CSE 332: Data Structures & Parallelism

Lecture 11:Hashing

Arthur Liu

Winter 2022

Thanks to Ruth Anderson for slides

General Reminders

- Exercise 5, 6 due date extended to this Friday
- P2 Checkpoint 1 due Friday

- Upcoming
  - Two last exercises before midterm (EX07, EX08) due next Wednesday
  - Midterm next Friday!

Today

- Dictionaries
  - Hash Tables

Motivating Hash Tables

For dictionary with n key/value pairs

	`insert`	**`find`**	**`delete`**
• 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	$O(n)$	$O(\log n)$	$O(n)$
• *Balanced* 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).

Motivating Hash Tables

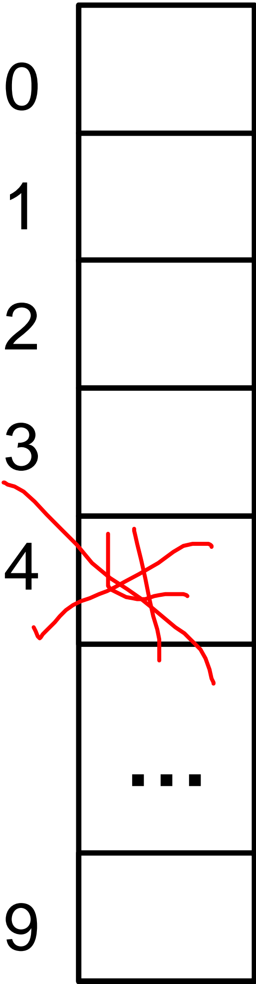
For dictionary with *n* key/value pairs

	`insert`	`find`	`delete`
• 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	$O(n)$	$O(\log n)$	$O(n)$
• *Balanced* tree	$O(\log n)$	$O(\log n)$	$O(\log n)$
• ***HashTables***	**$O(1)$**	**$O(1)$**	**$O(1)$**

* 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).

Really Big Array – my idea ☺

Really Big Array – my idea ☺



Keys: Student ID's

0 – 9,999,999

insert(4) - O(1)

find(4) - O(1)

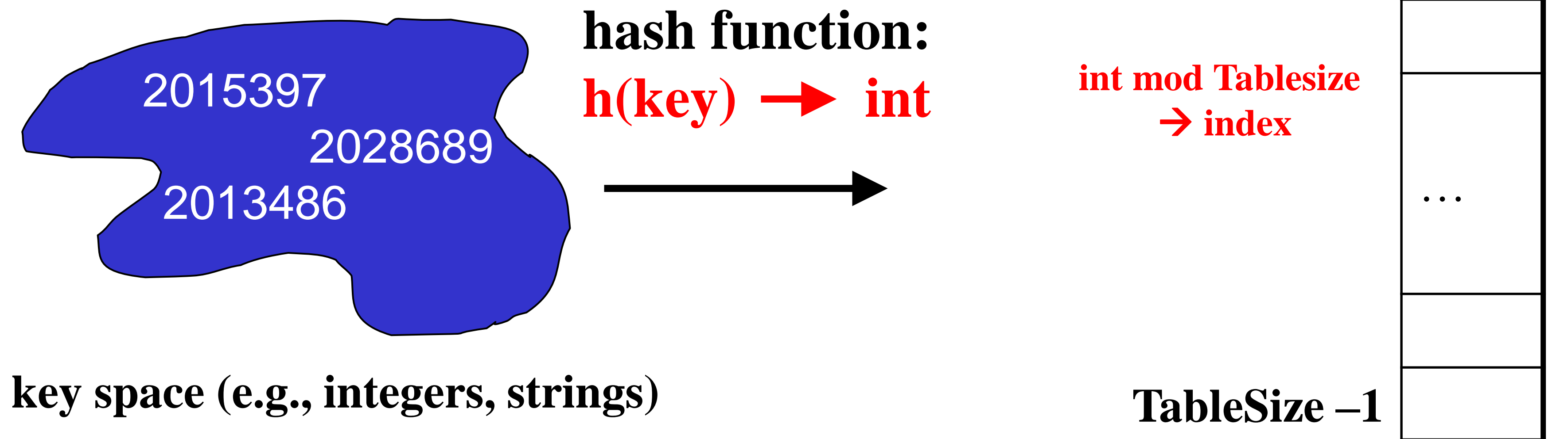
delete(4) - O(1)

