David Herrington

MSCI 240 Fall 2017, Instructor: Dr. Mark Hancock

December 4th, 2017

Project 3

Introduction:

This project focusses on three different dictionaries, comparing them to each other to assess performance and usefulness. A collection of books is scanned in and searched for unique words, recording the amount of unique words and how many times they appear. The three dictionaries are implemented using ArrayLists, a binary search using a TreeMap and a hash table using a HashMap.

Dictionary Implementations:

ArrayList Implementation:

a) Worst Case Growth

The code begins with a for loop that runs the entire given array of words. Nested within this for loop is another for loop. This nested for loop is based off the size of the newly made array, building it to compare to the given array. If the word already exists, it adds to the count of that word and exits early. If it is unique, it runs the entire for loop, exits, and the word is added by the following if statement. The worst possible runtime would be that every word is unique, and it runs the entire for loop every time. This would result in a growth rate of **T(n) = n^2**, meaning it is quadratic. This is shown to be extremely long for the amount words assessed.

b) Time to Count

I ran different amounts of words, this was done by deleting certain amounts of books from the source folder, to prove my growth rate.

Number of Words Read	Elapsed Time in Seconds
30419	0.2849313710000004
413146	17.961029409000002
757204	62.234894691
1532048	369.4800857553
4403310	3856.981687148

As we can see between lines 2 and 3, The number of words read increases by about a factor of 2 while the elapsed time increases by about 4. This is proof of **T(n) = n^2**

c) Unique Frequencies

This information was the same for all 3 dictionaries and thus will only be reported here once. This is the count when all books were read in

```
Finished reading 4403310 words.

ARRAYLIST IMPLEMENTATION
To count the events with unsorted lists took 3856.981687148seconds.
Found 52104 unique words.
The 20 most frequent words and their count are:
the     229810
and     146814
of      123259
to      114765
a       81318
i       72480
in      70337
that    55380
it      51120
he      49476
you     42303
was     40212
his     39831
with    36859
for     34284
as      32756
is      32755
not     32543
but     30407
s       29139
```

TreeMap Implementation:

a) Worst Case Growth

The TreeMap Implementation runs through the words by building a new TreeMap. The TreeMap is searched with .containsKey and if the word already exists the count is added to, while the word remains the same. If the TreeMap does not already contain the word it them adds it, and tracks the amount of new words it has added with a counter. Because of the way the TreeMap uses the get and put functions, it becomes log(n). (The Java API shows this) This is nested in a for loop, and thus the growth rate is **T(n) = nlog(n)**, or linearithmic.

b) Time to Count

Number of Words Read	Elapsed Time in Seconds
30419	0.0299909193
413146	0.24458516900000002
757204	0.36834421100000003
1532048	0.718418522
4403310	3.009173531

This is a little harder to see but it is obvious it is not linear. Comparing between lines 2 and 5 we get about a factor of 10, nlog(n) gives us about 11, and the factor for elapsed time is about 12. This is close enough for our standards and thus proven.

c) Unique Frequencies

This was shown in the ArrayList implementation

HashMap Implementation:

a) Worst Case Growth

The code for the HashMap dictionary is basically the same as the TreeMap dictionary. However, unlike the TreeMap, the HashMap uses the .get and .put functions differently and it is simply constant. This results in only the single for loop being n, and thus the worst-case growth being T(n) = n. The worst-case growth is linear.

b) Time to Count

Number of Words Read	Elapsed Time in Seconds
30419	0.010743377
413146	0.103446441
757204	0.14934640400000002
1532048	0.199635476
4403310	0.89489856900000001

As we can see between lines 1 and 2, number of words read in roughly increases by a factor of 10, and so does elapsed time. This shows the growth is in fact linear.

c) Unique Frequencies

This was shown in the ArrayList Implementation

Performance:

It is obvious that the ArrayList implementation is by far the slowest. It is absolutely ridiculous to compare and run massive amounts of words the way I did and you can see that it took more than 3000 seconds. Because of this, it is not even considered to be a contender in this situation. The TreeMap and HashMap implementations are very similar and with small numbers seem to have about the same run time. However, if we get into huge n numbers, the HashMap starts to become much more efficient than the TreeMap. Because of this, I would choose the HashMap to be the best implementation of the three.

Top 20 words method:

The method I used was to build a PriorityQueue. This queue gives values a priority based on heap specified, which in this case is the comparison of the amount of times a word shows up, or better known as the value count (In the WordCount class). Once the Queue is made, I used a for loop to print the first 20 values of it. This method proved easy and efficient.

