Advanced Topics in Communication Networks

Programming Network Data Planes

Laurent Vanbever

nsg.ee.ethz.ch

ETH Zürich

Oct 4 2018

Materials inspired from p4.org

Last week on

Advanced Topics in Communication Networks

We will start diving into the P4 ecosystem and look at our first practical usage

P4 environment	P4 language	P4 in practice
What is needed to program in P4?	Deeper-dive into the language constructs	in-network obfuscation
		[USENIX Sec'18]

P4 environment	P4 language	P4 in practice

What is needed to program in P4?

P4$_{16}$ introduces the concept of an *architecture*

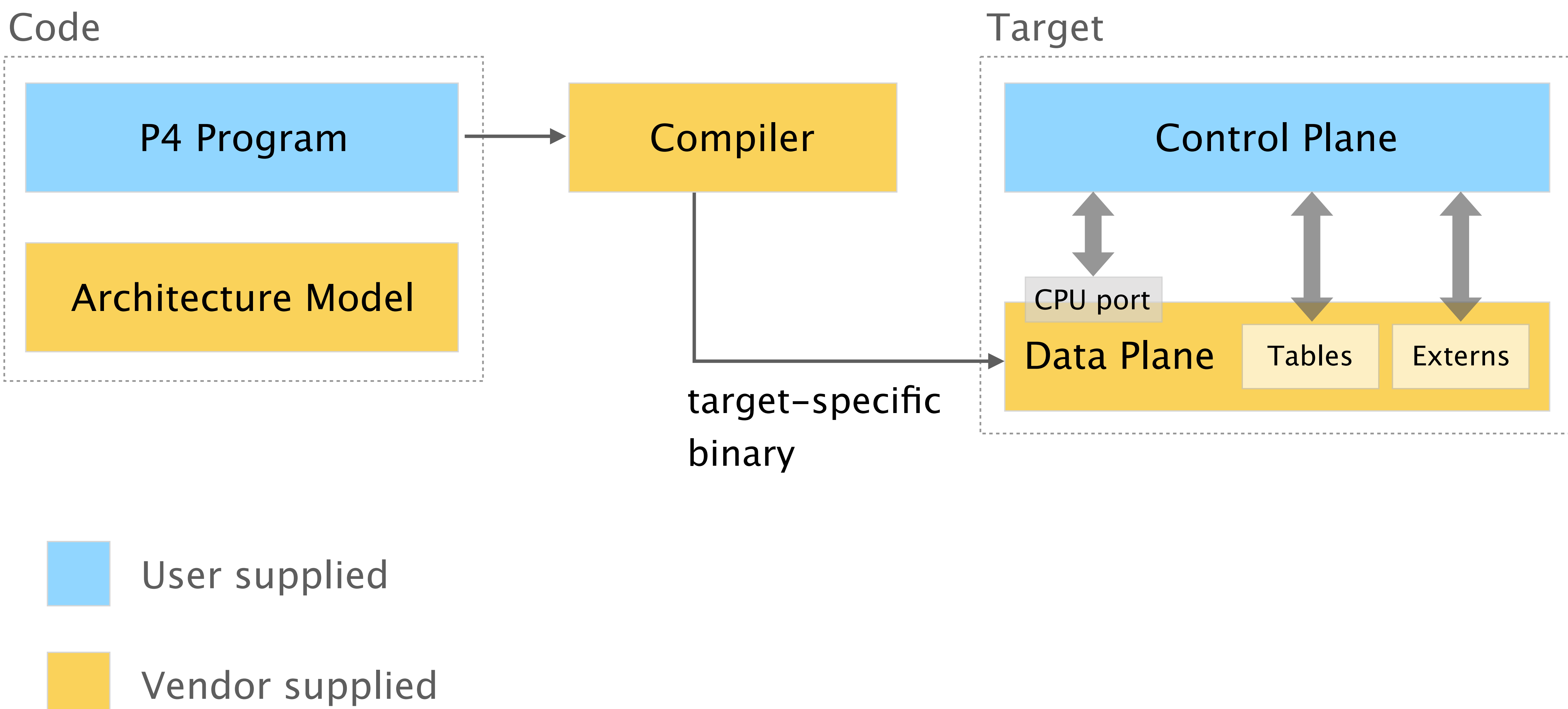
P4 Target

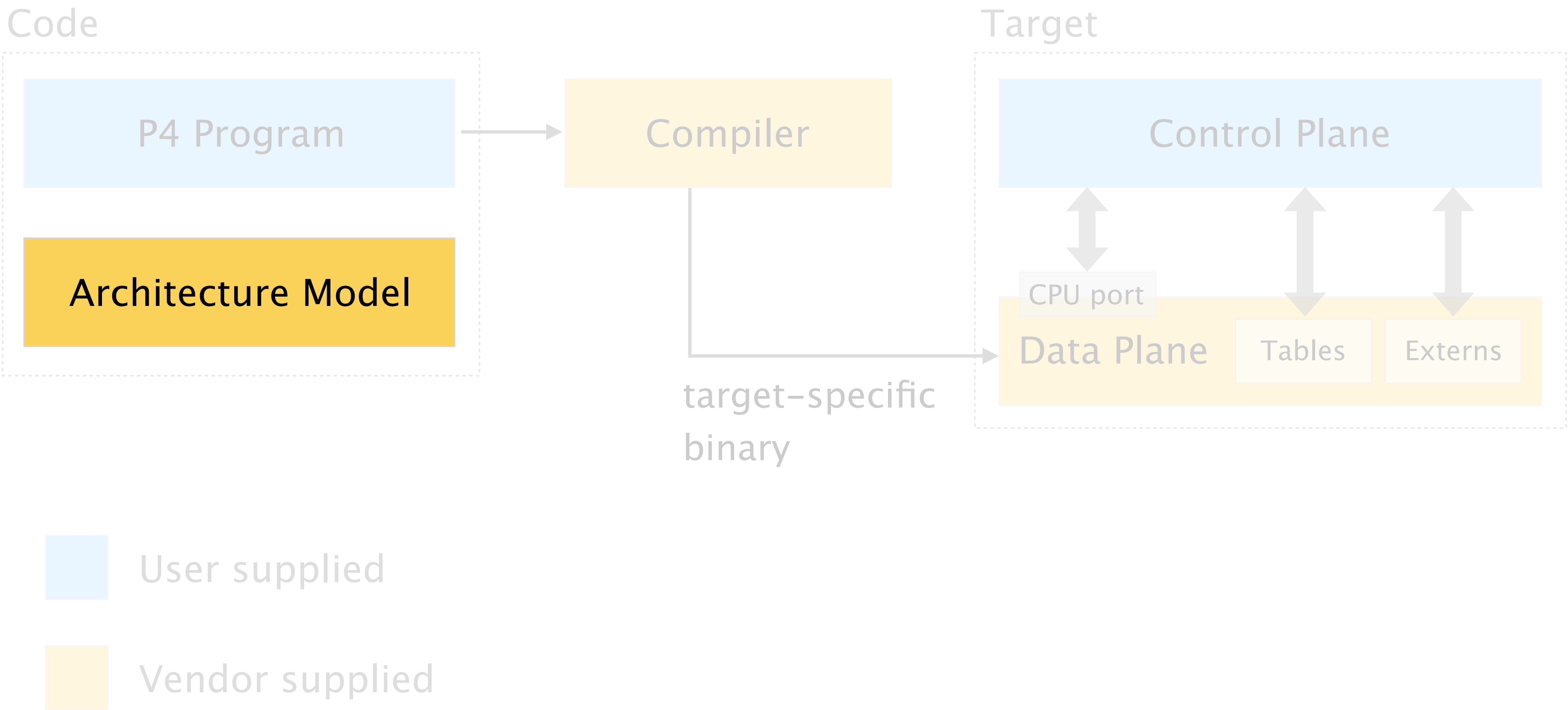
a model of a specific hardware implementation

P4 Architecture

an API to program a target

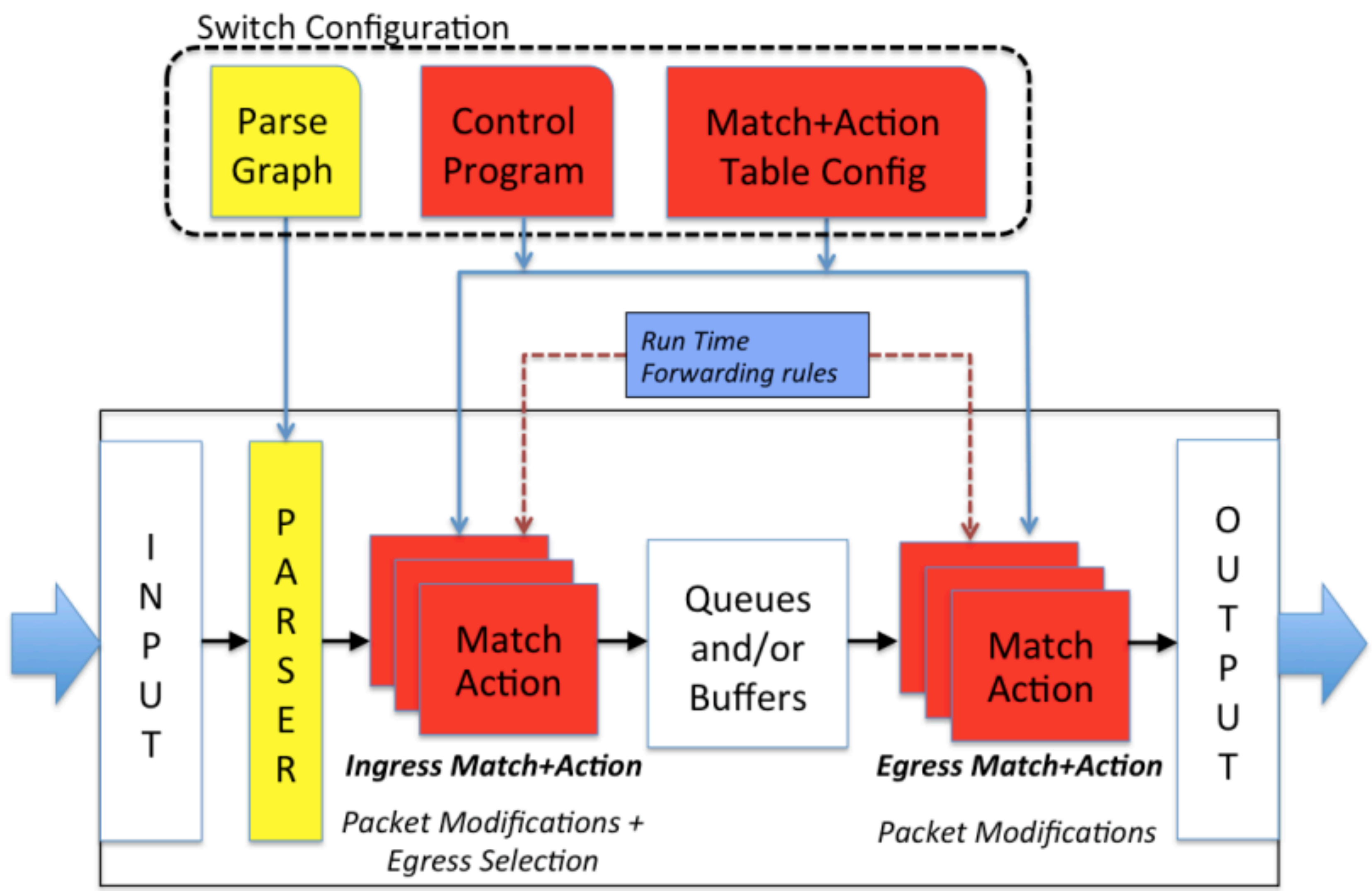
Programming a P4 target involves a few key elements




Code
P4 Program
Architecture Model
Compiler
Target
Control Plane
CPU port
Data Plane
Tables
Externs
target-specific
binary
User supplied
Vendor supplied

We'll rely on a simple P4$_{16}$ switch architecture (v1model) which is roughly equivalent to "PISA"

v1model/
simple switch



source https://p4.org/p4-spec/p4-14/v1.0.4/tex/p4.pdf

Each architecture also defines a list of "externs", i.e. blackbox functions whose interface is known

Most targets contain specialized components

which cannot be expressed in P4 (e.g. complex computations)

At the same time, $P4_{16}$ should be target-independent

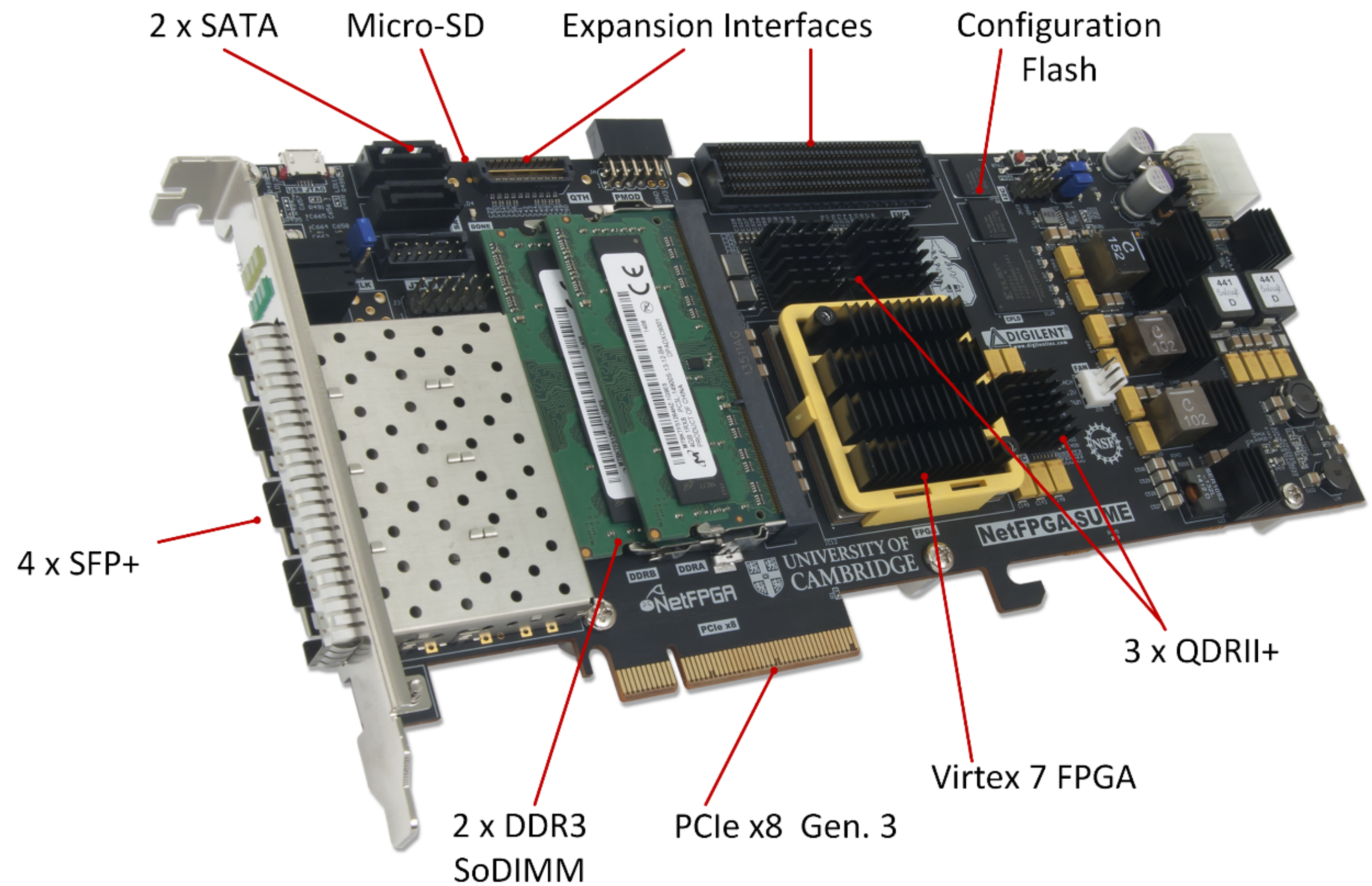
In $P4_{14}$ almost 1/3 of the constructs were target-dependent

Think of externs as Java interfaces

only the signature is known, not the implementation

≠ architectures → ≠ metadata & ≠ externs

NetFPGA-SUME



more info http://isfpga.org/fpga2018/slides/FPGA-2018-P4-tutorial.pdf

P4 environment	P4 language	P4 in practice

Deeper dive into
the language constructs (*)

(*) full info https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html

$P4_{16}$ is a statically-typed language with base types and operators to derive composed ones

`bool`	Boolean value
`bit<W>`	Bit-string of width W
`int<W>`	Signed integer of width W
`varbit<W>`	Bit-string of dynamic length ≤W
`match_kind`	describes ways to match table keys
`error`	used to signal errors
`void`	no values, used in few restricted circumstances
`float`	not supported
`string`	not supported

$P4_{16}$ is a statically-typed language with base types and operators to derive composed ones

Header

```
header Ethernet_h {
 bit<48> dstAddr;
 bit<48> srcAddr;
 bit<16> etherType;
}
```

Header stack

```
header Mpls_h {
 bit<20> label;
 bit<3>  tc;
 bit     bos;
 bit<8>  ttl;
}

Mpls_h[10] mpls;
```

Array of up to 10 MPLS headers

Header union

```
header_union IP_h {
 IPv4_h v4;
 IPv6_h v6;
}
```

Either IPv4 or IPv6 header is present

only one alternative

$P4_{16}$ is a statically-typed language with base types and operators to derive composed ones

Struct

Unordered collection
of named members

```
struct standard_metadata_t {
  bit<9> ingress_port;
  bit<9> egress_spec;
  bit<9> egress_port;
  …
}
```

Tuple

Unordered collection
of unnamed members

```
tuple<bit<32>, bool> x;
x = { 10, false };
```

P4 operations are similar to C operations and vary depending on the types (unsigned/signed ints, ...)

arithmetic operations	`+, -, *`	
logical operations	`~, &, |, ^, >>, <<`	
non-standard operations	`[m:l]`	Bit-slicing
	`++`	Bit concatenation
✗ *no* division and modulo		(can be approximated)

more info https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html

Variables have local scope and their values is not maintained across subsequent invocations

important — variables *cannot* be used to maintain state between different network packets

instead to maintain state — you can only use two stateful constructs

- tables — modified by control plane
- extern objects — modified by control plane & data plane

more on this next week

This week on

Advanced Topics in Communication Networks

stateful programming	statefulness in practice	probabilistic data structures
How do you build stateful apps?	fast network convergence	bloom filters
		part 1

stateful programming	statefulness in practice	probabilistic data structures
How do you build stateful apps?	fast network convergence	bloom filters
		part 1

Stateless and stateful objects in P4

Stateless objects	Stateful objects
Reinitialized for each packet	Keep state between packets

Variables
Headers

Tables
Registers
Counters
Meters
...

Stateless and stateful objects in P4

Stateless objects	Stateful objects
Reinitialized for each packet	Keep state between packets

Variables
Headers

Tables
Registers
Counters
Meters
...

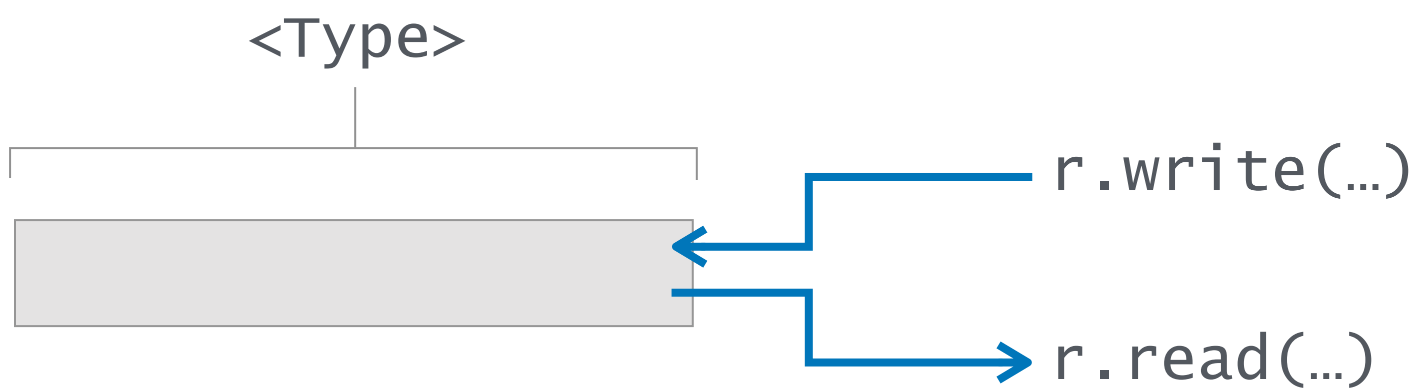
Stateful objects in P4

Table	managed by the control plane	
Register	store arbitrary data	externs in v1model
Counter	count events	externs in v1model
Meter	rate-limiting	externs in v1model
…	…	externs in v1model

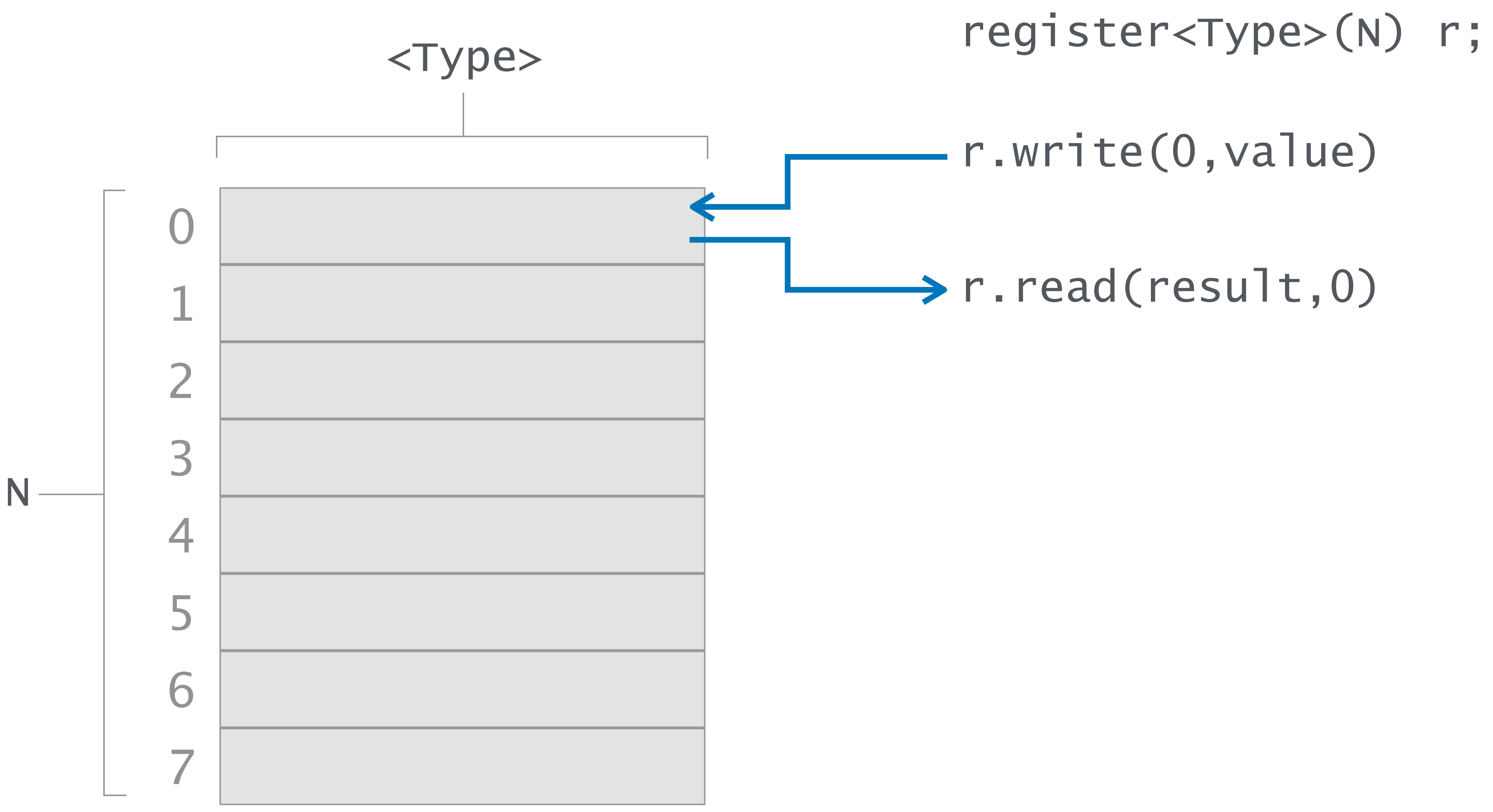
Stateful objects in P4

Table	managed by the control plane	
Register	store arbitrary data	externs in v1model
Counter	count events	externs in v1model
Meter	rate-limiting	externs in v1model
...	...	externs in v1model

