# CSE373: Data Structures & Algorithms
# Lecture 13: Hash Collisions

Lauren Milne

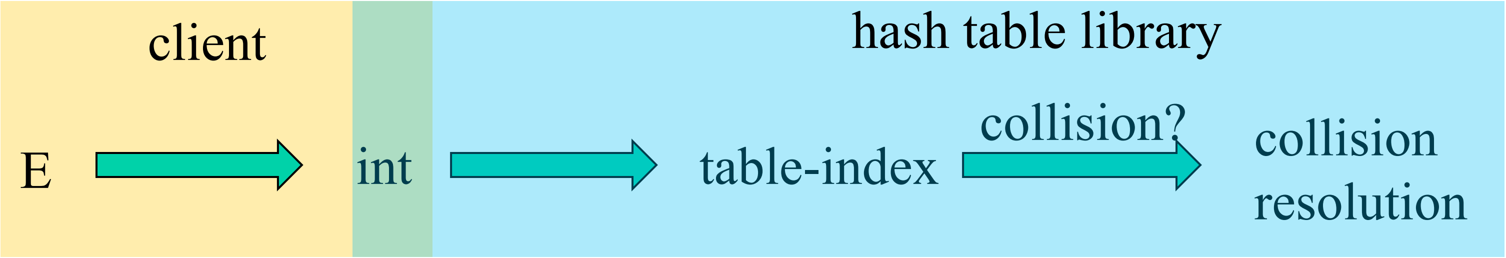
Summer 2015

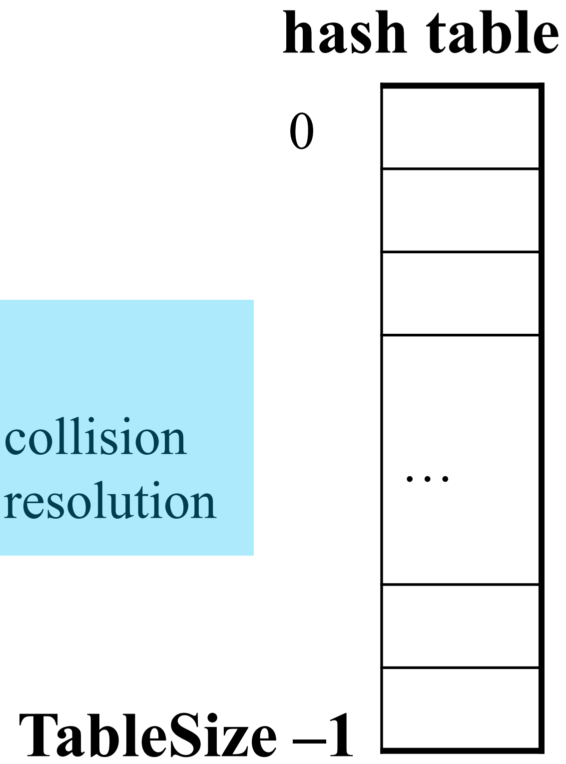
# *Announcements*

- Homework 4 is out
  - find a partner using the discussion board if you would like
  - fill out catalyst survey if you have a partner

# Hash Tables: Review

- Aim for constant-time (i.e., *O*(1)) **`find`**, **`insert`**, and **`delete`**
  - "On average" under some reasonable assumptions
- A hash table is an array of some fixed size
  - But grow-able as we'll see



# *Collision resolution*

Collision:

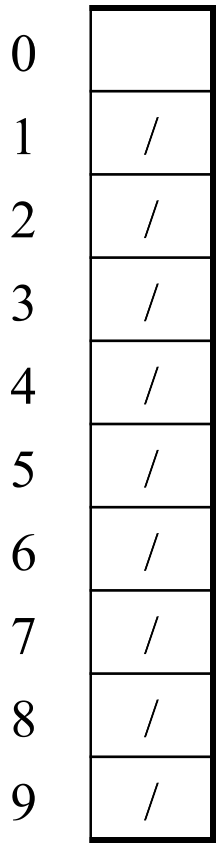
When two keys map to the same location in the hash table

We try to avoid it, but number-of-keys exceeds table size

So hash tables should support collision resolution

– Ideas?

# *Separate Chaining*




Chaining:

All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

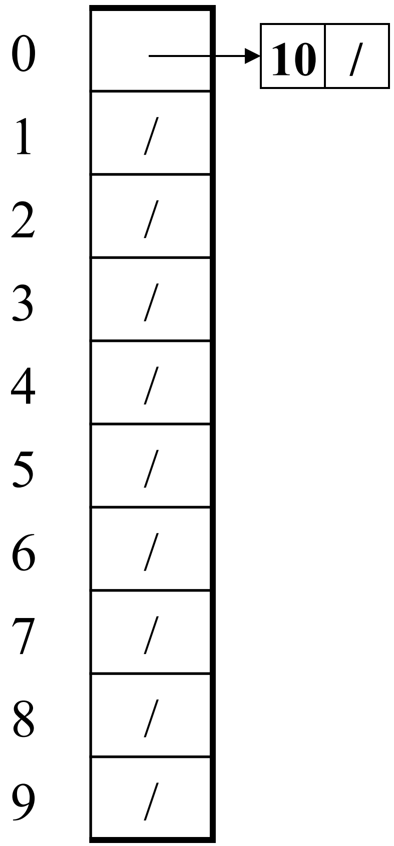
Example:

insert 10, 22, 107, 12, 42

with mod hashing

and **`TableSize`** = 10

# *Separate Chaining*




Chaining:

All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

As easy as it sounds

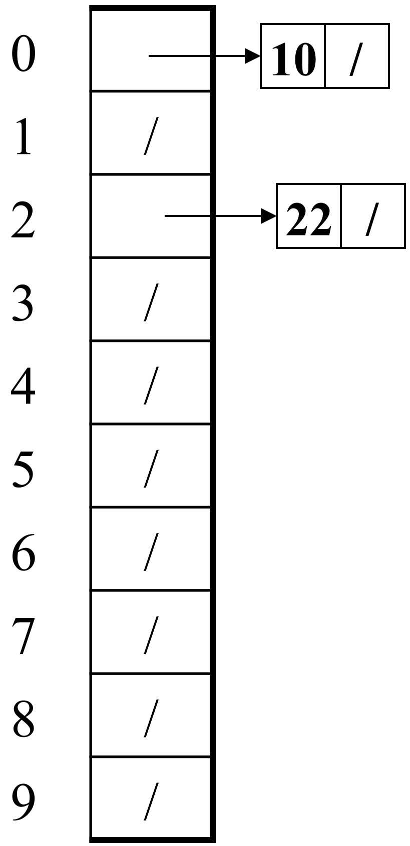
Example:

insert 10, 22, 107, 12, 42

with mod hashing

and **`TableSize`** = 10

# *Separate Chaining*




Chaining:

All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

As easy as it sounds

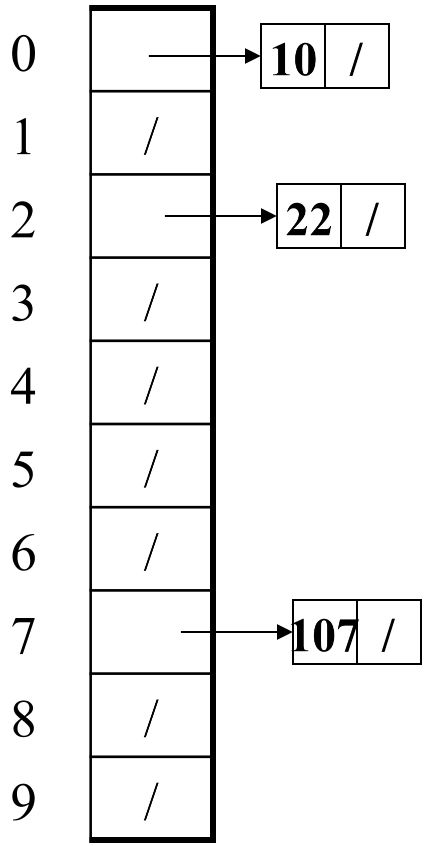
Example:

insert 10, 22, 107, 12, 42

with mod hashing

and **`TableSize`** = 10

# *Separate Chaining*




Chaining:

All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

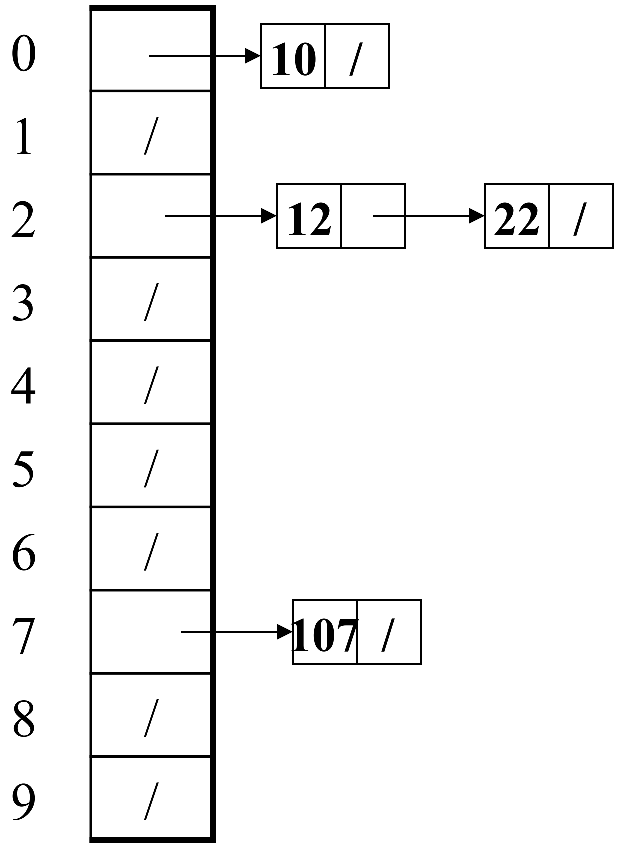
Example:

insert 10, 22, 107, 12, 42

with mod hashing

and **`TableSize`** = 10

# *Separate Chaining*




Chaining:

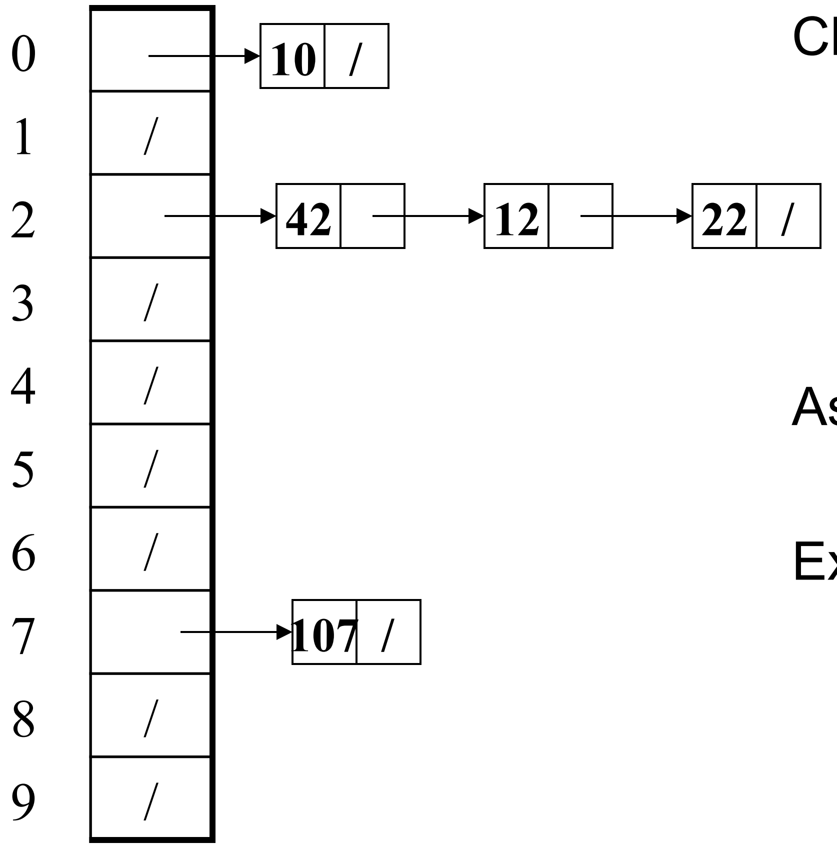
All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

