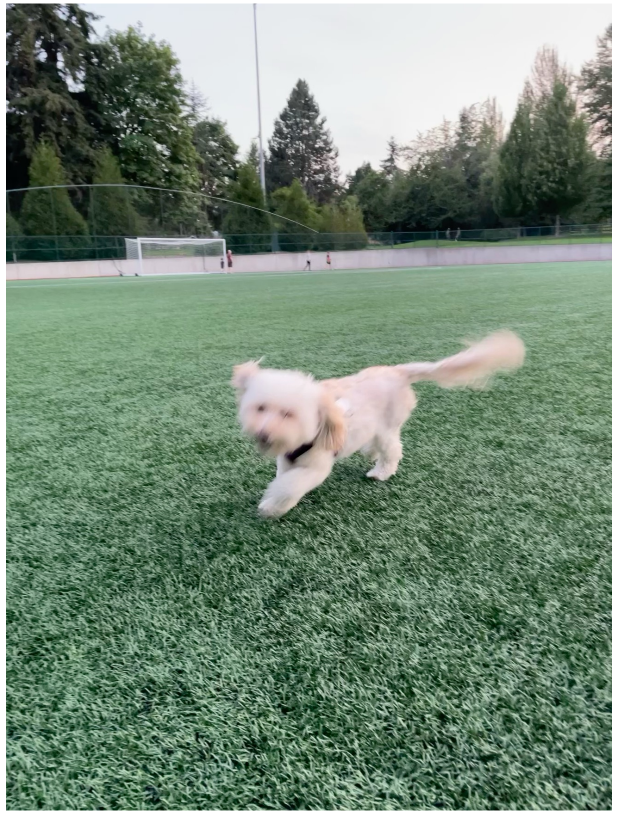
CSE 332: Data Structures & Parallelism
Lecture ~~9~~ 10: Hashing



Arthur Liu

Summer 2022

Announcements

- Reminder EX05 due tonight!
- P2 Writeup is significant! (A LOT TO WRITE!!)
- Midterm Monday
  - Review Session Today at 2:15 MORE 220

Outline for Today

- Hashing
  - Hashing
  - Collision Handling
    - Separate Chaining
    - Open Addressing

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

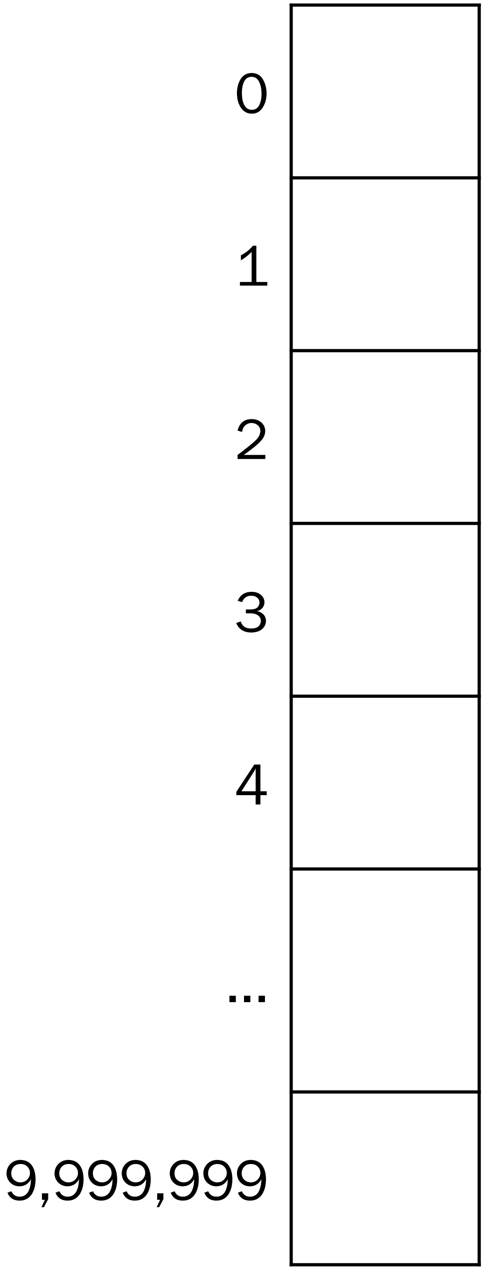
For dictionary with *n* key/value pairs

	`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)	(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).

