Programming Assignment #4
Incentive Date: Mar 26th 2024
Due Date: Mar 31st 2024
Hash Tables & Maps

Description:
For this programming assignment, you will implement a **Hash Table** which uses a string type as a key and an integer as the value associated with the key (i.e. an unordered_map<string, key>). You will **build three different implementations** of this Hash Table. The first implementation will use Separate Chaining (**ChainingHashTable.h**), the second implementation will use Linear Probing (**ProbingHashTable.h**), and the final implementation will be based on Double Hashing (**DoubleHashTable.h**).

starter.zip (Starter Code):
The following is a brief description of the starter code provided to you:

- **AbstractHashTable.h**
  - Declares the pure virtual functions of the hash table such as **insert**, **remove**, **get**, **contains**, **resizeAndRehash** that each of the three child classes must implement.
    - **void insert(string key, int val):** This function inserts the key with the associated value "val" in the hash table. If the key already exists in the hash table, the value associated with the key is updated. During insertion, when the load factor exceeds the maximum load factor, you must resize and rehash the table. The insert operation must avoid duplicate keys.
    - **int remove(string key):** This function removes the key from the hashtable if the key exists. If the key does not exist in the hash table, it throws a **std::out_of_range exception**.
    - **int get(string key):** This function returns the value associated with the key if the key exists. If the key does not exist in the hash table, it throws a **std::out_of_range exception**.
    - **bool contains(string key):** This function returns true if the key exists in the hashtable and false if it does not.
    - **void resizeAndRehash():** resizes and rehashes the hash table if current load factor exceeds the max load factor.
      - You can use the smallest prime number which is > (2 * current table size) as the new size for rehashing the table.
      - Deleted entries must not be rehashed; only existing elements must be rehashed to the new hash table
  - The AbstractHashTable parent class itself must implement a few methods that each of the child classes can use (as they are common to each implementation):
    - **int hash(std::string s):** returns the index to which the particular string "s" taken in as a parameter is hashed to. You can use the following pseudocode (where ***c*** is a positive constant chosen by you; experiment with various

positive constants to figure out which choice results in faster runtimes) to develop your hash function:

```
unsigned long hash = 0
int n = s.length
for (int i = 0; i < n; i++) {
    hash = c * hash + ASCII value of s[i]
}
index = hash % capacity // capacity is current memory size allocated for hash table
```

- **int findNextPrime(int n)**: returns the next prime number > **n**
- **bool isPrime(int n)**: return true if **n** is prime and false if not.
- **float load_factor()**: returns the current load factor of the hash table
  - Defined as: (number of elements mapped in the hash table / Number of slots in the hash table)
- **int getSize()**: returns the number of elements in the hash table

- The AbstractHashTable class also implements the following structs and variables that will be used by its child classes:
  - **struct HashEntry**: which consists of the following members -
    - **string key**: the key associated with the hash entry
    - **int val**: the value associated with the key
    - **bool isFilled**: is set to true if the hash entry is filled and false otherwise
      - This boolean flag helps indicate whether the HashEntry in the given slot is inserted (i.e. not empty)
        - This will be set to false if the inserted slot is deleted. Upon a new insertion to the deleted slot, this will be set to true again.
    - **bool DELETED**: is set to true if the hash entry is deleted and false otherwise.
      - This boolean flag is important as you must continue to search for a key if you've encountered a DELETED slot in the hash table (which means it was previously filled).
      - Upon the insertion of a key in the hash table, you must first ensure that this key does not already exist in the hash table. If the key did not exist previously, you can insert this new key in the first DELETED slot you come across in the probing sequence. If you do not encounter a DELETED slot in your probing sequence, you can insert it at the first empty slot encountered (i.e. not deleted and not filled). On the other hand, if the key already exists in the hash table, you can simply update

the value of the associated HashEntry of the key.

- **You can use a combination of the DELETED and isFilled variables during insertion / search / removal of a key to figure out the appropriate slot that you must work with in these operations.**
  - **A HashEntry that contains a key has: isFilled = true and DELETED = false**
  - **A HashEntry that has never contained a key before (i.e. has always been empty) has: isFilled = false and DELETED = false**
  - **A HashEntry that contained a key which has been deleted has: isFilled = false and DELETED = true**
- **Note that the isFilled and DELETED flags are only necessary for Linear Probing and Double Hashing, they do not need to be used for Separate Chaining.**

- **int capacity:** keeps track of the capacity of the hash table
- **int num_elements:** keeps track of the number of elements in the hash table
- **float maxLoadFactor:** sets the maximum load factor of the hash table; initialized in the constructor of the child classes (each child class can set its own unique max load factor).

- **ChainingHashTable.h, ProbingHashTable.h, DoubleHashTable.h**
  - These are derived classes of the AbstractHashTable class and must implement all of the pure virtual functions.
  - The ChainingHashTable class uses the **vector<list<HashEntry>> table** member to store the hash table and implements the hash table using separate chaining.
  - The ProbingHashTable and DoubleHashTable classes use the **vector<HashEntry> table** member to implement the hash table using open addressing. The only difference between the ProbingHashTable and the DoubleHashTable classes is the way in which these classes handle collisions. The DoubleHashTable class uses the private **int secondHash(string s)** method to compute the step size during collisions.
    - You should review and consider course materials and literature (the youtube video linked on the lecture slides pertaining to double hashing may be of help here) regarding double hashing and implement an efficient secondary hash function.
      - Note that the use of the largest prime number smaller than the current table capacity may come of good use for the second hash function (the int prevPrime member variable will help store this value).

Contract (TASK):

Your task is to implement the methods declared in the AbstractHashTable, ChainingHashTable, ProbingHashTable, and DoubleHashTable classes. For each of these classes, you have been provided with non-functional definitions of these methods in their respective header files. **You must implement these method definitions in the given header files**, by replacing the current non-functional definitions with your functional implementations. **Do not create additional cpp files to define these functions, make sure that they are part of the header files.**

[Advisory]:

- You may choose **11** to be the initial capacity of the vector when allocating memory in the constructor
- You must throw a **std::out_of_range** exception when you come across invalid calls to the get and remove methods
- Experiment with different positive constants in the hash function to ensure you minimize running time.
- You may pick a value between 0.5 - 0.7 as the load factor for the open addressing hash tables. You may pick a value between 0.9 - 3 as the load factor for the Separate Chaining Hash Table.
    - You are encouraged to refine these load factors to improve the runtime performance of your implementation.

Deliverables and Submissions:

In a zip folder named using the format **<FirstName>-<LastName>-<UIN>-<PA4>.zip** you must submit the following file(s) to Canvas:

- **AbstractHashTable.h** (containing the functional method definitions of the AbstractHashTable class)
- **ChainingHashTable.h** (containing the functional method definitions of the ChainingHashTable class)
- **ProbingHashTable.h** (containing the functional method definitions of the ProbingHashTable class)
- **DoubleHashTable.h** (containing the functional method definitions of the DoubleHashTable class)
- **PA4_Report.pdf** (containing your report - details mentioned below)

Do not use angular brackets in the name of the submission folder.

Grading:

This assignment is worth 80 points. The 80 points will be split up between Coding (60 points) and a Report (20 points).

Coding (60 points)

Each method of the three classes will be tested using an automatic testing infrastructure. You will be provided with the testing infrastructure which you can use to evaluate your implementations.

The grading rubric is as follows:

- Each of the following methods will be tested out for each of the three classes:

- insert only (depends on get, contains, getSize): 5 points
- remove only (depends on insert, contains, get, getSize): 5 points
- insert and remove together: 10 points

Report (20 Points):

- **Theoretical Statement:** State the time complexities for insert/removal operations (amortized and total) for a hash table.
- **Experimental Analysis:** Create graph(s) displaying the runtimes of **insert()** operations for all three implementations. Briefly compare the performances of the implementations based on your results.
  - **Your graph should begin with low sizes (example: 10) and go until at least a maximum size of 1 million (you're welcome to go higher). You should insert unique strings into the hash table.**
- **Discussion:** Write a few sentences explaining your experiment and outcomes. For example:
  - Were your results what you expected? Did your experimental data deviate from theoretical runtimes, or were they similar? If they deviated, why?
    - **Hint: think of how the discussion we had around caches in LE3 can potentially influence runtimes.**
  - Which implementation performs the best? Is there an implementation that performs better for all capacities, or are certain implementations more efficient at different capacities?