n = 150

Hash Tables

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

- Basic idea: take a key -> int -> index

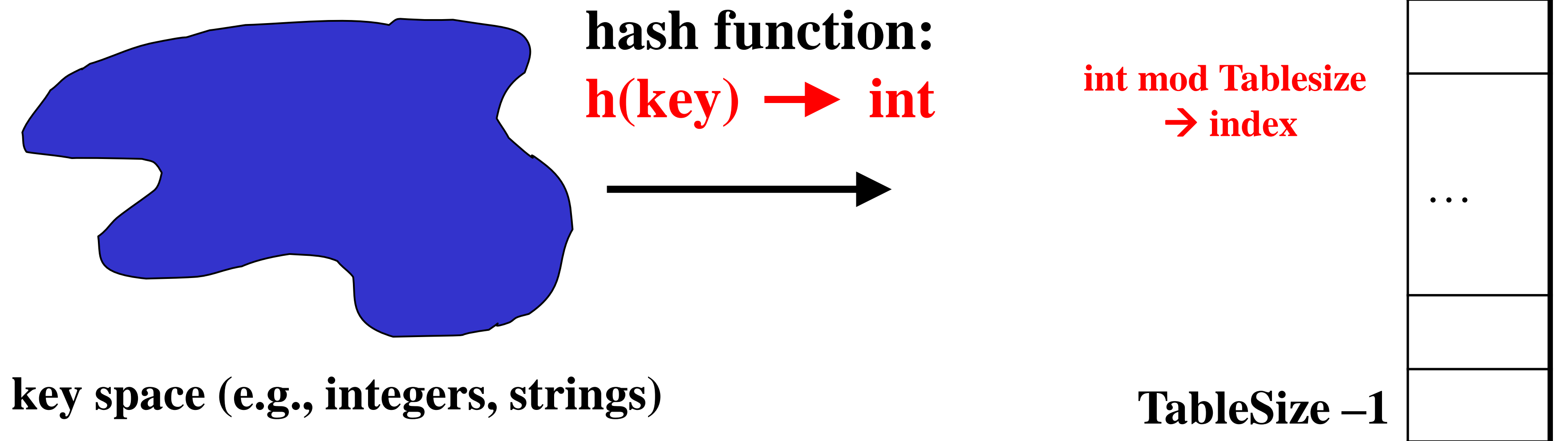


Hash Function Needs to Be Good!

h(s#) → 1st digit

An ideal hash function:

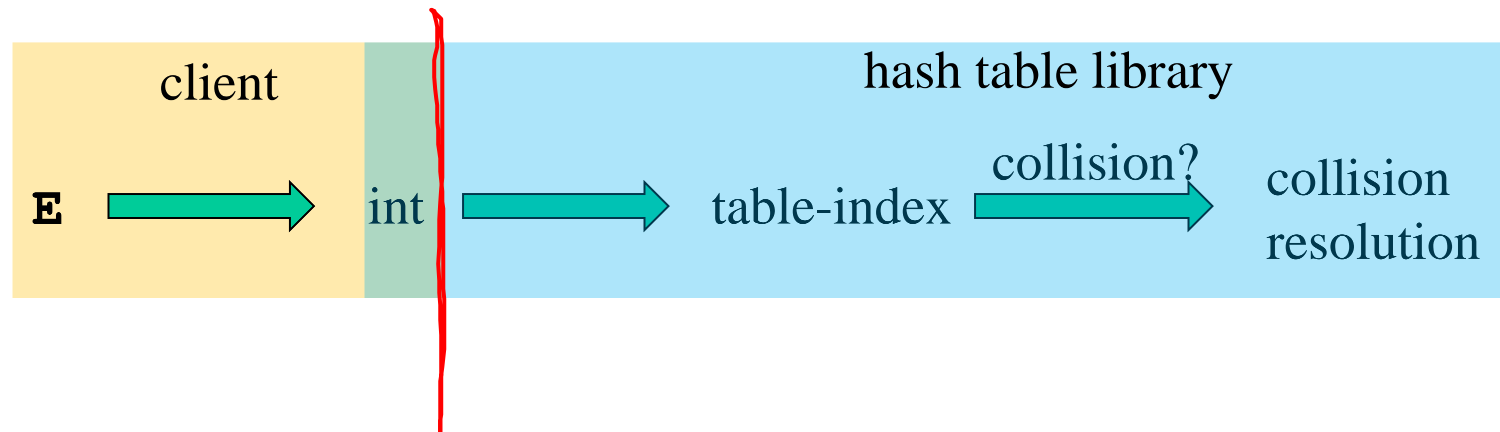
- Is fast to compute
- "Rarely" hashes two "used" keys to the same index
  - Often impossible in theory; easy in practice
  - Will handle *collisions* a bit later



Who's responsible for making it good?

- Clients write good hashcodes, so hash tables can be generic
  - To store keys of type **`E`**, we just need to be able to:
    1. Hashable: convert any **`E`** to an **`int`**
    2. Test equality: are you the **`E`** I'm looking for?

- When hash tables are a reusable library, the division of responsibility generally breaks down into two roles:



Ex: Java!

Constructors

Constructor and Description
`Object()`

Method Summary

Methods

Modifier and Type	Method and Description
`boolean`	`equals(Object obj)` Indicates whether some other object is "equal to" this one.
`int`	`hashCode()` Returns a hash code value for the object.

