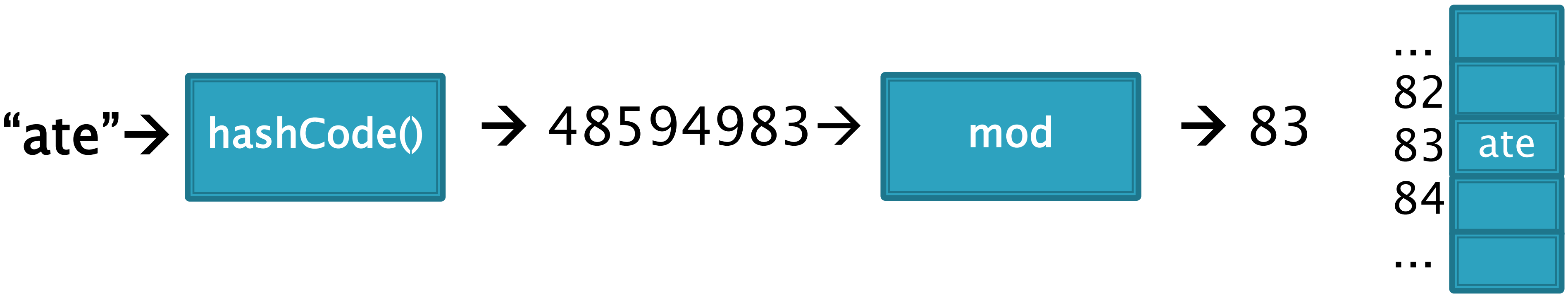
CSSE 230

Hash table basics

After today, you should be able to…
…explain how hash tables perform insertion in amortized O(1) time given enough space



Announcements and questions

1. EditorTrees M1 discussion
   Unit test points and commits
2. HW6 discussion
3. Test Tuesday, 7 pm
   I have all programming assignment solutions printed in my office if you want to check and discuss.
4. Look at HW7
5. Questions on test or HW6?

Hashing

Efficiently putting 5 pounds of data in a 20 pound bag

Reminder: sets hold unique items

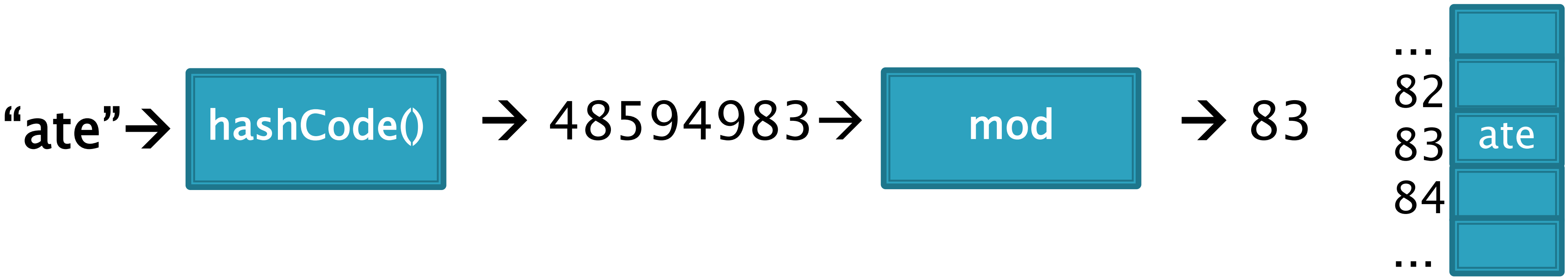
- **Implementation choices:**
  - **TreeSet** (and TreeMap) uses a balanced tree: O(log n)
    - Uses a red-black tree
  - **HashSet** (and HashMap) uses a hash table: amortized O(1) time
- Related: maps allow insertion, retrieval, and deletion of items by *key*:

  Since keys are unique, they form a set.

  The values just go along for the ride.

  We'll focus on sets.

Big ideas of hash tables

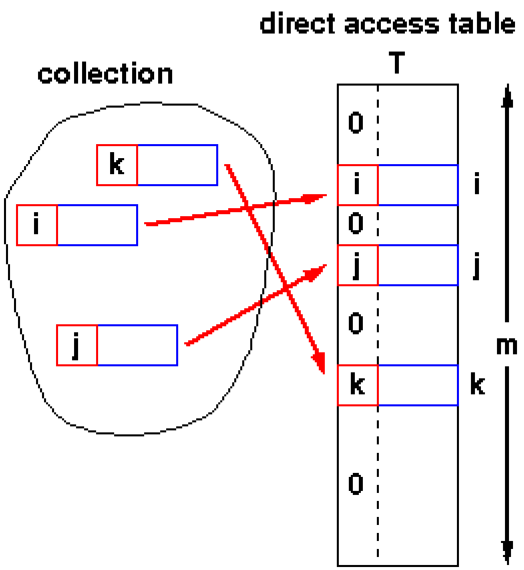


1. The underlying storage?
   Growable array

2. Calculate the index to store an item **from the item itself.** How?
   Hashcode. Fast but un-ordered.

3. What if that location is already occupied with another item?
   Collision. Two methods to resolve

Direct Address Tables



- Array of size **m**
- **n** elements with unique keys
- If n ≤ m, then use the key as an array index.
  - Clearly O(1) lookup of keys

- Issues?
  - Keys must be unique.
  - Often the range of potential keys is much larger than the storage we want for an array
    - Example: RHIT student IDs vs. # Rose students

Diagram from John Morris, University of Western Australia

We attempt to create unique keys by applying a .hashCode() function …

key → hashCode() → integer

Objects that are **`.equals()`**
MUST have the same **hashCode** values
A good hashCode() also
is **fast** to calculate and
distributes the keys, like:

hashCode("ate")= 48594983
hashCode("ape")= -76849201 (can be negative if overflows)
hashCode("awe") = 14893202

...and then take it mod the table size (m) to get an index into the array.

- Example: if m = 100:

hashCode("ate")= 48594983 → 83
hashCode("ape")= -76849201 → 46*
hashCode("awe") = 1489036 → 36

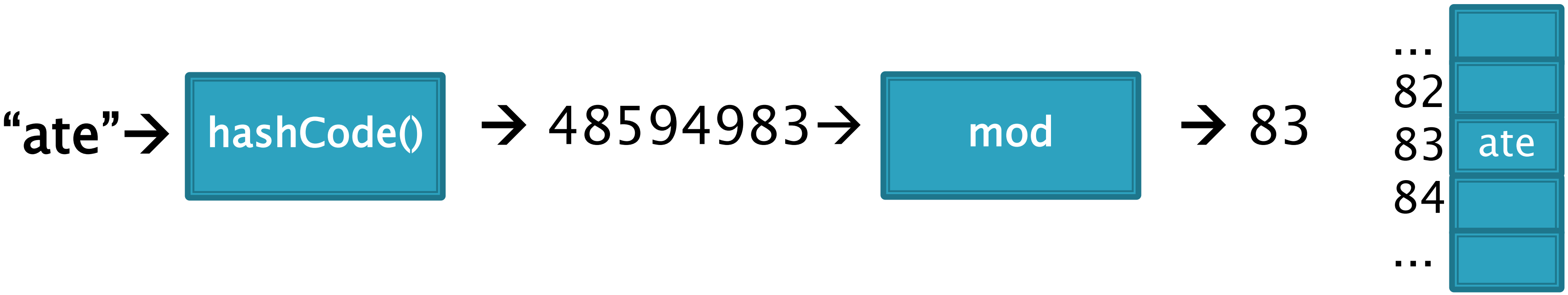
mod

*Note: since the hashCode is an integer, it might be negative, and negative numbers have negative remainders.

Trick: If it is negative, add Integer.MAX_VALUE to make it positive before you mod.

Index calculated from the object itself, not from a comparison with other objects

- How Java's **hashCode() is used:**



  - Unless this position is already occupied

a "collision"

Some hashCode() implementations

- Default if you inherit `Object`'s: memory location
- Many JDK classes override **hashCode()**
  - Integer: the value itself
  - Double: XOR first 32 bits with last 32 bits
  - String: we'll see shortly!
  - Date, URL, ...
- Custom classes should override hashCode()
  - Use a combination of **final** fields.
  - If key is based on mutable field, then the hashcode will change and you will lose it!
  - People usually use strings if possible.

A simple hash function for Strings is a function of every character

```
// This could be in the String class
public static int hash(String s) {
 int total = 0;
 for (int i=0; i<s.length(); i++)
   total = total + s.charAt(i);
 return total;
}
```

- Advantages?
- Disadvantages?

A better hash function for Strings uses place value

```
// This could be in the String class
public static int hash(String s) {
 int total = 0;
 for (int i=0; i<s.length(); i++)
   total = total*256 + s.charAt(i);
 return total;
}
```

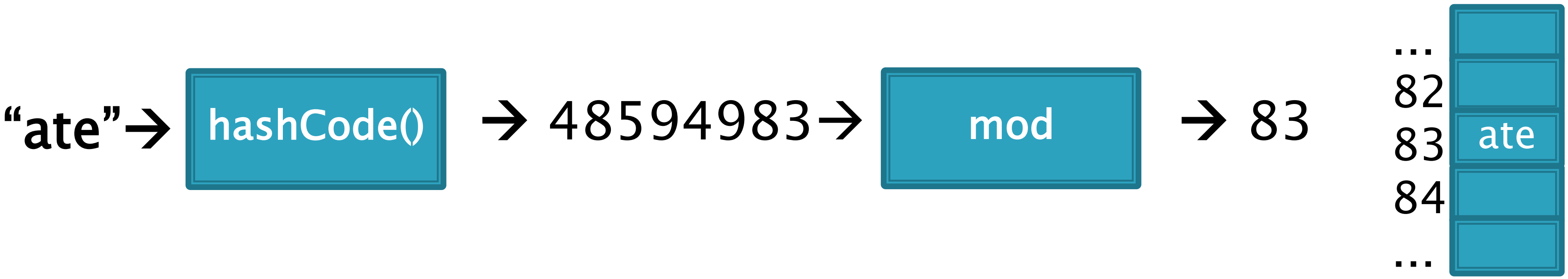
- Spreads out the values more, and anagrams not an issue.
- What about overflow during computation?
  - What happens to first characters?

A better hash function for Strings uses place value with a base that's prime

```
// This could be in the String class
public static int hash(String s) {
 int total = 0;
 for (int i=0; i<s.length(); i++)
   total = total*31 + s.charAt(i);
 return total;
}
```

- Spread out, anagrams OK, overflow OK.
- This is **String**'s `hashCode()` method.
- The $(x = 31x + y)$ pattern is a good one to follow.

Collisions are inevitable

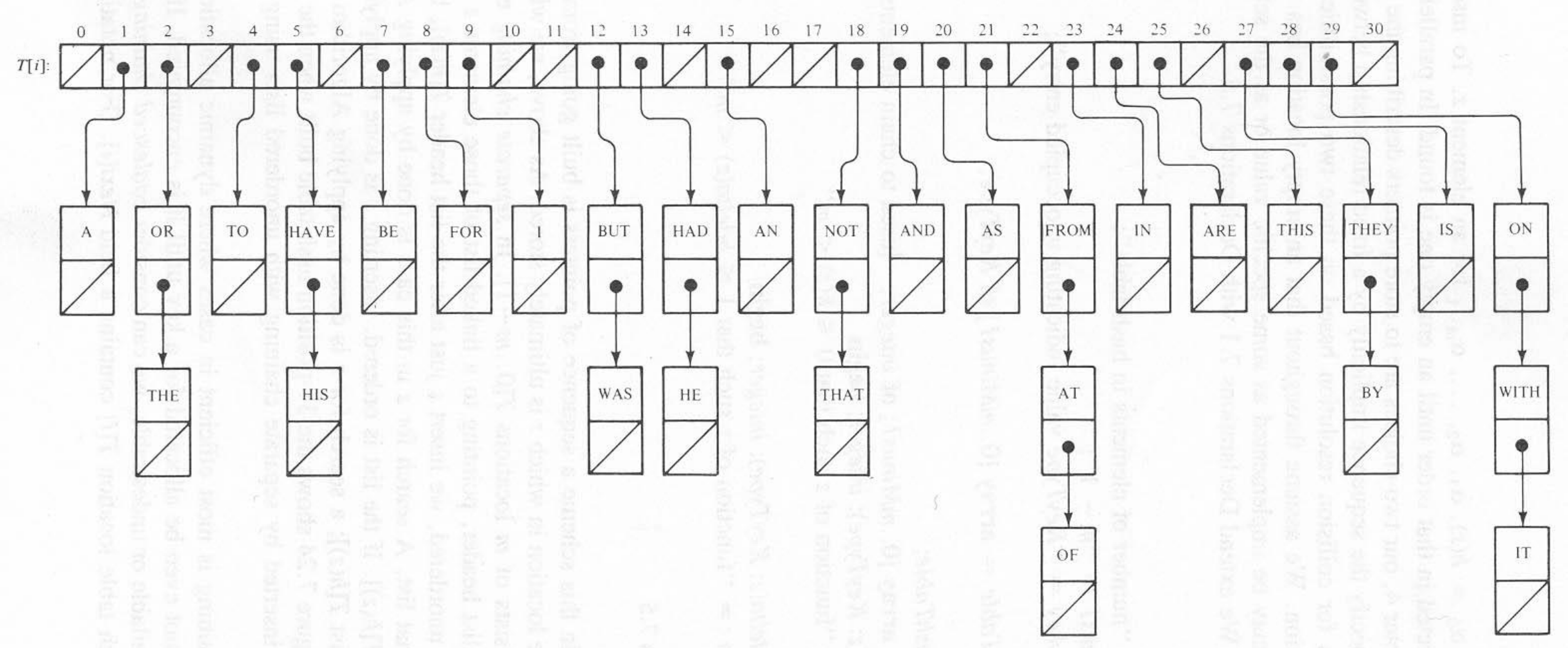


- A good hashcode distributes keys evenly, but collisions will still happen
- hashCode() are ints → only ~4 billion unique values.
  - How many 16 character ASCII strings are possible?
- If n is small, tables should be much smaller
  - mod will cause collisions too!
- Solutions:
  - Chaining
  - Probing (Linear, Quadratic)

Separate chaining: an array of linked lists

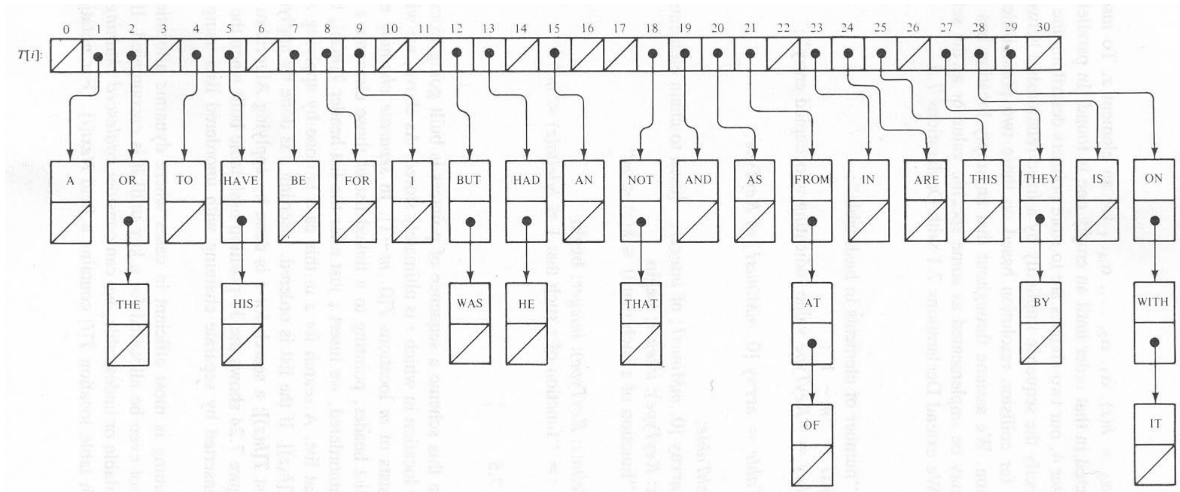
Grow in another direction

Examples: .get("at"), .get("him), (hashcode=18), .add("him"), .delete("with")



Java's **HashMap** uses chaining and a table size that is a power of 2.

Runtime of hashing with chaining depends on the load factor



m array slots,
n items.
Load factor, $\lambda = n/m$.

Runtime = $O(\lambda)$

Space-time trade-off

1. If m constant, then this is O(n). Why?

2. If keep m~0.5n (by doubling), then this is amortized O(1). Why?

Alternative: Store collisions in other array slots.

