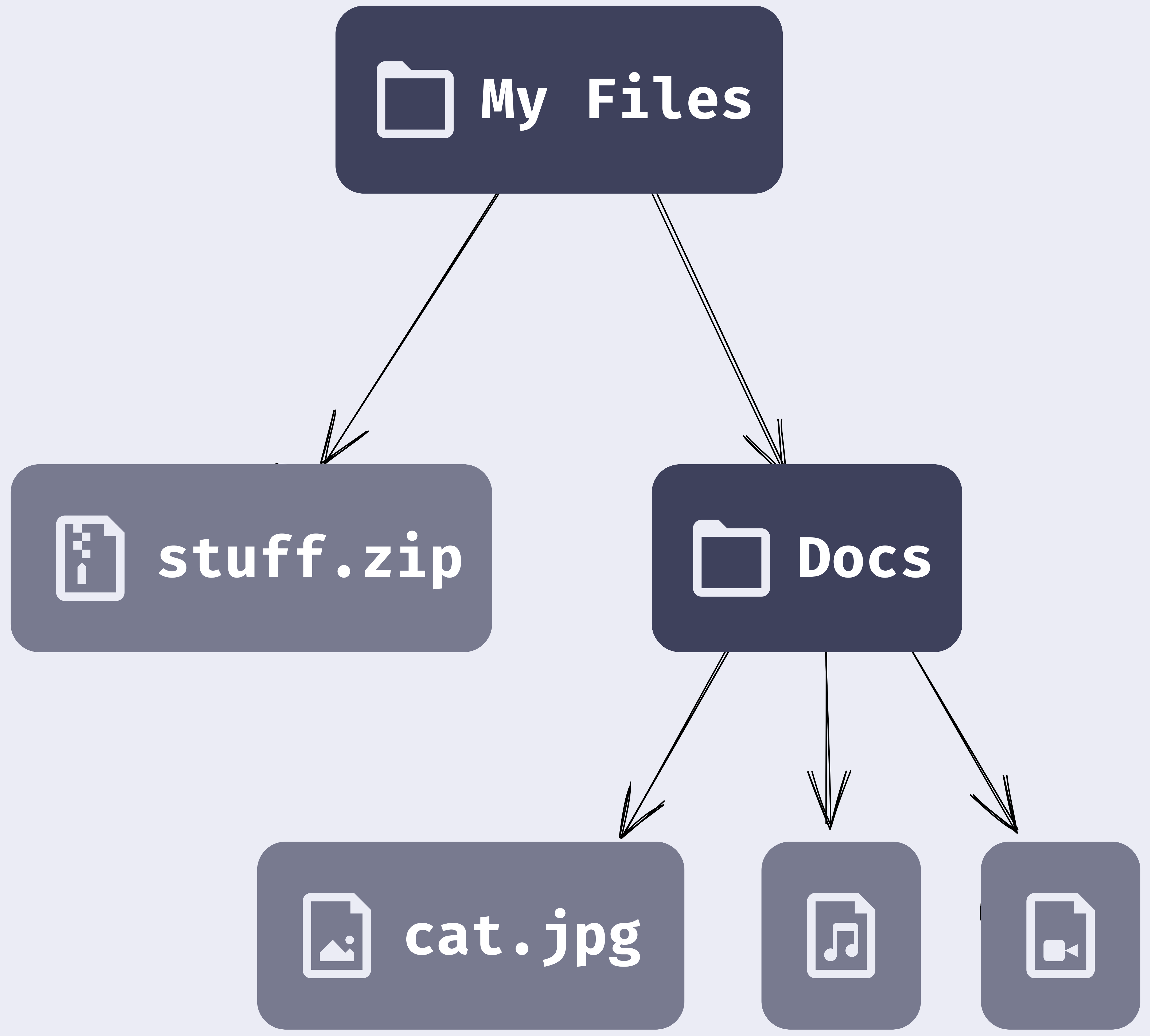
WNFS “v2” Munch & Learn

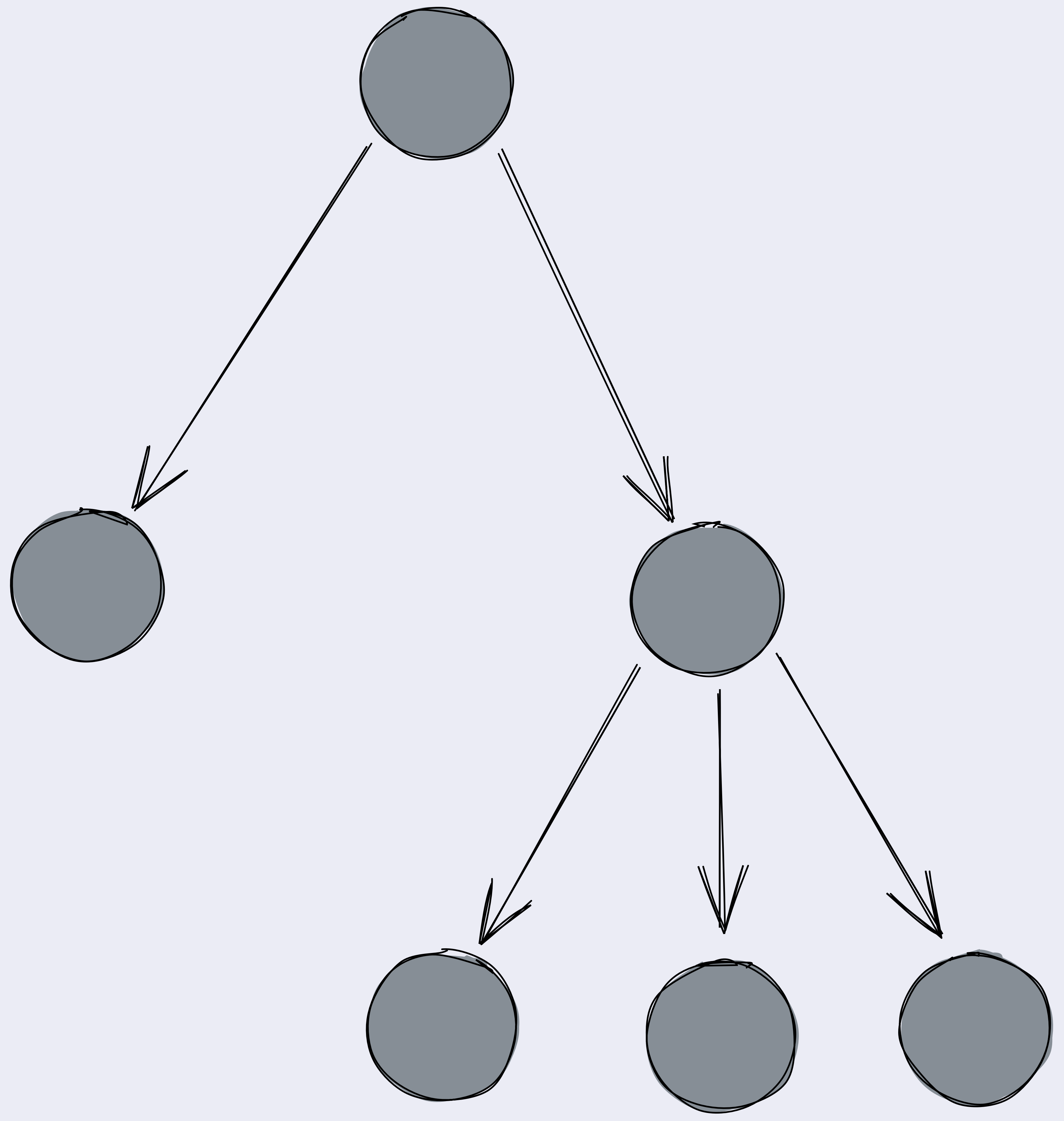
(**WNFS** = **W**eb **N**ative **F**ile **S**ystem)

Public WNFS



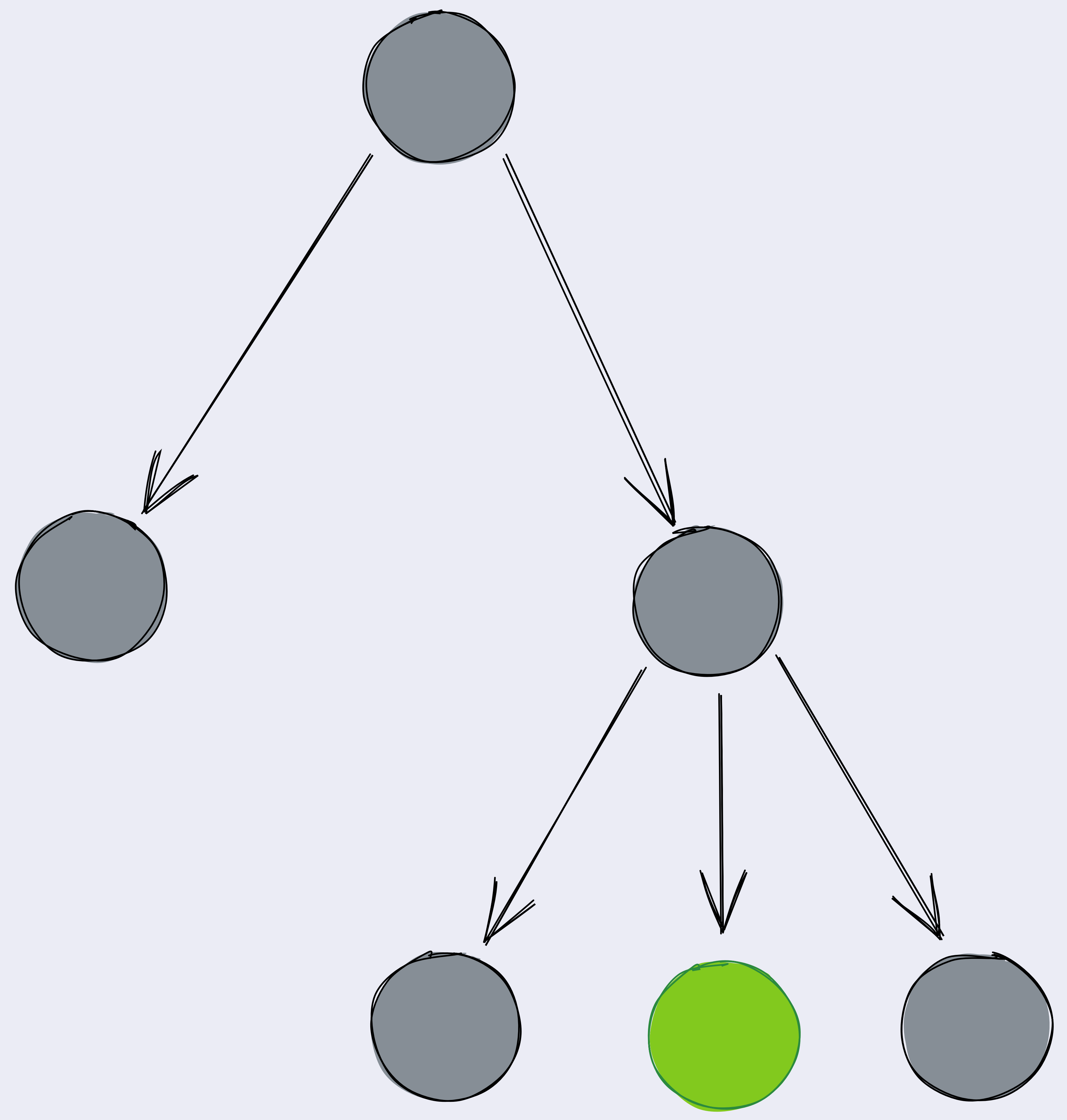
Public WNFS

- Based on UnixFS
- Merkle tree
  - Directories include hashes of children



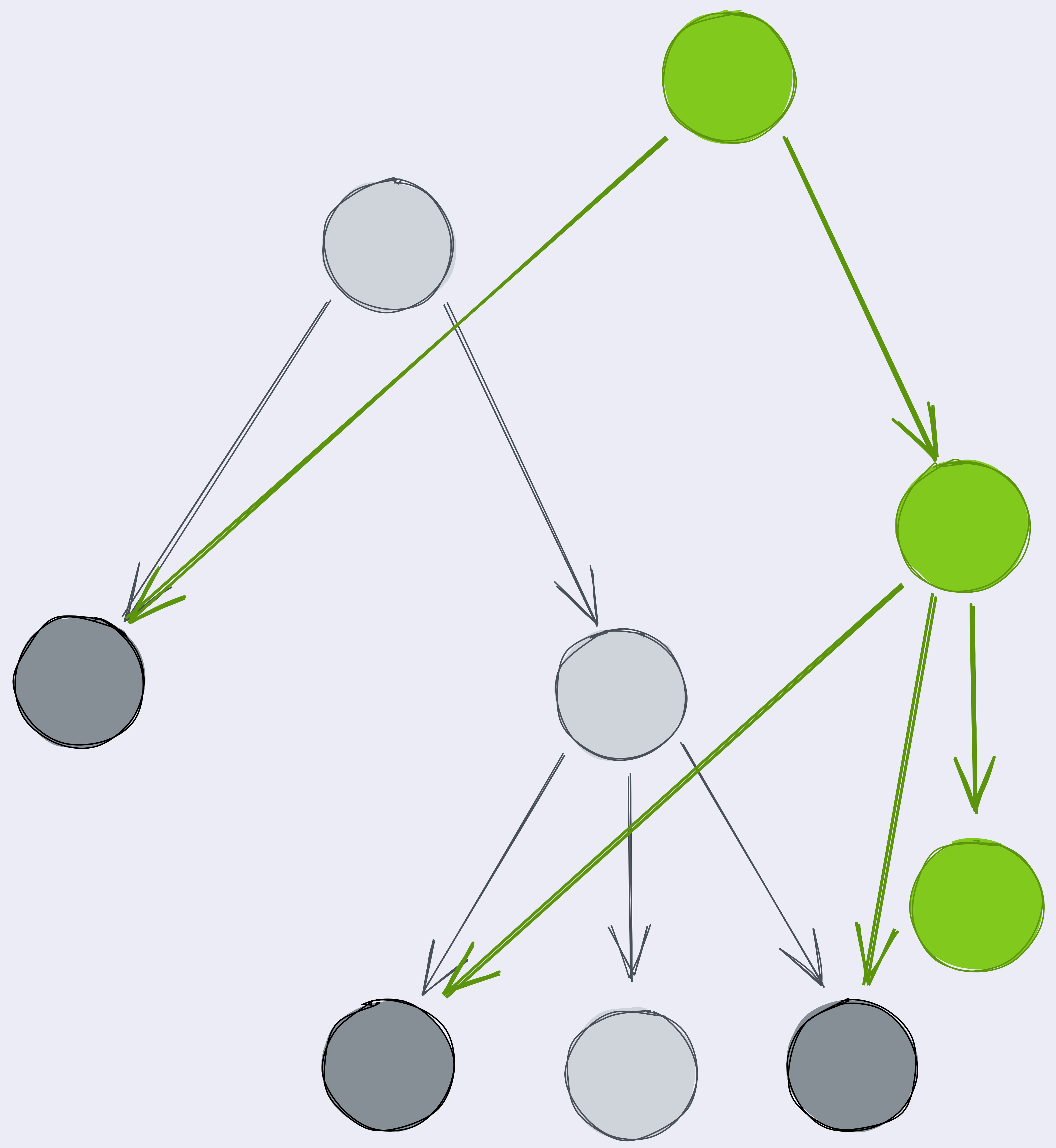
Public WNFS

- Based on UnixFS
- Merkle tree
    - Directories include hashes of children
- CIDs → inherent immutability
- “Changes”



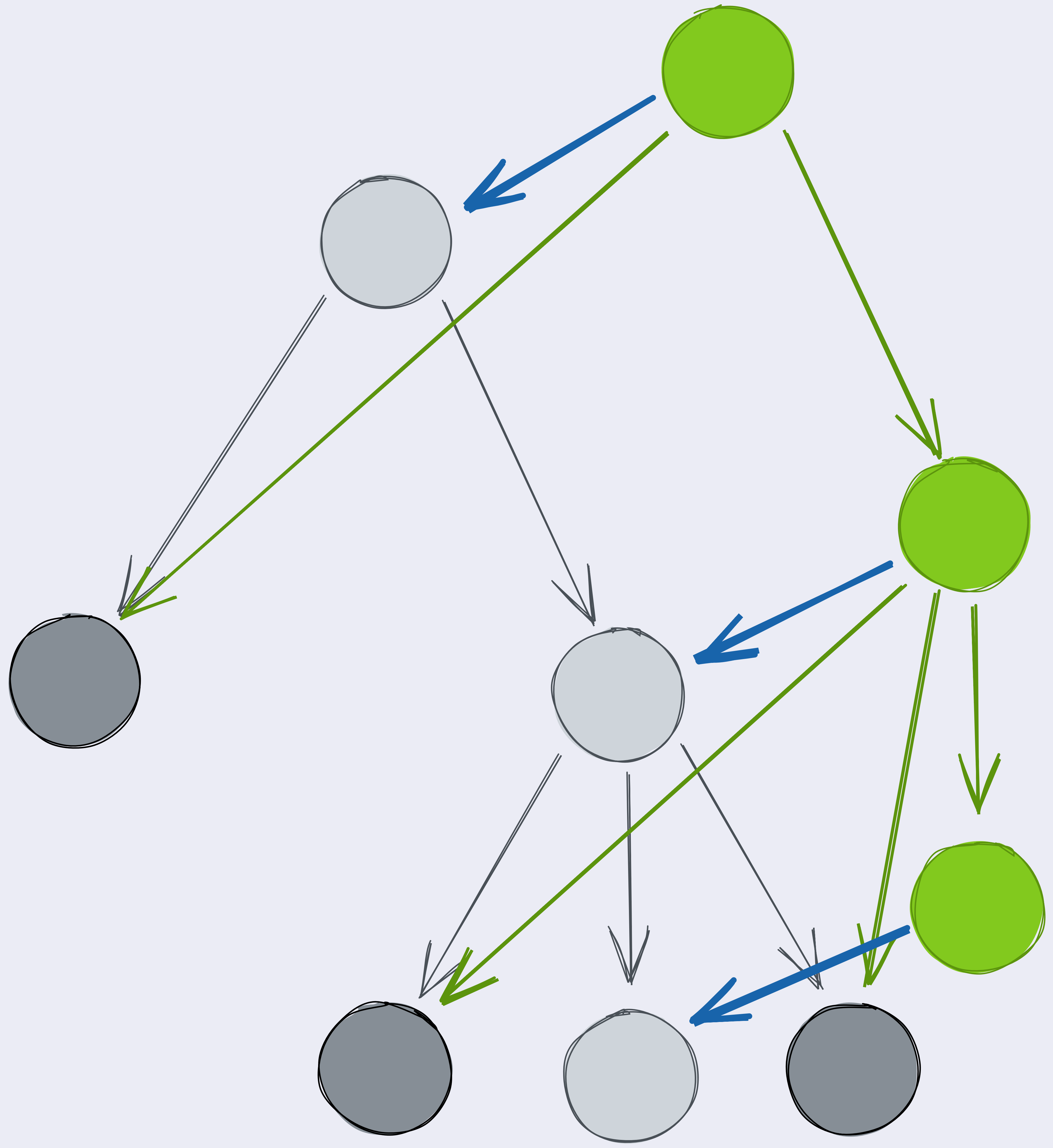
Public WNFS

- Based on UnixFS
- Merkle tree
  - Directories include hashes of children
- CIDs → inherent immutability
- “Changes”
  - → new blocks
  - → new root



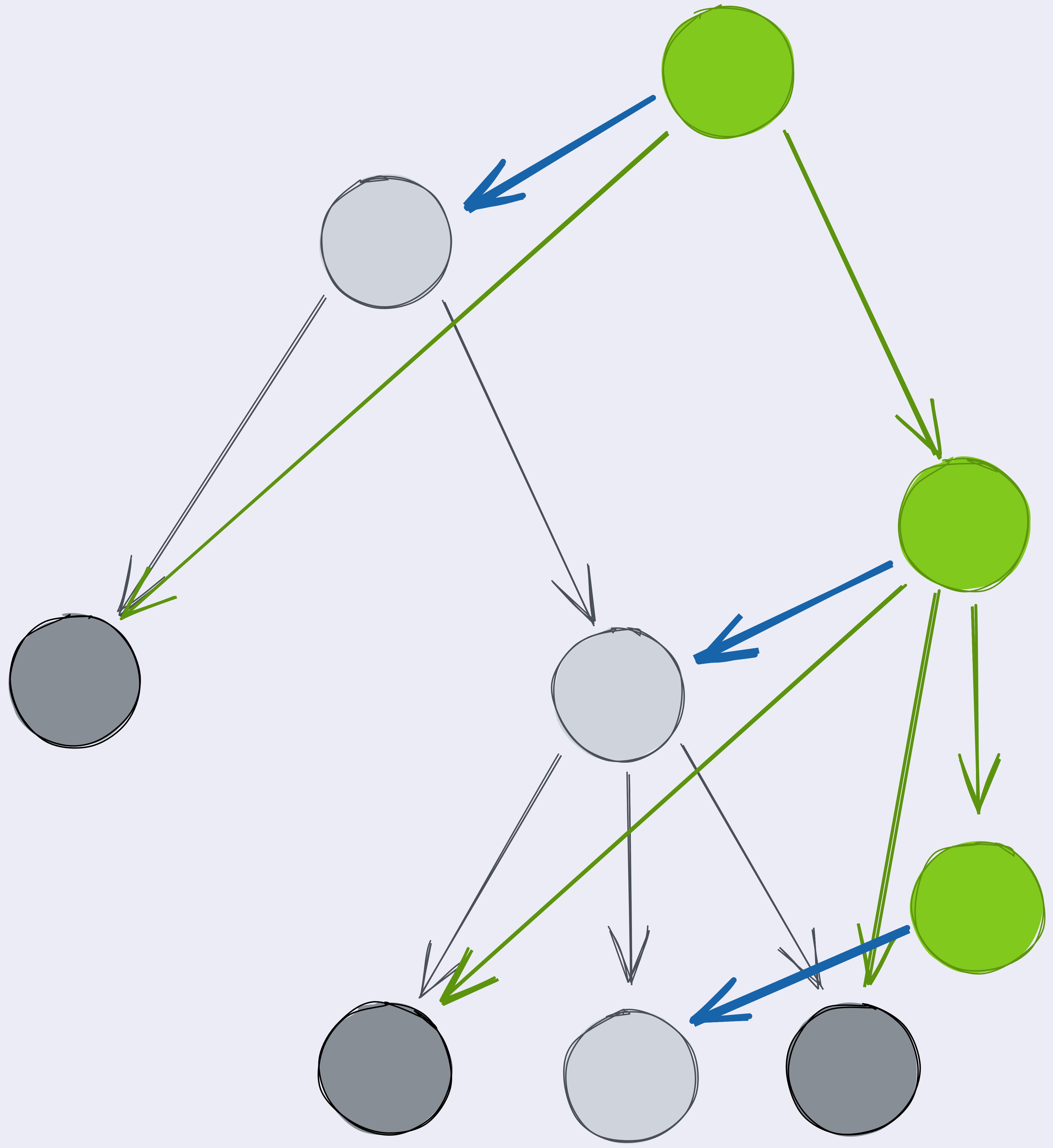
Public WNFS

- WNFS-specific:
- Backlinks (“previous”)
  - Versioning!
  - Preserves all information
  - → Allows WNFS merges

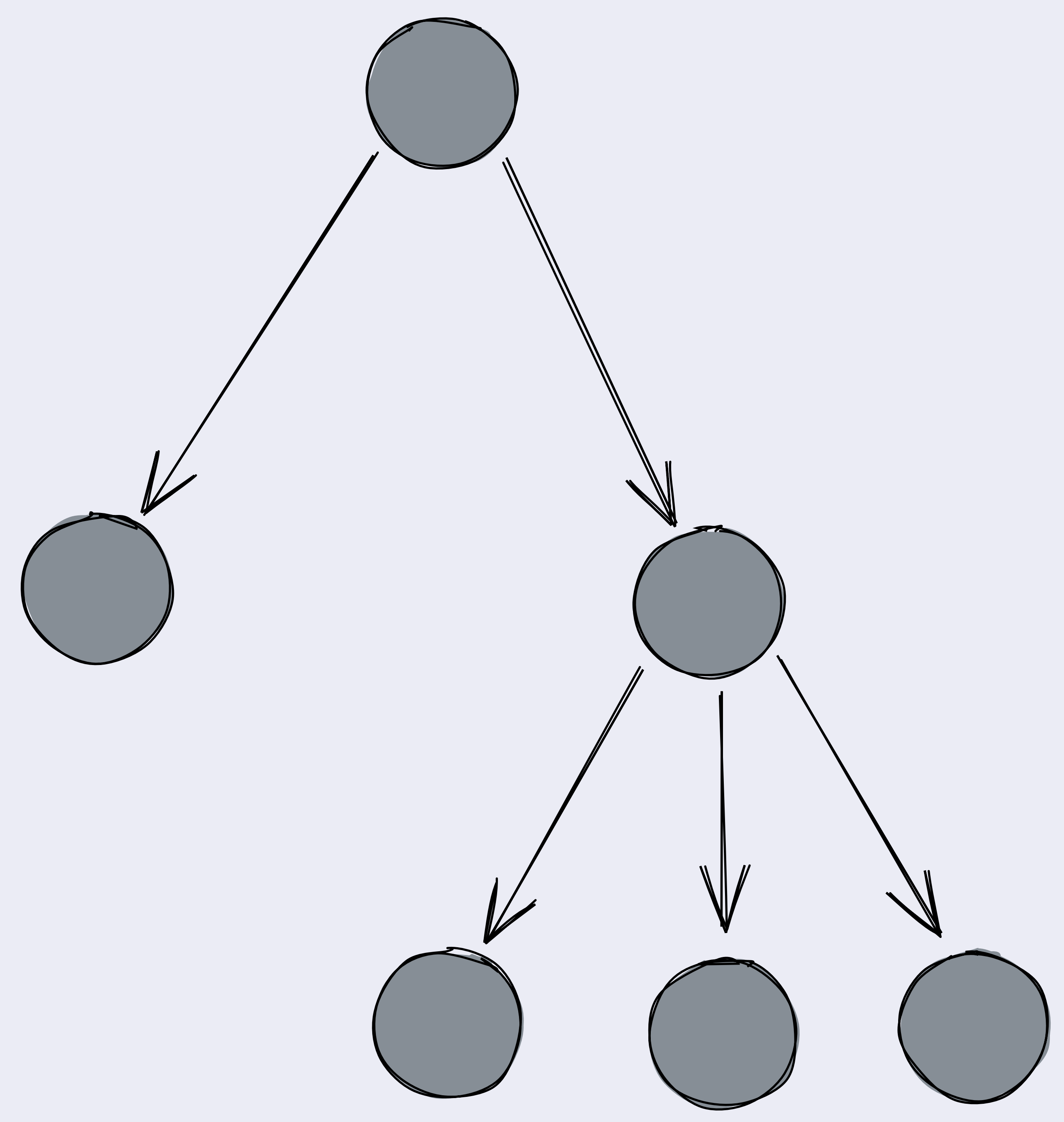


Public WNFS

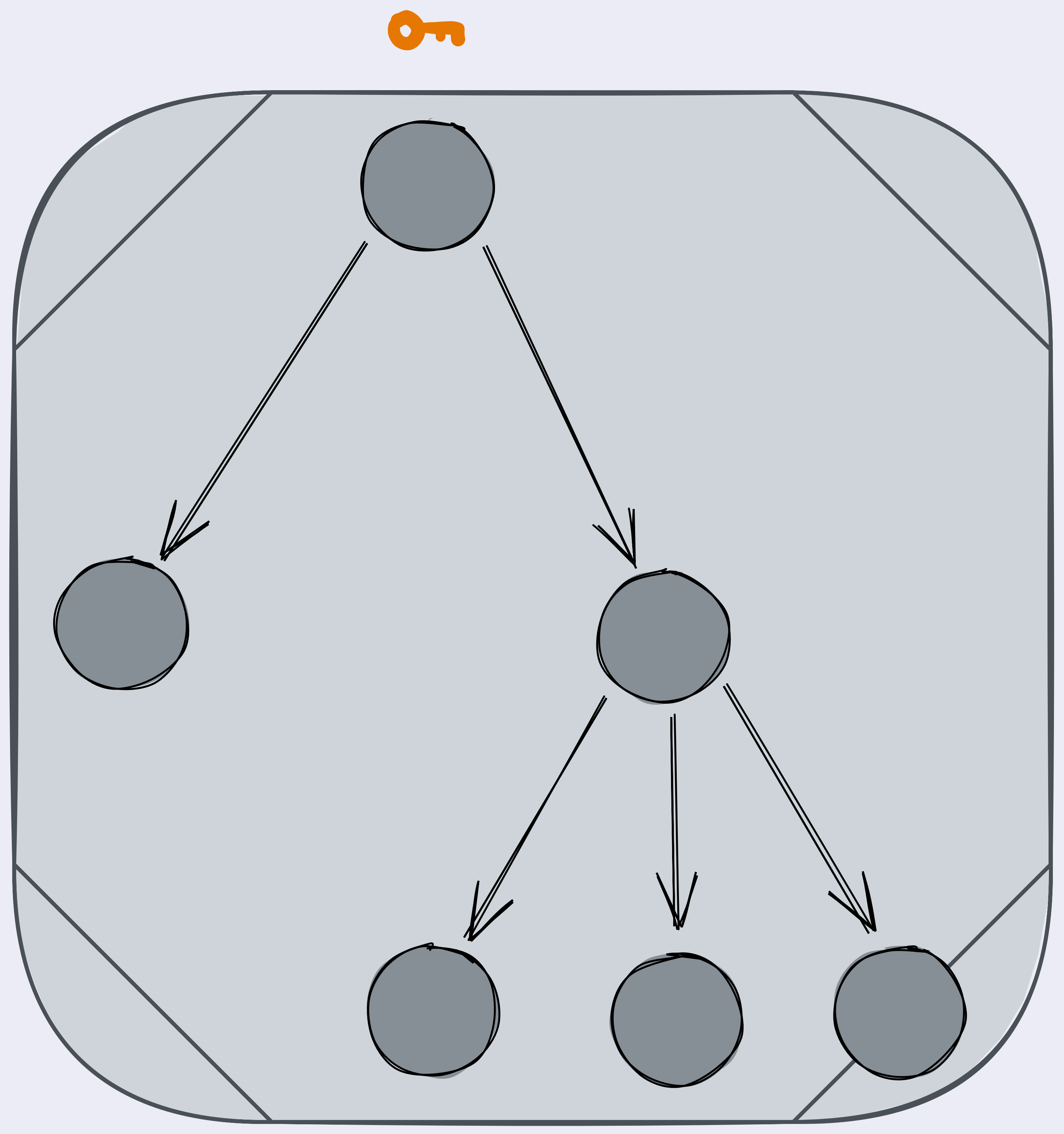
- WNFS-specific:
- Backlinks ("previous")
  - Versioning!
  - Preserves all information
  - → Allows WNFS merges
- That's almost everything!
- The rest:
  - Arbitrary metadata
  - Merge nodes have multiple "previous" links
  - Symlinks



Private WNFS

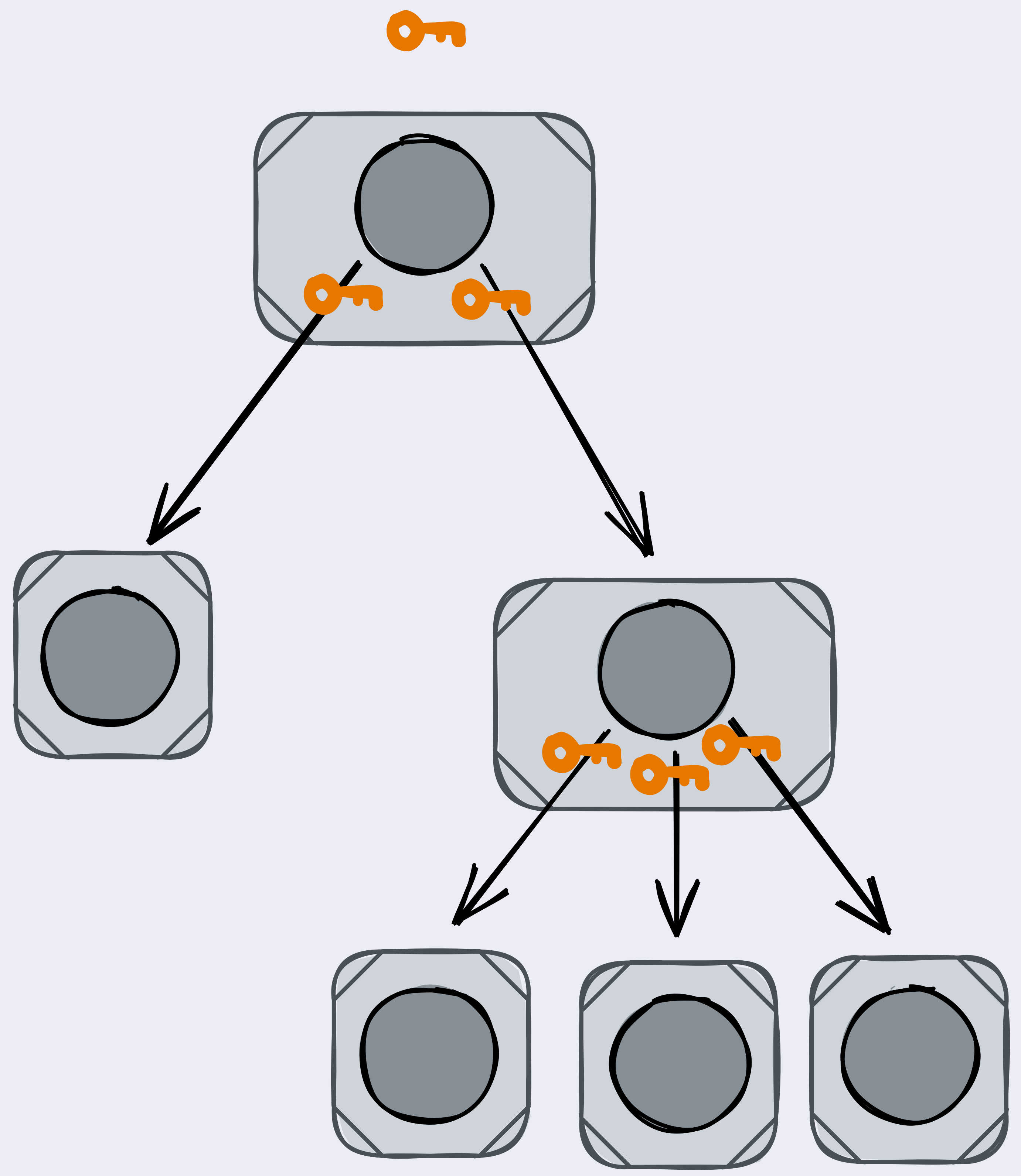


Private WNFS



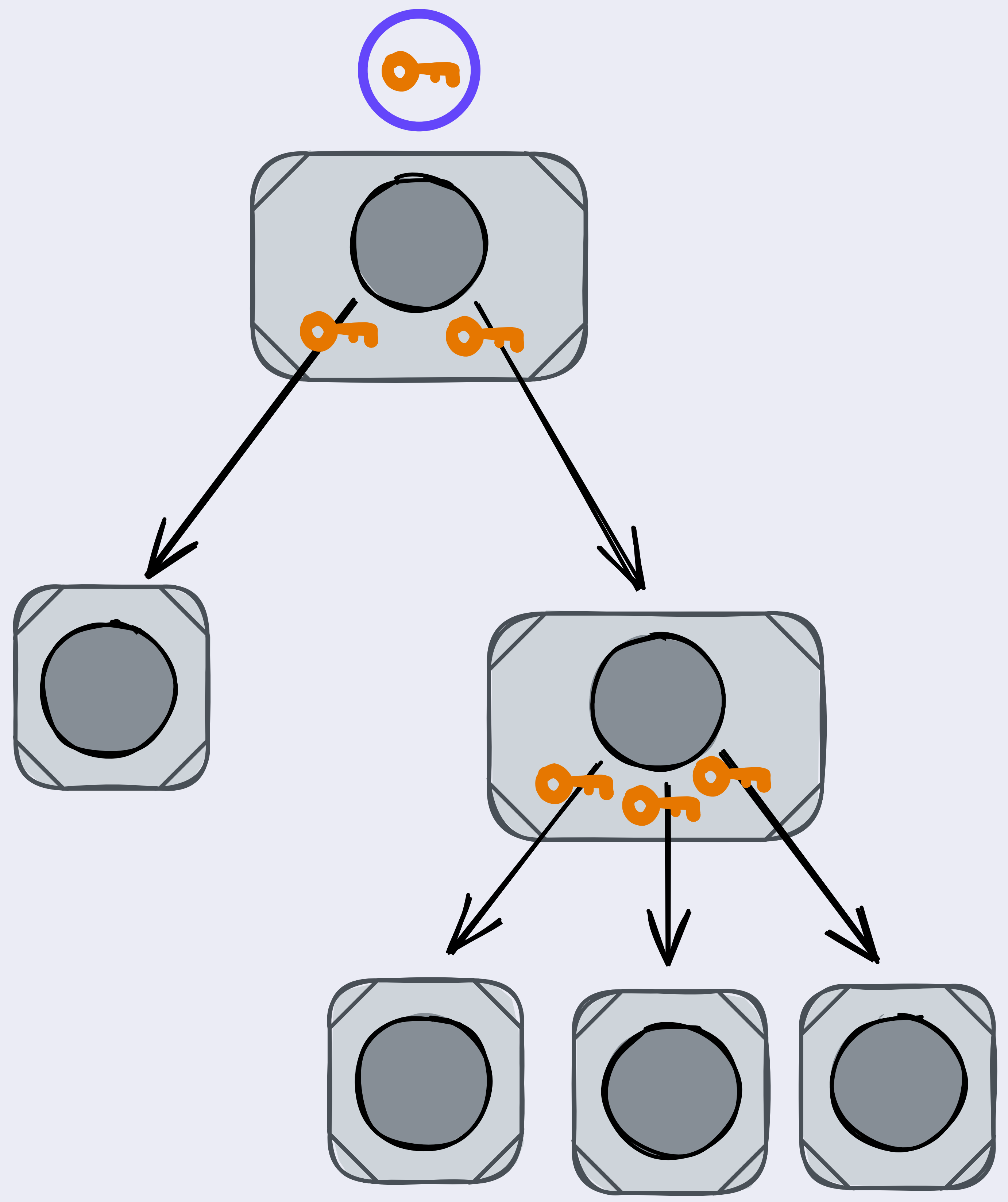
Private WNFS

- Based on cryptrees
- Encrypt all directories/files with symmetric encryption
- Include keys to decrypt children



