ICS 46
Homework Assignment 5

This programming assignment will consist of 4 parts:

1. Implementing and error testing the class *ChainedHashTable*.
2. Running experiments to empirically analyze your implementation.
3. Adjusting your class to collect additional data about its performance.
4. Using all of the above to compare the performance of various hash functions.

---

1. (30 points) Implementing and Testing class *ChainedHashTable*.

- Firstly, since we are storing both keys and values, we need to modify our *ListNode* struct (or class) to have two data members
  - one data member stores the key
    - for this assignment will be of type *string*
  - the other data member stores the value
    - for this assignment is a count, and thus type *int*

Note: you do not need to define ChainedHashTable as a templated class this week, but you can imagine how useful it would be to make it a templated class and generalizing what types of keys and values your hash table could be utilized for!

- For this assignment, each hash table should have an array capacity of 5000 elements

- Implementing and testing our *ChainedHashTable*...

Suggestion: create a *testHash.cpp* similar to the following:
(feel free to define classes in their own *.h* files and *#include* them)

```
//an abstract struct to parent your various hasher classes//
struct Hasher {
    virtual int hash(string s, int N) = 0;
};

//your first working hashing class//
struct GeneralStringHasher: Hasher {
    int hash(string s, int N) {
          //use the "General Hash Function for Strings"
          //algorithm from lecture
     }
};


//continued on the next page…
```

```
class ChainedHashTable {
Private:
        //data members
        //hint: have a data member that is a Hasher REFERENCE
        //(make sure you understand why!!!)
Public:
        ChainedHashTable(int capacity, Hasher & myHasher) {
            //implement your constructor
        }

        //implement your copy constructor
        //implement your destructor

        //good place to stop and create your first testing
        //functions and call them from your testHashTable
function
        //(see below). Run your testHashTable to check for
     errors.
        ////////////////////////////////////////////////////

        //continue incrementally adding methods to your class
        //and the respective test cases to testHashTable.
Continue
        //testing your class incrementally.
};

//individual method-testing functions//
void testConstructor(Hasher & hasher)
{
        //create an empty ChainedHashTable object
}

void testCopyConstructor(Hasher & hasher)
{
        //create a ChainedHashTable object
        //create a second object as a copy of the first object
}

//etc.

//overall tester function//
void testHash(char *inputFileName, Hasher & hasher)
{
        //call test functions

        //you may want to instantiate a ChainedHashTable
        //object to pass as a reference to some of your
        //more advanced testing functions
}
```

```
//continued on the next page...


int main()  {
        GeneralStringHasher h;
        testHash("random.txt", h);
}
```

- Define *operator[]* on your *ChainedHashTable*
  - If you have a *ChainedHashTable* object called *table* and a string object called *word*, then *table[word]*++, should result in incrementing the counter (ie. the *value*) of the element with *key* matching *word*
    - *hint: think carefully about what operator[] needs to return for this to work properly*

- **<u>For your report</u>**, you will analyze the time complexity of each of the following functions:
    - ChainedHashTable::insert()
    - ChainedHashTable::find()
    - ChainedHashTable::remove()
    - void insertAll(ChainedHashTable & h,...)
    - void findAll(ChainedHashTable & h,...)
    - void removeAll(ChainedHashTable & h,...)
    - int hash(string s, int N) // better be O(1)
    - int & operator[] (string s)

  For each of the above functions, do the following:
  - include copy-pasted/screenshotted code
  - show (either commented in the code or directly in the report) calculations leading to...
    - Worst case Time Complexity (Note LinkedList operatons are O(L) where L is length of list)
    - Typical case Time Complexity - note this should be better than Worst case Time Complexity

    (For these analyses, you may assume the hash function produces a uniform distribution of hashing indices across the keys it encounters, and that the data you are hashing does not degenerate the hash table performance)
  - **[optional]** as an easy and important extra practice, you may also calculate the worst-case time-complexities, however ***<u>please separate this from the other analyses and label it clearly as the "optional practice" as it will not be graded and may cause confusion otherwise</u>***

---------- End of Part 1 ----------

---

2. (30 points) Empirical Testing of Your ChainedHashTable

- You will be using two input files of words:
  - random.txt
  - words.txt.

- Create the following three global functions (not class methods) in testHash.cpp:
  - *insertAll*
  - *findAll*
  - *removeAll*

- Measure the time for each of the above functions in the following way...

```
For each input file,

    Starting with k=1, insert k/10th of the words (~4500*k
    words) into your hash table object,

    Using your global functions, measure the time to…
        - insert all the words from the file into your table
        - find all the words from the file in your table
        - remove all words from the file from your table
    Increment k and repeat until k = 10
```

- **For your report**, include two graphs ***OR*** two tables of the results. Here is a sample you can use/modify with your data.
  - Your graphs/tables should have
    - X axis: N (the number of inputs N (4500, 9000, … 45000))
    - Y axis: T(N) (time taken for running findAllWords on N inputs)

  Example:

random.txt				
	N (number of inputs)			
	4500	9000	...	45000
insertAll T(N)	0.1s	0.25s	...	1.0s
findAll T(N)	0.1s	0.25s	...	1.0s
removeAll T(N)	0.1s	0.25s	...	1.0s

words.txt	
...	……...

---------- End of Part 2 ----------

3. **(10 points) Preparing for Additional Evaluations**

- We can evaluate the performance of our hash table more thoroughly by keeping track of / calculating the following statistics:
  - maximum chain length
  - minimum chain length
  - average chain length
  - standard deviation

Clarification on the [Statistics] for the assignment

I have multiple questions regarding the statistics.

