CS2040S
Data Structures and Algorithms

Hashing IV

Competition: Speed Demon!

Simple data processing…. as fast as you can.

Problem Set: Automatic Writing!

Produce your own magnum opus, automatically!

Jaquet Droz: The Writer

Hashing overview

- What is a hash function?
- Collision resolution: chaining and open addressing
- Java hashing
- Table (re)sizing
- Sets

Abstract Data Types

Symbol Table

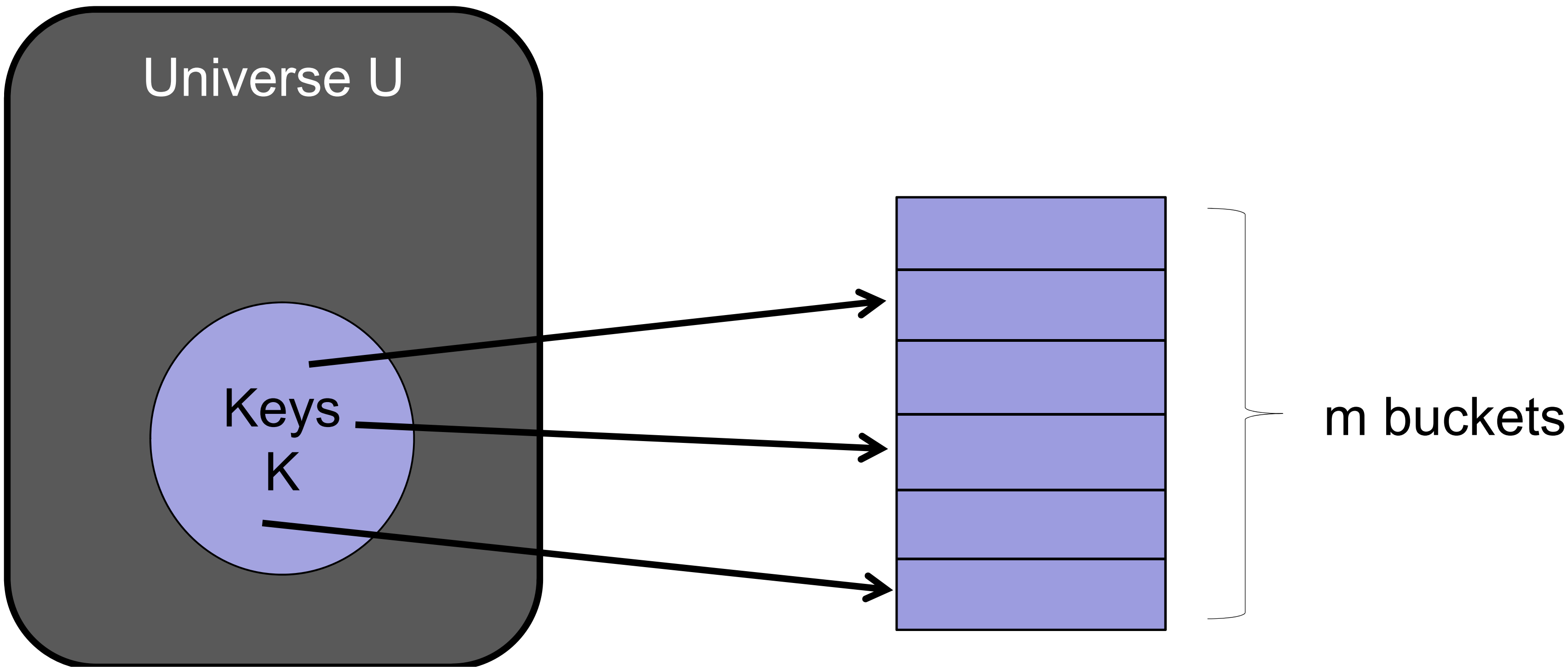
```
public interface  SymbolTable
-----------------------------------------------------------------
        void  insert(Key k, Value v)  insert (k,v) into table
       Value  search(Key k)           get value paired with k
        void  delete(Key k)           remove key k (and value)
     boolean  contains(Key k)         is there a value for k?
         int  size()                  number of (k,v) pairs
-----------------------------------------------------------------
```

Note: no successor / predecessor queries.

Hash Functions

Problem:

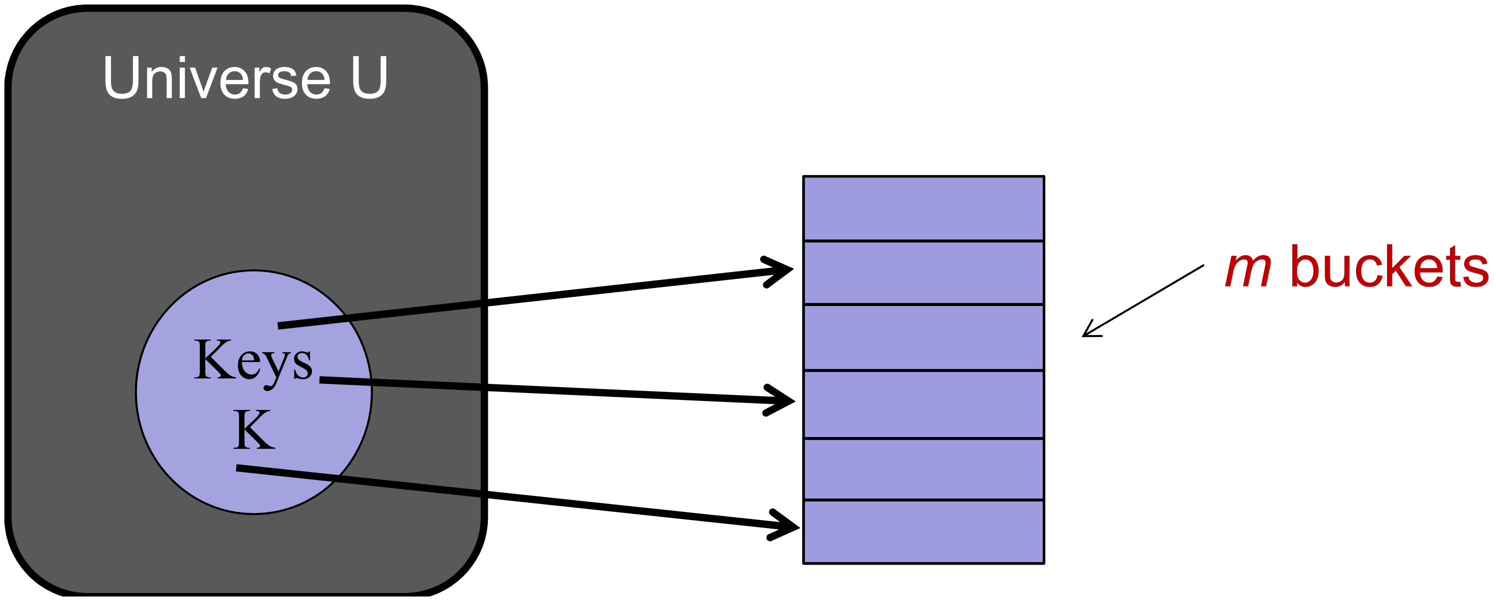
- Huge universe U of possible keys.
- Smaller number n of actual keys.
- How to map n keys to $m \approx n$ buckets?



Hash Functions

Define hash function $h : U \rightarrow \{1..m\}$

- Store key k in bucket $h(k)$.



Hash Functions

Collisions:

- We say that two distinct keys k_1 and k_2 collide if:

$$h(k_1) = h(k_2)$$

- Unavoidable!
  - The table size is smaller than the universe size.
  - The pigeonhole principle says:
    - There must exist two keys that map to the same bucket.
    - Some keys must collide!

Resolving Collisions

- Basic problem:
  - What to do when two items hash to the same bucket?
- Solution 1: Chaining
  - Insert item into a linked list.
- Solution 2: Open Addressing
  - Find another free bucket.

Table Size

How large should the table be?

- Assume: Hashing with Chaining
- Assume: Simple Uniform Hashing
- Expected search time: $O(1 + n/m)$
- Optimal size: $m = \Theta(n)$

  - if $(m < 2n)$: too many collisions.
  - if $(m > 10n)$: too much wasted space.

- Problem: we don't know n in advance.

Table Size

Idea:

- Start with small (constant) table size.
- Grow (and shrink) table as necessary.

Example:

- Initially, $m = 10$.
- After inserting 6 items, table too small! Grow…
- After deleting n-1 items, table too big! Shrink…

Table Size

How to grow the table:

1. Choose new table size m.
2. Choose new hash function h.

Not like Java hashCode!

   - Hash function depends on table size!
   - Remember: $h : U \rightarrow \{1..m\}$
3. For each item in the old hash table:
   - Compute new hash function.
   - Copy item to new bucket.

Table Size

Time complexity of growing the table:

- Assume:
  - Let m_1 be the size of the old hash table.
  - Let m_2 be the size of the new hash table.
  - Let n be the number of elements in the hash table.
- Costs:
  - Scanning old hash table: $O(m_1)$
  - Creating new hash table: $O(m_2)$
  - Inserting each element in new hash table: $O(1)$
  - Total: $O(m_1 + m_2 + n)$

How fast to grow?

Idea 1: Increment table size by 1

- if ($n == m$): $m = m+1$

- Cost of resize:
  - Size $m_1 = n$.
  - Size $m_2 = n+1$.
  - Total: $O(n)$

How fast to grow?

Idea 1: Increment table size by 1

- When $(n == m)$: $m = m+1$
- Cost of each resize: $O(n)$

Table size	8	8	9	10	11	12	…	n+1
Number of items	0	7	8	9	10	11	…	n
Number of inserts		7	1	1	1	1	…	1
Cost		7	8	9	10	11		n

- Total cost: $(7 + 8 + 9 + 10 + 11 + \ldots + n) = O(n^2)$

How fast to grow?

Idea 2: Double table size

- if $(n == m)$: $m = 2m$

- Cost of resize:
  - Size $m_1 = n$.
  - Size $m_2 = 2n$.
  - Total: $O(n)$

How fast to grow?

Idea 2: Double table size

– When ($n == m$): $m = 2m$
– Cost of each resize: $O(n)$

Table size	8	8	16	16	16	16	16	16	16	16	32	32	32	…	2n
# of items	0	7	8	9	10	11	12	13	14	15	16	17	18	…	n
# of inserts		7	1	1	1	1	1	1	1	1	1	1	1	…	1
Cost		7	8	1	1	1	1	1	1	1	16	1	1		n

– Total cost: $(7 + 15 + 31 + \ldots + n) = O(n)$

How fast to grow?

