CSE373: Data Structures & Algorithms

Lecture 6: Hash Tables

Hunter Zahn

Summer 2016

Motivating Hash Tables

For a **dictionary** with *n* key, value pairs

	`insert`	`find`	`delete`
• Unsorted linked-list	$O(1)$	$O(n)$	$O(n)$
• Unsorted array	$O(1)$	$O(n)$	$O(n)$
• Sorted linked list	$O(n)$	$O(n)$	$O(n)$
• Sorted array	$O(n)$	$O(\log n)$	$O(n)$
• *Balanced* tree	$O(\log n)$	$O(\log n)$	$O(\log n)$
• Magic array	$O(1)$	$O(1)$	$O(1)$

Sufficient "magic":

- Use key to compute array index for an item in $O(1)$ time [doable]
- Have a different index for every item [magic]

Motivating Hash Tables

- Let's say you are tasked with counting the frequency of integers in a text file. You are guaranteed that only the integers 0 through 100 will occur:

  For example: 5, 7, 8, 9, 9, 5, 0, 0, 1, 12

  Result: 0 → 2 1 → 1 5 → 2 7 → 1 8 → 1 9 → 2

  What structure is appropriate?

  Tree?

  List?

  Array?

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

Motivating Hash Tables

Now what if we want to associate name to phone number?

