Project 7: Trees versus Tables: The Countdown--Adithya Shastry

Summary:

This project uses a Hashtable and a Binary Search tree to find how many times the users of Reddit used certain words over the years. The computer science objective for this lab is to see that the Hashtable data structure is a much faster data structure when it comes to storing and accessing data. This project uses two main classes, the class for the data structure itself and a common class called the Wordcounter class. The wordCounter class was used to actually count the frequencies of the words and interact with the data structure class. The results of the project was a graph labeling the amount of time the computer took to process the data and files of every word used during a specific year and the number of times it was used next to it.

HashTable:

The code for the hash table implemented the MyMap interface and therefore had the same methods that the BSTree class had the last project.The put method, however, was definitely different when it came to how it was implemented. In order to deal with collisions, the hash table I implemented used open addressing. In open addressing, key-value pairs are added to an ArrayList at the index of the original ArrayList. This means that there will be multiple items at every index. The problem with this method is that as the number of collisions increase, the number of comparisons also increase. This will make the process very slow if there are a lot of collisions. I will attempt to reduce the number of collisions as one of my extensions.

```
@Override
//This is using open addressing to deal with collisions
public void put(K new_key, V new_value) {
    int hash = new_key.hashCode();//Generate hashcode
    int index = Math.abs(hash) % this.array.size();//Get the arraylist of keyvalue pairs from the index
    ArrayList<KeyValuePair<K,V>> sublist = this.array.get(index);//
    for (KeyValuePair<K,V> pair: sublist) {
        if (comp.compare(new_key, pair.getKey()) ==0) {//Will check to see if the key is already in the hash table
            pair.setValue( new_value );//Will set the new value
            return;
        }
    }
    sublist.add( new KeyValuePair<K,V>( new_key, new_value ) );//Otherwise it will create a new key value pair
    //We want to iterate the size counter here since we want it to only itereate when something new is created
    this.size++;

}
//The actual pointers new_key and New_value are garbage collected along with the actual method by being taken out of the stack because they are no longer in use any more
```

Word Counter:

The same Word Counter class that was used in the previous project was used in this project. There was one difference however, that was the main method of the class. Since in this lab all of the files needed to be run 5 times in order to find the averages needed for the graph, I decided to do this all in the java program, making it easier for me. This can be seen below:

```
String[] files = { "reddit_comments_2008.txt", "reddit_comments_2009.txt", "reddit_comments_2010.txt",
        "reddit_comments_2011.txt", "reddit_comments_2012.txt", "reddit_comments_2013.txt",
        "reddit_comments_2014.txt", "reddit_comments_2015.txt" };
String[] Ofiles = { "results2008", "results2009", "results2010", "results2011", "results2012", "results2013",
        "results2014", "results2015" };
for (int i = 0; i < files.length; i++) {
    System.out.println("This is for the file: "+files[i]);
    int x = 1;
    ArrayList<Double>times=new ArrayList<Double>();
    while (x <= 5) {
        WordCounter wc = new WordCounter();
        //Start the counter
        long startTime=System.currentTimeMillis();
        wc.loadFromOriginalWordsFile(files[i]);
        //Stop the counter, we just want to know the times for storing the data in the data structure
        long finalTime=System.currentTimeMillis();
        //Calculate the change in time in seconds
        double deltaTime=(finalTime-startTime)/1000.00;
        times.add(deltaTime);//This will add it to the list
        wc.writeWordCountFile(Ofiles[i] + ".txt");
        x++;
    }
    //Now drop the highest and lowest using 1 algorithm
    int lowIndex=0;
    int highIndex=0;
    for(int a=0;a<times.size();a++){

            //Find the minimum
            if(times.get(a)<=times.get(lowIndex)){
                //we found a new min!
                lowIndex=a;
            }
            //Now check the maximum
            if(times.get(a)>=times.get(highIndex)){
                highIndex=a;
            }
        }
    //Now we need to remove these values from the list
    times.remove(highIndex);
    if(highIndex<lowIndex){
        times.remove(lowIndex-1);
    }
```

Extensions:

Reducing Collisions:

I tried to see if the number of collisions, and therefore the amount of time the code took to process the information would be decreased if the keys were used twice instead of just once. This would make the hashed values much more unique. This can be seen below:

```
@Override
//This is using open addressing to deal with collisions
public void put(K new_key, V new_value) {
    Integer hash = new_key.hashCode();//Generate hashcode
    int hash2=hash.hashCode();
    int index = Math.abs(hash2) % this.array.size();//Get the arraylist of keyvalue pairs from the index
    ArrayList<KeyValuePair<K,V>> sublist = this.array.get(index);//
    for (KeyValuePair<K,V> pair: sublist) {
        if (comp.compare(new_key, pair.getKey()) ==0) {//Will check to see if the key is already in the hash table
            pair.setValue( new_value );//Will set the new value
            return;
        }
    }
    sublist.add( new KeyValuePair<K,V>( new_key, new_value ) );//Otherwise it will create a new key value pair
    //We want to iterate the size counter here since we want it to only itereate when something new is created
    this.size++;

}
//The actual pointers new_key and New_value are garbage collected along with the actual method by being taken out of the stack because they are no longer in use any more
```

The results can be seen in the results section. The dots for this extension is the 2xHashtable dots.

