# CSE 250
## Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

# Lec 38: Lossy Sets and Bloom Filters

# Announcements

- WA5 is out now, due Sunday @ 11:59PM
- Fill out your course evaluations!
- Comments about the TAs? Fill out a TA evaluation as well!

# Recall the `Set` ADT

Set has **`add`**, **`contains`** and **`remove`**…

*What if we didn't need contains to be perfect?*

*What would that entail? Why would that be useful? What would we gain?*

# Motivation

Let's say a hot new restaurant just opened up, but it's an hour away. You want to eat there, but don't want to make the drive only to find out they have no tables…*what do you do?*

# Motivation

Let's say a hot new restaurant just opened up, but it's an hour away. You want to eat there, but don't want to make the drive only to find out they have no tables…*what do you do?*

- **Call the restaurant**
  - **If they say they have no tables, don't go. You've saved yourself the trip**
  - **If they say they do…drive there. By the time you get there, they might have run out of tables, but you would have made the drive anyways**

# Motivation

**Another Example:** Reading from disk is expensive

Even when using a **B+ Tree**, if you ask for an element that doesn't exist you will need to do $\log_c(n)$ disk reads

**Idea:** Keep an in-memory summary of the data

- If the summary says the key is in a particular layer, access the layer
- If not, skip the layer and end the search early

*What guarantees do we need for this to work?*

# Motivation

- If our summary incorrectly says the key exists (false positive) ✓
  - We will read the layer only to find out it's not there
  - Extra work, but doesn't break anything…we would have done that work anyways

# Motivation

- If our summary incorrectly says the key exists (false positive) ✓
  - We will read the layer only to find out it's not there
  - Extra work, but doesn't break anything…we would have done that work anyways
- If our summary incorrectly says the key doesn't exist (false negative) ✗
  - We will stop the algorithm and return an incorrect answer

# Motivation

- If our summary incorrectly says the key exists (false positive) ✓
  - We will read the layer only to find out it's not there
  - Extra work, but doesn't break anything…we would have done that work anyways
- If our summary incorrectly says the key doesn't exist (false negative) ✗
  - We will stop the algorithm and return an incorrect answer

**False positives are OK (not ideal though), false negatives are not**

# Lossy Sets

`LossySet<T>`

`void add(T t)`

- Insert `t` into the set (kind of)

`boolean contains(T t)`

- If `t` is in the set **ALWAYS** return true
- If `t` is not in the set **USUALLY** return false (returning true is OK)

# Lossy Set

*What does this gain for us?*

**Idea:** If contains doesn't always need to be right, we don't need to store everything!

# A Trivial Example

```
1 class TrivialLossySet<T> extends LossySet<A> {
2   void add(T t) { /* do nothing */ }
3   boolean contains(T t) { return true; }
4 }
```

*Does this work?*

# A Trivial Example

```
1 class TrivialLossySet<T> extends LossySet<A> {
2   void add(T t) { /* do nothing */ }
3   boolean contains(T t) { return true; }
4 }
```

*Does this work?* **Yes! …but not really that useful**

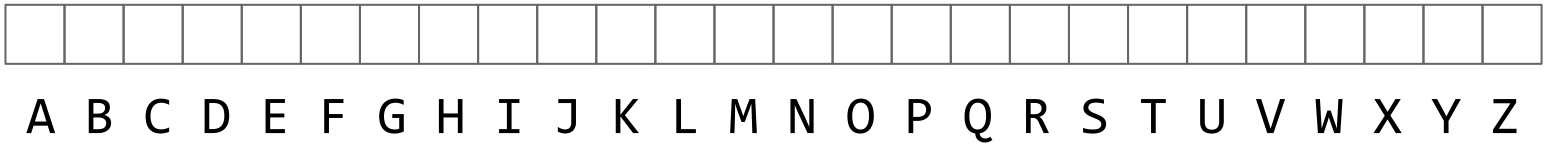
# An Improvement

**Idea:** Take inspiration from HashTables

- Bucketize the keys in some way
- Keep **one bit** per bucket
- **`add(T t):`** Set **`a`**'s bucket to 1
- **`contains(T t):`** Return true if the bit for **`t`**'s bucket is set

