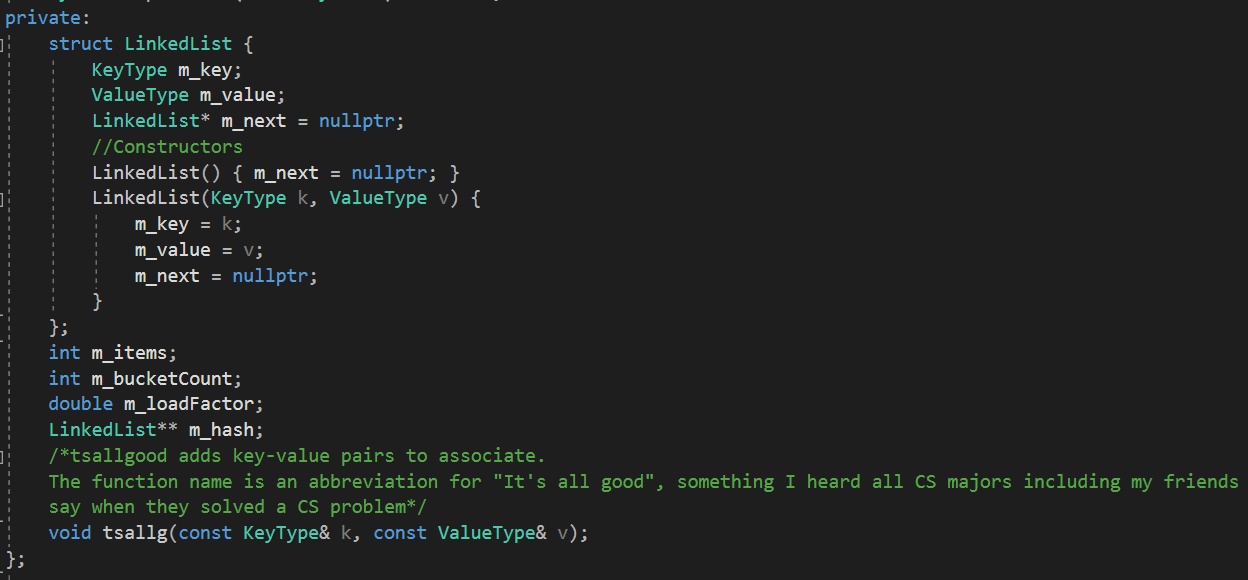
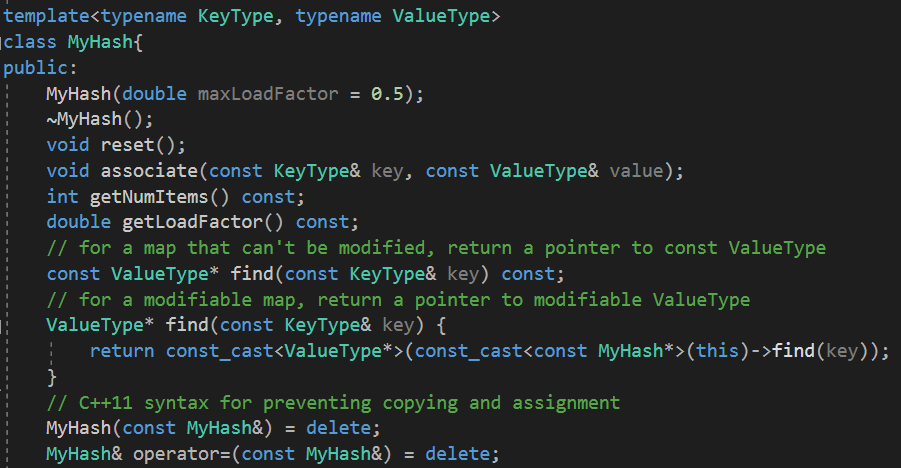
**Project 4 Report**

My classes have no known bugs or memory leaks. This was confirmed after running it through the sanity check posted on the class website using g32. The Big-O for each function described in the spec has been met, and this is confirmed by the program’s incredibly quick decryption/run time for all the encrypted messages provided throughout the spec, and towards the end of the spec for Project 4.

