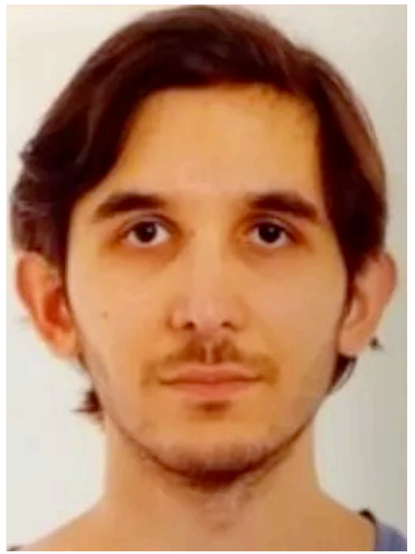
Timestamping through a Data-Structure Lens

Umang Mathur

Joint Work with Awesome Collaborators






A Tree Clock Data Structure for Causal Orderings in Concurrent Executions

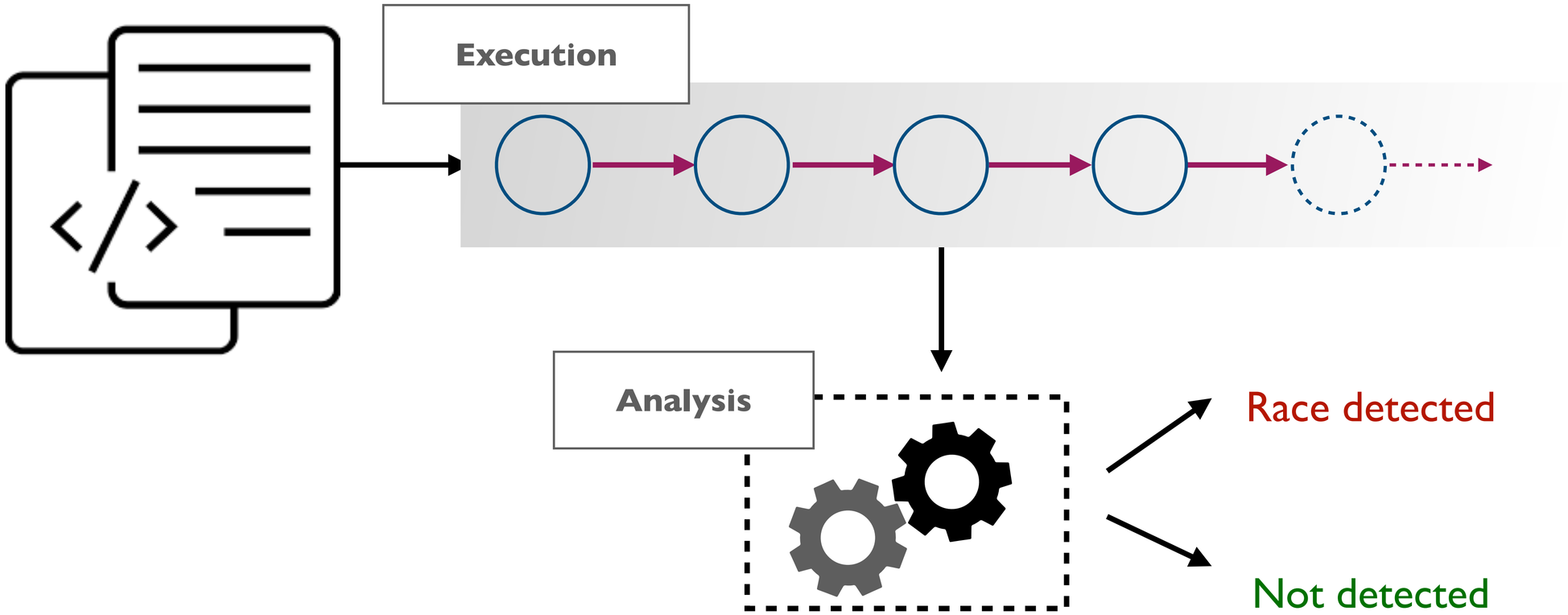
Umang Mathur
National University of Singapore
Singapore
umathur@comp.nus.edu.sg

Andreas Pavlogiannis
Aarhus University
Denmark
pavlogiannis@cs.au.dk

Hünkar Can Tunç
Aarhus University
Denmark
tunc@cs.au.dk

Mahesh Viswanathan
University of Illinois at Urbana-Champaign
USA
vmahesh@illinois.edu

What do I do?



Dynamic Analysis for Detecting Concurrency Bugs

Causality in Concurrent Computations

Causality in Concurrent Computations

Are two given events causally related?

Causality in Concurrent Computations

Are two given events causally related?

> **Causality—determining which event happens before what others—is vital in distributed computations.**

Causality in Concurrent Computations

Are two given events causally related?

> **Causality—determining which event happens before what others—is vital in distributed computations.**

Distributed Storage (Dynamo)

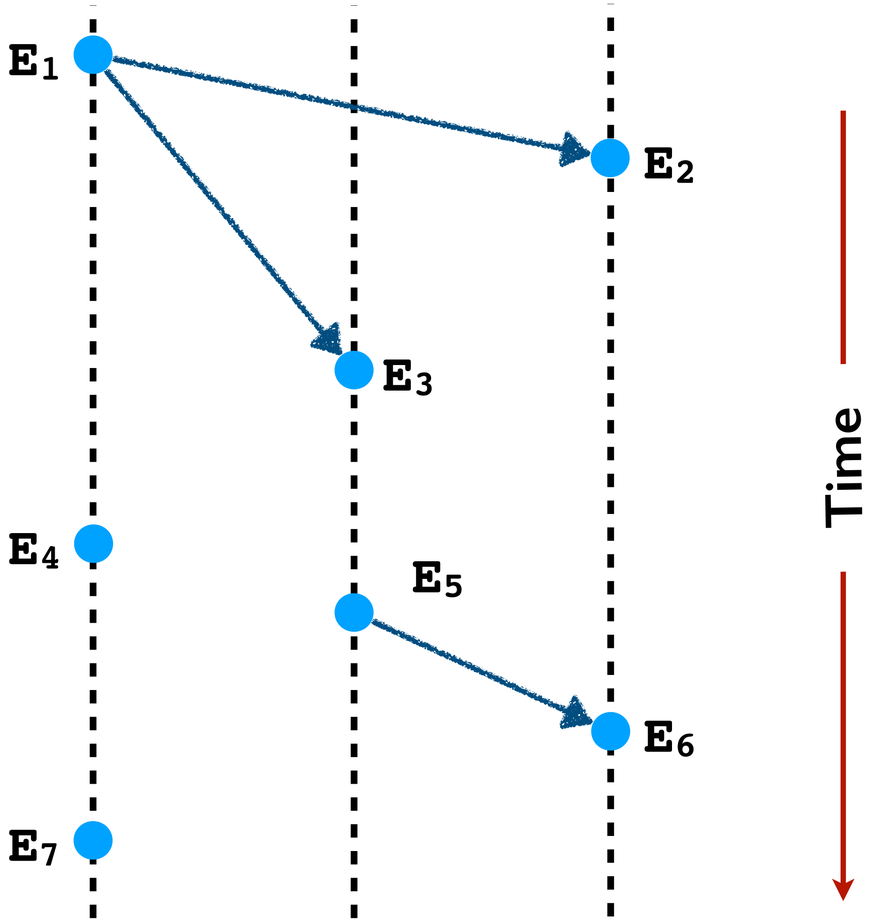
Debugging and Visualization

Program Analysis for Bugs Detection

Happens-Before for Capturing Causality

Happens-Before for Capturing Causality

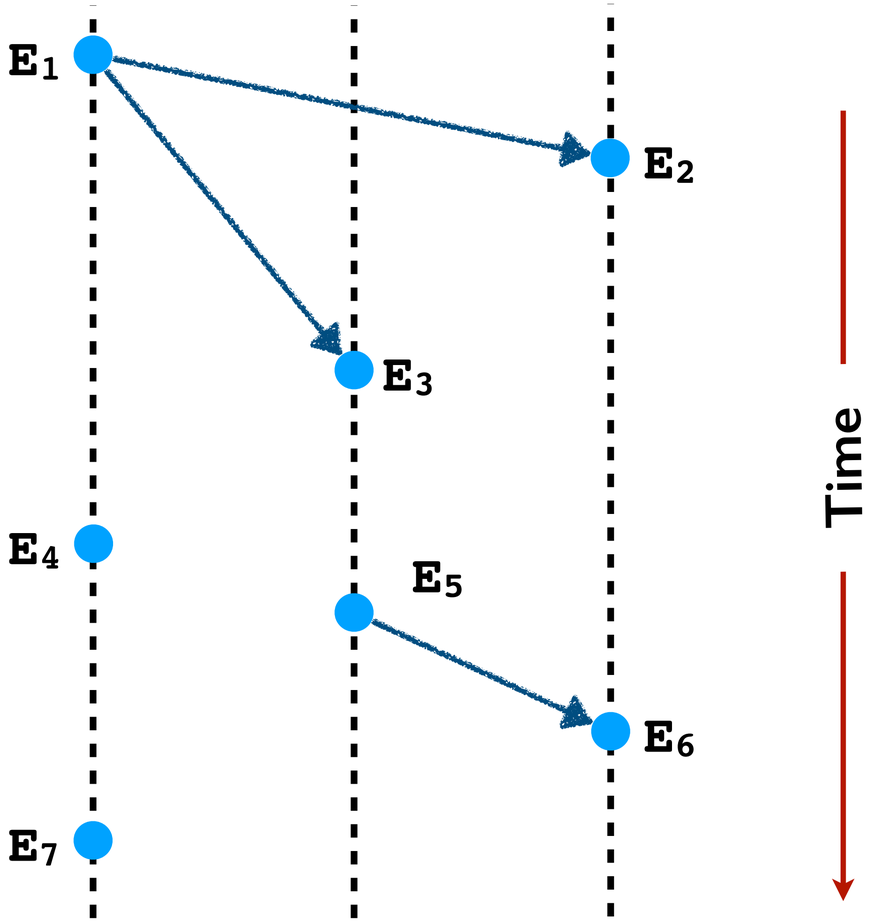
Are two given events causally related?



Happens-Before for Capturing Causality

Are two given events causally related?

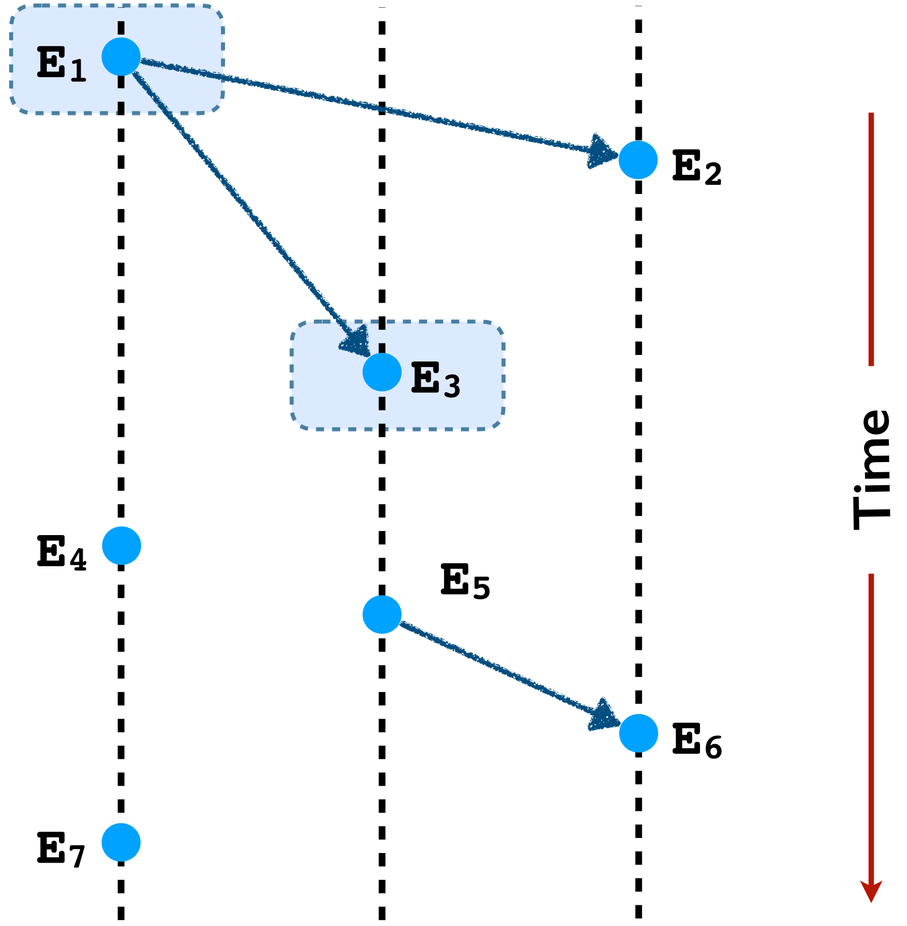
- Happens-before ($<_{HB}$) *partial order* on events
- $E_i <_{HB} E_j$ ("E_i happens before E_j") if
  - E_i and E_j are in the same process
  - E_j receives the message that E_i sends
  - transitively ($E_i <_{HB} E_k$ and $E_k <_{HB} E_j$)



Happens-Before for Capturing Causality

Are two given events causally related?

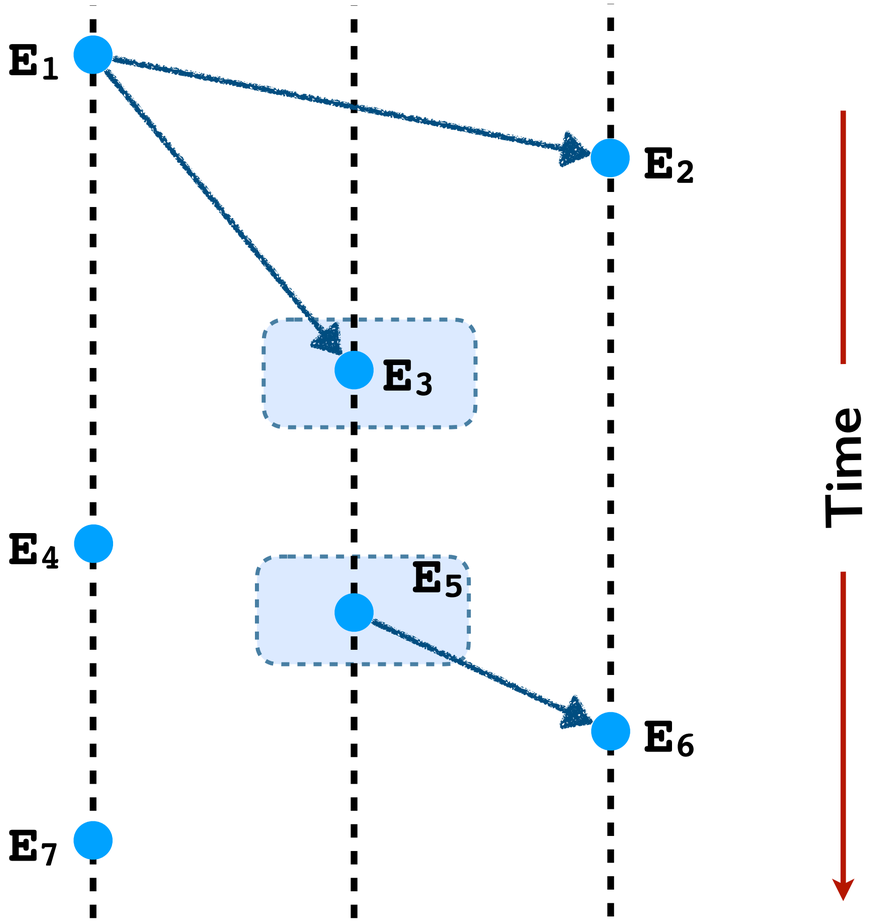
- $E_1 <_{HB} E_3$ (send-receive)



Happens-Before for Capturing Causality

Are two given events causally related?

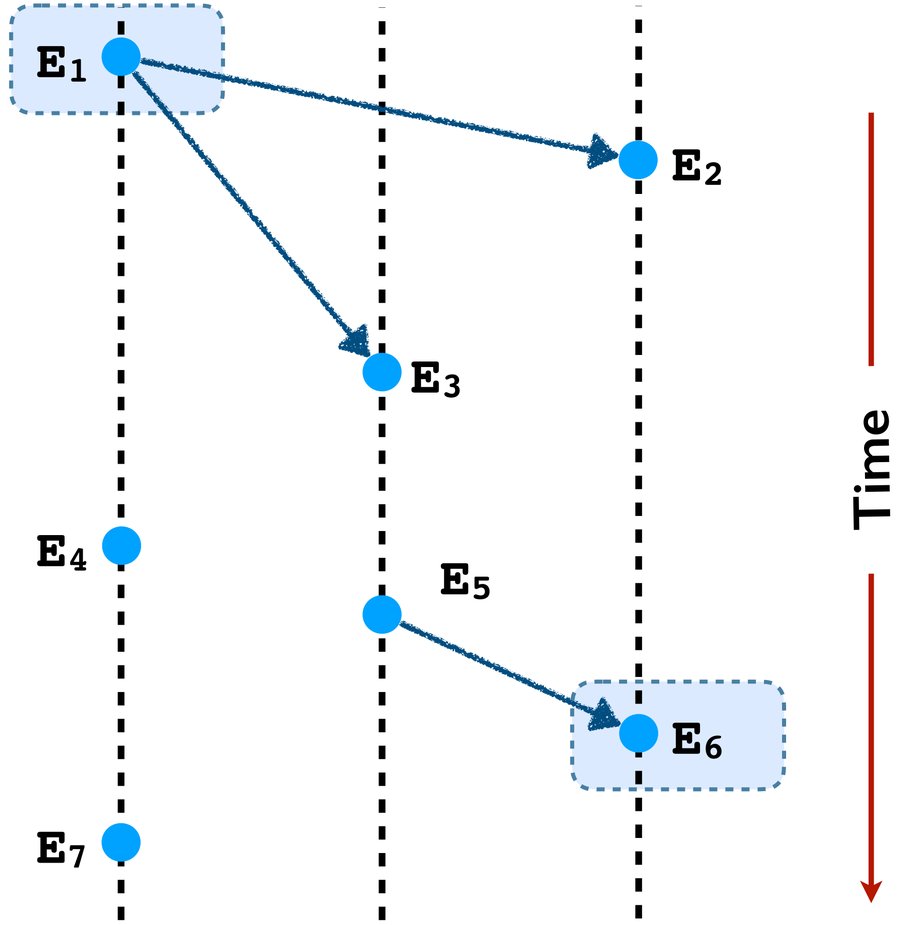
- $E_1 <_{HB} E_3$ (send-receive)
- $E_3 <_{HB} E_5$ (same process)



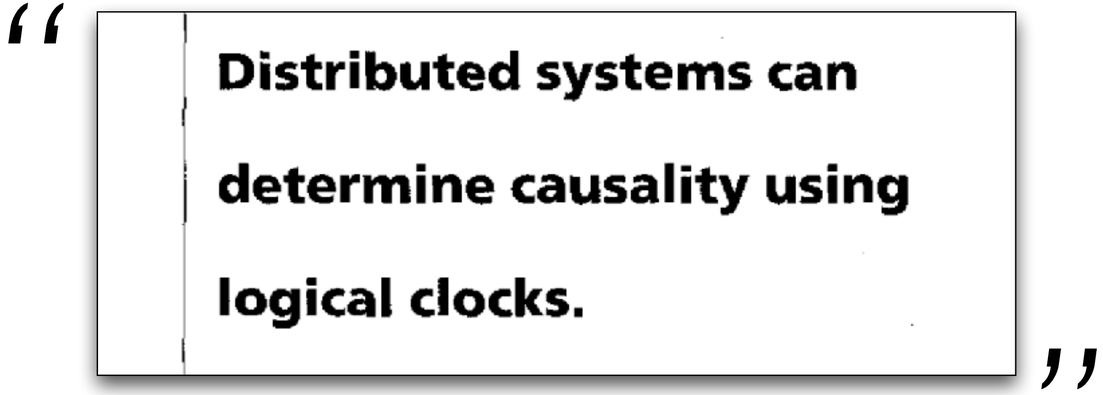
Happens-Before for Capturing Causality

Are two given events causally related?

- $E_1 <_{HB} E_3$ (send-receive)
- $E_3 <_{HB} E_5$ (same process)
- $E_1 <_{HB} E_6$ (transitivity)



Logical Timestamps



Logical Timestamps

Timestamp of event E

V_E : Processes → $\mathbb{N}$

- $V_E(p)$ = Number of events of process p that E *"knows"* about

"

> **Distributed systems can determine causality using logical clocks.**

"

Logical Timestamps

Timestamp of event E

V_E : Processes → $\mathbb{N}$

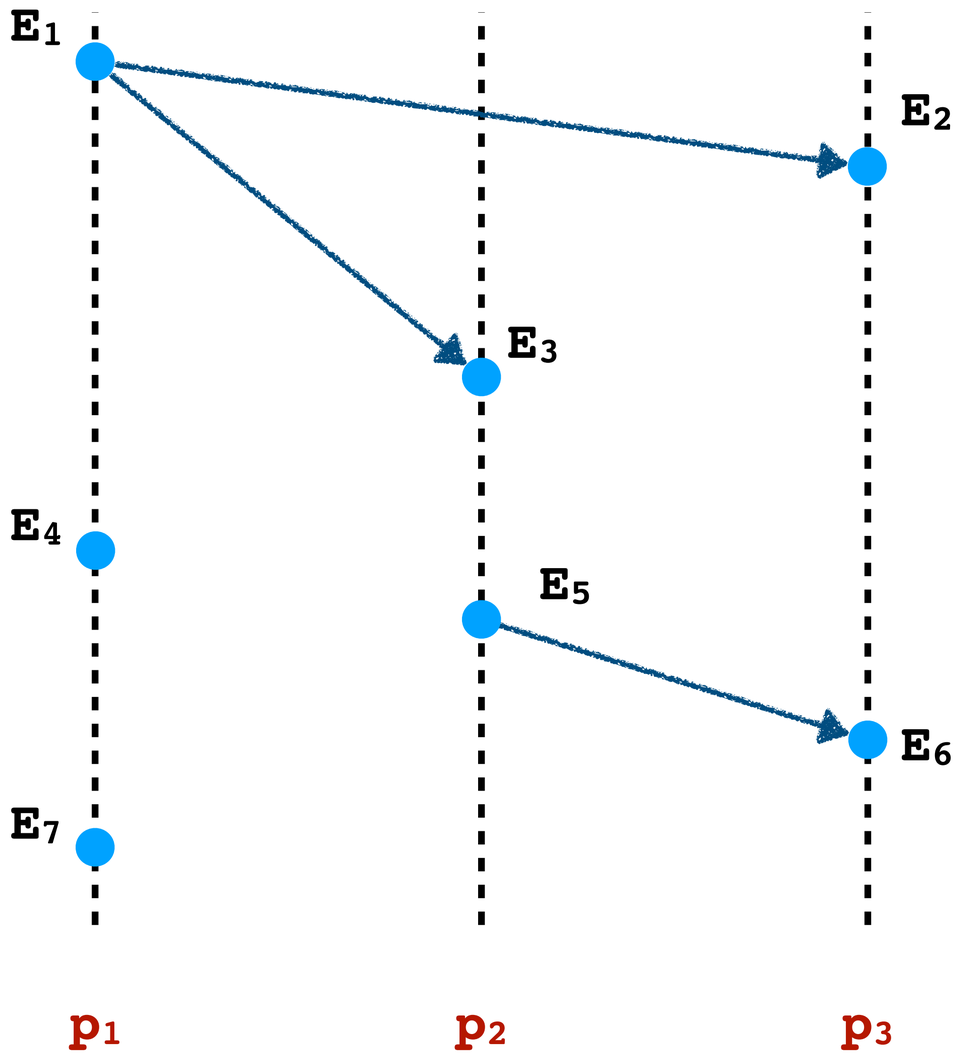
- $V_E(p)$ = Number of events of process p that E *"knows"* about

For all p,
$V_{E1}(p) \leq V_{E2}(p)$

$E1 \leq_{HB} E2$ iff $V_{E1} \sqsubseteq V_{E2}$

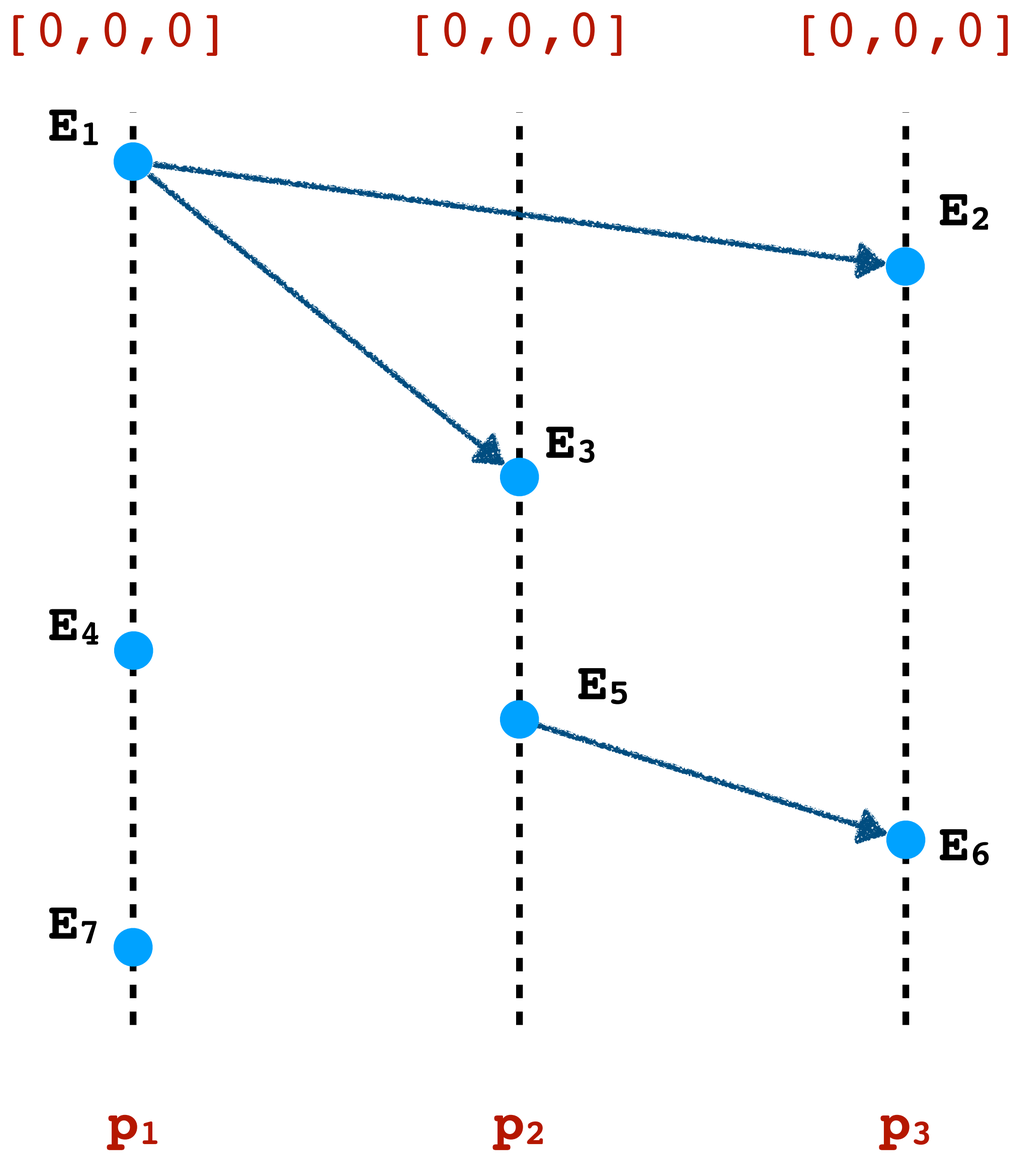
Computing Timestamps

- For every process p, maintain V_p



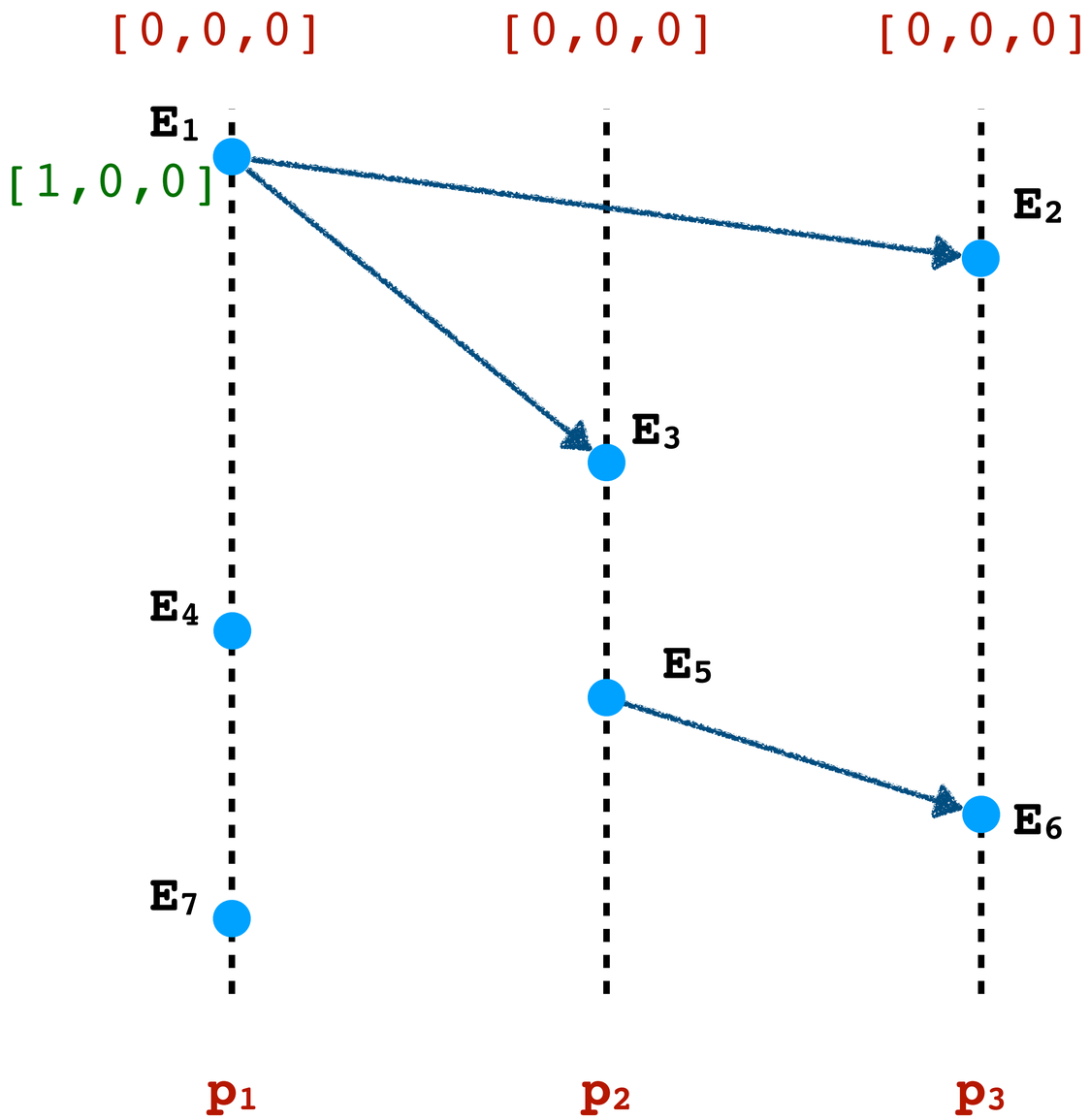
Computing Timestamps

- For every process p, maintain V_p
- Initially:
  - $V_p(q) = 0$ for every p, q



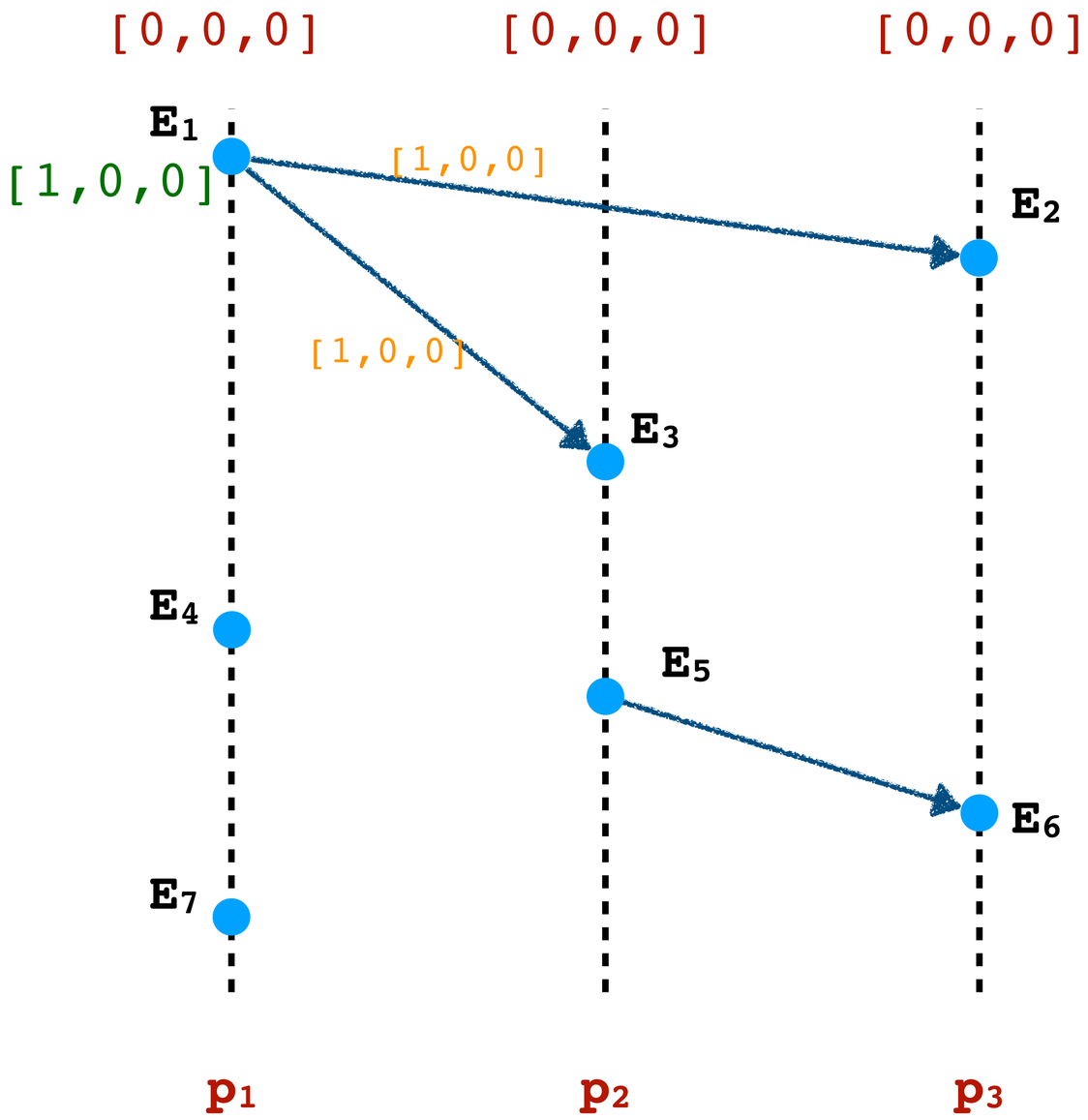
Computing Timestamps

- For every process p, maintain V_p
- Initially:
  - $V_p(q) = 0$ for every p, q
- Before executing an event in p:
  - $V_p(p) \leftarrow V_p(p) + 1$



Computing Timestamps

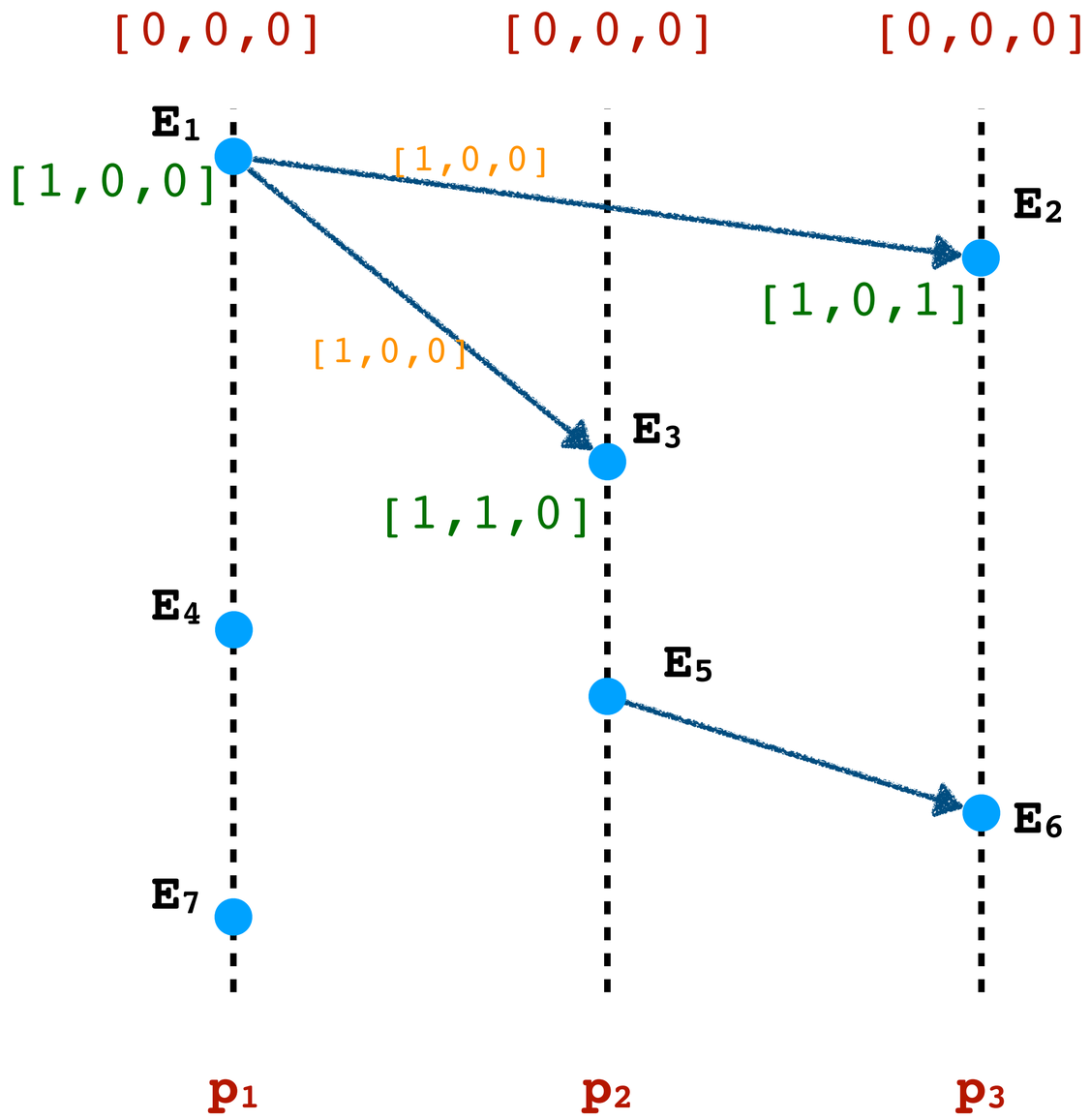
- For every process p, maintain V_p
- Initially:
  - $V_p(q) = 0$ for every p, q
- Before executing an event in p:
  - $V_p(p) \leftarrow V_p(p) + 1$
- At a send event in p:
  - attach a copy of V_p to message



Computing Timestamps

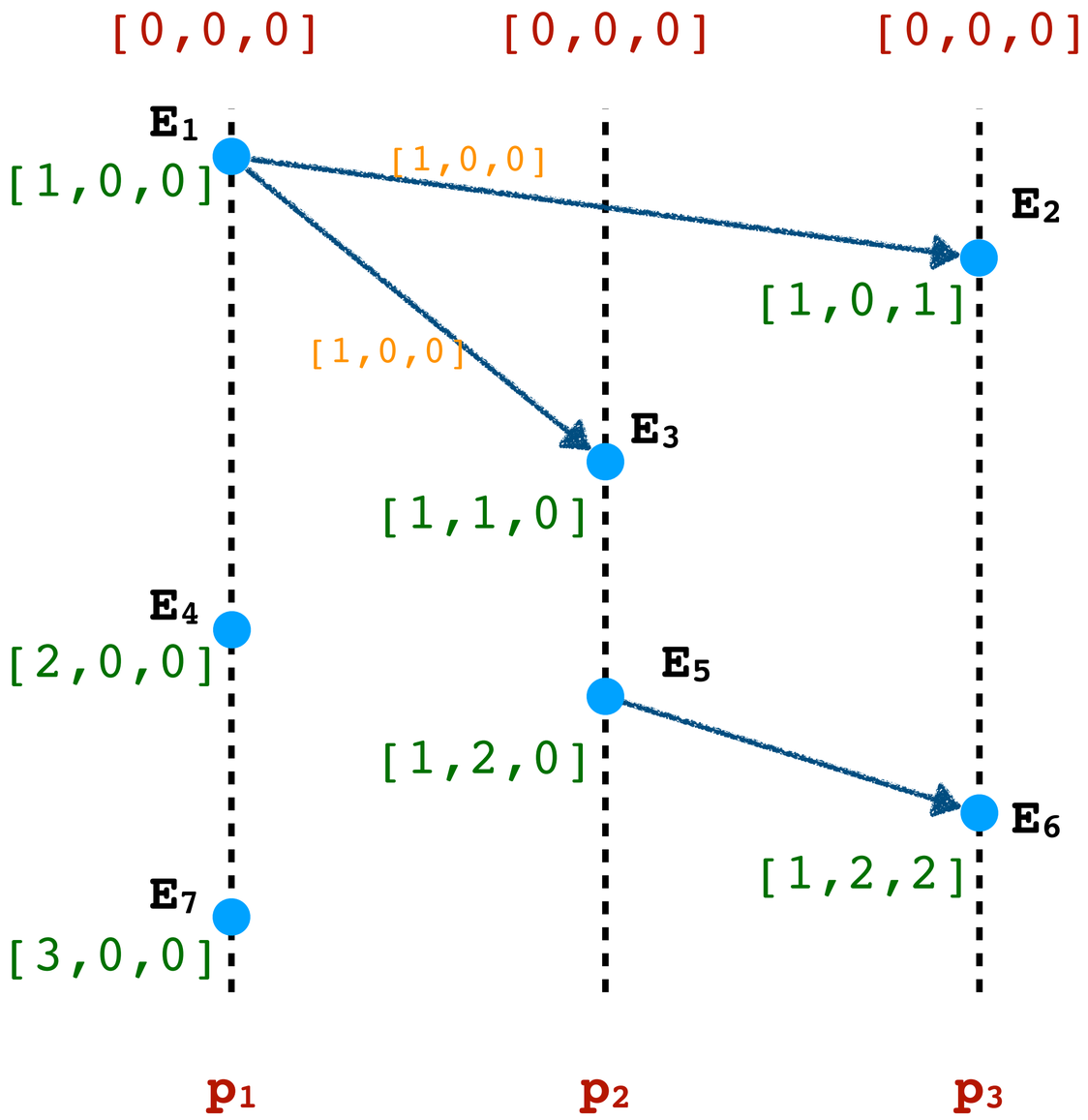
- For every process p, maintain V_p
- Initially:
  - $V_p(q) = 0$ for every p, q
- Before executing an event in p:
  - $V_p(p) \leftarrow V_p(p) + 1$
- At a send event in p:
  - attach a *copy* of V_p to message
- At a receive (of message m) event in p:
  - *merge* the timestamp of m
  - $V_p \leftarrow V_p \sqcup V_m$

Point-wise maximum or "Join"