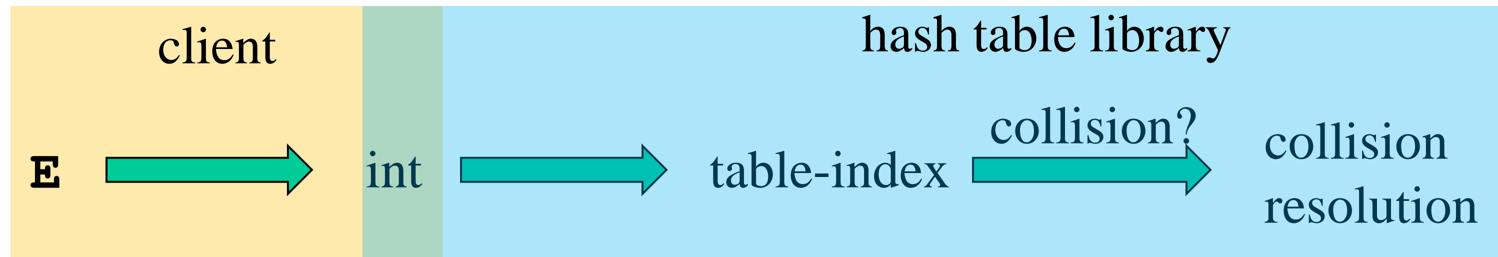
Class HashMap<K,V>

java.lang.Object
    java.util.AbstractMap<K,V>
        java.util.HashMap<K,V>

Type Parameters:

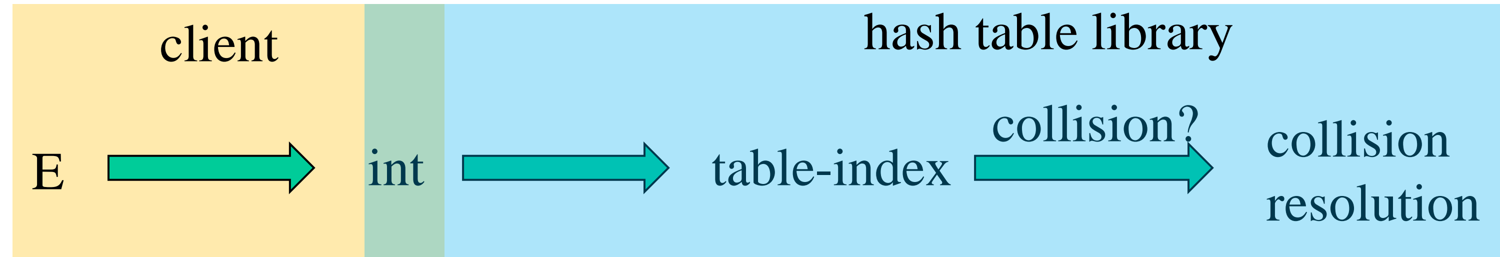
`K - the type of keys maintained by this map`

`V - the type of mapped values`



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

Each roles' responsibility to make it good



10,20

10

7

Two roles must both contribute to minimizing collisions (heuristically)

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

Hashing integers (try it out)

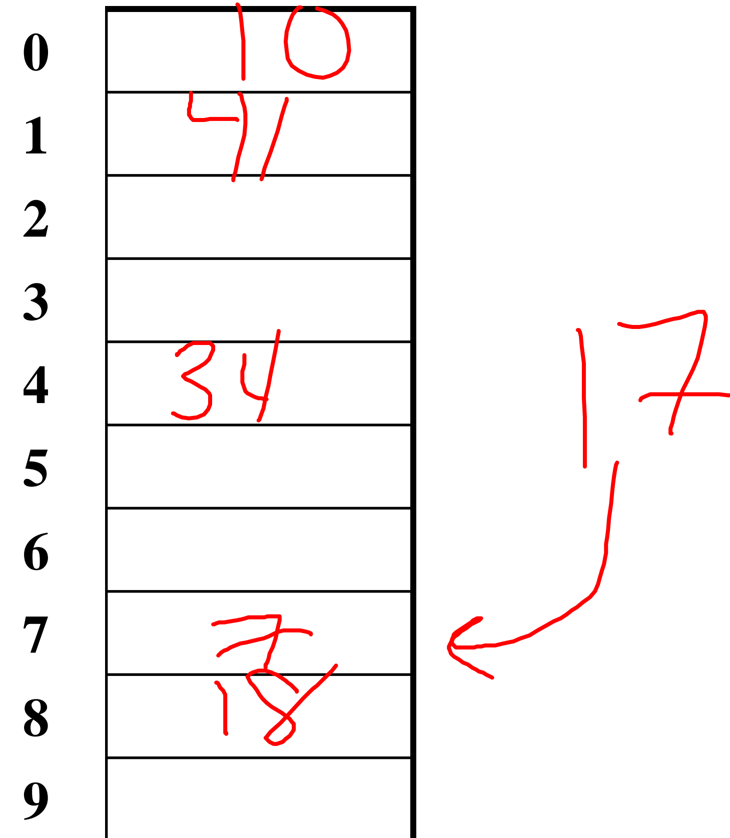
key space = integers

Simple hash function:

- Client: **`h(x) = x`**
- Library **`g(x) = h(x) % TableSize`**
- Fairly fast and natural

Example:

- TableSize = 10
- Insert **7, 18, 41, 34, 10**
- (As usual, ignoring corresponding data)



What to hash?

