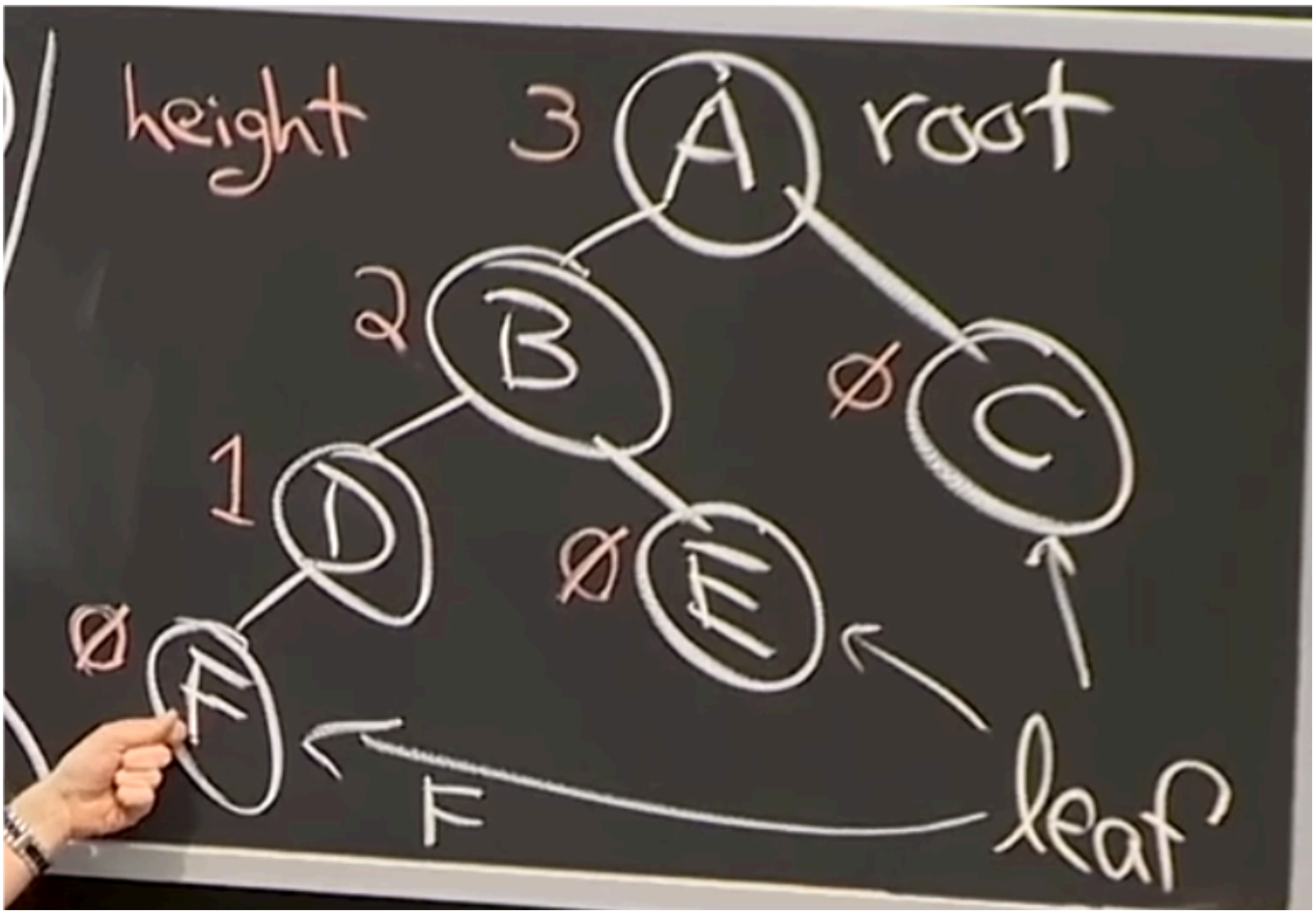
LEC7 AVL

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

-subtree augmentation
-sequence binary tree via subtree sizes
-height balance(AVL) via height augmentation
-rotations -rebalancing -sorting

Recall



-traversal order F.D.B.E.A.C
-subtree insert, delete.first/last. pred./succassor

- *subtree_find(node.k):*
  -if node is none: return
  -if k<node.item.key: recurse on node.left
  -if =: return node
  -if >: recurse on node.right

BST property

binary search tree

- traversal order = increasing key
- keys in subtree(node.left) < node.item.key < keys in subtree(node.right)

Application: Sequence binary tree

- traversal order = sequence order
- *size of node* = number nodes in subtree(node), include self
- subtree_at(node.i):
  -$n_L = size(node.left)$
  -if i < n_L: subtree_at(node.left.i)
  -if i = n_L: return node
  -if i > n_L: subtree_at(node.right,i-nL-1)
  runtime: O(h)

Subtree augmentation

subtree property: if change a node's property, only the subtree property of this node's subtree change. if the subtree which this node can't touch then nothing change.
-each node can store O(1) extra fields/property
-*subtree property node* can be computed from property of node's children (and node) in O(1) time
-size of node is a property. node.size = node.left.size + node.right.size + 1

- insert / delete
  in end: add or remove a leaf of tree
  -> update O(h) ancestors in order up the tree.

Example of subtree properties:

-sum, product, min, max of some feature of every node in subtree
-Not node's index, depth
don't use global properties of the tree
subtree properties is the only thing allowed to maintain in a tree. it costs less

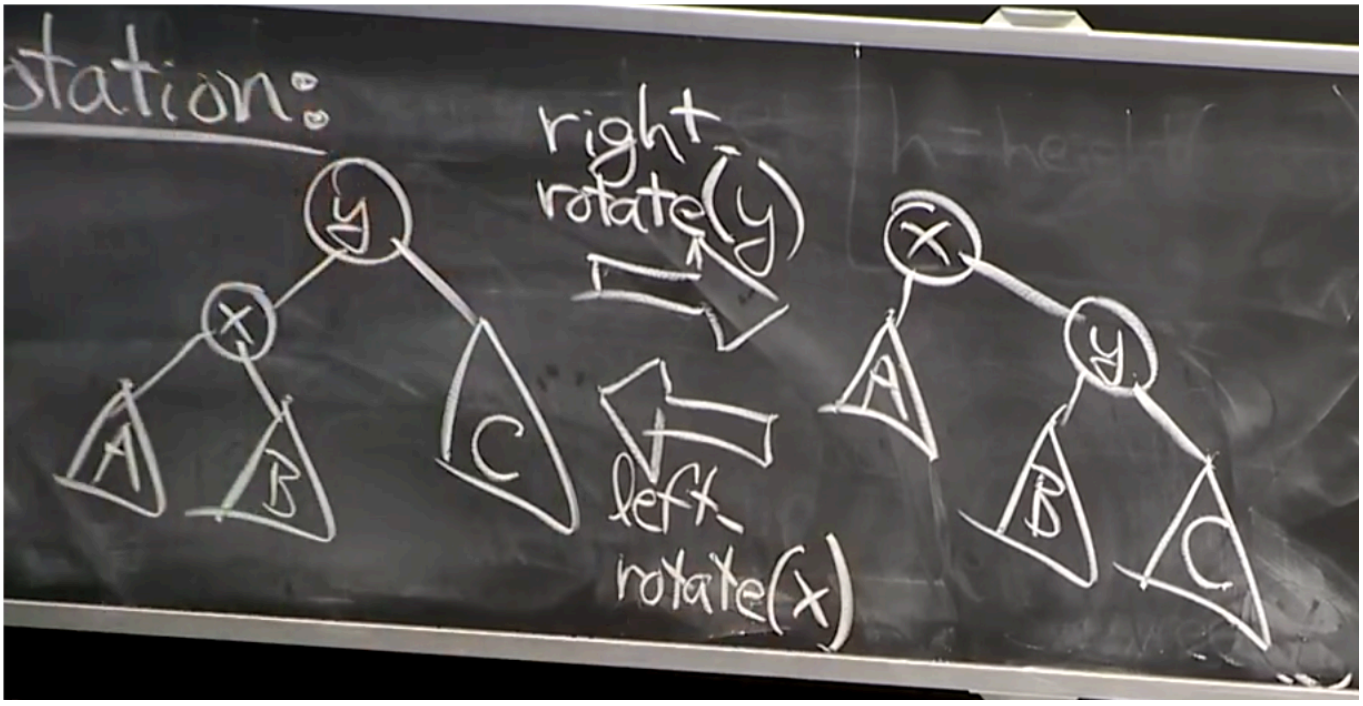
O(h) -> O(logn)

Sequence Data Structure	Operations $O(\cdot)$				
	Container	Static	Dynamic		
	build(X)	get_at(i) set_at(i,x)	insert_first(x) delete_first()	insert_last(x) delete_last()	insert_at(i, x) delete_at(i)
Binary Tree	n	h	h	h	h
AVL Tree	n	$\log n$	$\log n$	$\log n$	$\log n$

Set Data Structure	Operations $O(\cdot)$				
	Container	Static	Dynamic	Order	
	build(X)	find(k)	insert(x) delete(k)	find_min() find_max()	find_prev(k) find_next(k)
Binary Tree	$n \log n$	h	h	h	h
AVL Tree	$n \log n$	$\log n$	$\log n$	$\log n$	$\log n$

if h = O(logn): balanced binary tree

Rotation:



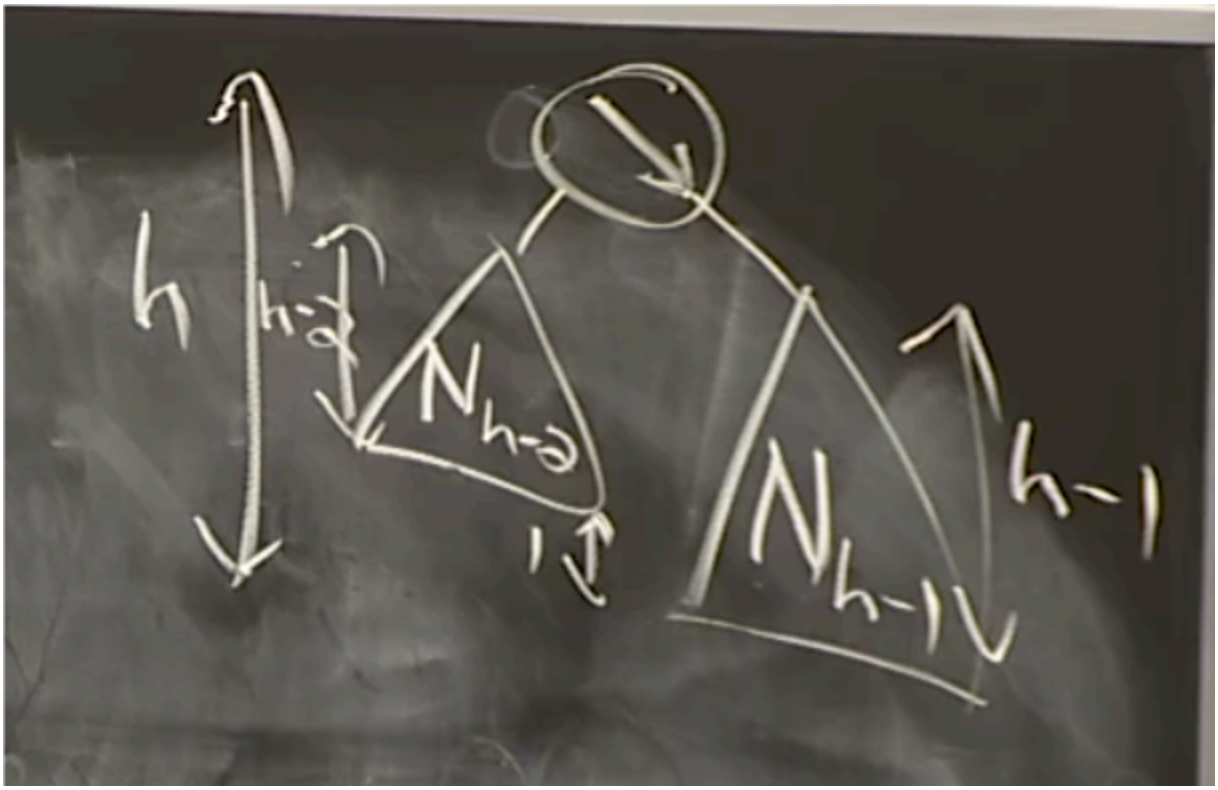
The traversal order is constant.
A X B Y C.

AVL Tree

maintain height balabce: skew(node) -> height(node.right) - height(node.left) $\in$ {-1,0,1}

Proof Height balanced => balanced

$N_h = N_{n-1} + N_{n-2} + 1 > N_{h-2} + N_{h-2} = 2N_{h-2} = 2^{h/2} (from\ recurse) => h <= 2 \log n$



How to maintain the height balance property

height is a subtree property

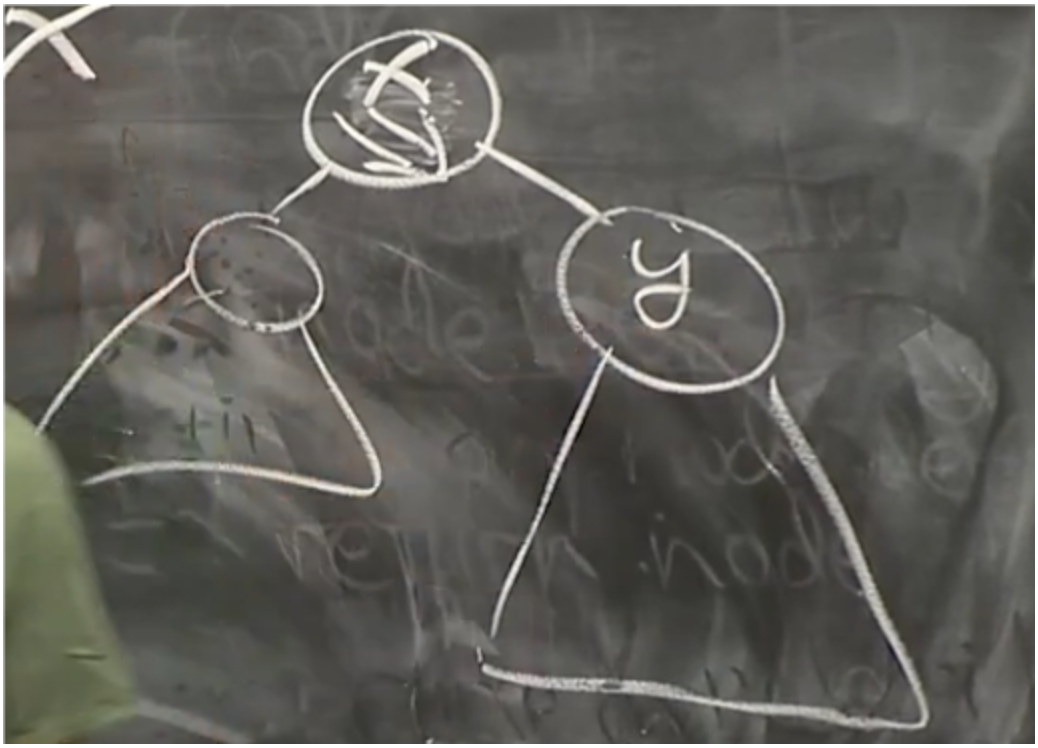
node.height = 1+max{node.left.height, node.right.height}

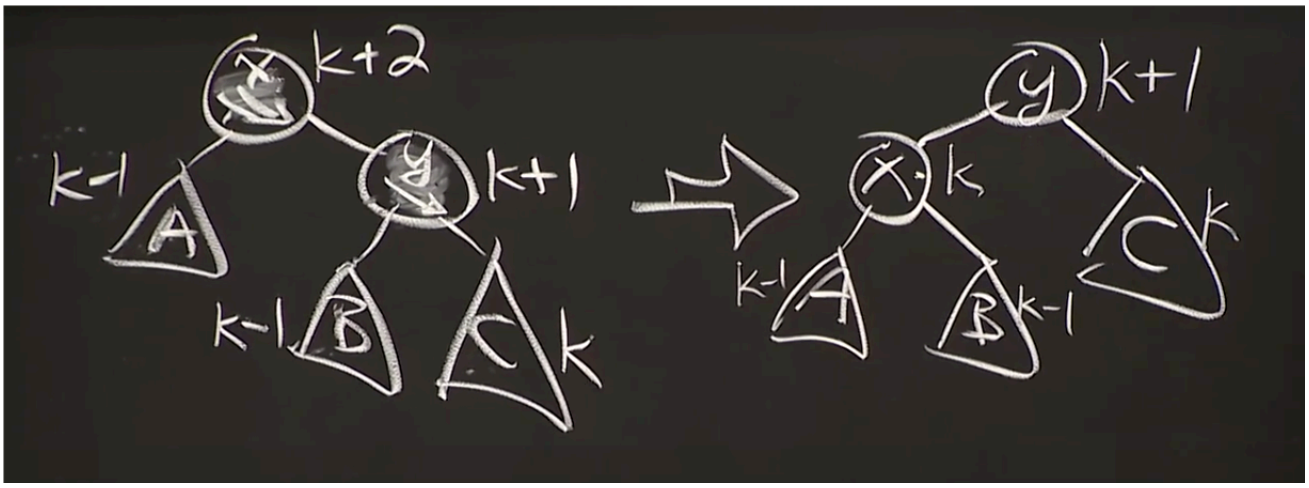
when do retation also have to update the subtree property

->when add a node or delete a ndoe from leaf, check all the ancestors of nodes in sequence and find on out of balance

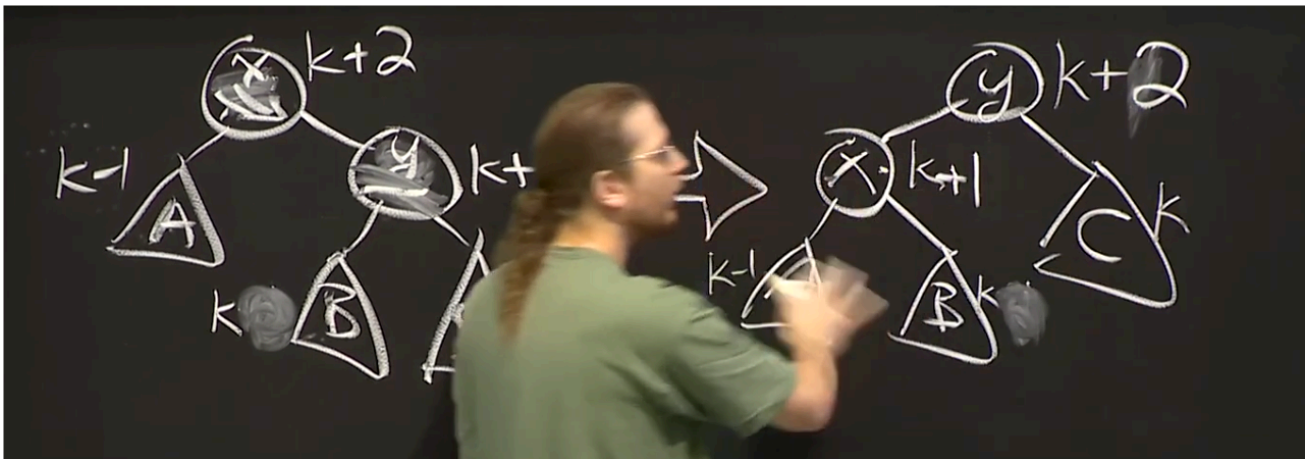
-consider lowest unbalanced node x

=> skew $\in$ {+2, -2}, say 2



- easy case *case 1* & *case 2*
  skew(y) = 1 or skew(y) = 0. => right.height - left.height = 0 or 1
  case1: do rotation
  

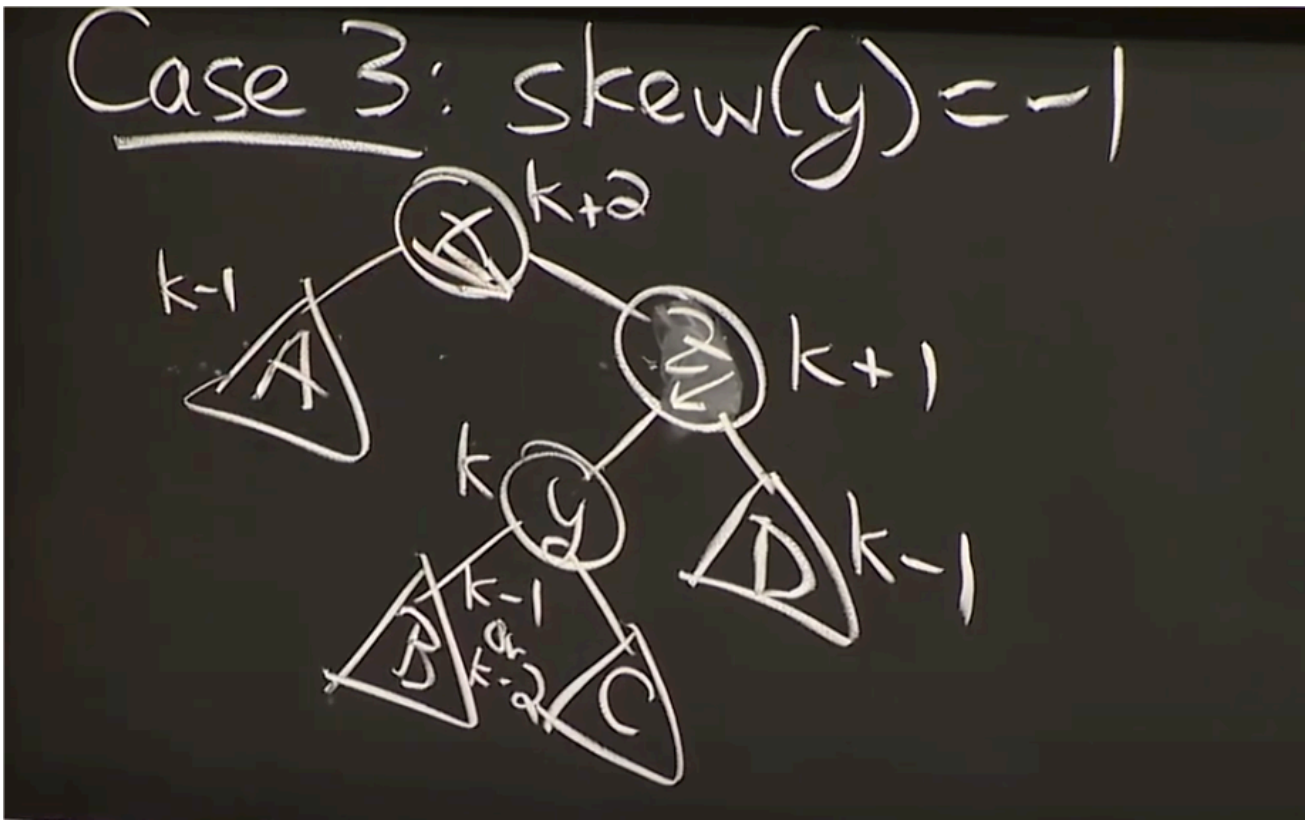
  case 2
  upper grap: y is k+1 and B is K => rotate => y is k + 2, x is k + 1



still balanced

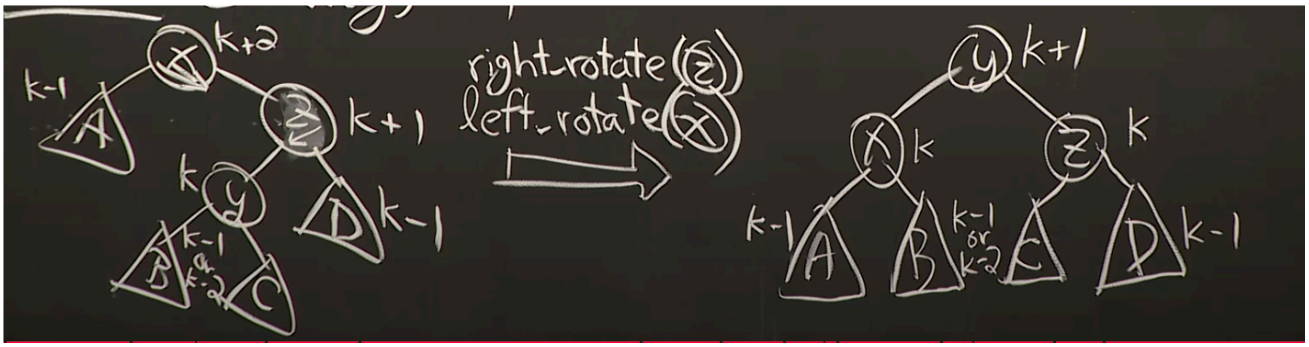
right subtree higher => right subtree.right subtree higher => left rotate right subtree.
left subtree higher => left subtree.right subtree higher =>right rotate left subtree

- hard case *case 3. when skew(y) = -1* => right.height - left.height = -1



height of A is lower 2 than Z

right_rotate(z) & left_rotate(x)



right subtree higher => right subtree.left subtree higher => right rotate right_subtree first then left rotate root.
left subtree higher => left subtree.left subtree higher => right rotate left_subtree first then right rotate root.

```
When add or delete a node from leaf =>
if (root.rc.h > root.lc.h) {  // 右子树比左子树高
    if (root.rc.rc.h > root.rc.lc.h) {  // 右子树的右子树高（RR情况）
        left_rotate(root);
    } else if (root.rc.lc.h > root.rc.rc.h) {  // 右子树的左子树高（RL情况）
        right_rotate(root.rc);
        left_rotate(root);
    }
} else if (root.lc.h > root.rc.h) {  // 左子树比右子树高
    if (root.lc.lc.h > root.lc.rc.h) {  // 左子树的左子树高（LL情况）
        right_rotate(root);
    } else if (root.lc.rc.h > root.lc.lc.h) {  // 左子树的右子树高（LR情况）
        left_rotate(root.lc);
        right_rotate(root);
    }
}
```