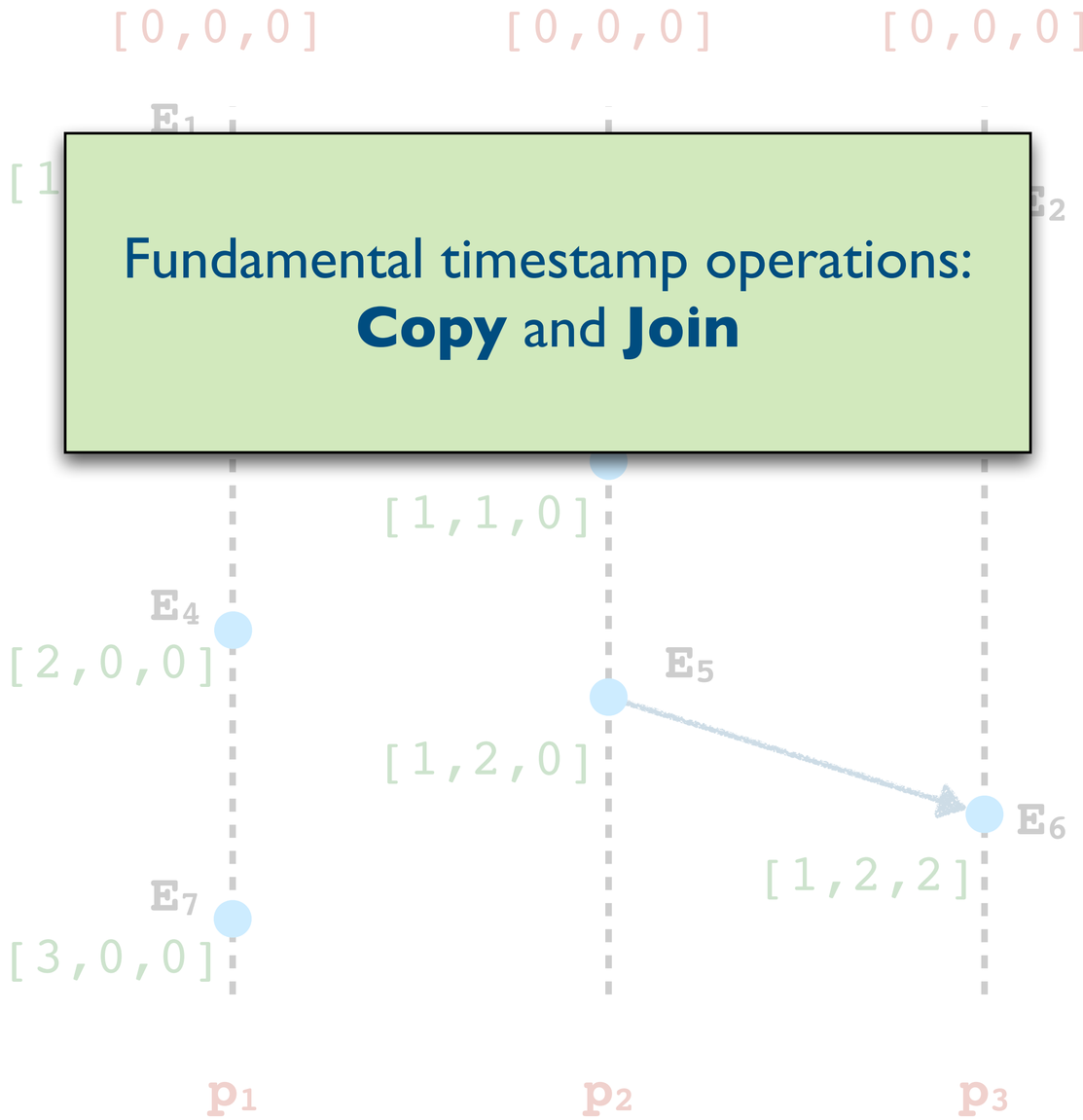
Computing Timestamps

- For every process p, maintain V_p
- Initially:
  - $V_p(q) = 0$ for every p, q
- Before executing an event in p:
  - $V_p(p) \leftarrow V_p(p) + 1$
- At a send event in p:
  - attach a *copy* of V_p to message
- At a receive (of message m) event in p:
  - *merge* the timestamp of m
  - $V_p \leftarrow V_p \sqcup V_m$



Computing Timestamps

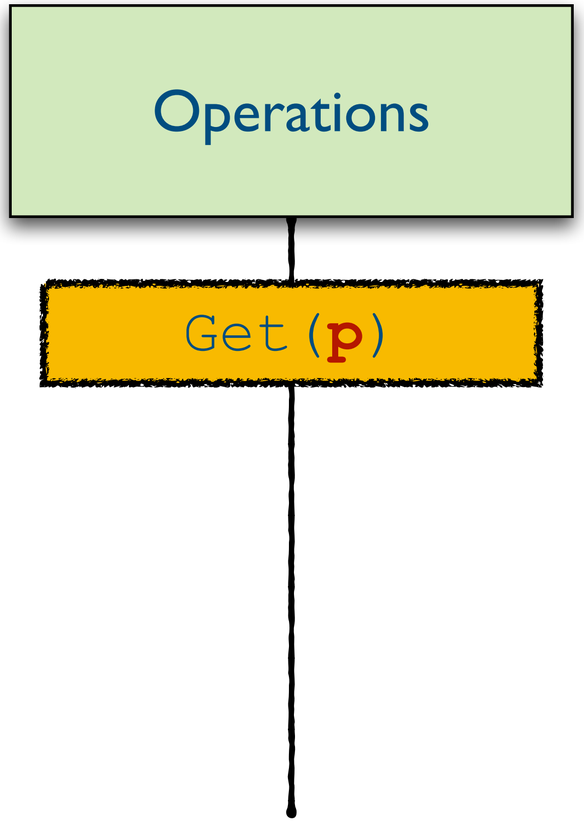
- For every process p, maintain V_p
- Initially:
  - $V_p(q) = 0$ for every p, q
- Before executing an event in p:
  - $V_p(p) \leftarrow V_p(p) + 1$
- At a send event in p:
  - attach a *copy* of V_p to message
- At a receive (of message m) event in p:
  - *merge* the timestamp of m
  - $V_p \leftarrow V_p \sqcup V_m$

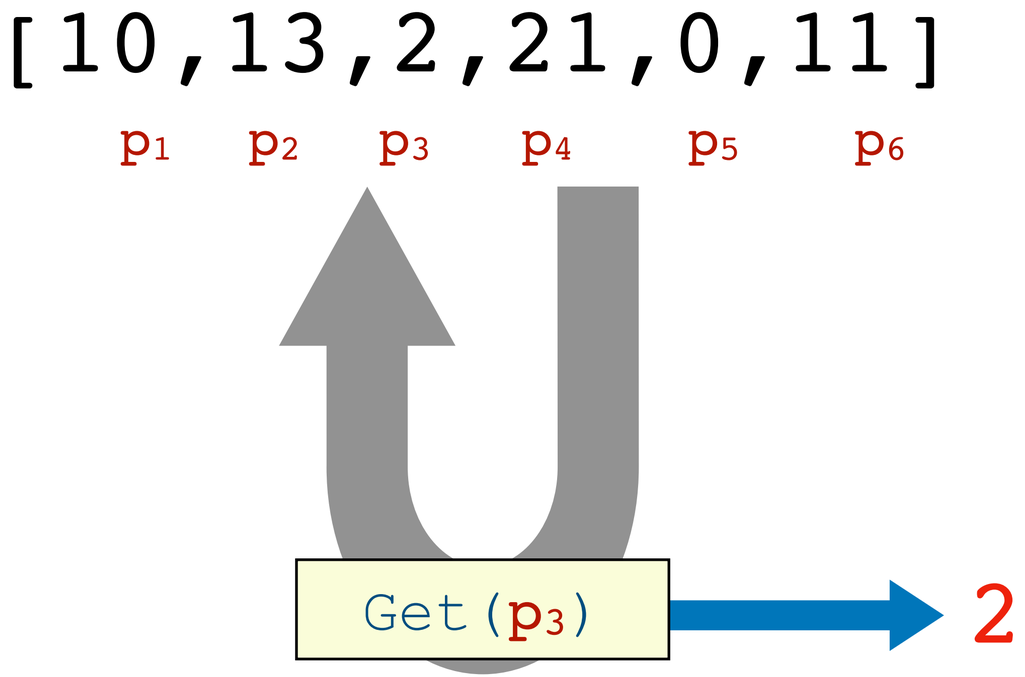


Fundamental timestamp operations:
Copy and **Join**

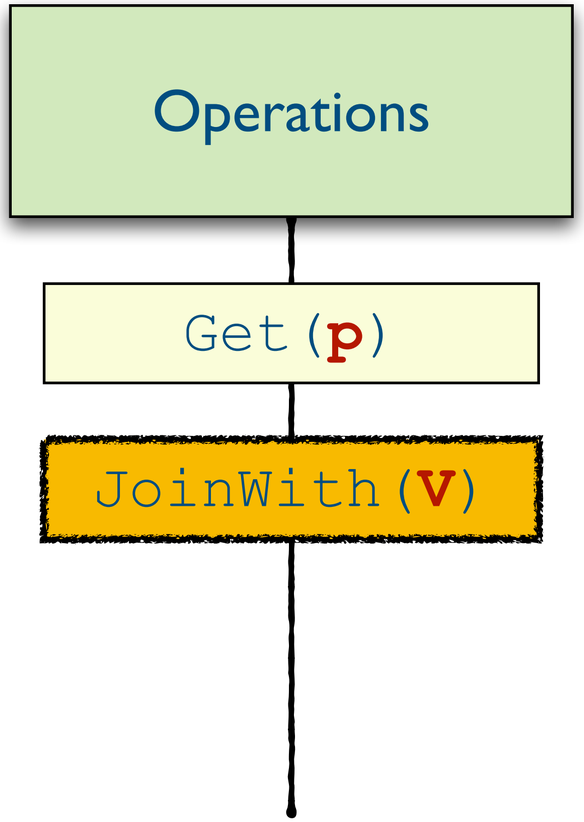
Data Structure for Timestamping

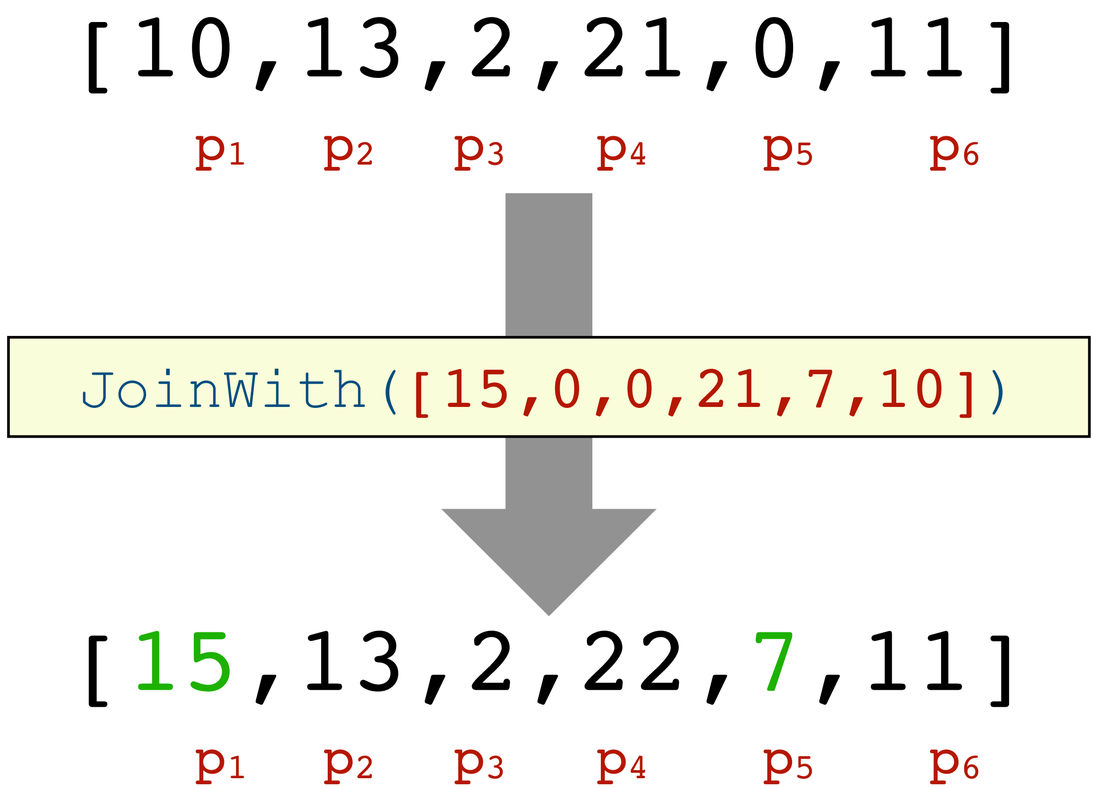
Data Structure for Timestamping



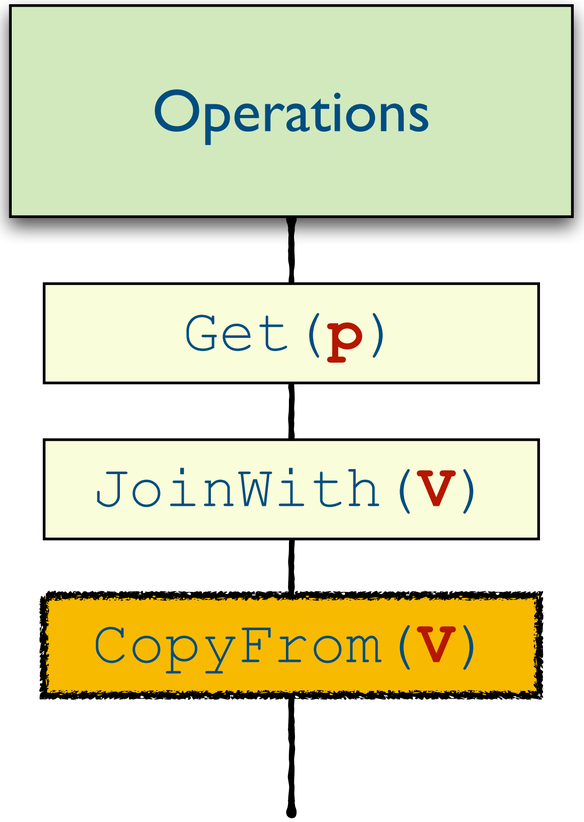


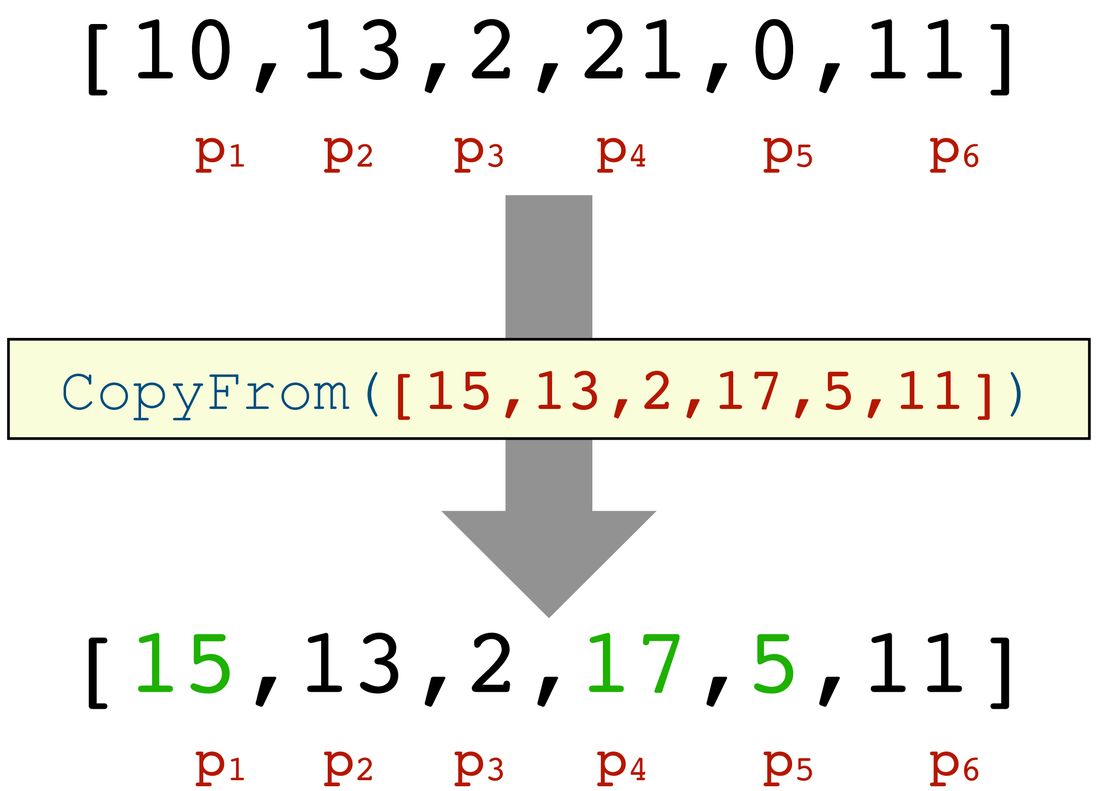
Data Structure for Timestamping



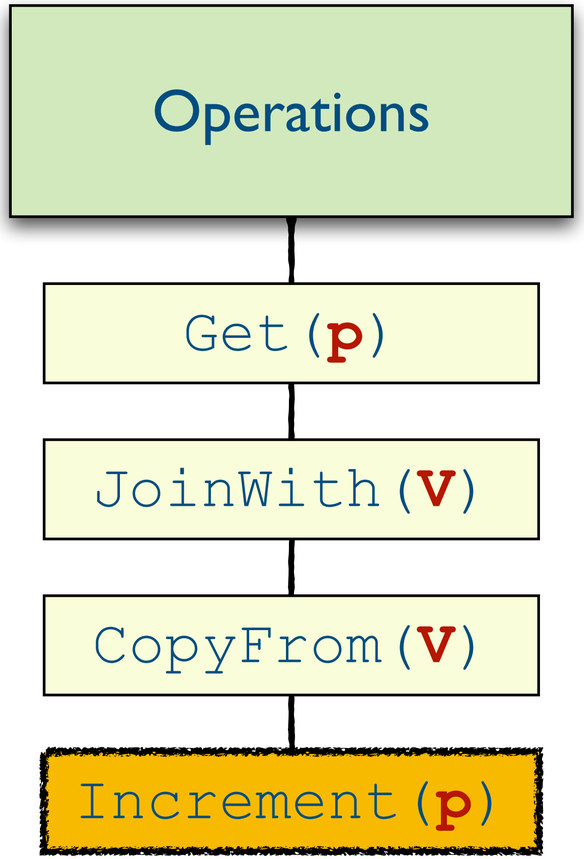


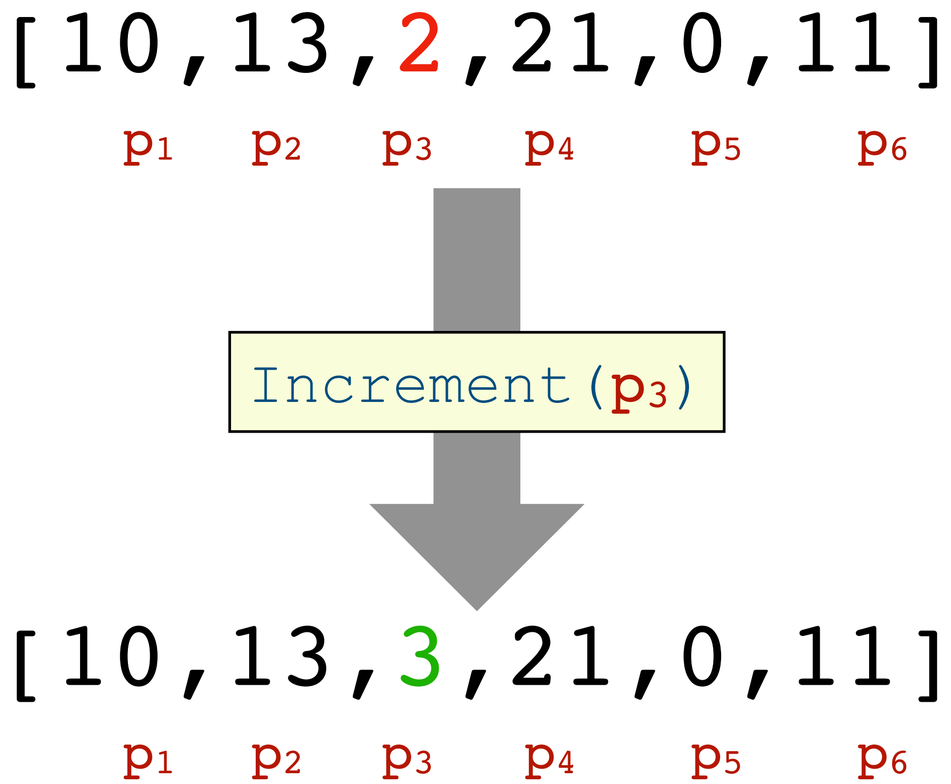
Data Structure for Timestamping



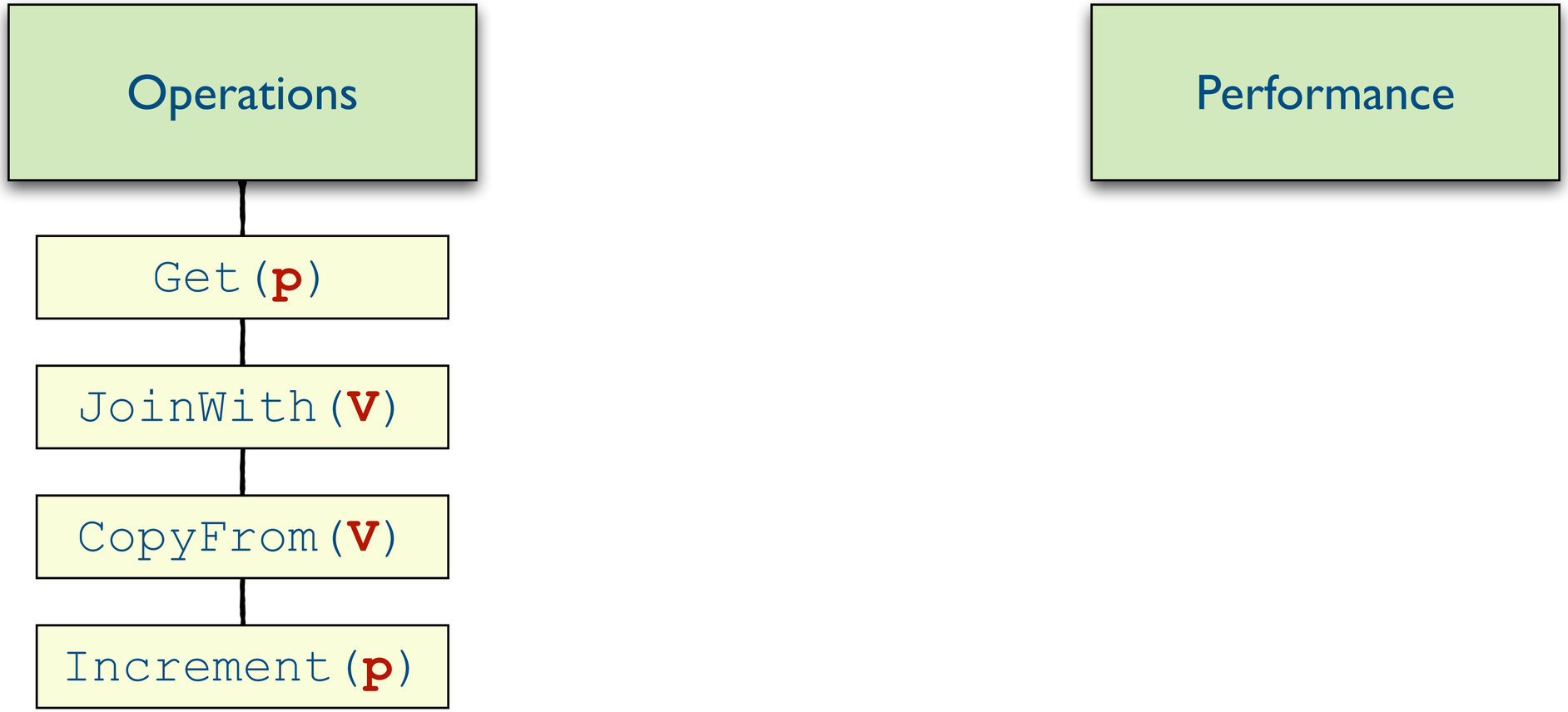


Data Structure for Timestamping

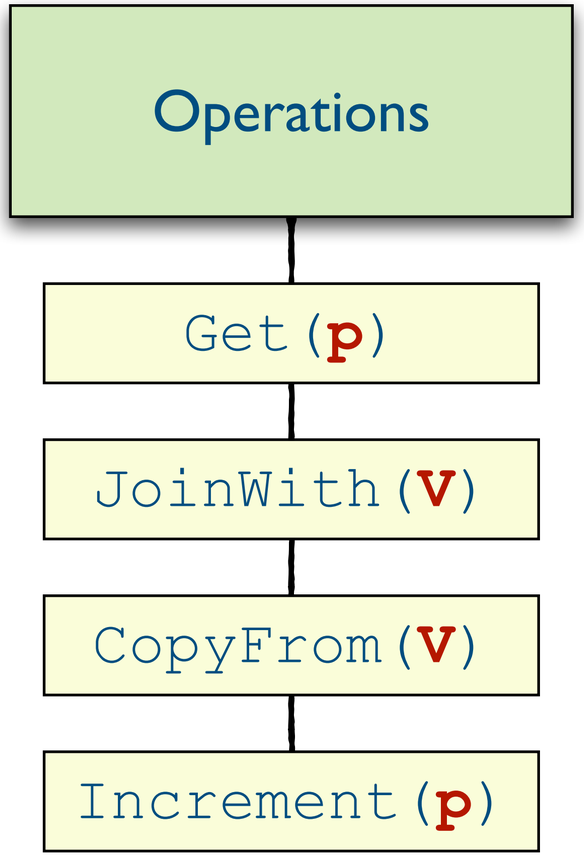


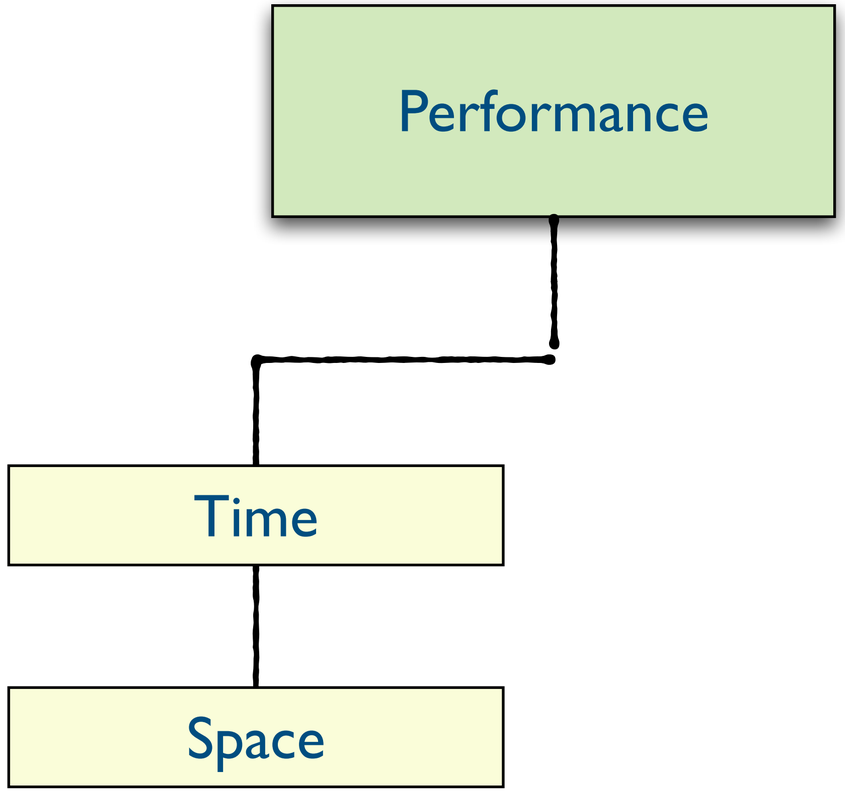


Data Structure for Timestamping

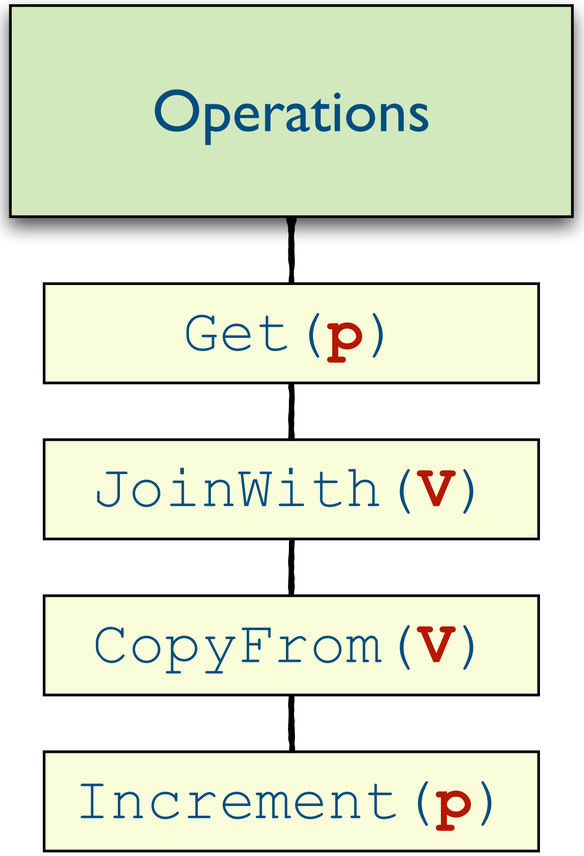


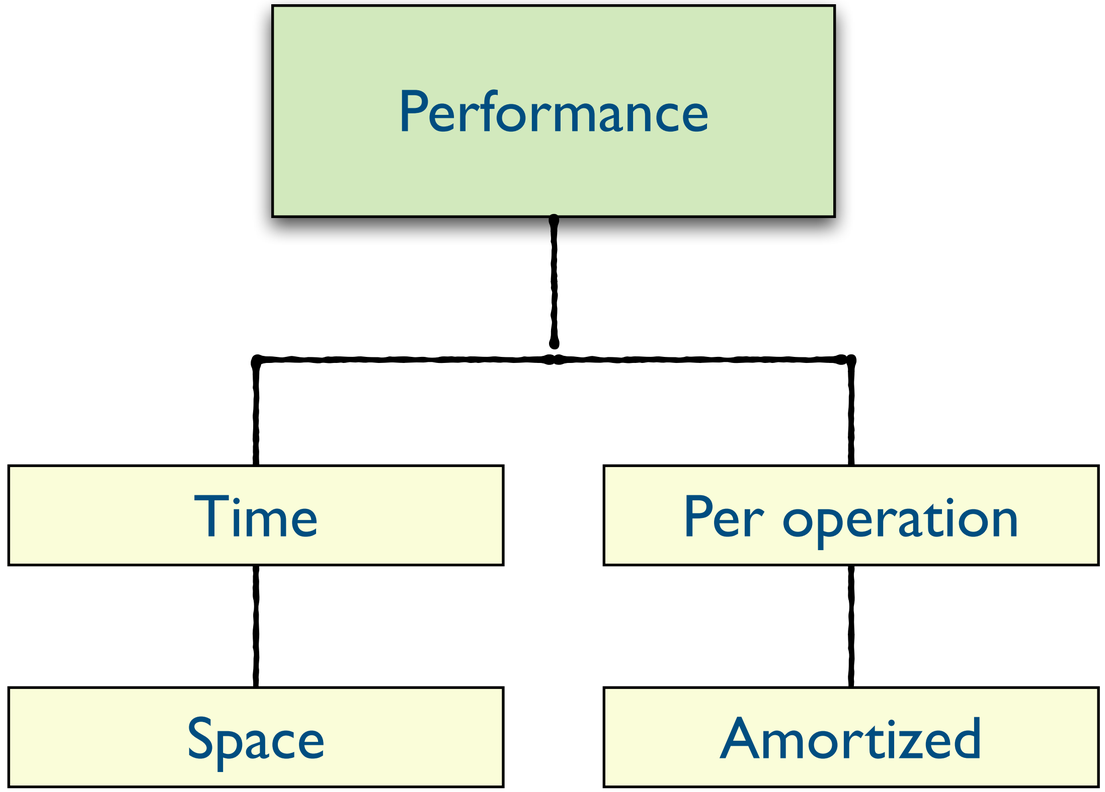
Data Structure for Timestamping





Data Structure for Timestamping

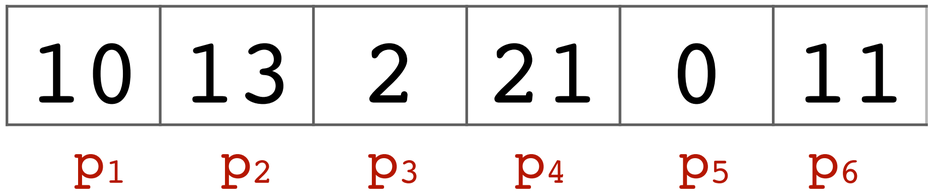




Vector Clock

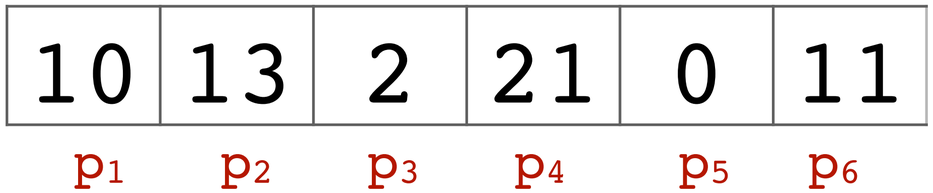
Vector Clock

Flat array/map, indexed by process ids



Vector Clock

Flat array/map, indexed by process ids

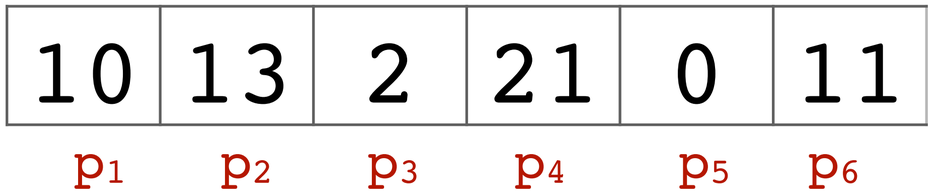


V1.JoinWith(**V2**)

```
for each process p:
    if V1.Get(p) < V2.Get(p):
        V1.p := V2.Get(p)
```

Vector Clock

Flat array/map, indexed by process ids



V1.JoinWith(**V2**)

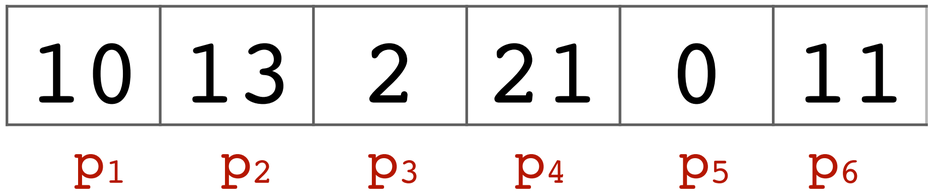
```
for each process p:
    if V1.Get(p) < V2.Get(p):
        V1.p := V2.Get(p)
```

V1.CopyFrom(**V2**)

```
for each process p:
    V1.p := V2.Get(p)
```

Vector Clock

Flat array/map, indexed by process ids



V1.JoinWith(**V2**)

```
for each process p:
    if V1.Get(p) < V2.Get(p):
        V1.p := V2.Get(p)
```

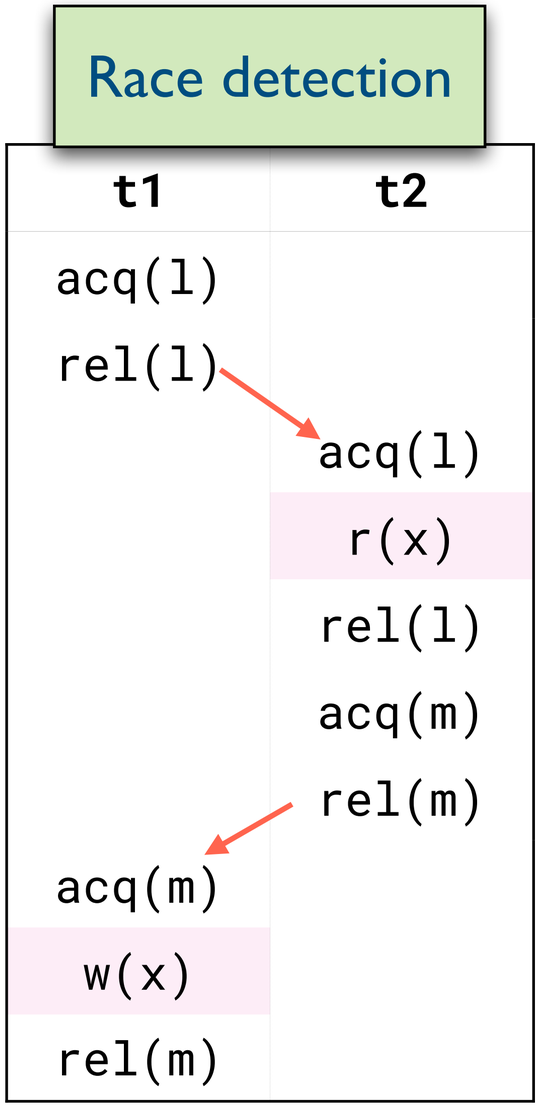
V1.CopyFrom(**V2**)

```
for each process p:
    V1.p := V2.Get(p)
```

Both **Join** and **Copy** take Θ(|Processes|) time for Vector Clocks

Vector Clock Overhead

Vector Clock Overhead



Vector Clock Overhead

- For every thread `t`, maintain vector clock V_t
- For every lock `lk`, maintain vector clock V_{lk}

```
procedure acq(t, lk)
    Vt.JoinWith(Vlk)

procedure rel(t, lk)
    Vlk.CopyFrom(Vt)
```

Race detection

t1	t2
acq(l)	
rel(l)	
	acq(l)
	r(x)
	rel(l)
	acq(m)
	rel(m)
acq(m)	
w(x)	
rel(m)	

Vector Clock Overhead

Total overhead = Θ(|Processes| * Number of Events)

- Typically, ~10 billion events and ~100 processes
- Immense slowdown when detecting data races

Can we do better?

How can we do better?

Sub-linear time in **join** and **copy**?

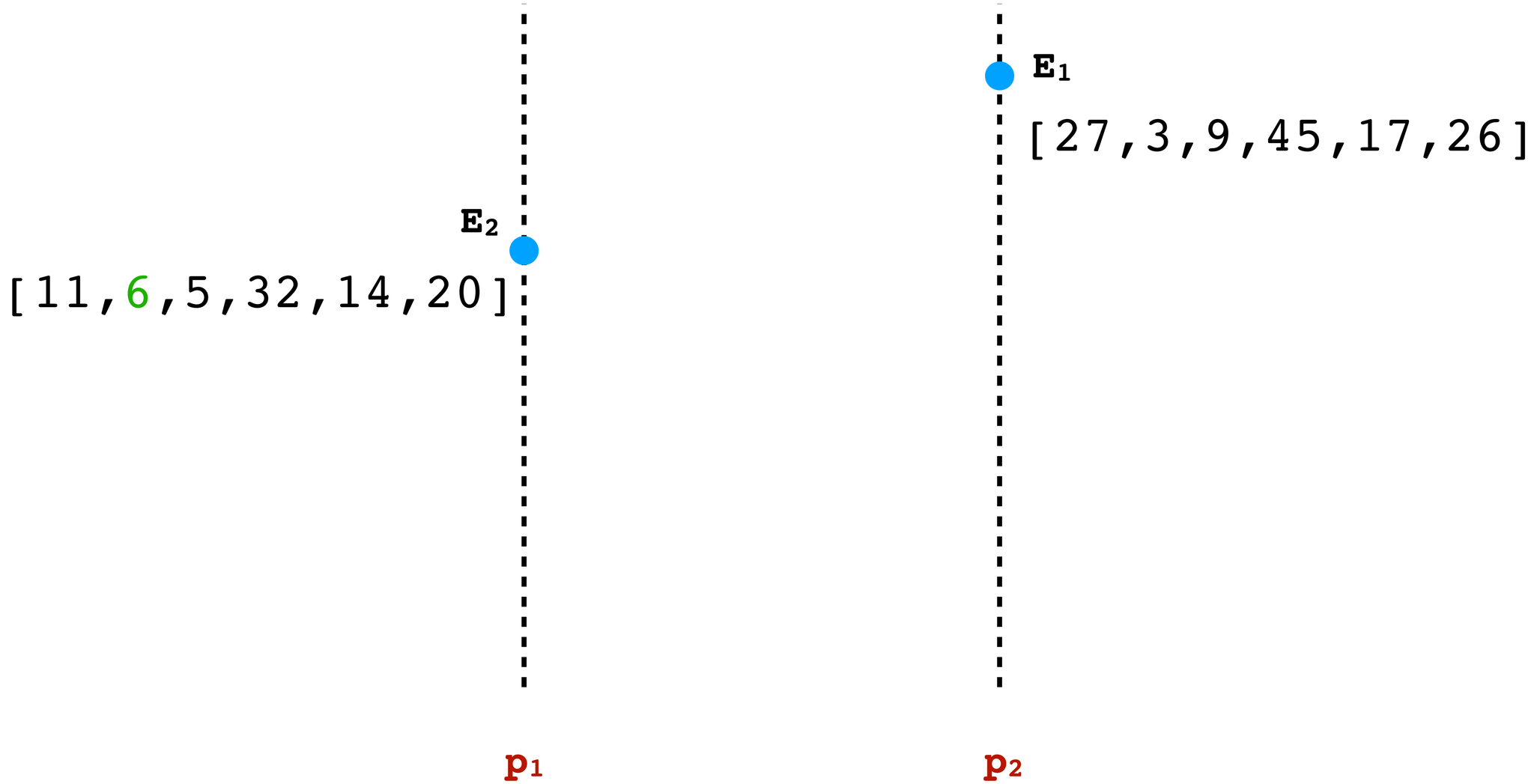
How can we do better?

Sub-linear time in **join** and **copy**?

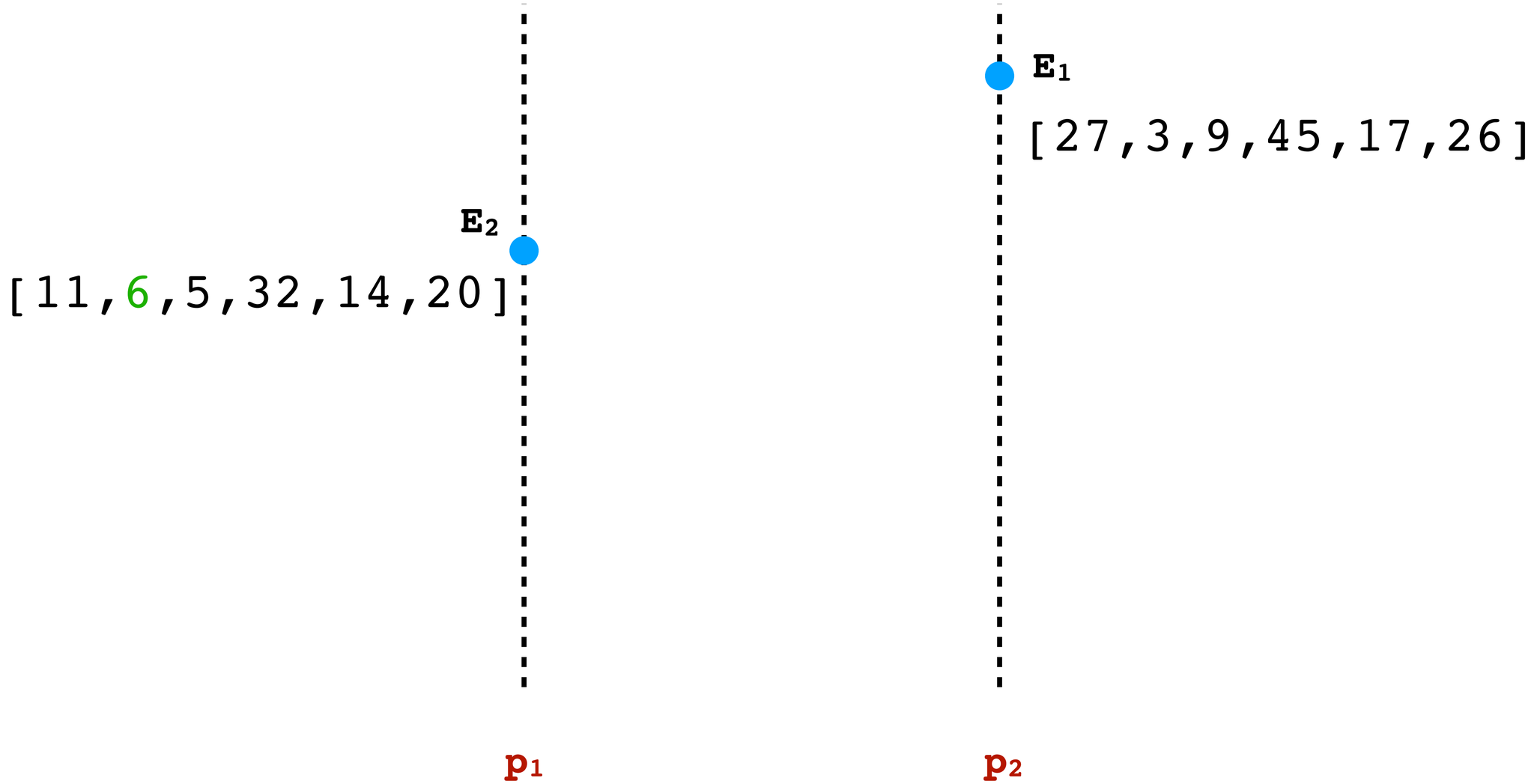
Skip looking at some entries

We *may* be able to do better!

