Reading Reddit Comments-Adithya Shastry

Summary:

This project was designed to implement the Binary Tree data structure. The Binary data structure was used to count the frequency of words used in Reddit comments. The project included two main classes: the BSTree class and the Word Counter Class. The former was used to actually store the data that was being read, and the latter was used to interact with the tree data structure in order to store the correct frequencies. The out put of the program was a text file that held every word used in the comments file and next to it, the number of times the word appeared in the file.

Binary Search Tree:

The Binary Search tree was a generic implementation of a binary tree. This included methods that allowed for interaction with the tree itself. Since the tree itself is organized, it is much quicker to search for values and much quicker to write the code for these methods. Many of these methods utilized recursion, where a method is called in itself. This allows for easy to read the code but can cause some issues if the code is not implemented correctly. One of the very difficult methods that I had implemented is the contains method, which can be seen below:

```
public boolean containsKey(Key key, Comparator<Key> sc) {
    // I will iterate through the tree and compare the values using an in
    // order method
    int i = sc.compare(key,this.getPair().getKey());
    if (i < 0) {
        if (this.left == null) {
            return false;
        }
        return this.left.containsKey(key, sc);
    } else if (i == 0) {
        System.out.println("reached here");
        return true;
    } else {
        if (this.right == null) {
            return false;

        }
        return this.right.containsKey(key, sc);
    }

}
```

As seen above, it this method utilizes the organization of the data in order to make the search quicker and it also uses recursion to get to deeper and deeper treeNodes in the actual tree.

Word Counter Class:

The Word counter class processed the reddit text files that were passed into it. This class was relatively easy to implement, but for my extension for this project I wanted to make the code as efficient as possible therefore more about the word counter class will be explained below.

Extensions:

WordCounter Class:

The Word Counter Class was changed to reduce the number of times the tree had to be searched. In order to do this, I created a method that would return the key Value pair instead of using the contains a method that most of my peers use. This allowed me to reduce my search of the tree to at a maximum twice, this would mean that in big-oh notation the function would take (n log n)(n log n) in terms of the order. This is a good goal to have when it comes to programming algorithms or processes because the amount of time the method will take could have been much worse. For example, most of my peers seemed to use the contains method and then the get method, and the put method in a sequence. This means the order with respect to n is (n log n)(n log n)(n log n). This might be even worse if they did not program these methods correctly. For example, like not using the organizational features of the Binary Search tree in order to make things faster. My code can be seen below:

```
public void loadFromOriginalWordsFile(String filename) throws IOException {
    try {
        BufferedReader reader = new BufferedReader(new FileReader(filename));
        String line = reader.readLine();
        String[] parse;
        while ((line != null)) {
            parse = line.split("[^a-zA-Z0-9']+");
            for (int i = 0; i < parse.length; i++) {
                String word = parse[i].trim().toLowerCase();
                if (word.equals("") == false) {
                    this.totalWordCount++;
                    KeyValuePair<String, Integer> kv = this.tree.getKeyValue(word);
                    if (kv == null) {
                        this.tree.put(word, 1);
                        this.uniqueWordCount++;
                    } else {
                        kv.setValue(kv.getValue() + 1);// this should
                                                       // iterate the
                                                       // counter

                    }
                }
            }
            line = reader.readLine();

        }
        reader.close();

    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

This is the code for the actual method in the BSTree class:

```
public KeyValuePair<Key, Value> getKeyValue(Key key, Comparator<Key> comp) {
    int i = comp.compare(key, this.getPair().getKey());
    if (i < 0) {
        if (this.left == null) {
            return null;
        }
        return this.left.getKeyValue(key, comp);
    } else if (i == 0) {
        return this.getPair();
    } else {
        if (this.right == null) {
            return null;

