CSE 332
Data Structures & Parallelism

Hashing 1

Melissa Winstanley
Spring 2024

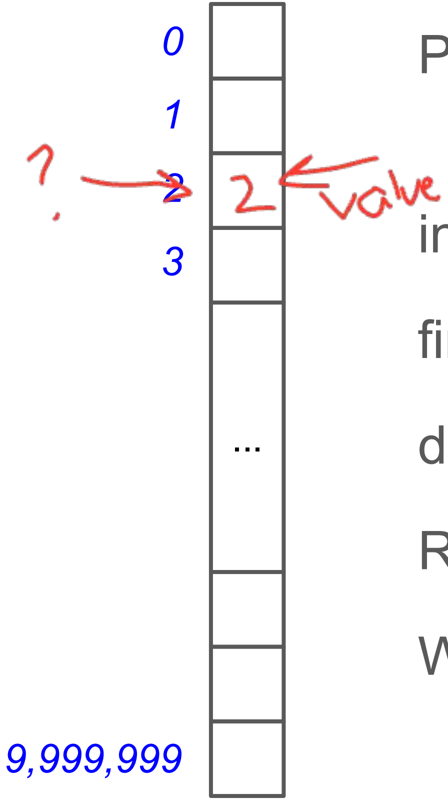
Motivating Hash Tables

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).

Idea: $\boxed{\text{“Big Array”}}$



Possible keys = student #

0 9,999,999

insert(2)

find(2)

delete(2)

Runtime of these operations? O(1)

Why haven’t we been using this?

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)
• **Hash table**	**O(1)****	**O(1)**	**O(1)**

**** Average complexity**

Hash Tables

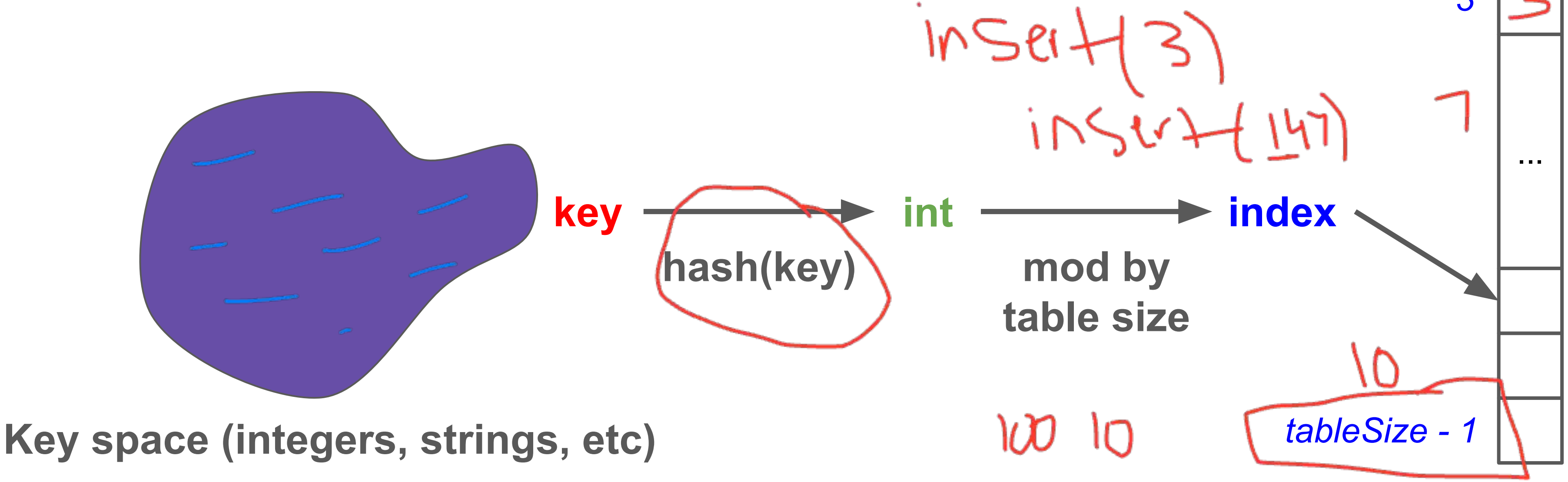
- m = possible keys (e.g. possible student no., 9,999,999)
- n = no. of expected keys (e.g. total students, 180 in CSE332)
  - We expect our table to have only n items
  - n is much less than m (often written $\boldsymbol{n \ll m}$)