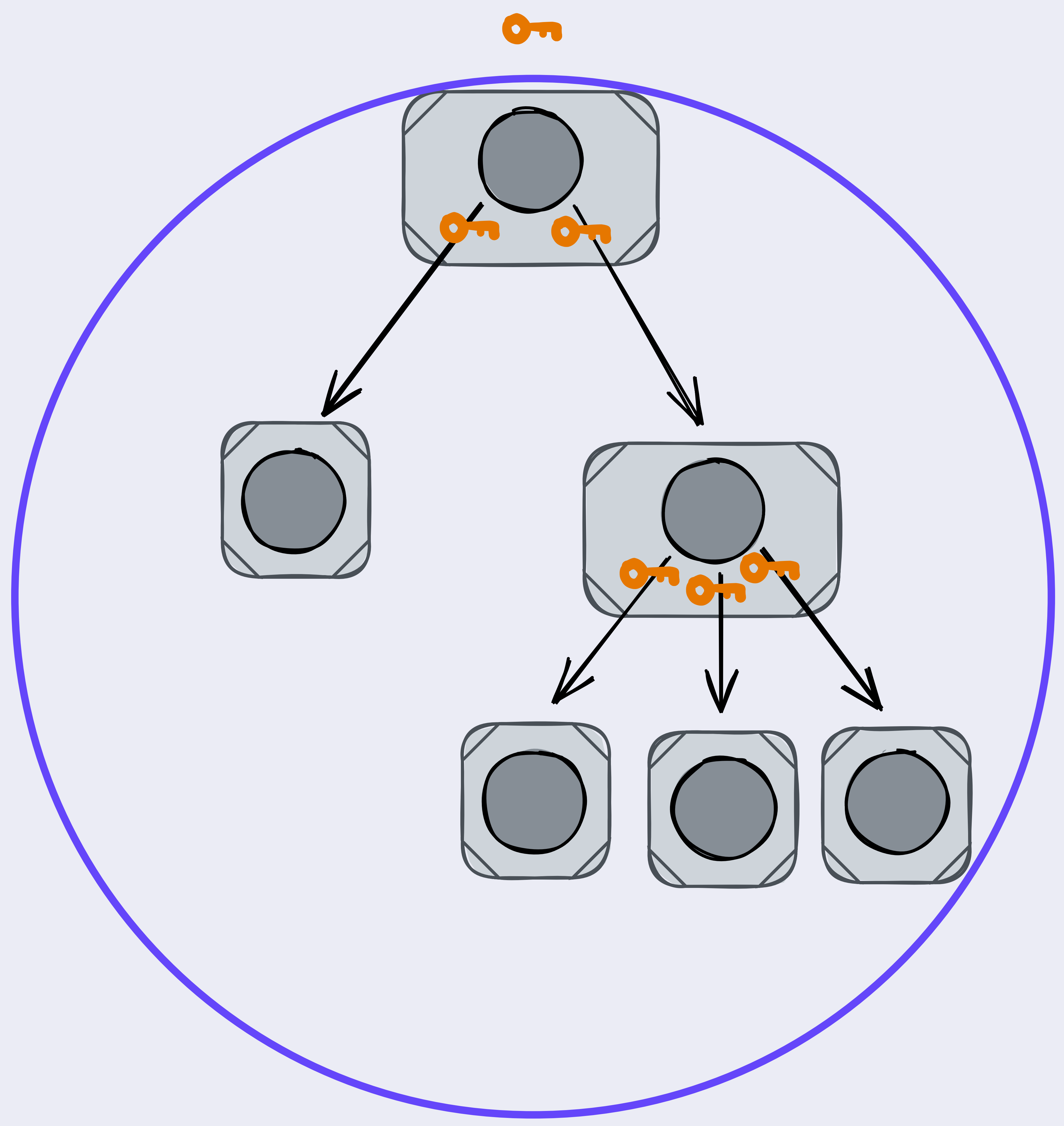
Private WNFS

- Based on cryptrees
- Encrypt all directories/files with symmetric encryption
- Include keys to decrypt children



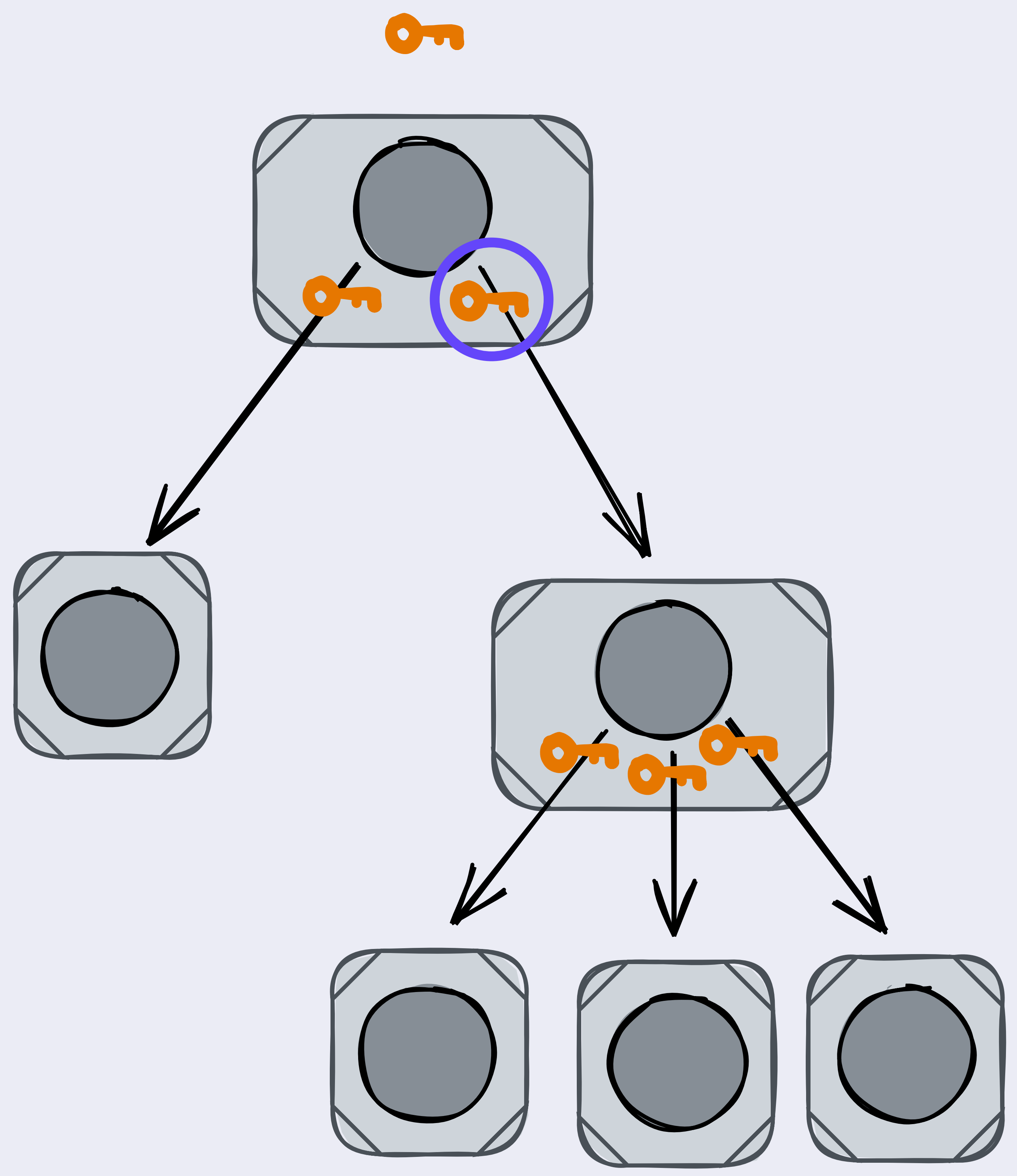
Private WNFS

- Based on cryptrees
- Encrypt all directories/files with symmetric encryption
- Include keys to decrypt children



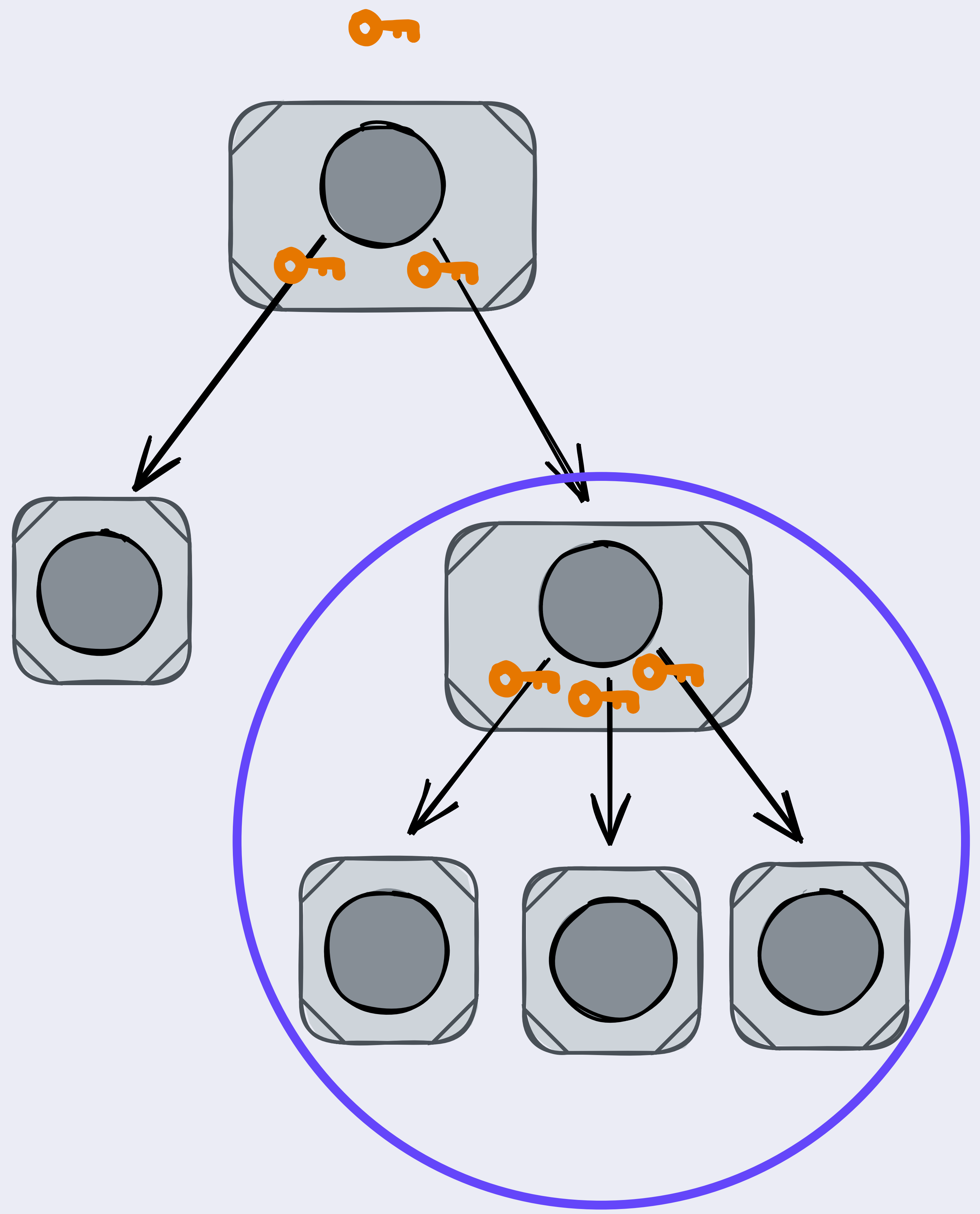
Private WNFS

- Based on cryptrees
- Encrypt all directories/files with symmetric encryption
- Include keys to decrypt children



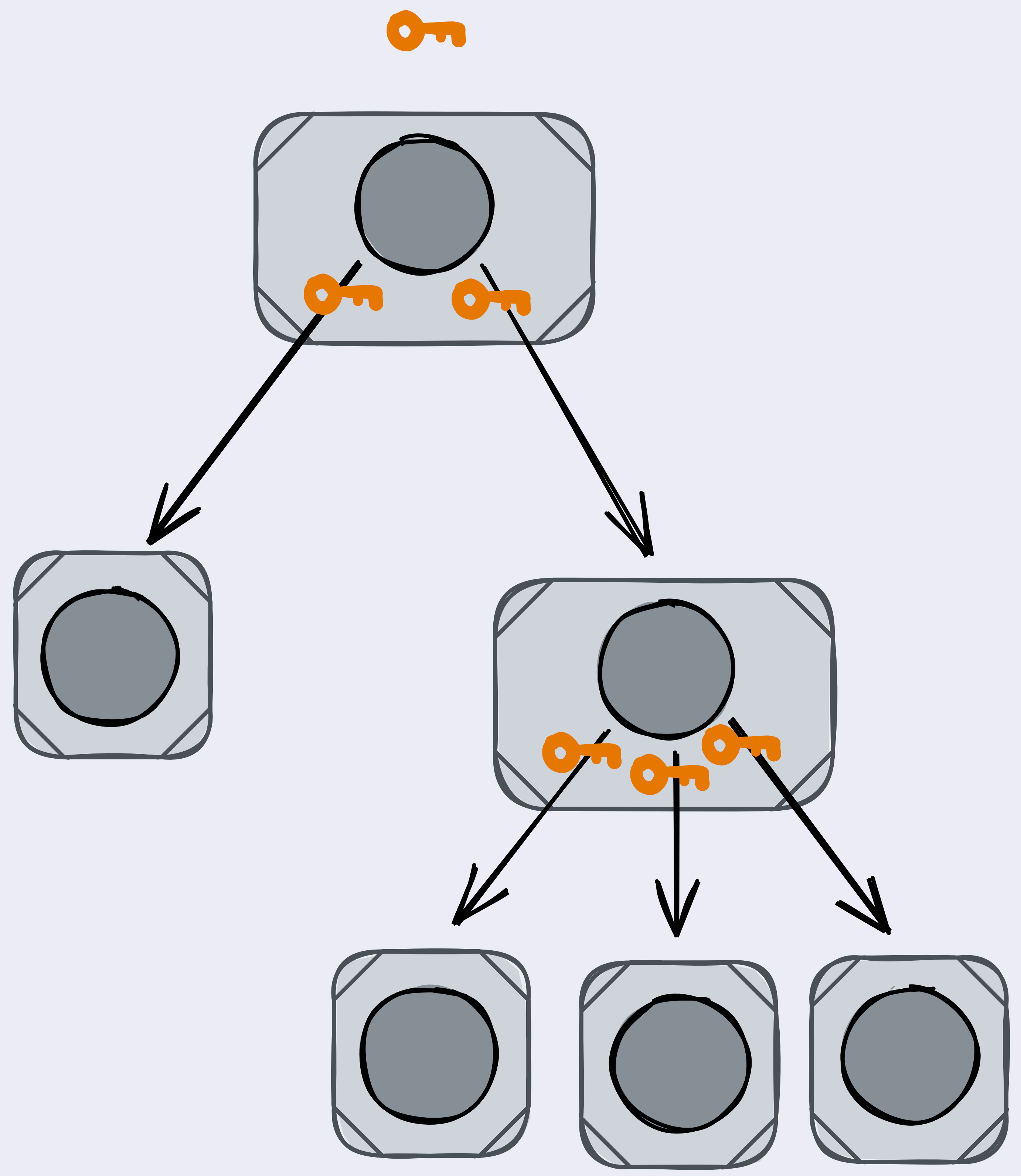
Private WNFS

- Based on cryptrees
- Encrypt all directories/files with symmetric encryption
- Include keys to decrypt children



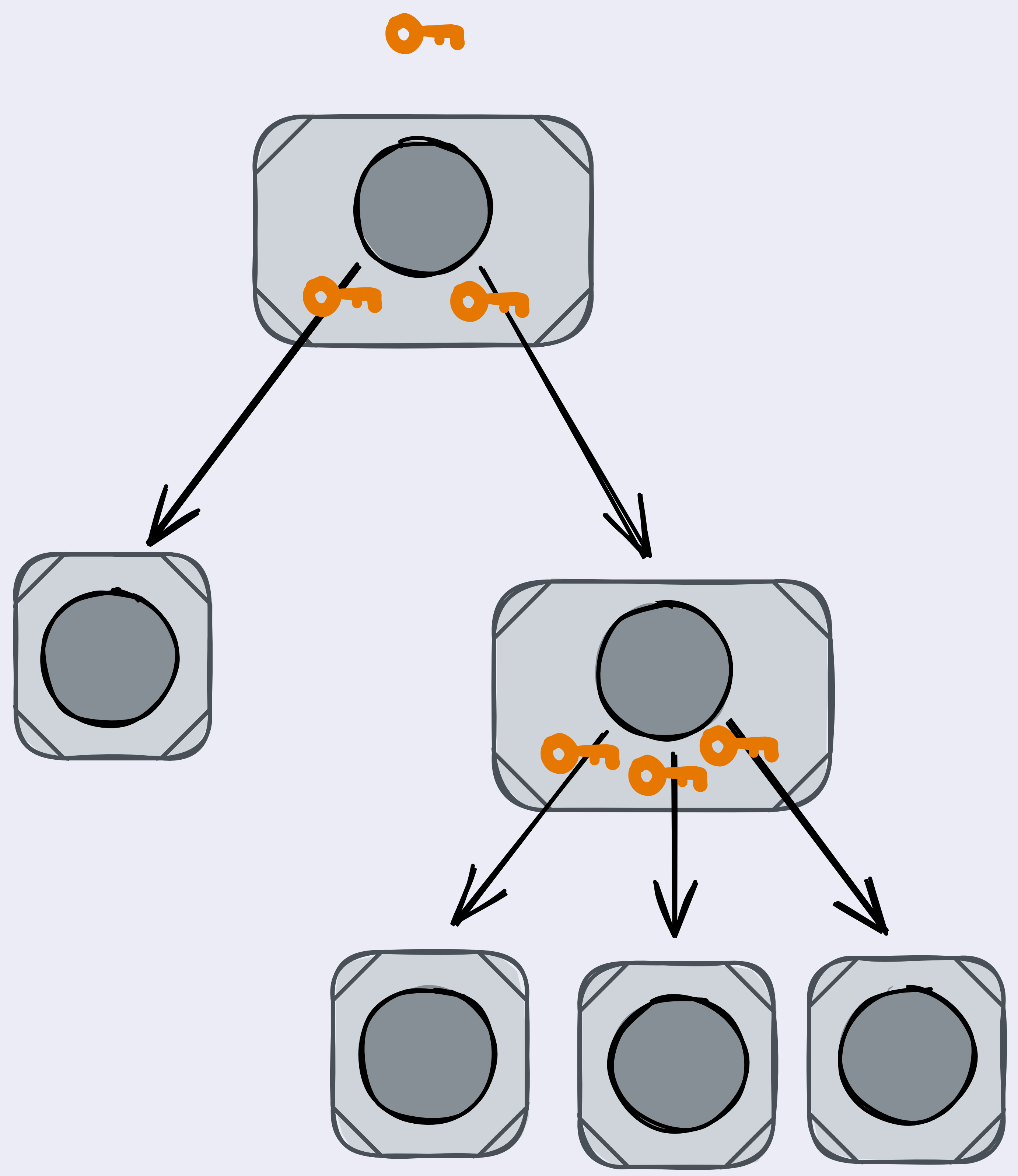
Private WNFS

- Based on cryptrees
- Encrypt all directories/files with symmetric encryption
- Include keys to decrypt children
- → A key gives access to its node & all children



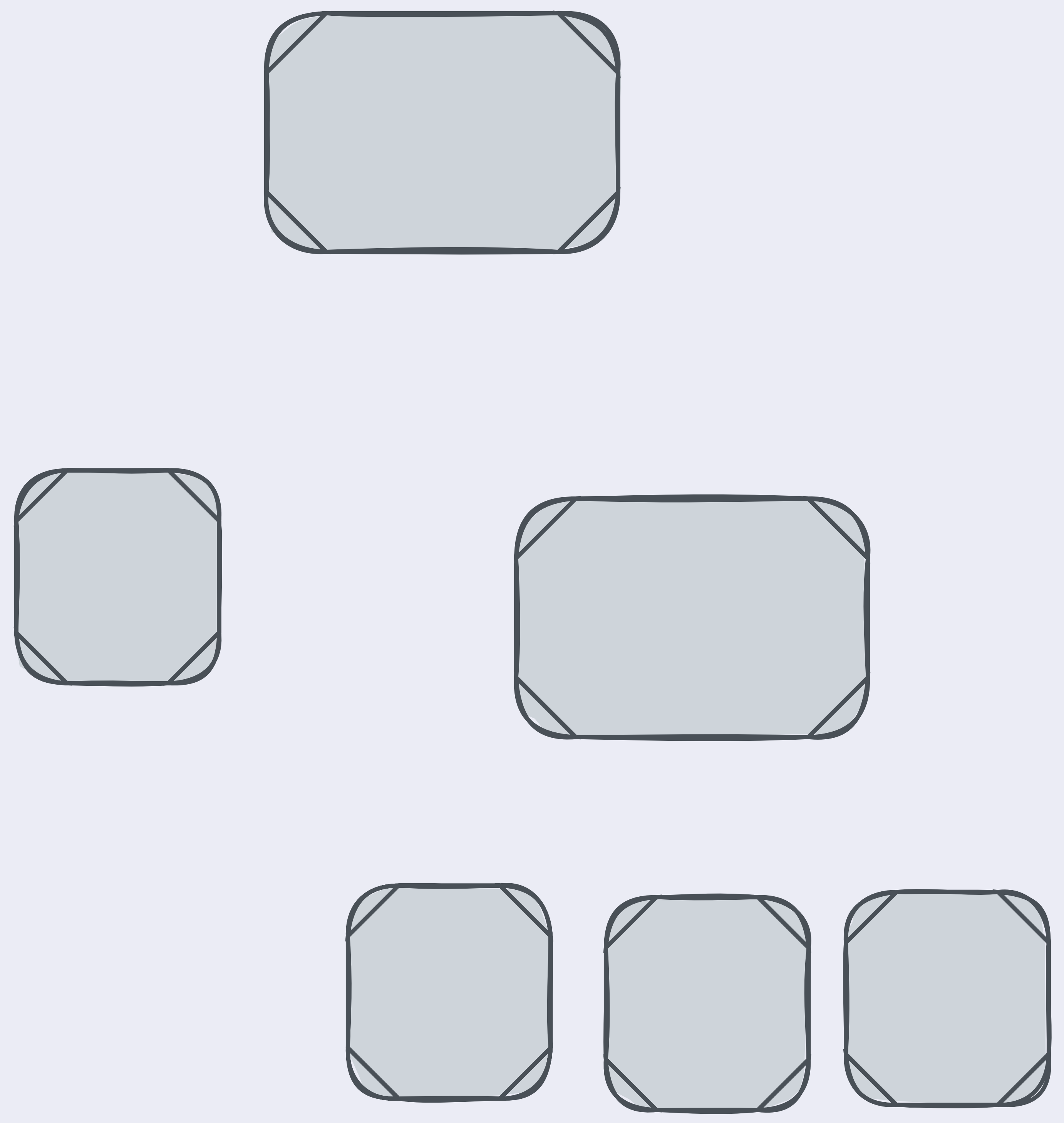
Private WNFS

- Links between nodes encrypted



Private WNFS

- Links between nodes encrypted

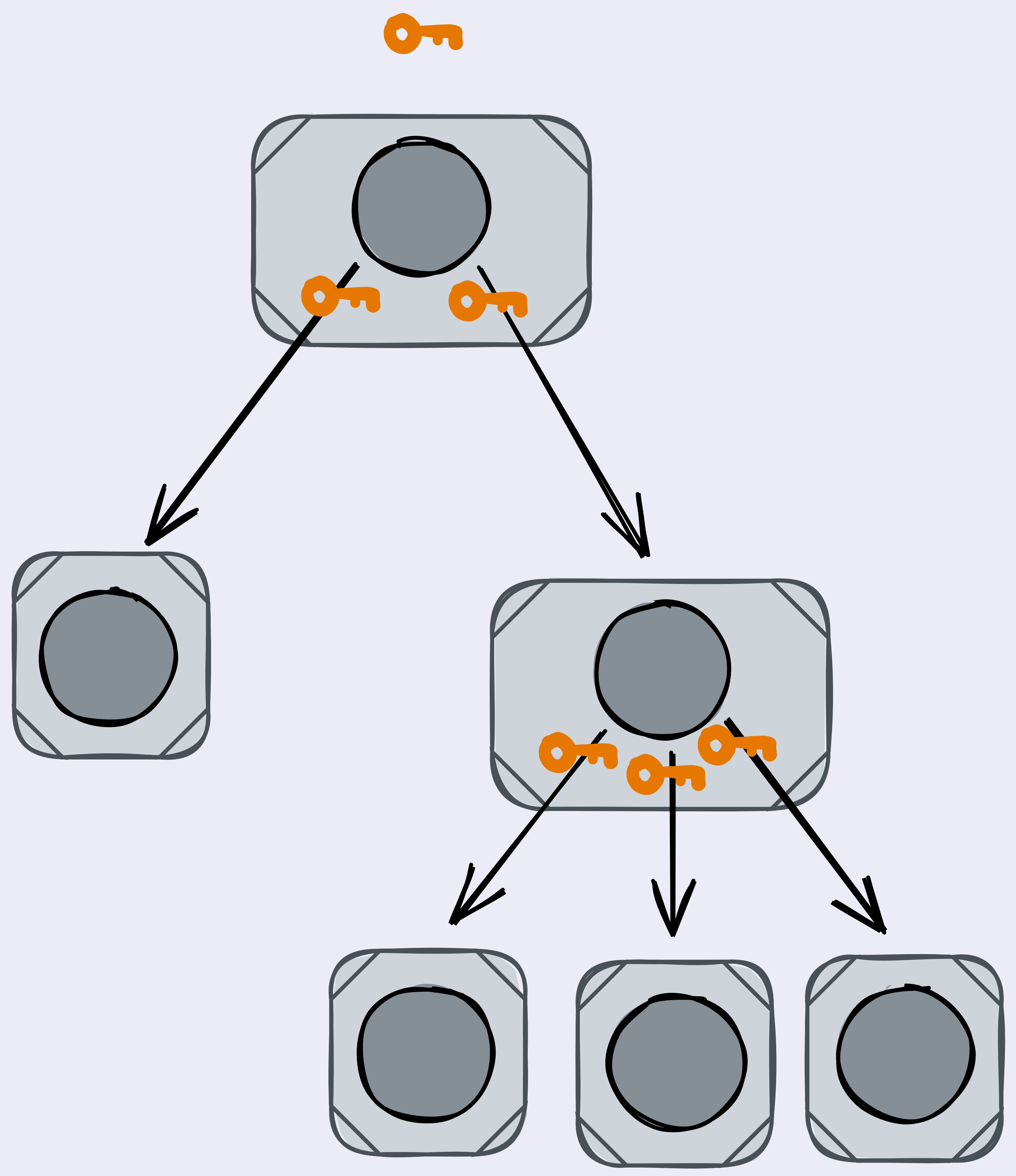


Private WNFS

- Links between nodes encrypted

  + Not leaking metadata

  – Can't walk tree (e.g. for pinning)



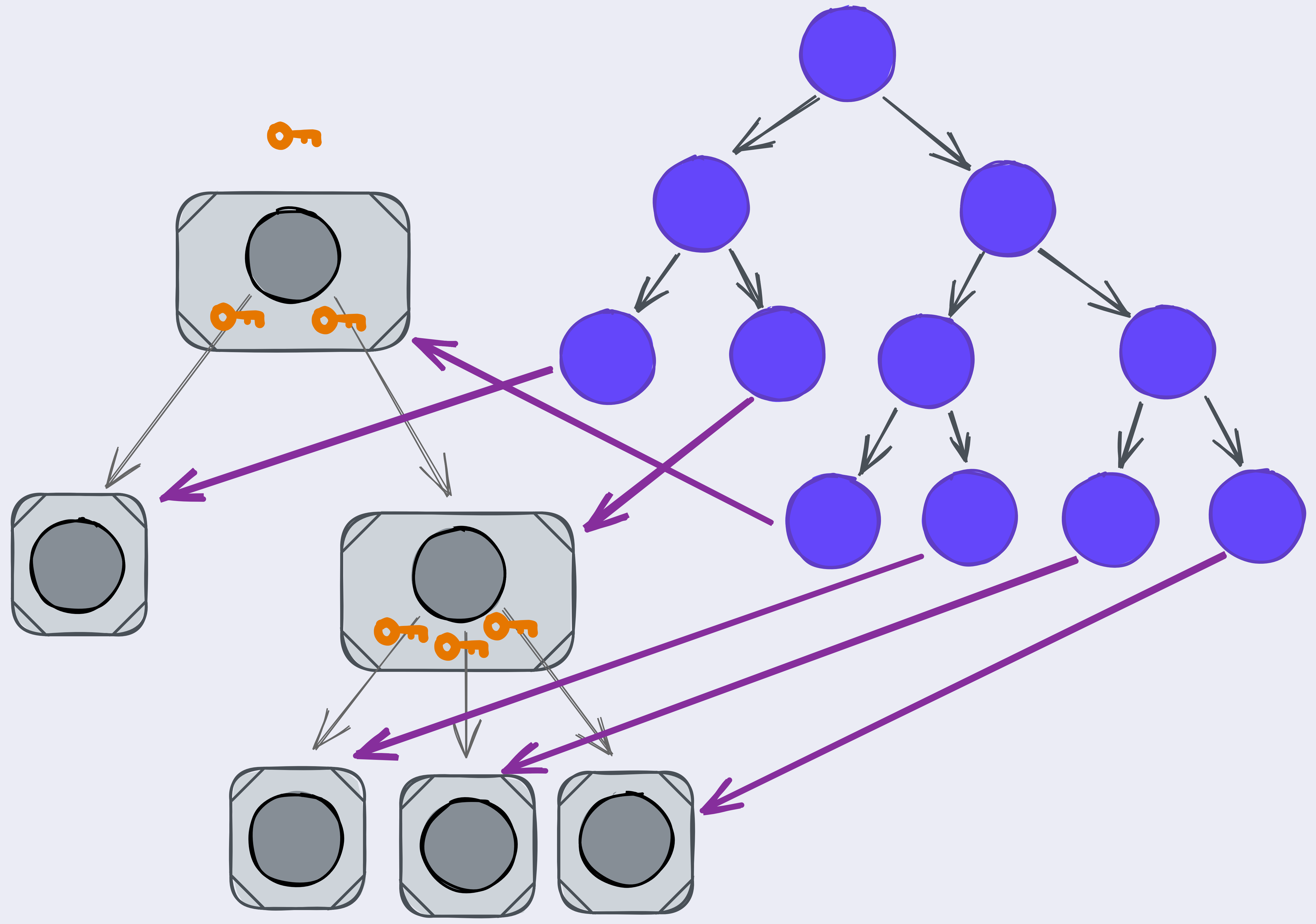
Private WNFS

- Links between nodes encrypted

  + Not leaking metadata

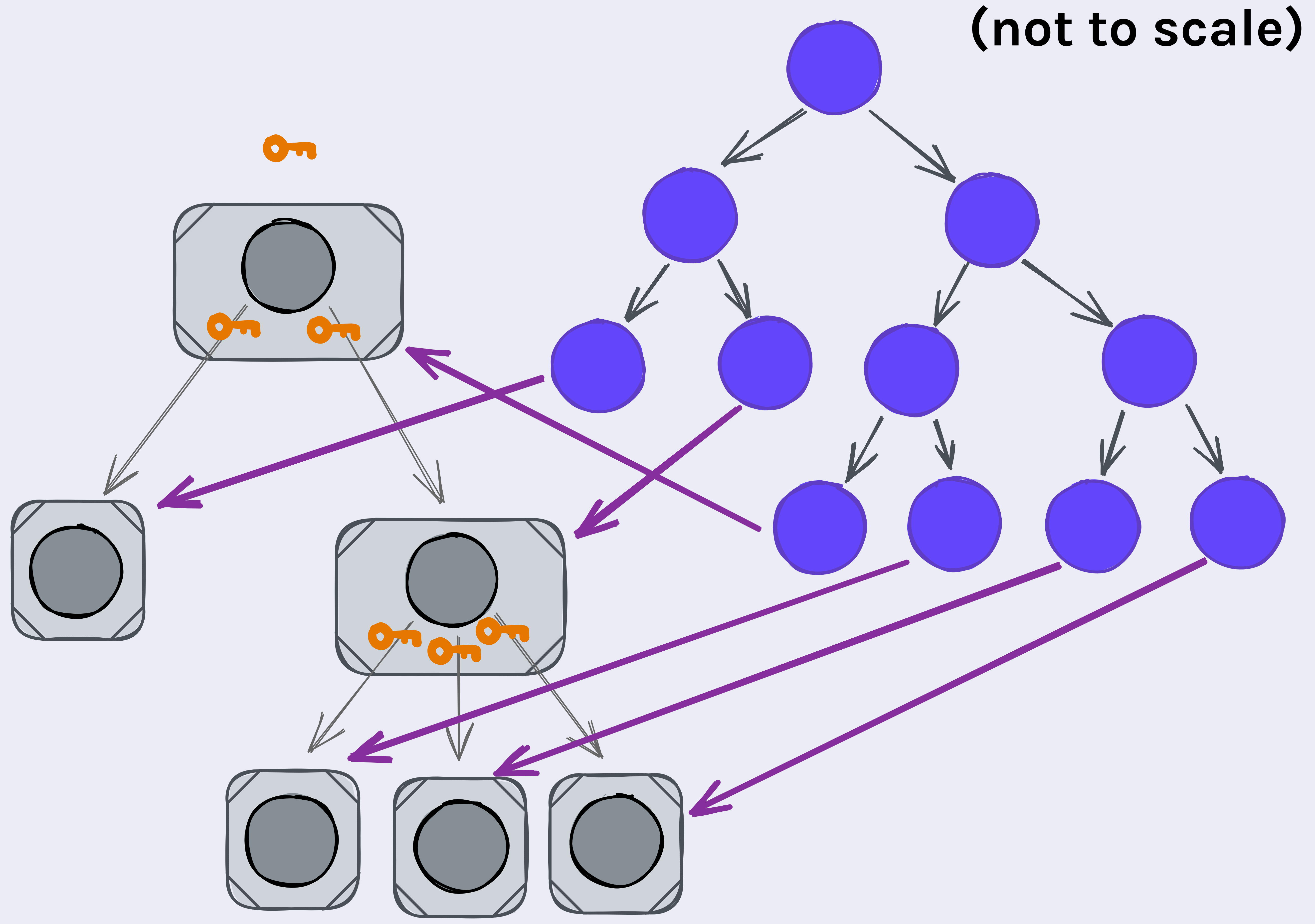
  - Can’t walk tree (e.g. for pinning)

