**Overview**

In my implementation, I use an unordered map with strings as keys and lists of strings as values. So the strings map to a list of other strings.

I chose an unordered map as my data structure because it would automatically handle collisions for me, it had random memory access, and I could automatically "link" a list of anagrams to each hash value since a map contains a pair of data types. Also, the hash function implementation was relatively straightforward and intuitive because I could just rearrange the letters in a given word.

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. This sorted combination of characters serves as the hash key for the string. I used the STL sort function in C++ to sort my strings' characters alphabetically.

Then, I use the hash key to look up the corresponding list in the unordered map. 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;  Assign sorted string to a bucket, which is defined by its sorted sequence of characters  Add it to the list of possible word anagrams contained in that bucket |
| --- |

**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;  Find bucket with the same key as the string of sorted letters in the unordered map;  If you found the associated list:  Iterate through the entire list (containing strings)  Callback the string (word) for each element |
| --- |

**Difficulties**

I had an inefficiency where I originally considered using a hash function that would convert my sorted string into an int. However, I realized that because I was using an unordered map, I could just specify what I wanted my pair values to be. This meant that I didn't have to use an int as my key to access my buckets; I could've just used my sorted string. This removed an unnecessary step in both my insert and lookup implementation.

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. I also needed to include a check for whether or not I was even at a valid bucket before trying to access a list otherwise I would've tried to access a map key that didn't exist or was out of bounds.