The Problem

For this assignment, I want to create an algorithm that stores the words from the lyrics of your favorite songs. This is a very interesting algorithm, as it shows different applications of data structures, and helps me practice, what I believe, is one of the most important skills for coding, working with pre-existing libraries and API.
For this application, I'll be comparing 2 different data structures for storing words and counting them, and I will also word with the Tekore library, an API that connects python with data from Spotify, and the Lyrics Genius library, another API that connects python with the data from the website Genius, to get the lyrics of songs.

Modifying the trie structure

In this first part, I'm using the feedback from my previous assignment to modify the Trie data-structure for it to be more efficient.
The first modification was storing the children nodes in dictionaries (hash tables) instead of lists, which decreases the time complexity of the lookup from $O(n)$ to $O(1)$.
The second modification was using a heap (with the heapq library) to store the list of repetitions instead of a list. This decreases the time complexity of the k_most_common from $O(nlogn)$ to $O(klogn)$. That is because by storing in a list we have to sort the whole list to get the k most common, and even the best sorting algorithm we've seen has a time complexity of $O(nlogn)$. However, if we store on a heap, we have $O(1)$ to get the smallest (or biggest) element and $O(logn)$ to take this element out of the heap in a way that we keep the heap properties, so if we do this k times, we have $O(klogn)$.

In [1]:

```
import heapq

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

    Parameters
    ----------
    The parameters for the Node class are not predetermined.
    However, you will likely need to create one or more of them.
    """

    def __init__(self, data = None, parent = None, count = 0):

        self.data = data
        self.parent = parent
        self.children = {}
        self.word_end = False
        self.word_count = count

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

    Parameters
    ----------
    The parameters for Trie's __init__ are not predetermined.
    However, you will likely need one or more of them.

    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.
    insert_word_list(self)
        inserts all the words in the word list in the trie.
    list_of_repetitions(self, root, wordsInOrder=[]):
        give a list of al the words in the trie and how many times they repeated
    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()

        if word_list:
            self.insert_word_list()

    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.
        """
        word = word.lower()  #makes all the letters lower-case
        current = self.root   #current node is the root
        for letter in word:   #iterate through the word

            #checking if the current letter exists in the current node children
            if letter in current.children:
                current = current.children[letter] #current update

            else:
                newLetter = Node(data=letter,parent=current)  #create new node for the letter
                current.children[letter] = newLetter #add the node to the children
                current = newLetter   #updates current

        current.word_end = True   #when its the last letter, set it to be the end of a word
        current.word_count += 1    #increasses the word count


    def insert_word_list(self):
        """inserts all the words in the word list in the trie.

        Parameters
        ----------
        None

        Returns
        -------
```

```
        None
        """
        for word in self.word_list:
            self.insert(word)


    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.
        """

        word = word.lower()   #makes all the letters lower-case
        current = self.root     #current node is the root
        for letter in word:     #iterate through the word

            #checking if the current letter exists in the current node children
            if letter in current.children:
                current = current.children[letter] #current update

            else:
                return False

        #checks if it is a prefix or a word
        if current.word_end:
            return True
        else:
            return False


    def list_of_repetitions(self, root, wordsInOrder=[]):
        """give a list of al the words in the trie and how many times they repeated

        Parameters
        ----------
        Node - root

        List of tupples - wordsInOrder

        Returns
        -------
        List of tupples
        """

        current = root  #current node is the root
        if current.children != {}:   #base case (when current node has no children)
            for child in list(current.children.values()):   #iterate through currents children
                if child.word_end:    #checks if the child is the end of a word
                    letter = child   #stores the child node in the var letter
                    word = ''       #initializing the word
                    while letter.parent is not None:     #goes up in the trie until it reaches the root
                        word += letter.data    #appends the letter in the word
                        letter = letter.parent   #updates the letter
                    word = word[::-1]    #since we climbed up the trie, we need to invert the word
                    heapq.heappush(wordsInOrder,(-child.word_count, word))  #inserts the word and the count in the
                self.list_of_repetitions(child,wordsInOrder)   # recursivelly calls the the function again using th
        return wordsInOrder


    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)]
```

```
180
181         This means that the word 'the' has appeared 154 times in the inserted text.
182         The second and third most common words both appeared 122 times.
183         """
184         completeList = self.list_of_repetitions(self.root, wordsInOrder=[])   #get the whole priority queue of word
185         km = [heapq.heappop(completeList) for i in range(k)]   #gets the k smallest (the values for repetion are ne
186         km = [(i[1],-i[0]) for i in km]
187         return km
```

In [2]:

```
1 #simple testing
2 wordbank = "oi oi a teste teste teste".split()
3 trie = Trie(wordbank)
4 trie.k_most_common(1)
```

Out[2]: [('teste', 3)]

Doing the same functions, but with a hash table instead of trie

As my second data-structure, I decided to go with a hash table. I'm simply using the built-in dictionaries since this is way more efficient and less time-consuming than trying to come up with my own hash functions and methods to solve collisions.
I think that using a hash table here is a great solution since it has $O(1)$ time complexity for lookup and for insertion. It is also better memory-wise since instead of having an object with a dictionary inside for every letter of a word, it simply has one key and one value for every different word.

