Randomized Algorithms

Rafal Szymanski

February 21, 2011

1 Introduction

The assignment asked to analyse three data structures: A skip list, a bloom filter, and a randomized binary search tree. The skip list and the randomized binary search tree are containers that theoretically exhibit $O(log_2 n)$ performance, while the bloom filter allows for $O(1)$ performance for all the major operations.

While the skip list and the search tree are guaranteed to be correct, the bloom filter allows for false positives. Additionally, the current implementation also allows for false negatives, which is actually not ideal. A more robust method would have been to implement a counting bloom filter, but then we have the space tradeoff of having to store so much additional data per bit, which defeats the purpose bloom filters which are meant to be highly compact.

The add, find, and delete methods for RBST and skip list are all recursive, while all the operations for the bloom filter are done in $O(1)$.

What follows is a short analysis of three containers and their respective add, find, and delete methods.

2 Analysis of addition methods

Addition means simply adding an element to the data structure. The analyse program counts the number of calls (due to the recursive nature of add in RBST and skip list), and plots the results as can be seen in Figure 1.

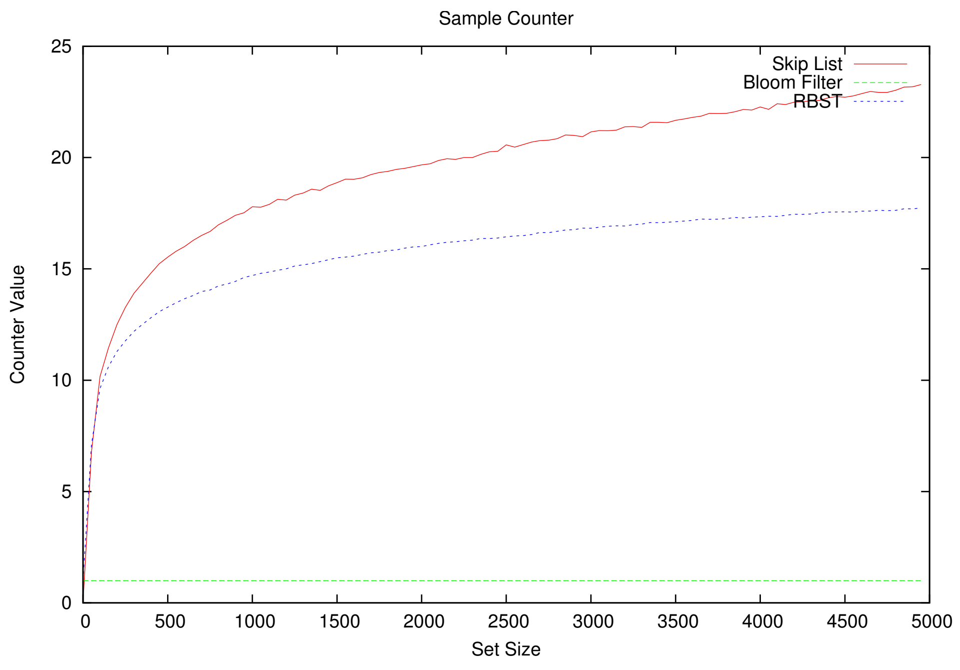

Figure 1: Add counter

