Blockchains & Distributed Ledgers

Lecture 02

Dimitris Karakostas

Slide credits: DK, Aggelos Kiayias, Dionysis Zindros, Christos Nasikas

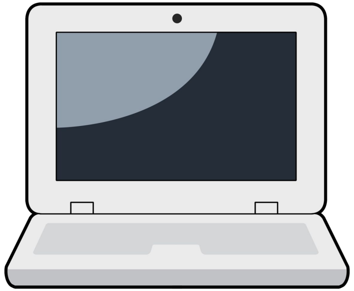
Overview

- Motivation: Server file storage
- Merkle trees to store lists
- Proofs-of-inclusion
- Merkle trees to store sets
- Proofs-of-non-inclusion
- Merkle–Patricia tries to store key:value pairs
- Blocks and blockchains

Authenticated Data Structures

- Like regular data structures, but cryptographically authenticated
- A **verifier** can store, retrieve, operate on data held by an **untrusted prover**

The authenticated file storage problem



Store file F with content D

Retrieve file F

Here is the content D

Dropbox

verifier

prover

The file storage problem

- Client wants to store a file on a server
- File has a name F and content D
- Clients wants to retrieve file F later

File storage: Basic protocol

- Client sends file F with content D to server
- Server stores (F, D)
- Client deletes D
- Client requests F from server
- Server returns D
- Client has recovered D

File storage: Basic protocol

- Client sends file F with content D to server
- Server stores (F, D)
- Client deletes D
- Client requests F from server
- Server returns D
- Client has recovered D

*What if **server is adversarial** and returns D’ != D?*

File storage: Protocol against adversaries

Trivial solution:

- Client does not delete D
- When server returns D’, client compares D and D’

...what if client doesn’t have enough memory to store D for a long time?

File storage: Hash-based protocol

- Client sends file F with data D to server
- Server stores (F, D)
- Client stores H(D), deletes D
- Client requests F from server
- Server returns D’
- Client compares H(D’) = H(D)

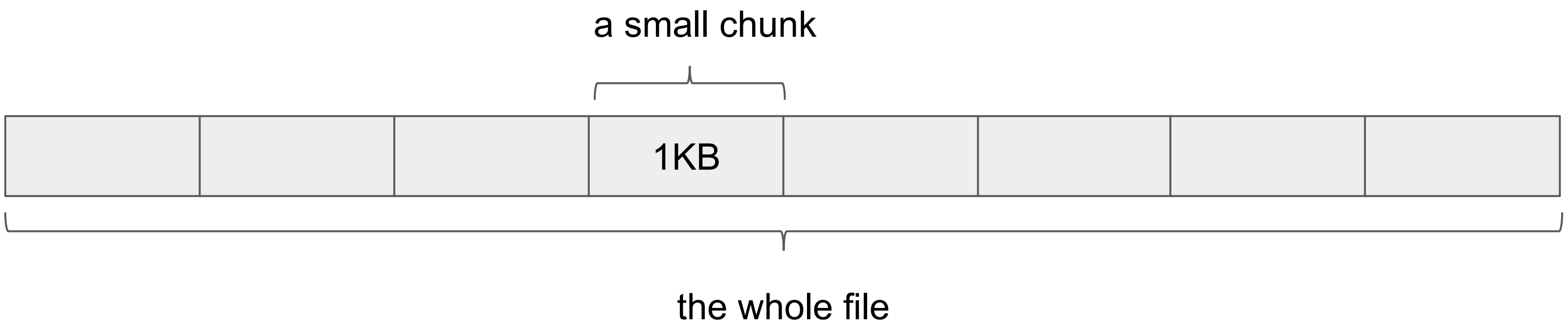
File storage: File chunks

- What if client wants to retrieve the 200,019th byte of the file?
- Must download the whole file…
- Merkle trees to the rescue!

Merkle Trees

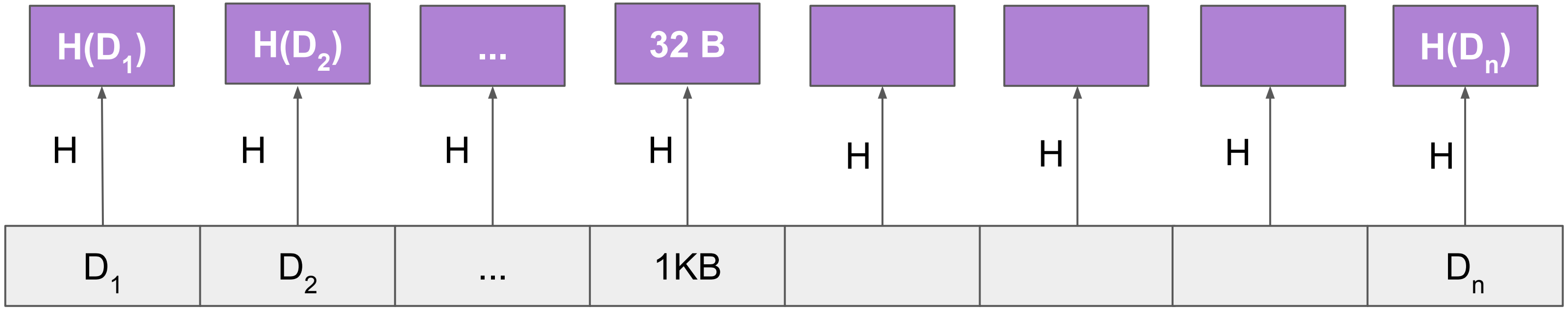
Merkle Tree

- An **authenticated** binary tree
- Split file into **chunks** of, say, 1KB

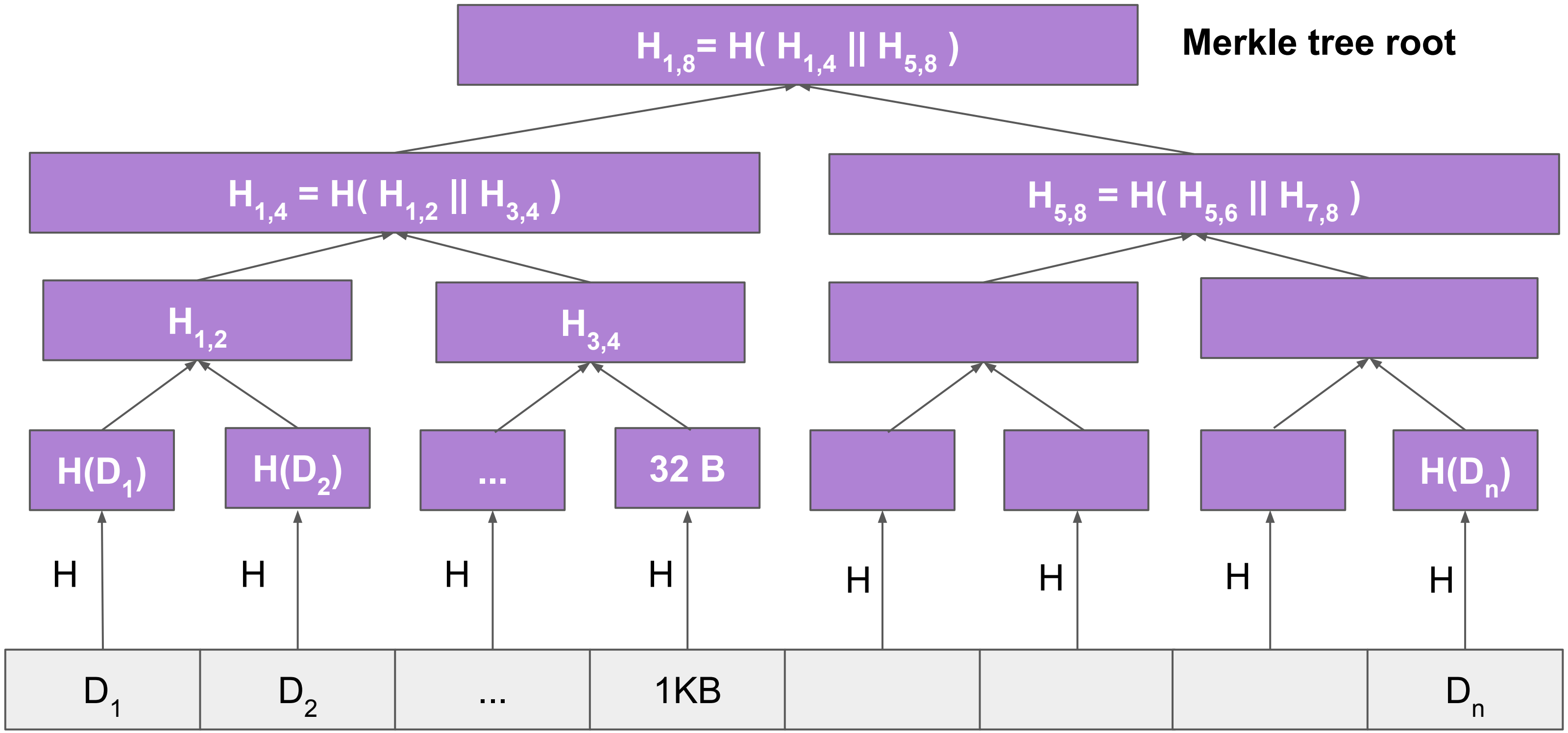


Merkle Tree

- **Hash** each chunk using a cryptographic hash function (SHA256)
- Convention: Arrows show direction of hash function application