As easy as it sounds

Example:

insert 10, 22, 107, 12, 42
with mod hashing
and **`TableSize`** = 10

# *Separate Chaining*




Chaining:

All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

As easy as it sounds

Example:

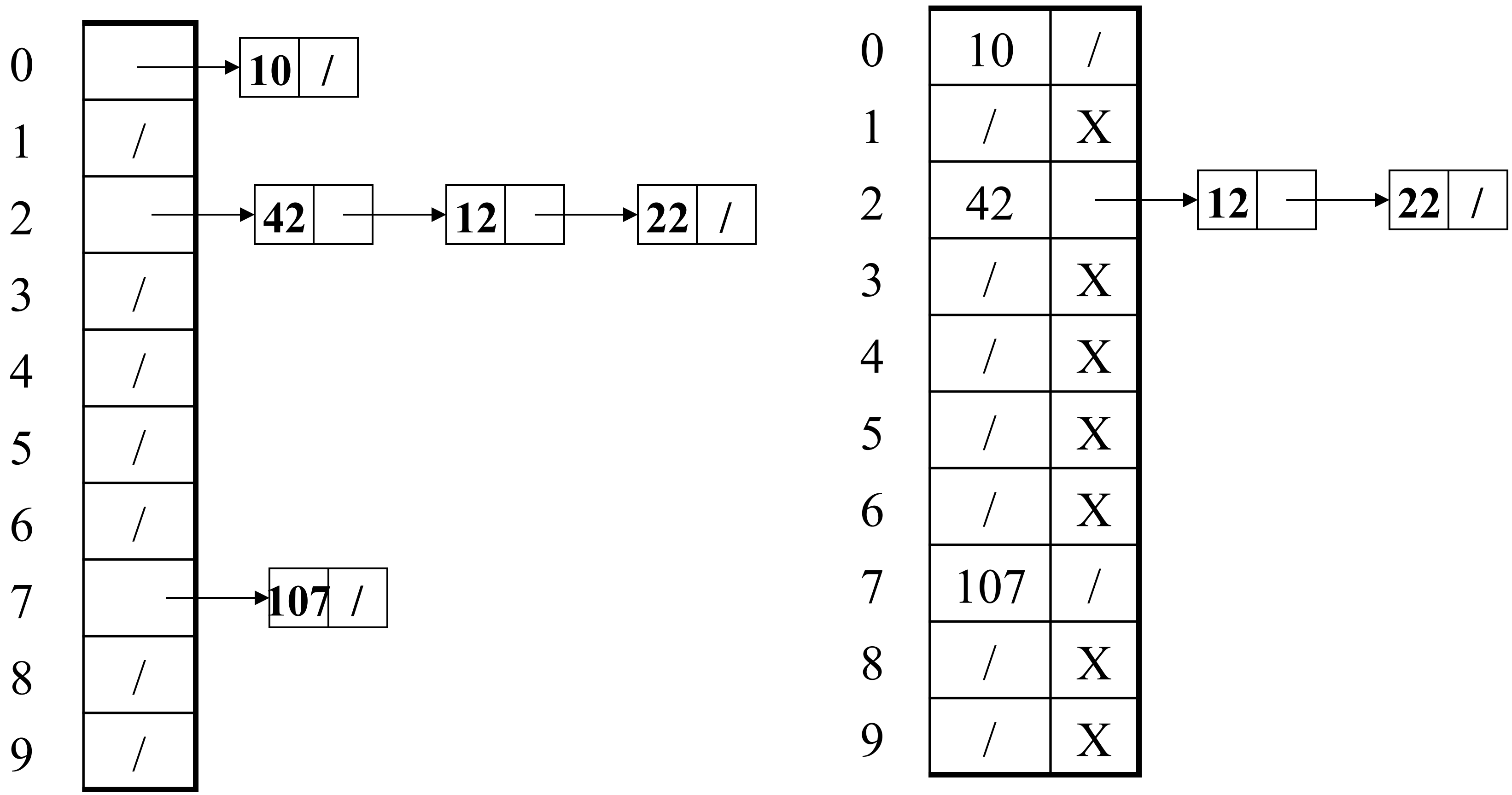
insert 10, 22, 107, 12, 42 with mod hashing and **`TableSize`** = 10

# *Thoughts on chaining*

- Worst-case time for **`find`**?
  - Linear
  - But only with really bad luck or bad hash function
  - So not worth avoiding (e.g., with balanced trees at each bucket)

- Beyond asymptotic complexity, some “data-structure engineering” may be warranted
  - Linked list vs. array
  - Maybe leave room for 1 element in the table itself, to optimize constant factors for the common case
    - A time-space trade-off…

# *Time vs. space (constant factors only here)*



# *More rigorous chaining analysis*

Definition: The load factor, $\lambda$, of a hash table is

$$\lambda = \frac{\text{N}}{\text{TableSize}} \qquad \leftarrow \textbf{number of elements}$$

Under chaining, the average number of elements per bucket is ___

# *More rigorous chaining analysis*

Definition: The load factor, $\lambda$, of a hash table is

$$\lambda = \frac{N}{\text{TableSize}} \quad \leftarrow \textbf{number of elements}$$

Under chaining, the average number of elements per bucket is $\lambda$

So if some inserts are followed by *random* finds, then on average:

- Each unsuccessful **`find`** compares against ____ items

# *More rigorous chaining analysis*

Definition: The load factor, $\lambda$, of a hash table is

$$\lambda = \frac{\text{N}}{\text{TableSize}} \qquad \leftarrow \textbf{number of elements}$$

Under chaining, the average number of elements per bucket is $\lambda$

So if some inserts are followed by *random* finds, then on average:

- Each unsuccessful **`find`** compares against $\lambda$ items
- Each successful **`find`** compares against ______ items

# *More rigorous chaining analysis*

Definition: The load factor, $\lambda$, of a hash table is

$$\lambda = \frac{\text{N}}{\text{TableSize}} \qquad \leftarrow \textbf{number of elements}$$

Under chaining, the average number of elements per bucket is $\lambda$

So if some inserts are followed by *random* finds, then on average:

- Each unsuccessful **`find`** compares against $\lambda$ items
- Each successful **`find`** compares against $\lambda / 2$ items

So we like to keep $\lambda$ fairly low (e.g., 1 or 1.5 or 2) for chaining

# *Alternative: Use empty space in the table*

- Another simple idea: If **`h(key)`** is already full,
  - try **`(h(key) + 1) % TableSize`**. If full,
  - try **`(h(key) + 2) % TableSize`**. If full,
  - try **`(h(key) + 3) % TableSize`**. If full…

- Example: insert 38, 19, 8, 109, 10

| | |
|---|---|
| 0 | / |
| 1 | / |
| 2 | / |
| 3 | / |
| 4 | / |
| 5 | / |
| 6 | / |
| 7 | / |
| 8 | 38 |
| 9 | / |

# *Alternative: Use empty space in the table*

- Another simple idea: If **`h(key)`** is already full,
  - try **`(h(key) + 1) % TableSize`**. If full,
  - try **`(h(key) + 2) % TableSize`**. If full,
  - try **`(h(key) + 3) % TableSize`**. If full…

- Example: insert 38, 19, 8, 109, 10

| | |
|---|---|
| 0 | / |
| 1 | / |
| 2 | / |
| 3 | / |
| 4 | / |
| 5 | / |
| 6 | / |
| 7 | / |
| 8 | 38 |
| 9 | 19 |

# *Alternative: Use empty space in the table*

- Another simple idea: If **`h(key)`** is already full,
  - try **`(h(key) + 1) % TableSize`**. If full,
  - try **`(h(key) + 2) % TableSize`**. If full,
  - try **`(h(key) + 3) % TableSize`**. If full…

- Example: insert 38, 19, 8, 109, 10

| | |
|---|---|
| 0 | 8 |
| 1 | / |
| 2 | / |
| 3 | / |
| 4 | / |
| 5 | / |
| 6 | / |
| 7 | / |
| 8 | 38 |
| 9 | 19 |

# *Alternative: Use empty space in the table*

- Another simple idea: If **`h(key)`** is already full,
  - try **`(h(key) + 1) % TableSize`**. If full,
  - try **`(h(key) + 2) % TableSize`**. If full,
  - try **`(h(key) + 3) % TableSize`**. If full…

- Example: insert 38, 19, 8, 109, 10

| | |
|---|---|
| 0 | 8 |
| 1 | 109 |
| 2 | / |
| 3 | / |
| 4 | / |
| 5 | / |
| 6 | / |
| 7 | / |
| 8 | 38 |
| 9 | 19 |

# *Alternative: Use empty space in the table*

- Another simple idea: If **`h(key)`** is already full,
  - try **`(h(key) + 1) % TableSize`**. If full,
  - try **`(h(key) + 2) % TableSize`**. If full,
  - try **`(h(key) + 3) % TableSize`**. If full…

- Example: insert 38, 19, 8, 109, 10

| | |
|---|---|
| 0 | 8 |
| 1 | 109 |
| 2 | 10 |
| 3 | / |
| 4 | / |
| 5 | / |
| 6 | / |
| 7 | / |
| 8 | 38 |
| 9 | 19 |

# *Probing hash tables*

Trying the next spot is called probing (also called open addressing)

- We just did linear probing
  - **`i`**$^{th}$ probe was **`(h(key) + i) % TableSize`**
- In general have some probe function **`f`** and use **`(h(key) + f(i)) % TableSize`**

Open addressing does poorly with high load factor $\lambda$

- Too many probes means no more $O(1)$
- How can we fix this (how can we decrease the load factor)?

# *Other operations with open addressing*

`insert` finds an open table position using a probe function

What about `find`?

- Must use same probe function to “retrace the trail” for the data
- Unsuccessful search when reach empty position

What about `delete`?

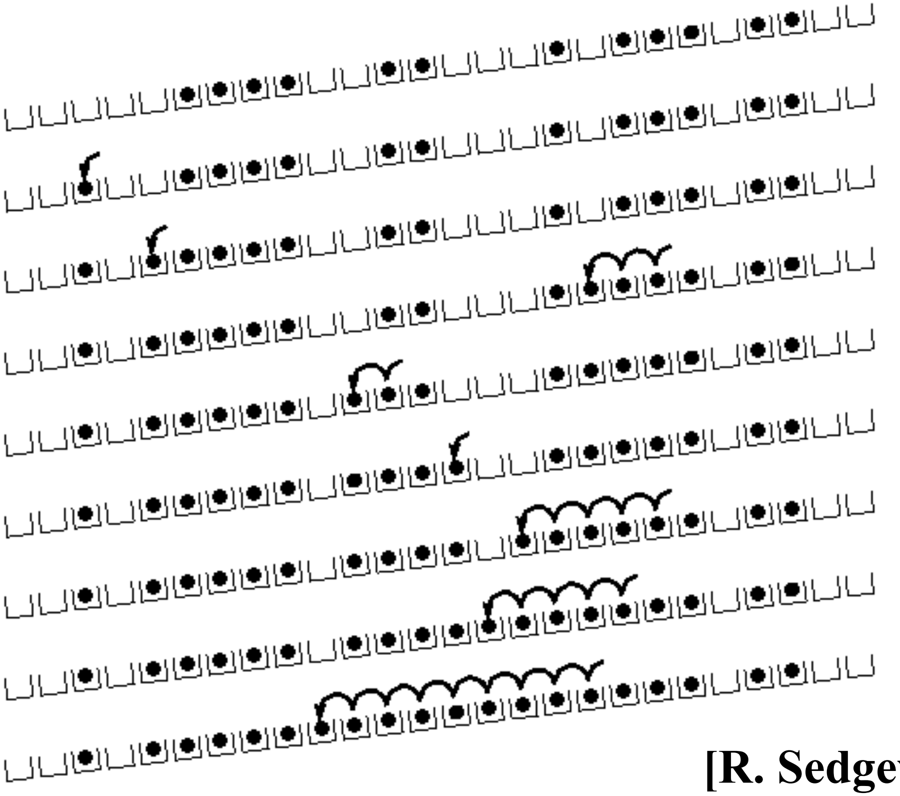
- ***Must*** use “lazy” deletion. Why?
  - Marker indicates “no data here, but don’t stop probing”
- Note: `delete` with chaining is plain-old list-remove

