Yang, Steven CS110 Assignment 3

November 21, 2021

NAME = "Steven H. Yang"

COLLABORATORS = "Robert Dumitru"

NOTE: I visited Ha (CS110) TA's OH to discuss this assignment and got her help.

---

1 CS110 Assignment 3 - Trie trees

Fell free to add more cells to the ones always provided in each question to expand your answers, as needed. Make sure to refer to the CS110 course guide on the grading guidelines, namely how many HC identifications and applications you are expected to include in each assignment.

If you have any questions, do not hesitate to reach out to the TAs in the Slack channel "#cs110-algo", or come to one of your instructors' OHs.

1.0.1 Submission Materials

Your assignment submission needs to include the following resources: 1. A PDF file must be the first resource and it will be created from the Jupyter notebook template provided in these instructions. Please make sure to use the same function names as the ones provided in the template. If your name is "Dumbledore", your PDF should be named "Dumbledore.pdf". 2. Your second resource must be a single Python/Jupyter Notebook named "Dumbledore.ipynb". You can also submit a zip file that includes your Jupyter notebook, but please make sure to name it "Dumbledore.zip" (if your name is Dumbledore!).

For details on how to create a nice PDF from a Jupyter notebook, refer again to the CS110 course guide.

1.0.2 HCs and LOs for this assignment

[#responsibility], [#PythonProgramming], [#CodeReadability], [#DataStructures], [#ComplexityAnalysis], [#ComputationalCritique]

1.1 Question 0

Take a screenshot of your CS110 dashboard on Forum where the following is visible: * your name. * your absences for the course have been set to excused up to the end of week 10 (inclusively).

This will be evidence that you have submitted acceptable pre-class and make-up work for a CS110 session you may have missed. Check the specific CS110 make-up and pre-class policies in the syllabus of the course.

Review Syllabus

Course Stats

CS110 - Problem Solving with Data Structures and Algorithms

- **3** Assignment extensions used
- **2** Total absences
  - **0** Absences - Documented
  - **2** Absences - Excused
    - 2 - Due to missed classes
    - 1 - Session 13 on Oct 21, 2021
    - 1 - Session 16 on Nov 2, 2021
  - **0** Absences - Unexcused

1.2 Overview

Auto-completion functionalities are now ubiquitous in search engines, document editors, and messaging apps. How would you go about developing 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 we have 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:

- 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 common 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.

1.3 Q1: Implement 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.

1.3.1 Q1a: Theoretical pondering

Two main approaches to building trees, you might recall from class, are 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.** You will use your chosen approach throughout the assignment, so don't rush this question.

It is a better idea to separate the problem into two classes. One needs to understand that if nodes and trees as a whole are handled at once, the following problems can occur:

1. The code architecture way is more confusing because it is not intuitive and clear enough for a person to read. #CodeReadabilty
2. When one debugs, it is more time-consuming as the size of the problem is bigger. (i.e. Just debugging node part vs. debugging the whole problem.)

These two reasons are practical reasons that one needs to consider dividing classes.

Other analytical reasons are following:

Having a node class separately allows specializing managing the nodes only. This means when one calls the methods in trees, one can protect the nodes from getting updated accidentally because of unexpected methods. Another reason for having a node class separately is it allows to pass down the data(characters) node to node more effectively – it does not depend on how tree methods work.

Having a trie class separately also makes the problem much simpler as it only deals with the tree structure but not the data themselves. As the data part is already controlled in the class node, the trie class can only have tree structures and make it simpler.

1.3.2 Q1b: Practical implementation

In the two cells below, there are two code skeletons. Depending on your answer to Q1a, 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 should, upon initiation, create the root Node. If you are coding a single class, use an attribute to mark the root node.

Finally, make sure 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 dictionary 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. In addition, create at least **three more tests** to demonstrate that your code is working correctly and justify why such test cases are appropriate.

[2]:
```
# VERSION 1 - Node + Trie classes

# Justification about the intentionality


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

```
    Parameters
    ----------
    character: char
        a character from the word that will be handled inside of the node.
    Attributes
    ----------
    now: char
        the current value(character) in node that is specifically handled.
    children: dict
        look for the following nodes which is the children of the current node
    last_char: bool
        The boolean tells whether the current node is the last character or not␣
↪of the word
    """

    def __init__(self, character):
        """This class represents one node of a trie tree.

        Parameters
        ----------
        character: char
            a character from the word that will be handled inside of the node.
        """
        # YOUR CODE HERE
        #Tells the current character
        self.now = character
        #Gives the dictionary of the children
        self.children = {}
        #As a default, this code assume the current character is not the last␣
↪character.
        self.last_char = False

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

    Parameters
    ----------
    word_list: arr
        an array that includes the words to be processed to develop a tree.

    Attributes
    ----------
    word_list: char
        an array that includes the words to be processed to develop a tree.
    root: node
        the root node is an empty node as the problem stated.
```

```
    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.
        """
        # YOUR CODE HERE
        self.word_list = word_list
        #As the problem stated, the root node is an empty node.
        self.root = Node('')
        #Generates the tree based on the word inputs character by character.
        for i in self.word_list:
            self.insert(i)

    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.
        """
        # YOUR CODE HERE
        #As the example shows, capitalize all letters.
        #This avoids potential problems come from lower/upper cases.
        word = word.upper()
        #Initial node, where the program starts, is the root node
        node = self.root


        for now in word:
            #if one of the children is the character that is identical to␣
↪current character:
            if now in node.children:
                #go to that child
                node = node.children[now]
            #if there's no children, make a new child
```

```
            else:
                #new node called child
                child = Node(now)
                #make this child an object
                node.children[now] = child
                #go to that child
                node = node.children[now]

        #once all characters are inserted, last node becomes true.
        node.last_char = 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
        """
        # YOUR CODE HERE
        #As the example shows, capitalize all letters.
        #This avoids potential problems come from lower/upper cases.
        word = word.upper()
        #Initial node, where the program starts, is the root node
        node = self.root

        #repeat letter by letter to find the word
        for now in word:
            #if there's matched character, follow that path
            if now in node.children:
                node = node.children[now]
            #if there's no child, this means there's no word.
            else:
                return False
        #double check whether the node is the last character
        if node.last_char:
            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

# Note: There are several ways in which we can condense the text cleaning␣
 ↪syntax,
# without repeating the method replace() multiple times,
# but we are leaving it this way for clarity.
```

[3]:
```
# YOUR NEW TESTS HERE
#Test Case 1
#This case includes ! -- a special character.
#This should return False as ! is not a character
#This is also a test to see if upper case matters
assert trie.lookup('Ai!') == False
assert trie.lookup('Ai') == True
#Test Case 2
#This case ommitted spaces between the words
assert trie.lookup('lauriëlantarlassi') == False
#Test Case 3
#This case includes a comma.
#This should return True as , is not a character and should not be processed
assert trie.lookup('súrinen') == True
```

