22-10-24

CS60021: Scalable Data Minin

Streaming Algorithms

Sourangshu Bhattacharya

Data Streams

- **In many data mining situations, we do not know the entire da set in advance**

- **Stream Management** is important when the input rate is contr **externally:**
  - Google Trends
  - Twitter or Facebook status updates

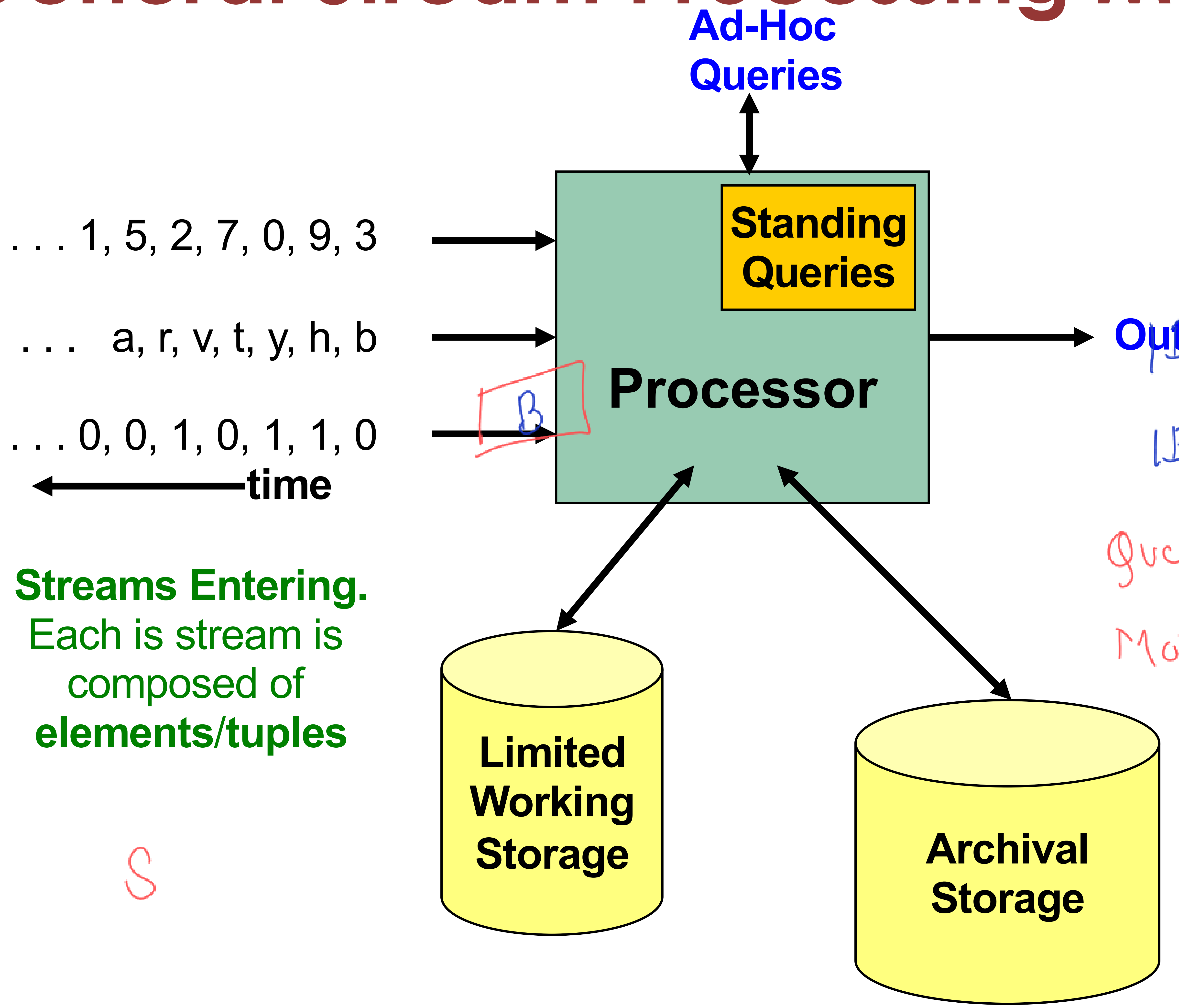
- We can think of the **data** as **infinite** and **non-stationary** (the distribution changes over time)

- Input **elements** enter at a rapid rate, at one or more input ports (i.e., **streams**)
  - **We call elements of the stream tuples**

- **The system cannot store the entire stream accessibly**

- **Q: How do you make critical calculations about the stream using a limited amount of (secondary) memory?**

General Stream Processing Model



$|B| << |S|$

$|B| = O(\log\log|S|)$

Query Time: $O(1)$

Maintaining time $= O(|B|)$

Reservoir Sampling

Maintaining a fixed-size sample

- **Problem: Fixed-size sample**
- **Suppose we need to maintain a random sample *S* of size exactly *s* tuples**
  - E.g., main memory size constraint
- **Why?** Don't know length of stream in advance
- **Suppose at time *n* we have seen *n* items**
  - **Each item is in the sample *S* with equal prob. *s/n***

How to think about the problem: say s = 2

Stream: a x c y z k c d e g…

At **n= 5,** each of the first 5 tuples is included in the sample **S** with equal prob.

At **n= 7,** each of the first 7 tuples is included in the sample **S** with equal prob.

Impractical solution would be to store all the *n* tuples seen so far and out of them pick *s* at random

Solution: Fixed Size Sample

- **Algorithm (a.k.a. Reservoir Sampling)**
  - Store all the first ***s*** elements of the stream to ***S***
  - Suppose we have seen ***n-1*** elements, and now the n^{th} element arrives ($n > s$)
    - With probability s/n, keep the n^{th} element, else discard it
    - If we picked the n^{th} element, then it replaces one of the ***s*** elements in the sample ***S***, picked uniformly at random
- **Claim:** This algorithm maintains a sample ***S*** with the desired property:
  - After ***n*** elements, the sample contains each element s
    so far with probability s/n

Proof: By Induction

- **We prove this by induction:**
  - Assume that after ***n*** elements, the sample contains each element seen so far with probability ***s/n***
  - We need to show that after seeing element ***n+1*** the sample maintains the property
    - Sample contains each element seen so far with probability ***s/(n***
- **Base case:**
  - After we see **n=s** elements the sample **S** has the desired property
    - Each out of **n=s** elements is in the sample with probability ***s/s =***

Proof: By Induction

- **Inductive hypothesis:** After ***n*** elements, the sample ***S*** contains each element seen so far with prob. ***s/n***
- **Now element *n+1* arrives**
- **Inductive step:** For elements already in ***S***, probability that the algorithm keeps it in ***S*** is:

$$\underbrace{\left(1-\frac{s}{n+1}\right)}_{\text{Element } \mathbf{n+1} \text{ discarded}}+\underbrace{\left(\frac{s}{n+1}\right)}_{\text{Element } \mathbf{n+1} \text{ not discarded}}\underbrace{\left(\frac{s-1}{s}\right)}_{\text{Element in the sample not picked}}=\frac{n}{n+1}$$

- So, at time ***n,*** tuples in ***S*** were there with prob. **s/n**
- Time ***n→n+1***, tuple stayed in ***S*** with prob. **n/(n+1)**
- So prob. tuple is in ***S*** at time ***n+1*** $= \frac{s}{n} \cdot \frac{n}{n+1} = \frac{s}{n+1}$

Type of sketch

→ Sample → Reservior Sampling

→ Hash Table

↳ False positive

→ Specialized data Structure

Queries

↳ Membership

↳ Count distinct

↳ min/max/avg/median

↳ moment.

Criteria Should be Satisfied

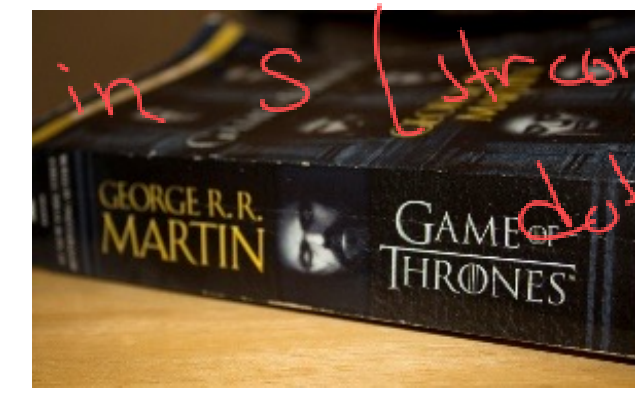
$P(|\varphi(S) - \varphi(B)| < \epsilon) > 1-\delta$

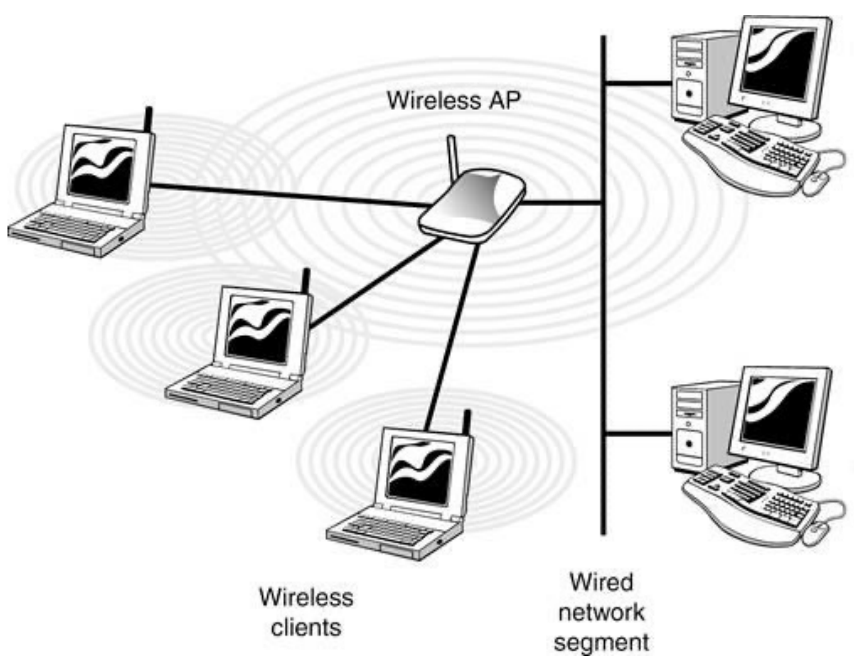
$|B| << |S|$

Bloom Filters

given dataset Z, whether Z appear in S (streaming dataset)

ISBN present in collection?



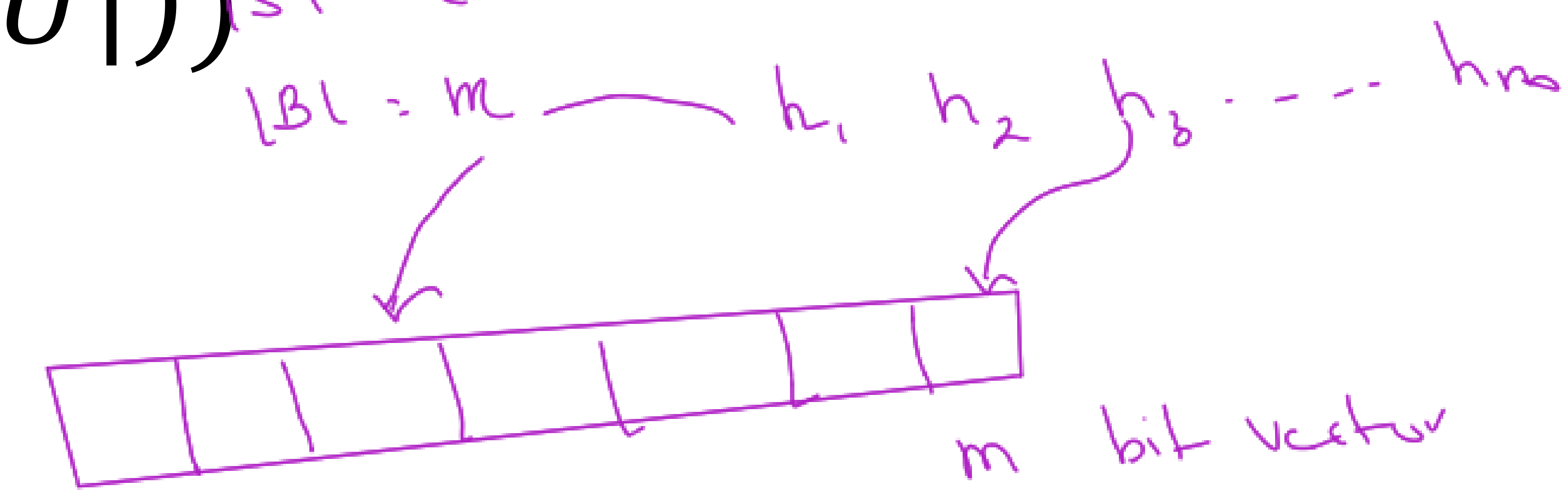


IP seen by switch?