Registers are useful for storing (small amounts of) arbitrary data



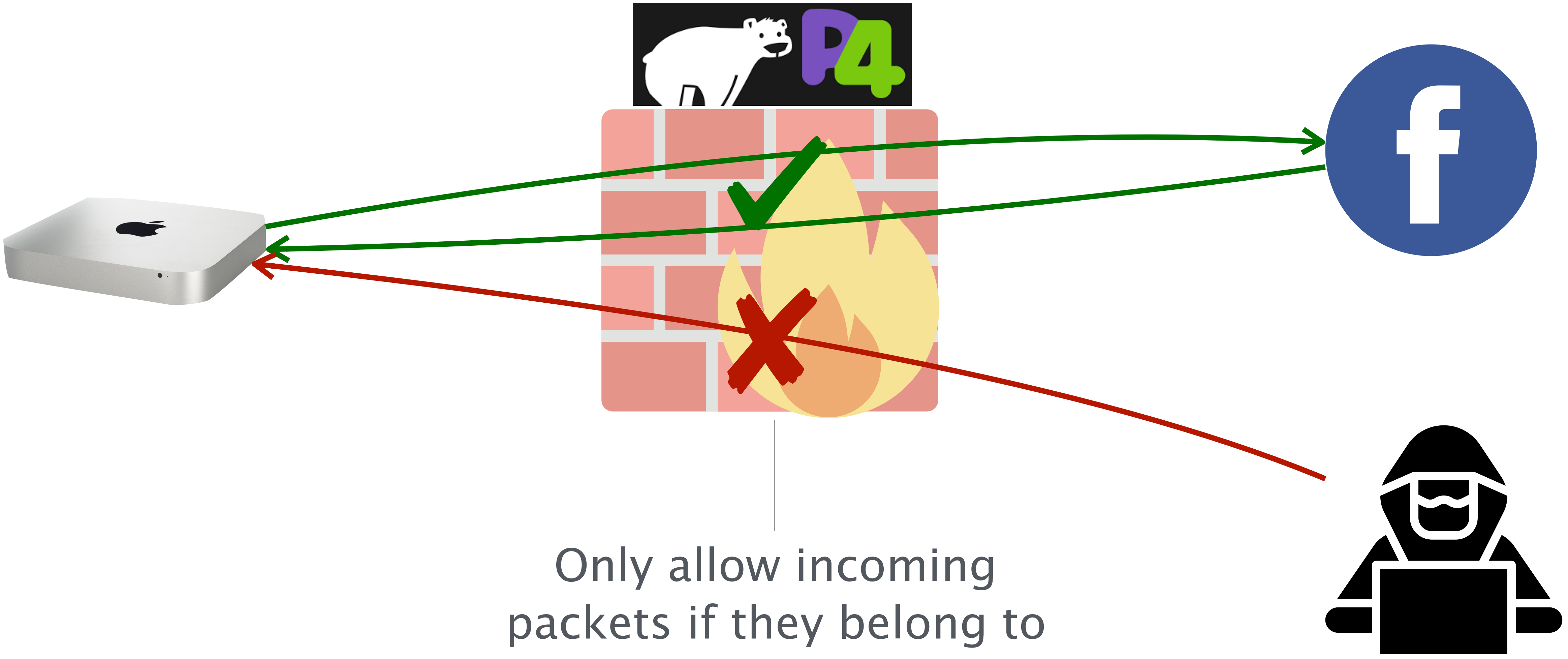
Registers are assigned in arrays



Example: Calculating inter packet gap

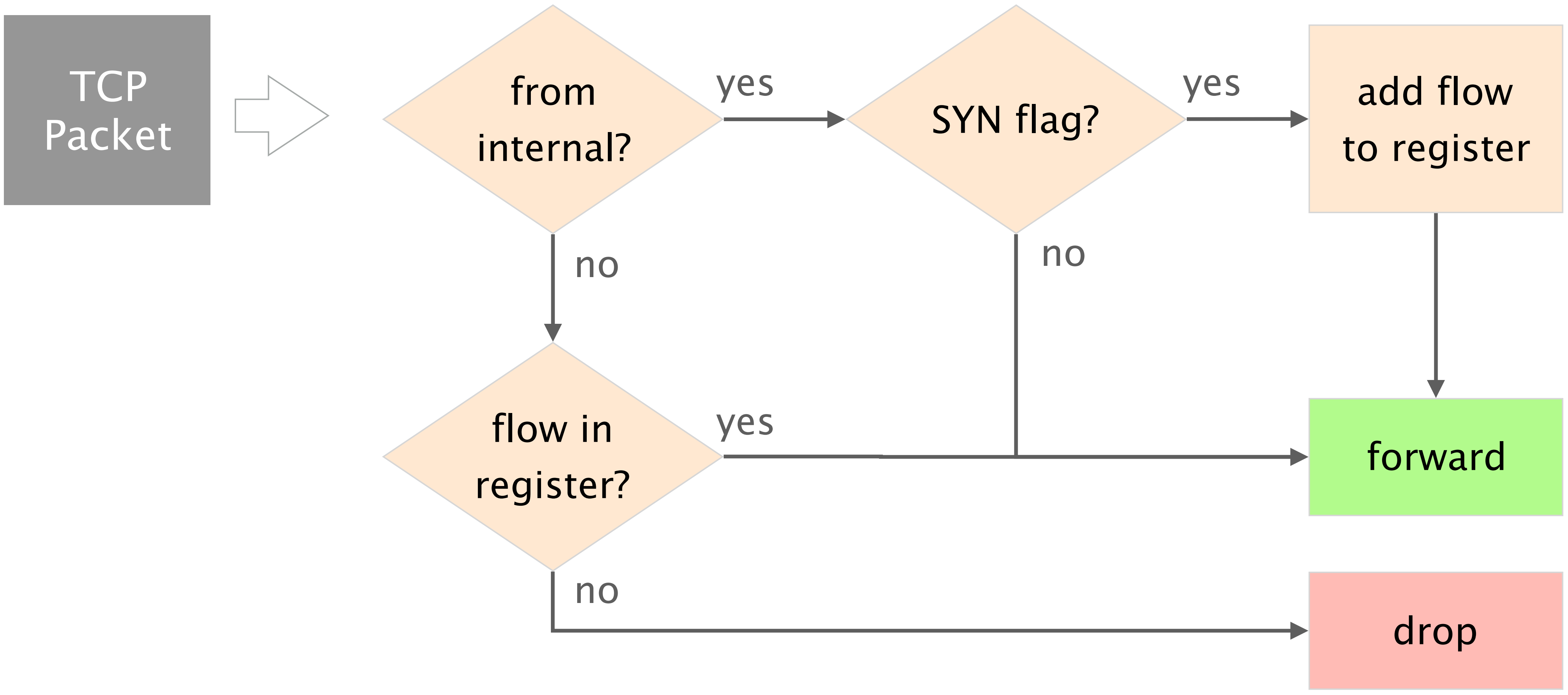
```
register<bit<48>>(16384) last_seen;

action get_inter_packet_gap(out bit<48> interval,bit<32> flow_id)
{
 bit<48> last_pkt_ts;

 /* Get the time the previous packet was seen */
 last_seen.read(last_pkt_ts, flow_id);

 /* Calculate the time interval */
 interval = standard_metadata.ingress_global_timestamp - last_pkt_ts;

 /* Update the register with the new timestamp */
 last_seen.write(flow_id, standard_metadata.ingress_global_timestamp);

 ...
}
```

Example: Stateful firewall



Only allow incoming packets if they belong to an established connection

Example: Stateful firewall



Example: Stateful firewall

```
control MyIngress(...) {
 register<bit<1>>(4096) known_flows;
 ...
 apply {
  meta.flow_id = ... // hash(5-tuple)
  if (hdr.ipv4.isValid()){
   if (hdr.tcp.isValid()){
    if (standard_metadata.ingress_port == 1){
     if (hdr.tcp.syn == 1){
      known_flows.write(meta.flow_id, 1);
     }
    }

    if (standard_metadata.ingress_port == 2){
     known_flows.read(meta.flow_is_known, meta.flow_id);
     if (meta.flow_is_known != 1){
      drop(); return;
     }
    }
   }
   ipv4_lpm.apply();
  }
 }
}
```

register to memorize established connections

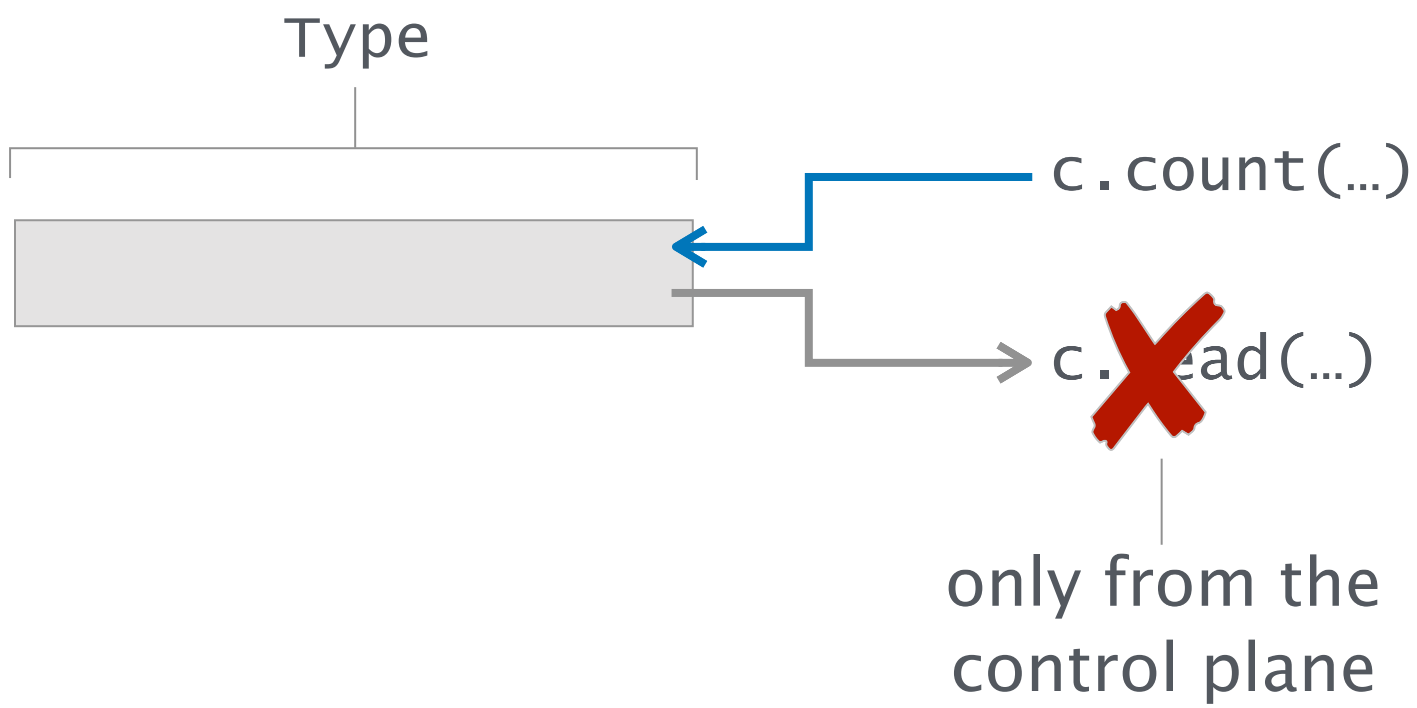
add to register if it it is a SYN packet from internal

drop if the packet does not belong to a known flow and comes from outside

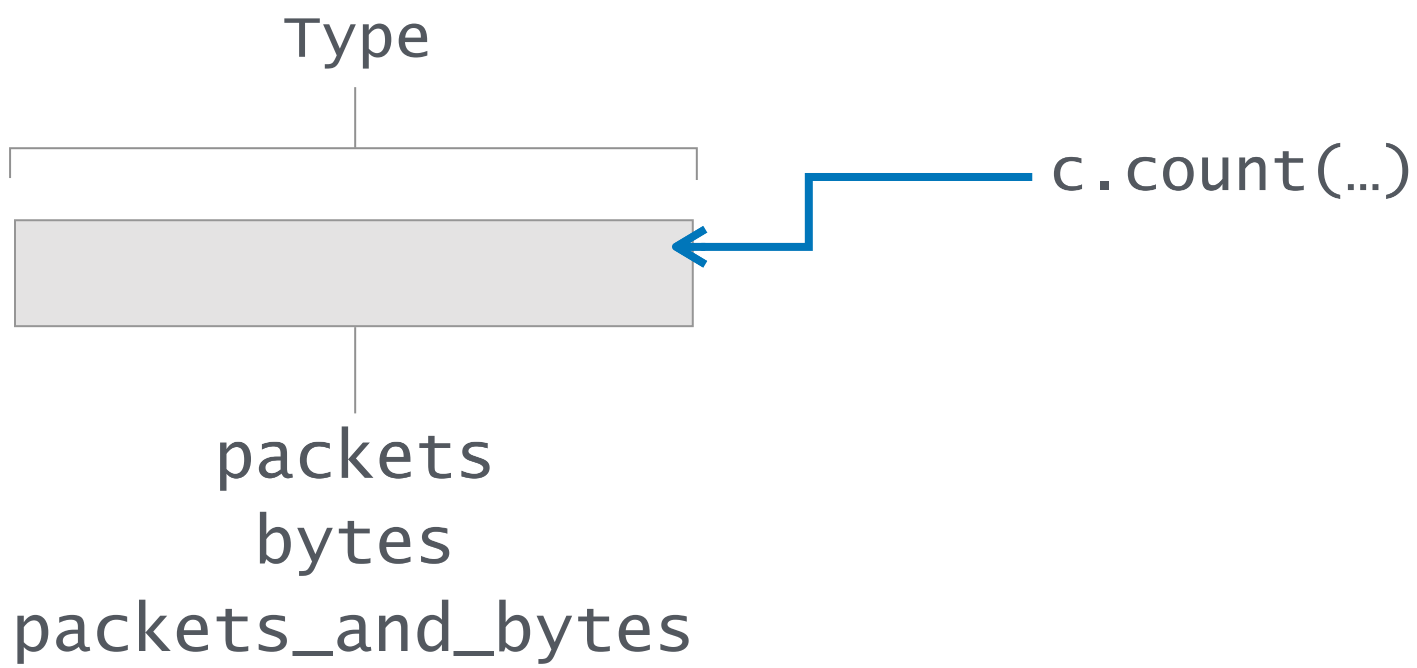
Stateful objects in P4

■ Table	managed by the control plane	
■ Register	store arbitrary data	externs in v1model
■ Counter	count events	externs in v1model
■ Meter	rate-limiting	externs in v1model
■ …	…	externs in v1model

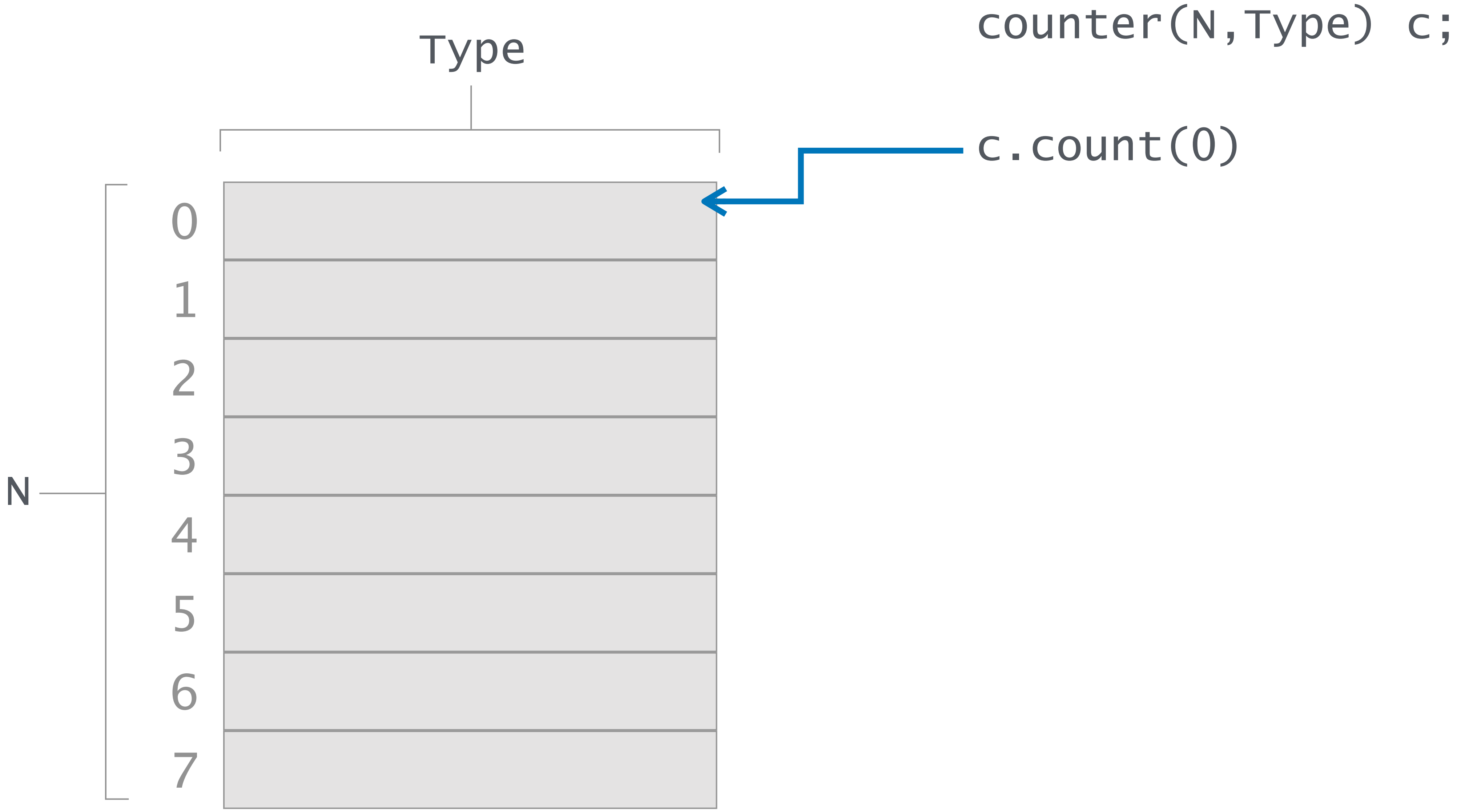
Counters are useful for… counting



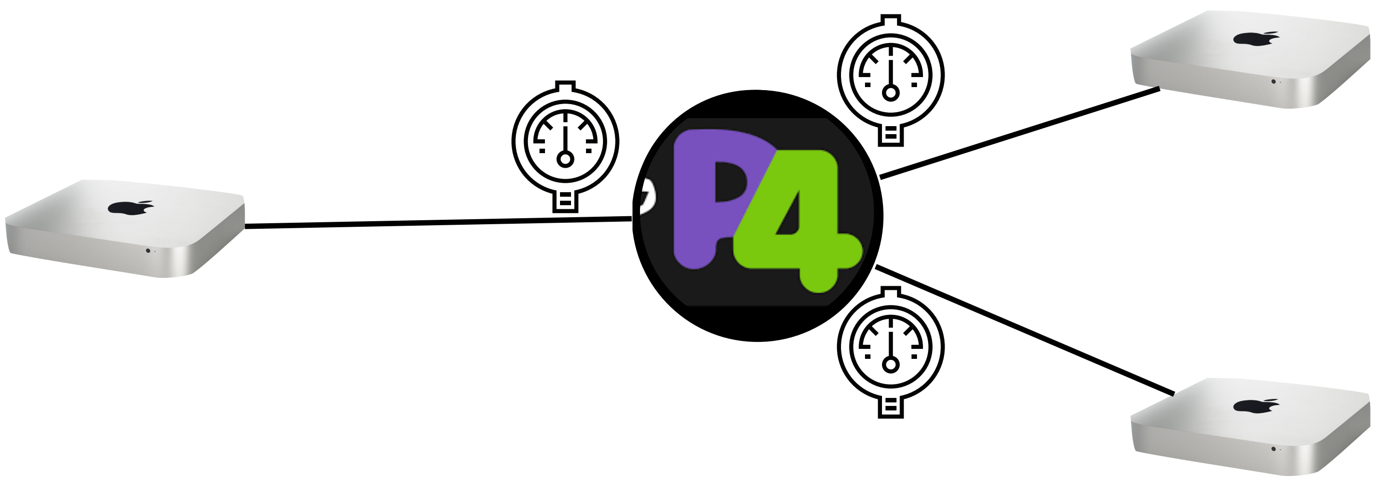
Counters can be of three different types



Like registers, counters are assigned in arrays



Example: Counting packets and bytes arriving at each port



```
control MyIngress(...) {

    counter(512, CounterType.packets_and_bytes) port_counter;

    apply {
        port_counter.count((bit<32>)standard_metadata.ingress_port);
    }
}
```

use the ingress port as counter index

Example: Reading the counter values from the control plane

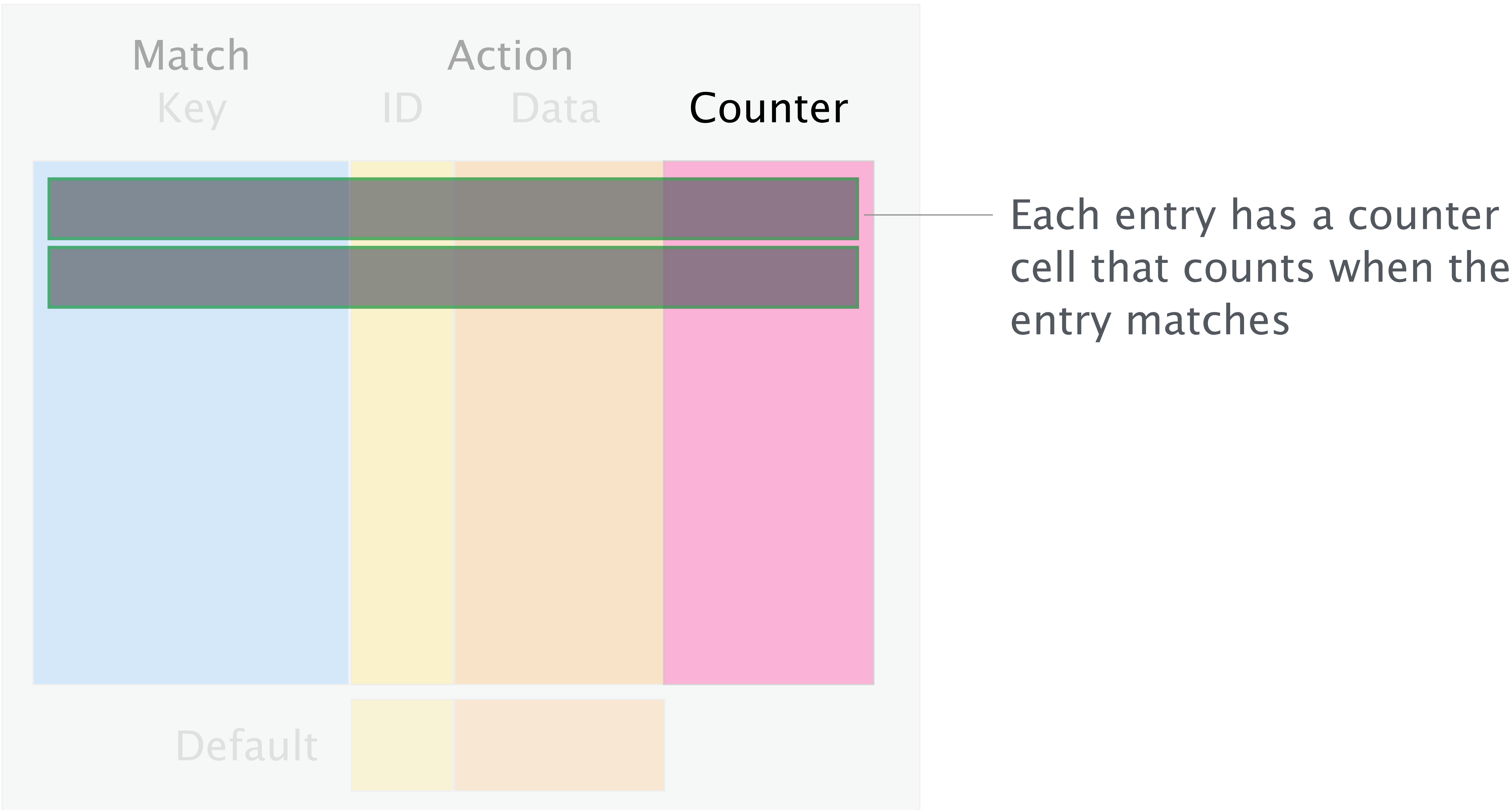
Control Plane

```
RuntimeCmd: counter_read MyIngress.port_counter 1
MyIngress.port_counter[1]=  BmCounterValue(packets=13, bytes=1150)
```

```
control MyIngress(...) {

    counter(512, CounterType.packets_and_bytes) port_counter;

    apply {
        port_counter.count((bit<32>)standard_metadata.ingress_port);
    }
}
```

use the ingress port as counter index

Direct counters are a special kind of counters that are attached to tables



Example: Counting packets and bytes arriving at each port *using a direct counter*

