CSCI 136
Data Structures &
Advanced Programming

Lecture 33

Fall 2017

Instructors: 64187692

Announcements

- No Lab This Week
- This Wednesday
  - Problem Set is due
- This Friday
  - SCS Forms
- Final Exam is Thursday, December 14
  - 9:30-noon in Biology 112
  - Cumulative, but focused on second half of course

Last Time

- Finished Prim's Algorithm for min-cost spanning tree problem
- Presented Dijkstra's Algorithm for single-source shortes paths problem

Today

- Maps & Hashing

Maps

Recall the *Dictionary Problem*

- Store (key, value) pairs
  - Key is unique (no repeated keys)
  - Each key is associated with a value
  - Different keys can hold same value
  - Key/value pairs can be replaced to change value
- Goal: Fast storage and retrieval of information

The Map Interface

- Key Methods for Map<K, V>
  - 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
- As well as
  - int size() - returns number of entries in map
  - boolean isEmpty() - true iff there are no entries
  - void clear() - remove all entries from map

Map Interface : Additional Methods

- 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
  - Structure<V> valueSet() - return set of values
  - Set<Association<K,V>> entrySet() - return set of key-value pairs from map

Simple Implementation: MapList

- Think back to Lab 2, but a list instead of a Vector
- Uses a SinglyLinkedList of Associations as underlying data structure
- 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
    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 the running time of:
  - containsKey(K key)?
  - containsValue(V val)?

- Bottom line: not O(1)!

Hashing in a Nutshell

- Can we beat the O(log n) performance of BST structures on add/remove/contains *without* requiring keys to be comparable?
- Yes: In certain situations/on average
- And Introducing....
  - int hashCode() - returns hash code associated with map
    - *All* object types support this method
  - Use the hashCode method for the key type
- hashCode returns an int which can be used as an index into an array

Hashing in a Nutshell

- Warning: hashCode() value can be negative
  - The String class hashCode method can return negative values
    - "abcdefg".hashCode() yields -1206291356
  - Use abs(key.hashCode()) % array.length to find index

```
int index = abs(key.hashCode()) % array.length ;
```

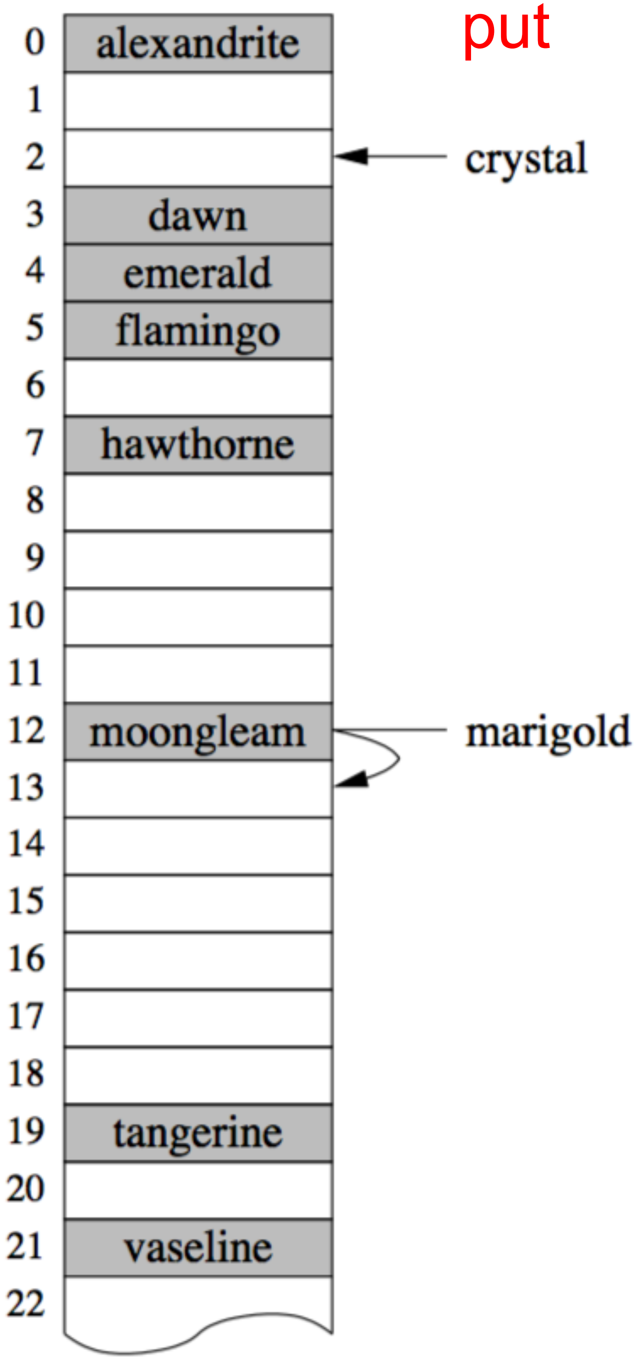
  - Or

```
int mask = 0b01111111_11111111_11111111_11111111;
int index = (key.hashCode() & mask) % array.length ;
```

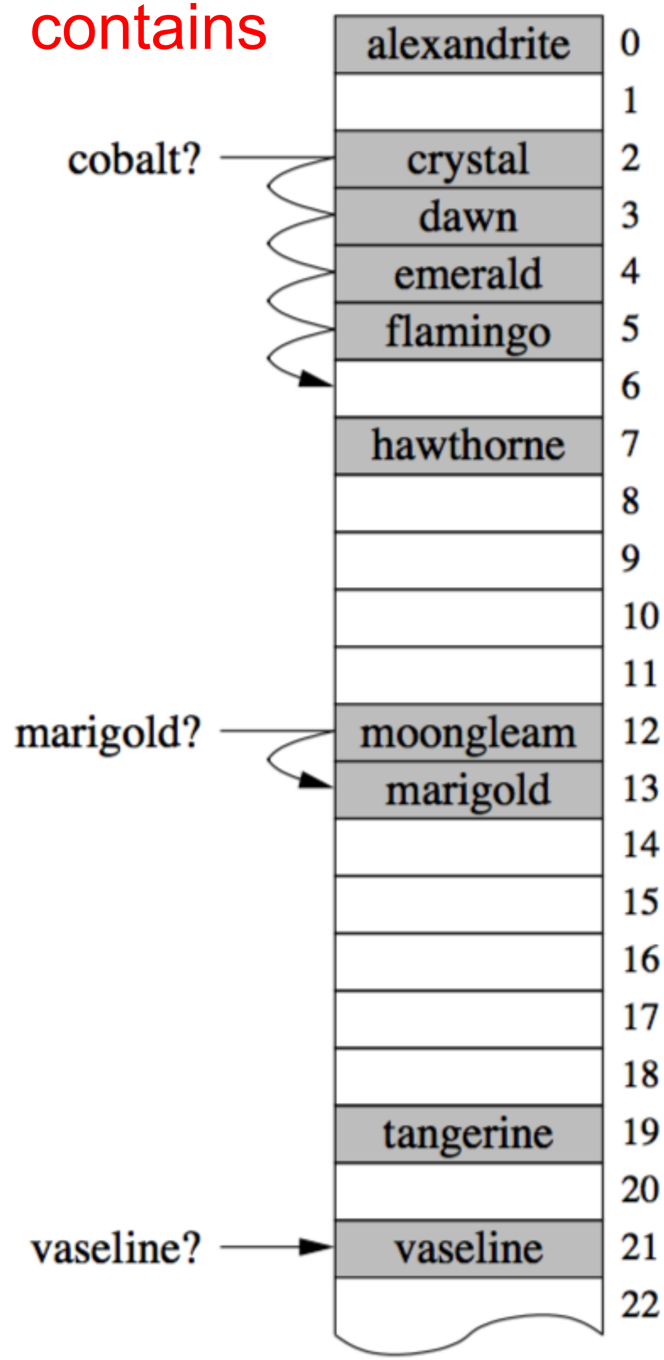
Hashing in a Nutshell

- Group objects into “bins” (indexed by ints)
- To add/remove/find an object
  - Compute its hashCode to get bin number
- If multiple objects hash to same bin (*collision!*), then search (somehow)
- Works best when objects are evenly distributed among bins