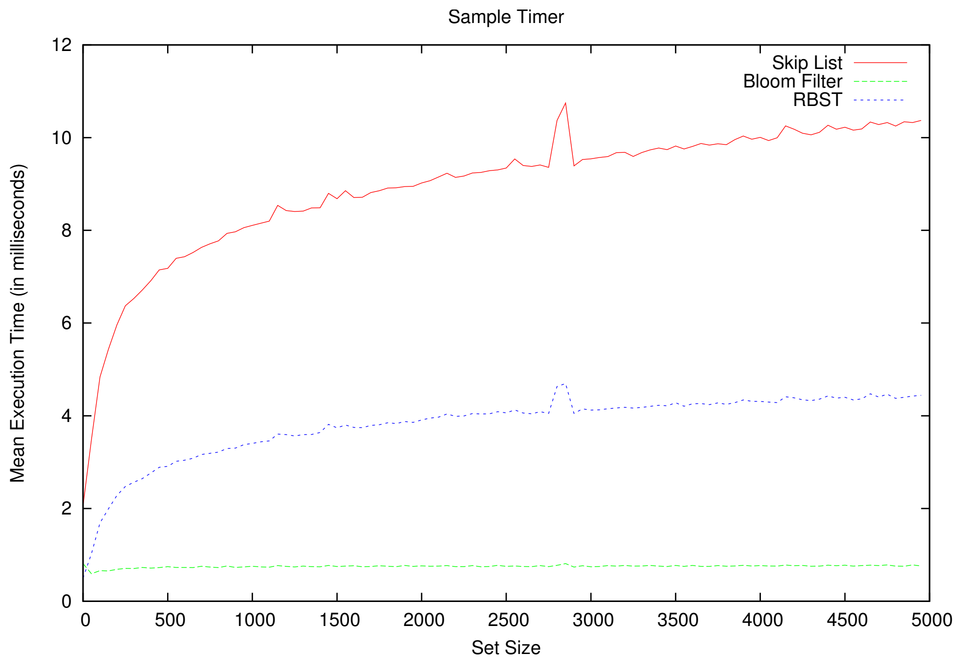

Figure 2: Add timer

Addition should theoretically be similar to both deletion and finding, $O(log_2 n)$. We can see that the plot follows this general complexity.

Unfortunately, the skip list seems to exhibit less and less logarithmic behaviour than the RBST, as can be seen in the following figure.

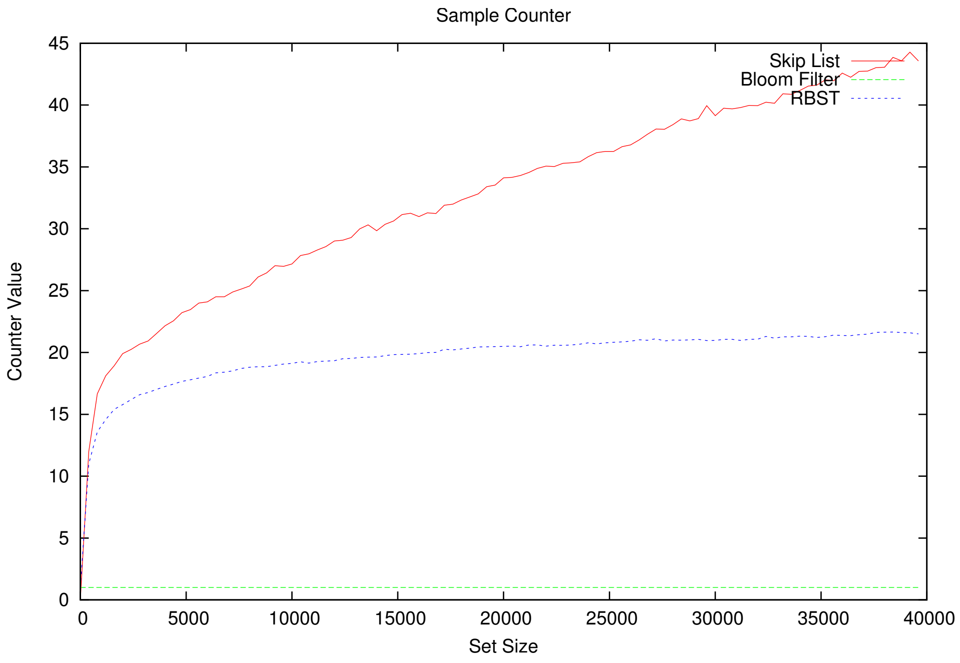

Figure 3: Add counter, 40000

This is most likely due to the fact we are adding more and more nodes at unfortunate positions that require more calls to add. If our skip list doesn't have enough high-level elements, or they are not connected in a nice binary fashion, we will be less and less likely to find the nice logarithmic behaviour, as can be observed in the diagram.

The explanation for the more and more linear behaviour of the skip list as n gets bigger is that our skip list has a max height of 5. After increasing the max height, the performance was improved and approached logarithmic complexity again.

Addition to a Bloom Filter is $O(1)$, non-recursive, it's simply flipping some bits to 1. Nevertheless, a slight increase in execution time can be noticed. This is most likely due to the fact that the bigger the set, the higher numbers we will receive as bits to hash, and hence the CPU will have to spend marginally more time computing the proper pocket and offset within pocket, since arithmetic on larger numbers will take longer than on shorter ones.

