Searching, Maps, Tries (hashing)

- Searching is a fundamentally important operation
  - We want to search quickly, very very quickly
  - Consider searching using Google, ACES, issues?
  - In general we want to search in a collection for a key

- We've searched using trees and arrays
  - Tree implementation was quick: O(log n) worst/average?
  - Arrays: access is O(1), search is slower

- If we compare keys, log n is best for searching n elements
  - Lower bound is $\Omega$(log n), provable
  - Hashing is O(1) on average, not a contradiction, why?
  - Tries are O(1) worst-case!! (ignoring length of key)

From Google to Maps

- If we wanted to write a search engine we'd need to access lots of pages and keep lots of data
  - Given a word, on what pages does it appear?
  - This is a map of words->web pages

- In general a map associates a key with a value
  - Look up the key in the map, get the value
  - Google: key is word/words, value is list of web pages
  - Anagram: key is string, value is words that are anagrams

- Interface issues
  - Lookup a key, return boolean: in map or value: associated with the key (what if key not in map?)
  - Insert a key/value pair into the map

Interface at work: MapDemo.java

- Key is a string. Value is # occurrences
  - Code below shows how Map interface/classes work

```
while (it.hasNext()) {
  String s = it.next();
  Counter c = map.get(s);
  if (c != null) c.increment();
  else map.put(s, new Counter());
}
```

- What clues are there for prototype of map.get and map.put?
  - What if a key is not in map, what value returned?
  - What kind of objects can be put in a map?

Replacing Counter with Integer

- With autoboxing (and unboxing) do we need class Counter?
  - What if we access a key that's not there?

```
while (it.hasNext()) {
  String s = it.next();
  if (map.containsKey(s)) {
    map.put(s, map.get(s)+1);
  } else map.put(s,1);
}
```

- What is key? What is value?
  - What if a key is not in map, what value returned?
  - Is use of get() to determine if key is present a good idea?
Getting keys and values from a map

- Access every key in the map, then get the corresponding value
  - Get an iterator of the set of keys: `keySet().iterator()`
  - For each key returned by this iterator call `map.get(key)`

- Get an iterator over (key, value) pairs, there's a nested class called `Map.Entry` that the iterator returns, accessing the key and the value separately is then possible
  - To see all the pairs use `entrySet().iterator()`

External Iterator without generics

- The `Iterator` interface access elements
  - Source of iterator makes a difference; cast required?

```java
Iterator it = map.keySet().iterator();
while (it.hasNext()){
    Object value = map.get(it.next());
}
Iterator it2 = map.entrySet().iterator();
while (it2.hasNext()){
    Map.Entry me = (Map.Entry) it.next();
    Object value = me.getValue();
}
```

External Iterator with generics

- Avoid `Object`, we know what we have a map of
  - Is the syntax worth it?

```java
Iterator<String> it = map.keySet().iterator();
while (it.hasNext()){
    Counter value = map.get(it.next());
}
Iterator<Map.Entry<String,Counter>> it2 = map.entrySet().iterator();
while (it2.hasNext()){
    Map.Entry<String,Counter> me = it2.next();
    Counter value = me.getValue();
}
```

Margo Seltzer

Herchel Smith Professor of Computer Science and Associate Dean at Harvard, CTO of Sleepycat Software

“Computer Science is the ultimate combination of Engineering and Science. It provides the tangible satisfaction that comes from building things and seeing them run with the thrill of discovery that is the hallmark of scientific exploration... I want to... work with people, and help young people become skilled computer scientists and engineers.”

“I have always maintained an active life outside work, playing soccer, studying karate, spending time with my family, and socializing as much as my schedule allows. When you have a job, you have plenty of time for these activities, but when you have a career, you have to make the time, and it’s an effort well worth making.”
Hashing: Log \((10^{100})\) is a big number

- Comparison based searches are too slow for lots of data
  - How many comparisons needed for a billion elements?
  - What if one billion web-pages indexed?

- Hashing is a search method: average case \(O(1)\) search
  - Worst case is very bad, but in practice hashing is good
  - Associate a number with every key, use the number to store the key
    - Like catalog in library, given book title, find the book
  - A hash function generates the number from the key
    - Goal: Efficient to calculate
    - Goal: Distributes keys evenly in hash table

Hashing details

- There will be collisions, two keys will hash to the same value
  - We must handle collisions, still have efficient search
  - What about birthday “paradox”: using birthday as hash function, will there be collisions in a room of 25 people?

- Several ways to handle collisions, in general array/vector used
  - Linear probing, look in next spot if not found
    - Hash to index \(h\), try \(h+1, h+2, \ldots\), wrap at end
    - Clustering problems, deletion problems, growing problems
  - Quadratic probing
    - Hash to index \(h\), try \(h+1^2, h+2^2, h+3^2, \ldots\), wrap at end
    - Fewer clustering problems
  - Double hashing
    - Hash to index \(h\), with another hash function to \(j\)
    - Try \(h, h+j, h+2j, \ldots\)

Chaining with hashing

- With \(n\) buckets each bucket stores linked list
  - Compute hash value \(h\), look up key in linked list table[\(h\)]
  - Hopefully linked lists are short, searching is fast
  - Unsuccessful searches often faster than successful
    - Empty linked lists searched more quickly than non-empty
    - Potential problems?

- Hash table details
  - Size of hash table should be a prime number
  - Keep load factor small: number of keys/size of table
  - On average, with reasonable load factor, search is \(O(1)\)
  - What if load factor gets too high? Rehash or other method

Hashing problems

- Linear probing, hash(\(x\)) = \(x\), (mod tablesiz)
  - Insert 24, 12, 45, 14, delete 24, insert 23 (where?)

- Same numbers, use quadratic probing (clustering better?)

- What about chaining, what happens?
What about hash functions

- Hashing often done on strings, consider two alternatives

```java
public static int hash(String s)
{
    int k, total = 0;
    for(k=0; k < s.length(); k++)
    {
        total += s.charAt(k);
    }
    return total;
}
```

- Consider `total += (k+1)*s.charAt(k)`, why might this be better?
  - Other functions used, always mod result by table size

- What about hashing other objects?
  - Need conversion of key to index, not always simple
  - Every object has method `hashCode()`!

Trie: efficient search words/suffixes

- A trie (from retrieval, but pronounced “try”) supports
  - Insertion: put string into trie (delete and look up)
  - These operations are $O(\text{size of string})$ regardless of how many strings are stored in the trie! Guaranteed!

- In some ways a trie is like a 128 (or 26 or alphabet-size) tree, one branch/edge for each character/letter
  - Node stores branches to other nodes
  - Node stores whether it ends the string from root to it

- Extremely useful in DNA/string processing
  - Very useful for matching suffixes: suffix tree

Trie picture/code (see TrieSet.java)

- To add string
  - Start at root, for each char create node as needed, go down tree, mark last node

- To find string
  - Start at root, follow links
    - If null, not found
    - Check word flag at end

- To print all nodes
  - Visit every node, build string as nodes traversed

- What about union and intersection, iteration?
  - Indicates word ends here

Guy L. Steele, Jr.

Co-invented/developed Scheme, continues to develop Java

If, several years ago, with C++ at its most popular, ... you had come to me, O worthy opponents, and proclaimed that objects had failed, I might well have agreed. But now that Java has become mainstream, popularizing not only object-oriented programming but related technologies such as garbage collection and remote method invocation, ... we may now confidently assert that objects most certainly have not failed.