**Forbes Turley CS 1501 A1 Write Up**

**Anagram.java**

My solution begins by storing the dictionary into either a DLB data structure or the provided “MyDictionary.” It then reads in, line-by-line, the file that contains the phrases to solve. The program removes all spaces from this input (since spaces do not count as a character for the purposes of finding all possible anagrams) and passes this as a StringBuilder object into the “tryString” function.

The “tryString” function recursively creates permutations of the string, and through backtracking avoids trying strings for which no suffix exists. It takes as arguments three StringBuilder objects: one containing characters that have not yet been placed into a solution string to form a word or words, one containing a word that is still in the process of being formed, and a third StringBuilder containing complete words that have been created from the input characters. As tryString adds characters onto the string that it is forming, these same characters are removed from the StringBuilder containing characters that are still unused. When tryString finds that a word it has formed is a complete word (and not just a prefix) it will try solutions containing this word as a whole followed by a space followed by permutations of the remaining characters. If this word is also found to be a prefix to other words, it will additionally attempt to find these words by creating permutations of the remaining characters added directly onto the end of the word.

To prevent unnecessary time from being spent creating strings that will never yield a valid word, tryString ceases attempting to permute onto a string if it is not a valid prefix. For each string, tryString first sees if there are any remaining characters that must be used (since a valid anagram uses all available characters). If there are not, the word is already complete and may be added to the solutions. If there are characters remaining, the function creates string by adding on a single one of each of the remaining characters to the current string. If this is found to create a valid prefix, tryString is again called with this new string as the current string. If it is instead found that this new string is itself a valid word, this whole word is appended to a string containing fully formed words, followed by a space. If this string is also a prefix, tryString is called once again with this string as the string to build upon. If the string is neither a valid prefix nor a valid word, no further attempts are made to add letters to it. Regardless of the outcomes of these trials, the string is restored to its original state at the end of the trial by removing the character that was added to the end, and the string that contains the unused characters has this same character added back to it.

**DLB.java**

The DLB class has an inner class named “Nodelet” that simply stored a character, a reference to a child Nodelet, and a reference to a right sibling Nodelet. The DLB keeps track of only a single top level Nodelet named firstNode, which then maintains references to children and sibling Nodelets containing the characters of all added strings. To denote that the parent Nodelet of a Nodelet is a complete word, its character value is set to ‘$’. The DLB class supports the operations boolean add(String), boolean remove(String), int searchPrefix(StringBuilder), and int searchPrefix(StringBuilder, int, int) (where the int parameters indicate subindices of StringBuilder object). The add operation transverses the DLB beginning at the first Nodelet, searching for each character in the StringBuilder object. At each level, if it cannot find the desired character, it adds it as the final right sibling in the level. If the current Nodelet has no child, it adds the next character as the child to the Nodelet. When it reaches the end of the StringBuilder, this operation is similarly performed for the ‘$’ character to denote the end of the word.

The searchPrefix(StringBuilder) performs a very similar operation. For each character in the StringBuilder, the function searches the DLB starting at the firstNode for the character. If any character is not found, the function returns 0. If the final character is found and the child level contains a ‘$’ as well as another character, a 3 is returned. If the child level contains only another character, a 1 is returned. If the child level contains only a ‘$’, a 2 is returned. The searchPrefix(StringBuilder, int, int) function simply creates a new StringBuilder object consisting of only the characters in the StringBuilder passed in between the two int indices, and passes this StringBuilder onto the previously described searchPrefix function.

The remove function is considerably more complex than the add function. It first determines if the last level of the word to be removed by calling the searchPrefix on the word passed in. If so, the function simply removes the ‘$’ on the level of the DLB following the last character in the string. If the word to be removed is not a prefix however, the function must determine the lowest level in the DLB of the characters within the string for which there are two or more items in the level. To to this, the DLB maintains references to the level before the last level at which two or more items were found as well as the character in the string that was found at this level, and updates these references whenever a lower level is found. Following the complete transversal of the string down the DLB, the function removes the reference to the character at the final level at which two or more items were found. Because this Nodelet could potentially be the first Nodelet on the level, the remove method actually begins attempting the removal by examining the child of the level above the level where two Nodelets were found and then examines the subsequent right siblings until the character of interest is found, and the Nodelet pruned out. All other Nodelets on the level are maintained, which means that if the Nodelet removed had right siblings, the reference to the right sibling becomes the new right sibling reference of the previous Nodelet on the level, or the child reference of the Nodelet a level above.

**Runtime Analysis**

Shown below are the runtimes for each of the test files using either the DLB or MyDictionary. Note that the times are in milliseconds.

|  |  |  |
| --- | --- | --- |
|  | DLB | Orig |
| Test1 | 48 | 8981 |
| Test2 | 69 | 27056 |
| Test3 | 274 | 558099 |
| Test4 | 757 | 1320706 |
| Test5 | 8741 | 17554909 |
| Test6 | 2788240 |  |

The DLB trie is approximately 2000x faster than the MyDictionary, with an intercept of -100 seconds. In both the DLB and MyDictionary, the same number of lookups were made, since the Anagram class pruned results in the same fashion using both types of dictionary. The runtime of the DLB itself it at worst O(N\*K), where K is the length of the string that is being permuted and N is the number of words in the dictionary. However **practically** it is O(K) This is because for each character, several comparisons may be necessary, but it will be far less than N for any normal dictionary. Furthermore, fewer than K comparisons could be necessary depending upon if the string is found to have no suffix. With the MyDictionary, the speed is dependent upon the size of the dictionary. Because the MyDictionary must iterate through the ArrayList which contains the dictionary, it is linearly dependent upon the size of the dictionary as well as being linearly dependent on the size of the word which is being looked up, making it O(N\*K) where K is the input length and N is the size of the dictionary. The runtime of the Anagram program is at worst O(K!) where K is the length of the input string. However, this runtime would only occur if no pruning took place meaning that every string formed was either a prefix of a full word. This is unrealistic, and practically the runtime will be far better. Its speed is also dependent upon the type of dictionary being utilized, and the type of data structure used to store the results. This makes anaysis of the runtime complex, as longer strings become slower both because of the greater number of results that must be stores and the number of permutations of the string that are possible.