10.0.21.102

- Universe U, but need to store a set of n items, n
  $|U|$.
- Hash table of size m:
  - Space $O(m + n\log(|U|))$
  - Query time $O(\frac{n}{m})$

|U| = N
|S| = n
|B| = m



if z has occured in S
g(z) is always 1

Sometimes if z has not occured
still g(z) can be 1 (False positive)

- Universe U, but need to store a set of n items, n … $|U|$.
- Hash table of size m:
  - Space $O(m + n\log(|U|))$
  - Query time $O(\frac{n}{m})$
- Bit array of size $|U|$
  - Space $|U|$.
  - Query time $O(1)$.

P(False Positive) = ?
P(False negative) = 0

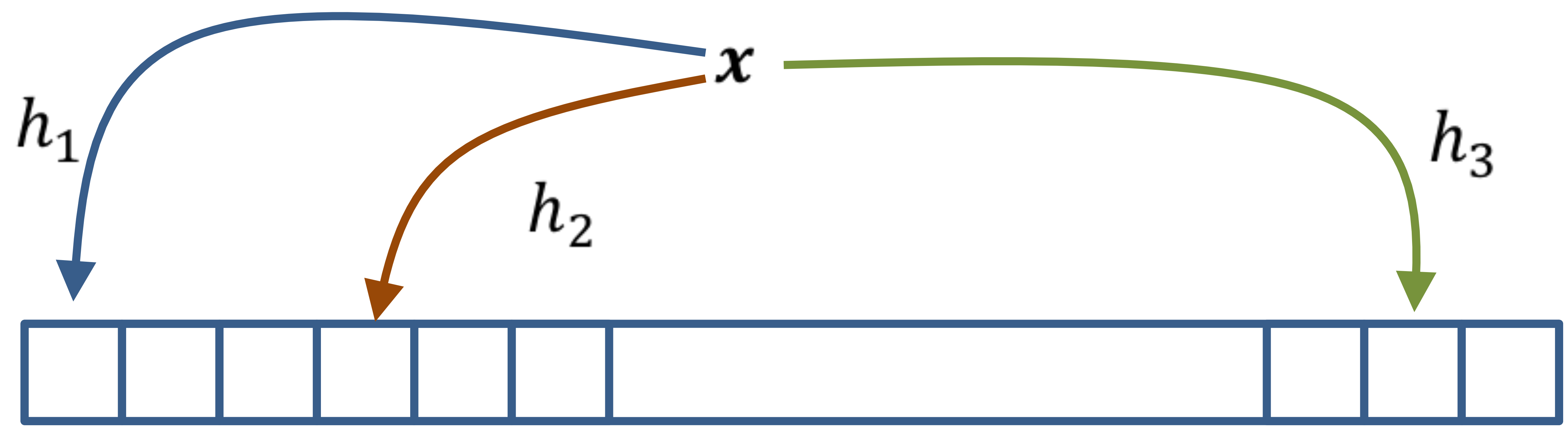
* False positive happens because of over-writing to 1
* Find the probability when one of hash function overwrites 1 while inserting x_{n+1}
* What is probability that $B[h_1(x_{n+1})] = 1$ already.
  $= \frac{\#\text{ 1's } B}{|B|}$
* Find the expected number of # 1's in B after n-operation.

- In hash table construction, we used random hash functi
  - we never return incorrect answer
  - query time is a random variable
  - These are Las Vegas algorithms

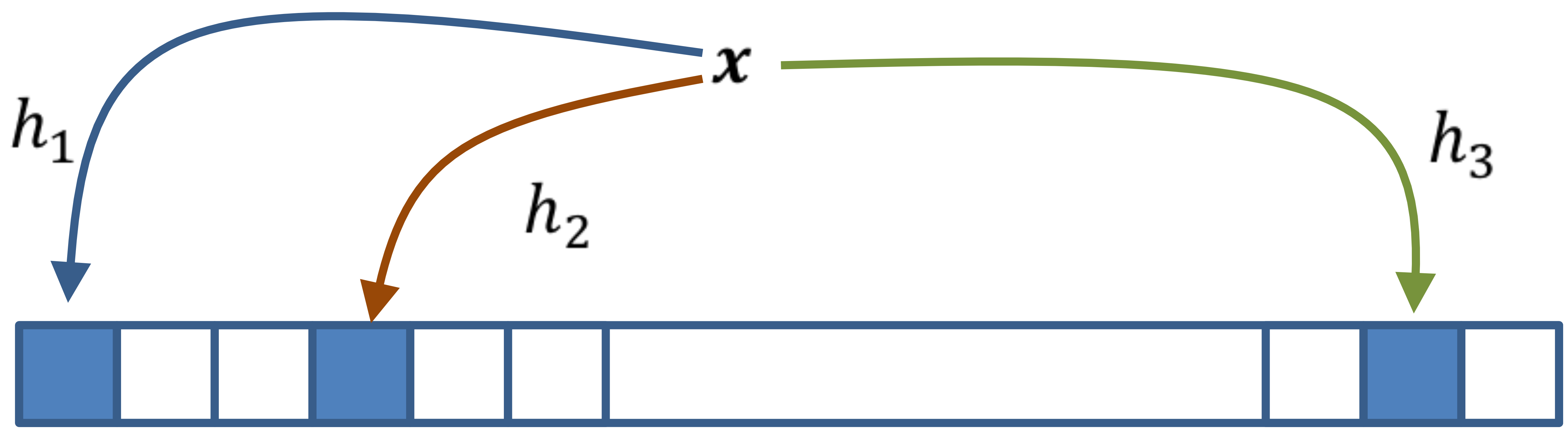
- In Monte-Carlo randomized algorithms, we are allowed
  return incorrect answers with (small) probability, say, δ

[Bloom, 1970]

- A bit-array B, $|B| = m$
- k hash functions, $h_1, h_2, \dots, h_k$, each $h_i \in U \to [m]$



- A bit-array B, $|B| = m$
- k hash functions, $h_1, h_2, \dots, h_k$, each $h_i \in U \to [m]$



- $Initialize(B)$
  - for $i \in \{1, .. m\}$, $B[i] = 0$

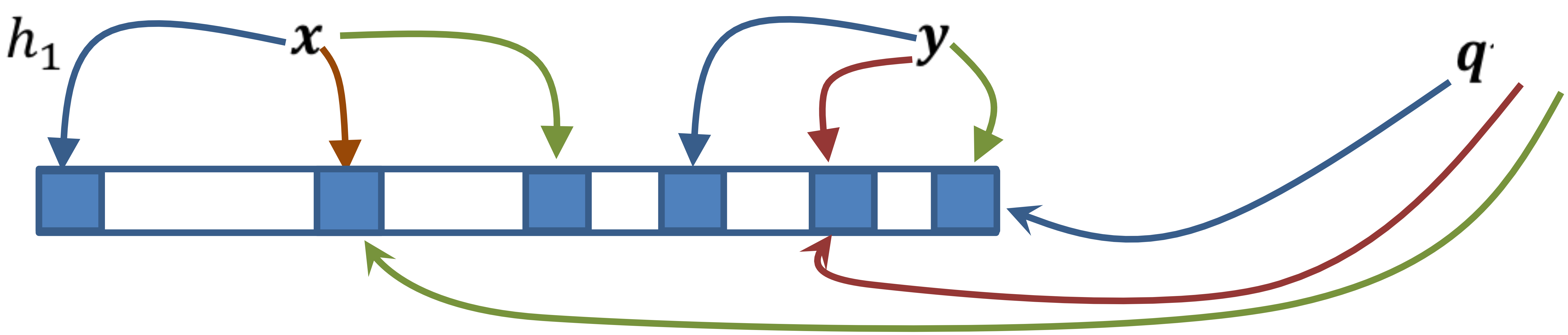
- $Insert\ (B, x)$
  - for $i \in \{1, .. k\}$, $B[h_i(x)] = 1$

- $Lookup\ (B, x)$
  - If $\bigwedge_{i \in \{1,\ldots k\}} B[h_i(x)]$, return PRESENT, else ABSENT

- If the element x has been added to the Bloom filter, then $Lookup(B, x)$ always return PRESENT

- If the element x has been added to the Bloom filter, then $Lookup(B, x)$ always return PRESENT

- If x has not been added to the filter before?
  - $Lookup$ sometimes still return PRESENT



- Want to minimize the probability that we retu
false positive
- Parameters $m = |B|$ and $k =$ number of has
functions
- $k = 1 \;\Rightarrow$ normal bit-array

- What is effect of changing k?

- Increasing k
  - Possibly makes it harder for false positives to happen in $Lookup$ because of $\wedge_{i\in\{1,\ldots k\}} B[h_i($
  - But also increases the number of filled up positions
- We can analyse to find out an “optimal k”

False positive analysis

- $m = |B|$, n elements inserted
- If x has not been inserted, what is the probability that $Lookup(B, x)$ returns PRESENT?

- $m = |B|$, n elements inserted
- If x has not been inserted, what is the probability that $Lookup(B, x)$ returns PRESENT?
- Assume $\{h_1, h_2, \ldots h_k\}$ are independent and $\Pr[h_i(\cdot) = j] = \frac{1}{m}$ for all positions j

↳ equal probable for all m block

- Probability of a bit being zero:

$$P[B_j = 0] = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-\frac{kn}{m}}$$

- The expected number of zero bits is given by: $me^{-kn/m}$.

- $P[lookup(B, x) = PRESENT] = \left(1 - e^{-\frac{kn}{m}}\right)^k$

Probability that particular B_j for h_j already set.

- We can choose k to minimize this probability.

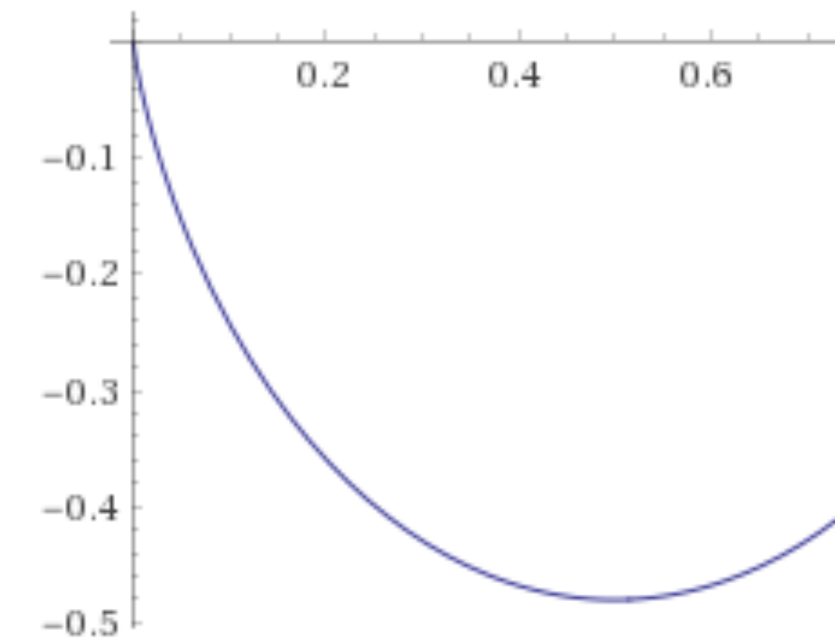
Probability that x is already inserted into B

functions

- $p = e^{-kn/m}$
- Log (False Positive) =

$$\log(1 \ - p)^k = k\log(1-p) = -\frac{m}{n}\log(p)\log(1-$$

Minimized at $p = \frac{1}{2}$, i.e. $k = \mathrm{m}\log(2)/\mathrm{n}$



- This "optimal" choice gives false positive = $2^{-m\log(2)/n}$

- If we want a false positive rate of δ , set $m = \left\lceil \frac{\log\left(\frac{1}{\delta}\right)n}{\log^2(2)} \right\rceil$

<u>Example</u>: If we want 1% FPR, we need 7 hash func

and total $10n$ bits

- Widespread applications whenever small false positives are tolerable

- Used by browsers
  – to decide whether an URL is potentially malicious: a BF is used in browser, a positives are actually checked with the server.

- Databases e.g. BigTable, HBase, Cassandra, Postgrepsql use to avoid disk lookups for non-existent rows/columns

- Bitcoin for wallet synchronization….