Idea 2: Double table size

- if ($n == m$): $m = 2m$

    - Cost of resize: O(n)
    - Cost of inserting n items + resizing: O(n)

- Most insertions: O(1)
- Some insertions: linear cost (expensive)
- Average cost: O(1)

How fast to grow?

Idea 3: Square table size

- When ($n == m$): $m = m^2$

Table size	Total Resizing Cost
8	?
64	?
4,096	?
16,777,216	?
…	…
m	?

How fast to grow?

Idea 3: Square table size

- if $(n == m)$: $m = m^2$

- Cost of resize:
  - Size $m_1 = n$.
  - Size $m_2 = n^2$.
  - Total: $O(m_1 + m_2 + n)$

    $= O(n + n^2 + n)$

    $= O(n^2)$

How fast to grow?

Idea 3: Square table size

- if ($n == m$): $m = m^2$

- Cost of resize:
  - Total: $O(n^2)$

- Cost of inserts:
  - Total: $O(n)$

How fast to grow?

Best (so far): Double table size

- if ($n == m$): $m = 2m$

- Cost of resize:
  - Size $m_1 = n$.
  - Size $m_2 = 2n$.
  - Total: $O(n)$

Deleting Elements

Basic procedure: (chained hash tables)

Delete(*key*)

1. Calculate hash of *key*.
2. Let L be the linked list in the specified bucket.
3. Search for item in linked list L.
4. Delete item from linked list L.

Cost:

– Total: $O(1 + n/m)$

Deleting Elements

What happens if too many items are deleted?

- Table is too big!
- Shrink the table…

- Try 1:
  - If $(n == m)$, then $m = 2m$.
  - If $(n < m/2)$ then $m = m/2$.

Deleting Elements

Rules for shrinking and growing:

- Try 1:
  - If $(n == m)$, then $m = 2m$.
  - If $(n < m/2)$ then $m = m/2$.

- Example problem:
  - Start: n=100, m=200
  - Delete: n=99, m=200 → shrink to m=100
  - Insert: n=100, m=100 → grow to m=200
  - Repeat…

Deleting Elements

Example execution:

- Start: n=100, m=200
- cost=100 • Delete: n=99, m=200 → shrink to m=100
- cost=100 • Insert: n=100, m=100 → grow to m=200
- cost=100 • Delete: n=99, m=200 → shrink to m=100
- cost=100 • Insert: n=100, m=100 → grow to m=200
- cost=100 • Delete: n=99, m=200 → shrink to m=100
- cost=100 • Insert: n=100, m=100 → grow to m=200
- Repeat…

Deleting Elements

Rules for shrinking and growing:

- Try 2:
  - If $(n == m)$, then $m = 2m$.
  - If $(n < m/4)$, then $m = m/2$.

- Is this right?
- How do we decide whether this works?

Deleting Elements

Rules for shrinking and growing:

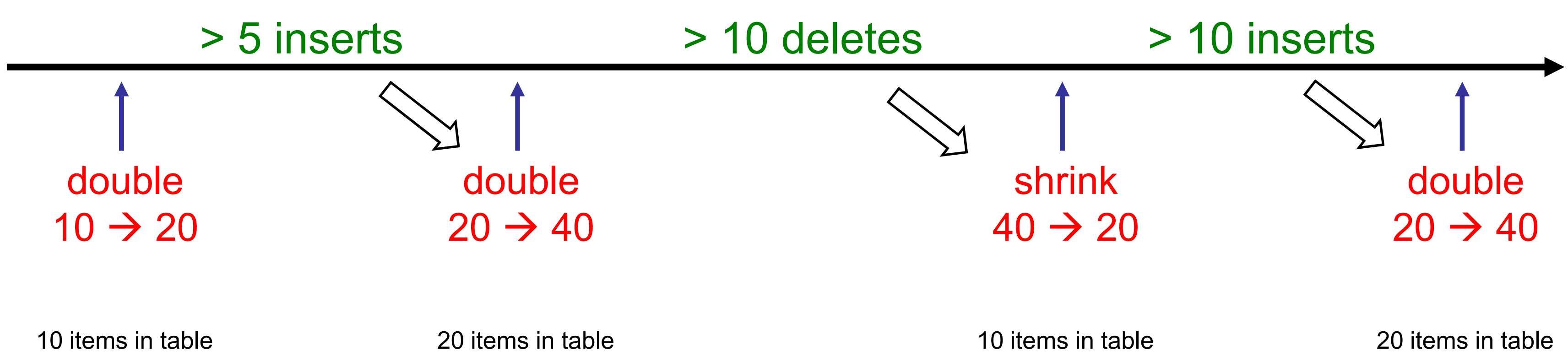
- Try 2:
  - If $(n == m)$, then $m = 2m$.
  - If $(n < m/4)$, then $m = m/2$.

- Claim:
  - Every time you double a table of size m, at least $m/2$ new items were added.
  - Every time you shrink a table of size m, at least $m/4$ items were deleted.

Deleting Elements

Claim:

- Every time you double a table of size m, at least $m/2$ new items were added.
- Every time you shrink a table of size m, at least $m/4$ items were deleted.



Amortized Analysis

Technique for analyzing "average" cost:

- Common in data structure analysis
- Like paying rent:
  - You don't pay rent every day!
  - Pay \$900/month = \$30/day.

Definition:

- Operation has <u>amortized cost</u> $\mathrm{T}(n)$ if for every integer k, the cost of k operations is $\leq k\,\mathrm{T}(n)$

Amortized Analysis

Definition:

– Operation has <u>amortized cost</u> $T(n)$ if for every integer k, the cost of k operations is $\leq k\,T(n)$

Example: amortized cost = 7

1. insert: 5
2. insert: 5
3. insert: 5
4. insert: 13
5. insert: 7

5 <= 7

5+5 <= 2*7 = 14

5+5+5 <= 3*7 = 21

5+5+5+13 <= 4*7 = 28

5+5+5+13+7 <= 5*7 = 35

Amortized Analysis

"amortized" is NOT "average"

Definition:

– Operation has <u>amortized cost</u> $T(n)$ if for every integer k, the cost of k operations is $\leq k\,T(n)$

Example: amortized cost **NOT** 7

1. insert: 13
2. insert: 5
3. insert: 5
4. insert: 5
5. insert: 7

13 > 7

13+5 > 2*7 = 14

13+5+5 > 3*7 = 21

13+5+5+5 <= 4*7 = 28

5+5+5+13+7 <= 5*7 = 35

Amortized Analysis

Definition:

- Operation has amortized cost $T(n)$ if for every integer k, the cost of k operations is $\leq k\,T(n)$

Example: (Hash Tables)

- Inserting k elements into a hash table takes time $O(k)$.
- Conclusion:

  The insert operation has amortized cost $O(1)$.

Amortized Analysis

Accounting Method (paying rent)

- Imagine a bank account **B**.
- Each operation adds money to the bank account.
- Every step of the algorithm spends money:
  - Immediate money: to perform the operation.
  - Deferred money: from the bank account.
- Total cost execution = total money
  - Average time / operation = money / num. ops

Amortized Analysis

Accounting Method Example (Hash Table)

- Each table has a bank account.
- Each time an element is added to the table, it adds O(1) dollars to the bank account, uses O(1) dollars to insert element.
- A table with *k* new elements since last resize has *k* dollars in bank.

0	null
1	null
2	**(k_1, A)**
3	null
4	null
5	null
6	null
7	null
8	**(k_2, B)**
9	null

Bank account
$2 dollars

Amortized Analysis

Accounting Method Example (Hash Table)

- Each table has a bank account.
- Each time an element is added to the table, it adds O(1) dollars to the bank account.

- Claim:
  - Resizing a table of size m takes $O(m)$ time.
  - If you resize a table of size m, then:
    - at least $m/2$ new elements since last resize
    - bank account has $\Theta(m)$ dollars.

Amortized Analysis

Accounting Method Example (Hash Table)

- Each table has a bank account.
- Each time an element is added to the table, it adds O(1) dollars to the bank account.
- Pay for resizing from the bank account!
- Strategy:
  - Analyze inserts ignoring cost of resizing.
  - Ensure that bank account always is big enough to pay for resizing.

Amortized Analysis

Total cost: Inserting *k* elements costs:

- Deferred dollars: O(*k*)   (to pay for resizing)
- Immediate dollars: O(*k*) for inserting elements in table
- Total (Deferred + Immediate): O(*k*)

Amortized Analysis

Total cost: Inserting k elements costs:

- Deferred dollars: $O(k)$ (to pay for resizing)
- Immediate dollars: $O(k)$ for inserting elements in table
- Total (Deferred + Immediate): $O(k)$

Cost per operation:

- Deferred dollars: $O(1)$
- Immediate dollars: $O(1)$
- Total: $O(1)$ / per operation

Example: Binary Counter

Counter ADT:

- increment()
- read()

0	0	0	0	0	0	0	0	0	0

Example: Binary Counter

Counter ADT:

- increment()
- read()

increment()

0	0	0	0	0	0	0	0	0	1

Example: Binary Counter

Counter ADT:

- increment()
- read()

increment(), increment()

0	0	0	0	0	0	0	0	1	0

Example: Binary Counter

Counter ADT:

- increment()
- read()

increment(), increment(), increment()

0	0	0	0	0	0	0	0	1	1

What is the worst-case cost of incrementing a counter with max-value n?

1. O(1)
2. O(log n)
3. O(n)
4. $O(n^2)$
5. I have no idea.

Example: Binary Counter

Counter ADT:

- increment()
- read()

Some increments are expensive…

0	1	1	1	1	1	1	1	1	1

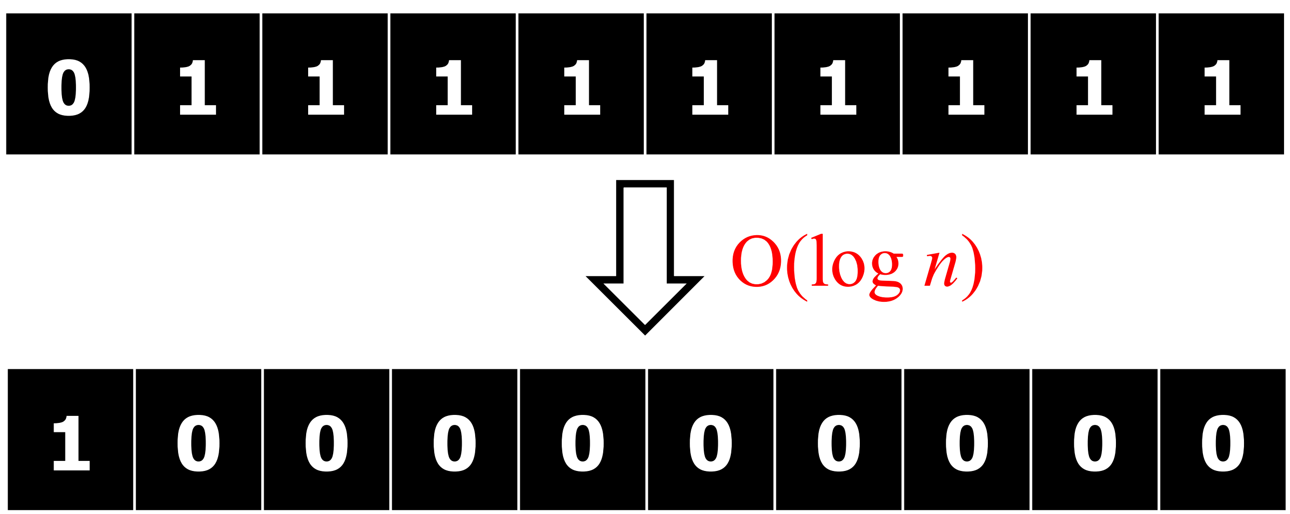
$O(\log n)$

1	0	0	0	0	0	0	0	0	0

Example: Binary Counter

Question: If we increment the counter to *n*, what is the amortized cost per operation?

