San José State University
Department of Computer Engineering

CMPE 180-92

Data Structures and Algorithms in C++

Fall 2016
Instructor: Ron Mak

Assignment #11B
(with extra credit)

Assigned: Sunday, November 6
Due: Wednesday, November 16 at 11:59 PM
URL: http://codecheck.it/codecheck/files/16110706254je0qpgyza0hg6hi5tzkxhhgo
Canvas: Assignment 11.b. Map and hash tables
Points: 100

Map and hash tables

This assignment will give you practice with the built-in STL map and with programmer-written hash tables. You will work with hash tables that handle collisions with linear probing, quadratic probing, and with separate collision chains. You can also experiment with different table sizes and hash functions.

Input data will be a text file of the U.S. constitution and its amendments:
http://www.cs.sjsu.edu/~mak/CMPE180-92/assignments/11B/USConstitution.txt

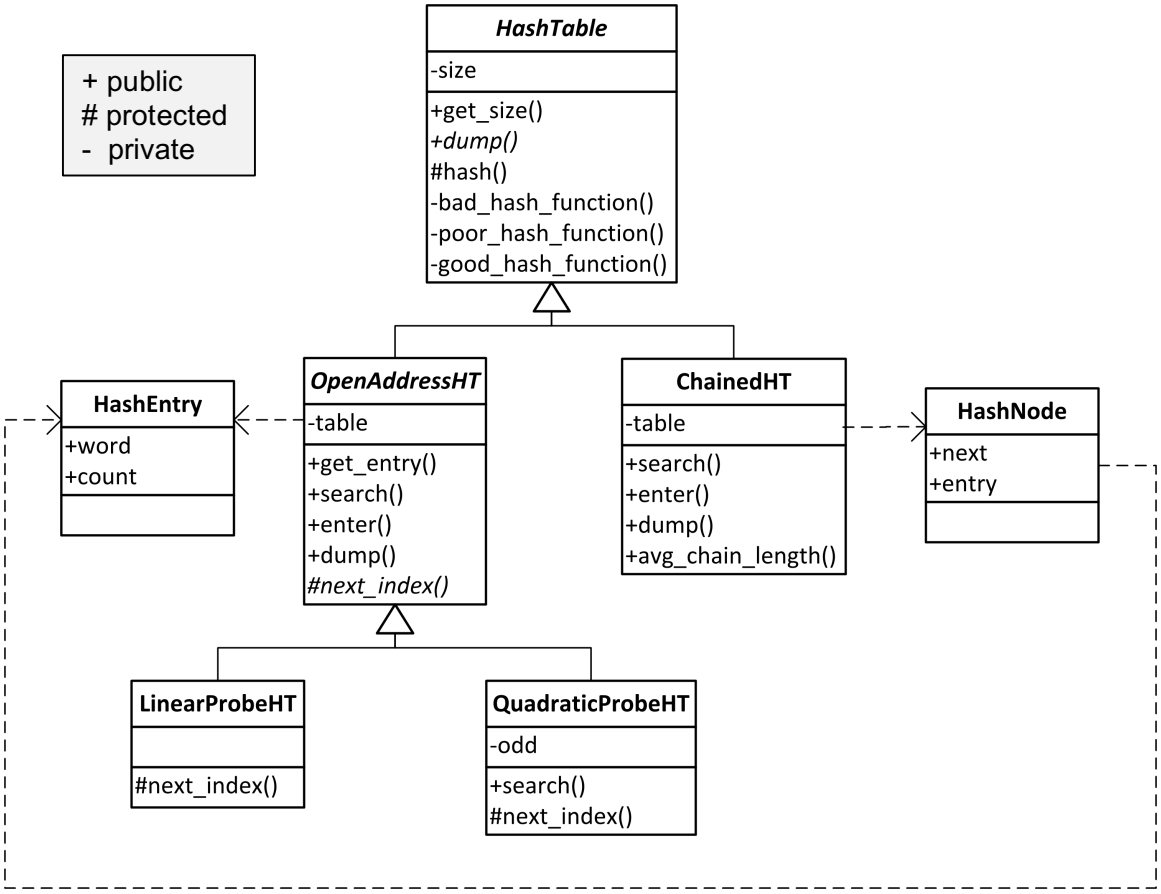
Enter the input file's words into the map and hash tables. Store with each word its frequency count – how many times it appears in the input text. In other words, the purpose of each map or hash table is to store, update, and enable access to the number of times each word occurs in the input file. For example, after processing the input file, if you look up the word "president" (using the word as the key), you should get the value 121 (which is how many times the word appears in the file). You will also count the total number of probes it took to search for words in each hash table.

