Binary Search trees for Resolving Hash Collisions

Alex Asch

December 2021

1 Abstract

This paper outlines the statistics for a binary search tree implemented using a Bloom filter, and a Hash table of binary trees. These are combined to form a firewall which reads user input and scans it for undesirable words from a previously established list of words. This is first achieved by adding a word into a Bloom filter, which is a bit vector, and set the bits which are the result of the given word hashed, modulo the vector length. This provides a way to check if a given word is in the list of words added to the Bloom Filter, because if the three bits obtained from hashing the word are set, the word may be in the Filter. However, since the vector is of a finite length, and two words can possible Hash to the same value, the hash table exists. The hash table is an array of binary search trees, which is the deterministic way to check if a word is in the given list. The hash table operates by hashing a word, and the index of the hashed number is a binary search tree containing each word that hashes to that index on the list. With the combination of the two above methods, it is possible to both quickly check if a word is in the array of words that must be censored in a deterministic manner.

2 Introduction

To begin, it is important to explain what Hashing is, Hashing a word means that we take a string, and using a hashing function, and make it an integer. The hashing function is made in a way that takes the given data, and makes a number such that hashing something will give the same output every time. Using this, we create a hash table, which is a table of key value pairs, such that the hashed key points to the items that hash to that key. It is important to note that hashing two distinct items may result in the same key. This is why we use a Bloom filter and hash table to make sure that words are flagged correctly. The Bloom filter hashes values three times using three different algorithms, and each of these keys are the set bits, thus an item is in the Bloom filter only if all three of it's keys are set bits. However, this method still has the chance of hash collisions giving false positives, thus the hash table is used. The hash table only hashes once, however, instead of simply storing a bit, each key has a binary tree

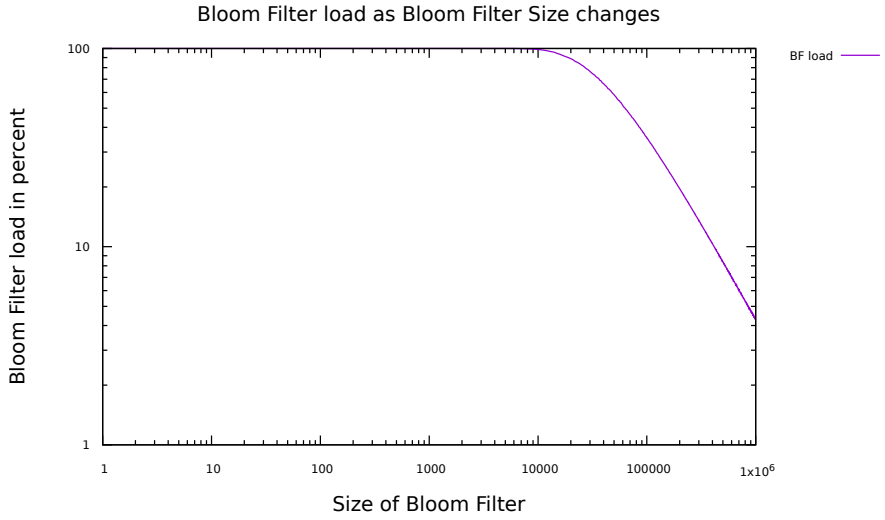
of values, and the binary tree is searched through, to determine if a given word is in the hash table. These two methods together are used, as stated before to check if a word is in the list of banned words quickly and accurately. This paper then is to evaluate the findings of different hash table and Bloom filter sizes on the statistics of the data, which are the load on the Bloom filter, and hash table, and the average height, size, and trees traversed of the binary search trees.

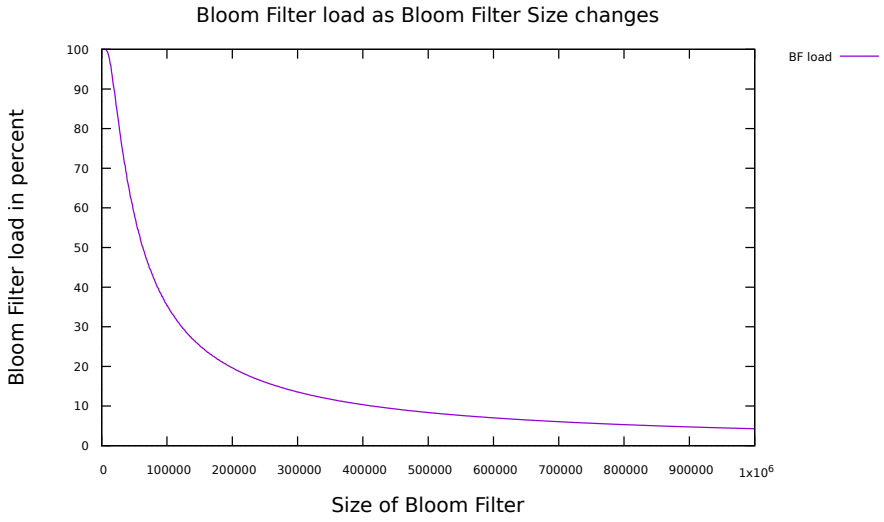
3 Materials and Methods

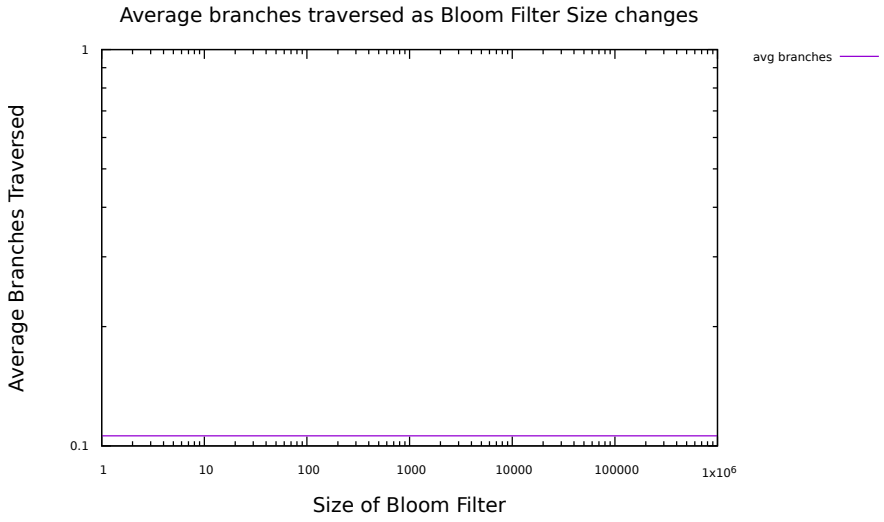
To gather our data, we repeated usage of the code with the same list of banned words, and message, and varied the size of the Bloom filter and hash table independently. From this, we gathered the data, which is the average height, branches traversed, and size of binary search trees, and the load in percent of the hash table and Bloom filter. The results gathered are demonstrated in the graphs below, with pairs of logarithmically scaled, and unscaled graphs.