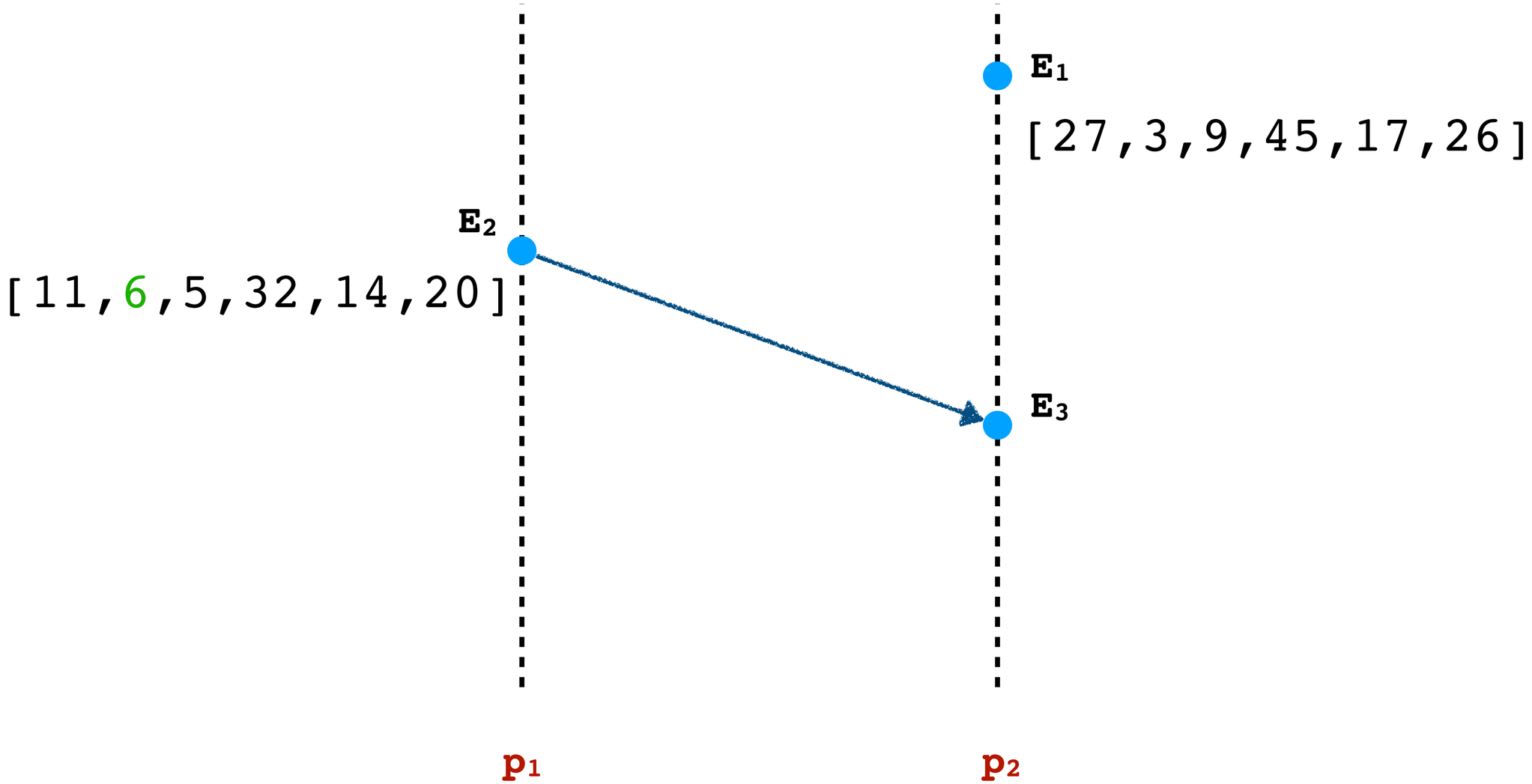
We *may* be able to do better!



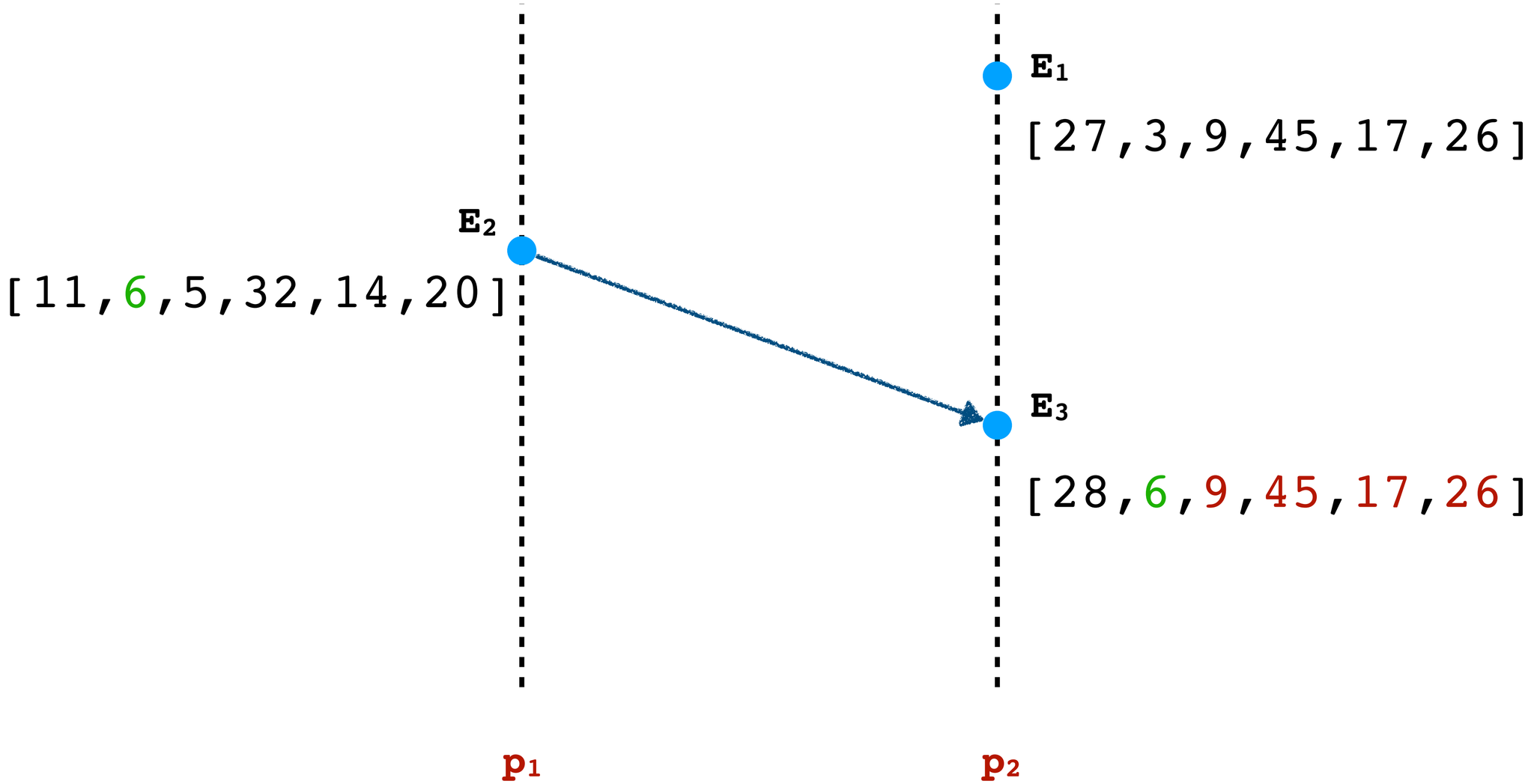
We *may* be able to do better!



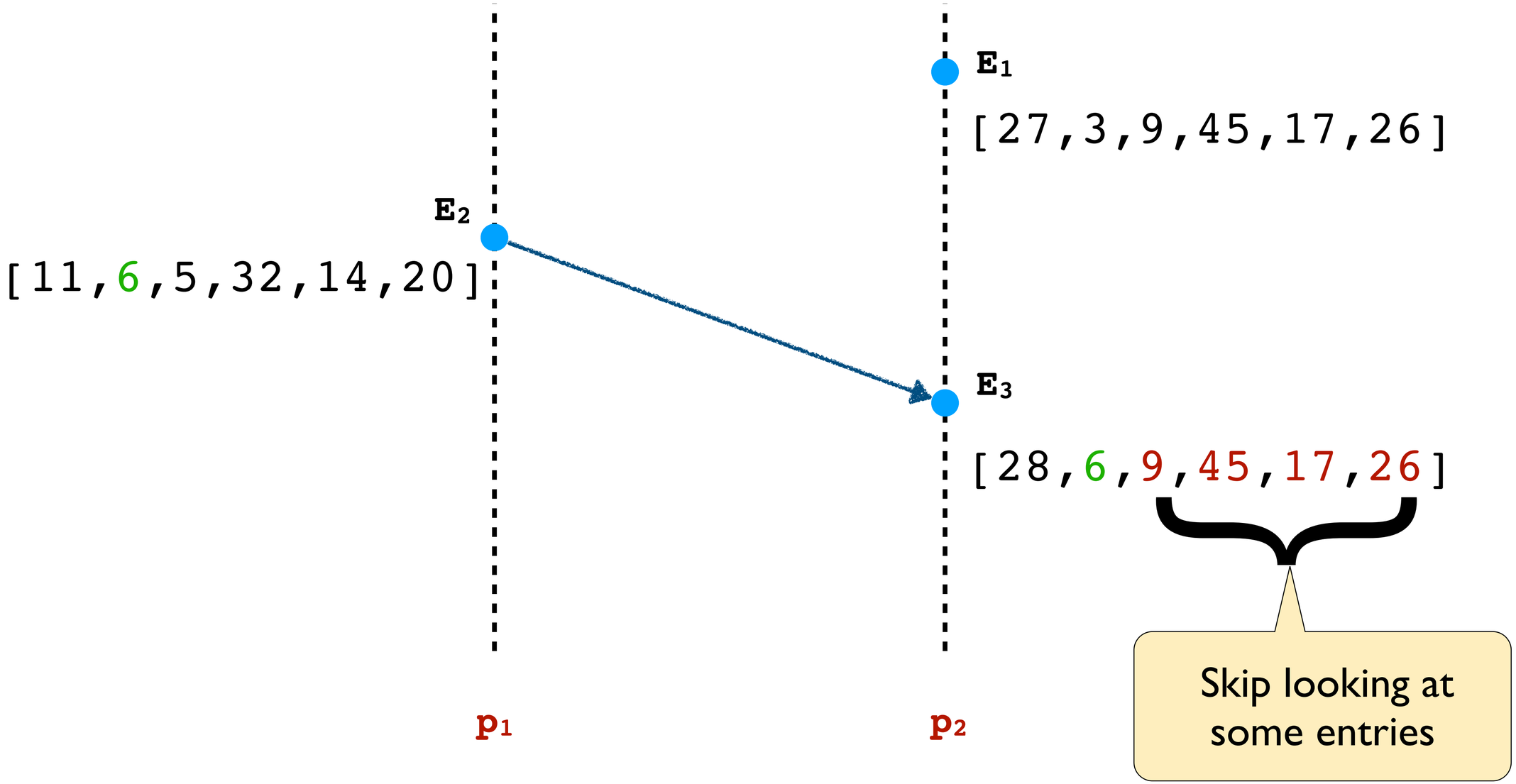
We *may* be able to do better!



We *may* be able to do better!



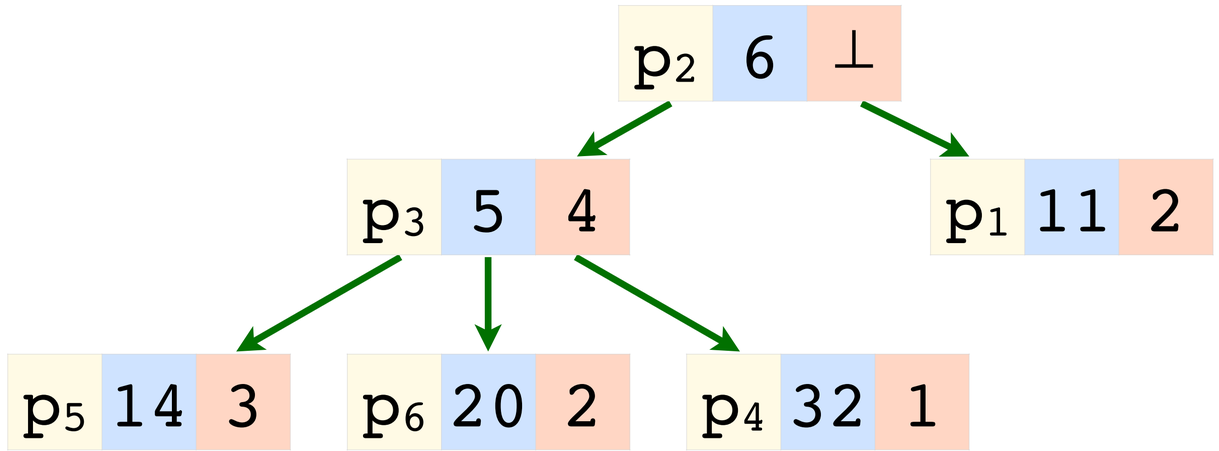
We *may* be able to do better!



Tree Clocks

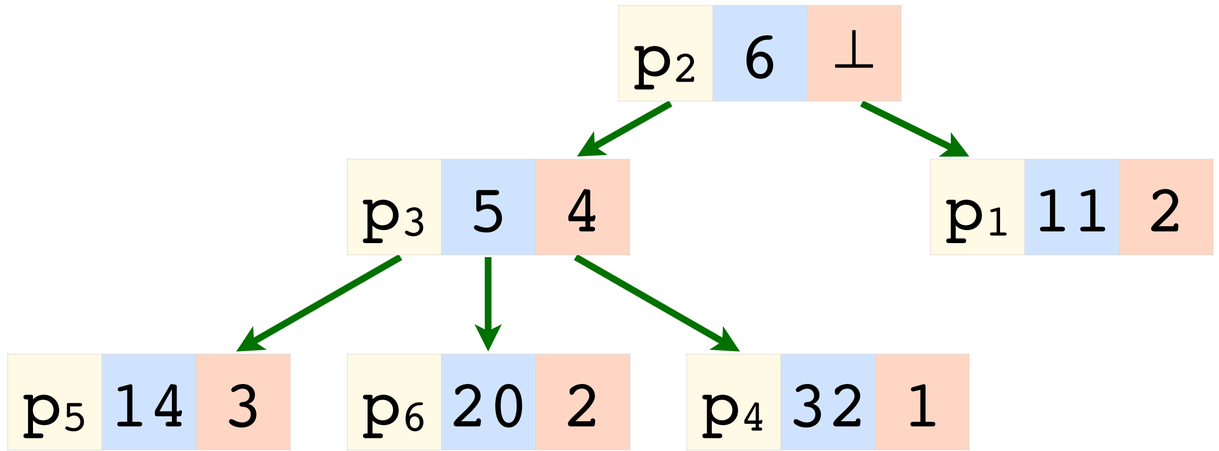
Tree Clocks

- Store **provenance information** hierarchically



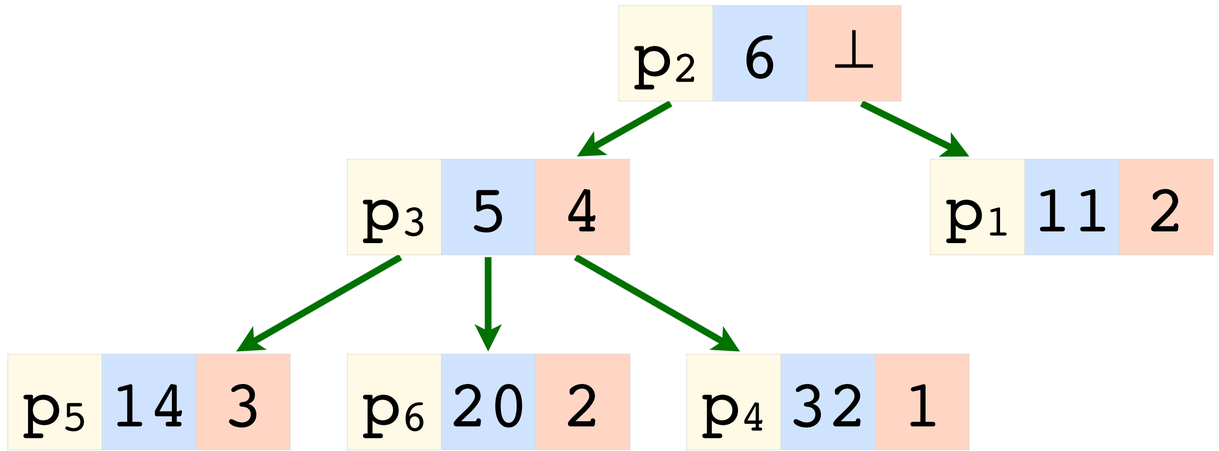
Tree Clocks

- Store **provenance information** hierarchically
  - Nodes store local times + extra metadata



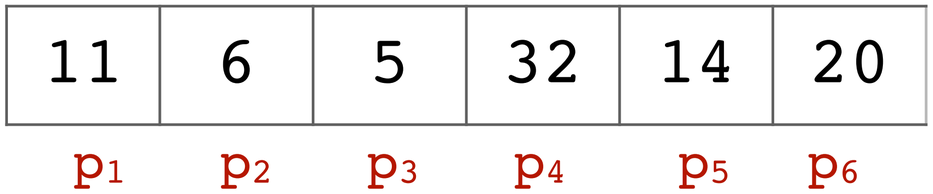
Tree Clocks

- Store **provenance information** hierarchically
  - Nodes store local times + extra metadata
  - Hierarchical structure: how and when information was obtained

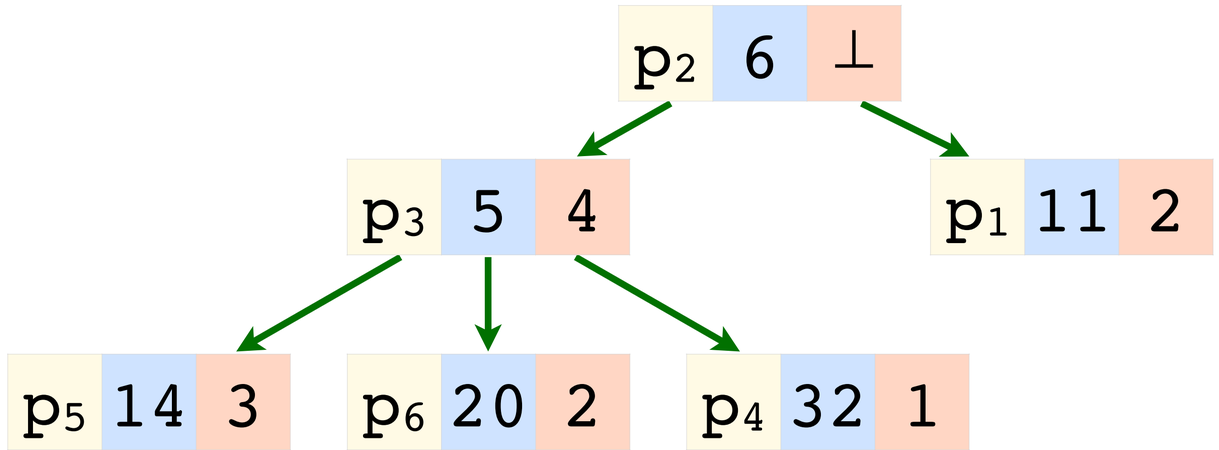


Tree Clocks

- Store **provenance information** hierarchically
  - Nodes store local times + extra metadata
  - Hierarchical structure: how and when information was obtained

