MyHash.h:

Data structure of MyHash is an array of pointers of nodes. MyHash has a private member variable of a pointer to the array of pointers.

Insert:

Insert checks whether adding a new node to the hashtable will cause the hashtable to exceed its max load factor. If so, it will call check size to resize accordingly. After resizing (or not if not needed) it gets added to the hashtable at a bucket calculated by hashfunction.

Checksize:

Checksize check whether adding an item will cause the max load to be exceeded. If so, it doubles in size and reallocates all the nodes that are currently present in the old hashtable to a new hashtable (of the doubled size) with a different bucket number. It also deletes all nodes stored in the previous dynamically allocated array.

Big- O:

MyHash(double maxLoadFactor):

Satisfies big O requirement of O(B), where B is 100 initially when initializing all pointers at each bucket to nullptr.

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

The destructor runs O(N) times, where N is number of items stored in the hashtable. Each item stored in the hashtable must be visited and deleted at destruction.

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

Reset runs O(N + B) times, where N is number of items stored in the hashtable. Each item stored in the hashtable must be visited and deleted when resizing. After all items have been deleted, reset will dynamically allocate a new array of pointers to nodes and set the pointers to nullptr, which will take O(B) time.

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

The runs O(1) times because given a key, this function gets a bucket number directly by calling the find function, which allows the valuetype associated with the keytype to be accessed directly.

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

This runs O(1) time because it uses the calculate hash function to access key at the bucket number directly. Since I printed out the bucket number while I was testing out this function, I know most items aren’t hashed to the same bucket.

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

This runs O(1) time because MyHash keeps a data member variable that stores how many items are in the hash table.

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

This runs O(1) time because MyHash keeps a data member variable that stores how many items are in the hash table and the number of buckets.

Tokenizer.cpp

Tokenizer keeps an unordered\_set of char that serve as separators of tokens.

TokenizerImpl::TokenizerImpl(string separators)

This runs O( P ) time because it inserts P items to the un\_ordered set stored as private member data of TokenizerImpl.

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

This run in O(S) time since when looping through and examining whether each character in the string is a separator, this function calls a helper function which uses the stl unordered set find function. This function only runs in P time in the worst case.

bool TokenizerImpl::isSeparator(const char c) const

This function is called when determining whether a character is listed as a separator in TokenizerImpl. It uses the stl find function which runs in O(1) time.

Wordlist.cpp

Wordlist associate different words in a wordlist by their pattern. It holds a private data member MyHash that stores these association in a node through a keytype of string and valuetype of a vector of strings.

WordListImpl::~WordListImpl()

This deletes all the dynamically allocated vector<string>, which are associated to a keytype of string in loadwordlist.

string WordListImpl::pattern(const string& s)

This function initializing a new hashtable the maps a char to another char. For each character in a string it maps it to another alphabet according to alphabetical sequence. The resulting pattern is the letter pattern of the given string. It only adds an association if a letter in the word passed into the function is not already mapped to a letter. At end of the function, it retrieves the value of each letter using the MyHash find function and append the values to a string “value” that gets return at the end of the function

bool WordListImpl::loadWordList(string filename)

This function loads the wordlist using ifstream and getline and append each string in a line so a temporary string “s”. It then uses a tokenizer to obtain a vector of the actual words in the string. For each word in the string, it checks whether the word is a valid word. If so, it is associated with a vector of string using the “pattern” function to store all these data in one node at a particular bucketnumber in WordListImpl’s privata data m\_hash.

This runs O(W) times because we assume there is some constant such that no word in the file has a length that exceeds that constant. Therefore when we are checking whether a word is valid by checking each letter in the word, and when we are checking whether there are duplicating words in a vector with words of all the same letter pattern, the big O is still essentially O(W) since there are so many words in a wordlist.

bool WordListImpl::contains(string word) const

Since we're assuming that there is some constant such that no word has a length that exceeds that constant, this function funs in O(1) time since it directly obtain the vector where the word may be present in using MyHash find. Unless many assocations are hashed to the same bucket, this runs in O(1) time.

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

Using the letter pattern of the cipherword, findCandidates retrieve the words in m\_hash that may be potential matches to the translation. It first check if there is even a possibility for a candidate by checking validity of currentTranslation and cipherWord. If the MyHash find function returns a vector of words, it uses the current translation to check whether each word matches the current translation and adds a word to a vector to be returned if it is a candidate.

This run in O(Q) time because the main iteration it does is looping through the vector of words returned by MyHash find function.

Translator.cpp

Translator is used using a vector of stl maps. It is implemented as a stack by always accessing the last item of in the vector as the current mapping.

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

Pushmapping checks mapping the plaintext argument with the ciphertext argument will result in duplicate mapping with the current mapping. Duplicating mapping happens when the letter mapping is not one to one. It first make a copy of the current mapping, adds the mapping onto the current mapping if allowed, then add this copy to the vector of maps stored as private data member of this class.

This runs O(N+L) times because it examine each letter in the ciphertext and check the mapping of each letter with the current mapping, which stores information about pairing among the 26 letters at each letter.

bool TranslatorImpl::popMapping()

This runs O(1) because it access the last element in the vector of maps and deletes it.

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

This retrieves current mapping of the ciphertext, catered to lower or uppercase according to the ciphertext.

bool TranslatorImpl::checkDupMapping (const char& cw, const char& pw, const map<char,char>& currMap) const

Check dup mapping checks whether two cipherword letters are being mapped to the same plain text word and vice versa. It returns true if adding the cipherword and plaintext word argument will result in a mapping that is not one to one, and return false if mapping is consistent.

Decrypter:

bool DecrypterImpl::load(string filename)

This function dynamically allocates a new word list because it has to discard old wordlist everytime this function is called. Because it delegates the actual loading part to the wordlist pointer data member of this class, it also runs in O(W) time.

void DecrypterImpl::decryptHelper(const string& ciphertext, const vector<string>& cipherwords,Translator& curMapping,vector<string>& solutionVec) //directly update solutionVec with each recursive call

This is the helper function for crack. It always uses the most current mapping when recursing since an updated translator object is passed into the function with each recursive call. It chooses the most untranslated words by calling another function count\_qmarks and storing the index of the most untranslated word. Using the most untranslated word, it finds all candidates that match the given cipherword and most updated translation. For each candidate, it checks if decrypting the cipherword with letter pattern of the candidate will result in inconsistency mapping and decrypt the message by using the translator. It records a valid translation if all ciphertext can be translated, recurses if not all words are translated, but all currently translated words are found in wordlist. If one of the decrypted words is not in the wordlist, it discards the current mapping and look at the next candidate. The function returns when a word not in the word list is generated. All valid solution are stored in “solutionVec”, which get returned in the crack function.

Bugs:

For *DecrypterImpl::crack(),* my implementation runs into an infinite recursion loop for this particular ciphertext.

Xjzwq gjz cuvq xz huri arwqvudiy fuk ufjrqoq svquxiy. -Lzjk Nqkkqcy