3 Analysis of delete methods

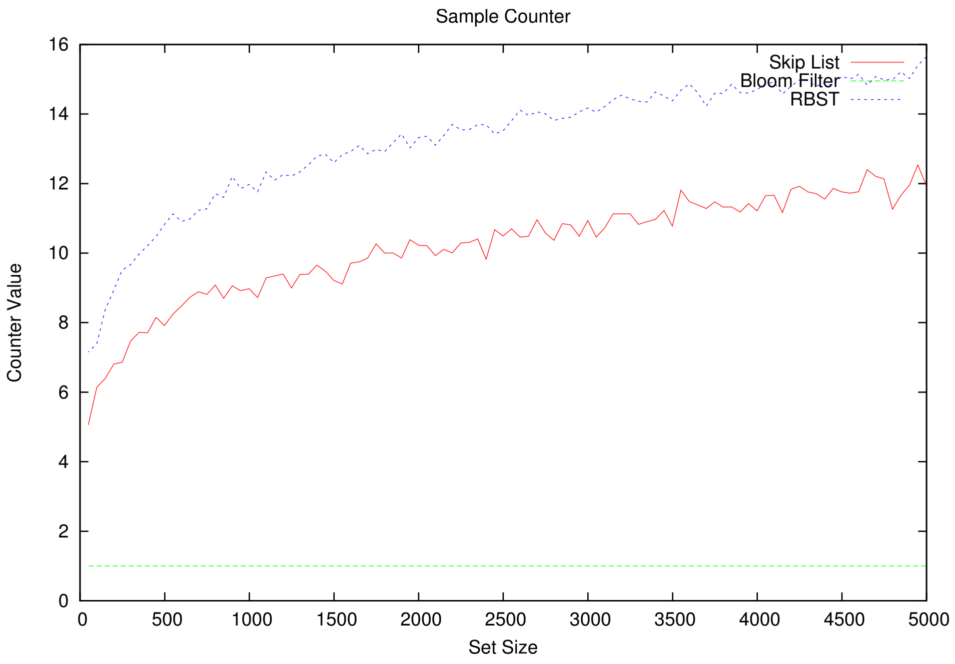

Figure 4: Deletion counter

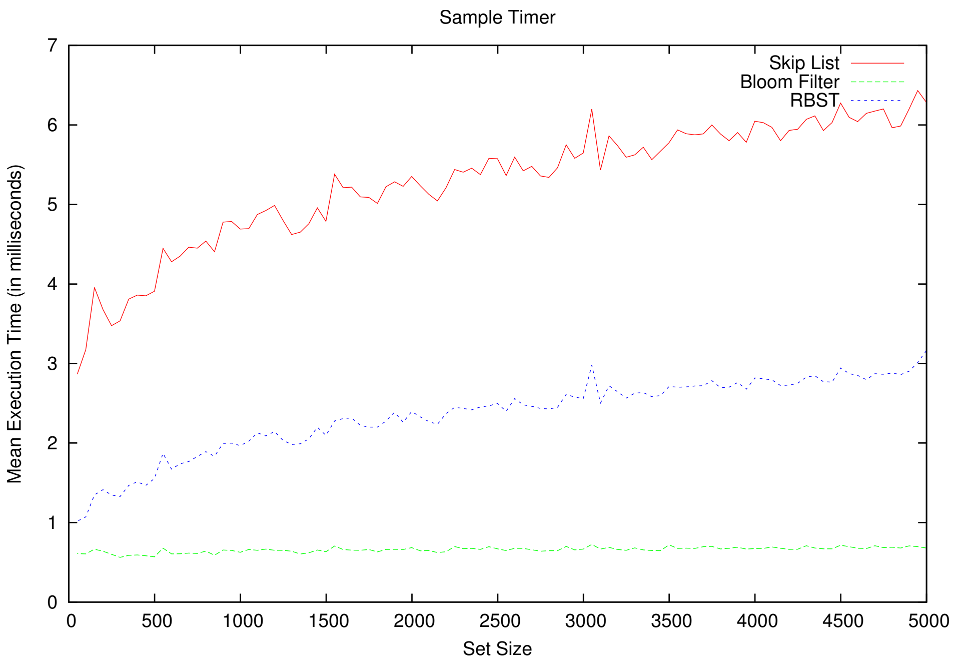

Figure 5: Deletion timer

Deletion on Skip Lists and RBSTs should theoretically be similar to both insertion and finding, $O(log_2 n)$. Looking at Figure 4, we can see that is is quite true for my implementation of the data structures. Nevertheless, let's increase the set size to 35000.

