



# CSE 332: Data Structures & Parallelism

## Lecture 10: Hashing

Ruth Anderson  
Winter 2019

# *Today*

- Dictionaries
  - Hashing

# Motivating Hash Tables

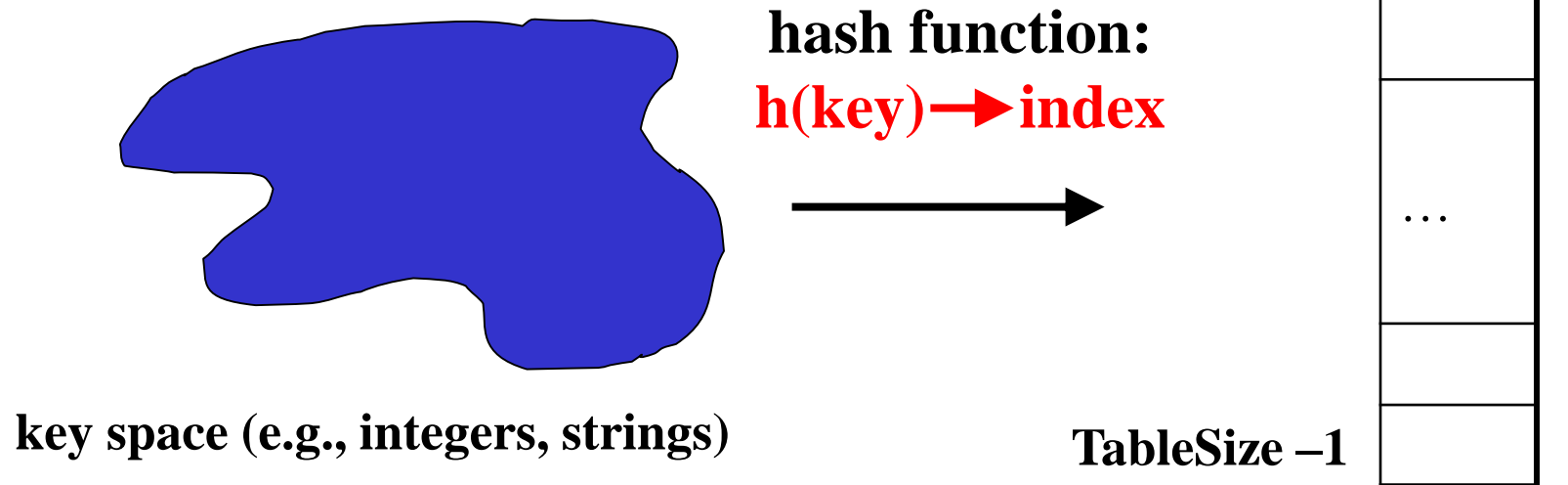
For dictionary with  $n$  key/value pairs

	<b>insert</b>	<b>find</b>	<b>delete</b>
• Unsorted linked-list	$O(n)$ *	$O(n)$	$O(n)$
• Unsorted array	$O(n)$ *	$O(n)$	$O(n)$
• Sorted linked list	$O(n)$	$O(n)$	$O(n)$
• Sorted array	$O(n)$	$O(\log n)$	$O(n)$
• <i>Balanced</i> tree	$O(\log n)$	$O(\log n)$	$O(\log n)$

\* Assuming we must check to see if the key has already been inserted.  
Cost becomes cost of a find operation, inserting itself is  $O(1)$ .

# Hash Tables

- 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
- Basic idea:



# Aside: Hash Tables vs. Balanced Trees

- In terms of a Dictionary ADT for just **insert**, **find**, **delete**, hash tables and balanced trees are just different data structures
  - Hash tables  $O(1)$  on average (*assuming* few collisions)
  - Balanced trees  $O(\log n)$  worst-case
- Constant-time is better, right?
  - Yes, but you need “hashing to behave” (must avoid collisions)
  - Yes, but what if we want to **findMin**, **findMax**, **predecessor**, and **successor**, **printSorted**?
    - Hashtables are not designed to efficiently implement these operations
    - Your textbook considers Hash tables to be a different ADT
    - Not so important to argue over the definitions

