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
  - Interface in code below shows how Map class works

```java
while (scanner.hasNext()) {
    String s = (String) scanner.next();
    Counter c = (Counter) map.get(s);
    if (c != null) c.increment();
    else m.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?

Accessing values in a map (e.g., print)

- 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

- The Iterator interface accesses 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();
}
```

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 tablesize)
  - 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
  - Ever object contains 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 and code (see Trie.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?