Abstract

In this project, we extend our word counting implementation from a Binary Search Tree (BST) to a hashmap. We use the same MapSet interface to implement a hashmap which uses a hash function and chaining to handle collisions. From here, we were able to compare the run times for both a BST and a hashmap and observed that a hashmap has a significantly better run time than a BST when it comes to word counting and building a word map. The only trend we were able to observe was that tree depth and the number of collisions in a hashmap seemed to have no significant impact on the run time. Finally, we used a website to generate a more diverse data set which we then used to determine the time complexity of a BST and hashmap experimentally. We were able to confirm that our BST had a time complexity of O(n) and that our hashmap had an approximate time complicity of O(1).

Results

This week's project is basically a continuation of last week's project. The only difference is the data structure that we are using, which is known as a hashmap! The basic implementation of a hashmap is that we have an array that holds our key-value pairs. The location of these pairs is determined by the key through a hash function. A hash function is a function that takes the key and converts it into an integer index. The most ideal hash function is one that is one-to-one and onto, this basically means that there is a unique value for every key, and every value is a proper index of our array. The other cool thing about a hash function is that is a one-way function, this basically means that if we only have an output from the hash function, we cannot get the input even if we know the algorithm.

Since this project is extremely similar to last week's project, I am only going to talk about the new classes that we implemented for this week on top of some modifications made to classes from last week's project. As stated earlier, we are using a new data structure, which means that we must create a new class to implement this data structure.

This is where the Hashmap class comes into play. This class implements the MapSet interface, so that means that the Hashmapp class is going to have the same methods as the BSTMap class from last week. As stated earlier, hashmaps use an array to hold key-value pair data. Since our arrays can't have an unlimited size, we know that our hash function wont be one-to-one which means that there is a possiblity that two different keys will be put in the same index. This is known as a collision and there really two different ways to handle this; the first way is to just put the other key in a different slot in the array. This way is great if you just want to implement a hashmap quickly, but the problem with this implementation is that you risk having to look through the entire array to get to an empty slot. The other issue is that the array will get filled up faster which means that we would need to extend our array more times. The second way to handle a collision is to use something called chaining. This is basically like using a linkedlist within the array, with each node containing the keys that would be at some calculated index. This collision handling method is the same one that Java uses, and it surprisingly has a constant time complexity.

For this project, I decided to use the chaining approach since I thought that it would be a pretty elegant implementation of a hashmap. The chaining approach requires me to have a type of linked list data structure, and instead of copying the linked list from a previous project, I decided to make

an inner class within the Hashmap class specifically made for a hashmap. This allowed me to use the expert pattern for the hashmap, which made implementing the methods really elegant. The inner class that I made is the MapNode class; this class implements a node data structure and it holds a key-value pair. There are also fields within the class which holds the next node and a comparator object. The MapNode class is basically identical to the TNode class from last week's project, the only difference is that we are not dealing with left and right subtrees.

The other different thing about the Hashmap class is that there is a method that handles how the data array is extended, a method that implements a hash function, and a method that returns the number of collisions. The first method I want to talk about is the makeHashcode method. This method takes in a key and then uses the inherited hashCode() method to get the key's hash code. According to the Java API, the hashCode method uses a hashing algorithm to create an integer value; an important thing to note is that the only way two objects can have an identical hash code is if they are equal under the equals method. There is an issue with this method and that is that it is possible for the hashCode method to return an integer that is out of bounds. The best way around this is to mod the integer value with the size of the array in order to guarantee that the index will never be out of bounds. This does however have the ability to increase the number of collisions even if two objects are not equal under the equals method. The next methods I want to talk about are the getNumberCollisions and the extend methods. These two methods are pretty straightforward, and the reason I wanted to talk about these is to point put the difference between the put method in the Hashmap class and the put method in the BSTMap class. In the Hashmap class, the put method first looks if the capacity of the data array is under 20% to see if the data array should be extended or not. The rest of the put class is pretty identical except for the part where the put method has to deal with collisions. Since I decided to use chaining, the put method just needs to add the word to the linked list in that array which is something that the TNode class handles.

The next two classes that I want to talk about are two classes which are pretty similar to each other in terms of function. First the WordCounter2 class, which is basically like the WordCounter class from last week, but it's modified to handle both a BST and a Hashmap. I also added a method which returns the data structure object that is being used, this was mostly done to gather data specific to the data structure which we are using. The next class I want to talk about is the TextStatistics class! This class is a continuation of my extension from last week, except that it is more tailored to specific data structures. The first important method in this class is the timeComplexity method, this method takes in a string containing the data structure as an input. The point of this method is to calculate the average time spent building a word map for all the reddit comments that we have. Like last week, I designed the method so that it looks through a directory for all the reddit comments text files. I like to put these files into a directory named Data, so that's the directory that the method looks through. This method also uses the average and timeData auxiliary methods to gather data like the average amount of time spent building a map, the file size, the year, the total number of words, and the number of unique words.

The next two methods are identical so I am not going to talk about them individually, these methods are the bstStatistics and the hashmapStatistics methods. They work like the timeComplexity method except that it doesn't average the run time, and it they write a csv file instead of outputting the results on the command line. Both methods return the year, total number of words, the number of unique words, the file size, and the run time, but bstStatistics also returns the tree's depth and hashmapStatistics returns the number of collisions.

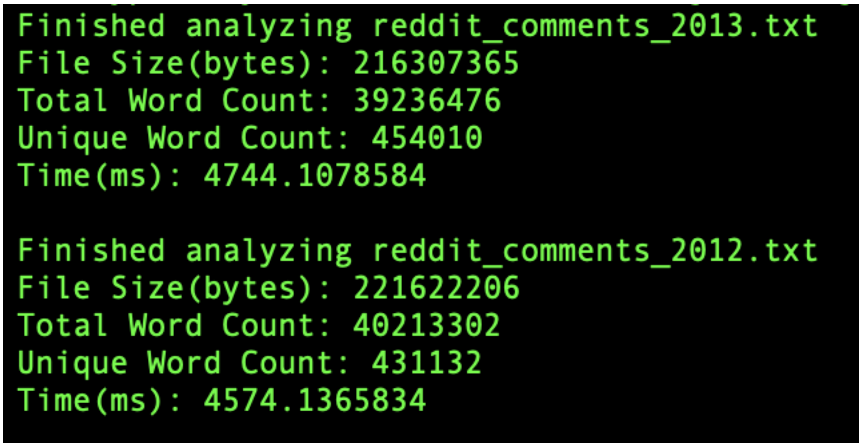


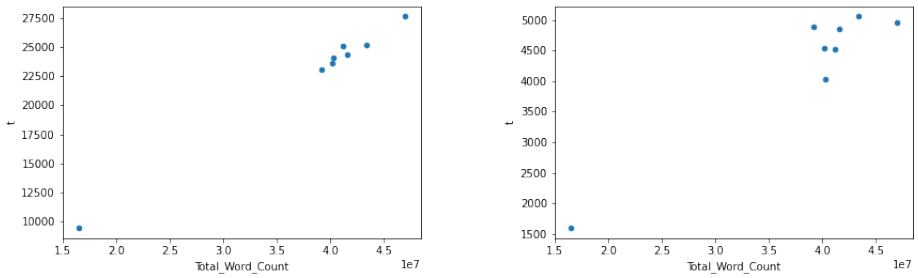
Figure 1: Output from timeComplexity method

The results from the TextStatistics class was actually pretty surprising. We already went through this process for the BST last week, so I am only going to talk about the hashmap and compare it to the BST. First, the hashmap is extremely fast when it comes to word counting and traversal. In comparison with the BST, the hashmap spent anywhere between 4 to 5 seconds building a word map whereas the BST took about 20 to 25 seconds. In terms of trends, it is pretty hard to tell if the hashmap is following a different trend than the BST or if it's a different trend. When comparing the run times to the year, total number of words, and number of unique words, the plots for both the BST and hashmap are the same as seen in Figure.2

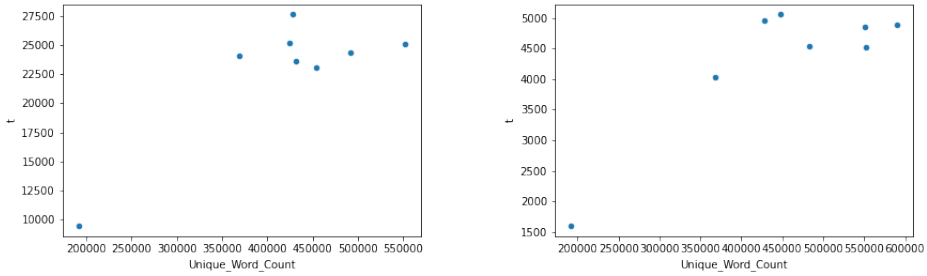
The trends in Figure.2 (b) and (c) are pretty identical, and there's really not much we can say due to the lack of data. We can say that the BST is significantly slower than the hashmap, but that is about it. In Figure.2 (a), we can see that there is a slight difference between the trends for the BST and hashmap. The BST looks like it has a linear relationship, whereas the hashmap seems to have the run time hover around a certain point. Again, we can't say much since we don't really have that much data to work with.

The next thing we can look at is how the number of collisions impacts the run time for a hashmap, and how the depth impacts the run time for a BST. I don't really think we can say much from those plots as well since our data set isn't diverse at all. We can expect the collisions and depths to be different for each file. I expect that there shouldn't be a relationship between the number of collisions, depths, and run times since these are purely impacted by the number of unique words. A very important thing to note is that the number of unique words are extremely small compared to the number of total words, that means that the time complexity is going to be completely dominated by the put method regardless of the depth and the number of collisions.

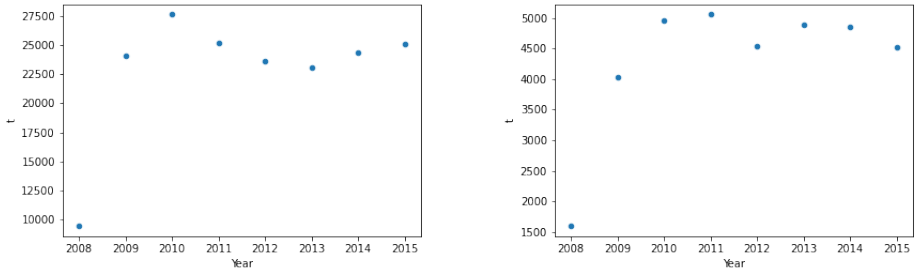
As seen on Figure.3 we seem to be getting a horizontal line, which is indicative that the two variables that we are plotting have no relationship whatsoever. This is exactly what we predicted before, and it makes sense. First for BST, the depth is going to be determined strictly by the number of unique words, which is much smaller than the total number of words. This means that the put method is going to be completely dominated by the number of words as opposed to being dominated by the number of levels that it needs to traverse. For the hashmap, the reason has to do with collision handling, since we are using chaining to handle the collisions, we don't have to spend more time looking for an empty slot and even more time extending the array, since it won't



(a) Total word count vs run time

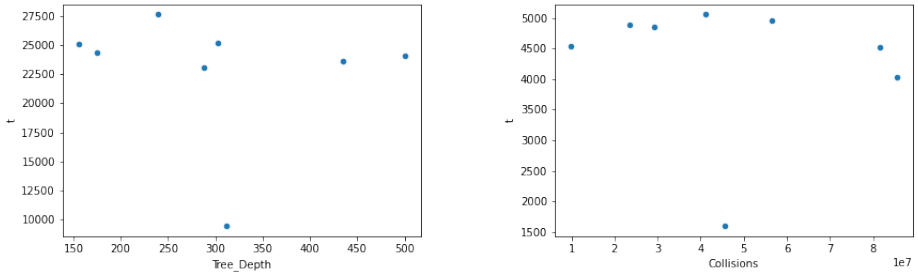


(b) Unique word count vs run time



(c) Year vs run time

Figure 2: Plots for both the bst (left) and hashmap (right) data structures



(a) Depth vs run time (b) Collisions vs run time

Figure 3: Plots showing how collisions and depth impacts the run time for the hashmap and BST data structures

be filled up frequently. One thing we can verify without significantly chaning the Hashmap class is to see how much of an impact extending our data array has on our run time. If we increase the threshold remaining capacity from 20% to 70%, we should expect to see an increase in run time. When we do this, we do see an increase in run time, but by 100 milliseconds, which means that extending our data array doesn't really have that much of an impact if we do it more frequently.

Overall, we can indeed conclude that the BST is significantly slower than the hashmap when it comes to word counting and building word maps. This makes sense since the an unbalanced BST has a time complexity of O(n) whereas the hashmap has an approximate time complexity of O(1).

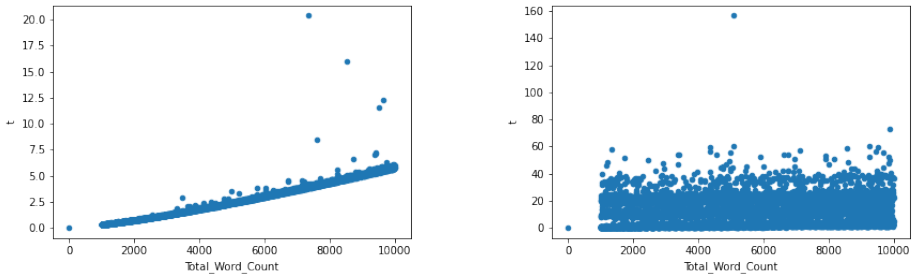
Extension

For this week's extension, I decided to try to gather a much more diverse data set in order to get a nice continuous time complexity curve. A very important thing to note is that I only wanted to focus on time complexity and not spatial complexity, this is because I couldn't think of a way to gather data about spatial complexity into a csv without having to write it down by hand.

The main problem in gathering a more diverse data set is actually finding one online. The data set I wanted to work with were text files which contain a given number of words in them, which is apparently way too specific of a data set to find online. The next best thing was to try and find an API that can be searched by using the number of words a certain data set contains. There was one API, but you needed to pay for it and I really don't want to drop money on an extension for a project. So the next best thing was to use a dictionary API, which are not only free, but they are also pretty easy to use. I decided to use the most straightforward API and that was a collection of 10000 words in this website (https://www.mit.edu/ ecprice/wordlist.10000). This is where the TextGenerator class comes in! This class is actually pretty sweet, it uses the Java Net package to get access to the website and create a data stream to our java application. From there, I just the Utility and IO packages to scan the data stream and write a text file containing random words. This entire process is done by the writeText method which inputs the a filename and the number of words that we want within that file. With the TextGenerator class, I was able to write about 9000 text files, each with about 1000 to 10000 words in them. Since we are picking random words from the website, we can definitely go above 10000 words, but then there will be memory issue.

The next class that I used for this extension is the TimeComplexity class, which reads a text file in the Data dirctory (like the TextStatistics class) and then outputs a csv containing data about the total number of words and the run time. I was worried that the data would be noisy, so I decided to use the average run time from three trials in order to avoid noise and also to avoid a memory error. The results from this extension were pretty awesome and we actually got to see a neat plot for the BST.

For the BST, we see that we are roughly getting a time complexity of O(n), which is awesome and exactly what we expect from an unbalanced BST. This means that we can definitely improve the run time by implementing a self-balancing BST instead of just having a normal BST. For the hashmap, the plot looks very weird! We do see that there doesn't seem to be any growth, which is great since this implies that the hashmap has a time complexity of O(1). We do see that there are moments in which the run time randomly shoots up, which can be a result of the computer doing background tasks on top of running the java program, or just noise within our data which is normal. Overall, we see that we don't really have much that we can do to improve the run time for



(a) BST time complexity plot (b) Hashmap time complexity plot

Figure 4: Time complexity plots for both a BST and Hashmap data structure

a hashmap, but we certainly have a lot of work we can do to improve the run time for a BST.

Reflection

For this project, we got to compare two different data structures and their run times for building a word map. I didn't really learn anything new, but I did get to see a real world example of how our choice of a data structure can impact the run time for the programs that we make. I also got to see how the implementation of a data structure can impact the run time and how it might be possible to significantly decrease our run times if we implement a data structure properly

References

For this project, I used https://www.mit.edu/ ecprice/wordlist.10000 to generate my data sets for my extension. I also used the Java API to learn how to use the Net package to access a website through java.