In [3]:

```
class HT:
    def __init__(self, word_list = None):
        """Creates the Hash Table instane, inserts initial words if provided.

        Parameters
        ----------
        word_list : list
            List of strings to be inserted upon creation.
        """
        self.word_list = word_list
        self.data = {}

        if word_list:
            self.insert_word_list()

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

        Parameters
        ----------
        word : str
            The word to be looked-up.

        Returns
        -------
        bool
            True if the word is present; False otherwise.
        """
        word = word.lower()   #makes all the letters lower-case
        return (word in self.data)


    def insert(self, word):
        """Inserts a word into the Hash Table.

        Parameters
        ----------
        word : str
            The word to be inserted.
        """

        word = word.lower()   #makes all the letters lower-case

        if self.lookup(word):
            self.data[word]+=1  #adds to the counter of the word if the word is already there
        else:
            self.data[word]=1   #simply uses a new slot of the dictionary if the word is not already there


    def insert_word_list(self):
        """inserts all the words in the word list in the Hash Table.

        Parameters
        ----------
        None

        Returns
        -------
        None
        """
        for word in self.word_list:
            self.insert(word)

    def k_most_common(self, k):
        """Finds k words inserted into the Hash Table 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.
        """

        k_most_common_keys = heapq.nlargest(k, self.data, key=self.data.get)   #gets the k largest elements, by cou
        return [(i,self.data[i]) for i in k_most_common_keys]
```

In [4]:
```
#simple testing
wordbank = "oi oi a teste teste teste".split()
ht = HT(wordbank)
ht.k_most_common(1)
```

Out[4]:
```
[('teste', 3)]
```

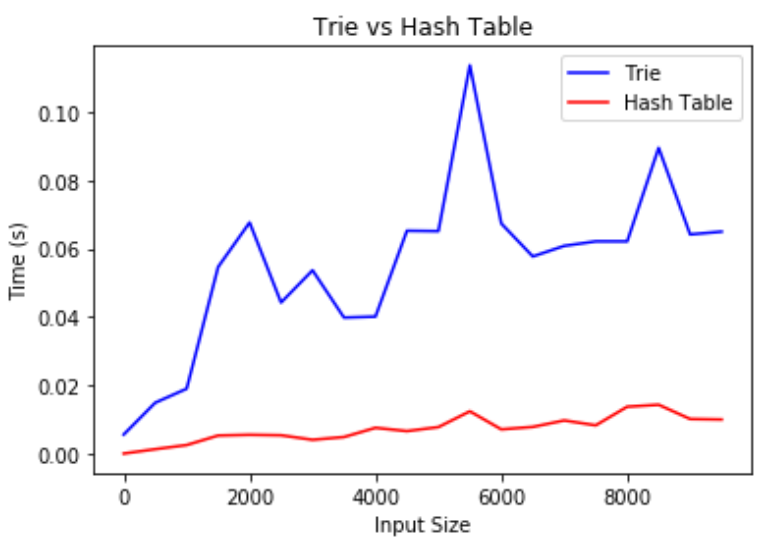
Comparing both data structures

In order to compare both structures, I'll use a more experimental approach. Using the big word bank from the previous assignment, I'll randomly (to make it less biased) select n words from it and measure the time that it takes for both structures to insert and give me the 1 most common and 1% (kind of) most common words, then I'll increase n and measure again. This will hopefully give me a graph that will model the asymptotic behavior of the functions.