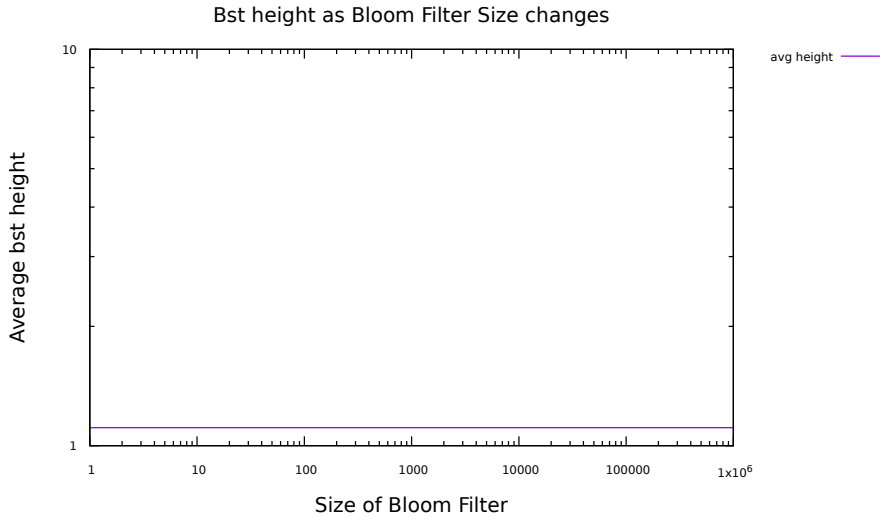
4 Results

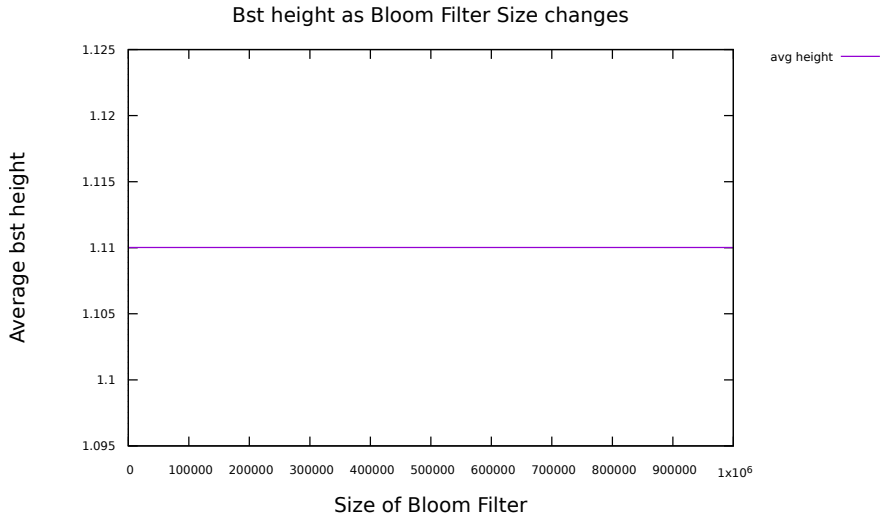
The results below show that for changes in Bloom filter size, all parameters except the load of the Bloom filter remained constant. This is due to the binary search trees not being tied to the size of the Bloom filter. We also see that the changes in hash table size effect the size of the binary search tree size and height in a manner that decreases as the hash table size increases. This will be further discussed in the next section.

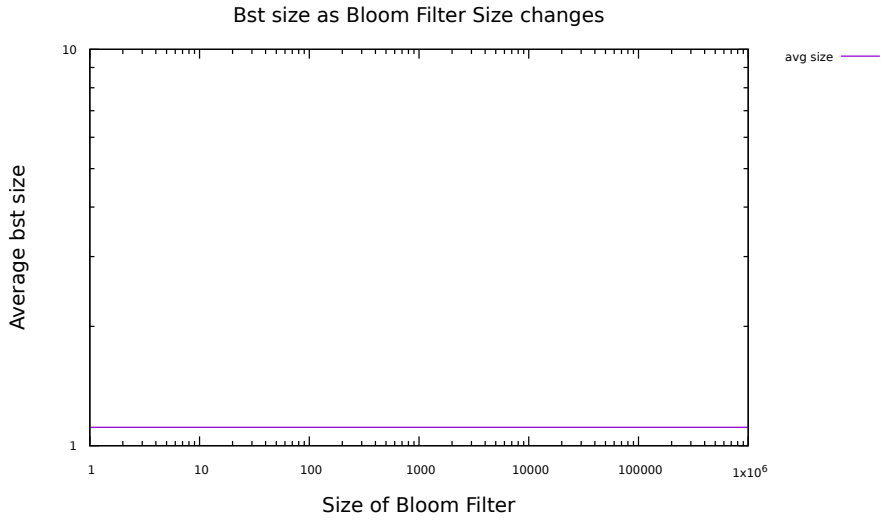

Bloom Filter load as Bloom Filter Size changes
Bloom Filter load in percent
100
10
1
1
10
100
1000
10000
100000
1x10^6
Size of Bloom Filter
BF load

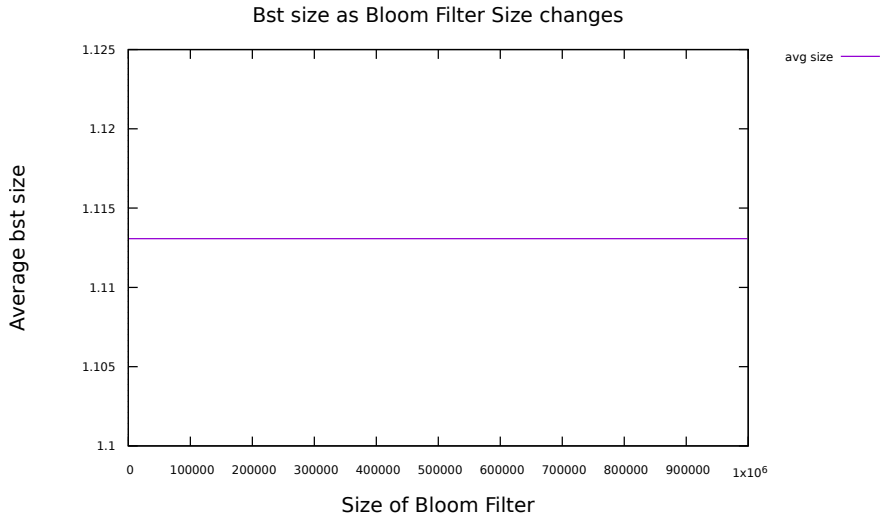

Bloom Filter load as Bloom Filter Size changes
Bloom Filter load in percent
Size of Bloom Filter
BF load
100
90
80
70
60
50
40
30
20
10
0
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1x10^6

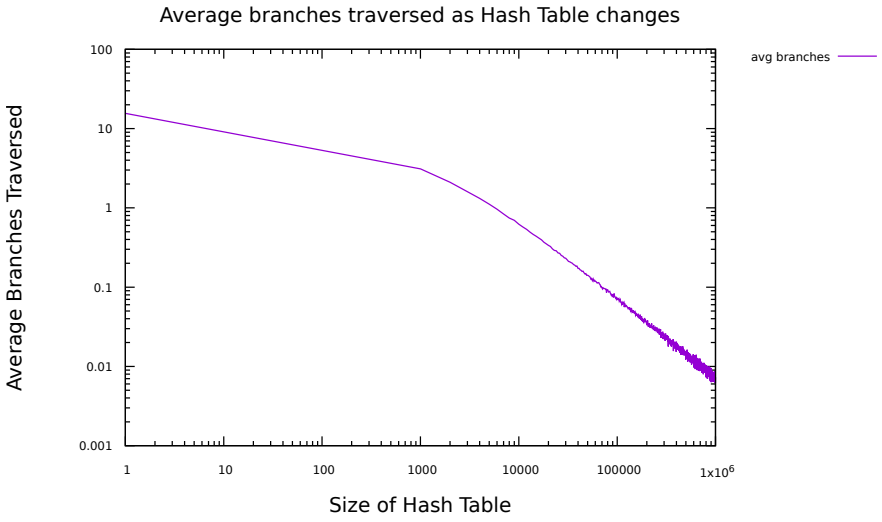

Average branches traversed as Bloom Filter Size changes
Average Branches Traversed
Size of Bloom Filter
avg branches
1
0.1
1
10
100
1000
10000
100000
1x10^6

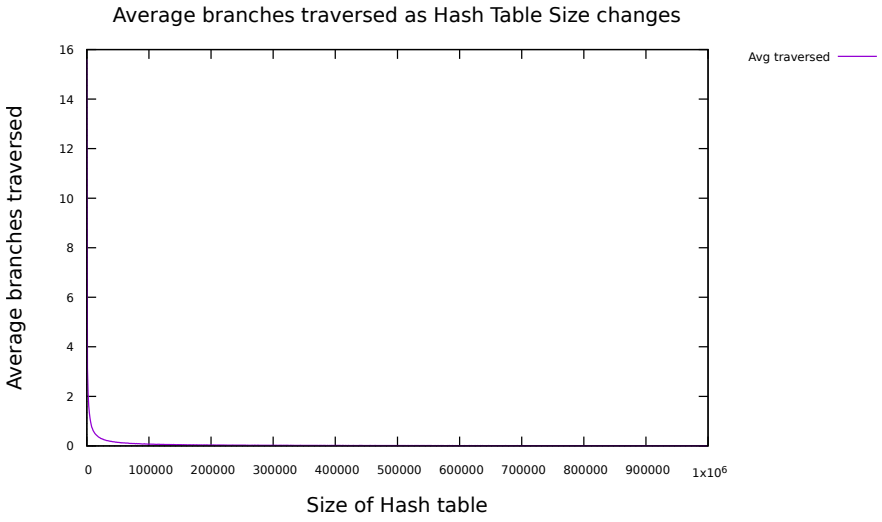

Bst height as Bloom Filter Size changes
avg height
Average bst height
Size of Bloom Filter
10
1
1
10
100
1000
10000
100000
1x10^6