First:

We are asked to [keep] statistics for max chain length, min chain length, average chain length, and standard deviation.

1) How are we expected to calculate [standard deviation]? I have no idea what it is.

Second:

2) How do we get these statistics?

A) Should we store them in [data members] as we [insert] items into our hash table?

(e.g. If I insert into a chain and it is longer than my current max chain length, updated the int max [data member])

OR

B) Should we write additional methods that traverse our table while counting, basically acquiring the statistics [AFTER] we already inserted all our items into the table?

(Updated) Third:

"....Also report the time to insert, find, and remove all words using that particular hash function..."

3) Should we use Timer.h to measure the time it takes for these functions?

Also,

4) Should we:

A) [print] the time it takes to perform the three functions in a row?

[OR]

B) Should we print the time it takes to do [insert], then print the time it takes to do [find], and then print the time it takes to do [remove]?

1) For standard deviation, you will be using this formula:

$$S = \sqrt{\frac{\Sigma(X - \overline{X})^2}{N}}$$

where S = the standard deviation of a sample,
Σ means "sum of,"
X = each value in the data set,
$\overline{X}$ = mean of all values in the data set,
N = number of values in the data set.

2) To calculate each of these statistics, you can make additional data members/calculation functions for each one, and update them after each modification to the table.

- Keep a list of LinkedList lengths, i.e. chained_list_lengths.
- To calculate min chained length,
  - `min(chained_list_lengths)`
- To calculate max chained length,
  - `max(chained_list_lengths)`
- To calculate average chained length,
  - `sum(chained_list_lengths) / len(chained_list_lengths)`
- To calculate the standard deviation,

```
for length in chained_list_lengths :
   variance_sum += (chained_list_lengths - average_chained_list_length)^2
variance = variance_sum / len(chained_list_lengths)
```

  - `std_dev = sqrt(variance)`

3) Yes, you can use Timer.h to measure execution time.

4) Both printing after each function or all together is fine.

- For your report, no output from this step is required (as the output will come from Part 4 below).

---------- End of Part 3 ----------

---

4. (30 points) Comparing the Performance of Three Different Hash Functions.

- The three hash functions to be compared are:
  - the one I gave in lecture that makes a hash by shifting in the lower 6 bits from each character
  - one that sums up the ASCII codes of the string

- one that multiplies the ASCII codes of the string together

- Create three additional Hasher-derived structs with respect to the abovementioned hash functions

```
//A rough idea of these Hashers - //
//some modifications may be needed//
struct SumHasher : Hasher {
     int hash(string s, int N) {
          int result = 0;
          for (int i=0; i<s.size(); ++i)
               result += s[i];
          return abs(result) % N;
     }
};

struct ProdHasher : Hasher {
     int hash(string s, int N) {
          int result = 1;
          for (int i=0; i<s.size(); ++i)
               result *= s[i];
          return abs(result) % N;
     }
};
```

- For your report, do the following for each hash function,
  - Insert all the words from random.txt, then print out the following statistics (using the functions constructed in Part 3):
    - min chain length
    - max chain length
    - average chain length
    - standard deviation
  - Report the time to insert, find, and remove all words using that particular hash function.

    Example console output:

```
Hash function 1 chain length statistics:
     min = 0; max = 400; average = 10.3; std_dev = 4.5
     insertAll = 70 sec
     findAll = 100 sec
     removeAll = 85 sec

Hash function 2 chain length statistics:
     //(etc…)
```

---------- End of Part 4 ----------

Submission Requirements:

1. Submit a zip of your program to Canvas.
    - The zip file should include your source code and Makefile.

2. Submit a pdf report to Gradescope with the following format:
    - Code Implementation & Time complexity Analysis
        - Five points off for each error in time complexity and for each incorrect method or function up to the maximum.
        - Copy/Paste your code for the functions (with their time complexity assuming N is the number of words in the input file)
            - ChainedHashTable::insert(string s, Object/int count)
            - ChainedHashTable::find(string s)
            - ChainedHashTable::remove(string s)
            - void insertAll(ChainedHashTable & h,...)
            - void findAll(ChainedHashTable & h,...)
            - void removeAll(ChainedHashTable & h,...)
            - int hash(string s, int N)
            - int & operator[] (string s)
    - Proof of execution under valgrind with no memory leaks
        - You can run you code with a reduced length input file if execution is taking a long time.
            - E.g., instead of using random.txt, use just the first 4500 words from random.txt
    - Testing output
        - 2 graphs OR 2 tables (1 for random.txt, 1 for words.txt) (see step #2)
            - Showing the results of measuring findAll using your ChainedHashTable implementation and the general string hash function, over different sizes of N
            - Your graphs/tables should have:
                - X axis: N (the number of inputs N (4500, 9000, … 45000))
                - Y axis: T(N) (time taken for running findAllWords on N inputs)
            - See example above in step #2
        - 3 console outputs (1 for each hash function, see step #4)
            - Showing chain length statistics for each hash function
            - Example console output:
                - Hash function 1 chain length statistics:
                    - min = 0; max = 400; average = 10.3; std_dev = 4.5
                - Hash function 2 chain length statistics...

Homeworks that are not formatted nicely will not be accepted. Pay careful attention to memory management. Do not share memory across objects. Obvious code copying, from a textbook, from another student, or within your own program, will be down-graded (write everything once (perhaps by writing auxiliary functions), then call the function or method each time that operation is required). Feel free to add additional functions or methods, but please use the data members outlined in the homework.