- 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;
}
```

4/10 → 366

- 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 ☹

What if the key is not an int?

- If keys aren't **`int`**s, the **client** must convert to an **`int`**
  - Trade-off: speed and distinct keys hashing to distinct **`int`**s

- Common and important example: Strings
  - Key space K $= s_0 s_1 s_2 \ldots s_{m-1}$
    - where s_i are chars: $s_i \in [0,256]$
  - Some choices: Which avoid collisions best? What strings would collide?

1. $h(K) = s_0$

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

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

ab ba

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

Calculation tricks

- Avoid heavy computation by using tricks!

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

```
String s;
h = 1;
for (int i = k - 1; i >= 0; i--) {
  h = 31 * h + s[i];
}
```

Specializing hash functions

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

http //

.com

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*

Okay so collisions happen…

key space = integers

Simple hash function:

- Client: `h(x) = x`
- Library `g(x) = h(x) % TableSize`
- Fairly fast and natural

Example:

- TableSize = 10
- Insert 7, 18, 41, 34, 10
- (As usual, ignoring corresponding data)

0	10
1	41
2	
3	
4	34
5	
6	
7	7
8	18
9	

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?

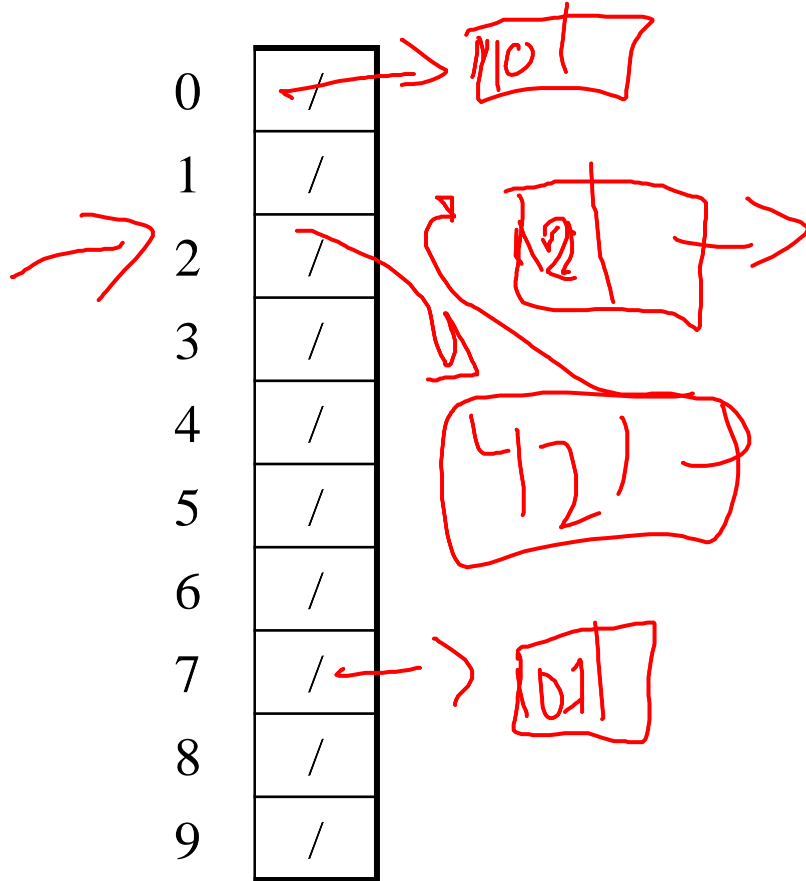
Flavors of Collision Resolution

Separate Chaining

Open Addressing

- Linear Probing
- Quadratic Probing
- Double Hashing

Separate Chaining

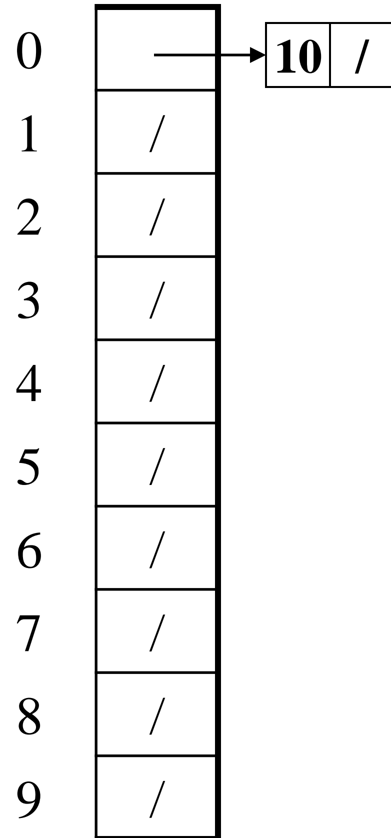


Chaining: All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

As easy as it sounds

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