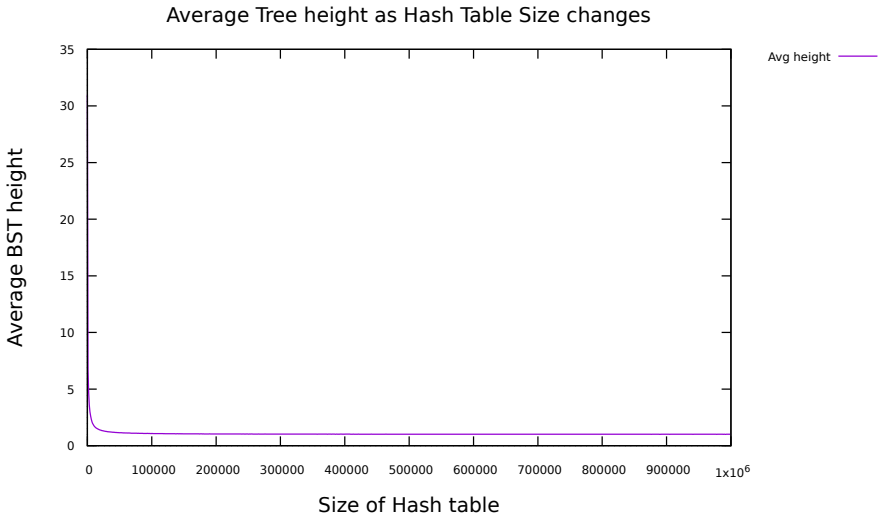

Bst height as Bloom Filter Size changes
avg height
Average bst height
Size of Bloom Filter
1.125
1.12
1.115
1.11
1.105
1.1
1.095
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1x10^6

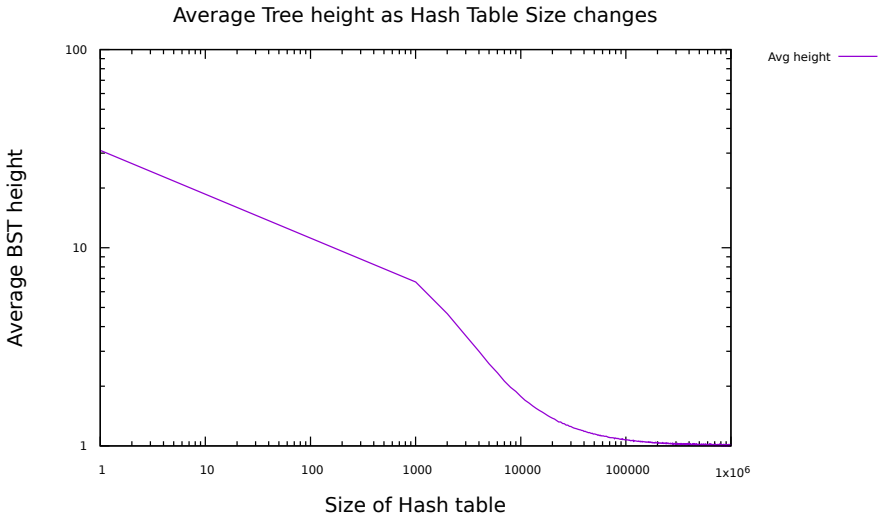

Bst size as Bloom Filter Size changes
Average bst size
Size of Bloom Filter
avg size
10
1
1
10
100
1000
10000
100000
1x10^6

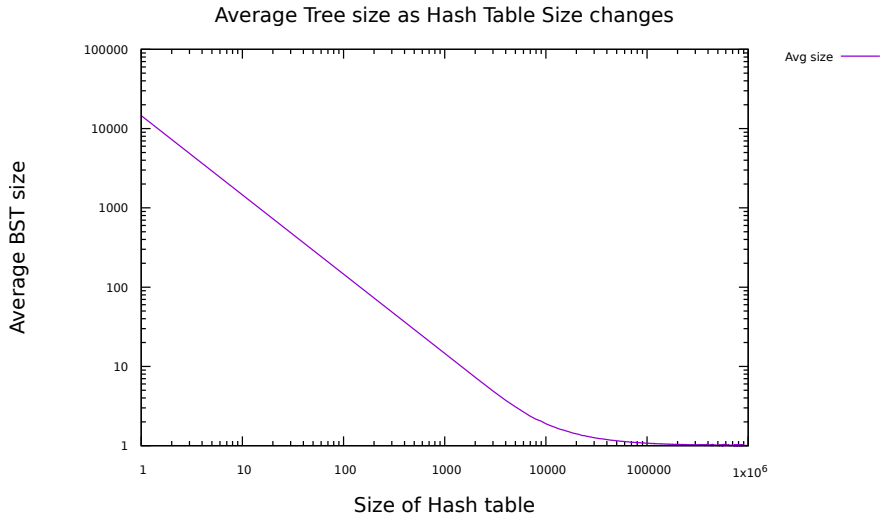

Bst size as Bloom Filter Size changes
Average bst size
Size of Bloom Filter
avg size
1.125
1.12
1.115
1.11
1.105
1.1
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1x10^6

