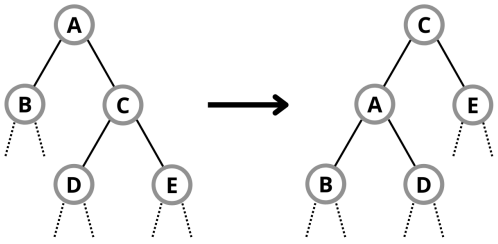
1 Introduction

AVL trees are a data structure which ensures that a tree stays balanced so that lookups are fast. Imagine a situation where 1, 2, 3, ... n are inserted into a binary tree in that order. A normal binary tree will keep inserting to the right of the only leaf of the tree, effectively making it a list. This means that any lookup will take $O(n)$ instead of the $O(\log n)$ that the cool binary trees advertise. AVL trees will perform "rotations" to avoid these types of situations, keeping the lookup time to $O(\log n)$.

2 Single Rotations

AVL tree nodes keep track of an extra value, the height of the node. This height is determined by the length of the longest path starting from the node itself in its subtree. To keep the tree balanced, a rotation is made on a node A whenever the heights of the two subtrees of A differ by at least 2. If a node has no child in a left or right position, the position will be considered to have heigh 0. A rotation on node A will change the tree in a way that makes the height differences at most 1 for children of all nodes involved in the rotation.

Rotations have two types, left and right, which are mirror images of each other. A left rotation on a specified node A will push the left side of node A down and pull the right side of node A up, and vice versa for a right rotation. An example of a left rotation and its pseudocode is shown below.




```
def left_rotation(A):
    C = right_child(A)
    right_child(A) = D; parent(D) = A
    left_child(C) = A; parent(A) = C
```

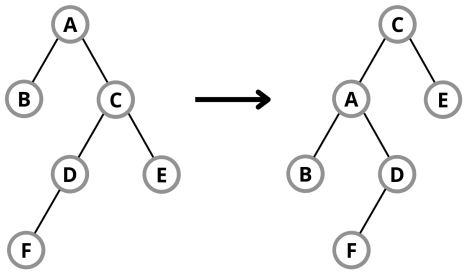
A right rotation is programmed similarly. Depending on what the heights of the subtrees of a given node are, a right or left rotation may be needed. If the right subtree of the node has greater height than the left subtree, a left rotation is needed and vice versa. Intuitively, a rotation should pull the side with greater height up while pushing the other side down to balance the heights.

An important feature about these rotations is that they preserve the binary tree condition, which makes using these rotations possible. In the result of

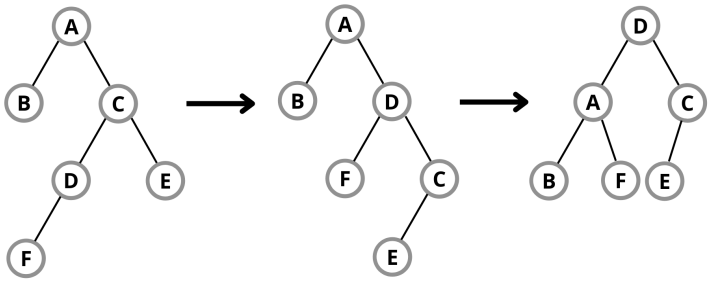
the example of the left rotation, A is larger than B and less than D, which is correctly seen in the starting position where B is to the left of A and D is in the right subtree of A. Similarly, C is larger than A and smaller than E, so overall, binary tree condition is preserved even after rotation.

3 Double Rotations

In some cases a "double rotation" may be needed. This happens when the node being pulled up in rotation that is a child of the node being rotated (C child of A in the above example) is heavy on the same side as the type of rotation. Heaviness is determined by which side of the node has the subtree of greater height. For a left rotation, if the node rotating up is left heavy, a right rotation before the left rotation is required and for a right rotation, if the node rotating up is right heavy, a left rotation before the right rotation is required. Without this double rotation, the tree would still be imbalanced. In the following example, a left rotation is called on node A because it is imbalanced. Applying our current left rotation gives a result that is still imbalanced.



However, by performing a right rotation on C first and then the left rotation on A to do results in a balanced tree.



Code modified to handle the second double rotation case for a left rotation is below.

```
def left_rotation_with_double(A):
    C = get_right_child(A)
    if get_right_child(C).height >= get_left_child(C).height:
        # C is right heavy
        right_child(A) = D; parent(D) = A
        right_child(C) = A; parent(A) = C
    else:
        # C is left heavy, right rotation
        D = get_right_child(C)
        X = get_right_child(D); Y = get_right_child(D)
        right_child(C) = Y; parent(Y) = C
        right_child(D) = C; parent(C) = D

        # Perform left rotation on A same as above
        ...
```

4 Overall Rotation Algorithm

Let's say that we want to balance some unbalanced node A using our rotations. The first step is to check whether it needs a left rotation or right rotation. If the left subtree of A has a greater height than the right subtree, then A needs a right rotation, otherwise it needs a left rotation. Next, we need to determine whether we need to perform a double rotation. Let C be the right node of A if we are doing a left rotation and the left node of A if it is instead a right rotation. Then, if we want to do a left rotation on A, if the left subtree of C has greater than than the right subtree, first perform a right rotation on node C, and then finally do the left rotation on A. Similarly if we want to do a right rotation on A and the if the right subtree of C has greater than than the left subtree, first perform a left rotation on node C, and then finally do the right rotation on A.

An overall psuedocode for performing rotations to balance an unbalanced node A is shown below, handling both left and right rotations at the same time.

```
def rotate(A):
    # Check for type of rotation needed
    type_of_rotation = None
    if get_left_child(A).height > get_right_child(A).height:
        type_of_rotation = left
    else:
        type_of_rotation = right

    # Check if double rotation is needed
    C = None
    double_rotation = None
    if type_of_rotation is left:
        C = get_right_child(A)
```

```
        if get_left_child(C).height > get_right_child(C).height:
            double_rotation = True
    else:
        C = get_left_child(A)
        if get_right_child(C).height > get_left_child(C).height:
            double_rotation = True

    # Rotating, rotating two times if double rotation
    if type_of_rotation is left:
        if double_rotation:
            right_rotation(C)
        left_rotation(A)
    else:
        if double_rotation:
            left_rotation(C)
        right_rotation(A)
```

5 Insertion

Insertion is relatively simple, as only the heights along the ancestral path (the path from the node to the root) of the inserted node will change. This means that only nodes along the ancestral path will ever need rotations, which we can check for and perform systematically. Note that the heights will have to changed as insertion is performed in order to properly check if the tree is imbalanced. Pseudocode for insertion is shown below.

```
def insert(value):
    # Tracking of ancestral path for rotations
    ancestral_path = []
    # Perform regular binary tree insertion while updating
    # heights and adding nodes to ancestral path
    ...

    # Checking for and performing rotations
    for node in ancestral_path:
        if node is unbalanced:
            rotate(node)
```

6 Deletion

Deletion is more complicated, as more nodes move around in the process. However, following the process of deletion, only the heights in the nodes of the ancestral path of the replacing node (the node replacing the deleted node) will be affected. This means that only the nodes along the ancestral path of the replacement node at its previous position will have to considered for rotation.

To get this path, before swapping the replacement node into its new location, save its parent into a variable and check along its ancestral path.

```
def delete(value):
    replacement_node_parent = None
    # Perform regular binary tree deletion and
    # assign replacement_node_parent when found
    ...

    ancestral_path = replacement_node_parent to root path

    # Checking for and performing rotations
    for node in ancestral_path:
        if node is unbalanced:
            rotate(node)
```