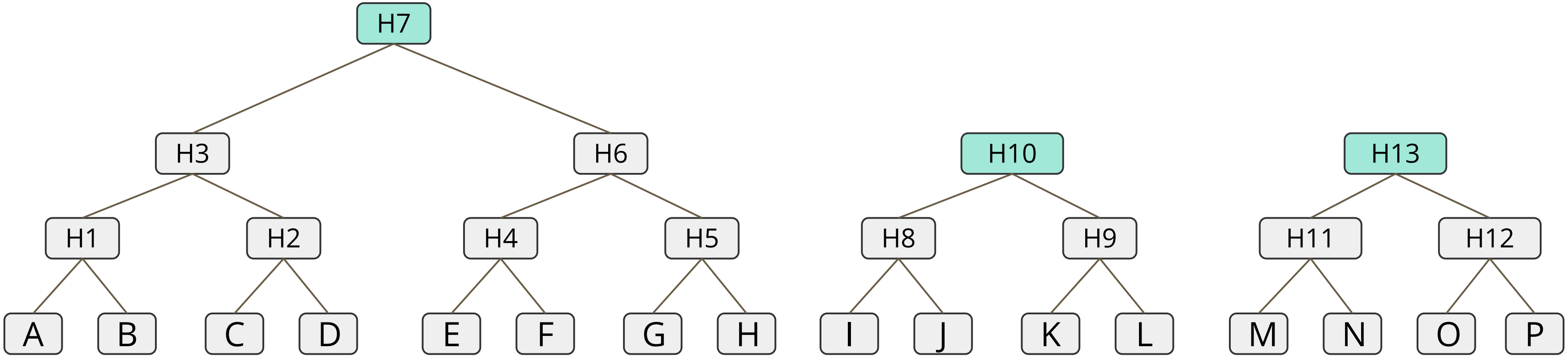
Utreexo - add 🌳

We just add an element to the right and compute the hashes that we can.



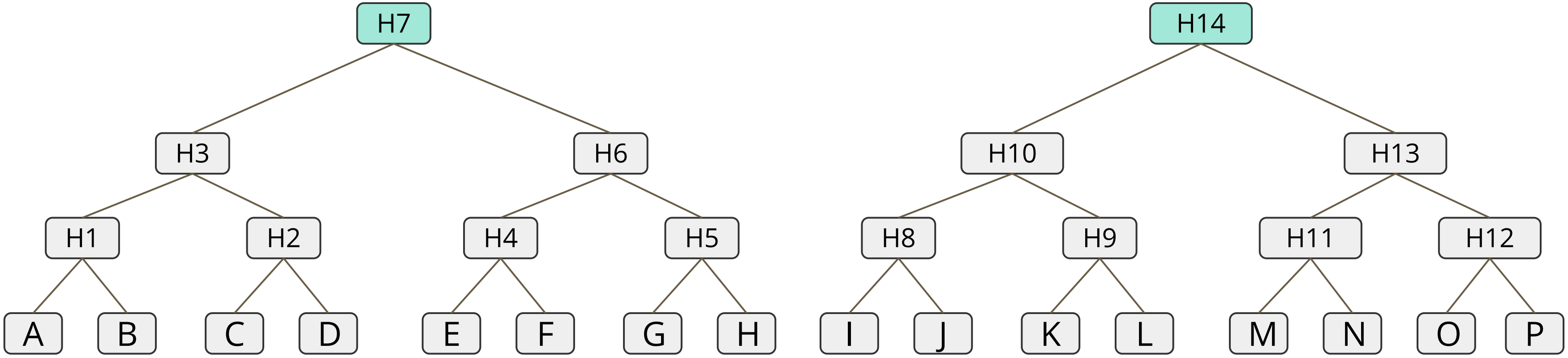
Utreexo - add 🌳

We just add an element to the right and compute the hashes that we can.



