APS Lab-7
Reference

AVL Tree | Set 1 (Insertion)

AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

Why AVL Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that height of the tree remains O(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations. The height of an AVL tree is always O(Logn) where n is the number of nodes in the tree (See this video lecture for proof).

Insertion

To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) < key(root) < keys(right)). 1) Left Rotation 2) Right Rotation

```
T1, T2 and T3 are subtrees of the tree         rooted with y (on left side)
or x (on right side)
                  y                                  x
                   / \     Right Rotation              / \
              x      T3     – – – – – – – – >            T1   y
                 / \       < - - - - - - - -                / \
            T1        T2    Left Rotation              T2         T3
Keys in both of    the above trees follow      the following order
                    keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)
So BST property    is not   violated anywhere.
```

Steps to follow for insertion

Let the newly inserted node be w

1) Perform standard BST insert for w.

2) Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.

3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:

a) y is left child of z and x is left child of y (Left Left Case) b) y
is left child of z and x is right child of y (Left Right Case)

c) y is right child of z and x is right child of y (Right Right Case)

d) y is right child of z and x is left child of y (Right Left Case)

Following are the operations to be performed in above mentioned 4 cases. In all of the cases, we only need to re-balance the subtree rooted with z and the complete tree becomes balanced as the height of subtree (After appropriate rotations) rooted with z becomes same as it was before insertion.

a) Left Left Case

```
T1, T2, T3 and T4 are subtrees.
         z  y
        / \ /   \
       y    T4    Right Rotate (z)  x     z
      / \   - - - - - - - - ->      /  \  /  \
     x   T3 T1  T2  T3  T4
    / \
  T1  T2
```

b) Left Right Case

```
    z          z            x
    / \                             /  \                        /  \
   y   T4  Left Rotate (y)        x    T4  Right Rotate(z)     y     z
  / \      - - - - - - - - ->    /  \      - - - - - - - ->  / \     / \
T1    x                         y   T3                     T1  T2 T3  T4
    / \                        / \
  T2   T3                    T1    T2
```

c) Right Right Case

```
    z             y
  /      \                                                 /  \
 T1   y           Left Rotate(z)                         z        x
         / \      - - - - - - - - ->                    / \        / \
        T2  x                                         T1  T2 T3  T4
           / \
      T3      T4
```

d) Right Left Case

```
  z                        x
 / \                             / \                            /  \
T1   y     Right Rotate (y)   T1     x     Left Rotate(z)     z     y
      / \  - - - - - - - - ->      / \  - - - - - - - - ->  / \      / \
  x      T4                     T2     y                  T1   T2  T3  T4
 / \                                  / \
T2   T3                              T3  T4
```

AVL Tree | Set 2 (Deletion)

We have discussed AVL insertion , we will follow a similar approach for deletion.

Steps to follow for deletion.
To make sure that the given tree remains AVL after every deletion, we must augment the standard BST delete operation to perform some re-balancing. Following are two basic

operations that can be performed to re-balance a BST without violating the BST property (keys(left) < key(root) < keys(right)). 1) Left Rotation 2) Right Rotation

```
T1,
T2
and
T3  roo
are ted
sub wit
tre h y
es  (on
of  lef
the t
tre sid
e   e)
or x (on right side)
                  y                                   x
                 / \     Right Rotation              / \
              x     T3   – – – – – – – >          T1   y
                / \      < - - - - - - -                / \
             T1     T2   Left Rotation              T2     T3
Keys in both of  the above trees follow  the following order
             keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)
So BST property  is not  violated anywhere.
```

Let w be the node to be deleted

1) Perform standard BST delete for w.

2) Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the larger height child of z, and x be the larger height child of y. Note that the definitions of x and y are different from insertion here.

3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:

a) y is left child of z and x is left child of y (Left Left Case) b) y is left child of z and x is right child of y (Left Right Case)

c) y is right child of z and x is right child of y (Right Right Case) d) y is right child of z and x is left child of y (Right Left Case)

Like insertion, following are the operations to be performed in above mentioned 4 cases. Note that, unlike insertion, fixing the node z won’t fix the complete AVL tree. After fixing z, we may have to fix ancestors of z as well

a) Left Left Case

```
T1, T2, T3 and T4 are subtrees.
         z  y
        / \ /   \
       y    T4    Right Rotate (z)  x      z
      / \   - - - - - - - - ->      /  \  /  \
     x   T3 T1  T2  T3  T4
    / \
  T1  T2
```

b) Left Right Case

```
    z          z            x
      / \                               /  \                           /  \
   y   T4  Left Rotate (y)             x   T4  Right Rotate(z)        y      z
  / \         - - - - - - - - - ->     /  \      - - - - - - - - -> / \       / \
T1     x                             y   T3                    T1     T2 T3  T4
      / \                           / \
  T2    T3                       T1    T2
```

c) Right Right Case

```
    z             y
   /       \                                                     /  \
  T1   y            Left Rotate(z)                              z        x
           / \      - - - - - - - - ->                         / \         / \
          T2  x                                               T1  T2 T3  T4
             / \
       T3      T4
```

d) Right Left Case

```
  z                        x
  / \                              / \                               /  \
T1    y       Right Rotate (y)   T1      x     Left Rotate(z)       z      x
         / \  - - - - - - - - - ->     / \   - - - - - - - - ->  / \          / \
   x        T4                     T2     y                  T1     T2   T3  T4
  / \                                    / \
T2    T3                                 T3  T4
```

Unlike insertion, in deletion, after we perform a rotation at z, we may have to perform a rotation at ancestors of z. Thus, we must continue to trace the path until we reach the root.

****