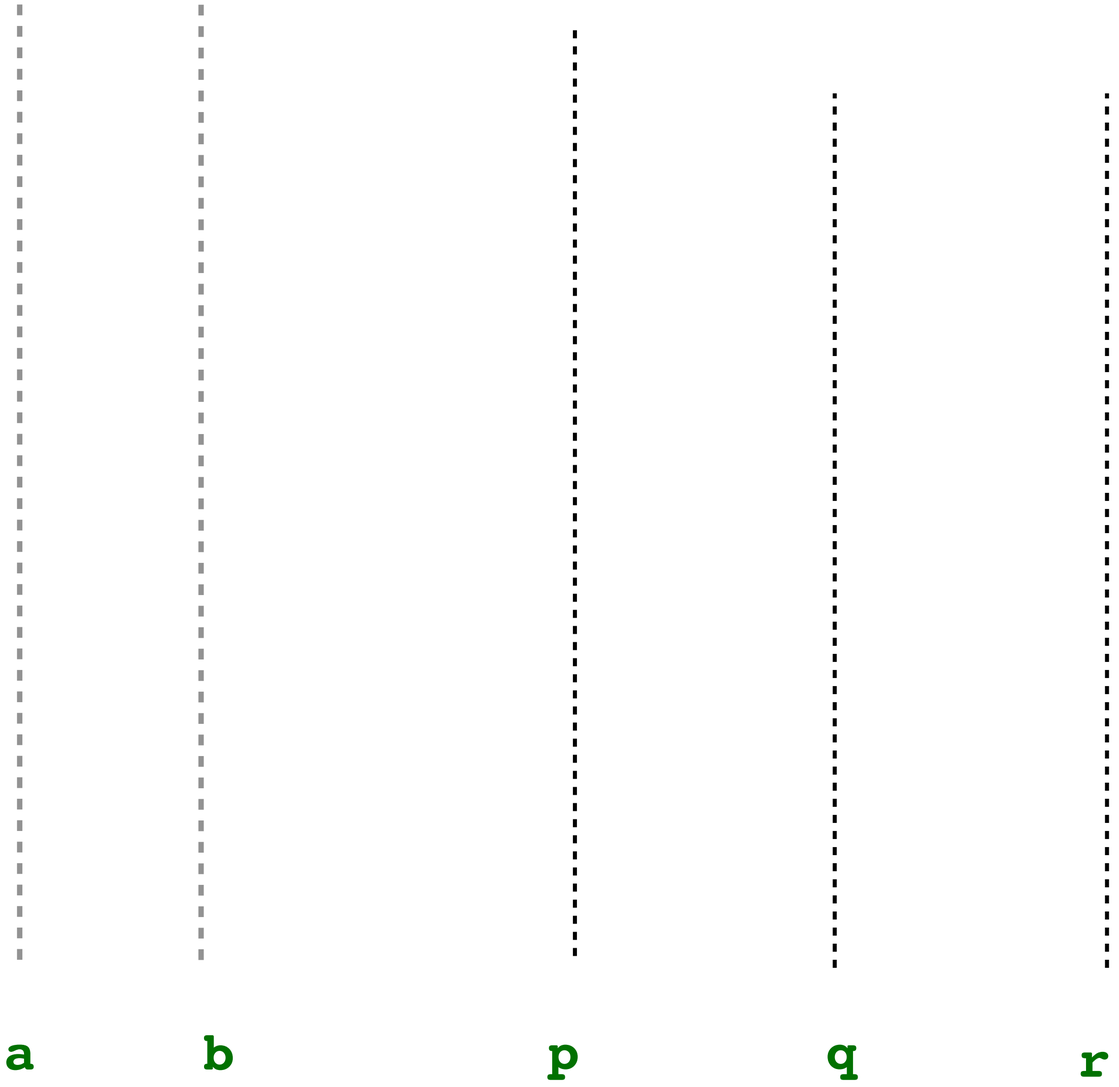


Vector Clock

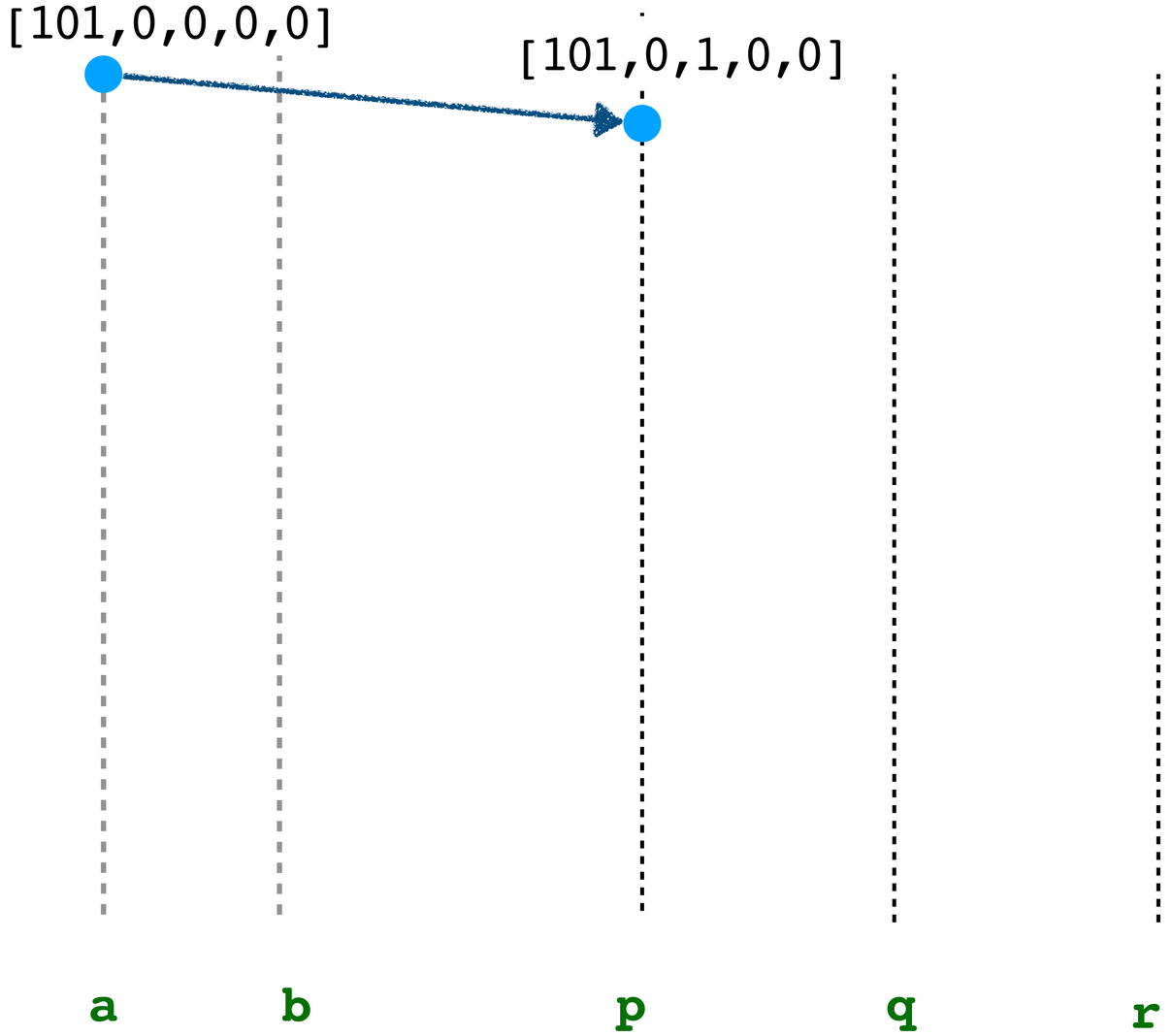


Tree Clock

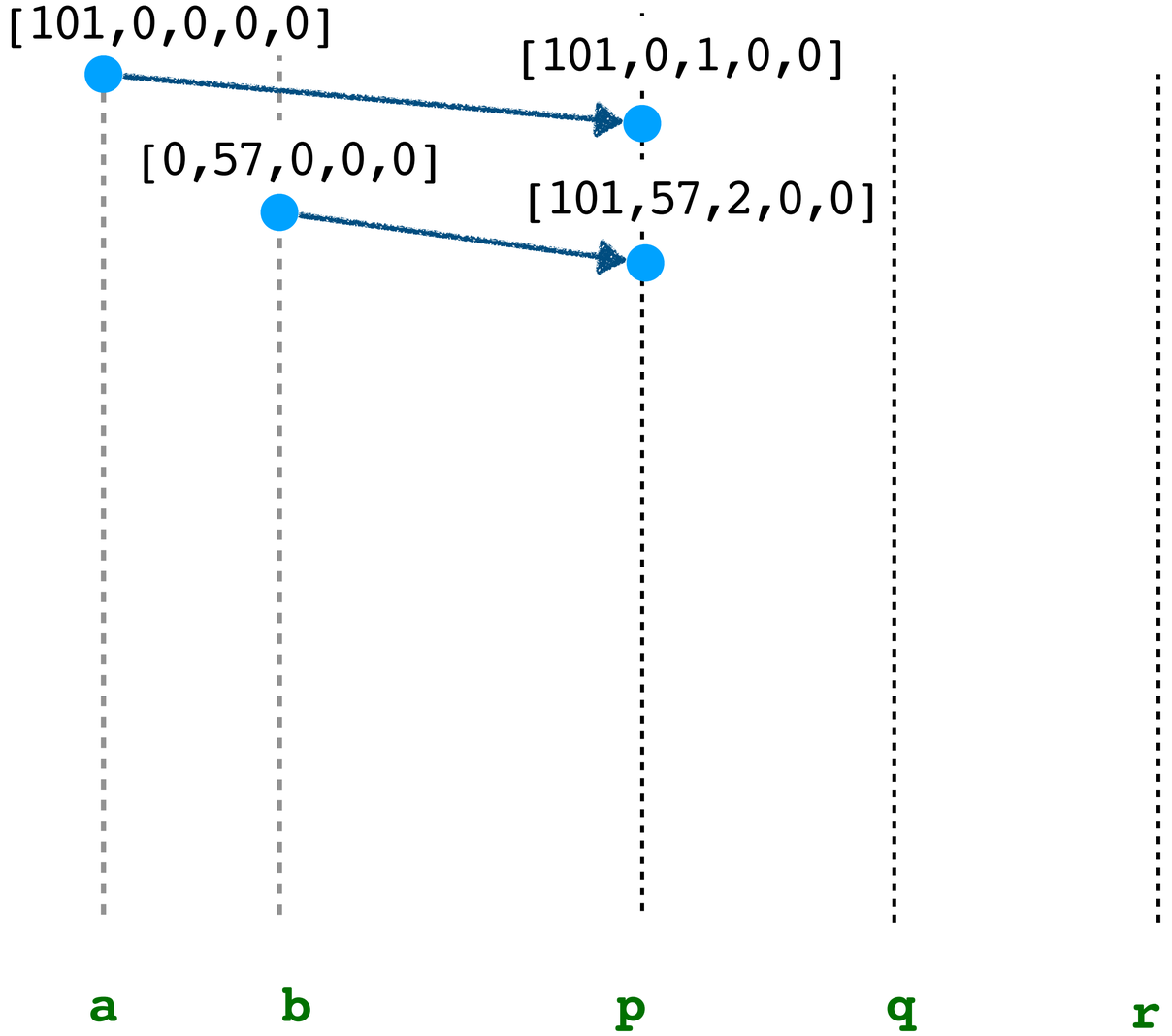
Provenance using **Transitivity**



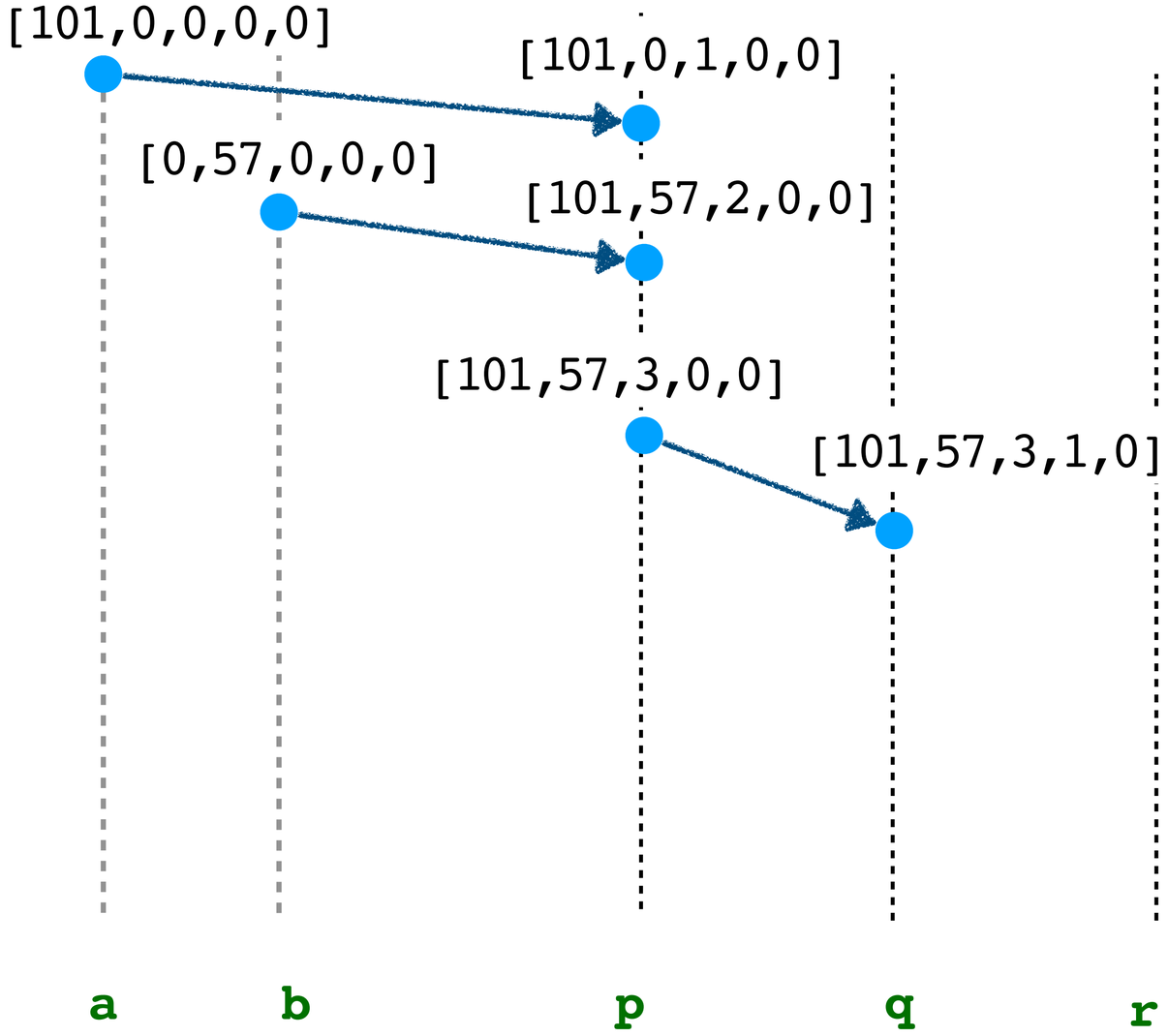
Provenance using **Transitivity**



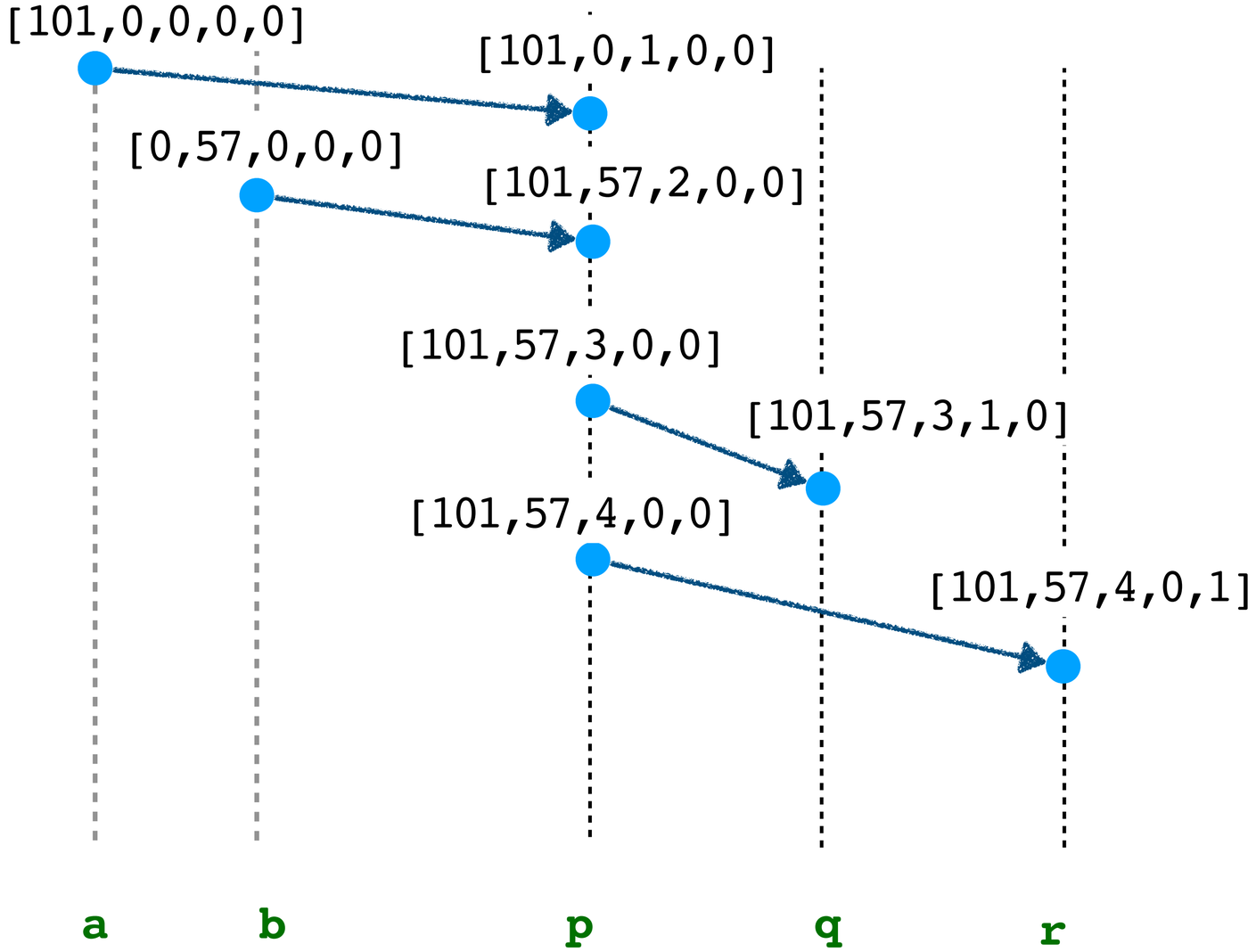
Provenance using **Transitivity**



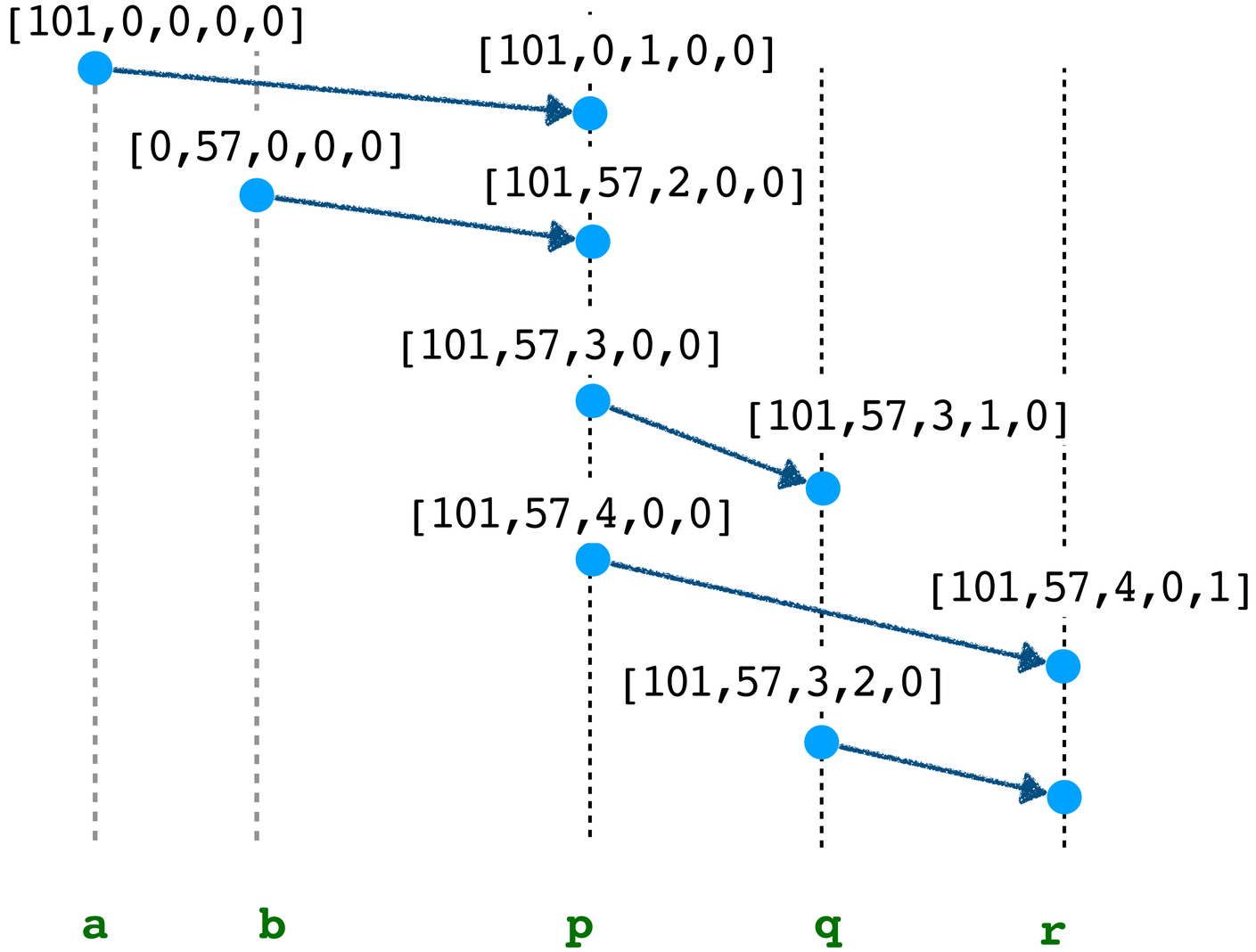
Provenance using **Transitivity**



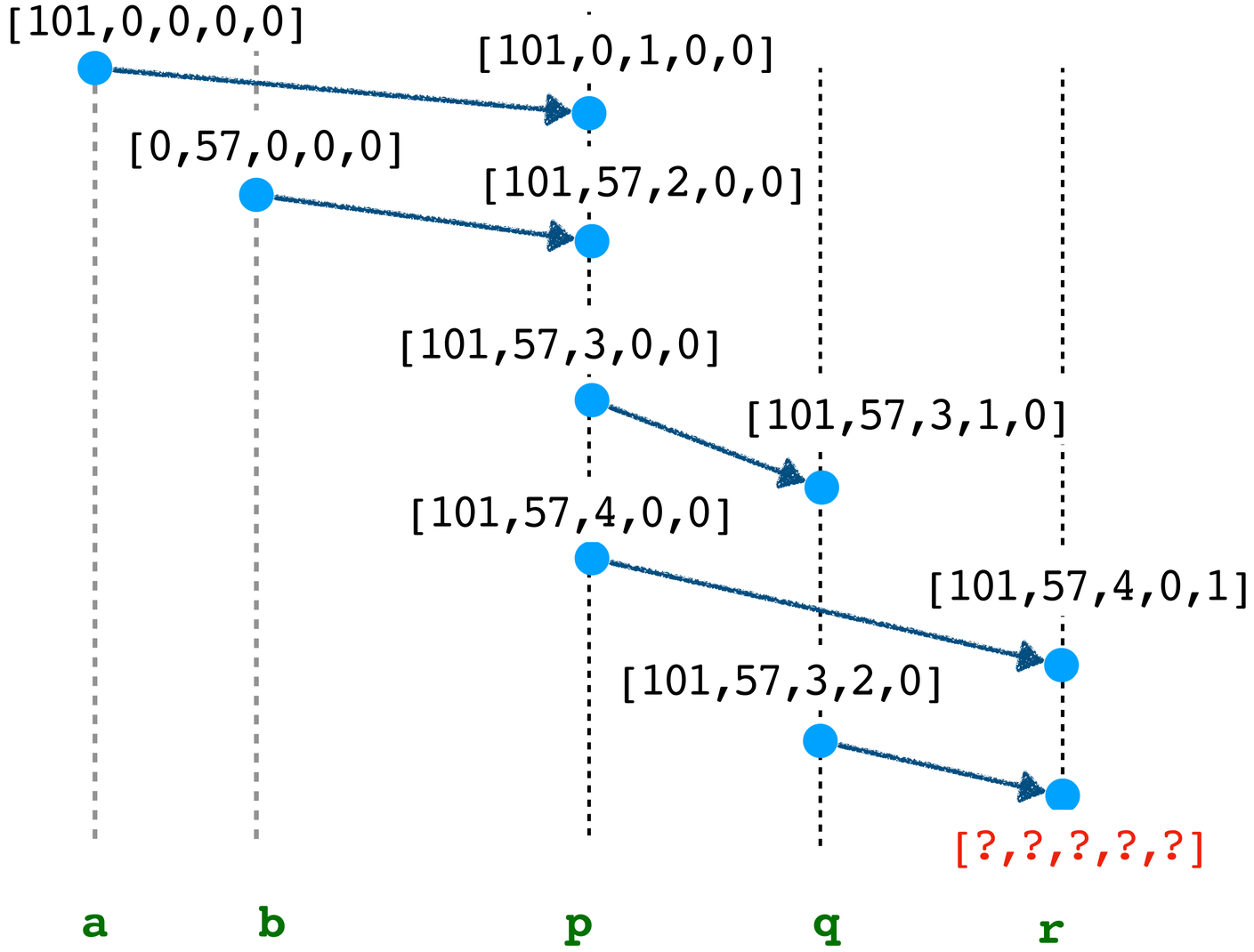
Provenance using **Transitivity**



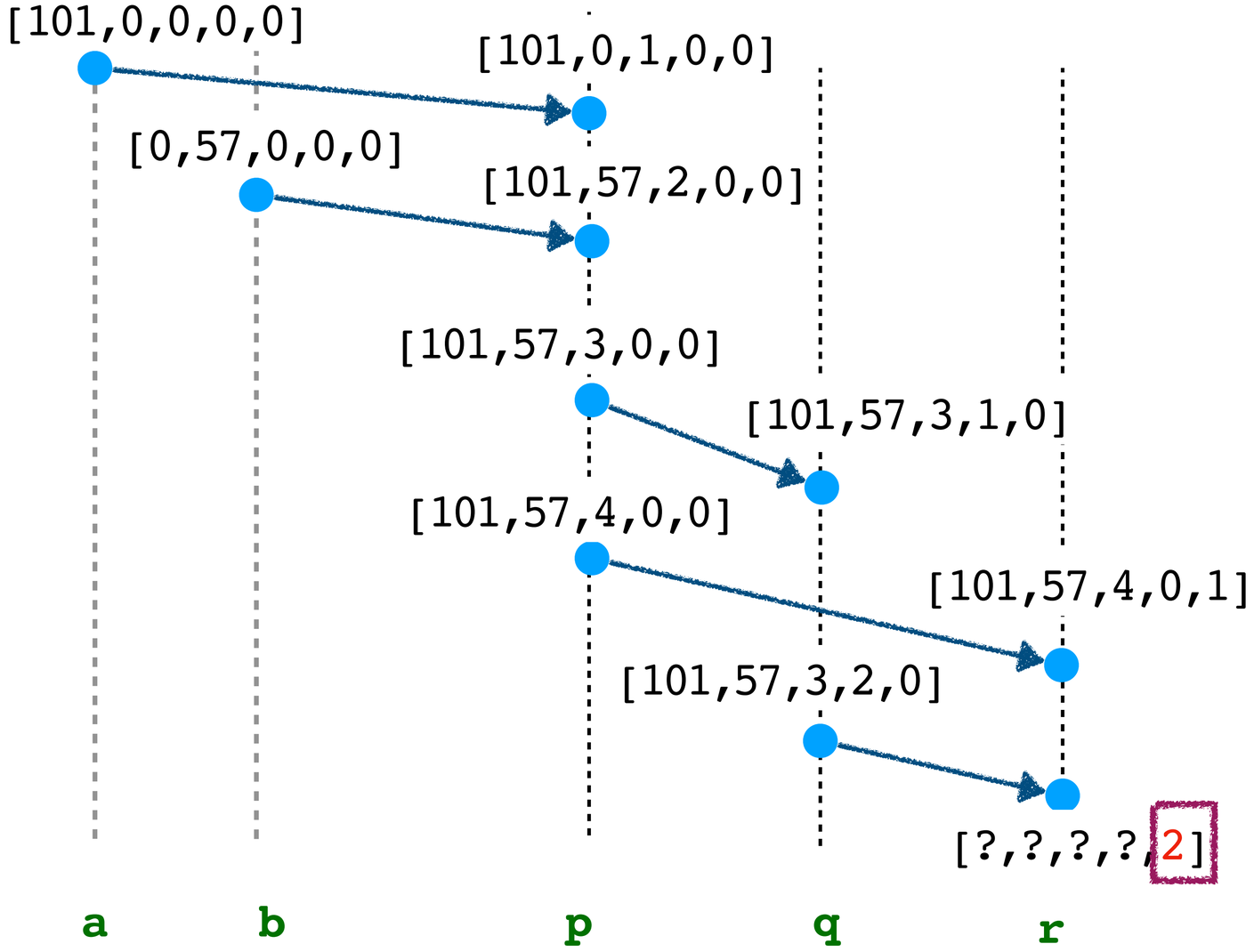
Provenance using **Transitivity**



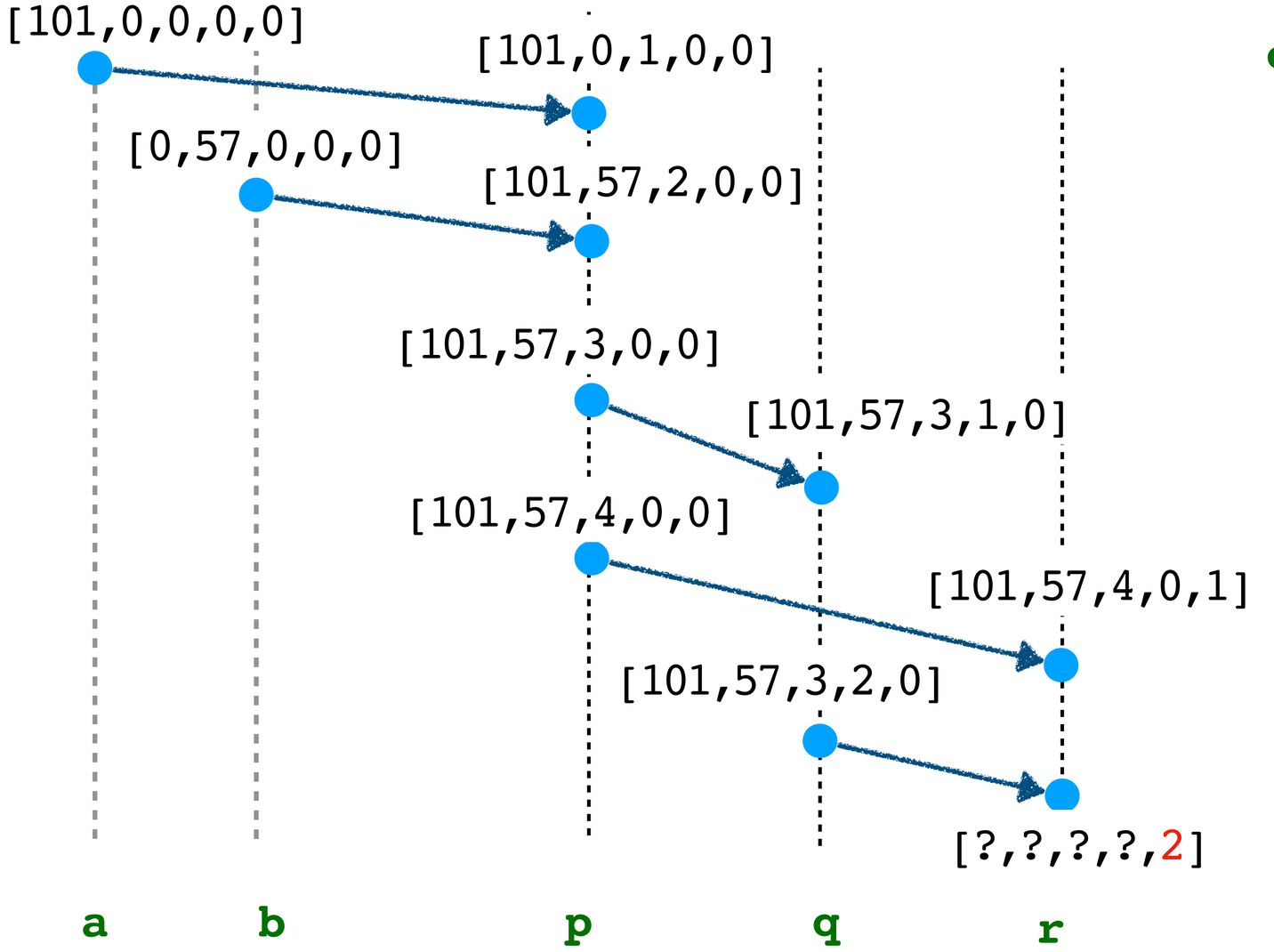
Provenance using **Transitivity**



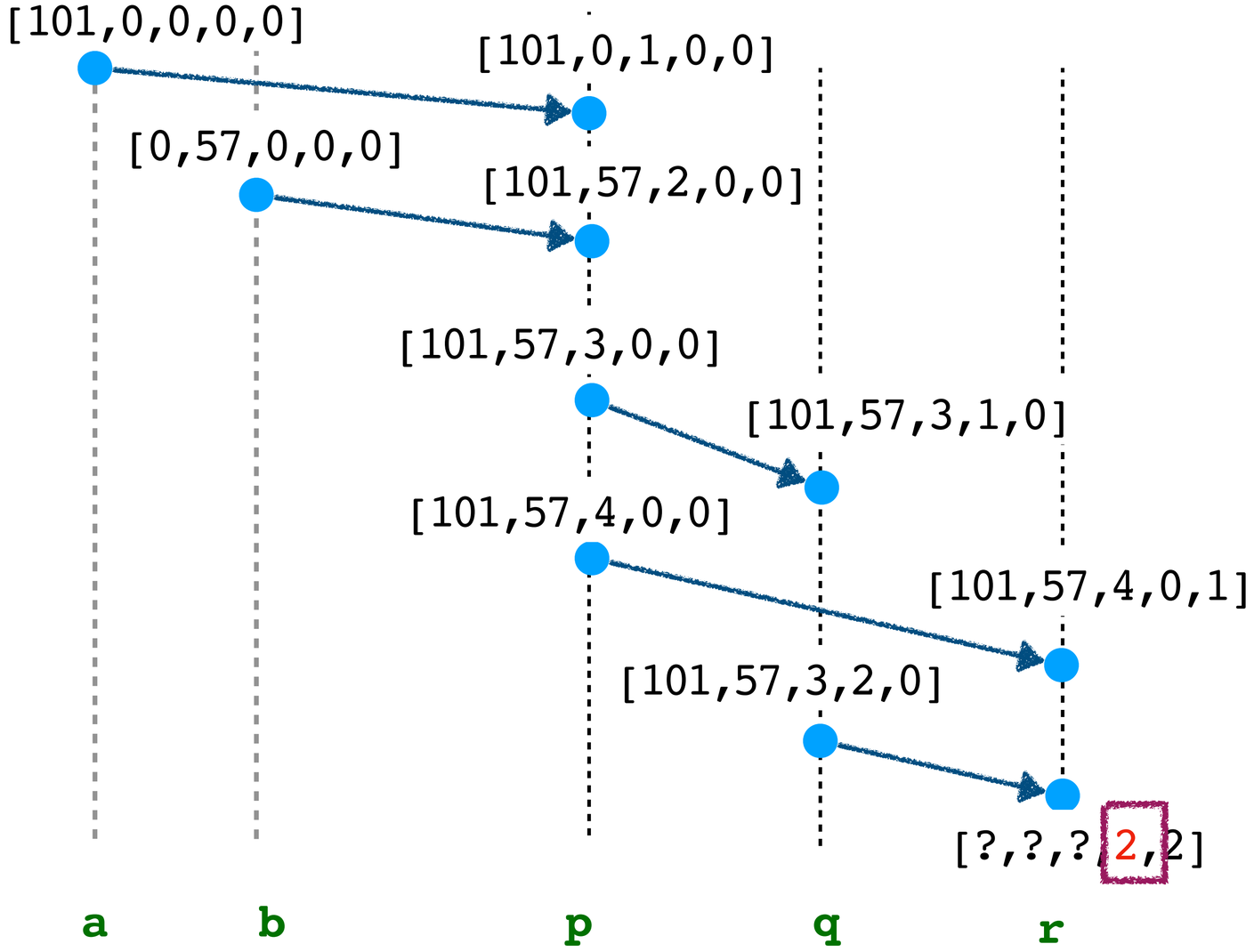
Provenance using **Transitivity**

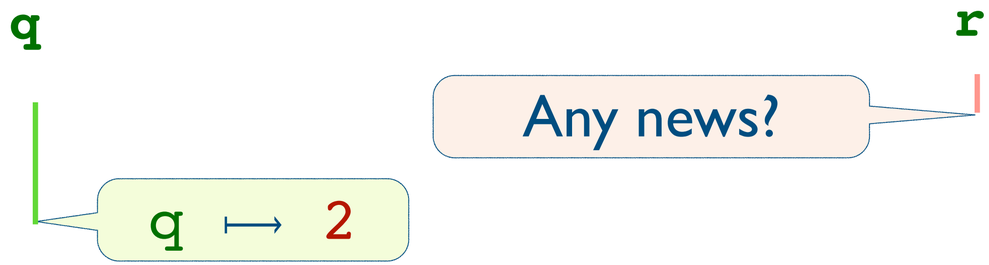


Provenance using **Transitivity**

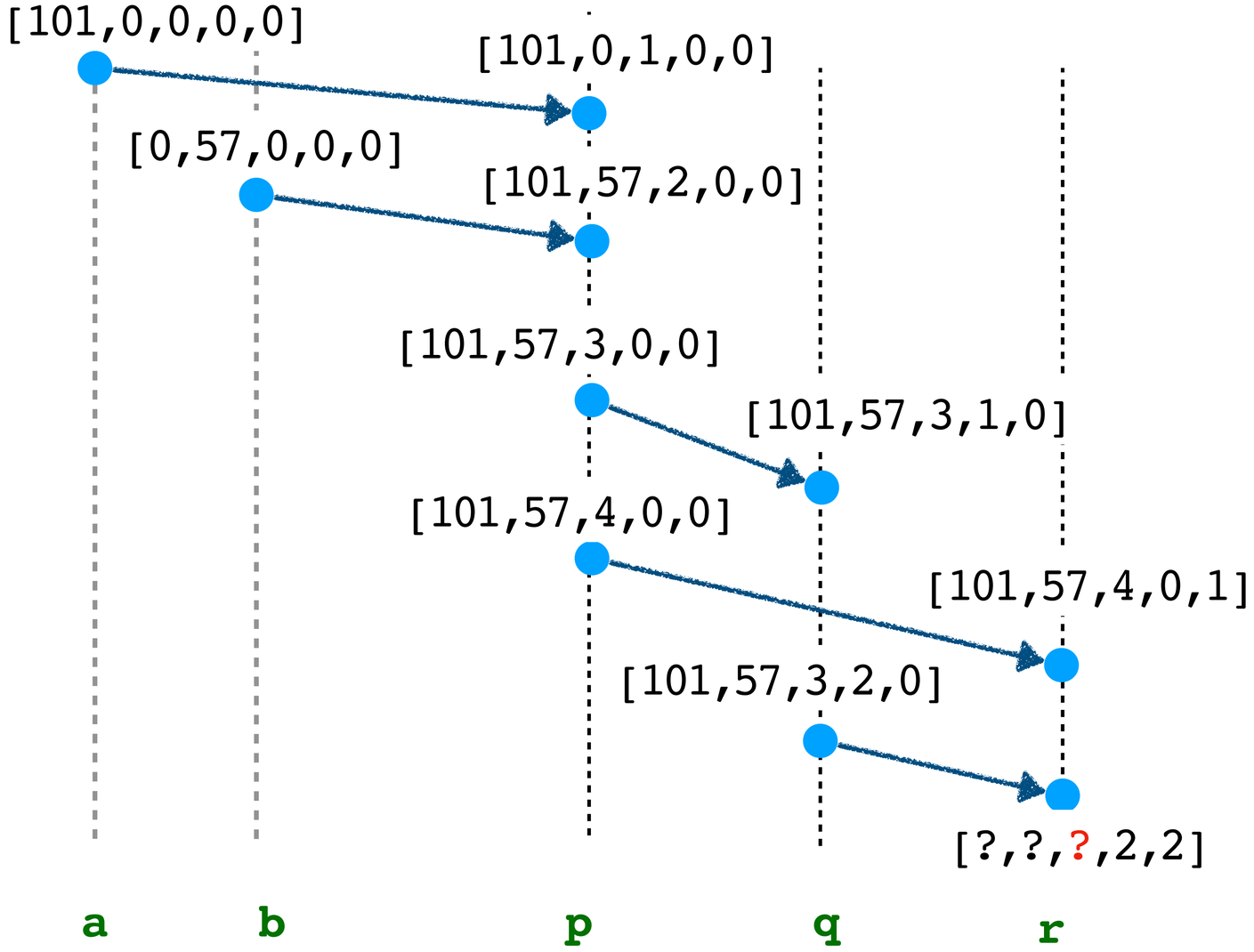


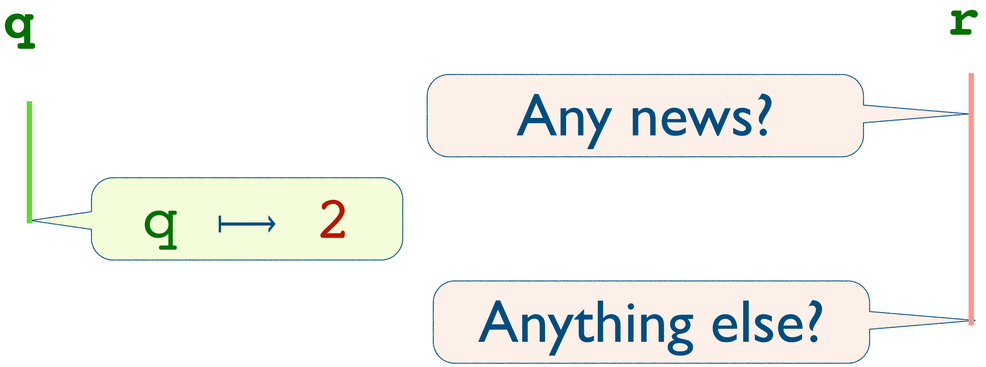
Provenance using **Transitivity**



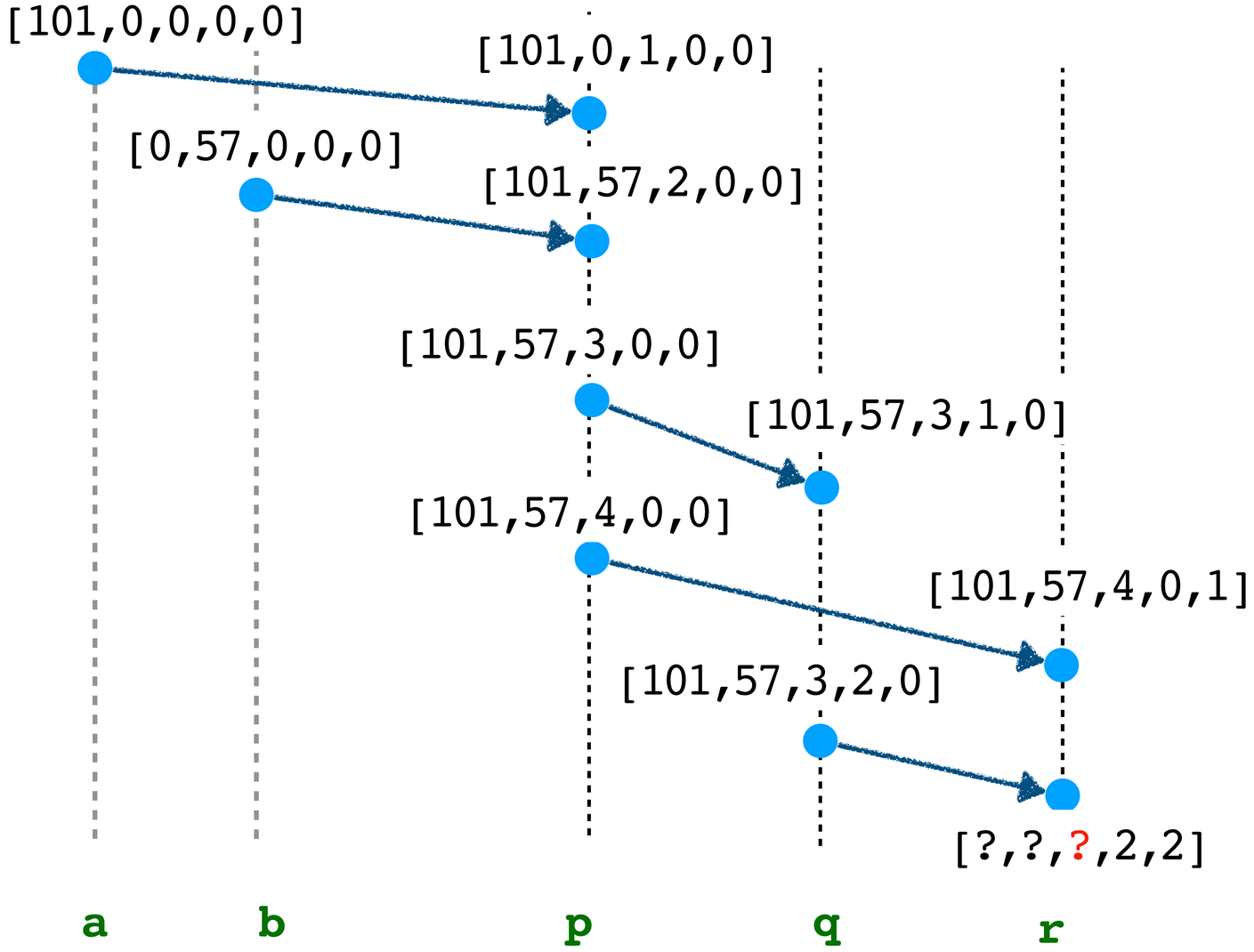


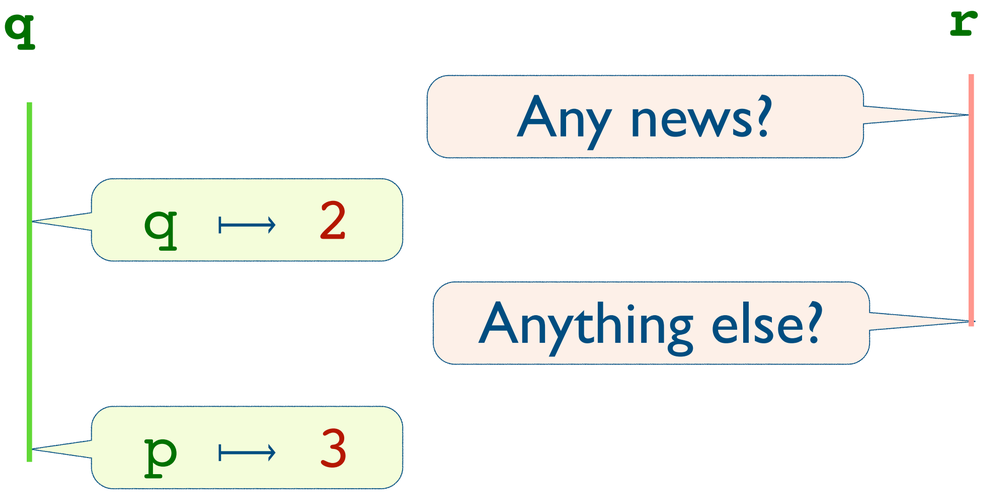
Provenance using **Transitivity**



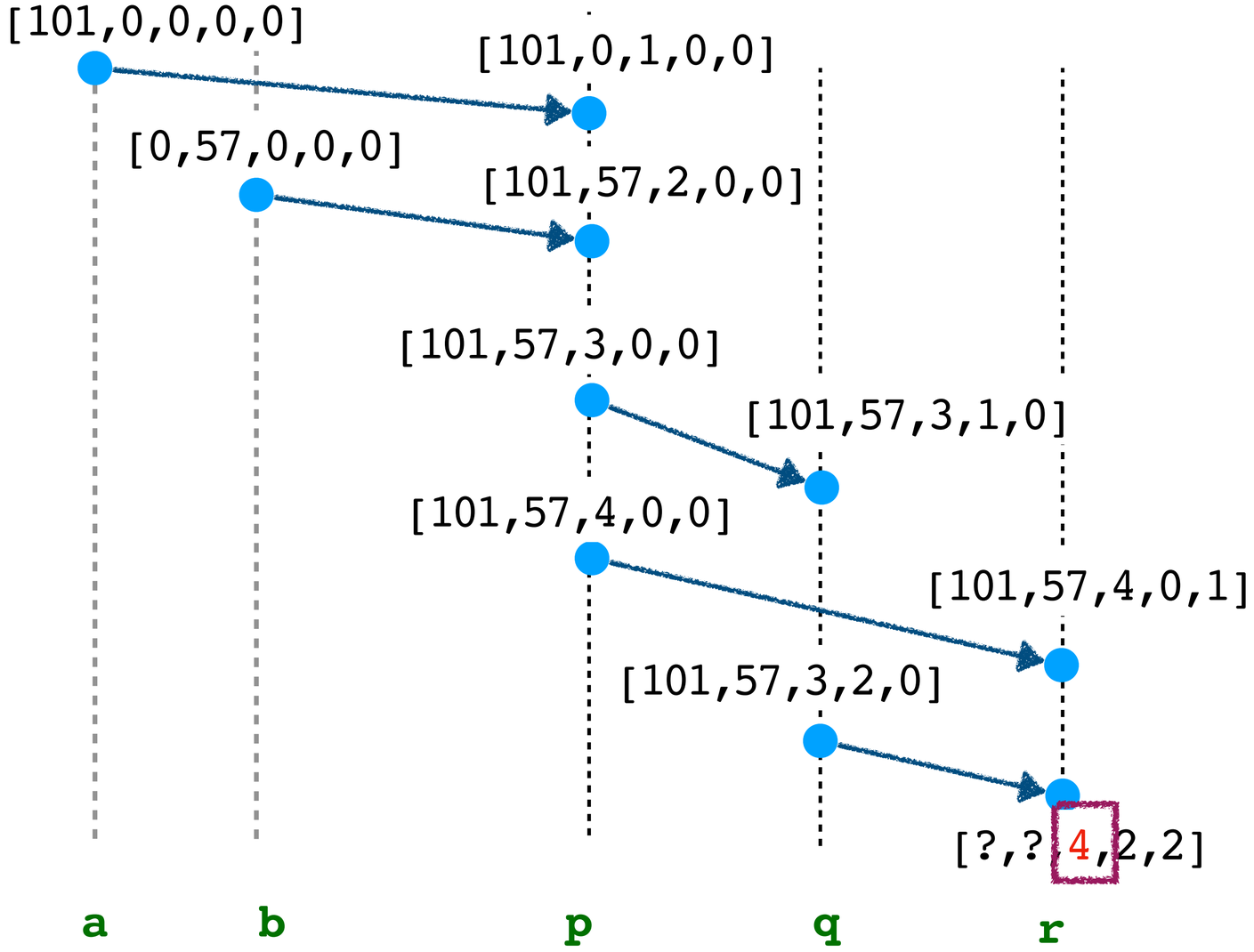


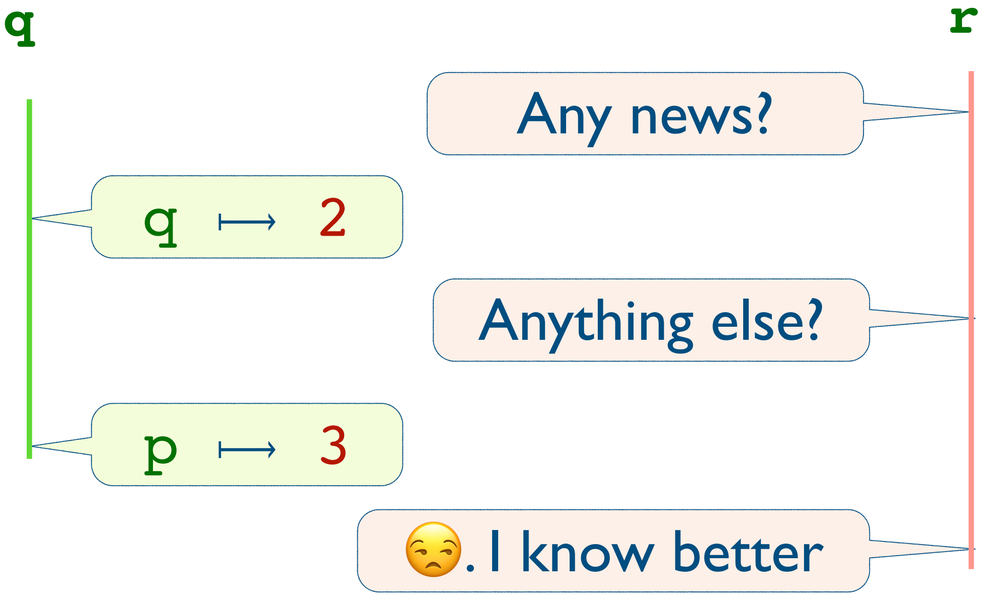
Provenance using **Transitivity**



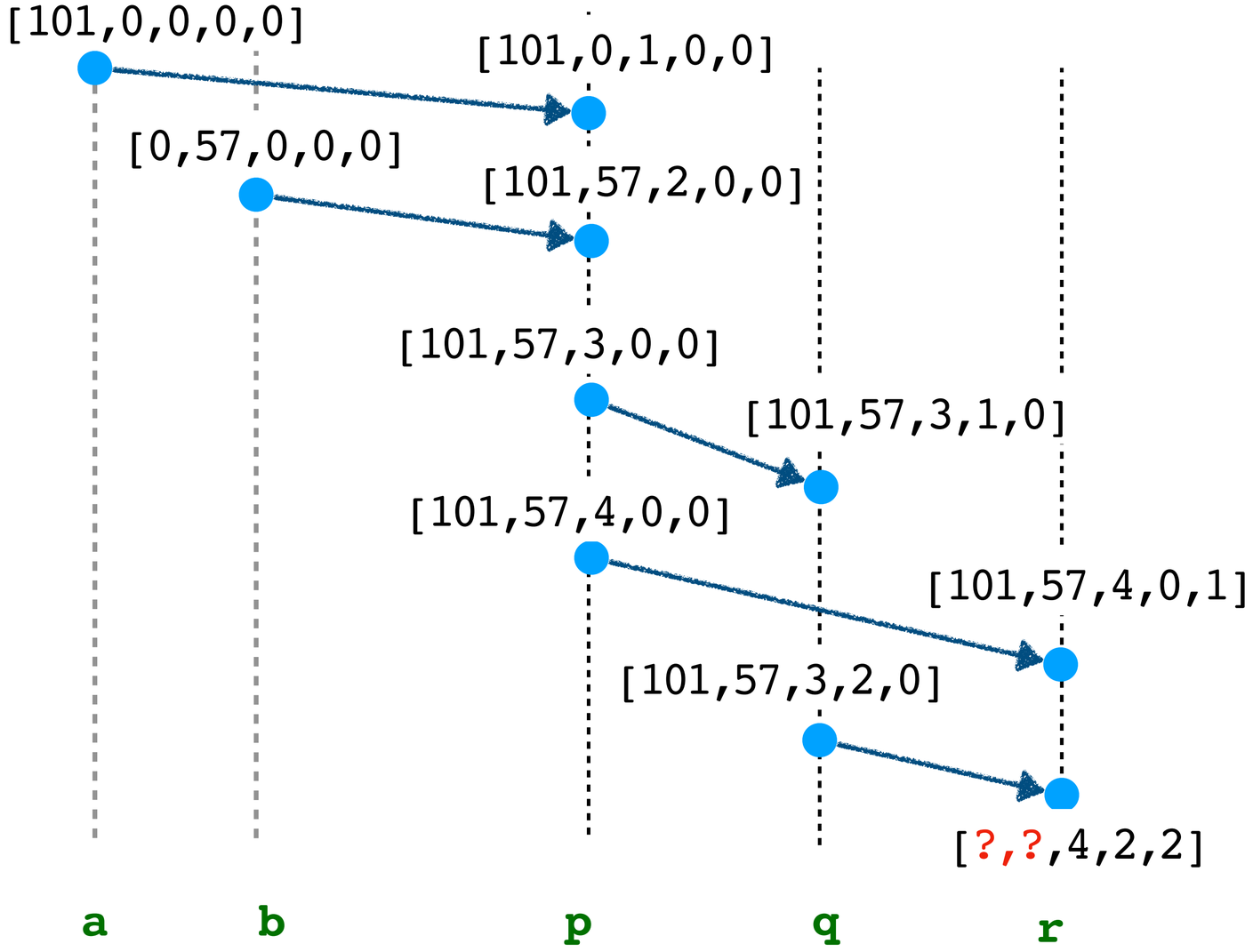


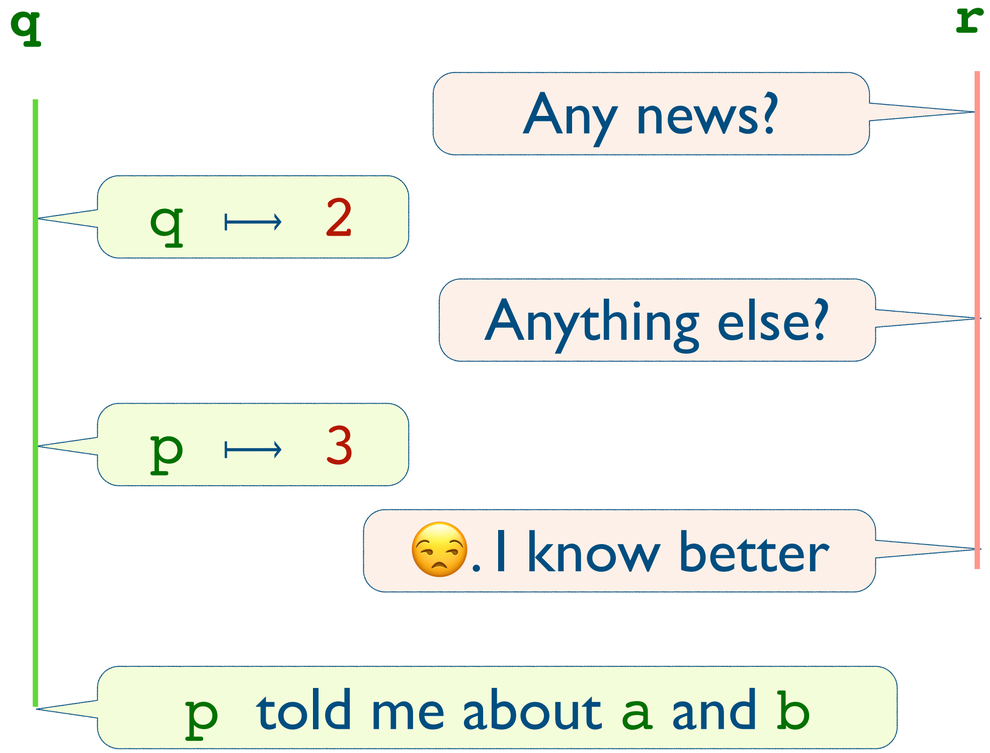
Provenance using **Transitivity**



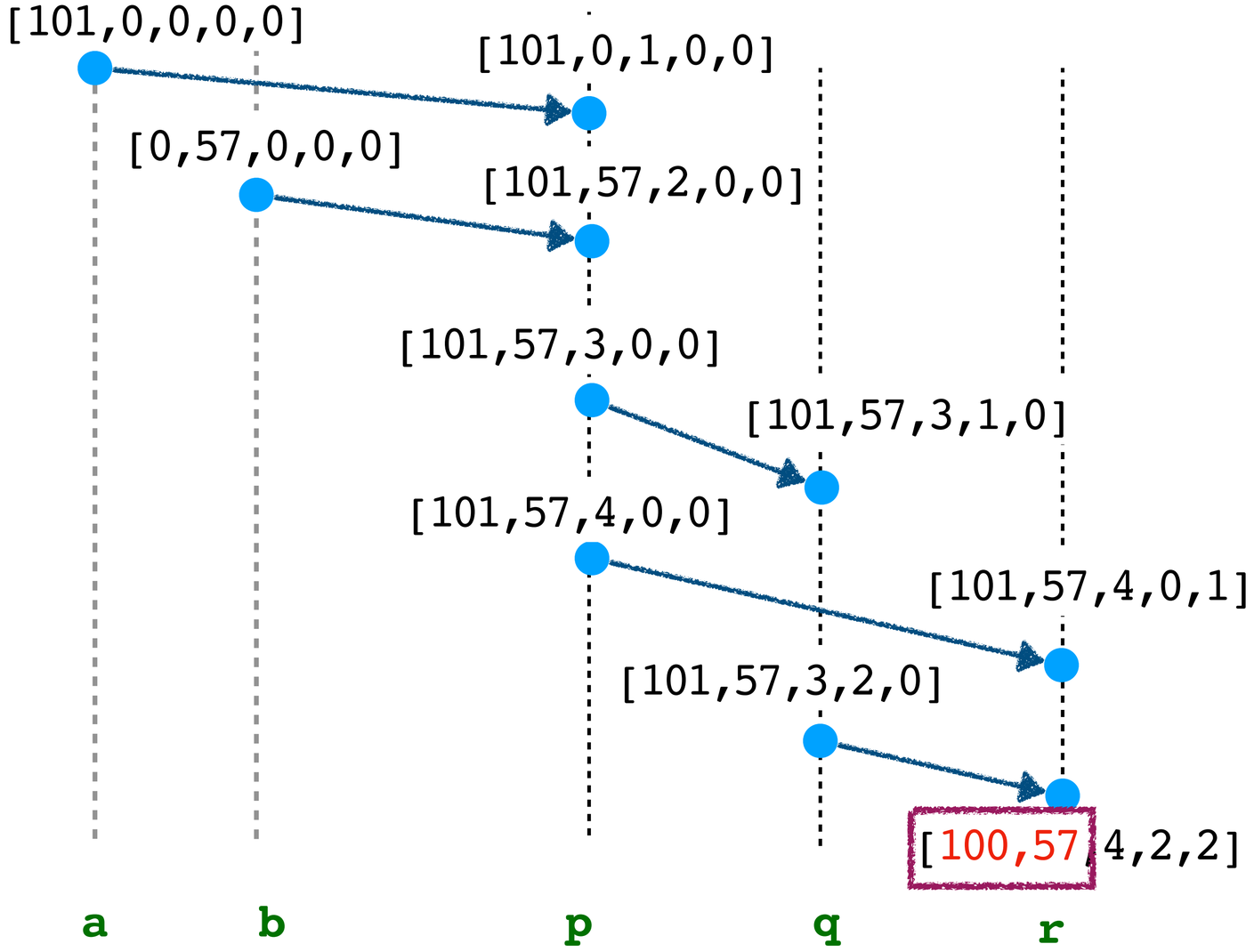


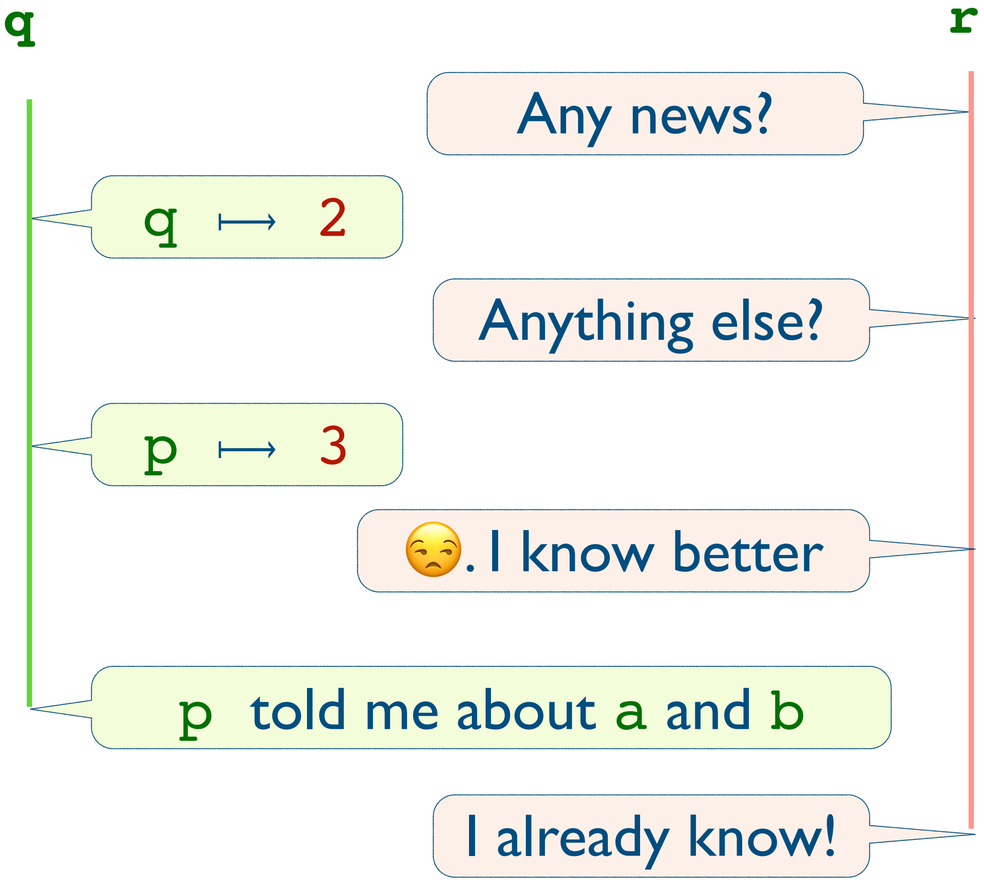
Provenance using **Transitivity**



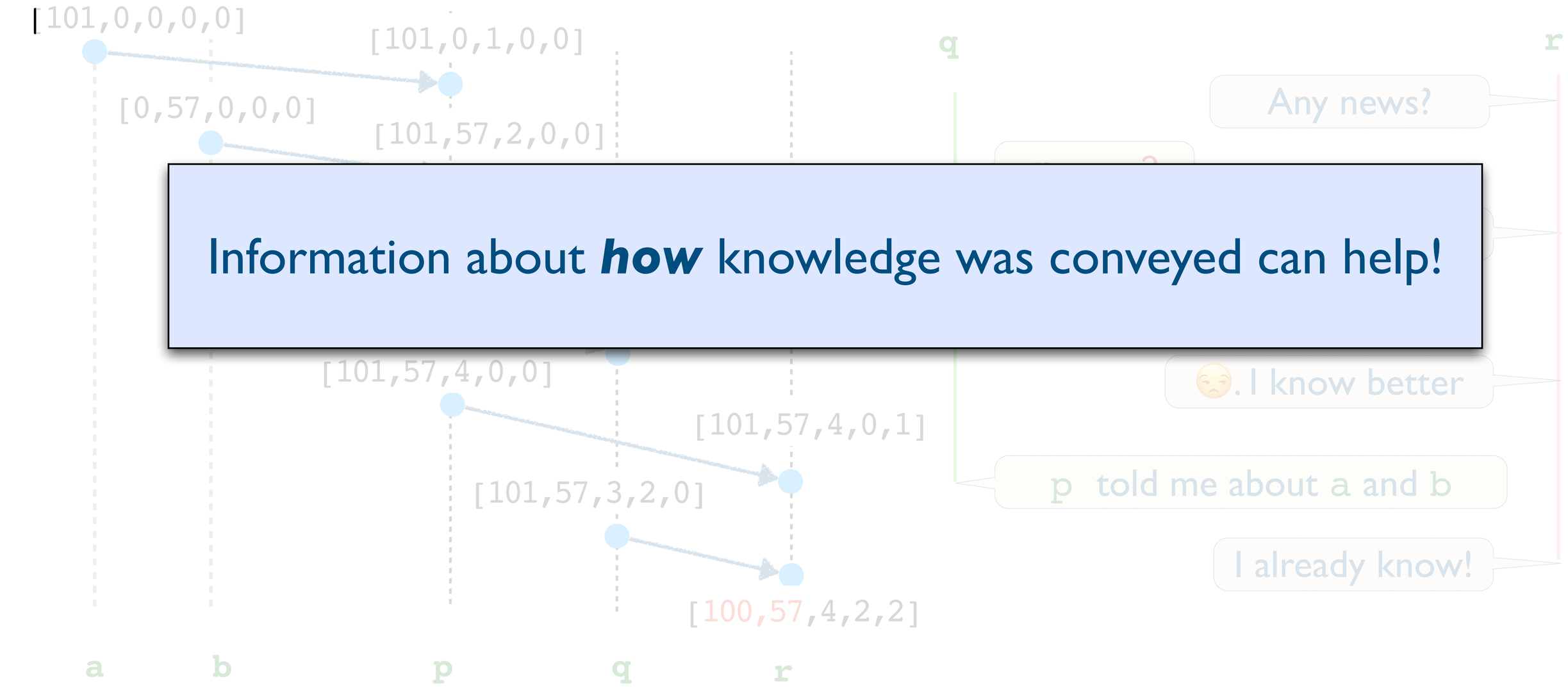


Provenance using **Transitivity**



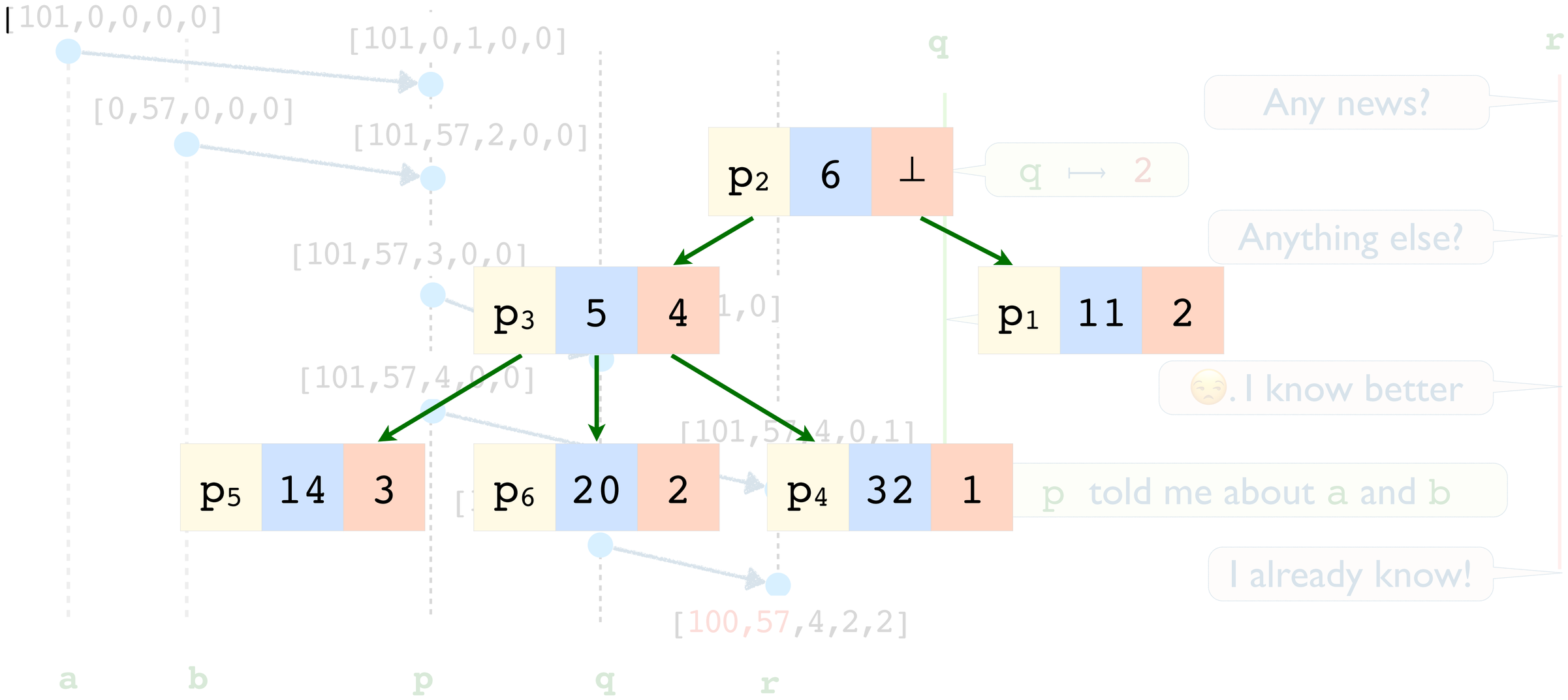


Provenance using **Transitivity**

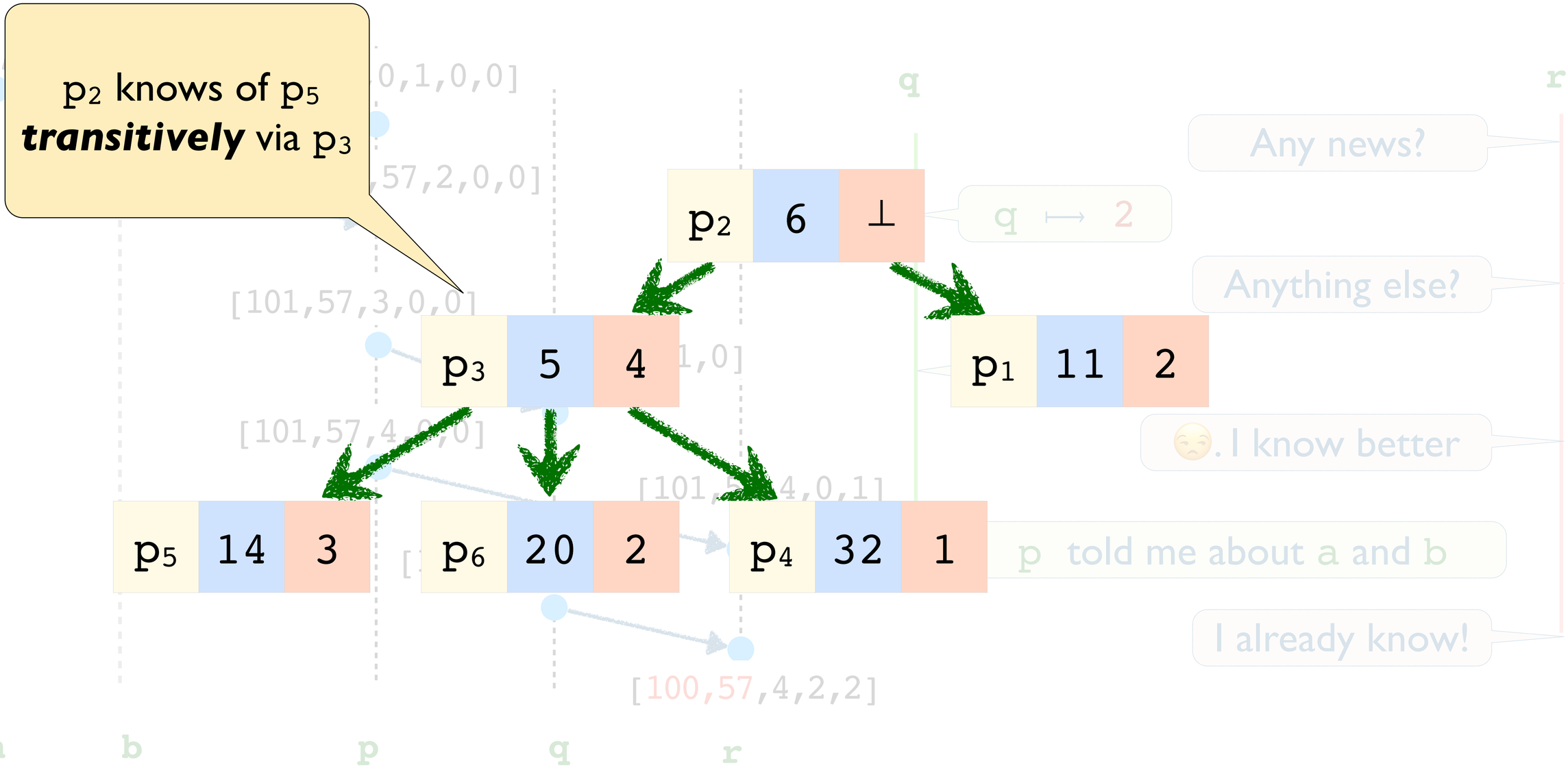


Information about ***how*** knowledge was conveyed can help!