In [5]:
```
#code adapted from the last assignment to get a good wordbank
import random
import string
import time
import matplotlib.pyplot as plt

import urllib.request
response = urllib.request.urlopen('http://bit.ly/CS110-Shakespeare')
bad_chars = [';', ',', '.', '?', '!', '1', '2', '3', '4', '5', '6', '7', '8', '9', '0', '_', '[', ']', '"', '&', "'
wordlist = []

for line in response:
    line = line.decode(encoding = 'utf-8')
    line = filter(lambda i: i not in bad_chars, line)
    words = "".join(line).split()
    for word in words:
        wordlist.append(word)
```

In [6]:
```
#this cell might take a couple of minutes to run, please wait until it is done to rune the next one

times_trie = []
times_HT = []

for i in range(1,10001,500):
    wordbank = random.sample(wordlist, i)

    time_trie = 0
    time_HT = 0

    for j in range(10):
        s = time.perf_counter()   #start timer
        trie = Trie(wordbank)
        trie.k_most_common(1)
        trie.k_most_common(i//100)
        e = time.perf_counter()  #end timer
        time_trie += (e-s)
    times_trie.append(time_trie/10)

    for j in range(10):
        s = time.perf_counter()   #start timer
        ht = HT(wordbank)
        ht.k_most_common(1)
        ht.k_most_common(i//100)
        e = time.perf_counter()  #end timer
        time_HT += (e-s)
    times_HT.append(time_HT/10)
```

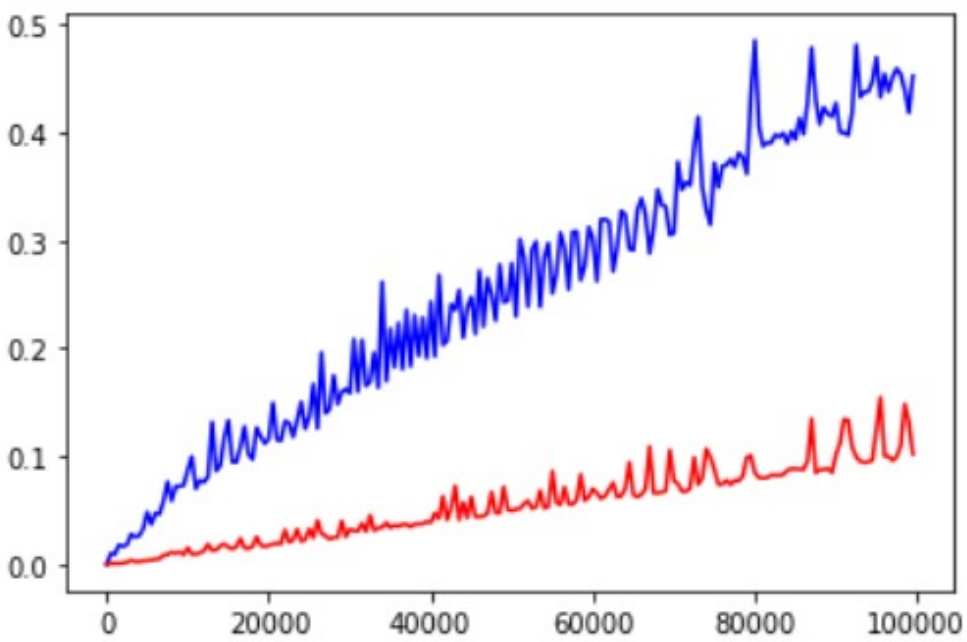
In [7]:
```
plt.plot(list(range(1,10001,500)),times_trie, c='blue')
plt.plot(list(range(1,10001,500)),times_HT, c='red')

plt.legend(['Trie','Hash Table'])
plt.xlabel('Input Size')
plt.ylabel('Time (s)')
plt.title('Trie vs Hash Table')

plt.show()
```



Since this code would take a very long time to run for a bigger number of words, here is an image of the same graph but with more data, so it will be easier to analize.

In [12]:
```
from IPython.display import Image
Image(filename="graph.jpg")
```

Out[12]:
```
Out[49]: [<matplotlib.lines.Line2D at 0x1de20258a58>]
```



The first we can see is that for every value of n, the hash table performs better. They both appear to have linear time complexity by this graph, however, the constant multiplier for the hash table is significantly lower.
If we analize both more analitically, we know that they both have $O(klogn)$ for the k_most_common, since they use the same basic "sort" to do it.
However, the hash table shines at the insertion by using a very simple constant complexity for each word, giving an overway complexiy of $O(n)$ for n words, while the Trie uses a more complicated code, which I meticulouslly described last assignment.
With these results, and the fact that the hash table code is simpler and more elegant, I chose to use it for my final implementation.