**MYHASH.H**



The struct called LinkedList keep track of key-value pairs, and contains a pointer to LinkedList\* object for traversal purposes. The two constructors allow for quick constructor of LinkedList\* objects in our Hash-Table and LinkedLists in each bucket of the hash table where needed. m\_items keeps track of the number of items inserted into our hash table. m\_bucketCount keeps track of the number of buckets in our hash table. m\_loadFactor keeps track of the load factor at run time, which helps in resizing purposes in the associate function. The LinkedList\*\* m\_hash is a pointer to a LinkedList pointers. This is initialized in the myHash constructor. Tsallg(k,v) is a helper function that is used in associate to add items to our hash table at the required bucket decided by our hash functions.

MyHash<KeyType,ValueType>::MyHash(double maxLoadFactor) {

//If negative or 0, maxLoadFactor = 0.5

//if greater than 2, then maxLoadFactor = 2.0

//Else update m\_loadFactor to the value passed in

//Initializing number of items to zero

//Initialize number of buckets to 100

//Initialize hash table with 100 buckets and set each object in the bucket to nullptr

}

MyHash<KeyType, ValueType>::~MyHash() {

//Traverse through all the buckets in the hash table

//Create a pointer p to the first LinkedList object in m\_hash

//if p is a nullptr, do nothing

//Otherwise Traverse through the linked list and delete all the nodes at that bucket

//Delete the last node

//Delete the hash table array

}

void MyHash<KeyType, ValueType>::reset() {

//Traverse through all the buckets in the hash table

//Create a pointer p to the first LinkedList object in m\_hash

//if p is a nullptr, do nothing

//Traverse through the linked list and delete all the nodes

//Delete the last node

//Delete the hashtable array

//reinitialize the member variables m\_bucketCount to 100 and items to 0

//create a new hash table with 100 buckets and set each object to nullptr

}

void MyHash<KeyType, ValueType>::associate(const KeyType& key, const ValueType& value) {

//See if the key already exists and update the value if it does

//Otherwise, increment the number of items

//Check if the current load factor is greater than the initial load factor

//If it is, Assign current hash table to a temp hash table

//Multiply bucket count by 2

//Set current hash table to a new, bigger hash table with twice the bucket size and initialize each object to a nullptr

//Associate every item in the temp hash table to the new hash table with twice the number of buckets

//Create a new LinkedList pointer and initialize it to the first item in the hash table

//If the bucket is empty, do nothing

//Otherwise add new item using tasallg

//Otherwise, traverse through the rest of the linkedList

//Set our traversal iterator to the next object

//Add the item at each point to our new hash table using tsallg

//Delete the temp hash table

//Traverse through all the buckets in the temp hash table

//Create a pointer p to the first LinkedList object in temp hash table //if p is a nullptr, do nothing

//Traverse through the linked list and delete all the nodes

//Delete the last node

//Delete the temp hashtable array

//Add the last key-value pair

}

int MyHash<KeyType, ValueType>::getNumItems() const

{

return m\_items;

}

double MyHash<KeyType, ValueType>::getLoadFactor() const

{

return ((double)(m\_items))/m\_bucketCount;

}

const ValueType\* MyHash<KeyType, ValueType>::find(const KeyType& key) const

{

//hash function prototype

//pass key through hash function and determine where to place it in the hash table

//if this bucket is empty, return nullptr

//If head node has the key, then return the value at headnode

//Initialize an iterator starting at this bucket in the hash table

//While it does not reach the end of the LinkedList at this bucket

//Set iterator to the next object

//If the key at the linkedList\* object matches up, return the value at this point.

