Post-Lab 5

Gabriel Groover (gtg3vv) – Lab 101 330-515

3/3/2016

TestFile1.txt:

Find “thinking” BST

Left Links: 2
Right Links: 3
Avg. Node Depth: 3

Find “thinking” AVL

Left Links: 1
Right Links: 1
Single Rotations: 2
Double Rotations: 2
Avg. Node Depth: 2

TestFile2.txt:

Find “flying” BST

Left Links: 0
Right Links: 5
Avg. Node Depth: 6

Find “flying” AVL

Left Links: 1
Right Links: 1
Single Rotations: 9
Double Rotations: 0
Avg. Node Depth: 2

TestFile3.txt:

Find “littered” BST

Left Links: 1
Right Links: 1
Avg. Node Depth: 3

Find “littered” AVL

Left Links: 0
Right Links: 0
Single Rotations: 1
Double Rotations: 2
Avg. Node Depth: 2

TestFile4.txt: “apple banana crumpet dagger elephant food gorilla home icecube jelly koala lemon monopoly noon observe pineapple quiet room steak time upset victory water xerox year zero”

Find “victory” BST

Left Links: 0
Right Links: 21
Avg. Node Depth: 12

Find “victory” AVL

Left Links: 1
Right Links: 2
Single Rotations: 21
Double Rotations: 0
Avg. Node Depth: 3

In order to demonstrate a scenario where AVL might be better than a BST, I created testfile4 to be a list of words in alphabetical order. When inserting comparable items in order to a binary search tree, each item will increase the depth by one and form a single chain. This format will cause the complexity of the find and insert methods to increase linearly. Very long lists of words will take an extremely long time to search through or add. An AVL allows the tree to be balanced so that the worst case complexity of both find and insert is log(n). In this worst case scenario, an AVL tree will always out perform a BST. We can see in the example used that finding a single word in the list required 21 node traversals for BST and only 3 for an AVL tree. This worst case scenario is what separates the two types of trees.

As discussed above, the AVL tree will always outperform the BST in the worst case scenario. However, they can perform similarly in the average case. If you are accepting only a few values, then the linear case is still adequate, and either tree will work. If the values are not sorted, a larger number of inputs may still function with a BST. An AVL tree becomes necessary when the values are being input in sorted order, or if the number of inputs is very large. For very large input, the complexity case is unknown. We know that the AVL tree will perform at least as well as BST so it is preferable. Although AVL trees are faster, there are tradeoffs in implementation. In an AVL tree, each node has a balance factor associated with it in addition to the value it holds. These values allow the tree to be adjusted and remain balanced. Holding these values requires extra space in memory. If we consider a binary search tree with 16 nodes and an AVL tree with the same number, the AVL tree will require much more memory. Other costs of the AVL tree are the difficulty of implementation and the speed required to balance. In order to maintain balance, nodes must rotate to ensure the height difference between leaves is never greater than one. There are four possible cases where rotations are required, and two of them require a double rotation. It is important to note that when the worst case insert outperforms the BST, the cost of rotations must be considered. In the example shown above with test 4, the find operation is extremely quick, but 21 rotations were required to balance the tree. It is significantly easier to search for a value than it is to add

a new one in an AVL tree. If you were only performing a single find, the difference between the two implementations would be marginal. These rotations also make writing the code more complicated. A BST allows a simple comparison of node values to carry out operations, while an AVL tree requires you to check balance every time. When a node is inserted into an AVL tree you must determine if the tree is unbalanced, and which of the four unbalanced cases it matches and write code to carry out the required rotations. To sum up, it is good to use AVL trees for large inputs, sorted inputs, or many find operations. Speed in these cases comes at the cost of memory, insertion speed and implementation.