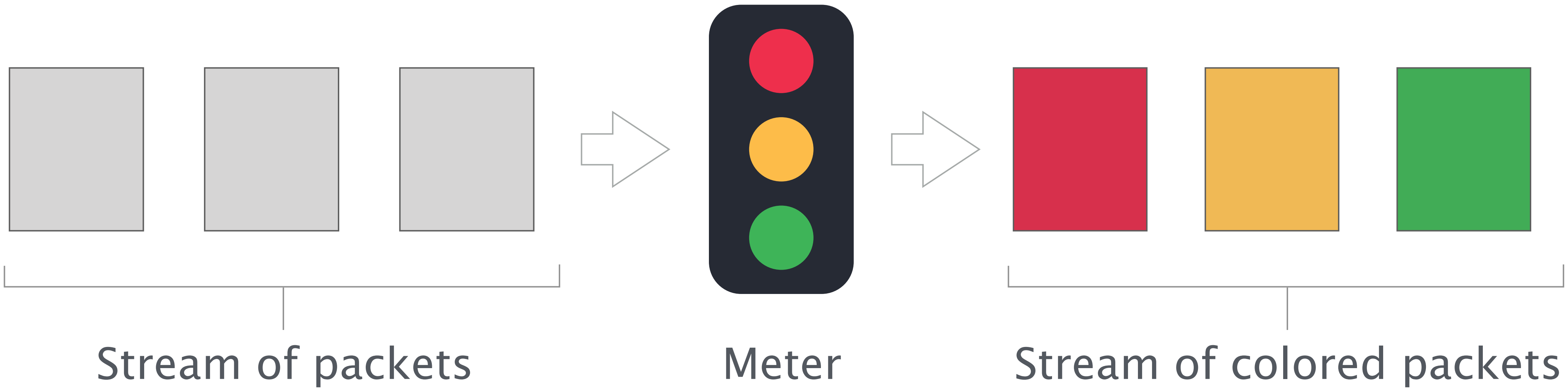
```
control MyIngress(...) {

 direct_counter(CounterType.packets_and_bytes) direct_port_counter;

 table count_table {
  key = {
   standard_metadata.ingress_port: exact;
  }
  actions = {
   NoAction;
  }
  default_action = NoAction;
  counters = direct_port_counter;
  size = 512;
 }

 apply {
  count_table.apply();
 }
}
```

attach counter to table

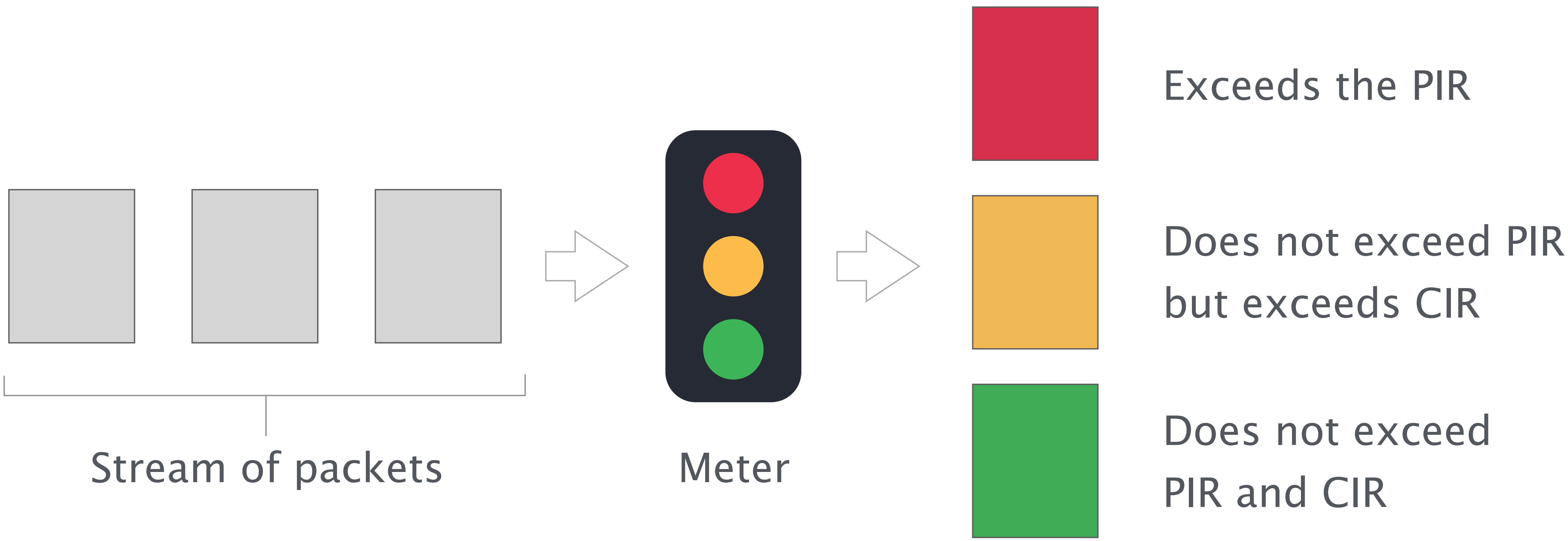
Stateful objects in P4

■ Table	managed by the control plane	
■ Register	store arbitrary data	externs in v1model (Register, Counter, Meter, …)
■ Counter	count events	
■ Meter	rate-limiting	
■ …	…	

Meters



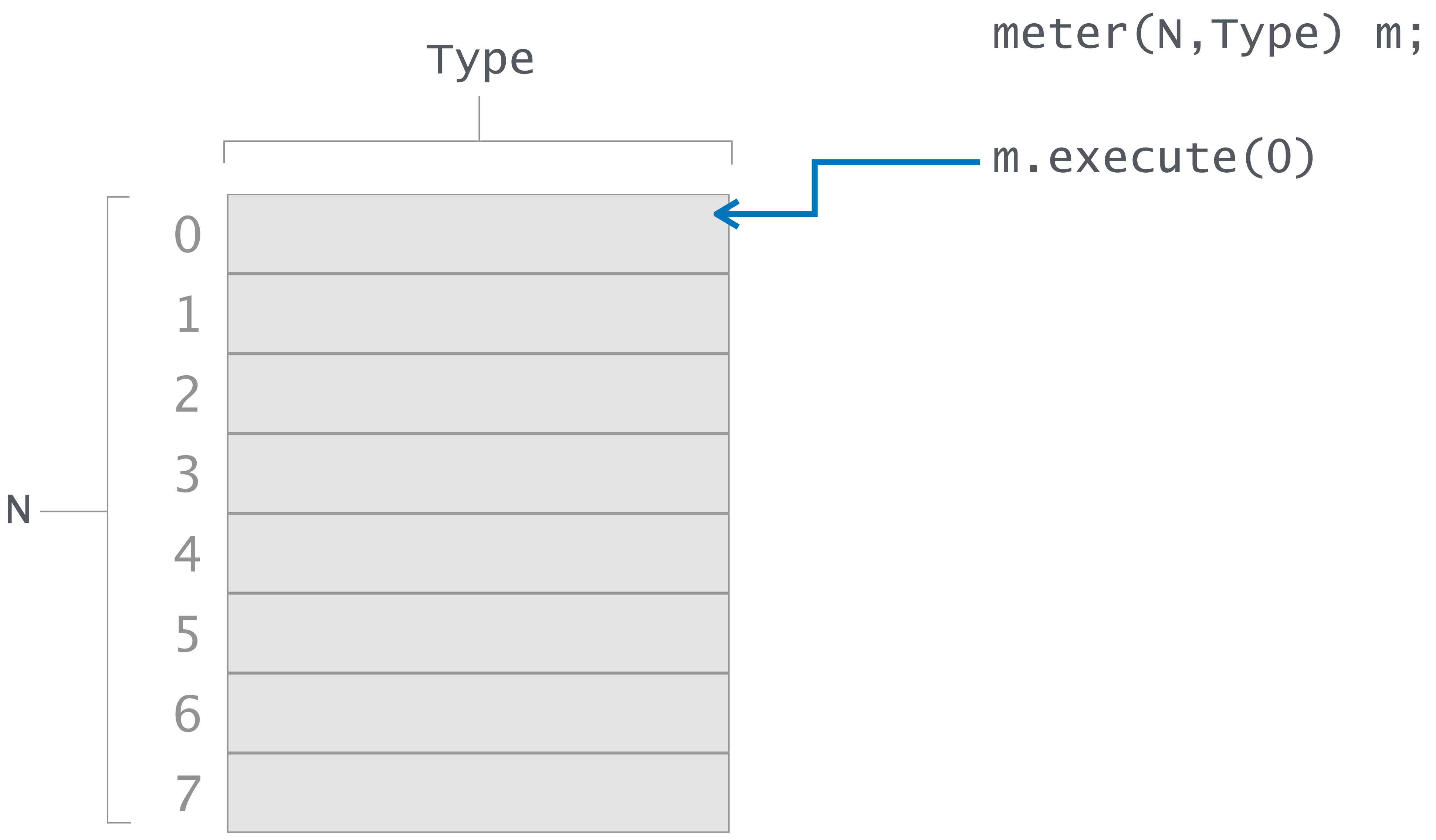
Meters



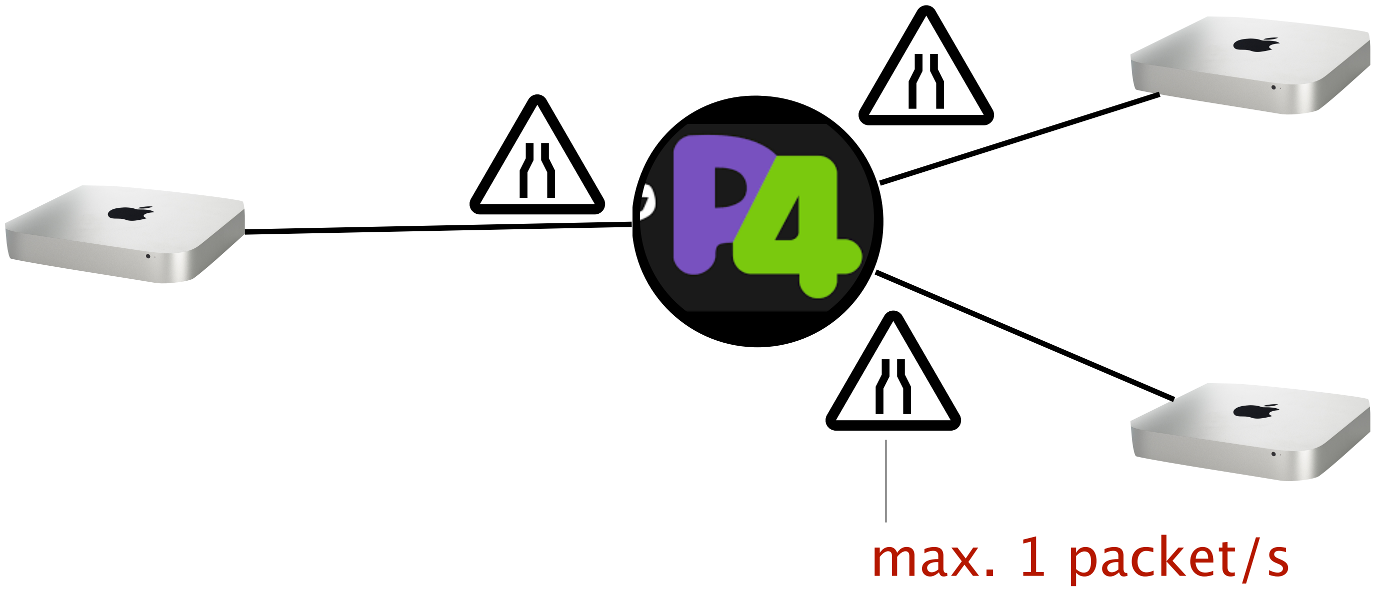
Parameters:	PIR	Peak Information Rate	[bytes/s] or [packets/s]
	CIR	Committed Information Rate	[bytes/s] or [packets/s]

more info https://tools.ietf.org/html/rfc2698

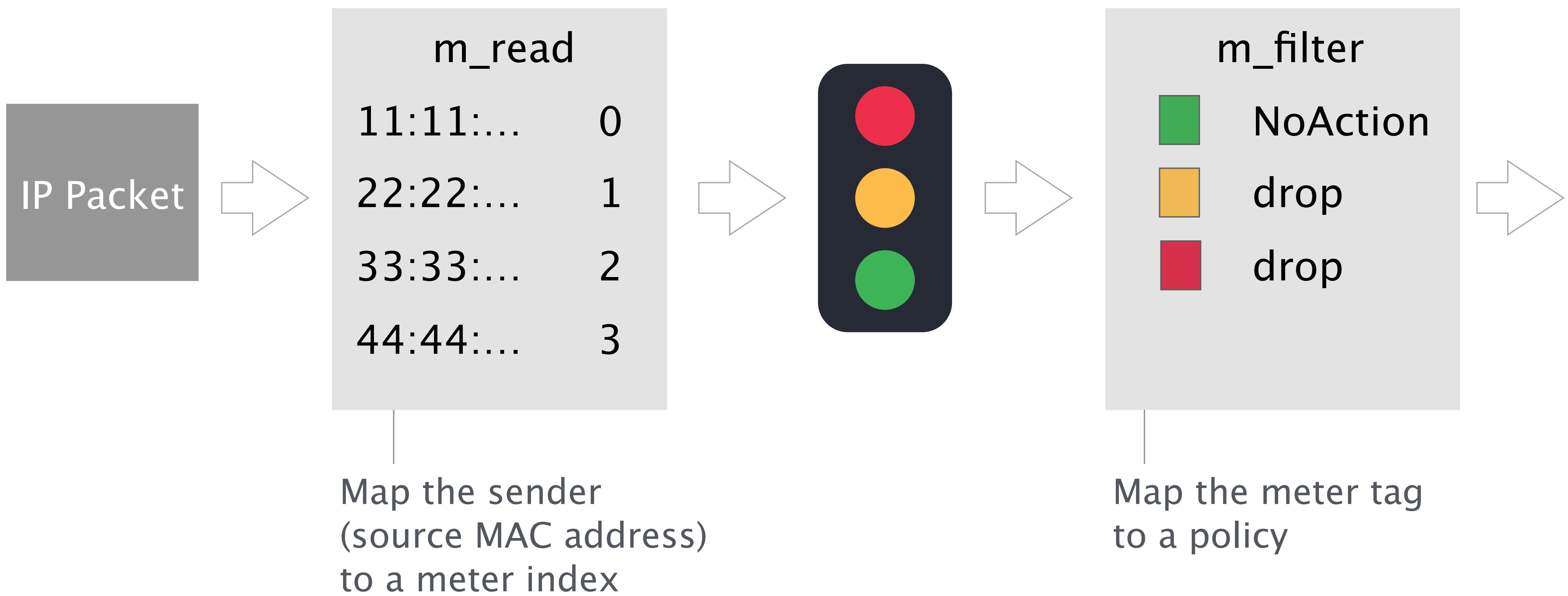
Like registers and counters, meters are assigned in arrays



Example: Using a meter for rate-limiting



Example: Using a meter for rate-limiting



Example: Using a meter for rate-limiting

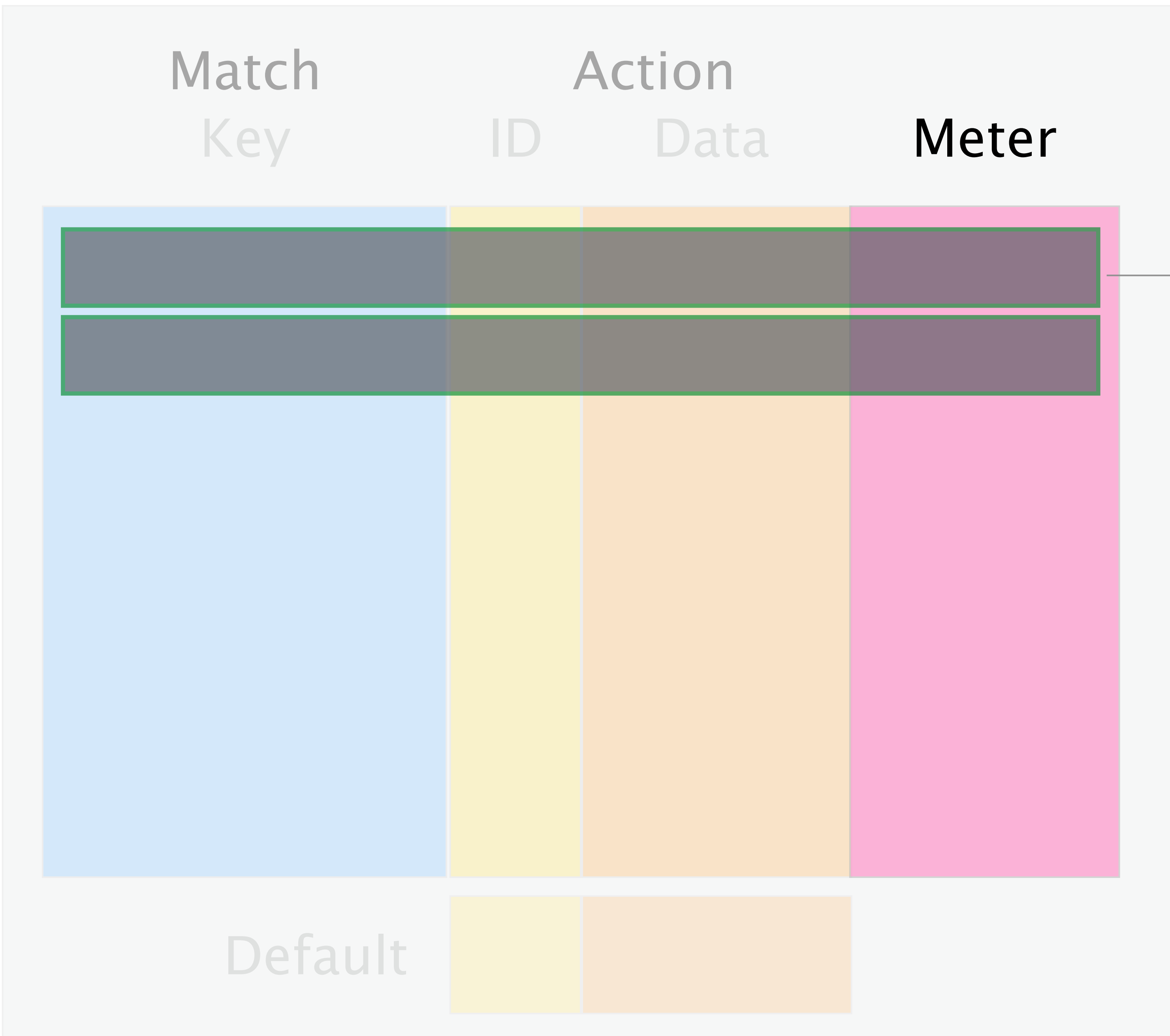
```
control MyIngress(...) {
 meter(32w16384, MeterType.packets) my_meter;

 action m_action(bit<32> meter_index) {
  my_meter.execute_meter<bit<32>>(meter_index, meta.meter_tag);
 }

 table m_read {
  key = { hdr.ethernet.srcAddr: exact; }
  actions = { m_action; NoAction; }
  ...
 }
 table m_filter {
  key = { meta.meter_tag: exact; }
  actions = { drop; NoAction; }
  ...
 }

 apply {
  m_read.apply();
  m_filter.apply();
 }
}
```

packet meter

execute meter

handle packets depending on meter tag

Direct meters are a special kind of meters that are attached to tables



Each entry has a meter cell that is executed when the entry matches

Example: Using a meter for rate-limiting

```
control MyIngress(...) {
 direct_meter<bit<32>>(MeterType.packets) my_meter;

 action m_action(bit<32> meter_index) {
  my_meter.read(meta.meter_tag);
 }

 table m_read {
  key = { hdr.ethernet.srcAddr: exact; }
  actions = { m_action; NoAction; }
  meters = my_meter;
  ...
 }
 table m_filter { ... }

 apply {
  m_read.apply();
  m_filter.apply();
 }
}
```

direct meter

read meter

attach meter to table

Summary

	Data plane interface		Control plane interface	
Object	read	modify/write	read	modify/write
Table	`apply()`	—	yes	yes
Register	`read()`	`write()`	yes	yes
Counter	—	`count()`	yes	reset
Meter	`execute()`		configuration only	

Stateless and stateful objects in P4

Stateless objects	Stateful objects
Reinitialized for each packet	Keep state between packets

Stateless objects	Stateful objects
Variables	Tables
Headers	Registers
	Counters
	Meters
	...

stateful programming	statefulness in practice	probabilistic data structures
How do you build stateful apps?	fast network convergence	bloom filters
		part 1

stateful
programming

How do you build
stateful apps?

statefulness
in practice

fast network
convergence

probabilistic
data structures

bloom
filters

part 1

Programming more advanced stateful data structures

Programming more advanced stateful data structures

We are provided with built-in stateful data structures
such as arrays of registers, counters or meters

We need to deal with severe limitations
such as a limited number of operations and memory

Programming more advanced stateful data structures

We are provided with built-in stateful data structures
such as arrays of registers, counters or meters

We need to deal with severe limitations
such as a limited number of operations and memory

Today: how can we implement a set with its usual methods
i.e., add an element, membership query, delete an element, lookup, listing

There are two common strategies to implement a set

	strategy #1	strategy #2
output	Deterministic	Probabilistic
number of required operations	Probabilistic	Deterministic

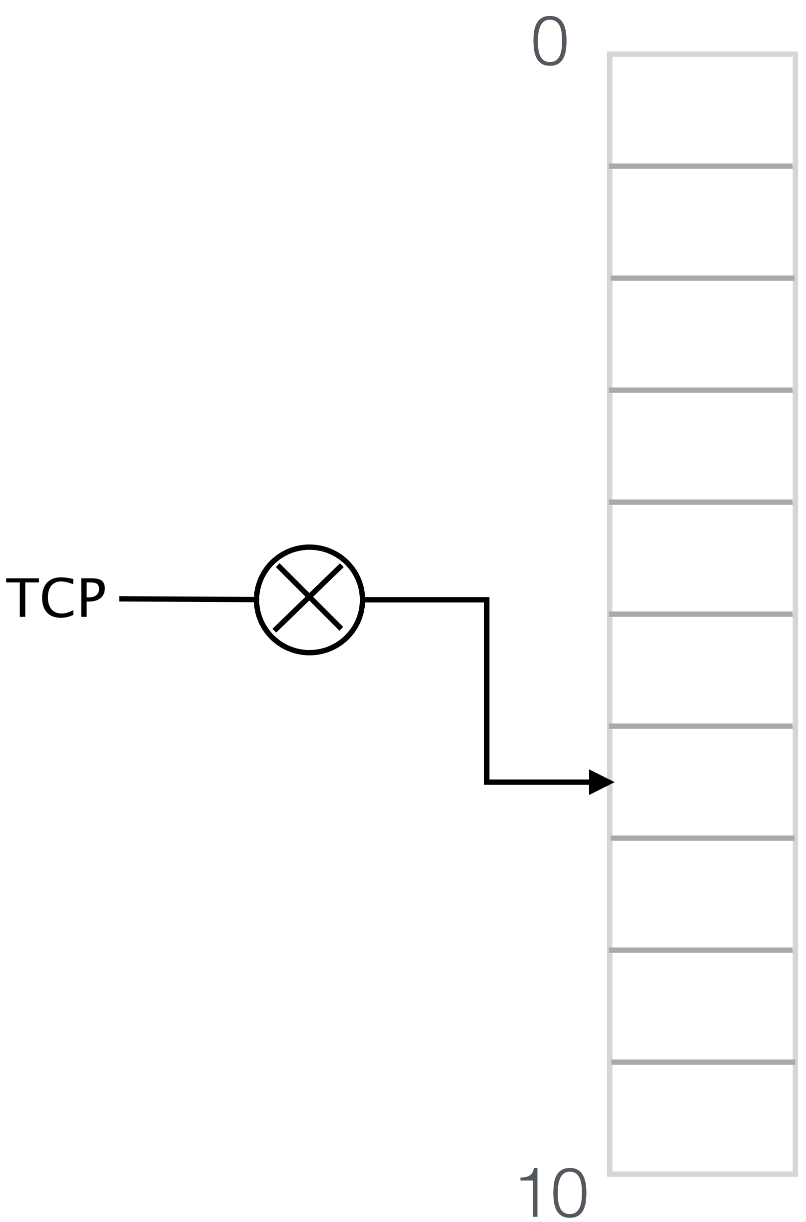
There are two common strategies to implement a set

	strategy #1	strategy #2
output	Deterministic	Probabilistic
number of required operations	Probabilistic	Deterministic

Intuitive implementation of a set

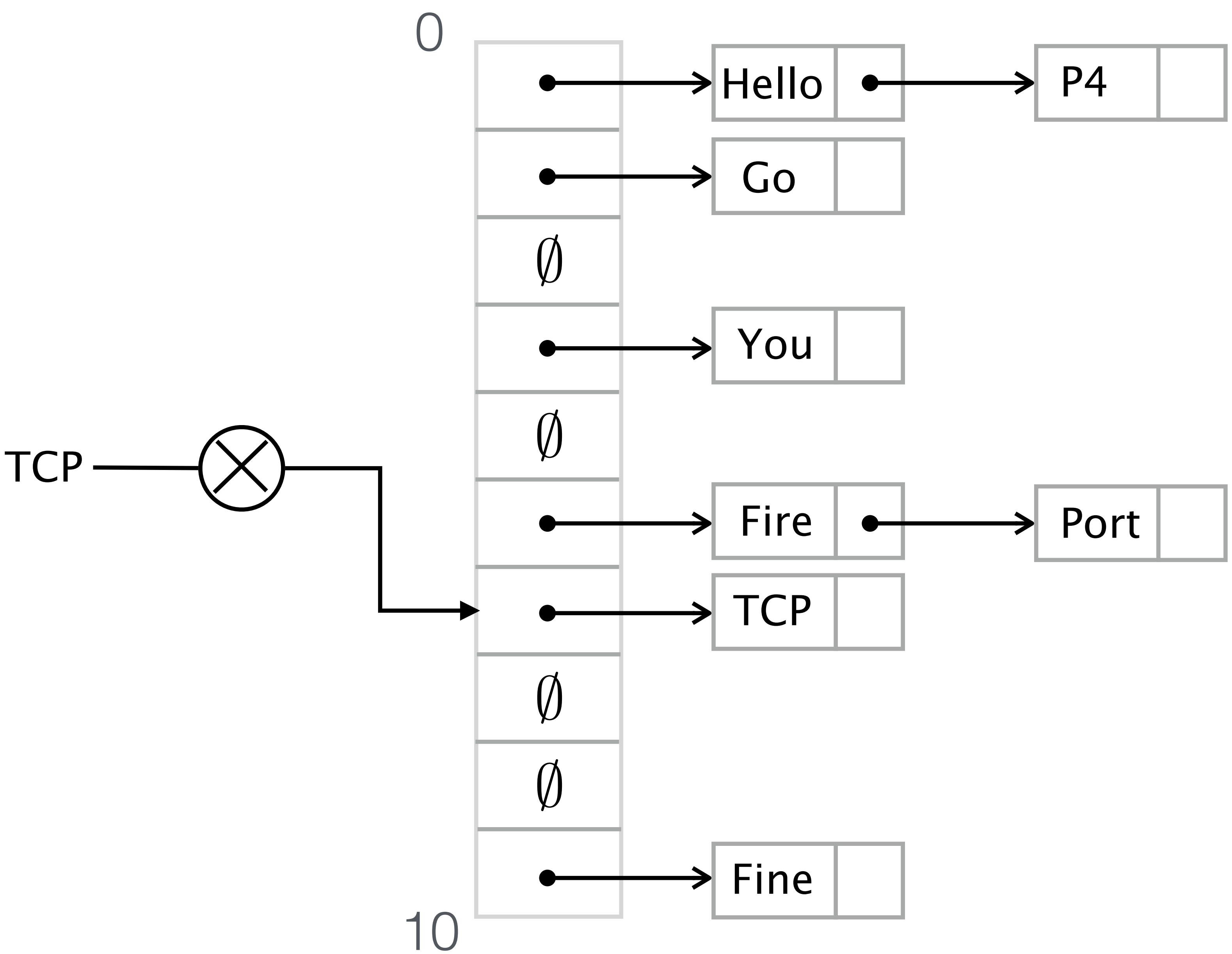
Intuitive implementation of a set

Separate-chaining



Intuitive implementation of a set

Separate-chaining



Intuitive implementation of a set

Separate-chaining

N elements and M cells

	list size
average	N/M
worse-case	N

Intuitive implementation of a set

Separate-chaining

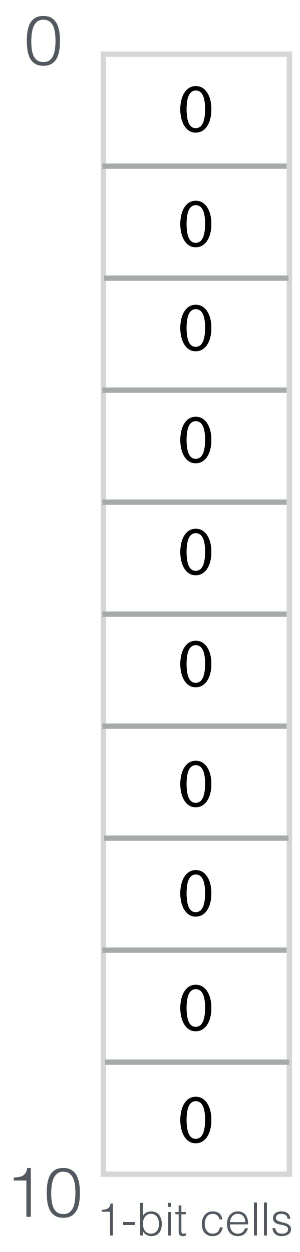
Pros: accurate and fast in the average case

