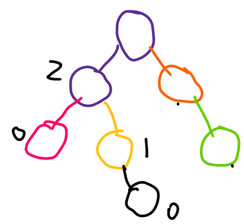
AVL Tree

- AVL tree is type of balanced binary search tree, it is named after its inventors Adelson-Velsky and Landis.
- An AVL tree satisfies **AVL property**: For each node x in a binary tree, the height of the subtree rooted at $x.left$ and the height of the subtree rooted at $x.right$ can differ by at most 1.

Before proving the upper bound of tree height, let's draw some small AVL trees with n nodes where $n = 3, 4, 5, 6, 7$.



Let's prove that when AVL property is satisfied in a binary search tree, then this binary search tree is balanced. We prove the following claim:

[Claim] A binary tree that satisfies AVL property of height h contains at least $\sqrt{2}^h$ nodes. $= n$

- We prove by induction on h:
- Base cases: when $h = 0$, then a binary tree has exactly 1 node, and $1 = \sqrt{2}^0$; and when $h = 1$, the binary tree has at least 2 nodes, and $2 > \sqrt{2}^1$.
- Induction hypothesis is that for $h = 0, \ldots, k$, the statement holds.
- Induction step: In a binary tree of height $k + 1$, we know that at least one of its left subtree and right subtree must have height k; and by the AVL property the other subtree must have height k or $k - 1$. Then this tree of height $k + 1$ contains at least this many nodes:
$$1 + \sqrt{2}^k + \sqrt{2}^{k-1} > \left(1 + \sqrt{2}\right) \cdot \sqrt{2}^{k-1} > 2 \cdot \sqrt{2}^{k-1} = \sqrt{2}^{k+1}$$

- Using the above claim, we have that an AVL tree with n nodes has height at most $\log_{\sqrt{2}} n$. In other words, a binary tree that satisfies AVL property has height $O(\lg n)$.

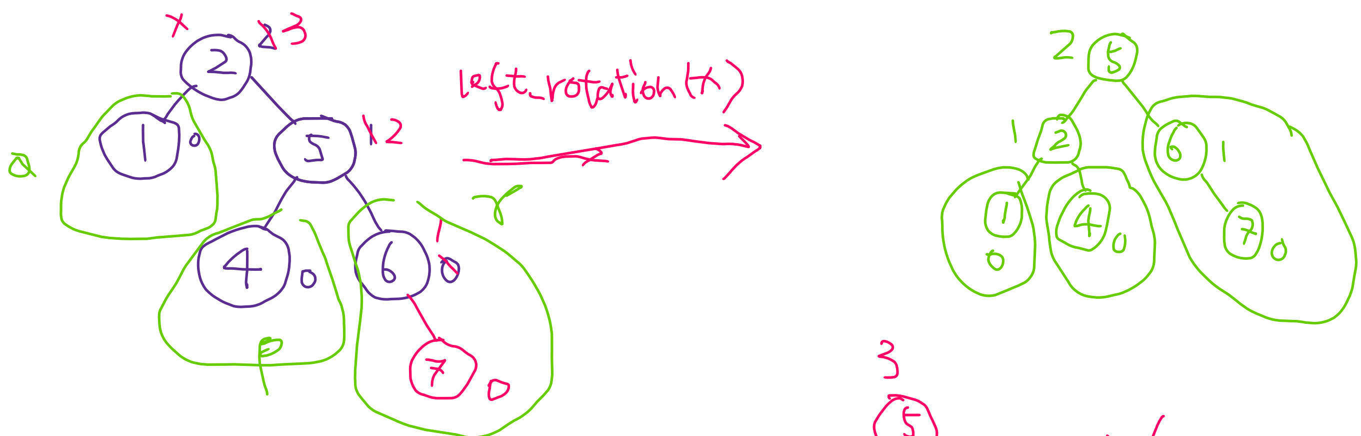
Insertion and Deletion in an AVL Tree

- Remind that, an AVL tree is a binary search tree. So, for those methods in an AVL tree that do not change the shape of the tree, we implement them in the same way as in a regular binary search tree: such as **tree_minimum**, **height**, **inorder_tree_walk, tree_search**…
- There are only two methods that change the shape of an AVL tree: insertion and deletion, so we need to update these two methods so that the AVL property is maintained after calling them.