```
#Test Case 4
#This case includes a dash.
#Even dash is a special character, it can be used to connect the words to make␣
 ↪one word
#Therefore it should return True
assert trie.lookup('lisse-miruvóreva') == True
#Test Case 5
#This case should return False as the word one tries to find is NOT a single␣
 ↪word.
assert trie.lookup('nu luini yassen tintilar') == False
#Test Case 6
#This case tests whether the algorithm catches differences between o and ó
#Also e and ë.
assert trie.lookup('unótimë') == True
assert trie.lookup('unotime') == False

#Test Case 7
#This case adds 'a' in between and at the end of lantar.
assert trie.lookup('lantara') == False
assert trie.lookup('lanatar') == False
```

The purpose of given and added test cases is to break the code to see if there are any edge cases that the program does not work.

To achieve this goal, I've tried to come up with some words include special characters, non-alphabets(?) like ë.

To provide more detail on what each test case does:

Test Case 1: It is mixed up with both upper and lower case letter. It also includes the special character. Since special characater is NOT part of the word itself; it should return False. Also whether the input word is in lower or upper case should not matter because that's not the factor differentiates the words.

Test Case 2: Since this input excludes the spaces between the words, it checks whether the word split is successfully happening and count all of them as different words.

Test Case 3: Since comma is a grammatical use, it is not part of the word.

Test Case 4: Dash is a special character; however, it is still part of the word.

Test Case 5: Oppositely from Test Case 2, it now includes the space – multiple words.

Test Case 6: To see whether non alphabets are still processed.

Test Case 7: To see whether an additional character in between or at the end of the word affect the code.

1.4 Q2: The computational complexity of tries

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 clearly explain your reasoning.

Compare your results to the runtime of the same operations on **a BST**. 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 circumstances could be common, why would someone even bother implementing a trie tree?

1. insert() The funciton will be called as much as the number of characters as each function processed one letter each time. This means, when the length of the word is l, it will result O(l) at the end. The uppering will also take O(l) as it applies to every characters.

Making a new node or choosing an existing node (changing a node) takes O(1) each as it's a constant behavior.

Therefore the time complexity in total would be: T(n) = 2*O(l) + 2*O(1)

As the length of the word grows, this function will show the asymptotic behavior; therefore in a simple form of the time complexity is: T(n) = O(l).

2. lookup() The funciton will be called as much as the number of characters as each function processed one letter each time. This means, when the length of the word is l, it will result O(l) at the end. The uppering will also take O(l) as it applies to every characters.

One thing different here, this is not creating or changing the node; instead, it has to check the last node is actually the last node. This will take O(1) as this is also a constant behavior.

Therefore the time complexity in total would be: T(n) = 2*O(l) + O(1)

As the length of the word grows, this function will show the asymptotic behavior; therefore in a simple form of the time complexity is: T(n) = O(l).

COMPARING WITH BST Where n stands for the number of nodes, BST has O(log n) complexity for both insert and lookup. It is strongly based on the structure of the BST: binary. The height of the BST should be much greater than the height of the tries beecause BST can only has two children while tries can have multiple children. In other words, the height of the trie is the same as the length of the longest word (l) in the word bank while BST is something that is greater than that — which can be greater than the longest length of the wrod from the wordbank (n). This is why O(l) < O(n).

BST chooses either left and right or the value greater or not. The worst case of BST is O(n), which is mentnioned above, when the tree is completely inbalanced; however, this rarely happens. With this understanding, the comparisons follow:

It is true that BST performs slightly better theoretically. However, in reality, it should not effect much because the average length of the English word is 5 letters but commonly used and meaningful words mostly go upto 13 characters (Source: Average word length dynamics as indicator of cultural changes in society - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/Average-word-length-in-the-English-language-Different-colours-indicate-the-results-for_fig1_230764201 [accessed 20 Nov, 2021]).

Even O(n) is all best/average/worst cases for all trie tree in insert and lookup method, this is not a huge difference with BST even it performs O(log n). This means that n will not be huge enough in most cases that'd affect the algorithmic behavior dramatically. Given this understanding, as

English language does NOT include such a long word like 100 characters, it is okay to use trie trees for dictionary.

1.5 Q3: Print a dictionary in alphabetical order.

Recall the meaning of pre-order traversal from your previous classes. On the data structure of a trie tree, pre-order traversal corresponds to an alphabetically sorted list of the words contained within (provided that your node children are sorted alphabetically).

For example, on the example trie given in the introduction, pre-order traversal would return [“A”, “AM,”AN“,”ANT“,”AR, “ART,”D” and “DO”]. However, since we are only interested in the actual words, we would not include “D” and “AR” in our list. To that end, you will need to include an attribute for each node, storing the information about whether its content is a word or not.

Copy your existing code to the code skeleton cell below, and add a new method to it, **alphabetical_list()**. This will be version two of your autocomplete script.

The method should **return a list**, whose elements will be the words contained in the tree, in alphabetical order. On top of passing the provided test, write at least **three more tests**, and explain why they are appropriate.

Approach choice: Remember the two possible approaches to the problem, as we’ve seen at the start of the course: iterative or recursive. Depending on your trie implementation, one might be preferred over the other. **Justify your choice of approach** in a few sentences (~100 words).

Copy-paste your previous code and make adjustments to this “new version”, so that you cannot break the old one :).

(Notes: If you choose a recursive approach, it might be useful to implement a helper method that is not called by the user but by preorder_traversal(). Also, watch out for the unintuitive Python behaviour if defining functions with mutable default parameter values.)

[4]:
```
# depending on your choice of approach,
# add the method either to the Node or the Trie class.

# VERSION 2 - Autocomplete Script

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

    Parameters
    ----------
    character: char
        a character from the word that will be handled inside of the node.
    Attributes
    ----------
    now: char
        the current value(character) in node that is specifically handled.
    children: dict
        look for the following nodes which is the children of the current node
    last_char: bool
```

```
        The boolean tells whether the current node is the last character or not␣
↪of the word
    """

    def __init__(self, character):
        """This class represents one node of a trie tree.

        Parameters
        ----------
        character: char
            a character from the word that will be handled inside of the node.
        """
        # YOUR CODE HERE
        #Tells the current character
        self.now = character
        #Gives the dictionary of the children
        self.children = {}
        #As a default, this code assume the current character is not the last␣
↪character.
        self.last_char = False

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

    Parameters
    ----------
    word_list: arr
        an array that includes the words to be processed to develop a tree.

