CSCI 136
Data Structures & Advanced Programming

Spring 2020

Lecture 28

Profs 2070567 and 68465

Last Time

- Hash tables implement the Map interface
  - [obj.hashCode() % array.length] assigns objects to bins
  - Collisions occur when multiple objects map to the same bin
  - We can resolve collisions using:
    - Linear probing (aka open addressing)
    - External chaining

Today’s Outline

- Correct our “straw man” Linear Probing
- External Chaining to resolve collisions
- Managing load factor
- A look at a real hash function

Linear Probing Review

- A hash function maps a key-value pair to a bin
- If two keys hash to the same bin, we have a collision
- Linear probing scans and places the collided element in the first available bin, creating a run

Linear Probing Challenge

- When we delete an element from a run, we create a “hole”
  - Challenge: How do we tell if the run has ended, or if the hole is from a deletion?
  - Solution: Insert a “placeholder”
    - If we see the placeholder during a lookup, we treat it as a collision
    - If we see the placeholder during insertion, we treat it as an open spot
      - We must still scan the run to see if our key is present

Hashtable.java

```
public class Hashtable<K,V> implements Map<K,V>, Iterable<V> {

    /* A single key-value pair to be used as a token
     * indicating a reserved location in the hashtable.
     * Reserved locations are available for insertion,
     * but cause collisions on lookup. */
    protected static final String RESERVED = "RESERVED";

    /* The data associated with the hashtable. */
    protected Vector<HashAssociation<K,V>> data;
```

Hashtable.java

```
protected int locate(K key) {
          // initial hash code
          int hash = Math.abs(key.hashCode() % data.size());
          // keep track of first unused slot, in case we need it
          int reservedSlot = -1;
          boolean foundReserved = false;
          while (data.get(hash) != null) {
                    // loop until end of run OR find target key
                    if (data.get(hash).reserved()) {
                              // remember reserved slot if we fail to locate value
                              if (!foundReserved) {
                                        reservedSlot = hash;
                                        foundReserved = true;
                              }
                    } else {
                              // value located? return the index in table
                              if (key.equals(data.get(hash).getKey())) return hash;
                    }
                    hash = (1+hash)%data.size();
          }
          // return first empty slot we encountered
          if (!foundReserved)
                    return hash;
          else
                    return reservedSlot;
}
```

Hashtable.java

```
public V get(K key) {
       // find bin where key lives (after resolving collisions)
       int hash = locate(key);

       // if the key is not found, the resulting location
       // is either null or “RESERVED”
       if (data.get(hash) == null ||
           data.get(hash).reserved())
              return null;

       // key was found, so return associated value
       return data.get(hash).getValue();
}
```

Hashtable.java

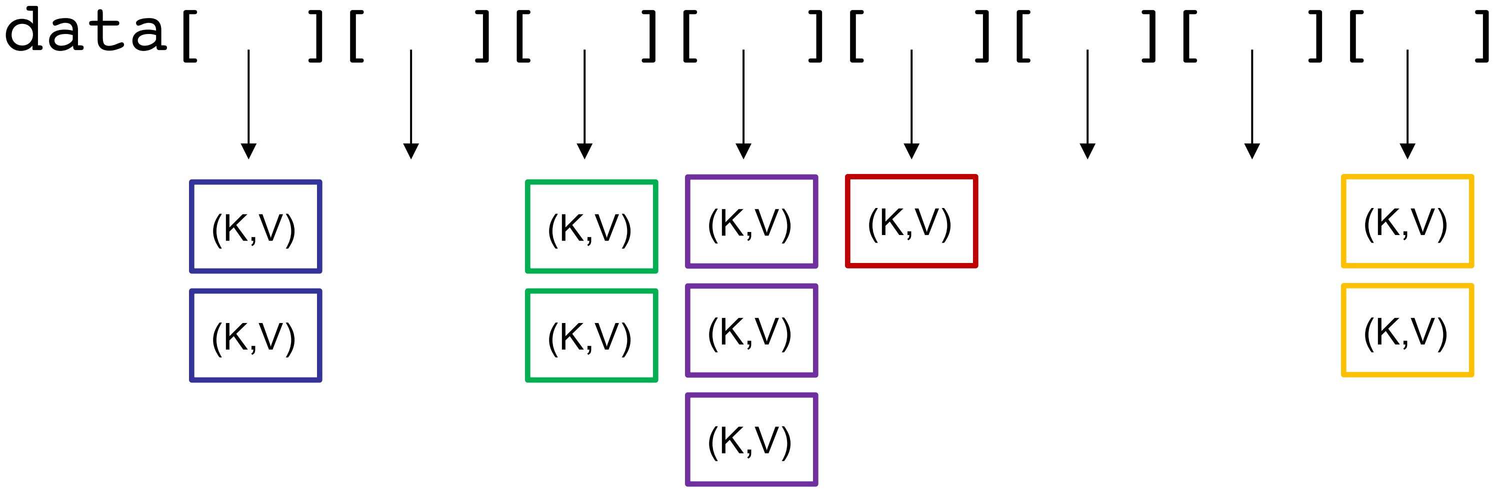
```
public V remove(K key) {
       // find bin where key lives (after resolving collisions)
       int hash = locate(key);

       // if the key is not found, the resulting location
       // is either null or “RESERVED”
       if (data.get(hash) == null ||
           data.get(hash).reserved())
             return null;

       // key was found, so remove, then return old value
       count--;
       V oldValue = data.get(hash).getValue();
       data.get(hash).reserve();
       return oldValue;
}
```

Observations

- Code becomes more complicated, but manageable
- The length of a run dictates the performance
- Reserving elements does not “shrink” the run–it defers the work to other operations
  - Keeping our runs small is important, so we may want to reexamine design decisions if we expect a lot of deletions

External Chaining

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



- Everything that hashes to bin$_i$ goes into list$_i$
  - `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)

- Run/Chain size is important. How do we control cluster/chain length?

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, grow size of array (typically threshold = 0.6)
  - Challenges?

Growing the Underlying Array

- Cannot just copy values
  - Why?
  - Key-value pairs' bins may change
  - Example: suppose (`key.hashCode() == 11`)
    - 11 % 7 = 4;
    - 11 % 13 = 11;
- Result: must recompute all hash codes, then reinsert key-value pairs into new array
- Also: try to keep array sizes relatively prime
  - Redistribute "clumps"

Good Hashing Functions

- Important point: All of this hinges on using “good” hash functions that spread keys “evenly”
- Good hash functions:
  - Are fast to compute
  - Distribute keys uniformly
- Unfortunately, we often have to test “goodness” empirically

Example Hash Functions

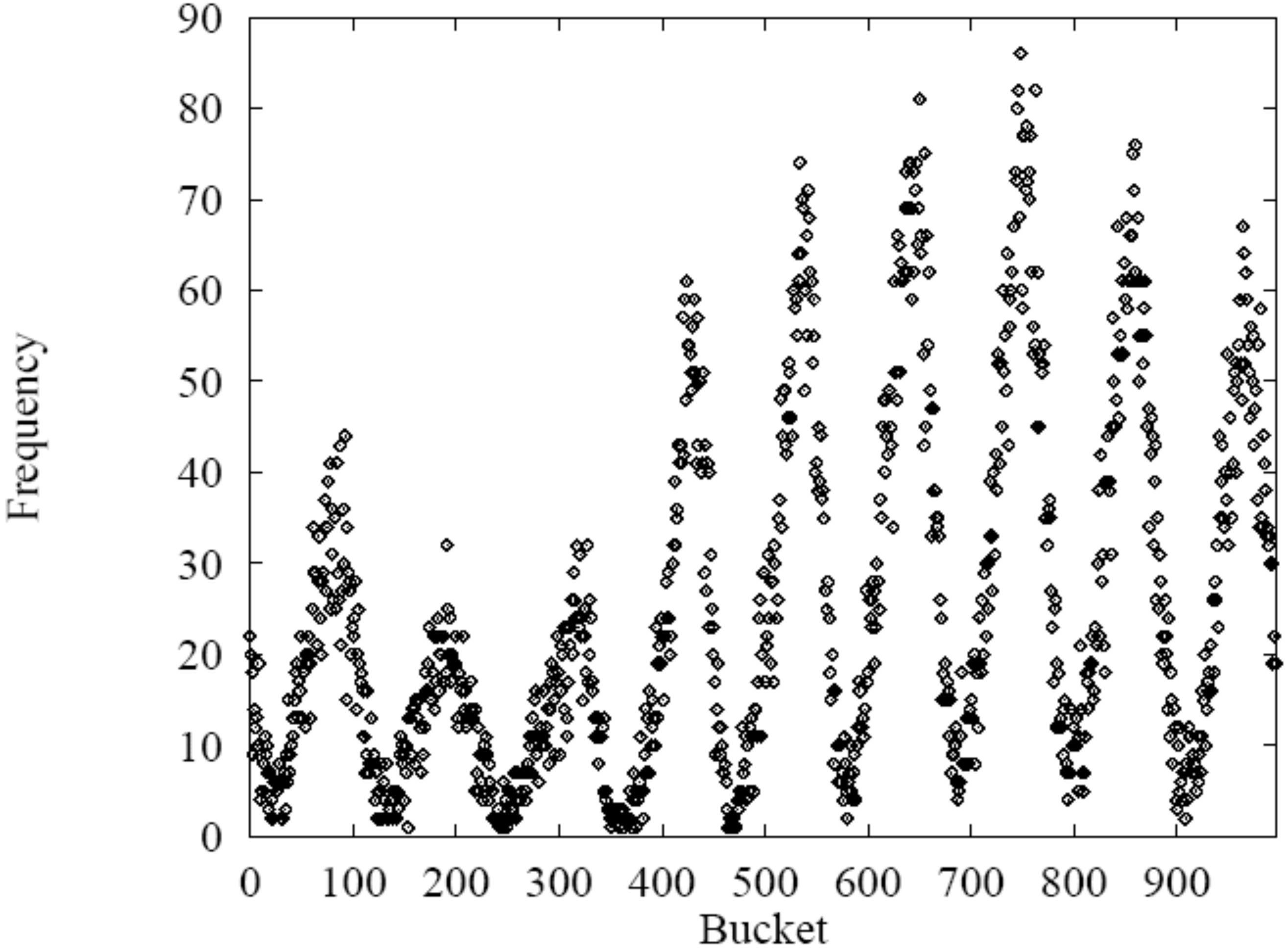
- What are some feasible hash functions for Strings?
    - Use the first char’s ASCII value?
        - 0-255 only
        - Not uniform (some letters more popular than others)
    - Sum of all characters’ ASCII values?
        - Not uniform - lots of small words
        - smile, limes, miles, slime are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