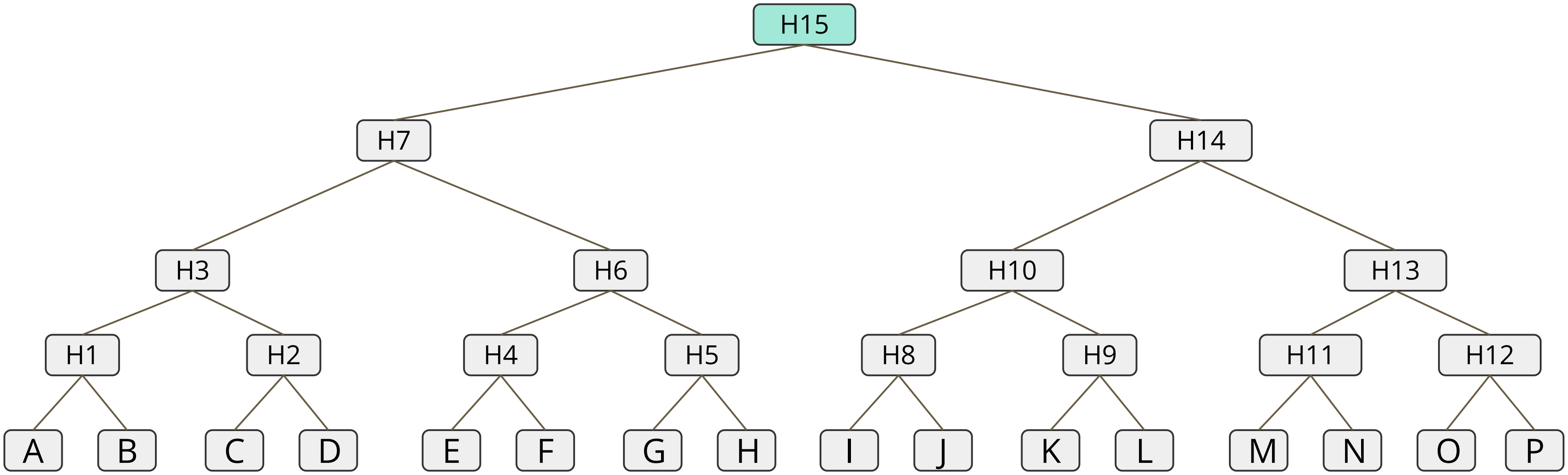
Utreexo - add 🌳

We just add an element to the right and compute the hashes that we can.



Utreexo - add 🌳

We just add an element to the right and compute the hashes that we can.



Utreexo - add 🌳

We just add an element to the right and compute the hashes that we can.

Utreexo - add 🌳

We just add an element to the right and compute the hashes that we can.

Utreexo - add 🌳

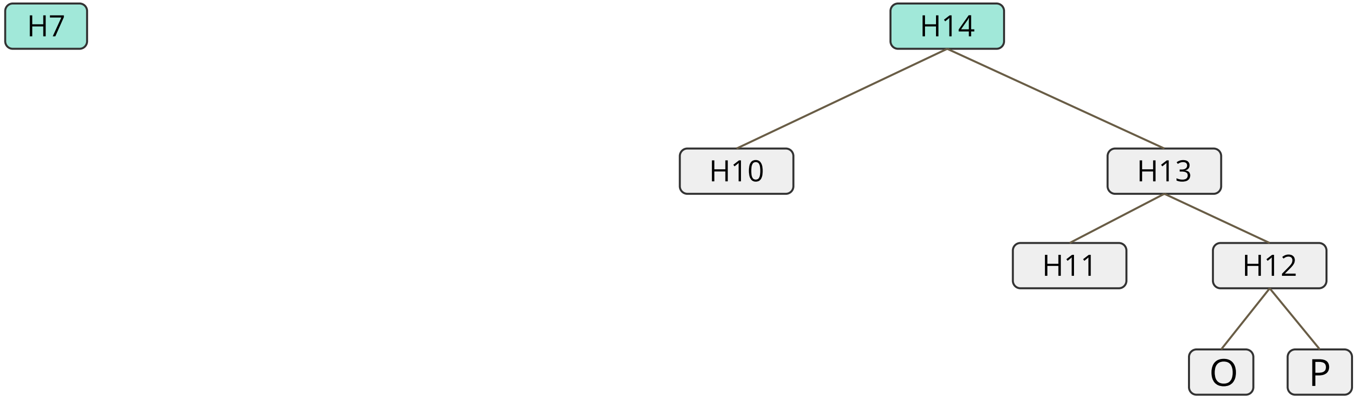
We just add an element to the right and compute the hashes that we can.

Utreexo - add 🌳

We just add an element to the right and compute the hashes that we can.

