CSCI 136
Data Structures &
Advanced Programming

Spring 2018

Lecture 34

Profs 2070567 and 74655

Administrative Details

Reminders

- No lab this week
  - Many TAs will be holding normal hours to answer questions about labs and practice exams
- Final exam
  - Monday, May 21 at 9:30am in Chemistry 123
  - Covers everything, with strong emphasis on post-midterm
  - Study guide, sample exam will be posted on handouts page
- Review session
  - Friday May 18
    - Time?

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

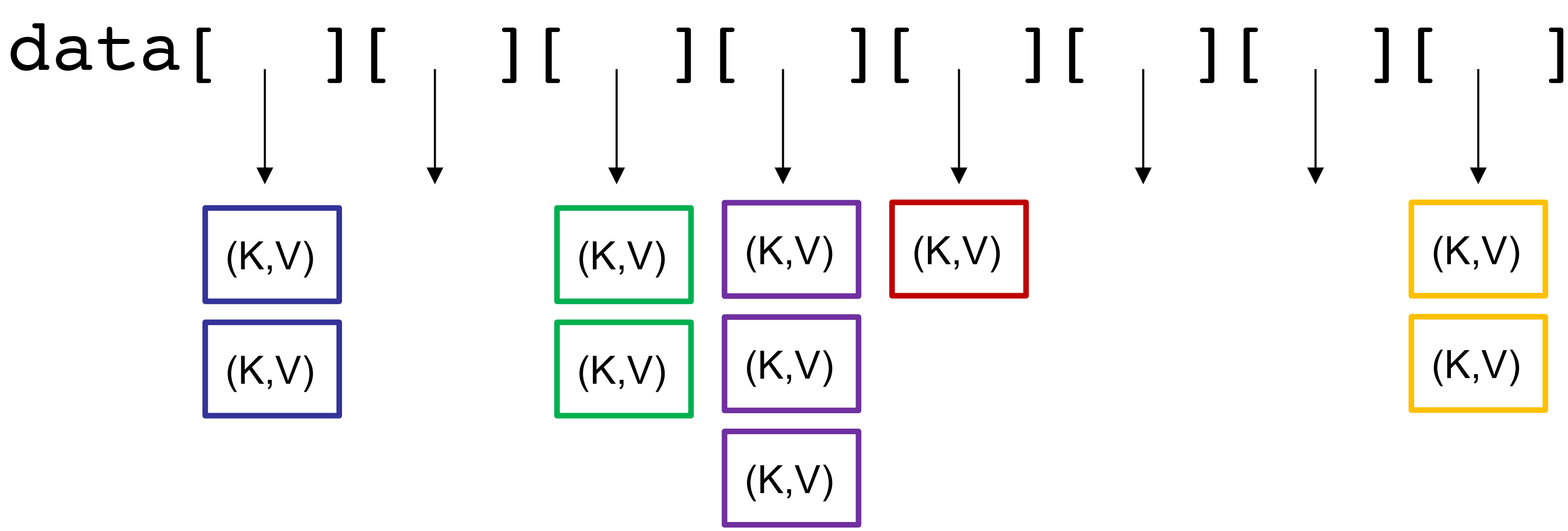
- External Chaining to resolve collisions
- Fun hashing applications (not on exam)
  - Cuckoo hashing
  - Bloom Filters
  - Verification/integrity
  - Deduplication

Linear Proving 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
  - When we remove, must add a placeholder so we don't artificially break up runs

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
- We almost always have to test "goodness" empirically