# Hash Tables

- There are  $m$  possible keys ( $m$  typically large, even infinite)
- We expect our table to have only  $n$  items
- $n$  is much less than  $m$  (often written  $n \ll m$ )

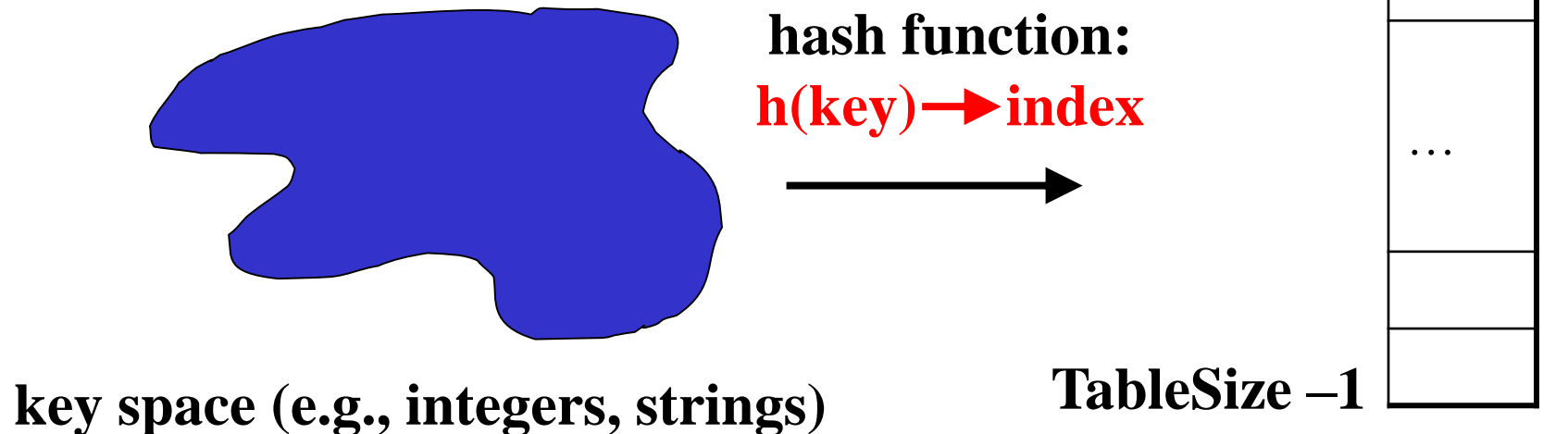
Many dictionaries have this property

- Compiler: All possible identifiers allowed by the language vs. those used in some file of one program
- Database: All possible student names vs. students enrolled
- AI: All possible chess-board configurations vs. those considered by the current player
- ...

# Hash functions

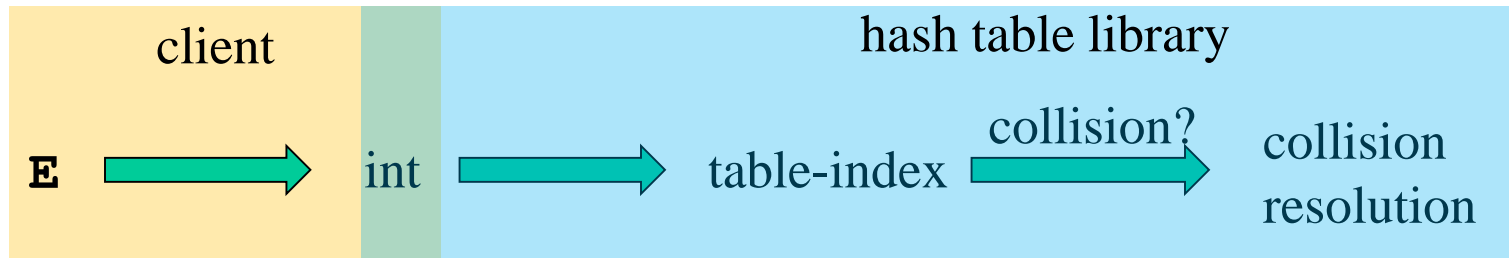
An ideal hash function:

- Is fast to compute
- “Rarely” hashes two “used” keys to the same index
  - Often impossible in theory; easy in practice
  - Will handle *collisions* a bit later



# Who hashes what?

- Hash tables can be generic
  - To store keys of type **E**, we just need to be able to:
    1. Test equality: are you the **E** I'm looking for?
    2. Hashable: convert any **E** to an **int**
- When hash tables are a reusable library, the division of responsibility generally breaks down into two roles:

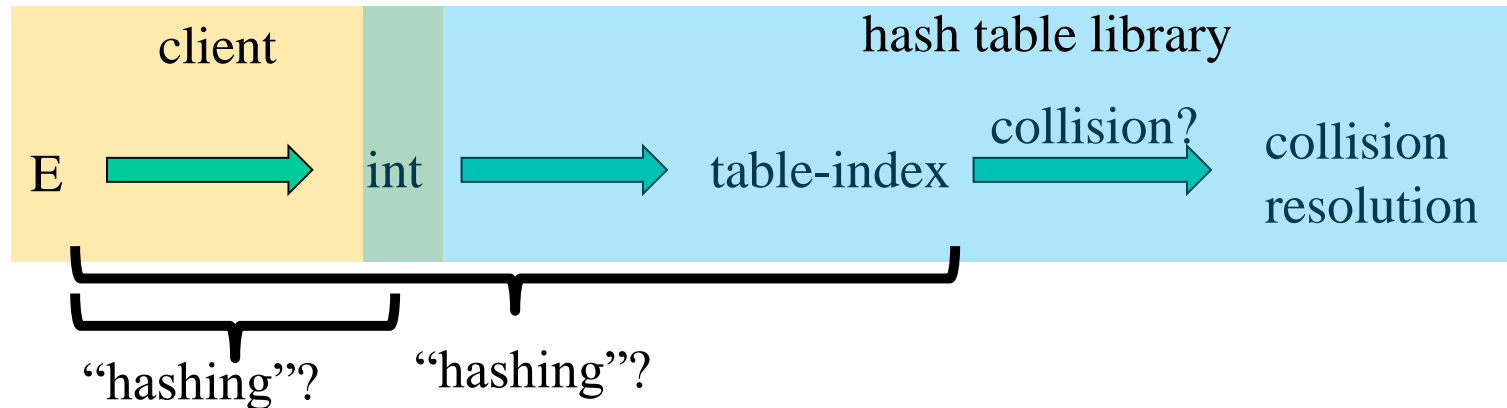


- We will learn both roles, but most programmers “in the real world” spend more time as clients while understanding the library



# More on roles

Some ambiguity in terminology on which parts are “hashing”



Two roles must both contribute to minimizing collisions (heuristically)

- Client should aim for different ints for expected items
  - Avoid “wasting” any part of **E** or the 32 bits of the **int**
- Library should aim for putting “similar” **ints** in different indices
  - conversion to index is almost always “mod table-size”
  - using prime numbers for table-size is common

# *What to hash?*

- We will focus on two most common things to hash: ints and strings
- If you have objects with several fields, it is usually best to have most of the “identifying fields” contribute to the hash to avoid collisions
- Example:

```
class Person {  
    String first; String middle; String last;  
    Date birthdate;  
}
```
- An inherent trade-off: hashing-time vs. collision-avoidance
  - Use all the fields?
  - Use only the birthdate?
  - Admittedly, what-to-hash is often an unprincipled guess ☹️

# Hashing integers

key space = integers

Simple hash function:

$$h(\text{key}) = \text{key} \% \text{TableSize}$$

- Client:  $f(x) = x$
- Library  $g(x) = f(x) \% \text{TableSize}$
- Fairly fast and natural

Example:

- $\text{TableSize} = 10$
- Insert 7, 18, 41, 34, 10
- (As usual, ignoring corresponding data)

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

# *Collision-avoidance*

- With “**x % TableSize**” the number of collisions depends on
  - the ints inserted (obviously)
  - **TableSize**
- Larger table-size tends to help, but not always
  - Example: 70, 24, 56, 43, 10  
with **TableSize = 10** and **TableSize = 60**
- Technique: Pick table size to be prime. Why?
  - Real-life data tends to have a pattern
  - “Multiples of 61” are probably less likely than “multiples of 60”
  - We’ll see some collision strategies do better with prime size

# *More arguments for a prime table size*

If **TableSize** is 60 and...