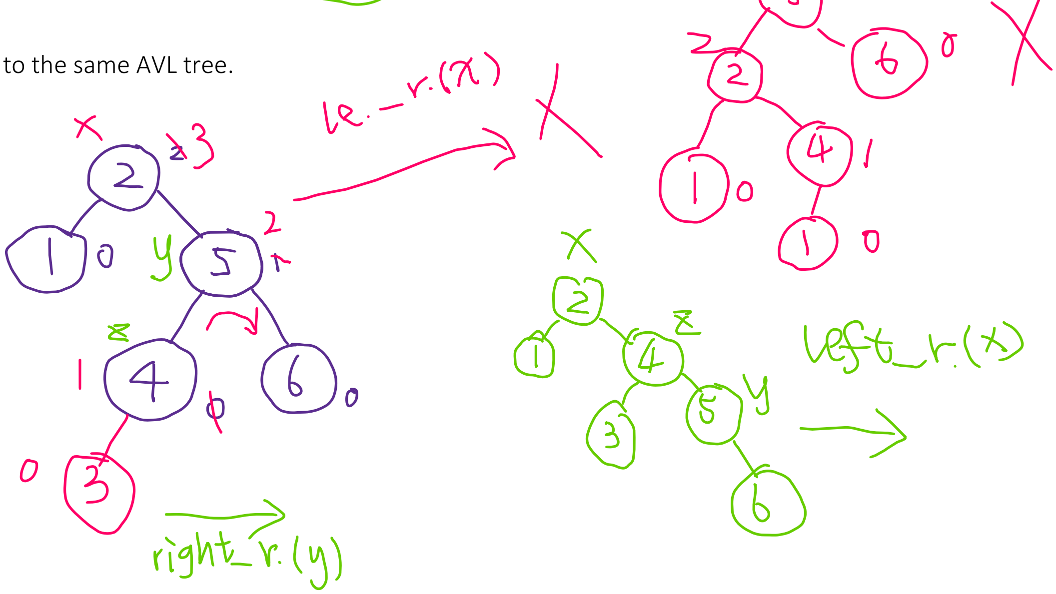
- For both methods, the algorithm is actually quite simple: *insert and delete as a regular binary search tree, then fix AVL properties using rotations*. Before looking into this algorithm, let's look at some examples first. https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

- In the above example, we can see that after each insertion and deletion, we update the heights of **all nodes between the inserted/deleted nodes and root**, then if there is some node x on this path violates AVL property, (x has $height(x.left) - height(x.right) > 1$ or < -1) we rotate at x accordingly.

1. Insert 7 to the following AVL tree.

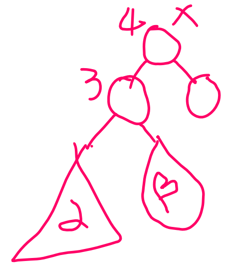


2. Insert 3 to the same AVL tree.



- From the above examples, we see that when we use $\textbf{left\_rotation}(x)$ to fix AVL property at node x, we need to make sure that $x.right$ has a "larger" right child, or else we need to call $\textbf{right\_rotation}(x.right)$ first.
  1) Similarly, when we use $\textbf{right\_rotation}(x)$ to fix AVL property at node x, we need to make sure that $x.left$ has a "larger" left child, or else we need to call $\textbf{left\_rotation}(x.right)$ first.

- We get the following method $\textbf{rebalance}(x)$ to fix AVL property at node x.



balance(x)
return height$(x.left)$ − **height**$(x.right)$

rebalance(x)
if balance$(x) > 1$ and **balance**$(x.left) \geq 0$:
  right_rotation(x)
if balance$(x) > 1$ and **balance**$(x.left) \leq 0$:
  left_rotation$(x.left)$
  right_rotation(x)
if balance$(x) < -1$ and **balance**$(x.left) \leq 0$:
  left_rotation(x)
if balance$(x) < -1$ and **balance**$(x.left) > 0$:
  right_rotation$(x.right)$
  left_rotation(x)

- In either case of **rebalance**, there are at most two rotations involved, so its running time is $O(1)$.