- → Collect nodes in HAMT



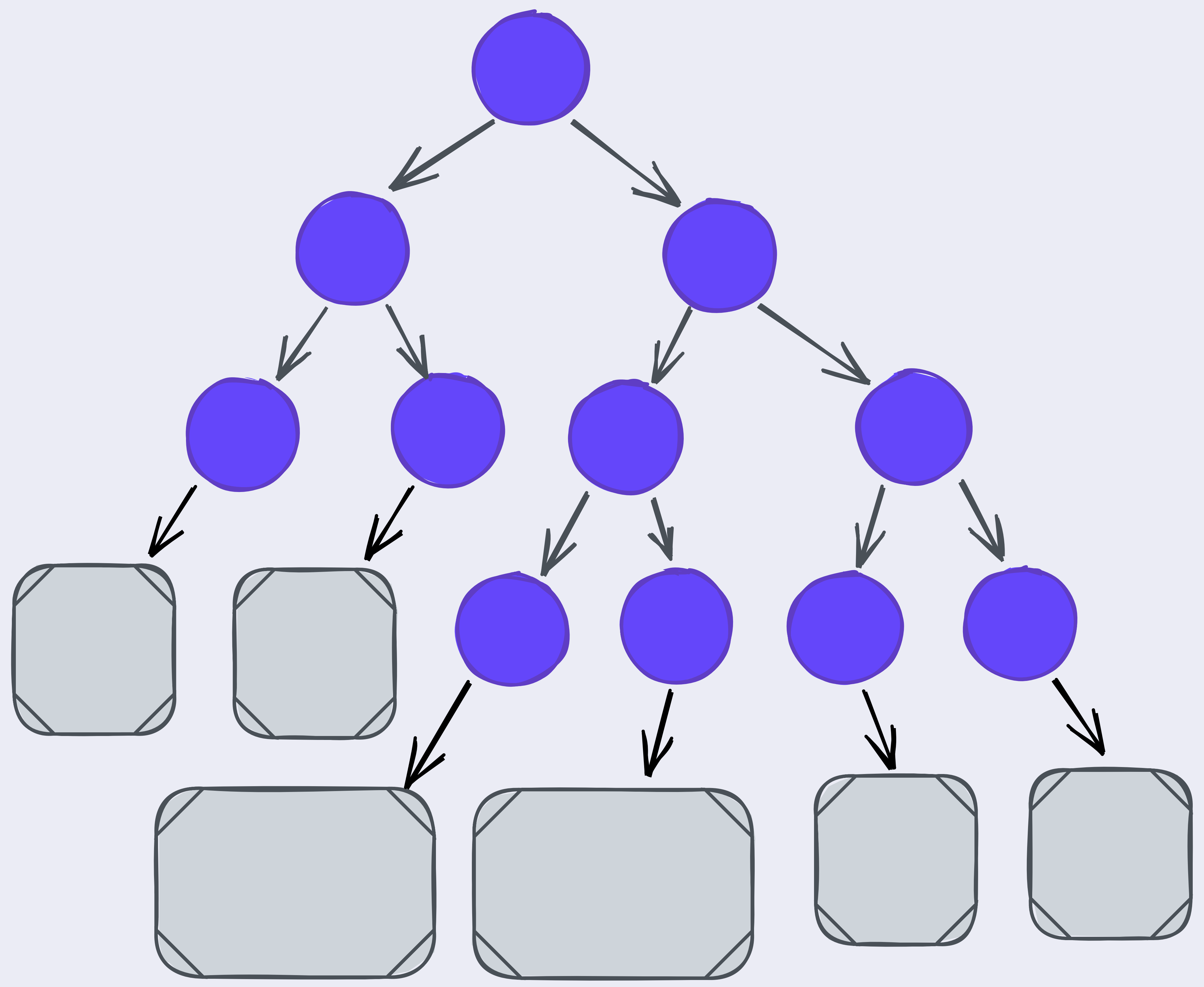
Private WNFS

- Links between nodes encrypted
  - + Not leaking metadata
  - – Can’t walk tree (e.g. for pinning)
- → Collect nodes in HAMT



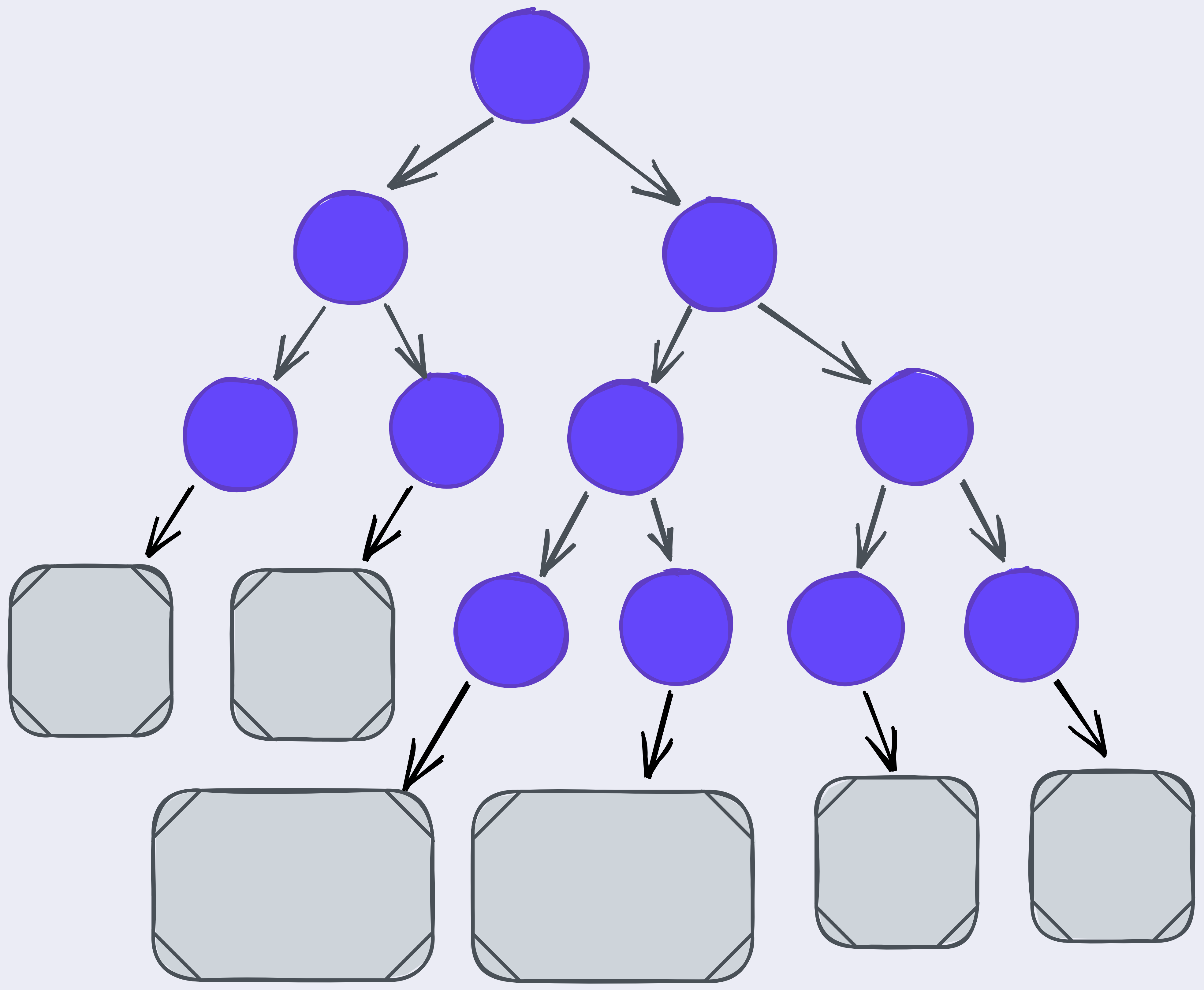
WNFS HAMT

- Essentially a huge hash map
- Efficient encoding in immutable contexts by being a balanced tree



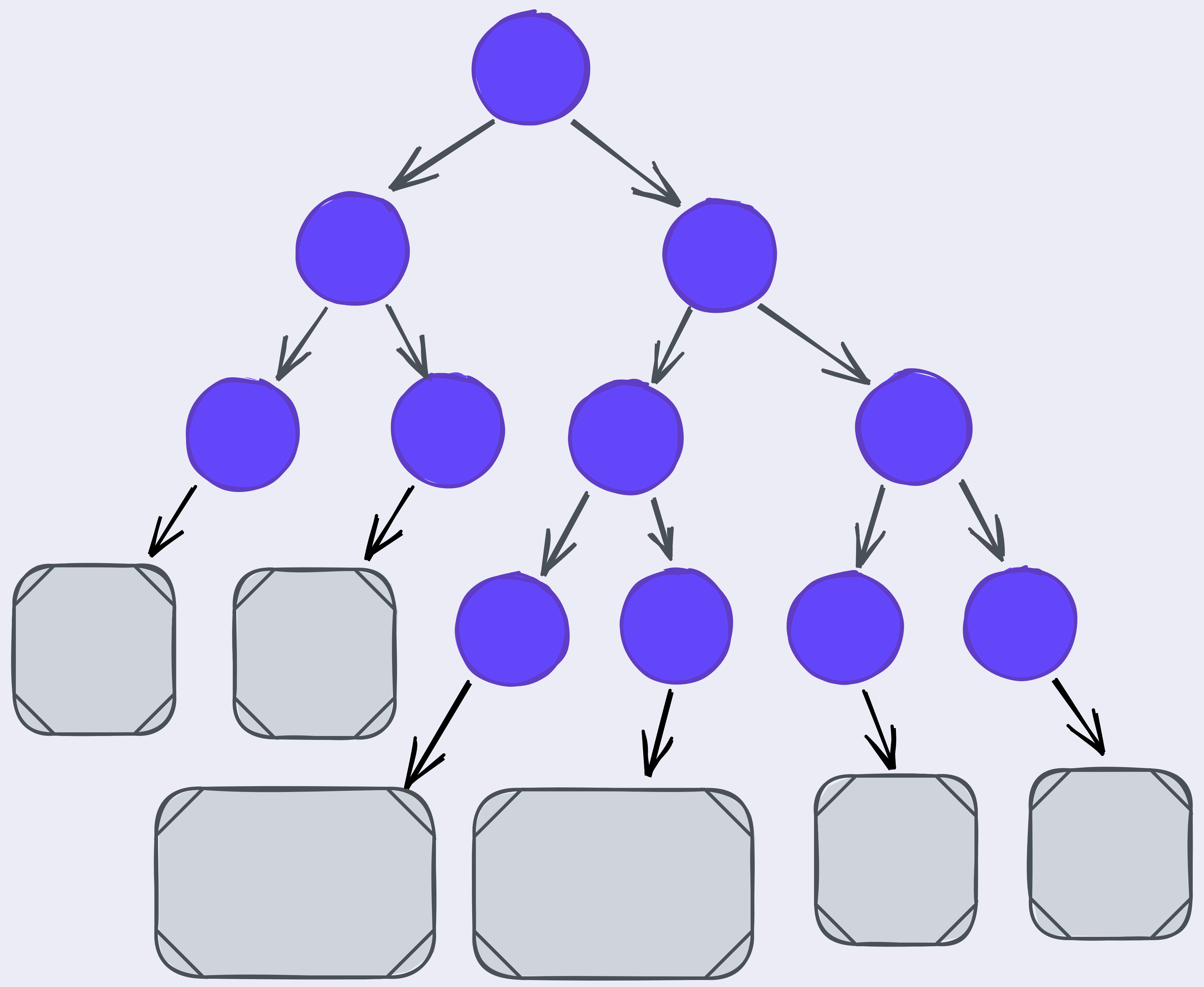
WNFS HAMT

- Essentially a huge hash map
- Efficient encoding in immutable contexts by being a balanced tree
- This is what a third party sees



WNFS HAMT

- Essentially a huge hash map
- Efficient encoding in immutable contexts by being a balanced tree
- This is what a third party sees
  → Hides directory structure



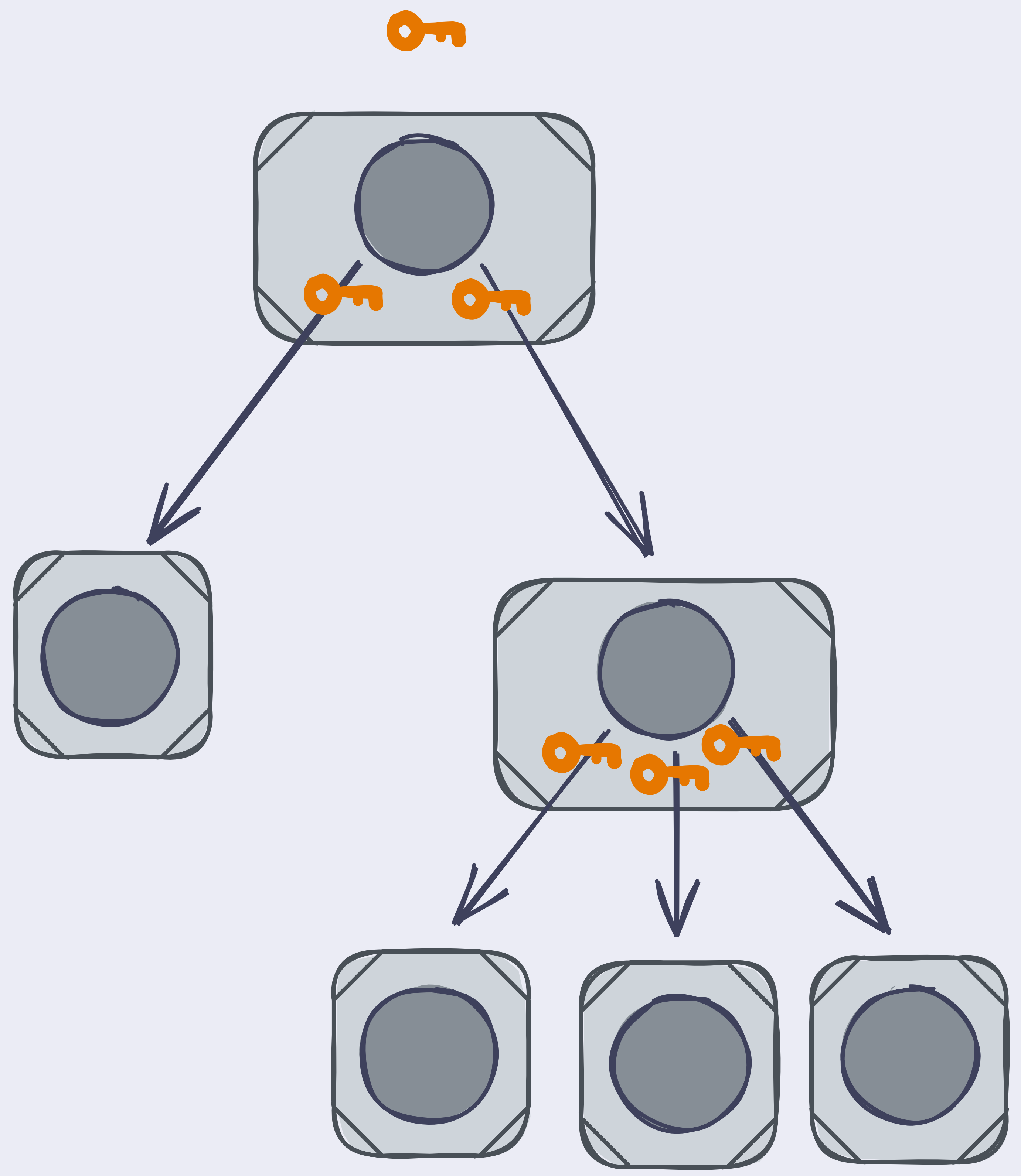
Private WNFS: Write Access

- Goals
  - Write access to a directory gives write access to subdirectories

Private WNFS: Write Access

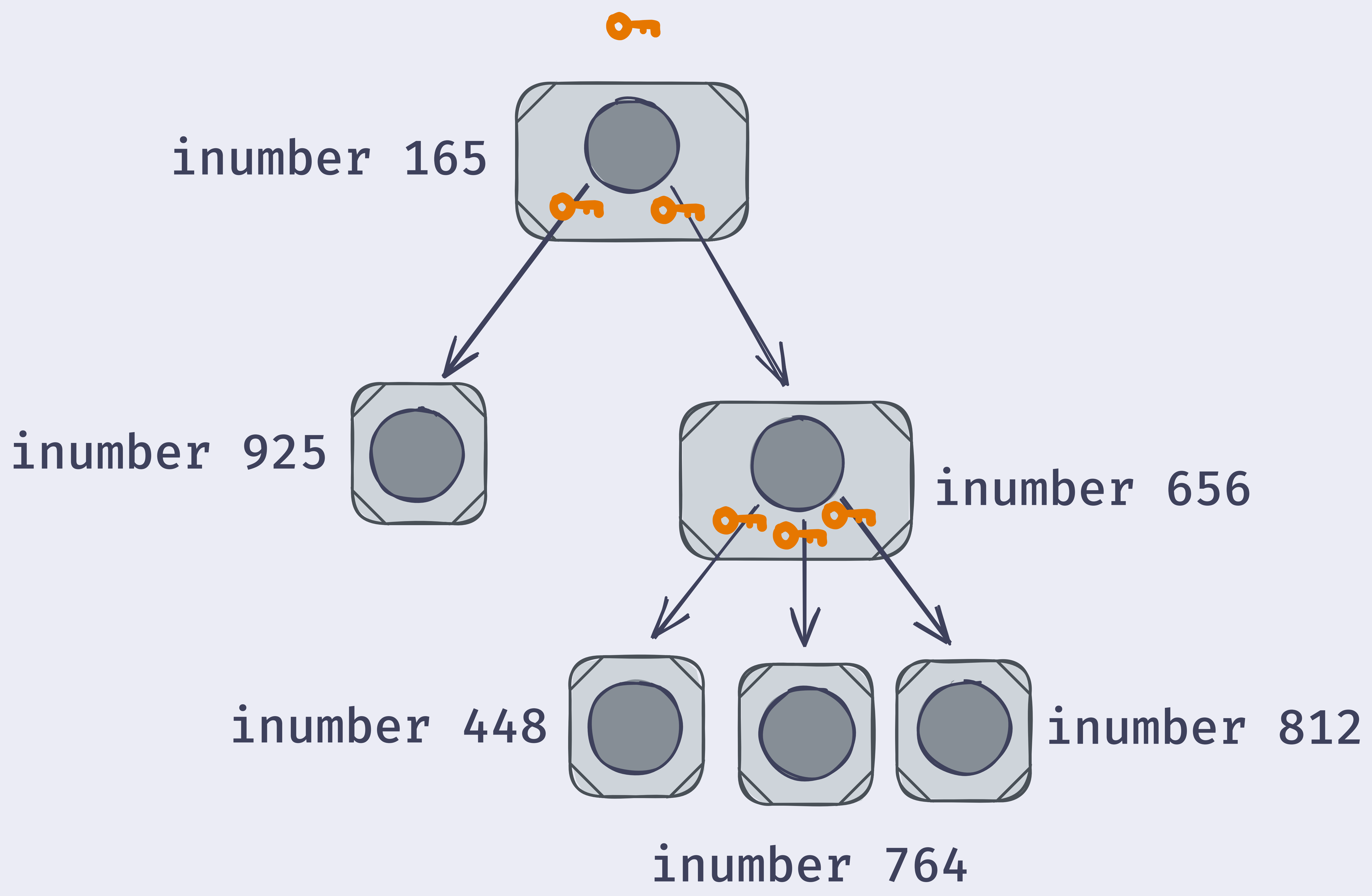
- Goals
  - Write access to a directory gives write access to subdirectories
  - Verifying write access doesn’t require read access

Private WNFS: Write Access



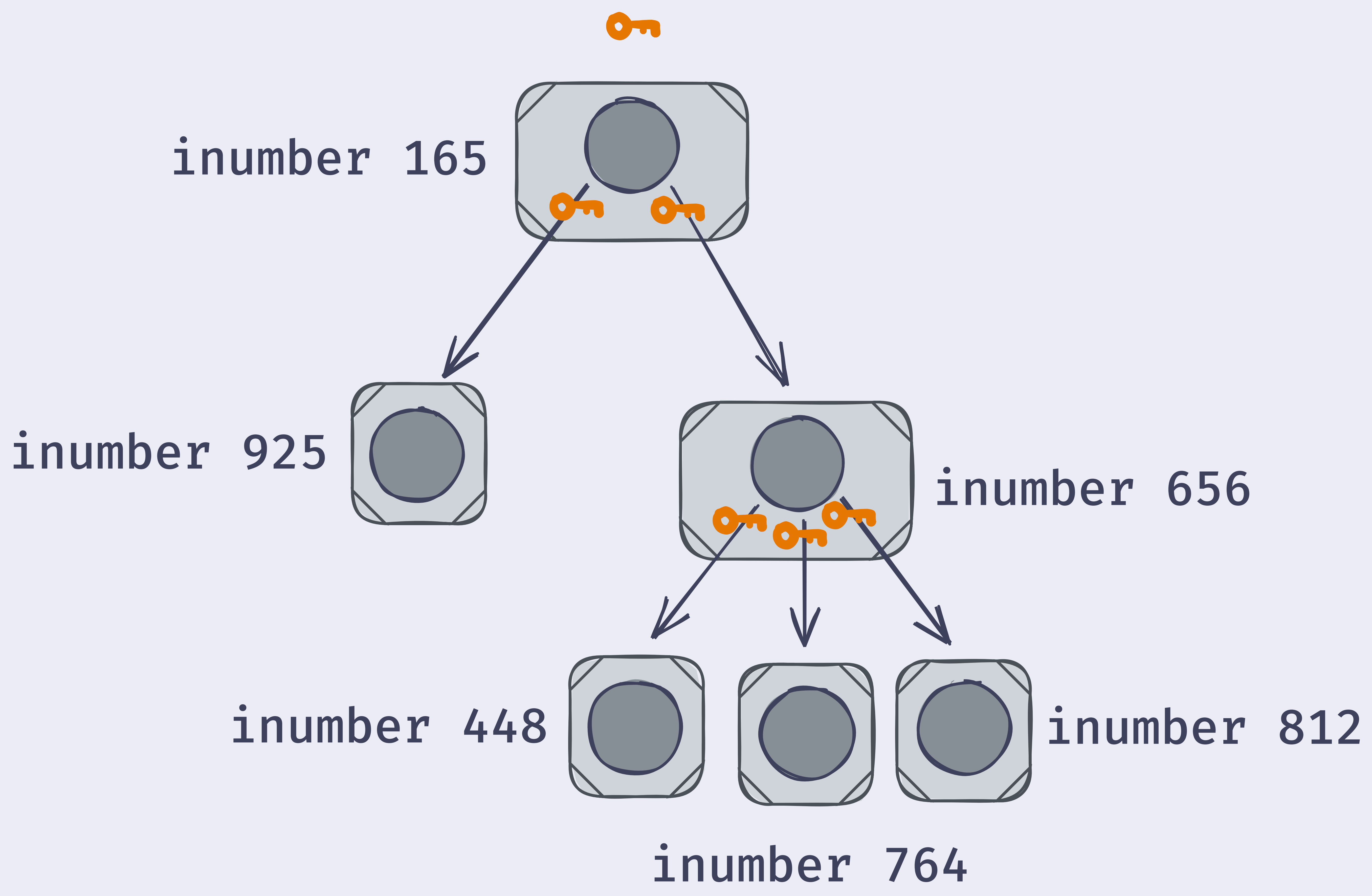
Private WNFS: Write Access

- Associate an “inumber” with each private node



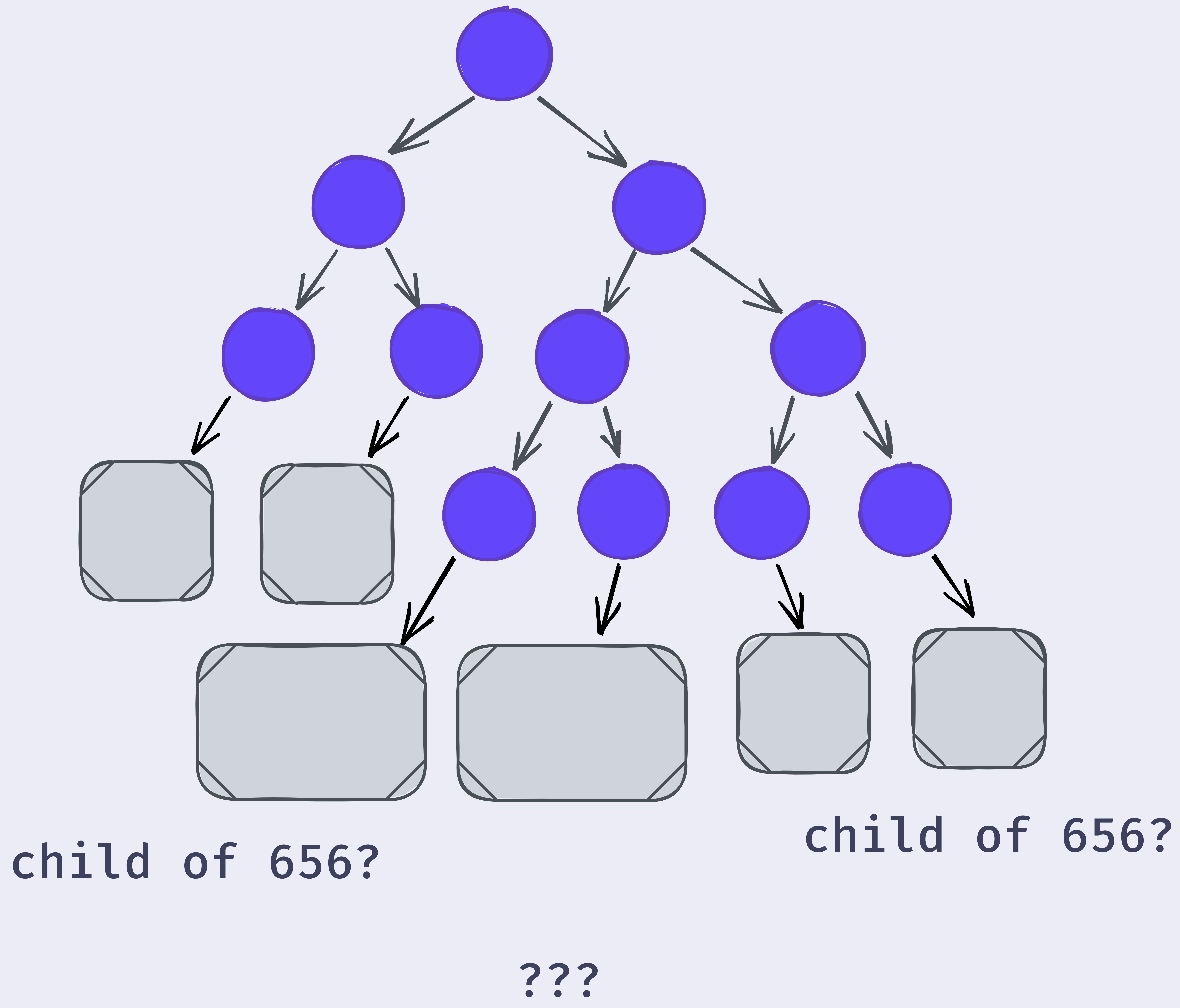
Private WNFS: Write Access

- Associate an “inumber” with each private node
- inumbers identify what subset of nodes you have access to



Private WNFS: Write Access

How does a third party know whether a value is a subdirectory of an inumber?



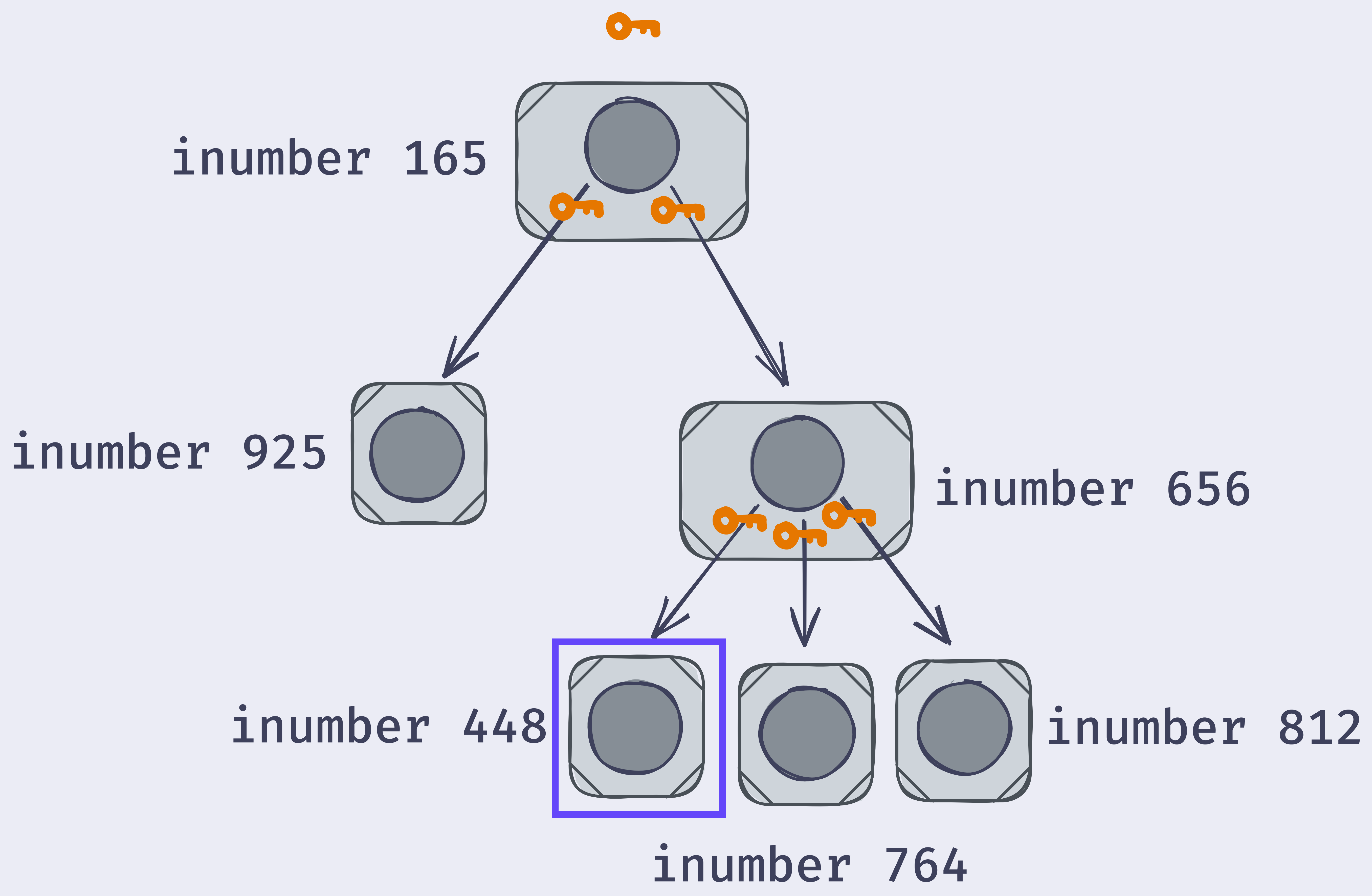
🎉 Cryptographic Accumulators 🎉

Short Intro: Cryptographic Accumulators

- “Like a set of values”
- Given only the accumulator, can’t derive what’s inside
- Given a x, anyone can compute whether x is in the accumulator
- In WNFS: Symmetric (Nyberg) accumulators

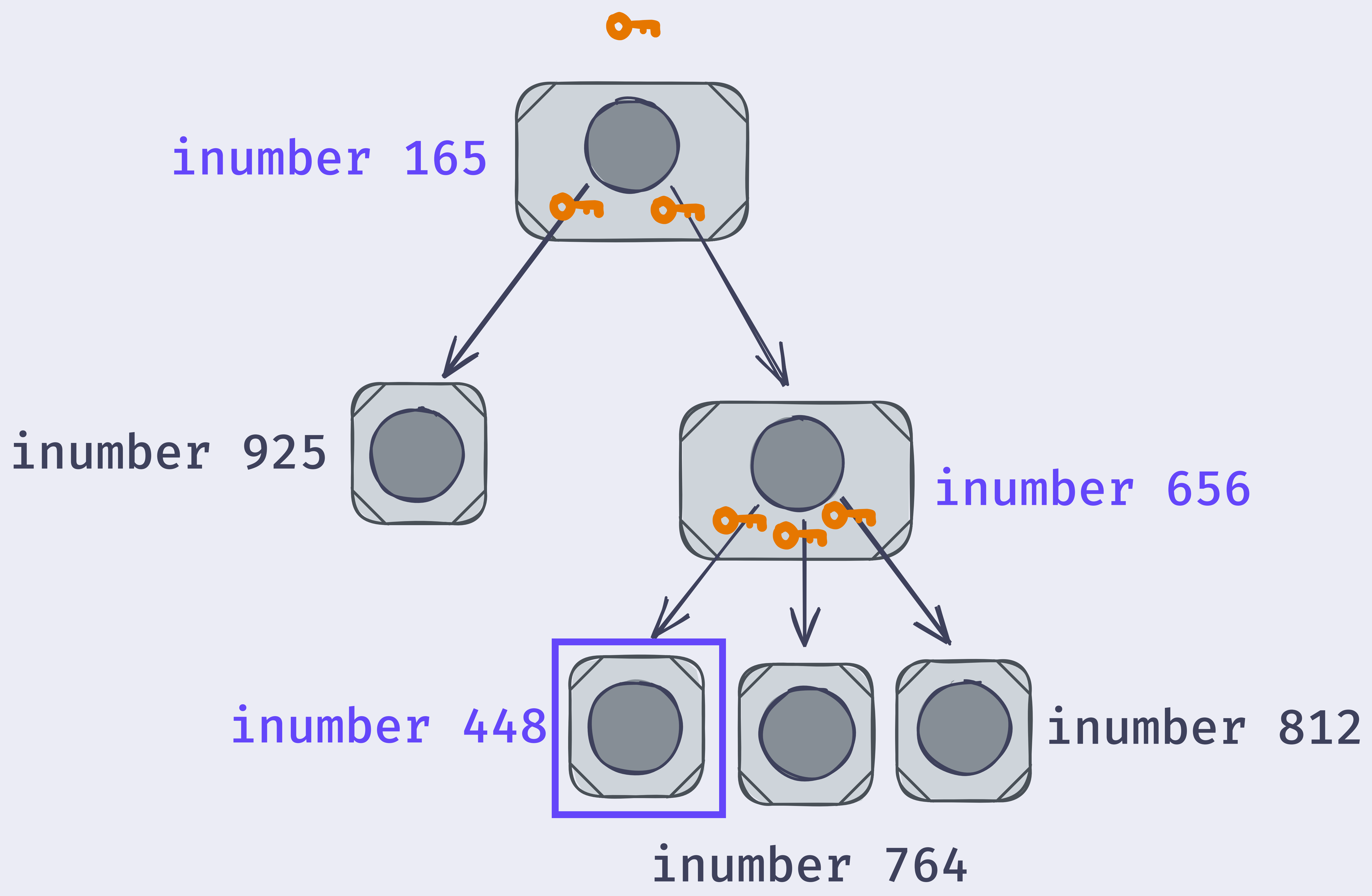
Namefilters

- Private wnfs nodes are referred to by their “namefilter”



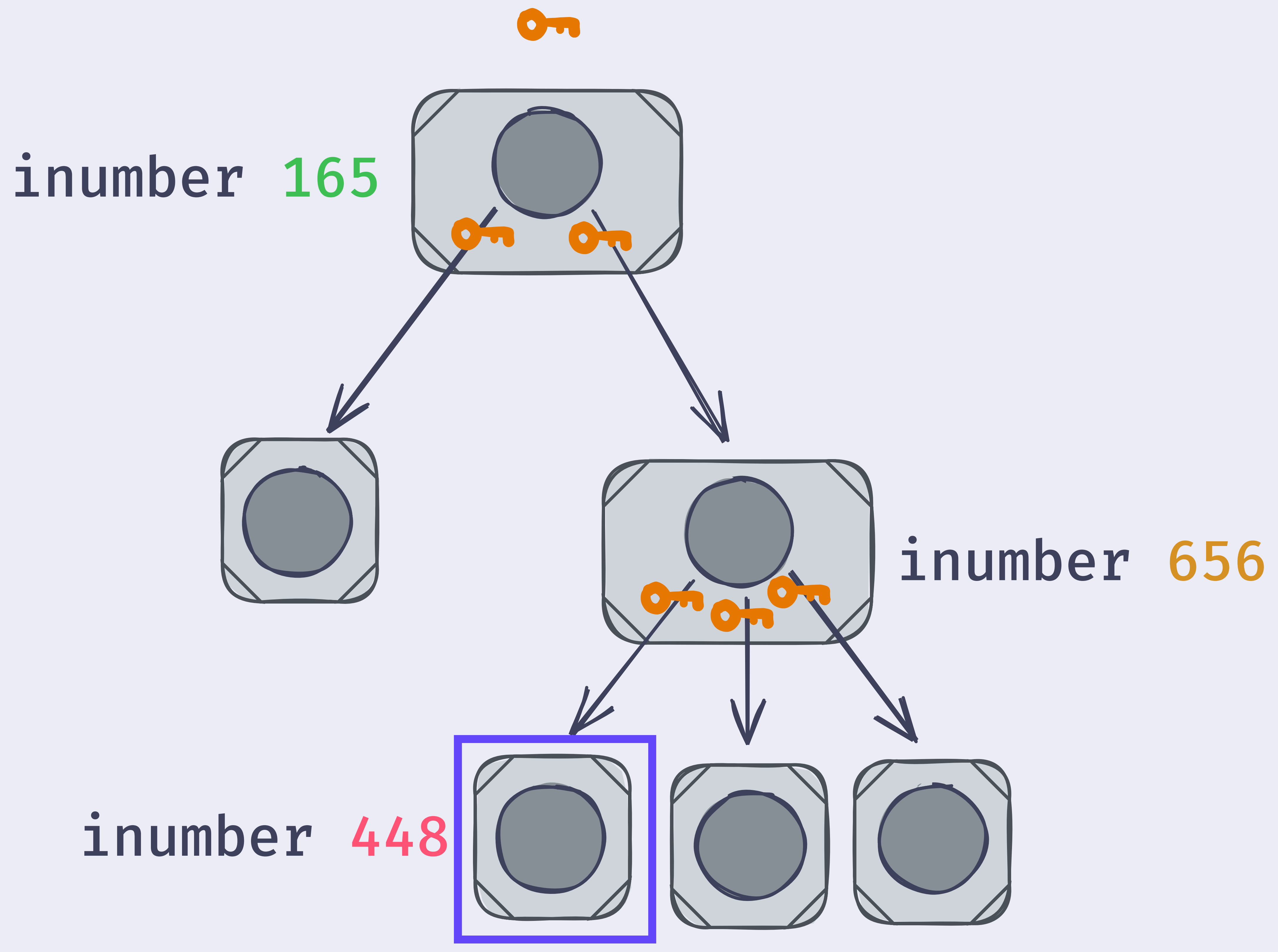
Namefilters

- Private wnfs nodes are referred to by their “namefilter”
- Their namefilter is a cryptographic accumulator of:
  - The “inumber”s a block’s spine



