In order to set up this problem, I made use of a DLBNode private inner class, StringBuilder currentPrefix, DLBNode currentNode, StringBuilder currString, and DLBNode tempNode. The DLBNode class was used to store each letter in the DLB dictionary. The currentPrefix kept track of the current String based on the user input. The currentNode kept track of the node in the dictionary which contained the last letter of the currentPrefix. If the currentPrefix was not in the dictionary, the currentNode was simply set to null. The tempNode kept track of the node containing the last letter of the currentPrefix that was in the dictionary. The currString represented the longest string part of currentPrefix that was in the dictionary. If curentPrefix was in the dictionary, currString was equal to currentPrefix and tempNode was equal to currentNode.

I implemented my add method in a very similar manner to the put method from the DLB lab. To summarize, the add method made use of a private helper function. In this helper function, I continued to traverse down the DLB trie, recursively calling the add method on either the child or the child’s sibling of the previous node until we had traversed down the length of the string being added. If the node was null, I created a new node and added it on. If the node already contained the appropriate letter, we could traverse from that node. Otherwise, if the child node does exist, but does not contain the correct data, we need to traverse through its siblings until we reach null or find a sibling node with the data. If we reach the case where the word already exists, we can return false. In this way, each word was added to the dictionary.

In my advance method, we were tasked with appending a character to the currentPrefix, and returning true only if the currentPrefix after appending was a word. In order to implement this method in O(1) runtime, I traversed from the position in the dictionary of the last letter in currentPrefix, instead of traversing from the root of the DLB and checked if the isWord field of the node after traversing to the character, was true or false. However, this implementation was not so simple. There were several special cases I had to take into account. If we had an empty currenPrefix, I had to reset the values of the variables as well/ From there, if the root data matched with the character, I could set currentNode to be the root. Otherwise, I traversed through the siblings of the root, and set currentNode to the node which matched the data if there was any. If the currentPrefix was equal to the currString, this meant that the currentPrefix was in the dictionary and that the currentNode would not be null. In this case, we simply traversed to the child or the child’s siblings until we found a node that had the character, or we traversed through all of the siblings without finding a node with the character(in which case we set currentNode to null). There was another case where the currentNode was initially null. This would have happened if the currentPrefix was not in the dictionary. In this case, we can simply append on to currentPrefix and return false without having to make any traversals through the DLB. Additionally, it is important to note that tempNode, currString, and currentNode updated differently for each case, and I had to make sure to update them accordingly. However, regardless of which case the program went through, currentPrefix was always updated by appending on the character.

In order to implement the retreat method, I utilized a similar implementation as the advance method but backwards this time. Instead of appending the character on, I deleted the last character of currentPrefix-and possible currString. If the currentPrefix was longer than the (currString + 1), this meant that the currentPrefix was not in the dictionary even after deleting. Furthermore, in this case, I only had to update currentPrefix without updating the currString or the tempNode. However, if currentPrefix was not longer than the (currString + 1), I had to traverse back up the dictionary to update the tempNode and currentNode. When traversing, if the current node was a sibling node, I had to traverse backwards using previousSibling. Once I reached the direct child node, I could traverse upwards to get to the parent node.

In the getNumberOfPredictions, I simply returned the size of the currentNode. The size variable was maintained in the very beginning with the first add method. In the add method, whenever I created a new node, I set the size to 1 for the node. And whenever, I traversed down to an already existing node with the same character I needed, I incremented the size of that node because the same node was being used for an additional word in the dictionary. When implementing the retrievePrediction() method, I used a similar implementation as the get method from DLB.java. However, instead of searching for a specific key, I was now searching for a node where the isWord field was marked True.

There were two main overarching challenges I faced in this assignment. The first challenge was understanding how to implement the advance method so that it would be in O(alphabet size) time. Initially, I was simply calling the getNode method inside advance to obtain the currentNode. However, the runtime of the advance became too high because the getNode was starting from the root and traversing the entire length of currentPrefix until it found the end currentNode. In order to resolve this issue, I decided to not use getNode at all. I started the traversal from currentNode(instead of the root), and now simply had to traverse down one node in order to obtain the currentNode. The second issue I faced was understanding how to retreat if the currentPrefix was not even present in the dictionary. Initially, I was updating the currentNode if the currentPrefix was not in the dictionary. But this was obviously causing issues because currentNode was null. In order to resolve this issue, I created two additional variables-tempNode and currString. tempNode kept track of the node with the last letter of currentPrefix that was in the dictionary. And currString kept track of the longest string from currentPrefix that existed in the dictionary. I was then able to use these variables in my retreat method where I updated currentNode only if the currentPrefix was actually a word after deleting the last character.

|  |  |  |
| --- | --- | --- |
| Function | Worst-case Runtime | Explanation |
| add(String word) | O(alphabet size\*word.length()) | This is the worst-case runtime I implemented because when adding each word, the number of levels we traverse down is equal to the word length. But additionally, we need to take into account that for each level we traverse down, we may need to traverse through all the siblings before reaching a node with the character we need or before we reach null.  I was able to implement my program with this runtime |
| advance(char c) | O(alphabet size) | My program had this worst-case runtime because when we advance by a single character, we start the traversal from the last node of the currentPrefix(currentNode). This allows us to traverse down to the child node in O(1) time. However, we also have to consider the worst case where we need to traverse through all of the siblings of the child in order to reach the appropriate node.  I was able to implement my program with this runtime |
| retreat() | O(alphabet size) | My program used this worst-case runtime. The logic behind the runtime of this program is very similar to that of the advance method. When traversing up a node in worst case, the program needs to traverse through all of its previous siblings before it finds the child node after which it can traverse upwards to find the parent node.  I was able to implement my program with this runtime |
| reset() | O(1) | My program implemented this function in O(1) time because we simply reset the currentPrefix and set currentNode to null which can both be done in O(1) time.  I was able to implement my program with this runtime |
| isWord() | O(1) | My program implemented this function in O(1) time because we simply return true if the isWord field of currentNode is true, else we return false.  I was able to implement my program with this runtime |
| add() | O(alphabet size\*length of currentPrefix()) | This is the worst-case runtime for the same reason as the worst-case runtime for add(String word). The only difference is that we are now taking into account the length of currentPrefix specifically because that is the length of the word we are adding into the dictionary.  I was able to implement my program with this runtime |
| getNumberOfPredictions() | O(1) | My program implemented this function with this worst-case runtime because this function obtained the number of predictions by simply returning the value of the size field of currentNode.  I was able to implement my program with this runtime |
| retrievePrediction() | O(prediction.length() - currentPrefix.length()) | My program implemented the function with this worst-case runtime because traversal starts from the last node of the currentPrefix(currentNode) and continues to traverse down levels of the DLB until it reaches a node where the isWord field is marked true and this word is then returned as the prediction. Furthermore, the number of traversals through the DLB is the length of the prediction word minus the length of the currentPrefix.  I was able to implement my program with this runtime |