Experimenting with the Tekore API

I decide to use this open source library mainly for it's simplicity. It has a really complete documentation, well designed code that easilly connects you to the WEB API for Spotify , and even a Discord server to discuss problems.

https://pypi.org/project/tekore/ (https://pypi.org/project/tekore/)

In [13]:
```
pip install tekore
```

```
Requirement already satisfied: tekore in c:\programdata\anaconda3\lib\site-packages (3.4.1)
Requirement already satisfied: httpx<0.17,>=0.11 in c:\programdata\anaconda3\lib\site-packages (from tekore) (0.16.1)
Requirement already satisfied: httpcore==0.12.* in c:\programdata\anaconda3\lib\site-packages (from httpx<0.17,>=0.11->tekore) (0.12.2)
Requirement already satisfied: certifi in c:\programdata\anaconda3\lib\site-packages (from httpx<0.17,>=0.11->tekore) (2019.6.16)
Requirement already satisfied: sniffio in c:\programdata\anaconda3\lib\site-packages (from httpx<0.17,>=0.11->tekore) (1.2.0)
Requirement already satisfied: rfc3986[idna2008]<2,>=1.3 in c:\programdata\anaconda3\lib\site-packages (from httpx<0.17,>=0.11->tekore) (1.4.0)
Requirement already satisfied: h11==0.* in c:\programdata\anaconda3\lib\site-packages (from httpcore==0.12.*->httpx<0.17,>=0.11->tekore) (0.11.0)
Requirement already satisfied: idna; extra == "idna2008" in c:\programdata\anaconda3\lib\site-packages (from rfc3986[idna2008]<2,>=1.3->httpx<0.17,>=0.11->tekore) (2.8)
Note: you may need to restart the kernel to use updated packages.


WARNING: You are using pip version 20.0.2; however, version 20.3.1 is available.
You should consider upgrading via the 'C:\ProgramData\Anaconda3\python.exe -m pip install --upgrade pip' command.
```

In [14]:
```
import tekore as tk

#geting the autentication token
conf = ("47721e5988094299b2a980889eea6403", "8bccab59ae3b428496a12572fcb36ad6","https://sites.google.com/view/cs110
token = tk.prompt_for_user_token(*conf, scope=tk.scope.every)
spotify = tk.Spotify(token)

top_tracks = spotify.current_user_top_tracks(limit=10)  #gets the 10 top tracks
```

```
Opening browser for Spotify login...
Please paste redirect URL: https://sites.google.com/view/cs110-final-project/home?code=AQCe0J5aHKrOK3tIxtZEfMZNhydoAif
Y5Ha_cqeQymuu-VISMsJITXMxLcMIUh0RWxHA7-6zHrYRS09VZjKyAyKJD-b1O7v5laj73Gy2CmLP0W5nRNsIPqKQfujfRGIPon-ESGq-bDeU3S5JWHkSe
iT4xkerHUCYu8MJSOp1US55kVp_UbNq39EZSR_GeZdLI6Zo4r9Vs89Vp1QWQn0mx2tcW-1grqyTeyf8UuBms9vEfLvPMgTXTJGPML0qD-M7EndQJ5-4aQl
2rRBVwckv35oTsM0FtRkY-1pGo7-iLaCPiRU1QHcCaI6Gj8P1EbidN86HkxK4lUkbXOGQcdOqFoUoQaiq0cRIrwXp3klqvntwJA_RDzojl4xYY3A3qxjFF
RfOBPNg_HDHypYlpV_wQiaIMBT7haLWg8r-SE7U9bs_mHe9Y6kXkl-jPRF5epB_HIttMFCeHQ7uJmCTZAF4x9TsYVMMU-83cZBifueRjq3fVVvIbLGlvLi
WkXEk3KsxJDAP4x5xpLWYoMGUVOcNIR9PKMomXsAHBjE1mP3HNJ5cQuFKB24maPRweWZuTXrnTunLx5KdA1QMmaPkuUEkluyNW53Y3Tidr_kkPlclDwVX4
Z2Yk661nfKfLaSpusJ0QshkP8OC0m4kKsJpirPei94FJYyvMJtxRVHTOmXeITlDN7y0M6IpDQPh6UIPbr7jgCEtjKL-fevdVtCOqA&state=NgnZjkuykJ
JU__M1Doju4VdVD_W-G6g80uOXQVmCoBY (https://sites.google.com/view/cs110-final-project/home?code=AQCe0J5aHKrOK3tIxtZEfMZ
NhydoAifY5Ha_cqeQymuu-VISMsJITXMxLcMIUh0RWxHA7-6zHrYRS09VZjKyAyKJD-b1O7v5laj73Gy2CmLP0W5nRNsIPqKQfujfRGIPon-ESGq-bDeU3
S5JWHkSeiT4xkerHUCYu8MJSOp1US55kVp_UbNq39EZSR_GeZdLI6Zo4r9Vs89Vp1QWQn0mx2tcW-1grqyTeyf8UuBms9vEfLvPMgTXTJGPML0qD-M7End
QJ5-4aQl2rRBVwckv35oTsM0FtRkY-1pGo7-iLaCPiRU1QHcCaI6Gj8P1EbidN86HkxK4lUkbXOGQcdOqFoUoQaiq0cRIrwXp3klqvntwJA_RDzojl4xYY
3A3qxjFFRfOBPNg_HDHypYlpV_wQiaIMBT7haLWg8r-SE7U9bs_mHe9Y6kXkl-jPRF5epB_HIttMFCeHQ7uJmCTZAF4x9TsYVMMU-83cZBifueRjq3fVVv
IbLGlvLiWkXEk3KsxJDAP4x5xpLWYoMGUVOcNIR9PKMomXsAHBjE1mP3HNJ5cQuFKB24maPRweWZuTXrnTunLx5KdA1QMmaPkuUEkluyNW53Y3Tidr_kkP
lclDwVX4Z2Yk661nfKfLaSpusJ0QshkP8OC0m4kKsJpirPei94FJYyvMJtxRVHTOmXeITlDN7y0M6IpDQPh6UIPbr7jgCEtjKL-fevdVtCOqA&state=Ng
nZjkuykJJU__M1Doju4VdVD_W-G6g80uOXQVmCoBY)
```