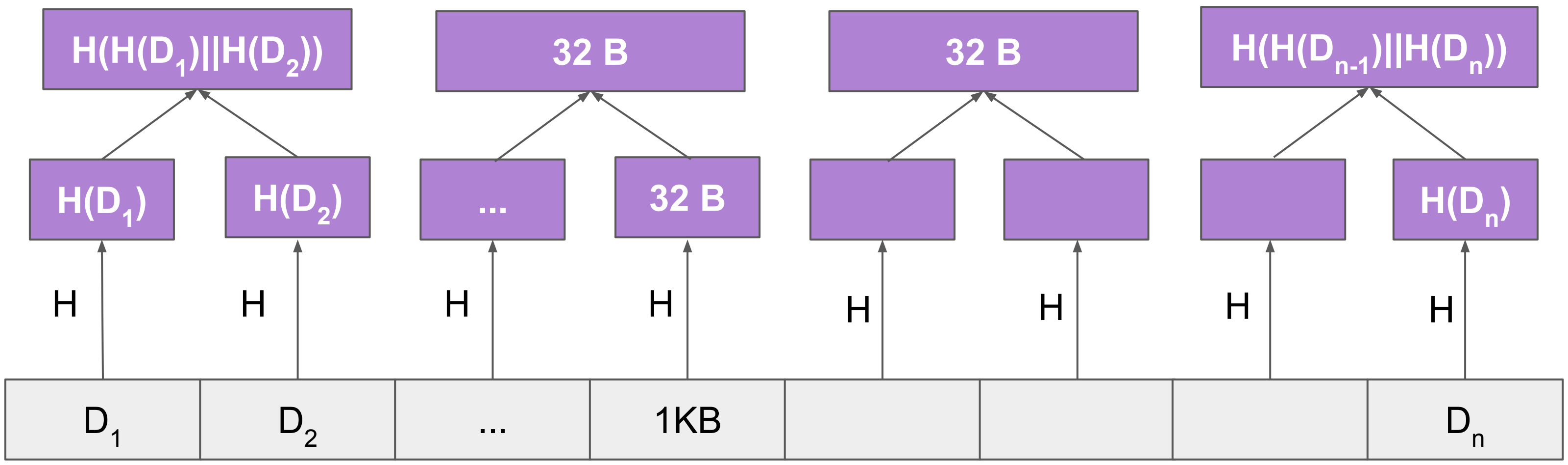
Merkle Tree



H1,8= H(H1,4 || H5,8)
Merkle tree root
H1,4 = H(H1,2 || H3,4)
H5,8 = H(H5,6 || H7,8)
H1,2
H3,4
H(D1)
H(D2)
...
32 B
H(Dn)
H
H
H
H
H
H
H
H
D1
D2
...
1KB
Dn

Merkle Tree

- **Combine** them by two to create a (complete binary) tree
- Each node stores the **hash** of the **concat** of its children

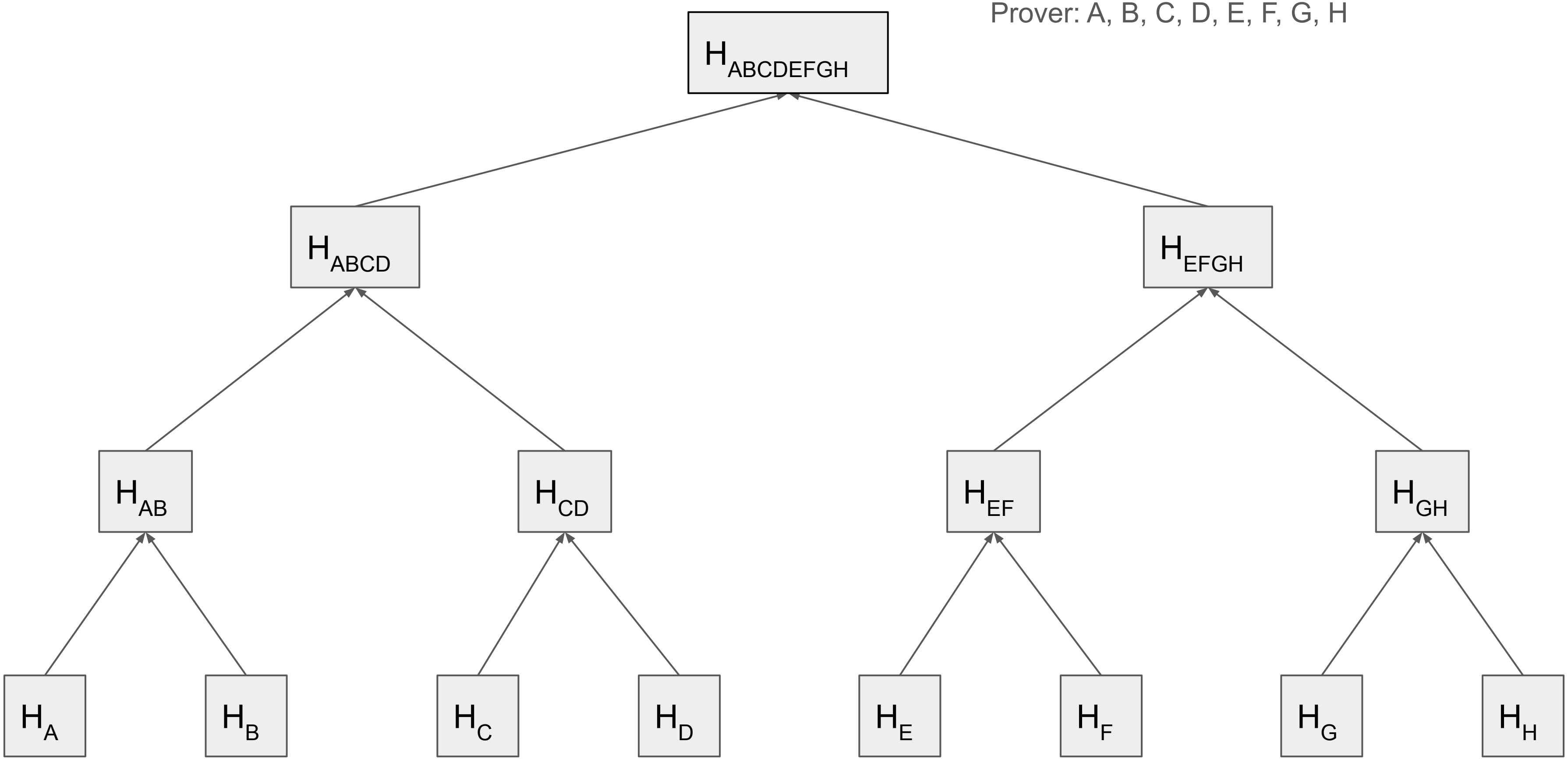


Proofs-of-inclusion

- Client creates Merkle Tree root **MTR** from initial file data D
- Client sends file data D to server
- Client deletes data D, but stores MTR (32 bytes)
- Client requests chunk x from server
- Server returns chunk x and short proof-of-inclusion π
- Client checks that chunk x is included in MTR using proof π

