7/8/2020 - AVL Trees

Wednesday, July 8, 2020 4:01 PM

Seg faults - array out of bounds, using an arrow on a pointer that isn't pointing to a struct.

arr[i] - I is out of bounds
p->weight - but p isn't pointing to an actual struct

```
struct person* p = NULL;
struct person* q;
strcpy(p->name, "Bob");
strcpy(q->name, "Alice");
```

p → X

q → ?

main
------
trienode* root = init();
readStuff(root)

root →

readStuff(trienode* root))
----
root

insert(root, …)

insert(trienode* root, …)

root

Binary Search Trees

Avg Case - insert, search, delete is O(lg n), n = # items in tree
Worst Case is O(n), if the tree is badly balanced.
Easy to create data: insert 1, insert 2, insert 3, …
It would be nice to use the same Binary Search Tree idea, but guarantee a worst case O(lg n) run time for insert, search, delete.
Looking for: A "balanced" binary search tree.
Run times are all O(h), where h is the height of the tree, so if we can "limit" the height, that would be great.

First Discovered Balanced Binary Search Tree is called an AVL Tree.
(named after inventors Adelson-velsky and Landis)

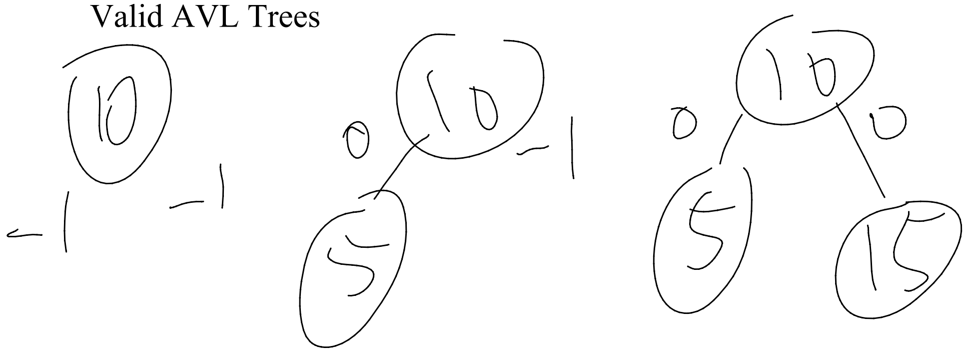
From <https://en.wikipedia.org/wiki/AVL_tree>

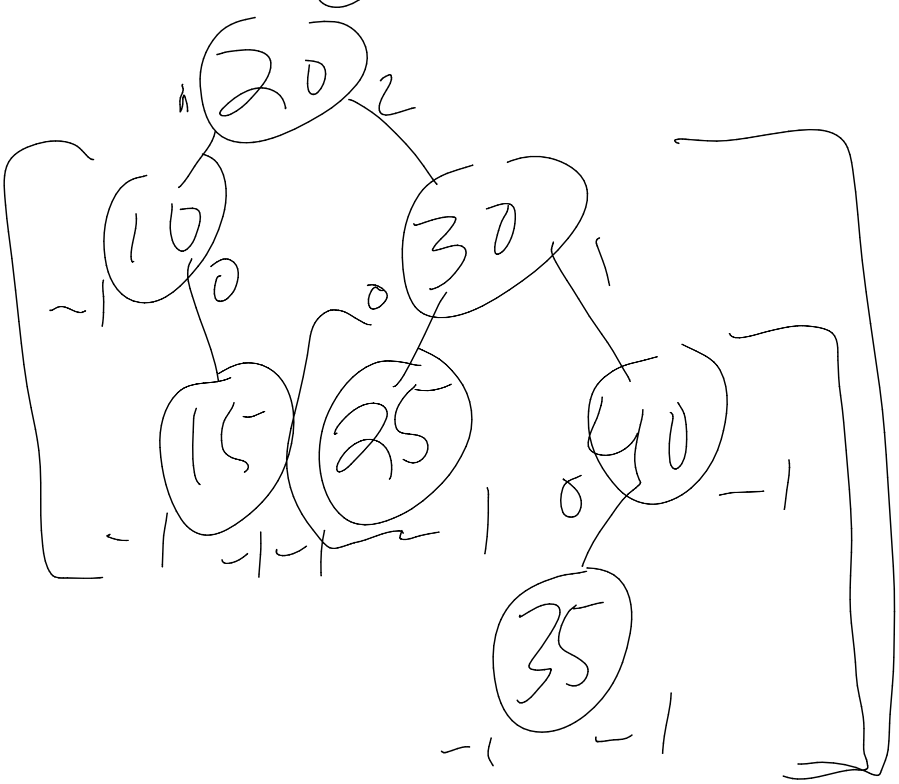
Imagine creating a BST, but trying to force the heights of the left and the right to be close to one another (specifically, within 1.)

So for every node, we require that | height(left) - height(right) | <= 1

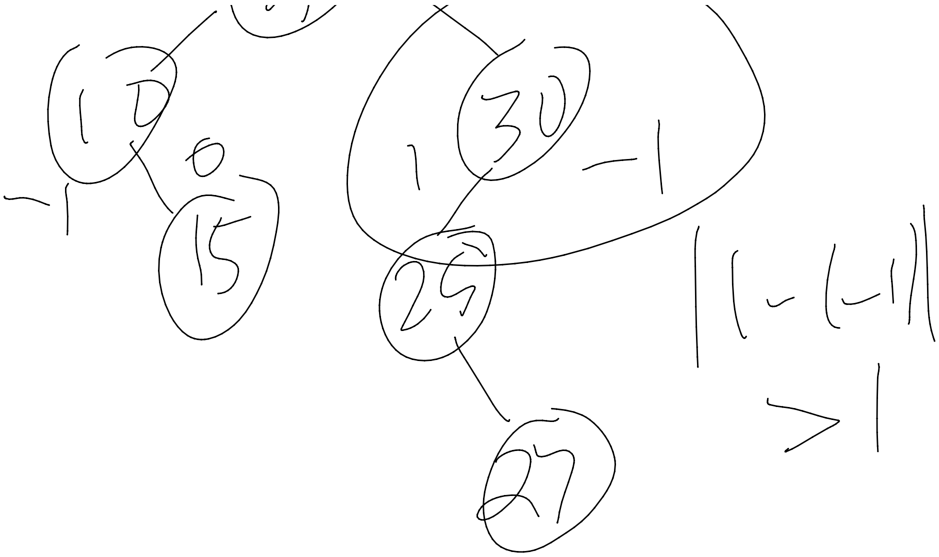
Of course, we keep the same search tree property (go left for smaller items, right for larger ones)

Valid AVL Trees





Invalid



Some inserts will make the property fail (as will some deletes).
How do we fix the tree?

First though, let's prove that any binary search tree that adheres to this properly will give us a tree that has a height h = O(lg n), where n is the number of nodes in the tree.

Let T(h) = fewest number of nodes in a AVL tree of height h.
We will prove that T(h) = F_{h+3} - 1, where F_n = the nth Fibonacci number.

Fib: 1, 1, 2, 3, 5, 8, 13, 21, 34 $F_1 = 1$, $F_2 = 1$, $F_n = F_{n-1} + F_{n-2}$.

