Text Analysis with BSTs

Unfortunately, communities in America have a long history of banning controversial books. For example, this tarball (labs/banned.tgz) contains the full text of several often-banned books which now lie in the public domain.

This still happens, but our online lives have caused our public speech to be both much more plentiful, and much more closely watched. For example, here (labs/badWords.txt) is a list of words and phrases, revealed by a Freedom of Information Act request in 2011, which was used by the Department of Homeland Security to flag suspicious online text. I wonder which of our frequently-banned books most use words and phrases from this list?

The Assignment

We're going to need a way of identifying if a word from our book appears on the list. When we are concerned only about membership tests (whether something is or is not in a list) this is exactly the job of a Set! We can load one up with a list of banned words, and then use `if bookword in badWordList` to test if a word is suspicious. Recall that the two important methods of a set are `insert(key)` and `contains(key)`. In Python, if we want to be able to use the "in" keyword like in the above example, we can implement the reserved function `__contains__(key)`.

We know there are a few different data structures we can use to implement a Set. In this lab, we will try it with both an Array and a Binary Search Tree, to compare their performance.

Step 1: 10 points. Create a file called `bst.py`. In that file, create a class called `ArraySet`. This class should have an `__init__` function which takes no arguments and initializes a class variable `self.array` to an empty array. You should also write a `insert(key)` function which appends a data item (key) to the array, and a `__contains__(self, key)` function which returns True if the specified key is in the array and False if not.

Step 2: 50 points. In `bst.py`, create the following: contains a `Node` class and a `TreeSet` class. The Node class should just contain a constructor, which accepts only the data for the Node. The TreeSet class should have:

- **(5 points)**. A `Node` class, which has an `__init__` function that accepts a data value (like our Node class from linked lists) and sets `self.left` and `self.right` to `None`.
- **(45 points)**. A `TreeSet` class which implements the Set ADT using a Binary Search Tree. This class should have:
    - An `__init__` function which takes no arguments and sets `self.root` to `None`.
    - **(15 points)**. An `insert(self, key)` function which takes a key and inserts it into the correct place in the BST. In class we have been putting numbers into BSTs, but they will work fine with any data types that are comparable. In Python, strings are comparable (you can ask if "abc" < "def") so inserting them into a tree will work exactly the same as it does with numbers. You may find it helpful to create additional functions to accomplish this, feel free to do so.
    - **(15 points)**. A `__contains__(self, key)` function which takes a key and returns True if that key exists in the tree and False if it does not.
    - **(15 points)**. A `printTree(self)` function which prints the entire tree, one key per line, using inorder traversal. This means that, if your tree is working correctly, it should print out all the keys in sorted order.

Step 3: 20 points. Create a program called `oneWord.py` which imports your classes from `bst.py` and fills TreeSet with each line from the `badWords.txt` linked to above. Your program should then take a file name from the command-line arguments, and search for each individual word from the book in the tree of bad words; if it appears,

it should print the word. At the end, it should print the number of words that appeared.

For example:

```
$ python3 oneWord.py huckleberryFinn.txt
shooting
who
who
authorities
(...lots more...)
facility
help
Count: 334
```

Doing this will require that you strip all punctuation from your words from the books, and make them lower-case. Google can probably help you figure out how to do that!

Step 4: 20 points. Use the `time` command in Linux to time the execution of your `oneWord.py` program when run on the text of *Ulysses*. Modify your code so that it uses an `ArraySet` instead of a `TreeSet` and time the execution again. Create a README file and answer the following questions:

- What was the difference in execution times between the two data structures?
- What are the Big-O runtimes of your `insert` and `__contains__` functions for TreeSet and ArraySet?
- Does the difference in execution times match what you would expect from the difference in Big-O runtimes of the algorithms? Explain.

Repeat the same two tests above, but this time using this file (labs/badWordsBig.txt) for your blacklisted words. This is a random sampling of words that I created from a dictionary file, not real blacklisted words. Answer the following questions:

- What was the difference in execution times between the two data structures?
- Does the difference in execution times match what you would expect from the difference in Big-O runtimes of the algorithms? Explain.
- Why are these results different from the first test?

(Optional) Step 5: 20 points extra credit. Create a program called `phrases.py` that puts into a TreeSet the suspicious phrases from this file (labs/badPhrases.txt) and checks a file for those phrases, just like your `oneWord.py` program above. The phrases in this file are no more than five words long. Most of the phrases in this file are specific, modern terms like "central intelligence agency" and "national operations center" which would not appear in the books above. Instead, use this text of the 9/11 report (labs/911Report.txt) to test your program.