Blockchain

Last Time

- Coding SHA-256 and RSA
- Certificates
- Goals of security
  - CIA Triad (Confidentiality, Integrity, Availability)
  - Authentication, Authorization, and Least Privilege
- Examples
  - HTTPS: Confidentiality, Integrity, and server Authentication through certificates and symmetric encryption
  - File permissions: Authentication and Authorization through users, groups, and 9-bit permission flags

Learning objectives

- Finish up file permissions
- Introduce blockchain (enough to both understand and implement)

Part 1: more on file permissions

File permissions review

- 9-bit flag in 3 3-bit parts: `uuugggooo` and `rwxrwxrwx`
    - `u` if process owner $=$ file's owning user
    - `g` if process owner $\in$ file's owning group
    - `o` otherwise
- Files:
    - `r` = see its bytes
    - `w` = change its bytes
    - `x` = execute like a program (if and only if also have `r`)
- Directories:
    - `r` = see what it contains
    - `w` = change what it contains (including renaming contents)
    - `x` = navigate through it to other files/directories

`sudo`, root, and kernel mode

- Permissions are handled by
  1. User code executes a `syscall`, switching to kernel mode
  2. Kernel code compares the process's owner and the requested action to the permission bits
  3. If permitted, kernel does the action
  4. Kernel switches back to user mode and code
- For the superuser `root`, step 2 always resolves to "permitted"
- `su` and `sudo` request changing the user of the current (`su`) or new (`sudo`) process
  - Some users have this permission, others do not
  - Extra password to keep process from doing this without your knowledge

File permissions and least privilege

Least Privilege
  A party granted rights to do x should not also get rights to do y

Can Unix-style permissions be used to implement the principle of least privilege (w.r.t. files)?

- If **yes**, explain scenario with all needed, no unneeded rights
- If **no**, explain scenario where needed rights come with unneeded rights

Example scenarios:

- All CS faculty and student have accounts on shared computer
- Need to run an application but don’t trust its creator to be non-malicious
- Shared file server for 100-employee tech company

Part 2: blockchain

Goal of first successful blockchain

Digital cash with no managing entity

- **Anonymous**: no connection between online account and physical person
- **Unforgeable**: brute force cannot break system
- **Single-owner**: coins move, not copied
- **Decentralized**: many computers share control, no one in charge

Anonymous and Unforgeable

Anonymous:

- User ID = a public key
- One human can have any number of User IDs

Unforgeable:

- Cryptographic signatures and hashes

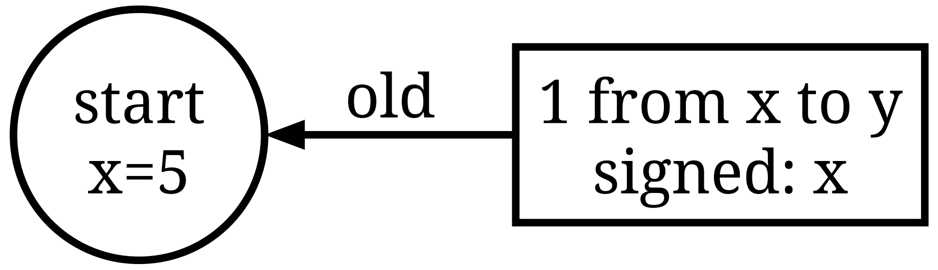
Single-owner, move not copy

- Owner *signs* a document saying “I’m giving this coin to x”
- Proof I had coin to give?
  - Chain of documents back to coin creation
- Proof I didn’t give same coin to two different people?
  - A complete set of all documents ever produced
  - Each document identifies its previous document
  - Only one sequence of states is “correct”

Pointer-based ledger

Node = signed transfer (called a **block**)

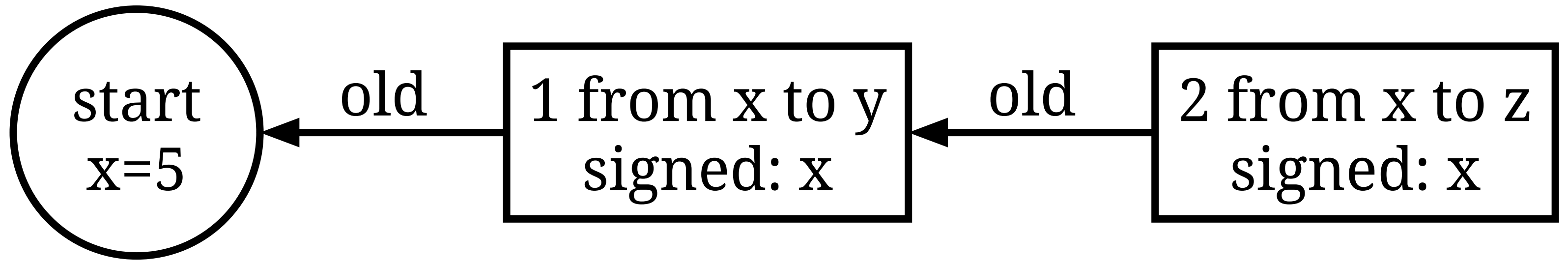
Points to previous block



Pointer-based ledger

Node = signed transfer (called a **block**)