//Otherwise return nullptr

}

void MyHash<KeyType, ValueType>::tsallg(const KeyType& key, const ValueType& val)

{

//Hash function prototype

//Setting h to bucket in our hash table using hash function

//If empty bucket, create new node and store it

//Else

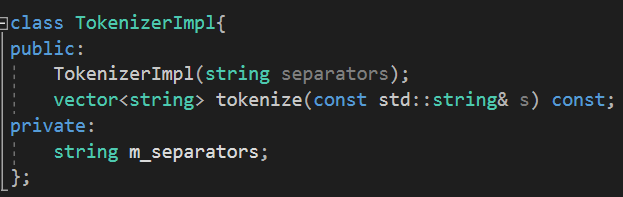
//Create a node pointer pointing to the bucket in the hash

//Create a new node and store its pointer in the node pointer created above

//Move to the next node

}

**TOKENIZER.CPP**



The private member variable m\_separators stores the separators passed in during the creation of a Tokenizer/TokenizerImpl object.

TokenizerImpl::TokenizerImpl(string separators)

{

m\_separators = separators;

}

vector<string> TokenizerImpl::tokenize(const std::string& s) const {

//create temp word

//create a vector of strings

//Let an int i = 0

//while i is less than the size of the string

//Let there be an int k that finds the first position where one of the separators exist in the string

//Use find\_first\_of and pass in m\_separators, i

//Update temp to the substr from i to the point k-i

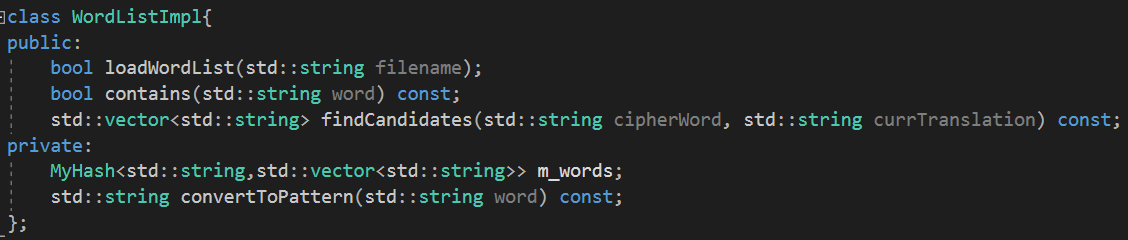
//update i using find\_first\_not\_of\_ passing m\_separators and k.

//push temp onto the vector we initially created

//return the vector

}

**WORDLIST.CPP**



This class contains a MyHash object called m\_words. Since MyHash is a templated class, our MyHash Object essentially maps strings to a vector of strings. This allows us to store different word patterns as keys, and all words that have that pattern in a vector as a valuetype for that corresponding key. The convertToPattern function takes in a word and converts into numerical equivalents, that is, a number pattern in terms of its letters. For example, if the word is Hello, convertToPattern will return 12334.

std::string WordListImpl::convertToPattern(std::string word) const {

//Create a temporary string

//Create a character array that can take 27 characters and set each character to a dash (-)

//Initialize a character i = '1'. It will be used as a counter.

//Go through all the characters in the word passed into the function

//if the character is not an alphabet, continue the loop without executing any other lines

//Else if the character is a -

//set the character at that position to i

//Increment i

//Concatenate this element in the string to the temporary string

//Return the temporary string

}

bool WordListImpl::loadWordList(std::string filename){

//Reset hashTable every time this function is called

// create an ifstream to read input from file

//std:: ifstream infile("wordlist.txt");

//Check if failed to connect file to ifstream

//Create a temp string s

//Read lines from the text file until it is in an invalid state

//For each string s, go through all of its characters

//If not an alphabet or a slash, leave the for loop

//Set the character to a lowercase version of itself

//Convert the word from the text file into a number pattern

//Create a temp vector

//If the pattern doesn't exist in the hash table

//push the word back onto the vector

//add it to the hash table

//Else push it back onto the vector at that key

//Return true

}

bool WordListImpl::contains(std::string word) const {

//Search for the pattern. If there does exist a key with this pattern

//create a temp vector and set it to the vector(value) at that key

// traverse through this temp vector

//If a word from this vector matches to one of the words we want

//Return true

//return false if the word is not in the word list

}

std::vector<std::string> WordListImpl::findCandidates(std::string cipherWord, std::string currTranslation) const{