Provenance using **Transitivity**

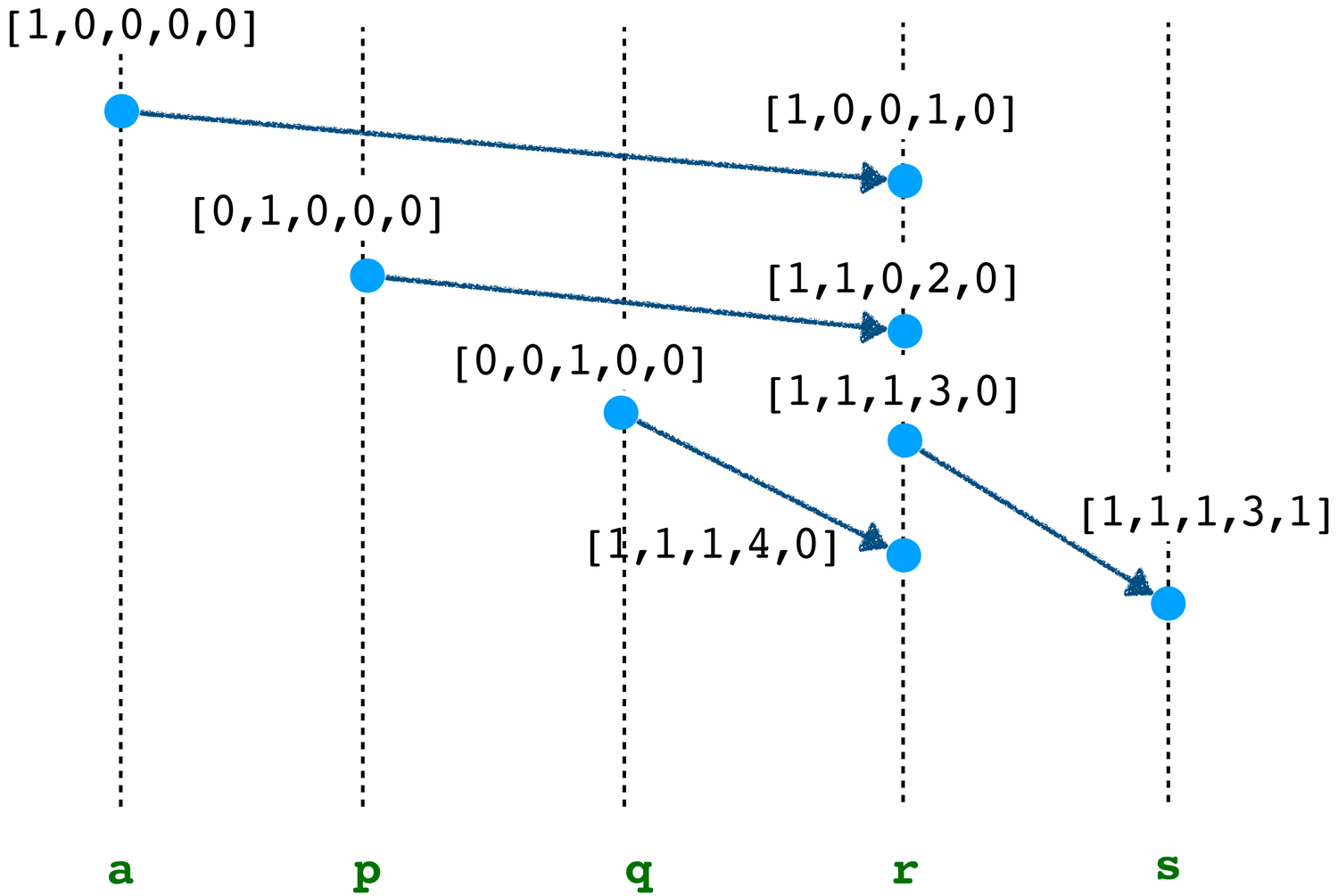


Provenance using **Transitivity**

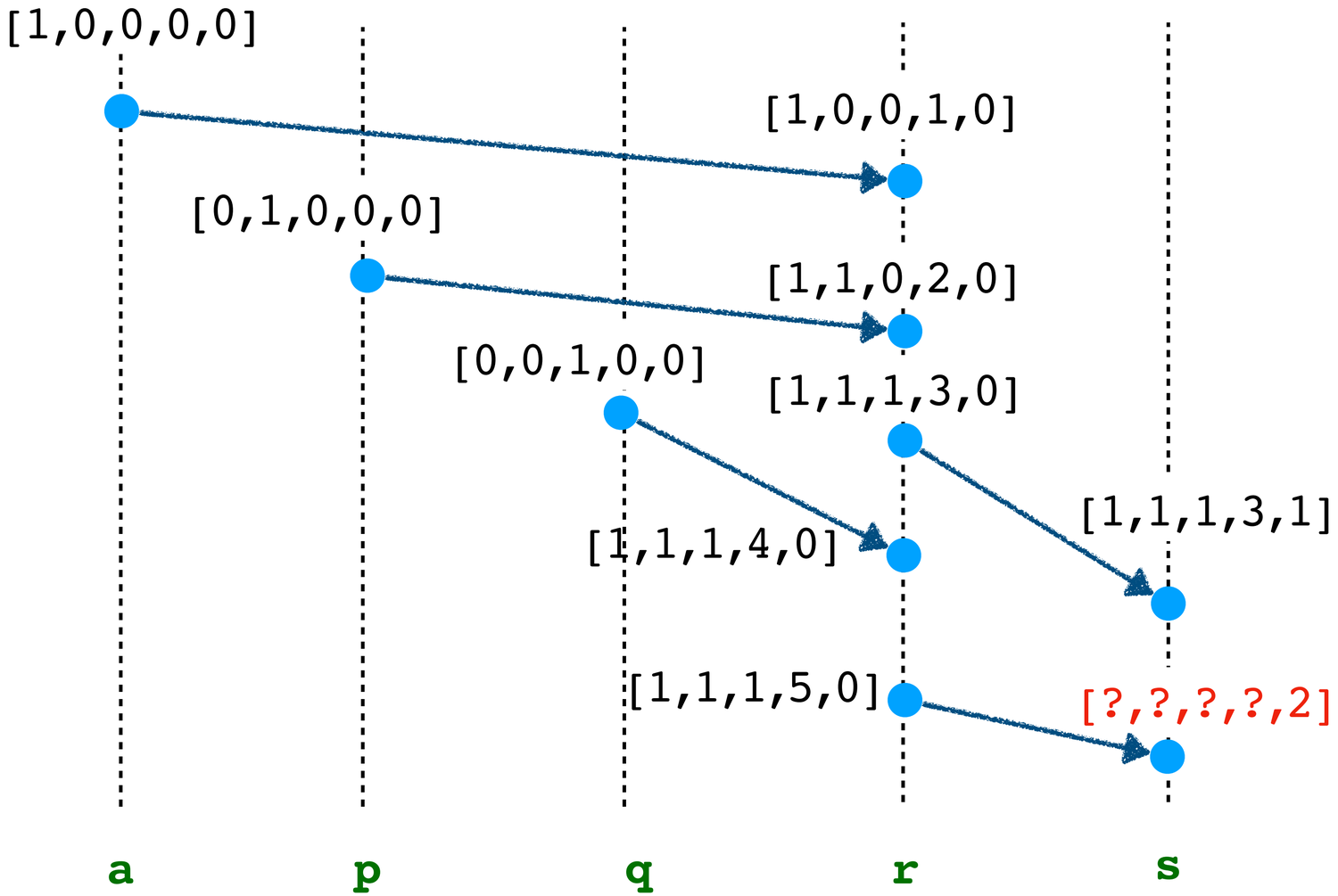


Provenance using **Time of Arrival**

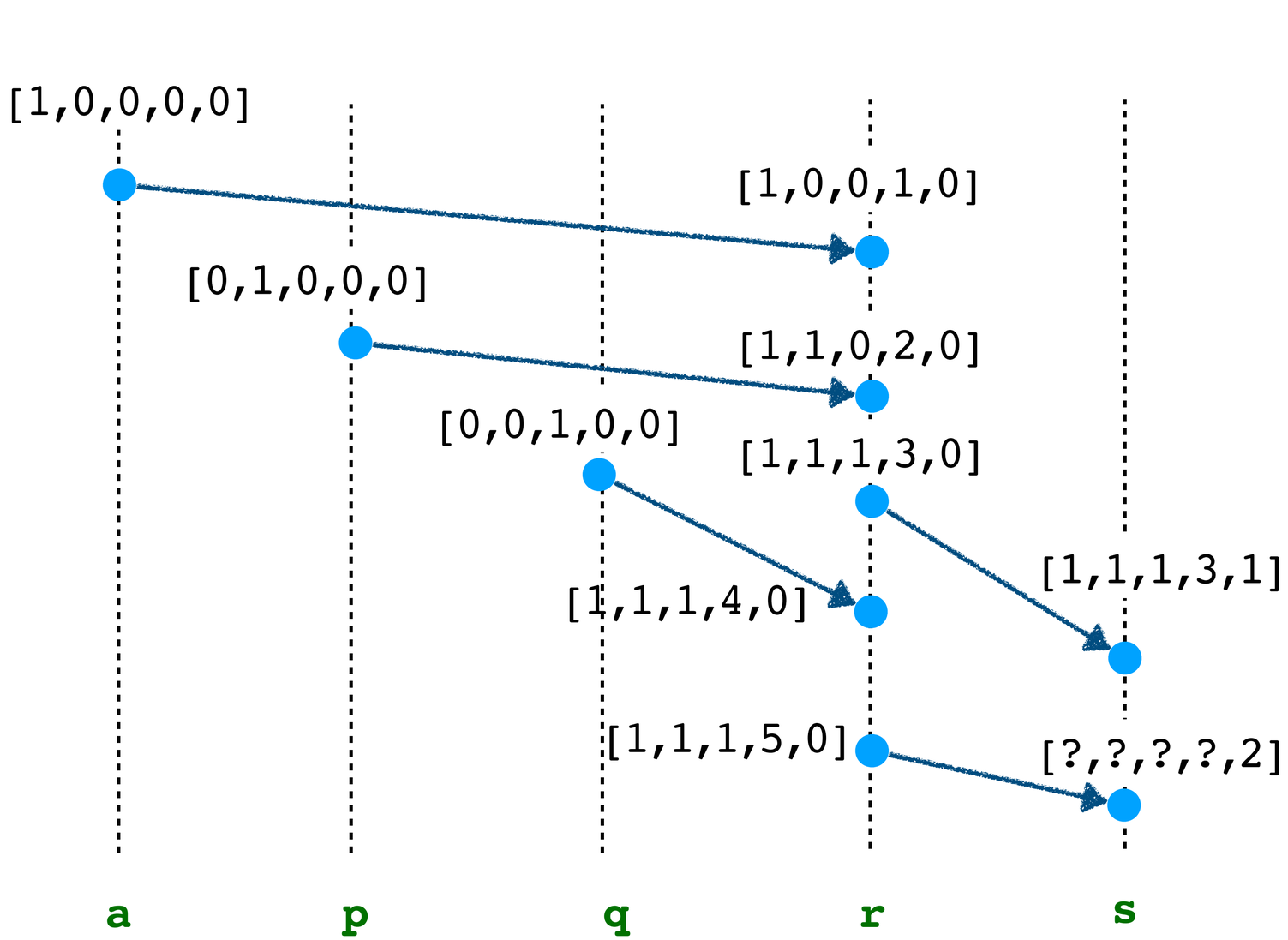
Provenance using **Time of Arrival**



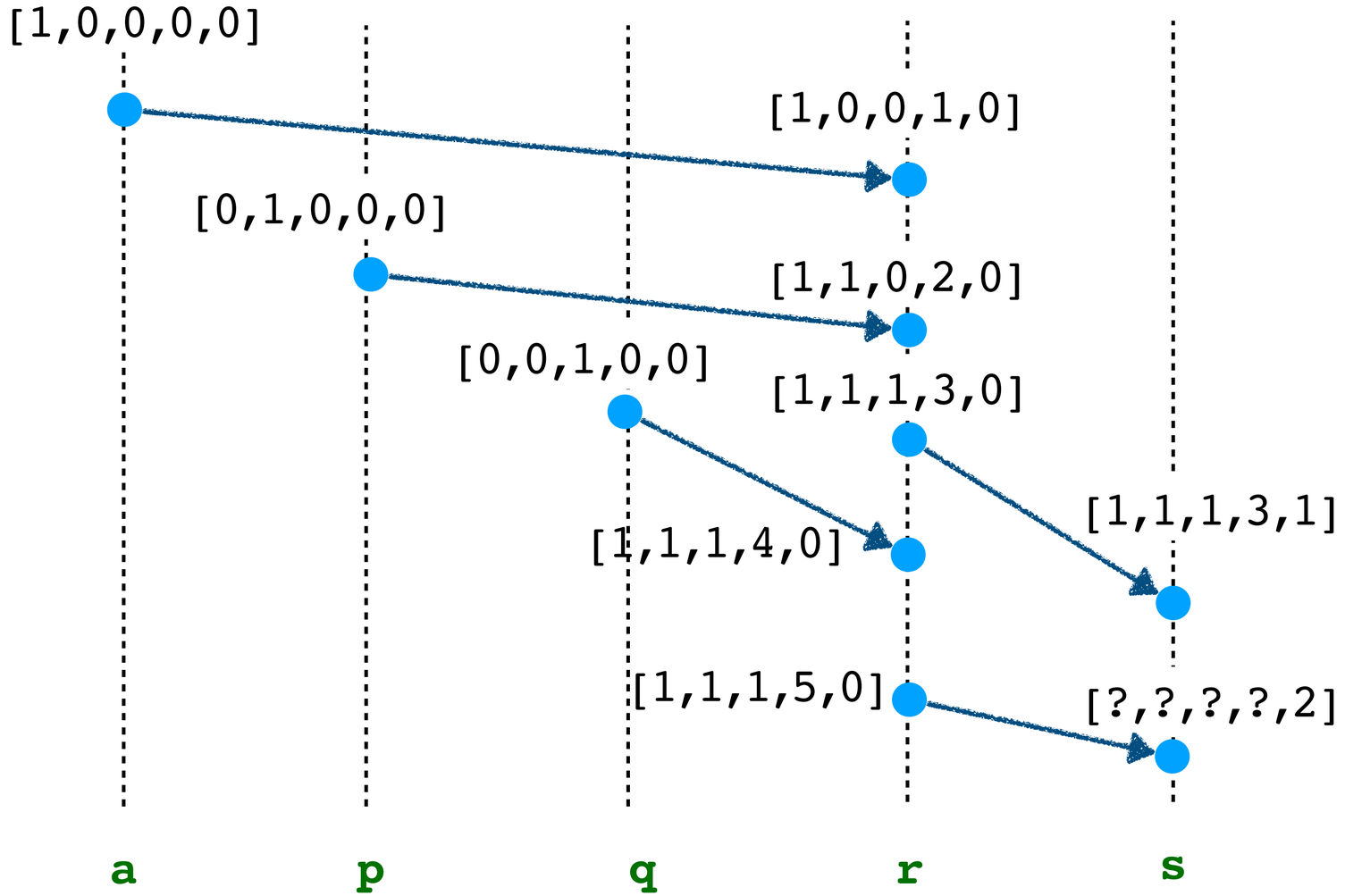
Provenance using **Time of Arrival**



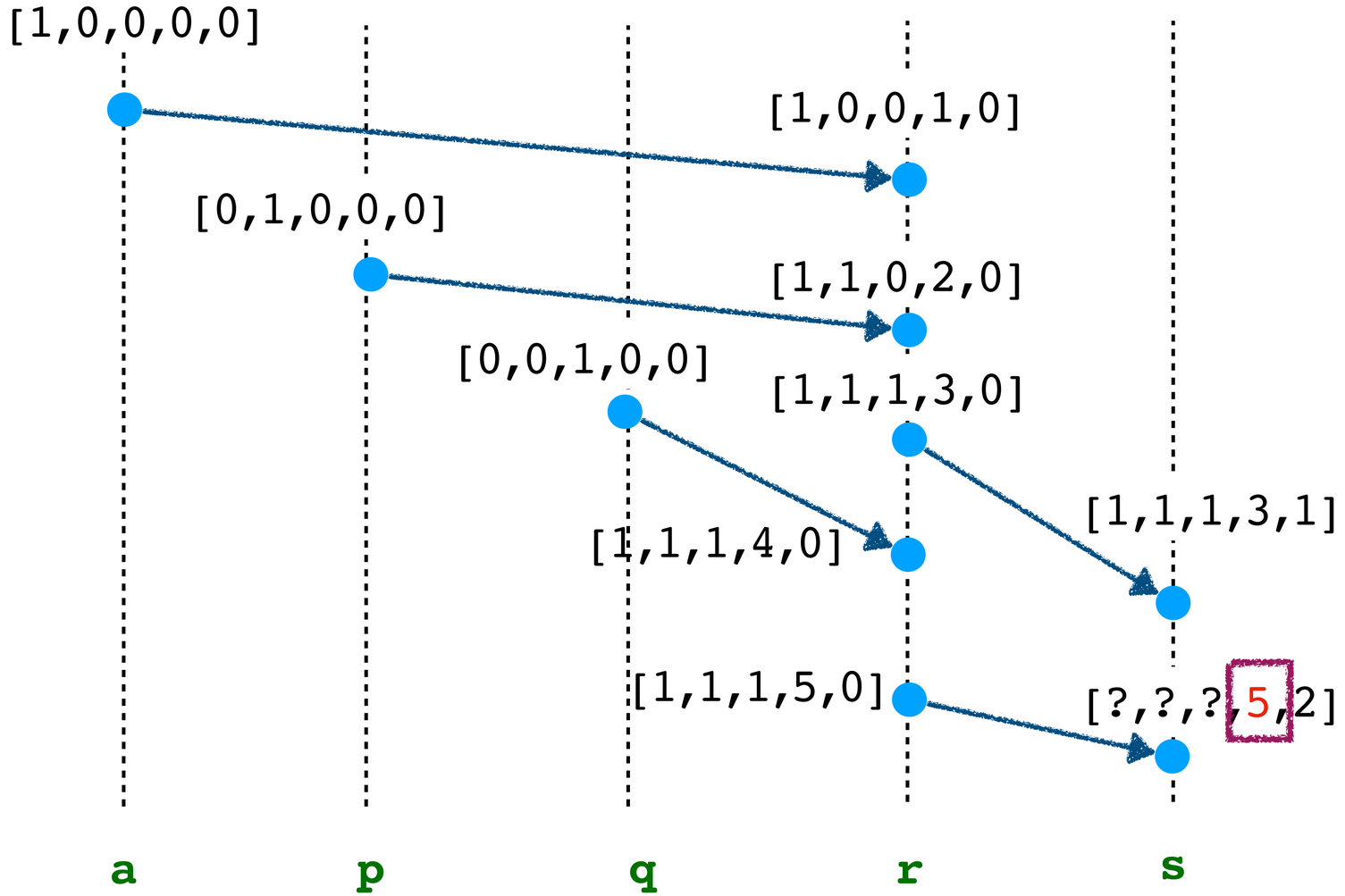
Provenance using **Time of Arrival**



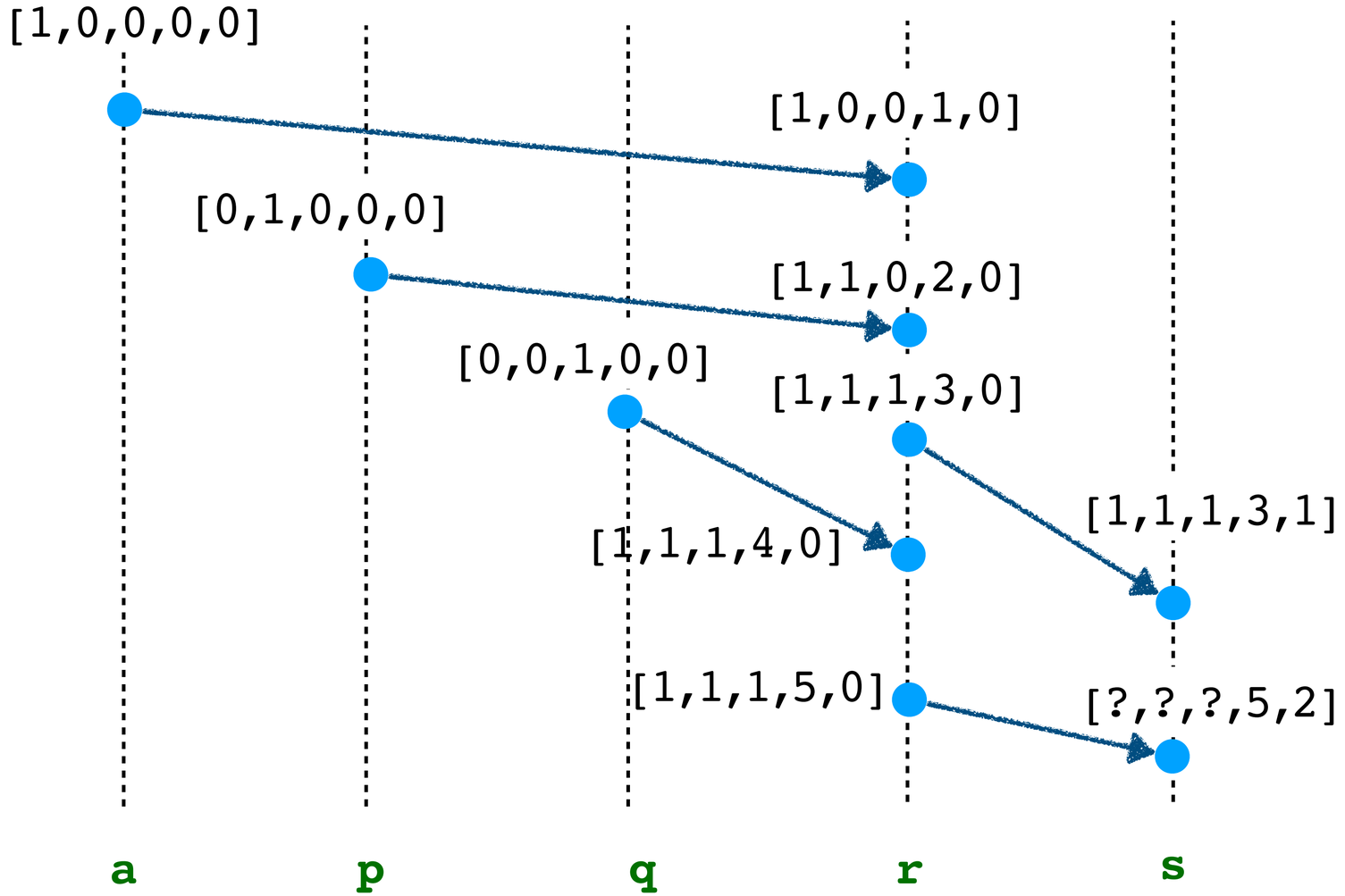
Provenance using **Time of Arrival**

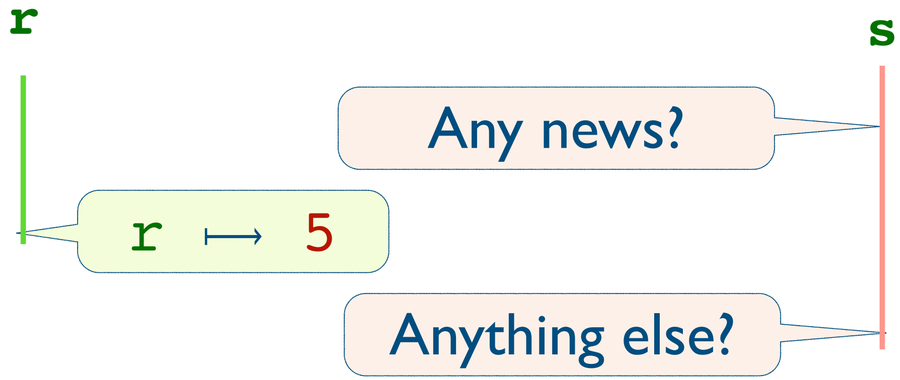


Provenance using **Time of Arrival**

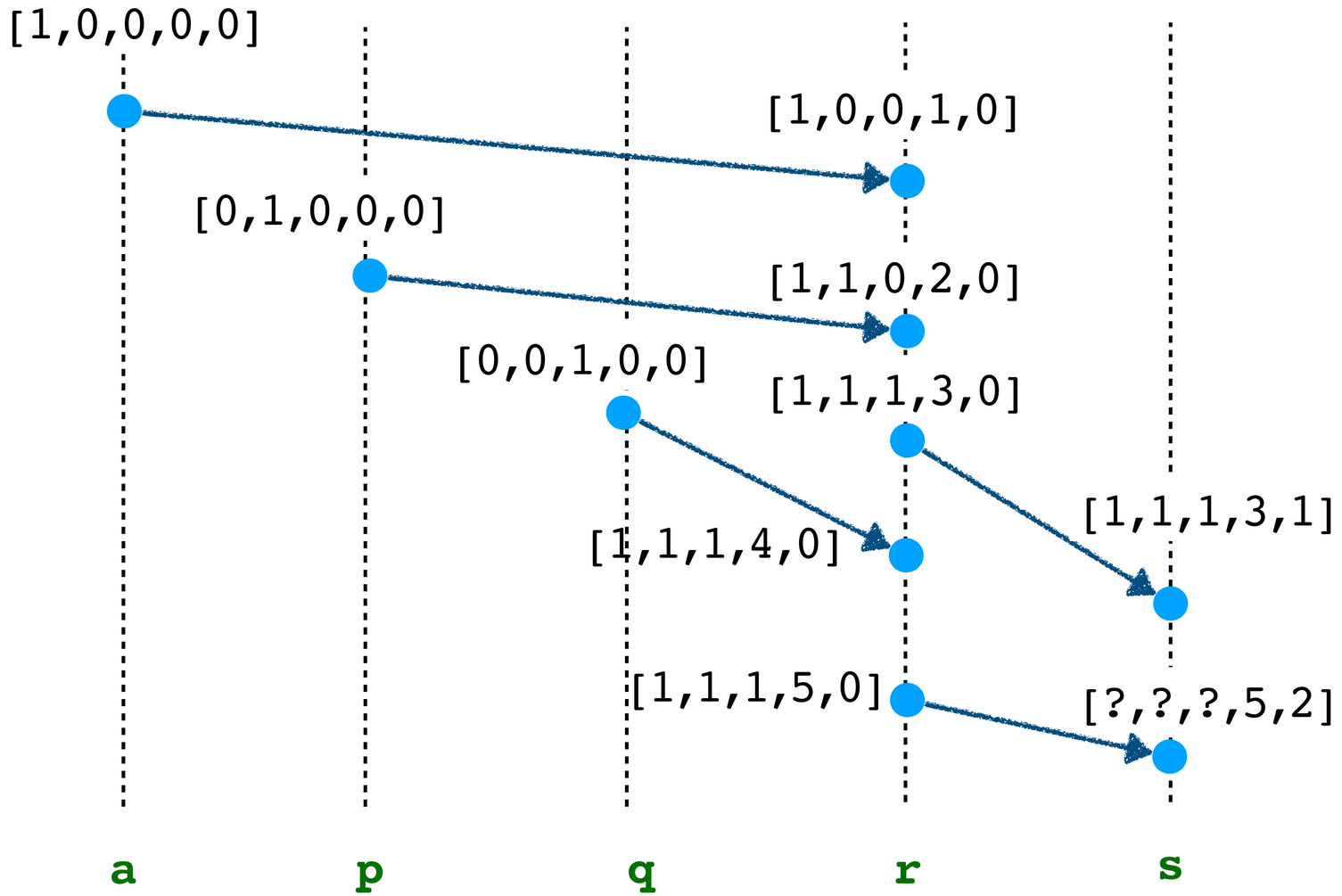


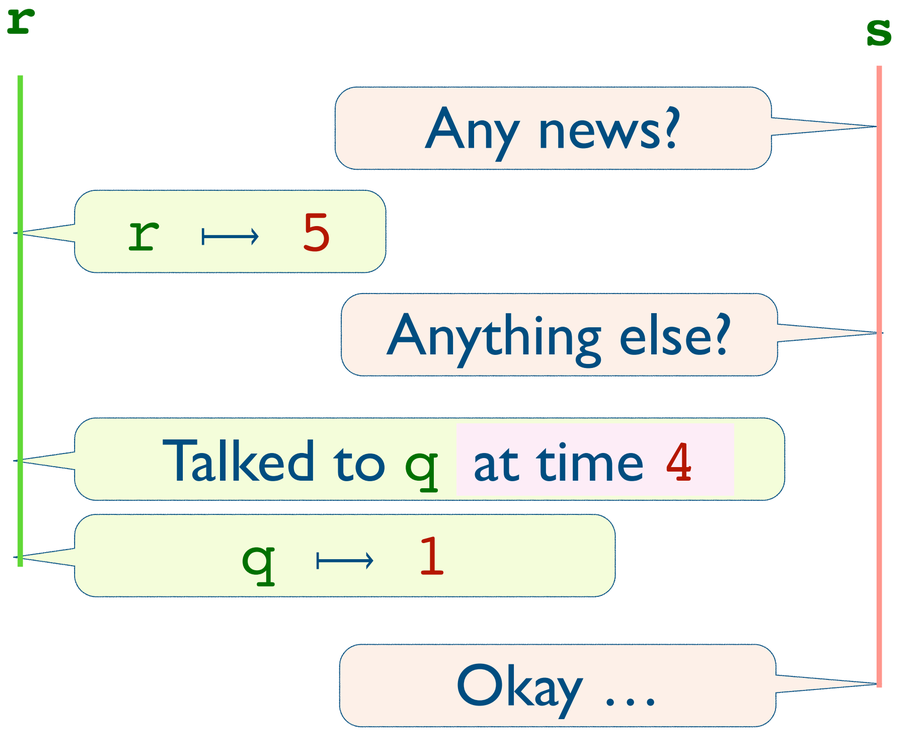
Provenance using **Time of Arrival**



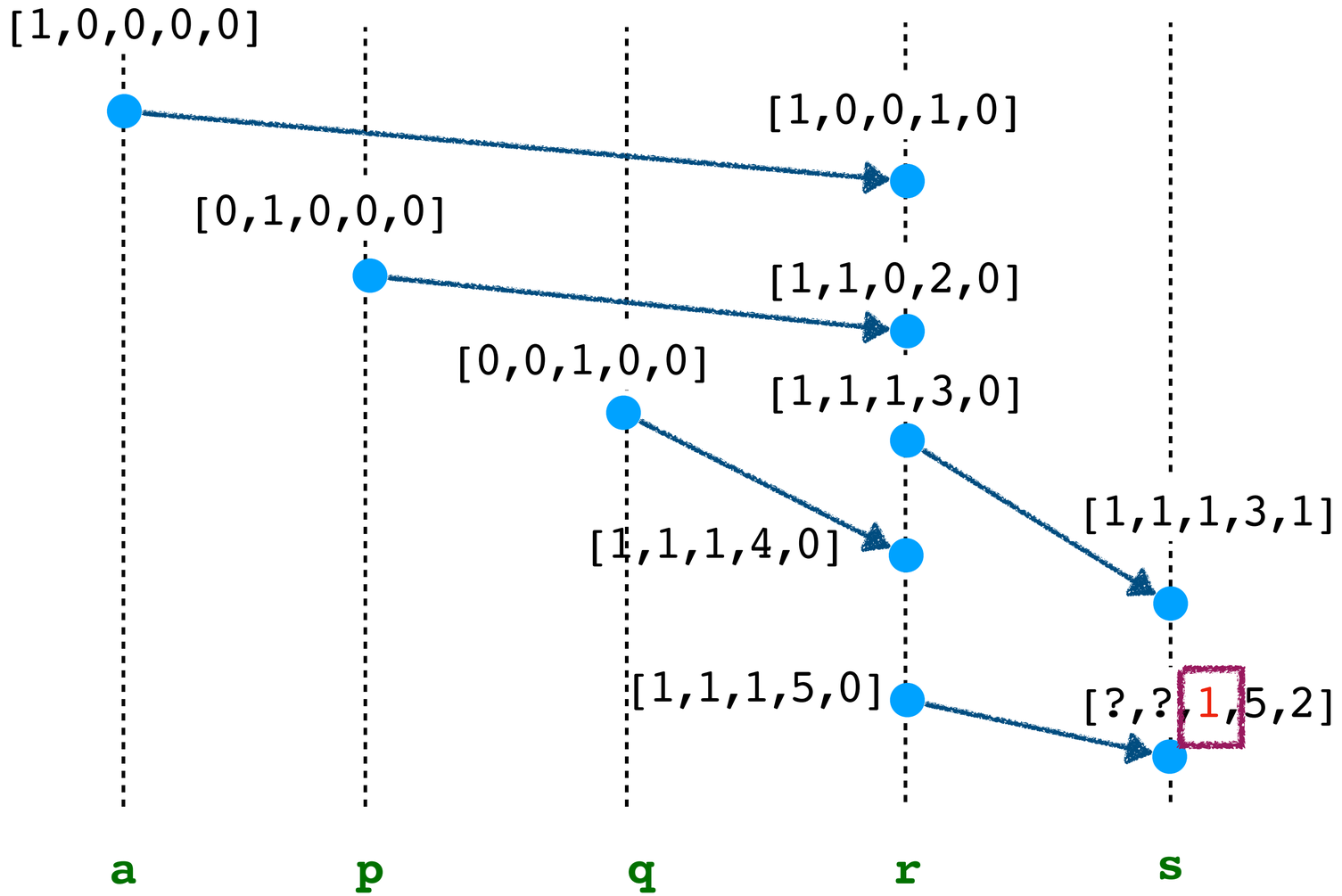


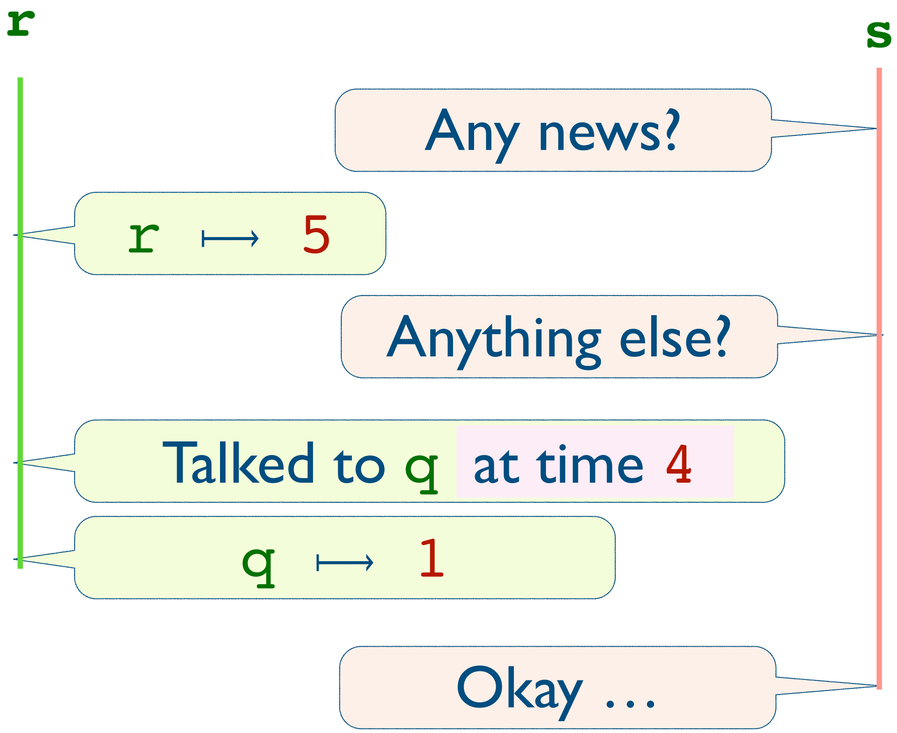
Provenance using **Time of Arrival**



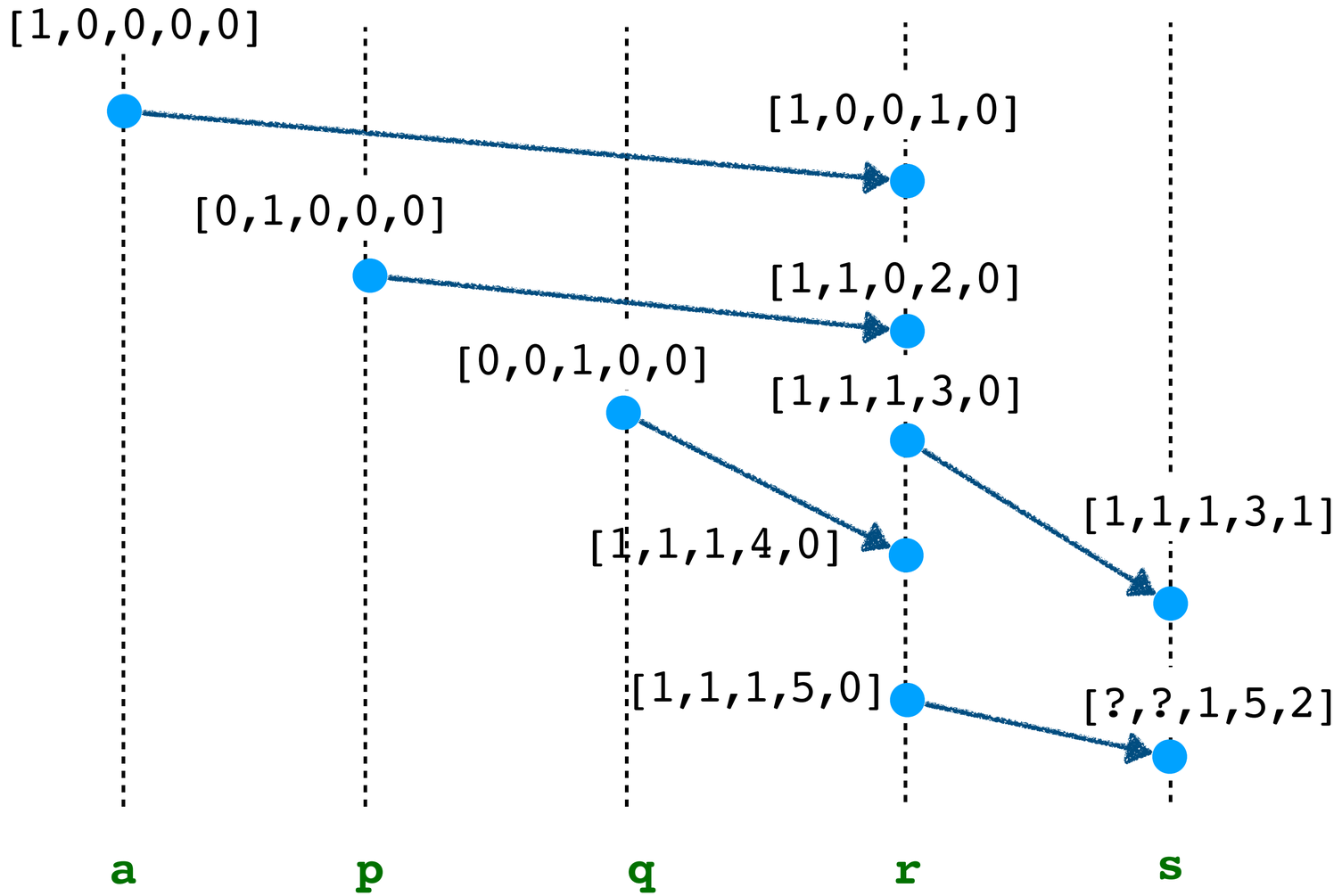


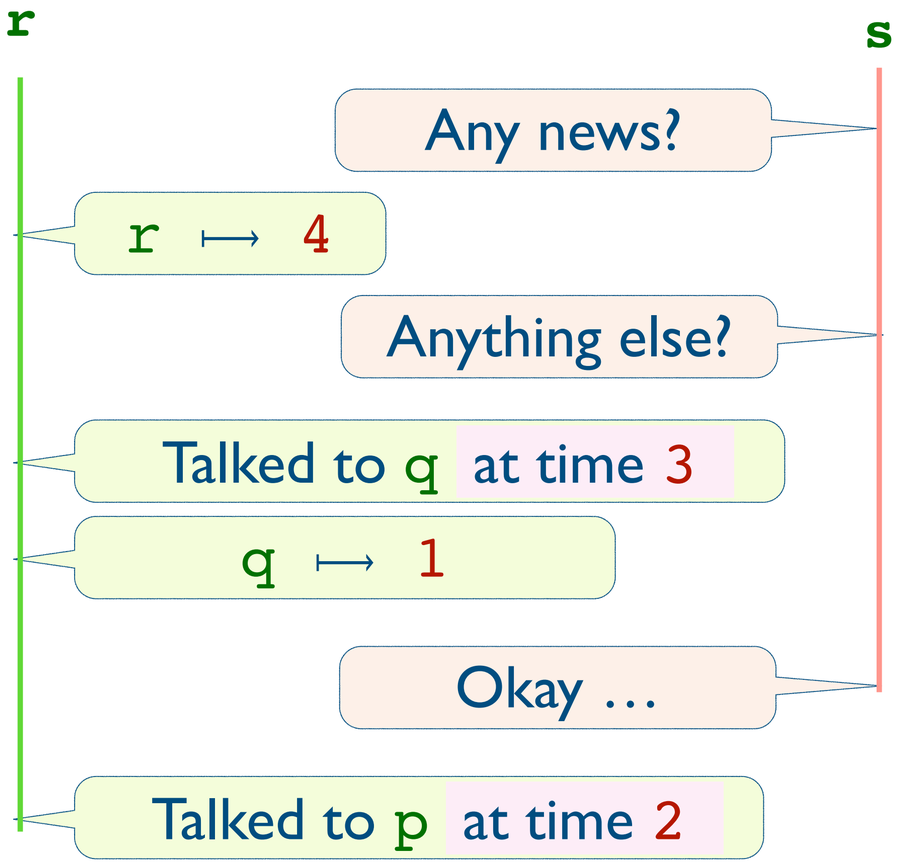
Provenance using **Time of Arrival**



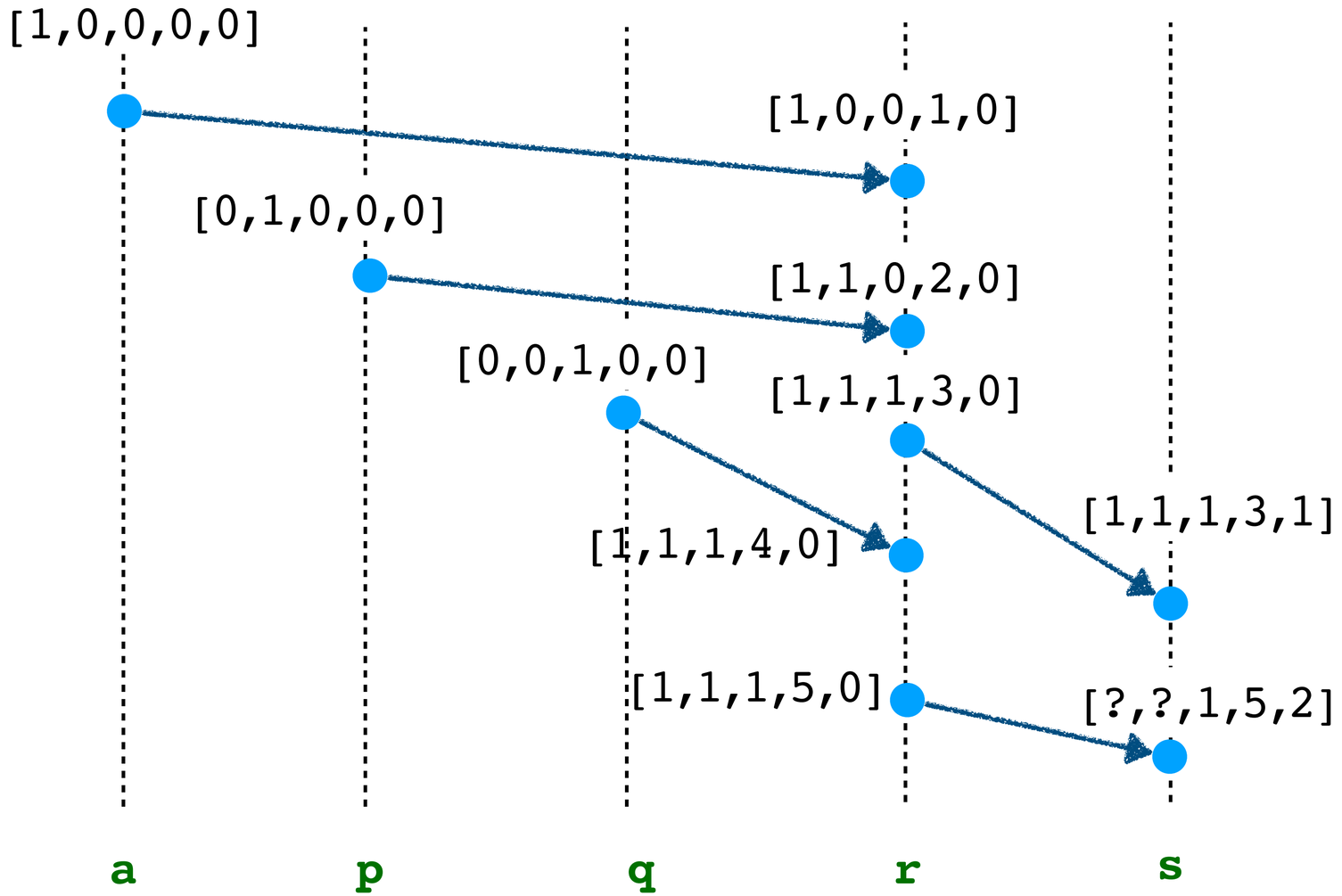


