CIS 303 Analysis of Algorithms
Spring 2020

Matthew Scaccia
Assignment 5b

04/30/2020

Problem

For this assignment we're to modify the AVLNode.java source code by implementing the balancing algorithm for an AVL tree. We're to add the four rotation methods, the cases that will trigger them, and any helper methods necessary. From here we're to test the tree independently for validity and correctness. Then we're to do a time cost analysis of insertion on both our AVL implementation and the given BST implementation.

Hypothesis

I hypothesize that the worst case performance of insertion for an Adelson-Velsky and Landis tree will be $\Theta(logn)$ and for a binary search tree we will see $O(n^2)$ performance. To get the worst case performance on insertion we need to be sequentially inserting values into both trees. This will cause the binary search tree to become horribly lopsided and gain worse performance than that of a linked list. The AVL tree however should self balance and retain its performance throughout the insertion process.

Methods

1. Implementation

```
private AVLNode balance (AVLNode rt) {
   if (rt == null)
        return rt;
    updateHeight(rt);
    /**
     * Recall balance factor is calculated as:
     * height of left subtree - hight of right subtree
     */
    int balanceFactor = getBalanceFactor(rt);
    /**
    * If we have a negative number we have a right leaning tree
    */
    if(balanceFactor < -1){
        if(height(rt.right.right) > height(rt.right.left)){
            rt = rotateLeft(rt);
        }else{
            rt = doubleRotateRightLeft(rt);
        }
    }
    /**
    * If we have a positive number we have a left leaning tree
    */
    else if( balanceFactor > 1){
        if(height(rt.left.left) > height(rt.left.right)){
            rt = rotateRight(rt);
        }else{
            rt = doubleRotateLeftRight(rt);
        }
    }
    return rt;
}

/**
 * Method for rotating right case, juggles pointers to rebalance right
 * @param rt AVL node
 * @return temp new root after single rotation of this rt left
 */
private AVLNode rotateRight(AVLNode rt) {
    AVLNode temp = rt.left;
    rt.left = temp.right;
    temp.right = rt;
```

```
        updateHeight(temp.right);
        updateHeight(temp);
        return temp;
    }
    /**
     * Method for rotating left case, juggles pointers to rebalance left
     * @param rt AVL node
     * @return temp new root after single rotation of this rt left
     */
    private AVLNode rotateLeft(AVLNode rt) {
        AVLNode temp = rt.right;
        rt.right = temp.left;
        temp.left = rt;
        updateHeight(temp.left);
        updateHeight(temp);
        return temp;
    }

    /**
     * Method for doing a Left Right rotation, does a left
     * rotation on the child node followed by a right rotation
     * on the grandparent node
     * @param rt the grandparent node we are rotating about
     * @return rt the node we have done a rotation about
     */
    private AVLNode doubleRotateLeftRight(AVLNode rt) {
        rt.left = rotateLeft(rt.left);
        return rotateRight(rt);
    }

    /**
     * Method for doing a Right Left rotation, does a right
     * rotation on the child node followed by a left rotation
     * on the grandparent node
     * @param rt the grandparent node we are rotating about
     * @return rt the node we have done a rotation about
     */
    private AVLNode doubleRotateRightLeft(AVLNode rt) {
        rt.right = rotateRight(rt.right);
        return rotateLeft(rt);
    }

    /**
     * Method: sets the height of the current node to the
     * maximum value of it's two subtrees
```

```
     * @param root the node we are currently traversing
     */
    private void updateHeight(AVLNode root){
        root.height = 1 + Math.max(height(root.left), height(root.right));
    }

    /**
     * Method takes the root node and caluclates the balance factor for the
     * left and right subtrees, returning either 0 if root is null or the
     * differnce of left - right subtree height.
     * @param root the node whos subtrees we will be checking
     * @return either 0 or the differnce of the left and right subtree heights
     */
    private int getBalanceFactor(AVLNode root){
        if(root == null) return 0;
        else return (height(root.left) - height(root.right));
    }
```

2. The general approach I took to this lab was to first: read the assigned course materials and watch the recommended videos on AVL rotations. While doing the latter I made sure to hand draw the trees and get a feel for rotations and how the cases would work. This allowed me to see the balance factor in action on a smaller scale and how this could be generalized. For my implementation I chose to keep the code as simple as possible. My state diagrams helped me to see I could handle the left and right cases with a positive and negative balance factor. Furthermore, I could also keep my methods simple as well by reusing my base rotations in the double forms. I also added in helper methods to update the height and get the balance factor of a given subtree.

   In approaching the experiment I felt that time testing both trees would be sufficient. I would be able to write the results to a file and then plot those results on a graph. I went a step further with this in doing ten insertions of N values and averaging the time. This was done to help me see the data, initially due to a bug, and it helped me to visualize what was happening with respect to growth time complexity.

3. The data structures used in this lab are AVL trees and binary search trees.

4. The experiments were two fold:
   For the AVL tree I did two experiments to stress test the tree. Included in the main directory of this program is the initial AVLClient.java used to test various rotations. I did this in order to compare the traversal output with that of Visualgo's output. I also decided to stress test the experiment before implementing it this same way, inserting 1-99 into my tree and verifying the output.

   For the actual experiment I felt that simply adding 2-N, for some value of N, in sequential order would be sufficient. This would lead to the binary search tree becoming unbalanced and force the AVL tree to do a series of re-balancing. Furthermore, I decided to do this ten

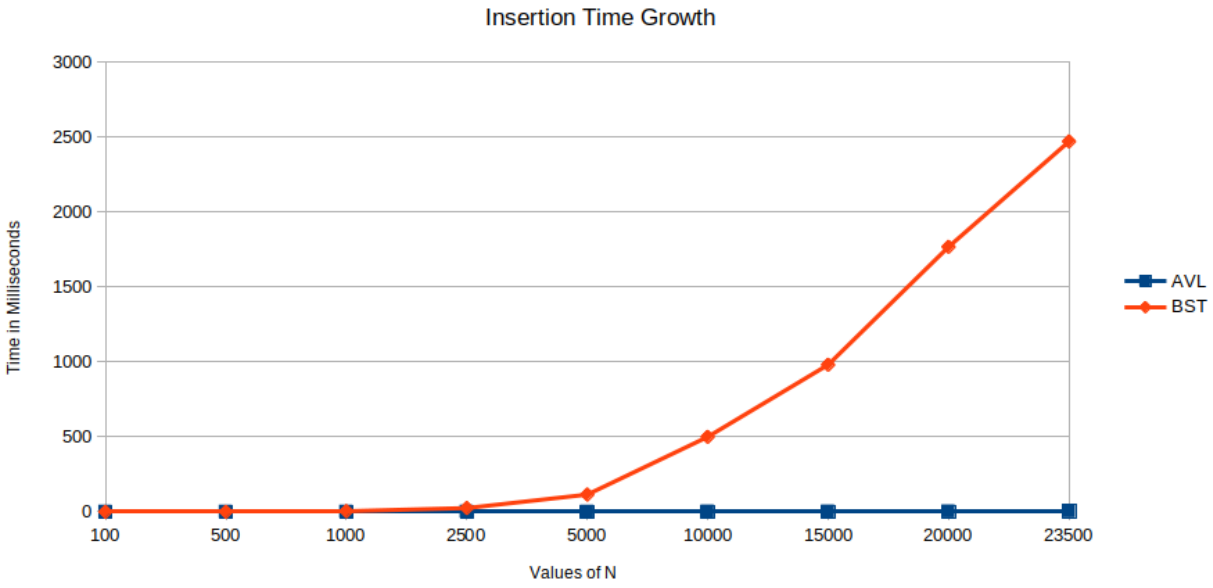
time for each value of N, I took the average of these insertion times and wrote it to a file for each tree. This was done initially to overcome a bug, but seeing the larger data plot with averaging helped to verify the code was correct and working as intended.

(a) The set of values: 100,500,1000,2500,5000,10000,15000, 20000, 23500 each serving as the total number of values being inserted. Originally the experiment called for 25000 as the last value, this had to be subbed in for 23500 due to the system that was used to test the program.

(b) For the AVL tree I tested all cases of the rotations, for the actual experiment only the worst case insertion was tested.

(c) For the AVL: I tested handful of times using various data sets that matched up with state diagrams and visualgo.
For the experiment I tested the averaging on the AVL up to 500,000 values. BST crashed around 23,500. Prior to writing the values in this paper to file, I tested full experiment a handful of times.

Results and Discussion



Time Growth Complexity		
AVL Tree	BST	Number of Sequential Insertions
0	0	1 00
0	0	500
0	4	1000
0	25	2500
0	115	5000
1	500	10000
1	981	15000
2	1768	20000
3	2471	23500

The results of the experiment show that when inserting N number of values sequentially the binary tree insertion time grows exponentially at N^2 vs the AVL which grows at a logarithmic time. The table more clearly shows the logarithmic growth of the AVL insertion time as N increases. The graph clearly demonstrates the exponential time growth of the binary search tree as N increases. I would have liked to use a different system with more processing power to push the N values higher to provide more data. With that said I feel that this is sufficient evidence to back up my hypothesis.

Conclusion

I feel that the data from my experiments support my hypothesis that the worst case insertion time complexity for AVL is logarithmic and for a binary search tree it is exponential. I think that the graph shows the latter and that with more data I could better visualize the time complexity growth of both structures. Given more time I would have liked to use a different system to run my experiments and up the number of insertions performed as I think this would reinforce my hypothesis. If I were to continue with this lab in any form I would further build the AVLNode and then test it for correctness and compare it with other implementations to better improve it, with documentation for reference.