CSE 332: Data Structures & Parallelism

Lecture 9:Hashing

Yafqa Khan

Summer 2025

Today

- Dictionaries
  - Hashing

Annoncements:

- EX03 due Today
- EX04 released today
- Exam 1 week 5 Friday

Motivating Hash Tables

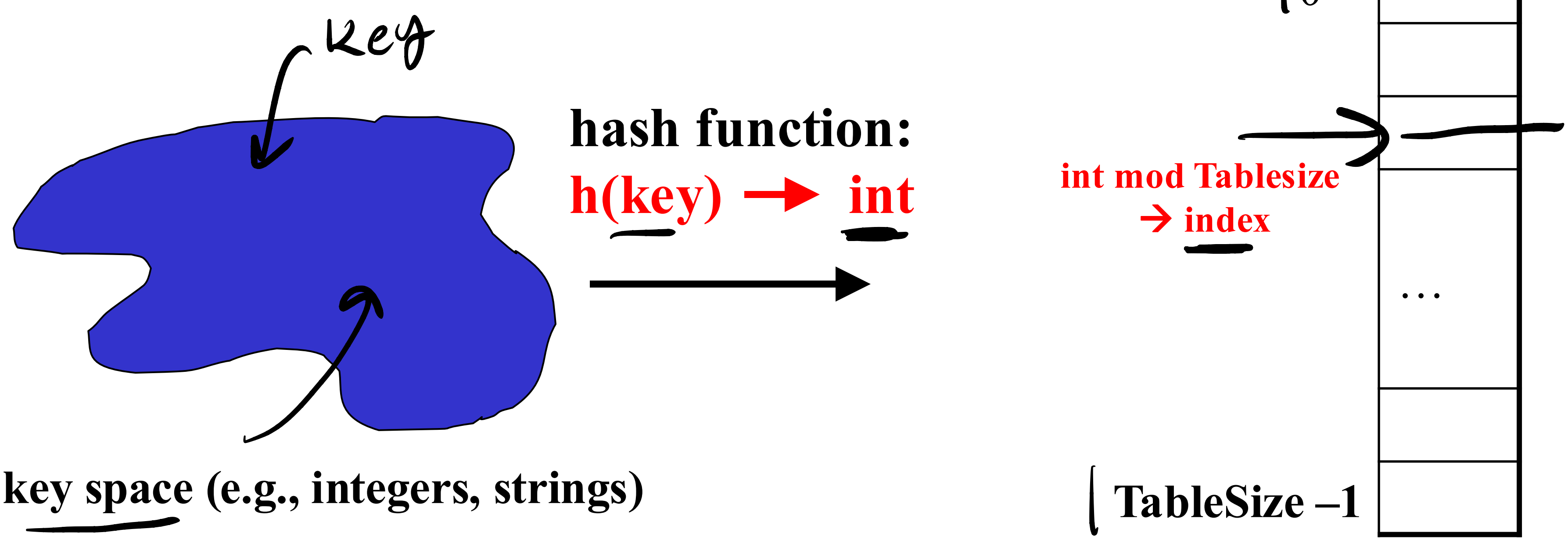
For dictionary with *n* key/value pairs

	`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).

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* << *m*)

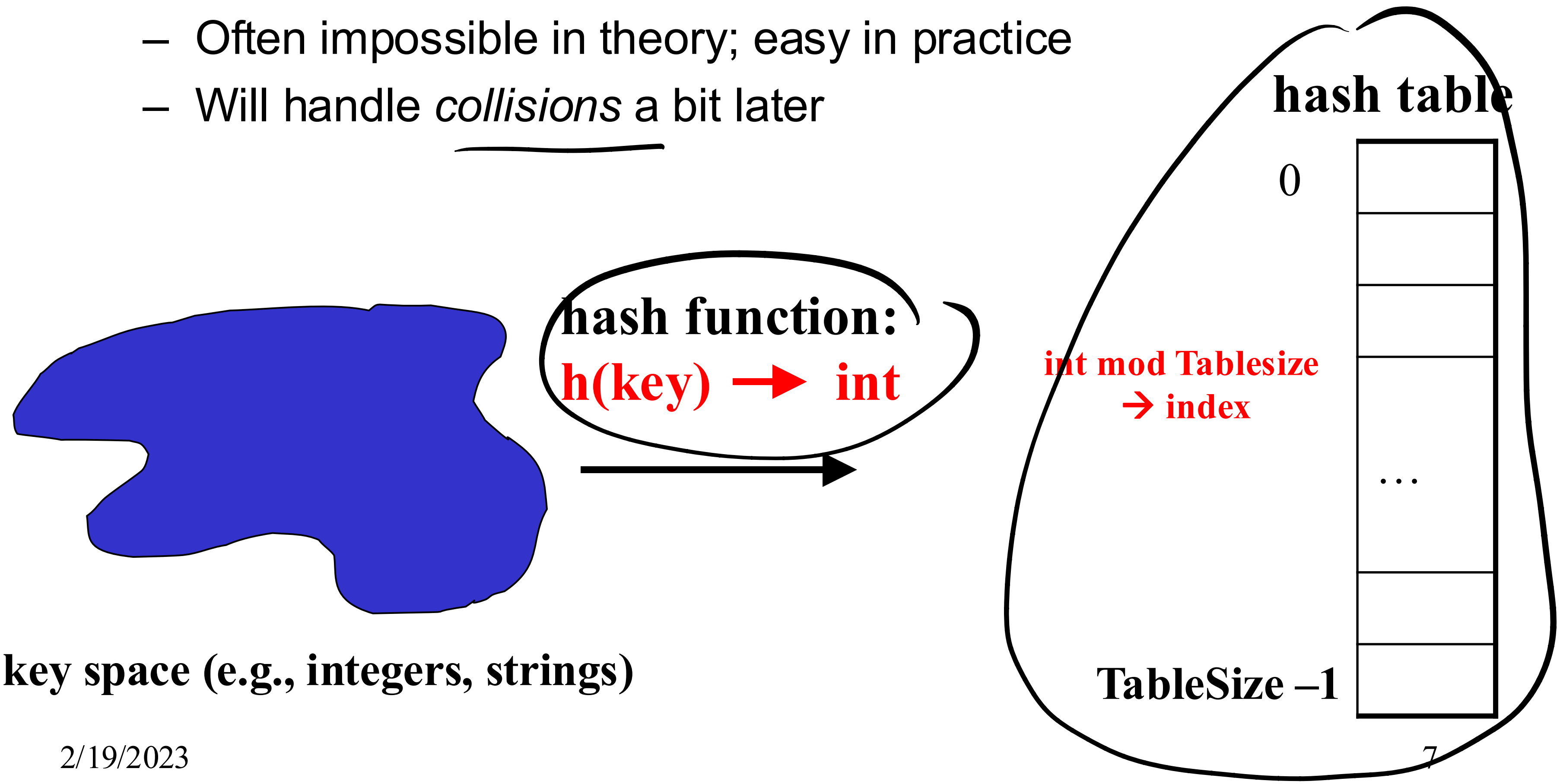
Many dictionaries have this property

- 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:

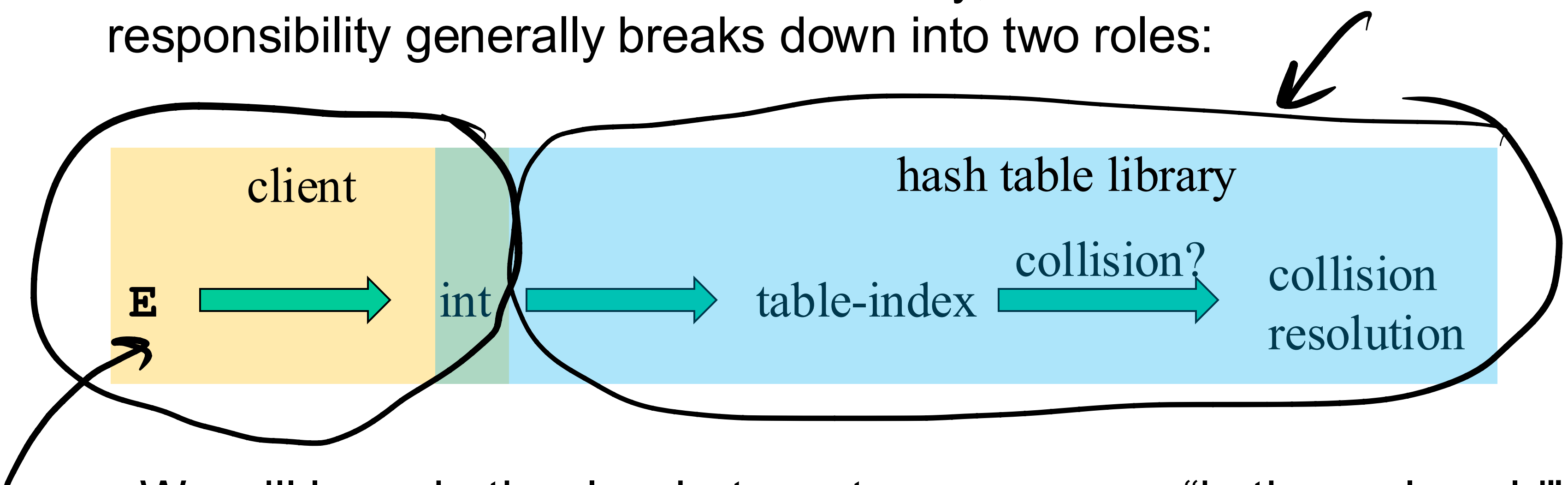
- Is fast to compute
- "Rarely" hashes two "used" keys to the same index
  - Often impossible in theory; easy in practice
  - Will handle *collisions* a bit later



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`**

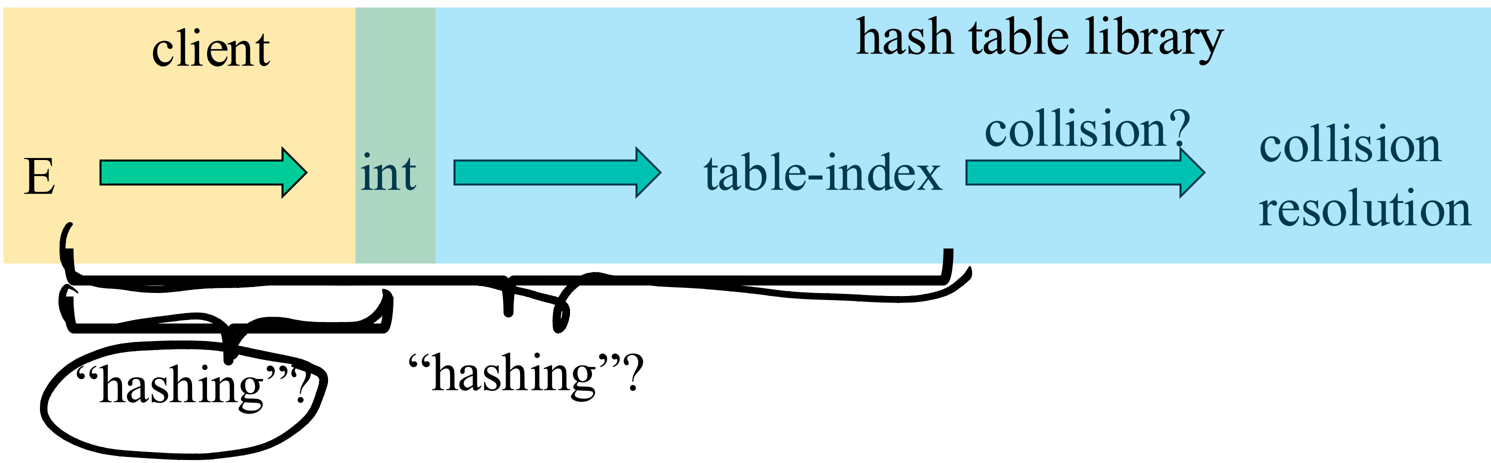
- When hash tables are a reusable library, the division of responsibility generally breaks down into two roles:



- 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” **`int`s** 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:

- 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)

h(7) = 7

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

Hashing integers (Soln)

key space = integers

Simple hash function:

- Client: **`h(x) = x`**
- Library **`g(x) = f(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-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 **`a`** and **`p`** are "co-prime" (means **`gcd(a,p)==1`**), then

`ax = b (mod p) will always have a solution`

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

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 = $s_0 s_1 s_2 \ldots s_{m-1}$
    - where s_i are chars: $s_i \in [0,256]$
  - Some choices: Which avoid collisions best?