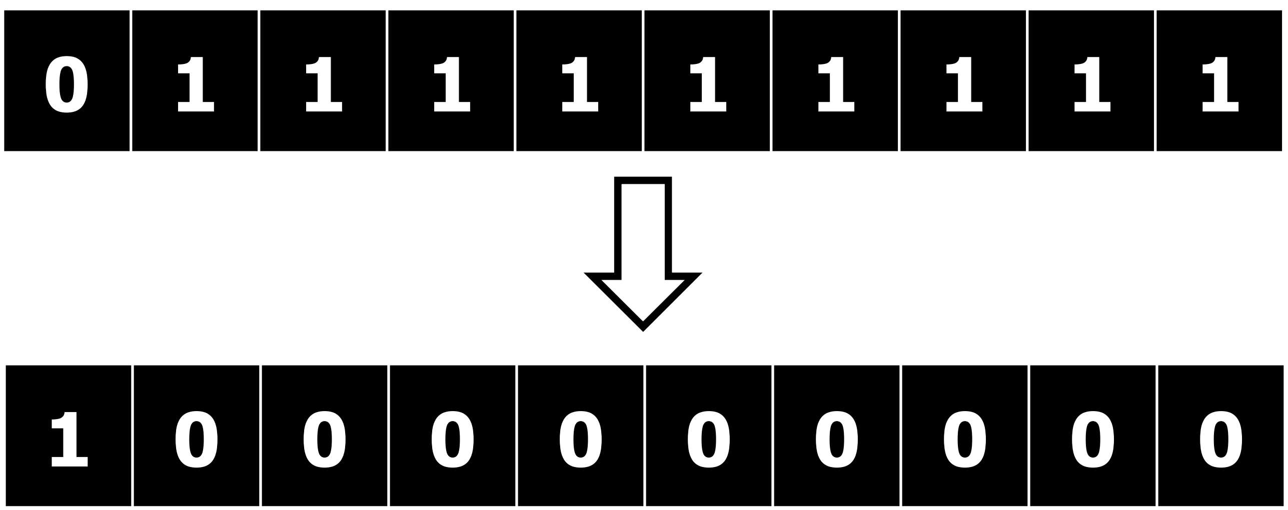
- Easy answer: O(log *n*)
- More careful analysis….



Example: Binary Counter

Observation:

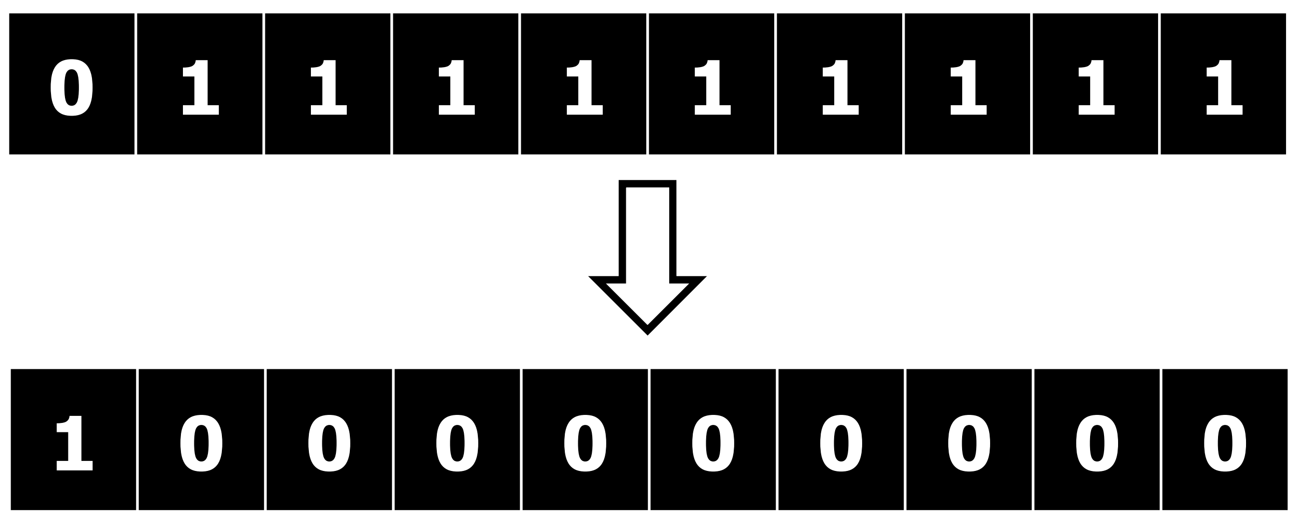
During each increment, only <u>one</u> bit is changed from: 0 → 1



Example: Binary Counter

Observation:

During each increment, <u>many</u> bits may be changed from: 1 → 0

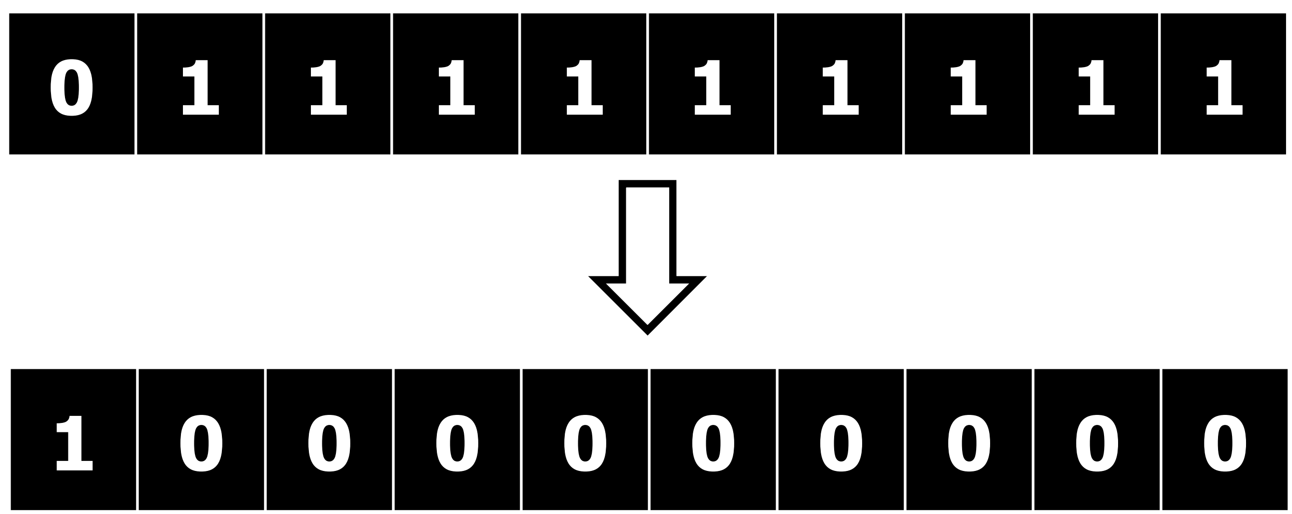


Example: Binary Counter

Observation:

Accounting method: each bit has a bank account.

Whenever you change it from 0→1, add one dollar.



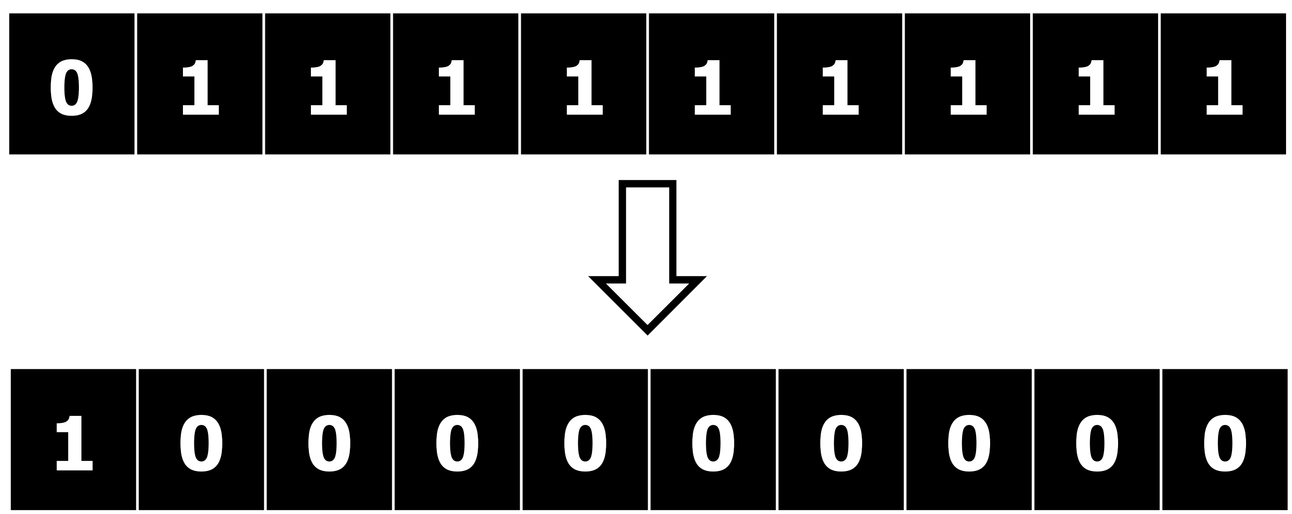
Example: Binary Counter

Observation:

Accounting method: each bit has a bank account.

Whenever you change it from 0→1, add one dollar.

Whenever you change it from 1→0, pay one dollar.



