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 created 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.

1. **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).

1. **indexGenerator and characterGenerator:**

These two functions are used for generating the location for each key-character (A, C, G, T). In the indexGenerator function, using if statement to flag each character’s location (A = 0, C =1, G = 2, T = 3). If key-character is not A, C, G or T, then its location would become “-1”. In the characterGenerator function, using if statement to declare each location’s 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).

1. **size:**

This function is about to check the number of keys are stored in the Trie structure. In the function, we are calling function “preTraversal” to search the keys in the Trie structure through doing the loop of the hashmap. Once we find key, we would count it and flag the number of keys we have found in the variable “number\_of\_key”. Since we have to traverse each node in the structure to look for the keys, the time complexity of this function is O(n) in the worst case.

1. **get**:

It is used for getting the value which corresponding to the key. It sends the parameter “(String) key” to find the matched key 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 through the same way as size function. 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(n) in the worst case.

1. **remove:**

The function is about to the remove the value that corresponding to the selected key. 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 and further remove the value, we have to check the characters which build up the keys. Hence, we need the for-loop to run through each key-character. We also need to use the created function “indexGenerator” to create location for each key in order to conveniently search the character. If the key we selected that would affect other keys, we will only remove the value. If the selected key is at the end, then we will remove the whole node which contain the key and value because it is useless node. Since there is only one for-loop to traverse the key, the time complexity of it is O(n).

1. **put:**

This function is about inserting the value into the structure with the corresponded key. If there is a value which has same key, the new value will replace the old one. To achieve this function, we have to find out the characters that build up the keys through doing for-loop to run through each key-character. Same as “remove” function, we have to call created function “indexGenerator” to create location for each key in order to conveniently search the character. The method is using a for-loop and if statements to achieve the method. Hence, the time complexity of it is O(n).

1. **containPrefix:**

This a boolean type of created function. This method is used for checking whether the prefix we want matches the prefix that the structure includes. It sends two parameters “(String) prefix” and “(String) target”. If the length of the prefix is greater than the length of the target, then it would return false. Else it would run through the characters of prefix by doing for-loop. It would return false unless the prefix we want match the target prefix that the structure has. The method uses the for-loop to check the characters of prefix, the time complexity would be O(n).

1. **countKeysMatchingPrefix:**

It is about counting the number of the keys that built up by the selected prefix. To count all of the keys that start with the given prefix, we have to traverse through the whole structure by calling the function “preTraversal” and further doing loop with “preTraversal”. To find out the keys that contains the selected prefix, we call function “containPrefix” to check any key that includes the prefix we want. Each matched key we found, we will count it. Since the function is doing the loop of hashmap, the time complexity of the function is O(k).

1. **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” and further doing loop with “preTraversal”. To find out the keys that contains the selected prefix, we call function “containPrefix” to check any key that includes the prefix we want. As we find the keys that contain the given prefix, we will store these keys 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) in the worst case.

1. **getKeysMatchingPrefix:**

This function is about counting the number of the unique prefixes that possibly have depends on the prefixes included in the structure. We call the function “preTraversal” to traverse the whole structure. At the same time, we also need to find the keys we want, so we do the other for-loop to store them into an arraylist and further return the arraylist. Since we need to traverse the structure and store the keys we need in the arraylist, we use double loop to achieve the method. Hence, the time complexity of this function is O(n) \* O(k) = O(k\*n).

1. **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 and count the length of all keys through calling the function “preTraversal”. 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 about 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.

1. **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.

1. **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”.

1. **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.

1. **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.

1. **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.

1. **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.

1. **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.

1. **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.

1. **testsumKeyLengths():**

This is testing whether the sum of lengths of all keys will be the same as what we expect or not.

1. **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.