Project 4 Report

There are no known errors or bugs in any of the classes.

All functions satisfy time complexity requirements

MyHash:

Created two structs, one (called Pair) to hold the key and value, the other (called Node) to hold a Pair and a pointer to the next Node. Used a dynamically allocated array of Nodes as the hash table. Each bucket contained a pointer to a linked list of Nodes.

The constructor appropriately sets the max load factor. It then allocates an array of 100 Node\* elements. The constructor run in O(B) (satisfying requirement).

The reset() function deletes the current table with all the Nodes, and allocates an empty array of 100 Node\* elements. Runs in O(B) (satisfying requirement).

The find() function generates the bucket number using the provided hash function. The linked list at that element is then traversed to find the Node with the appropriate key value. If found, the address of value is returned, otherwise nullptr is returned. Runs in O(1) (O(X) in pathological case) (satisfying requirement).

The associate() function first calls find to see if the key already exists. If yes, the value is updated, and the function returns. If by adding an element, the load factor does not exceed the max load factor, a new node with the key value pair is created. The provided hash function gives a bucket number based on the key. The node is inserted at the beginning of the linked list at that bucket. The function returns. Otherwise, a new dynamic array of double the current size is made the underlying hash table, with a temporary pointer pointing to the old one. Associate() then calls itself with the new array as the hash table, and elements from the old table being added with every call. The new key value pair is associated. The old table is finally destroyed. Runs in O(1) (O(B) if number of buckets changes) (satisfying requirement).

getNumItems() and getLoadFactor() each return the required value (member variables keep track of number of occupants of the table and the number of buckets). Run in O(1) (satisfying requirement).

Destructor deletes the hash table. Runs in O(B) (satisfying requirement)

TokenizerImpl (Implementation class of Tokenizer)

Used a dynamically allocated array of characters to hold the separators.

The constructor allocates the array, adding the separators from the string one at a time. Runs in O(P) (satisfying requirement).

The tokenize() function uses a local vector of strings to hold the separated words. The passed string is looped through; each character is checked to see if it is present in the array of separators. If not , the character is added to the current word string. If yes, he current word is added to the vector and is itself made the empty string. After the loop, the final word is added to the vector. The vector is returned. Runs in O(SP) (satisfying requirement).

The destructor deletes the array of separators.

(A destructor was added to the public interface of Tokenizer)

WordListImpl (Implementer class for WordList)

Uses two MyHash objects. One with keytype string and valuetype string. This holds the wordlist, with the words themselves being the keys. The other has keytype string and valuetype vector of strings. This holds the wordlist, with the character pattern of a particular word as the key. Also uses a struct called Link which holds a string and a pointer to the next Link.

The loadWordList() function resets both the MyHash objects. If the input file can’t be opened, it returns false. It keeps getting words from the file using a loop. Each word is converted to lowercase. A character pattern is made for the word, using character starting from A in the ascii table. Every new character encountered in the word uses the next ascii character. For example , “animal” would give “ABCDAE”. If the words contains a non-letter and a non-apostrophe, the next word is gotten. The word is added to both the MyHash objects, with itself as the key in one. For the second hash table, the vector of words with the same pattern is gotten. The current word is appended. If the pattern does not already exist in the hash table, an new association is made. After the loop, if the eof was not reached, returns false, otherwise returns true. Runs in O(W) (satisfying requirement)

The contains() function converts the word to lowercase and calls the find() function of the hash table using the words themselves as the key. Returns a Boolean depending on whether the word was found. Runs in O(1) (satisfying requirement)

The findCandidates() function first checks if the cipherword and currtranslation are valid (do not contain characters that are not allowed). The character pattern is made for the cipherword using the same algorithm as previously mentioned. The second hash table is used to find the vector of words with the required pattern. If no such pattern exists, the function returns an empty vector. Every word in the in the vector is checked with the currtranslation to ensure there are no discrepancies. Only those words that match are added to the vector to be returned. The candidate words vector is returned. Runs in O(Q) (satisfying requirement).

