CSCI 136
Data Structures &
Advanced Programming

Fall 2019

Lecture 34

2070567 & 82879

Last Time

- Maps & Hashing

Announcements

- No Lab Today!
- Review Session
  - Thursday, Dec. 12: 4:00-5:30 pm
  - SSL 030a
  - BYOQ
- Course Evaluations
  - Blue Sheets: In class on Friday
  - On-line form: bring a computer to class on Friday
    - Also computers in the library

Today

- Hashing Wrap-up

hashCode()

- What properties do we want hashCode to have so that it is useful to find the right bin?
- Should always give the same result for a given object
- Should always give the same result for two equal objects (meaning equals() is true)
- Should not (too often) give the same result for two unequal objects

What happens if a bin is full?

- Let's say we store objects in an array
- We get unlucky – two are assigned to the same slot

- What do we do?

Linear Probing

- If a collision occurs at a given bin, just move forward (linearly) until an empty slot is available

- Let's implement put(key, val) and get(key)…

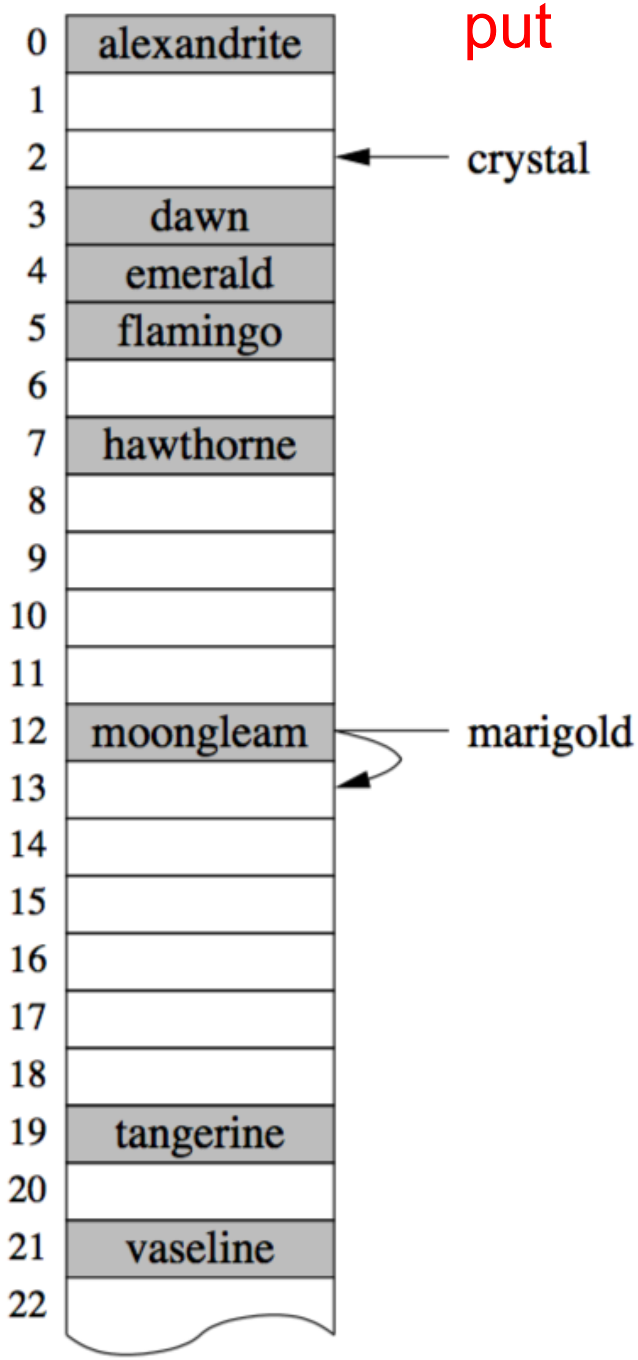
First Attempt: put(K)

```
public V put (K key, V value) {
   int bin = key.hashCode() % data.length;
   while (true) {
      Association<K,V> slot = (Association<K,V>) data[bin];
      if (slot == null) {
         data[bin] = new Association<K,V>(key,value);
         return null;
      }
      if (slot.getKey().equals(key)) { // already exists!
         V old = slot.getValue();
         slot.setValue(value);
         return old;
      }
      bin = (bin + 1) % data.length;
   }
}
```