        }
        return this.right.getKeyValue(key, comp);
    }
}
```

By implementing this method I was able to bring down the processing time to about 20 seconds, a time that is much lower than the amount of time it took my peers' programs to process the same data. This is a part of computer science that really interests me, which is why I ended taking the 4-day extension. I think I have made this code as efficient as possible, in fact, this is what Dr.Skrien and Dr.Codabux said when they looked at my code. I am really excited to go on to the next data structure because according to Dr.Skrien, Hash Tables are the only way to make this code more efficient.

Only printing the Words with data over 10000:

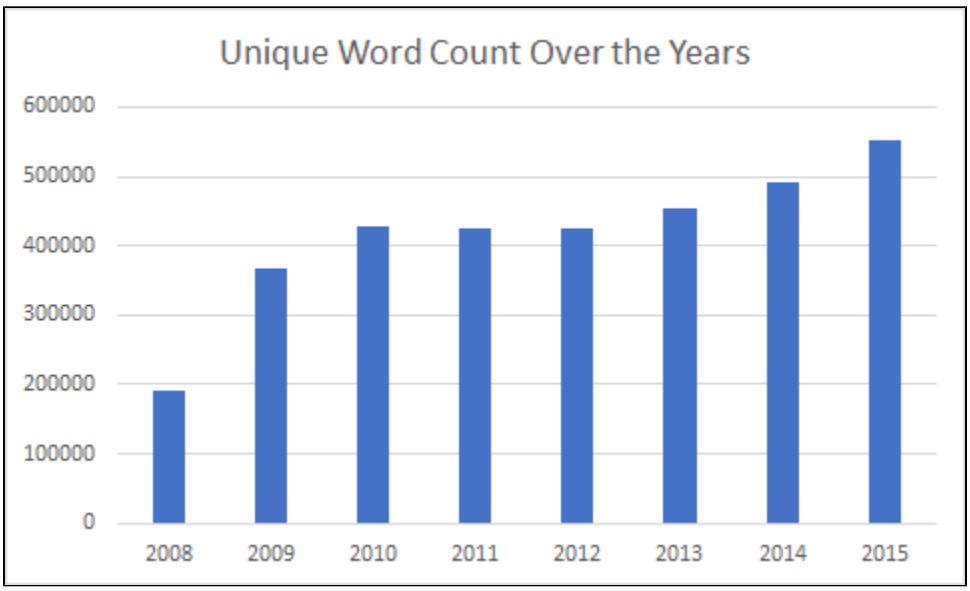
This was a very easy extension to implement, but at the same time very useful. I found that many of the words had less than 100 uses, but were still being printed on to the page because the code would print all of them. I saw this as a lot of unnecessary information, therefore I limited the output to only words with over 10000 uses. This code can be seen below:

```
public void writeWordCountFile(String filename) {
    BufferedWriter bw = null;
    FileWriter fw = null;

    try {

        fw = new FileWriter(filename);
        bw = new BufferedWriter(fw);
        bw.write("Total Count is " + this.totalWordCount + "\n");
        bw.write("The Total Unique Word Count is " + this.uniqueWordCount + "\n");
        for (KeyValuePair<String, Integer> kv : this.tree.getPairs()) {
            if (kv.getValue() >= 10000) {// {Print only if the value is
                                         // above
                // 10000
                bw.write(kv.getKey() + " " + kv.getValue() + "\n");
            }
        }
        bw.close();

        System.out.println("Done");

    } catch (IOException e) {

        e.printStackTrace();

    }
}
```

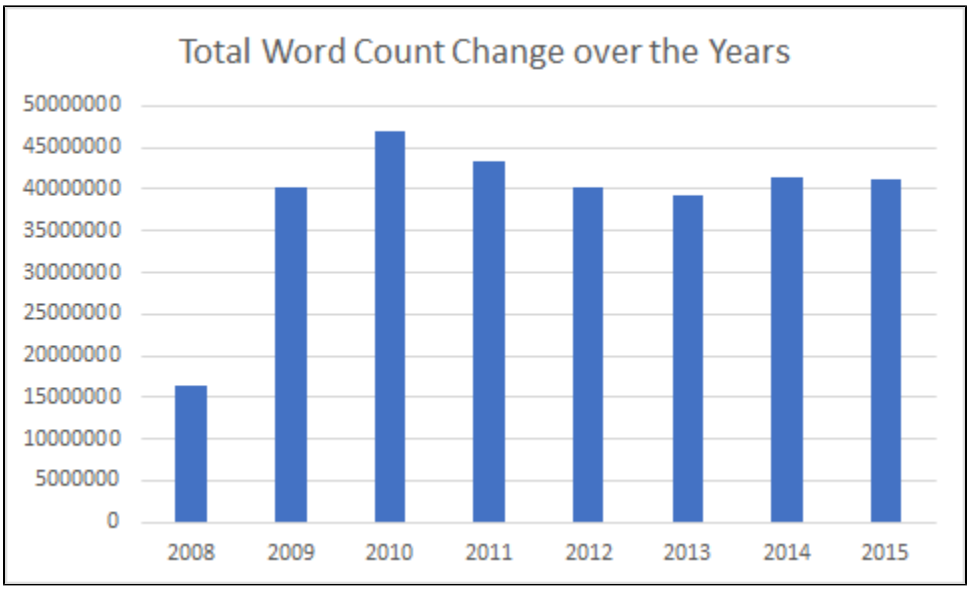
Results:

These are the results of the project:

Unique Word Count over the years:



Total Word Count over the years:



The Specific data is given in the txt files named result followed by the year.

Acknowledgements:

Thank you to Dr.Skrien, Dr.Codabux, Dr.Eaton, and all the TA's that helped me debug this code and helped me make it more efficient!