

R7 Balanced Binary Tree

3.24 https://github.com/GUMI-21/MIT6.006_note

a tree on n nodes is balanced if its height is $O(\log n)$. Then all the $O(h)$ -time operations we talked about last time will only take $O(\log n)$ time.

(Red-Black Trees, B-Trees, 2-3 Trees, Splay Trees, etc.) The oldest (and perhaps simplest) method is called an AVL Tree.

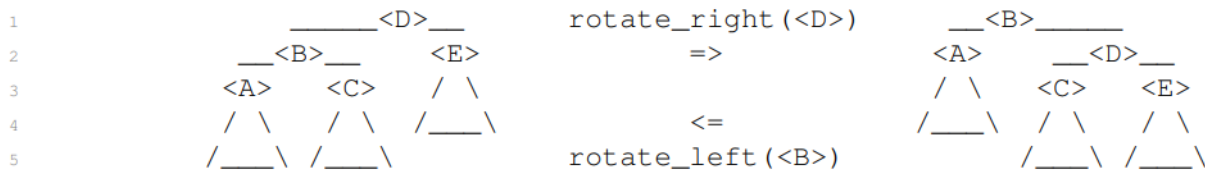
- *skew* of a node

its right subtree minus the height of its left subtree.

Then a node is height-balanced if it's skew is either -1 , 0 , or 1 .

A tree is height-balanced if every node in the tree is height-balanced. Height-balance is good because it implies balance!

Rotations



A rotation takes a subtree that locally looks like one the following two configurations and modifies the connections between nodes in $O(1)$ time to transform it into the other configuration.

- rotate_right(D):
left_child.right -> goto node.left

```
def subtree_rotate_right(D):
    if D is None or D.left is None:
        return D # can't rotate
    left = D.left
    lc_r = left.right

    # rotate
    D.left = lc_r
    if lc_r: lc_r.parent = D

    left.right = D
    left.parent = D.parent
    if D.parent:
        if D.parent.left is D: D.parent.left = left
```

```
        else: D.parent.right = left
    D.parent = left
    return left
```



- rotate_left(D):

```
def subtree_rotate_left(D):
    if D is None or D.right is None: return D
    right = D.right
    rc_l = right.left
    #rotate
    D.right = rc_l
    if rc_l: rc_l.parent = D
    right.left = D

    right.parent = D.parent
    # update ancestor's parent
    if D.parent:
        if D.parent.left is D:
            D.parent.left = right
        else:
            D.parent.right = right
    D.parent = right
    return right
```



Maintaining Height-Balance

think adding or removing a leaf from a AVL tree. -> the only nodes in the tree whose subtrees have changed after the leaf modification are ancestors of that leaf (at most $O(h)$ of them)

- *Rebalance in AVL*
algorithm see in lec7 last

```
def skew(A):
    return height(A.right) - height(A.left)

def rebalance(A)
    if A.skew() == 2: # right child higher
        if A.right.skew() < 0: # rc.lc > rc.rc
            A.right.subtree_rotate_right()
        A.subtree_rotate_left()
    elif A.skew() == -2: # left child higher
        if A.left.skew() > 0: # lc.rc > lc.lc
            A.left.subtree_rotate_left()
        A.subtree_rotate_right()
```

```
def maintain(A):
    A.rebalance()
    A.subtree_update()
    if A.parent: A.parent.maintain() # rebalance A.ancestors
```

- if don't maintain node.hight at each node, there will cost $\Omega(n)$ time to count hight of every node.

```
def height(A): # omega(n)
    if A is None: return -1
    return 1 + max(height(A.left), height(A.right))
```

- *so we need to store & maintain subtree augmentation*
when the structure of the tree changes, we will need to update and recompute the height at nodes whose height has changed.

```
def height(A)
    if A: return A.hight
    else: return -1
def subtree_update(A)
    A.height = 1 + max(height(A.left), height(A.right))
```

then in dynamic operations, calls subtree_update in every functions.

To augment the nodes of a binary tree with a subtree property $P(x)$, you need to:

- clearly define what property of 's subtree corresponds to $P()$, and
- show how to compute $P(x)$ in $O(1)$ time from the augmentations of 's children.

```
def maintain(A): #  $O(\log n)$ 
    A.rebalance()
    A.subtree_update()
    if A.parent:
        A.parent.maintain()
```

all AVL tree code see *Binary Node Implementation with AVL Balancing* (the summary of R6&R7 codes)

https://ocw.mit.edu/courses/6-006-introduction-to-algorithms-spring-2020/resources/mit6_006s20_r07/

Application: Sequence

To use a Binary Tree to implement a Sequence interface, we use the traversal order of the tree to store the items in Sequence order.