

### Assignment 3

#### Trie Trees and Auto-Complete Bots

</div>

In this assignment, you will utilise a new tree-based data structure to build an auto-complete engine. Important notes:

- Watch [this video](#) recorded by Joram Erbarth (M23) with advice on how to prepare for CS110 assignments. Most of the suggestions will also apply to other CS courses, so make sure to bookmark this video for future reference.
- Make sure you fill in any place that says `###YOUR DOCSTRING HERE` , `###YOUR CODE HERE` or `"YOUR ANSWER HERE"` .
- Feel free to add more cells (Markdown and coding) to the ones always provided in each question to expand your answers, as needed.
- Given the Covid-19 pandemic, please follow all the local guidelines rigorously.
- Please be aware of the weight of this assignment which should reflect both time and effort management. Please plan accordingly.
- Please refer to the [CS110 course guide](#) on how to submit your assignment materials.
- Do not submit code screenshots. Instead, your PDF should originate from a well-presented Jupyter notebook. You will receive an overall grade on #professionalism.
- If you have any questions, do not hesitate to reach out to the TAs in the Slack channel `#cs110-algo` , or come to the instructors' OHs.

#### 🚩 Setting up:

Start by stating your name and identifying your collaborators. Please comment on the nature of the collaboration (for example, if you briefly discussed the strategy to solving problem 1, say so, and explicitly point out what exactly did you discuss).

**Name:** Yousaf Qamar

**Collaborators:** YOUR ANSWER HERE

**Details:** YOUR ANSWER HERE

## An Overview

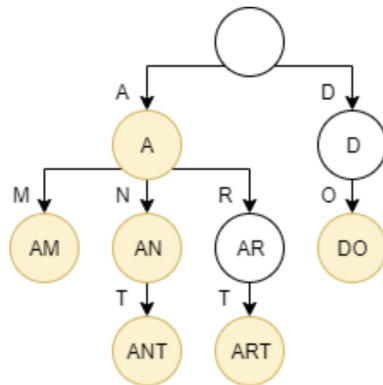
Auto-completion functionalities are now ubiquitous in search engines, document editors, and messaging apps. How would you develop an algorithmic strategy from scratch to implement these computational solutions? In this assignment, you will learn about a new data structure and use it to build a very simple auto-complete engine. Each question in the assignment guides you closer to that objective while encouraging you to contrast this novel data structure to the other ones discussed in class.

A [trie tree](#), or a prefix tree, is a common data structure that stores a set of strings in a collection of nodes so that all strings with a common prefix are found in the same branch of the tree. Each node is associated with a letter, and as you traverse down the tree, you pick up more letters, eventually forming a word. Complete words are commonly found on the leaf nodes. However, some inner nodes can also mark full words.

Let's use an example diagram to illustrate several important features of tries:

In [1]:

```
from IPython.core.display import Image, display
display(Image(url='https://drive.google.com/uc?id=1NxHPsTzU3xEz2Ivck1NyqKpOix4Sc5iE', width=200))
```



Here are a few things to note from the schematics above:

- Nodes that mark valid words are marked in yellow. Notice that while all leaves are considered valid words, only some inner nodes contain valid words, while some remain only prefixes to valid words appearing down the branch.

- The tree does not have to be balanced, and the height of different branches depends on its contents.
- In our implementation, branches never merge to show common suffixes (for example, both ANT and ART end in T, but these nodes are kept separate in their respective branches). However, this is a standard first line of memory optimization for tries.
- The first node contains an empty string; it “holds the tree together.”

Your task in this assignment will be to implement a functional trie tree. You will be able to insert words into a dictionary, lookup valid and invalid words, print your dictionary in alphabetical order, and suggest appropriate suffixes like an auto-complete bot.

The assignment questions will guide you through these tasks one by one. To stay safe from breaking your own code, and to reinforce the idea of code versioning, under each new question first **copy your previous (working) code**, and only then **implement the new feature**. The code skeletons provided throughout will make this easier for you at the cost of repeating some large portions of code.

## Q1—Python implementation of a trie tree

In this question, you will write Python code that can take a set/list/tuple of strings and:

- insert them into a trie tree and
- lookup whether a specific word/string is present in the trie tree.

### A. Theoretical pondering

You might recall two main approaches to building trees from class: making separate **Tree and Node** classes, or only making a **Node** class. Which method do you think is a better fit for trie trees, and why? Justify your reasoning in around 100 words (provide a word count). Throughout the assignment, you will use your chosen approach, so don't rush this question.

By separating the two classes we will make it easier for me to work with the code. This would allow us to separate the classes which would basically mean that instead of just having one class where every method that involves using the entire structure as a whole would be defined alongside the node structure itself instead of separately. By separating them, it will be easier for us identify the parts of the code that are supposed to use the entire structure as a whole while also handling the specific parts in the form of node which will have their own attributes. This will help us a lot when it comes to debugging, as the classes are separated while also making additions to the code, for future improvement in functionality, much easier, by separating the node class and that class specifically having its attributes we can easily add more attributes.

### B. Practical implementation

In the two cells below, there are two code skeletons.

- Depending on your answer to question **A**, either implement a `Node` and a `Trie` class or implement a `Node` class. **Choose the corresponding code cell and delete the other one.**
- For your class(es), write `insert()` and `lookup()` methods, which will insert a word into the trie tree and look it up, respectively. Use the code skeleton and examine the specifications of its docstrings to guide you on the details of inputs and outputs to each method.
- If you are coding two classes, your `Trie` class should, upon initiation, create the root `Node`. If you are coding a single class, use an attribute to mark the root node.
- Finally, ensure that the trie can be initiated with a wordbank as an input. This means that a user can create a trie and feed it an initial set of words at the same time (like in the tests below), which will be automatically inserted into the trie upon its creation. Likely, this will mean that your `__init__()` has to make some calls to your `insert()` method.
- Several test cases have been provided for your convenience, and these include some, but not all, possible edge cases. If the implementation is correct, your code will pass all the tests. Note: there are several ways in which we can condense the text cleaning syntax that we have provided for you without repeating the method `replace()` multiple times. However, we are leaving it this way for clarity.
- In addition, create **at least three more tests** to demonstrate that your code is working correctly and **justify** why such test cases are appropriate.