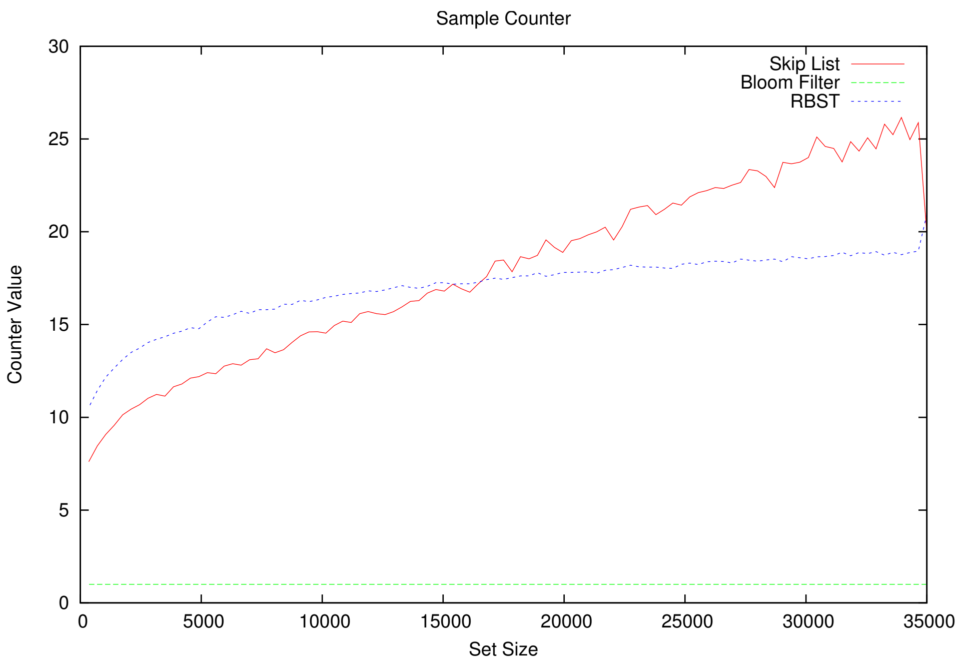

Figure 6: Deletion counter, 35000

We can see something interesting over here. We notice that around set size of 16000, the RBST has less calls than the Skip List and exhibits better complexity. This is due to the same reason as in the addition case.