Namefilters

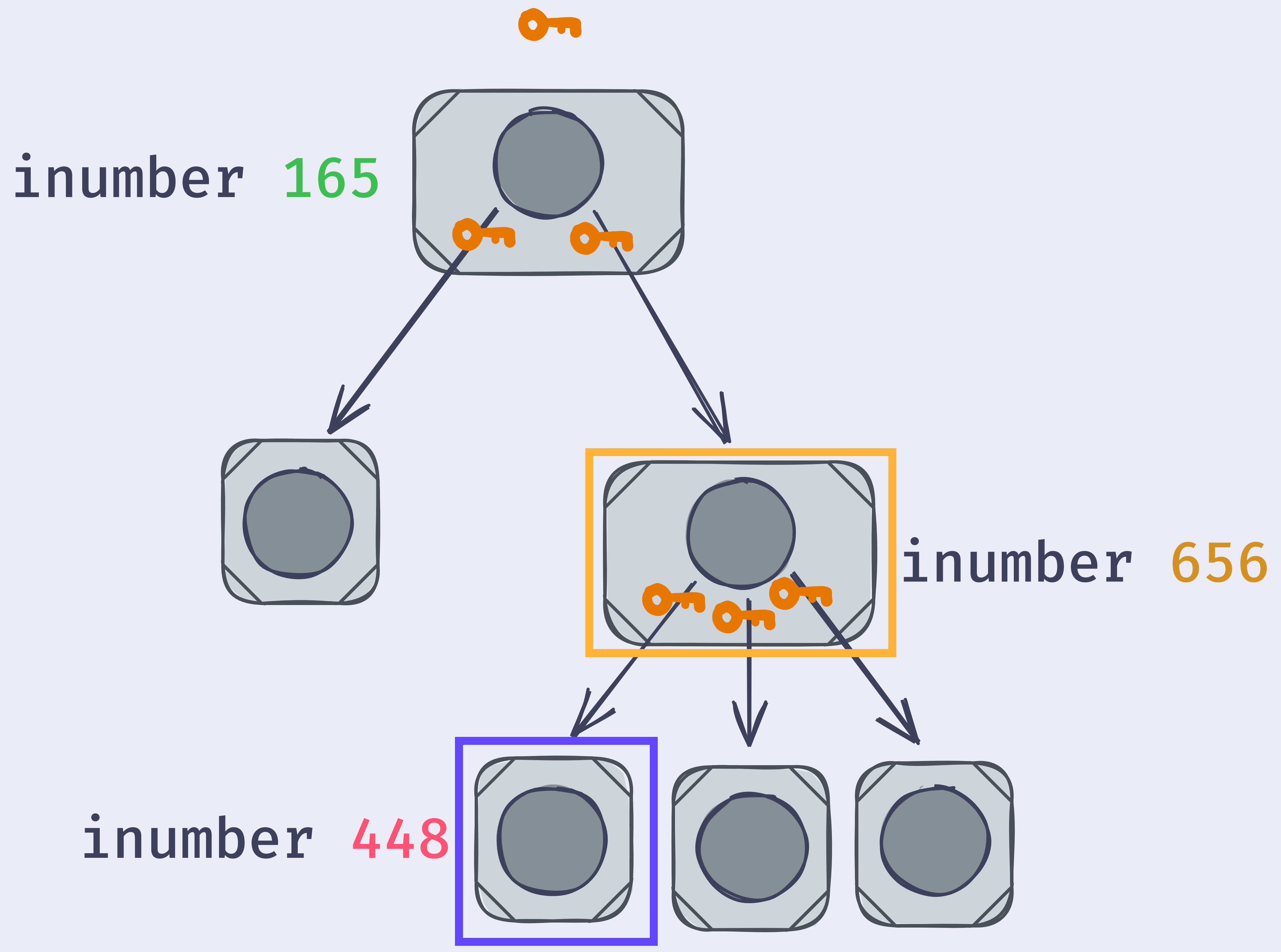
- Private wnfs nodes are referred to by their “namefilter”
- Their namefilter is a cryptographic accumulator of:
  - The “inumber”s a block’s spine
  - The block’s revision



```
namefilter = accumulate(165,656,448,<revision>)
```

Namefilters

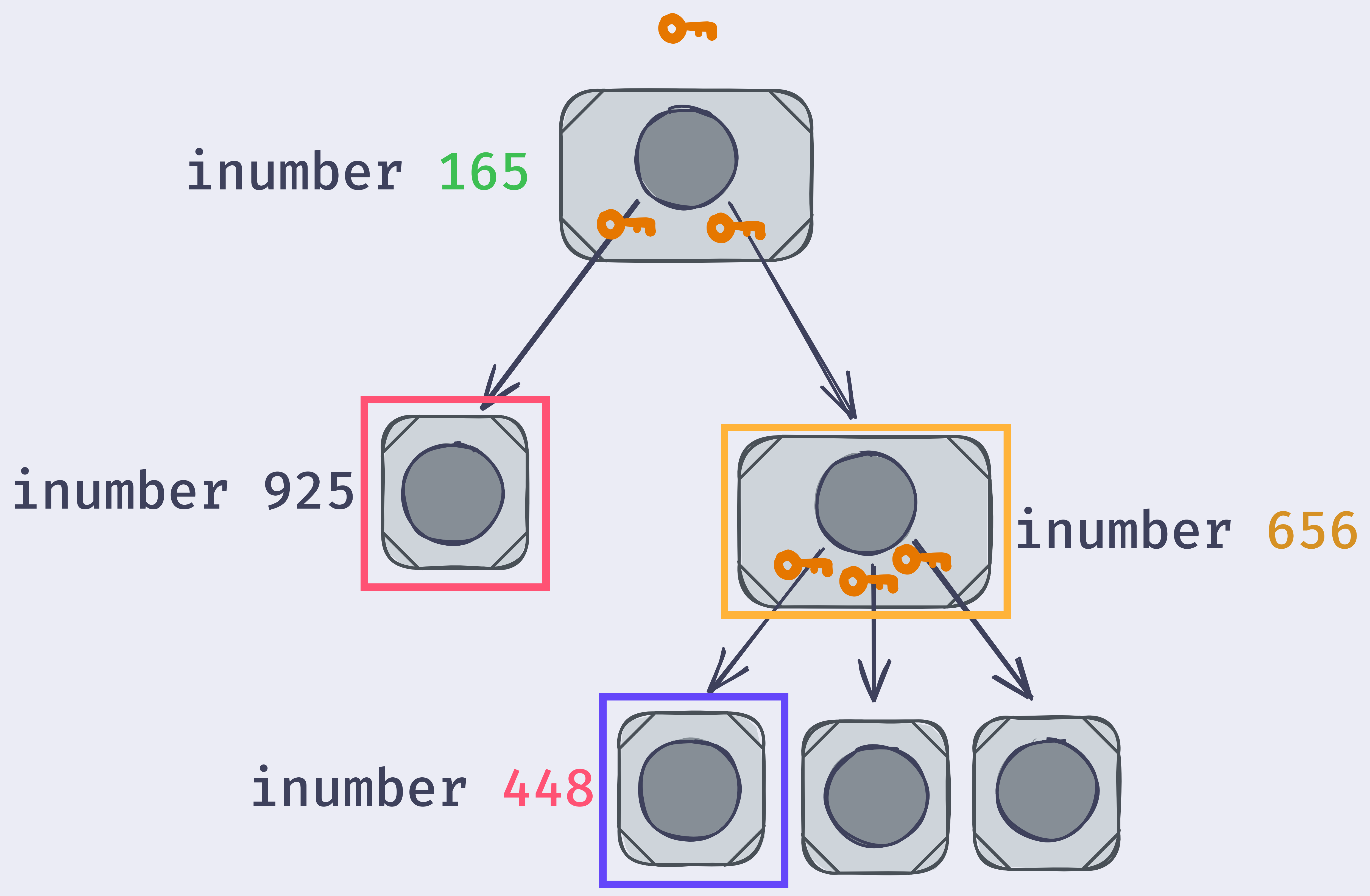
- Private wnfs nodes are referred to by their “namefilter”
- Their namefilter is a cryptographic accumulator of:
  - The “inumber”s a block’s spine
  - The block’s revision



```
namefilter = accumulate(165,656,448,<revision>)
namefilter = accumulate(165,656,<revision>)
```

Namefilters

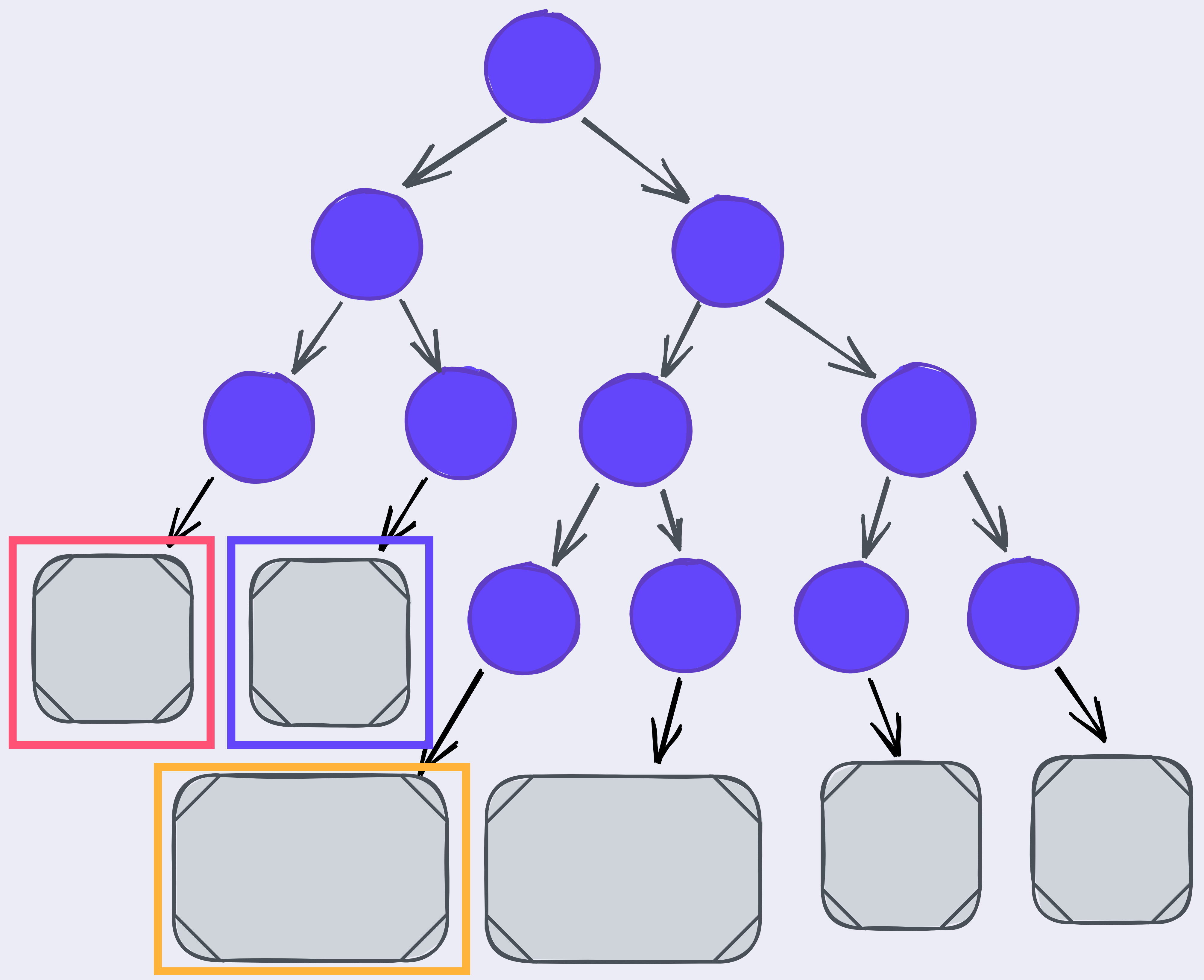
- Private wnfs nodes are referred to by their “namefilter”
- Their namefilter is a cryptographic accumulator of:
  - The “inumber”s a block’s spine
  - The block’s revision



```
namefilter = accumulate(165,656,448,<revision>)
namefilter = accumulate(165,656,<revision>)
namefilter = accumulate(165,925,<revision>)
```

WNFS HAMT

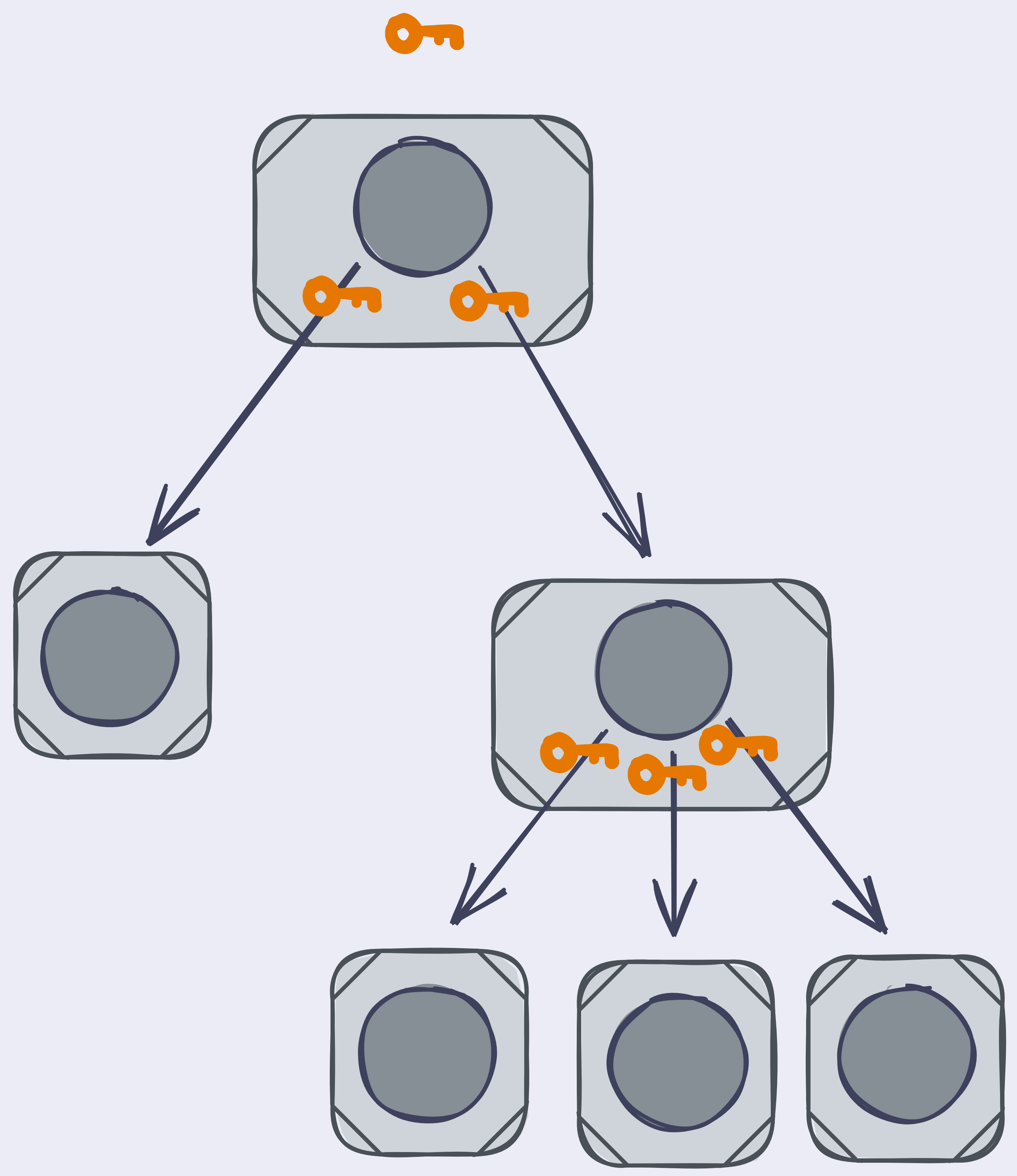
- A private block’s key is its namefilter
- Given an inumber, a third party can compute the set of nodes that are children



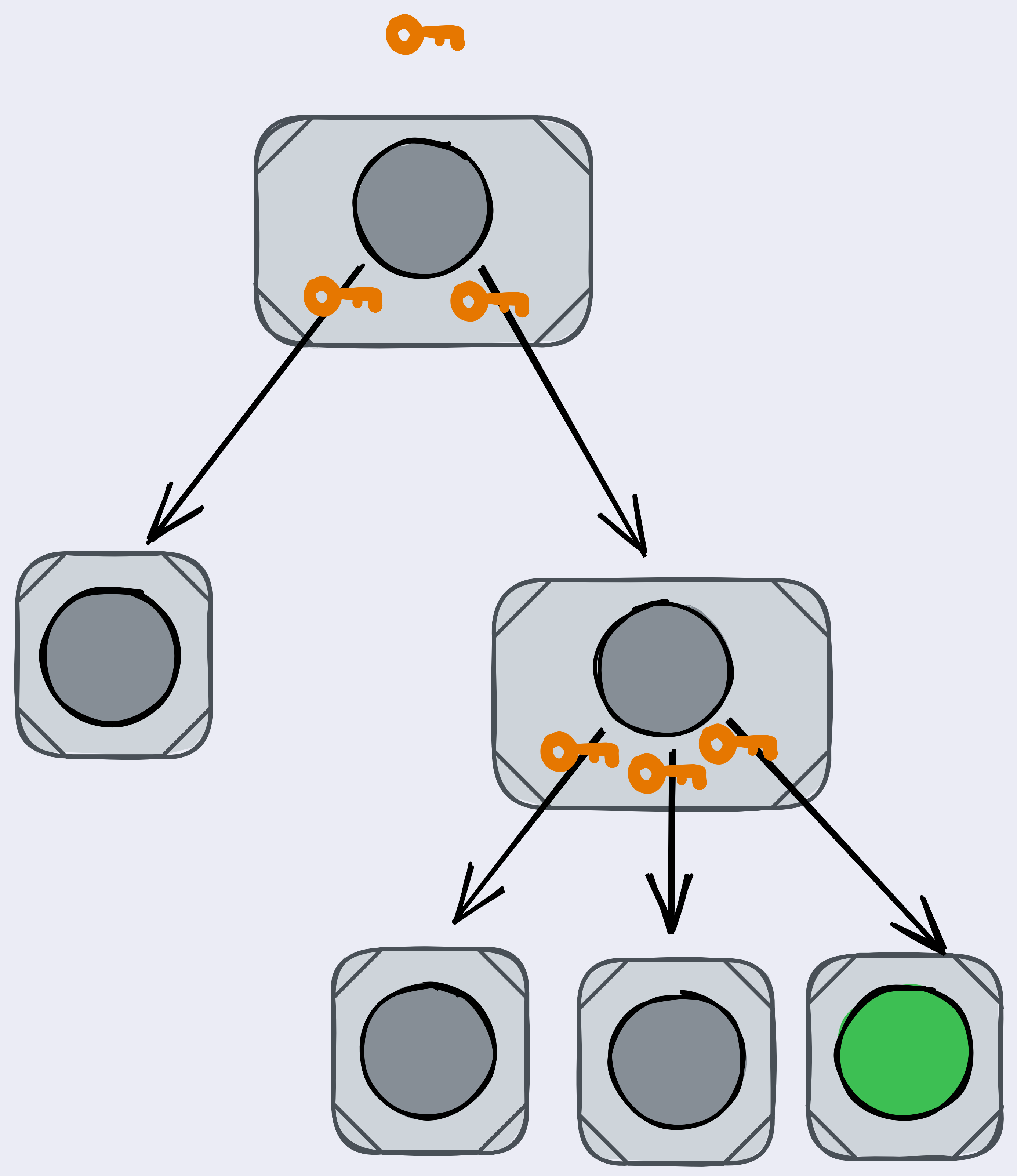
```
namefilter = accumulate(165,656,448,<revision>)
namefilter = accumulate(165,656,<revision>)
namefilter = accumulate(165,925,<revision>)
```

Private WNFS: Versioning

Private WNFS: Versioning

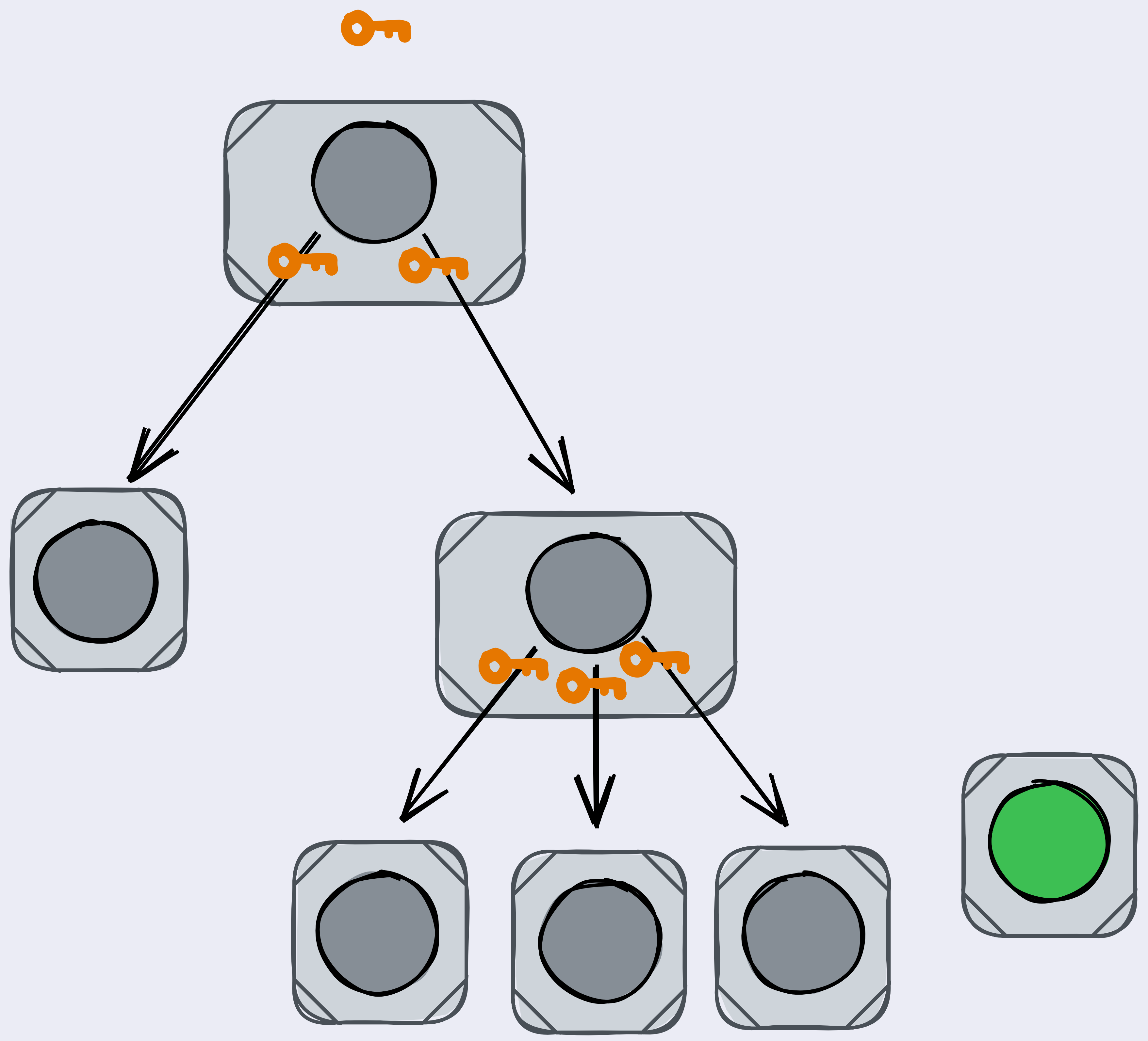


Private WNFS: Versioning



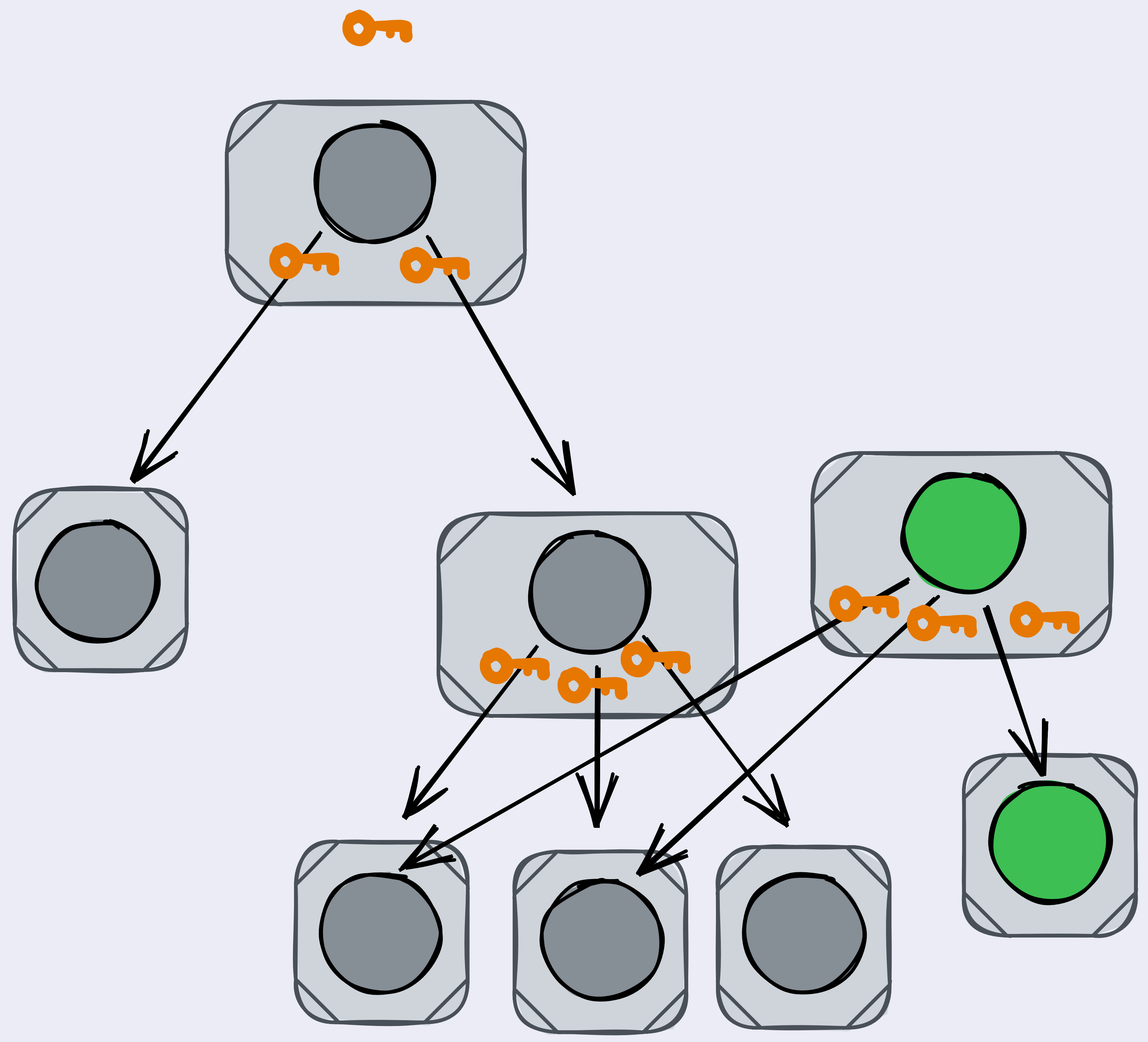
Private WNFS: Versioning

- Copy-on-write to preserve history



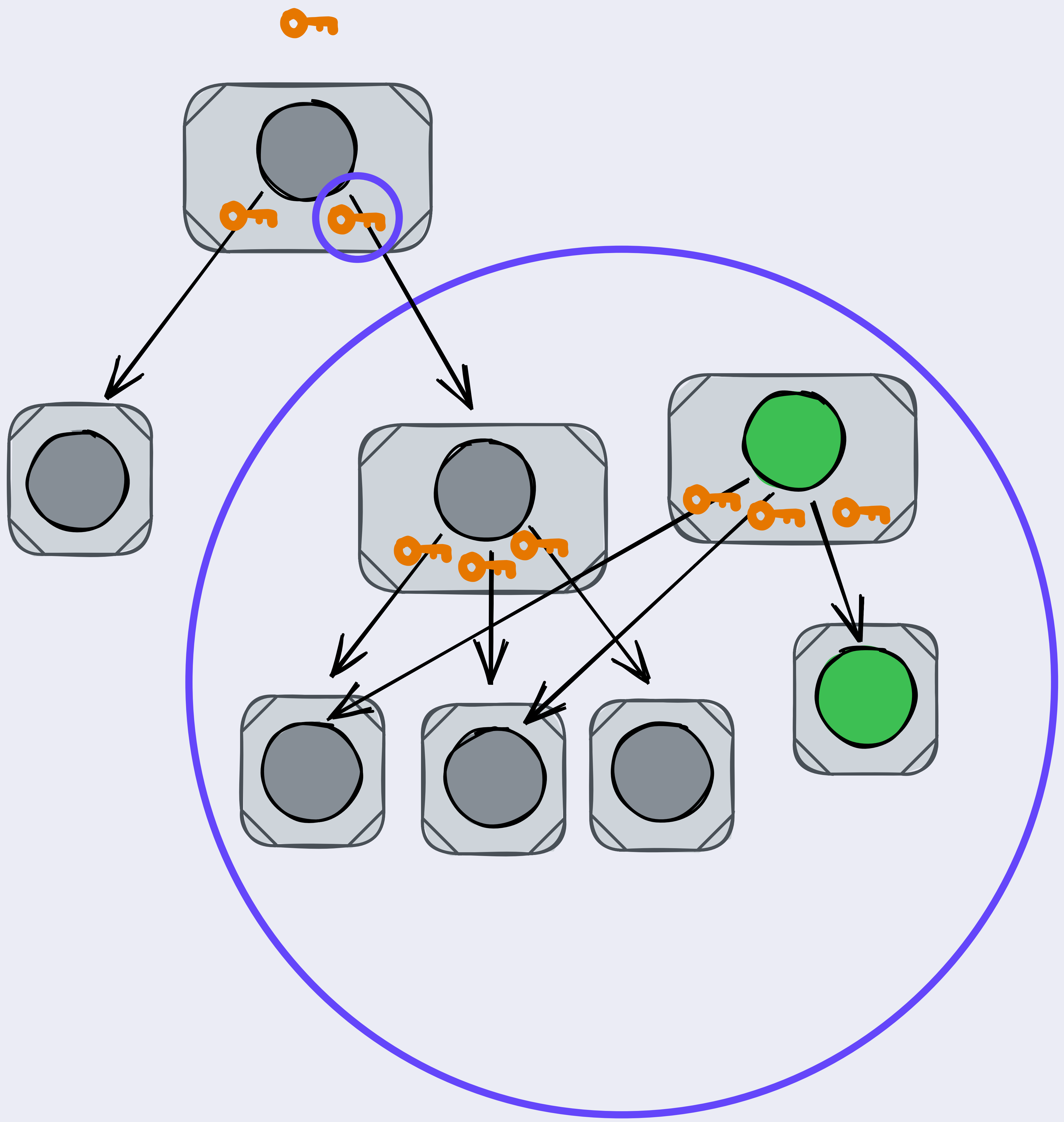
Private WNFS: Versioning

- Copy-on-write to preserve history
- Fix links along the path from the root



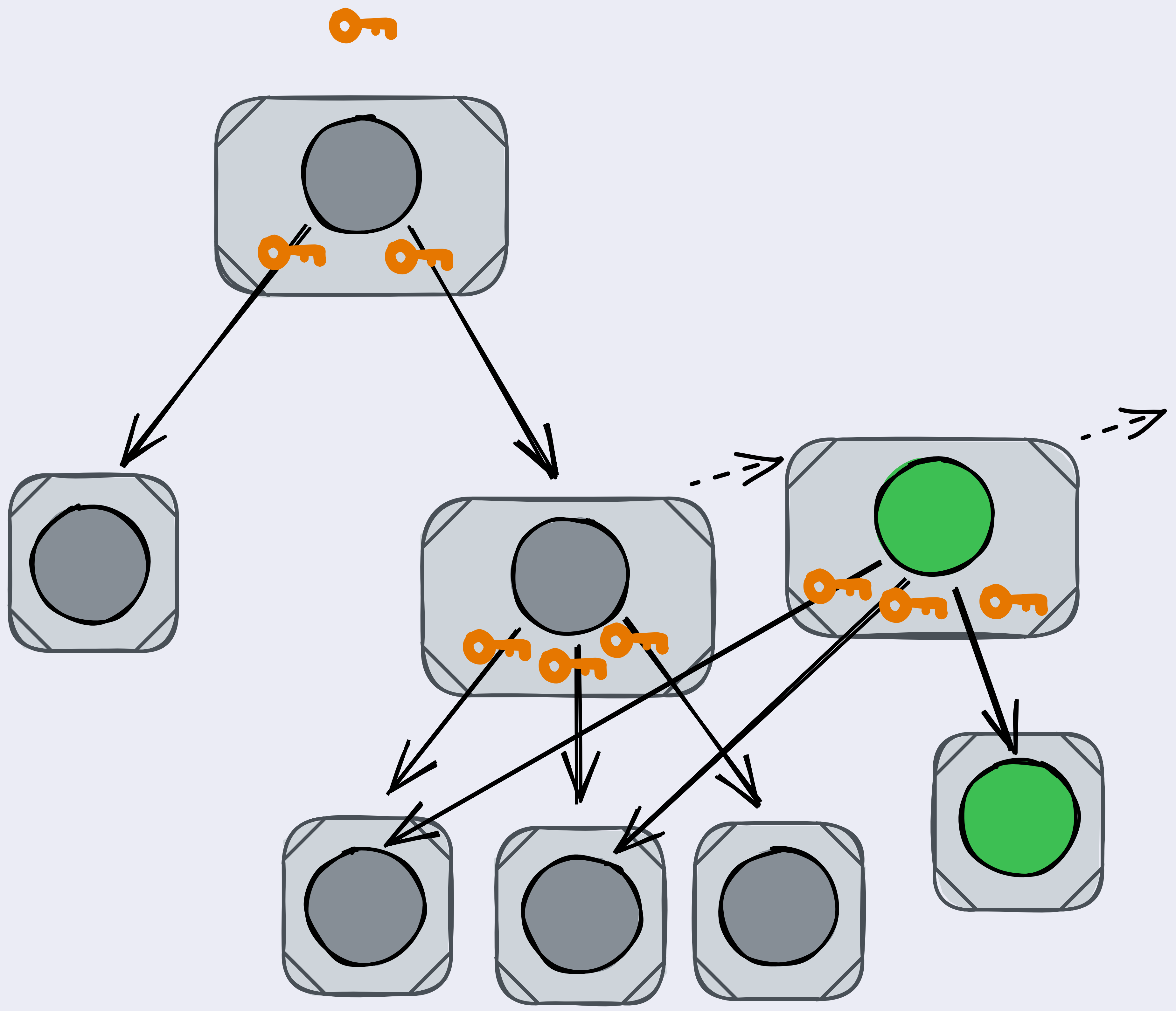
Private WNFS: Versioning

- Copy-on-write to preserve history
- Fix links along the path from the root
- Problem: Clients might only have access to a subtree