put



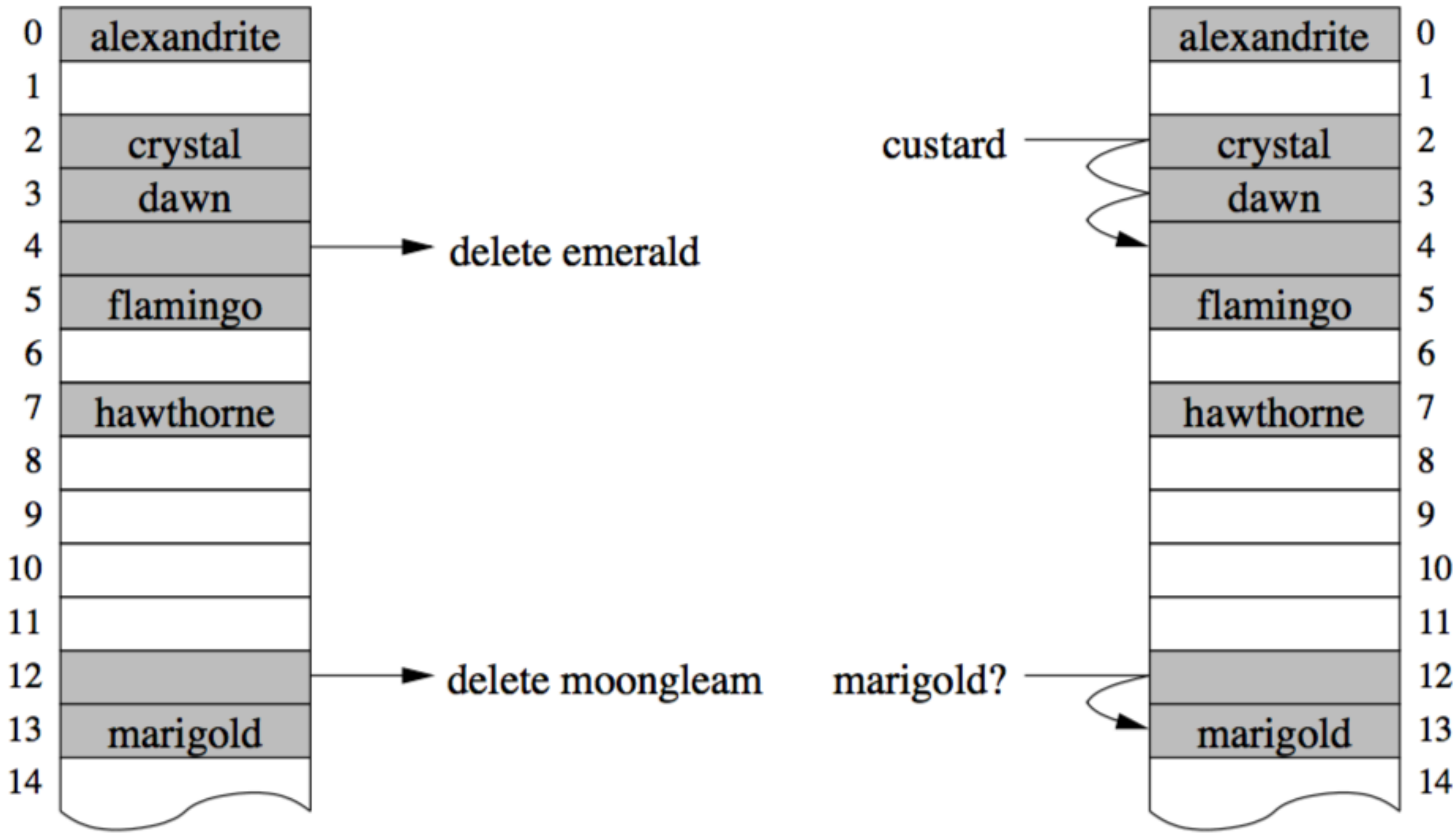
contains



Implementing HashTable

- How do we add Associations to the array?
  - Can get complicated if collisions occur
- Two approaches
  - Open addressing (using probing)
  - External chaining

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 clustering: keys with *the same* hash value fill in consecutively probed slots
  - Secondary clustering: keys with *different* hash value fill in consecutively probed slots

External Chaining

- Downsides of linear probing
  - What if array is almost full?
  - Linear probing is inefficient on almost-full arrays
- How can we avoid this problem?
  - Keep all values that hash to same bin in a "collection"
    - Usually a SLL
  - External chaining "chains" objects with the same hash value together

How Efficient is Hashing

- Linear probing:
  - put/get/remove all depend on time to find correct bin
- External chaining
  - put/get/remove depend on
    - time to find bin, plus
    - time to find element in bin's chain
- How can we optimize time to find right bin?