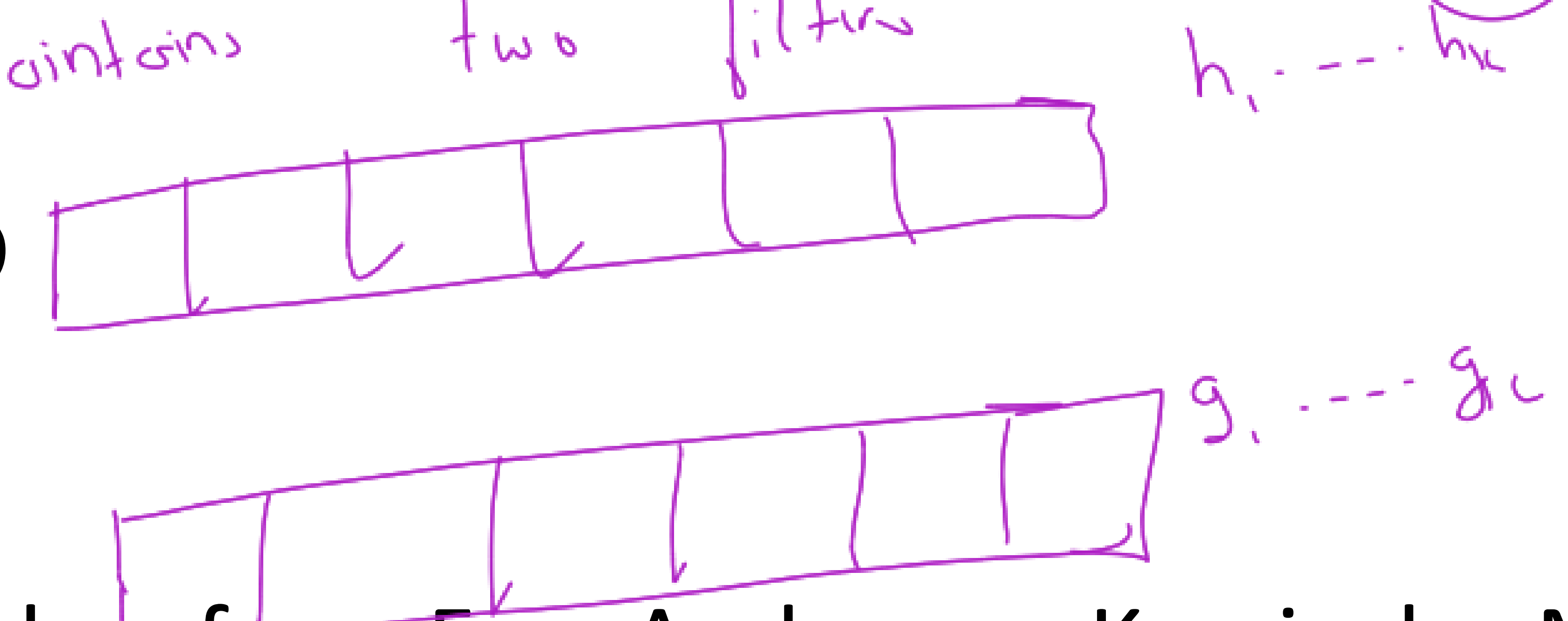
- Chief drawback is that BF does not allow deletions

[Fan et al 00]

- Counting Bloom Filter
  - Every entry in BF is a small counter rather than a single bit
  - $Insert(x)$ increments all counters for $\{h_i(x)\}$ by 1
  - $Delete(x)$ decrements all $\{h_i(x)\}$ by 1
  - maintains 4 bits per counter
  - False negatives can happen, but only with low probability

CUCKOO FILTERS

So if for x_{n+1} $h_1(x_{n+1})$ is already
1. Go for $g_1(h\, x_{n+1})$ if that is not
1. Set it to 1. if one do something
maintains two filters



Slides taken from Fan, Andersen, Kaminsky, M

- **Fingerprint**(x): A hash value of x
  - Lower false positive rate $\boldsymbol{\varepsilon}$, longer fingerprint

→ Problem : on given n-data points from N Count the number of distinct number

Can do in $O(\log N)$?

given data point $x \longrightarrow i$ ($\log_2 N$ bits)

$x_1 \longrightarrow h(x) \longrightarrow$ binary $\longrightarrow 11\underline{000}_{3}$

⋮

i (trailing 0_s in x)

Calculate for all i and

if max is max_i

So the number of distinct will be 2^{max_i}

R: max_i

Flajolet–Martin Algorithm

0:

1:

2:

3:

4:

5:

6:

7:

$\hookrightarrow m(1-\epsilon)^n < 2^R \leq m(1+\epsilon)^n$

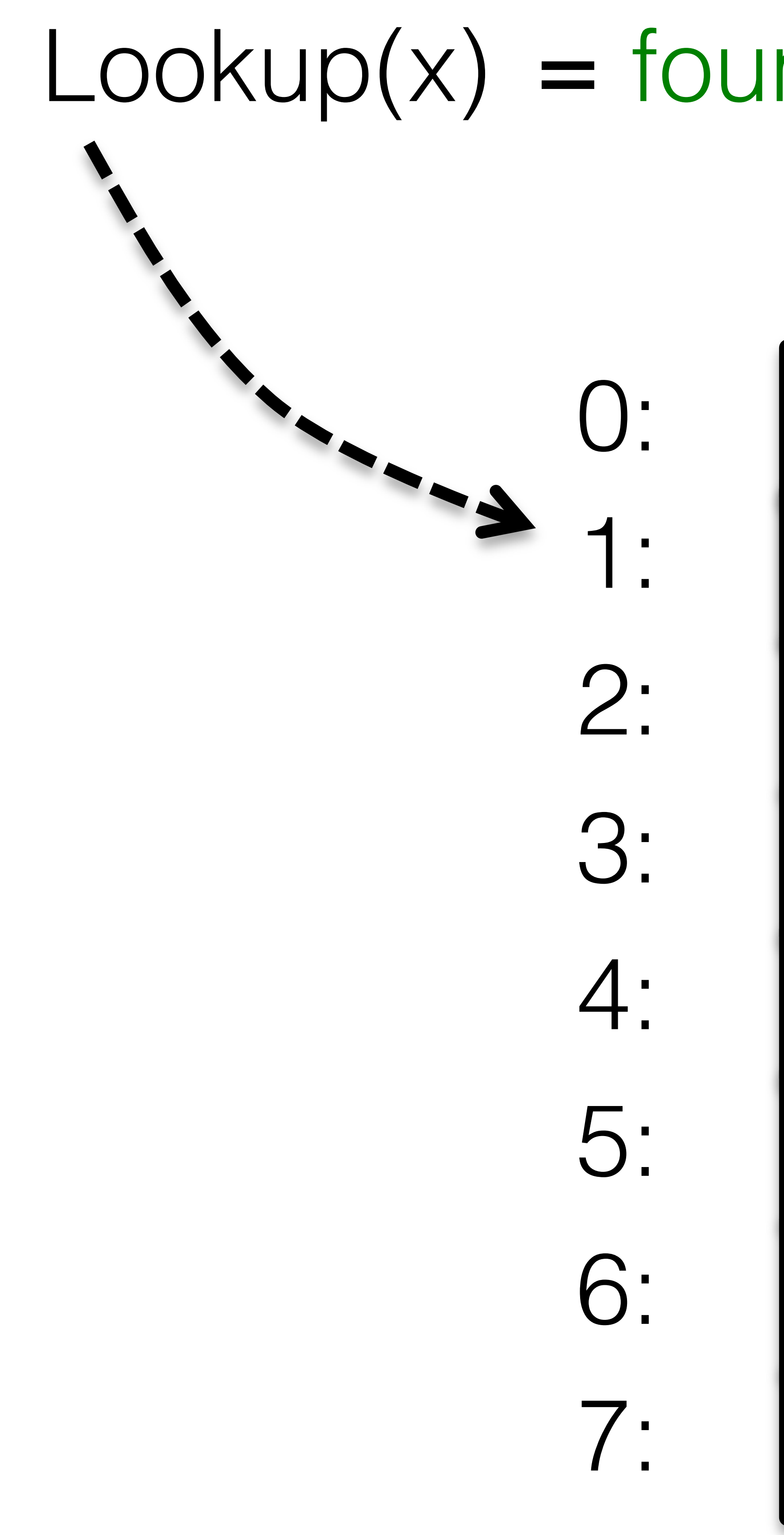
R: max number of trailing zeros in any x

- **Fingerprint**(x): A hash value of x
  - Lower false positive rate $\boldsymbol{\varepsilon}$, longer fingerprint

- **Insert**(x):
  - add **Fingerprint(x)** to hash table

0:
1:
2:
3:
4:
5:
6:
7:

- **Fingerprint**(x): A hash value of x
  - Lower false positive rate ε, longer fingerprint
- **Insert**(x):
  - add **Fingerprint(x)** to hash table
- **Lookup**(x):
  - search **Fingerprint**(x) in hashtable

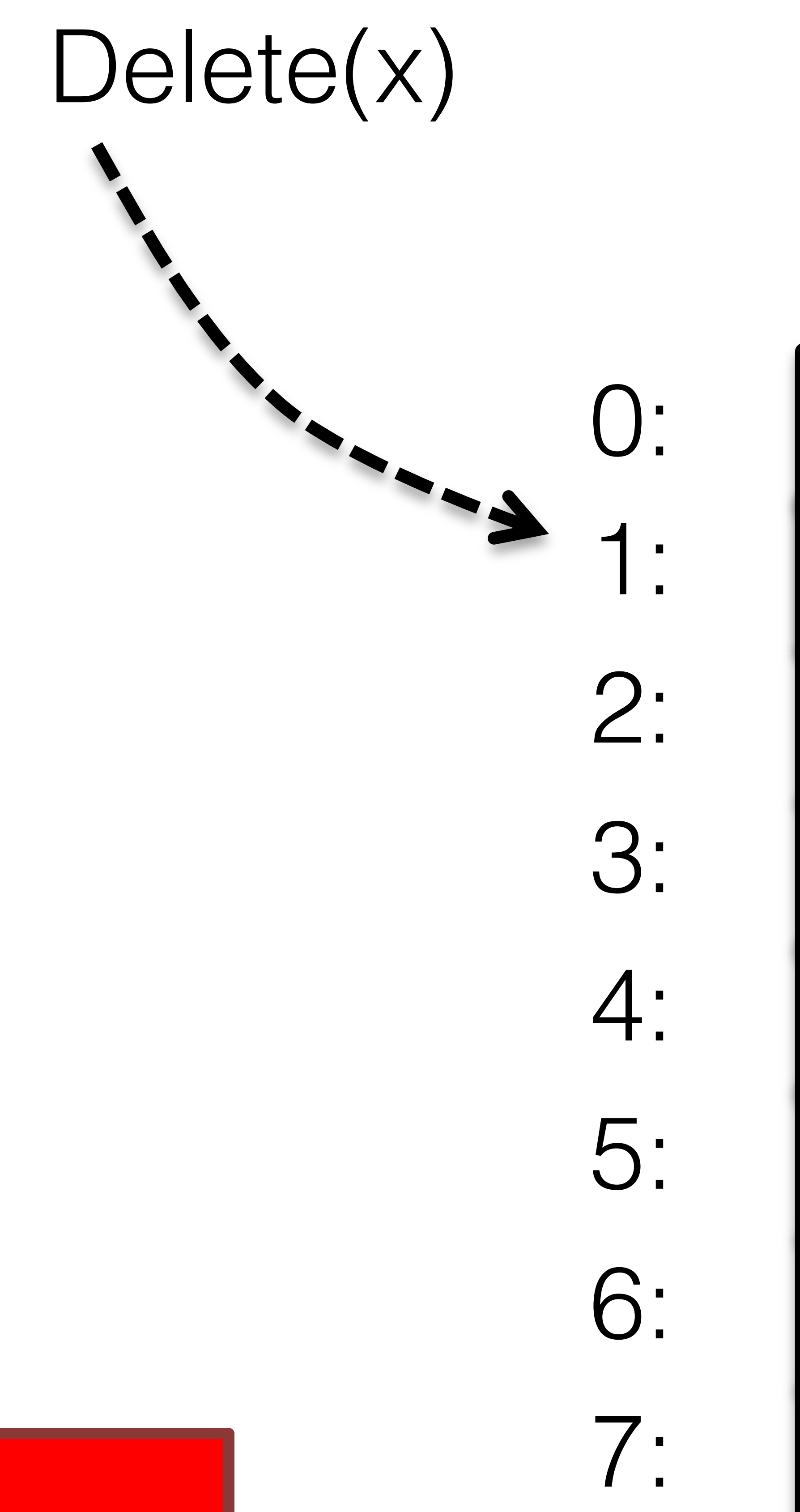


- **Fingerprint**(x): A hash value of x
  - Lower false positive rate ε, longer fingerprint