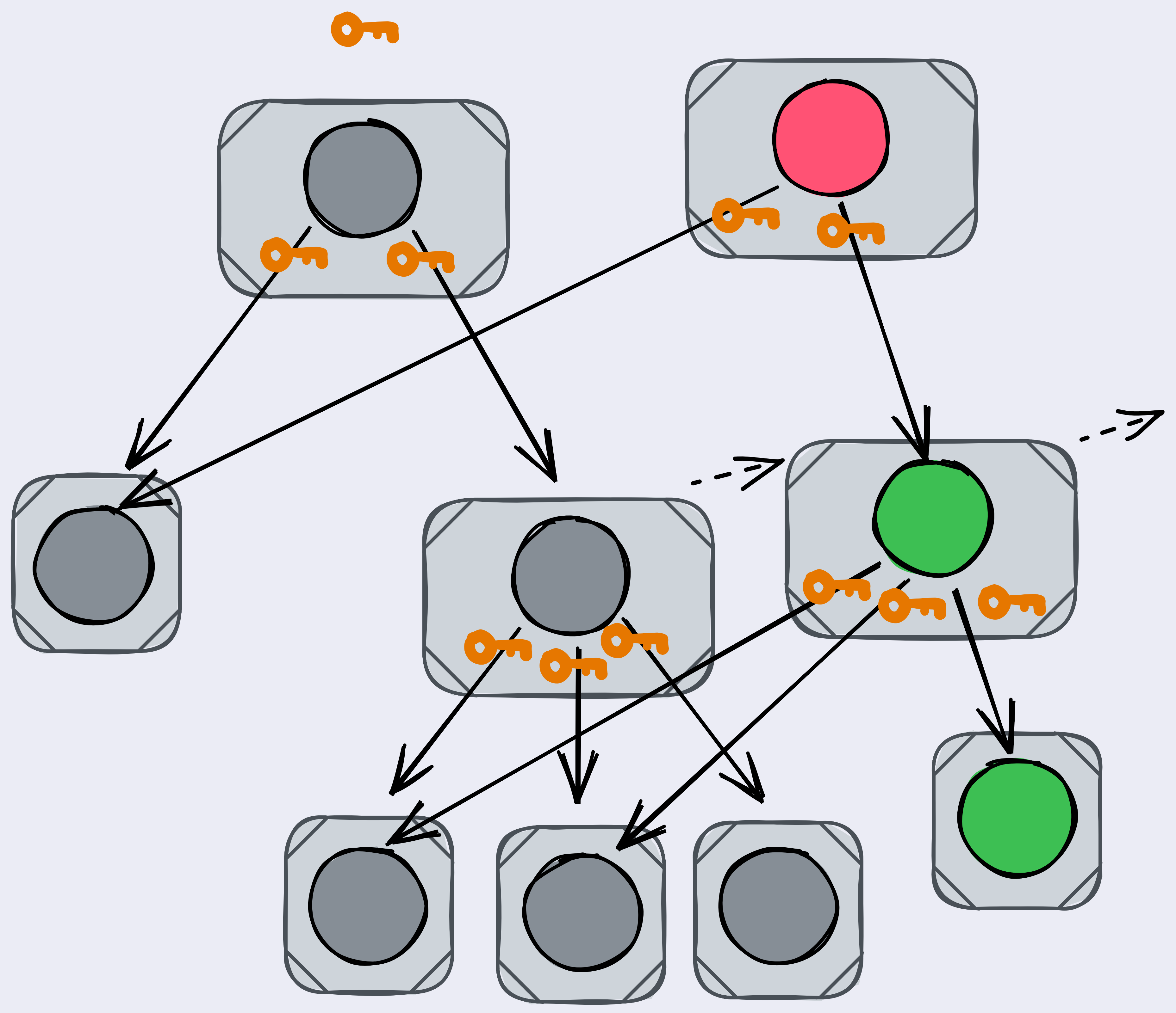
Private WNFS: Versioning

- Private nodes include their *namefilter without a revision*
- Allows seeking new versions
- Seeking necessary while walking unfamiliar private trees



Private WNFS: Versioning

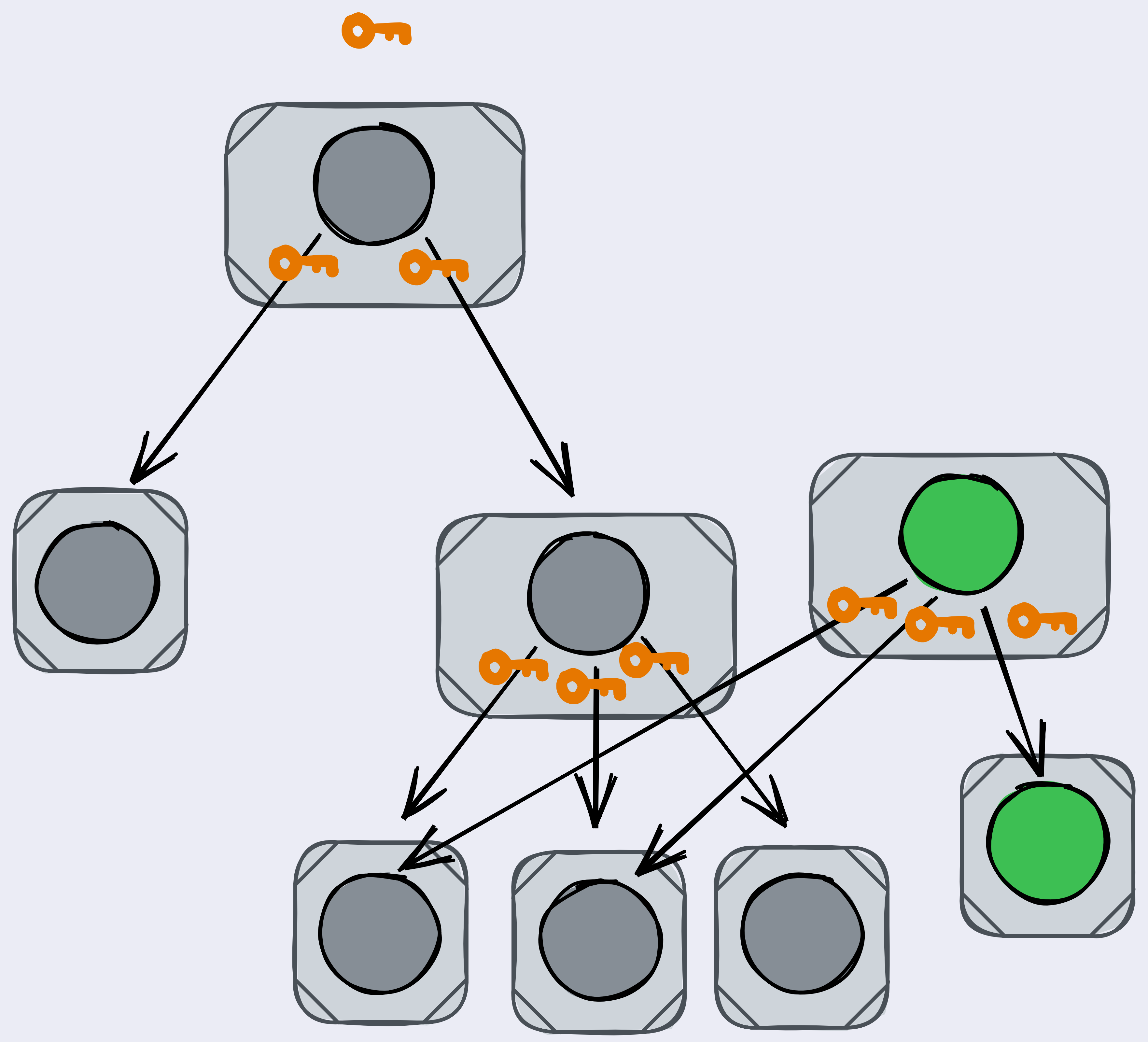
- Private nodes include their *namefilter without a revision*
- Allows seeking new versions
- Seeking necessary while walking unfamiliar private trees
- Someone with write access can repair the links later



Private WNFS: Backward Secrecy

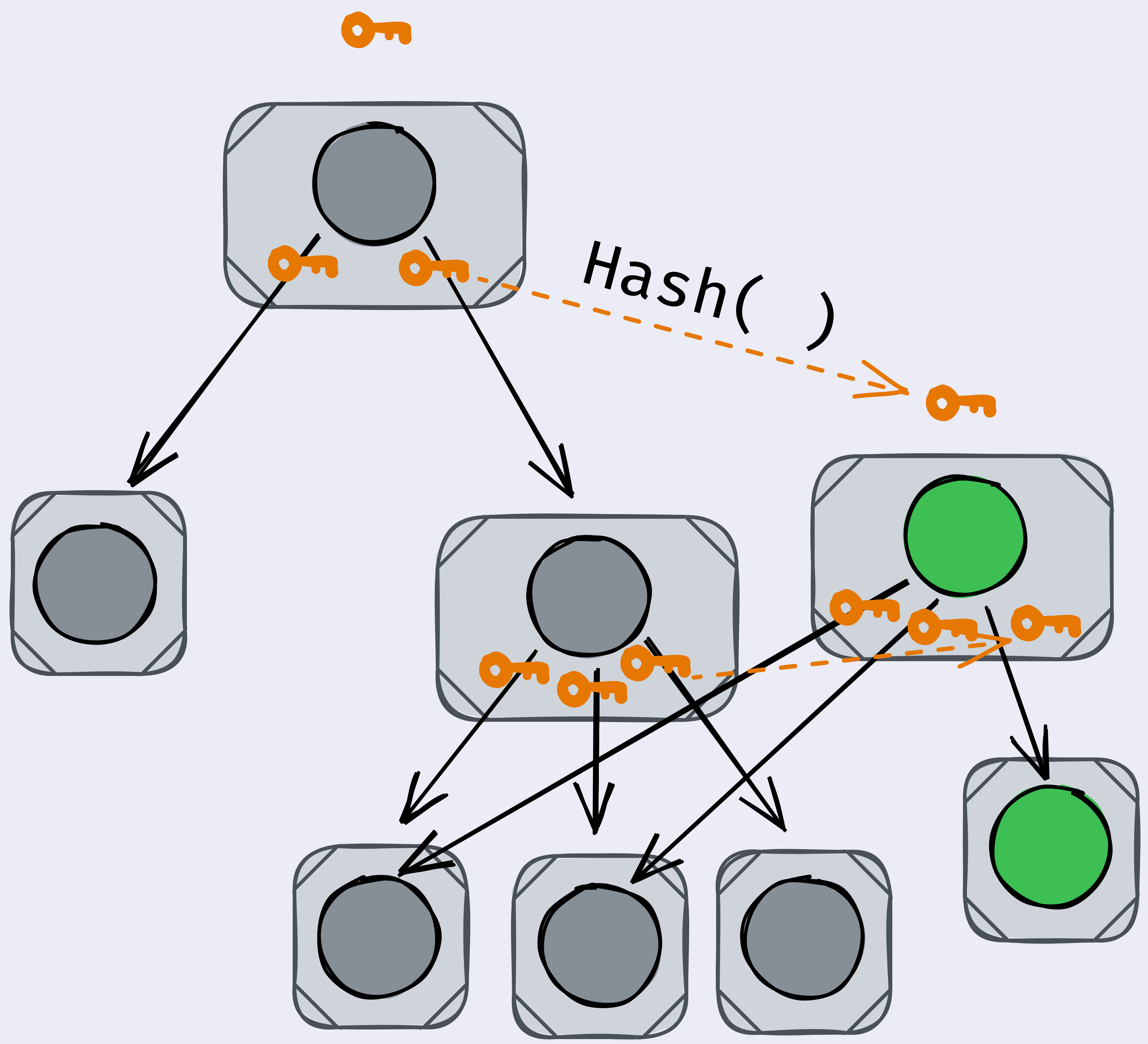
Goal: Granting read access to a directory shouldn’t *also* grant you read access to all previous versions.

Backward Secrecy



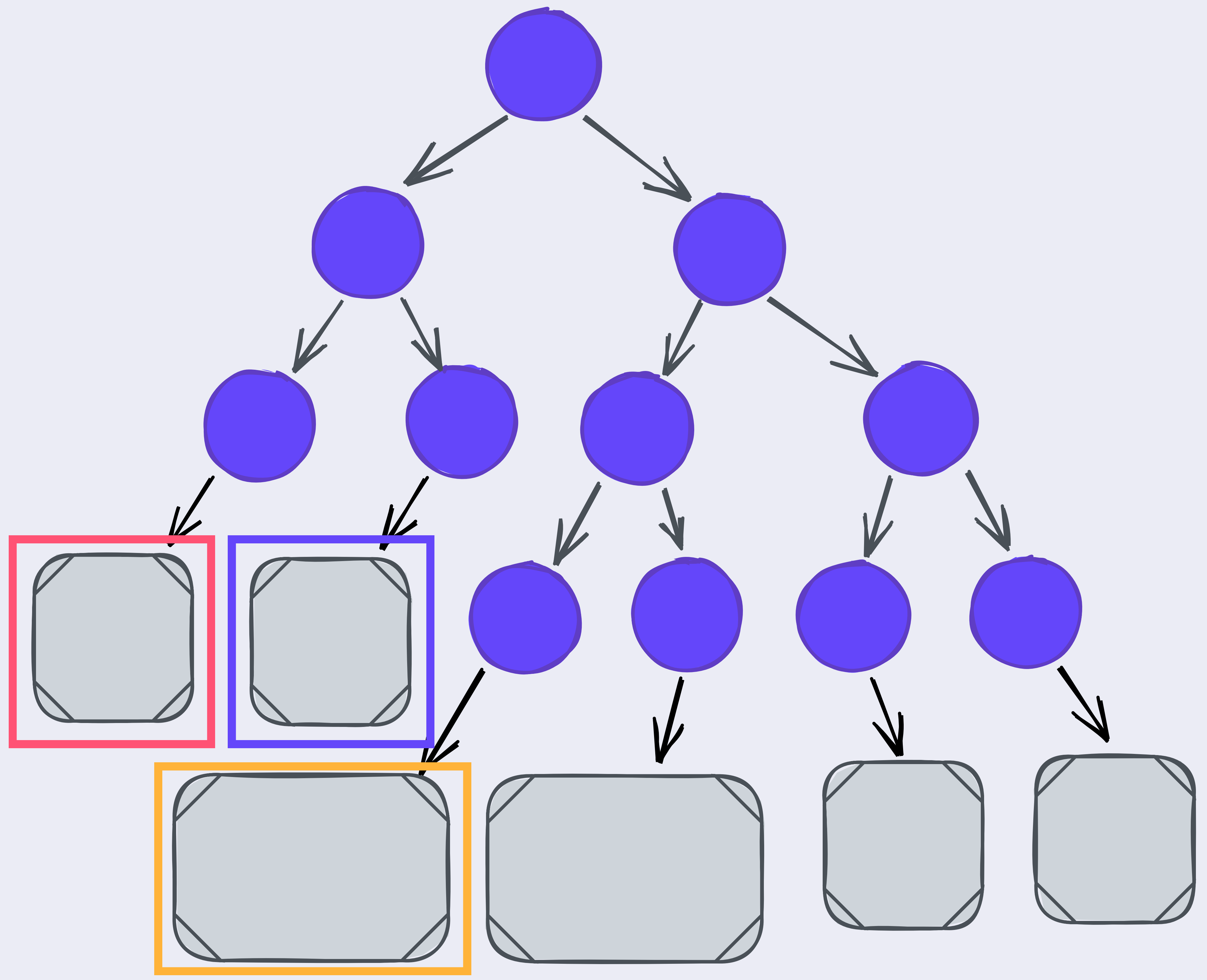
Backward Secrecy

- The next revision of a block is encrypted with essentially a hash of the current revision's key



Backward Secrecy

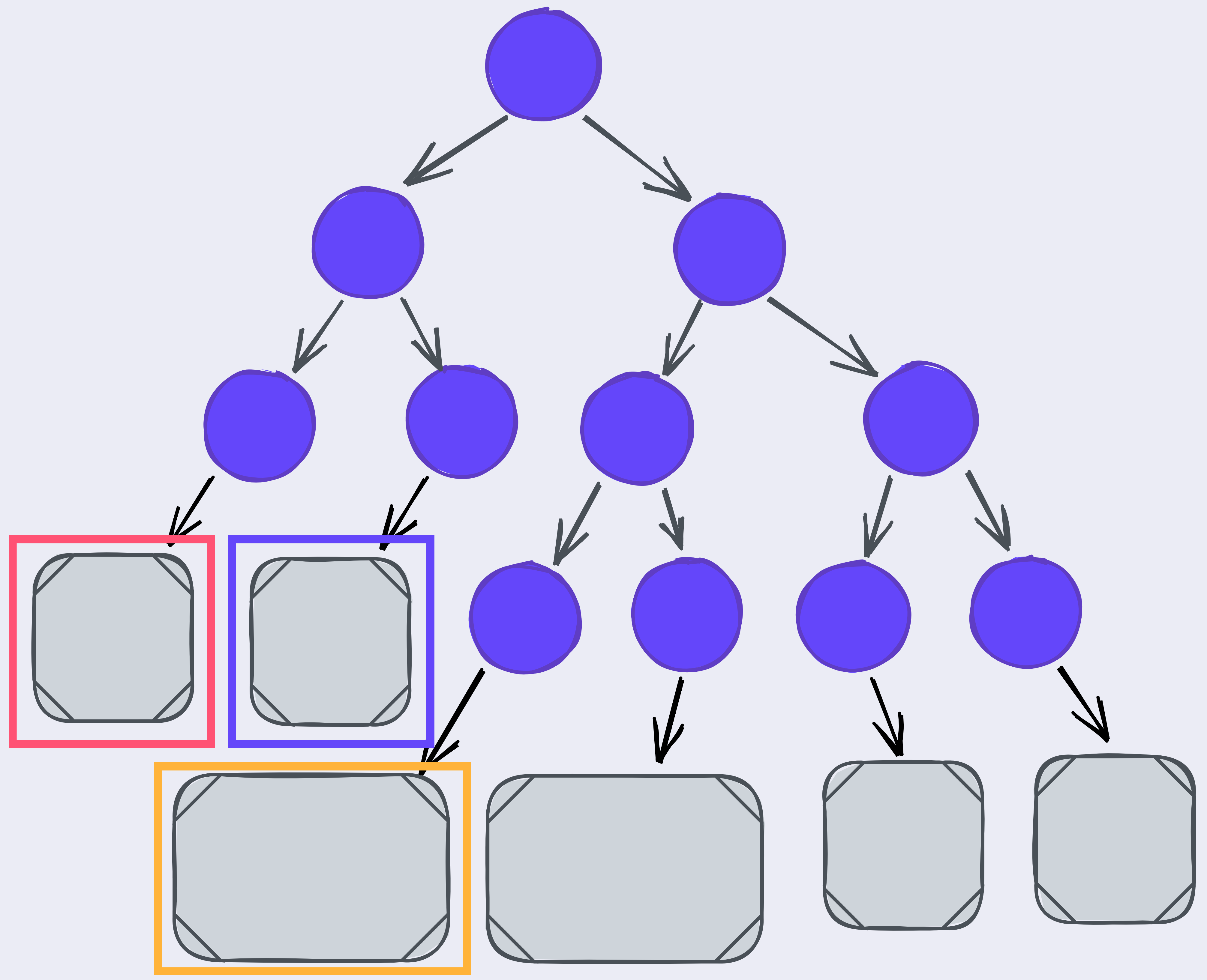
- The next revision of a block is encrypted with essentially a hash of the current revision’s key
- This key doubles as the revision identifier



```
namefilter = accumulate(165,656,448,<revision>)
namefilter = accumulate(165,656,<revision>)
namefilter = accumulate(165,925,<revision>)
```

Backward Secrecy

- The next revision of a block is encrypted with essentially a hash of the current revision's key
- This key doubles as the revision identifier



```
namefilter = accumulate(165,656,448,<sym key>)
namefilter = accumulate(165,656,<sym key>)
namefilter = accumulate(165,925,<sym key>)
```

Problems with Huge Seeks?
Solution: Skip Ratchet

Not in this talk, sorry!

Read the paper* 🙂

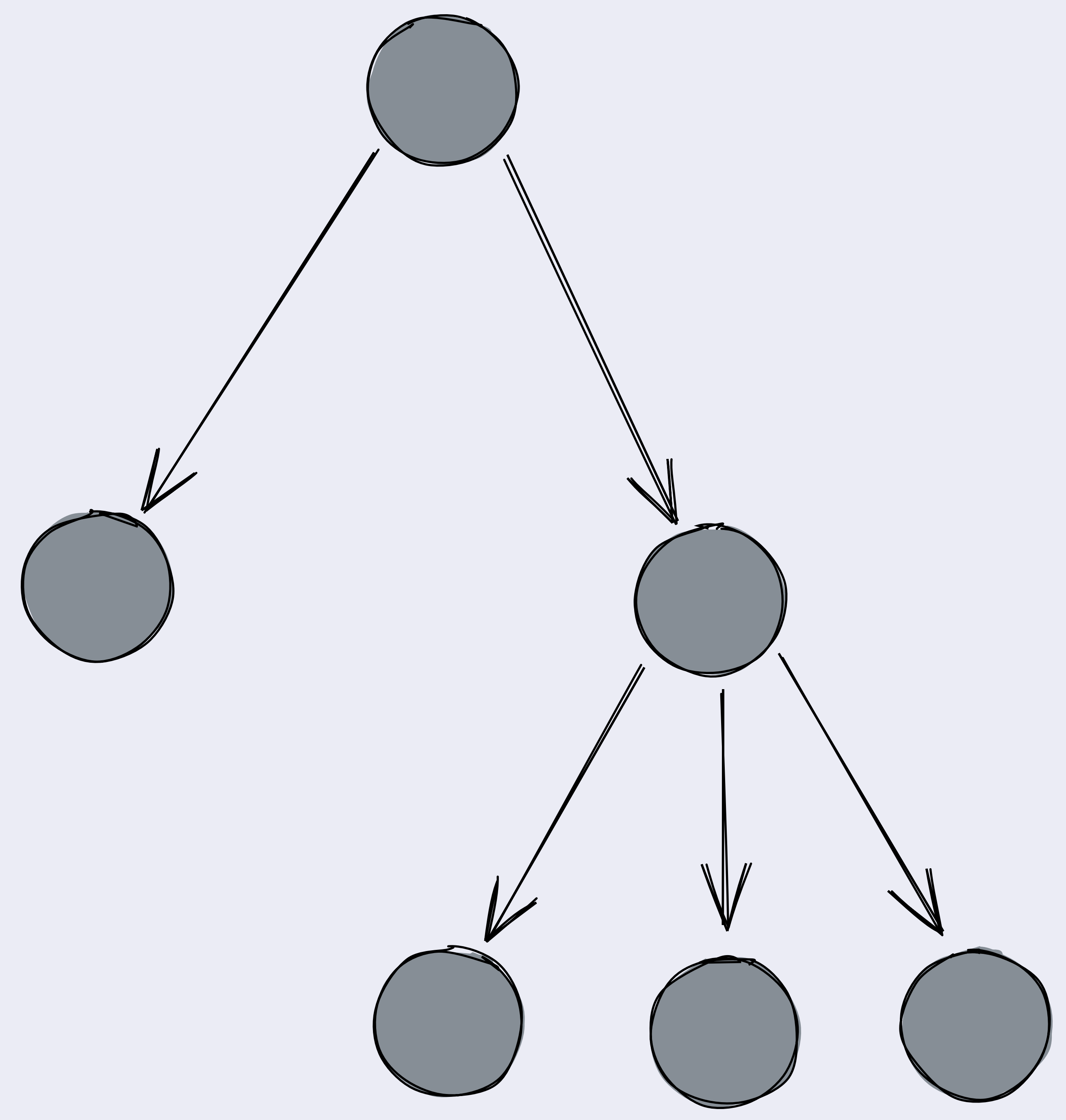
* https://github.com/fission-suite/skip-ratchet-paper

WNFS Implementation Considerations

Consider Light Clients:

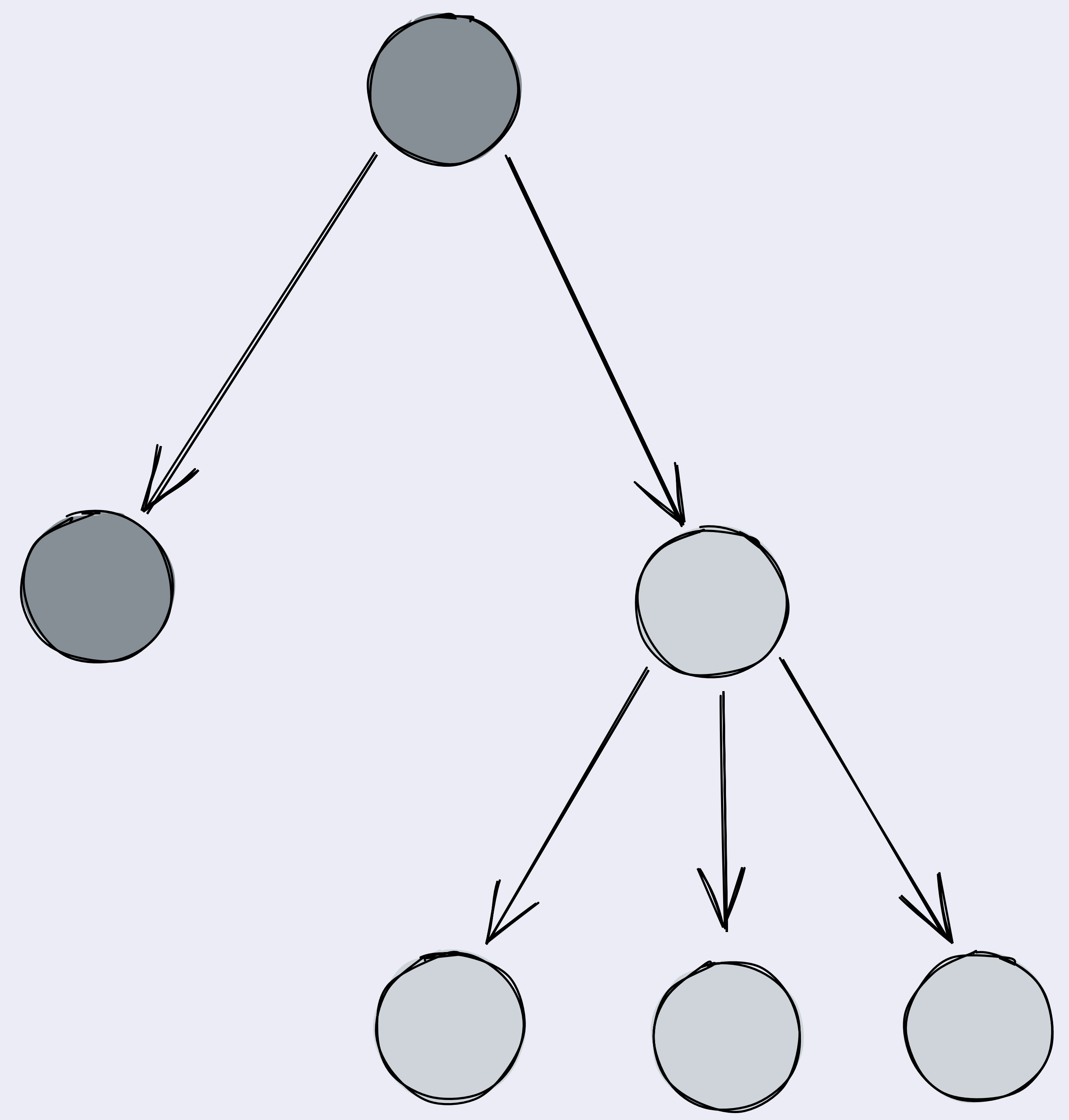
Working with partially replicated WNFS in browsers

Light Clients: Partial Storage



Light Clients: Partial Storage

- Only store what you touch
- Only touch things once your user's actions request that
- "lazy"
- Especially important with versioning!



Light Clients: Partial Storage

- Only store what you touch
- Only touch things once your user's actions request that
- "lazy"
- Especially important with versioning!
- Only decode single layers at a time
- Local block cache takes care of the rest

```
type Entry
  = File
  | Directory
  | ...

interface Directory {
  metadata: ...
  children: {
    [name: string]: CID
  }
  previous?: CID
}

decodeEntry(cid: CID): Promise<Entry>
encodeEntry(entry: Entry): Promise<CID>
```

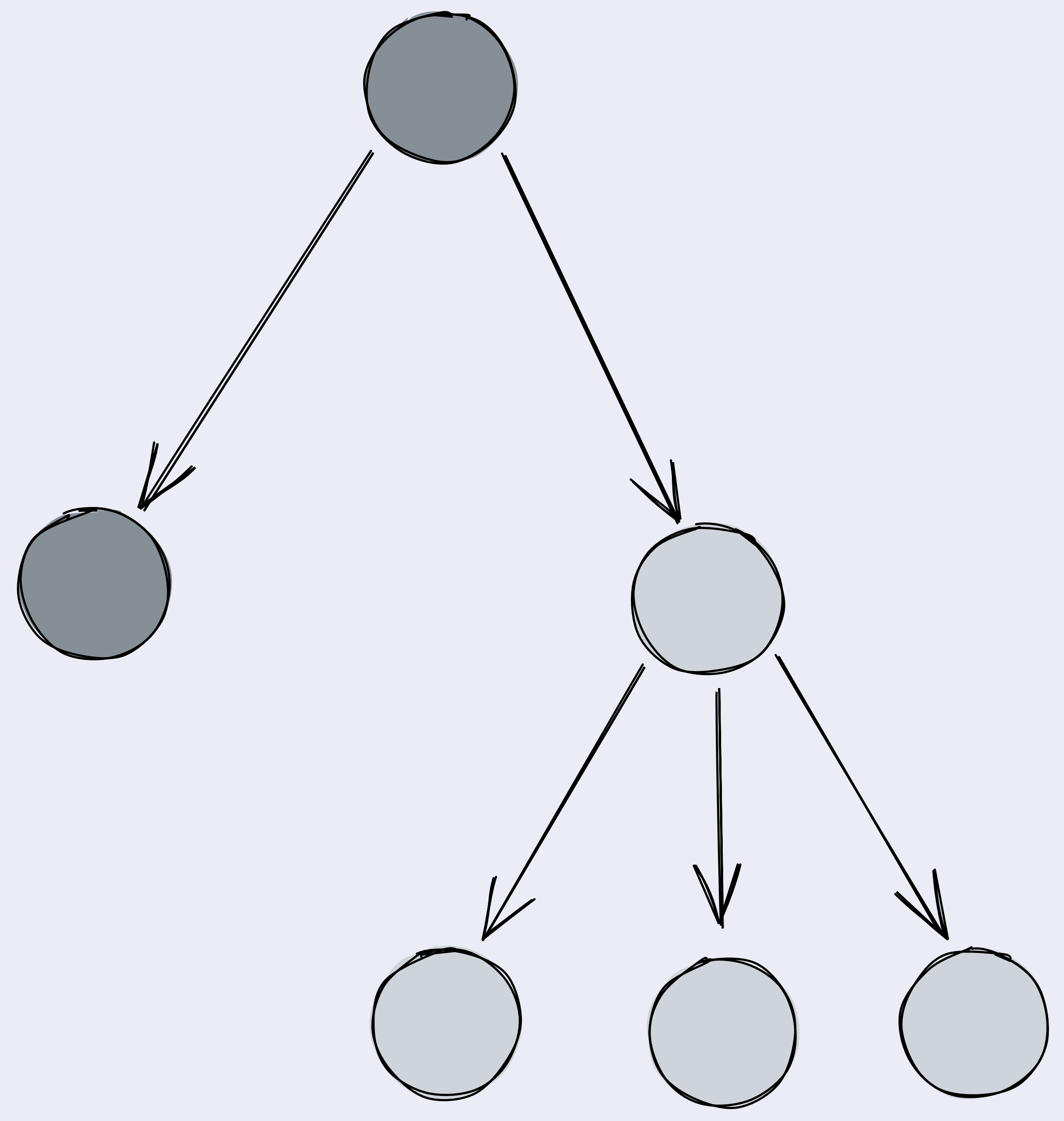
Consider Compressing Changes

Syncing WNFS over Bitswap takes at least

1 round-trip “per IPLD tree depth”

