CSCI 136
Data Structures & Advanced Programming

Bill Jannen

Lecture 34

May 8, 2017

Administrative Details

- Lab 10 due today
  - Questions?
- Final exam - self scheduled
  - You get 2.5 hours to complete it
  - 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
- LinkedLists
- Stacks
- Queues
- Iterators
- Bitwise operations
- Comparables/Comparators
- OrderedStructures
- Binary Trees
- Priority Queues
- Heaps
- Binary Search Trees
- Graphs
- Maps/Hashtables

Last Time

- Discussed graphs, traversal algorithms (Ch 16)
  - Depth first search (stack)
  - Breadth first search (queue)
- Other important graphs concepts/algorithms
  - Cycle detection
  - Shortest path (Dijkstra's algorithm)
  - Minimum Spanning Trees
- Any questions?

Today's Outline

- Maps (#2 Interface of all time)
  - Naïve implementation from Lab 2
  - Hash tables (finally)
    - Hash functions
    - "Load factor"
    - Collisions and how to handle them
  - You should also read Ch 15 for more info

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 set of values
  - boolean equals() - used to compare two maps
  - int hashCode() - returns hash code associated with 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 dictionary needs?

`successor(key)`, `predecessor(key)`

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)!

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 *roughly* O(1) time!
    - » (ask me at the end of class the other reason this is a lie)

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
- 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

Implementing a HashTable

- How can we represent bins?
- Slots in array (or Vector, but arrays are faster)
  - Initial size of array is a fixed-length prime number
- How do we find a 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

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()

Implementing HashTable

- How do we add Associations to the array?
  - `array[o.hashCode() % array.length] = o`?
- Collisions make life hard
- Two approaches
  - Open addressing (using *linear probing*)
  - External chaining

Linear Probing

- If a collision occurs at a given bin, just scan forward (linearly) until an empty slot is available
  - Specify trivial hash function: index of first letter of word
  - Initial array size = 8
  - Add "air hockey" to hash table
  - Add "doubles ping pong"
  - Add "quidditch"
- Let's implement put(key, val) and get(key)…
- What happens when we remove "air hockey", and then lookup "quidditch"?
  - Need a "placeholder" for removed values…

External Chaining

- Downsides of linear probing
  - What if array is almost full?
  - Linear probing is extremely difficult 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

Probing vs. Chaining

- put
  - O(1 + cluster length)
  - O(1 + chain length)
- get
  - O(1 + cluster length)
  - O(1 + chain length)
- remove
  - O(1 + cluster length)
  - O(1 + chain length)
- 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, double size of array (typically threshold = 0.6)
  - Challenges?

Doubling Array

- Cannot just copy values
  - Why?
  - Hash values may change (i.e., element "list")
  - Example
    - key.hashCode() % 8 = 3;
    - key.hashCode() % 16 = 11;
- Recompute all hash codes, reinsert into new array