In [2]:

```
# VERSION 1 - Node + Trie classes

class Node:
    """This class represents one node of a trie tree.

    Parameters
    -----
    character: char
        value of the node, letter from the word for which we are creating a node

    Attributes
    -----
    data: char
        the alphabet, value, stored in the node
    children: dict
        it keeps track of the alphabets that follow the node, in other terms children of the node
    word_end: bool
        variable that keeps track of the word showing whther an alphabet marks the end of the word or not
```

```

"""

def __init__(self, character):
    """Creates a Node instance

    Parameters
    -----
    character: char
        The letter which will be set as the node's value
    """
    self.data = character
    self.children = {}
    self.word_end = False

class Trie:
    """This class represents the entirety of a trie tree.

    Parameters
    -----
    word_list: arr
        a list or array of strings which hold the words that are to be inserted in the trie tree

    Attributes
    -----
    word_list: arr
        a list or array of strings which hold the words that are to be inserted in the trie tree
    root: node
        initialized as an empty string so it can act as a starting point for our insert and lookup methods

    Methods
    -----
    insert(self, word)
        Inserts a word into the trie, creating nodes as required.
    lookup(self, word)
        Determines whether a given word is present in the trie.
    """

    def __init__(self, word_list = None):
        """Creates the Trie instance, inserts initial words if provided.

        Parameters
        -----
        word_list : list
            List of strings to be inserted into the trie upon creation.
        """

```

```

self.word_list = word_list
self.root = Node('')

#tree is generated by using insert method on each character in the word_list
for word in word_list:
    self.insert(word)

def insert(self, word):
    """Inserts a word into the trie, creating missing nodes on the go.

    Parameters
    -----
    word : str
        The word to be inserted into the trie.
    """

    #simple check to see if word entered is of type string
    if type(word) != str:
        return False

    #initial node is set to root
    node = self.root
    #words are changed to lower case for consistency during comparisons
    word = word.lower()

    #iterating through all characters in the word list
    for char in word:
        if char in node.children:
            #if character is present in the tree than node pointer is changed to the children
            node = node.children[char]
        else: #if character is not present, we make a new node
            #making object
            child_node = Node(char)
            #children is set equal to the object
            node.children[char] = child_node
            #again setting the node pointer to the child
            node = node.children[char]

    #after the full word has been inserted, the last node's boolean will be updated indicating it is the end of the w
    node.word_end = True

def lookup(self, word):
    """Determines whether a given word is present in the trie.

    Parameters
    -----

```

```

word : str
    The word to be looked-up in the trie.

Returns
-----
bool
    True if the word is present in trie; False otherwise.

Notes
-----
Your trie should ignore whether a word is capitalized.
E.g. trie.insert('Prague') should lead to trie.lookup('prague') = True
"""

#simple check to see if word entered is of type string
if type(word) != str:
    return False

#initial node is set to root
node = self.root
#words are changed to lower case for consistency during comparisons
word = word.lower()

#iterating through all characters in the word list
for char in word:
    if char in node.children:
        #if character is present in the tree than node pointer is changed to the children
        node = node.children[char]
    else:#if character is not found we will return falseas the word will not be present in the tree
        return False

#checking if the node we have is the words ending
if node.word_end:
    return True
else:
    return False

# Here are several tests that have been created for you.
# Remeber that the question asks you to provide several more,
# as well as to justify them.

# This is Namárië, JRRT's elvish poem written in Quenya
wordbank = "Ai! laurië lantar lassi súrinen, yéni unótimë ve rámar aldaron! \
Yéni ve lintë yuldar avánier mi oromardi lisse-miruvóreva Andúnë pella, \
Vardo tellumar nu luini yassen tintilar i eleni ómaryo airetári-lírinen. \

```

```

SÍ man i yulma nin enquantuva? An sí Tintallë Varda Oiolossëo ve fanyar máryat Elentári ortanë, \
ar ilyë tier undulávë lumbulë; ar sindanóriello caita mornië i falmalinnar imbë met, \
ar hísië untúpa Calaciryo míri oialë. Sí vanwa ná, Rómello vanwa, Valimar! Namárië! \
Nai hiruvalyë Valimar. Nai elyë hiruva. \
Namárië!".replace("!", "").replace("?", "").replace(".", "").replace(",", "").replace(";", "").split()

```

```

trie = Trie(wordbank)
# be careful about capital Letters!
assert trie.lookup('oiolossëo') == True
# this is a prefix, but also a word in itself
assert trie.lookup('an') == True
# this is a prefix, but NOT a word
assert trie.lookup('ele') == False
# not in the wordbank
assert trie.lookup('Mithrandir') == False

```

In [3]:

```

#Test cases
#one of the most common letter patterns seen in the list is "la", using this test case we see that the whole word is checked
assert trie.lookup('lantar') == True
#as we are using lower() function inserting a capitalized letter should not cause any problems
assert trie.lookup('Vardo') == True
#These two test cases act in unison where I was checking that the tree inserts the special "é" into the tree and it
#shouldn't work with the simple "e"
#Pair of test cases to see the tree can differentiate between special character like "á" from simple characters like "a"
assert trie.lookup('rámar') == True
assert trie.lookup('ramar') == False

```

Through my test cases I am trying to see whether the algorithm can handle edge cases.

The first test case tries to see if the tree is able to handle different words with same starting prefix correctly. Looking at the wordbank, on a glance it seems that "la" is the most common starting prefix, so based on this prefix I used the second word with this prefix, "lantar", and tried to see if the algorithm is able to follow through correctly.

The second test case is used to check if the lower() function is able to work properly. So the algorithm should be able to handle mixed cases or other types of inputs as inevitably we will convert it to lower case. To check I just used a random capitalized word from the word bank, "Vardo".

The last test case, is a pair of assert functions. Its sole goal is to check whether special characters and normal characters can be differentiated by the tree. So by using the special character the same way as in the wordbank we should get True however, changing it to a normal character should give us false, we checked this through the assert functions.



## Q2—The computational complexity of tries

Explain your answers to these questions as clearly as you can.

- Evaluate the computational complexity of the `insert()` and `lookup()` methods in a trie.

What are the **relevant variables** for runtime? You might want to consider how the height of a trie is computed to start addressing this question. Make sure to explain your reasoning clearly.

- Compare your results to the runtime of the same operations on a BST.
  - To address this question, you will need to describe explicitly how a BST would store the same information stored in the trie tree.
  - Can you think of specific circumstances where the practical runtimes of operations supported by tries are higher than for BSTs? Explain your answer.
  - If you believe such cases could be expected, why would someone even bother implementing a trie tree?

### Time complexity of Trie Trees:

#### **insert():**

The number of times this function has to be called is dependent on the length of the word that we are given. If we take the length of a word as  $n$ , we see that this becomes a complexity of  $O(n)$ . We can see that the time complexity of inbuilt functions like `lower()` will work in constant time so it takes  $O(1)$ . The making of new nodes and changing of the nodes takes constant time so it can be written as  $O(1)$ .

Thus we get the following time recurrence equation:

$$T(n) = O(n) + O(1) + O(1) + O(1) = O(n) + 3O(1)$$

As the length of the word increases, reaching asymptotic behaviour, we can ignore  $O(1)$  as this part of the equation will become negligible as compared to the growth in  $O(n)$ . So we are left with:

$$T(n) = O(n) \text{ where } n \text{ is the number of characters in the word.}$$

#### **lookup():**

As the `lookup()` method is a case sensitive implementation of the insert method with no addition or removal of loops that might affect the complexity, the overall time complexity of both these method will be the same. However, we need to consider the fact that their space complexity might differ as lookup method does not add anything to the existing Trie structure. On the other hand an insert method does add to Trie structure. This would mean that the the possible space complexity of lookup method will be  $O(1)$  whereas we need to store  $n$  letter in the insert method the complexity will be  $O(n)$ .

### Comparison to BST:

For Trie trees the runtime is dependent on the length of the word that is being inputted, as it is a  $k$ -array search tree, while for BST it is dependent on the height of the tree as we have to compare across the height in order to carry out insert or lookup method. Thus for BST the methods are dependent on  $O(h)$ , where  $h$  is the height of the tree.