Con: only works in hardware if there is a low number of elements (e.g. < 100)

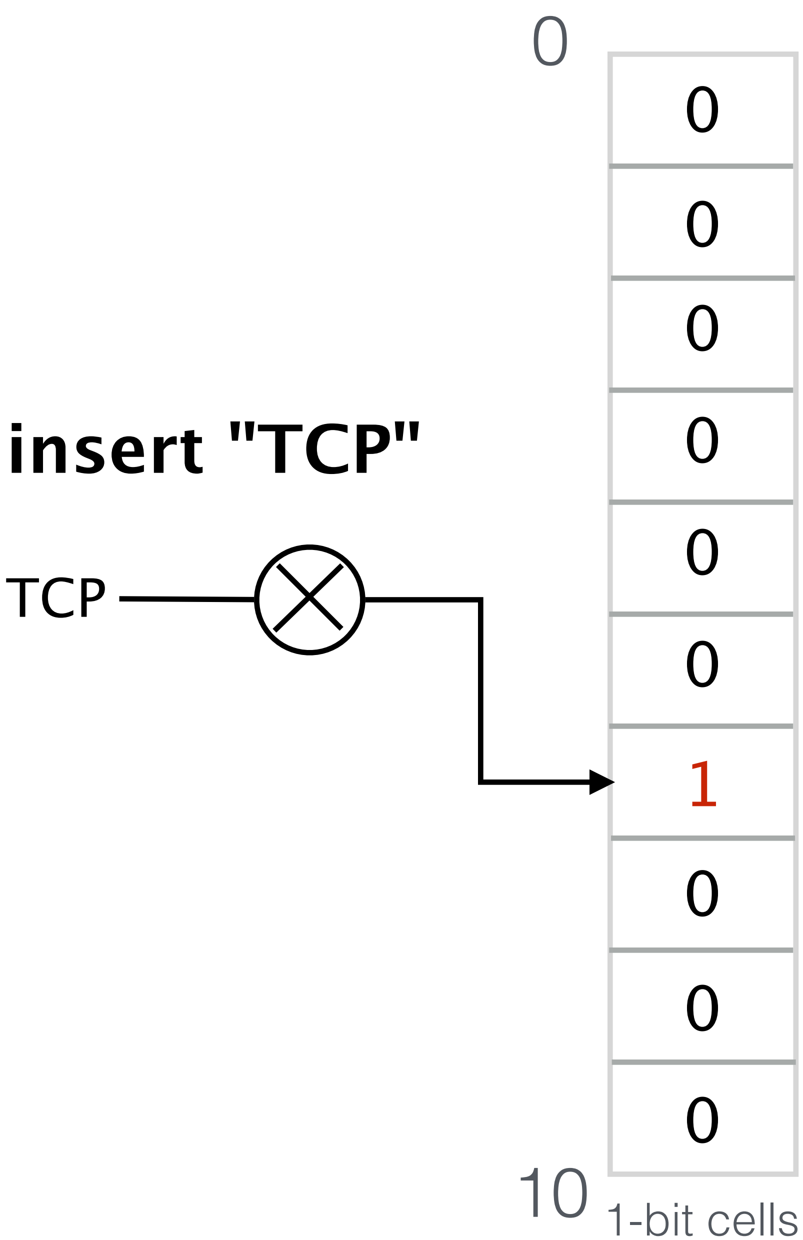
There are two common strategies to implement a set

	strategy #1	strategy #2
output	Deterministic	Probabilistic
number of required operations	Probabilistic	Deterministic

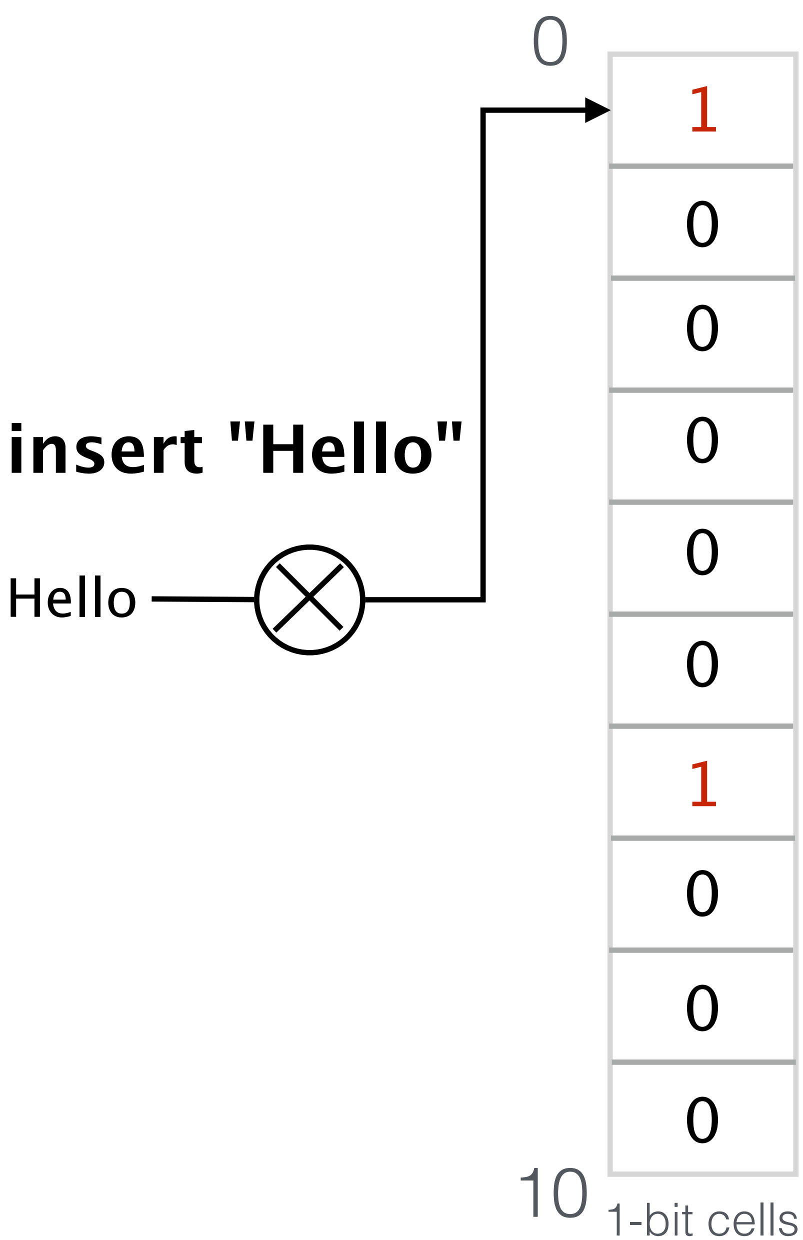
A simple approach for insertions and membership queries



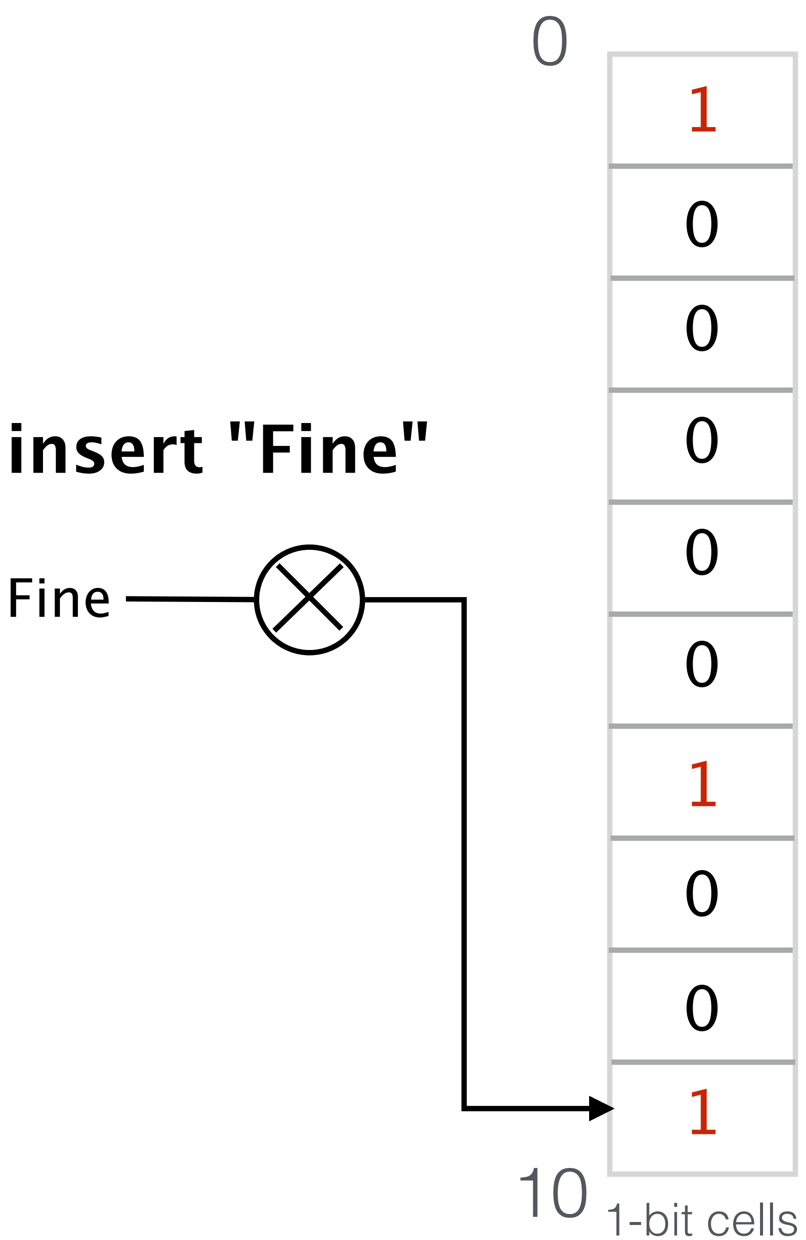
A simple approach for insertions and membership queries



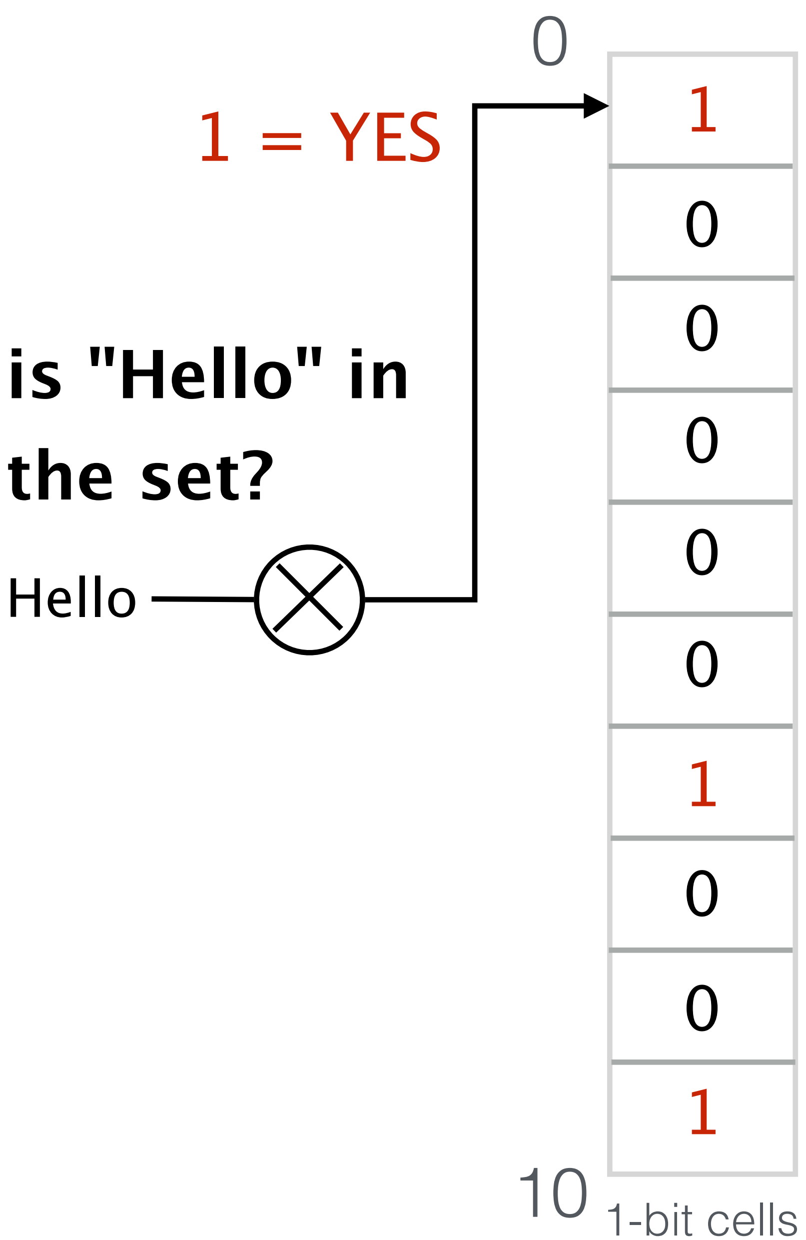
A simple approach for insertions and membership queries



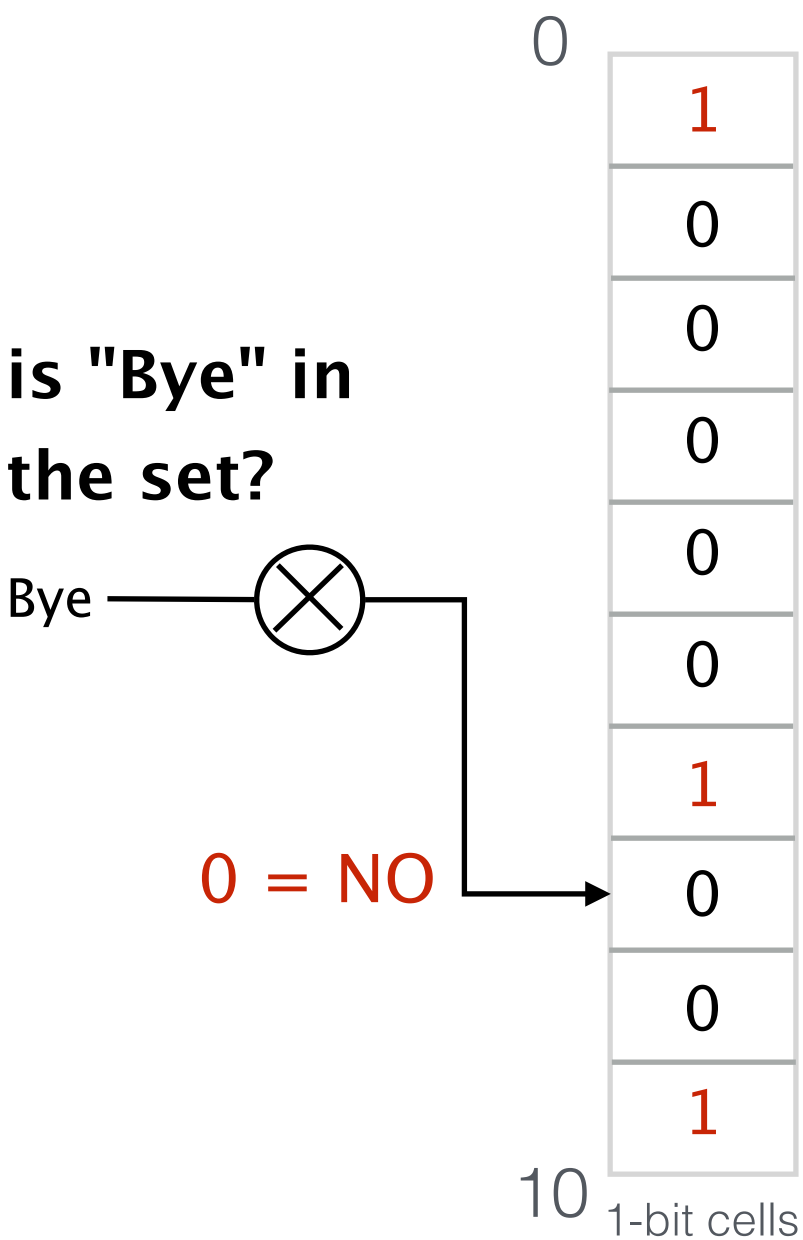
A simple approach for insertions and membership queries



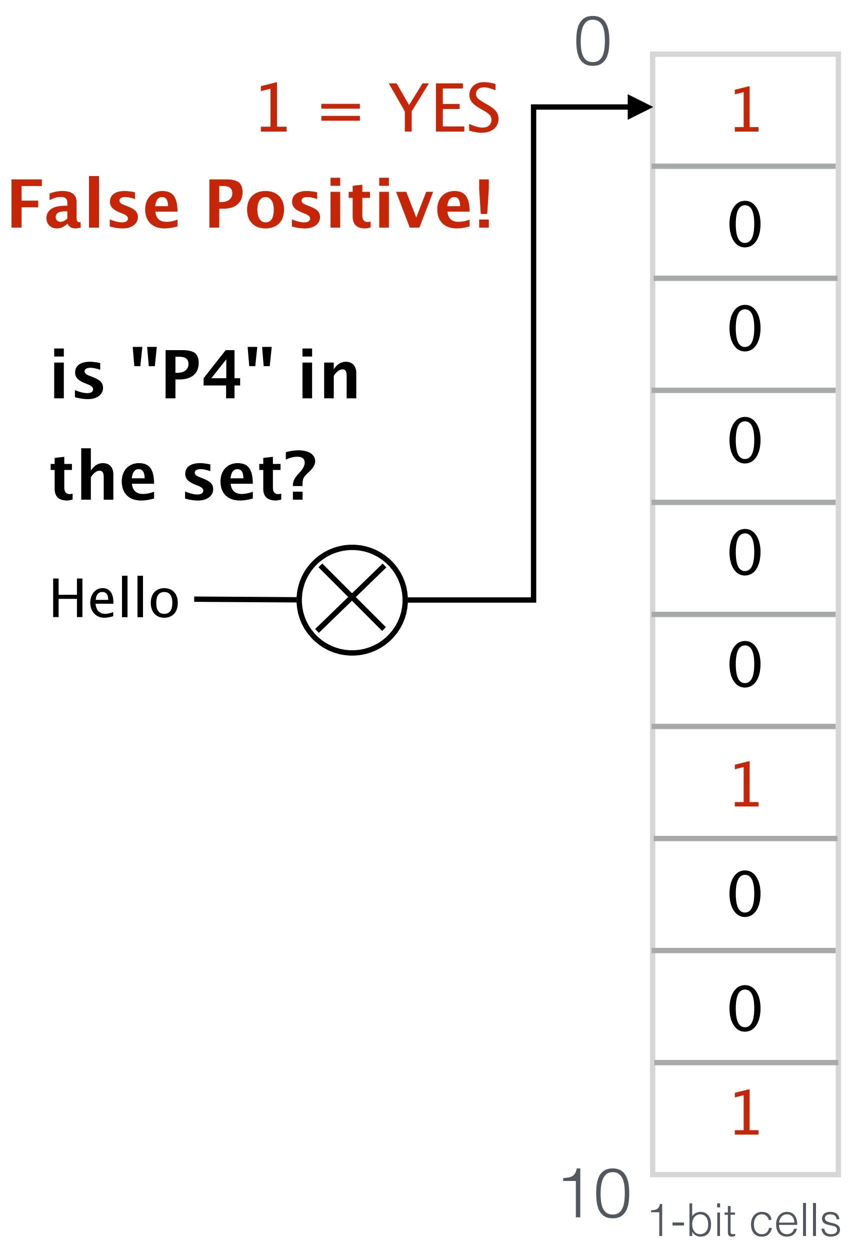
A simple approach for insertions and membership queries



A simple approach for insertions and membership queries



A simple approach for insertions and membership queries



A simple approach for insertions and membership queries

N elements and M cells

probability of an element to be mapped into a particular cell	$\frac{1}{M}$
probability of an element not to be mapped into a particular cell	$1 - \frac{1}{M}$
probability of a cell to be 0	$(1 - \frac{1}{M})^N$
false positive rate (FPR)	$1 - (1 - \frac{1}{M})^N$
false negative rate	0

A simple approach for insertions and membership queries

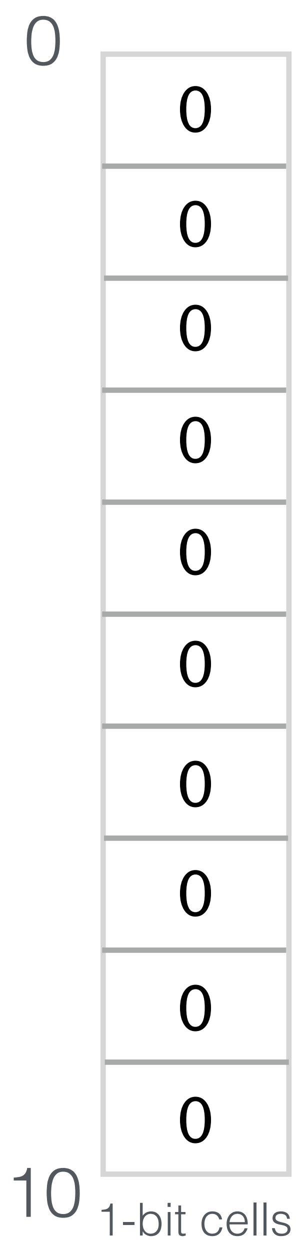
# of elements	# of cells	FPR
1000	10000	9.5%
1000	100000	1%

A simple approach for insertions and membership queries

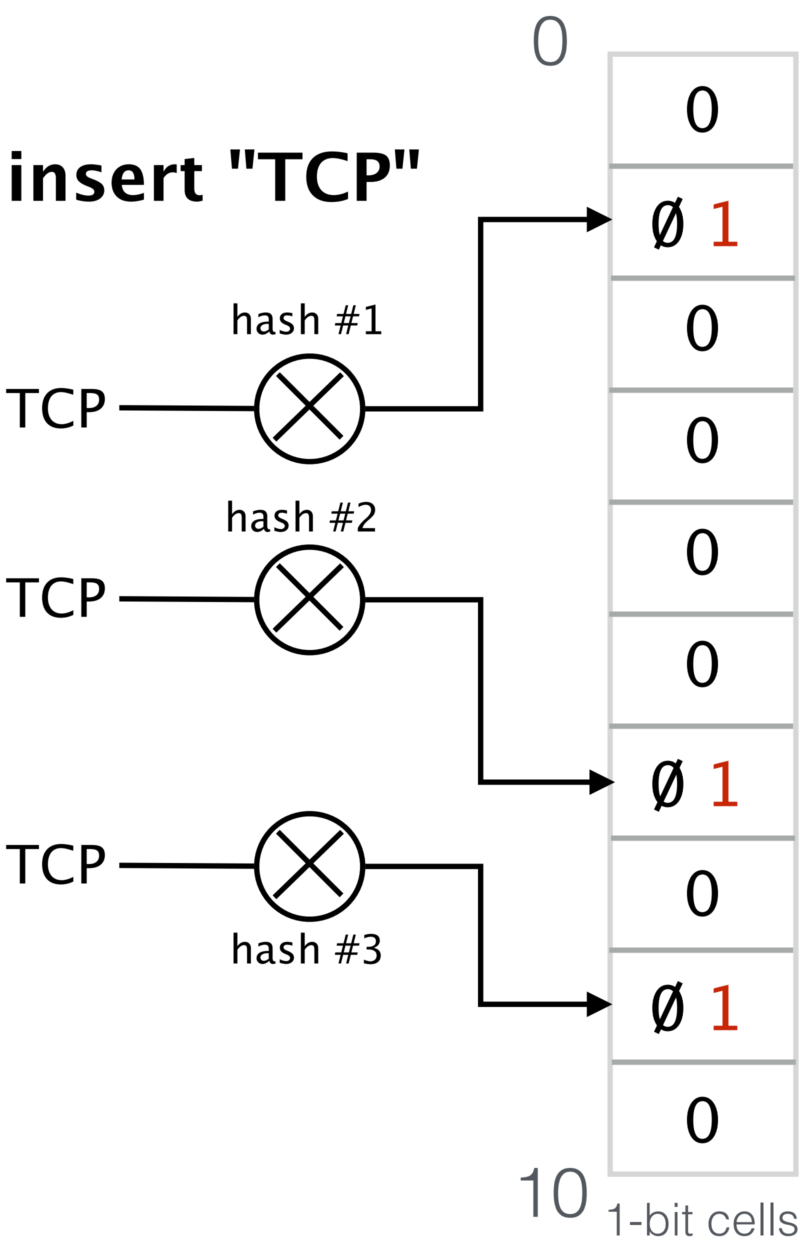
Pros: simple and only one operation per insertion or query