- **Insert**(x):
  - add **Fingerprint(x)** to hash table

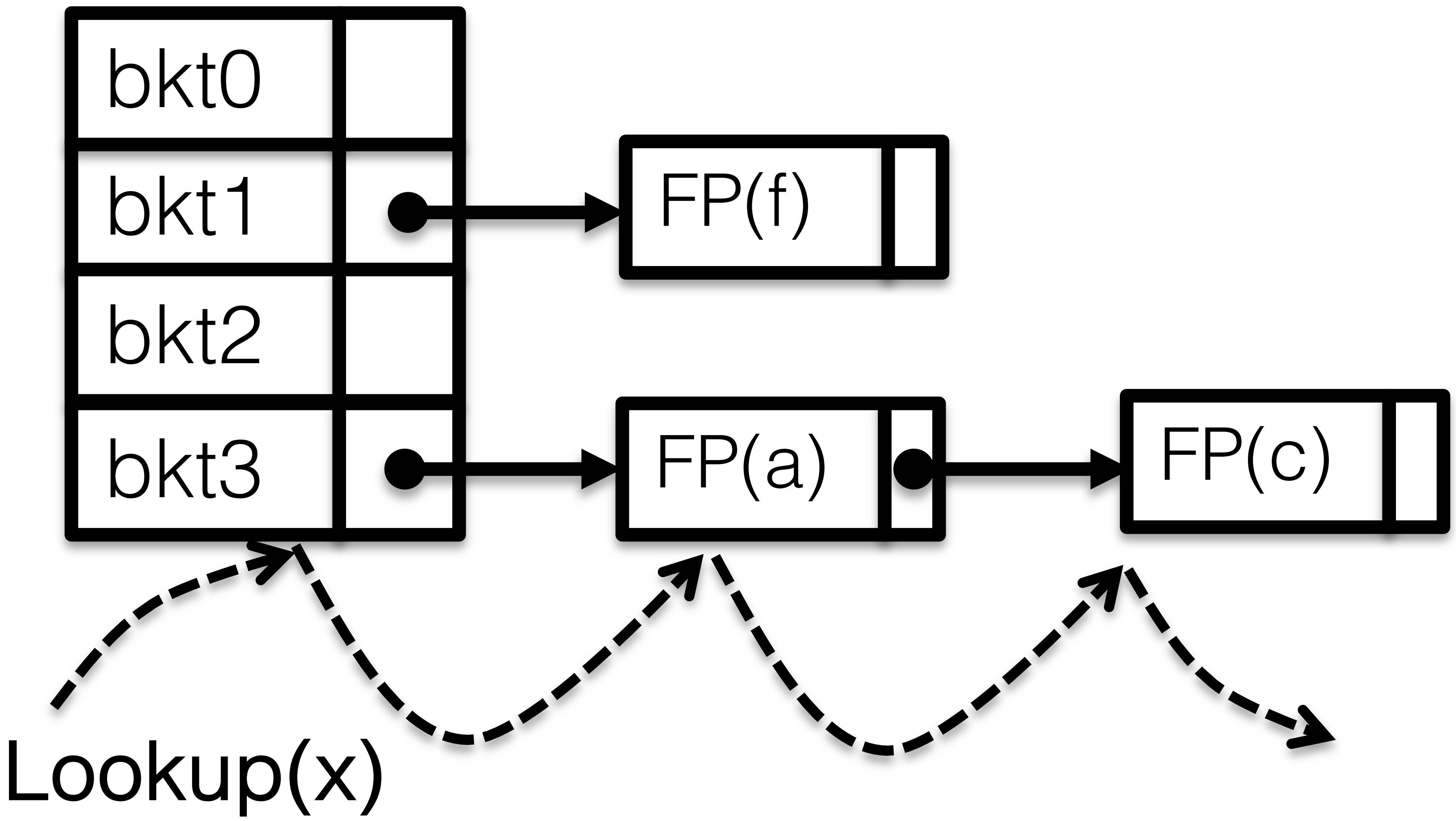
- **Lookup**(x):
  - search **Fingerprint**(x) in hashtable

- **Delete**(x):
  - remove **Fingerprint**(x) from hashtable



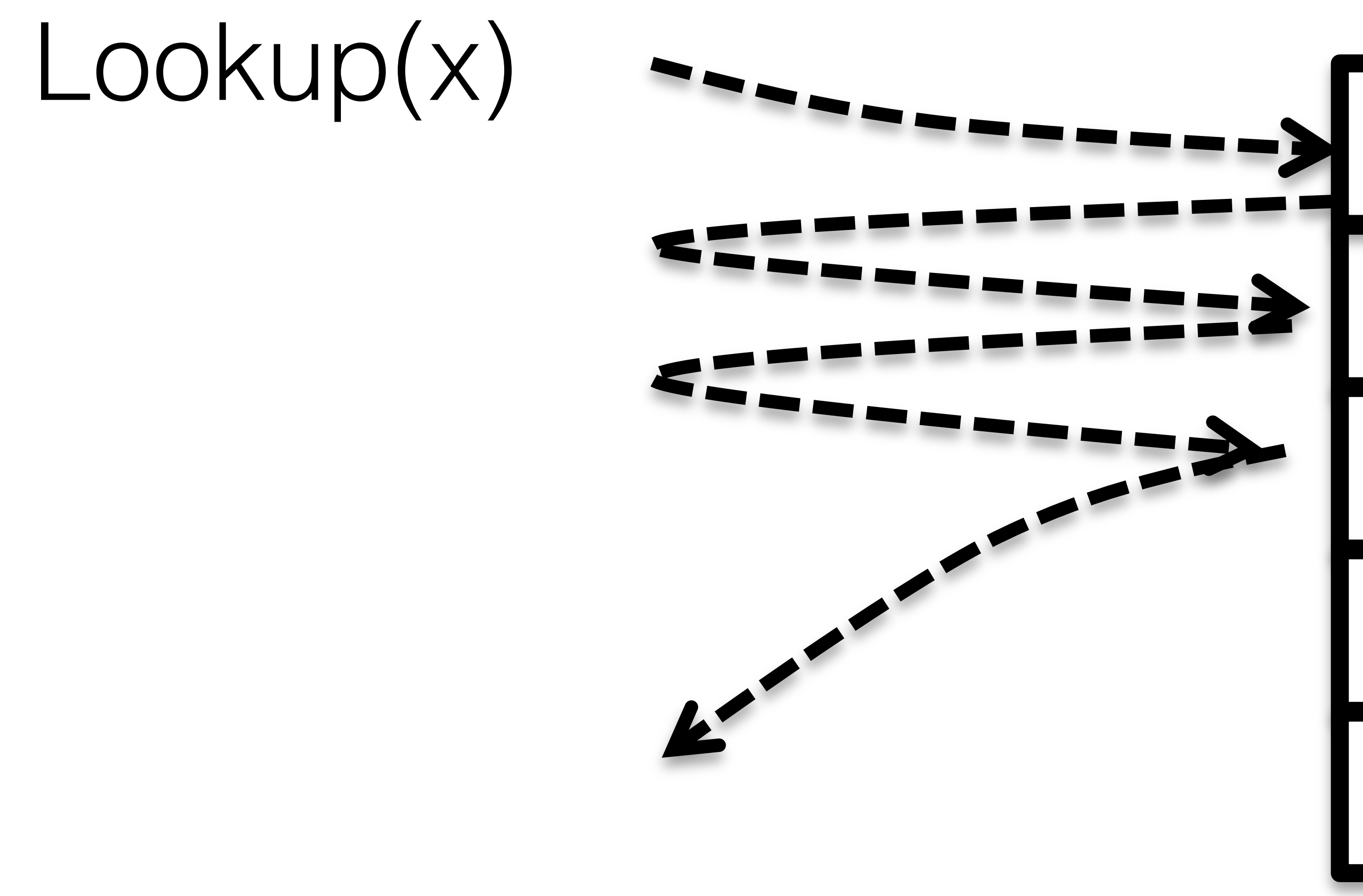
How to Construct Hashtable?

- Chaining :