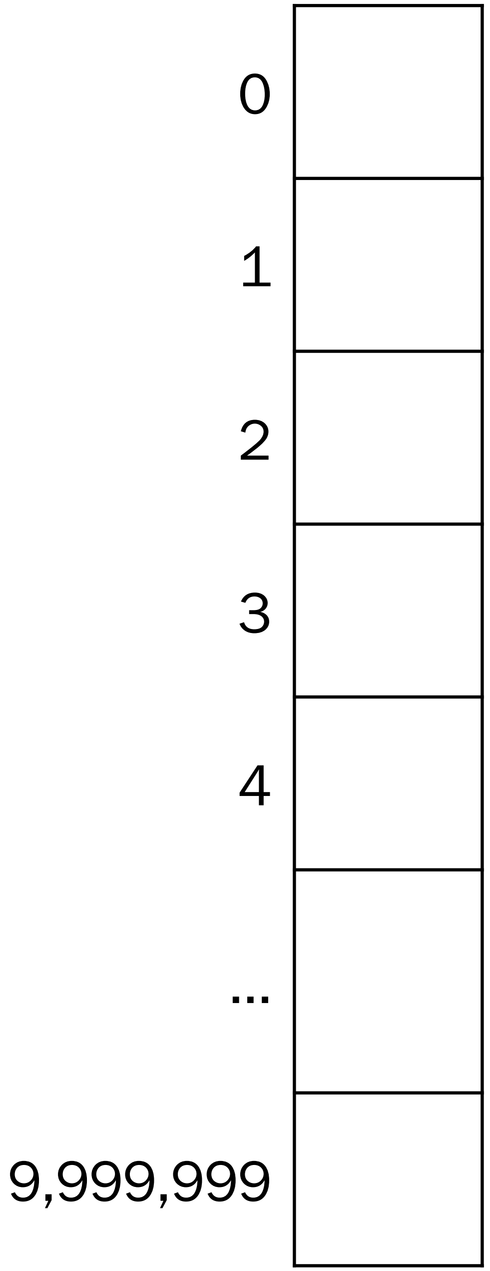
Really Big Array – my idea ☺

Really Big Array – my idea ☺



Keys: Student ID’s
0 – 9,999,999

Really Big Array – my idea ☺



Keys: Student ID's
0 – 9,999,999

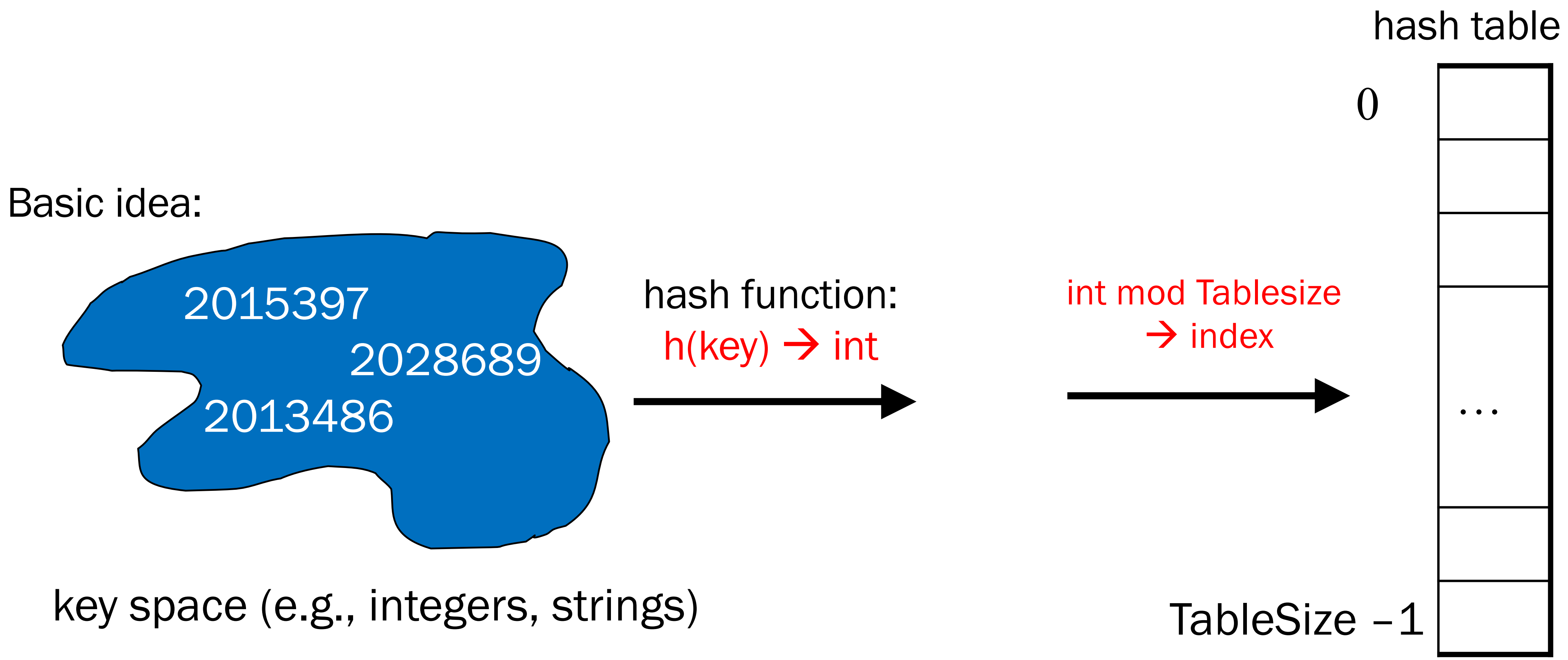
insert(4)

find(4)

delete(4)

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

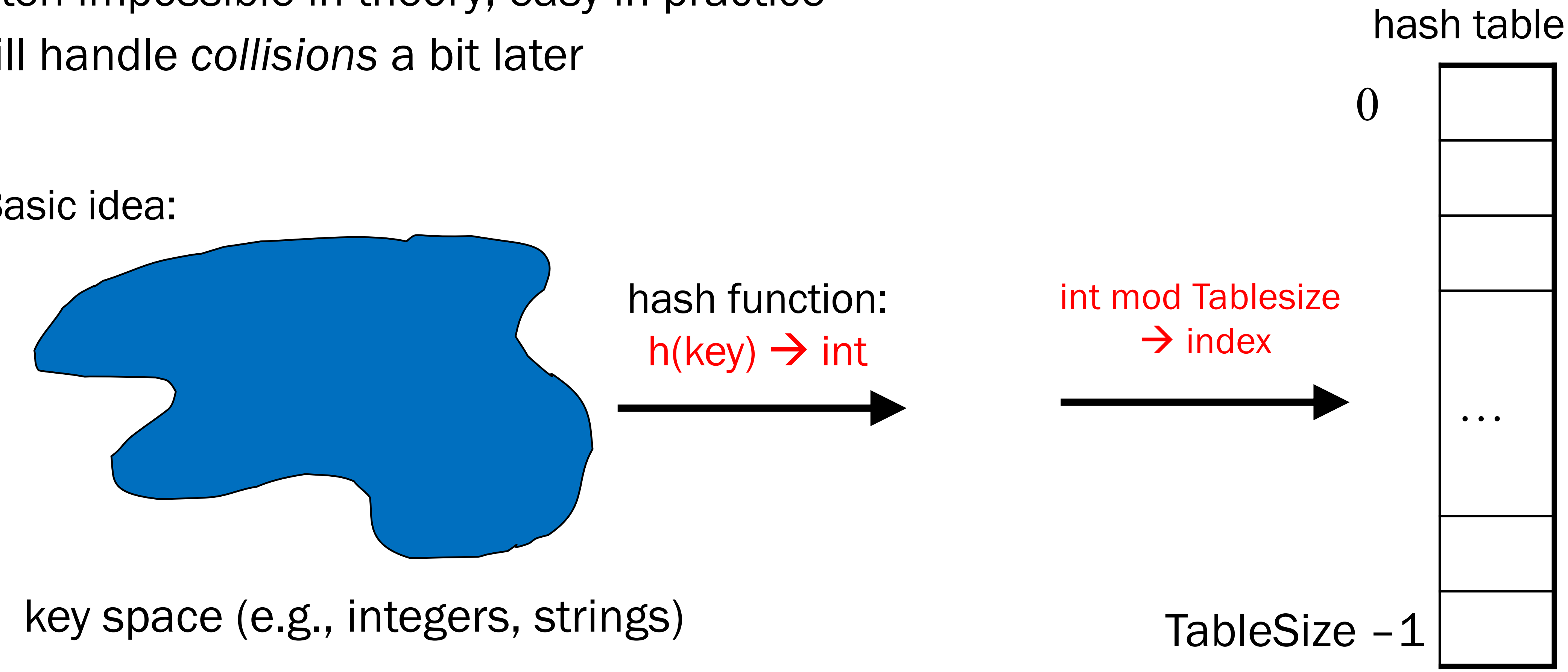


Hash Functions

An *ideal* hash function:

- Is fast to compute
- Is different for any two objects where `.equals() == false`
  - Often impossible in theory; easy in practice
  - Will handle *collisions* a bit later

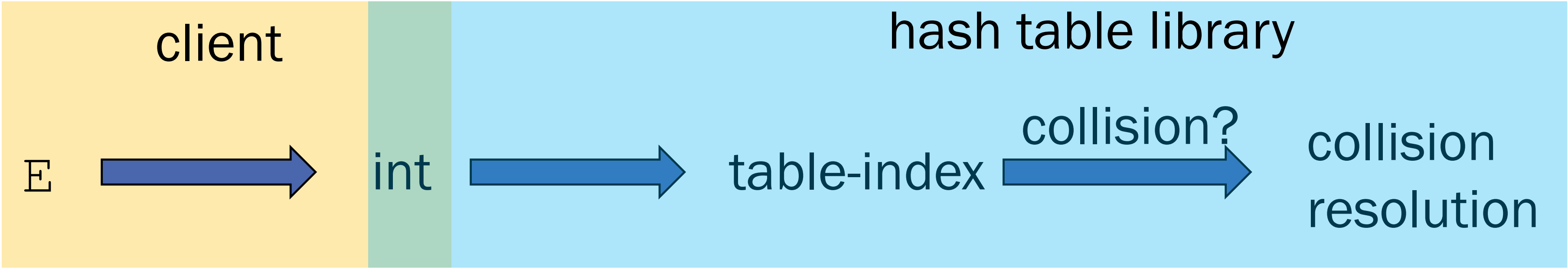
Basic idea:



Who’s Responsible for Making it good

- Clients write good hashCodes for their custom objects, 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.

