Analysis - The Great Firewall of Santa Cruz: Bloom Filters, Binary Search Trees and Hash Tables

Student name: *Serjo Barron*

Course: *CSE13S* – Professor: *Prof. Long*
Due date: *12/5/21*

1 Introduction

There are plenty of available data structures to store and lookup data. But in this specific paper, we will examine two data structures: a hashtable consisting of many binary search trees and a bloomfilter. There will be many graphs to observe the behavior and efficiency of these data structures as the size/length of the data structure increases.
Since we are only examining the data structures that have been used in the program, there is no direct comparisons being made between other existing data structures. Rather this paper will focus on the strengths of using a hashtable consisting of binary search trees as a means of storing words, and the time complexity of inserting and looking up said words. The same will also be done for the bloomfilter, which will be interesting as it is a more probabilistic form of storing data as opposed to the hashtable.

2 Methods

The data presented in this paper has been produced and made in the C language. The numbers are often unsigned integers when dealing with whole number data such as the sizes of the hashtable and bloomfilter, and the amount of branches and lookups. Though for data that requires more computation such as the average size of binary search trees in a hashtable were represented using long floats up to 6 decimal places of precision.
The graphs used in this paper have been plotted using a tool called gnuplot. It should also be noted that since the sizes of the hashtable and bloomfilter being plotted are quite large, the graphs have been adjusted accordingly to use a proper increment value.

2.1 Graph Variables:

To familiarize what the axes of the graphs in the Results/Analysis portion of the paper represent, the various variables and variable computations are listed below.

1. hashtable size
2. bloomfilter size
3. average BST size

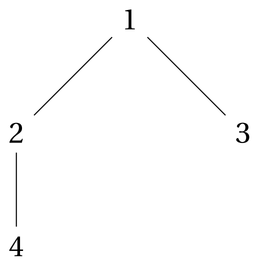
  - calculated as: sum of non-NULL BST sizes / ht_ count
  - a size of a BST is equivalent to the number of nodes in the tree

4. average BST height

   - calculated as: sum of non-NULL BST heights / ht_ count

     The height of a BST is essentially the depth of a BST, depth meaning what is the lowest/farthest to reach node (so the depth is the amount of traverses made in order to reach that node).

     example binary tree:

     The height = depth, so the height is 2, since it takes 2 traverses starting from '1' to reach node '4'. The path is then [1->2->4].

5. average BST branches traversed

   - branches (for every instance of a right/left traversal of a BST)
   - lookups (a lookup is every instance of inserting/looking up on the hashtable of BST's)
   - the average branches traversed is calculated as: $\frac{branches}{lookups}$