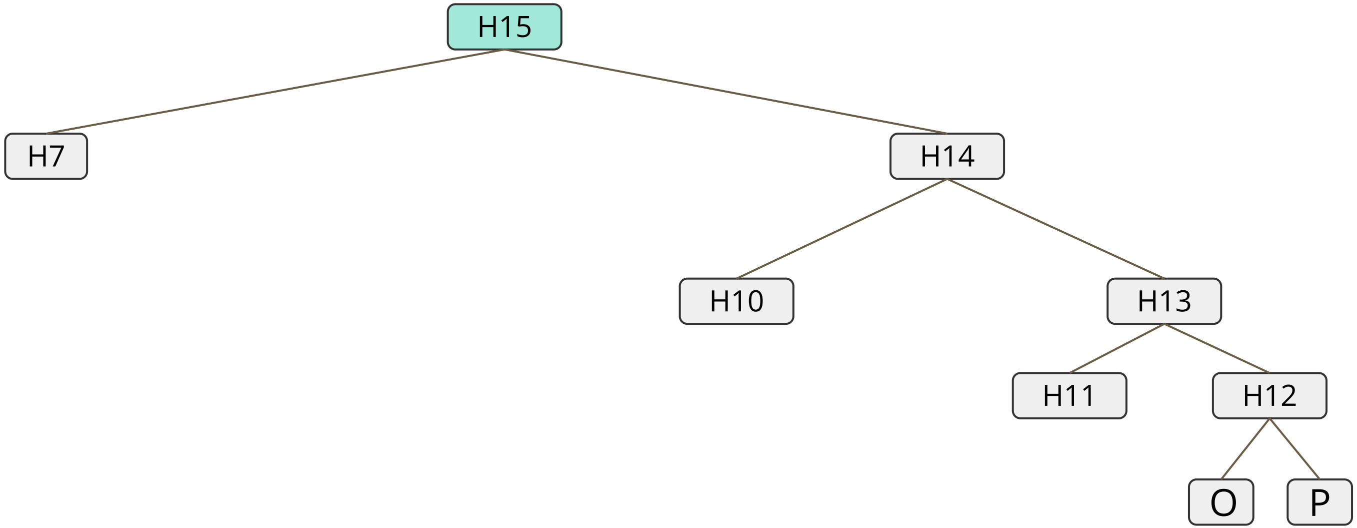
Utreexo - add 🌳

We just add an element to the right and compute the hashes that we can.



Utreexo - add 🌳

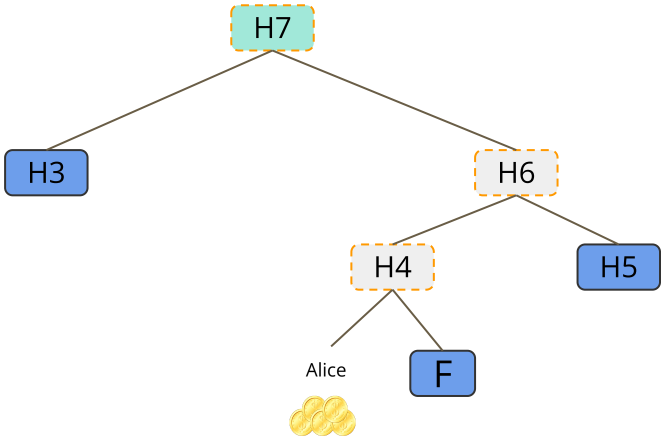
We just add an element to the right and compute the hashes that we can.



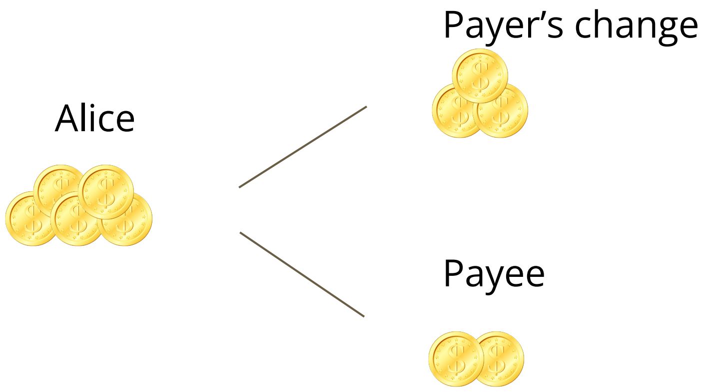
Utreexo transaction

Transaction has additional proofs added to prove the inputs are in the UTXO set.

