Robert Alexander
RMA3MZ
10/9/13

In-Lab 4 AVL vs. BST

My testfile4.txt:

"apples baked carefully dont ever forget great happiness if john knows like making nice objects per quick requests sent towards underlings voting"

To compare the AVL tree implementation with the BST implementation, I chose a list of words that increases in value by each word (a<b<c<d…). This meant that the binary search tree would place all of the nodes in one long path (with each node, or word, having only one child) as opposed to creating a nicely balanced tree. I then chose to have the machine look for the word "voting" and it looked through only 4 right links for the AVL tree, but it took 21 right link searches for the BST. This clearly is much less efficient than the AVL tree, which will sort them in a more balanced manner (the BST has a much larger average node depth when compared to that of the AVL tree). It appears as though AVL trees are much better in circumstances where you are reading information as opposed to writing it. It doesn't cost too much to gain this increased performance, as I note in my testFile4 observations below.

Observations

- testFile1
  - BST and AVL take in 19 words but add in 17 nodes. This is explained by the nature of these trees: they cannot contain duplicate nodes.
  - AVL tree had 6 single rotations and 2 double rotations for an average node length of 4, while BST had an average of 6.
  - Search for "same"
    - BST left links followed: 2, right links followed 2
    - AVL left links followed: 1 right links followed 2
    - AVL was slightly faster
- testFile2
  - 16 words, BST/AVL create 16 nodes (no repeated words)
  - AVL had 9 single rotations for an average node length of 4, while BST had 11
  - Search for "in"
    - BST followed 8 right links
    - AVL followed 1 left link and 1 right link, clearly faster/more efficient
- testFile3
  - 13 words, BST/AVL create 13 nodes
  - AVL made 5 single rotations and 2 double rotations for an average node length of 3, while BST had an average of 5
  - Search for "orange"
    - BST followed 2 left links and 2 right links
    - AVL followed 1 left link and 2 right links, slightly more efficient

- testFile4
    - 22 words (increasing in alphabetic order as the words progress), 22 nodes in each tree.
    - AVL made 17 single rotations for an average node depth of 4, while BST had an average of 20.
    - Search for "voting"
        - BST followed 21 right links, very inefficient
        - AVL followed4 right links, clearly much more efficient in this case

Post-Lab

After performing multiple experiments on AVL and unbalanced binary search trees, it is quite clear that the AVL tree comes out on top in a few circumstances. For example, the worst case of an unsorted binary search tree is that the number of nodes is the same as the height of the tree (a situation that I created in my testfile4, where I made each successive word greater than the previous). The worst case scenario for find on an AVL tree is O(log N) as opposed to O(N) for a binary search tree. Since the binary search tree is unsorted, it just keeps adding each successive element (in the case of my test4) onto one side of the tree, while the AVL tree quickly rotates nodes around to create a more efficient and sorted tree, with a much smaller average node length. If you end goal is finding (or reading) elements in a tree, then in most cases it would appear as though an AVL tree would be preferable, since it will balance itself and doesn't require elements to be sorted before-hand.

While AVL trees appear to be the best implementation for a situation where you need to mostly read from the tree (as opposed to writing onto the tree), there are some costs involved with creating this type of tree. The main costs appear

whenever you modify an AVL tree. Since the tree is inherently balanced, each time you insert or delete you must shuffle the tree's nodes to get an equal balance, or else it would not be an AVL tree. If you need to call insert or remove far more often than you need to find elements in the tree, then a BST might be the better implementation (or other trees that we have not discussed in this lab).

In summary, it is clear that in the case when you are mostly reading and not doing much writing on a tree, the clear winner is the AVL tree, since finding in this tree is much faster than a BST. However, when doing more writing as opposed to reading, you clearly would be better served with another type of tree, such as a BST.