

# CS 600.226: Data Structures

## Michael Schatz

Nov 9, 2018

Lecture 30: Advanced Hash Tables



# HW7

## Assignment 7: Whispering Trees

Out on: November 2, 2018

Due by: November 9, 2018 before 10:00 pm

Collaboration: None

Grading:

Packaging 10%,

Style 10% (where applicable),

Testing 10% (where applicable),

Performance 10% (where applicable),

Functionality 60% (where applicable)

### Overview

The seventh assignment is all about ordered maps, specifically fast ordered maps in the form of balanced binary search trees. You'll work with a little program called Words that reads text from standard input and uses an (ordered) map to count how often different words appear. We're giving you a basic (unbalanced) binary search tree implementation of OrderedMap that you can use to play around with the Words program and as starter code for your own developments.

# HW8

## Assignment 8: Competitive Spelling Bee

Out on: November 2, 2018

Due by: November 9, 2018 before 10:00 pm

Collaboration: None

Grading:

Packaging 10%,

Style 10% (where applicable),

Testing 10% (where applicable),

Performance 30% (where applicable),

Functionality 40% (where applicable)

### Overview

Your "one" task for this assignment is to take the simple spell checker we give you and to turn it into the fastest, most memory-efficient spell checker in the course, subject to the constraints detailed below. You are expected to do this by (once again) implementing the Map interface, this time using one of several hash table techniques (your choice, see below).

# Part I: Hash Tables

# Maps, Sets, and Arrays

*Sets as Map<T, Boolean>*

Mike	-> True
Peter	-> True
Joanne	-> True
Zack	-> True
Debbie	-> True
Yair	-> True
Ron	-> True

*Array as Map<Integer, T>*

0	-> Mike
1	-> Peter
2	-> Joanne
3	-> Zack
4	-> Debbie
5	-> Yair
6	-> Ron

Maps are extremely flexible and powerful,  
and therefore are extremely widely used

Built into many common languages: Awk, Python, Perl, JavaScript...

*Could we do everything in  $O(\lg n)$  time or faster?*  
=> Balanced Search Trees

# Hashing

$\text{Array}[13] = 10; \text{Array}[42] = 15$

$O(1)$

$\text{Array}["Mike"] = 10; \text{Array}["Peter"] = 15$

$BST: O(\lg n) \rightarrow Hash: O(1)$

**Hash Function enables Map<K, V> as Map<Integer, V> as Array<V>**

- $h(): K \rightarrow \text{Integer}$  for any possible key K
- $h()$  should distribute the keys uniformly over all integers
- if  $k_1$  and  $k_2$  are “close”,  $h(k_1)$  and  $h(k_2)$  should be “far” apart

**Typically want to return a small integer, so that we can use it as an index into an array**

- An array with  $4B$  cells is not very practical if we only expect a few thousand to a few million entries
- How do we restrict an arbitrary integer  $x$  into a value up to some maximum value  $n$ ?

$$0 \leq x \% n < n$$

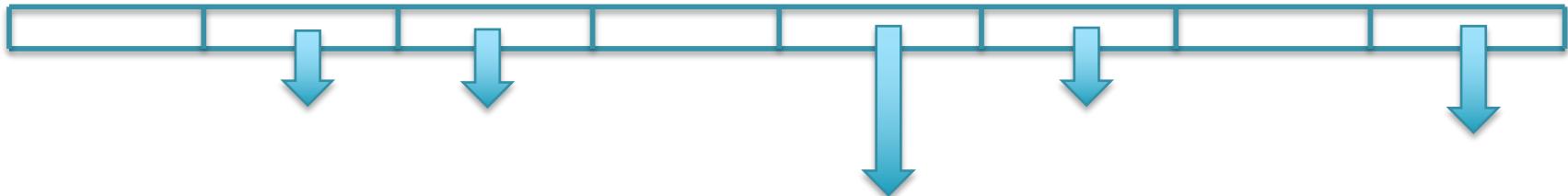
**Compression function:  $c(i) = \text{abs}(i) \% \text{length}(a)$**

Transforms from a large range of integers to a small range (to store in array a)

# Collision Strategies

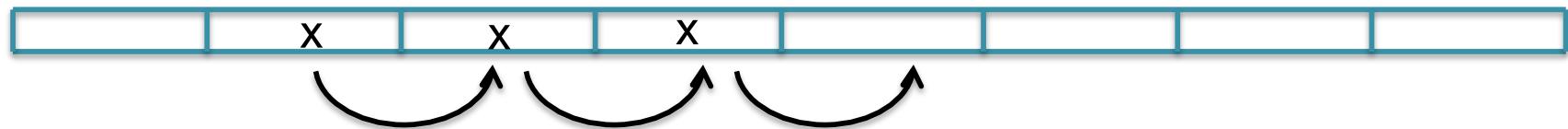
## 1. Separate chaining:

More object overhead, but degrades gracefully as load approaches 1



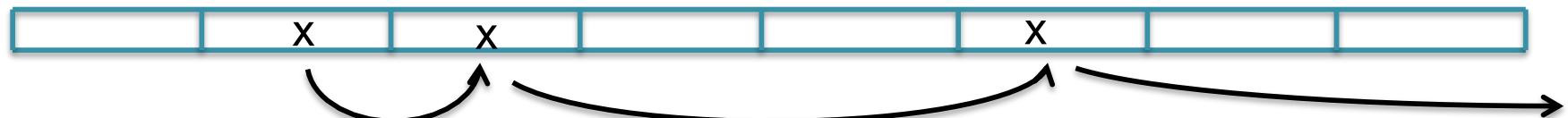
## 2. Linear Probing

Minimal overhead, easy to implement, but tends to form large clusters



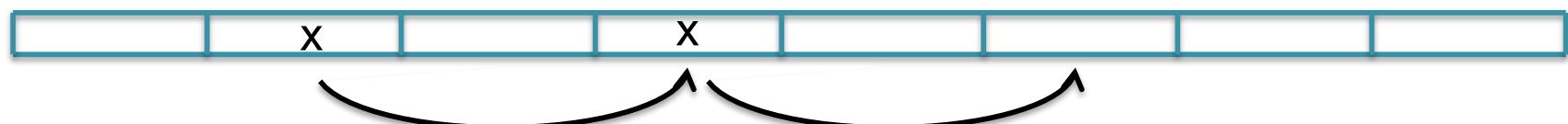
## 3. Quadratic Probing

Slightly more complex, but skips over larger and larger amounts to find holes



## 4. Double Hashing

Stride length depends on  $h_2(\text{key})$



# (Bad) String Hash Functions

**Hash Function enables Map<K, V> as Map<Integer, V> as Array<V>**

- $h(): K \rightarrow \text{Integer}$  for any possible key K
- $h()$  should distribute the keys uniformly over all integers
- if  $k_1$  and  $k_2$  are “close”,  $h(k_1)$  and  $h(k_2)$  should be “far” apart
- Should be fast, as we are aiming for  $O(1)$  performance
  - Unlike cryptographic hashes that are slow but give very few collisions

**Hash of String: First char**

- $h(s) = (\text{int}) s.\text{at}(0)$
- Super simple, but does not map to all possible integers, does not separate out values

**Hash of String: sum of chars**

$$h(s) = \sum_{i=0}^{s.\text{length}()-1} s.\text{at}(i)$$

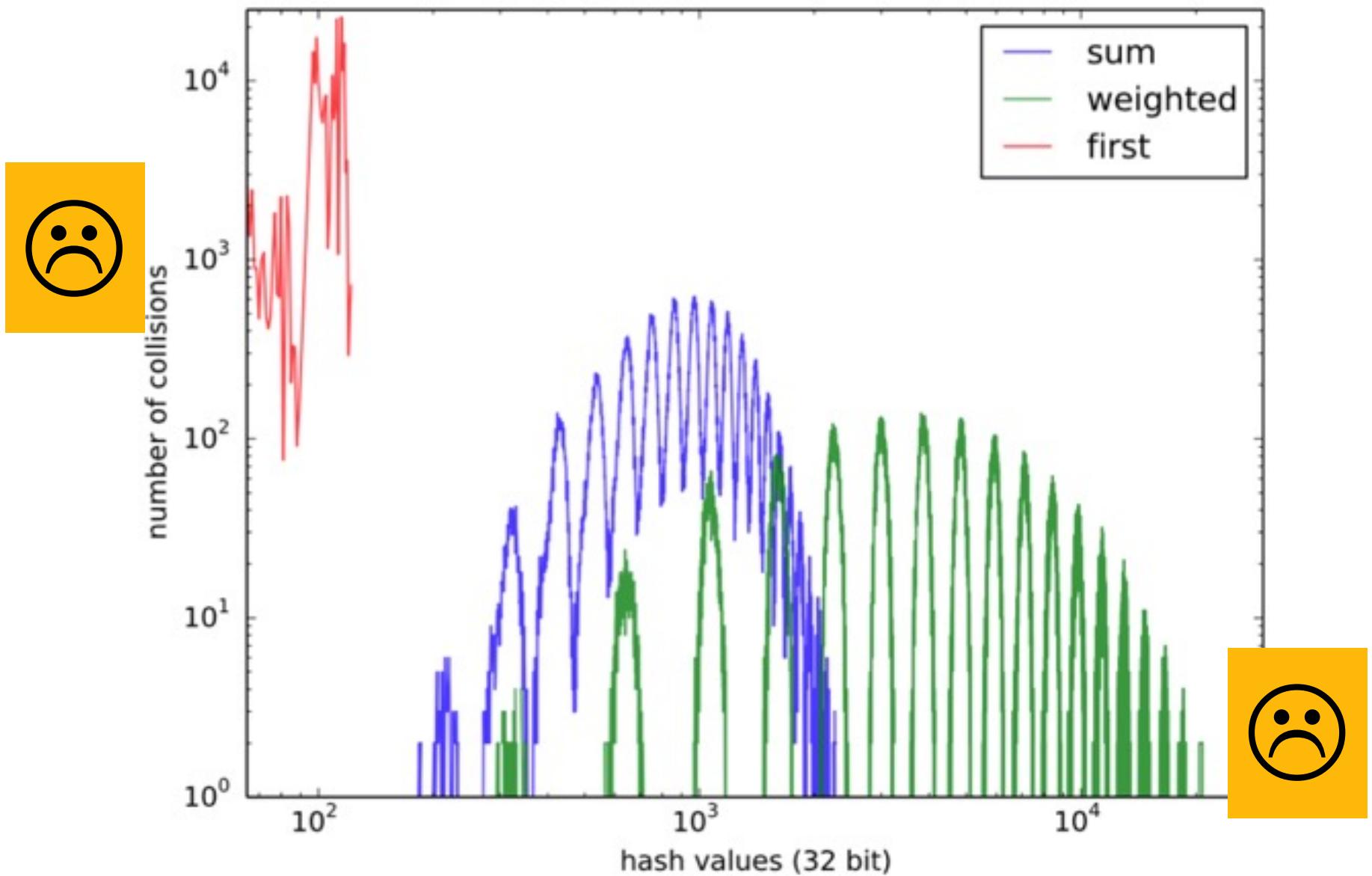
- Simple, but anagrams map to same value  
(stop, pots, tops, ..)

**Hash of String: weighted sum of chars**

$$h(s) = \sum_{i=0}^{s.\text{length}()-1} (i+1) \times s.\text{at}(i)$$

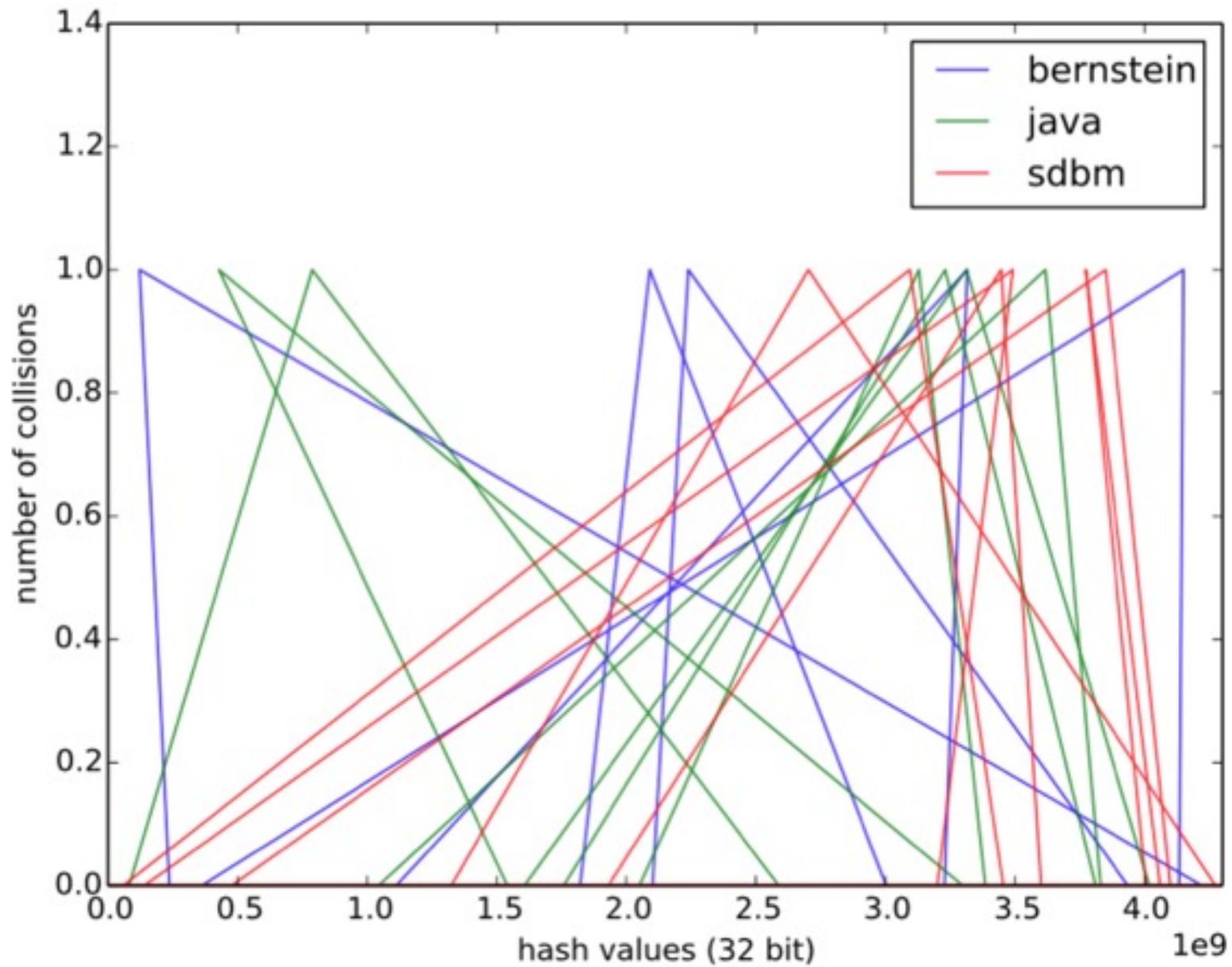
- Better: depends on both characters used and their order
- Small values for typical words

