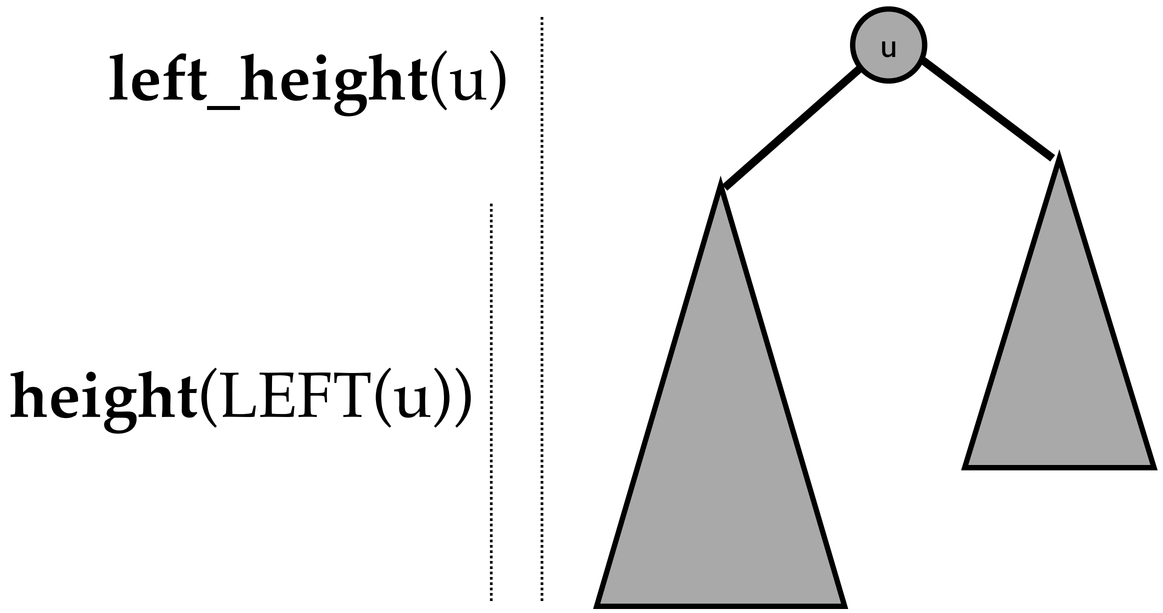
Balanced Trees

CMSC 420: Lecture 7

Balance

$$\textbf{left\_height}(u) = \begin{cases} 0 & \text{if LEFT(u) = NULL} \\ 1 + \textbf{height}(\text{LEFT(u)}) & \text{otherwise} \end{cases}$$



right_height defined analogously

balance(u) := **right_height**(u) - **left_height**(u)

Positive when right subtree is taller than left subtree

0 when the trees are the same height

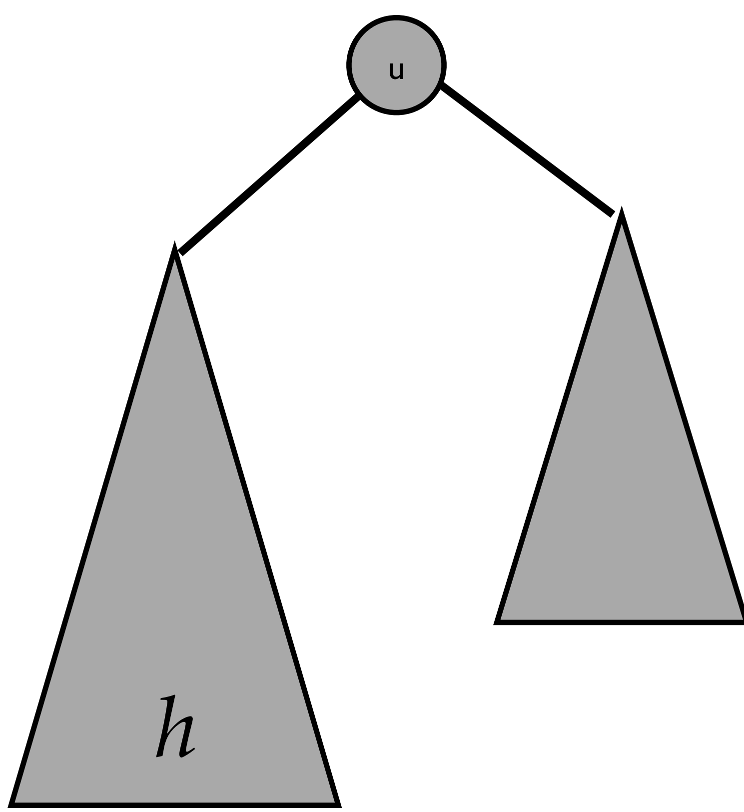
Negative when left subtree is taller than right subtree

AVL Trees

- A binary tree is an *AVL tree* if

$$\textbf{balance}(u) \in \{-1, 0, +1\} \text{ for every node } u$$

- I.e. the heights of LEFT(u) and RIGHT(u) are “about the same” for every node u.



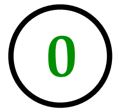
What are the possible heights for this subtree?