Con: roughly 100x more cells are required than the number of element we want to store for a 1% false positive rate

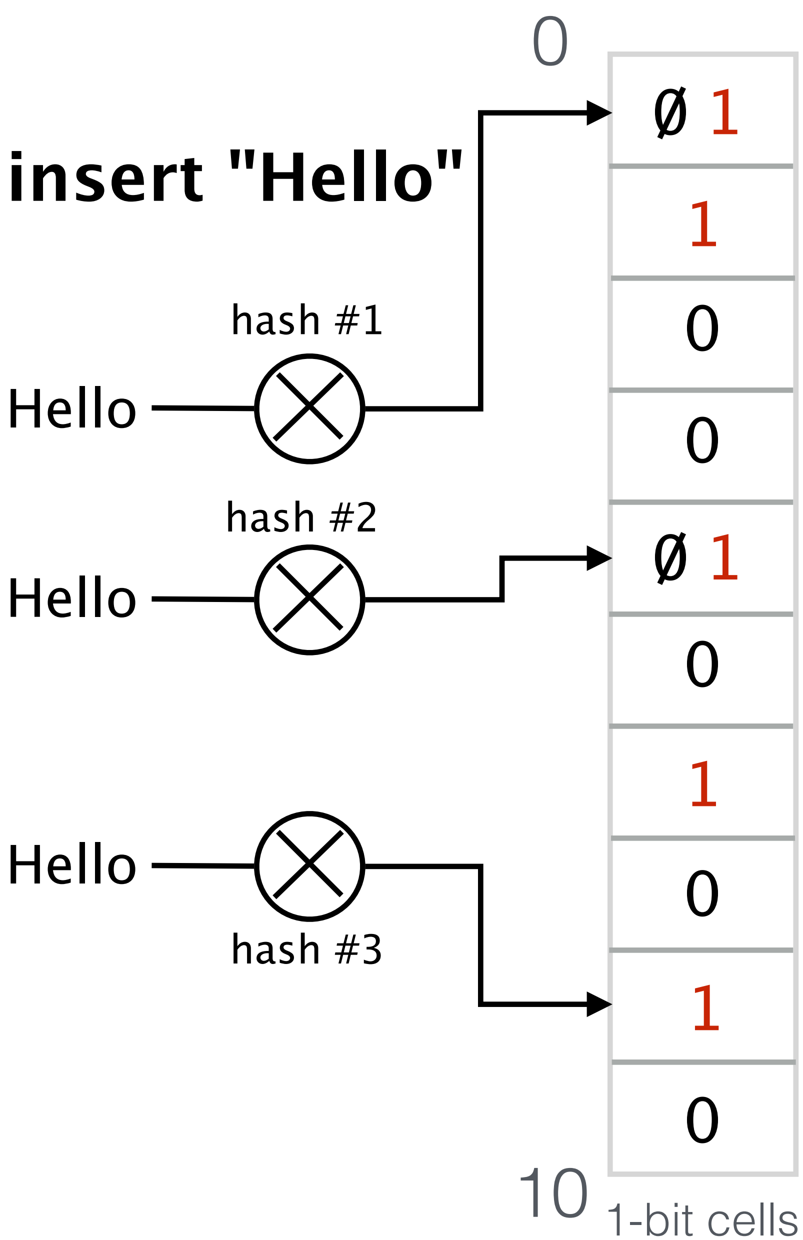
Bloom Filters: a more memory-efficient approach for insertions and membership queries



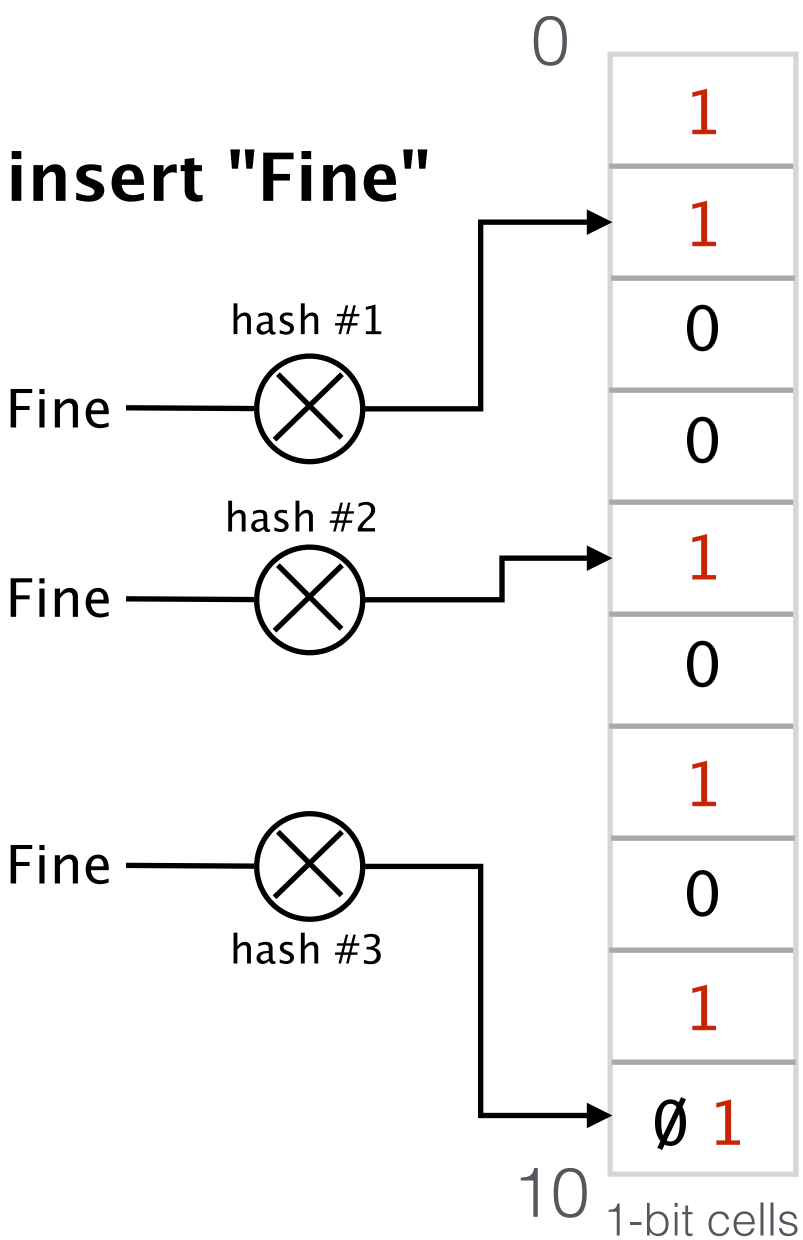
Bloom Filters: a more memory-efficient approach for insertions and membership queries



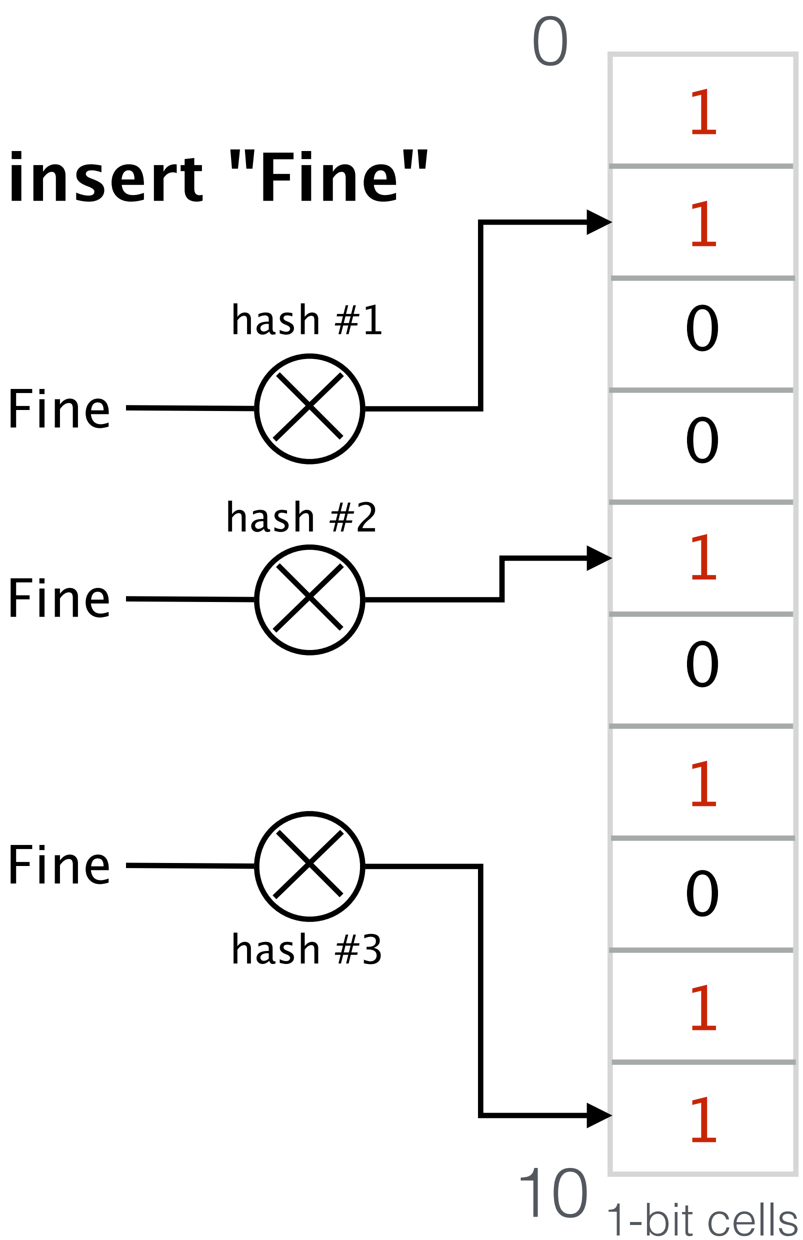
Bloom Filters: a more memory-efficient approach for insertions and membership queries



Bloom Filters: a more memory-efficient approach for insertions and membership queries



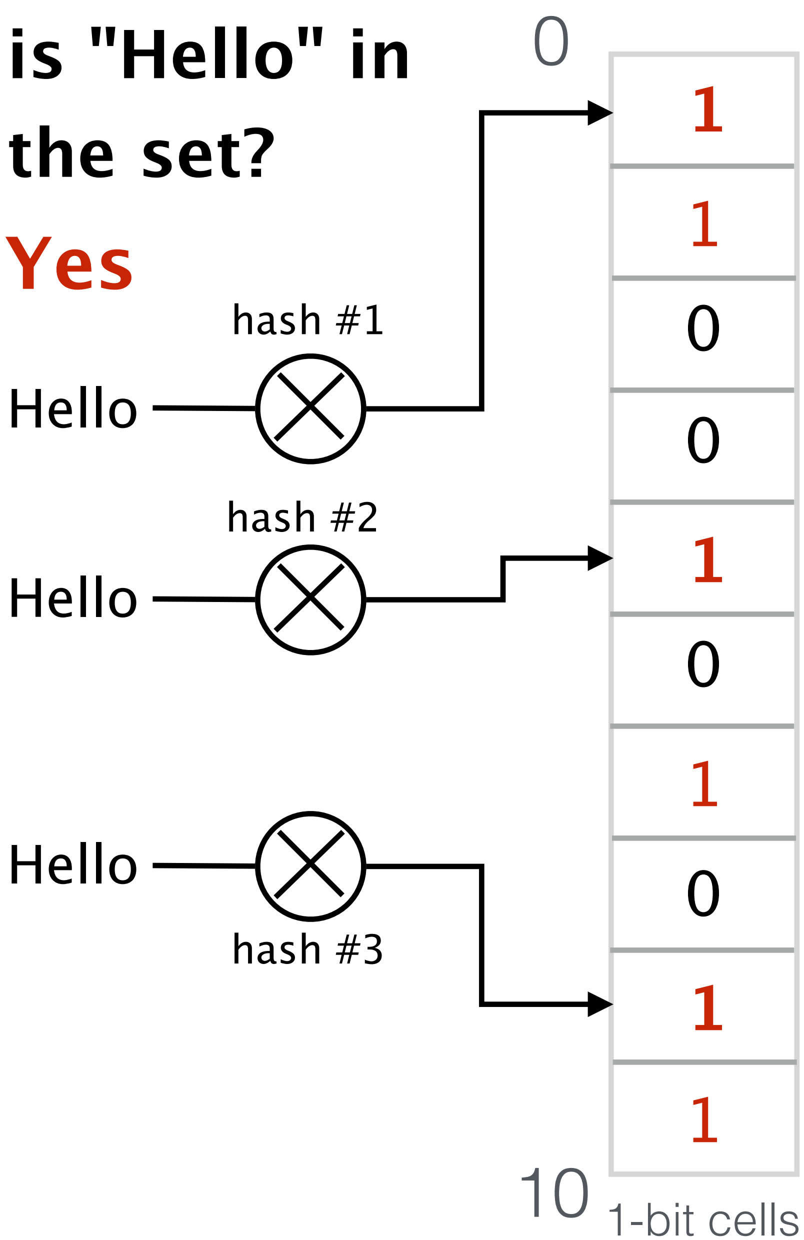
Bloom Filters: a more memory-efficient approach for insertions and membership queries



An element is considered in the set if **all** the hash values map to a cell with 1

An element is not in the set if **at least** one hash value maps to a cell with 0

Bloom Filters: a more memory-efficient approach for insertions and membership queries

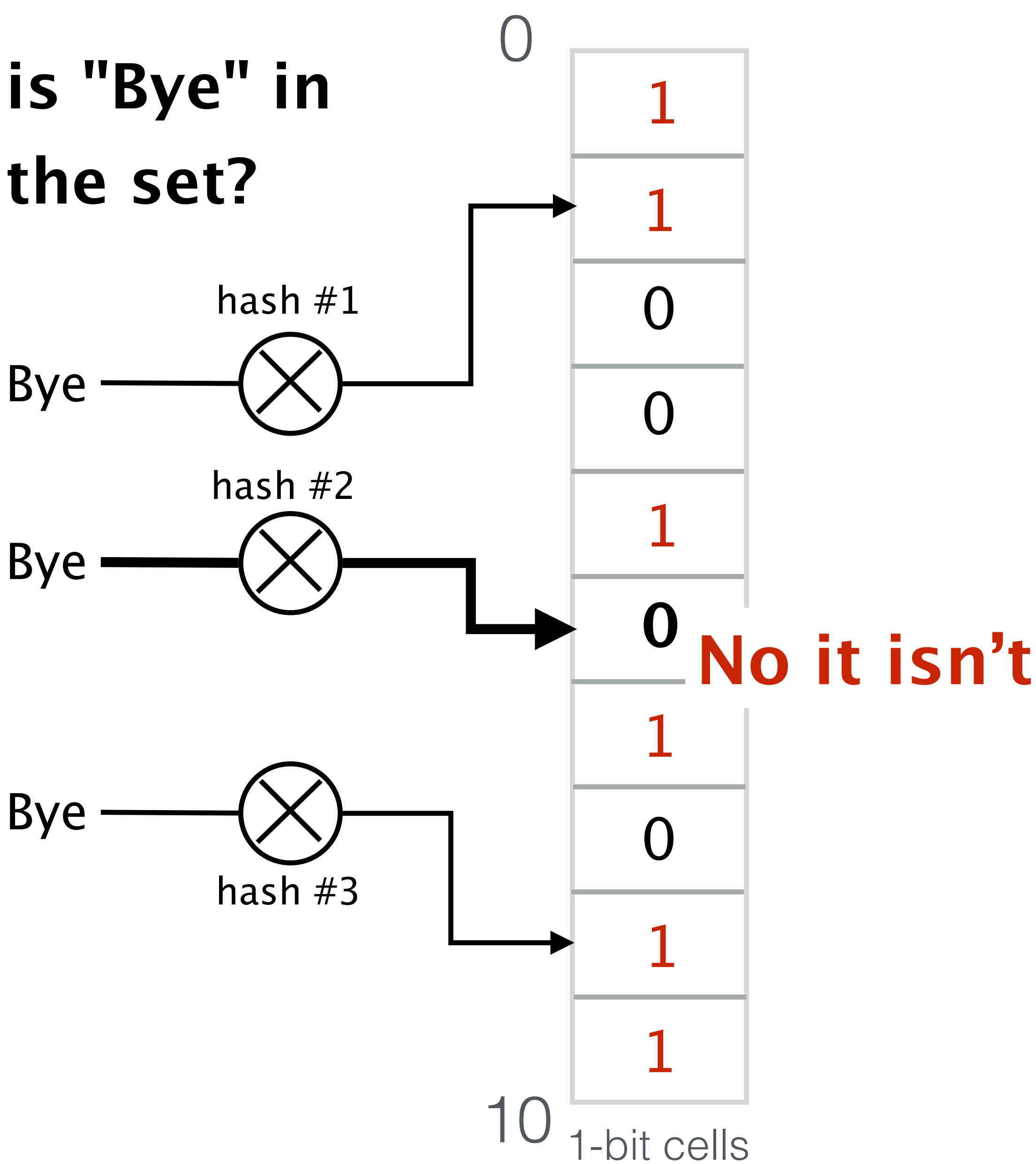


An element is considered in the set if **all** the hash values map to a cell with 1

An element is not in the set if **at least** one hash value maps to a cell with 0

Bloom Filters: a more memory-efficient approach for insertions and membership queries

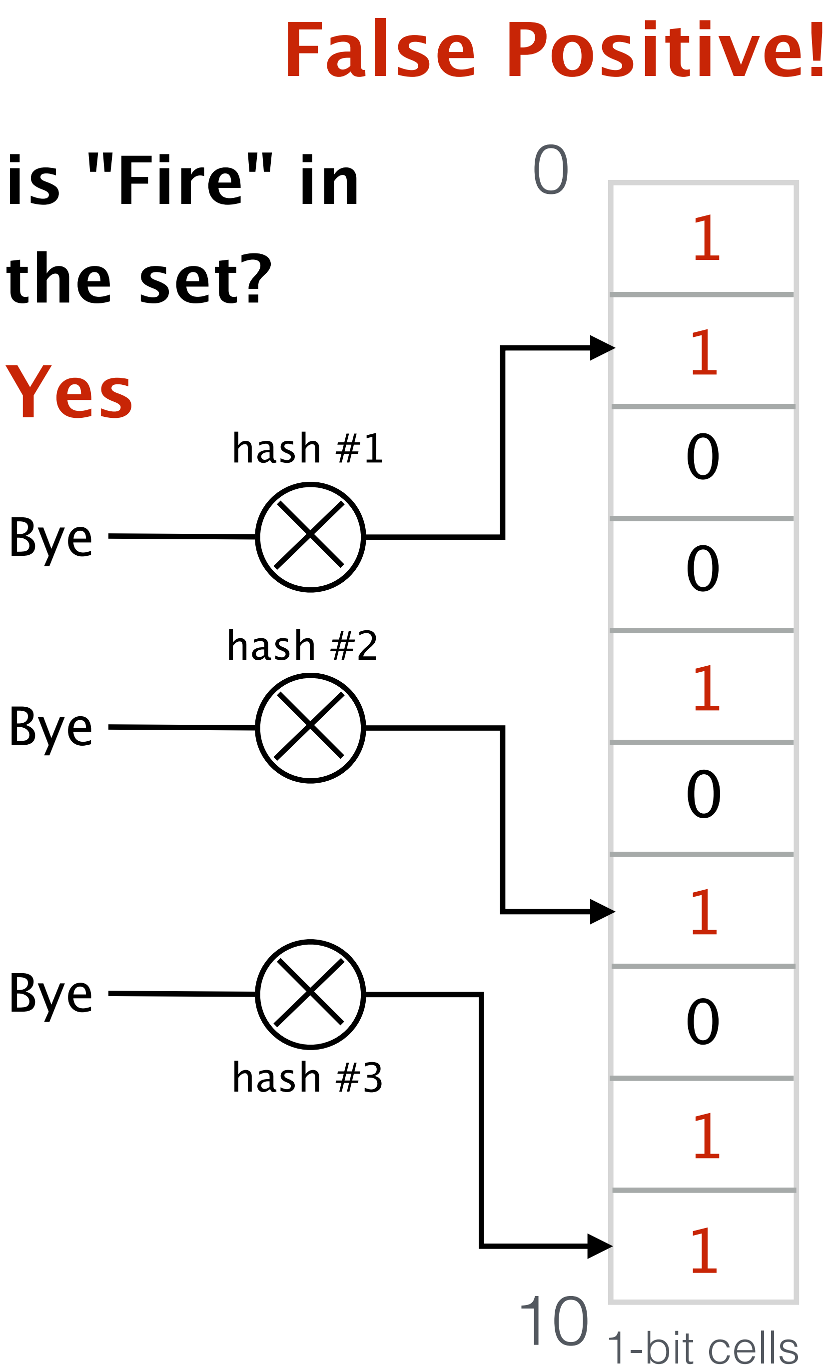
is "Bye" in the set?



An element is considered in the set if **all** the hash values map to a cell with 1

An element is not in the set if **at least** one hash value maps to a cell with 0

Bloom Filters: a more memory-efficient approach for insertions and membership queries



An element is considered in the set if **all** the hash values map to a cell with 1

An element is not in the set if **at least** one hash value maps to a cell with 0

Bloom Filters: a more memory-efficient approach for insertions and membership queries

N elements, M cells and K hash functions

probability of an element to be mapped into a particular cell	$\frac{1}{M}$
probability of an element not to be mapped into a particular cell	$1 - \frac{1}{M}$
probability of a cell to be 0	$(1 - \frac{1}{M})^{KN}$
false positive rate	$(1 - (1 - \frac{1}{M})^{KN})^{K}$
false negative rate	0

Bloom Filters: a more memory-efficient approach for insertions and membership queries

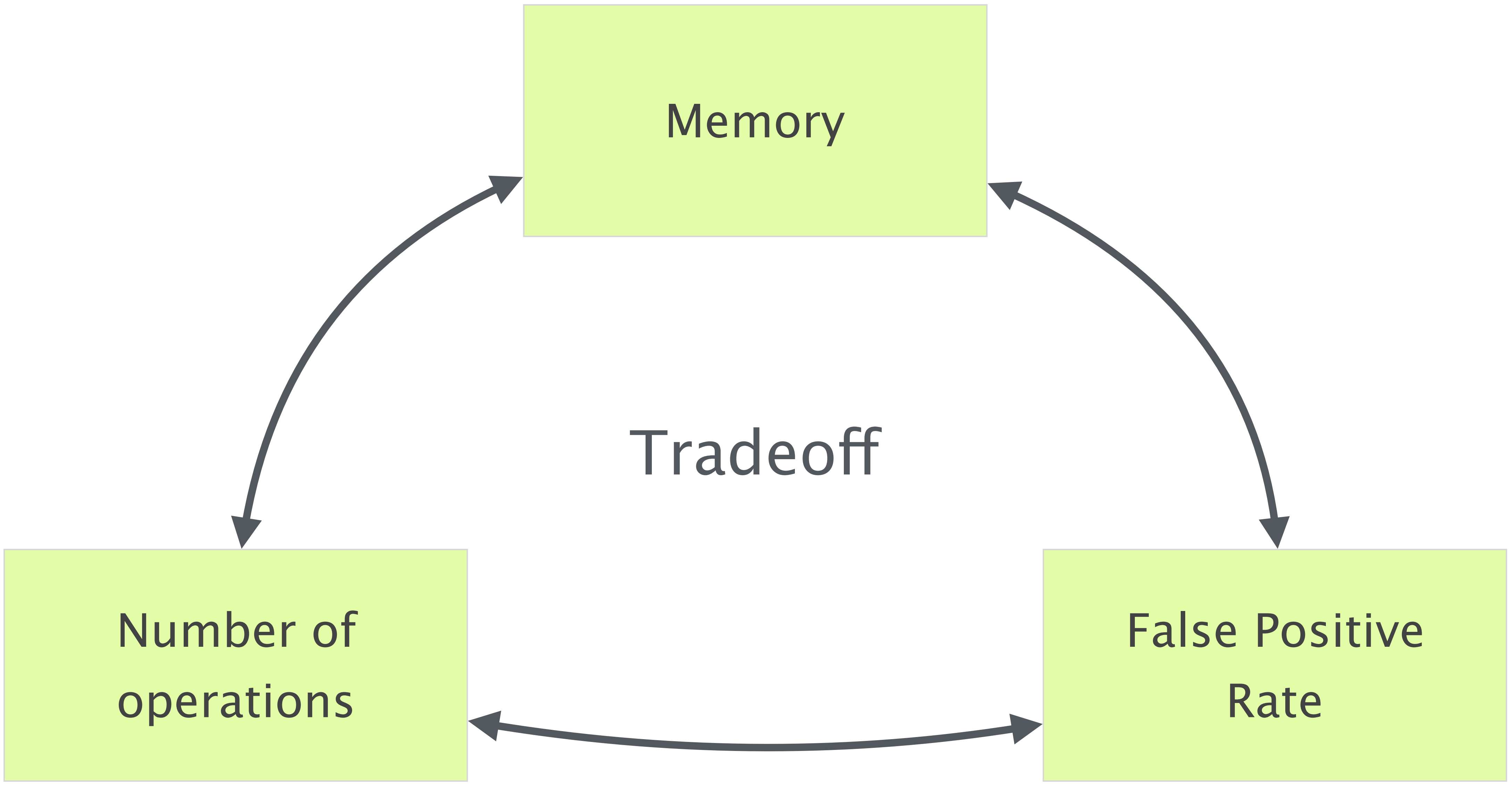
# of elements	# of cells	# hash functions	FPR
1000	10000	7	0.82%
1000	100000	7	$\approx$ 0%

