AVL Trees

CMPT-225

Recall: • A BST is
- a binary tree
- with nodes labelled by keys
- for every two nodes u, v:
  - if u is in the left subtree of v then $key(u) < key(v)$
  - if u is in the right subtree of v, then $key(u) > key(v)$

• BST operations take time proportional to the tree height, which might be the same as the number of keys.

• AVL Trees are a kind of "self-balancing" BST. Their height is always at most $2\log_2 n$, where n is number of keys.

- An **AVL Tree** is a BST that satisfies the following **height-balance invariant**:

  For every node v,

  $$|\text{height}(\text{left}(v)) - \text{height}(\text{right}(v))| \leq 1$$

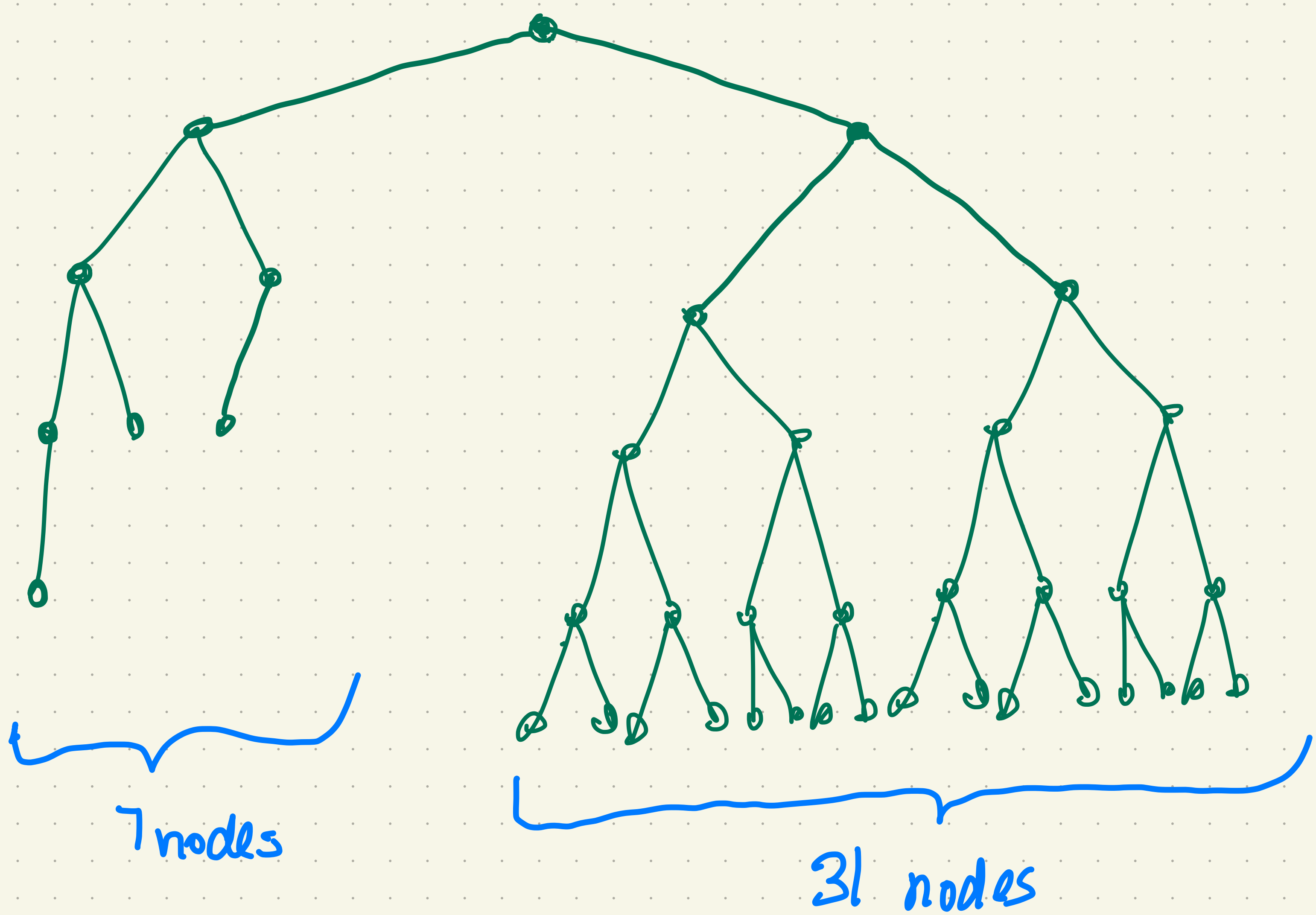
  (We define height(left(v)) = -1 if left(v) does not exist, + similarly for right(v)).

- Implementing the Operations:

  1. Perform BST operation, then
  2. repair balance if needed.

How unbalanced can an AVL tree be?

Ex: A "maximally unbalanced" height-5 AVL Tree



How tall can an AVL Tree be?

Let $N(h)$ = min. # of nodes in an AVL tree of height h.

<u>Observe</u>: $N(0) = 1$

$N(1) = 2$

$$N(h) = N(h-1) + N(h-2) + 1$$

$$> 2N(h-2)$$

$$> 2 \cdot 2N(h-4)$$

$$> 2^3 N(h-6)$$

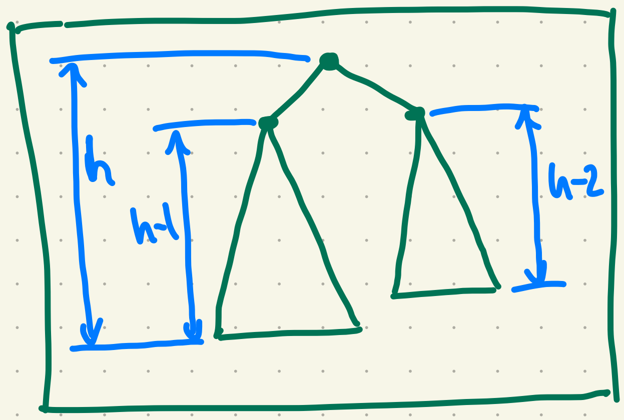
$\vdots$

$$> 2^i N(\underbrace{h-2i})$$

If h is even, we end when $h - 2i = 0 \Rightarrow i = h/2$

$\vdots$

$$> 2^y \cdot c = 2^{h/2} \cdot 1$$



Claim: $N(h) > 2^{h/2}$

Proposition: $N(h) > 2^{h/2}$ (for all $h \geq 0$)

Pf: By ind. on h.