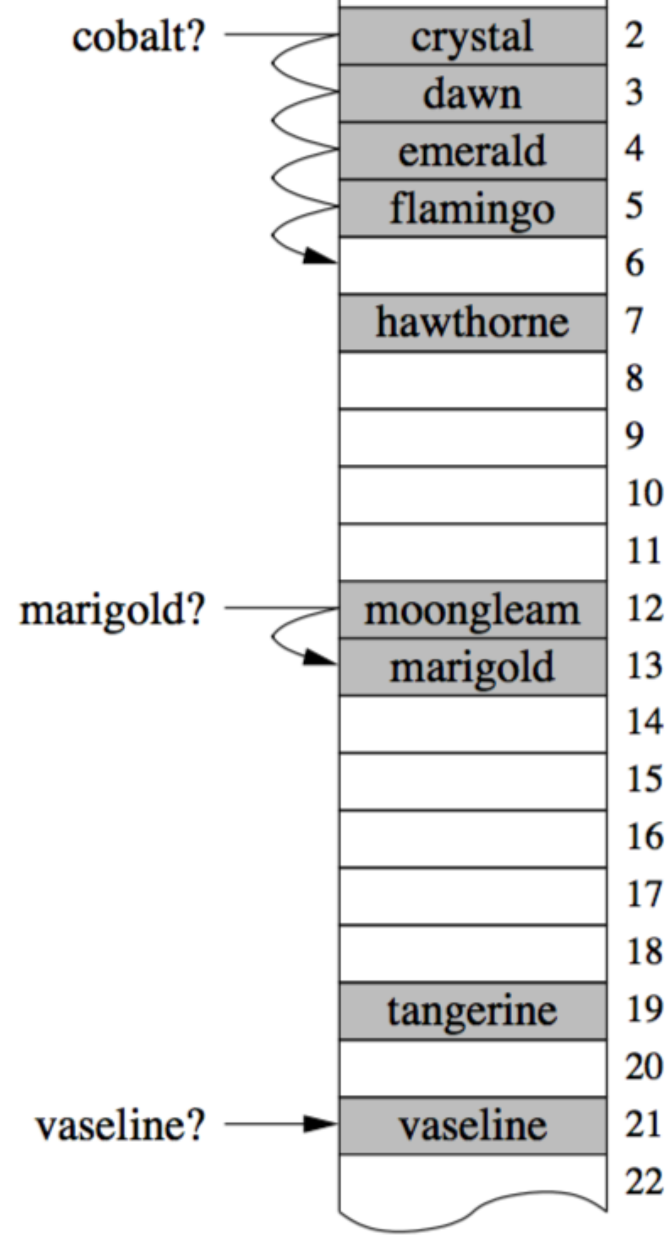
First Attempt: get(K)

```
public V get (K key) {
   int bin = key.hashCode() % data.length;
   while (true) {
      Association<K,V> slot = (Association<K,V>) data[bin];
      if (slot == null)
         return null;

      if (slot.getKey().equals(key))
         return slot.getValue();

      bin = (bin + 1) % data.length;
   }
}
```