4 Analysis of find methods

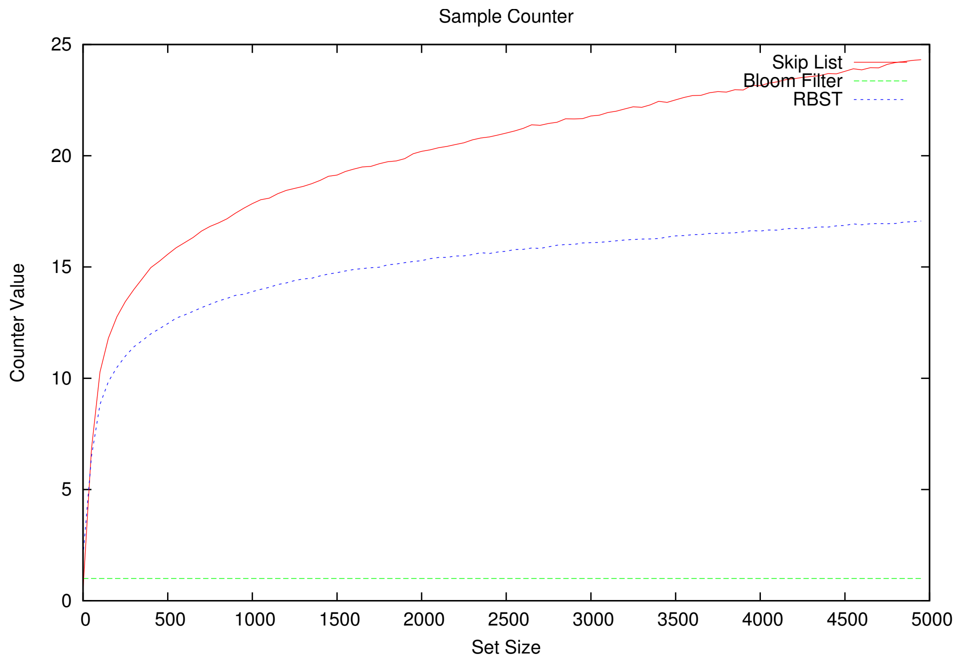

Figure 7: Find counter

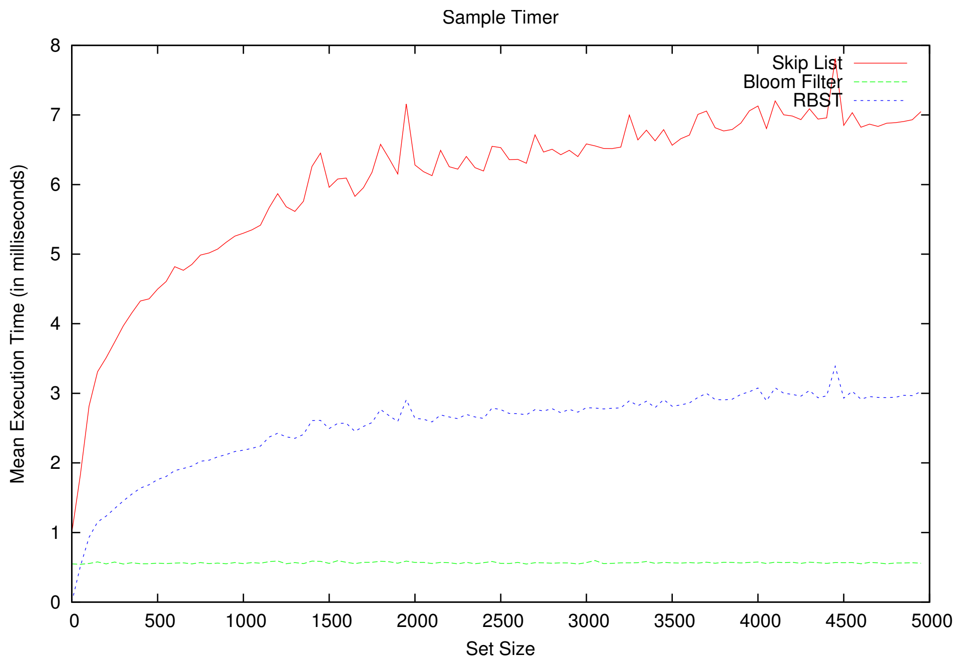

Figure 8: Find timer

Theoretically, finding something in a skip list is $O(log_2 n)$. Figure 7 does portray that finding something in the skip list I implemented is of that order, with some constant multiplier. Unfortunately, the higher n, the skip list is beginning to exhibit more and more linear property, unlike the RBST, which is keeping its logarithmic complexity. For example, look at the following Figure which shows the counter values at input size of 20000.

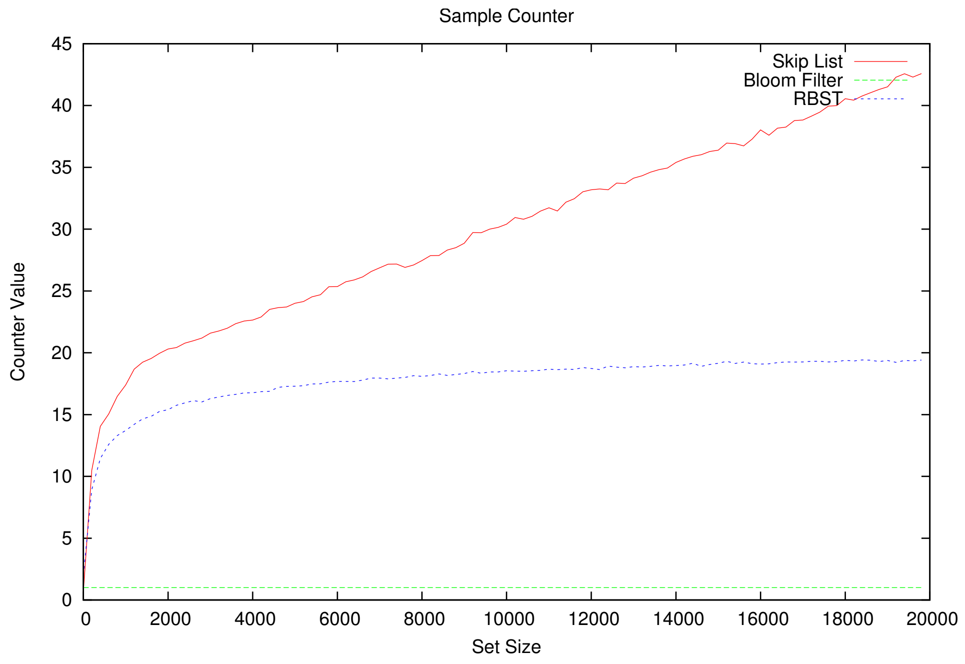

Figure 9: Find counter, 20000

This is again due to the fact that our max-height is 5, as described previously.

As always, the search operation on a Bloom Filter is $O(1)$, non-recursive.

In Figure 5, we can notice some spikes in the execution time. These are most likely caused by OS overhead and context switches, due to the fact that at these points, the points in Figure 7 remain smooth.

5 Summary

In general, the RBST follows its theoretical model and beats Skip List as the input size increases. The larger the size, the more linear the operations on the Skip List seem to become. This is due to a restricted max-height, which means our skip list can't fully exhibit logarithmic behvaviour. In the case the max-height is increased, the complexity of all the operations becomes more logarithmic, but at the cost of memory.

There is not much analysis to be done on the Bloom Filter, since all the operations on it take $O(1)$ time. The problem with the current implementation of the Bloom filter is that it allows for both false negatives and false positives.