1. I had a problem in MyHash function where there was an access error if I tried to look at a key in an empty bucket. This wouldn’t allow me to run the sanity checker for MyHash and Wordlist because it resulted in a SEGV error. However, I think WordList (with the use of a correct MyHash function) works correctly.

Also, I believe the general implementation of Decrypter was correct, but it was difficult to find a way to try not to translate the already translated solutions.

1. Class MyHash {

I found my find function to be nontrivial. What I did was I went to the designated bucket via the key (which of course required scaling it to the number of buckets). Instead of immediately checking to see if the value was there, it was important to remember that it is a table of linked lists, not single items; therefore, if the node was not NULL, we chose to iterate through the function. When we found the value, returned the address.

I found my associate function to be nontrivial. We had to deal with two cases: if it was in the hash table or if it was not, which required updating the existing relationship or inserting a new Node. For this reason, there was an if conditional statement after looking up the key.

I found my resizing function to be nontrivial. What I essentially ended up doing is taking a temporary reference to the existing bucket (with too large of load factor). After this was noted, I was able to set the existing bucket to a newly allocated hash table. After this, I had to transfer from hash to hash: this consisted of looping through the temporarily referenced hash table and transferring values where there were values in the hash table.

Data Structures:

In implementing my hash, I made sure to use the hash table (what it was essentially was an array of linked lists). I decide to use this not only because it was a requirement, but also we need to have a general table for storing data (and if you wanted to alphabetize things in this table, you could do that with an implementation of your own.

}

class WordList {

I found findCandidates to be nontrivial.

I started by making a pattern for the word passed in (which of course should be lowercase, otherwise you’ll get a different pattern!). I then had a helper function check if the pattern matched up with the pattern of the potential English word. If it did we pushed it into the vector. If not, we continued until we ran out of words to look at.

In implementing Wordlist, I made sure to use a hash table(string to a vector string). I used this because we did not need the time complexity of ordering as these words were organized by key (not alphabetical order. We were also able to retrieve items quickly in approximately O(1) steps versus things like an array or a list, etc.

}

class Tokenizer {

My implementation of tokenize was not completely trivial, the idea was an easy one, but I went about tokenizing in a unique way; initially, I created a placeholder (to take the position in the string). What I did was I looked through the word once. For every letter, I checked to see if it was one of the separators. If it was not, we were able to move on. If not, we took the substring from start to that point(in the beginning our placeholder started at position 0). We set the new start position as the current pos and continued with this process until you reached the end of the string.

For the majority of my implementation in Tokenizer, I felt comfortable with using a vector (usually of strings, for the most part). This was fine because something like a hash table would be unnecessary (as we only have one parameter) and at the same time, we never need orderings of our list (so items like a set with a larger time complexity would be absolutely unnecessary).

class Translator {

I found my popMapping function to be nontrivial in some parts. The popping off the back of the vector was easy; the tricky part was the case of if the vector of maps was empty. Essentially what you want to happen is for the current map to be automatically set to the one mapping with all question marks. So, what I did was ask if the vector was empty first; if it was, I had a map as a private member that was just the question mark mapping, so I set the pointer of the current mapping to this map. If not, we just set it to the second most recent map in the vector (the back of the vector, front of the stack).

For the implementation of each individual mapping, I made sure to use a map. I felt the time complexity cost was worth it because we could iterator through maps; I say this because you could have a character with a start character equal to the letter a and increment this character based on ASCII properties.

For the implementation of the mappings, I made sure to use a vector (which we could essentially push and pop into) It allowed us to have the stack that we were aiming for because we could look at the back value of a vector (which would essentially be the top of a stack). In addition, we could pop from the back of a vector, which is equivalent to popping from the top of a stack!

}

class Decrypter {

The implementation for crack was definitely nontrivial. The general idea is to pass in an empty vector with this vector eventually returning all possible solutions (most likely out of order) and the ciphertext we were given. This function was a recursive function. With every recursive call, we translated it as far as we could, tokenized these partially translated words, and continued to look for potential candidates for these partially translated words. If we had fully/partially translated words that worked, we moved on to the next word with the current mapping. If the fully translated word was a bad word, we reverted back to the older mapping and continued with a new word. We continued with this until we found solutions (words to crack was equal to the words translated). If you found a solution, you reverted back one map to try different words. This all ended when we ran out of candidates.

For nearly all of my implementation I used vectors to store most of my data (usually strings). Most of the work involved taking a recent version of something and either going with it or popping it off (as seen with the mappings). Also, our eventual result stemmed from a vector (we needed to return a vector of strings from our helper function and sort them and show the results). There was never a moment were I found another structure to be necessary (whether that would be for ordering or something else unnecessary).

}



MyHash(double maxLoadFactor) and ~MyHash() (O(B) where B is number of buckets)

void reset(): Yes (O(B+average size of linked list which is small)) = O(B)

void associate() (O(B))

Roughly O(1) time, you can look up the key (O(1)), if nothing is there insert it, if not iterator through the linked list and look for it or insert on the end.

find (O(1)): Yes (O(1+average size of linked list which is small)) = O(1)

getNumItems: Yes

getLoadFactor: Yes

Tokenzier(“ “):Mine was essentially O(1) because I included my separators as string to iterate through later.

Tokenize (O(SP)): Yes

loadWordList: Yes

contains (O(1)): O(V) Because you had to iterate through the vector string to see if the word was in there.

findCandidates: O(V) Because you had to iterate through the vector string to see if the word was in there. It was not a data structure in which I could use a find function to look up something.

pushMapping (O(N+L): It would’ve been O(1) but I also decided to update the map with values in this function.

popMapping(O(26)): Yes

load (O(W)): Yes (same as WordList).