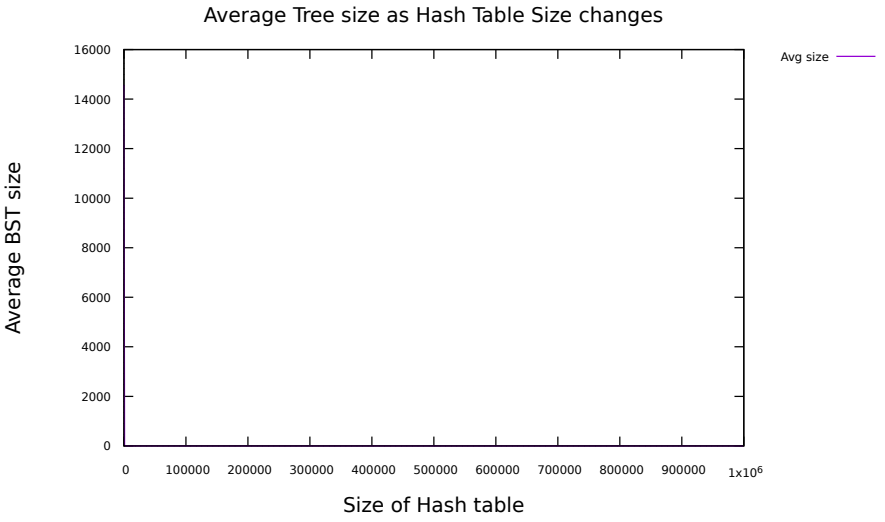

Average branches traversed as Hash Table changes
Average Branches Traversed
Size of Hash Table
avg branches
100
10
1
0.1
0.01
0.001
1
10
100
1000
10000
100000
1x10^6

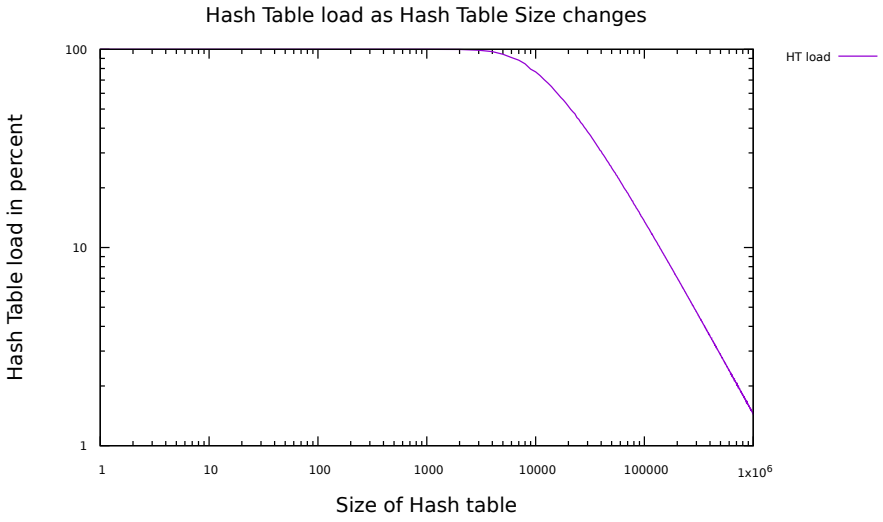

Average branches traversed as Hash Table Size changes
Average branches traversed
Size of Hash table
Avg traversed
16
14
12
10
8
6
4
2
0
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1x10^6

