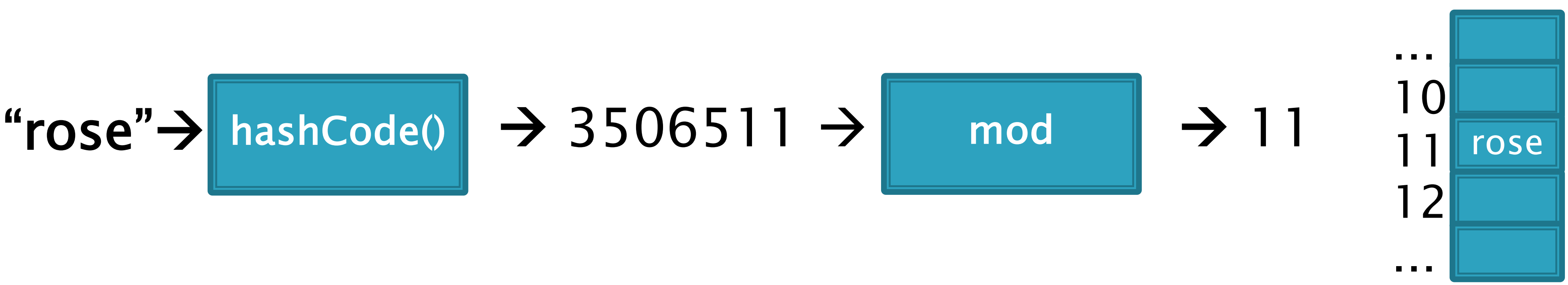
CSSE 230

Hash table basics

How can hash tables perform both **contains()** in O(1) time and **add()** in amortized O(1) time, given enough space?



Hashing

Efficiently putting 5 pounds of data in a 20 pound bag

Reminder: sets hold unique items

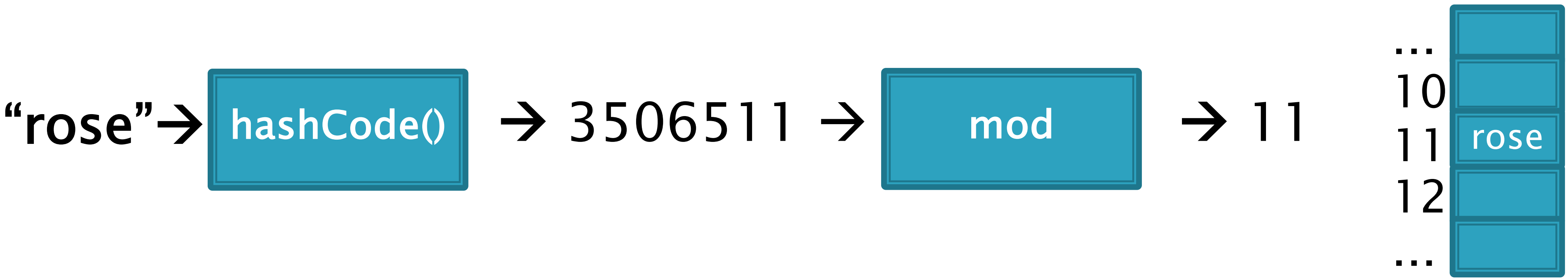
- **Implementation choices**:
  - **TreeSet** (and TreeMap) uses a balanced tree: O(log n)
    - Uses a red-black tree
  - **HashSet** (and HashMap) uses a hash table: amortized O(1) time
- Related: maps allow insertion, retrieval, and deletion of items by *key*:

  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

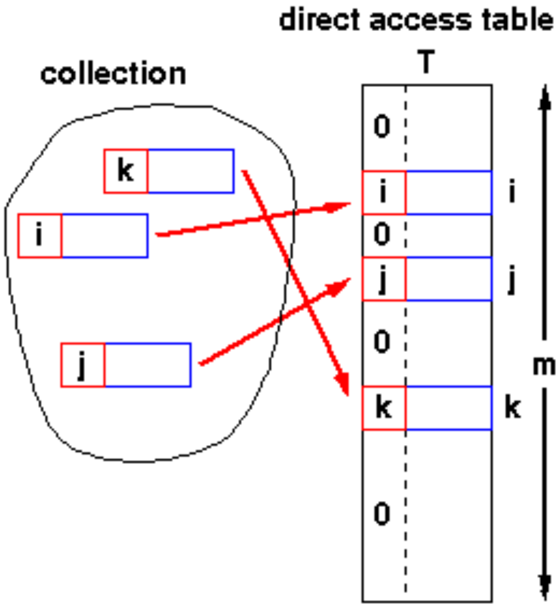
2. Calculate the index to store an item **based on the item itself.** How?

   Hashcode. Fast but un-ordered.

3. What if that location is already occupied with another item?

   Collision. Two approaches to resolve this

Introductory Idea: Direct Address Tables



- n elements with unique keys in range [0,k)
- Array of size m
- k < m, then use the key as an array index.
  - Clearly O(1) lookup of keys

- Main Issue?
  - The range of potential keys [0,k) is usually 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

More Practical: Hash Tables

Three step process used for accessing hash tables:

1. Transform *key* into an integer x
2. Use a calculation on x to generate a integer y in the range $[0..m-1]$, where m = array capacity
3. Use y to index into the hash table array, i.e., `hTable[y]`

Step 3

“rose”→ Step 1 → 3506511 → Step 2 → 11

...
10
11 rose
12
...

- Step 1 is handled by Java’s hashCode() method
  - Javadoc prototype for Object’s hashCode() method:

```
int hashCode()
```
Returns a hash code value for the object

- Step 2 is often implemented by: $y = x \bmod m$
  - Using *mod* operation is called the ‘Division Method’
  - ‘Multiplication Methods’ also exist

Step 1. hashCode()

key → hashCode() → integer

Required property of Java's hashCode() method:

- Given x.equals(y), i.e., x is equal to y, then x.hashCode() == y.hashCode()

Desirable properties:

- Should be **fast** to calculate
- Should produce integers that have a nice uniform distribution

"rose".hashCode()= 3506511
"hulman".hashCode()= -1206158341 (can be negative if overflows)
"institute".hashCode() = 36682261

Step 2. Convert int to index

- Example: if $m = 100$:

hashCode("rose") = 3506511		→11
hashCode("hulman") = -1206158341	mod m	→07*
hashCode("institute") = 36682261		→61

* Note: since the hashCode is an integer, it might be negative…

- If it is negative, add Integer.MAX_VALUE + 1 to make it positive before you mod. (Same as ANDing with 0x7fffffff, or removing sign bit from two's complement)
- This mimics what's actually done in practice: when m is a power of 2, say 2^k, we can just truncate, keeping the last k bits (instead of taking mod m). Sign bit is lost.

Step 3. Access array[index]

- Insert element at array[index]

"rose"→ hashCode() → 3506511 → mod → 11

…	
10	
11	rose
12	
…	

  - Unless this position is already occupied

a "collision"

Some hashCode() implementations

- Default if you inherit `Object`'s: memory location (platform-specific, actually)

- 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!
  - Developers often use strings when feasible

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

```
class String {
  public int hashCode() {
    int total = 0;
    for (int i = 0; i < this.length(); i++)
      total = total + this.charAt(i);
    return total;
  }
}
```

- Advantages?
- Disadvantages?