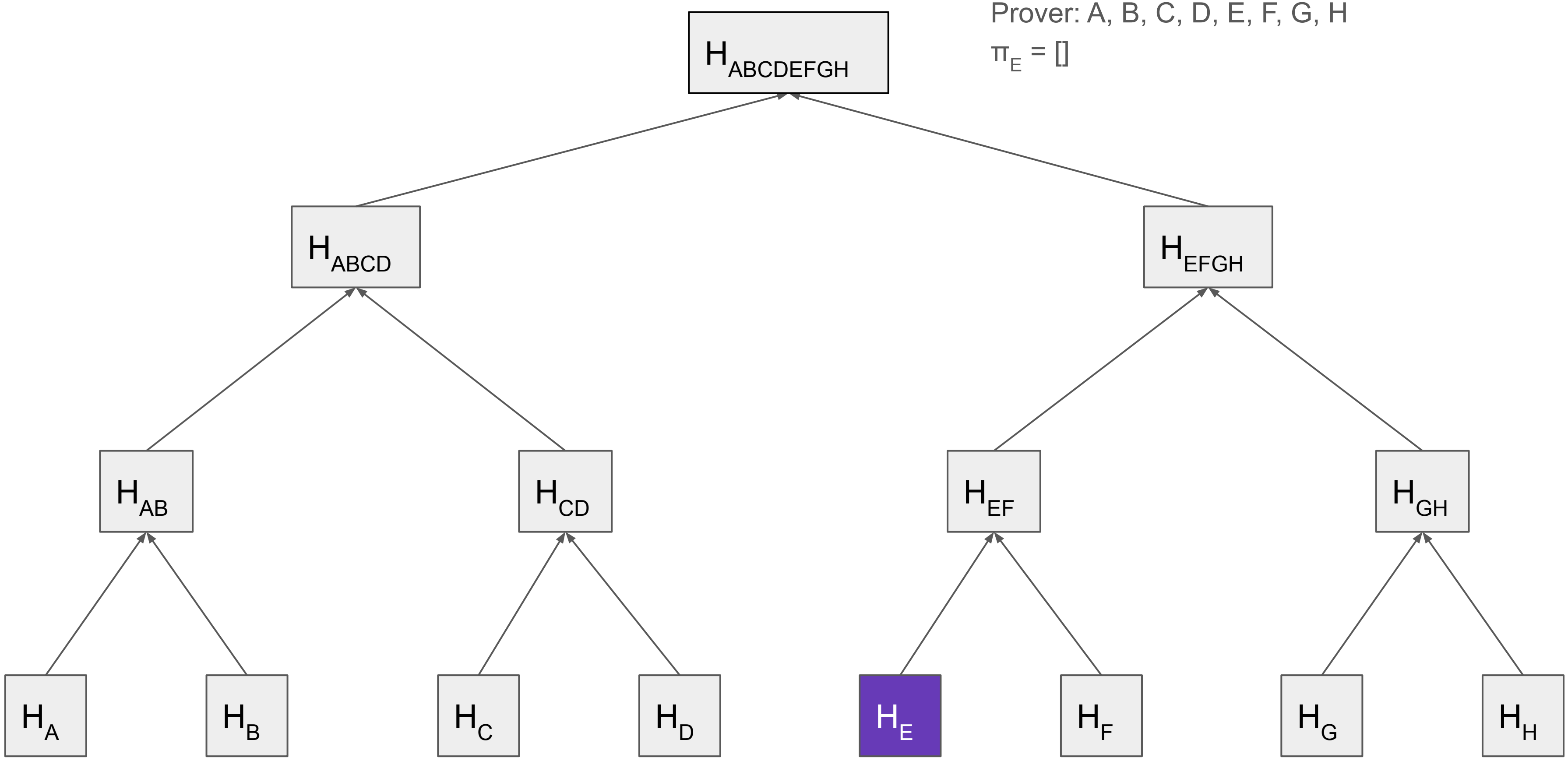
Merkle tree: proof of inclusion

Verifier: $H_{ABCDEFGH}$
Prover: A, B, C, D, E, F, G, H



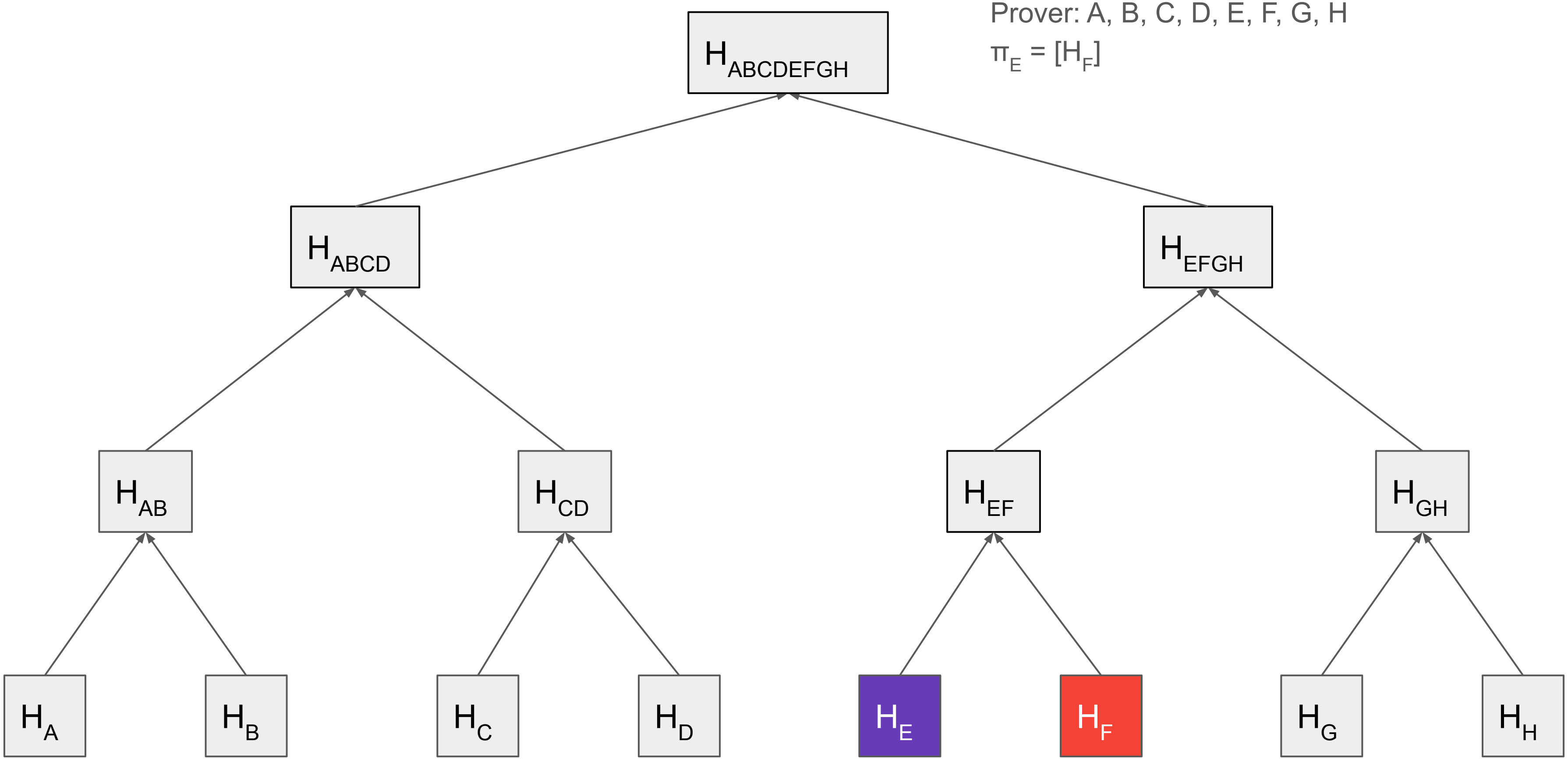
Merkle tree: proof of inclusion

Verifier: $H_{ABCDEFGH}$, E
Prover: A, B, C, D, E, F, G, H
π_E = []