Looking at both Trie trees and BST in practical terms, we can see that the trie will mostly take more time, as it can possibly have more than two children, unlike BST, the number of children will simply be bound by the number of valid characters. Having more children would mean that each level we will have to compare across more than two values, unlike BST, with the possibility of comparing across all the characters at each level. Even without the possibility of having a maximum, there is a high probability that having more than two children would make Trie trees worse than BST. However, if we look at the specific implementation of words and literature, we would not be able to achieve the same level of functionality with a BST as it would not be able to all the amount of data required, so Tri trees will be preferred in these cases.

### Q3—Find the $k$ most common words in a speech.

To mathematically determine the overall connotation of a speech, you might want to compute which words are most frequently used and then run a sentiment analysis. To this end, add a method to your code, `k_most_common()` that will take as an input  $k$ , an integer, and return a list of the  $k$  most common words from the dictionary within the trie. The structure of the output list should be such that each entry is a tuple, the first element being the word and the second an integer of its frequency (see docstring if you're confused).

- To complete this exercise, you don't have to bother with resolving ties (for example, if  $k = 1$ , but there are two most common words with the same frequency, you can return either of them). You can consider resolving ties as an extra challenge and let us know if you believe you solved it.
- The test cell below downloads and preprocesses several real-world speeches, and then runs the  $k$ -most-common word analysis of them; your code should pass the tests. There are cleaner and more concise ways to write that testing code, but this way should be easily understandable. The tests contain the following speeches:

- Mehreen Faruqi - Black Lives Matter in Australia: <https://bit.ly/CS110-Faruqi>
- John F. Kennedy - The decision to go to the Moon: <https://bit.ly/CS110-Kennedy>

- Martin Luther King Jr. - I have a dream: <https://bit.ly/CS110-King>
- Greta Thunberg - UN Climate Summit message: <https://bit.ly/CS110-Thunberg>
- Vaclav Havel - Address to US Congress after the fall of Soviet Union: <https://bit.ly/CS110-Havel>

- As usual, add at least **three** more tests, and **justify** why they are relevant to your code (feel free to find more speeches to start analysing too!).
- Again, copy-paste your previous code and adjust to this *new version*. Note that the method is indented on purpose.
- This task will probably require your nodes to store more information about the frequency of words inserted into the tree. One data structure that might be very useful to tackle the problem of traversing the tree and finding the most common words is heaps—you **are** allowed to use the `heapq` library or another alternative for this task.

In [19]:

```
class Node:
    """This class represents one node of a trie tree.

    Parameters
    -----
    character: char
        value of the node, letter from the word for which we are creating a node

    Attributes
    -----
    data: char
        the alphabet, value, stored in the node
    children: dict
        it keeps track of the alphabets that follow the node, in other terms children of the node
    word_end: bool
        variable that keeps track of the word showing whether an alphabet marks the end of the word or not

    """

    def __init__(self, character):
        """Creates a Node instance

        Parameters
        -----
        character: char
            The letter which will be set as the node's value
        """
        self.data = character
        self.children = {}
        self.word_end = False
```

```
#counter specifically for the k most common method
self.counter = 0
```

```
class Trie:
```

```
    """This class represents the entirety of a trie tree.
```

```
    Parameters
```

```
    -----
```

```
    word_list: arr
```

```
        a list or array of strings which hold the words that are to be inserted in the trie tree
```

```
    Attributes
```

```
    -----
```

```
    word_list: arr
```

```
        a list or array of strings which hold the words that are to be inserted in the trie tree
```

```
    root: node
```

```
        initialized as an empty string so it can act as a starting point for our insert and lookup methods
```

```
    Methods
```

```
    -----
```

```
    insert(self, word)
```

```
        Inserts a word into the trie, creating nodes as required.
```

```
    lookup(self, word)
```

```
        Determines whether a given word is present in the trie.
```

```
    word_counter(self, starting_node, word, words_count_list)
```

```
        Returns a list of all the words alongside their respective counts
```

```
    k_most_common(self, k)
```

```
        Finds k words inserted into the trie most often.
```

```
    """
```

```
def __init__(self, word_list = None):
```

```
    """Creates the Trie instance, inserts initial words if provided.
```

```
    Parameters
```

```
    -----
```

```
    word_list : list
```

```
        List of strings to be inserted into the trie upon creation.
```

```
    """
```

```
    self.word_list = word_list
```

```
    self.root = Node('')
```

```
    #tree is generated by using insert method on each character in the word_list
```

```
    for word in word_list:
```

```
        self.insert(word)
```

```

def insert(self, word):
    """Inserts a word into the trie, creating missing nodes on the go.

    Parameters
    -----
    word : str
        The word to be inserted into the trie.
    """
    #simple check to see if word entered is of type string
    if type(word) != str:
        return False

    #initial node is set to root
    node = self.root
    #words are changed to lower case for consistency during comparisons
    word = word.lower()

    #iterating through all characters in the word list
    for char in word:
        if char in node.children:
            #if character is present in the tree than node pointer is changed to the children
            node = node.children[char]
        else: #if character is not present, we make a new node
            #making object
            child_node = Node(char)
            #children is set equal to the object
            node.children[char] = child_node
            #again setting the node pointer to the child
            node = node.children[char]

    #incrementing the value of the counter allows us to keep track of the count of the number of times a word appears
    node.counter += 1
    #after the full word has been inserted, the last node's boolean will be updated indicating it is the end of the w
    node.word_end = True

def lookup(self, word):
    """Determines whether a given word is present in the trie.

    Parameters
    -----
    word : str
        The word to be looked-up in the trie.

```

## Returns

-----

bool

True if the word is present in trie; False otherwise.

## Notes

-----

Your trie should ignore whether a word is capitalized.

E.g. trie.insert('Prague') should lead to trie.lookup('prague') = True

"""

*#simple check to see if word entered is of type string***if** type(word) **!=** str:    **return** False*#initial node is set to root*

node = self.root

*#words are changed to lower case for consistency during comparisons*

word = word.lower()

*#iterating through all characters in the word list***for** char **in** word:    **if** char **in** node.children:        *#if character is present in the tree than node pointer is changed to the children*

node = node.children[char]

**else:** *#if character is not found we will return false as the word will not be present in the tree*        **return** False*#checking if the node we have is the words ending***if** node.word\_end:    **return** True**else:**    **return** False**def** word\_counter(self, starting\_node, word, words\_count\_list):    *"""Returns a list of all the words alongside their respective counts*

## Parameters

-----

starting\_node: node

The starting point of the method

word: str

The current running word

```

words__count_list: arr
    Array of tuples holding the words and their respective counts
"""

#assigning node the value of the starting_node so it is at the starting point
node = starting_node

#check to see if the node has any children
if node.children:
    #iterating through all the children of the node
    for char in node.children:
        #adding the character we found to the ongoing running_word
        running_word = word + char

        #check to see if the node is indicating the end of the word
        if node.children[char].word_end == True:
            #append the word and its frequency to the list
            words_count_list.append((running_word, node.children[char].counter))

        #recursively calling the method so we can go through all branches of the tree
        self.word_counter(node.children[char], running_word, words_count_list)

#initializing the word to be blank before we start with another branch
word = ""

return words_count_list

def k_most_common(self, k):
    """Finds k words inserted into the trie most often.

    Parameters
    -----
    k : int
        Number of most common words to be returned.

    Returns
    -----
    list
        List of tuples.

        Each tuple entry consists of the word and its frequency.
        The entries are sorted by frequency.

    Example
    -----
    >>> print(trie.k_most_common(3))