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

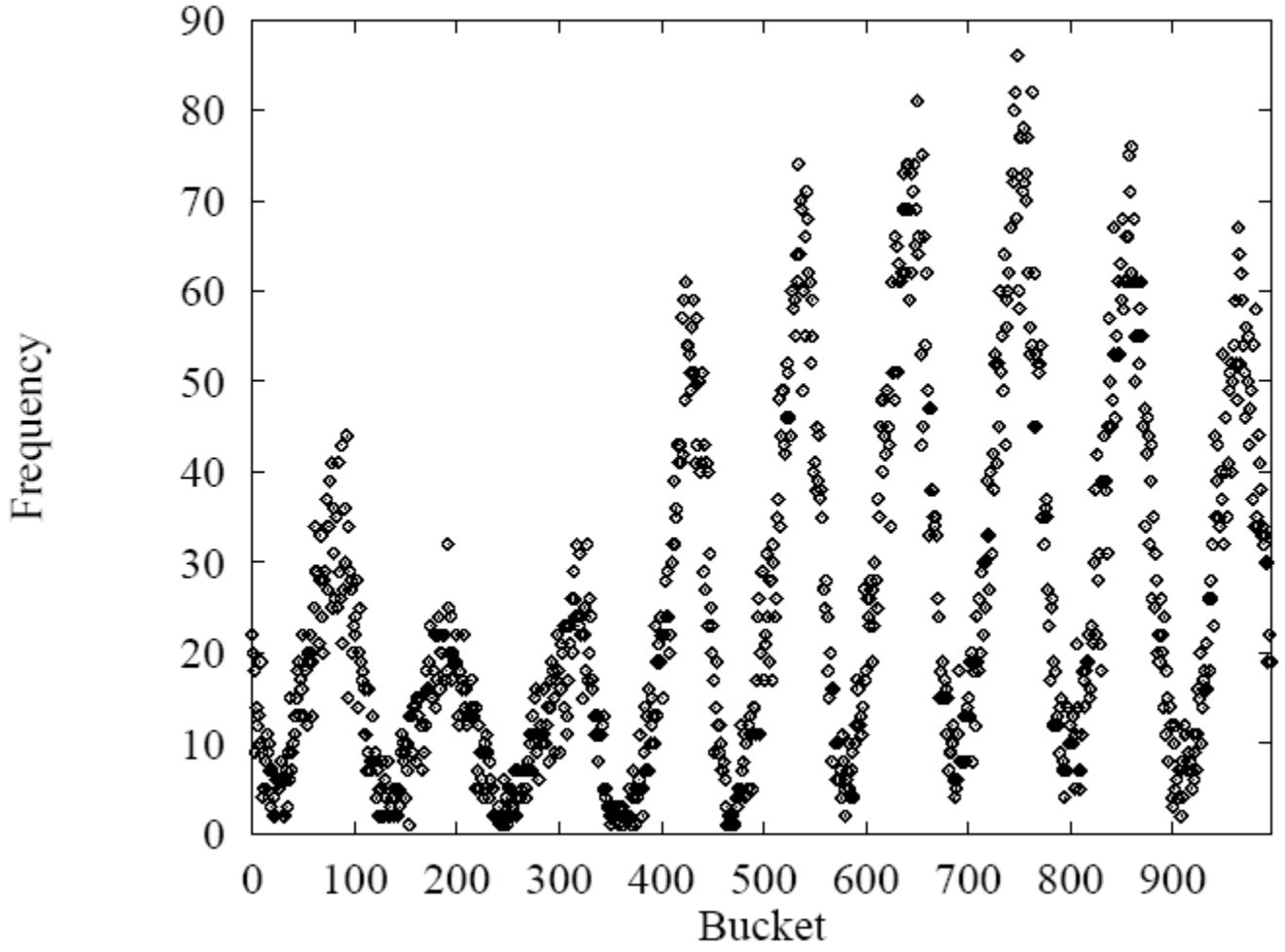
- What are some feasible hash functions for Strings?
  - First char ASCII value mapping
    - 0-255 only
    - Not uniform (some letters more popular than others)
  - Sum of ASCII characters