1) Proofs for transaction inputs

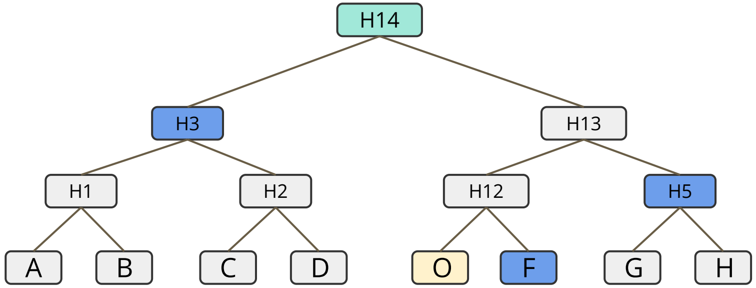


2) Transaction

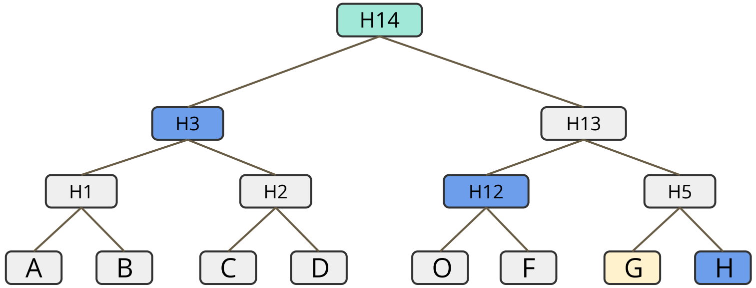


Utreexo optimizations: Combining proofs

Inclusion proof for O

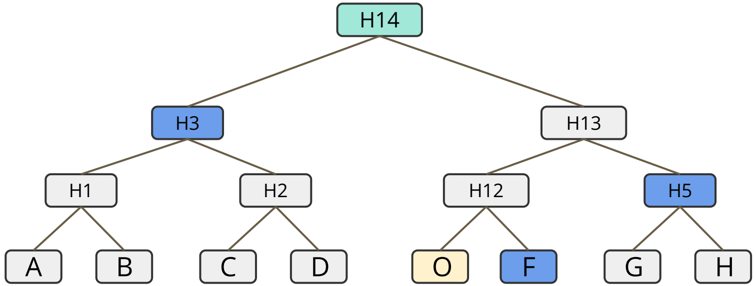


Inclusion proof for G

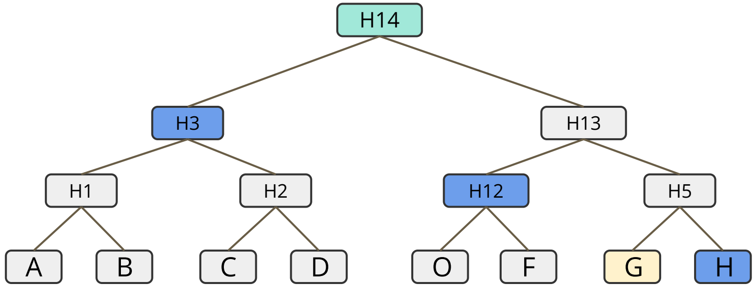


