Recitation 6

AVL Trees

AVL trees are **balanced binary search trees**: BSTs on n keys that support the normal BST operations, while guaranteeing the height of the tree is always $O(\log n)$. The goal of this recitation is to review data structure augmentation, as well as rotations which can be used to maintain balance in the BST. An AVL tree guarantees $O(\log n)$ height by maintaining an even stronger invariant on the **skew** of a node: the height of its right subtree minus the height of its left subtree. An AVL tree is then a binary tree where each node satisfies the following property.

> **AVL Property:** For every node in an AVL tree, the magnitude of skew is at most one.

Exercise: Show that the height of any AVL tree on n nodes is $O(\log n)$.

Solution: We show that every AVL tree with height h contains at least $n = \Omega(2^{h/2})$ nodes, so $h = O(\log n)$. Let $T(h)$ be the minimum number of nodes in an AVL Tree; this minimum will be achieved in a tree where left and right subtrees of the root have heights that differ by one, and are also minimal, i.e., $T(h) = 1 + T(h-1) + T(h-2)$. Since $T(h)$ increases with h, then $T(h) \geq 2T(h-2)$. We show that $n \geq T(h) \geq 2^{h/2} = \Omega(2^{h/2})$ by induction. As a base case, the only tree with height 0 contains one node, $T(0) = 1 \geq 2^{0/2} = 1$, and the claim holds. Alternatively, $2^{h/2} \geq 2(2^{(h-2)/2}) = 2^{h/2}$, proving the claim. □

Data Structure Augmentation

To speed up and simplify our ability to evaluate the AVL Property at each node, we will ask each node to store and maintain its own height and skew. Augmentation is a common technique for customizing a data structure to fit your application.

```
class AVL(BST):
    def __init__(self, item = None, parent = None):
        super().__init__(item, parent)
        self.height = 0
        self.skew = 0
```

If we've already calculated the height and skew of the descendants of a node, it is straight forward to calculate the height and skew of the node itself. The height is one more than the maximum height of its children, while the skew is simply the difference in height.

```
def _update(self):
      left_height  = self.left.height  if self.left  else -1
      right_height = self.right.height if self.right else -1
      self.height  = max(left_height, right_height) + 1
      self.skew    = right_height - left_height
```

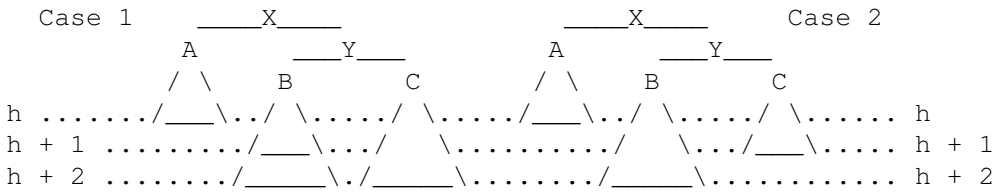
Exercise: Draw a BST and compute some subtree properties at all nodes from the leaves to the root. Example properties: subtree minimum, subtree height, number of even (or odd, or prime) keys in subtree, etc.

Maintenance

Suppose we have a BST that satisfies the AVL Property. Inserting or deleting a single node in the tree can only change the height of any other node by one. Which nodes could change height? Any ancestor of the node that was inserted or deleted. Let's delegate the responsibility of maintaining subtree properties at ancestors to the inserted node, or to the parent of the node we deleted. This is the node on which our BST code called the empty stub function `maintain`.
Because any ancestor of the delegated node can change height, it might also violate the AVL Property. Since the AVL Property is calculated from child properties, our approach will be to repeatedly balance ancestors from the delegated node to the root during recursive calls to `maintain`. An inserted node can't have any children so it trivially satisfies the AVL Property. Alternatively the parent of a deleted node might have one child, but the child's height could not have been affected by the deletion so it also satisfies the AVL Property. In either case, we are sure that all descendants of the node that we delegated to balance the tree satisfy the AVL Property. So we recursively `maintain` and balance its ancestors!