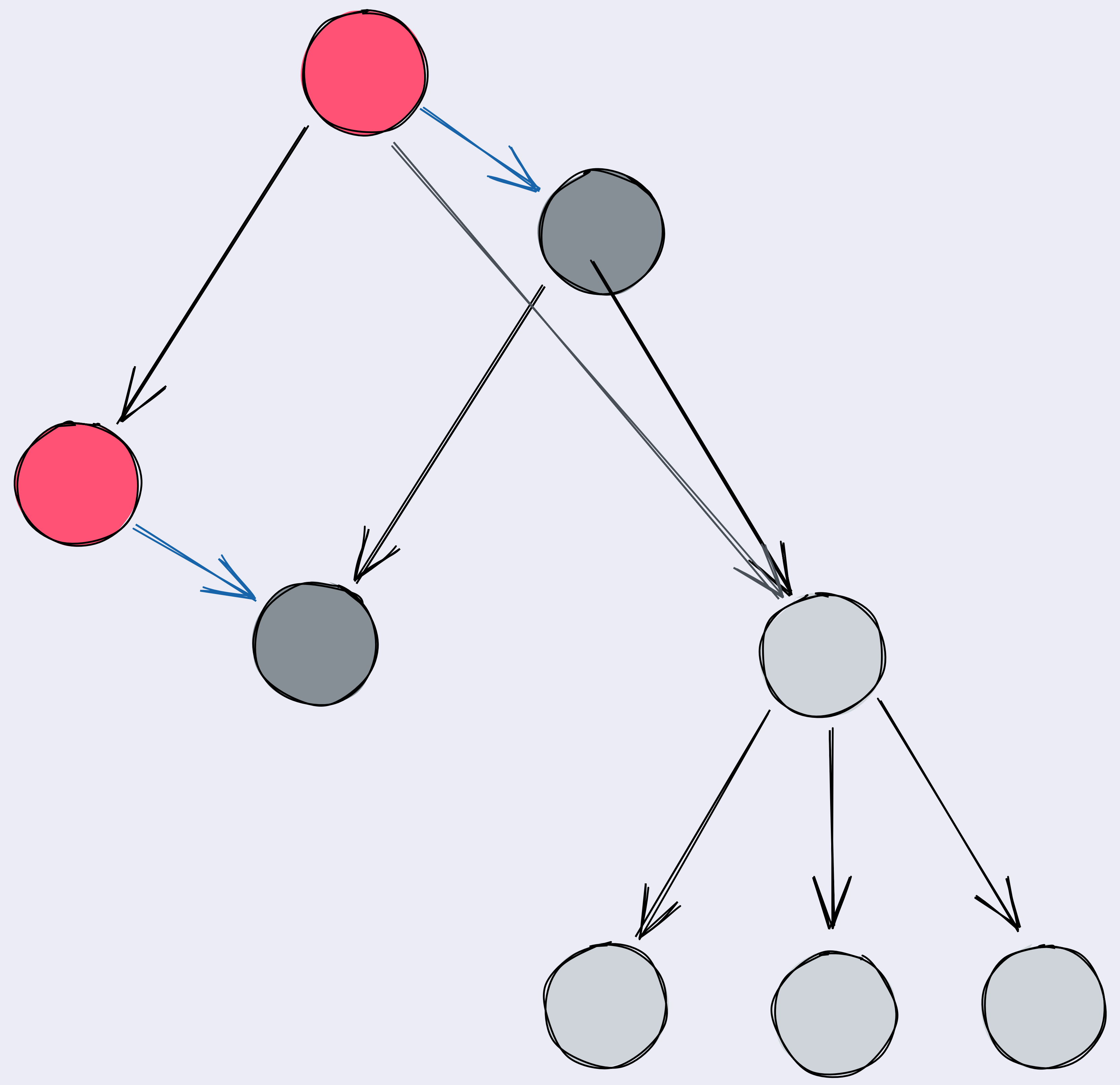
→ 1 round trip more per revision

Compress Changes



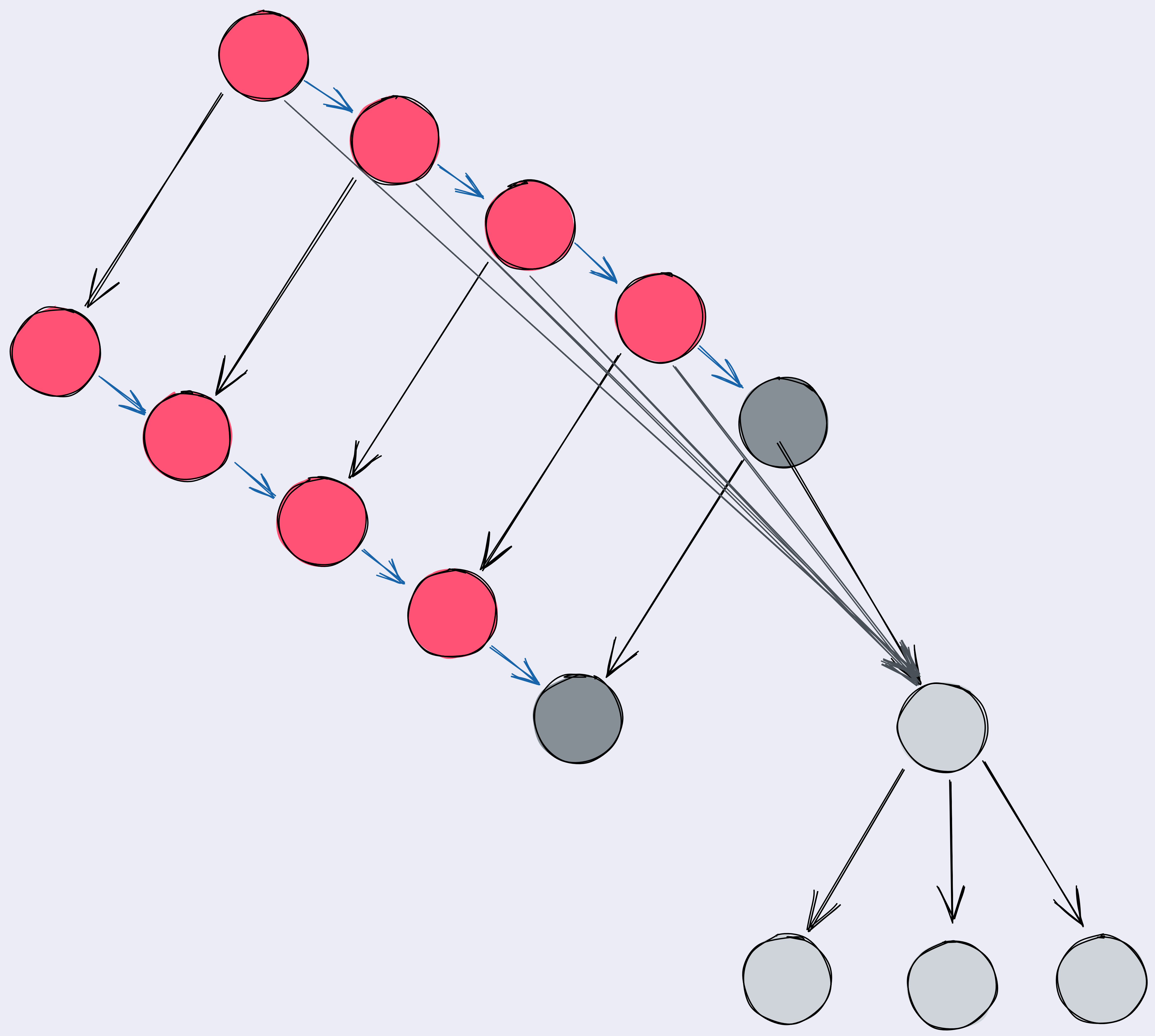
Compress Changes

- Problem:
    - Changes cause new revisions



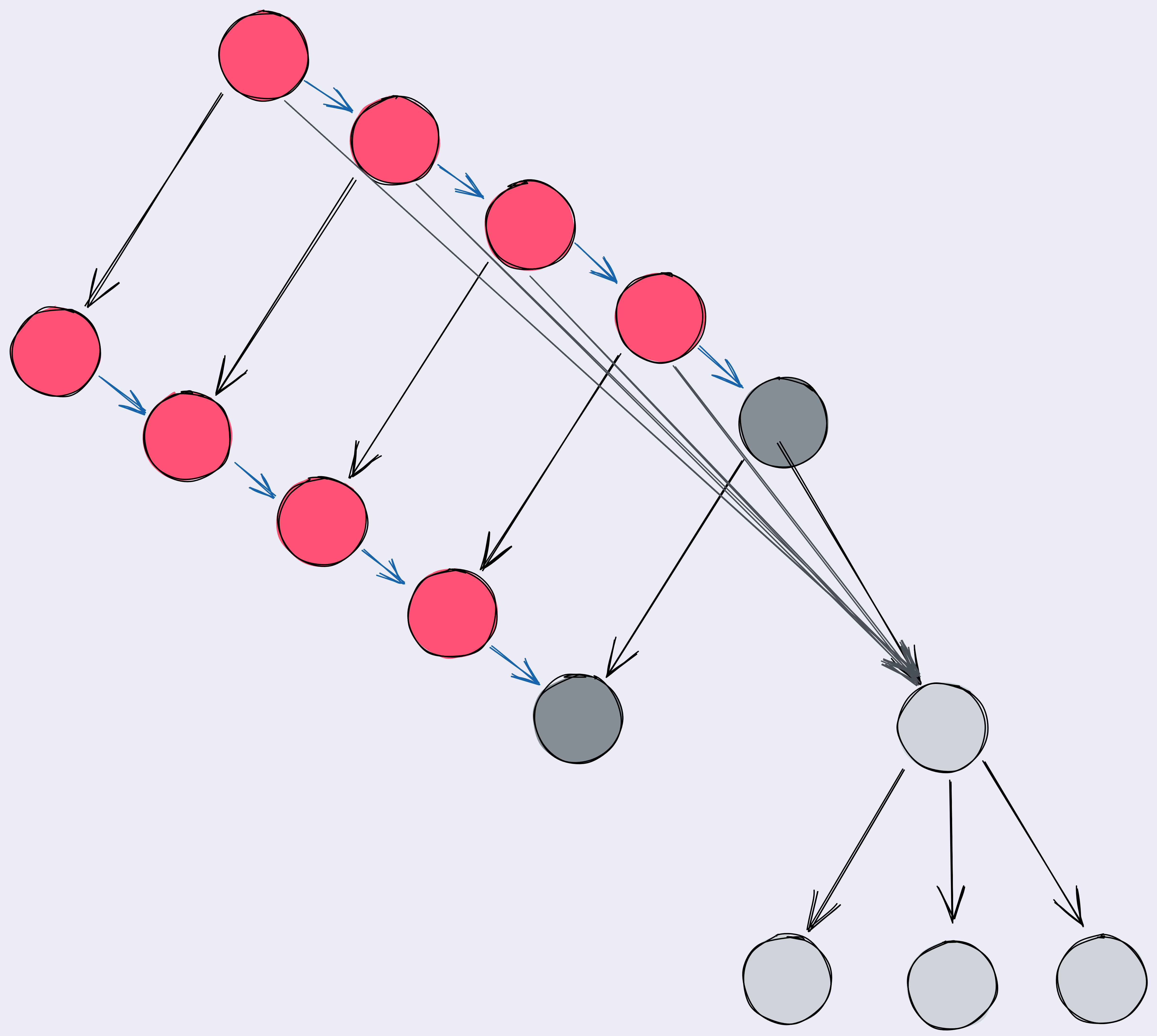
Compress Changes

- Problem:
    - Changes cause new revisions
    - Developers don’t necessarily want a revision for every change



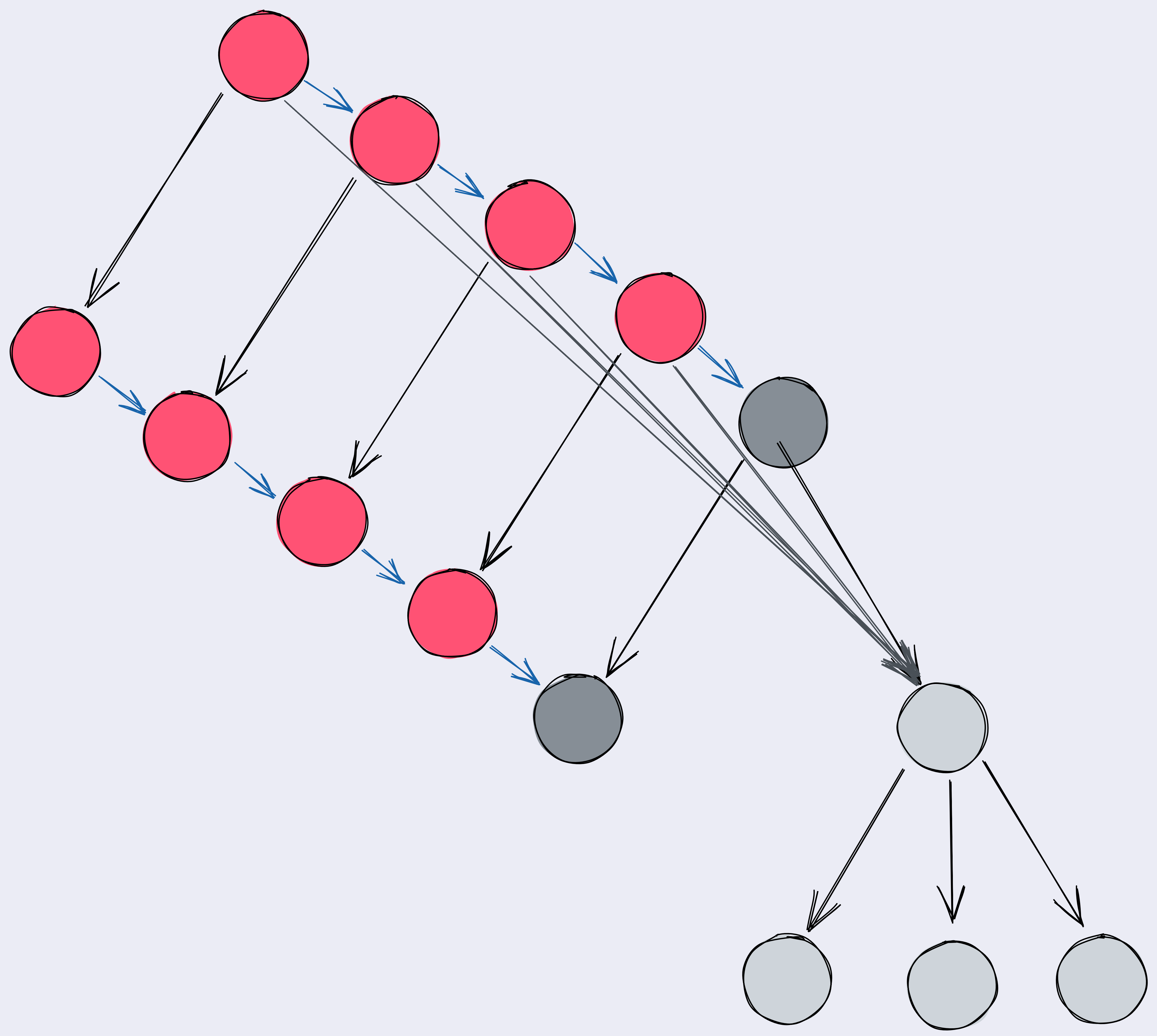
Compress Changes

- Problem:
  - Changes cause new revisions
  - Developers don't necessarily want a revision for every change
  - Each revision adds 2 * latency to sync-over-bitswap



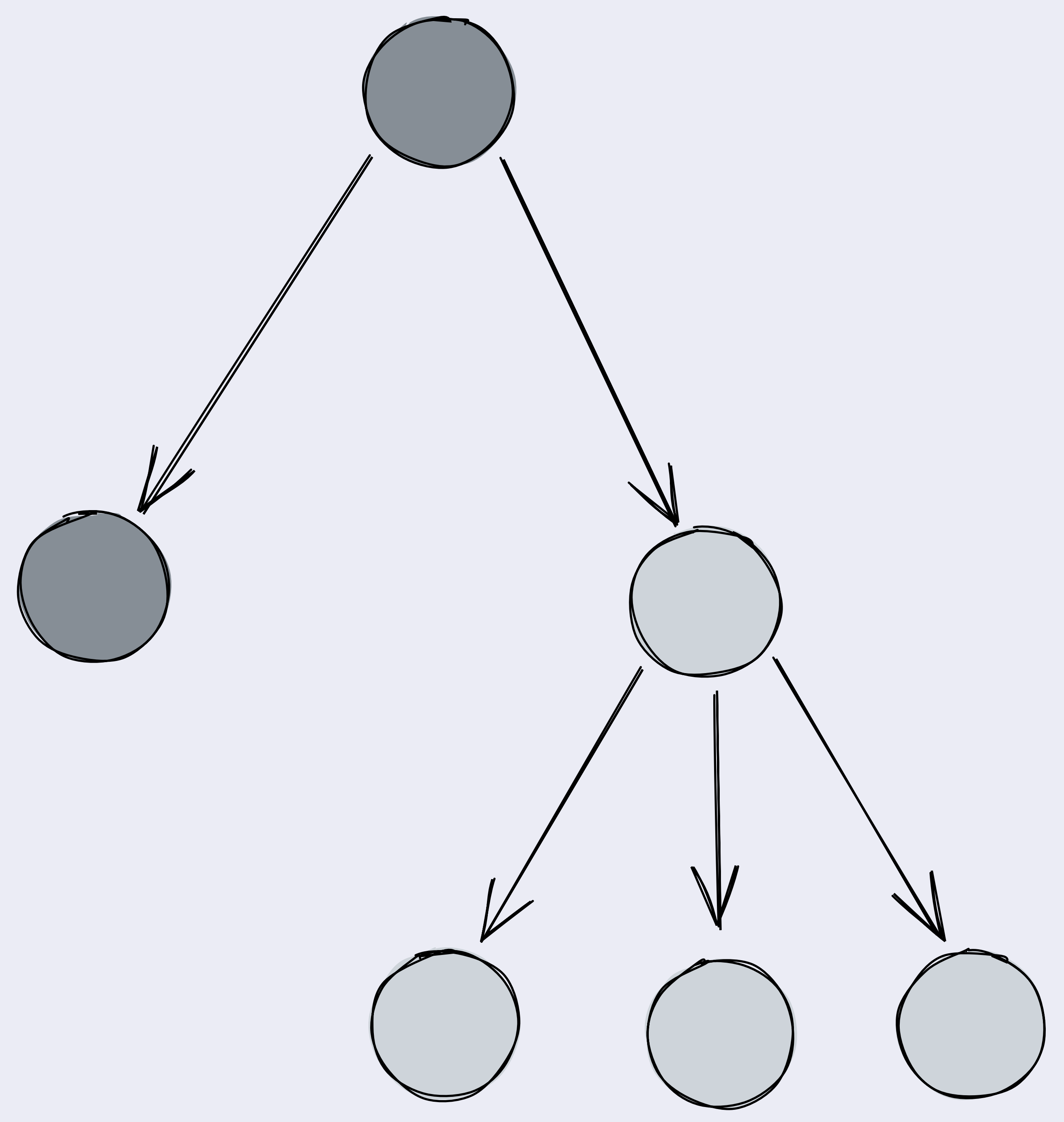
Compress Changes

- Problem:
  - Changes cause new revisions
  - Developers don’t necessarily want a revision for every change
  - Each revision adds 2 * latency to sync-over-bitswap
  - Also: Hash-linking means we need to serialize each in-between version



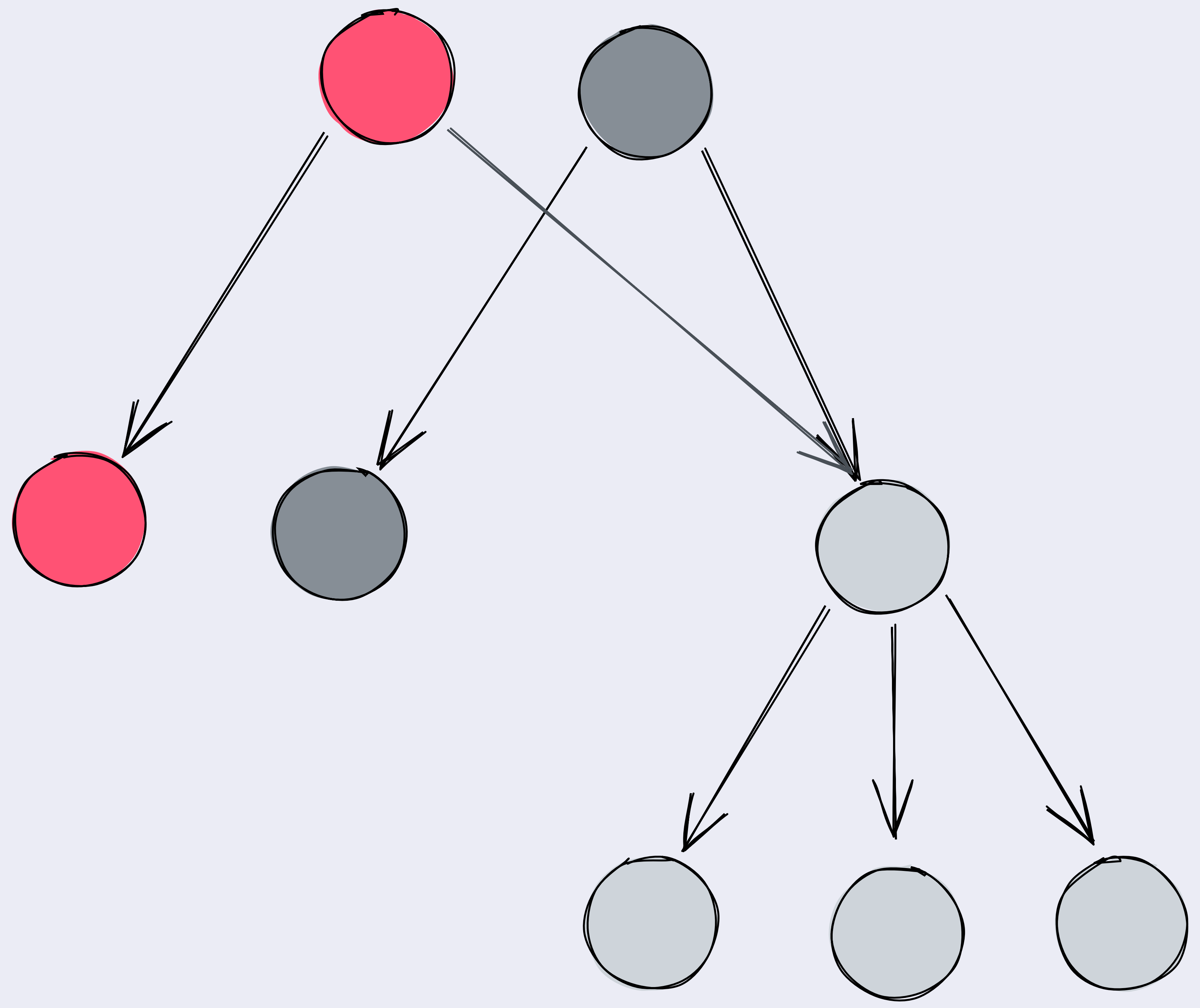
Compress Changes

- Solution:
  - Update the previous pointer lazily



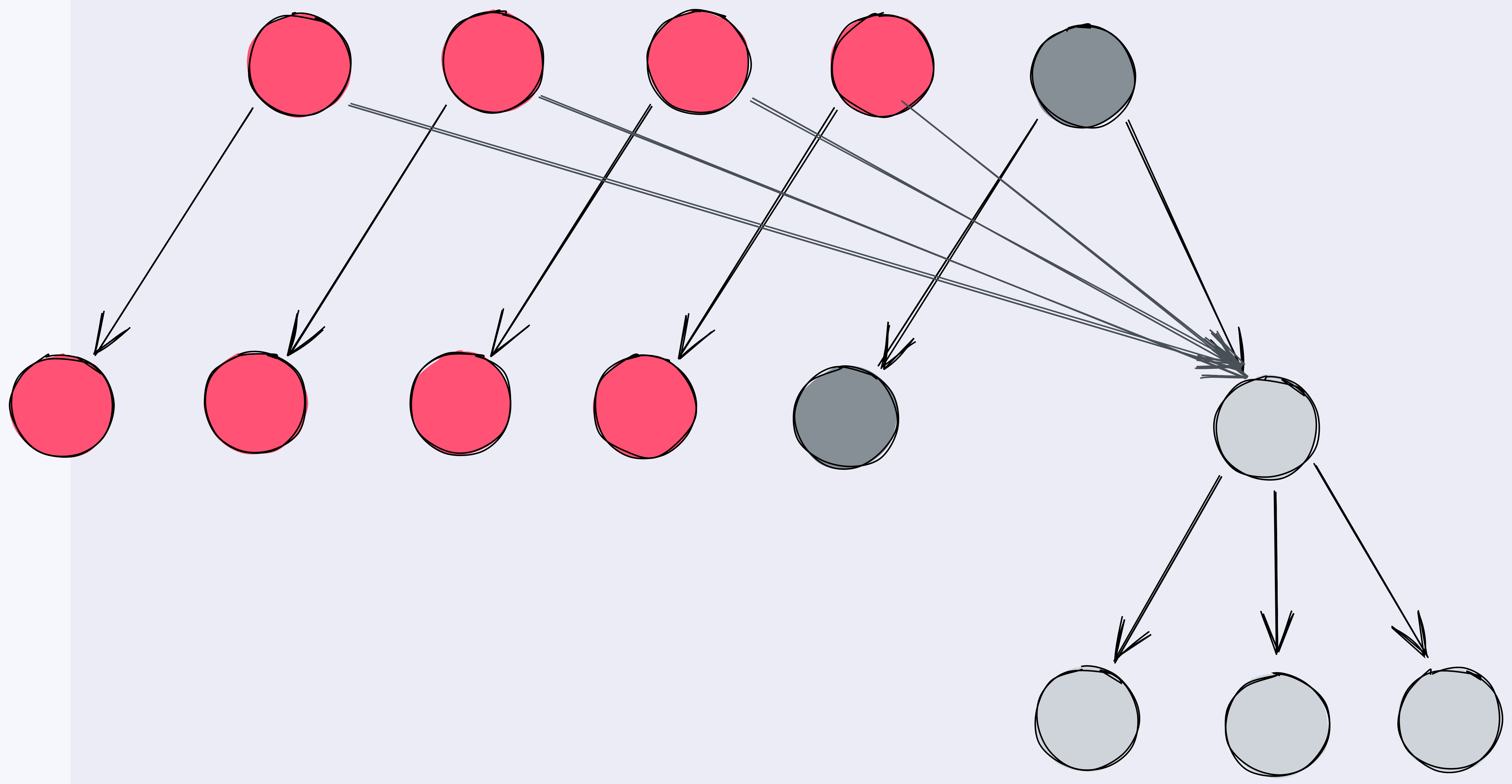
Compress Changes

- Solution:
  - Update the previous pointer lazily



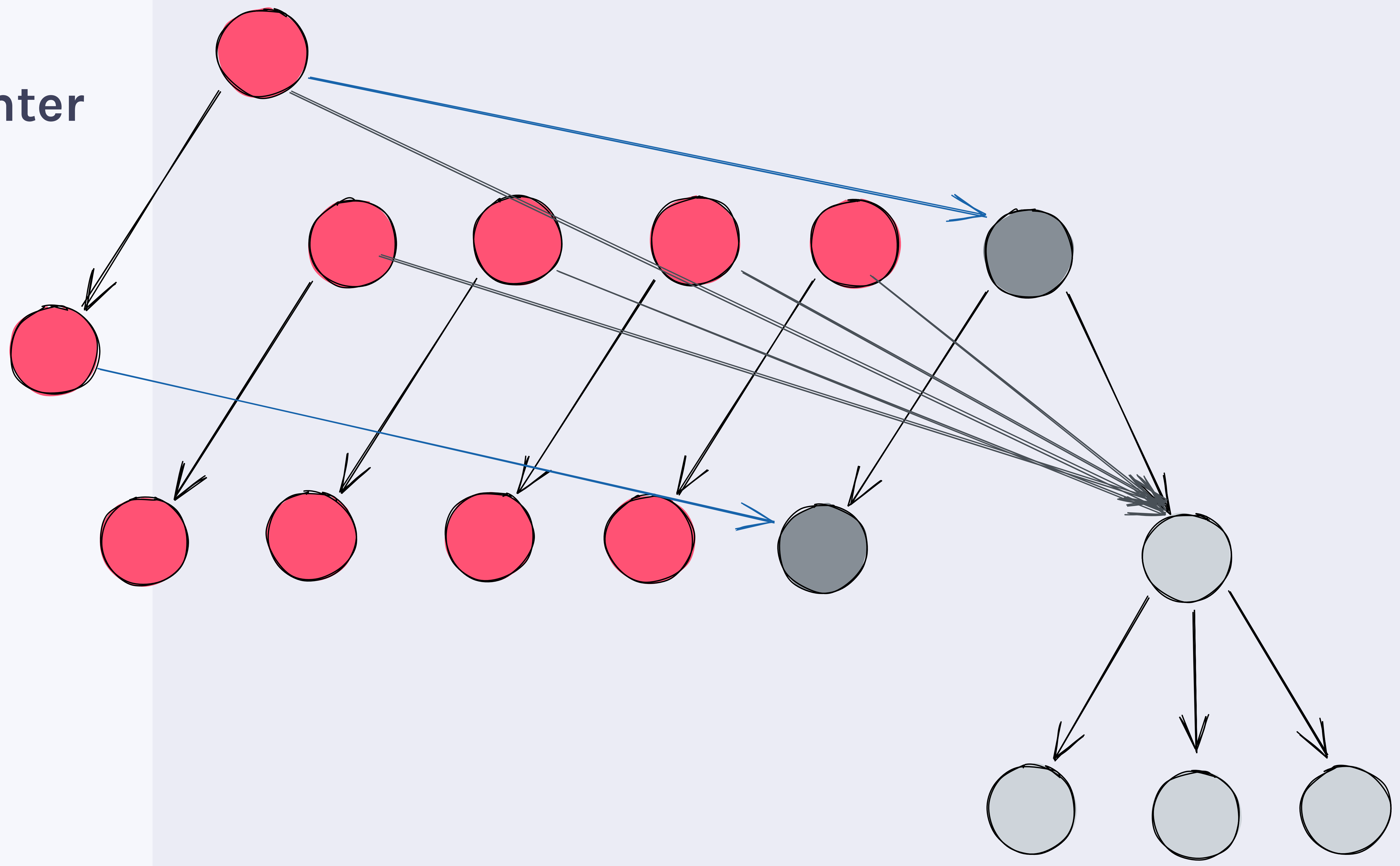
Compress Changes

- Solution:
  - Update the previous pointer lazily
  - Copy deserialized nodes



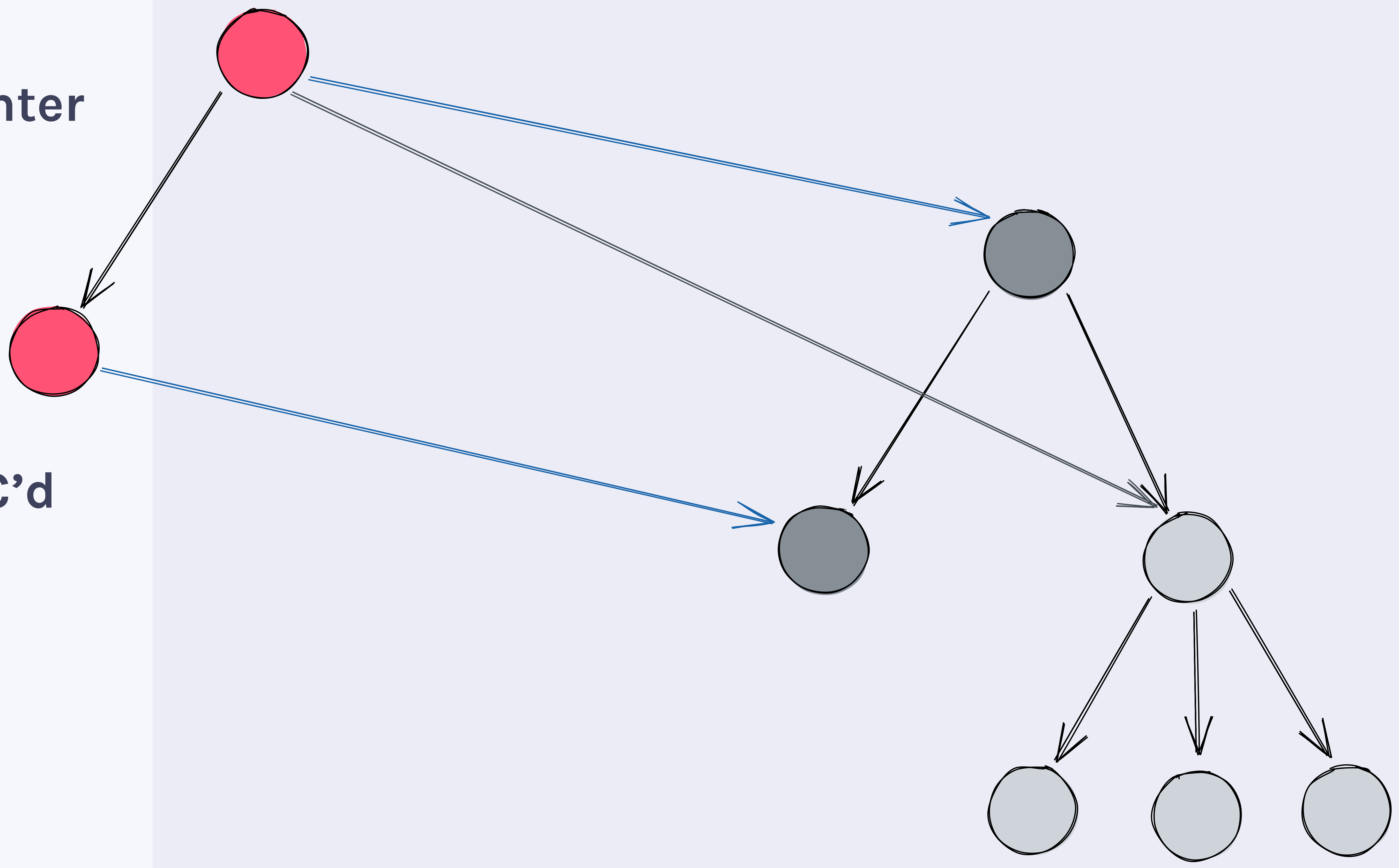
Compress Changes

- Solution:
  - Update the previous pointer lazily
  - Copy deserialized nodes
  - Finalize by updating the previous pointer



Compress Changes

- Solution:
  - Update the previous pointer lazily
  - Copy deserialized nodes
  - Finalize by updating the previous pointer
- In-between nodes can be GC'd by the host language



Compress Changes

- Solution:
  - Update the previous pointer lazily
  - Copy deserialized nodes
  - Finalize by updating the previous pointer
- In-between nodes can be GC'd by the host language

```
type VirtualEntry
  = VirtualFile
  | VirtualDirectory
  | ...

interface VirtualDirectory {
  metadata: ...
  children: {
    [name: string]: CID | VirtualEntry
  }
  previous?: CID
}
```

Compress Changes

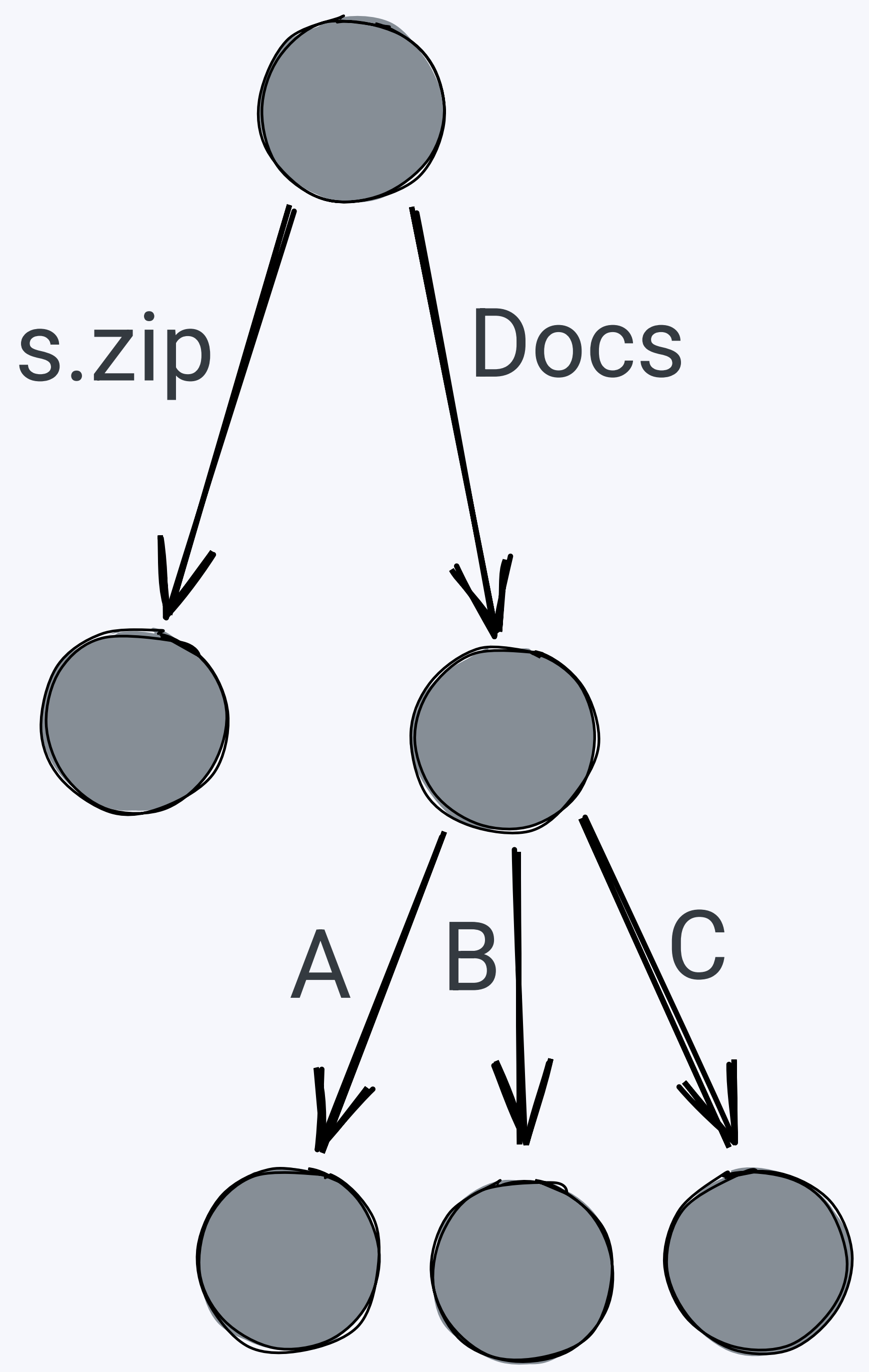
- Solution:
  - Update the previous pointer lazily
  - Copy deserialized nodes
  - Finalize by updating the previous pointer
- In-between nodes can be GC'd by the host language

```
{
  metadata: ...
  children: {
    "stuff.zip": {
      metadata: ...
      content: CID(bafy...)
    }
    "Docs": CID(bafy...)
  }
  previous: CID(bafy...)
}
```

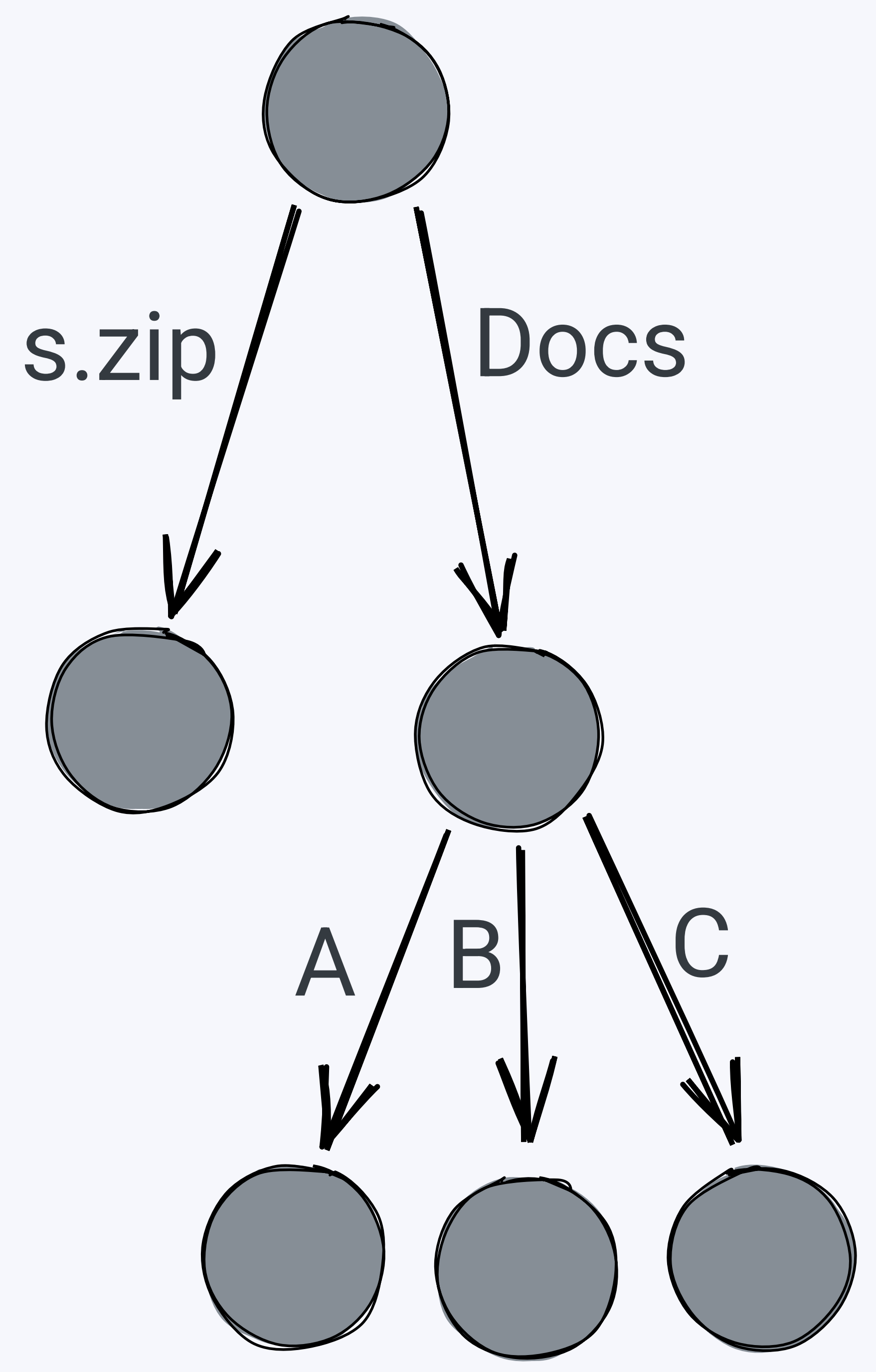
Consider Nonlocal Concurrency

Other devices make progress while being offline.

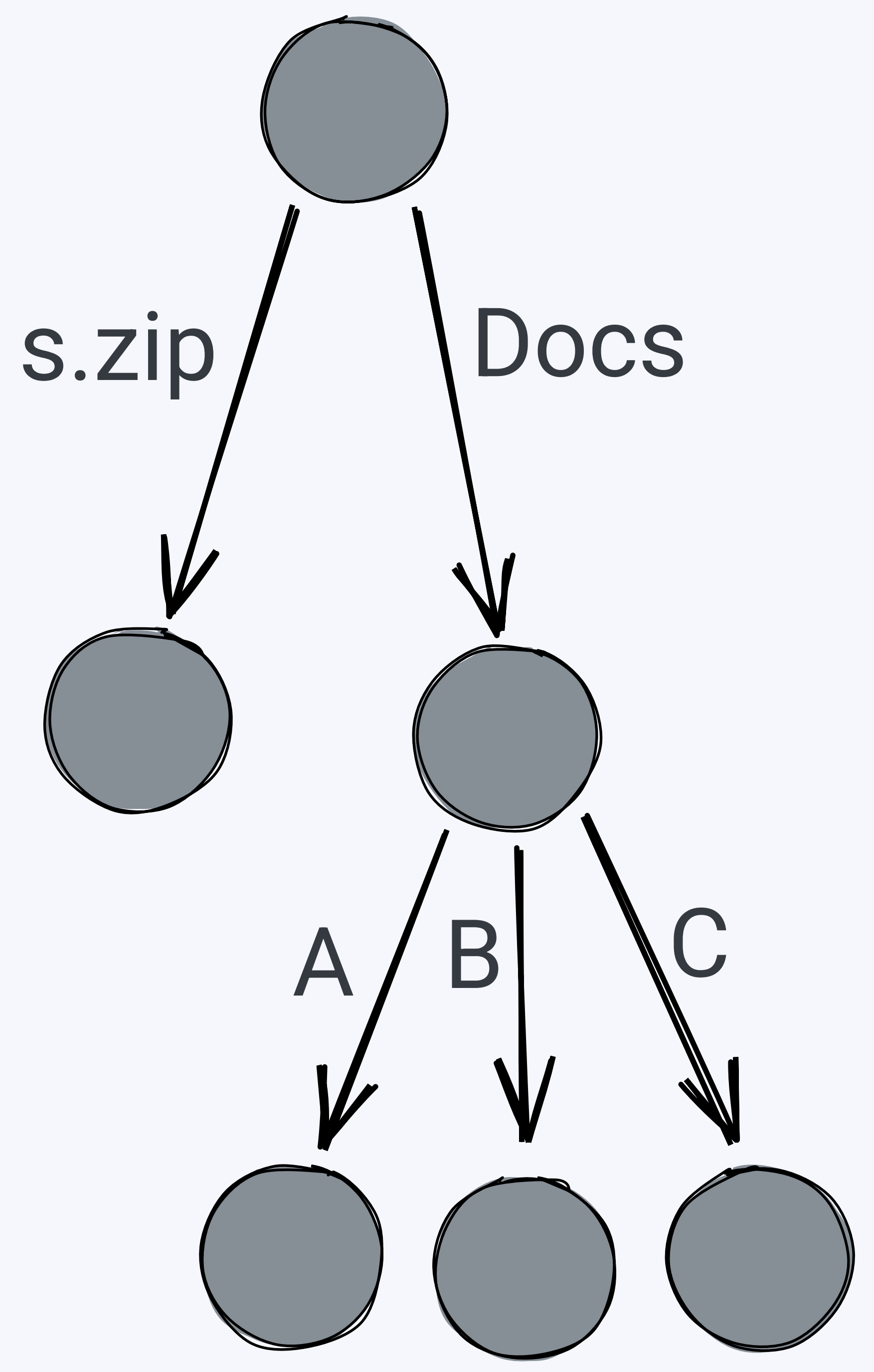
Local-First!

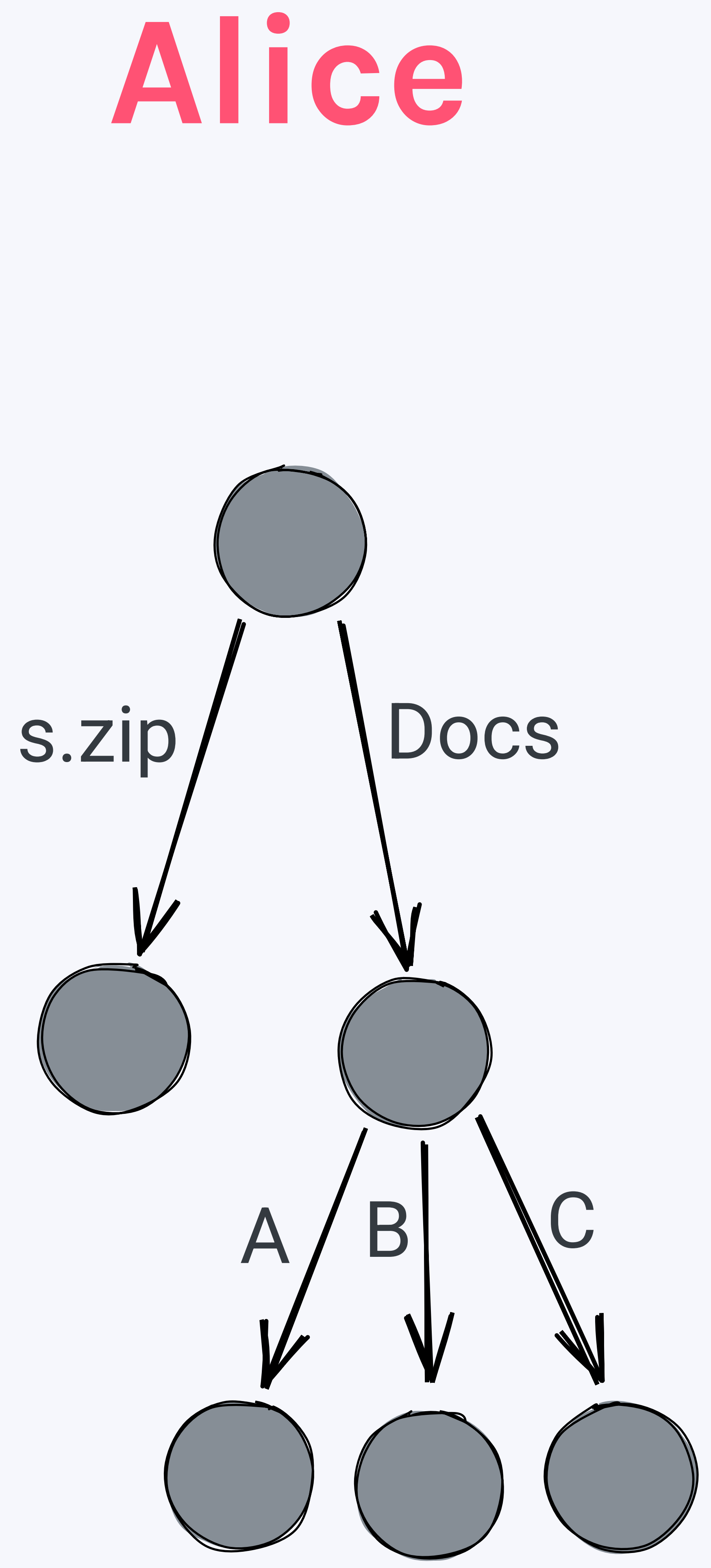

s.zip
Docs
A
B
C

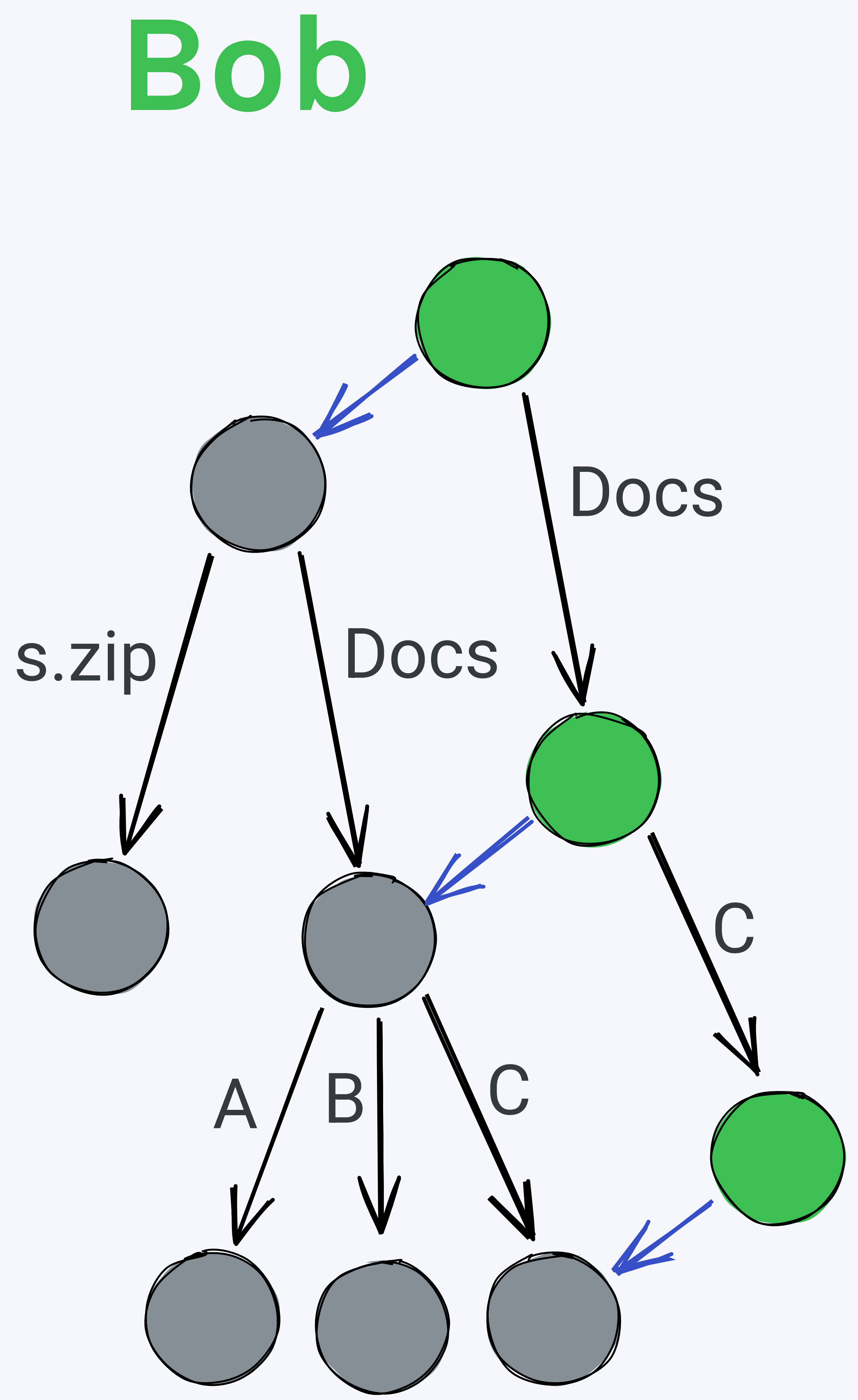
Alice



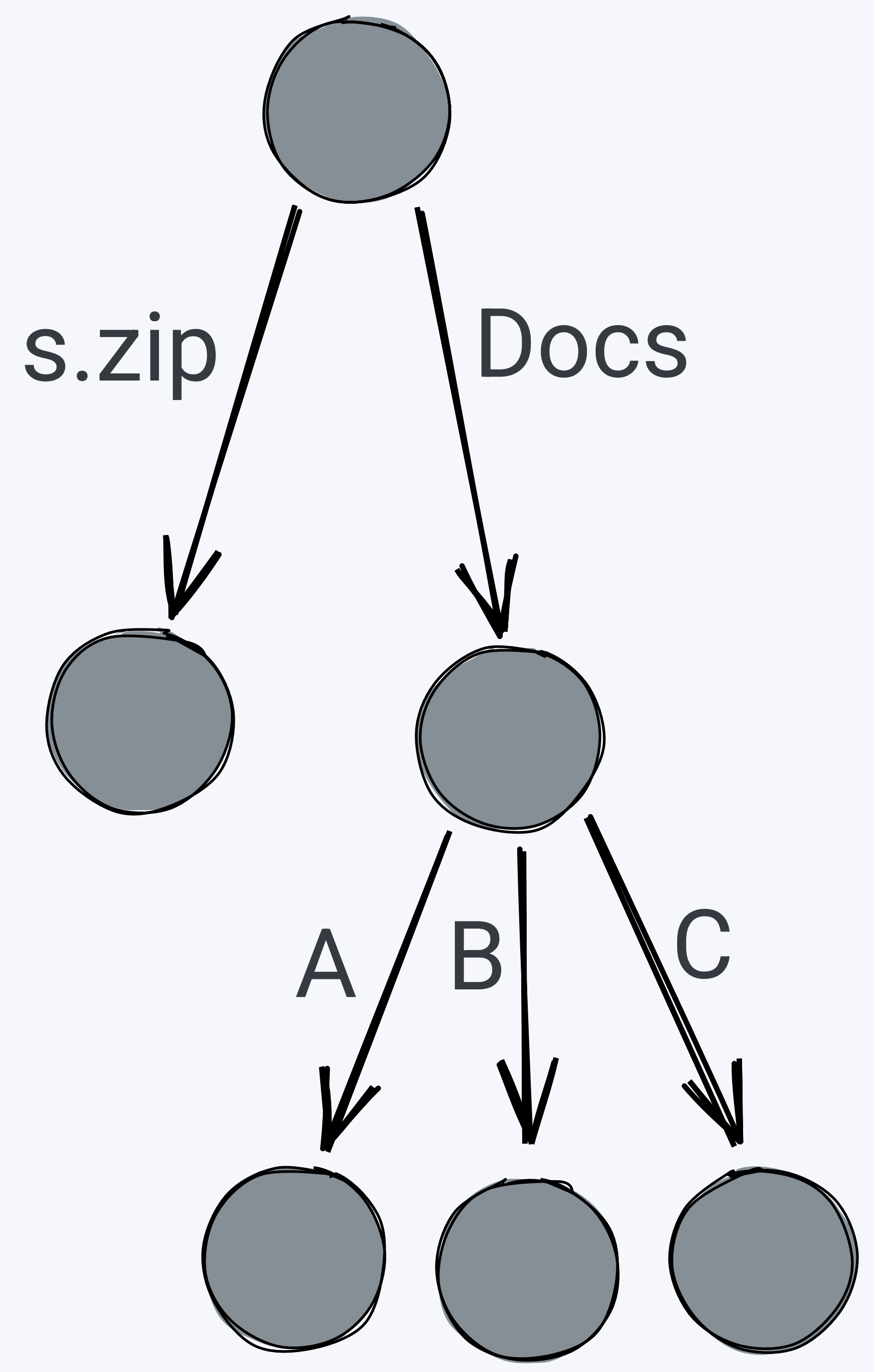
Bob



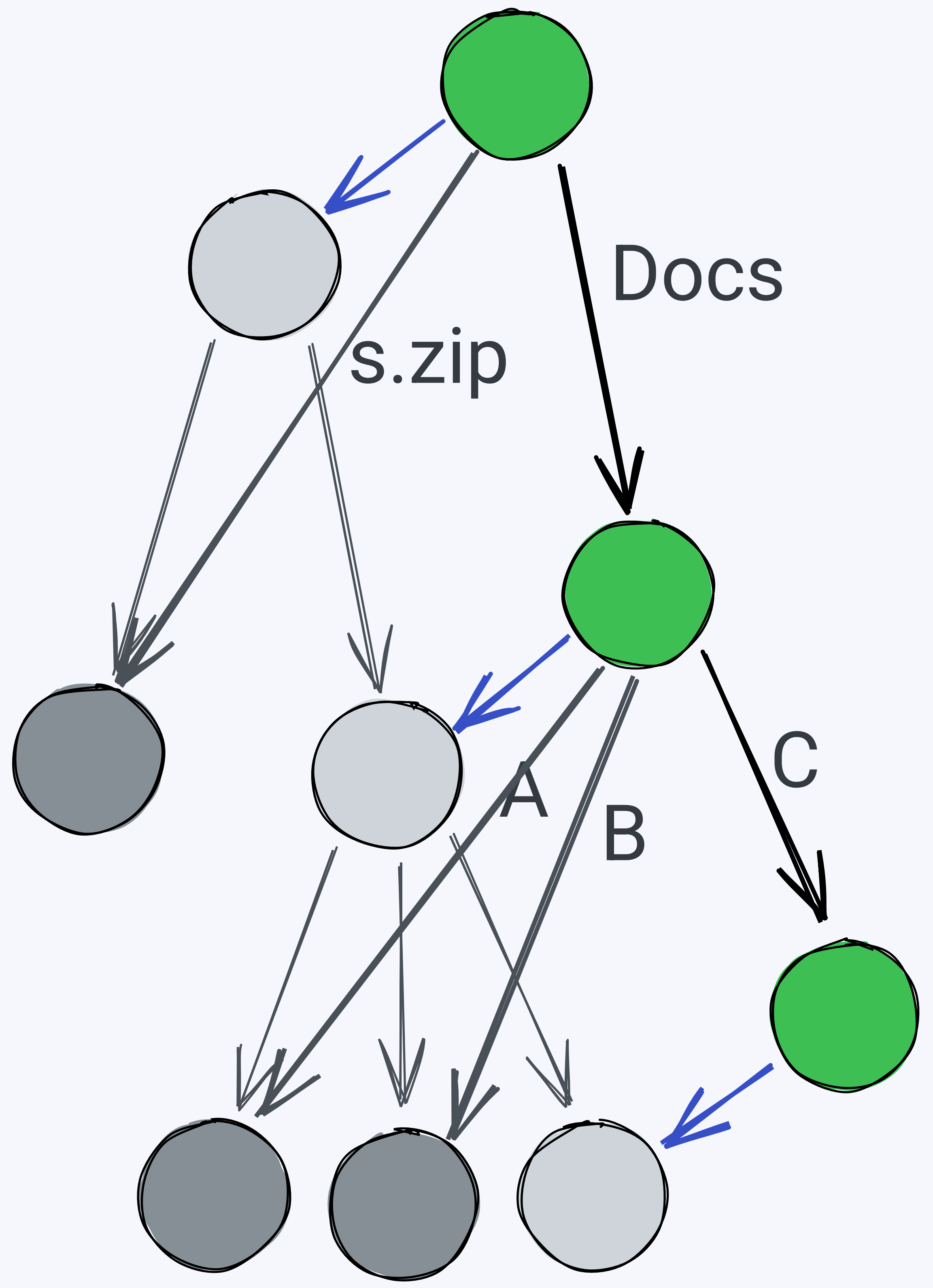

Alice
s.zip
Docs
A
B
C


Bob
Docs
s.zip
Docs
C
A
B
C

Alice

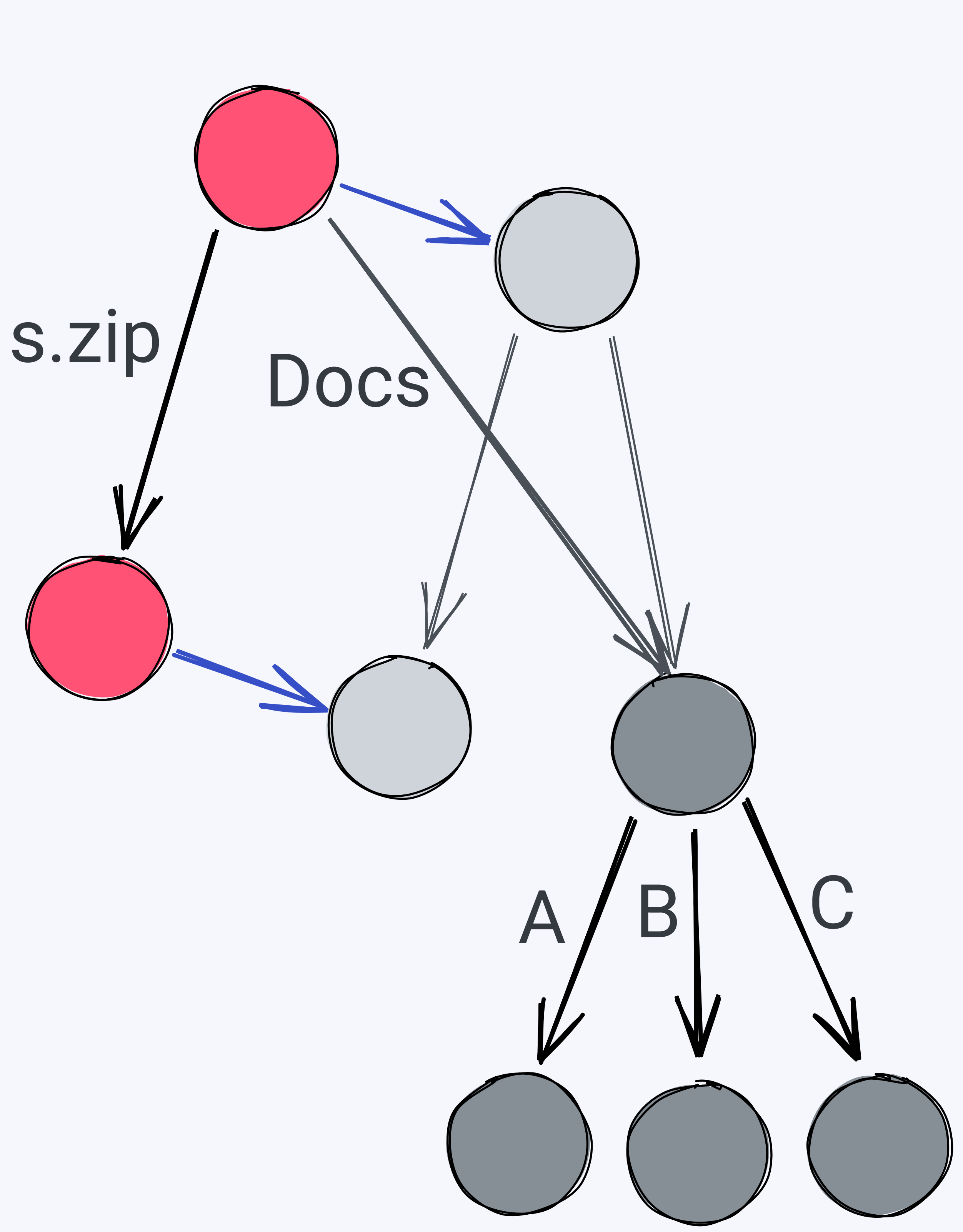


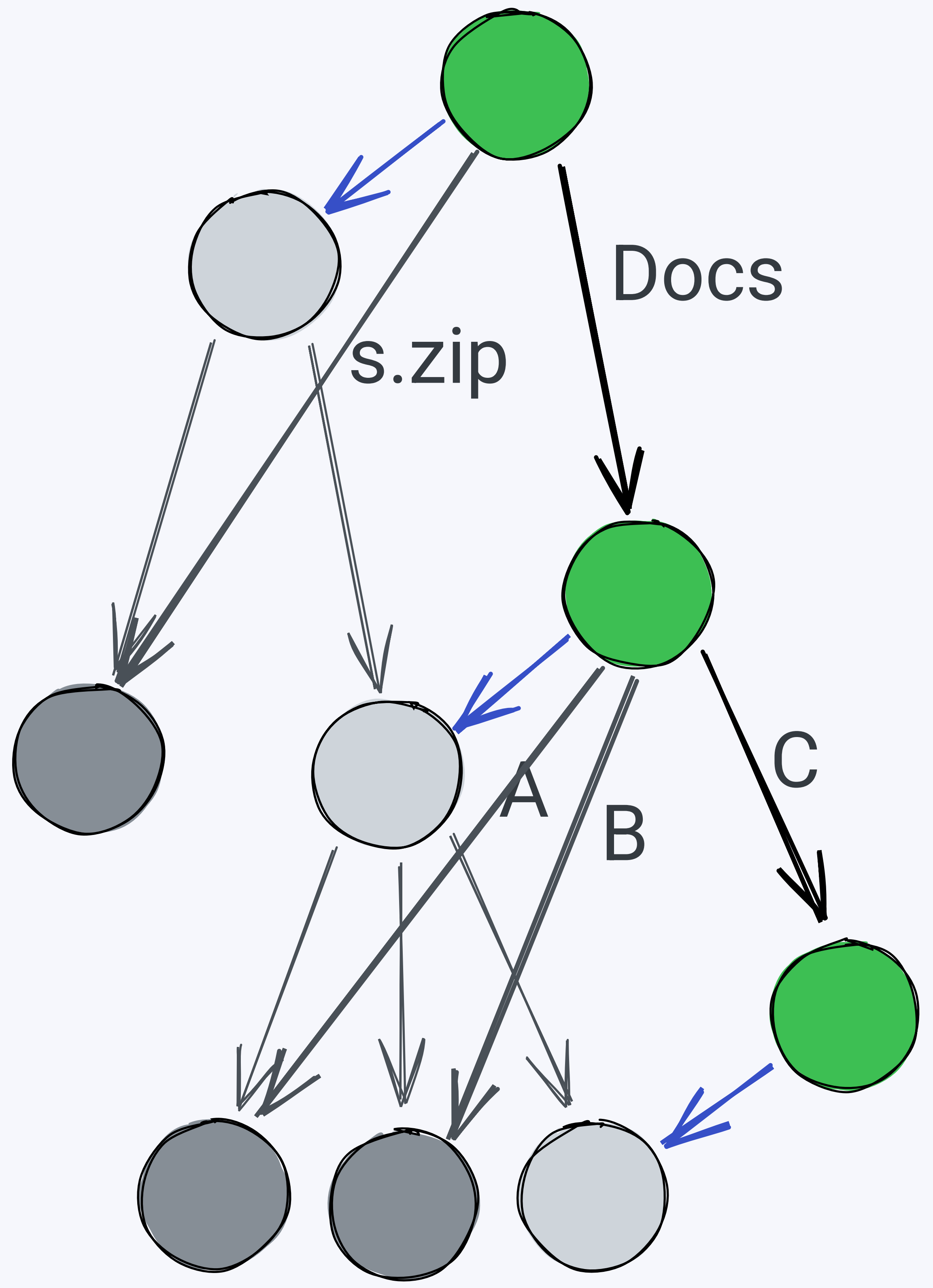
Bob

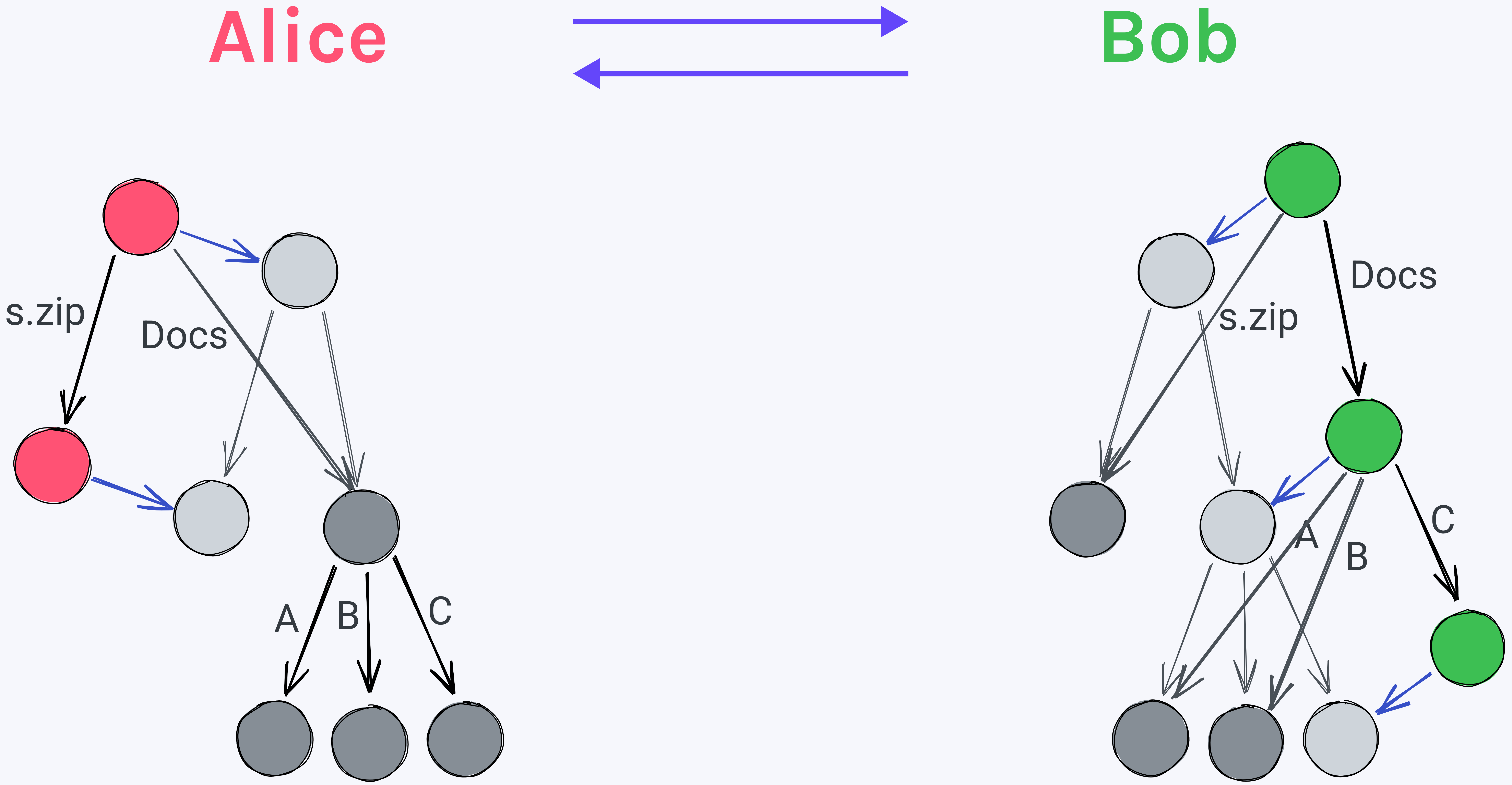


Alice

Bob

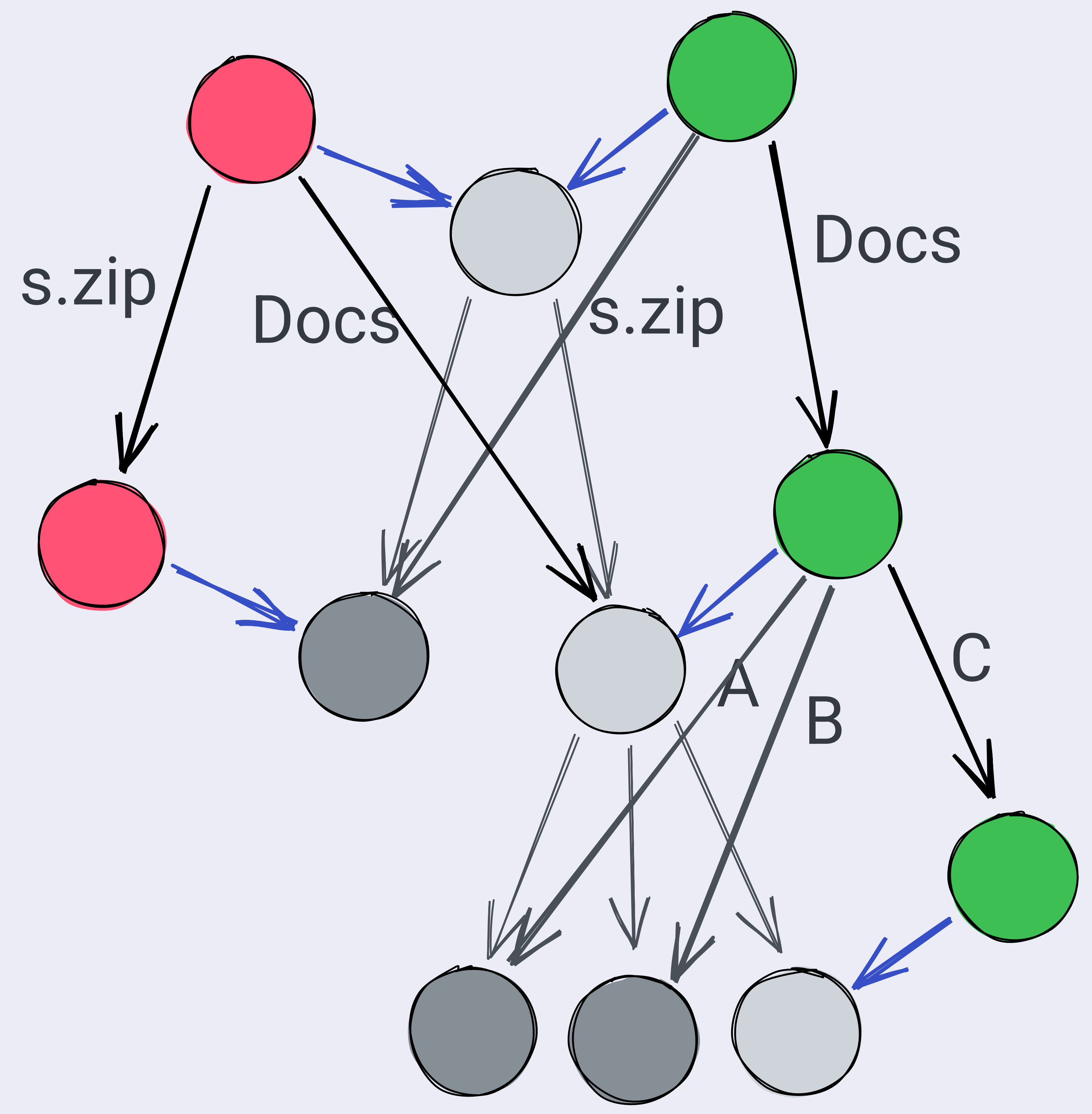





Alice
Bob
s.zip
Docs
A
B
C
Docs
s.zip
A
B
C

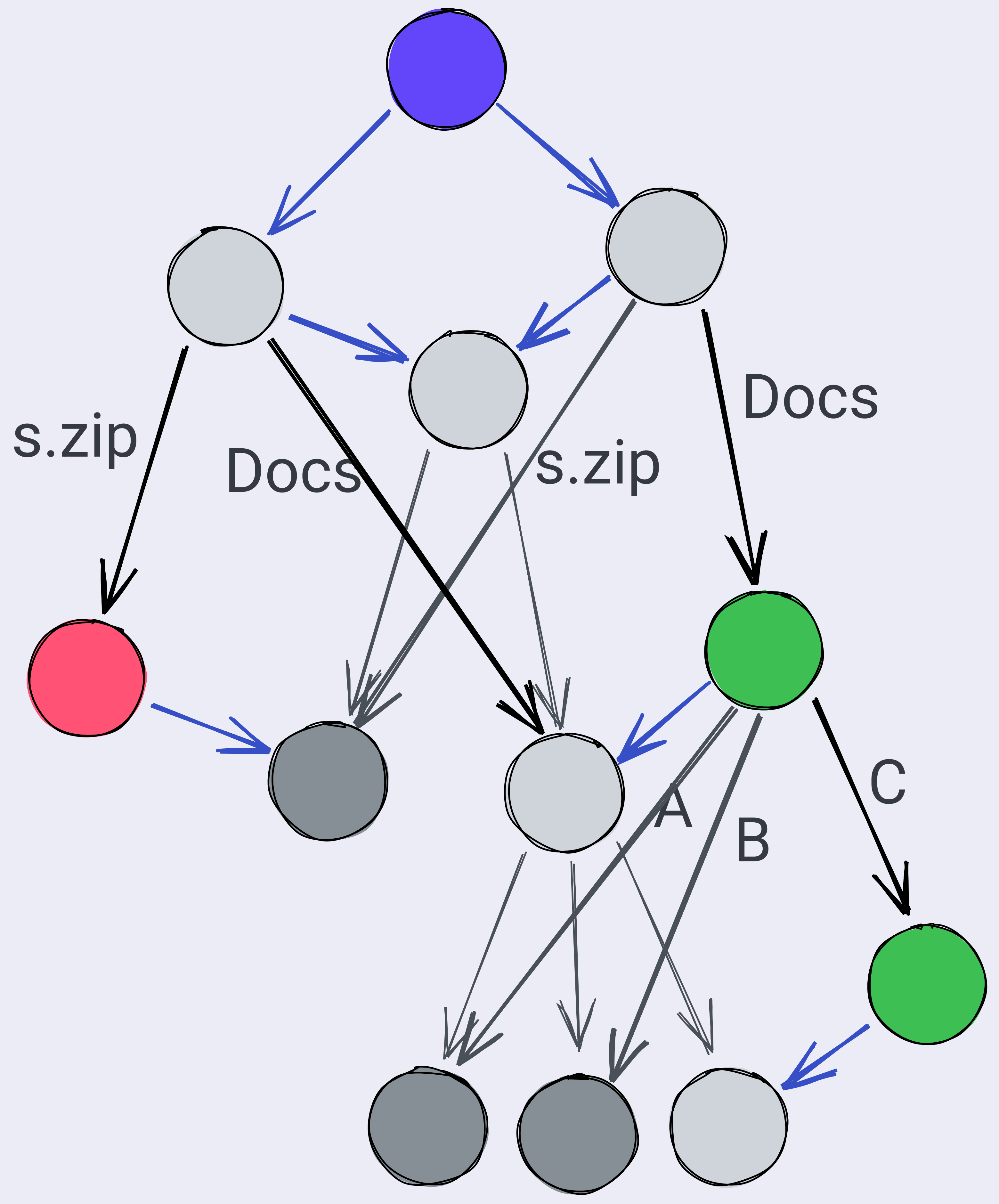
WNFS Merge

- Two (or more) roots
- Detect divergence
    - Look if one node is included in the other's DAG