```

```
[('the', 154), ('a', 122), ('i', 122)]
```

I.e. the word 'the' has appeared 154 times in the inserted text.  
The second and third most common words both appeared 122 times.  
"""

*#initial node is set to root*

```
node = self.root
```

*#getting the list of all the words that are present in the tree with their respective counts*

```
most_common = self.word_counter(self.root, node.data, [])
```

*#sorting the list alphabetically, A-Z*

```
most_common.sort(key = lambda x:x[0])
```

*#further sortting in terms of count so our output is both numerically and alphabetically arranged*

```
most_common.sort(key = lambda x:x[1], reverse = True)
```

*#returning a list of tuples untill a certain number which is in the parameters*

```
return most_common[0:k]
```

In [20]:

```
!pip install requests
```

```
from requests import get
```

```
speakers = ['Faruqi', 'Kennedy', 'King', 'Thunberg', 'Havel']
```

```
bad_chars = [';', ',', '.', '?', '!', '_',  
            '["', ']', ':', '"', '"', '"', ' ', '-']
```

```
for speaker in speakers:
```

*# download and clean up the speech from extra characters*

```
speech_full = get(f'https://bit.ly/CS110-{speaker}').text
```

```
just_text = ''.join(c for c in speech_full if c not in bad_chars)
```

```
without_newlines = ''.join(c if (c not in ['\n', '\r', '\t']) else " " for c in just_text)
```

```
just_words = [word for word in without_newlines.split(" ") if word != ""]
```

```
trie = Trie(just_words)
```

```
# trie = Node(just_words)
```

```
if speaker == 'Faruqi':
```

```
Faruqi = [('the', 60), ('and', 45), ('to', 39), ('in', 37),  
          ('of', 34), ('is', 25), ('that', 22), ('this', 21),  
          ('a', 20), ('people', 20), ('has', 14), ('are', 13),  
          ('for', 13), ('we', 13), ('have', 12), ('racism', 12),
```



```

        ('black', 11), ('justice', 9), ('lives', 9), ('police', 9)]
    assert trie.k_most_common(20) == Faruqi

    elif speaker == 'Kennedy':
        Kennedy = [('the', 117), ('and', 109), ('of', 93), ('to', 63),
                    ('this', 44), ('in', 43), ('we', 43), ('a', 39),
                    ('be', 30), ('for', 27), ('that', 27), ('as', 26),
                    ('it', 24), ('will', 24), ('new', 22), ('space', 22),
                    ('is', 21), ('all', 15), ('are', 15), ('have', 15), ('our', 15)]
        assert trie.k_most_common(21) == Kennedy

    elif speaker == 'Havel':
        Havel = [('the', 34), ('of', 23), ('and', 20), ('to', 15),
                  ('in', 13), ('a', 12), ('that', 12), ('are', 9),
                  ('we', 9), ('have', 8), ('human', 8), ('is', 8),
                  ('you', 8), ('as', 7), ('for', 7), ('has', 7), ('this', 7),
                  ('be', 6), ('it', 6), ('my', 6), ('our', 6), ('world', 6)]
        assert trie.k_most_common(22) == Havel

    elif speaker == 'King':
        King = [('the', 103), ('of', 99), ('to', 59), ('and', 54), ('a', 37),
                 ('be', 33), ('we', 29), ('will', 27), ('that', 24), ('is', 23),
                 ('in', 22), ('as', 20), ('freedom', 20), ('this', 20),
                 ('from', 18), ('have', 17), ('our', 17), ('with', 16),
                 ('i', 15), ('let', 13), ('negro', 13), ('not', 13), ('one', 13)]
        assert trie.k_most_common(23) == King

    elif speaker == 'Thunberg':
        Thunberg = [('you', 22), ('the', 20), ('and', 16), ('of', 15),
                     ('to', 14), ('are', 10), ('is', 9), ('that', 9),
                     ('be', 8), ('not', 7), ('with', 7), ('i', 6),
                     ('in', 6), ('us', 6), ('a', 5), ('how', 5), ('on', 5),
                     ('we', 5), ('all', 4), ('dare', 4), ('here', 4),
                     ('my', 4), ('people', 4), ('will', 4)]
        assert trie.k_most_common(24) == Thunberg

```

Requirement already satisfied: requests in c:\users\josep\anaconda3\lib\site-packages (2.25.1)  
 Requirement already satisfied: idna<3,>=2.5 in c:\users\josep\anaconda3\lib\site-packages (from requests) (2.10)  
 Requirement already satisfied: urllib3<1.27,>=1.21.1 in c:\users\josep\anaconda3\lib\site-packages (from requests) (1.26.4)  
 Requirement already satisfied: chardet<5,>=3.0.2 in c:\users\josep\anaconda3\lib\site-packages (from requests) (4.0.0)  
 Requirement already satisfied: certifi>=2017.4.17 in c:\users\josep\anaconda3\lib\site-packages (from requests) (2020.12.5)

In [22]: `for speaker in speakers:`

```

# download and clean up the speech from extra characters
speech_full = get(f'https://bit.ly/CS110-{speaker}').text
just_text = ''.join(c for c in speech_full if c not in bad_chars)
without_newlines = ''.join(c if (c not in ['\n', '\r', '\t']) else " " for c in just_text)
just_words = [word for word in without_newlines.split(" ") if word != ""]

trie = Trie(just_words)
# trie = Node(just_words)

if speaker == 'Faruqi':
    #Looking for the 0th value with empty list
    King = []
    #Should just return and empty list as we are just caling the root node.
    assert trie.k_most_common(0) == King

elif speaker == 'Havel':

    Havel = [('the', 34), ('of', 23), ('and', 20), ('to', 15), ('in', 13), ('a', 12), ('that', 12),
              ('are', 9), ('we', 9), ('have', 8), ('human', 8), ('is', 8), ('you', 8), ('as', 7),
              ('for', 7), ('has', 7), ('this', 7), ('be', 6), ('it', 6), ('my', 6), ('our', 6),
              ('world', 6), ('all', 5), ('democracy', 5), ('not', 5), ('still', 5), ('than', 5),
              ('will', 5), ('i', 4), ('one', 4), ('only', 4), ('other', 4), ('responsibility', 4),
              ('time', 4), ('us', 4), ('an', 3), ('been', 3), ('being', 3), ('but', 3), ('by', 3),
              ('experience', 3), ('from', 3), ('more', 3), ('never', 3), ('or', 3), ('something', 3),
              ('two', 3), ('where', 3), ('who', 3), ('about', 2), ('actions', 2), ('advantage', 2),
              ('any', 2), ('approaching', 2), ('at', 2), ('better', 2), ('can', 2), ('consciousness', 2),
              ('countries', 2), ('czechs', 2), ('don't', 2), ('given', 2), ('great', 2), ('horizon', 2),
              ('if', 2), ('life', 2), ('longer', 2), ('may', 2), ('me', 2), ('mere', 2), ('months', 2),
              ('move', 2), ('nations', 2), ('people', 2), ('school', 2), ('sense', 2), ('slovaks', 2),
              ('social', 2), ('someone', 2), ('soviet', 2), ('special', 2), ('sphere', 2), ('system', 2),
              ('therefore', 2), ('they', 2), ('totalitarian', 2), ('toward', 2), ('under', 2), ('union', 2),
              ('was', 2), ('way', 2), ('which', 2), ('years', 2), ('17th', 1), ('above', 1), ('absurd', 1),
              ('accumulated', 1), ('ahead', 1), ('always', 1), ('am', 1)]

    #Pushing the algorithm a bit by seeing if it can run upto 100 unique words and find their count.
    assert trie.k_most_common(100) == Havel

elif speaker == 'Kennedy':
    #inserting a very big number should return
    Kennedy = [('the', 117), ('and', 109), ('of', 93), ('to', 63),
                ('this', 44), ('in', 43), ('we', 43), ('a', 39),
                ('be', 30), ('for', 27), ('that', 27), ('as', 26),
                ('it', 24), ('will', 24), ('new', 22), ('space', 22),
                ('is', 21), ('all', 15), ('are', 15), ('have', 15),
                ('our', 15), ('but', 14), ('i', 13), ('on', 13),