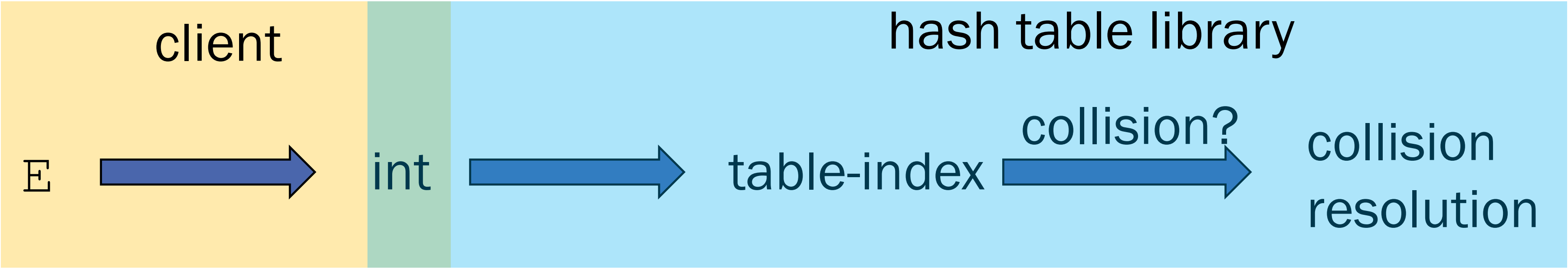
Class HashMap<K,V>

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

Type Parameters:

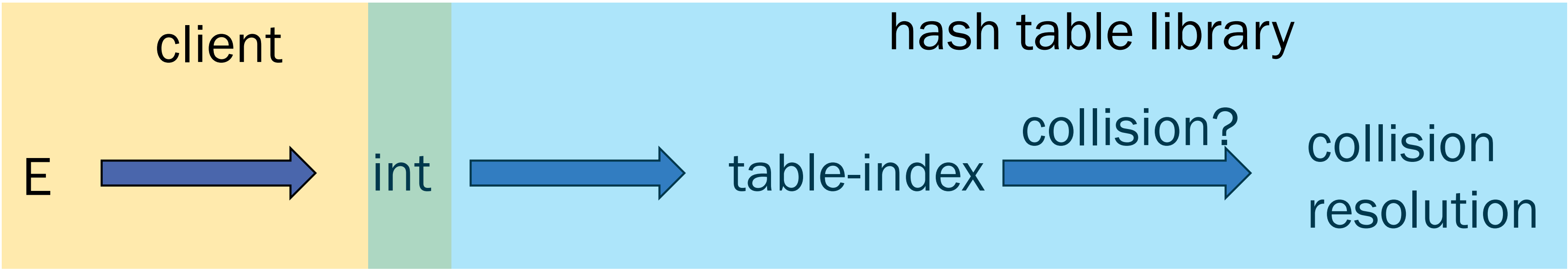
`K` - the type of keys maintained by this map

`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 Role's Responsibility to Make It Good



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)

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

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;
}
```

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`

- 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

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

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];
}
```

```
Math.Pow(37, i)   // bad
```

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*

Outline for Today

- Hashing
  - Hashing
  - Collision Handling
    - Separate Chaining
    - Open Addressing

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?