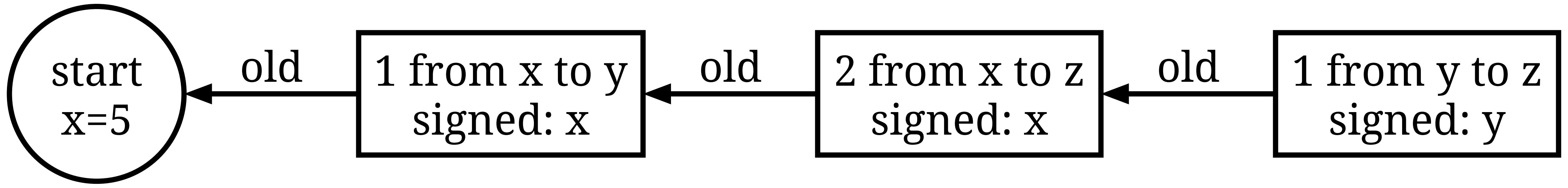
Points to previous block



Pointer-based ledger

Node = signed transfer (called a **block**)

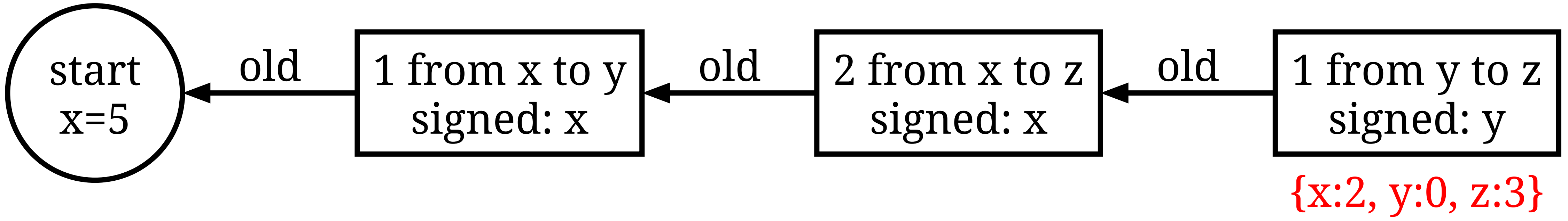
Points to previous block



Pointer-based ledger

Node = signed transfer (called a **block**)

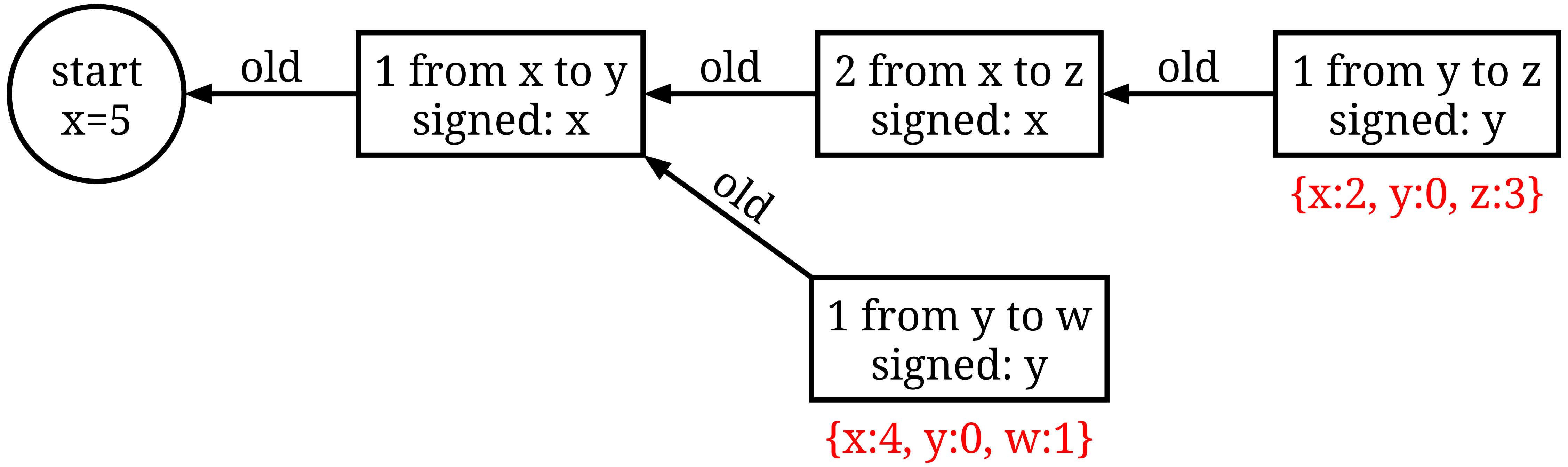
Points to previous block



Pointer-based ledger

Node = signed transfer (called a **block**)

