CS2040S
Data Structures and Algorithms

Hashing! (Part 3)

× 1, 4, 2, 6
6

2, 4, 6
2, 6
× 1, 5, 6

Puzzle of the Week:

You throw a dice repeatedly until you get a 6.

Conditioned on the event that all throws gave even numbers, what is the expected number of throws (including the throw giving 6)?

Plan: this week and next

Third day of hashing

- Brief review
- Table resizing
- Sets (and Bloom Filters, time permitting)

Plan: this week and next

- Third day of hashing
  - Brief review
  - Table resizing
  - Sets (and Bloom Filters, time permitting)

- Looking ahead: next week
  - Open addressing
  - Start unit on graph algorithms

Plan: this week and next

Third day of hashing

- **Brief review**
- Table resizing
- Sets (and Bloom Filters, time permitting)

Quick Review

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.

Quick Review

Hash Table

- Implements a symbol table.
- Goal:
  - O(1) insert
  - O(1) lookup
- Idea:
  - Store data in a large array.
  - Hash function maps key to slot in the array.
  - Challenge: choosing a good hash function.

Quick Review

java.util.Map

public interface	java.util.Map<Key, Value>	
void	clear()	*removes all entries*
boolean	containsKey(Object k)	*is k in the map?*
boolean	containsValue(Object v)	*is v in the map?*
Value	get(Object k)	*get value for k*
Value	put(Key k, Value v)	*adds (k,v) to table*
Value	remove(Object k)	*remove mapping for k*
int	size()	*number of entries*

Note: no successor / predecessor queries.

Quick Review

Example: HashMap

```
Map<String, Integer> ageMap = new HashMap<String, Integer>();

ageMap.put("Alice", 32);
ageMap.put("Bernice", 84);
ageMap.put("Charlie", 7);

Integer age = ageMap.get("Alice");
System.out.println("Alice's age is: " + age + ".");
```

- Key-type: String
- Value-type: Integer

Quick Review

- Every Object x needs to support:
  - `equals(Object obj)`: checks if x is “equal” to obj, where equality defined in a context-specific way

`Quick` Review

- Every Object `x` needs to support:
  - `equals(Object obj)`: checks if `x` is "equal" to `obj`, where equality defined in a context-specific way
  - `hashCode()`: conversion of `x` to an integer
    - Invoking on same object twice should return same value
    - If `x.equals(obj)`, then you **must** have `hashCode(x)==hashCode(obj)`.
    - If `hashCode(x)==hashCode(obj)`, it's **recommended** that `x.equals(obj)`.

Quick Review

From an older Java implementation of `HashMap` but simpler than what we saw last lecture:

```
final int hash(Object key) {
      return key == null ?
             0 :
             Math.abs(key.hashCode() % buckets.length);
}
```

```
public V get(Object key) {
      int idx = hash(key);
      HashEntry<K, V> e = buckets[idx];
      while (e != null) {
            if (equals(key, e.key))
                  return e.value;
            e = e.next;
      }
      return null;
}
```

Quick Review

Hash Table with Chaining

- Each array slot stores a linked list.
- All items mapped to the same slot are stored in the linked list.

Plan: this week and next

Third day of hashing

- Brief review
- **Table resizing**
- Sets (and Bloom Filters, time permitting)

Table Size

How large should the table be?

– Assume: Hashing with Chaining

– Assume: Simple Uniform Hashing

– Expected search time: $O(1 + \underbrace{n/m}_{\text{load factor}})$

– 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 6 items, table too big! Shrink…

Table Size

How to grow the table:

1. Choose new table size *m*.
2. Choose new hash function h.
   - 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: $O(n + m_1)$
  - Inserting each element in new hash table: $O(1)$ / element
  - Total: $O(m_1 + n)$

Table Size

Time complexity of growing the table:

- Assume:
  - Size $m_1 < 2n$.
  - Size $m_2 < \ldots$??
- Costs:
  - Total: $O(m_1 + n)$.

    $= O(n)$

Table Size

Time complexity of growing the table:

Wait! What is the cost of initializing the new table?

– Initializing a table of size x takes x time!

– Costs:

Total: $O(m_1 + m_2 + n)$

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): $m = m+1$

  Full table

- Cost of resize:
  - Size $m_1 = n$.
  - Size $m_2 = n+1$.

In this case, what is the cost of resizing the table from m to m+1?

$m_1 = n$ $\quad$ $m_2 = n+1$

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

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: $O(n)$

Initially: $m = 8$
What is the cost of inserting n items?

1. O(n)
2. O(n log n)
3. O(n^2)
4. O(n^3)
5. None of the above.

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 $\xrightarrow{+1}$	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: $(7 + 8 + 9 + 10 + 11 + \ldots + n) = \underline{O(n^2)}$

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

x2

cheap

O(n)

- Total resizing cost: $(8 + 16 + 32 + \ldots + n) = O(n)$

How fast to grow

Idea 2: Double table size

Cost of Resizing:

Table size	Total Resizing Cost
8	8
16	(8 + 16)
32	(8 + 16 + 32)
64	(8 + 16 + 32 + 64)
128	(8 + 16 + 32 + 64 + 128)
…	…
m	$<(\underline{1+2+4}+8+\ldots+m) \leq 2m$

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)

  O(n)

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

How fast to grow?

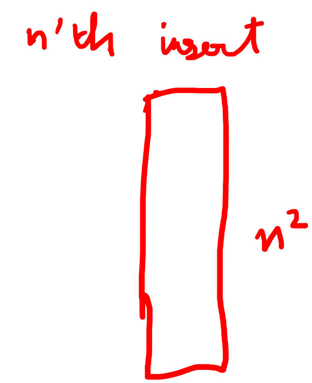
Idea 3: Square table size

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

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

Assume: square table size
What is the cost of inserting *n* items?

1. O(log n)
2. O($\sqrt{n}$)
3. O(n)
4. O(n log n)
5. O(n^2)
6. O(2^n)
7. None of the above.



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

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

# Items	Total Resizing Cost
8	64
64	(64 + 4,096)
4,096	(64 + 4,096 + …)
…	…
n	$(\ldots + \ldots + \ldots + n^2)$
	$= O(n^2)$

How fast to grow?

Idea 3: Square table size

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

# Items	Resizing Cost	Insert Cost
8	64	8
64	(64 + 4,096)	64
4,096	(64 + 4,096 + …)	4,096
…	…	…
n	$(\ldots + \ldots + \ldots + n^2)$	n
	$= O(n^2)$	$O(n)$

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: $O(n)$

Why else is squaring the table size bad?