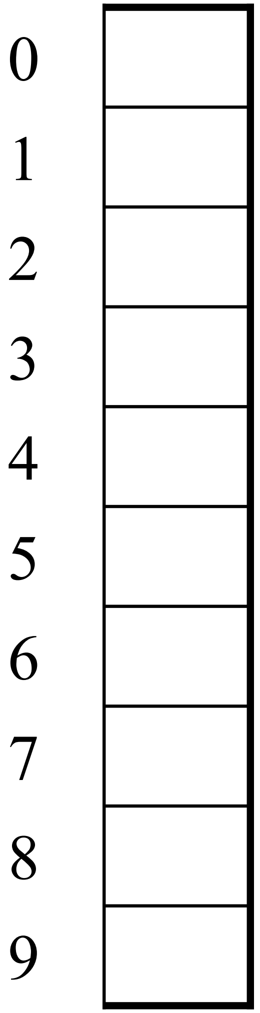
Flavors of Collision Resolution

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

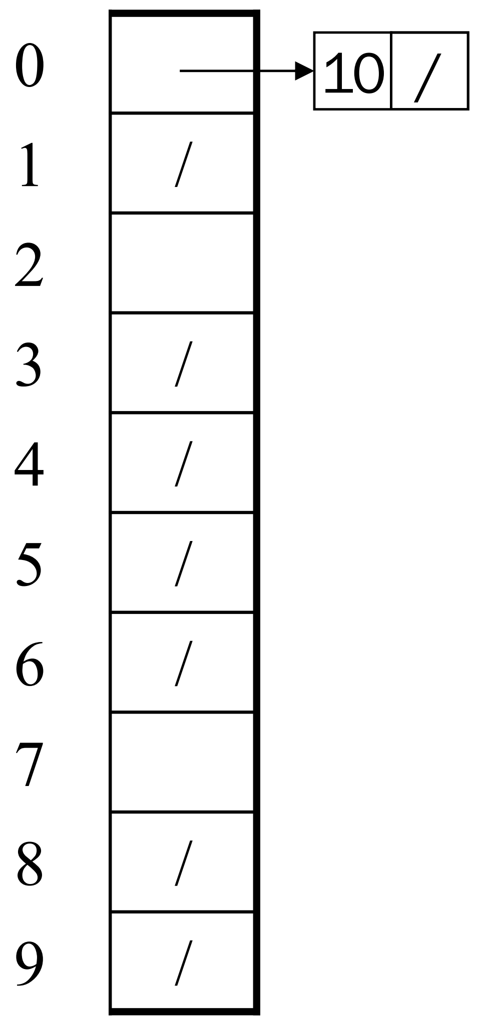
0	/
1	/
2	/
3	/
4	/
5	/
6	/
7	/
8	/
9	/

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

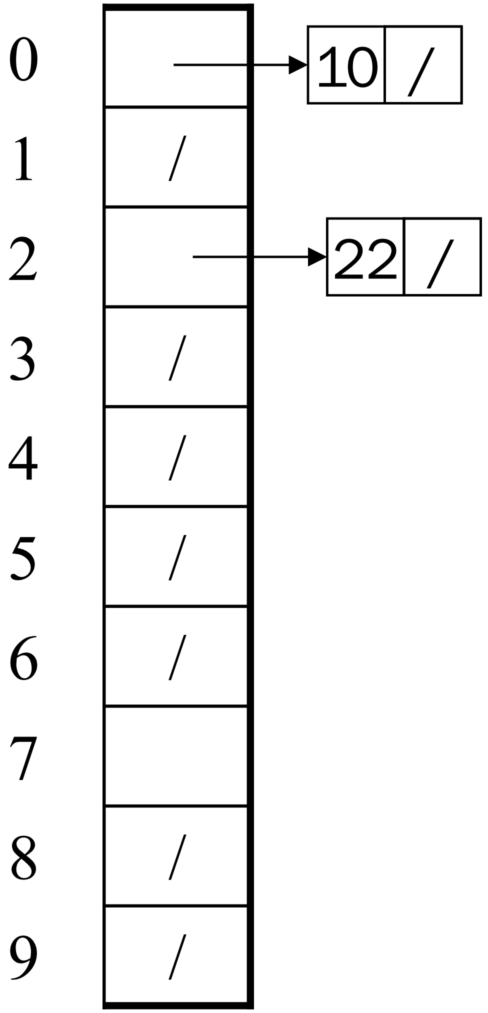


Chaining: All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

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

Separate Chaining

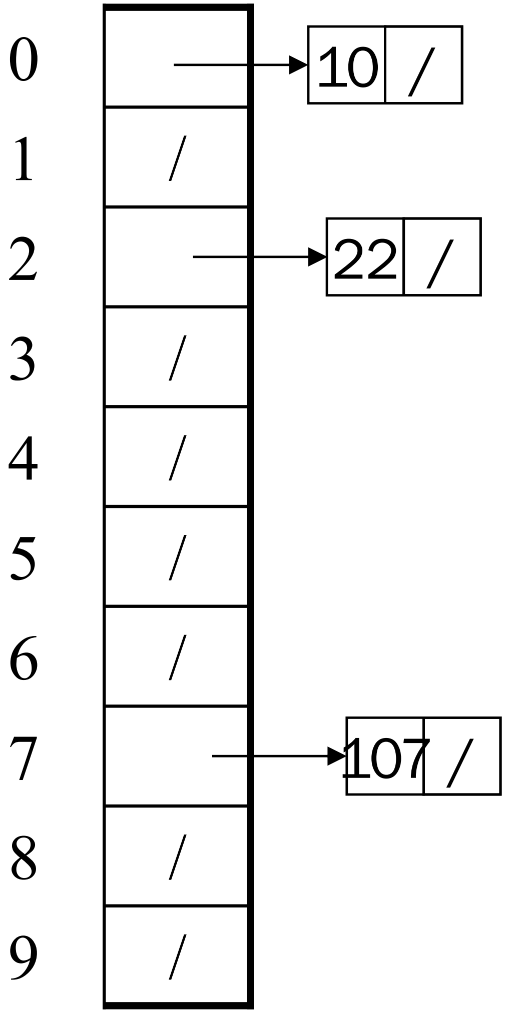


Chaining: All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

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

Separate Chaining

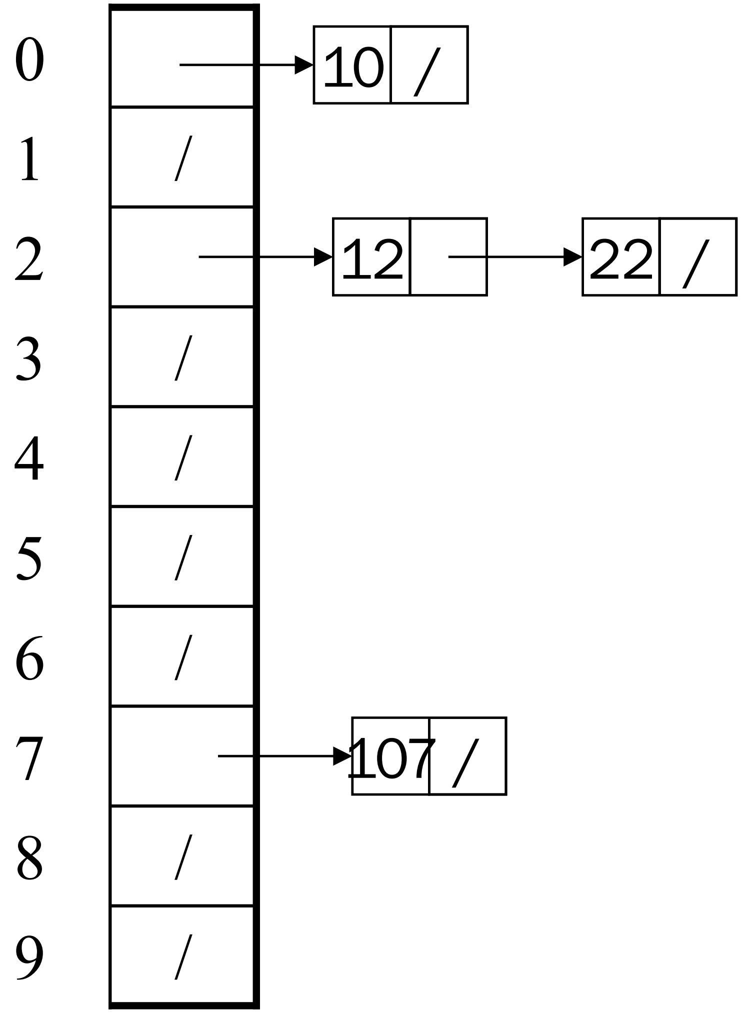


