Searching, Maps, Tables

- **Searching is a fundamentally important operation**
  - We want to do these operations quickly
  - Consider searching using google.com, ACES, etc.,
  - In general we want to search in a *collection* for a *key*

- We’ve seen searching in context of the linked lists and vectors

- If we compare keys, we cannot do better than log n to search n elements
  - Lower bound is Ω(log n), provable
  - Hashing is O(1) on average, not a contradiction, why?
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

- 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: tmapcounter.cpp

- **Key is a string, Value is # occurrences (like multiset)**
  - Interface in code below shows how tmap class works

```cpp
while (input >> word) {
    if (map->contains(word)) {
        map->get(word) += 1;
    } else {
        map->insert(word,1);
    }
}
```

- **What clues are there for prototype of map.get and map.contains?**
  - Reference is returned by get, not a copy, why?
  - Parameters to contains, get, insert are same type, what?
Accessing values in a map (e.g., print)

- We can apply a function object to every element in a map, this is called an *internal iterator*
  - Simple to implement (why?), relatively easy to use
    - See Printer class in tmapcounter.cpp
  - Limited: must visit every map element (can’t stop early)

- Alternative: use Iterator subclass (see tmapcounter.cpp), this is called an *external iterator*
  - Iterator has access to “guts” of a map, iterates over it
    - Must be a friend-class to access guts
    - Tightly coupled: container and iterator
  - Standard interface of Init, HasMore, Next, Current
  - Can have several iterators at once, can stop early, can pass iterators around as parameters/objects
Other map examples

- **Anamap.cpp, alternative program for finding anagrams**
  - Maps Anaword: *key to tvector<Anaword>: value*
  - Look up Anaword, associate all equal Anawords with first one stored in map
  - To print, loop over all keys, grab vector, print if ???

- **Parsing arithmetic expressions**
  - Inheritance hierarchy and somewhat complex code
  - Map string/variable name: *key to Expression *: value*
    - Map x \(\rightarrow\) y + 3, what’s value of x when y = 7?
    - What happens if we map x \(\rightarrow\) y and y \(\rightarrow\) x?
From interface to implementation

- **First the name:** STL uses map, Java uses map, we’ll use map
  - Other books/courses use table, dictionary, symbol table
  - We’ve seen part of the map interface in tmapcounter.cpp
    - What other functions might be useful?
    - What’s actually stored internally in a map?

- **The class tmap is a templated, abstract base class**
  - Advantage of templated class (e.g., tvector, tstack, tqueue)
  - Base class permits different implementations
    - UVmap, BSTVap, HMap (stores just string->value)
  - Internally combine key/value into a pair
    - <pair.h> is part of STL, standard template library
    - Struct with two fields: first and second
Using templated classes

- Client code includes (typically) only .h file
  - Where is the .cpp file, why not access via #include?
  - Difference between compilation and linking
  - Is foo.h included in foo.cpp? Why?

- Template .cpp file is NOT code, it’s a code generator/template
  - When template is *instantiated* by client, code is generated
  - To instantiate, need access to template source
  - Templated foo.h typically has #include “foo.cpp”
    - Why is this better in foo.h than in client program?

- If you don’t call a templated function it’s not generated
  - Template instantiation creates code, but not every member function (not created if not called)
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**
  - 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 $h+1$, $h+2$, $...$, wrap at end
    - Clustering problems, deletion problems, growing problems
  - Quadratic probing
    - Has 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 % tablesiz**e
  - Insert 24, 12, 45, 14, delete 24, insert 23

  ![Hashing diagram 1](image1)

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

  ![Hashing diagram 2](image2)

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

- Hashing often done on strings, consider two alternatives

```c
unsigned hash(const string& s) {
    unsigned int k, total = 0;
    for(k=0; k < s.length(); k++) {
        total += s[k];
    }
    return total;
}
```

- What about total += k*s[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
  - HMap (subclass of tmap) maps string->values
  - Why not any key type (only strings)?