CS 201: Data Structures
Hashing I

Aaron Bauer

Winter 2021

Motivating Hash Tables

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

	`put`	`contains`	`remove`
• 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	$O(n)$	$O(\mathtt{log}\ n)$	$O(n)$
• *Balanced* tree	$O(\mathtt{log}\ n)$	$O(\mathtt{log}\ n)$	$O(\mathtt{log}\ n)$
• Magic array	$O(1)$	$O(1)$	$O(1)$

Sufficient "magic":

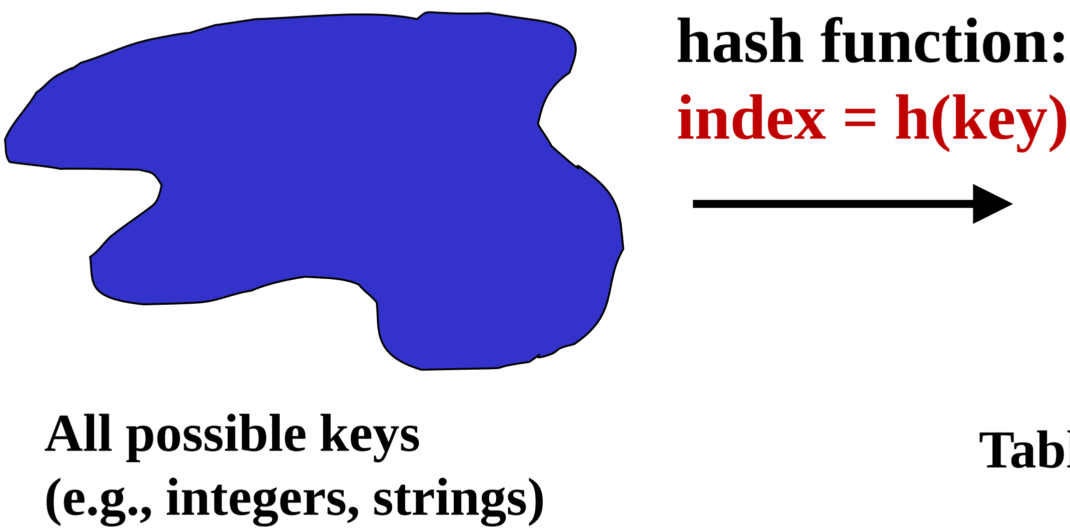
- Remove the requirement that elements are *ordered* [easy]
- Use key to compute array index for an item in $O(1)$ time [doable]
- Have a different index for every item [magic]

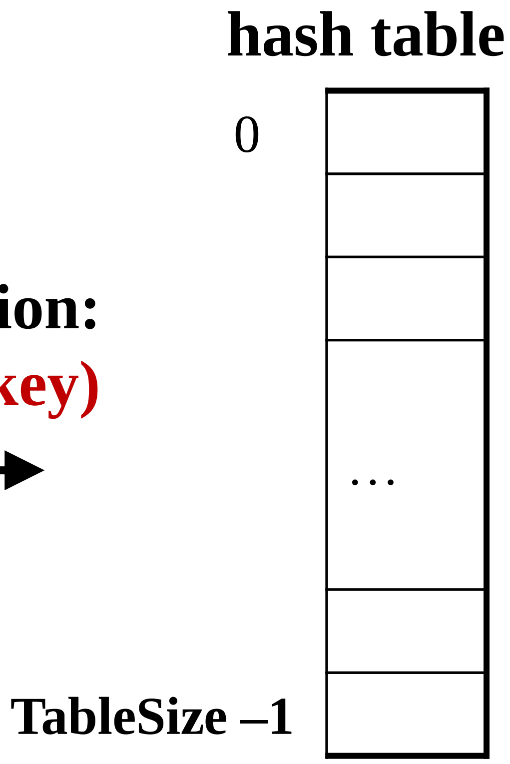
Hash Tables: Turn Key Into Array Index

- Aim for constant time **`put`**, **`get`**, **`contains`**, and **`remove`**
  - "On average" under some often-reasonable assumptions

- A hash table is an array of some fixed size

- Basic idea:





Hash Tables: More Keys Than Spots

- 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*)

