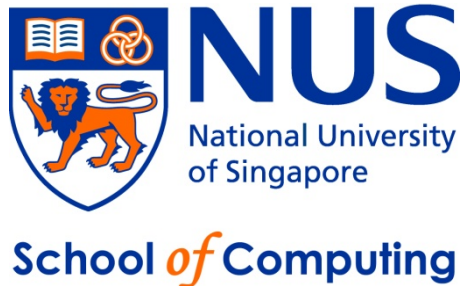


# CS2040 – Data Structures and Algorithms

Lecture 11 – Balancing Act ~ AVL Tree

[chongket@comp.nus.edu.sg](mailto:chongket@comp.nus.edu.sg)



# Outline

## Binary Search Tree (BST): A Quick Revision

### The Importance of a **Balanced** BST

- To keep  $h = O(\log N)$

### Adelson-Velskii Landis (AVL) Tree

- Principle of “Height-Balanced”
- Keeping AVL Tree balanced via rotations
- Code is not given (implement this for ~~optional~~ DS challenge THL !)



Eat All Foods In Moderation

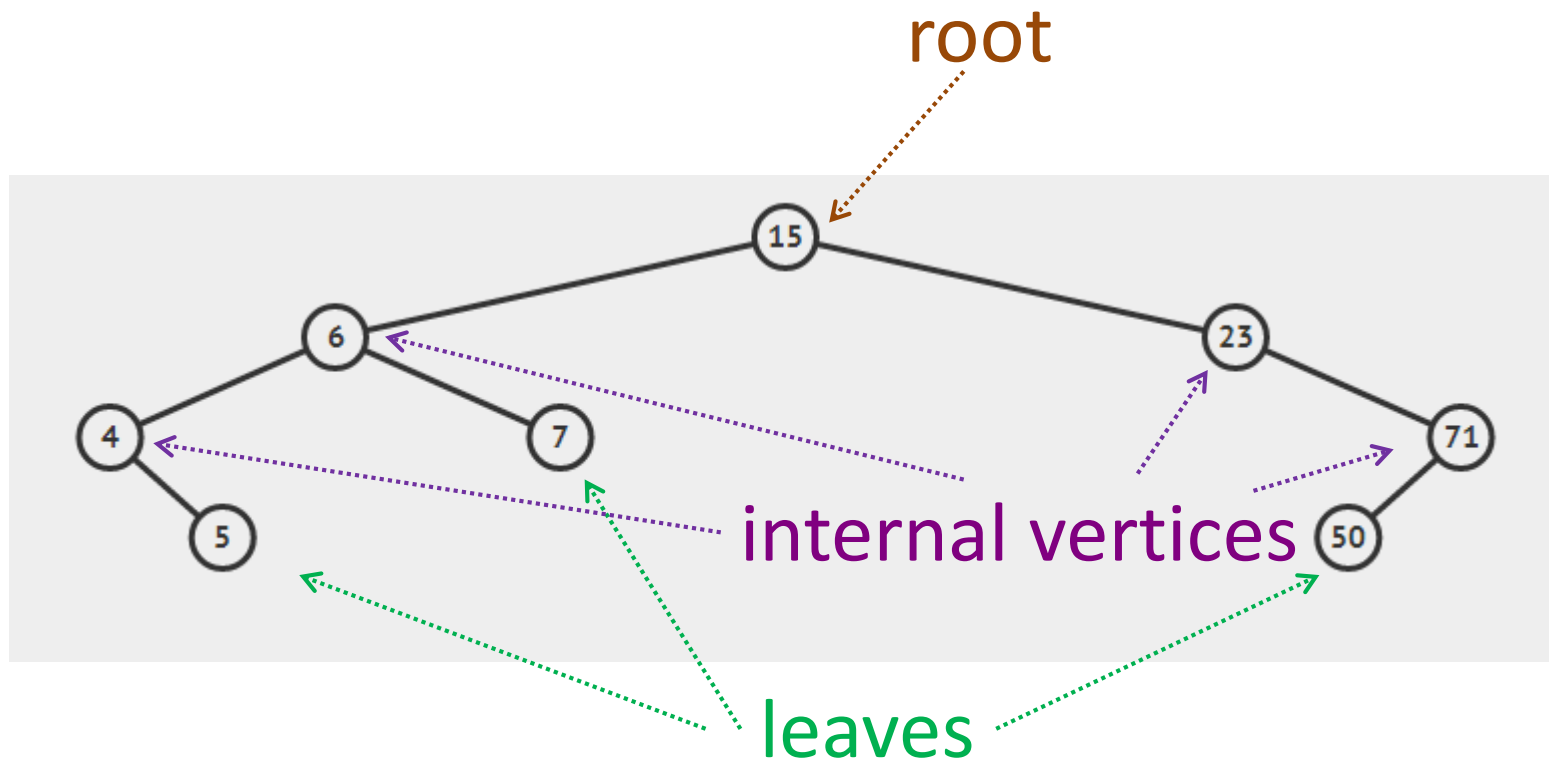
Reference in CP3 book: Page 43-47 + **380-382**

Page 56-58 (for newer edition)

# BST Web-based Review

---

<https://visualgo.net/bst>



# Binary Search Tree: Summary

---

Operations that **modify** the BST (*dynamic* data structure):

- insert:  $O(h)$
- delete:  $O(h)$

Query operations (the BST structure remains the same):

- search:  $O(h)$
- findMin, findMax:  $O(h)$
- predecessor, successor:  $O(h)$
- inorder traversal:  $O(N)$  – the only one that does not depend on  $h$ 
  - PS: We also have preorder and postorder traversals for tree structure
- select/rank: ? (we have not discuss this yet)

# More BST Attributes: Height and Size

---

Two more attributes at each BST vertex: Height and Size

Height: #edges on the path from this vertex to deepest leaf

Size: #vertices of the subtree rooted at this vertex

These values are recursively defined/computed:

$x.\text{height} = -1$  (if  $x$  is an empty tree)

$x.\text{height} = \max(x.\text{left}.\text{height}, x.\text{right}.\text{height}) + 1$  (all other cases)

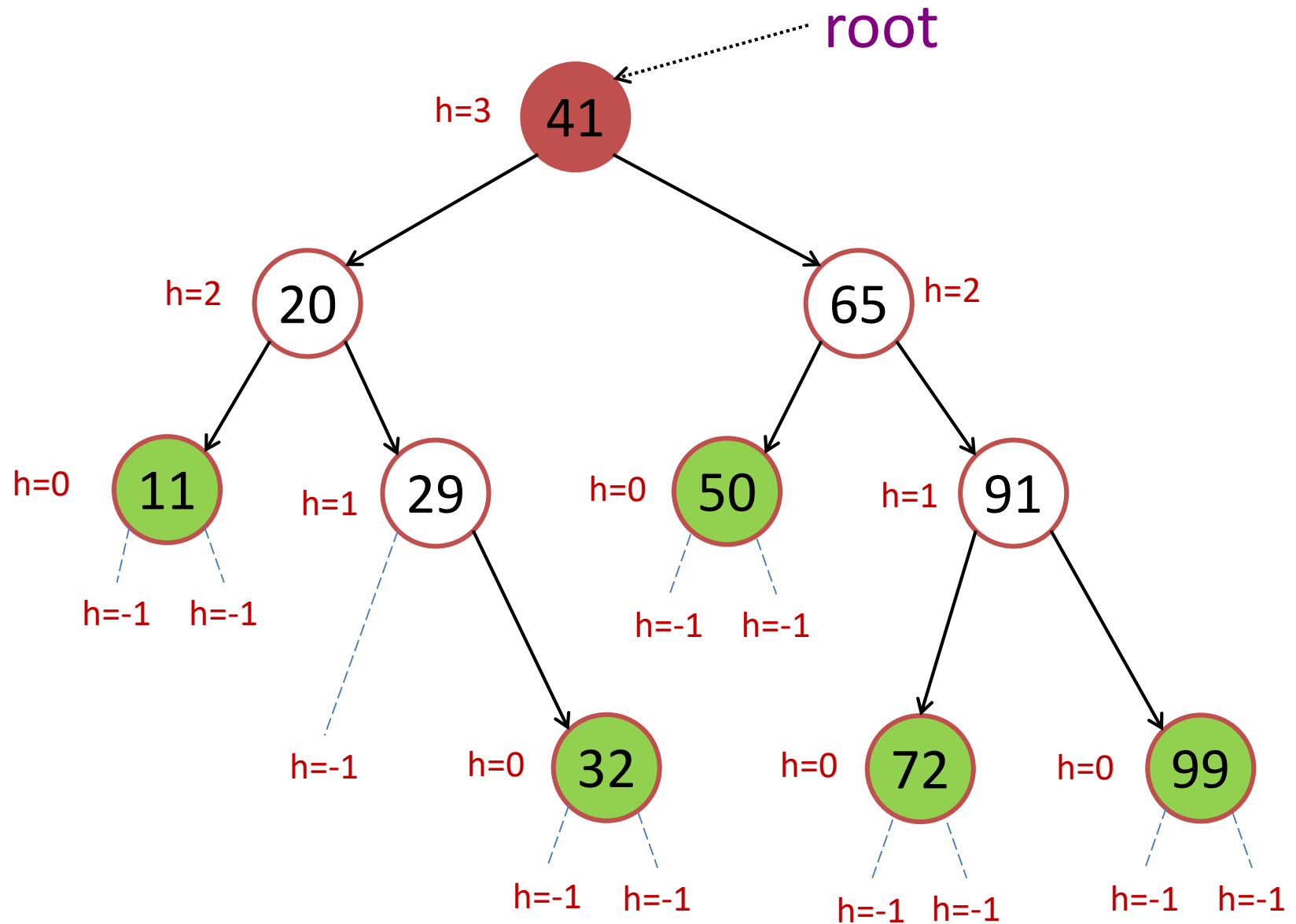
$x.\text{size} = 0$  (if  $x$  is an empty tree)

$x.\text{size} = x.\text{left}.\text{size} + x.\text{right}.\text{size} + 1$  (all other cases)

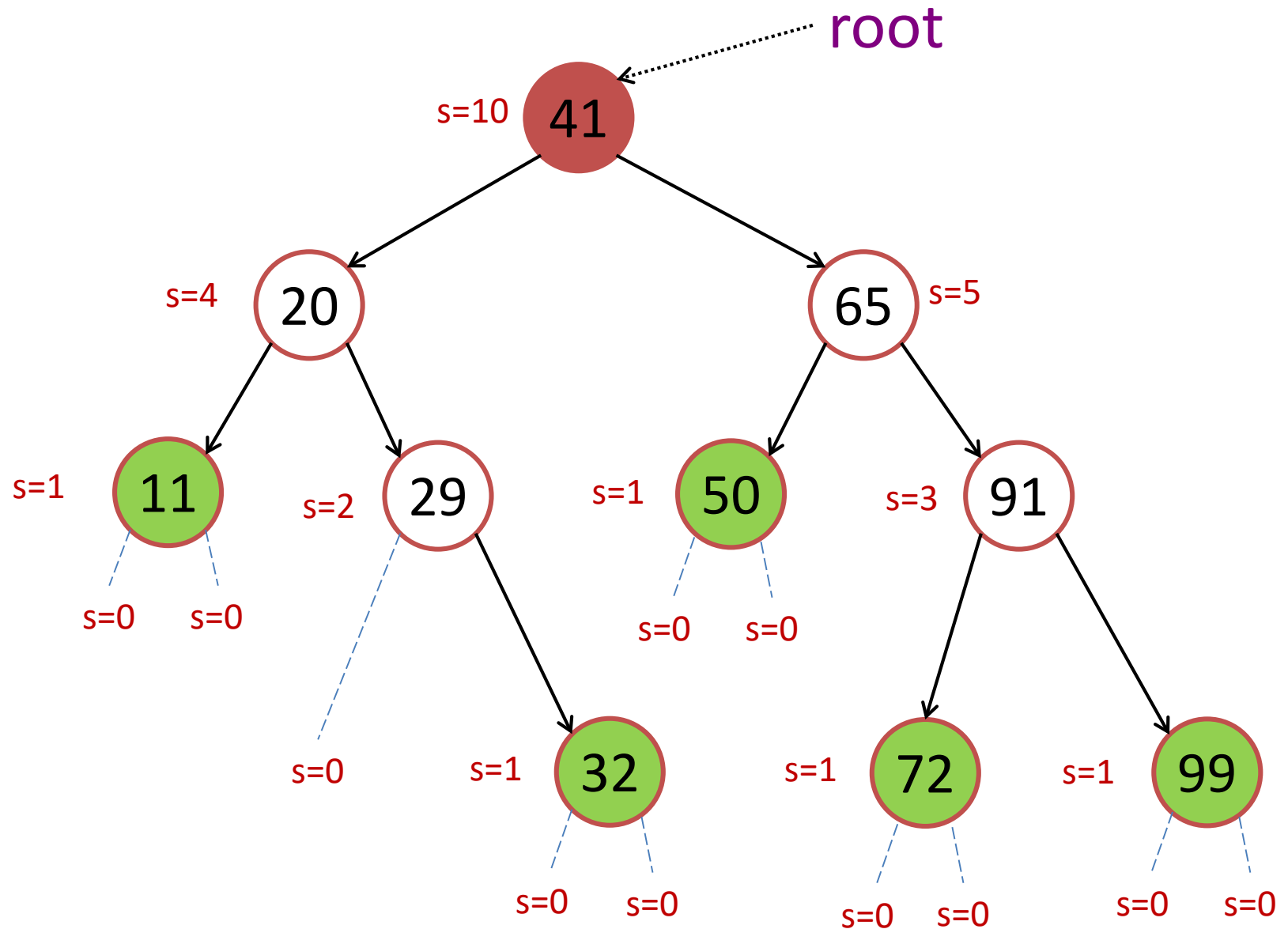
The height of the BST is thus:  $\text{root}.\text{height}$

