Hashing
A hash function is a function that:

- When applied to an Object, returns a number
- When applied to *equal* Objects, returns the *same* number for each
- When applied to *unequal* Objects, is *very unlikely* to return the same number for each

Hash functions turn out to be very important for searching, that is, looking things up fast

This is their story....
Searching

- Consider the problem of searching an array for a given value
  - If the array is not sorted, the search requires $O(n)$ time
    - If the value isn’t there, we need to search all $n$ elements
    - If the value is there, we search $n/2$ elements on average
  - If the array is sorted, we can do a binary search
    - A binary search requires $O(\log n)$ time
    - About equally fast whether the element is found or not
  - It doesn’t seem like we could do much better
    - How about an $O(1)$, that is, constant time search?
    - We can do it if the array is organized in a particular way
Hashing

- Suppose we were to come up with a “magic function” that, given a value to search for, would tell us exactly where in the array to look
  - If it’s in that location, it’s in the array
  - If it’s not in that location, it’s not in the array
- This function would have no other purpose
- If we look at the function’s inputs and outputs, they probably won’t “make sense”
- This function is called a hash function because it “makes hash” of its inputs
Example (ideal) hash function

Suppose our hash function gave us the following values:

<table>
<thead>
<tr>
<th>Hash Code Value</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>apple</td>
</tr>
<tr>
<td>3</td>
<td>watermelon</td>
</tr>
<tr>
<td>8</td>
<td>grapes</td>
</tr>
<tr>
<td>7</td>
<td>cantaloupe</td>
</tr>
<tr>
<td>0</td>
<td>kiwi</td>
</tr>
<tr>
<td>9</td>
<td>strawberry</td>
</tr>
<tr>
<td>6</td>
<td>mango</td>
</tr>
<tr>
<td>2</td>
<td>banana</td>
</tr>
</tbody>
</table>

```java
hashCode("apple") = 5
hashCode("watermelon") = 3
hashCode("grapes") = 8
hashCode("cantaloupe") = 7
hashCode("kiwi") = 0
hashCode("strawberry") = 9
hashCode("mango") = 6
hashCode("banana") = 2
```
Sets and tables

- Sometimes we just want a set of things—objects are either in it, or they are not in it.
- Sometimes we want a map—a way of looking up one thing based on the value of another.
  - We use a key to find a place in the map.
  - The associated value is the information we are trying to look up.
- Hashing works the same for both sets and maps.
  - Most of our examples will be sets.

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>141</td>
<td></td>
</tr>
<tr>
<td>142</td>
<td>robin</td>
</tr>
<tr>
<td></td>
<td>robin info</td>
</tr>
<tr>
<td>143</td>
<td>sparrow</td>
</tr>
<tr>
<td></td>
<td>sparrow info</td>
</tr>
<tr>
<td>144</td>
<td>hawk</td>
</tr>
<tr>
<td></td>
<td>hawk info</td>
</tr>
<tr>
<td>145</td>
<td>seagull</td>
</tr>
<tr>
<td></td>
<td>seagull info</td>
</tr>
<tr>
<td>146</td>
<td></td>
</tr>
<tr>
<td>147</td>
<td>bluejay</td>
</tr>
<tr>
<td></td>
<td>bluejay info</td>
</tr>
<tr>
<td>148</td>
<td>owl</td>
</tr>
<tr>
<td></td>
<td>owl info</td>
</tr>
</tbody>
</table>
Finding the hash function

- How can we come up with this magic function?
- In general, we cannot--there is no such magic function 😞
  - In a few specific cases, where all the possible values are known in advance, it has been possible to compute a perfect hash function
- What is the next best thing?
  - A perfect hash function would tell us exactly where to look
  - In general, the best we can do is a function that tells us where to start looking!
Example imperfect hash function

- Suppose our hash function gave us the following values:
  - hash("apple") = 5
  - hash("watermelon") = 3
  - hash("grapes") = 8
  - hash("cantaloupe") = 7
  - hash("kiwi") = 0
  - hash("strawberry") = 9
  - hash("mango") = 6
  - hash("banana") = 2
  - hash("honeydew") = 6

- Now what?
Collisions

- When two values hash to the same array location, this is called a **collision**
- Collisions are normally treated as “first come, first served”—the first value that hashes to the location gets it
- We have to find something to do with the second and subsequent values that hash to this same location
Handling collisions

- What can we do when two different values attempt to occupy the same place in an array?
  - **Solution #1:** Search from there for an empty location
    - Can stop searching when we find the value *or* an empty location
    - Search must be end-around
  - **Solution #2:** Use a second hash function
    - ...and a third, and a fourth, and a fifth, ...
  - **Solution #3:** Use the array location as the header of a linked list of values that hash to this location

- All these solutions work, provided:
  - We use the same technique to *add* things to the array as we use to *search* for things in the array
Insertion, I

- Suppose you want to add **seagull** to this hash table
- Also suppose:
  - `hashCode(seagull) = 143`
  - `table[143]` is not empty
  - `table[143] != seagull`
  - `table[144]` is not empty
  - `table[144] != seagull`
  - `table[145]` is empty
- Therefore, put **seagull** at location 145
Searching, I

- Suppose you want to look up **seagull** in this hash table
- Also suppose:
  - `hashCode(seagull) = 143`
  - `table[143]` is not empty
  - `table[143] != seagull`
  - `table[144]` is not empty
  - `table[144] != seagull`
  - `table[145]` is not empty
  - `table[145] == seagull`
- We found **seagull** at location 145
Searching, II

- Suppose you want to look up **COW** in this hash table.

- Also suppose:
  - `hashCode(cow) = 144`
  - `table[144]` is not empty
  - `table[144] != cow`
  - `table[145]` is not empty
  - `table[145] != cow`
  - `table[146]` is empty

- If **COW** were in the table, we should have found it by now.

- Therefore, it isn’t here.
Suppose you want to add **hawk** to this hash table.

Also suppose:
- `hashCode(hawk) = 143`
- `table[143]` is not empty
- `table[143] != hawk`
- `table[144]` is not empty
- `table[144] == hawk`

**hawk** is already in the table, so do nothing.
Suppose:

- You want to add `cardinal` to this hash table
- `hashCode(cardinal) = 147`
- The last location is 148
- 147 and 148 are occupied

Solution:

- Treat the table as circular; after 148 comes 0
- Hence, `cardinal` goes in location 0 (or 1, or 2, or ...)

```
141
142  robin
143  sparrow
144  hawk
145  seagull
146
147  bluejay
148  owl
```
Clustering

- One problem with the above technique is the tendency to form “clusters”
- A cluster is a group of items not containing any open slots
- The bigger a cluster gets, the more likely it is that new values will hash into the cluster, and make it ever bigger
- Clusters cause efficiency to degrade
- Here is a non-solution: instead of stepping one ahead, step $n$ locations ahead
  - The clusters are still there, they’re just harder to see
  - Unless $n$ and the table size are mutually prime, some table locations are never checked
Efficiency

- Hash tables are actually surprisingly efficient
- Until the table is about 70% full, the number of probes (places looked at in the table) is typically only 2 or 3
- Sophisticated mathematical analysis is required to prove that the expected cost of inserting into a hash table, or looking something up in the hash table, is $O(1)$
- Even if the table is nearly full (leading to occasional long searches), efficiency is usually still quite high
Solution #2: Rehashing

- In the event of a collision, another approach is to **rehash**: compute another hash function
  - Since we may need to rehash many times, we need an easily computable sequence of functions

- Simple example: in the case of hashing Strings, we might take the previous hash code and add the length of the String to it
  - Probably better if the length of the string was not a component in computing the original hash function

- Possibly better yet: add the length of the String plus the number of probes made so far
  - Problem: are we sure we will look at every location in the array?

- Rehashing is a fairly uncommon approach, and we won’t pursue it any further here
Solution #3: Bucket hashing

- The previous solutions used **open hashing**: all entries went into a “flat” (unstructured) array.
- Another solution is to make each array location the header of a linked list of values that hash to that location.

<table>
<thead>
<tr>
<th>141</th>
<th>142</th>
<th>143</th>
<th>144</th>
<th>145</th>
<th>146</th>
<th>147</th>
<th>148</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>robin</td>
<td></td>
<td>sparrow</td>
<td></td>
<td></td>
<td>bluejay</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>hawk</td>
<td></td>
<td></td>
<td></td>
<td>owl</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
</tr>
</tbody>
</table>
The `hashCode` function

- `public int hashCode()` is defined in `Object`
- Like `equals`, the default implementation of `hashCode` just uses the address of the object—probably not what you want for your own objects
- You can override `hashCode` for your own objects
- As you might expect, `String` overrides `hashCode` with a version appropriate for strings
- Note that the supplied `hashCode` method can return *any* possible `int` value (including negative numbers)
  - You have to adjust the returned `int` value to the size of your hash table
Why do you care?

- Java provides `HashSet`, `Hashtable`, and `HashMap` for your use.
- These classes are very fast and very easy to use.
- They work great, without any additional effort, for Strings.
- But...

They will **not work** for your own objects **unless either:**

- You are satisfied with the inherited `equals` method (no object is equal to any other, separately created object).

**Or:**

- You have defined `equals` for your objects *and*
- You have *also* defined a `hashCode` method that is *consistent with* your `equals` method (that is, equal objects have equal hash codes).
Writing your own `hashCode()`

- A `hashCode()` method **must**:
  - Return a value that is (or can be converted to) a legal array index
  - Always return the same value for the same input
    - It can’t use random numbers, or the time of day
  - Return the same value for *equal* inputs
    - Must be consistent with your `equals` method
  - It does **not** need to guarantee different values for different inputs
  - A *good* `hashCode()` method **should**:
    - Make it *unlikely* that different objects have the same hash code
    - Be efficient to compute
    - Give a uniform distribution of values
    - *Not* assign similar numbers to similar input values
Other considerations

- The hash table might fill up; we need to be prepared for that
  - Not a problem for a bucket hash, of course
- You cannot easily delete items from an open hash table
  - This would create empty slots that might prevent you from finding items that hash before the slot but end up after it
  - Again, not a problem for a bucket hash
- Generally speaking, hash tables work best when the table size is a prime number
Java provides classes `Hashtable`, `HashMap`, and `HashSet` (and many other, more specialized ones)

- **Hashtable** and **HashMap** are maps: they associate keys with values
- **Hashtable** is synchronized; that is, it can be accessed safely from multiple threads
  - `Hashtable` uses an open hash, and has a rehash method, to increase the size of the table
- **HashMap** is newer, faster, and usually better, but it is not synchronized
  - `HashMap` uses a bucket hash, and has a remove method
- **HashSet** is just a set, not a collection, and is not synchronized
Hash table operations

- HashSet, Hashtable and HashMap are in java.util
- All have no-argument constructors, as well as constructors that take an integer table size
- The maps have methods:
  - public Object put(Object key, Object value)
    - (Returns the previous value for this key, or null)
  - public Object get(Object key)
  - public void clear()
  - public Set keySet()
    - Dynamically reflects changes in the hash table
  - ...and many others
Bottom line

- You do **not** have to write a `hashCode()` method if:
  - You never use a built-in class that depends on it, or
  - You put only `Strings` in hash sets, and use only `Strings` as keys in hash maps (values don’t matter), or
  - You are happy with `equals` meaning `==`, and don’t override it

- You **do** have to write a `hashCode()` method if:
  - You use a built-in hashing class for your own objects, and you override `equals` for those objects

- Finally, if you ever override `hashCode`, you **must** also override `equals`
The End