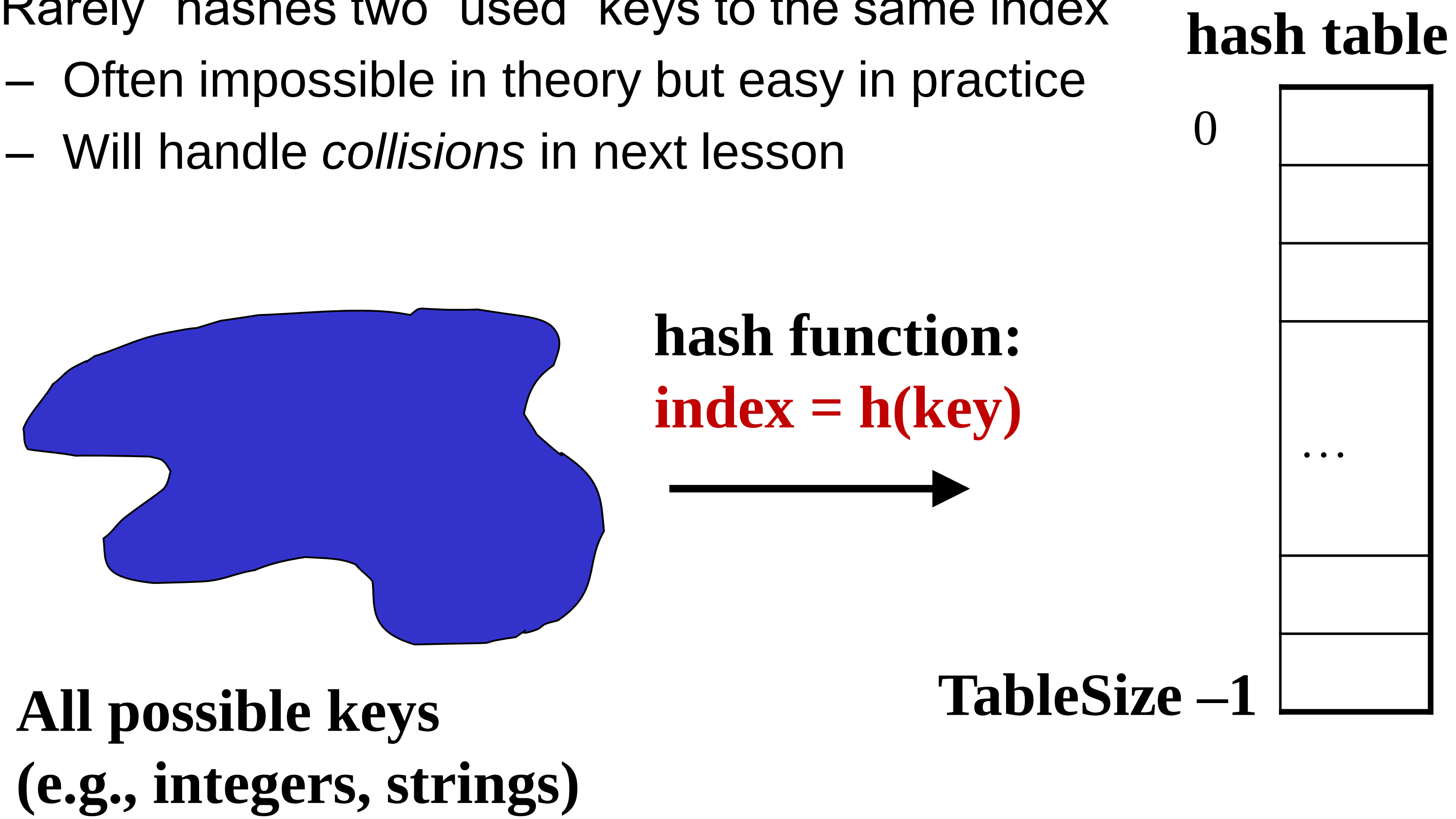
Many maps have this property

- Compiler: All possible identifiers allowed by the language vs. those used in some file of one program
- Database: All possible volunteer names vs. volunteers signed up with Bauer for MN
- 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* in next lesson



Hashing integers

- key space = integers
- Simple hash function:

  `h(key) = key % TableSize`

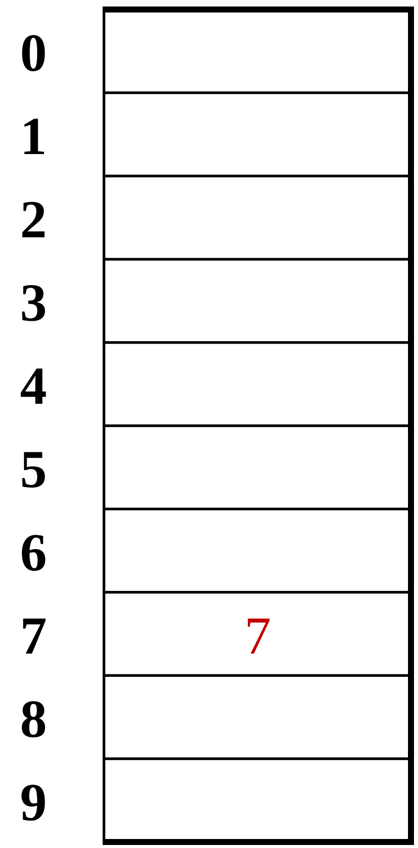
  – Fairly fast and natural
- Example:
  – **`TableSize`** = 10
  – Insert 7, 18, 41, 34, 10
  – (ignoring the values of these key-value pairs—we'll think of these as data "along for the ride")

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hashing integers

- key space = integers
- Simple hash function:

  `h(key) = key % TableSize`
  - Fairly fast and natural
- Example:
  - **`TableSize`** = 10
  - Insert 7, 18, 41, 34, 10
  - (As usual, ignoring data “along for the ride”)

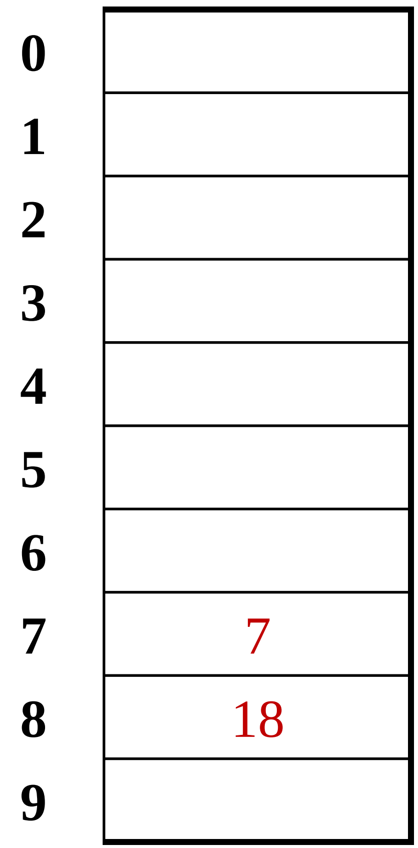


Hashing integers

- key space = integers
- Simple hash function:

  `h(key) = key % TableSize`

  - Fairly fast and natural
- Example:
  - **`TableSize`** = 10
  - Insert 7, 18, 41, 34, 10
  - (As usual, ignoring data “along for the ride”)



Hashing integers

- key space = integers
- Simple hash function:

  `h(key) = key % TableSize`
  - Fairly fast and natural
- Example:
  - **`TableSize`** = 10
  - Insert 7, 18, 41, 34, 10
  - (As usual, ignoring data “along for the ride”)

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

Hashing integers

- key space = integers
- Simple hash function:

  `h(key) = key % TableSize`

  – Fairly fast and natural
- Example:
  – **`TableSize`** = 10
  – Insert 7, 18, 41, 34, 10
  – (As usual, ignoring data “along for the ride”)

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

Hashing integers

- key space = integers
- Simple hash function:

  `h(key) = key % TableSize`

  – Fairly fast and natural

- Example:
  – **`TableSize`** = 10
  – Insert 7, 18, 41, 34, 10
  – (As usual, ignoring data “along for the ride”)

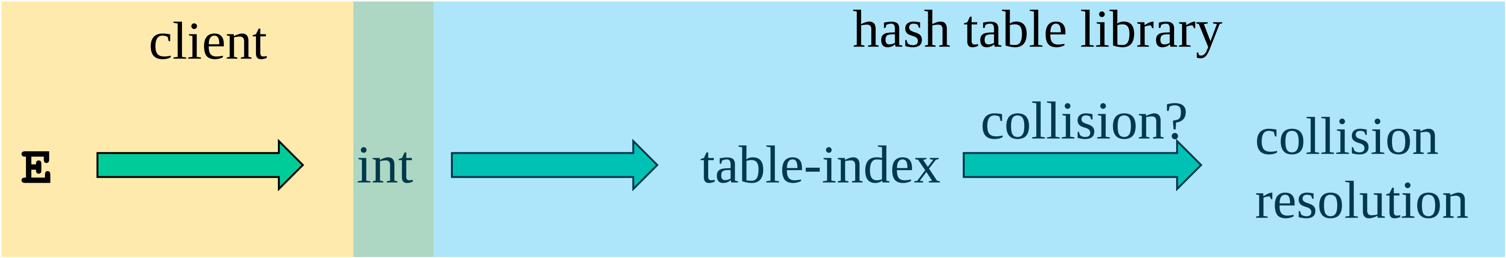
0	10
1	41
2	
3	
4	34
5	
6	
7	7
8	18
9	

Who hashes what?