Many dictionaries have this property

- Compiler: variable names in a file << possible variable names
- Database: enrolled student names << possible student names
- AI: Chess-board configurations considered by the current player vs. All possible chess-board configurations

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:



Hash Tables vs Balanced Trees

- Both implement the Dictionary ADT:
  - **`find`**, **`delete`**, **`insert`**
  - Hash tables O(1) on average (assuming few collisions)
  - Balanced trees O(log n) worst-case
- Constant-time is better, right?
  - Yes, but what if we want to **`findMin`**, **`findMax`**, **`predecessor`**, **`successor`**, **`printSorted`**?
  - Hashtables are not designed to efficiently implement these sortedness operations
- Your textbook considers hash tables to be a different ADT
  - Not so important to argue over the definitions

Hash Functions

An ideal hash function:

- Is **fast** to compute $O(n)$ $O(\log n) < O(1)$
- “Rarely” hashes two “used” keys to the **same index**
    - Often impossible in theory; easy in practice
    - Will handle collisions a bit later

What would be the hash function $\boxed{\text{signature}}$ if our keys are student #s?

int → int

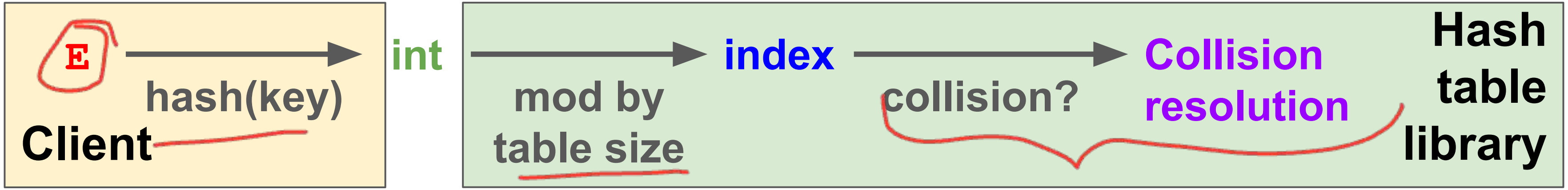
What would a **bad** hash function if our keys are student #s?

- 1st # X

Who hashes what?

Hash tables can be generic.

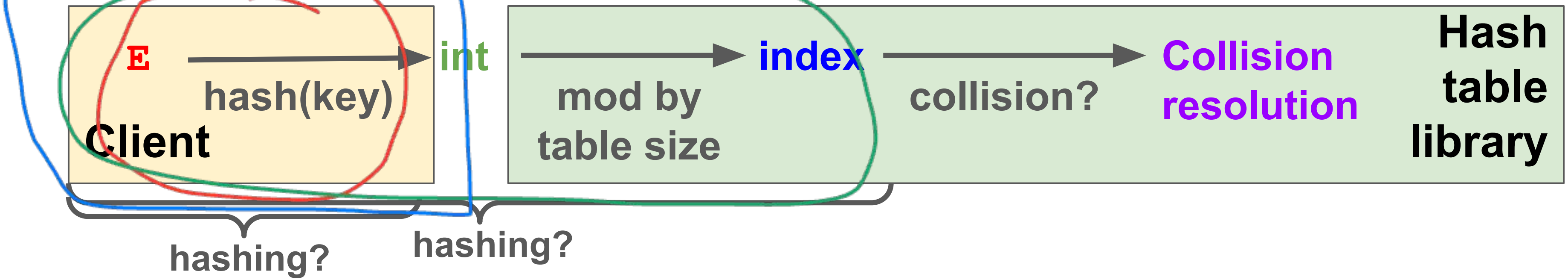
- To store keys of type **E**, we just need to be able to:
  1. Test equality: are you the **E** I'm looking for?
  2. Hash: convert any **E** to an int
- When hash tables are a reusable library, the division of responsibility generally breaks down into two roles:



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"



