Sayan Moitra
CS 242 - Data Structures
Mahadev
Programming Project 4 - Report

Difficulties

This was definitely the most difficult project out of the 4 programming assignments we've had. What made this one difficult in particular was the fact that parts of it are different from the traditional ways of solving.

Take for example the Find method in the Binary Search Tree portion of the project.
In a standard Find method, it basically just searches for the data by comparing against both the left and right sides of the tree. What was different about the Find method we had to code was the IntObject count. This was the function that generated the #s of comparisons which technically isn't a necessary part of a find function in general, but was for the purposes of our assignment. So in order to create the find method here, I had to look at IntObject and see exactly what it was doing. I saw it had two functions, getData() and setData(). So I had to understand what they did, and how they would fit in. The code involved to create the assignment's particular find function was only about 3 or 4 lines extra and took all of a couple minutes of pure coding time, but the effort to conceptualize and understand why exactly I had to, and more importantly how, took significantly longer.

The most difficult task was the rebalanceLeft method in the AVL Tree. Because it was given to us I assumed it was correct, but it was actually building the tree wrong so not only did I have to build the rebalanceRight as per the assignment, I also had to go back and trace to figure out what was wrong with rebalanceLeft and use the solution to that to build rebalanceRight. That took the most significant amount of my time.

Time Dispersion by Project Part:

BST: ~2 hours
AVL: ~8 hours (cumulative between several days)
RB: ~30 minutes

Total: ~11 hours

The Outputs:

For the BST portion, the surprising part about my output initially was that it was unable to successfully search Mog's Christmas. This was particularly interesting because it was simultaneously able to determine that Mog's Christmas was the minimum element, but not able to actually find and compare it. When I went to Prof. Mahadev about it, I learned that the solution was I wasn't accurately calling the smallest element. I fixed that and it worked perfectly.

For the AVL portion, the height was the surprising part. There were several comparison numbers that were also off, but the height being off suggested that there was something wrong with the way the tree was moving downwards. I later learned it was a problem with the initial rebalance method.

For the RB portion, nothing surprised me. I traced the code accurately and was able to get the output correct on the first time around.

Learning Opportunities

From this project I learned the importance of not just passing the tests, but also coding to account for edge cases as well. In the BST's smallest element search, I had it set to a left.left check, but in the event that the tree was arranged strictly to the right, it would've never been able to find it. So even though it was able to initially find Mog's Christmas and accurately display it as the minimum element, it could not actually find it within the tree due to its shape. That's why edge cases are so important.

Another thing I learned that it's really important to have a conceptual understanding of what you code and why. Especially for this project, it would've been significantly harder if I didn't understand at the beginning that trees function the same on both sides, left and right, just in different orders. If I were to have tried and brute force code through the opposite methods, it would've taken me far longer, but because I was smart about how I approached the problem, it took much less time.

Another thing I learned was how to trace someone else's code. For the previous assignments, I operated under the assumption that the code given to me was always correct, so I only focused on what I had to complete. This time, even though I was doing exactly what I was supposed to be doing, because the initial code I was given was incorrect, I was doing it wrong. This was the case in the AVL tree where the rebalanceLeft method was coded improperly. If I had taken the opportunity to actually look it over first before operating under the same assumptions as before, it would've taken me much less time.

Real-Life Applications

BSTs: Binary Search Trees in my opinion would only be useful if they are balanced to begin with. If you have a situation similar to Mog's Christmas outlined above, where the tree is in a strict right only orientation, then it's effectively degenerated into a worst-case form, in which you're much better off using something like a Linked List instead.

AVL: Now if you have a balanced binary search tree, which is an AVL, then you have something worth looking into. If they are balanced, you have better performance of O(logn). This can be used to structure databases to help find things much easier. If you had to use a lookup function on a massive data set, using a list would take forever, but an AVL or balanced BST would be able to do it in a fraction of the time.

RB: Red Black trees would primarily be used where instead of searching for something, you'd actually want to insert something, since RB trees are optimized for height balancing. So if you take the database example from the AVL application, if now instead of looking up something you wanted to insert or delete something, an RB tree would be more efficient than an AVL tree for the same data instance.