- Hash tables can be generic
  - To store elements of type **`E`**, we just need **`E`** to be:

    Hashable: convert any **`E`** to an **`int`**

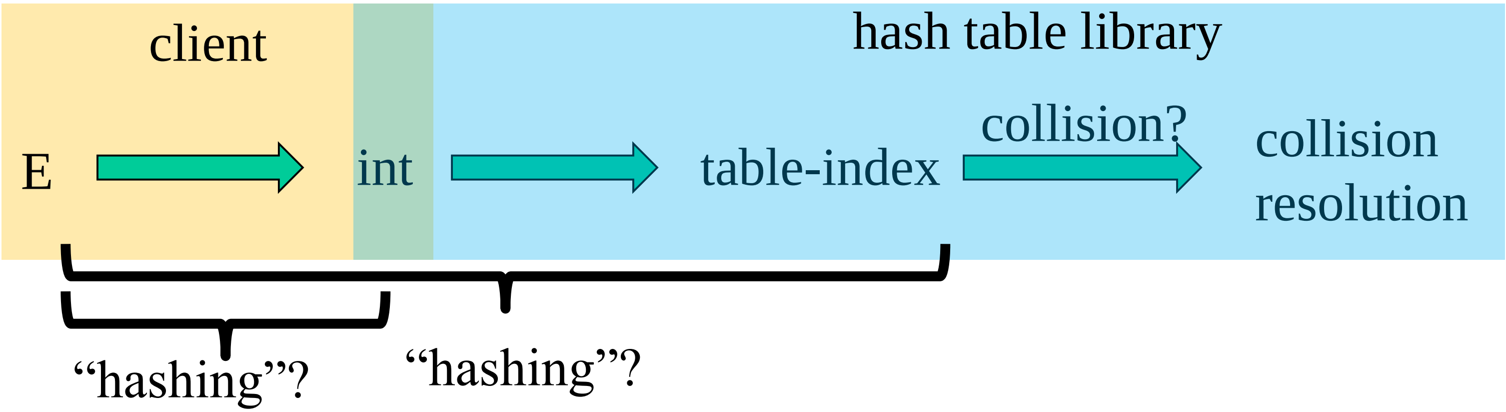
- When hash tables are a reusable library (as they are in Java), the division of responsibility generally breaks down into two roles:



- 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
- 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

Okay, what about keys that aren't ints?

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

- Very important example: Strings
  - Key space K = $s_0 s_1 s_2 \ldots s_{m-1}$
    - (where s_i are chars: between 0 to 65,535, inclusive)
  - Some choices: Which avoid collisions best?

1. $h(K) = s_0$ % TableSize

2. $h(K) = (s_0 + s_1 + \ldots s_{m-1})$ % TableSize

3. $h(K) = (s_0 + s_1 * 31 + s_2 * 31^2 + \ldots s_{m-1} * 31^{m-1})$ % TableSize

   (I'll demonstrate this is what Java does)

Specializing hash functions

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

Hashing and comparing

- Need to emphasize a critical detail:
  - We initially *hash* key **`E`** to get a table index
  - To confirm that index has what we're looking for we check if our key equals the key stored at that index
- So a hash table needs a hash function and a way to compare keys
  - The Java library uses an object-oriented approach: each object has methods **`equals`** and **`hashCode`**

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

Equal Objects Must Hash the Same

- The Java library make a crucial assumption clients must satisfy
  - And all hash tables make analogous assumptions
- Object-oriented way of saying it:

  If `a.equals(b)`, then `a.hashCode() == b.hashCode()`
- Why is this essential?
  - Necessary in order for correct hash table behavior
- Why is this up to the client?
  - Both methods depend on private fields, so library can’t do it
- So *always* override `hashCode` *correctly* if you override `equals`
  - Many libraries use hash tables on your objects

Example

CalendarDate.java in VS Code

Tougher example

- Suppose you had a **`Fraction`** class where **`equals`** returned **`true`** for 1/2 and 3/6, etc.

- Then must override **`hashCode`** and cannot hash just based on the numerator and denominator
  - Need 1/2 and 3/6 to hash to the same int

- If you write software for a living, you are less likely to implement hash tables from scratch than you are likely to encounter this issue

One expert suggestion

- int result = 17;
- for-each field f
  - int fieldHashcode =
    - boolean: (f ? 1: 0)
    - byte, char, short, int: (int) f
    - long: (int) (f ^ (f >>> 32))
    - float: Float.floatToIntBits(f)
    - double: Double.doubleToLongBits(f), then as long above
    - Object: object.hashCode( )
  - result = 31 * result + fieldHashcode

Conclusions and notes on hashing

- The hash table is one of the most important data structures
  - Supports **`contains`**, **`put`**, **`get`** and **`remove`** efficiently (constant time!)
  - We can iterate over the keys and/or values, but *they are not guaranteed to be in any particular order*

- Important to use a good hash function

- Side-comment: hash functions have uses beyond hash tables
  - Examples: Cryptography, check-sums

- Big remaining topic: Handling collisions