Report

Jingjing Nie 304567417

1. There are no bugs in my classes
2. **MyHash:**

The class MyHash uses the data structure hash table and uses a linked list to deal with conflictions. Therefore, to initialize a hash table, an array of pointers pointing to a pair (consists of a kay and value) has been created using a pointer, and then all the pointers inside the array have been set to nullptr.

void associate(const KeyType& key, const ValueType& value)

when a new pair of key and value is associated into the hash table, the pattern of the key is first determined and then hashed to a corresponding int. Therefore, the key and value pair can be hashed into the table after its position has been determined by its pattern. Then, since a linked list is used to deal with collision, if there are already pairs hashed to the position, this new pair will be added to the back of the original linked list. If the key is already present in the spot, its value will be updated.

If the load factor has exceeded the maximum level, a new hash table with twice the size has been created and all the old hash table was then traversed to hash all its contents to the new table. The old table was then freed.

**Tokenizer:**

This class is used to take in separators and then use the separators to separate the incoming string. A vector is used to store the separators and to store the strings after being tokenized.

**WordList:**

This class, through the use of MyHash as the data member, takes charge of loading all words from wordlist.txt file. Therefore, it uses a hash table as the data structure. For the keyType and valueType for MyHash data member, we use a string type for kryType, and a vector for valueType.

bool WordListImpl::loadWordList(string filename)

To load the words, each word is extracted from the text file and the pattern for each of them is determined. Then, similarly to what is described in the MyHash class, each word in wordlist.txt is hashed to the hash table. The key for the world is its corresponding pattern, and the value is a vector of words that match with this pattern.

bool WordListImpl::contains(string word) const

This function is used to determine whether a word is contained in the wordlist hash table. Firstly, the pattern of the word is found. Then, the index of “word” will be determined, and thus the linked list linked to that index will be traversed to see if “word” is contained in the table.

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

This function is used to find and return all plaintexts that can possibly be the translation word for “cipherWord” based on its current translation. A vector is used to store and return the qualified plaintexts.

**Translator:**

This class is used to either fully translate or partially translate a cipher word. To dynamically store all temporary mappings, a vector is used. This vector contains pair, which is used to map each cipher letter to plain letter. To initialize the mapping, all cipher letters are mapped to ‘?’.

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

For each time when this function is called, a new mapping is created, and all the old items in old mappings are copied into this new mapping. Then it starts to check if the plaintext can map to the ciphertext without invalidating the mappings that are already existing. If at least one criteria has not been met, the current mapping will be discarded and return false. If all the criteria are met, the mapping is pushed into the vector container and return true.

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

This function returns the partially or fully translated text for ciphertext based on the current mapping. Since the container to store mapping is a vector, the current mapping will be the last one in this vector. Then, the corresponding translated word can be found and returned.

**Decrypter**

To decrypt a ciphertext, a pointer pointing to WordList class is used as a data member, which can be used to load the wordlist. It also involves two data members, one of type Tokenizer, which can be used to tokenize an incoming string, and one of type Translator, which can be used to find new possible mappings and get translated words. The function vector<string> DecrypterImpl::crack(const string& ciphertext) follows the algorithm listed in the spec and can be used to decrypt an encrypt word.

1. All the methods have satisfied the big-O requirements.