Chaining: All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

As easy as it sounds

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

Separate Chaining

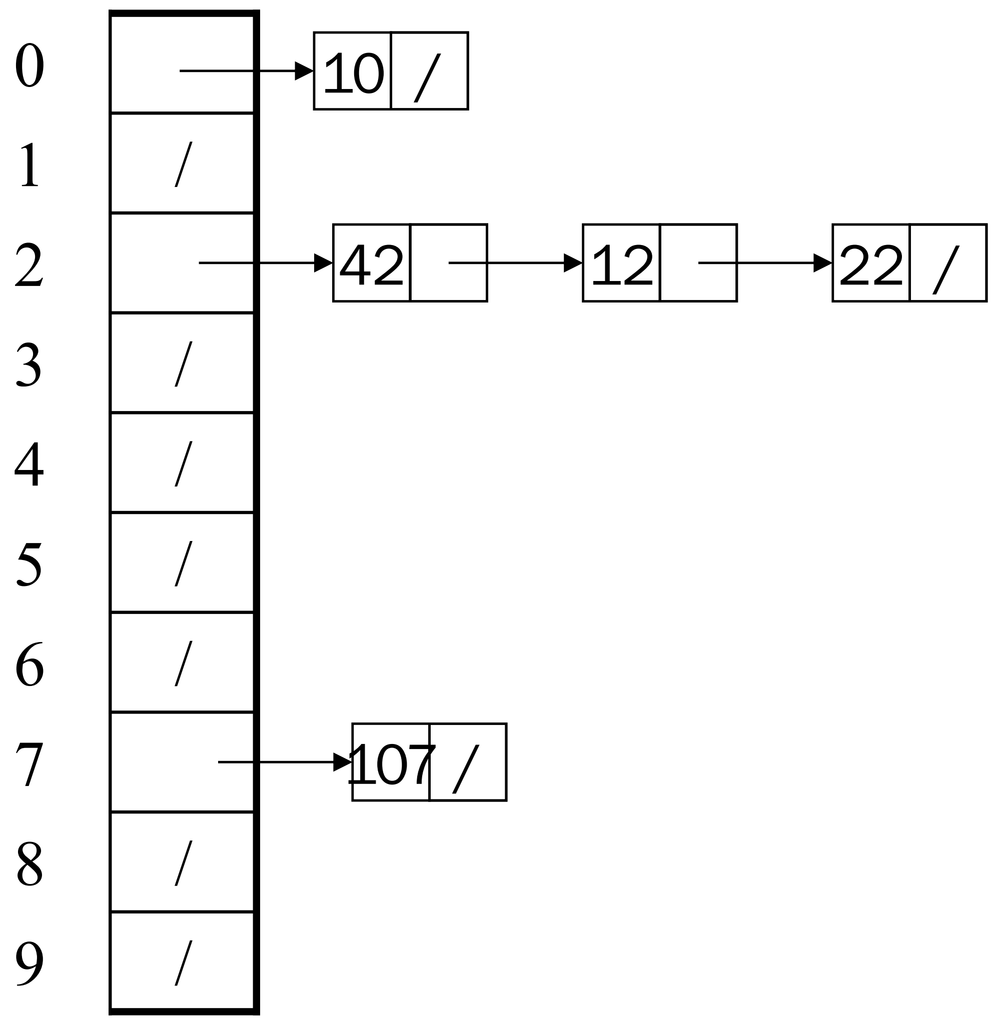


Chaining: All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

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

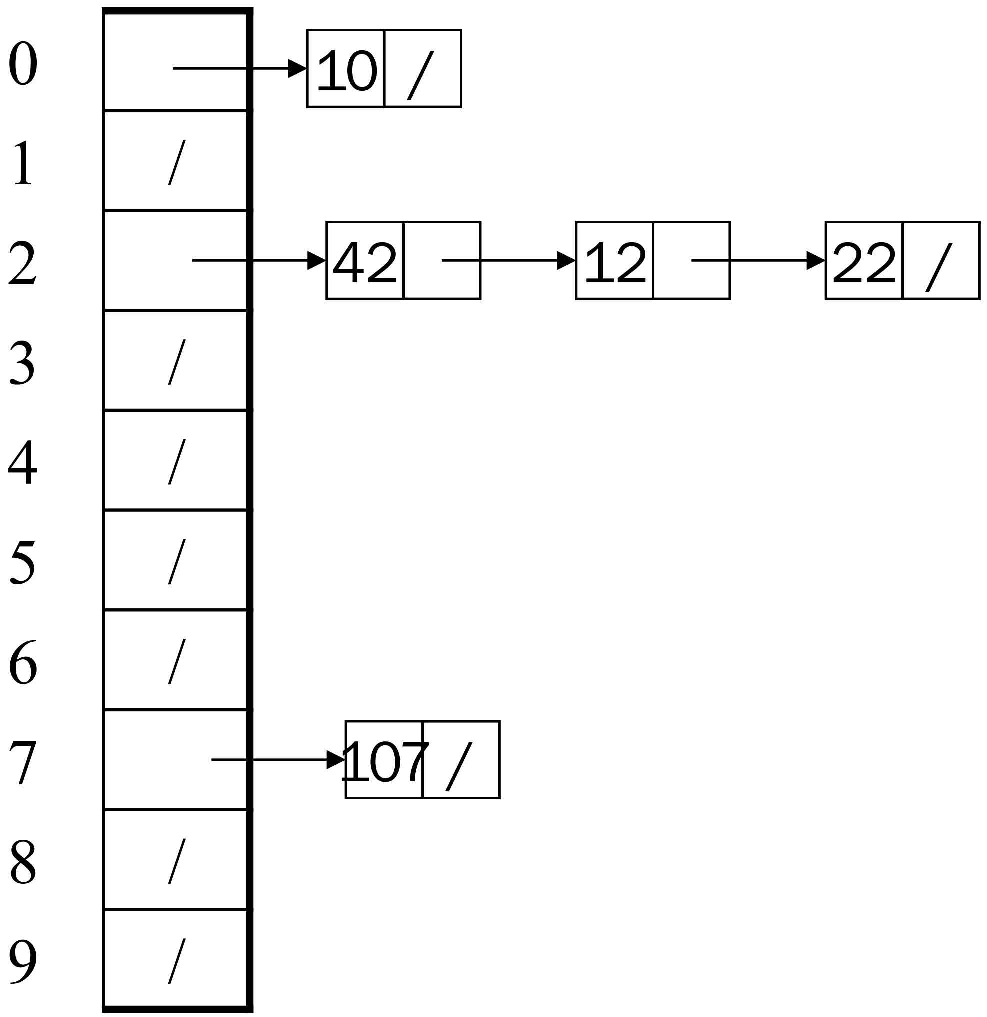
Worst-case time for **`find`**?

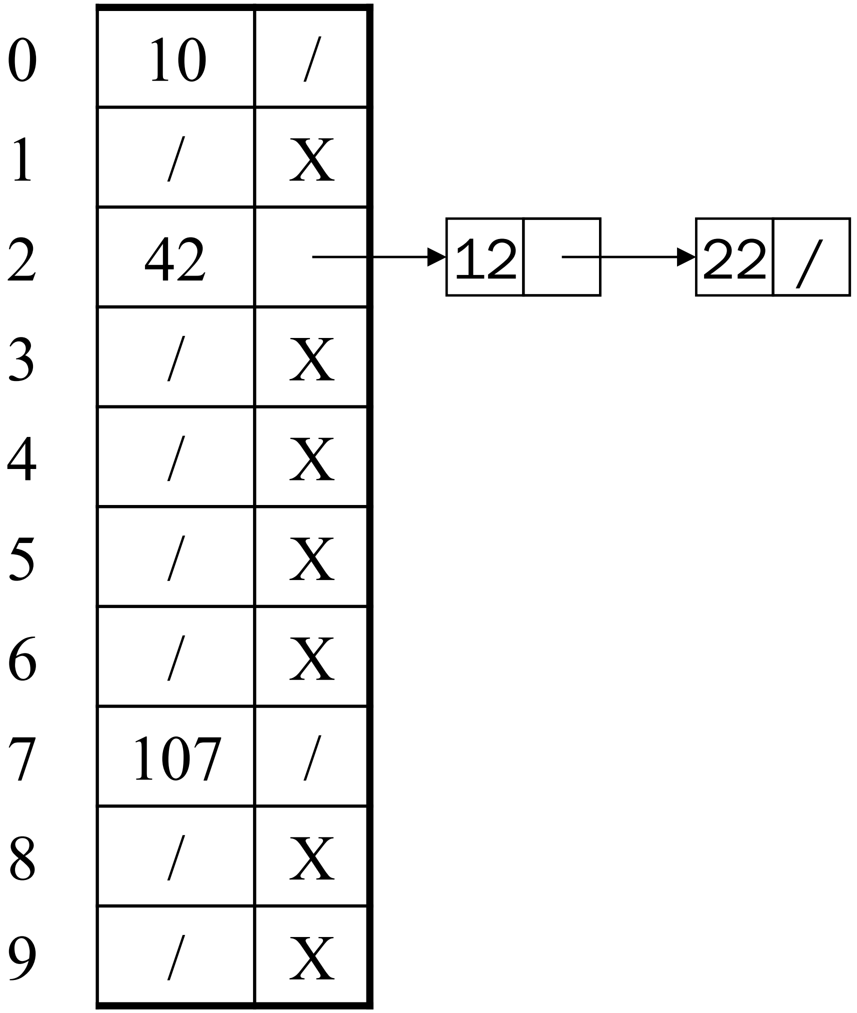
- Linear
- But only with really bad luck or bad hash function
- So not worth avoiding (e.g., with balanced trees at each bucket)
  - Keep # of items in each bucket small
  - Overhead of AVL tree, etc. not worth it if small # items per bucket