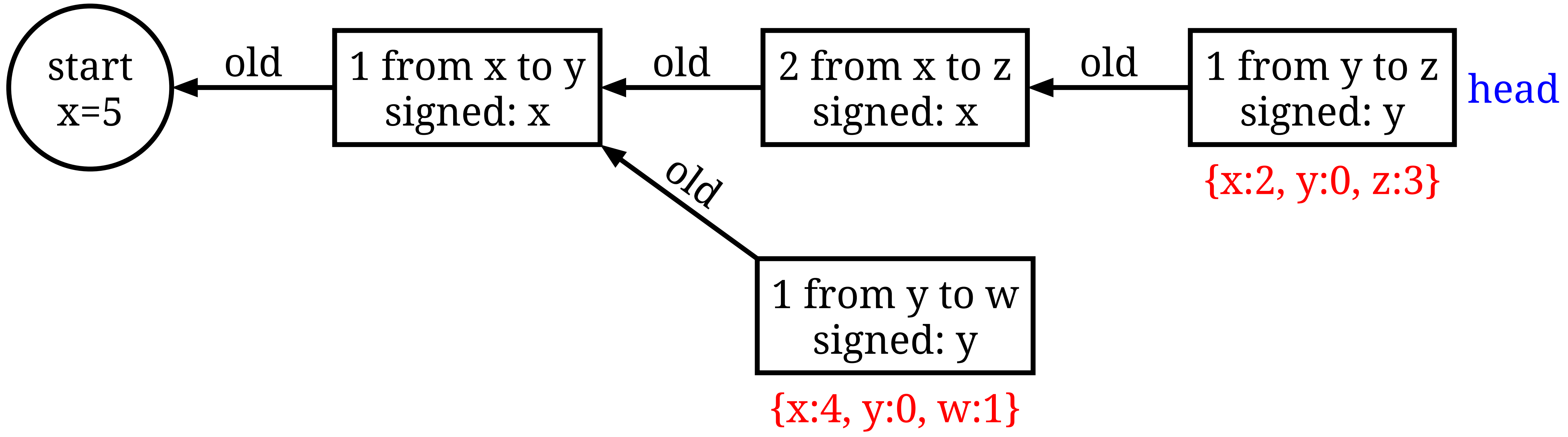
Points to previous block



Pointer-based ledger

Node = signed transfer (called a **block**)

Points to previous block

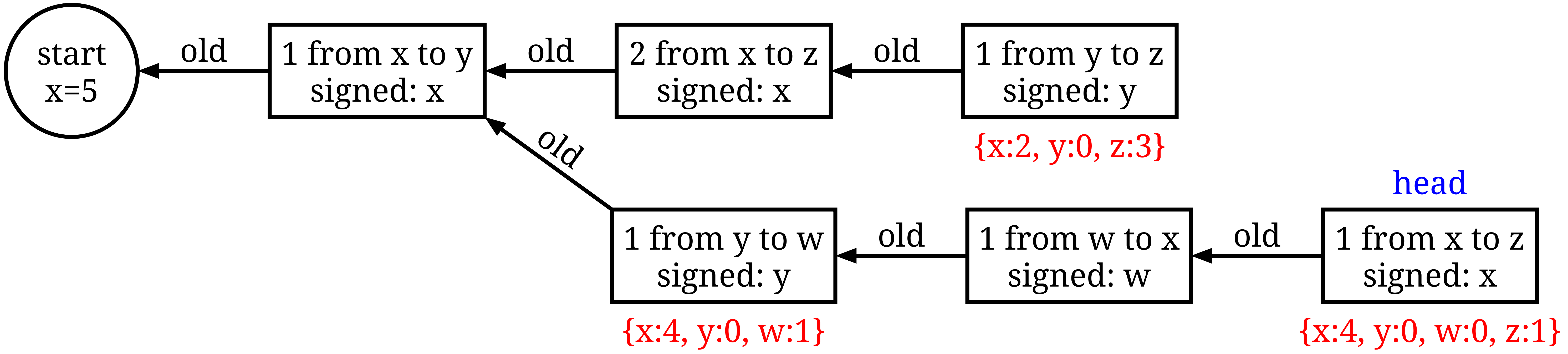


- The **head** of the chain is
  - farthest block from start (with tie-breaking rule)
- Only the state represented by the head is “real”