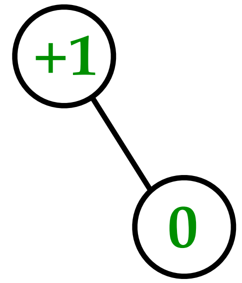
h, h-1, h+1

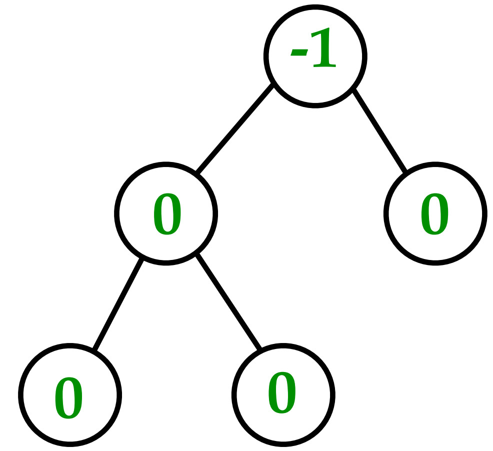
(Adelson-Velskii & Landis, 1962)

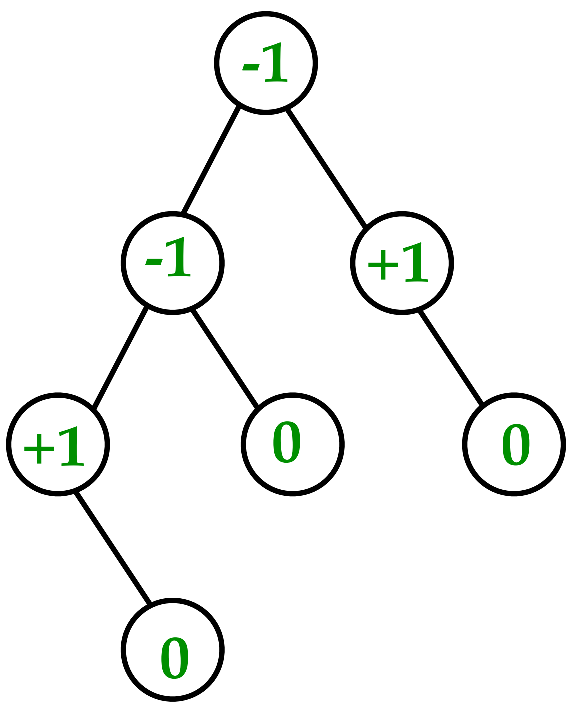
Examples

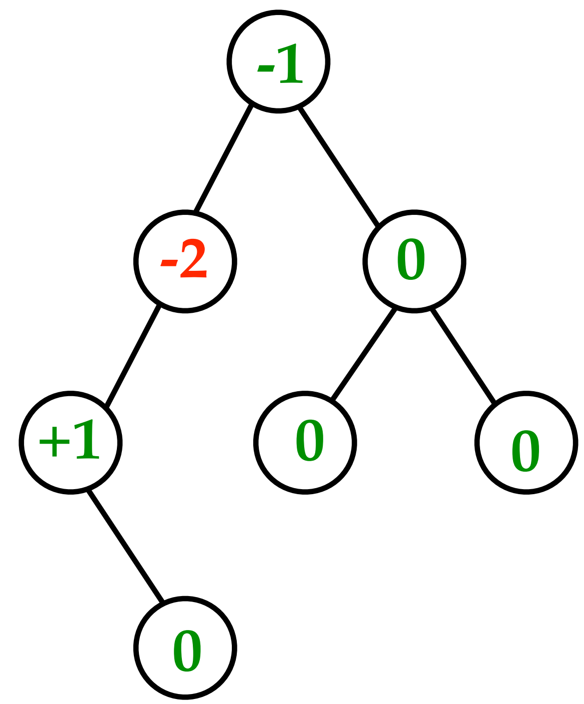
balance(u) := **right_height**(u) - **left_height**(u)











NOT an AVL tree

Properties & Notes

- All leaves have balance = 0
- AVL tree with n nodes has height $O(\log n)$.

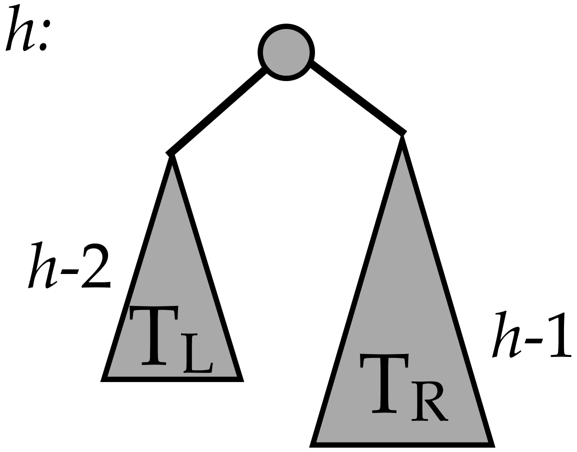
  $\Rightarrow$ *find* will run in $O(\log n)$ time if AVL has binary search tree property.
- *insert*, *delete* can be implemented in $O(\log n)$ time.

$\Rightarrow$ Good structure to implement *dictionary* or *sorted set* ADTs.

AVL Height is O(log n)

What's the smallest n we can fit into an AVL tree of a given height h?

Let T be a smallest AVL tree with height h:



One of T_L and T_R has height h-1.
Wlog, assume **height**(T_R) = h-1.

Then **height**(T_L) is either h-1 or h-2, but since T is smallest tree it must be h-2.

So, if $w(h)$ is number of nodes in smallest tree of height h, then

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

$$w(h) = F_{h+3} - 1$$

where F_i is the i^{th} Fibonacci number.

Fact. $F_i > \phi^i / \sqrt{5} - 1$.