TranslatorImpl (Implementation class of Translator)

Uses a struct called MapPair that itself uses a struct called Pair that contains a char and an int. MapPair contains two arrays of Pairs, each of size 26 (number of alphabets). TranslatorImpl uses a dynamically allocated MapPair and a vector of MapPair pointers. The two Pair arrays represent the cipher character set and the translated character set. Each pair in the arrays holds an alphabet and the index in the other array that that alphabet is linked to.

The constructor creates a new MapPair. MapPair itself has a constructor that initializes both the Pair arrays with each pair getting a letter from the alphabet in order, and each pair’s integer being -1. The current map is set to the new map just created.

The pushMapping() function creates a new MapPair using the current map (an overloaded constructor was provided). The function loops through the cipher text. If a letter is invalid, the new map is deleted and the function returns false. If a translation is made that is inconsistent (checked using the two pair arrays), the new map is deleted and the function returns false. Otherwise, the map pairs of the new map are updated, with the integers in the pairs being set to the index of the element in the opposite map that is linked to that pair. For example, if “ABC” were translated to “XYZ”, the first three elements in the “fromPairs” would be {{‘a’ , 23}, {‘b’, 24}, {‘c’, 25} …}, and the last three elements in the “toPairs” would be { … {‘x’ , 1}, {‘y’, 2}, {‘z’, 3} }. The current map is added to the end of the vector, and the new map is made the current map. Returns true. Runs in O(N + L) , which is O(1) (satisfying requirement)

The popMapping() function checks if the member vector is empty. If yes, it returns false. It makes the last element of the vector the current map, deletes the required map (the current map to be deleted pointed to by a temp pointer). Returns true. Runs in O(1) (satisfying requirement)

The getTranslation() function loops though the cipher text. It adds non-alphabets to the translated version without change. It gets the position of the current letter in the alphabet (from 0 - 25), uses that as the index in the array representing the cipher text letters. Taking the pair from that array, the index of the corresponding element in the array representing the plain text letters is gotten and that letter is added to the translation, with appropriate case. If the pair in the “from array” contains a -1 as the integer, a ? is added to the translation instead of an alphabet. The translation is returned.

The destructor removes and deletes every MapPair from the vector. It deletes the current map as well.

DecrypterImpl (Implementation of Decrypter)

Uses a WordList, a Translator, a Tokenizer, a vector of strings and a set of ints. The vector of strings holds the possible decryption of the ciphertext. The set of ints stores the indices of the words in the ciphertext that have already been chosen to create a map until that recursive call.

The constructor sets the member variable (separators) of the member tokenizer.

The load() function calls the wordlist’s loadWordList() function. Runs in O(W) (satisfying requirement)

The crack() function

first checks if the set of ints is empty. If yes, it empties the vector of possible answers.

A vector of the cipher words is gotten using the tokenizer’s tokenize()

A word is chosen from the cipher words vector such that it has not already been chosen (not in the chosen words set) and it contains the most number of unmapped letters among those not chosen

The index of the chosen word is added to the set

A translation of the chosen word is gotten using translator’s getTranslation (), with the chosen cipher word as argument.

The possible matching words in the dictionary are gotten using wordlist’s findCandidates() function, with the cipher word as the first argument, the translation as the second. The possible words are gotten as a vector

The possible words are sorted by using a temporary set.

If there are no possible words, the current mapping in translator is popped, the chosen word’s index is removed from the set, and an empty vector is returned.

Looping through every word in possible words

Try to push a mapping into translator (from the cipher word to the current possible word)

If not possible, continue to the next iteration of the loop.

Using the newly pushed map, the cipher text is translated using translator’s getTranslation()

The translated text is tokenized

For every word in the translation

if it is not present in wordlist’s dictionary.

Set a flag notFoundInWordList to true

break

if the word has unknown characters (?s).

set the flag notCompletelyDone to true

if(notFoundInWordList)

pop the mapping added to translator

continue with the next word in possible words

if(notCompletelyDone)

call crack() recursively

continue with the next word in possible words

add the plaintext translation to the vector of possible answers

pop the current mapping from translator

pop the current mapping from the translator

remove the current chosen cipher word’s index from the chose words set

return the vector of possible answers