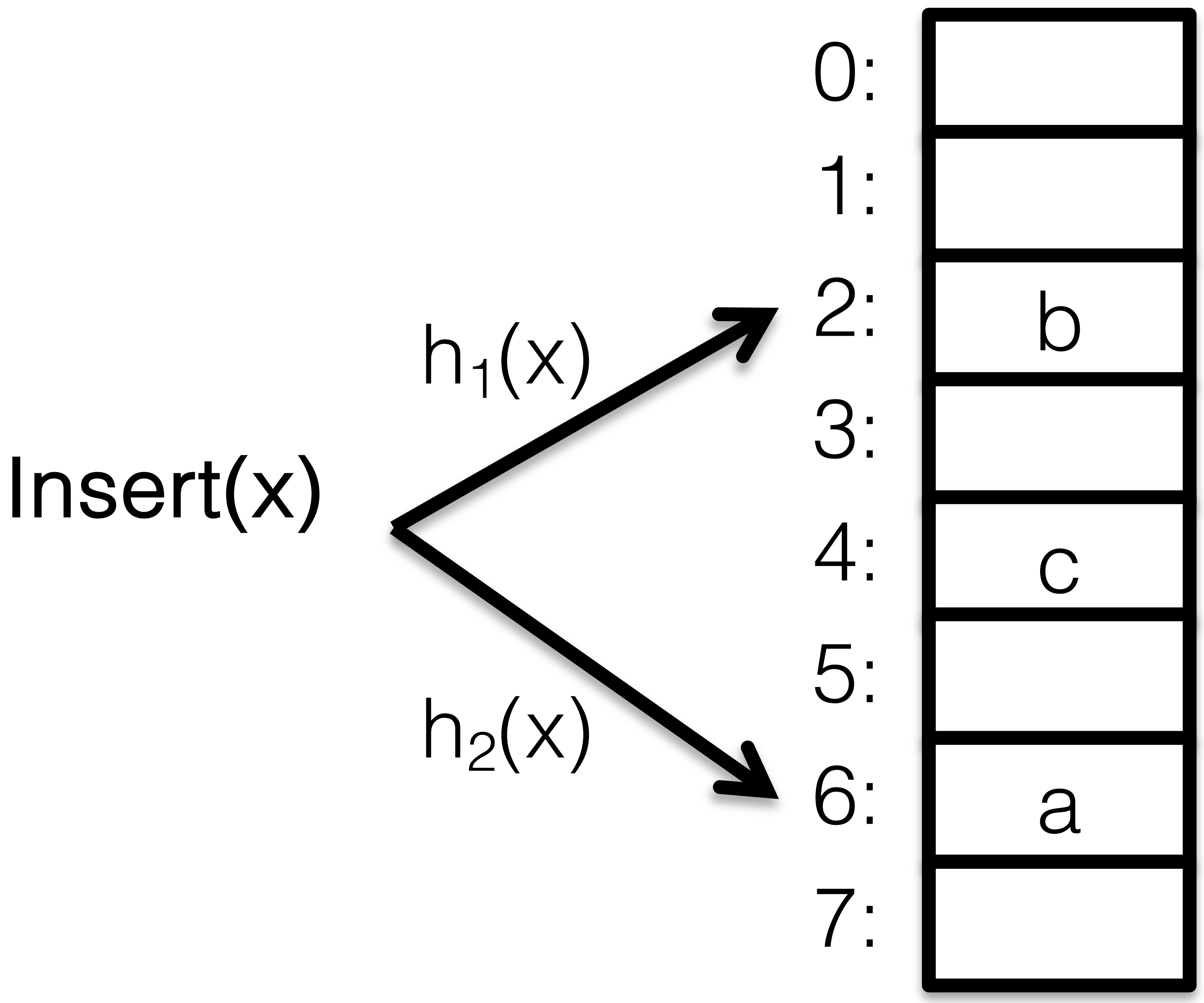
- Pointers ➔ low space utilization

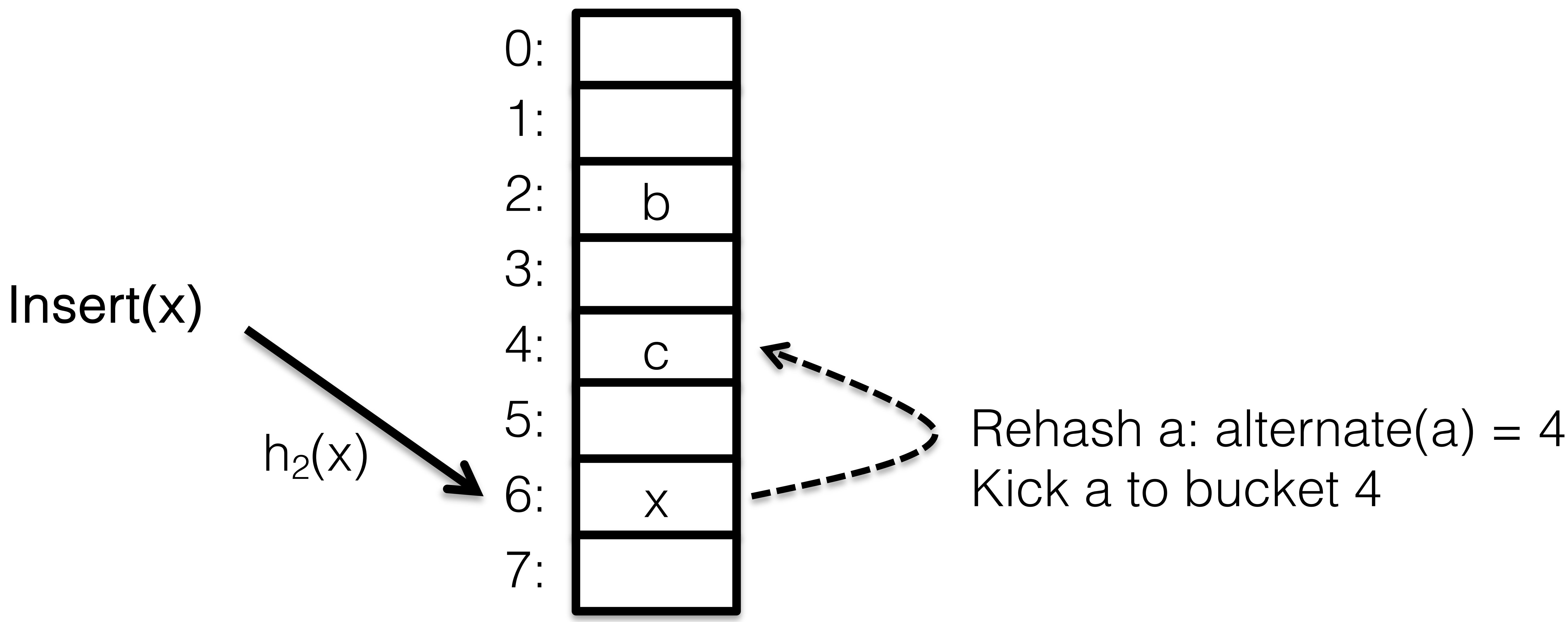
- Linear Probing



- Making lookups O(1) requi % table empty ➔ low space utilization
- Compare multiple fingerpri sequentially ➔ more false positives

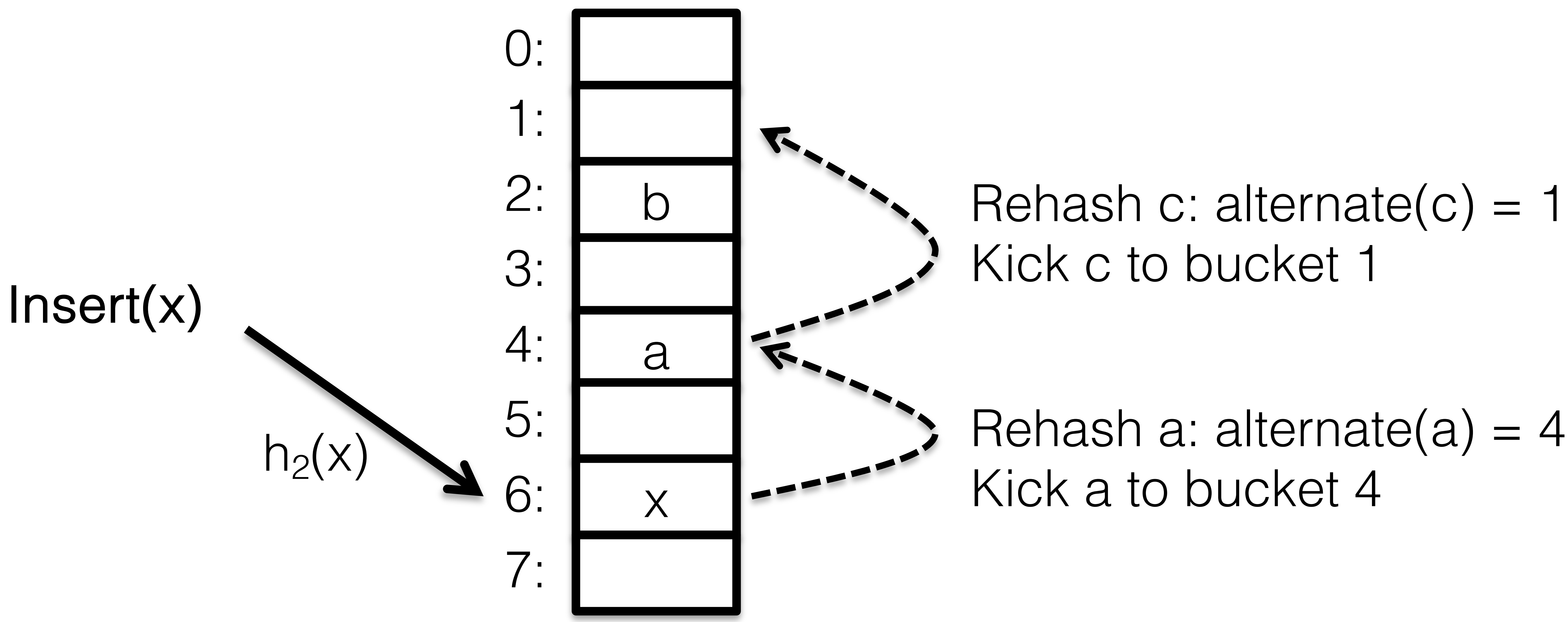
Item



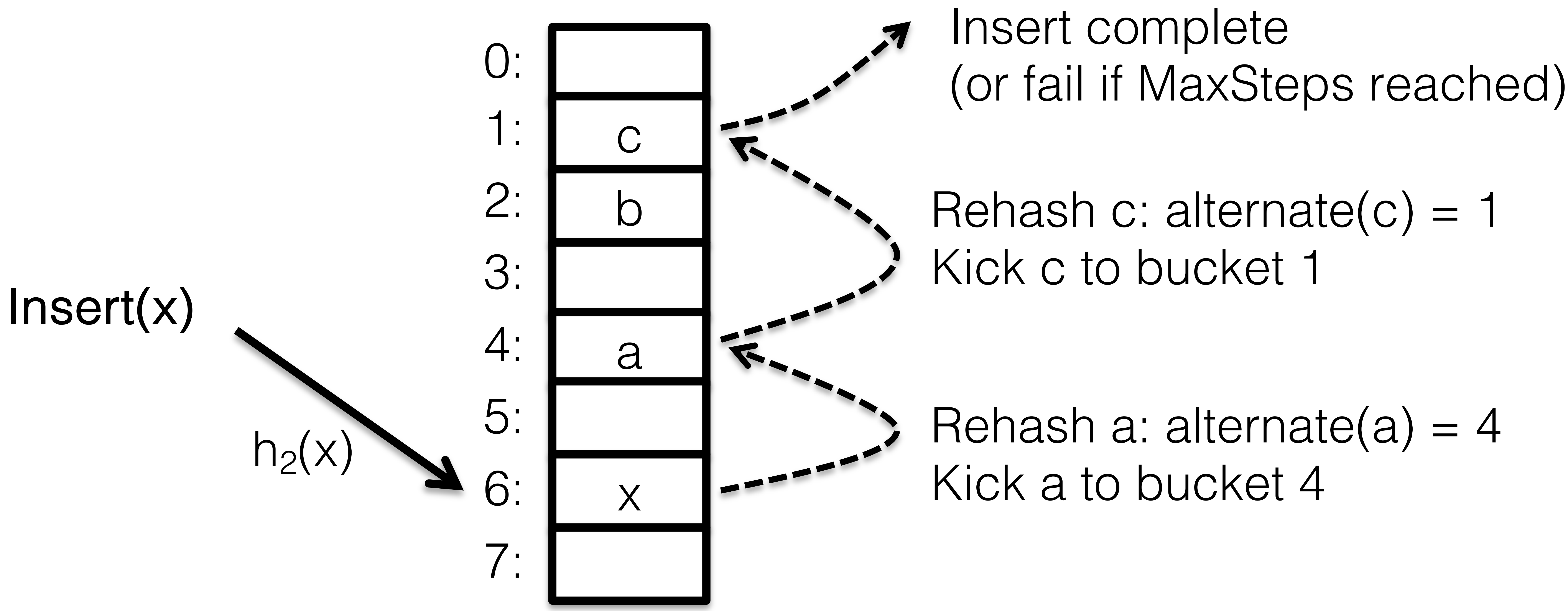


Insert(x)
$h_2(x)$
0:
1:
2:
3:
4:
5:
6:
7:
b
c
x
Rehash a: alternate(a) = 4
Kick a to bucket 4

Item

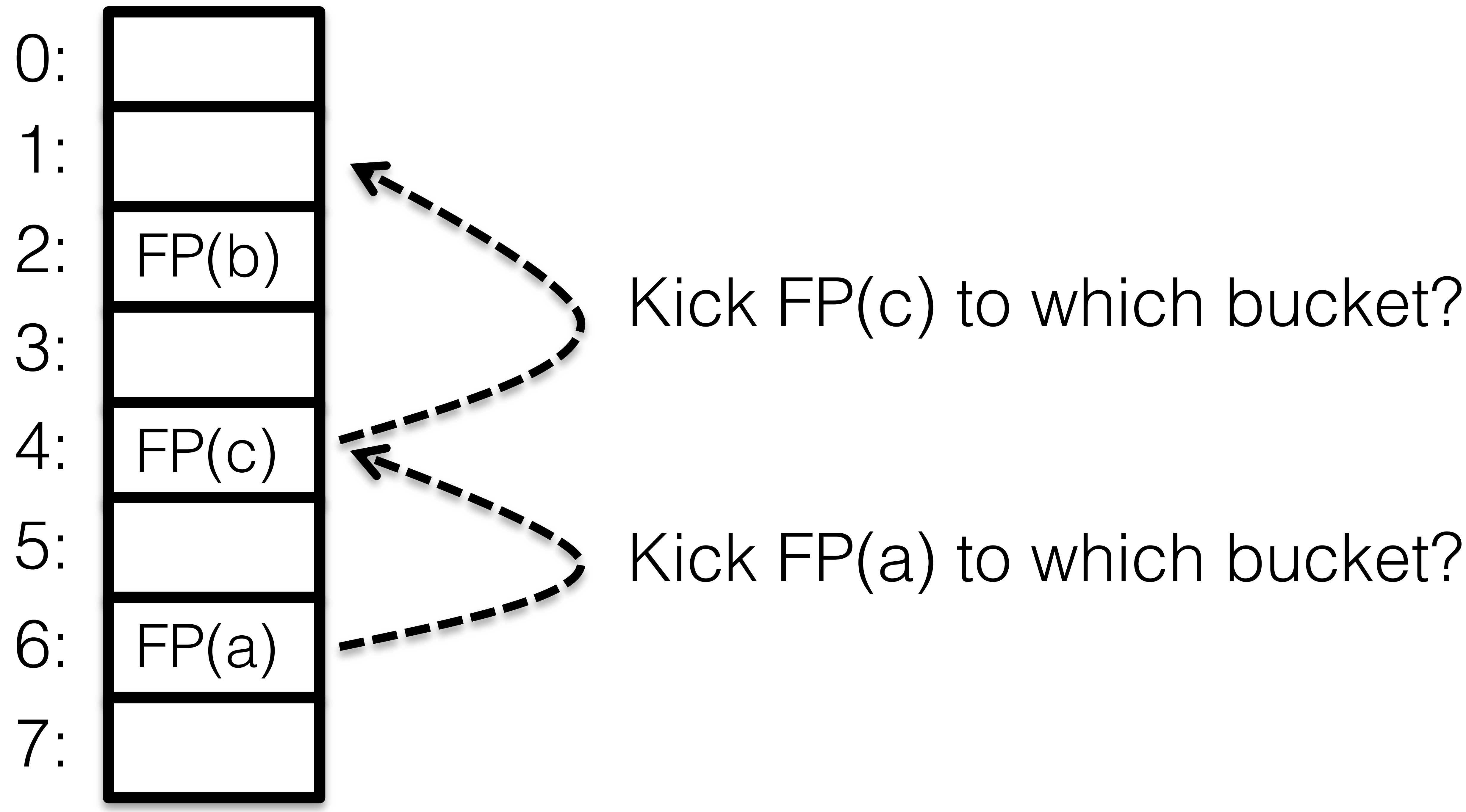

Insert(x)
h2(x)
0:
1:
2:
3:
4:
5:
6:
7:
b
a
x
Rehash c: alternate(c) = 1
Kick c to bucket 1
Rehash a: alternate(a) = 4
Kick a to bucket 4

Item



Insert complete
(or fail if MaxSteps reached)
Rehash c: alternate(c) = 1
Kick c to bucket 1
Rehash a: alternate(a) = 4
Kick a to bucket 4
Insert(x)
$h_2(x)$
0:
1:
2:
3:
4:
5:
6:
7:
c
b
a
x

- Cuckoo hashing requires rehashing and displacing e
items



With only fingerprint,
how to calculate item's alternate bucket?

