Discovering Affixes Automatically

Claudia Borg

November 16, 2016

1 Introduction

One of the components in computational morphology that we will delve into during this course is that of automatically acquiring 'morphological' information. In this assignment, we will be looking specifically at establishing whether some string sequence can be considered an affix or not by using transitional probability. In order to facilitate this task, we will be using a data structure called a Trie[1] (pronounced "tree", from the word re-trie-val, or "try", for differentiation), which provides us with an efficient implementation both in terms of storage of data and as well quick access of data for our queries.

The Trie is a type of tree structure whereby each node represents a single character. A sample is shown in figure 1. The root node is a symbolic node that is used as a point of entry to the whole structure. From the root, we can then access all the different children. Each node can have several children, but each must be unique — for instance, the root node can only have one child representing the character 'a'. All words starting with 'a' will share this node. In our example below, the whole string sequence 'report' is shared by several words, and then it branches off to continue representing the remaining words, 'reporter', 'reportable', etc.

We will be using a word list, together with frequency counts, complied from a corpus[2]. This list will be in the form of a text file, with one word per line, together with its token count[3]. The first part of the assignment will focus on the basic building blocks to get the Trie data structure up and running in a rudimentary shape. The second part of the assignment will

[1]For a more comprehensive explanation of a Trie Dictionary refer to `http://en.wikipedia.org/wiki/Trie` as an initial description. Last accessed 5-Dec-2016

[2]Frequency lists will be given to you for both English and Maltese. You can choose to work with either one, or to try and use both. The Maltese list is taken from the Maltese Language Resource Server (MLRS) Corpus which is also available through an online interface `http://mlrs.research.um.edu.mt/index.php?page=31`

[3]Token or Frequency count refers to how many times a particular word appears in the whole corpus.

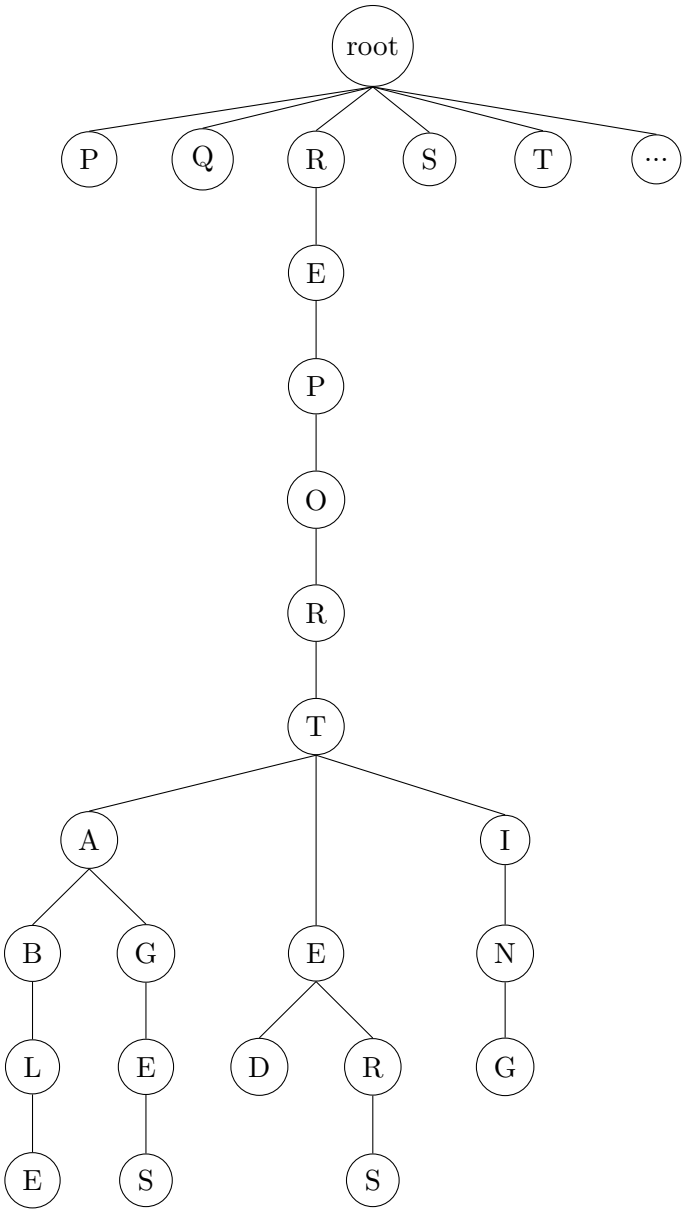

Figure 1: Trie Example — The vanilla version for the dictionary entries: report, reportable, reportage, reportages, reported, reporter, reporters, reporting

then look at the necessary extensions required in order to use transitional probability to be able to calculate potential prefixes and suffixes.

2 Trie Implementation

The first basic task will consist of implementing a 'rudimentary' Trie. There are three primary objects that are required:

1. Node — The node data structure will provide us with the 'blueprint' to create all the required nodes.
2. TrieDictionary — The Trie data structure provides us with an 'interface' to access our Trie. All requests will be done to this structure, which will have access to the nodes.
3. Main Class — From where we interact with the Trie.

General Hint: Read through Tasks 1–3 several times before starting to implement them. First build a mental model and make sure that you understand the general principles of Nodes and Tries, then scribble it on paper/s, and only then start implementing.

2.1 Task 1: Implement a simple Node structure

The first task consists of creating the appropriate Node skeleton structure. Remember that this class will act as a blueprint for the many nodes that we will need to create. Start by spending some time and analysing what different information needs to be stored in order to handle the different type of nodes:

1. Root Node — this one is special. It does not represent anything really, its like a dummy node, but we need it so that we always have one single node through which we access everything else. This is marked as the red node in figure 2.
2. Regular Node — This is our normal node. It represents one single character, and it might or might not have children. These are marked as black nodes in figure 2. There are several of them, and this is our most basic building block.
3. End Token Node — This is similar to our regular node, with the only difference that we set a marker in this node to note that the path from the root to this node represents a valid word. This is marked in a double-lined blue circle in figure 2 and you can easily make out the different words from the root to each of these blue nodes.

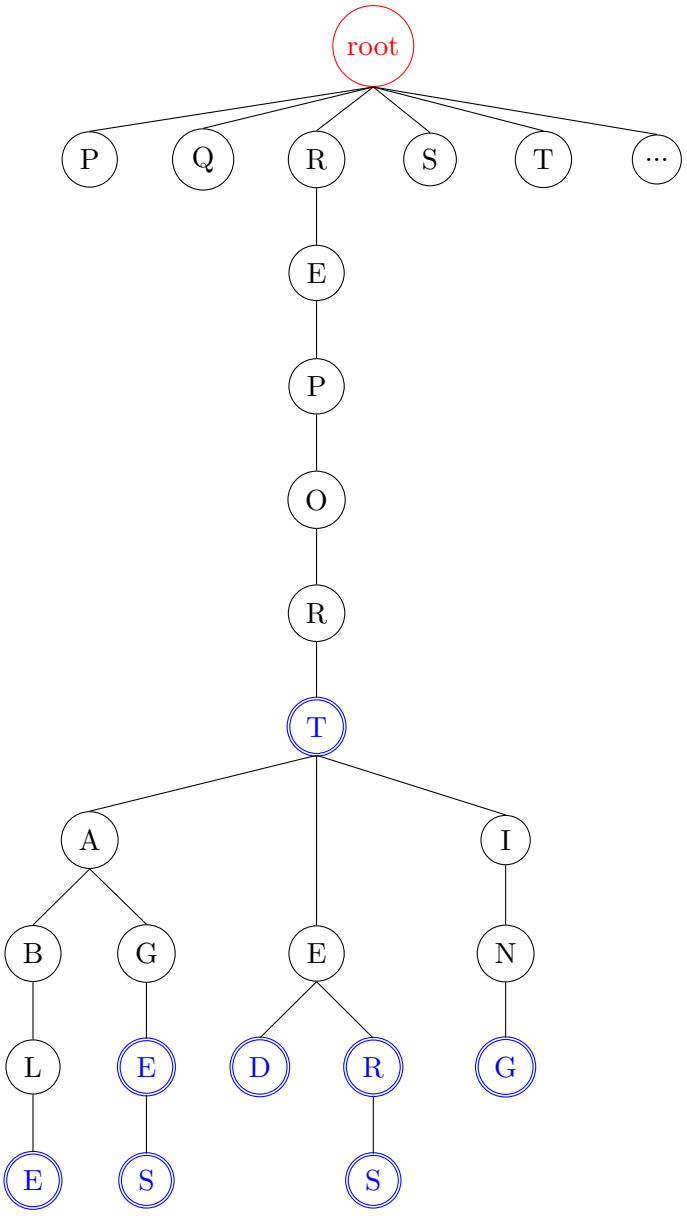

Figure 2: Trie Example — The coloured version for the dictionary entries: report, reportable, reportage, reportages, reported, reporter, reporters, reporting. The red node is the root node, the black nodes are normal nodes and the blue double-lined nodes are the end token nodes.

If we look closely to the diagram, we can reason out a few things that could help us during the implementation:

1. Only the root (i.e. the 1^{st}) node is red.
2. Black nodes always occur internally and are never found as leafs (i.e. nodes on the very edge, without any children) — why is this?
3. Blue node can occur both at internal points but also occur as end points (leafs) of the Trie — why?
4. What is the primary difference between the blue nodes and the black nodes?
5. Do we need different classes for these two nodes or can we implement one class and cover both types of nodes?
6. How about the root node — does it require special implementation, or can we get away with implementing all these three types of nodes in one?

Short answer: SINGLE node implementation

Requirement: In your documentation take care to explain clearly why only one node class is required, and demonstrate how you cater for the different node 'types' with one class.

Next item to consider: each node can have several children. How would you store these children? Remember, these are nodes as well. So in your node class, you need storage for a list of children of type `Node`.

Once you have thought of the data ('variables') that the node will hold, next you will need to think about the methods that will be required.

1. Constructor — we need to be able to create a node. We know that each node represents a character, so we can pass on this character to our constructor. However, does every single node really represent a character? What about the root node?
2. Get Node Character — this might be useful when we want to probe which character a node represents.
3. Search For Child — we will need a searching facility to either access a particular child, or to check if it exists.
4. Add an Input String — one of our main functions that will be used when a string is entered into the dictionary.

Listing 1 provides you with the basic 'code' if you had to implement this in Java. There are *holes* in the code which you are meant to come up with your own solution and technique. Make sure that you document your choices and reasoning. Feel free to change the code/variables/naming that is used. Make this your code that you are able to understand and reason about.

Listing 1: `TrieNode` Class

```
class TrieNode:
    def __init__(self, myCharName):
        #A TrieNode my always be associated with a character,
        #Except the Root node
        self.endToken = False #True only when the node represents the end of the
            word
        self.ChildNodes = []#This is a list of children nodes that this node has
        self.char = myCharName#Setting the name character of this node

    def getNodeChar(self):
        #This method simply returns its name character
        return self.char

    def getChildNode(self, childChar):
        #Search through the list of ChildNodes
        #until you find the node with the character 'childChar'
        #and then return that childNode
        #But if you don't find it, return None
        return None

    def addEntry(self, substring):
        #Take the first character of the substring
        #check whether this character already exists as a childNode
        #if NOT, then create a new node, name it for that character,
        #and store the new node as one of the childNodes
        #Once the child node is identified, pass on the remainder of the substring
        #to the child node so that it continues to process it
        #If you reached the last character of the substring
        #set the end token to True, and finally return True.
        return None
```

2.2 Task 2: Implement a Trie Dictionary Class

The second task is the basic implementation of the Trie Dictionary class. This is necessary because as programmers, we do not access the nodes directly, not even the root node — we always access the dictionary as a whole. Reflect on what type of queries we will be doing with out Trie Dictionary:

1. Constructor — create a new dictionary. This would be empty at this stage, but surely the one thing that we know is that a dictionary must have a root node in order to access the remaining of the trie. At `TrieDictionary` level, we only care about the root node. The dictionary has no other knowledge about the rest of the nodes.

2. Add a word — we want the facility to add a word to the dictionary. How? We simply tell the root node to add the word we would like it to add. How it does it? From a `TrieDictionary` level we don't care — we let the root node handle this. Remember that the root node is an instance of a `Node`, and has the facility to add an input string.

3. Query Word — we also would like the facility to check whether a word exists in our dictionary or not. This is something that we have not considered during the `Node` class implementation, however it is a very handy feature to have. The technique is simple — we ask the root node if a word exists or not. The tricky part is that we must go back to the `Node` class and include a method to search for the path that represents the query word. Remember, we know that a word exists in the dictionary when we have a path from the root node to a *blue* node — this part is implemented in the `Node` class. From a `TrieDictionary` level we only care about the final answer from the node. As a hint, implement this only when you fully understand how a word is added to the dictionary, because searching for a word is quite a similar process.

Listing 2: `TrieDictionary` Class

```
1 class TrieDictionary:
2     def __init__(self):
3         print("I am the Root Node, I have no name")
4         self.RootNode = TrieNode('') #create an instance node, but root has no
              character!
5
6     def addWord(self, word):
7         return self.RootNode.addEntry(word)
```

Listing 3: Console output after running `Main` Class

```
-------- Adding Words to Trie --------
Adding report true
Adding reporter true
Adding reporters true
Adding reported true
Adding reportable true
Adding reportage true
Adding reportages true
Adding reporting true
-------- Checking if all good words return true --------
Checking for report true
Checking for reporter true
Checking for reporters true
Checking for reported true
Checking for reportable true
Checking for reportage true
Checking for reportages true
Checking for reporting true
-------- Checking that all list of bad words return false --------
Checking for repo false
Checking for save false
Checking for t false
Checking for ejis false
Checking for repe false
```

2.3 Task 3: Implement a Main Class

This task will simply be a way of interacting with the Trie Dictionary. We will add entries, check if entries exist, and so on. We will do this gradually in simple subtasks.

1. Create a Trie Dictionary
2. Create two lists of words — an array of 10 words in each one should be fine as an initial test. One list will be an input list to the Trie, the other list will contain different words that should not exist in the Trie.
3. Iterate through word list 1 and add each word to the Trie
4. Iterate through word list 1 and check if each word is in the Trie Dictionary
5. Iterate through word list 2 and check if each word is in the Trie Dictionary (this one shouldn't be).

The above is quite easy, and can be similar to listing 4, showing an output described in listing 3.

Requirement: What worked, what didn't? Take note of your errors and include a short description in your documentation. Did you get the expected output immediately right? What did you do to fix your problem?

The next required extension is that you are able to take the word list directly from a file. The file will contain one word per line and will have the associated token count for each word, tab delimited.

Create a function in your `Main` class that will read the word list and store it in an appropriate data structure of your preference.

1. What storage did you choose?
2. Give a short description of how this structure meets your requirements.

This is a preparation for the next section of the assignment, where apart from adding words in the dictionary, we will also need to take note of the token count amongst other things. For now, you can test your `TrieDictionary` with the full word list (ignoring token counts) — how fast is the input? Check a few entries and ensure that the behaviour is as expected.

Listing 4: `Main` Class

```
1 if __name__ == '__main__':
2     goodwords = ["report", "reporter", "reporters", "reported", "reportable", "
          reportage"]
3     badwords = ["repo", "abi"]
4
5     MyTrie = TrieDictionary()
6     for word in goodwords:
7         MyTrie.addWord(word)
```

Table 1: Example of iterating through the word **reporters** and the equivalent representation of $\alpha AB\beta$

Representation	α	A	B	β
r-eporters	–	r	e	porters
re-porters	r	e	p	orters
rep-orters	re	p	o	rters
repo-rters	rep	o	r	ters
repor-ters	repo	r	t	ers
report-ers	repor	t	e	rs
reporte-rs	report	e	r	s
reporter-s	reporte	r	s	–

3 Finding Affixes

So far we have carried out the basic implementation of the Trie Dictionary structure. Now, we are going to use it with respect to finding affixes. From this point onwards, we refer to a Trie Dictionary, or Dictionary, as the above data structure completed with all the words in the word list added to it. To simplify the explanation, we will only consider suffixes for the time being.

For each word that we process, we want to determine where is the most likely boundary between stem and suffix. This is done by iterating through the wordlist and for each word we consider all possible word boundaries, i.e. a possible word boundary to be at every letter of the word. Thus, if we have the word 'report', we will need to consider the word boundary being at 'r' (giving us 'r' as the stem, 'eport' as a suffix), at 'e' (giving us 're' as the stem, 'port' as a suffix) and so forth. In order to be able to express formally the necessary probabilities, we will consider the representation of $\alpha AB\beta$ as the whole word, where αA is the stem, and $B\beta$ is the suffix. We formally say that we are considering the word boundary at A. Table 1 shows how the word 'reporters' would be processed, and represented at each stage of our iteration.

