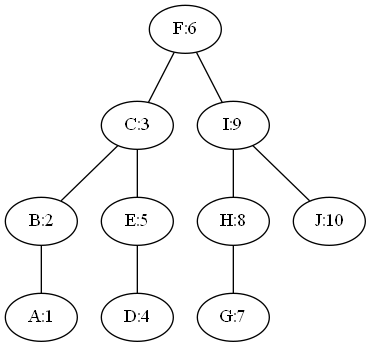
## 5. Creating a B`alanced Search Tree



This is the output that I get. I used :

|  |  |
| --- | --- |
| sortArr(self, root\_node, arr) | createdBalancedTree(self, arr) |
| 1. Root node -> the root itself 2. arr -> an empty arr | 1. arr -> the sorted array |

Using the sortArr function to sort the number ascendingly and append it into the temp array.

Make the middle number of the array to be the root. Recursively do the same for the left half and the right half. Using the concept of merge sort, Repeatly find the middle value of the left half to be left child. Also, repeatly find the middle value of the right half to be the right child. Return the root to be the new root.

## 6. Searching for a Key in BST

The function is searching for the value that is inside the root. While the root is not empty, if the key is larger than the input key, the node will shift to the left else it will shift to the right. This process will loop till the key is found. If there is no key being found, it will return none.

## 7. Finding the Size of a Subtree in BST

Using 2 function to find the size of the nodes. If the node is empty, it will return 0. The root will keep running in a while loop till it search the selected key. Aftermath, it will proceed to the next size2() and recursively increase the size of the left and right node if it is not none.

## 8. Finding the Depth of a Node in BST

The function will search for the key that is given by the user and call depth2() when found. It will recursively calculate the height of the tree to the left of the root, follow by the height of the tree to the right of the root recursively.

## 9. Finding the Height of a Node in BST

The function will search for the key that is given by the user and call height2 when found. The height will recursively calculate the height of the left and right subtree. Lastly it will calculate the max of the heights of two children and add 1 to it.

## 10. PreOrder, InOrder and PostOrder Traversal of Tree

### 1) PreOrder

Get the root of the key and value before recursively left and right the root.

### 2) InOrder

Recursively call the left subtree, get the root of the key and value and lastly recursively call the right subtree.

### 3) PostOrder

Recursively call the left and right subtree follow by the root of the key and value

## 12. Performance Measurement

a)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Number of input N | | | | | |
|  | 50 | 100 | 200 | 400 | 800 |
| BST | 0.0015586999999999962 | 0.0017014000000000057 | 0.0031856999999999996 | 0.018607399999999996 | 0.0771007 |
| LLRBT | 0.001007999999999995 | 0.0011082999999999926 | 0.0002647999999999956 | 0.0038293999999999967 | 0.0081537 |

Time taken to insert N different numbers in sequential order

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Number of input N | | | | | |
|  | 50 | 100 | 200 | 400 | 800 |
| BST | 0.0012053999999999954 | 0.001367099999999996 | 0.001326899999999992 | 0.004057900000000003 | 0.03095809999999999 |
| LLRBT | 0.0019156999999999924 | 0.0014207999999999998 | 0.004076099999999999 | 0.0015106000000000008 | 0.0014271000000000006 |

Time taken to search N different numbers in sequential order

c)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Number of input N | | | | | |
|  | 50 | 100 | 200 | 400 | 800 |
| BST | 0.0013149000000000077 | 0.0012918000000000027 | 0.0011417999999999984 | 0.0012056000000000011 | 0.0025281000000000053 |
| LLRBT | 0.0010739000000000026 | 0.0016579000000000038 | 0.0019456999999999947 | 0.0038031000000000037 | 0.009288699999999997 |

Time taken to insert N different numbers in random order

d)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Number of input N | | | | | |
|  | 50 | 100 | 200 | 400 | 800 |
| BST | 0.0004045000000000021 | 0.0004553999999999947 | 0.0005544999999999994 | 0.0011668999999999985 | 0.0017614999999999992 |
| LLRBT | 0.0012355000000000005 | 0.001129000000000005 | 0.0013084000000000012 | 0.0010818999999999968 | 0.0015017999999999976 |

Time taken to search N different numbers in random order

This experiment is using an unbalance BST and LLBRT. After examining, it is shown that the LLBRT have the fastest complete time compare to the BST. However, the random insertion of LLRBT seems to take a longer time.

Based on LLBRT, the sequential insertion is faster comparing to the random insertion. This is because the LLBRT is constantly adding to the right as the values of the number is only increasing. The worst case that could be happen to the insertion is O(lg N) and the average case will be O(lg N) too. Random insertion will cause LLBRT to trace and add the node to the binary tree as well as keep the tree balance. That’s the reason why the random insertion seems to take a longer time. The search worst and average case of LLBRT is O(lg N). There is not much difference in the time between the random and sequential search.



For the BST, it is unbalanced. The complexity of the insertion for the worst case is O(lg N) and the average case is O(lg N). The time of the insertion based on the random and sequential does not make any huge difference with the timing. The worst case for search is O(N) and the average case is O(lg N).

Hence, LLBRT have a faster time completing the tree compare to BST. It also helps to balance out the tree and can complete quicker with a large amount of data being input. The only drawback is the random insertion of LLBRT where the tree needs to trace back.