T(0) = 1, T(1) = 2, T(2) = 4

Use mathematical induction on h to prove the theorem.

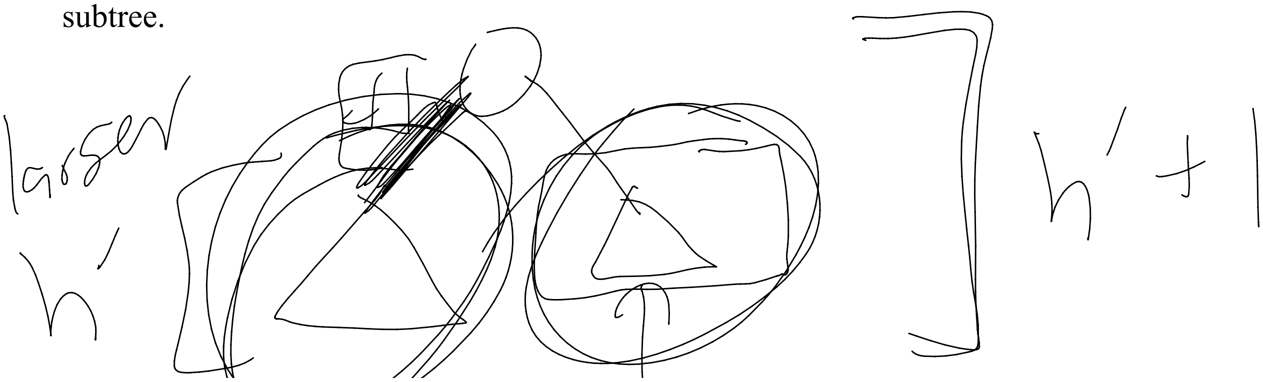
Assume that this is true for all values h less than or equal to arbitrarily chosen h'. (Assume it's true for T(0), T(1), T(2), …, T(h')

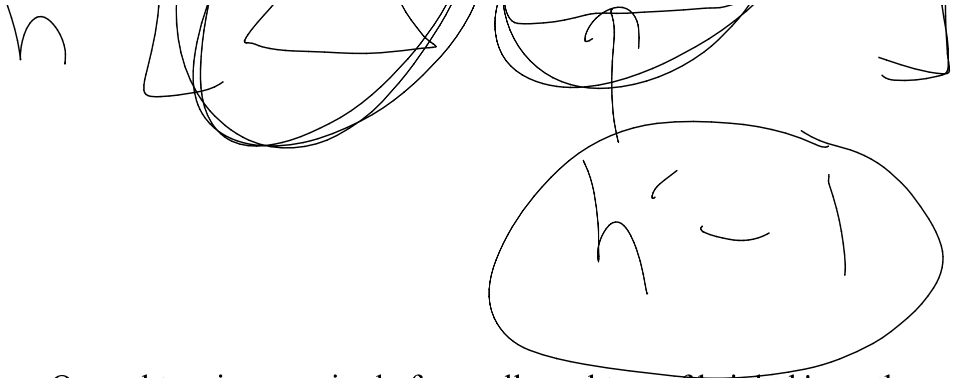
Prove the statement is true for h'+1

T(h'+1) = $F_{h'+4}$ - 1

Proof

Imagine trying to form a valid AVL tree with height h'+1 with as few nodes as possible. We must have a root node, a left subtree and a right subtree.





Our avl tree is comprised of a smaller avl tree of height h', another smaller avl tree of height h'-1 and the root.

T(h'+1) = T(h') + T(h'-1) + 1 (add fewest # of nodes larger subtree, plus the fewest # of nodes in smaller subtree plus the root)

$= (F_{h'+3} - 1) + (F_{h'+2} - 1) + 1$
$= F_{h'+4} - 1$ (this completes the proof.)

So, let n = fewest # of nodes in an avl tree of height h:

$n = F_{h+3} - 1$
$(n+1) = F_{h+3}$
$(n+1) \sim \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+3}$

Golden Ratio ~ 1.6

$$\sqrt{5}(n+1) = 1.6^{h+3}$$

$$\log_2 \sqrt{5}(n+1) = \log_2 1.6^{h+3}$$

$$\log_2(\sqrt{5}(n+1)) = (h+3)(.67)$$

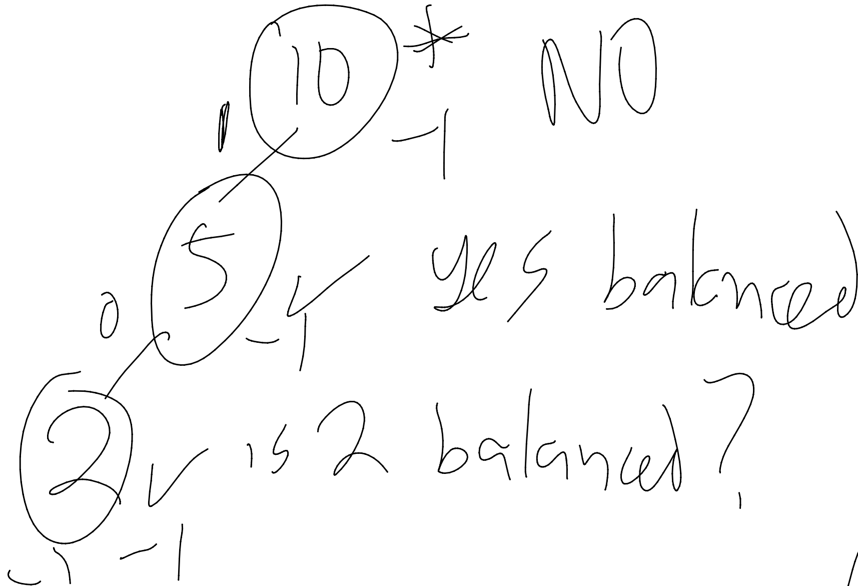
$$c\log_2(\sqrt{5}(n+1)) - 3 = h$$

$$= O(\lg n)$$

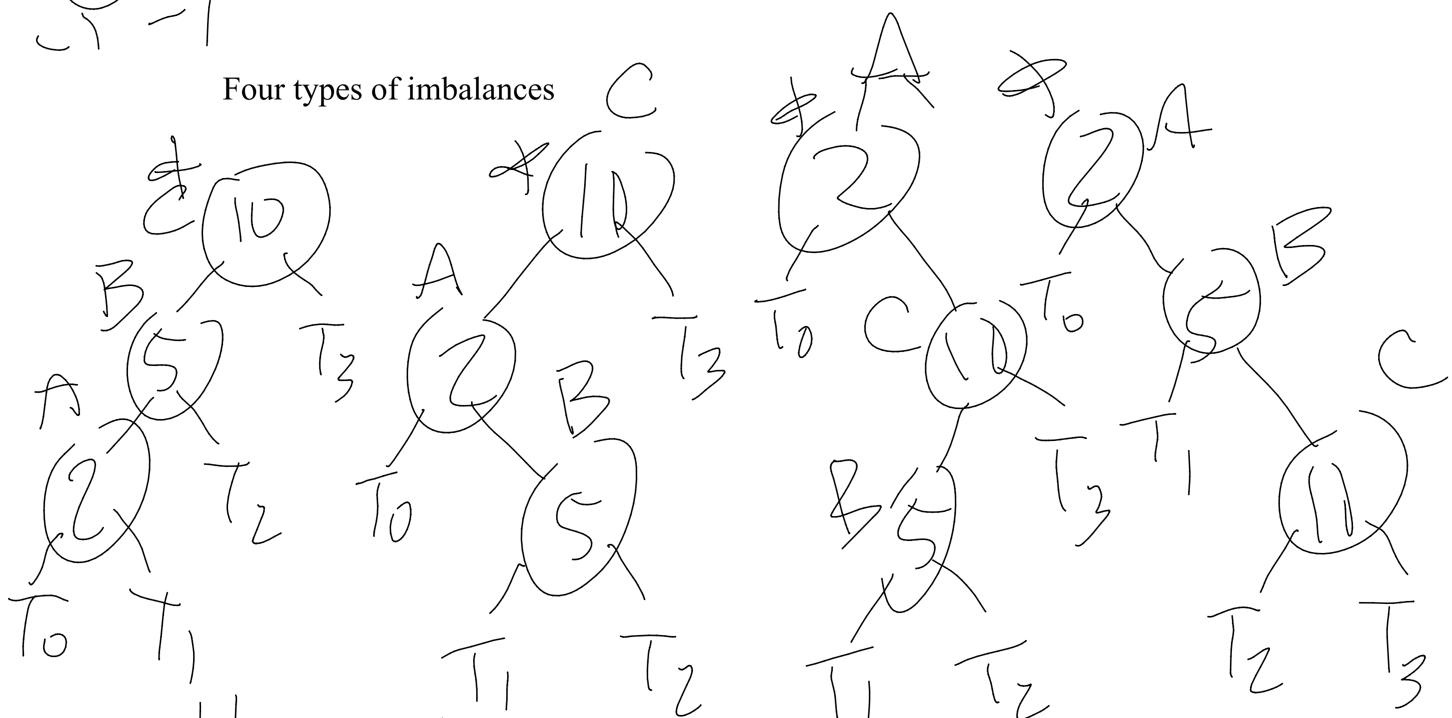
How do I maintain this property?

There is a rebalance operation which will occur any time a node is unbalanced.

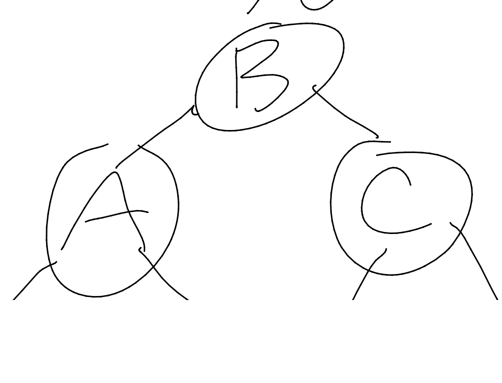
I'll teach through example.

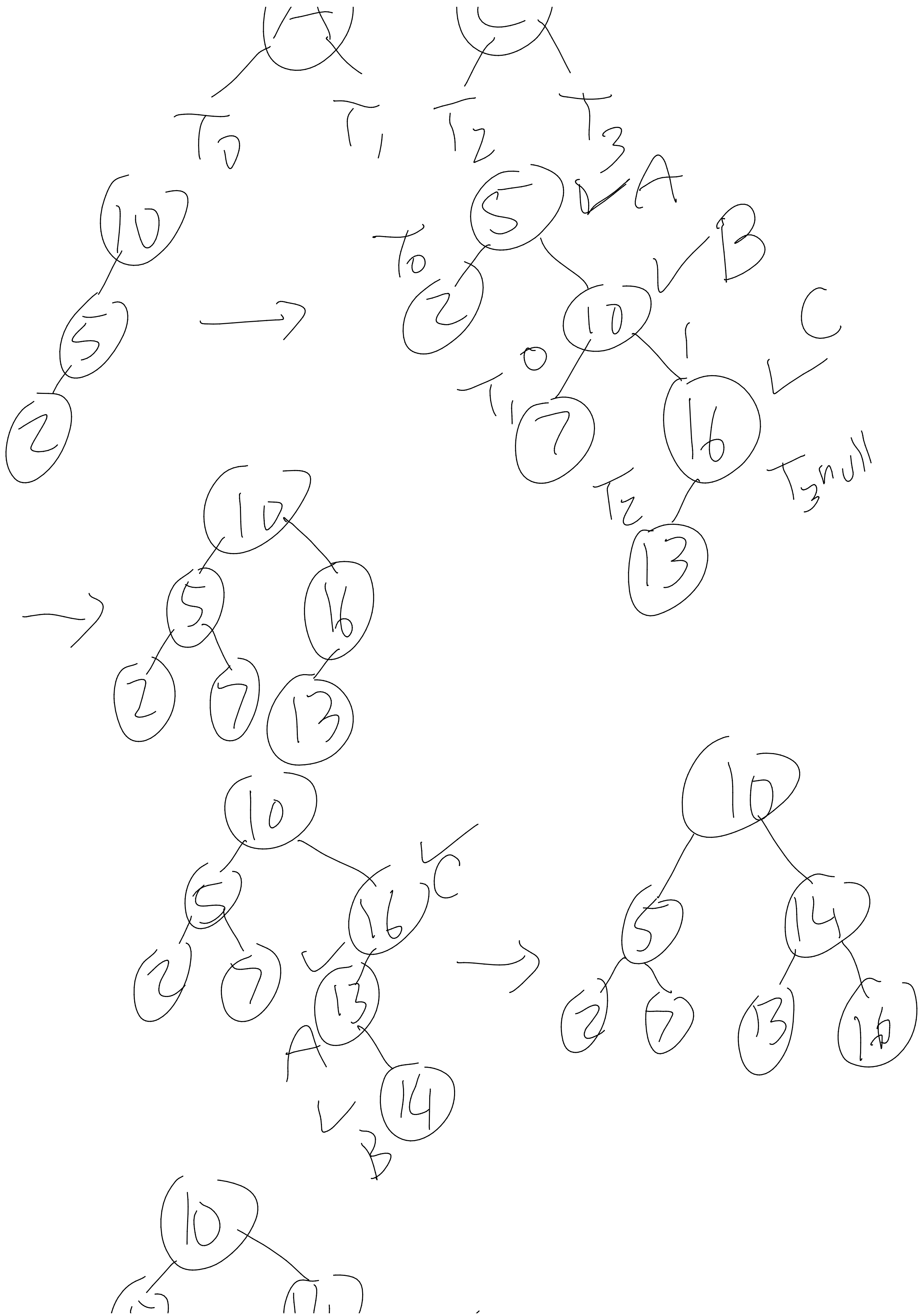


Four types of imbalances

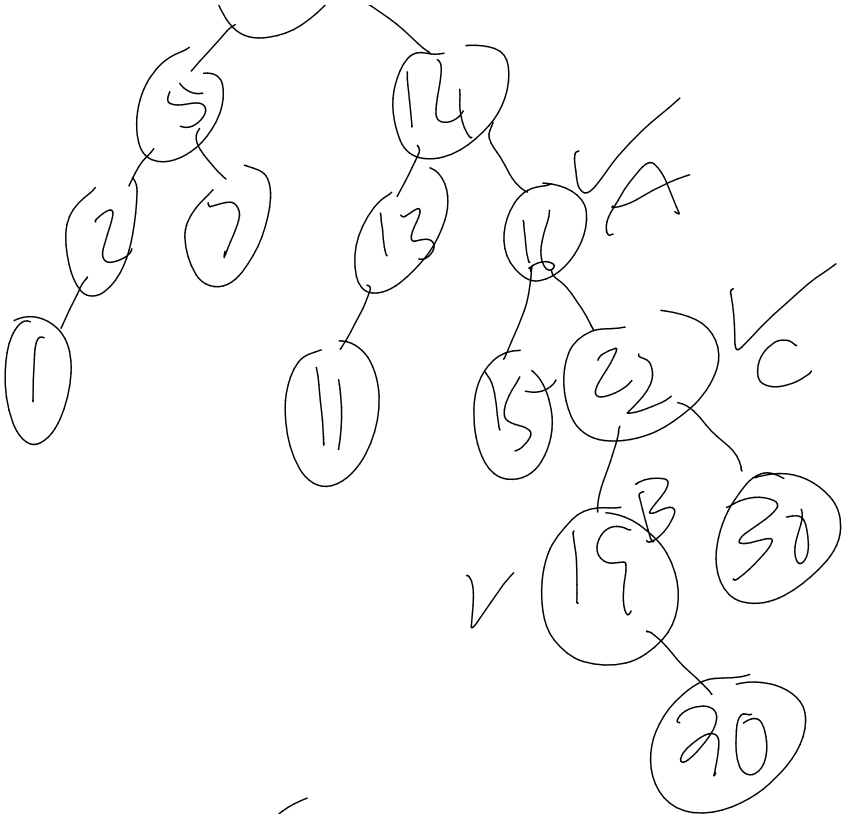


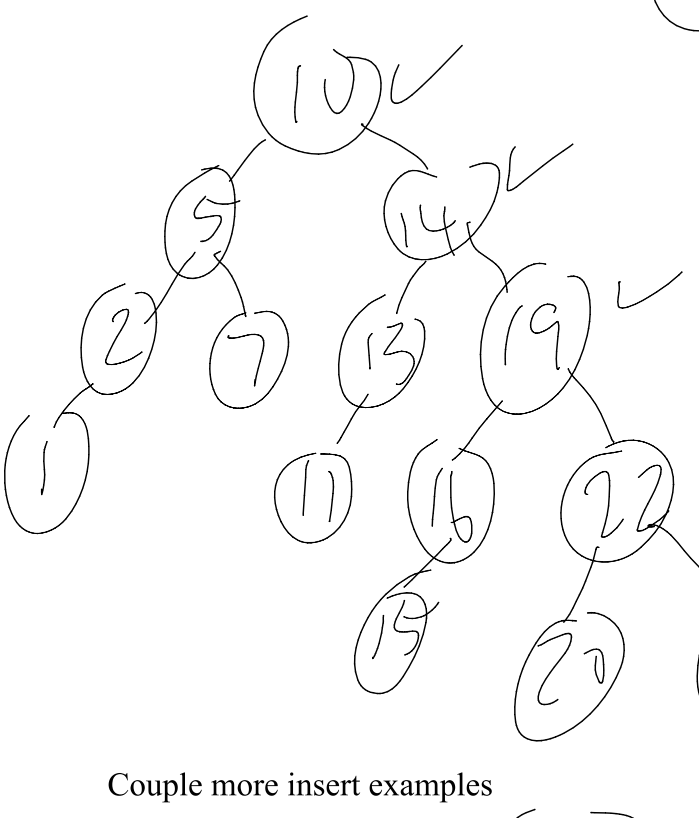
all get fixed like this





T0
T1
T2
T3
A
B
C
T0
T1
T2
T3 null
0
1
10
5
2
7
16
13
C
A
B
14

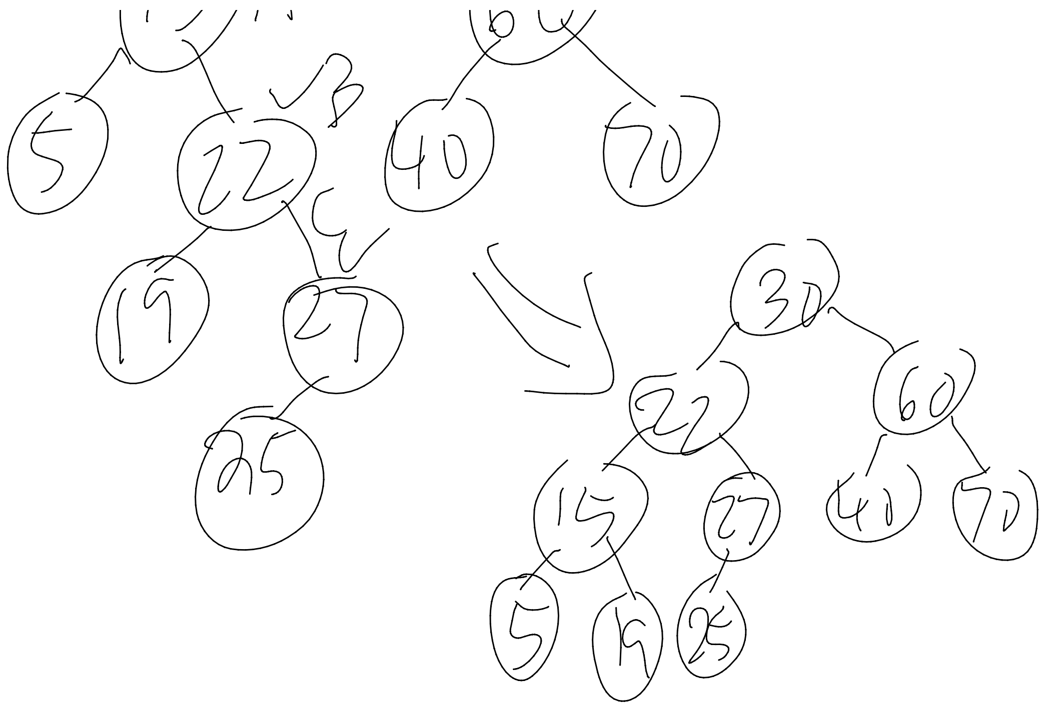


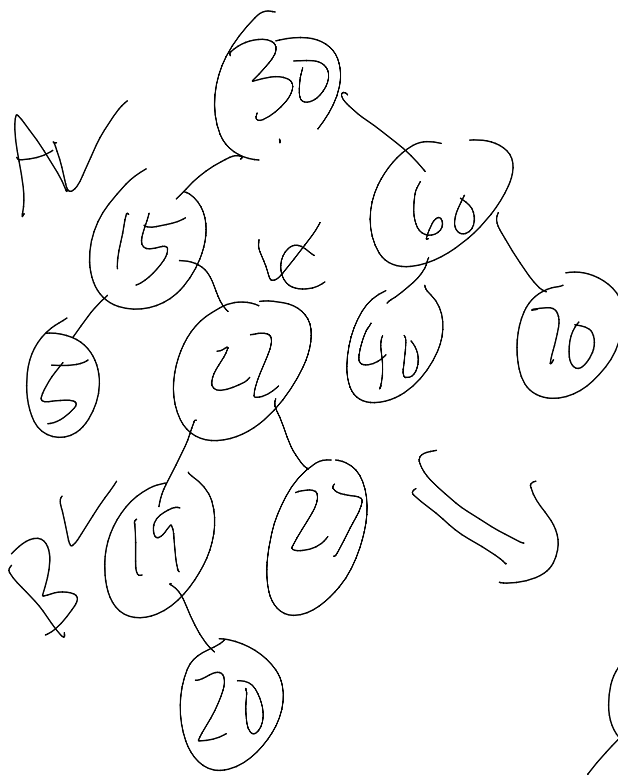


Algorithm:
Insert, then trace back up the ancestral path. For each node on this path, see if it's imbalanced. If so, rebalance it, and continue up the path. It can be shown that after an insert, no more than one rebalance will ever be done.

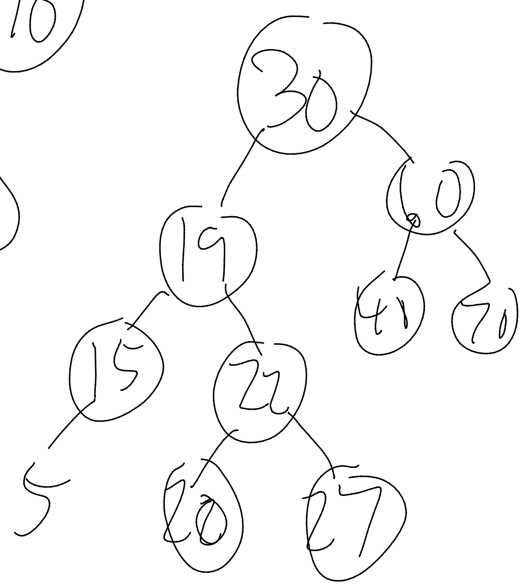
Couple more insert examples

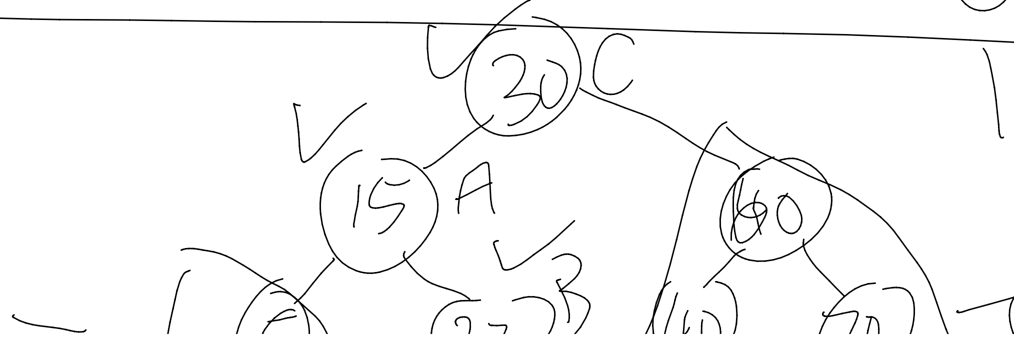
Insert 25





Insert 20



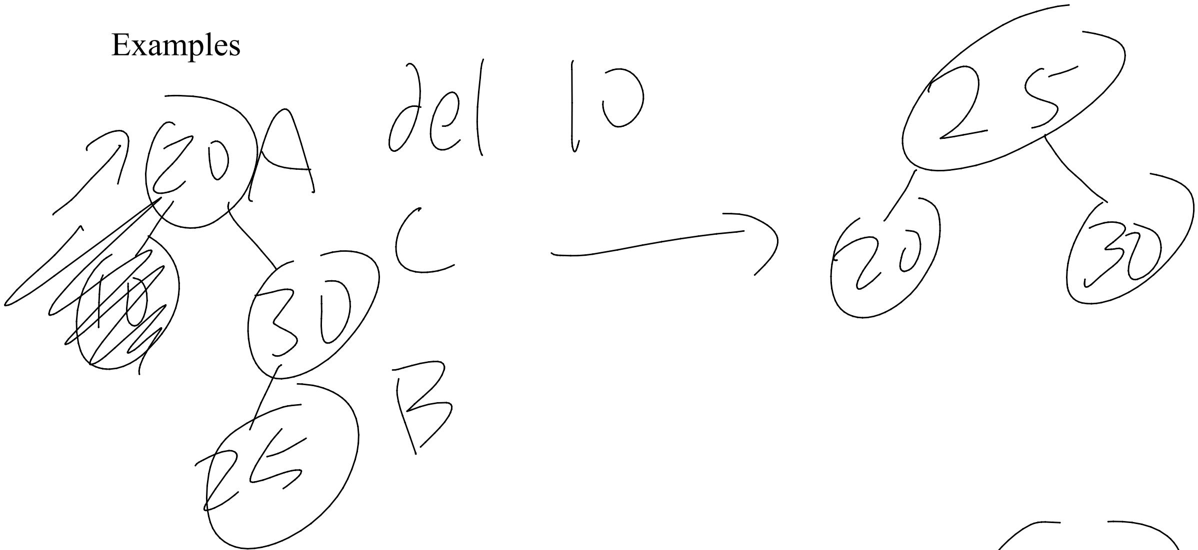


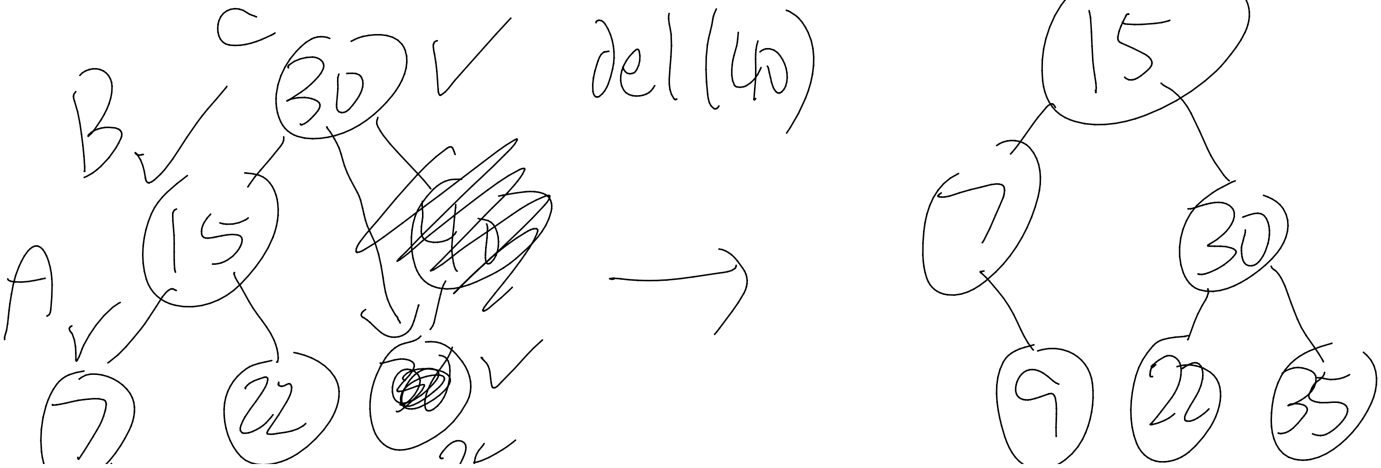
Insert 17
Insert 7

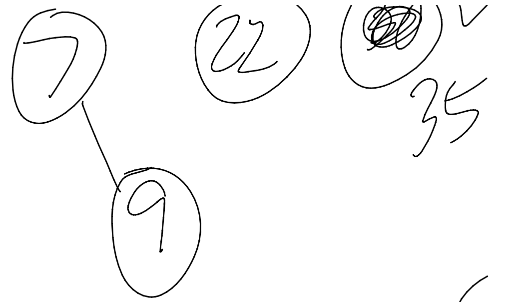
Delete:
We first do the delete (will always be either a leaf node or a node with one child.)

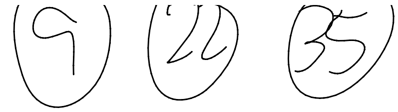
If leaf, we will basically do the same thing as insert, trace up the ancestral path. If we see an imbalance, we fix it using the exact same procedure as before. Then continue tracing up the tree. But, for delete, we may have a rebalance at multiple levels.

Examples



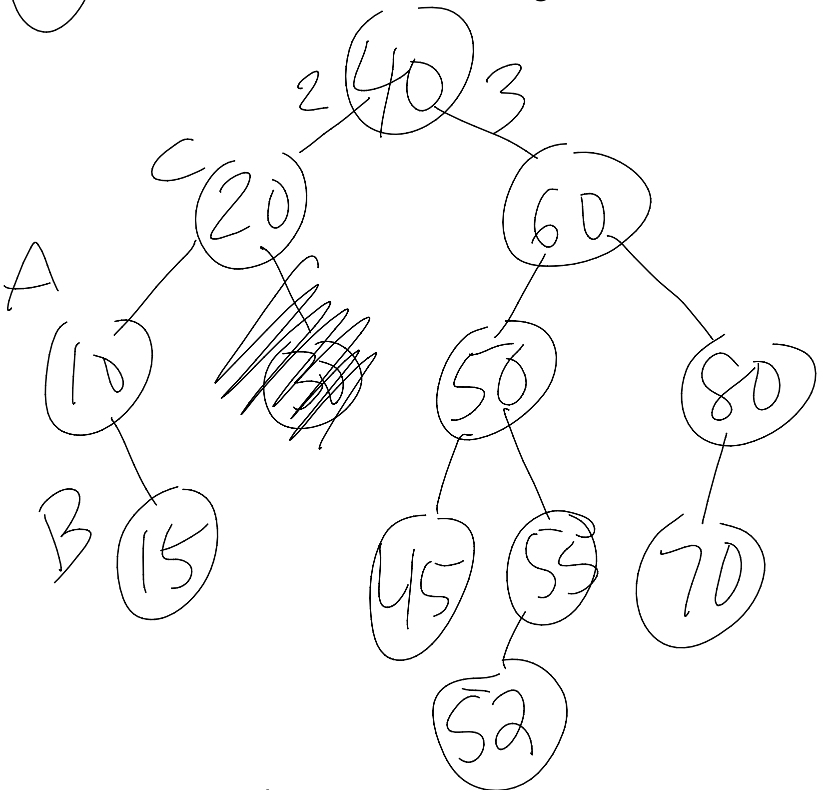


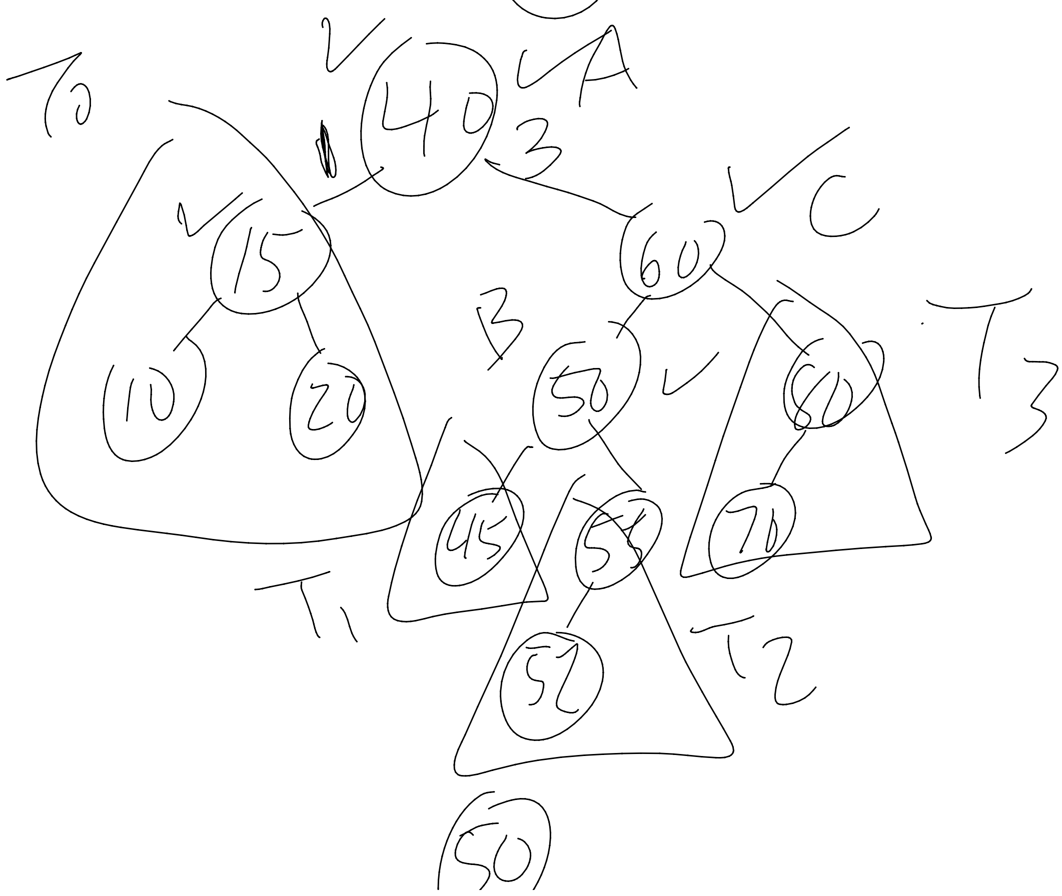


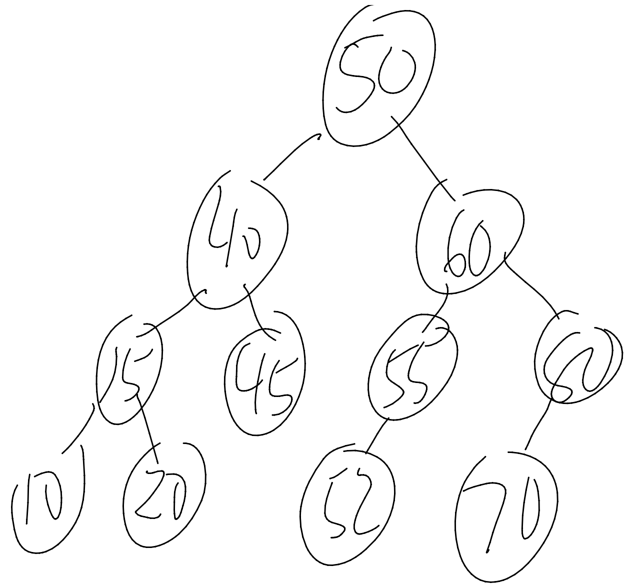


Tougher part about delete is that when a node is imbalanced, it's not the trail you previously traced. Rather, you have to go to the longer subtree, and then its longer subtree. If there is a tie, you can go either way.

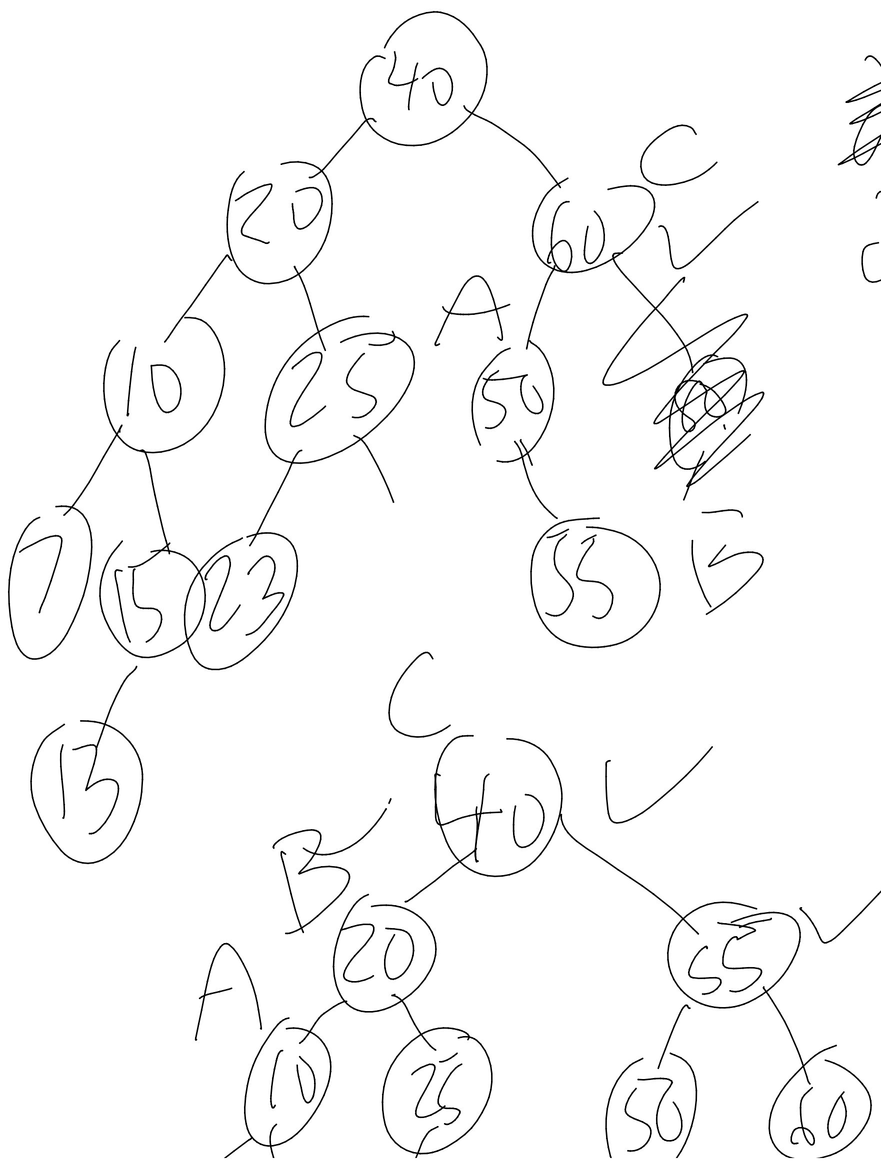
del(30)



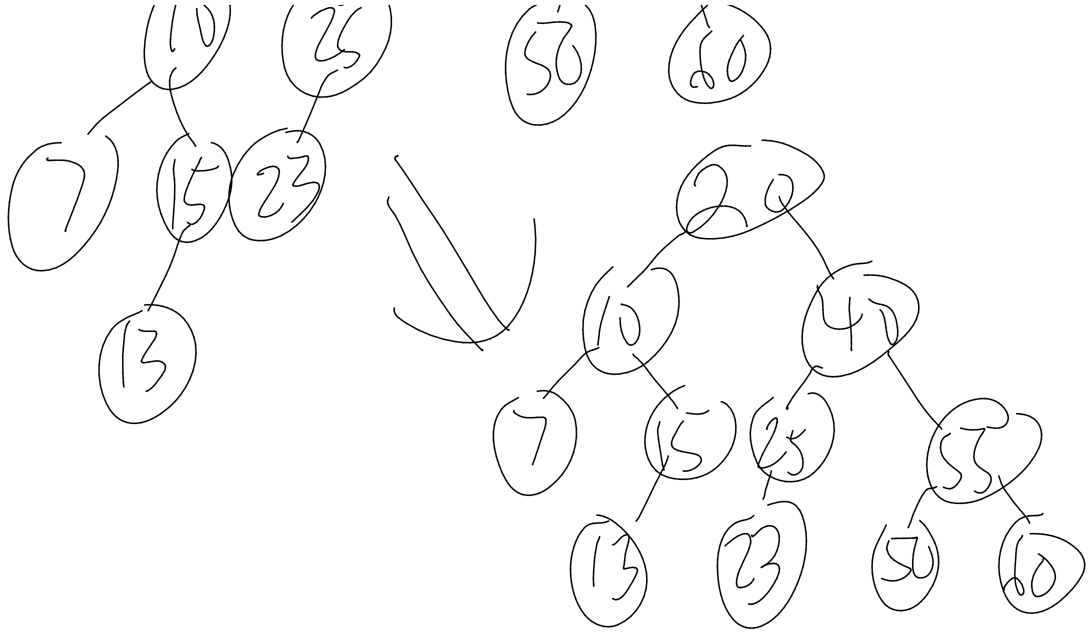




Run time proofs:
regular insert, delete is O(lg n), since h = O(lg n).
As we go back up the tree, we do at most O(lg n) rebalances, and each rebalance is O(1) time. To total time is O(lg n) + O(lg n)*O(1) = O(lg n)



~~del~~ del(80)



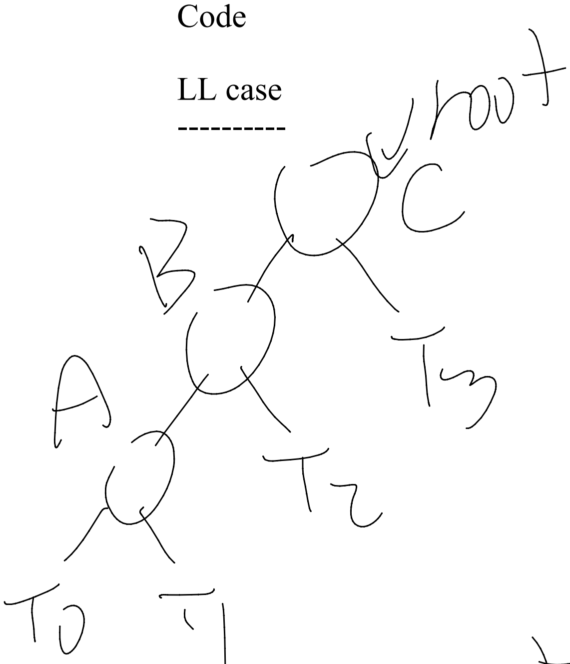
```
int a = x > 0 ? 4 : 5

int a;
if (x > 0)
   a = 4;
else
   a = 5;
```

This is what the question mark operator does.

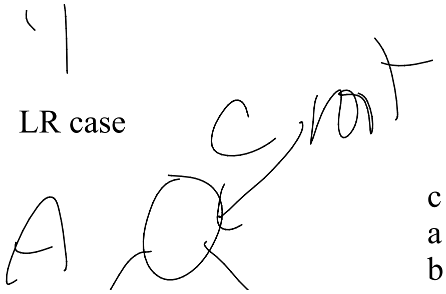
Code

LL case

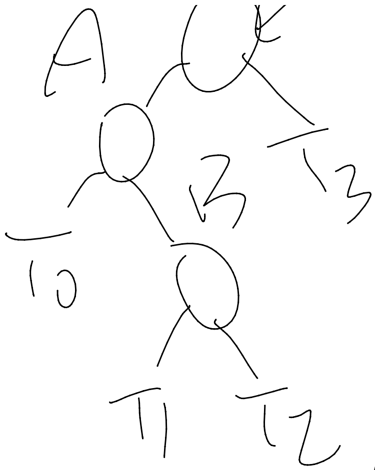


c = root
b = root->left
a = root->left->left
t0 = a->left
t1 = a->right
t2 = b->right
t3 = c->right
yellow is true for all 4 cases

LR case

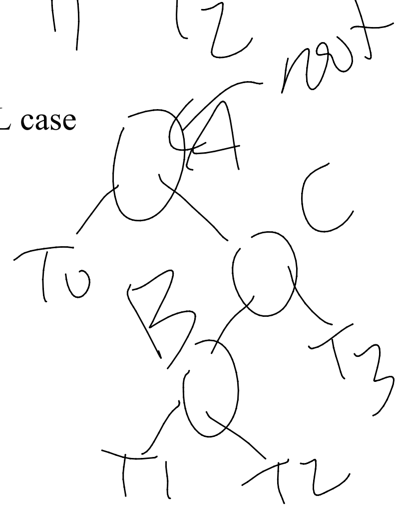


c = root
a = root->left
b = root->left->right



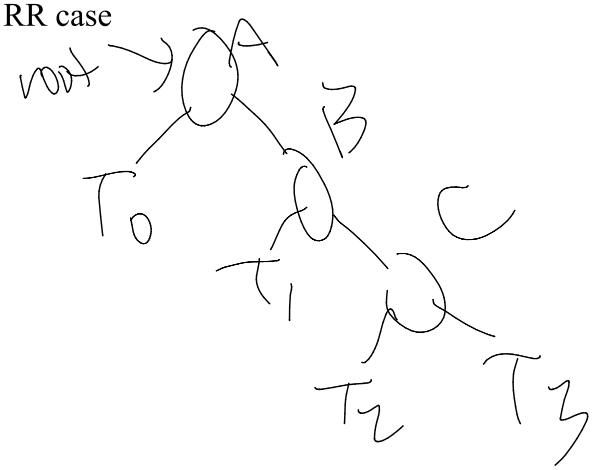
c = root
a = root->left
b = root->left->right
t1 = b->left
t2 = b->right

RL case



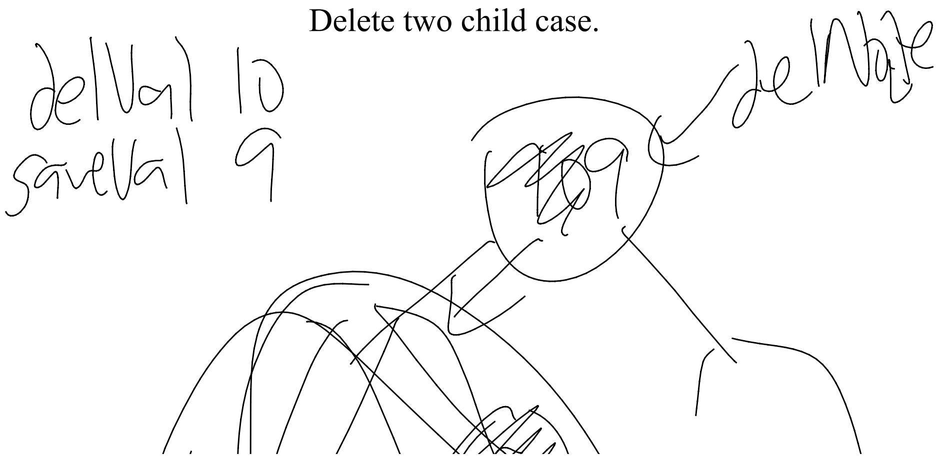
a = root
c = root->right
b = root->right->left
t1 = b->left
t2 = b->right

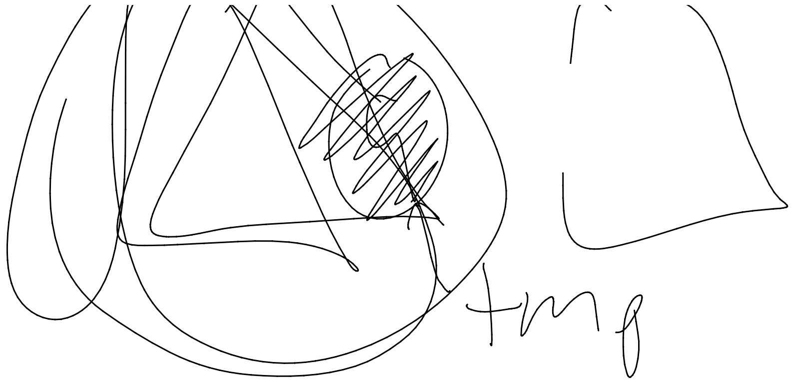
RR case



a = root
b = root->right
c = root->right->right
t1 = b->left
t2 = c->left

Delete two child case.




tmp