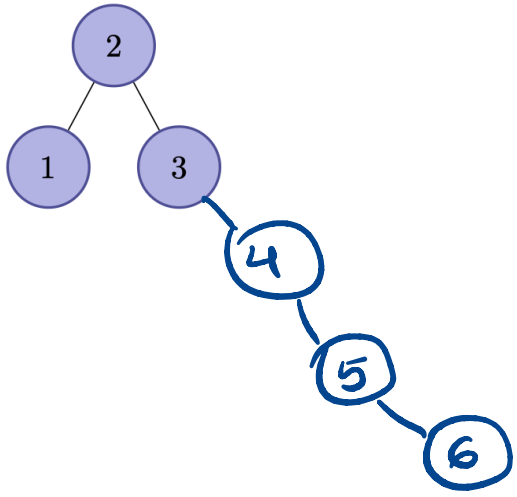
AVL Trees

CSC263 Week 4

Insert 4, 5, and 6 into the following tree

Regular BST



Shape is determined by insertion order & relative values

the height of a tree rooted at v is the number of edges on the longest path from v to a leaf

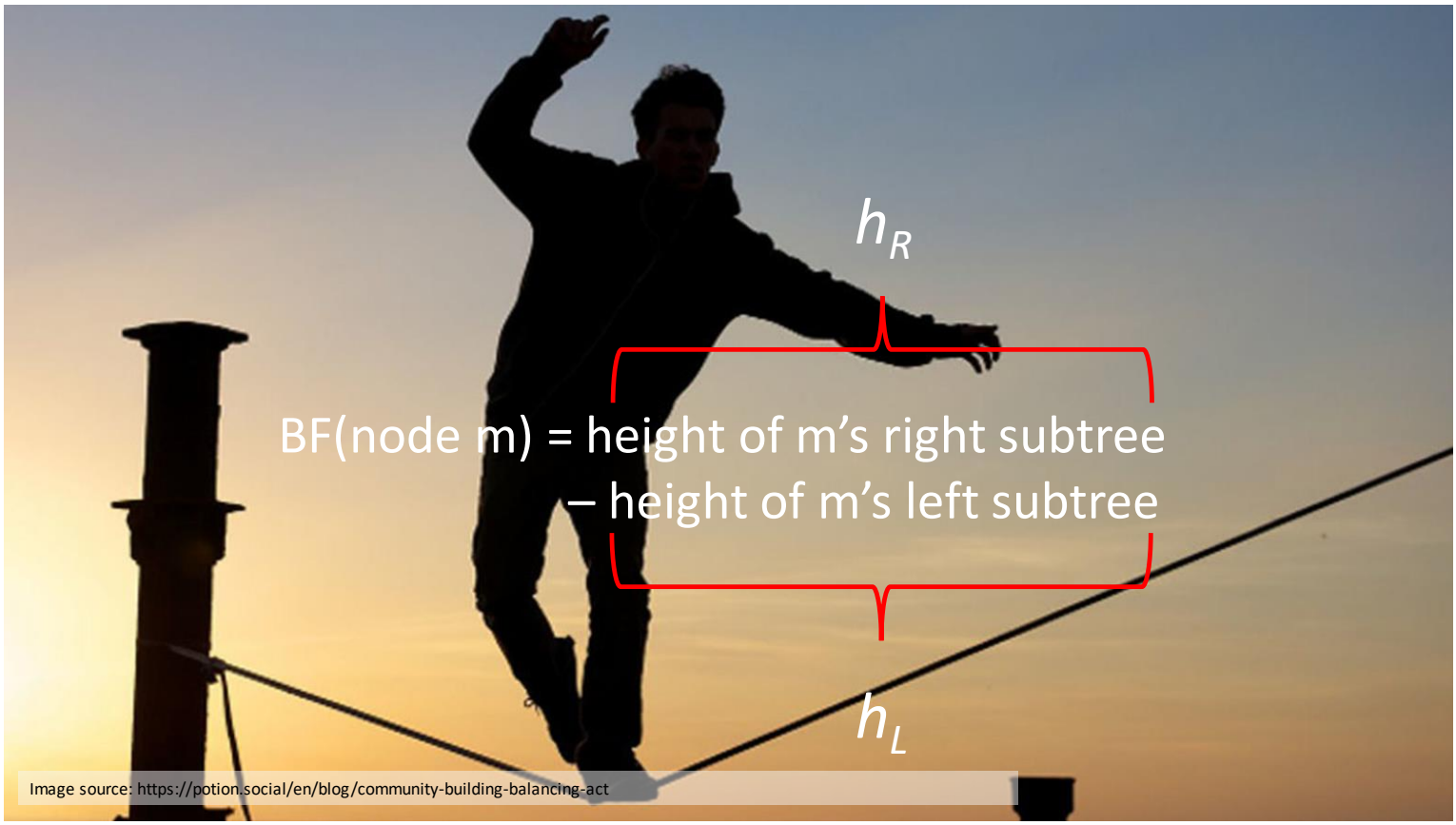
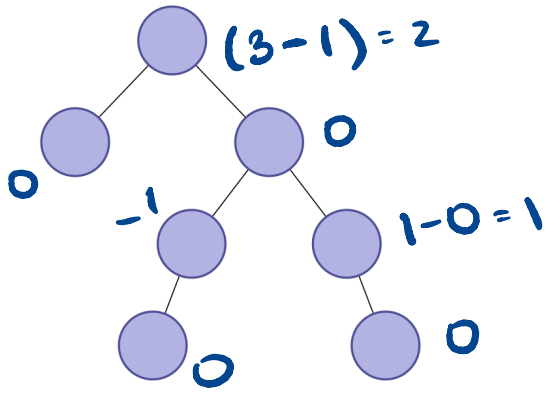


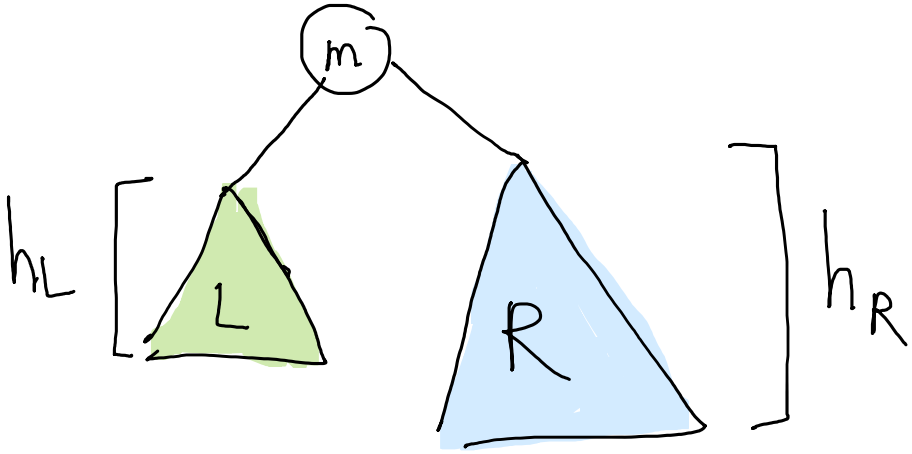
Image source: https://potion.social/en/blog/community-building-balancing-act

height of empty subtree = -1

Calculate BF for all nodes in the following tree:



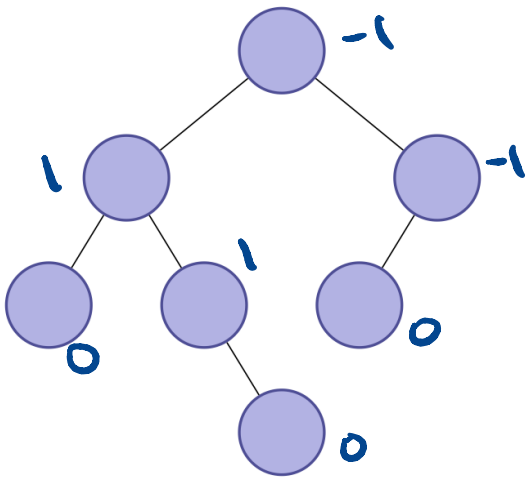
Height Balance Property

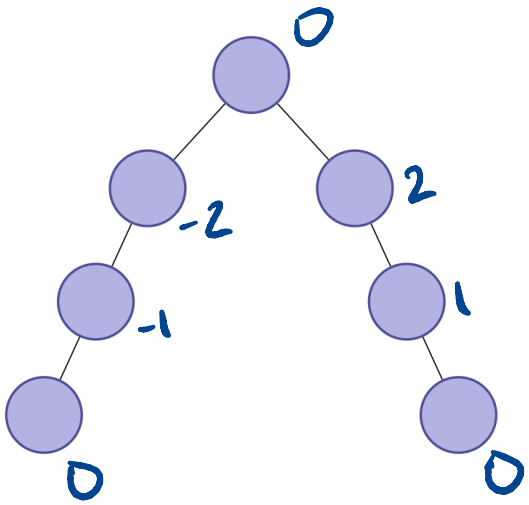


If $h_R - h_L = 0$ m is balanced
If $h_R - h_L = 1$ m is right-heavy (but AVL balanced)
If $h_R - h_L = -1$ m is left-heavy (still AVL balanced)
Any other values, the tree is not AVL balanced

Poll

Which tree AVL balanced?





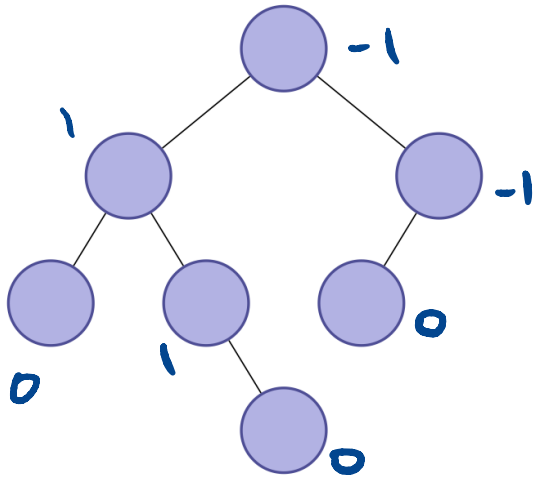
A: The tree on the left B: The tree on the right C: Both D: Neither

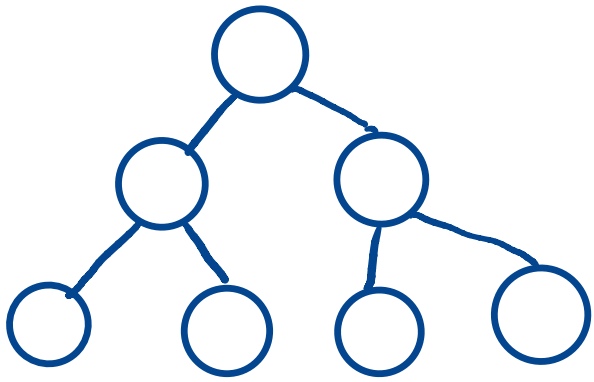
can only contain (0, -1, 1) to be balanced

For a tree to be AVL balanced, every node should be AVL balanced

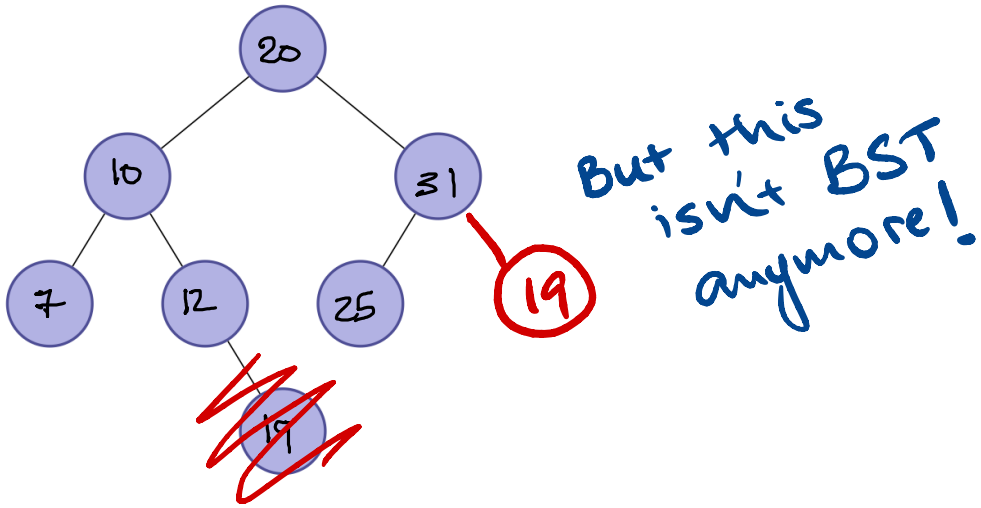
What is an ideally balanced version of this tree?

nearly complete tree

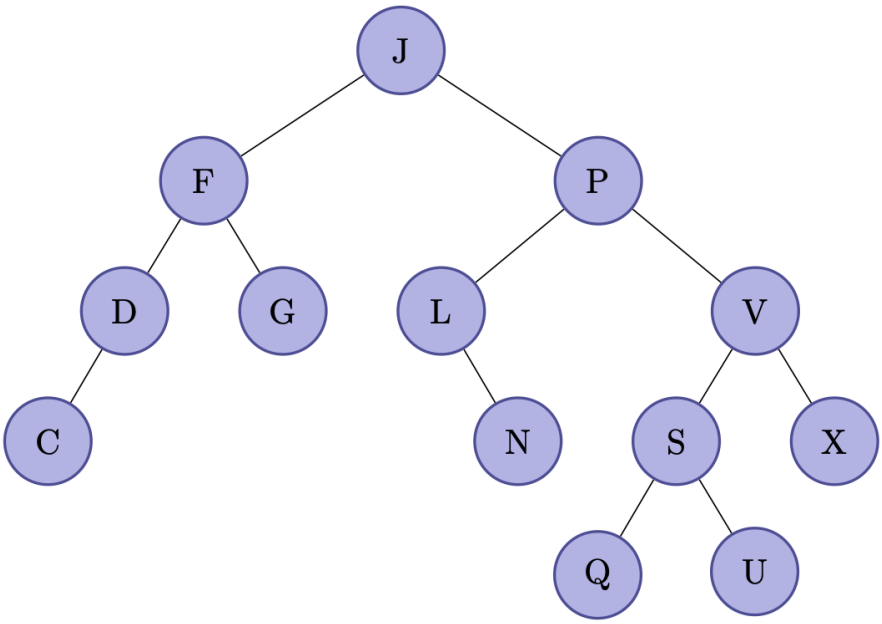




What is an ideally balanced version of this tree?



AVL Trees



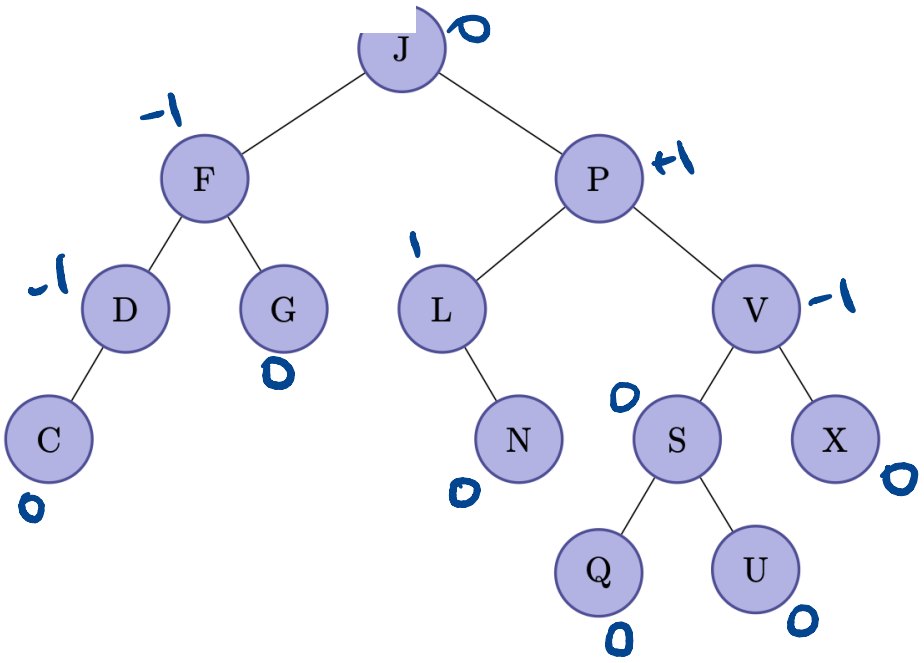
AVL Supplementary Material: Quercus Lectures Page

IMP

Search algorithm is same as BST search
O(height)

$$h \leq 1.44 \log_2(n+2)$$

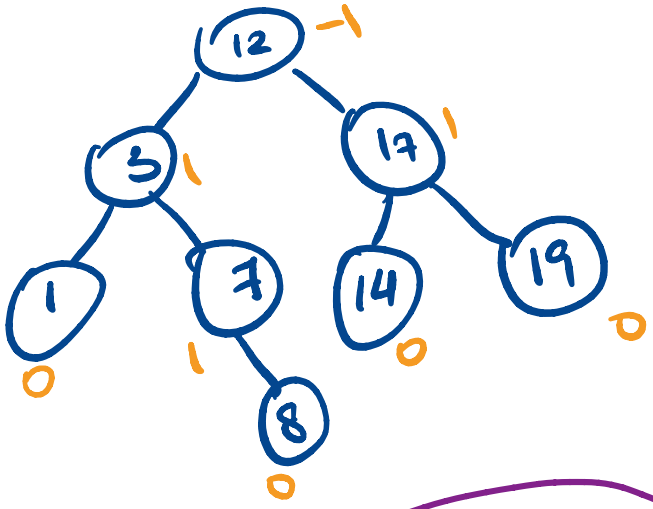
rees



Worksheet

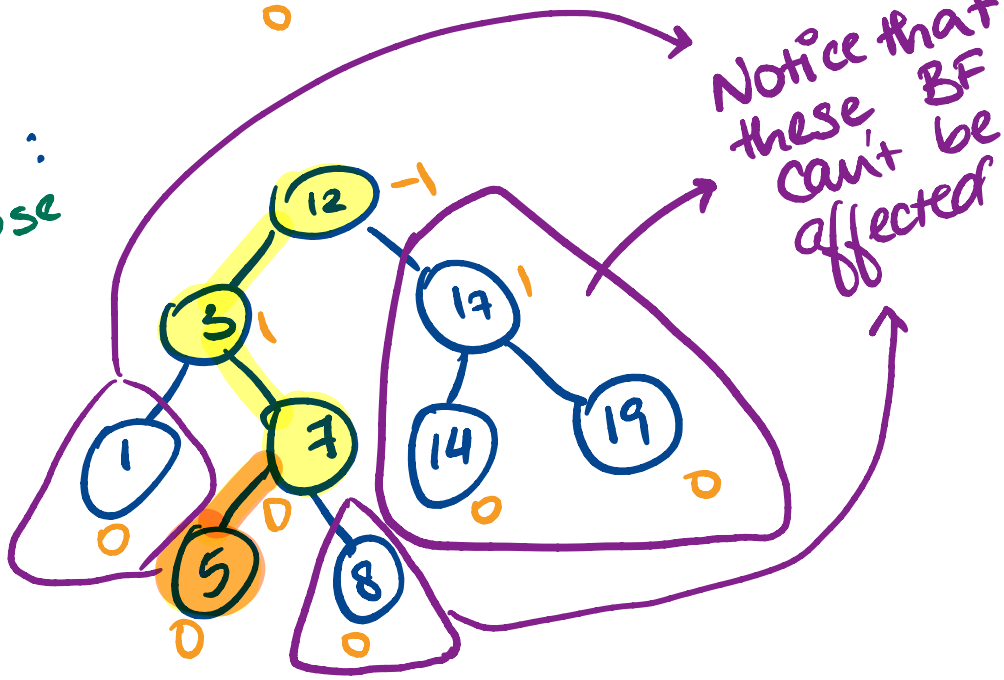
Do questions 1-3 from the worksheet

Q2 :

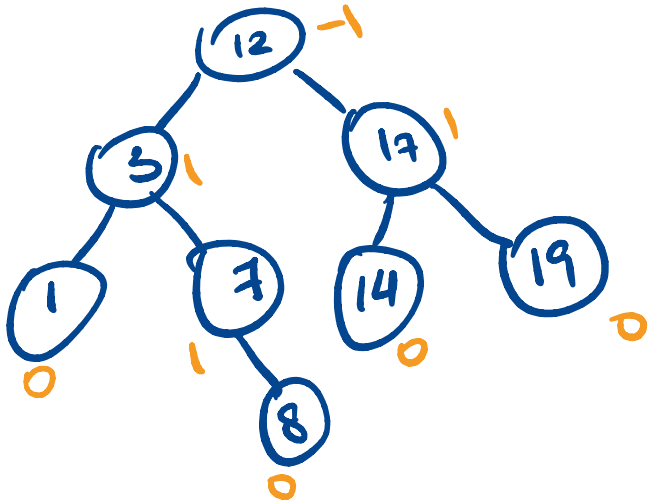


Inserting 5 :

Only ones whose BF can be changed is highlighted in yellow (its ancestors)

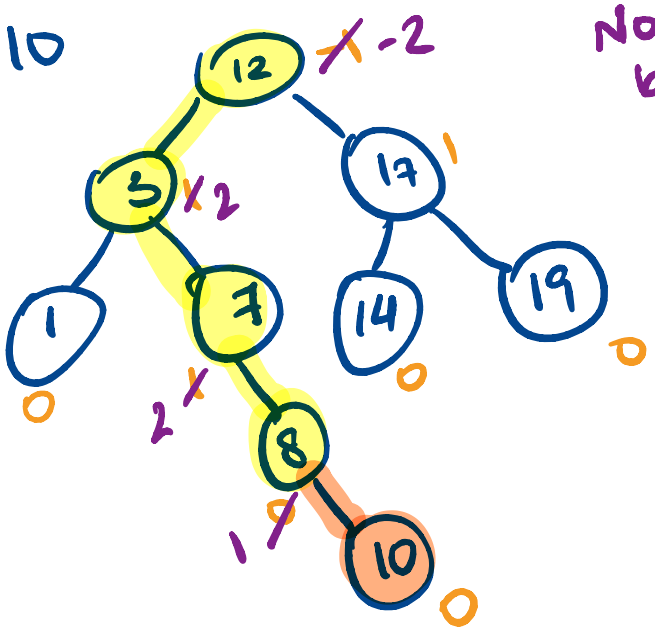


Before inserting



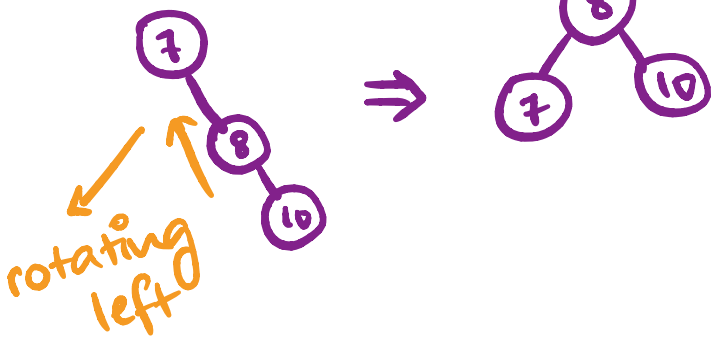
Worksheet Solutions

Q3: Insert 10



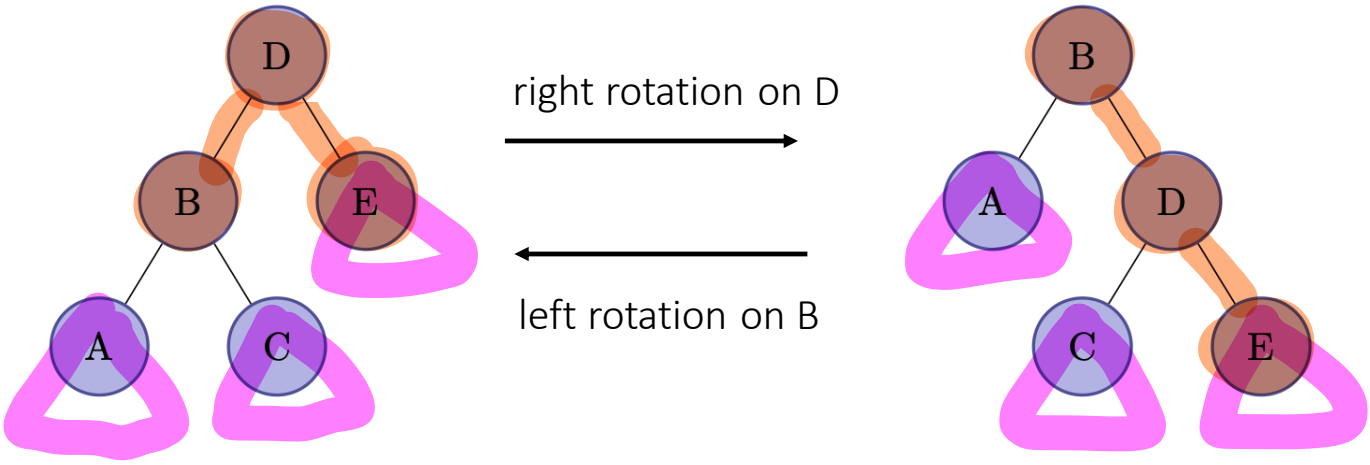
Not AVL balanced anymore

To balance:



Rotate first BDE and then add child A, C
A and C could be subtrees instead of nodes

Single Rotations



h_L = 1 + max(h(C),h(A))

h_R = h(E)

BF = h(E) - 1 - max(h(C),h(A))

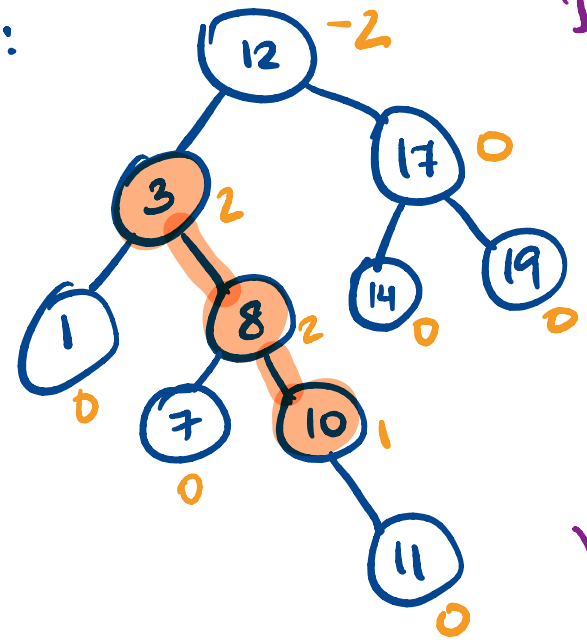
h_L = h(A)

h_R = 1 + max(h(C),h(E))

BF = 1 + max(h(C),h(E)) - h(A)

Worksheet Solutions

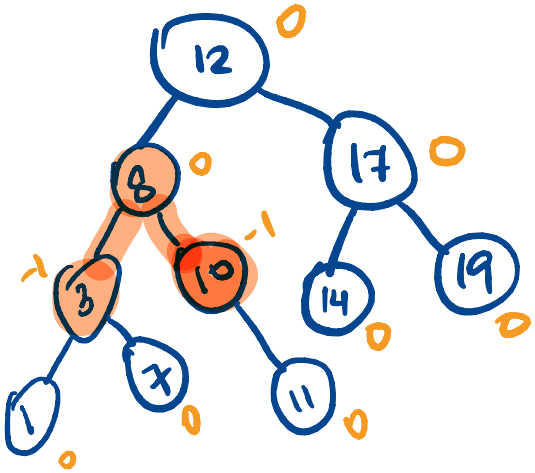
Q4:



It is NOT balanced

Rotate lowest node that is not AVL balanced. If that node is a 2, then its right-heavy & we need to rotate left. If its -2, its

left rotation cuz the original is right heavy



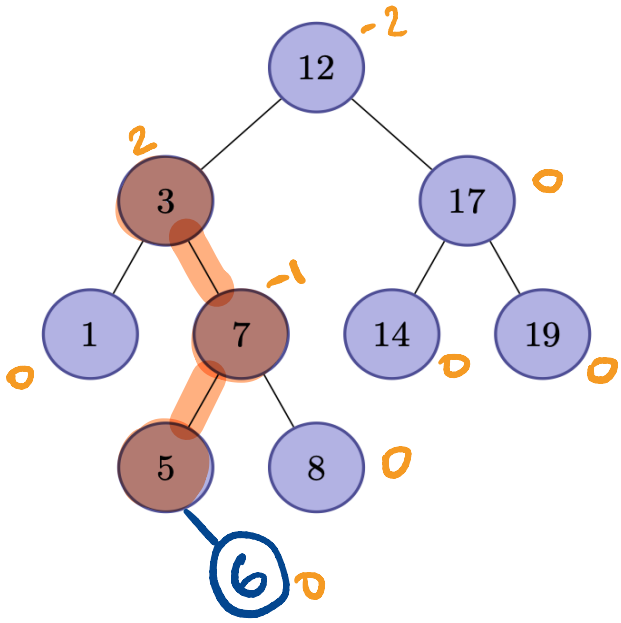
notice that left subtree of 12 at the end is height 2 and the original left subtree before we add the insert was height 2

Single Rotation Properties

1. Insertion only affects the balance factors of its ancestors (justification in the supplementary pdf on Quercus)
2. The root BF depends on h(A), h(C), h(E)
3. The overall height of the rotated tree remains the same (as before insertion), so that nothing beyond it is affected in terms of height or balance factors

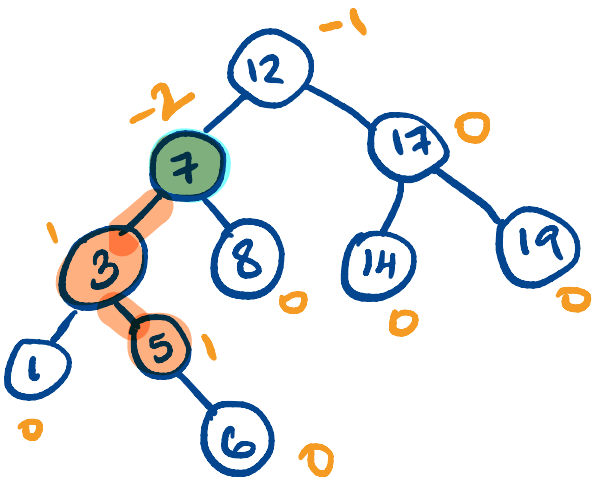
Worksheet

AVL Worksheet Question 6

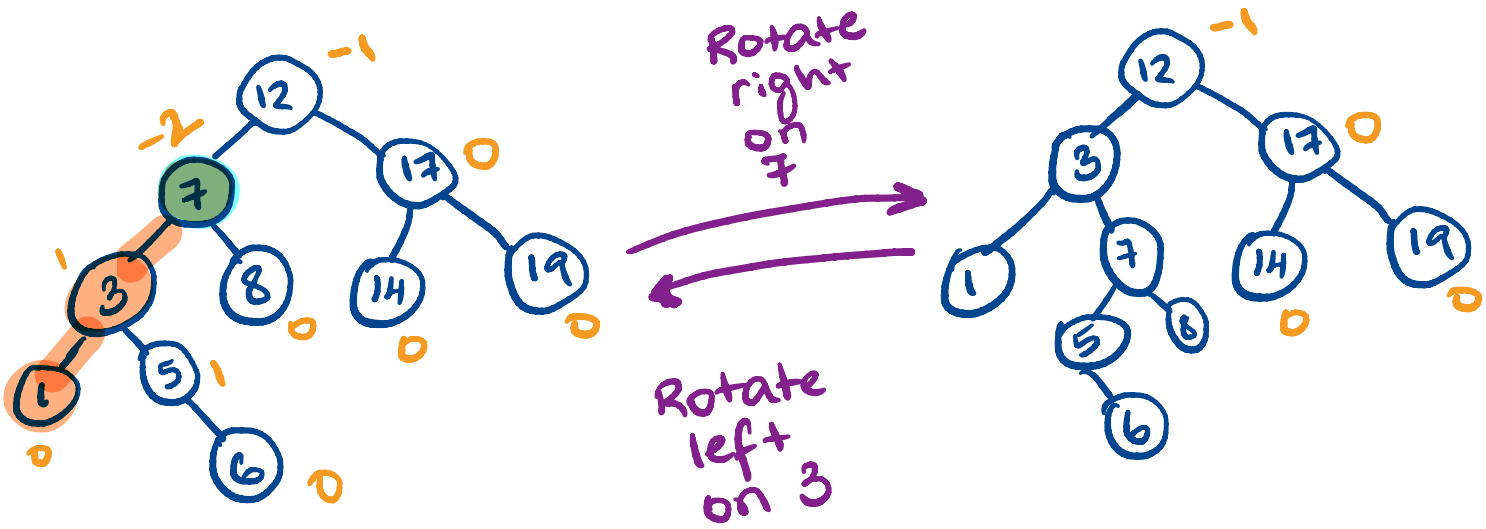


Task: Insert(6)

Balanced => Rotate left on 3



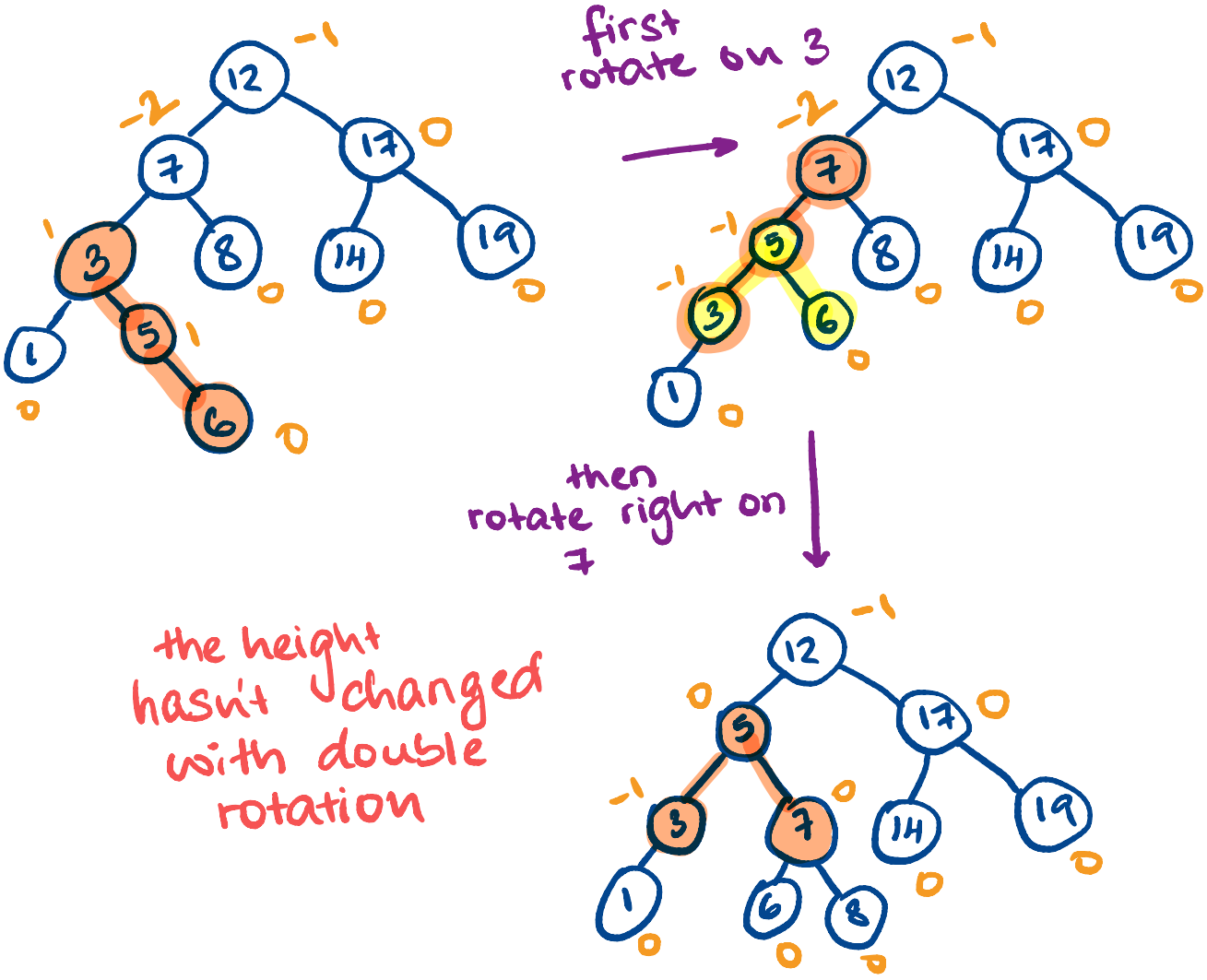
still not balanced! Rotate again



Worksheet Solutions

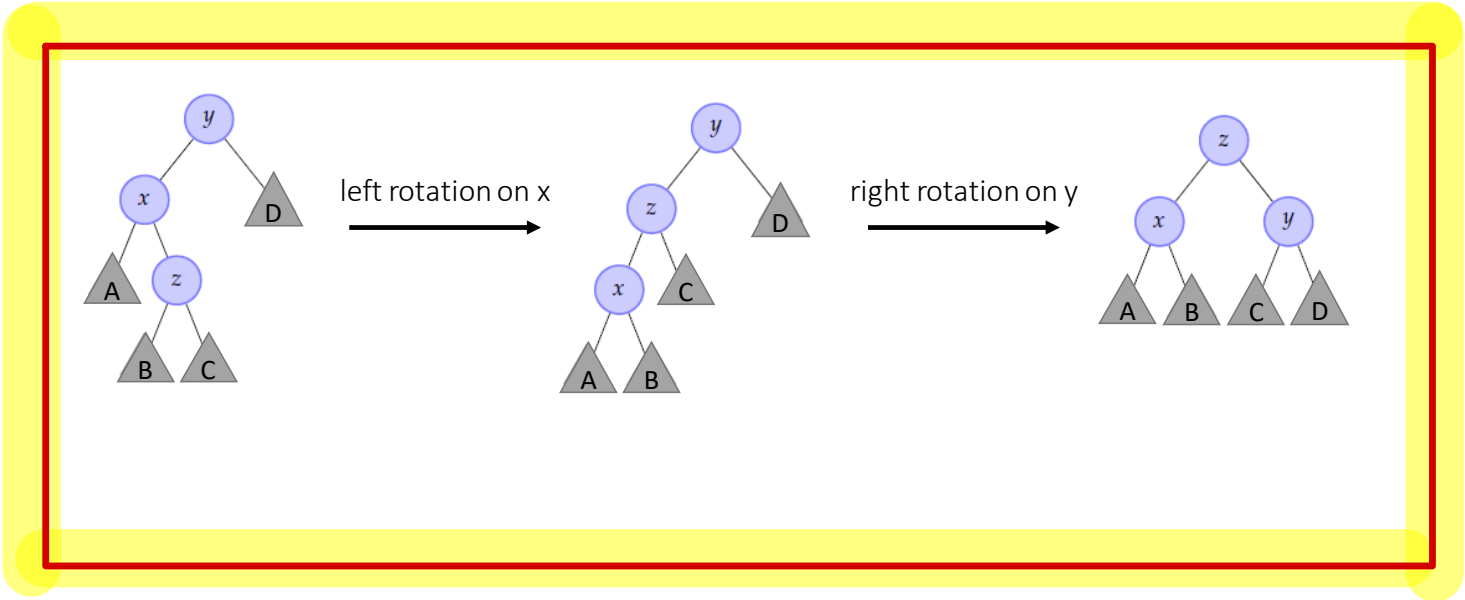
AVL Worksheet Question 6

To solve this issue, DOUBLE ROTATE

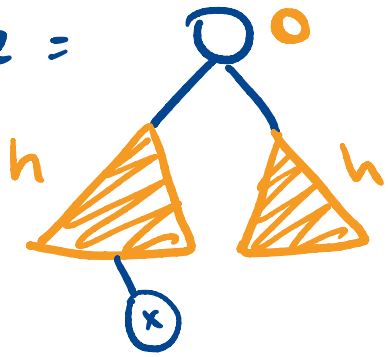


the height hasn't changed with double rotation

Double Rotations: Left Right Rotation



If tree =

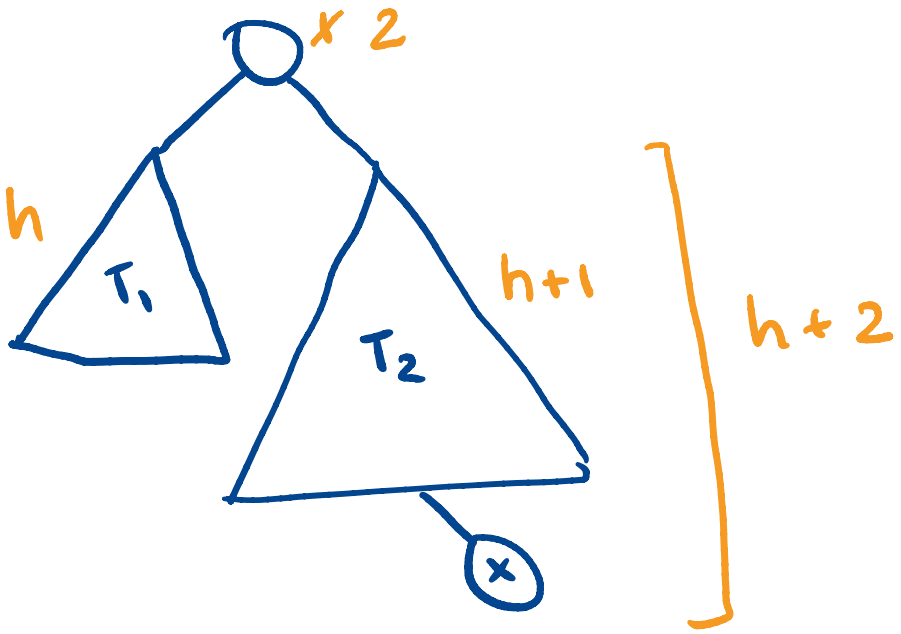


Inserting to a tree with root BF = 0, needs no rotation. Root BF will just change to 1 or -1.

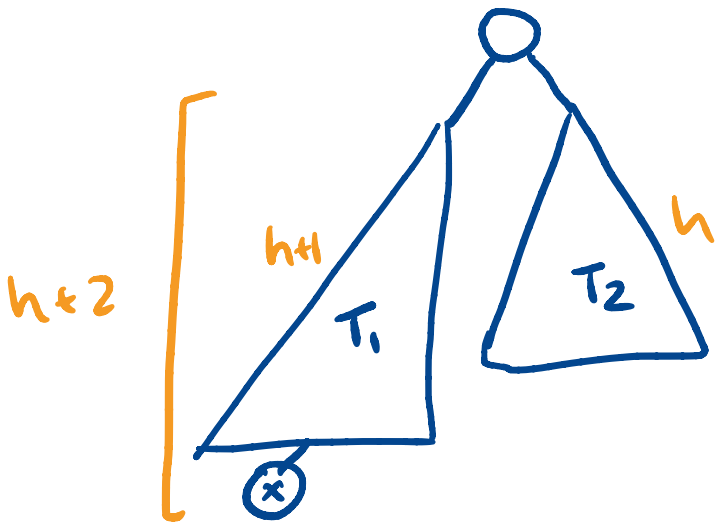
Chalk Talk

When do we encounter trouble after inserting into an AVL?

Case 1: When AVL tree is right-heavy before insert



Case 2: when AVL tree is left-heavy before insert

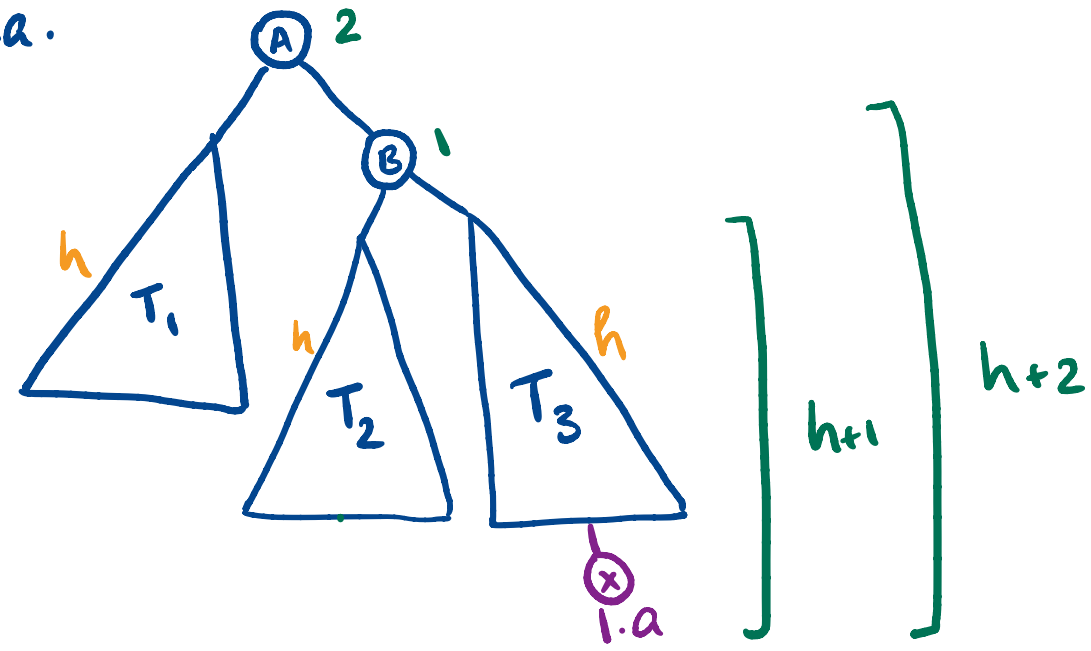


Chalk Talk

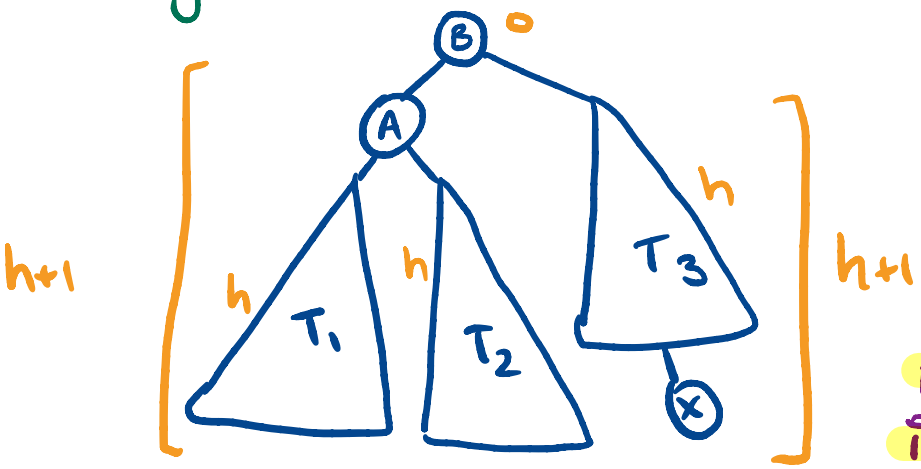
Case 1: Tree is right heavy + inserted node goes to the right subtree.

Assumption : A is the first ancestor that becomes unbalanced after inserting x

Case 1a.



Need single left rotation on A



BST order ✓

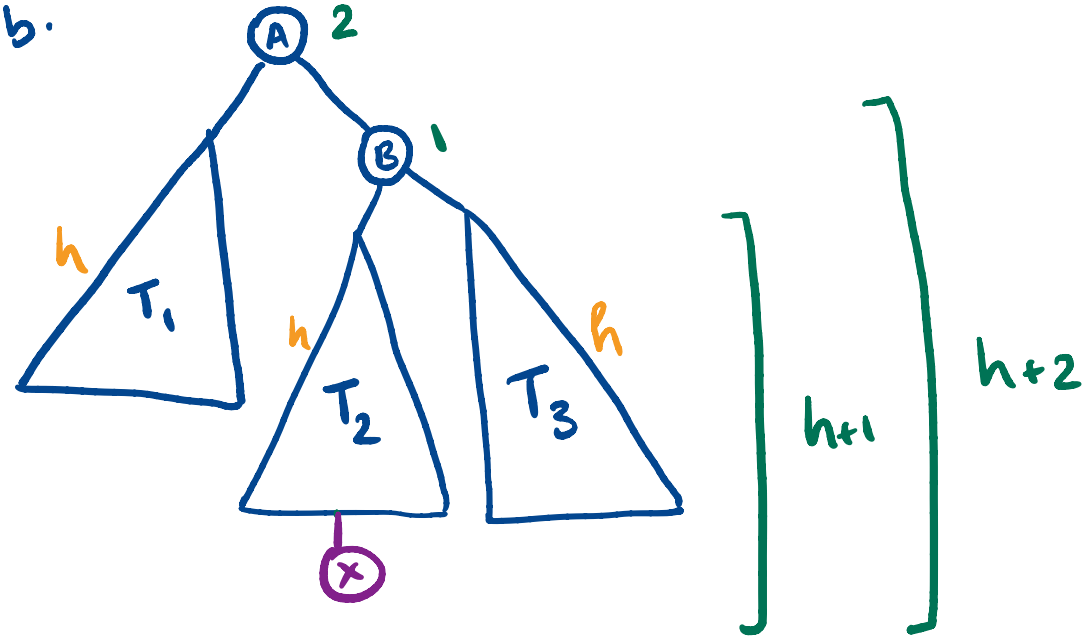
AVL balance ✓

complexity ?

Constant time

height before insersion = height after insersion

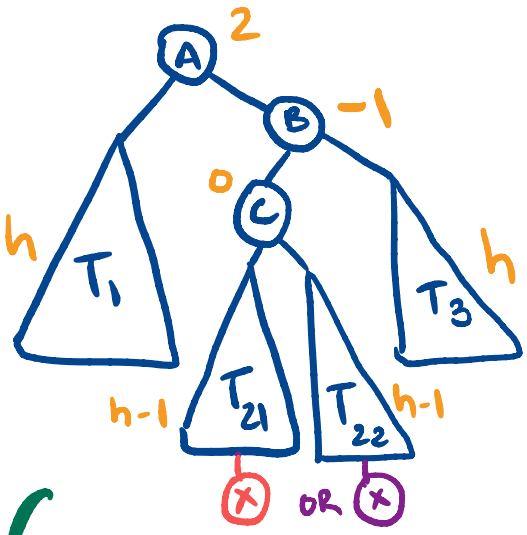
Case 1b.



Chalk Talk

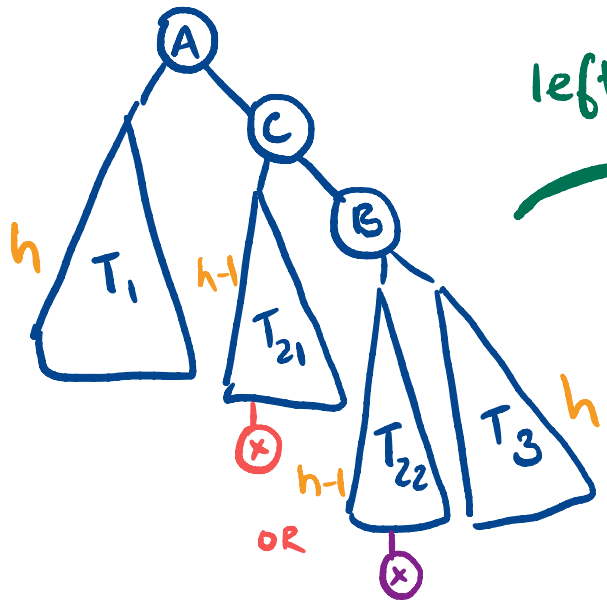
Case 1.b

Redrawing the tree

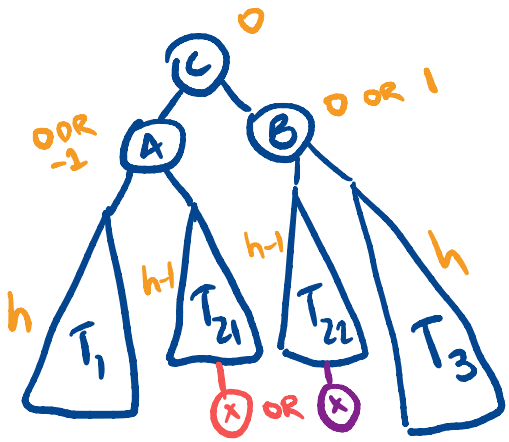


Need double right-left rotation on A

right on B

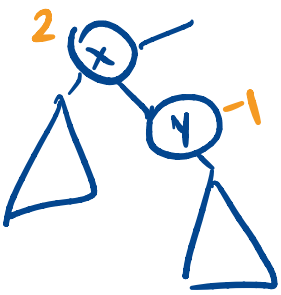


left on A

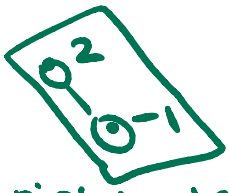


BST ✓
AVL balance ✓
height after = height before
complexity ? constant

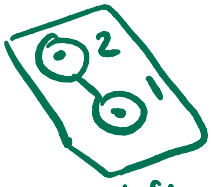
When do we do double rotation



we see 2, we know we need to left rotate. Look at its right child, if it has a BF of opposite sign, like -1, we need to right rotate first.
If it was +ve, then single rotation

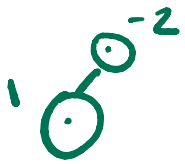


right - left rotation



left rotation

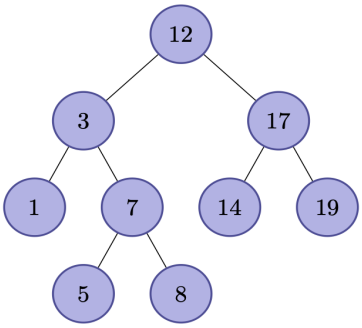
right rotation



left - right rotation

Chalk Talk

Insertion Algorithm

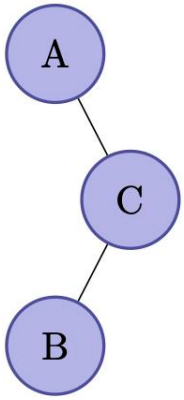



```
insert as normal BST

walk back from new node towards root: #assume i is the
parent of the new node

if new node is inserted right, add 1 to BF[i]
if new node is inserted left, subtract one from BF[i]

if BF[i] becomes zero, then done
if BF[i] becomes 1 or -1: don't rotate but continue up
the path and repeat
if BF[i] becomes to 2 or -2: do the rotation (R, L,
LR, RL) and then done

How many rotates in worst case?
```

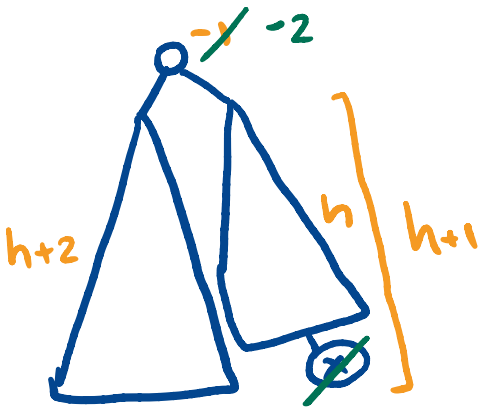
$O(\log n)$

Zoom Activity

The tree below is a special case because T1, T2, and T3 are empty. Can you still do a double rotation to fix the BFs?



All cases end up being deleting a leaf

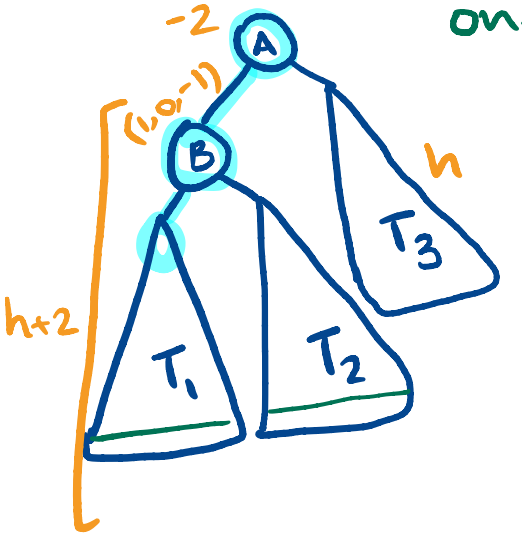


when we delete x, AVL tree not balance

Both these cases are symmetric

Chalk Talk

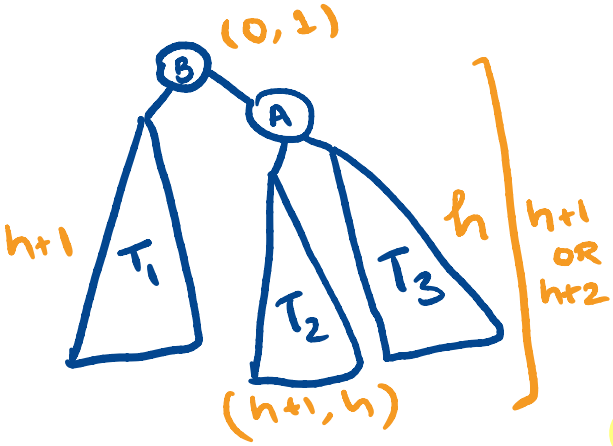
Deletion Cases



one of T_1 or T_2 is $h+1$

T_1	T_2	BF(B)	
h	$h+1$	1	left rotation
$h+1$	$h+1$	0	right-rotation
$h+1$	h	-1	

Rotate right on A

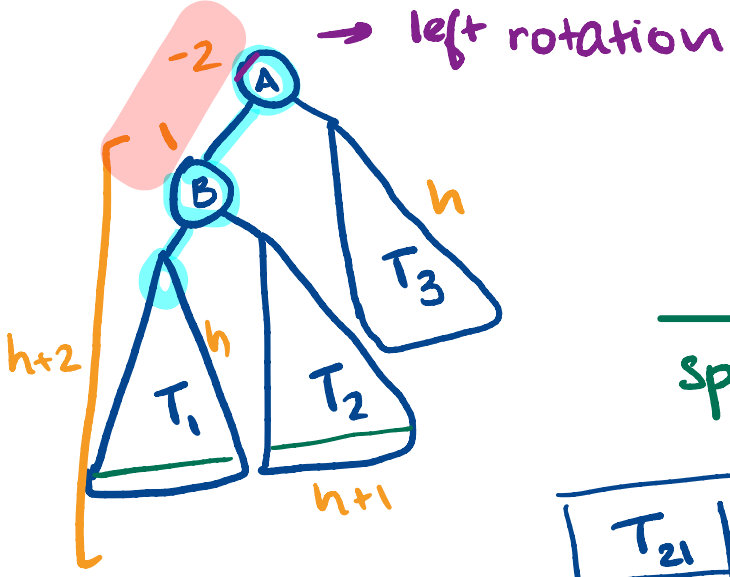


BST ✓
complexity? single rotation - O(1) ✓

height after may be shorter

AVL balance ✓

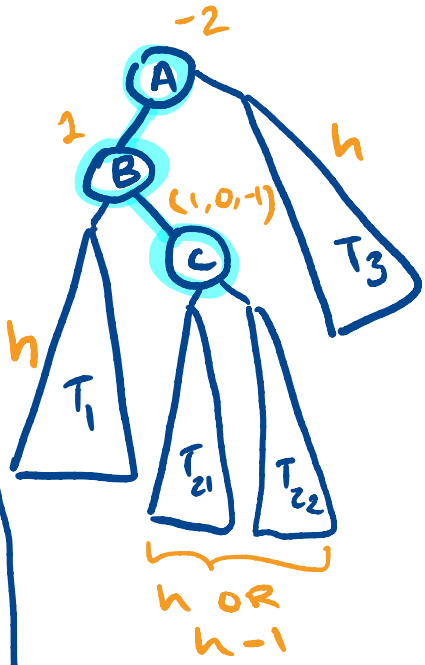
next left rotation on B



→ left rotation

split up T_2

T_{21}	T_{22}	BF(C)
h	h	1
h-1	h	0
h	h-1	-1



Chalk Talk

Deletion with double rotation

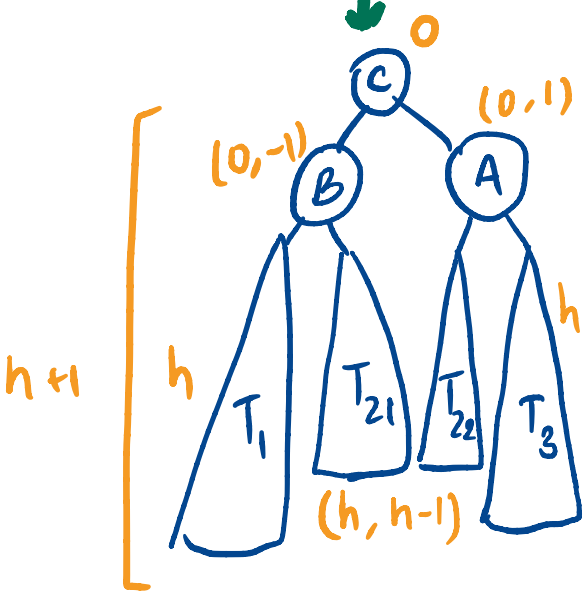
BST ✓

AVL ✓

constant time ✓

height decreased

left rotate on A



Chalk Talk

Deletion Algorithm

```
find node p to delete

on route from p back to root at each node i:

if BF(i) was previously 0:
                update BF and exit
If BF(i) was previously +1 or -1:
        if it becomes 0 after deletion, then we shortened tree rooted at i so
        continue up path
If BF(i) was previously +1 or -1 and change makes it +2 or -2:
        do appropriate rotation
        if rotation shortened tree rooted at i, then continue up tree
        if not, stop
```

How many rotates in worst case? $O(\log n)$ rotations

Image credit: https://www.earlytorise.com/the-big-idea/

Grand Ideas

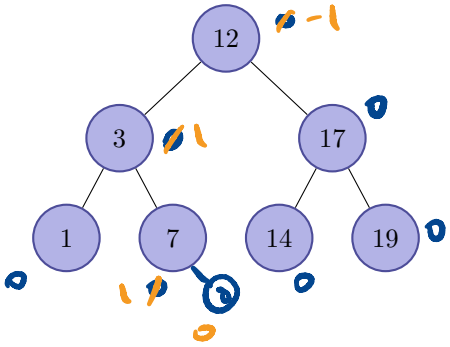
Deliberate Compromise

Perfect balance is too expensive to maintain (e.g. *O(n)*) and does not buy us anything asymptotically. Therefore, we settle for something good enough (e.g. BF -1 and 1), so the AVL tree is almost balanced but still *O(logn)*

Selective Augmentation

Storing extra information on existing data structures extends possible operations with no additional run time. For example, BFs in AVL trees are just an augmentation of a BST!

1. Into the following binary search T1 insert 8. Indicate all the balance factors on the resulting tree T2. Is the tree still AVL balanced?



2. Redraw T2 (the resulting tree from Q1) here and insert 5. Is the new resulting tree, T3, still AVL balanced? If not, how can you restore the balance? Yes

3. Let's go back to T2 (the result from Question 1) and draw it here. Now insert 10 into it. Is the resulting tree, T4, still AVL balanced? Can you see a way to rebalance this tree? In lecture, we will rebalance the tree and call it T5. No, balance shown

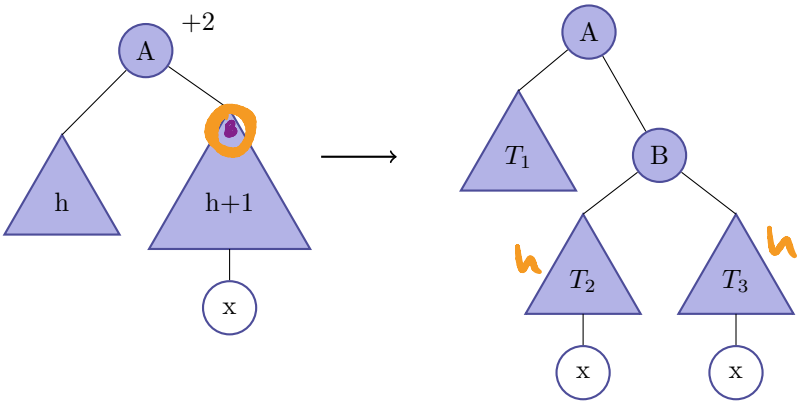
No, shown

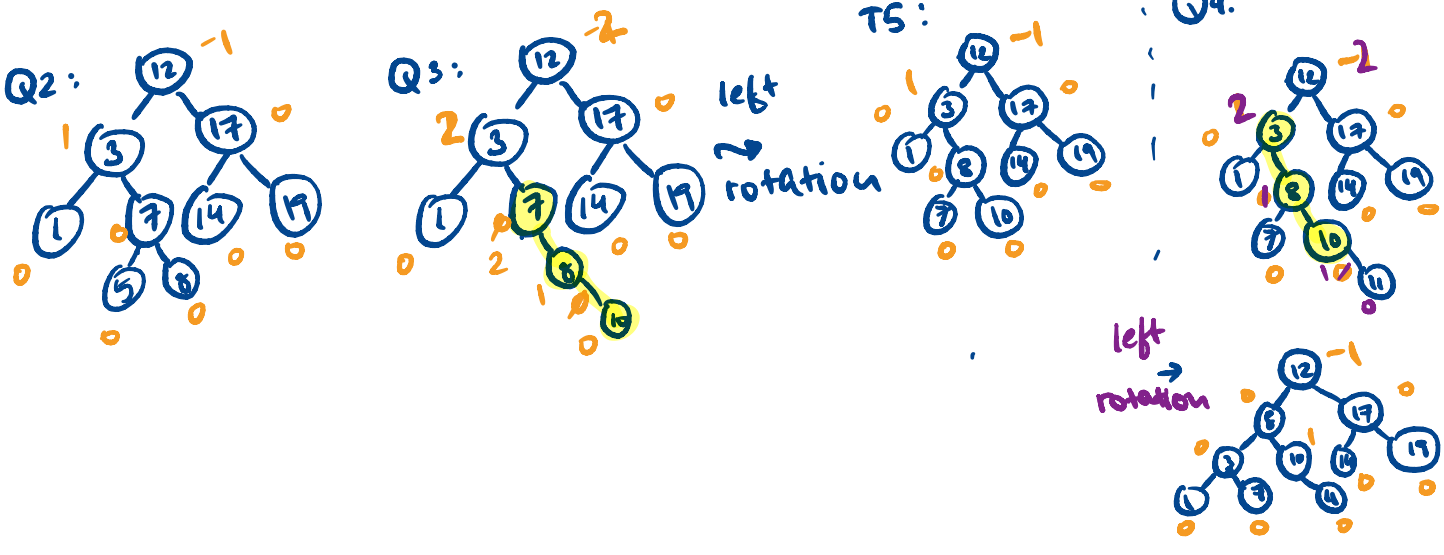
4. Now Insert 11 into T5. Is it still AVL balanced? If not, restore the balance using a rotation we learned in lecture.

5. (Practice Homework Question) Take the original T1 and insert 13, 15 and 12.5. With each insert, check the balance and do a rotation if necessary to fix the balance. Shown.

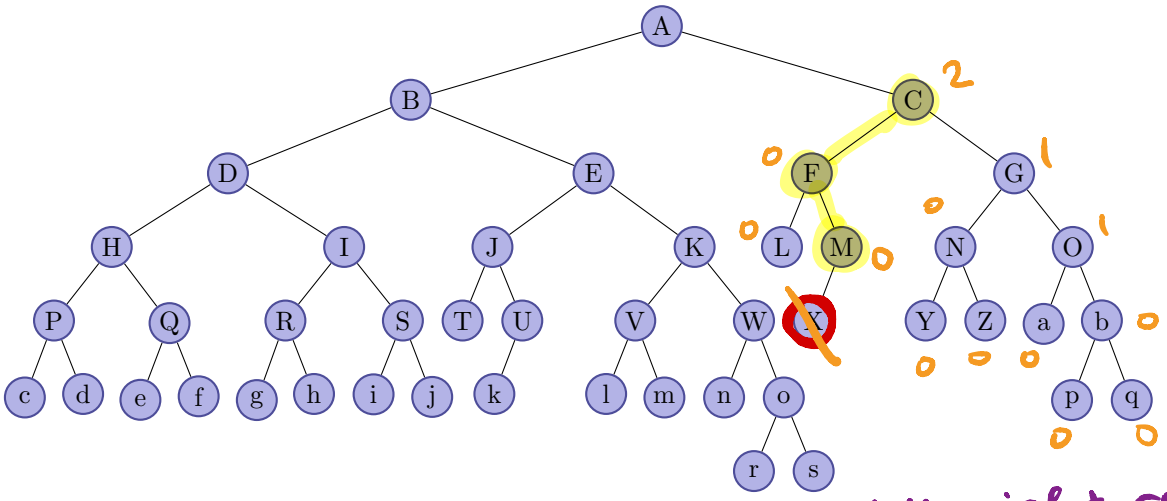
6. Go back to the original tree T1 from Question 1, Now insert 8, 5, then 6. Is it still AVL balanced? If not, can you restore the balance using a rotation that we learned in class? What is the problem? No. need double rotation

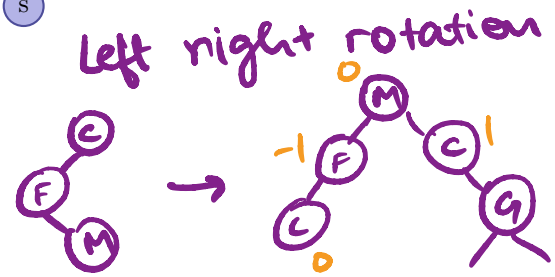
7. In the following tree, A becomes unbalanced (i.e. BF = 2) after x is inserted. Assume that A is the deepest ancestor on the path from x to the root that becomes unbalanced. In the expanded image of this tree, why can we assume that both of B'subtrees (T2, T3) have the equal height of h? Note that x could end up in either T2 or T3 (i.e. not both), but it's illustrated on both to cover the possible cases.



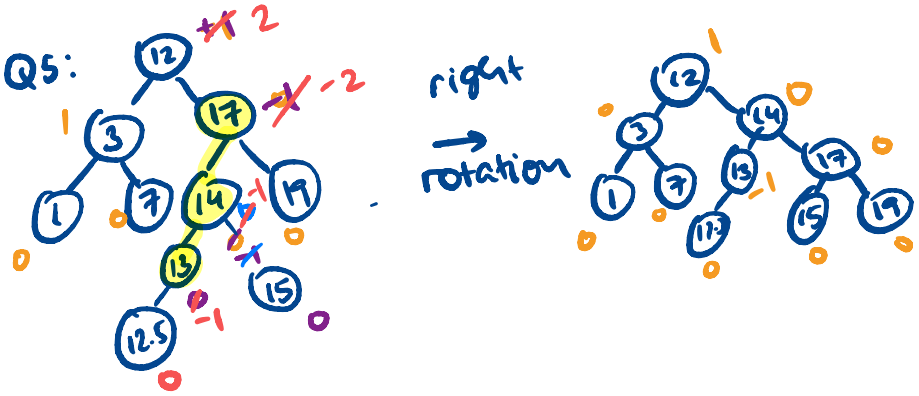


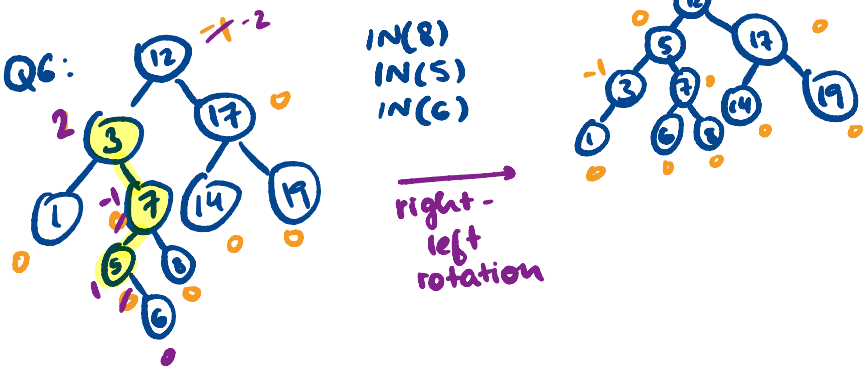
8. Delete X from the following AVL tree. Notice that the letters in the nodes are not key values but labels. (I.e. they are not in sorted order, nor do they need to be.)





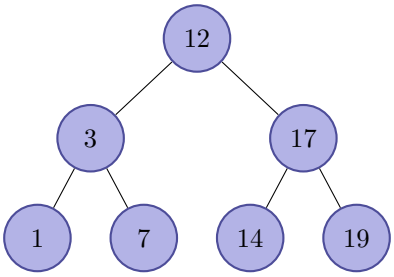
IN(13)
IN(15)
IN(12.5)





SOLUTIONS: Questions 1 through 4 were taken up during lecture. Other questions were left as optional homework exercises and so solutions are provided here.

1. Into the following binary search T1 insert 8. Indicate all the balance factors on the resulting tree T2. Is the tree still AVL balanced?



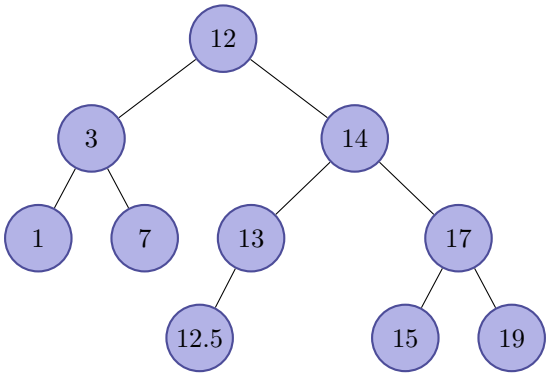
2. Redraw T2 (the resulting tree from Q1) here and insert 5. Is the new resulting tree, T3, still AVL balanced? If not, how can you restore the balance?

3. Let's go back to T2 (the result from Question 1) and draw it here. Now insert 10 into it. Is the resulting tree, T4, still AVL balanced? Can you see a way to rebalance this tree? In lecture, we will rebalance the tree and call it T5.

4. Now Insert 11 into T5. Is it still AVL balanced? If not, restore the balance using a rotation we learned in lecture.

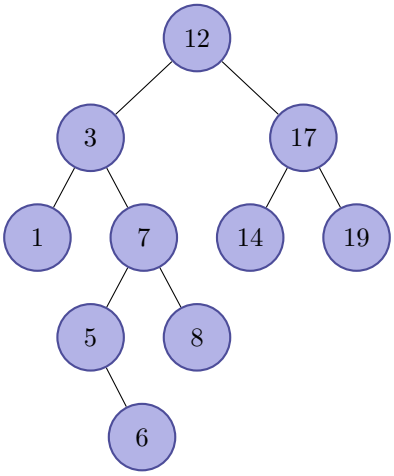
5. (Practice Homework Question) Take the original T1 and insert 13, 15 and 12.5. With each insert, check the balance and do a rotation if necessary to fix the balance.

   SOLUTION: After inserting 13, BF[14] becomes -1, BF[17] becomes -1, BF[12] becomes +1, and no rotations are necessary. After inserting 15, BF[14] becomes 0, BF[17] remains -1, BF[12] remains +1, and no rotations are necessary. After inserting 12.5, BF[13] becomes -1, BF[14] becomes -1 but BF[17] becomes -2 and BF[12] becomes +2 so a right rotation on 17 is necessary. The final tree, after all insertions and rotations becomes this:



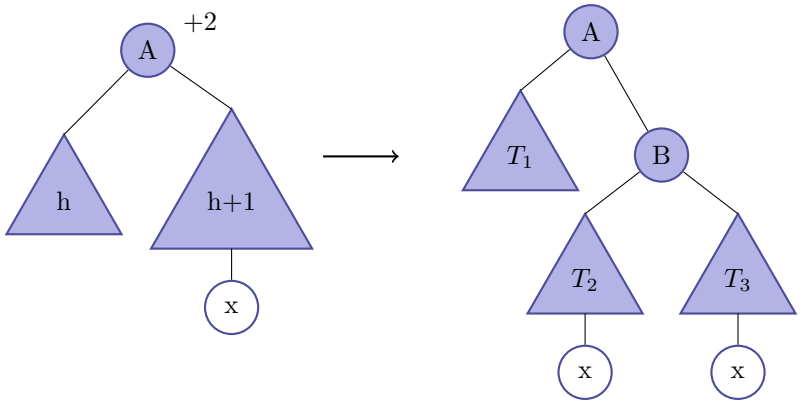
6. Go back to the original tree T1 from Question 1, Now insert 8, 5, then 6. Is it still AVL balanced? If not, can you restore the balance using a rotation that we learned in class? What is the problem?

   SOLUTION: The following tree is the result of inserting 8, 5, and 6 into T1:



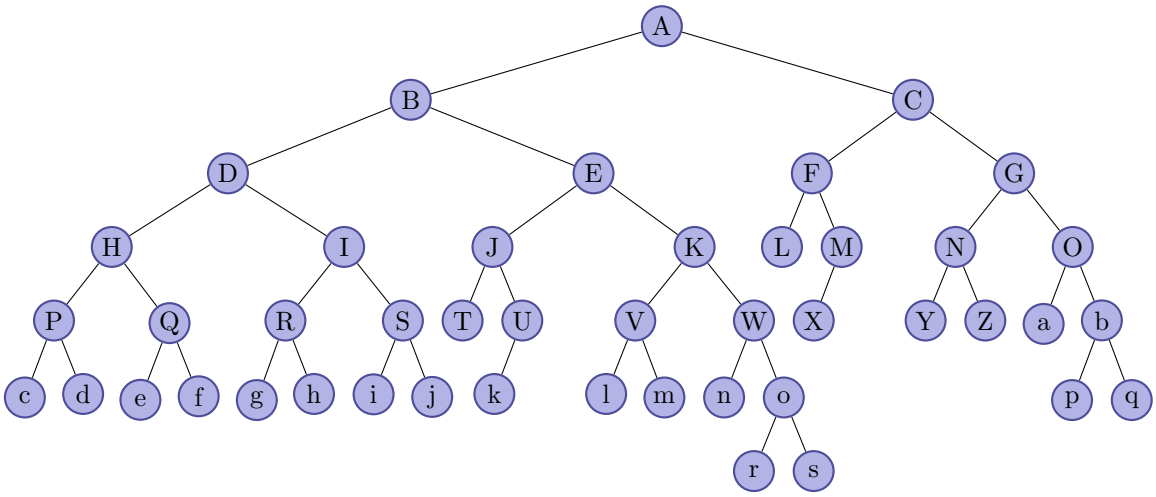
3 will be the deepest ancestor of 6 that becomes unbalanced with BF[3] = +2. Attempting a single left rotation on 3 won't work however because the subtree rooted at 7 is left heavy, and the 5-6 branch will just oscillate from 7 to 3. The resulting left rotated tree on 3 (now rooted at 7) will have BF[7] = -2. The difference in sign between BF[3] and BF[7] is the problem, because any single rotation will accentuate the unbalance in one direction, and therefore requires a double rotation to fix.

7. In the following tree, A becomes unbalanced (i.e. BF = 2) after x is inserted. Assume that A is the deepest ancestor on the path from x to the root that becomes unbalanced. In the expanded image of this tree, why can we assume that both of B'subtrees (T2, T3) have the equal height of h? Note that x could end up in either T2 or T3 (i.e. not both), but it's illustrated on both to cover the possible cases.



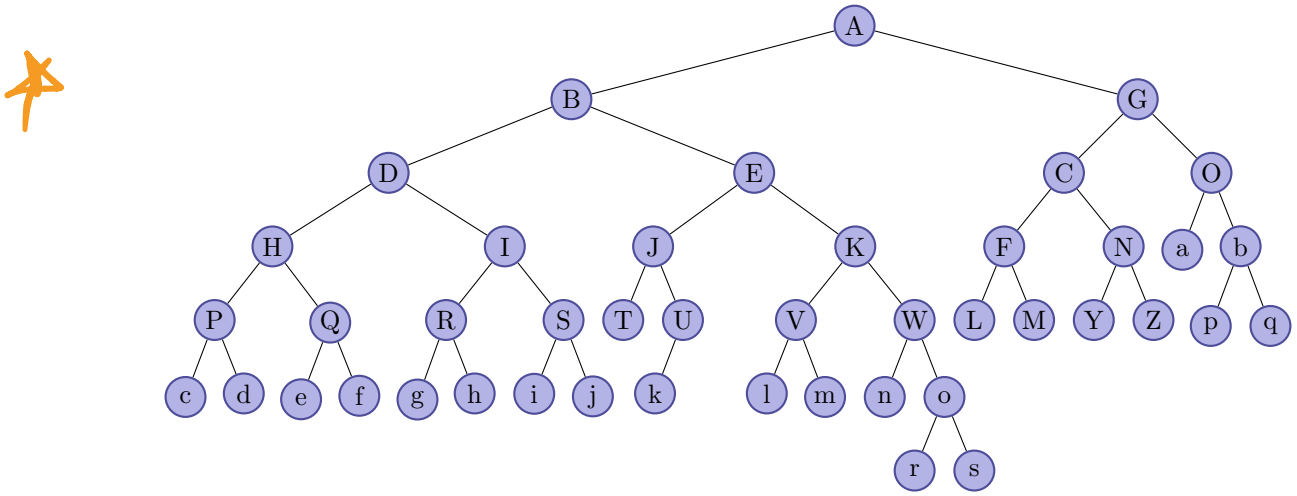
Solution: To prove that our assumption is correct, we will do a proof by contradiction showing that no other heights are possible for T2 and T3. First, assume that either T2 or T3 or both have height h + 1. That cannot be true or else A would have been unbalanced even before x is inserted. Second, assume that T2 has height h-1 and T3 has height h and x goes into T2. Then A won't become unbalanced which contradicts our assumption that A is unbalanced. Third, assume that T2 has height h-1 and T3 has height h and x goes into T3. Then B is the deepest ancestor of x that becomes unbalanced, which contradicts our assumption that A is the deepest ancestor of x to become unbalanced. The remaining two symmetrical cases (i.e. case 2 and 3 but for T3) also contradicts our initial assumptions, so they are not possible.

8. Delete X from the following AVL tree. Notice that the letters in the nodes are not key values but labels. (I.e. they are not in sorted order, nor do they need to be.)

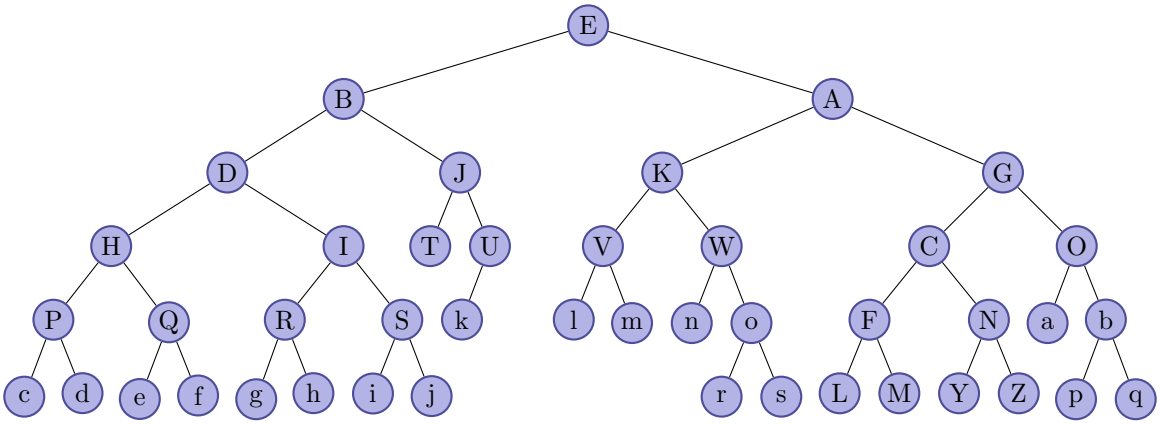


Solution

X is a leaf so we can just remove it. Now we need to rebalance the tree. X was the left child of M so the balance factor of M is increased from -1 to 0. This means that the tree rooted at M is balanced but has been shortened. So move up to M's parent F. Since we just shortened the right subtree of F, decrease F's balance factor (from 1 to 0). Move up to F's parent C. Since we just shortened the left subtree of F, increase the balance factor of C. Observe that the balance factor of C is now 2 and the balance factor of C.right is 1. So, we need to do a single left rotation on C which gives us the following tree.



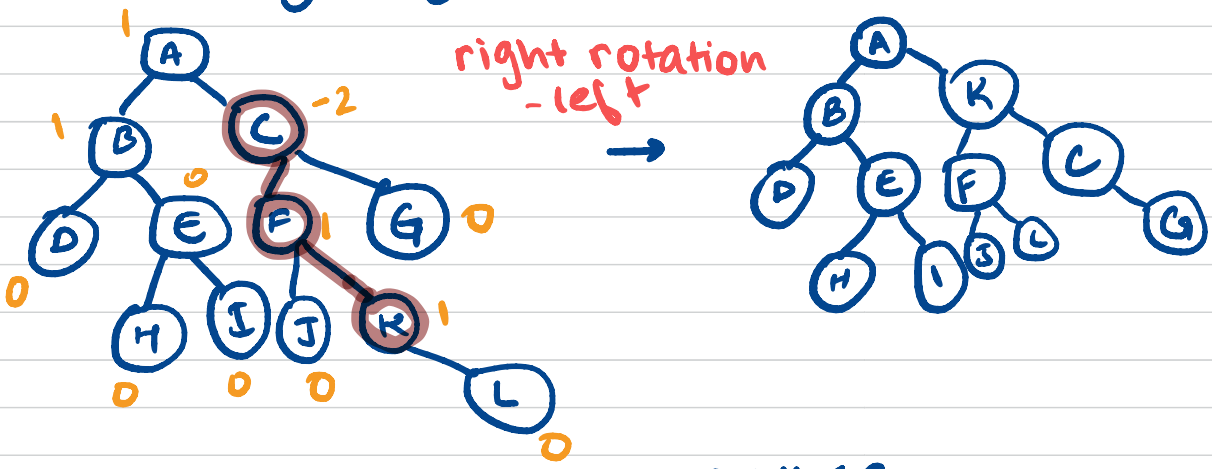
Doing a single rotation, decreases the height of the subtree on which it was performed. So we have just decreased the height of the right subtree of A. We move to A and decrease the balance factor of A. It was already -1 and so now it is -2 and we will need to rotate. A's left child B has a balance factor of +1, so we need a double left right (left on B, right on A) to fix this. The final tree appears on the next page. Now that we are at the root, we are done.


E
B
A
D
J
K
G
H
I
T
U
V
W
C
O
P
Q
R
S
k
l
m
n
o
F
N
a
b
c
d
e
f
g
h
i
j
r
s
L
M
Y
Z
p
q

WEEK 4 PREP

1. AVL tree →
   while inserting at most 3 rotations
   while deleting at most log(n) rotations

2. Max depth between 2 leafs of an AVL tree can be anything!

3. 

F < K < C

4. Delete (13) Replace with successor