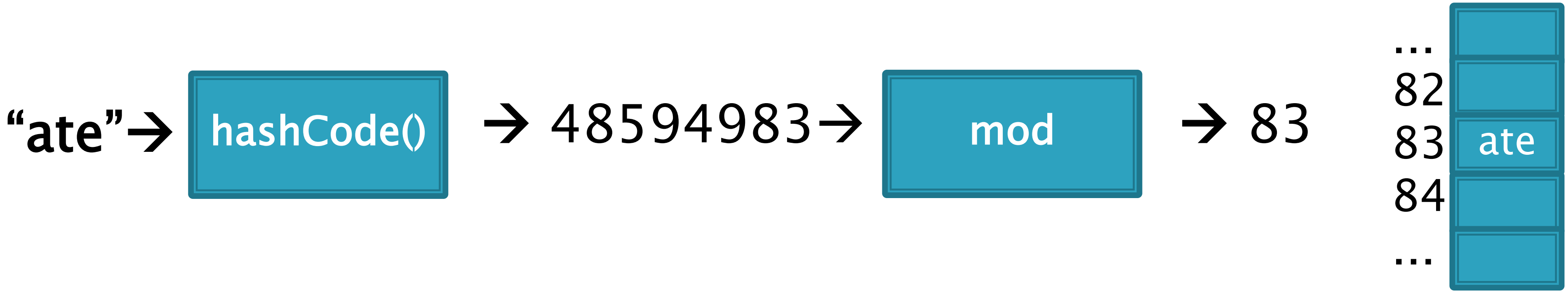
CSSE 230

Hash table basics

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



Reminder: Exam 2

- Topics: weeks 1-6
  - Reading, programs, in-class, written assignments.
  - Especially
    - Algorithm analysis in general
    - Binary trees, including BST, AVL, R/B, and threaded
      - Traversals and iterators, size vs. height, rank
    - Backtracking / Queens problem

- Questions on this or anything else course-related?

Hashing

Efficiently putting 5 pounds of data in a 20 pound bag

Big picture: a map gives dictionary storage

- Map: insertion, retrieval, and deletion of items by *key*: Examples:
  - Map<String, Integer> wordCounts;
  - Map<Integer, Student> students;
  - count = wordCounts.get(“best”);
  - students.add(56423302, new Student(…))l

- **Implementation choices:**
  - **TreeMap** uses a balanced tree
    - **TreeSet** is a **TreeMap** with no values
    - The BST assignment is an unbalanced TreeSet

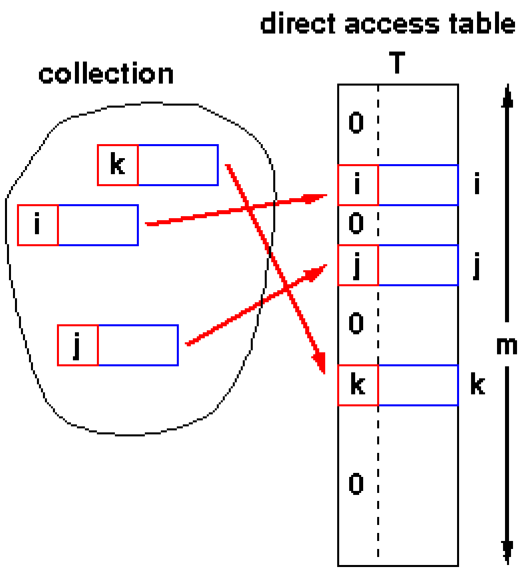
  - **HashMap** uses a hash table
    - **HashSet** is a **HashMap** with no values

A hash table is a very fast approach to dictionary storage

- Insertion and lookup are constant time!
  - With a good "hash function"
  - And large enough storage array
- Doesn't keep items ordered
  - So NOT for sorted data

On average

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
hashCode(“awe”) = 14893202

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

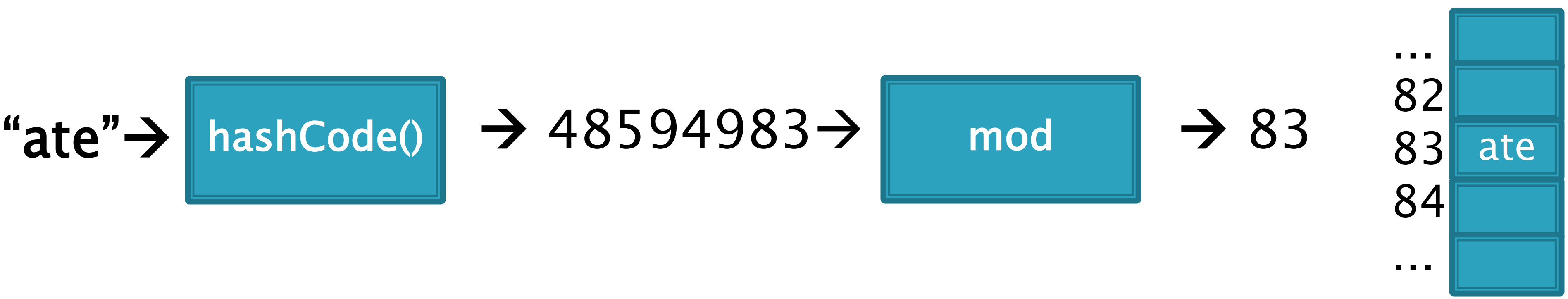
- Example: if m = 100:

hashCode("ate")= 48594983 → 83
hashCode("ape")= 76849201 → 01
hashCode("awe") = 1489036 → 36

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!

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 Math.abs(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 Math.abs(total);
}
```

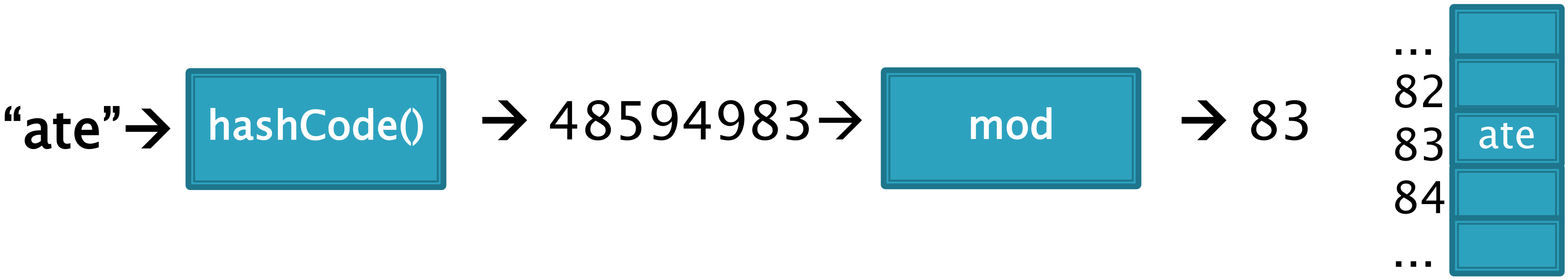
- Spreads out the values more, and anagrams not an issue.
- What about overflow during computation?
  - What happens to first characters?

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 Math.abs(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

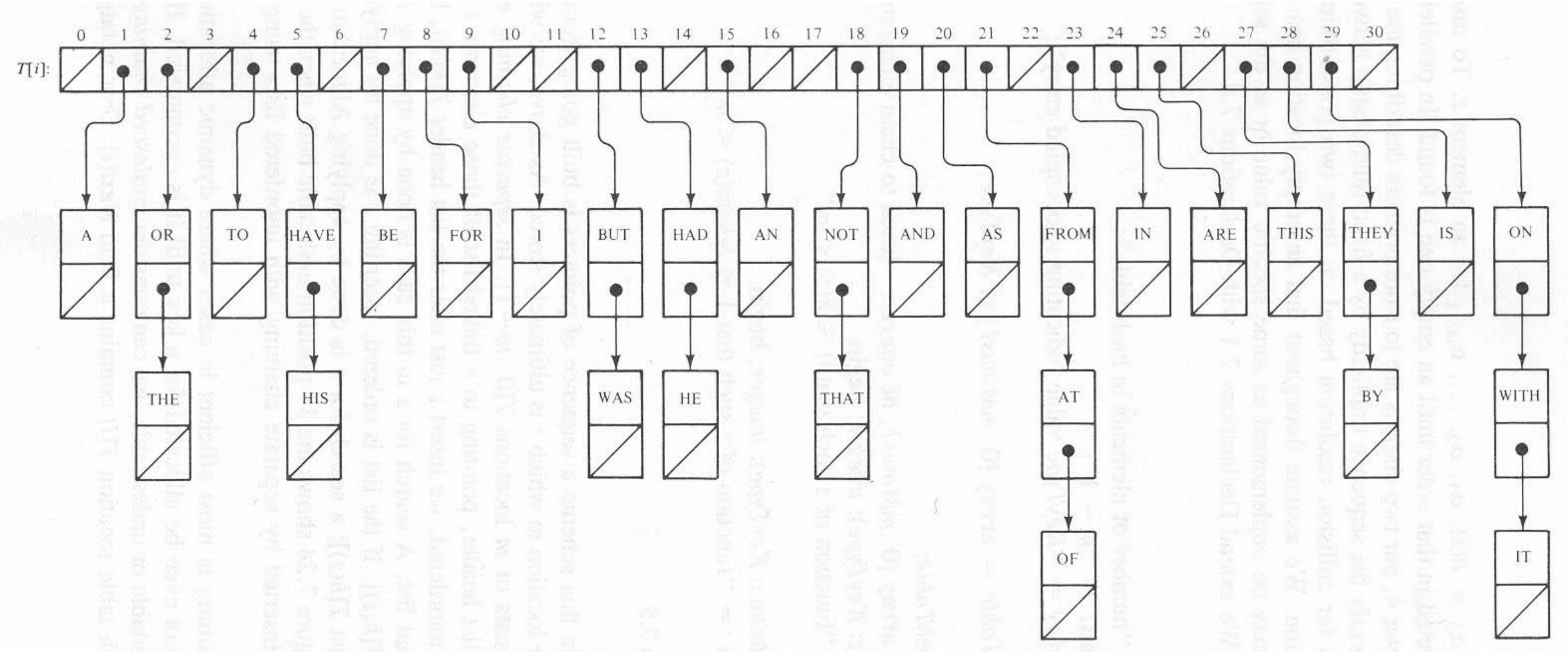


- A good hashcode distributes keys evenly, but collisions will still happen
- hashCode() are ints → only ~4 billion unique values.
  - How many 16 character ASCII strings are possible?
- If n is small, tables should be much smaller
  - mod will cause collisions
- Solutions:
  - Chaining
  - Probing (Linear,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

Separate chaining: an array of linked lists

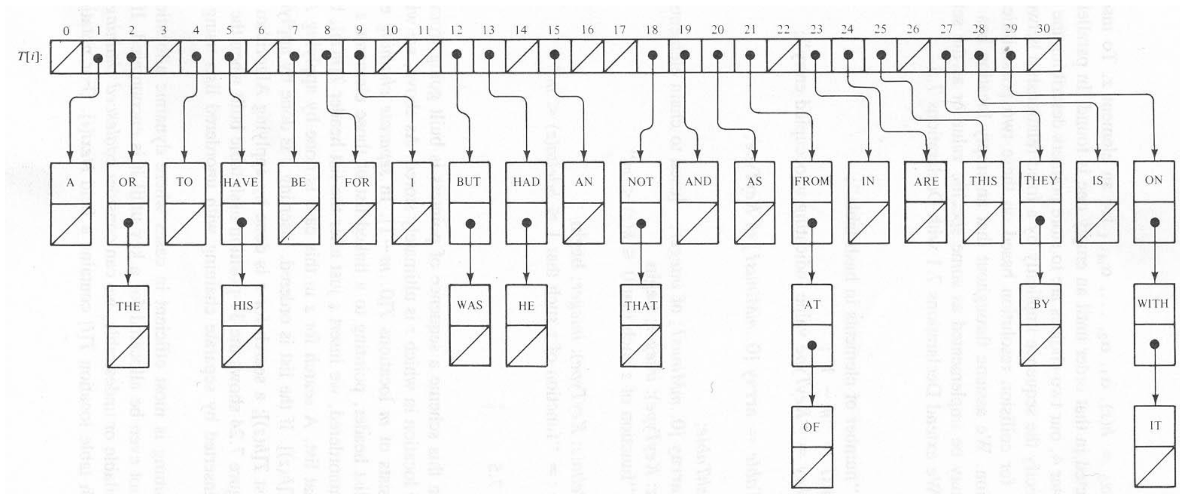
Easy to code
Easy to deal with collisions

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

Alternative: Store collisions in other array slots.

- No memory required for pointers
  - Historically, this was important!
- Will need to keep load factor ($\lambda = n/m$) low or else collisions degrade performance
- The logic is slightly more complicated
  - And uses some interesting math

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

- 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
```

	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

Figure 20.4

Linear probing hash table after each insertion

Good example of clustering and wraparound

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

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

- 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
- Simple code has fast run time per probe
- 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

- Linear probing:
  - Collision at H? Try H, H+1, H+2, H+3,...
- 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
  - **No cell is probed twice**

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 runs quickly if we implement it correctly

- **Use an algebraic trick to calculate next index**
  - Difference between successive probes yields:
    - Probe i location, $H_i = (H_{i-1} + 2i - 1) \% M$

1. Just use bit shift to multiply i by 2
   - probeLoc= probeLoc + (i << 1) - 1;

   …faster than multiplication

2. Since i is at most M/2, can just check:
   - if (probeLoc >= M)
     probeLoc -= M;

   …faster than mod

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 $\lambda < 0.5$

- If you are interested, you can do the optional HashSet exercise.
  - http://www.rose-hulman.edu/class/csse/csse230/201430/InClassExercises/
- This week's homework takes a couple questions from there.

Summary:
Hash tables are fast for some operations

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

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

Answers:

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