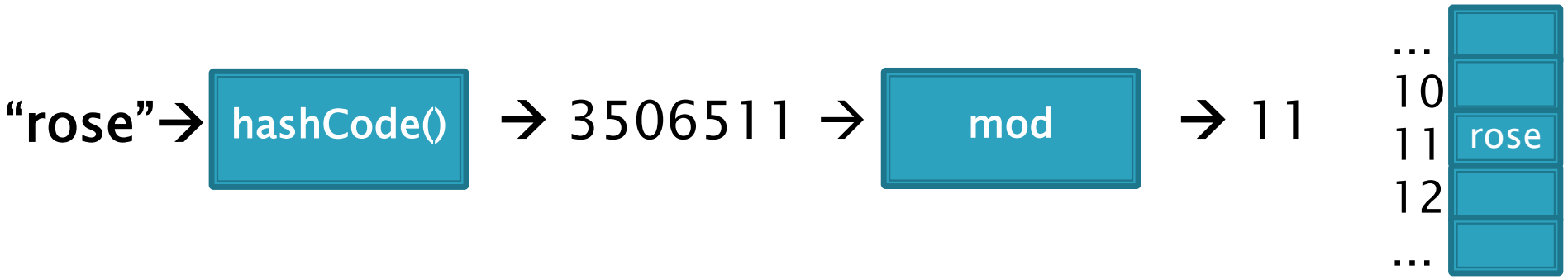
CSSE 230
Hash table basics

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

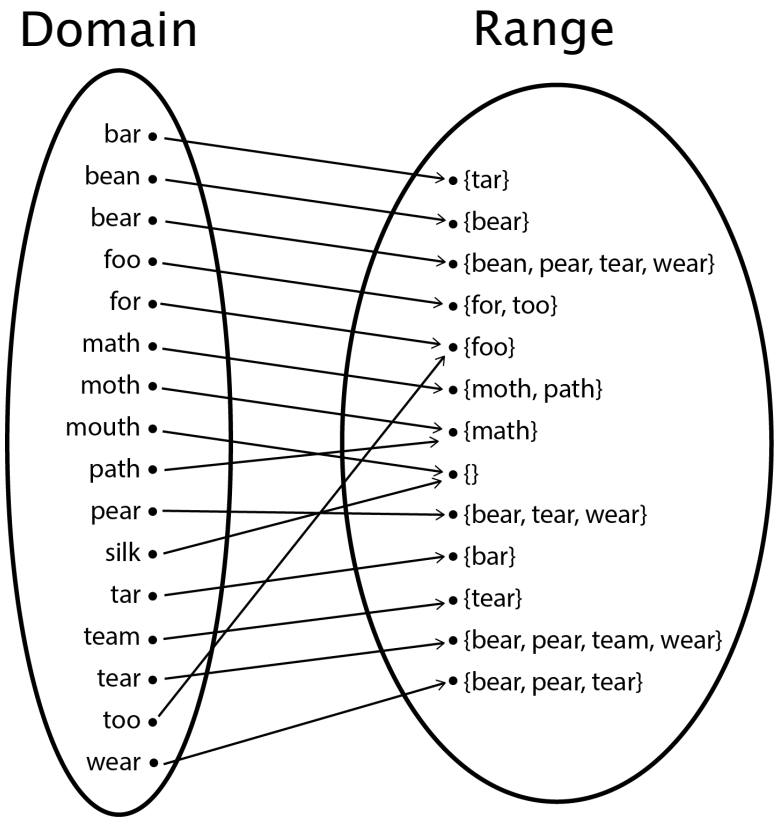
- **Implementation choices:**
  - **TreeSet** (and TreeMap) uses a balanced tree: O(log n)
    - Uses a red-black tree
  - **HashSet** (and HashMap) uses a hash table: amortized O(1) time

HashSet<E> – Method Summary (some, not all)

```
void clear()
boolean contains(Object o)
boolean isEmpty()
boolean add(E e)
boolean remove(Object o)
int size()
```