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??**

More rigorous separate chaining analysis

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

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

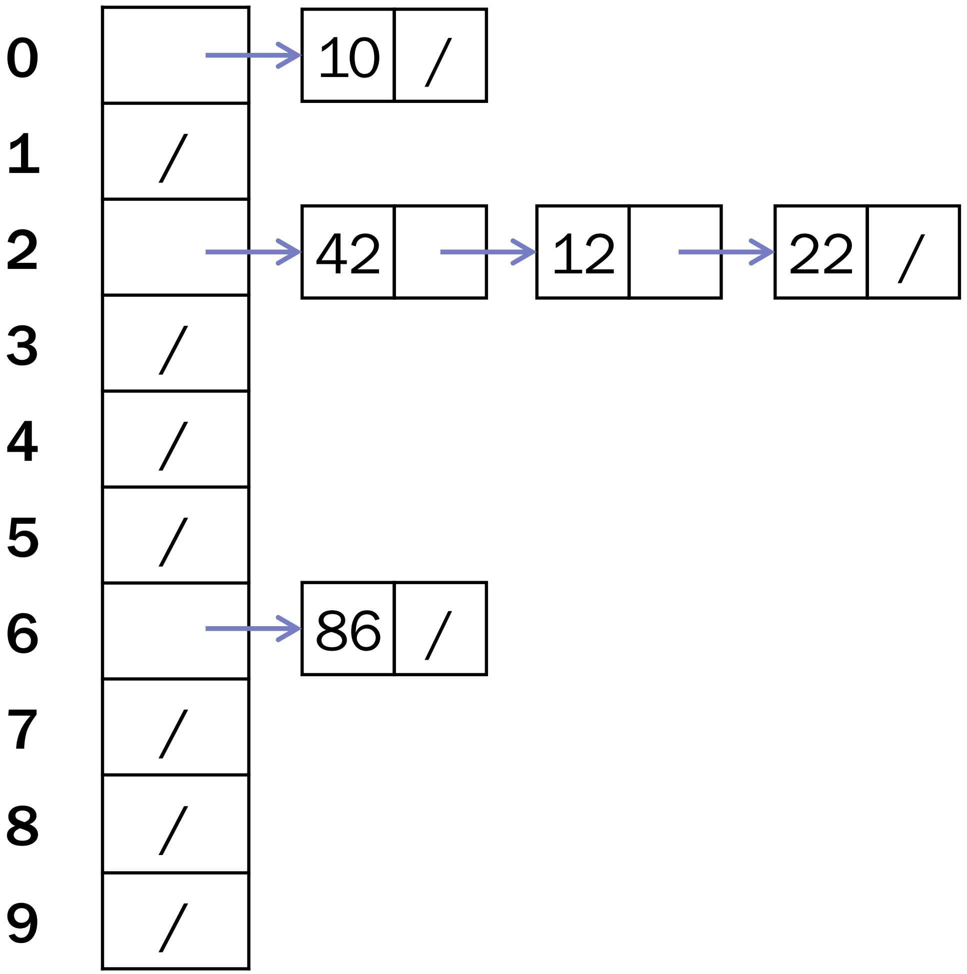
← number of elements

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

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

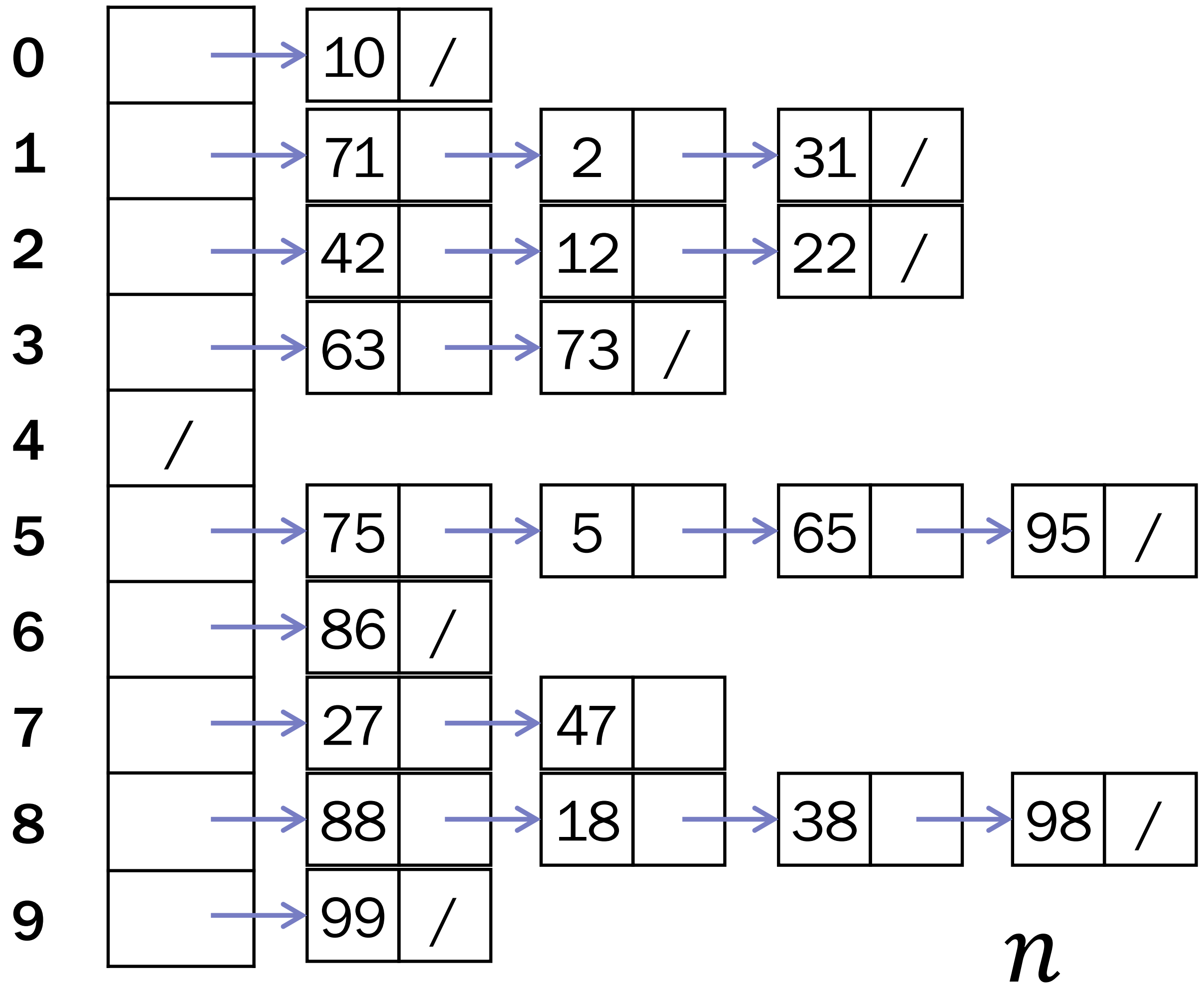
- **Each unsuccessful `find` compares against λ items**
- **Each successful `find` compares against $\lambda / 2$ items**
- If λ is low, find & insert likely to be O(1)
- We like to keep λ around 1 for separate chaining

Load Factor?



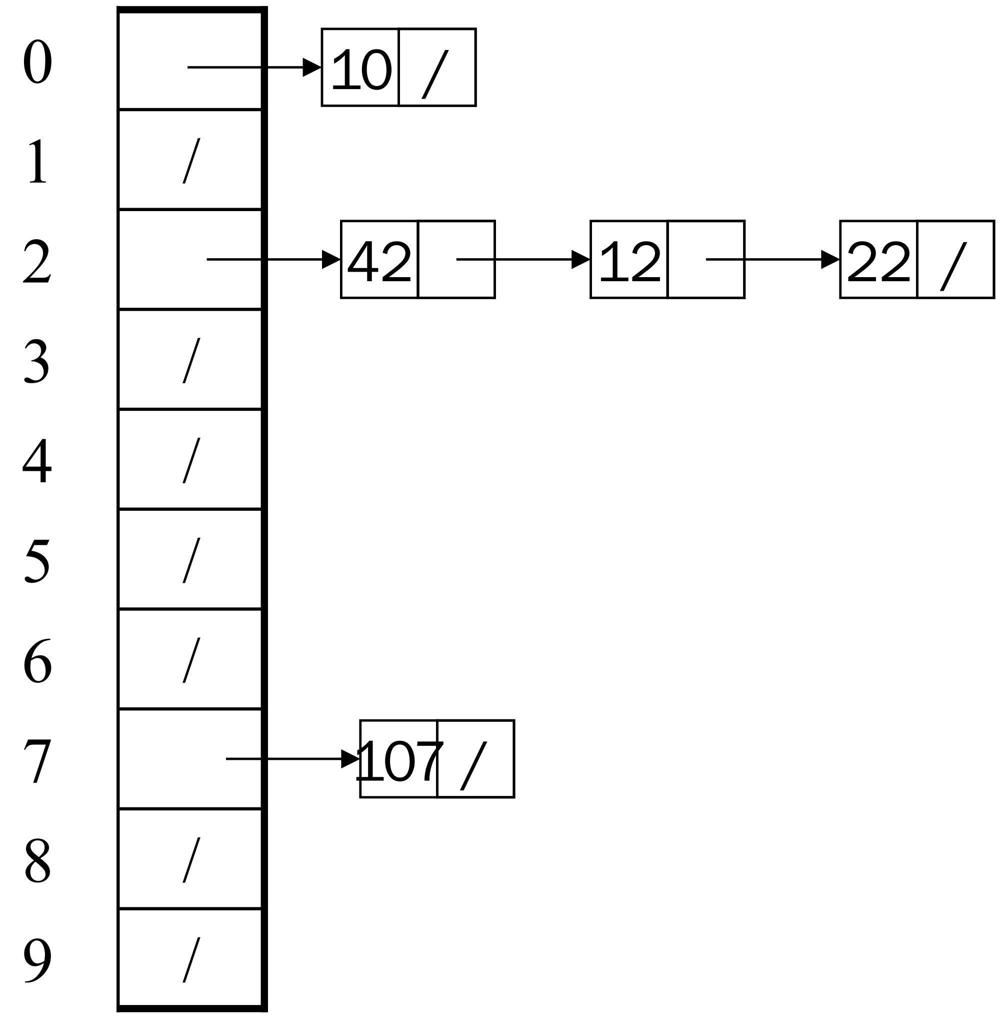
$$\lambda = \frac{n}{TableSize} = ?$$

Load Factor?



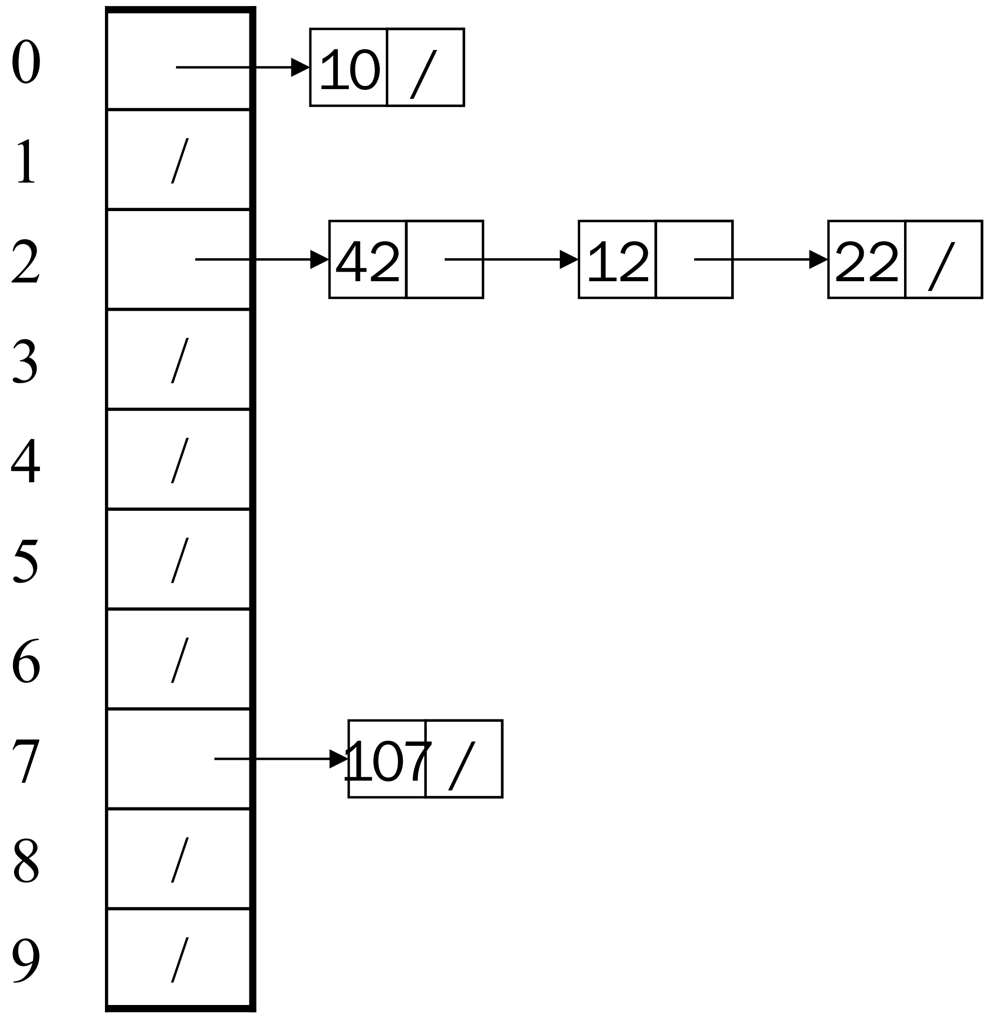
$$\lambda = \frac{n}{TableSize} = ?$$

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

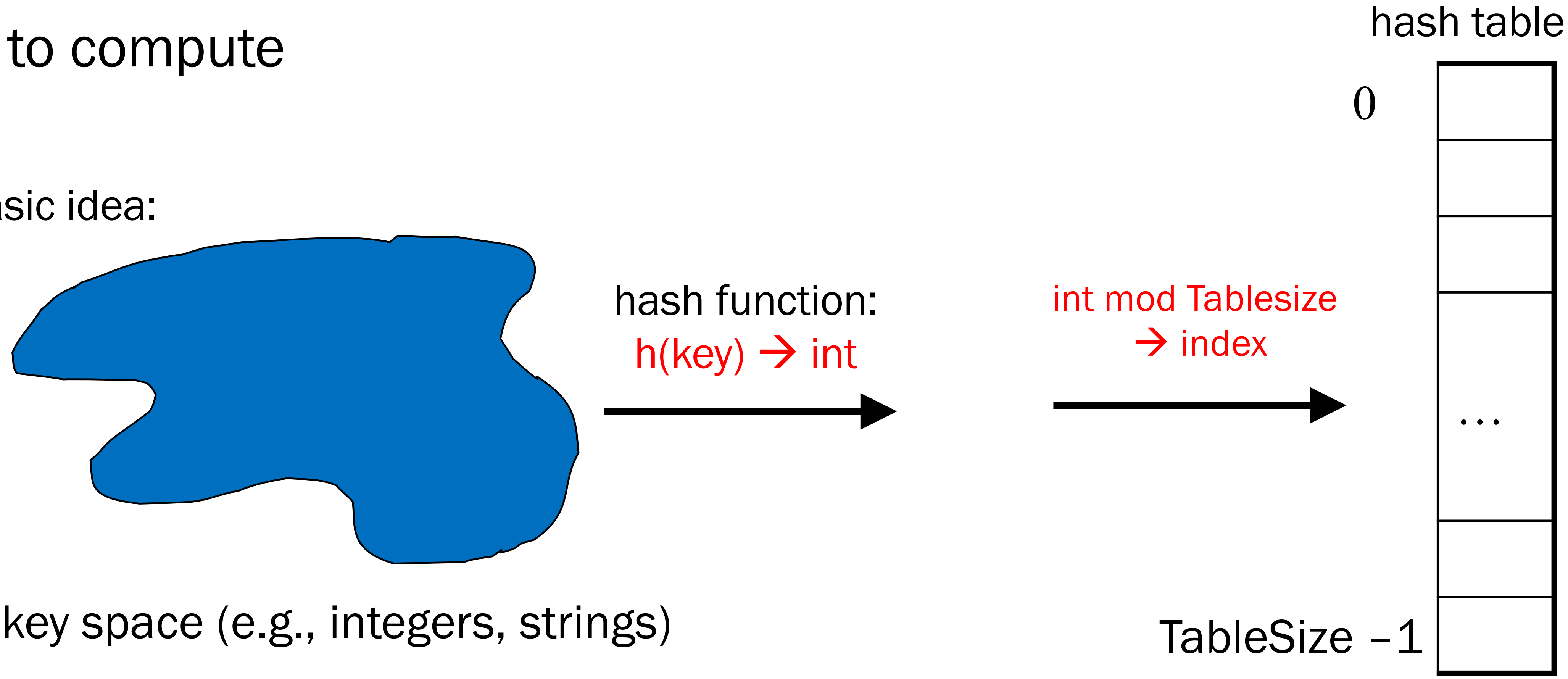
- Constant-time is better, right?
  - Yes, but you need "hashing to behave" (must avoid collisions)
  - Yes, but what if we want to **`findMin`**, **`findMax`**, **`predecessor`**, and **`successor`**, **`printSorted?`**
    - Hashtables are not designed to efficiently implement these operations

