Introduction

In 1879, Lewis Carroll proposed the following puzzle to the readers of Vanity Fair : transform

one English word into another by going through a series of intermediate English words, where each word in the sequence differs from the next by only one substitution. To transform head to tail, one can use four intermediates: head → heal → teal → tell → tall → tail. We refer to the smallest number of substitutions necessary to transform one word to another as the Lewis Carroll distance between the two words.

Requirements

● Fill in WordSet.cpp

○ Your implementation must be done via a hash table, as described in lecture.

○ Your collision resolution policy must be quadratic probing.

○ You must use a dynamically-allocated array, not a std::vector or similar container,

as the basis in your WordSet.

○ Your implementation must fit the interface given

○ Your implementation does not need to be templated -- nor should it be, for the

purposes of this assignment.

■ In fact, doing so will cause an issue for some of our provided tests.

○ You do need to implement the destructor.

○ Do not hard code your table for the uses we’ll have in the next section.

○ You must also fill in the desired hash function: treating words as numeric types.

■ Be careful about when you take the modulus within the hash function.

■ Be careful about when you take the modulus outside the hash function.

Write function std::string convert(std::string s1, std::string s2, const

WordSet & words) in proj3.cpp

○ This function will return the conversion between s1 and s2, according to the

lowest Lewis Carroll distance. Separate consecutive words with

[space]-->[space]. If there are two or more equally least Lewis Carroll distance

ways to convert between the two words, you may return any of them.

○ It is recommended that you compute the distance via a breadth-first search. To

visualize this, imagine a graph where the words are vertices and two vertices

share an (undirected) edge if they are one letter apart.

○ A good thing to do the first time you see a word in the previous part is to place it

into a map<string, string>, where the key is the word you just saw and the value

is the word that you saw immediately before it . This will allow you to later

produce the path: you know the last word, and you know the prior word for each

word in the path except the first. Furthermore, if the key isn’t in that map, this tells

you that you haven’t seen it before.

○ Your implementation does not have to be the most efficient thing ever, but it

cannot be “too slow.” In general, any test case that takes over a minute on the

grader’s computer may be deemed a wrong answer, even if it will later return a

correct one.