    Attributes
    ----------
    word_list: char
        an array that includes the words to be processed to develop a tree.
    root: node
        the root node is an empty node as the problem stated.

    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.
        """
        # YOUR CODE HERE
        self.word_list = word_list
        #As the problem stated, the root node is an empty node.
        self.root = Node('')
        #Generates the tree based on the word inputs character by character.
        for i in self.word_list:
            self.insert(i)

    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.
        """
        # YOUR CODE HERE
        #As the example shows, capitalize all letters.
        #This avoids potential problems come from lower/upper cases.
        word = word.upper()
        #Initial node, where the program starts, is the root node
        node = self.root

        for now in word:
            #if one of the children is the character that is identical to␣
↪current character:
            if now in node.children:
                #go to that child
                node = node.children[now]
            #if there's no children, make a new child
            else:
                #new node called child
                child = Node(now)
                #make this child an object
                node.children[now] = child
                #go to that child
                node = node.children[now]

        #once all characters are inserted, last node becomes true.
        node.last_char = 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
        """
        # YOUR CODE HERE
        #As the example shows, capitalize all letters.
        #This avoids potential problems come from lower/upper cases.
        word = word.upper()
        #Initial node, where the program starts, is the root node
        node = self.root
        #repeat letter by letter to find the word
        for now in word:
            #if there's matched character, follow that path
            if now in node.children:
                node = node.children[now]
            #if there's no child, this means there's no word.
            else:
                return False
        #double check whether the node is the last character
        if node.last_char:
            return True
        else:
            return False

    def word_storage(self, start, word, words):
        """Get all words and put them in a storage

        Parameters
        ----------
        start: node
            starting node
        word: str
            the word that the code currently works on
        words: arr
```

```
            list of the words

        Returns
        -------
        words: arr
            list of the words
        """
        #This indicates that our starting node as a root node
        node = start

        #if the node has a children
        if node.children:
            #the code should check every child
            for char in node.children:
                #add this letter to form a word
                new_word = word + char

                #if the node is the last character:
                if node.children[char].last_char == True:
                    #add it to the words as that is a word
                    words.append(new_word)

                #keep move down the tree as there might be more words
                self.word_storage(node.children[char], new_word, words)
        #reset word to start again
        word = ''

        return words

    def alphabetical_list(self):
        """Delivers the content of the trie in alphabetical order.

        You can create other methods if it helps you,
        but the tests should use this one.

        Returns
        ----------
        list
            List of strings, all words from the trie in alphabetical order.
        """
        # YOUR CODE HERE
        node = self.root
        return sorted(self.word_storage(node, node.now, []))
```

[5]:
```
# intiate the test by uncommenting one of the lines below, depending on your␣
↪approach
```

```
wordbank = """Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis␣
 ↪pulvinar. Class aptent taciti sociosqu ad
litora torquent per conubia nostra, per inceptos hymenaeos. Nunc dapibus tortor␣
 ↪vel mi dapibus sollicitudin. Etiam quis
quam. Curabitur ligula sapien, pulvinar a vestibulum quis,
facilisis vel sapien.""".replace(",", "").replace(".", "").split()

trie = Trie(wordbank)
# trie = Node(wordbank)
# Change the following assertion into upper case as I used upper()
# assert trie.alphabetical_list() == ['a','ad','adipiscing','amet','aptent',
#                                     'class','consectetuer','conubia',
#                                     'curabitur','dapibus','dolor','duis',
#                                     'elit','etiam','facilisis','hymenaeos',
#                                     'inceptos','ipsum','ligula','litora',
#                                     'lorem','mi','nostra','nunc','per',
#                                     'pulvinar','quam','quis','sapien',
#                                     'sit','sociosqu','sollicitudin','taciti',
#                                     'torquent','tortor','vel','vestibulum']
assert trie.alphabetical_list() == ['A', 'AD', 'ADIPISCING', 'AMET', 'APTENT',
                                    'CLASS', 'CONSECTETUER', 'CONUBIA',
                                    'CURABITUR', 'DAPIBUS', 'DOLOR', 'DUIS',
                                    'ELIT', 'ETIAM', 'FACILISIS', 'HYMENAEOS',
                                    'INCEPTOS', 'IPSUM', 'LIGULA', 'LITORA',
                                    'LOREM', 'MI', 'NOSTRA', 'NUNC', 'PER',
                                    'PULVINAR', 'QUAM', 'QUIS', 'SAPIEN',
                                    'SIT', 'SOCIOSQU', 'SOLLICITUDIN', 'TACITI',
                                    'TORQUENT', 'TORTOR', 'VEL', 'VESTIBULUM']
```

[6]:
```
# YOUR NEW TESTS HERE
#Test Case 1, Common American Names
#This is the test for the random order
#https://www.ssa.gov/oact/babynames/
names = """Liam Olivia Noah Emma Oliver Ava Elijah Charlotte William Sophia
James Amelia Benjamin Isabella Lucas Mia
Henry Evelyn Alexander Harper""".replace(",", "").replace(".", "").split()
names_test = []
for a in names:
    names_test.append(a.upper())
trie_names = Trie(names_test)
assert trie_names.alphabetical_list() == ['ALEXANDER', 'AMELIA', 'AVA',␣
 ↪'BENJAMIN', 'CHARLOTTE',
                                   'ELIJAH', 'EMMA', 'EVELYN', 'HARPER',␣
 ↪'HENRY',
                                   'ISABELLA', 'JAMES', 'LIAM', 'LUCAS', 'MIA',
                                   'NOAH', 'OLIVER', 'OLIVIA', 'SOPHIA',␣
 ↪'WILLIAM']
```

```
#Test Case 2, Common American Names, Reversed.
#This is the test for the random order
names = """Liam Olivia Noah Emma Oliver Ava Elijah Charlotte William Sophia
James Amelia Benjamin Isabella Lucas Mia
Henry Evelyn Alexander Harper""".replace(",", "").replace(".", "").split()
names_test = []
for a in names:
    names_test.append(a.upper())
names_test = sorted(names_test, reverse = True)
trie_names_reverse = Trie(names_test)
#print(names_test)
assert trie_names_reverse.alphabetical_list() == ['ALEXANDER', 'AMELIA', 'AVA',␣
↪'BENJAMIN', 'CHARLOTTE',
                                    'ELIJAH', 'EMMA', 'EVELYN', 'HARPER',␣
↪'HENRY',
                                    'ISABELLA', 'JAMES', 'LIAM', 'LUCAS', 'MIA',
                                    'NOAH', 'OLIVER', 'OLIVIA', 'SOPHIA',␣
↪'WILLIAM']