# *(Primary) Clustering*

It turns out linear probing is a *bad idea*, even though the probe function is quick to compute (which is a good thing)

Tends to produce *clusters*, which lead to long probing sequences

- Called primary clustering
- Saw this starting in our example



[R. Sedgewick]

# *Analysis of Linear Probing*

- Trivial fact: For any $\lambda < 1$, linear probing will find an empty slot
  - It is "safe" in this sense: no infinite loop unless table is full

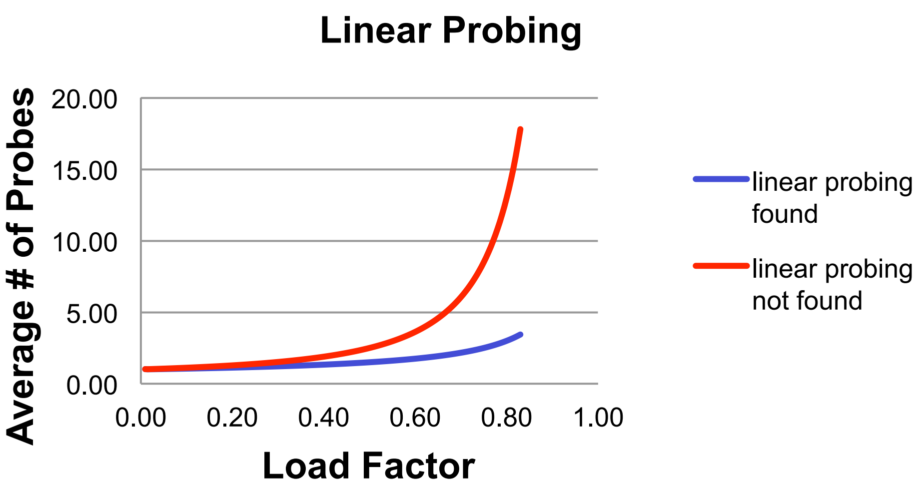
- Non-trivial facts we won't prove:

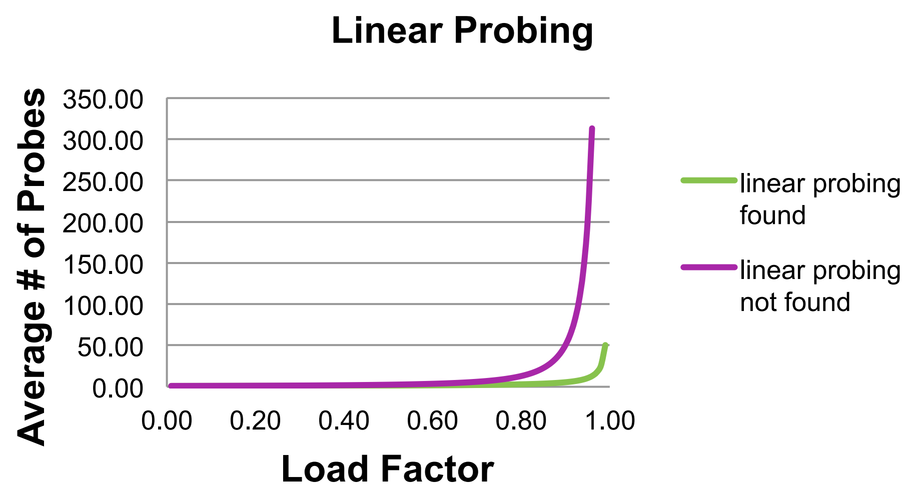
  Average # of probes given $\lambda$ (in the limit as **`TableSize`** $\rightarrow \infty$ )
  - Unsuccessful search: $$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$$
  - Successful search: $$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)}\right)$$

- This is pretty bad: need to leave sufficient empty space in the table to get decent performance (see chart)

# *In a chart*

- Linear-probing performance degrades rapidly as table gets full
  - (Formula assumes "large table" but point remains)




- By comparison, chaining performance is linear in $\lambda$ and has no trouble with $\boldsymbol{\lambda > 1}$

# *Quadratic probing*

- We can avoid primary clustering by changing the probe function

  **`(h(key) + f(i)) % TableSize`**

- A common technique is quadratic probing:

  $\mathtt{f(i)} = \mathtt{i}^2$

  – So probe sequence is:
    - 0th probe: **`h(key) % TableSize`**
    - 1st probe: **`(h(key) + 1) % TableSize`**
    - 2nd probe: **`(h(key) + 4) % TableSize`**
    - 3rd probe: **`(h(key) + 9) % TableSize`**
    - …
    - ith probe: $\mathtt{(h(key) + i^2) \% TableSize}$

- Intuition: Probes quickly "leave the neighborhood"

# *Quadratic Probing Example*

| | |
|---|---|
| 0 | |
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | |
| 7 | |
| 8 | |
| 9 | |

**TableSize=10**
**Insert:**
**89**
**18**
**49**
**58**
**79**

**$i^{th}$ probe:** `(h(key) + i²) % TableSize`

# *Quadratic Probing Example*

| | |
|---|---|
| 0 | |
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | |
| 7 | |
| 8 | |
| 9 | 89 |

