TSBBST

CP-3

Robert Boerwinkle

Spring 2023

1 The Start: Binary Basics, Sources, Trees

Binary trees are a data structure akin to a linked list. They have nodes with values that are linked together with memory references. Instead of having just 1 forward link, each node has 2 children (like a linked list, they can also have links to parents). These children, typically named "left" and "right", indicate relative value. Any node to the left will always be smaller than the parent node (and right is larger). This means a very efficient binary search for values in the tree.

This project will look at 3 variations of trees: custom implementations of a basic binary search tree and a balanced tree, and a pre-existing implementation of a balanced tree. All trees are implemented in Python. Wikipedia has good articles on binary search trees: https://wikipedia.org/wiki/Binary_search_tree and (spoilers) the balanced trees: https://wikipedia.org/wiki/AVL_tree.

2 Truly Simplistic Basic Binary Search Trees

All of these trees are based on this binary tree data structure. The basic operation is finding a value in the tree. All of the time complexity for insertion and deletion comes from the traversal. Traversal starts at the root node. At each node, if the target value is larger then the node, the right path is taken. If the target is lower than the current node, the left path is taken.

This structure alone isn't super useful. In technicality, this data type has O(n) worst case time complexity for lookups (adding and removing only have a constant time increase on top of lookup). Imagine adding elements in sequential order. This would result in a tree where every node is to the right of the previous node. At this point, traversing the tree is like traversing a linked list. To correct for this, the tree has to remain balanced. The "balance factor" of each node is the difference in depth of the left and right children. If that difference becomes more than 1 in either direction, the tree must be rotated.

3 The Self Balancing Binary Search Tree

When a node gets unbalanced, it needs to be rotated. In Figure 1, if subtrees A or B are higher than C, then node X is unbalanced and needs to be rotated to the right. After every insertion, up to 2 rotations need to be done.

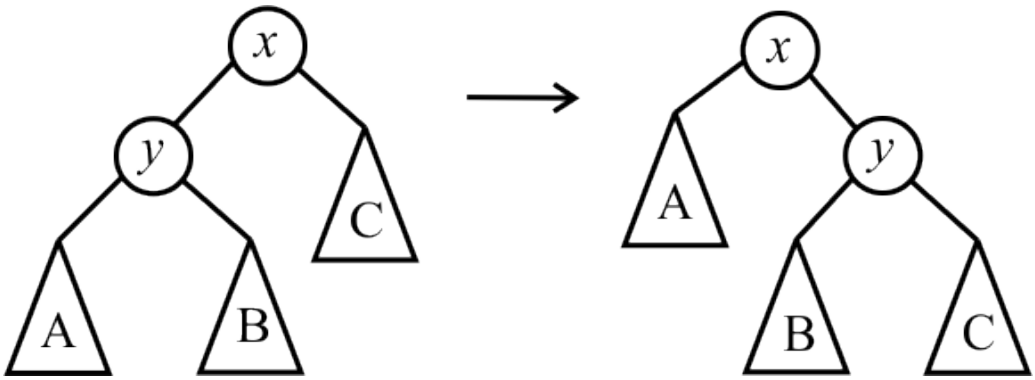

Figure 1: A right rotation of node X (http://www.cs.ecu.edu/karl/2530/fall18/Notes/lec43.html)

This particular type of balancing tree is called an AVL tree. The custom implementation only rebalances on insertion. Some, like the chosen pre-existing implementation (https://github.com/mozman/bintrees), rebalance on removal as well. Because the height is kept lower, the traversal time is kept to a minimum. The trees were tested on a random list of non-repeating integers.