```

```

('do', 12), ('not', 12), ('than', 11), ('man', 10),
('some', 10), ('years', 10), ('at', 9), ('knowledge', 9),
('by', 8), ('first', 8), ('its', 8), ('more', 8), ('us', 8),
('with', 8), ('if', 7), ('moon', 7), ('they', 7), ('ago', 6),
('an', 6), ('because', 6), ('city', 6), ('decade', 6), ('from', 6),
('great', 6), ('made', 6), ('no', 6), ('now', 6), ('one', 6), ('or', 6),
('science', 6), ('them', 6), ('there', 6), ('was', 6), ('who', 6), ('why', 6),
('behind', 5), ('can', 5), ('done', 5), ('go', 5), ('has', 5), ('here', 5),
('history', 5), ('last', 5), ('most', 5), ('states', 5), ('these', 5),
('united', 5), ('well', 5), ('which', 5), ('you', 5), ('before', 4),
('during', 4), ('far', 4), ('high', 4), ('hostile', 4), ('houston', 4),
('intend', 4), ('less', 4), ('must', 4), ('nation', 4), ('national', 4),
('only', 4), ('peace', 4), ('power', 4), ('progress', 4), ('say', 4),
('sea', 4), ('shall', 4), ('so', 4), ('stay', 4), ('think', 4), ('those', 4),
('times', 4), ('what', 4), ('year', 4), ('40', 3), ('about', 3), ('become', 3),
('budget', 3), ('choose', 3), ('climb', 3), ('country', 3), ('delighted', 3),
('despite', 3), ('effort', 3), ('ever', 3), ('every', 3), ('good', 3), ('greater', 3),
('growth', 3), ('his', 3), ('ignorance', 3), ('make', 3), ('many', 3), ('may', 3),
('million', 3), ('mr', 3), ('my', 3), ('noted', 3), ('nuclear', 3), ('other', 3),
('others', 3), ('part', 3), ('people', 3), ('per', 3), ('president', 3), ('rocket', 3),
('satellites', 3), ('saturn', 3), ('scientists', 3), ('spacecraft', 3), ('state', 3),
('still', 3), ('then', 3), ('three', 3), ('time', 3), ('two', 3), ('use', 3), ('week', 3),
('were', 3), ('whether', 3), ('world', 3), ('yet', 3), ('advanced', 2), ('against', 2),
('age', 2), ('ahead', 2), ('am', 2), ('america', 2), ('any', 2), ('ask', 2), ('automobiles', 2),
('been', 2), ('best', 2), ('both', 2), ('built', 2), ('came', 2), ('canaveral', 2), ('cannot', 2),
('cape', 2), ('center', 2), ('cents', 2), ('challenge', 2), ('college', 2), ('combined', 2),
('come', 2), ('congressman', 2), ('conquest', 2), ('control', 2), ('costs', 2), ('created', 2),
('did', 2), ('does', 2), ('earth', 2), ('efforts', 2), ('eight', 2), ('end', 2), ('engines', 2),
('expects', 2), ('exploration', 2), ('explored', 2), ('facilities', 2), ('fact', 2), ('field', 2),
('fires', 2), ('five', 2), ('frontier', 2), ('furthest', 2), ('generating', 2), ('given', 2),
('goal', 2), ('going', 2), ('greatest', 2), ('had', 2), ('he', 2), ('heat', 2), ('help', 2),
('hopes', 2), ('hour', 2), ('how', 2), ('industry', 2), ('i'm', 2), ('know', 2), ('learned', 2),
('little', 2), ('look', 2), ('manned', 2), ('man's', 2), ('mean', 2), ('measure', 2), ('meet', 2),
('miles', 2), ('missile', 2), ('money', 2), ('months', 2), ('number', 2), ('office', 2), ('old', 2),
('ought', 2), ('outpost', 2), ('over', 2), ('own', 2), ('pace', 2), ('pay', 2), ('planets', 2),
('powerful', 2), ('rice', 2), ('said', 2), ('sail', 2), ('school', 2), ('scientific', 2), ('see', 2),
('seen', 2), ('set', 2), ('solve', 2), ('span', 2), ('such', 2), ('sure', 2), ('tall', 2), ('terms', 2),
('texas', 2), ('therefore', 2), ('though', 2), ('together', 2), ('university', 2), ('unknown', 2),
('unprotected', 2), ('venus', 2), ('very', 2), ('vowed', 2), ('want', 2), ('war', 2), ('waves', 2),
('we're', 2), ('where', 2), ('while', 2), ('whole', 2), ('without', 2), ('won', 2), ('your', 2),
('$1', 1), ('$200', 1), ('$5400', 1), ('$60', 1), ('10', 1), ('10000', 1), ('12', 1), ('1630', 1),
('19', 1), ('1961', 1), ('24', 1), ('240000', 1), ('25000', 1), ('300', 1), ('35', 1), ('40yard', 1),
('45', 1), ('48', 1), ('5', 1), ('50', 1), ('50000', 1), ('50year', 1), ('accelerators', 1),
('accept', 1), ('accompanied', 1), ('accuracy', 1), ('act', 1), ('actions', 1), ('administration', 1),
('admit', 1), ('adventure', 1), ('adventures', 1), ('aeronautics', 1), ('again', 1), ('air', 1),

