**Searching, Maps, Hashing**

- **Searching is a very important programming application**
  - Consider google.com and other search engines
  - In general we search a *collection* for a *key*
    - Vector/List, Tree: $O(n)$ and $O(\log n)$
    - If we compare keys when searching we cannot do better than $O(\log n)$ for searching
    - This is a lower bound, commonly expressed as $\Omega(n)$
  - Why isn’t this good enough?
- **What about searching in a thumb-indexed dictionary?**
  - Why is this more than comparing keys?
- **Suppose we use last four-digits of social security number and a vector of 10,000 elements?**

**Log (google) 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 that has 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
    - Efficient to calculate
    - 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 $h+1$, $h+2$, …, wrap at end
    - Clustering problems, deletion problems, growing problems
  - Quadratic probing
    - Hash to index $h$, try $h+1^2$, $h+2^2$, $h+3^2$, …, wrap at end
    - Fewer clustering problems
  - Double hashing
    - Hash to index $h$, with another hash function to $j$
    - Try $h$, $h+j$, $h+2j$, …

**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
  - Same numbers, use quadratic probing (clustering better?)
- What about chaining, what happens?

What about hash functions

- Hashing often done on strings, consider two alternatives
  - unsigned hash(const string& s)
    - unsigned int k, total = 0;
    - for(k=0; k < s.length(); k++)
      - total += s[k];
    - return total;
  - Other functions used, always mod result by table size
- What about hashing other objects?
  - Sometimes address of value used to hash it, not always useful, e.g., for strings

Implementation issues

- We want to consider map, table, dictionary (what's in a name?)
  - STL uses map, Java uses map, we'll use map
  - What operations should a map support: keys and values
  - What is interface to individual key/value pairs, what are alternatives?
  - If we get a value from the map, can we change it?
  - What about key from map, change it?

From sdmap to tmap

- Abstract base class tmap<...> is in tmap.h
  - Supports keys mapped to values, iterators
    - See tmapcounter.cpp, contrast to sdmap version
- Supported by subclasses BSTMap, HMap (and UVMAP)
  - HMap works only for strings mapped to xxx
    - See bstmap.h, hmap.h, and uvmap.h
  - Iterators are complex to implement, not trivial to use
- How could we use a map to implement a multiset?
  - What is a multiset a map from/to?
  - What about a set?