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

- **Recall simple implementations of sets (trees and lists)**
  - Tree implementation was quick
  - Vector of linked lists was fast, but how to make it faster?

- **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: `tmapcounter.cpp`

- **Key is a string, Value is # occurrences**
  - 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
Internal iterator and applyOne

- Applicant subclass: applied to key/value pairs stored in a map
  - The applicant has an applyOne function, called from the map/collection, in turn, with each key/value pair
  - The map/collection has an applyAll function to which is passed an instance of a subclass of Applicant

```cpp
class Printer : public Applicant<string, int>
{
    public:
        virtual void applyOne(string& key, int& value) {
            cout << value << "\t" << key << endl;
        }
};
```

- Applicant class is templated on the type of key and value
  - See tmap.h, tmapcounter.cpp, and other examples
Internal iterator applyAll

- Map class applies the applicant to all key/value pairs in map
  - Calls applyOne for every key/value pair

```cpp
template <class Key, class Value>
void BSTMap<Key,Value>::doApply(Applicant<Key,Value> & app,
                               Tree<pair<Key,Value> > * tree)
{
    if (tree != 0)
    {
        doApply(app,tree->left);
        obj.applyOne(tree->info.first,tree->info.second);
        doApply(app,tree->right);
    }
}
```

- Applicant class (and tree) templated on the type of key and value
  - See tmap.h, bstmap.cpp, and other examples
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
External Iterator

- The Iterator base class is templated on `pair<key,value>`, makes for ugly declaration of iterator pointer
  - (note: space between `> ` in code below is required why?)

```cpp
Iterator<pair<string,int>> * it =
    map->makeIterator();
for(it->Init(); it->HasMore(); it->Next()) {
    cout << it->Current().second << "\t";
    cout << it->Current().first << endl;
}
```

- We ask a map/container to provide us with an iterator
  - Don't know how the map is implemented, just want an iterator
  - Map object is an iterator factory: makes/creates iterator
Tapestry tmap v STL map

● See comparable code in `tmapcounterstl.cpp`
  ➢ Instead of `get`, use overloaded `[]` operator
  ➢ Instead of `contains` use `count` --- returns an int

● Instead of Iterator class with `Init`, `HasMore`, ...
  ➢ Use `begin()` and `end()` for starting and ending values
  ➢ Use `++` to increment iterator [compare with `Next()` ]
  ➢ Instead of `Current()`, dereference the iterator

● STL map uses a balanced search tree, guaranteed O(log n)
  ➢ Nonstandard hash_map is tricky to use in general
  ➢ We've seen AVL trees, STL uses red-black (one pass)
Map example: finding anagrams

- mapanagram.cpp, alternative program for finding anagrams
  - Maps string (normalized): key to tvector<string>: value
  - Look up normalized string, associate all "equal" strings with normalized form
  - To print, loop over all keys, grab vector, print if ???

- Each value in the map is list/collection of anagrams
  - How do we look up this value?
  - How do we create initial list to store (first time)
  - We actually store pointer to vector rather than vector
    - Avoid map->get()[k], can't copy vector returned by get
- See also mapanastl.cpp for standard C++ using STL
  - The STL code is very similar to Tapestry (and to Java!)
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 $h+1$, $h+2$, $...$, 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 (where?)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>24</td>
<td>45</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>24</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>45</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

- Hashing often done on strings, consider two alternatives

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

- Consider total += (k+1)*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)?
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! \textit{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.cpp)

- 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/0, not found
  - Check word flag at end

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

- What about union and intersection?

  - 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.