Holt Crews, 10/11/17, Section: 6:30pm

testfile4.txt contents:

a ab abc abcd abcde abcdef abcdefg abcdefgh abcdefghi abcdefghij abcdefghijk abcdefghijkl abcdefghijklm abcdefghijklmn abcdefghijklmno abcdefghijklmnop abcdefghijklmnopq abcdefghijklmnopqr abcdefghiklmnopqrs abcdefghijklmnopqrst abcdefghijklmnopqrstu abcdefghijklmnopqrstuv abcdefghijklmnopqrstuvw abcdefghijklmnopqrstuvwx abcdefghijklmnopqrstuvwxy abcdefghijklmnopqrstuvwxyz

This text file is a good example of when a binary search tree would be inefficient and similar to a linked list. Each element in the file is larger than the previous element. As these elements are read and then placed into the tree the element is always inserted as the right child of the deepest node. This essentially creates a linked list of the elements, an inefficient structure when compared to a tree. An AVL tree on the other hand would rearrange the tree in a much more balanced way, thus cutting down on the find time. A find on the resulting AVL tree in general took fewer moves through the tree than the binary search.

testfile1.txt test: When “thinking” is searched in the BST there are a total of 5 links followed, however, with the AVL, only 2 links are followed. The BST average node depth is: 3.16 and the AVL average node depth is: 2.26

testfile2.txt test: When “mauve” is searched in the BST there are a total of 9 links followed, however, with the AVL, only 1 link is followed. The BST average node depth is: 6.063 and the AVL average node depth is: 2.5

testfile3.txt test: When “like” is searched in the BST there are a total of 4 links followed, however with the AVL, only 3 links are followed. The BST average node depth is: 3.23 and the AVL average node depth is: 2.23

testfile4.txt test: When “abcdefghi” is searched in the BST there are a total of 8 links followed, however, with the AVL, only 4 links are followed. The BST average node depth is: 12.5 and the AVL average node depth is: 3

An AVL tree is preferable to a BST whenever there is a chance of the tree becoming imbalanced. An AVL tree restructures itself when it detects an imbalance, evening out the spread of the nodes across all branches of the tree. If some of the data has the potential to be sorted when it is inserted into the tree then it could make the resulting tree imbalanced. These imbalances lead to inefficiencies when search through a tree because the find function now has to iterate through more branches and nodes in the tree than theoretically necessary. An AVL tree however does a good job of maintaining the balance in a tree and thus cutting down on the number of operations required of the find function. Basically any time that there is a substantial amount of random-ish data being put into a tree, an AVL tree will work better in the long run by cutting down on the time for the find function to run.

However, an AVL tree does have a few caveats. AVL trees are much harder to implement than BSTs because rotation operations have to be created to perform when the tree starts to become imbalanced. When the tree is imbalanced, it must be restructured in a particular way in order to maintain the integrity of the tree while also eliminating the imbalance. Thus, it is a good bit harder to code and takes more physical lines of code than a BST. Also, there is a problem with the time it takes for the tree to be created. Whenever the imbalance arises, the rotation functions have to be called in order to fix the tree. This naturally takes more

time and so the creation of an AVL tree is generally slower than that of a BST. Another caveat is that an AVL tree requires that a height must be stored for each node so that it can calculate the imbalance in the tree while a BST does not necessarily need this. This means that the AVL tree will take up more space than the BST.