WARMUP

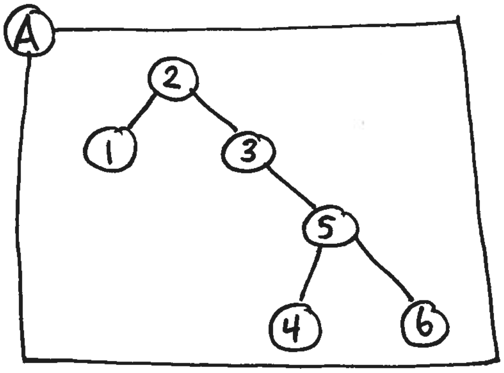
R04

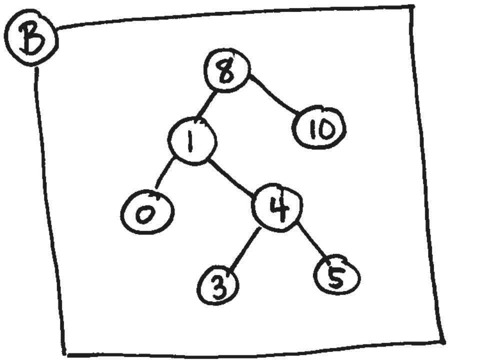
Playing with AVL trees: http:// www.site.uottawa.ca/~stan/csi2514/applets/av
... /BT.h

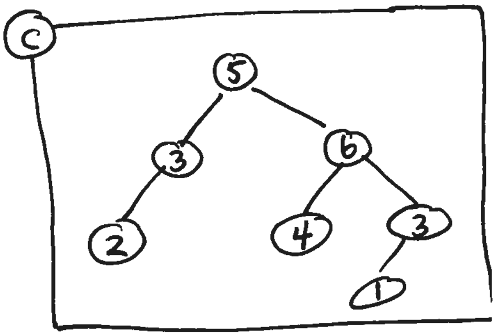
Rotations: [CLRS p 277-279] for psuedocode.

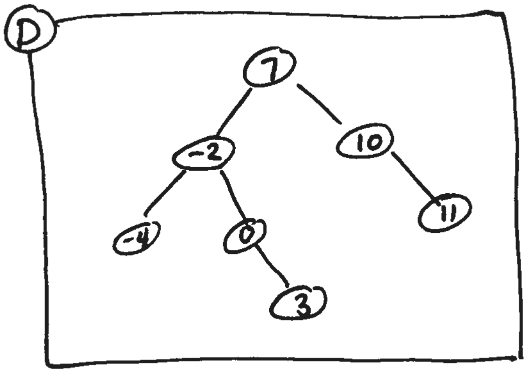
Determine if each of the following is ① a valid BST ② maintains the AVL ~~invar~~ invarient

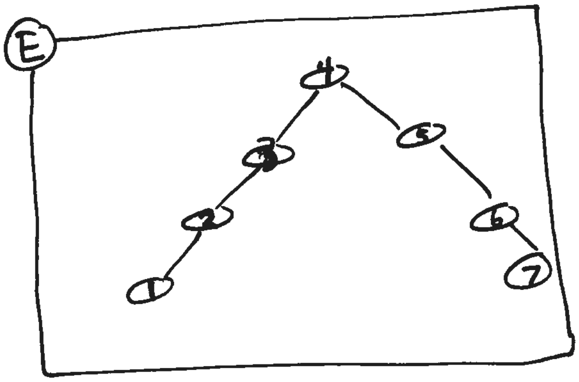
1 and label the nodes w/ their height, indicate where AVL violation is

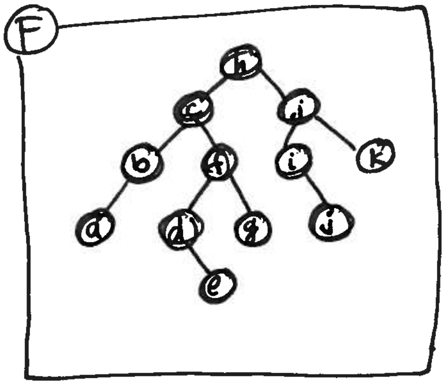








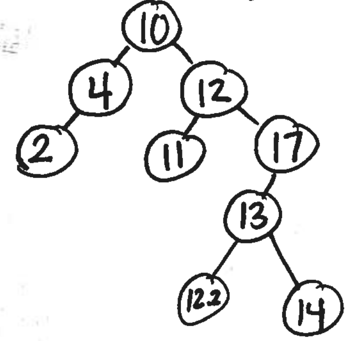




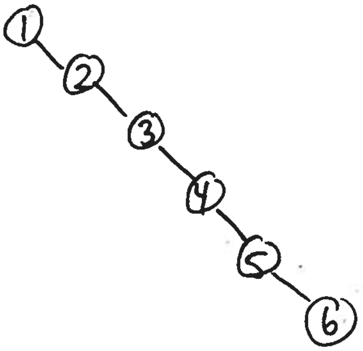
Node deletion in a BST - no need to balance!

2

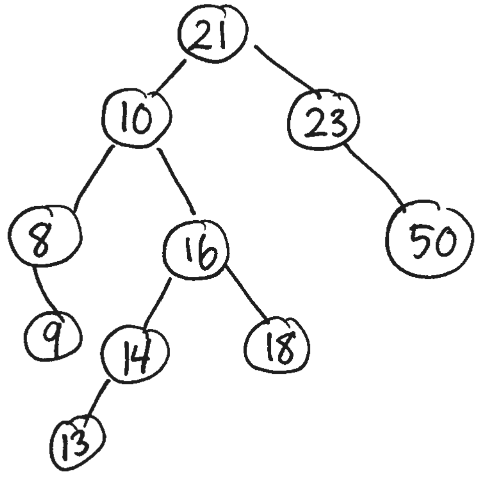
A delete 11, del 17



B delete 3



C delete 10



Recitation 4

Agenda

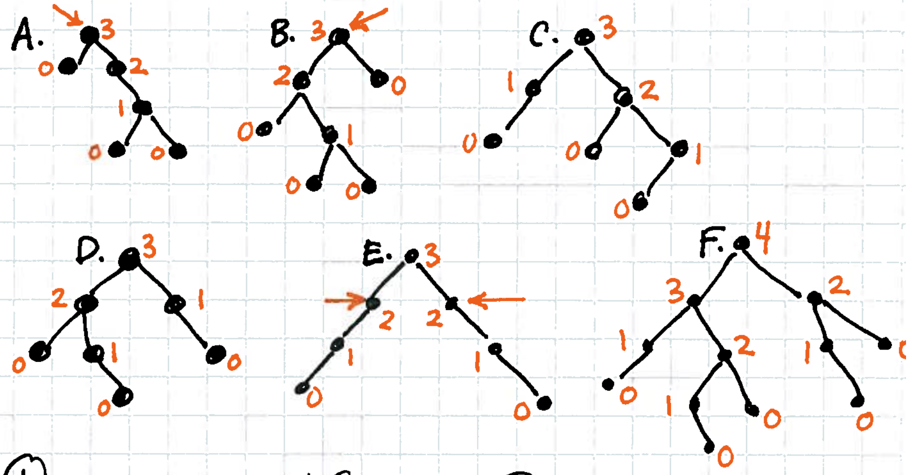
- overhead
- warmup
- AVL
  - recap lecture
  - rotation + rebalance
  - excersizes

Reminders

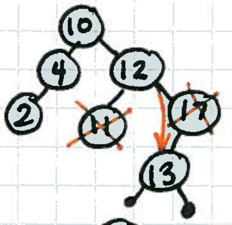
- notes + code posted & will be in future ← there were bugs in code - fixed.
- pset 1 : idea for timing, timeit
- run test code!

Warmup Answers

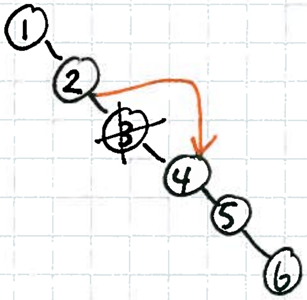
		BST	AVL
1.	A.	✓	✗
	B.	✓	✗
	C.	✗	✓
	D.	✓	✓
	E.	✓	✗
	F.	✓	✓

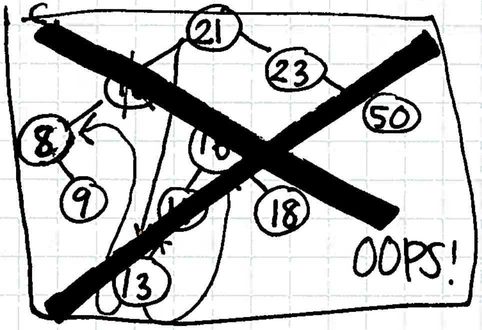


2. A.

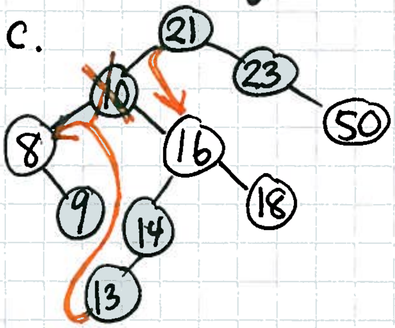


B





C.

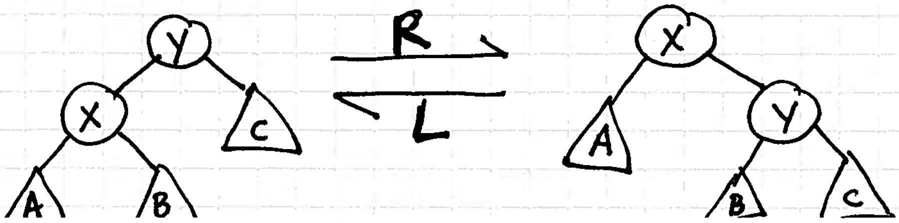


AVL trees

idea: balance trees are ☺. pay extra constant to keep balanced to maintain good order of growth running time for operations.

force: height(left child) and height(right child) to differ by at most 1

how? perform BST operations then rebalance trees through rotations.



only 2 rotations needed to rebalance tree!

Exercises

1] write a method to perform left (or right) rotation on a node x.
- you may assume x is a python object w/ fields x.left and x.right
- return the new root to the subtree previously rooted at x.

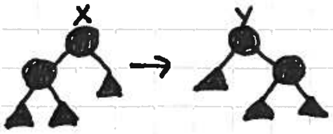
(2) write a recursive function to determine the height (as defined in class) of a given node.

(3) if I replaced ~~the~~ the subtree rooted at x with another arbitrary AVL subtree, which nodes would potentially break the AVL invarient?

(4) when the AVL invarient is broken, why do we correct it starting at the deepest node that violates the invarient rather than working from the root down?

Excersize 1:

right rot.



```
def right_rot(x)
    y = x.left
    A = y.left
    B = y.right
    C = x.left
    y.right = x
    x.left = B

    return y
```

OR

```
y = x.left
x.left = y.right
y.right = x
```

- error checking omitted !! ☺
- similar for left.

Exercise 2:

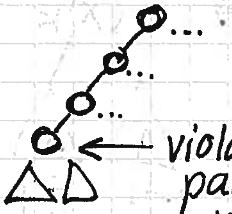
```
def height(x):

    if leaf(x): return 0
    return max(height(x.right), height(x.left)) + 1
```

ex3:

all parents of x may break the invarient

ex4: it takes fewer rotations to fix the tree from the bottom up. we will need to do at most 2. because once the location of the change is made AVL compliant, its parents will be too.



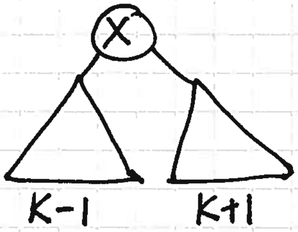
if we started "fixing" here the lower would still be violated.

violation causes parents to be violated too → so fixing it fixes parents.

---

review from lecture: violation fixing.

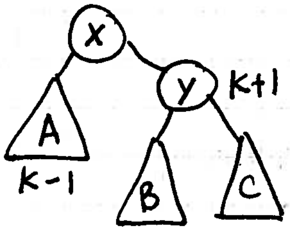
x violates AVL invarient then



- why only a difference of 2? not 3 or 4?

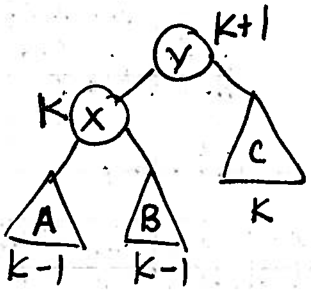
because we assume the violation occured b/c of another operation like insert or delete.
uhoh!
use mutable nodes.

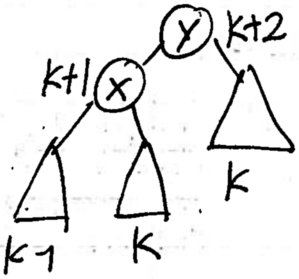
So then we break down the tree more:

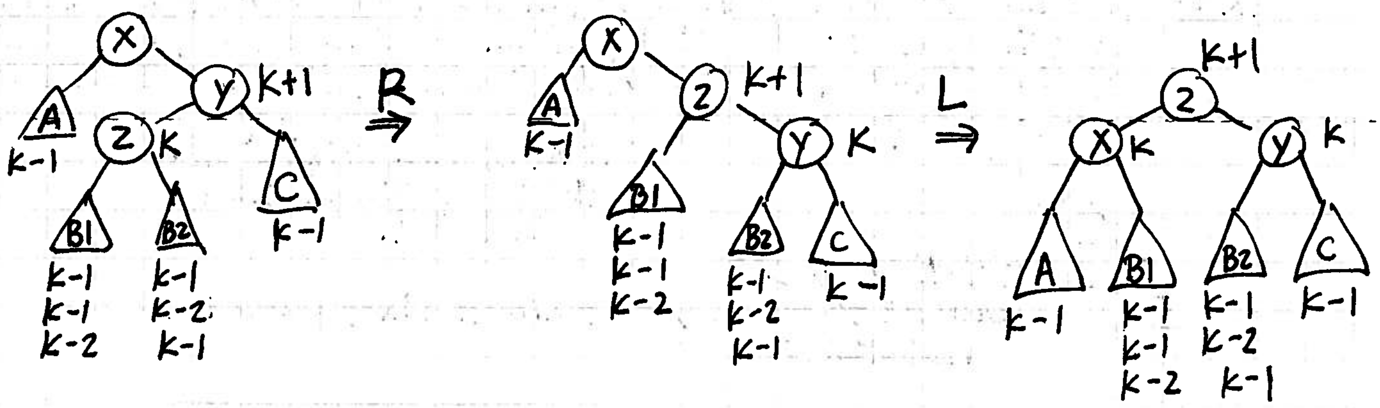


← one must be k to make (y) have height k+1

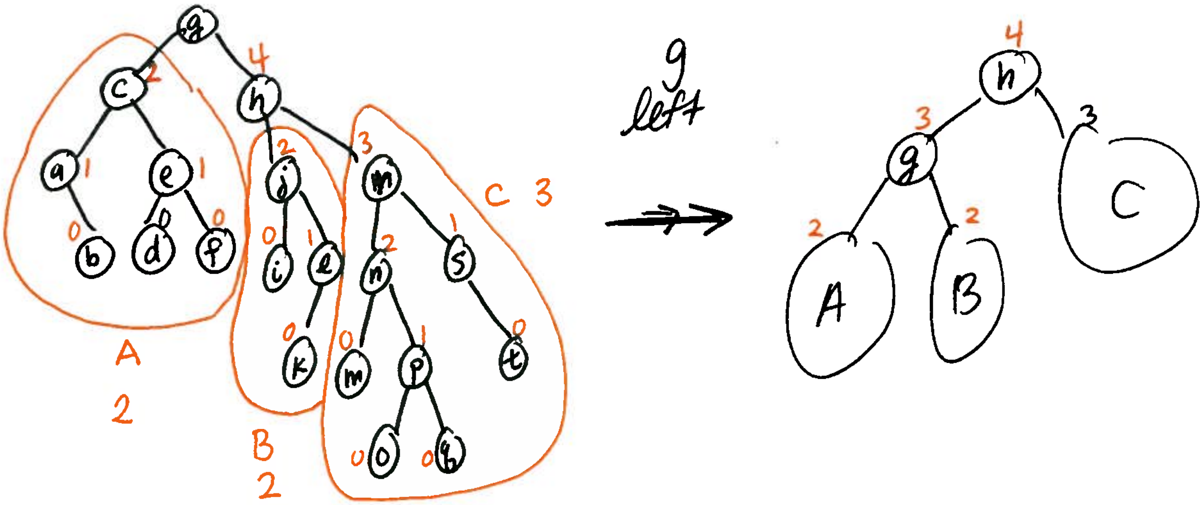
case 1:	k-1	k	⇒ left rotate
case 2:	k	k	⇒ left rotate
case 3:	k	k-1	⇒ break down furthar

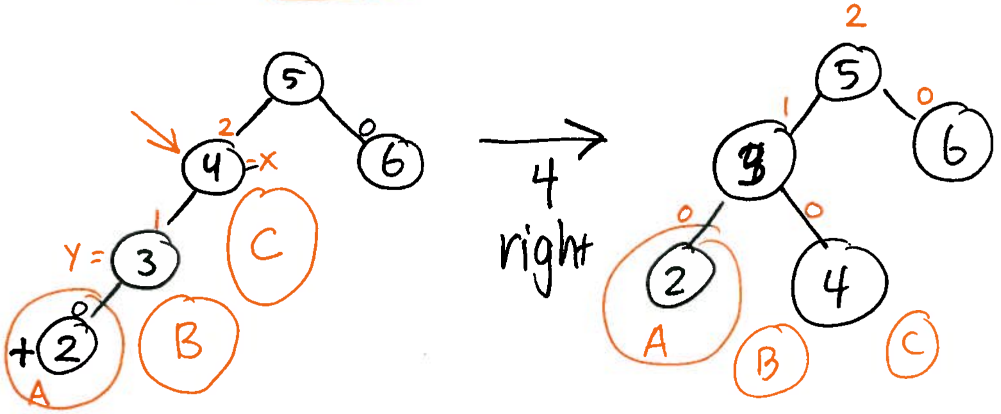


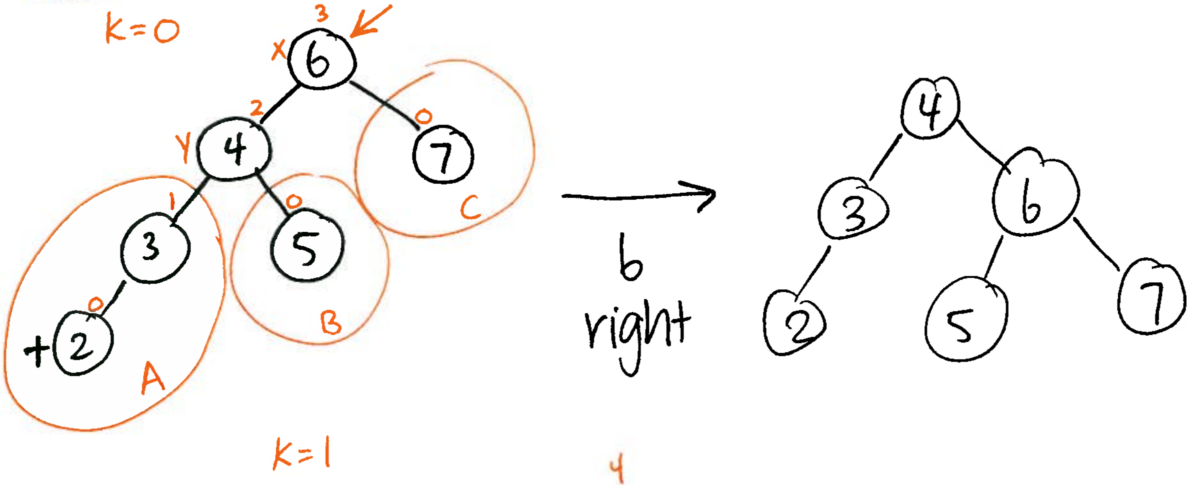


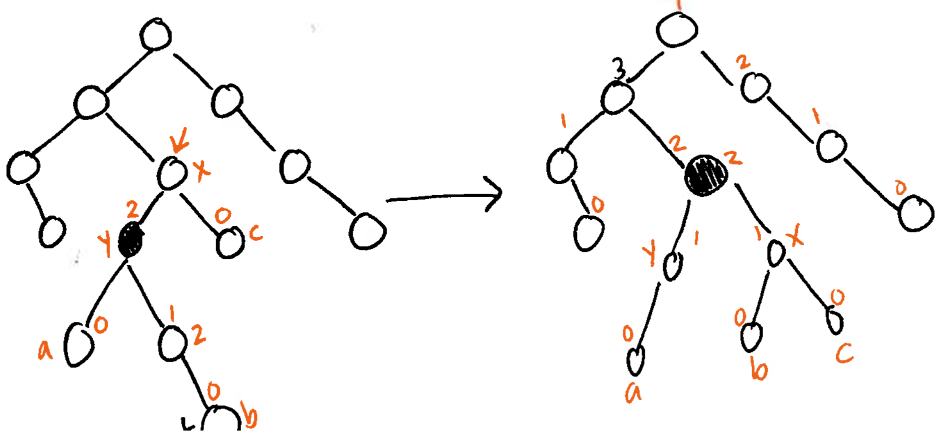


Answers









oops. 2 steps.
write out each.