Timothy Liu

Dis 1A

**Overview**

In my implementation, I use a vector containing lists of strings. So each "bucket" is just a list containing all of a word's anagrams. I chose a vector as my data structure because it had a dynamic size, random memory access, and range checking when accessing elements. All of which make it quick to find anagrams and adds extra bounds checking.

First, each word in the dictionary is transformed into a sorted order by rearranging all its characters alphabetically. So the word "team" and "mate" both become "aemt" after sorting. I used the STL sort function in C++ to sort my strings' characters alphabetically. I then applied the STL hash function on the sorted letters to get the hash value, and used the modulo operator on the returned unsigned int to ensure the hash value is within the container.

This sorted combination of characters serves as the hash key for the string and the index into the correct list in the vector. Then, I use the hash key to go to the corresponding list with all its anagrams. Each list contains all the valid anagrams of a particular sequence of sorted characters. So the hash key "aemt" would point to a linked list that stores all the valid words that can be formed from "aemt." For every sorted sequence of characters, a bucket will be created. If the word trying to be inserted has other anagrams already inserted, it would be inserted at the end of the linked list. If the word has no anagrams, it would be in a list with only 1 element.

To find anagrams for a given word, I remove all non-letter characters and convert it into its sorted form. Then, I use this sorted form as the hash key to access the associated list from the unordered map. Then I traverse through the entire linked list and return all the words that represent the possible valid combinations of the sorted characters.

**Implementations**

**void DictionaryImpl::insert(std::string word)**

| Format the inputted string (word) so that it's a recognizable string (remove numbers, spaces, convert to lowercase etc.);  If the inputted string (word) is a valid word:  Sort formatted string so all characters appear in alphabetical order;  Apply hash function to ordered string get index  Access list at corresponding index in vector  Insert original word into end of the list |
| --- |

**void DictionaryImpl::lookup(std::string letters, void callback(std::string)) const**

| Check if valid word and/or string  Format the string so that it's a proper word (remove numbers, spaces, convert to lowercase etc.);  If word is empty:  Return  Sort string so all characters appear in alphabetical order;  Apply hash function on sorted string to get vector index  Access the list in the vector at the found index  Iterate through the found list start to end  If current word in list contains the exact same characters as inputted word:  Output that string (anagram word) in the list |
| --- |

**Difficulties**

When I ran the g32 tester, it would return a word that wasn't an anagram of the inputted word. To fix this, I changed the conditions needed to callback an anagram. Originally, I just iterated through the list and outputted every string in that list. However, I sorted the current word in the list and compared it to whether it was the same as the sorted word that was inputted and it would only callback if they were the same word. This removed any false positives.

I also struggled with traversing through my list of strings. There were lots of tricky and specific syntax I had to use. I realized that I needed to use a const\_iterator instead of a normal iterator because I wasn't able to modify anything inside my list.

**Test Cases**

**Words**

| dog | 3 Letter word |
| --- | --- |
| oceanography | Long word with no anagrams |
| abcdefg | No valid words |
| angle | Multiple anagrams |
| pears | Multiple anagrams |
| permutation | Long word with 1 anagram |
| D0000…000OG  (dog with 650 5s inbetween) | Multiple filler characters |
| AnGlE! | Alternating capitalization, filler character |

**XCode**

|  | Original | Mine |
| --- | --- | --- |
| dog | ~ 9ms | ~ 0.003ms |
| oceanography | > 30s | ~ 0.005ms |
| abcdefg | > 5s | ~ 0.002ms |
| angle | ~ 130ms | ~ 0.003ms |
| AnGlE! | ~ 130ms | ~ 0.004ms |
| pears | ~ 130ms | ~ 0.004ms |
| permutation | > 30s | ~ 0.004ms |
| d555…555og  (dog with 650 5s inbetween) | ~ 9ms | ~ 0.100ms |

**g32 Fast**

|  | Original | Mine |
| --- | --- | --- |
| dog | ~ 0.001ms | ~ 0.0005ms |
| oceanography | ~ 2.2s | ~ 0.0007ms |
| abcdefg | ~ 1.8ms | ~ 0.0004ms |
| angle | ~ 0.005ms | ~ 0.0009ms |
| AnGlE! | ~ 0.005ms | ~ 0.0011ms |
| pears | ~ 0.005ms | ~ 0.0007ms |
| permutation | ~ 440ms | ~ 0.0011ms |
| d555…555og  (dog with 650 5s inbetween) | ~ 0.002 | ~ 0.0008ms |