The 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;
  Day birthday; Month birthmonth; Year birthyear; 100
}
```

1200 x 30

- 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

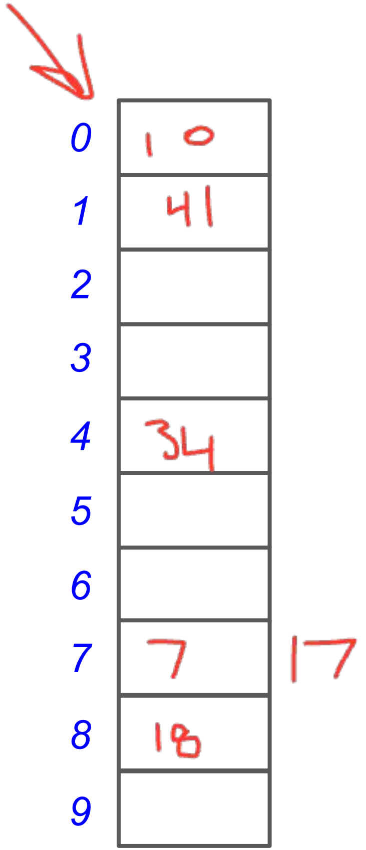
key space = integers

Simple hash function:

- Client: `h(x) = x`
- Library: `g(x) = h(x) % TableSize`
- `index = x % TableSize`

Example:

- TableSize = 10
- Insert 7, 18, 41, 34, 10, 17
- (As usual, ignoring corresponding data)



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?

wrap

1. $h(K) = s_0$

w 129

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

warp

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

Aside: Don't use pow

prime

17

31

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

$$= S_0 \cdot 37^0 + S_1 \cdot 37^1 + S_2 \cdot 37^2 + \cdots + S_{m-1} \cdot 37^{m-1}$$

Use Horner's Rule (to simplify):

$$= S_0 + 37\,(S_1 + 37\,(S_2 + 37\,(\ldots + 37 \cdot S_{m-1})\,)\,)$$