//Check if cipherWord and currTranslation are the same size

//Return an empty vectorif this is true

//Process cipherword and currTranslation according to given spec

//cipherword should only contain letters and apostrophes

//Return empty vector if false

//currTranslation should only contain letters, apostrophes and question marks

//Returns empty vector if false

//Check if the bucket is a nullptr

//If it is, return an empty vector

//Otherwise get the vector at that bucket and save it in a vector (say, vector\_words)

//Create an empty temp vector

//Traverse through vector\_words

//Create a new string and initialize it to the word at the jth index in this vector

//Go through each character of this word

//if i == word.size()-1, push it on the vector since it is a potential candidate

//push the word onto the vector

//Break out of this for loop so you can move onto the next word

//If the character is an apostrophe, make sure it is in the same place

//In this case, just continue to the next iteration

//if the character is a question mark

//In this case, just continue to the next iteration

//if it is a letter, then the letters in currTranslation and the word must match at this position irrespective of case

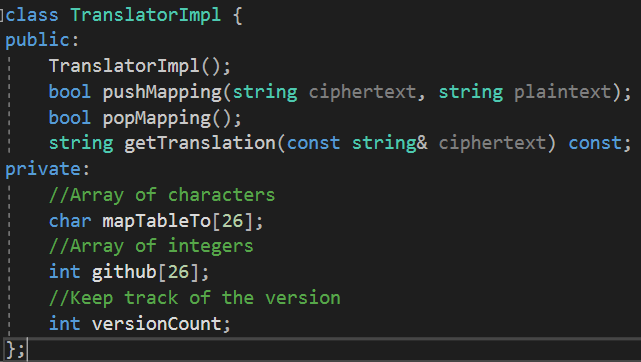
//If true, continue to next character

//Else break out of the inner for loop

//return the vector

}

**TRANSLATOR.CPP**



TranslatorImpl::TranslatorImpl() {

//Initialize version count to zero

//Go through a loop 26 times to represent the 26 alphabets

//Populate mapTableTo with ?

//Populate version control (to keep track of which level of the stack) gitHub variable to 0

}

bool TranslatorImpl::pushMapping(string ciphertext, string plaintext) {

//Check if the two parameters have the same length, else return false

//Ensure that if the character is an alphabet and doesn't map to ?, it can only map to one letter

//Check for collisions

//Run through all the 26 slots of the array

//Run through all the characters in plaintext

//Ensure that two characters don't map to the same letter

//If they might potentially overlap, return fase;

//Update version count if push-mapping is successful

//while i is less than the length of ciphertext

//If it is not an alphabet, continue

//Create an int j and set it to the character at ciphertext[i]

//find the letter in our 2D character array

//If the jth character in the character array is a ?

//assign the jth character of plaintext to it

//the the jth integer in gitHub to the current version count

//Otherwise return true

}

bool TranslatorImpl::popMapping() {

//If the version is zero, return false

//Pop the last version off the stack

//Go through all 26 slots in the array

//If the version at github[i] == versionCount

//Set the version back to zero

//Set the mapping back to ?

//Decrement version count

//Return true if all goes correctly

}

string TranslatorImpl::getTranslation(const string& ciphertext) const {

//Create an empty string called plaintext

//Go through all the characters in ciphertext

//create a temp character and set it to a lowercase version of the current character