Example: Binary Counter

Counter ADT

0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
0	0	0	0	0	0	0	0	0	0

Example: Binary Counter

Counter ADT

increment()

0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
0	0	0	0	0	0	0	0	0	0

$\Downarrow$

0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**
0	0	0	0	0	0	0	0	0	1

Example: Binary Counter

Counter ADT

increment(), increment()

0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**

0 0 0 0 0 0 0 0 0 1

0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**

0 0 0 0 0 0 0 0 1 0

Example: Binary Counter

Counter ADT

increment(), increment(), increment()

0	0	0	0	0	0	0	0	1	0

0 0 0 0 0 0 0 0 1 0

0	0	0	0	0	0	0	0	1	1

0 0 0 0 0 0 0 0 1 1

Example: Binary Counter

Counter ADT

increment()

0	1	1	1	1	1	1	1	1	1

0 1 1 1 1 1 1 1 1 1

$\Downarrow$

1	0	0	0	0	0	0	0	0	0

1 0 0 0 0 0 0 0 0 0

Example: Binary Counter

Observation:

Amortized cost of increment: 2

– One operation to switch <u>one</u> 0→1
– One dollar (for bank account of switched bit).

(All switches from 1→0 paid for by bank account.)

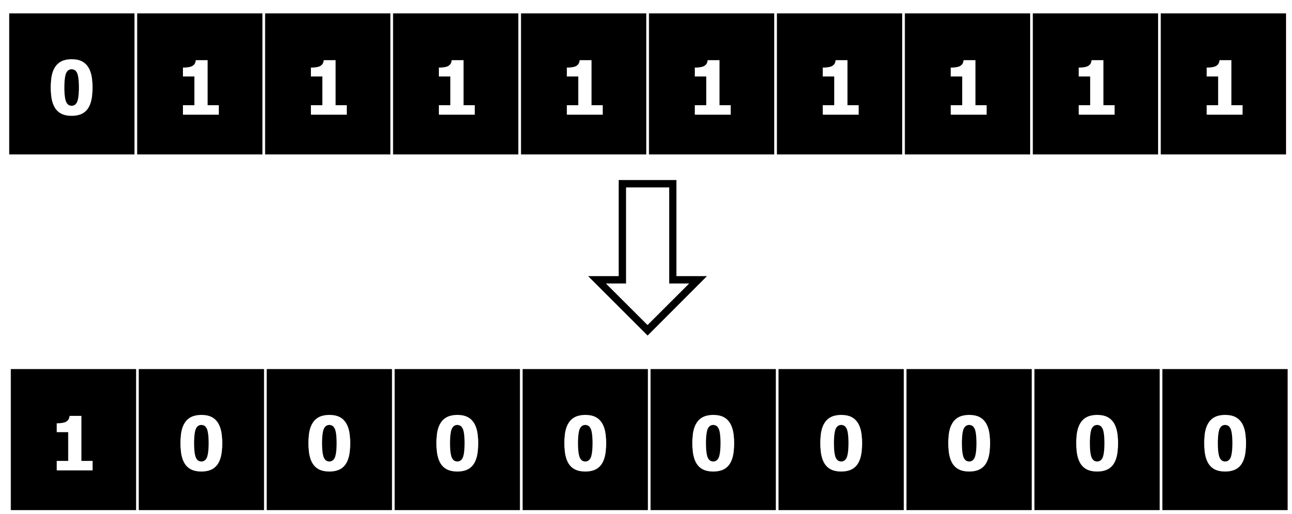


Table Size Rules

Rules for shrinking and growing:

- If $(n == m)$, then $m = 2m$.
- If $(n < m/4)$, then $m = m/2$.

– Claim:

- Every time you double a table of size m, at least $m/2$ new items were added.
- Every time you shrink a table of size m, at least $m/4$ items were deleted.

Amortized Analysis

Accounting Method

- Each table has a bank account.
- Each time an element is added to the table, it adds O(1) dollars to the bank account.

- Claim:
  - Resizing a table of size m takes $O(m)$ time.
  - If you resize a table of size m, then:
    - at least $m/2$ new elements since last resize
    - bank account has $\Theta(m)$ dollars.

Amortized Analysis

Total cost: Inserting *k* elements costs:

- Deferred dollars: O(*k*)   (to pay for resizing)
- Immediate dollars: O(*k*) for inserting elements in table
- Total (Deferred + Immediate): O(*k*)

Cost per operation:

- Deferred dollars: O(1)
- Immediate dollars: O(1)
- Total: O(1) / per operation

Hash Table Resizing

Conclusion: Hashing with Chaining

– with Simple Uniform Hashing Assumption (SUHA)

Cost per operation:

– Insert operation: amortized O(1)

– Search operation: expected O(1)

Notes:

– Inserts are amortized because of table resizing.

– Inserts are not randomized (because no searching for duplicates).

– Searches are expected (but not amortized) since no resizing on a search.

Hashing overview

- What is a hash function?
- Collision resolution: chaining and open addressing
- Java hashing
- Table (re)sizing
- Sets

Abstract Data Type

Symbol Table

```
public interface  SymbolTable<Key, Value>
```

`void`	`insert(Key k, Value v)`	*insert (k,v) into table*
`Value`	`search(Key k)`	*get value paired with k*
`void`	`delete(Key k)`	*remove key k (and value)*
`boolean`	`contains(Key k)`	*is there a value for k?*
`int`	`size()`	*number of (k,v) pairs*

Note: no successor / predecessor queries.

Abstract Data Type

Set

```
public class  Set<Key>
```

void	insert(Key k)	*Insert k into set*
boolean	contains(Key k)	*Is k in the set?*
void	delete(Key k)	*Remove key k from the set*
void	intersect(Set<Key> s)	*Take the intersection.*
void	union(Set<Key> s)	*Take the union.*

Properties:

- No defined ordering.
- Speed is critical.
- Space is critical.

Abstract Data Type

Set

```
public class  Set<Key>
--------------------------------------------------------------------------
         void  insert(Key k)               Insert k into set
      boolean  contains(Key k)             Is k in the set?
         void  delete(Key k)               Remove key k from the set
         void  intersect(Set<Key> s)       Take the intersection.
         void  union(Set<Key> s)           Take the union.
--------------------------------------------------------------------------
```

Java: HashSet<…> implements Set<…>

A few examples

Facebook:

- I have a list of (names) of friends:
  - John
  - Mary
  - Bob
- Some are online, some are offline.
- How do I determine which are on-line and which are off-line?

Maintain a set of online (or offline) friends…

A few examples

Spam filter:

- I have a list bad e-mail addresses:
  - @ mxkp322ochat.com
  - @ info.dhml212oblackboard.net
  - @ transformationalwellness.com
- I have a list of good e-mail addresses:
  - My mom.
  - *.nus.edu.sg
- How do I quickly check for spam?

Maintain a set…

Abstract Data Type

Set

public class	**Set<Key>**	
void	insert(Key k)	*Insert k into set*
boolean	contains(Key k)	*Is k in the set?*
void	delete(Key k)	*Remove key k from the set*
void	intersect(Set<Key> s)	*Take the intersection.*
void	union(Set<Key> s)	*Take the union.*

Solution 1: Implement using a Hash Table

Implementing a Set

Use a hash table:

hash(“`www.gmail.com`”)

hash(“`www.apple.com`”)

hash(“`www.microsoft.com`”)

hash(“`www.nytimes.com`”)

0	**0**
1	**0**
2	**www.gmail.com**
3	**www.apple.com**
4	**0**
5	**0**
6	**www.microsoft.com**
7	**0**
8	**www.nytimes.com**
9	**0**

Which problem does a hash table not solve?

1. Fast insertion
2. Fast deletion
3. Fast lookup
4. Small space
5. All of the above
6. None of the above

A hash table takes **more** space than a simple list!

Implementing a Set

Use a hash table:

hash("`www.gmail.com`")

hash("`www.apple.com`")

hash("`www.microsoft.com`")

hash("`www.nytimes.com`")

0	**0**
1	**0**
2	**www.gmail.com**
3	**www.apple.com**
4	**0**
5	**0**
6	**www.microsoft.com**
7	**0**
8	**www.nytimes.com**
9	**0**

Implementing a Set

Use a hash table:

Why do we store the URL data in the hash table?

hash(“`www.microsoft.com`”)

hash(“`www.nytimes.com`”)

0	**0**
1	**0**
2	**www.gmail.com**
3	**www.apple.com**
4	**0**
5	**0**
6	**www.microsoft.com**
7	**0**
8	**www.nytimes.com**
9	**0**

Implementing a Set

Use a hash table:

Why do we store the URL data in the hash table?

hash

So that we can resolve collisions!

hash

0	0
1	0
2	www.gmail.com
3	www.apple.com
4	0
5	0
6	www.microsoft.com
7	0
8	www.nytimes.com
9	0

Abstract Data Type

Set

public class	Set<Key>	
void	insert(Key k)	*Insert k into set*
boolean	contains(Key k)	*Is k in the set?*
void	delete(Key k)	*Remove key k from the set*
void	intersect(Set<Key> s)	*Take the intersection.*
void	union(Set<Key> s)	*Take the union.*

Solution 2: Implement using a Fingerprint Hash Table

Implementing a Set

Use a fingerprint:

- Only store/send *m* bits!

hash(“`www.gmail.com`”)

hash(“`www.apple.com`”)

hash(“`www.microsoft.com`”)

hash(“`www.nytimes.com`”)

0	**0**
1	**0**
2	**1**
3	**1**
4	**0**
5	**0**
6	**1**
7	**0**
8	**1**
9	**0**

Fingerprints

Set Abstract Data Type

– Maintain a vector of 0/1 bits.

```
insert(key)
1. h = hash(key);
2. table[h] = 1;

lookup(key)
1. h = hash(key);
2. return (table[h] == 1);
```

The key difference of a Fingerprint Hash Table (FHT) is:

1. A FHT prevents collisions.
2. A FHT does not store the key in the table.
3. A FHT works with simpler hash functions.
4. A FHT saves time calculating hashes.
5. I don’t understand how an FHT is different.

Implementing a Set

Use a fingerprint:

hash(“`www.gmail.com`”)

hash(“`www.apple.com`”)

hash(“`www.microsoft.com`”)

hash(“`www.nytimes.com`”)

0	**0**
1	**0**
2	**1**
3	**1**
4	**0**
5	**0**
6	**1**
7	**0**
8	**1**
9	**0**

Implementing a Set

What happens on collision?

hash(“`www.gmail.com`”)

hash(“`www.apple.com`”)

hash(“`www.microsoft.com`”)

hash(“`www.nytimes.com`”)

0	**0**
1	**0**
2	**0**
3	**1**
4	**0**
5	**0**
6	**1**
7	**0**
8	**1**
9	**0**

Implementing a Set

Lookup operation:

hash(“`www.microsoft.com`”)

0	**0**
1	**0**
2	**0**
3	**1**
4	**0**
5	**0**
6	**1**
7	**0**
8	**1**
9	**0**

If the URL is in the web cache, it will always report **true**.

