Lesson 25-26

Objectives

- AVL Tree
- Why to use AVL Tree
- Rotation in AVL Tree
- Operations on AVL Tree
- Steps for Insertion
  - LL case
  - LR case
  - RR case
  - RL case
- Steps for Insertion
  - LL case
  - LR case
  - RR case
  - RL case
- Implementation

AVL:

Concept:

- An AVL (Adelson-Velskii and Landis) tree is a binary search tree with a *balance* condition.
- The balance condition must be easy to maintain, and it ensures that the depth of the tree is $O(\log n)$.
- The simplest idea is to require that the left and right sub trees have the same height.
- In AVL trees each node stores an additional piece of data: the difference between the heights of its left and right sub-trees.
- the height of a node's left sub-tree may be no more than
- one level different from the height of its right sub-tree.
- AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree
- Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

Balance Factor (k) = height (left(k)) - height (right(k))

- If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.
- If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.
- If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

- An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. therefore, it is an example of AVL tree.

AVL Tree Rotations

In AVL tree, after performing every operation like insertion and deletion we need to check the balance factor of every node in the tree. If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced. We use rotation operations to make the tree balanced whenever the tree is becoming imbalanced due to any operation. Rotation operations are used to make a tree balanced. There are four rotations and they are classified into two types:

Single Left Rotation (LL Rotation)

In LL Rotation every node moves one position to left from the current position.

Single Right Rotation (RR Rotation)

In RR Rotation every node moves one position to right from the current position.

Left Right Rotation (LR Rotation)

The LR Rotation is combination of single left rotation followed by single right rotation. In LR Rotation, first every node moves one position to left then one position to right from the current position.

Right Left Rotation (RL Rotation)

The RL Rotation is combination of single right rotation followed by single left rotation. In RL Rotation, first every node moves one position to right then one position to left from the current position.

Operations on AVL tree

Due to the fact that, AVL tree is also a binary search tree therefore, all the operations are performed in the same way as they are performed in a binary search tree. Searching and traversing do not lead to the violation in property of AVL tree. However, insertion and deletion are the operations which can violate this property and therefore, they need to be revisited.

Insertion

Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations.

Deletion

Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the tree.

Steps to follow for insertion

Let the newly inserted node be w

- Perform standard BST insert for w.
- 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
- Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that need 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. (See this video lecture for proof)

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
```

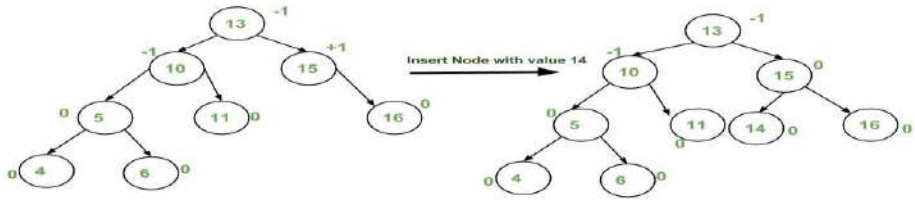
c) Right Right Case

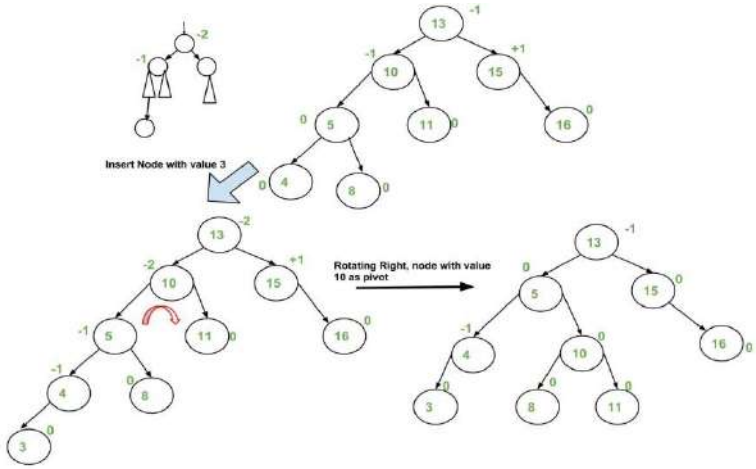
```
  z                                      y
 /  \                                  /   \
T1   y     Left Rotate(z)             z      x
    /  \   - - - - - - - - ->        / \    / \
   T2   x                           T1  T2 T3  T4
       / \
     T3  T4
```

d) Right Left Case

```
   z                                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
```

Insertion Examples:





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 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.
```

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 (See this video lecture for proof)