1. $h(K) = s_0$ → a, ab, aa, b

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

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

3. $h(K) = \left(\sum_{i=0}^{m-1} s_i \cdot 37^i \right)$ → a b c — 37^0 37^1 37^2

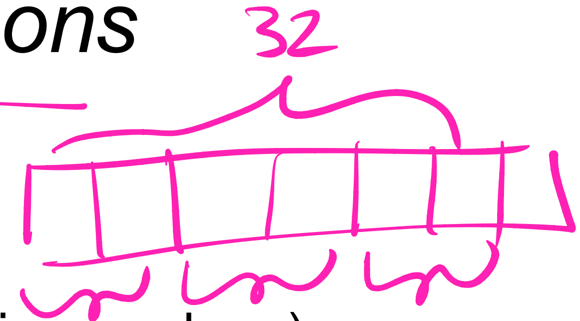
Java Implementation of ***String.hashCode***

```java
public static int hashCode(byte[] value) {
    int h = 0;
    int length = value.length >> 1;
    for (int i = 0; i < length; i++) {
        h = 31 * h + getChar(value, i);
    }
    return h;
}
```

$$(((31 s_0) + s_1) 31 + s_2) 31 + s_3$$

$$= \sum (31^i) s_{n-1-i}$$

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

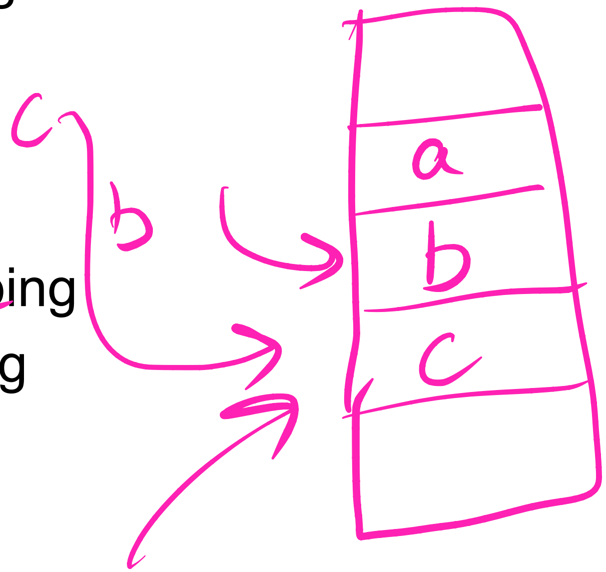
– Ideas?

