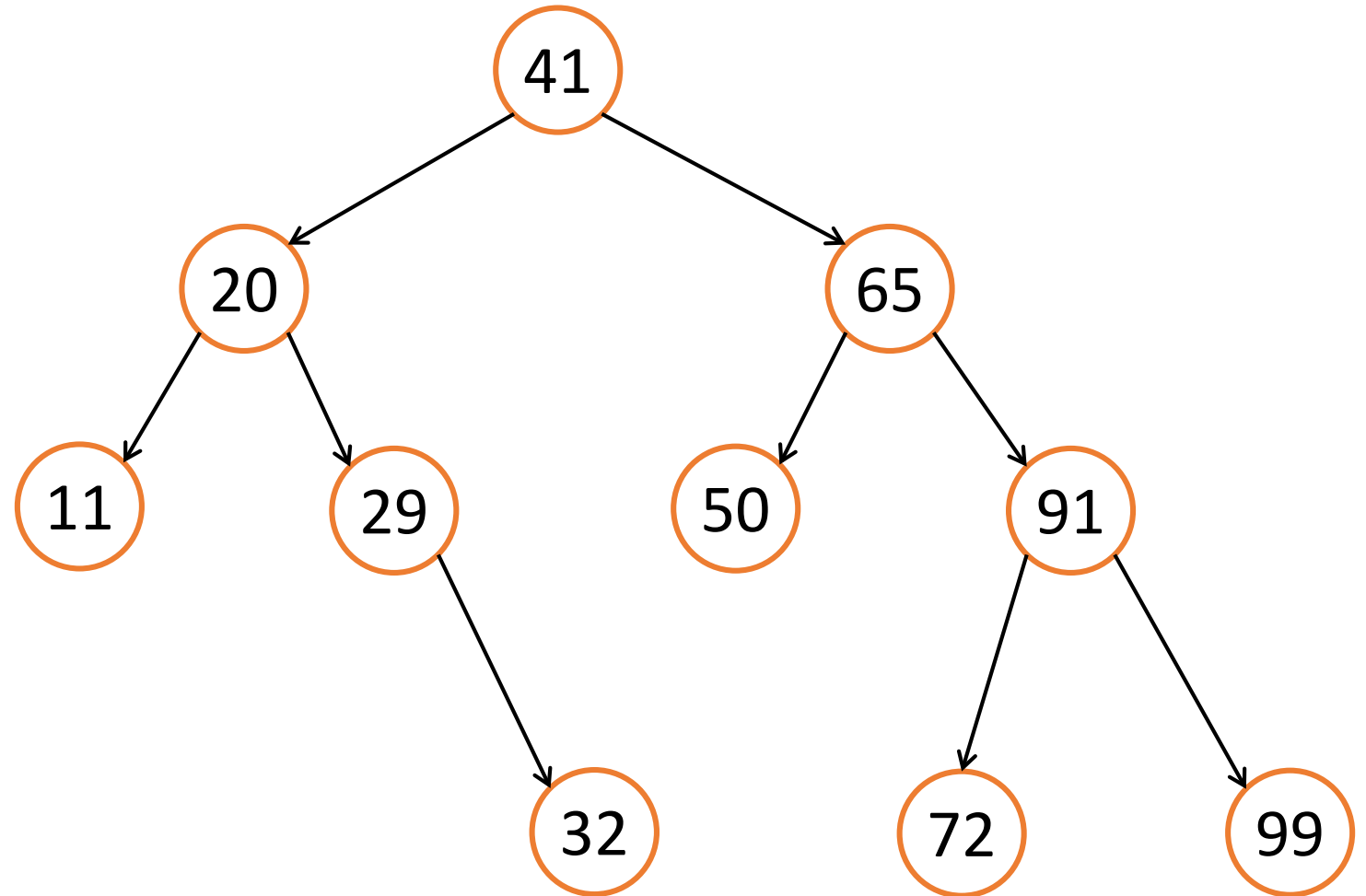


BST

(Deletion of a node)

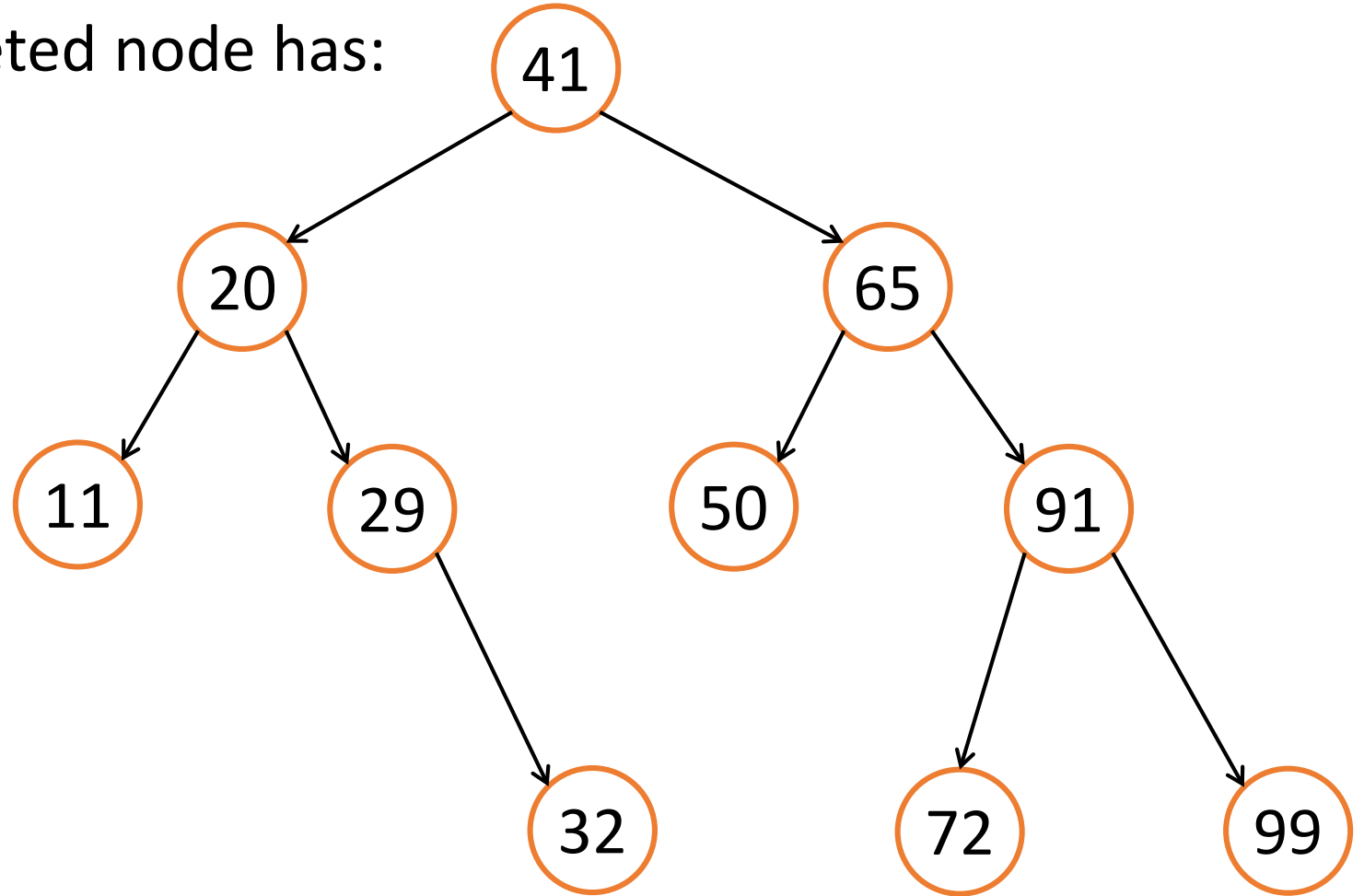
Deletion of a node



Three cases of Deleting a Node

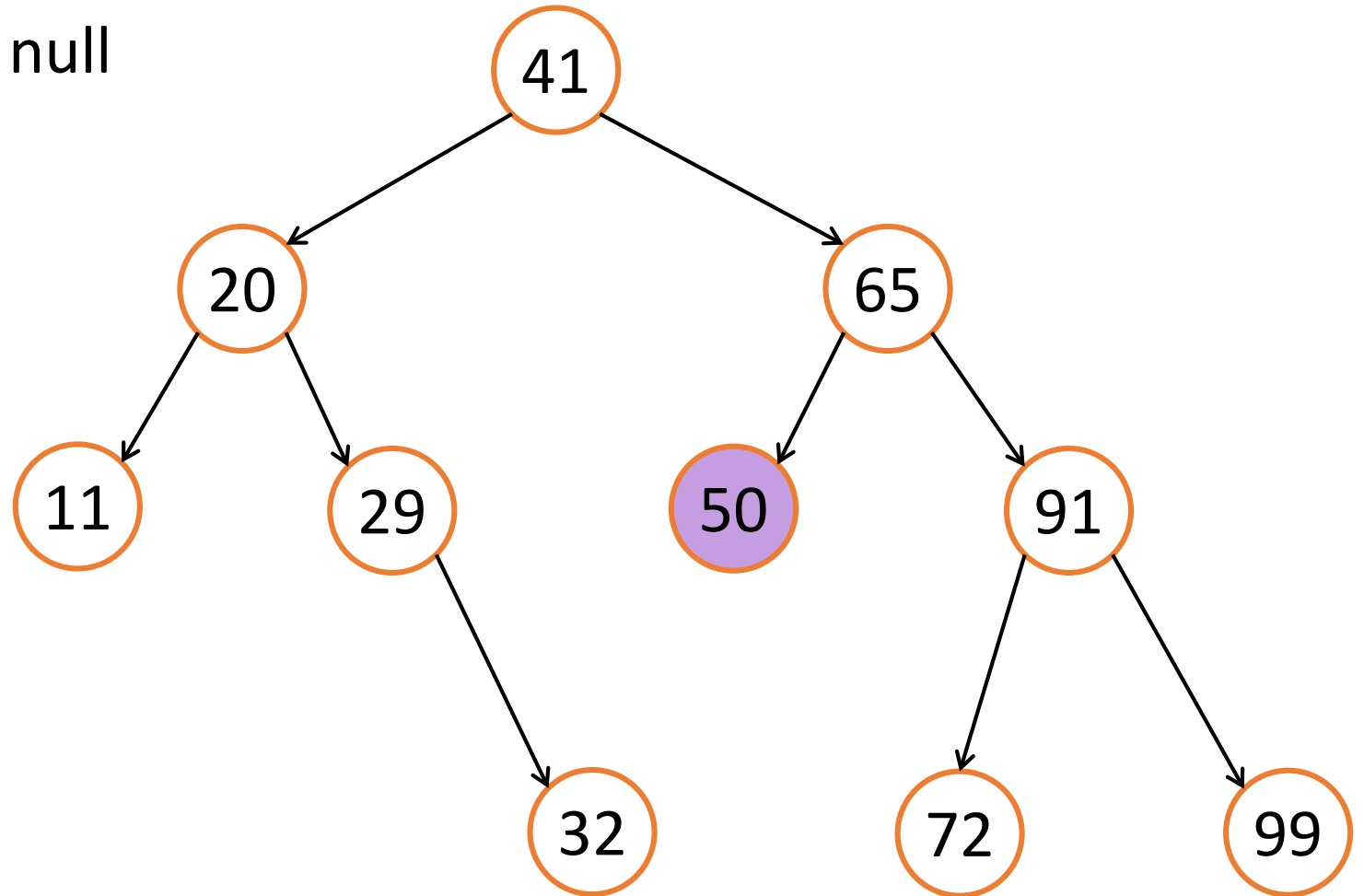
- If the going-to-be-deleted node has:

- No children
- 1 child
- 2 children



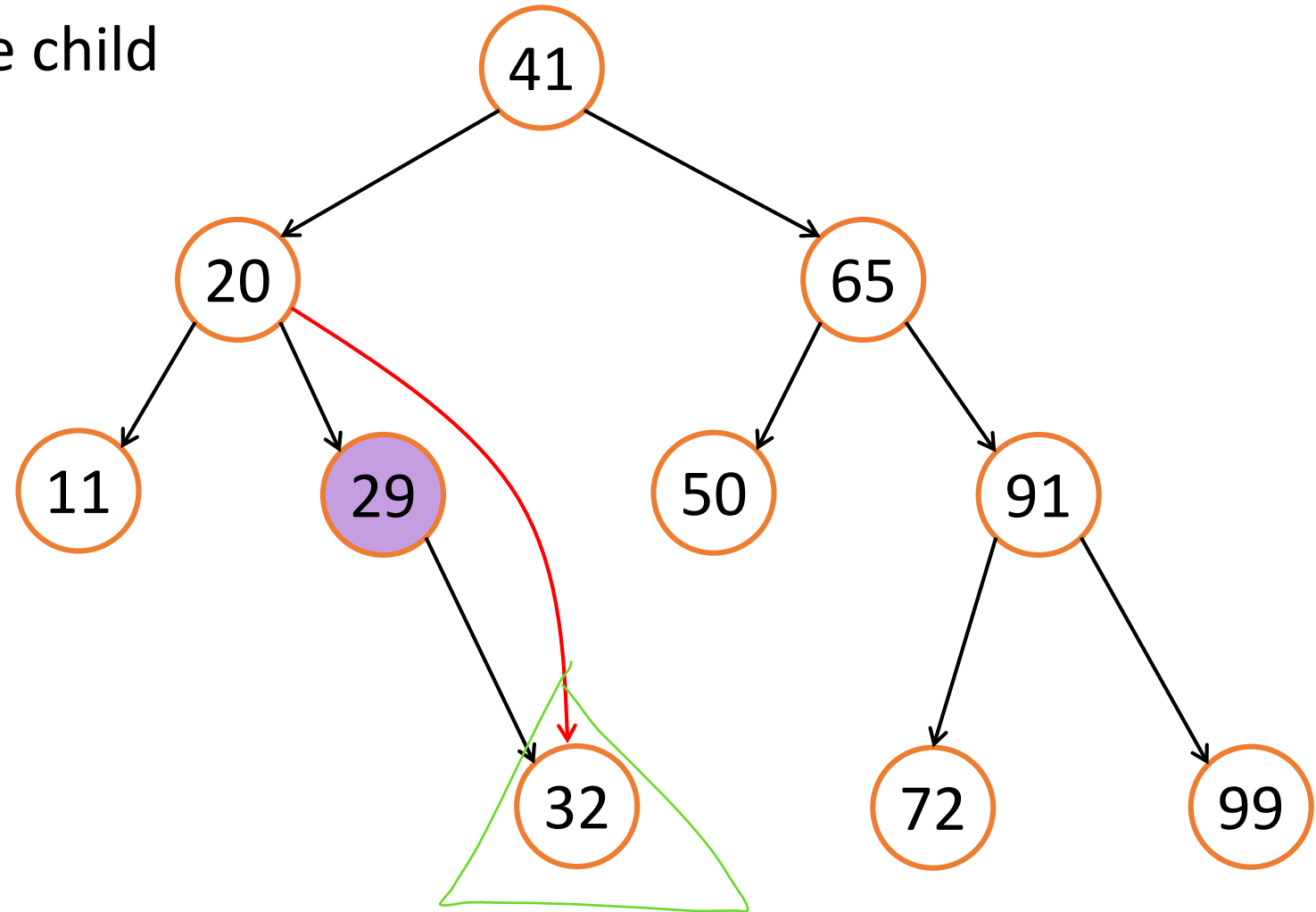
Case 1: No Children (delete(50))

- Set your parent's child to null
- Delete yourself



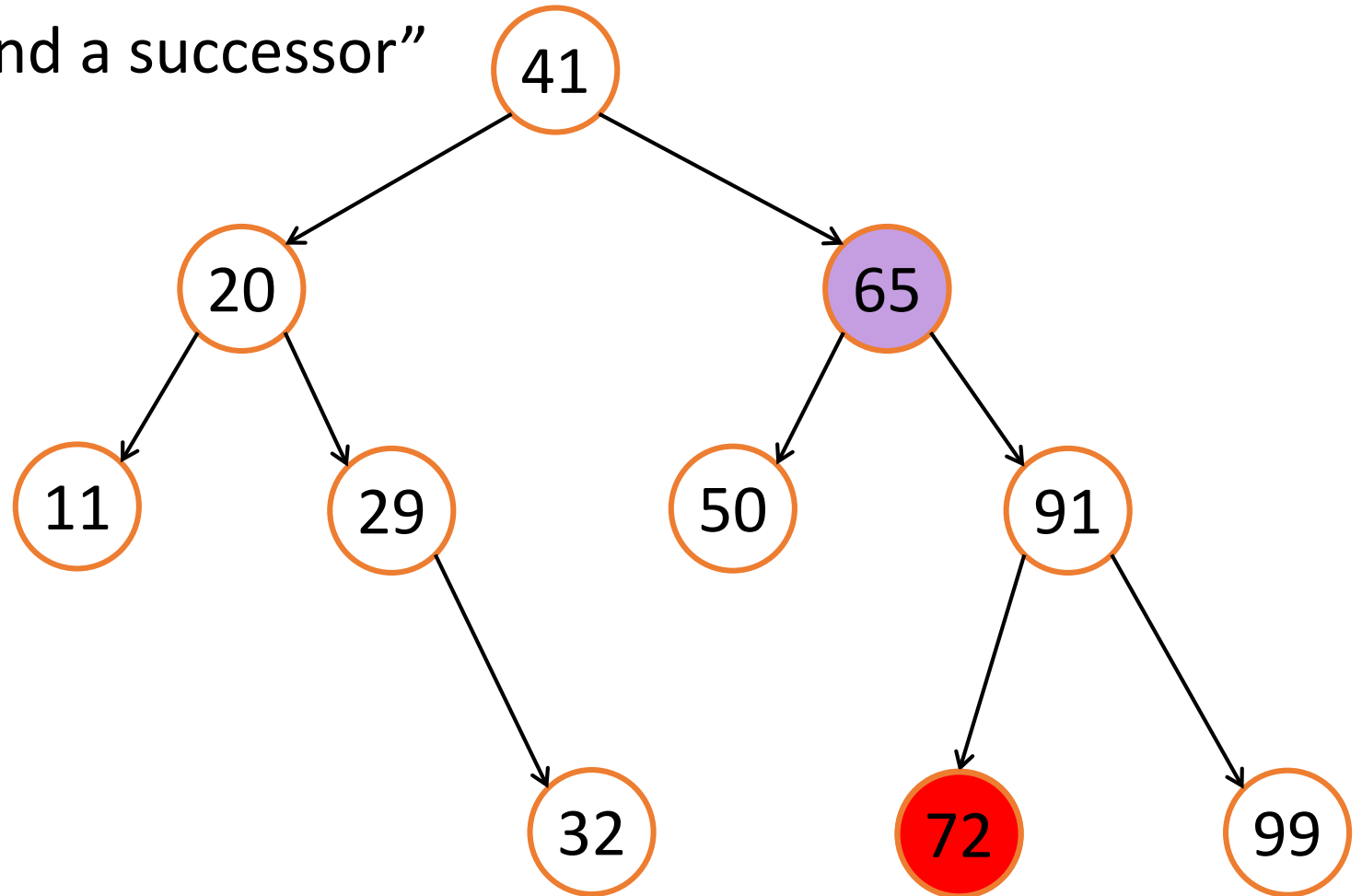
Case 2: One Child (delete(29))