A potential word boundary is viewed positively if it passes from ALL of the following three tests:

1. The root being considered must itself be present as a word in the corpus. Therefore αA must be a word in the corpus. In our implementation this means that if we had to check whether αA exists in the Dictionary, it would return `true` because the variable `endToken` is set to `true` at the end of the word (indicating an end of path). From

table 1 we know that this is only possible when we are considering 'report-ers' and 'reporter-s'. Our programme is not as intuitive, and must check each and every boundary in table 1.

2. The number of words in the corpus (`tokenCount`) starting with the possible stem sequence (αA) must be approximately similar to the number of words starting with the stem less one character (α). For example, if we are considering 'report-ers', the number of words starting with 'report' and 'repor' should be approximately the same. Formally, this is expressed as:

$$\Pr(A \mid \alpha) \approx 1 \tag{1}$$

3. The number of words in the in the corpus starting with the root (αA) must be higher than the number of words starting with the root plus one character (αAB). So similarly, we should find more words starting with 'report' than with 'reporte' because intuitively we know that 'report' is found also in other words such as 'reportable', etc. Thus this would indicating a probable word boundary.

$$\Pr(B \mid \alpha A) < 1 \tag{2}$$

When a potential word boundary $B\beta$ passes the above 3 tests, we increment its score by 19. If it fails any of the tests, we decrease its score by 1. These scores are not arbitrary, but are purposely selected to have a 5% difference between them. Read Keshava and Pitler (2006) for more details on this.

Ultimately, we are after a list of potential suffixes together with their associated scores which indicates the likelihood of each affix. Now we need to carry out some further implementation of our `TrieDictionary` in order to be able to consider the necessary calculations.

3.1 Task 4: Extending the Node Structure

In order to be able to calculate probabilities at different potential word boundaries, we now need to include some figures into our node structure.

Recall equations 1 and 2. Both specify "the number of words in the corpus starting with *** ". This means that we now have to include the token count in our dictionary. Let us have a dummy token count of our word list, shown in table 2:

Now consider the wording once again and more specifically — in equation 1: "the number of words in the corpus starting with the root sequence αA". If we are at the stage where $\alpha AB\beta$ is 'repo-rters', and therefore 'repo' is αA — what is the number of words in the corpus starting with the root

Table 2: Token Counts for our sample word list

Word	Token Count
report	3900
reporter	241
reporters	82
reported	609
reportable	5
reportage	63

sequence 'repo'? All the words in table 2 start with 'repo', so the answer is all the counts.

The `Node` implementation now needs to cater for the following variables, which we will find useful:

- `tokenCount` represents the number of words that a particular path is found in the corpus. This means that this count include the frequency of the words as found in the corpus.
- `endToken` is a boolean value that indicates that the path from the root to this node is a valid word in the corpus.

To summarise, task 4 is about extending the implementation of the method `addEntry` in the `TrieNode` class. Before proceeding with your implementation, ensure that the algorithm works without taking into account `tokenCount`, and then later on extend it by including `tokenCount`.

3.2 Task 5: Extending the Trie Structure to cater for Transitional Probability

In order to be able to calculate the transitional probabilities, we need to be able to carry out three tests. The first test simply checks whether a potential stem is a word in the Trie and this method is already in place (`hasEntry`).

Next two tests require the token count of particular string sequences. As explained above, if we need to check the stem boundary for the word 'repo-rters', we need the token count for 'rep' (alpha), 'repo' (alphaA), and 'repor' (alphaAB). Therefore, we need to have a function in our `TrieNode` that will return the token count of a particular sequence.

The function `getTokenCount(String input)` is quite similar to the function `hasEntry` — the main difference is that one is returning a `boolean` whether a word exists or not, whilst the other needs to return an integer

indicating the number of words in the corpus which start with the string sequence `substring`.

As a sample code, look at the listings below to get an idea of the implementation/skeleton of the methods that we now required.

