CSE373: Data Structures & Algorithms
Lecture 13: Hash Tables

Linda Shapiro

Winter 2015

Announcements

Motivating Hash Tables

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

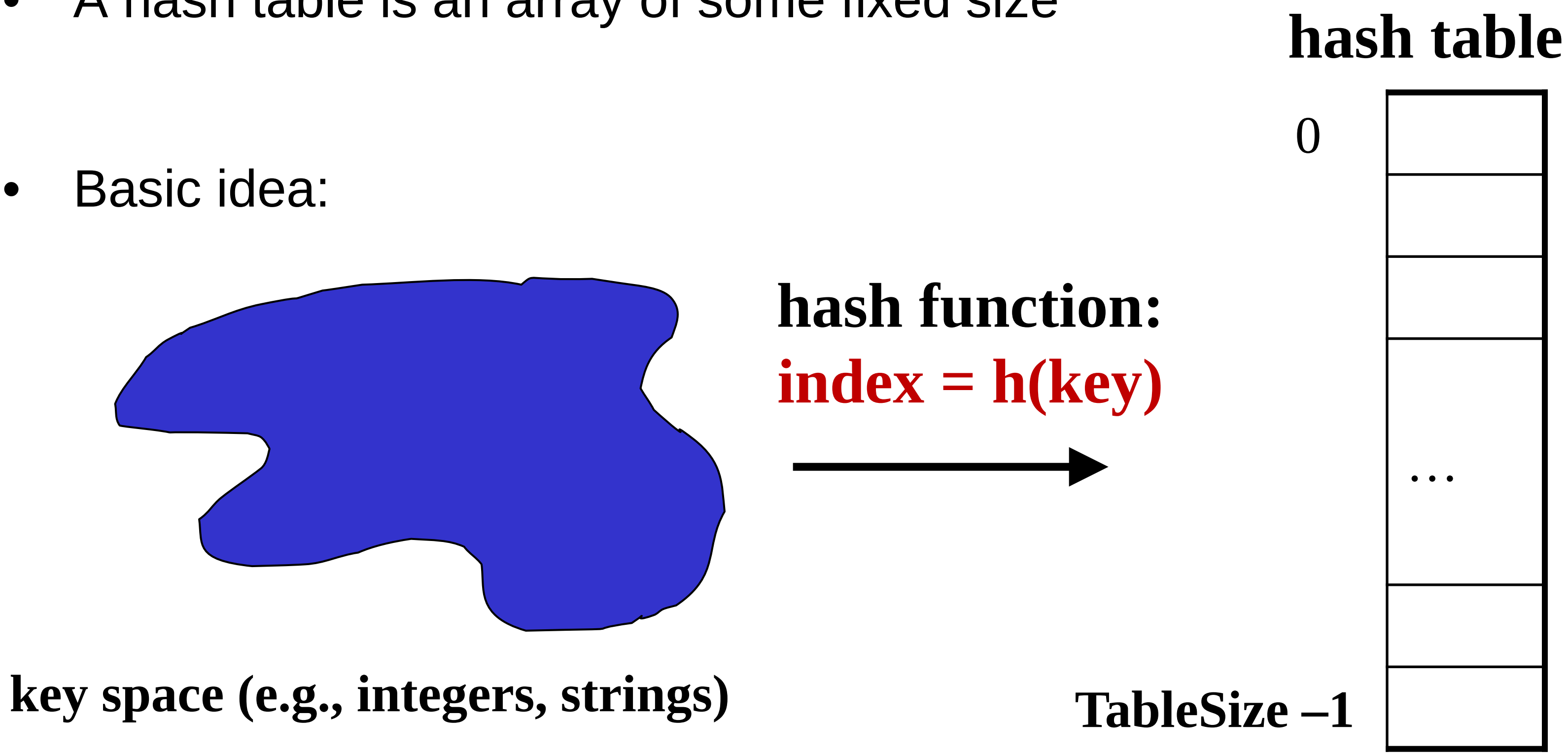
	`insert`	`find`	`delete`
• Unsorted linked-list	$O(1)$	$O(n)$	$O(n)$
• Unsorted array	$O(1)$	$O(n)$	$O(n)$
• Sorted linked list	$O(n)$	$O(n)$	$O(n)$
• Sorted array	$O(n)$	$O(\log n)$	$O(n)$
• *Balanced* tree	$O(\log n)$	$O(\log n)$	$O(\log n)$
• Magic array	$O(1)$	$O(1)$	$O(1)$

Sufficient "magic":