Related: map concept

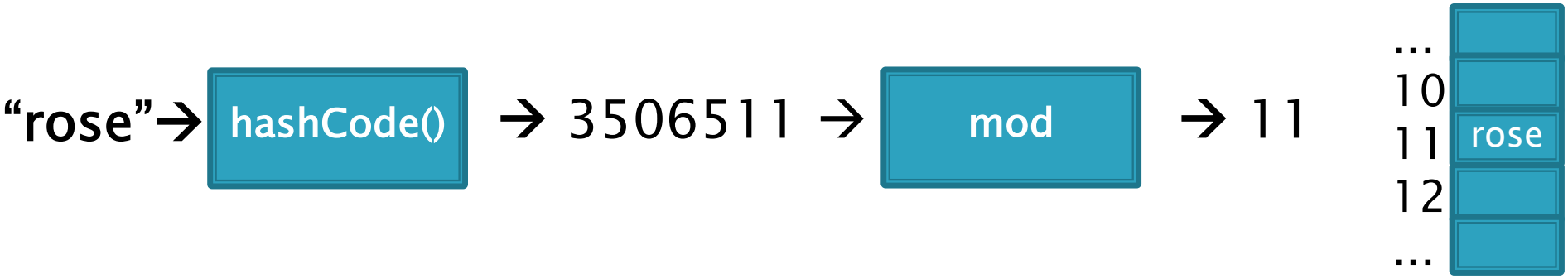
- Allows us to keep track of the mapping from domain to range values
- Map is a container:
  - insert(key, value)
  - delete(key)
  - lookUp(key)
- Domain contains *key* values
- This is *function*, so key values must be unique



HashMap<K,V> - Method Summary (some, not all)

```
void clear()
boolean containsKey(Object key)
V get(Object key)
boolean isEmpty()
V put(K key, V value)
V remove(Object key)
V replace(K key, V value)
int size()
```

Big ideas of hash tables

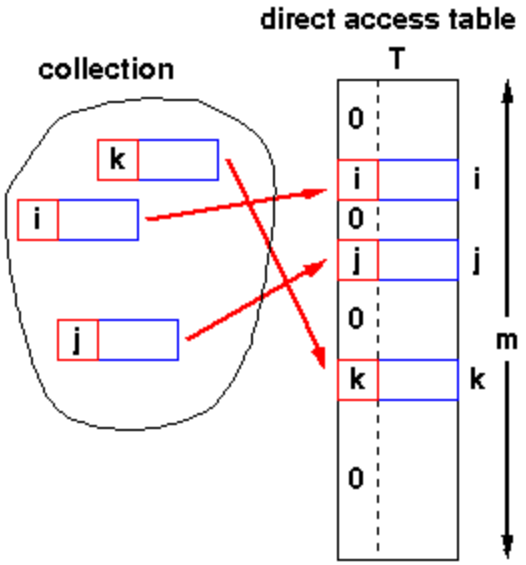


1. The underlying storage?
   Growable array

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

When Direct Address Tables are not feasible …

Three step process used for accessing hash tables:

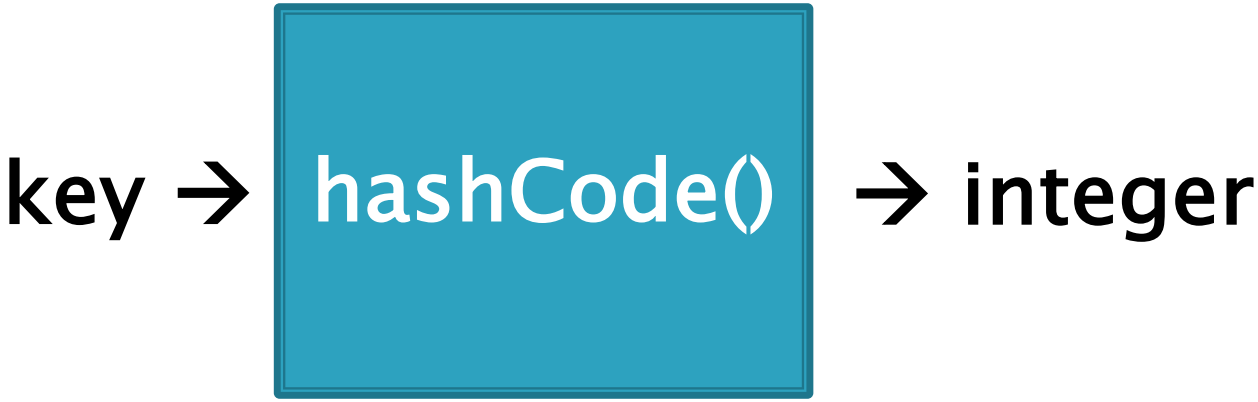
1. Transform *key* into an integer *X*
2. Use a calculation on *X* to generate a natural number *Y* in the range [0..*m*-1]
3. Use *Y* to index into the hash table array, i.e., `hTable[Y]`