Pointer-based ledger

Node = signed transfer (called a **block**)

Points to previous block



- The **head** of the chain is
  - farthest block from start (with tie-breaking rule)
- Only the state represented by the head is “real”
- The head (and thus real state) can change discontinuously

Distributed pointers

- We expect *many* blocks, with many computers agreeing on state
- To check if action can happen, each computer needs the *entire* set of blocks
    - Too many to send all with each update
- Solution: communicate just the new block and its `old` pointer
    - Pointer = hash of pointed-to thing
        - *Big* hashes to ensure no collisions
    - Hash already needed for signature: re-use that hash
    - Requires that blocks never change (or else hash would change)
        - **immutable**, indelible, purely functional, persistent

A block with anonymous users might be:

```
{
    "change": {
        "old": hash_value_of_previous_block,
        "src": user1_pub, # public key = user ID for anonymity
        "dst": user2_pub,
        "n": 3, # how much to transfer from src to dst
        "memo": "text to identify the purpose of this block"
    },
    "signature": priv(hash_of(change_above), user1_priv, user1_pub)
}
```

A pointer to a block is `hash_of(block["change"])`

Our blocks will be:

```
{
    "change": {
        "old": hash_value_of_previous_block,
        "src": "user1", # for grading, you can't be anonymous
        "dst": "user2",
        "n": 3, # how much to transfer from src to dst
        "memo": "text to identify the purpose of this block"
    },
    "signature": priv(hash_of(change_above), user1_priv, user1_pub)
}
```

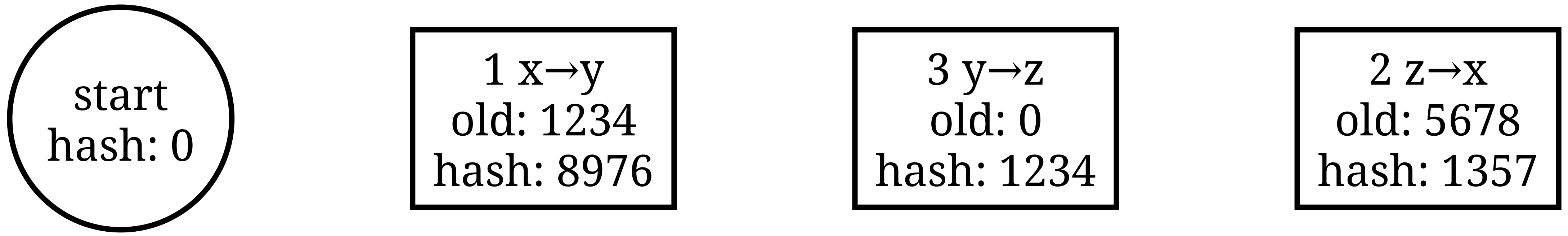
A pointer to a block is `hash_of(block["change"])`

Hashing `dict`s

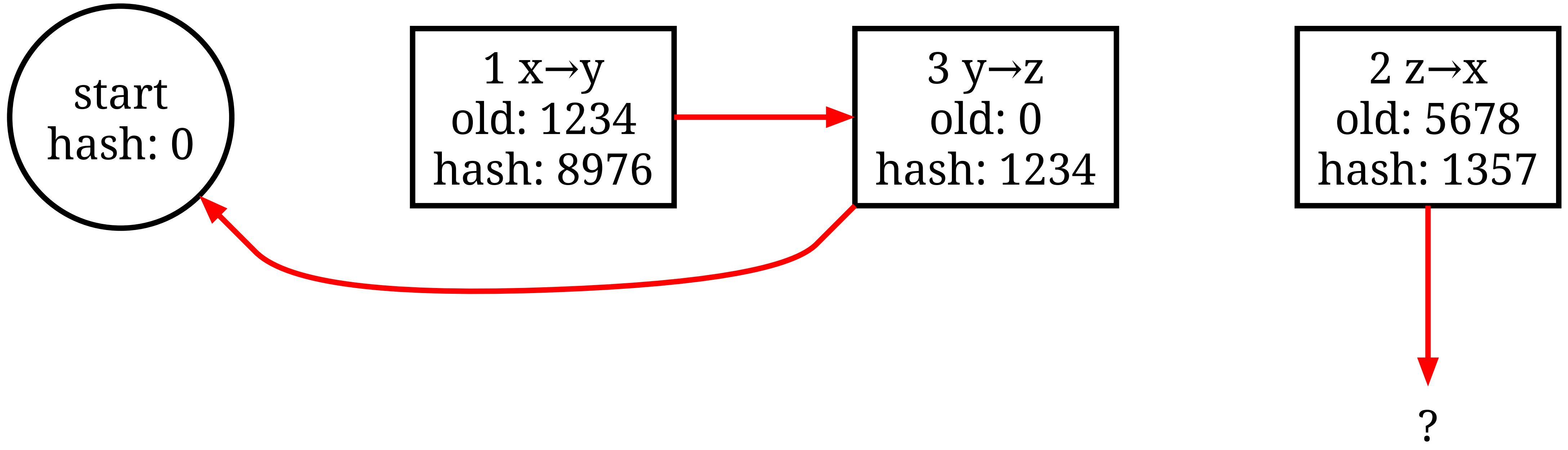
- SHA-256 can hash any `bytes`
- UTF-8 can turn any `str` to `bytes`
- JSON can turn `block["change"]` to `str`, in several ways:
  - `{"x": "y"}` vs. `{"x":"y"}`
  - `{"x": 1, "y": 2}` vs. `{"y": 1, "x": 2}`
  - `{"smile": "☺"}` vs. `{"smile": "\u263A"}`
  - For hashes to work as pointers and in signatures,
    we need just one unambiguous JSON format.

