**CS 32 Project 4**

**Known bugs or problems:**

The program takes a long time to decrypt complex ciphers. However, the decryption is ultimately successful.

**Description of data structures and algorithms:**

**MyHash Class**

Private members:

A Node struct that contains variables of ValueType and KeyType as specified by the template definition. The struct also contains a Node pointer named next. This serves as the basis for a linked list data structure.

We also have a Node pointer to an array of Node pointers called map.

Each element of this array contains a pointer to a linked list with members as specified above.

We also have variables to hold the number of items in the map and the number of buckets (which are essentially the number of array elements).

Algorithm for reset function:

Repeat steps until you reach the last bucket (array element)

For each element, repeat steps until the current element is a nullptr

Set a temporary pointer to the next node in the linked list.

Delete what the current pointer is pointing to and set it to the temporary pointer.

End while loop

End for loop

Delete the entire map structure

Dynamically allocate a new map structure with 100 elements (buckets) and set each element to a nullptr

Algorithm for associate function: (Uses open hash table concept)

Call a helper function to get the index of the bucket that the current key hashes to.

Create a new node with the specified key and value

If the bucket hashed to is empty, insert the new node and increment the number of items.

If the bucket has other nodes associated with it, run through all the nodes in the bucket.

If any node in the bucket has the same key as the key passed to the function, simply update the value associated with that key.

Otherwise, insert the new node at the end of the linked list and increment the number of items.

If the load factor upon insertion exceeds the max load factor, create a new map with double the number of buckets, each initialized to nullptr.

Then rehash every key in the original map and copy key value pairs to their appropriate buckets in the new map.

Run through all the buckets in the original map and delete all the nodes in the linked lists associated with every bucket

Delete the original map and assign the new map to the original map’s pointer

Algorithm for find function:

Call a helper function to get the index of the bucket that the current key hashes to

Create a pointer to the element at that index in the hash map.

Run through all the nodes in the linked list associated with the bucket and if the key is found return the address of the value associated with it.

If the key is not found return nullptr.

**Wordlist Class**

Private members:

Pointer named wordlist to a hash map (Key-string, Value- Vector of strings)

Algorithm for loadWordList:

(Here we use a hash map where keys represent word patterns and the values associated with them are vectors containing all the words with the same word pattern)

Delete the contents of the current wordlist hash map

Allocate a new hash map to the MyHash pointer

Create a file input stream object attached to the given file

If it cannot be created, display error message and return false

Repeat the following steps as you get each line in the file

Run through the line and see if there is any character other than alphabets and an apostrophe. If there is such a character, ignore this line.

If there is no such character, find the vector of all words with the same pattern as the current word in the hash map. Add the current word to that vector and re-associate the vector to the pattern in the hash map. Return true.

Algorithm for contains:

Find the vector containing all the words with the same pattern as the given word.

If such a vector exists,

Run through the vector and compare each word in the vector with the given word to see if they match. This comparison is done character by character taking their lower case in order to do a case-insensitive equality comparison.

If a match exists, return true.

Otherwise return false.

Algorithm for findCandidates:

If the cipher word and the current translation are of different lengths, return an empty vector.

Run through the characters of the cipher word and the current translation. If the cipher word contains any characters other than alphabets or an apostrophe or if the current translation contains any characters other than alphabets or an apostrophe or a question mark, return an empty vector.

Find the vector in the hash map containing all the words with the same pattern as the cipher word.

Create an empty vector to hold the candidate words.

If this vector does exist, return an empty vector.

Otherwise, run through the vector and do the following checks

If the corresponding character in the current translation is an alphabet, the same must be true for the cipher word. If it isn’t, return an empty vector.

The same must also be true for the current word in the vector. This alphabet must also match the alphabet in the current translation. If this is not true, disregard the current word.

If the corresponding character in the current translation is a question mark, the corresponding character in the cipher text must be an alphabet. If it isn’t, return an empty vector.

The same must be true for the current word in the vector. If it isn’t, disregard the current word.

