Elon Glouberman

ID: 904769785

Project 4

1. Classes that have known bugs or other problems you should know about
   1. N/A. Completed all the problems.
2. High level description of what data structures and algorithms you chose for each of my classes’ non-trivial methods and data structures.
3. Added onto each of my descriptions.

**HashMap**

For my hashMap, I implemented an array m\_array, of Node\* that were dynamically allocated. Within each Node there are Node\*’s m\_next, which points to the next node in that bucket, KeyType m\_key, which contains the key , ValueType m\_value, which contains the value of a certain key, and a ~Node() object, which acts as a destructor to destruct each of the buckets within each node, which enables me to simply call delete [] m\_array in my hashMap destructor. I also have private variables m\_numBuckets to tell me how many buckets there are, an int m\_items to tell me how many items there are total, an int m\_size to tell me the full capacity of my array, and a m\_loadFactor to tell me the load Factor.

Additionally, I also have a findBucketNumber function that accepts a key and a size of the array in order to compute where the index of the new insertion should go. My associate function encounters three cases: if the array at the index of where the key should be inserted is a nullptr, we insert it without an issue. If there is already a node there, then we remap a new node to point to the head of the node that’s already in the bucket and we map the array’s slot to the new node we are inserting. If we find that a key is already in an array, we update it and we don’t allocate any more memory. For my associate function, it runs big-O(1).

If the load factor gets too high, then I copy all the contents of the current hashMap into another doubly sized one that recounts the correct index. This runs O(X) times, where X is the number of items in the hash table in order to copy everything over.

**Tokenizer**

My tokenizer class implements a queue to establish the separators in a certain string and to return back a vector of words without those separators. I first have a private variable string m\_separators that the user inputs to keep track of the separators that the tokenizer is looking out for.

We first iterate through every index of the inputted word.

If that index is not a separator, we add it the queue

If that index is a separator

If queue is not empty

while the queue is not empty, we append the front of the queue, then pop it to a string called “word”

we push “word” to a vector that will later return to the user.

If that index is not a separator, we push the character onto the queue

In the case where we don’t end on punctuation, we pop the remaining characters in the queue and add it to the returned vector

This function runs O(SP), where S is the number of characters in s and P is the # of separators.

**WordList**

My wordlist takes an input file name and loads the words into a my hashmap that maps an int to a list of strings. The int is a pattern for a certain word (“google” => “122134”). I convert each word from the file to pattern and double check it with the find method in the hashmap class to see if it is currently in the hashmap. If it is, I append the new string value into the list <string> in that bucket. If it is not, I associate a new value of the word pattern to the list< string> string word. My loadWordList function returns false if the file cannot be found. My function also checks to see if each word getting loaded in is a letter or an apostrophe. The Big-O complexity of my loadWordList function is O(W) where W is each unique word in the file.

Findcandidates simply checks all the strings in the bucket that has the association of the word pattern. It compares each character in parameter (currTranslation) to each character in the bucket’s word. If there is no match with the character that is not a ‘?’ then we go to the next word in the list to compare. This runs O(Q) times where Q is the number of words in the owrd list that match the letter pattern of cipherWord.

**Translator**

My translator class implements a list<map<char,char>> as my stack to keep track of all the pushed mapping tables to my class. I also have a map<char, char> cipherToPlain map as well as a map<char,char> plainToCipher to find the associations of the maps going in both directions. This is the most efficient way to check whether a certain mapping already is mapped when we need to check for that. My findInMapping function checks to see if a certain char that we are trying to add into the map has already been mapped.

My getTranslation function takes in a parameter string ciphertext. It utilizes my plainToCipher map of char to char to figure out what each letter should map to. If there is no mapping, I return a ‘?’ instead. If there is a mapping, then I convert the mapping char into the appropriate case (upper case or lower case according to the cipher text) and then I append it my word I return.

My pushmapping function runs O(n+l) times, but since n and l are both constants, it is essentially O(1). All we do here is add something to a map.

My pushmapping function runs O(L) times where L is every letter in the English alphabet, but L is a constant so it is essentially O(1).

**Decrypter**

My decrypter class utilizes the entire algorithm that is seen in the spec. To help, I have several private variables to keep track of certain things during that algorithm. I have a Wordlist m\_wordlist to load the file into a hashmap, a tokenizer m\_tokenizer class to take care of the tokenization of the strings that we are trying to decipher, a Translator m\_translator to map the current mappings and push/pop mappings that work/don’t work, a vector<vector<string>> m\_validSolutions to keep track of the solutions in the code that are perfectly translated and that work, a detokenize(word, vector< string >) function that inserts all the separators that we initially took out, and finally a function called tryCandidate(vector<string> candidate, word w, vector<string> cipherWords) that tries a candidate that matches a word pattern and attempts to map the rest of the crypted words according to that word mapping. This function is in my crack function which also recursively calls crack again if the solution we find is not perfect but promising.

My load function utilizes my wordlist private variable, so it runs O(W) times where w is the number of words in the file.