Adarsh Solanki As5nr
3/1/12
2150-102

textfile4: aaa bbb ccc ddd eee fff ggg hhh iii jjj kkk lll mmm nnn ooo ppp qqq rrr sss ttt uuu vvv www xxx yyy zzz

BST:
Left links followed = 0
Right links followed = 0
Total number of nodes = 26
**<u>Avg. node depth = 12.5</u>**

AVL Tree:
Left links followed = 0
Right links followed = 0
Total number of nodes = 26
Single Rotations = 21
Double Rotations = 0
**<u>Avg. node depth = 3</u>**

Enter word to lookup > ddd
BST:
Left links followed = 0
Right links followed = 3
Total number of nodes = 26
Avg. node depth = 12.5

AVL Tree:
Left links followed = 2
Right links followed = 0
Total number of nodes = 26
Single Rotations = 21
Double Rotations = 0
Internal Path Length: 78
Avg. node depth = 3

The reason that there is a disparity between the results of an AVL-tree and a regular BST is because of the alphabetical order. Without reordering, a BST ends up with a very poor balance, so the amortized time or reordering the nodes (implemented in the AVL tree) makes the tree much more efficient. This is apparent from the average node depth which is a representation of the average look-up time.

To discuss when AVL trees are preferable to BSTs, it is important to first specify the differences in performances. An AVL tree sacrifices insertion/deletion time by adding the extra optional step of reordering. Through double and single rotations, an AVL tree can assure a tree with all children of every node differing in height by one at the most. This leads to a look-up time closest to O(log(n)).

This sacrifice in insertion/deletion is in order to optimize future look-ups to the data structure. This means an AVL tree is preferable when look-up time is of the utmost importance. If a data structrure is to be prepared before hand and then accessed in real-time, an AVL tree is ideal. Also, if there are many repeated insertions and deletions, an AVL tree is not altered, it remains balanced, whereas a regular BST will trend towards imbalance based on the remove() function's tendency to remove more nodes from a certain side (left or right, depends on implementation, can be randomized to minimize this factor).

In conclusion, an AVL tree is a very useful data structure that can retrieve data faster than most other tree implementations.  When the insert/delete time is not of utmost importance in the measure of performance, an AVL tree can be ideal due to its strict balancing rules and depth-optimization.