- Standard Cuckoo Hashing: two independ hash functions for two buckets

  $\texttt{bucket1 = hash}_1\texttt{(x)}$

  $\texttt{bucket2 = hash}_2\texttt{(x)}$

- Partial-key Cuckoo Hashing: use one buc and fingerprint to derive the other [Fan2013

  $\texttt{bucket1 = hash(x)}$

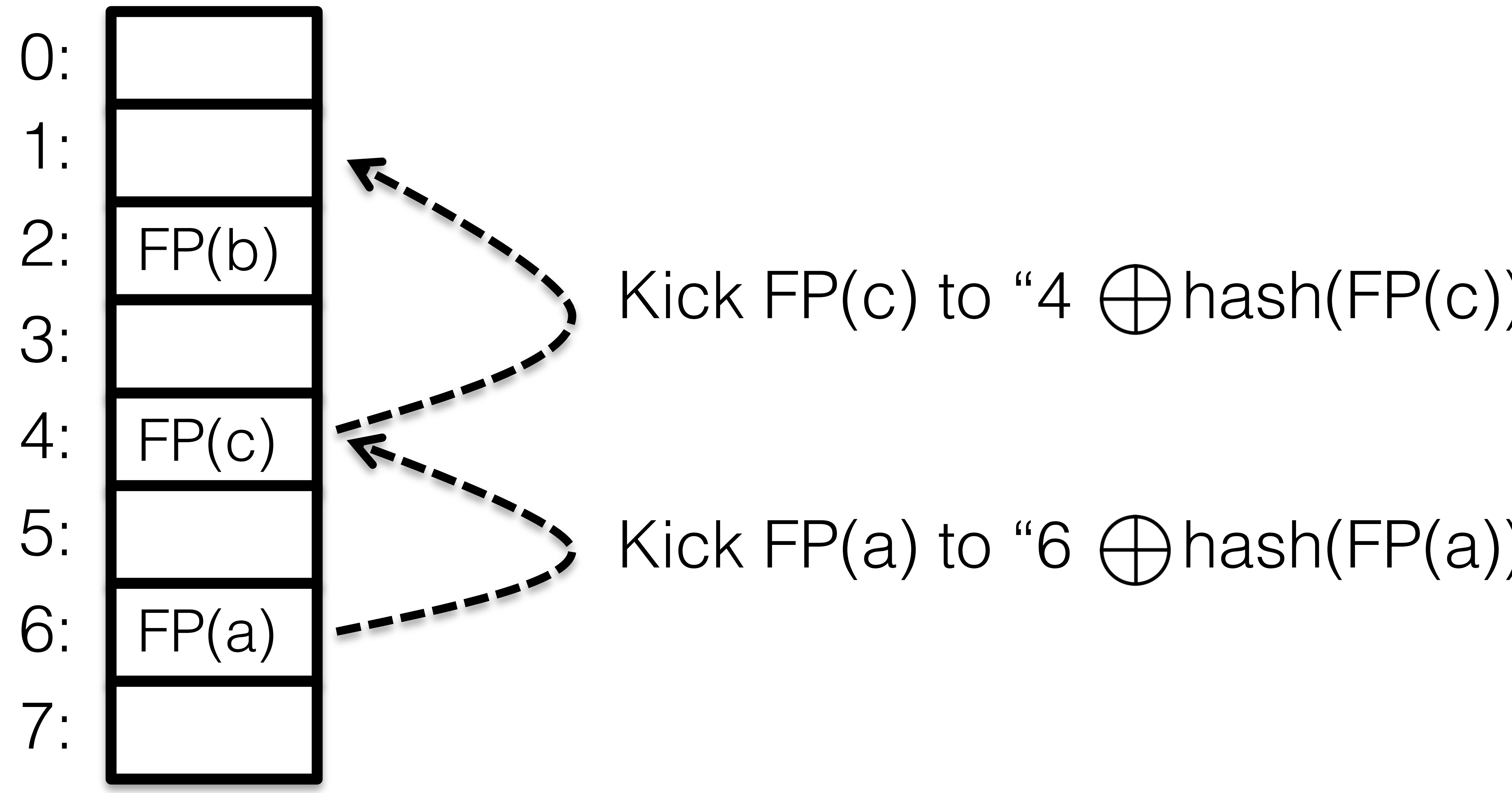
  $\texttt{bucket2 = bucket1} \oplus \texttt{hash(FP(x))}$

  To displace existing fingerprint:

  $\texttt{alternate(x) = current(x)} \oplus \texttt{hash(F}$

[Fan2013] MemC3: Compact and Concurrent Mem with Dumber Caching and Smarter Hashing

- Perform cuckoo hashing on fingerprint



Can we still achieve high space utilization with partial-key cuckoo hashing?

Algorithm 1: `Insert(x)`

```
f = fingerprint(x);
i1 = hash(x);
i2 = i1 ⊕ hash(f);
if bucket[i1] or bucket[i2] has an empty entry then
    add f to that bucket;
    return Done;
// must relocate existing items;
i = randomly pick i1 or i2;
for n = 0; n < MaxNumKicks; n++ do
    randomly select an entry e from bucket[i];
    swap f and the fingerprint stored in entry e;
    i = i ⊕ hash(f);
    if bucket[i] has an empty entry then
        add f to bucket[i];
        return Done;
// Hashtable is considered full;
return Failure;
```

Algorithm 2: `Lookup(x)`

f = fingerprint(x);
i_1 = hash(x);
$i_2 = i_1 \oplus$ hash(f);
if bucket[i_1] or bucket[i_2] has f **then**
    return `True`;
return `False`;

Algorithm 3: `Delete(x)`

f = fingerprint(x);

i_1 = hash(x);

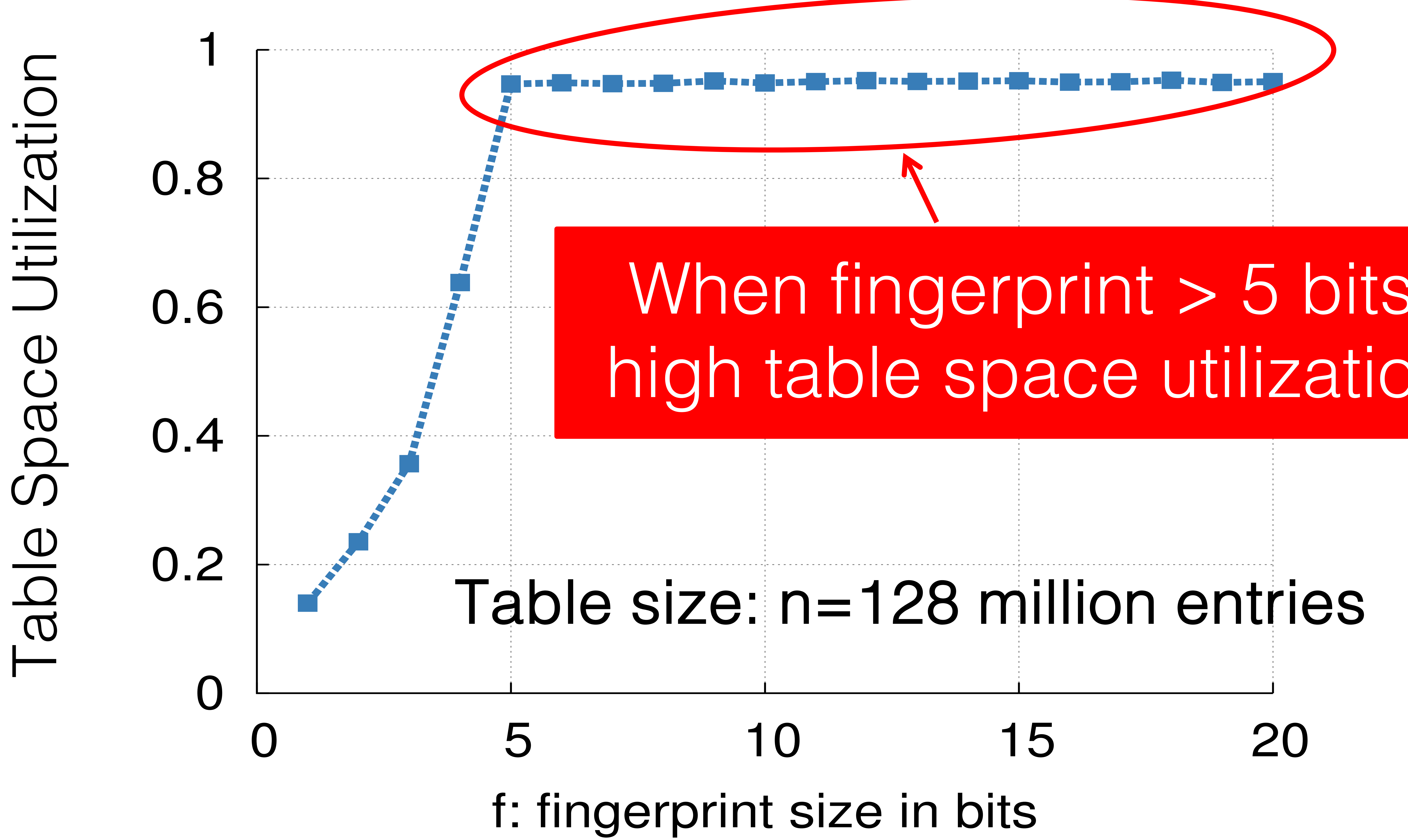
$i_2 = i_1 \oplus$ hash(f);

if bucket[i_1] or bucket[i_2] has f **then**

  remove a copy of f from this bucket;

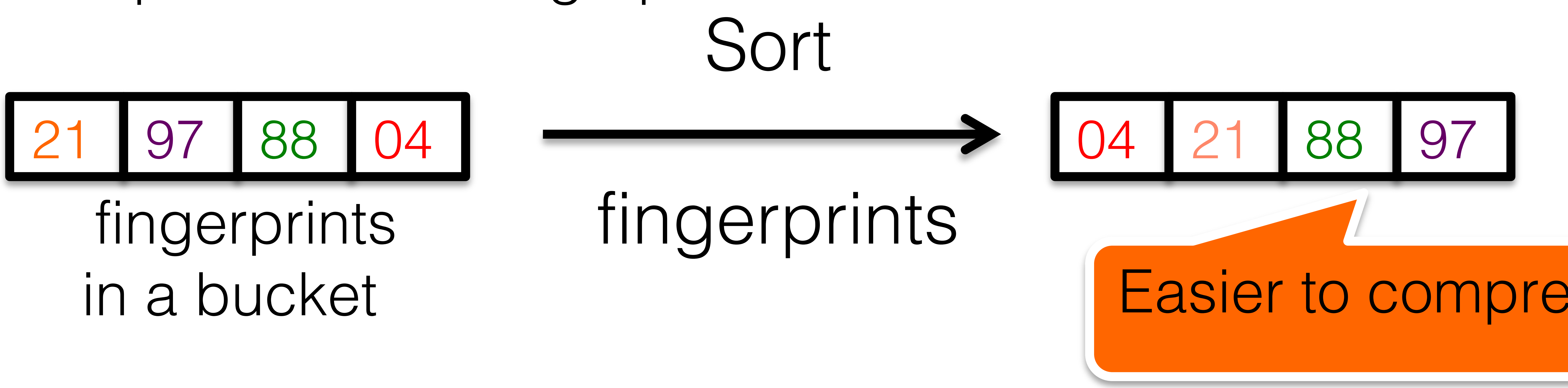
  return `True`;

return `False`;