- Lots of keys are multiples of 5, wasting 80% of table
- Lots of keys are multiples of 10, wasting 90% of table
- Lots of keys are multiples of 2, wasting 50% of table

If **TableSize** is 61...

- Collisions can still happen, but 5, 10, 15, 20, ... will fill table
- Collisions can still happen but 10, 20, 30, 40, ... will fill table
- Collisions can still happen but 2, 4, 6, 8, ... will fill table

In general, if **x** and **y** are “co-prime” (means  $\text{gcd}(\mathbf{x}, \mathbf{y}) == 1$ ), then

$$(\mathbf{a} * \mathbf{x}) \% \mathbf{y} == (\mathbf{b} * \mathbf{x}) \% \mathbf{y} \text{ if and only if } \mathbf{a} \% \mathbf{y} == \mathbf{b} \% \mathbf{y}$$

- Given table size **y** and keys as multiples of **x**, we'll get a decent distribution if **x** & **y** are co-prime
- So good to have a **TableSize** that has no common factors with any “likely pattern” **x**

# What if the key is not an int?

- If keys aren't `ints`, the **client** must convert to an `int`
  - Trade-off: speed and distinct keys hashing to distinct `ints`
- Common and important example: Strings
  - Key space  $K = s_0s_1s_2 \dots s_{m-1}$ 
    - where  $s_i$  are chars:  $s_i \in [0,256]$
  - Some choices: Which avoid collisions best?

1.  $h(K) = s_0$

2.  $h(K) = \left( \sum_{i=0}^{m-1} s_i \right)$

3.  $h(K) = \left( \sum_{i=0}^{m-1} s_i \cdot 37^i \right)$

Then on the **library** side we typically mod by Tablesize to find index into the table

# *Specializing hash functions*

How might you hash differently if all your strings were web addresses (URLs)?

# *Aside: Combining hash functions*

A few rules of thumb / tricks:

1. Use all 32 bits (careful, that includes negative numbers)
2. Use different overlapping bits for different parts of the hash
  - This is why a factor of  $37^i$  works better than  $256^i$
3. When smashing two hashes into one hash, use bitwise-xor
  - bitwise-and produces too many 0 bits
  - bitwise-or produces too many 1 bits
4. Rely on expertise of others; consult books and other resources
5. If keys are known ahead of time, choose a *perfect hash*



# *Collision resolution*

## Collision:

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

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

So hash tables should support **collision resolution**

– Ideas?