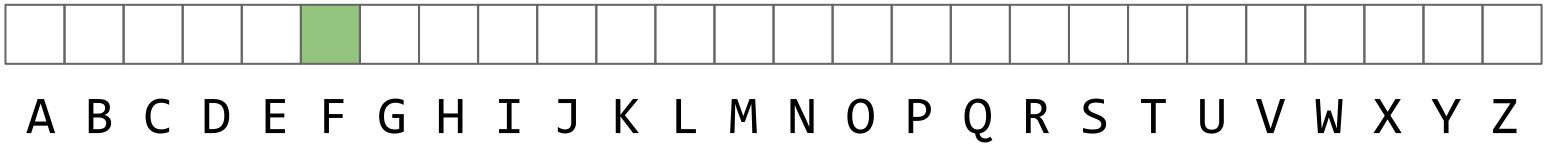
# Setting Bins

```
add("Frankenstein")
```



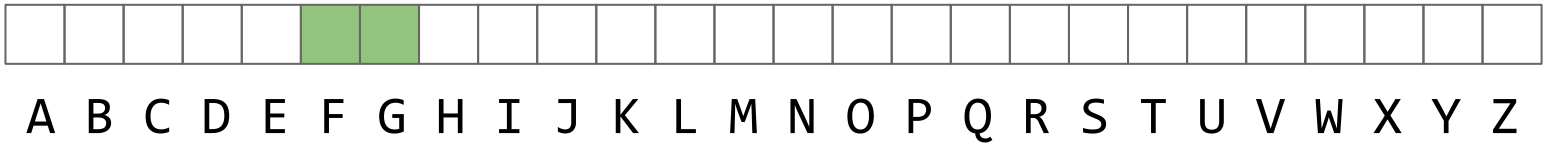
# Setting Bins

```
add("Frankenstein")
```



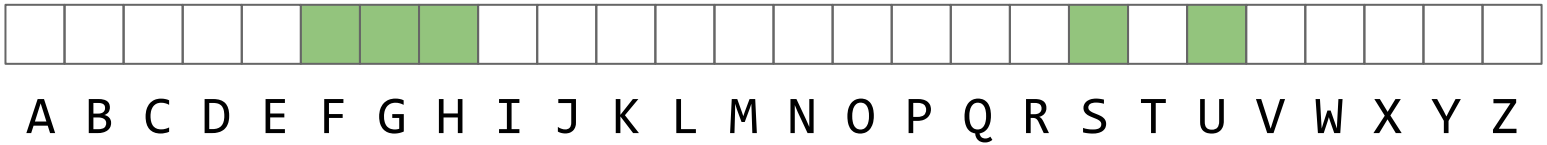
# Setting Bins

```
add("Frankenstein")
add("Get Out")
```



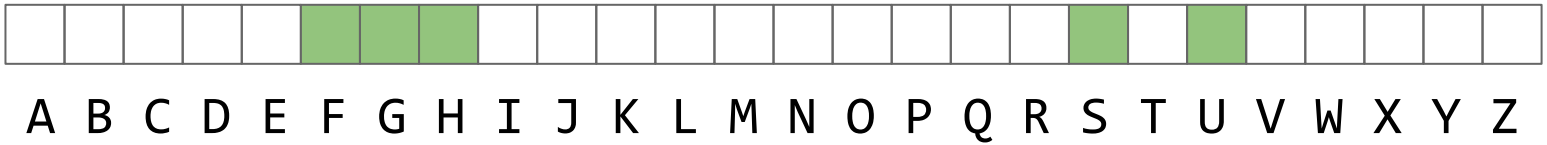
# Setting Bins

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
```



# Setting Bins

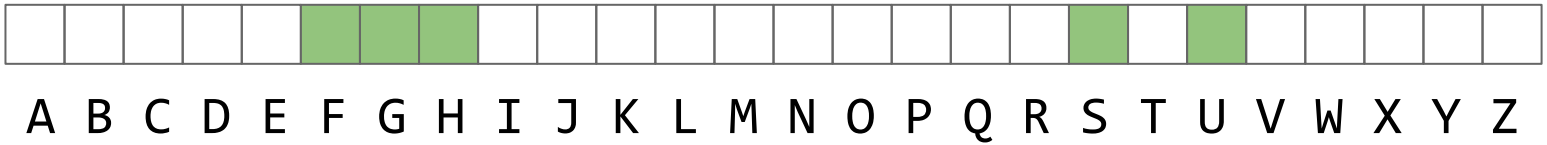
```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```