Client Collision Avoidance: Recall Our Ideal

An *ideal* hash function:

- <u>Is different for any two objects where `.equals() == false`</u>
  - <u>Often impossible in theory; easy in practice</u>
  - <u>Will handle *collisions* a bit later</u>
- Is fast to compute

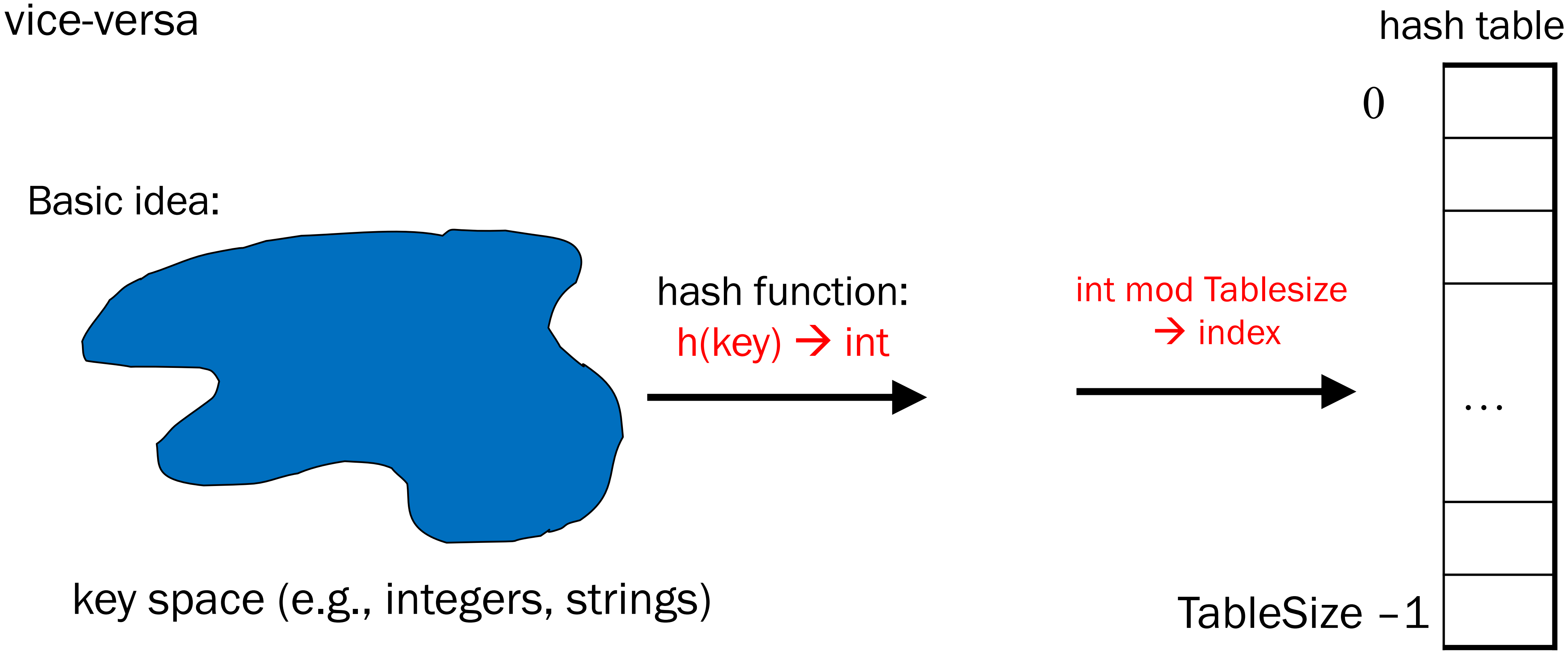
Basic idea:



But making Sure It's Still Correct

A *correct* hash function:

- Any two objects where `.equals() == true` must return the same hashcode!
  - If you update .equals(), you should update your hashCode() and vice-versa



🚨 🚨 🚨 Sneaky Bug Alert 🚨 🚨 🚨

Spot the bug >:C

pollev.com/artliu

ArrayList a

arr	13	15	12	0	0	0
size	3					

ArrayList b

arr	13	15	12	0	0	0	-3	-5
size	3							

```
// not the most ideal hashcode, but
// there's a fatal error
int hashCode() {
  int hash = 0;
  for (int i = 0; i < arr.length; i++) {
    hash += arr[i];
  }
  return hash;
}
```

Are these two ArrayList's equal()?

What's the error with the hashCode()?

Hashing and Equality

- Our use of int key can lead to us overlooking a critical detail:
  - We initially *hash* **`E`** to get a table index
  - While chaining or probing we need to determine if this is the **`E`** that I am looking for… ie: **equality testing!!!**

- So a hash table needs a hash function and an equality testing
  - In the Java library each object has an **`equals`** method and a **`hashCode`** method

```
class Object {
  boolean equals(Object o) {…}
  int hashCode() {…}
  …
}
```

Equal objects must hash the same

The Java library (and your project hash table) make a very important assumption that clients must satisfy…

- Object-oriented way of saying it:

  If `a.equals(b)`, then we must require `a.hashCode()==b.hashCode()`

- Function object way of saying it:

  If `c.compare(a,b) == 0`, then we must require

  `h.hash(a) == h.hash(b)`

- If you ever override equals
  - You need to override hashCode also in a consistent way
  - See CoreJava book, Chapter 5 for other "gotchas" with equals

By the way: comparison has rules too

We have not emphasized important "rules" about comparison for:

- All our dictionaries
- Sorting (next major topic)

Comparison must impose a consistent, total ordering:

For all **`a`**, **`b`**, and **`c`**,

- If **`compare(a,b) < 0`**, then **`compare(b,a) > 0`**
- If **`compare(a,b) == 0`**, then **`compare(b,a) == 0`**
- If **`compare(a,b) < 0`** and **`compare(b,c) < 0`**, then **`compare(a,c) < 0`**

Outline

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