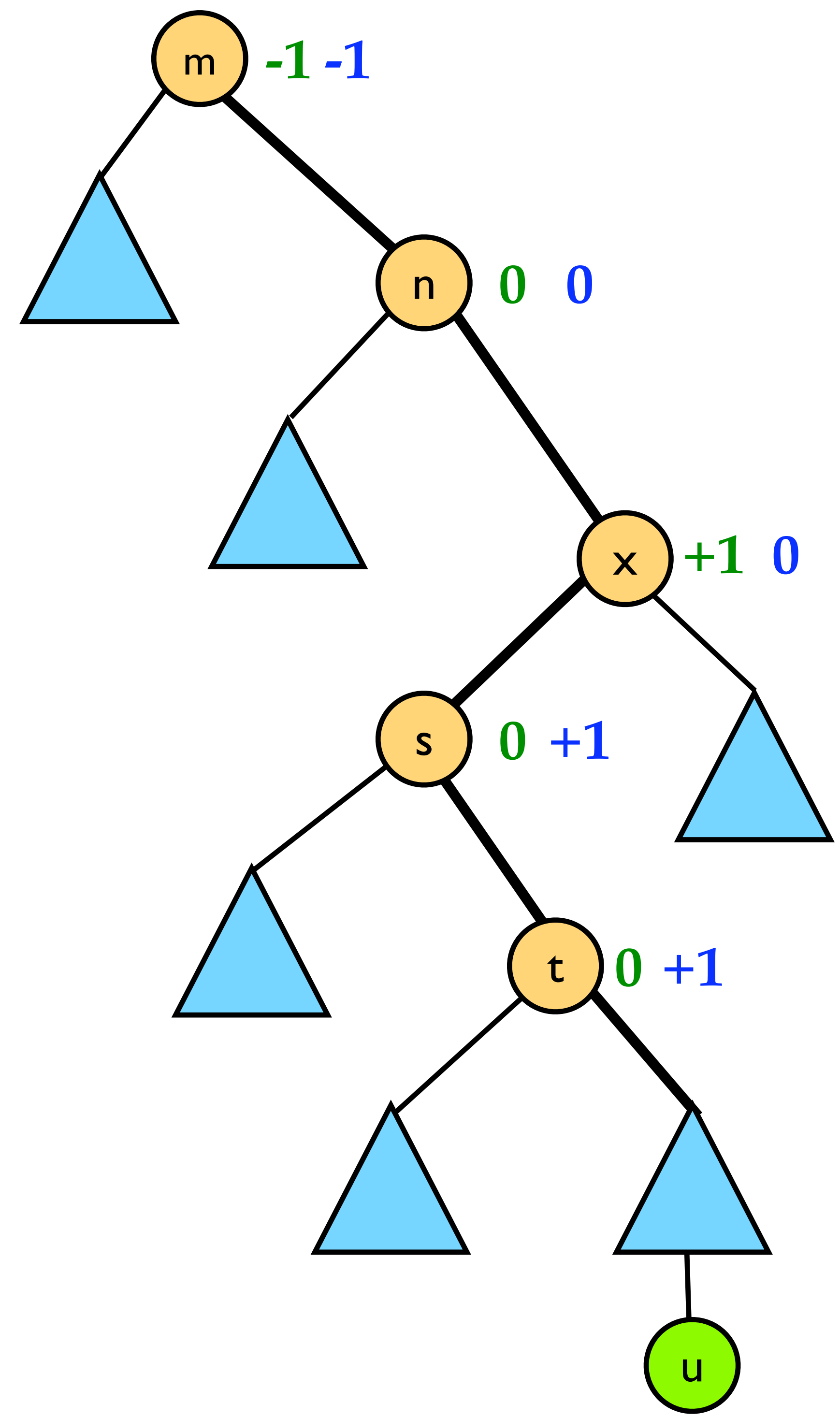
So, $n \geq w(h) > \phi^{h+3} / \sqrt{5} - 2$.

Solve for h: $h < \log(\sqrt{5}(n+2) / \phi^3)$

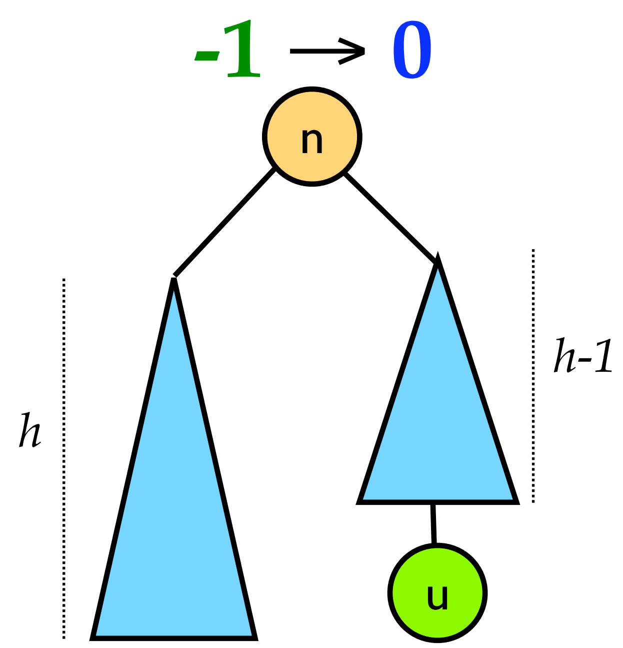
Thus: $h < O(\log n)$.

AVL Insert

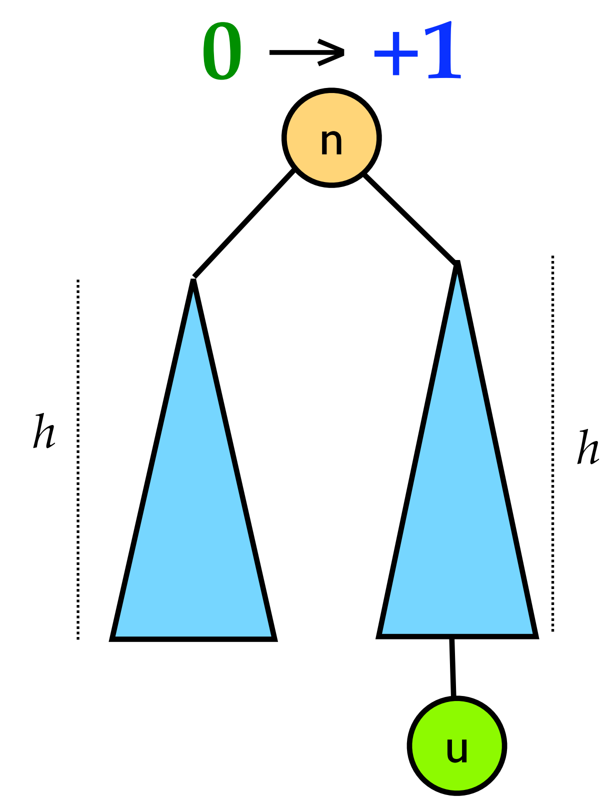
- First, do a standard BST insert: do a find and add node where you "fall off the tree."
- Walk insertion path back up to root, updating balances.
- If node was added to the left subtree, *decrement* balance by 1, otherwise *increment* balance by 1. Stop when node's height doesn't change.
- If a balance becomes +2 or -2, *fix it.*