6. hashtable load

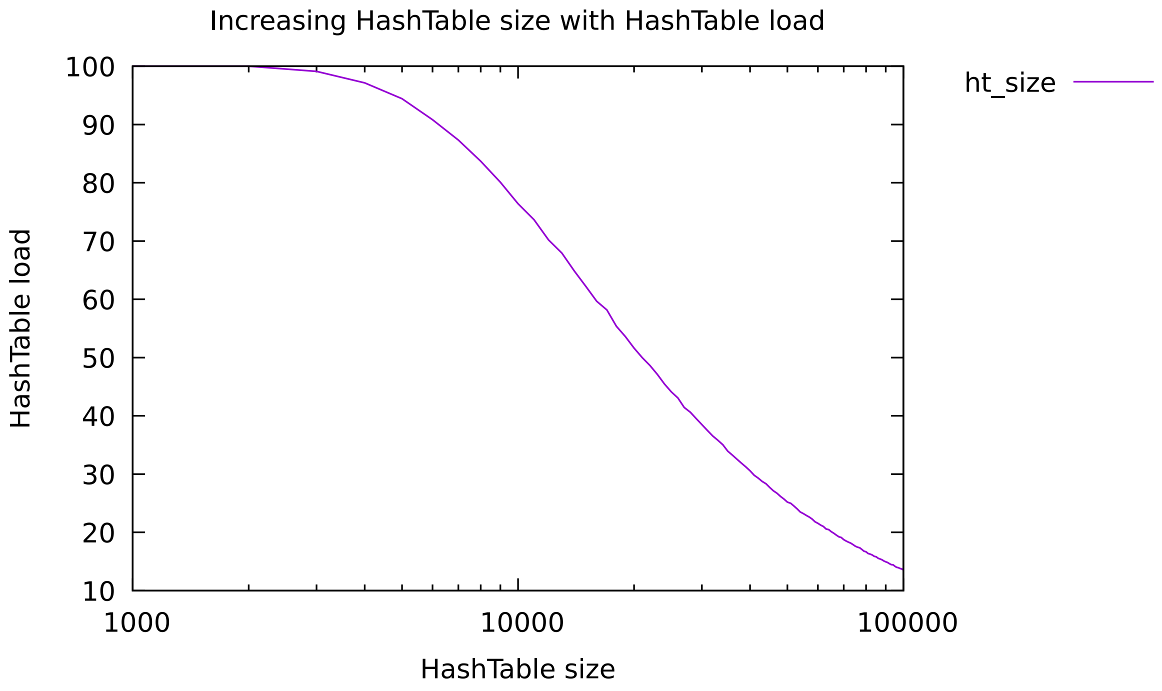
   - calculated as: $\frac{ht\_count}{ht\_size}$
   - ht_ count is the number of non-NULL BST's in the hash table

7. bloomfilter load

   - calculated as: $\frac{bf\_count}{bf\_size}$
   - bf_ count is the amount of set bits in the bloomfilter (set bits are bits that are 1, so 0 is not counted as a set bit)