In [15]:
```
#some songs have some info in the title that can make the website with the lyics confused, like "Here I Go Again (2
bad_for_song = ['(','-']

for i in top_tracks.items:
    for j in range(len(i.name)-1):
        if i.name[j] in bad_for_song:   #if I find a ( or a -, I delete whatever comes after it from the song title
            i.name = i.name[:j-1]
            break
    print(i.name,[j.name for j in i.artists])
```

```
Relaxa! ['Haikaiss', 'Neo Beats', 'Cortesia da Casa', 'Ursso']
Linda, Louca e Mimada ['Oriente', 'Rebeca']
Pride and Joy ['Stevie Ray Vaughan']
Sultans Of Swing ['Dire Straits']
Céu Azul ['Charlie Brown Jr.']
Philipa ['The Touré-Raichel Collective', 'Idan Raichel', 'Vieux Farka Touré']
Here I Go Again ['Whitesnake']
Beastly ['Vulfpeck']
Zero ['Liniker e os Caramelows']
Boredom ['Tyler, The Creator', 'Rex Orange County', 'Anna of the North']
```

Experimenting with LyricsGenius API

This was also the simplest API I found for working with lyrics of songs. Using an API of an existing database like this one is a much more effective and easier way of getting the job done without having to do a whole web search protocol from scratch in order to get some simple lyrics from a song.

https://pypi.org/project/lyricsgenius/ (https://pypi.org/project/lyricsgenius/)

In [16]:
```
pip install lyricsgenius
```

```
Requirement already satisfied: lyricsgenius in c:\programdata\anaconda3\lib\site-packages (2.0.2)
Requirement already satisfied: beautifulsoup4>=4.6.0 in c:\programdata\anaconda3\lib\site-packages (from lyricsgenius)
(4.7.1)
Requirement already satisfied: requests>=2.20.0 in c:\programdata\anaconda3\lib\site-packages (from lyricsgenius) (2.2
2.0)
Requirement already satisfied: soupsieve>=1.2 in c:\programdata\anaconda3\lib\site-packages (from beautifulsoup4>=4.6.
0->lyricsgenius) (1.8)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in c:\programdata\anaconda3\lib\site-packages
(from requests>=2.20.0->lyricsgenius) (1.24.2)
Requirement already satisfied: chardet<3.1.0,>=3.0.2 in c:\programdata\anaconda3\lib\site-packages (from requests>=2.2
0.0->lyricsgenius) (3.0.4)
Requirement already satisfied: certifi>=2017.4.17 in c:\programdata\anaconda3\lib\site-packages (from requests>=2.20.0
->lyricsgenius) (2019.6.16)
Requirement already satisfied: idna<2.9,>=2.5 in c:\programdata\anaconda3\lib\site-packages (from requests>=2.20.0->ly
ricsgenius) (2.8)
Note: you may need to restart the kernel to use updated packages.


WARNING: You are using pip version 20.0.2; however, version 20.3.1 is available.
You should consider upgrading via the 'C:\ProgramData\Anaconda3\python.exe -m pip install --upgrade pip' command.
```

