Joshua Koshy

Algorithm Implementation

5th June 2017

Project 1 Comparisons and Summary (W-Section)

Part A of the assignment required me to be able to implement MyDictionary.java and it’s interface, DictInterface.java, in so that I would be able to read dictionary words from a file and form a dictionary with them. This was solved by utilizing the functions of “Scanner”, “File”, and “PrintWriter”. I would use Scanners to read in any input files (in this case being the dictionary.txt and the input file, whose name is stated by the user) and File to refer to the files themselves in the directory. PrintWriter could help me create the output file after my program had finished making the anagrams of the input file.

The first step was to iterate through the dictionary words given and fill up a new MyDictionary.java class with these words. By scanning the methods of MyDictionary.java, it became apparent very quickly that I only needed to use the add method with each line of text from the dictionary text that is given. With the dictionary now filled, I would now need to iterate through all the words in the input file given by the user to place them all in an ArrayList. I made sure to replace any whitespace characters as I iterated through each line/word of the input file to make searching for anagrams easier. Then, I knew that the next step was to iterate through the ArrayList and go through the process of finding an anagram for each word. So, I made a for-loop to iterate through these words. However, I knew that the recursive method to make for this for-loop was going to be the hard part.

I haven’t worked well with recursive methods in the past, but I was familiar with the structure that this method was going to shape to be. I understood that I would be working closely with the searchPrefix method of MyDictionary.java within this method. It would need to check each permutation of the word I’m giving it. I realized that this program wouldn’t work unless I iterated through each letter of the current word and appended the letter to a string to search from prefixes/words within the dictionary. The characters, however, would need to be marked in some way in so that the program knows that the letter/permutation has been tested or is currently being tested. This was the tricky part, but I think I found an nice fix for the problem. Seeing as how we are working with each letter of the word individually as we shift and combine it with the other letters, it seemed that perhaps making a character array of the inputted word would help me edit and shift each characters more easily. And this solution would make it especially easy for me to “mark” the letter’s being tested. And if the tested character is not meshing well with the test string and not forming any prefixes/words to use, I would simply place it back in it’s given array index as well as delete the character from the test string. The index would be tracked by the overall for-loop that is iterating through each letter.

So, a letter would be tested for a word/prefix. So, say for instance, the input is “pnxish”. The character ‘p’ would be tested first. It would be marked and added to the test string to be given to the dictionary. The dictionary would determine whether there exists a prefix ‘p’. In this example, duh. P begins a huge tree of words. So, it signals back that, yes, it is a prefix. However, it also signals that isn’t a word yet. This brought me to the issue of what to do in three cases:

1.) When a string is a prefix, but not a word

2.) When a string is a word, but not a prefix

3.) When a string is a prefix AND a word

Each situation presented new problems to the recursion method, but I tried to focus on one situation first and figure out the issues. The first case seemed simple enough. If it’s a prefix and not a word, then keep going! We simply give our current test string along with the array of letters. I decided to mark any tested characters with an ‘!’ character, as it seemed that no dictionary words would be using such a character. If any inputs were using an ‘!’ anyway, the program should correctly reply with no anagrams as any combination with an ‘!’ doesn’t exist in the dictionary given. An if-statement right inside the for-loop iterating through the characters of the inputted word would be able to skip any marked/tested characters.

With the first case handled, the second and third cases seemed a little trickier to handle. The second case could have a word showing up with leftover characters, meaning that the found dictionary word was smaller than the inputted word. We would need to continue iterating through the other characters to find another word to combine with this already found word. This would allow us to have multiple-word solutions. I made an if-statement for this case and played around with the inputs of the recursive method call before I had it working properly. The second case could also have a tested string have no characters leftover. In this case, we have successfully found an anagram and we add it to our list. And from here, it seemed I was successful at completing the method. I had a few hiccups within debugging when it came to what was needed in the method calls as well as needing to place a break for when the word was done testing.

I’ve had a lot of experience with LinkedLists in the past, so making DLB.java wasn’t as challenging. I stayed to the structure given by the DictInterface. The searchPrefix method gave me a bit of a challenge but splitting up the method in so that only the first call searchPrefix initiated the recursive calls helped a lot. I just had to pay careful attention to the child and neighbors, making sure that I accounted for them when my program searched through the dictionary.

|  |  |  |
| --- | --- | --- |
|  | MyDictionary.java | DLB.java |
| Test1.txt | 2282 ms | 170 ms |
| Test2.txt | 8847 ms | 190 ms |
| Test3.txt | 193283 ms | 329 ms |
| Test4.txt | >200000 ms | 564 ms |
| Test5.txt | >200000 ms | 5143 ms |

A linear search is an immense toll on the performance of this program. We are then limited to an O(N) look-up time, which punishes us when we have more entries within our dictionary. With a DLB implementation with our dictionary, we would need to evaluate the performance by seeing how many valid characters our dictionary’s alphabet would have as well as how many characters our test string would have. Let’s label the number of valid characters, S. And let’s label the number of characters our test string would have, K. We would be making S amount of comparisons within each level, comparing our character to the dictionary’s on that level. With each match, we would need to go through up to K levels in order to see if this test string exists in our dictionary (or at least holds a prefix). So, our performance would be O(KS). This may sound bad until it’s realized that most of the levels will have very few characters held in them. S may not be so big. In the average case, our performance would be O(K) time. A huge improvement over O(N), when N is the number of entries in the dictionary.

I should clarify how the DLB is structured. The nodes of the DLB each hold a character, and a child. It also recognizes it’s neighboring nodes, in so that navigating the DLB is much easier. The child of each node holds the next character of the word that stem of the DLB trie. So the word “apple” would have a stem of 5 nodes, starting with a node holding the character ‘a’. The child would hold the character ‘p’. And THAT child would hold another ‘p’. This repeats on and on until the word is complete. We mark the word’s completion with a special character in my form of the DLB. Neighboring nodes of a node would hold other characters if multiple words use the same prefixes. So the word, “apricots” would stem off the third character in “apples”. The third child, the second ‘p’ in “apples”, would neighbor a new node ‘r’, which would go on to create more children, forming the word “apricots”. With a given string, we can search through these nodes recursively and search a DLB dictionary that is formed from the start of the program. This tree would be filled with many different parent nodes from the start (up to 26 characters). However, as we go down the levels of the tree, we can see that there aren’t as many characters in the levels below each parent. With each prefix in a dictionary word, less and less characters can fulfill a valid prefix. This is the cause of why searching a DLB can be very fast, as the low amount of levels allows us to traverse the tree very quickly and with very few comparisons (compared to that of MyDictionary.java).