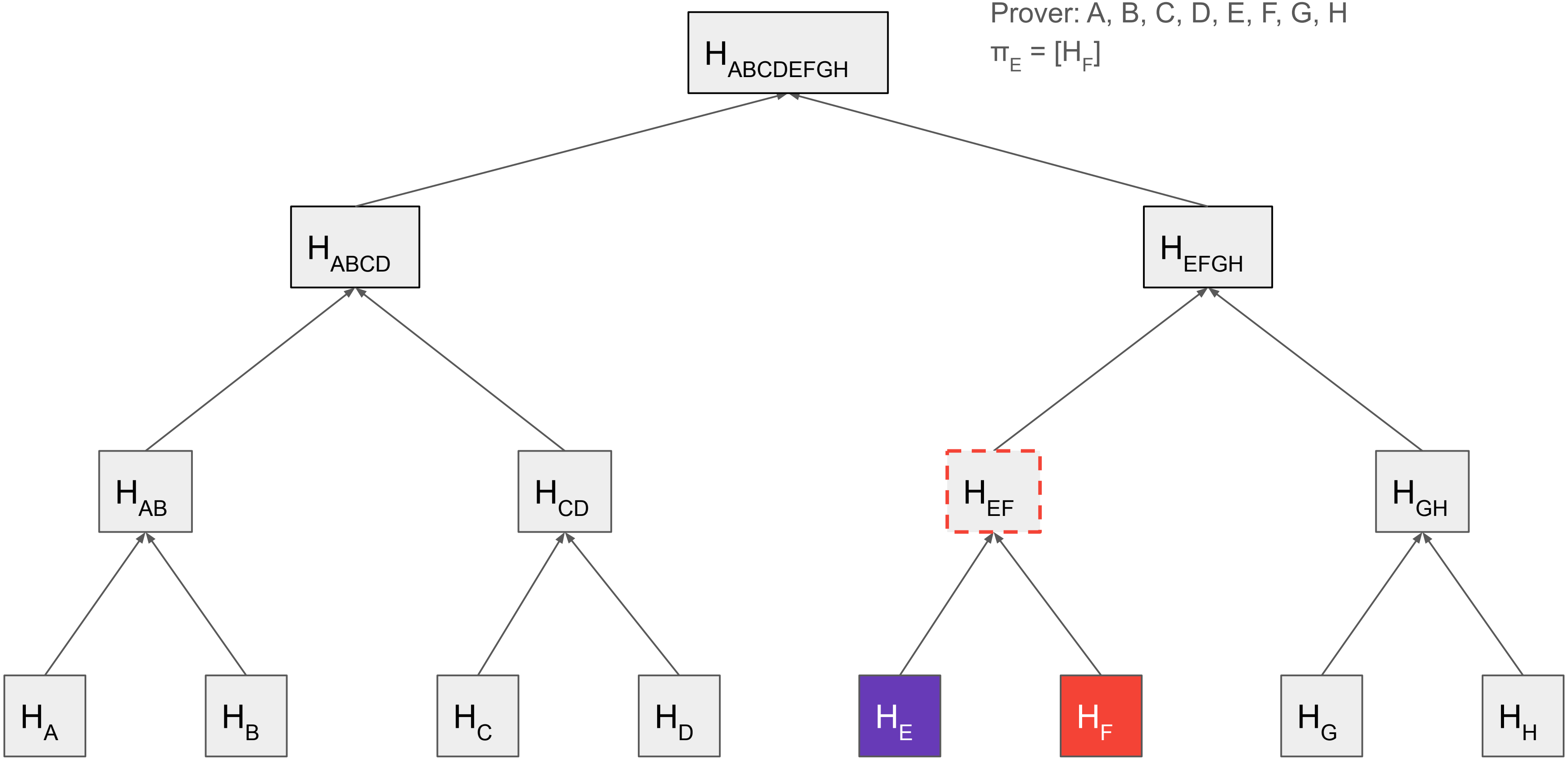
Merkle tree: proof of inclusion

Verifier: $H_{ABCDEFGH}$, E
Prover: A, B, C, D, E, F, G, H
$\pi_E = [H_F]$



Merkle tree: proof of inclusion

Verifier: $H_{ABCDEFGH}$, E
Prover: A, B, C, D, E, F, G, H
$\pi_E = [H_F]$

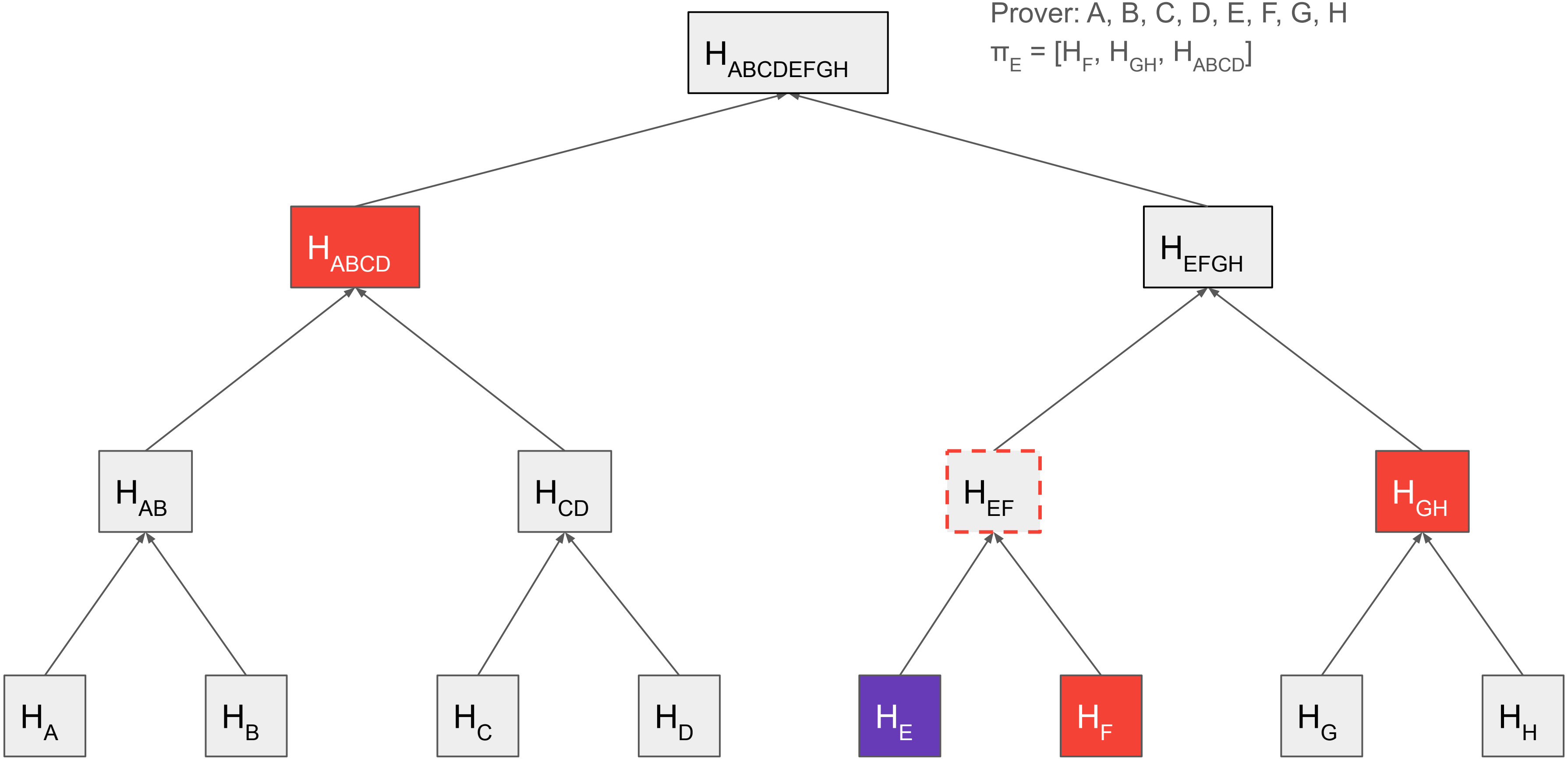


Merkle tree: proof of inclusion

Verifier: $H_{ABCDEFGH}$, E

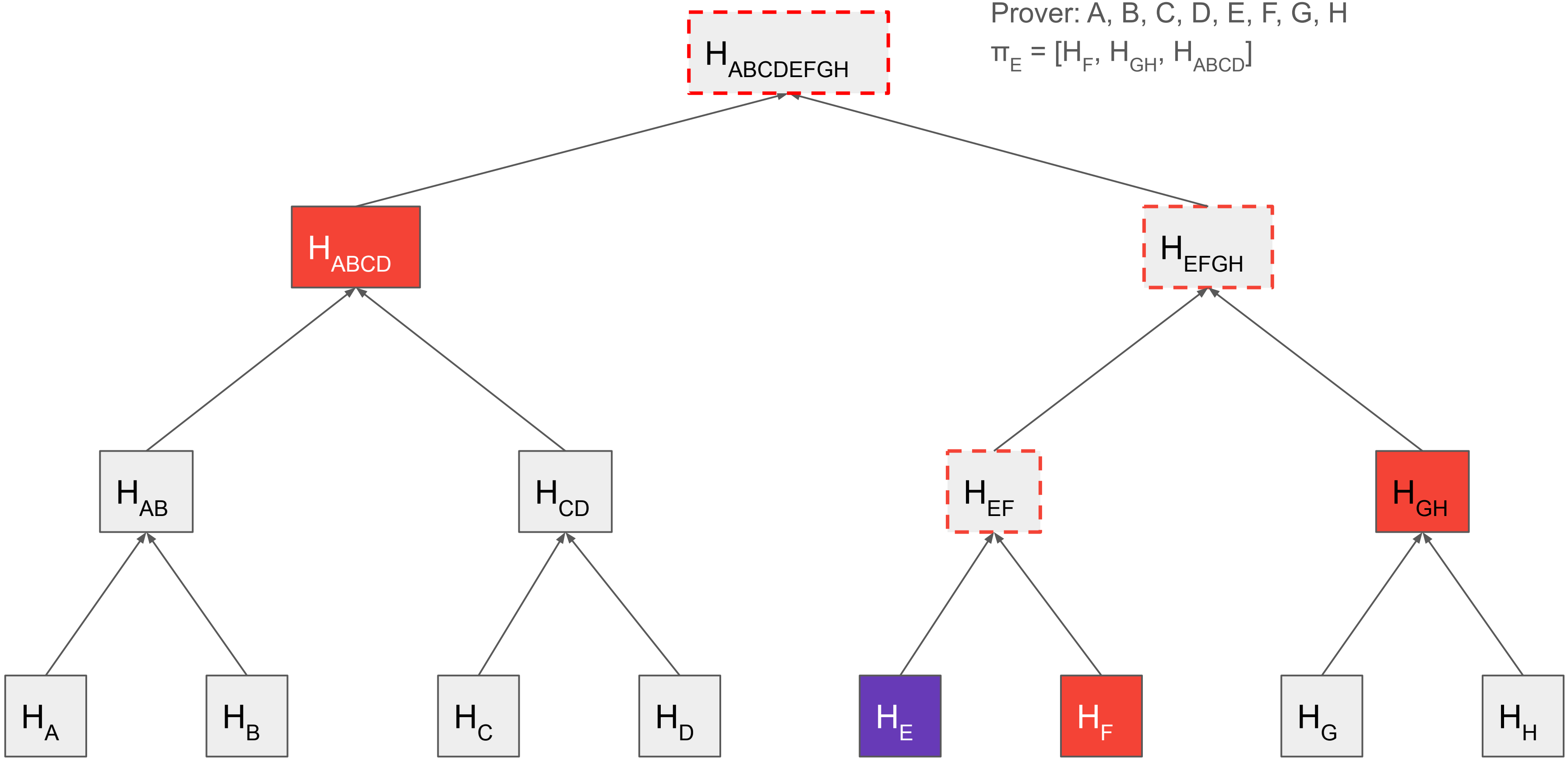
Prover: A, B, C, D, E, F, G, H

$\pi_E = [H_F, H_{GH}, H_{ABCD}]$



Merkle tree: proof of inclusion

Verifier: $H_{ABCDEFGH}$, E
Prover: A, B, C, D, E, F, G, H
$\pi_E = [H_F, H_{GH}, H_{ABCD}]$



Merkle Tree proof-of-inclusion

- Prover sends chunk
- Prover sends **siblings** along path connecting leaf to MTR
- Verifier computes hashes along the path connecting leaf to MTR
- Verifier checks that computed root = MTR
- How big is proof-of-inclusion?

Merkle Tree proof-of-inclusion

- Prover sends chunk
- Prover sends **siblings** along path connecting leaf to MTR
- Verifier computes hashes along the path connecting leaf to MTR
- Verifier checks that computed root = MTR
- How big is proof-of-inclusion?

$$|\pi| \in \Theta(\log_2|D|)$$

Merkle tree applications

- BitTorrent uses Merkle Tree to exchange files
- Bitcoin uses Merkle trees to store transactions
- Ethereum uses Merkle–Patricia tries for storage and transactions