3 Results/Analysis

3.1 Hashtable size Against Load

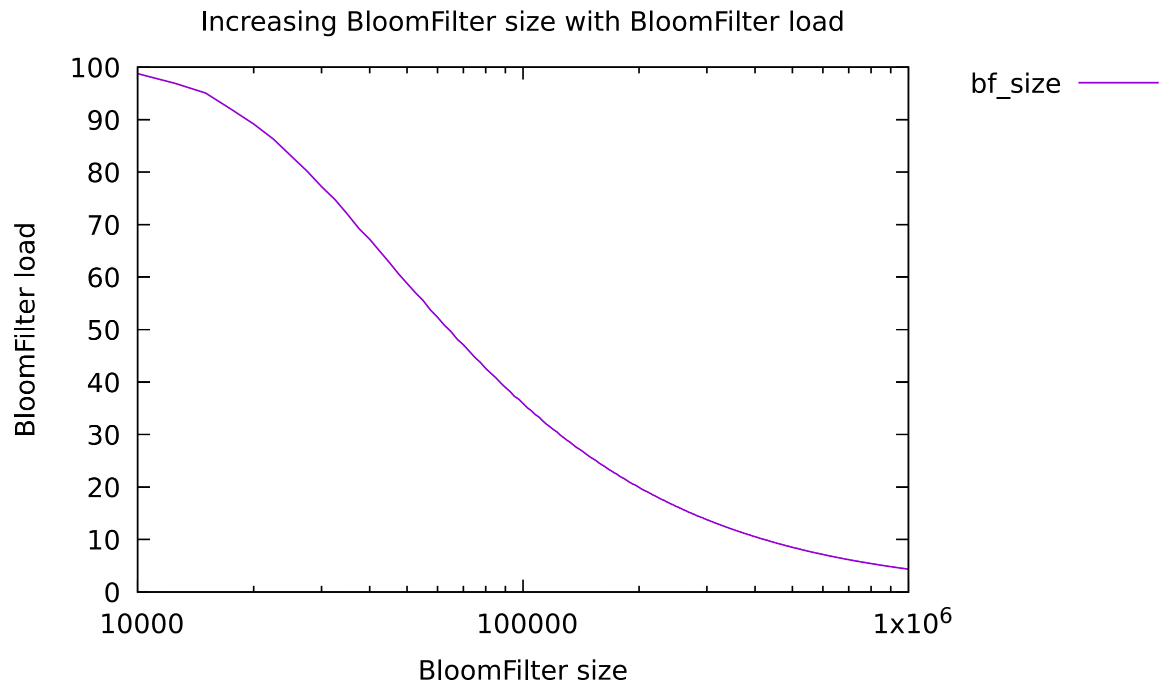


The graph above represents how the load of the hashtable behaves as the size of the hashtable increases, and as seen in the graph above there is an obvious behavior that the load decreases as the hashtable size increases.

The hashtable load is important since it gives an idea of the probability of hash collisions. Hash collisions are scenarios in which two keys result to the same piece of data. This of course is avoidable if the size of the hash table is bigger than the amount of data stored (in this case we are storing binary search trees). So in order to avoid hash collisions we want the load of the hash table to not be full.

As observed in the graph and the fact that the default hashtable size for reference is 2^{16} or 65536 which gives us a hashtable load of around 20%, which means that around only 20% of the BST's in the hashtable are being used. Hence, the chance of a hash collision occurring would be very low.