Basis: $N(0) = 1 \geq 2^0 = 1$ ✓

$N(1) = 2 > 2^{1/2} = \sqrt{2}$ ✓

Assume, for some $h \geq 1$, that $N(h) > 2^{h/2}$

Now $N(h+1) > 2N(h-1) \geq 2 \cdot 2^{\frac{h-1}{2}} = 2^{1+\frac{h-1}{2}} = 2^{\frac{h-1+2}{2}} = 2^{\frac{h+1}{2}}$ ✓

So: $N(h) > 2^{h/2} \Rightarrow \log_2 N(h) > h/2 \Rightarrow h < 2 \log_2 N(h) \leq 2 \log_2 n$

We have: for every AVL tree with n nodes, and height h,

$$h < 2 \cdot \log_2 n = O(\log n)$$

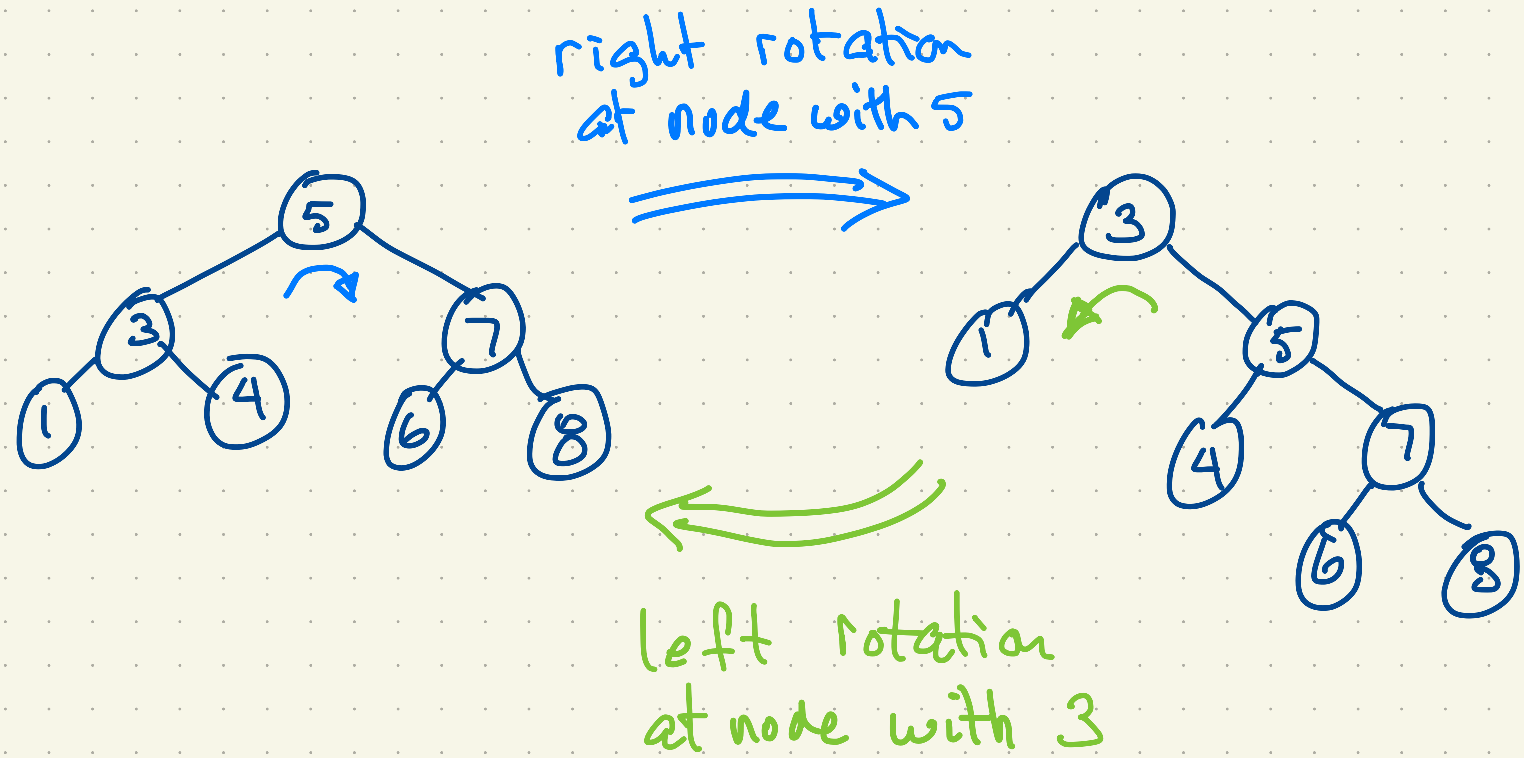
Thus: AVL Tree search takes time that is $O(\log n)$

Max AVL Tree height vs BST height

(Worst case # nodes visited by AVL-tree vs. BST search)

n	$2\log_2 n$
10	7
100	14
1000	20
10^4	27
10^5	33
10^6	40
10^7	47
10^8	53
10^9	60
10^{10}	66

Unbalanced sub-trees are "repaired" using rotations

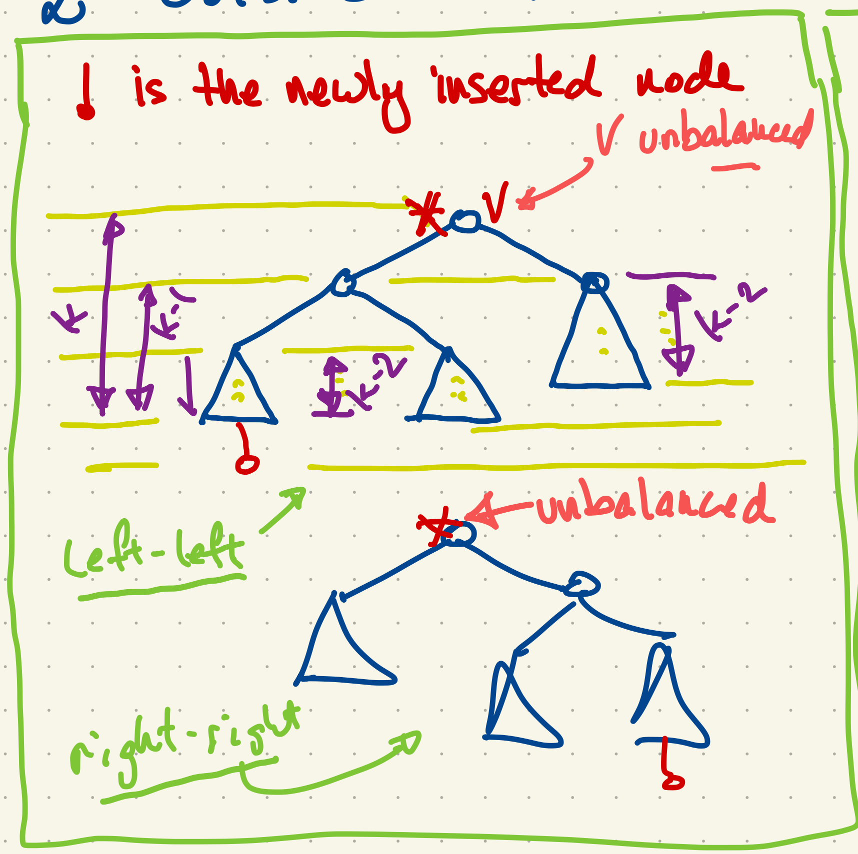


AVL Tree insertion.

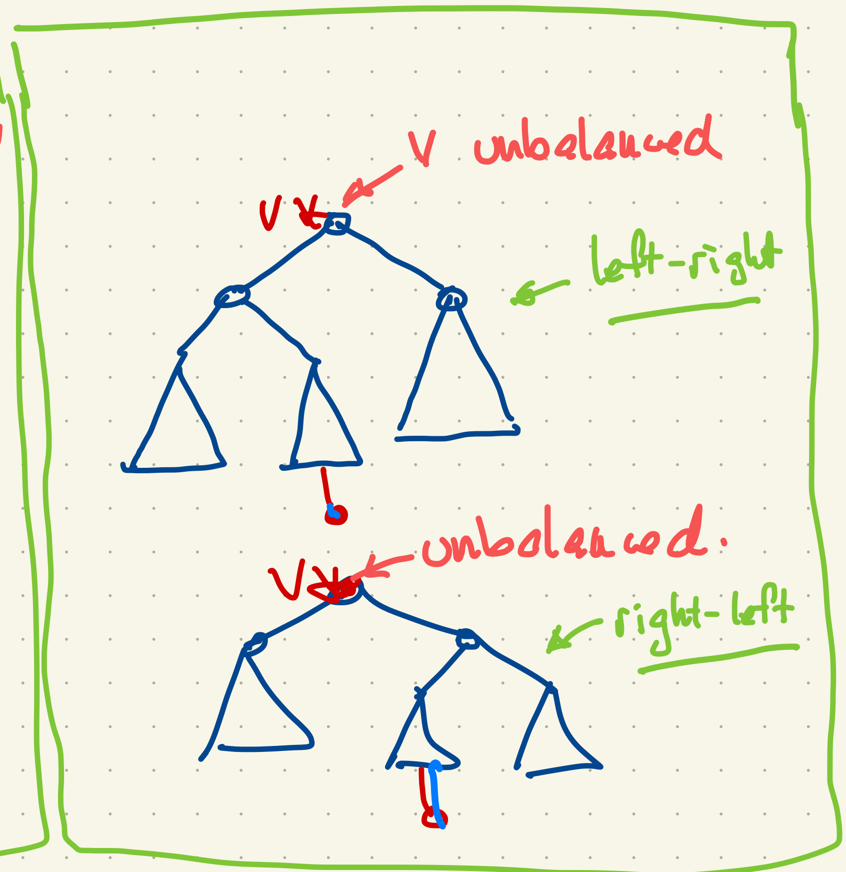
1. Do BST insertion.
2. If there is an unbalanced node,
   - let v be the unbalanced node of greatest depth.*
   - repair the imbalance at v.

Consider 4 cases:

2 "Outside" Cases

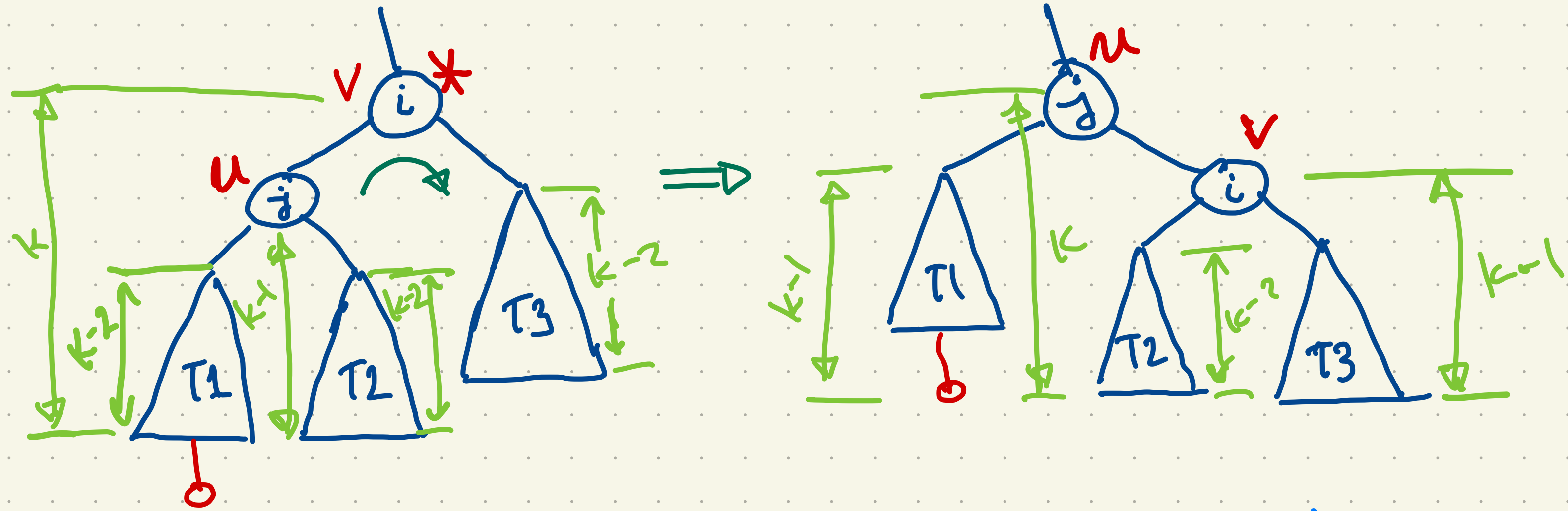


2 "Inside" Cases



* It must be on the path from the new leaf to the root.

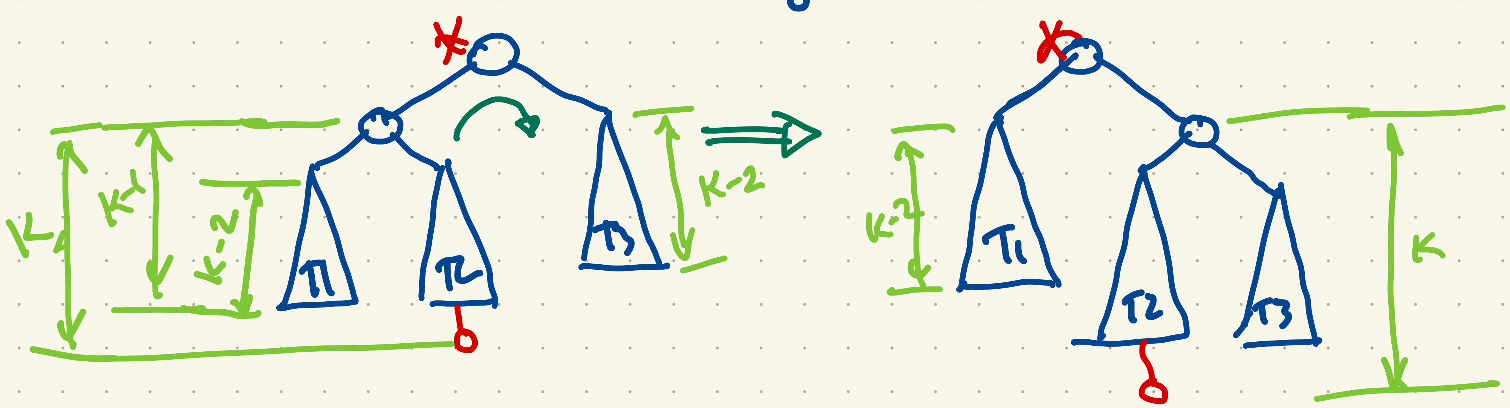
To fix the "outside" cases: do 1 rotation at the unbalanced node.



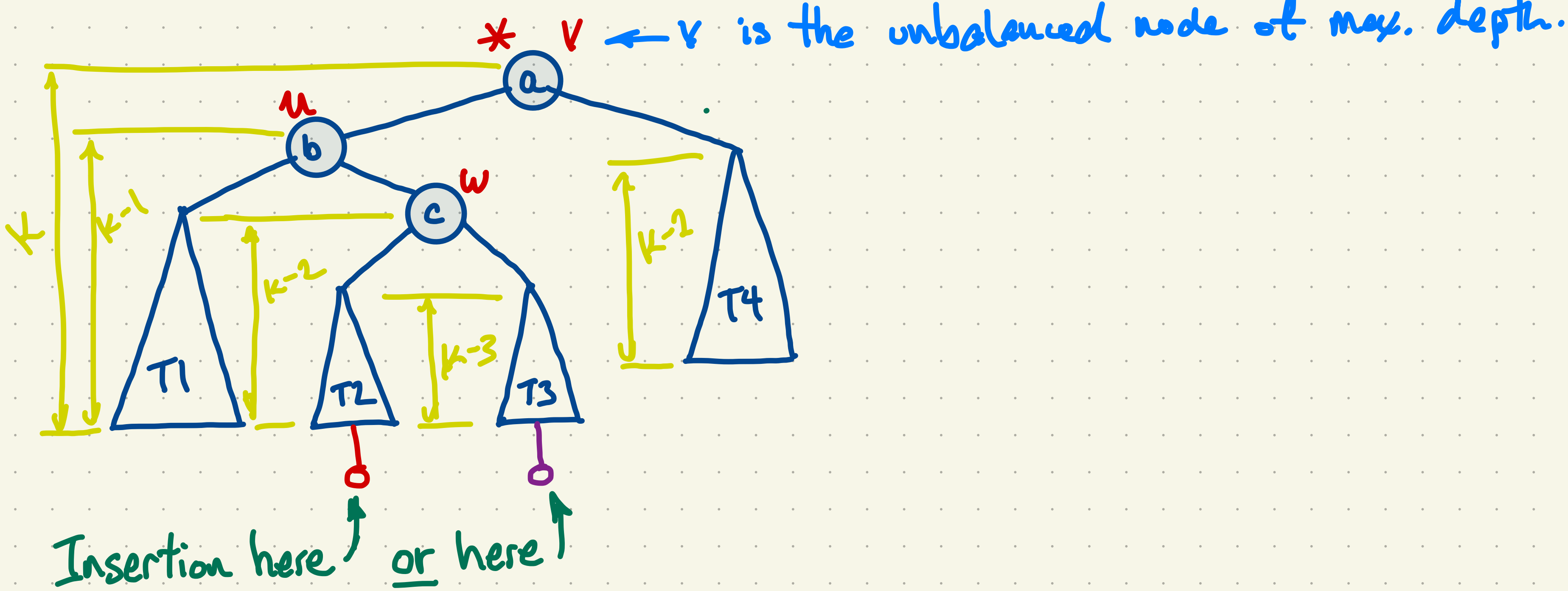
** The final height of u is k, so the tree now is balanced.