Utreexo optimizations: Combining proofs

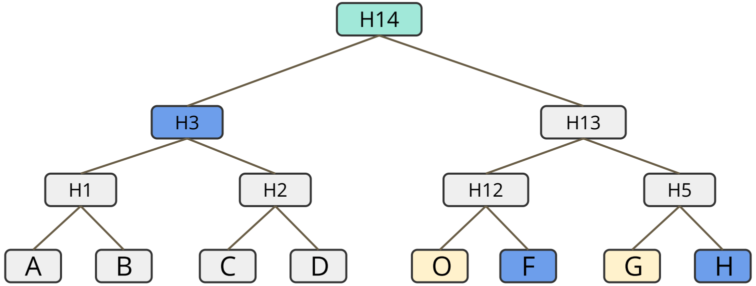
Inclusion proof for O



Inclusion proof for G

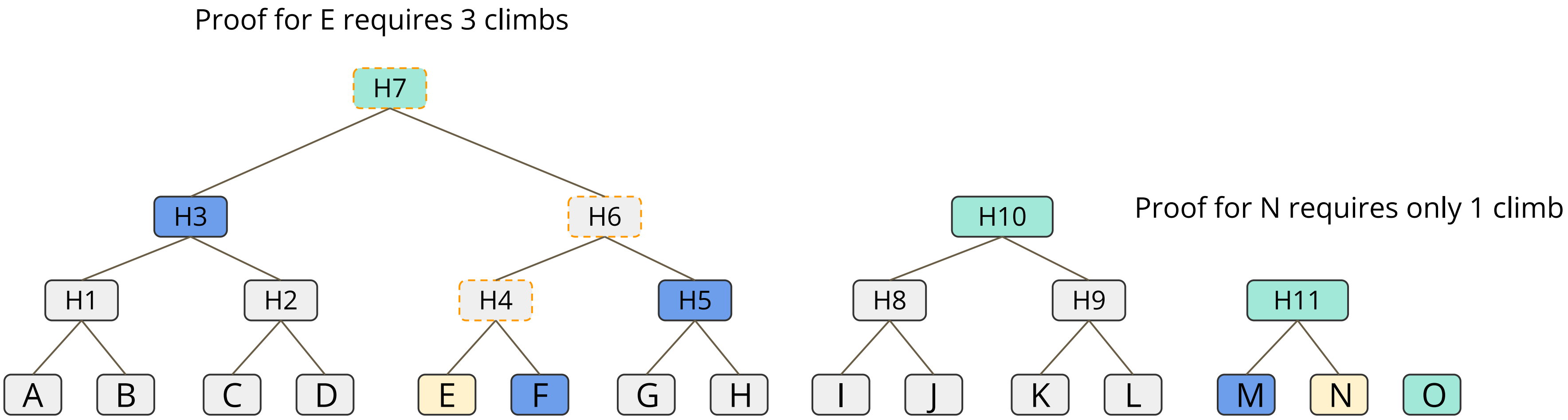


Inclusion proof for O and G

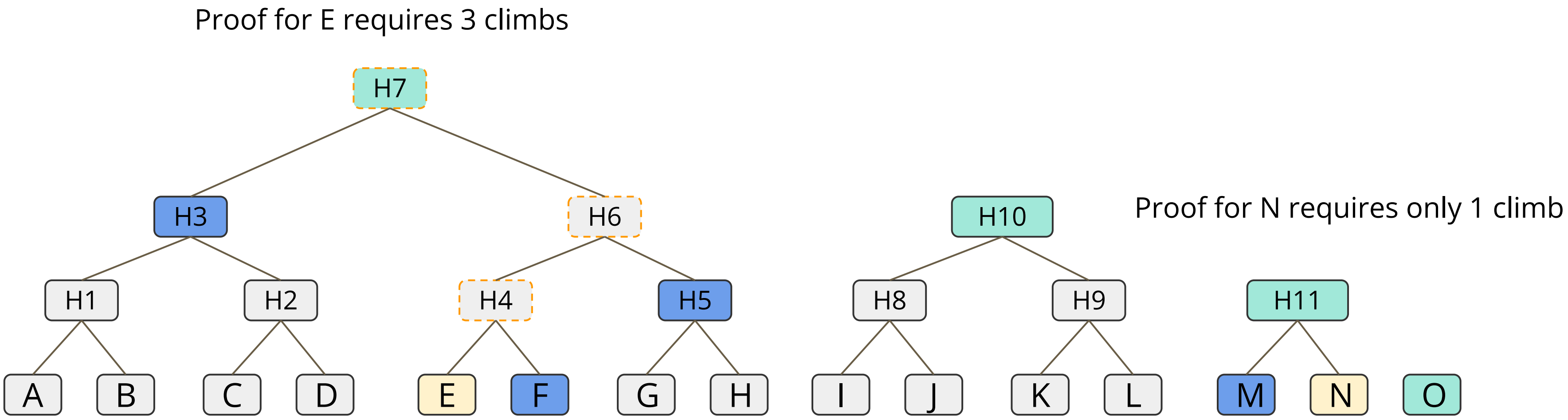


Instead of sending 2*4 pieces, we send only 5 and can still prove both O and G.