Listing 5: Console output for the iterative boundary check

```
Boundary check: r − eporters
Test 1 for r is False
Test 2: Freq of alphaA / Freq of alpha = is it approx equal to 1?
alphaA: 4900 / alpha: 0 = False
Test 3: Freq of alphaAB / Freq of alphaA = is it much less than 1?
alphaAB: 4900 / alphaA: 4900 = False

Boundary check: re − porters
Test 1 for re is False
Test 2: Freq of alphaA / Freq of alpha = is it approx equal to 1?
alphaA: 4900 / alpha: 4900 = True
Test 3: Freq of alphaAB / Freq of alphaA = is it much less than 1?
alphaAB: 4900 / alphaA: 4900 = False

Boundary check: rep − orters
Test 1 for rep is False
Test 2: Freq of alphaA / Freq of alpha = is it approx equal to 1?
alphaA: 4900 / alpha: 4900 = True
Test 3: Freq of alphaAB / Freq of alphaA = is it much less than 1?
alphaAB: 4900 / alphaA: 4900 = False

Boundary check: repo − rters
Test 1 for repo is False
Test 2: Freq of alphaA / Freq of alpha = is it approx equal to 1?
alphaA: 4900 / alpha: 4900 = True
Test 3: Freq of alphaAB / Freq of alphaA = is it much less than 1?
alphaAB: 4900 / alphaA: 4900 = False

Boundary check: repor − ters
Test 1 for repor is False
Test 2: Freq of alphaA / Freq of alpha = is it approx equal to 1?
alphaA: 4900 / alpha: 4900 = True
Test 3: Freq of alphaAB / Freq of alphaA = is it much less than 1?
alphaAB: 4900 / alphaA: 4900 = False

Boundary check: report − ers
Test 1 for report is True
Test 2: Freq of alphaA / Freq of alpha = is it approx equal to 1?
alphaA: 4900 / alpha: 4900 = True
Test 3: Freq of alphaAB / Freq of alphaA = is it much less than 1?
alphaAB: 932 / alphaA: 4900 = True

Boundary check: reporte − rs
Test 1 for reporte is False
Test 2: Freq of alphaA / Freq of alpha = is it approx equal to 1?
alphaA: 932 / alpha: 4900 = False
Test 3: Freq of alphaAB / Freq of alphaA = is it much less than 1?
alphaAB: 323 / alphaA: 932 = True

```

```
Boundary check: reporter − s
Test 1 for reporter is True
Test 2: Freq of alphaA / Freq of alpha = is it approx equal to 1?
alphaA: 323 / alpha: 932 = False
Test 3: Freq of alphaAB / Freq of alphaA = is it much less than 1?
alphaAB: 82 / alphaA: 323 = True
```

3.3 Task 6: Using a real word list

The final task is to consider a realistic programming scenario. You are given a wordlist - each line has a word, tab, number of tokens. You need to (i) read the file, (ii) process the input i.e. store the string word, and convert the number of tokens into an integer, (iii) build your TrieDictionary and input all the words and their respective token counts.

Once all this is in place, you can iterate through the word list and calculate potential suffixes. An additional trie can then be created to treat prefixes.

Listing 6: `Main` Class **UPDATED**

```
1
2  if __name__ == '__main__':
3      #Here goes the code to open the wordlist file
4      #and store every word and its token count
5      #From this, you then populate your Trie Datastructure
6
7
8      #For now, we have a fictitious list of words and
9      #their token counts
10     listofwords = {}
11     listofwords["report"] = 3900
12     listofwords["reporter"] = 241
13     listofwords["reporters"] = 82
14     listofwords["reported"] = 609
15     listofwords["reportable"] = 5
16     listofwords["reportage"] = 63
17
18
19     badwords = ["repo", "abi"]
20
21     # This part is only temporary to ascertain that your code works well.
22     MyTrie = TrieDictionary()
23     for k,v in listofwords.iteritems():
24         print("Adding " + k + " to my Trie returns: " + str(MyTrie.addWord(k,v)))
25     print
26     #Checking if hasEntry is functioning properly!
27     for k in listofwords:
28         print("Check for " + k + " gives: " + str(MyTrie.hasEntry(k)))
29     print
30     for i in badwords:
31         print("Check for " + i + " gives: " + str(MyTrie.hasEntry(i)))
32
33     #Here would go the algorithm to loop
34     # through the word list, retrieve the
35     # info required for the three tests
36     # and carry out the necessary calculations
37     # to decide whether we have a potential
38     # suffix or not.
```