# Bad Hash Function Performance



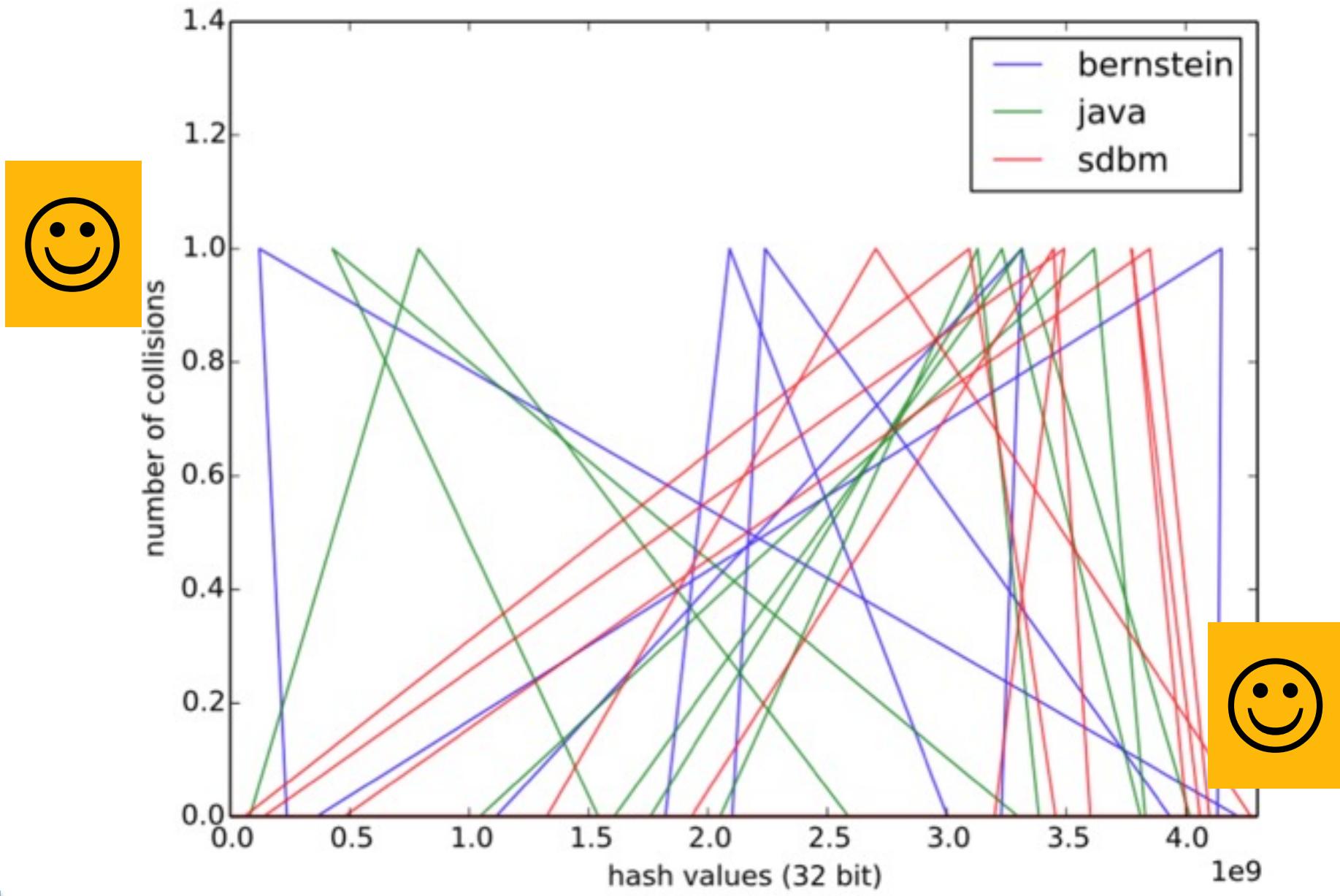
Hash every word in dictionary.txt and plot #collisions versus hash value

# Good Hash Function Performance



Hash every word in dictionary.txt and plot #collisions versus hash value

# Good Hash Function Performance



Hash every word in dictionary.txt and plot #collisions versus hash value

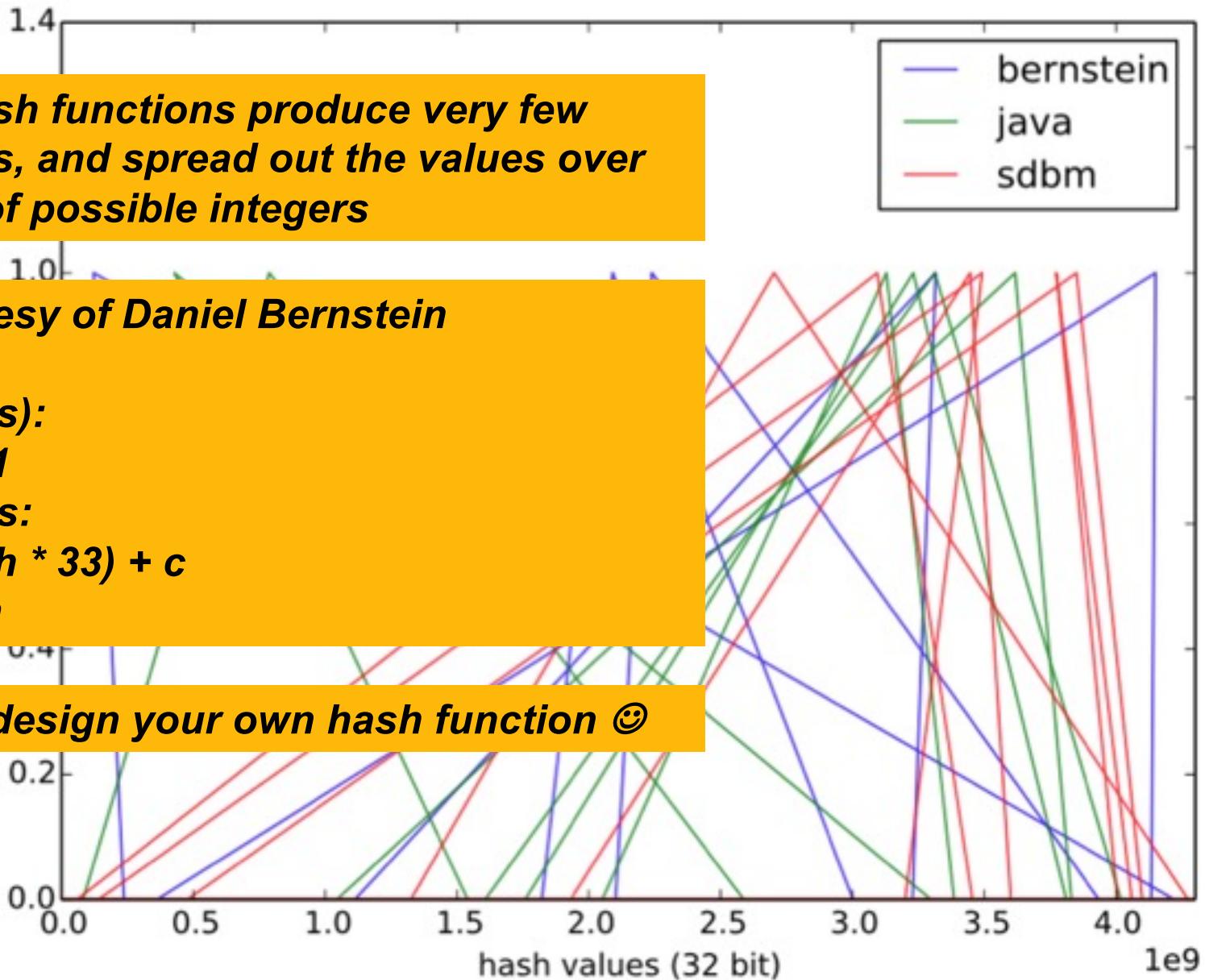
# Good Hash Function Performance

**Good hash functions produce very few collisions, and spread out the values over billions of possible integers**

**## Courtesy of Daniel Bernstein**

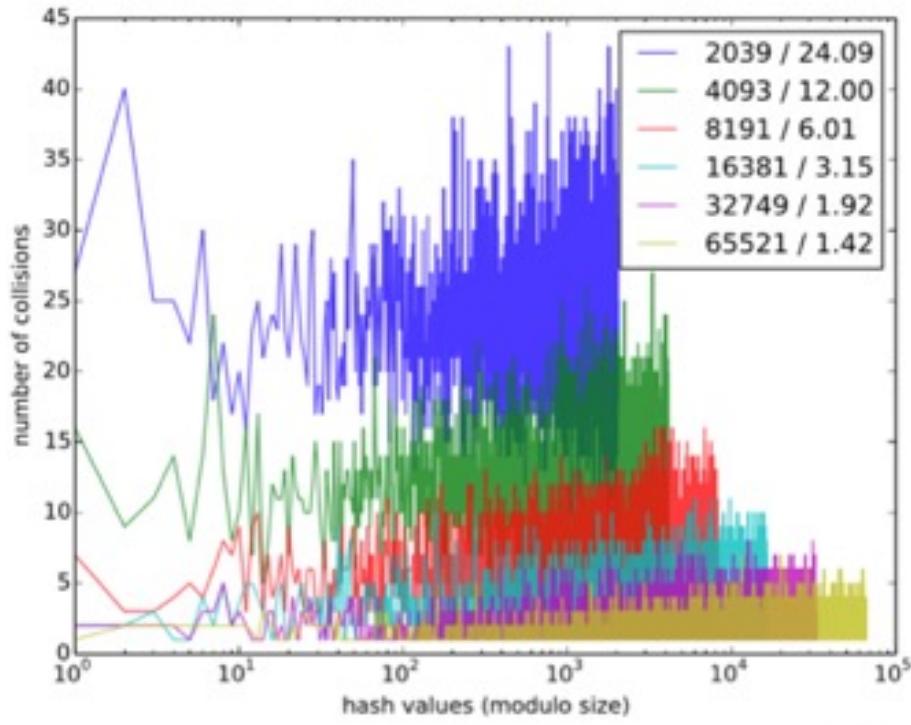
```
djbhash(s):  
    h = 5381  
    for c in s:  
        h = (h * 33) + c  
    return h
```

**Don't design your own hash function ☺**

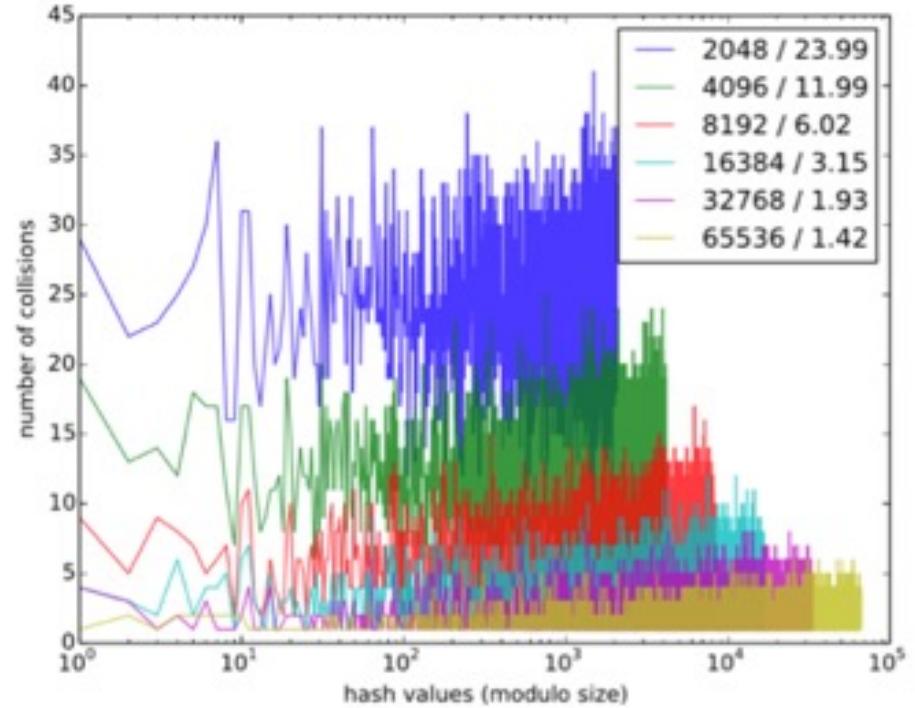


Hash every word in dictionary.txt and plot #collisions versus hash value

# Hash Table Size (Berstein Hash)



Prime Hash Table Sizes



Powers of Two

In theory, using a prime number as the size of your hash table will have fewer collisions, but in practice using powers of two works about as well.

The major exception is if you are using quadratic probing: with a load factor  $< 0.5$ , a prime size will guarantee you explore at least half of the possible slots, but a non-prime size make no guarantees

# Part 3: HashMap Implementation

# HashMap<K,V> (I)

```
import java.util.ArrayList;
import java.util.Iterator;

// doesn't resize/rehash

public class HashMap<K, V> implements Map<K, V> {

    private static class Entry<K, V> {
        K key;
        V value;

        Entry(K k, V v) {
            this.key = k;
            this.value = v;
        }

        public boolean equals(Object that) {
            return (that instanceof Entry)
                && (this.key.equals(((Entry) that).key));
        }

        public int hashCode() {
            return this.key.hashCode();
        }
    }
}
```

Uses an  
ArrayList<ArrayList>>  
to store the data

Nested  
Entry<K,V>  
class

Equals  
checks the  
key only

Use java hashCode()  
function on the key  
For Strings:

$$h(s) = \sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i}$$

# HashMap<K,V> (2)

```
// inner class, not nested! look ma, no static!
private class HashMapIterator implements Iterator<K> {
    private int returned = 0;
    private Iterator<ArrayList<Entry<K, V>>> outer;
    private Iterator<Entry<K, V>> inner;
    HashMapIterator() {
        this.outer = HashMap.this.data.iterator();
        this.inner = this.outer.next().iterator();
    }
    public boolean hasNext() {
        return this.returned < HashMap.this.size;
    }
    public K next() {
        if (this.inner.hasNext()) {
            this.returned += 1;
            return this.inner.next().key;
        } else {
            while (!this.inner.hasNext() && this.outer.hasNext()) {
                this.inner = this.outer.next().iterator();
            }
            if (this.inner.hasNext()) {
                this.returned += 1;
                return this.inner.next().key;
            } else {
                return null;
            }
        }
    }
    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

Iterator is a bit complicated because it has to iterate over the outer list buckets and then the inner list of Entry's in each bucket

# HashMap<K,V> (3)

```
private static final int INITIAL_SIZE = 8;
private ArrayList<ArrayList<Entry<K, V>>> data;
private Entry<K, V> fake;
private int size;

public HashMap() {
    this.data = new ArrayList<>(); // Java 7
    for (int i = 0; i < INITIAL_SIZE; i++) {
        this.data.add(new ArrayList<Entry<K, V>>());
    }
    this.fake = new Entry<K, V>(null, null);
}

public boolean has(K k) {
    return this.find(k) != null;
}

private Entry<K, V> find(K k) {
    int slot = this.hash(k);
    this.fake.key = k;
    int index = this.data.get(slot).indexOf(this.fake);
    if (index == -1) {
        return null;
    } else {
        return this.data.get(slot).get(index);
    }
}
```

Use 8 buckets,  
initialized with an  
empty list in each

fake helps with searching  
so we can call  
ArrayList.indexOf() which  
uses Entry.equals()

# HashMap<K,V> (4)

```
private int hash(Object o) {  
    return this.abs(o.hashCode()) % this.data.size();  
}  
  
private int abs(int i) {  
    if (i < 0) {  
        return -i;  
    } else {  
        return i;  
    }  
}  
  
public void put(K k, V v) throws UnknownKeyException {  
    Entry<K, V> e = this.findForSure(k);  
    e.value = v;  
}
```

hash() returns  
hashCode % num\_buckets

```
public V get(K k) throws UnknownKeyException {  
    Entry<K, V> e = this.findForSure(k);  
    return e.value;  
}
```

findForSure guarantees the  
reference is not null

# HashMap<K,V> (5)

```
private Entry<K, V> findForSure(K k) throws UnknownKeyException {  
    Entry<K, V> e = this.find(k);  
    if (e == null) {  
        throw new UnknownKeyException();  
    }  
    return e;  
}
```

findForSure guarantees the reference is not null

```
public void insert(K k, V v) throws DuplicateKeyException {  
    if (this.has(k)) {  
        throw new DuplicateKeyException();  
    }  
    Entry<K, V> e = new Entry<K, V>(k, v);  
    int slot = this.hash(k);  
    this.data.get(slot).add(e);  
    this.size += 1;  
}
```

Just adds the item to the appropriate bucket, but does NOT resize!

