CSCI 136
Data Structures &
Advanced Programming

Fall 2019

Lecture 33

2070567 & 82879

Administrative Details

Reminders

- No lab this week
- Final exam
    - Monday, December 16 at 9:30 in TCL 123 (Wege)
    - Covers everything, with strong emphasis on post-midterm
    - Study guide, sample exam will be posted on handouts page

Topics Covered

- Vectors (and arrays)
- Complexity (big O)
- Recursion + Induction
- Searching
- Sorting
- Linked Lists (SLL & DLL)
- Stacks
- Queues
- Iterators
- Bitwise operations
- Comparables/Comparators
- OrderedStructures
- Binary Trees
- Priority Queues
- Heaps
- Binary Search Trees
- Graphs
- Maps/Hashtables

Last Time

- Graph applications (more in Ch 16)
  - Dijkstra's Algorithm for shortest paths
    - Single source
  - Prim's algorithm for MCST

Today’s Outline

- Quick Dijkstra’s example
- Maps
  - Revisit Naïve implementation from Lab 2
  - structure5.Hashtable (finally)
    - Hash functions
    - “Load factor”
    - Collisions and how to handle them
  - You should also read Ch 15 for more info

Final Topic: Maps and Hashing

Map Interface

- Used in GraphList and GraphMatrix
- Also (essentially) used elsewhere
  - Lab 2 (wordgen): Table - for each string seen in the text, what is the FrequencyList of following characters?
  - Lab 6 (postscript): SymbolTable - for each stored symbol, what did the user define as its value?

Map Interface

Methods for `Map<K, V>`

- `int size()` - returns number of entries in map
- `boolean isEmpty()` - true iff there are no entries
- `boolean containsKey(K key)` - true iff key exists in map
- `boolean containsValue(V val)` - true iff val exists at least once in map
- `V get(K key)` - get value associated with key
- `V put(K key, V val)` - insert mapping from key to val, returns value replaced (old value) or null
- `V remove(K key)` - remove mapping from key to val
- `void clear()` - remove all entries from map

Map Interface

Other methods for Map<K,V>:

- void putAll(Map<K,V> other) - puts all key-value pairs from Map other in map
- Set<K> keySet() - return set of keys in map
- Set<Association<K,V>> entrySet() - return set of key-value pairs from map
- Structure<V> valueSet() - return structure containing all the values
- boolean equals() - used to compare two maps
- int hashCode() - returns hash code associated with data in map (stay tuned…)

Dictionary.java

```
public class Dictionary {

    public static void main(String args[]) {
        Map<String, String> dict = new Hashtable<String, String>();
        …
        dict.put(word, def);
        …
        System.out.println("Def: " + dict.get(word));
    }

}
```

What's missing from the Map API that a BST provides?

`successor(key), predecessor(key)`

Maps do NOT preserve order!

Simple Implementation: MapList

- Uses a SinglyLinkedList of Associations as underlying data structure
  - Think back to Lab 2, but a List instead of a Vector
- How would we implement `get(K key)`?
- How would we implement `put(K key, V val)`?

MapList.java

```
public class MapList<K, V> implements Map<K, V>{

    //instance variable to store all key-value pairs
    SinglyLinkedList<Association<K,V>> data;

    public V put (K key, V value) {
        Association<K,V> temp =
                            new Association<K, V> (key, value);
        // Association equals() just compares keys
        Association<K,V> result = data.remove(temp);

        data.addFirst(temp);
        if (result == null)
                return null;
        else
                return result.getValue();
    }
}
```

Simple Map Implementation

- What is MapList's running time for:
  - containsKey(K key)?
  - containsValue(V val)?
- Bottom line: not O(1)!

Search/Locate Revisited

- How long does it take to search for objects in Vectors and Lists?
  - O(n) on average
- How about in BSTs?
  - O(log n)
- Can this be improved?
  - Hash tables can locate objects in *really quickly*!
    - (we will cover two reasons that O(1) performance is a fuzzy claim)

Example from Bailey

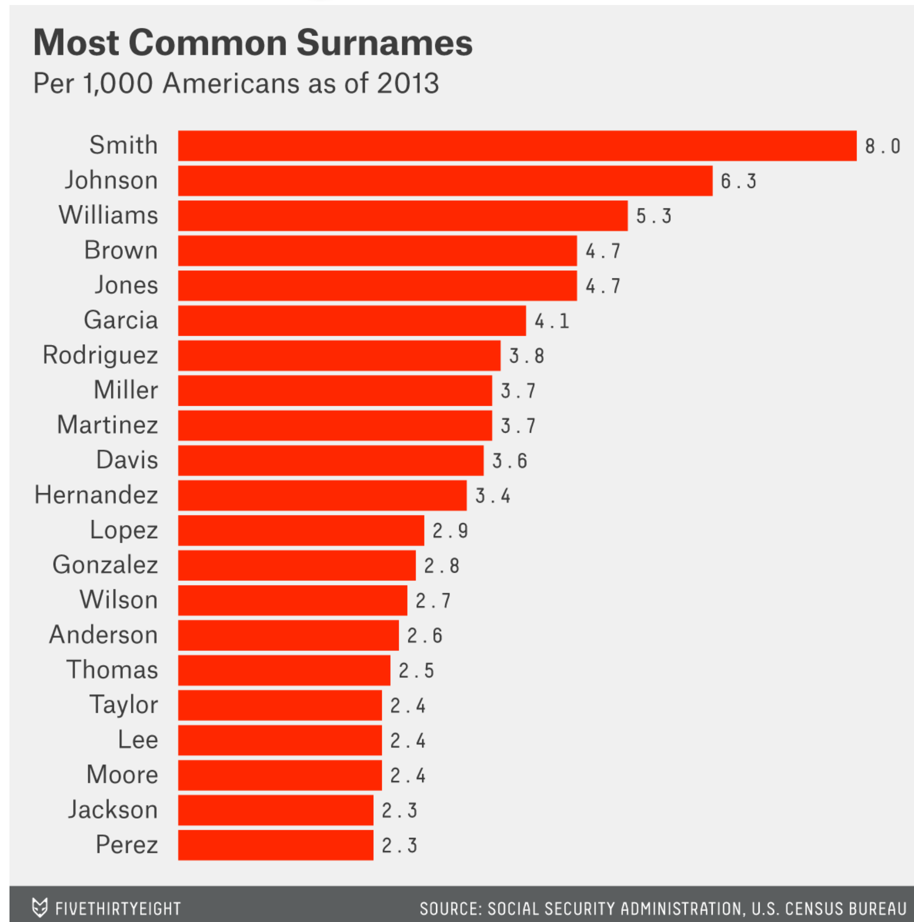
"We head to a local appliance store to pick up a new freezer. When we arrive, the clerk asks us for the last two digits of our home telephone number! Only then does the clerk ask for our last name. Armed with that information, the clerk walks directly to a bin in a warehouse of hundreds of appliances and comes back with the freezer in tow."

- Thoughts?
  - What is Key? What is Value?
  - Are names evenly distributed?
  - Are the last 2 phone digits evenly distributed?

Hashing in a Nutshell

- Assign objects to “bins” based on key
- When searching for object, go directly to appropriate bin (and ignore the rest)
- If there are multiple objects in bin, then search for the correct one
- Important Insight: Hashing works best when objects are evenly distributed among bins
  - Phone numbers are randomly assigned, last names are not

Hashing in a Nutshell



- Phone numbers are randomly assigned, last names are not

Implementing a HashTable

- How can we represent bins?
- Slots in array (for Associations)
- 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
- What if number is too big?
  - Take modulo array size (% in Java)

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

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()

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
  - Specify easy hash function: length
  - Initial array size = 7
  - Add “algorithms” to hash table
  - Add “data”
  - Add “bit”
- Let’s implement put(key, val) and get(key)…

First Attempt: put(K)

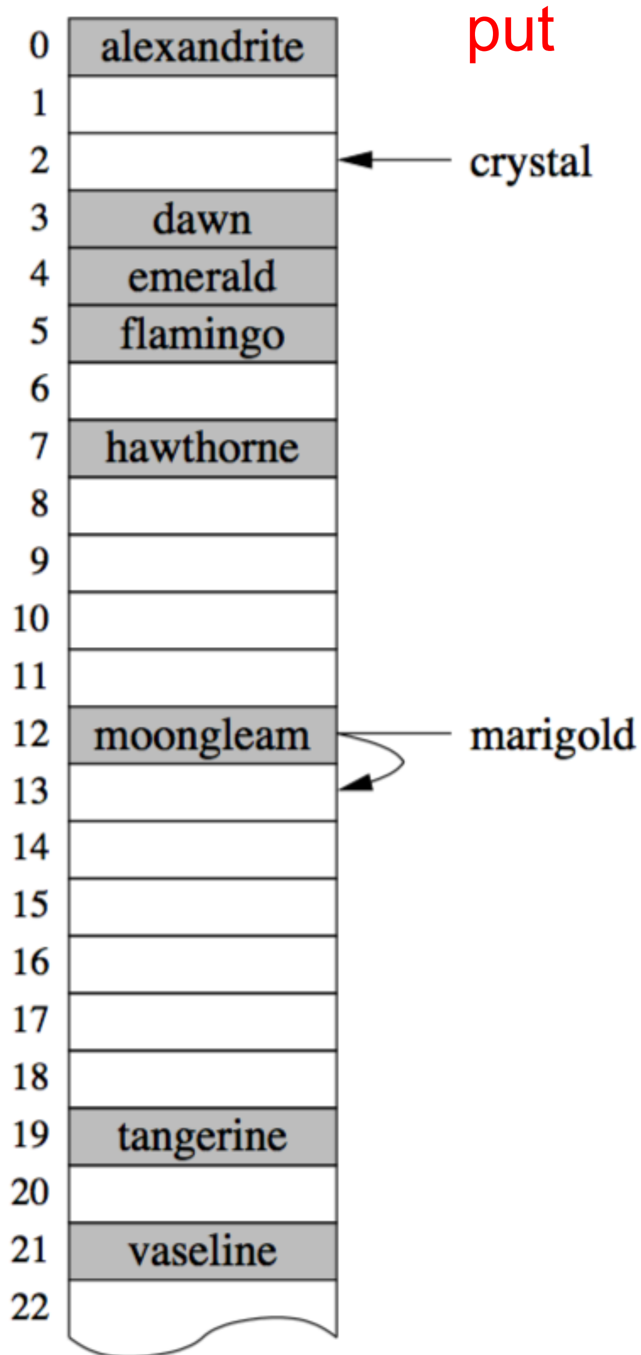
```
public V put (K key, V value) {
   int bin = key.hashCode() % data.length;
   while (true) {
      Association<K,V> slot = (Association<K,V>) data[bin];
      if (slot == null) {
         data[bin] = new Association<K,V>(key,value);
         return null;
      }
      if (slot.getKey().equals(key))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

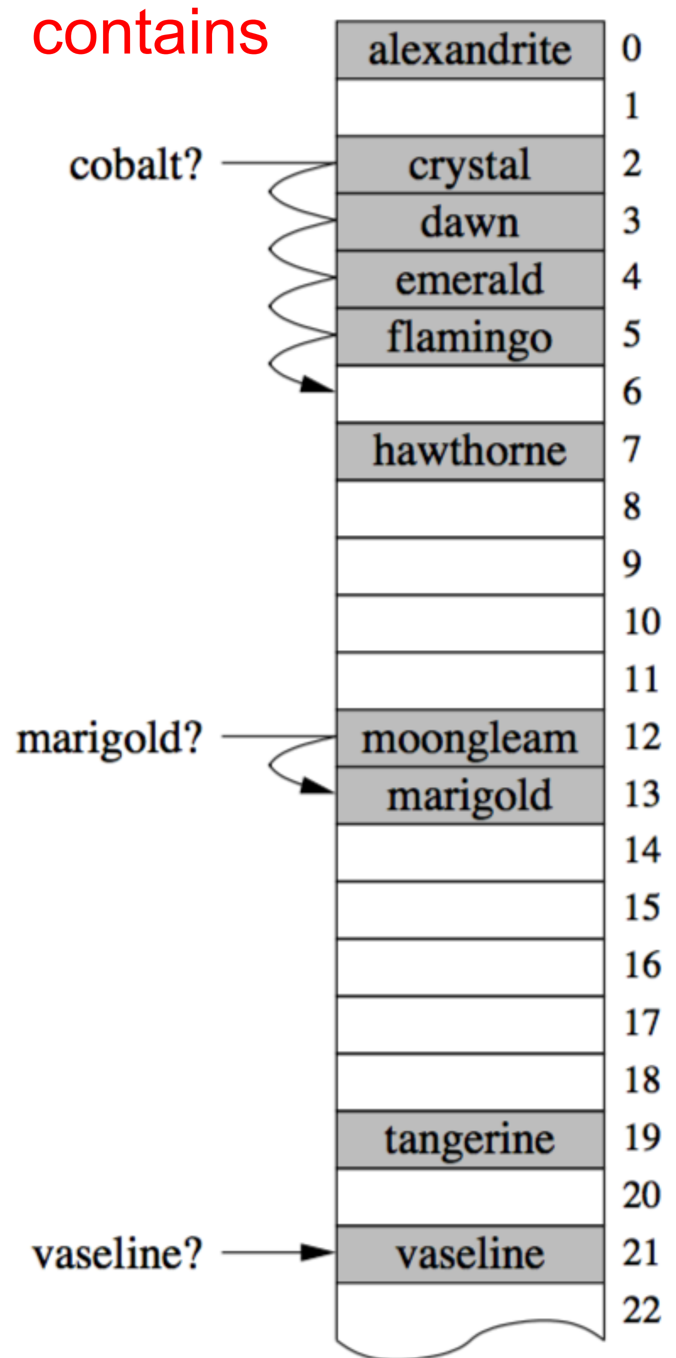
First Attempt: get(K)