Utreexo optimizations: Proof length reduction



Utreexo optimizations: Proof length reduction

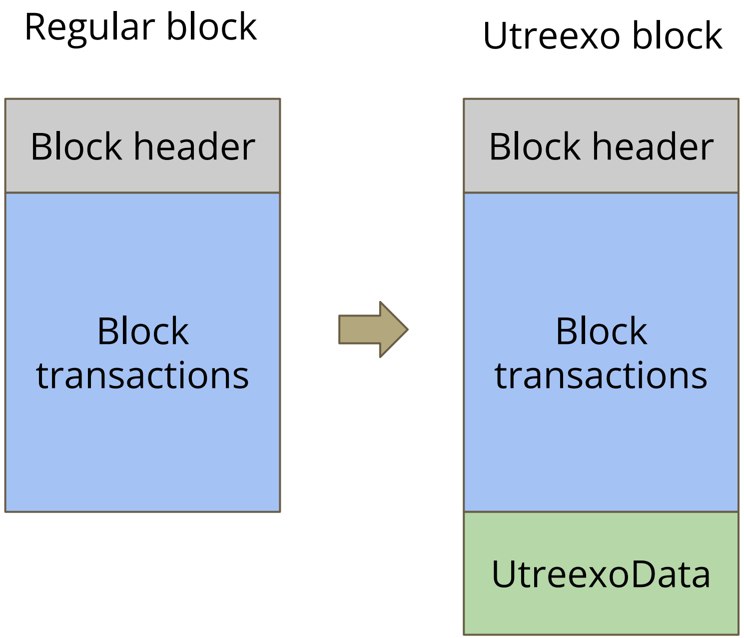


In Bitcoin, the majority of outputs are spent soon after they're created. **If we make sure new outputs are on the right, they'll have shorter proofs!**

Utreexo optimizations: Look-ahead caching

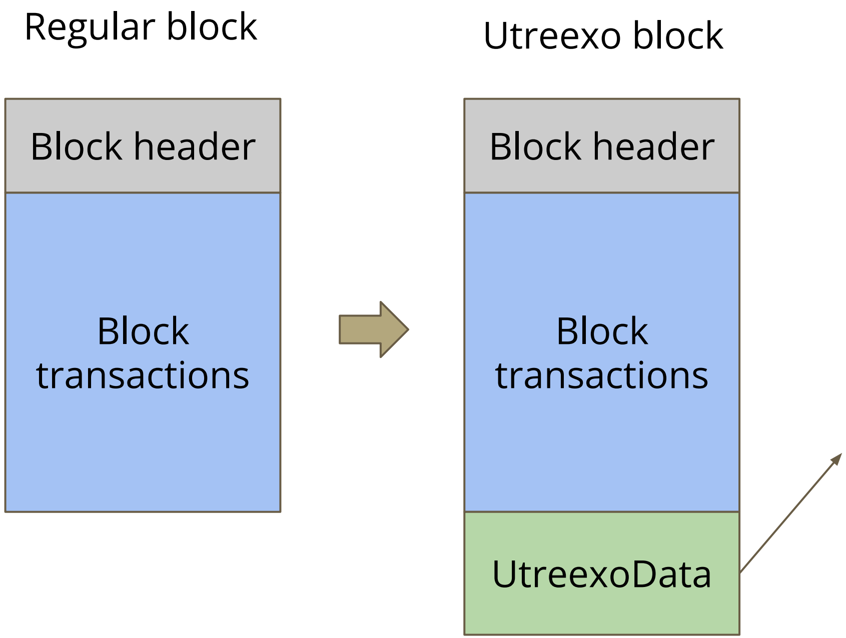
The node knows what happens in the future and can give us hints to save bandwidth and computation.

Utreexo block



~30% more bandwidth

Utreexo block



~30% more bandwidth

What Utreexo data looks like in btcd implementation

```go
// UData contains data needed to prove the existence and validity of all inputs
// for a Bitcoin block.  With this data, a full node may only keep the utreexo
// roots and still be able to fully validate a block.
type UData struct {
        // AccProof is the utreexo accumulator proof for all the inputs.
        AccProof utreexo.Proof

        // LeafDatas are the tx validation data for every input.
        LeafDatas []LeafData

        // All the indexes of new utxos to remember.
        RememberIdx []uint32
}
```

Status

Focus:

- Btcd fork implementation (no work on Core at the moment)
- Very likely to get into some wallets (TBD)

Core things = 3 BIPS + 1 BIP Fix

- BIP for the accumulator
- BIP for how block/tx verification works with the accumulator
- BIP for P2P messaging with utreexo proofs
- BIP that fixes BIP30

Links:
Utreexo full node - https://github.com/utreexo/utreexod
Utreexo - https://github.com/utreexo/utreexo

Questions?