- No need to grow in second direction
- No memory required for pointers
  - Historically, this was important!
  - Still is for some data…
- Will still need to keep load factor ($\lambda = n/m$) low or else collisions degrade performance
  - We'll grow the array again

Collision Resolution: Linear Probing

- Probe H (see if it causes a collision)
- Collision? Also probe the next available space:
  - Try H, H+1, H+2, H+3, ...
  - Wraparound at the end of the array
- Example on board: .add() and .get()

- Problem: Clustering

- Animation:
  - http://www.cs.auckland.ac.nz/software/AlgAnim/hash_tables.html

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

Figure 20.4

Linear probing hash table after each insertion

Good example of clustering and wraparound

	After insert 89	*After insert 18*	*After insert 49*	*After insert 58*	*After insert 9*
0			49	49	49
1				58	58
2					9
3					
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Linear probing efficiency also depends on load factor, $\lambda = n/m$

- For probing to work, $0 \le \lambda \le 1$.

- For a given λ, what is the expected number of probes before an empty location is found?

Rough Analysis of Linear Probing

- Assume all locations are equally likely to be occupied, and equally likely to be the next one we look at.
- Then the probability that a given cell is full is λ and probability that a given cell is empty is $1-\lambda$.
- What's the expected number?

$$\sum_{p=1}^{\infty} \lambda^{p-1}(1-\lambda)p = \frac{1}{1-\lambda}$$

Better Analysis of Linear Probing

- Clustering!
  - Blocks of occupied cells are formed
  - Any collision in a block makes the block bigger
- Two sources of collisions:
  - Identical hash values
  - Hash values that hit a cluster
- Actual average number of probes for large λ:

$$\frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$$

For a proof, see Knuth, The Art of Computer Programming, Vol 3: Searching Sorting, 2nd ed, Addision-Wesley, Reading, MA, 1998.

Why consider linear probing?

- Easy to implement
- Works well when load factor is low
  - In practice, once $\lambda > 0.5$, we usually **double the size of the array** and rehash
  - This is more efficient than letting the load factor get high

To reduce clustering, probe farther apart

- Reminder: Linear probing:
  - Collision at H? Try H, H+1, H+2, H+3,...
- New: **Quadratic** probing:
  - Collision at H? Try H, $H+1^2$. $H+2^2$, $H+3^2$, ...
  - Eliminates primary clustering. "Secondary clustering" isn't as problematic

Quadratic Probing works best with low λ and prime m

- Choose a prime number for the array size, m
- Then if $\lambda \leq 0.5$:
  - Guaranteed insertion
    - If there is a "hole", we'll find it
  - **So no cell is probed twice**

- **Can show with m=17, H=6.**

For a proof, see Theorem 20.4:

Suppose that we repeat a probe before trying more than half the slots in the table

See that this leads to a contradiction

Contradicts fact that the table size is prime

Quadratic probing analysis

- No one has been able to analyze it!
- Experimental data shows that it works well
  - Provided that the array size is prime, and $\lambda < 0.5$

Summary:
Hash tables are fast for some operations

Structure	insert	Find value	Find max value
Unsorted array			
Sorted array			
Balanced BST			
Hash table			

- Finish the quiz.
- Then check your answers with the next slide

Answers:

Structure	insert	Find value	Find max value
Unsorted array	Amortized $\theta(1)$	$\theta(n)$	$\theta(n)$
Sorted array	$\theta(n)$	$\theta(\log n)$	$\theta(1)$
Balanced BST	$\theta(\log n)$	$\theta(\log n)$	$\theta(\log n)$
Hash table	Amortized $\theta(1)$	$\theta(1)$	$\theta(n)$

Review: discuss with a partner

- Why use 31 and not 256 as a base in the String hash function?
- Consider chaining, linear probing, and quadratic probing.
  - What is the purpose of all of these?
  - For which can the load factor go over 1?
  - For which should the table size be prime to avoid probing the same cell twice?
  - For which is the table size a power of 2?
  - For which is clustering a major problem?
  - For which must we grow the array and rehash every element when the load factor is high?

In practice

- Constants matter!
- 727MB data, ~190M elements
  - Many inserts, followed by many finds
  - Microsoft's C++ STL

Structure	build (seconds)	Size (MB)	100k finds (seconds)
Hash map	22	6,150	24
Tree map	114	3,500	127
Sorted array	17	727	25

- Why?
- Sorted arrays are nice **if** they don't have to be updated frequently!
- Trees still nice when interleaved insert/find