put



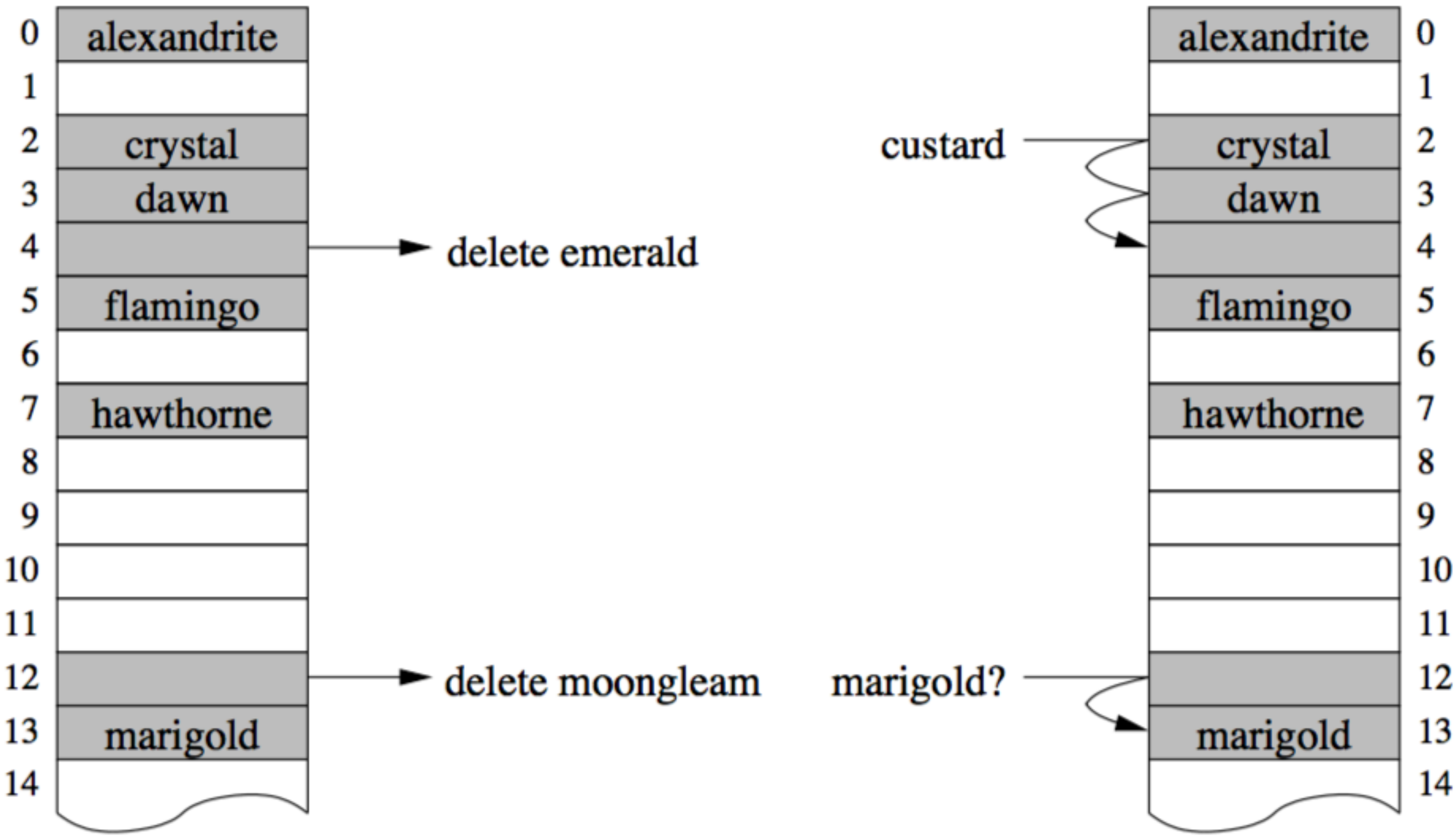
contains



Linear Probing

- If a collision occurs at a given bin, just move forward (linearly) until an empty slot is available
- Let's implement put(key, val) and get(key)…
- What happens when we remove "moongleam", and then lookup "marigold"?
  - Need a "placeholder" for removed values…

Reserving Empty Slots



The Locate Method

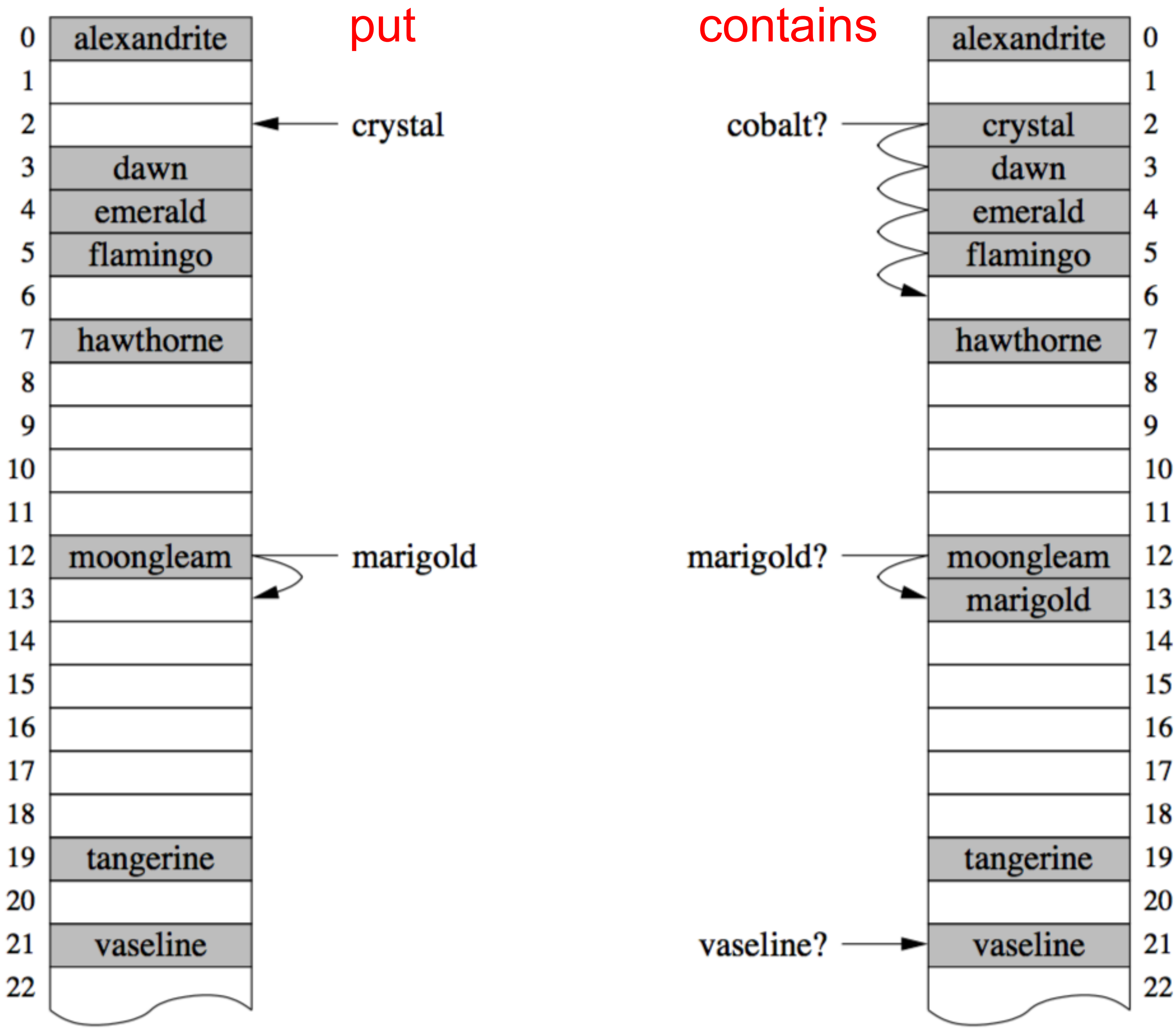
```
protected int locate(K key) {

   int hash = Math.abs(key.hashCode() % data.size());
   int reservedSlot = -1;
   boolean foundReserved = false;
   while (data.get(hash) != null)  {
      if (data.get(hash).reserved()) {
         if (!foundReserved) {
            reservedSlot = hash;
            foundReserved = true;
          }
      }
      else  {
         if (key.equals(data.get(hash).getKey()))
            return hash;
      }
      hash = (1+hash)%data.size();
   }
   if (!foundReserved) return hash;
   else return reservedSlot;
}
```