```

```

('airplanes', 1), ('alive', 1), ('alloys', 1), ('already', 1), ('america's', 1), ('among', 1),
('animals', 1), ('answerable', 1), ('anything', 1), ('appreciate', 1), ('area', 1), ('around', 1),
('asked', 1), ('assembled', 1), ('assure', 1), ('atlantic', 1), ('atlas', 1), ('atmosphere', 1),
('available', 1), ('await', 1), ('away', 1), ('backwash', 1), ('banner', 1), ('bay', 1), ('became', 1),
('began', 1), ('being', 1), ('bell', 1), ('benefits', 1), ('better', 1), ('between', 1), ('beyond', 1),
('billion', 1), ('blessing', 1), ('block', 1), ('body', 1), ('bold', 1), ('booster', 1),
('bradford', 1), ('breathtaking', 1), ('brief', 1), ('british', 1), ('building', 1), ('c1', 1),
('capable', 1), ('capsule', 1), ('carrying', 1), ('cart', 1), ('causing', 1), ('caves', 1),
('celestial', 1), ('certain', 1), ('change', 1), ('child', 1), ('christianity', 1),
('cigarettes', 1), ('cigars', 1), ('circled', 1), ('citizens', 1), ('clustered', 1),
('collective', 1), ('colony', 1), ('coming', 1), ('communications', 1), ('community', 1),
('companies', 1), ('comparable', 1), ('complex', 1), ('comprehension', 1), ('computers', 1),
('condense', 1), ('conflict', 1), ('conquered', 1), ('conscience', 1), ('construct', 1),
('contract', 1), ('cool', 1), ('cooperation', 1), ('courage', 1), ('course', 1), ('cover', 1),
('create', 1), ('dangerous', 1), ('dangers', 1), ('deal', 1), ('decide', 1), ('decision', 1),
('decisions', 1), ('demands', 1), ('depends', 1), ('deserves', 1), ('destruction', 1),
('determined', 1), ('deterred', 1), ('develop', 1), ('die', 1), ('difficulties', 1),
('direct', 1), ('dispels', 1), ('distinguished', 1), ('doing', 1), ('don't', 1),
('double', 1), ('doubling', 1), ('dropping', 1), ('each', 1), ('easy', 1), ('education', 1),
('electric', 1), ('embarked', 1), ('emerged', 1), ('energies', 1), ('engine', 1),
('engineering', 1), ('engineers', 1), ('enriched', 1), ('enterprised', 1), ('environment', 1),
('equipment', 1), ('equivalent', 1), ('even', 1), ('everest', 1), ('except', 1), ('expect', 1),
('expenditures', 1), ('expenses', 1), ('experienced', 1), ('explorer', 1), ('extending', 1),
('eyes', 1), ('f1', 1), ('failures', 1), ('faith', 1), ('fast', 1), ('fear', 1), ('feeding', 1),
('feet', 1), ('fellow', 1), ('felt', 1), ('filled', 1), ('finally', 1), ('finest', 1), ('firing', 1),
('fitted', 1), ('flag', 1), ('flight', 1), ('floor', 1), ('fly', 1), ('food', 1), ('football', 1),
('force', 1), ('forest', 1), ('forwardand', 1), ('founder', 1), ('founding', 1), ('freedom', 1),
('fulfilled', 1), ('fully', 1), ('gained', 1), ('gains', 1), ('gear', 1), ('generation', 1),
('gentlemen', 1), ('george', 1), ('giant', 1), ('glenn', 1), ('globe', 1), ('god's', 1),
('governed', 1), ('governor', 1), ('grasp', 1), ('gravity', 1), ('greatly', 1), ('ground', 1),
('guests', 1), ('guidance', 1), ('half', 1), ('halfcentury', 1), ('hard', 1), ('hardships', 1),
('harvest', 1), ('having', 1), ('hazardous', 1), ('hazards', 1), ('heart', 1), ('helping', 1),
('highest', 1), ('home', 1), ('honorable', 1), ('honorary', 1), ('hope', 1), ('hot', 1), ('hours', 1),
('however', 1), ('human', 1), ('hurricanes', 1), ('icebergs', 1), ('ill', 1), ('ills', 1),
('important', 1), ('increase', 1), ('increases', 1), ('incumbency', 1), ('industrial', 1),
('industries', 1), ('infancy', 1), ('institutions', 1), ('instrument', 1), ('instruments', 1),
('into', 1), ('intricate', 1), ('invented', 1), ('invention', 1), ('invest', 1), ('investment', 1),
('itself', 1), ('itwe', 1), ('january', 1), ('job', 1), ('jobs', 1), ('john', 1), ('join', 1),
('just', 1), ('kinds', 1), ('known', 1), ('laboratory', 1), ('ladies', 1), ('land', 1), ('large', 1),
('laughter', 1), ('launched', 1), ('lead', 1), ('leader', 1), ('leadership', 1), ('leading', 1),
('learning', 1), ('least', 1), ('lecture', 1), ('length', 1), ('lengths', 1), ('lights', 1),
('like', 1), ('lines', 1), ('literally', 1), ('long', 1), ('longer', 1), ('low', 1), ('mallory', 1),
('mankind', 1), ('manpower', 1), ('mapping', 1), ('mariner', 1), ('mass', 1), ('mastered', 1),
('me', 1), ('meaning', 1), ('medicine', 1), ('men', 1), ('metal', 1), ('midnight', 1), ('miller', 1),
('minute', 1), ('mission', 1), ('mistakes', 1), ('misuse', 1), ('modern', 1), ('month', 1),

```

```

('mount', 1), ('mountain', 1), ('move', 1), ('moved', 1), ('mysteries', 1), ('nations', 1),
('nation's', 1), ('need', 1), ('needed', 1), ('needs', 1), ('never', 1), ('newton', 1), ('next', 1),
('obligations', 1), ('observation', 1), ('occasion', 1), ('occupies', 1), ('ocean', 1), ('once', 1),
('opening', 1), ('opportunity', 1), ('organize', 1), ('ours', 1), ('ourselves', 1), ('outer', 1),
('outlays', 1), ('outstrip', 1), ('outthen', 1), ('overcome', 1), ('paid', 1), ('particularly', 1),
('peaceful', 1), ('penicillin', 1), ('person', 1), ('personnel', 1), ('pitzer', 1), ('plant', 1),
('platform', 1), ('play', 1), ('playing', 1), ('plymouth', 1), ('population', 1), ('position', 1),
('postpone', 1), ('precision', 1), ('preeminence', 1), ('prejudice', 1), ('presidency', 1),
('press', 1), ('previous', 1), ('printing', 1), ('priorityeven', 1), ('problems', 1),
('professor', 1), ('program', 1), ('promise', 1), ('propulsion', 1), ('provided', 1),
('public', 1), ('putting', 1), ('quest', 1), ('race', 1), ('rate', 1), ('reached', 1),
('reaching', 1), ('realize', 1), ('reap', 1), ('reasons', 1), ('recorded', 1), ('reentering', 1),
('regard', 1), ('region', 1), ('related', 1), ('repeating', 1), ('require', 1), ('rest', 1),
('rested', 1), ('return', 1), ('revolution', 1), ('reward', 1), ('right', 1), ('rights', 1),
('rise', 1), ('rode', 1), ('safely', 1), ('safer', 1), ('salaries', 1), ('same', 1),
('security', 1), ('senator', 1), ('send', 1), ('serve', 1), ('several', 1), ('shake', 1),
('share', 1), ('shattered', 1), ('shelter', 1), ('shift', 1), ('ships', 1), ('short', 1),
('shot', 1), ('should', 1), ('sit', 1), ('site', 1), ('sixties', 1), ('skilled', 1),
('skills', 1), ('skins', 1), ('somewhat', 1), ('soon', 1), ('sophisticated', 1), ('source', 1),
('soviet', 1), ('spacefaring', 1), ('speaking', 1), ('speeds', 1), ('stadium', 1),
('staggering', 1), ('stand', 1), ('standard', 1), ('standing', 1), ('stands', 1), ('stars', 1),
('stated', 1), ('station', 1), ('steam', 1), ('steer', 1), ('storms', 1), ('story', 1),
('strength', 1), ('stresses', 1), ('stretches', 1), ('strife', 1), ('striking', 1), ('structure', 1),
('succeeds', 1), ('sum', 1), ('sunalmost', 1), ('supplied', 1), ('surely', 1), ('surprising', 1),
('survival', 1), ('teaches', 1), ('technical', 1), ('techniques', 1), ('technology', 1),
('telephones', 1), ('television', 1), ('temperature', 1), ('tens', 1), ('terrifying', 1),
('testing', 1), ('thank', 1), ('theater', 1), ('their', 1), ('things', 1), ('thomas', 1),
('thousands', 1), ('tiros', 1), ('today', 1), ('todayand', 1), ('tonight', 1), ('too', 1),
('tools', 1), ('transit', 1), ('unanswered', 1), ('under', 1), ('understanding', 1),
('unfinished', 1), ('unfolds', 1), ('union', 1), ('universe', 1), ('unprecedented', 1),
('untried', 1), ('unwilling', 1), ('up', 1), ('used', 1), ('vast', 1), ('vice', 1), ('vision', 1),
('visiting', 1), ('vistas', 1), ('vows', 1), ('wait', 1), ('waited', 1), ('warnings', 1), ('waste', 1),
('watch', 1), ('wave', 1), ('way', 1), ('weapons', 1), ('webb', 1), ('west', 1), ('wheels', 1),
('wide', 1), ('wiley', 1), ('william', 1), ('willing', 1), ('win', 1), ('wished', 1), ('within', 1),
('woman', 1), ('work', 1), ('working', 1), ('world's', 1), ('would', 1), ('writ', 1), ('write', 1),
('yeara', 1), ('year's', 1)]
#algorithm should stil be able to return even if the number used is greater than the number of words in the spec
assert trie.k_most_common(1000000000) == Kennedy

