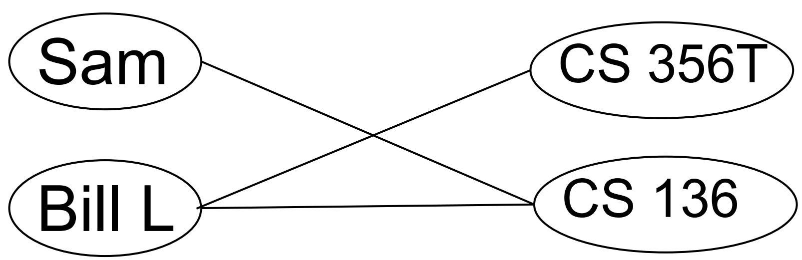
CSCI 136
Data Structures &
Advanced Programming

Lecture 35

Fall 2019



Announcements

- Final Class 😭
- Help Opportunities😀
  - Office Hours Next Week
    - Bill: Wed. (12/11) and Fri. (12/13): 1:00-3:00 pm
  - Review Session: Thursday, Dec. 12: 4:00-5:30 pm
- Final Exam is Monday, Dec. 16😬
  - 9:30-noon in TCL 123 (Wege)
  - Cumulative, but focused on second half of course
  - Sample exam and 2-page study sheet are on-line

Last Time

- Maps & Hashing

Today

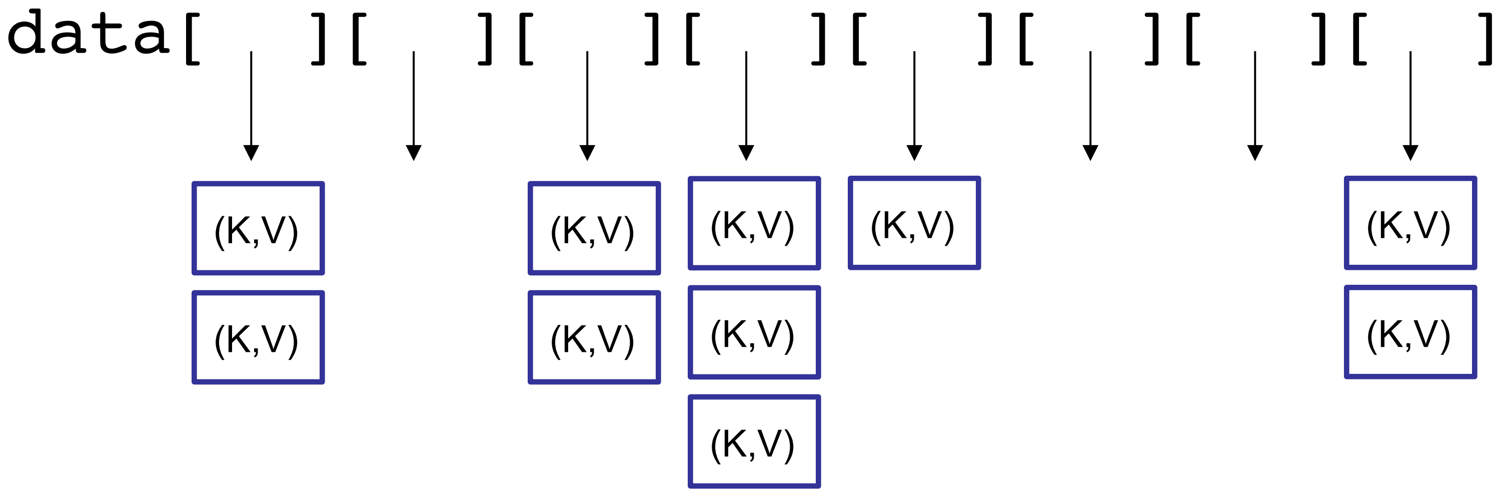
- Hashing Wrap-up
- Course Wrap-up
- Blue Sheets

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
      - Hashtable.java enforces a load factor of 0.6
- 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!