#Test Case 3, Same Prefixes
#This is the test the words have same prefixes
#It also checks the numbers too!
prefix = """CS114 CS113 CS111 CS110 CS164 CS152""".replace(",", "").replace(".
↪", "").split()
prefix_test = []
for a in prefix:
    prefix_test.append(a.upper())
#print(prefix_test)
trie_prefix = Trie(prefix_test)
assert trie_prefix.alphabetical_list() == ['CS110', 'CS111', 'CS113', 'CS114',␣
↪'CS152', 'CS164']

#Test Case 4, Repetitions
#This case tests the repetitions
repetition = """Minerva uses Forum that is developed by Minerva and Minerva
students use Minerva Forum""".replace(",", "").replace(".", "").split()
repetition_test = []
for a in repetition:
    repetition_test.append(a.upper())
#print(repetition_test)
trie_repetition = Trie(repetition_test)
assert trie_repetition.alphabetical_list() == ['AND', 'BY', 'DEVELOPED',␣
↪'FORUM', 'IS',
                                               'MINERVA', 'STUDENTS', 'THAT',␣
↪'USE', 'USES']
```

```
#Test Case 5, 's
#This case tests the possesive word
possession = """Steven's iPhone and MacbookPro""".replace(",", "").replace(".",␣
↪"").split()
possession_test = []
for a in possession:
    possession_test.append(a.upper())
#print(possession_test)
trie_possession = Trie(possession_test)
assert trie_possession.alphabetical_list() == ['AND', 'IPHONE', 'MACBOOKPRO',␣
↪"STEVEN'S"]
```

Each test case is created to break the code – edge cases.

I used recursive approach here. It is because recursive approach here can benefit the code that one can start in the middle of the trie instead of going all the way up to the root node.

For example, if one is checking AND and ANT, when one finishes AN, one can just start from there to go D or T. If one goes back to all the way upto the root node it will take more time and make the program slower. Since recursive approach here holds what one has checked already, it also helps to keep track on what has been processed rather than losing what has been processed.

Personally, it is also easier to debug because I only had to debug inside of one function.

If it was an iterative approach, the approach pre-order traversal / depth-first search approach is much more complicated issue than it can be implemented in recursive approach.

Also, as mentioned above, the word length is not too long, using recursive does not damage the cost a lot.

1.6 Q4: 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), but consider it an extra challenge and let us know if you believe you managed to solve 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. 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 make adjustments to this "new version". The first cell has been locked to stop you from accidentally deleting the docstrings.

Completing this question well will help you to tackle Q5!

(Hint: 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 most common words is heaps — you are allowed to use the heapq library or another alternative for this task.)

[7]:
```
# depending on your choice of approach,
# add the method either to the Node or the Trie class
# VERSION 3 - K-most common words

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

    Parameters
    ----------
    character: char
        a character from the word that will be handled inside of the node.
    Attributes
    ----------
    now: char
        the current value(character) in node that is specifically handled.
    children: dict
        look for the following nodes which is the children of the current node
    last_char: bool
        The boolean tells whether the current node is the last character or not␣
↪of the word
    frequency: int
        number of words being used repeatedly respectively
    """

    def __init__(self, character):
        """This class represents one node of a trie tree.

        Parameters
        ----------
        character: char
            a character from the word that will be handled inside of the node.
        """
        # YOUR CODE HERE
        #Tells the current character
        self.now = character
        #Gives the dictionary of the children
        self.children = {}
        #As a default, this code assume the current character is not the last␣
↪character.
        self.last_char = False
        #This frequncy will count the number of words being used repeatedly␣
↪respectively.
```

```
        self.frequency = 0

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

    Parameters
    ----------
    word_list: arr
        an array that includes the words to be processed to develop a tree.

    Attributes
    ----------
    word_list: char
        an array that includes the words to be processed to develop a tree.
    root: node
        the root node is an empty node as the problem stated.

    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.
        """
        # YOUR CODE HERE
        self.word_list = word_list
        #As the problem stated, the root node is an empty node.
        self.root = Node('')
        #Generates the tree based on the word inputs character by character.
        for i in self.word_list:
            self.insert(i)

    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.
        """
        # YOUR CODE HERE
        #As the example shows, capitalize all letters.
        #This avoids potential problems come from lower/upper cases.
        #This is updated to be lower as all the sample test cases are in lower␣
↪cases.
        word = word.lower()
        #Initial node, where the program starts, is the root node
        node = self.root

        for now in word:
            #if one of the children is the character that is identical to␣
↪current character:
            if now in node.children:
                #go to that child
                node = node.children[now]
            #if there's no children, make a new child
            else:
                #new node called child
                child = Node(now)
                #make this child an object
                node.children[now] = child
                #go to that child
                node = node.children[now]

        #once all characters are inserted, last node becomes true.
        node.last_char = True
        #As the word ends, count 1 up to the frequency.
        node.frequency += 1

    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
        """
        # YOUR CODE HERE
        #As the example shows, capitalize all letters.
        #This avoids potential problems come from lower/upper cases.
        #This is updated to be lower as all the sample test cases are in lower␣
↪cases.
        word = word.lower()
        #Initial node, where the program starts, is the root node
        node = self.root
        #repeat letter by letter to find the word
        for now in word:
            #if there's matched character, follow that path
            if now in node.children:
                node = node.children[now]
            #if there's no child, this means there's no word.
            else:
                return False
        #double check whether the node is the last character
        if node.last_char:
            return True
        else:
            return False

    def word_storage(self, start, word, words):
        """Get all words and put them in a storage

        Parameters
        ----------
        start: node
            starting node
        word: str
            the word that the code currently works on
        words: arr
            list of the words

        Returns
        -------
        words: arr
            list of the words
        """
        #This indicates that our starting node as a root node
        node = start

        #if the node has a children
        if node.children:
            #the code should check every child
```

```
            for char in node.children:
                #add this letter to form a word
                new_word = word + char

                #if the node is the last character:
                if node.children[char].last_char == True:
                    #add it to the words as that is a word
                    words.append(new_word)

                #keep move down the tree as there might be more words
                self.word_storage(node.children[char], new_word, words)
        #reset word to start again
        word = ''

        return words

    def alphabetical_list(self):
        """Delivers the content of the trie in alphabetical order.

        You can create other methods if it helps you,
        but the tests should use this one.

        Returns
        ----------
        list
            List of strings, all words from the trie in alphabetical order.
        """
        # YOUR CODE HERE
        node = self.root
        return sorted(self.word_storage(node, node.now, []))

    def frequency_list(self, start, word, freq_words):
        """Returns a list with the words and frequencies respectively.