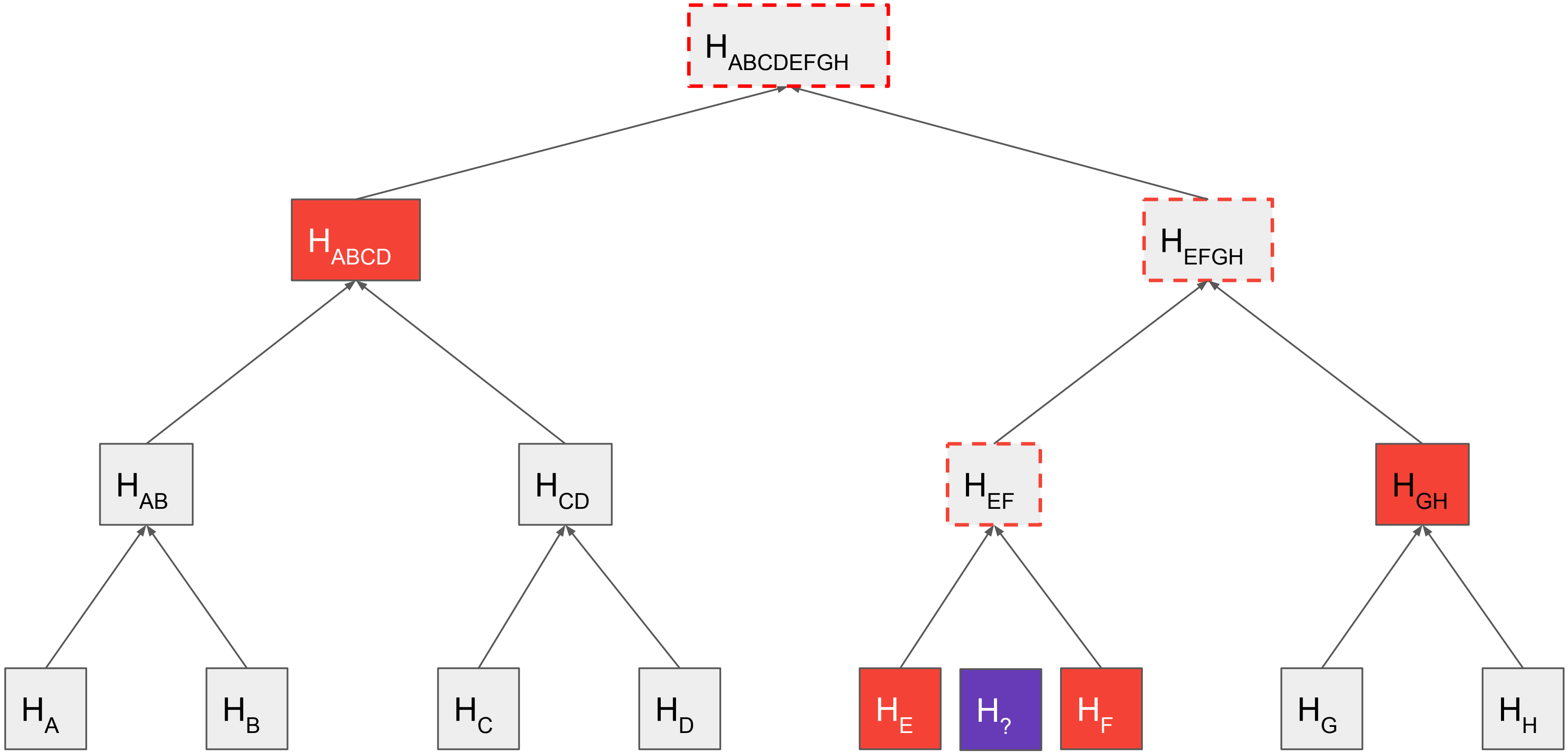
Storing *sets* instead of lists

- Merkle Trees can be used to store sets of keys instead of lists
- Verifier asks prover to store a set of keys
- Verifier deletes set
- Verifier later asks prover if key belongs to set
- Prover provides proof-of-inclusion or proof-of-non-inclusion
- Prover can be adversarial

Merkle trees for set storage

- Verifier sorts set elements
- Creates MTR on sorted set
- Proof-of-inclusion as before
- Proof-of-non-inclusion for x
  - Show proof-of-inclusion for previous $H_<$ and next $H_>$ element in set
  - Verifier checks that $H_<$, $H_>$ proofs-of-inclusion are correct
  - Verifier checks that $H_<$, $H_>$ are adjacent in tree
  - Verifier checks that $H_< < x$ and $H_> > x$
  - The two proofs-of-inclusion can be compressed into one

Merkle tree: proof of inclusion / non-inclusion



Tries

Tries

- Also called radix or prefix tree
- Search tree: ordered data structure
- Used to store a set or an associative array (key/value store)
- Keys are usually strings

Tries

- Supports two operations: **add** and **query**
- **add** adds a string to the set
- **query** checks if a string is in the set (true/false)
- **Initialize**: Start with empty root

Tries: add(string)

- Start at root
- Split string into characters
- For every character, follow an edge labelled by that character
- If edge does not exist, create it
- Mark the node you arrive at

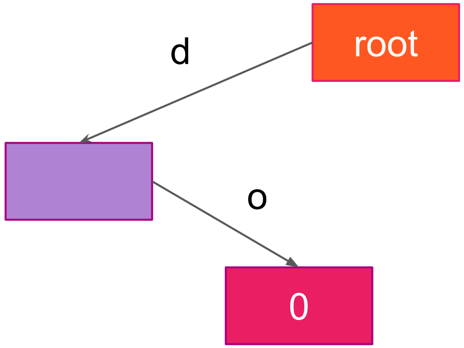
Tries: query(string)

- Start at root
- Split string into characters
- For every character, follow an edge labelled by that character
- If edge does not exist, return false
- When you arrive at a node and your string is consumed, check if node is marked
  - If it is marked, return **yes** (and marked value)
  - Otherwise, return **no**

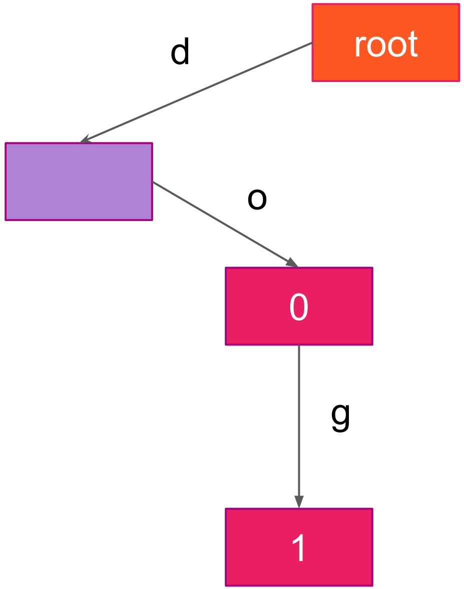
{ **do**: 0, **dog**: 1, **dax**: 2, **doge**: 3, **dodo**: 4, **house**: 5, **houses**: 6 }

{ **do**: 0, **dog**: 1, **dax**: 2, **doge**: 3, **dodo**: 4, **house**: 5, **houses**: 6 }

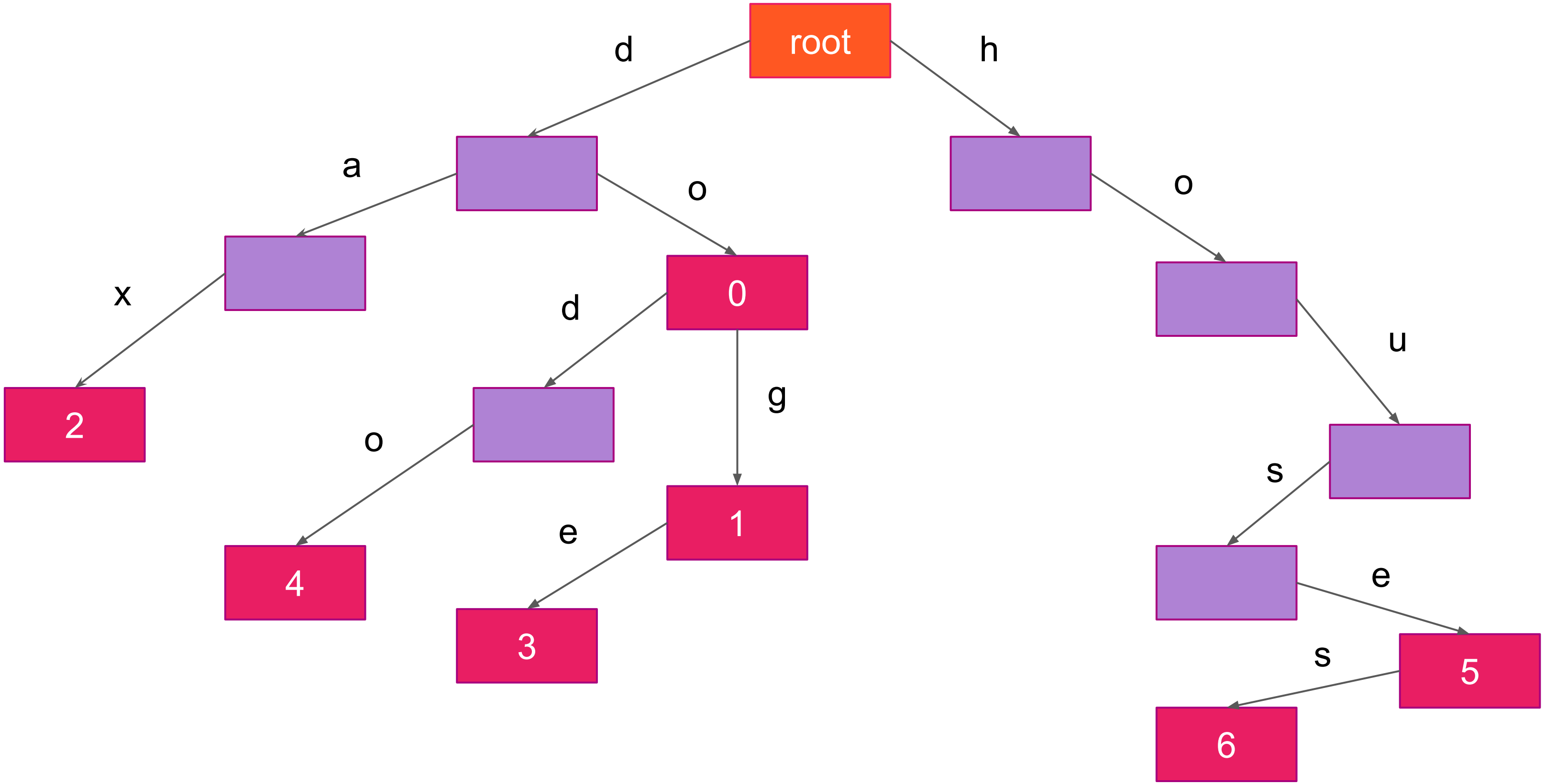
{ **do**: 0, **dog**: 1, **dax**: 2, **doge**: 3, **dodo**: 4, **house**: 5, **houses**: 6 }



{ **do**: 0, **dog**: 1, **dax**: 2, **doge**: 3, **dodo**: 4, **house**: 5, **houses**: 6 }



{ **do**: 0, **dog**: 1, **dax**: 2, **doge**: 3, **dodo**: 4, **house**: 5, **houses**: 6 }



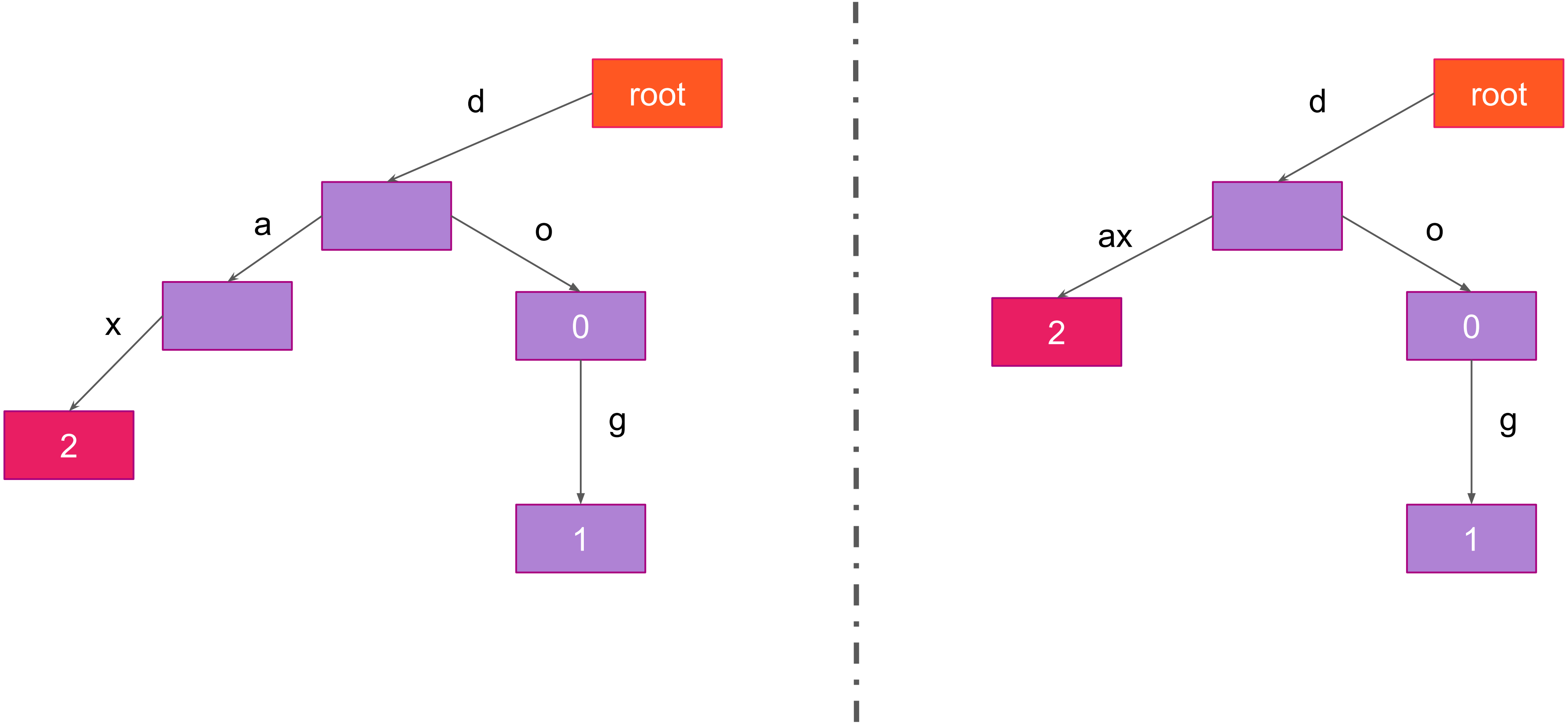
Particia (or radix) tree

- Space-optimized trie
- An isolated path, with *unmarked* nodes which are *only children*, is merged into single edge
- The label of the merged edge is the concatenation of the labels of merged nodes

Tries / Patricia tries as key/value store

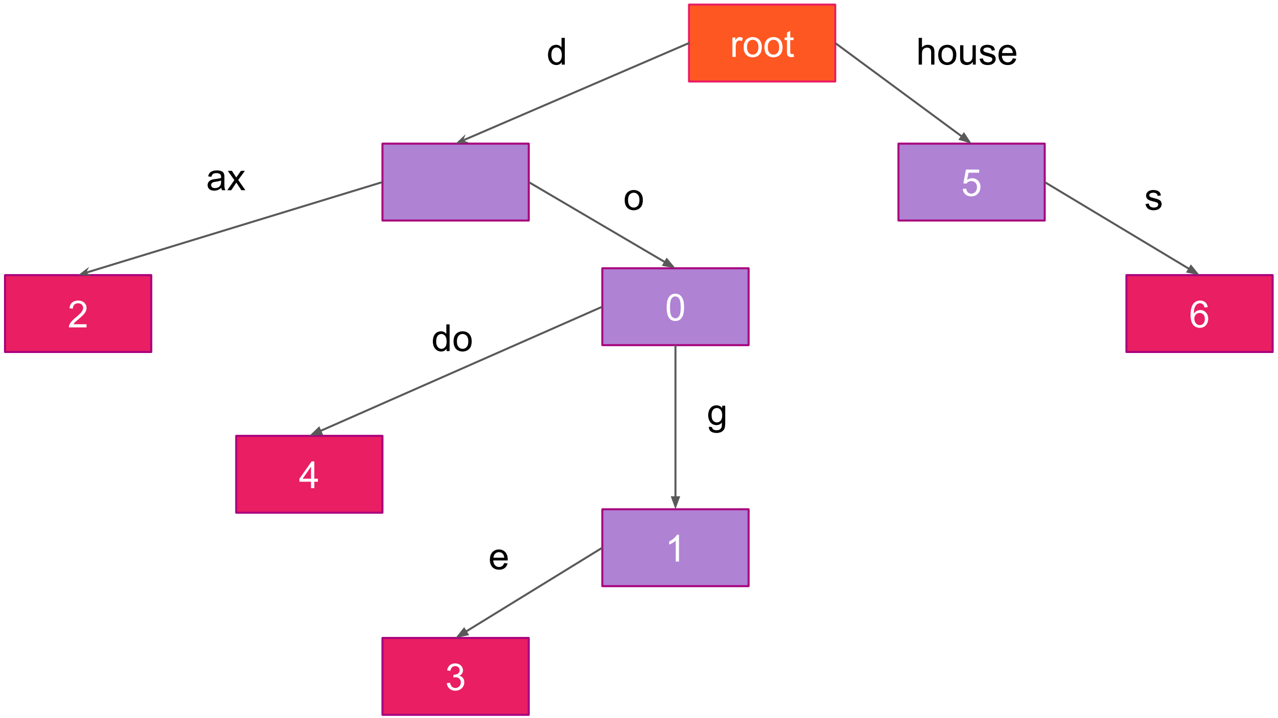
- Marking can contain arbitrary value
- This allows us to map keys to values
- **add(key, value)**
- **query(key) → value**

Trie vs. Patricia trie



Patricia trie

{ **do**: 0, **dog**: 1, **dax**: 2, **doge**: 3, **dodo**: 4, **house**: 5, **houses**: 6 }