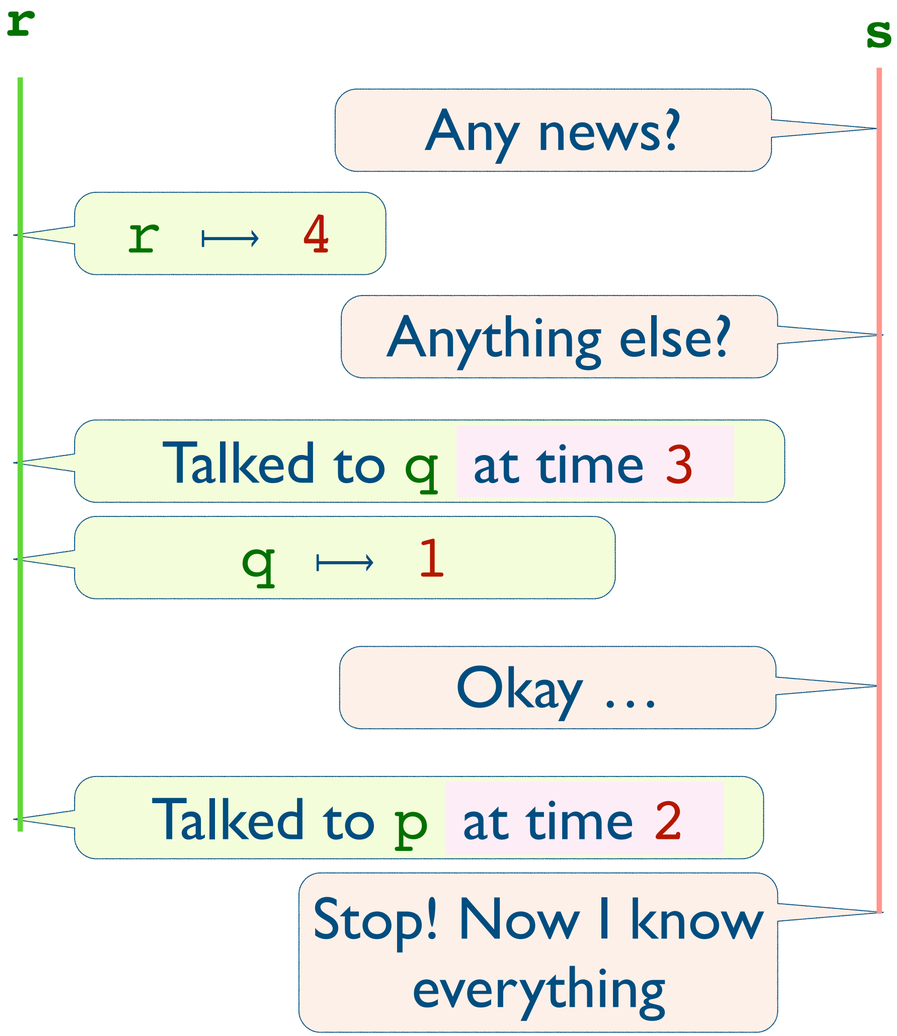
Provenance using **Time of Arrival**



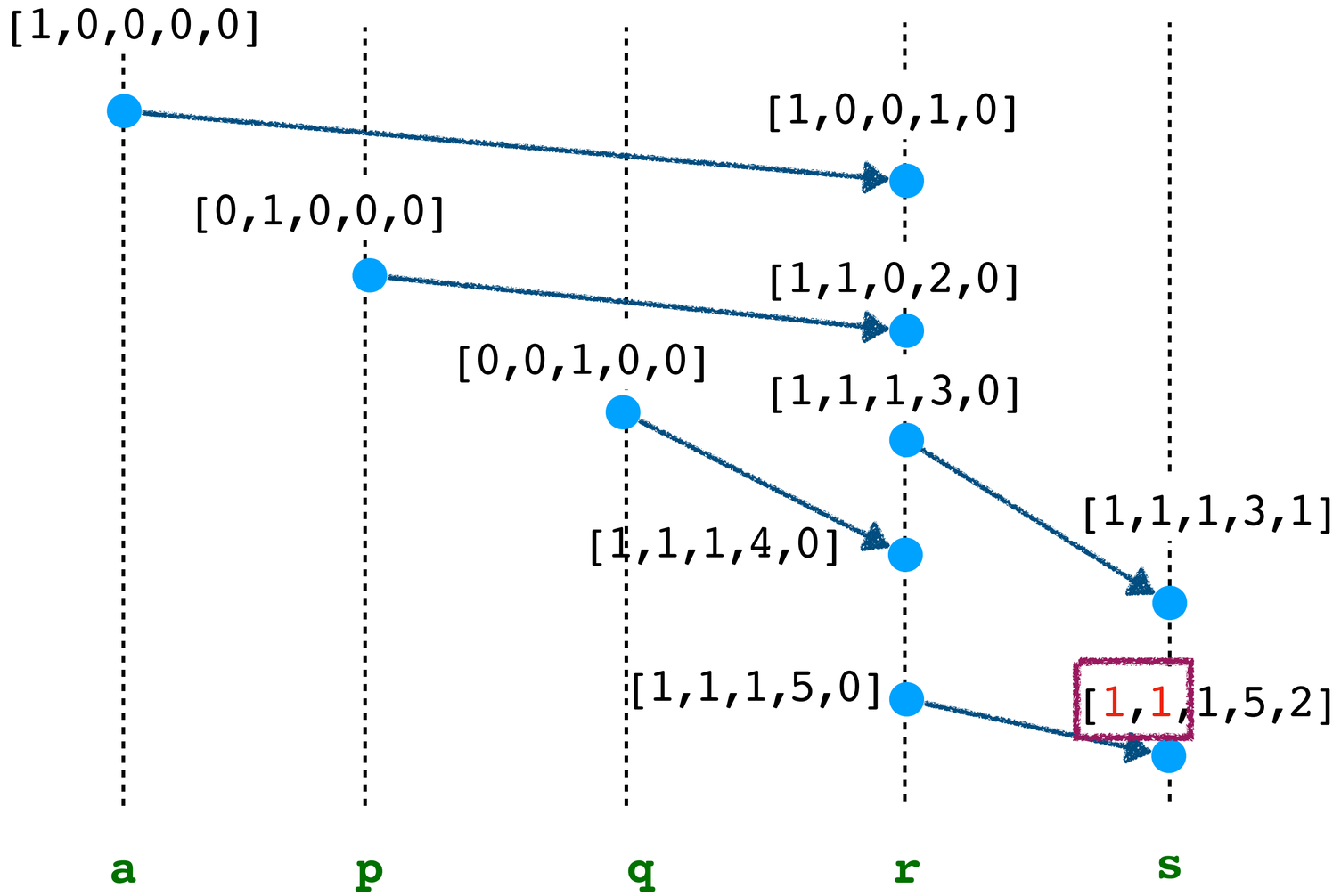


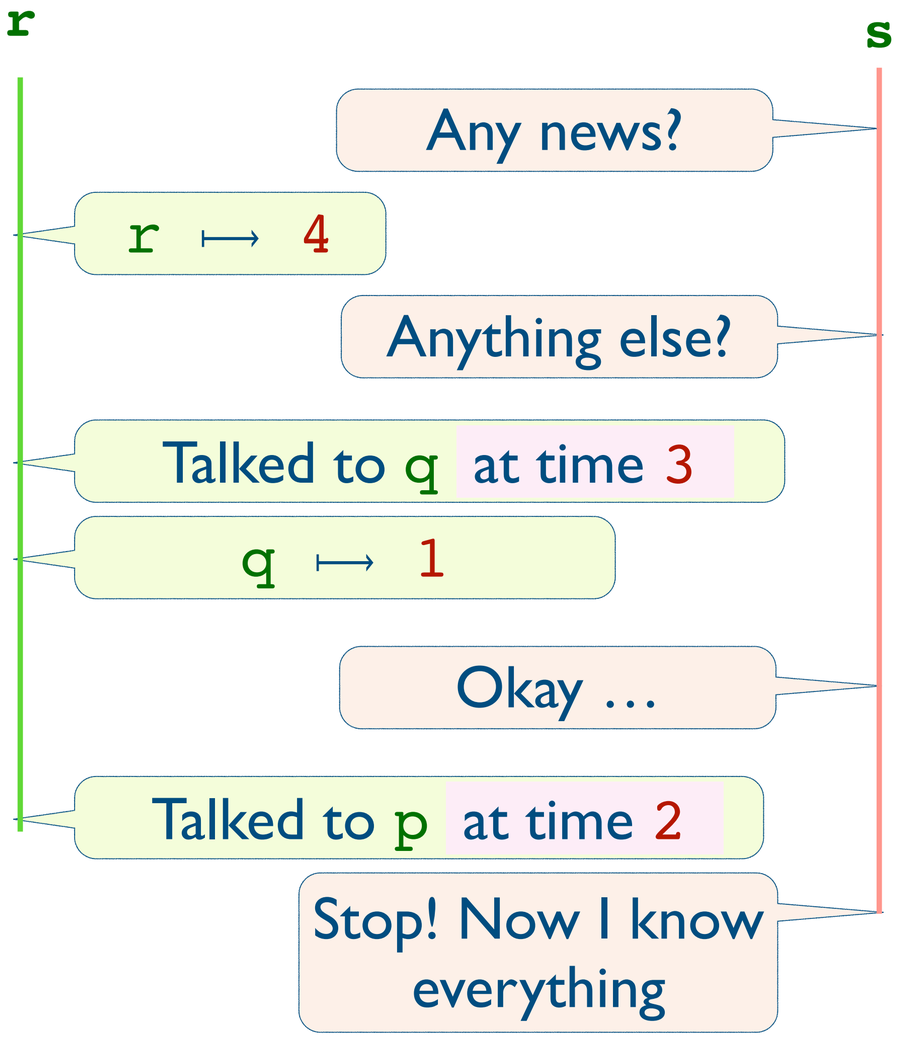
Provenance using **Time of Arrival**



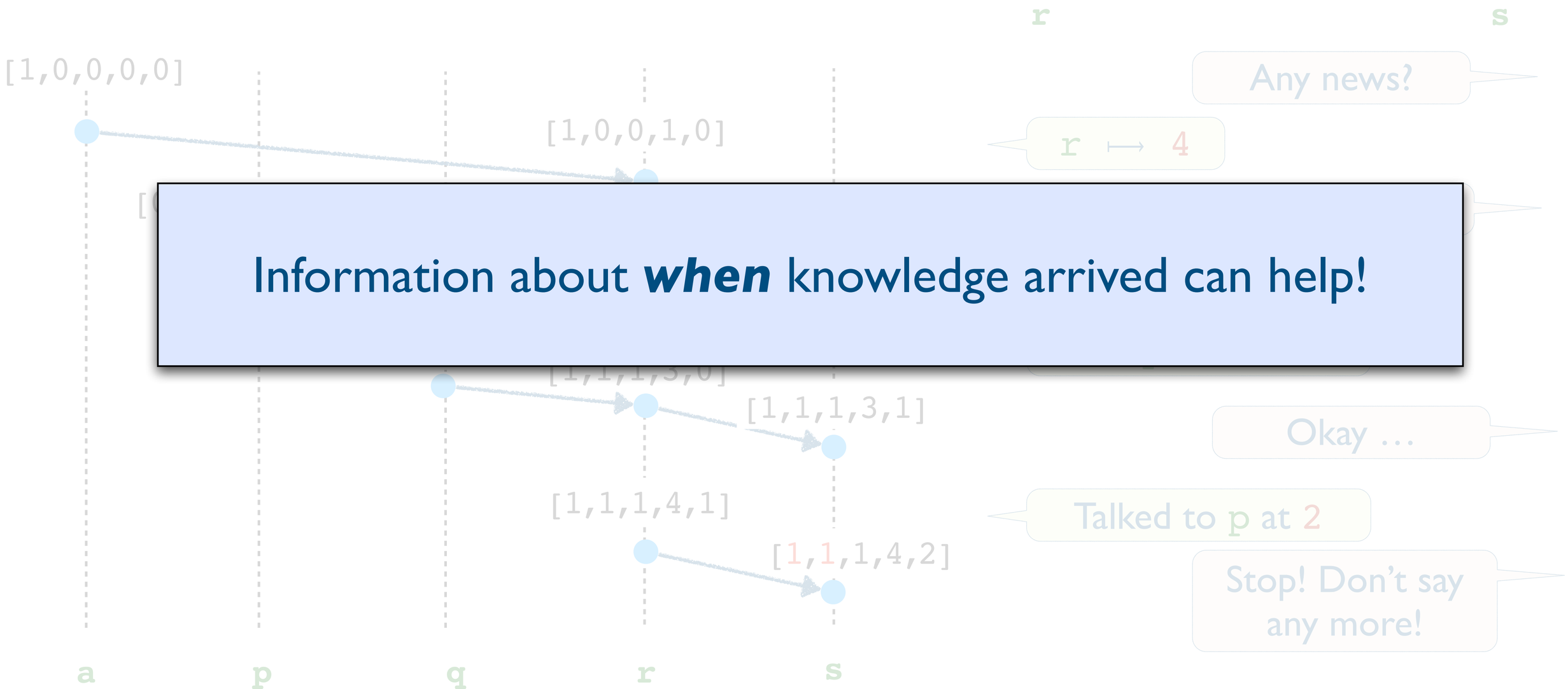


Provenance using **Time of Arrival**

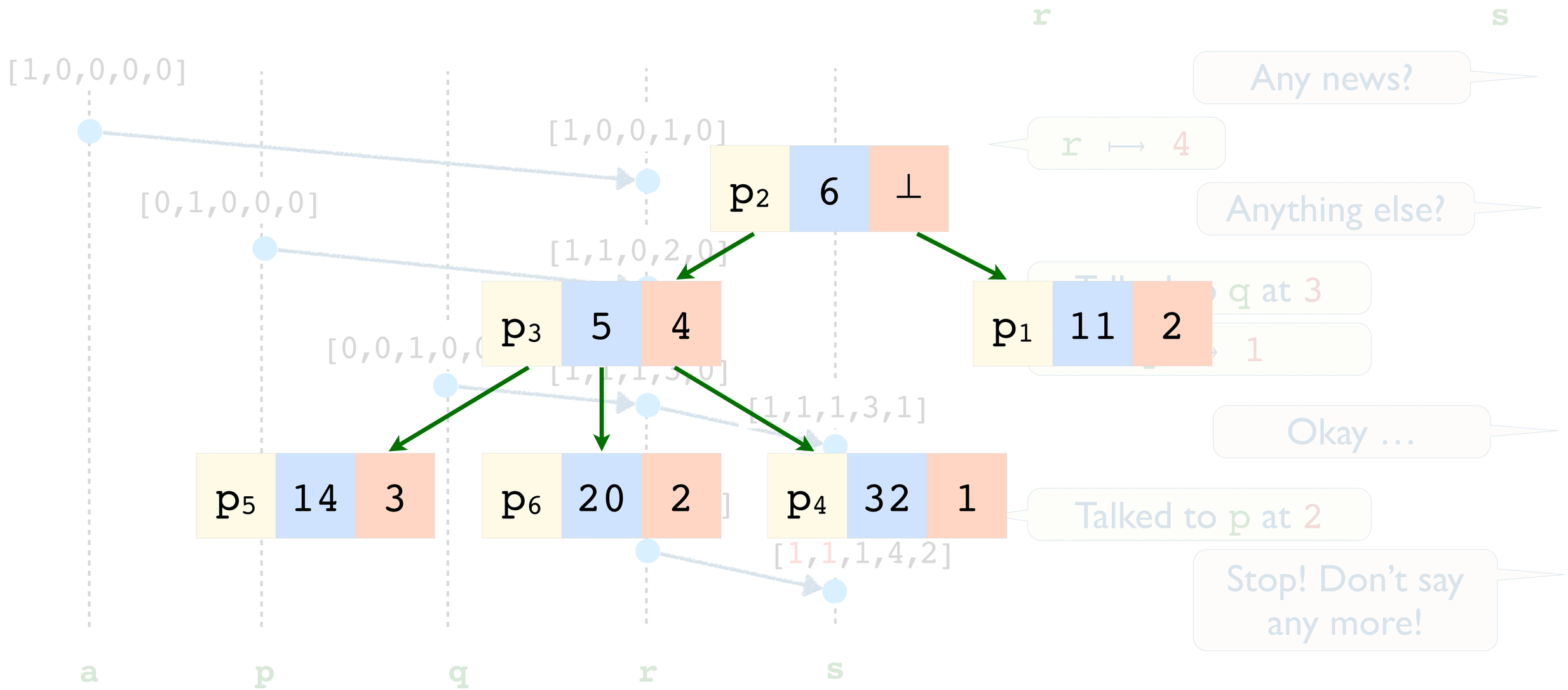




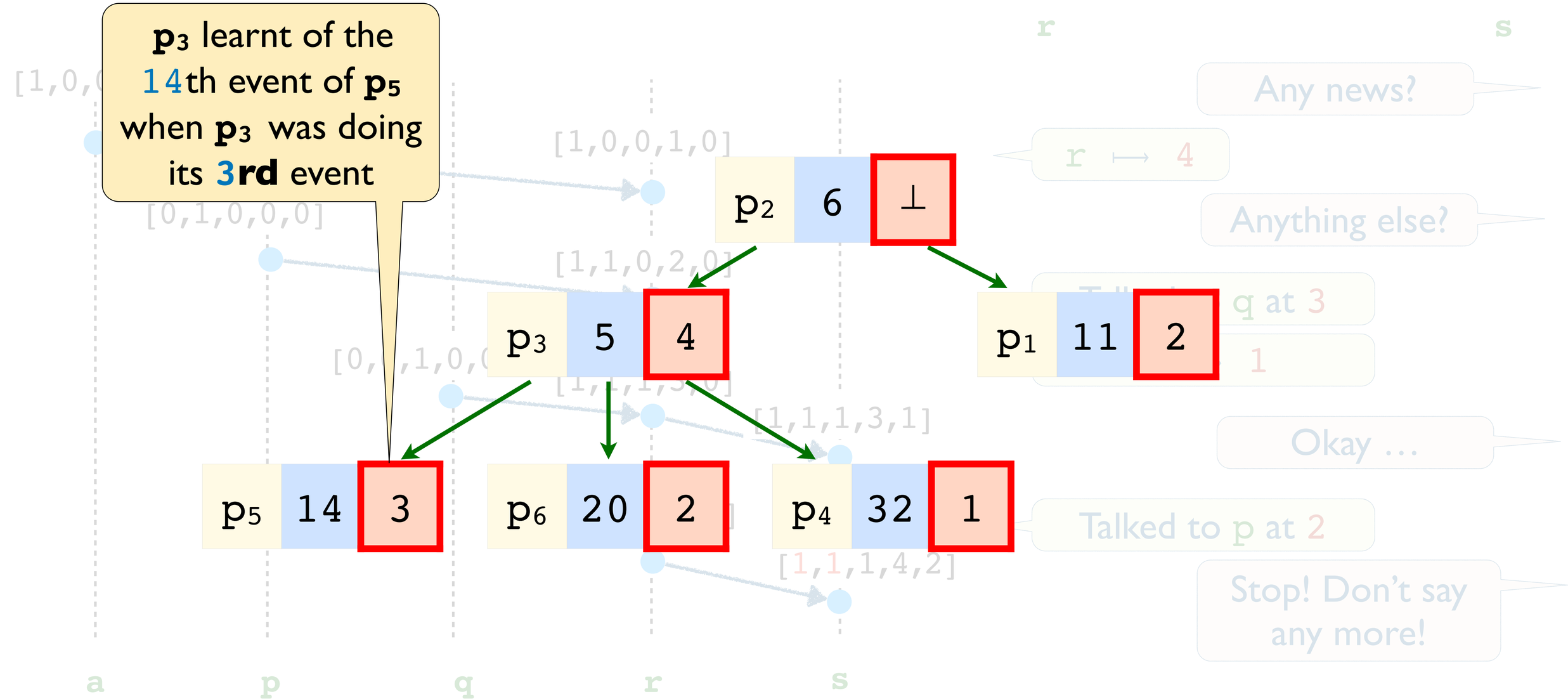
Provenance using **Time of Arrival**



Provenance using **Time of Arrival**



Provenance using **Time of Arrival**



Provenance

How
knowledge arrived

+

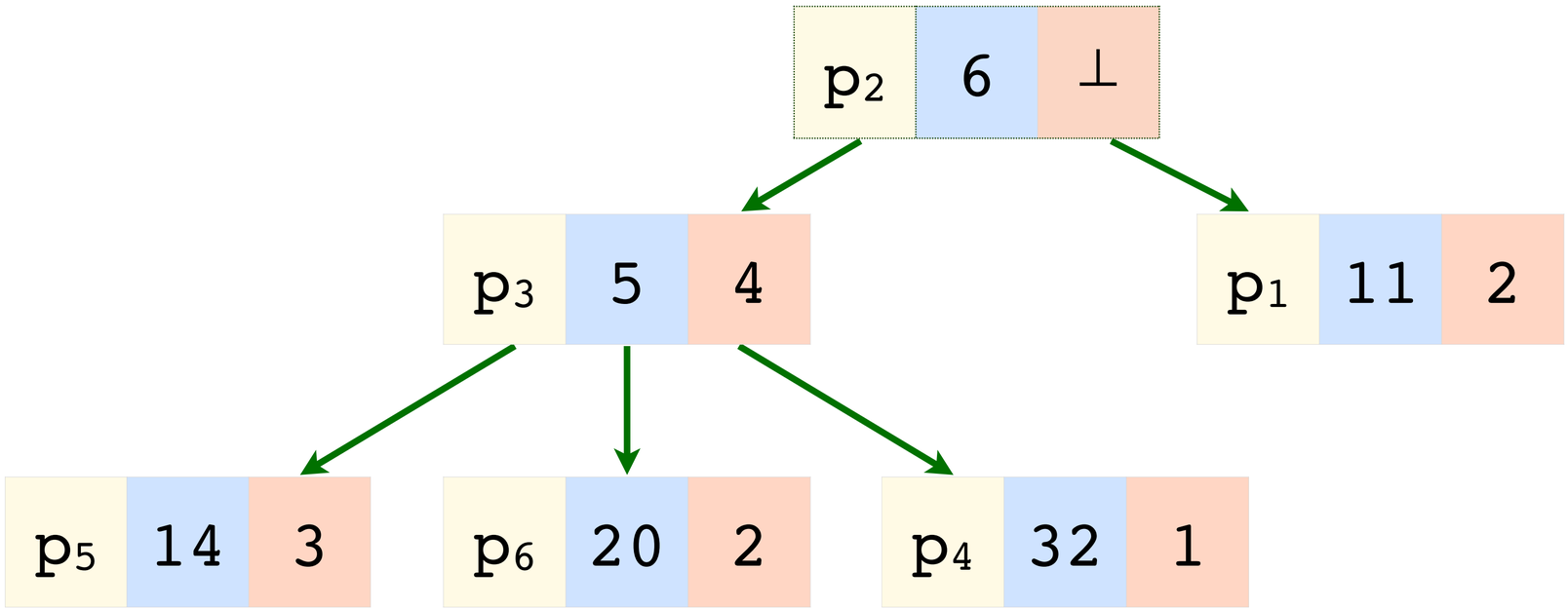
When
knowledge arrived

=

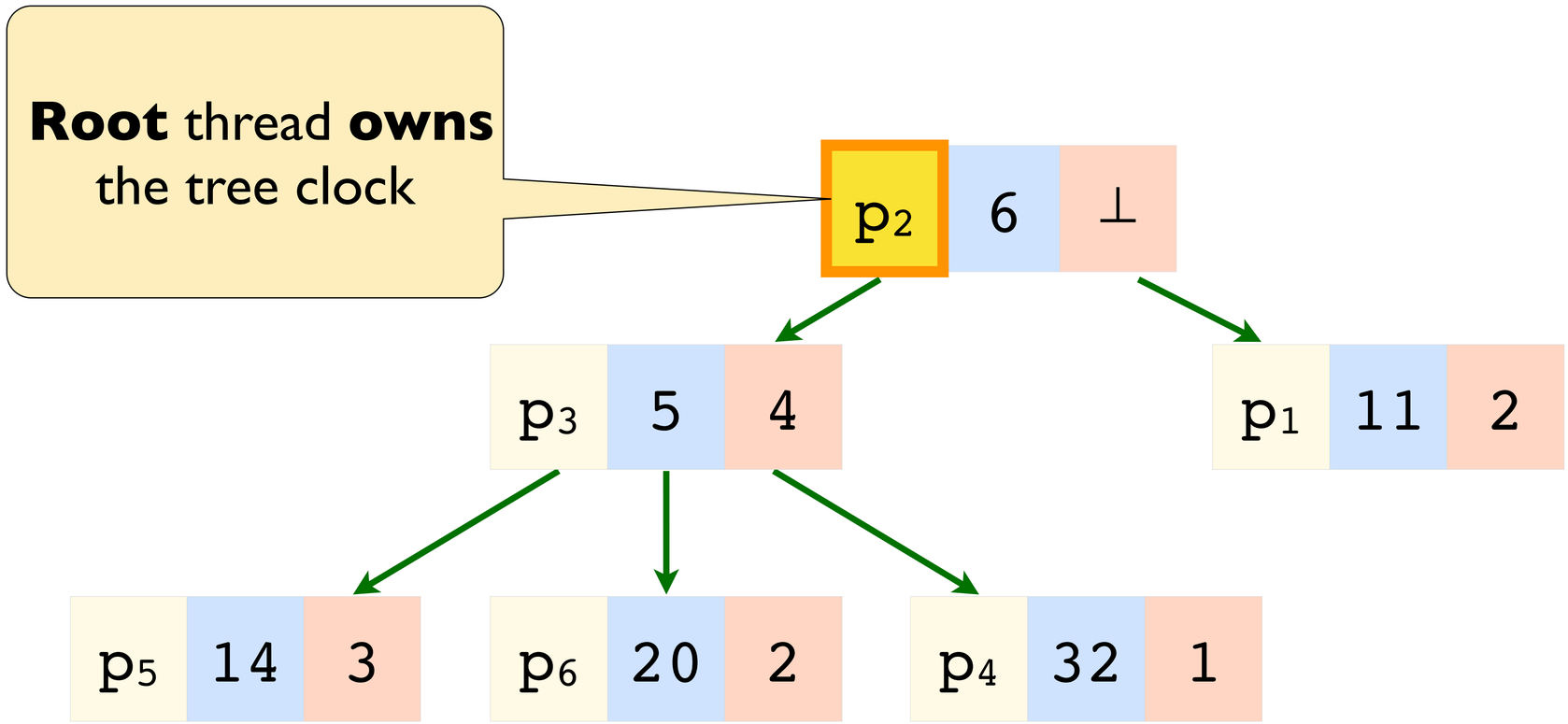
Sub-linear time

Tree Clock

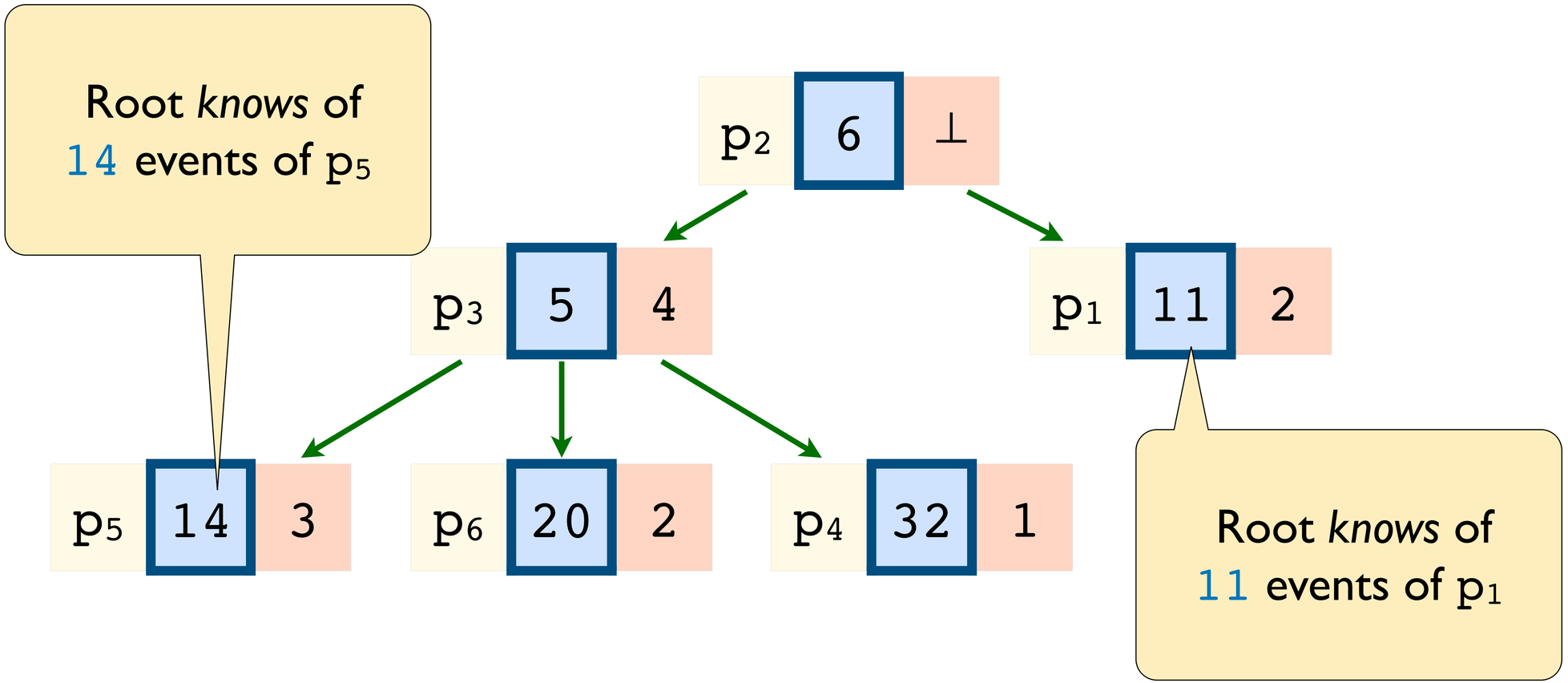
Tree Clock



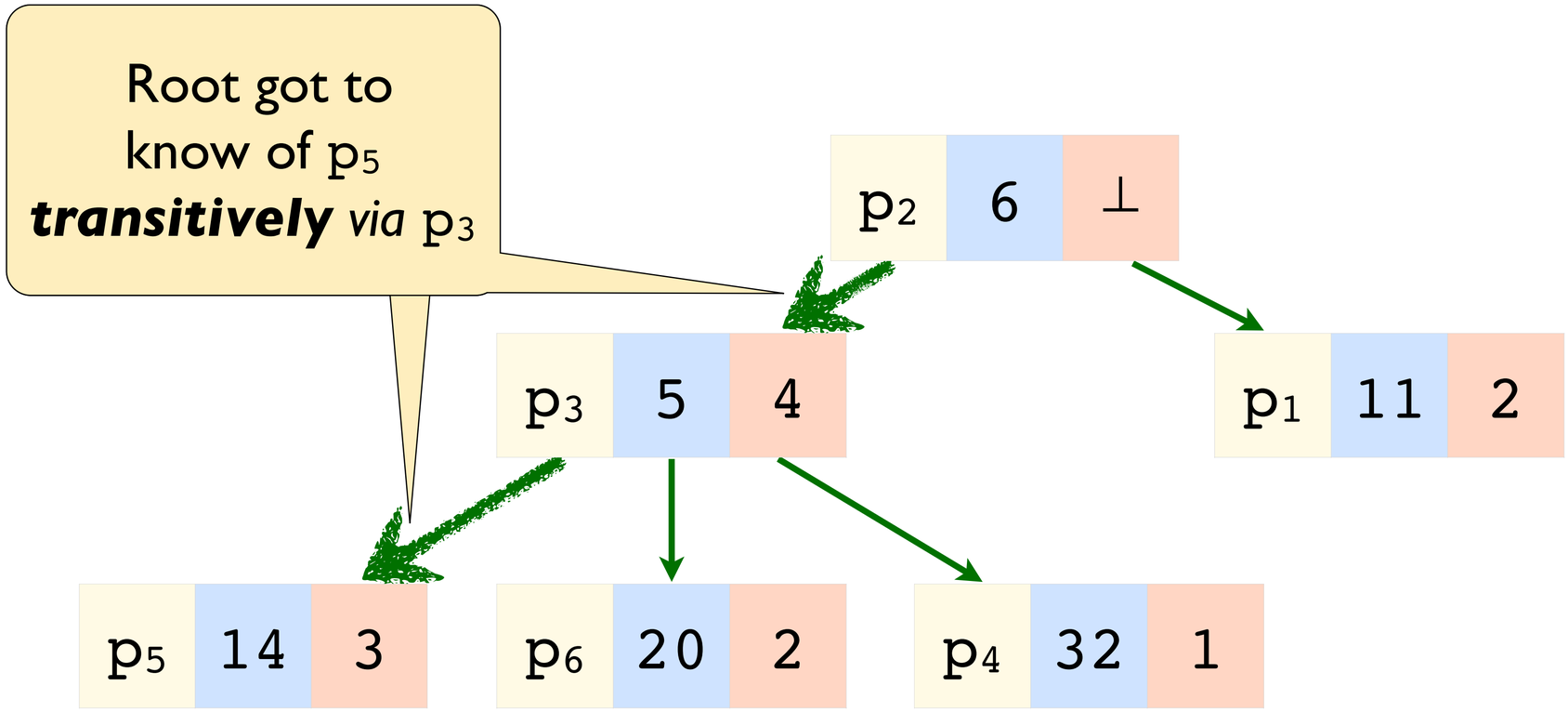
Tree Clock


Root thread **owns** the tree clock
p_2 6 $\perp$
p_3 5 4
p_1 11 2
p_5 14 3
p_6 20 2
p_4 32 1

Tree Clock

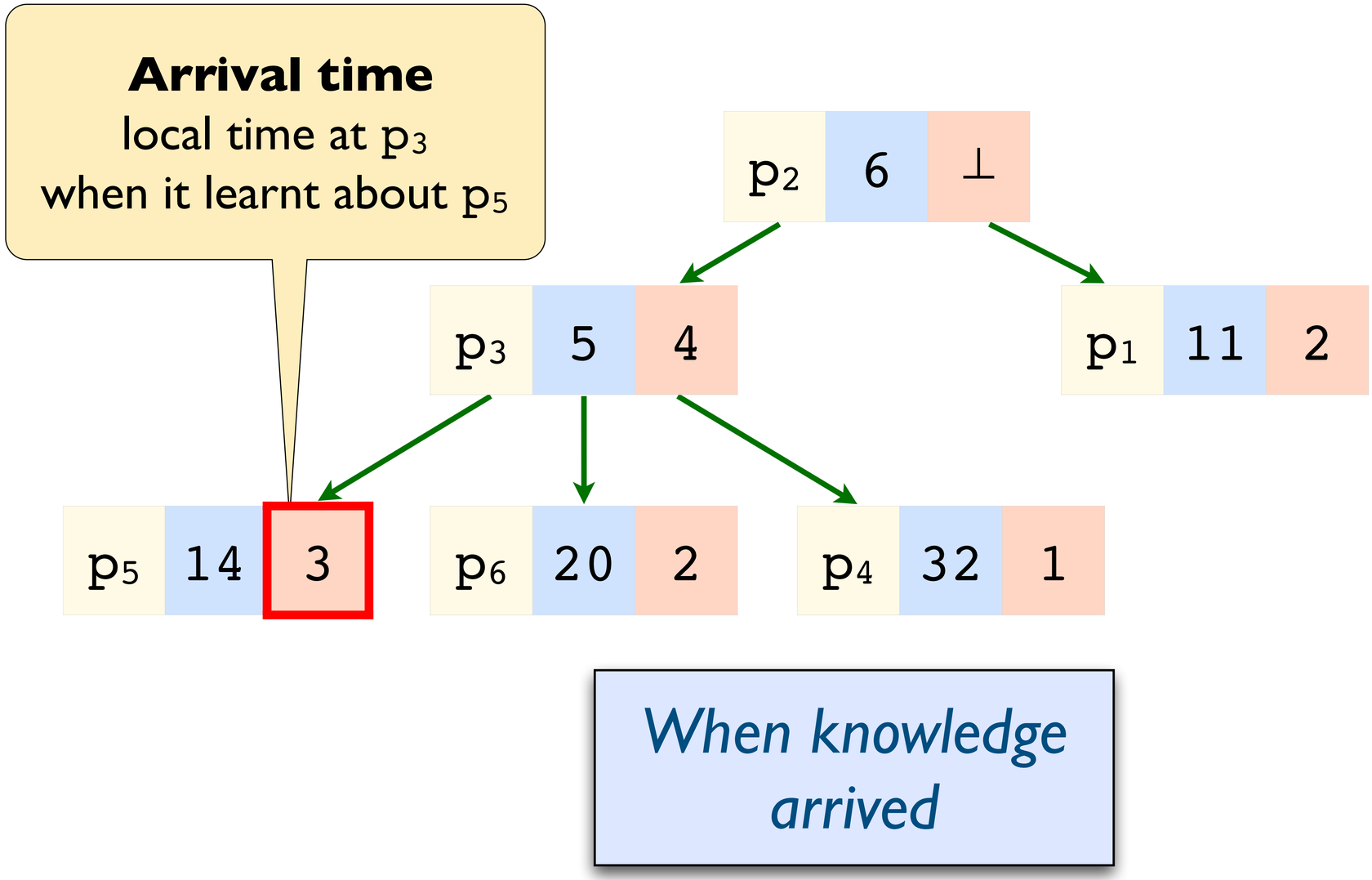

Root *knows* of
14 events of p_5
p_2 6 ⊥
p_3 5 4
p_1 11 2
p_5 14 3
p_6 20 2
p_4 32 1
Root *knows* of
11 events of p_1

