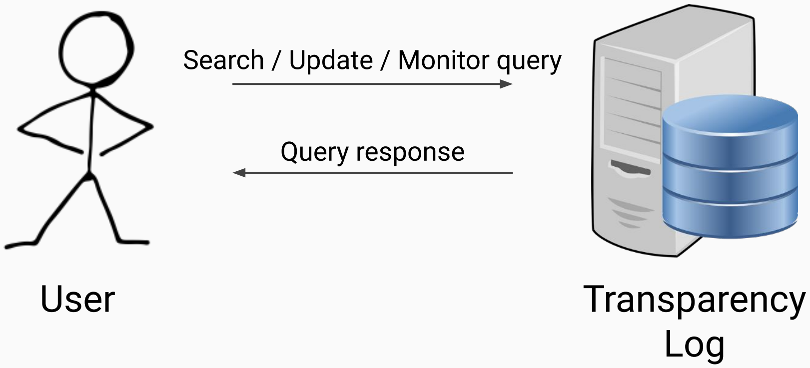
draft-keytrans-mcmillion-protocol

Brendan McMillion
IETF 120 / July 26, 2024

Basic Model

1. **Search:** What's the value of this key?
2. **Update:** Here's a new value for this key!
3. **Monitor:** What's new with my keys?

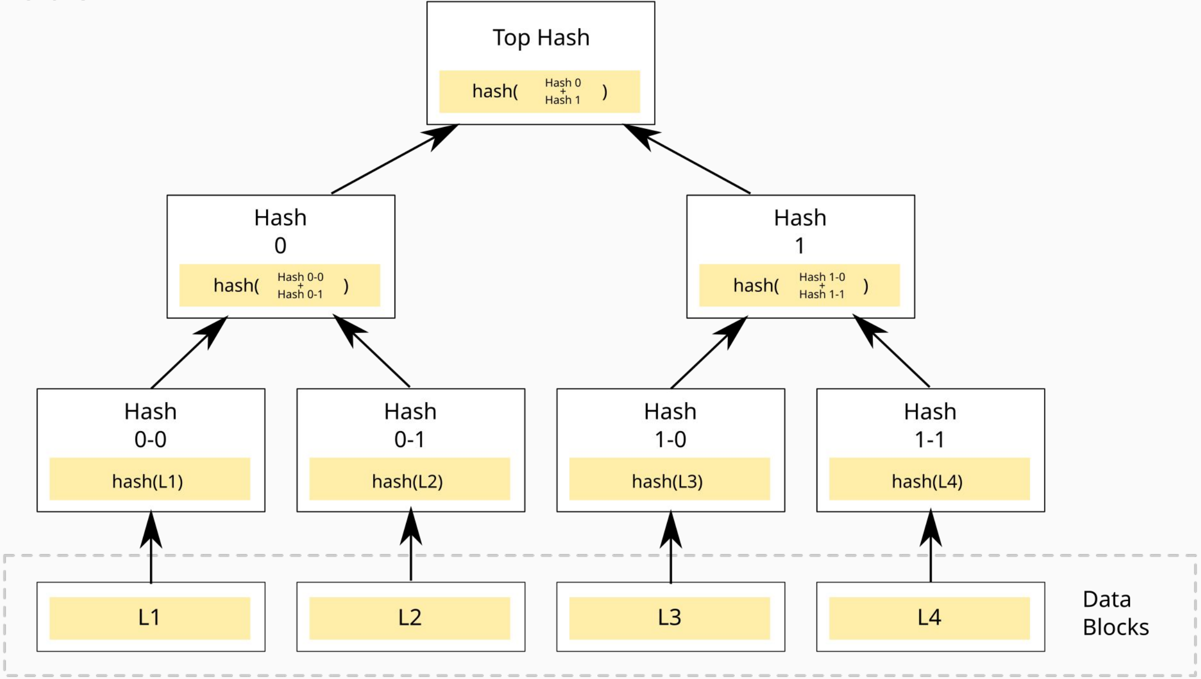
- Looks like a key-value database
- Transparency Log enforces access control rules by simply rejecting queries that aren't allowed
- User (generally) only needs direct communication with the Transparency Log



Tree Construction

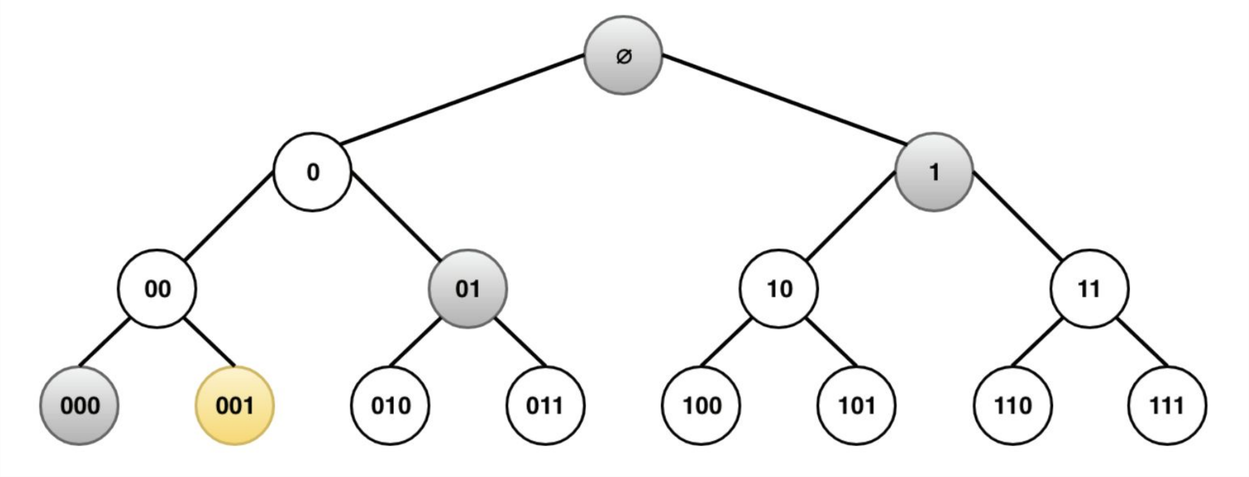
Hash trees generally

- **Leaf Node:** Contains the hash of some data
- **Intermediate Node:** Contains the hash of its children nodes
- **Root Node:** Topmost node



Hash trees generally (cont.)

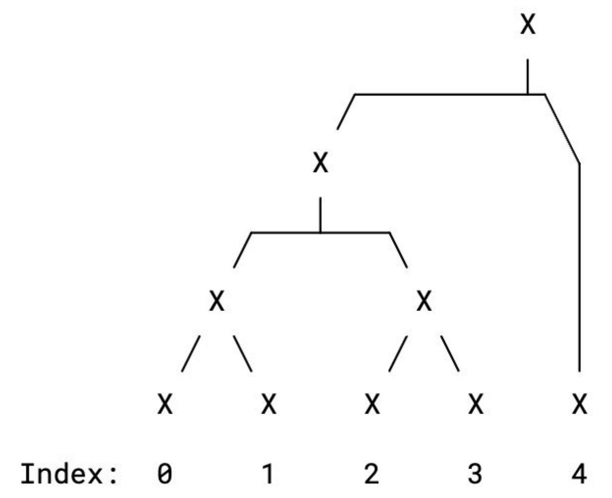
- **Copath:** List of nodes that I can hash to get the root



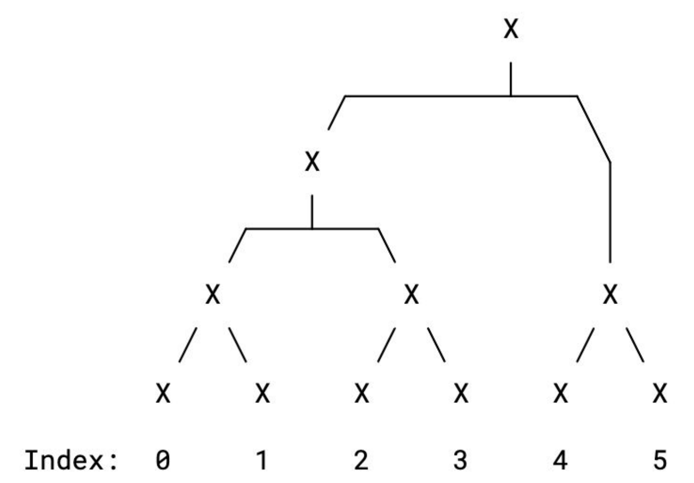
Log Tree

Left-Balanced Binary Tree: Every left subtree contains more/equal nodes as the right subtree

- Easy to append new data
- Easy to provide consistency proofs



(Five leaves)

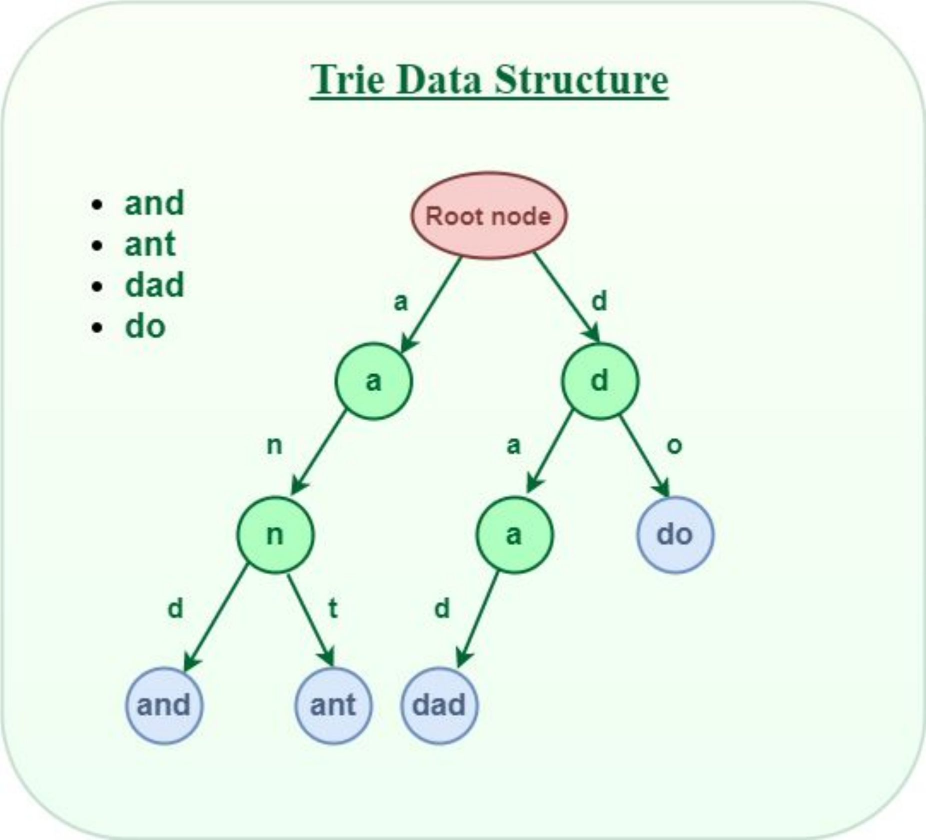


(Six leaves)

Prefix Tree

Prefix Tree: Every path from the root to a leaf corresponds to the prefix of a string stored in the trie

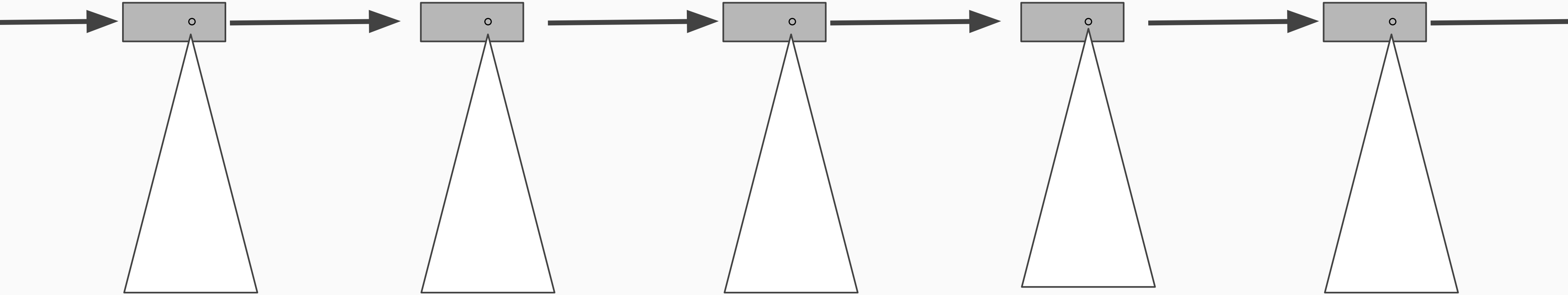
- Easy to provide proof that a specific string is or isn’t in the tree



Combined Tree

Combined Tree: Log Tree where each leaf / log entry contains:

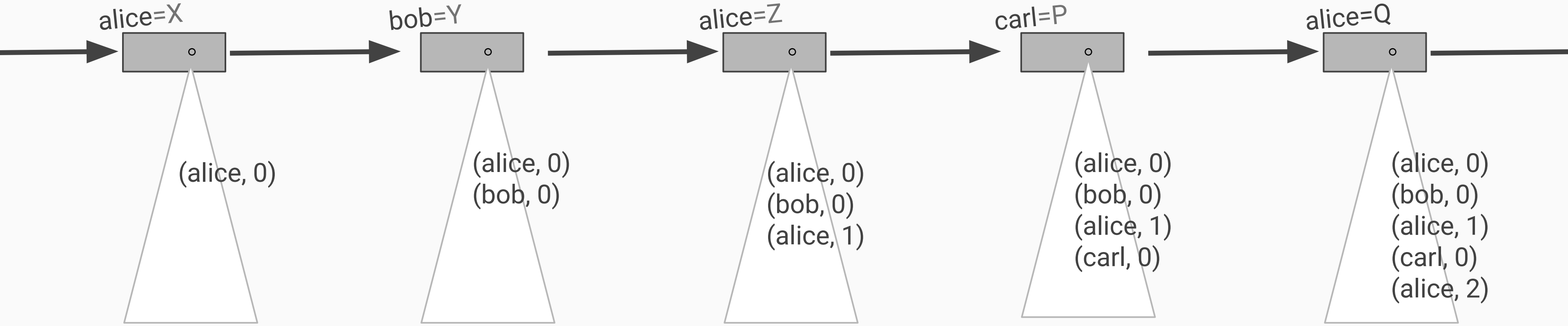
1. Cryptographic commitment
2. Root hash of a prefix tree



Combined Tree (cont.)

Combined Tree: Log Tree where each leaf / log entry contains:

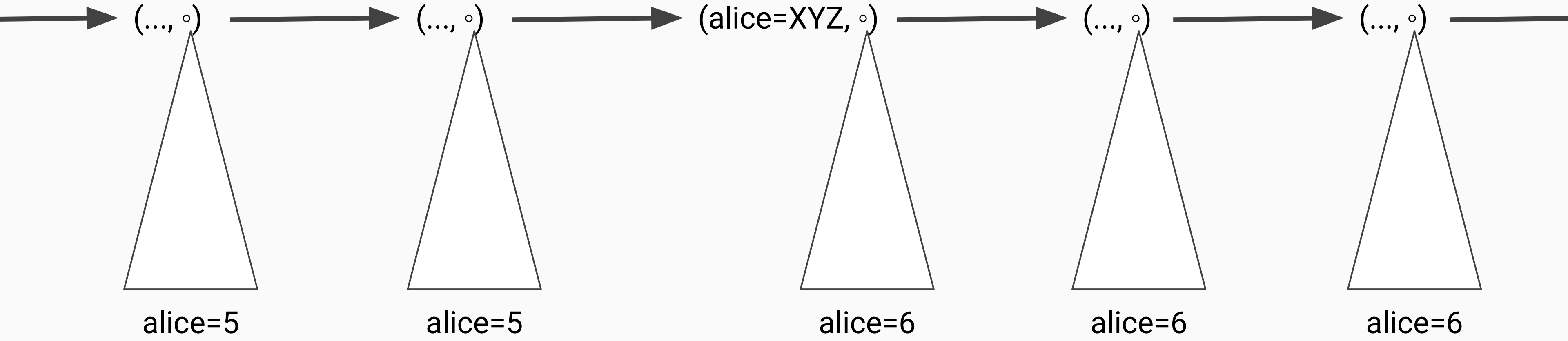
1. Cryptographic commitment (to statement that key-value pair was inserted)
2. Root hash of a prefix tree (containing total set of key-version pairs)



Searching the Tree

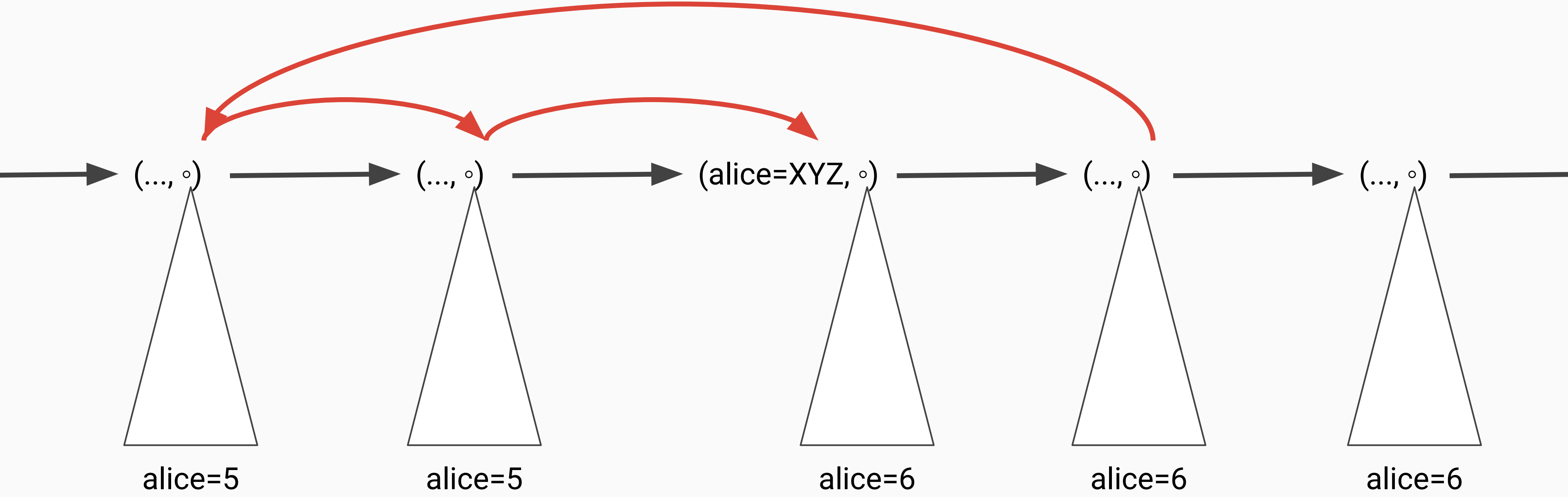
Implicit Binary Tree

- Combined Tree is essentially a WAL
- Prefix Tree in each log entry shows the greatest version of a key that exists, as of a given log entry



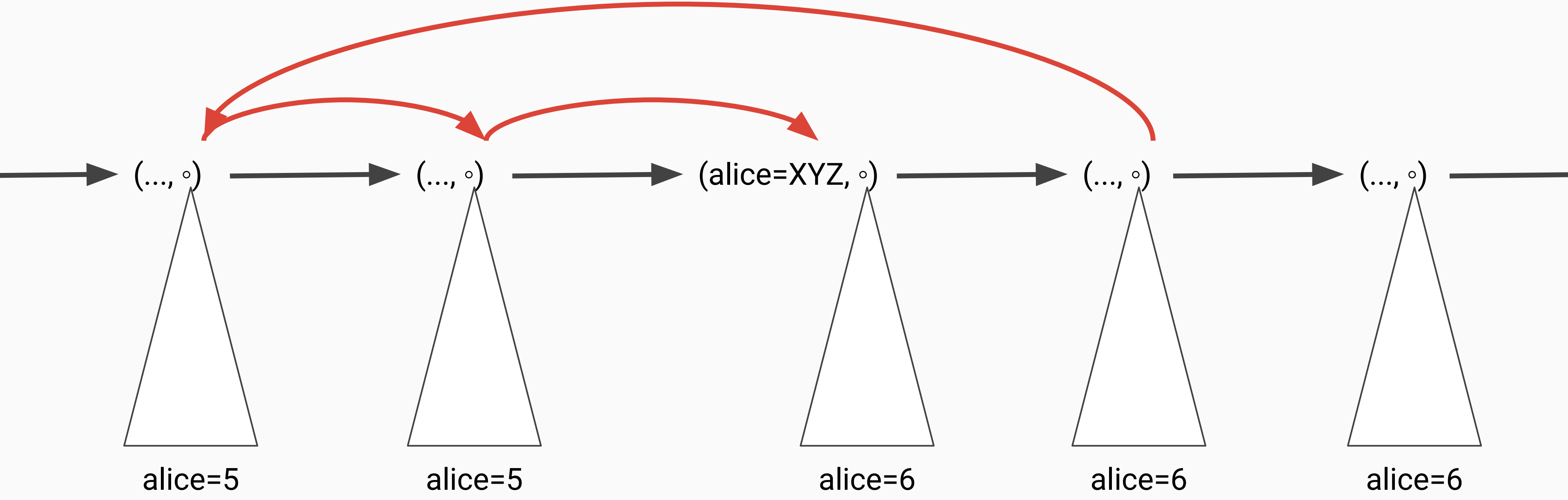
Implicit Binary Tree (cont.)

- Searching for specific version of a key = binary search



Monitoring

- Users need to ensure searches for their keys converge to log entry they expect
  - Security of this comes from binary search paths changing infrequently
  - Efficiency comes from binary search paths being logarithmic



Search

Algorithm:

1. Do binary search. For each log entry in binary search path:
    a. **Look in prefix tree to find the greatest version of the key that it contains** ⬅ How to do this?
    b. Use this to determine which log entry to inspect next
2. Once final log entry is found, open commitment to get key's value

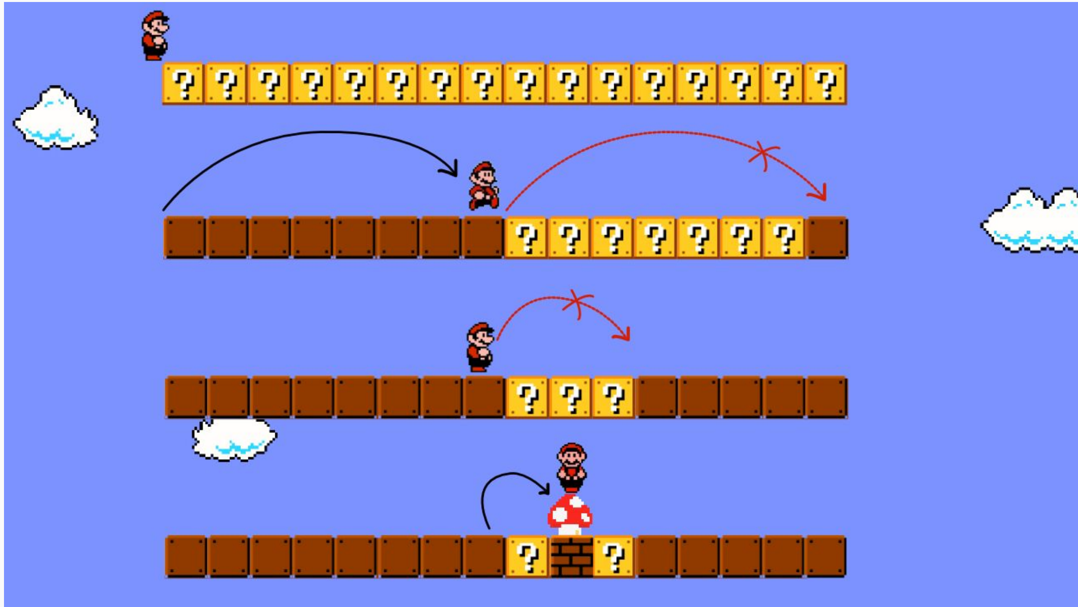
Binary Ladder

Need a way to get most recent version from Prefix Tree:

- Prefix Tree can't contain ONLY recent version for privacy reasons
    - Finding greatest version of key = kinda also binary search!!

Example:

- Does 1 exist? Yes
- Does 2 exist? Yes
- Does 4 exist? Yes
- Does 8 exist? No!!
- Does 6 exist? Yes
- Does 7 exist? No!!
    - Must be 6!



Search (cont.)

Final Algorithm:

1. Do binary search. For each log entry in binary search path:
   a. Look in prefix tree to find the greatest version of the key that it contains (= do another binary search)
   b. Use this to determine which log entry to inspect next
2. Once final log entry is found, open commitment to get key’s value

Recap

Security: Users can search for and monitor keys with confidence that everyone is seeing the same thing

Privacy: To an outsider, every change just looks like one new entry to the Log Tree, one new entry to the Prefix Tree

Efficiency: Every client-side algorithm is ~logarithmic