Exercise: 1. Draw the right-right case in detail
2. Draw them with minimal sized T1, T2, T3.

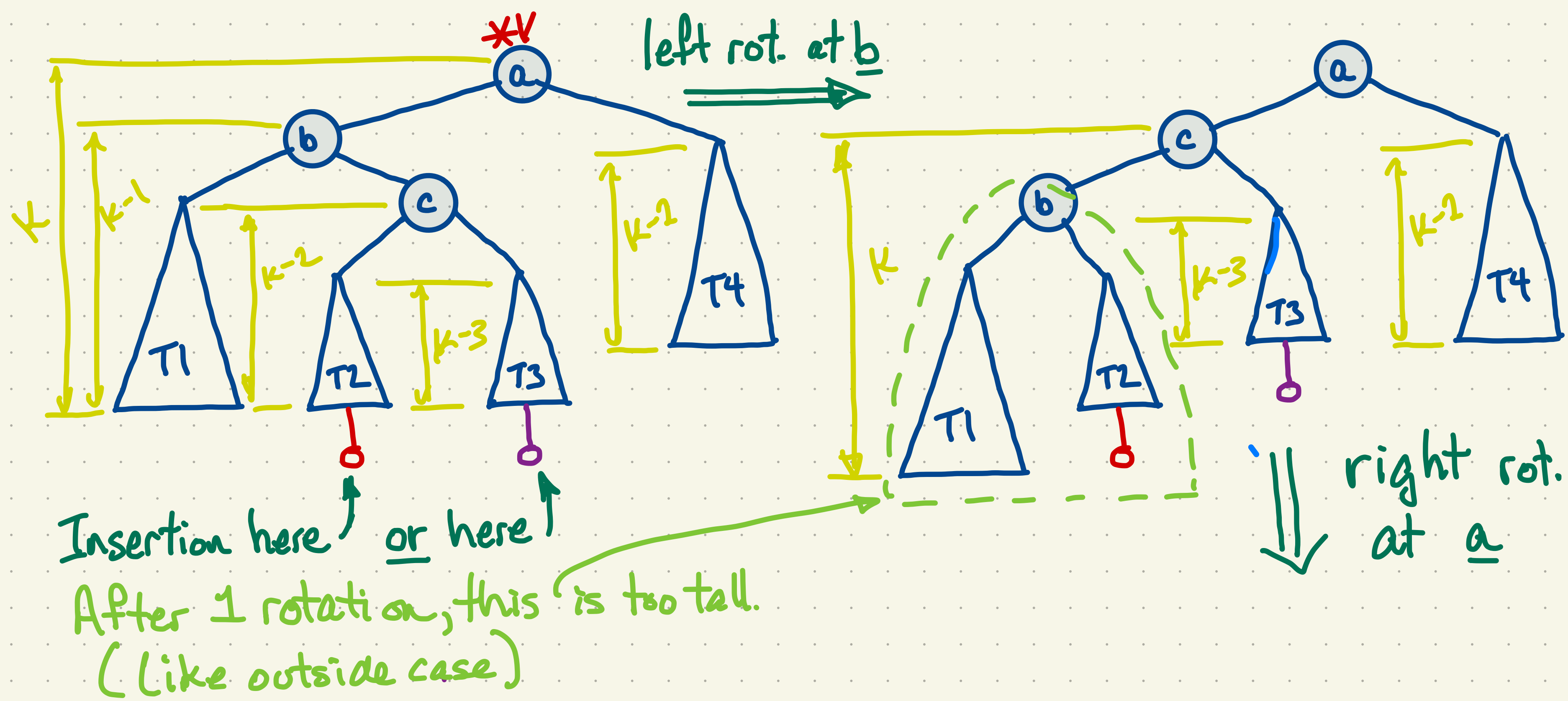
The "inside cases" are not fixed by this rotation:



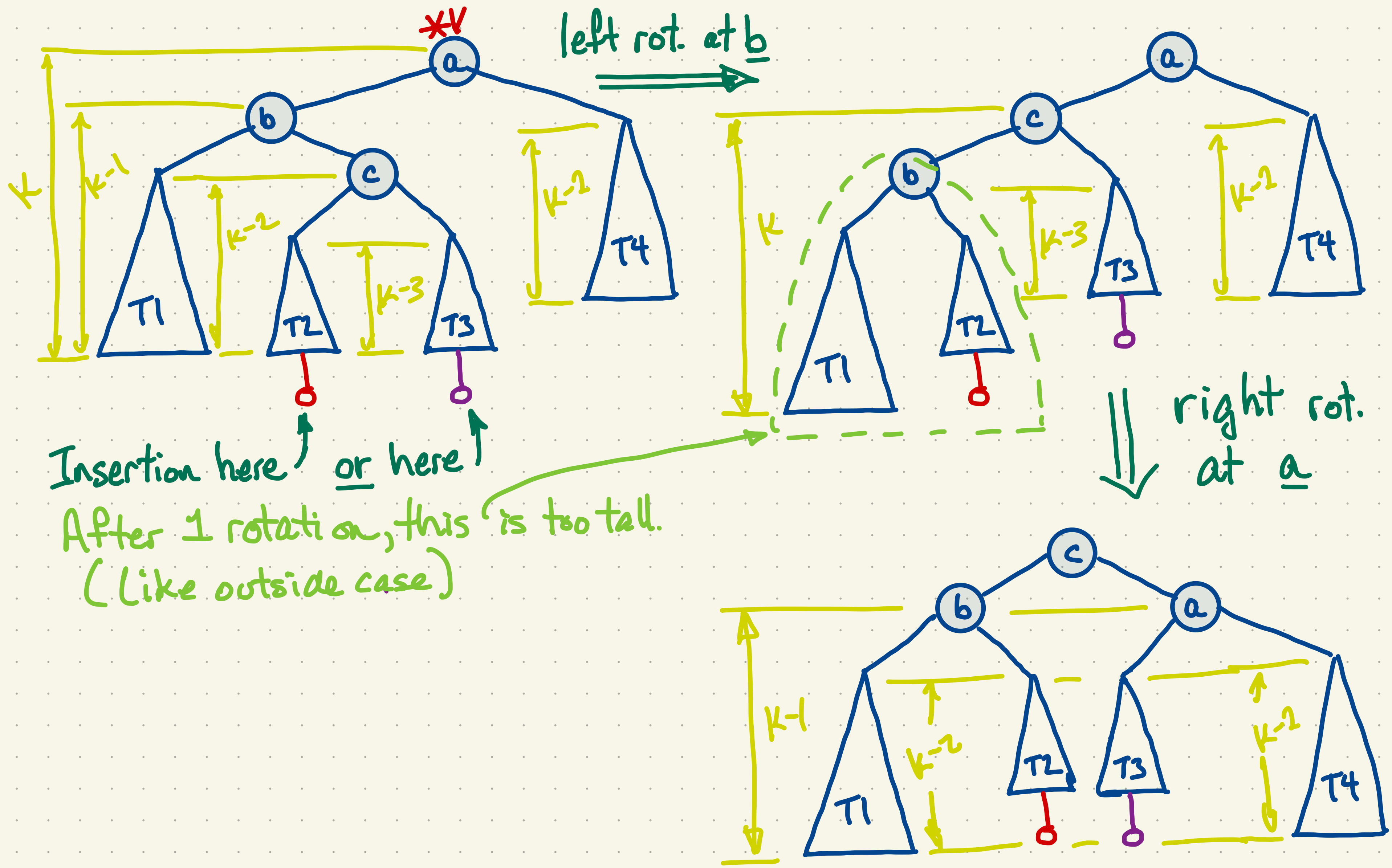
To fix the "inside" cases, we use two rotations:



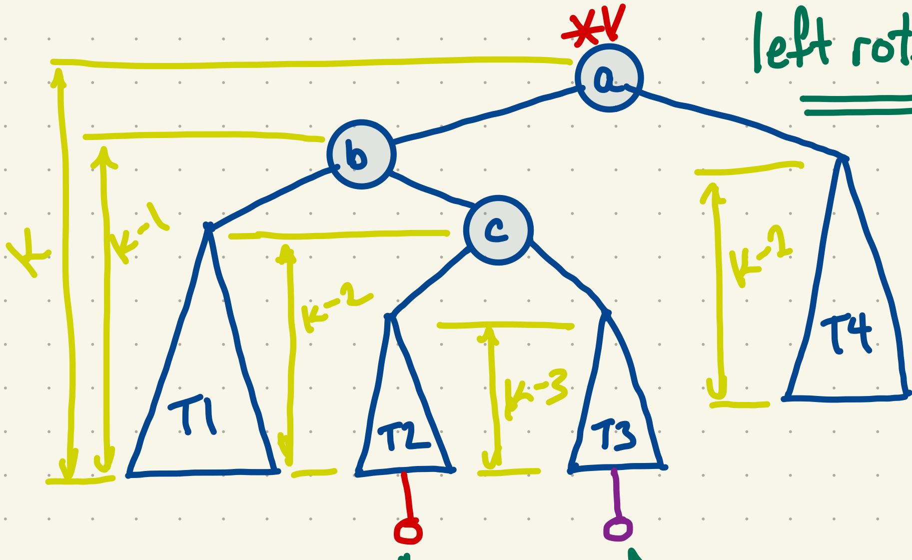
To fix the "inside" cases, we use two rotations:



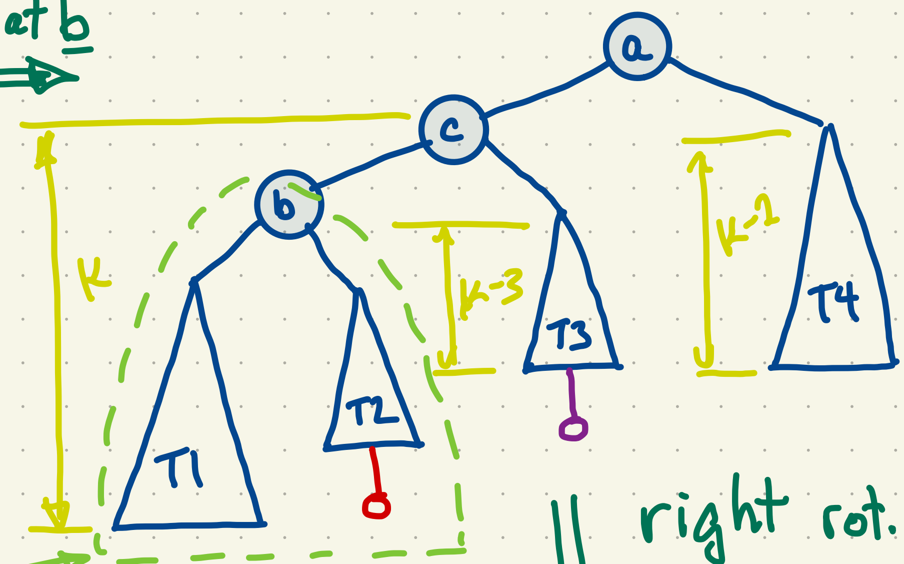
To fix the "inside" cases, we use two rotations:



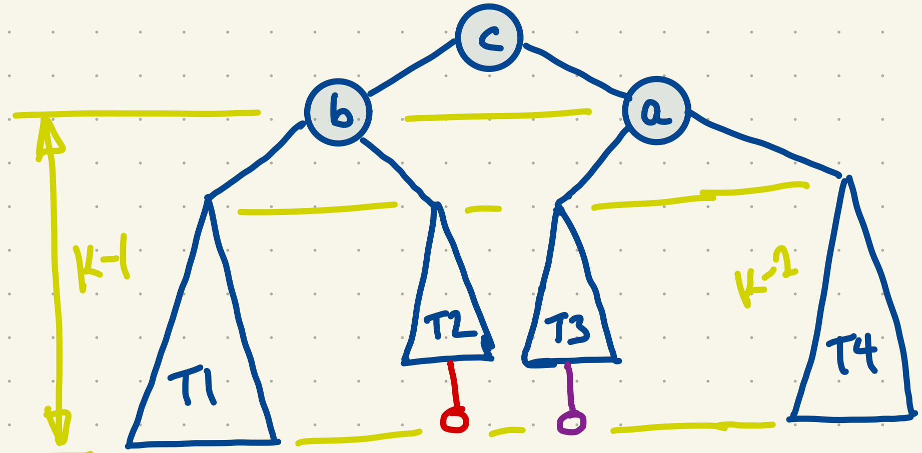
To fix the "inside" case, we use two rotations:



left rot. at b ⟹



right rot. at a ⇓



Insertion here or here

After 1 rotation, this is too tall.
(Like outside case)

The entire operation is:*
left(c) ← b
right(c) ← a
left(a) ← T3
right(b) ← T2
change parent(a) to be parent(c)

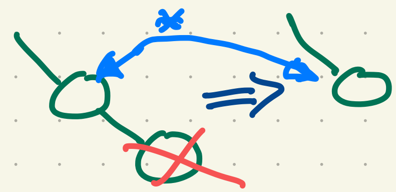
(* 1 rotation = 3 assignments; 2 rotations = 6 assigns; double rot. = 5 assigns.)

AVL Tree Removal.

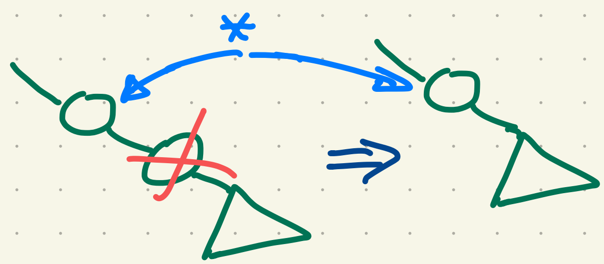
1. Do BST removal.
2. Rebalance.

Define "the parent of the deleted node" (*) by cases:

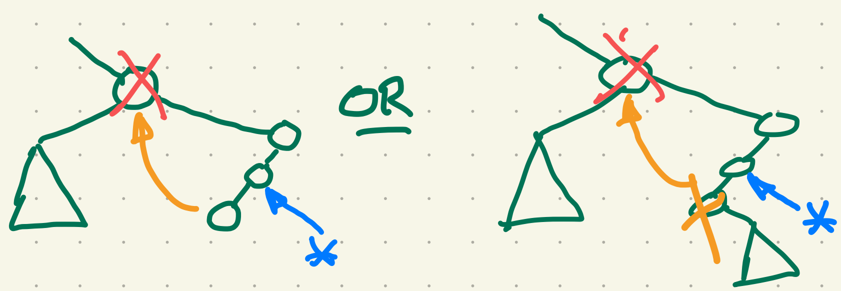
1. The deleted key was at a leaf:



2. The deleted key was at a node with one child:



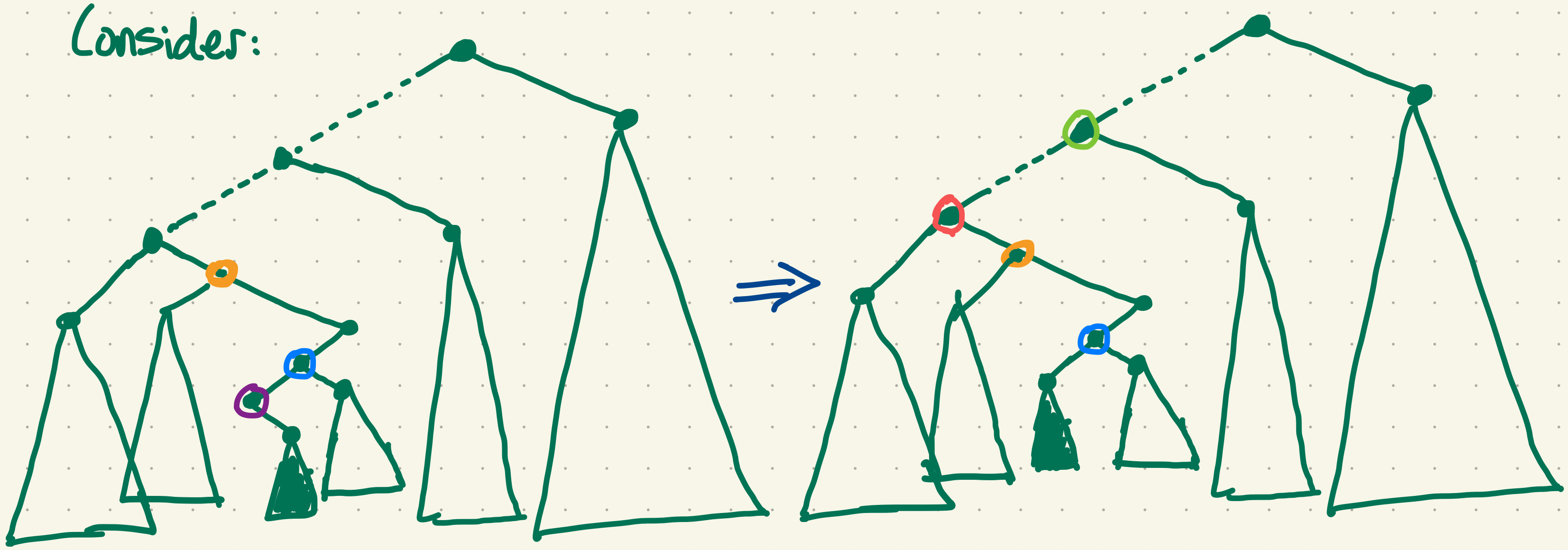
3. The deleted key is at a node with 2 children:



Fact: After doing a BST removal in an AVL tree, there is at most 1 unbalanced node, and it is on the path from the parent of the deleted node to the root.

(If the deleted node was the root, the "parent of the deleted node" does not exist – but also there can be no unbalanced node.)

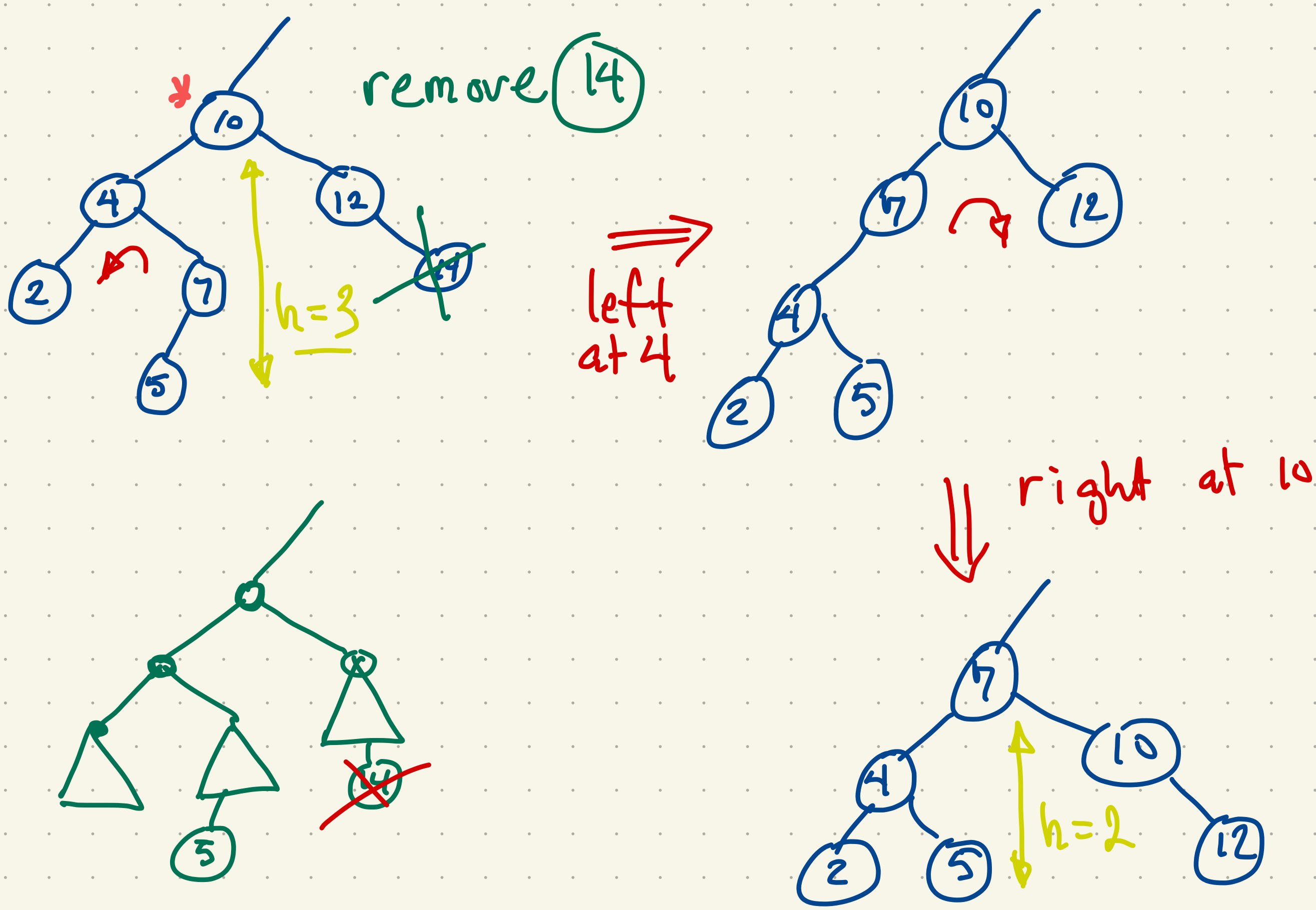
Consider:



- Node with key to be deleted
- Deleted node
- Parent of deleted node.
- Unbalanced node.
- This node, for example, cannot be unbalanced.

An AVL tree removal that illustrates:

1. Need to re-balance after removal
2. Re-balancing node u may reduce the height of a subtree, resulting in an ancestor of u being unbalanced.



Rebalance (for deletion):

$w \leftarrow$ parent of deleted node, if it exists
for(each node $u = w \ldots root$ on path from $w \ldots root$){
- if u is unbalanced
  - Let T be the subtree rooted at u
  - rebalance T using suitable rotations*
  - if height of T did not get smaller, return

}

* either a single or double rotation, based on case analysis similar to that used for insertion.

Correctness of the algorithm involves two properties:

a) There is at most 1 unbalanced node after deletion

b) Rebalancing w may make an ancestor of w unbalanced.

Complexity of AVL tree operations

- Every AVL tree with n nodes has height $O(\log n)$.
- The worst case amount of work for main operations is:
  - search: $O(\log n)$
    - one traversal from root to leaf: $O(\log n)$
  - insert: $O(\log n)$
    - two traversals from root to leaf (down & back up): $(O(\log n))$
    - two rotations: $O(1)$
  - remove: $O(\log n)$
    - two traversals from root to leaf (down & back up): $(O(\log n))$
    - at most, two rotations at each node on that path: $O(1) \cdot O(\log n) = O(\log n)$.

$\Rightarrow$ All three major operations in $O(\log n)$ time.

End