Tree Clock

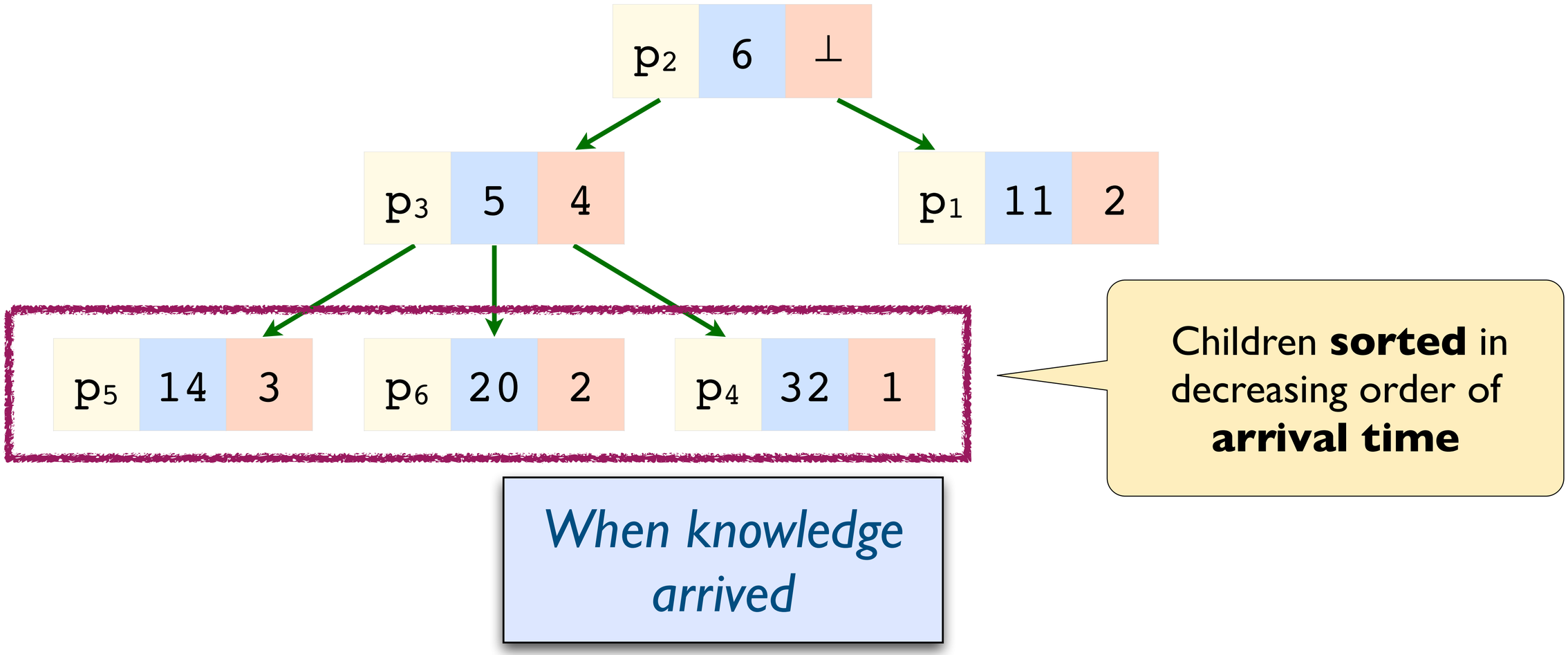


How knowledge arrived

Tree Clock

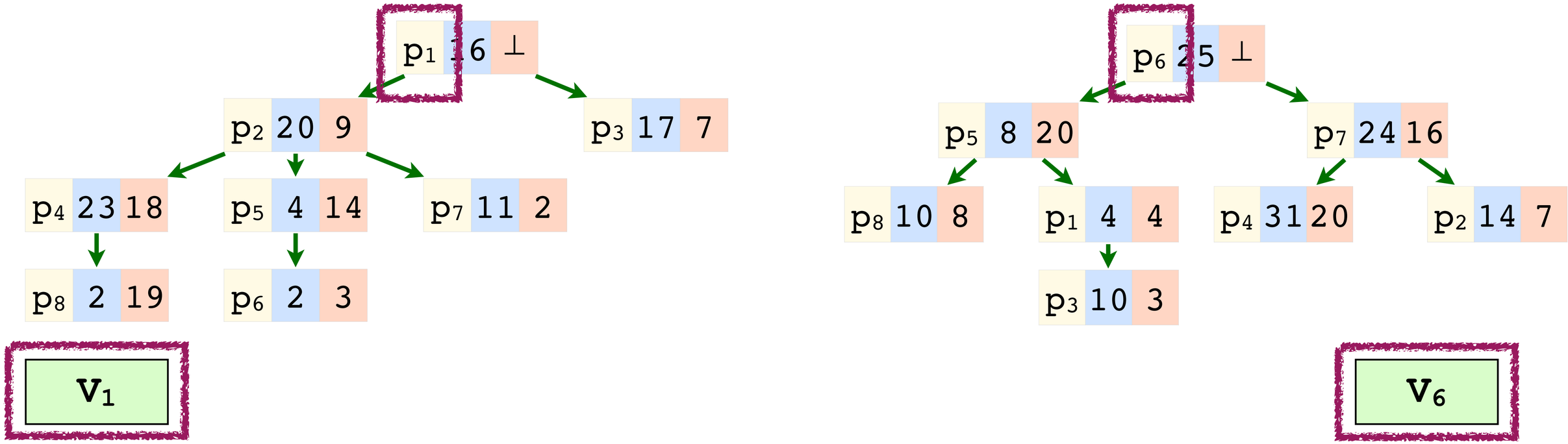

Arrival time
local time at p_3
when it learnt about p_5
p_2 6 ⊥
p_3 5 4
p_1 11 2
p_5 14 3
p_6 20 2
p_4 32 1
When knowledge arrived

Tree Clock

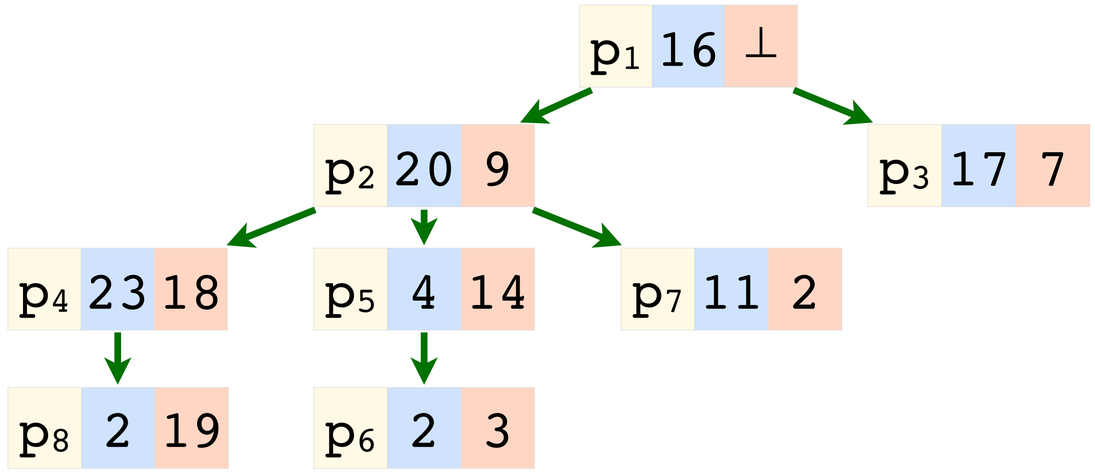

p2 6 ⊥
p3 5 4
p1 11 2
p5 14 3
p6 20 2
p4 32 1
Children **sorted** in decreasing order of **arrival time**
When knowledge arrived

Tree Clock Join

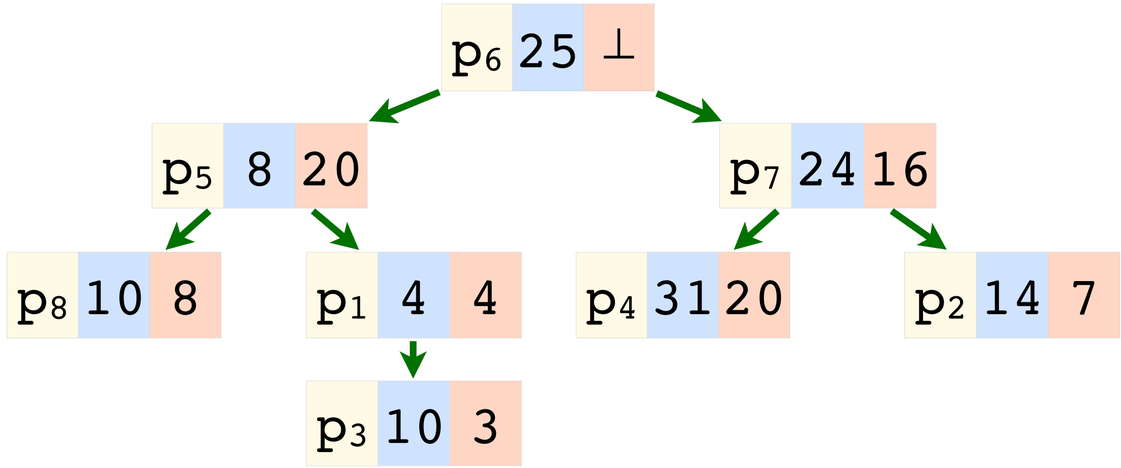
Tree Clock Join



Tree Clock Join



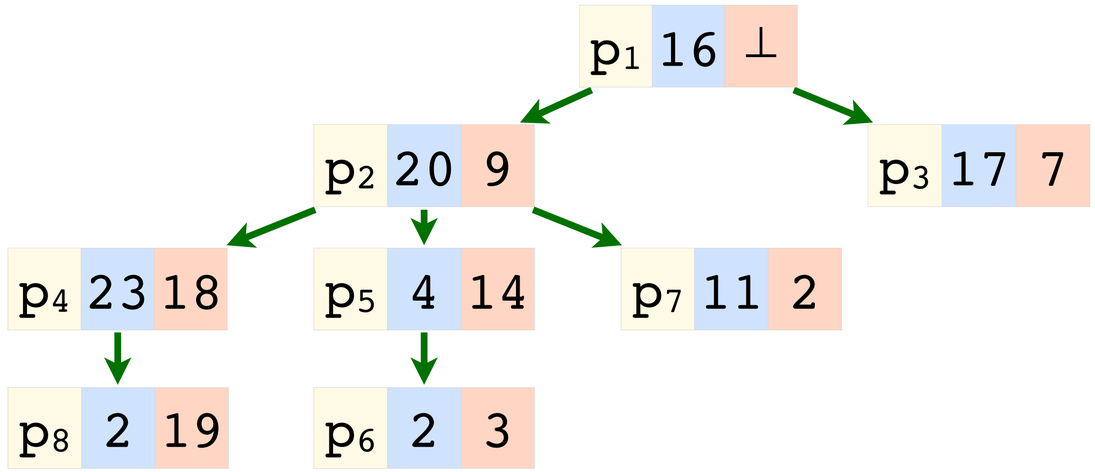
$\mathbf{V_1}$

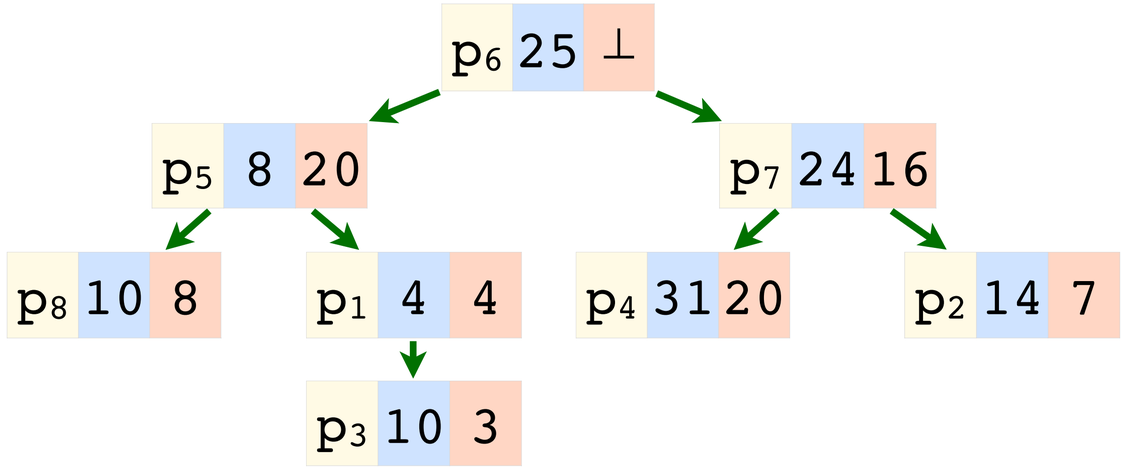


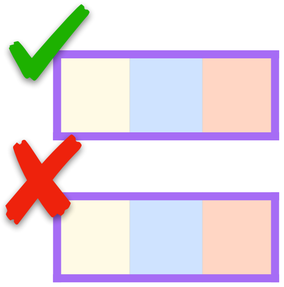
$\mathbf{V_6}$

$\mathbf{V_6}$.JoinWith($\mathbf{V_1}$)