Separate Chaining

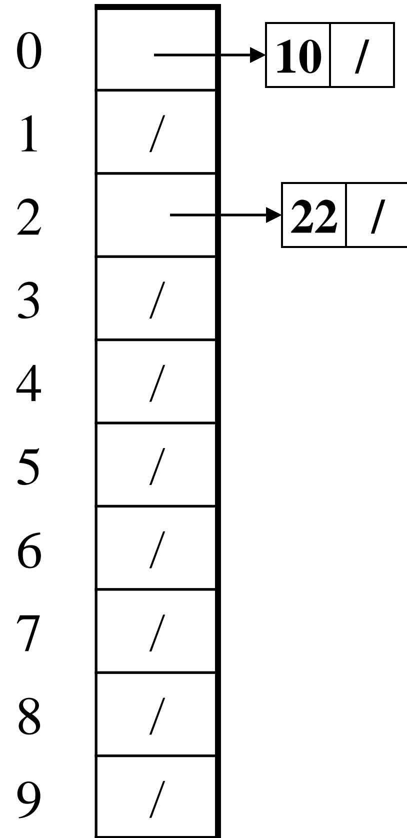


Chaining: All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

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

Separate Chaining

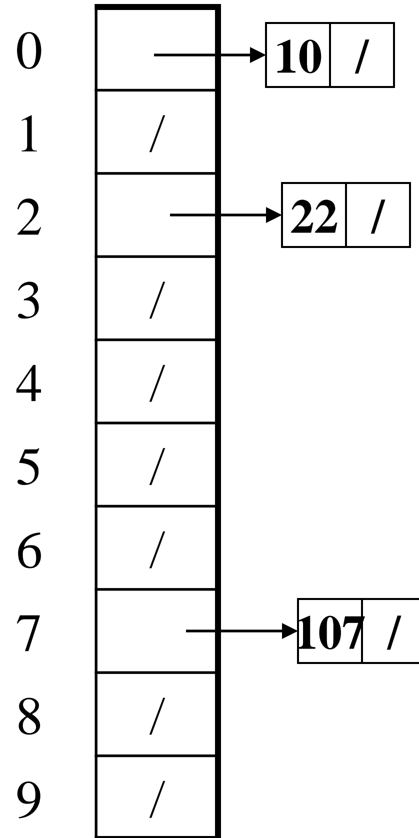


Chaining: All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

As easy as it sounds

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

Separate Chaining

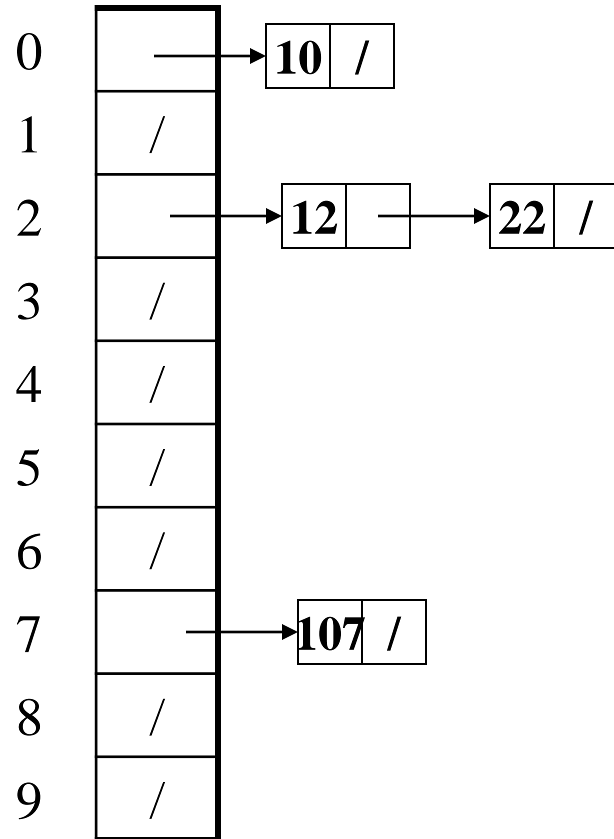


Chaining: All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

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

Separate Chaining

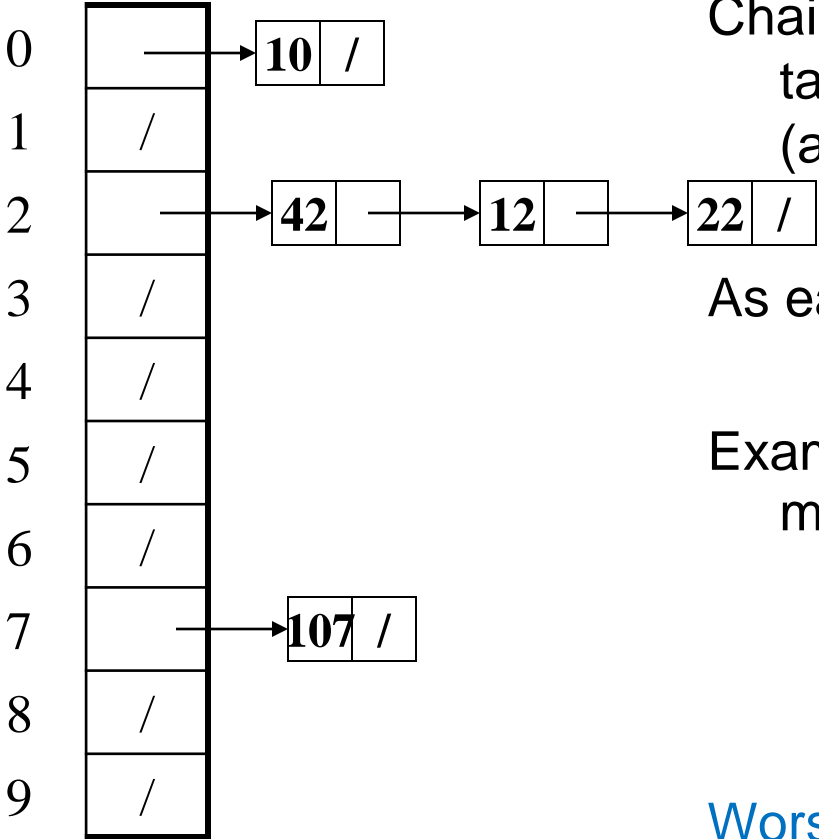


Chaining: All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

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`**?

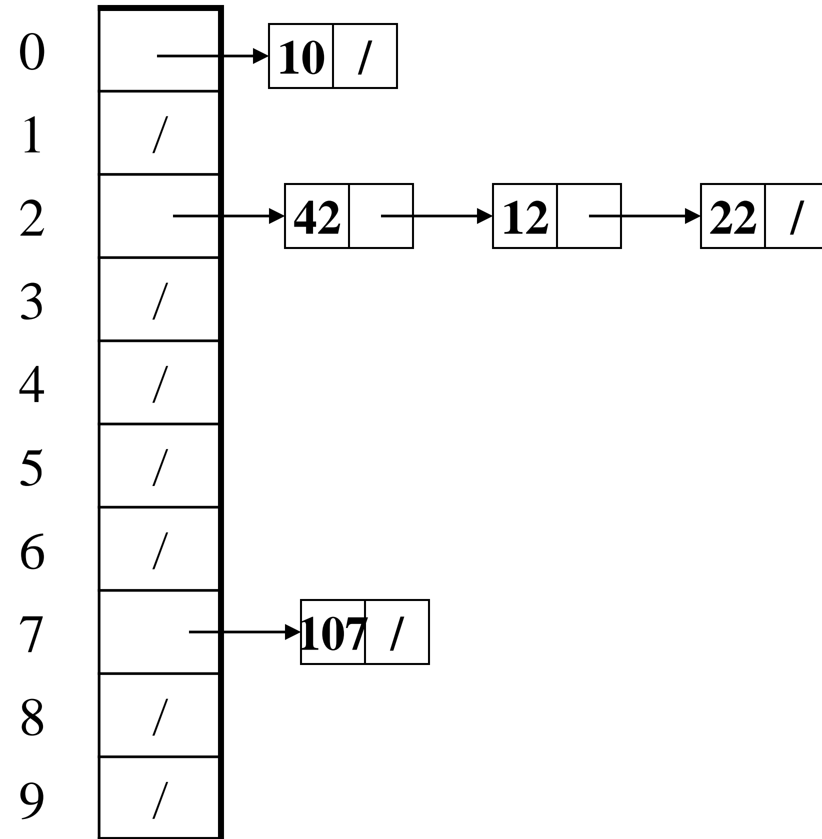
- Linear
- But only with really bad luck or bad hash function