- Fingerprint must be $\Omega(\log(n)/b)$ bits in theory
  - n: hash table size, b: bucket size

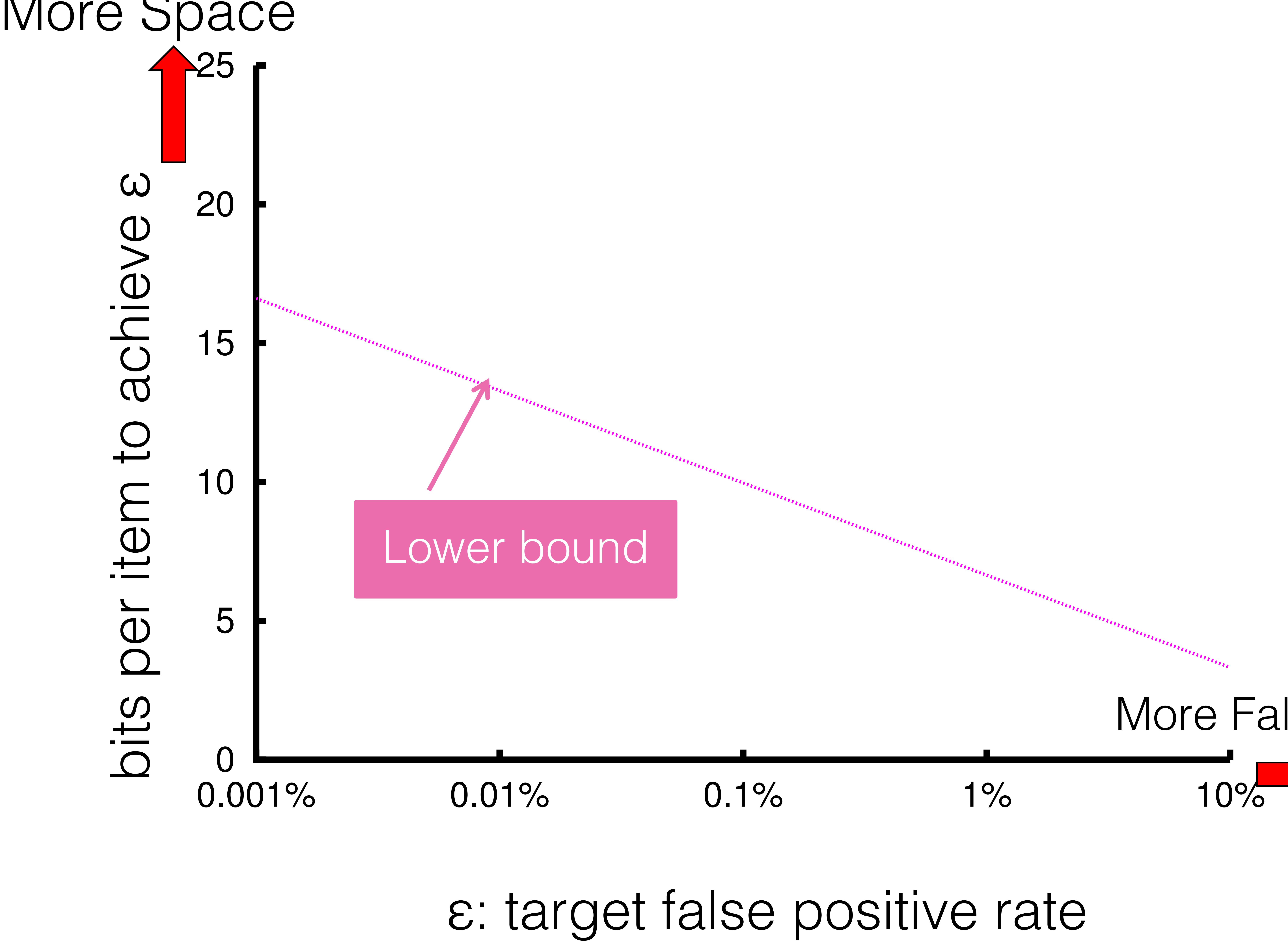
- Based on observation:
  - A monotonic sequence of integers is easier to compress[Bonomi2006]
- Semi-Sorting:
  - Sort fingerprints sorted in each bucket
  - Compress sorted fingerprints

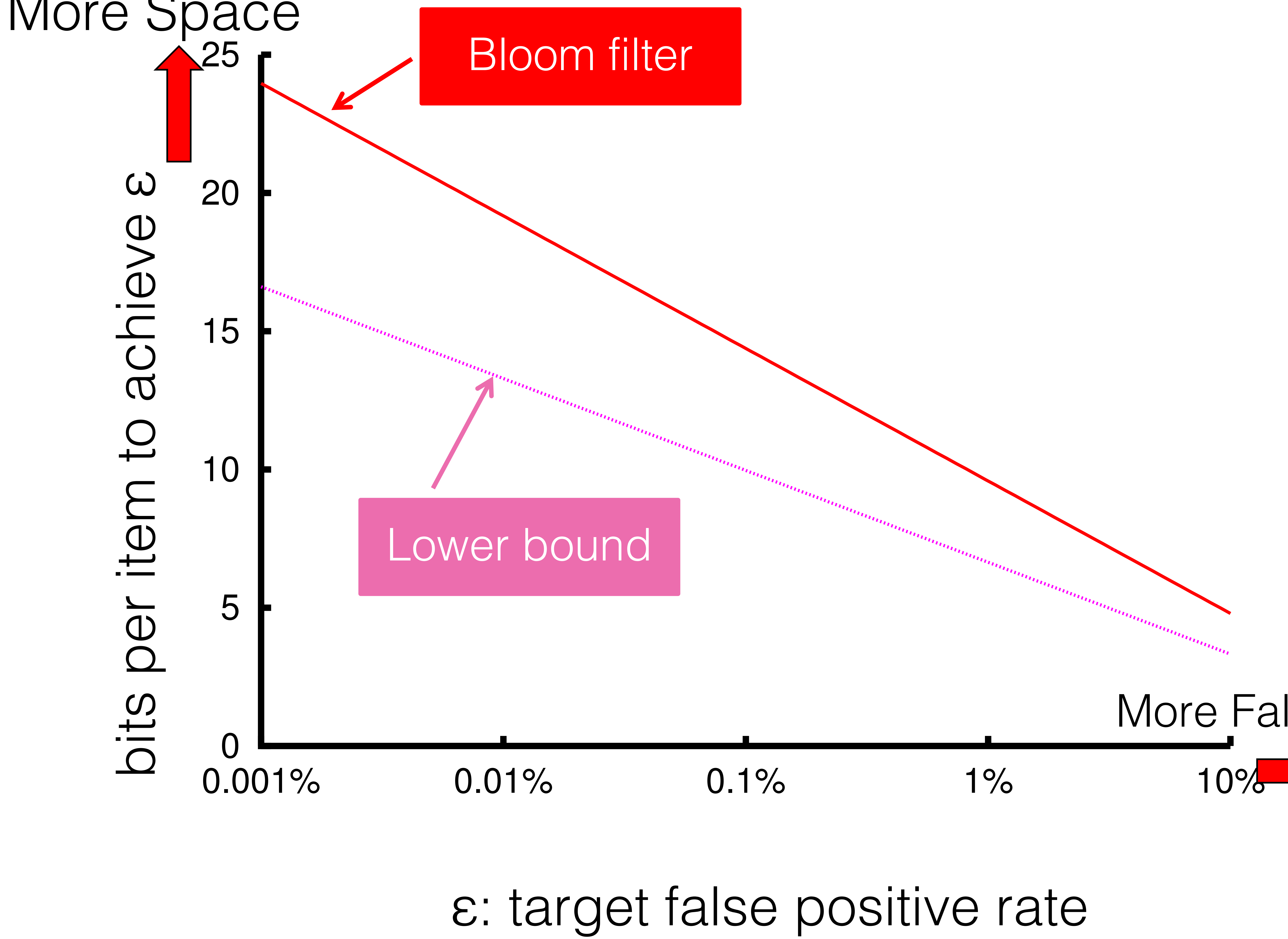


+ For 4-way bucket, save one bit per item

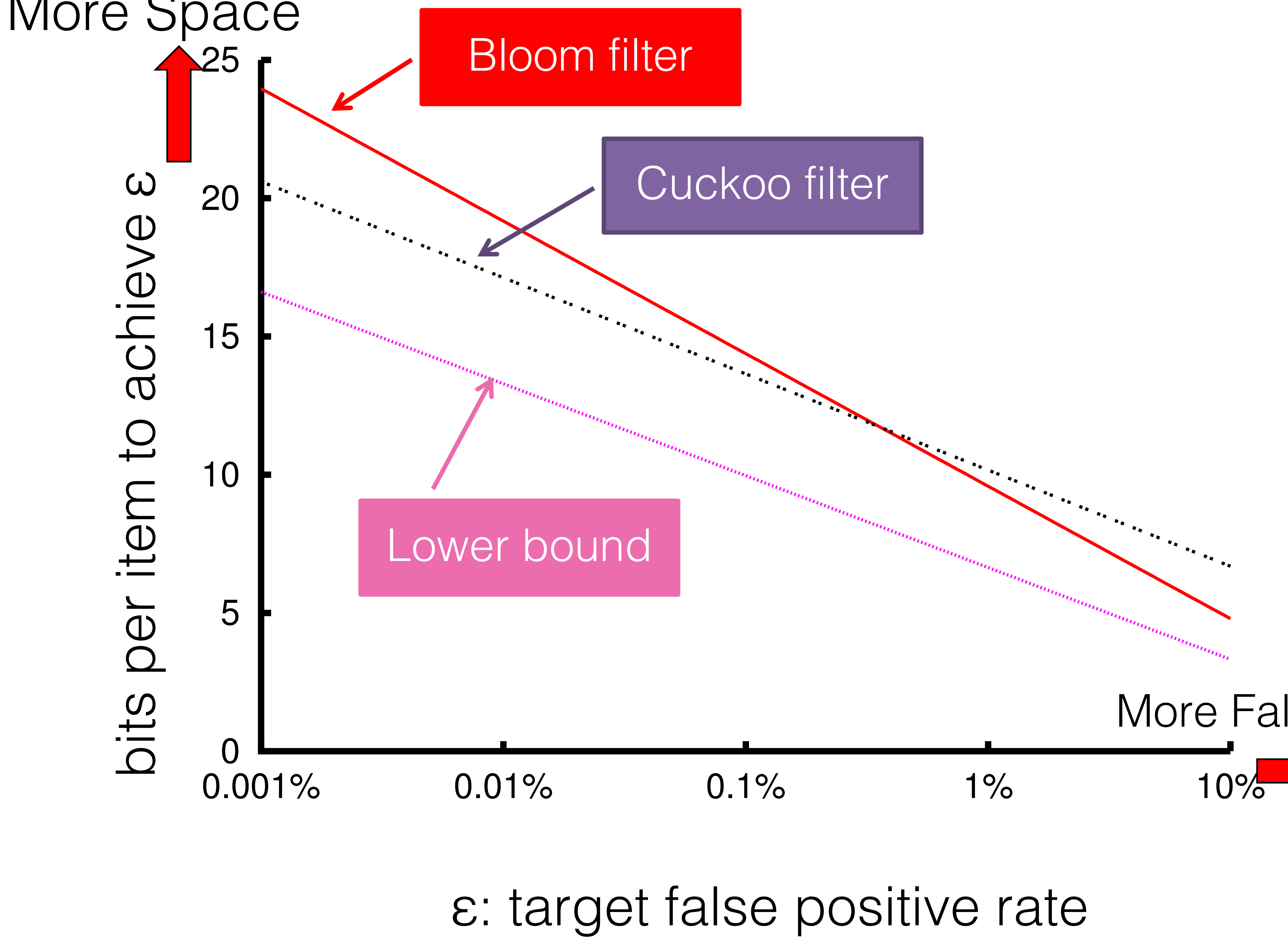
-- Slower lookup / insert

[Bonomi2006] Beyond Bloom filters: From approximate membership checks t
proximate state machines.