- Insertion and deletion in an AVL tree can be done as follows:

AVL_tree_insertion $(T, item)$
1 **tree_insertion** $(T, item)$ $//O(h) = O(\lg n)$
2 let z be the inserted node
3 **for** each $node$ from z to $T.root$ $//O(h) = O(\lg n)$ iterations
4   update $height(node)$ $//\ O(1)$?
5   **rebalance**$(node)$ $//O(1)$

AVL_tree_deletion $(x, item)$
1 **tree_deletion** $(x, item)$
2 let z be the node that is deleted
3 **for** each $node$ from z to x
4   update $height(node)$
5   **rebalance**$(node)$

- In both methods, line 1 takes $O(\lg n)$ time since an AVL tree has height $\Theta(\lg n)$, the for loop in line 3 runs $O(\lg n)$ times and it takes only constant time in each iteration. Thus, both methods have time complexity $O(\lg n)$.

AVL tree implementation

Let's implement AVL tree by updating the Python code for the Binary Search Tree class. Here I only kept methods that are related to our design.

- I will make the following updates:
  1) Since we need to use the height nodes very frequently, it makes no sense to recalculate this value all the time (especially the method $height(x)$ take $\Theta(n)$ time). We will make $height$ as an attribute of a node. Since we use the default construction method of a node all the time, we don't allow users to set left child

and right child of a new node anymore. And since we keep *height* as an attribute now, we need to maintain the value all the time (after rotation, insertion, deletion)

2) Since we need to calculate the *height* and *balance* for **None** node frequently, we need to create static methods to calculate height and balance even if we input a **None** node.
3) Implement method **rebalance** following the pseudo-code above.
4) Call **rebalance**(x) in rec_insert(x) and rec_delete(x).

```
def get_height(x):
        if x is None:
                return - 1
        else:
                return x.height

def get_balance(x):
        if x is None:
                return 0
        else:
                return get_height(x.left) - get_height(x.right)


class ~~BinarySearchTree~~ AVLTree:

    class Node:

        def __init__(self, val~~, left = None, right = None~~)
                self.val = val
                self.left = ~~left~~ None
                self.right = ~~right~~ None
                self.height = 0

        def left_rotation(self):
                b = self.right

                alpha = self.left
                beta = b.left
                gamma = b.right

                b.left = alpha
                b.right = beta
                self.left = b
                self.right = gamma

                self.val, b.val = b.val, self.val
                b.height = 1 + max(get_height(alpha), get_height(beta))
                self.height = 1 + max(get_height(b), get_height(gamma))

        def right_rotation(self):
                #similar to left_rotation()


    def __init__(self, root = None):
        self.root = root

    def height(self):
```

```
        ~~def rec_height(x: BinarySearchTree.Node):~~
                ~~if x is None:~~
                        ~~return −1~~
                ~~else:~~
                        ~~return 1 + max (rec_height(x.left), rec_height(x.right))~~

        return ~~rec_height(self.root)~~   self.root.height


def rebalance(x):
        if get_balance(x) > 1 and get_balance(x.left) ≥ 0:
                x.right_rotation()
        elif ...


def __contains__(self, item):

        def rec_contains(x):
                if x is None:
                        return False
                elif x.val == item:
                        return True
                elif item < x.val:
                        return rec_contains(x.left)
                else:
                        return rec_contains(x.right)

        return rec_contains(self.root)


def insert(self, item):

        def rec_insert(x):
                if x is None:
                        x = BinarySearchTree.Node(item)
                elif item < x.val:
                        x.left = rec_insert(x.left)
                else:
                        x.right = rec_insert(x.right)

                # update height of x
                # call rebalance(x)

                return x

        self.root = rec_insert(self.root)


def __delitem__(self, item):

        def rec_delete(x):

                if item < x.val:
                        x.left = rec_delete(x.left)
```

```
        elif item > x.val:
            x.right = rec_delete(x.right)
        else:
            if x.left is None and x.right is None:
                x = None
                return x
            elif x.left is None:
                x = x.right
            elif x.right is None:
                x = x.left
            else:
                y = x.right
                while y.left is not None:
                    y = y.left
                x.val, y.val = y.val, x.val
                x.right = rec_delete(x.right)


        # update height of x
        # call rebalance(x)

        return x

    assert item in self
    self.root = rec_delete(self.root)
```