```
public V remove(K k) throws UnknownKeyException {  
    Entry<K, V> e = this.findForSure(k);  
    int slot = this.hash(k);  
    this.data.get(slot).remove(e);  
    this.size -= 1;  
    return e.value;  
}
```

Removes the item from the containing bucket using ArrayList.remove

```
public Iterator<K> iterator() {  
    return new HashMapIterator();  
}
```

# Benchmarks

## ***BinarySearchTreeMap***

```
## These are the standard, unbalanced binary search trees.
```

```
$ java Words <manifesto.txt >bla  
0.527914894  
$ java Words <wealth.txt >bla  
1.629583552  
$ java Words <war.txt >bla  
1.865477425  
$ java -Xss2048k Words <cracklib.txt >bla  
43.321007802  
$ java -Xss16384k Words <dictionary.txt >bla  
2905.769206485
```

## ***HashMap***

```
## First for a bucket array with only eight slots
```

```
$ java Words <manifesto.txt >bla  
0.612431685  
$ java Words <wealth.txt >bla  
4.413707022  
$ java Words <war.txt >bla  
7.679160966  
$ java Words <cracklib.txt >bla  
16.050761321  
$ java Words <dictionary.txt >bla  
541.798612936
```

Even a small  
hash table  
outperforms a  
naïve BST

# Benchmarks

## **HashMap**

```
## Now for 4096 slots
## That's still a pretty high load factor:
## dictionary.txt has 234937 distinct words so alpha = 57.36

$ java Words <manifesto.txt >bla
0.529792145
$ java Words <wealth.txt >bla
1.485476444
$ java Words <war.txt >bla
1.647926909
$ java Words <cracklib.txt >bla
1.562509081
$ java Words <dictionary.txt >bla
4.221646673
```

## **AvlTreeMap**

```
$ java Words <manifesto.txt >bla
0.64004931
$ java Words <wealth.txt >bla
1.744705285
$ java Words <war.txt >bla
2.062956075
$ java Words <cracklib.txt >bla
2.256693694
$ java Words <dictionary.txt >bla
7.379325237
```

HashMap is noticeably faster than AVLTreeMap even though AVLTreeMap guarantees O(lg n) performance

# Benchmarks

## ***HashMap***

```
## Now for 4096 slots
## That's still a pretty high load factor:
## dictionary.txt has 234937 distinct words so alpha = 57.36

$ java Words <manifesto.txt >bla
0.529792145
$ java Words <wealth.txt >bla
1.485476444
$ java Words <war.txt >bla
1.647926909
$ java Words <cracklib.txt >bla
1.562509081
$ java Words <dictionary.txt >bla
4.221646673
```

## ***TreapMap***