        Parameters
        ----------
        start: node
            starting node
        word: str
            the word that the code currently works on
        freq_words: arr
            An array with the words and their frequencies
        Returns
        ----------
        freq_words: arr
            a list with the words and frequencies respectively.
        """
```

```
        #starting node
        node = start

        #if the node has a children
        if node.children:
            #the code should check every child
            for char in node.children:
                #add this letter to form a word
                new_word = word + char

                #if the node is the last character:
                if node.children[char].last_char == True:
                    #add it to the words as that is a word
                    freq_words.append((new_word, node.children[char].frequency))

                #keep move down the tree as there might be more words
                self.frequency_list(node.children[char], new_word, freq_words)
        #reset word to start again
        word = ''

        return freq_words

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

        You will have to tweak some properties of your existing code,
        so that it captures information about repeated insertion.

        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.
        """
        # YOUR CODE HERE
        #starting node
        node = self.root
        #store a list that we got from frequency_list as a variable
        common = self.frequency_list(self.root, node.now, [])
        #alphabetically sort the list
        common.sort(key = lambda x:x[0])
        #and nunmerically sorr the list
        common.sort(key = lambda x:x[1], reverse = True)

        return common[0:k]
```

[8]:
```
# depending on your choice of approach,
# uncomment one of the lines in the for loop to initiate the test

# you might have to run 'pip install requests' before running this cell
# since you're downloading data from an online resource
# please note this might take a while to run

# 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

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

# Note: There are cleaner and more concise ways to write the code above,
# but this way it should be easily understandable.
```

[9]:
```
# YOUR NEW TESTS HERE
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)
    #Used google spread sheet and its count/countif function
    #to count the words and make a list
    #Test Case 1
    #If one tries to get no k-common words.
    if speaker == 'Kennedy':
        Kennedy = []
        assert trie.k_most_common(0) == Kennedy
    #Test Case 2
    #If one tries to get the most common word of the speech
    elif speaker == 'Thunberg':
        Thunberg = [('you', 22)]
        assert trie.k_most_common(1) == Thunberg
    #Test Case 3
    #If one tries to get every frequency for all words
    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), ('day', 12),
                ('ring', 12), ('dream', 11), ('come', 10), ('every', 10),
                ('nation', 10), ('back', 9), ('for', 9), ('go', 9),
                ('today', 9), ('able', 8), ('are', 8), ('by', 8),
                ('justice', 8), ('must', 8), ('satisfied', 8),
                ('their', 8), ('you', 8), ('all', 7), ('together', 7),
                ('when', 7), ('but', 6), ('cannot', 6), ('it', 6),
                ('long', 6), ('men', 6), ('now', 6), ('there', 6),
                ('white', 6), ('america', 5), ('check', 5), ('children', 5),
                ('faith', 5), ('free', 5), ('great', 5), ('has', 5),
                ('my', 5), ('new', 5), ('on', 5), ('shall', 5),
                ('time', 5), ('which', 5), ('years', 5), ('american', 4),
                ('an', 4), ('at', 4), ('black', 4), ('can', 4), ('down', 4),
                ('hope', 4), ('hundred', 4), ('into', 4), ('land', 4),␣
↪('later', 4),
                ('mississippi', 4), ('mountain', 4), ('so', 4), ('still', 4),
                ('until', 4), ('up', 4), ('us', 4), ('who', 4), ('alabama', 3),
                ('brotherhood', 3), ('georgia', 3), ("god's", 3), ('here', 3),
                ('his', 3), ('injustice', 3), ('its', 3), ('join', 3),
                ('last', 3), ('little', 3), ('made', 3), ('make', 3),
```

```
                ('never', 3), ('no', 3), ('note', 3), ('out', 3),
                ('people', 3), ('rights', 3), ('rise', 3), ('sing', 3),
                ('some', 3), ('stand', 3), ('state', 3), ('sweltering', 3),
                ('they', 3), ('valley', 3), ("we've", 3), ('where', 3),
                ('words', 3), ('again', 2), ('allow', 2), ('am', 2),
                ('been', 2), ('believe', 2), ('boys', 2), ('brothers', 2),
                ('brutality', 2), ('came', 2), ('cash', 2), ('cities', 2),
                ('color', 2), ('content', 2), ('continue', 2), ('creative', 2),
                ('despair', 2), ('destiny', 2), ('dignity', 2), ('end', 2),
                ('even', 2), ('force', 2), ('former', 2), ('funds', 2),
                ('girls', 2), ('hands', 2), ('heat', 2), ('hill', 2),
                ('history', 2), ('if', 2), ('insufficient', 2), ('jail', 2),
                ('knowing', 2), ('liberty', 2), ('life', 2), ('like', 2),
                ('live', 2), ('meaning', 2), ('mighty', 2), ('mountainside', 2),
                ("negro's", 2), ('off', 2), ('physical', 2), ('places', 2),
                ('police', 2), ('promissory', 2), ('quest', 2), ('racial', 2),
                ('realize', 2), ('refuse', 2), ('say', 2), ('segregation', 2),
                ('slaves', 2), ('sons', 2), ('south', 2), ('stone', 2),␣
↪('struggle', 2),
                ('suffering', 2), ('thee', 2), ('those', 2), ('true', 2),␣
↪('urgency', 2),
                ('vote', 2), ('walk', 2), ('was', 2), ('work', 2), ('would',␣
↪2), ('york', 2),
                ("'tis", 1), ('**we', 1), ('ago', 1), ('ahead', 1),␣
↪('alleghenies', 1),
                ('almighty', 1), ('alone', 1), ('also', 1), ('always', 1),␣
↪('architects', 1),
                ('areas', 1), ('asking', 1), ('autumn', 1), ('awakening', 1),␣
↪('bad', 1),
                ('bank', 1), ('bankrupt', 1), ('basic', 1), ('battered', 1),␣
↪('beacon', 1),
                ('beautiful', 1), ('become', 1), ('beginning', 1), ('believes',␣
↪1),
                ('bitterness', 1), ('blow', 1), ('bodies', 1), ('bound', 1),
                ('bright', 1), ('business', 1), ('california', 1), ('capital',␣
↪1),
                ('captivity', 1), ('carolina', 1), ('catholics', 1), ('cells',␣
↪1),
                ('chains', 1), ('changed', 1), ('character', 1), ('citizens',␣
↪1),
                ('citizenship', 1), ('city', 1), ('civil', 1), ('colorado', 1),
                ('community', 1), ('concerned', 1), ('condition', 1),␣
↪('conduct', 1),
                ('constitution', 1), ('cooling', 1), ('corners', 1),␣
↪('country', 1),
```