```
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;
   }
}
```

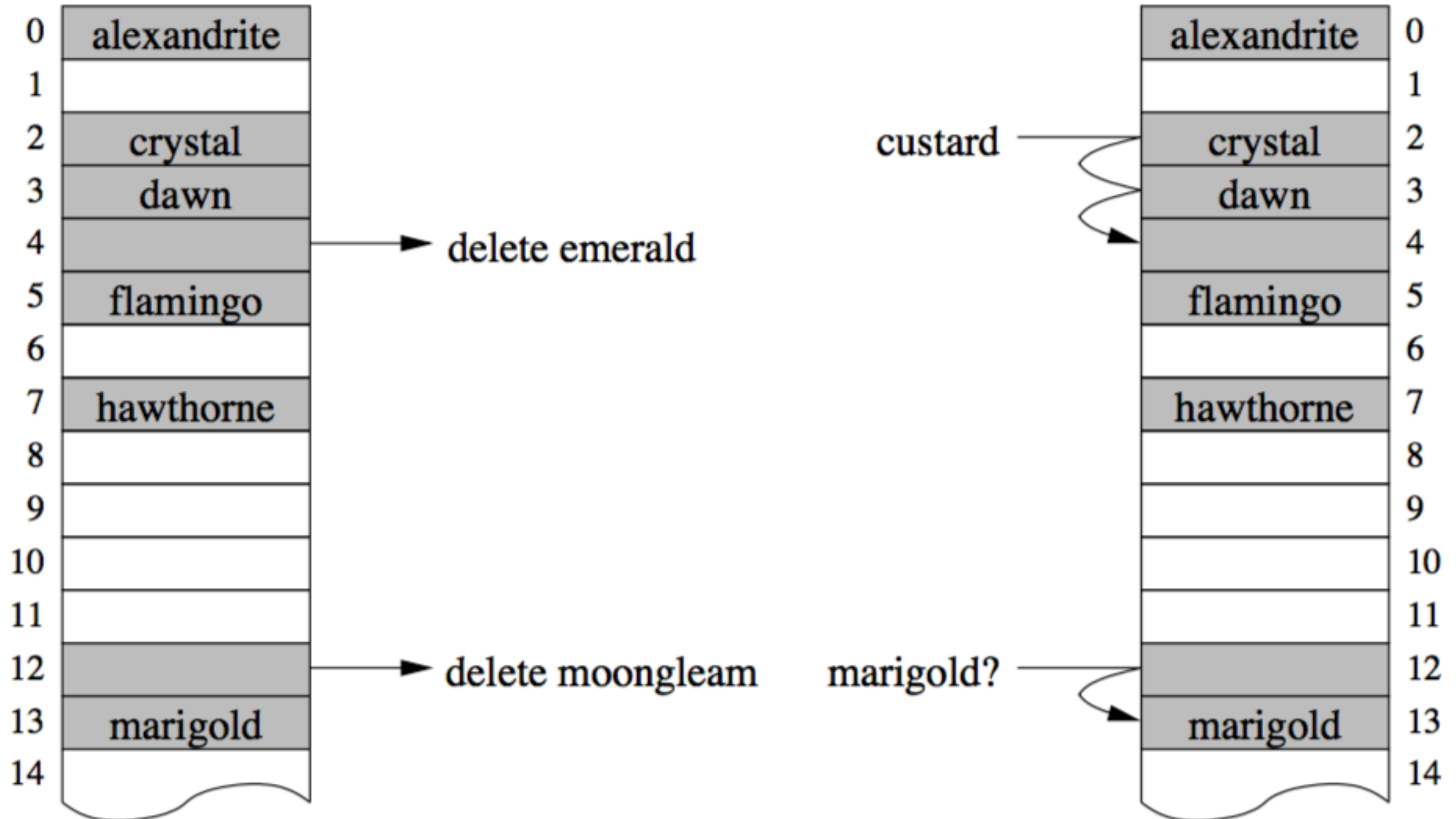

put
0
alexandrite
1
2
crystal
3
dawn
4
emerald
5
flamingo
6
7
hawthorne
8
9
10
11
12
moongleam
marigold
13
14
15
16
17
18
19
tangerine
20
21
vaseline
22


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

Linear Probing

- If a collision occurs at a given bin, just move forward (linearly) until an empty slot is available
  - Specify easy hash function: length
  - Initial array size = 7
  - Add “algorithms” to hash table
  - Add “data”
  - Add “bit”
- 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



Collisions & Clustering

- 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 cluster: A cluster that forms around the original hash location of a key
  - Secondary cluster: A cluster that forms along the sequence of rehashing locations of a key

Linear Probing

- NaiveProbing.java
  - We specify a dummy hash function: index of first letter of word
  - Initial array size = 8
  - Add “air hockey” to hash table
  - Add “doubles ping pong”
  - Add “quidditch”
- What happens when we remove “air hockey”, and then lookup “quidditch”?
  - Our *run* was broken up!
  - We need a “placeholder” for removed values to preserve runs…
- See Hashtable.java in structure5

Open Addressing : Quadratic Probing

- With linear probing, the ith probe for key k occurs at location (h(k) + i) % arraySize (assuming h(k) is non-negative)
- With *quadratic probing*, the ith probe for key k occurs at location (h(k) + i^2) % arraySize (starting with i = 0)
- Quadratic probing helps to avoid primary (but not secondary) 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 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

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

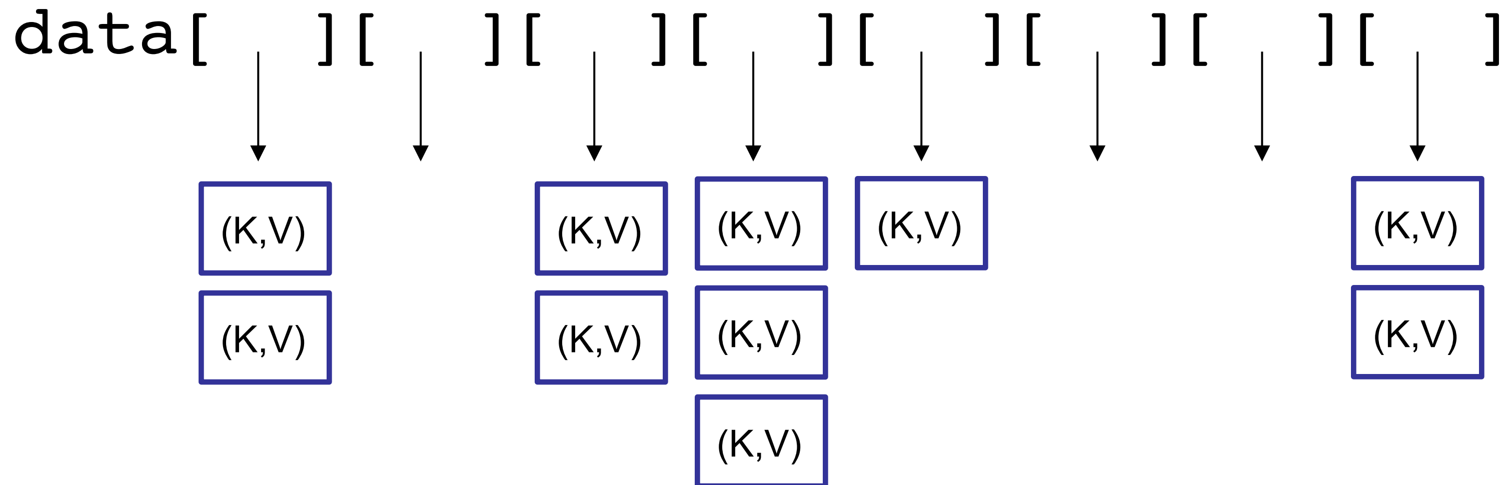
- With *double hashing* a second hashing function h'(k) is used to determine the probe sequence of k if location h(k) is full
- The ith probe for key k occurs at location (h(k) + i*h'(k) % arraySize (starting with i = 0)
- A good secondary hashing function needs to ensure that
  - $h'(k) \neq 0$
  -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

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

Example Hash Functions

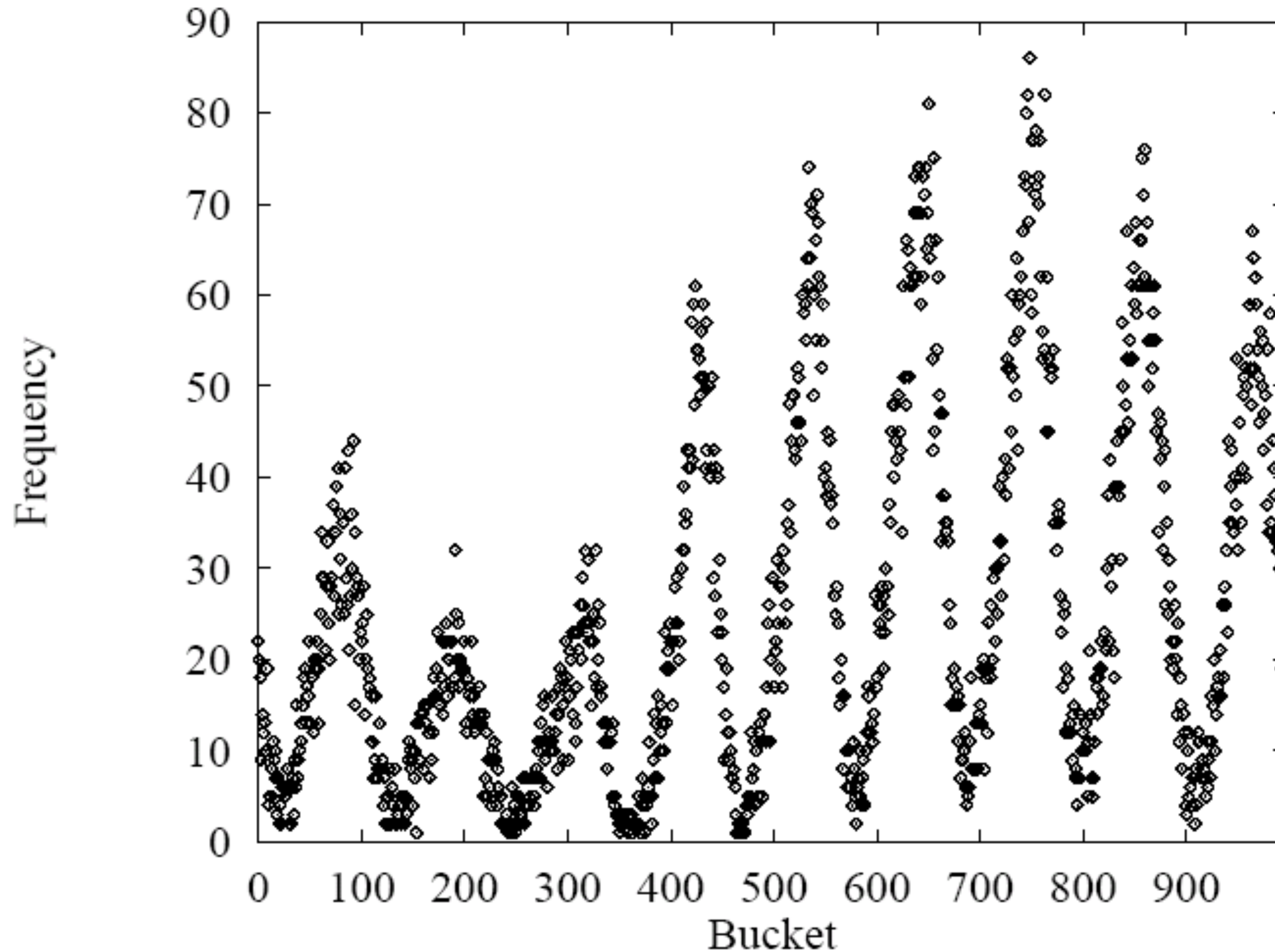
- What are some feasible hash functions for Strings?
  - First char ASCII value mapping
    - 0-255 only
    - Not uniform (some letters more popular than others)
  - Sum of ASCII characters
    - Not uniform - lots of small words
    - smile, limes, miles, slime are all the same

Example Hash Functions

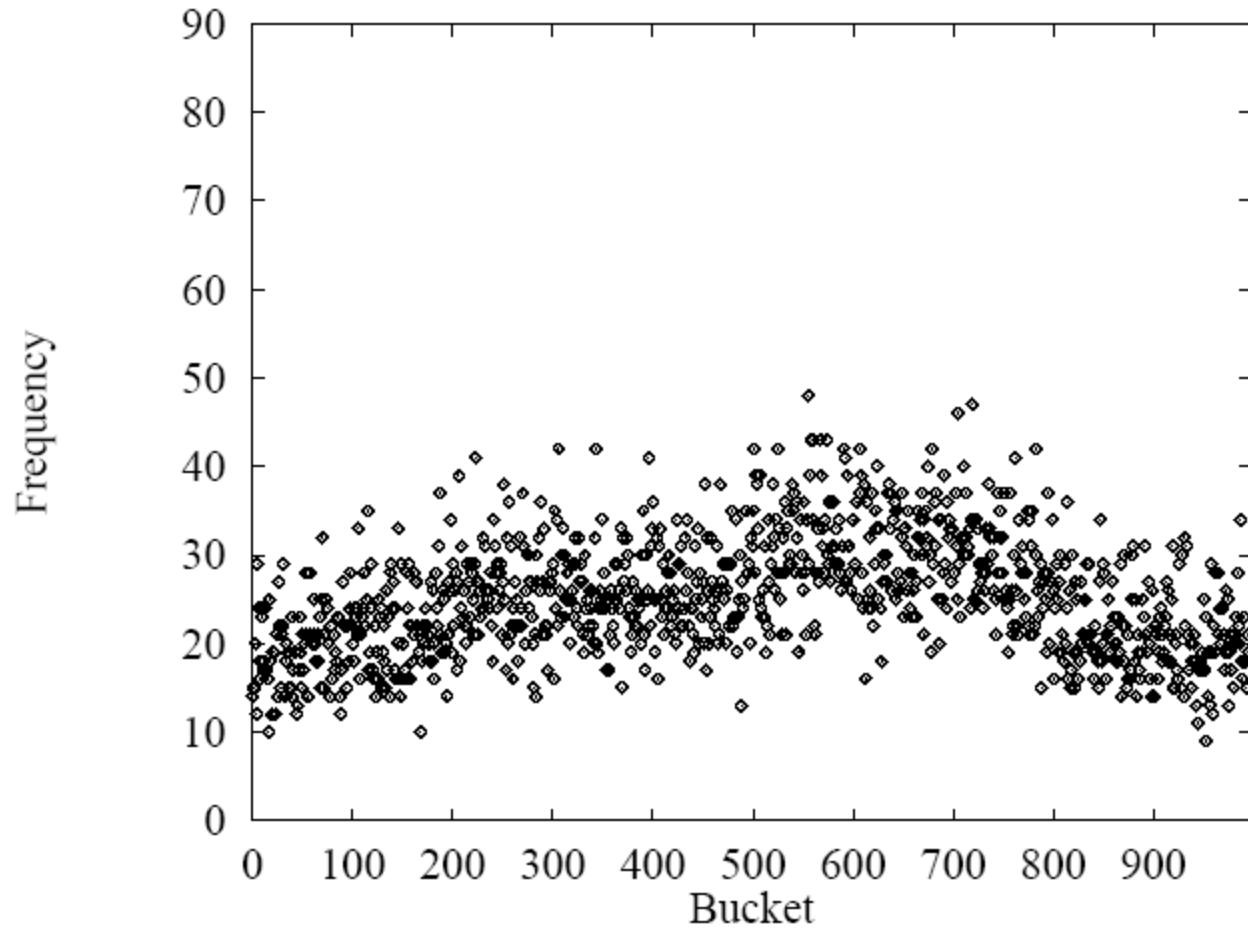
- String hash functions
  - Weighted sum
    - Small words get bigger codes
    - Distributes keys better than non-weighted sum
  - Let's look at different weights…

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

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

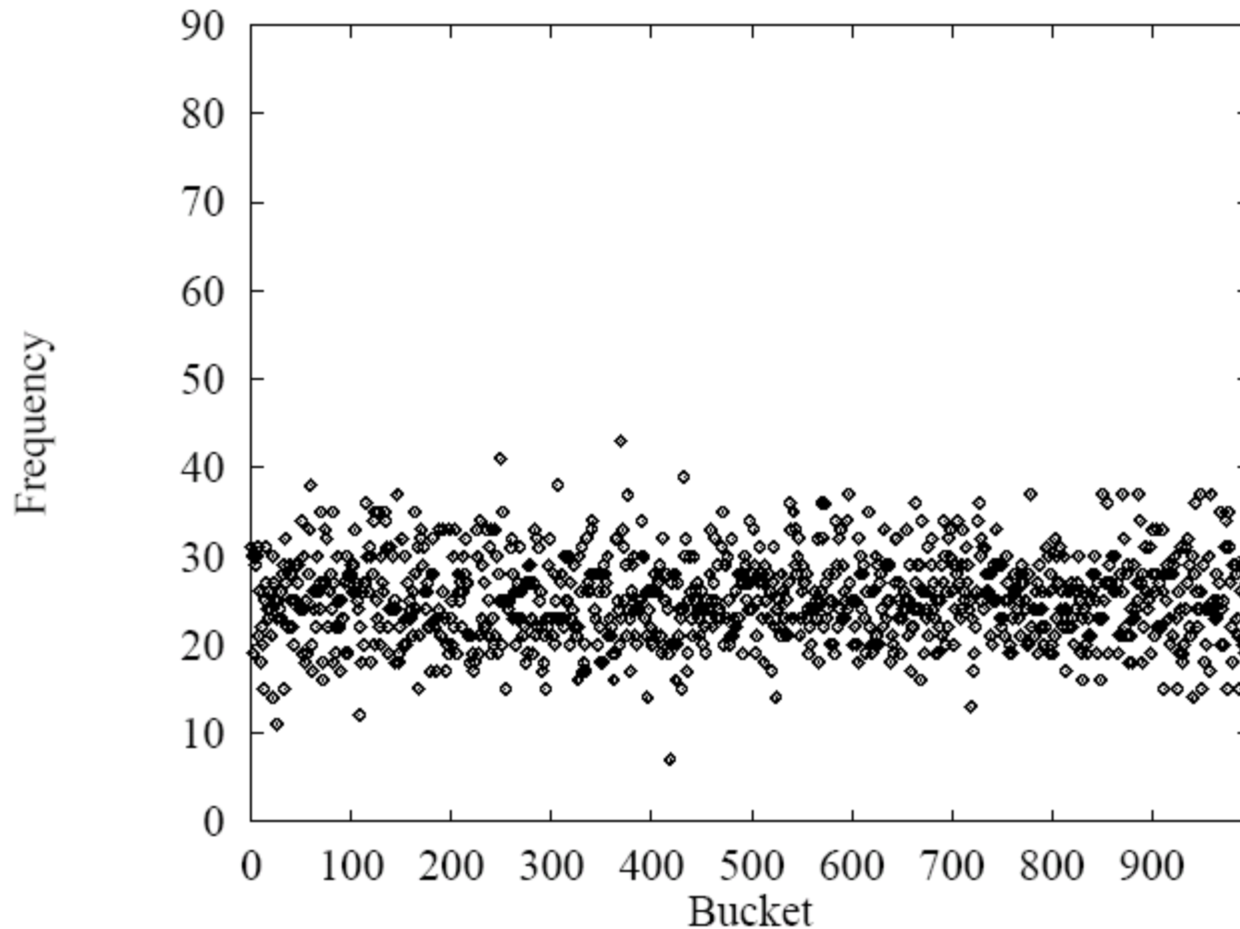


$$\sum_{i=0}^{n} s.charAt(i) * 2^{i}$$



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

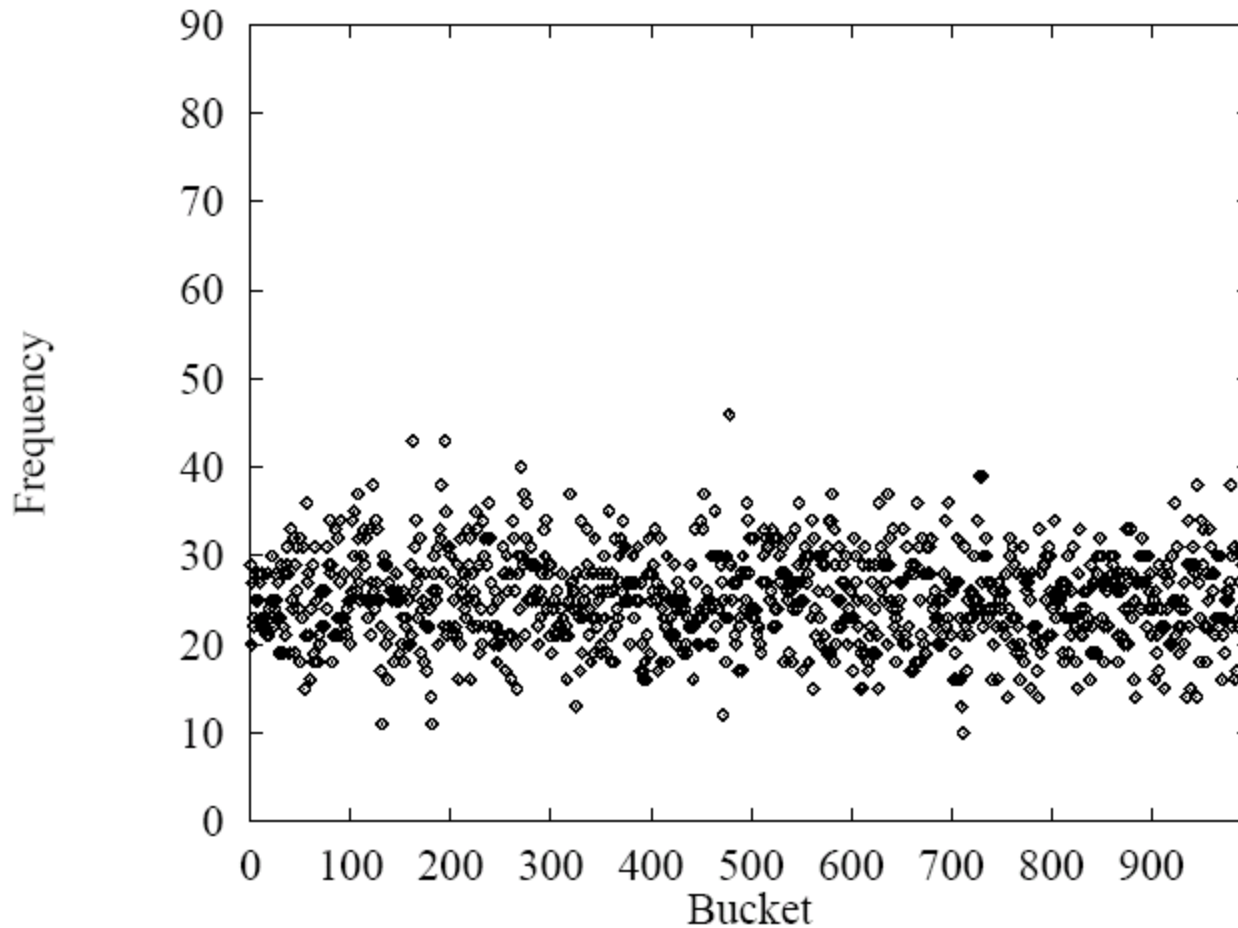
This looks pretty good, but 256^i is big…



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

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)