testInfrastructure.zip (Test Methodology):
In order to familiarize you with the auto-grading environment on our end we are providing you with an automated test infrastructure that can be used to check the correctness of your code. The testing infrastructure is constructed as follows:

- **dictionary.txt:** This text file contains around 1 million words (not all unique).
- **hash_test.cpp:**
  - **hash_test.cpp** implements a **word frequency algorithm**. In order to insert a word into a hashtable, hash_test.cpp uses your three classes; it checks whether the word already exists in the hashtable before insertion, if so it obtains the current value and inserts (i.e. updates) the key with the obtained value + 1. If the key does not exist in the hashtable, it inserts the key with the value 1.
  - This cpp file uses **dictionary.txt** to test your code out on the test functions that have been mentioned in the grading rubric above. It also runs these tests and outputs your final score.

- **test.py:** This file compiles hash_test.cpp in order to produce the executable. It then runs this executable which will indicate your score (as well as the test cases you have passed or failed).

Note: There is a possibility that one or more of your methods causes a segmentation fault which in turn causes the test script to crash when calling that particular method. In such a case, we will not be able to calculate your score for the assignment and you will be assigned a default grade of 0. After the grades are posted, you will be given 1 week to rectify your error(s) and resubmit the assignment. You will be subject to a 25% penalty on the grade obtained from the resulting resubmission of the assignment.

How to test your program natively (run your own test cases):
You can use main.cpp provided in the starter.zip folder to instantiate ChainingHashTable, ProbingHashTable, and DoubleHashTable objects. You can then test the functionality of your methods by calling these methods and checking whether what they return is what you expect.
To compile your own tests in main.cpp, **you can use the makefile provided within starter.zip to compile your program.** In your terminal:

- Run **make** on the terminal (this will create an executable named main)
- Next, run **./main** on your terminal to run your native tests

How to use testInfrastructure:
To check your score using testInfrastructure, you can follow either of the two options mentioned below:

- **Option 1:** Move all your .h files into the testInfrastructure directory. After this, run **python3 test.py**
- **Option 2:** Move all your .h files into the testInfrastructure directory. **Ensure that the makefile in this directory is the one given to you originally in the testInfrastructure.zip folder and not in the starter.zip folder.** In your terminal:
  - Run **make** in order to compile the program (which will compile hash_test.cpp and create an executable named hash_test).
  - Next, run **./hash_test** on your terminal to actually run the tests and output your score.

Please also make sure that you run the test script a few times (3 - 4 times should be sufficient) to ensure that your code is consistently passing test cases in different random runs of the test script.

PA4 - Extensions (Not worth a grade)

For this extension, you will be using a Hash set (std::unordered_set in C++) to solve an algorithmic challenge. You can think of a hash set as essentially a map without any values (i.e. no duplicate keys exist in a hash set).

Algorithmic Challenge:

Given a string, find the length of the longest substring without repeating characters. The problem of finding the longest substring without repeating characters is a classic algorithmic problem that involves efficiently identifying the longest substring within a given string that does not contain any repeated characters.

Example 1:

Input: "abcabcbb"

Output: 3 (The longest substring without repeating characters is "abc", which has a length of 3.)

Example 2:

Input: "bbbbb"

Output: 1 (The longest substring without repeating characters is "b", which has a length of 1.)

Example 3:

Input: "pwwkew"

Output: 3 (The longest substring without repeating characters is "kew", which has a length of 3. “pwke” is a subsequence and not a substring)

Example 4:

Input: “babbacac”

Output: 3 (The longest substring without repeating characters is "bac", which has a length of 3.)

Example 5:

Input: “cdfcdfab”

Output: 5 (The longest substring without repeating characters is "cdfab", which has a length of 5.)

Hint: It is possible to generate all possible substrings and check each substring. However, this is a brute force algorithm. Can we use the unordered_set to do better on runtime?

You can code out the solution on a separate file and use some of these sample test cases on your code. No starter file will be given for this extension. You can additionally use the C++ STL libraries to develop your code.