Christian Ropchock

EECS 2510 Non-Linear Data Structures

03-23-18

**Binary Search Tree Performance Comparisons Report**

**Introduction**

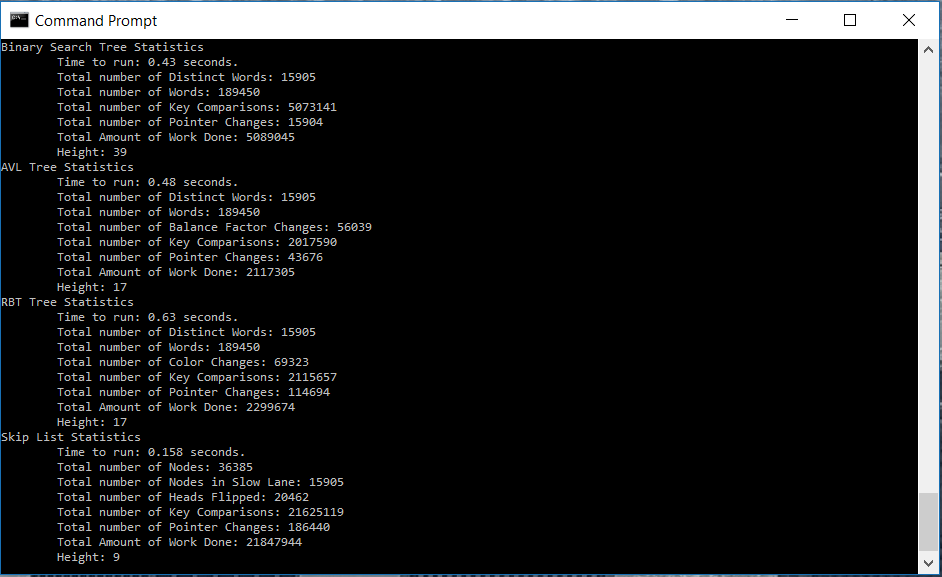
This project involves programming the standard Binary Search Tree, AVL Search Tree, Red-Black Search Tree, and the Skip List Data Structure to compare their performances on inserting nodes into the trees. There are two different types of performance measurements created for each tree in this project; the first method is timing how long the process of creating a tree (Skip Lists are included in this group) takes, and the other method involves measuring the total amount of work it takes to create the tree which involves number of key comparisons to the new node being inserted, the number of child/parent pointer changes, as well as any tree specific changes.

**Expected Outcomes**

In terms of timing results, I expected the AVL Tree to run the fastest, because even though the Binary Search Tree doesn’t have to deal with any rotations or color/BF changes, it does grow into a larger size, which means that it involves longer paths for traversing on average then the AVL and RBT Trees. The Skip List should run the slowest on average, since most of the times, the average case scenario will still be with Big-O of the worst-case scenario. For example, each node in the slow lane has a potential to contain an express lane node, in which the worst performance of the Linked List could show (N) in which every node could potentially be visited in the express lane, before working down the express lanes. The best case can only come out of the Skipped List, if for each lane in the Skip List, the express lane above it has a node for every other node.