Performance : Probing vs. Chaining

- `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)

- Cluster/chain length depend on choice of hashCode method

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

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()

Example : String Hash Functions

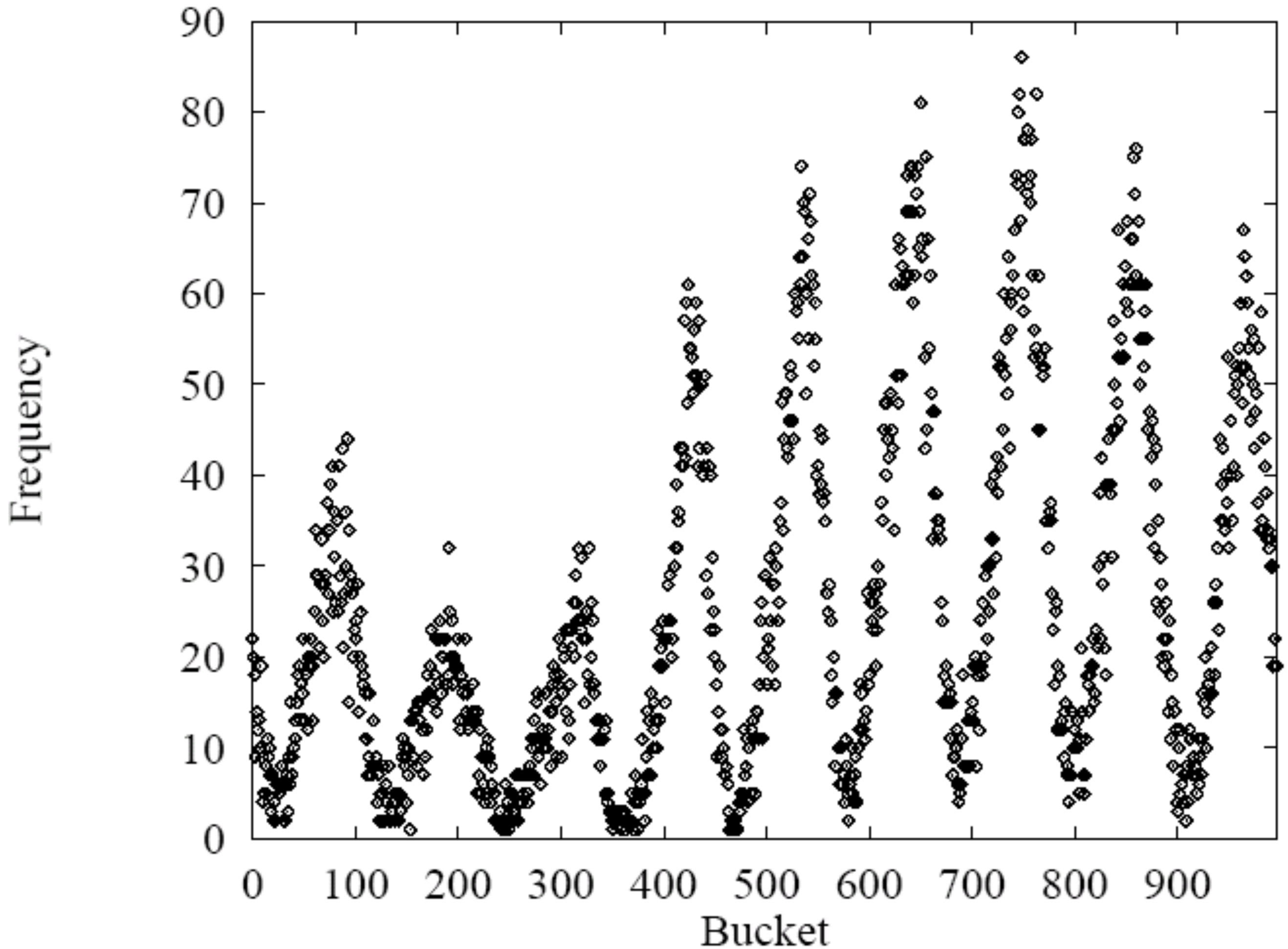
- What are some feasible hash functions for Strings?
  - First char ASCII value mapping
    - 0-255 only
    - Not uniform (some letters more popular than others)
  - Sum of ASCII characters
    - Not uniform - lots of small words
    - smile, limes, miles, slime are all the same

