Data Structures: 505 22240 / ESOE 2012

Computer Assignment 4: Hash Tables
Due: three weeks from today, 9:00pm
*Hand in online via **CEIBA***

Total score: 300

This homework will teach you about hash tables, hash codes, and compression functions.

Part I (180)

Implement a class called ***HashTableChained***, a hash table with chaining. ***HashTableChained*** extends an abstract class called ***Dictionary***, which defines the set of methods [like *insert*(), *find*() and *remove*()] that a dictionary needs.

The methods you will implement insert() an entry (key + value) into a hash table, find() an entry with a specified key, remove() an entry with a specified key, return the size() of the hash table (in entries), and say whether the hash table isEmpty(). There is also a makeEmpty() method, which removes every entry from a hash table, and two ***HashTableChained*** constructors. One constructor lets applications specify an estimate of the number of entries that will be stored in the hash table; the other uses a default size. Both constructors should create a hash table that uses a prime number of buckets. (Several methods for identifying prime numbers were discussed earlier in the semester.) In the first constructor, shoot for a load factor between 0.5 and 1. In the second constructor, shoot for 100 buckets. Descriptions of all the methods may be found in **Dictionary.h** and **HashTableChained.h**.

Do not change **Dictionary.h** or any prototypes in **HashTableChained.h**. Most of your solution should appear in **HashTableChained.cpp**, but other classes are permitted. You will probably want to use a linked list code, of your choice.

Look up the hashCode method in the STL or other C++ libraries. Besides the lecture notes, hashCode methods are also covered in Section 18.4 of textbook. Assume that the objects used as keys to your hash table have a hashCode() method that returns a "good" hash code between Integer.MIN_VALUE and Integer.MAX_VALUE (i.e., between -2147483648 and 2147483647). Your hash table should use a compression

function, as described in lecture, to map each key's hash code to a bucket of the table. Your compression function should be computed by the compFunction() method in ***HashTableChained***. Your *insert*(), *find*(), and *remove*() should all use this *compFunction*() method. For simplicity, the methods *insert*() and *remove*() do NOT return an entry, unlike the discussion in lecture. Similarly, the method *find*() returns a boolean indicating whether the key is found or not.

Compression functions

Consider the following statement for a hash code k and an N-bucket hash table:

$h(k) = | ak + b | \bmod N$,

is "a more sophisticated compression function" than

$h(k) = | k | \bmod N$.

Actually, the "more sophisticated" function causes "exactly" the same collisions as the less sophisticated compression function; it just shuffles the buckets to different indices. The better compression function is

$h(k) = ((ak + b) \bmod p) \bmod N$

where p is a large prime that's substantially bigger than N. (You can replace the parentheses with absolute values if you like; it doesn't matter much.)

For this assignment, the simplest compression function might suffice. The bottom line is whether you have too many collisions or not in Part II. If so, you'll need to improve your hash code or compression function or both. To test the codes in Part I, you will need to store objects using your hash table. Two classes ***Double*** and ***String*** store one single variable of **double** and **string**, respectively. They will be used as the key of *Entry* for inserting into the bucket of your hash table.

Part II (120)

It is also often useful to hash data structures other than strings or integers. For example, game tree search can sometimes be sped by saving game boards and their evaluation functions. The class ***CheckerBoard*** represents an 8 × 8 game board. Each position has one of three values: 0, 1, or 2.

Now, your job is to fill in two missing methods: equals() and hashCode() in the three classes, namely, ***Double***, ***String***, and ***CheckerBoard***. The *equals*() operation should be true whenever the objects have the same stored values for ***Double*** and ***String*** and the same pieces in the same locations for ***CheckerBoard***. The *hashCode*() function should return a "good" hash code. In particular, if two objects are *equals*(), they

have the same hash code. Besides the lecture notes, you can refer to Section 18.4 of textbook (or other sources) for more detail on hash code functions, particularly for ***Double*** and ***String***.

You will be graded on how "good" your hash code and compression function are. By "good" we mean that, regardless of the table size, the hash code and compression function evenly distribute objects throughout the hash table. Your solution will be graded in part on how well it distributes a set of randomly constructed objects. Hence, for ***CheckerBoard***, the sum of all the cells is not a good hash code, because it does not change if cells are swapped. The product of all cells is even worse, because it's usually zero. What's better? One idea is to think of each cell as a digit of a base-3 number (with 64 digits), and convert that base-3 number to a single int. (Be careful not to use floating-point numbers for this purpose, because they round off the least significant digits, which is the opposite of what you want. Better to round off the most significant digits, which is what happens when an int gets too big.)

Do not change any prototypes in **CheckerBoard.h**. The file **TestHashTable.cpp** is provided to help you test your ***HashTableChained*** and your ***CheckerBoard*** together. Note that **TestHashTable.cpp** does NOT test all the methods of ***HashTableChained*** and other key classes such as ***Double*** and ***String***; you should write additional tests of your own. Moreover, you will need to write a test to see if your hash code is doing a good job of distributing ***CheckerBoard*** (***Double*** or ***String***) evenly through the table.

A tutorial on collision probability

Students are always surprised when they find out how many collisions occur in a working hash table. You might have the misimpression that there won't be many collisions at all until the table is nearly full. Let's analyze how many collisions you should expect to see if your hash code and compression function are good.

If you have N buckets and a good (pseudorandom) hash function, the probability of any two keys colliding is $1/N$. So when you have i keys in the table and insert key $i+1$, the probability that the new key does NOT collide with any old key is $(1-1/N)^i$. If you insert n distinct items, the expected number that WON'T collide with any previous item is

$$\sum_{i=0}^{n-1}(1-1/N)^i = N - N(1-1/N)^n$$

so the expected number of collisions is

$$n - N + N(1 - 1/N)^n$$

Now, for any n and N you test, you can just plug them into this formula and see if the number of collisions you're getting is in the ballpark of what you should expect to get. For example, if you have $N = 100$ buckets and $n = 100$ keys, expect about 36.6 collisions.