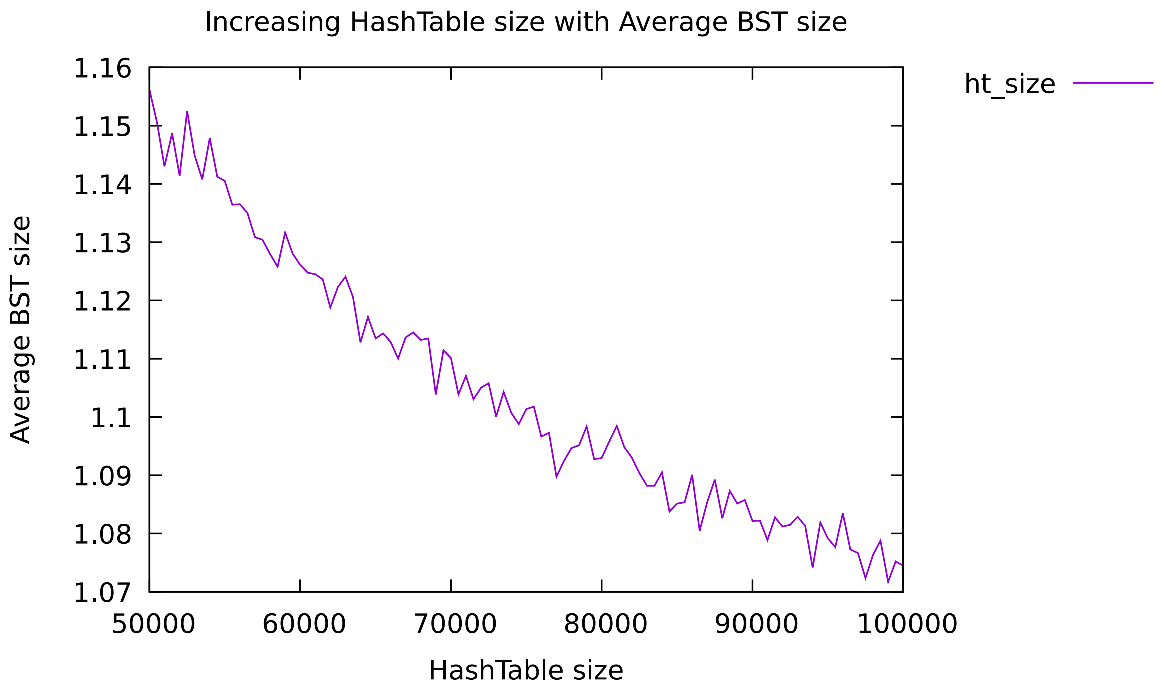
3.2 Bloomfilter size Against Load



The graph above represents how the load of the bloomfilter behaves as the size of the bloomfilter increases, and as seen in the graph the load decreases as the bloomfilter size increases. Similar to the key to data pair of a hashtable, a bloomfilter also uses functions (keys) to decide where to set a bit. But this may result in the case of a false positive, so, to avoid this the bloomfilter graphed above makes use of multiple functions (keys), 3 to be exact. By using multiple functions $f(w)$, $g(w)$, and $h(w)$, this will ensure that $f(w) \neq g(w) \neq h(w)$ is likely to be the case with little room for the case of a false positive.

The chance of a false positive is only lessened when the size of the bloomfilter is increased. For reference the default size of the bloomfilter is 2^{20} or 1048576, which is slightly over what is represented in the graph. But as seen in the graph at a size of 1000000 for the bloomfilter gives us a load of 4%, which means that only 4% of the bits in the bloomfilter (essentially a very large bit vector), are set to 1.

3.3 Hashtable size Against Average BST size of Hashtable



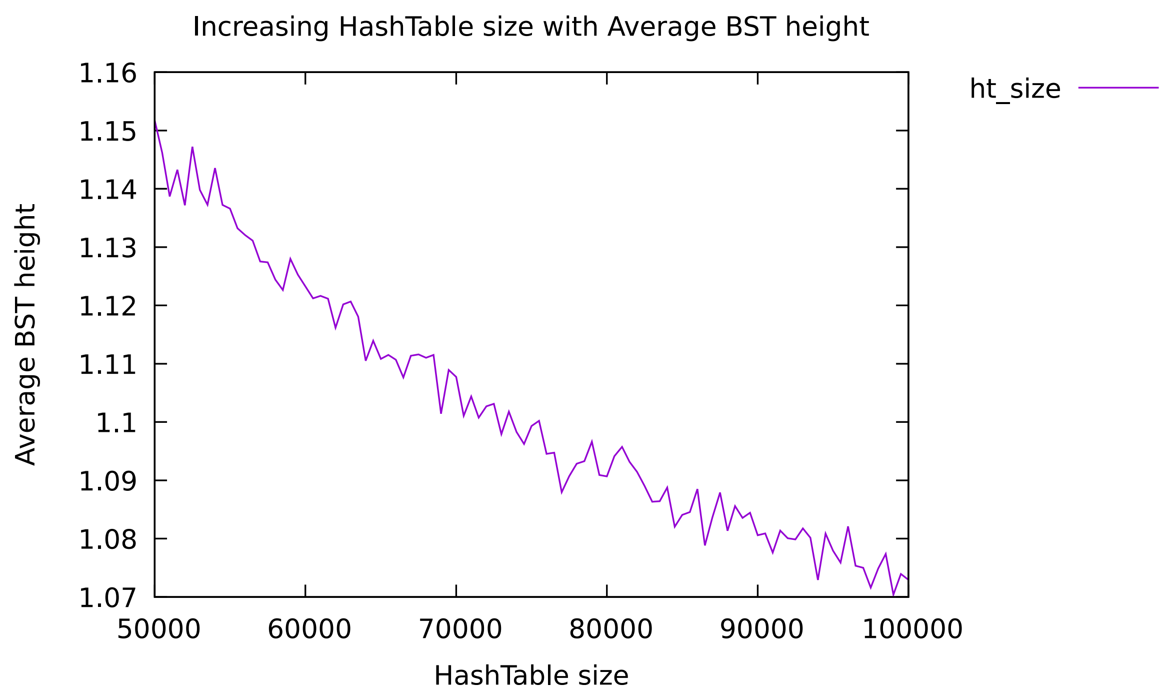
The graph above represents how the average BST size behaves as the size of the hashtable increases. As seen above, the average BST size (the amount of nodes present per non-NULL binary tree) also decreases.