```
                ('created', 1), ('creed', 1), ('crippled', 1), ('crooked', 1),␣
↪('cup', 1),
                ('curvaceous', 1), ('dark', 1), ('daybreak', 1),␣
↪('declaration', 1),
                ('decree', 1), ('deeds', 1), ('deeply', 1), ('defaulted', 1),
                ('degenerate', 1), ('demand', 1), ('democracy', 1),␣
↪('demonstration', 1),
                ('desolate', 1), ('devotees', 1), ('died', 1), ('difficulties',␣
↪1),
                ('discipline', 1), ('discontent', 1), ('discords', 1),␣
↪('discrimination', 1),
                ('distrust', 1), ('dramatize', 1), ('drinking', 1),␣
↪('dripping', 1),
                ('drug', 1), ('emancipation', 1), ('emerges', 1), ('engage',␣
↪1), ('engulfed', 1),
                ('equal', 1), ('equality', 1), ('evidenced', 1), ('exalted',␣
↪1), ('exile', 1),
                ('face', 1), ('fall', 1), ('fatal', 1), ('fathers', 1),␣
↪('fatigue', 1),
                ('fierce', 1), ('finds', 1), ('five', 1), ('flames', 1),␣
↪('flesh', 1),
                ('forever', 1), ('foundations', 1), ('four', 1), ('fresh', 1),␣
↪('friends', 1),
                ('gain', 1), ('gaining', 1), ('gentiles', 1), ('ghetto', 1),␣
↪('ghettos', 1),
                ('give', 1), ('given', 1), ('glory', 1), ('god', 1),␣
↪('governor', 1),
                ('gradualism', 1), ('granted', 1), ('greatest', 1),␣
↪('guaranteed', 1),
                ('guilty', 1), ('had', 1), ('hallowed', 1), ('hamlet', 1),␣
↪('hampshire', 1),
                ('happens', 1), ('happiness', 1), ('happy', 1), ('hatred', 1),␣
↪('having', 1),
                ('he', 1), ('heavy', 1), ('heightening', 1), ('heights', 1),␣
↪('heir', 1),
                ('her', 1), ('hew', 1), ('high', 1), ('highways', 1), ('hills',␣
↪1),
                ('hilltops', 1), ('himself', 1), ('hold', 1), ('honoring', 1),␣
↪('horrors', 1),
                ('hotels', 1), ('independence', 1), ('inextricably', 1),␣
↪('insofar', 1),
                ('instead', 1), ('interposition', 1), ('invigorating', 1),␣
↪('island', 1),
                ('jangling', 1), ('jews', 1), ('joyous', 1), ('judged', 1),␣
↪('languished', 1),
```

```
                ('larger', 1), ('lead', 1), ('leads', 1), ('left', 1),␣
↪('legitimate', 1),
                ('lift', 1), ('light', 1), ('lips', 1), ('lives', 1),␣
↪('lodging', 1),
                ('lonely', 1), ('lookout', 1), ('lord', 1), ('louisiana', 1),␣
↪('low', 1),
                ('luxury', 1), ('magnificent', 1), ('majestic', 1),␣
↪('manacles', 1), ('many', 1),
                ('march', 1), ('marked', 1), ('marvelous', 1), ('material', 1),␣
↪('meeting', 1),
                ('midst', 1), ('militancy', 1), ('millions', 1), ('mobility',␣
↪1), ('molehill', 1),
                ('moment', 1), ('momentous', 1), ('motels', 1), ('mountains',␣
↪1), ('narrow', 1),
                ("nation's", 1), ('needed', 1), ('neither', 1), ('night', 1),␣
↪('nineteen', 1),
                ('nor', 1), ('northern', 1), ('nothing', 1), ('nullification',␣
↪1), ('oasis', 1),
                ('obligation', 1), ('obvious', 1), ('ocean', 1), ('old', 1),␣
↪('only', 1),
                ('only**', 1), ('opportunity', 1), ('oppression', 1), ('or',␣
↪1), ('overlook', 1),
                ('own', 1), ('owners', 1), ('palace', 1), ('pass', 1), ('path',␣
↪1),
                ('pennsylvania', 1), ('persecution', 1), ("pilgrim's", 1),␣
↪('place', 1),
                ('plain', 1), ('plane', 1), ('pledge', 1), ('poverty', 1),␣
↪('pray', 1),
                ('presence', 1), ('pride', 1), ('process', 1), ('proclamation',␣
↪1),
                ('prodigious', 1), ('promise', 1), ('promises', 1),␣
↪('prosperity', 1),
                ('protest', 1), ('protestants', 1), ('pursuit', 1),␣
↪('quicksands', 1),
                ('racists', 1), ('real', 1), ('reality', 1), ('red', 1),␣
↪('redemptive', 1),
                ('remind', 1), ('republic', 1), ('rest', 1), ('returns', 1),␣
↪('revealed', 1),
                ('revolt', 1), ('riches', 1), ('right', 1), ('righteousness',␣
↪1), ('rightful', 1),
                ('robbed', 1), ('rock', 1), ('rockies', 1), ('rolls', 1),␣
↪('rooted', 1), ('rough', 1),
                ('rude', 1), ('sacred', 1), ('sadly', 1), ('satisfy', 1),␣
↪('score', 1), ('seared', 1),
```

```
                ('security', 1), ('see', 1), ('seek', 1), ('selfevident', 1),␣
↪('selfhood', 1),
                ('sense', 1), ('shadow', 1), ('shake', 1), ('shameful', 1),␣
↪('signed', 1),
                ('signing', 1), ('signs', 1), ('sisters', 1), ('sit', 1),␣
↪('situation', 1),
                ('sixtythree', 1), ('skin', 1), ('slave', 1), ('slopes', 1),␣
↪('slums', 1),
                ('smaller', 1), ('snowcapped', 1), ('society', 1), ('solid',␣
↪1), ('somehow', 1),
                ('something', 1), ('soul', 1), ('speed', 1), ('spiritual', 1),␣
↪('spot', 1),
                ('staggered', 1), ('stating', 1), ('steam', 1), ('storms', 1),␣
↪('straight', 1),
                ('stream', 1), ('stripped', 1), ('summer', 1), ('sunlit', 1),␣
↪('sweet', 1),
                ('symbolic', 1), ('symphony', 1), ('table', 1), ('take', 1),␣
↪('tennessee', 1),
                ('thank', 1), ('these', 1), ('thirst', 1), ('though', 1),␣
↪('threshold', 1),
                ('tied', 1), ('tomorrow', 1), ('tranquility', 1),␣
↪('tranquilizing', 1),
                ('transform', 1), ('transformed', 1), ('travel', 1), ('trials',␣
↪1),
                ('tribulations', 1), ('truths', 1), ('turn', 1),␣
↪('unalienable', 1),
                ('unearned', 1), ('unmindful', 1), ('unspeakable', 1), ('upon',␣
↪1),
                ('usual', 1), ('vast', 1), ('vaults', 1), ('veterans', 1),␣
↪('vicious', 1),
                ('victim', 1), ('village', 1), ('violence', 1), ('wallow', 1),␣
↪('warm', 1),
                ('waters', 1), ('well', 1), ('were', 1), ('what', 1),␣
↪('whirlwinds', 1),
                ('whites', 1), ('whose', 1), ('winds', 1), ('withering', 1),␣
↪('wrongful', 1),
                ('wrote', 1), ('yes', 1), ('your', 1)]
        assert trie.k_most_common(10000000) == King
#Test Case 4
#Gettysburg Address
#Got the speech from: https://rmc.library.cornell.edu/gettysburg/good_cause/
 ↪transcript.htm
#I counted manually the top 3 common words.
gettysburg = '''Four score and seven years ago our fathers
brought forth upon this continent, a new nation, conceived
in Liberty, and dedicated to the proposition that all men are created equal.
```