```
$ java Words <manifesto.txt >bla
0.543315182
$ java Words <wealth.txt >bla
1.788253795
$ java Words <war.txt >bla
2.196759232
$ java Words <cracklib.txt >bla
1.5308888
$ java Words <dictionary.txt >bla
3.602697315
```

The only thing faster  
than HashMap is the  
TreapMap

Random almost  
always wins ☺

## Part 4: Advanced Topics

# Separate Chaining Analysis

**Assume the table has just 1 cell:**

All n items will be in a linked list => O(n) insert/find/remove 😞

**Assume the hash function  $h()$  always returns a constant value**

All n items will be in a linked list => O(n) insert/find/remove 😞

**Assume table has  $m$  cells AND  $h()$  evenly distributes the keys**

- Every cell is equally likely to be selected to store key k, so the n items should be evenly distributed across m slots
- Average number of items per slot:  $n/m$ 
  - Also called the *load factor* (commonly written as  $\alpha$ )
  - Also the probability of a collision when inserting a new key
    - Empty table:  $0/m$  probability
    - After 1<sup>st</sup> item:  $1/m$
    - After 2<sup>nd</sup> item:  $2/m$

**Expected time for unsuccessful search:**

$O(1+n/m)$

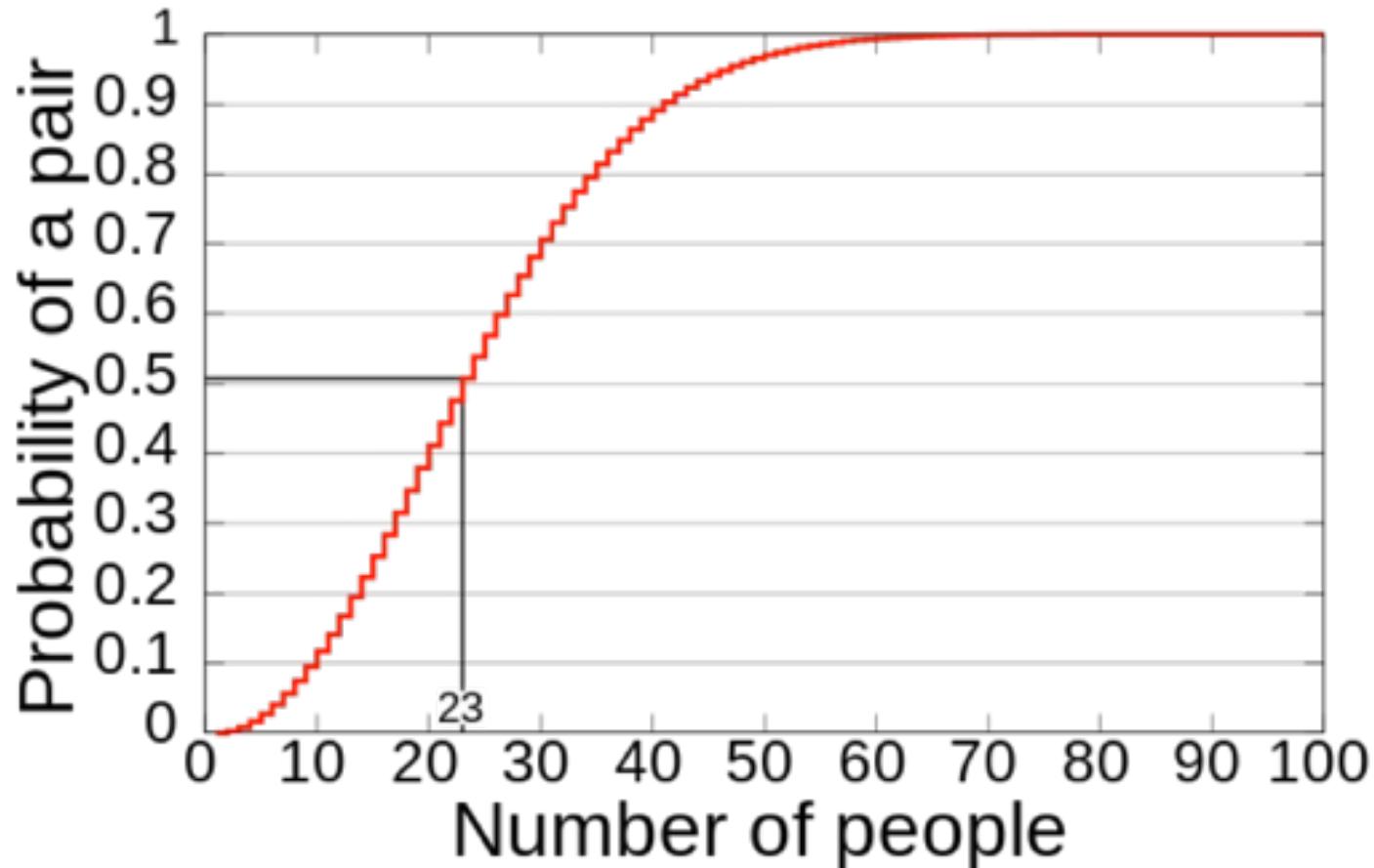
**Expected time for successful search:**

$O(1+n/m/2) \Rightarrow O(1+n/m)$

**If  $n < c*m$ , then we can expect constant time! 😊**

# Birthday Paradox

Some collisions are essentially unavoidable at modest load



There is a  $364/365$  chance 2 people don't share the same birthday

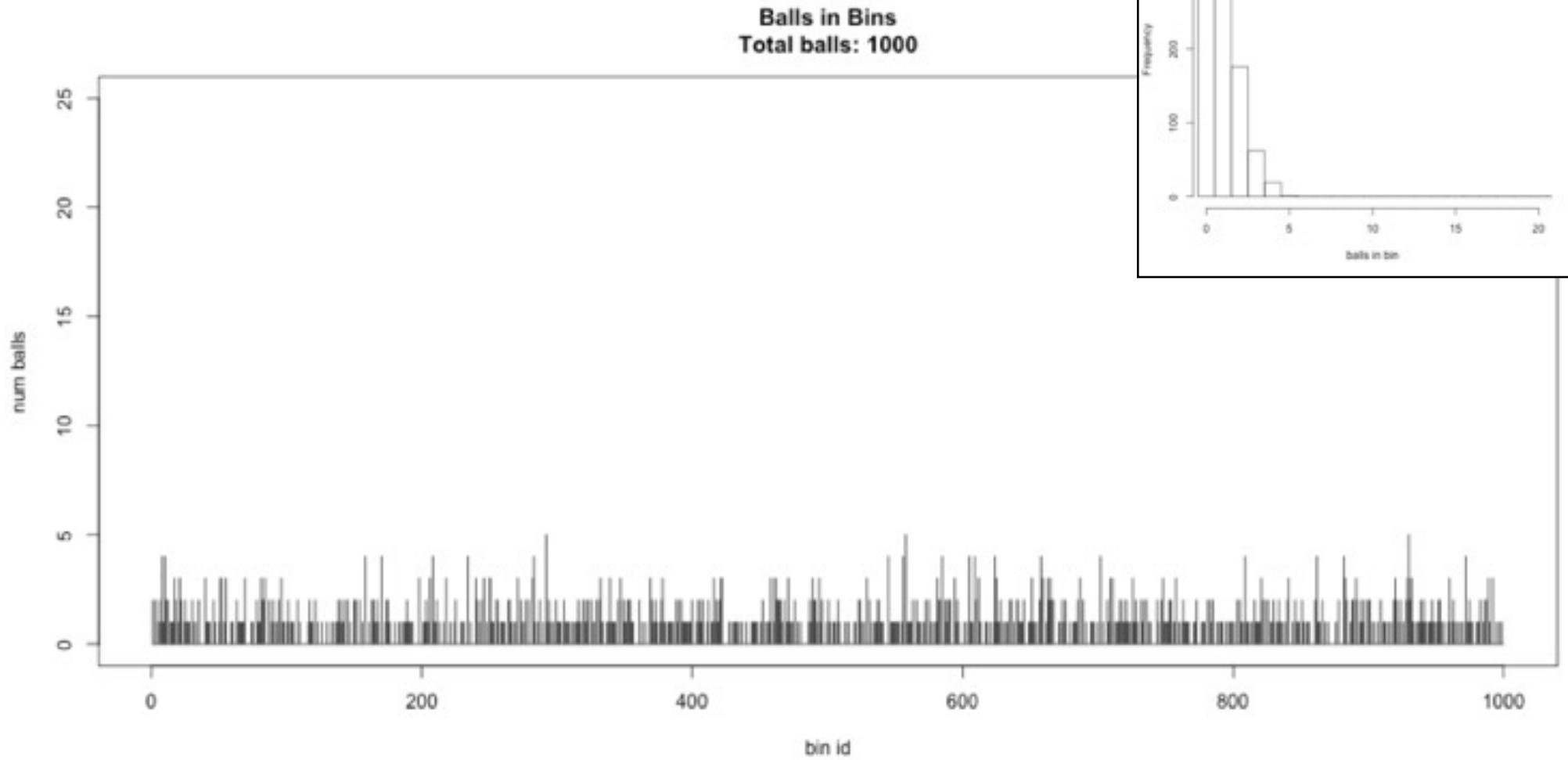
There is a  $364/365 * 363/365$  chance 3 people don't share

There is a  $364/365 * 363/365 * 362/365$  chance 4 people don't share

...

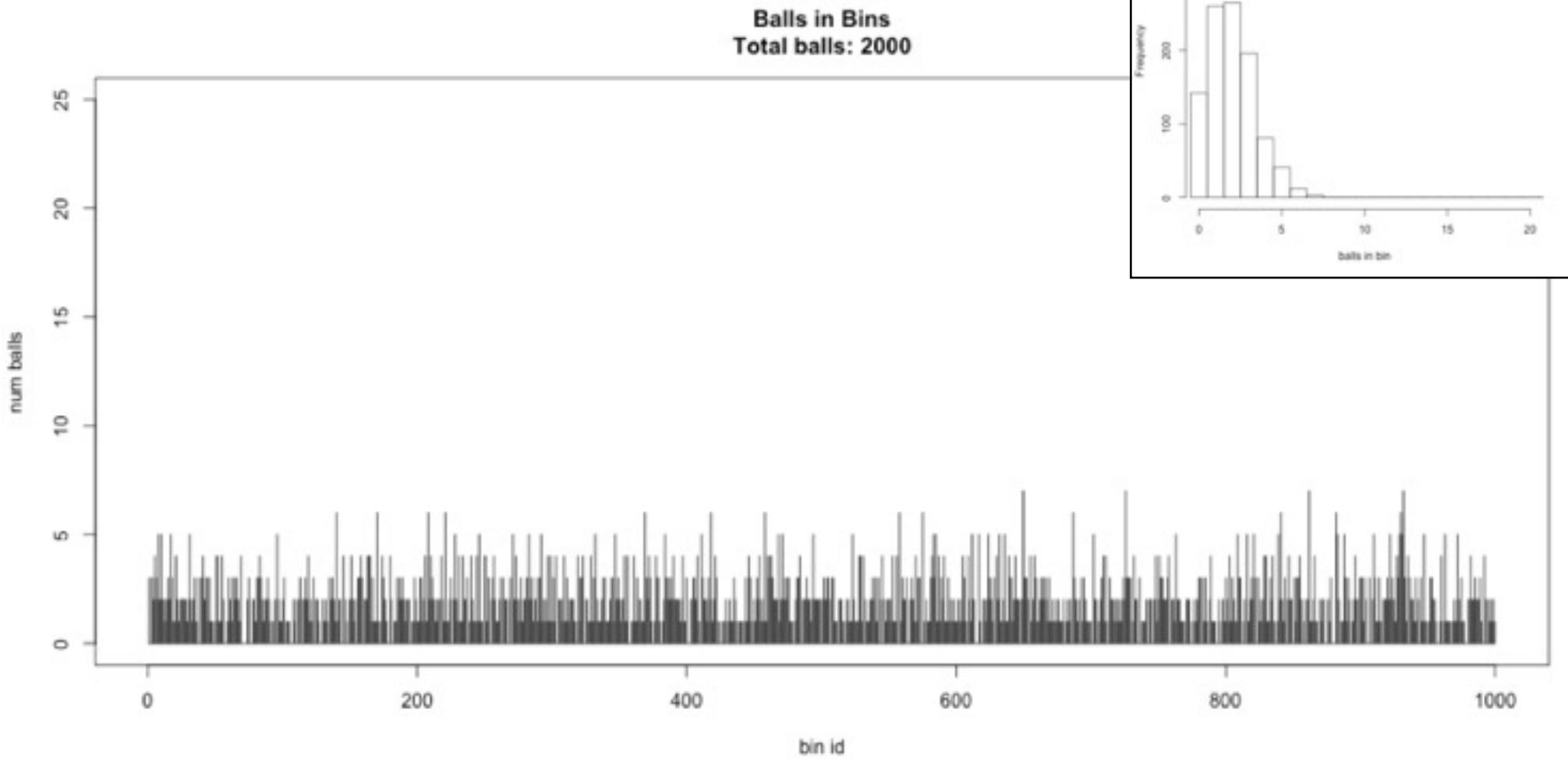
There is a >90% chance two people in this room share the same birthday

# Load = 1



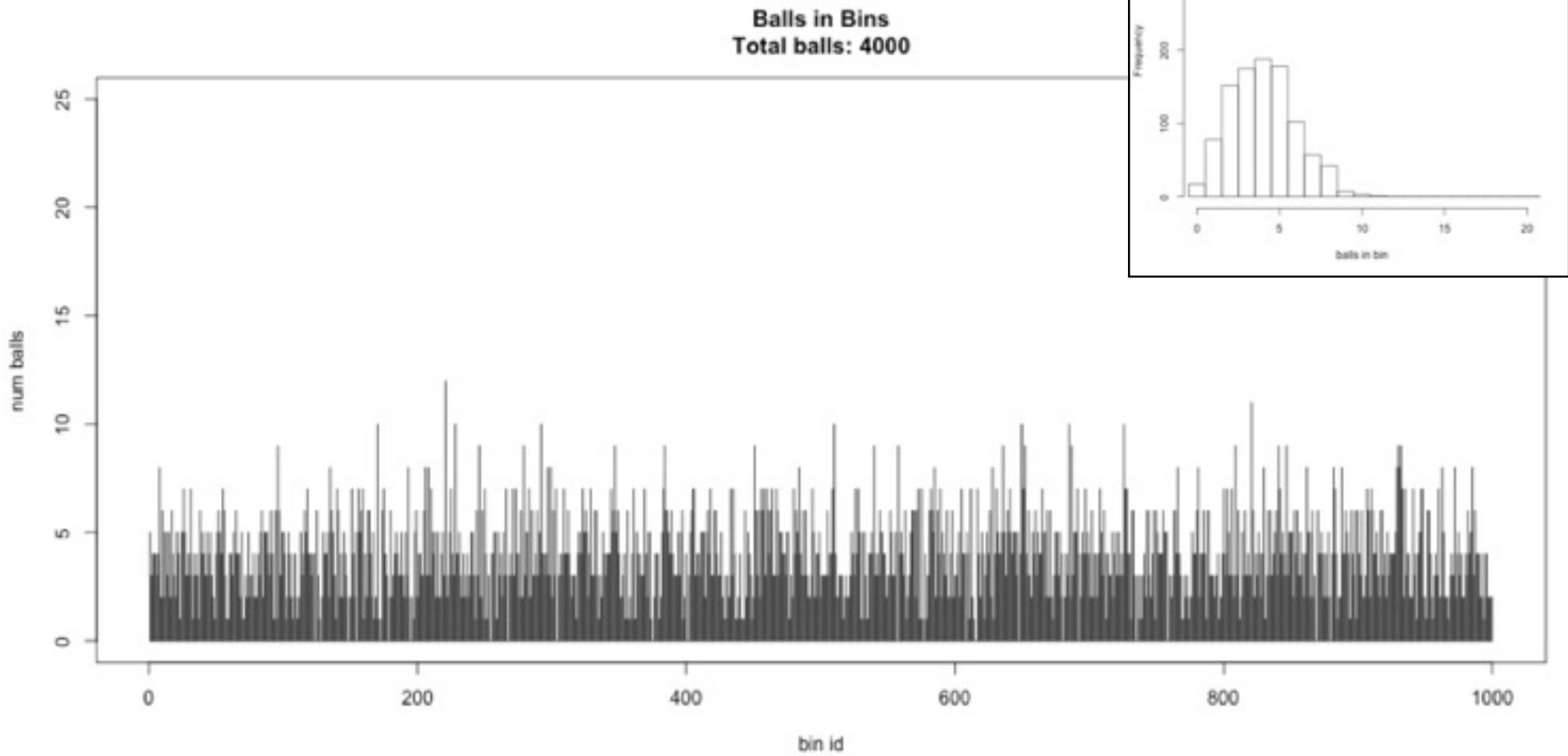
On average, there will be 1 entry per cell, but by random chance about 35% of the slots are empty, about 38% have 1 entry, and 27% have 2 or more

# Load = 2



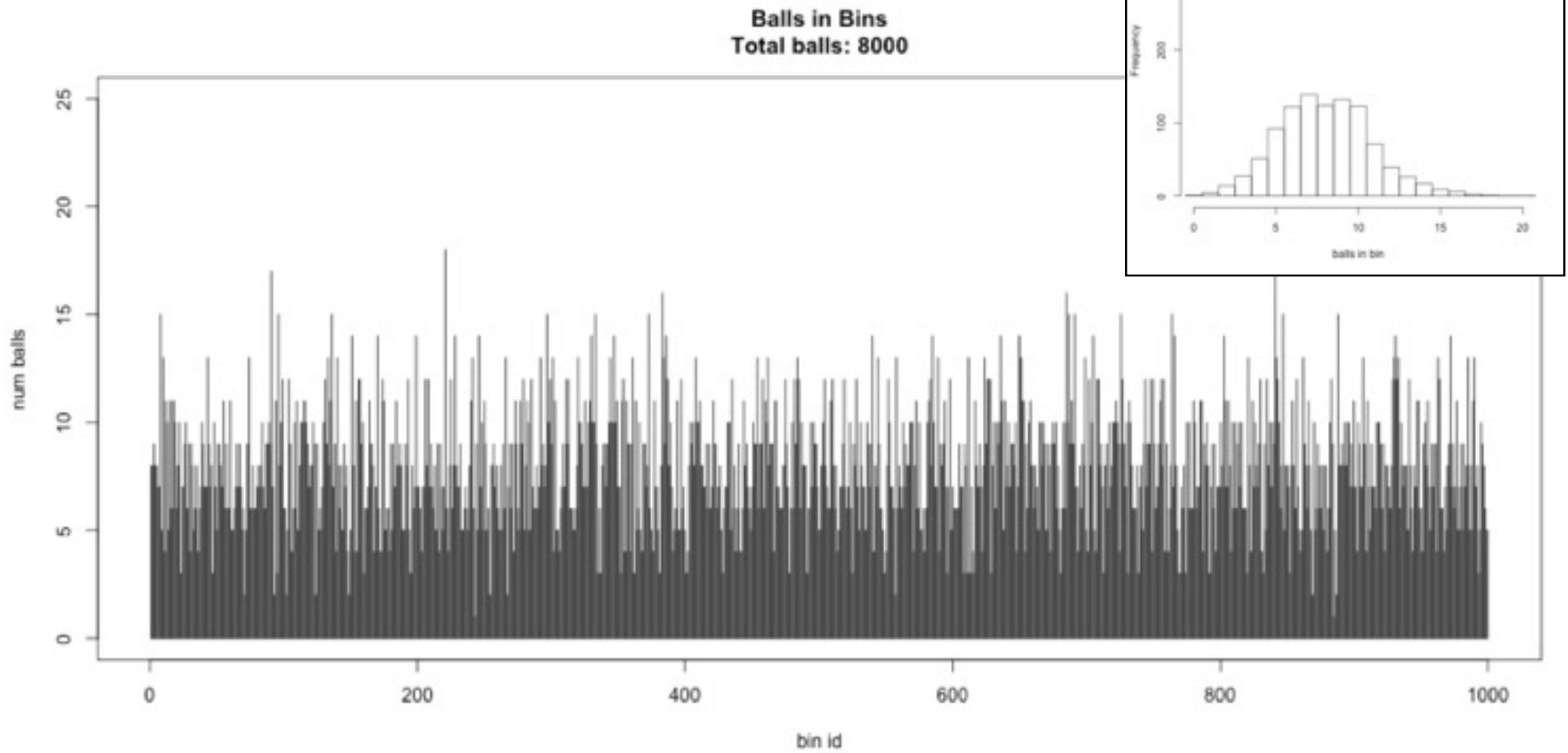
On average, there will be 2 entries per cell, but by random chance ~15% of the slots are empty, ~25% have 1 entry, ~26% have 2 entries, and 34% have 3 or more

# Load = 4



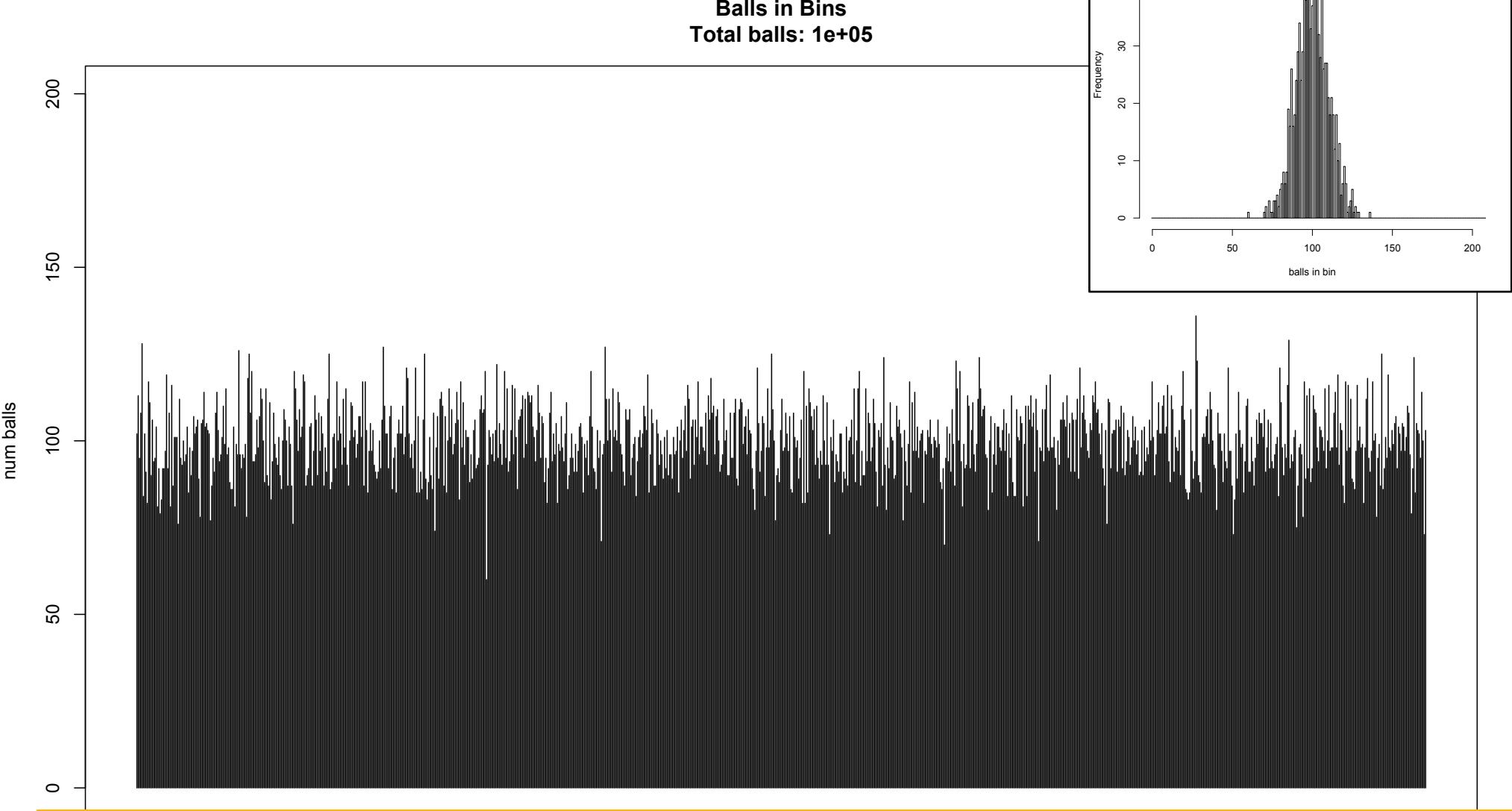
On average, there will be 4 entries per cell, but by random chance ~2% of the slots are empty, ~8% have 1 entry, ~15% have 2, 17% have 3, 18% have 4, and 40% have more than 4

# Load = 8



On average, there will be 8 entries per cell, but by random chance ~0.1% of the slots are empty, about ~30% have over 8, and some may have 20 entries

# Load = 100



At high load we expect every slot to be used, although some slots may have 150 or more entries

# Poisson Distribution

The probability of a given number of events occurring in a fixed interval of time and/or space if these events occur with a known average rate and independently of the time since the last event.

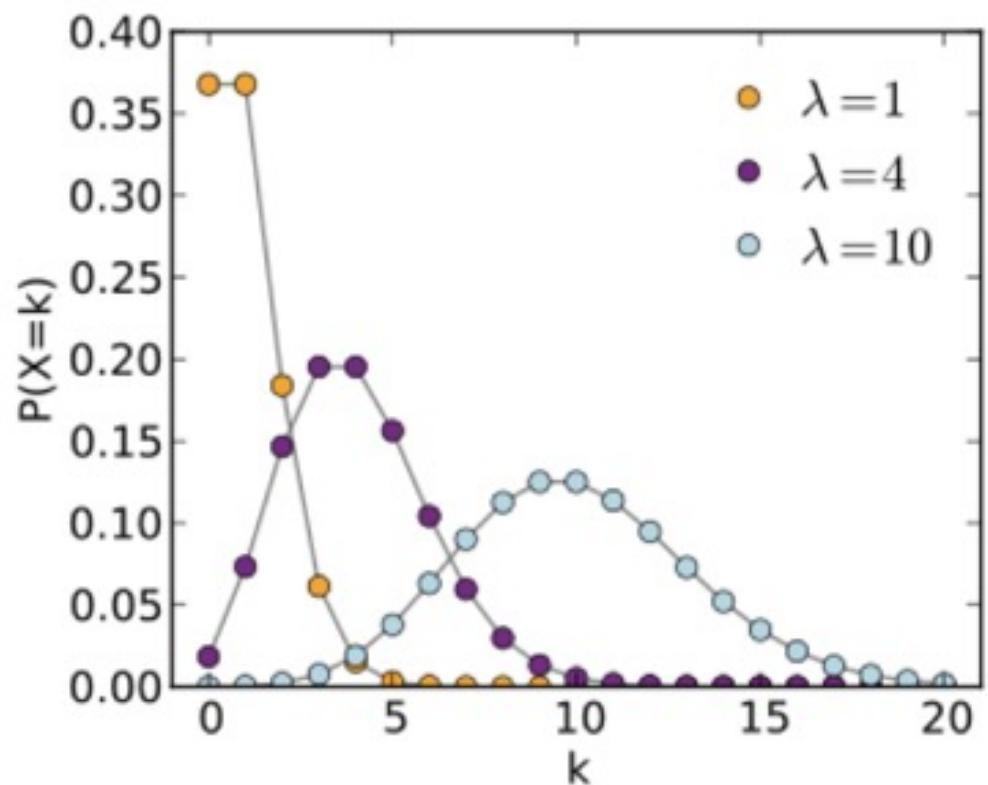
Formulation comes from the limit of the binomial equation

Resembles a normal distribution, but over the positive values, and with only a single parameter.

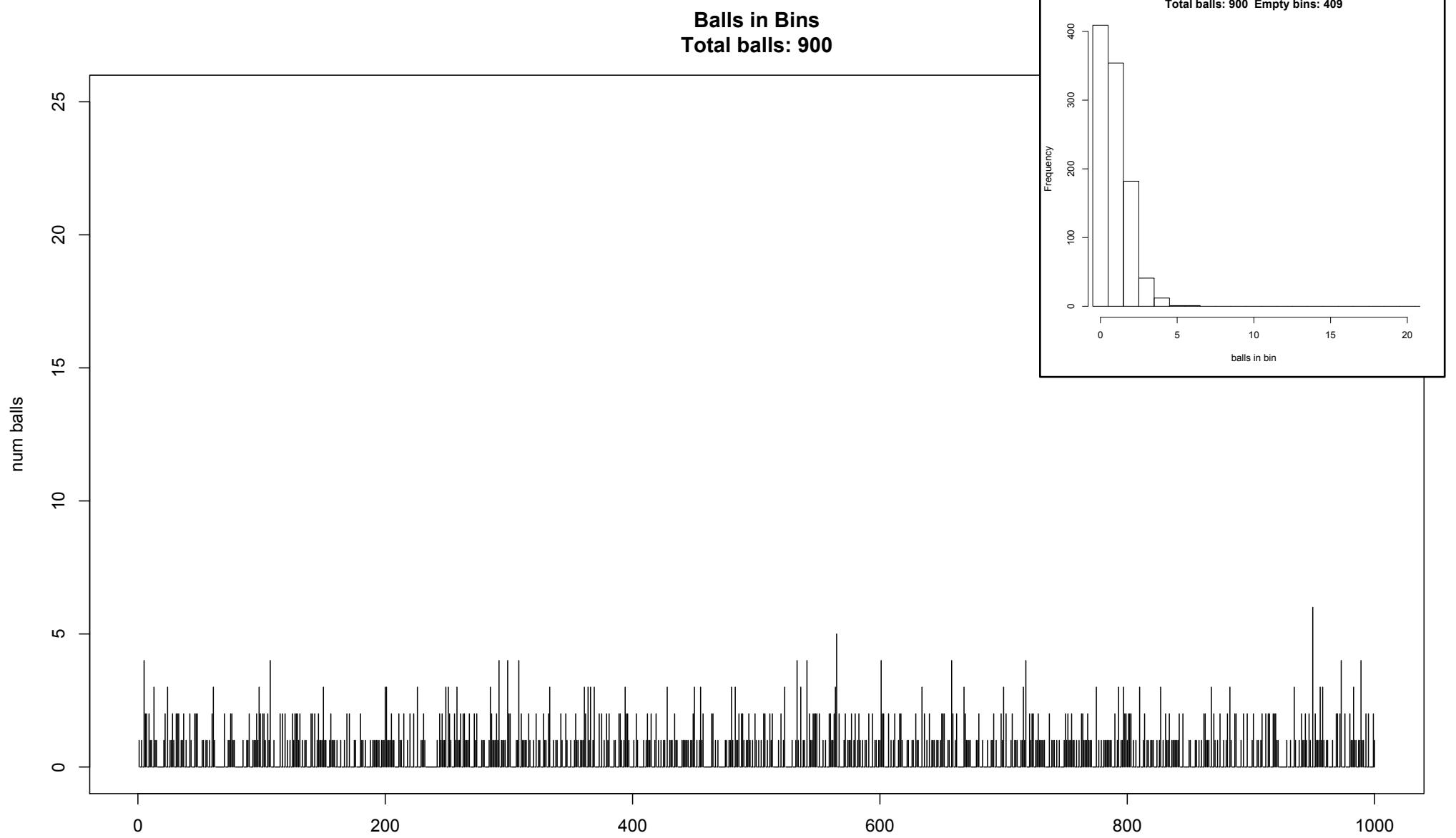
## **Key property:**

- ***The standard deviation is the square root of the mean.***

$$P(k) = \frac{\lambda^k}{k!} e^{-\lambda}$$



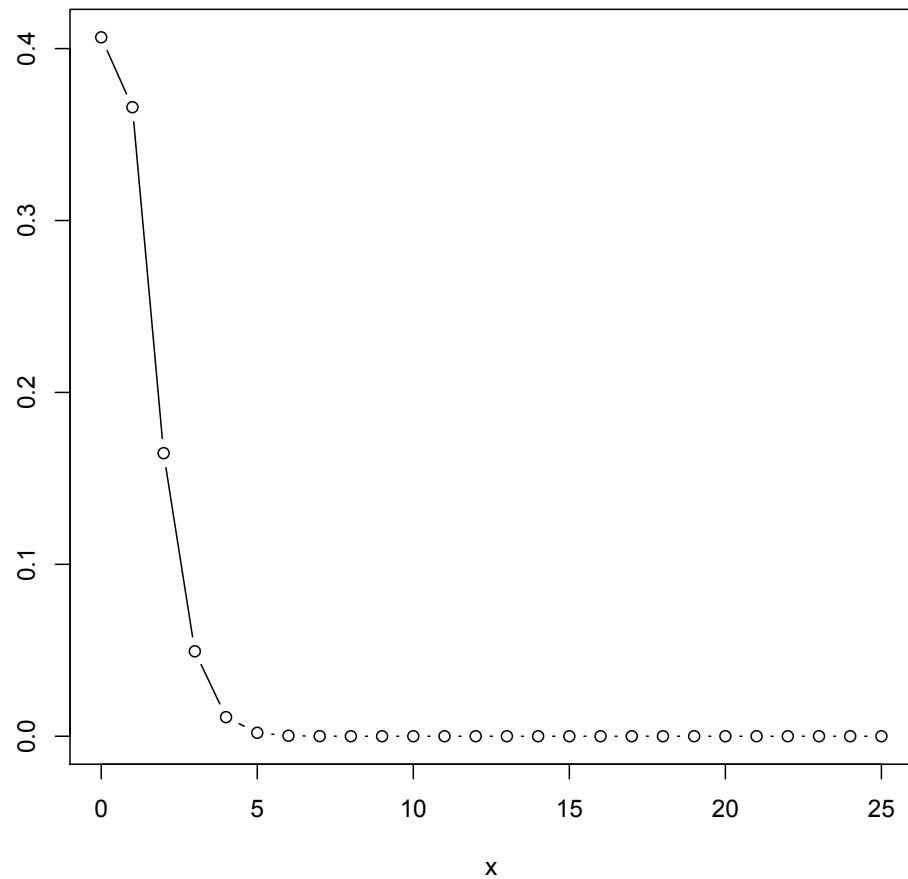
# Load = .9



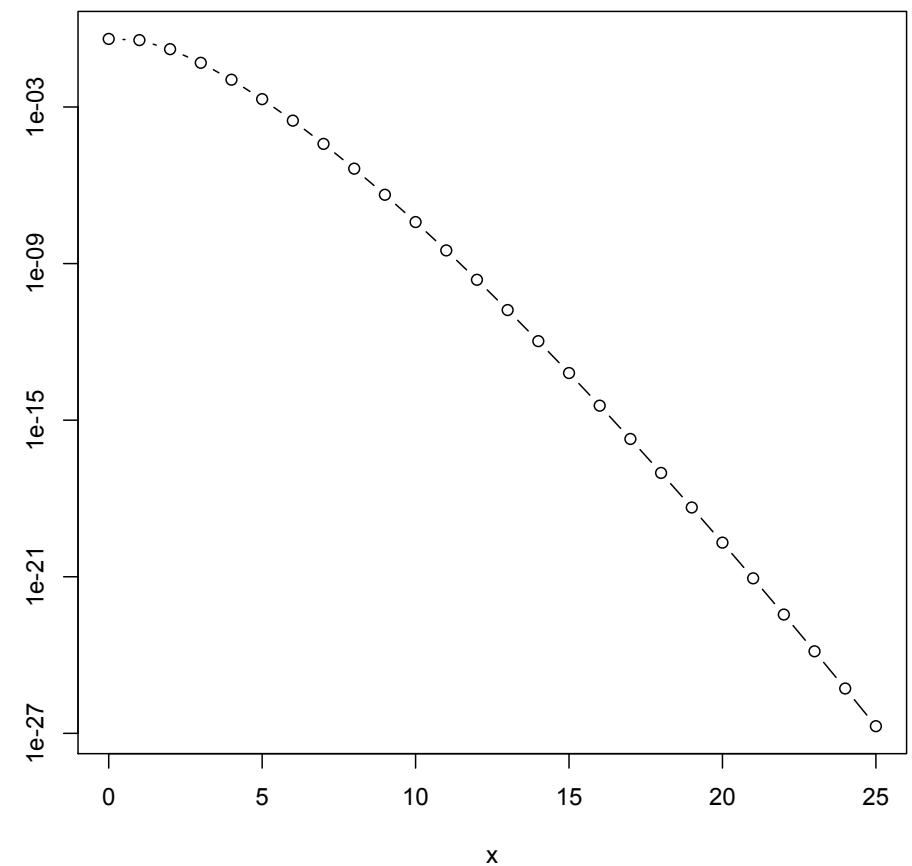
With 90% load, about 40% of the slots will be empty  
and most buckets have <5 entries

# Load = .9

Probability of  $x$  balls in a single bin (load=0.9)



Probability of  $x$  balls in a single bin (load=0.9)



Probabilities get exceedingly small to have large number of entries in 1 bucket  
Expected worse-case cost of a lookup in a chained hash table is  $O(\lg n / \lg \lg n)$