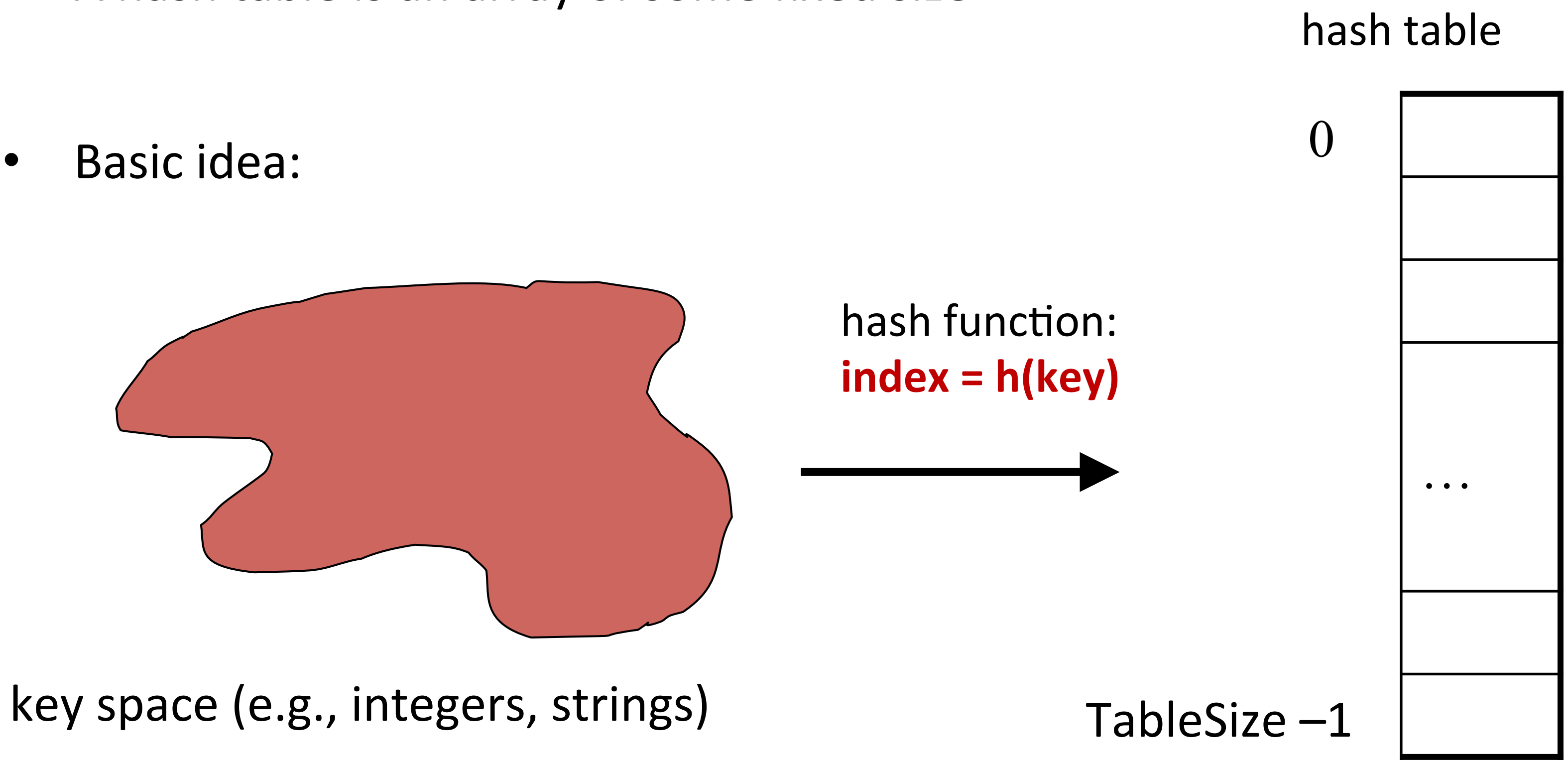
Suppose keys are first, last names

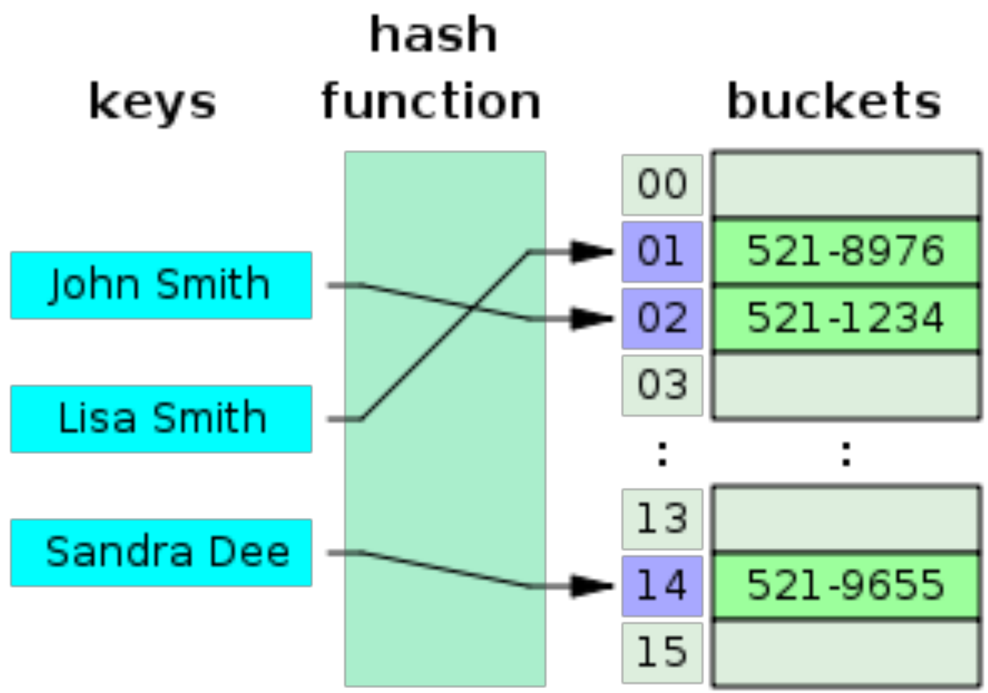
– how big is the key space?