- Step #1 is handled by Java's hashCode() method
- Step #2's *m* is the size of the hash table array
- Step #2 is often implemented by: $Y = X \bmod m$
  - Using *mod* operation is called the 'Division Method'
  - 'Multiplication Methods' also exist

Javadoc prototype for Object's hashCode() method:

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

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



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

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

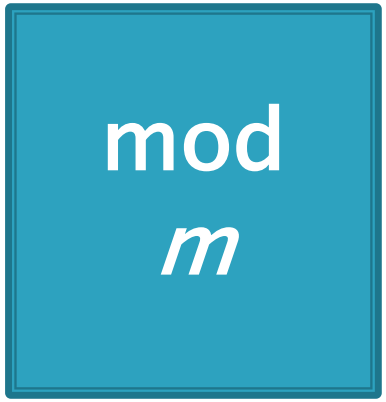
…and then take it mod the table size (m) to get an index into the array.

- Example: if $m = 100$:

hashCode("rose") = 3506511

hashCode("hulman") = -1206158341

hashCode("institute") = 36682261



→11

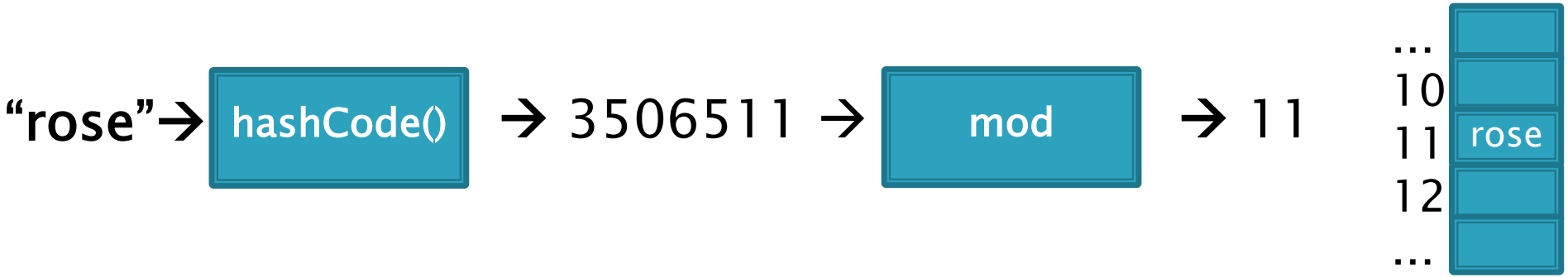
→07*

→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.

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 (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

```
// 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 hashCode 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?

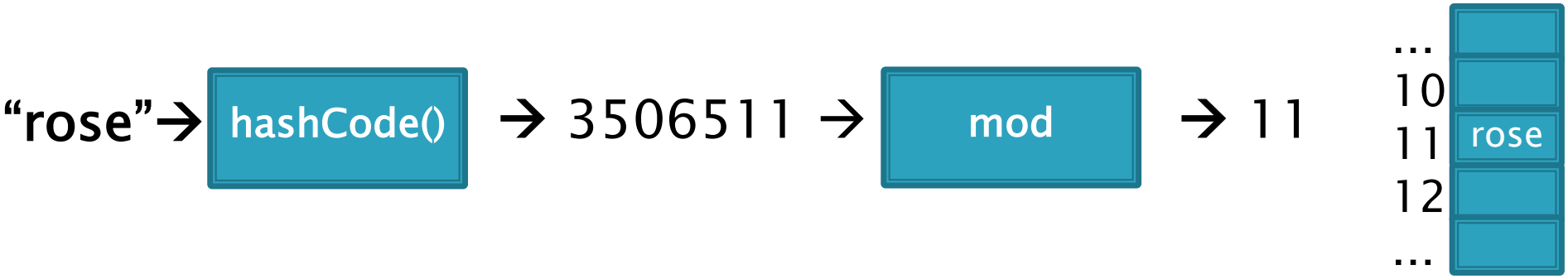
A better hashCode function for Strings uses place value with a base that's prime


```
// 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.