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, need to double size of array (a typical threshold is 0.6)
  - How?

Doubling Array

- Cannot just copy values---why?
  - Hash values may change
  - Example
    - Suppose key.hashCode() = 27. Then
      - key.hashCode() % 8 = 3;
      - key.hashCode() % 16 = 11;
- Have to recompute all hash codes

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

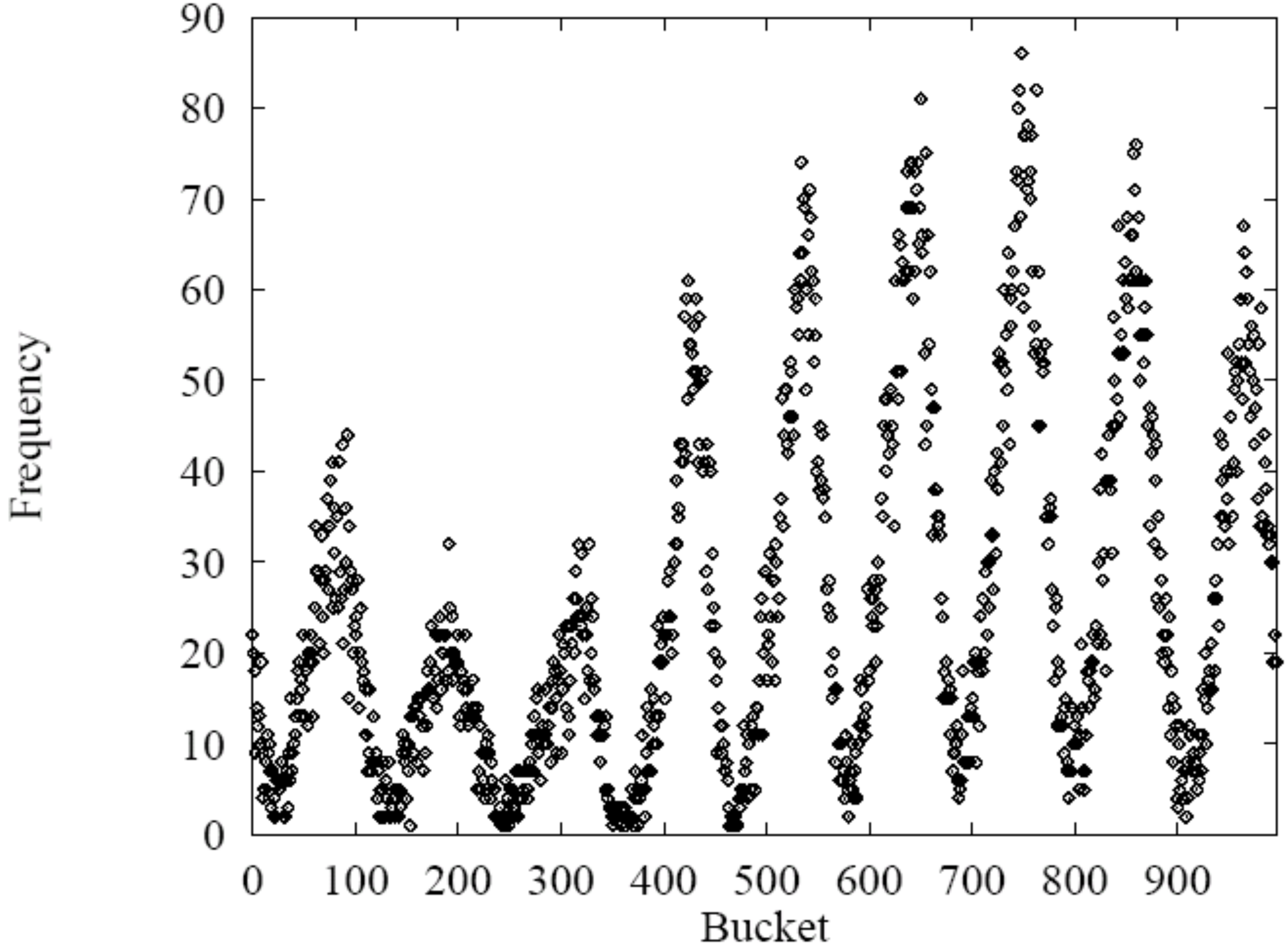
- What are some feasible hash functions for Strings?
  - First char ASCII value mapping
    - 0-255 only
    - Not uniform (some letters more popular than others)
  - Sum of ASCII characters
    - Not uniform - lots of small words
    - smile, limes, miles, slime are all the same