# Calling Contains

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```

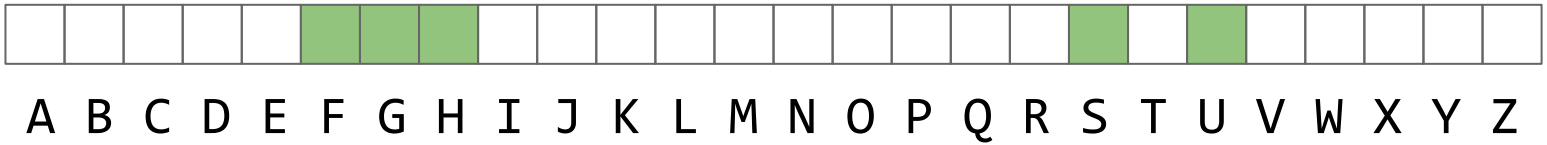
```
contains("Scream")?
```



# Calling Contains

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```

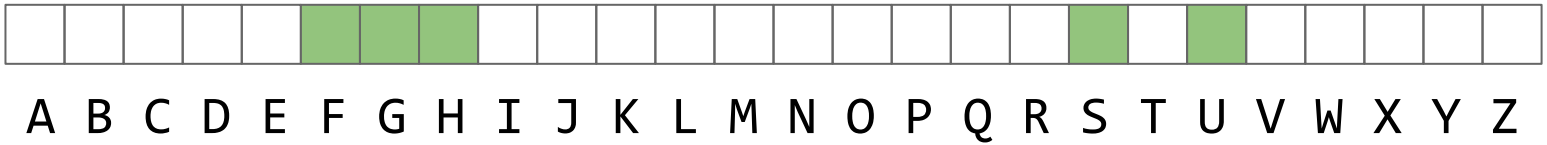
```
contains("Scream")? T
```



# Calling Contains

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```

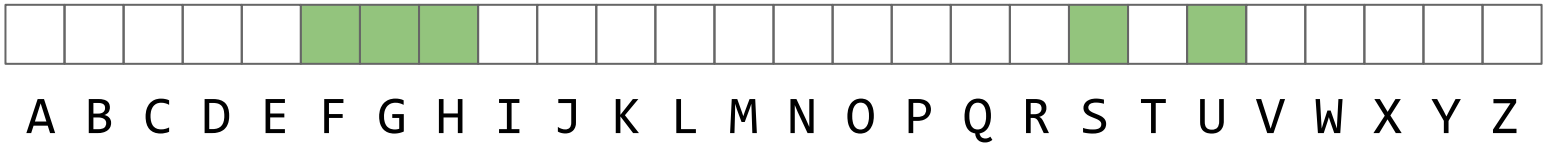
```
contains("Scream")? T
contains("Saw")?
```



# Calling Contains

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```

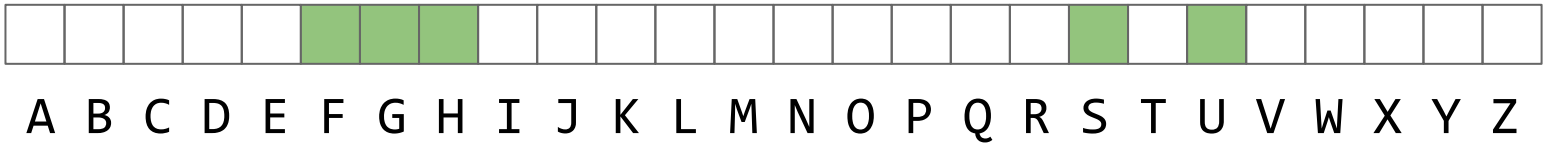
```
contains("Scream")? T
contains("Saw")? T
```



# Calling Contains

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```

```
contains("Scream")? T
contains("Saw")? T
contains("The Candyman")?
```



# Calling Contains

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```

```
contains("Scream")? T
contains("Saw")? T
contains("The Candyman")? F
```

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

# Calling Contains

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```

```
contains("Scream")? T
contains("Saw")? T
contains("The Candyman")? F
contains("Dracula")? F
contains("Friday the 13th")? T
```

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

# Calling Contains

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```

```
contains("Scream")? T
contains("Saw")? T
contains("The Candyman")? F
contains("Dracula")? F
contains("Friday the 13th")? T
```

**So this works…how can we improve its accuracy?**

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

# Motivating Example

## Raise your hand to answer the following questions
## Take note of who else raises their hand at the same time

# Which Pokemon do you prefer?

1. Squirtle
2. Pikachu
3. Charizard
4. What the heck is a Pokemon?

# Which movie genre do you prefer?

1. Comedy
2. Action
3. Horror
4. Drama

# Which sport would you rather play?

1. Football (American)
2. Soccer
3. Hockey
4. Please don't make me touch grass…

# Lossy Sets

**Observation:** We have a lot of overlap on just one feature (question). Less overlap if we consider two at the same time. Even less if we consider only those who match on all three.

