Maps and Hashing

Eric Roberts
CS 106B
February 15, 2013

Simplifying the Map Abstraction

- Although templates offer considerable flexibility when you are designing a collection class, they also complicate both the interface and the implementation, making them harder to follow.
- To make sure that you understand the details of the various strategies for implementing maps, Chapter 15 simplifies the interface so that both the keys and values are always strings. The resulting class is called StringMap.
- Although the book includes a more expansive set of methods, this lecture looks only at put, get, and containsKey.
- Once you understand how to implement the StringMap class using each of the possible representations, you can add the remaining methods and use the C++ template mechanism to generalize the key and value types.

The stringmap.h Interface

```cpp
#ifndef _hashmap_h
#define _hashmap_h
#include <string>
class StringMap {
public:
    StringMap();
    ~StringMap();
    std::string get(const std::string & key) const;
    void put(const std::string & key, const std::string & value);
    bool containsKey(const std::string & key) const;
};
#endif
```

An Illustrative Mapping Application

- Suppose that you want to write a program that displays the name of a state given its two-letter postal abbreviation.
- This program is an ideal application for the StringMap class because what you need is a map between two-letter codes and state names. Each two-letter code uniquely identifies a particular state and therefore serves as a key for the StringMap; the state names are the corresponding values.
- To implement this program in C++, you need to perform the following steps, which are illustrated on the following slide:
  1. Create a StringMap containing the key/value pairs.
  2. Read in the two-letter abbreviation to translate.
  3. Call get on the StringMap to find the state name.
  4. Print out the name of the state.
The *PostalLookup* Program

```cpp
void main() {
    StringMap stateMap;
    while (true) {
        cout << code << " = " << stateMap.get(code) << endl;
        if (code == "") break;
        string code = getline();
    }
}
```

The Idea of Hashing

- The third strategy on the preceding slide shows that one can make the `get` and `put` operations run very quickly, even to the point that the cost of finding a key is independent of the number of keys in the table. This \( O(1) \) performance is possible only if you know where to look for a particular key.
- To get a sense of how you might achieve this goal in practice, it helps to think about how you find a word in a dictionary. Most dictionaries have thumb tabs that indicate where each letter appear. Words starting with \( A \) are in the \( A \) section, and so on.
- The most common implementations of maps use a strategy called hashing, which is conceptually similar to the thumb tabs in a dictionary. The critical idea is that you can improve performance enormously if you use the key to figure out where to look.

### Hash Codes

- The rest of today’s lecture focuses on the implementation of the `StringMap` class that uses the hashing strategy.
- The implementation requires the existence of a free function called `hashCode` that transforms a key into a nonnegative integer. The hash code tells the implementation where it should look for a particular key, thereby reducing the search time dramatically.
- The important things to remember about hash codes are:
  1. Every string has a hash code, even if you don’t know what it is.
  2. The hash code for any particular string is always the same.
  3. If two strings are equal (i.e., they contain the same characters), they have the same hash code.

### The GetHashCode Function for Strings

```cpp
int hashCode(const string & str) {
    const int HASH_SEED = 5381;
    const int HASH_MULTIPLIER = 33;
    int hash = HASH_SEED;
    for (int i = 0; i < str.length(); i++)
        hash = HASH_MULTIPLIER * hash + str[i];
    return (hash & HASH_MASK);
}
```

### The Bucket Hashing Strategy

- One common strategy for implementing a map is to use the hash code for each key to select an index into an array that will contain all the keys with that hash code. Each element of that array is conventionally called a bucket.
  - In practice, the array of buckets is smaller than the number of hash codes, making it necessary to convert the hash code into a bucket index, typically by executing a statement like
    ```cpp
    int index = hashCode(key) % nBuckets;
    ```
  - The value in each element of the bucket array cannot be a single key/value pair given the chance that different keys fall into the same bucket. Such situations are called collisions.
  - To take account of the possibility of collisions, each elements of the bucket array is a linked list of the keys that fall into that bucket, as illustrated on the next slide.
Simulating Bucket Hashing

```
stateMap.put("AK", "Alaska")
hashCode("AK") = 5862129
5862129 % 7 = 0
The key "AK" therefore goes in bucket 0.
```

Simulating Bucket Hashing

```
stateMap.put("AL", "Alabama")
hashCode("AL") = 5862120
5862120 % 7 = 1
The key "AL" therefore goes in bucket 1.
```

Simulating Bucket Hashing

```
stateMap.put("AK", "Arkansas")
hashCode("AK") = 5862126
5862126 % 7 = 0
The key "AK" also goes in bucket 0.
```

Simulating Bucket Hashing

```
stateMap.get("AK")
hashCode("AK") = 5862126
5862126 % 7 = 0
```

Simulating Bucket Hashing

Linked lists are usually written left to right.

Simulating Bucket Hashing

```
stateMap.put("NH", "New Hampshire")
hashCode("NH") = 5862139
5862139 % 7 = 5
```

Simulating Bucket Hashing

```
stateMap.put("WI", "Wisconsin")
hashCode("WI") = 5862140
5862140 % 7 = 1
```

Simulating Bucket Hashing

```
stateMap.get("NH")
hashCode("NH") = 5862139
5862139 % 7 = 5
```

Simulating Bucket Hashing

```
stateMap.get("WI")
hashCode("WI") = 5862140
5862140 % 7 = 1
```

Simulating Bucket Hashing

```
Suppose you call stateMap.get("NC")
hashCode("NC") = 5862150
5862150 % 7 = 4
The key "NC" must therefore be in bucket 4 and can be located by searching the chain.
```

Simulating Bucket Hashing

```
stateMap.put("CT", "Connecticut")
hashCode("CT") = 5862136
5862136 % 7 = 5
```

Simulating Bucket Hashing

```
stateMap.put("CA", "California")
hashCode("CA") = 5862156
5862156 % 7 = 1
```

Simulating Bucket Hashing

```
stateMap.get("CT")
hashCode("CT") = 5862136
5862136 % 7 = 5
```

Simulating Bucket Hashing

```
stateMap.get("CA")
hashCode("CA") = 5862156
5862156 % 7 = 1
```

Simulating Bucket Hashing

```
Suppose you call stateMap.put("CA")
hashCode("CA") = 5862156
5862156 % 7 = 1
```
Achieving $O(1)$ Performance

- The simulation on the previous side uses only seven buckets to emphasize what happens when collisions occur: the smaller the number of buckets, the more likely collisions become.

- In practice, the implementation of StringMap would use a much larger value for $n_{\text{Buckets}}$ to minimize the opportunity for collisions. If the number of buckets is considerably larger than the number of keys, most of the bucket chains will either be empty or contain exactly one key/value pair.

- The ratio of the number of keys to the number of buckets is called the load factor of the map. Because a map achieves $O(1)$ performance only if the load factor is small, the library implementation of HashMap increases the number of buckets when the table becomes too full. This process is called rehashing.

**The stringmap.cpp Implementation**

```cpp
#include <string>
#include "stringmap.h"
using namespace std;

StringMap::StringMap() {
    nBuckets = INITIAL_BUCKET_COUNT;
    buckets = new Cell*[nBuckets];
    for (int i = 0; i < nBuckets; i++) {
        buckets[i] = NULL;
    }
}
```

**The stringmap.cpp Implementation**

```cpp
StringMap::~StringMap() {
    for (int i = 0; i < nBuckets; i++) {
        Cell *cp = buckets[i];
        while (cp != NULL) {
            Cell *oldCell = cp;
            cp = cp->link;
            delete oldCell;
        }
    }
}
```

**The stringmap.cpp Implementation**

```cpp
StringMap::Cell *StringMap::findCell(int bucket, const string & key) const {
    Cell *cp = buckets[bucket];
    while (cp != NULL && key != cp->key) {
        cp = cp->link;
    }
    return cp;
}
```

**The stringmap.cpp Implementation**

```cpp
void StringMap::put(const string & key, const string & value) {
    int bucket = hashCode(key) % nBuckets;
    Cell *cp = findCell(bucket, key);
    if (cp == NULL) {
        cp = new Cell;
        cp->key = key;
        cp->link = buckets[bucket];
        buckets[bucket] = cp;
    }
    cp->value = value;
}
```

**The stringmap.cpp Implementation**

```cpp
bool StringMap::containsKey(const string & key) const {
    return findCell(hashCode(key) % nBuckets, key) != NULL;
}
```