Flavors of Collision Resolution

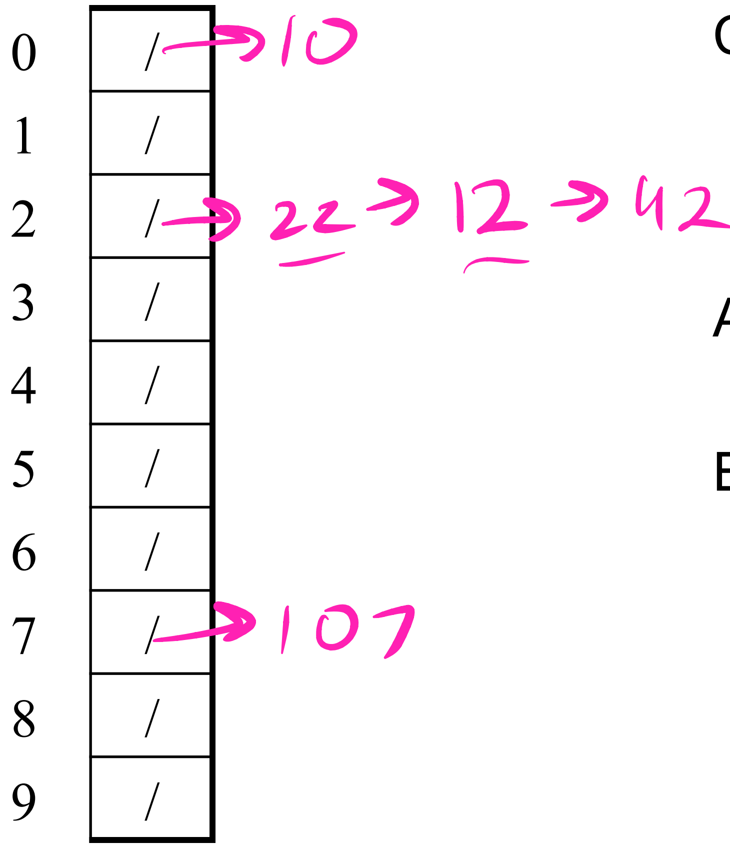
Separate Chaining

Open Addressing

- Linear Probing
- Quadratic Probing
- Double Hashing



Separate Chaining

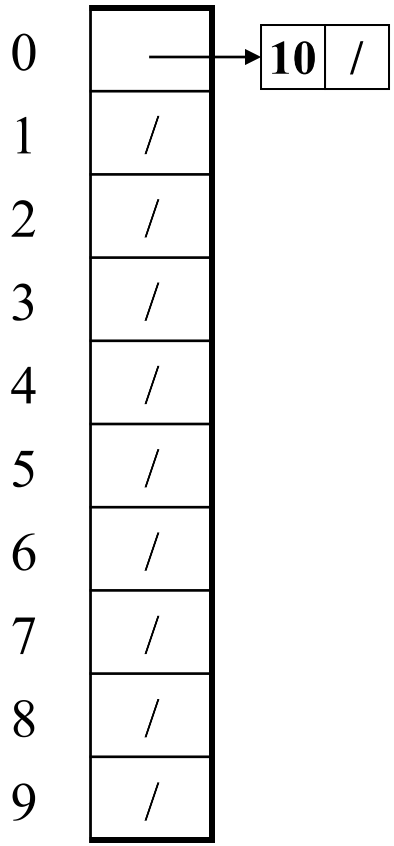


Chaining: All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

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

Separate Chaining

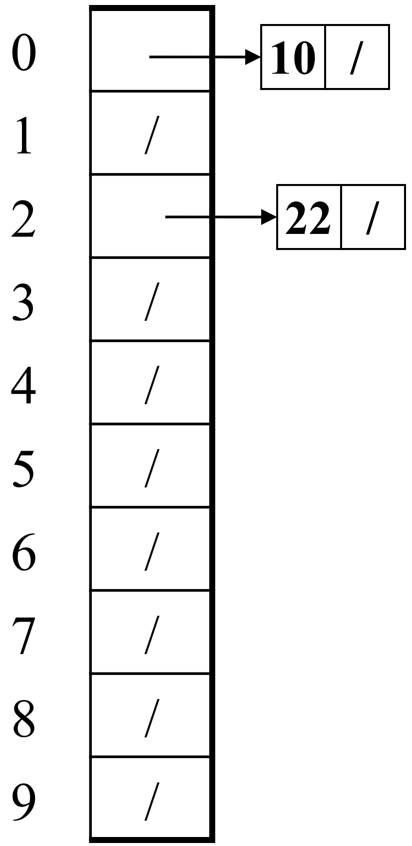


Chaining: All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

As easy as it sounds

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

Separate Chaining

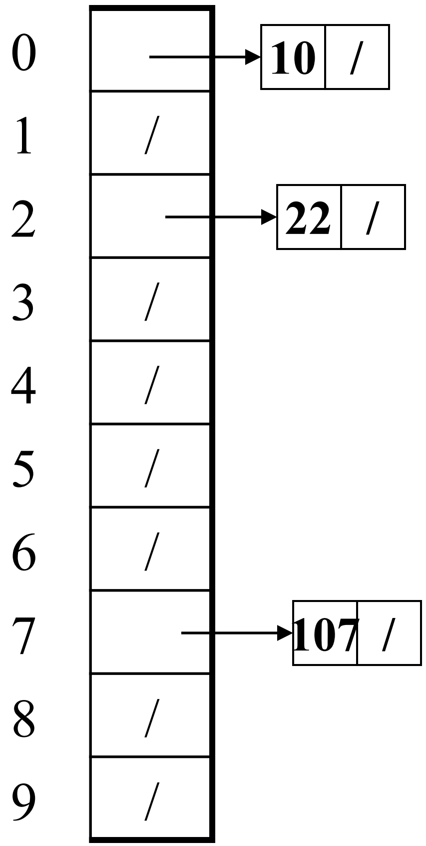


Chaining: All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

As easy as it sounds

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

Separate Chaining

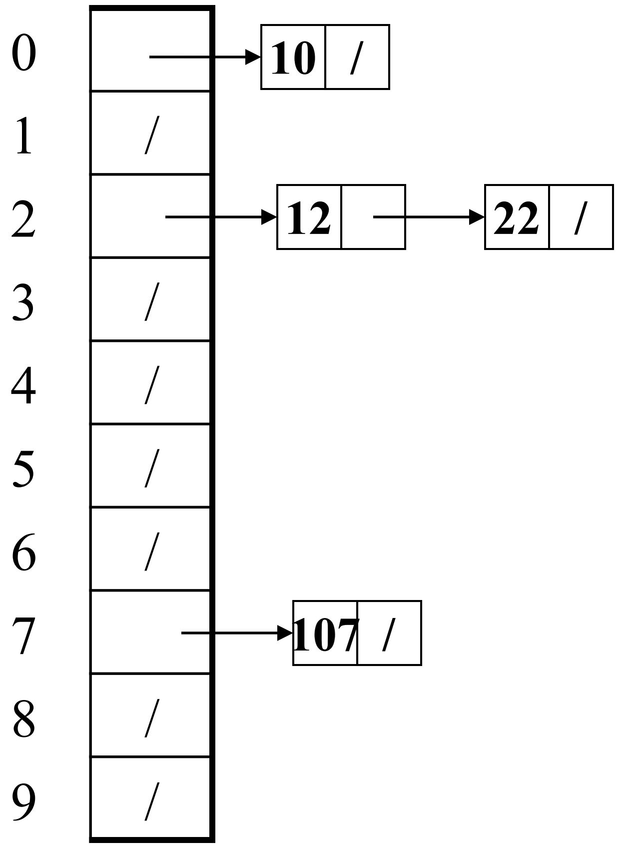


Chaining: All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

As easy as it sounds

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

Separate Chaining

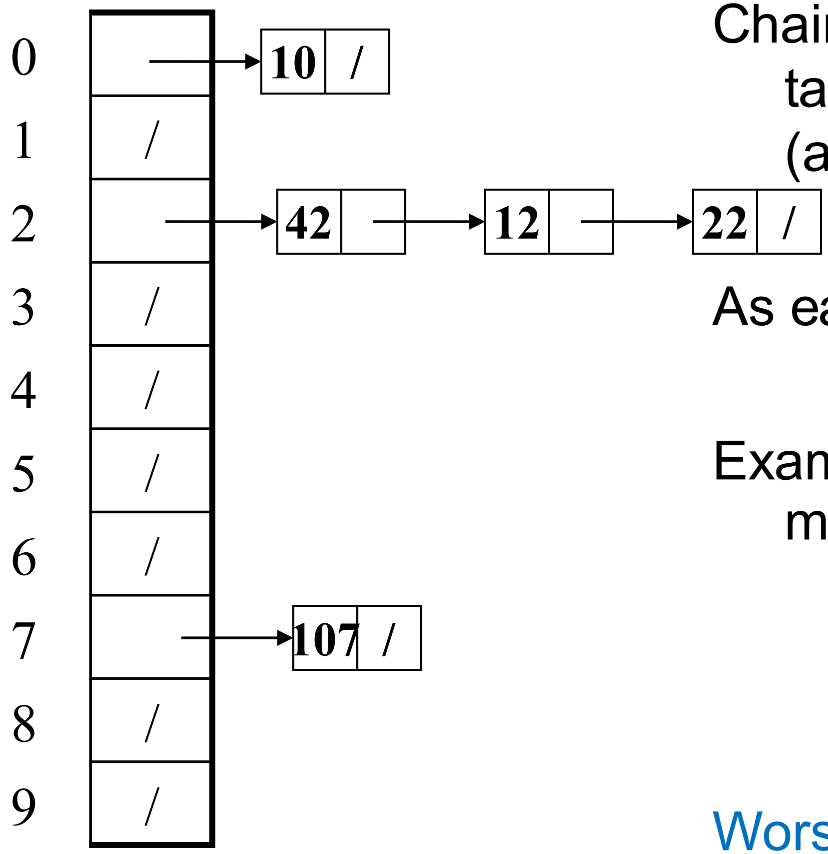


Chaining: All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

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?