Maybe we only care about students

Hash Tables

- Aim for constant-time (i.e., *O*(1)) **`find`**, **`insert`**, and **`delete`**
  - "On average" under some often-reasonable assumptions
- A hash table is an array of some fixed size
- Basic idea:




keys
hash function
buckets
John Smith
Lisa Smith
Sandra Dee
00
01
02
03
13
14
15
521-8976
521-1234
521-9655

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* we follow good practices)
  - Balanced trees $O(\texttt{log}\ n)$ worst-case
- Constant-time is better, right?
  - Yes, but you need "hashing to behave" (must avoid collisions)
  - Yes, but **`findMin`**, **`findMax`**, **`predecessor`**, and **`successor`** go from $O(\texttt{log}\ n)$ to $O(n)$, **`printSorted`** from $O(n)$ to $O(n\ \texttt{log}\ n)$
    - Why your textbook considers this to be a different ADT

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 << m***)

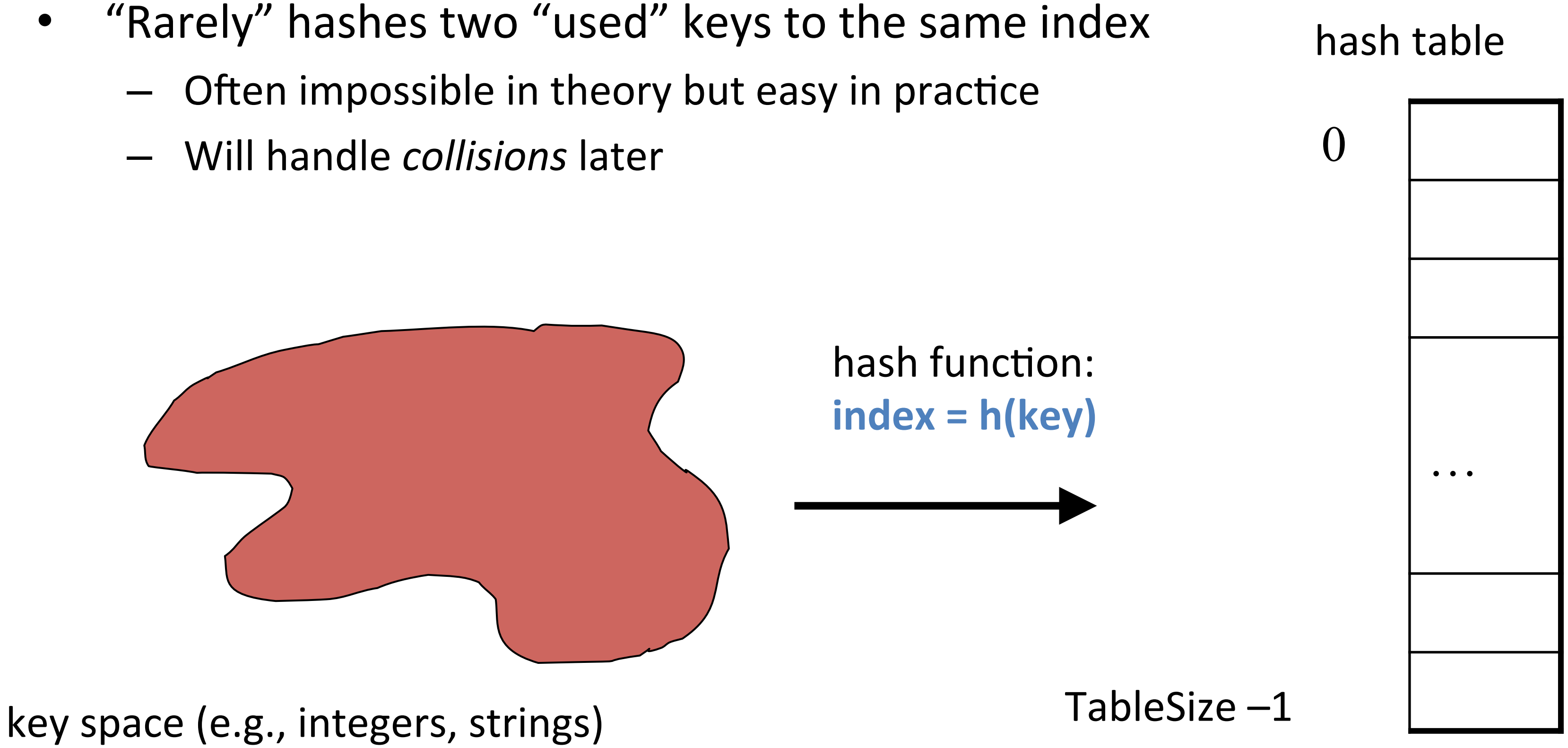
Many dictionaries have this property

- 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:

- Fast to compute
- "Rarely" hashes two "used" keys to the same index
  - Often impossible in theory but easy in practice
  - Will handle *collisions* later



Simple Integer Hash Functions

- key space K = integers
- TableSize = 7

- h(K) = K % 7