What this means necessarily is that since there is less nodes per tree, that will also respectively decrease the height of a binary tree which will be examined in the next subsection. And with the height of a binary tree being synonymous with the depth of a binary tree, this will respectively reduce the overall insert and lookup time for nodes. Examples of the time complexity of different binary tree scenarios will be given in the next subsection.

As mentioned previously the default hashtable size is 2^{16} or 65536, which by looking at the graph is approximately a BST size of 1 for the non-NULL BST's of the hashtable. This means that the tree is on average 1 node. And since the size is 1, the lookup time of a particular node of a binary tree would be $O(log(1))$ taking into account that the average tree is balanced.

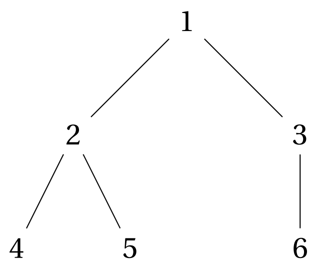
Now to discuss the oscillating nature of the graph. The likely reason behind this is the hashing, which decides what index of the hashtable to go to. Hashing will cause randomness in what index in calculated, hence the reason the graph appears to oscillate randomly. What this means is the hashtable could sometimes visit one index more often than other, which will cause more nodes to be inserted to a BST more often, which inevitably leads to a higher average BST size.

3.4 Hashtable size Against Average BST height of Hashtable



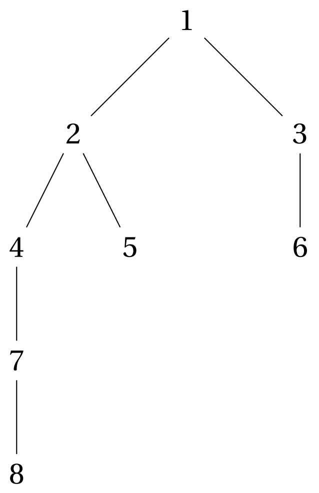
The graph above represents how the average BST height behaves as the size of the hashtable increases, and as seen above, the average BST height (also known as the depth of a binary tree) also decreases.

As discussed previously this graph will have quite similar behavior and oscillations to the previous graph comparing how the average BST size changes as the hashtable size increases. This is since the height of a binary tree is calculated as: $1 + max(left, right)$ (left & right being subtrees), which should be near $\frac{1}{2}$ of the value of the binary tree size - 1. This though is assuming we are always dealing with balanced binary trees, which isn't the case.
example of balanced binary tree:



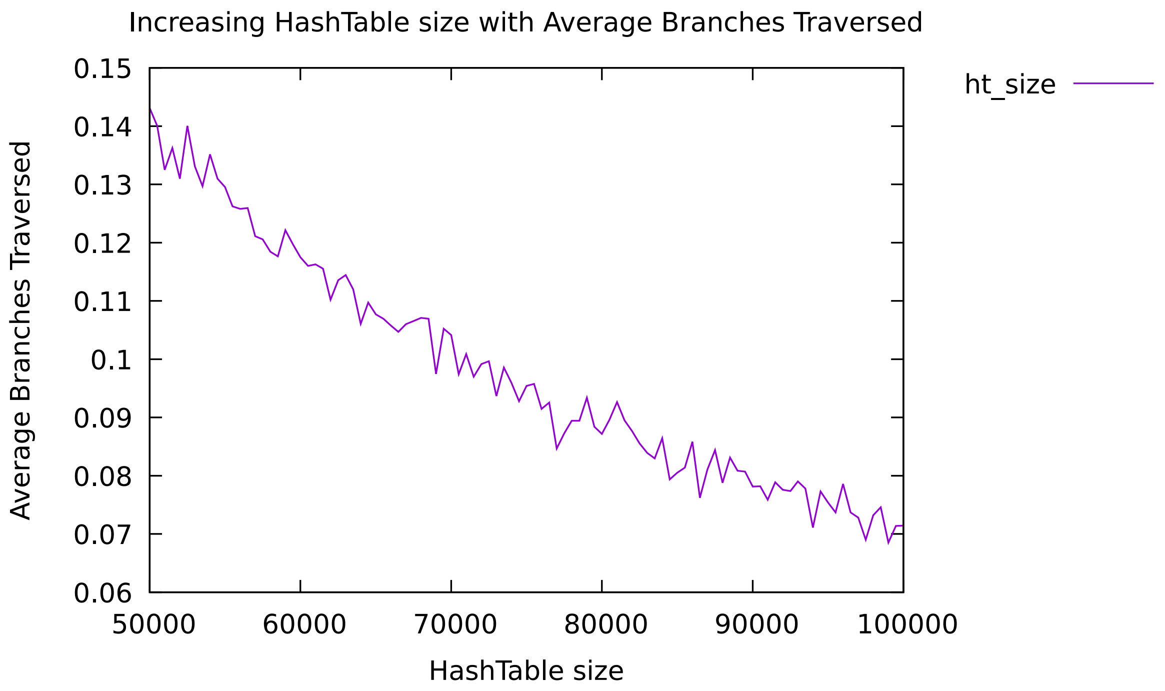
For the most part the binary trees being dealt with the hashtable are balanced as observed by the behavior of the graphs of the average BST size and height. In this case the insertion and lookup time of a node will be $O(log(n))$, which is quite fast, and taking into account the smaller BST heights when the size of the hashtable is 100000 or more, it only becomes faster. However, since not all the trees are balanced, imbalanced trees do exist amongst the trees of the hashtable, and an example of one is below.

example of imbalanced binary tree:



In the case of imbalanced trees, meaning that the difference being the height of the left and right subtree rooted at the root of the tree (in this case the root is '1') is greater than 1. For imbalanced trees, the insertion and lookup time of a node will be $O(n)$, which is noticeably slower than the $O(log(n))$ time of balanced trees.

3.5 Hashtable size Against Average Branch Traversal



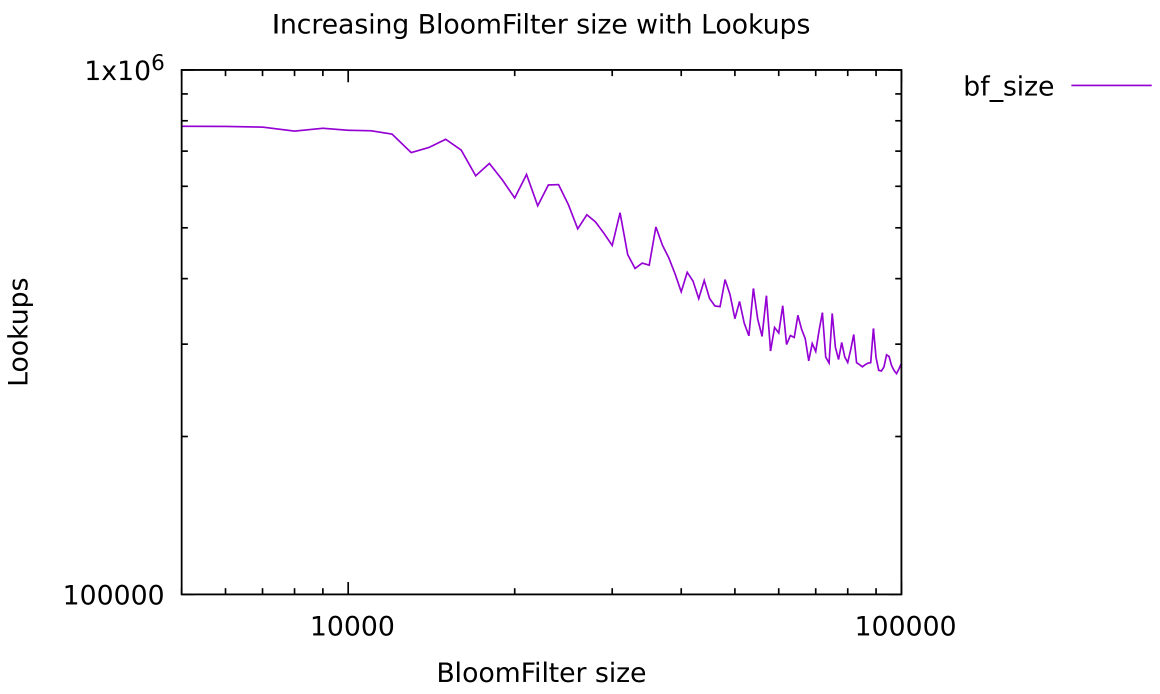
The graph above demonstrates how the average branches traversed in the hashtable as the size of the hashtable increases. And as seen in the graph, the average branches traversed decrease as the size of the hashtable increases.