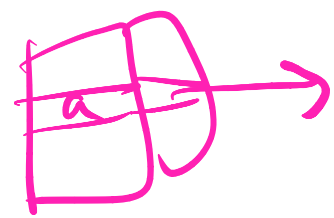
Thoughts on separate chaining

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

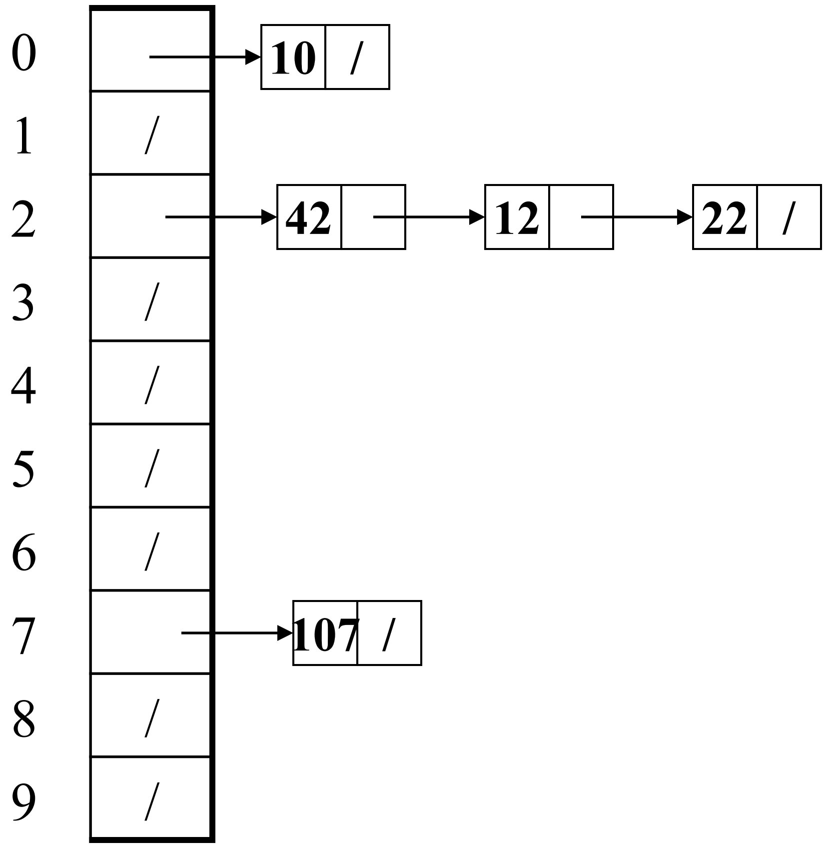
Beyond asymptotic complexity, some "data-structure engineering" can improve constant factors

- 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…

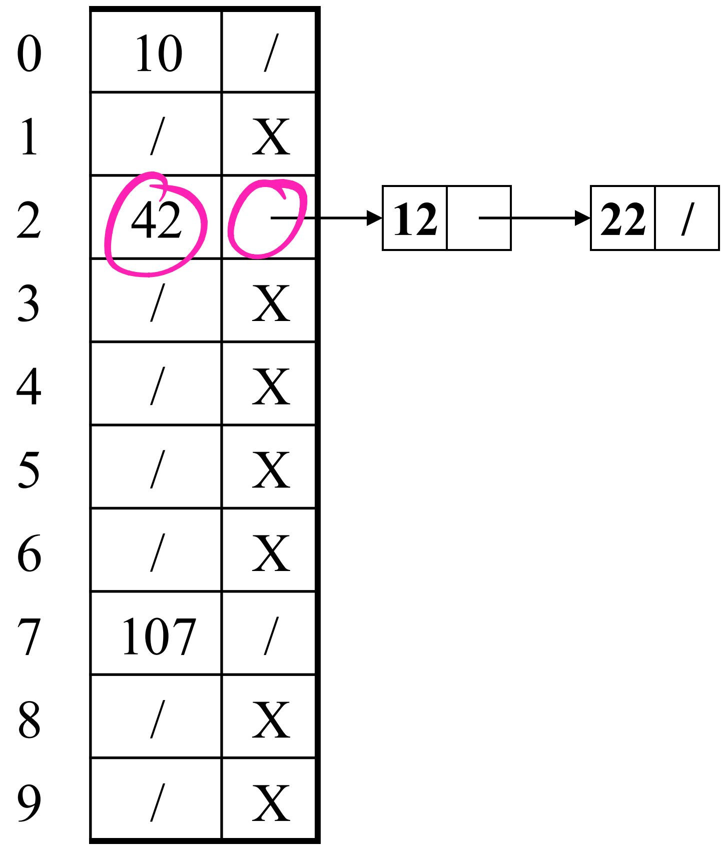


Time vs. space

(only makes a difference in constant factors)



more time



more space

More rigorous separate chaining analysis

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

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

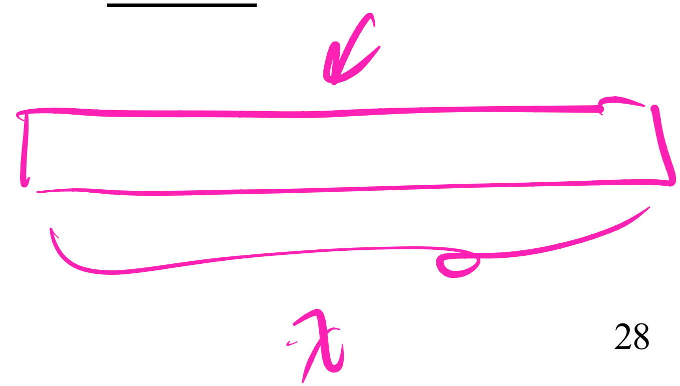
Under chaining, the average number of elements per bucket is ___ $\lambda \leq 3.0$

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

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

- How big should TableSize be??

$\lambda \leq 3.0$



More rigorous separate chaining analysis

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

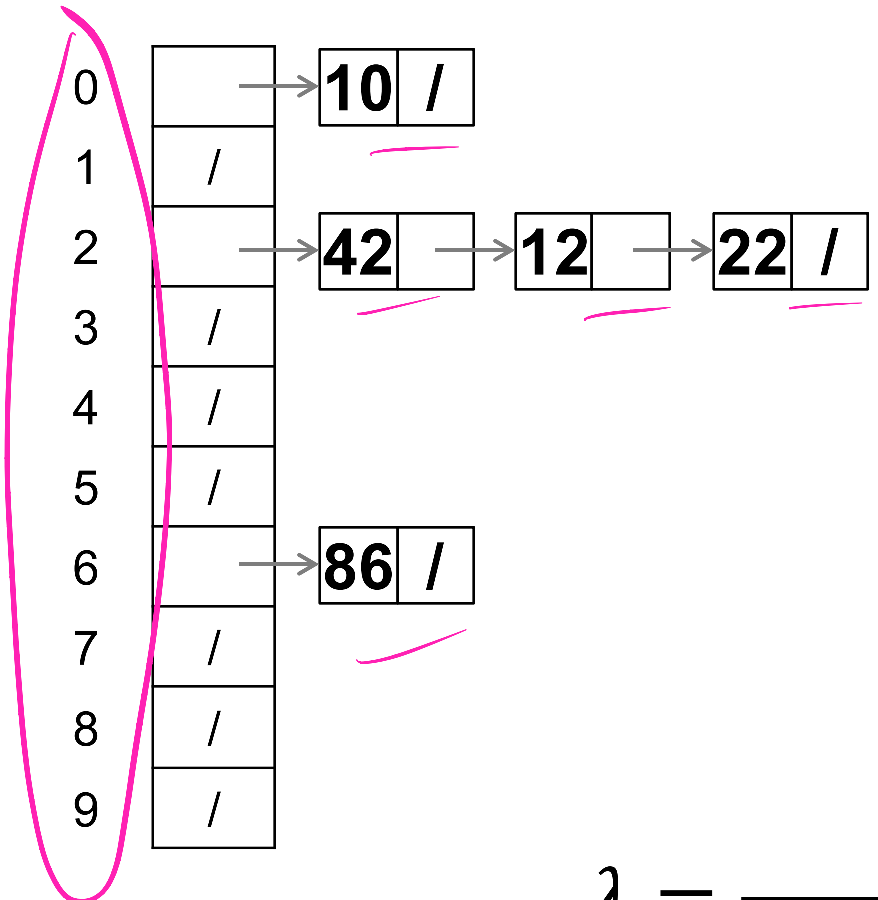
$$\lambda = \frac{\text{N}}{\text{TableSize}} \quad \leftarrow \textbf{number of elements}$$

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
- Each successful **`find`** compares against $\lambda / 2$ items
- If λ is low, find & insert likely to be O(1)
- We like to keep λ around 1 for separate chaining

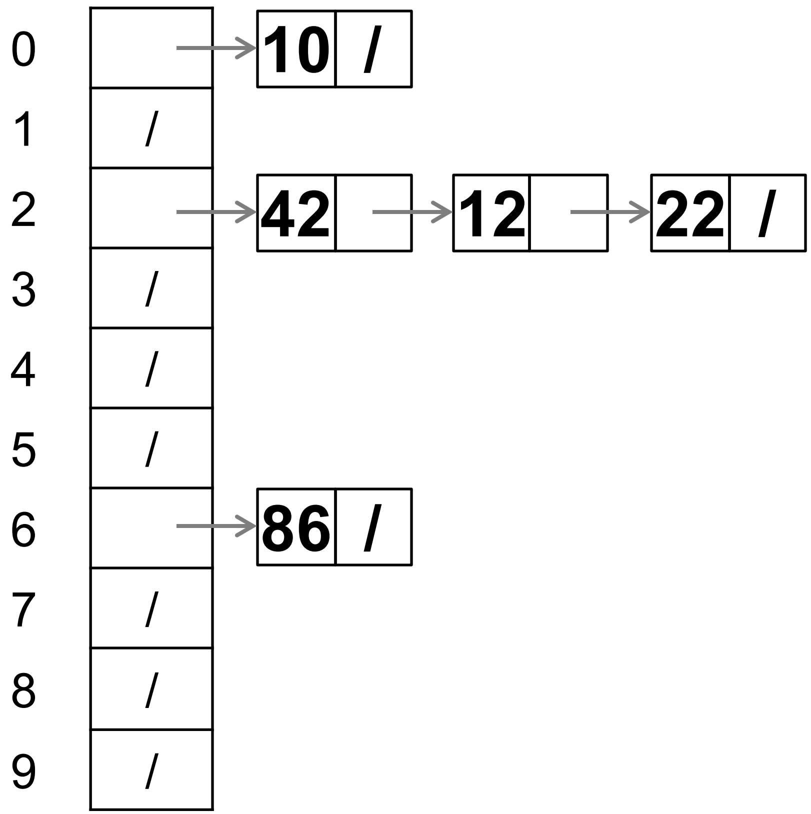
Load Factor?



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

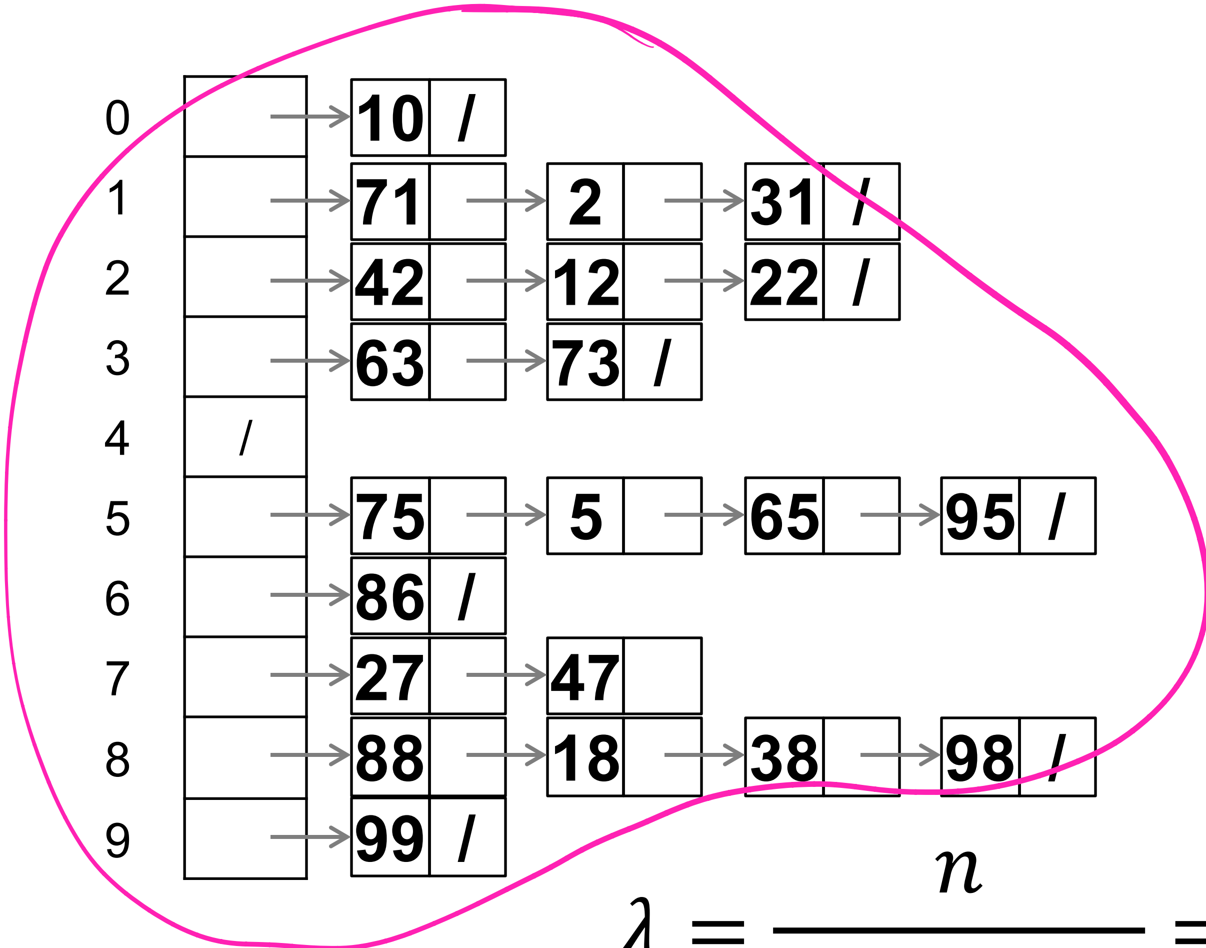
$\frac{5}{10} = 0.5$

Load Factor?