- **Insert**: 7, 18, 41

0	7
1	
2	
3	
4	18
5	
6	41

Simple Integer Hash Functions

- key space K = integers
- TableSize = 10
- h(K) = ??
- **Insert**: 7, 18, 41, 34
  - What happens when we insert 44?

0	
1	41
2	
3	
4	34
5	
6	
7	7
8	18
9	

Aside: Properties of Mod

To keep hashed values within the size of the table, we will generally do:

h(K) = function(K) % TableSize

(In the previous examples, function(K) = K.)

Useful properties of mod:

- (a + b) % c = [(a % c) + (b % c)] % c
- (a b) % c = [(a % c) (b % c)] % c
- a % c = b % c → (a – b) % c = 0

Designing Hash Functions

Often based on **modular hashing**:

$$h(K) = f(K) \% P$$

P is typically the TableSize

P is often chosen to be prime:

- Reduces likelihood of collisions due to patterns in data
- Is useful for guarantees on certain hashing strategies (as we'll see)

Equivalent objects MUST hash to the same location

Designing Hash Functions:

- h(K) = f(K) % P
  - f(K) = ??

Some String Hash Functions

key space = strings

$K = s_0\ s_1\ s_2\ \ldots\ s_{m-1}$ (where s_i are chars: $s_i \in [0, 128]$)

1. $h(K) = s_0$ % TableSize   H("batman") = H("ballgame")

2. $h(K) = \left(\sum_{i=0}^{m-1} s_i \right)$ % TableSize   H("spot") = H("pots")

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

What to hash?

