Joshua Rodstein

1501 – Summer 2017

jor94@Pitt.edu

**Project 1**: Anagram and De La Briandais Dictionary

The goal of Project 1 was to develop an algorithm to find anagrams of specified words and to check those permutations against words in a provided dictionary file. At first glance it seemed a bit daunting, but after some careful consideration everything I had been exposed to in 0445 started to come back to me. If I had it to do over would attempt to come up with a more eloquent solution, however I feel that despite a small bug, my solutions is very effective. I will attempt to deliver some understanding as to my algorithm and implementation choices with this write up.

For ease of getting started, I decided to use the provided multi-way TrieST provided to us as a handout in class. It took a bit to trace and understand the logic behind its functionality but through some trial and error I came to a decent understanding. My recursive algorithm technically involves three iterations of a basic method. The first is contained within a master program loop within my main method that calls the other 3 recursive methods.

The master loop reads one string at a time, and iterates over its alphabetically sorted characters one at a time, so that all solutions are in alphabetical order once collected. For example… with the string “abcd”, all permutations starting with a, then b, then c, and so on are found and stored. This same logic is reused within the 2 recursive functions with a “for” loop iterating over a suffix which changes length at different levels of recursion. The first method trieBuilder() attempts to find all one word solutions.

trieBuilder() receives a prefix StringBuilder and a suffix StringBuilder. I decided to use the StringBuilder objects for ease of amending the sequences of characters. However for my DLB Dictionary implementation, I decided to use Strings for the search and find methods simply because it required a far less malleable type of data. It was easy enough to extract a character from a String and search for it.

On the first call of trieBuilder(), prefix will contain one character and suffix will contain the rest of the chars from the input string. The process is to take one char from suffix, test whether it is a word or a prefix, and append it to the prefix if it is valid. When the recursive call to triBuilder() is made, I passed the newly appended prefix and the suffix minus the appended character. In essence I move the first char from suffix to the end of the prefix and pass those values to the recursive call.

If the searchPrefix() method returns a 0, then the method returns 0, and move back to the previous state. When the method returns from a recursive call, it immediately breaks and the loop iterates to the next character of the suffix to try the process again, effectively pruning after a backtrack. This proved effective in finding all one word solutions and was working relatively quickly. However, the next and far more daunting task was to implement the multiWordTrieBuilder() which finds all solutions containing more than one word.

I decided to write a new method utilizing the same control flow as the previous, which would receive one more argument. A build StringBuilder. This would represent the current partial solution. The reason for this was that if a prefix was deemed to be a word but not a prefix, a space would need to be appended to build, and prefix would need to be cleared so that a new word could be permutated from the remaining characters in the suffix. The tricky part was figuring how to branch into two recursive calls if the prefix was deemed both a prefix to a larger word, and a word in and of itself. Two calls to the multi word method were made, passing an un-amended prefix to one, and a blank prefix with an appended build to the other. It took some finagling, but eventually I got it working.

My initial tests before implementing the DLB were with MyDictionary, which took an iterative approach to the storing and searching for words within a dictionary structure. Test files 1 and 2 took only a few seconds, with test three taking about 5 seconds to finish. However, test 4 took a couple minutes on average, with test 5 forcing me to test on a faster machine, and still taking about 10 minutes to complete. The DLB Dictionary structure was much faster. Test 5 took about 1 minute to complete on my slow machine, and a matter of seconds on my fast machine.

This is a pretty solid representation of the difference between a linear search without pruning, and a DLB Retrieval with pruning of invalid nodes and paths. If my slow old machine cut the solve time of test 5 by 30 min with the DLB, then the difference must be significant.

Bugs:

I have one bug, which I developed a work around for. My algorithm successfully finds all solutions, however for some multi-word solutions it was adding one or two wrong results. Somehow my algorithm was causing an append of 2 adjacent white spaces which increased the length of the solution. If the length matched the length of the initial string then my sort and print logic would obviously print it out. I was able to omit any solution that contained adjacent white space characters, and it now prints as it is supposed to.