The Easy Cases



Node was added to the shorter subtree

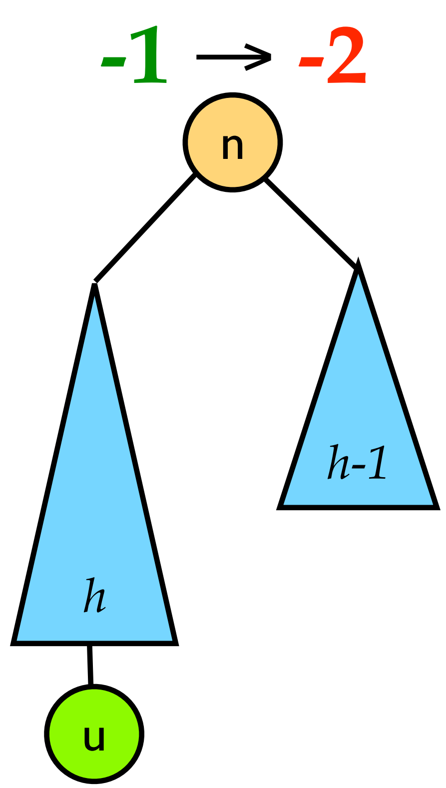


Subtrees were equal, now slightly unbalanced

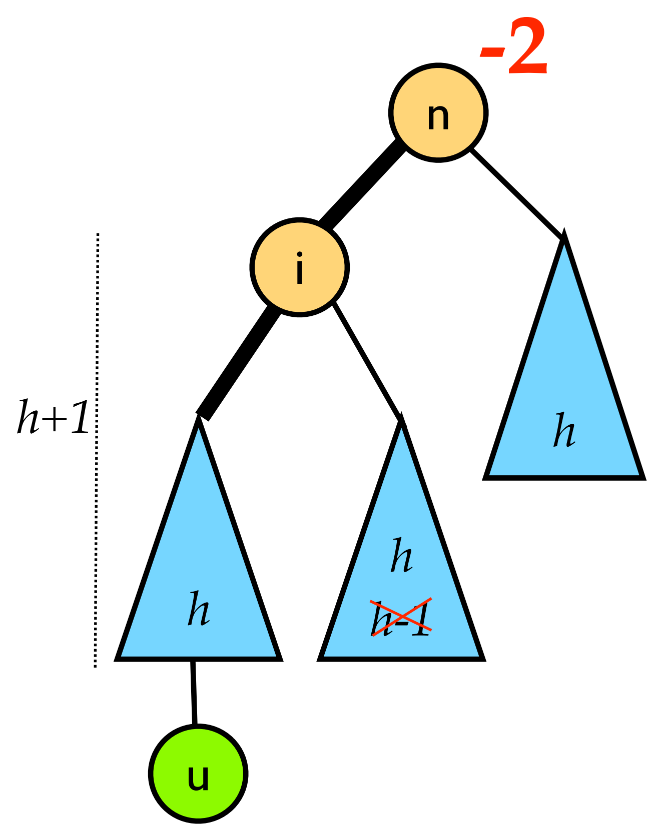
The symmetric cases (when left subtree was shorter, e.g.) are handled the same way.

The Somewhat Less Easy Cases

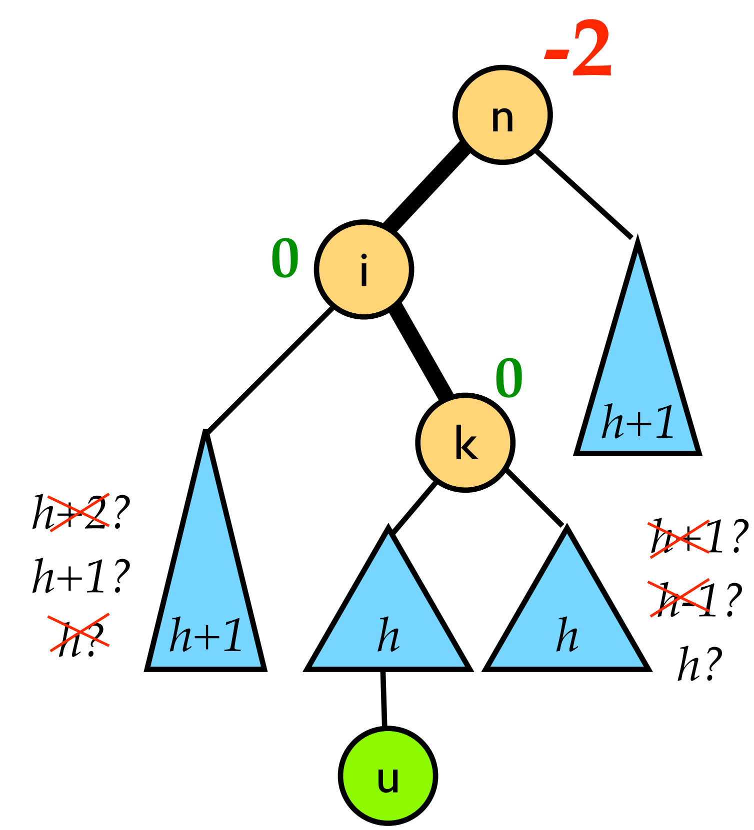
What to do? Two cases:



Suppose n is the lowest node that would become -2

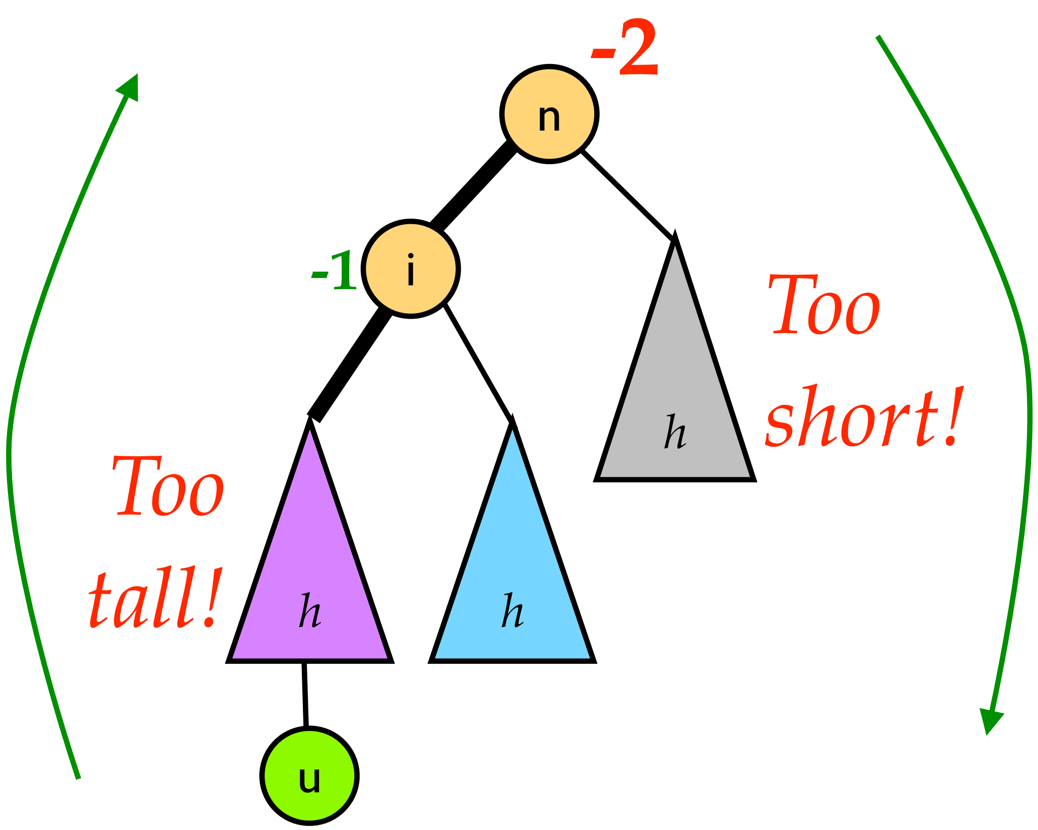


Left, Left

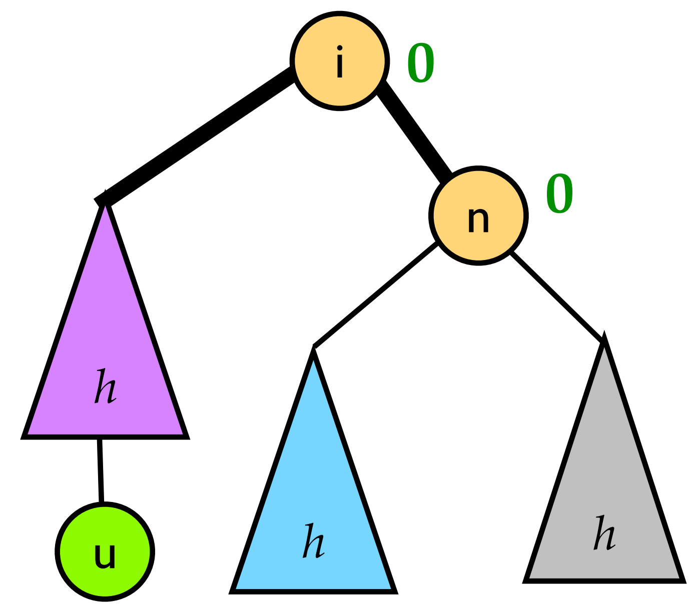


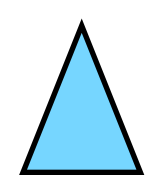
Left, Right

Left, Left Case

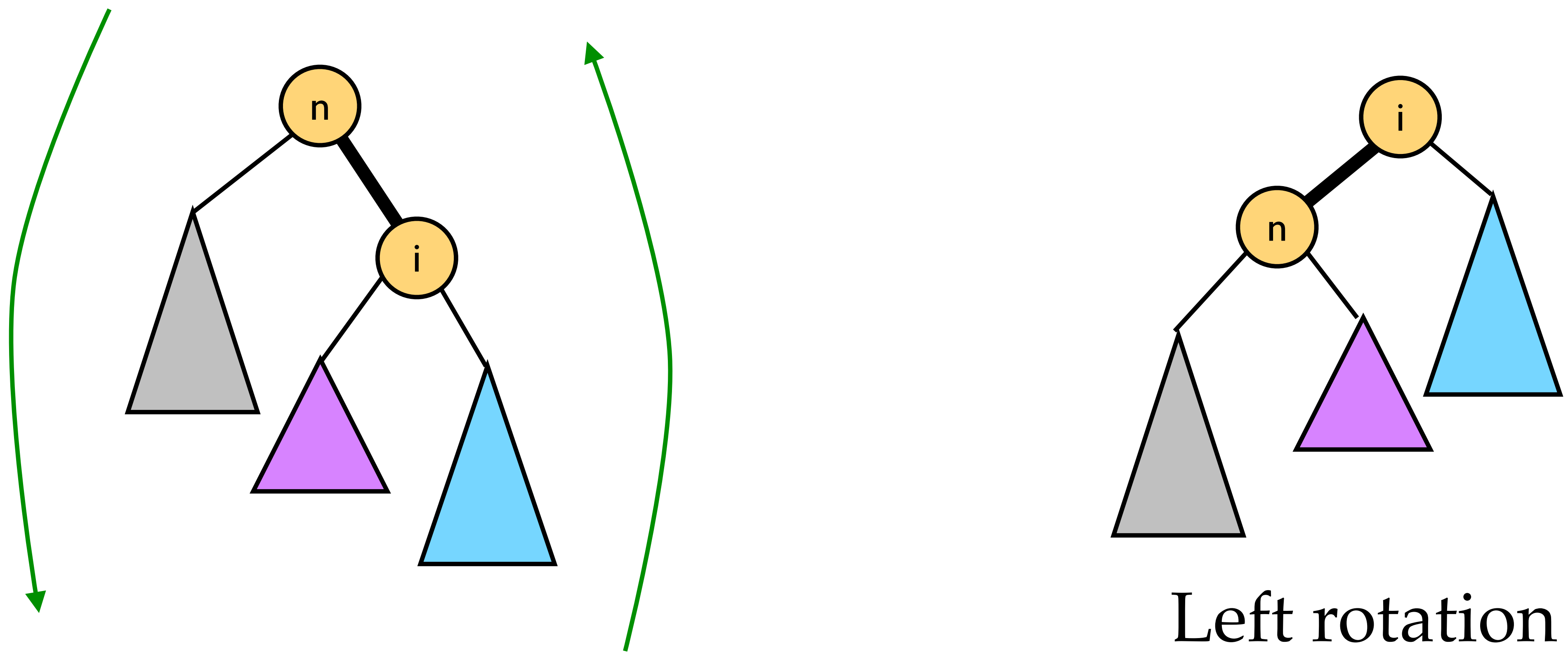


Right rotation
(aka clockwise rotation)



Why does  obey BST ordering?

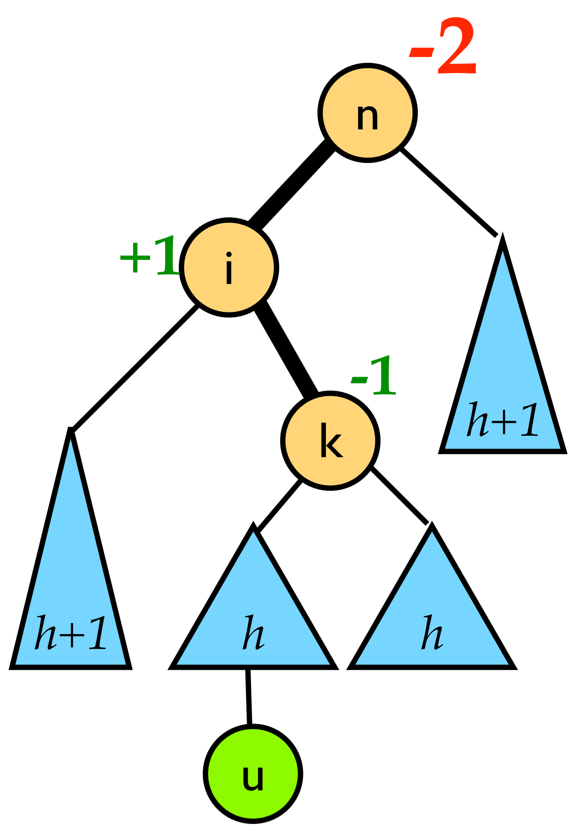
Symmetric Left Rotation:



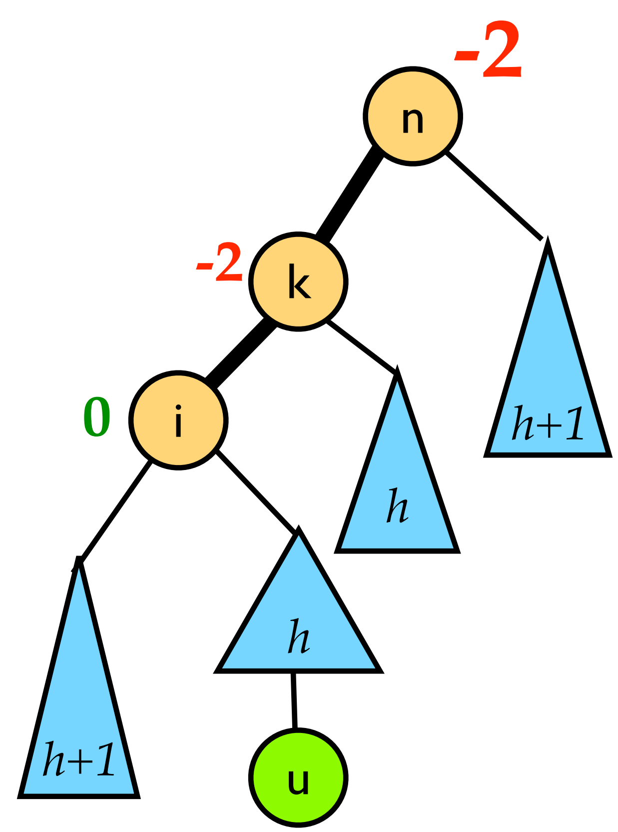
Left rotation
(aka counterclockwise rotation)

Only a constant # of pointers need to be updated for a rotation: O(1) time

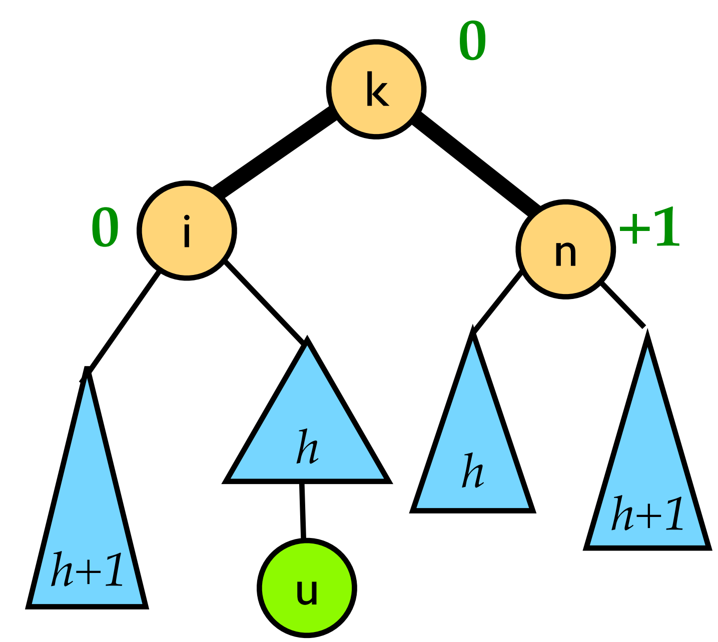
Left, Right Case:



Left, Right



(1) Left rotation at i

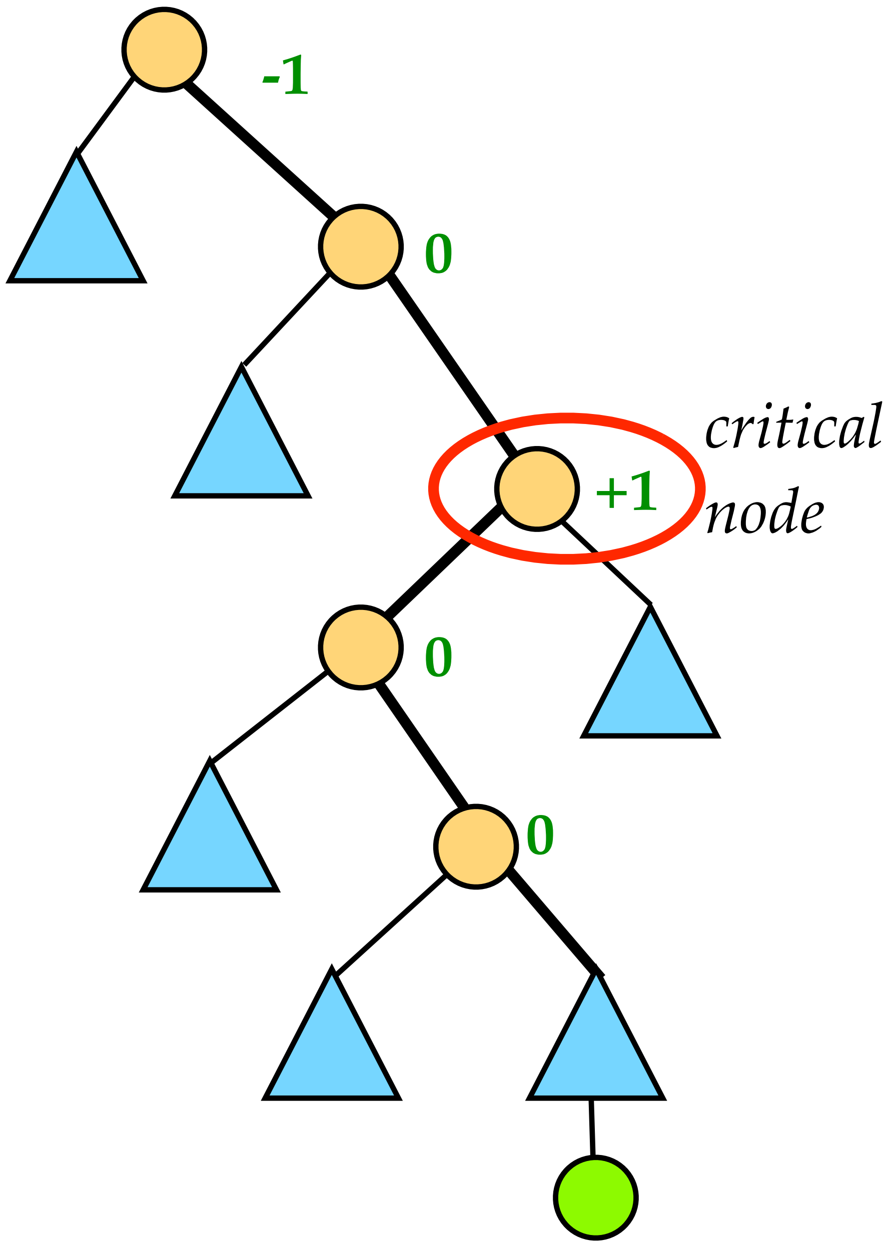


(2) Then right rotation at n

The Critical Node

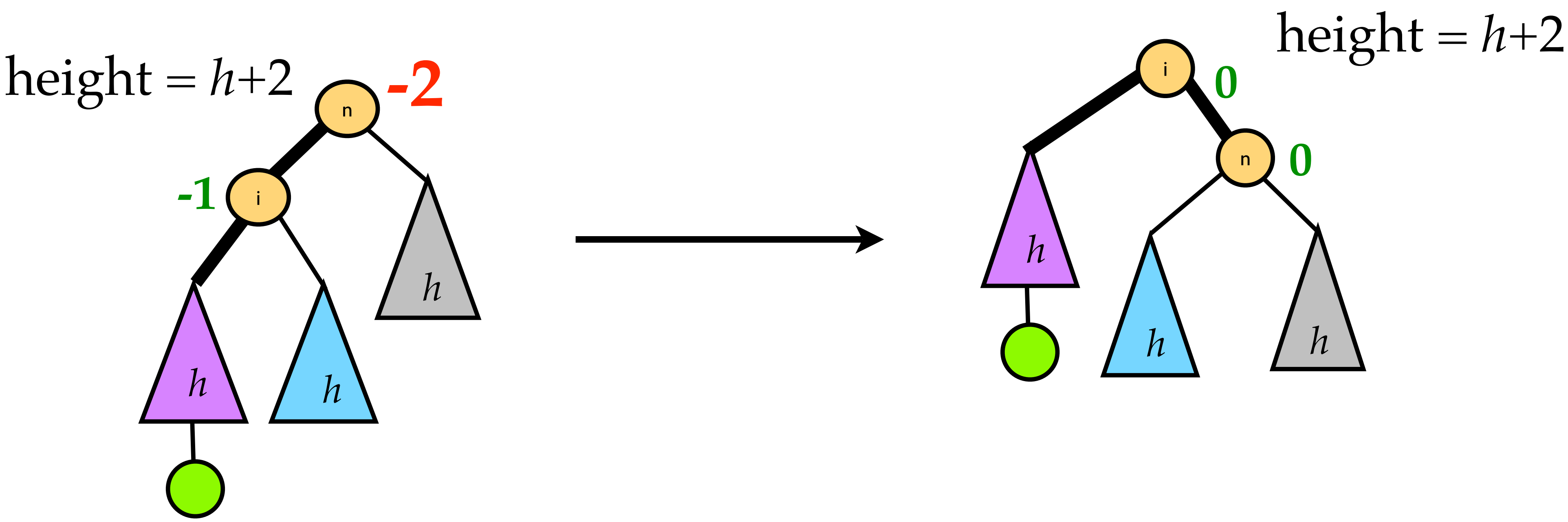
The *critical node* is the node on the insertion path closest to the leaves with balance ≠ 0