Specializing hash functions

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

1 2 3

http https ftp

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   % mod size

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

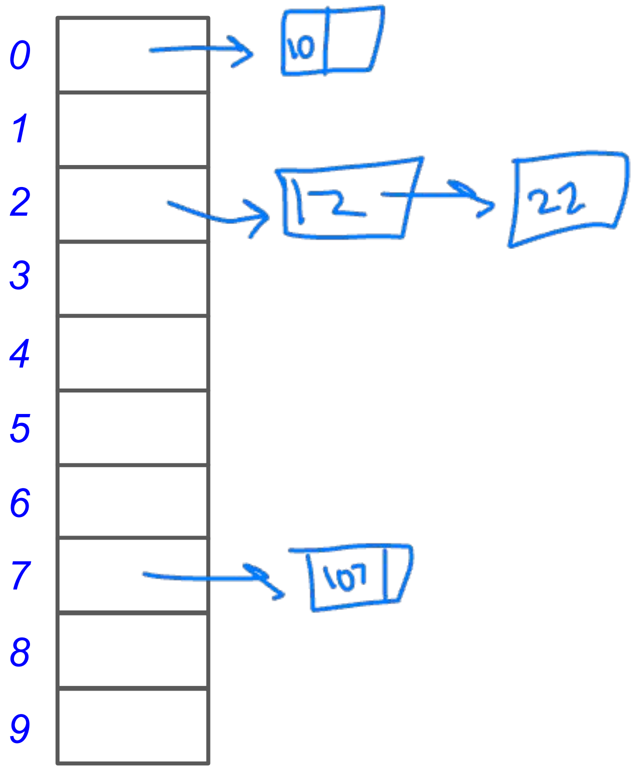
Separate Chaining

Open Addressing

- Linear Probing
- Quadratic Probing
- Double Hashing

Fri

Separate Chaining



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

Insertion Algorithm:

1. Check if duplicate exists
   - h(K) -> int -> index
   - LL.find(K) at index
2. If no duplicate, LL.insert(K) at index

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

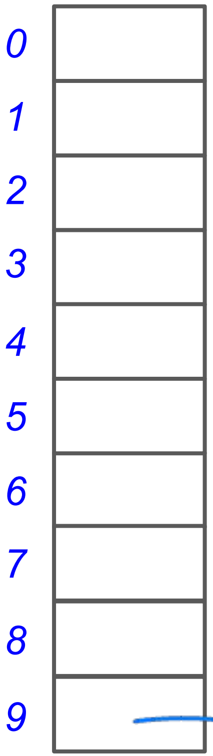
Delete?

Separate Chaining

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

Worst case time for find?

$O(n)$



Thoughts on separate chaining

Worst-case time for find?

prime

- 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…

More rigorous separate chaining analysis

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

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

number of elements

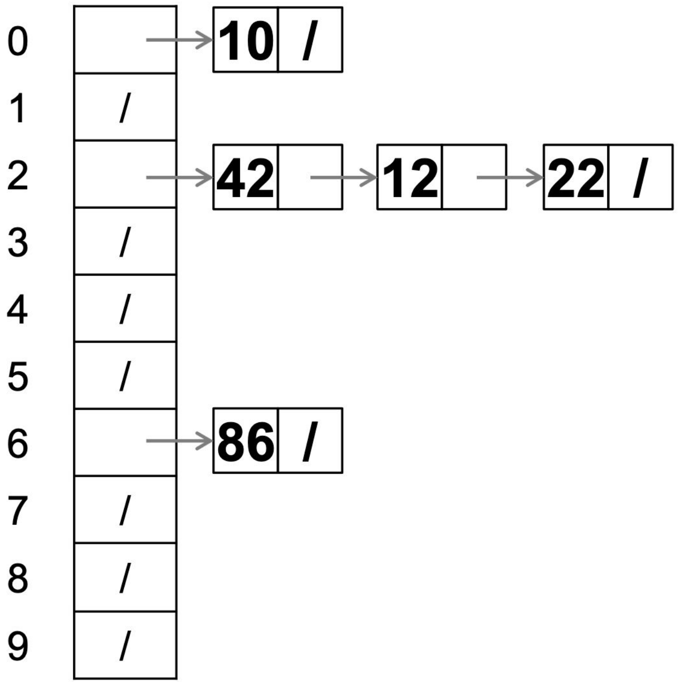
Under chaining, the average number of elements per bucket is λ

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

- Each unsuccessful **`find`** compares against λ items $\lambda = 1$
- Each successful **`find`** compares against $\lambda/2$ items
- How big should **`TableSize`** be??

$\lambda \leq 1$

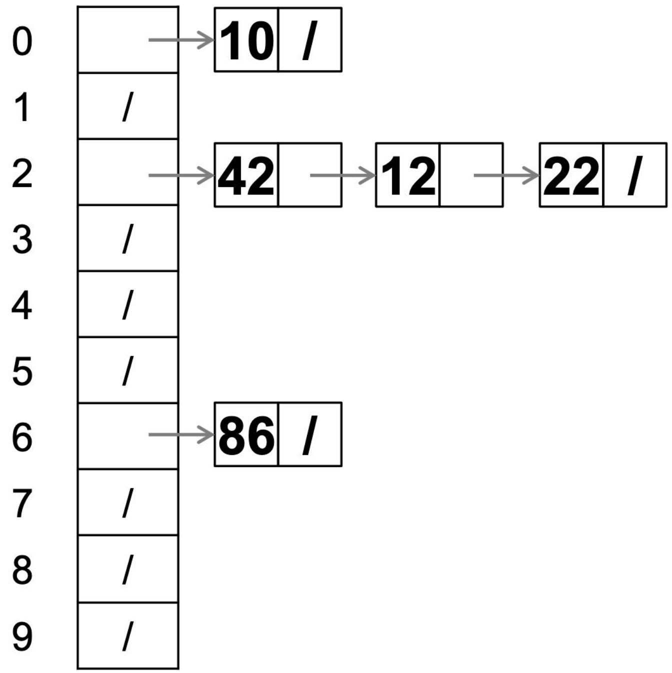
Load factor?



$\frac{5}{10}$

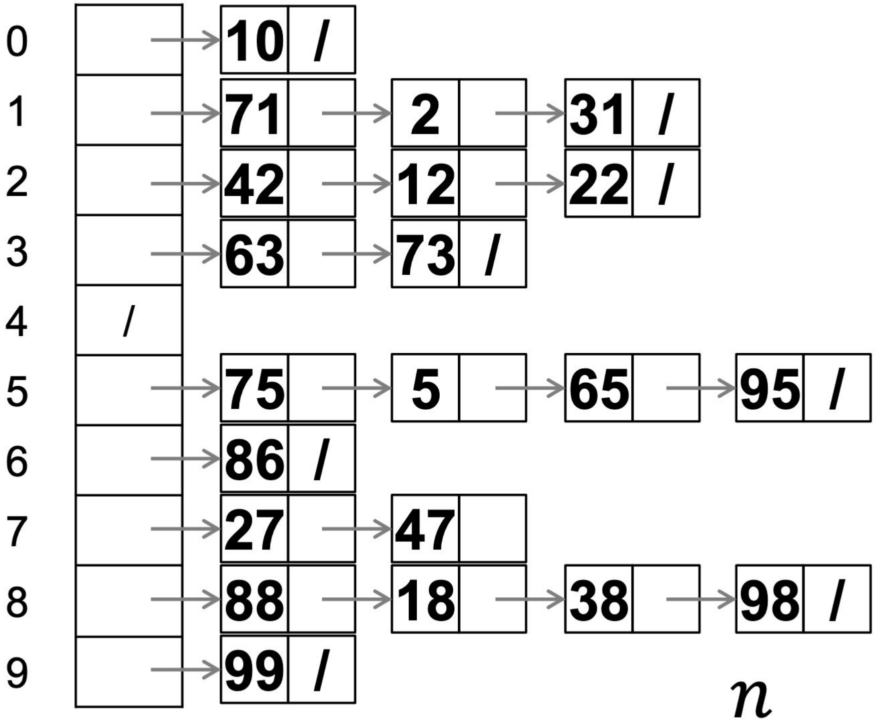
$$\lambda = \frac{n}{TableSize} = ?$$

Load factor?



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

Load factor?

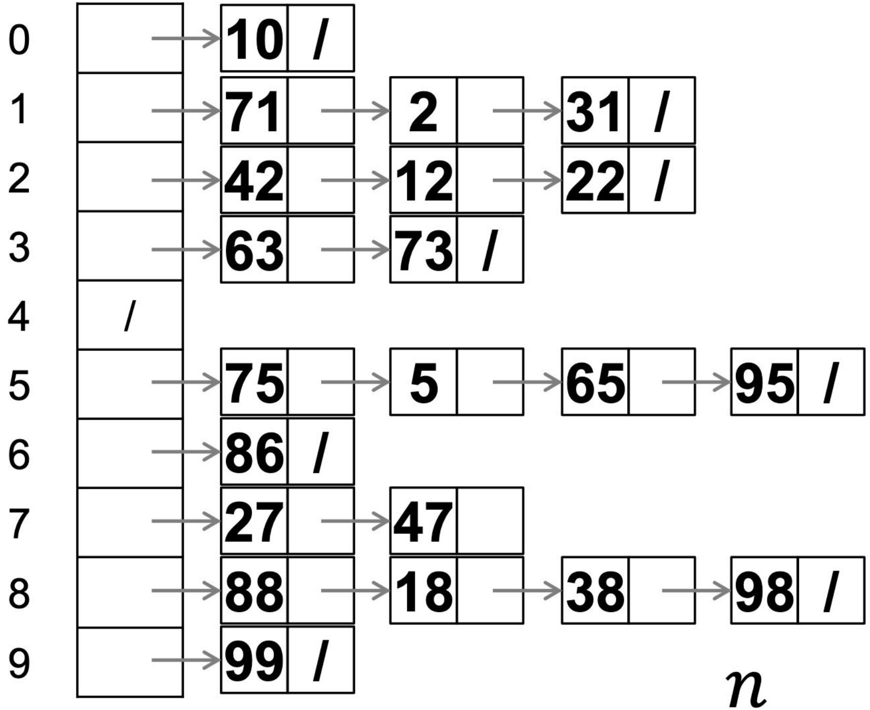


$$\lambda = \frac{n}{TableSize} = ?$$

$\frac{21}{10}$

~ 2.1

Load factor?



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