Implementation Highlights

- The locate method returns
  - The index to which key hashes if that slot is empty, or
  - The index in which the key is stored if key is in table, or
  - The index of the first empty location after the index to which the key hashes
- Locate is used by, put, get, containsKey
- The expand method
  - Creates a larger hash table when one is needed
- The HashtableIterator
  - Iterates in the order in which data is stored in Vector
  - Question: Why doesn't it use the Vector's iterator?

Collisions & Clustering

- Linear probing leads to *clustering*
  - We need to look through the entire "cluster" of contiguous nonempty slots to find an element (or to find an empty slot for a new element)
  - So put(key,val) takes O(C) time when inserting into a cluster of size C, and get(key) takes O(C) time when the item is in a clusert of size C


put
contains
0 alexandrite
1
2 crystal
3 dawn
4 emerald
5 flamingo
6
7 hawthorne
8
9
10
11
12 moongleam marigold
13
14
15
16
17
18
19 tangerine
20
21 vaseline
22
cobalt?
alexandrite 0
1
crystal 2
dawn 3
emerald 4
flamingo 5
6
hawthorne 7
8
9
10
11
marigold? moongleam 12
marigold 13
14
15
16
17
18
tangerine 19
20
vaseline? vaseline 21
22

Open Addressing

- Linear probing is an “open addressing” strategy
- If an item does not fit in its original slot, we store it in another slot
- Are there other strategies that might perform better?

Open Addressing : Quadratic Probing

- With linear probing, the i^{th} probe for key k occurs at location (h(k) + i) % arraySize (assuming h(k) is non-negative)
- With *quadratic probing*, the i^{th} probe for key k occurs at location (h(k) + i^2) % arraySize (starting with i = 0)
- Quadratic probing helps to avoid primary clustering.

- Quadratic probing may not always find an empty slot!
  - But as long as table is at most half-full, an empty slot will be found
  - This can be shown with simple modular arithmetic assuming that the size of the table is prime

Open Addressing : Quadratic Probing

- If two items hash to the same slot, they will share all the same addresses in quadratic probing
- But if they hash to different slots, they won't (even if they start right next to each other)
- This is sometimes called "primary clustering" vs "secondary clustering"

- This helps avoid collisions, but linear probing is seen much more often in practice
- Remember that quadratic probing (as described here) does best when the hash table size is prime.

Load Factor

- Need to keep track of how full the table is
  - Why?
  - What happens when array fills completely?
- Load factor is a measure of how full the hash table is
  - LF = (# elements) / (table size)
- When LF reaches some threshold, double size of array
  - For linear probing, typical threshold = 0.6
  - For quadratic probing, typical threshold = 0.5

Doubling Array

- Cannot just copy values
  - Why?
  - Hash values may change
  - Example: suppose (`key.hashCode() == 11`)
    - 11 % 8 = 3;
    - 11 % 16 = 11;
- Result: must recompute all hash codes, reinsert into new array