How many buckets would our HashTable need to store info for all three features?

# Lossy Sets

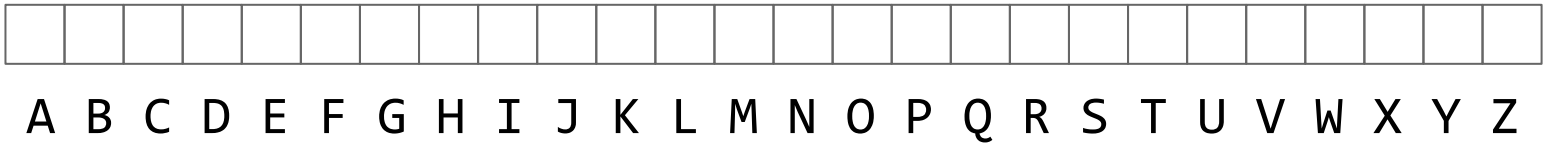
**Observation:** We have a lot of overlap on just one feature (question). Less overlap if we consider two at the same time. Even less if we consider only those who match on all three.

**Idea:** Use the same set of buckets for all features

- ie store movies by first AND last letter in the title

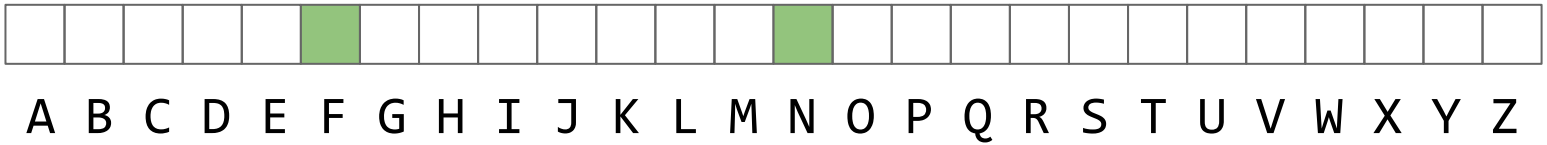
# Setting Bins

```
add("Frankenstein")
```



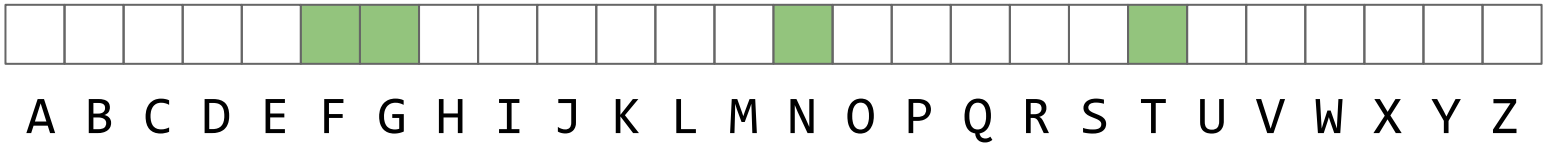
# Setting Bins

```
add("Frankenstein")
```



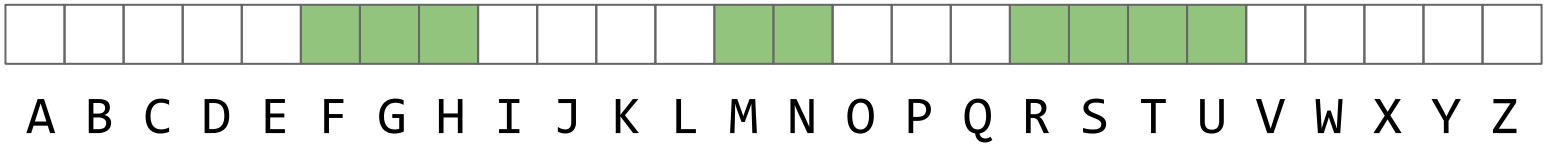
# Setting Bins

```
add("Frankenstein")
add("Get Out")
```