Example Hash Functions

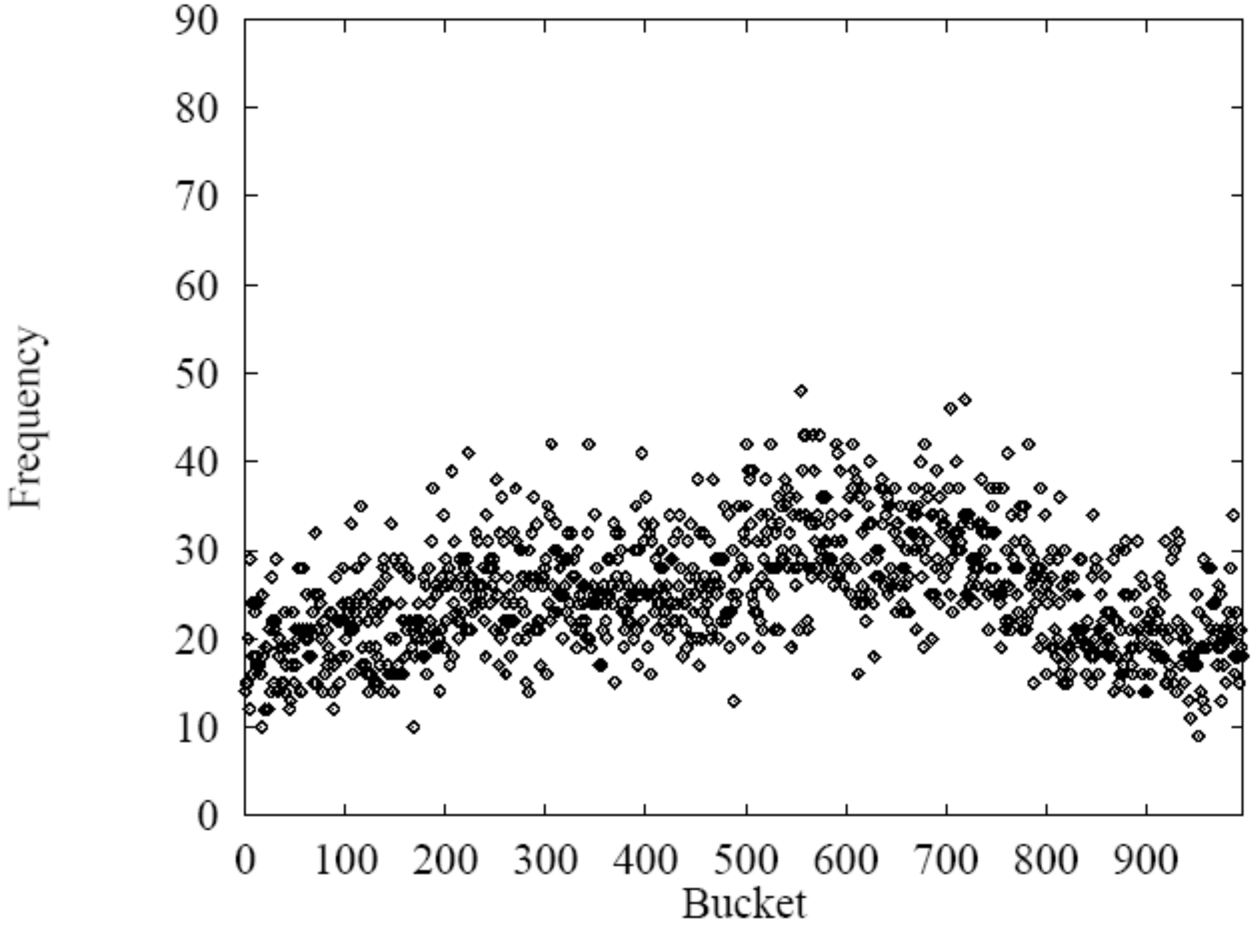
- String hash functions commonly use weighted sums
  - Character values weighted by position in string
    - Long words get bigger codes
    - Distributes keys better than non-weighted sum
  - Let’s look at different weights…

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

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

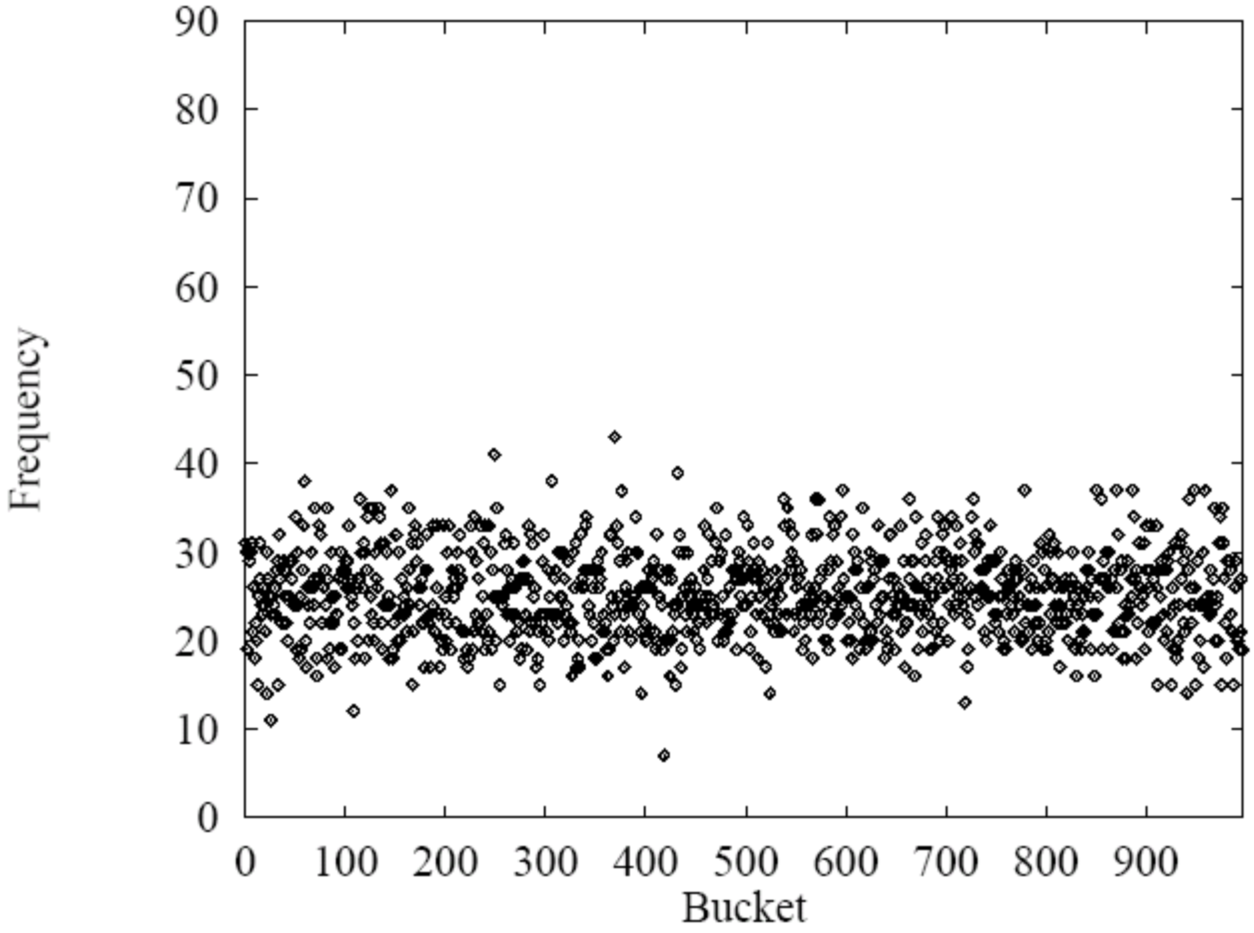


$$\sum_{i=0}^{n} s.charAt(i) * 2^i$$



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

