CSCI 136
Data Structures &
Advanced Programming

Lecture 34

Fall 2019

2070567 & 82879

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: On your own schedule
    - Please fill them out!

Last Time

- Maps & Hashing

Today

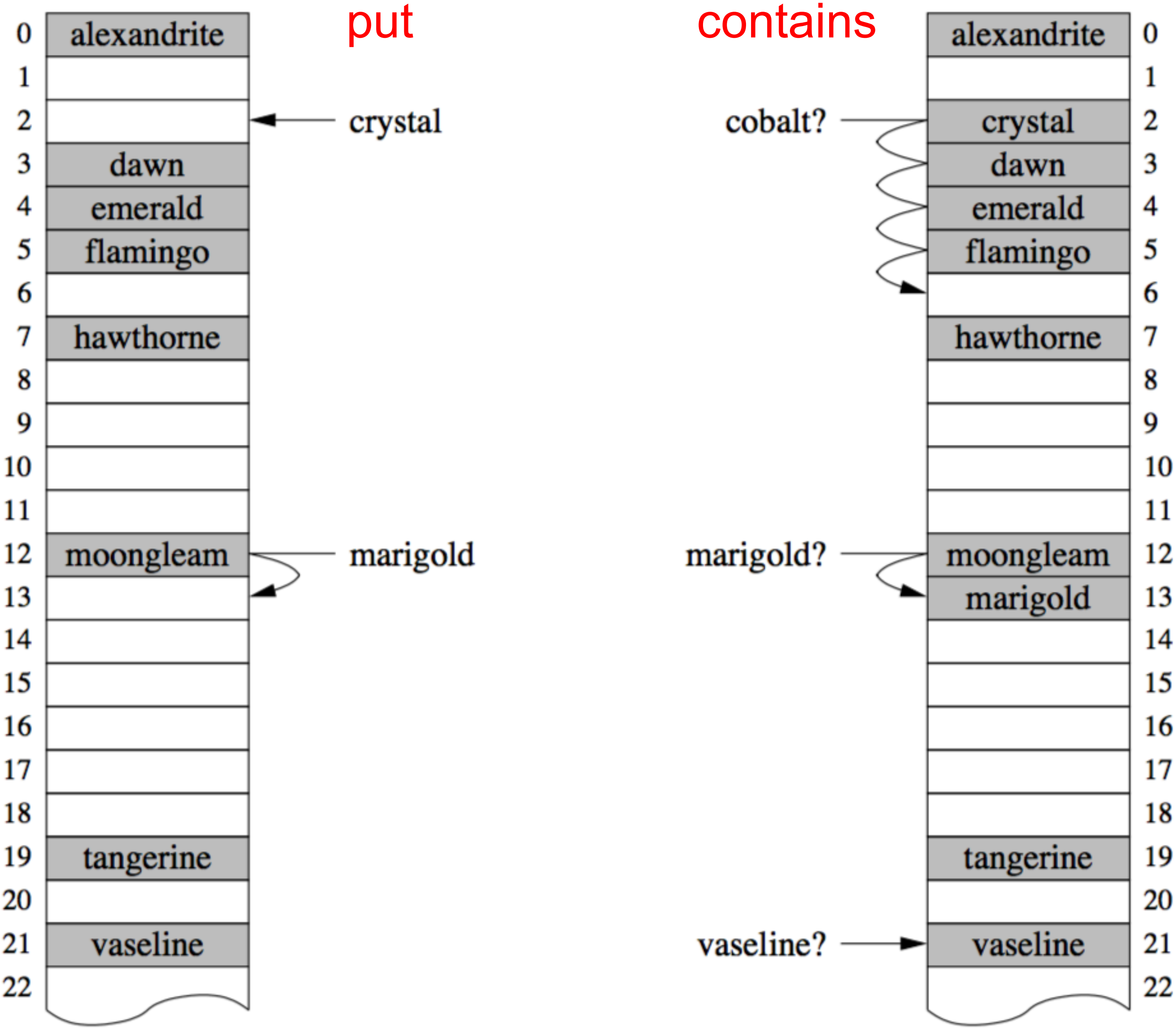
- Hashing Wrap-up

Hashing in a Nutshell

- Store a value based on a key
- Assign values to "bins" based on keys
- When searching for object, go directly to appropriate bin (and ignore the rest)
- If there are multiple objects associated with bin, then search for the correct one
- Important Insight: Hashing works best when objects are evenly distributed among bins