If the corresponding character in the current translation is an apostrophe, the corresponding character in the cipher text must also be an apostrophe. If it isn’t, return an empty vector.

The same must be true for the current word in the vector. If it isn’t, disregard the current word.

If the current word in the vector meets the above conditions, push it into the vector of candidate words.

After checking all the words in the vector, return the vector of candidates.

**Tokenizer Class**

Private members:

A pointer named sep to a hash map(KeyType- char, ValueType- bool)

Algorithm for constructor:

Run through the separators string and associate each separator with the boolean value true in the created hash map.

Algorithm for tokenize:

Create a vector to store the individual words of the string.

Create a copy of the given string and concatenate a space to the end of it. (Deal with single word strings)

Create a string to store the characters of each word till you hit a separator

Run through the characters of the string.

If the current character is found in the hash map containing the separators, push the current word (containing all the characters we have encountered between the last separator and the current separator) into the vector of words. Then change the string to an empty string to store the next series of characters.

Otherwise, simple concatenate the current character to the string.

Once all the characters of the string have been parsed, return the vector of words.

**Class Translator**

**Stack is implemented using a linked list**

Private members:

A Node struct containing a hash table (KeyType-char, ValueType- char), a string named inverted containing the new characters mapped to and a Node pointer.

A Node pointer to the headof the linked list

A Node pointer to the top of the stack (last node of linked list)

A string named mapped containing the characters mapped to

On construction, all lower-case versions of the alphabets are mapped to a question mark. This is done in the head’s hash table.

Algorithm for pushMapping:

A new temporary Node is created and its table is given a copy of all the mappings in the top node of the stack. (Done using associate function and find function)

If the ciphertext and the plaintext are of different lengths, delete the temporary node and return false.

Run through the characters of the ciphertext and plaintext.

If any of the characters is not an alphabet, delete the temporary node and return false.

If the temporary table contains a translation for the current ciphertext character being checked and it isn’t a question mark, delete the temporary node and return false.

Run through the characters in the mapped string

If any of the characters in the mapped string are the same as the current plaintext character we are checking, delete the temporary node and return false.

If the above conditions are met, associate the current ciphertext character with the plaintext character in the temporary node’s table.

Add the plaintext character to the mapped string.

Add the plaintext character to the temporary node’s inverted string.

After going through all the characters in the ciphertext

Add the temporary node to the top of the stack and set top to point to that node.

Return true.

Algorithm for popMapping:

Do nothing if there is only one Node.

Run through the linked list and stop before reaching the topmost node

Delete the topmost node and set it to the node just before it.   
Its next pointer is changed to nullptr.

Algorithm for getTranslation:

Create a string to hold the translation

Run through the characters of the cipherText

If the character is not an alphabet, simply add it to the translation string.

Otherwise,

Find the translation of the character in the table of the topmost node

If that translation is a question mark, add it to the translation string

Otherwise if the ciphertext character is uppercase add the uppercase version of its translation to the translation string

otherwise if the ciphertext character is lowercase add the lowercase version of its translation to the translation string

After running through all the characters of the ciphertext

Return the translation string

**Decrypter Class**

Private members:

A pointer to a wordlist object containing the words in the file provided

A translator object to handle all the translations

A tokenizer object to handle all the tokenizing

A MyHash object (keytype- int, valuetype-bool) to store the positions in the vector of cipher words that have already been used

For the load function, simply delete the current wordlist object, create a new one and ask the object to load the word list

I used a private countQuestionMarks function to count the question marks in a given string. It simply runs a loop through all the characters in the string and uses a counter variable to count the number of question marks.

I used a private chooseEncryptedWord function to choose what word I would want to check given a vector containing the words in the ciphertext.

It also uses a vector of strings to store translations of each word in the cipher string. These translations are obtained using the most current mapping the translator object has.

A loop then runs through the translated words vector and counts the number of question marks in each word using the function above.

The word that hasn’t been used yet and has the greatest number of question marks will be the chosen word.