Example Hash Functions

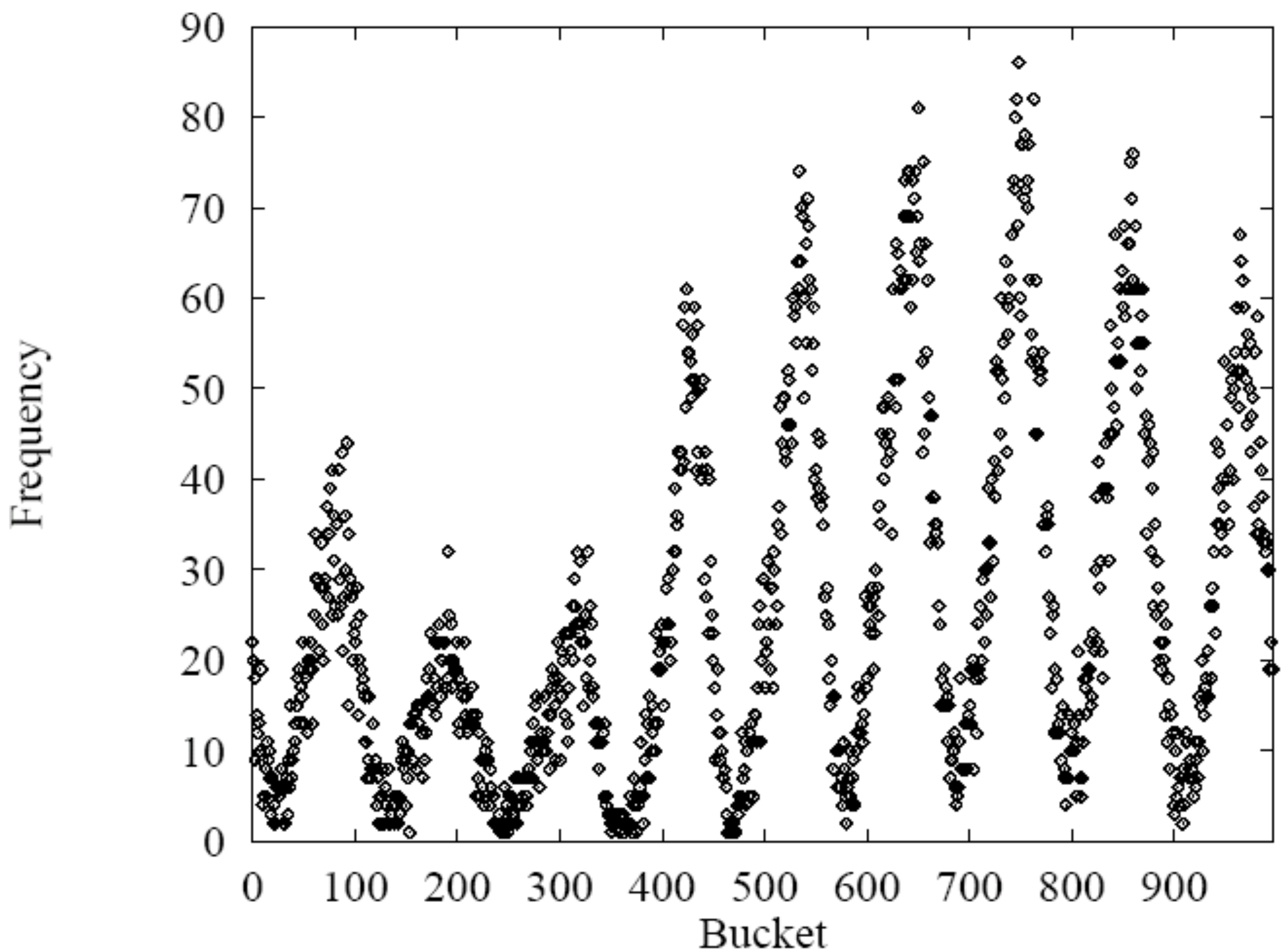
- What are some feasible hash functions for Strings?
  - Use the first char's ASCII value?
    - 0-255 only
    - Not uniform (some letters more popular than others)
  - Sum of all characters' ASCII values?
    - Not uniform - lots of small words
    - smile, limes, miles, slime are all the same