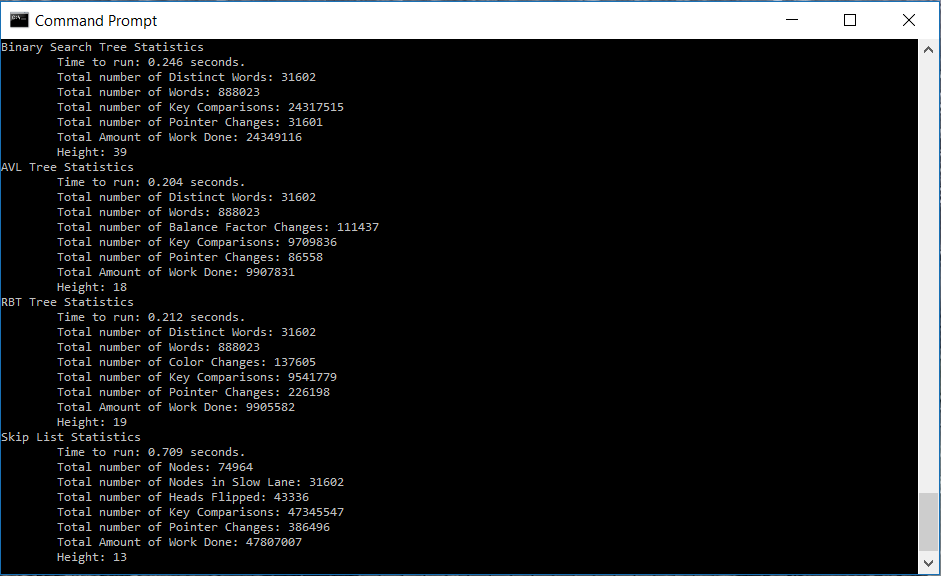
When it comes to overall work done, the most work is expected to come out of the Red-Black tree. The reasoning behind this is that even though the Binary Search Tree contains more levels and AVL performs rotations, the Red-Black tree not only has to traverse every node in a (lgN) fashion, but to rebalance the Red-Black tree it has to potentially perform color changes as well as rotations all the way from the insertion point all the way up back to the root. The AVL tree only must perform a rotation and BF changes at the node where it went out of balance (went to either +/- 2). The Skip List in its best performance does a (lgN) search to get to the insertion point, but then it must rely on probability to insert new nodes above the inserted node, which has a more likely chance to create more nodes then what is needed for an ideal Skip-List.

**Results**

****

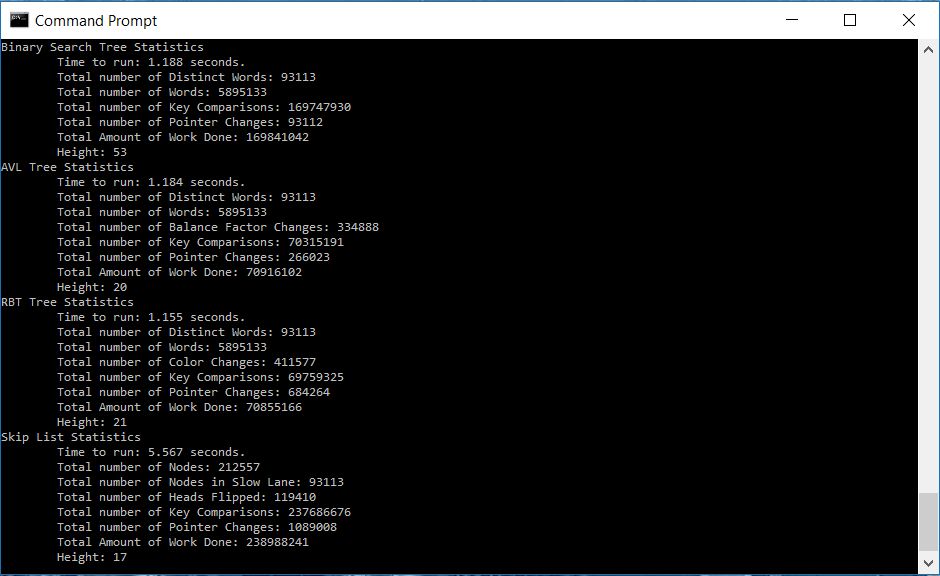
**Figure 1. Search Tree and Skip List Analysis on 1.057MB File**

As seen in Figure 1 above, when it comes to a smaller file, the Skip List outperforms the three Binary Search Trees in terms of timing. The reason for the Skip List outperforming the trees, is that there is more work for the Skip List to perform then the trees as the height gets larger from the potential of having to do more than one check at each level, and since the Skip List is still at a relatively low height, there aren’t any large columns for the Skip List to have to traverse yet. But, as one can see in the total work load done, there is more work done in the Skip List then the Binary Search Trees, so there is potential for this work load to start to reflect in the timing as the files get larger. The same also applies to the Binary Search Trees, the total amount of work hasn’t had much affect on the time for the Binary Search Tree yet due to the smaller file size, which is why it ran faster then the AVL and RBT for the smaller text size.