Bloom Filters: a more memory-efficient approach for insertions and membership queries

Pro: consumes roughly 10x less memory than the simple approach

Con: Requires slightly more operations than the simple approach (7 hashes instead of just 1)

Dimension your Bloom Filter



Dimension your Bloom Filter

N elements

M cells

K hash functions

FP false positive rate

Dimension your Bloom Filter

N elements

M cells

K hash functions

FP false positive rate

asymptotic approx.

$$FP = (1 - (1 - \frac{1}{M})^{KN})^K \approx (1 - e^{-KN/M})^K$$

with calculus you can dimension your bloom filter

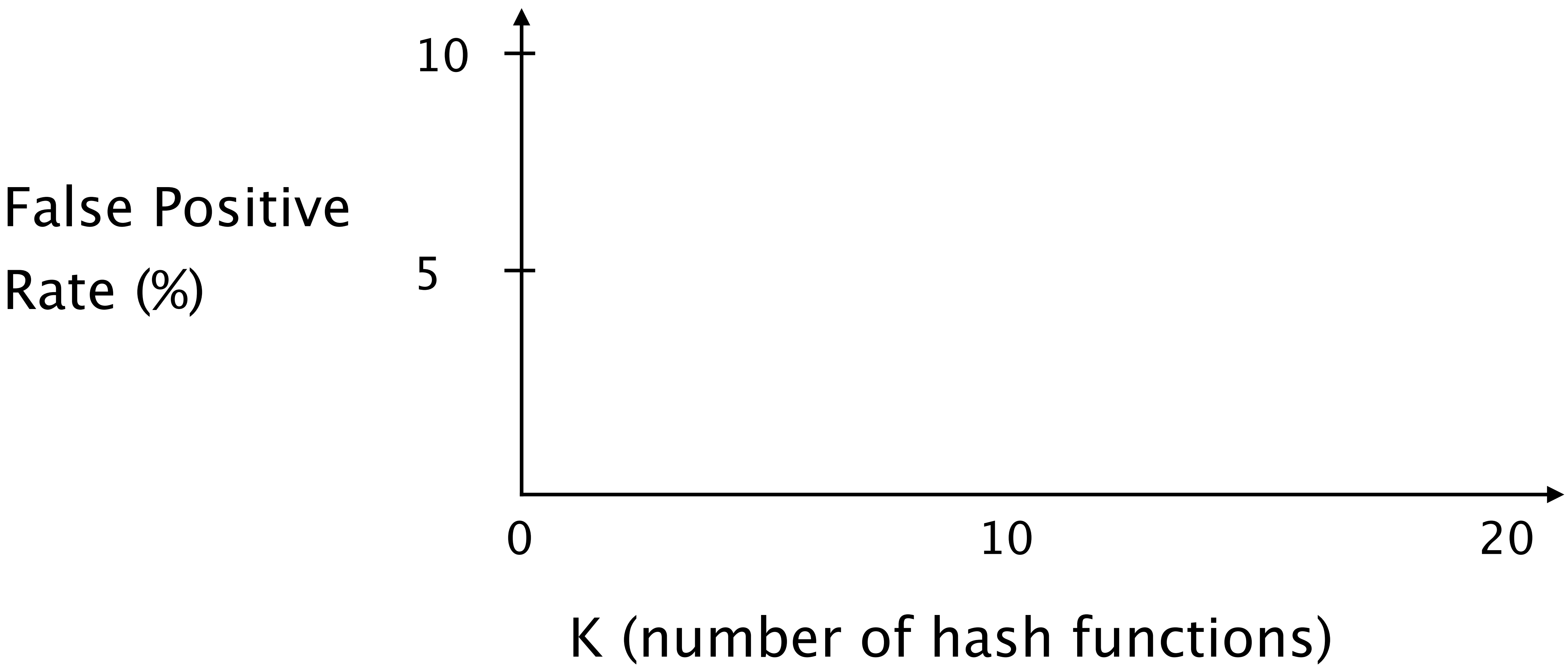
Dimension your Bloom Filter

N = 1000

M = 10000

K hash functions

FP false positive rate



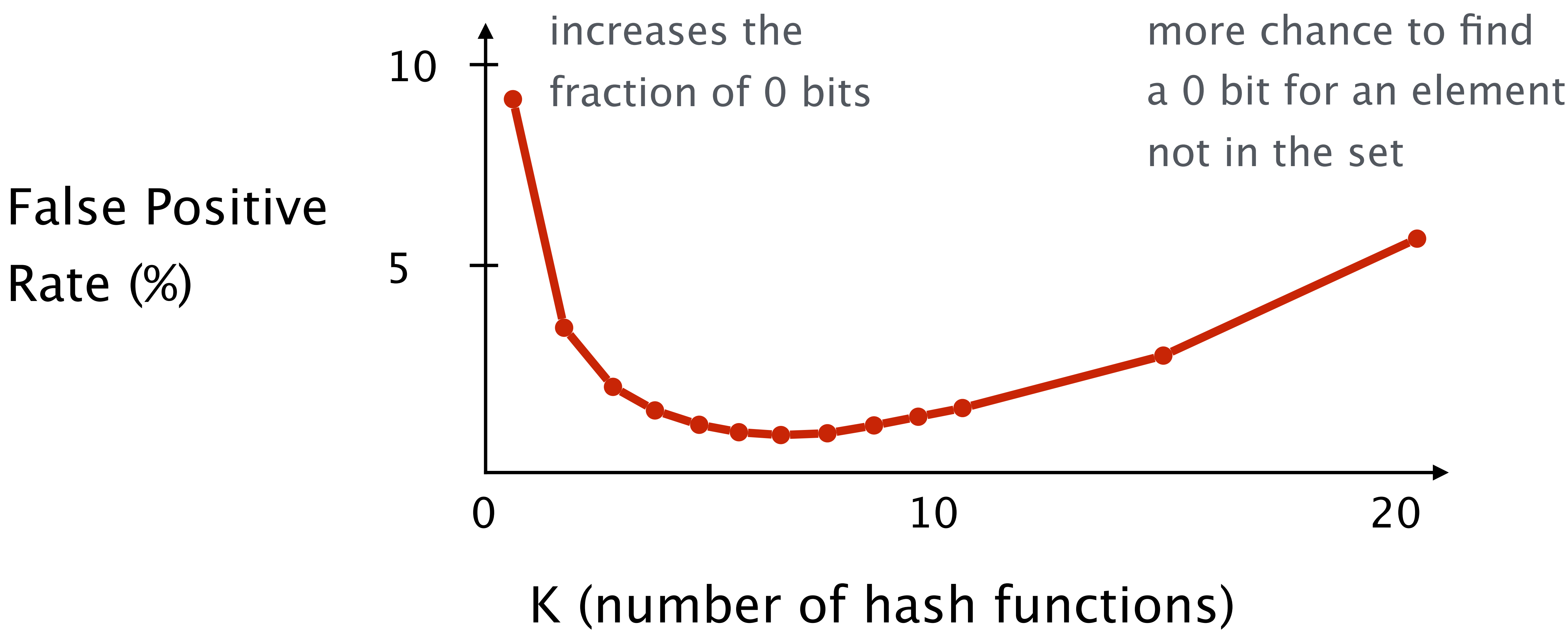
Dimension your Bloom Filter

N = 1000

M = 10000

K hash functions

FP false positive rate



Dimension your Bloom Filter

N = 1000

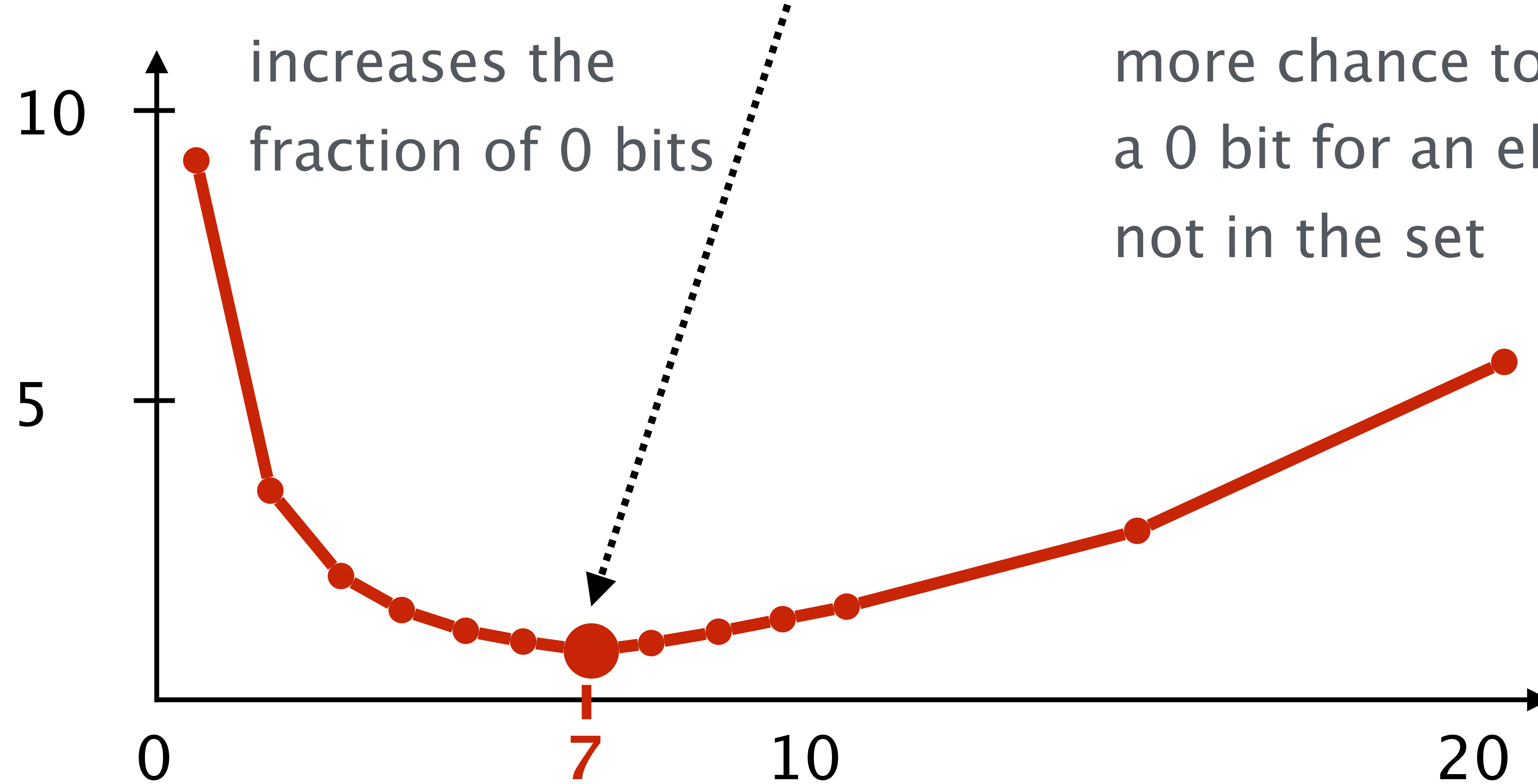
M = 10000

K hash functions

FP false positive rate

there is always a global minimum when $K = \ln 2 * (M/N)$ found by taking the derivative of $\approx (1 - e^{-KN/M})^K$

False Positive Rate (%)



K (number of hash functions)

Implementation of a Bloom Filter in $P4_{16}$

You will have to use hash functions

v1model

```
enum HashAlgorithm {
    crc32,
    crc32_custom,
    crc16,
    s,
    random,
    identity,
    csum16,
    xor16
}
```

```
extern void hash<O, T, D, M>(out O result,
  in HashAlgorithm algo, in T base, in D data, in M max);
```

more info https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4

Implementation of a Bloom Filter in $P4_{16}$

You will have to use hash functions, as well as registers

v1model

```
extern register<T> {

    register(bit<32> size);

    void read(out T result, in bit<32> index);
    void write(in bit<32> index, in T value);
}
```

more info https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4

Implementation of a Bloom Filter in $P4_{16}$

with 2 hash functions

```
control MyIngress(…) {

  register register<bit<1>>(NB_CELLS) bloom_filter;
```

Implementation of a Bloom Filter in $P4_{16}$

with 2 hash functions

```
control MyIngress(…) {

  register register<bit<1>>(NB_CELLS) bloom_filter;

   apply {
      hash(meta.index1, HashAlgorithm.my_hash1, 0,
        {meta.dstPrefix, packet.ip.srcIP}, NB_CELLS);
      hash(meta.index2, HashAlgorithm.my_hash2, 0,
        {meta.dstPrefix, packet.ip.srcIP}, NB_CELLS);
```

Implementation of a Bloom Filter in $P4_{16}$

with 2 hash functions

```
control MyIngress(…) {

  register register<bit<1>>(NB_CELLS) bloom_filter;

   apply {
      hash(meta.index1, HashAlgorithm.my_hash1, 0,
        {meta.dstPrefix, packet.ip.srcIP}, NB_CELLS);
      hash(meta.index2, HashAlgorithm.my_hash2, 0,
        {meta.dstPrefix, packet.ip.srcIP}, NB_CELLS);

      if (meta.to_insert == 1) {
        bloom_filter.write(meta.index1, 1);
        bloom_filter.write(meta.index2, 1);
      }

      if (meta.to_query == 1) {
        bloom_filter.read(meta.query1, meta.index1);
        bloom_filter.read(meta.query2, meta.index2);

        if (meta.query1 == 0 || meta.query2 == 0) {
          meta.is_stored = 0;
        }
        else {
          meta.is_stored = 1;
        }
      }
   }
}
```

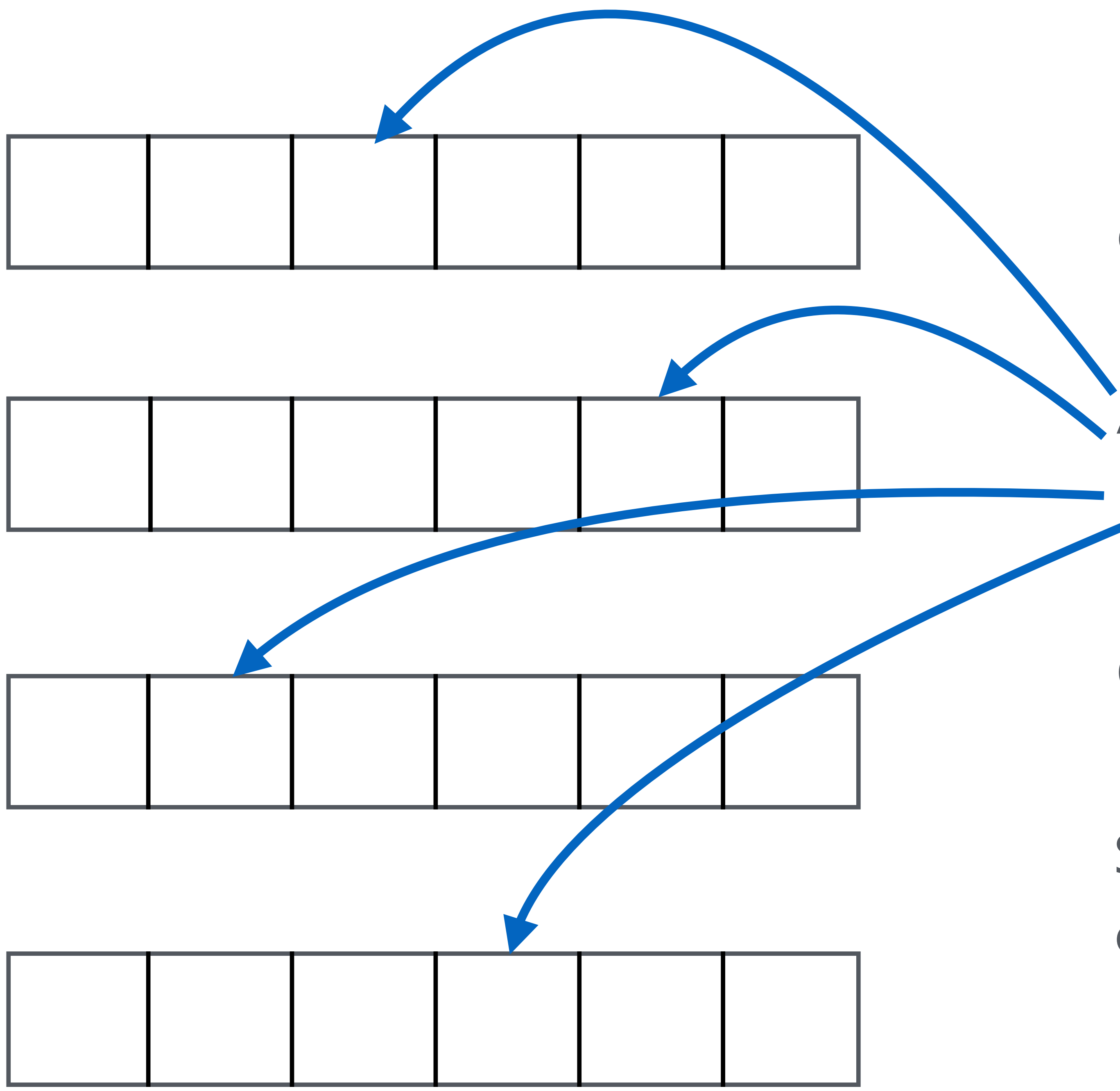
Implementation of a Bloom Filter in P4$_{16}$

with 2 hash functions

```
control MyIngress(…) {

  register register<bit<1>>(NB_CELLS) bloom_filter;
   apply {
     hash(meta.index1, HashAlgorithm.my_hash1, 0,
       {meta.dstPrefix, packet.ip.srcIP}, NB_CELLS);
     hash(meta.index2, HashAlgorithm.my_hash2, 0,
       {meta.dstPrefix, packet.ip.srcIP}, NB_CELLS);

     if (meta.to_insert == 1) {
       bloom_filter.write(meta.index1, 1);
       bloom_filter.write(meta.index2, 1);
     }

     if (meta.to_query == 1) {
       bloom_filter.read(meta.query1, meta.index1);
       bloom_filter.read(meta.query2, meta.index2);

       if (meta.query1 == 0 || meta.query2 == 0) {
         meta.is_stored = 0;
       }
       else {
         meta.is_stored = 1;
       }
     }
  }
}
```

Everything in bold red must be adapted for your program

Depending on the hardware limitations, splitting the bloom filter might be required



M cells are split into M/K disjoint groups

An element is hashed to K cells, one in each group

One hash function per group

Same performance, may be easier to implement or parallelize

Because deletions are not possible, the controller may need to regularly reset the bloom filters

Resetting a bloom filter takes some time during which it is not usable

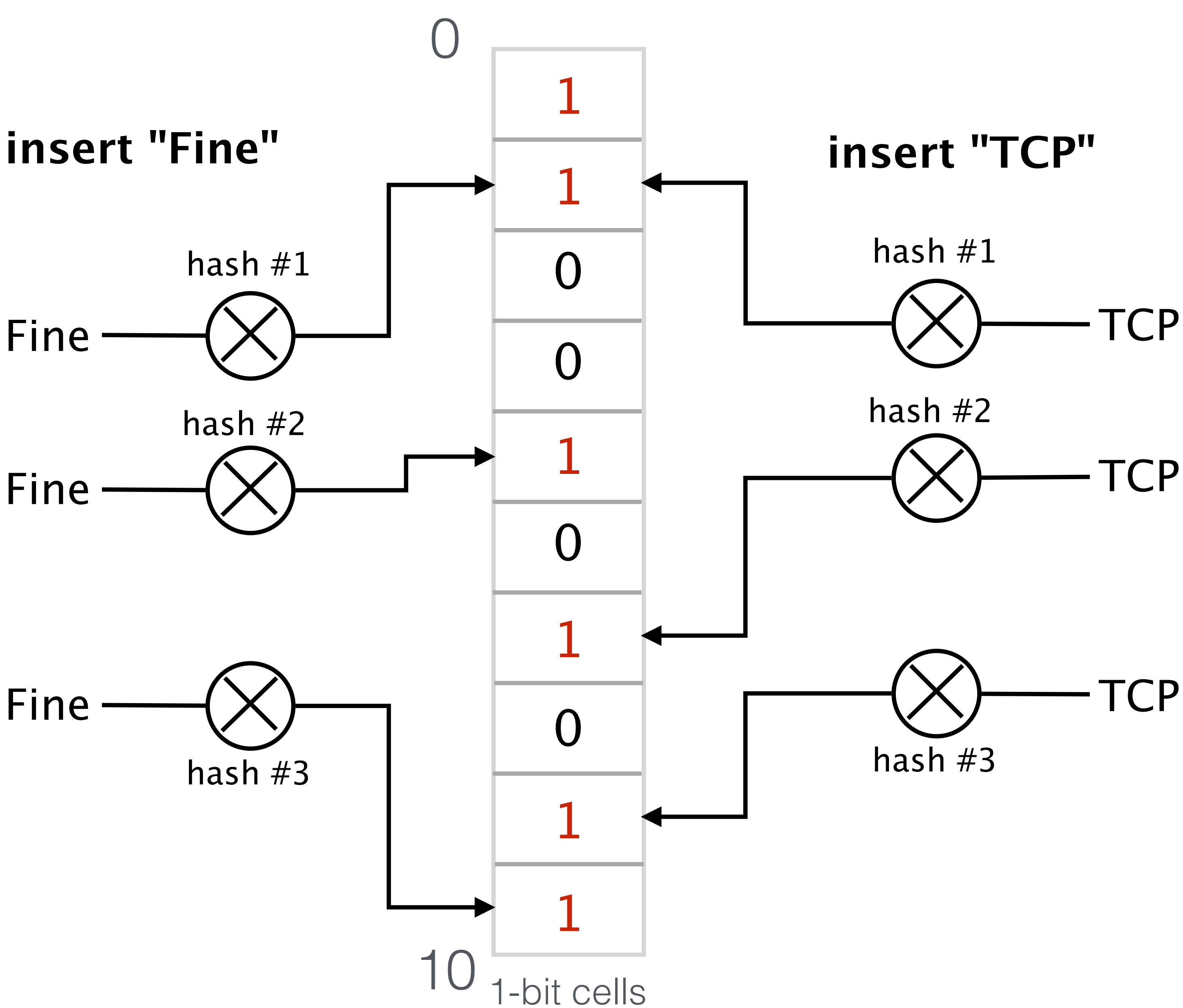
Common trick: use two bloom filters and use one when the controller resets the other one

So far we have seen how to do insertions and membership queries

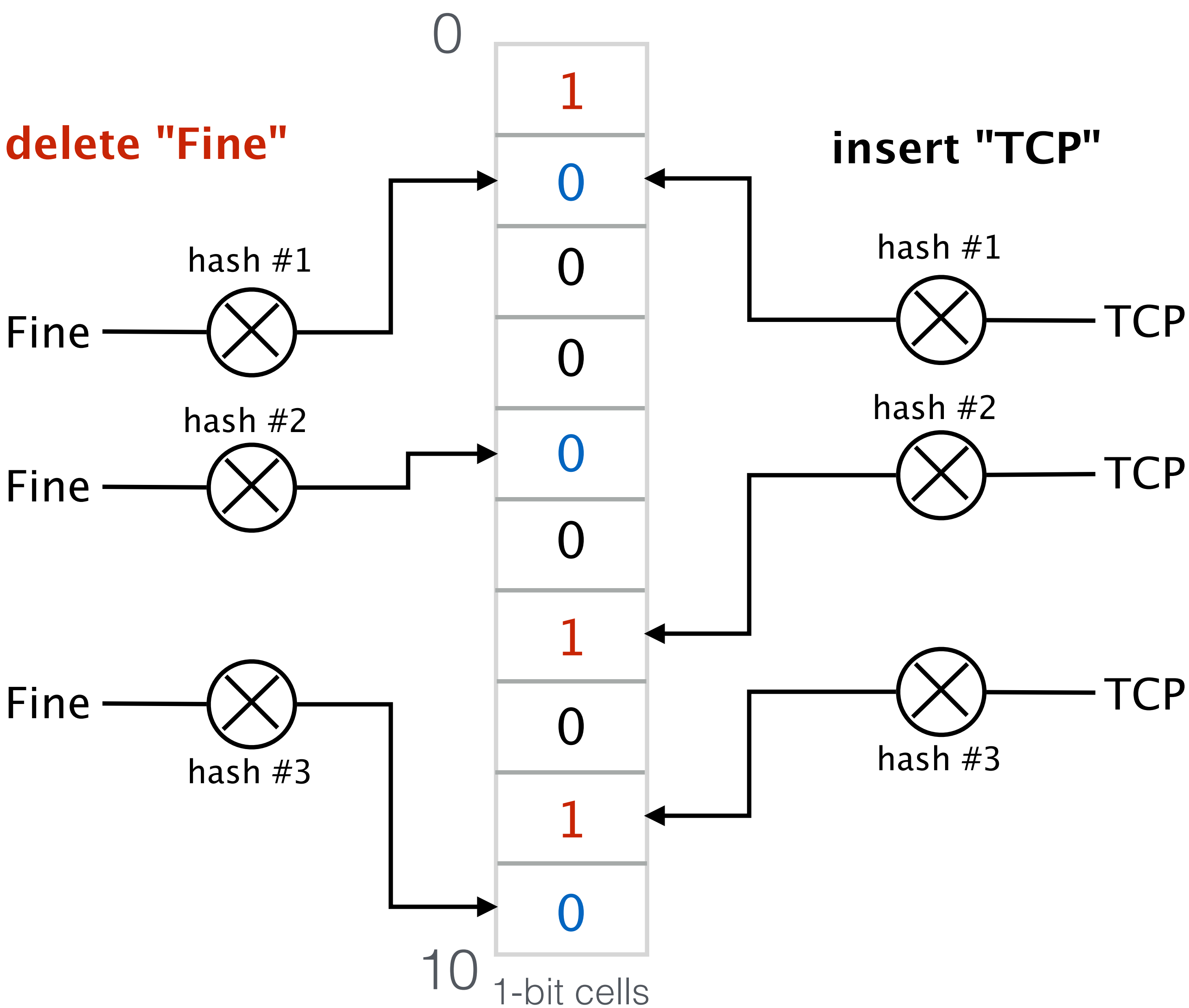
	strategy #1	strategy #2
output	Deterministic	Probabilistic
number of required operations	Probabilistic	Deterministic