Example Hash Functions

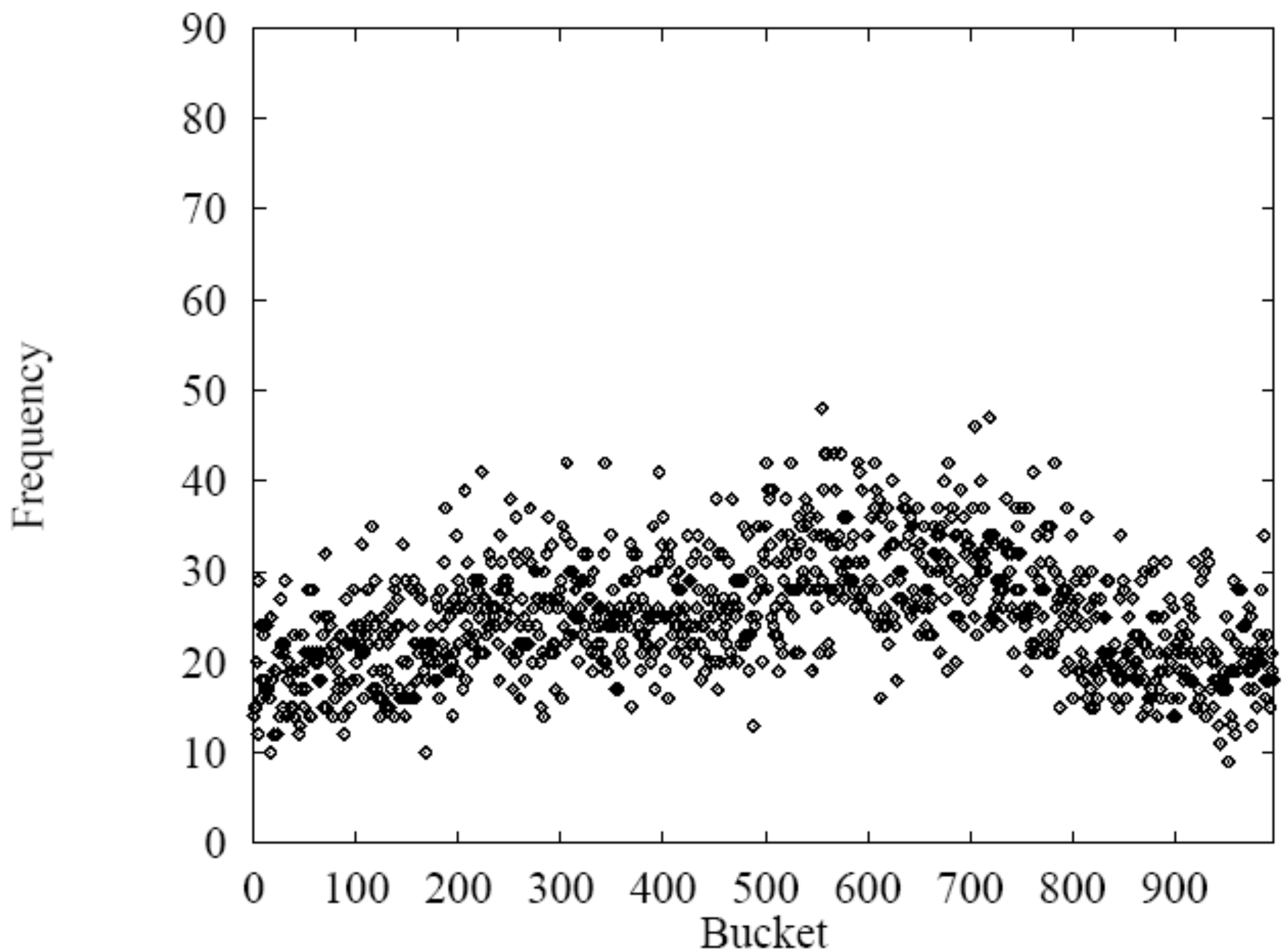
- String hash functions commonly use weighted sums
  - Character values weighted by position in string
    - Long words get bigger codes
    - Distributes keys better than non-weighted sum
  - Let's look at different weights…