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

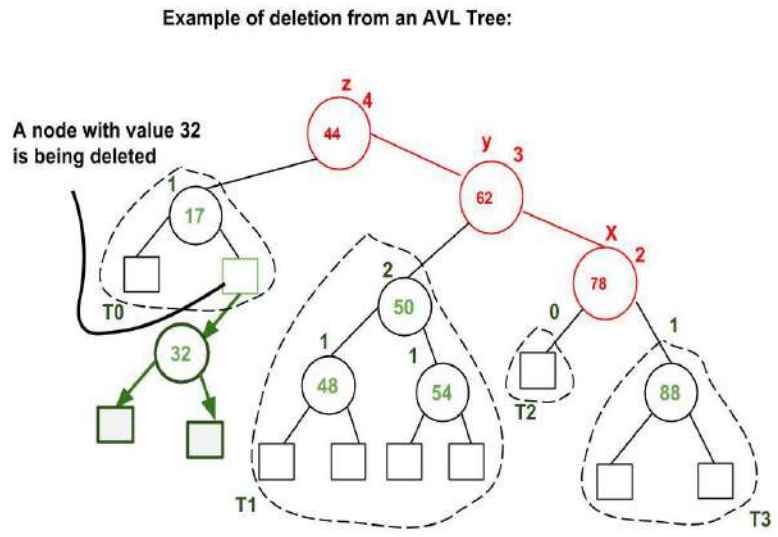
```
  z                         y
 / \                       /  \
T1   y     Left Rotate(z)       z      x
    / \  - - - - - - - - ->    / \    / \
   T2   x                     T1  T2 T3  T4
       / \
      T3  T4
```

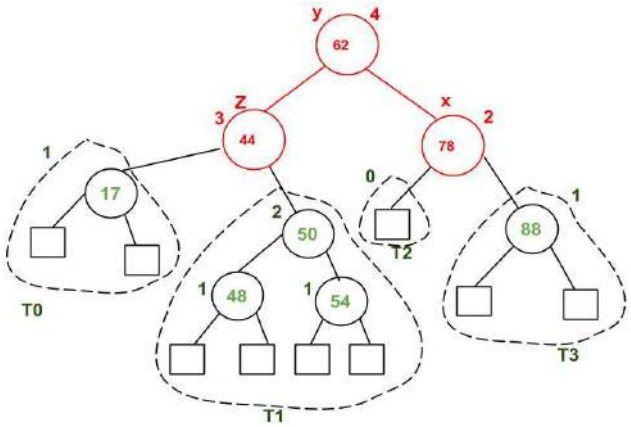
d) Right Left Case

```
   z                       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
```

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.

Example:





A node with value 32 is being deleted. After deleting 32, we travel up and find the first unbalanaced node which is 44. We mark it as z, its higher height child as y which is 62, and y's higher height child as x which could be either 78 or 50 as both are of same height. We have considered 78. Now the case is Right Right, so we perform left rotation.

Recommended: Please solve it on "PRACTICE" first, before moving on to the solution.

Java implementation

Following is the C implementation for AVL Tree Deletion. The following C implementation uses the recursive BST delete as basis. In the recursive BST delete, after deletion, we get pointers to all ancestors one by one in bottom up manner. So we don't need parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the deleted node.

- Perform the normal BST deletion.
- The current node must be one of the ancestors of the deleted node. Update the height of the current node.
- Get the balance factor (left subtree height – right subtree height) of the current node.
- If balance factor is greater than 1, then the current node is unbalanced and we are either in Left Left case or Left Right case. To check whether it is Left Left case or Left Right case, get the balance factor of left subtree. If balance factor of the left subtree is greater than or equal to 0, then it is Left Left case, else Left Right case.
- If balance factor is less than -1, then the current node is unbalanced and we are either in Right Right case or Right Left case. To check whether it is Right Right case or Right Left case, get the balance factor of right subtree. If the balance factor of the right subtree is smaller than or equal to 0, then it is Right Right case, else Right Left case.