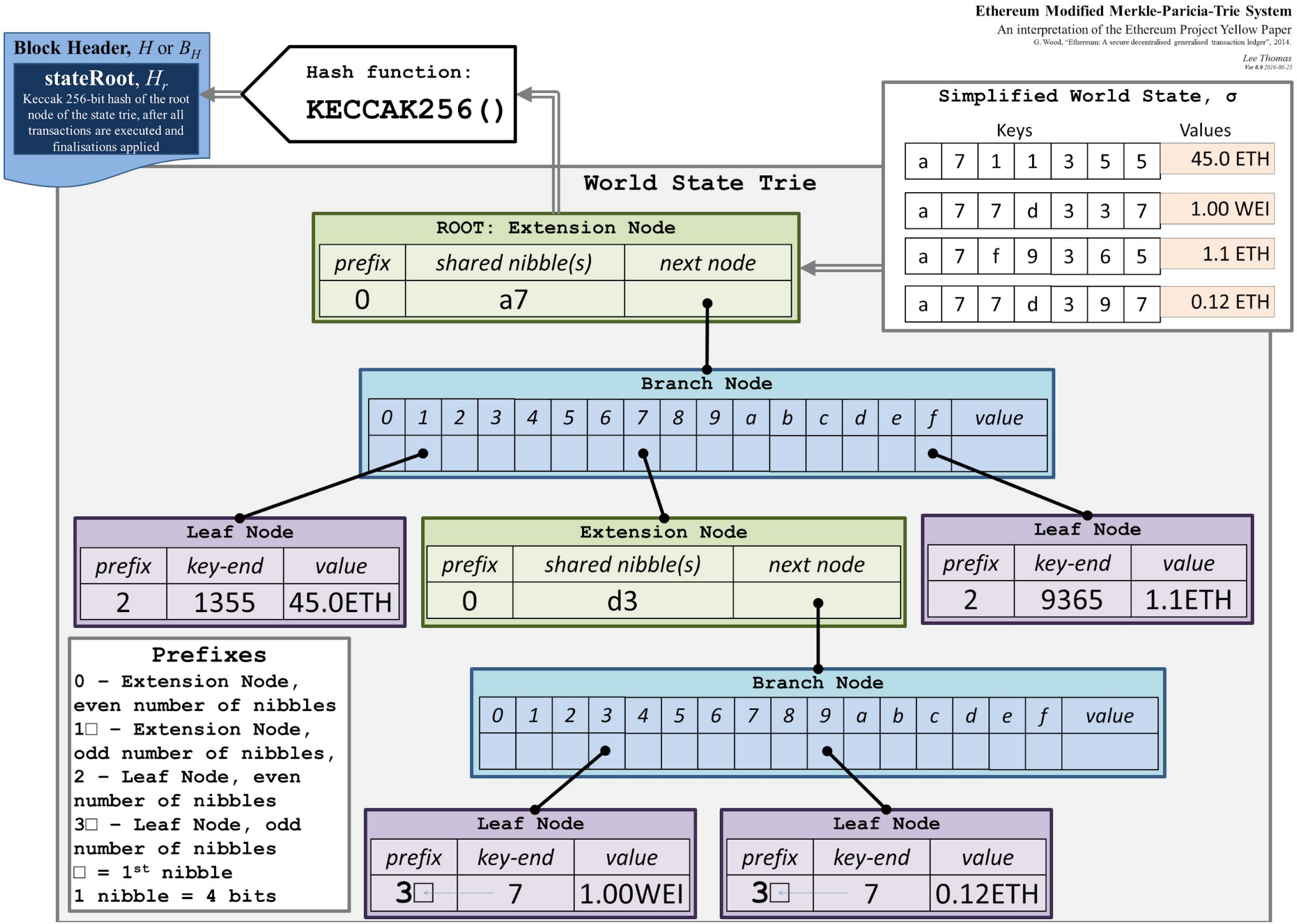


Merkle Patricia trie

- Authenticated Patricia trie
- First implemented in Ethereum
- Allows proof of inclusion (of key, with particular value)
- Allows proof of non-inclusion (by showing key does not exist in trie)

Merkle Patricia trie

- Split nodes into three types:
  - **Leaf**: Stores edge string leading to it, and **value**
  - **Extension**: Stores **string** of a single edge, **pointer** to next node, and **value** if node marked
  - **Branch**: Stores one pointer to another node per alphabet symbol, and **value** if node marked
- We encode keys as hex, so alphabet size is 16
- We encode all child edges in every node with some encoding (e.g. JSON)
- Pointers are by hash application
- Arguments for correctness and security are same as for Merkle Trees

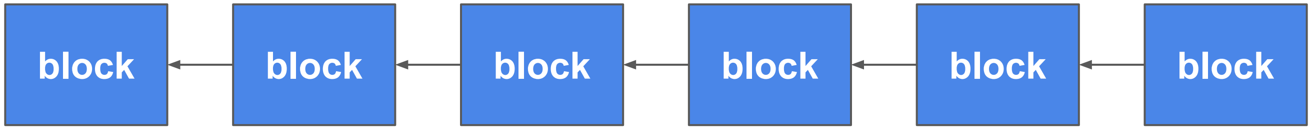


Ethereum Modified Merkle-Paricia-Trie System
An interpretation of the Ethereum Project Yellow Paper
G. Wood, "Ethereum: A secure decentralised generalised transaction ledger", 2014.
Lee Thomas
Ver 0.0 2016-06-23
Block Header, H or B_H
stateRoot, H_r
Keccak 256-bit hash of the root node of the state trie, after all transactions are executed and finalisations applied
Hash function:
KECCAK256()
Simplified World State, σ
Keys
Values
a 7 1 1 3 5 5
45.0 ETH
a 7 7 d 3 3 7
1.00 WEI
a 7 f 9 3 6 5
1.1 ETH
a 7 7 d 3 9 7
0.12 ETH
World State Trie
ROOT: Extension Node
prefix
shared nibble(s)
next node
0
a7
Branch Node
0 1 2 3 4 5 6 7 8 9 a b c d e f
value
Leaf Node
prefix
key-end
value
2
1355
45.0ETH
Extension Node
prefix
shared nibble(s)
next node
0
d3
Leaf Node
prefix
key-end
value
2
9365
1.1ETH
Prefixes
0 – Extension Node, even number of nibbles
1□ – Extension Node, odd number of nibbles,
2 – Leaf Node, even number of nibbles
3□ – Leaf Node, odd number of nibbles
□ = 1st nibble
1 nibble = 4 bits
Branch Node
0 1 2 3 4 5 6 7 8 9 a b c d e f
value
Leaf Node
prefix
key-end
value
3□
7
1.00WEI
Leaf Node
prefix
key-end
value
3□
7
0.12ETH

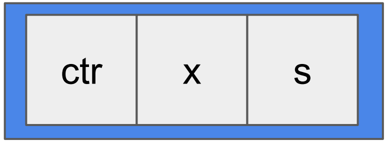
Authenticated data in blockchains

Blockchain

- Each block references a **previous** block
- This reference is by **hash** to its **previous** block, similar to Merkle Trees
- This linked list is called the **blockchain**
- Convention: Arrows show authenticated inclusion



Blocks



- Data structure with three parts:
  - nonce (ctr), data (**x**), reference (s)
  - Typically called the **block header**
- data (**x**) is application-dependent
  - In Bitcoin it stores financial data (“UTXO”-based)
  - In Ethereum it stores contract data (account-based)
- Block validity:
  - Data must be valid (application-defined validity)
- *s*: pointer to the previous block by hash

Proof-of-work in blocks

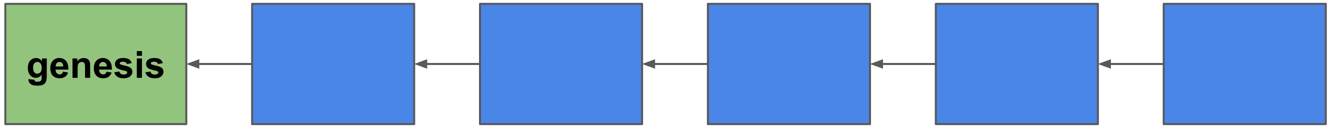
- Blocks must satisfy proof-of-work equation

$$H(ctr \mid\mid \mathbf{x} \mid\mid s) <= T$$

- for some (protocol-specific) T
- ctr is the nonce used to solve proof-of-work
- The value $H(ctr \mid\mid x \mid\mid s)$ is known as the **blockid**

Blockchain

- The **first** block of a blockchain is called the Genesis Block

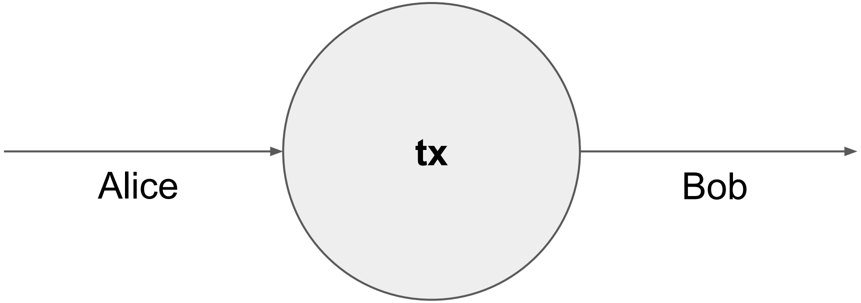


Transactions

A simple transaction for financial data

- Input: contains a proof of spending an existing UTxO
- Output: contains a verification procedure and a value

UTxO = “Unspent Tansaction Output”



Transaction Verification

scriptPubKey (output): OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG

scriptSig (input): <sig> <pubKey>

Bitcoin Script that verifies
the spending of the UTXO

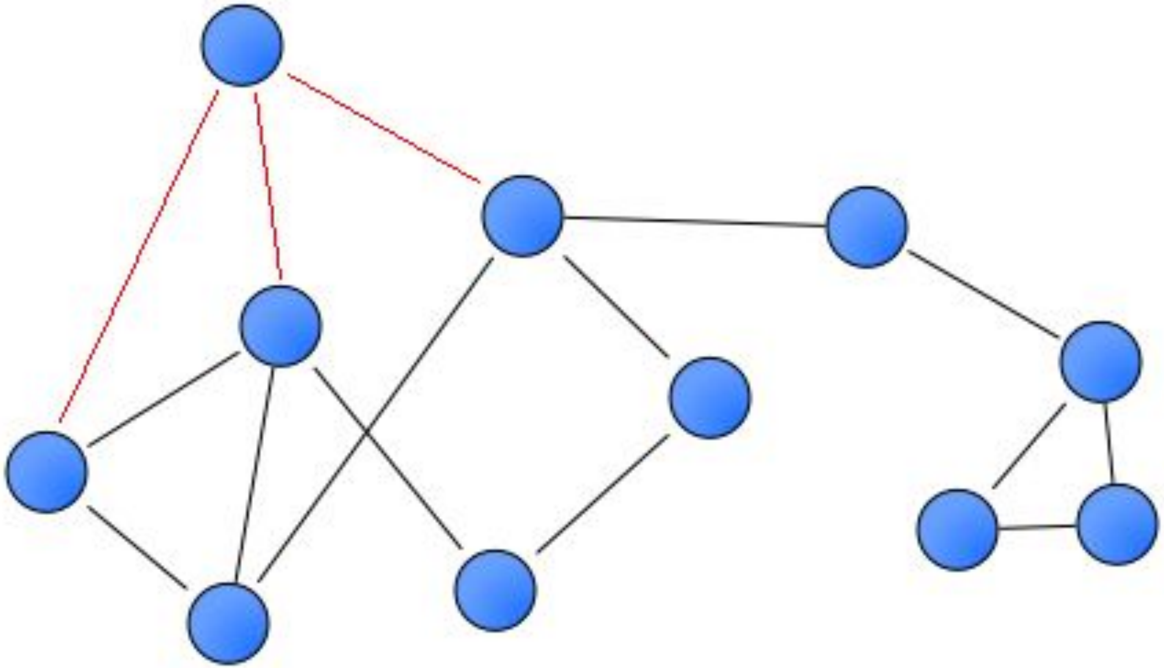
Financial data and Transactions

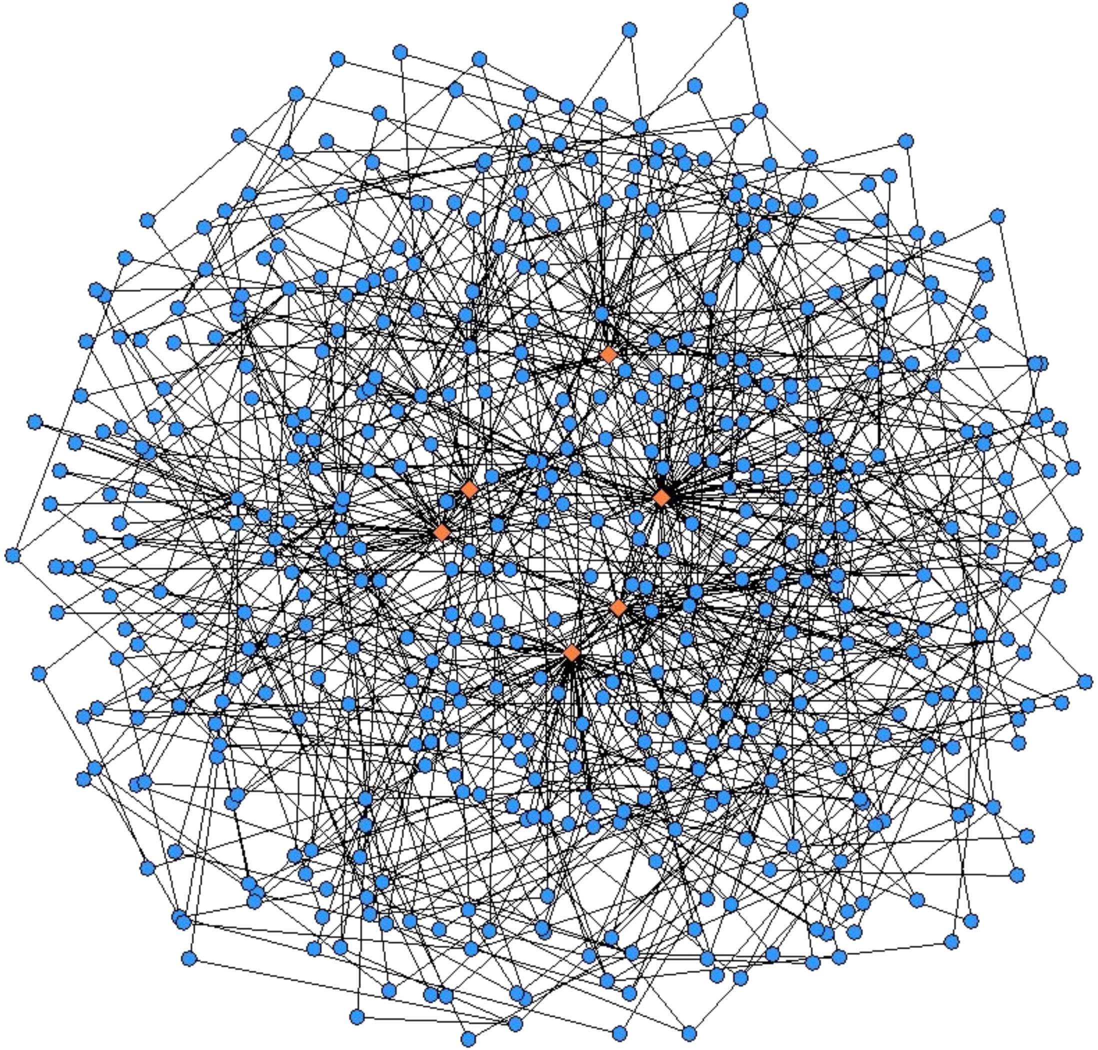
- Financial data is encoded in the form of *transactions*
- Each block organizes transactions in an authenticated data structure
  - Bitcoin: Merkle Tree
  - Ethereum: Merkle Patricia Trie
- Every transaction is sent on the network to everyone via a gossip protocol

The Bitcoin network

The bitcoin network

- All bitcoin nodes connect to a common p2p network
- Each node runs the code of bitcoin
- A node can run on a phone, computer, etc
- Open source code
- Each node connects to its neighbours
- They continuously exchange data
- Each node can **freely** enter the network -- no permission needed!
    - A “permissionless network”
- **The adversarial assumption:**
  There is no trust on the network! Each neighbour can lie.





GLOBAL BITCOIN NODES DISTRIBUTION

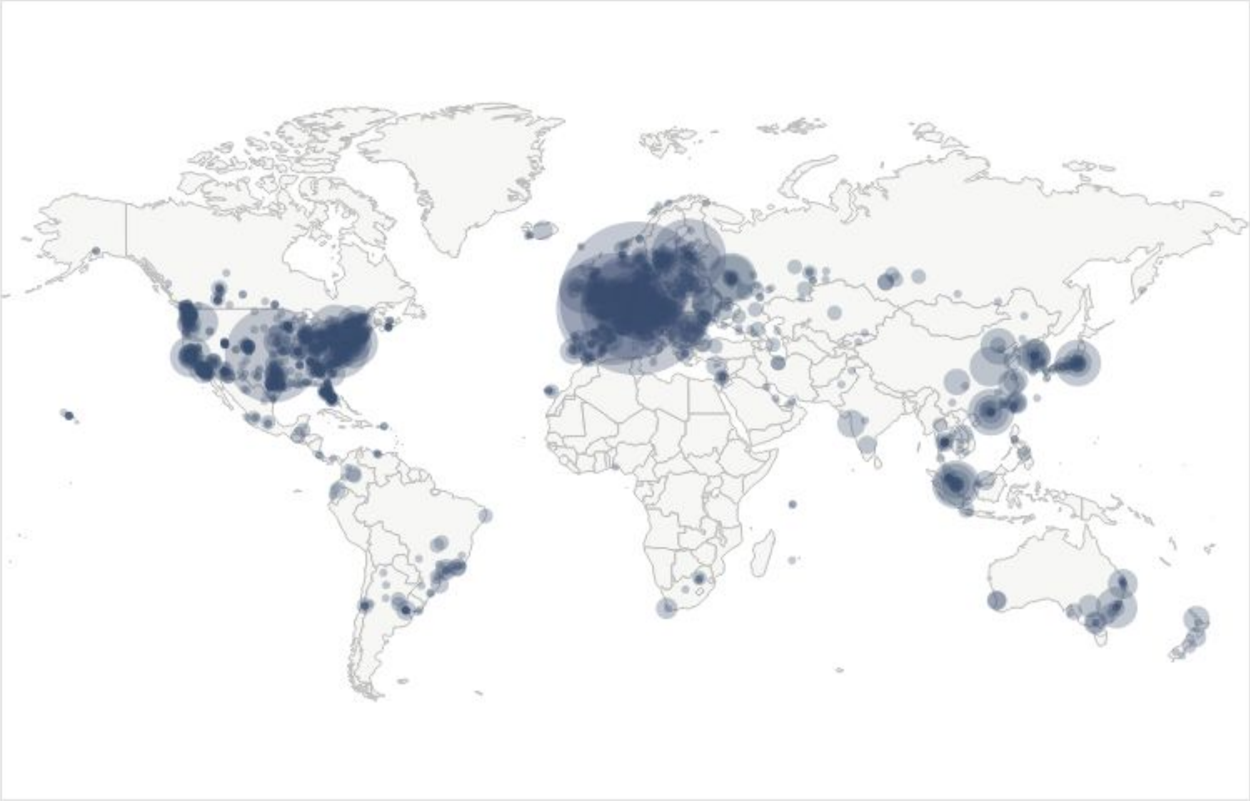
Reachable nodes as of Tue Sep 21 13:27:26 2021 BST.

11784 NODES

24h 90d 1y

Top 10 countries with their respective number of reachable nodes are as follow.

RANK	COUNTRY	NODES
1	n/a	4355 (36.96%)
2	United States	1833 (15.55%)
3	Germany	1804 (15.31%)
4	France	549 (4.66%)
5	Netherlands	381 (3.23%)
6	Canada	313 (2.66%)
7	United Kingdom	254 (2.16%)
8	Russian Federation	196 (1.66%)
9	Finland	186 (1.58%)
10	China	133 (1.13%)



https://bitnodes.io/

Peer discovery

- Each node stores a list of peers (by IP address)
- When Alice connects to Bob, Bob sends Alice his own known peers
- That way, Alice can learn about new peers

Bootstrapping the p2p network

- Peer-to-peer nodes come “preinstalled” with some peers by IP / host
- When running a node, you can specify extra “known peers”

The *gossip* protocol

- When a node **Alice** generates some new data...
- Alice **broadcasts** data to its peers
- Each peer broadcasts this data to *its* peers
- If a peer has seen this data before, it ignores it
- If this data is new, it broadcasts it to its peers
- That way, the data spreads like an epidemic, until the whole network learns it
- This process is called **diffuse**

Eclipse attacks

- Isolate the some honest nodes in the network effectively causing a “net split” in two partitions A and B
- If peers in A and peers in B are disjoint and don’t know about each other, the networks will remain isolated
  - Recent attack: Erebus
- The connectivity assumption:
  - There is a path between two nodes on the network
  - **If a node broadcasts a message, every other node *will* learn it**