A better hashCode function for Strings uses place value

```
class String {
  public int hashCode() {
    int total = 0;
    for (int i = 0; i < this.length(); i++)
      total = total*256 + this.charAt(i);
    return total;
  }
}
```

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

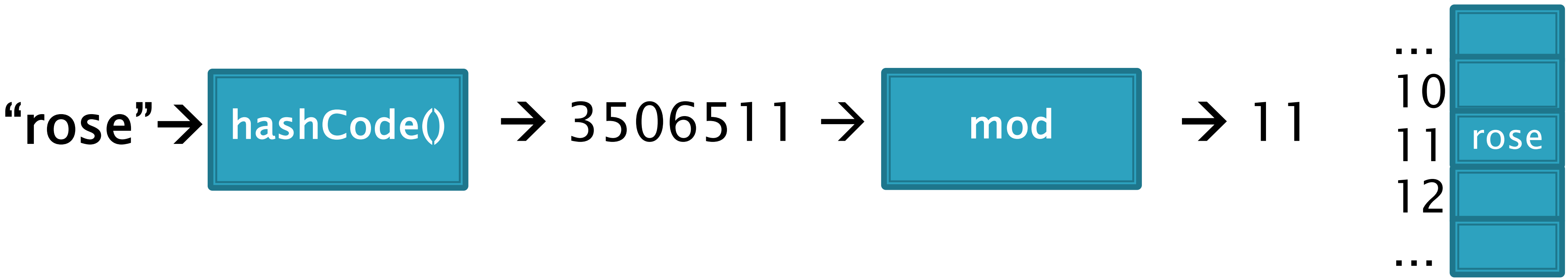
A better hashCode function for Strings uses place value with a base that's prime

```
class String {
  public int hashCode() {
    int total = 0;
    for (int i = 0; i < this.length(); i++)
      total = total*31 + this.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.

- See https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#hashCode--

Collisions are inevitable



- A good hashCode operation distributes keys uniformly, but collisions will still happen

- hashCode() are ints → only ~4 billion unique values.
  - How many 16 character ASCII strings are possible?

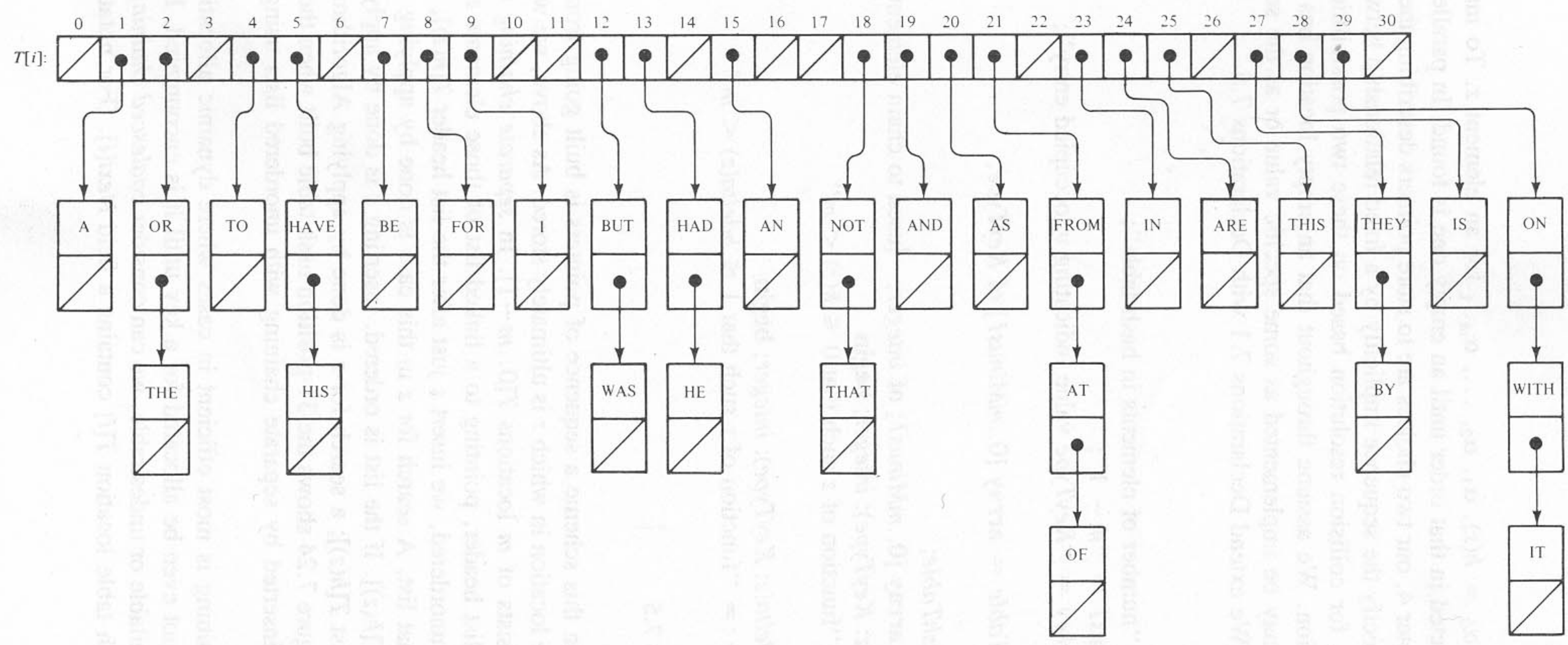
- If n is small, tables should be much smaller
