ICS 46 Summer 2023

Project #5: Cuckoo Word Ladders

Due Monday, August 14, 2023 at 12:00 p.m.

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, assuming we are using a standard English dictionary to determine words: 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.

In this project you will be responsible for implementing a solution to the aforementioned word ladder problem. In your solution you are required to use a *set* to determine the validity of words, seeing as the word ladder chain itself is dependent on the dictionary that you use. For example, the above word ladder chain only works if head, heal, teal, tell, tall, and tail are all present in the dictionary. I could just as easily tell you to solve the word ladder problem using a dictionary that doesn't contain those words, and you might come up with a different solution.

Before solving the word ladder problem, you will implement a data structure to store the corpus of words. This data structure will effectively be a set of strings, and in this project is called the "WordSet". You can think of this set like an `unordered_set` from the standard library that only stores strings. You will leverage the Cuckoo hashing techniques learned in class to implement this word set.

Part 1

```
unsigned polynomialHashFunction(const std::string & s, unsigned base,
unsigned mod)
```

Before implementing this function I highly recommend you read the "Polynomial Hash Codes" section of the class textbook in section 9.2.

Since your WordSet structure will be hashing strings, you will need a hash function in order to map said strings to integers. This will indicate which index a given string is stored at in the hash table.

This function will interpret the string parameter, which will contain only lower-case letters, as coefficients for a polynomial of degree equal to the length of the string. You will evaluate it at the point "base," which can be thought of (if you prefer) as interpreting the string as a base-that integer. Treat 'a' as 1, 'b' as 2, and so on. It returns the smallest positive integer that is a representation of that string, in that base, mod the given modulus.

Example:

`polynomialHashFunction`(“abz”, 5, 10) = $(1 * 5^2 + 2 * 5^1 + 26 * 5^0)$ % 10.

Be careful about when you take the modulus within the hash function, as these numbers can get quite large. As a hint, you will want to recall some properties of modulus operations from previous classes (the textbook also shows these properties in the aforementioned section). Be careful about when you take the modulus outside the hash function for the same reason. For this function you can only assume that `base * base` will not overflow an unsigned 4-byte integer. That is, the range of base is: 2 <= base < $\sqrt{2^{32} - 1}$.

Do not assume any parameters match the named constants in the file! You should only compute the hash value based on the parameters passed to this function.

```
WordSet::WordSet(unsigned initialCapacity, unsigned evictionThreshold)
```

This is the constructor for the WordSet class. You are required to use Cucko hashing to implement your hash table. You should initialize any private member data here. There are two parameters to this function.

`initialCapacity` - This is the starting size of your two arrays used in Cuckoo hashing.
`evictionThreshold` - This is the number of “evictions” that are permitted before resizing the table. Note, this is different from building a Cuckoo graph as seen in class. This is a simplified *heuristic* for determining a good time to resize, rather than needing to *prove* that a resize is necessary with the Cuckoo graph..

Your WordSet must be implemented using the cuckoo hashing policy. This means there will be two underlying arrays, as the cuckoo hashing policy will use both for the eviction mechanism, as described in class. Your two tables must be implemented as dynamically allocated C-style arrays. This means no vectors. initialCapacity will be the starting size for each of these arrays. So, initially your WordSet will look like the following, assuming an initialCapacity of 5 is used:

T_1	""	""	""	""	""
T_2	""	""	""	""	""

Note: In lecture we used T_0 and T_1. The project uses T_1 and T_2. Sorry for the discrepancy!

You will see two constants in the WordSet class `BASE_H1` and `BASE_H2`. Under the cuckoo hashing policy you will sometimes be inserting strings into T_1 and sometimes into T_2. When determining the index of a string for T_1 you should call `polynomialHashFunction` and use `BASE_H1` as the “base” parameter. For example, let’s say we want to insert “hello” into T_1 we would call

`polynomialHashFunction(“hello”, BASE_H1, initialCapacity)`. The result of this would be the index we would store “hello” at. You would use `BASE_H2` if you need to insert into T_2. For example, if “hello” were to be evicted by a subsequent insertion.

If `evictionThreshold` evictions happen you should resize your hash table. This is the heuristic we discussed in class. More on this in the documentation for the **`insert()`** function.

Also observe that this class is not templated because it will only store strings. Yay for not needing to test other types :D

```
WordSet::~WordSet()
```

You need to implement the destructor for this class. Remember, the underlying storage for your hash table must be implemented using dynamically allocated C-style arrays. You should figure out a way to free the memory you used for those arrays here.

```
void WordSet::insert(const std::string & s)
```

You should insert strings into your WordSet set using the cuckoo hashing resolution policy. As mentioned in the constructor documentation there will be two underlying arrays storing the strings. When inserting a string using the cuckoo hashing policy you should always insert the new string into its corresponding slot in T_1 (determined by calling polynomialHashFunction correctly). If there is already a string there, the existing string should be evicted and inserted into T_2. This eviction should continue up to evictionThreshold (from the constructor) times, at which point a resize should occur.

When you resize and rehash, the next table size should be the *smallest* prime number that is no smaller than twice the current table size. For example, if your current table size is 11, your next one is 23. If your current table size were somehow 13, your next one would be 29.

It *is* possible for *nested* rehashing to occur. This means that while you are rehashing and reinserting elements a *second* rehash could be triggered. Your code should handle this case. The provided wordset dictionary is guaranteed to yield a stable cuckoo hash table eventually. If you are seeing multiple rehash cycles your hash function is probably incorrect and colliding too frequently.

```
bool WordSet::contains(const std::string & s) const
```

Returns true if string s is in the WordSet, false if not.

```
unsigned WordSet::getCount() const
```

Returns the number of distinct strings in the WordSet.

```
unsigned WordSet::getCapacity() const
```

Returns the current capacity of the WordSet. In other words, how large each of the underlying arrays are.

Part 2

```
std::vector<std::string> convert(const std::string & s1, const std::string
& s2, const WordSet & words)
```

This function will return the conversion between s1 and s2, according to the lowest *Lewis Carroll* distance. The first element of the vector should be s1, the last s2, and each consecutive should be one letter apart. Each element should be a valid word (i.e. it is present in the `words` WordSet passed in). If there are two or more equally least Lewis Carroll distance ways to convert between the two words, you may return any of them.

If there is no path between s1 and s2, return an empty vector.
If s1 and s2 are the same word, return a vector only containing that word.

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. If you do not know what this means, please ask any of the helpers or post on EdStem.

You *may* use std::queue -- you do not need to write your own

A good thing to do the first time you see a word in the previous part is to place it into a std::unordered_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.

Restrictions

Implementation

- You **may not** use any classes that are part of the C++ standard template library in implementing your Wordset. You *may* use simple functions, such as std::swap or computing a logarithm if you so choose.
- You may use std::queue, std::map, std::stack, and/or std::set in convert.cpp, but you *may not* use them in place of your WordSet: you must use that in the appropriate places. *A solution for convert.cpp that does not use WordSet to determine if something is a word will receive no credit*.

Timing

- You must be able to insert 40,000 words into the WordSet in 1 second on OpenLab. Note: the provided words.txt file contains around this many words.
- The `convert` tests will be given time proportional to the ladder length. Though, they should run extremely quickly. To be concrete, the "BankingToBrewing" test will be given 2.5 seconds to run.

Grading

This is an additional warning that the public tests are not comprehensive. Remember that the compiler *does not* compile functions which are not used. Thus, at the bare minimum you should add additional unit tests which get all of your code to compile.

This project will use a modified 40% public, 40% private, 20% memory leak distribution.