Overview:

**getCandidates**

This method takes in a prefix, creates a list of strings and find the node that corresponds to the last letter of the prefix that we are searching. If that node is null it returns an empty list as there are no words starting with that prefix. Then I created a length comparator engine to sort my list. If node is not null that means we have words starting with that prefix. To find those words, I needed to traverse the entire children map to find those words. That’s why I created a traverse method. After traverse method adds every word starting with that prefix to the list we created, I sort the list according to length and alphabetical order. After this we have a very important if statement that dictates and puts the most recent word choices at the top of the list, so that most recently words come up first. The list in this if statement that we call through node.getValue is different than the list we used initially in this method.

After that we check our modified list to see if the size is greater than 20. If it is we create a sublist of the first 20 words.

**traverse**

Traverse method takes a node, prefix and a list of strings as a method and checks that the current node is at an EndState. If it is that means we have a proper word, so it adds this word to the list. If not, we traverse the entire childrenMap of that Node, using the keys, and change the prefix to prefix+key so that we can move forward in the children map (i.e. she+l= shel, shel+l=shell and so forth). After we traverse the entire children map, all the words that start with that original prefix that we passed to this method will have been added to the list.

**pickCandidate**

Just like getCandidate we find the node that corresponds to the last letter of the prefix. Then we run an if statement to see if the candidate word exists. If not we add the candidate word to our dictionary using put. Then we check, assuming the candidate word starts with the prefix, if it’s been used before. If it is, we remove the candidate word from that recently used words list so that the word does not show up twice and finally add it to the first place of that list.

Here, the list refers to the most recently used words list. We create this list and add the candidate word to the top of the list and put this list to the value type of our node. Eventually, we switch to this list in the getCandidates method. This is how we implement the most recently used words feature.

Analysis and Observations

My program provides a couple more features to make this project more realistic. Firstly, if you start with a prefix, say “tame” and try to pick the word “bro”, since “bro” is not in the list and does not even start with “tame”, which is by default something that no user would do (as you switch to another word different list comes up – not doable here by the nature of the program), I print "This is not what you intended to type" and do NOT add the word to the dictionary even though the test program prints that it is learned. You can also NOT add a word if it is shorter than the prefix, as it is also not logical. These features are implemented using startsWith method in Strings. However if you add something that is not in the list but BEGINS with the prefix, it gets added to the dictionary and you can even search with the prefix “tam” and still be able to find that word(assuming it falls under the top 20) even if you pick it after the prefix search “tame”.

The heart of the program is the fact that we use a list to keep track of the words and then update that list as we pic words by displacing the word from its actual place to the first place in that list. If a word does not exist but gets picked, then it’s created and gets put in the first place.

\*I have tested my program using RunAutoComplete.java

TAMER AVCI

#2105024

I worked on this assignment alone.