As mentioned previously the average branches traversed is calculated as $\frac{branches}{lookups}$. What this tells us is a fair idea of the average amount of left or right traversals made when inserting or looking up a node of a BST inside of the hashtable. So as a result, assuming we are using a default hashtable size of 2^{16} or 65536, it only takes 0.1 branch traversals for the insertion and lookup operations of the hashtable.

One may wonder why the average amount of branches traversed is less than 1. This is the case since when the hashtable size is large (such as 65536), many of the BST's have a size of 1 as displayed in subsection **3.3**. Thus, there is no need to traverse left or right since the binary tree only consists of 1 node, the root node. This not only reduces the probability of hash collisions as previously discussed, but it also speeds up the process of looking up and inserting nodes to the BST's of the hashtable.

And just like the previous 2 graphs, this graph also exhibits visible oscillations. This is since as discussed in subsection **3.4** the BST's in the hashtable are not guaranteed to be balanced. Hence, the insertion and lookup time will be longer which means more traversing on average.

3.6 Bloomfilter Size Against Lookups



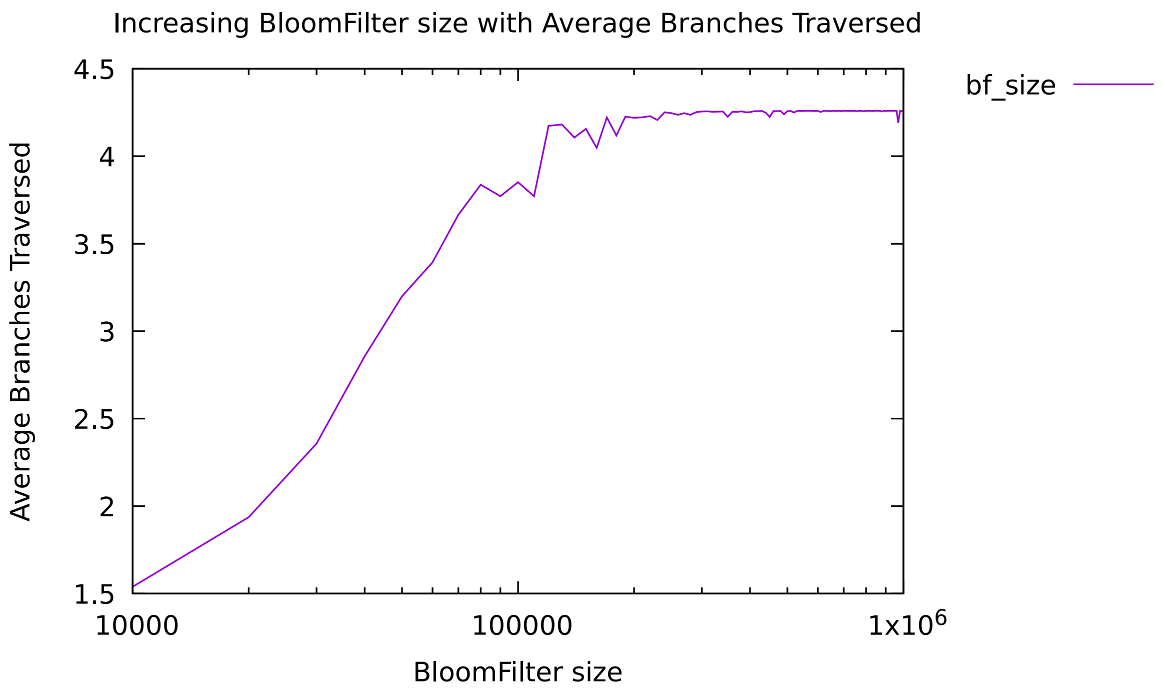
This next graph displays how the number of lookups changes as the bloomfilter size is increased. And as seen in the graph above, the lookups decrease as the bloomfilter size is increased.