# *Flavors of Collision Resolution*

Separate Chaining

Open Addressing

- Linear Probing
- Quadratic Probing
- Double Hashing

# Separate Chaining

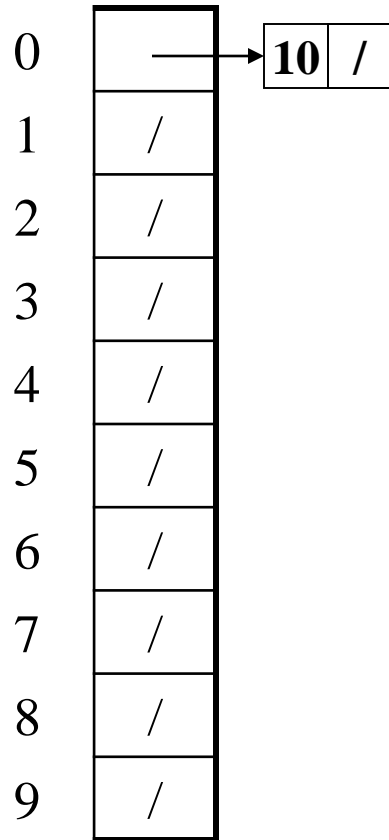
0	/
1	/
2	/
3	/
4	/
5	/
6	/
7	/
8	/
9	/

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

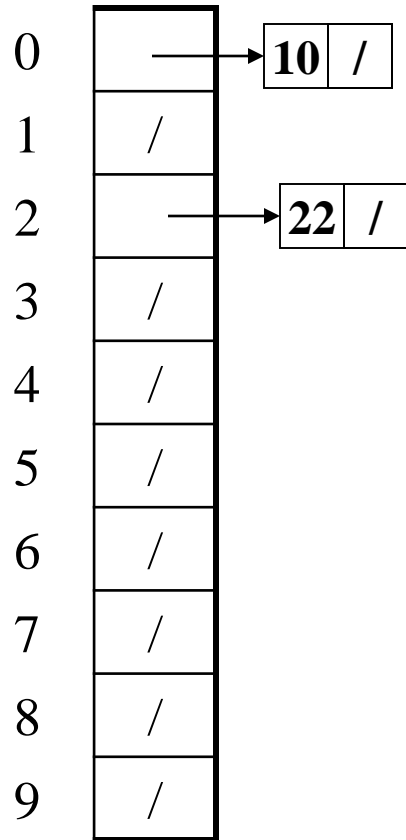


Chaining: All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

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

# Separate Chaining

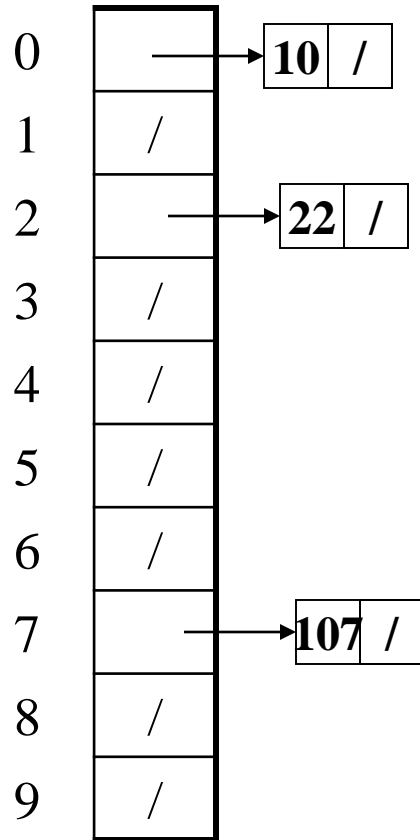