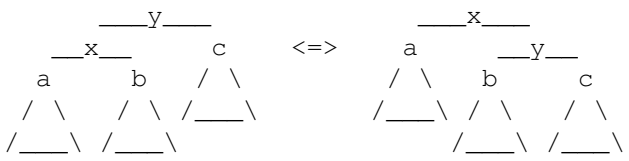
Rotations

Suppose we have a BST node whose descendants all satisfy the AVL Property, but the node itself does not. How can we balance the tree? Because heights change by at most one after the dynamic operation, any node that does not satisfy the AVL Property has skew of magnitude at most two. Locally, such a node `X` must look like one of the following two cases: either `Y` is (not skewed or skewed in the same direction as `X`), or `Y` is skewed in the opposite direction.



How can we balance these trees? Rotations! A rotation locally rearranges nodes in a way that maintains the BST Property.

Going from the left picture to the right picture below is a **right rotation** on y, while going from the right to the left is a **left rotation** on x. As you can see, a right rotation moves the a subtree up and moves the c subtree down, while keeping b at the same height.



To balance Case 1, all we need to do is perform a rotation at X in the direction that balances the skew offset. In the above example, a left rotation lowers A and raises C which balances the tree. Case 2 is a little more complicated: perform a rotation to reverse the skew of Y, followed by a rotation to balance the skew at X[1]. In the above example, a right rotation raises B and lowers C, and then a left rotation at X balances the tree. In either case, we reestablish the AVL property while maintaining the BST Property. So to balance the whole tree, we first update the height and skew of the node that we're at, balance the node using either one or two rotations, and then repeat the process at all ancestors, in order towards the root.

```
def _maintain(self):
    self._update()              # recompute subtree properties
    if self.skew == 2:          # must have right child
        if self.right.skew == -1:
            self.right._right_rotate()
        self._left_rotate()
    elif self.skew == -2:       # must have left child
        if self.left.skew == 1:
            self.left._left_rotate()
        self._right_rotate()
    if self.parent:
        self.parent._maintain()
```

From a coding standpoint, rotations are simple but tedious. Simply re-link the necessary pointers and update the heights of X and Y. Before showing them the answer, ask students to try to figure out all the pointer manipulations that need to happen for a single rotation. Note that in order to maintain the root node, we avoid moving the top node, swapping items instead.

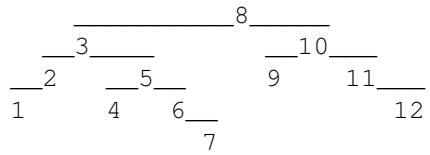
[1]Some additional analysis is needed to confirm that these two rotations actually lead to a tree that satisfies the AVL Property. We don't detail the argument here, but the approach is to do a case analysis on the children of B. The tricky case is when the first rotation on Y creates a violation of the AVL Property at Y. Fortunately if that happens, the second rotation at X will always fix it.

```
def _right_rotate(self):
    node, c = self.left, self.right
    a, b    = node.left, node.right
    self.item, node.item = node.item, self.item
    if a:   a.parent = self
    if c:   c.parent = node
    self.left, self.right = a, node
    node.left, node.right = b, c
    node._update()
    self._update()
```

```
def _left_rotate(self):
    a, node = self.left, self.right
    b, c    = node.left, node.right
    self.item, node.item = node.item, self.item
    if a:   a.parent = node
    if c:   c.parent = self
    self.left, self.right = node, c
    node.left, node.right = a, b
    node._update()
    self._update()
```

What are AVL trees good for? Maintaining a **dynamic order**. If you just have a fixed set of items, you can't do much better than storing the items in a simple sorted array. Alternatively, if you just want to add and delete items without maintaining order, a hash table is the way to go (next week!). But if you need your items in order *and* want to support a constantly changing set of items, balanced binary search trees are the way to go.

Prepared Exercise: An example operation requiring three rotations during maintenance deletes key 9 from the following tree. Confirm that each node in the tree satisfies the BST and AVL properties, then delete 9 from the tree, and update nodes, performing any necessary rotations up the tree. (**Solution:** Left rotation at 10, left rotation at 3, right rotation at 8).



Interactive Exercise: Make a BST by inserting student chosen keys one by one. If the tree ever does not satisfy the AVL Property, re-balance its ancestors going up the tree.

We've made a CoffeeScript BST/AVL visualizer which you can find here:
`https://codepen.io/mit6006/pen/NOWddZ`