```
Now we are engaged in a great civil war, testing whether that nation,
or any nation so conceived and so dedicated, can long endure. We are met on a␣
 ↪great battle-field
of that war. We have come to dedicate a portion of that field, as a final␣
 ↪resting place
for those who here gave their lives that that nation might live. It is␣
 ↪altogether fitting
and proper that we should do this. But, in a larger sense, we can not dedicate-
we can not
consecrate we can not hallow this ground. The brave men, living and dead, who␣
 ↪struggled here,
have consecrated it, far above our poor power to add or detract. The world will␣
 ↪little note,
nor long remember what we say here, but it can never forget what they did here.
It is for us the living, rather, to be dedicated here to the unfinished work␣
 ↪which
they who fought here have thus far so nobly advanced.
It is rather for us to be here dedicated to the great task
remaining before us-that from these honored dead we
take increased devotion to that cause for which they gave the last full measure␣
 ↪of
devotion-that we here highly resolve that these dead shall
not have died in vain-that this nation, under God, shall have
a new birth of freedom-and that government of the people, by the people,
for the people, shall not
perish from the earth.'''.replace(",", "").replace(".", "").replace(";", "").
 ↪split()
trie = Trie(gettysburg)
assert trie.k_most_common(3) == [('the', 11), ('that', 10), ('we', 9)]
```

These are all edge cases that can potentially break the code.

The first case is designed to see if the k-common can result an empty list.

The second case is designed to see when one finds the most common word.

The third case is designed to see if k-common works with a huge number.

The fourth case is designed to see if this code workds for a speech that is not given as a default.

1.7 Q5: 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 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 following tests. 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 specifically evaluate 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?

(Hint: 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.)

This is a thoroughly frequentist approach to the problem, which is not the only method, and in many cases not the ideal method. 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!

[10]:
```
# depending on your choice of approach,
# add this method to your Node or Trie class
# VERSION 4 - Autocomplete

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

    Parameters
    ----------
    character: char
        a character from the word that will be handled inside of the node.
    Attributes
    ----------
    now: char
        the current value(character) in node that is specifically handled.
    children: dict
        look for the following nodes which is the children of the current node
    last_char: bool
        The boolean tells whether the current node is the last character or not␣
 ↪of the word
    frequency: int
        number of words being used repeatedly respectively
    """

    def __init__(self, character):
        """This class represents one node of a trie tree.

        Parameters
        ----------
        character: char
            a character from the word that will be handled inside of the node.
        """
        # YOUR CODE HERE
```

```
        #Tells the current character
        self.now = character
        #Gives the dictionary of the children
        self.children = {}
        #As a default, this code assume the current character is not the last␣
 ↪character.
        self.last_char = False
        #This frequncy will count the number of words being used repeatedly␣
 ↪respectively.
        self.frequency = 0

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

    Parameters
    ----------
    word_list: arr
        an array that includes the words to be processed to develop a tree.

    Attributes
    ----------
    word_list: char
        an array that includes the words to be processed to develop a tree.
    root: node
        the root node is an empty node as the problem stated.

    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.
        """
        # YOUR CODE HERE
        self.word_list = word_list
        #As the problem stated, the root node is an empty node.
        self.root = Node('')
        #Generates the tree based on the word inputs character by character.
```

```
        for i in self.word_list:
            self.insert(i)

    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.
        """
        # YOUR CODE HERE
        #As the example shows, capitalize all letters.
        #This avoids potential problems come from lower/upper cases.
        #This is updated to be lower as all the sample test cases are in lower␣
↪cases.
        word = word.lower()
        #Initial node, where the program starts, is the root node
        node = self.root

        for now in word:
            #if one of the children is the character that is identical to␣
↪current character:
            if now in node.children:
                #go to that child
                node = node.children[now]
            #if there's no children, make a new child
            else:
                #new node called child
                child = Node(now)
                #make this child an object
                node.children[now] = child
                #go to that child
                node = node.children[now]

        #once all characters are inserted, last node becomes true.
        node.last_char = True
        #As the word ends, count 1 up to the frequency.
        node.frequency += 1

    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
        """
        # YOUR CODE HERE
        #As the example shows, capitalize all letters.
        #This avoids potential problems come from lower/upper cases.
        #This is updated to be lower as all the sample test cases are in lower␣
↪cases.
        word = word.lower()
        #Initial node, where the program starts, is the root node
        node = self.root
        #repeat letter by letter to find the word
        for now in word:
            #if there's matched character, follow that path
            if now in node.children:
                node = node.children[now]
            #if there's no child, this means there's no word.
            else:
                return False
        #double check whether the node is the last character
        if node.last_char:
            return True
        else:
            return False

    def word_storage(self, start, word, words):
        """Get all words and put them in a storage

        Parameters
        ----------
        start: node
            starting node
        word: str
            the word that the code currently works on
        words: arr
            list of the words

        Returns
        -------
```

```
        words: arr
            list of the words
        """
        #This indicates that our starting node as a root node
        node = start

        #if the node has a children
        if node.children:
            #the code should check every child
            for char in node.children:
                #add this letter to form a word
                new_word = word + char

                #if the node is the last character:
                if node.children[char].last_char == True:
                    #add it to the words as that is a word
                    words.append(new_word)

                #keep move down the tree as there might be more words
                self.word_storage(node.children[char], new_word, words)
        #reset word to start again
        word = ''

        return words

    def alphabetical_list(self):
        """Delivers the content of the trie in alphabetical order.

        You can create other methods if it helps you,
        but the tests should use this one.