(No false negatives.)

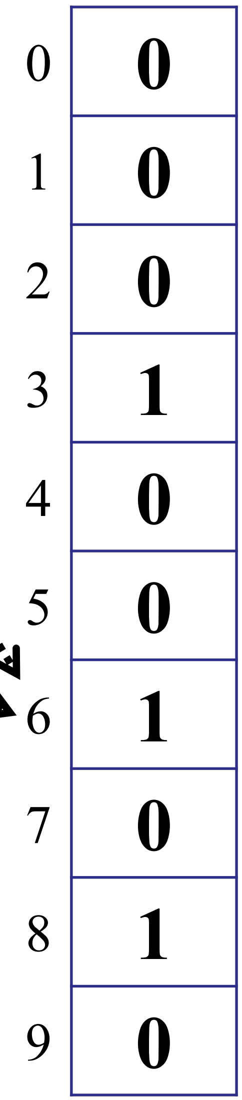
Fingerprint Hash Table

Insert operation:

hash("`www.microsoft.com`")

Lookup operation:

hash("`www.rugby.com`")



Even if the URL is NOT in the set,

it may *sometimes* report **true**.

(False positives.)

Facebook example: if the FHT stores the set of online users, then you might:

1. Believe Fred is on-line, when he is not.
2. Believe Fred is offline, when is not.
3. Never make any mistakes.

Spam example: it is better to store in the Fingerprint Hash Table:

1. The set of **good** e-mail addresses.
2. The set of **bad** e-mail addresses
3. It does not matter.

I think it is better to mistakenly
accept a few SPAM e-mails
than to accidently reject an e-mail
from my mother!

Fingerprint Analysis

Probability of a false negative: 0

Fingerprint Analysis

Probability of a false positive?

On lookup in a table of size m with n elements,
Probability of **no** false positive:

$$\left(1 - \frac{1}{m}\right)^n \approx \left(\frac{1}{e}\right)^{n/m}$$

chance of no collision

Fingerprint Analysis

Probability of collision?

hash(“`www.gmail.com`”)

0	**0**
1	**0**
2	**0**
3	**1**
4	**0**
5	**0**
6	**1**
7	**0**
8	**1**
9	**0**

What is the probability that no other URL is in slot 3?

Fingerprint Analysis

Probability of **no** false positive: (simple uniform hashing assumption)

$$\left(1 - \frac{1}{m}\right)^n \approx \left(\frac{1}{e}\right)^{n/m}$$

Probability of a false positive, at most:

$$1 - \left(\frac{1}{e}\right)^{n/m}$$

Fingerprint Analysis

Assume you want:

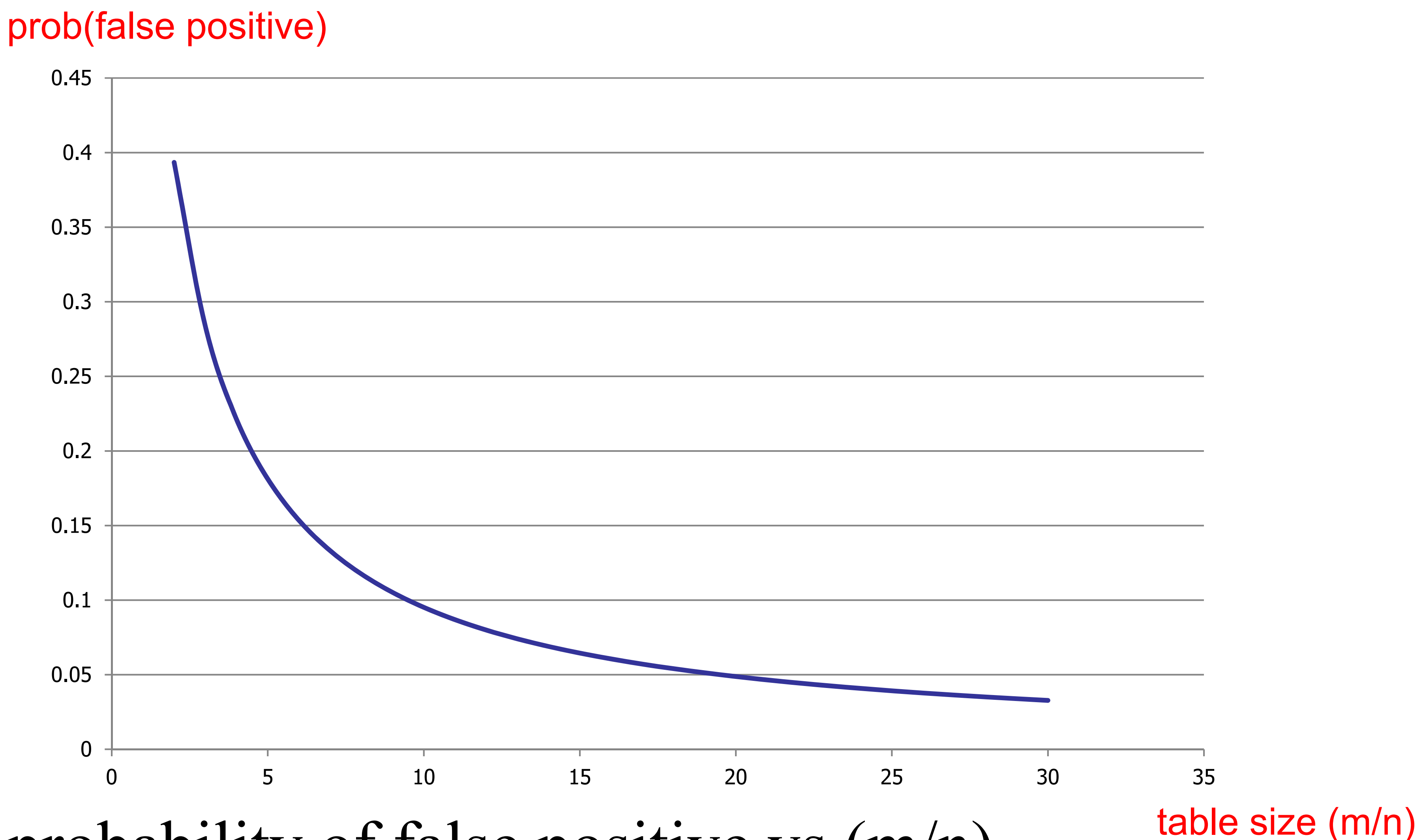
- Probability of false positives $< p$
  - Example: at most 5% of queries return false positive.

    $p = .05$

- Need: $\frac{n}{m} \leq \log\left(\frac{1}{1-p}\right)$

  - Example: $m >= (13.5)n$

Fingerprint Analysis



probability of false positive vs (m/n)

Summary So Far

Fingerprint Hash Functions

- Don't store the key.
- Only store 0/1 vector.

Summary So Far

Fingerprint Hash Functions

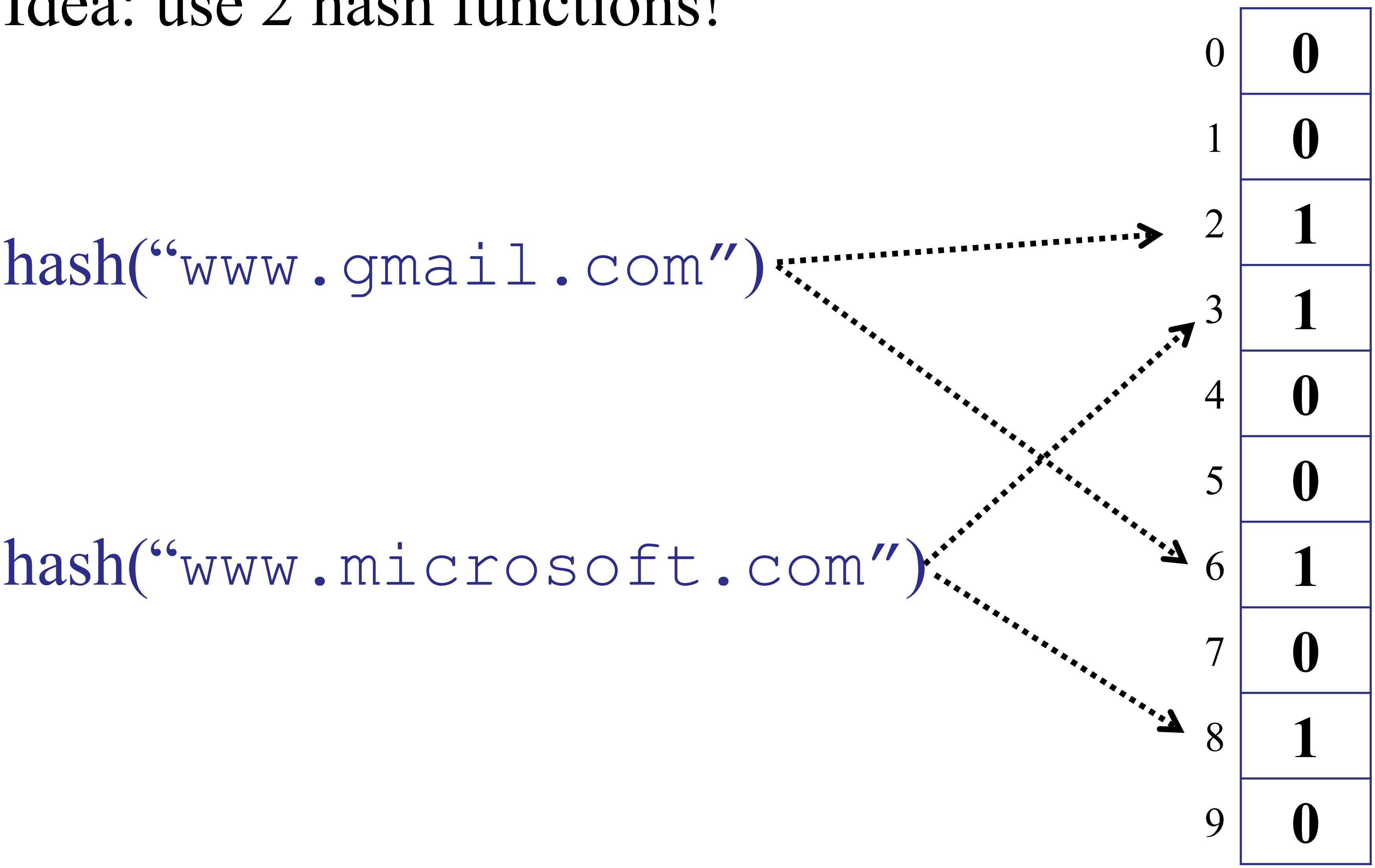
- Don’t store the key.
- Only store 0/1 vector.
- Trade-off:
  - Reduced space: only 1-bit per slot
  - Increase space: bigger table to avoid collisions

Fingerprint Hash Table

Can we do better?

