* Entered Values = (14, 34, 63, 9, 93, 49, 82, 53, 75, 11, 67, 29, 99, 80, 23)

* For the values that I entered in each binary search tree, for the AVL tree I ended up getting 63 as the root, a tree depth of 4, and 7 total leaf nodes. This is not an unexpected outcome since AVL trees are much more intuitive than either of the others we are learning about. Originally 14 and 34 was the root but that changed when I added 63 and 67, respectively, and the tree became unbalanced.
* For the Red-Black tree, the root ended up being 63, with a tree depth of 4 and 7 total leaves, but this was a lot more confusing than the AVL. At one point, when I added the value 67 to the tree, two reds appeared in a row, and it had to shift roots from 34 to 67 instead of just getting the blackness inherited to the red nodes. Also, for the value 23, the Red-Black tree must first make that a right child to the grandparent, since it is a left child to a right child. Then, it makes 23 a root of the values 14 and 23, which is confusing why it must first make it a right child and then make it the root.
* For the splay tree, 23 ended up as the root node, with a tree depth of 5 and only 4 leaves, this was the most confusing one of all. On the surface, it sounds intuitive and straight-forward but once I started inputting numbers, I quickly realized there was a lot more going on than just implementing a binary search tree as a stack. For example, 80 was entered directly before 23 which is the root but 80 is not the next Node available. It is the grand-child to the root node and I just can’t wrap my head around how that is implemented in the code.