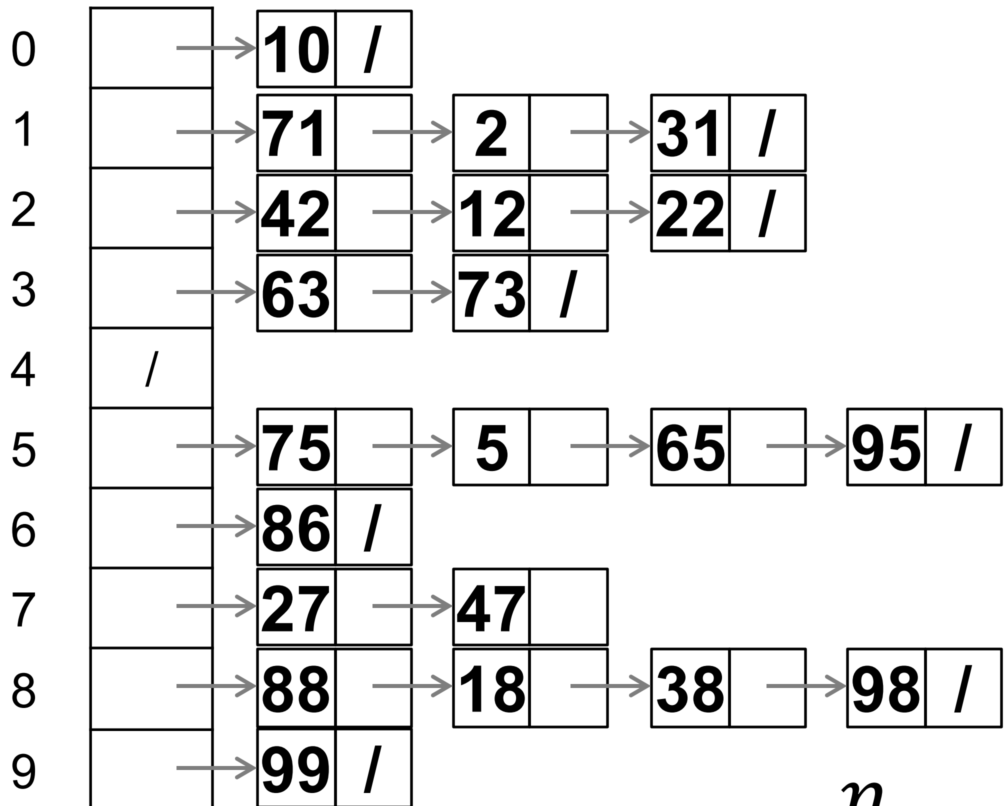
$$\lambda = \frac{n}{TableSize} = \frac{5}{10} = 0.5$$

Load Factor?



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

Load Factor?

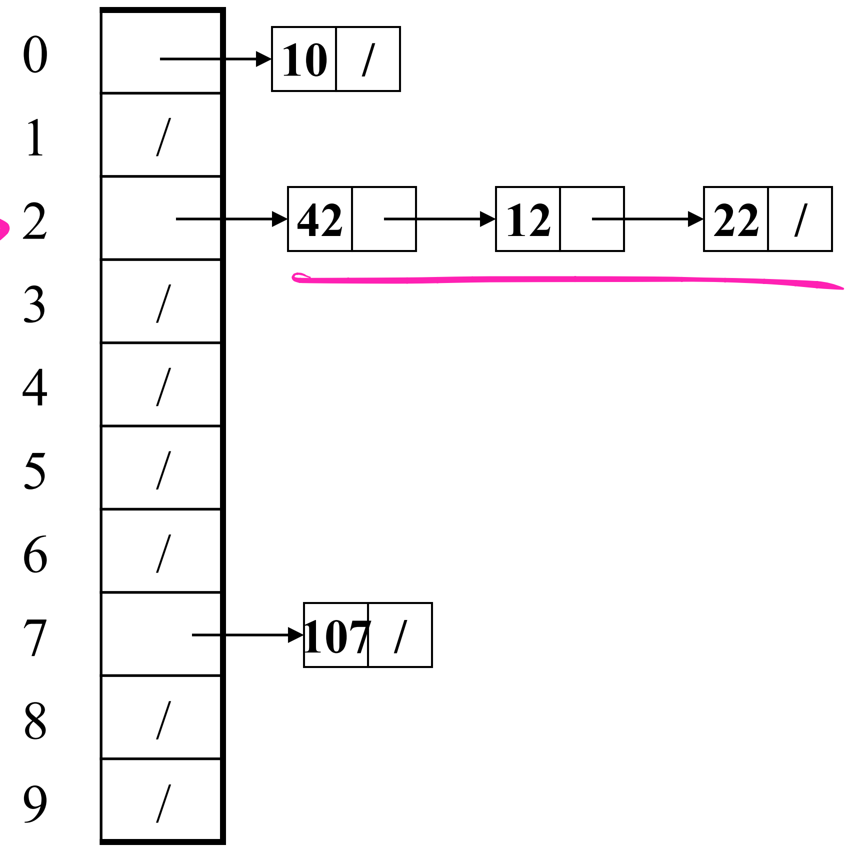


$$\lambda = \frac{n}{TableSize} = \frac{21}{10} = 2.1$$

Separate Chaining Deletion?

