Nicholas Orsini

README Project 2: SpellChecker:

This project demonstrates the applications of hash tables and hash functions. It does this by using a subset of the English dictionary. The project contains the following:

Colors.h: Defines the character codes to print in various colors

Dictionary.h and Dictionary.cpp: Defines the class dictionary who contains a HashTable. In essence, a dictionary is just a hash table with a defined type and some additional features. The user has the choice to change the hash function being used, more of that discussed in *options* discussion, print a summary of the dictionary which summarizes the hash table data that makes up said dictionary. This ranges to information on build time, bucket characteristics, and spread of cell depth throughout the table.

HashFunctions.h: Defines hashing functions that can be used in a hash table. Currently there are only two compatible hashing functions “ComputeHashString1” and “ComputeHashString3”. The former is an average performing hash function and the latter performs very poorly but is included to demonstrate a user’s ability to change the hashing function in an already created dictionary using the hashingfunction pointer.

HashTable.h and HashTable.cpp: Defines a templated hash table. A hash table is just an array of LinkedLists. An improvement to the structure of this class would be to make the size of the table flexible, i.e a pointer variable, instead of a fixed size. Furthermore, the HashTable class and the file HashFunctions.h work very much together. HashTable.cpp just deals with table, it itself doesn’t know what hashing function is being used. That ordeal is dealt with through the Dictionary because that is a type of Hash Table, that type if std::string. Dictionary accepts hash functions that have only one parameter, a string. There are other functions in Hashing functions but are not compatible with the dictionary. Hash Table just deals with inserting and searching lists (mostly). It also calculates and writes to a file. The hash table class calculates a rank [defined ranking based on cell counts]. I defined this function as an average based on how many nodes are in a cell. If a cell has no nodes in it, a score of -100 is added, if there is 1 node in the table, a score of 90 is added, if there are multiple nodes in the list the score of 90 decays based on the number in the list. 90^(1-(#nodes/100). By this, 2 nodes will receive a score of 82. It is highly possible that this ranking is too harsh in the sense that it decays too quickly. Either way this calculate happens for each list in the table. The scores are summed along the way. Then at the end the score is divided by the size of the hash table, thereby, an average. The closer the score is to 90 the better the hash function, and a score of -100 would be the worst hash function possible. The biggest bucket size, count (number of cells with this many nodes), and time are also calculated. The smallest bucket size, count, and time are also calculated. And lastly the time to empty the table, i.e delete all contents in the table, is also timed and outputted to a text file, “SummaryTable”.

LinkedList.h and LinkedList.cpp: Defines a templated linked list. Additionally, there are functions to retrieve the number of nodes in the list (just a getter, as number of nodes is an attribute of the list), and search for a list given a data.

Node.h: Defines a templated struct. The node stores the actual data (a string in this case) and the string’s associated hash value. Again, as I tried in Lab7, I can’t seem to figure out how to correctly implement operator overloads for templated structs. I can’t seem do it for non-templated structs. As a result, I discarded the need for this, but I wanted to overload the << and == operators for nodes.

UtilityGeneric.h: This file defines functions that are useful everywhere in code. It contains string modification functions, many of which are defined in the standard template library, but I wanted to avoid using their functions, so I created my own. This includes:

* Calculating a string’s length
* Parsing the first word off of a space-delimited list of words
* Calculating the number of spaces + 1 are in a string as that is equivalent to how many words are in a space-delimited list.
* Removing all punctuations in a string (, . ? !)
* Removing sentence punctuation which only removes a punctuation marking if it is followed by a space or the end of the string
* MakeLower makes all the letters in the string lower-cased.
* RemoveDuplicant removes all but one duplicated word from a space-delimited list in a string. This was critical for my calculating of edit words.
* Return next space returns the index where the next space occurs given a location in a string. This was crucial for parsing a word that does not start of the string.
* IsWithin returns true if a word is within a space-delimited list
* WhereamI returns the index where a word begins in a space-delimited list and -1 if it is not in the list
* Replace given a list of words, another word, and an index the word in this list beginning at the index is replaced with the solo input word. In my code I describe the situation as a host and parasite, the parasite infects the host (with its word) and kills the word that is there.
* RemoveCharacter removes the character at a given index and returns the string without it

Menus.h: This file contains all menu-related functions. The last few labs and projects I had been doing my menus the same way and thereby, writing 100-200 lines of code for each type of menu where each or more or less the same thing just with different number of options and prompts. For this project I wanted to no longer do this and make it simpler. Instead there is only 1 menu function that takes 4 inputs, 3 strings and an int and returns an int. 1 of the 3 strings aren’t used but I kept it there since it gives a lay of expandability if I wanted to make the menu more complex later or if another function needs another prompt. Only 2 strings are needed for most menus, an opening and a prompt but the main menu takes 3 prompts. There are many namespaces in this file to try to encapsulate functions. Since the prompts are in a namespace it is not directly accessible, easily accessible but not direct. Each namespace specializes in a different type of menu. In each namespace are const strings that define the prompts, the main includes all of these namespaces.