****

**Figure 2. Search Tree and Skip List Analysis on 4.936MB File**

Now that the file size had a chance to increase, one can see in Figure 2 the difference in the timing now that the Skip List’s height has increased, and more words must be processed through the higher columns. The amount of work that the Skip List must do has now grown to 2x that of the BST, due to the number of nodes being larger then that of the BST. The reason why the BST is starting to take longer and doing more work then the AVL and RBT is, because the BST is beginning to grow taller then the other two Binary Search Trees.

****

**Figure 3. Search Tree and Skip List Analysis on 31.837MB File**

The true nature of the differences between the four trees can be observed in Figure 3. Now that the file size is exceptionally larger, the amount of work that each tree must perform is beginning to reflect in the run times of each tree. The Skip List has a total work load of 238,988,241 which includes: the number of heads flipped (new nodes created in the express lane), number of key comparisons, and the number of pointer changes. The next higher work load is the Binary Search Tree at 169,841,042 which is because the BST has a significantly higher height then the other three trees, so the BST will visit more nodes on average…

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Binary Search Tree | AVL Tree | RBT Tree | Skip List |
| Number of Words = N | 5895133 | 5895133 | 5895133 | 5895133 |
| Number of Levels for Worst Case | **N**  = 5895133 | **1.44\*lg(N)**  ≈ 33 | **2\*lg(N)**  ≈ 45 | ∞ |
| Levels from Testing | 53 | 20 | 21 | 17 |

**Figure 4. Testing Heights vs. Probabilistic Heights (Tested on 31.837MB File)**

As one can see, even though the worst-case scenario for the heights of some of the trees can be exceptionally high, they tend to “stray” away from the extreme worst-case scenarios. Even though the Skip List has the potential to infinitely flip heads, statistically there is a lg(N) chance of a new lane, where N is the number of nodes in the slow lane, because the average-case tends to favor a 50/50 outcome.

For Pugh’s claim, the Skip List did run faster then the Binary Search Trees for smaller text file, but as the files began to get exceptionally large, the Skip List’s overhead for the number of express lanes began to slow down it’s performance. So, his claim is only true for smaller binary/text files, but the Skip List for the most part didn’t begin to overwhelmingly slow down with the large text files.

In terms of the run times,

**Figure 5. Line of Best Fit (Scatter Plot Representation) of number of words in text file vs the time it takes to insert into trees**

As shown in Figure 5 above, the amount of time for the 3 Binary Search Trees would roughly around the same time, and the Skip List performed the best at first, but there was a point in which the Skip List intercepted the other three Binary Search Trees and began to slow down in relative performance. My expected outcome was for the Skip List to perform the slowest, in which it does in the above figure. As one can also see, in some of the lines of best fit, there were some points that dipped below the line, this shows that the relative ordering of the words that comes in can also impact the timed performance of the tree, since all the trees performed better on pieces of text around the 1,000,000 – 2,000,000 word range, which is where I had Shakespeare’s text in my text files.

**Figure 6. Line of Best Fit (Scatter Plot Representation with Forward Forecasting) of amount of work performed for each tree compared to time to insert into each tree**

Even though on average, the performance of the AVL and RBT is better in terms of timing performance for larger text files, as one can see in the above plot, there is a trade off in which the AVL and RB Trees also have to form more work, in which AVL performed the most work in doing key comparisons, pointer changes, as well as Balance Factor changes. I originally expected the Red-Black tree to perform the most work, but there was more work put into doing rotations and Balance Factor changes on average for an AVL Tree then the rotation and color changes for the RB Tree.