Separate Chaining Deletion

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



h(k)% tablesize

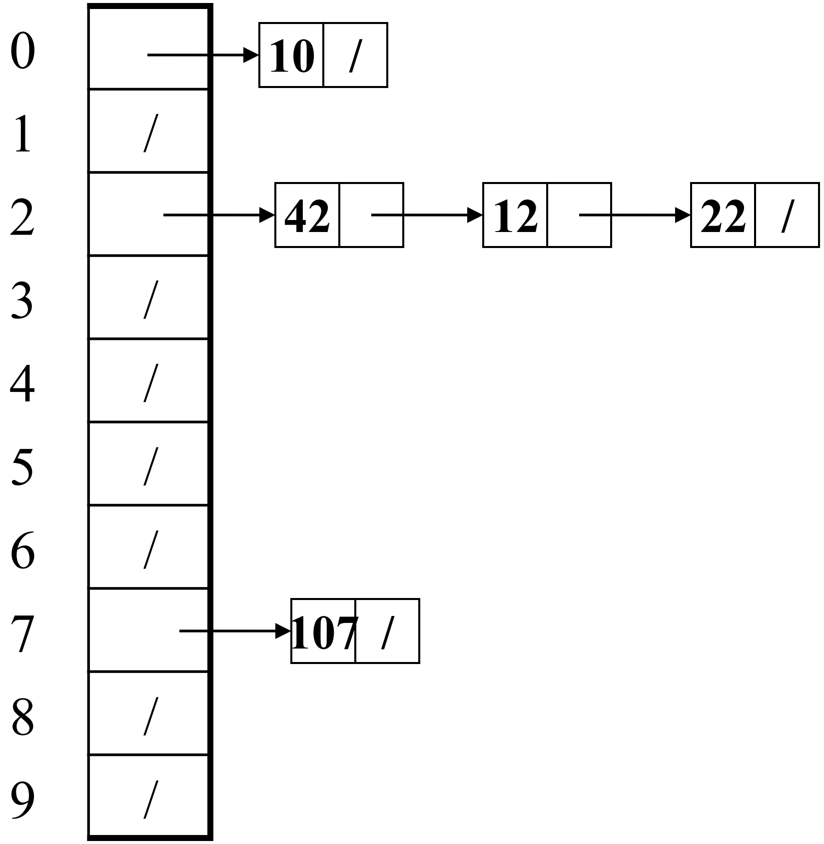
O(n)

O(λ) ⇒ O(1)

≤ 360

Separate Chaining Deletion

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

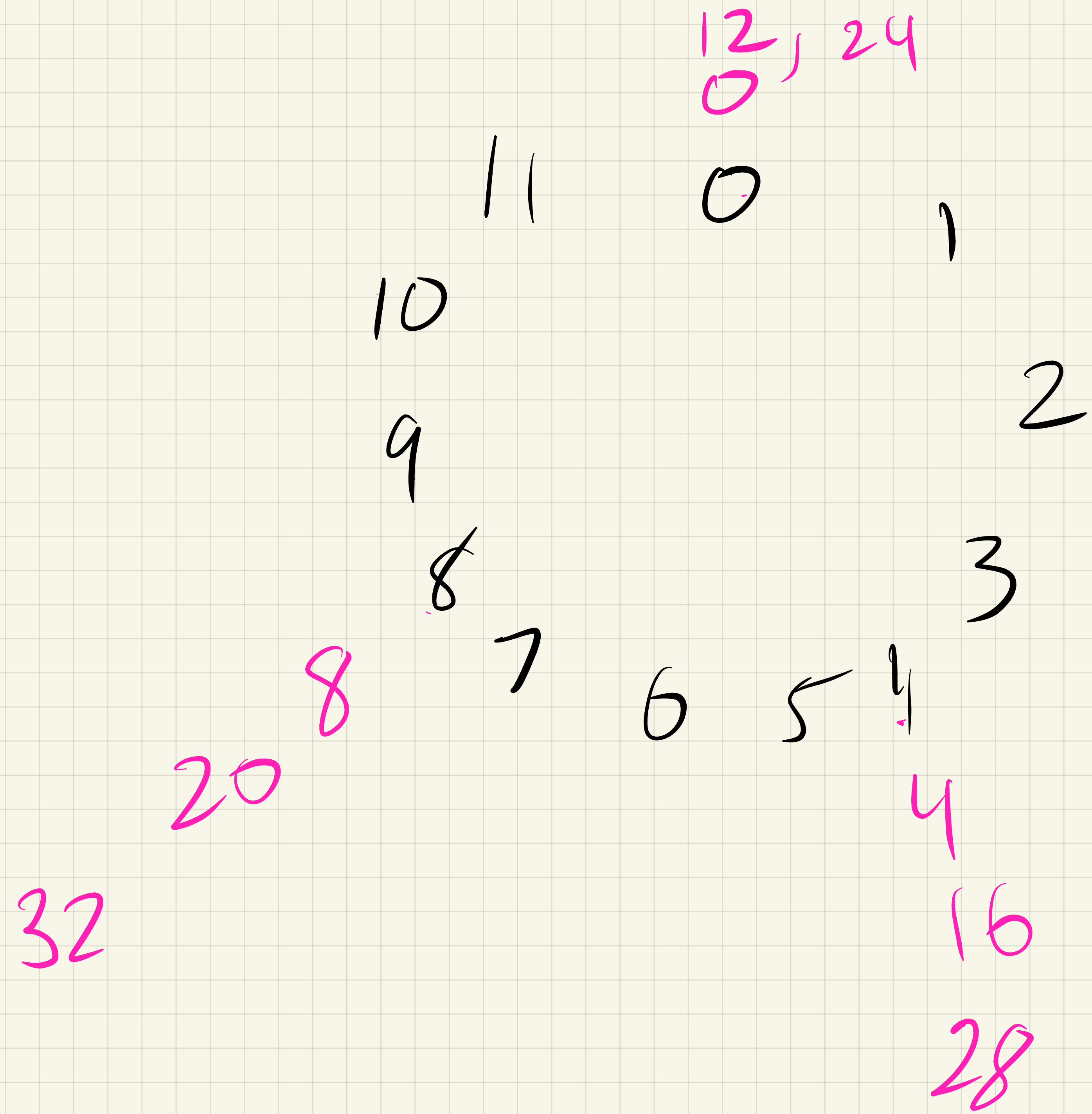


Example:

- Seperate Chaining hash table
- $h(x) = 4x$ (key space is integers)
- initial table size of 12
- (resize by doubling table size)

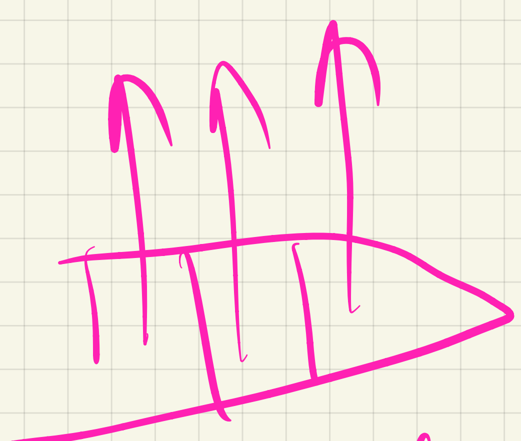
- What's wrong w/ this hash table?

5 min



$h(x) = 4x$

- only 25% of indices ever used



effectively only 3 bukets – lots of collisions!

insert into AVL:

~~1~~, ~~8~~, ~~6~~, ~~3~~, ~~7~~, ~~8~~, ~~9~~