This specific graph is also reading a file containing the entirety of the bible (bible.txt), so the lookups in this scenario is larger as expected. So, to account for such a large file the range of bloomfilter size displayed above shows where the most noticeable changes of lookups occur.

Having experimented with how the number of lookups change as the bloomfilter size is changed while the hashtable size is constant (default size of 2^{16}), it has become apparent that the bloomfilter load is a big factor in how lookups perform.

Since the bloomfilter data structure is probabilistic in that it can generate false positives, if the size is something extremely small such as 1 while dealing with a file as large as the bible, an enormous amount of false positives will be triggered. Hence, the reason that more lookups occurred is since the bloomfilter reports a certain word exists (when it doesn't) which makes the program think it is in hashtable when it doesn't exist.

3.7 Bloomfilter Size Against Average Branches Traversed



The graph above displays the average branches traversed as the size of the bloomfilter is increased. And similar to the previous graph this also reading a file containing the bible (bible.txt).

Different from all previous graphs, this one doesn't appear to have a downward trend, but it is not necessarily a bad thing. As observed by the previous graph, increasing the size of the bloomfilter receives less lookups, and since the hashtable size in the example above is constant (default size of 2^{16}), that means the number of branches are bound to also be relatively constant.

The graph more so like the previous graph is representing that the number of lookups in comparison to the number of branches is always decreasing (on average). Hence, we can conclude that if the bloomfilter reports no false positives, then the size of the bloomfilter is not directly correlated to how the number of branches is summed. And this is due to the fact that branches as mentioned previously is calculated as a +1 whenever a right or left traversal is made on a BST. So if the hashtable size is constant, then that must mean the branches summed are bound to always be in a similar given range.

4 Conclusion

After analyzing the time complexities of various BST operations, namely insert and lookup (also known as find) and the graphs, it has been made evident that the size of the hashtable plays a major role in the efficiency speed of said BST operations. The sizes of the hashtable and bloomfilter also determine the probability of hash collisions in the case of the hashtable and false positives in the case of the bloomfilter. Both events which are undesirable, and as observed by the graphs, we have reached the conclusion that increasing the size of the two data structures will greatly reduce the probability of hash collisions and false positives. This was done by providing more room for the data to be stored than available data. By doing so, the data being stored is never cramped and will have plenty of room to be dispersed among the other data.

So what necessarily did we gain from examining the two data structures: the bloomfilter and hashtable? It has been made evident that important factors (namely size) effect the load, average BST sizes/heights, and the average branches traversed. And in turn, the load represents how full the data structure is which gives us an idea of the chance of hash collisions and reports of false positives as previously mentioned. All of which are interconnected and will determine the efficiency and speed of inserting and looking up nodes in the hashtable.

Being aware of the cases of hash collisions and false positives among the two data structures, and how they are avoided by default by the program is important to understand. Being aware of the why behind default large sizes of the hashtable and bloomfilter gives us as computer scientists insight to why a data structure operates the way it does. And since both data structures are somewhat probabilistic in nature, avoiding the bad scenarios as often as possible is of high priority.