Class hierarchy

You are provided the skeleton of a class hierarchy shown below in the UML (Unified Modeling Language) class diagram.

Class **`HashTable`** is the base class. It has two subclasses, **`OpenAddressHT`** and **`ChainedHT`**, which are hash tables that use open addressing and separate collision chains, respectively. Class **`OpenAddressHT`** in turn has two subclasses, **`LinearProbeHT`** and **`QuadraticProbeHT`**, which are open addressing hash tables that use linear probing and quadratic probing, respectively, to resolve collisions.

You enter objects of class **`HashEntry`** into the hash tables. Each object has a **`word`** (from the input file) and its frequency **`count`**. The word is also the hash key. A **`ChainedHT`** hash table uses objects of class **`HashNode`** to create collision chains. Each node has a pointer to the **`next`** node in the chain, and an **`entry`** pointer to a **`HashEntry`** object.



Base class `HashTable`

Class **`HashTable`** has private member **`size`** for the size of the hash table. The private hash tables (implemented as vectors) are in subclasses **`OpenAddressHT`** and **`ChainedHT`**. Class **`HashTable`** is abstract because of its abstract member function **`dump()`**. Because they are abstract, the names **`HashTable`** and **`dump`**() appear in the class diagrams in a *slanted type*. All abstract functions such as **`dump`**() must be defined in a subclass. Member function **`hash()`** is the shared hash function for all the hash tables of this assignment.

As their names suggest, private hash functions **`bad_hash_function()`**, **`poor_hash_function()`**, and **`good_hash_function()`** are hash functions with different levels of quality. Member function **`hash()`** selects one of them to call and return its value. *You can experiment with other hash functions.*

Subclass `OpenAddressHT`

Hash tables that use open addressing store all their data within the hash table itself. They maintain collision chains internally, and not in separate linked lists. When searching for an entry, such a hash table can use linear probing or quadratic probing.

`OpenAddressHT` (and therefore its subclasses) uses a fixed-size vector to store the hash table. Each vector element is either nullptr or a pointer to a **`HashEntry`** object. The vector is protected (rather than public or private) to allow access only by the subclasses.

Whether it uses linear or quadratic probing, subclasses of **`OpenAddressHT`** share definitions of member functions **`get_entry()`**, **`search()`**, **`enter()`**, and **`dump()`**.

For each word being entered into a hash table, if the word is already in the table, simply update its frequency count. If the word isn't already in there, enter it with a count of 1.

Class `HashEntry`

The hash tables will store objects of class **`HashEntry`**. Each object contains a **`word`** from the input file and the word's frequency **`count`**. The word is also the hash key.

Subclass `LinearProbeHT`

`LinearProbeHT` inherits most of what it needs from **`OpenAddressHT`**. It only has to define member function **`next_index()`**. Given the current value of the **`index`** of the hash table slot that was just probed, what is the **`index`** value of the next slot to probe?

Subclass `QuadraticProbeHT`

Similarly, **`QuadraticProbeHT`** inherits most of what it needs from **`OpenAddressHT`**. It has to define member function **`next_index()`**. Given the current value of the **`index`** of the hash table slot that was just probed, what is the **`index`** value of the next slot to probe? Tip: Recall the proof by induction that $n^2 = 1 + 3 + 5 + 7 + ... + 2n - 1$. Use private member variable **`odd`** to help compute the squares. When do you reset **`odd`** to 1?

Subclass `ChainedHT`

`ChainedHT`, like **`OpenAddressHT`**, must define member functions **`search()`**, **`enter()`**, and **`dump()`**, although with different return values.

As in **`OpenAddressHT`**, a fixed-size vector stores the hash table. Each of this vector's elements is either nullptr or a pointer to the head of a collision chain.

Class `HashNode`

Collision chains are made up of **`HashNode`** objects. Each object has a pointer to the **`next`** node in the chain and a pointer to a **`HashEntry`** object.

The `main()`

File **HashTests.cpp** contains several tests.

Based on the number of distinct words in the input file (1138, as determined by an early run), the size of the **`LinearProbeHT`** table is rounded up to 1150. This leaves a few extra slots, and so the table will be sensitive to the quality of the hash function.

