



Hashing





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:

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

0	kiwi
1	
2	banana
3	watermelon
4	
5	apple
6	mango
7	cantaloupe
8	grapes
9	strawberry



Why hash tables?

- We don't (usually) use hash tables just to see if something is there or not—instead, we put **key/value pairs** into a **map**
 - We use a *key* to find a place in the table
 - The *value* holds the information we are actually interested in

...	<i>key</i>	<i>value</i>
141		
142	robin	robin info
143	sparrow	sparrow info
144	hawk	hawk info
145	seagull	seagull info
146		
147	bluejay	bluejay info
148	owl	owl info



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:

- $\text{hash}(\text{"apple"}) = 5$
 $\text{hash}(\text{"watermelon"}) = 3$
 $\text{hash}(\text{"grapes"}) = 8$
 $\text{hash}(\text{"cantaloupe"}) = 7$
 $\text{hash}(\text{"kiwi"}) = 0$
 $\text{hash}(\text{"strawberry"}) = 9$
 $\text{hash}(\text{"mango"}) = 6$
 $\text{hash}(\text{"banana"}) = 2$
 $\text{hash}(\text{"honeydew"}) = 6$

- Now what?

0	kiwi
1	
2	banana
3	watermelon
4	
5	apple
6	mango
7	cantaloupe
8	grapes
9	strawberry



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

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



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

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



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

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



Insertion, II

- 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

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



Insertion, III

- Suppose:
 - You want to add **cardinal** to this hash table
 - $\text{hashCode}(\text{cardinal}) = 147$
 - 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 long searches), efficiency is usually still quite high

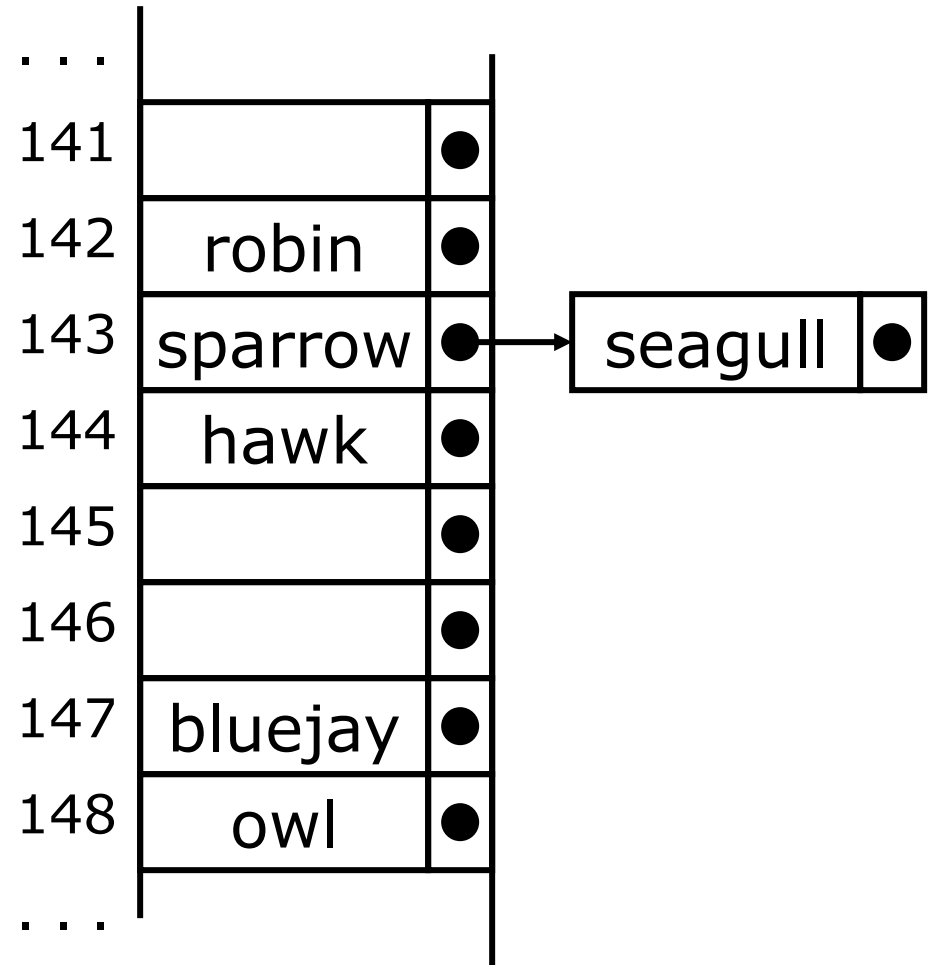


Solution #2: Rehashing

- 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 address 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





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 *does not know the size of your array*—you have to adjust the returned `int` value yourself



Writing your own hashCode method

- 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 return different values for different inputs
- A *good* hashCode method *should*:
 - Be efficient to compute
 - Give a uniform distribution of array indices
 - *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 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



Hash tables in Java

- Java provides **HashSet**, **Hashtable** and **HashMap**
- **HashSet** is a set; things are in it, or they aren't
- **Hashtable** and **HashMap** are **maps**: they associate *keys* with *values*
- **Hashtable** is synchronized; 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



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
- `HashSet` has methods `add`, `contains`, `remove`, `iterator`, etc.
- `Hashtable` and `HashMap` have these 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



The End