Task I: Code via Link-List Insertion, Deletion Display and all Orders

Code:

```
class Node {
    int key, height;
    Node left, right;

    Node(int d) {
        key = d;
        height = 1;
    }
}
```

```
class AVLTree {

    Node root;

    // A utility function to get the height of the tree
    int height(Node N) {
        if (N == null)
            return 0;

        return N.height;
    }

    // A utility function to get maximum of two integers
    int max(int a, int b) {
        return (a > b) ? a : b;
    }

    // A utility function to right rotate subtree rooted with y
    // See the diagram given above.
    Node rightRotate(Node y) {
        Node x = y.left;
        Node T2 = x.right;

        // Perform rotation
        x.right = y;
        y.left = T2;

        // Update heights
        y.height = max(height(y.left), height(y.right)) + 1;
        x.height = max(height(x.left), height(x.right)) + 1;

        // Return new root
        return x;
    }

    // A utility function to left rotate subtree rooted with x
    // See the diagram given above.
    Node leftRotate(Node x) {
        Node y = x.right;
        Node T2 = y.left;

        // Perform rotation
        y.left = x;
        x.right = T2;

        //  Update heights
        x.height = max(height(x.left), height(x.right)) + 1;
        y.height = max(height(y.left), height(y.right)) + 1;

        // Return new root
        return y;
    }

    // Get Balance factor of node N
```

```
int getBalance(Node N) {
    if (N == null)
        return 0;

    return height(N.left) - height(N.right);
}

Node insert(Node node, int key) {

    /* 1.  Perform the normal BST insertion */
    if (node == null)
        return (new Node(key));

    if (key < node.key)
        node.left = insert(node.left, key);
    else if (key > node.key)
        node.right = insert(node.right, key);
    else // Duplicate keys not allowed
        return node;

    /* 2. Update height of this ancestor node */
    node.height = 1 + max(height(node.left),
                          height(node.right));

    /* 3. Get the balance factor of this ancestor
          node to check whether this node became
          unbalanced */
    int balance = getBalance(node);

    // If this node becomes unbalanced, then there
    // are 4 cases Left Left Case
    if (balance > 1 && key < node.left.key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node.right.key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node.left.key) {
        node.left = leftRotate(node.left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node.right.key) {
        node.right = rightRotate(node.right);
        return leftRotate(node);
    }

    /* return the (unchanged) node pointer */
    return node;
}

// A utility function to print preorder traversal
```

```
    // of the tree.
    // The function also prints height of every node
    void preOrder(Node node) {
        if (node != null) {
            System.out.print(node.key + " ");
            preOrder(node.left);
            preOrder(node.right);
        }
    }
 Node deleteNode(Node root, int key)
    {
        // STEP 1: PERFORM STANDARD BST DELETE
        if (root == null)
            return root;

        // If the key to be deleted is smaller than
        // the root's key, then it lies in left subtree
        if (key < root.key)
            root.left = deleteNode(root.left, key);

        // If the key to be deleted is greater than the
        // root's key, then it lies in right subtree
        else if (key > root.key)
            root.right = deleteNode(root.right, key);

        // if key is same as root's key, then this is the node
        // to be deleted
        else
        {

            // node with only one child or no child
            if ((root.left == null) || (root.right == null))
            {
                Node temp = null;
                if (temp == root.left)
                    temp = root.right;
                else
                    temp = root.left;

                // No child case
                if (temp == null)
                {
                    temp = root;
                    root = null;
                }
                else // One child case
                    root = temp; // Copy the contents of
                                 // the non-empty child
            }
            else
            {

                // node with two children: Get the inorder
                // successor (smallest in the right subtree)
                Node temp = minValueNode(root.right);

                // Copy the inorder successor's data to this node
```

```
                root.key = temp.key;

                // Delete the inorder successor
                root.right = deleteNode(root.right, temp.key);
            }
        }

        // If the tree had only one node then return
        if (root == null)
            return root;

        // STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
        root.height = max(height(root.left), height(root.right)) + 1;

        // STEP 3: GET THE BALANCE FACTOR OF THIS NODE (to check whether
        // this node became unbalanced)
        int balance = getBalance(root);

        // If this node becomes unbalanced, then there are 4 cases
        // Left Left Case
        if (balance > 1 && getBalance(root.left) >= 0)
            return rightRotate(root);

        // Left Right Case
        if (balance > 1 && getBalance(root.left) < 0)
        {
            root.left = leftRotate(root.left);
            return rightRotate(root);
        }

        // Right Right Case
        if (balance < -1 && getBalance(root.right) <= 0)
            return leftRotate(root);

        // Right Left Case
        if (balance < -1 && getBalance(root.right) > 0)
        {
            root.right = rightRotate(root.right);
            return leftRotate(root);
        }

        return root;
    }


    public static void main(String[] args) {
        AVLTree tree = new AVLTree();

        /* Constructing tree given in the above figure */
        tree.root = tree.insert(tree.root, 10);
        tree.root = tree.insert(tree.root, 20);
        tree.root = tree.insert(tree.root, 30);
        tree.root = tree.insert(tree.root, 40);
        tree.root = tree.insert(tree.root, 50);
        tree.root = tree.insert(tree.root, 25);
```

```
        /* The constructed AVL Tree would be
             30
            /  \
          20   40
         /  \     \
        10  25    50
        */
        System.out.println("Preorder traversal" +
                           " of constructed tree is : ");
        tree.preOrder(tree.root);
    }
}
```