    - Not uniform - lots of small words
    - smile, limes, miles, slime are all the same

Example Hash Functions

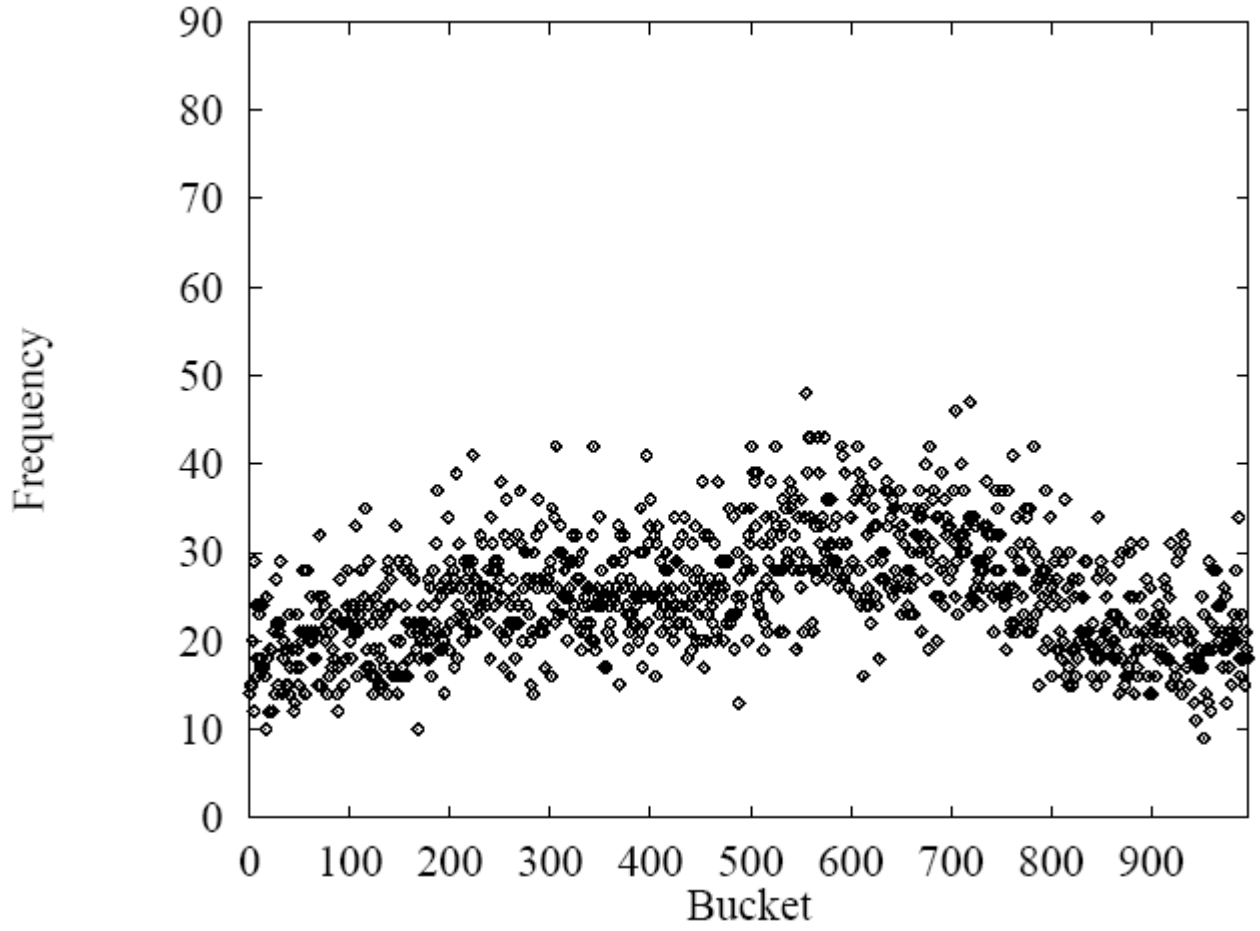
- String hash functions
  - Weighted sum
    - Small words get bigger codes
    - Distributes keys better than non-weighted sum
  - Let's look at different weights…

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

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

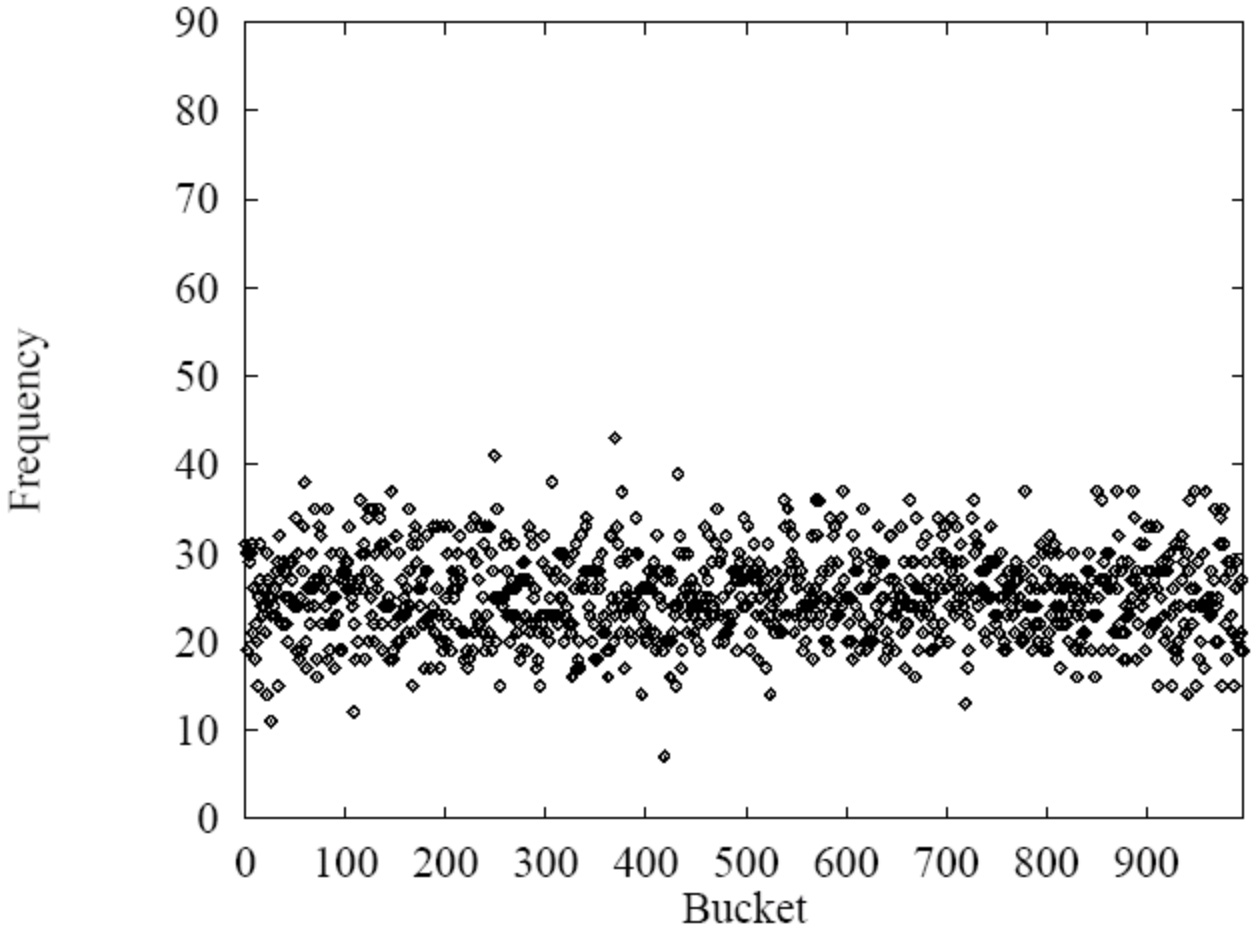


$$\sum_{i=0}^{n} s.charAt(i) * 2^i$$



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

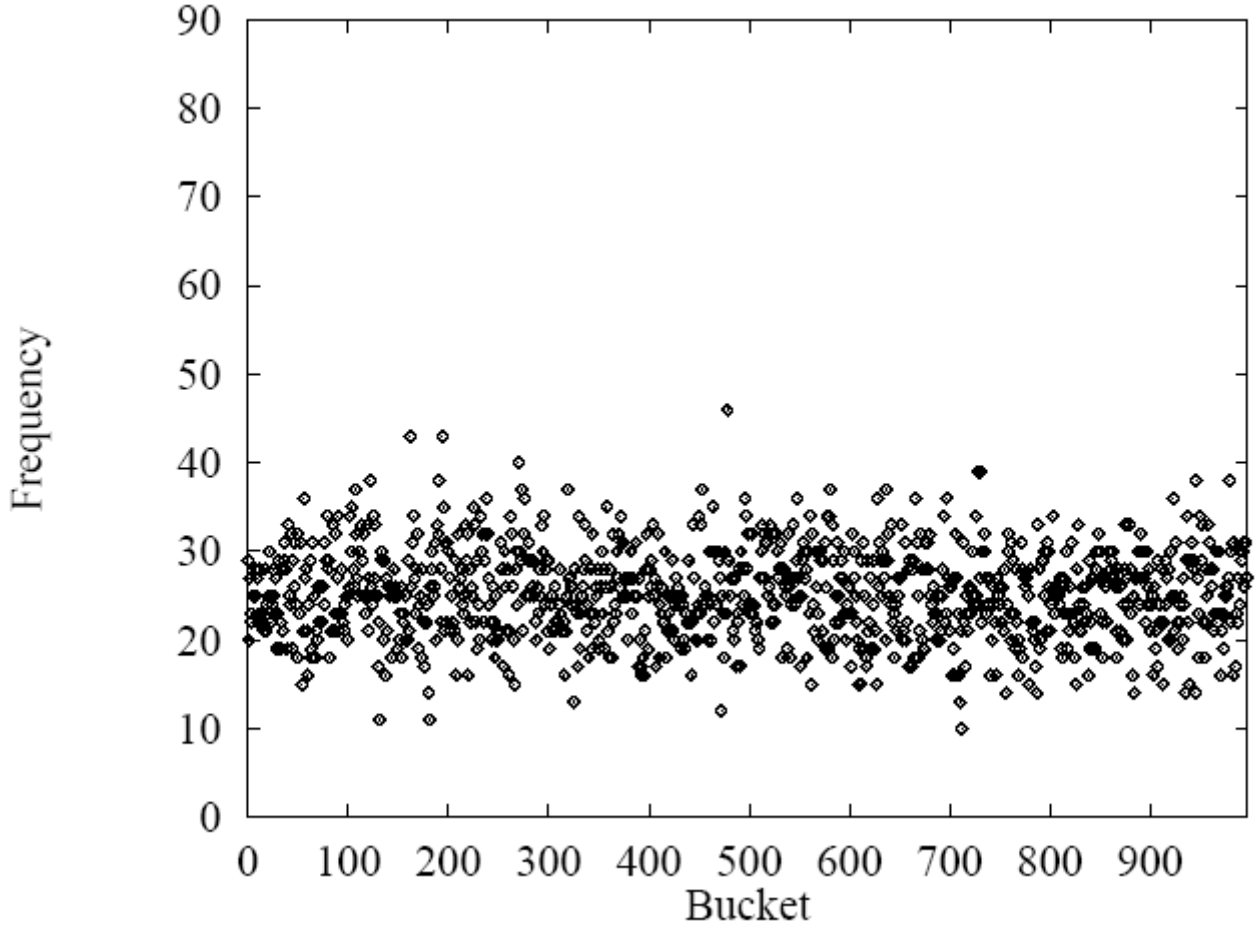
This looks pretty good, but 256^i is big…



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

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)