Chaining: All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

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

# Separate Chaining

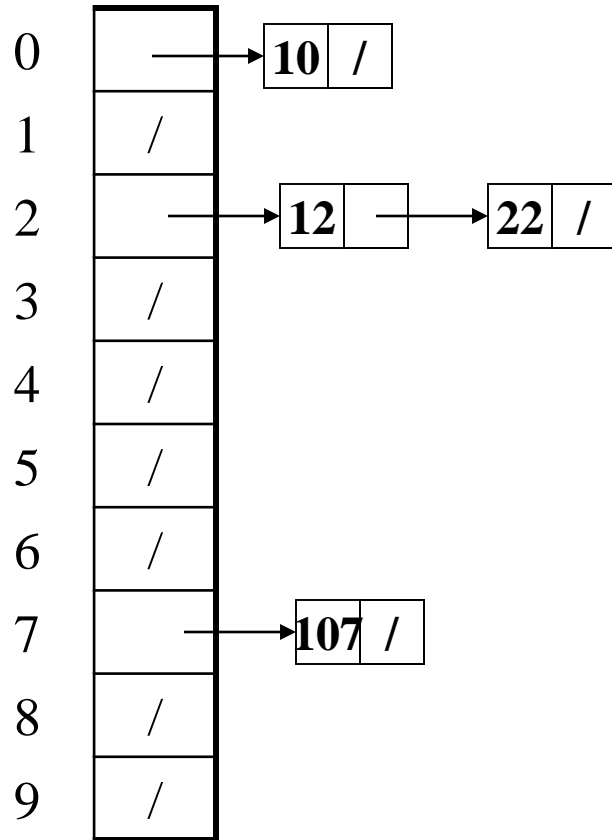


Chaining: All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

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

# Separate Chaining

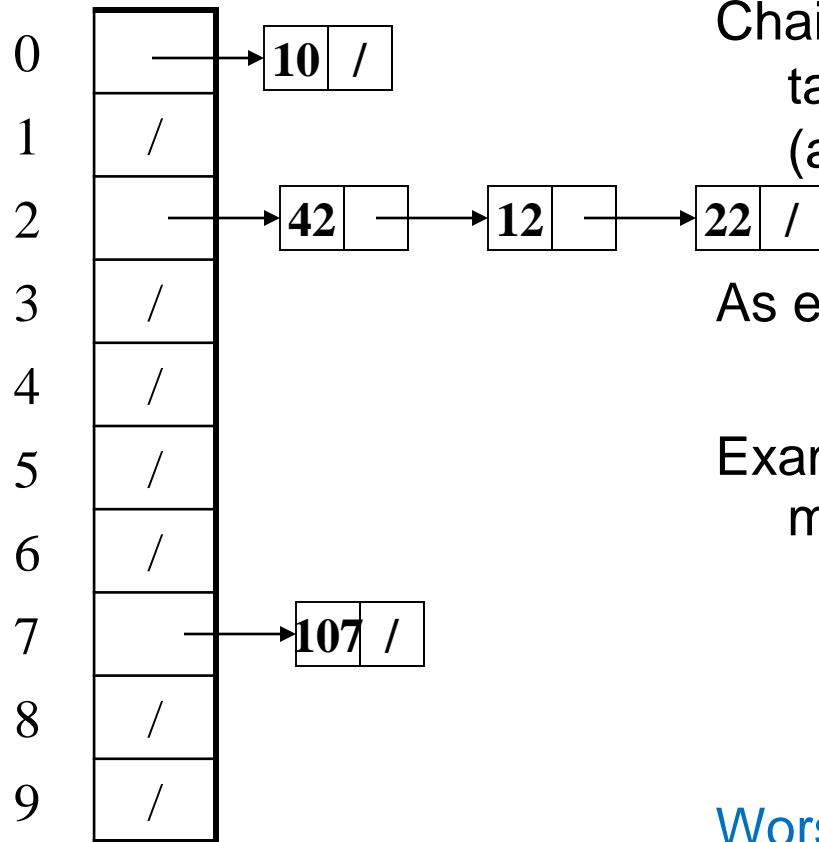


Chaining: All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

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

Worst case time for find?

