Lab 6 – Red-Black Trees

Submitted by: Adam Mak, Bhavna Pereira & Hamrish Saravanakumar

March 05, 2021

Height of a red-black Tree

Although the lab did not explicitly state to explain the height of a red-black tree, we found that by proving its height in relation to a BST, we got a much better understanding of the time complexity of the insertion methods for this data structure. Tt is known that a red-black tree is essentially a special case of a binary tree. For a general binary tree, where the minimum number of nodes on all root to null paths is equal to greater than or equal to log(n+1) with base 2. For example, if the minimum number of nodes that was described earlier was equal to 3, there would have to be at least 7 nodes. From here, the properties of the red-black tree allow us to state that there is a path from the root node to the leaf node with a maximum of log2(n+1) black nodes AND that the number of black nodes is at least n/2. Therefore, with n nodes, it can be concluded that the red-black tree would have a height no more than 2*log2(n+1).

Analysis: Insertion of numbers from 1 – 10000

It is important to note that this proof was not to indicate what the approximate height would be, but rather what the upper bounds would be based on the properties of a red-black tree. Using our implementation of the RBT and running an identical test to the one described in the question, we got a height of 24 regardless of how many times we ran the test. This led us to believe that there had to be an issue with either the insert method or in the test ran in order for a height of 16 to occur. The implementation we created ensured that all 4 properties of a normal red-black tree were upheld. Every node had to either be red or black where the root node was always black, there did not exist two adjacent red nodes, and every path from a node to its descendant has the same number of black nodes. The failure to uphold one of these properties may have induced variability in the construction of the RBT, and thus resulting in a height of 16 to be returned.

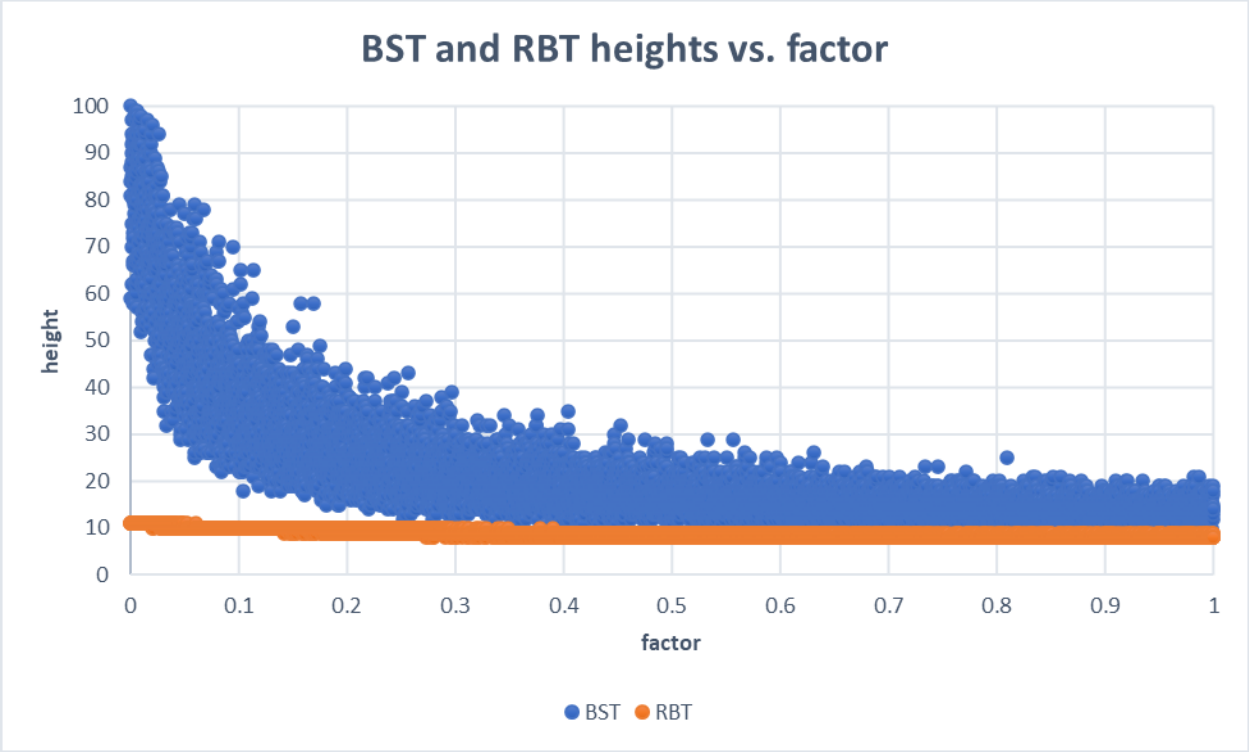
Average Height Difference: RBT vs BST

A test was conducted to compare the average height of a BST and RBT implementation when the same list of random elements of length 10,000 were inserted. The function trees_random_height(n) returned the average distance of the heights of the 2 data structures over an "n" number of trials, which we set to 100 in order to get a relatively large pool of data.

The height of the red-black tree and a regular BST were discussed above. In reference to this proof, there would not be a case where the height of the BST is outright smaller than that of the RBT. For this reason, we decided to examine runtime of the insertion methods of both. The average runtime for both structures is expected to be O(logN). For RBTs, in theory, there would be more time taken for some insertions because the tree needs additional time for the rebalancing in comparison to a BST, but it would still uphold the O(logN) runtime. However, there are cases for BSTs where the insertion time can in fact be O(n) in the worst. This would be seen when inserting elements in either increasing or decreasing order. Therefore, the BST would potentially be beneficial in a scenario where it is known that there will be a significant number of insertions in a randomized order, and there isn't the need to do other operations that rely on the height of the tree, such as searching.

Near-sorted List Performance: RBT vs BST

We decided to test the BST and RBT heights vs. the near sorted factor of a list of length 100. Although the specifications were to test with lists of length 10000, the BST height() method encountered recursion errors with near sorted lists of lengths greater than 996. Since the BST's heights were spread out, we incremented the factor by 0.0001 for each iteration to get the best representation.



The BST height is consistently greater than the RBT height, even for fully randomized lists. Also, while RBT heights are constant with respects to the factor, BST heights are dependent. This is because for lists that are almost sorted, the BST gets closer to becoming a linked list. In the worst-case where the list is fully sorted, then the height of BST is exactly the length of the list. As the factor grows, the BST height decreases logarithmically, but is never less than the RBT height.