In [17]:
```
import lyricsgenius

#token for the API
genius = lyricsgenius.Genius("CHIZ8mkPTjY4IJBKxF6dni13TDPlsBhxLN04YgXv1El3yBgGA8OOkpn3I35Eo0WK")

genius.response_format = 'plain' #setting the text as plain text


songs_with_lyrics = [genius.search_song(i.name, i.artists[0].name) for i in top_tracks.items] #gets the lyrics of m

songs_with_lyrics
```

```
Searching for "Relaxa!" by Haikaiss...
Done.
Searching for "Linda, Louca e Mimada" by Oriente...
Done.
Searching for "Pride and Joy" by Stevie Ray Vaughan...
Done.
Searching for "Sultans Of Swing" by Dire Straits...
Done.
Searching for "Céu Azul" by Charlie Brown Jr....
Done.
Searching for "Philipa" by The Touré-Raichel Collective...
No results found for: 'Philipa The Touré-Raichel Collective'
Searching for "Here I Go Again" by Whitesnake...
Done.
Searching for "Beastly" by Vulfpeck...
Done.
Searching for "Zero" by Liniker e os Caramelows...
Done.
Searching for "Boredom" by Tyler, The Creator...
Done.
```

Out[17]:
```
[('Relaxa!', 'Haikaiss'),
 ('Linda, Louca e Mimada', 'Oriente'),
 ('Pride and Joy', 'Stevie Ray Vaughan and Double Trouble'),
 ('Sultans of Swing', 'Dire Straits'),
 ('Céu Azul', 'Charlie Brown Jr.'),
 None,
 ('Here I Go Again', 'Whitesnake'),
 ('Beastly', 'Vulfpeck'),
 ('Zero', 'Liniker e os Caramelows'),
 ('Boredom', 'Tyler, The Creator')]
```

Combining all

In [18]:

```python
class LyricsTable:
    def __init__(self, word_list = None, bad_words = []):
        """Creates the Hash Table instane, inserts initial words if provided.

        Parameters
        ----------
        word_list : list
            List of strings to be inserted upon creation.

        bad_words : list
            List of words that won't be counted in the most common words.
        """
        self.word_list = word_list
        self.data = {}
        self.bad_words = bad_words

        if word_list:
            self.insert_word_list()

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

        Parameters
        ----------
        word : str
            The word to be looked-up.

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

        word = word.lower()   #makes all the letters lower-case
        return (word in self.data)


    def insert(self, word):
        """Inserts a word into the Hash Table.

        Parameters
        ----------
        word : str
            The word to be inserted.
        """

        word = word.lower()   #makes all the letters lower-case

        if self.lookup(word):   #if the word is there
            if word in self.bad_words:    #if it is one of the word I don't want to count in the list of repetitior
                self.data[word] -= 1
            else:
                self.data[word] += 1
        else:
            if word in self.bad_words:   #if it is one of the word I don't want to count in the list of repetitions
                self.data[word] = -1
            else:
                self.data[word] = 1


    def insert_word_list(self):
        """inserts all the words in the word list in the Hash Table.

        Parameters
        ----------
        None

        Returns
        -------
        None
        """
        for word in self.word_list:
            self.insert(word)

    def k_most_common(self, k):
        """Finds k words inserted into the Hash Table 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.
```

```
        """

        k_most_common_keys = heapq.nlargest(k, self.data, key=self.data.get)  #gets the k largest elements, by cou
        return [(i,self.data[i]) for i in k_most_common_keys]

    def repetitions(self, word):
        """finds the amount of times a word shows up in the HT.

        Parameters
        ----------
        word : str
            word you wnat to know the amount of times it shows up.

        Returns
        ----------
        int
            number of times the word shows up.

        """

        if self.lookup(word):  #if the word is in the HT
            return abs(self.data[word])  #return the counter of the word (abs because the bad words are negative)
        return 0

def initSpotify():
    """Initializes the autentication for the Spotify API.

        Parameters
        ----------
        None

        Returns
        ----------
        None
        """
    conf = ("47721e5988094299b2a980889eea6403", "8bccab59ae3b428496a12572fcb36ad6","https://sites.google.com/view/c
    token = tk.prompt_for_user_token(*conf, scope=tk.scope.every)
    spotify = tk.Spotify(token)

def getTopSongs(n, time_range):
    """Gets your top tracks from Spotify.

        Parameters
        ----------
        n : int
            number you tracks to get

        time_range : srt
            range of the top track (long_term (forever), medium_term (last 6 months), short_term (last month))

        Returns
        ----------
        list
            list if top tracks
        """
    top = spotify.current_user_top_tracks(limit=n, time_range=time_range)  #gets top tracks

    #some songs have some info in the title that can make the website with the lyics confused, like "Here I Go Agai
    bad_for_song = ['(','-']

    for i in top.items:
        for j in range(len(i.name)-1):
            if i.name[j] in bad_for_song: #if I find a ( or a -, I delete whatever comes after it from the song tit
                i.name = i.name[:j-1]
                break
    return top

def getLyrics(song):
    """Gets the lyrics of a song.

        Parameters
        ----------
        song : song object
            song

        Returns
        ----------
        song object
            song with the lyrics as an atribute
        None
            if no song
        """
    if song:
        return genius.search_song(song.name, song.artists[0].name)
    else:
        return None

def List_of_lyrics(n, time_range):
    """Creates a list of song lyrics from your top tracks on Spotify.