Implementing a HashTable

- How can we represent bins?
- Slots in array
  - Storing in Vector is slower (and problematic)
  - Initial size of array is typically a prime number
- How do we find a key's bin number?
  - We use a *hash function* that converts keys into integers
  - In Java, all Objects have `public int hashCode()`
    - Hashing function is one way: key   fingerprint
    - Hashing function is deterministic



put

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

contains

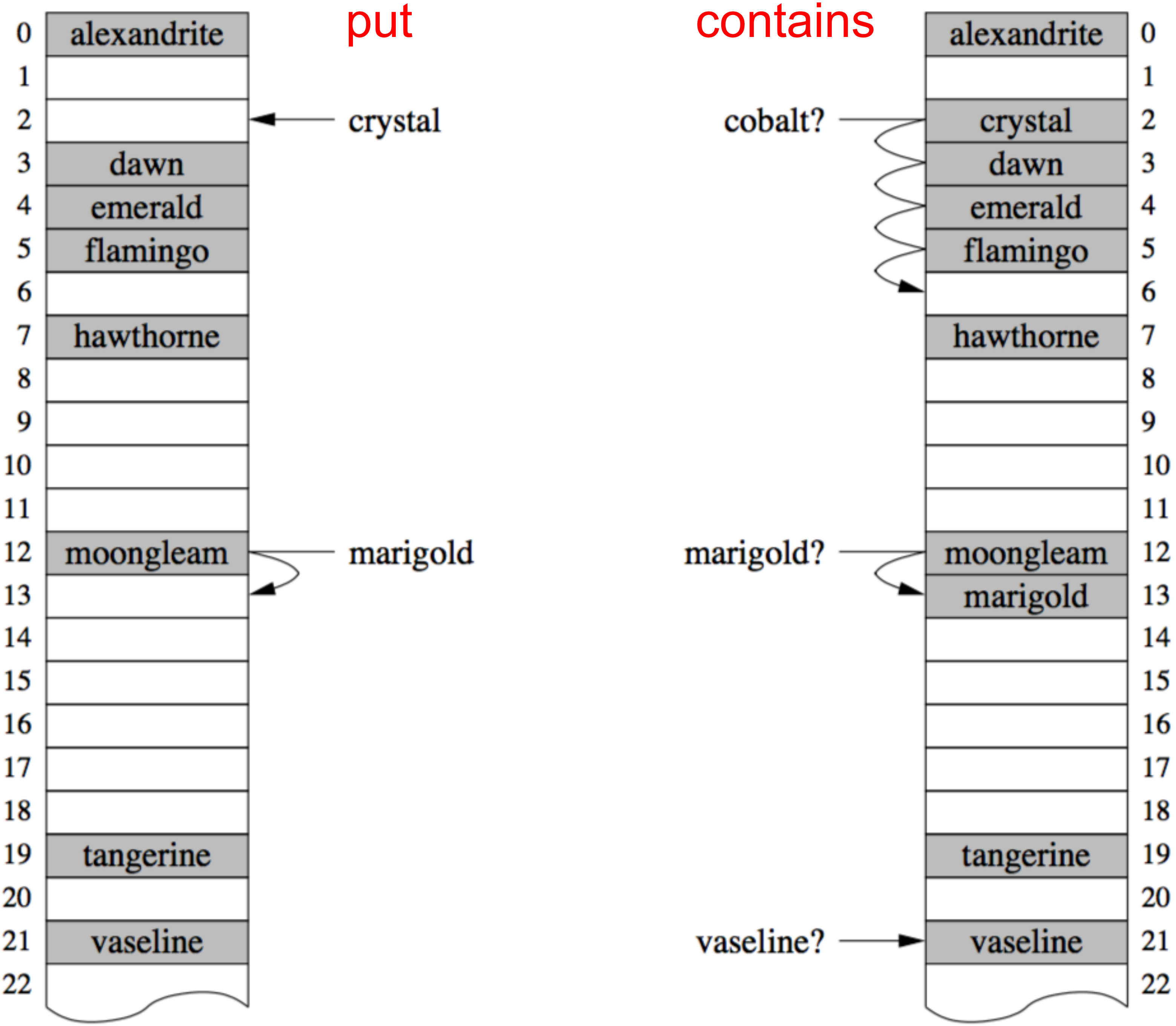
alexandrite	0
	1
crystal	2
dawn	3
emerald	4
flamingo	5
	6
hawthorne	7
	8
	9
	10
	11
moongleam	12
marigold	13
	14
	15
	16
	17
	18
tangerine	19
	20
vaseline	21
	22

Implementing HashTable

- Store (Key, Value) pair as Association
- How do we add Associations to the array?
  - `array[o.hashCode() % array.length] = o;` ?
    - What's "aaaaaa".hashCode() ?
    - Use absolute value unless you're sure o.hashCode() ≥ 0
- Collisions make life hard
- Two approaches
  - Open Addressing
    - Linear or Quadratic Probing
    - Double Hashing
  - External chaining