Bloom Filter

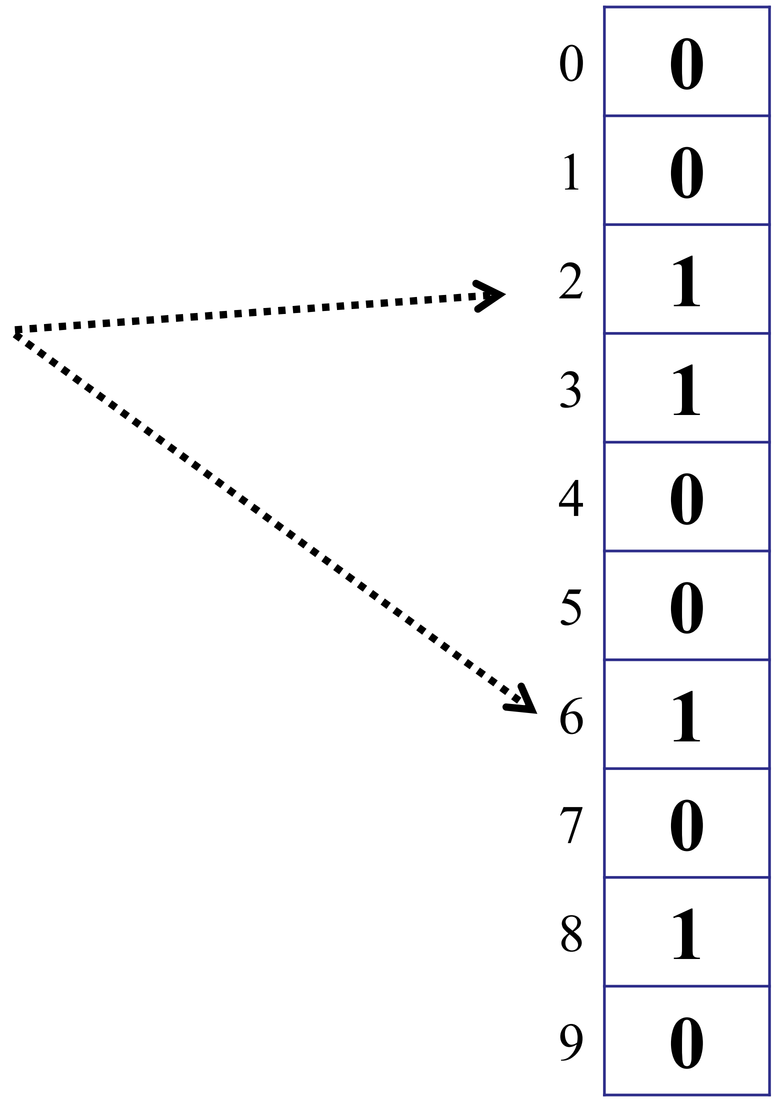
Idea: use 2 hash functions!



Bloom Filter

Idea: use 2 hash functions!

hash(“`www.gmail.com`”)



insert(URL)

k_1 = hash$_1$(URL);

k_2 = hash$_2$(URL);

T[k_1] = 1;

T[k_2] = 1;

Bloom Filter

Idea: use 2 hash functions!

query(URL)

  k_1 = hash$_1$(URL);

  k_2 = hash$_2$(URL);

  if (T[k_1] && T[k_2])

    return true;

  else return false;

0	**0**
1	**0**
2	**1**
3	**1**
4	**0**
5	**0**
6	**1**
7	**0**
8	**1**
9	**0**

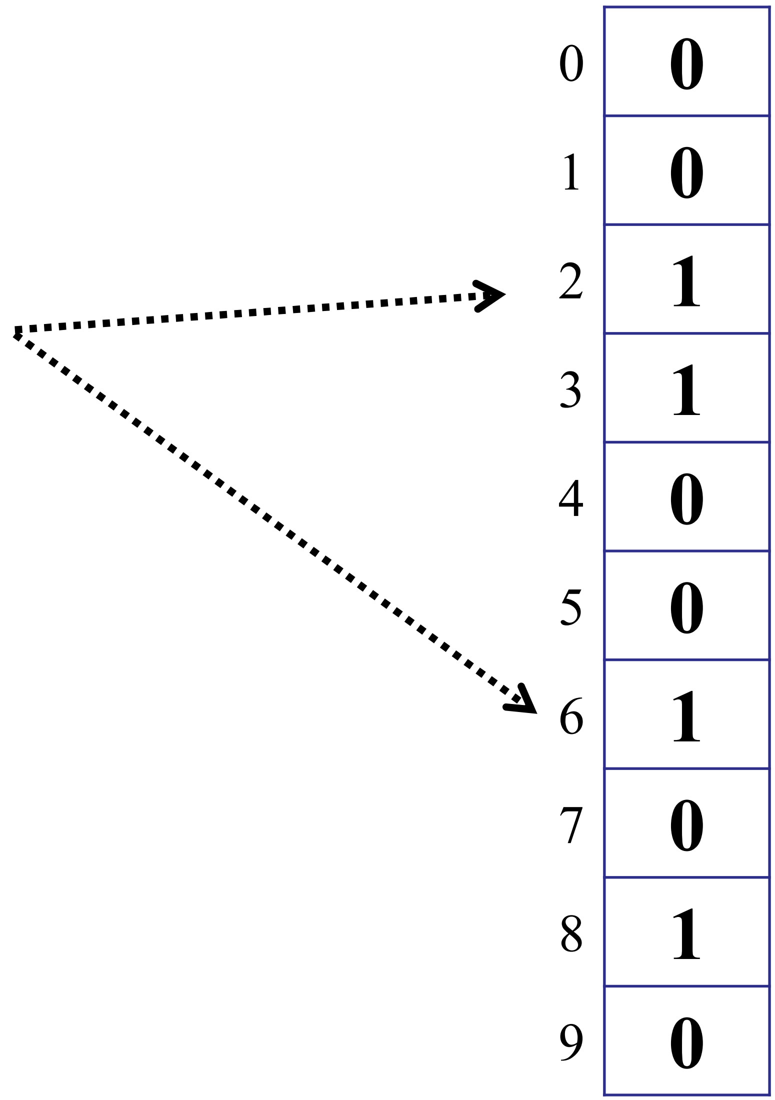
A Bloom Filter can have:

✓ 1. Only false positives.
2. Only false negatives.
3. Both false positives and negatives.
4. Wait, which is which again?

Bloom Filter

Idea: use 2 hash functions!

hash(“`www.gmail.com`”)



- No false negatives.
- Possible false positives.

Bloom Filter

Idea: use 2 hash functions!

query(URL)

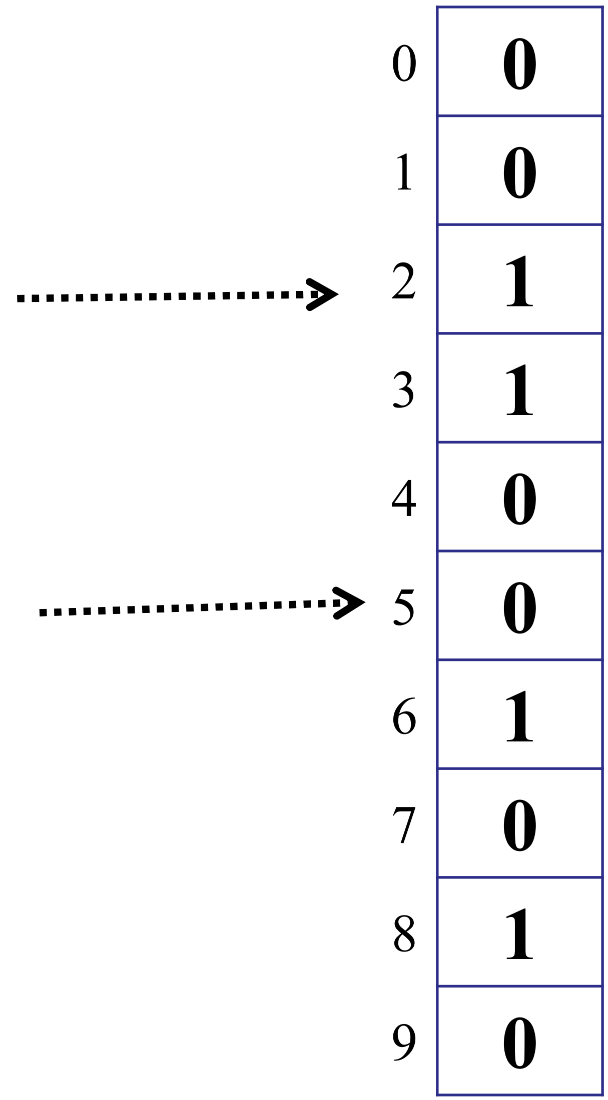
k_1 = hash$_1$(URL);

k_2 = hash$_2$(URL);

if (T[k_1] && T[k_2])

return true;

else return false;



Bloom Filter

Idea: use 2 hash functions!

Trade-off:

- Each item takes more "space" in the table.
- Requires two collisions for a false positive.

0	**0**
1	**0**
2	**1**
3	**1**
4	**0**
5	**0**
6	**1**
7	**0**
8	**1**
9	**0**

Bloom Filter Analysis

Probability a given bit is 0:

$$\left(1 - \frac{1}{m}\right)^{2n} \approx \left(\frac{1}{e}\right)^{2n/m}$$

chance hash does not choose this bit

each of n items sets 2 bits in the table

Bloom Filter Analysis

Probability a given bit is 0:

$$\left(1 - \frac{1}{m}\right)^{2n} \approx \left(\frac{1}{e}\right)^{2n/m}$$

Probability of a false positive: (1 set in both slots)

$$\left(1 - \left(\frac{1}{e}\right)^{2n/m}\right)^{2}$$

Question:

1. What analytic mistake did I make on the previous slide?
2. The slots are not independent!
3. If one slot is a 1, then the other slot is less likely to be a 1.

Bloom Filter Analysis

Probability a given bit is 0:

$$\left(1 - \frac{1}{m}\right)^{2n} \approx \left(\frac{1}{e}\right)^{2n/m}$$

Probability of a false positive: (1 set in both slots)

$$\left(1 - \left(\frac{1}{e}\right)^{2n/m}\right)^{2}$$

* Assuming BOGUS fact that each table slot is independent…

Bloom Filter Analysis

Assume you want:

- probability of false positives < p
  - Example: at most 5% of queries return false positive.

