AVL Trees

An AVL tree is a self-balancing binary search tree which has the following properties

- The differnce in heights of the left and right subtrees cannot be more than one for any node in the tree
- Building the tree in O(nlogn)
- Height of tree is O(logn). Implies
    1. Insertion in O(logn)
    2. Deletion in O(logn)
    3. Searching in O(logn)

Implementation

An AVL tree like any other binary tree can be implemented using linked lists.
Each node stores its value and the pointers to its left child, right child and parent.

Insertion

concept:

```
We find the first ancestor node which is disbalnced from t
he leaves and perform right or left rotations according to
```

```
 the structure of the tree to make it balanced and follow
the AVL property.
```

pseudocode:

```
struct node
  int val, height;              // Value of node
  node* left, right, parent;    // Pointers to left, right
 and parent nodes

v -> value to be inserted

// Function returns the new root to the caller
node* insert(root, v):
    if root == NULL:
        return newNode(v)      // newNode() returns a newl
y allocated node with value v
    else if (v less than root->val):
        root->right = insert(root->right, v)         // In
sert into right
    else:
        root->left = insert(root->left, v)           // In
sert into left

    root->height = max(root->left->height, root->right->he
ight) + 1;    // Update the height of the node now

```

```
    // Calculate difference between the heights of left an
d right subtrees
    int diff = root->left->height - root->right->height



    if (diff is less than 2 and greater than -2)
        return root                                  //  B
alanced
    if ( diff < -1 )
        //  Right tree larger
        if ( v is greater than or equal to val of right ch
ild )
            // Right right case
            root = rotate_left (root)
        else
            // Right left case
            root->right = rotate_right (root->right)
            root = rotate_left(root)
    else
        //  Left tree larger
        if ( v is less than val of left child )
            //  Left left case
            root = rotate_right (root)
        else
            //  Left right case
            root->left = rotate_left (root->left)
            root = rotate_right (root)
    return root
```

```
// Right rotate the node
rotate_right (root)

    let x = left child of root
    let y = right child of x
    // Perform rotations
    x -> right = root
    root -> left = y
    // Update heights of root and x ( Note height of y doesnt change )
    height of root = max( height of left subtree of x, height of right subtree of x)
    height of x = max( height of left subtree of x, height of right subtree of x)

    return x    //  x is new root

// Left rotate is similar and symmetric to right
```

With this we come to the end of insertion. :)

Deletion

idea

- First we delete exactly as we do in BST.
- Then we balance heights if we get a disbalance.

pseudocode

```
node* delete (node* root, ll v)
{

Step 1:

    if root is NULL
        return NULL;

    else if (v is less than root->val)
        root->left = delete(root-left, v);

    else if(v is greater than root->val)
        root->right = delete(root->right, v);

    else
        // it means that we have found the node we have to
 delete

        // if it is a leaf
        if (root->left==NULL and root->right==NULL)
            free(root);
            return NULL;

        // if the node has only one child
        else if (root->left==NULL)
            return (right Child of root);

        else if (root->right==NULL)
            return (left Child of root);


        // if the node has two children
        else
            node* temp = successor (root);       // find t
he next minimum element
            root->val = temp->val;               // replac
e this node's value with successor's value;
            delete (root->right, temp->val);     // delete
 successor

Step 2:

    root->height = max(height of left child, height of rig
ht child) + 1;


    ll diff = (height of left child) - (height of right ch
ild);
    ll left_diff = (similar analogy as above)          //
 diff of left child of root
    ll right_diff = (similar analogy as above)         //
 diff of right child of root
```

```
    // if diff if -1, 0, or 1, then no need to worry.

    if (diff > 1 and left_diff >= 0)
        return rotate_right(root);

    else if (diff > 1 and left_diff < 0)
        root->left = rotate_left(root->left);
        return rotate_right(root);

    else if (diff < -1 and right_diff <= 0)
        return rotate_left(root);

    else if (diff < -1 and right_diff > 0)
        root->right = rotate_right(root->right);
        return rotate_left(root);
}
```

With this, we come to the end of deletion.