```

```
        Parameters
        ----------
        n : int
            number you tracks to get

        time_range : srt
            range of the top track (long_term (forever), medium_term (last 6 months), short_term (last month))

        Returns
        ----------
        list
            list if top tracks with lyrics as an atribute
        """

    initSpotify()

    top_songs = getTopSongs(n, time_range)

    return [getLyrics(i) for i in top_songs.items]

def createWordbank(n, time_range):
    """creates a bank of the words in the lyrics of your top tracks on Spotify

        Parameters
        ----------
        n : int
            number you tracks to get

        time_range : srt
            range of the top track (long_term (forever), medium_term (last 6 months), short_term (last month))

        Returns
        ----------
        list
            list words
        """
    lyrics = List_of_lyrics(n, time_range)  #list of songs with lyrics
    wordbank = []
    for i in lyrics:  #iterates through the songs
        if i:  #if there is a song
            #adds the words to the wordbank
            wordbank += i.lyrics.replace("!", "").replace("?", "").replace(".", "").replace(",", "").replace(";", "
    return wordbank
```

Feel free to play around with this part of the code and experiment with different numbers of top tracks, most common words, and time ranges.
For any authentication errors, please contact me, it might be that the keys have expired and I have to provide new ones.
If you do not use Spotify please refer to the output of the code in the pdf with outputs from my account.