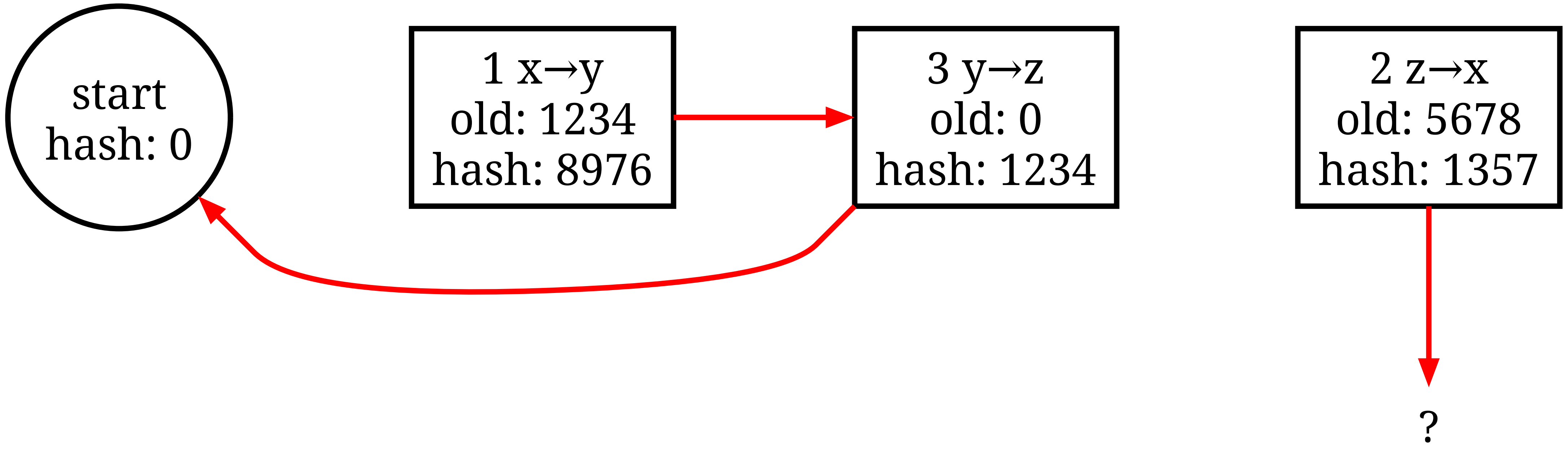
```
json.dumps(value,
    separators=(',',':'),
    indent=None,
    sort_keys=True,
    ensure_ascii=False)
```

Add arrows to this set of blocks:



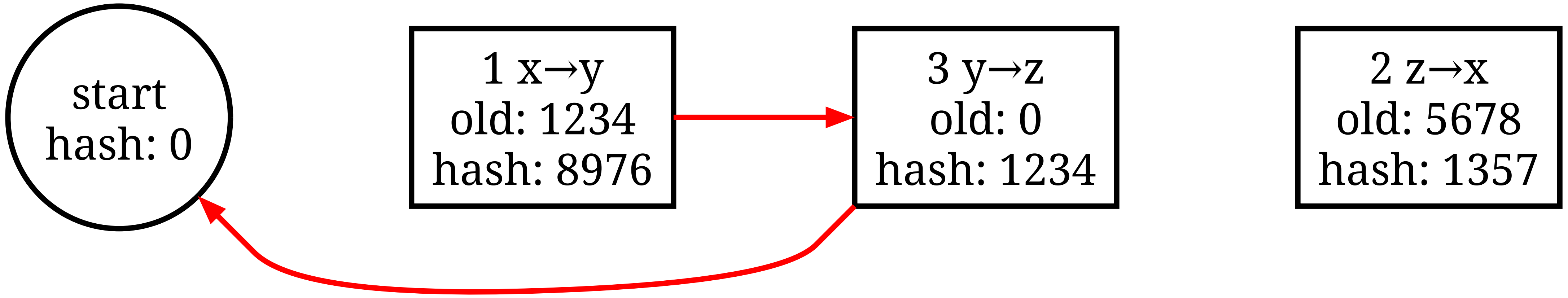
Add arrows to this set of blocks:





Can arise for two reasons:

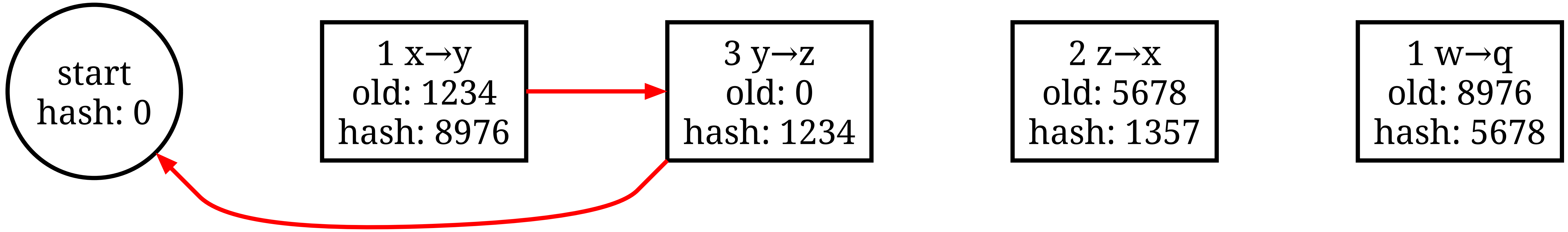
- Someone gave you bad block 1357
- Author of 1357 saw 5678 before you did



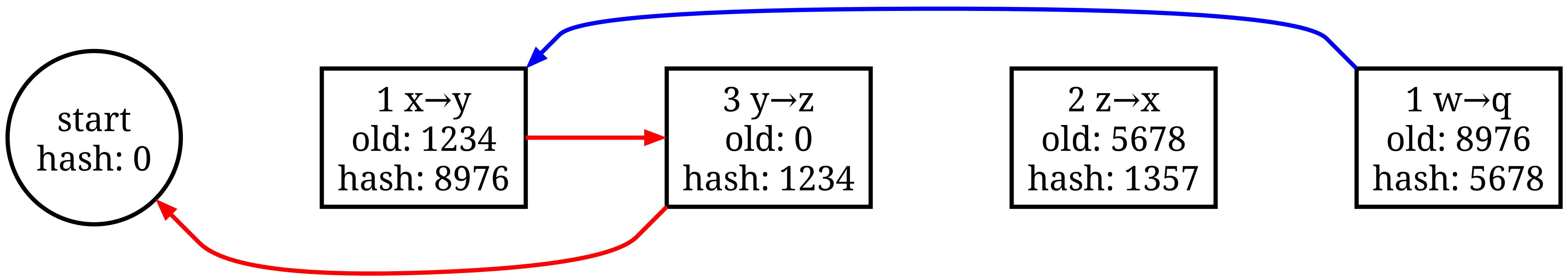
Two kinds of (valid) blocks:

- Connected via `old` pointers back to start
- Disconnected

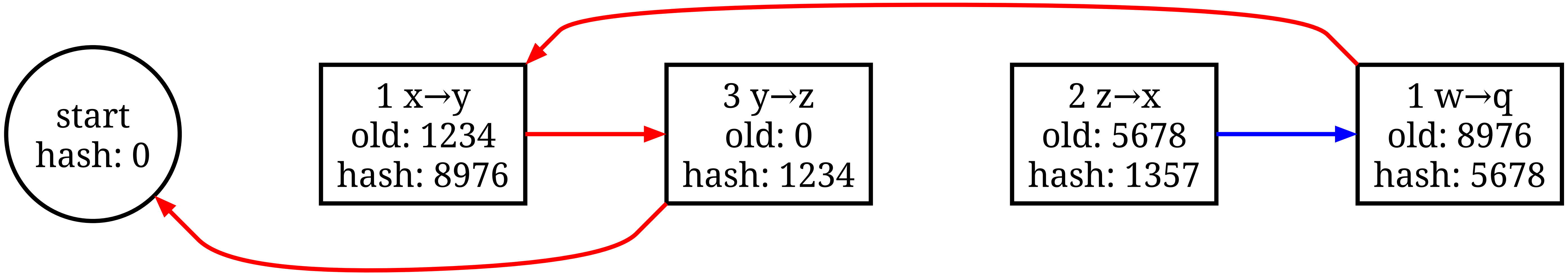
Assume a new block arrives:



Connect if the `old` is known:

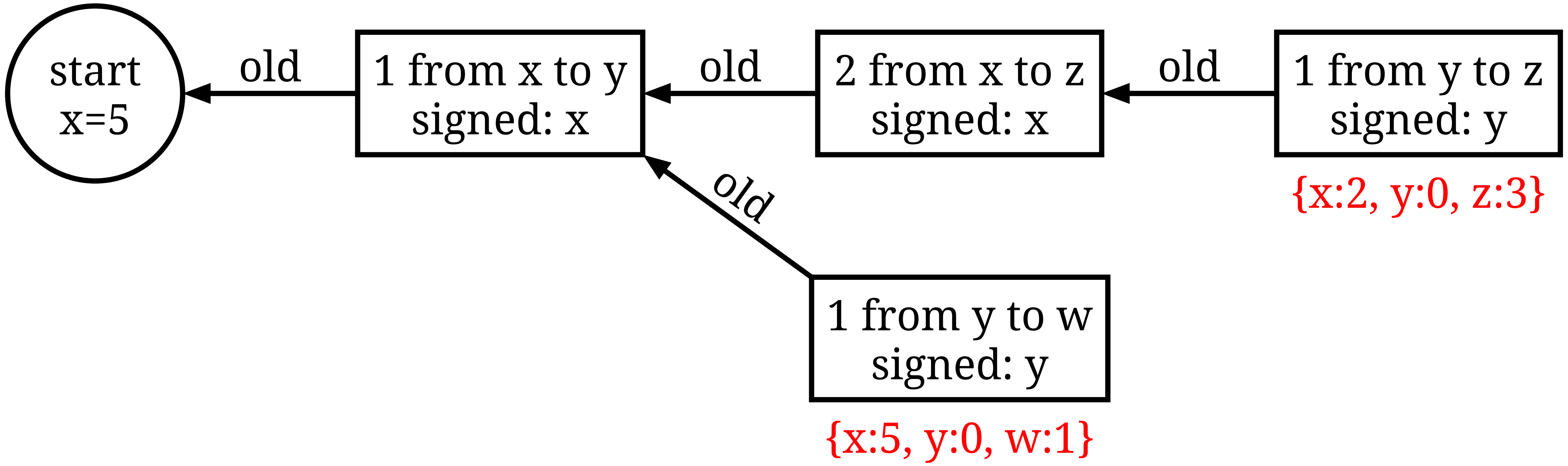


Then check for any disconnected blocks that might now be connectable



Forking

The blockchain is said to have **forked** if it isn't just one list



Causes of forking

- Malice (I pay you, then go back to a state from before the payment)
- Race conditions

Review: race conditions

From our text (emphasis added):

> In general a race condition is any time that the outcome of a program depends on the order in which different threads do different things.
>
> A common version of a race condition has the following structure:
>
> 1. Thread 1 checks some value in memory to decide what to do.
> 2. Thread 2 changes that value.
> 3. Thread 1 does the thing that the old value wanted it to do,
>   but that is no longer correct because of the change thread 2 made.

The “common version” is an example of a **data race**

Data race is impossible with blockchain – why?

- Blocks are **immutable** – there’s no state that can change.

Blockchain race condition

Users/agents on different computers, not different threads

User 1

1. create a block from head
2. add to own chain
3. send to other users
4. update head

1. recieve block
2. add to own chain
3. update head

User 2

1. create a block from head
2. add to own chain
3. send to other users
4. update head

1. recieve block
2. add to own chain
3. update head

- Can use synchronization primitives to make each box above atomic
- Cannot use synchronization primitives to control what other computers are doing
  - Forks might arise by resulting distributed race condition

Intentionally forking the blockchain

Attack:

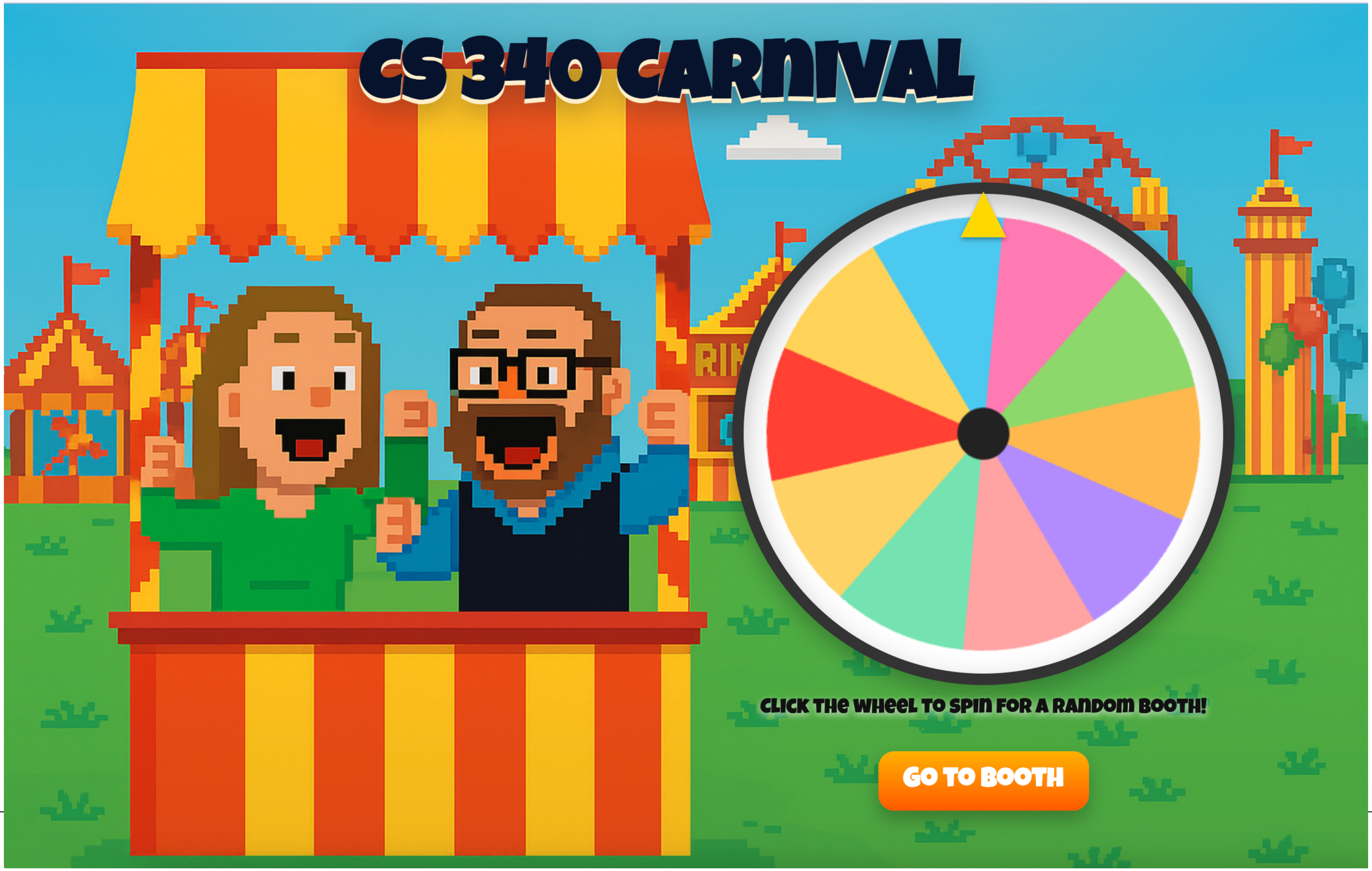
1. Pay for something on blockchain
2. Receive what you paid for
3. Add many transactions
   starting from the block *before* your payment
   to make a new fork that becomes the head

Defenses:

- **Proof of Work**: Adding a block takes hours of compute and expensive electricity
- **Proof of Stake**: Users must have large accounts to add a block
- **Non-anonymous**: Bad actors (and their forks) are banned

Our final project

- Some students wanted gambling, some did not
    - Solution: carnival



Final project design

- MP10
  - `blockchain.py` – you write, implement blockchain
  - `bc_agent.py` – we wrote, implement blockchain communication and web API
  - `configs/pub.json` and `configs/priv_`*`user`*`.json` – contains the allowed keys
    - Testing files with MP10; project files will be placed on each VM
- Project
  - `game.py` – web service part of your carnival booth
    - Sends HTTP requests to `bc_agent.py`
  - `game.html` – front-end of your carnival booth
    - Use of AI to create this file is encouraged

File	Author	Contents	Communication
`blockchain.py`	**Students (MP10)**	Blockchain logic	None
`bc_agent.py`	Staff	Blockchain API	Between agents
`configs/*.json`	Staff	Public keys, URLs, etc.	N/A
`game.py`	**Students (Project)**	Game logic	HTTP requests to `bc_agent.py`
`game.html`	**AI (Project)**	Game UI	HTTP requests to `game.py`

What backs cryptocurrency?

- Currency's value is our trust that others will value it
  - requires stability of supply
  - requires backing to bootstrap trust
- Backed by another currency
  - bank account, casino token, gold standard
- Backed by goods and services
  - gift card, arcade token, event ticket
- Backed by force
  - government taxes/fines, criminal ransom/protection/blackmail

Once backed, strengthens through use and speculation