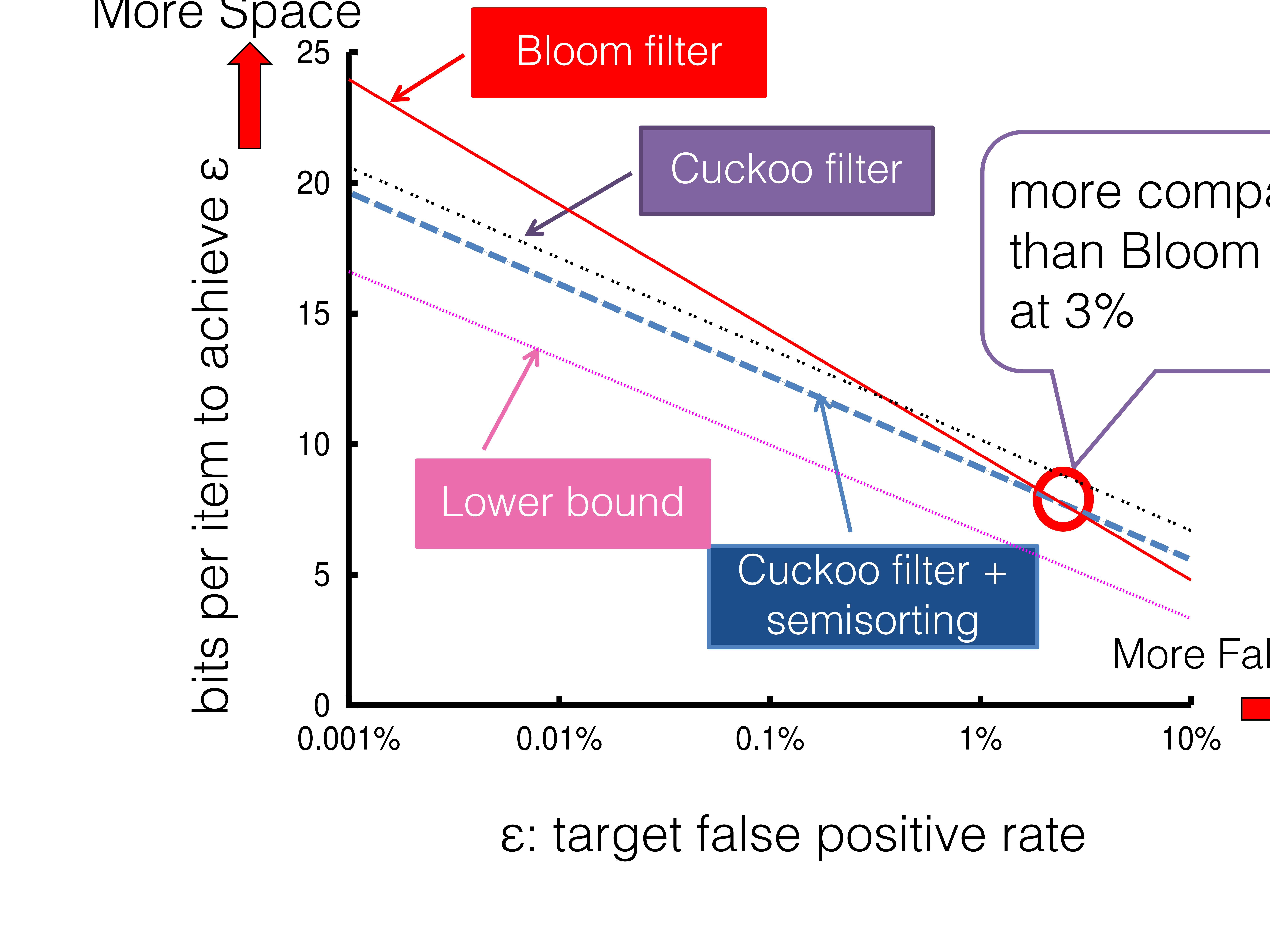

More Space
25
20
15
10
5
0
bits per item to achieve ε
Lower bound
More Fal
0.001%
0.01%
0.1%
1%
10%
ε: target false positive rate



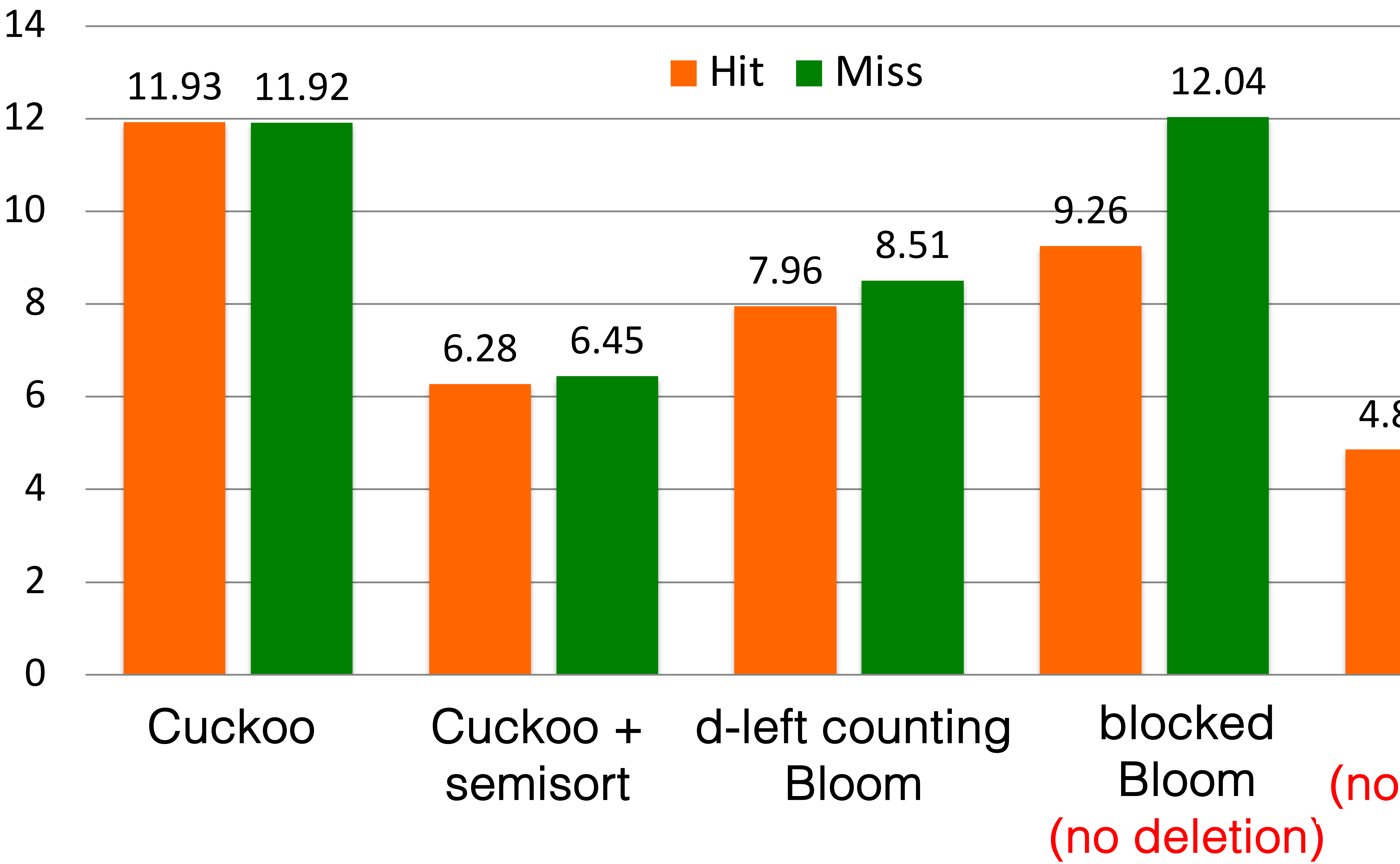
More Space
Bloom filter
Lower bound
More Fal
bits per item to achieve ε
ε: target false positive rate
25
20
15
10
5
0
0.001%
0.01%
0.1%
1%
10%



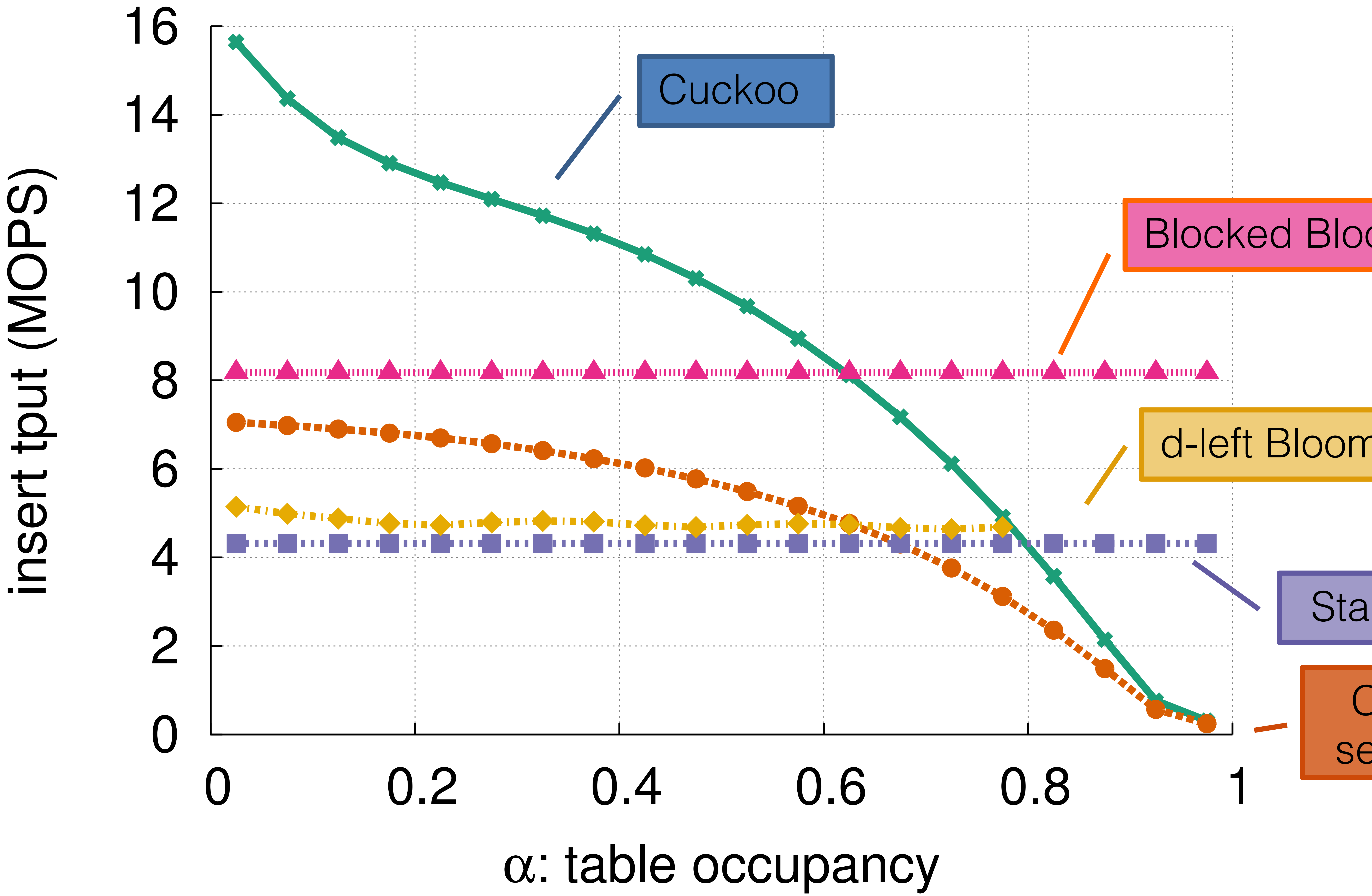
More Space
Bloom filter
Cuckoo filter
Lower bound
More Fal
bits per item to achieve ε
25
20
15
10
5
0
0.001%
0.01%
0.1%
1%
10%
ε: target false positive rate



More Space
Bloom filter
Cuckoo filter
more compa
than Bloom
at 3%
Lower bound
Cuckoo filter + semisorting
More Fal
bits per item to achieve ε
25
20
15
10
5
0
0.001%
0.01%
0.1%
1%
10%
ε: target false positive rate



Cuckoo filter is among the fastest regardless worklo



Cuckoo filter has decreasing insert rate, but over
is only slower than blocked Bloom filter.

References:

- Mining massive Datasets by Leskovec, Rajaraman, Ullman, Chapter 4.

- Primary reference for this lecture
  - Survey on Bloom Filter, Broder and Mitzenmacher 2005, https://www.eecs.harvard.edu/~michaelm/postscripts/im2005b.pdf

- Others
  - Randomized Algorithms by Mitzenmacher and Upfal.

- Cuckoo filter: Fan, Bin, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. "Cuckoo filter: Practically better than bloom." In Proceedings the 10th ACM International on Conference on emerging Networking Experiments and Technologies, pp. 75-88. 2014.