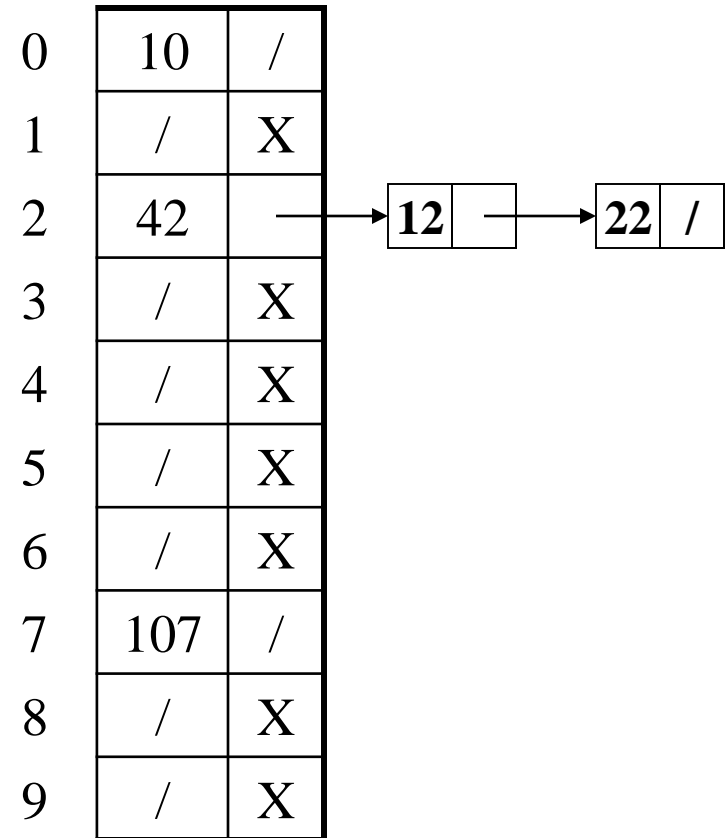
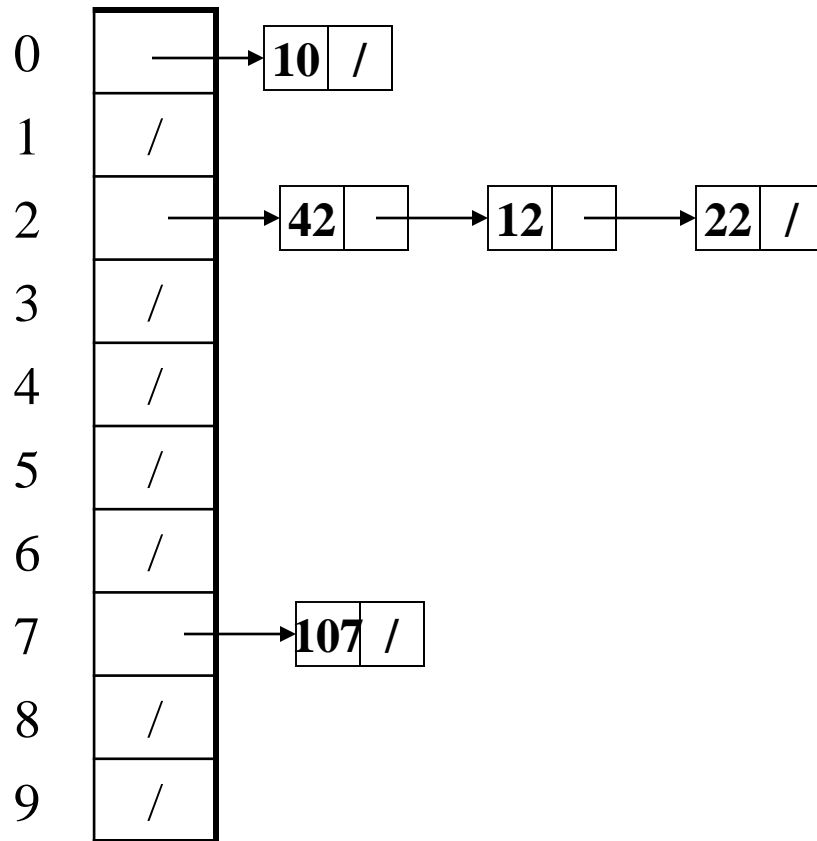


# *Thoughts on separate chaining*

- 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)
    - Keep # of items in each bucket small
    - Overhead of AVL tree, etc. not worth it for small n
- Beyond asymptotic complexity, some “data-structure engineering” can improve constant factors
  - Linked list vs. array or a hybrid of the two
  - Move-to-front (part of Project 2)
  - Leave room for 1 element (or 2?) in the table itself, to optimize constant factors for the common case
    - A time-space trade-off...

# Time vs. space

(only makes a difference in constant factors)



# More rigorous separate chaining analysis

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

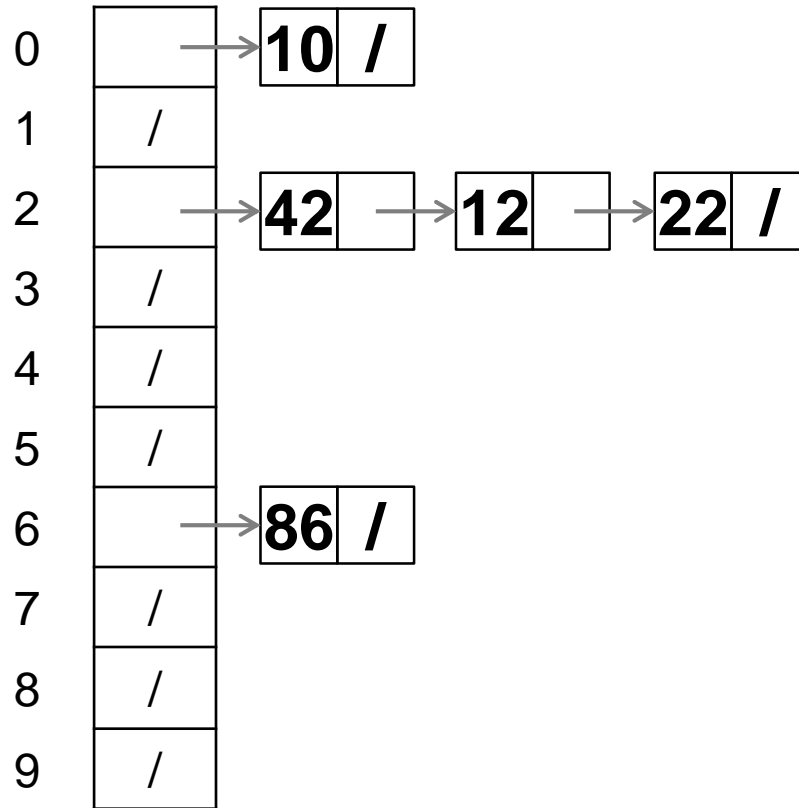
$$\lambda = \frac{N}{\text{TableSize}} \quad \leftarrow \text{number of elements}$$