Open Addressing : Double Hashing

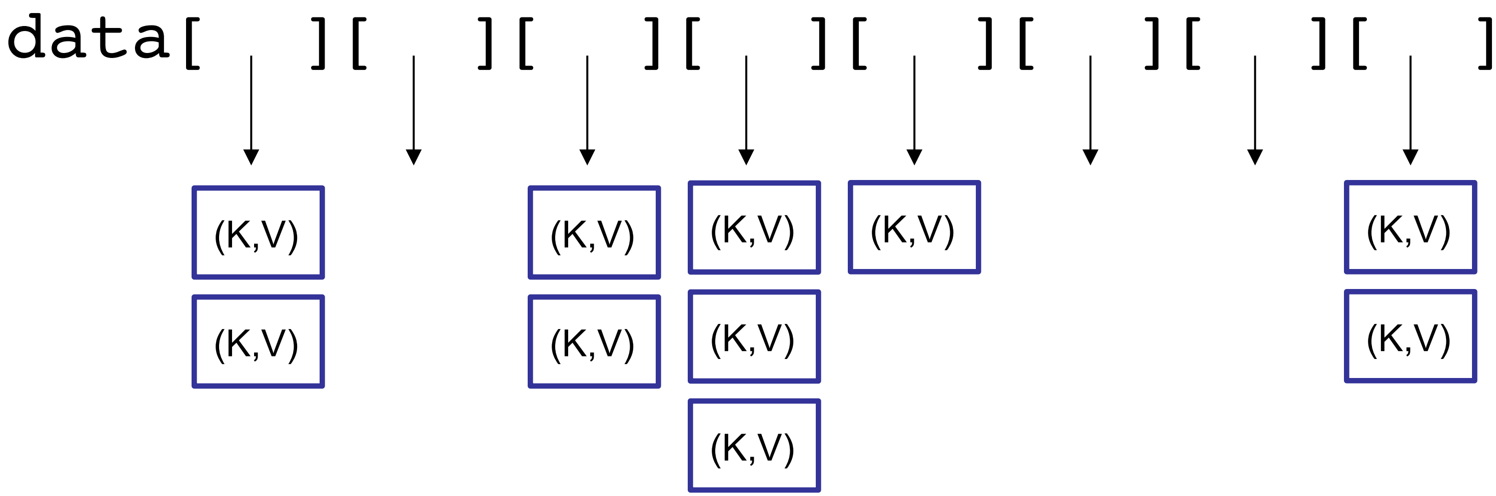
- With *double hashing* a second hashing function h'(k) is used to determine the probe sequence of k if location h(k) is full
- The i^{th} probe for key k occurs at location (h(k) + i*h'(k) % arraySize (starting with i = 0)
- A good secondary hashing function needs to ensure that
  - h'(k) ≠ 0
  - h'(k) should not share any factors with arraySize (to ensure that all array locations can be probed if needed.

Open Addressing Limitations

- Downsides of open addressing?
  - What if array is almost full?
    - Loooong runs for every lookup…
    - Array doubling or periodic table rehashing is needed
- How can we avoid these problems?
  - Keep all values that hash to same bin in a `Structure`
    - Usually a SLL
  - *External chaining* "chains" objects with the same hash value together

External Chaining

- Instead of runs, we store a list in each bin



- get(), put(), and remove() only need to check one slot's list
- No placeholders!

Probing vs. Chaining

What is the performance of:

- `put(K, V)`
  - LP: O(1 + cluster length)
  - EC: O(1 + chain length)
- `get(K)`
  - LP: O(1 + cluster length)
  - EC: O(1 + chain length)
- `remove(K)`
  - LP: O(1 + cluster length)
  - EC: O(1 + chain length)
- How do we control cluster/chain length?

Good Hashing Functions

- Important point:
  - All of this hinges on using “good” hash functions that spread keys “evenly”
- Good hash functions
  - Fast to compute
  - Uniformly distribute keys
- Almost always have to test “goodness” empirically

Example Hash Functions

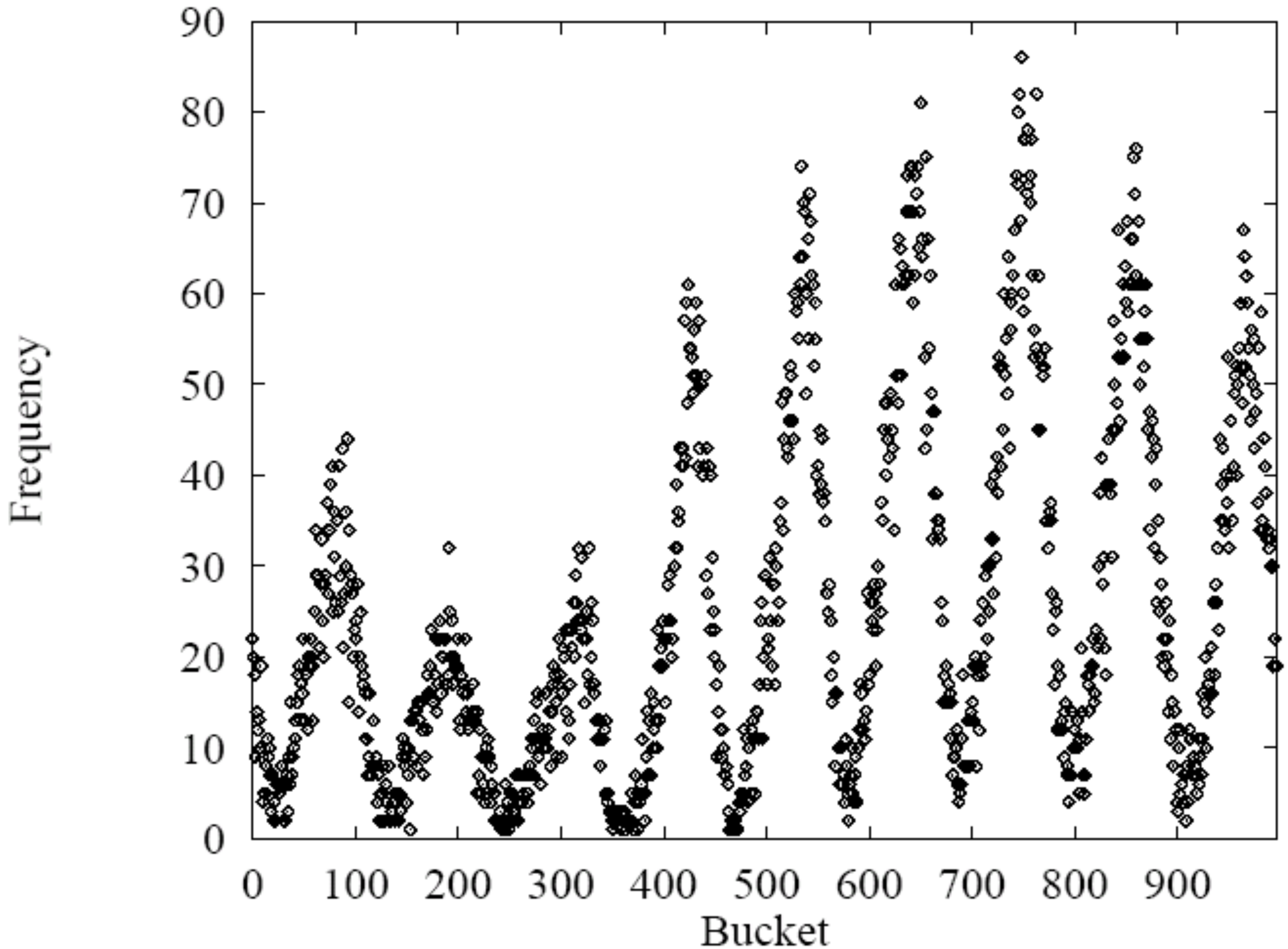
- What are some feasible hash functions for Strings?
  - First char ASCII value mapping
    - 0-255 only
    - Not uniform (some letters more popular than others)
  - Sum of ASCII characters
    - Not uniform - lots of small words
    - smile, limes, miles, slime are all the same

Example Hash Functions

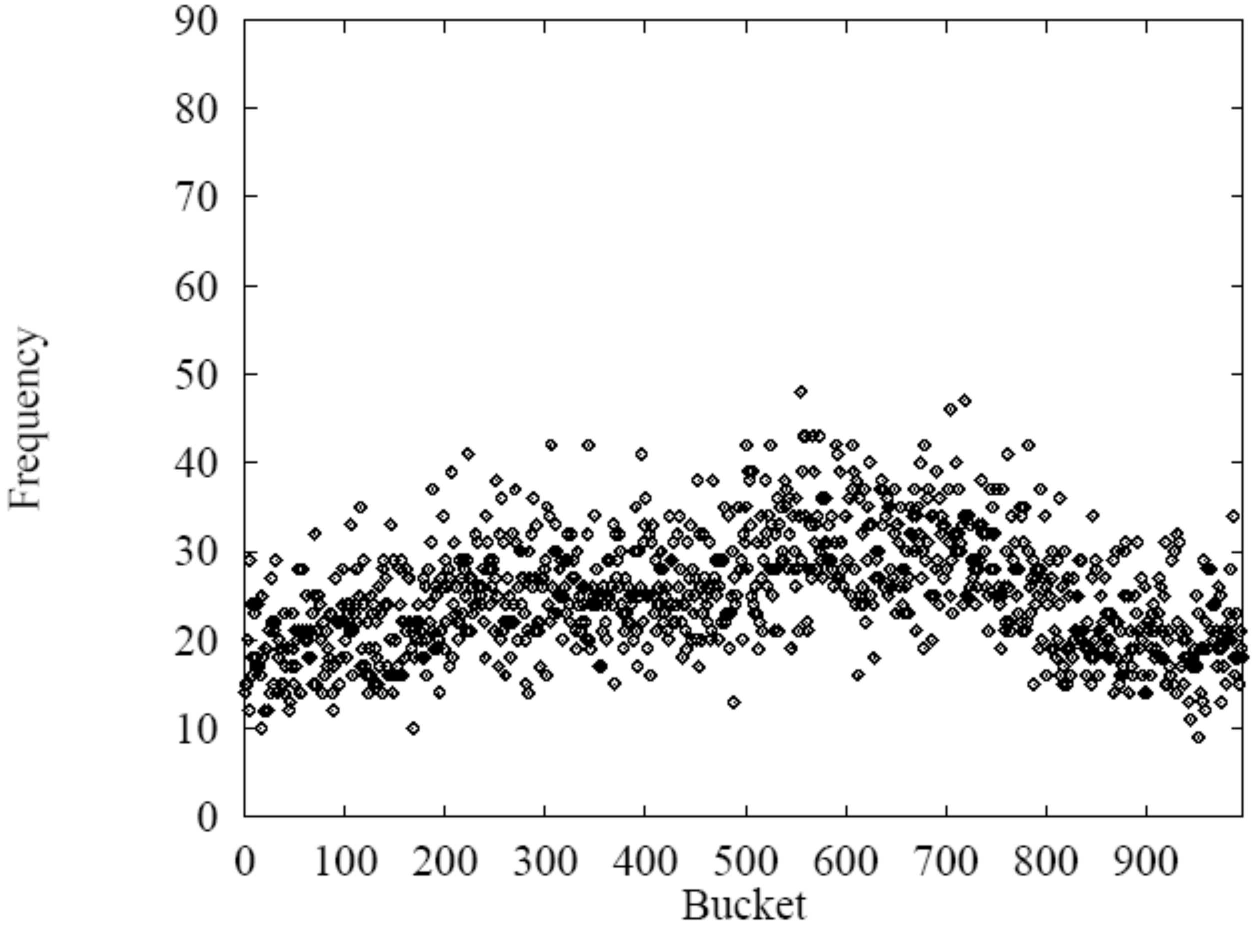
- String hash functions
  - Weighted sum
    - Small words get bigger codes
    - Distributes keys better than non-weighted sum
  - Let's look at different weights…

$$\sum_{i=0}^{n=s.length()} s.charAt(i)$$

Hash of all words in UNIX spelling dictionary (997 buckets)

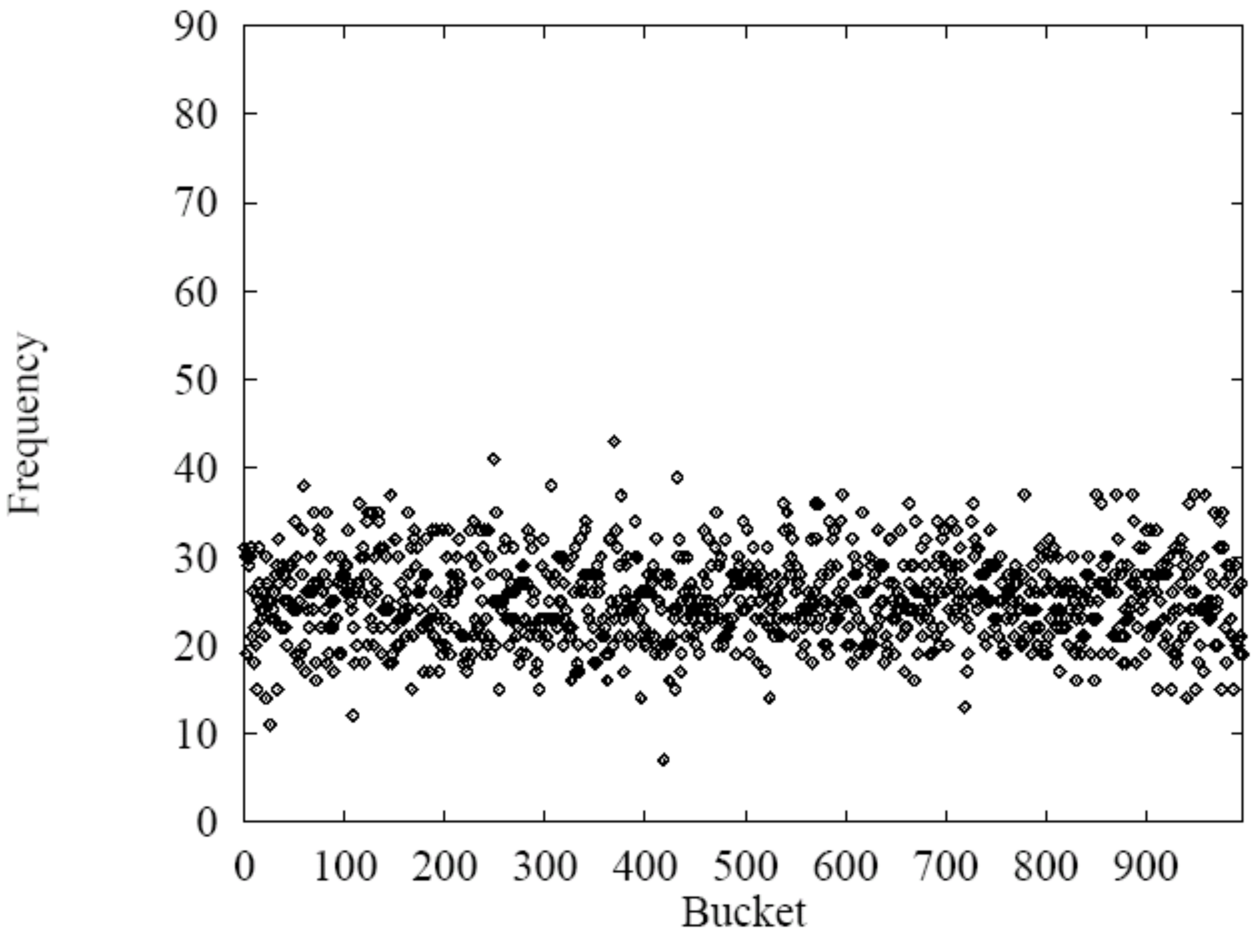


$$\sum_{i=0}^{n} s.charAt(i) * 2^i$$



$$\sum_{i=0}^{n} s.charAt(i) * 256^{i}$$

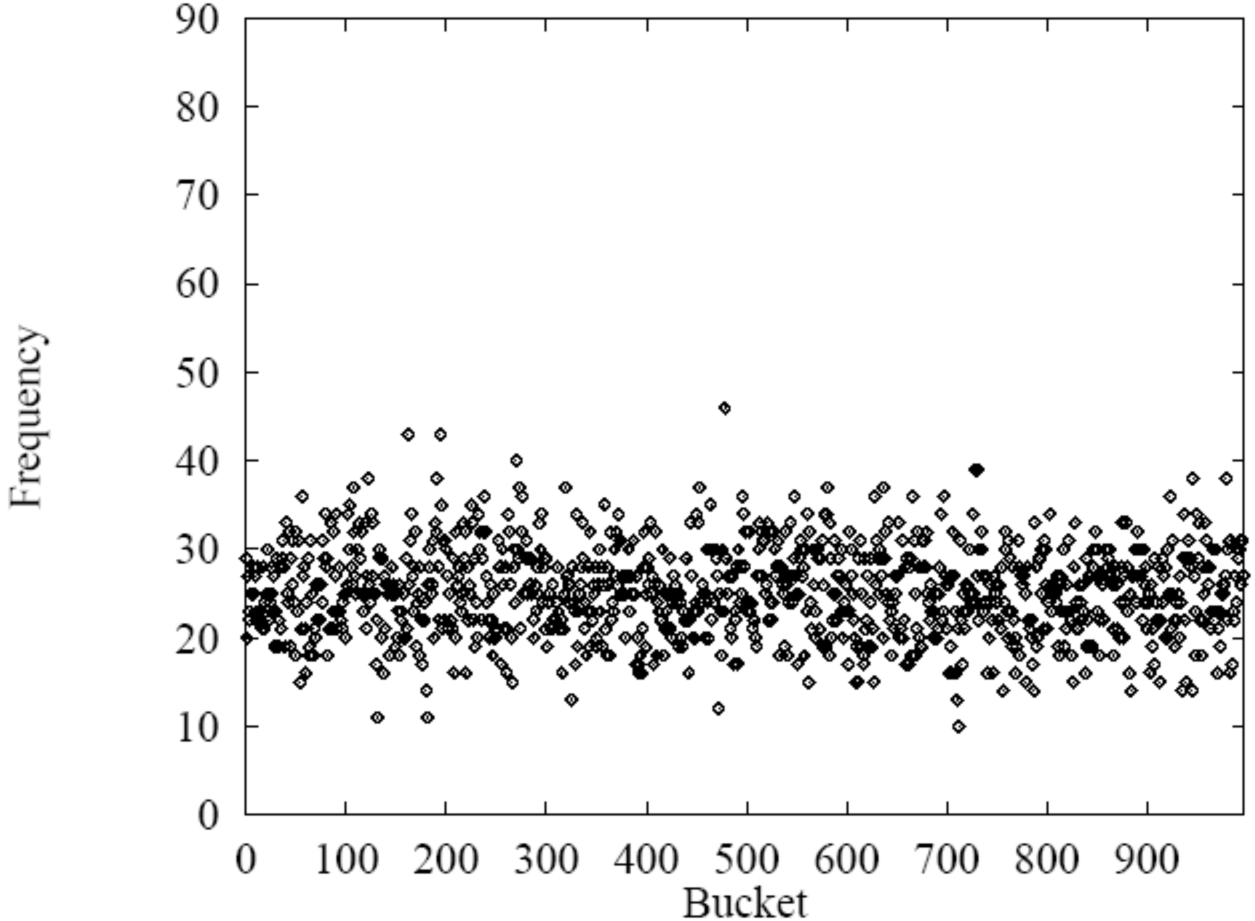
This looks pretty good, but 256^i is big…



$$\sum_{i=0}^{n} s.charAt(i) * 31^{i}$$

Java uses:

$$\sum_{i=0}^{n} s.charAt(i) * 31^{(n-i-1)}$$



Hashtables: O(1) operations?

- How long does it take to compute a String's hashCode?
  - O(s.length())
- Given an object's hash code, how long does it take to find that object?
  - O(run length) or O(chain length) times cost of .equals() method

Hashtables: O(1) operations?

- If items are assigned to a random slot, and the load factor is a constant, then:
  - The run length is O(1) on average
  - The chain length is O(1) on average

- Conclusion: for a good hash function (fast, uniformly distributed) and a low load factor (short runs/chains), we say hashtables are O(1)

Summary

	put	get	space
unsorted vector	O(n)	O(n)	O(n)
unsorted list	O(n)	O(n)	O(n)
sorted vector	O(n)	O(log n)	O(n)
balanced BST	O(log n)	O(log n)	O(n)
array indexed by key	O(1)	O(1)	O(key range)