The size of the BST is thus:  $\text{root}.\text{size}$

# Binary Search Trees: Height (h)

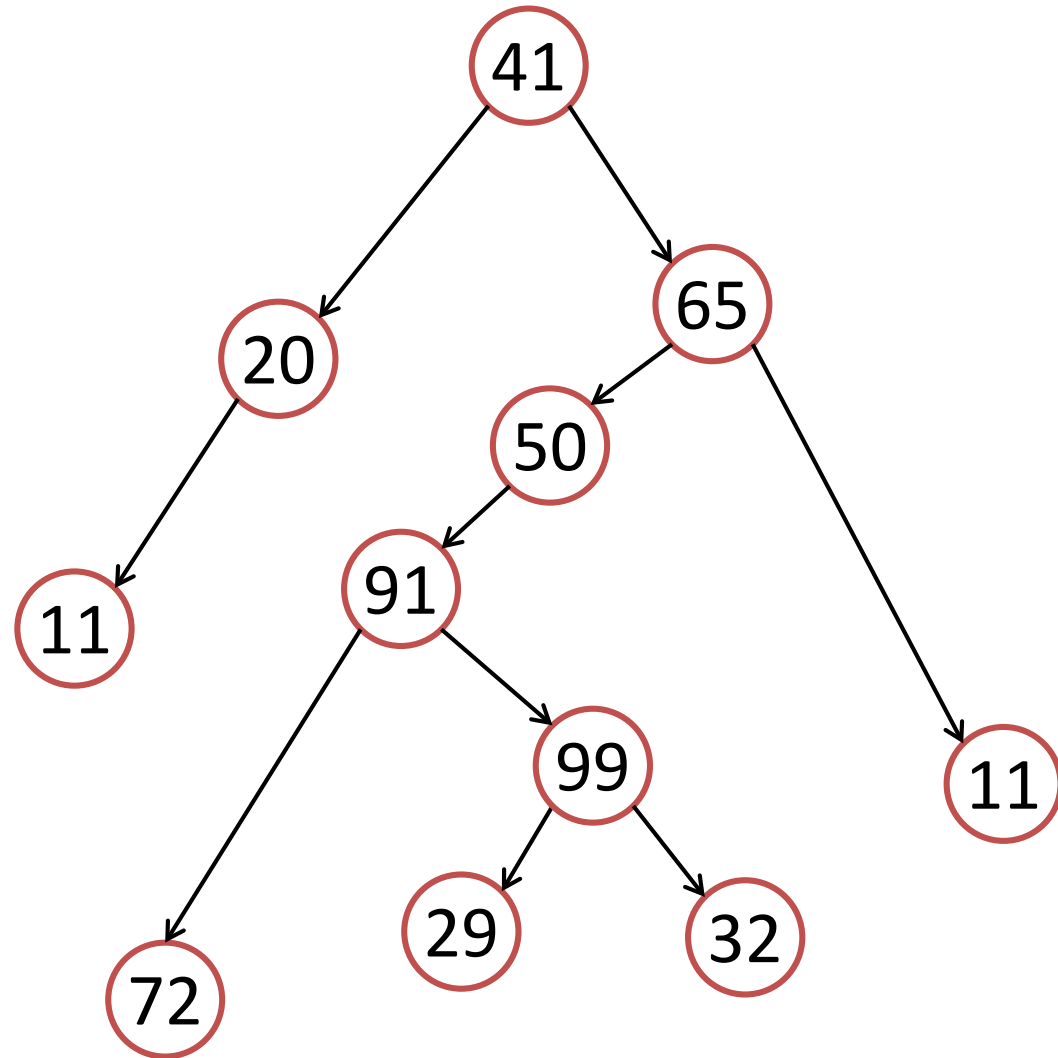


# Binary Search Trees: Size (s)



# The height of this tree is?

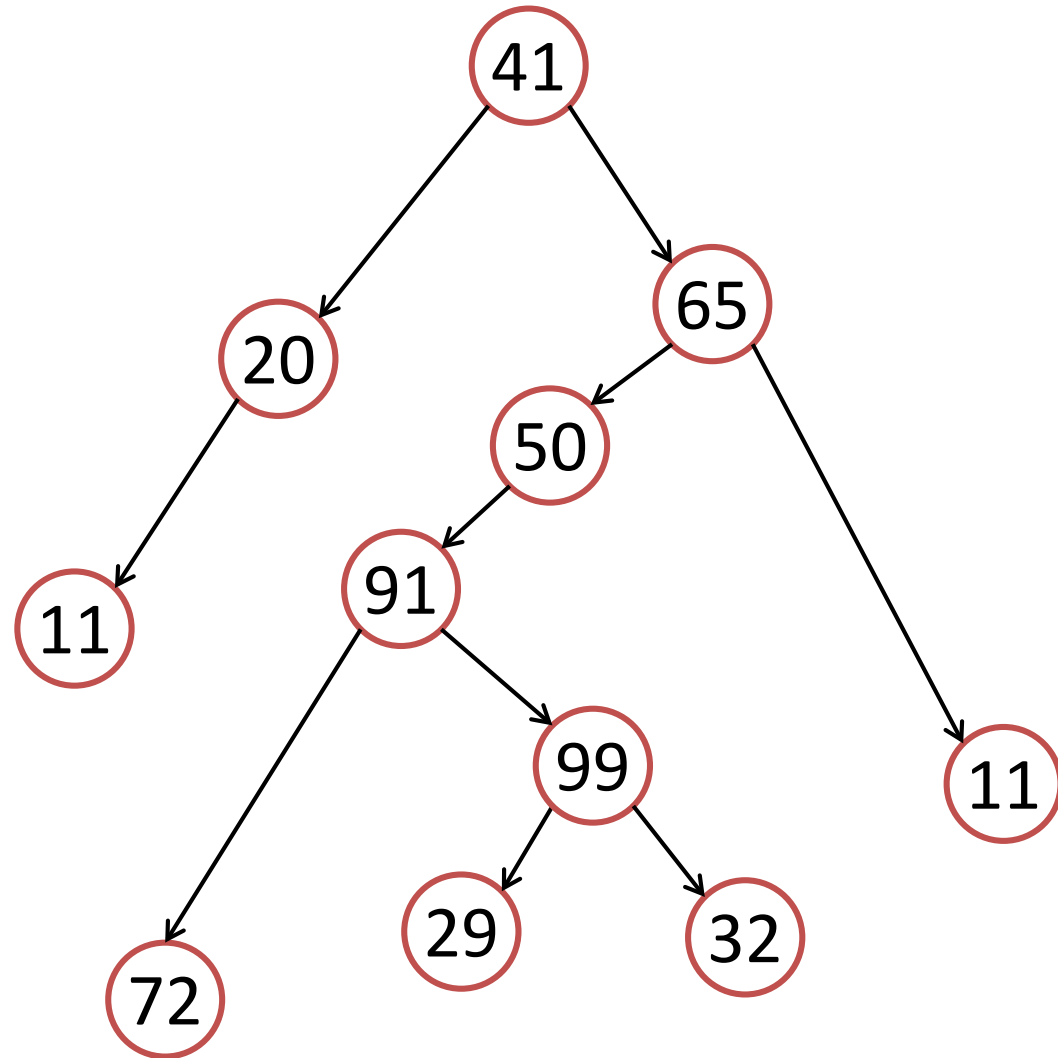
1. 2
2. 4
3. 5
4. 6
5. 7
6. 42





# The size of this tree is?

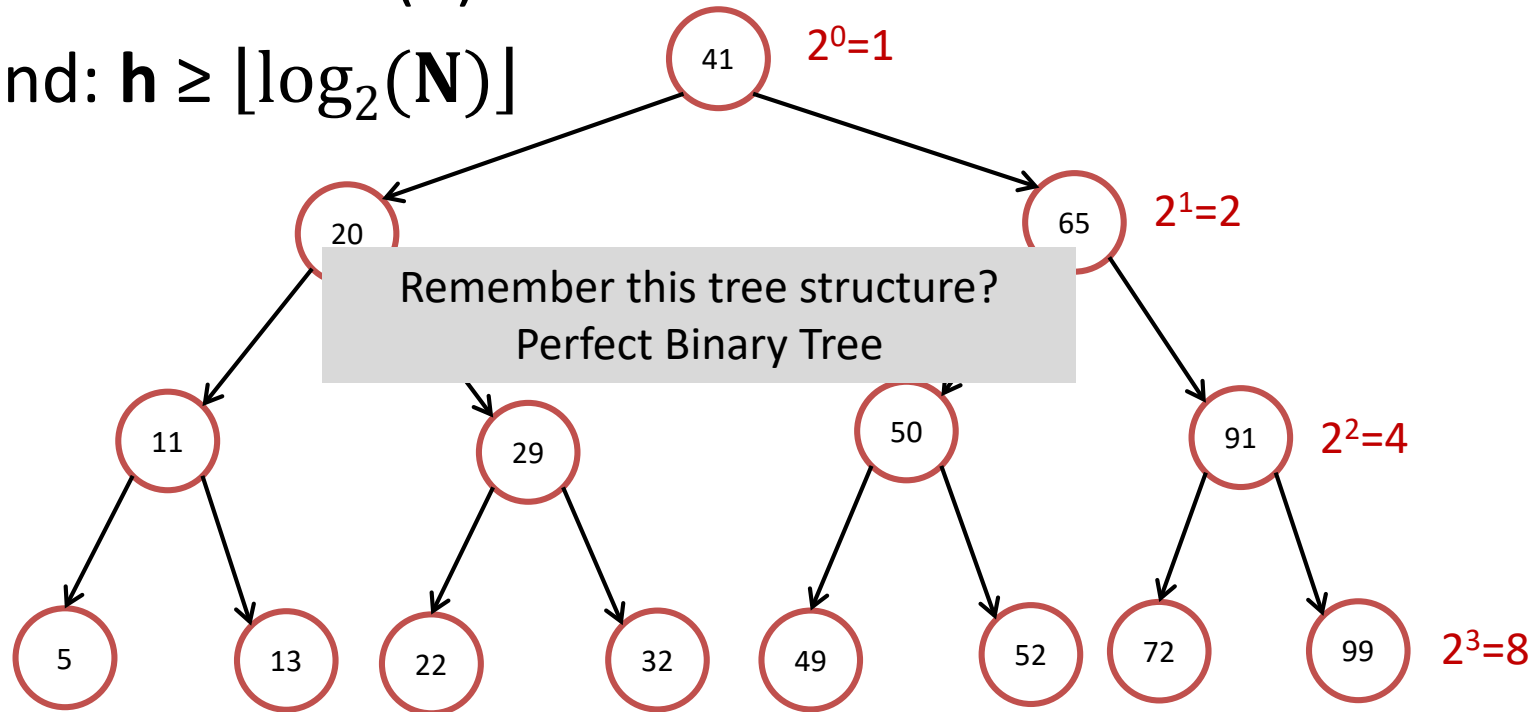
1. 10
2. 11
3. 12
4. 13
5. 14
6. 15



# The Importance of Being Balanced

Most operations take  $O(h)$  time

Lower bound:  $h \geq \lfloor \log_2(N) \rfloor$



$$N = 1 + 2 + 4 + \dots + 2^h = 2^0 + 2^1 + 2^2 + \dots + 2^h$$

$$= 2^{h+1} - 1 < 2^{h+1} \text{ (sum of geometric progression)}$$

$$\log_2(N) < \log_2(2^{h+1}) \Rightarrow \log_2(N) < (h+1) * \log_2(2) \Rightarrow h > \log_2(N) - 1$$

$$\Rightarrow h \geq \lfloor \log_2(N) \rfloor$$

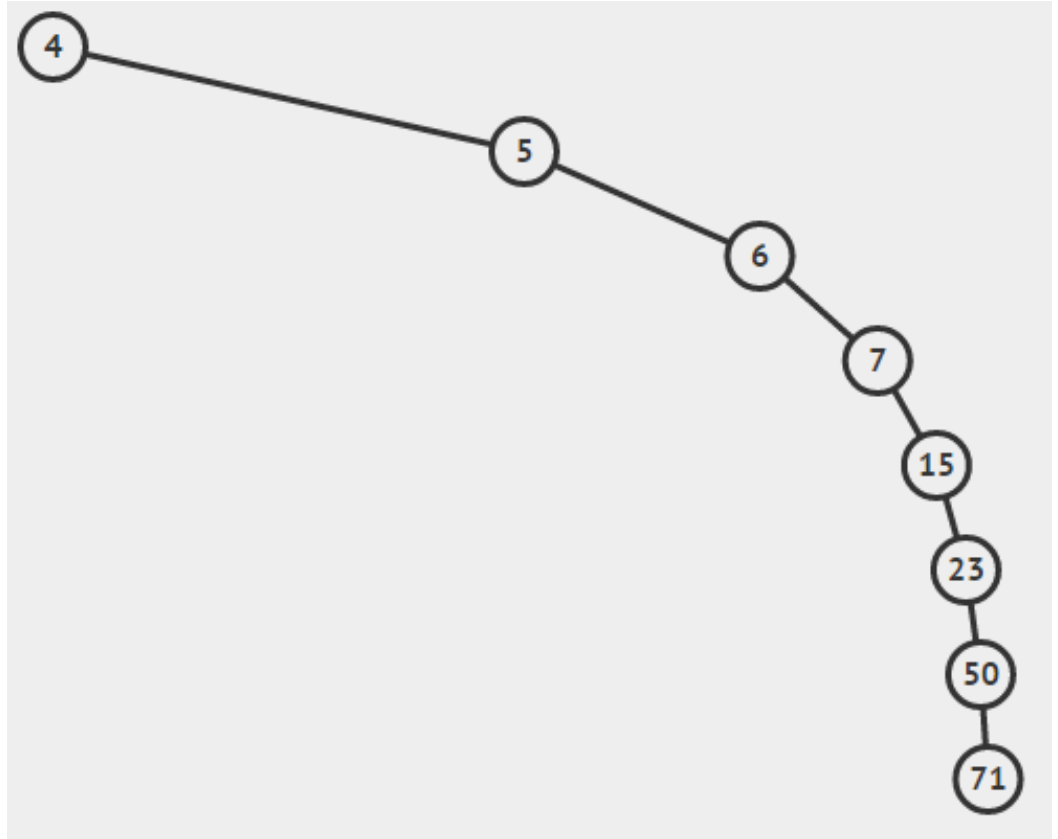
# The Importance of Being Balanced

---

Most operations take  $O(h)$  time

Upper bound:  $h \leq N-1 \rightarrow h < N$

Remember this tree structure?  
The worst case for BST...



# The Importance of Being Balanced

Most operations take  $O(h)$  time

Combined bound:  $\lfloor \log_2(N) \rfloor \leq h < N$

$\log_2(N)$  versus  $N$  in picture (revisited with larger numbers):

$N = 500$



Without 2<sup>nd</sup> half of CS2040

$\log_2(N) \sim 9$



After learning CS2040 😊

Without 2<sup>nd</sup> half of CS2040

$N = 1\,000$



$\log_2(N) \sim 10$



After learning CS2040 😊

We say a BST is balanced if  $h = O(\log N)$ , i.e.  $\underline{c} * \log N$

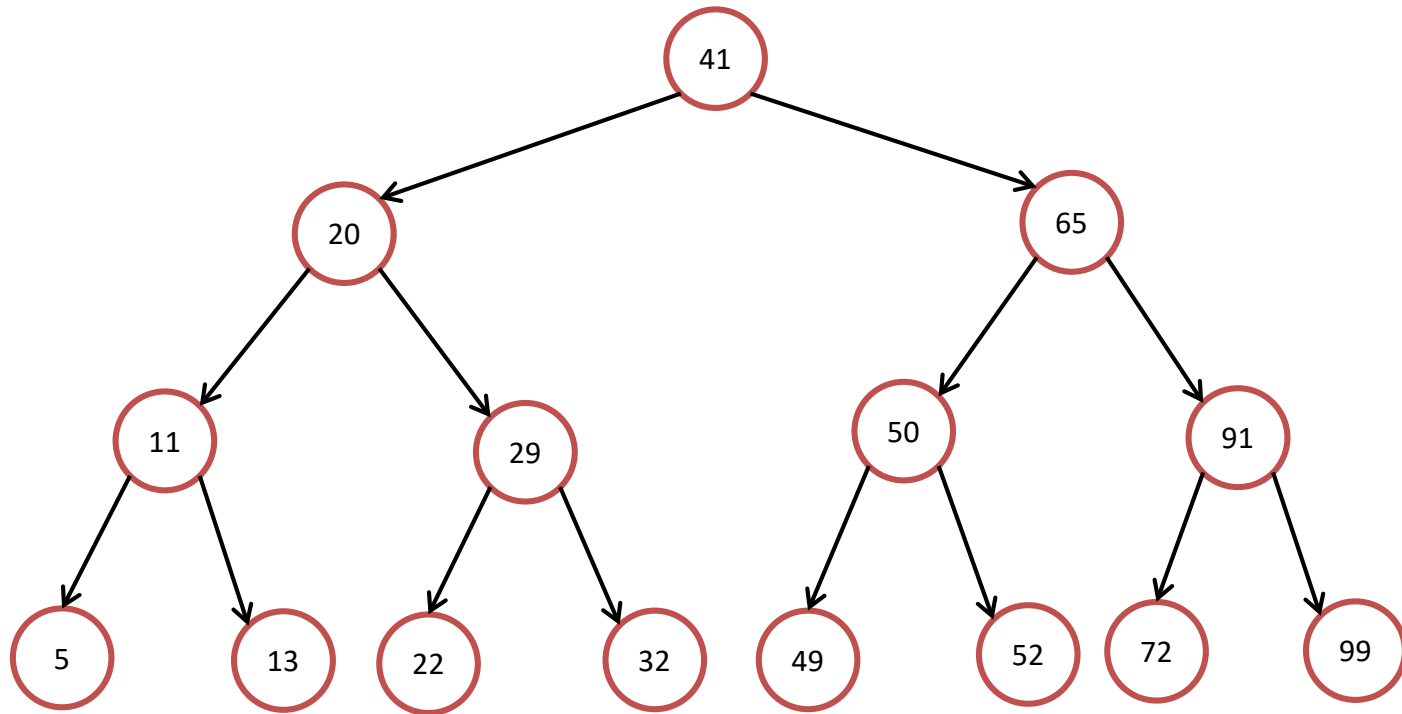
On a balanced BST, all operations run in  $O(\log N)$  time

# The Importance of Being Balanced

---

Example of a perfectly balanced BST:

This is hard to achieve though...



# The Importance of Being Balanced

---

How to get a balanced tree:

- Define a good property of a tree
- Show that if the good property holds, then the tree is **balanced**
- After every insert/delete, make sure the good property still holds
  - If not, fix it!

Adelson-Velskii & Landis, 1962 (~57 years ago... :O)

Can be a little bit frustrating if you are not comfortable with recursion  
Hang on...

# **AVL TREES**

# AVL Trees [Adelson-Velskii & Landis 1962]

---

## Step 1: Augment (i.e. add more information)

In every vertex  $x$ , we also store its height:  **$x.height$**

(Note that  $x$  already has:  **$x.left$** ,  **$x.right$** ,  **$x.parent$** , and  **$x.key$** )

During insertion and deletion, we *also* update **height**:

```
insert(x, v)
```

```
    // ... same as before ...
```

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

```
// update height during deletion too (same as above)
```

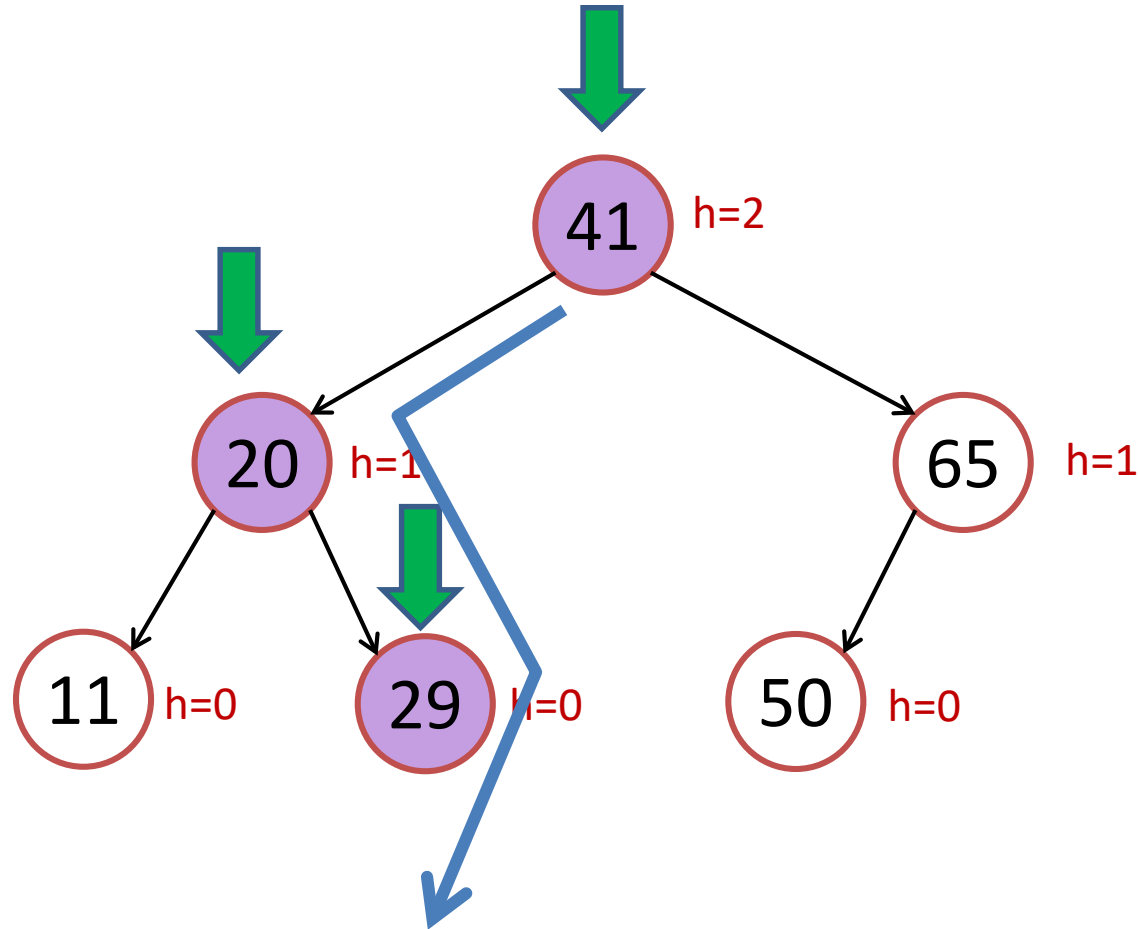
```
// update on attribute size can be done similarly
```



# Binary Search Trees

Height of empty trees are ignored in this illustration (all -1)

insert(27)

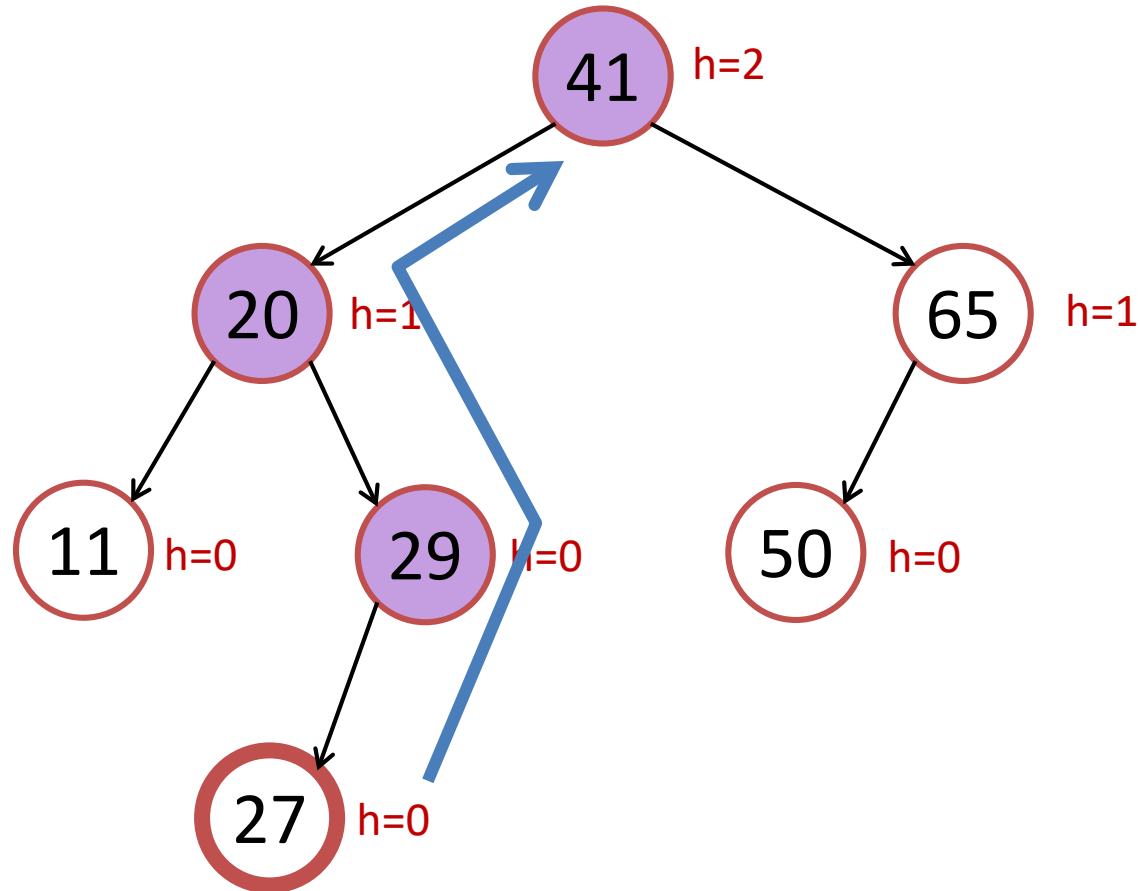


Height information during insertion/deletion is not shown in VisuAlgo (yet)

# Binary Search Trees

---

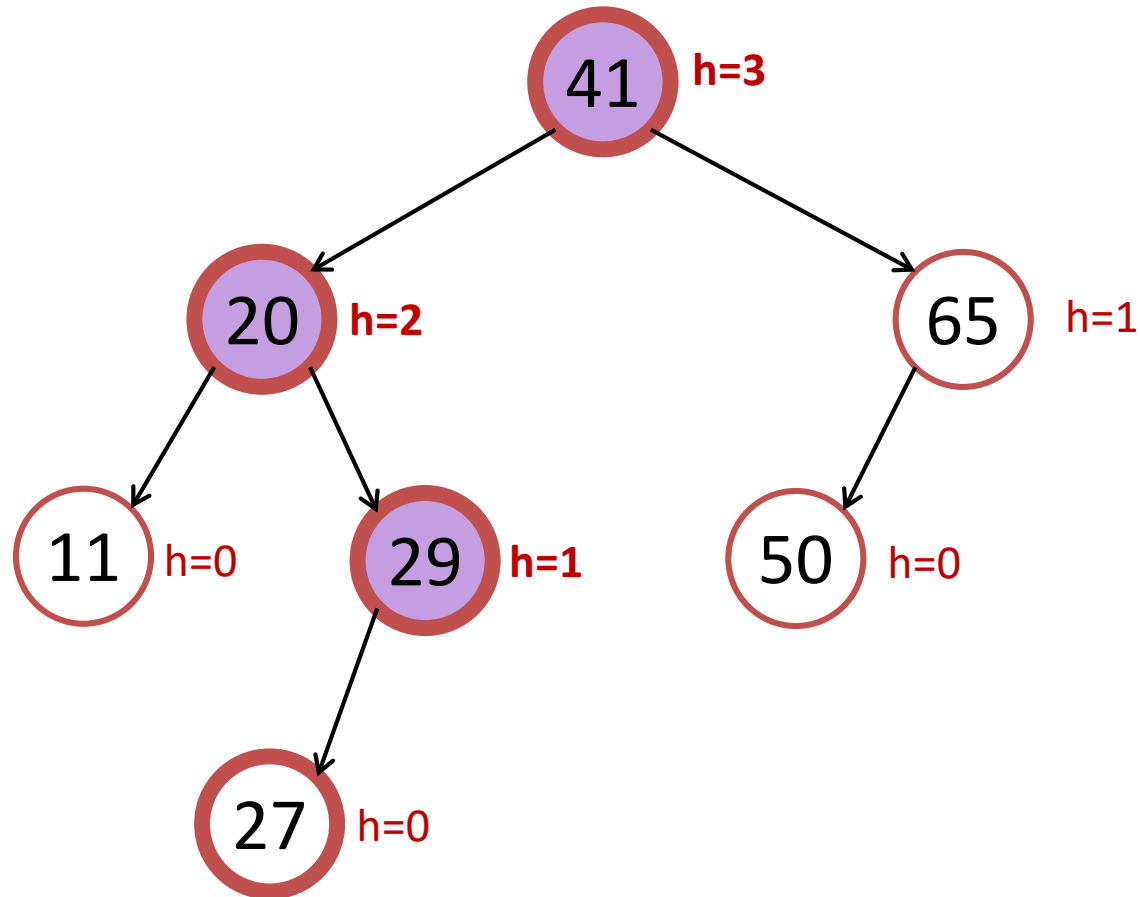
insert(27)



# Binary Search Trees

insert(27)

Notice that only vertices along the insertion path may have their height attribute updated...



# AVL Trees [Adelson-Velskii & Landis 1962]

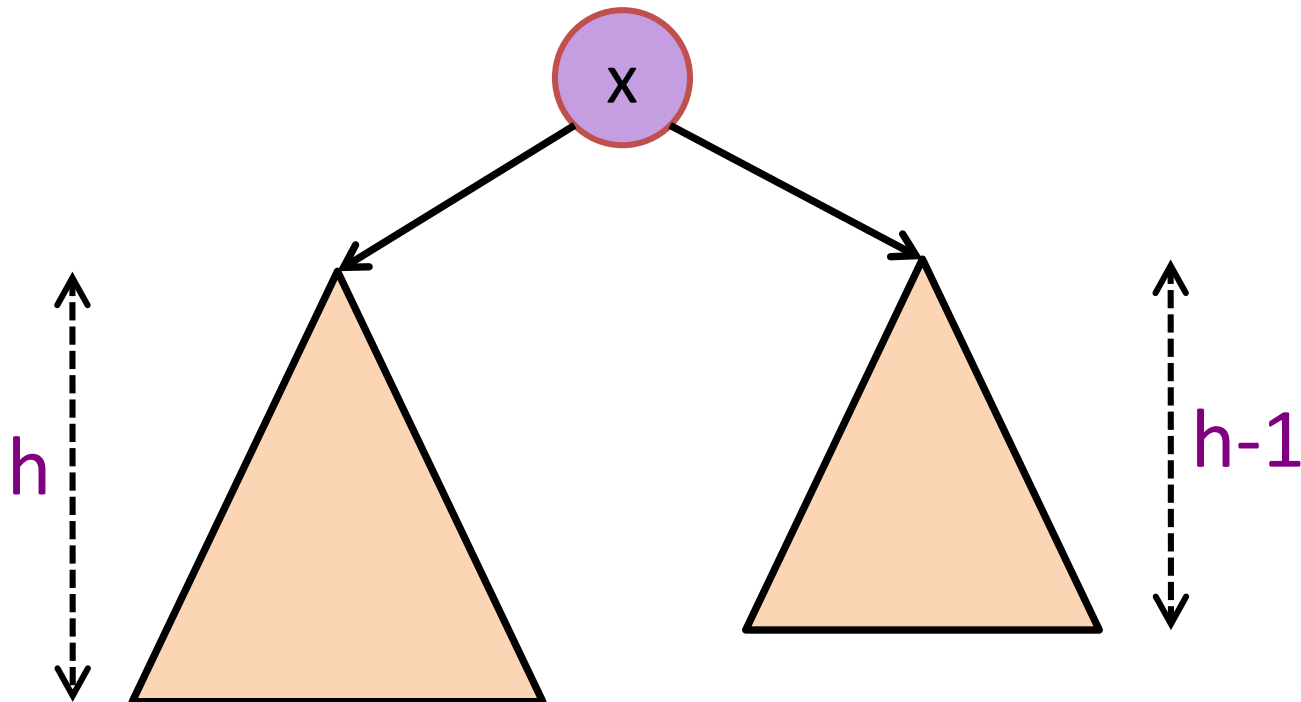
---

Step 2: Define Invariant (something that will not change)

A vertex  $x$  is said to be height-balanced if:

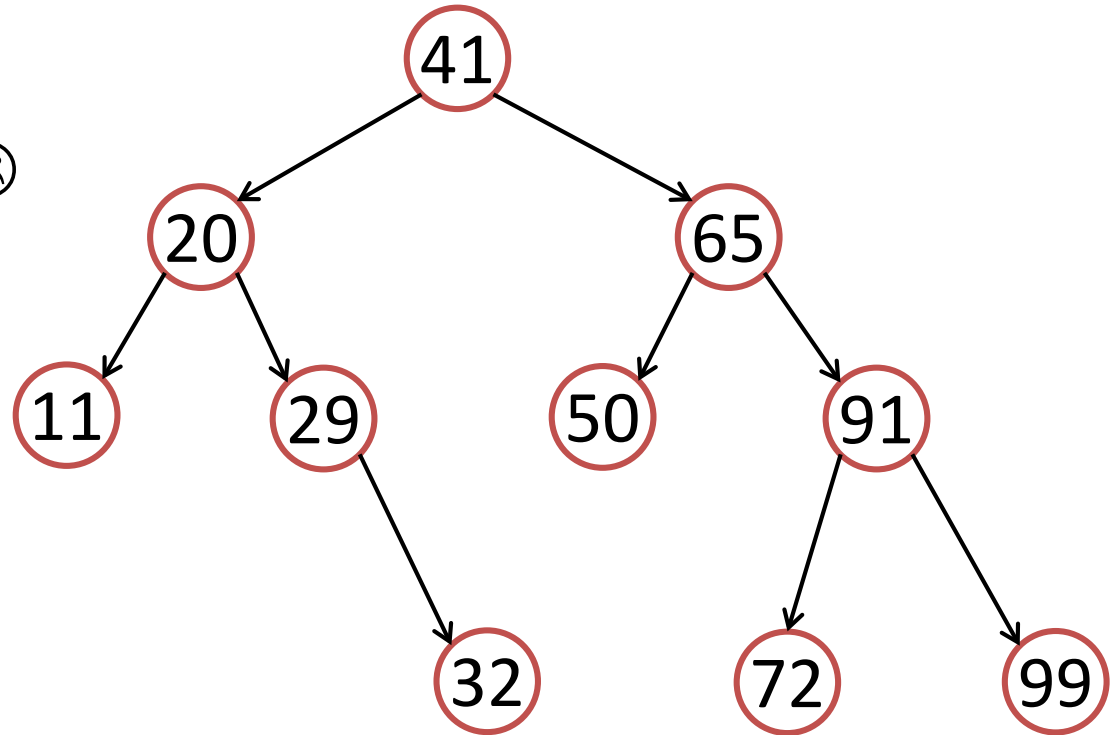
$$|x.\text{left.height} - x.\text{right.height}| \leq 1$$

A binary search tree is said to be height balanced if:  
*every vertex* in the tree is height-balanced



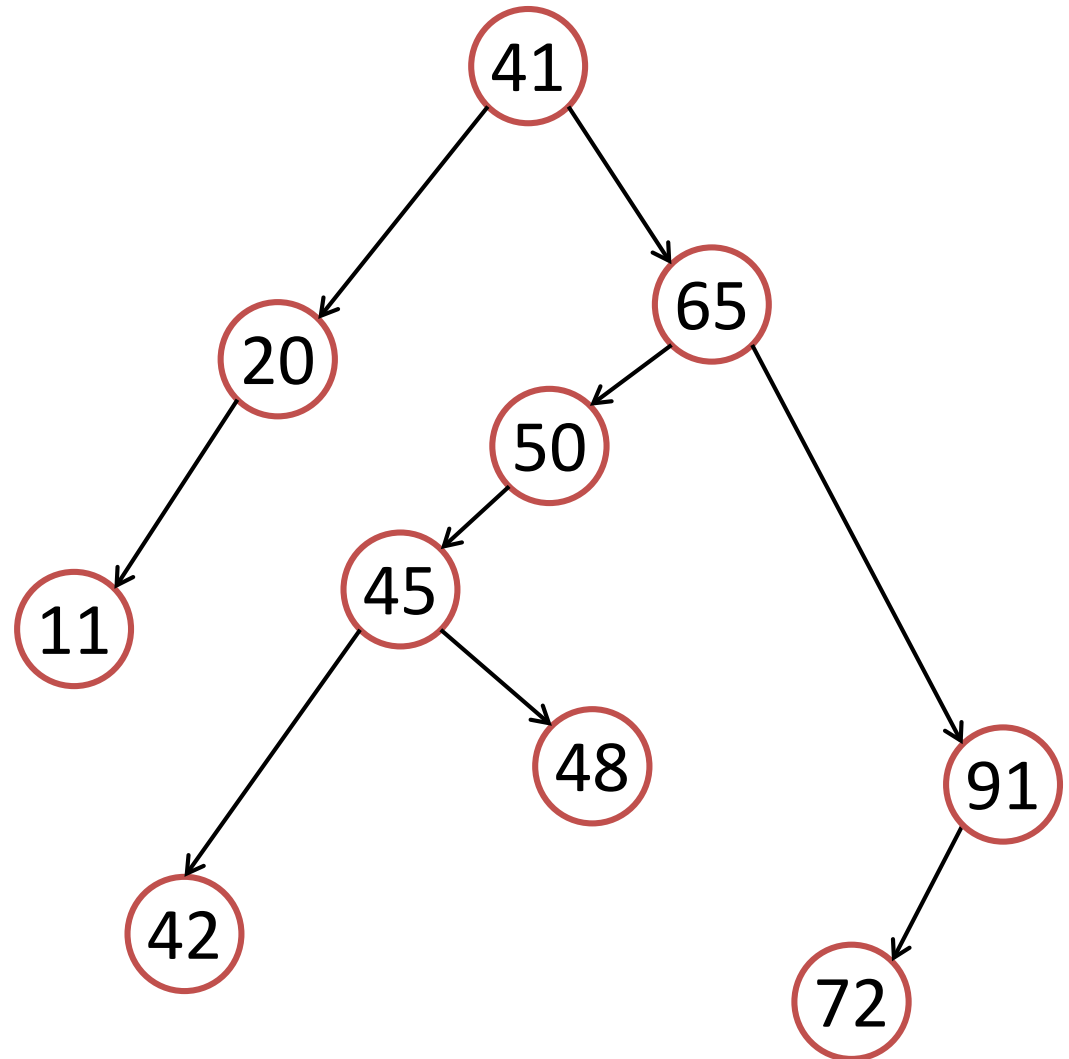
# Is this tree height-balanced according to AVL?

1. Yes
2. No
3. I am confused... 😞



# Is this tree height-balanced according to AVL?

1. Yes
2. No
3. I am confused... 😞



# Height-Balanced Trees

---

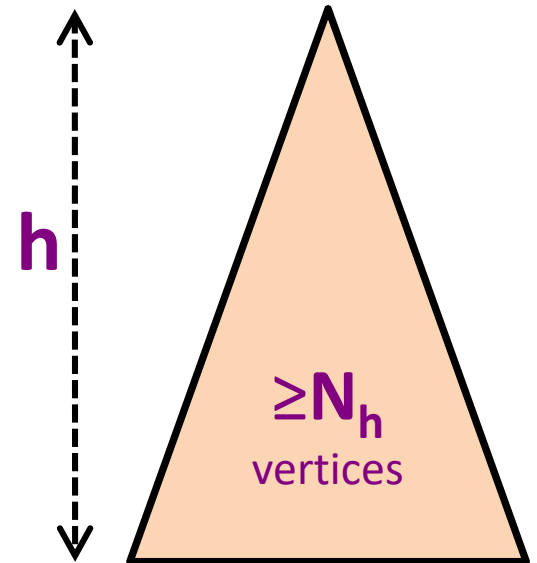
Claim:

A height-balanced tree with  $N$  vertices  
has height  $h < 2 * \log_2(N)$

Proof (do not be scared):

Let  $N_h$  be the minimum number of vertices  
in a height-balanced tree of height  $h$

The actual number of vertices  $N \geq N_h$



# Height-Balanced Trees

Proof:

Let  $N_h$  be the minimum number of vertices in a height-balanced tree of height  $h$

$$N_h = 1 + N_{h-1} + N_{h-2}$$

$$N_h > 1 + 2N_{h-2} \text{ (as } N_{h-1} > N_{h-2} \text{)}$$

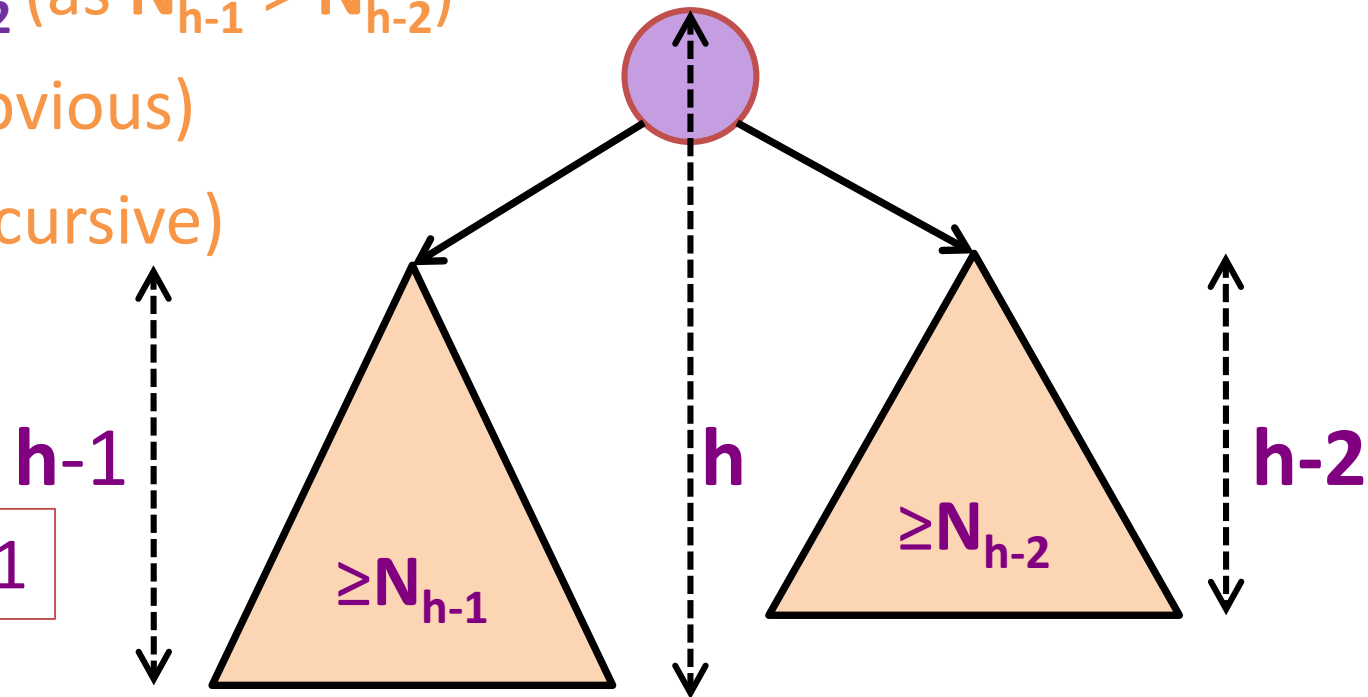
$$N_h > 2N_{h-2} \text{ (obvious)}$$

$$= 4N_{h-4} \text{ (recursive)}$$

$$= 8N_{h-6}$$

$$= \dots$$

Base case:  $N_0 = 1$





# Height-Balanced Trees

---

Proof:

Let  $N_h$  be the minimum number of vertices in a height-balanced tree of height  $h$

$$N_h = 1 + N_{h-1} + N_{h-2}$$

$$N_h > 1 + 2N_{h-2}$$

$$\begin{aligned} N_h &> 2N_{h-2} \\ &> 4N_{h-4} \\ &> 8N_{h-6} \\ &> \dots \end{aligned}$$

As each step we reduce  $h$  by 2,  
Then we need to do this step  $h/2$  times  
to reduce  $h$  (assume  $h$  is even) to 0

Base case:  $N_0 = 1$

$$N_h > 2^{h/2} N_0$$

$$N_h > 2^{h/2}$$

# Height-Balanced Trees

---

Claim:

A height-balanced tree is balanced,  
i.e. has height  $h = O(\log(N))$

We have shown that:  $N_h > 2^{h/2}$  and  $N \geq N_h$

$$N \geq N_h > 2^{h/2}$$

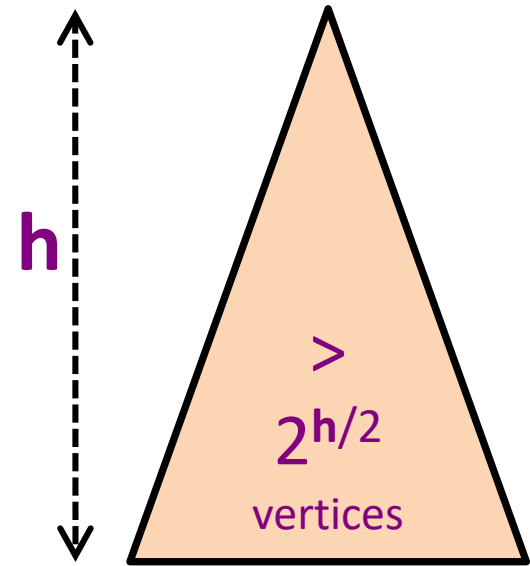
$$N > 2^{h/2}$$

$$\log_2(N) > \log_2(2^{h/2}) \text{ (}\log_2 \text{ on both side)}$$

$$\log_2(N) > h/2 \text{ (formula simplification)}$$

$$2 * \log_2(N) > h \text{ or } h < 2 * \log_2(N)$$

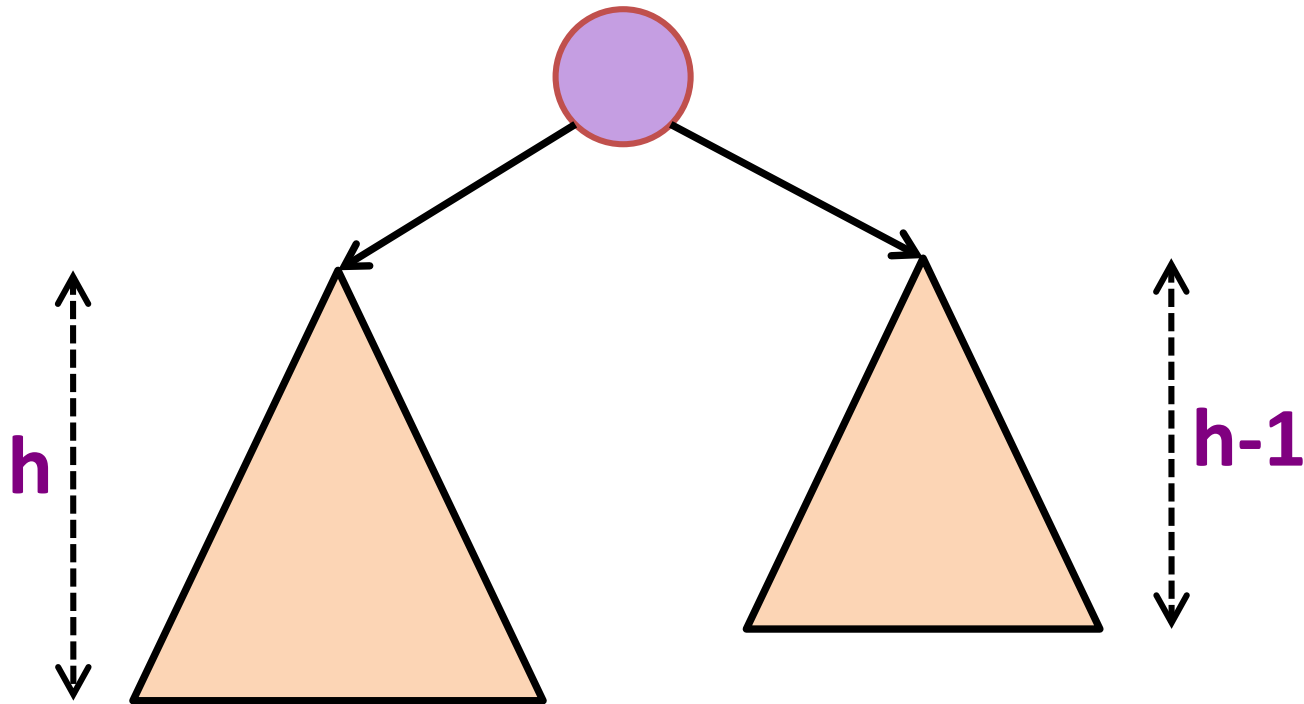
$$h = O(\log(N))$$



# AVL Trees [Adelson-Velskii & Landis 1962]

---

Step 3: Show how to maintain height-balance



# Insertion to an AVL Tree

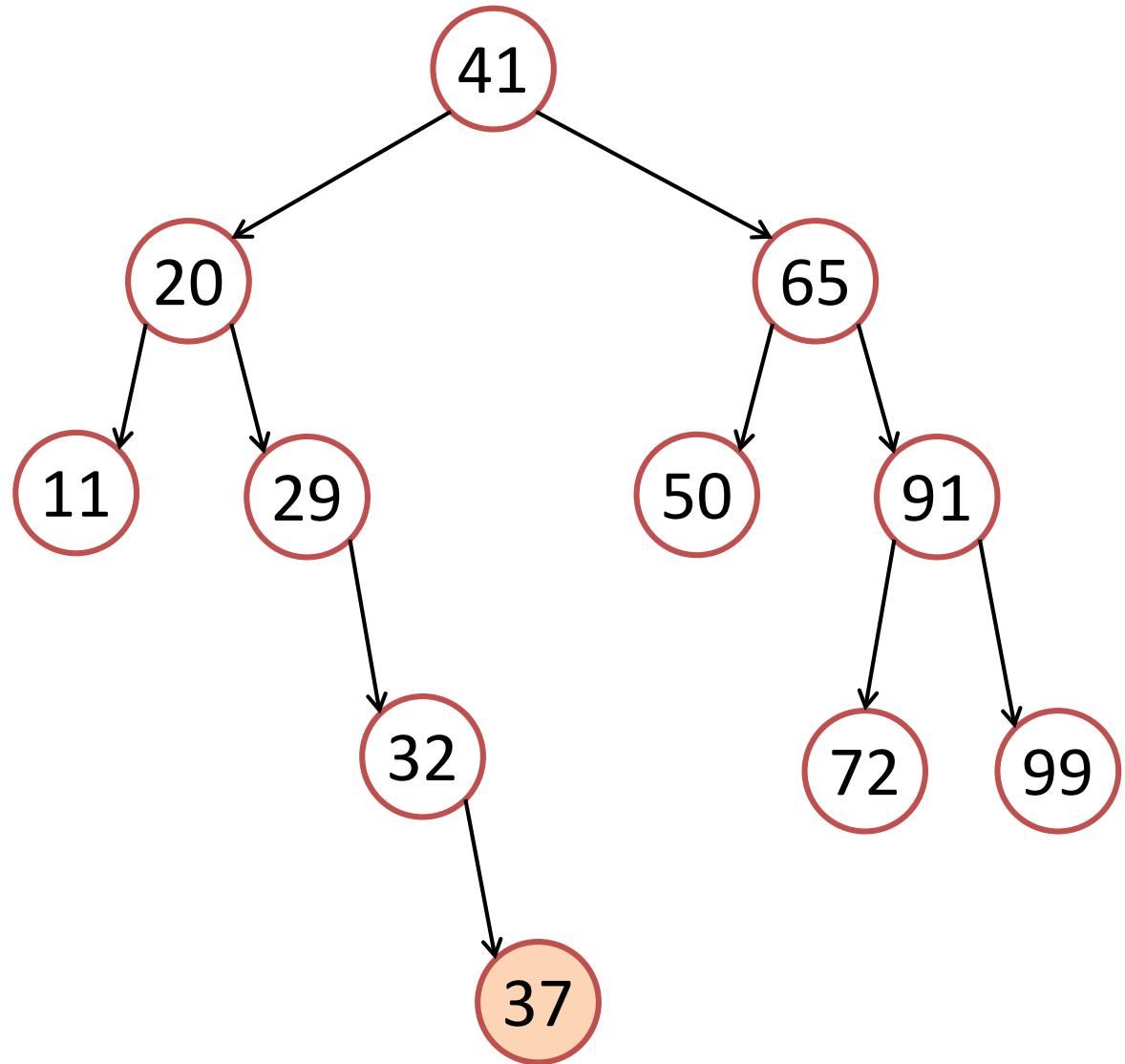
insert(37)

Initially balanced

But no longer  
balanced after  
Inserting 37

Need to rebalance!

But how?



“Infinite more” examples in VisuAlgo...

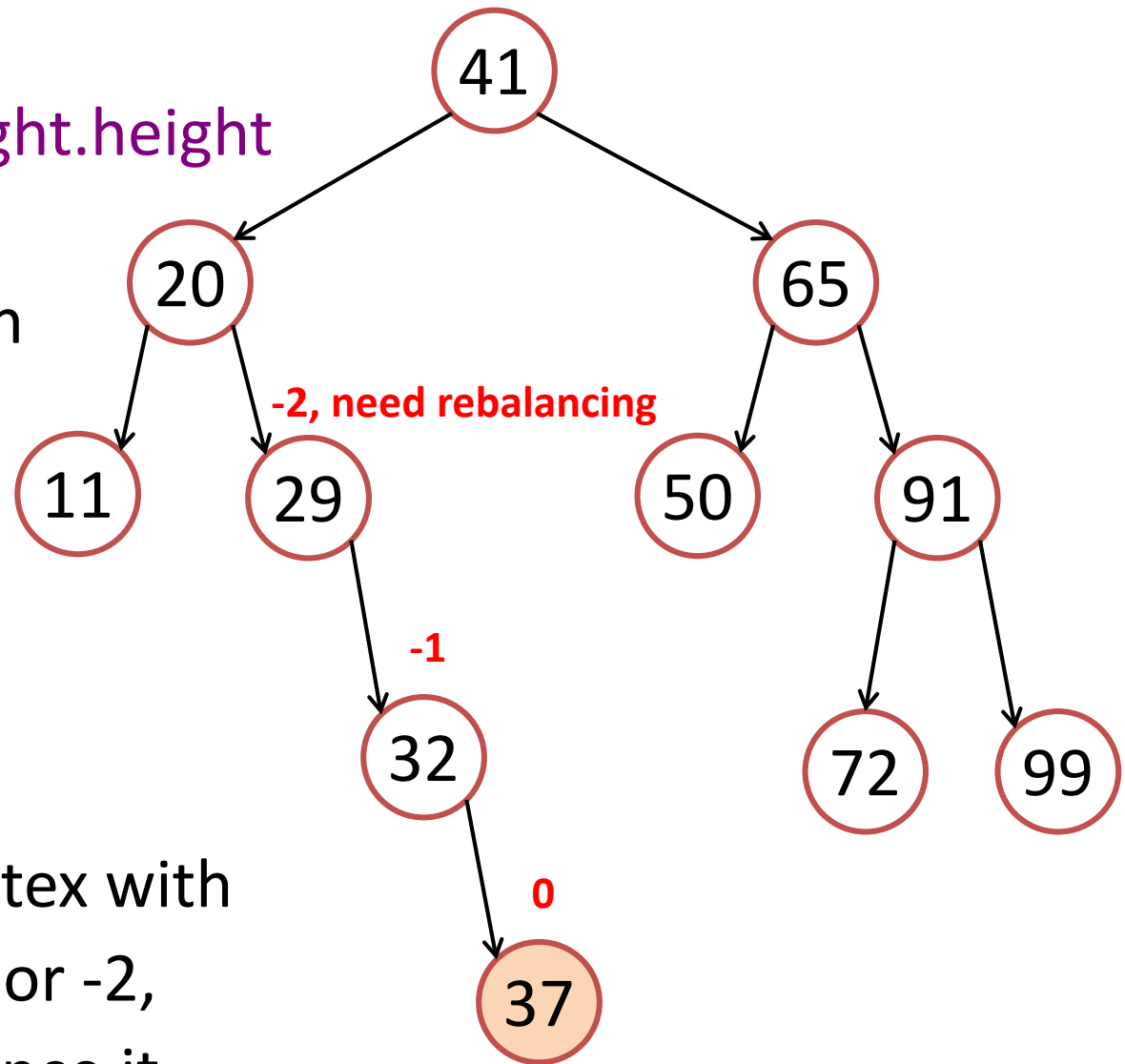
# Balance Factor (bf(x))

$bf(x) =$

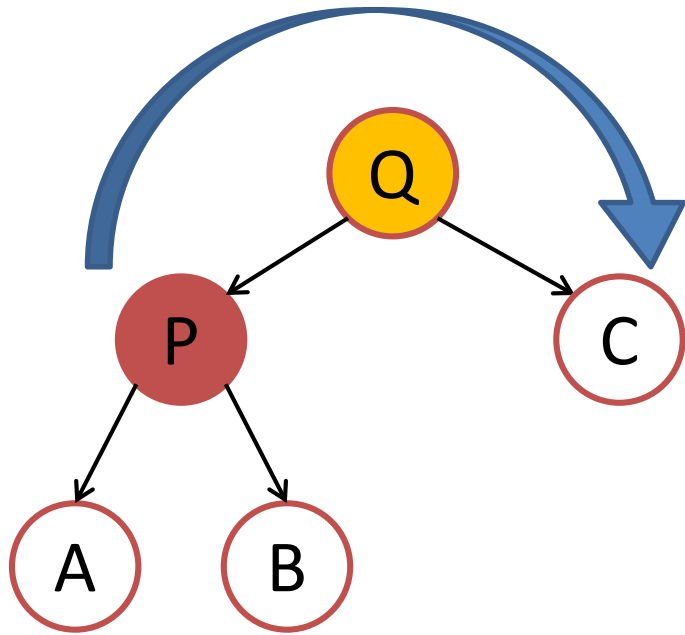
$x.left.height - x.right.height$

From the insertion point, check the balance factor of each vertex up to the root

Once we have vertex with balance factor +2 or -2, we have to rebalance it



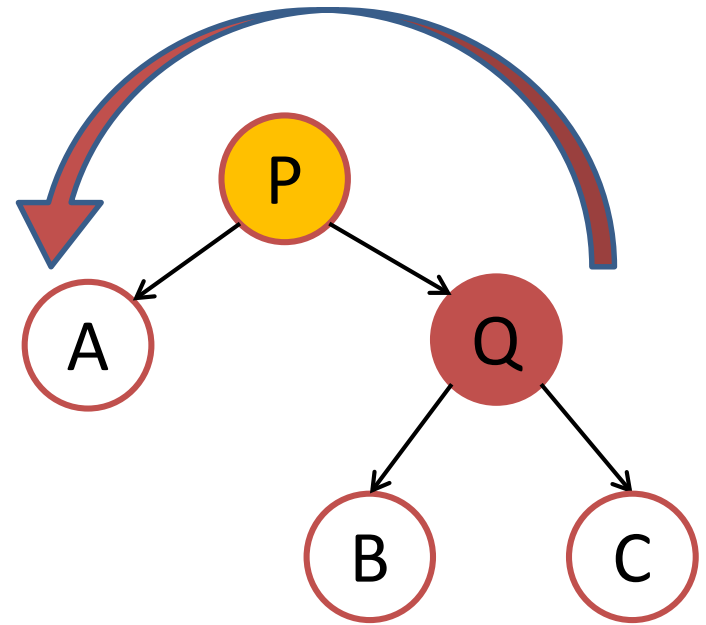
# Tree Rotations (1)



Right  
Rotation  
about Q



Left  
Rotation  
about P



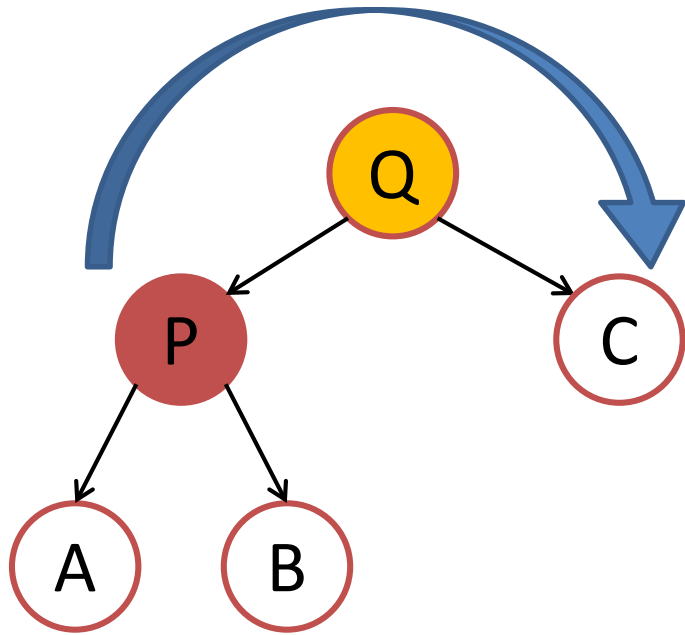
## Right Rotation

- Need Q to have a left child P
- Make Q right child of P
- Other manipulations ...


## Left Rotation

- Need P to have a right child Q
- Make P left child of Q
- Other manipulations ...

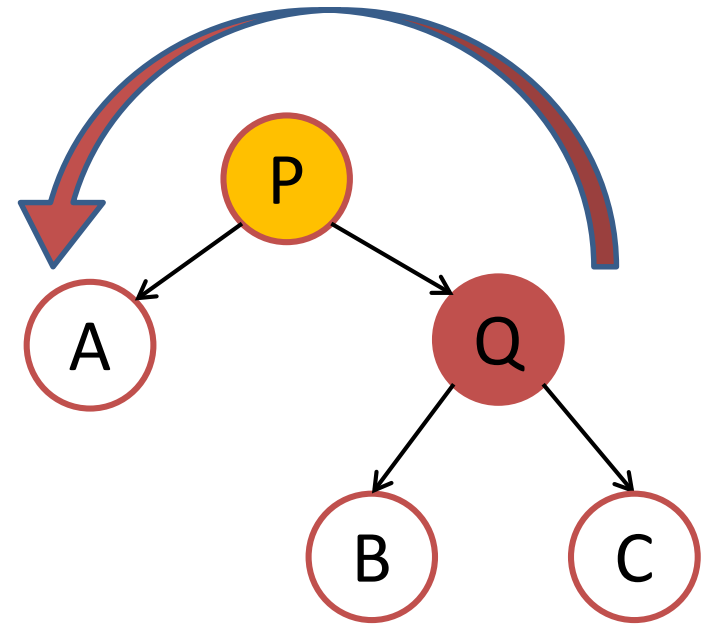

# Tree Rotations (2)



Right  
Rotation



Left  
Rotation



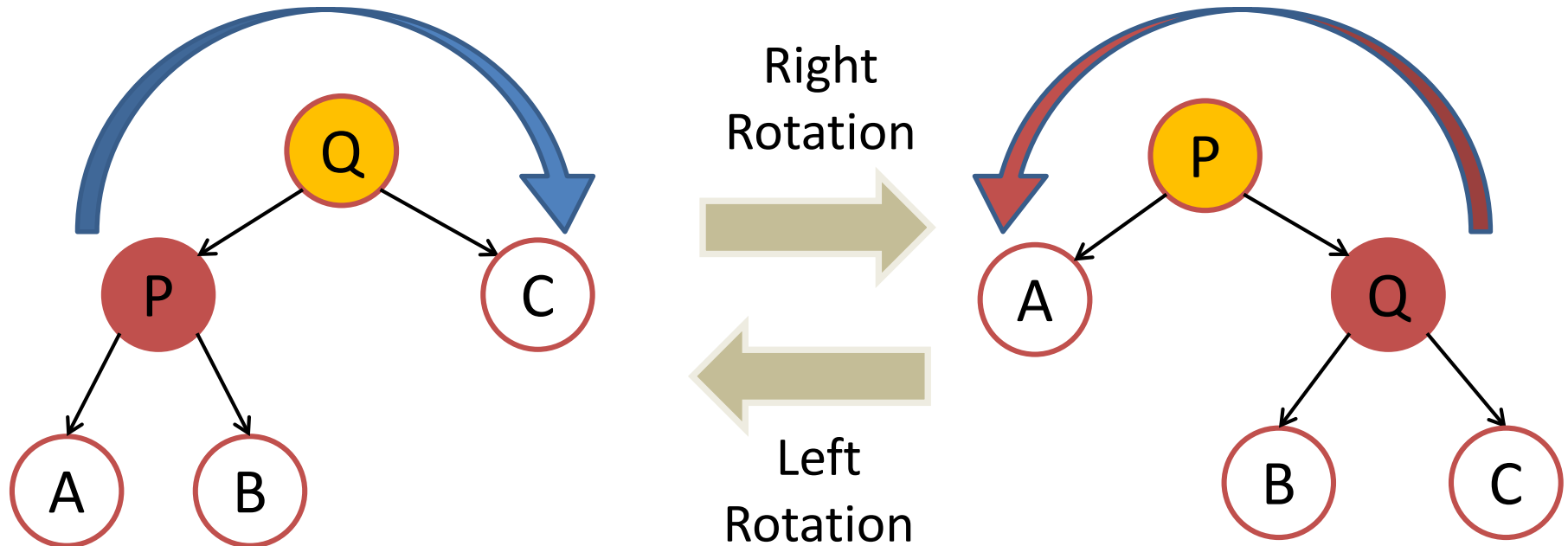
## Right Rotation

- Need Q to have a left child P
- Make Q right child of P
- **Make B (right child of P) left child of Q**

## Left Rotation

- Need P to have a right child Q
- Make P left child of Q
- **Make B (left child of Q) right child of P**

# Tree Rotations (3)



Rotations maintain ordering of keys

$\Rightarrow$  Maintains BST property (*see vertex B where  $P \leq B \leq Q$* )



# Tree Rotations Pseudo Code $\rightarrow O(1)$

```
BSTVertex rotateLeft(BSTVertex T) // pre-req: T.right != null
```

```
    BSTVertex w = T.right
```

```
    w.parent = T.parent
```

```
    T.parent = w
```

```
    T.right = w.left
```

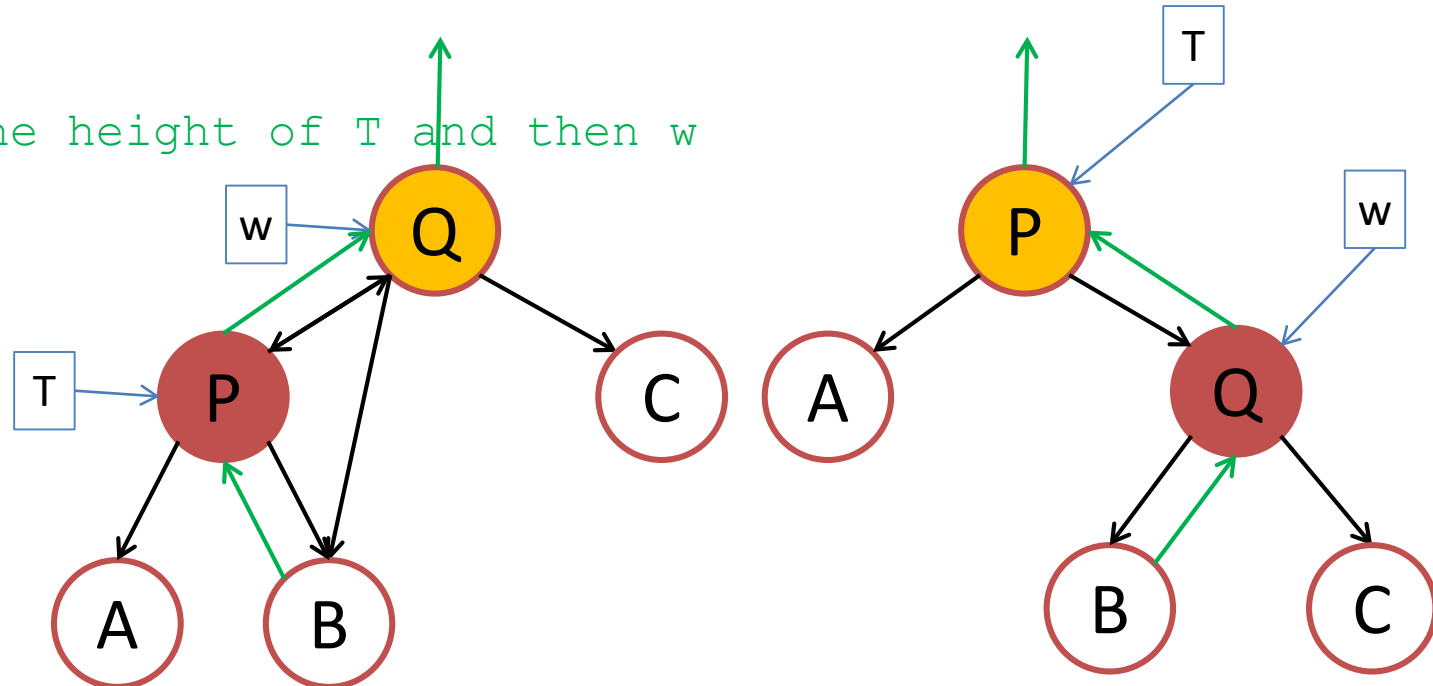
```
    if (w.left != null) w.left.parent = T
```

```
    w.left = T
```

```
    // Update the height of T and then w
```

```
    return w
```

rotateRight is the mirrored version of this pseudocode



This slide is  
can be  
confusing  
without the  
animation

# Four Possible Cases

$bf(x) = +2$  and  $0 \leq bf(x.left) \leq 1$

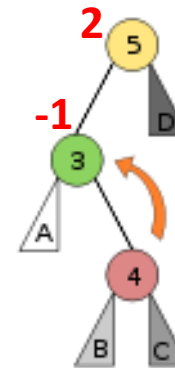
rightRotate(x)

$bf(x) = +2$  and  $bf(x.left) = -1$

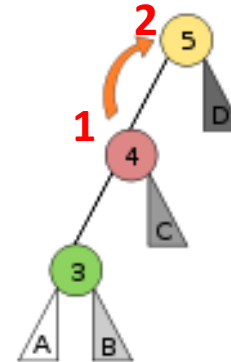
leftRotate(x.left)

rightRotate(x)

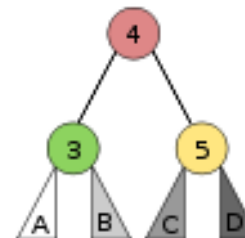
Left Right Case



Left Left Case



Balanced



# Four Possible Cases

$bf(x) = -2$  and  $-1 \leq bf(x.right) \leq 0$

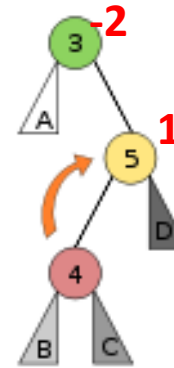
leftRotate(x)

$bf(x) = -2$  and  $bf(x.right) = 1$

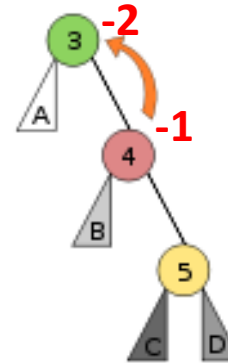
rightRotate(x.right)

leftRotate(x)

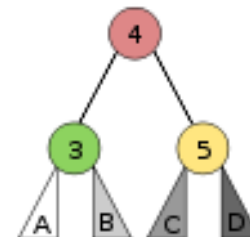
Right Left Case



Right Right Case



Balanced

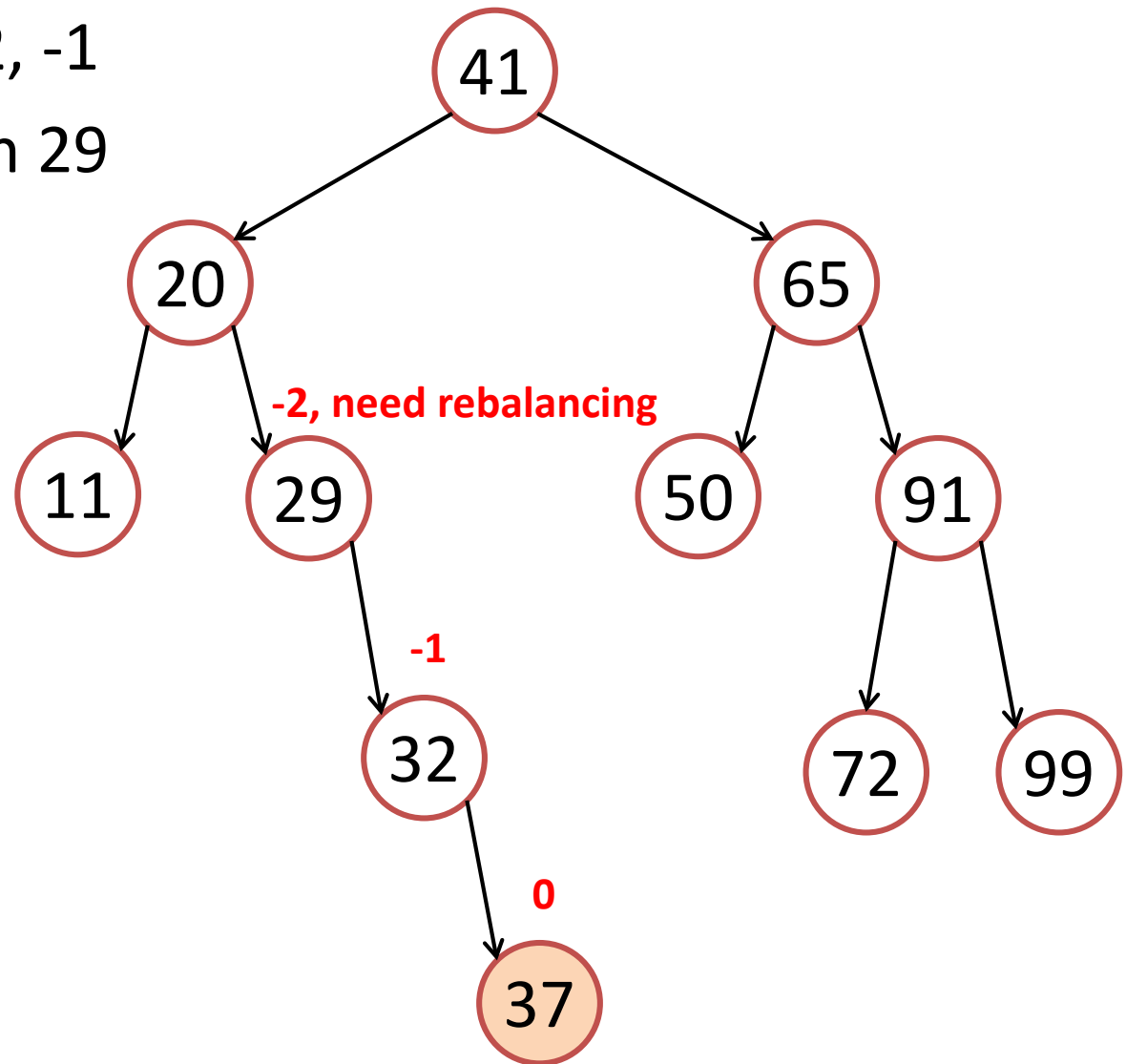


# Rebalancing (1)

---

This is a case of -2, -1

Do left rotate on 29

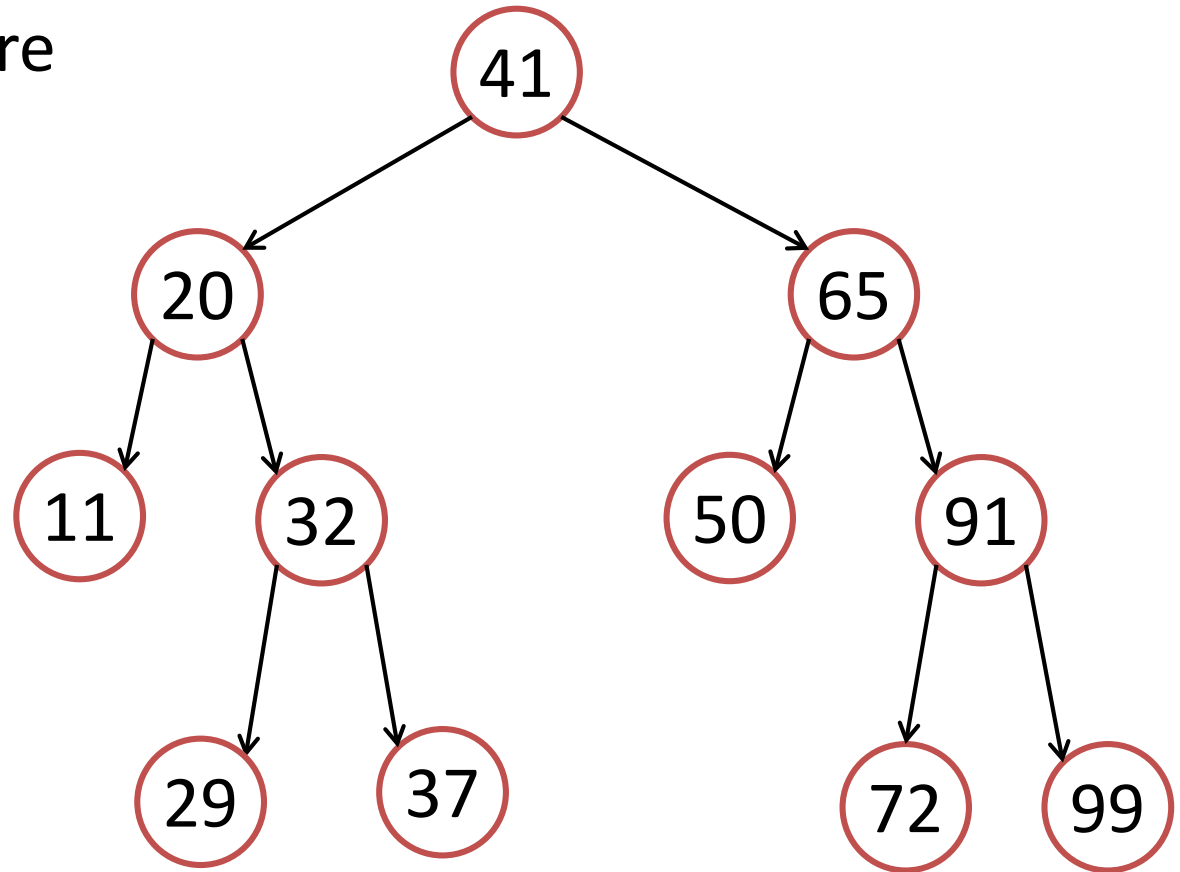


“Infinite more” examples in VisuAlgo...

# Rebalancing (2)

---

Now all vertices are  
balanced again



“Infinite more” examples in [VisuAlgo AVL Tree Visualization](#)

# Insertion to an AVL Tree

---

## Summary:

- Just insert the key as in normal BST
- Walk up the AVL tree from the insertion point to root:
  - At every step, update height & check balance factor
  - If a certain vertex is out-of-balance (+2 or -2), use rotations to rebalance
    - During insertion to an AVL tree, you can only trigger one of the four possible rebalancing cases as shown earlier once !

# Deletion from an AVL Tree

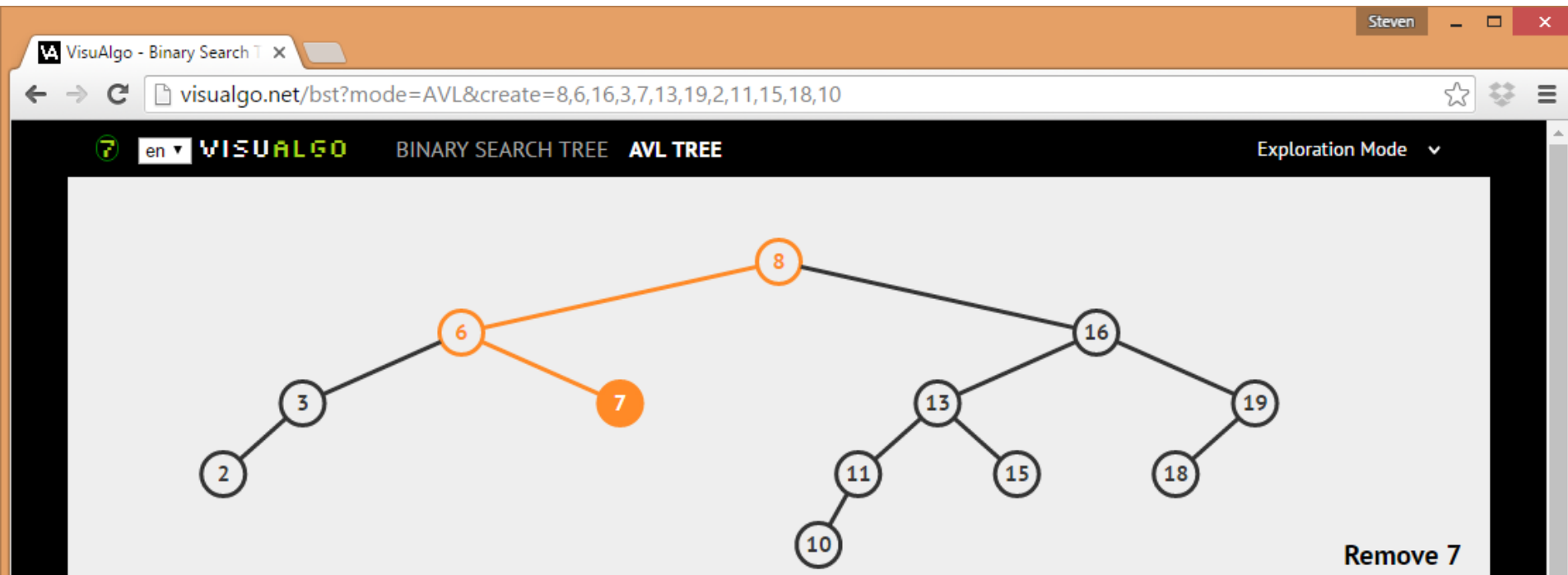
---

Deletion is quite similar to Insertion:

- Just delete the key as in normal BST
- Walk up the AVL tree from the deletion point to root:
  - At every step, update height & check balance factor
  - If a certain vertex is out-of-balance (+2 or -2), use rotations to rebalance
    - The main difference compared to insertion into AVL tree is that you may trigger one of the four possible rebalancing cases several times, up to  $h = \log n$  times :O, see this example (next slide)

# AVL Tree Web-based Review

Create an AVL Tree using 8,6,16,3,7,13,19,2,11,15,18,10



Try **Remove (Delete)** vertex 7, it triggers **two (more than one)** rebalancing actions

Then try various **Insert operations** and notice that at most it will only trigger one (out of the four cases) of rebalancing actions



# Balanced Search Trees

---

Many different flavors of balanced search trees

- AVL trees (Adelson-Velskii & Landis, 1962)
  - Discussed in this lecture...
- B-trees / 2-3-4 trees (Bayer & McCreight, 1972)
- BB[ $\alpha$ ] trees (Nievergelt & Reingold, 1973)
- Red-black trees (see CLRS 13)
- Splay trees (Sleator and Tarjan, 1985)
- Skip Lists (Pugh, 1989)
- Treaps (Seidel and Aragon, 1996)

# Now, after we learn balanced BST

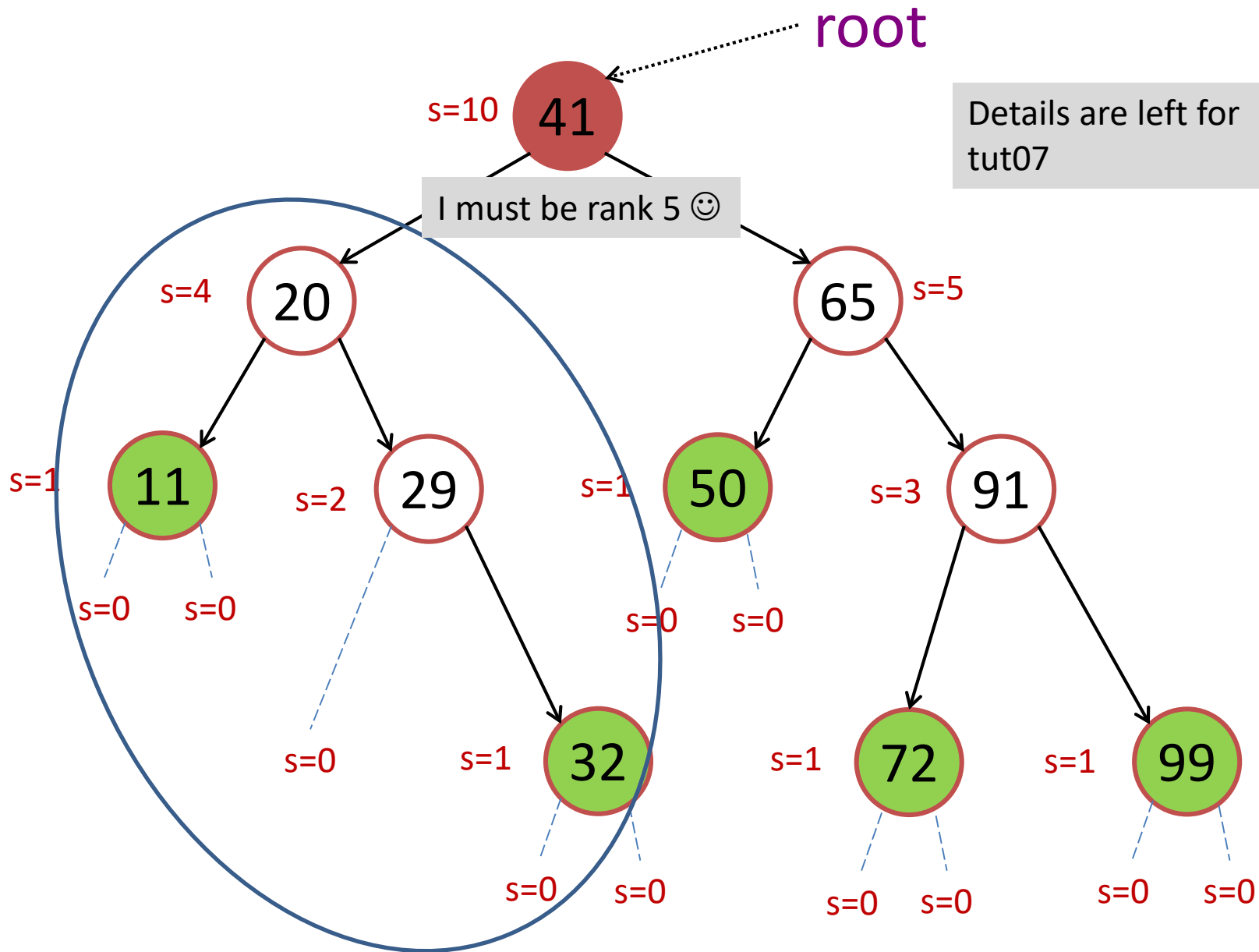
No	Operation	Unsorted Array	Sorted Array	<u>b</u> BST
1	Search(age)	$O(N)$	$O(\log N)$	$O(\log N)$
2	Insert(age)	$O(1)$	$O(N)$	$O(\log N)$
3	FindOldest()	$O(N)$	$O(1)$	$O(\log N)$
4	ListSortedAges()	$O(N \log N)$	$O(N)$	$O(N)$
5	NextOlder(age)	$O(N)$	$O(\log N)$	$O(\log N)$
6	Remove(age)	$O(N)$	$O(N)$	$O(\log N)$
7	GetMedian()	$O(N \log N)$	$O(1)$	????
8	NumYounger(age)	$O(N \log N)$	$O(\log N)$	????

$$\text{NumYounger}(\text{age}) = \text{rank}(\text{age}) - 1$$

---

Now, how to get  $\text{rank}(v)$  efficiently?

# Binary Search Trees: Size (s)



# Balanced BST

---

## Summary:

- The Importance of Being Balanced
- Height Balanced Trees
- Tree Rotations
- AVL trees