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

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

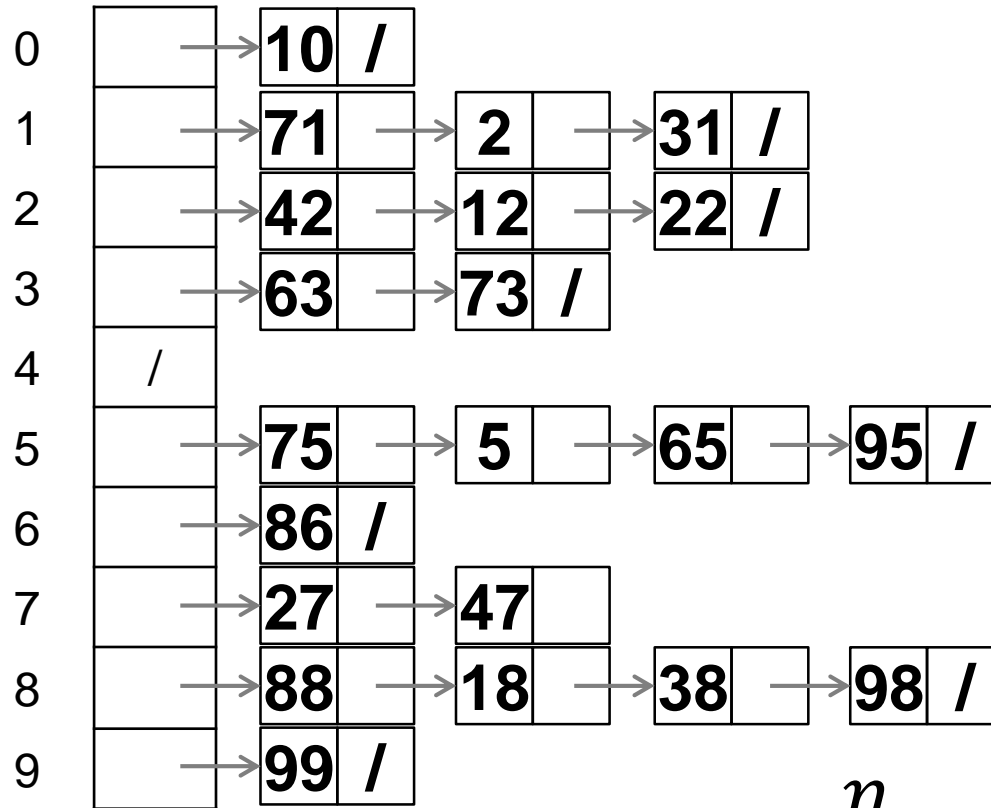
- Each unsuccessful **find** compares against \_\_\_\_ items
- Each successful **find** compares against \_\_\_\_ items
  
- How big should TableSize be??

# Load Factor?



$$\lambda = \frac{n}{TableSize} = ?$$

# Load Factor?



$$\lambda = \frac{n}{TableSize} = ?$$

# *Separate Chaining Deletion?*

# Separate Chaining Deletion

- Not too bad
  - Find in table
  - Delete from bucket
- Say, delete 12
- Similar run-time as insert