- Use key to compute array index for an item in $O(1)$ time
- Have a different index for every item

Hash Tables

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



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 **`findMin`**, **`findMax`**, **`predecessor`**, and **`successor`** go from $O($**`log`** $n)$ to $O(n)$, **`printSorted`** from $O(n)$ to $O(n$ **`log`** $n)$
    - Why your textbook considers this to be a different ADT

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*)

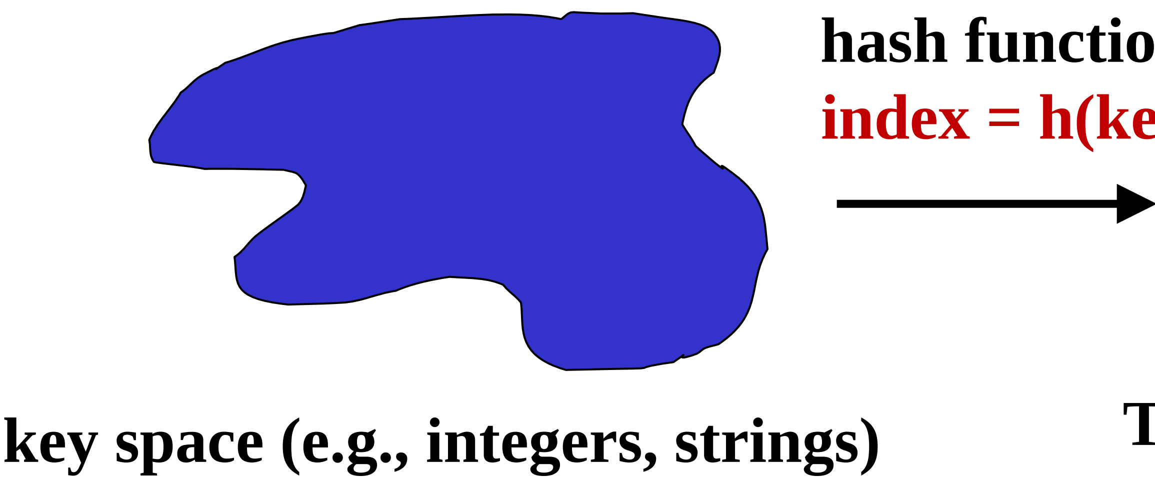
Many dictionaries have this property

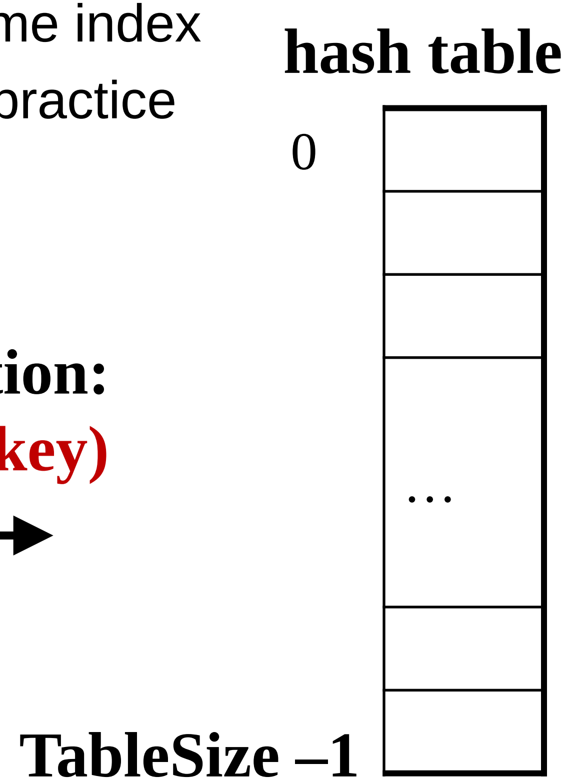
- Compiler: All possible identifiers allowed by the language vs. those used in some file of one program
- Database: All possible student names vs. students enrolled
- AI: All possible chess-board configurations vs. those considered by the current player
- …

Hash functions

An ideal hash function:

- 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 lecture





Collisions

key1

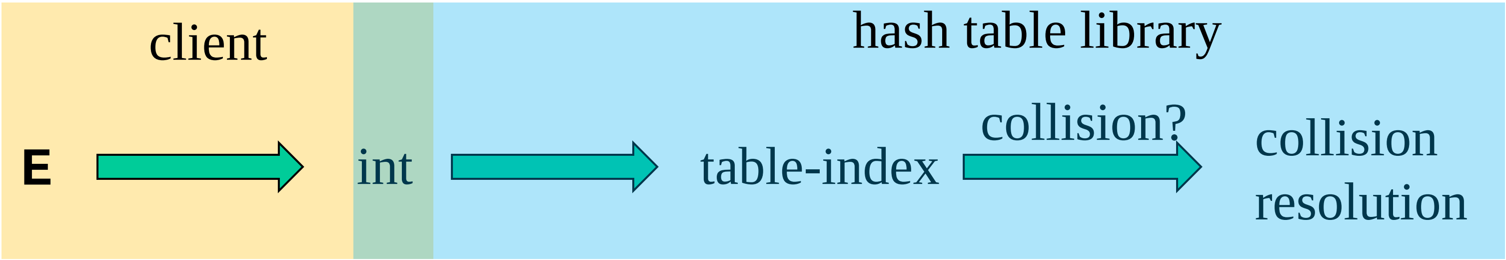
hash to same index

key2

Who hashes what?

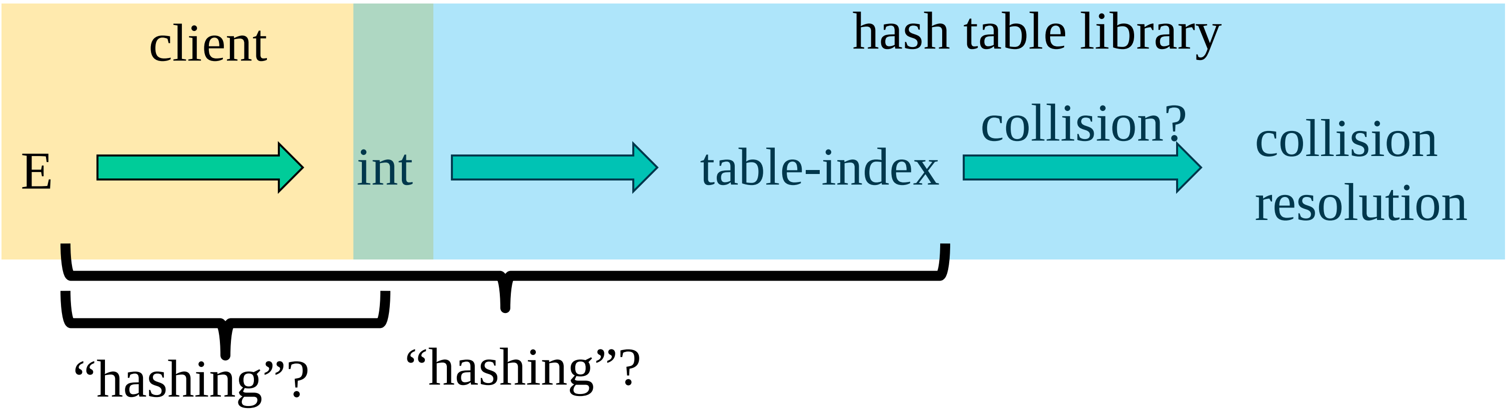
- Hash tables can be generic
  - To store elements of type **E**, we just need **E** to be:
    1. *Hashable*: convert any **E** to an **`int`**
    2. *Comparable*: order any two **E** (**only when dictionary**)

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



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 the two most common things to hash: ints and strings

- For objects with several fields, 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
  - Bad idea(?): Use only first name
  - Good idea(?): Use only middle initial
  - Admittedly, what-to-hash-with is often unprincipled ☹

Hashing integers

- key space = integers
- Simple hash function:
  `h(key) = key % TableSize`
  - Client: **`f(x) = x`**
  - Library **`g(x) = x % TableSize`**
  - Fairly fast and natural
- Example:
  - **`TableSize`** = 10
  - Insert 7, 18, 41, 34, 10
  - (As usual, ignoring data “along for the ride”)

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

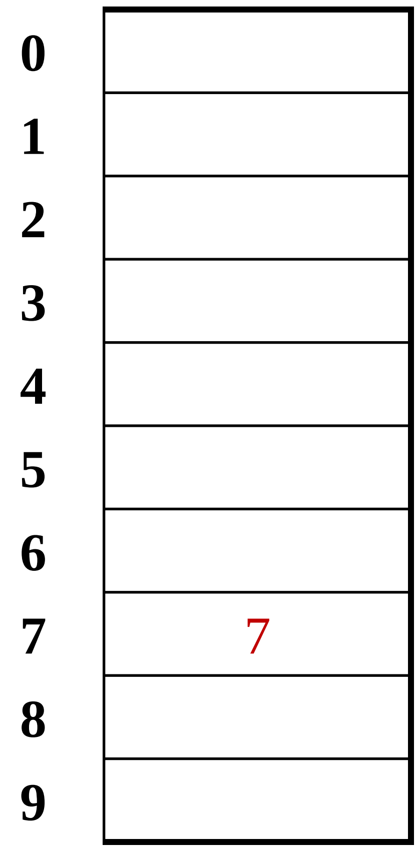
Hashing integers

- key space = integers

- Simple hash function:

  `h(key) = key % TableSize`
  - Client: **`f(x) = x`**
  - Library **`g(x) = x % 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`

  – Client: **`f(x) = x`**

  – Library **`g(x) = x % TableSize`**

  – Fairly fast and natural

- Example:

  – **`TableSize`** = 10

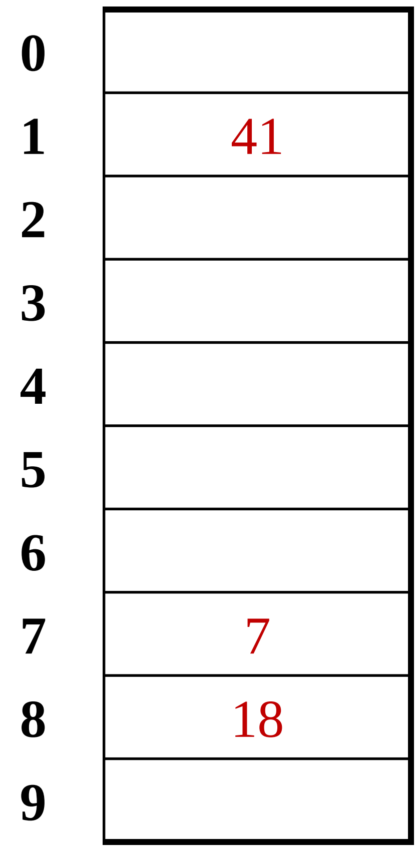
  – Insert 7, 18, 41, 34, 10

  – (As usual, ignoring data "along for the ride")

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

Hashing integers

- key space = integers
- Simple hash function:
  `h(key) = key % TableSize`
  – Client: **`f(x) = x`**
  – Library **`g(x) = x % 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`

  – Client: **`f(x) = x`**

  – Library **`g(x) = x % 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`
  - Client: **`f(x) = x`**
  - Library **`g(x) = x % 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	

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"
  - One collision-handling strategy does *provably* well with prime table size

Back to the client

- 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: $s_i \in [0,52]$ or $s_i \in [0,256]$ or $s_i \in [0,2^{16}]$)
  - Some choices: Which avoid collisions best?

1. h(K) = s_0 % TableSize

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

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

Specializing hash functions

Thought question:

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

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
3. When smashing two hashes into one hash, use bitwise-xor
4. Rely on expertise of others; consult books and other resources
5. If keys are known ahead of time, choose a *perfect hash* that maps distinct keys to distinct integers with no collisions.

Hashing and comparing

- Need to emphasize a critical detail:
  - We initially *hash* key **E** to get a table index
  - To check an item is what we are looking for, *compareTo* **E**
    - Does it have an equal key?

- So a hash table needs a hash function and a comparator
  - The Java library uses a more 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?
- Why is this up to the client?
- So *always* override **`hashCode`** *correctly* if you override **`equals`**
  - Many libraries use hash tables on your objects

Example

```
class MyDate {
  int month;
  int year;
  int day;

  boolean equals(Object otherObject) {
     if(this==otherObject) return true; // common?
     if(otherObject==null) return false;
     if(getClass()!=other.getClass()) return false;
     return month = otherObject.month
           && year = otherObject.year
           && day = otherObject.day;
  }
}
```

Example

```
class MyDate {
  int month;
  int year;
  int day;

  boolean equals(Object otherObject) {
     if(this==otherObject) return true; // common?
     if(otherObject==null) return false;
     if(getClass()!=other.getClass()) return false;
     return month = otherObject.month
            && year = otherObject.year
            && day = otherObject.day;
  }
  // wrong: must also override hashCode!
}
```

Conclusions and notes on hashing

- The hash table is one of the most important data structures
  - Supports only **`find`**, **`insert`**, and **`delete`** efficiently
  - Have to search entire table for other operations

- Important to use a good hash function

- Important to keep hash table at a good size

- Side-comment: hash functions have uses beyond hash tables
  - Example: Cryptography

- Big remaining topic: Handling collisions