        Returns
        ----------
        list
            List of strings, all words from the trie in alphabetical order.
        """
        # YOUR CODE HERE
        node = self.root
        return sorted(self.word_storage(node, node.now, []))

    def frequency_list(self, start, word, freq_words):
        """Returns a list with the words and frequencies respectively.

        Parameters
        ----------
        start: node
            starting node
```

```
        word: str
            the word that the code currently works on
        freq_words: arr
            An array with the words and their frequencies
        Returns
        ----------
        freq_words: arr
            a list with the words and frequencies respectively.
        """
        #starting node
        node = start

        #if the node has a children
        if node.children:
            #the code should check every child
            for char in node.children:
                #add this letter to form a word
                new_word = word + char

                #if the node is the last character:
                if node.children[char].last_char == True:
                    #add it to the words as that is a word
                    freq_words.append((new_word, node.children[char].frequency))

                #keep move down the tree as there might be more words
                self.frequency_list(node.children[char], new_word, freq_words)
        #reset word to start again
        word = ''

        return freq_words

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

        You will have to tweak some properties of your existing code,
        so that it captures information about repeated insertion.

        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.
        """
        # YOUR CODE HERE
        #starting node
        node = self.root
        #store a list that we got from frequency_list as a variable
        common = self.frequency_list(self.root, node.now, [])
        #alphabetically sort the list
        common.sort(key = lambda x:x[0])
        #and numerically sort the list
        common.sort(key = lambda x:x[1], reverse = True)

        return 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.
        """
        # YOUR CODE HERE
        #starting node
```

```
        node = self.root

        #if there's no prefix, return itself
        if prefix == '':
            return prefix

        #iterate letter by letter in the prefix
        for char in prefix:
            #the code should check every child
            if char in node.children:
                #go to that child
                node = node.children[char]
            #if there's no match, no word is there
            #therefore return itself
            else:
                return prefix
        #as recursively call the function, one of the possible outcome can be␣
 ↪erased.
        #Therefore, save it as a variable
        related_words = [(prefix, node.frequency)]
        #and then save it to the list
        dictionary = self.frequency_list(node, prefix, related_words)
        #get the frequency to have most common match
        dictionary.sort(key = lambda x:x[1], reverse = True)
        #return the best related word
        return dictionary[0][0]
```

[11]:
```
# depending on your choice of approach, uncomment one of the lines below
# The Complete Works of William Shakespeare is a LARGE book,
# so the code might take a while to run

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 != ""]

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'
```

[12]:
```
# YOUR NEW TESTS HERE
#Test Case 1
#I believe there's no my name in Shakesphere
#It should return my name itself
assert SH_trie.autocomplete('StevenHYang') == 'StevenHYang'
#Test Case 2
#An empty prefix
#It should return empty
assert SH_trie.autocomplete('') == ''
#Test Case 3
#When the input is just one letter
assert SH_trie.autocomplete('s') == 'so'
#Test Case 4
#If the word is going over the match and more characters are added
assert SH_trie.autocomplete('venusandjupiter') == 'venusandjupiter'
```

[13]:
```
SH_trie.autocomplete('tapo')
#I wanted to have pentapolis as a result.
#As you can see in this example, this algorithm does NOT catch the characters␣
 ↪in between of the words.
#It is because of the trie structure that starts from the first character.
#This can be a potential room of improvement.
```

[13]:
```
'tapo'
```

These are all edge cases that can potentially break the code.

The first test case tests if there's no matched word at all.

The second test case tests if there's an empty prefix.

The third test case tests if the input is just one letter and still can find the best match.

The fourth test case tests if the input matches until the end of the word but has more characters after the match.

First, I tried to understand the algorithm in general when one uses the dictionary. I tried some words like Minerva, computer, and apple at https://www.oxfordlearnersdictionaries.com/. During this trial, I understood that the algorithm returns the most commonly searched word or the word that is used frequently.

The complexity of this can be calculated as similar to the previous problems.

Setting a root node and creating an empty list is O(1) an each, as this is a single behavior.

As mentioned above, since the algorithm handles the words letter by letter, the loop checks the word in the tree takes O(l), where l is the length of the word.

To see whether the prefix is a word or not (to see whether that's empty or not) take O(1).

As there could be multiple words that share the prefixes, one must understand that the complexity would be O(l) * O(n), where n is the number of possible words. It is because this searching happens simultaneously.

Finding the most frequent word takes O(n) as it analyzes only in the list where all the most frequent words live.

To add up all the complexities described above:

T(n) = 3* O(1) + O(l) + O(l)*O(n) + O(n)

As the length of the word is not a significant factor that affects the whole algorithm, this takes O(n) as the leading complexity. Therefore, T(n) = O(n).

In this case, using an autocomplete is better than having max heaps because the complexity of max heaps is O(n log n). Of course, n log n is greater than n if n is greater than 10. In reality, having a dictionary that includes only or less than ten words is meaningless because one would rather memorize the whole ten words rather than make this massive effort to make a dictionary. Assume that we will have more than ten words in the dictionary; n is much smaller than n log n. Therefore, it is better to use autocomplete.

1.8 HC Appendix

1. #strategize: I explained the strengths and weaknesses of having only one class and two separate classes. This strategy worked because it made the debugging process much easier and less time-consuming, which is amazing when one does the assignment. This kind of strategizing skill is significant before hard coding; because it depends on how one starts the problem and approaches. The future codes and algorithms that one should handle come with very different difficulties.

2. #analogies: Rather than giving many details of the BST, I used this HC to address the similarities between the BST and the tries effectively. This is helpful because when a person who does not have a strong background in algorithms reads this assignment, one can easily understand some parts they should know from BST but not all. Of course, it is much more helpful for one if they know a lot of BST; however, it is also important to effectively address some points that one can understand real quick to follow the rest of the assignment.

3. #algorithms: I identified an appropriate algorithmic strategy, a recursive approach, as it is better to start from the middle rather than going back to the root node. I clearly commented on codes so one can easily understand every step of the code.

1.9 Outside References

1. McDowell, G. L. (2016, September 27). Data Structures: Tries. YouTube. https://www.youtube.com/watch?v=zIjfhVPRZCg&feature=youtu.be
2. Tech Dose. (2020, January 15). Auto complete feature using trie. YouTube. https://www.youtube.com/watch?v=DfkLGiW8vNA&feature=youtu.be

3. The Code Mate. (2020, August 10). TRIE - Implement an Auto Complete System - Design a Data Structure | The Code Mate. YouTube. https://www.youtube.com/watch?v=E_IOs3buLOQ&feature=youtu.be

Special thanks to Ha, the CS110 TA and Robert for helping me to understand the problems, and my roommates for giving such a heartwarming support until I finish this assignment.

Very Special Thanks to Prof. Richard regard of the outside classroom support and understanding my personal circumstances.