//If it isn't an alphabet, just add it on to plaintext

//Otherwise

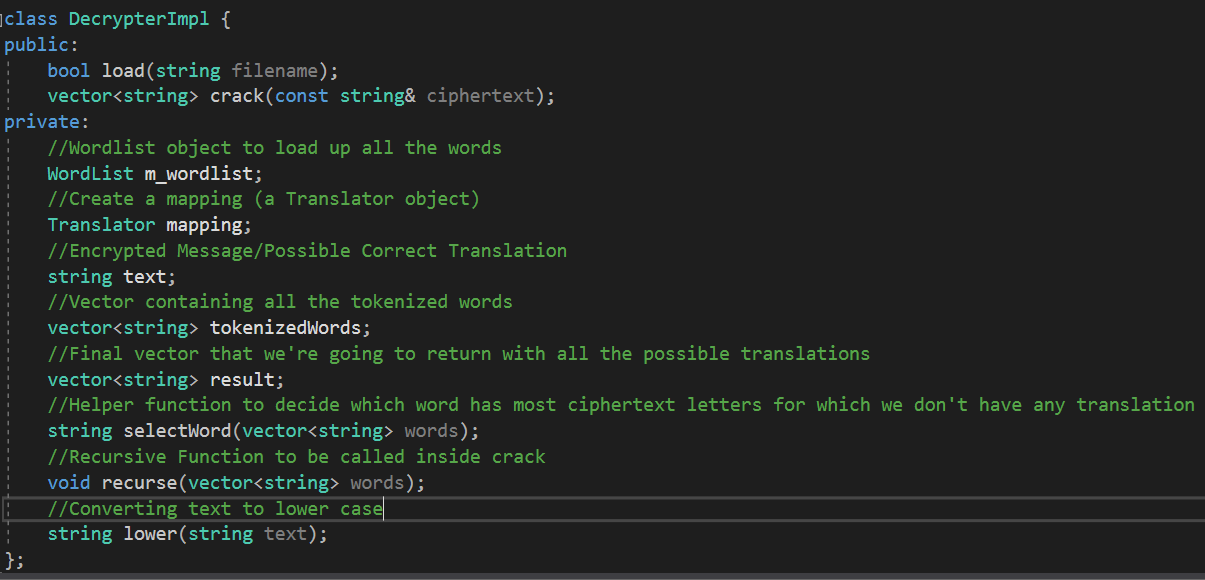
//If it is uppercase, concatenate an uppercased version to plaintext

//Else if it is lowercase, concatenate a lowercase version to plaintext

//Return plaintext

}

**DECRYPTER.CPP**



string DecrypterImpl::selectWord(vector<string> word) {

//Create an empty string called result which will be returned at the end

//Create a counter for question marks

//Create an int variable to keep track of the word with the maximum number of question marks

//While i is less than the length of the vector

//Initializ the question mark counter to zero each time the loop is called

//Get the current translation of the word in the vector

//Count the number of question marks in the current translation of the word

//If the character in the translation of the word is a question mark, increment the questionCount counter

//If this word has more question marks than the previous word with maxCount, set result to this word

//Update maxCount to this value of QuestionCount

//Return the word with the most number of question marks

}

string DecrypterImpl::lower(string text) {

//Create a temporary string

//Run through all the characters in the parameter string

//Convert them to lower case and concatenate to our temp string

//Return the temp string

}

void DecrypterImpl::recurse(vector<string> words) {

//Base Case

//If the vector is empty, then we've tried all the words

//Algorithm

//Pick the word that is least translated

//Pop this word from the vector to indicate that it has already been chosen

//Get a partial translation for the selected encrypted word

//Find the vector of words that the selected encrypted word maps to

//Recurse through the remaining words that match up to the currest partial decryption

//Assume the temporary mapping is correct and push this mapping. If it returns false, it tries to create a temp mapping to the next word.

/\*Since this mapping works, translate all the words in the ciphertext

using this new/updated mapping and push them back onto the new vector of strings\*/

//Evaluate the just-decrypted message to see if all fully-translated words are in the word list

//If the word does not have a question mark, it is completely translated

//pop the mapping

//break out of the function

//Else you increment the word counter

//If the number of words completely translated equals to the tokenized vector's length

//Push the translation onto the result vector

//Pop the latest mapping

//Otherwise the translation is promising but it is not complete

//Call the recursive function again (called recurse)

//pop the last mapping

}

bool DecrypterImpl::load(string filename) {

//if it is able to load the wordlist using the wordlist member variable object,

//return true

//if it is unsuccessful, return false

}

vector<string> DecrypterImpl::crack(const string& ciphertext) {

//Set text to ciphertext

//If the translator object has any mappings, pop them.

//Reset the result vector so that it is empty

//Create a tokenizer with all the separators

//Update the tokenizedWords vector with all the words after they are tokenized

//Call our recurse on this vector of tokenizedWords

//Sort the results in order

//Return result

}