**CS32 Project 4 Report**

**Ben He**

**UID: 804962948**

Algorithms & Data Structures

The data structure that I decided to use to store the dictionary words is a member open hash table in the DictionaryImpl class. The hash function I use calculates where to place a given word by assigning each letter of the alphabet to a certain prime number then multiplying the letters’ assigned numbers together in a word. This takes advantage of the fact that the product of any unique combination of prime numbers is also completely unique. I also purposefully make it so that the hash of a string depends only on the letters it holds and not their order in the string. However, I take the modulo of this product outside of the function because I use the return value of my hash function for two purposes: to decide where to place the word in my hash table and to identify an anagram.

Each Node structure in my open hash table holds three data members: the ID, the word, and a pointer to the next Node in the list. The ID holds the prime product of the word that’s calculated using my hash function (not modulo’d), which is later used by the lookup function, and the word is the actual word that was inserted into the table and is the “value” of the Node.

In my DictionaryImpl’s insert(string word) function, I call the provided removeNonLetters function on the word, then I put the passed word through my hash function then I take the product of my hash function modulo 49,999 (which was what I chose for the size of my hash table as it’s the largest prime number less than or equal to the spec’s limit of 50,000 to maximize search speed). I then look at the bucket in the DictionaryImpl’s hash table that corresponds with the modulo’d value and place a Node there containing information about the word accordingly.

Then, for the lookup(string letters, void callback(string)) function, I call the removeNonLetters function again on letters, then I run letters through my hash function to get letters’s prime product then take modulo 49,999 to locate the word’s bucket. Then I search through the linked list and call the callback function on the word of every Node that has an ID matching the hashed value of letters because that means that they have the same characters as letters and are thus anagrams of letters. Using this strategy eliminates the need for the generateNextPermutation function that’s provided in the original code and speeds up searching.

The reason I chose to use a hash table instead of the STL list that was in the original algorithm is because searching for elements in a hash table is significantly faster than in a linked list, as searching through a hash table without collisions is O(1) over a linked list’s O(N) with N being the linked list’s size. I chose an open hash table over a closed one because it’s generally more efficient, and it is not very difficult to implement.

Pseudocode

DictionaryImpl(){

for every bucket of m\_words:

set pointer in current bucket to nullptr;

}

~DictionaryImpl(){

For every bucket of m\_words:

from first Node of current bucket, while current Node is not null:

set temporary Node pointer to current Node address;

move to next Node;

delete Node pointed to by temporary Node pointer;

}

unsigned long hash(const string& word) const{

initialize value of ID to 1;

for every element of word:

multiply value of ID by prime number assigned to current element;

return value of ID;

}

void insert(string word){

remove all nonletters from word with removeNonLetters function;

if the word is empty:

return;

calculate ID (prime number product) of word using hash function;

determine bucket of hash table for placement using ID % (size of table);

if the bucket at (ID % (size of table)) points to null:

create new Node;

assign Node’s id variable to ID;

assign Node’s word variable to word (which was passed into function parameter);

assign Node’s next pointer to null;

assign bucket to address of this Node;

else:

while current Node is not at the end of bucket’s linked list:

move to next Node in the linked list;

create new Node;

assign Node’s id variable to ID;

assign Node’s word variable to word (which was passed into function parameter);

assign Node’s next pointer to null;

assign current Node’s next pointer to address of the newly created Node;

}

void lookup(string letters, void callback(string)) const{

if no callback function provided:

return;

remove all nonletters from letters with removeNonLetters function;

if letters is empty string:

return;

calculate ID (prime number product) of letters using hash function;

at m\_words[ID % (size of hash table)], while the current Node in the linked list is not null:

if the ID of letters is the same as the ID of the current Node:

callback(current Node’s word);

move to next Node in linked list;

}

//the rest of the algorithms were provided in the original code

Known Bugs/Inefficiencies/Problems

As far as I know, there are no bugs, problems, or serious inefficiencies with my algorithms as I ran my code through the correctness and speed tester provided on g32 and it completed the complete correctness test with a times below 10 ms. My code assumes that the letters passed to it are all lowercase, so I’m not sure it would work with uppercase, but the original solution also seemed to assume that, so I kept a upper-to-lower case converter call out to maximize speed, but it could easily be checked by calling the tolower function on every element of s at the end of the removeNonLetters function.