1. Known Bugs or Problems

One possible bug or inefficiency that I have is in my wordList.cpp in the loadWordList function which processes the entire wordlist.txt multiple times, thus causing the program to run very, very slowly. So for very large word files, my program will be very inefficient or slow.

I did not have enough time to implement the entire Decrypter class, so I simply had it check for return cases without actually decrypting.

2. Description

For MyHash.h

* Data members: m\_items to keep track of # of items in hashtable, m\_buckets to keep track of # of buckets in hashtable, m\_maxLF to keep track of max loading factor, and Node\*\* hashTable as hashTable is a pointer to Node pointers because we will be having hashTable point to a dynamically allocated array of Node pointers. Node structure with m\_key, m\_value, and a Node\* next
* MyHash(double maxLoadFactor = 0.5)

Check if the value that user passes in for maxLoadFactor

Set the m\_maxLF based on user input

Allocate an array of Node\* with 100 initial buckets

Loop through each bucket and set each bucket to nullptr

* ~MyHash()

Loop through each bucket

Create a temp Node\* to point to the Node in each bucket

While the temp pointer is not null

Create another Node\* to point this Node’s next

Delete this node and set the temp pointer to next

Delete the hashTable using [ ]

* void reset()

Loop through each bucket and delete each Node similar to destructor above

Delete the hashTable using [ ]

Allocate a new empty hash table and reset its m\_bucket to 100 and m\_items to 0

Loop through each bucket and set the pointer to nullptr

* void associate(const KeyType& key, const ValueType& value)

Generate a bucketNum using hash function

If the bucket is empty

simply allocate a new node into it and increment m\_items

Otherwise

Search through the bucket to find a matching key or the last node

If we did not find an existing key, create new node and add it to the end

Otherwise, simply update the Node’s value

Check if we exceeded our maxLoadFactor

Allocate a new hash table with double the buckets

Move old hashTable’s items to new hashTable

Delete the old hashTable

* const ValueType\* find(const KeyType& key)

Hash into a specific bucket number

If the bucket is empty, return nullptr

Otherwise, search through the bucket’s linked list of Nodes

Return nullptr if not found, or return address of the node’s value

For Tokenizer.cpp

* Data members: vector of characters to hold each separator character

vector<char> m\_separators

* TokenizerImpl(string separators)

Push each separator character into m\_separators

* vector<string> tokenize(const string& s) const

Create a vector to hold each token and a string to build each token

Loop through every character of the string

If the string char is not a separator, build onto the temp string

If we run into a separator, push the temp string into the vector

Reset the temp string to empty

Push the remaining temp if it is not a separator and return the vector of tokens

For WordList.cpp

* Data members: MyHash<string, vector<string>> m\_hashTable

Created a hashTable of strings to indicate pattern, and a vector of strings to indicate the

words with that pattern

* bool loadWordList(string filename)

Reset the hashtable to discard all of its old contents

Create an ifstream to load the text file

Loop through every word in the text file

Generate a word pattern for each word

Ignore a word that contains non-letters or apostrophes

Insert all unique patterns into a vector

For each pattern, create a vector of words with matching patterns

Insert each pattern with all the words matching its pattern into m\_hashTable

* bool contains(string word)

Generate a word pattern for word

Find the vector containing matching patterns of the word

If we did not find a vector, return false

Otherwise loop through the vector to find our word

Return true if found

Otherwise return false

* vector<string> findCandidates(string cipherWord, string currTranslation) const

Check if cipherword and currTranslation are valid, return empty vector if not

Generate word pattern for cipher word and find vector of matching pattern words

For each words with a matching pattern as cipherword

Check if all chars are matching according to letters, ?, or apostrophes

Push all the matches into a vector

Return that vector when finished

* String generateWordPattern(string word) const

Find all unique letters and apostrophes in the word and append to a temp string

Using each unique character, encode the word

Return the word

For Translator.cpp

* Data Members: MyHash<char, char> charMap as the mapping table

vector<MyHash<char, char>\*> mapStack as the stack of mappings

* TranslatorImpl()

Initialize our charMap with ciphertext of alphabet letters and plaintexts of ?

* bool pushMapping(string ciphertext, string plaintext)

Check that ciphertext and plaintext are valid and consistent

Return false if not valid/consistent

Save the current map into mapStack by creating a pointer and allocating a new mapping table

Push this new map into the mapStack

Update the current mapping table to incorporate the new mappings

Return true when completed

* bool popMapping()

If there are no maps, return false

Create a pointer to a mapping table

Point it at the most recently added mapping table in mapStacks

Make the most recent map from stack to the current map mapChar

Delete the map from the stack and pop it

Return true when done

* string getTranslation(const string& ciphertext) const

Create a temporary string and set it equal to ciphertext

Loop through each character of temp string

If the character in ciphertext is a letter

Find the plaintext translation of that letter

Set temp’s char to the plaintext char

Return temp when finished

3.

For MyHash

* MyHash() satisfies the big O requirement of O(B).
* ~MyHash() satisfies the big O requirement of O(B)
* Void reset() satisfies the big O requirement of O(B).
* Void associate() satisfies the big O requirement of O(1)
* const ValueType\* find() satisfies the big O requirement of O(1)
* int getNumItems() satisfies the big O requirement of O(1)
* double getLoadFactor() satisfies the big O requirement of O(1)

For Tokenizer

* TokenizerImpl() satisfies the big O requirement of O(P)
* vector<string> tokenize() satisfies the big O requirement of O(SP).

For WordList:

* bool loadWordList() does not satisfy the big O requirement of O(W) because for each word, it generates a pattern and pushes that pattern into a vector, in which I loop through the vector of size P to check for matching pattern words, thus my time complexity is about O(WP)
* Bool contains() satisfies the big O requirement of O(1)
* vector<string> findCandidates() satisfies the big O requirement of O(Q).

For Translator:

* bool pushMapping() does not satisfy the big O requirement of O(N + L) because my algorithm goes through each character of ciphertext and within that loop, it also loops through each letter of the alphabet, thus my big O is O(NL).
* bool popMapping() satisfies the big O requirement of O(L)
* string getTranslation()