- Rotations leave subtree rooted at critical node balanced with ***unchanged height.***

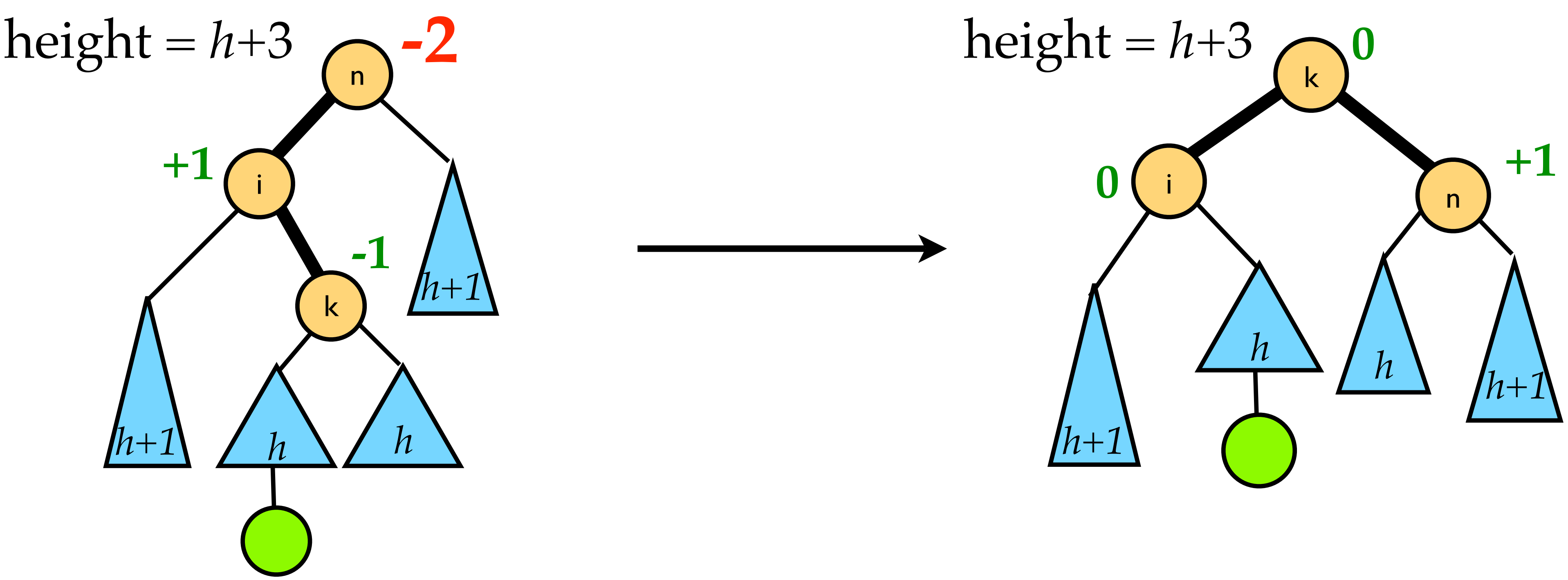


Rotations preserve height of critical subtree

Left, Left Case:



Left, Right Case:



Optimized Insert

- Because height of critical subtree doesn't change, it can't effect the balance of any nodes higher up in the tree.

- We can stop processing once we process the critical node.

- Therefore, only one rotation will occur.

- Optimization:
  - on first pass down the tree to insert a node, remember the critical node (last node with non-zero balance)
  - Then, to adjust balances, start at critical node and rewalk the path down to inserted node.

AVL Trees

- **Nice Features:**
  - Worst case O(log n) performance guarantee
  - Fairly simple to implement

- **Problem though:**
  - Have to maintain extra balance factor storage at each node.

- **Splay trees (Sleator & Tarjan, 1985)**
  - remove extra storage requirement,
  - even simpler to implement,
  - heuristically move frequently accessed items up in tree
  - amortized O(log n) performance
  - worst case single operation is $\Omega(n)$

Splay Trees

splay(T, k): if $k \in$ T, then move k to the root. Otherwise, move either the inorder successor or predecessor of k to the root.

Without knowing how *splay* is implemented, we can implement our usual operations as follows:

- *find*(T, k): *splay*(T, k). If *root*(T) = k, return k, otherwise return **not found**.

- *insert*(T, k): *splay*(T, k). If *root*(T) = k, return **duplicate!**; otherwise, make k the root and add children as in figure.

- *concat*(T_1, T_2): Assumes all keys in T_1 are < all keys in T_2. *Splay*(T_1, ∞). Now root T_1 contains the largest item, and has no right child. Make T_2 right child of T_1.

- *delete*(T, k): *splay*(T, k). If root r contains k, *concat*(LEFT(r), RIGHT(r)).