We will focus on the two most common things to hash: *ints* and *strings*

- For objects with several fields, 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;
}
```

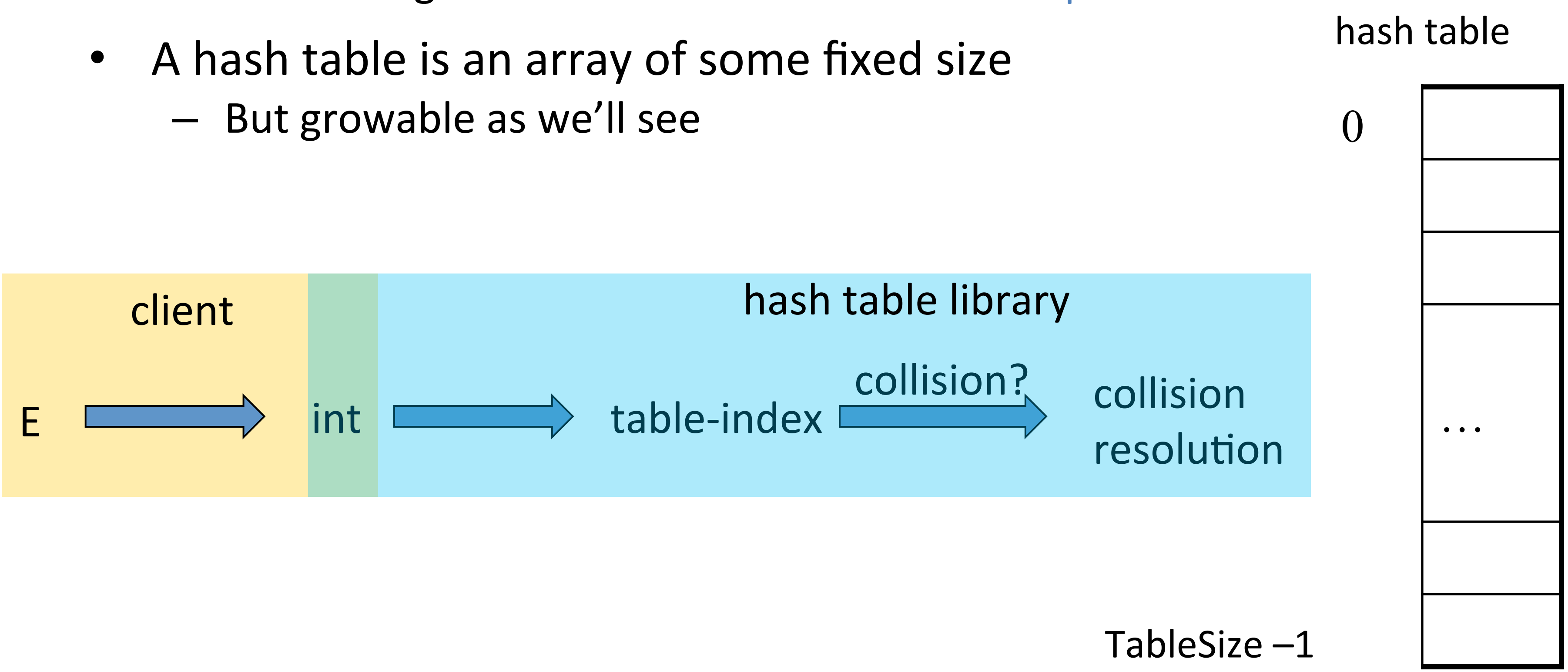
- An inherent trade-off: hashing-time vs. collision-avoidance
  - Bad idea(?): Use only first name
  - Good idea(?): Use only middle initial? Combination of fields?
  - Admittedly, what-to-hash-with is often unprincipled ☹

Deep Breath

- Recap

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 growable as we'll see



Collision resolution

Collision:

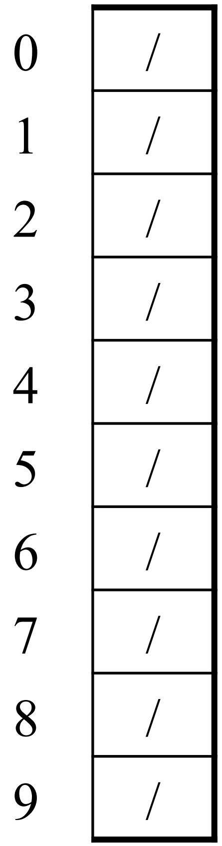
When two keys map to the same location in the hash table

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

