Eric Roberts
CS 106B

Handout #38
February 15, 2013

Maps and Hashing

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

```
/*
 * File: stringmap.h
 * -----------------
 * This interface exports the StringMap class, which maintains a collection
 * of key/value pairs, both of which are of type string.  These slides
 * further simplify the interface by including only the methods get, put,
 * and containsKey.
 */

#ifndef _hashmap_h
#define _hashmap_h

#include <string>

class StringMap {

public:

/*
 * Constructor: StringMap
 * Usage: StringMap map;
 * ---------------------
 * Initializes a new empty StringMap.
 */

   StringMap();
```

The `stringmap.h` Interface

```
/*
 * Destructor: ~StringMap
 * ----------------------
 * Frees any heap storage associated with this map.
 */

   ~StringMap();

/*
 * Method: get
 * Usage: string value = map.get(key);
 * -----------------------------------
 * Returns the value for key or the empty string, if key is unbound.
 */

   std::string get(const std::string & key) const;

/*
 * Method: put
 * Usage: map.put(key, value);
 * ---------------------------
 * Associates key with value in this map.
 */

   void put(const std::string & key, const std::string & value);
```

The `stringmap.h` Interface

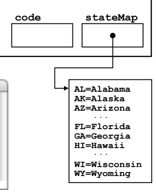
```
/*
 * Method: containsKey
 * Usage: if (map.containsKey(key)) . . .
 * --------------------------------------
 * Returns true if there is an entry for key in this map.
 */

   bool containsKey(const std::string & key) const;

 Private section goes here.

};

/*
 * Function: hashCode
 * Usage: int hash = hashCode(key);
 * --------------------------------
 * Returns a hash code for the specified key, which is always a
 * nonnegative integer.  This function is overloaded to support
 * all of the primitive types and the C++ <code>string</code> type.
 */

int hashCode(const std::string & key);

#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`** ontaining 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

```
int main() {
   StringMap stateMap;
   initStateMap(stateMap);
   while (true) {
      cout << "Enter two-letter state abbreviation: ";
      string code = getLine();
      if (code == "") break;
      if (stateMap.containsKey(code)) {
         cout << code << " = " << stateMap.get(code) << endl;
      } else {
         cout << code << " = ???" << endl;
      }
   }
}
```




```
HI = Hawaii
Enter two-letter state abbreviation: WI
WI = Wisconsin
Enter two-letter state abbreviation: VE
VE = ???
Enter two-letter state abbreviation:
```

Implementation Strategies for Maps

There are several strategies you might choose to implement the map operations **`get`** and **`put`**. Those strategies include:

1. ***Linear search***. Keep track of all the name/value pairs in an array. In this model, both the **`get`** and **`put`** operations run in $O(N)$ time.
2. ***Binary search***. If you keep the array sorted by the two-character code, you can use binary search to find the key. Using this strategy improves the performance of **`get`** to $O(\log N)$.
3. ***Table lookup in a grid***. In this specific example, you can store the state names in a 26×26 **`Grid<string>`** in which the first and second indices correspond to the two letters in the code. Because you can now find any code in a single step, this strategy is $O(1)$, although this performance comes at a cost in memory space.

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 `hashCode` Function for Strings

```
const int HASH_SEED = 5381;
const int HASH_MULTIPLIER = 33;
const int HASH_MASK = unsigned(-1) >> 1;

/*
 * Function: hashCode
 * Usage: int code = hashCode(key);
 * --------------------------------
 * This function takes a string key and uses it to derive a hash code,
 * which is nonnegative integer related to the key by a deterministic
 * function that distributes keys well across the space of integers.
 * The specific algorithm used here is called djb2 after the initials
 * of its inventor, Daniel J. Bernstein, Professor of Mathematics at
 * the University of Illinois at Chicago.
 */

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

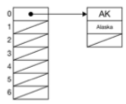
The Bucket Hashing Strategy

- 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

```
int index = hashCode(key) % nBuckets;
```

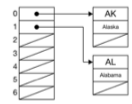
- The value in each element of the bucket array cannot be a single key/value pair given the chance that different keys fall into the same bucket. Such situations are called ***collisions***.
- To take account of the possibility of collisions, each elements of the bucket array is a linked list of the keys that fall into that bucket, as illustrated on the next slide.