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

Collisions are inevitable

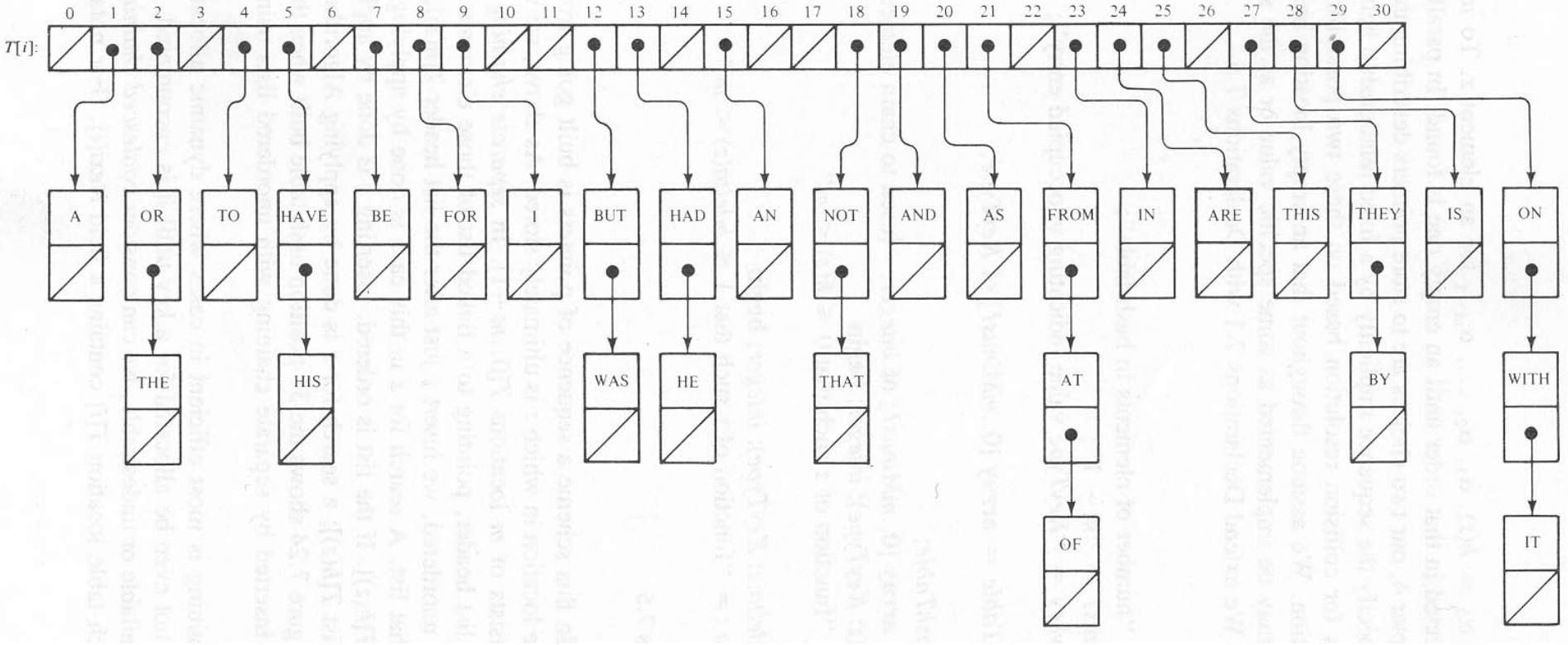


- A good hashCode operation distributes keys uniformly, but collisions will still happen
- hashCode() are ints → only ~4 billion unique values.
  - How many 16 character ASCII strings are possible? "aaaaaaaaaaaaaaaa" <- Here's one
- If n is small (n = # of keys), tables should be much smaller
  - mod will cause collisions too!
- Solutions:
  - Chaining
  - Probing (Linear, Quadratic)