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

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

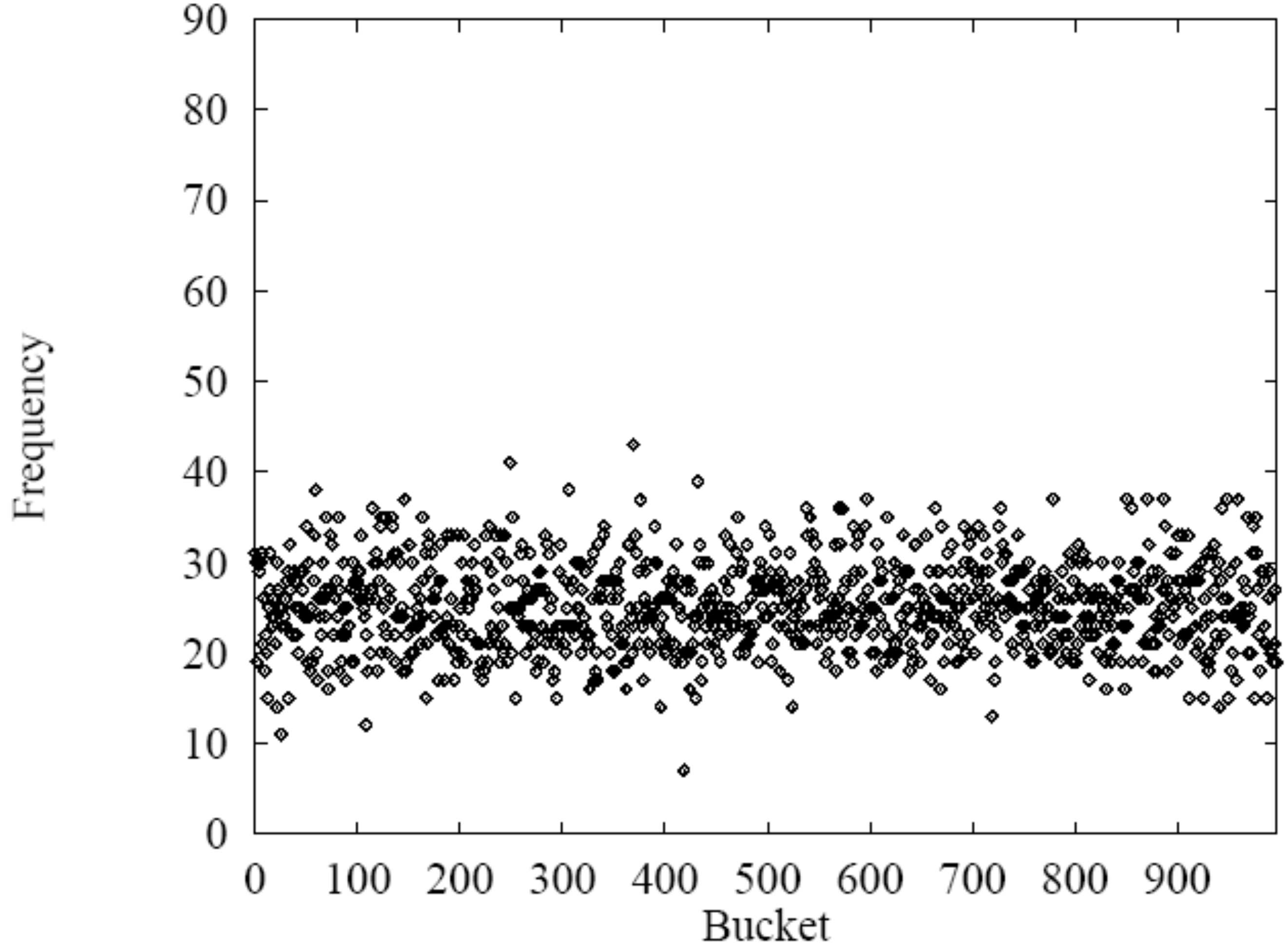


$$\sum_{i=0}^{n} s.charAt(i) * 2^i$$



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

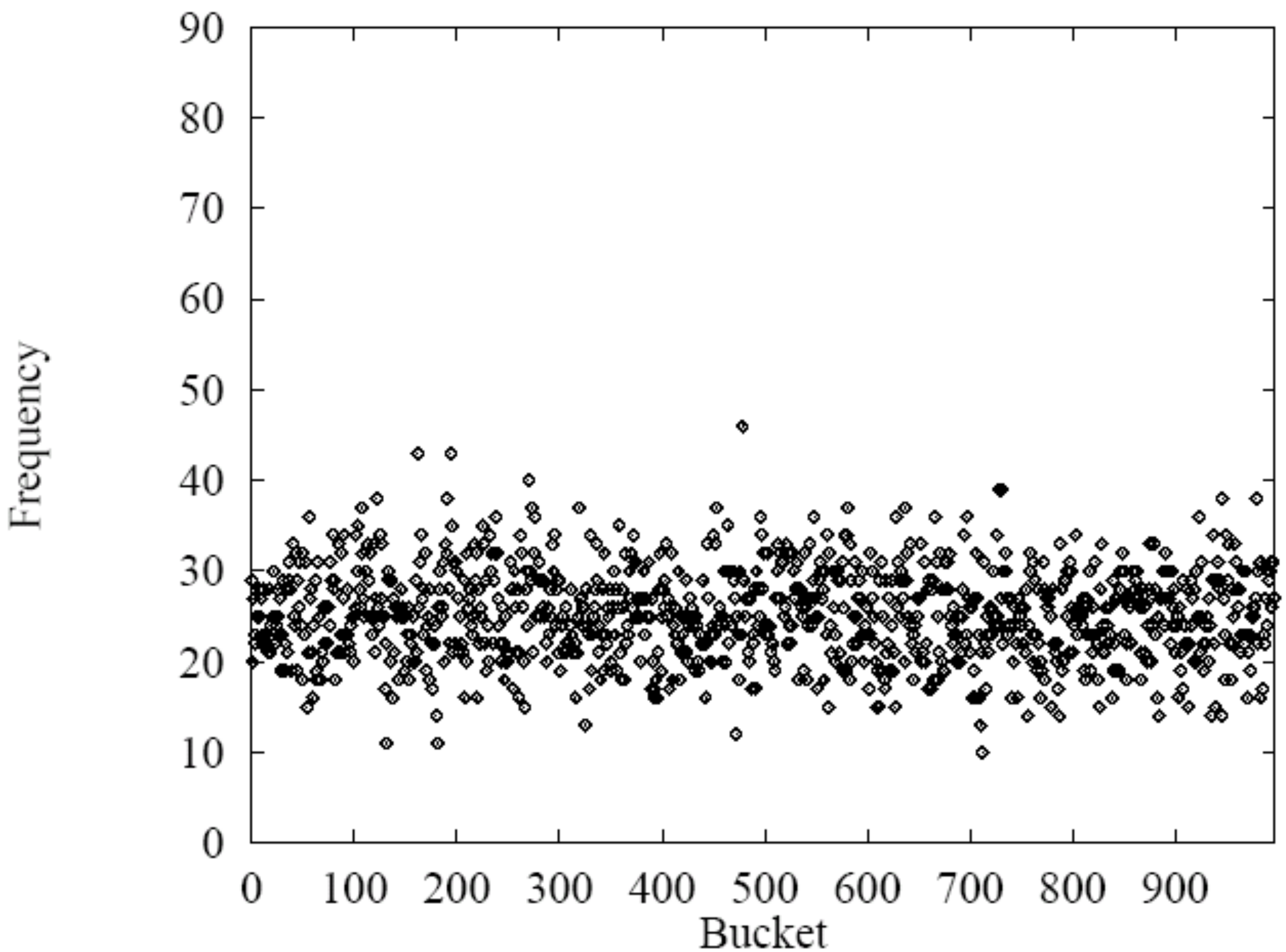
This looks pretty good, but 256^i is big…



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

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)