Report for LAB 3

Leticia Dupleich Smith

Deadline: December 1, 2024 23:59

Task description

The objective of this project was to develop an efficient spell-checking program. The program begins by asking for two inputs: a text and a dictionary. It then processes the provided text and outputs the words that are not in the dictionary, and displays the total count of those words. This project was divided into two parts. In the first part, the provided preliminary code for a basic spell checker used arrays to store and process the dictionary. The problem with this code was that it read statements like "one,two" as one word ("onetwo") rather than two separate words - our task was to fix this issue. In the second part, we had to replace the array-based implementation with tries, which results in a more time-effective code. Thus, the main goal for this lab was to be able to create a successful spell checker that would correctly read all the words in the inputted text and would also be time efficient when longer texts were provided.

Structures and Algorithms

The first part of this lab used arrays as the main data structure. Dictionaries were stored in an inefficient way: using an array of strings. The preliminary code provided also included a custom data structure called *dict* (short for "dictionary") which included two integers, *numWords* and *maxWords*, and a character pointer *words* to store words. The preliminary code had multiple helper functions: *check* (to check if a word is in the dictionary), *addWord* (to add the word to the dictionary), *freeDict* (to free the dictionary) and *trimWord* (to remove non-alphabetic characters and convert words into lowercase). The program also consisted of a main function where the dictionary and text would be inputted and read. Here is where this program ran into the small issue stated previously: it would read "one,two" as "onetwo". To address this issue, the while loop had to be changed to read character by character rather than sequences of characters. Alphabetical characters were appended to the word while non alphabetical characters served to terminate a word. This allowed punctuation marks to serve as word terminators

In the second part of this lab, the code was modified to function with tries rather than arrays, which significantly reduced the time it took for the code to run. The dictionary *dict* was redefined using *typedef struct TrieNode* dict* where a *TrieNode* consisted of a boolean *endNode*, to indicate the end of a word, and an array *dict children[26]*, depicting all 26 nodes corresponding to each letter in the alphabet. All of the helper functions (*check, addWord, freeDict,* and *trimWord*) were changed to function with tries instead. Also, a new function that mapped a to 0, b to 1, etc was implemented to facilitate the new *check* function.The main reason to do this has to do with the time complexity of the two implementations. Using an array would require a complexity of O(kn) where k is the number of words in the dictionary and n is the length of the words. This is because checking for a word requires the traversal of the entire dictionary. In contrast, using a trie approach allows for a complexity of O(n) where n is the

length of the word being checked. This is a much better approach because searching for a word in the dictionary using a trie involves traversing the tree one character at a time.

Evaluation of the program

To test both parts of the project, I used the *make test* and *make longtest* commands as well as CodeGrade. The first program that was implemented with arrays passed the *make test* command easily, showing that it worked for the two provided test cases. However, it took an extremely long time to pass the *make longtest* command and would not pass all of the tests on CodeGrade because of how long it took. However, for the second part of the assignment, all of the tests passed which showed me how much more time effective it was to use tries.

Throughout this project I ran into two main issues. The first one occurred when adjusting most of the code to work with tries. I kept attempting to use **dict* to store words in the trie because that is what was used for the array implementation. Although the *make test* command would still pass, I kept getting the warning "incompatible pointer types." I soon realized that this was because arrays require memory reallocation whenever the size of the data (string) grows. Tries do not require memory reallocation because the memory is handled like a tree: memory is allocated per node. The second issue I ran into had to do with formatting. In the CodeGrade tests, the code would not pass the final test "clang format". I kept trying to change the indentations and line spacing but it would not work. Eventually, I discovered that it was the comments that were causing this. My comments were initially after the line of code which would cause me this issue. I ended up placing the comments before the lines which solved this issue.

Conclusion and Self-Reflection

This assignment helped me to learn more about data structures and how some are more effective than others depending on the situation. While both arrays and tries were options for the spell checker, tries were much more time effective when it came to handling large inputs. It is very interesting to see how even though a code can be done in multiple ways, there is always going to be one way that is more effective. This assignment also helped me to understand the importance of time complexities and how they affect code performance

What I found relatively easy in this assignment was rewriting the functions to work for tries. I based my code on the previously done *triesearch* weekly exercise. That being said, implementing the *addword* function was a bit challenging for me because I could not really visualize how adding a word would look like on a trie diagram. To facilitate this for myself, I drew it on paper and was able to visualize what happened to the trie when a word was added.

This project made me curious about when to use which data structure. Although time complexities hint at which data structure is most effective, in the future I would like to learn how else we can determine which data structure is the most appropriate to use in each situation.