Example Hash Functions

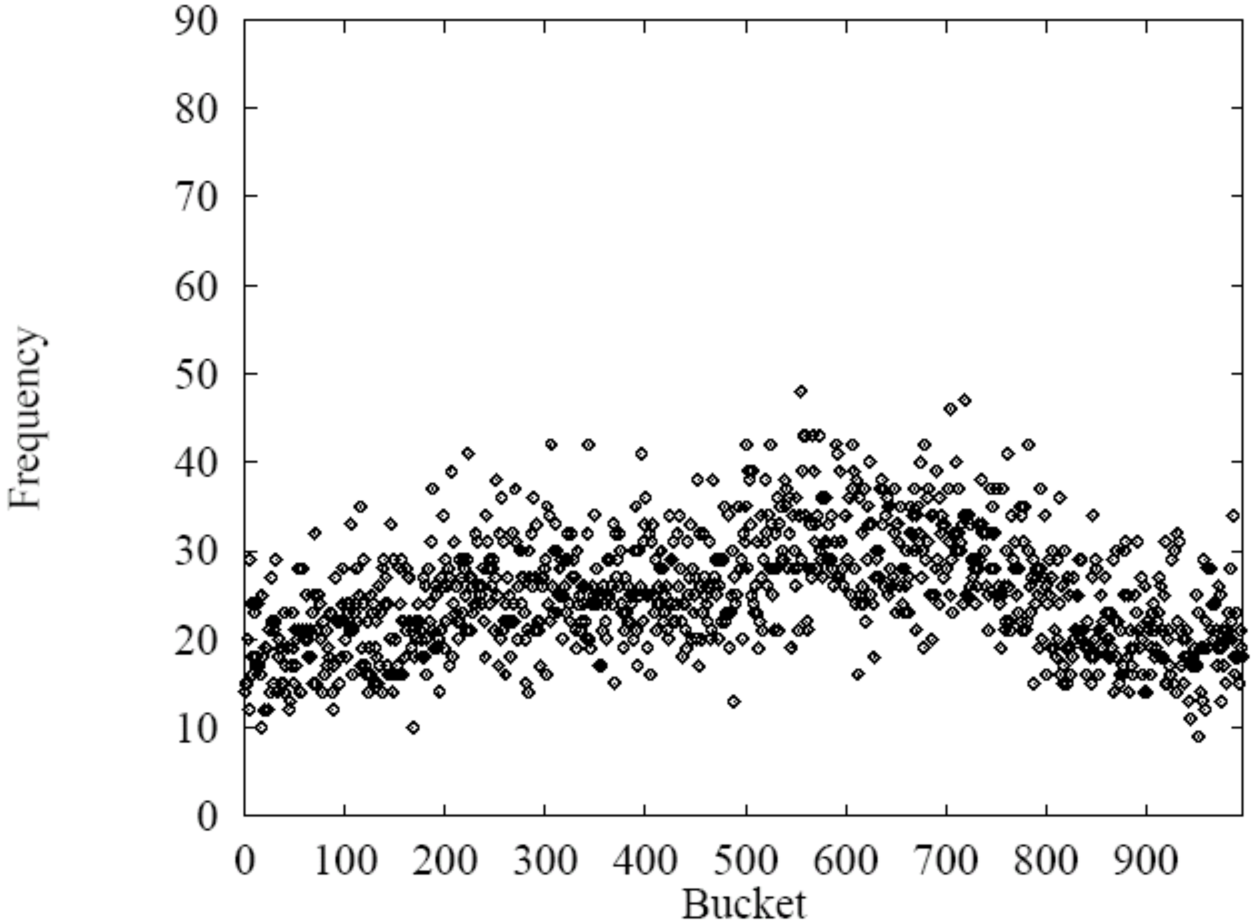
- String hash functions
  - Weighted sum
    - Small words get bigger codes
    - Distributes keys better than non-weighted sum
  - Let's look at different weights…

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

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

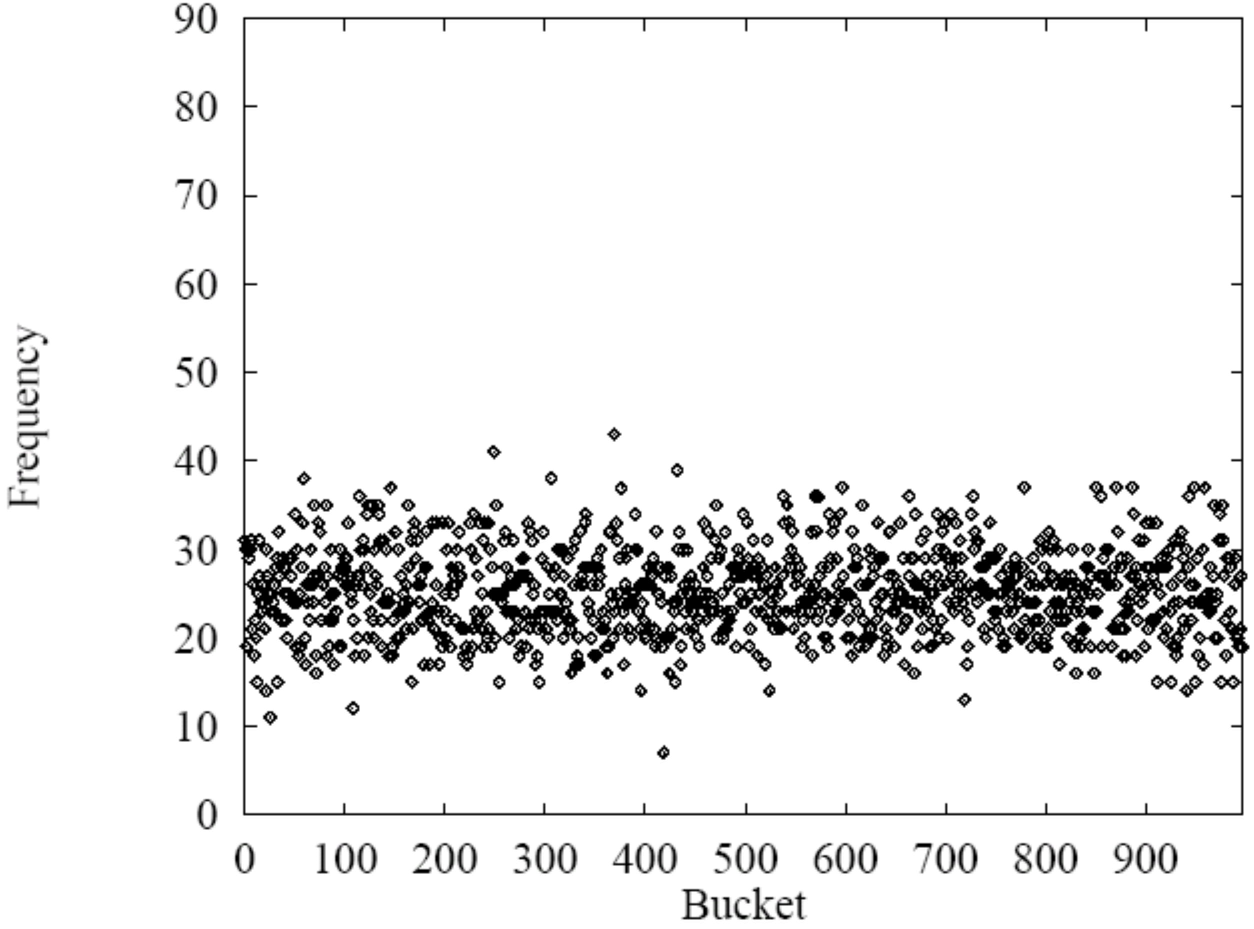


$$\sum_{i=0}^{n} s.charAt(i) * 2^i$$



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

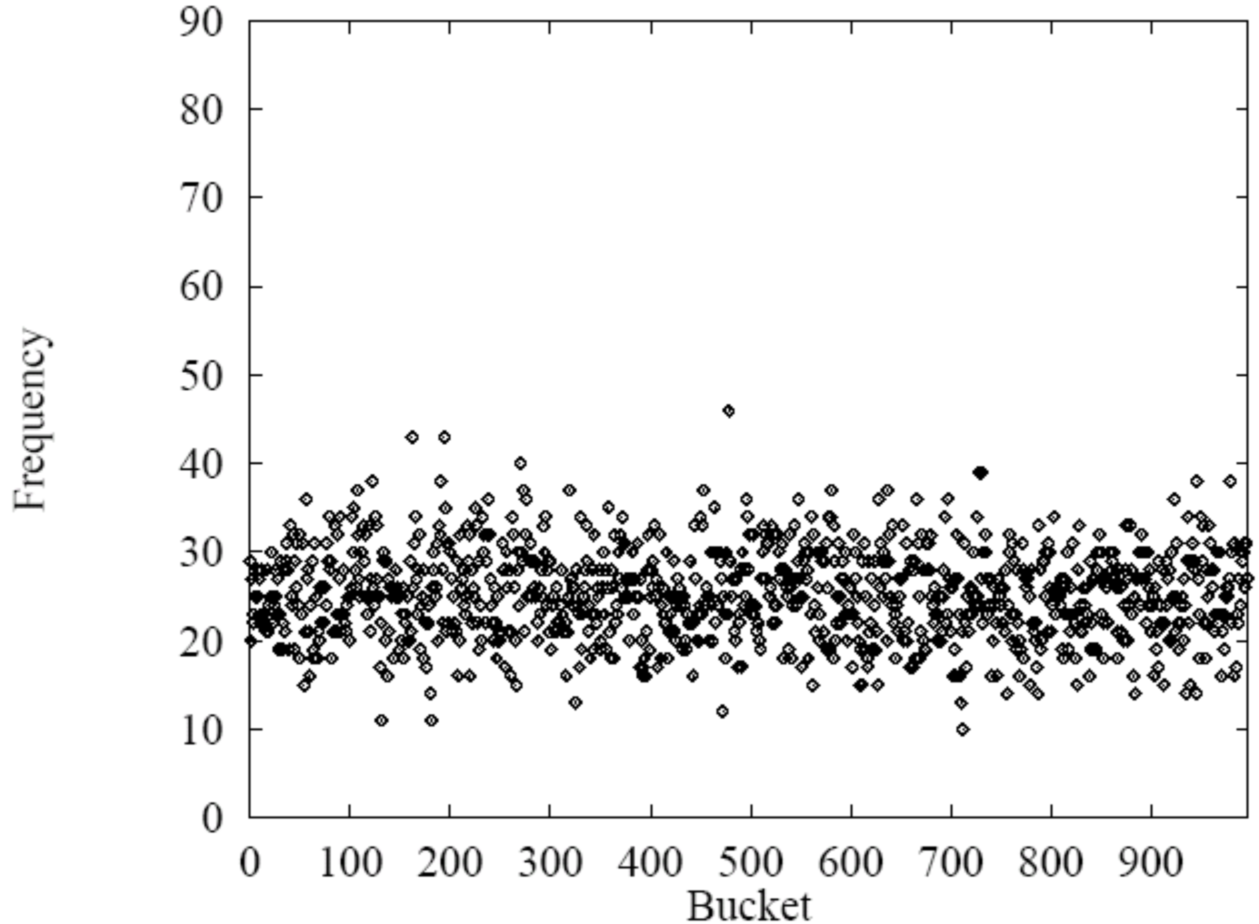
This looks pretty good, but 256^i is big…



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

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) PLUS cost of .equals() method
- Conclusion: for a good hash function (fast, uniformly distributed) and small load factor, we say operations take O(1) time
  - But that's not strictly true….

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)

*PolitiFact Rating: not quite Pants on Fire

What Can We Say For Sure?!

For external chaining

- Assuming the hashing function is equally likely to hash to any slot

Theorem: A search will take O(1 + m/n) time, on average

- n is table size, m is number of keys stored
- True for both successful and unsuccessful searches
  - Based on expected chain length

What Can We Say For Sure?!

For open addressing

- Assuming that all probe sequences are equally likely [which is unlikely!]
- Assuming load factor $\alpha < 1$

Theorem: An unsuccessful search will perform, on average, $O(1 + \alpha)$ probes

Theorem: A successful search will perform, on average, $O(\frac{1}{\alpha} \log \frac{1}{1-\alpha})$ probes

More probe sequences ⇒ better average case

Perfect Hashing

In certain cases, it is possible to design a hashing scheme such that

- Computing the hash takes O(1) time
- There are no collisions
  - Different keys always have different hash values

This is called a *perfect hashing scheme*

Perfect Hashing

If keyspace is smaller than array size

- Handcraft the hashing function
  - Ex: Reserved words in programming languages
- Make array really big
  - Ex: All ASCII strings of length at most 4
    - Hash is 32 bit number
    - Array of size 4.3 billion will suffice
    - Example: IP (v4) addresses

Wrapping Up

Why Data Structures?

Dictionary Structures	put	get	space
unsorted vector	O(n)	O(n)	O(n)
unsorted list	O(n)	O(n)	O(n)
sorted vector (comparable)	O(n)	O(log n)	O(n)
balanced BST (comparable)	O(log n)	O(log n)	O(n)
hash table	O(1)*	O(1)*	O(key range)

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

Data Structure Selection

- Choice of most appropriate structure depends on a number of factors
  - How much data?
    - Static (array) vs dynamic structure (vector/list)
  - Which operations will be performed most often?
    - Lots of searching?
      - Ordered structure vs Map: Which operations matter?
    - Mostly traversing where order doesn't matter: List
  - Is worst case performance crucial? Average case?
    - AVL tree vs SplayTree

Why Complexity Analysis?

- Provides *performance* guarantees
  - Captures effects of scaling on time and space requirements
- Independent of hardware or language
- Can guide appropriate data structure selection

Why Correctness Analysis?

- Provides *behavior* guarantees
- Independent of hardware or language
- Reduce wasted effort developing code
- A powerful debugging tool
  - Program incorrect: Try to prove it *is* correct and see where you get stuck
  - Frequently, such proofs are *inductive*

Why Java?

What makes it worth having to type (or read!)

```
Map<Airport,ComparableAssociation<Integer,
      Edge<Airport,Route>>> result = new
      Table<Airport,ComparableAssociation<Integer,
      Edge<Airport,Route>>>();
```

Why Java?

- Java provides many features to support
  - Data abstraction : Interfaces
  - Information hiding : public/protected/private
  - Modular design : classes
  - Code reuse : class extension; abstract classes
  - Type safety : types are known at compile-time
- As well as
  - Parallelism, security, platform independence, creation of large software systems, embeddability in browsers, ...

Why structure(5)?

- Provides a well-designed library of the most widely-used fundamental data structures
  - Focus on core aspects of implementation
    - Avoids interesting but distracting “fine-tuning” code for optimization, backwards compatibility, etc
  - Allows for easy transition to Java’s own Collection classes
  - Full access to the source code
    - Don’t like Duane’s HashMap---change it!

Why So Many Labs?

Because it's fun and you got a chance to

- Implement a (simple) game - Coinstrip
- Learn about textual analysis - WordGen
- Grapple with large search problems
  - Recursion, Two Towers, Exam Scheduling
- Do some data mining - Sorting
- Write (part of) a PL interpreter – PostScript
- Implement Data Structures
  - Linked Lists and Lexicon
- Model and Simulate a Business Process

Want to Learn More?

- CS 237: Computer Organization
  - Learn about the many levels of abstraction from high-level language → assembly language → machine language → processor hardware
- CS 256: Algorithm Design and Analysis
  - We've only scratched the surface of what elegant algorithm and data structure design can accomplish. For a deeper dive, go here.
- A number of CS electives require one of these two courses

Want to Learn More?

- CS 334: Principles of Programming Languages
  - There are many different types of programming languages: imperative, object-oriented, functional, list-based, logic, ... Why!? What is required to support languages of these kinds?
- CS Colloquium
  - Weekly (Fridays at 2:30pm) presentations from active researchers in CS from across the country
- Talk to Faculty and CS Majors
  - They do interesting things!