1. Resize takes too long to find items to copy.
2. Inefficient space usage.
3. Searching is more expensive in a big table.
4. Inserting is more expensive in big table.
5. Deleting is more expensive in a big table.

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

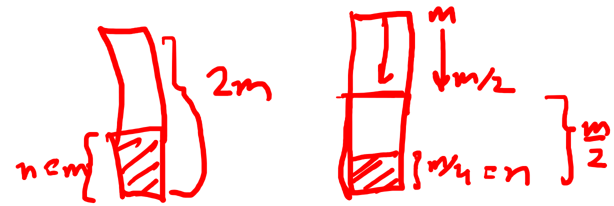
Example execution:

- Start: n=100, m=200 $O(n)$
- cost=100 Delete: n=99, m=200 → shrink to m=100 $O(n)$
- cost=100 Insert: n=100, m=100 → grow to m=200 $O(n)$
- cost=100 Delete: n=99, m=200 → shrink to m=100 $O(n)$
- cost=100 Insert: n=100, m=100 → grow to m=200 $O(n)$
- cost=100 Delete: n=99, m=200 → shrink to m=100
- cost=100 Insert: n=100, m=100 → grow to m=200
- Repeat… $O(n^2)$

Deleting Elements

Rules for shrinking and growing:

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



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

    not $m/2$

- Claim:
  - After every change: the table is half full, half empty.
  - Every time you double a table of size m, at least $m/2$ new items were added since last change.
  - Every time you shrink a table of size m, at least $m/4$ items were deleted since last change.

Deleting Elements

Example execution:

- Start: n=100, m=200

cost=350 • Delete 50: n=50, m=200 → shrink to m=100

cost=350 • Insert 50: n=100, m=100 → grow to m=200

cost=20 • Delete 20: n=80, m=200 → unchanged

cost=720 • Insert 120: n=200, m=200 → grow to m=400

cost=100 • Insert 100: n=300, m=400 → unchanged

Summary

Basic idea:

- When table is full, double the size.
- When table is ¾ empty, half the size.
- Most operations are O(1). Amortized analysis.
- Some operations cost O(n).
- On average, operations cost O(1).

n operations (insert/delete) ⇒ O(n) time

Question: Can we ensure that all operations are O(1)?

- Do a little bit of table resizing with every operation?

Plan: this week and next

Third day of hashing

- Brief review
- Table resizing
- **Sets (and Bloom Filters, time permitting)**

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?

A few examples

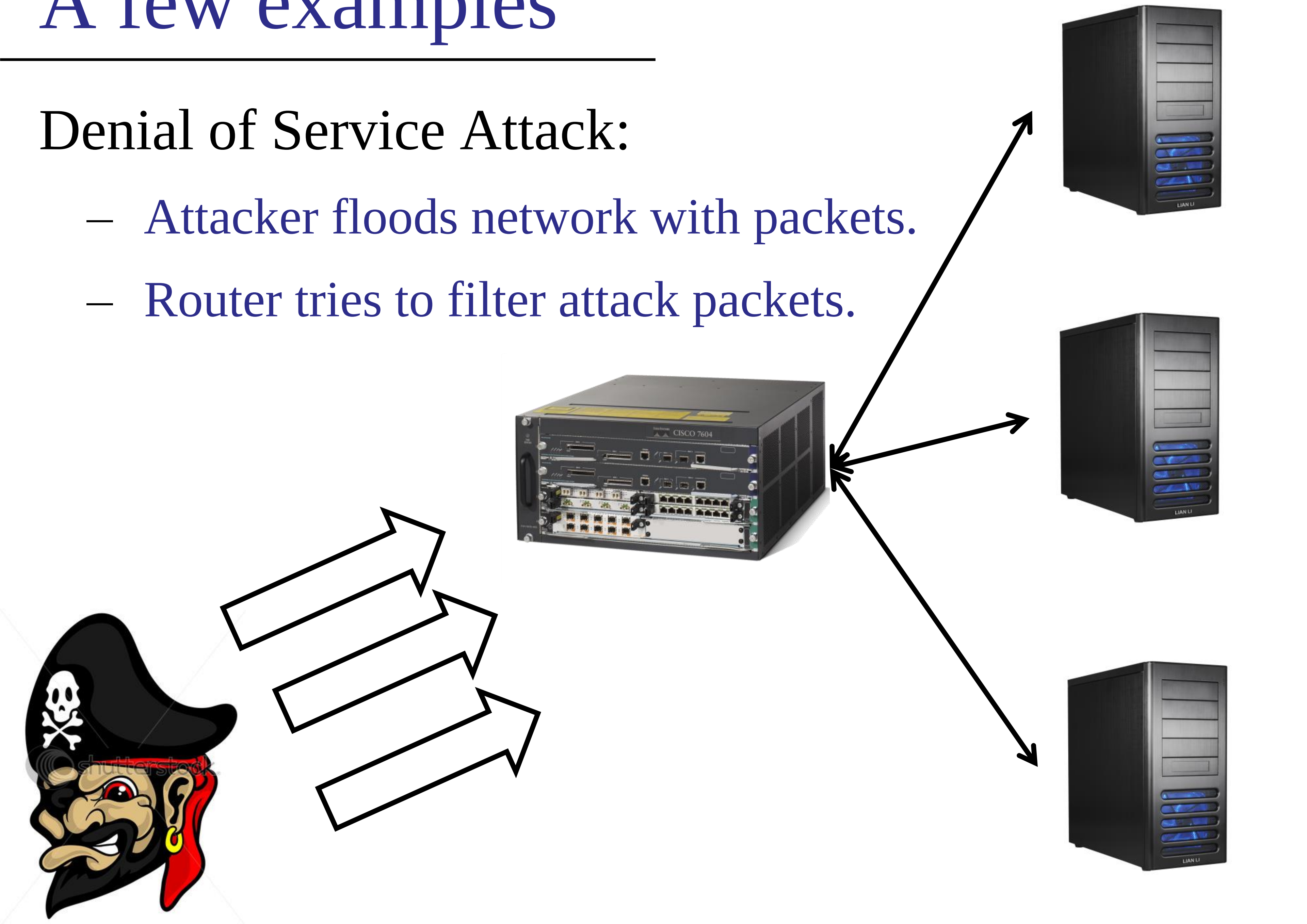
Spam filter:

- 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?

A few examples

Denial of Service Attack:

- Attacker floods network with packets.
- Router tries to filter attack packets.



A few examples

Denial of Service Attack:

- Attacker floods network with packets.
- Router tries to filter attack packets.

1. Keep list of bad IP addresses. (Same as spam solution.)

2. Only allow 100 packets/second from each IP address.

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<…>

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**

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:

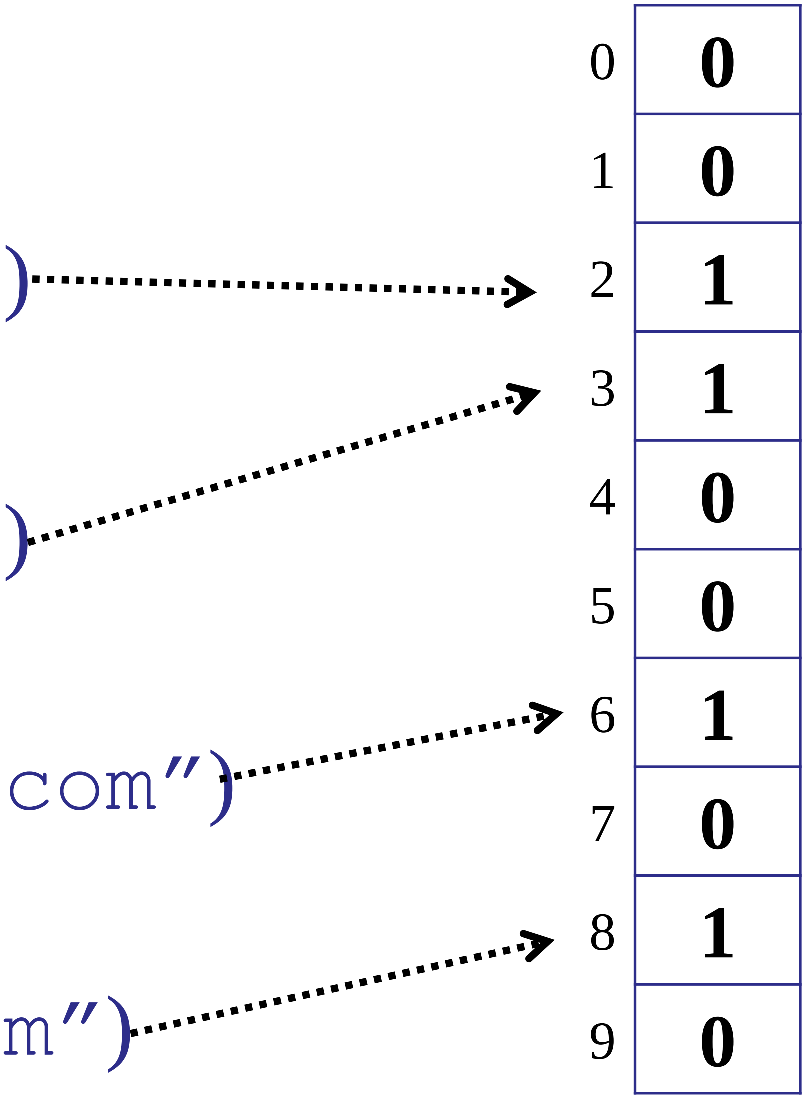
- Only store/send *m* bits!

hash(“`www.gmail.com`”)

hash(“`www.apple.com`”)

hash(“`www.microsoft.com`”)

hash(“`www.nytimes.com`”)



Fingerprints

Set Abstract Data Type

- Maintain a vector of 0/1 bits.

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

lookup(key)
1. h = hash(key);
2. return (m_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:

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`”)

hashcode = 2726

Lookup operation:

hash(“`www.rugby.com`”)

hashcode = 5426

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

m = 10

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 negative: 0

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 a false negative: 0

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:

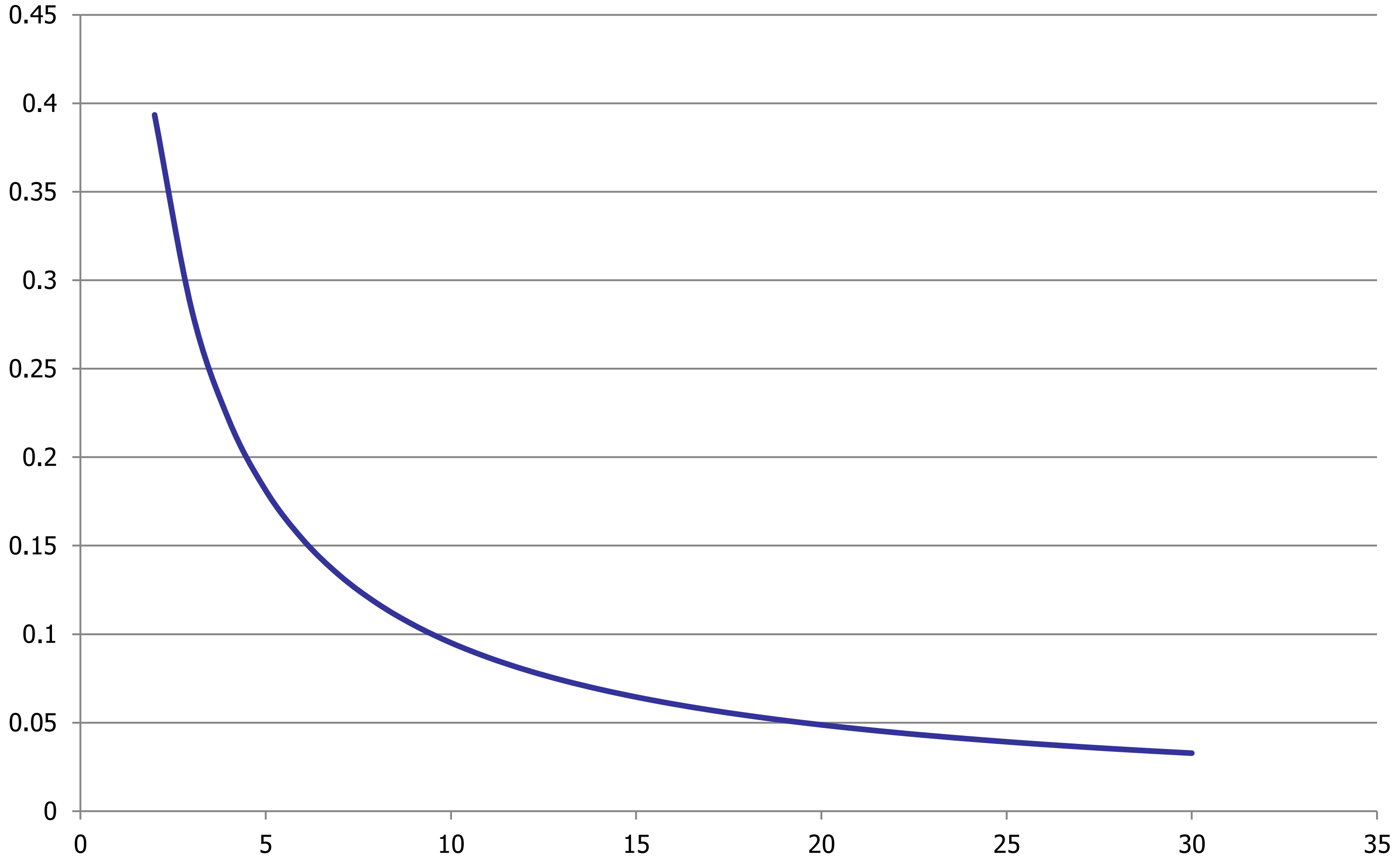
- probability of false positives $< p$
  - Example: at most 1% of queries return false positive.

    $p = .01$

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

  - Example: $m >= (68.97)n$

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

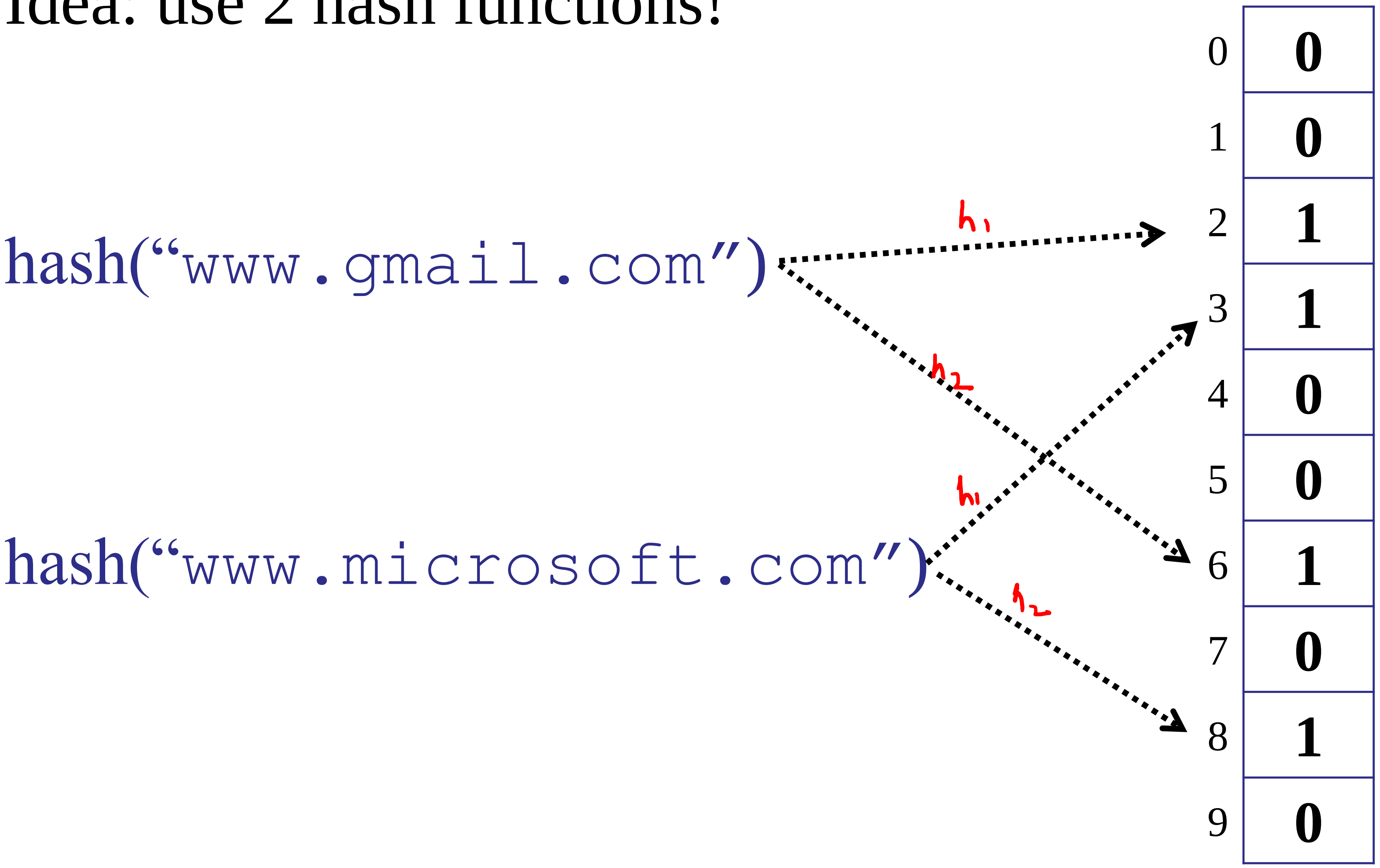
– Don’t store the key.
– Only store 0/1 vector.
– Trade-off:
  - Reduced space: only 1-bit per slot
  - Increase space: bigger table to avoid collisions

Fingerprint Hash Table

Can we do better?

Bloom Filter

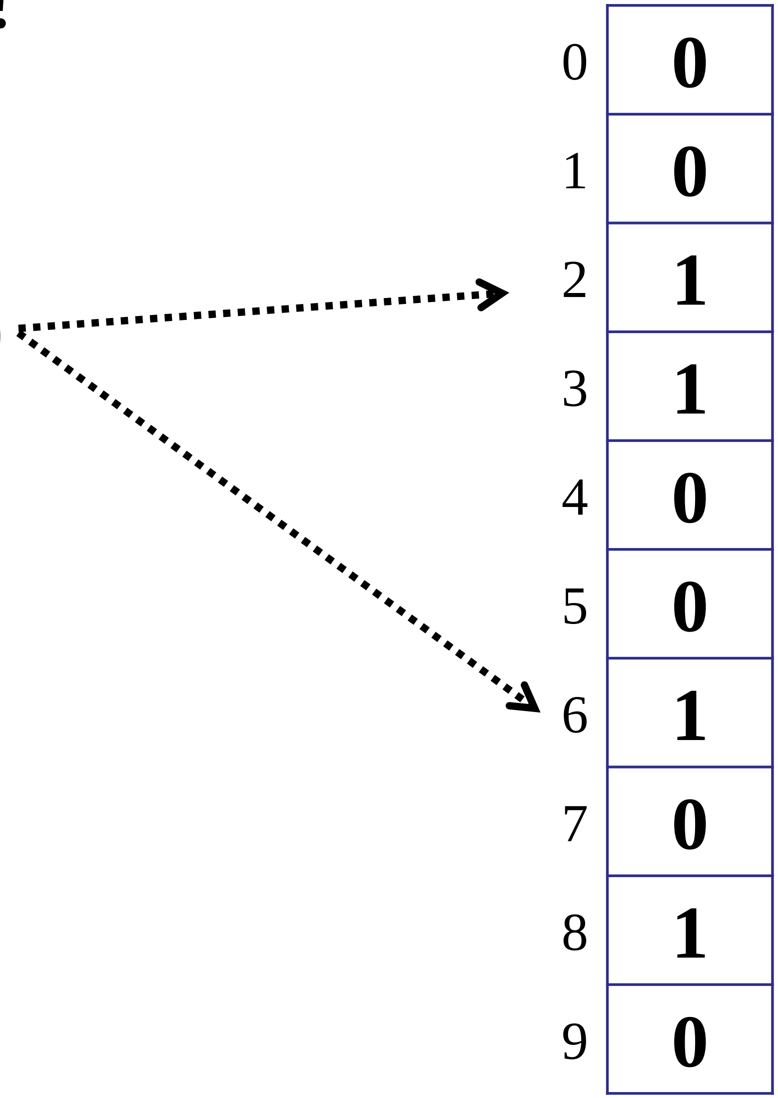
Idea: use 2 hash functions!



Bloom Filter

Idea 2: use 2 hash functions!

hash(“`www.gmail.com`”)



insert(URL)

k_1 = hash$_1$(URL);

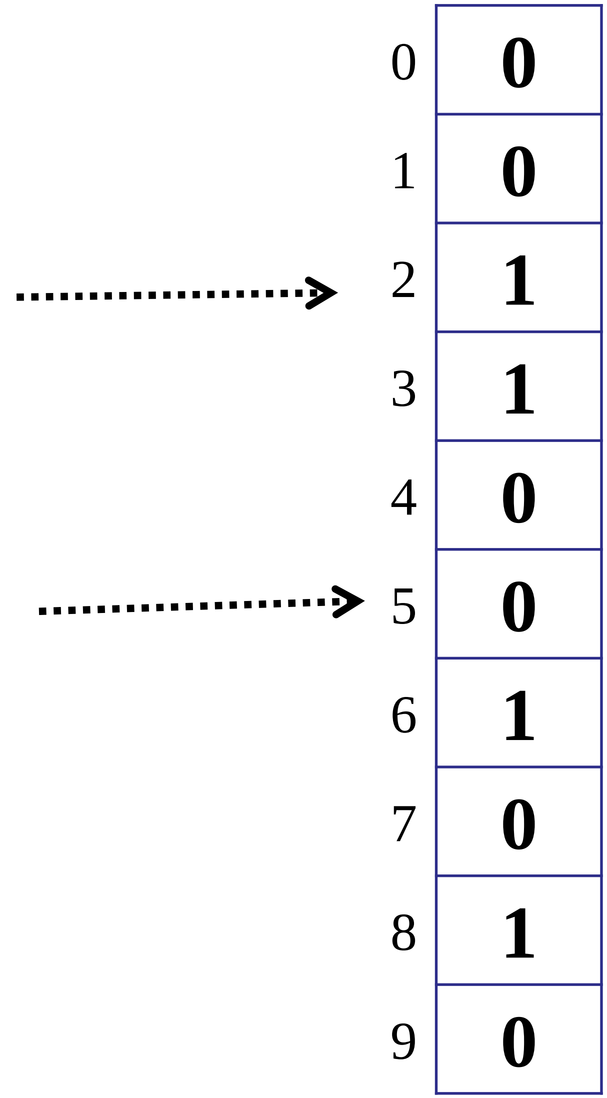
k_2 = hash$_2$(URL);

T[k_1] = 1;

T[k_2] = 1;

Bloom Filter

Idea 2: 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;

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!

query(URL)

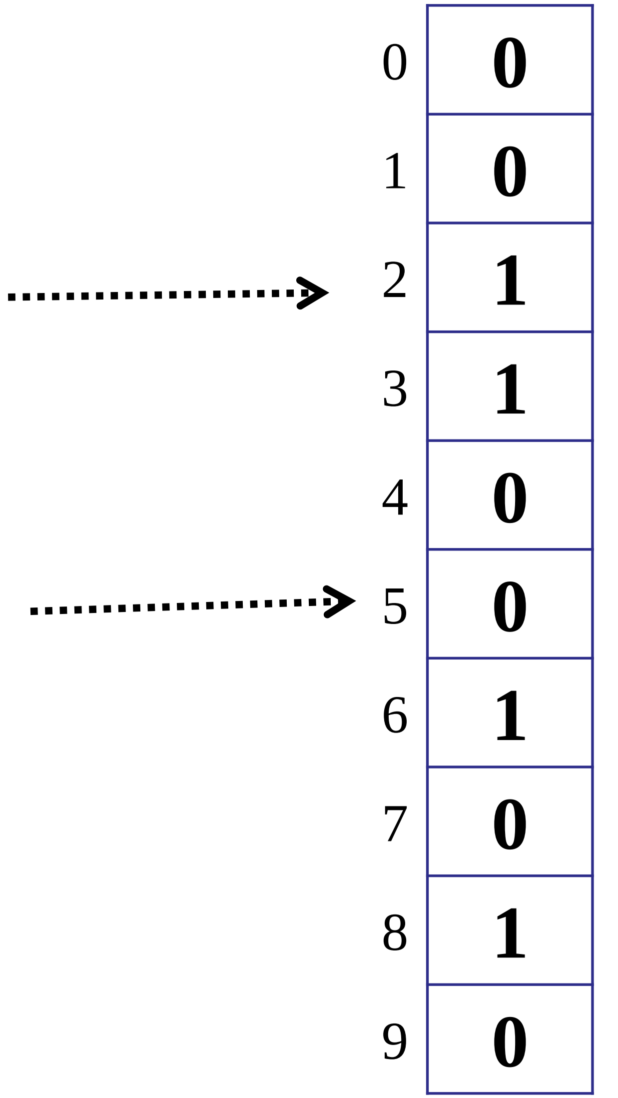
k_1 = hash$_1$(URL);

k_2 = hash$_2$(URL);

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

return true;

else return false;



Bloom Filter

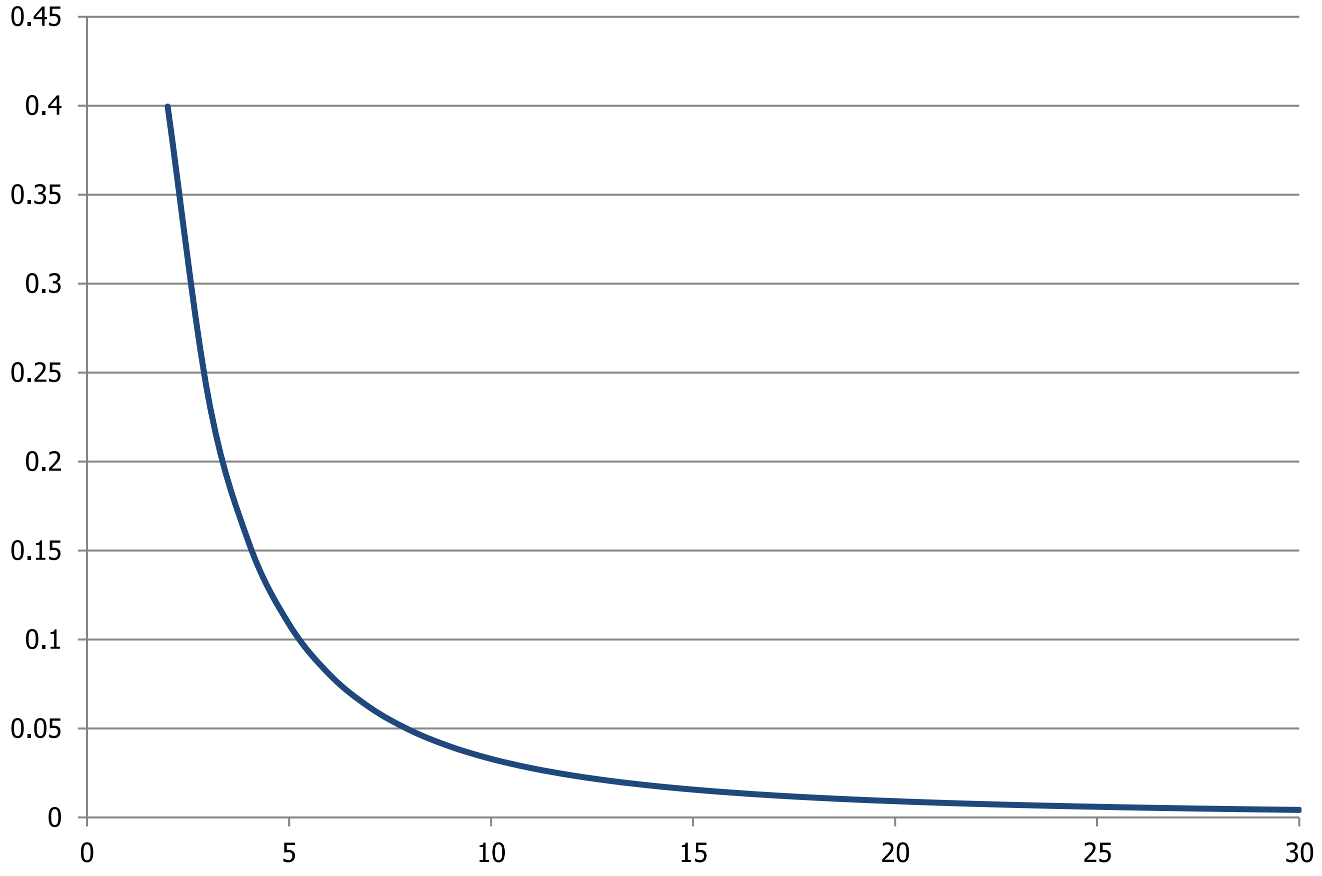
Idea 2: use 2 hash functions!

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**

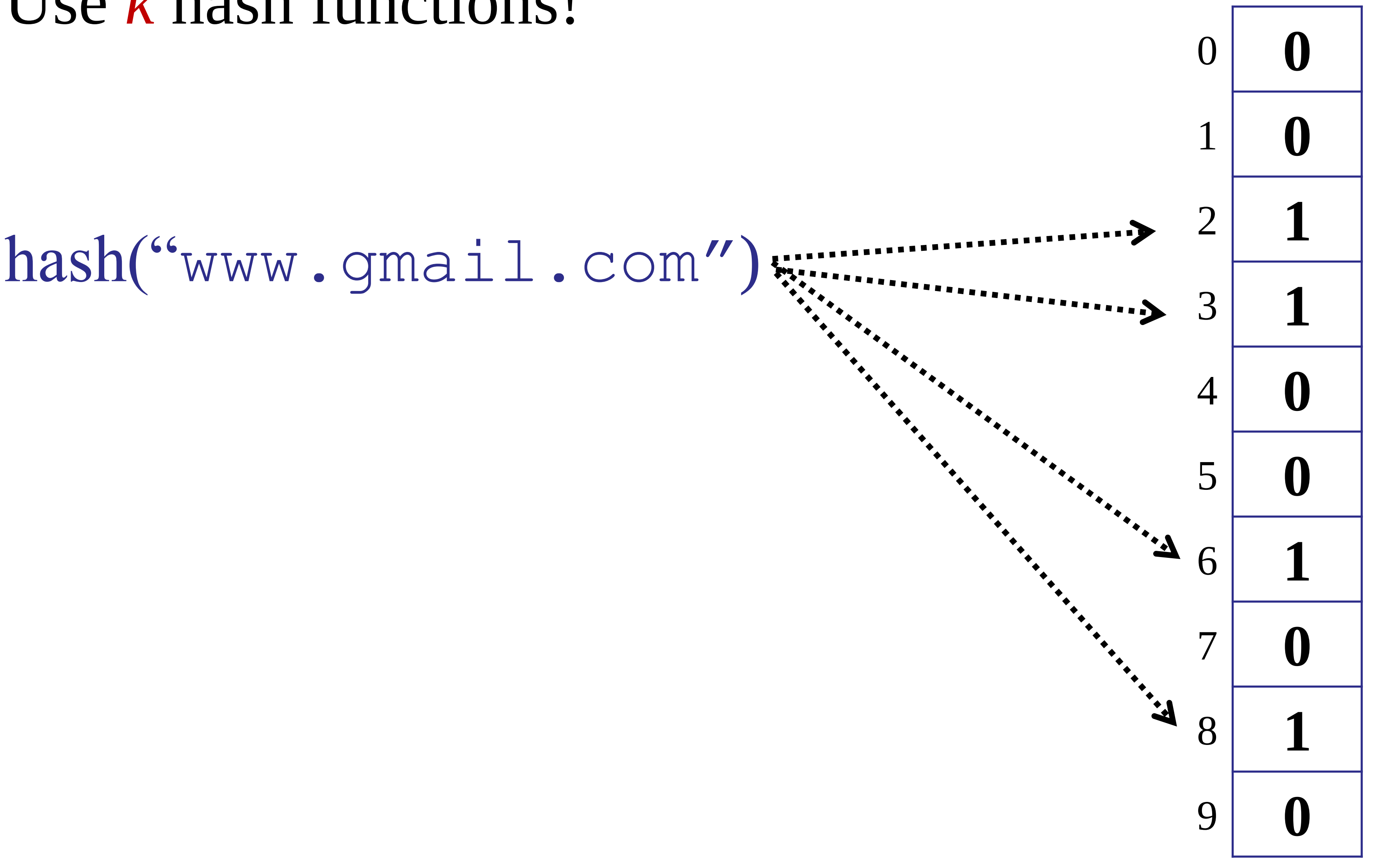
Bloom Filter



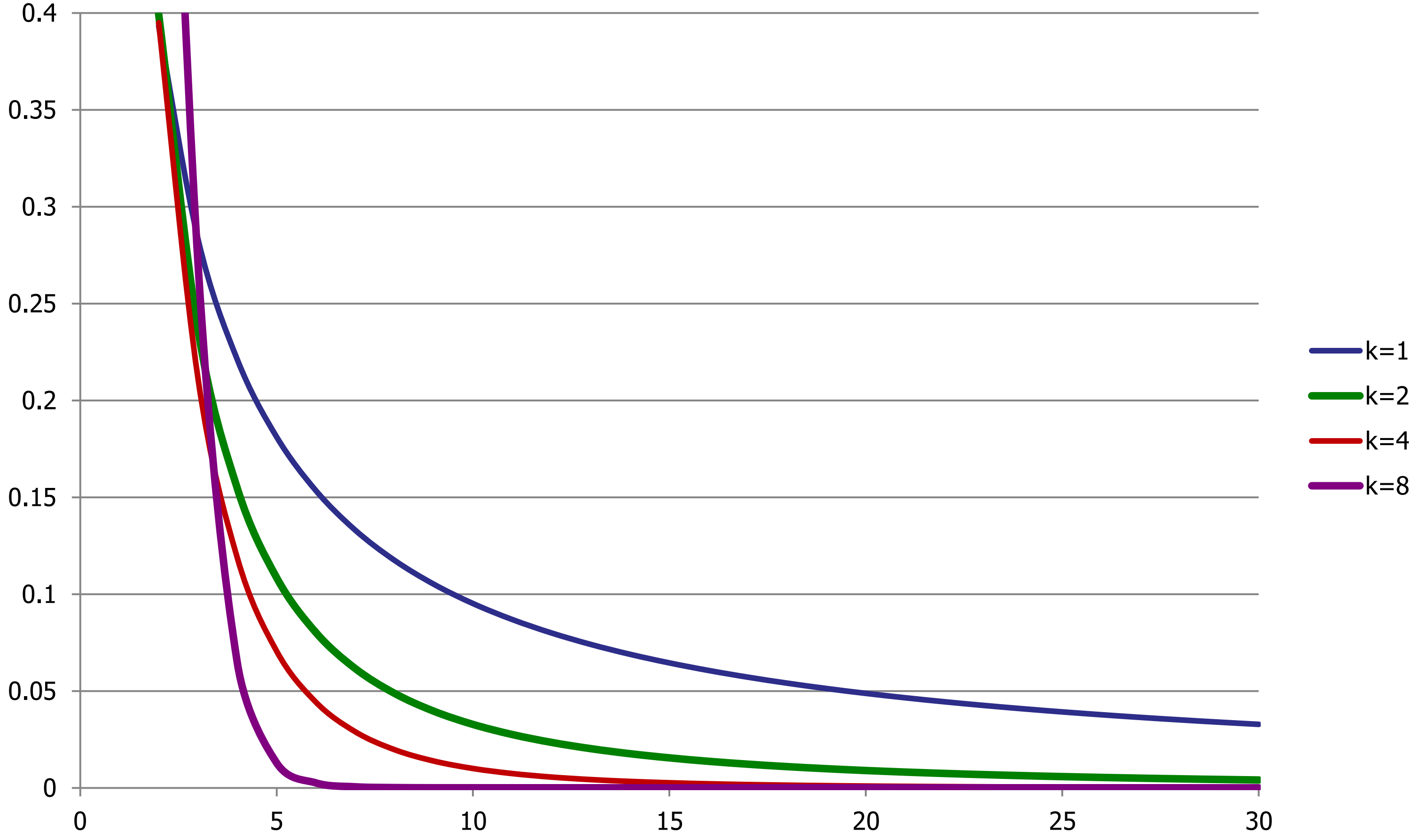
False positives rate vs. (m/n)

Bloom Filters

Use *k* hash functions!

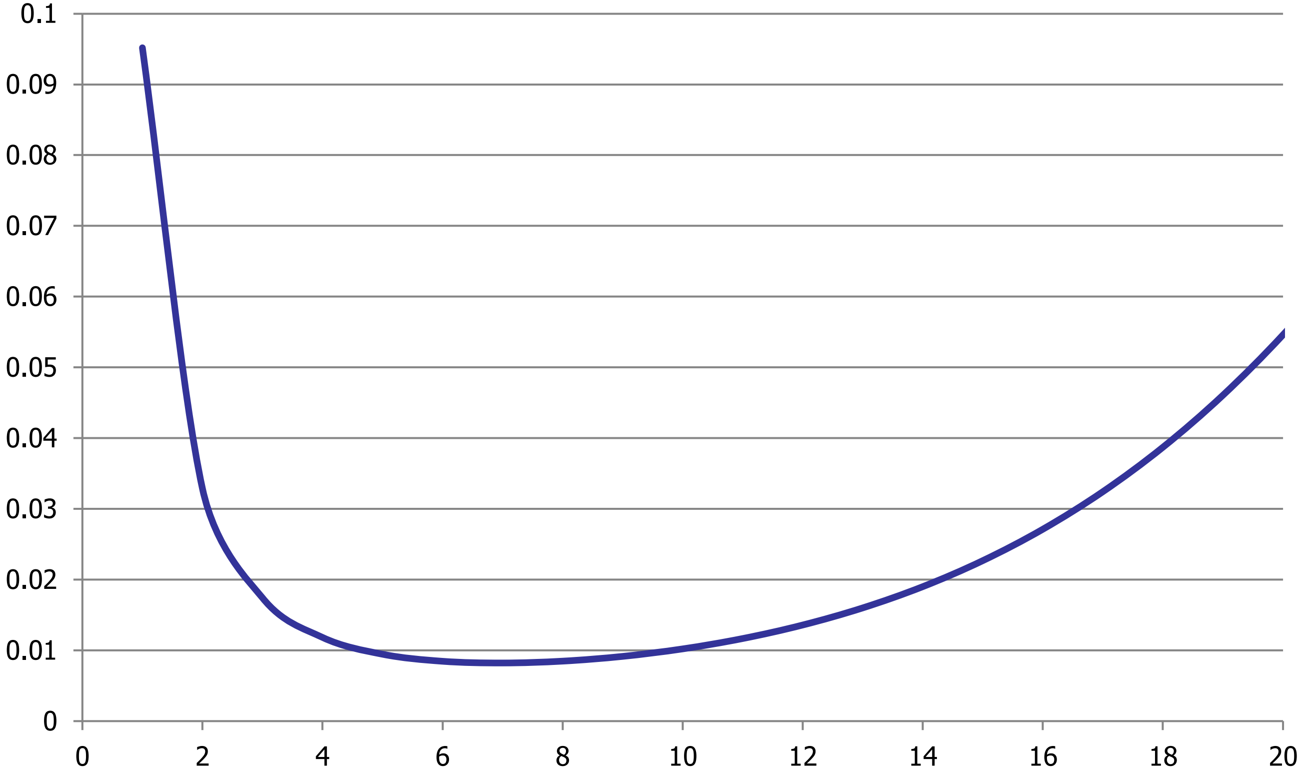


Bloom Filter



false positive rate vs. (m/n)

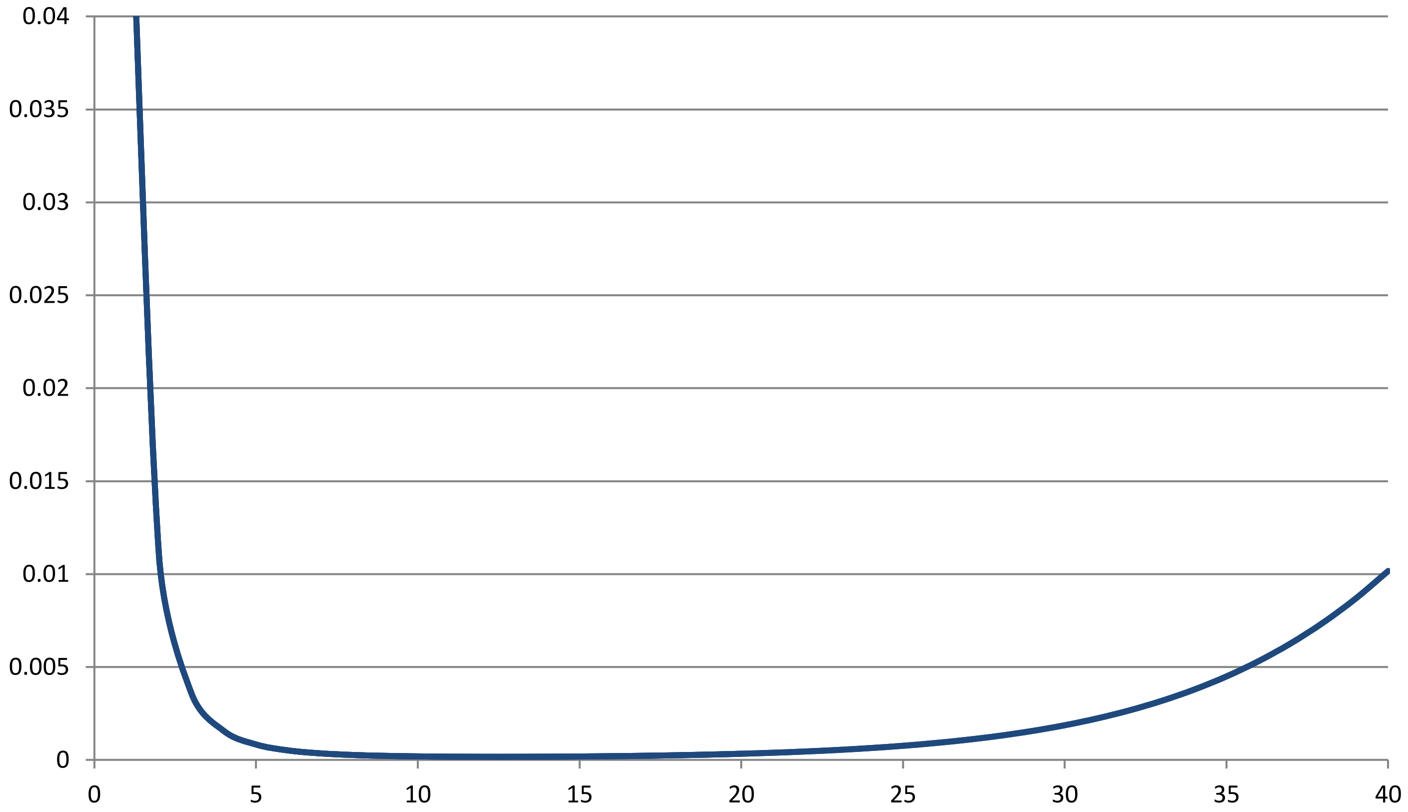
Bloom Filter



false positive rate vs *k*

m = 10n

Bloom Filter



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}

Some applications

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

Summary So Far

- 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

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