Changing attributes of the hash table to see if it will change the processing time:

In order to do these, I changed the size of the original ArrayList being used to store the inner arraylists of key-value pairs, to see if a bigger value would change the amount of time taken to process. The default value was 1000, but I tried 450 and 100 to see if there would be any noticeable change. This can be seen in the data labeled Hash(size=100),Hash(size=10),and Hash(size=45). The results can be seen in the results section.

Attempt to decrease the processing time by creating a new method:

I used a similar getKeys method like I did in the previous project to reduce the number of searches the hashtable must do in order to accomplish the same task. This can be seen below:

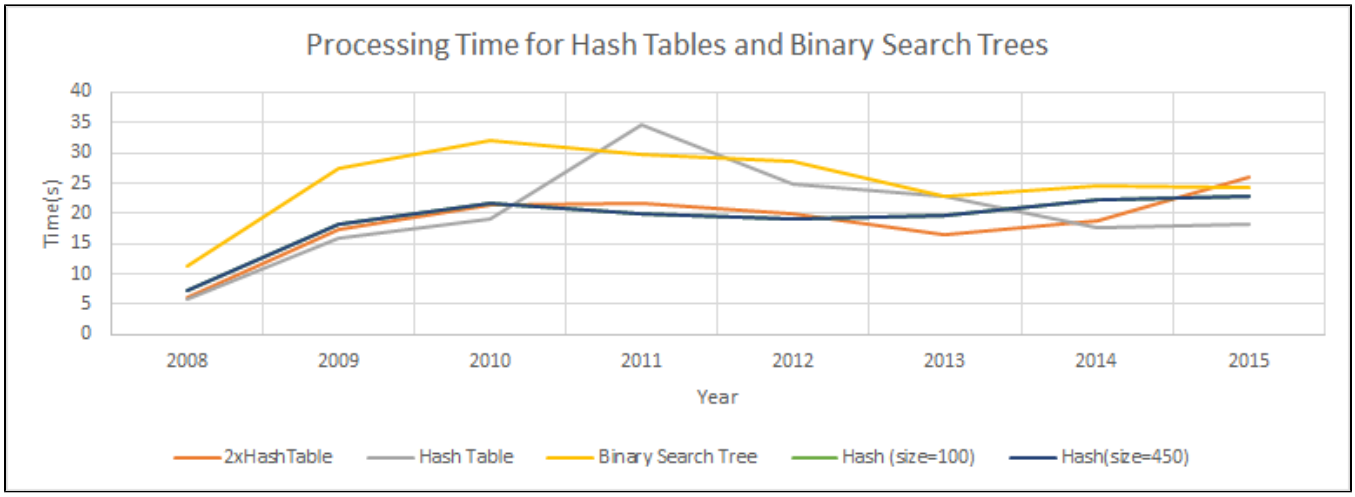
```
//I only changed this method since this is the only one that really interacted with the WordCounter class
    public KeyValuePair<K,V> getKeyValue(K Key) {
        //this will return a key value pair
        Integer hash=Key.hashCode();//This will generate a hash code
        int hash2=hash.hashCode();
        int index=Math.abs(hash2)%this.array.size();//this will get the index
        //now I will check to see if the arraylist within this index has the value I am looking for
        for(KeyValuePair<K,V> pair:this.array.get(index)){
            //now I will use the comparator to compare the values
            if(this.comp.compare(pair.getKey(), Key)==0){
                return pair;
            }
        }
        return null;
    }
    }
```

Tracking memory loss:

This can be seen in the actual code as comments.

Results:

These are the results of the project. It is clear that the attempts to reduce the number of collisions was very effective! For example, hashing the values twice actually sped up the hash table by quite a margin, similar to changing the size of the ArrayList(the Hashtable). We can see that the size of the table and the time the program takes to process the data are inversely related.There seems to also be a trend in the time it took for the computer to process the specific files. When the sizes of each of the respective files are consulted, it is clear that some of the years have much bigger sizes and therefore take longer to process. For example, 2011 takes exceptionally longer when compared to all other years for all trials because 2011 is a much bigger file, with more words for my program to process. It is also interesting to notice that 2010 has a similar file size, but takes a little less amount of time to program. This may have to do with the number of unique words in each of these lists. For example, 2015 may have more unique words and because processing unique words are the slowest part of the program, it affects the time it takes for the program to process the information.



Conclusion:

Overall, this project was a great way to look at the efficiency of my code and see where I can improve it. I now take this aspect of efficiency into account whenever I write anything.