INFO1105 Report

This report is for explaining how to do the Assignment2 through introducing the Trie data structure and introduce the implementations and methods that are created and are used for the Assignment2. Additionally, this report would also analyze the runtime of the methods and the test cases, which are created by the individual.

Interface Methods:

1. **preTraversal:**
   This is a function that used for searching keys in the Trie structure and further store the key-value pairs in the HashMap. It sends two parameters which are "(Node) root" and "(String) prefixs" when calls the function. Once it traverses all the nodes, it returns the hashmap. This function is called by six functions, which are "size", "get", "countKeyMatchingPrefix", "getKeysMatchingPrefix", "countPrefixes", "sumKeyLengths". The function will search the keys in the Trie structure, plus the key-value pairs will be stored in the hashmap. Hence the complexity of this function will be O(n) in the worst case.

2. **isEmpty:**
   This function is just to check whether the nodes store keys or not. Since it just need to check if the selected node is empty or not, the time complexity of it would be O(1).

3. **indexGenerator and characterGenerator:**
   These two functions are used for generating the index/character for each key-character (A, C, G, T). In the indexGenerator function, using if statement to flag each character's index (A = 0, C =1, G = 2, T = 3). If key-character is not A, C, G or T, then its index would become "-1". In the characterGenerator function, using if statement to declare each index key-character (0 = A, 1 = C, 2 = G, 3 = T). If the location is not 0 ,1 ,2 or 3, then its key-character would become "N". These functions do not need to traverse the structure. Hence, the time complexity of these are O(1).

4. **size:**
   This function is an Assignment private variable, which will start initially 0 when constructor was called. When put is called and new key was added, size will increases by 1. When remove was called and a key was deleted, size will decrease by one.

5. **get**:
   It is used for getting the value which corresponding to the key. It sends the parameter "(String) key" to find the matched value in the Trie structure. Since we have to find the key we want. We need to call "preTraversal" to search the keys in the Trie structure to return the hashmap containing key-value pair. If the key we found match the key we want, then the function will return its value. Since

we have to traverse the structure to find the matched key, the time complexity of it is O(n1+n2) = O(n) in the worst case.

6. **remove:**
   It sends the parameter "(String) key" to find the matched key in the Trie structure and further remove the corresponding value. To find out the matched key, we have to check the characters which build up the keys. Hence, we loop starting from root checking if the first character in the key exists in its child, and if it does, go down a level. We use the indexGenerator function to assign index to corresponding character for comparison. Moreover, we also check and keep track of the first node that has prefix_num equals 1 because that's the first unnecessary node that we could delete if we found the key. Once the key goes to the last character, it will start by going back to the node with prefix_num equals 1 and point the child to null.

7. **put:**
   This function is about inserting the value into the structure with the corresponded key. To achieve this function, we start from the root and go down to its child and check its child if there's the character from the keys. If not found, create the node and add the characters to its child. We will call function "indexGenerator" to create the right index for each character of key to search. The method used a for-loop and if statements to achieve the method. Hence, the time complexity of it is O(n).

8. **containPrefix:**
   We use this function for getKeysMatchingPrefix function, to compare prefix and target string, first if prefix size is larger and not equal return false, otherwise return true.

9. **countKeysMatchingPrefix:**
   It is about counting the number of the keys that built up by the selected prefix. To implement this function, we first get complete the getKeyesMatchingPrefix function, which will return a list of keys matching prefix and get its size.

10. **getKeysMatchingPrefix:**
    It is about collecting the keys that start with the given prefix. We have to traverse through the whole structure by calling the function "preTraversal" to get all the existing keys in the Trie. To find out the keys that contains the selected prefix, we call function "containPrefix" to check any key that includes the prefix we want. When we find a key, we will store the key in the arraylist and further return the arraylist. Since the function is using loop of hashmap to traverse the keys. Hence, the complexity of the function is O(n*m) in the worst case. (m is the length of prefix)

11. **sumKeyLengths:**
    This function is about to return the sum of the length of all keys in the structure. To achieve it, we just need to traverse the whole structure using preTraversal function which will return hashmap and we can add the size of all keys which contains not null value. Since we just need to traverse the structure, the time complexity of the function is O(n) in the worst case.

Testcase:

1. **testsize():**
   It is to test whether the number of the keys that have been stored in the structure is the same as what we expect or not. It also tests whether the size would be affected by the remove function or not.

2. **testget():**
   It is testing whether the output value is the same as the one we expect if we call function “get” with the corresponded keys. It also tests whether the output value will be null if the keys we called are not in the structure.

3. **testput():**
   Testing whether the key-value pairs are really inserted in the structure or not. We can test the result through calling function “size”, “countKeysMatchingPrefix”, “getKeysMatchingPrefix” and “sumKeysLengths”.

4. **testremove():**
   Testing the whether values are really removed from the structure through calling function “size” comparing the size of structure is the same as what we expect. It also tests what will the size of the structure be if we remove the end of the keys.

5. **testRemoveNstuff():**
   Testing the whether values are really removed from the structure through calling function “countPrefix” and “sumKeyLengths” comparing those outputs of structure are the same as what we expect.
6. **testcontainPrefix():**
   This is testing whether the prefix we call match the target keys or not, and also tests whether the function is useful or not.
7. **testcountKeysMatchingPrefix():**
   This is testing whether the number of keys that corresponding to the selected prefixes are the same as what we expect or not. It also tests whether the output of it will be affected if the last keys are removed or not.
8. **testgetKeysMatchingPrefix():**
   This is testing whether the collection of the keys that corresponding to the selected prefixes are the same as what we expect or not. Since the method will not print out integer type of output, we will also call function “size” to check the number of the keys we collect.

9. **testcountPrefixes():**
   This test case is testing whether the number of the possible prefixes due to the prefixes in the structure will be the same as the output we expect or not.
10. **testsumKeyLengths():**
    This is testing whether the sum of lengths of all keys will be the same as what we expect or not.
11. **testException():**
    This test case is testing whether the output will be “IllegalException” if the prefixes or the values are empty. It also tests whether the output will be “MalformedKeyException” if the prefixes contain any character other than A, C, G, T or not.