ECS529U Coursework

Benjamin Robson - 170140692

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

  if not self.root:
      return 0

  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 having 3 children (3 pointers to 3 child nodes) instead of 2 pointers and the nodes contain characters instead of numbers. Word tree's begin at the root, and nodes (characters) 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 the 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 character is not a part of the word, and the word trails to either the left or the right. 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".

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. 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).

The method for counting the number of occurrences of a word in a tree is done by first searching for a word in the tree, as outlined earlier, and then checking the multiplicity of the last node (character), and returning it. As outlined in Listing 1, I chose to approach the problem recursively as trees are naturally recursive data structures. There are two functions, one is the main call for count, simply checking whether the string exists in the tree, and then calls the recursive auxillary function _countRec, which then returns the multiplicity of the word. The implementation in Listing 1 tracks the the current character of the string in the variable head, which always points to the first character in the string passed to the recursive function. The string passed to the function only changes if we move to the next middle node (the midl pointer). The string is cut in such a way that the first character is removed (the head), and the rest

of the string is passed to the next call (the tail of the string). Otherwise if we go left or right, we pass the same string as we had in the previous call because we have not read a character from the string or from the tree. This achieves two things, firstly we are able to maintain a pointer to the current character in the string we are searching, and second we are able to maintain a pointer to a node in the tree. 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 occurence of the word in the tree), then we return the multiplicity of the node, which is the value returned by the main count function.