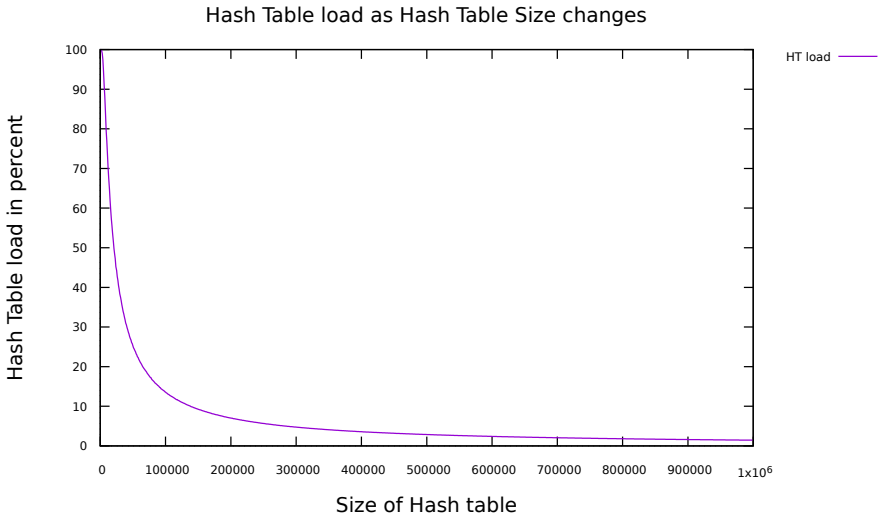

Average Tree height as Hash Table Size changes
Average BST height
Size of Hash table
Avg height
0
5
10
15
20
25
30
35
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1x10^6


Average Tree height as Hash Table Size changes
Average BST height
Size of Hash table
Avg height
1
10
100
1
10
100
1000
10000
100000
1x10^6


Average Tree size as Hash Table Size changes
Average BST size
Size of Hash table
Avg size
100000
10000
1000
100
10
1
1
10
100
1000
10000
100000
1x10^6


Average Tree size as Hash Table Size changes
Average BST size
Size of Hash table
Avg size
16000
14000
12000
10000
8000
6000
4000
2000
0
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1x10^6


Hash Table load as Hash Table Size changes
Hash Table load in percent
Size of Hash table
HT load
100
10
1
1
10
100
1000
10000
100000
1x10^6


Hash Table load as Hash Table Size changes
Hash Table load in percent
Size of Hash table
HT load
100
90
80
70
60
50
40
30
20
10
0
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1x10^6

5 Discussion

To begin, we will discuss are results pertaining to the changes in Bloom Filter size. As we can see, other than the load of the Bloom filter, no statistics vary as Bloom filter size increases. This is because the binary search trees are tied to the keys in the hash table, and not the Bloom filter. While a smaller Bloom filter will increase the amount of lookups needed, as there will be more false positives, we are measuring average branches traversed, instead of total trees traversed, the data for the average branches traversed does not change either. Now onto the statistic of importance when discussing the Bloom filter size changes, the Bloom filter load. As we can see from the non logarithmically scaled graph, the load of the Bloom filter first decreases sharply as the size of the Bloom filter changes, but then begins to decrease at a slower rate. This is because, at first, with a Bloom filter of one, the filter is always full, as the hashed key modulo the length will always point to the same number. From this, as the number of indices in the Bloom filter increases, the data points begin to spread out more rapidly, and thus do not overlap as much, and also do not fill the table to the same degree. However, this trend of rapid decreasing changes as there will always be enough bits set in the filter as appropriate for the hashed keys, in addition, as the filter gets longer, the set bits are more sparse, thus making each added bit less likely to free a significant amount of space. Furthermore, we also tested the data with a change in the size of the Hash table. This nets more interesting results as the load of the hash table is tied to the statistics of the binary trees. As we saw before, the binary search tree branches traversed, height, and tree size decrease as the hash table size increases. This is because as the size of the hash table increases, the load of the hash table decreases, thus the number of hash collisions decreases. We can also see the slope of the change flattening as the size gets larger and larger. This is because the binary search trees get larger as hash collisions occur, and as the size of the hash table gets larger, each added index to the size frees up less and less hash collisions. This is because as the indices in the hash table are calculated using a modulo operation, as the number that is used to take the modulo increases, the number of inputs with the same result decreases. In addition, we can also observe that the size, height and branches traversed tend towards one, this is because in a hash table with no collisions, each index that is populated will be populated with only one node, thus being a tree with a height and size of 1, and can be searched by only viewing one element.

6 Acknowledgment

In this section, I would first like to acknowledge Professor Long for the necessary skills in coding in the C language to implement the above functions, and the guidance on both coding and analysis via the assignment document. In addition, I would also like to acknowledge Professor Long for the provided header files that guided implementation of the code.

In addition, I would like to acknowledge Eugene Chou for the guidance and starting ideas for implementation of the code in sections.