Open Addressing : Linear Probing

- *Collision:* A key hashes to a location occupied by another key
- On collision, begin *linear probing* to find a slot
  - Add k (for some k>0) to current index; repeat
  - Insert data into first available slot
- Note: If k divides n, we can only access n/k slots
  - So, either set k = 1 or choose n to be prime (or both)!
- This method leads to *clustering*
  - Primary clustering: Keys with same hash value collide with one another
  - Secondary clustering: Keys with unequal hash values collide with one another



put

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		

contains

	alexandrite	0
		1
cobalt?	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

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;
   }
}
```

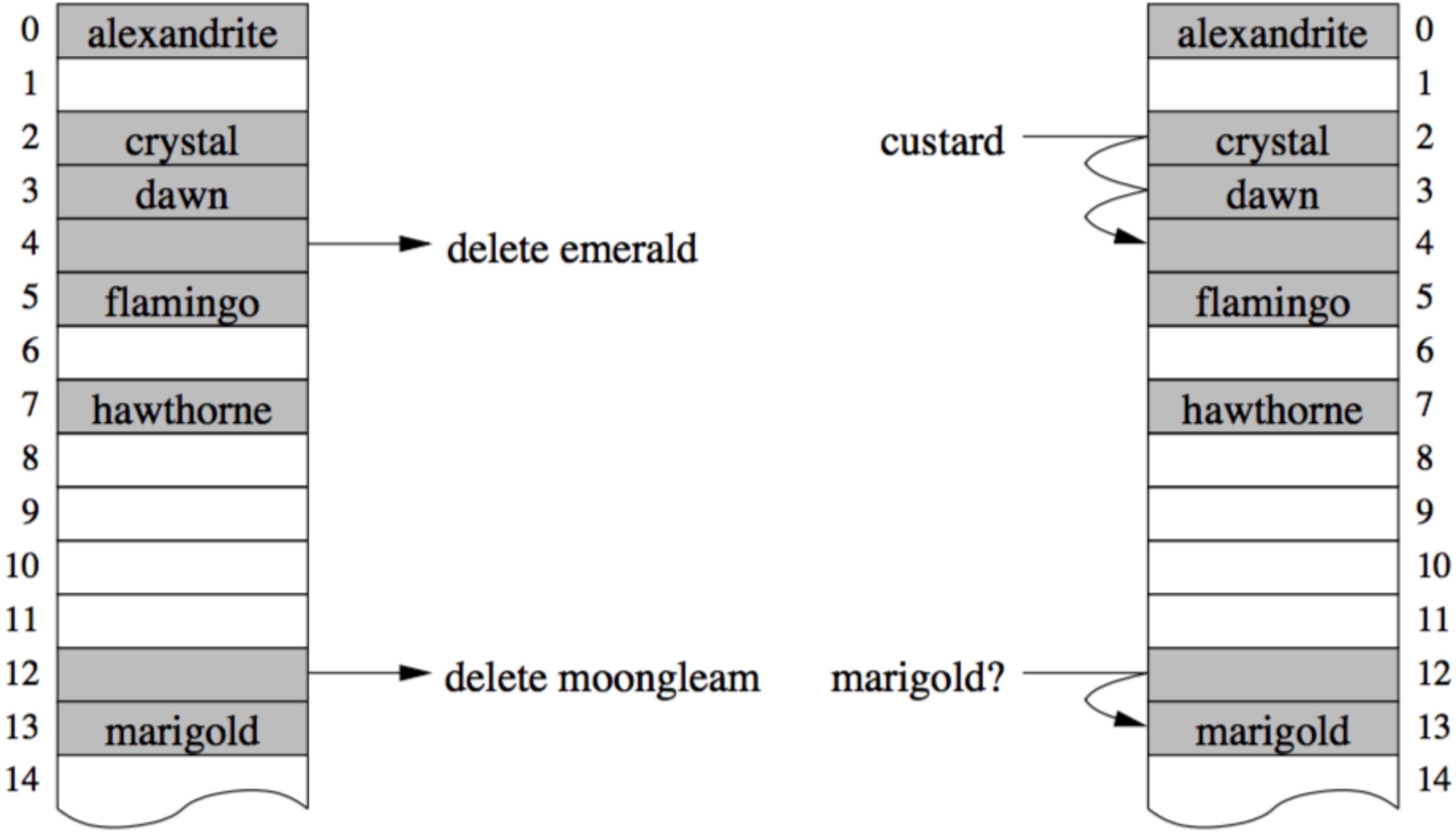
First Attempt: get(K)

```
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;
   }
}
```

Reserving Empty Slots

- If a collision occurs at a given bin, move to subsequent bins until an empty slot is available
  - Specify easy hash function: length
  - Initial array size = 7
  - Add “algorithms” to hash table
  - Add “data”
  - Add “bit”
- What happens when we remove “algorithms”, and then lookup “bit”?
  - Need a “placeholder” for removed values…

Reserving Empty Slots



The HashAssociation Class

How can we mark an array entry as ’reserved’?

- A second array of ‘reserved’ flags?
- Pair each Association with a ‘reserved’ flag?
- Make a ‘reserved’ flag part of the Association?

The HashAssociation uses the third approach

- Adds a ‘reserved’ instance variable
- Adds ‘reserve’ and ’reserved’ method

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 (or reserved) location after the index to which the key hashes
- Locate is used by, put, get, containsKey
- The expand method
  - Creates a larger Vector
  - Rehashes each element from original Vector to larger one
- The HashtableIterator
  - Iterates in the order in which data is stored in Vector
  - Question: Why doesn’t it use the Vector’s iterator?

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

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;
}
```