Listing 7: `TrieDictionary` Class **UPDATED**

```
class TrieDictionary:
    def __init__(self):
        self.RootNode = TrieNode('')

    def addWord(self, word, count):
        return self.RootNode.addEntry(word, count)

    def hasEntry(self, word):
        return self.RootNode.hasEntry(word)

    def getTokenCount(self, word):
        return self.RootNode.getTokenCount(word)
```

Listing 8: `TrieNode` Class **UPDATED**

```
class TrieNode:
    def __init__(self, myCharName):
        #A TrieNode my always be associated with a character,
        #Except the Root node
        self.endToken = False #True only when the node represents the end of the
            word
        self.ChildNodes = []#This is a list of children nodes that this node has
        self.char = myCharName#Setting the name character of this node
        self.tokenCount = 0#Setting the count initially as 0

    def getNodeChar(self):
        #This method simply returns its name character
        return self.char

    def getChildNode(self, childChar):
        #Search through the list of ChildNodes
        #until you find the node with the character 'childChar'
        #and then return that childNode
        #But if you don't find it, return None
        return None

    def addEntry(self, substring, count): #new addition: count
        #Take the first character of the substring
        #check whether this character already exists as a childNode
        #if NOT, then create a new node, name it for that character,
        #and store the new node as one of the childNodes
        #Once the child node is identified, pass on the remainder of the substring
        #to the child node so that it continues to process it
        #If you reached the last character of the substring
        #set the end token to True, and finally return True.
        #But now you must also include the information about the count
        #EVERY node will also store the count apart from the character
        return None

    def hasEntry(self, substring):
        #Similar to addEntry in principle. But the function differs:
        #If the substring exists as a word in the Trie, return True
        #Else return False
        return None

    def getTokenCount(self, substring):
        #Return the count of a particular substring
        #For now, returning 0, but this should be changed.
        return 0
```

4 Research Question

A part of your assignment includes a small research component through which you are expected to search for additional information and provide a discussion reflecting what you read, and your own informed opinion where possible.

4.1 A computational perspective

Option 1:

One problem encountered during this assignment is that of storing the child nodes. Discuss the implementation that you chose, and why. What features does the implementation that you chose offered which determined your selection?

Option 2:

The TrieNode implementation given in the assignment originally specified that the children of that node are maintained as an array of type TrieNode. We know that any node can have as many children as there are letters in the alphabet, e.g. 30 in Maltese. So the root node can have 30 children, each of those children can have 30 more, etc. This can not only make the data structure itself quite huge, but also quite computationally intensive for searching. Although we add the children node to the array on a first-come-first-in basis, this might mean that a particular search might take up more time than others, because its characters always happen to be the last node in the children array. That means that, if we are searching for a 10-letter word, and each letter happens to be at the worst-case scenario (stored at the last position in our children array the 30^{th} place), this would mean that our programme would take 300 "checks" to finally get to the last letter of this word.

1. Is there a way of improving our search facility? — Discuss.
2. Does the programming language provide libraries for searching?
3. Change your code to implement this improvement, and report on whether the improvements are visible in the way that your code now executes.

4.2 A linguistic perspective

What are your observations on the output of your algorithm? Discuss the instances where the algorithm is successful in its task, and where it fails. What are the reasons behind the success / failures? Are there potential

improvements that could be implemented to improve the analysis? Discuss and explain whether, with the improvements that you suggest, the algorithm remains unsupervised or not.

5 Important Information

5.1 Submission

Submission dates: in the VLE you have the option to submit part 1 by the 20th December, and part 2 by the 8th January. These are NOT final submissions, but if you need feedback it is imperative that you submit something. Your final submission in the VLE is divided in two parts - Code is to be submitted by the 13th January, and documentation is to be submitted after the end of the exams. The exact date for this will be set later, so please check the VLE regularly.

5.2 Assessment

The assessment will include a short interview to (a) ensure that the code compiles/runs properly, and (b) to ensure that you understand your own code.

The grading structure for your assignment will follow this table:

Table 3: Grading value of the different sections

T#	Desc	Mark
T1-3	Trie Dictionary	15%
T4	Added Functionality	15%
T5	Added Functionality	15%
T6	Final Scenario	25%
R1	Computational Perspective	15%
R2	Linguistic Perspective	15%
	Total:	100%