ECS529U Coursework

Benjamin Robson - 170140692

```python
def count(self, st):
    if st == "" or st is None or self.root is None:
        return None

    return self._countRec(self.root, st)

def _countRec(self, node, st):
    if len(st) == 0 or node is None:
        return 0

    head = st[0]
    tail = st[1:]

    if head < node.data:
        return self._countRec(node.left, st)
    elif head > node.data:
        return self._countRec(node.right, st)
    else:
        if node.mult >= 1 and len(tail) == 0:
            return node.mult
        return self._countRec(node.midl, tail)
```

Listing 1: Count function in word tree

1 Searching and Counting in Word Trees

A word tree is a data structure specifically meant for storing strings in a tree structure. Word trees are a form of ternary search trees which are essentially binary search trees with the exception of node's having 3 children (3 pointers to 3 child nodes) instead of 2 pointers and the nodes contain characters instead of numbers. Word trees begin at the root, where nodes are added to the root. Each node in the tree contains 3 pointers (left, right, midl), a single lowercase character of the English alphabet, and a multiplicity. The multiplicity denotes two things: whether the node is the last character of a word in the tree, and the number of times that word exists in the tree. The node's pointers work with the following logic: pointers to the left are lexicographically smaller, pointers to the right are lexicographically greater, and pointers in the middle are the path of that word which the character is a part of. If a node has a pointer to the left and/or to the right, it means that the node's character is not a part of those words, and therefore the word may trail to either the left or the right skipping that node's character. Word trees are particularly useful for text autocomplete and suggestion. For example in cases where the user has typed a partial word such as "abs", the word tree could determine that the word the user is trying to type is "absolute", "absinthe" or "abstinence" based on the given prefix.

Word trees can be searched for a given string (word) by maintaining two pointers: one following the current character in the string, and one pointing to a node in the tree beginning at the root. Thereafter, we make decisions for the direction of our traversal. If the string pointer is the same as the node pointer's character, then we have a match and continue to the next node, the midl pointer. If the characters do not match, we compare the characters: if the current string pointer is smaller than the node pointer's character, then we move left otherwise we move right. We continue this process until 3 conditions hold: the string pointer has reached the end of the string, the node pointer points to the node holding the same character as the string pointer and the node has a multiplicity greater than or equal to 1. If we do not fulfill these conditions, then we know the word is not in the tree. Searching on average has the time complexity of O(log(n)), whilst the worst case is O(n), but in general the time complexity varies significantly depending on the words in the tree, and works most efficiently with short words, which share similar prefixes.

The method for counting the number of occurrences of a word in the word tree is done by first searching for a word in the tree, as outlined earlier with the traversing algorithm, and then checking the multiplicity of the last node and returning it. Outlined in Listing 1, I chose to approach the problem recursively as trees are naturally recursive data structures - trees are built of trees. The main function count checks whether the string exists in the tree and then does a call to the auxiliary function _countRec which returns the multiplicity of the word. The function _countRec traverses the tree and simultaneously iterates over the word passed to it. This is done by maintaining a pointer to the current node in the tree starting from the root, and with the same traversing algorithm seen before, either changing to the middle, left, or right pointer. The string is iterated by keeping track of two things: the head (the first character in the string) and the tail (everything proceeding the head). During each recursive call, we compare the head which is the current character, with the current nodes left, right, and middle pointers. If we move to the middle pointer, then we pass in the tail of the string to the next recursive call because we have found a node which shares the same character. Otherwise if we move either to the left or to the right of a node, we have not found a matching character and therefore we pass in the same string as was passed earlier, to the next recursive call. Essentially we are cutting the string shorter and shorter until it has no characters left. Once the tail of the string is empty (there is only one character remaining in the string which is the head), and the current nodes multiplicity is greater than or equal to 1 (we have at least one occurrence of the word in the tree), we return the multiplicity of the node, which is the value returned by the main count function.