- Link up the parent and the child
- Delete yourself



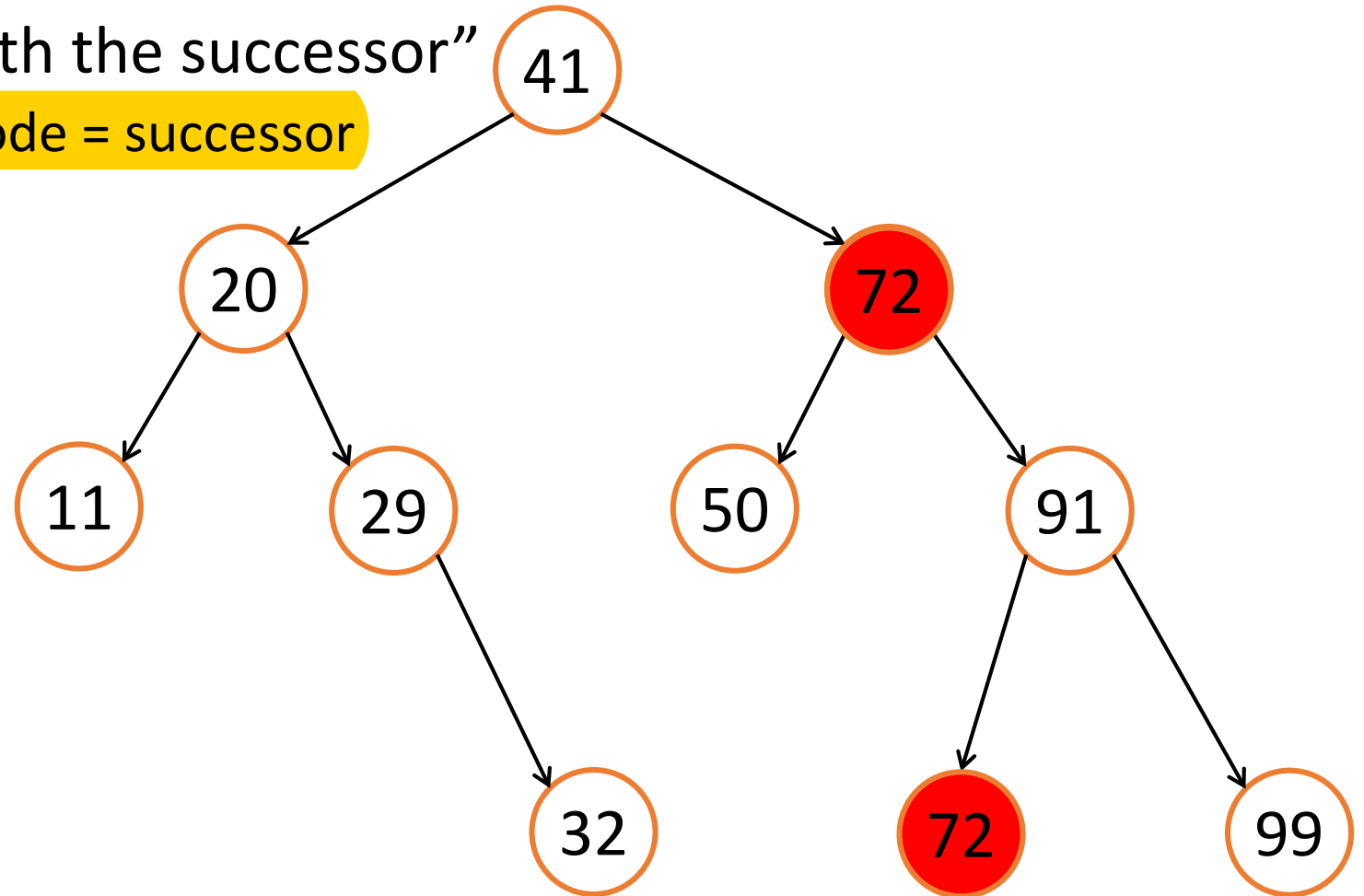
Case 3: 2 Children (delete(65))

- “Before a king is gone, find a successor”



Case 3: 2 Children (delete(65))

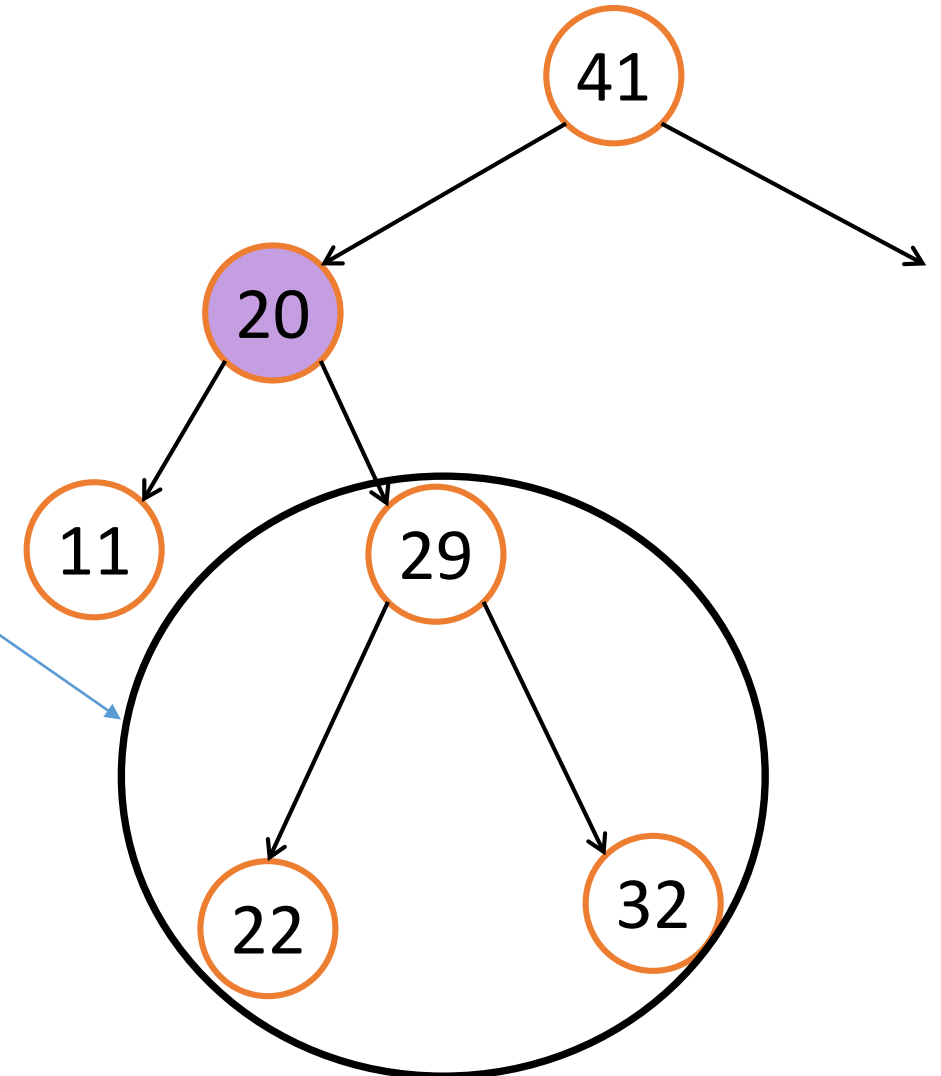
- “And replace the king with the successor”
 - 🗑️ “going-to-be-deleted” node = successor
 - delete successor



Wait! what if the
successor has two
children?

Recap: Successor Queries

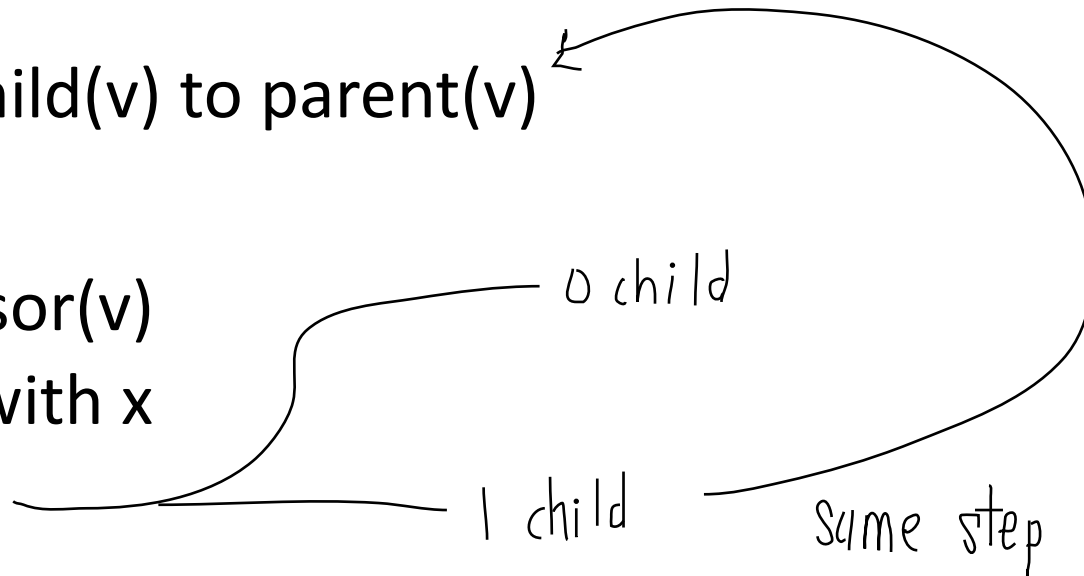
- Under the assumption that
 - The node is in the tree
 - The node has two children
- Which one is the successor?
- The minimum of its right subtree!



Will this successor
has two children?

Three Cases of Deleting a Node in BST

- No children:
 - remove v
- 1 child:
 - remove v
 - connect $\text{child}(v)$ to $\text{parent}(v)$
- 2 children
 - $x = \text{successor}(v)$
 - replace v with x
 - remove x



AVL Tree Balancing

“The importance of being balanced”

Binary Search Tree (BST)

- Modifying Operations
 - insert: $O(h)$
 - delete: $O(h)$
- Query Operations:
 - search: $O(h)$
 - predecessor, successor: $O(h)$
 - findMax, findMin: $O(h)$
 - in-order-traversal: $O(n)$

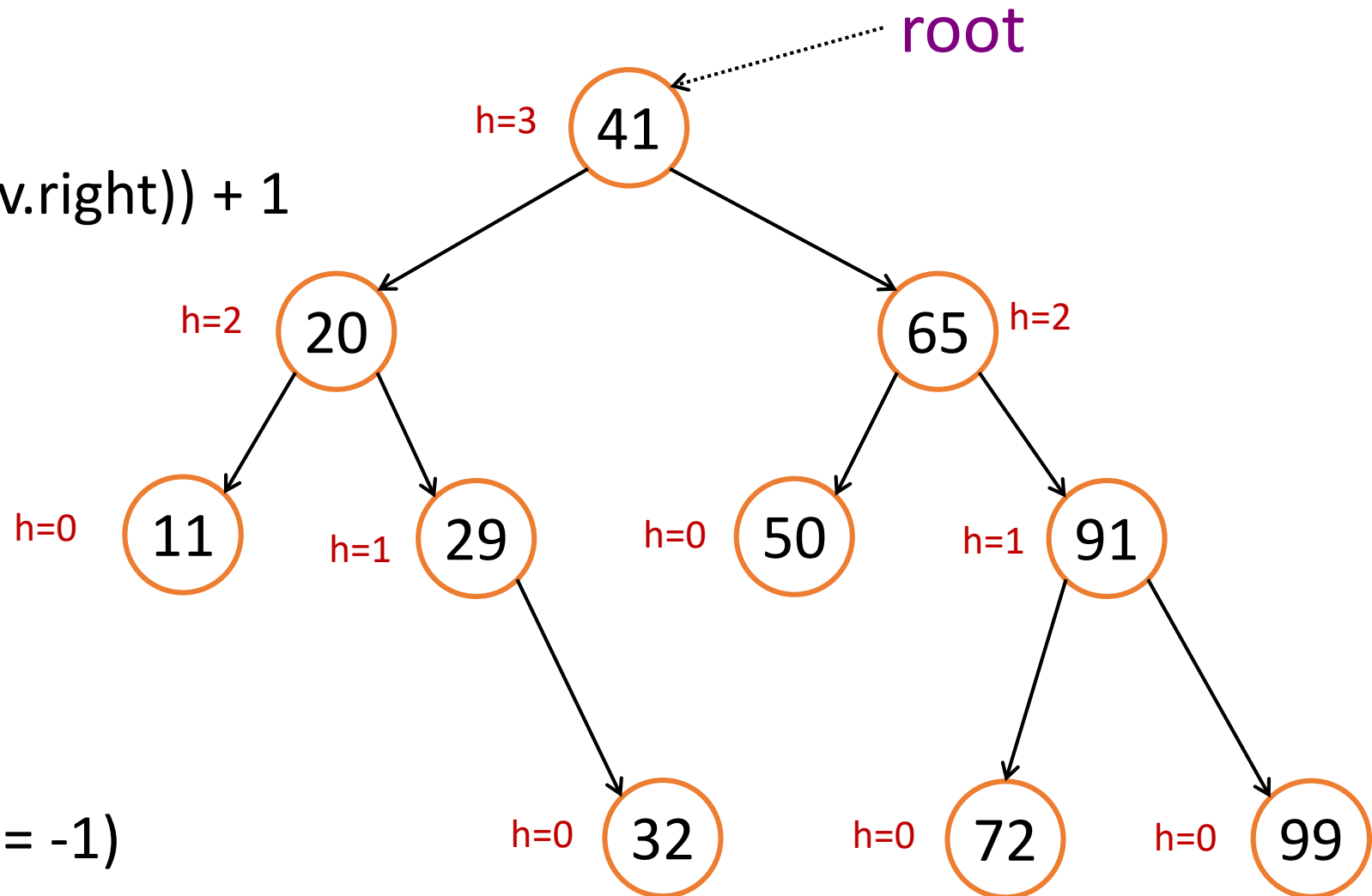
Plan

- On the importance of being balanced
 - Height-balanced binary search trees
 - AVL trees



Recap: Tree Heights

- For a tree node v
- $h(v) = 0$ if v is a leaf
- $h(v) = \max(h(v.\text{left}), h(v.\text{right})) + 1$

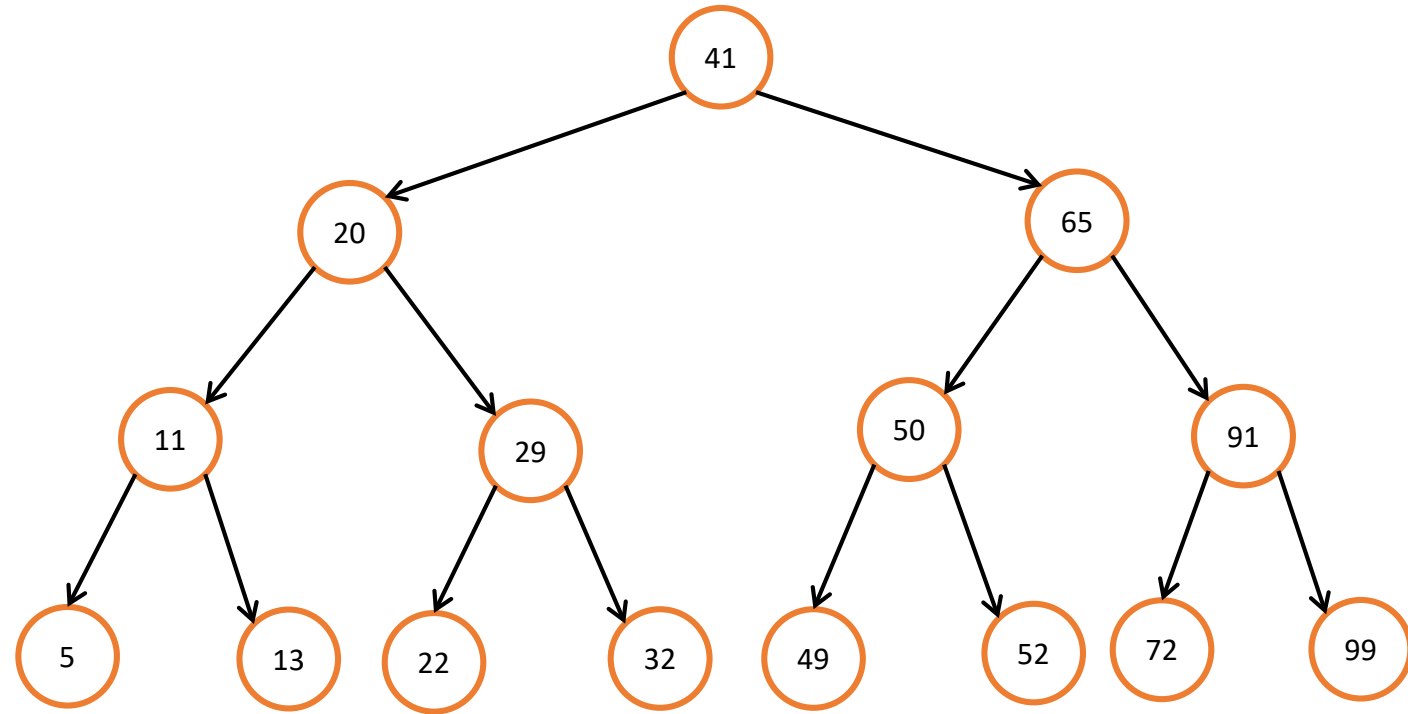
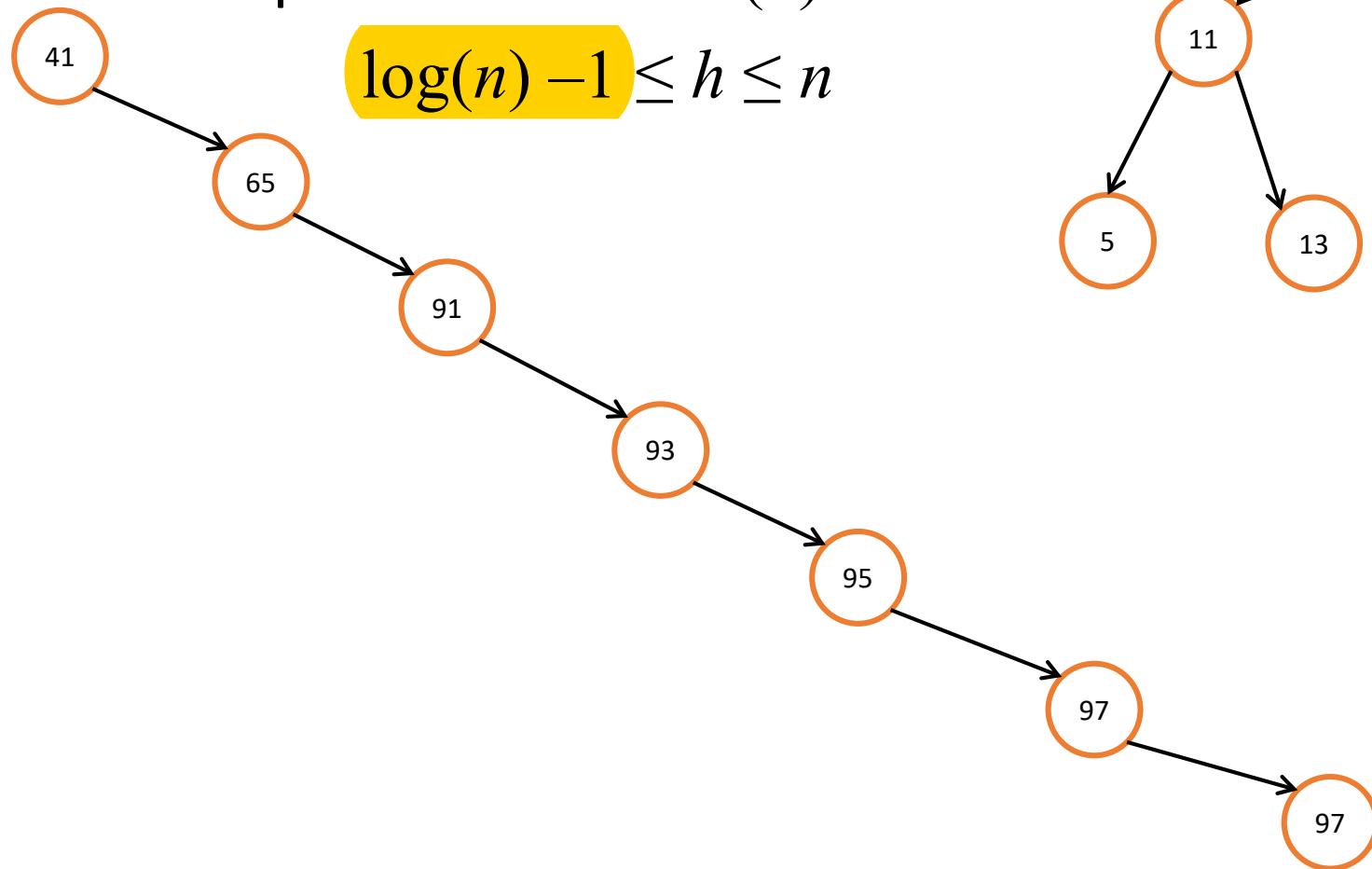


- (For simplicity: $h(\text{null}) = -1$)

Tree Height

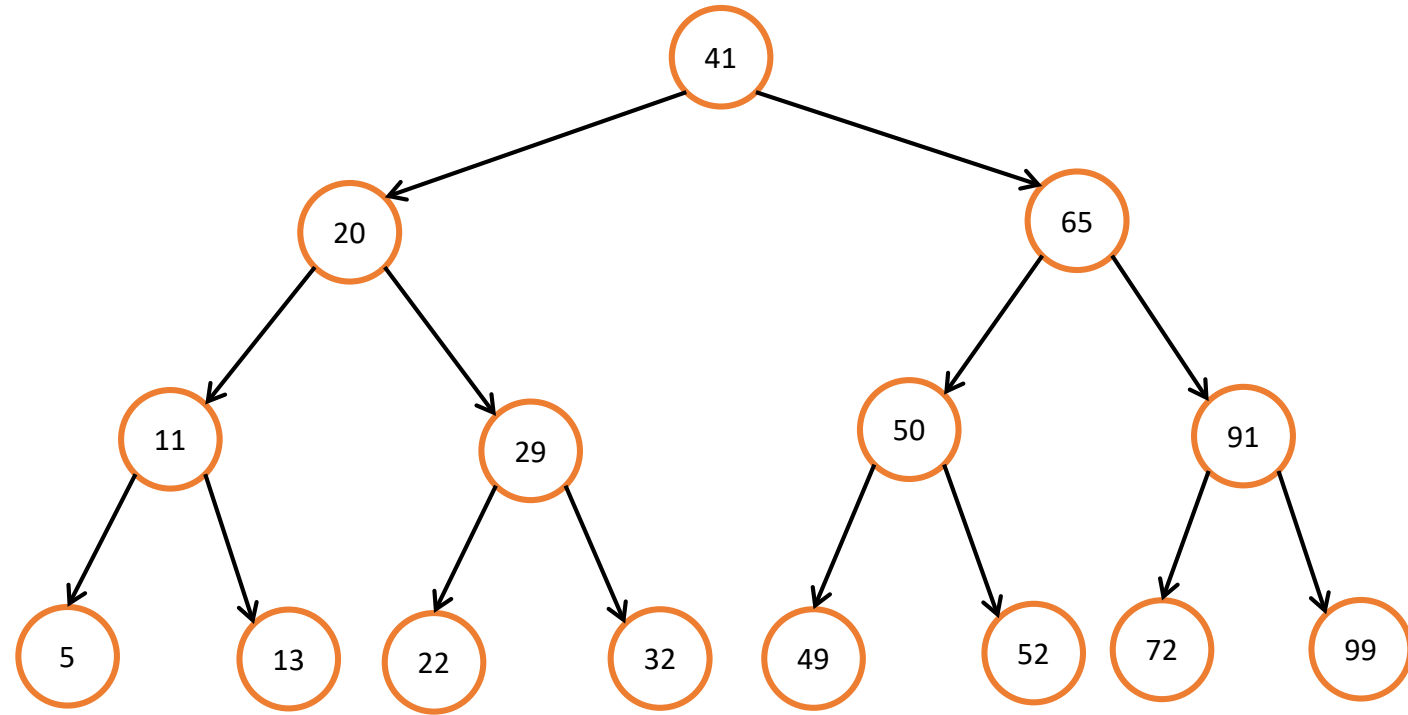
- Operations take $O(h)$ time:

$$\log(n) - 1 \leq h \leq n$$



Tree Height

- Operations take $O(h)$ time:
 $\log(n) - 1 \leq h \leq n$
- Operations take $O(\log n)$ time if the tree is balanced



Many different flavors of balanced search trees

- AVL trees (Adelson-Velsii & Landis, 1962)
- B-trees / 2-3-4 trees (Bayer & McCreight, 1972)
- BB[α] trees (Nievergelt & Reingold 1973)
- Red-black trees (see CLRS 13)
- Splay trees (Sleator and Tarjan 1985)
- Treaps (Seidel and Aragon 1996)
- Skip Lists (Pugh 1989)
- Scapegoat Trees (Anderson 1989)

Step 1: Augment the nodes

- In every node v , store height:

$v.\text{height} = h(v)$

- When we do insert/delete, we update the heights

```
insert(x)
```

```
    if (x < key)
```

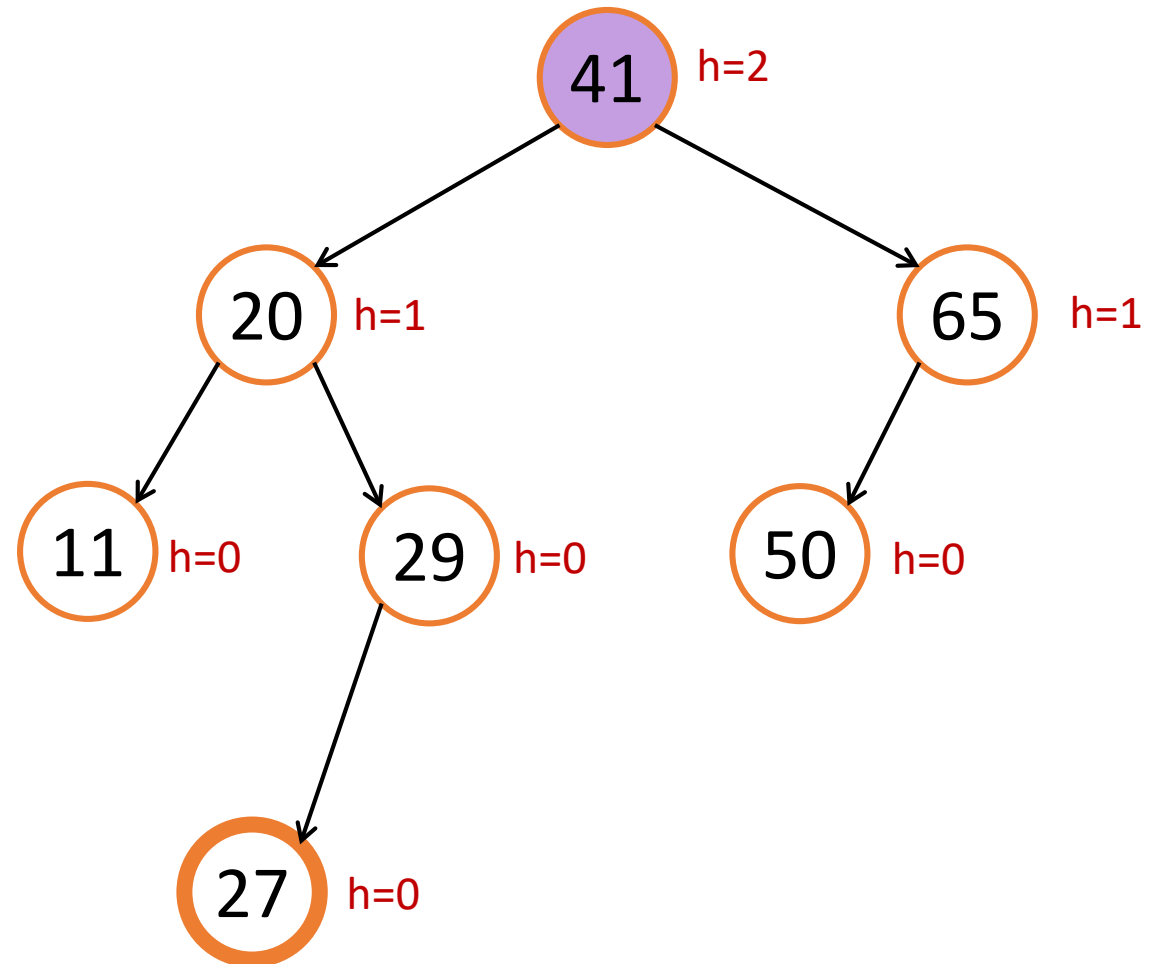
```
        left.insert(x)
```

```
    else right.insert(x)
```

```
    height = max(left.height, right.height)+1
```

Example: Insert(27)

- Which node's height is not correct anymore after insertion?



Step 1: Augment the nodes

- In every node v , store height:

$v.\text{height} = h(v)$

- When we do insert/delete, we update the heights

```
insert(x)
```

```
    if (x < key)
```

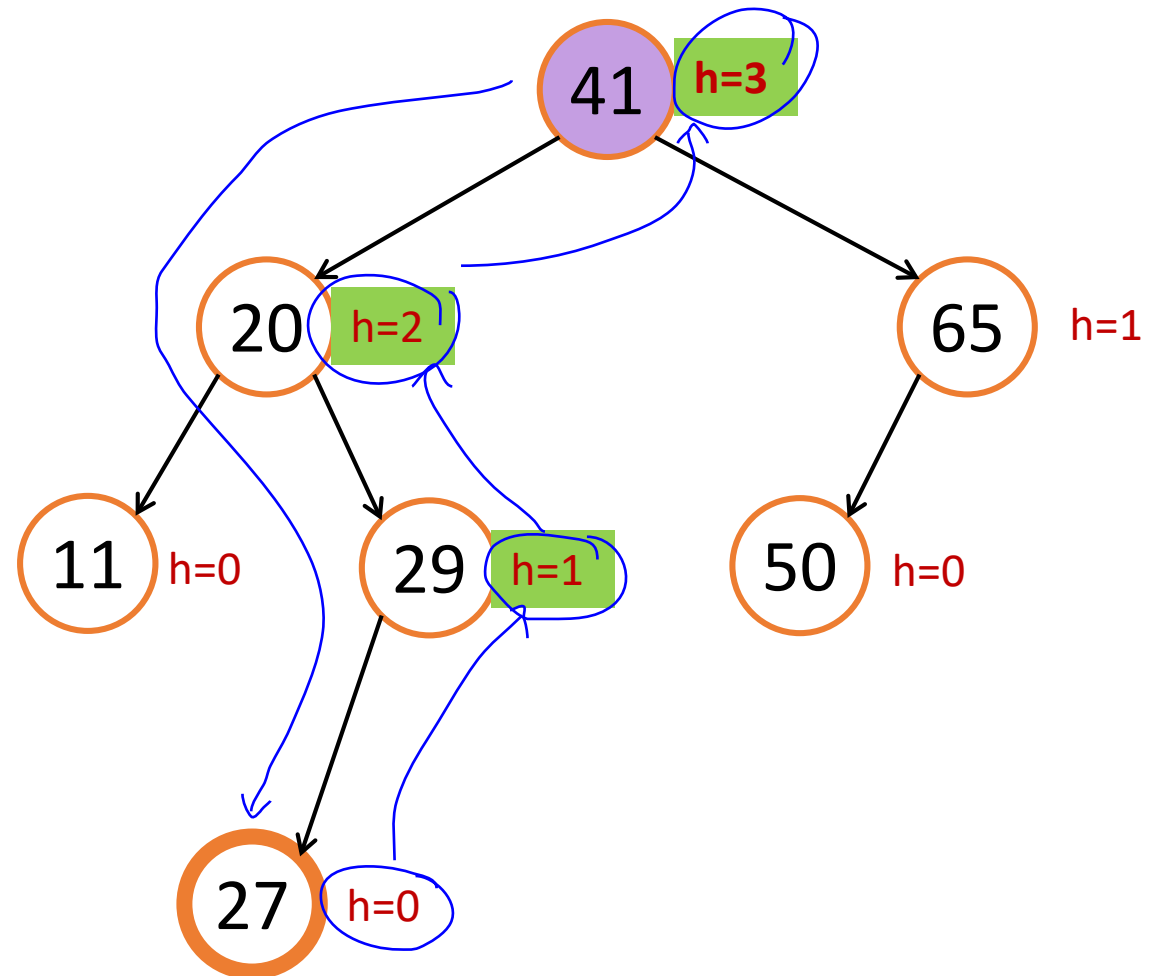
```
        left.insert(x) ||
```

```
    else right.insert(x) ||
```

```
    height = max(left.height, right.height) + 1
```

Example: Insert(27)

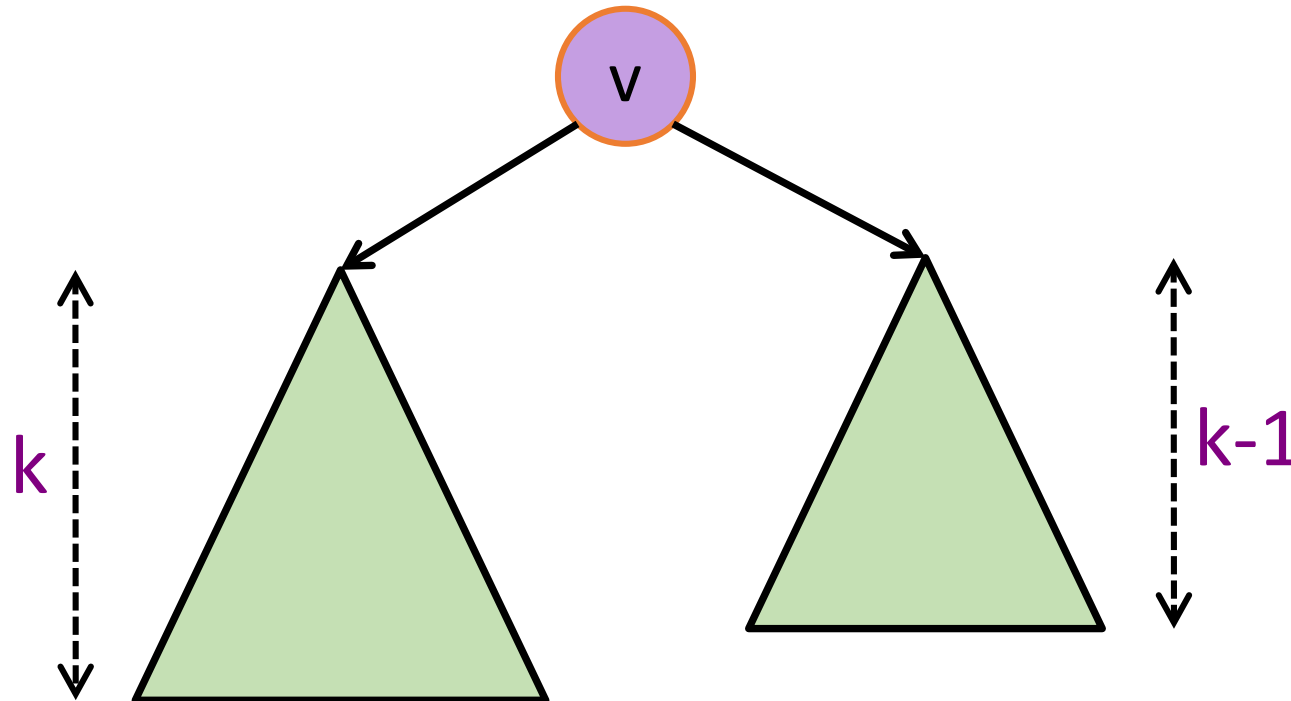
- Updating after each recursion



Step 2: Define Invariant

- AVL Trees [Adelson-Velskii & Landis 1962]
- A node v is **height-balanced** if:

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



Step 2: Define Invariant

- AVL Trees [Adelson-Velskii & Landis 1962]

- A node v is **height-balanced** if:

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

- A binary search tree is **height-balanced** if every node in the tree is height-balanced.



So
What

Height-Balanced Trees

- A height-balanced binary search tree with n nodes has at most height:

$$h < 2 \log n$$

- But I don't know how to prove this
- Instead, I tried to prove...

That makes our
tree operations
 $O(h) = O(\log n)$

Height-Balanced Trees

- A height balanced binary search tree with n nodes has at most height:

$$h < 2 \log n$$

$$\Leftrightarrow h/2 < \log(n)$$

$$\Leftrightarrow 2^{h/2} < 2^{\log(n)}$$

$$\Leftrightarrow 2^{h/2} < n$$

\Leftrightarrow For a tree with height h , a height-balanced tree contains at least $n > 2^{h/2}$ nodes

Proof of Height Balanced Tree $h = O(\log n)$

- Claim:

- ~~• A height-balanced tree is balanced, i.e., has height $h = O(\log n)$.~~
- For a tree with height h , a height-balanced tree contains at least $n > 2^{h/2}$ nodes

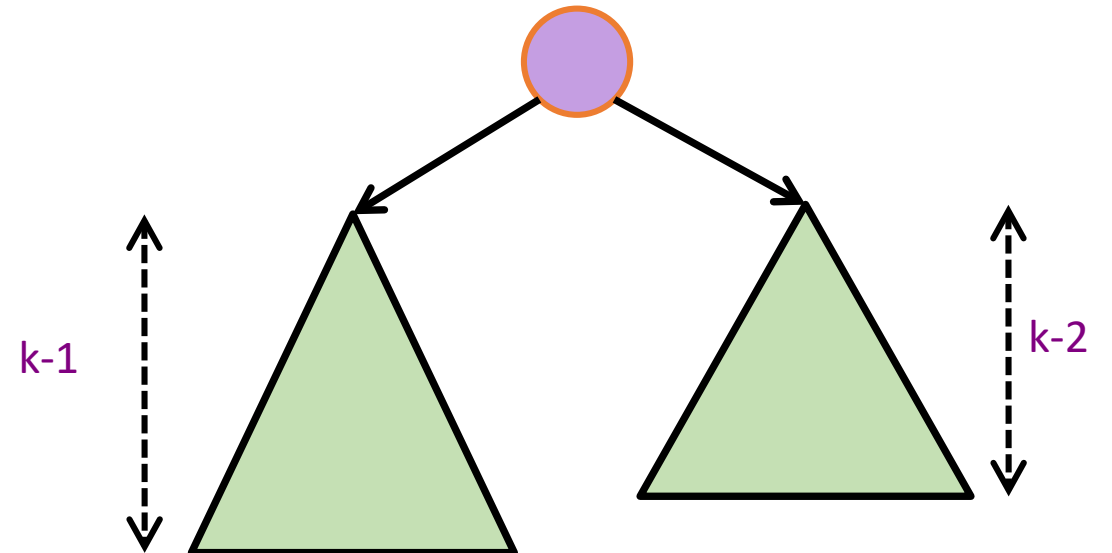
- Prove by Induction:

- Denote as n_h the minimal no. of nodes of a height-balanced tree of height h
- Assuming it is true for $h < k$.
 - Namely, $n_h > 2^{h/2}$ for $h < k$
- Base case:
 - For $h = 0$, $n_h = 1$, it is a perfectly balanced tree with one node

Proof of Height Balanced Tree $h = O(\log n)$

- Claim:
 - For a tree with height h , a height-balanced tree contains at least $n > 2^{h/2}$ nodes
- Prove by Induction:
 - Assuming it is true for $h < k$.
 - For $h = k$,

$$\begin{aligned}n_k &\geq 1 + n_{k-1} + n_{k-2} \\&\geq 2 n_{k-2} \\&\geq 2 \times 2 n_{k-4} \\&\geq 2 \times 2 \times 2 n_{k-6} \\&\geq 2 \times 2 \times 2 \times 2 n_{k-8}\end{aligned}$$



Proof of Height Balanced Tree $h = O(\log n)$

- For $h = k$, the minimal no. of node will be at least

$$n_k \geq 1 + n_{k-1} + n_{k-2} \geq 2 n_{k-2}$$

$$\geq 2 \times 2 n_{k-4}$$

$$\geq 2 \times 2 \times 2 n_{k-6}$$

$$\geq 2 \times 2 \times 2 \times 2 n_{k-8}$$

$$\geq 2^i n_{k-2i}$$

$$\geq 2^{k/2} n_0 = 2^{k/2}$$

$$i = \frac{k}{2}$$

- For $h = k$, $n_k \geq 2^{k/2}$

Height-Balanced Trees

- A height balanced binary search tree with n nodes has at most height:

$$h < 2 \log n$$

$$\Leftrightarrow h/2 < \log(n)$$

$$\Leftrightarrow 2^{h/2} < 2^{\log(n)}$$

$$\Leftrightarrow 2^{h/2} < n$$

\Leftrightarrow For a tree with height h , a height-balanced tree contains at least $n > 2^{h/2}$ nodes

Proven

Height-Balanced Trees

- A height balanced binary search tree with n nodes has at most height $h < 2 \log n = O(\log n)$

Q.E.D.

Step 3: How to Maintain Height-balance

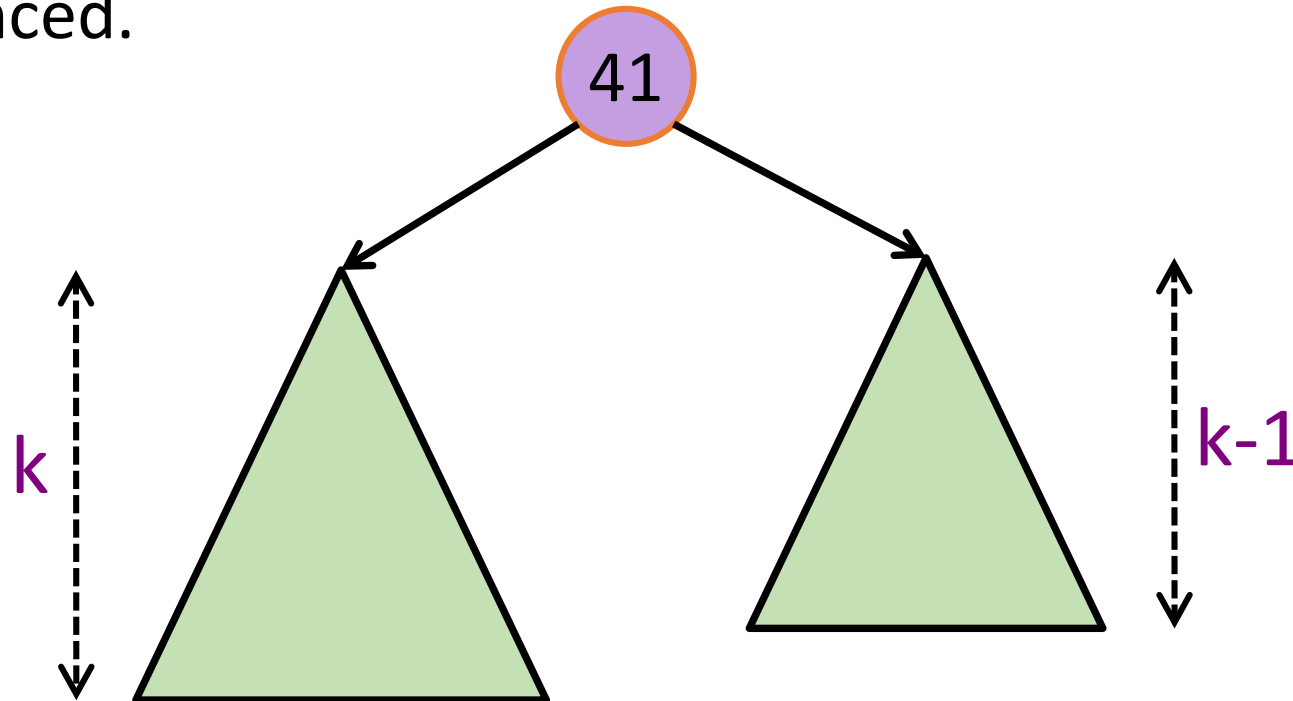


Step 3: How to Maintain Height-balance

- A node v is **height-balanced** if:

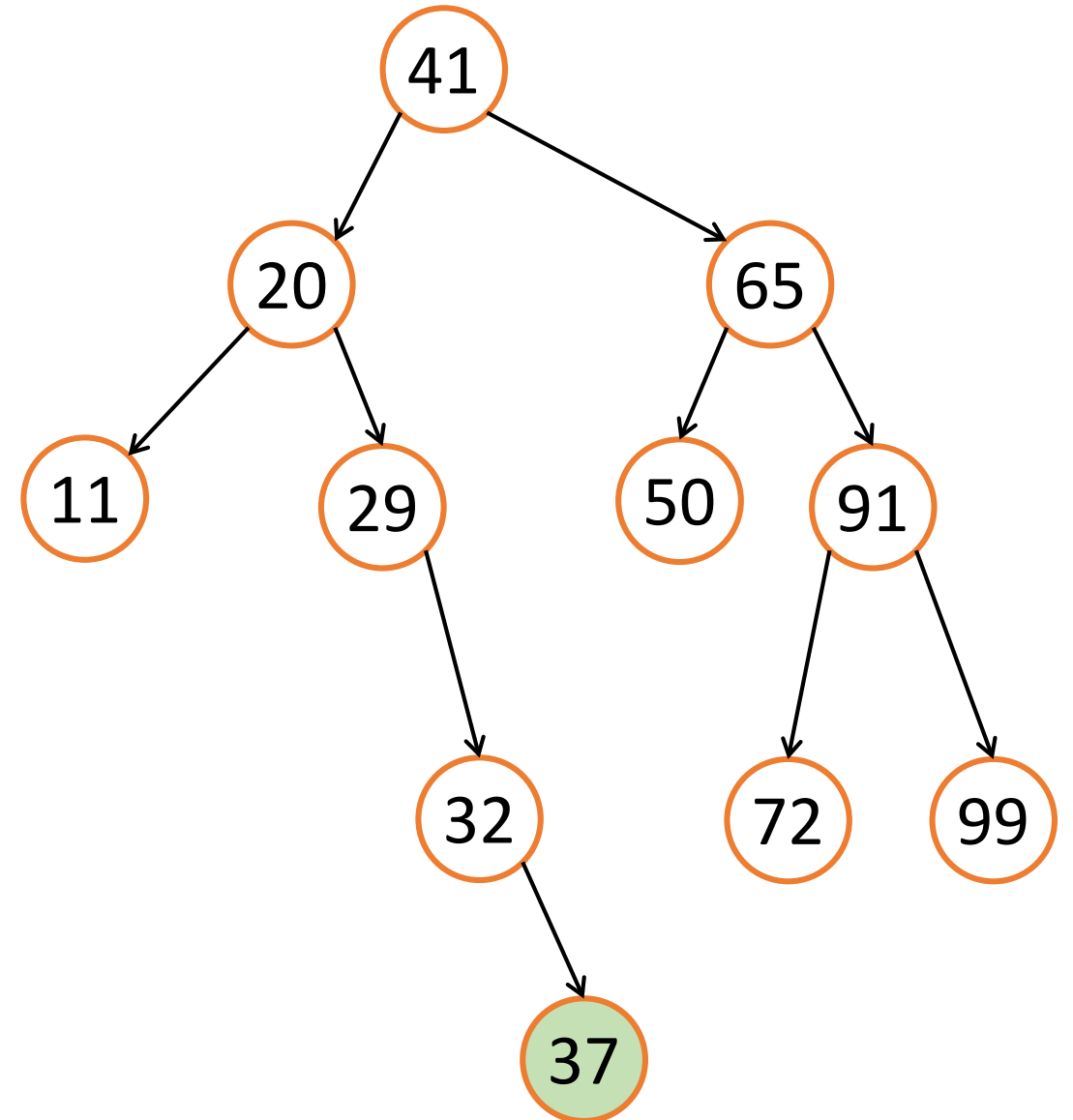
$$\text{abs}(v.\text{left}.\text{height} - v.\text{right}.\text{height}) \leq 1$$

- A binary search tree is **height balanced** if every node in the tree is height-balanced.



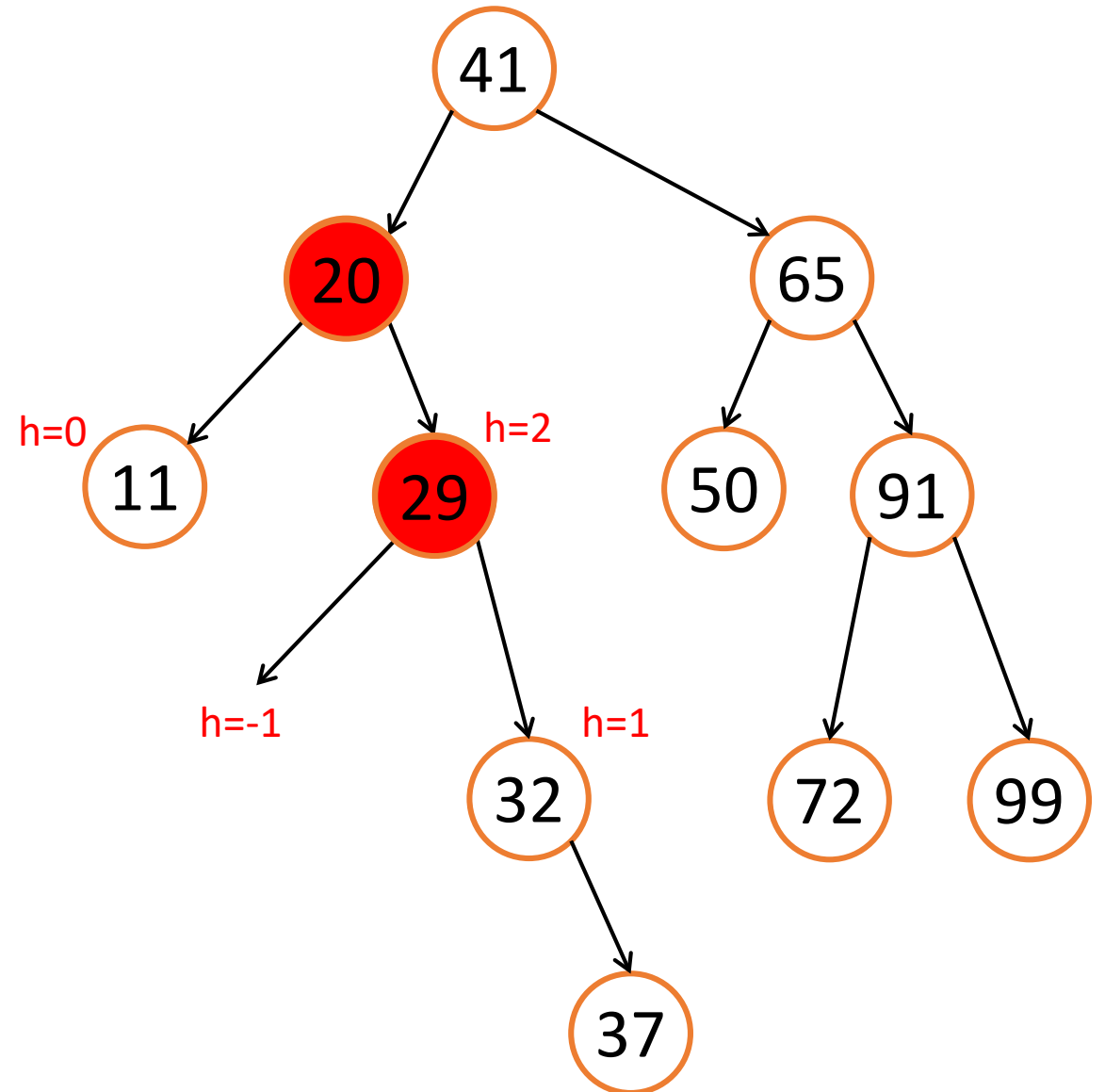
Example: Insert(37)

- Before insertion, the tree is balanced
- But not anymore after inserting 37
- Need to do something to make it balanced



Example: Insert(37)

- What are the nodes that are NOT balanced?



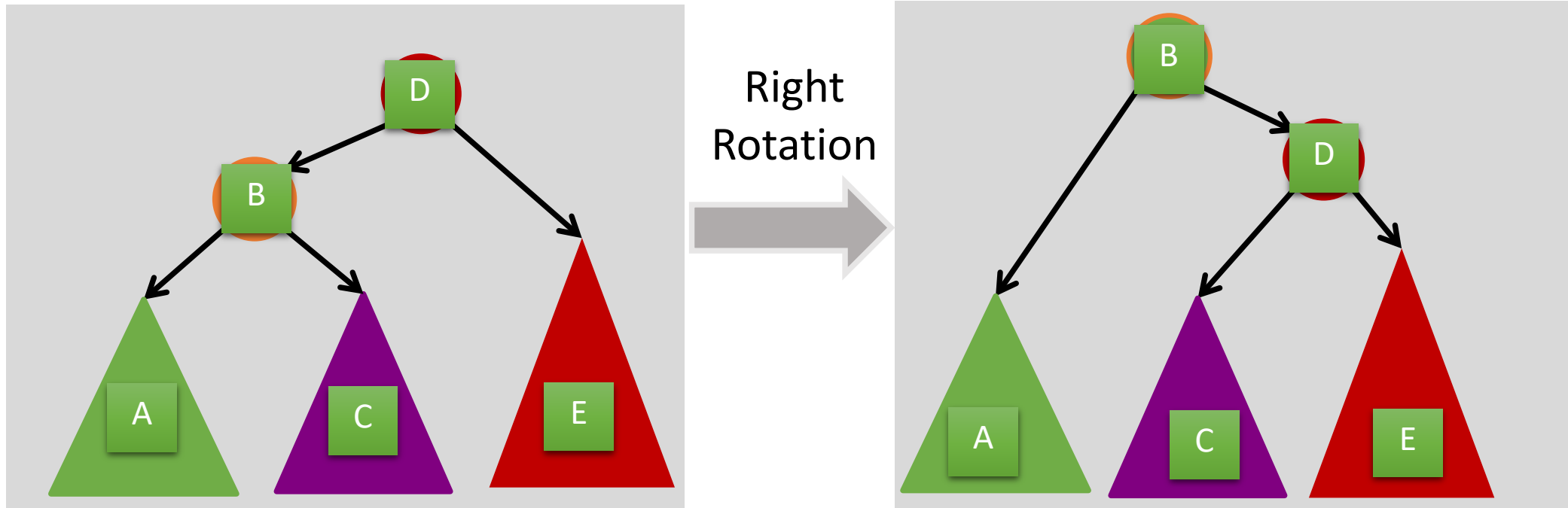
Tricks to Rebalance the Tree

- Tree Rotation!

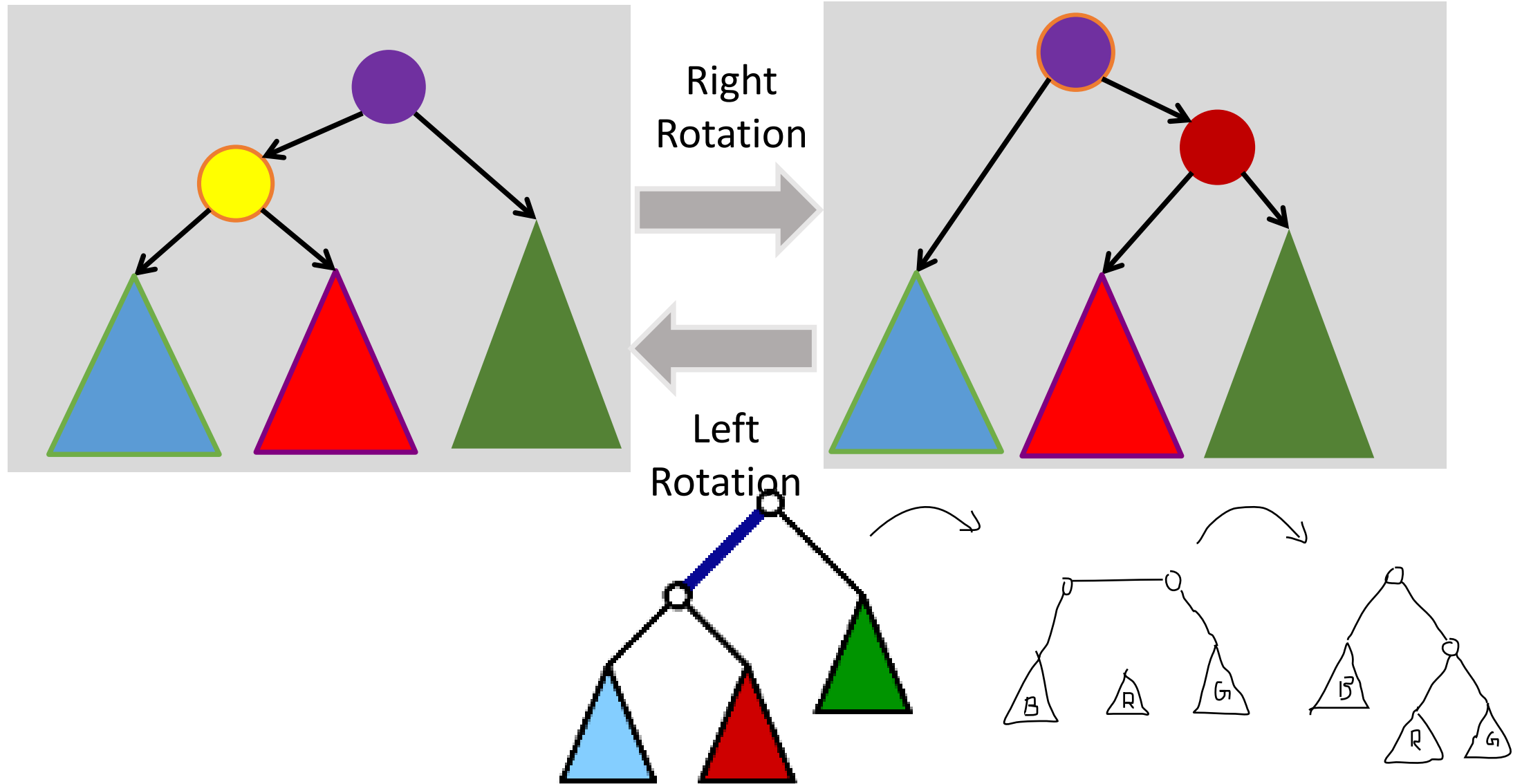


Tree Rotation

- $A < B < C < D < E$
- Rotations maintain ordering of keys. \Rightarrow Maintains BST property.

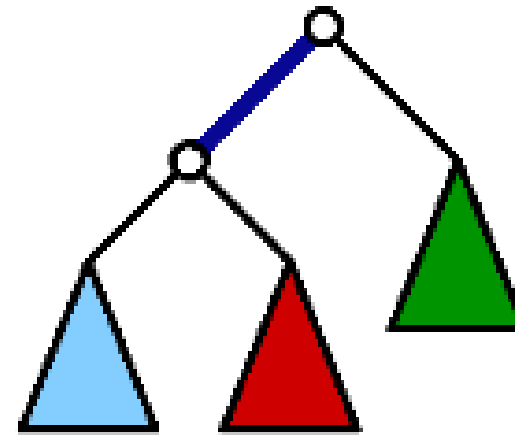
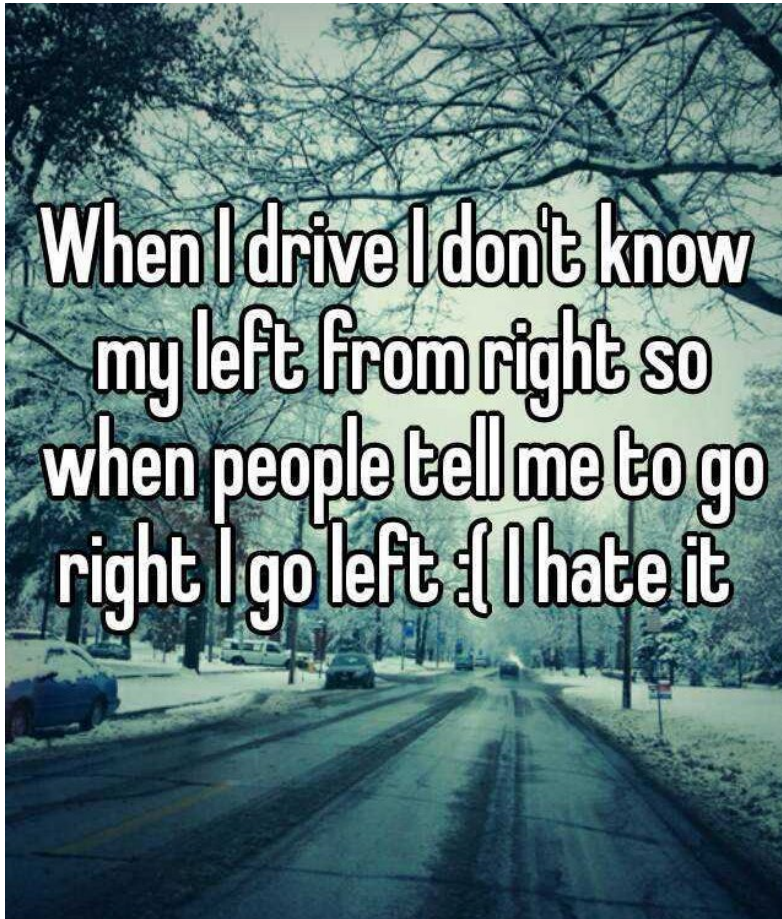


Left and Right Tree Rotations



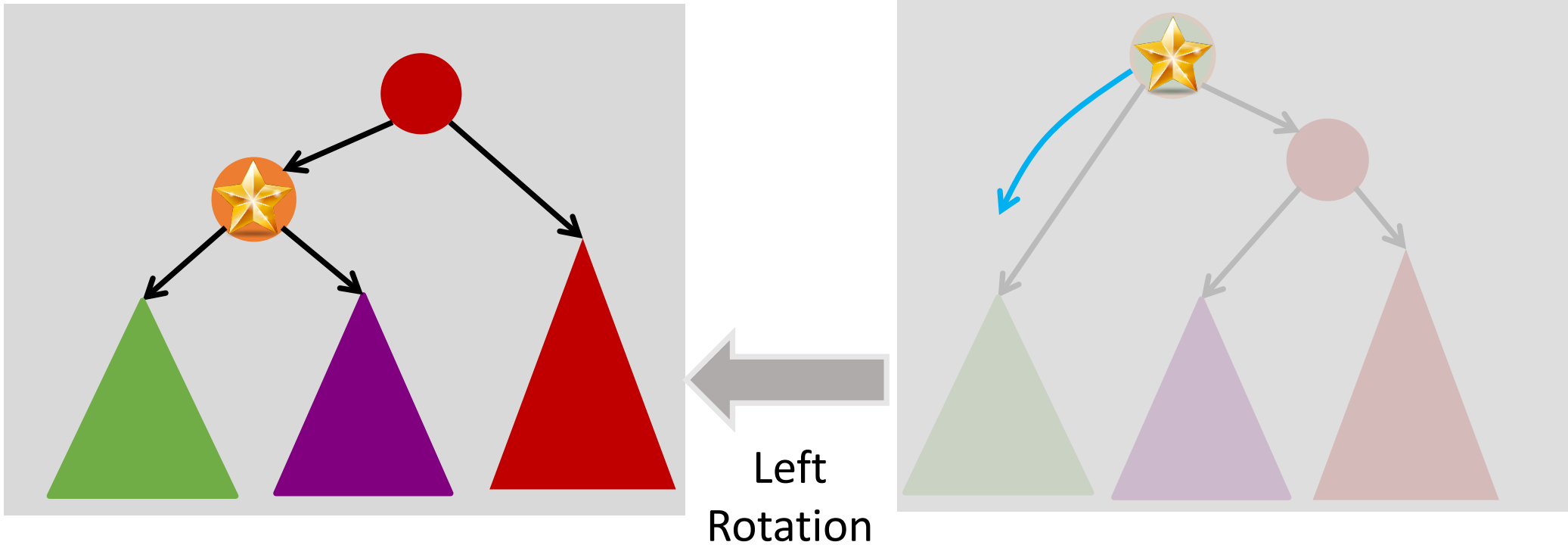
I cannot tell the left from the right

- Wait, what is a left rotation and what is a right rotation?



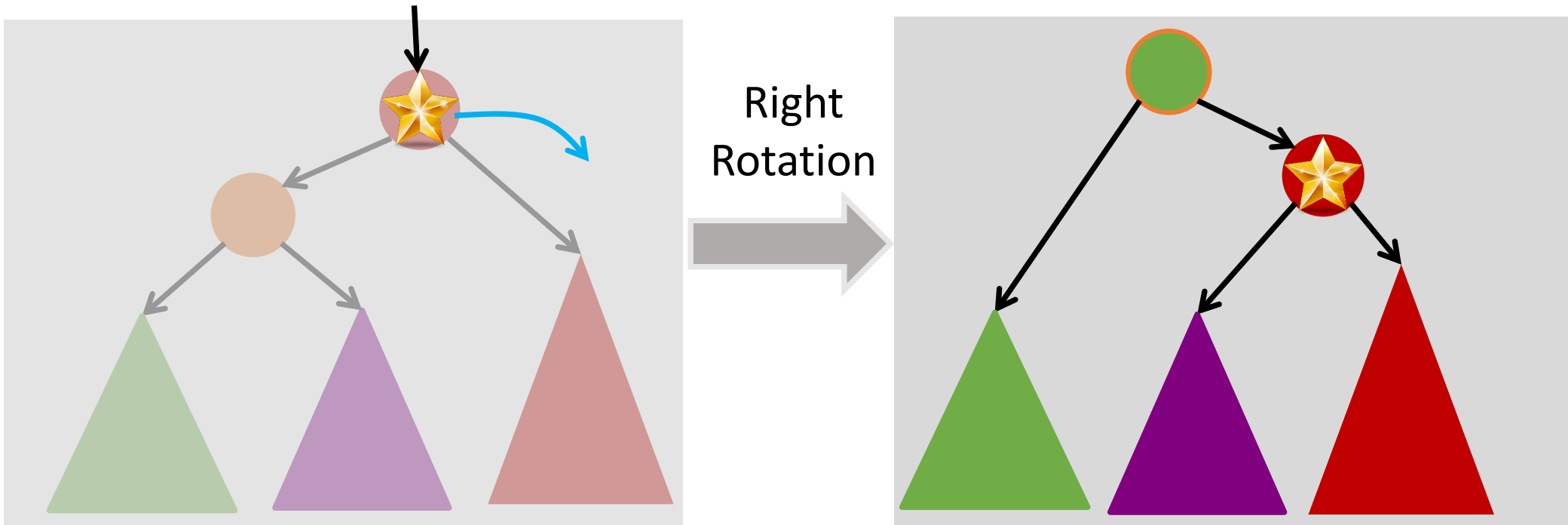
My way to remember it

- Left rotation = The original root of the subtree moves left



My way to remember it

- Right rotation = The original root of the subtree moves right



Right Rotation

```
right-rotate(v)  // assume v has left != null
```

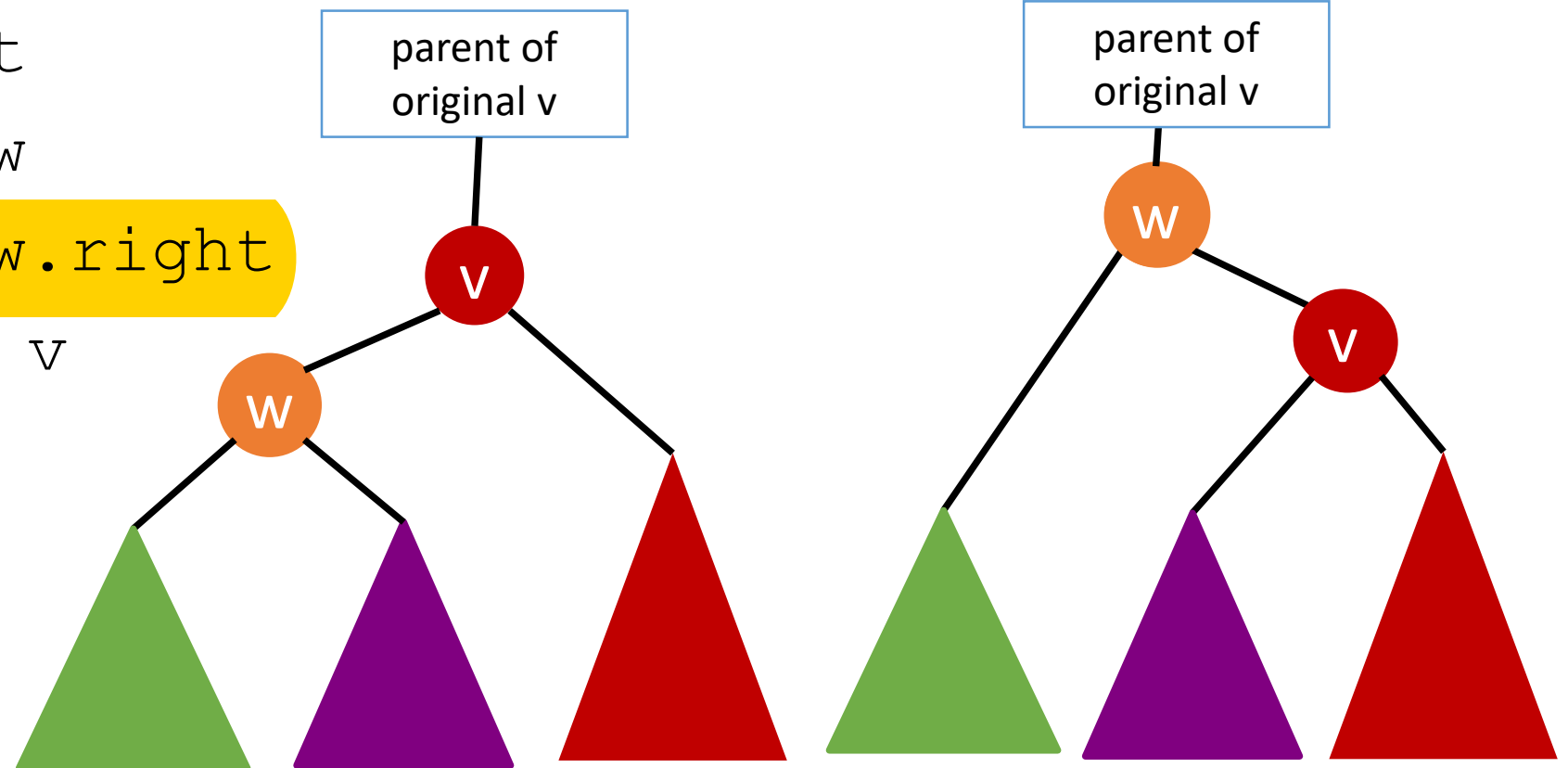
Let the parent of v is the pointer "parent"

```
w = v.left
```

```
parent = w
```

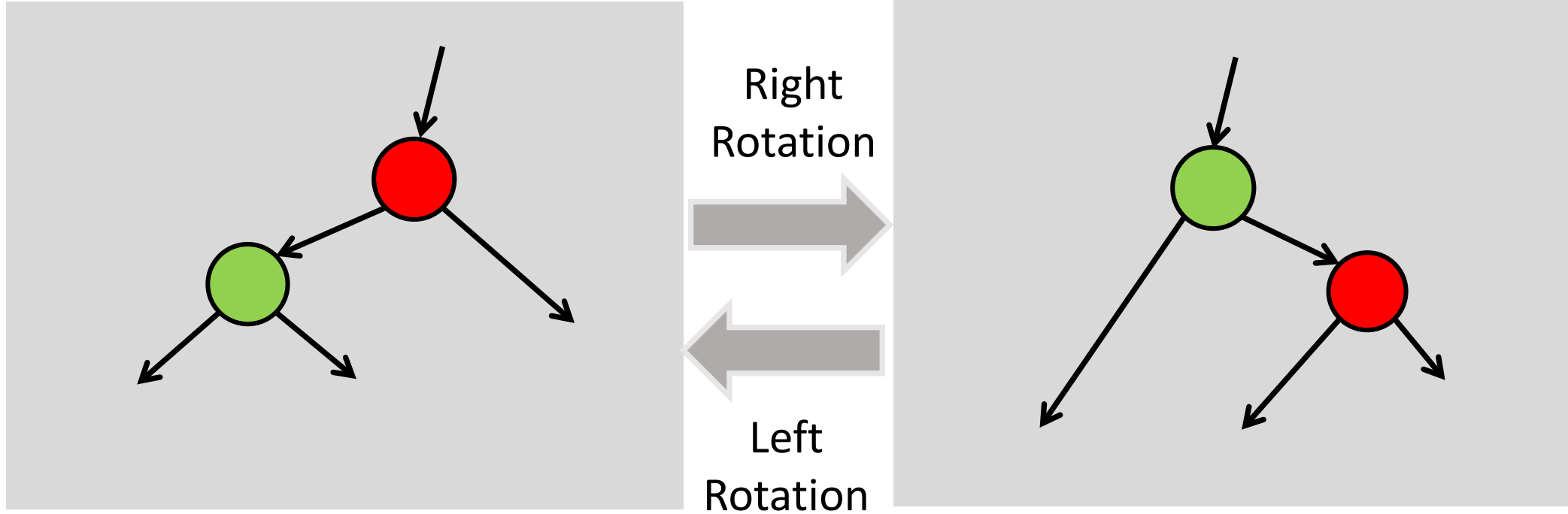
```
v.left = w.right
```

```
w.right = v
```



Children Requirements

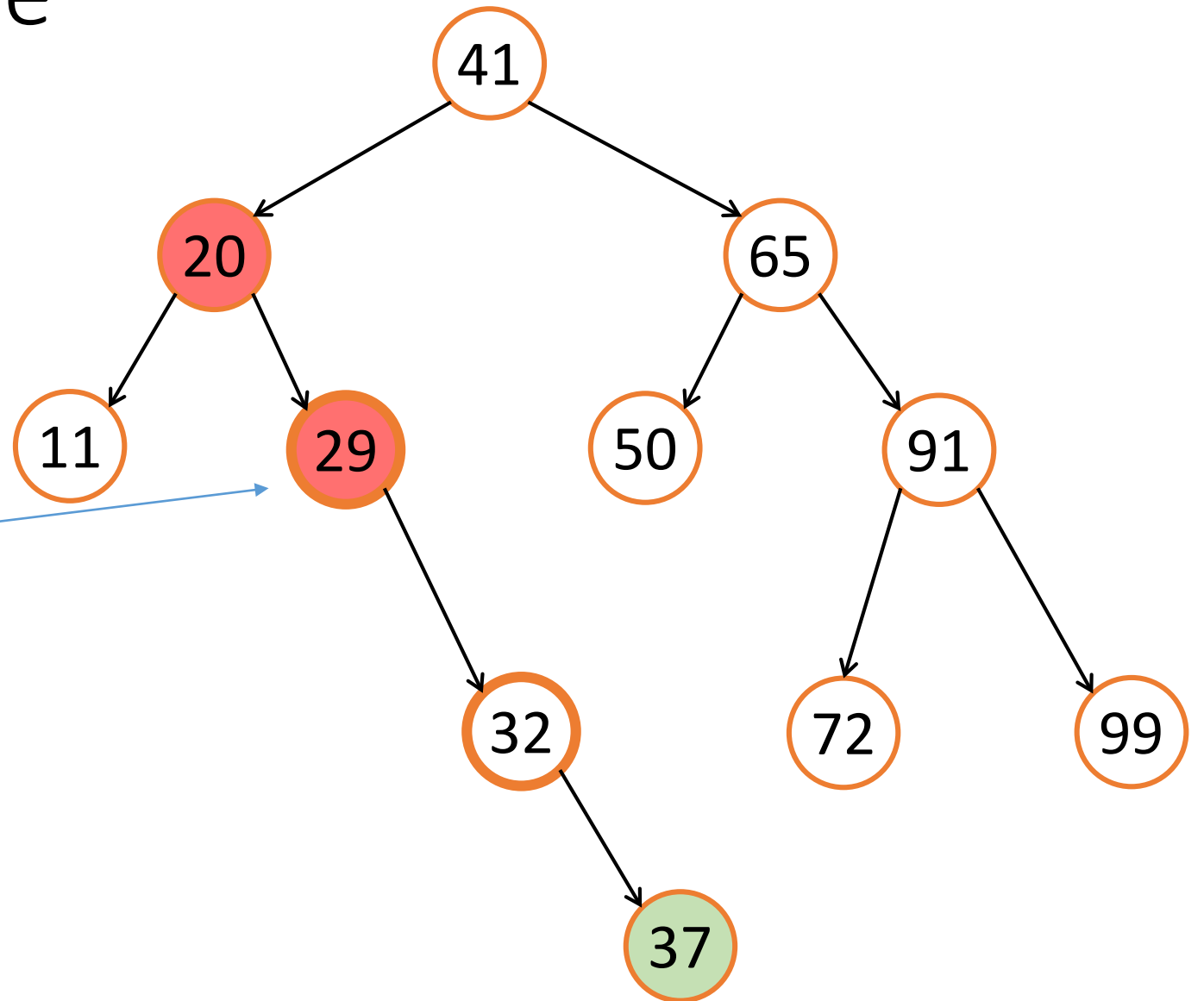
- rotate-right requires a left child
- rotate-left requires a right child



Balancing by Rotations

Insertion in AVL Tree

- After inserting 37, the tree is no longer balanced
- A node that is out of balance can be
 - left heavy or
 - right heavy



If a Node is **Left** Heavy and Needs Balancing

If v is out of balance and **left heavy**:

```
| v.left is balanced: right-rotate(v)  
| v.left is left-heavy: right-rotate(v)  
| v.left is right-heavy:  
    left-rotate(v.left) then right-rotate(v)
```

• Or

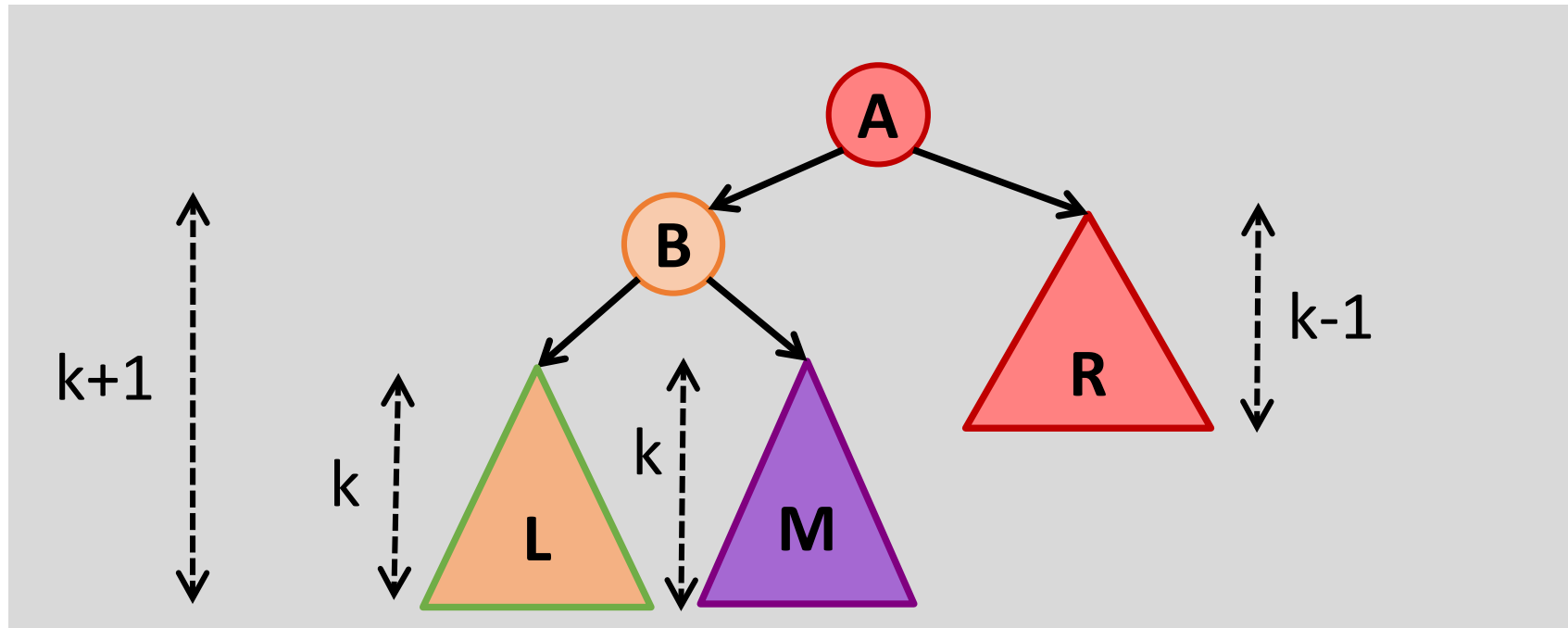
If v is out of balance and **left heavy**:

```
if v.left is right-heavy: left-rotate(v.left)  
right-rotate(v)
```

The gist of AVL Tree Rotations

Proof of Correctness (A is left heavy)

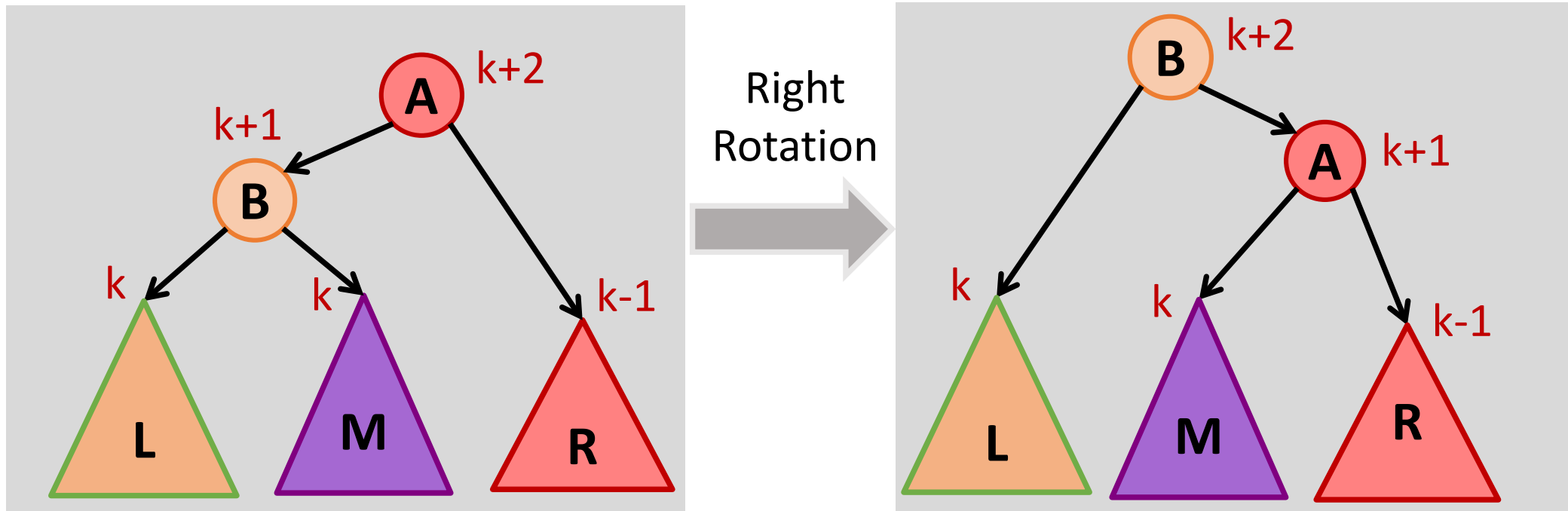
- Assume A is the lowest node in the tree violating balance property



- Case 1: B is balanced

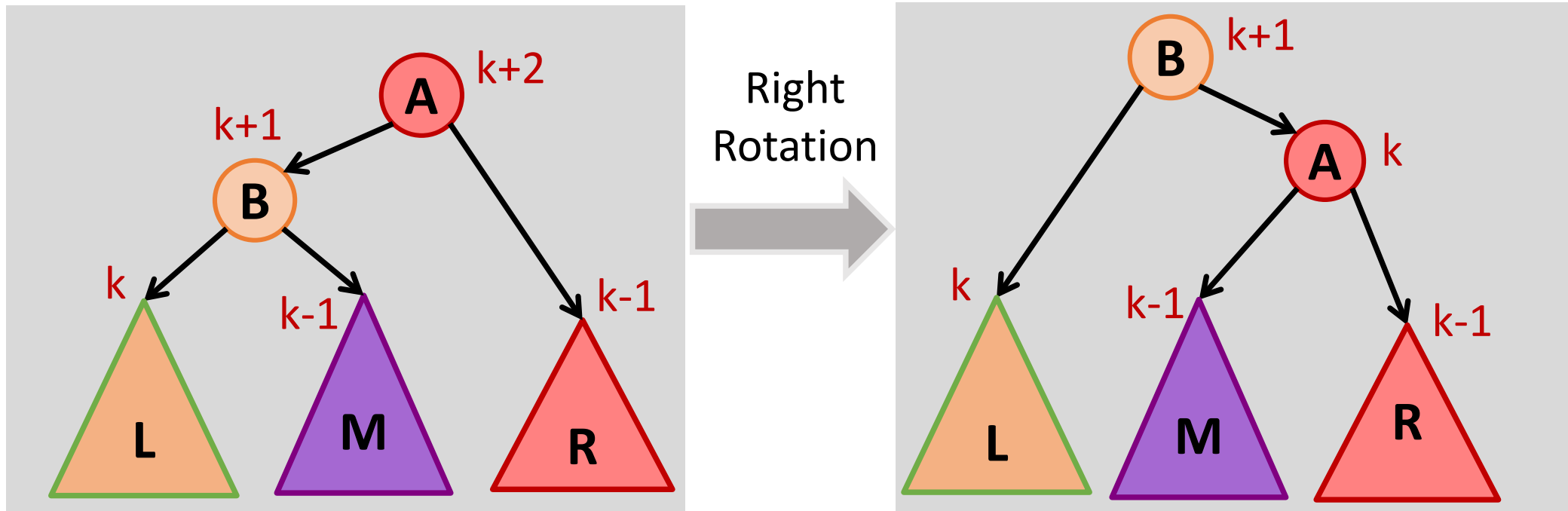
Case 1: A is left-heavy, B is balanced

- After right-rotate(A)



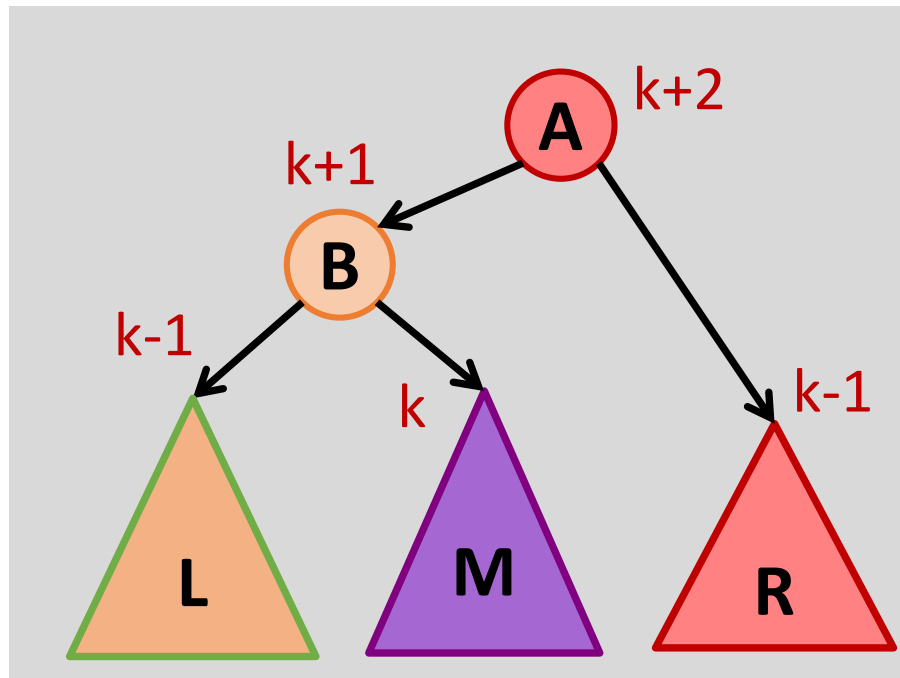
Case 2: A is left-heavy, B is **left-heavy**

- After right-rotate(A)

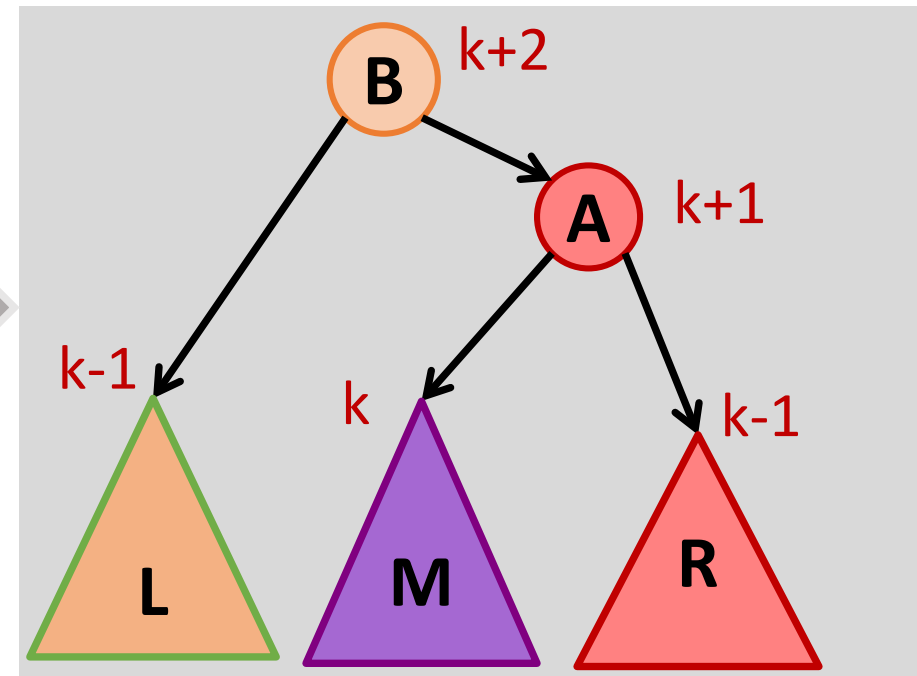


Case 3: A is left-heavy, B is right-heavy

- Just do a right-rotate(B)?

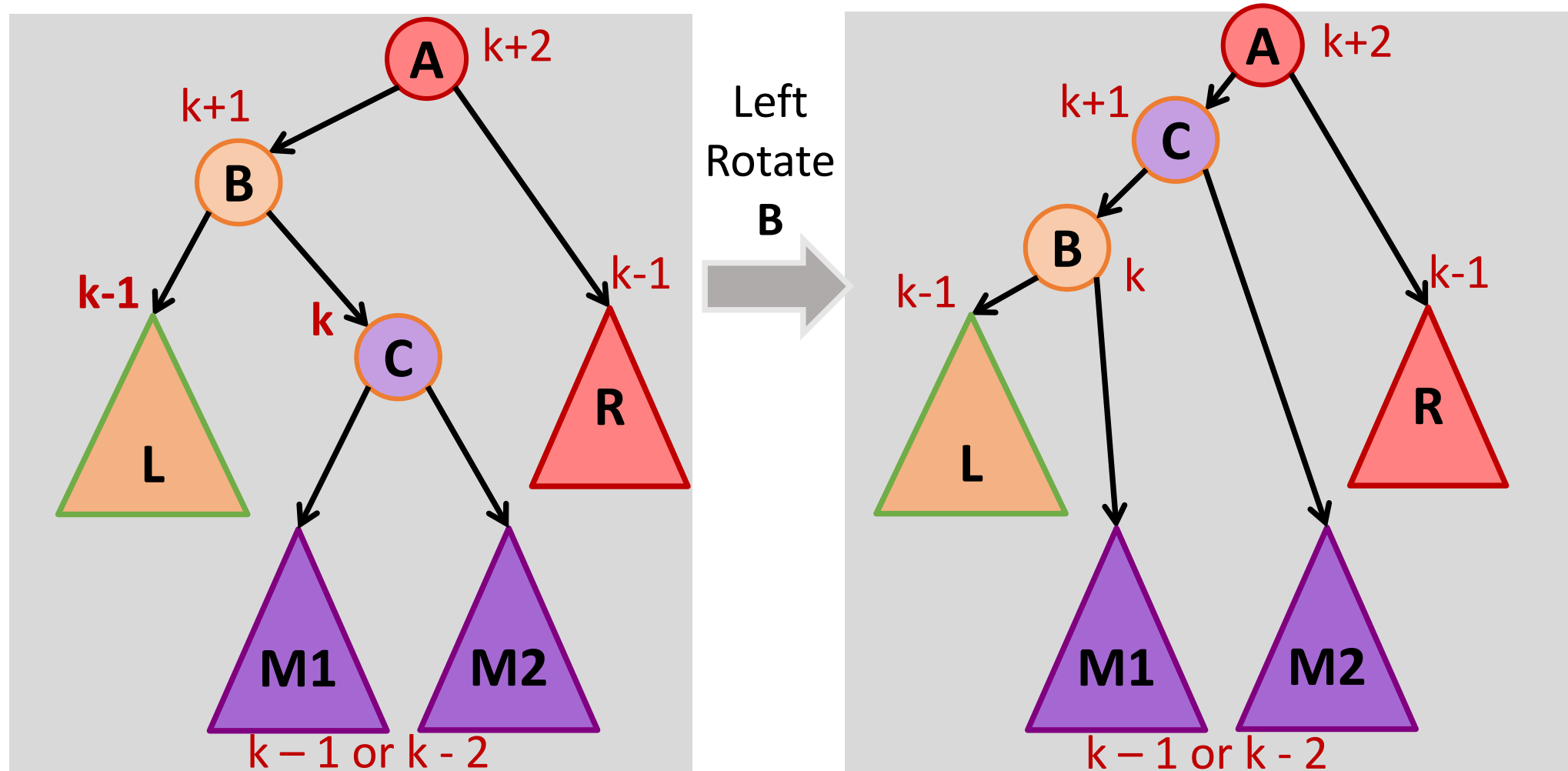


Right
Rotation



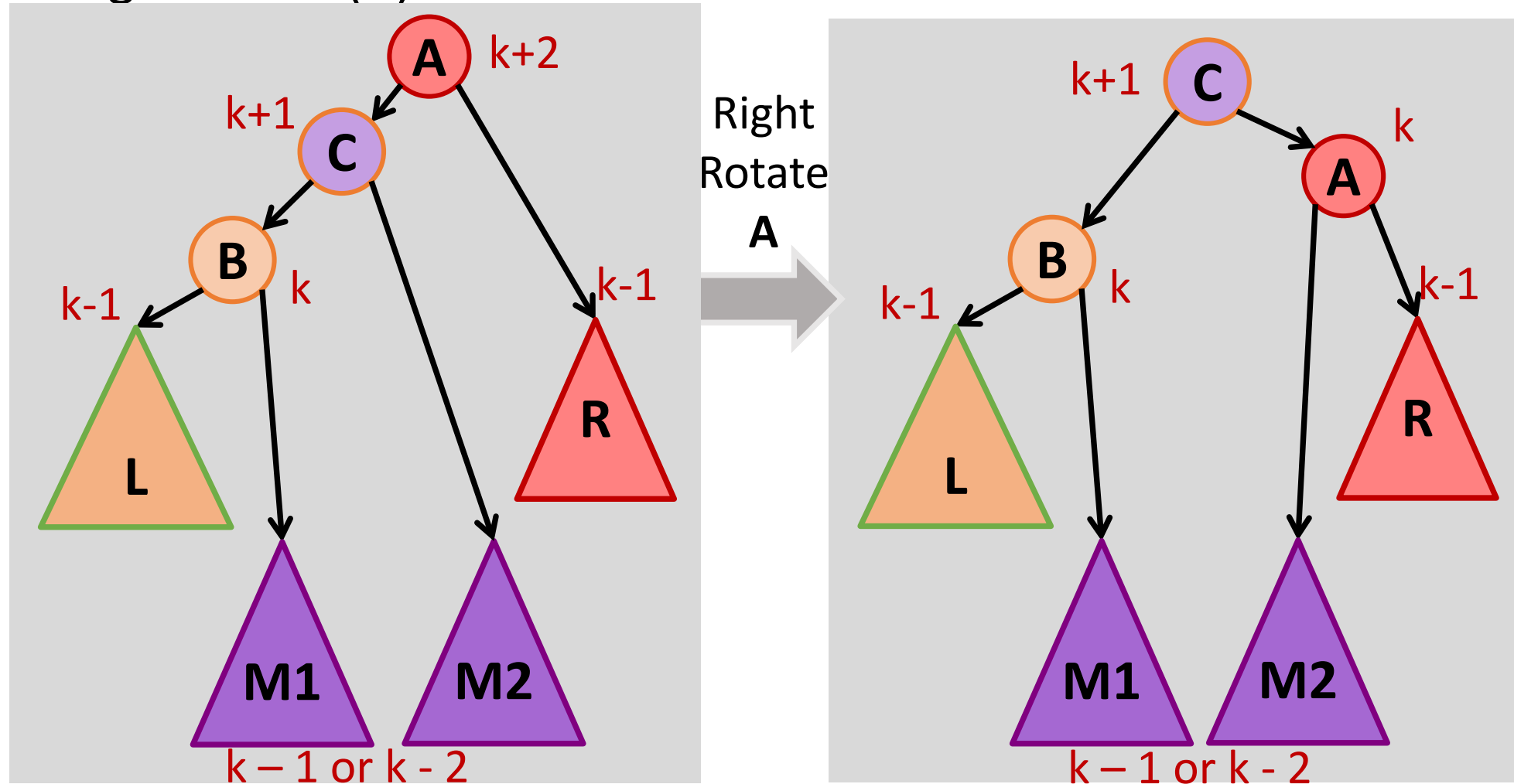
Case 3: A is left-heavy, B is right-heavy

- Left-rotate(B) first



Case 3: A is left-heavy, B is right-heavy

- Then right-rotate(A)



If a Node is **Left** Heavy and Needs Balancing

If `v` is out of balance and **left heavy**:

`v.left` is balanced: `right-rotate(v)`

`v.left` is left-heavy: `right-rotate(v)`

`v.left` is right-heavy:

`left-rotate(v.left)` then `right-rotate(v)`

- Or

If `v` is out of balance and **left heavy**:

if `v.left` is right-heavy: `left-rotate(v.left)`

`right-rotate(v)`

If a Node is **Right Heavy** and Needs Balancing

If `v` is out of balance and **right heavy**:

`v.right` is balanced: `left-rotate(v)`

`v.right` is right-heavy: `left-rotate(v)`

`v.right` is left-heavy:

`right-rotate(v.right)` then `left-rotate(v)`

- Or

If `v` is out of balance and **right heavy**:

if `v.right` is left-heavy: `right-rotate(v.right)`

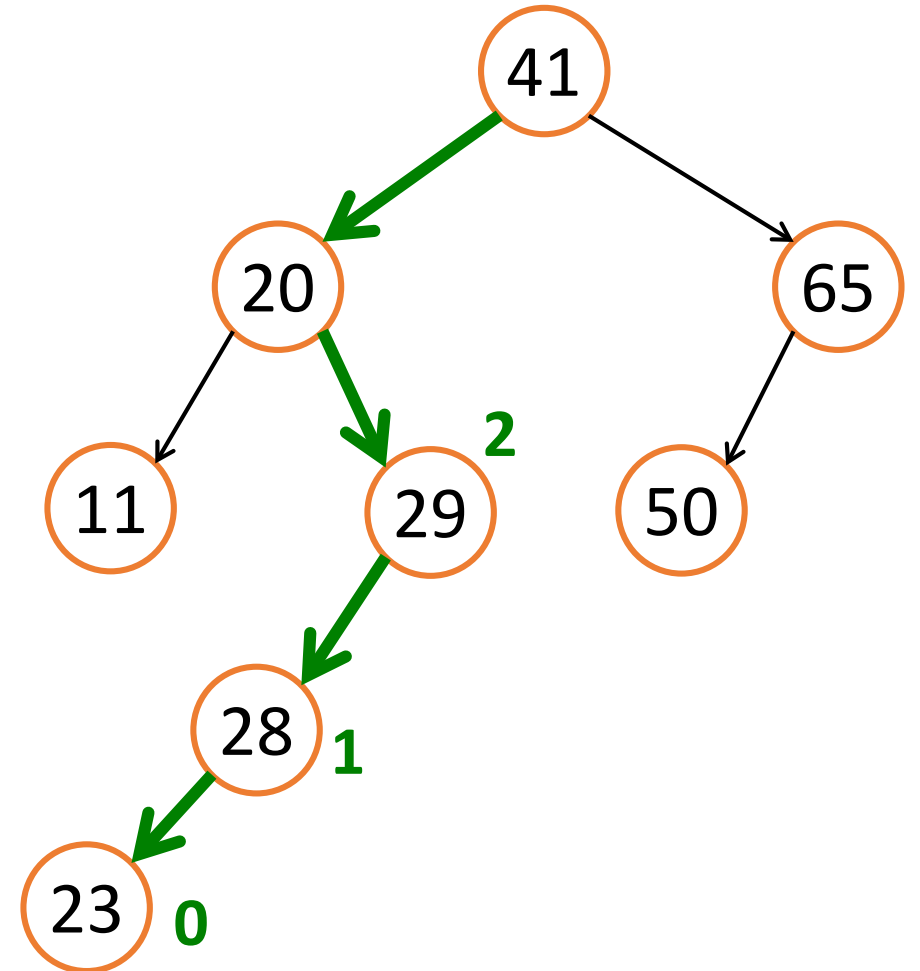
`left-rotate(v)`

AVL Tree Insertion with Balancing

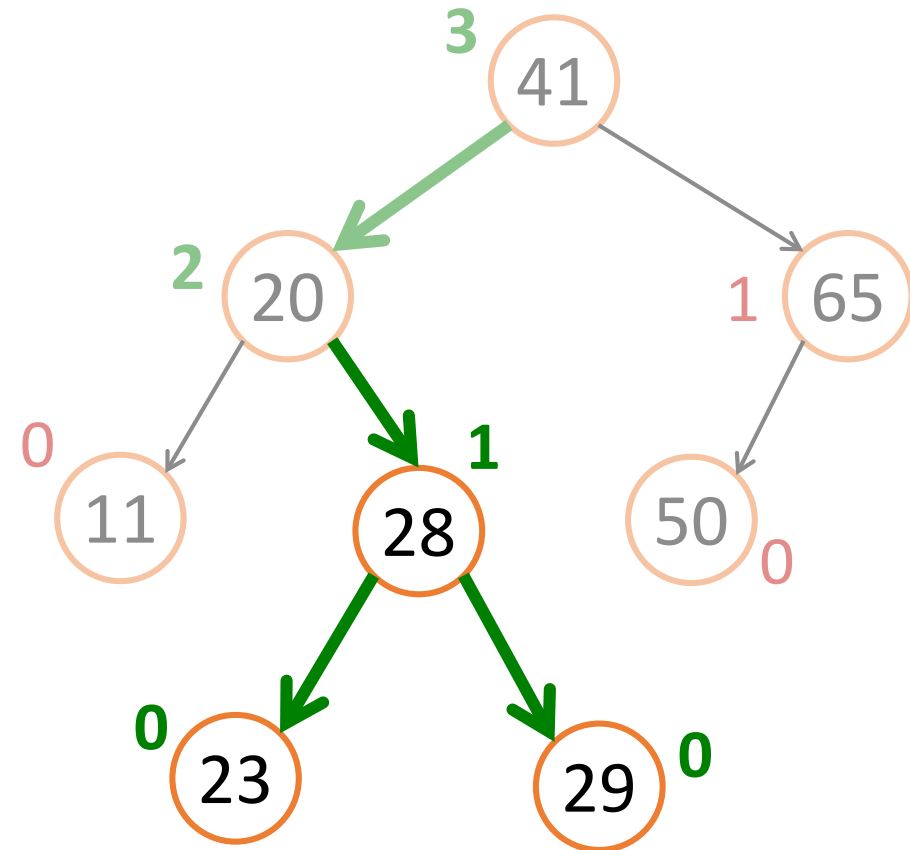
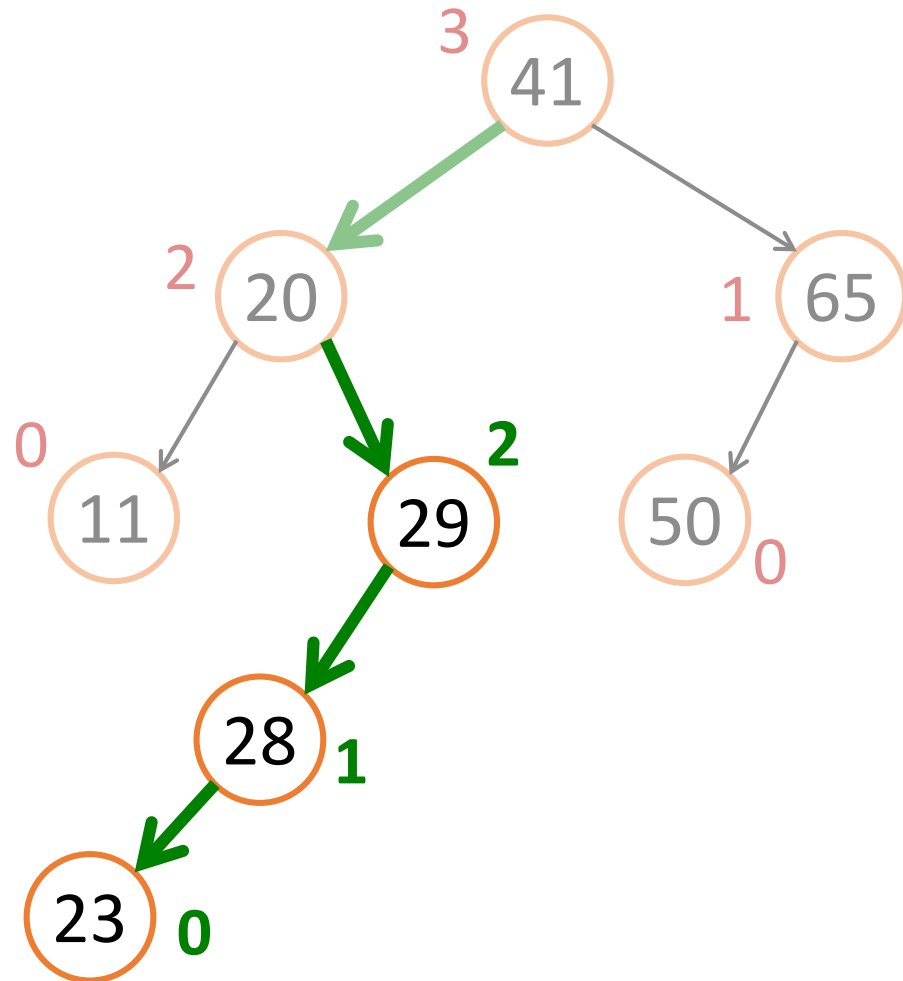
- Steps:
 - Insert key in BST.
 - Walk up tree:
 - At every step, check for balance.
 - If out-of-balance, use rotations to rebalance.
- Note: only need to perform at most two rotations
 - Why?
 - In each case, reduce height of sub-tree by 1
 - What about Case 1, above?

Example: Insert(23)

- Which node should we rotate?
- Out of the 3 cases, which case is it?
 - Case2: Left heavy and Left.left heavy
- Just do one right-rotate!

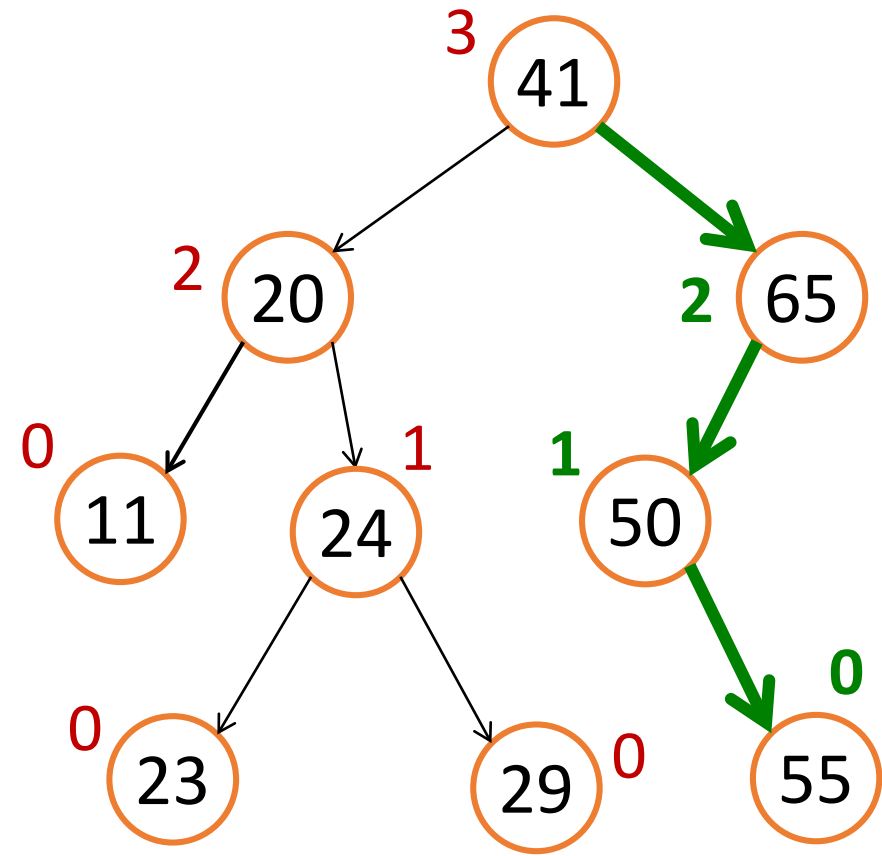


right-rotate(29)

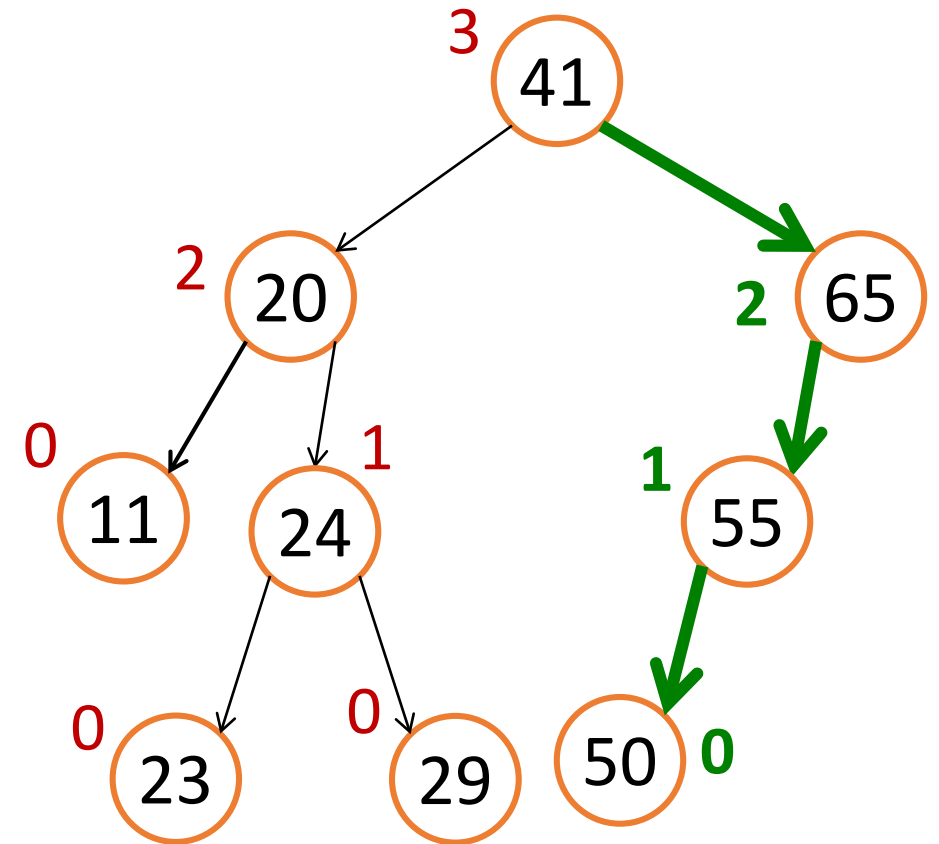
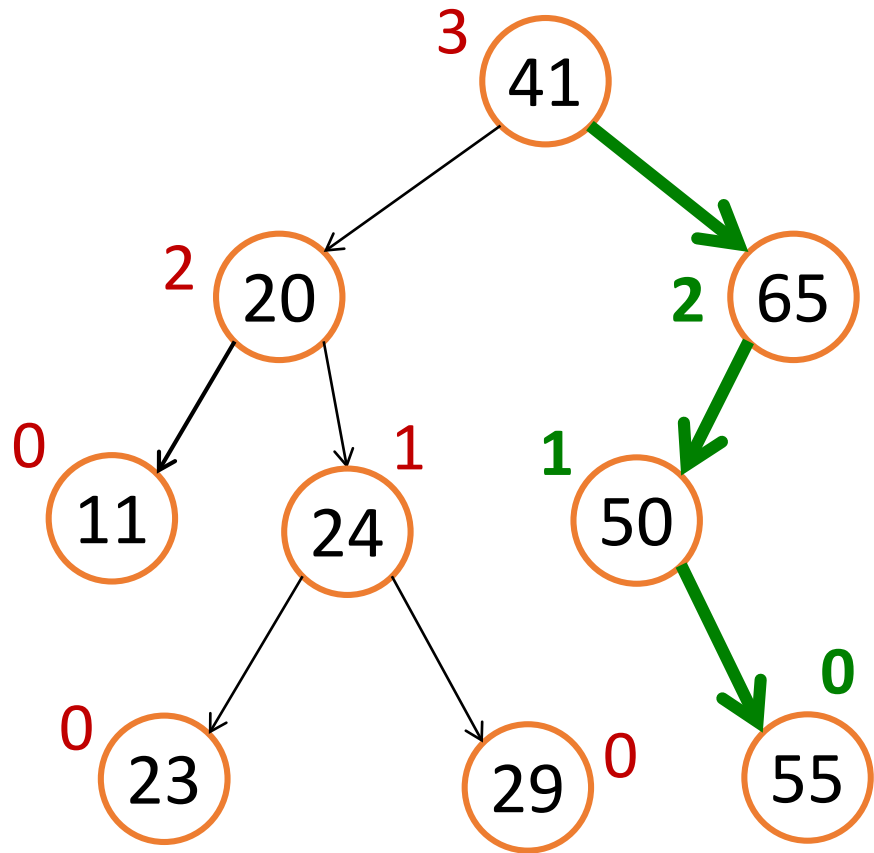


Another Example: Insert(55)

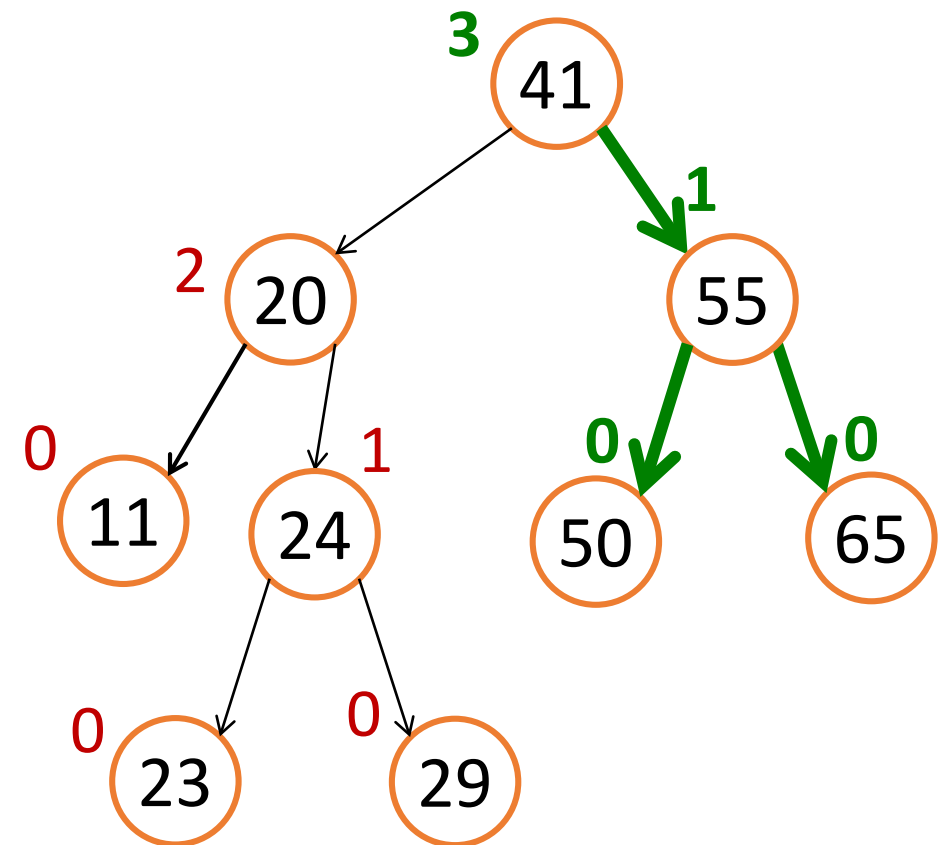
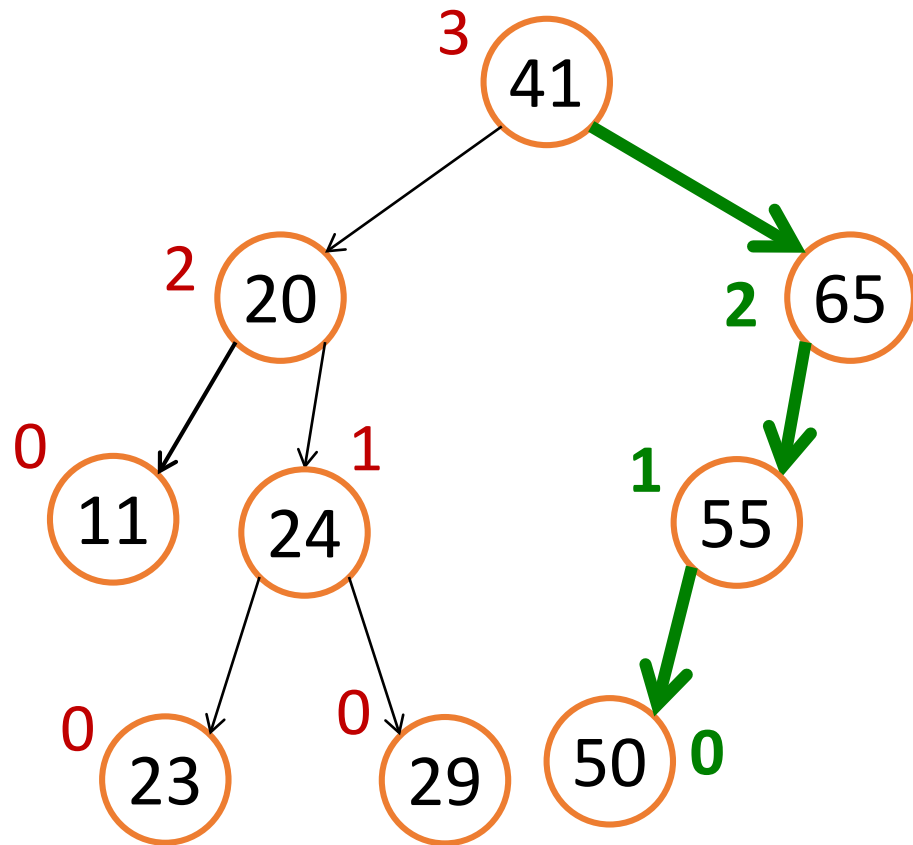
- Which node should we rotate?
- Out of the 3 cases, which case is it?
 - Case3: Left heavy and Left.right heavy
- Do two rotations!
 - left-rotate(50)
 - right-rotate(65)



left-rotate(50)



right-rotate(65)

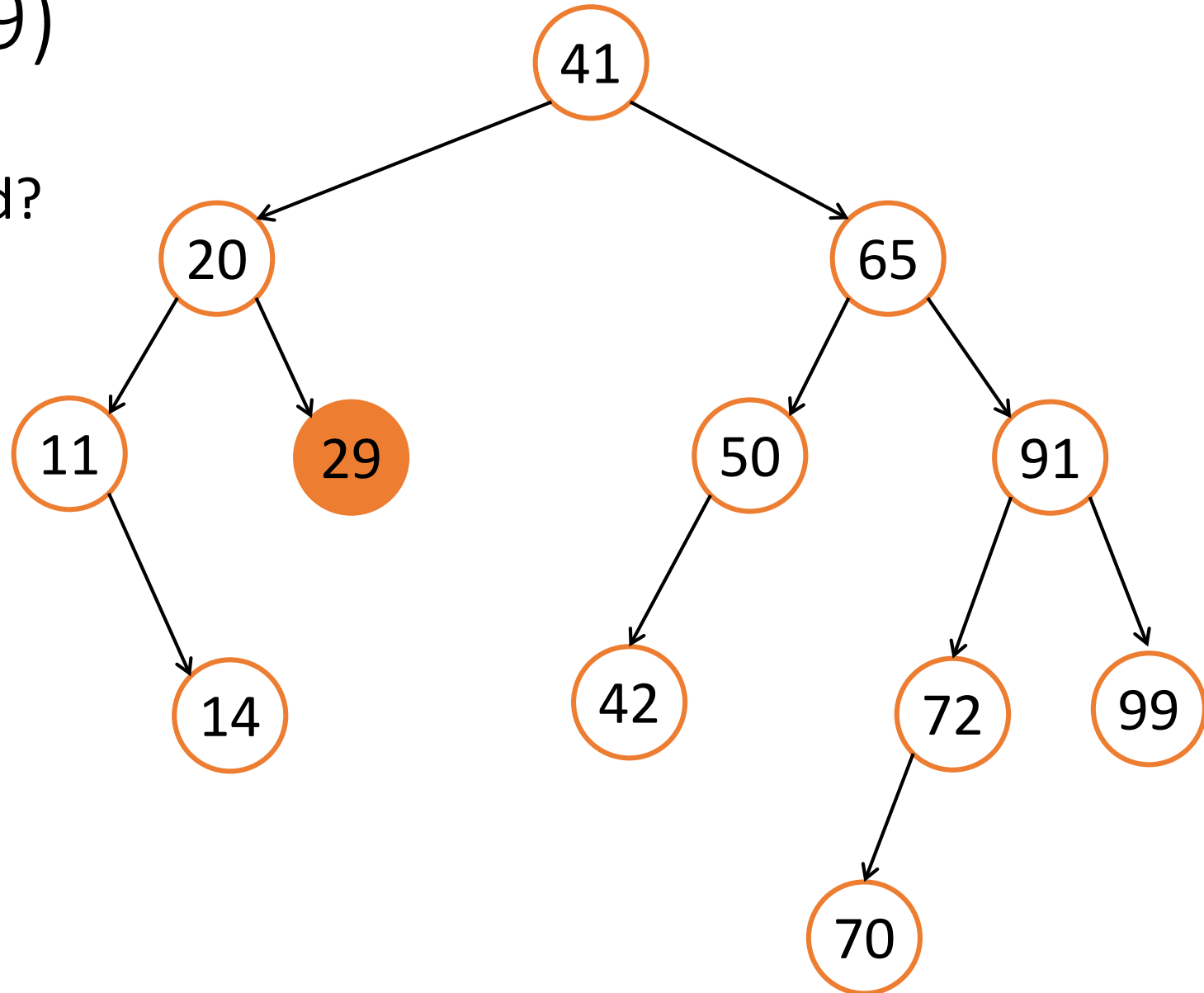


How about balancing after deletion?

- From the deleted node, walk up to the root to check every parent if they are balance
- If not balanced, perform the balancing like insertion

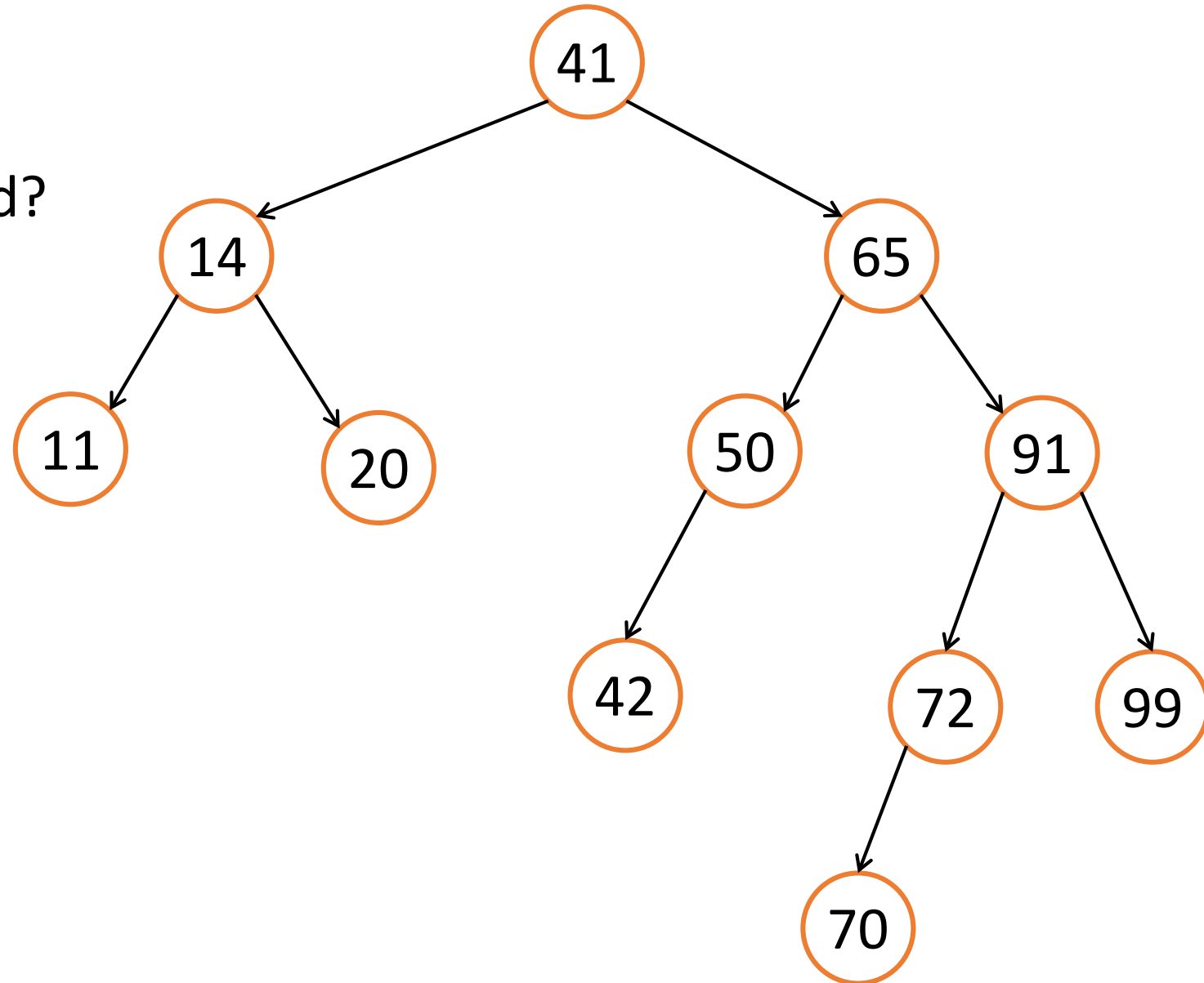
Example: delete(29)

- Which node is not balanced?
- Which case is that?
 - Case 3
- left-rotate(11)
- right-rotate(20)



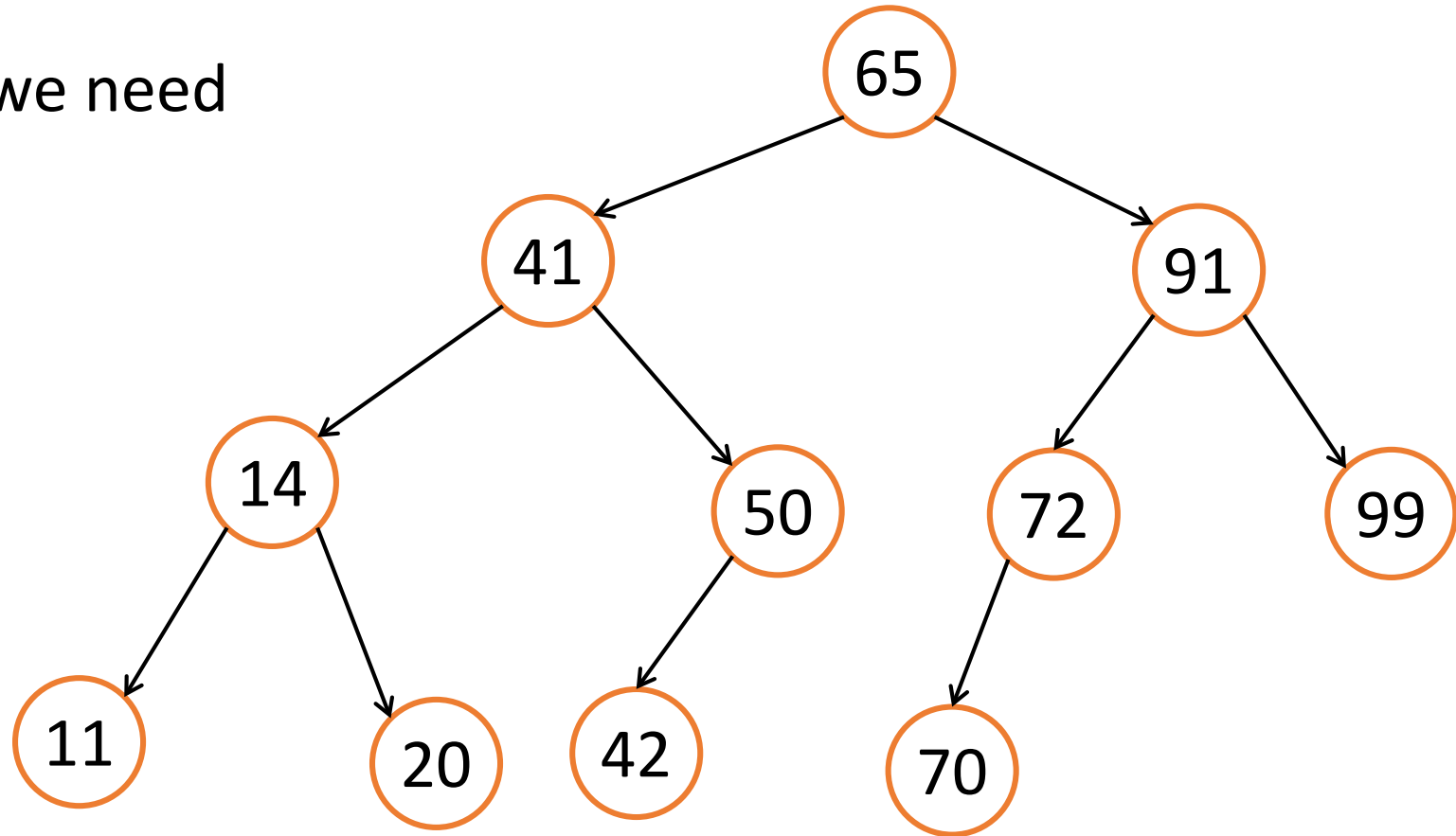
Are we done?

- Which node is not balanced?
- Which case is that?
 - Case 2: (RR case)
- left-rotate(41)



Finally

- Are we done?
- How many rotations do we need for *any* deletion?



AVL Tree Balancing

- Insertion:
 - Needs 0 to 2 rotations to balance
- Deletion:
 - Needs up to $O(\log n)$ rotations to balance
 - Tricks to deal with it:
 - Just mark the deleted node as “deleted” and leave it in the tree, instead of removing it from memory
 - Performance degrades over time
 - Clean up later? (Amortized performance...)

Many different flavors of balanced search trees

- AVL trees (Adelson-Velsii & Landis, 1962)
- B-trees / 2-3-4 trees (Bayer & McCreight, 1972)
- BB[α] trees (Nievergelt & Reingold 1973)
- Red-black trees (see CLRS 13)
- Splay trees (Sleator and Tarjan 1985)
- Treaps (Seidel and Aragon 1996)
- Skip Lists (Pugh 1989)
- Scapegoat Trees (Anderson 1989)

Red-Black trees

- More loosely balanced
- Rebalance using rotations on insert/delete
- $O(1)$ rotations for all operations.
- Java TreeSet implementation
- Faster (than AVL) for insert/delete
- Slower (than AVL) for search