So hash tables should support collision resolution

– Ideas?

Separate Chaining



Chaining:

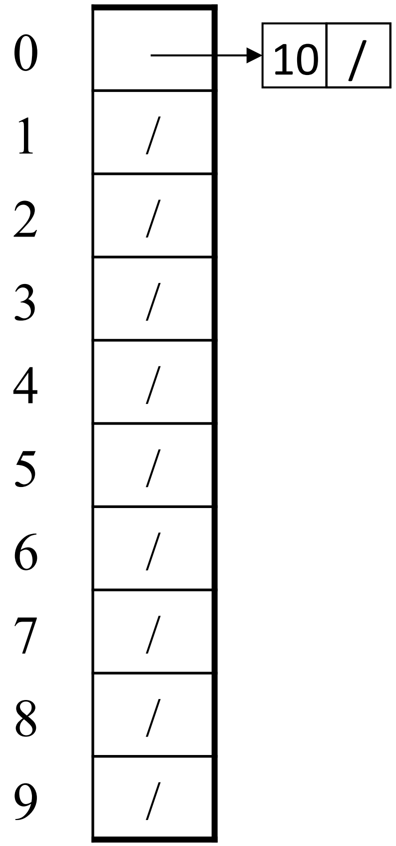
All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

As easy as it sounds

Example:

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

Separate Chaining



Chaining:

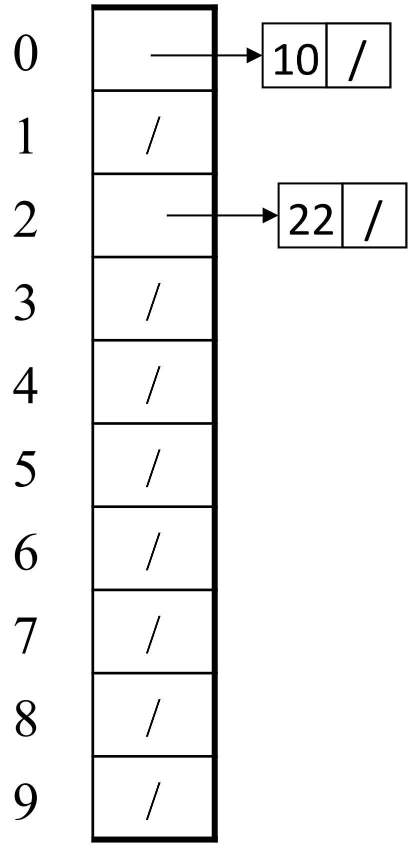
All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

As easy as it sounds

Example:

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

Separate Chaining



Chaining:

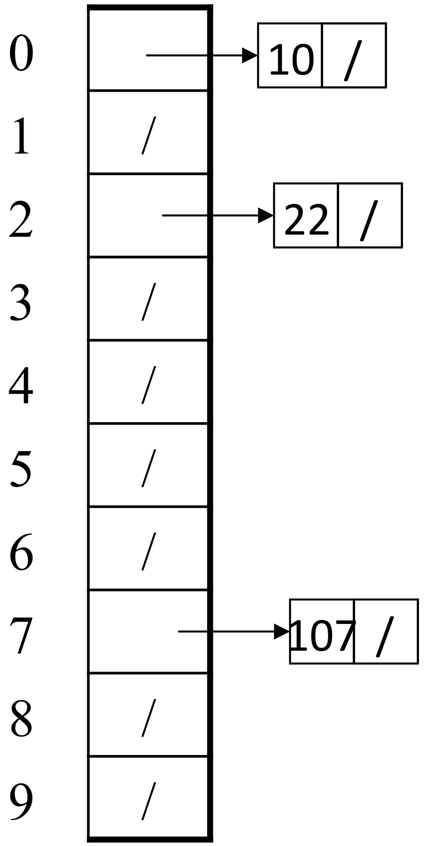
All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

As easy as it sounds

Example:

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

Separate Chaining



Chaining:

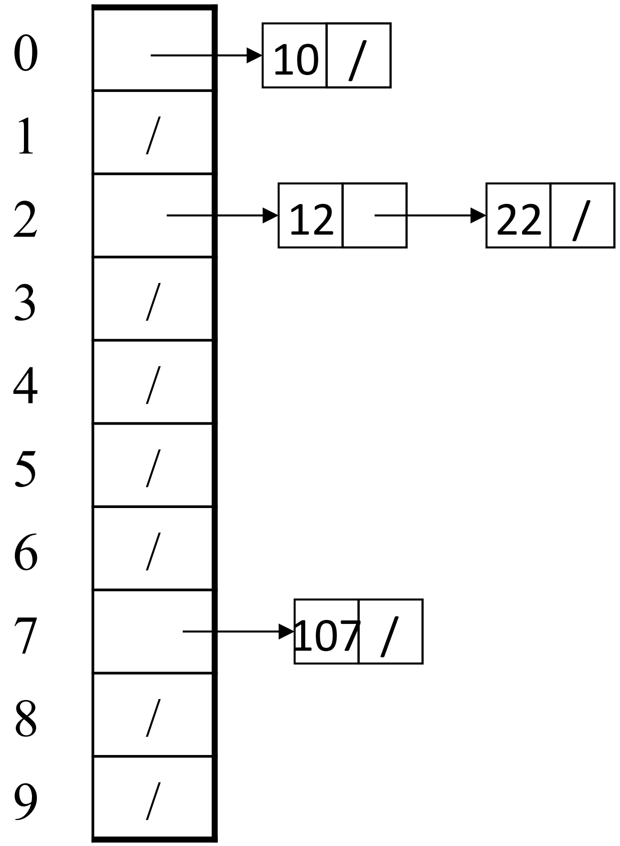
All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

As easy as it sounds

Example:

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

Separate Chaining



Chaining:

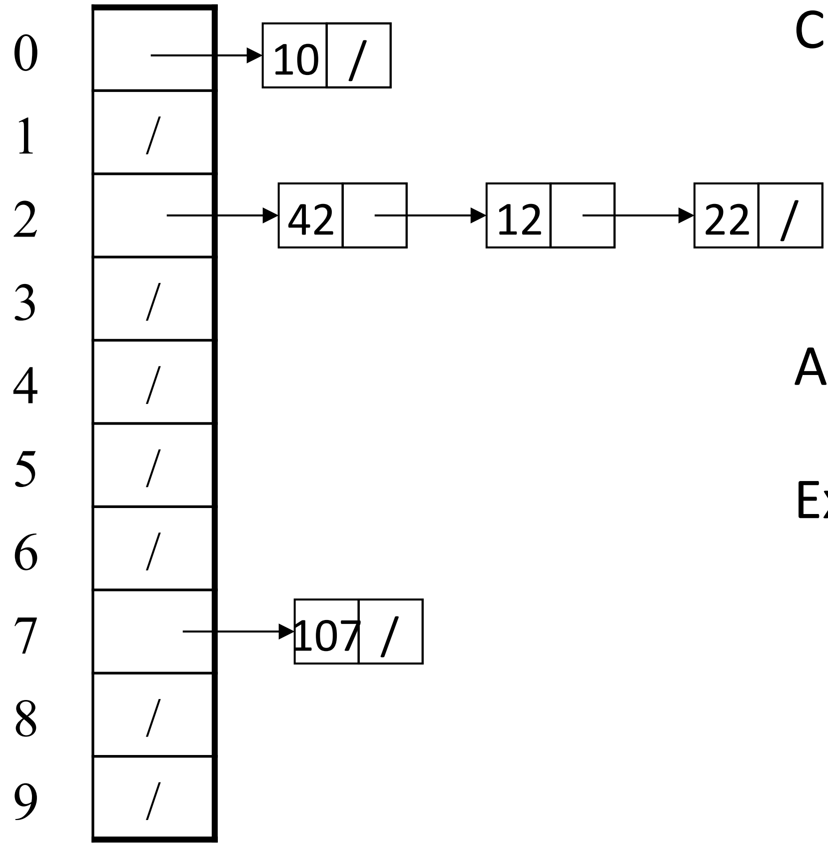
All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

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

More rigorous chaining analysis

Definition: The load factor, λ, of a hash table is

$$\lambda = \frac{\text{N}}{\text{TableSize}}$$

← number of elements

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

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

- Each "unsuccessful" `find` compares against λ items

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