Bloom Filters

However Bloom Filters do not handle deletions



However Bloom Filters do not handle deletions



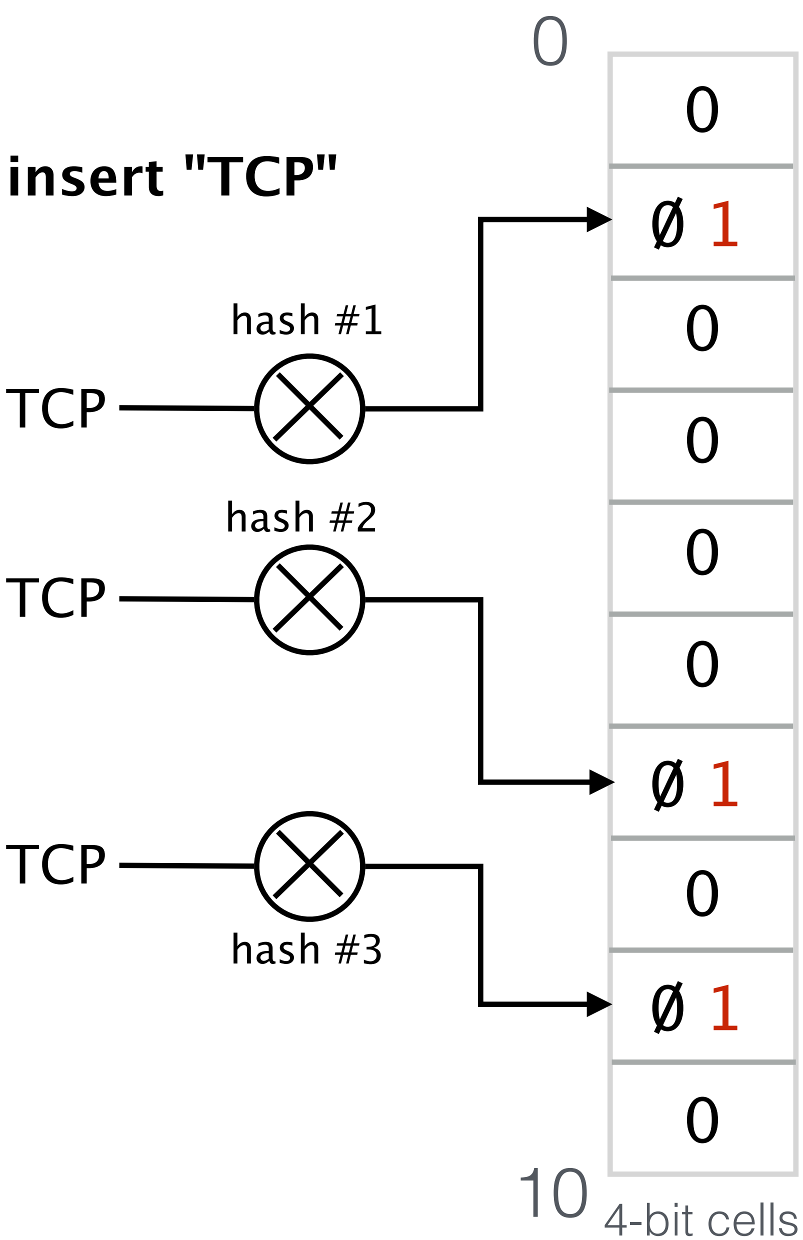
If deleting an element means resetting 1s to 0s, then deleting "Fine" also deletes "TCP"

But we can easily extend them to handle deletions

This extended version is called a Counting Bloom Filter

But we can easily extend them to handle deletions

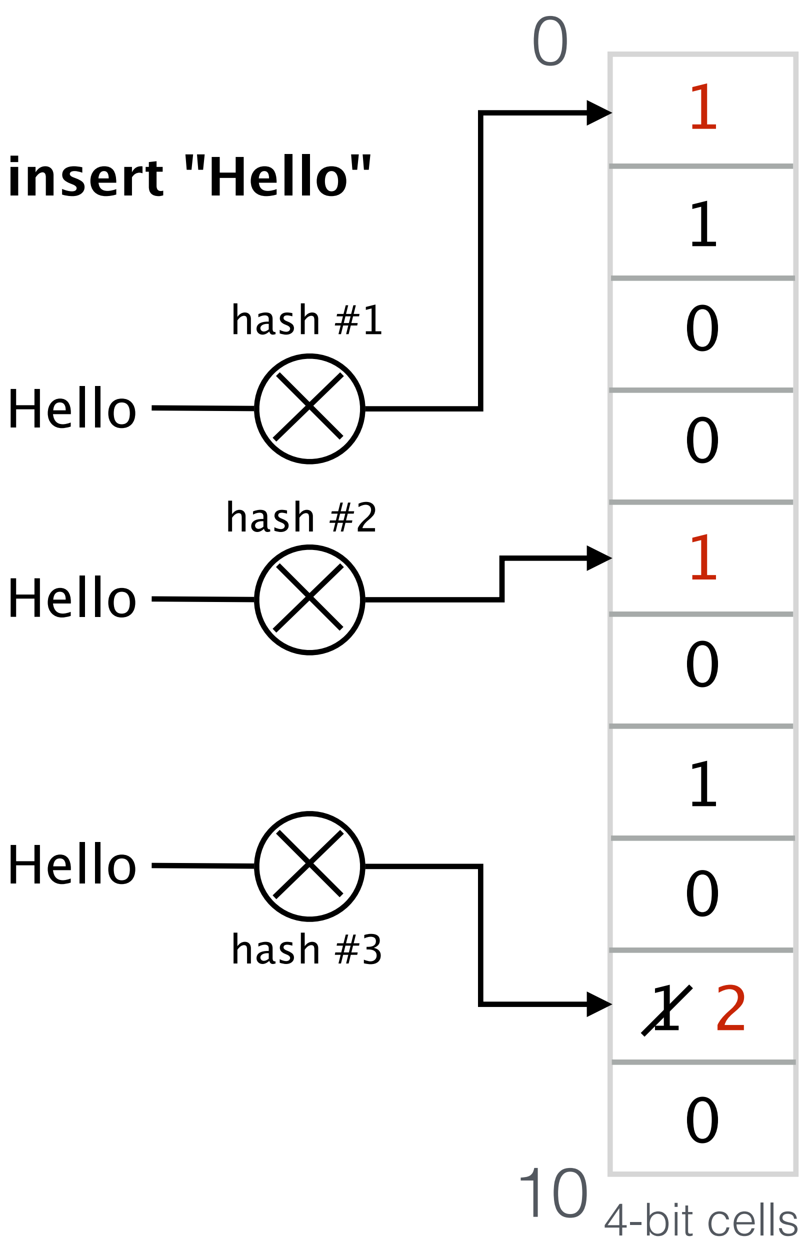
This extended version is called a Counting Bloom Filter



To add an element, increment the corresponding counters

But we can easily extend them to handle deletions

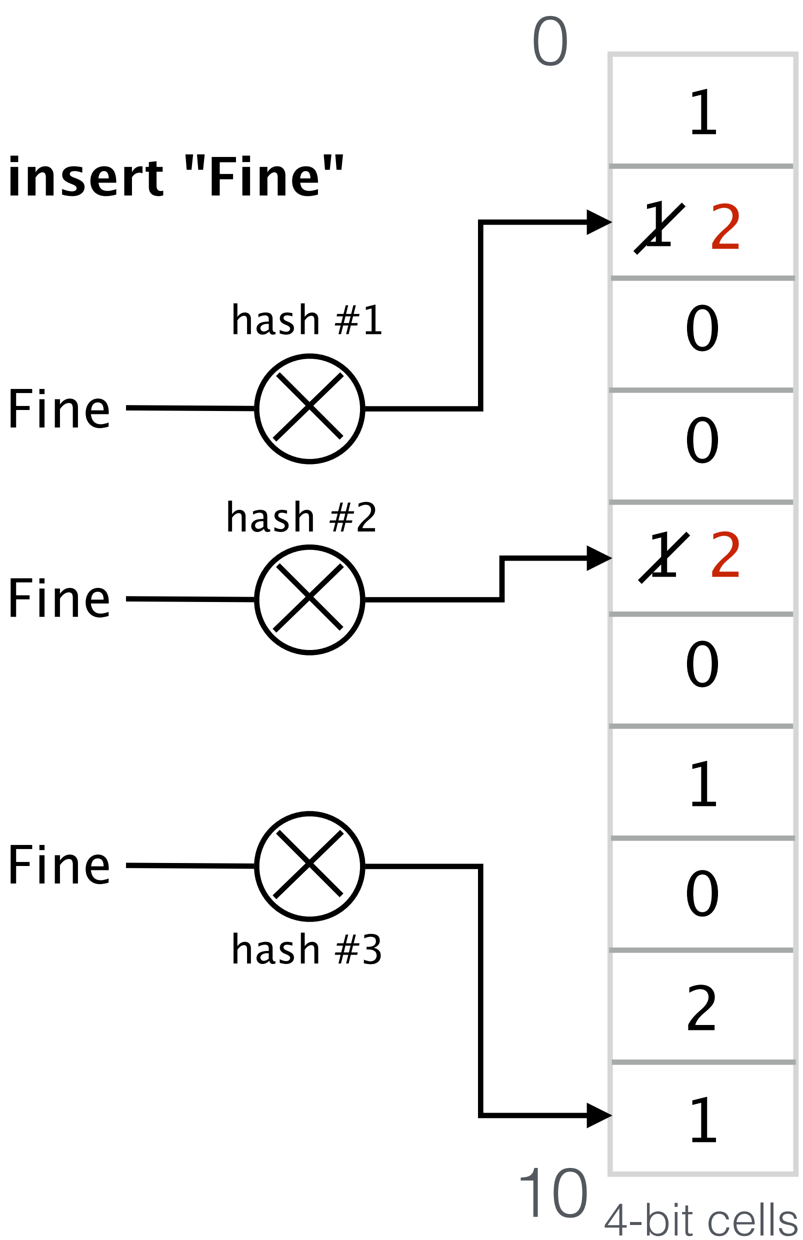
This extended version is called a Counting Bloom Filter



To add an element, increment the corresponding counters

But we can easily extend them to handle deletions

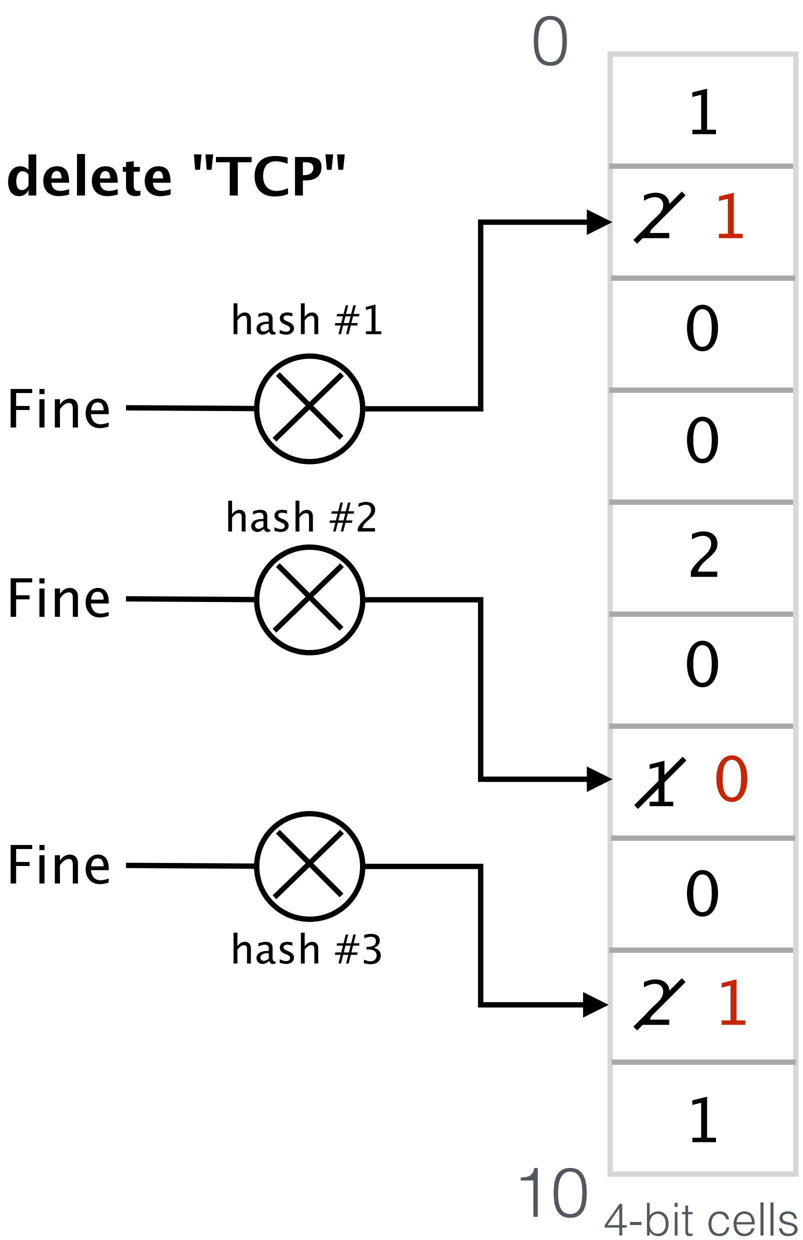
This extended version is called a Counting Bloom Filter



To add an element, increment the corresponding counters

But we can easily extend them to handle deletions

This extended version is called a Counting Bloom Filter

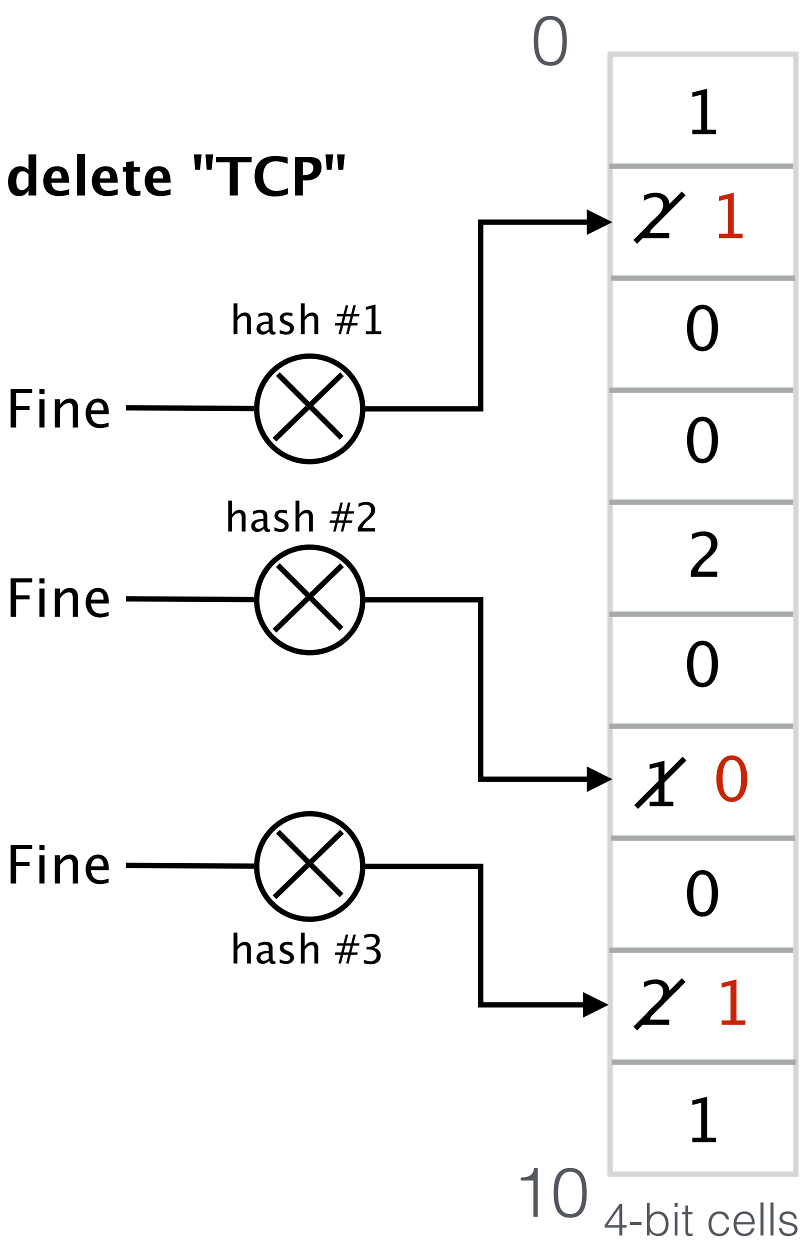


To add an element, increment the corresponding counters

To delete an element, decrement the corresponding counters

But we can easily extend them to handle deletions

This extended version is called a Counting Bloom Filter



To add an element, increment the corresponding counters

To delete an element, decrement the corresponding counters

All of our prior analysis for standard bloom filters applies to counting bloom filters

Counting Bloom Filters do handle deletions
at the price of using more memory

Counting Bloom Filters do handle deletions
at the price of using more memory

Counters must be large enough to avoid overflow

If a counter eventually overflows, the filter may yield false negatives

Counting Bloom Filters do handle deletions
at the price of using more memory

Counters must be large enough to avoid overflow
If a counter eventually overflows, the filter may yield false negatives

Poisson approximation suggests 4 bits/counter
The average load (i.e., $\frac{NK}{M}$) is $\ln 2$ assuming $K = \ln 2 * (M/N)$
With N=10000 and M=80000 the probability that some counter overflows if we use b-bit counters is at most
$M * Pr(Poisson(\ln 2) \geq 2^b) = 1.78\text{e}{-11}$

Implementation of a Counting Bloom Filter in $P4_{16}$

with 2 hash functions

Add a new element

```
control MyIngress(…) {

  register register<bit<4>>(NB_CELLS) bloom_filter;

   apply {
      hash(meta.index1, HashAlgorithm.my_hash1, 0,
        {meta.dstPrefix, packet.ip.srcIP}, NB_CELLS);
      hash(meta.index2, HashAlgorithm.my_hash2, 0,
        {meta.dstPrefix, packet.ip.srcIP}, NB_CELLS);

      // Add a new element if not yet in the set
      bloom_filter.read(meta.query1, meta.index1);
      bloom_filter.read(meta.query2, meta.index2);

      if (meta.query1 == 0 || meta.query2 == 0) {
        bloom_filter.write(meta.index1, meta.query1 + 1);
        bloom_filter.write(meta.index2, meta.query2 + 1);
      }
  }
}
```

Implementation of a Counting Bloom Filter in $P4_{16}$

with 2 hash functions

Delete an element

```
control MyIngress(…) {

  register register<bit<32>>(NB_CELLS) bloom_filter;

   apply {
      hash(meta.index1, HashAlgorithm.my_hash1, 0,
        {meta.dstPrefix, packet.ip.srcIP}, NB_CELLS);
      hash(meta.index2, HashAlgorithm.my_hash2, 0,
        {meta.dstPrefix, packet.ip.srcIP}, NB_CELLS);

      // Delete a element only if it is in the set
      bloom_filter.read(meta.query1, meta.index1);
      bloom_filter.read(meta.query2, meta.index2);

      if (meta.query1 > 0 && meta.query2 > 0) {
        bloom_filter.write(meta.index1, meta.query1 - 1);
        bloom_filter.write(meta.index2, meta.query2 - 1);
      }
   }
}
```

So far we have seen how to do insertions, deletions and membership queries

	strategy #1	strategy #2
output	Deterministic	Probabilistic
number of required operations	Probabilistic	Deterministic

Bloom Filters
Counting Bloom Filters

Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing

Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing

Each cell contains three fields

count which counts the number of entries mapped to this cell

keySum which is the sum of all the keys mapped to this cell

valueSum which is the sum of all the values mapped to this cell

Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing

count	keySum	valueSum
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0

Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing

Add a new key-value pair (assuming it is not in the set yet)

For each hash function
  hash the key to find the index

  Then at this index
    increment the count by one
    add key to keySum
    add value to valueSum

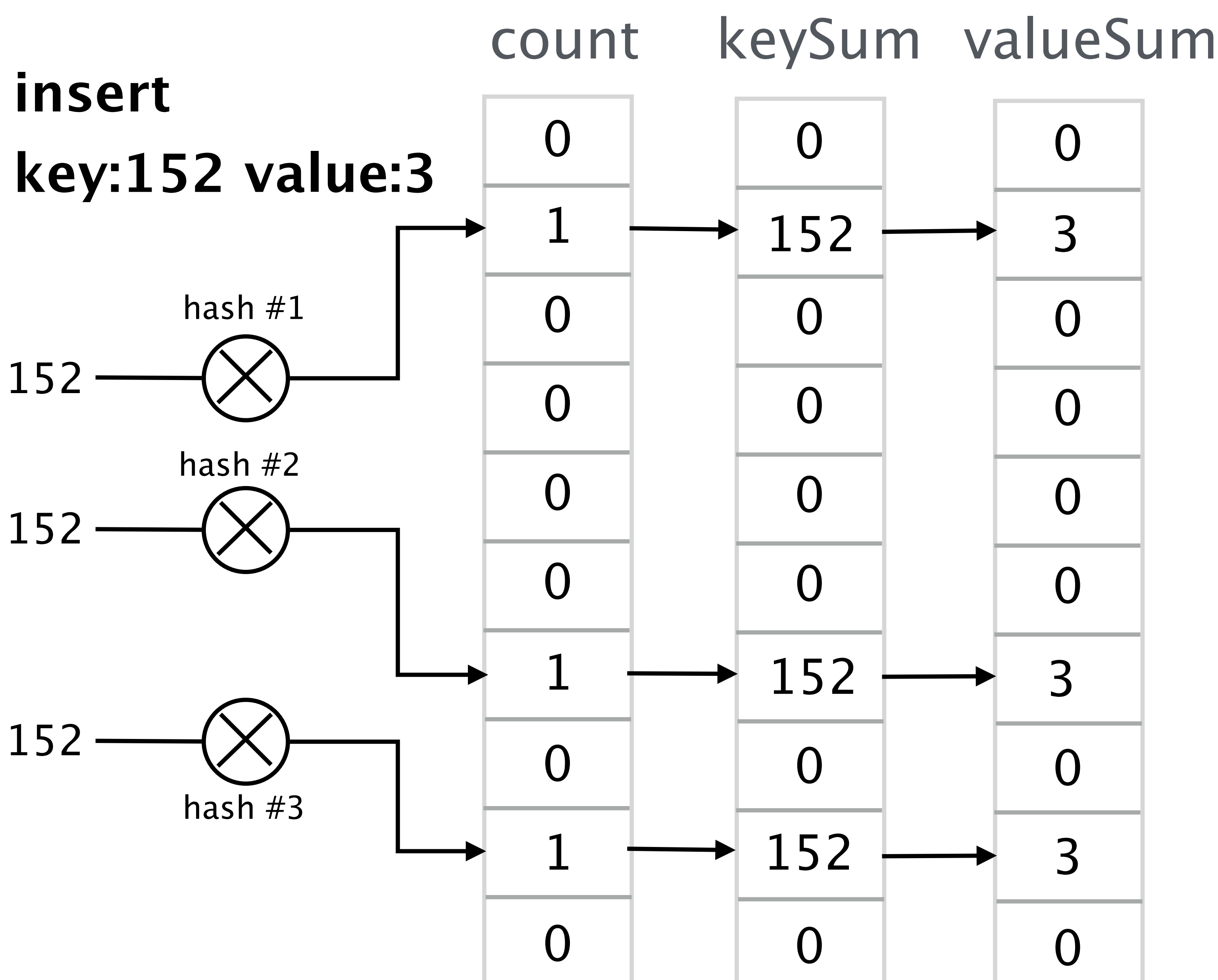
Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing

Delete a key-value pair (assuming it is in the set)

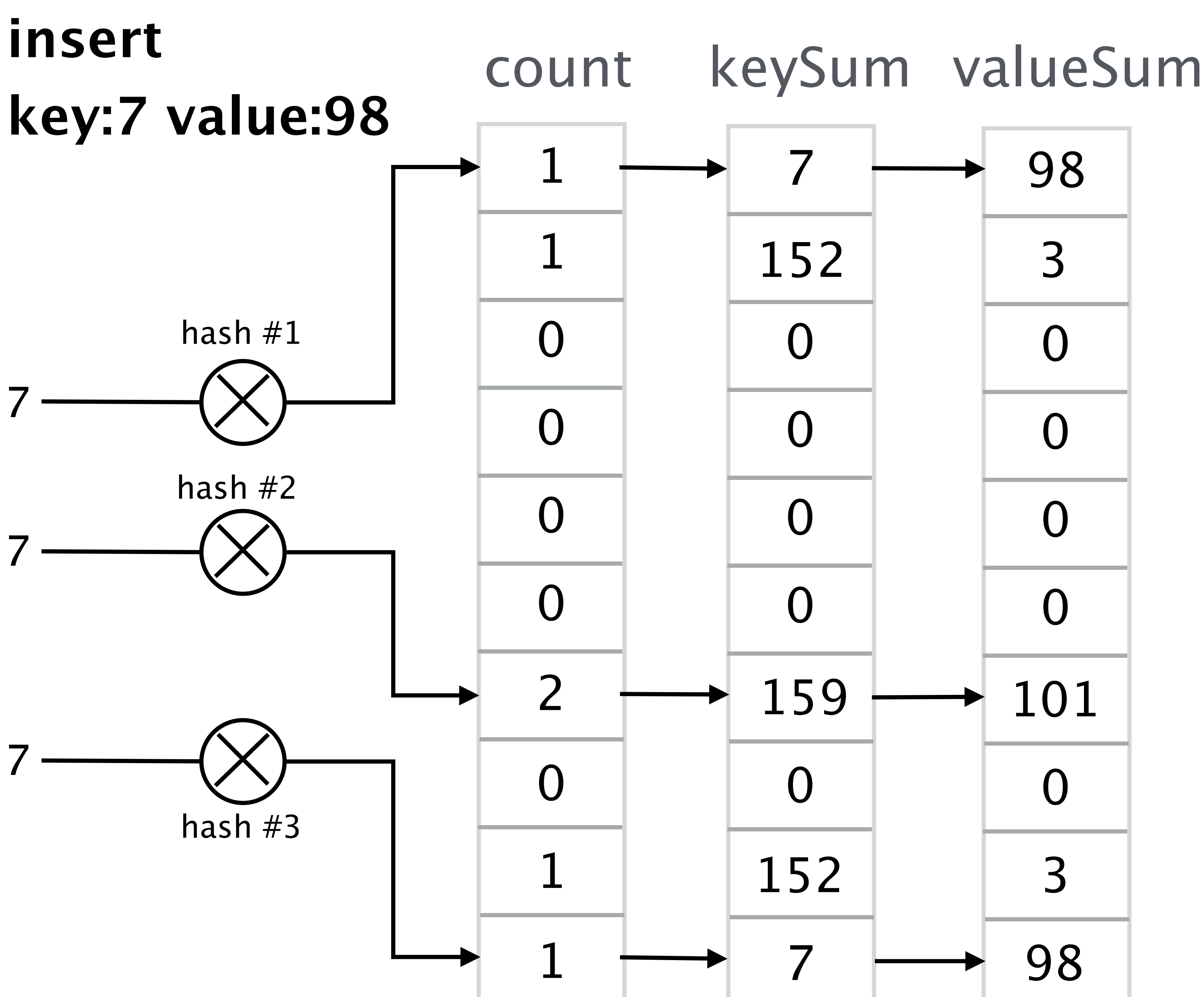
For each hash function
  hash the key to find the index