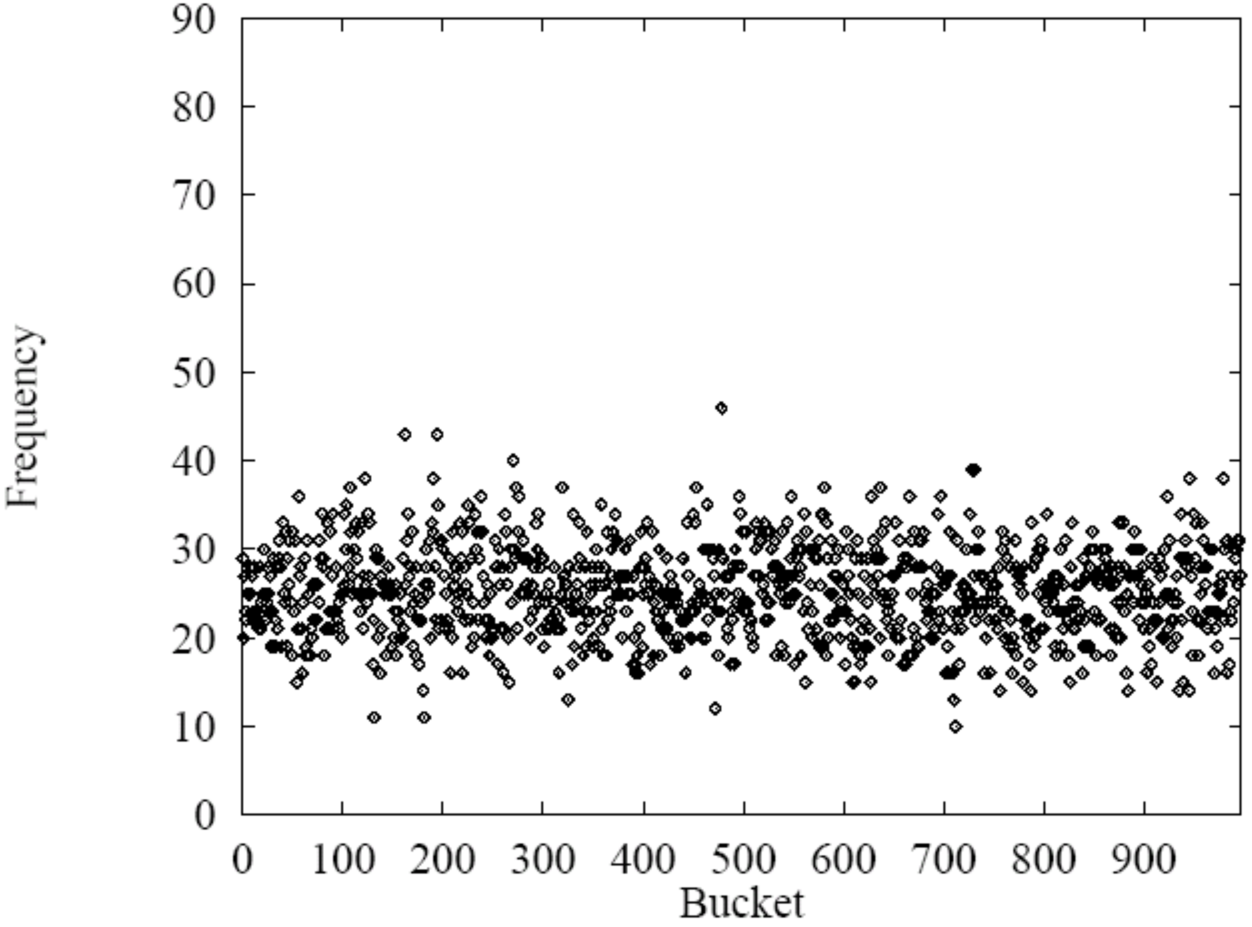
This looks pretty good, but 256^i is big…



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

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) PLUS cost of .equals() method
- 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)