Last week we covered binary search tree and how it can be implemented in Python. To reach its full potential the search tree must be balanced all the time. With small changes we can turn BST to an AVL tree which has this property.

The node class requires the least amount changes. The class stores a new variable `balance`. The value of the balance variable is rather

- 0: the tree spanning from the node is balanced
- 1: the depth of the right subtree is greater than the left subtree by one.
- -1: the depth of the left subtree is greater than the right subtree by one.

```
class AVLNode:
    # Initialize new node
    def __init__(self, key):
        self.key = key
        self.left = self.right = None
        self.balance = 0
```

Like previously we create a separate class for AVL itself which stores the actual tree maintaining the AVL tree property. This class gets a new variable `is_balanced` which has an important role in addition process. Be careful not to mix it with the `balance` variable from `AVLNode`.

```
class AVL:
    # Initialize new tree
    def __init__(self) -> None:
        self.root = None
        self.is_balanced = True
```

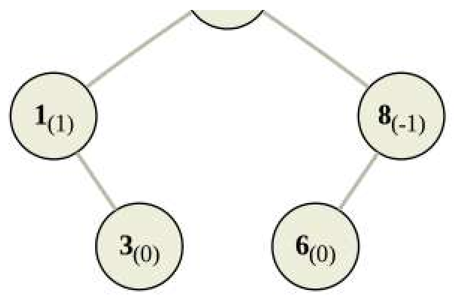

Figure 1: A Simple AVL tree

Inserting new keys follows the same principle as BST but after each recursive call we have to check if any rotations needs to be done to maintain the balance.

```
# Inserts a new key to the search tree
def insert(self, key: int):
    self.root = self.insert_help(self.root, key)


# Help function for insert
def insert_help(self, root, key):
    if not root:
        root = AVLNode(key)
        self.is_balanced = False
    elif key < root.key:
        root.left = self.insert_help(root.left, key)
        if not self.is_balanced:                          # Check for possible rotations
            if root.balance >= 0:                      # No Rotations needed, update balance variables
                self.is_balanced = root.balance == 1
                root.balance -= 1
            else:                                      # Rotation(s) needed
                if root.left.balance == -1:
                    root = self.right_rotation(root)    # Single rotation
                else:
                    root = self.left_right_rotation(root)   # Double rotation
                self.is_balanced = True
    elif key > root.key:
        root.right = self.insert_help(root.right, key)
        # ...

    return root
```

When the inserted node is the left child of the left child we must do one rotation to the right:

```
# Single rotation: right rotation around root
def right_rotation(self, root):
    child = root.left                  # Set variable for child node
    root.left = child.right            # Rotate
    child.right = root
    child.balance = root.balance = 0    # Fix balance variables
    return child
```

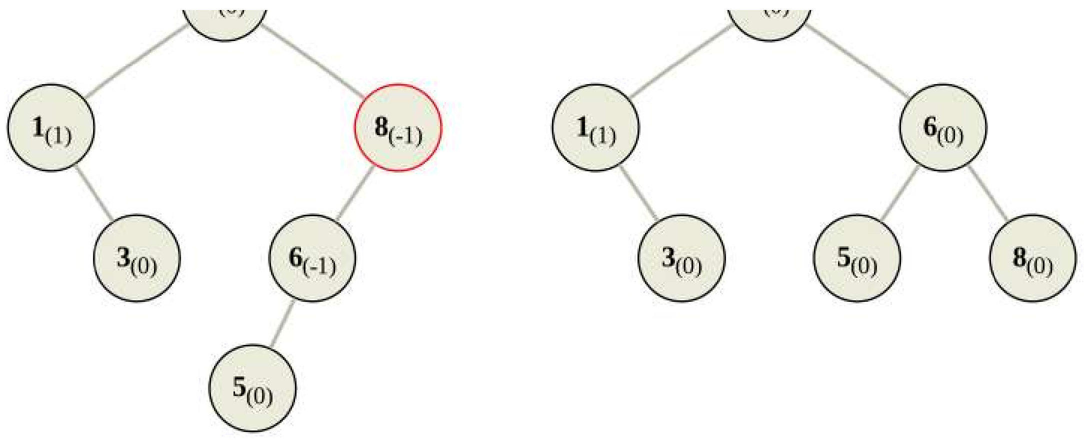

Figure 2: After adding number 5 to the AVL tree a right rotation has to be done around node 8.

When the inserted node is the right child of the left child we must do left rotation around the right child of the root node followed by right rotation around the root node. This is called left-right rotation (double rotation).

```
# Double rotation: left rotation around child node followed by right rotation around root
def left_right_rotation(self, root: AVLNode):
    child = root.left
    grandchild = child.right            # Set variables for child node and grandchild node
    child.right = grandchild.left       # Rotate
    grandchild.left = child
    root.left = grandchild.right
    grandchild.right = root
    root.balance = child.balance = 0    # Fix balance variables
    if grandchild.balance == -1:
        root.balance = 1
    elif grandchild.balance == 1:
        child.balance = -1
    grandchild.balance = 0
    return grandchild
```

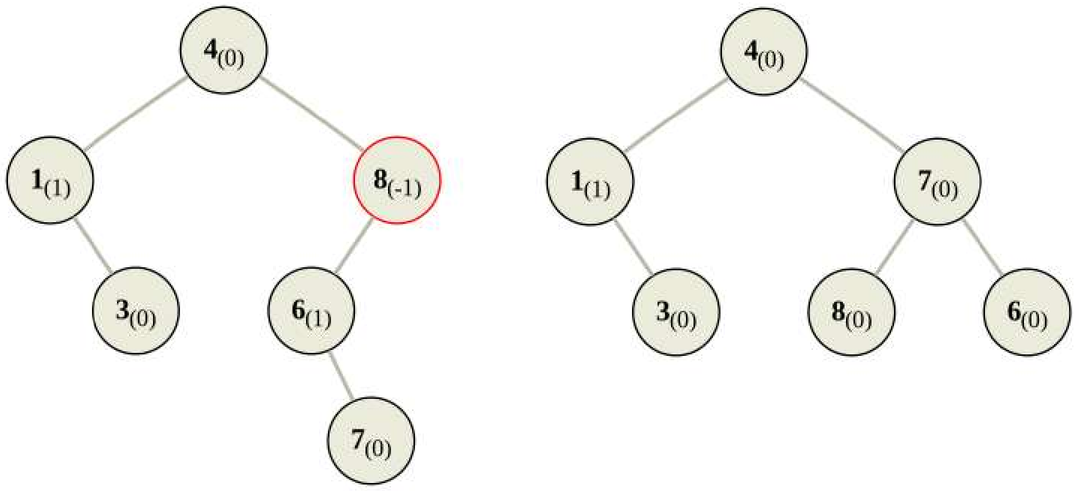

Figure 3: After adding number 6 to the AVL tree a left-right rotation has to be done around node 8.

The last two rotations are symmetrical to the presented rotations. Your task is to implement them in this week's programming assignment.

Summary: The AVL Tree

```
class AVLNode:
    # ...


class AVL:
    # ...


if __name__ == "__main__":
    Tree = AVL()
    for key in [9, 10, 11, 3, 2, 6, 4, 7, 5, 1]:
        Tree.insert(key)
    Tree.preorder()     # 9 4 2 1 3 6 5 7 10 11
```

The output

```
$ python3 AVL.py
9 4 2 1 3 6 5 7 10 11
```

Last modified: Wednesday, 26 June 2024, 2:32 PM