  - mod will cause collisions too!

- Solutions:
  - Chaining
  - Probing (Linear, Quadratic)

Separate chaining: an array of linked lists

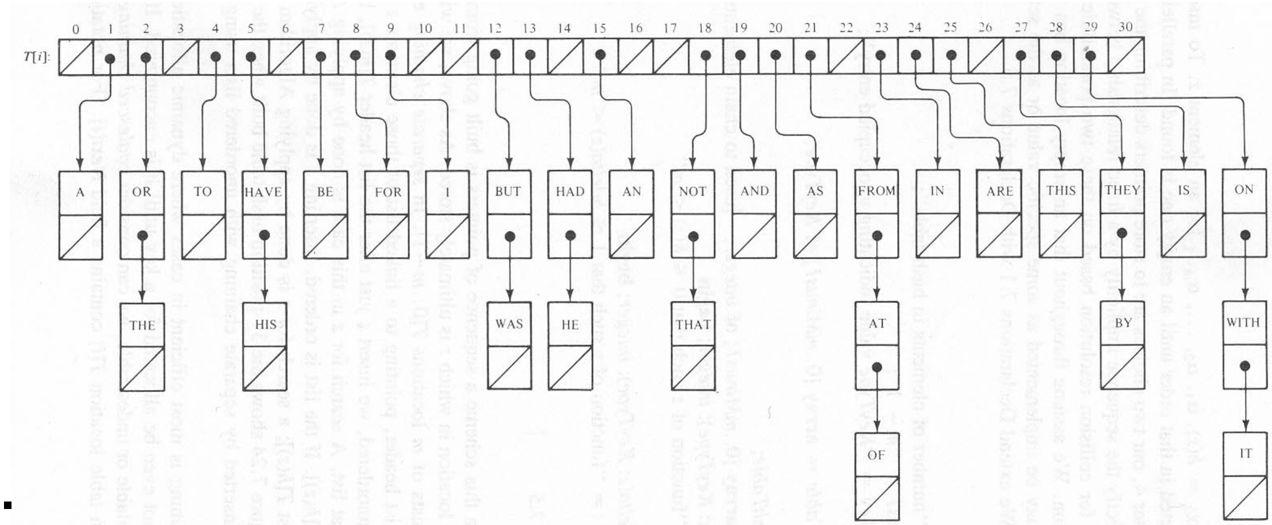
Grow in another direction

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, $\lambda = n/m$.

Average length of chain is $O(\lambda)$, so
Average runtime of search is $O(\lambda)$.

Space-time trade-off

1. If m constant, then this is O(n). Why?
2. If keep (say) $n \leq 0.75m$, by doubling m when appropriate, then this is O(1). Why?
3. Also, insertion/deletion is also amortized O(1)

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
  - Applet deprecated on most browsers
  - Moodle has a video captured from there
  - Or see next slide for a few freeze-frames.

Figure 20.4

Linear probing hash table after each insertion

Good example of clustering and wraparound

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

	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

Clustering Example