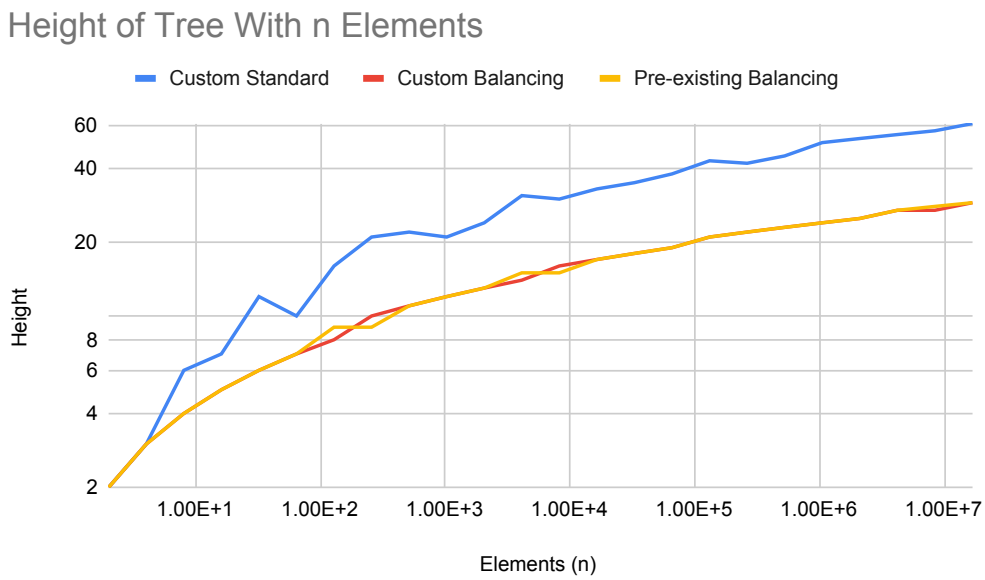

Figure 2: Heights of balanced and unbalanced trees

4 The Sendoff: Bidding Bye; So Tragic

After these improvements, the time taken for the various operations is considerably improved. The time complexity is (experimentally) the same, though. The time to add 2^{24} elements was 391s for the unbalanced, 254s for the balanced, and 295s for the pre-existing balanced. The time cost difference between the custom and the pre-existing is probably because of extra infrastructure. The pre-existing tree can store values at each of the keys and is doubly linked (nodes' parents are stored). This does not affect the height, though. Overall, both custom trees behaved satisfactorily.

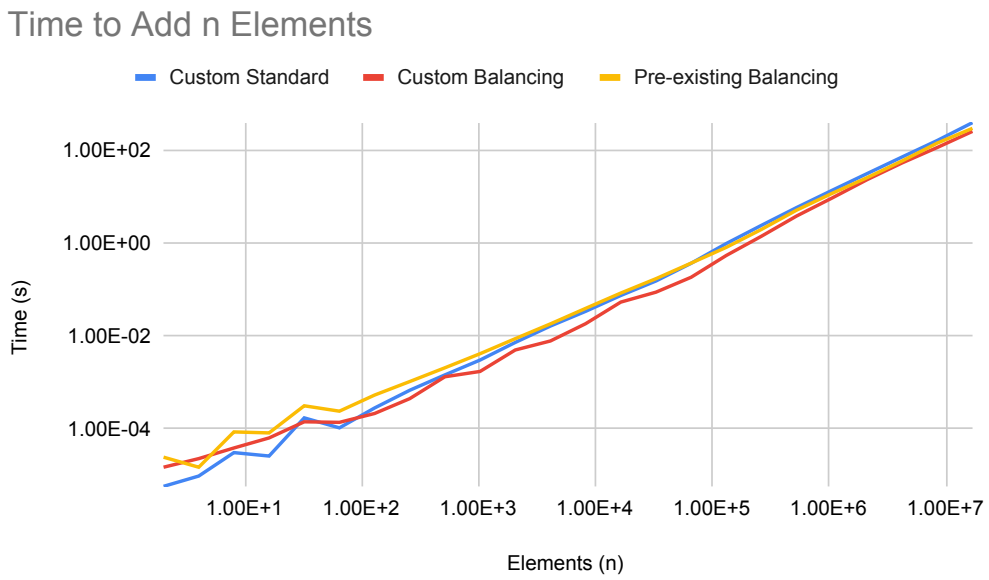

Figure 3: Adding n elements (log scale)

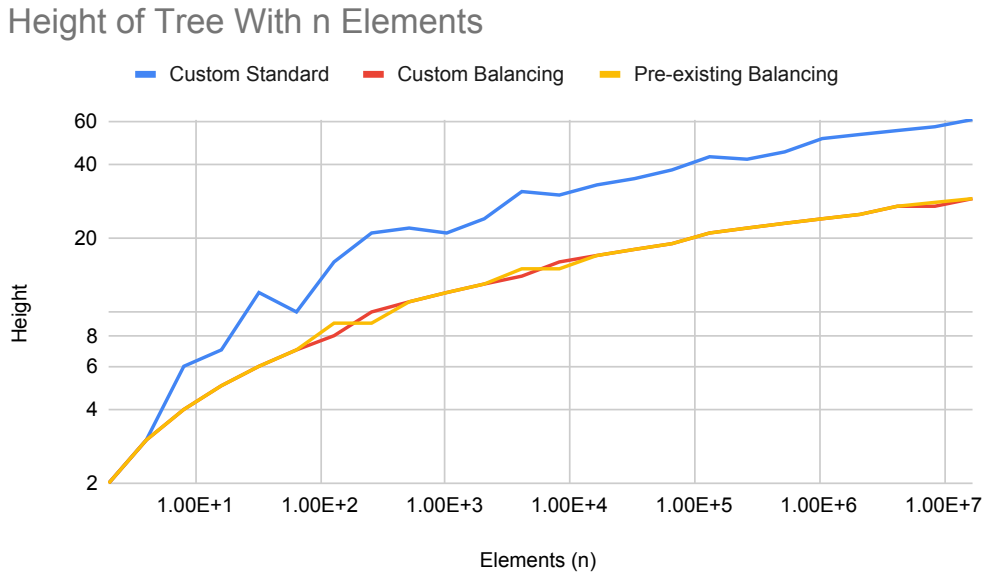

Figure 4: Removing n elements (log scale)