Eideen Mozaffari

UID : 105988436

Project 4 Report

**Design**

For my design of the Dictionary class, I chose to implement a hash table. I implemented the hash table using a vector of lists of lists of strings (i.e. vector<list<list<string>>>), as shown in the following diagram:



Each bucket in the vector contains a list of lists of anagrams. To account for collisions, implementing a list of lists allows us to group anagrams in sub-lists, saving time when we perform searches. Since each inner list (sub-list) should only contain anagrams of each other, it allows us to perform one comparison between the user inputted word and the first word of each inner list to select out the correct list of anagrams, after arriving at the bucket.

When inserting a word into the hash table, we first alphabetically order the word (sort alphabetically) to find its bucket. If an anagram of that word already exists in one of the sub-lists at that bucket, we simply add our word to the end of the sub-list. Otherwise, we must create a new sub-list at that bucket and insert two items: an alphabetically ordered version of the word and the word itself. The reason for inserting an alphabetically ordered version of the word is so that when we perform a search, we can simply compare the alphabetically ordered version of the user’s input with the first entry in each sub-list (I’ll explain this in more detail now).

When performing a search given a user’s input, we first alphabetically order the input to find its bucket. Once we are at the right bucket, we compare the alphabetically ordered input with the first entry in each sub-list (notice the first entry in each sub-list will contain an alphabetically ordered version of the anagrams in that sub-list due to our insertion choice). If we find a sub-list containing anagrams of the input, we then simply loop through that sub-list performing some callback function, starting at the second entry in the sub-list (notice the first entry contains an alphabetically ordered version of the anagram, so every successive entry will be the correct anagrams to operate on) and return to the caller. On the other hand, if we have iterated through the entire outer list, visiting the first entry in each sub-list, and cannot find a sub-list containing anagrams of the input, we know the input is not in the Dictionary (hash-table).

**Pseudocode for Insert:**

Dictionary::Insert(word)

removeNonLetters(word)

if word is emtpy

return

otherwise

sorted\_word = sort(word)

key = hash(sorted\_word) mod number of buckets

for each sub-list in the bucket's list

if sub-lists's first entry == sorted\_word

add word to this sub-list

return

if no existing sub-list contains an anagram of the word

add a new sublist

add sorted\_word to the sub-list

add the word to the sub-list

**Pseudocode for Lookup:**

Dictionary:: Lookup(word, callback function)

if callback is nullptr

return

removeNonLetters(word)

if word is empty

return

otherwise

sorted\_word = sort(word)

key = hash(sorted\_word) mod buckets

for each sub-list in the bucket's list

if sub-list's first entry == sorted\_word

for each anagram in the sub-list starting at the second entry

callback(anagram)

return

**Notes on Bugs, Inefficiencies**

I had initially tried to implement my own *sort* function with linear time complexity, but I don’t think I wrote it properly because it was slower than the STL’s O(N log N) sort, which shouldn’t be possible if I had actually implemented it correctly. So, I ultimately just ended up using STL’s sort to alphabetize.

Also, when running my program using the efficiency tester, 90% of the time my speed result was around 8ms, but 10% of the time it spiked to 17ms. I tried figuring out what was causing the spikes, and I believe it could have resulted from my laptop’s internal bandwidth or overflow on the CS32 seasnet Linux server.