```

## Q4–Implement an autocomplete with a Shakespearean dictionary!

Your task is to create a new `autocomplete()` method for your class, which will take a string as an input, and return another string as an output. If the string is not present in the tree, the output will be the same as the input. However, if the string is present in the trie tree, your task is to find the most common word to which it is a prefix and return that word instead (this can still turn out to be itself).

- To make the task more interesting, use the test cell code to download and parse *The Complete Works of William Shakespeare*, and insert them into a trie. Your autocomplete should then pass the tests below. This is a *large* book, so the code might take a while to run...!
- As usual, add at least **three** more test cases, and explain why they are appropriate (you can use input other than Shakespeare for them).
- Make sure to include a minimum **100 word-summary critically evaluating** your autocomplete engine. How does it really work? Your critical reflection needs to precisely assess the role of the different data structures used by their algorithm and what is the overall complexity that the algorithm offers. Can we do better? If so, how and by how much?
- Again, depending on how you choose to implement it, your `autocomplete()` might make calls to other helper methods. However, make sure that `autocomplete()` is the method exposed to the user in order to pass the tests.)\*
- *Note:* this is a manifestly frequentist approach to the problem, which is not the only method to solve this computational problem, and in many cases, not the ideal technique. However, if you were tasked with implementing something like [this](#) or [this](#), it might just be enough, so let's give it a go. Good luck!\*

In [9]:

```
class Node:
    """This class represents one node of a trie tree.

    Parameters
    -----
    character: char
        value of the node, letter from the word for which we are creating a node

    Attributes
    -----
    data: char
        the alphabet, value, stored in the node
    children: dict
        it keeps track of the alphabets that follow the node, in other terms children of the node
    word_end: bool
        variable that keeps track of the word showing whther an alphabet marks the end of the word or not

    """

    def __init__(self, character):
```

```

        """Creates a Node instance

        Parameters
        -----
        character: char
            The letter which will be set as the node's value
        """
        self.data = character
        self.children = {}
        self.word_end = False
        #counter specifically for the k most common method
        self.counter = 0

class Trie:
    """This class represents the entirety of a trie tree.

    Parameters
    -----
    word_list: arr
        a list or array of strings which hold the words that are to be inserted in the trie tree

    Attributes
    -----
    word_list: arr
        a list or array of strings which hold the words that are to be inserted in the trie tree
    root: node
        initialized as an empty string so it can act as a starting point for our insert and lookup methods

    Methods
    -----
    insert(self, word)
        Inserts a word into the trie, creating nodes as required.
    lookup(self, word)
        Determines whether a given word is present in the trie.
    word_counter(self, starting_node, word, words_count_list)
        Returns a list of all the words alongside their respective counts
    k_most_common(self, k)
        Finds k words inserted into the trie most often.
    """

    def __init__(self, word_list = None):
        """Creates the Trie instance, inserts initial words if provided.

        Parameters
        -----

```

```

word_list : list
    List of strings to be inserted into the trie upon creation.
"""

self.word_list = word_list
self.root = Node('')

#tree is generated by using insert method on each character in the word_list
for word in word_list:
    self.insert(word)

def insert(self, word):
    """Inserts a word into the trie, creating missing nodes on the go.

    Parameters
    -----
    word : str
        The word to be inserted into the trie.
    """
    #simple check to see if word entered is of type string
    if type(word) != str:
        return False

    #initial node is set to root
    node = self.root
    #words are changed to lower case for consistency during comparisons
    word = word.lower()

    #iterating through all characters in the word list
    for char in word:
        if char in node.children:
            #if character is present in the tree than node pointer is changed to the children
            node = node.children[char]
        else: #if character is not present, we make a new node
            #making object
            child_node = Node(char)
            #children is set equal to the object
            node.children[char] = child_node
            #again setting the node pointer to the child
            node = node.children[char]

    #incrementing the value of the counter allows us to keep track of the count of the number of times a word appears
    node.counter += 1

```



*#after the full word has been inserted, the last node's boolean will be updated indicating it is the end of the word*  
 node.word\_end = True

```
def lookup(self, word):
    """Determines whether a given word is present in the trie.

    Parameters
    -----
    word : str
        The word to be looked-up in the trie.

    Returns
    -----
    bool
        True if the word is present in trie; False otherwise.

    Notes
    -----
    Your trie should ignore whether a word is capitalized.
    E.g. trie.insert('Prague') should lead to trie.lookup('prague') = True
    """
    #simple check to see if word entered is of type string
    if type(word) != str:
        return False

    #initial node is set to root
    node = self.root
    #words are changed to lower case for consistency during comparisons
    word = word.lower()

    #iterating through all characters in the word list
    for char in word:
        if char in node.children:
            #if character is present in the tree than node pointer is changed to the children
            node = node.children[char]
        else: #if character is not found we will return false as the word will not be present in the tree
            return False

    #checking if the node we have is the words ending
    if node.word_end:
        return True
    else:
        return False
```

```

def word_counter(self, starting_node, word, words_count_list):
    """Returns a list of all the words alongside their respective counts

    Parameters
    -----
    starting_node: node
        The starting point of the method
    word: str
        The current running word
    words_count_list: arr
        Array of tuples holding the words and their respective counts
    """
    #assigning node the value of the starting_node so it is at the starting point
    node = starting_node

    #check to see if the node has any children
    if node.children:
        #iterating through all the children of the node
        for char in node.children:
            #adding the character we found to the ongoing running_word
            running_word = word + char

            #check to see if the node is indicating the end of the word
            if node.children[char].word_end == True:
                #append the word and its frequency to the list
                words_count_list.append((running_word, node.children[char].counter))

            #recursively calling the method so we can go through all branches of the tree
            self.word_counter(node.children[char], running_word, words_count_list)

    #initializing the word to be blank before we start with another branch
    word = ""

    return words_count_list

def k_most_common(self, k):
    """Finds k words inserted into the trie most often.

    Parameters
    -----
    k : int
        Number of most common words to be returned.

    Returns
    """

```

-----

list

List of tuples.

