Project 3 - 11/14/2017
Gavin Kyte

For this project that requires a binary tree type class that compares two lists, I chose to implement an AVL tree with a Node and Pointer style. Additionally, I used a single BST for the entire dataset, rather than two separate data structures. First I will explain my choice to use an AVL tree. Then I will explain my choice of Nodes and Pointers. Then I will explain how and why I used one tree to hold the data set.

Contrary to most, I chose the AVL tree because it is more difficult to implement, and because it is faster to insert in the case of mostly sorted data. Granted I would support using a normal BST because inserting is faster without rotations involved. However, the case of sorted data being used convinced me to not use a BST. I suspect that the insert cost of the AVL can outweigh the benefit of balance offered in an average case. In sum, inserting in BST is simpler, and arguably faster than AVL for certain data, but I the AVL will perform better in the worst case.

Furthermore, I chose to use Nodes and Pointers rather than a linked list, vector, or other array-esk structure because it intuitively made the most sense to implement, my implementation requires data be stored about each value which hints at using nodes. Since nodes are involved, it seemed strange to have a vector full of nodes so I opted for pointers, but I assume the former would also work. Further, my tree may be balanced, but it is not full, meaning I would not be able to directly apply known mathematical assumptions about child and parent indexing, but rather would need to make accommodations to make them valid, which seemed excessive. Last, I used Nodes because storing information such as height allows for a better AVL tree, and because of my choice to use one tree.

Finally, I used a single tree by tracking which datum came from which file inside every Node. By this I mean that every unique value in both files is represented once and only once inside my tree, but within each representation, i.e. Node, there is a `typeCounter` which increments index 0, or 1 per whether the data being read is from file 1 or 2 respectively. This single tree implementation makes it possible to traverse the tree once and print out where every string object belongs. Practically, it will take 3 sweeps to display this in the requested format. Despite the repetition here, 3n is vastly superior to the nlog(n) solution of searching tree B for every element in tree A twice (once for A-B, once for A∩B), and vise versa (for B-A) to achieve the same results.

Results from speed tests (Averaged 20 test results on very large file):

Operation	Time (microseconds)
AVL.loadData()	11
AVL.compareLists()	53
AVL.loadData()	11
Norm.compareLists()	54

Not sure how accurate the results are as figures, but, the pattern outside of the listed results is that the two are very similar, but the normal BST is slower on average. Additionally the compareLists() prints to stdout, while the load does not, which explains the logically slower operation being faster (load should be the slowest but isn't because of cout).