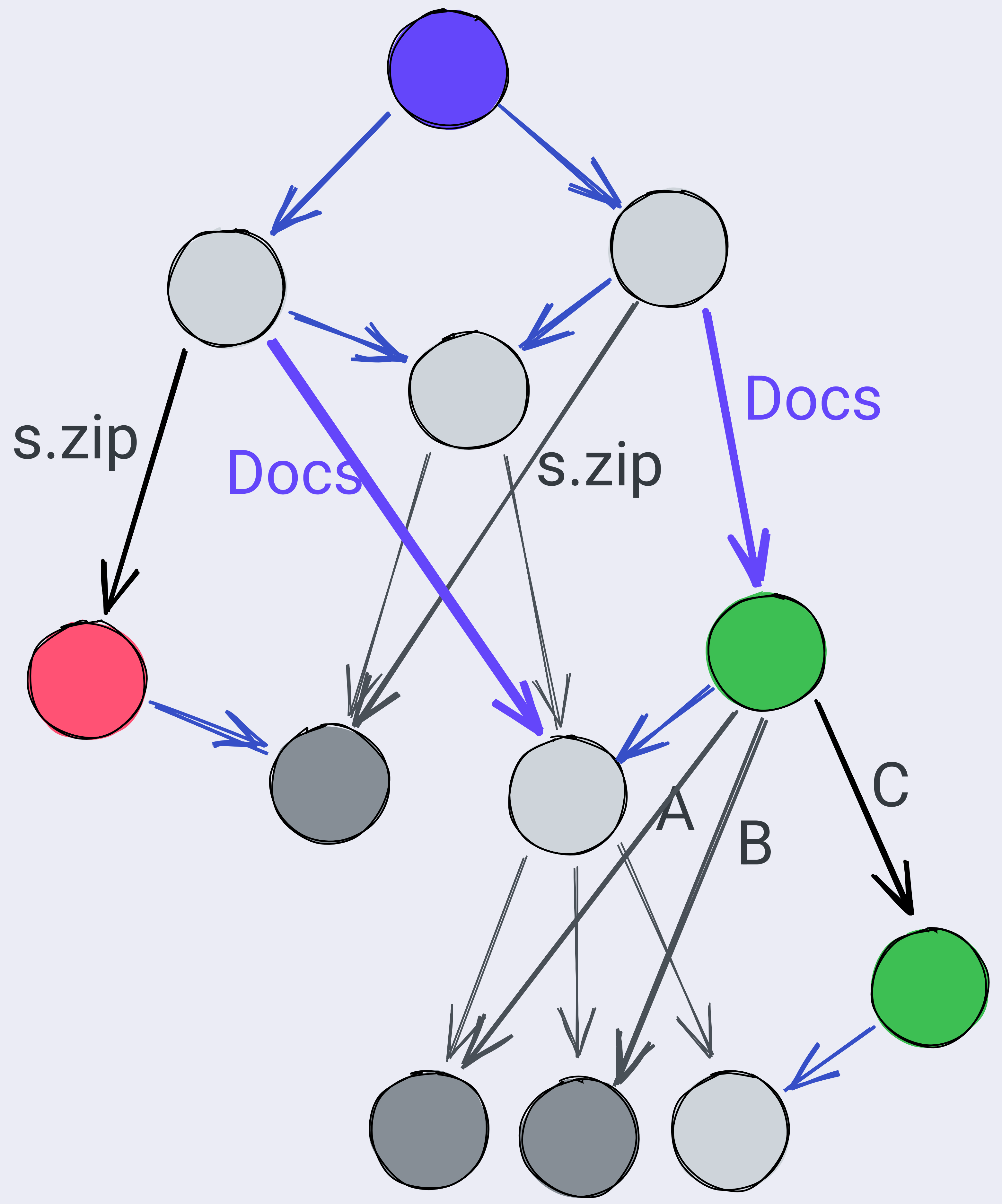
WNFS Merge

- Two (or more) roots
- Detect divergence
  - Look if one node is included in the other's DAG
- Start merging
  - Create merge node (two previous links)



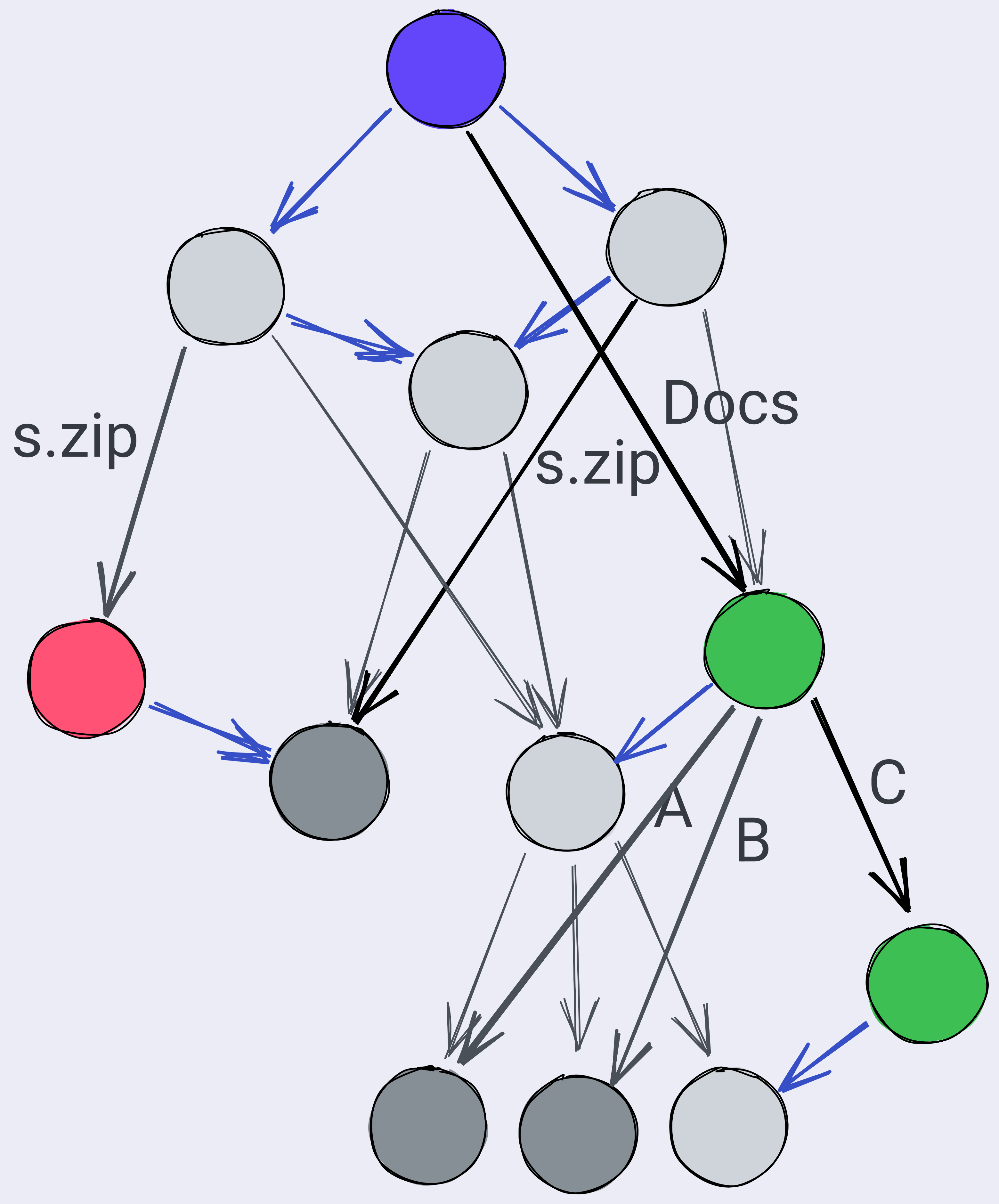
WNFS Merge

- Two (or more) roots
- Detect divergence
    - Look if one node is included in the other’s DAG
- Start merging
    - Create merge node (two previous links)
    - Recursive descent



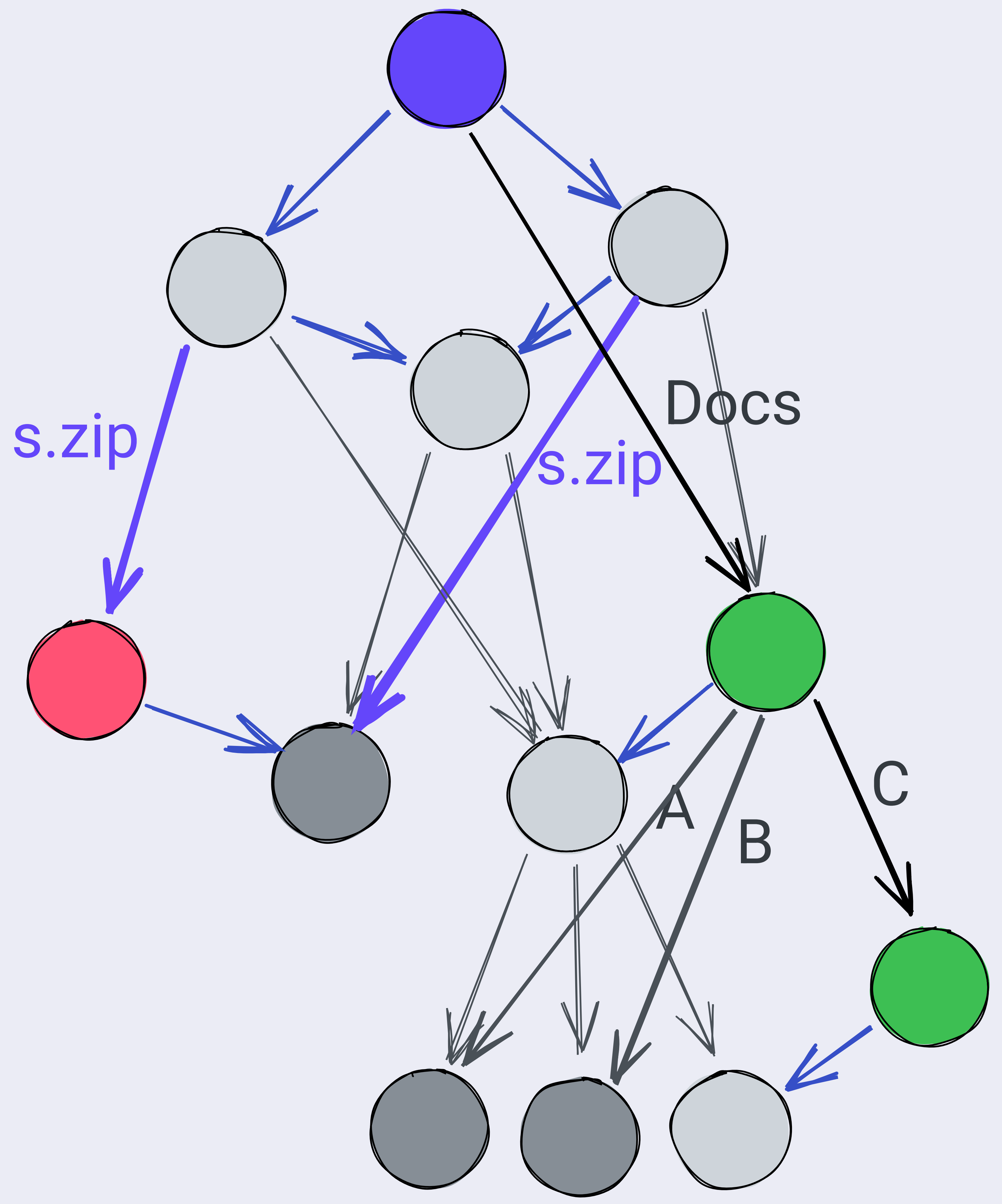
WNFS Merge

- Two (or more) roots
- Detect divergence
  - Look if one node is included in the other's DAG
- Start merging
  - Create merge node (two previous links)
  - Recursive descent
  - Short-circuit if a fast-forward is possible



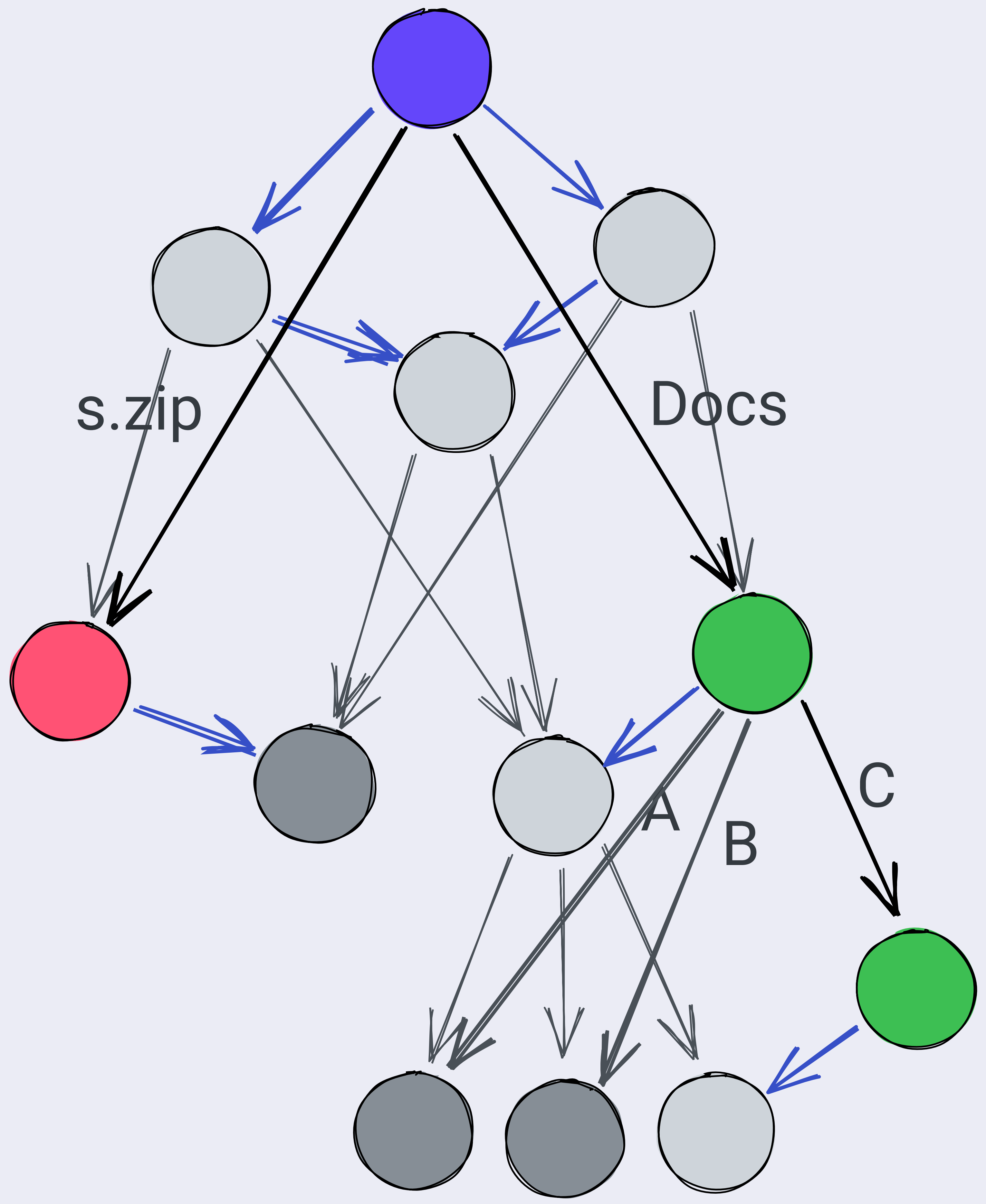
WNFS Merge

- Two (or more) roots
- Detect divergence
  - Look if one node is included in the other's DAG
- Start merging
  - Create merge node (two previous links)
  - Recursive descent
  - Short-circuit if a fast-forward is possible



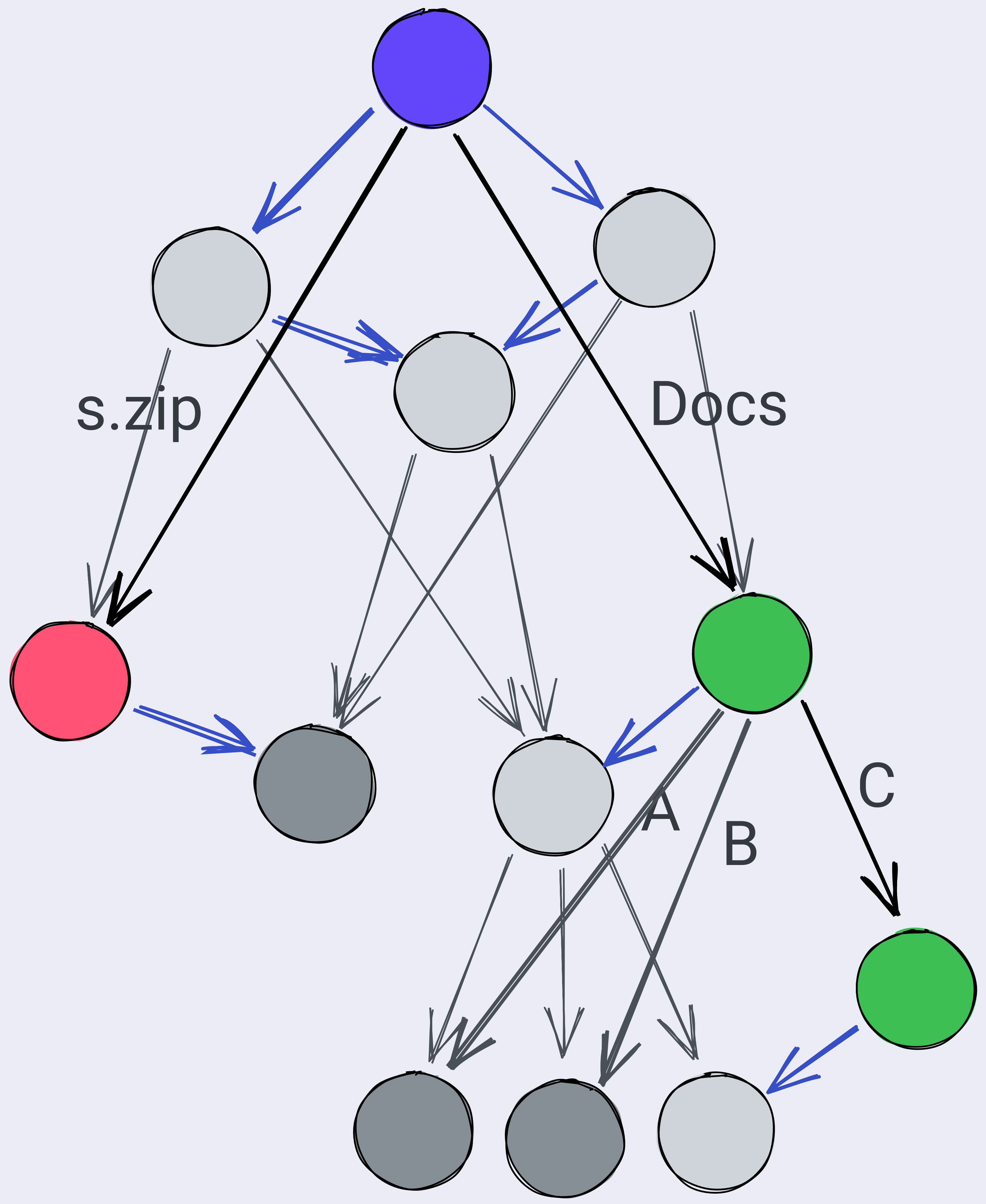
WNFS Merge

- Two (or more) roots
- Detect divergence
  - Look if one node is included in the other's DAG
- Start merging
  - Create merge node (two previous links)
  - Recursive descent
  - Short-circuit if a fast-forward is possible



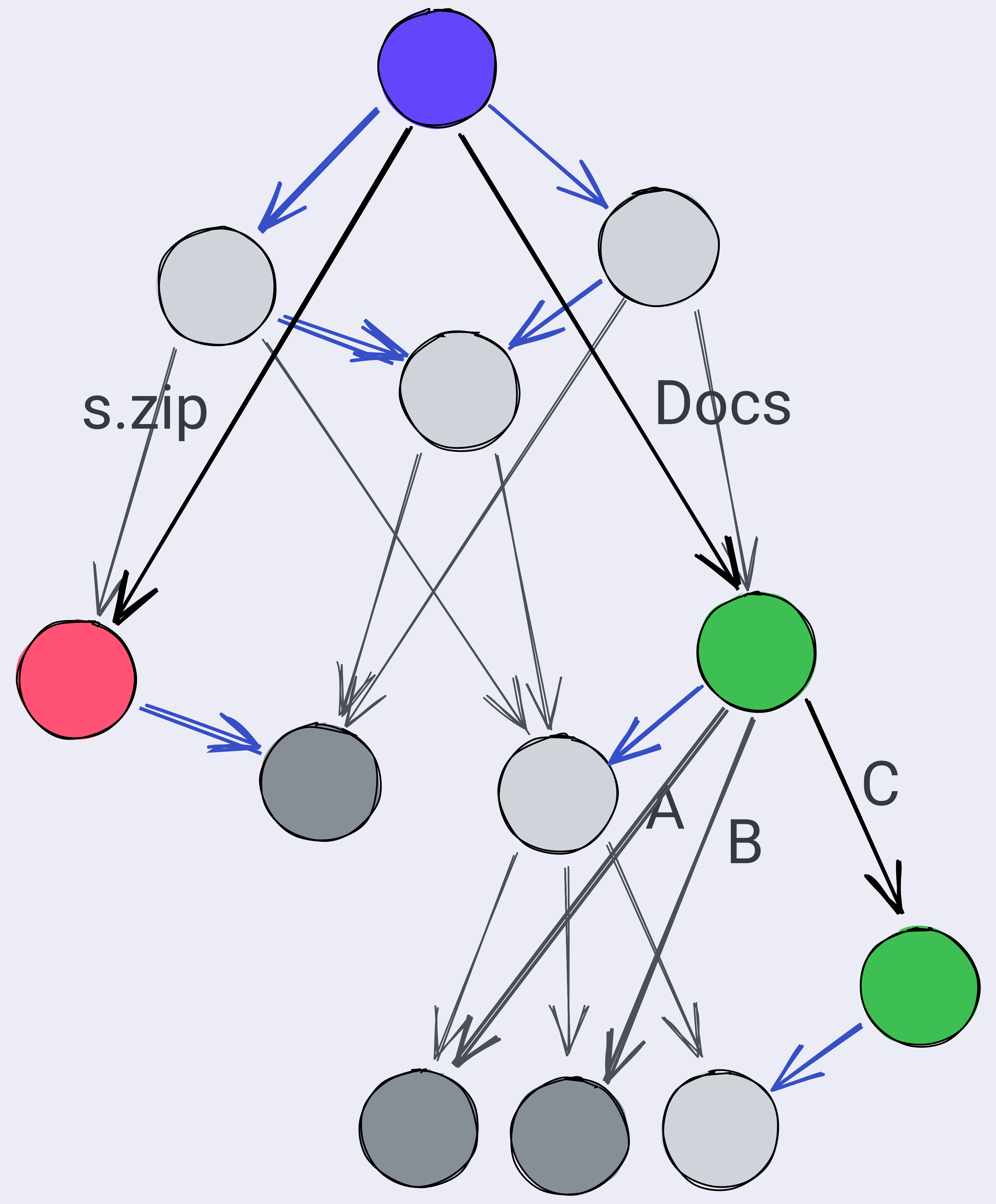
WNFS Merge

- Works similarily on the private side
- Not perfect
  - Moving & modifying concurrently result in two copies
  - Conflicts on files need to be handled “by coin flip” (lower hash wins)
- (I’m leaving out some details)



WNFS Merge

Immutable internal data structures make working with multiple trees at the same time easier



Consider Local Concurrency

You're in a browser and a button causes WNFS changes.
Congratulations, you need to care about local concurrency!

Local Concurrency

- WNFS operations are async
  - That’s a good thing! Not blocking UI thread
  - Non-async is impossible: WebCrypto API is async
- You *could* solve this using WNFS merge
  - But exploiting local context can give better results!

Local Concurrency

- Transactional API
  - Each transaction builds its own WNFS tree (isolation)

```
const fs = // ...

await fs.transaction(async tx ⇒ {
  // this will be re-run if conflicts are detected
  await tx.write("public/a/b/file.txt", "Hello, World!")
  const num = parseInt(await tx.read("private/number.txt"))
  await tx.write("private/number.txt", (num * 2).toFixed(2))
})
```

Local Concurrency

- Transactional API
  - Each transaction builds its own WNFS tree (isolation)
- Software Transactional Memory
  - Keep track of what nodes were read/written
  - Re-run transactions if reads are invalidated
  - Conflict-free transactions can be stichted together

```
const fs = // ...

await fs.transaction(async tx ⇒ {
  // this will be re-run if conflicts are detected
  await tx.write("public/a/b/file.txt", "Hello, World!")
  const num = parseInt(await tx.read("private/number.txt"))
  await tx.write("private/number.txt", (num * 2).toFixed(2))
})
```

- read "public/a/b/file.txt"
- read "private/number.txt"
- wrote "private/number.txt"

Local Concurrency

- Transactional API
  - Each transaction builds its own WNFS tree (isolation)
- Software Transactional Memory
  - Keep track of what nodes were read/written
  - Re-run transactions if reads are invalidated
  - Conflict-free transactions can be stichted together
- Exploit things you can do locally

```
const fs = // ...

await fs.transaction(async tx ⇒ {
  // this will be re-run if conflicts are detected
  await tx.write("public/a/b/file.txt", "Hello, World!")
  const num = parseInt(await tx.read("private/number.txt"))
  await tx.write("private/number.txt", (num * 2).toFixed(2))
})
```

- read "public/a/b/file.txt"
- read "private/number.txt"
- wrote "private/number.txt"

BlockStore & PrivateStore Abstractions

BlockStore

- Abstracts side effects

```
type CodecID = { code: number; name: string }

interface BlockStore {
  getBlock(cid: CID):
    Promise<Uint8Array | null>

  putBlock(bytes: Uint8Array, codec: CodecID):
    Promise<CID>
}
```

BlockStore

- Abstracts side effects
- One property:
  - You can get what you’ve put

```
type CodecID = { code: number; name: string }

interface BlockStore {
  getBlock(cid: CID):
    Promise<Uint8Array | null>

  putBlock(bytes: Uint8Array, codec: CodecID):
    Promise<CID>
}
```

BlockStore

- Abstracts side effects
- One property:
    - You can get what you've put
- Doesn't handle chunking (!)
- Implementations could be
    - Retrieving from memory
    - Retrieving from IndexedDB
    - Retrieving from Bitswap

```
type CodecID = { code: number; name: string }

interface BlockStore {
  getBlock(cid: CID):
    Promise<Uint8Array | null>

  putBlock(bytes: Uint8Array, codec: CodecID):
    Promise<CID>
}
```

BlockStore

- Abstracts side effects
- One property:
  - You can get what you've put
- Doesn't handle chunking (!)
- Implementations could be
  - Retrieving from memory
  - Retrieving from IndexedDB
  - Retrieving from Bitswap
- (Similar abstractions exist in in js-ipfs & go-ipfs)

```
type CodecID = { code: number; name: string }

interface BlockStore {
  getBlock(cid: CID):
    Promise<Uint8Array | null>

  putBlock(bytes: Uint8Array, codec: CodecID):
    Promise<CID>
}
```

BlockStore

- Abstracts side effects
- One property:
    - You can get what you've put
- Doesn't handle chunking (!)
- Implementations could be
    - Retrieving from memory
    - Retrieving from IndexedDB
    - Retrieving from Bitswap
- (Similar abstractions exist in in js-ipfs & go-ipfs)
- BlockStores compose!

```
type CodecID = { code: number; name: string }

interface BlockStore {
  getBlock(cid: CID):
    Promise<Uint8Array | null>

  putBlock(bytes: Uint8Array, codec: CodecID):
    Promise<CID>
}
```

BlockStore

- BlockStores compose!

```
type CodecID = { code: number; name: string }

interface BlockStore {
  getBlock(cid: CID):
    Promise<Uint8Array | null>

  putBlock(bytes: Uint8Array, codec: CodecID):
    Promise<CID>
}

function inMemoryBlockStore(base: BlockStore) {
  const map = {}
  return {
    async getBlock(cid) {
      return map[cid] || await base.getBlock(cid)
    },
    async putBlock(bytes, codec) {
      const cid = new CID(hash(bytes), codec)
      map[cid] = bytes
      return cid
    },
    async commitToBase() {
      for (const [cid, bytes] of Object.entries(map)) {
        await base.putBlock(bytes, cid.codec)
      }
    }
  }
}
```

BlockStore

- BlockStores compose!
  - Reads propagate
  - Writes don’t immediatly propagate

```
type CodecID = { code: number; name: string }

interface BlockStore {
  getBlock(cid: CID):
    Promise<Uint8Array | null>

  putBlock(bytes: Uint8Array, codec: CodecID):
    Promise<CID>
}

function inMemoryBlockStore(base: BlockStore) {
  const map = {}
  return {
    async getBlock(cid) {
      return map[cid] || await base.getBlock(cid)
    },
    async putBlock(bytes, codec) {
      const cid = new CID(hash(bytes), codec)
      map[cid] = bytes
      return cid
    },
    async commitToBase() {
      for (const [cid, bytes] of Object.entries(map)) {
        await base.putBlock(bytes, cid.codec)
      }
    }
  }
}
```

BlockStore

- BlockStores compose!
  - Reads propagate
  - Writes don’t immediatly propagate

```
const ipfsBlockStore = // ...

// Manage your side-effects
const tempBlockStore =
  inMemoryBlockStore(ipfsBlockStore)
const newRootCID = await wnfs.write(
  currentRootCID,
  tempBlockStore
)

// now commit your block store
tempBlockStore.commitToBase()
// or just throw it away
```

BlockStore

- BlockStores compose!
  - Reads propagate
  - Writes don’t immediatly propagate
- Applications include:
  - Tiered caches
  - Logging
  - Isolation
  - Testing
  - WNFS in WASM

```
type CodecID = { code: number; name: string }

interface BlockStore {
  getBlock(cid: CID):
    Promise<Uint8Array | null>

  putBlock(bytes: Uint8Array, codec: CodecID):
    Promise<CID>
}
```

PrivateStore

- Like a BlockStore, but for encrypted data
- Not indexed by CID, but by namefilters
- Can be composed like BlockStores

```
interface PrivateRef {
  key: SymmetricKey
  name: PrivateName
}

interface PrivateStore {
  getBlock(ref: PrivateRef):
    Promise<Uint8Array | null>

  putBlock(ref: PrivateRef, plaintext: Uint8Array):
    Promise<void>
}
```

PrivateStore

- Like a BlockStore, but for encrypted data
- Not indexed by CID, but by namefilters
- Can be composed like BlockStores
- **PrivateNames** can be someting abstract
    - Avoid cryptographic accumulator construction

```
interface PrivateRef {
  key: SymmetricKey
  name: PrivateName
}

interface PrivateStore {
  getBlock(ref: PrivateRef):
    Promise<Uint8Array | null>

  putBlock(ref: PrivateRef, plaintext: Uint8Array):
    Promise<void>
}
```

PrivateStore

- Like a BlockStore, but for encrypted data
- Not indexed by CID, but by namefilters
- Can be composed like BlockStores
- **PrivateNames** can be someting abstract
  - Avoid cryptographic accumulator construction
- Can locally skip encryption

```
interface PrivateRef {
  key: SymmetricKey
  name: PrivateName
}

interface PrivateStore {
  getBlock(ref: PrivateRef):
    Promise<Uint8Array | null>

  putBlock(ref: PrivateRef, plaintext: Uint8Array):
    Promise<void>
}
```

WNFS in WASM

WNFS in WASM

- WASM is observationally pure

WNFS in WASM

- WASM is observationally pure
- Idea: Implement algorithms & WNFS DAG surgery in WASM
- Dependency-inject Network & Storage using the BlockStore

WNFS in WASM

- WASM is observationally pure
- Idea: Implement algorithms & WNFS DAG surgery in WASM
- Dependency-inject Network & Storage using the BlockStore
- Problem: (only applies to Browsers)
    - BlockStore methods are async
    - WASM function imports don’t support async functions natively
    - Lots of complexity for “conceptually synchronous” operations

WNFS in WASM

- WASM is observationally pure
- Idea: Implement algorithms & WNFS DAG surgery in WASM
- Dependency-inject Network & Storage using the BlockStore
- Problem: (only applies to Browsers)
  - BlockStore methods are async
  - WASM function imports don't support async functions natively
  - Lots of complexity for "conceptually synchronous" operations
- Solution
  - Put WASM into WebWorker
  - Write a small JS shell around WASM
  - Turn asynchronous BlockStore calls from UI Worker into synchronous calls using **`SharedArrayBuffers`** and **`Atomics.wait`**

Conclusion

Conclusion

- Separate functional core from mutable shell

Conclusion

- Separate functional core from mutable shell
  - Lots of complicated, pure functions that ingest DAG(s) and produce a DAG:
    - DAG surgery: `write`, `mkdir`, WNFS merge
    - Namefilters, encryption, skip ratchet

Conclusion

- Separate functional core from mutable shell
  - Lots of complicated, pure functions that ingest DAG(s) and produce a DAG:
    - DAG surgery: `write`, `mkdir`, WNFS merge
    - Namefilters, encryption, skip ratchet
  - Mutable shell handles:
    - BlockStore implementations: Networking, Storage
    - Key management
    - Root WNFS pointer

Conclusion

- Separate functional core from mutable shell
  - Lots of complicated, pure functions that ingest DAG(s) and produce a DAG:
    - DAG surgery: `write`, `mkdir`, WNFS merge
    - Namefilters, encryption, skip ratchet
  - Mutable shell handles:
    - BlockStore implementations: Networking, Storage
    - Key management
    - Root WNFS pointer
- WASM lends itself well for the functional core
- I hope you learned something about WNFS today!

Links

- https://whitepaper.fission.codes/file-system/file-system-basics
- WNFS v2 prototype branch: https://github.com/fission-suite/webnative/tree/matheus23/wnfs2
- wnfs-go WNFS v2 implementation: https://github.com/qri-io/wnfs-go
  (will all eventually move to a wnfs-wg github org)
- WASM worker experimentation: https://github.com/matheus23/gca-rust/blob/8de902d052e8168b1809f108a63c94f539083ba7/js/worker.js