Implementation Issues:

I did not come across any specific issues, however, the ArrayList Implementation took 3856 seconds to run through the 4 million words. That is 64 minutes, I didn’t even believe it was working right until I ran a counter (that I later deleted) and was in disbelief that it would take this long.

Acknowledgements:

Java API
Tutorials Point: https://www.tutorialspoint.com/java/java_hashtable_class.htm

Help Given:

none

Program Class:

```
package Project2;
import java.io.*;
import java.util.*;


package Project3;

import java.io.File;
import java.io.FilenameFilter;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.PriorityQueue;
import java.util.TreeMap;

public class Program {

    public static void main(String[] args) {
        try {
            // TODO: Copy and paste your directory here. Your directory should
            // ONLY contain the books!
            String directoryContainingBooks = "C:/Users/David/Documents/bruh/Tut6/p3-
books";

            File dirInfo = new File(directoryContainingBooks);
            if (!dirInfo.exists()) {
                throw new Exception("Directory " + directoryContainingBooks + " does
not exist.");
            }

            // We are going to read all of the words in and store them in a list
            // to make the rest of your project easier for you. This is NOT the
            // way one would normally operate. There is no reason to store
            // all of the data in RAM. We only need to read words from the file,
            // count them, and then move on to the rest of the file.
            ArrayList<String> words = new ArrayList<>();
            words.ensureCapacity(4500000); // pre-allocate needed space
            FilenameFilter filter = new FilenameFilter() {
                @Override
                public boolean accept(File dir, String name) {
                    return name.toLowerCase().endsWith(".txt");
                }
            };

            for (File file : dirInfo.listFiles(filter)) {
                byte[] bytes = Files.readAllBytes(file.toPath());
                tokenize(new String(bytes, StandardCharsets.UTF_8), words);
            }

            System.out.println("Finished reading " + words.size() + " words.");
            // *********************** IMPORTANT
            // **************************************
            //
            // NOTE: You may NOT change the words list! Do NOT change the words
            // list!
            // You are to only read the words from index 0 to index
            // words.Count-1.
            // Do NOT sort the list of words or change it in any way.
            //
            //
************************************************************************

            // ----------------------------------------------------------
            //
            // TODO: Add your code here to count the words, etc. You
            // will do best to create many methods to organize
            // your work and experiments.
            //

            ArrayList<WordCount> wordCounts = new ArrayList<WordCount>();

            Stopwatch stopwatch = new Stopwatch();
            stopwatch.start();
            int numUniqwords = 0;


            for(int i = 0; i < words.size(); i++){

             String currentword = words.get(i);
             boolean exists = false;


             for(int j = 0; j < wordCounts.size(); j++){

                         if(wordCounts.get(j).getWord().equals(currentword)){
                               wordCounts.get(j).addCount();
                               exists = true;
                         }
             }

             if(exists==false){
                   wordCounts.add(new WordCount(currentword, 1));
                   numUniqwords++;
                   exists=true;

             }

            }

            PriorityQueue<WordCount> ordered = new PriorityQueue<WordCount>();

            for(int i = 0; i<wordCounts.size(); i++){
             if(wordCounts.get(i) != null){
                   ordered.offer(wordCounts.get(i));
```

```
                 }
                }

                System.out.println();
                System.out.println("ARRAYLIST IMPLEMENTATION");
                System.out.println("To count the words with unsorted lists took " +
stopwatch.getElapsedSeconds() + "seconds.");
                System.out.println("Found " + numUniqwords + " unique words.");
                System.out.println("The 20 most frequent words and their counts are:");

                int limit = Math.min(20, ordered.size());
                for(int i = 0; i < limit; i++){
                 WordCount object = ordered.poll();
                 String word = object.getWord();
                 int freq = object.getCount();
                 System.out.printf(word+ "\t" + freq + "\n");
                }
                System.out.println();
                //
                //
                //
                // Binary search tree
                //
                //
                //
                TreeMap<String, Integer> Treewcounts = new TreeMap<String,Integer>();


                Stopwatch stopwatch2 = new Stopwatch();
                stopwatch2.start();
                int numTreewords = 0;
                for(int i = 0; i < words.size(); i++){

                 String currentword = words.get(i);

                 if(Treewcounts.containsKey(currentword)){
                        Treewcounts.replace(currentword, 1+Treewcounts.get(currentword));

                 }
                 else{
                        Treewcounts.put(currentword, 1);
                        numTreewords++;
                 }
               }

               stopwatch2.stop();


                PriorityQueue<WordCount> ordered2 = new PriorityQueue<WordCount>();

                for(String key: Treewcounts.keySet()){
                ordered2.add(new WordCount(key, Treewcounts.get(key)));
                }

                System.out.println("TREEMAP IMPLEMENTATION");
```

```
            System.out.println("To count the words with a treemap took " + 
stopwatch2.getElapsedSeconds() + "seconds.");
            System.out.println("Found " + numTreewords + " unique words.");
            System.out.println("The 20 most frequent words and their counts are:");
             int limit2 = Math.min(20, ordered2.size());
             for(int i1 = 0; i1 < limit2; i1++){
             WordCount object = ordered2.poll();
             String word = object.getWord();
             int freq = object.getCount();
             System.out.printf(word+ "\t" + freq + "\n");
             }

             System.out.println();




             //
             //
             //
             // Hashmap
             //
             //
             //

         HashMap<String, Integer> Hashwcounts = new HashMap<String,Integer>();


         Stopwatch stopwatch3 = new Stopwatch();
         stopwatch3.start();
         int numHashwords = 0;
         for(int i = 0; i < words.size(); i++){

               String currentword = words.get(i);

               if(Hashwcounts.containsKey(currentword)){
                     Hashwcounts.replace(currentword, 1 + 
Hashwcounts.get(currentword));

               }
               else{
                     Hashwcounts.put(currentword, 1);
                     numHashwords++;
               }
        }

        stopwatch2.stop();


         PriorityQueue<WordCount> ordered3 = new PriorityQueue<WordCount>();

         for(String key: Hashwcounts.keySet()){
         ordered3.add(new WordCount(key, Hashwcounts.get(key)));
         }
         System.out.println("HASHMAP IMPLEMENTATION");
         System.out.println("To count the events with a hashtable took " +
stopwatch3.getElapsedSeconds() + "seconds.");
         System.out.println("Found " + numHashwords + " unique words.");
         System.out.println("The 20 most frequent words and their counts are:");

         int limit3 = Math.min(20, ordered3.size());
         for(int i1 = 0; i1 < limit3; i1++){
              WordCount object = ordered3.poll();
              String word = object.getWord();
              int freq = object.getCount();
              System.out.printf(word + "\t" + freq + "\n");
         }

          System.out.println();


     }








            // ------------------------------------------------------------
          catch (Exception ex) {
            System.err.println("Caught unhandled exception: " + ex.getMessage());
            ex.printStackTrace();
         }
     }

     // This function takes a string and breaks it up into "words" and
     // adds them to the given list.
     //
     // Based on SimpleTokenizer by Trevor Strohman,
     // http://www.lemurproject.org/galago.php
     static public void tokenize(String text, ArrayList<String> words) {
         text = text.toLowerCase();

         int start = 0;

         int i;
         for (i = 0; i < text.length(); ++i) {
            char c = text.charAt(i);
            if (!Character.isLetter(c)) {
                if (start != i) {
                    String token = text.substring(start, i);
                    words.add(token);
                }
                start = i + 1;
            }
         }
```

```
            if (start != i) {
                words.add(text.substring(start, i));
            }
        }
    }
```

WordCount Class:

```
public class WordCount implements Comparable<WordCount> {

    public String word;
    public int count;

    public WordCount(String word){
        this.word = word;
        this.count = 0;
        }

    public WordCount(String word, int count){
        this.word = word;
        this.count = count;
    }

    public void addCount(){
        count++;
    }

    public String getWord(){
        return word;
    }

    public int getCount(){
        return count;
    }

    public int compareTo(WordCount other){
      return Integer.compare(other.count, this.count);
    }
}
```

Acknowledgment of Receiving Assistance or Use of Others' Ideas

----

I received the following help, assistance, or any ideas from

classmates, other knowledgeable people, books or non-course

websites (please include a description of discussions with

the TA or the instructor):

Record of Giving Assistance to Others

I gave the following help, assistance, or ideas to the following

classmates (please describe what assistance to whom was given

by you):

Declaration

I declare that except for the assistance noted above, assistance

provided on the course website, and material provided by the

instructor and/or TAs that this is my original work.

I have neither given nor received an electronic or printed version

of any part of this code to/from anyone.

I declare that any program output submitted as part of the

assignment was generated by the program code submitted and not

altered in any way.

SIGNATURE: