**Description of Algorithms and Data Structures**

HashNode Class

For the implementation of the hash table, I first had to make a class for the HashNode. The Hash Node contains the members key (int), value (string), and next (HashNode\*). The implementation is similar to that of a linked list, but the node has a key-value pair.

HashMap Class

The HashMap class makes use of the HashNode class. The HashTable is an array of pointers to Hash Nodes. A private member variance that is a pointer to a HashNode pointer would point to the HashTable. The values in the constructor are used to determine the size of the HashTable array, and I would keep a private data member known as maxsBuckets to records the number of total buckets in the hashtable. The destructor of the HashMap would first destroy the linked list within the bucket, and then would go to the next bucket and repeat until all the buckets and linked lists have been destroyed.

In order to take into account for every value in the bucket, I would have another function known as return\_p that would store a vector of the all the values within the bucket a value exists in. This ensures that I am taking into account every item within the bucket in order to avoid my function not catching repeats.

My strategy for implementing the insert function and permutations would be to pre-sort the word to determine which bucket it would go in. In order to do this, I had to sort the string within the hash function before hashing the value itself. This made it so that the values would be approximately in their own buckets. However, I noticed that unwanted collisions would still occur. For instance, “importunate” and “beechwood” would collide to the same bucket even though they were not anagrams for each other. After some time thinking about a solution, I realized that I could just implement an if statement within the lookup function in DictionaryImpl to check if the word was actually an anagram. This made it so that the function would not call the callback function on words that are not permutations.

Psuedocode

~HashMap()

for every bucket

entry = current bucket

while entry != nullptr

delete every item within the list

set the current bucket = nullptr

delete[] table

//Returning vector of approximate permutations

void return\_p(const string& value, vector<string>& p)

removeNonLetters

hashValue = HashFunction(value);

HashNode\* entry = appropriate bucket

while (entry != nullptr)

push the value of the entry on to the vector

go next

void put(const string& value)

removeNonLetters

unsigned long hashVluae = HashFunction(sort\_value);

HashNode\* prev = nullptr

HashNode\* entry = bucket of value

while(entry != nullptr)

loop to the end of the list

Create HashNode

if (prev == nullptr)

set the table pointer to entry

else

set prev-> next to point to entry

unsigned int HashMap::HashFunction(string s) const

string sorted\_string = s;

sort string

FNV-1 hashing method

h = h % maxBuckets;

return h;

void DictionaryImpl::lookup(string letters, void callback(string))

if (callback == nullptr)

do nothing

removeNonLetters

if (letters.empty())

do nothing

string permutations = letters;

sort permutations

vector<stirng> p\_vector

Call retunr\_p on p\_vector

for every item in the vector

string temp = current element being examined in vector

sort temp

if temp = permutation

callback(current element being examined in vector)

Notable Obstacles

A notable obstacle that I overcame was figuring out how to improve efficiency. When I first approached the problem, I first implemented a regular HashTable and inserted the strings into the hashtable normally. I would then use the getPermtuation function to compare the values within the hashtable. Although this made a fast implementation for words with a small number of characters, this was highly efficient for larger words. This was the fact that the getPermuation function would grow at a rate of factorial as the number of characters within the string increased. In order to mitigate this, I had to redo my hashtable and hash function to pre-sort the data and check whether the values within the buckets were actually anagrams of the word when the lookup function gets called