Separate chaining: an array of linked lists

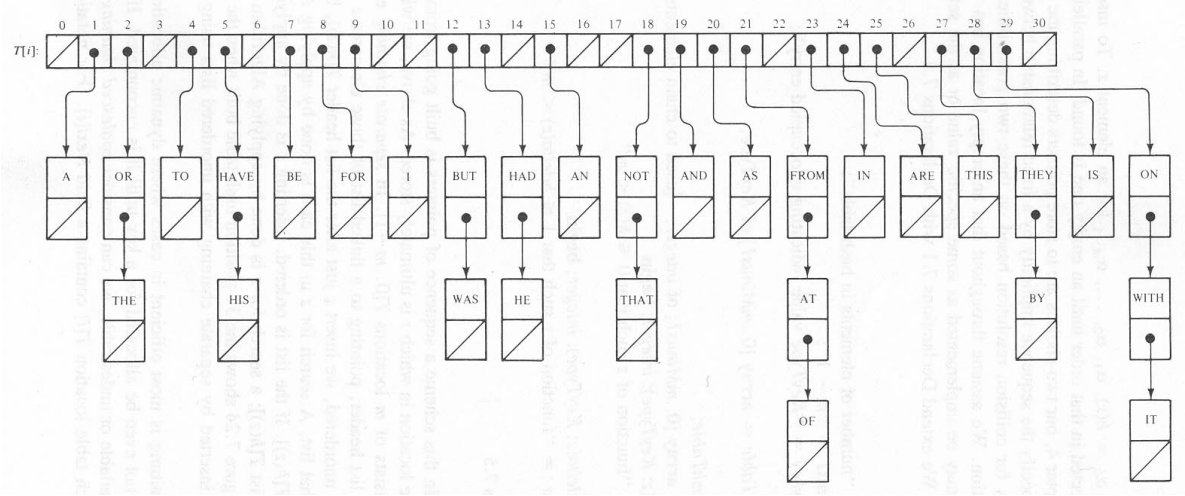
Grow in another direction

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

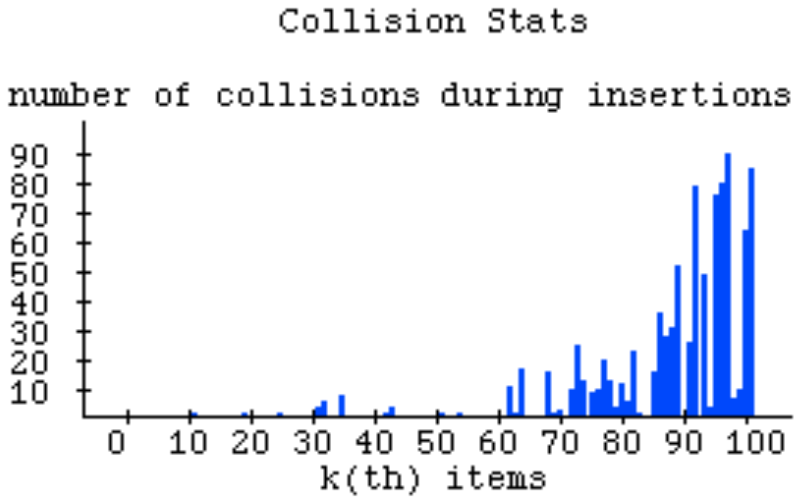
Collision Resolution: Linear Probing

- 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
  - Applet deprecated on most browsers.
  - See next slide for a few freeze-frames.

Clustering Example



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
- λ = probability that a given cell is full
  $(1-\lambda)$ = probability a given cell is empty
- What's the expected number of probes to find an empty cell?

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

If $\lambda = 0.5$
Then $\frac{1}{1-0.5} = 2$

From https://en.wikipedia.org/wiki/List_of_mathematical_series:

$$\sum_{k=1}^{n} kz^k = z\frac{1-(n+1)z^n+nz^{n+1}}{(1-z)^2}$$

Rough Analysis of Linear Probing

- $\lambda = 0.5$
- p = # of probes
- prob(1) = $\lambda^{(p-1)}$ * (1– λ)p = (1–0.75) = 0.25
- prob(2) = $\lambda^{(2-1)}$ * (1– λ)2 = 0.375
  1st probe found full cell, 2nd probe found empty
- prob(3) = $\lambda^{(3-1)}$ * (1– λ)3 = ~0.422

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

If $\lambda = 0.5$
Then $\frac{1}{1-0.5} = 2$

From https://en.wikipedia.org/wiki/List_of_mathematical_series:

$$\sum_{k=1}^{n} kz^k = z\frac{1-(n+1)z^n+nz^{n+1}}{(1-z)^2}$$

Start Here for 2nd Day on Hashing

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. (1st edition = 1968)

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
- Works well with caching

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 the table size is prime, and that we repeat a probe before trying more than half the slots in the table

See that this leads to a contradiction

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

When growing array, can't double!

- Can use, e.g., BigInteger.nextProbablePrime()

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 $\theta(1)$	$\theta(1)$	$\theta(n)$

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

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?

Today's worktime

…Next week's Small Programming HW 4 is StringHashSet – it will be posted by tonight – good idea to work on it after Milestone 2 is completed

…is acceptable to use for EditorTrees Milestone 2 group worktime, especially if you have questions for me