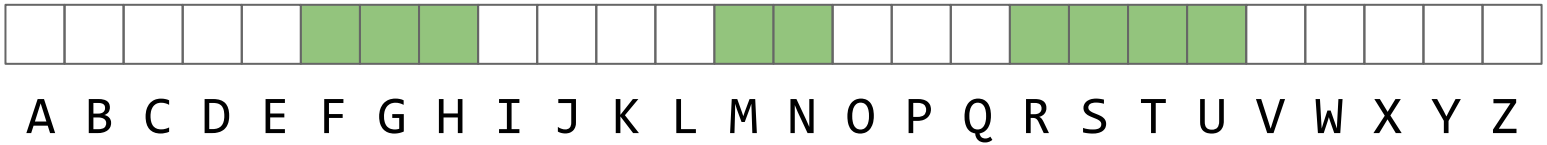
# Setting Bins

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
```



# Setting Bins

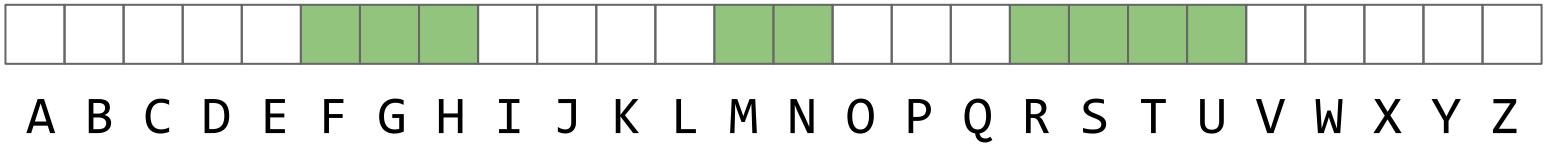
```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```



# Calling Contains

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```

```
contains("Scream")?
```



# Calling Contains

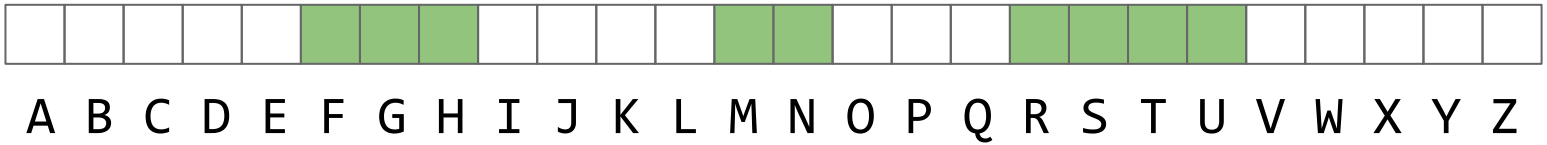
```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```

```
contains("Scream")? T
```

# Calling Contains

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```

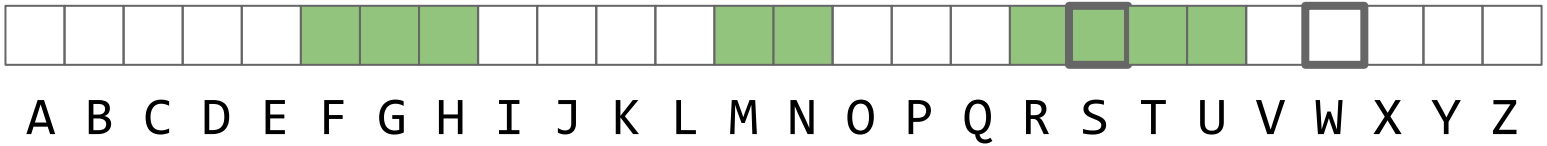
```
contains("Scream")? T
contains("Saw")?
```



# Calling Contains

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```

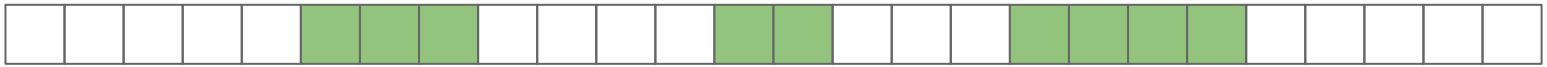
```
contains("Scream")? T
contains("Saw")? F
```



# Calling Contains

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```

```
contains("Scream")? T
contains("Saw")? F
contains("Dracula")? F
contains("Friday the 13th")? T
```

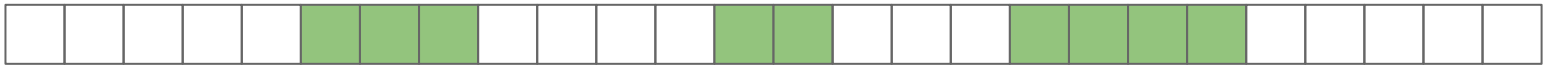


A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

# Calling Contains

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```