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 secondary (but not primary) clustering.
- Quadratic probing may not always find an empty slot!
  - Can have h(k) + i^2 = h(k) + j^2 % p for i ≠ j and i,j < p
  - 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 p 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

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 ith 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
    - $h'(k) \neq 0$
    - h’(k) should not share any factors with arraySize (to ensure that all array locations can be probed if needed.

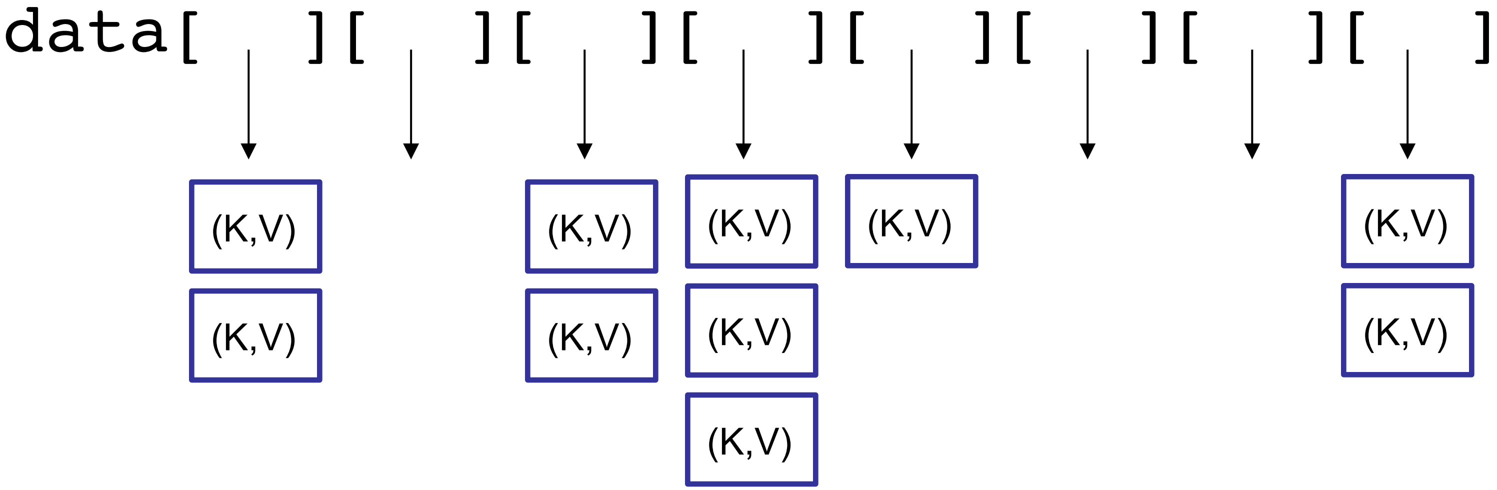
Lecture 34 Ended Here

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:

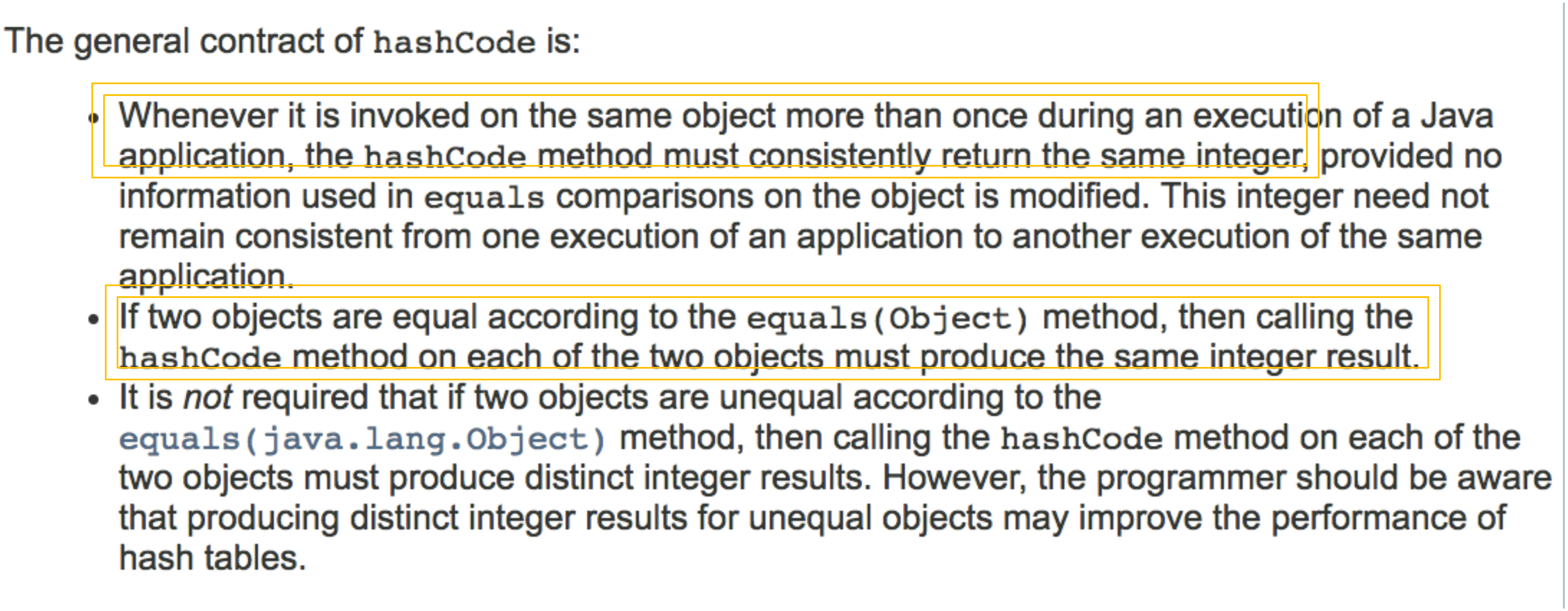
- `put(K, V)`
  - LP: O(1 + run length)
  - EC: O(1 + chain length)
- `get(K)`
  - LP: O(1 + run length)
  - EC: O(1 + chain length)
- `remove(K)`
  - LP: O(1 + run length)
  - EC: O(1 + chain length)

- 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

hashCode() rules



The general contract of `hashCode` is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is *not* required that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#hashCode()

Example : String Hash Functions

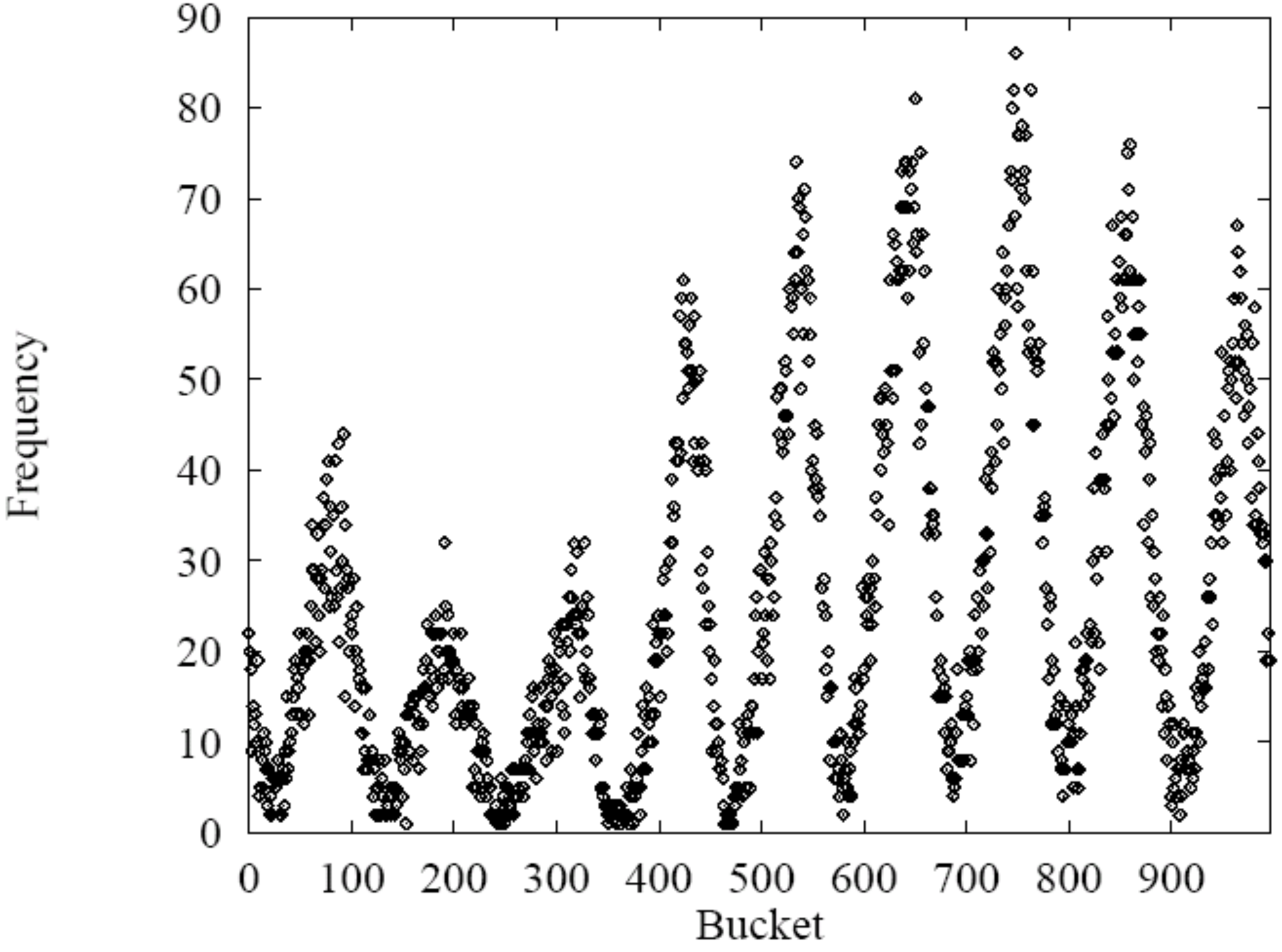
- What are some feasible hash functions for Strings?
  - First char ASCII value mapping
    - 0-255 only
    - Not uniform (some letters more popular than others)
  - Sum of ASCII characters
    - Not uniform - lots of small words
    - smile, limes, miles, slime are all the same