Lastly,

Main.cpp: This is the head-honcho, and is where the user plays. When the program begins it reads in data from a dictionarydirectory defined in Dictionary.h, through the options menu this can be change. If POS is on (print on startup), is set to true, then a summary of the building of the dictionary and associated hash table will be displayed. From the main menu the user has the option to:

Spell Check: a string (word/phrase(s)), empty strings are not allowed and if there are no words then you just return to menu as there is nothing to correct. If there are no spelling errors, based on the directory, in the sentence the user is congratulated. If there is an error the errors are displayed. The user is then prompted if they want to add the word to their dictionary for each wrong word. Then the user enters a position of some decision making, they can choose to either fix their sentence or just return to the spell check menu. First the user is presented all words 1 edit distance (i.e any 1 change in the word they entered). The time is outputted into a text file. An edit word is formally defined as any word that can be formed by changing 1 character. This includes swapping character with another character, adding a character to either the front or end and removing any character from the word. After this is calculated the list of 1 edit words is displayed and user has the chance to fix their word now or move on to see 2 edit word suggestions. If they correctly fix the word they will not be shown 2 edit words. If they try to fix it and fail to enter a word in the directory, they will just be prompted to enter a 0 and move onward. This is somewhere I can improve the code by giving them the option to add to the directory then and there, but I see that as a double-edged sword and not actually sure if it would improve things (Maybe include that as an option in Options- to be prompted to add to directory when and what conditions). The user then moves on to be prompted with 2 edit words. I first approached this problem non-recursively and had a working function but it took looked very messy. Our recent discussions on dynamic program made me want to reapproach it. Although, not dynamic programming, it is a better algorithm and uses recursion. An edit of an edit on a word is the same as 2 edits on a word. Therefore, the problem is just recursion. A dynamic approach could be found however, given that some redundancies will occur and that is why I call “removeduplicant” at the end of both of these functions. A dynamic solution would fix this if found. The user then has to make an input until the input is a word or they choose to add their word to the dictionary. Once this occurs the original input is updated with the fixes for all misspelled or absent-from-directory words and the user returns to the spell check menu.

Dictionary Options: Enables the users to study words from the dictionary that also coincide with a text file compiled from many GRE study term suggests with their definitions. The user can study a subset of these terms at their own pace. The other option the user has here is to display a single list from the hash table given a string. The string is hashed and the whole list will be displayed. That way the user can analyze if the current hash function is good at hashing differently similar things or see if there is clustering based on similarity. If the word they inputted is not in the dictionary they can once again, add it to the directory.

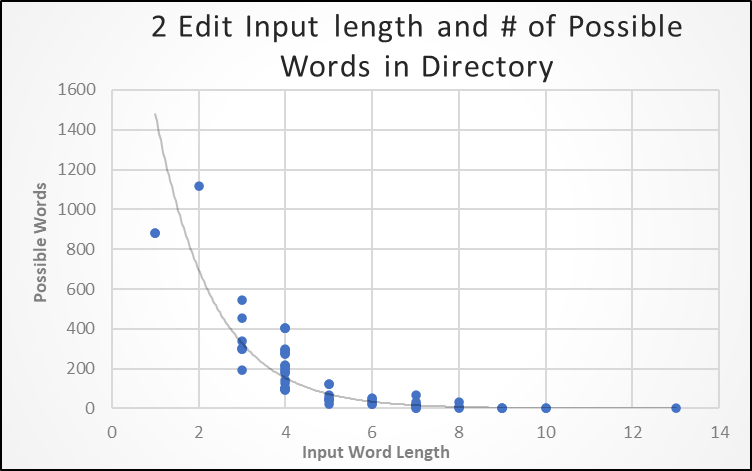
Options Menu: Options the user can interact with. The first is About, just a small synopsis of the program. The second is the ability to modify the currently enabled hash function. There are currently only 2 hash functions enabled, but these feature makes it easy to add more hashing functions working in the code. If the user changes the hash function data is cleared from the dictionary and it is reread in using the new hash function. The third option is to change the current dictionary directory. If the user would prefer to use their own directory, they can input the name of their text file without the .txt at the end, permitted that text file is in the proper folder. If the file they enter does not exist nothing will happen, if it does data in the dictionary will be dumped and reread in using the new directory but the same hashing function. The fourth option is where I would compile all miscellaneous options if more are to be added. Currently there is only one option within this section of the options. That one is to turnoff print table summary on startup. Lastly, the fifth returns them to the main menu.

Hash Table Analysis: This section lets the user look at the different hash functions slightly more in depth. They can see the table summary of the dictionary’s hash table. They can also compare hashing functions enabled. This creates a temporary hash table for each implemented hashing function with a 1 string parameter. The tables are built, by using the iterator of the for loop transformed into a string. Therefore, the values stored in these tables are string ranging from “0” to the size of the hashtable-1 as a string. This is where problems turnout because all of these are similar-ish given that each character has only so much as a 9-value difference compared to words which could have a 52 range if only letters. There are more empty cells on these tables than the dictionary, but again the second function is far worse.

Quit: the user can quit the program.

GetTwoEdits Runtime:

It is clear that this function runs in O(c^n). Even more evidence that I should seek a dynamic programming method to solve this issue. But at the end of the day this is an exponential situation since repetitions are allowed and order matters. .



GetOneEdits Runtime:

This if far less obvious and strange looking. The timer stops after all the duplicate words found have been removed. The bunch that took much longer likely had many duplicate words in the list that is built in the function. I was surprised to see an input of length 13 had that short of time. This illustrates the lengthiness of the RemoveDuplicant function and begs for a better approach to hand duplicates. The higher cluster points to linear time (doubling the input does not result in double time). There is a different Get1Editswords function that is used for 2editwords that does not remove duplicates, output data, or check if it is a valid word, the difference is that it is built to be more recursive-friendly for Get2editwords. This function on the other hand includes all of that which throws to much variation in the mix. The reason I did it this way was because I was curious how long the whole function took, not until now do I see differently.

