1. The codes are working as intended. No memory leaks and no known bugs found.

2.

**MyHash class**

MyHash was build on a dynamic array, which has pointer to a linked list as elements.

Every new item will be hashed into the appropriate linked list and inserted from the front of the linked list.

If max load factor is exceed, it will rehashed into the appropriate linked list in the new hash table.

* Associate function

If key found

Edit value

Add node

Find appropriate bucket

Insert new node into the bucket’s linked list from the head.

If > maxLoadFactor

Rehash

Create temporary hash table

Visit each bucket in old hash table

//Rehashing doesn’t have to check if key is not unique

//Algorithms for inserting node into old hashtable already ensure they’re

unique.

Traverse through linked list in each bucket

For every item, find new hash and insert in the appropriate bucket of

temporary hash table

Delete old hash table

Re-assign hash table variable with temporary hash table

Double current bucket number

**Tokenizer class**

* Tokenize function

**Translator class**

Translator is created using a struct of two arrays: one array maps from cipher to value, and the other value to cipher. The two array would be consistent at all times, thus achieving a one-to-one mapping essentially. Together, this is one map.

Mapping will be stored in uppercase to ensure case-insensitivity.

To keep track of previous mapping, it’ll stored in a vector. Pushing and popping whenever appropriate. This is the stack.

-PushMapping function

If inputted cipher and plain are of different lengths

return false

Create temporary map

Looping through each letter of the cipher and plain

Capitalize plain and cipher letter

Convert plain and cipher letter to an int (0-25)— plain and cipher index.

Access plainToCipher array and cipherToPlain array with their respective indexes.

If both ‘?’

Update both arrays at the appropriate places

else

if consistent

continue checking other letters

If inconsistent

return false.

Reassign map variable to temp map

Push new map to stack.

Return true.

* GetTranslation function

Make an empty string.

Loop thru each letter of inputted cipher

If letter is not an alphabet

add to string

continue to next letter

if cipher is lower case

access the cipherToPlain array and give plain letter mapping in lower case

else

access the cipherToPlain array and give plain letter mapping

return string

**WordList Class**

Wordlist class is build from two myHashes. One myHash was keep track of worded that have been added(a string to bool mapping). The other myHash was used to map a word pattern to a vector of words from the wordlist that match that pattern.

* GetStructure function (abstract out the pattern of a given word)

Create an empty string

Current symbol (uppercase letter, starting from ‘A’)

Substantiate an array of size 26 (this will be used to keep track of what has been added)

Loop thru each letter of the word

If letter is \

add \ to string

continue to next letter

Convert upper-cased letter into an int (0 to 25). This is the letter’s index

Use that index to access the array

If array returns a nullptr

add current symbol into that array

increment current symbol (now it’s ‘B’ etc..)

Add the appropriate symbol from the array into the string

return string

* LoadWordList function

Open file

If can open

Read every single line of that file

Check every letter of that line

If any invalid character

continue to next word

Get structure of word, making sure to capitalize each letter and

preserving the ‘. Code is similar to getStructure function

Using the structure as key, push word into the vector of words with

matching pattern.

Using the word as key, push word with a true into the second myHash.

return true

Else

return false

* FindCandidates function

Capitalize every alphabet of both cipher and translation word.

Get structure of cipher

Create a vector to contain possible candidates

If can’t find pattern

return empty vector

If size of cipher and translation is different

return empty vector

Iterate through vector of possible candidates

Loop thru each letter of the candidate word

If letter in translation doesn’t match with candidate word, while letter in

translation isn’t a ‘?’

Go to next candidate

If all match

Push candidate to vector

return vector

**Decrypter class**

Decrypter will have a wordList, tokenizer and translator object. It’ll process a given cipher word with help of these classes.

* Recursion function (arguments: 1. Output vector to store final answers, passed by reference

2. Vector to store usedToken, passed by value (so, it’s only valid across recursive calls)

3. Cipher text to be decrypted

4. Translator object, passed by reference

Pseudo-code:

Tokenize cipher

Order tokenized cipher by most untranslated tokens

Remove used tokens from ordered token

Get the word to be processed, which is the first element of the vector

Add word to be processed into usedToken vector

Find candidate for word to be processed

If candidate vector is empty

popMapping

return to previous recursive call

Loop thru each candidates

if pushMapping failed

continue to next candidate

Translate the cipher again with new mapping

Tokenize new translated cipher

Create a vector for words that are translated fully

(push, if word doesn’t have a ‘?’)

Create a vector for word that haven’t been translated fully

(push, if word contains at least one ‘?’)

Loop thru each word in the tokenized new translation

If word is fully translated

Push to fully translated vector

else

Push to the other vector

If some words are fully translated (vector for fully translated is NOT empty)

if not all fully translated found in wordList

popMapping

return to previous recursion

else

if all words are fully translated (the vector containing

untranslated is empty)

push new translation into the output vector

popMapping

go to next candidate

else

recurse to build on the new mapping

Popping mapping after all candidates have been checked

Return to previous recursion

3.

MyHash class

- Constructor and destructor are both running at O(B) time, where B = number of buckets.

- Reset function is running at O(B) time, where B = number of buckets.

- Associate function runs as fast as required by spec

When bucket number doesn’t change:

Function works as fast as spec. It first calls find function, which is

effectively O(1), as it only have to traverse the linked list — a fraction

of the number of items. If key isn’t found, the new node is inserted from

the front of the appropriate bucket’s linked list. So, effectively O(1).

When bucket number change:

The function runs in O(x2B) time, where X = number of items in the

hash table & B = number of bucket. The function loop thru each item in the old hash table and rehash it in the appropriate bucket of the new hash table. Since B>>x, the function effectively runs on O(B) time.

- Find function runs as fast as the spec. O(1) find and, at worst, O(X), if items are hashed

in only a few buckets.

- getNumItems and getLoadFactor also words as fast as a spec.

Tokenizer class

- Constructor works in O(1) time. Function just assign separators with a private member

variable.

- Tokenize function works in O(SP)— as fast as required by spec— time, where S =

number of letter in a string and P = number of separators).

WordList class

- loadWordList function runs as fast as speed required in the spec of O(W) time, where W

= number of words. Function loops through every word and checked every single letter of that word, to find its underlying structure and push to their respective hash table. The whole process runs in O(nW), where n = number of letters in a word. Since W >> n,

effectively the function runs in O(W).

- contains function calls on myHash’s find function, which runs on O(1). So, it runs as fast as the spec required.

- findCandidate function runs on O(nQ + 2l). The function first makes the cipher text

case-insensitive, then finds the structure of the cipher text O(2l). Then, it loop thru each

found candidates, and checking each letter of the candidate if they’re consistent with the

current translation. Since Q >>n & Q >> l, the function runs on effectively O(Q) time, which

is what the spec requires.

Translator class

- pushMapping runs at O(1), as required by the spec. The function was implemented using

the 1-1 mapping defined in the translator pseudo-code. It loops through each letter in

the cipher text. Since it’s letter in cipher <= number of alphabet and alphabet is a

constant, thus it runs on O(1).

- popMapping just pops out the previous map from the vector. It’s O(1) time.

- getTranslation works on O(W), where W = length of the cipher text. No time constraint.