Example Hash Functions

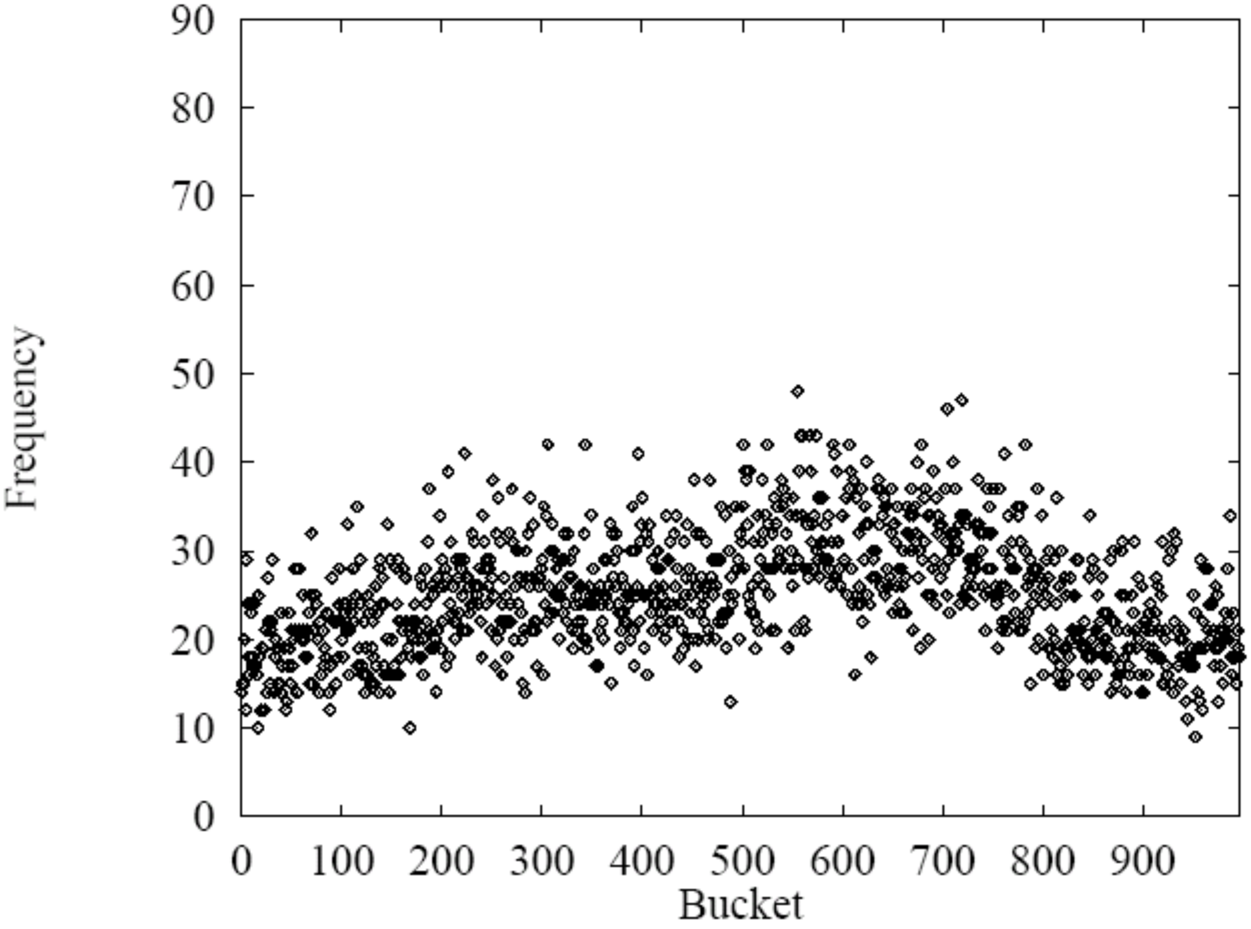
- String hash functions
  - Weighted sum
    - Small words get bigger codes
    - Distributes keys better than non-weighted sum
  - Let’s look at different weights…

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

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

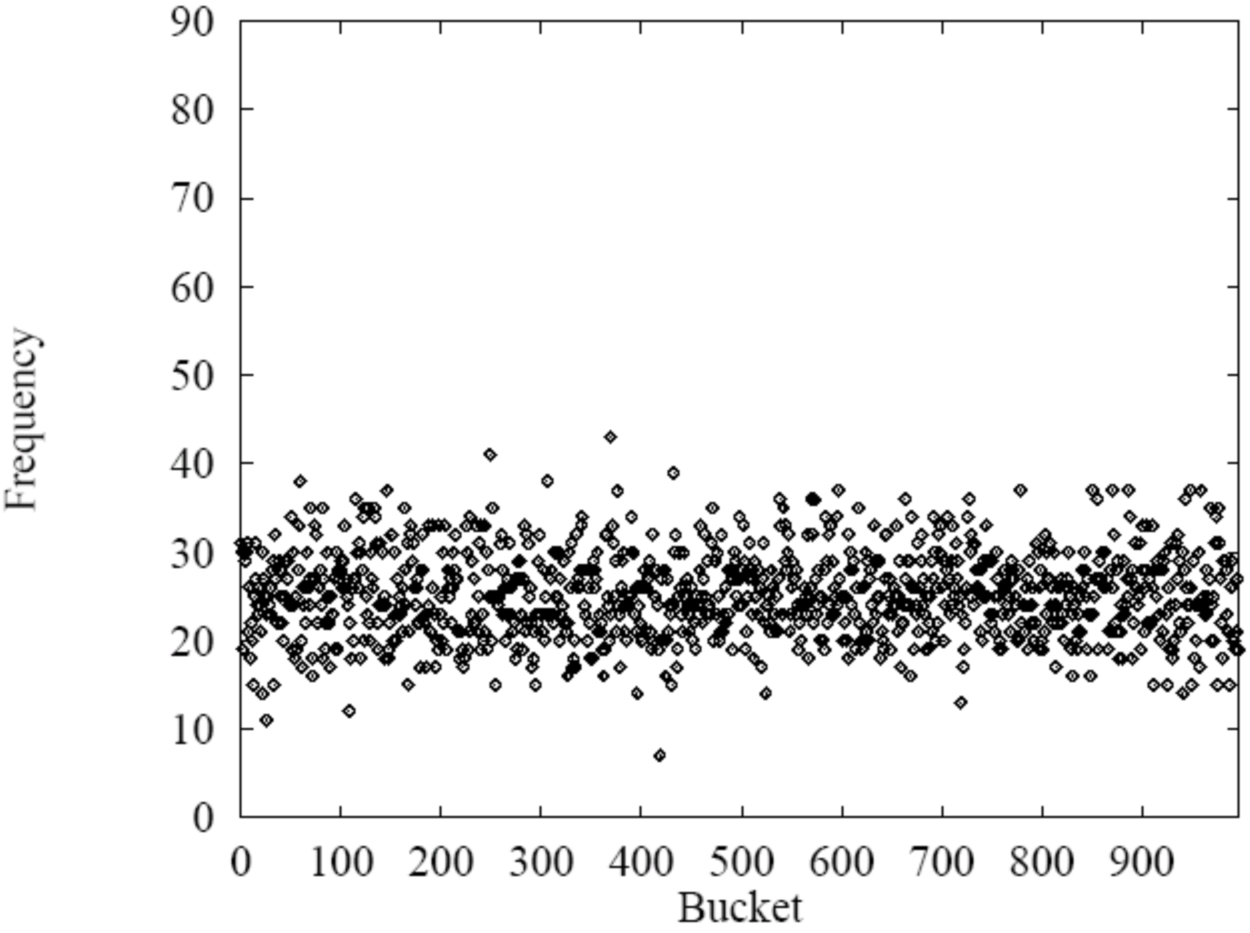


$$\sum_{i=0}^{n} \text{s.charAt(i)} * 2^{i}$$



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

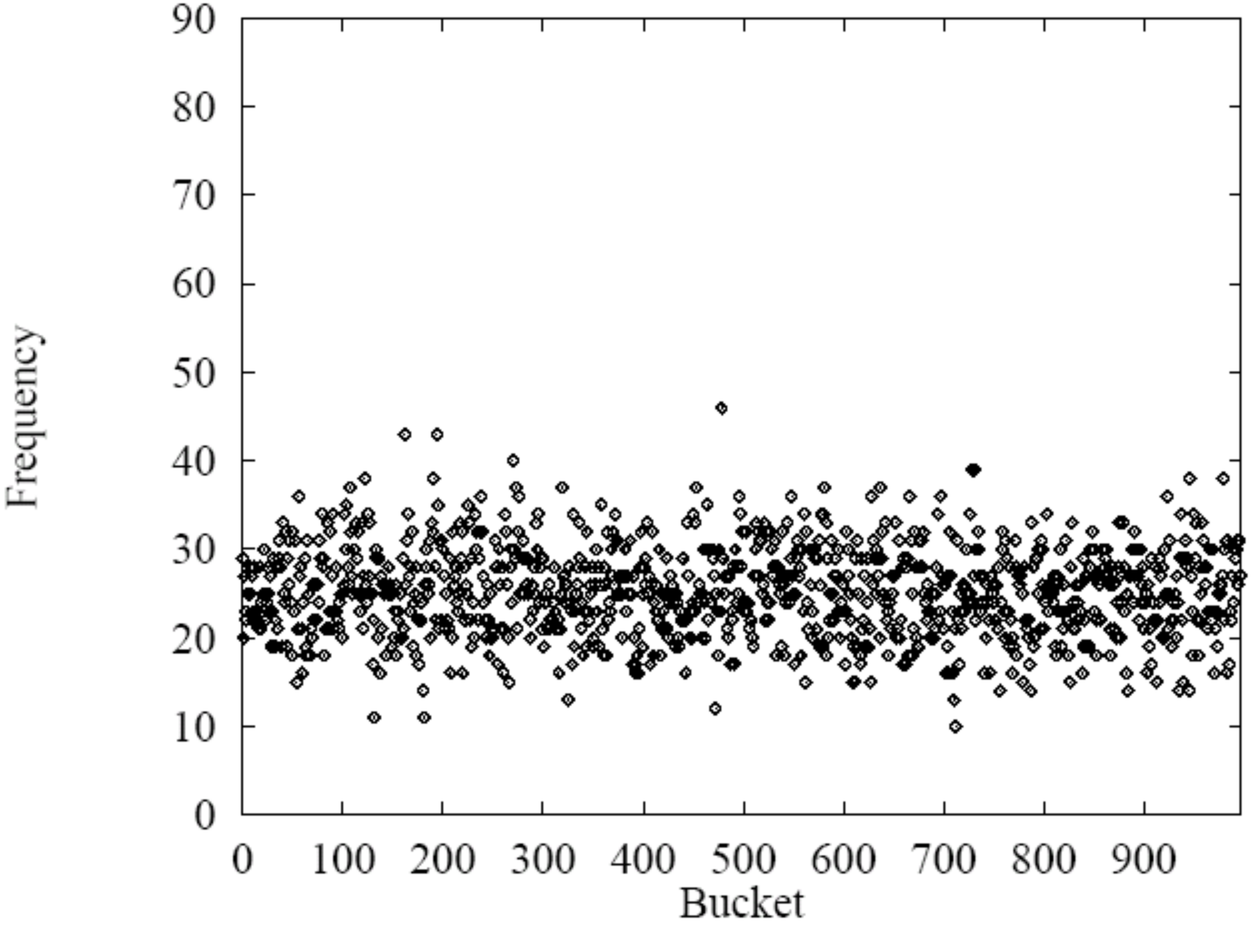
This looks pretty good, but 256^{i} is big…



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

Java uses:

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



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) PLUS cost of .equals() method
- Conclusion: for a good hash function (fast, uniformly distributed) and small load factor, we say operations take O(1) time
  - But that's not strictly true….

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)

*PolitiFact Rating: not quite Pants on Fire

What Can We Say For Sure?!

For external chaining

- Assuming the hashing function is equally likely to hash to any slot

Theorem: A search will take O(1 + m/n) time, on average

- n is table size, m is number of keys stored
- True for both successful and unsuccessful searches
  - Based on expected chain length

What Can We Say For Sure?!

For open addressing

- Assuming that all probe sequences are equally likely [which is unlikely!]
- Assuming load factor α = m/n

Theorem: An unsuccessful search will perform, on average, O(1 + α) probes

Theorem: A successful search will perform, on average, O($\frac{1}{\alpha} \log \frac{1}{1-\alpha}$) probes

More probe sequences ⇒ better average case

Perfect Hashing

In certain cases, it is possible to design a hashing scheme such that

- Computing the hash takes O(1) time
- There are no collisions
  - Different keys always have different hash values

This is called a *perfect hashing scheme*

Perfect Hashing

If keyspace is smaller than array size

- Handcraft the hashing function
  - Ex: Reserved words in programming languages
- Make array really big
  - Ex: All ASCII strings of length at most 4
    - Hash is 32 bit number
    - Array of size 4.3 billion will suffice