```
contains("Scream")? T
contains("Saw")? F
contains("Dracula")? F
contains("Friday the 13th")? T
contains("The Candyman")?
```

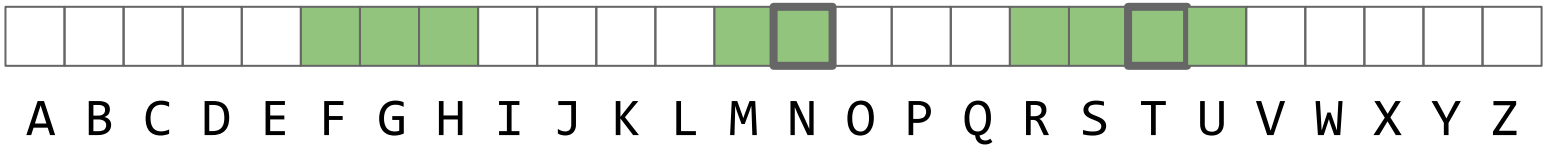


A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

# Calling Contains

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```

```
contains("Scream")? T
contains("Saw")? F
contains("Dracula")? F
contains("Friday the 13th")? T
contains("The Candyman")? T
```



# False Positives

**Notice we still got a false positive (but for a different reason)**

- We can never avoid false positives, just want to make them less likely

# False Positives

**Notice we still got a false positive (but for a different reason)**

- We can never avoid false positives, just want to make them less likely
- If we use more hash functions, it becomes less likely that movie A and B have the exact same set of buckets

# False Positives

**Notice we still got a false positive (but for a different reason)**

- We can never avoid false positives, just want to make them less likely
- If we use more hash functions, it becomes less likely that movie A and B have the exact same set of buckets
- **BUT** if we use more hash functions, we fill up more buckets which make it more likely that movie A's buckets are filled in by other movies

**We will therefore need to find a good balance (more on that later)**

# Hash Functions

**Observation:** Our current example is not very uniform…movie titles are much more likely to start with some letters and not others

**Idea:** Use hash functions!

```
1  class LossyHashSet1<T> extends LossySet<T> {
2    private int size;
3    private boolean[] bits;
4    public LossyHashSet1(int s) {
5      size = s; bits = new boolean[size];
6    }
7    public void add(T t) {
8      int bucket = t.hashCode() % size;
9      bits[bucket] = true;
10   }
11   public boolean contains(T t) {
12     int bucket = t.hashCode() % size;
13     return bits[bucket];
14   }
15 }
```

# Lossy Hash Set w/1 Hash Function

# Lossy Hash Set

Assume we `add(a)` then `contains(b)`

What does `contains(b)` return, and when?

- **True:** When `hash(a) == hash(b) % size`
- **False:** When `hash(a) != hash(b) % size`

What is the probability of each, with ***N*** buckets?

# Lossy Hash Set

Assume we **`add(a)`** then **`contains(b)`**

What does **`contains(b)`** return, and when?

- **True:** When **`hash(a) == hash(b) % size`**
- **False:** When **`hash(a) != hash(b) % size`**

What is the probability of each, with ***N*** buckets?

- **True:** 1/N ← only wrong 1/N of the time
- **False:** (N-1)/N

```
1  class LossyHashSet2<T> extends LossySet<T> {
2    /* ... */
3    public int hash1(T t) { /* ??? */ }
4    public int hash2(T t) { /* ??? */ }
5    public void add(T t) {
6      bits[hash1(t) % size] = true;
7      bits[hash2(t) % size] = true;
8    }
9    public boolean contains(T t) {
10     return bits[hash1(t) % size] && bits[hash2(t) % size];
11   }
12 }
```

# Lossy Hash Set w/2 Hash Functions

# Lossy (Double) Hash Set

Assume we **`add(a)`** then **`contains(b)`**

What does **`contains(b)`** return, and when?

- **True:** When $\texttt{hash}_1\texttt{(a)==hash}_1\texttt{(b)}$ AND $\texttt{hash}_2\texttt{(a)==hash}_2\texttt{(b)}$ *`(% size)`*
- **False:** Otherwise

What is the probability of each, with ***N*** buckets?

# Lossy (Double) Hash Set

Assume we **`add(a)`** then **`contains(b)`**

What does **`contains(b)`** return, and when?

- **True:** When $\texttt{hash}_1\texttt{(a)==hash}_1\texttt{(b)}$ AND $\texttt{hash}_2\texttt{(a)==hash}_2\texttt{(b)}$ *`(% size)`*
- **False:** Otherwise

What is the probability of each, with ***N*** buckets?

- **True:** $\sim(1/N)^2$ ← only wrong $1/N^2$ of the time!
- **False:** $!((N-1)/N)^2$

# Multiple Hash Functions

So how do we make multiple hash functions?

