William Kurek

CS 1501

Writing Assignment 1

**The Anagram Problem:**

The anagram problem presented in this project required a recursive backtracking algorithm to compute all possible anagram solutions of a given input sting. The algorithm beings by calling the function findAnagrams() with the following arguments: an empty StringBuilder, which we will call the solution string, and an ArrayList containing the characters of the input string. Also, the current number of words is stored as a global variable. In this function, the algorithm adds a single letter, or character, from the ArrayList to the initially empty solution string. The resulting string is then passed to the searchPrefix() function to determine whether it is a word, prefix, or neither. If the solution is neither a prefix or a word, the algorithm tries the next character in the data structure. If the given string is a prefix but not a word, a recursive call is made, repeating the process with the prefix as the solution string and the prefix characters as the new input string. If the solution string is a word, there are two other cases to consider before proceeding. In the first case, if the solution string contains all the characters, the string is then added to the anagrams solutions array. In the second case, if the solution string does not contain all the characters, a space is appended to the solution string. Then, another recursive call is made with the current string and the characters remaining from the input string. The number of words, implemented via a global integer, is incremented as well. Since the number of words is greater than one, this recursive call requires a different course of action, where the function searchPrefix() should only operate after the appended space on the recursive call.

**Runtimes:**

Rather than calculating the runtime of each test file via a stopwatch, I implemented a simple timer within the program which outputs the runtime of the algorithm. But, because test files four and five required several hours to execute on the MyDictionary implementation, runtime approximations, given previous results, were necessary.

|  |  |  |
| --- | --- | --- |
| **TEST FILE RUNTIMES (milliseconds)** | | |
|  | **MyDictionary** | **DLB** |
| **Test1.txt** | 2422 | 219 |
| **Test2.txt** | 8301 | 235 |
| **Test3.txt** | 214658 | 360 |
| **Test4.txt** | 621493 (estimated) | 641 |
| **Test5.txt** | 1228806 (estimated) | 4521 |
| **Test6.txt** | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | 1060415 |

After thorough analysis of the test files and their respective runtime, it was apparent that multiple variables influenced the results. First, the number of words in the dictionary has a direct impact on the dictionary load time. Also, the numbers of words could also have a negative effect on the search time. In the worst case, it is possible that each level of the DLB must be traversed, which would significantly increase the asymptotic runtime. which could have a negative effect on the search time. But, for the MyDictionary class, the asymptotic runtime is directly impacted by the number of words in the dictionary regardless the case. Second, the number of characters in a word also influences the runtime. Depending on the dictionary, an increased number of characters has the potential to increase the amount of multiple word solutions. Third, the number of solutions also has the potential to increase the runtime of the algorithm. For example, sorting the solutions at the end of program would increase the asymptotic runtime, because sorting is not executed in constant time. So, many solutions will have a negative impact on the runtime of the program. Finally, the number of words in a solution effects the runtime of the program as well. As the number of words in a solution increases, the number of repetitions, including recursive calls, of the algorithm increases as well. Adding up the recursive calls asymptotically will give us the overall runtime. However, this is bounded by the length of the string, as a string of length N can at most have N words to it. Therefore, in the worst case, the runtime is multiplied by N.

**Implementation Comparison:**

Although the MyDictionary and DLB versions of DictInterface follow a similar structure and commute the same output, their respective implementations are, in various regards, quite different.

In each implementation, data is stored different ways, which directly influences memory consumption. In the MyDictionary implementation, data strings are stored in a sorted ArrayList whose size is equivalent to the alphabet size. In the lower levels of the data structure, there are many unused pointers. As a result, as the alphabet size grows, the amount of memory used increases as well. In the DLB implementation, data entries are stored in a linked list. Each entry of the linked list contains a character, a child reference node, and a sibling reference node. Rather than storing all possible alphabet pointers, the DLB only stores pointers that are used in the linked list, which can save a significant amount of memory in comparison to MyDictionary.

Further, MyDictionary and DLB implement different prefix search methods. In the MyDictionary implementation, the algorithm iterates through the ArrayList until iterates until the end of the list, or until the key is found. Then, it iterates through the key and the current String in the ArrayList character by character. If the end of the string in the ArrayList has been reached, the algorithm stops with that string, and proceeds to the next one. If the current character in the key is less than the current character in the string, the search is terminated all together. Otherwise, the algorithm determines whether the characters match, and if it has reached the end of the key and/or string and respective Booleans are set. As a result, the algorithm can determine if the string is a word, prefix, both, or neither. In comparison, the DLB implementation utilizes recursion in the search method. At each level of the linked list, the algorithm follows the right sibling references of each node until the character is matched or until NULL is reached. If the character is matched, the algorithm proceeds to the next level of the linked list. The DLB proceeds down the levels of the structure until the size of the string matches the size of the key. If the child reference of the current node is not null, then the string is a prefix. If the end of the string character matches the end of the key and the sibling reference of the current node is not null, then the string is a prefix and a word. Otherwise, the string is only a word. As a result of this implementation, the search time at each level is potentially linear in regard to the number of characters in the alphabet. Since the number of characters in the key is equal to the number of levels necessary to get to the end of the key, the potential linear run time at each level can significantly affect the overall runtime. But, since this is worst-case runtime, it is highly unlikely. Since most of the levels in the DLB implementation will have very few characters, the overall runtime for search is equal to the number of characters in the key, in the average-case.

Although it has a worst-case runtime of O(SK), where S is the size of the alphabet and K is the size of the key, the DLB implementation of DictInterface is superior to MyDictionary in testing for prefixes. In most cases, DLB will have a runtime of O(K), which is significantly better than that of MyDictionary. Also, since DLB only utilizes necessary pointers, it requires far less memory as well. Since the MyDictionary implementation requires a fixed number of pointers per level depending on the size of the alphabet, DLB is more efficient is regards to memory as well. Overall, DLB is the superior algorithm in comparison to MyDictionary.