Example Hash Functions

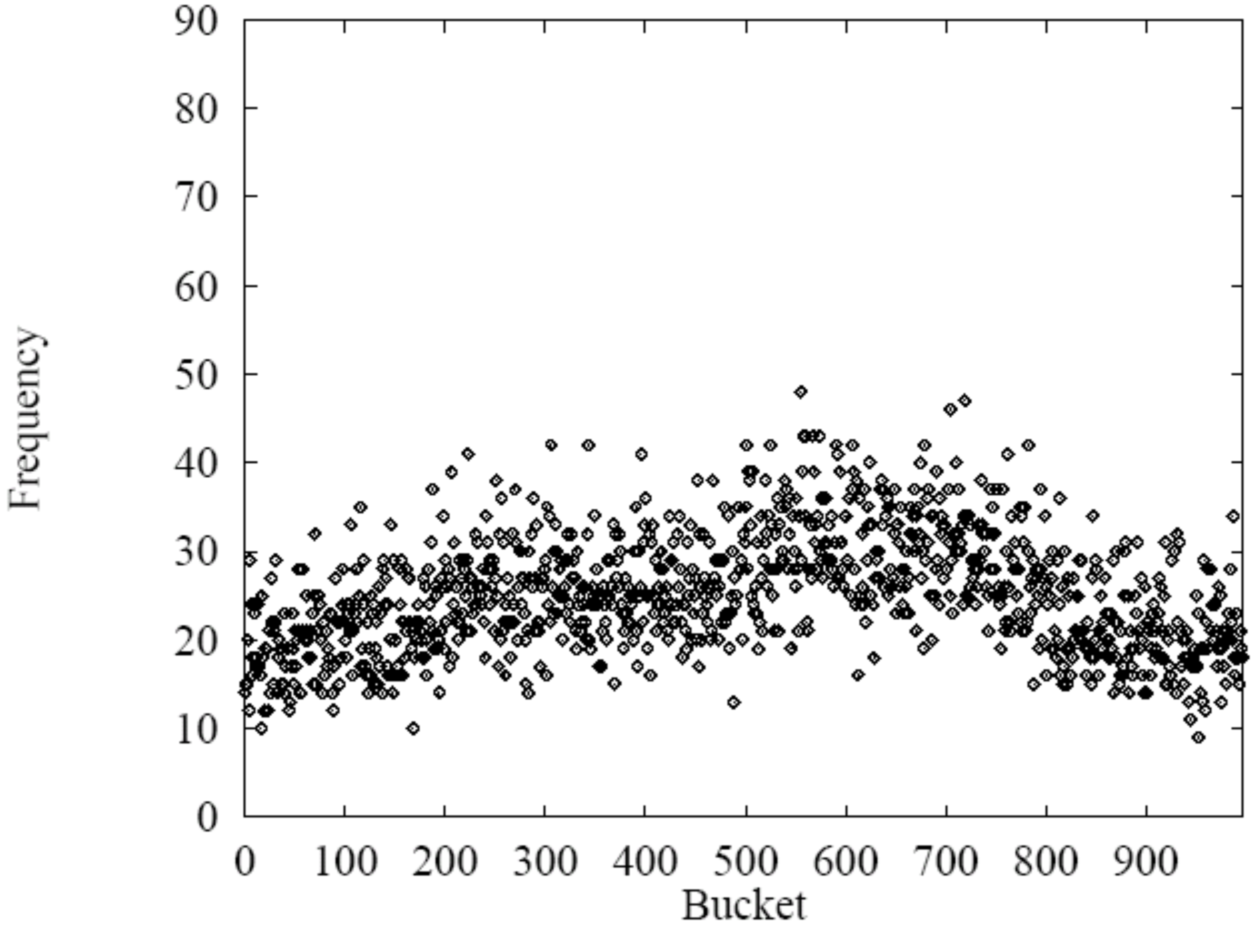
- String hash functions
  - Weighted sum
    - Small words get bigger codes
    - Distributes keys better than non-weighted sum
  - Let's look at different weights…