```
1  static int SEED1 = 123104912035;
2  static int SEED2 = 406923456234; // Avoid sequentially adjacent seeds
3
4  public int hash1(T t) {
5    return (SEED1 ^ t.hashCode()).hashCode(); // Use ^ instead of +
6  }
7  public int hash2(T t) {
8    return (SEED1 6 t.hashCode()).hashCode();
9  }
```

# Multiple Hash Functions

So how do we make multiple hash functions?

**We can make as many as we want!**

```
1 static int SEEDS = [123104912035, 406923456234, 304173420014, ...]
2 public int hashK(int k, T t) {
3   return (SEEDS[k] ^ t.hashCode()).hashCode(); // Use ^ instead of +
4 }
```

```
1  class LossyHashSetK<T> extends LossySet<T> {
2    /* ... */
3    public void add(T t) {
4      for (int k = 0; k < K; k++) {
5        bits[hashK(k, t) % size] = true;
6      }
7    }
8    public boolean contains(T t) {
9      for (int k = 0; k < K; k++) {
10       if (bits[hashK(k, t) % size] == false) return false;
11     }
12     return true;
13   }
14 }
```

# Lossy Hash Set w/k Hash Functions

```
1  class BloomFilter<T> extends LossySet<T> {
2    /* ... */
3    public void add(T t) {
4      for (int k = 0; k < K; k++) {
5        bits[hashK(k, t) % size] = true;
6      }
7    }
8    public boolean contains(T t) {
9      for (int k = 0; k < K; k++) {
10       if (bits[hashK(k, t) % size] == false) return false;
11     }
12     return true;
13   }
14 }
```

This is called a Bloom Filter!

# Lossy Hash Set w/k Hash Functions → A Bloom Filter!

# Bloom Filter

**BloomFilters have two parameters:**

- SIZE ← Number of buckets in the bit array
  - **Intuitively:** More space means fewer collisions
- HASHES ← Number of hash functions
  - **Intuitively:** More hash functions means…
    - A higher chance that one of **b**'s bits is unset (lower false+ chance)
    - More bits get set (higher false+ chance)

# Bloom Filter

**BloomFilters have two parameters:**

- SIZE ← Number of buckets in the bit array
  - **Intuitively:** More space means fewer collisions
- HASHES ← Number of hash functions
  - **Intuitively:** More hash functions means…
    - A higher chance that one of **b**'s bits is unset (lower false+ chance)
    - More bits get set (higher false+ chance)

**Increasing SIZE trades space for a lower false+ rate**
**For HASHES there's a sweet spot in the middle**

# Bloom Filters: Collision Probability

The probability that 1 bit is **set** by 1 hash function:

$$\frac{1}{\text{SIZE}}$$

# Bloom Filters: Collision Probability

The probability that 1 bit is **not set** by 1 hash function:

$$1 - \frac{1}{\text{SIZE}}$$

# Bloom Filters: Collision Probability

The probability that 1 bit is **not set** by HASHES hash functions:

$$\left(1 - \frac{1}{\text{SIZE}}\right)^{\text{HASHES}}$$

# Bloom Filters: Collision Probability

The probability that 1 bit is **not set** by HASHES hash functions

…over ***n*** distinct calls to **add**:

$$\left(1 - \frac{1}{\texttt{SIZE}}\right)^{\texttt{HASHES} \cdot n}$$

# Bloom Filters: Collision Probability

The probability that 1 bit is **set by at least one** of HASHES hash functions

…over ***n*** distinct calls to **add**:

$$1 - \left(1 - \frac{1}{\texttt{SIZE}}\right)^{\texttt{HASHES} \cdot n}$$

# Bloom Filters: Collision Probability

The probability that **all** HASHES randomly selected bits of element **b**

…are **set by at least one** of the HASHES hash functions

… over ***n*** distinct calls to **add**:

$$\left(1 - \left(1 - \frac{1}{\texttt{SIZE}}\right)^{\texttt{HASHES} \cdot n}\right)^{\texttt{HASHES}}$$

# Bloom Filters: Collision Probability

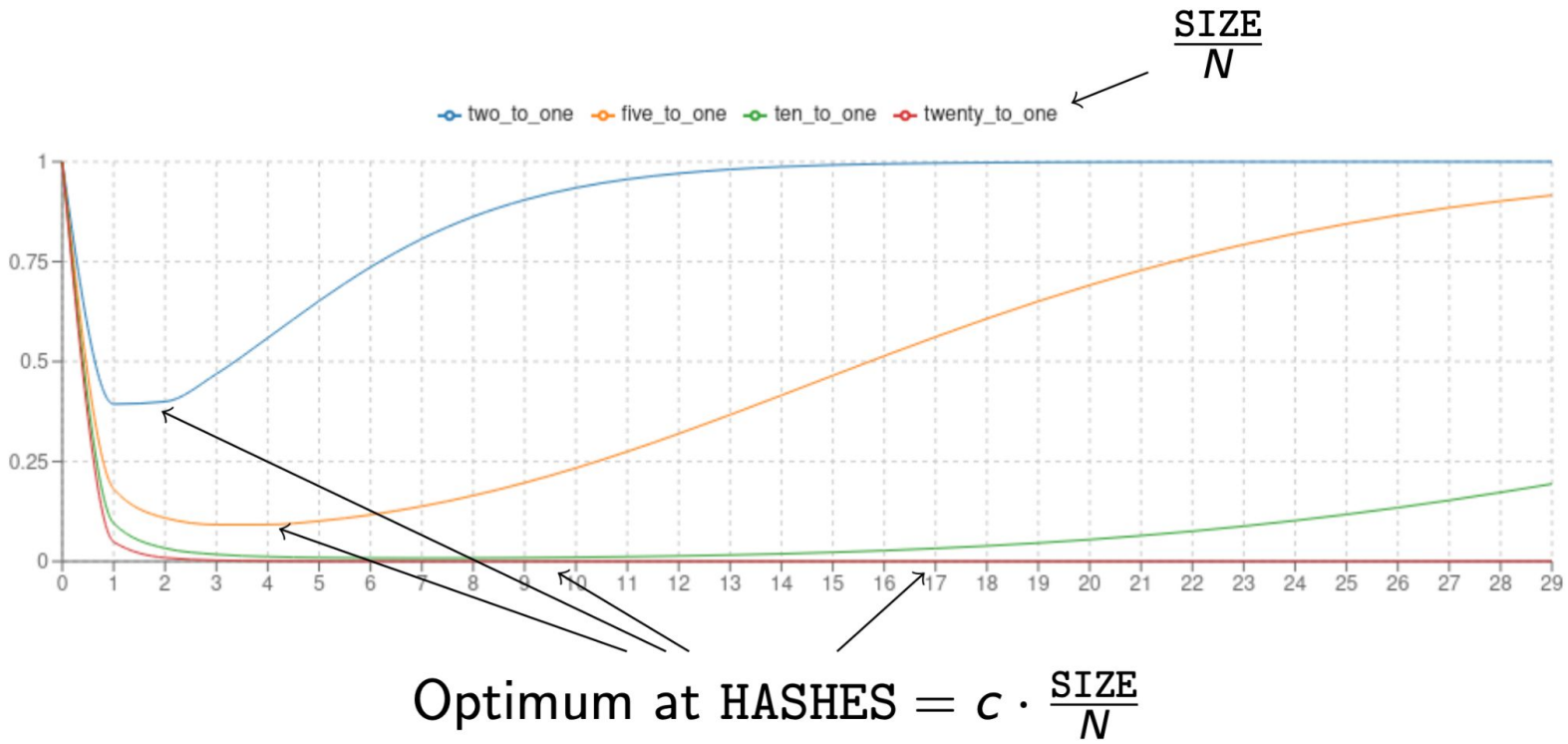
The probability that **all** HASHES randomly selected bits of element **b**

…are **set by at least one** of the HASHES hash functions

… over ***n*** distinct calls to **add**:

$$\approx \left(1 - e^{-\frac{\texttt{HASHES} \cdot n}{\texttt{SIZE}}}\right)^{\texttt{HASHES}}$$

# Bloom Filter: Collision Probability



# Bloom Filter: Analysis

$$\texttt{HASHES} = c \cdot \frac{\texttt{SIZE}}{n}$$

$$n = c \cdot \frac{\texttt{SIZE}}{\texttt{HASHES}}$$

***n*** and SIZE are linearly related (SIZE = ***O*(*n*)**), but…

# Bloom Filters: In Practice

$$\frac{\texttt{SIZE}}{n} = 5 \implies \sim 10\% \text{ collision chance}$$

$$\frac{\texttt{SIZE}}{n} = 10 \implies \sim 1\% \text{ collision chance}$$

10 **bits** per element vs:

- 32 bits for one integer (3 to 1 savings)
- 64 bits for one double/long (6 to 1 savings)
- 512 bits for a 64 byte record (50 to 1 savings)