Each tuple entry consists of the word and its frequency.  
The entries are sorted by frequency.

Example

-----

```
>>> print(trie.k_most_common(3))
[('the', 154), ('a', 122), ('i', 122)]
```

I.e. the word 'the' has appeared 154 times in the inserted text.  
The second and third most common words both appeared 122 times.  
"""

```
#initial node is set to root
node = self.root
```

```
#getting the list of all the words that are present in the tree with their respective counts
most_common = self.word_counter(self.root, node.data, [])
```

```
#sorting the list alphabetically, A-Z
most_common.sort(key = lambda x:x[0])
```

```
#further sortting in terms of count so our output is both numerically and alphabetically arranged
most_common.sort(key = lambda x:x[1], reverse = True)
```

```
#returning a list of tuples untill a certain number which is in the parameters
return most_common[0:k]
```

```
def autocomplete(self, prefix):
    """Finds the most common word with the given prefix.
```

You might want to reuse some functionality or ideas from Q4.

Parameters

-----

prefix : str

The word part to be "autocompleted".

Returns

-----

str

The complete, most common word with the given prefix.

## Notes

-----

The return value is equal to prefix if there is no valid word in the trie.  
 The return value is also equal to prefix if prefix is the most common word.  
 """

*#initial node is set to root*

node = self.root

*#simple check to see if the prefix is an empty list, which is a special case for this method*

```
if prefix == "":
    return prefix
```

*#iterating through each character of the prefix*

```
for char in prefix:
```

```
    if char in node.children:
```

*#if character is present in the tree than node pointer is changed to the children*

```
        node = node.children[char]
```

*#otherwise we will simply return the prefix as there is no word in the tree with this specific prefix*

```
    else:
```

```
        return prefix
```

*#creating a list of all possible words with that prefix*

```
suggestions = [(prefix, node.counter)]
```

*#creating a list of tuples with the possible words we get from the same prefix with their respective count*

```
final_result = self.word_counter(node, prefix, suggestions)
```

*#sorting the list based on the count of every word*

```
final_result.sort(key = lambda x:x[1], reverse = True)
```

*#returning the most common occurring word which will just be the very first element of the first tuple of the words*

```
return final_result[0][0]
```

In [10]:

```
from requests import get
bad_chars = [';', ',', '.', '?', '!', '1', '2', '3', '4',
            '5', '6', '7', '8', '9', '0', '_', '[', ']']
```

```
SH_full = get('http://bit.ly/CS110-Shakespeare').text
```

```
SH_just_text = ''.join(c for c in SH_full if c not in bad_chars)
```

```
SH_without_newlines = ''.join(c if (c not in ['\n', '\r', '\t']) else " " for c in SH_just_text)
```

```
SH_just_words = [word for word in SH_without_newlines.split(" ") if word != ""]
```

*#### depending on your choice of approach, uncomment one of the lines below*

```
SH_trie = Trie(SH_just_words)
# SH_trie = Node(SH_just_words)

assert SH_trie.autocomplete('hist') == 'history'
assert SH_trie.autocomplete('en') == 'enter'
assert SH_trie.autocomplete('cae') == 'caesar'
assert SH_trie.autocomplete('gen') == 'gentleman'
assert SH_trie.autocomplete('pen') == 'pen'
assert SH_trie.autocomplete('tho') == 'thou'
assert SH_trie.autocomplete('pent') == 'pentapolis'
assert SH_trie.autocomplete('petr') == 'petruchio'
```

In [23]:

```
#Empty string test to see if passing an empty string returns an empty string
assert SH_trie.autocomplete('') == ''

#check based on personal knowledge, the most common english word "the" should be returned if we pass t
assert SH_trie.autocomplete('t') == 'the'

#random prefix which will most likely not be present in the text, the prefix should be returned like it is
assert SH_trie.autocomplete('Microsoft') == 'Microsoft'
```

My autocomplete engine will find all the possible words that we can get from the prefix, and then arrange them based on their count with the most common word being picked.

### Time Complexity:

Things taking  $O(1)$  time:

Setting starting node, creating an empty list, check to see if the prefix itself is the word.

The loop that checks whether each letter is present in the tree takes  $O(m)$  time where  $m$  is the length of the input.

Finding all the possible words with the given prefix takes  $O(n * m)$  time where  $m$  is the length of the longest possible word and  $n$  is the number of words with that prefix.

Finding the most common word in the list takes  $O(n \log n)$  time as we are using `sort()` function. Here  $n$  is the number of words in the list, all the possible words we get from the prefix.

So the time recursion equation will be:

$$T(n) = 3O(1) + O(m) + O(n \log n) + O(n*m)$$

As we approach asymptotic behaviour the constants values will become negligible so we will be left with:

$$T(n) = O(n \log n) + O(n)$$

This is a good implementation of an autocomplete engine however, making use of heaps will further improve it. The complexity of building a heap is  $O(n)$ , with the complexity of Heapify being  $O(\log n)$ , thus giving us a running complexity of  $O(n \log n)$  and finding the max value is done in constant time.

Thus looking at the leading terms for both approaches shows that using heaps will provide us with a faster implementation of the algorithm.

With Sort:

$$T(n) = O(n \log n) + O(n)$$

With Heaps:

$$T(n) = O(n \log n)$$

## Q5–Reflecting on your LOs feedback

Include below the most critical feedback you have received from your PCW workbook or in-class polls. Explain, about 50 words, how you used that feedback to improve your LO applications in this assignment (please include a word count). Be as specific as possible.

**Complexity Analysis:** I got a 2 in session 19 as my comments regarding complexity were too general. In order to overcome this whenever I carried out a complexity analysis in this assignment I would carry out the full calculations to get the complexity while also explaining why a certain line of code will show a certain type of complexity.

## Q6–HC applications


Identify three HCs you have utilized in this assignment and reflect on your application in at least 50 words each (please include a word count).


**audience:** My response to all the questions was tailored in such a way that it would be easily understood by my classmates or anyone with the basic knowledge of this course. By going in depth with my explanations and through the inclusion of detailed doc strings and comments I am easily able to convey to my audience what is happening at each step while still ensuring that the presentation of the assignment is professional.

**algorithms:** I effectively used code to fill out the missing gaps within the classes by writing the codes for the methods. The code is well presented and easily readable, and with inclusion of docstrings and comments each line of code's use is well justified.

**biased identification:** By only running the test cases provided we are falling under confirmation bias, and could possibly just make our code so that it just runs the test cases. The same thing can happen if our self made test cases are very similar to the provided test cases. My understanding of confirmation biases allowed me to make a working piece of code that was not just applicable to the test cases provided.

Finally:

- Refer to the [CS110 course](#) guide on important notes about how to submit your assignment materials and the other sections that refer to the grading policy. Failure to include your Jupyter Notebook as part of your assignment resources will result in a 1 in #PythonProgramming.
- Before you turn this assignment in, make sure everything runs as expected and that there are no errors from running the code. Click on Kernel  Restart & Run All.

 Congratulations on finishing the assignment!

In [ ]: