Assignment 7: Banhammer

Joshua Revilla

In this assignment, we were instructed to build a sort of text parser. What we do is read through stdin and check if any of the words that were inputted match a list of illegal words. These words are in newspeak.txt and badspeak.txt. In order to make this process extremely fast, we use a couple of data structures including: Hash Tables, Bloom Filters, and Binary Search Trees. These data structures make it so that we can check if the word is in the set of illegal **extremely** fast. The two data structures that really effect the speed however, are the Bloom Filter and Hash Table. Because these structures have $O(1)$ look up time, and the Bloom Filter has to just check 3 bits, we can look up any word passed in to stdin really fast. However, we need to make sure that these structures are large enough to actually make it faster.

Here's the gist of what we're doing with these structures. We are passed a word and we hash it, we then check the Bloom Filter at the index of the hash, if it's one that means that the word might be illegal. We then check the Hash Table, which is slightly slower, at the index of the hash to really check if the word is illegal. In order to deal with hash collisions, we put Binary Search Trees at every index of the Hash Table so that there can be multiple words at the same index. Now think about the Hash Table with only one index. Every single element would be on the same Binary Search Tree, which means look ups would take far longer since we have to search through a far larger tree. Although look ups, in a balanced tree, aren't horrible slow, $O(log(n))$, it's still far slower than searching through a Binary search tree with only one element. However, a Hash Table of size 1 is an *extreme* case, and

also kind of redundant, The default size for the Hash Table is 2^{16}. The size of the Bloom Filter also affects the runtime, if the Bloom Filter is too small then the program will have to perform more look ups since we'll have far more false positives. The default size of the Bloom Filter is 2^{20}. With default values for the size of both of these data structures and the list of badspeak and newspeak supplied to us in the resources, the Hash Table load and the Bloom Filter load are at 19.97223% and 4.08268% respectively. The load of these data structures are defined by the amount of non NULL entries compared to the total size.

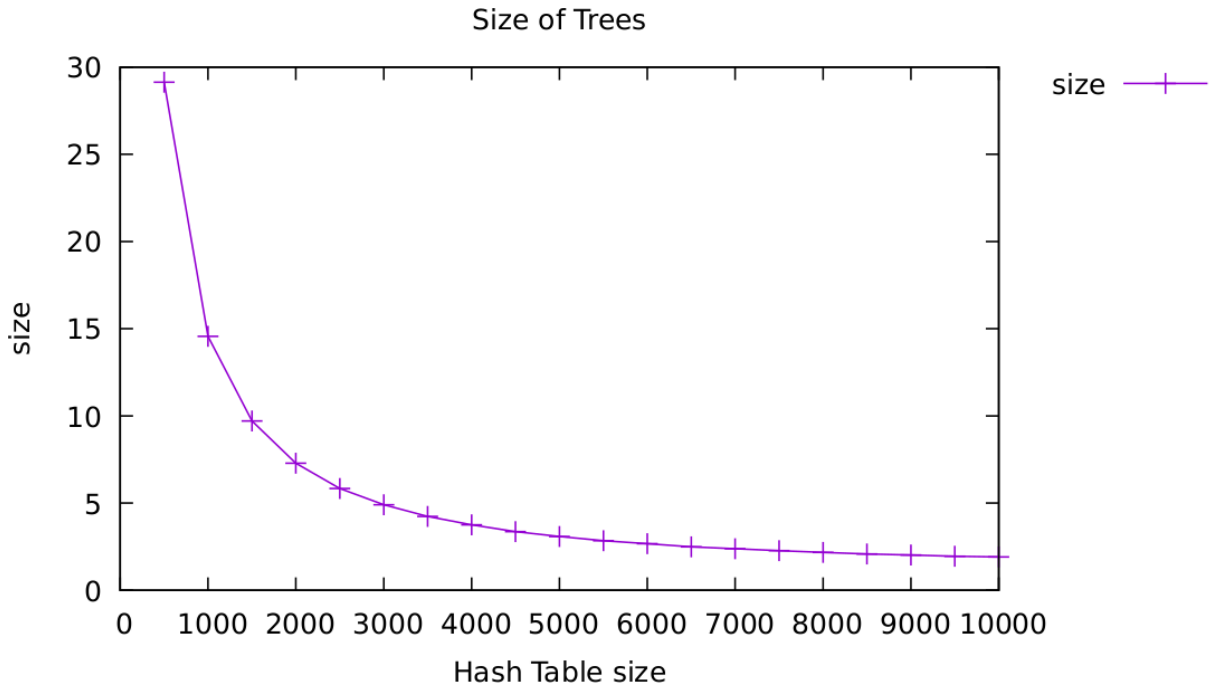
$$BloomFilterload = \frac{BloomFiltercount}{BloomFiltersize}$$

$$HashTableload = \frac{HashTablecount}{HashTablesize}$$

1. Tree Average Size

Another extremely important statistic for run time is the average size of a tree in the Hash Table. With all default settings, the average size of a tree is 1.113072. The size is defined by how many nodes there are in the tree. Since the number is close to one, that means, for the most part, we have avoided hash collisions and look ups within the trees should be pretty fast. The size of the Hash Table will directly effect the size since we would then need to fit more words in a smaller amount of trees.

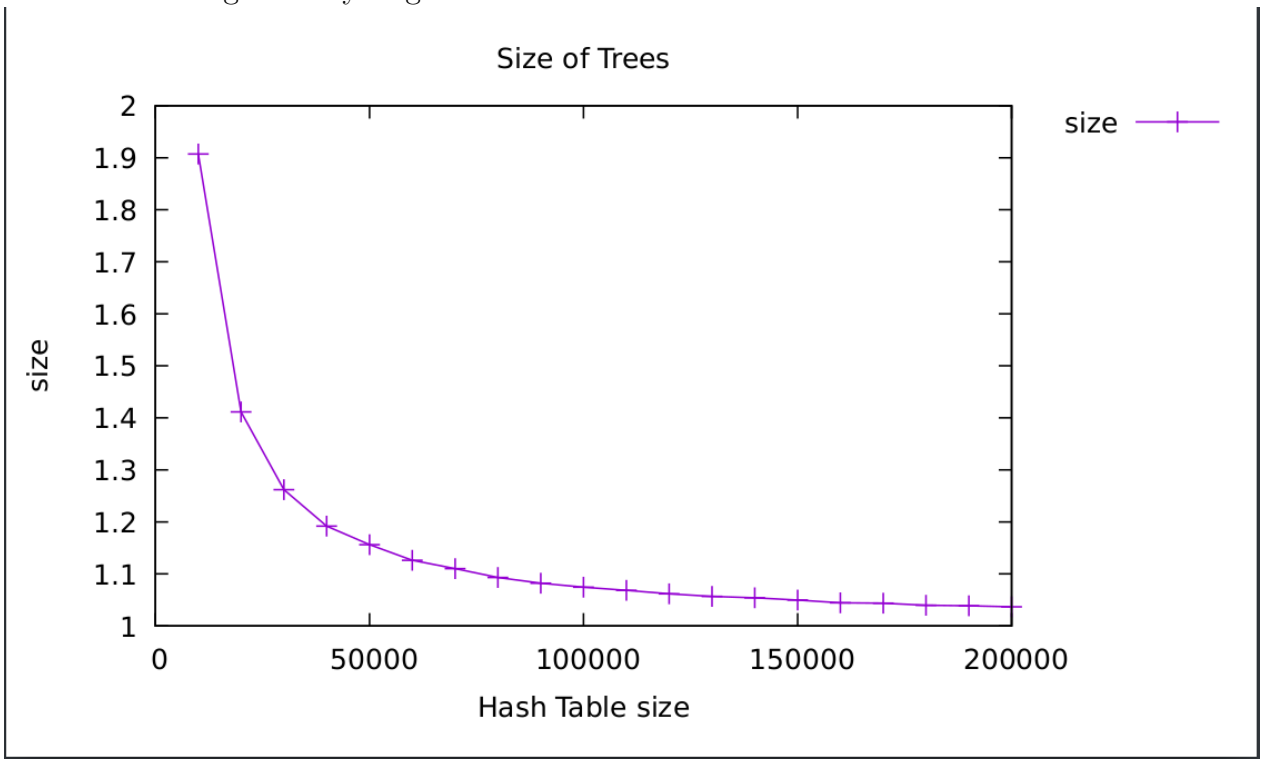
Here is a graph showing the average height of trees with small, but growing, sizes of the Hash Table:



These numbers are **far** smaller than the default value, but here we can see some drastic changes. We start at the Hash Table having 500 entries. At 500 Entries, the average size of a tree in the table is 29.14 which isn't horribly large, but will slow us down a great bit when compared to trees with the average size of around 1. as we start increasing the amount of entries the average size of the trees go down, However, it seems to start to decline how fast the average size decreases. At size 10000, the average size is 1.9071 which isn't horribly far off from the average size from the default value of 2^{16}, which is **insanely** larger. So clearly there is diminishing returns in making the Hash Table beyond astronomically large. The Hash Table load for the smaller values is at 100, which means at every entry there is a tree while at 10000 the Hash Table load is at around 76%. For comparison, the Hash Table load for the default settings is only at around 20%.

As stated before, the larger the Hash Table gets, the less it starts to change the average size of these trees. Here is a graph that shows how little the average size of the tree changes as

the Hash Table gets fairly large:

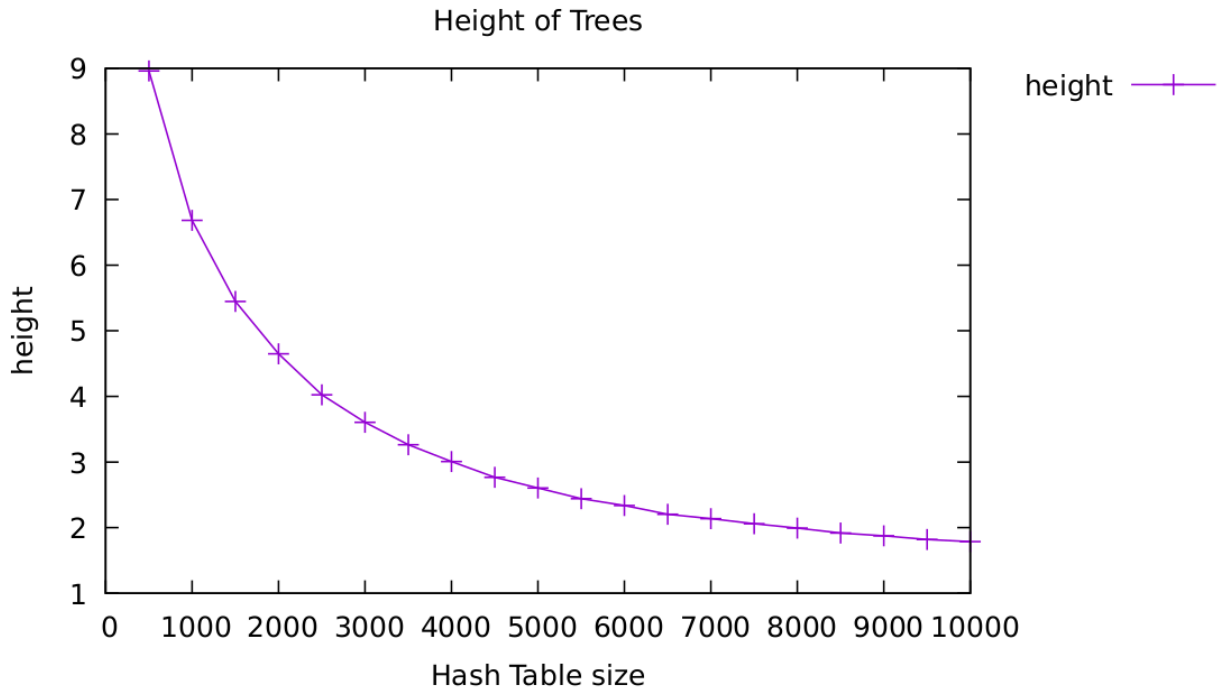


You can see that this graph looks similar to the first which means that the size will start to matter less and less as the Hash Table gets **really** large and is pretty mush not worth the cost in memory.

2. Tree Average Height

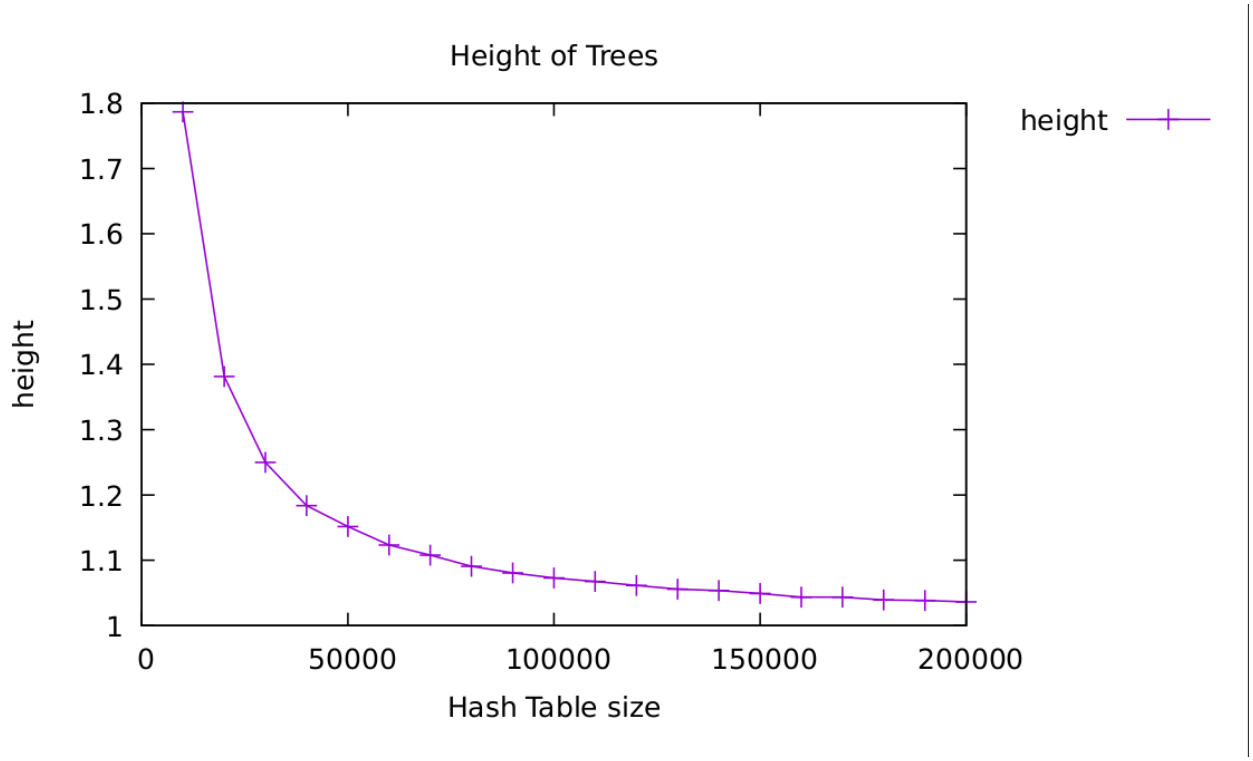
Another statistic that is fairly intertwined with size is average height. This is the average height of all the binary search trees. The higher the height, the more branches the program has to traverse to find, or not find, the word it is looking for. With default settings, the average height is 1.110016 which means most trees are simply one node. Height is directly tied with how long it might take for a word to be found in a tree. at worst case you would need to search height amount of times to find a certain node so you really want the height to be small.

Here is a graph that shows the average height of the trees as the Hash Table is pretty small:



as you can see, The height starts out pretty high at around 9. This means you may have to traverse more branches when searching for a node. It quickly decreases as the Hash Table gets bigger in a similar pattern to the size decrease. These statics are fairly intertwined since the large the tree the bigger the height. Similarly to the size of trees, there is diminishing returns to how big the Hash Table and the average size of trees.

Here is a graph showing the average height of trees as the size of the Hash Table starts getting **really** large



As you can see, At one point, the height of these trees start to not get significantly smaller at a certain threshold of Hash Table size. At some point the trade off between the memory usage, and the speed of searching through these trees is simply not worth it. The average height of the tree with the Hash Table size of 200000, isn't that much smaller than the default size of 2^{16} which is far smaller.

3. Bloom Filter and look ups

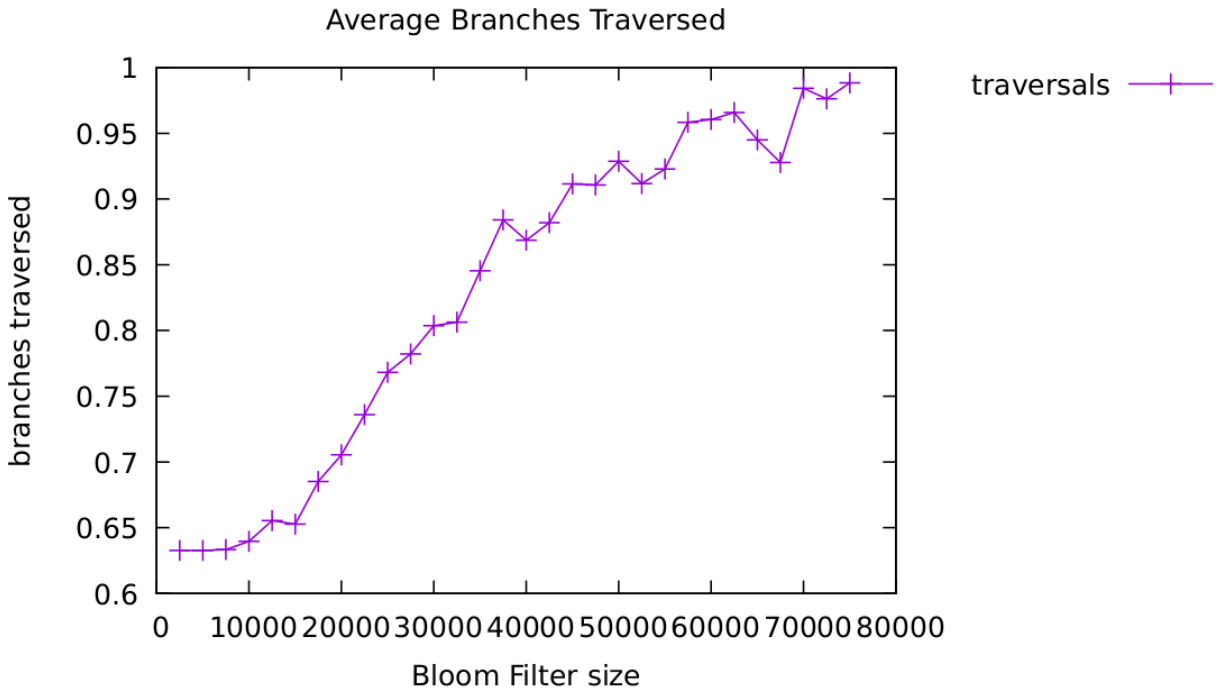
The Bloom Filter is an extremely fast way to make sure something isn't an illegal word. You check 3 bits and one of them is not a 1 then that thing isn't an illegal word. Ideally, you would only have to go to the Hash Table to find a word if you know its Illegal, but false positives do occur in Bloom Filters. One way we can minimize the false positives is by making the Bloom Filter bigger so that false positives are extremely rare. Look ups are extremely costly, so you wouldn't want to keep looking stuff up in your Hash Table if you

don't need too. The default size of the Bloom Filter is 2^{20} which is extremely large. With the default values and text files, the Bloom Filter load is only about 4% which means only 4% of the bits are set to 1 which means that false positives are pretty rare.

3.1. Testing

- In order to test the amount of look ups performed, we need to first pass it a thing of text so that it has words to look up. For this test we will be passing in the Alice's Adventures in Wonderland from the resource repository. All of these tests were also done with the default Hash Table size.

Here's a graph that shows average branch traversals as the Bloom Filter gets larger:



The amount of average branches goes up due to the fact that the amount of look-ups being performed is **far** less. Because we find more information from the Bloom Filter if it's bigger, we can decide we don't need to go to the Hash Table more often since we get way less false positives. While the graph may look slightly misleading, The program runs far faster when

the Bloom Filter is longer so that we look through the Hash Table far less.

4. Conclusion

Overall, this assignment was really interesting because it has a lot of *go fast* optimizations. We do so many things, like the Binary Search Tree and the Bloom Filter, that are purely there to make this process way faster. The Bloom Filter is insanely smart and is a huge optimization I would never have though of. We also got to see how magical hashing is. This process only works because hashing is fast and deterministic so we can rely on the numbers that come out of the hash. Taking out one of these structures, usually by setting their size to one, makes the process of checking these words that flow in **incredibly** slower than what it is normally. In conclusion this was an incredible assignment that really made me feel like a real engineer making speed versus memory trade-offs.

4.1. Personal Note

This class has been the hardest thing I've accomplished, but it has made me feel like way more of a programmer than ever before. To whoever may read this, thank you, no one has effected the way I program quite Like Eugene, Professor Long, and the extremely helpful staff of this class.