# Denial of Service Attacks

The screenshot shows a web browser window with the URL [www.ocert.org/advisories/ocert-2011-003.html](http://www.ocert.org/advisories/ocert-2011-003.html). The page is titled "oCERT Advisories" and displays an advisory for issue #2011-003. The advisory is about multiple implementations denial-of-service via hash algorithm collision. It includes sections for Description, Affected version, and a list of affected languages and frameworks.

#2011-003 multiple implementations denial-of-service via hash algorithm collision

Description:

A variety of programming languages suffer from a denial-of-service (DoS) condition against storage functions of key/value pairs in hash data structures, the condition can be leveraged by exploiting predictable collisions in the underlying hashing algorithms.

The issue finds particular exposure in web server applications and/or frameworks. In particular, the lack of sufficient limits for the number of parameters in POST requests in conjunction with the predictable collision properties in the hashing functions of the underlying languages can render web applications vulnerable to the DoS condition. The attacker, using specially crafted HTTP requests, can lead to a 100% of CPU usage which can last up to several hours depending on the targeted application and server performance, the amplification effect is considerable and requires little bandwidth and time on the attacker side.

The condition for predictable collisions in the hashing functions has been reported for the following language implementations: [Java](#), [jRuby](#), [PHP](#), [Python](#), [Rubinius](#), [Ruby](#). In the case of the Ruby language, the 1.9.x branch is not affected by the predictable collision condition since this version includes a randomization of the hashing function.

The vulnerability outlined in this advisory is practically identical to the one reported in 2003 and described in the paper [Denial of Service via Algorithmic Complexity Attacks](#) which affected the Perl language.

The reporter's own advisory can be found at [http://www.nruns.com/\\_downloads/advisory28122011.pdf](http://www.nruns.com/_downloads/advisory28122011.pdf).

Affected version:

- [Java](#), all versions
- [jRuby](#) <= 1.6.5
- [PHP](#) <= 5.3.8, <= 5.4.0RC3
- [Python](#) <= 2.6.7, <= 2.7.3, <= 3.1.4, <= 3.2.2
- [Rubinius](#), all versions
- [Ruby](#) <= 1.8.7-p356
- [Apache Geronimo](#), all versions
- [Apache Tomcat](#) <= 5.5.34, <= 6.0.34, <= 7.0.22
- [Oracle Glassfish](#) <= 3.1.1
- [Jetty](#) <= 7.5.4
- [Plone](#), all versions
- [Rack](#) <= 1.3.5, <= 1.2.4, <= 1.1.2

However, if an adversary knows what kind of hash table we are using, can force all the entries into a single bucket and slow everything down to linear time

# Perfect hashing

## **Perfect Hashing**

- If all keys are known ahead of time, a perfect hash function can be constructed to ***create a perfect hash table that has no collisions.***
- Perfect hashing allows for constant time lookups in all cases
- The original construction of Fredman, Komlós & Szemerédi (1984) uses a two-level scheme to map a set  $S$  of  $n$  elements to a range of  $O(n)$  indices, and then map each index to a range of hash values.
  - ***Relatively expensive to compute:*** key parameters found in polynomial time by choosing values randomly until finding one that works; but once computed can be used over and over and over again ☺
  - The hash function requires  $O(n)$  storage space

## **Minimal Perfect Hashing**

- If minimal perfect hashing is used, every slot in the hash table is used
- The best currently known minimal perfect hashing schemes can be represented using approximately 2.6 bits per key

***Guaranteed constant time and linear space lookups ☺***

# Universal hashing

## ***Universal Hashing***

- If all keys are not known ahead of time, ***an adversary may pick the keys so that all of them hash to the same bin!***
- Even without an adversary, you may get unlucky and you will have more collisions than desired, but just rehashing the data with the same function wont help
- The solution to these problems is to pick a function randomly from a family of hash functions!

- ***A family of functions  $H = \{h : U \rightarrow [m]\}$  is called a universal family if***

$$\forall x, y \in U, x \neq y : \Pr_{h \in H} [h(x) = h(y)] \leq \frac{1}{m}$$

## ***Universal Hash Function for Integers***

- The original proposal of Carter and Wegman (1979) was to pick a prime  $p > m$  and define:

$$h_{a,b}(x) = ((ax + b) \bmod p) \bmod m$$

where  $a, b$  and randomly chosen integers modulo  $p$

# Cuckoo Hashing



# Cuckoo Birds

These species are ***obligate brood parasites***, meaning that they only ***reproduce by laying their eggs in the nests of other birds***. The best-known example is the European common cuckoo.

***The cuckoo egg hatches earlier than the host's, and the cuckoo chick grows faster; in most cases the chick evicts the eggs or young of the host species.***



The chick has no time to learn this behavior, so it must be an instinct passed on genetically. The chick encourages the host to keep pace with its high growth rate with its rapid begging call and the chick's open mouth which serves as a sign stimulus.

# Cuckoo Hashing

- **Worst case  $O(1)$  look up ☺**
- **Worse case  $O(1)$  remove ☺**
- **Insertions are usually  $O(1)$ ,** but occasionally need to rehash everything in  $O(n)$  steps => expected  $O(1)$  ☺
- Uses two simple hash functions,  $h_1$  and  $h_2$ , from a universal family and two tables  $T_1$  and  $T_2$  ☺
- **Keys can be stored at  $h_1$  in  $T_1$ , or  $h_2$  in  $T_2$ , but no where else! :-?**
- If those cells are already filled, kick out one of the existing keys and shift that key to its alternate location :-!
- Repeat until every key has been correctly shifted or until no solution is found – if no solution is found, rehash everything with different random hash functions in  $O(n)$  time ☺

Pagh & Rodler, 2001

32
84
59
93
58

$T_1/h_1$

97
26
41
23
53

$T_2/h_2$

# Cuckoo Hashing

- To insert an element  $x$ , start by attempting to insert it into table 1
- If  $h_1(x)$  is empty, place  $x$  there
- Otherwise place  $x$  there, evict the old element  $y$ , and try placing  $y$  into table at  $h_2$
- Repeat this process bouncing between tables until all elements stabilize
- If the locations don't stabilize quick enough ( $O(\lg n)$ ) rehash the table

32
84
59
93
58

$T_1/h_1$

97
26
41
23
53

$T_2/h_2$

# Cuckoo Hashing

- To insert an element  $x$ , start by attempting to insert it into table 1
- If  $h_1(x)$  is empty, place  $x$  there
- Otherwise place  $x$  there, evict the old element  $y$ , and try placing  $y$  into table at  $h_2$
- Repeat this process bouncing between tables until all elements stabilize
- If the locations don't stabilize quick enough ( $O(\lg n)$ ) rehash the table

insert(42)

32
84
59
93
58

$T_1/h_1$

97
26
41
23
53

$T_2/h_2$

# Cuckoo Hashing

- To insert an element  $x$ , start by attempting to insert it into table 1
- If  $h_1(x)$  is empty, place  $x$  there
- Otherwise place  $x$  there, evict the old element  $y$ , and try placing  $y$  into table at  $h_2$
- Repeat this process bouncing between tables until all elements stabilize
- If the locations don't stabilize quick enough ( $O(\lg n)$ ) rehash the table

insert(42)

32
84
59
93
58

$T_1/h_1$

97
26
41
23
53

$T_2/h_2$

# Cuckoo Hashing

- To insert an element  $x$ , start by attempting to insert it into table 1
- If  $h_1(x)$  is empty, place  $x$  there
- Otherwise place  $x$  there, evict the old element  $y$ , and try placing  $y$  into table at  $h_2$
- Repeat this process bouncing between tables until all elements stabilize
- If the locations don't stabilize quick enough ( $O(\lg n)$ ) rehash the table

insert(42)

32
84
42
93
58

$T_1/h_1$

97
26
41
23
53

$T_2/h_2$

# Cuckoo Hashing

- To insert an element  $x$ , start by attempting to insert it into table 1
- If  $h_1(x)$  is empty, place  $x$  there
- Otherwise place  $x$  there, evict the old element  $y$ , and try placing  $y$  into table at  $h_2$
- Repeat this process bouncing between tables until all elements stabilize
- If the locations don't stabilize quick enough ( $O(\lg n)$ ) rehash the table

insert(42)

32
84
42
93
58

$T_1/h_1$

97
26
41
59
23
53

$T_2/h_2$

# Cuckoo Hashing

- To insert an element  $x$ , start by attempting to insert it into table 1
- If  $h_1(x)$  is empty, place  $x$  there
- Otherwise place  $x$  there, evict the old element  $y$ , and try placing  $y$  into table at  $h_2$
- Repeat this process bouncing between tables until all elements stabilize
- If the locations don't stabilize quick enough ( $O(\lg n)$ ) rehash the table

insert(42)

32
84
42
93
58

$T_1/h_1$

59
26
41
97
23
53

$T_2/h_2$

# Cuckoo Hashing

- To insert an element  $x$ , start by attempting to insert it into table 1
- If  $h_1(x)$  is empty, place  $x$  there
- Otherwise place  $x$  there, evict the old element  $y$ , and try placing  $y$  into table at  $h_2$
- Repeat this process bouncing between tables until all elements stabilize
- If the locations don't stabilize quick enough ( $O(\lg n)$ ) rehash the table

insert(42)

32
84
42
93
58

$T_1/h_1$

59
26
41
23
53

$T_2/h_2$

# Cuckoo Hashing

- To insert an element  $x$ , start by attempting to insert it into table 1
- If  $h_1(x)$  is empty, place  $x$  there
- Otherwise place  $x$  there, evict the old element  $y$ , and try placing  $y$  into table at  $h_2$
- Repeat this process bouncing between tables until all elements stabilize
- If the locations don't stabilize quick enough ( $O(\lg n)$ ) rehash the table

insert(42)

32
84
42
93
97

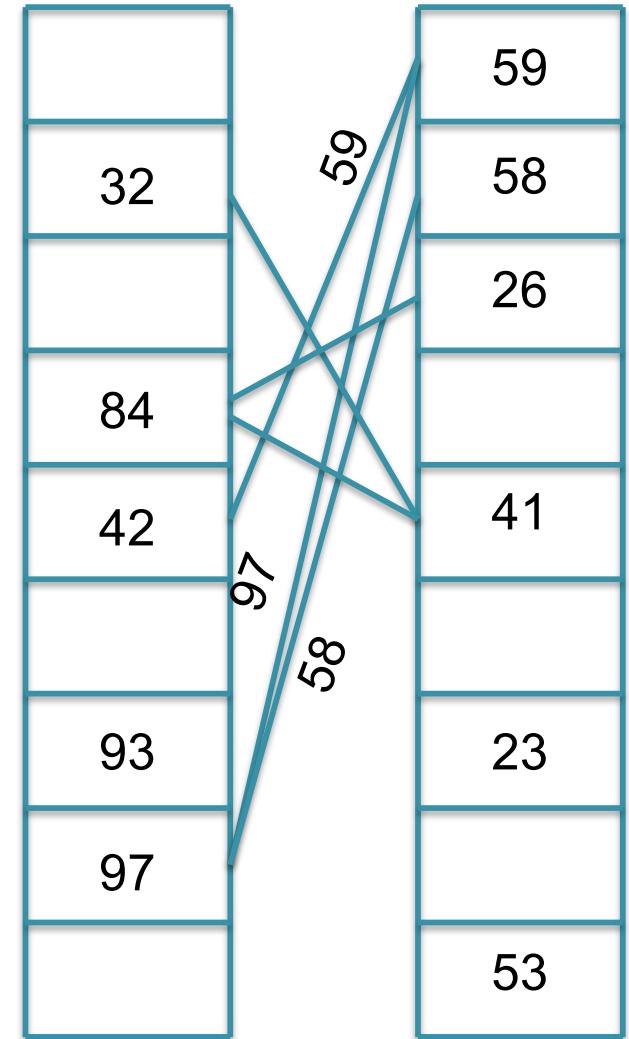
$T_1/h_1$

59
26
41
23
53

$T_2/h_2$

# Cuckoo Graph

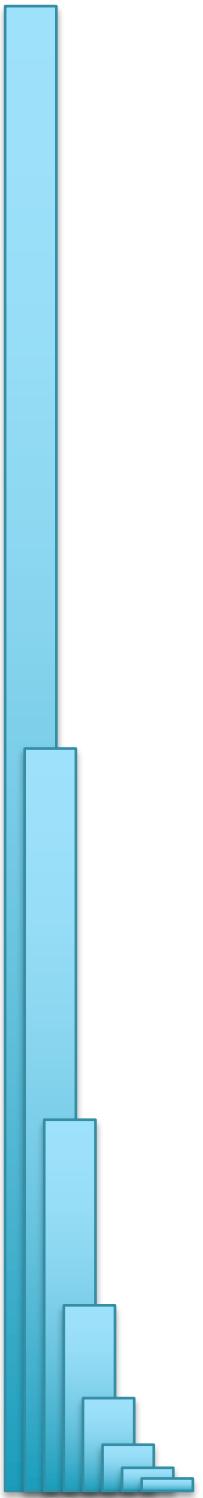
- **Analyzing the performance is complicated because:**
  - Elements can move around and bin in one of two different places
  - The sequence of displacements can jump chaotically over the table
- **Overcome these challenges by analyzing paths of the cuckoo graph**
  - Each table slot is a node
  - Each element is an edge
  - Edges link where each element can be
  - Each insertion introduces a new edge
- **Tricky, but by analyzing the structure and paths of the cuckoo graph, can prove the amortized cost of an insertion is  $O(1 + e)$**



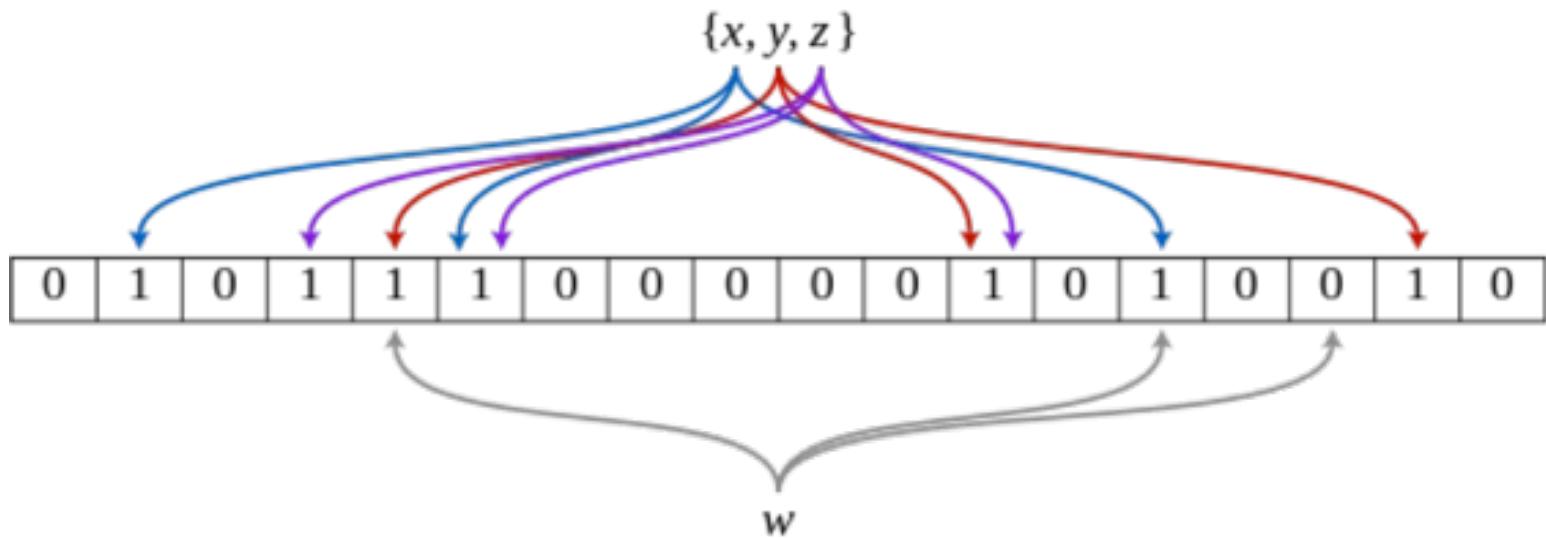
$T_1/h_1$

$T_2/h_2$

# Bloom Filters



# Bloom Filters



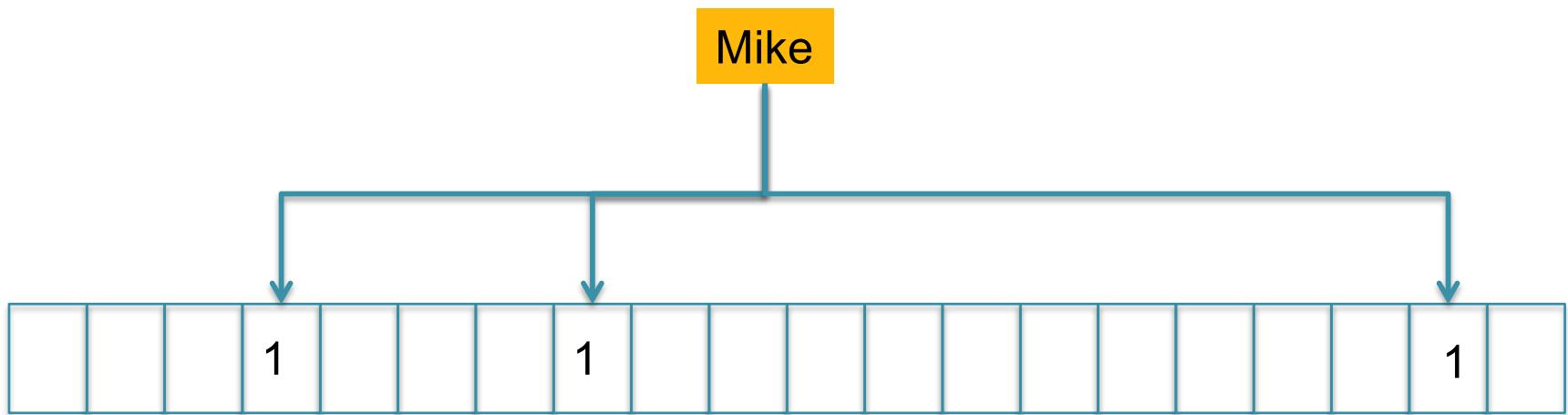
**Extremely space efficient data structure for recording members of a set**

- Uses just a few **\*bits\*** in a bit array to record that a key is or is not in the set, independent of the size or number of elements in the set
- Works by applying a small number ( $k$ ) of hash functions to the key and checking those cells of the bit array
- Initialize bit array to all false; Insert an element set by setting the appropriate bits to true as determined by the  $k$  hash functions:  $O(1)$  time
- If any of the bits are false, the key is definitely not in the set!  $O(1)$  time
- If all of the bits are true, the key is **\*probably\*** in the set:  $O(1)$  time

Bloom, 1970

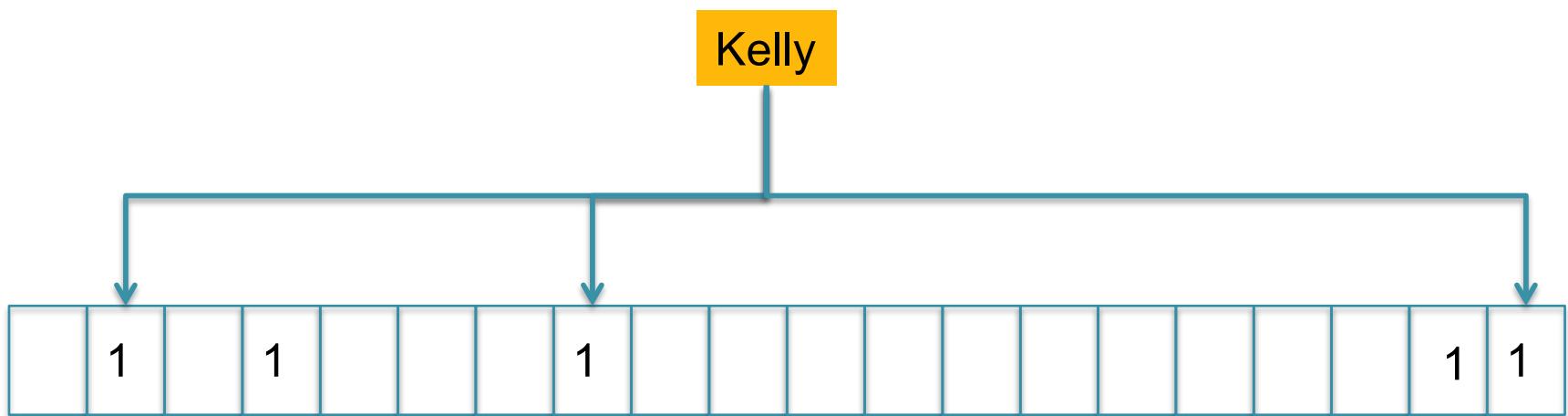
# Bloom Filters

- Insert the following values: Mike, Kelly, James, Katherine, Sydney



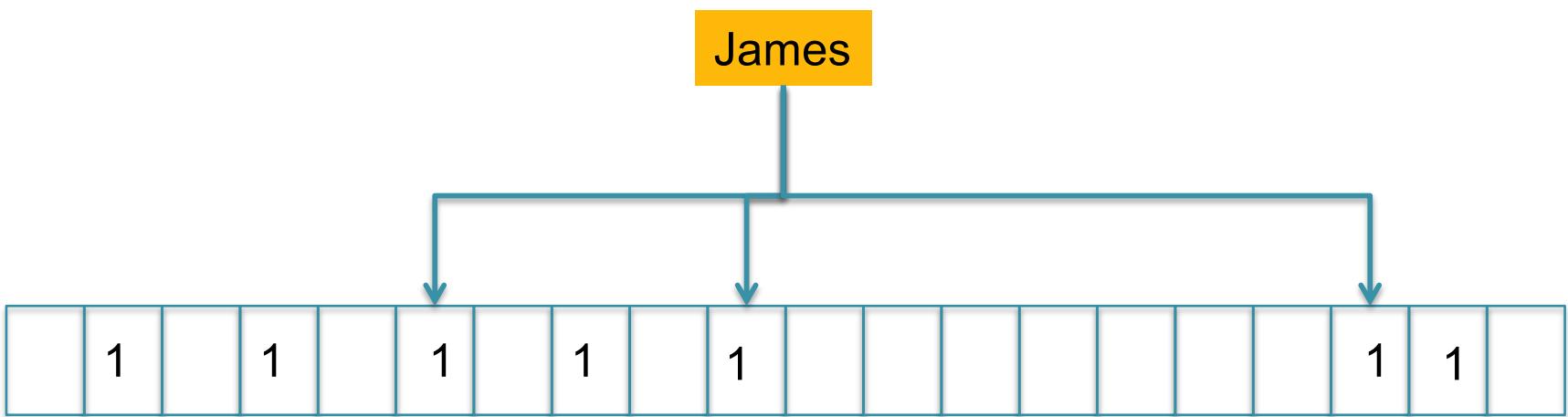
# Bloom Filters

- Insert the following values: Mike, Kelly, James, Katherine, Sydney



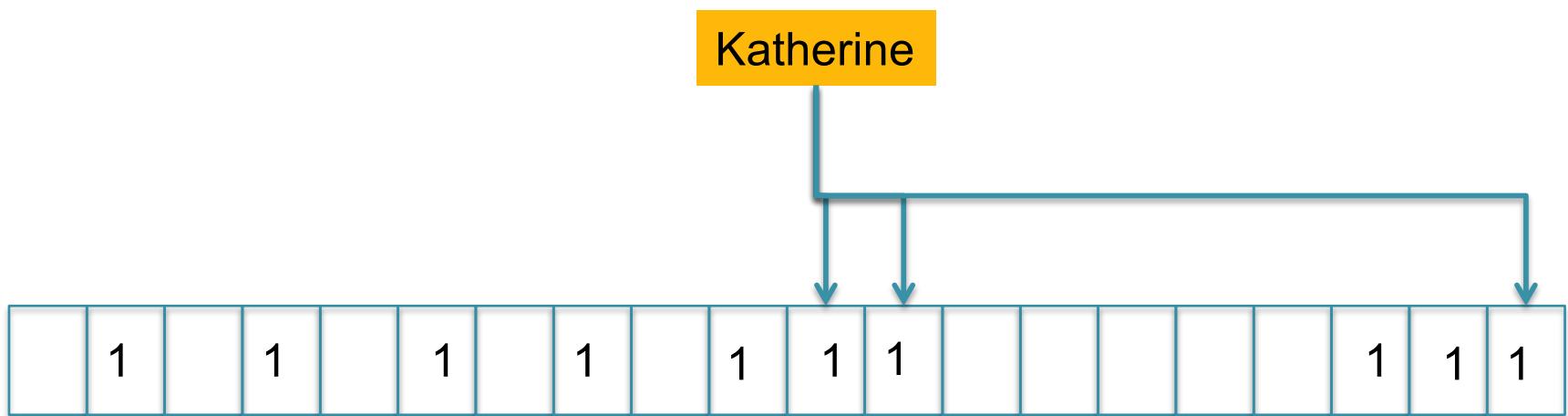
# Bloom Filters

- Insert the following values: Mike, Kelly, James, Katherine, Sydney



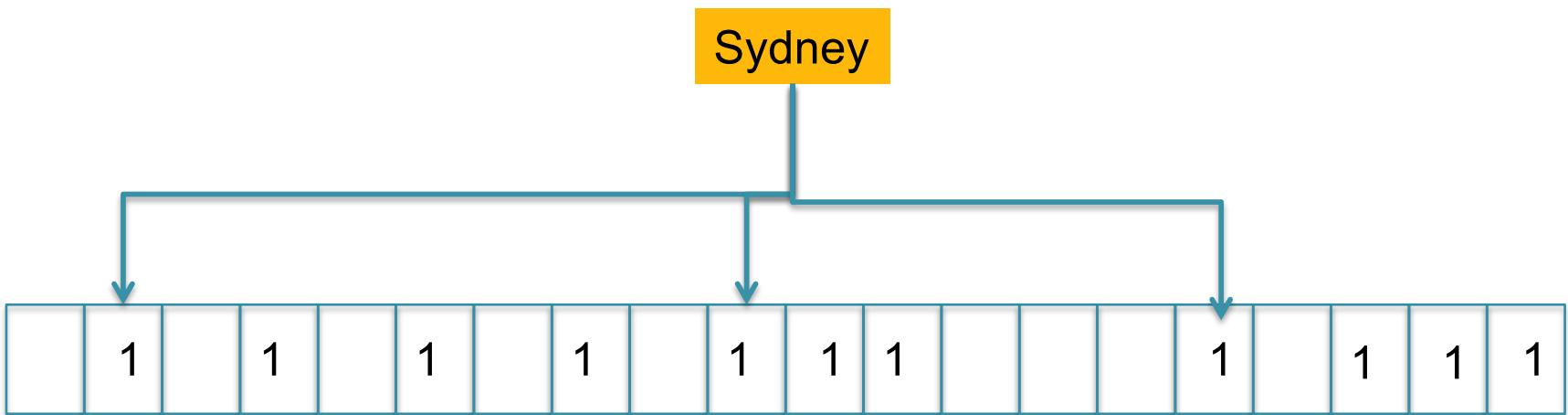
# Bloom Filters

- Insert the following values: Mike, Kelly, James, Katherine, Sydney



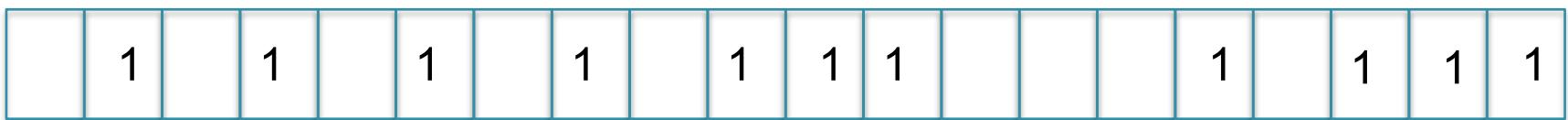
# Bloom Filters

- Insert the following values: Mike, Kelly, James, Katherine, Sydney



# Bloom Filters

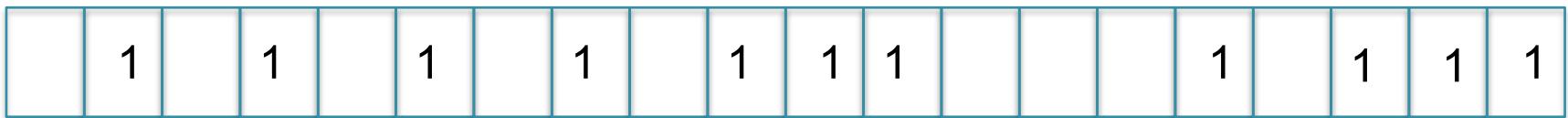
- My entire family is loaded using 20 bits instead of 29 characters!
  - Order of magnitude reduction in space is very common
- As many as  $3 \times 5 = 15$  bits could be true
  - But only 11 are actually set to true



# Bloom Filters

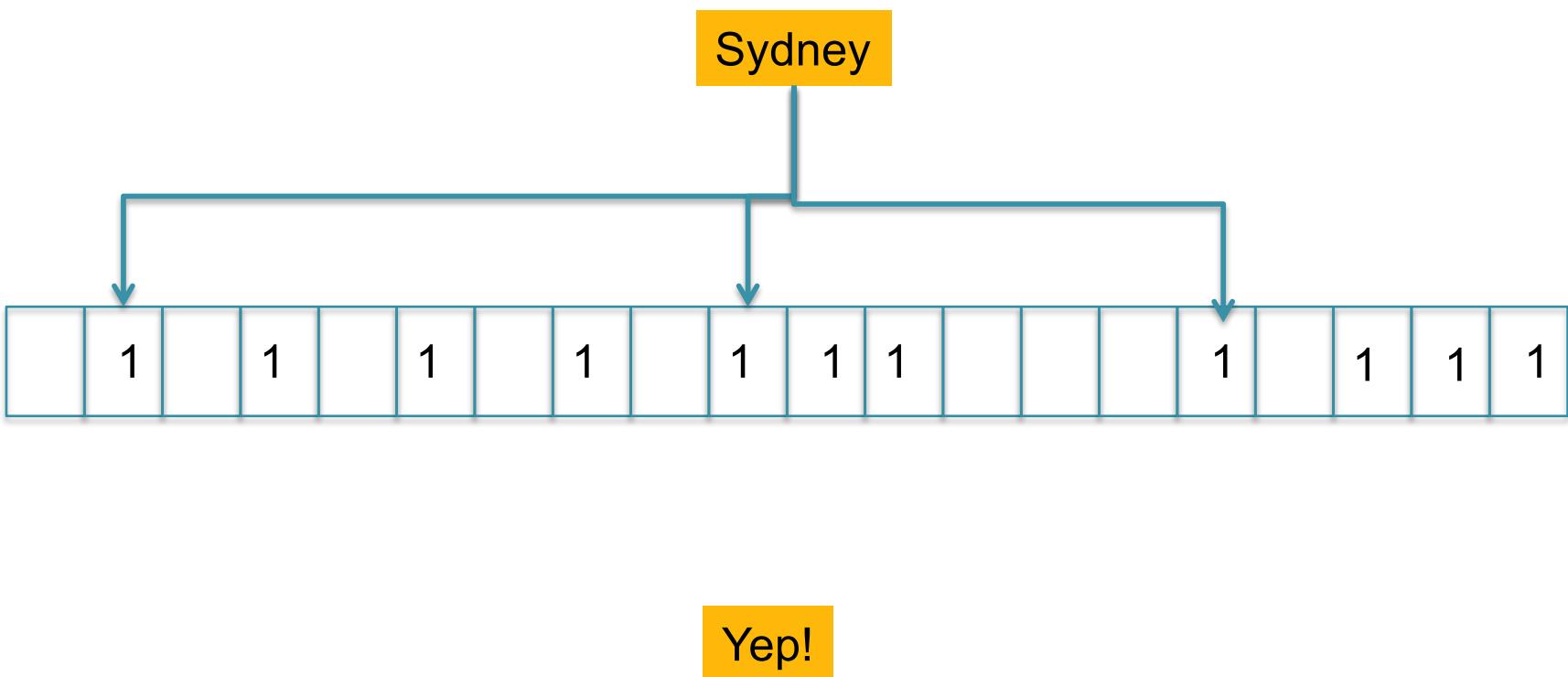
- Is Sydney in the set?

Sydney



# Bloom Filters

- Is Sydney in the set?



# Bloom Filters

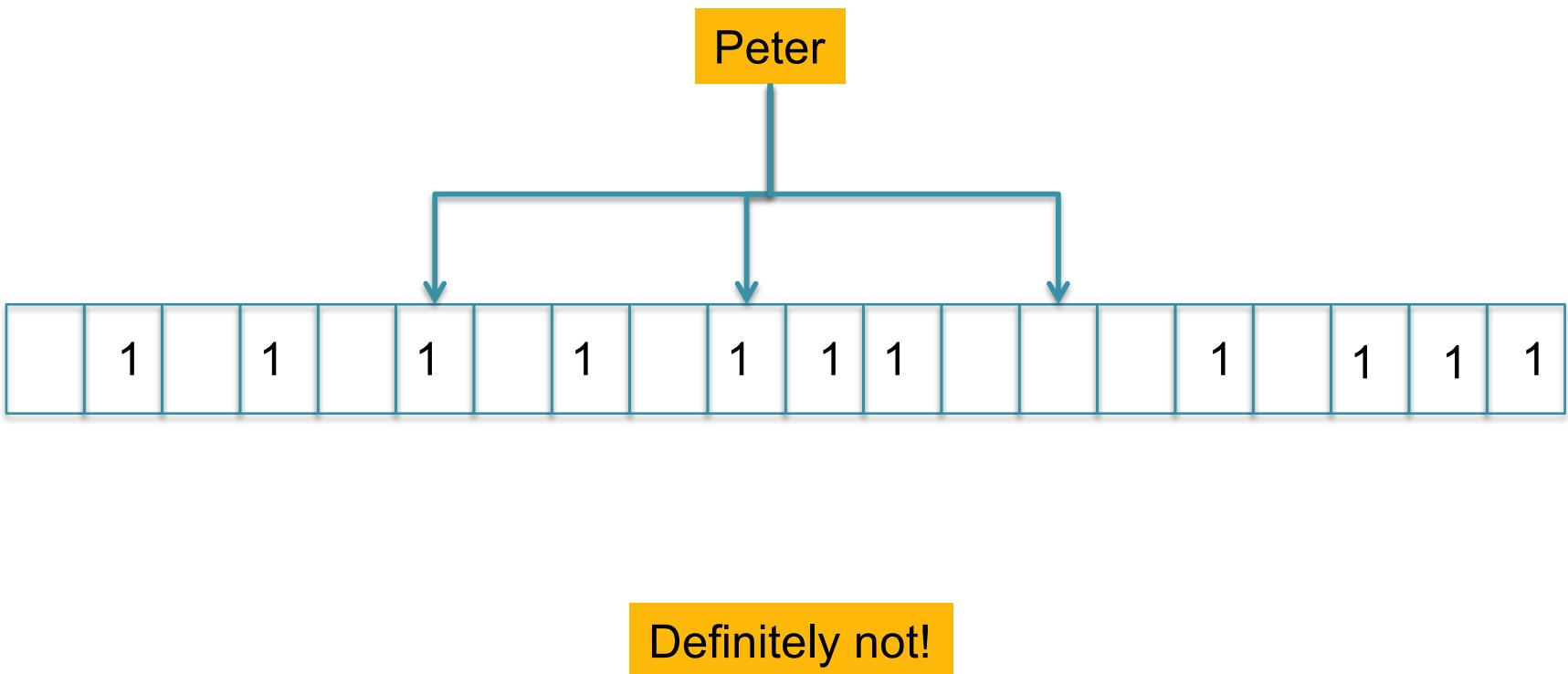
- Is Peter in the set?

Peter

	1	1	1	1	1	1	1	1			1	1	1	1	1
--	---	---	---	---	---	---	---	---	--	--	---	---	---	---	---

# Bloom Filters

- Is Peter in the set?



# Bloom Filters

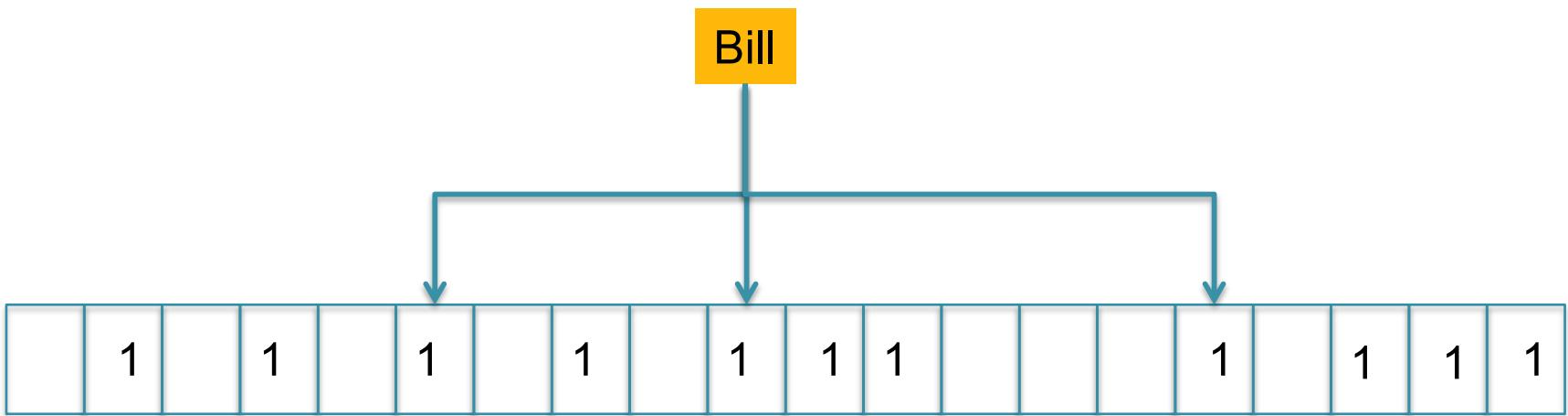
- Is Bill in the set?

Bill



# Bloom Filters

- Is Bill in the set?

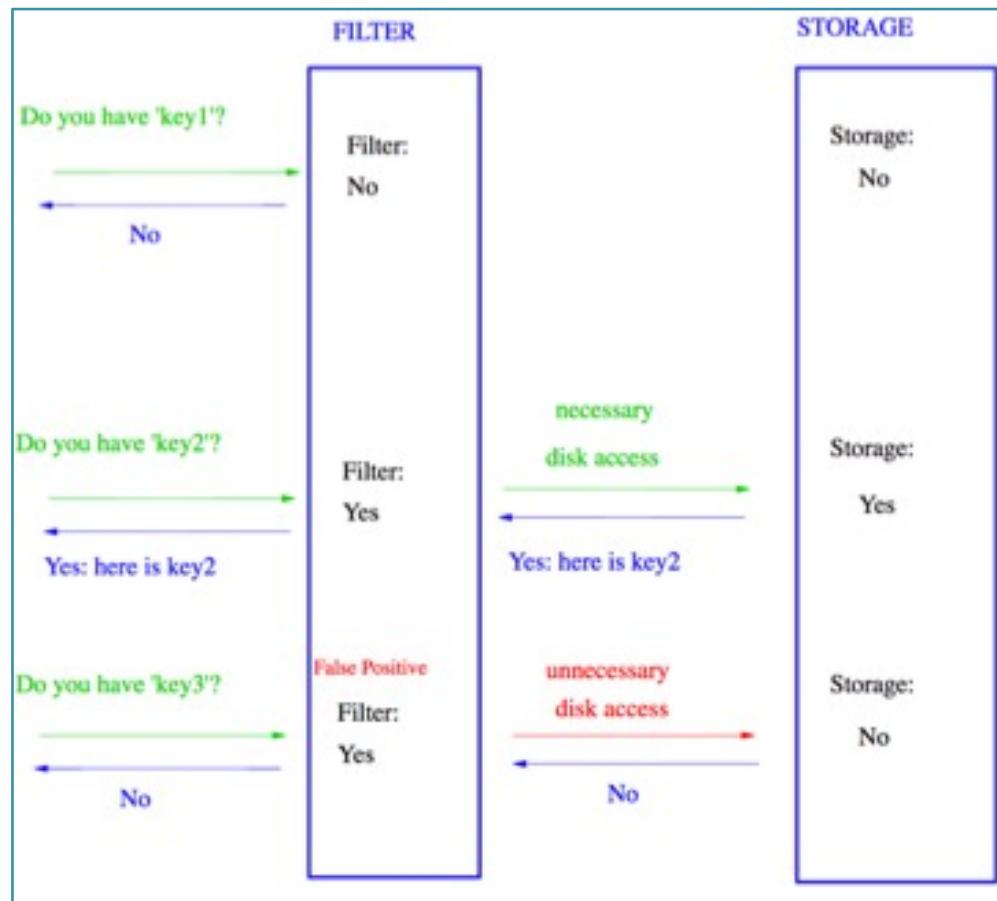


Hmm... I think so, but not totally sure :-/

# Why Bloom Filters?

***When a Bloom filter returns no, we are certain that it is not present, allowing us to skip unnecessary work***

- Disk is ~100,000x slower than RAM
- What it says yes, we might be “tricked” with a false positive
  - Either do a more expensive check, or accept your result may have a (small) amount of error)



# How many false positives?

**Assume a single hash function picks a cell with probability  $1/m$**

- Probability that it doesn't pick the cell:  $1-1/m$
- Probability that all  $k$  hashes don't pick the cell:  $(1-1/m)^k$
- Probability that a certain bit is 0 after  $n$  insertions:  $(1-1/m)^{kn}$

**Probability that a certain bit is 1 after  $n$  insertions:**

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

**Probability that all  $k$  bits will be set by chance: the false positive rate**

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

**Size of table to guarantee  
a desired false positive rate**

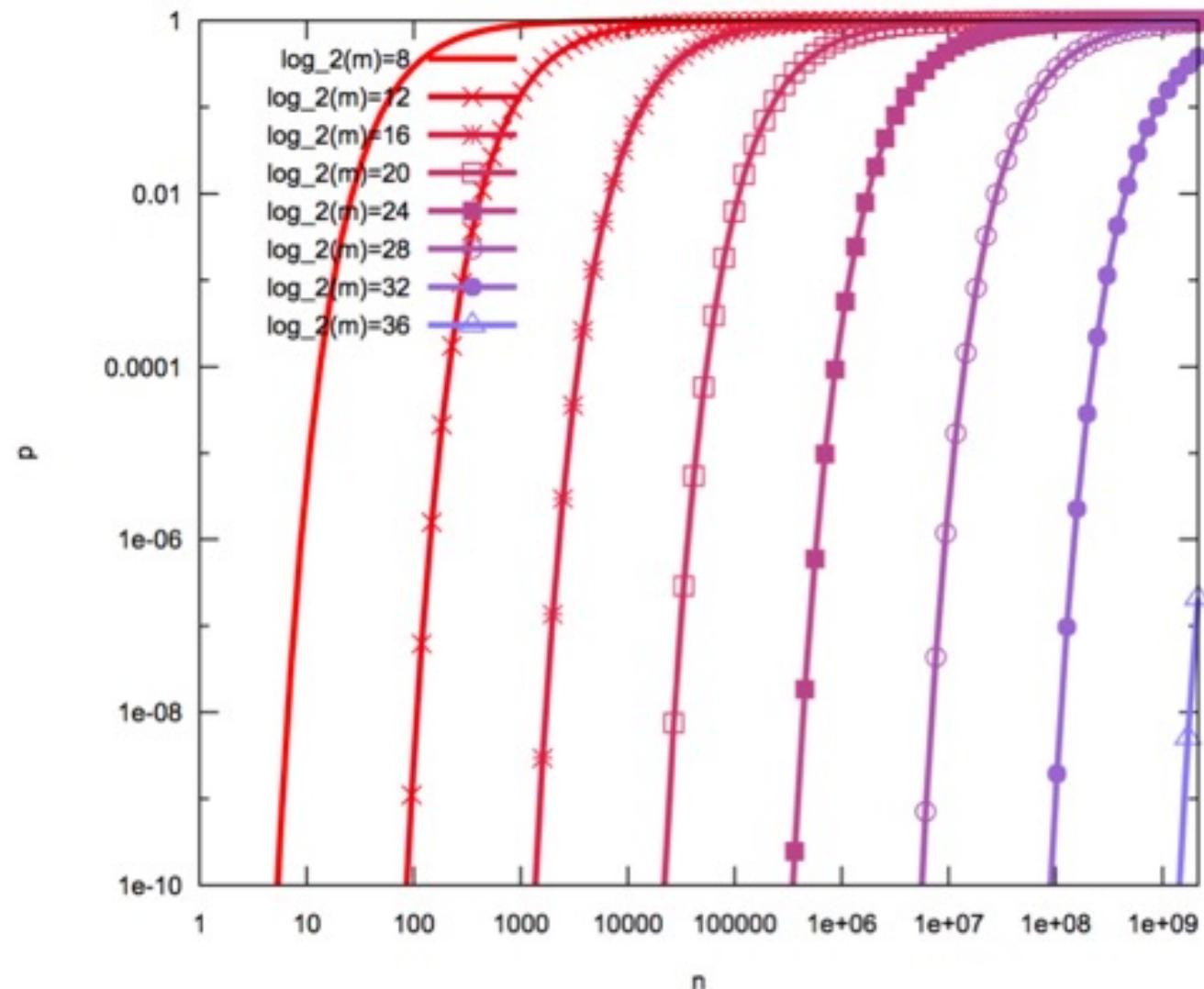
$$m = -\frac{n \ln p}{(\ln 2)^2}$$

**Optimal number  
of hash functions**

$$k = \frac{m}{n} \ln 2$$

Only 9.6 bits/element and 7 hash functions for a 1% false positive rate!  
Only 14.4 bits and 10 hash functions for a 0.1% false positive rate!

# How many false positives?



The false positive probability  $p$  can be made arbitrarily small for a certain number of elements  $n$  using a filter of size  $m$ , assuming an optimal number of hash functions  $k=(m/n) \ln 2$ .

# Bloom Filters Union

***Schatz Family***

	1	1	1	1	1	1	1	1	1			1	1	1	1	1
--	---	---	---	---	---	---	---	---	---	--	--	---	---	---	---	---

+ ***Frohlich Family***

1	1			1				1		1		1		1		1
---	---	--	--	---	--	--	--	---	--	---	--	---	--	---	--	---

---

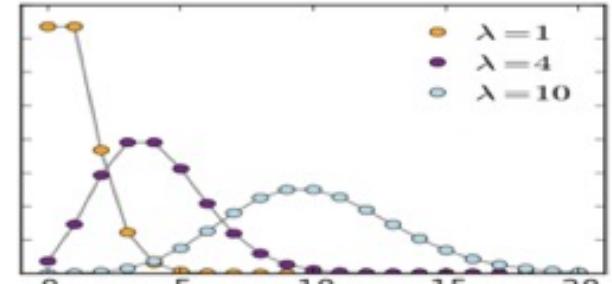
***Schatz+Frohlich Family***

1	1		1	1	1		1	1	1	1	1		1		1	1
---	---	--	---	---	---	--	---	---	---	---	---	--	---	--	---	---

# Advanced Hashing Summary

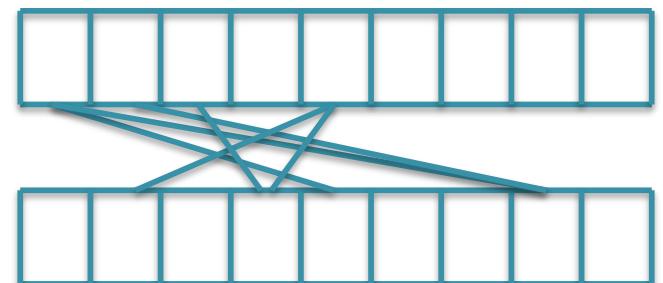
## ***Collisions***

- Be careful of collisions!
- Perfect hashing guarantees we avoid collisions, universal hashing lets us pick a new hash function from a family as needed



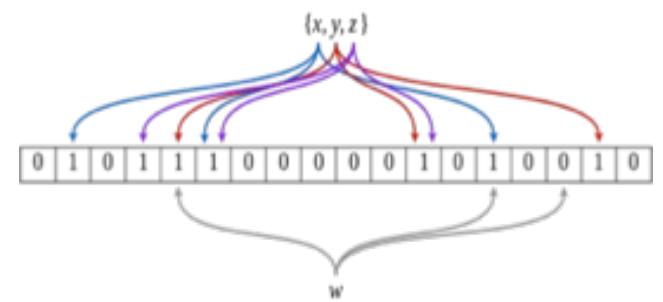
## ***Cuckoo Hashing***

- Simple open addressing technique with  $O(1)$  lookup with expected  $O(1)$  insert
- Often outperforms other collision management schemes



## ***Bloom Filters***

- Store a huge set in a tiny amount of space allowing for a small rate of false positives.
- Used as a quick pre-filter to determine if the slow operation needs to take place



# Next Steps

- I. Reflect on the magic and power of Hash Tables!
- I. Assignment 8 due Friday November 16 @ 10pm