**TableSize=10**

**Insert:**

**89**

**18**

**49**

**58**

**79**

**$i^{th}$ probe: `(h(key) + i`$^2$`) % TableSize`**

# *Quadratic Probing Example*

| | |
|---|---|
| 0 | |
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | |
| 7 | |
| 8 | 18 |
| 9 | 89 |

**TableSize=10**

**Insert:**

**89**

**18**

**49**

**58**

**79**

**$i^{th}$ probe: `(h(key) + i²) % TableSize`**

# *Quadratic Probing Example*

| | |
|---|---|
| 0 | 49 |
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | |
| 7 | |
| 8 | 18 |
| 9 | 89 |

**TableSize=10**

**Insert:**

**89**

**18**

**49**

**58**

**79**

**$i^{th}$ probe:** `(h(key) + i`$^2$`) % TableSize`

# *Quadratic Probing Example*

| | |
|---|---|
| 0 | 49 |
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | |
| 7 | |
| 8 | 18 |
| 9 | 89 |

**TableSize=10**
**Insert:**
**89**
**18**
**49**
**58**
**79**

**i^th probe:** `(h(key) + i²) % TableSize`

# *Quadratic Probing Example*

| | |
|---|---|
| 0 | 49 |
| 1 | |
| 2 | 58 |
| 3 | |
| 4 | |
| 5 | |
| 6 | |
| 7 | |
| 8 | 18 |
| 9 | 89 |

**TableSize=10**
**Insert:**
**89**
**18**
**49**
**58**
**79**

$i^{th}$ **probe:** `(h(key) + i`$^2$`) % TableSize`

# *Quadratic Probing Example*

| | |
|---|---|
| 0 | 49 |
| 1 | |
| 2 | 58 |
| 3 | |
| 4 | |
| 5 | |
| 6 | |
| 7 | |
| 8 | 18 |
| 9 | 89 |

**TableSize=10**
**Insert:**
**89**
**18**
**49**
**58**
**79**

**$i^{th}$ probe: `(h(key) + i²) % TableSize`**

# *Quadratic Probing Example*

| | |
|---|---|
| 0 | 49 |
| 1 | |
| 2 | 58 |
| 3 | 79 |
| 4 | |
| 5 | |
| 6 | |
| 7 | |
| 8 | 18 |
| 9 | 89 |

**TableSize=10**
**Insert:**
**89**
**18**
**49**
**58**
**79**

**i**$^{th}$ **probe:** `(h(key) + i`$^2$`) % TableSize`

# *Another Quadratic Probing Example*

| | |
|---|---|
| 0 | |
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | |

**TableSize = 7**

**Insert:**

| | |
|---|---|
| **76** | **(76 % 7 = 6)** |
| **40** | **(40 % 7 = 5)** |
| **48** | **(48 % 7 = 6)** |
| **5** | **( 5 % 7 = 5)** |
| **55** | **(55 % 7 = 6)** |
| **47** | **(47 % 7 = 5)** |

**$i^{th}$ probe: `(h(key) + i²) % TableSize`**

# *Another Quadratic Probing Example*

| | |
|---|---|
| **0** | |
| **1** | |
| **2** | |
| **3** | |
| **4** | |
| **5** | |
| **6** | 76 |

**TableSize = 7**

**Insert:**

| | |
|---|---|
| **76** | **(76 % 7 = 6)** |
| **40** | **(40 % 7 = 5)** |
| **48** | **(48 % 7 = 6)** |
| **5** | **( 5 % 7 = 5)** |
| **55** | **(55 % 7 = 6)** |
| **47** | **(47 % 7 = 5)** |

**$i^{th}$ probe: `(h(key) + i²) % TableSize`**

# *Another Quadratic Probing Example*

| | |
|---|---|
| **0** | |
| **1** | |
| **2** | |
| **3** | |
| **4** | |
| **5** | 40 |
| **6** | 76 |

**TableSize = 7**

**Insert:**

| | |
|---|---|
| **76** | **(76 % 7 = 6)** |
| **40** | **(40 % 7 = 5)** |
| **48** | **(48 % 7 = 6)** |
| **5** | **( 5 % 7 = 5)** |
| **55** | **(55 % 7 = 6)** |
| **47** | **(47 % 7 = 5)** |

**$i^{th}$ probe: `(h(key) + i²) % TableSize`**

# *Another Quadratic Probing Example*

| | |
|---|---|
| **0** | 48 |
| **1** | |
| **2** | |
| **3** | |
| **4** | |
| **5** | 40 |
| **6** | 76 |

**TableSize = 7**

**Insert:**

| | |
|---|---|
| **76** | **(76 % 7 = 6)** |
| **40** | **(40 % 7 = 5)** |
| **48** | **(48 % 7 = 6)** |
| **5** | **( 5 % 7 = 5)** |
| **55** | **(55 % 7 = 6)** |
| **47** | **(47 % 7 = 5)** |

**i$^{th}$ probe: `(h(key) + i`$^2$`) % TableSize`**

# *Another Quadratic Probing Example*

| | |
|---|---|
| **0** | 48 |
| **1** | |
| **2** | 5 |
| **3** | |
| **4** | |
| **5** | 40 |
| **6** | 76 |

**TableSize = 7**

**Insert:**

| | |
|---|---|
| **76** | **(76 % 7 = 6)** |
| **40** | **(40 % 7 = 5)** |
| **48** | **(48 % 7 = 6)** |
| **5** | **( 5 % 7 = 5)** |
| **55** | **(55 % 7 = 6)** |
| **47** | **(47 % 7 = 5)** |

**i$^{th}$ probe: `(h(key) + i`$^2$`) % TableSize`**

# *Another Quadratic Probing Example*

| | |
|---|---|
| **0** | 48 |
| **1** | |
| **2** | 5 |
| **3** | 55 |
| **4** | |
| **5** | 40 |
| **6** | 76 |

**TableSize = 7**

**Insert:**

| | |
|---|---|
| **76** | **(76 % 7 = 6)** |
| **40** | **(40 % 7 = 5)** |
| **48** | **(48 % 7 = 6)** |
| **5** | **( 5 % 7 = 5)** |
| **55** | **(55 % 7 = 6)** |
| **47** | **(47 % 7 = 5)** |

**$i^{th}$ probe: `(h(key) + i²) % TableSize`**

# *Another Quadratic Probing Example*

| | |
|---|---|
| **0** | 48 |
| **1** | |
| **2** | 5 |
| **3** | 55 |
| **4** | |
| **5** | 40 |
| **6** | 76 |

**TableSize = 7**

**Insert:**

| | |
|---|---|
| **76** | **(76 % 7 = 6)** |
| **40** | **(40 % 7 = 5)** |
| **48** | **(48 % 7 = 6)** |
| **5** | **( 5 % 7 = 5)** |
| **55** | **(55 % 7 = 6)** |
| **47** | **(47 % 7 = 5)** |

Yikes!: For all *n*, **`((n*n) +5) % 7 is 0, 2, 5, or 6`**

- Excel shows takes “at least” 50 probes and a pattern
- Turns out 47 is never placed in the table

# *From Bad News to Good News*

- Bad news:
  - Quadratic probing can cycle through the same full indices, never terminating despite table not being full
- Good news:
  - If **`TableSize`** is *prime* and $\lambda < 1/2$, then quadratic probing will find an empty slot in at most **`TableSize/2`** probes
  - So: If you keep $\lambda < 1/2$ and **`TableSize`** is *prime*, no need to detect cycles
  - Proof is posted in **`lecture13.txt`**
    - Also, slightly less detailed proof in textbook
    - Key fact: For prime $\mathbf{T}$ and $\mathbf{0 < i,j < T/2}$ where $\mathbf{i \neq j}$, $\mathbf{(k + i^2) \% T \neq (k + j^2) \% T}$ (i.e., no index repeat)

# *Clustering reconsidered*

- Quadratic probing does not suffer from primary clustering: no problem with keys initially hashing to the same neighborhood

- But it’s no help if keys initially hash to the same index
  - Called secondary clustering

- Can avoid secondary clustering with a probe function that depends on the key: double hashing…

# *Double hashing*

Idea:

- Given two good hash functions *h* and *g*, it is very unlikely that for some *key*, **`h(key) == g(key)`**
- So make the probe function **`f(i) = i*g(key)`**

Probe sequence:

- $0^{th}$ probe: **`h(key) % TableSize`**
- $1^{st}$ probe: **`(h(key) + g(key))   % TableSize`**
- $2^{nd}$ probe: **`(h(key) + 2*g(key)) % TableSize`**
- $3^{rd}$ probe: **`(h(key) + 3*g(key)) % TableSize`**
- …
- $i^{th}$ probe: **`(h(key) + i*g(key)) % TableSize`**

Detail: Make sure **`g(key)`** cannot be **`0`**

# *Double-hashing analysis*

- Intuition: Because each probe is "jumping" by **`g(key)`** each time, we "leave the neighborhood" *and* "go different places from other initial collisions"
- But we could still have a problem like in quadratic probing where we are not "safe" (infinite loop despite room in table)
  - It is known that this cannot happen in at least one case:
    - **`h(key) = key % p`**
    - **`g(key) = q – (key % q)`**
    - **`2 < q < p`**
    - **`p`** and **`q`** are prime

# *More double-hashing facts*

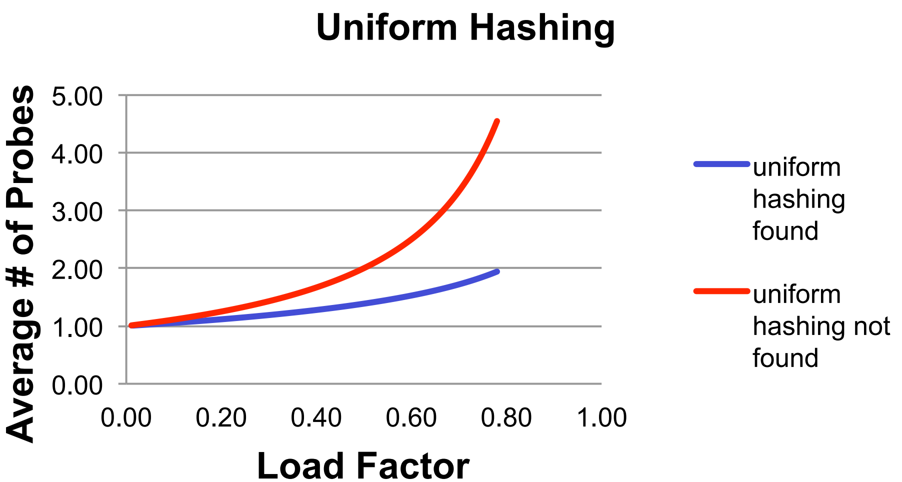
- Assume "uniform hashing"
  - Means probability of `g(key1) % p == g(key2) % p` is `1/p`

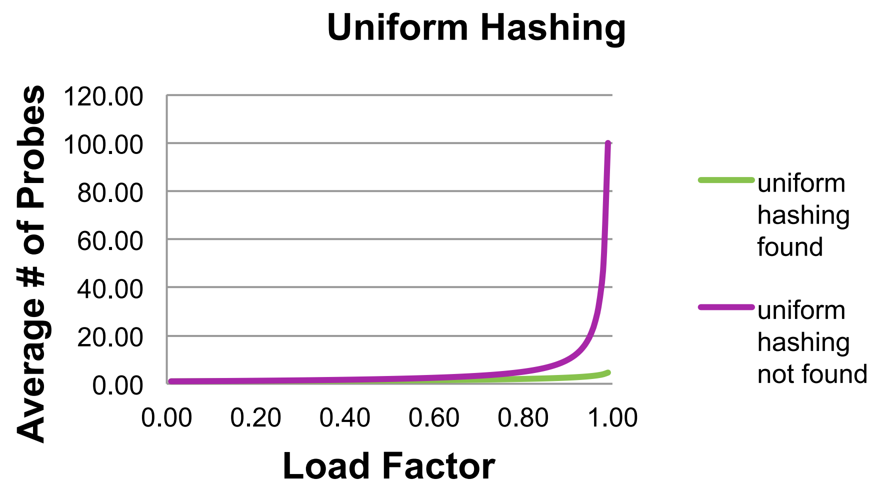
- Non-trivial facts we won't prove:

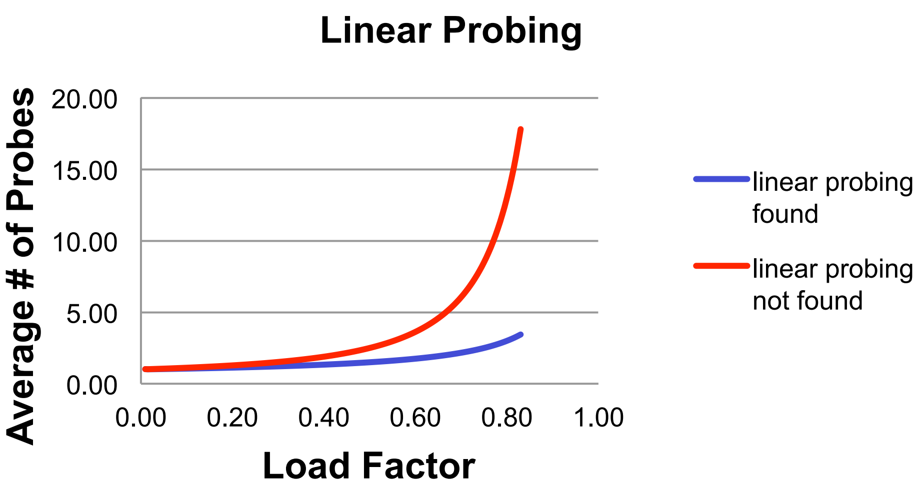
  Average # of probes given $\lambda$ (in the limit as `TableSize` $\rightarrow \infty$ )
  - Unsuccessful search (intuitive): $$\frac{1}{1-\lambda}$$
  - Successful search (less intuitive): $$\frac{1}{\lambda}\log_e\left(\frac{1}{1-\lambda}\right)$$

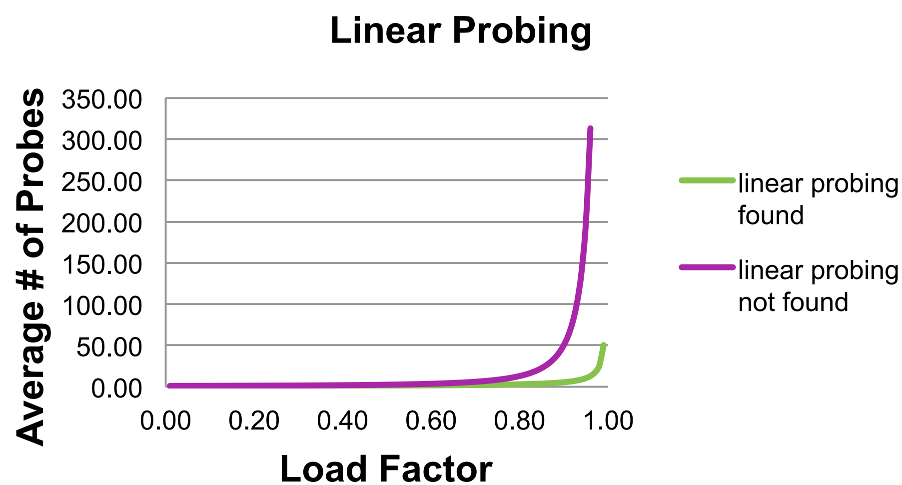
- Bottom line: unsuccessful bad (but not as bad as linear probing), but successful is not nearly as bad

# *Charts*


Uniform Hashing
Average # of Probes
5.00
4.00
3.00
2.00
1.00
0.00
0.00
0.20
0.40
0.60
0.80
1.00
Load Factor
uniform hashing found
uniform hashing not found


Uniform Hashing
Average # of Probes
120.00
100.00
80.00
60.00
40.00
20.00
0.00
0.00
0.20
0.40
0.60
0.80
1.00
Load Factor
uniform hashing found
uniform hashing not found


Linear Probing
Average # of Probes
20.00
15.00
10.00
5.00
0.00
0.00
0.20
0.40
0.60
0.80
1.00
Load Factor
linear probing found
linear probing not found


Linear Probing
Average # of Probes
350.00
300.00
250.00
200.00
150.00
100.00
50.00
0.00
0.00
0.20
0.40
0.60
0.80
1.00
Load Factor
linear probing found
linear probing not found

# *Rehashing*

- As with array-based stacks/queues/lists, if table gets too full, create a bigger table and copy everything
- With chaining, we get to decide what "too full" means
  - Keep load factor reasonable (e.g., < 1)?
- For probing, half-full is a good rule of thumb
- New table size
  - Twice-as-big is a good idea, except that won't be prime!
  - So go *about* twice-as-big
  - Can have a list of prime numbers in your code since you won't grow more than 20-30 times

# *Hashtable Scenarios*

- For each of the scenarios, answer the following questions:
  - Is a hashtable the best-suited data structure?
  - If so, what would be used for the keys? Values?
  - If not, what data structure would be best-suited?
  - What other assumptions are you making about the scenario?

- Catalog of items (product id, name, price)
- Bookmarks in a web browser (favicon, URL, bookmark name)
- IT support requests (timestamp, ticket id, description)
- Character frequency analysis (character, # of appearances)
- Spell-checking (all or most words in a language)

# *Homework 4*

- Read through the provided code files
- Implement  DataCount[] getCountsArray(DataCounter counter) method of WordCount
  - use iterator of counter to get elements and put in a new array (which is returned
- Implement compare(string, string) method of StringComparator
  - return 0 if the same, negative number if first argument comes alphabetically first
- Implement two implementations of DataCounter
  - HashTable_SC: hash table using separate chaining
  - HashTable_OA: hash table using open addressing
  - StringHasher: hash function for string
- Fill in code in Correlator.java to compare documents
- Test your solutions and turn in testing code
- README
  - do some timing, write another hash function