- So not worth avoiding (e.g., with balanced trees at each bucket)
  - Keep # of items in each bucket small
  - Overhead of AVL tree, etc. not worth it if small # items per bucket

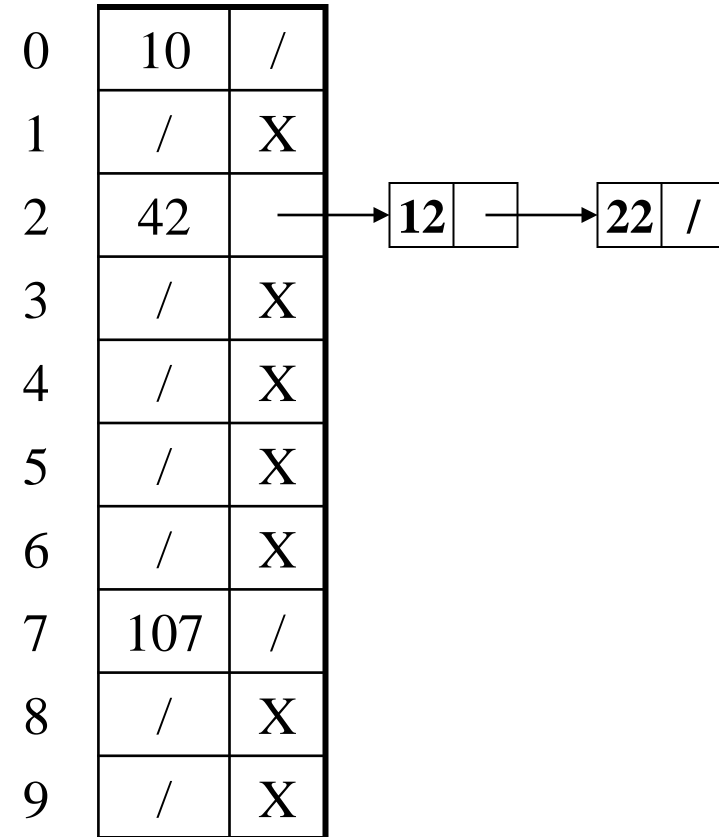
Beyond asymptotic complexity, some “data-structure engineering” can improve constant factors

- 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, λ, of a hash table is

$$\lambda = \frac{N}{\text{TableSize}}$$ **← number of elements**

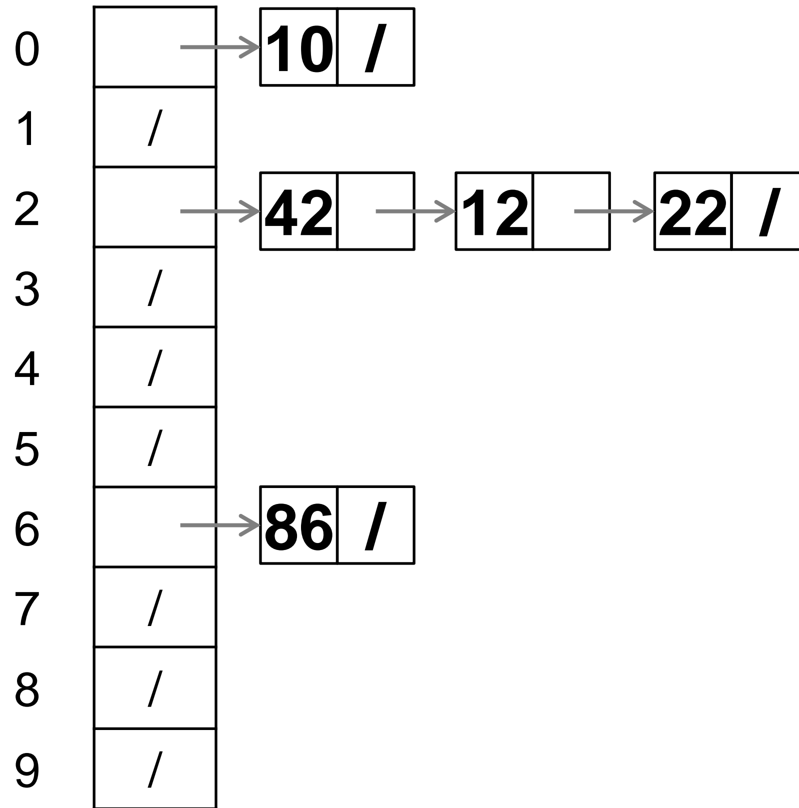
Under chaining, the average number of elements per bucket is ___

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??

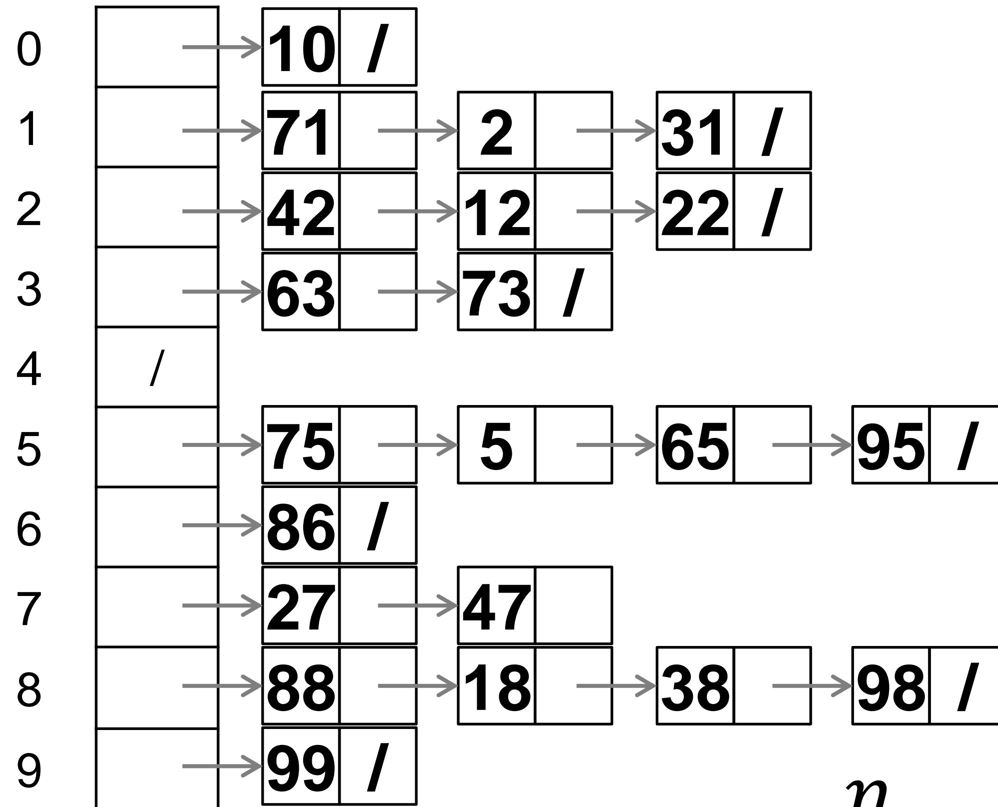
Load Factor?



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

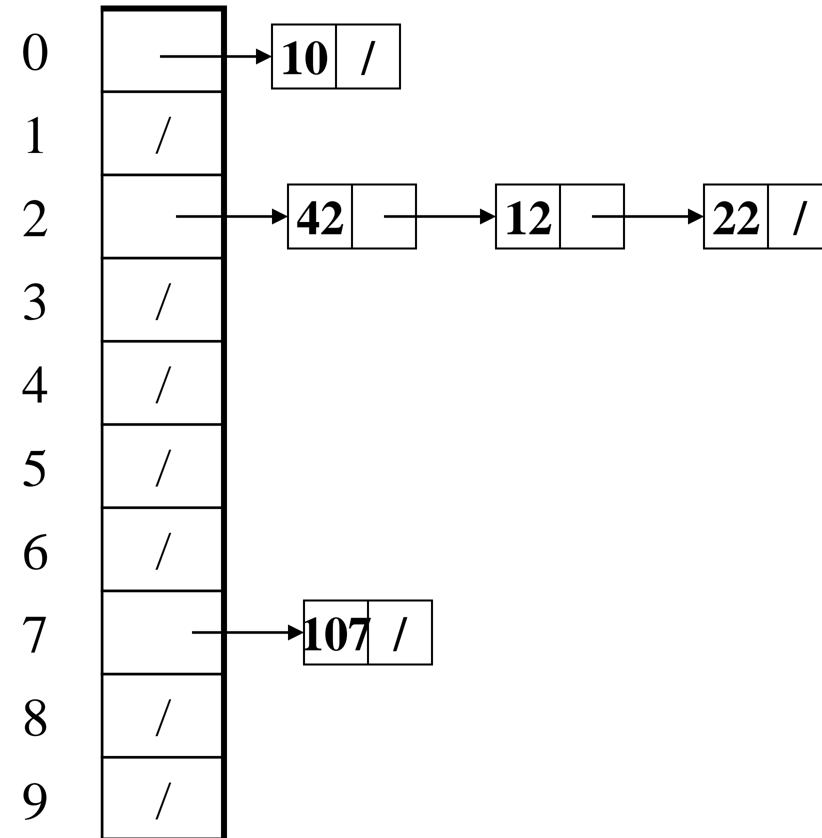
$\frac{5}{10} = 0.5$

Load Factor?



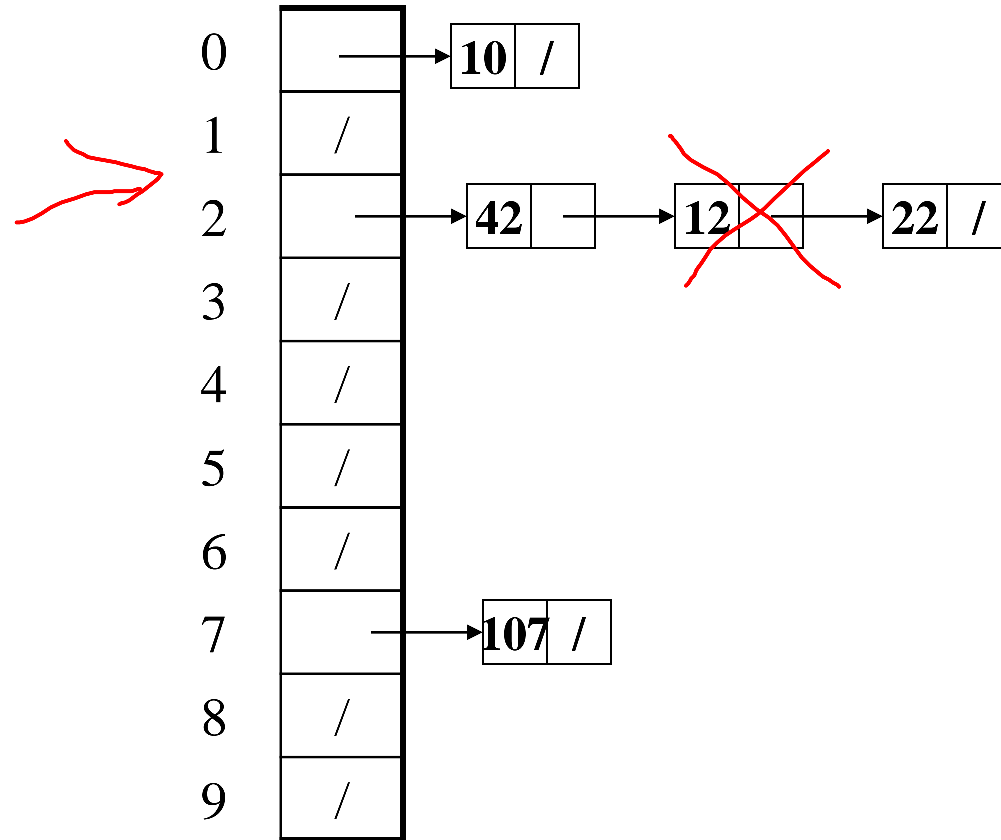
$$\lambda = \frac{n}{TableSize} = ?$$

Separate Chaining Deletion?



Separate Chaining Deletion

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



Motivating Hash Tables

For dictionary with n key/value pairs

	`insert`	**`find`**	**`delete`**	
• 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	$O(n)$	$O(\log n)$	$O(n)$	
• *Balanced* tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	
• ***HashTables***	**$O(1)$**	**$O(1)$**	**$O(1)$**	**(average)**

* 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).

Why Hash Tables are a great approximation of our Really Big Array

Not that many elements that we need to store

- 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
- …

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

Wrapup

- How hashing works
- Guidelines for good hashcodes
- Intro to collision handling

- Next time:
  - 3 flavors of open addressing (collision handling)
  - More hashing in practice