Computer Science 112 - Fundamentals of Programming II
Lab Project 9
Due on Github 11:59pm Monday 29 November

Part I: From BST to Dictionary

To get started, we'll create a Dictionary class in a way that we know to be sub-optimal, namely, a BST. So, copy your BST class from a previous assignment. Then, in a file **dictionaries.py**, create a class that uses your **`BST`** class to implement a dictionary's core functionality.

As usual, the smart way to do this is to write the **`BstDictionary`** tests first, then write stubbed methods for **`BstDictionary`**, then finally implement the core functionality that we discussed. I also found it helpful to copy the **`Entry`** class from the slides right away, along with the Dictionary **`__setitem__`** and **`_getitem__`**methods, so I didn't forget those later.

By relying on your **`BST`** class, you should be able to implement each **`BstDictionary`** method in a tiny amount of code, typically just one line per method. The exception was **`__getitem__`**, where I had to write a recursive function.

Here's the test suite I used for this part:

```
print('Part I: BST Dictionary ------------')

d = BstDictionary()

d['a'] = 1
d['b'] = 2
d['c'] = 3

print(d)

print('Should be 2: ' + str(d['b']))
print('Should be None: ' + str(d['e']))

print('Should be True: ' + str('a' in d))
print('Should be True: ' + str('b' in d))
print('Should be False: ' + str('e' in d))

for key in d:   # should show {a: 1, b: 2, c: 3}
    print(key)
```

Part II: Hashing

As we discussed in class, the optimal implementation of a dictionary involves a hash. To get started on this part, simply copy/paste/rename your **`BstDictionary`** class into a new **`HashDictionary`** class in the same **dictionaries.py** file, and copy/paste/modify the test suite into your **`main`**.

Next, of course, you'll remove the BST parts of your **`HashDictionary`** class to make it work instead with the hashing approach from the lecture slides. As before, I found it useful to have some of the

methods just do **`pass`** or **`return False`** to start, and then fill them in with actual code once I figured things out.

As we did in class, it's helpful to start out by copy/pasting from the lecture slides. To do this, I relied initially on the **`HashSet`** code from the lectures slides; however, as in previous assignments, I was able to simplify the code to avoid **`AbstractSet`** and other complications, using an ordinary Python list instead of the Array class used in the slides. Rather than worrying about implementing removal, I focused on the missing part from the slides; namely, entering a new item. To do this, I added a **`next`** field to the **`Entry`** class (default value **`None`**) to make it have the features of both a dictionary entry and a linked-list node. At that point, I realized that much of the remaining **`HashSet`** code was unnecessary for my **`HashDictionary`** implementation, and I ended up with very simple constructor:

```
class HashDictionary:

    def __init__(self, maxsize=1000):

        self.array = [None] * maxsize
```

As always, you want a "just right" test that will be big enough to test the hard part of the code (collisions) but not so big that it's difficult to see whether the code is working. So for this part I tried a small array size (5) and added entries keyed by letters of the alphabet from *a* through *h*.

Part III: Moby Dictionaries

Right below the link where you downloaded this PDF is a link for the full text of *Moby Dick*. To begin this final part of the lab, download and unzip this text into your current folder.

Thanks to the string-processing awesomeness of Python, you can write a single line of code to open, read, and split this text into a list of words. Some googling revealed the following trick to remove punctuation and other non-essentials, once you've done **`import string`** at the top of your code:

```
words = [s.translate(str.maketrans('', '', string.punctuation)) for s in words]
```

Another line of code will give you the vocabulary (set of unique words), using the built-in **`set`** function of course! When I did this I got 115,314 total words with a vocabulary of 14,816.

To finish this lab, you're going to use each of your two dictionary implementations to make a dictionary containing word counts for each word in the novel. First, write a line of code using a list comprehension to make a count of the words. (This was by far the slowest part of the program, taking around 20 seconds on my computer). Next, do your standard timing trick to see how long it takes to enter the counts into the BST dictionary versus the hash dictionary. (*Hint*: Use **`zip`** to pair each vocabulary item with its count in your loop). By adjusting the size of the array in my **`HashDictionary`**, I was able to beat **`BstDictionary`**. Here's what my output looked like for this part:

```
Part III: Moby Dictionaries ------------
Making word counts; will take a while ...

Making BST dictionary
0.23015213012695312 seconds

Making Hash dictionary
0.026352882385253906 seconds
```

When your code is working upload your **dictionaries.py** and **bst.py**. You don't need to upload mobydick.txt.