    $p = .05$

- Need: $$\frac{n}{m} \le \frac{1}{2}\log\left(\frac{1}{1-p^{1/2}}\right)$$
  - Example: $m >= (7.9)n$

* Assuming BOGUS fact that each table slot is independent…

Bloom Filter

prob(false positive)

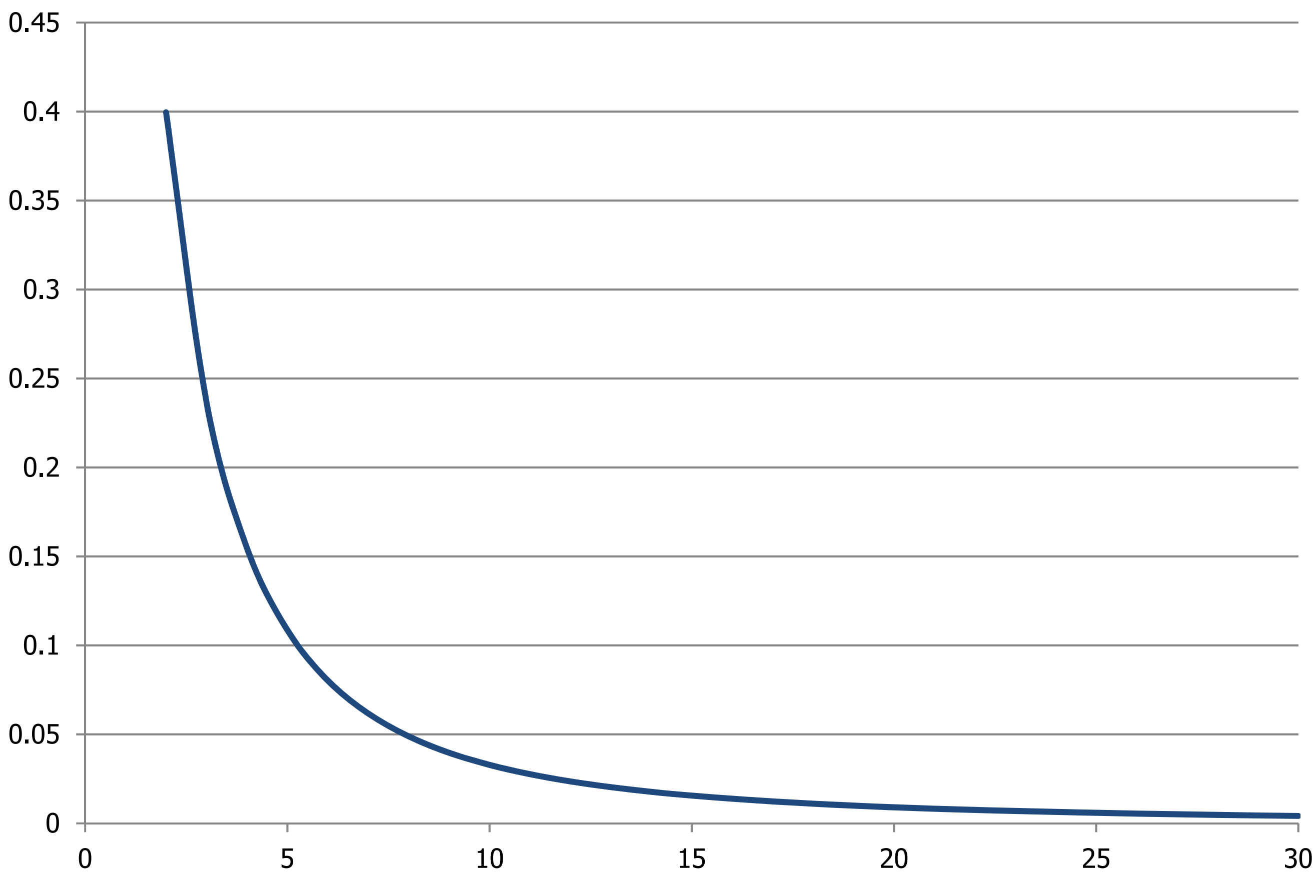


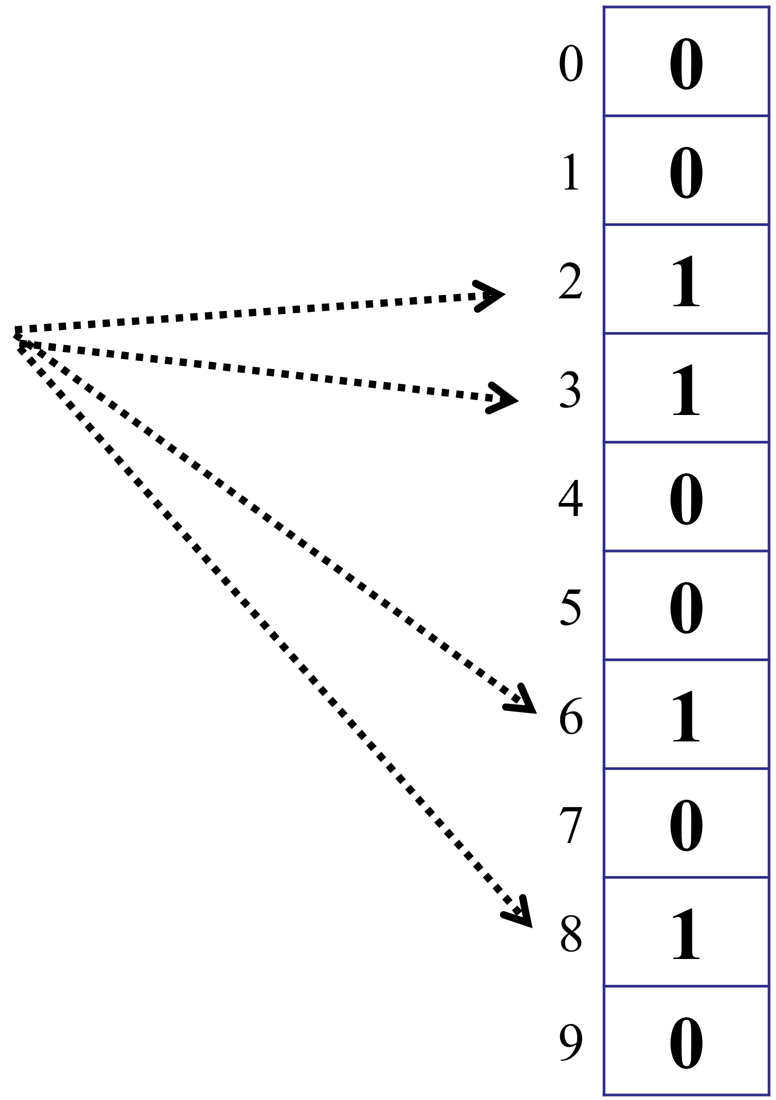
table size (m/n)

False positives rate vs. (m/n)

Bloom Filters

Use *k* hash functions!

hash(“`www.gmail.com`”)



Bloom Filter Analysis

Probability a given bit is 0:

$$\left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}$$

Bloom Filter Analysis

Probability a given bit is 0:

$$\left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}$$

Probability of a collision at one spot:

$$1 - e^{-kn/m}$$

* Assuming BOGUS fact that each table slot is independent…

Bloom Filter Analysis

Probability of a collision at one spot:

$$1 - e^{-kn/m}$$

Probability of a collision at all *k* spots:

$$\left(1 - e^{-kn/m}\right)^k$$

* Assuming BOGUS fact that each table slot is independent…

Bloom Filter

prob(false positive)