Tree Clock Join



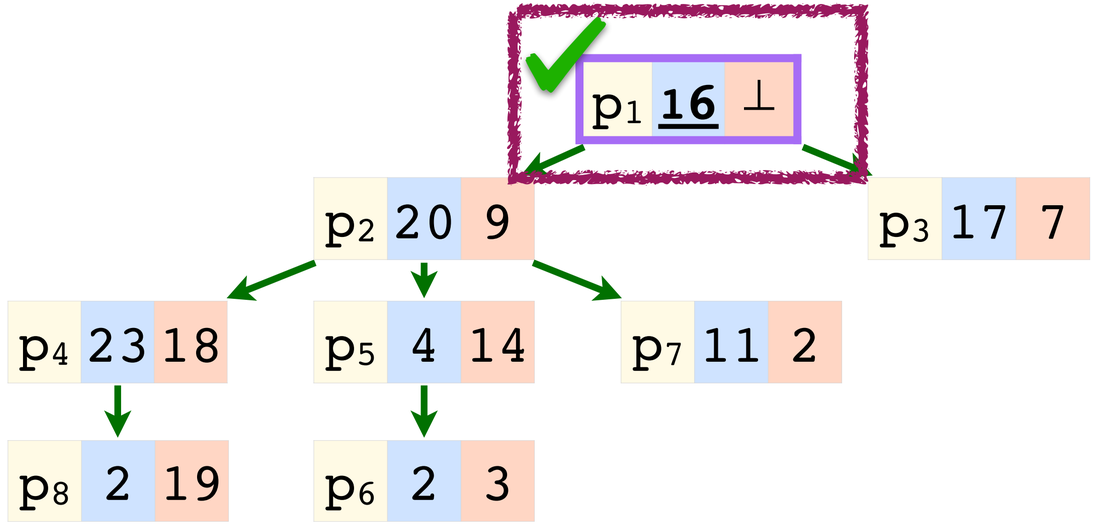


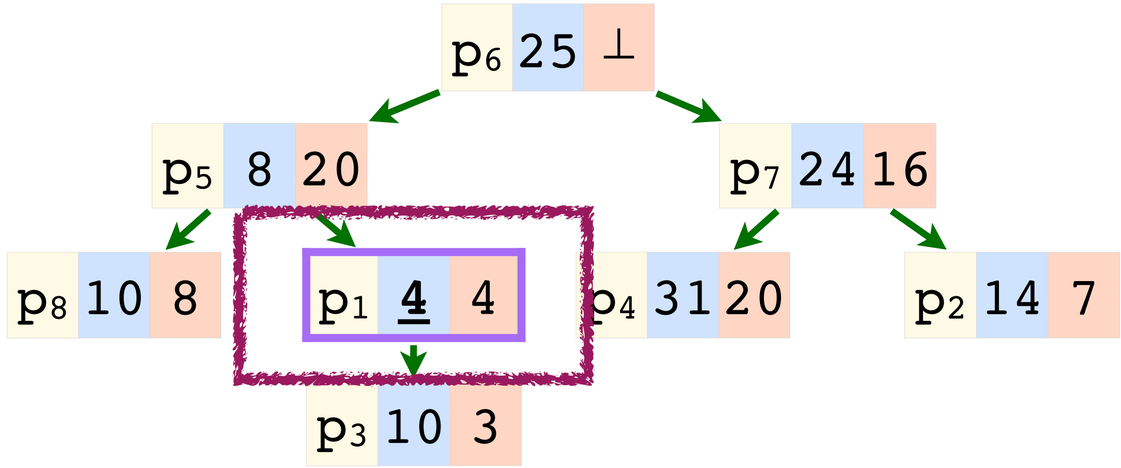


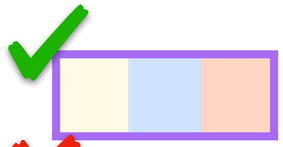
Traversed and Updated

Only Traversed

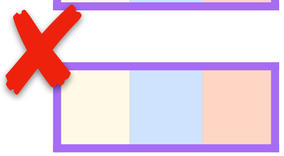
Tree Clock Join





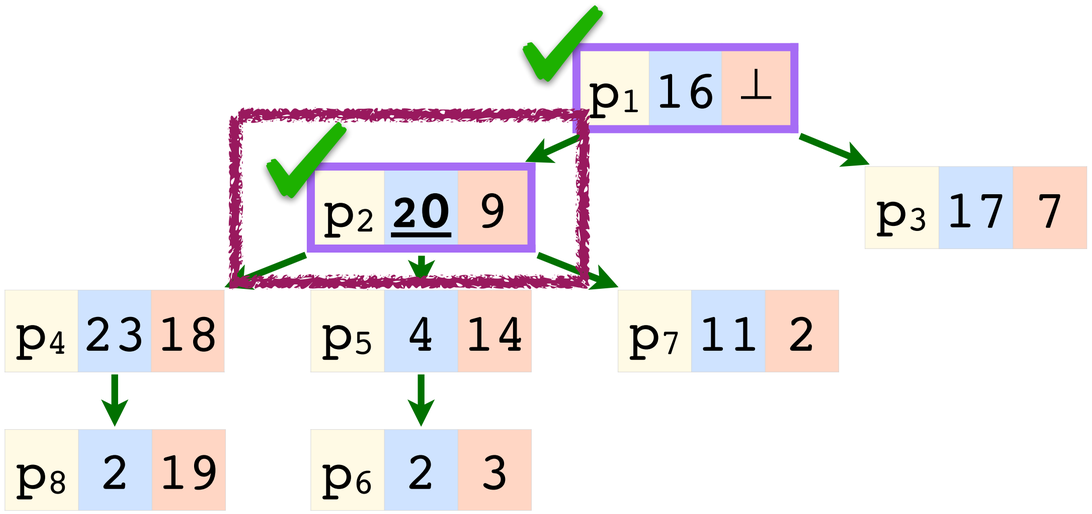


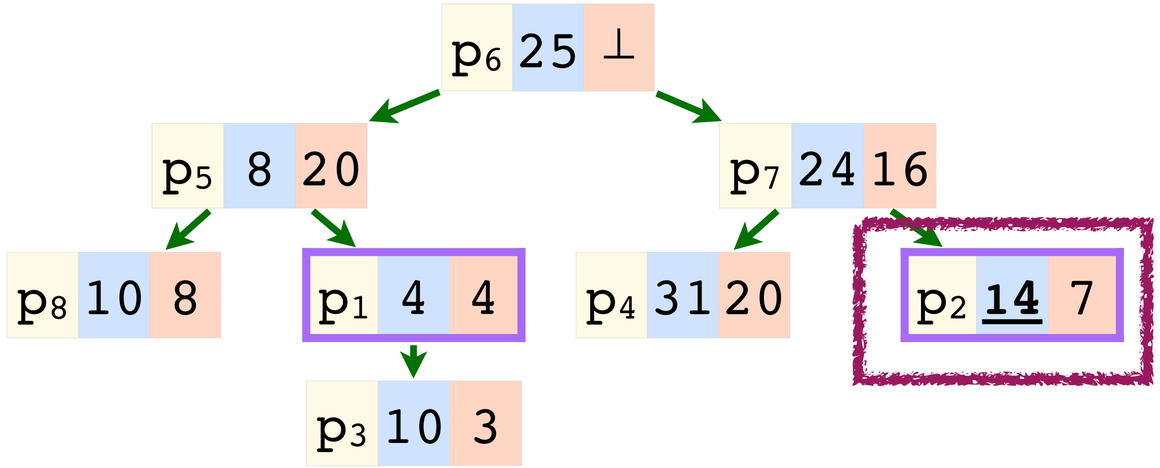
Accessed and Updated

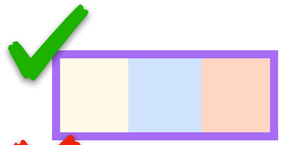


Only Accessed

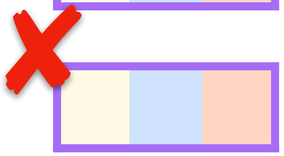
Tree Clock Join





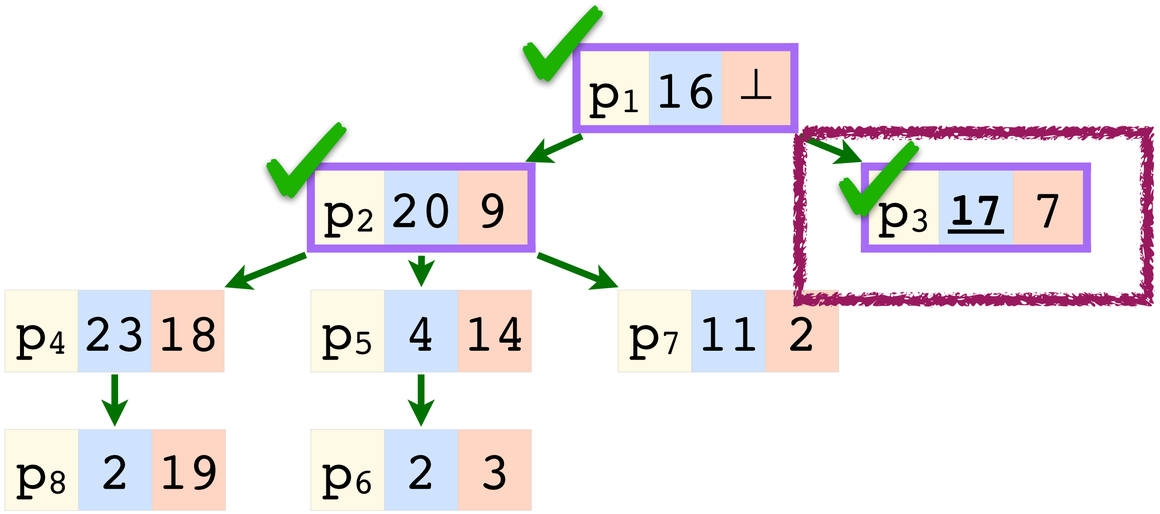


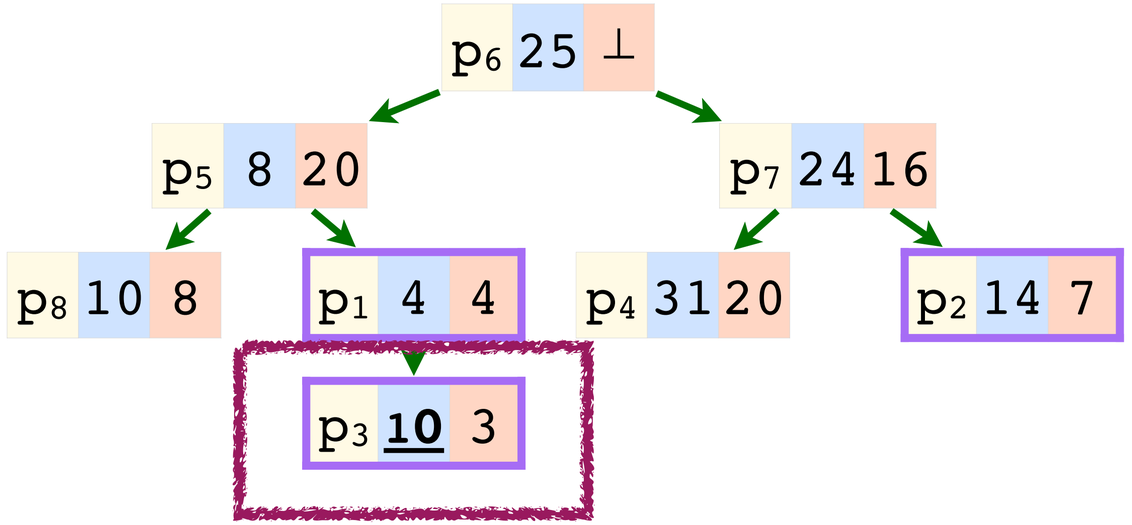
Accessed and Updated

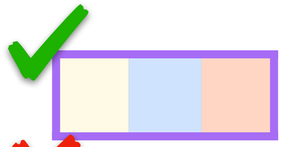


Only Accessed

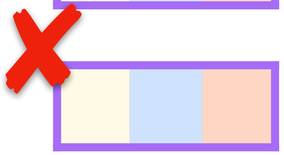
Tree Clock Join





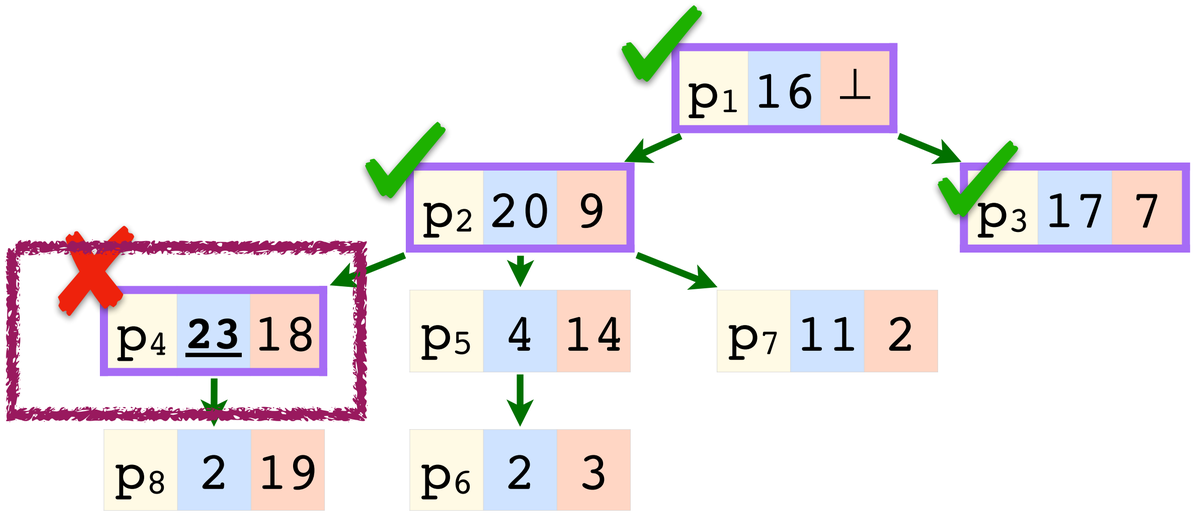


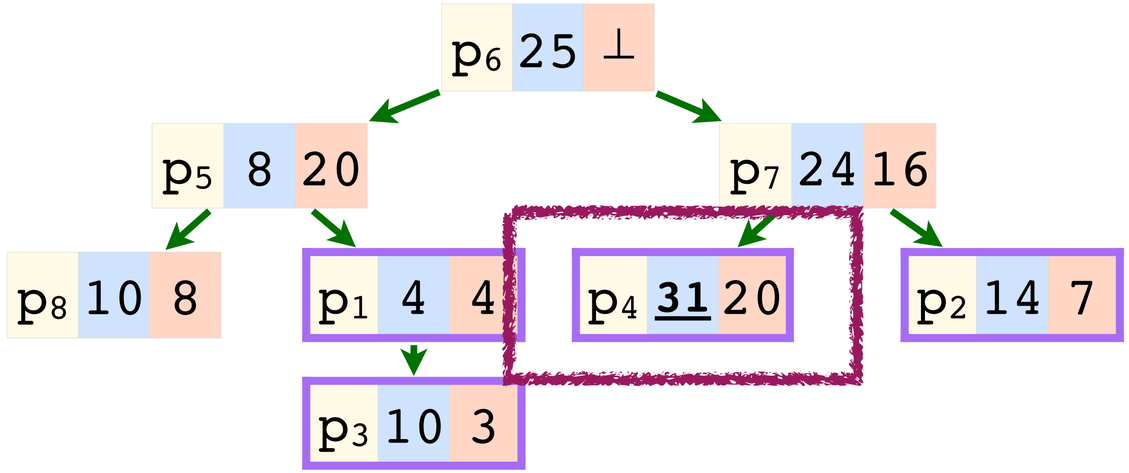
Accessed and Updated

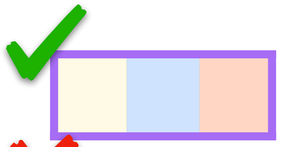


Only Accessed

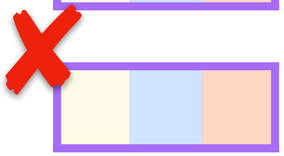
Tree Clock Join





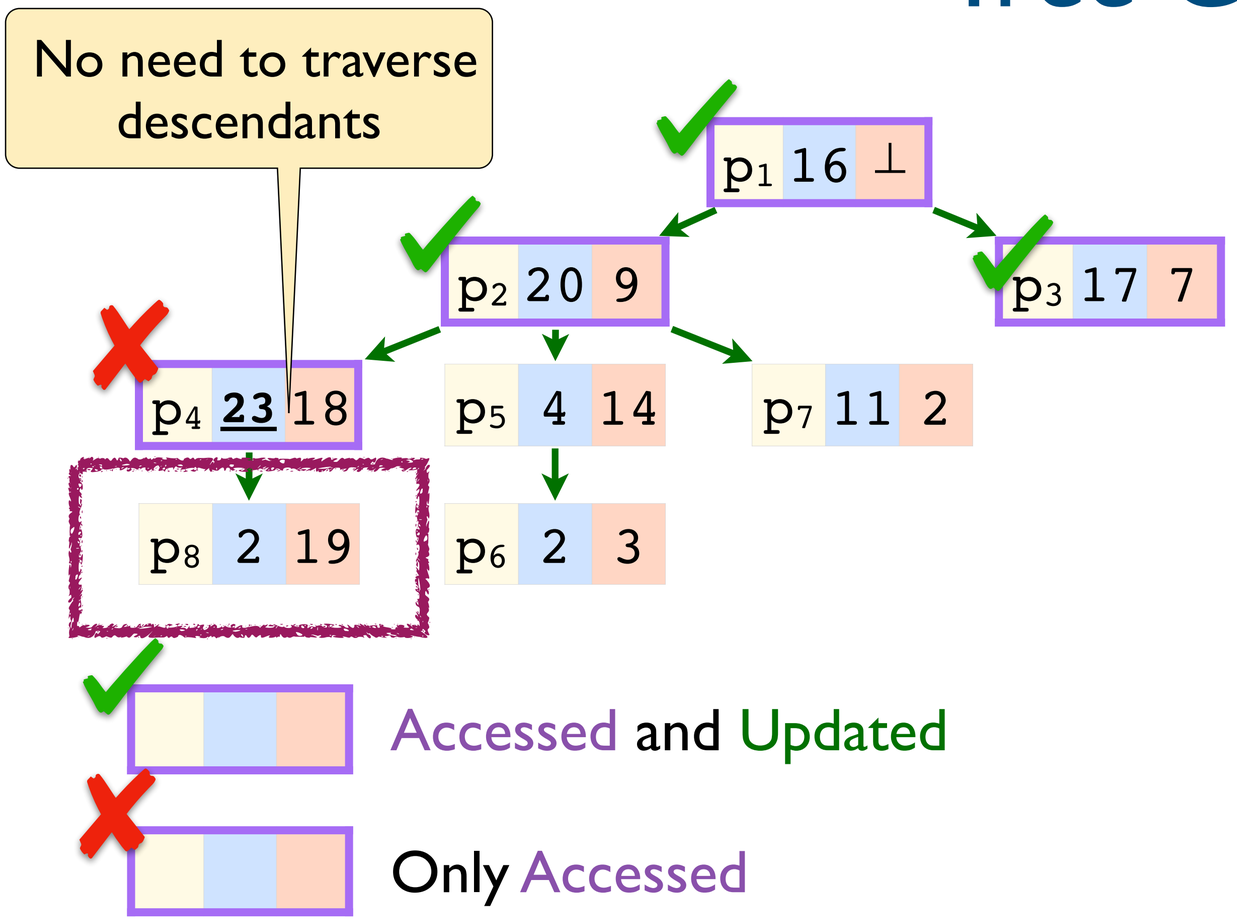


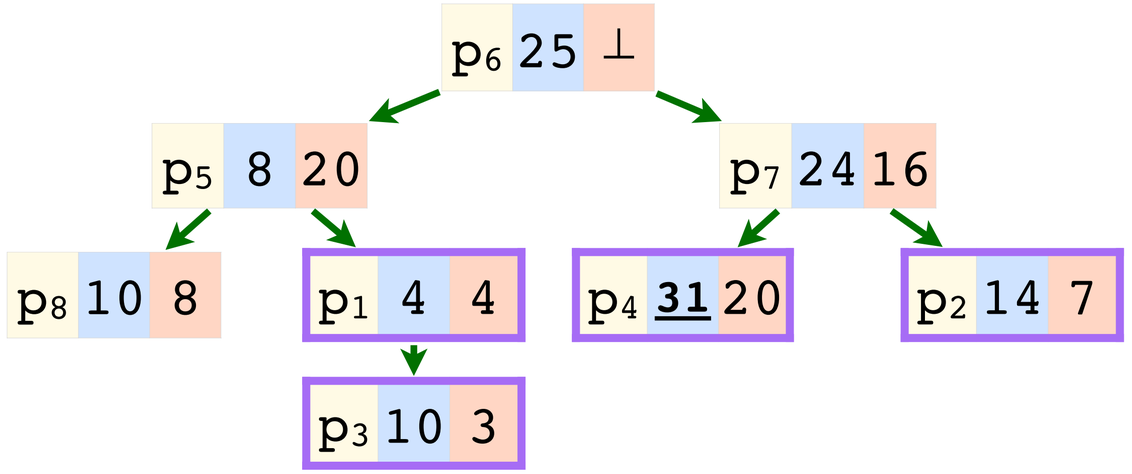
Accessed and Updated



Only Accessed

Tree Clock Join

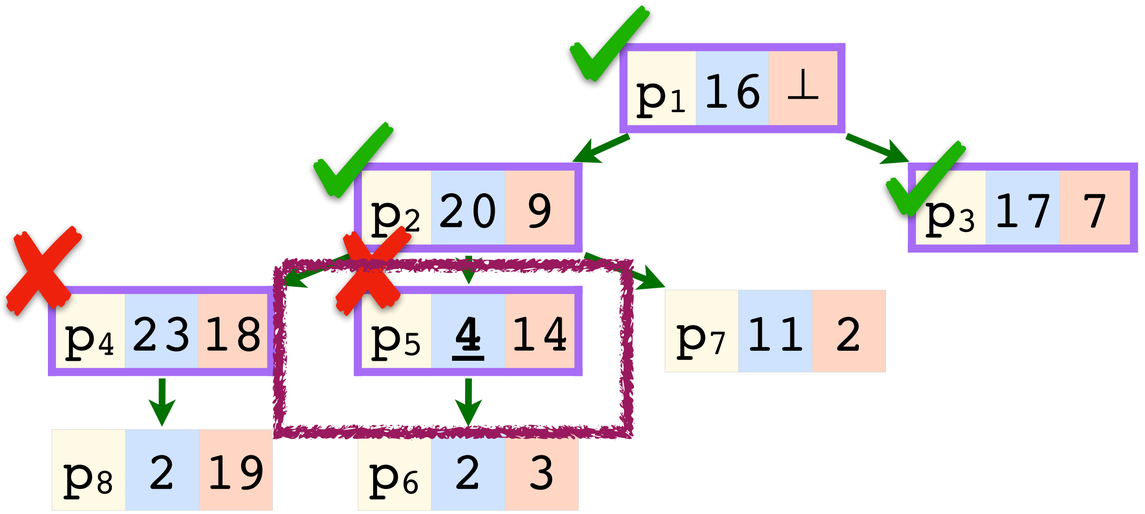


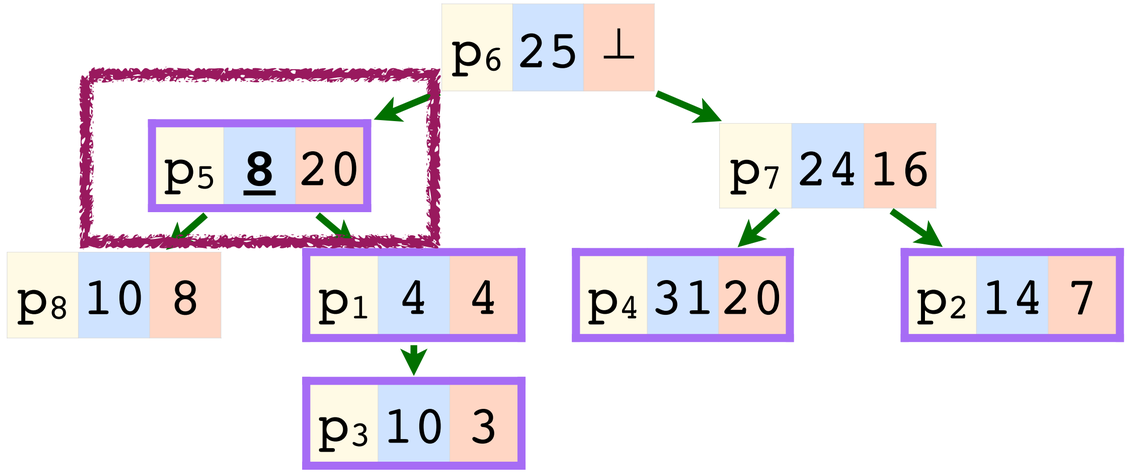


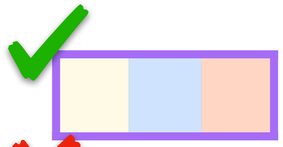
Accessed and Updated

Only Accessed

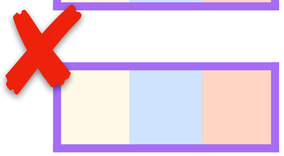
Tree Clock Join





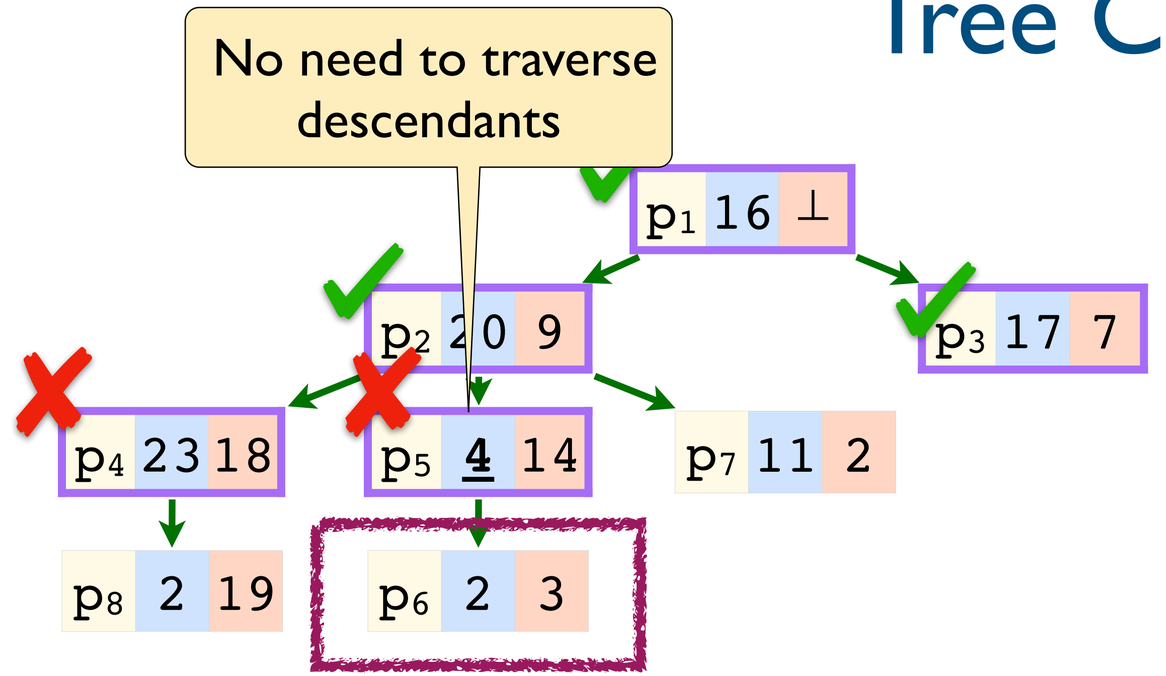


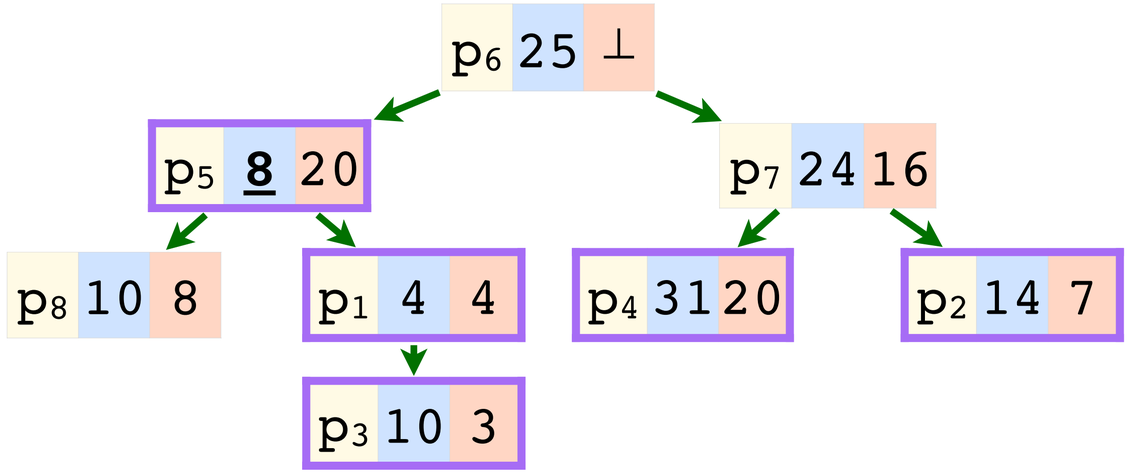
Accessed and Updated

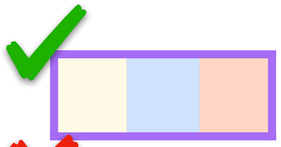


Only Accessed

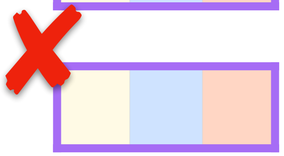
Tree Clock Join





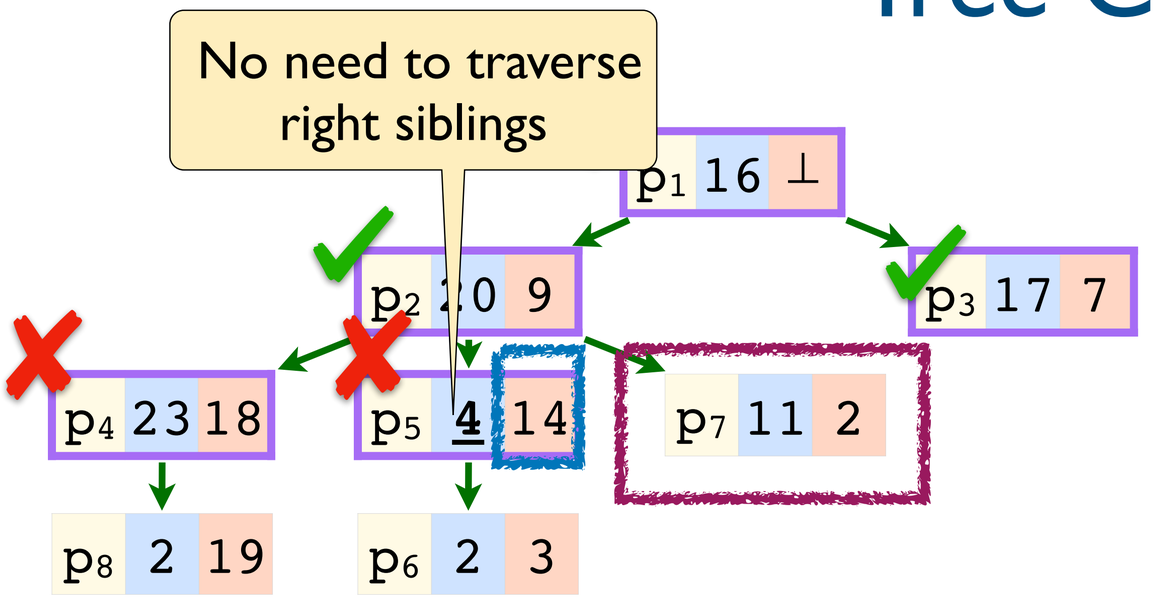


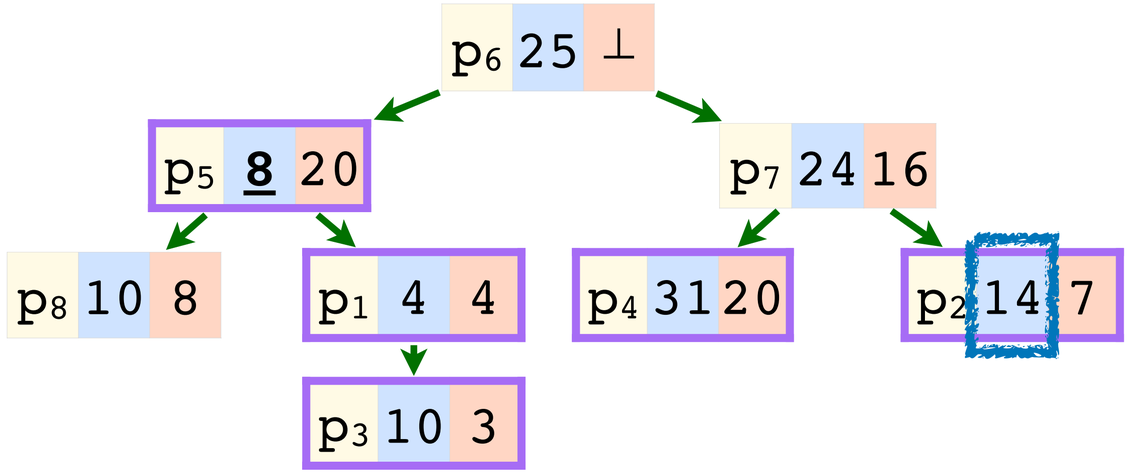
Accessed and Updated

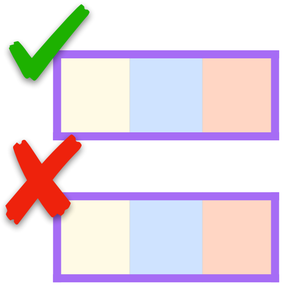


Only Accessed

Tree Clock Join



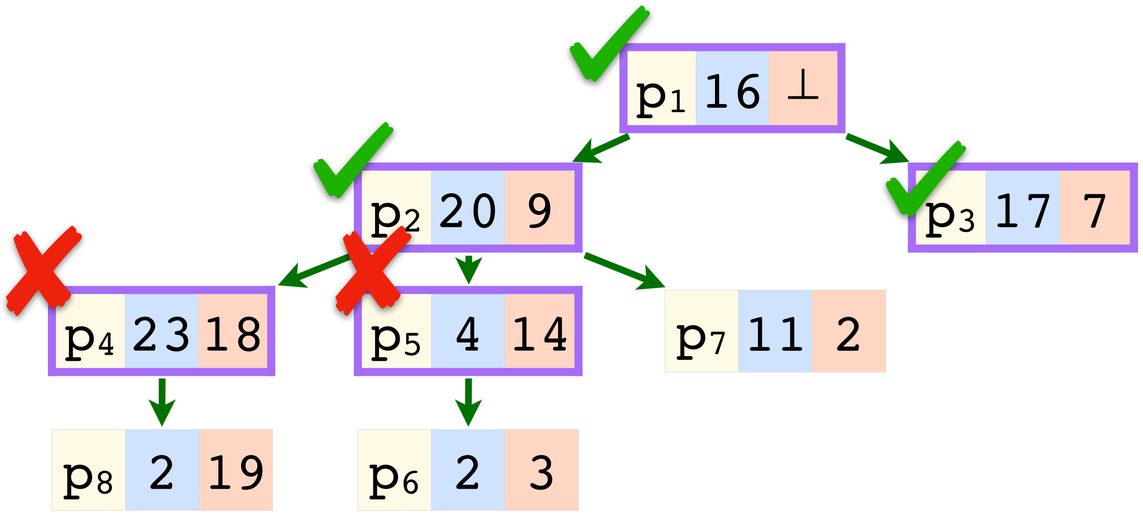


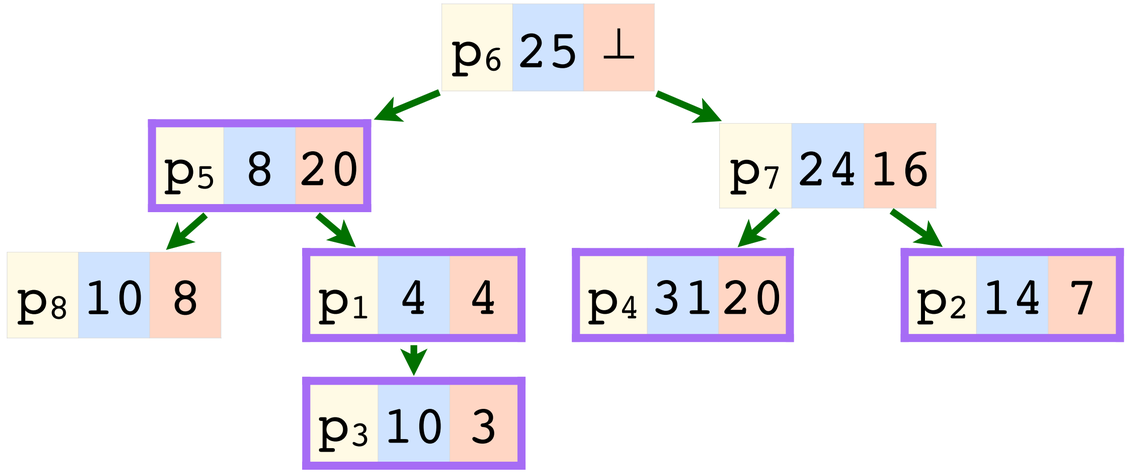


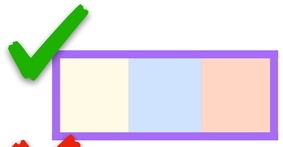
Accessed and Updated

Only Accessed

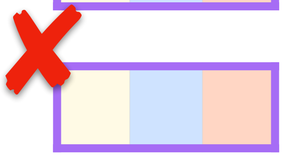
Tree Clock Join



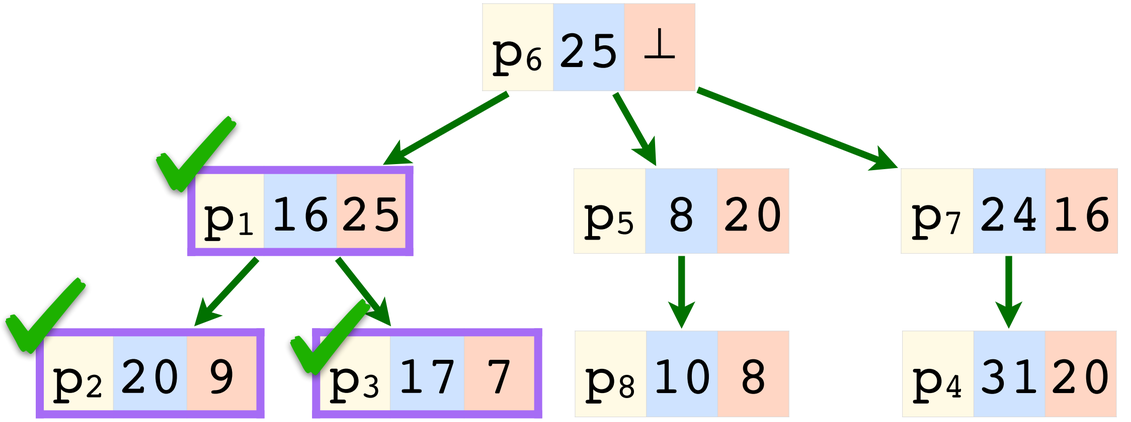




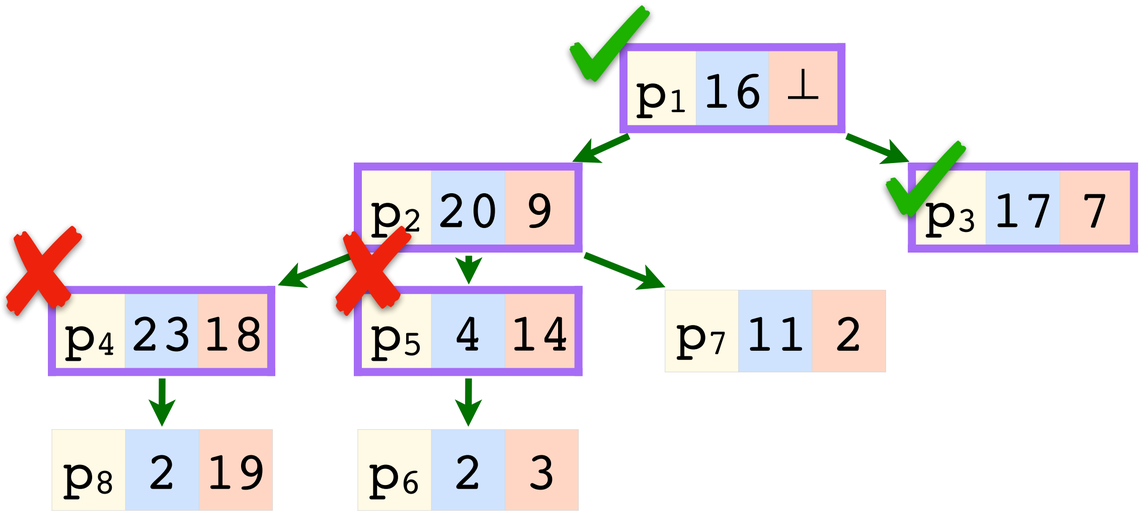
Accessed and Updated

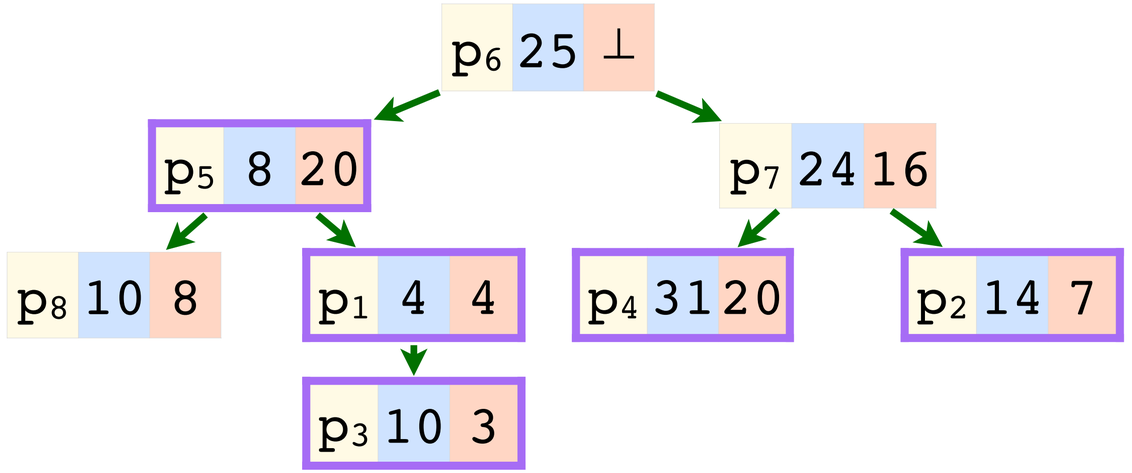


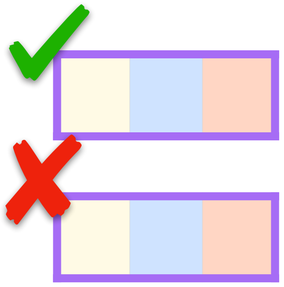
Only Accessed



Tree Clock Join



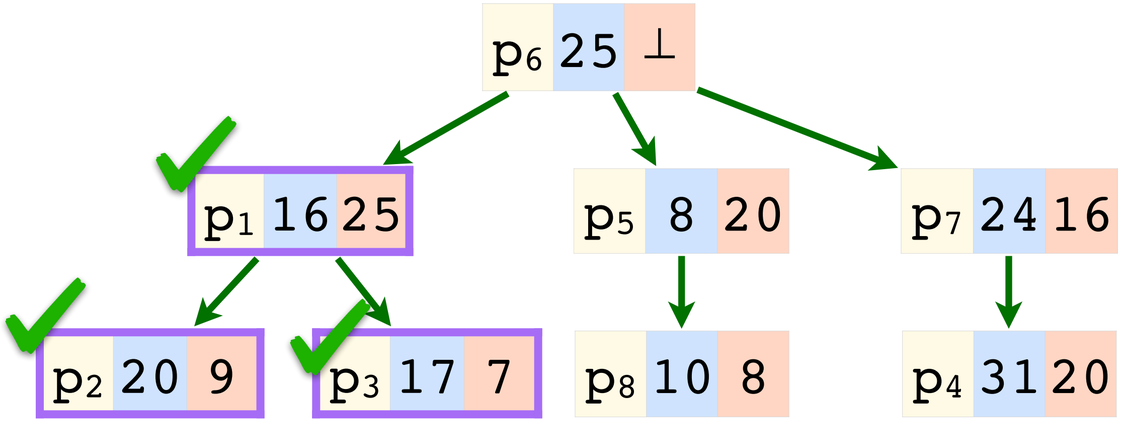




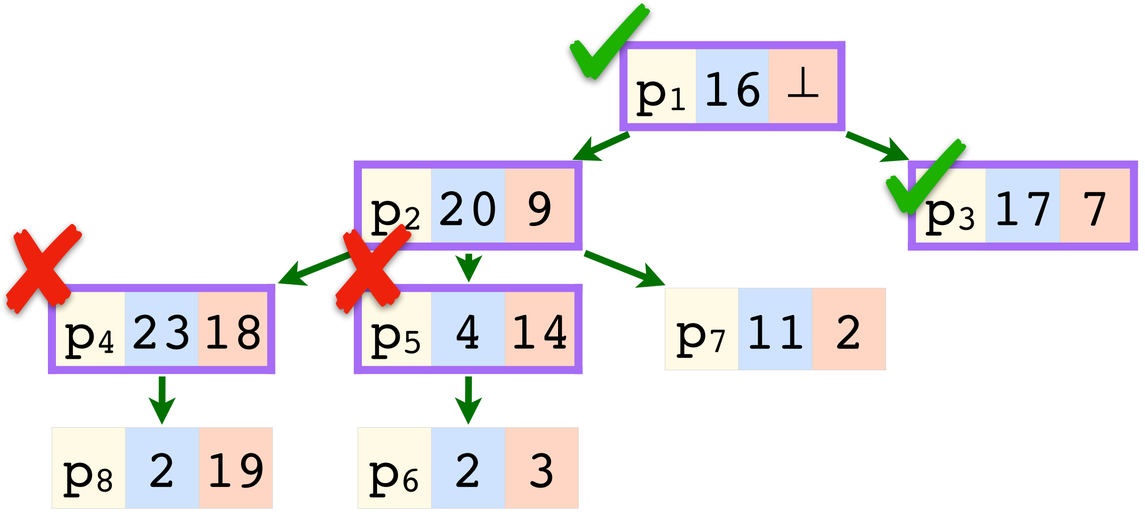
Accessed and Updated

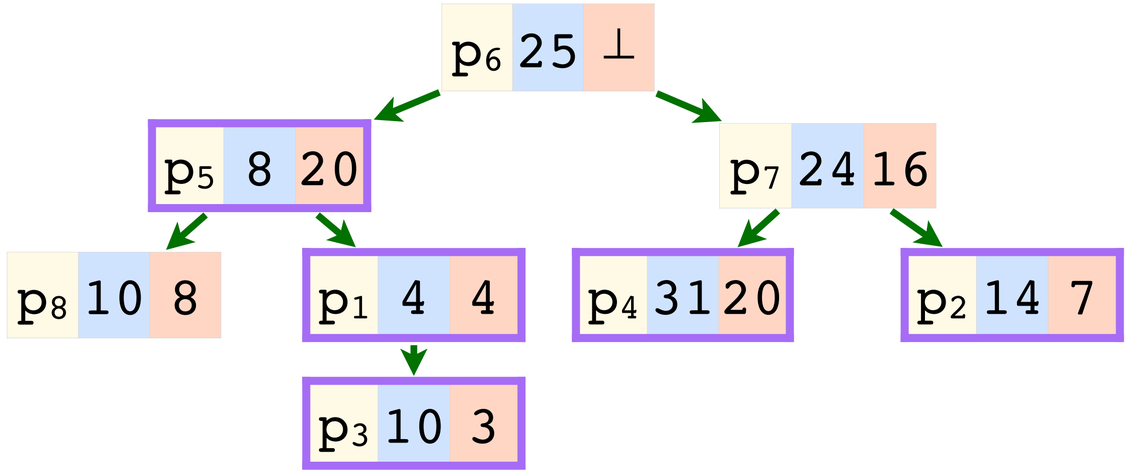
Only Accessed

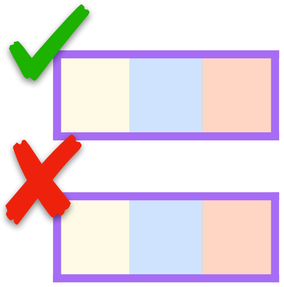
16	20	17	23	4	2	11	2
p1	p2	p3	p4	p5	p6	p7	p8



Tree Clock Join

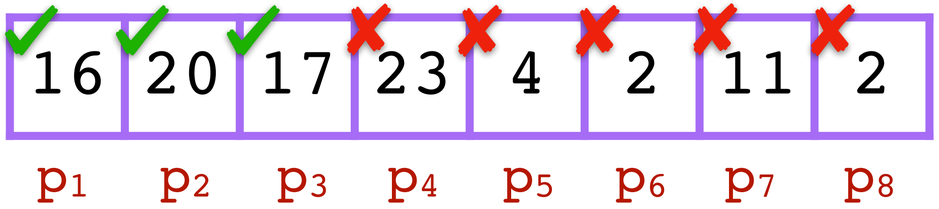


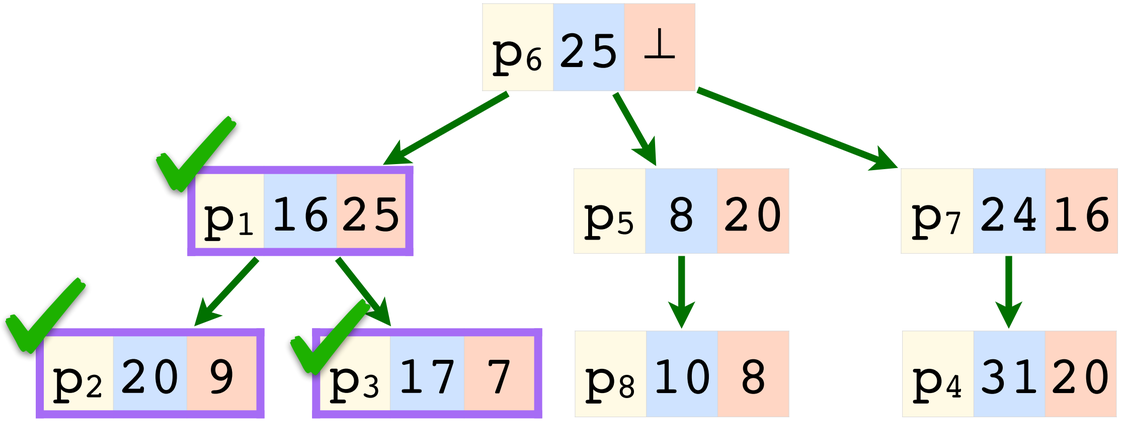




Accessed and Updated

Only Accessed





Tree Clock Copy?

Tree Clock Copy?

In general, tree clock copy will take O(|proceses|) time

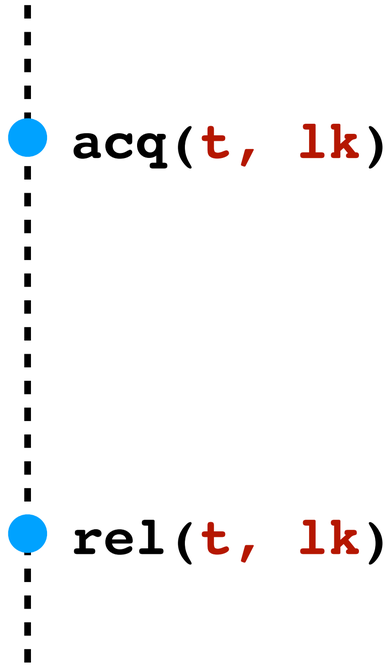
Tree Clock Copy?

In general, tree clock copy will take O(|proceses|) time

Race detection

```
procedure acq(t, lk)
    Vt.JoinWith(Vlk)

procedure rel(t, lk)
    Vlk.CopyFrom(Vt)
```



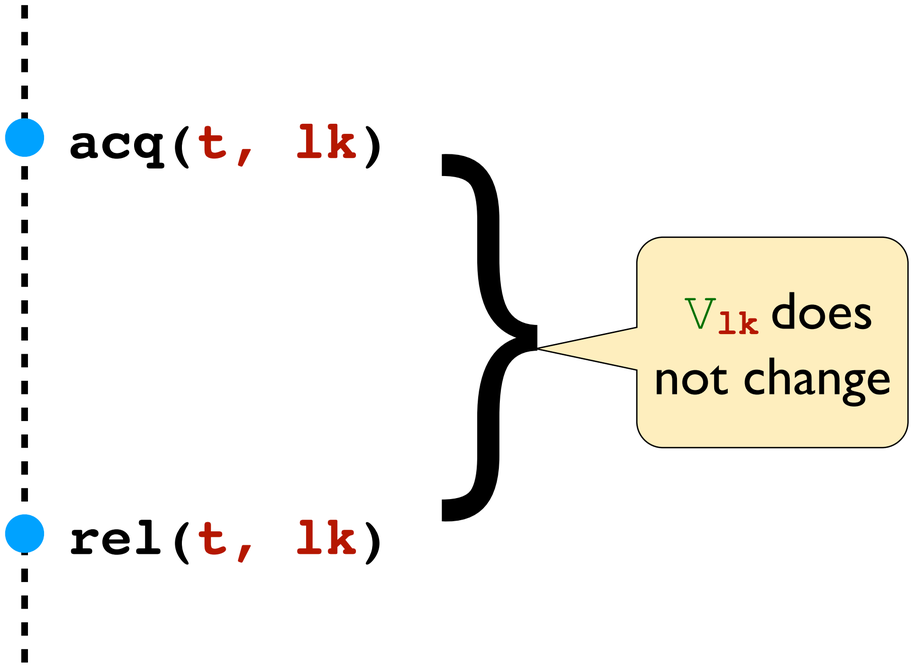
Tree Clock Copy?

In general, tree clock copy will take O(|proceses|) time

Race detection

```
procedure acq(t, lk)
    Vt.JoinWith(Vlk)

procedure rel(t, lk)
    Vlk.CopyFrom(Vt)
```



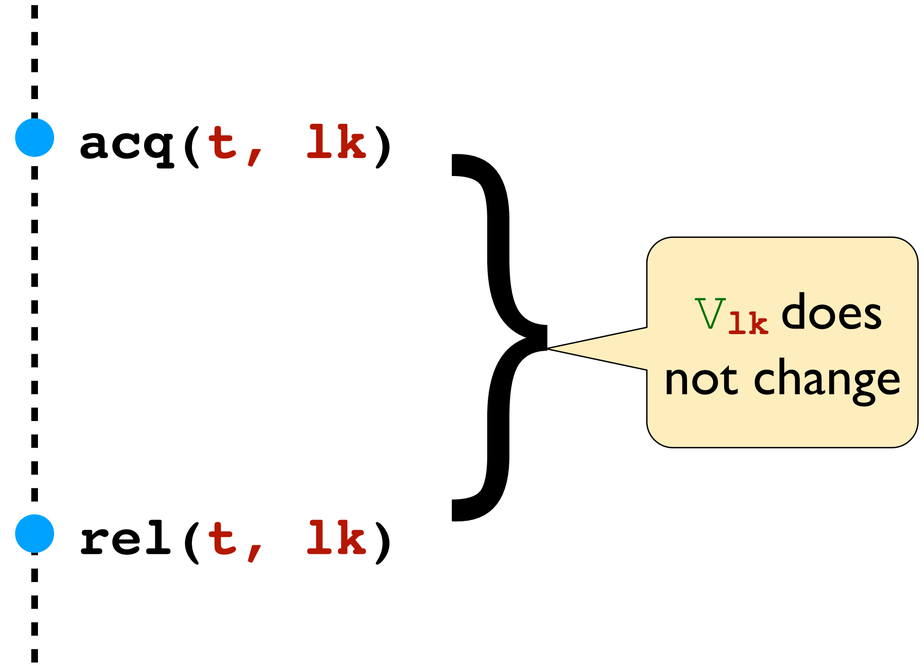
Tree Clock Copy?

In general, tree clock copy will take O(|proceses|) time

Race detection

```
procedure acq(t, lk)
    Vt.JoinWith(Vlk)

procedure rel(t, lk)
    Vlk.MonotoneCopyFrom(Vt)
```



Copy is monotone ⇒ **same semantics as join!**

Data Structure Optimality

Data Structure Optimality

Optimality of data-structure

Data structure D is **optimal**

if

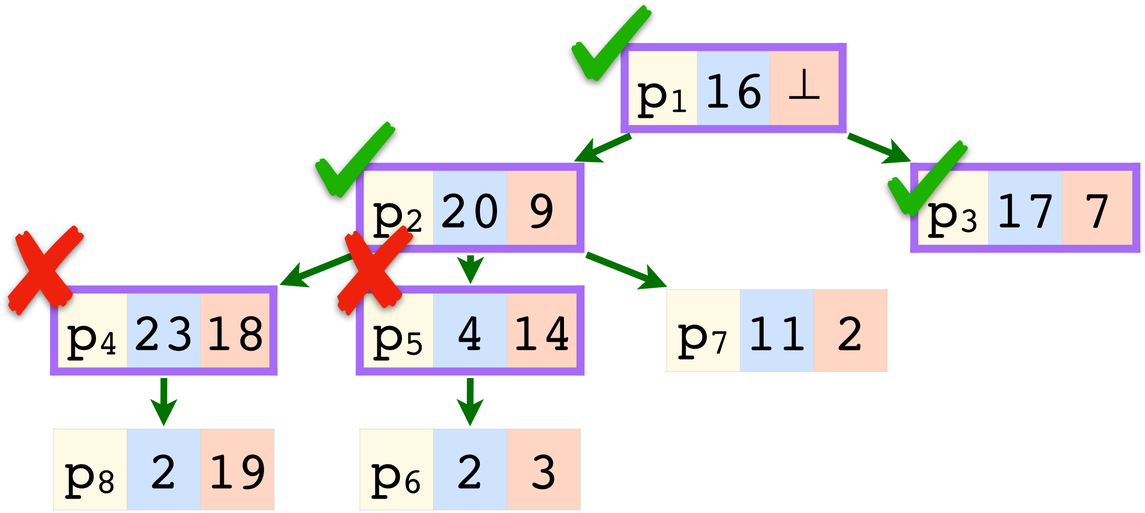
no other data structure can offer asymptotically better performance

Tree Clock Optimality for Race Detection

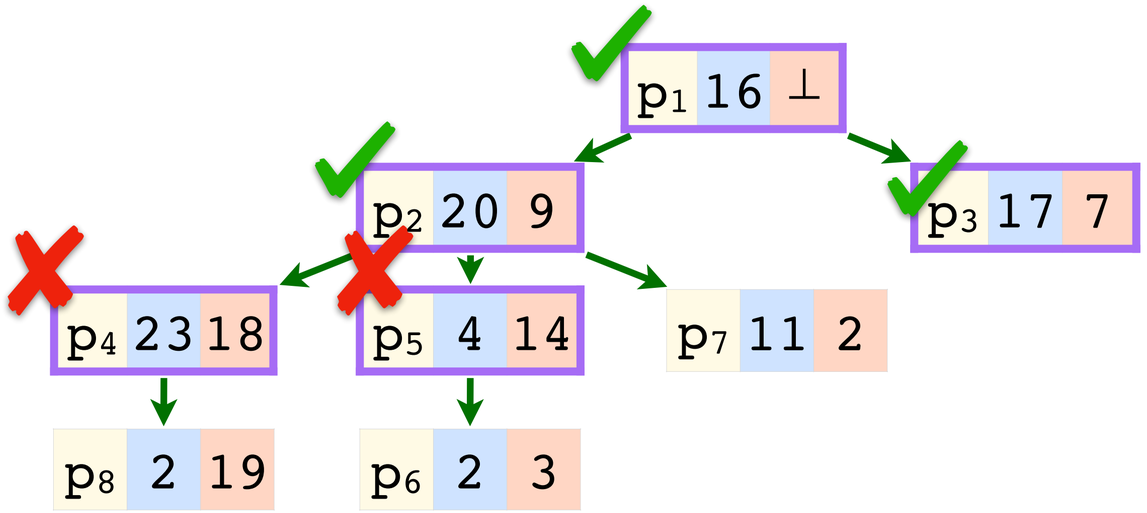
Optimality of Tree-Clocks

Tree Clocks offer **optimal** performance when computing Happens-Before partial order for the purpose of data race detection

Tree Clock Optimality for Race Detection



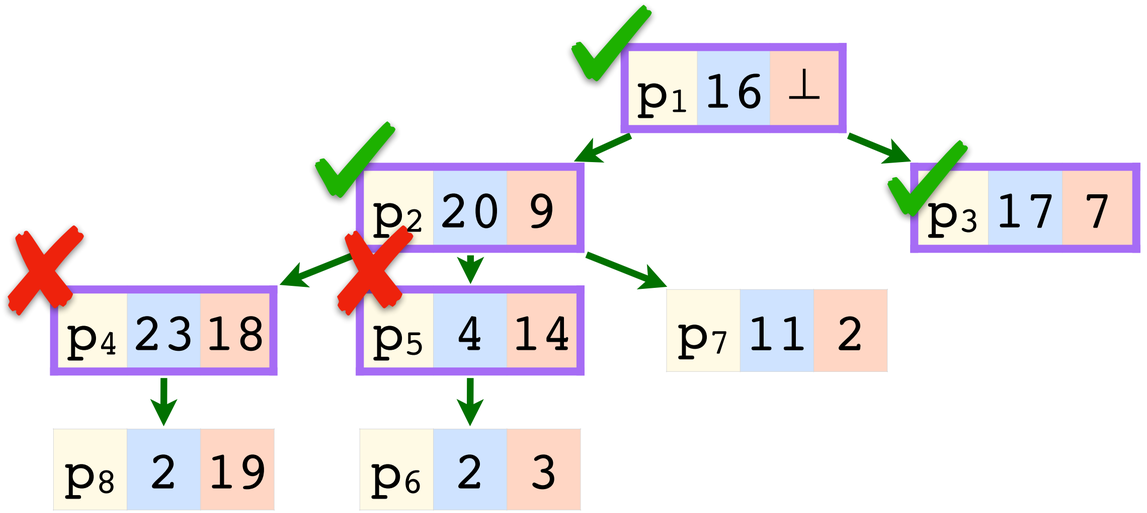
Tree Clock Optimality for Race Detection



Minimum Timestamp Work
MinWork(σ)

Smallest number of data structure accesses when analysing σ

Tree Clock Optimality for Race Detection



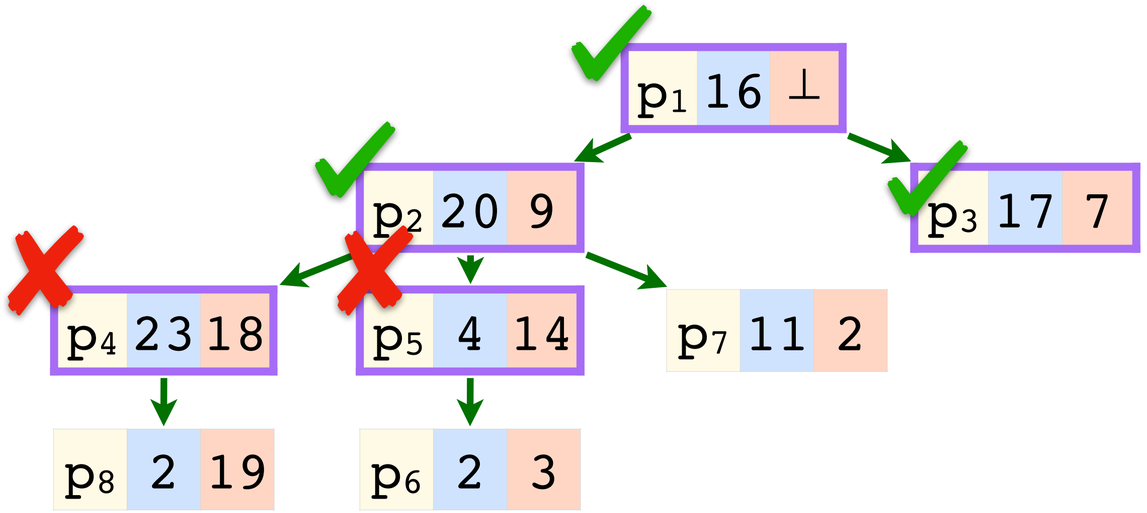
Minimum Timestamp Work
MinWork(σ)

Smallest number of data structure accesses when analysing σ

TreeClock Work
TCWork(σ)

Number of tree clock nodes traversed when analysing σ

Tree Clock Optimality for Race Detection



Minimum Timestamp Work
MinWork(σ)

Smallest number of data structure accesses when analysing σ

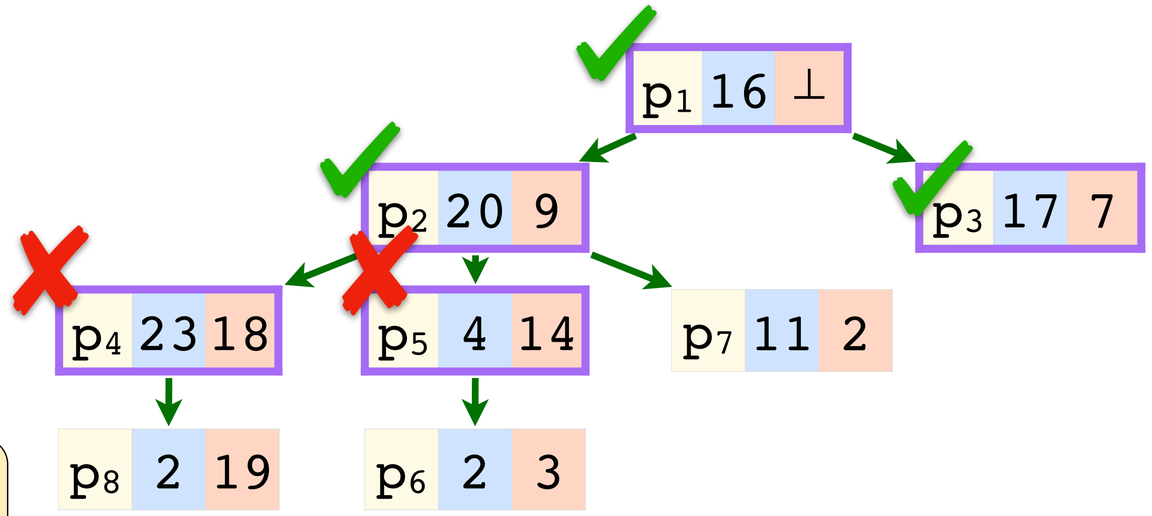
TreeClock Work
TCWork(σ)

Number of tree clock nodes traversed when analysing σ

VectorClock Work
VCWork(σ)

Number of vector clock entires accessed when analysing σ

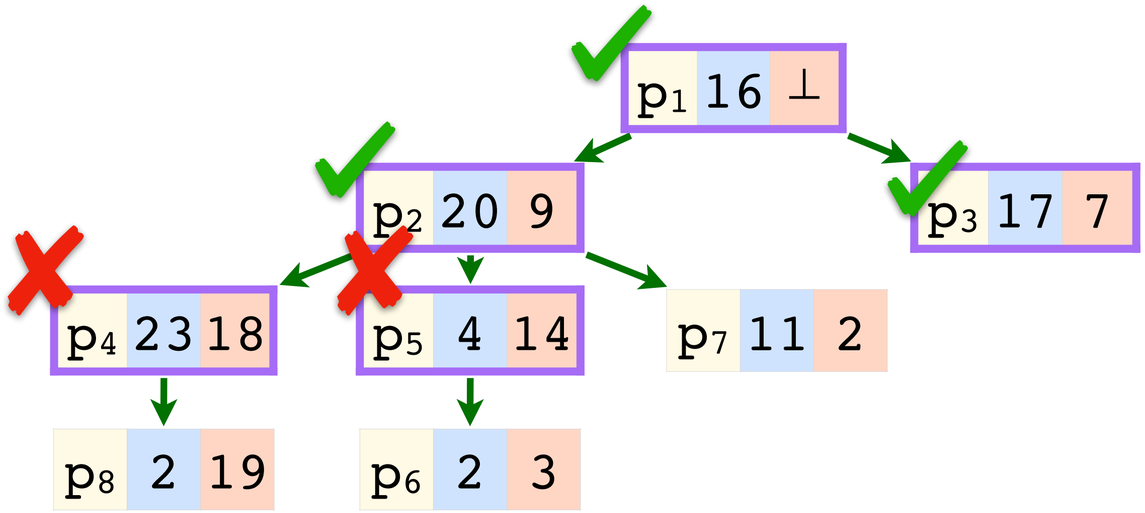
Tree Clock Optimality for Race Detection



k processes

VCWork(σ) ≃ k * MinWork(σ)

Tree Clock Optimality for Race Detection



VCWork(σ) ≃ k * MinWork(σ)

Optimality of Tree-Clocks

TCWork(σ) ≤ 3 * TimeWork(σ)

Does it work?

Does it work?


github.com/umangm/rapid

- 153 benchmarks
- Trace sizes - 51 to 2.1B

Does it work?

github.com/umangm/rapid

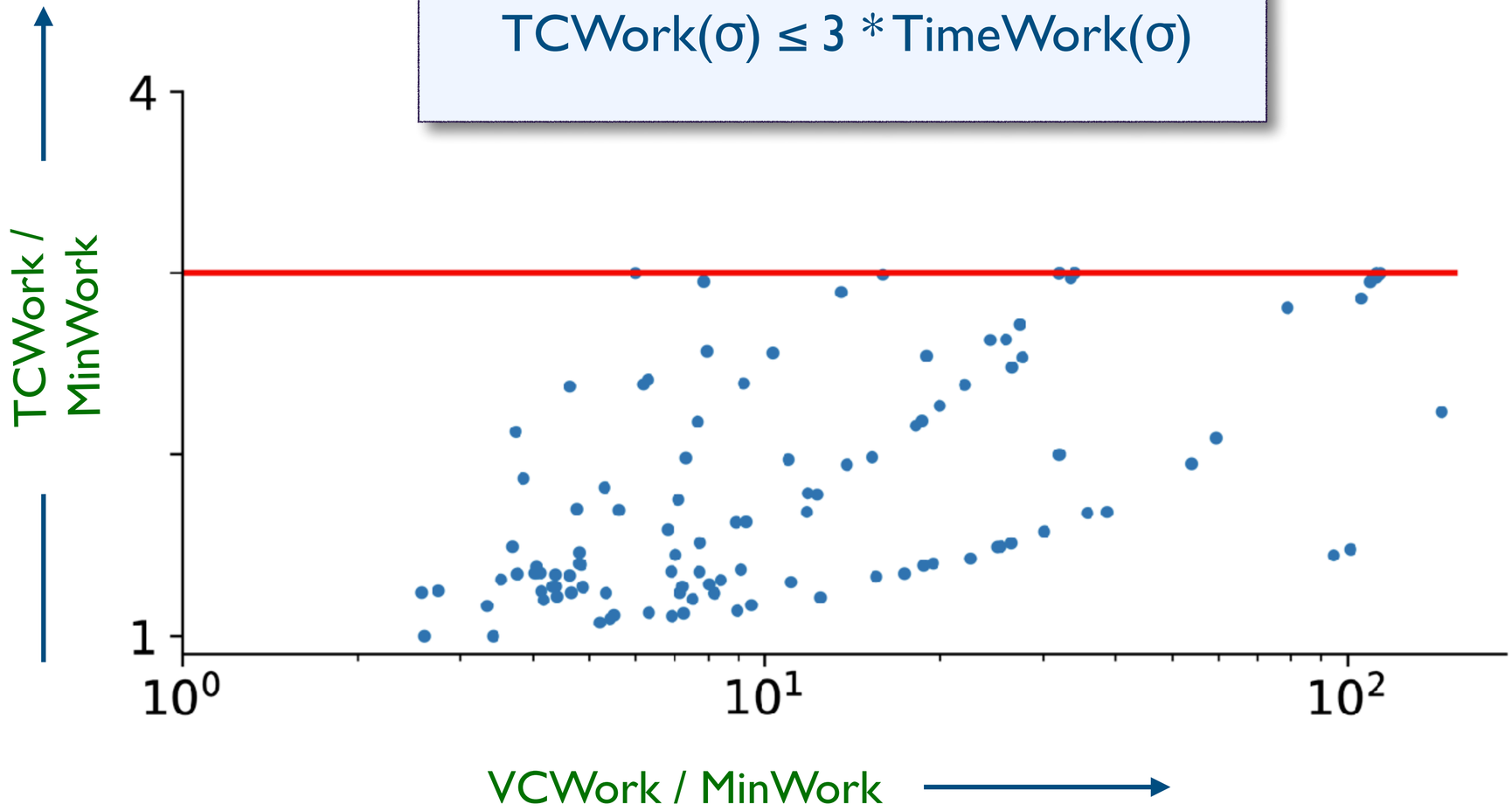
- 153 benchmarks
- Trace sizes - 51 to 2.1B

Task	HB	SHB	Maz
Partial Order	2.97x	2.66x	2.02x
Partial Order + Race Detection	1.11x	1.80x	1.49x

Does it work?

Optimality of Tree-Clocks

$\text{TCWork}(\sigma) \leq 3 * \text{TimeWork}(\sigma)$



To Summarize

To Summarize

- **Timestamping via a data structure lens**
  - Abstract operations stay same
  - Implementation details differ, and matter!

To Summarize

- **Timestamping via a data structure lens**
  - Abstract operations stay same
  - Implementation details differ, and matter!
- **Tree clocks**
  - Store how and when knowledge was discovered
  - Optimal data structure for data race detection
  - Impressive speedups

To Summarize

- **Timestamping via a data structure lens**
  - Abstract operations stay same
  - Implementation details differ, and matter!
- **Tree clocks**
  - Store how and when knowledge was discovered
  - Optimal data structure for data race detection
  - Impressive speedups
- **What's next?**
  - Tree clocks in other domains
  - Design/discovery of new data structures

Thank You!