The size of the **`QuadraticProbeHT`** table is fixed at 2281, which is the smallest prime number larger than 2 x 1138 = 2276. This size guarantees that quadratic probing (if done correctly) will always find an empty slot while entering objects into the table.

Finally, the size the **`ChainedHT`** hash table is arbitrarily fixed at 500. This size will determine the length of the collision chains.

You can experiment with other hash table sizes.

Function **`process_input()`** does most of the work with the STL map and the hash tables. It reads each word and enters it into the map and into each hash table. Note that it enters only distinct words, and all the letters are made lower case. The various hash table member functions must keep track of the total number of probes used to search for words. One probe is a check of a single hash table slot for a word match.

Function **`print_stats()`** prints some useful statistics.

Function **`test_word()`** tests the map and the three hash tables with a set of words from the input file, one at a time. If all went well, the map and each hash table should give the same frequency counts for each test word. The function also prints the number of probes to find each word in each hash table. The probe counts may vary.

For debugging purposes, each hash table has a **`dump()`** member function that sequentially prints the contents of the table. Normally, don't dump the tables. But if there is a **`--dump`** argument on the command line that runs the program, then execute the **`dump()`** calls.

Note the parameters to **`main()`**. The **`argc`** parameter contains the count of the number of command line arguments, and the **`argv[]`** parameter is an array of pointers to C-strings of the command line arguments. By convention, **`argv[0]`** points to the name of the program being run, **`argv[1]`** points to the first argument, etc.

Sample output

Your output without the dumps of the STL map and the hash tables should be similar to:

```
Statistics for file USConstitution.txt:

                  Lines: 865
            Total words: 7541
         Distinct words: 1138

       Linear table size: 1150
     Total linear probes: 58646
   Average linear probes: 7.77695

    Quadratic table size: 2281
  Total quadratic probes: 8464
Average quadratic probes: 1.1224

      Chained table size: 500
    Average chain length: 2.54586
    Total chained probes: 13884
  Average chained probes: 1.84114

Tests with the word 'amendment':

                  STL map: 35-amendment
   Linear probe hashtable: 35-amendment (1 probes)
Quadratic probe hashtable: 35-amendment (1 probes)
        Chained hashtable: 35-amendment (1 probes)

Tests with the word 'article':

                  STL map: 28-article
   Linear probe hashtable: 28-article (1 probes)
Quadratic probe hashtable: 28-article (1 probes)
        Chained hashtable: 28-article (5 probes)

...

Tests with the word 'vote':

                  STL map: 16-vote
   Linear probe hashtable: 16-vote (1 probes)
Quadratic probe hashtable: 16-vote (1 probes)
        Chained hashtable: 16-vote (2 probes)


Done!
```

Your statistics may not be exactly the same, but they should be close. For the tests of each word, the frequency counts should be the same for the STL map and the hash

tables, although the number of probes may differ. CodeCheck will not compare your output.

Make a separate run outside of CodeCheck with the dumps of the STL map and the hash tables, and make a text file of the output. For sample dump output, see http://www.cs.sjsu.edu/~mak/CMPE180-92/assignments/11B/Assignment11B-output.txt

What to submit

Submit the signed zip file into **Canvas: Assignment 11.b. Map and hash tables**. Also submit the text file containing the output from the map and table dumps.

You can submit as many times as necessary to get satisfactory results, and the number of submissions will not affect your score. When you're done with your program, click the "Download" link at the very bottom of the Report screen to download the signed zip file of your solution.

Extra credit (20 points)

Find a hash function that lowers the average number of probes for all three hash tables (with the given table sizes) with the input file. Submit a copy of your **`hash()`** function in class **`HashTable`**. Make a separate run using your hash function and submit a copy of the output that shows lower probe count averages. Include map and hash table dumps in this run.

Rubrics

Criteria	Maximum points
Statistics (should be the same as the sample output) • Lines • Total words • Distinct words	**15** • 5 • 5 • 5
Probe counts and averages (should be close to the sample output) • Linear probe hash table • Quadratic probe hash table • Chained hash table	**45** • 15 • 15 • 15
Word tests (20 words, must have the same frequency counts for each word)	**20**
Map and hash table dumps (in a separate output text file) • Map • Linear probe hash table • Quadratic probe hash table • Chained hash table	**20** • 5 • 5 • 5 • 5
Extra credit (A hash function with all lower probe count averages) • Function code and output with hash table dumps	**20**