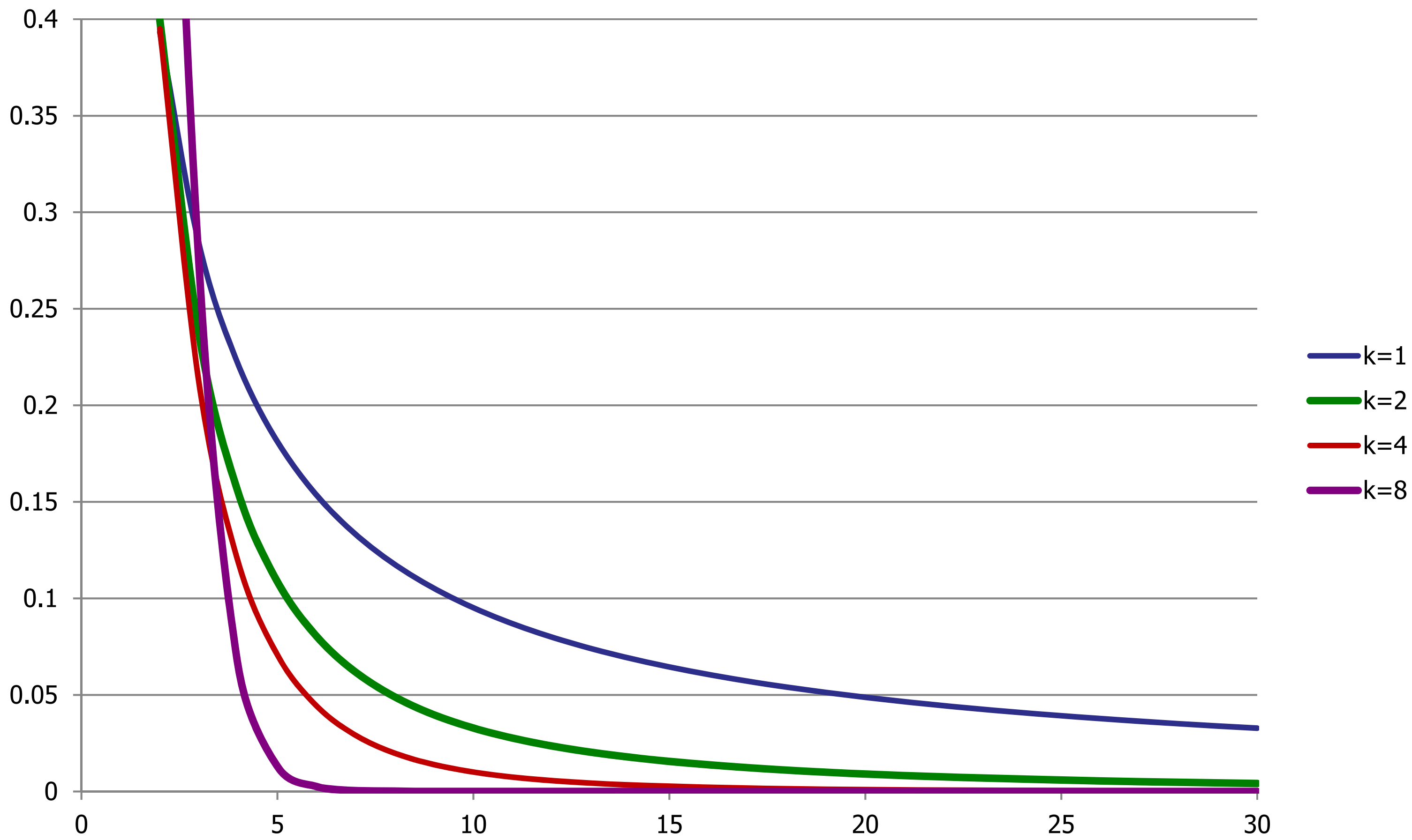
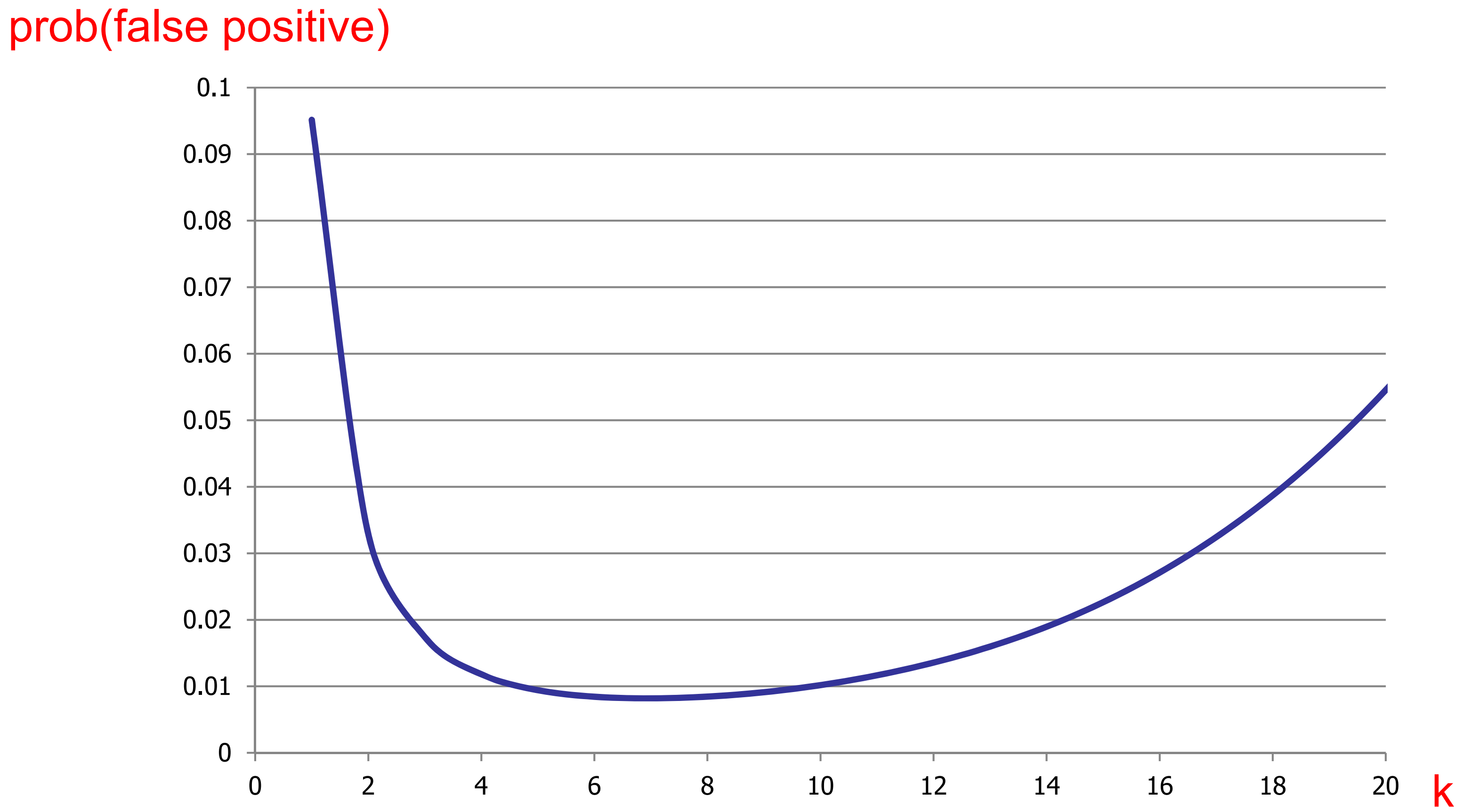


table size (m/n)

false positive rate vs. (m/n)

Bloom Filter

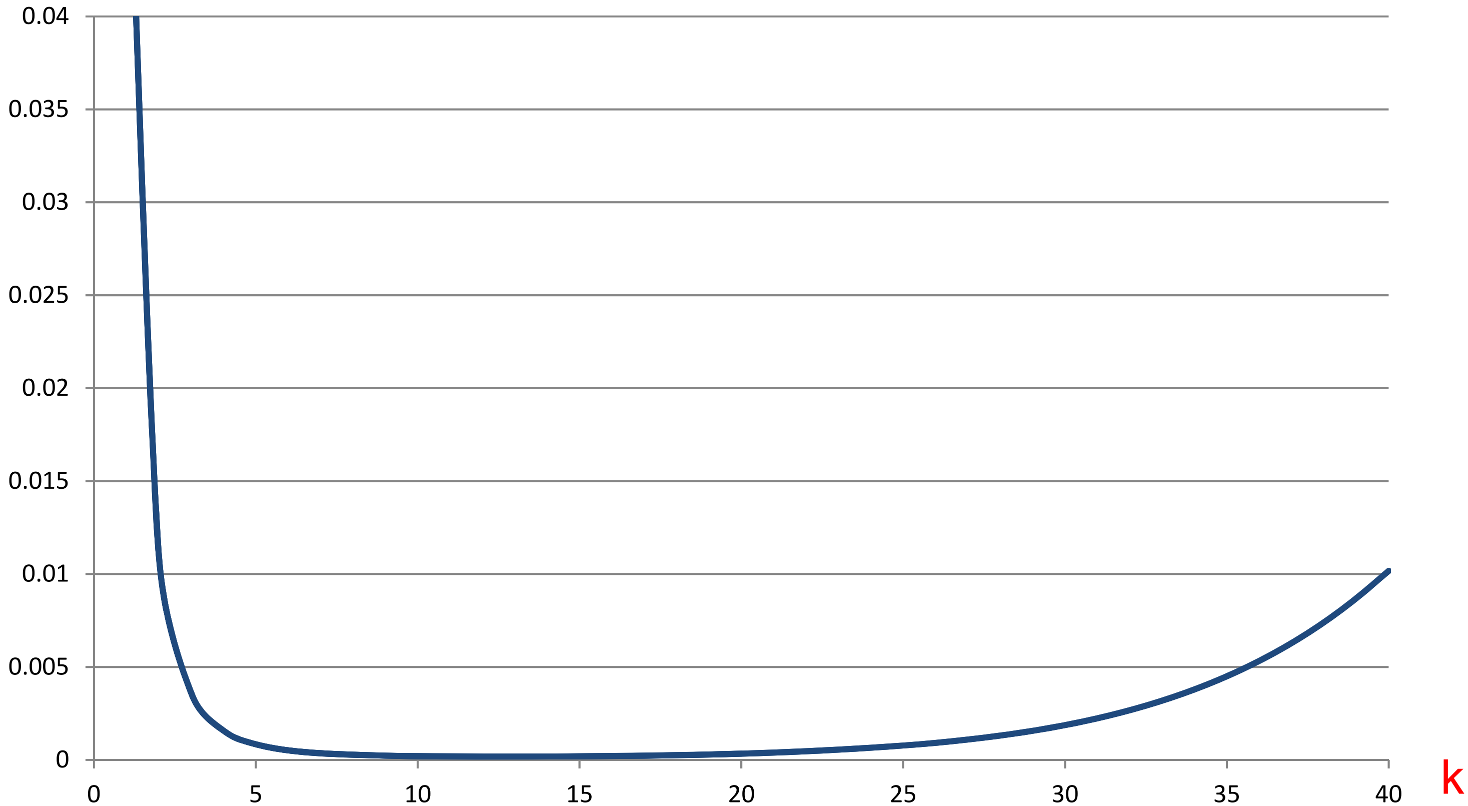


false positive rate vs *k*

m = 10n

Bloom Filter

prob(false positive)



false positive rate vs *k*

m = 18n

Bloom Filter

What is the optimal value of k?

- Probability of false positive:

$$\left(1 - e^{-kn/m}\right)^k$$

- Choose: $k = \frac{m}{n} \ln 2$

- Error probability: 2^{-k}

Implementing Sets

- Fingerprint Hash Functions
  - Don’t store the key.
  - Only store 0/1 vector.
- Bloom Filter
  - Use more than one hash function.
  - Redundancy reduces collisions.
- Probability of Error
  - False positives
  - False negatives

Fingerprint Hash Table

What about deletion?

insert(“`www.gmail.com`”)

insert(“`www.apple.com`”)

delete(“`www.gmail.com`”)

0	**0**
1	**0**
2	**0**
3	**1**
4	**0**
5	**0**
6	**1**
7	**0**
8	**1**
9	**0**

Bloom Filters

What about deleting an element?

- Store counter instead of 1 bit.
- On insert: increment.
- On delete: decrement.

Beware:

- If counter is big, then no space saving.
- If collisions are rare, counter is small: only a few bits.

Bloom Filters

Implementation of Set ADT:

- insert: O(k)
- delete: O(k)
- query: O(k)

Bloom Filters

Implementation of Set ADT:

- intersection
  - Bitwise AND of two Bloom filters
  - Time: O(m)

0	**0**	**&**	**0**
1	**0**	**&**	**1**
2	**0**	**&**	**0**
3	**1**	**&**	**1**
4	**0**	**&**	**0**
5	**0**	**&**	**0**
6	**1**	**&**	**0**
7	**0**	**&**	**0**
8	**1**	**&**	**1**
9	**0**	**&**	**0**

Bloom Filters

Implementation of Set ADT:

- intersection
  - Bitwise AND of two Bloom filters: O(m)
- union
  - Bitwise OR of two Bloom filters: O(m)

Other applications

- Chrome browser safe-browsing
  - Maintains list of “bad” websites.
  - Occasionally retrieves updates from google server.
- Spell-checkers
  - Storing all words takes a lot of space.
  - Instead, store a Bloom filter of the words.
- Weak password dictionaries

Summary

When to use Bloom Filters?

- Storing a set of data.
- Space is important.
- False positives are ok.

Interesting trade-offs:

- Space
- Time
- Error probability

Today: Hash Tables (continued)

- Table (re)sizing
  - Proper hash table size
  - Amortized analysis
- Sets
  - Hash table sets
  - Bloom Filters