Simulating Bucket Hashing



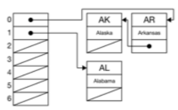
`stateMap.put("AK", "Alaska")`
`hashCode("AK")` → 5862129
5862129 `%` `7` → 0
The key `"AK"` therefore goes in bucket 0.

Simulating Bucket Hashing



`stateMap.put("AL", "Alabama")`
`hashCode("AL")` → 5862130
5862130 `%` `7` → 1
The key `"AL"` therefore goes in bucket 1.

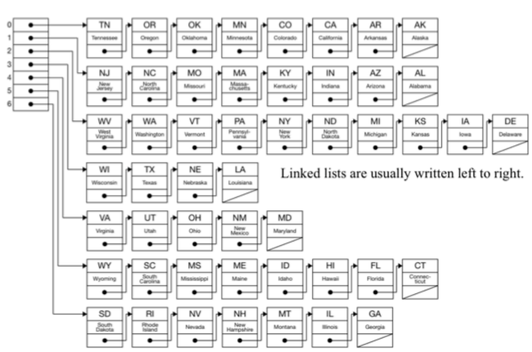
Simulating Bucket Hashing



`stateMap.put("AR", "Arkansas")`
`hashCode("AR")` → 5862136
5862136 `%` `7` → 0
The key `"AR"` also goes in bucket 0.

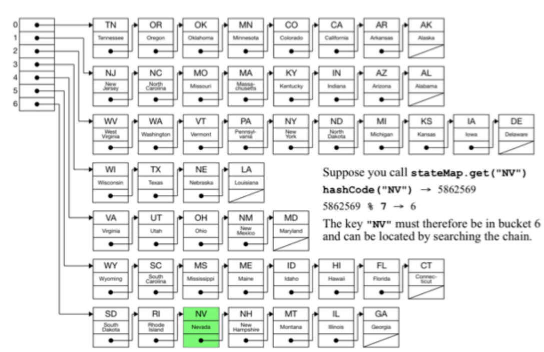
Simulating Bucket Hashing

Simulating Bucket Hashing



Linked lists are usually written left to right.

Simulating Bucket Hashing



Suppose you call `stateMap.get("NV")`
`hashCode("NV")` → 5862569
5862569 `%` `7` → 6
The key `"NV"` must therefore be in bucket 6 and can be located by searching the chain.

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 `nBuckets` 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***.

Private Section of the `StringMap` Class

```
/* Private section */

private:

/* Type definition for cells in the bucket chain */

   struct Cell {
      std::string key;
      std::string value;
      Cell *link;
   };

/* Instance variables */

   Cell **buckets;     /* Dynamic array of pointers to cells */
   int nBuckets;       /* The number of buckets in the array */
   int count;          /* The number of entries in the map   */

/* Private method prototypes */

   Cell *findCell(int bucket, std::string key);
```

The `stringmap.cpp` Implementation

```
/*
 * File: stringmap.cpp
 * -------------------
 * This file implements the stringmap.h interface using a hash table
 * as the underlying representation.
 */

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

/*
 * Implementation notes: StringMap constructor and destructor
 * ----------------------------------------------------------
 * The constructor allocates the array of buckets and initializes each
 * bucket to the empty list.  The destructor frees the allocated cells.
 */

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

The `stringmap.cpp` Implementation

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

/*
 * Implementation notes: get
 * -------------------------
 * This method calls findCell to search the linked list for the matching
 * key.  If no key is found, get returns the empty string.
 */

string StringMap::get(const string & key) const {
   Cell *cp = findCell(hashCode(key) % nBuckets, key);
   return (cp == NULL) ? "" : cp->value;
}
```

The `stringmap.cpp` Implementation

```
/*
 * Implementation notes: put
 * -------------------------
 * The put method calls findCell to search the linked list for the
 * matching key.  If a cell already exists, put simply resets the
 * value field.  If no matching key is found, put adds a new cell
 * to the beginning of the list for that chain.
 */

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

```
/*
 * Implementation notes: containsKey
 * ---------------------------------
 * This method simply checks whether the result of findCell is NULL.
 */

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

/*
 * Private method: findCell
 * Usage: Cell *cp = findCell(bucket, key);
 * ----------------------------------------
 * Finds a cell in the chain for the specified bucket that matches key.
 * If a match is found, the return value is a pointer to the cell
 * containing the matching key.  If no match is found, findCell
 * returns NULL.
 */

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;
}
```