# Data Structures and Algorithms 2 Assignment Report BST vs AVL vs RBT

## Name & ID: Nicholas Falzon 19603(H)

**Please Note:** Throughout this report I will primarily be focusing on evaluating the pros and cons of the different tree structures rather than explaining my implementation details. I feel that I explained my implementation wherever required through the use of inline comments throughout my code. I have also included references to sites from which I adapted any of my code in the jupyter notebook provided.

## Statement of Completion

| Item | Completed |
|---|---|
| Created sets X, Y and Z without duplicates and showing intersections. | Yes |
| AVL tree insert | Yes |
| AVL tree delete | Yes |
| AVL tree search | Yes |
| RB tree insert | Yes |
| RB tree delete | Yes |
| RB tree search | Yes |
| Unbalanced BST insert | Yes |
| Unbalanced BST delete | Yes |
| Unbalanced BST search | Yes |
| Discussion comparing tree data structures | Yes |

## Testing

In order to test if my tree implementations were done correctly, I included a function for each tree structure which outputs the inorder traversal of each tree. I used this function to print out the in order traversal of each tree after inserting the elements of X and also after deleting the elements of Y. Since the values were printed out in a properly sorted manner, I knew that the values inserted/deleted were being placed in their appropriate positions inside the tree.

## Evaluation of Tree Structures

### Binary Search Trees

The first thing that I feel should be addressed is that in general, a Binary Search Tree should never be used over an AVL tree or a Red Black Tree. After repeatedly performing the binary tree operations which I implemented, (insertion, deletion and searching) I noticed that Binary Search Trees consistently obtain the least desired results.

More specifically, after insertion and deletion, the Binary Search Tree always provided the largest value in terms of tree height. This is of course due to the fact that BSTs have no balancing rules and this will therefore allow the binary tree to skew in either direction. Increased height is undesirable in a binary tree since the time taken to insert, delete and search is dependent on the height of the tree. In the worst case, these operations will have a time complexity of O(n) where n is the number of nodes, which is poor compared to the worst case time complexity of O(log n) for AVL trees and RBTs. Therefore, searching will likely take longer to perform using a binary search tree than using any of the other two trees, particularly if a large number of nodes are present inside the trees. Despite this, inserting and deleting nodes is generally more time efficient using BSTs since no rotations are required to be performed, however this does become less efficient as the height of the tree grows.

Additionally, after performing all three operations, the Binary Search Tree always produced the largest value in terms of comparisons performed, which goes to show that the less the height of the tree is, the less comparisons will need to be done, which ultimately leads to the tree being more efficient in terms of inserting, deleting and searching.

The only advantage that I believe BSTs have over the other two trees is that they are much easier to implement than the other two trees.

## AVL vs Red Black Trees

Overall, AVL trees and red black trees behave relatively similar but each have key characteristics which determine the applications in which they should be used. In terms of similarities, the heights of both trees are very similar to each other and from the testing which I have conducted, their heights after inserting and deleting the same elements have never differed by more than a value of two. In spite of this, I am quite certain that it is possible for the heights to have a greater difference than just two since the red black tree has less strict balancing properties than the AVL tree. In an AVL tree, the difference between the depth of the right and left subtrees of each node must be less than two, but red black trees do not have this property meaning there are cases where the height of the red black tree can be larger than that of the AVL tree (even without considering the TNULL leaf nodes).

From the tests which I have done, AVL trees consistently have a height fewer than that of red black trees, however this is usually only by a value of one which is caused by the addition of the null nodes which are the leaves of the red black tree, causing the tree to grow by a height of one. In fact, because of these null nodes, red black trees always have (double + 1) the number of nodes that the BST or the AVL tree have. Because of the similar height, the number of comparisons required to perform the tree operations is also very similar, with the AVL tree always requiring slightly less comparisons than the red black tree.

Although both tree structures have a search time complexity of O(log n), because of the slightly lower height, the AVL tree will generally perform better when it comes to searching.

The notable difference between both these tree structures can be seen when observing the number of rotations performed. Although both have an insertion and deletion worst case and average case time complexity of O(log n), red black trees will always require less rotations to be performed when inserting or deleting as they firstly check if it is possible to perform recolouring before performing a rotation. This gives red black trees an edge over AVL trees when it comes to insertion and deletion time efficiency because performing recolouring is less time consuming than performing rotations.

## Real-World Applications

### Binary Search Trees

Because of the possibility of O(n) search time, binary search trees should never be used in applications where a real-time or close to real-time response is expected. Rather, binary search trees should be used when the user is not expecting an immediate response. They can be used within small-scale database applications, particularly if insertion and deletion will be common operations and searching will be an uncommon operation.

### AVL Trees

Since AVL trees provide the fastest searching time on average, this is the structure which should be used if a scenario requires a real-time response. Since they provide a fast search time on average, AVL trees are fit to be used in large-scale database applications where insertions and deletions are less common but frequent data lookups (searches) are conducted.

### Red Black Trees

Red black trees are the best option out of the three structures to implement for a large-scale database application in which the user is confident that insertions and deletions will be common. Although search time will be slower, it is a tradeoff which is worth making for many applications since the lower number of rotations required will benefit applications where the data stored is changed regularly.

## Conclusion

From the tests which I have performed, I am able to conclude a couple of statements. Firstly, binary search trees should never be used to store a large amount of data, but can do the job for a small amount of data since the height of the tree won't grow to a point where searching is slow. Secondly, I've learned that RBTs can be used for practically every scenario as they perform well all-round and don't have a standout weak point besides the fact that they are difficult to implement. Finally, I can conclude that AVL trees perform the best in terms of searching for and should be used over the other tree structures if the data inside the tree won't be changing much.