If you encounter a word with the same number of question marks and it hasn’t been used yet and its length is larger than the length of the current max word, that word will be chosen.

This word is associated with the Boolean value true and added to the hashtable

The word chosen is then returned.

Algorithm for crack is as mentioned in the project spec.

The crack function used string vectors to store the cipherwords and the all the valid translations for the cipher text. It uses a string vector to store the candidates for each encrypted word chosen.

It calls a sort function that uses insertion sort before it returns the vector containing valid translations.

**Big-O requirements:**

**For MyHash functions, linked list traversal has a complexity dependent on the number of items mapped to each bucket. This has been accounted for by using variables like N or X to denote the number of items.**

**MyHash constructor - Satisfies**

Runs in O(B) time

**MyHash destructor - Satisfies**

for(int i=0;i<numBuckets;i++) //O(BN)

{ //O(N)

while(map[i]!=nullptr) //O(N)

{

Node\* temp = map[i]->next; //O(1)

delete map[i]; //O(1)

map[i] = temp; //O(1)

}

}

delete [] map; //O(1)

In its worst case, if all the items hash to a few buckets, it runs in O(BN) time where N is the number of items in each bucket.

At best, it runs in O(B) time

**MyHash reset() - Satisfies**

for(int i=0;i<numBuckets;i++) //O(BN)

{ //O(N)

while(map[i]!=nullptr) //O(N)

{

Node\* temp = map[i]->next; //O(1)

delete map[i]; //O(1)

map[i] = temp; //O(1)

}

}

delete [] map; //O(1)

map = new Node\*[100]; //O(1)

numBuckets=100; //O(1)

for(int i=0;i<numBuckets;i++) //O(B)

map[i]=nullptr; //O(1)

items=0; //O(1)

In its worst case, if all the items hash to a few buckets, it runs in O(BN) time where N is the number of items in each bucket.

At best, it runs in O(B) time

**MyHash associate - Satisifes**

In its worst case, if all the items hash to a few buckets, it runs in O(X) time when X is the number of items in each bucket.

At best, it runs in O(1) time and O(B) time when number of buckets change.

**MyHash find - Satisfies**

At best, it runs in O(1) time

In its worst case, when most keys hash to just a few buckets, it runs in O(X) time where X is the number of buckets

**MyHash getNumItems - Satisfies**

Runs in O(1) time

**MyHash getLoadFactor - Satisfies**

Runs in O(1) time

**Tokenizer constructor - Satisfies**

Runs in O(P) time where P is the number of separators

Time complexity varies when associate has its worst-case O(X) or when number of buckets change, O(B) complexity.

In those cases it runs in O(PX) and O(PB) time

**Tokenizer tokenize() - Satisfies**

Runs in O(S) time where S is the number of characters in the input string.

In the find functions worst case of O(P) where P is the number of separators, tokenize runs in O(SP) time.

**WordList loadWordList - Satisfies**

Runs in O(W) time where W is the number of unique words in the file.

**WordList contains - Satisfies**

Runs in O(1) time assuming all words have a constant length

**WordList findCandidates**

Runs in O(Q) time where Q is the number of words in the wordlist that match the pattern of cipherword. We are assuming all words have a constant length.

**Translator pushMapping – Not satisfied**

Runs in O(NM + L) where N is the length of the parameter strings, L is the number of alphabets and M is the number of characters already mapped to.

I run a for loop through the characters already mapped to (stored in a string) in the current translator mapping scheme and check if any of the plaintext characters match that. That would mean some other character already maps to that plaintext character and the mapping is therefore invalid.

**Translator popMapping – Not satisfied**

It runs in O(N+B) time where N is the number of mappings that have been pushed to the translator object and B is the number of buckets in the top Node’s hash table.

A linked list is used to implement the translator object’s stack like behavior. Each node contains a MyHash object.

**Decrypter load - Satisfies**

Runs in O(W) time where W is the number of unique words in the file.