  Then at this index
    subtract one to the count
    subtract key to keySum
    subtract value to valueSum

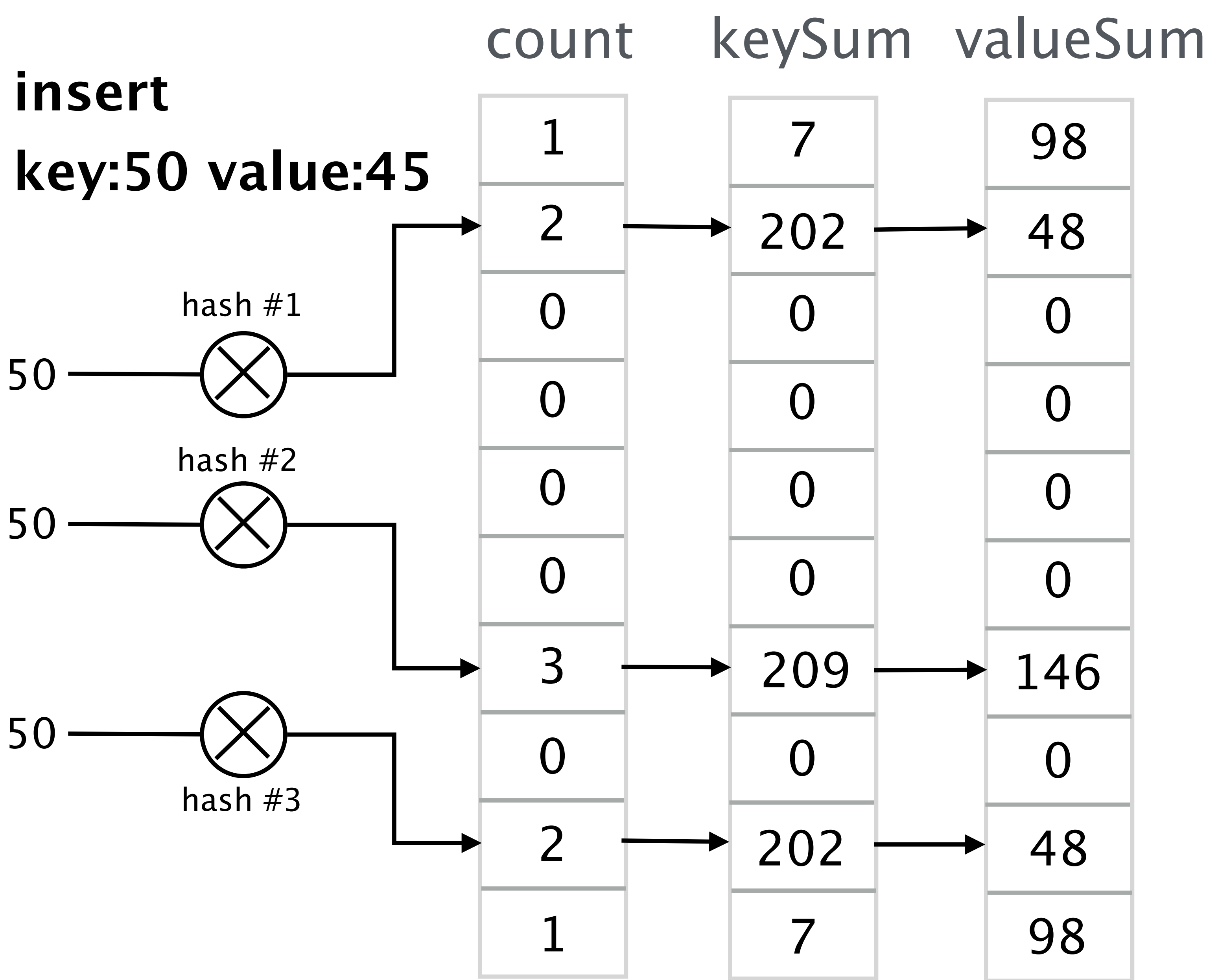
Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing



Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing



Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing



Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing

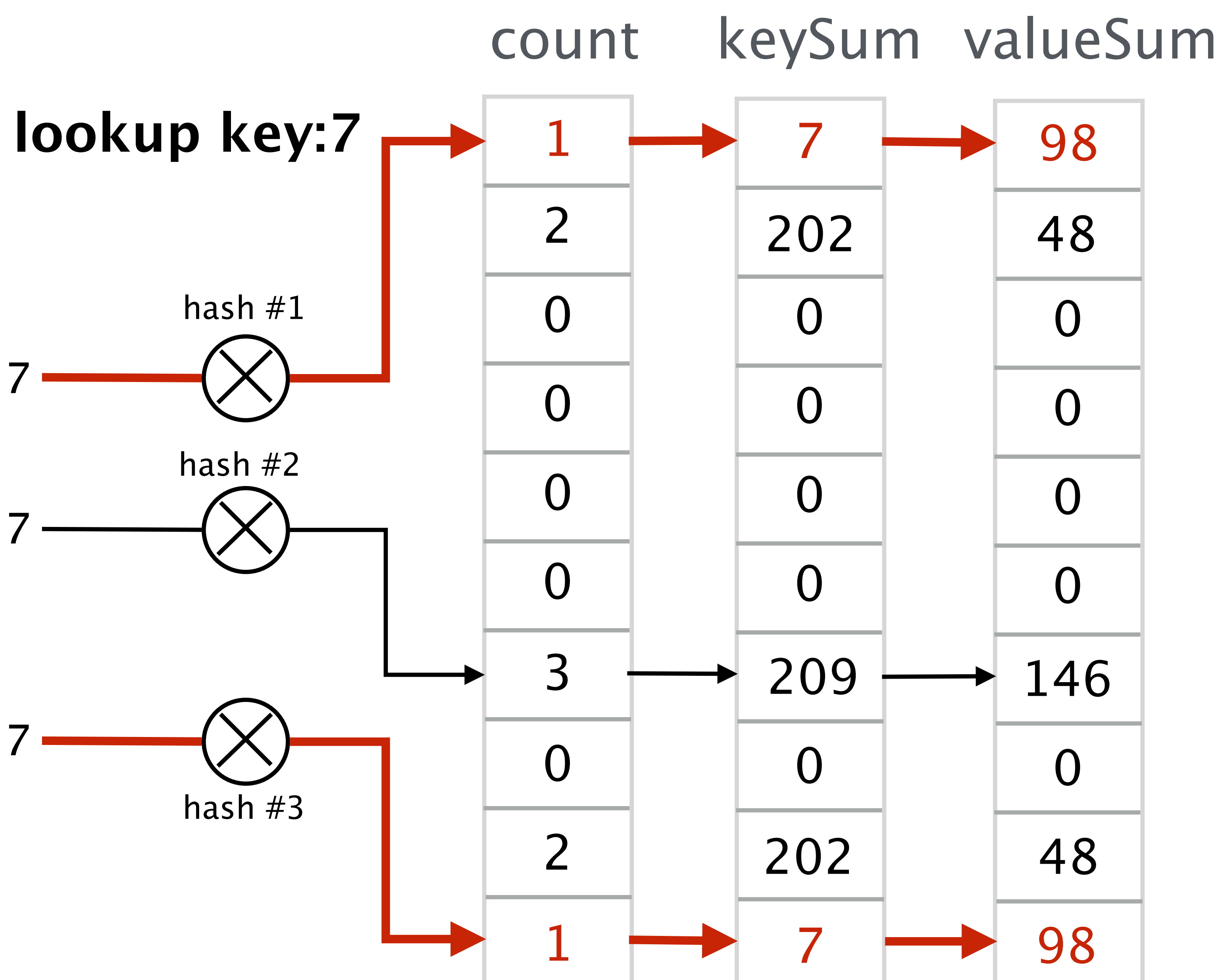
Key-value pair **lookup**

The value of a key can be found
if the key is associated to **at least**
one cell with a count = 1

Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing

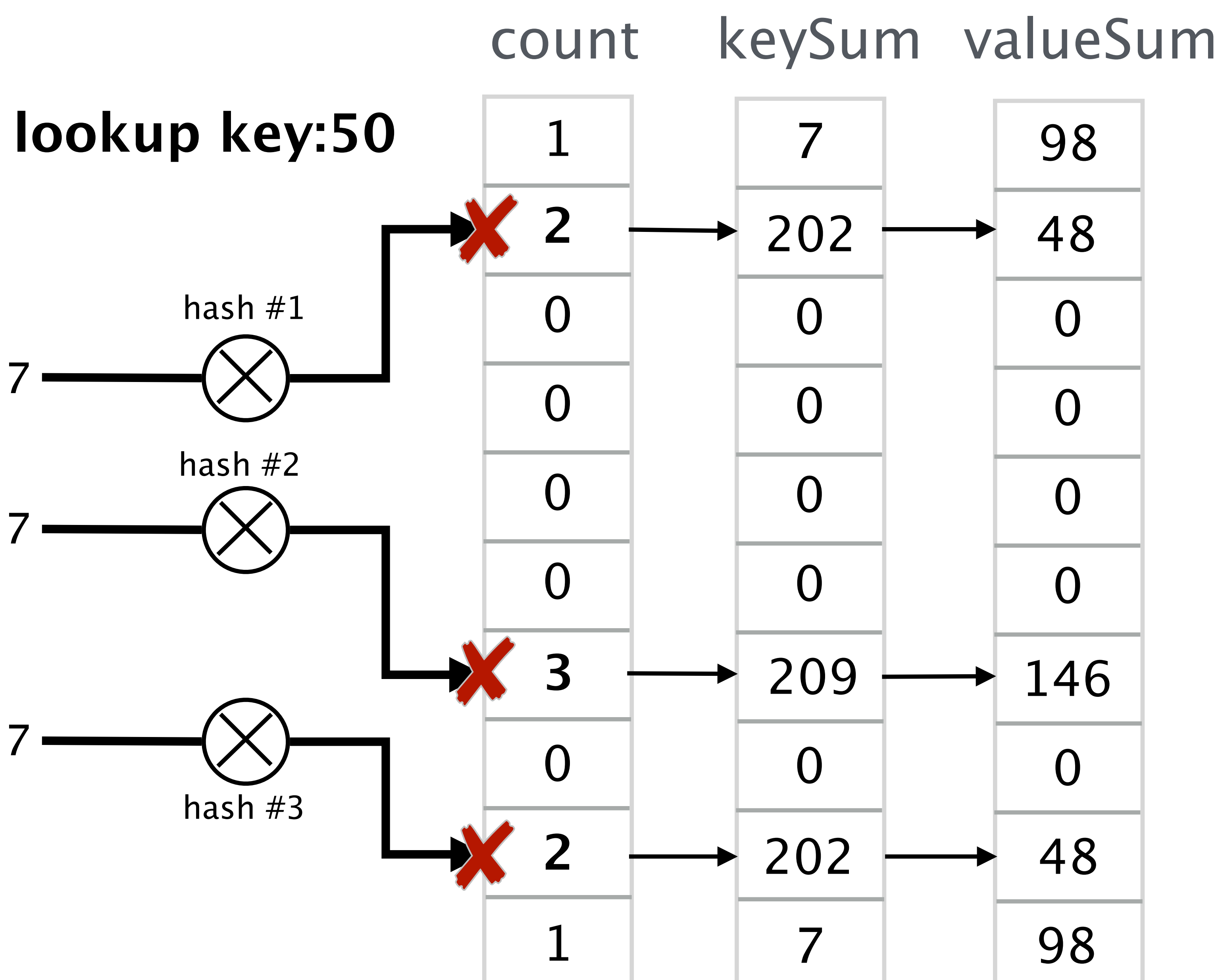
count	keySum	valueSum
1	7	98
2	202	48
0	0	0
0	0	0
0	0	0
0	0	0
3	209	146
0	0	0
2	202	48
1	7	98

Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing



Key 7 has the value 98

Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing



The value for the key 50 can't be found

Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing

Listing the IBLT

While there is an index for which count = 1
  Find the corresponding key-value pair and return it
  Delete the corresponding key-value pair

Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing

Listing the IBLT

While there is an index for which count = 1
  Find the corresponding key-value pair and return it
  Delete the corresponding key-value pair

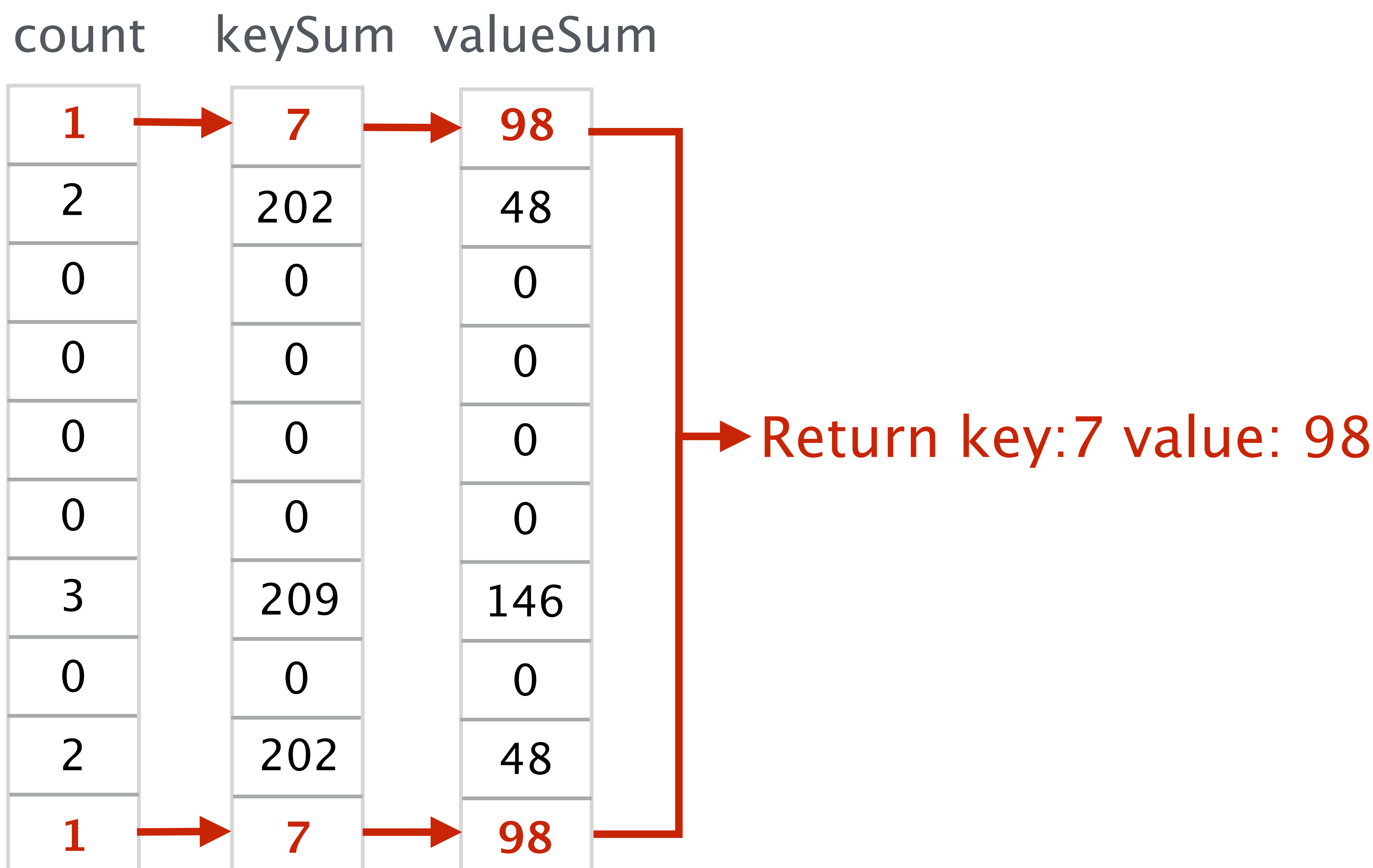
Unless the number of iterations is very low, loops can't be implemented in hardware

The listing is done by the controller

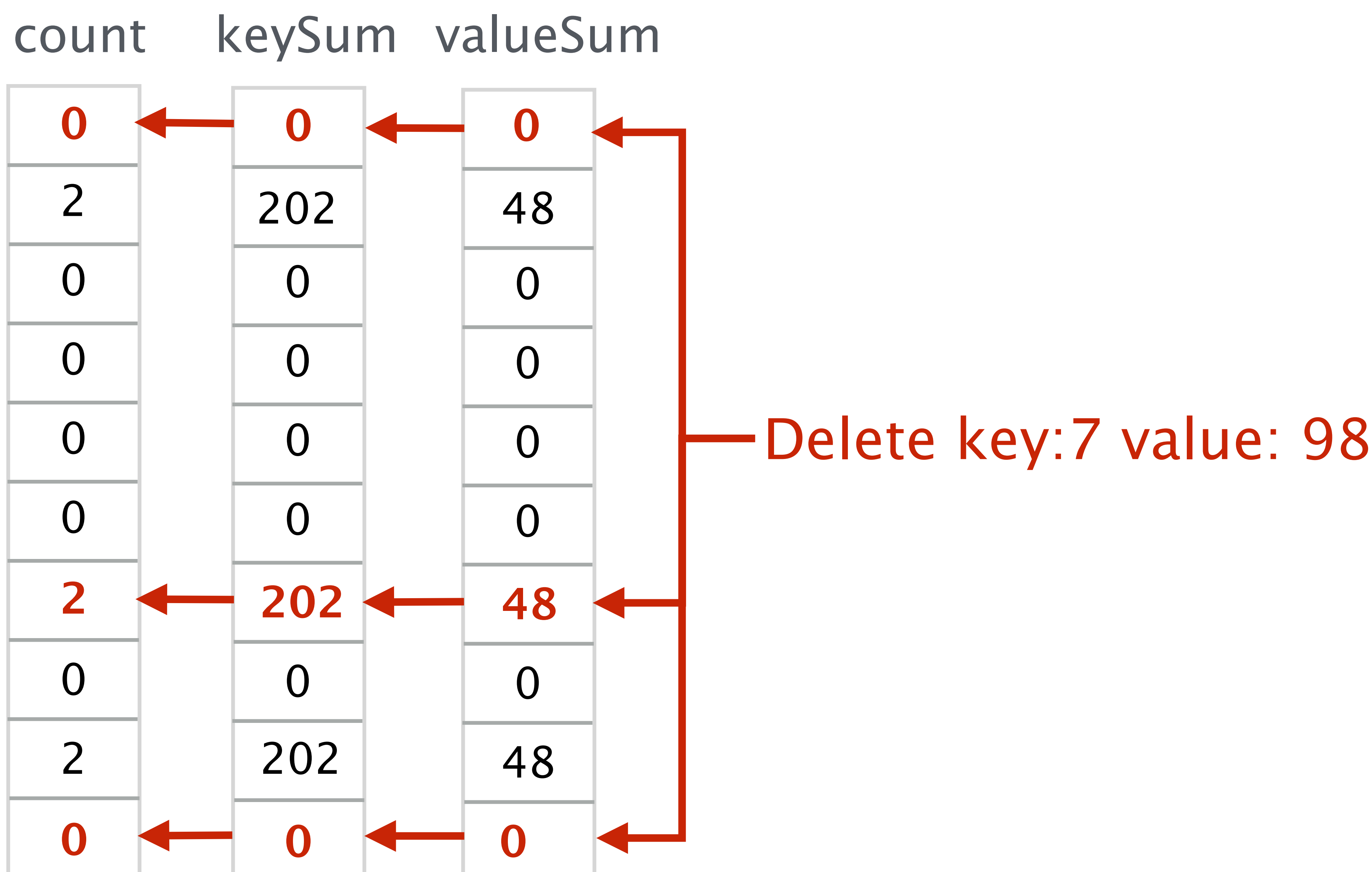
Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing

count	keySum	valueSum
1	7	98
2	202	48
0	0	0
0	0	0
0	0	0
0	0	0
3	209	146
0	0	0
2	202	48
1	7	98

Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing



Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing



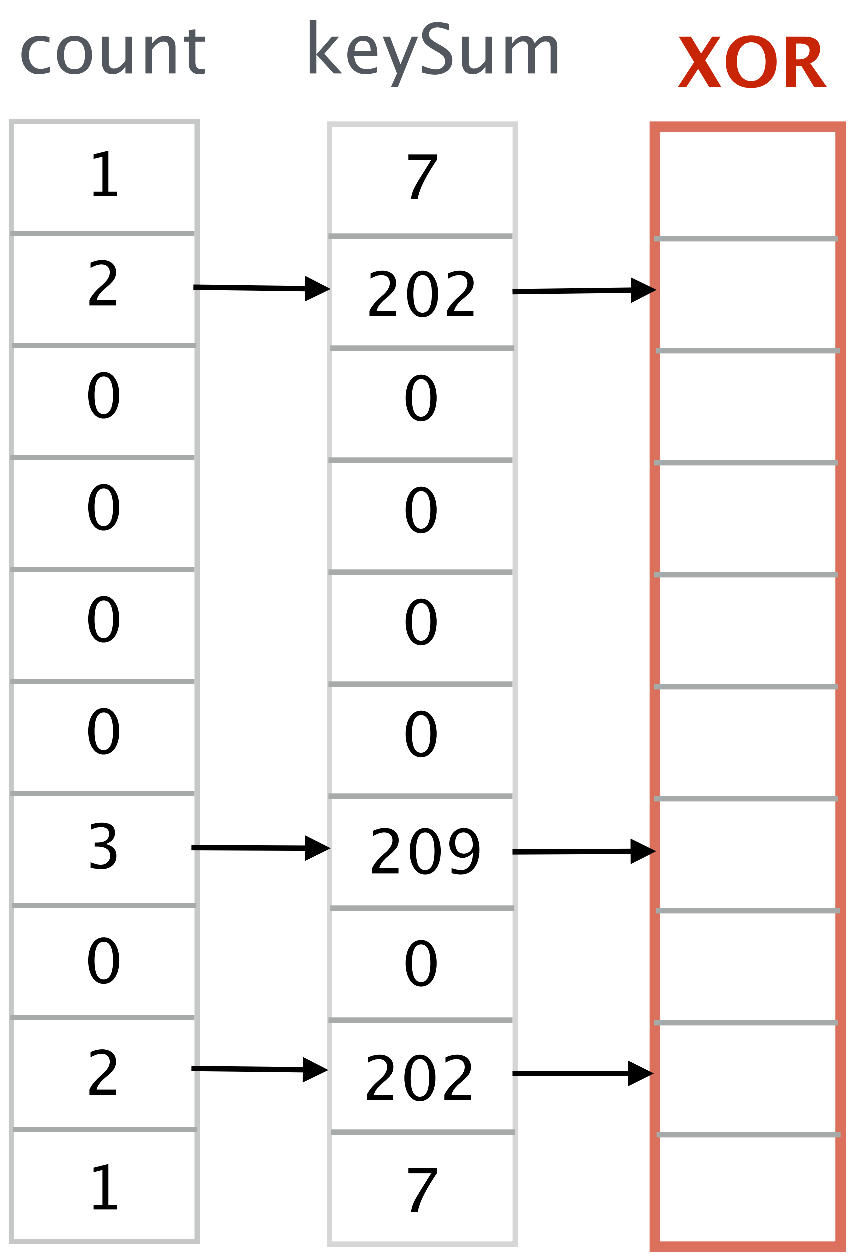
Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing

count	keySum	valueSum
0	0	0
2	202	48
0	0	0
0	0	0
0	0	0
0	0	0
2	202	48
0	0	0
2	202	48
0	0	0

In this example, a complete listing is not possible

In many settings, we can use XORs in place of sums

For example to avoid overflow issues



For further information about Bloom Filters, Counting Bloom Filters and IBLT

Space/Time Trade-offs in Hash Coding with Allowable Errors. Burton H. Bloom. 1970.

Network Applications of Bloom Filters: A Survey. Andrei Broder and Michael Mitzenmacher. 2004.

Invertible Bloom Lookup Tables. Michael T. Goodrich and Michael Mitzenmacher. 2015.

FlowRadar: A Better NetFlow for Data Centers Yuliang Li et al. NSDI 2016.

Advanced Topics in Communication Networks

Programming Network Data Planes

Laurent Vanbever

nsg.ee.ethz.ch

ETH Zürich

Oct 4 2018