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

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

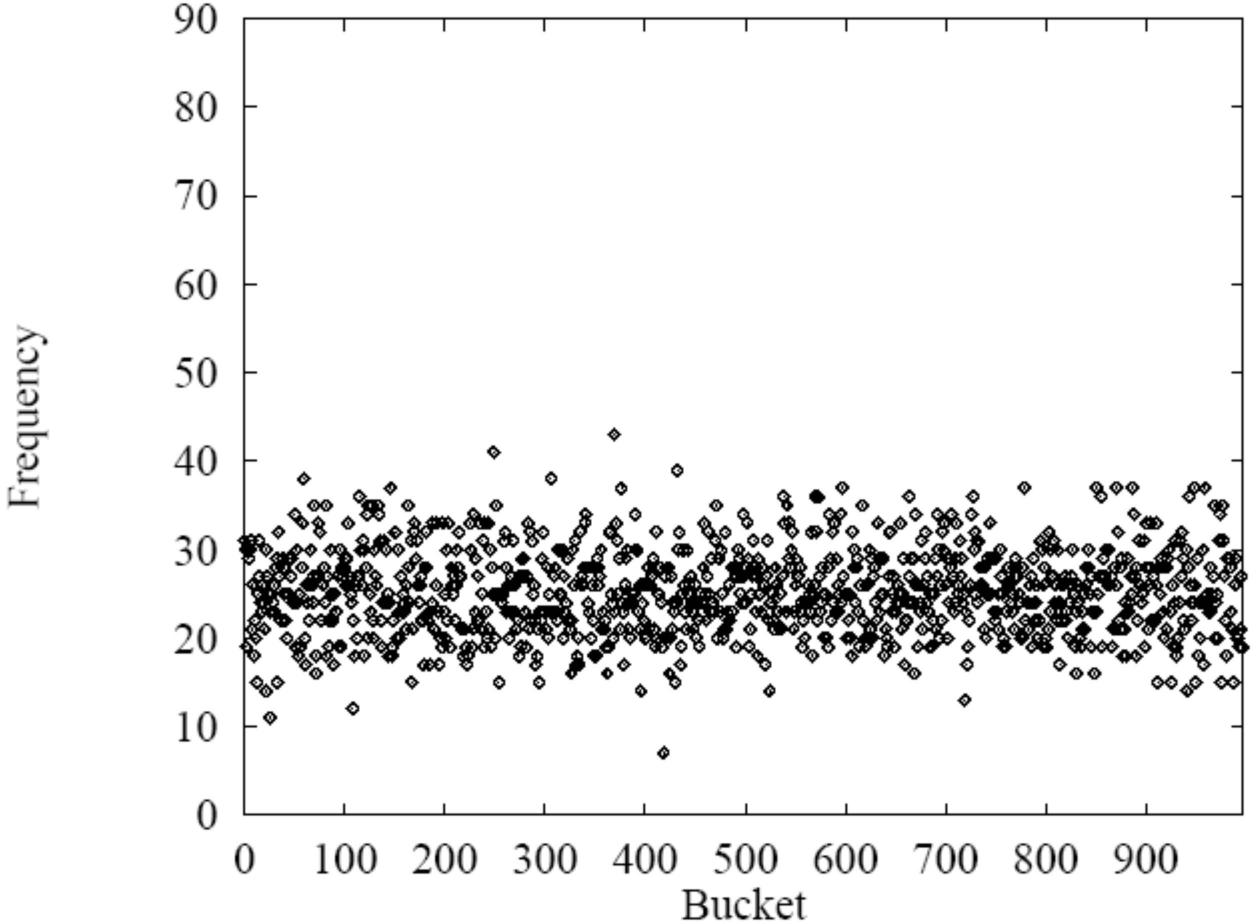


$\sum_{i=0}^{n} s.charAt(i) * 2^i$



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

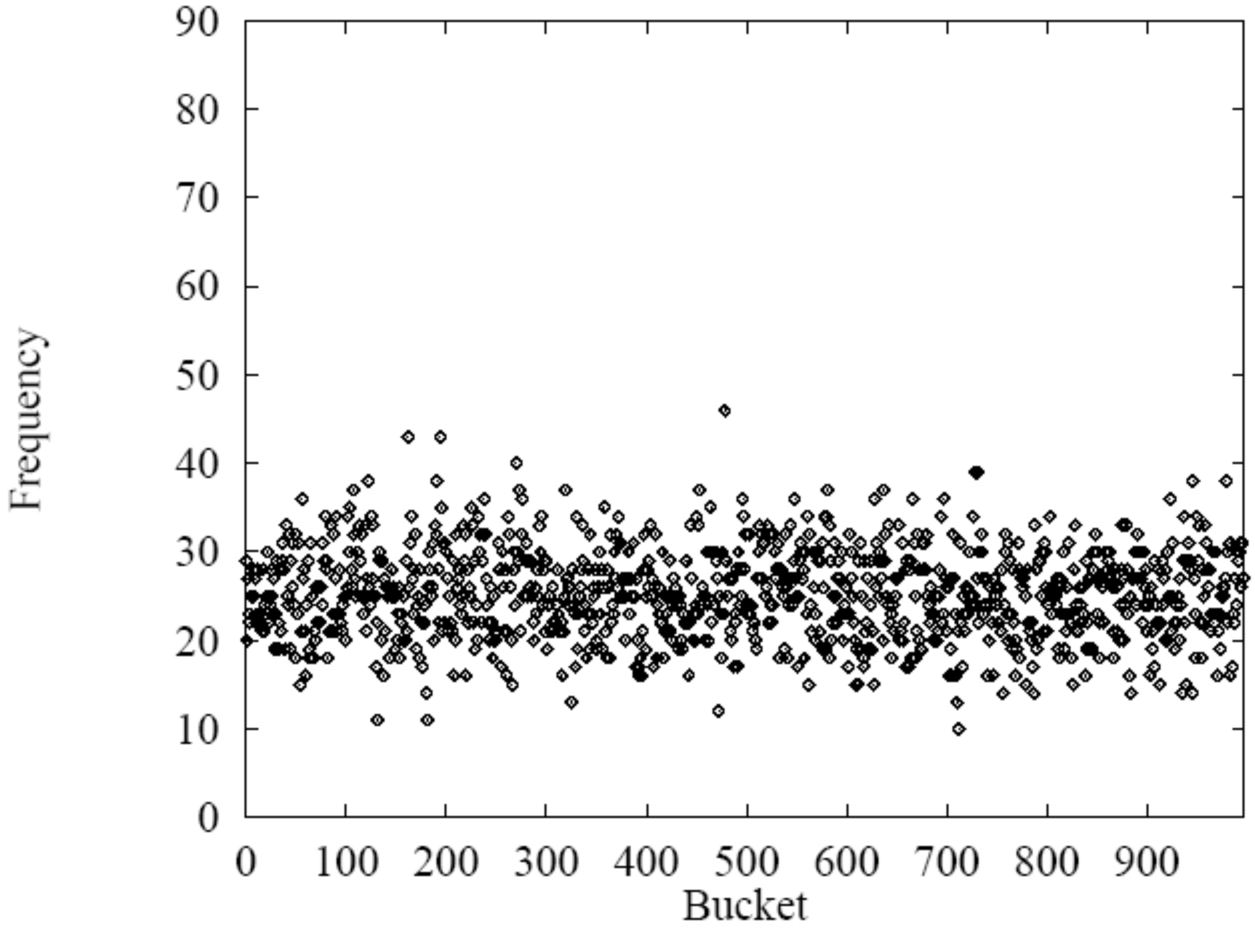
This looks pretty good, but 256^i is big…



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

Java uses:

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



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)

*On average---with good design---Don't forget!

The Search for the Perfect Hash

What would a “perfect” hashing scheme look like?

- If key1 ≠ key2 then key1.hashCode() ≠ key2.hashCode()
- hashCode values are in small range (a..b) (for array indexing)
  - Table size would be no larger than maximum key set
- hashCode can be computed quickly

Is such a thing possible?

- Yes---if key set is known and most keys will be used
  - Size of table will be proportional to size of key universe
  - Use external chaining
    - Replace SLL with secondary hash function