In [19]:
```
wordbank = createWordbank(10,'long_term')

#excluding the words I don't want to count in my list of repetitions. Feel free to add more or take some out as you
bad_words = ['a','o','e','and','or','i','they','to','para','ele','ela','he','she','it','was','is','am','you','tu','
             'pra','the','que','do','eu','de','é','of',"she's","he's",'me','um','se','te','that','on', 'in', "i'm",
            'for', 'be','[chorus]',"i've",'so','this','if']


songTable = LyricsTable(wordbank,bad_words)

print(songTable.k_most_common(20))
```

```
Opening browser for Spotify login...
Please paste redirect URL: https://sites.google.com/view/cs110-final-project/home?code=AQAkDB1BLnDXqYO5zh8WaB7W1ZhqXzm
gTJsc85RfqoCSbvobdkPUBLIibSX2SlThOVVgZYO0yxUjEJRMXPel_pQegDpG4KQHxJUcDUB_DsbfnECL8qOVwis31FG4dxXbZyi5u0SnzPD42NF0cakyw
xSibOgGaLdRT0dOLNLZQ3vq4n2ajLr-0YS2ZkC603rminveMx6dePloLI35494cdY7z6MPVJd5KU01_VeQey_QasEKoxTBPru6FxOfpPCULMjxOlE77LW5
FiV9VS3-WTmk89hcNINF8awVA12n8Gu8ynhBE3IUQUloXFEEIhnvufW27IH4y73twX3uo2DHW-eY7xPonq-WnDX76pZOV0iKQ_XhF_23ALTA7hObEPJLig
2o_pds0BTSWFpyNQ-z9RuX-jq7y0vR_C6XOjUlF5mkB41nHbGq_N7zwDEEoNXCZ6CrjdLUdBgpMzcAZkuGuHanj95yYa2Tna6QFIa7VV2u7tic8kELrLCp
1A-NROD_PKviC4QLza81HSds871cQwVLa_uwzAHgNe1fbrq0UjEPcMffXd-3TCbnWCw4sAoRfyFgobq4lgOqchx_aFP7wrhm_EPZ1nHJOej5PyPz-cPYOx
mbiRLQq2jKWGz-Tq4Ysu3UvjC4AG_7PgRUZufFkJi1q1zfZOZ1BxLHTI3nBlv94QLd7408IA6eXZs2B_4NR8SdJ2Mz2FXwQ7a94Ew&state=YJoRG09FnI
p8jfA1fv1gRwKpV8dXsL21M74yfsFB69M (https://sites.google.com/view/cs110-final-project/home?code=AQAkDB1BLnDXqYO5zh8WaB7
W1ZhqXzmgTJsc85RfqoCSbvobdkPUBLIibSX2SlThOVVgZYO0yxUjEJRMXPel_pQegDpG4KQHxJUcDUB_DsbfnECL8qOVwis31FG4dxXbZyi5u0SnzPD42
NF0cakywxSibOgGaLdRT0dOLNLZQ3vq4n2ajLr-0YS2ZkC603rminveMx6dePloLI35494cdY7z6MPVJd5KU01_VeQey_QasEKoxTBPru6FxOfpPCULMjx
OlE77LW5FiV9VS3-WTmk89hcNINF8awVA12n8Gu8ynhBE3IUQUloXFEEIhnvufW27IH4y73twX3uo2DHW-eY7xPonq-WnDX76pZOV0iKQ_XhF_23ALTA7h
ObEPJLig2o_pds0BTSWFpyNQ-z9RuX-jq7y0vR_C6XOjUlF5mkB41nHbGq_N7zwDEEoNXCZ6CrjdLUdBgpMzcAZkuGuHanj95yYa2Tna6QFIa7VV2u7tic
8kELrLCp1A-NROD_PKviC4QLza81HSds871cQwVLa_uwzAHgNe1fbrq0UjEPcMffXd-3TCbnWCw4sAoRfyFgobq4lgOqchx_aFP7wrhm_EPZ1nHJOej5Py
Pz-cPYOxmbiRLQq2jKWGz-Tq4Ysu3UvjC4AG_7PgRUZufFkJi1q1zfZOZ1BxLHTI3nBlv94QLd7408IA6eXZs2B_4NR8SdJ2Mz2FXwQ7a94Ew&state=YJ
oRG09FnIp8jfA1fv1gRwKpV8dXsL21M74yfsFB69M)
Searching for "Passionfruit" by Drake...
Done.
Searching for "Relaxa!" by Haikaiss...
Done.
Searching for "Weird Fishes/ Arpeggi" by Radiohead...
Done.
Searching for "Fake Plastic Trees" by Radiohead...
Done.
Searching for "Pull Me Under" by Dream Theater...
Done.
Searching for "Whole Lotta Love" by Led Zeppelin...
Done.
Searching for "Linda" by Projota...
Done.
Searching for "Passarinhos" by Emicida...
Done.
Searching for "Here I Go Again" by Whitesnake...
Done.
Searching for "Fly Me To The Moon" by Frank Sinatra...
Done.
[('my', 36), ('no', 32), ('love', 24), ('não', 21), ('again', 15), ('whole', 15), ('here', 15), ('você', 14), ('all',
 14), ('go', 14), ('want', 13), ('out', 12), ('meu', 12), ('assim', 12), ('wears', 12), ('pull', 12), ('under', 12),
 ('baby', 12), ('lotta', 12), ('up', 11)]
```

In [21]:
```
songTable.repetitions('need')
```

Out[21]: 5

LOs and HCs

LOs

#DataStructes - Used, modified, and explained different data-structures, providing technical information about them. Also compared 2 different types of data-structures for the same application, using both an experimental and analytical approach.

#RandomizationTechniques - Used randomization techniques when doing the test to compare the data structures in order to get a less biased bank of words and reduce the chance of skewing my results in favor of one data-structure over the other.

#ComputationalSolutions - Effectively broke down the different steps of the problem, formulating different algorithmic solutions, in order to later combine them and design an elegant application.

#ComputationalCritique - Identified strengths and weaknesses of the different data-structures and decided which one was the best to use given my scenario.

#ComplexityAnalisis - Carefully analyzed the asymptotic behavior of my solutions both analytically and experimentally, using this result to decide which one was more effective.

#PythonPrograming - Effectively used libraries and APIs, combined with my own solutions and codes to design a python application to my problem.

HCs

#communicationdesign - Designed a code that is easy to use and provided a clear explanation on how to use it and what to do if it does not work, using memes and informal language as a way for the user to better engage with the application.

#selfawareness - On my initial proposal I had another optional item that I was planning to do as well. However, I realized that it would be better for me to focus on doing one great and 2 bad. Trying to do the other application on time would have resulted in 2 bad applications instead of 1 good one.

#organization - Clearly organized my code dividing it into sub-parts and providing good comments and explanations that will help anyone that reads it to understand it better.

In [ ]:

```
1
```