CS2020
Data Structures and Algorithms

Welcome!

Today’s Plan

Today's Plan

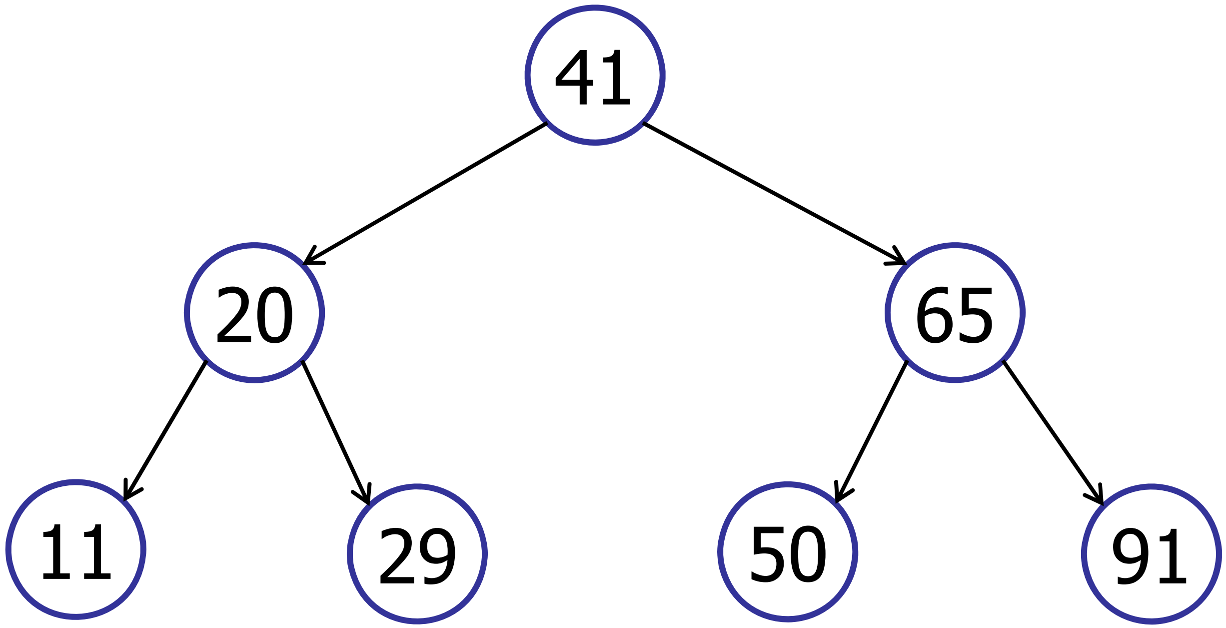
On the importance of being balanced

Today's Plan

On the importance of being balanced

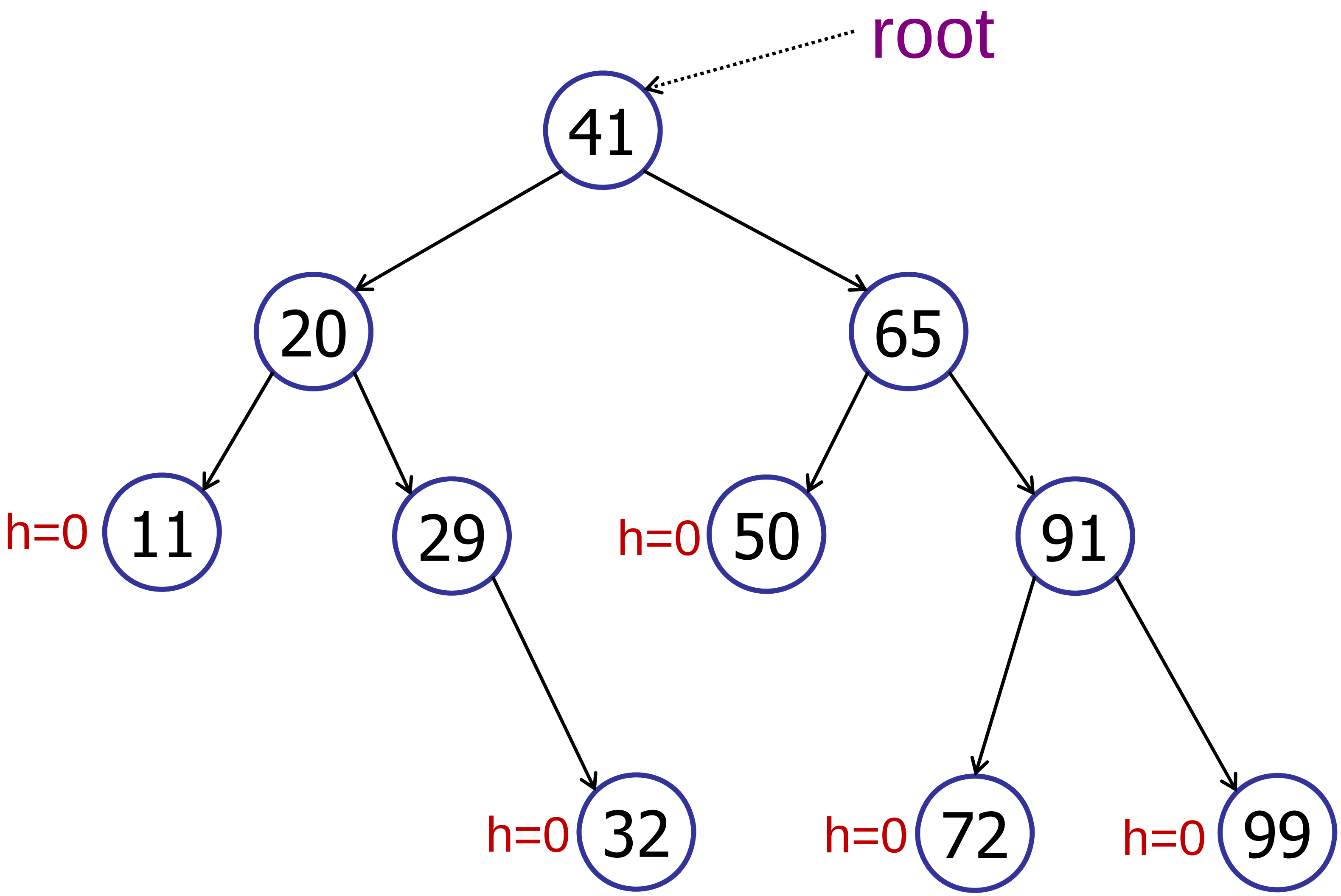
- Height-balanced binary search trees
- AVL trees
- Splay trees

Recap: Binary Search Trees

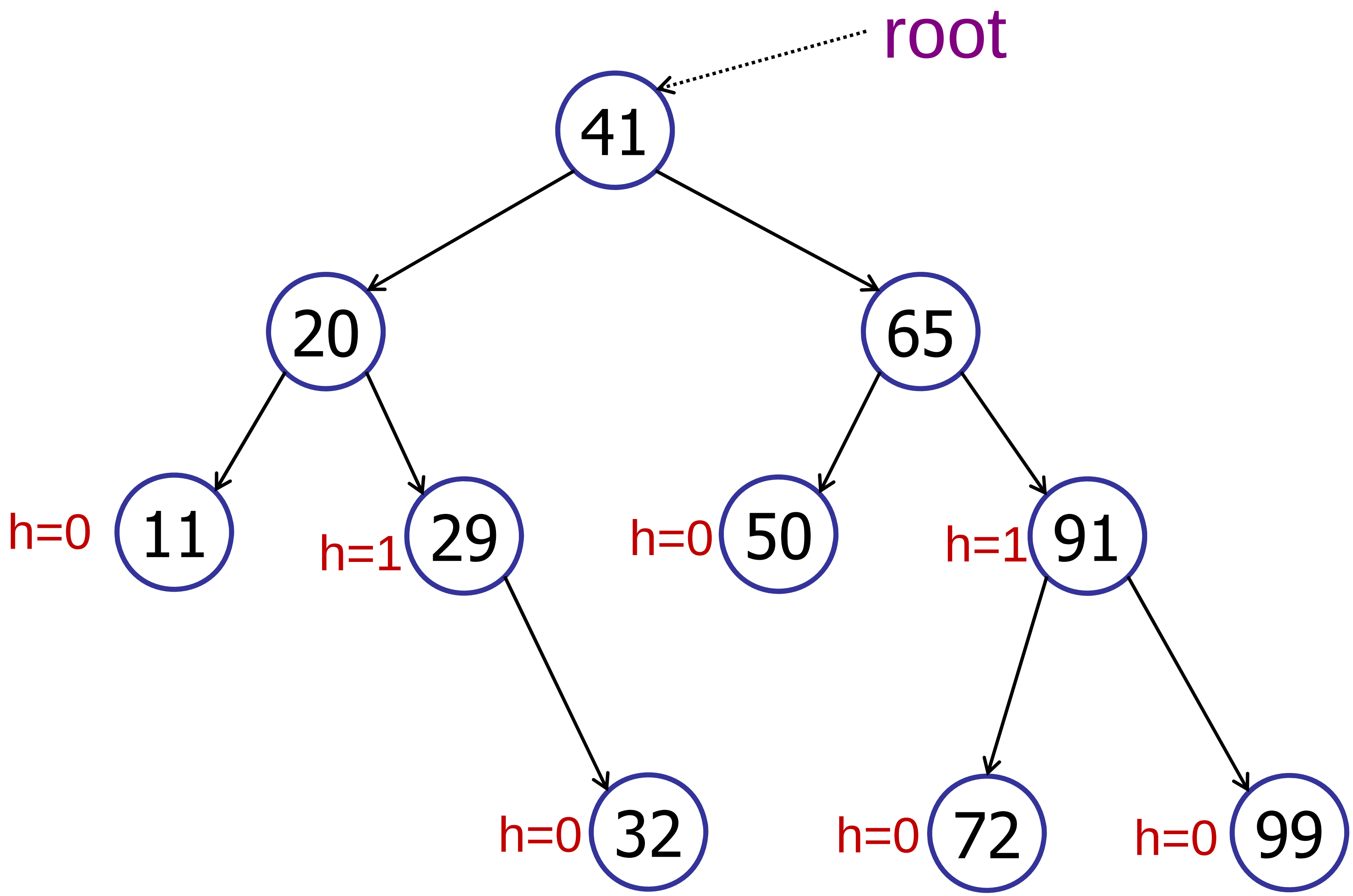


- Two children: v.left, v.right
- Key: v.key
- **BST Property**: all in left sub-tree < key < all in right sub-right

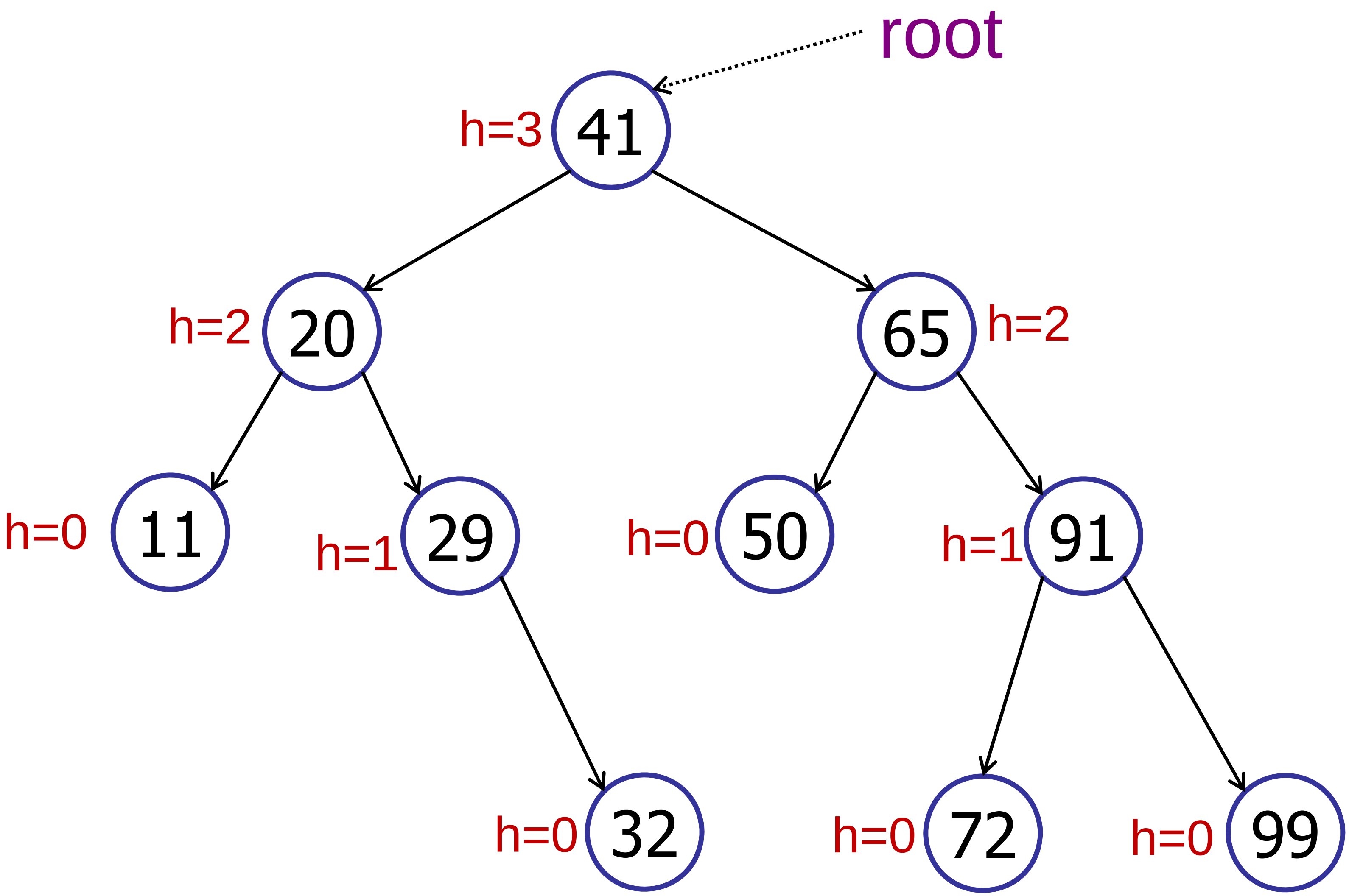
Binary Search Trees Heights



Binary Search Trees Heights



Binary Search Trees Heights



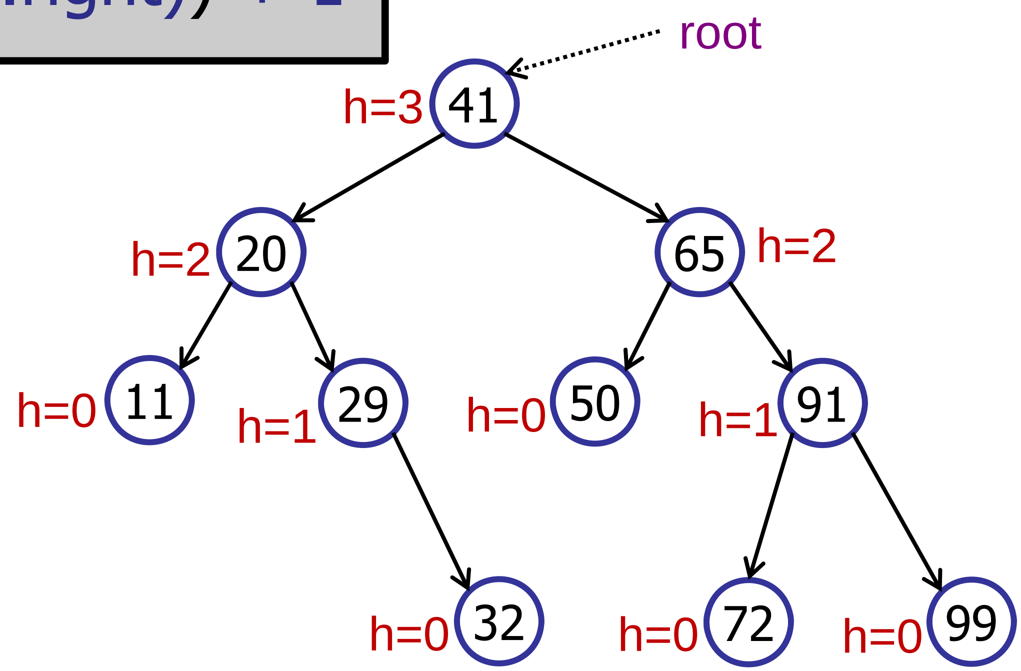
Binary Search Trees Heights

Height:

Number of edges on longest path from root to leaf.

h(v) = 0 (if v is a leaf)

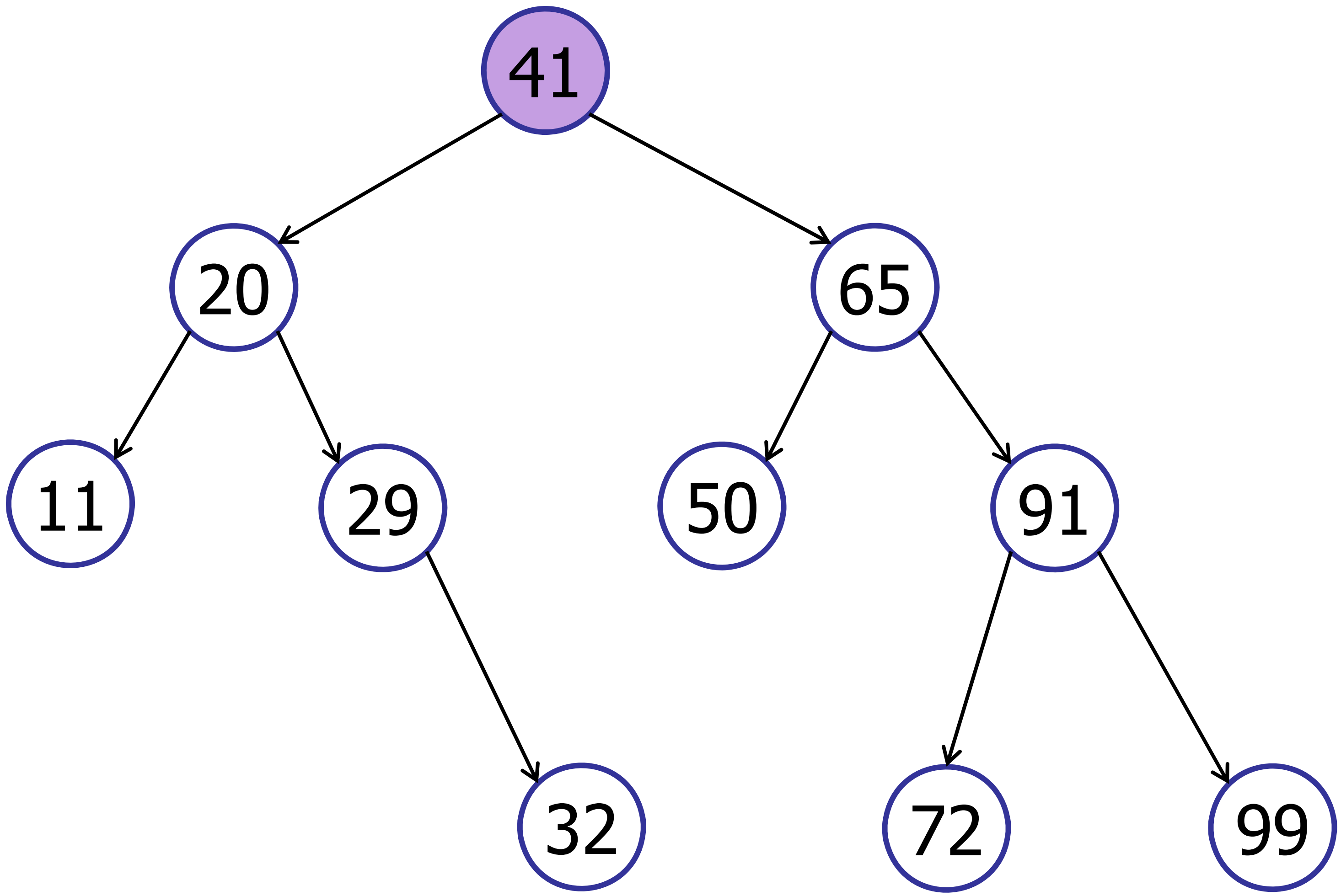
h(v) = max(h(v.left), h(v.right)) + 1



(For simplicity: h(null) = -1)

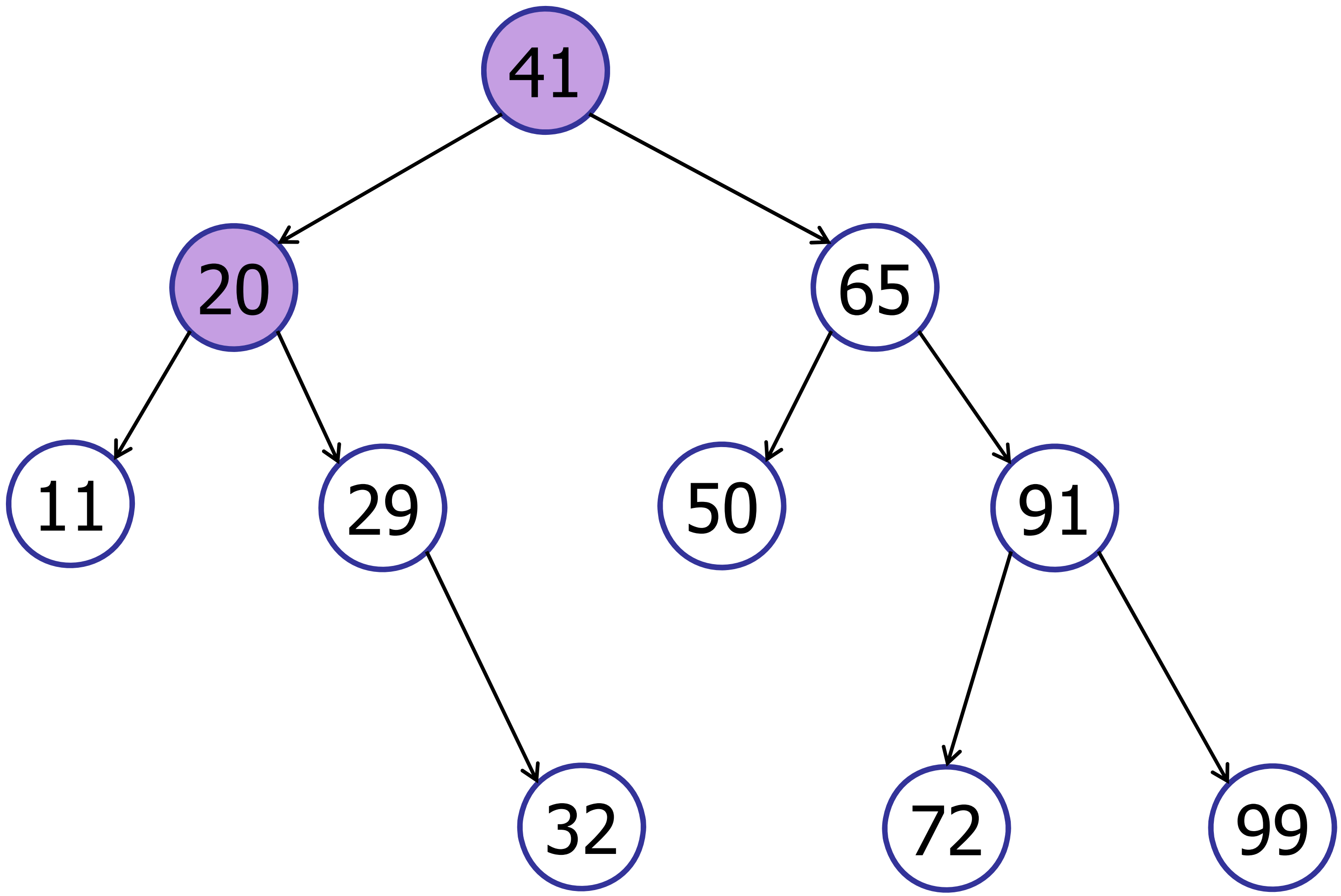
Binary Search Trees (review)

insert(27)



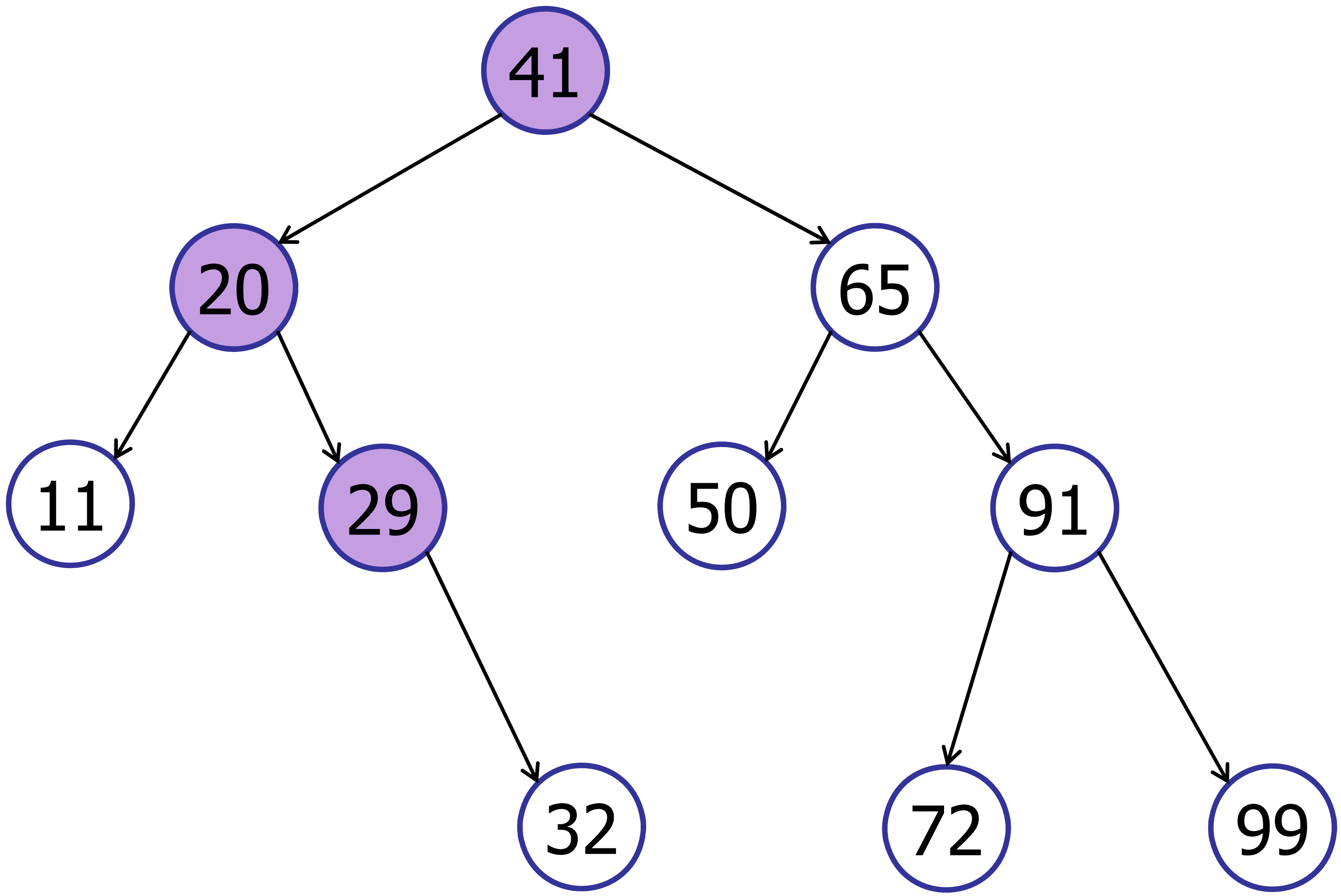
Binary Search Trees (review)

insert(27)



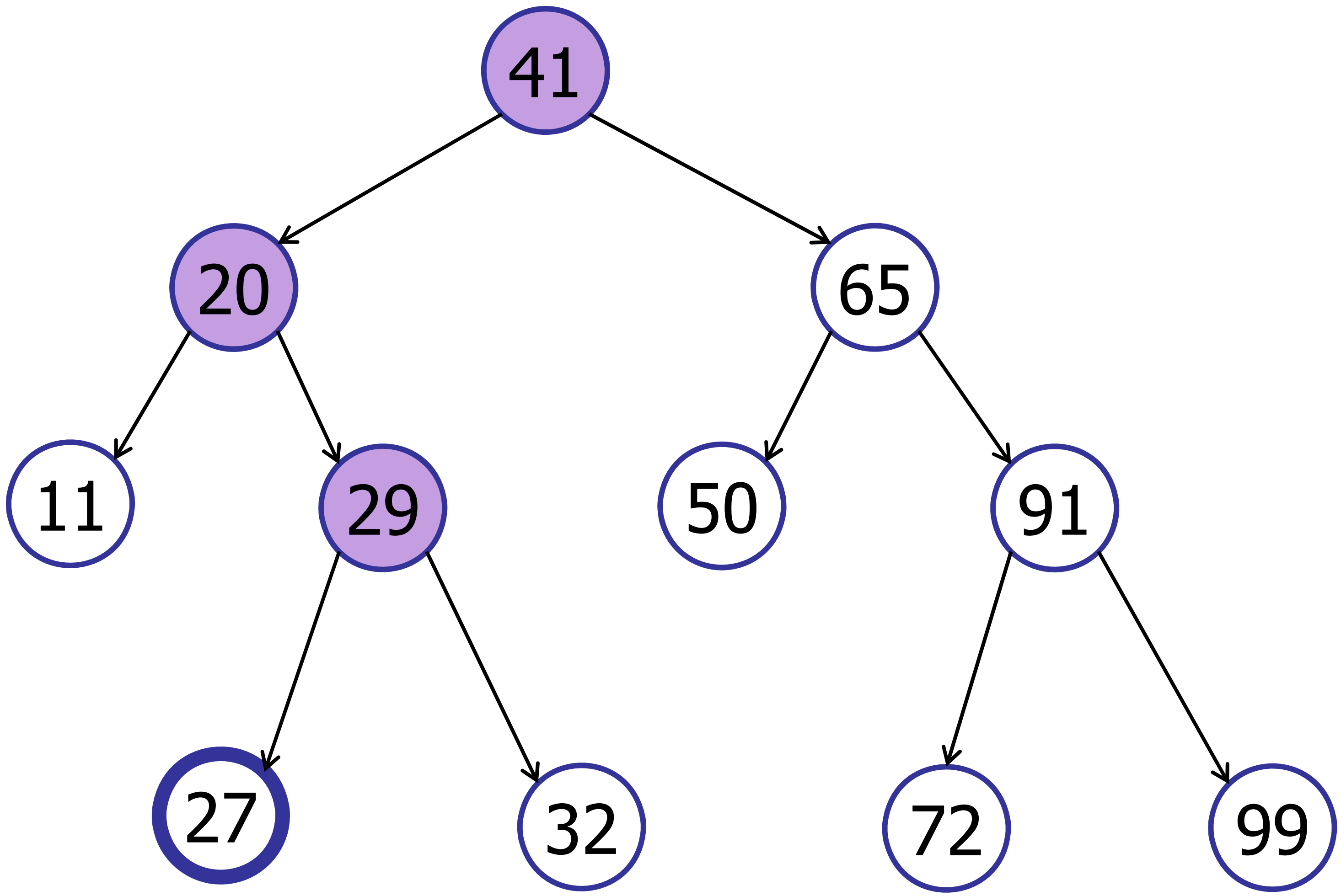
Binary Search Trees (review)

insert(27)



Binary Search Trees (review)

insert(27)



Binary Search Tree

Modifying Operations

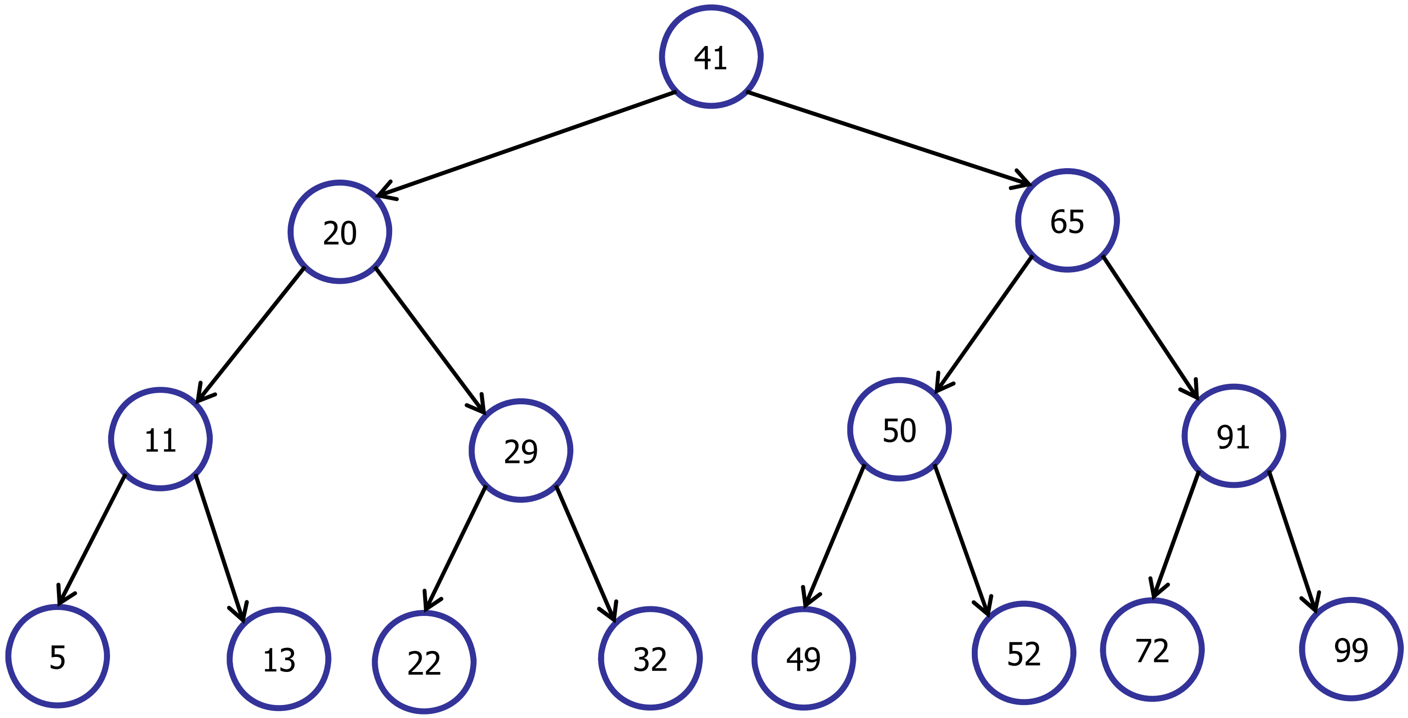
- insert: O(h)
- delete: O(h)

Query Operations:

- search: O(h)
- predecessor, successor: O(h)
- findMax, findMin: O(h)
- in-order-traversal: O(n)

The Importance of Being Balanced

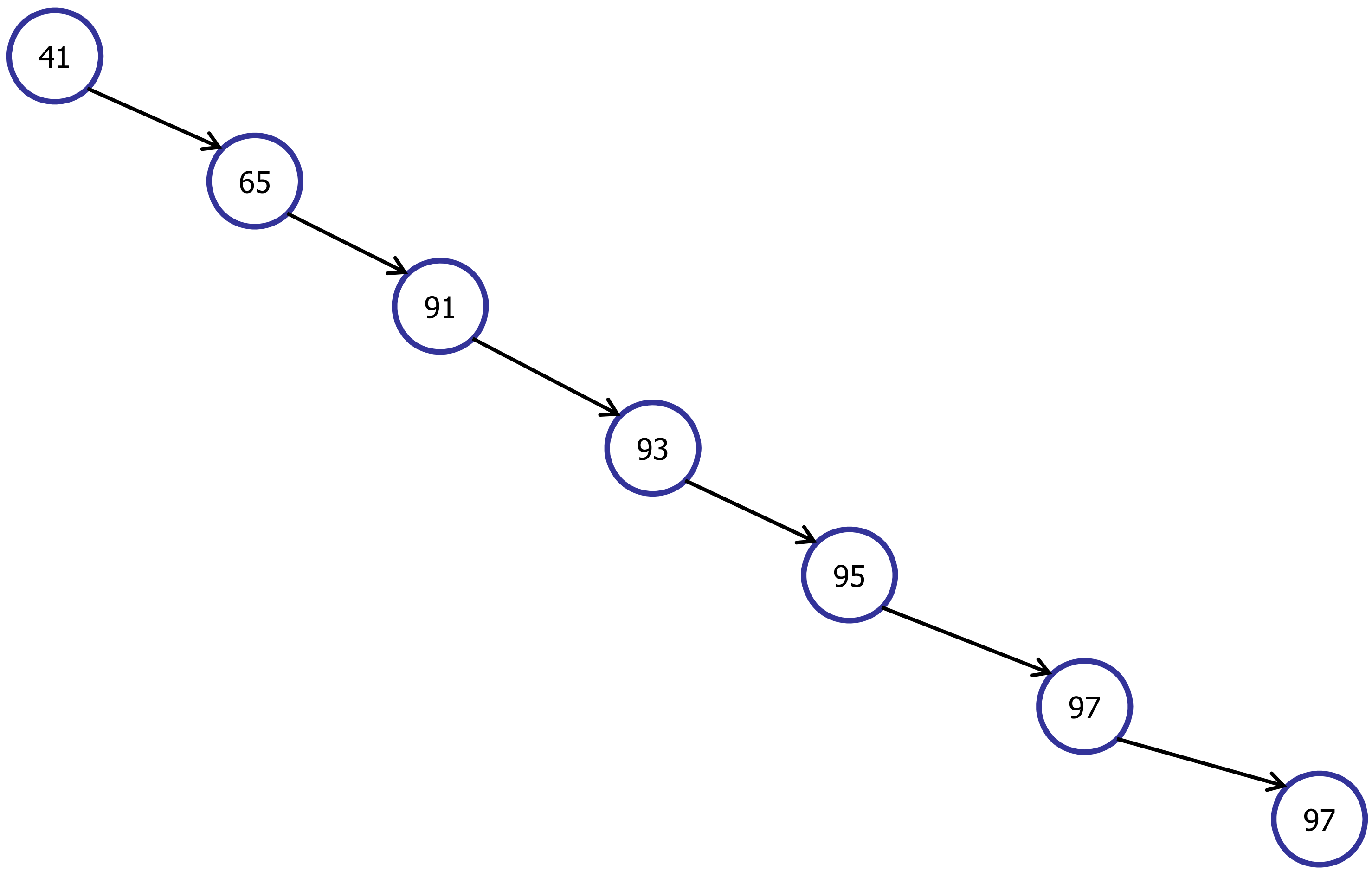
Operations take O(h) time



The Importance of Being Balanced

Operations take O(h) time

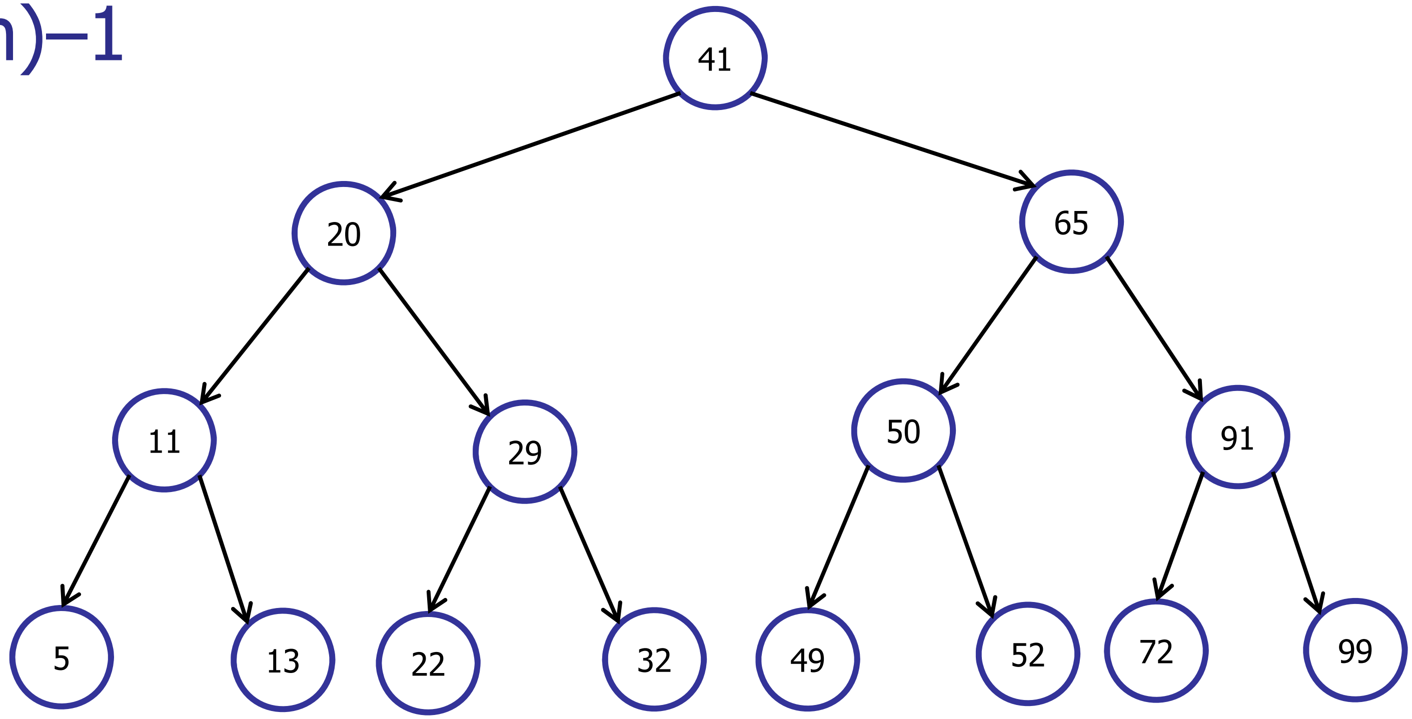
$h \le n$



The Importance of Being Balanced

Operations take O(h) time

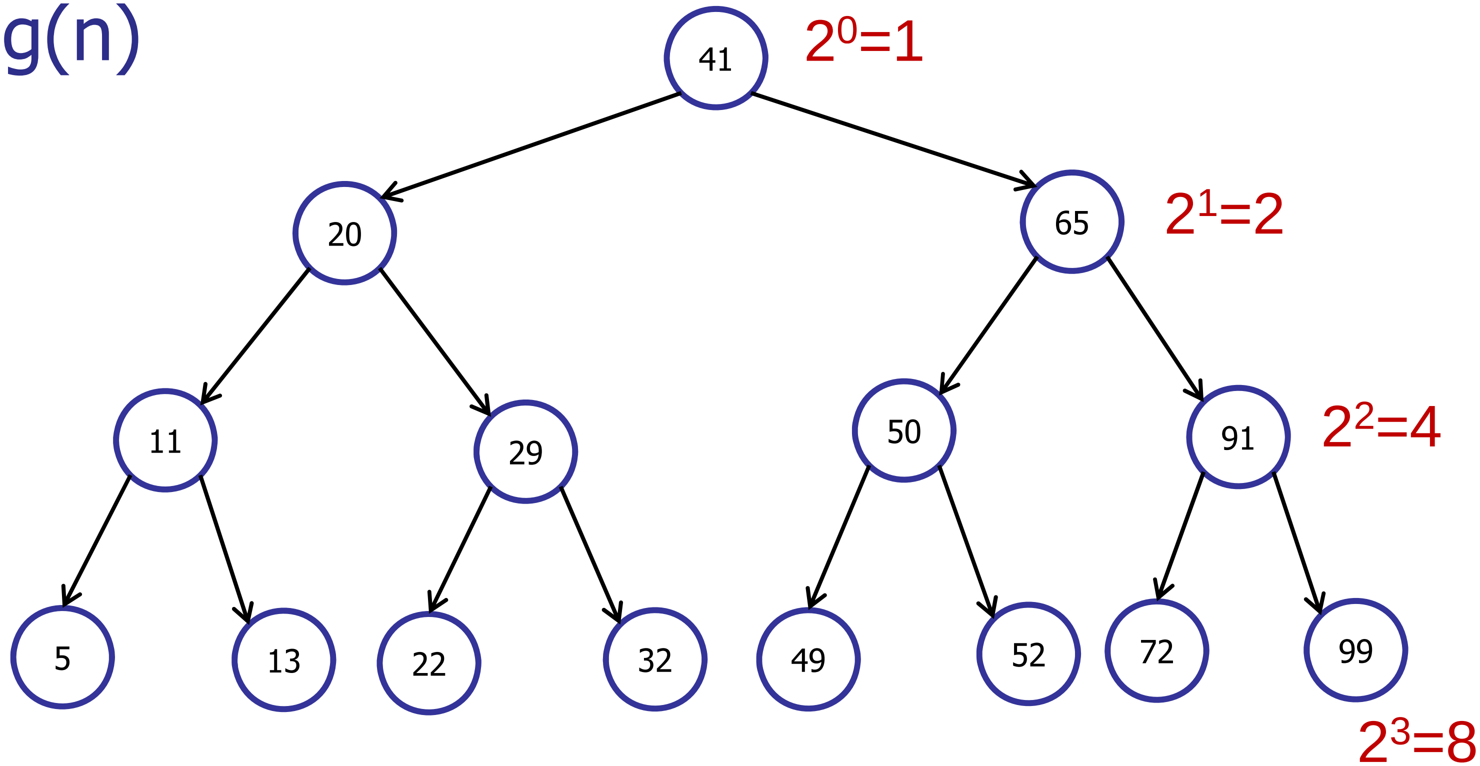
$h \geq \log(n) - 1$



The Importance of Being Balanced

Operations take O(h) time

$h+1 \geq \log(n)$



$$n \leq 1 + 2 + 4 + \ldots + 2^h$$

$$\leq 2^0 + 2^1 + 2^2 + \ldots + 2^h < 2^{h+1}$$

The Importance of Being Balanced

Operations take $O(h)$ time

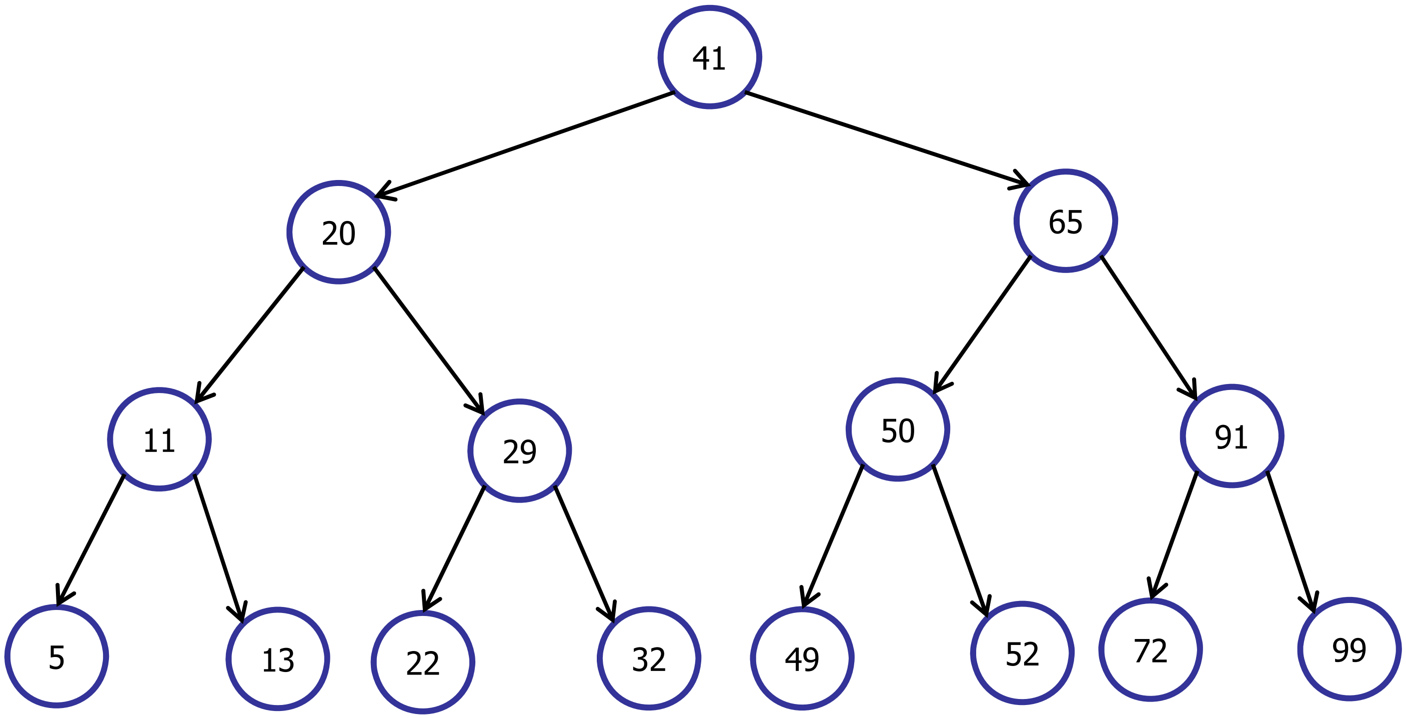
$\log(n) - 1 \leq h \leq n$

A BST is <u>balanced</u> if $h = O(\log n)$

On a balanced BST: all operations run in $O(\log n)$ time.

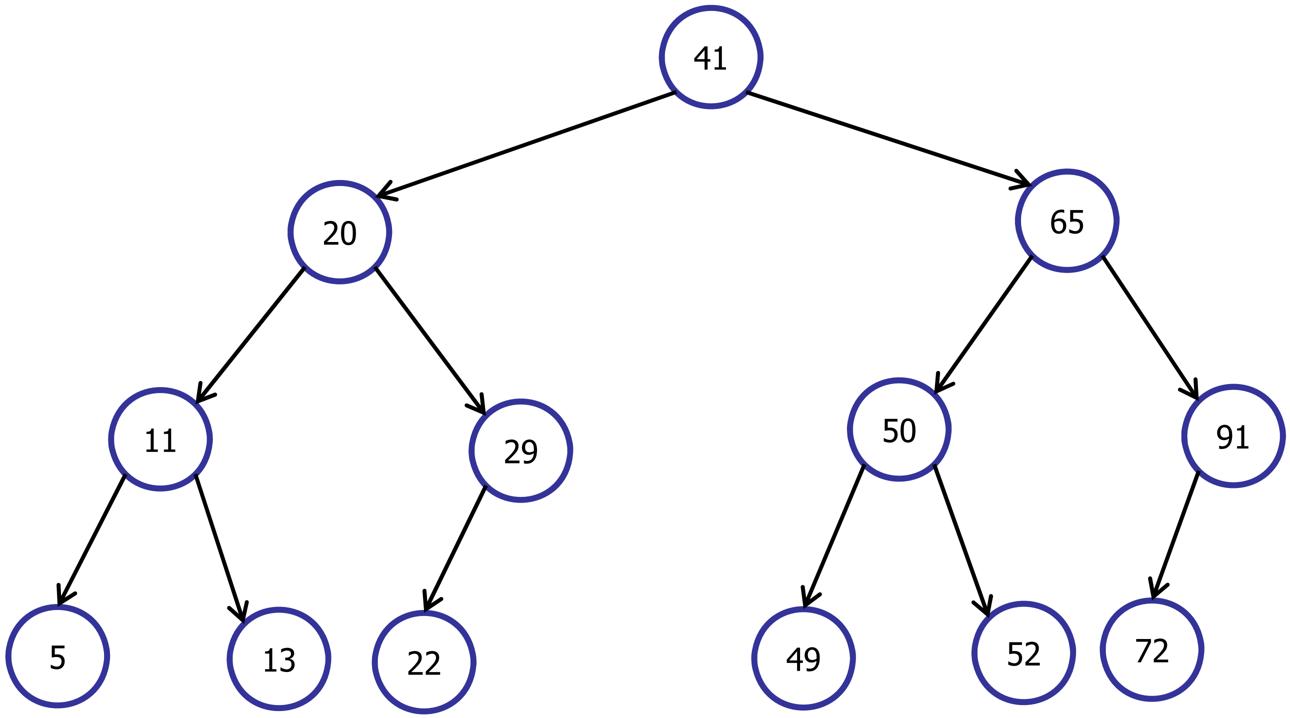
The Importance of Being Balanced

Perfectly balanced:



The Importance of Being Balanced

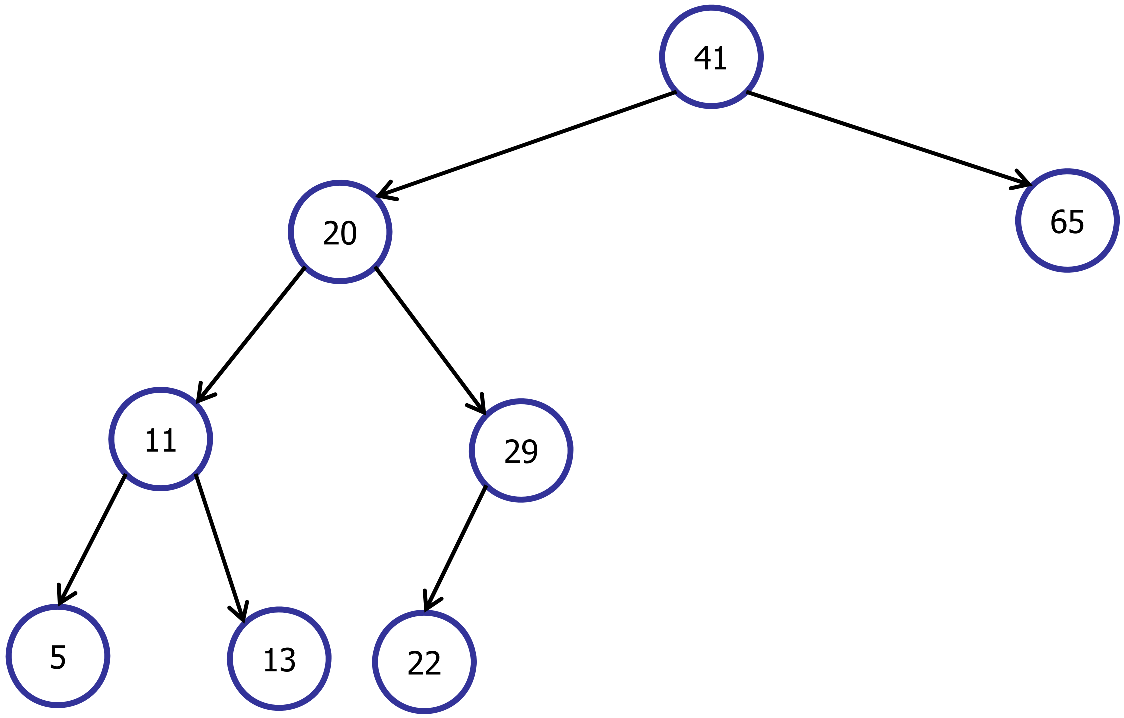
Almost perfectly balanced:



Every subtree has (almost) the same number of nodes.

The Importance of Being Balanced

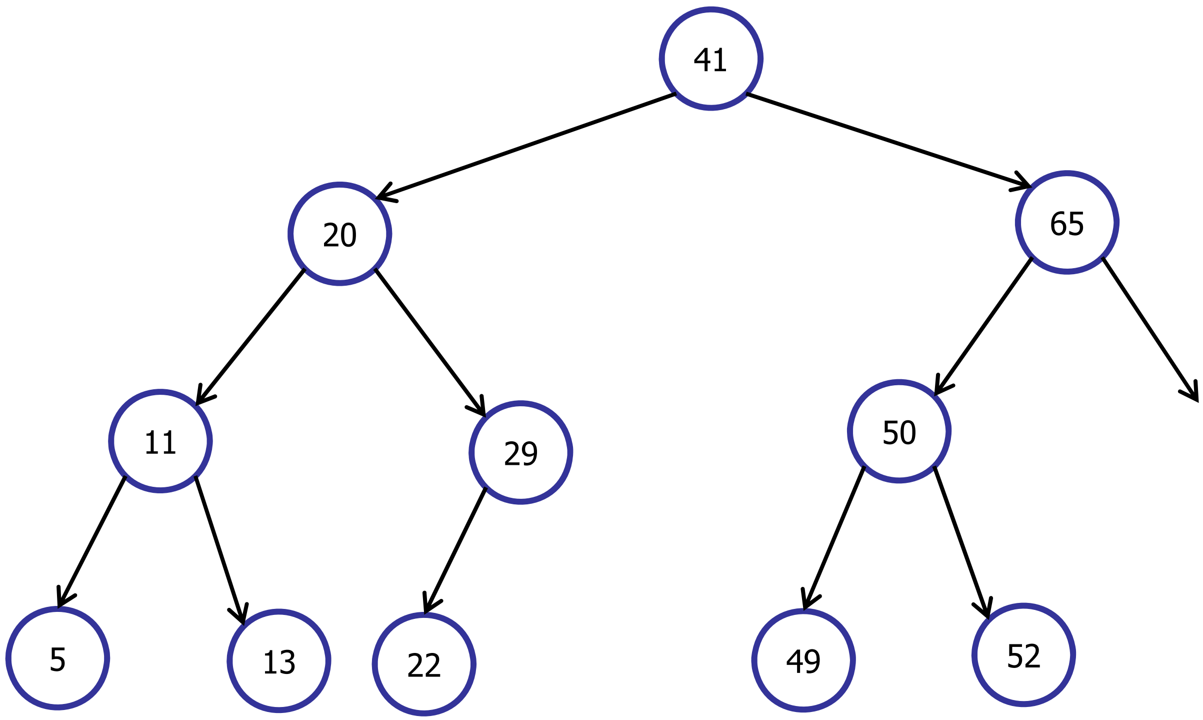
Not perfectly balanced:



Left tree has 6, right tree has 1.

The Importance of Being Balanced

Not perfectly balanced:



Balanced Search Trees

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)

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.

AVL Trees [Adelson-Velskii & Landis 1962]

Step 1: Augment

- In every node v, store height:

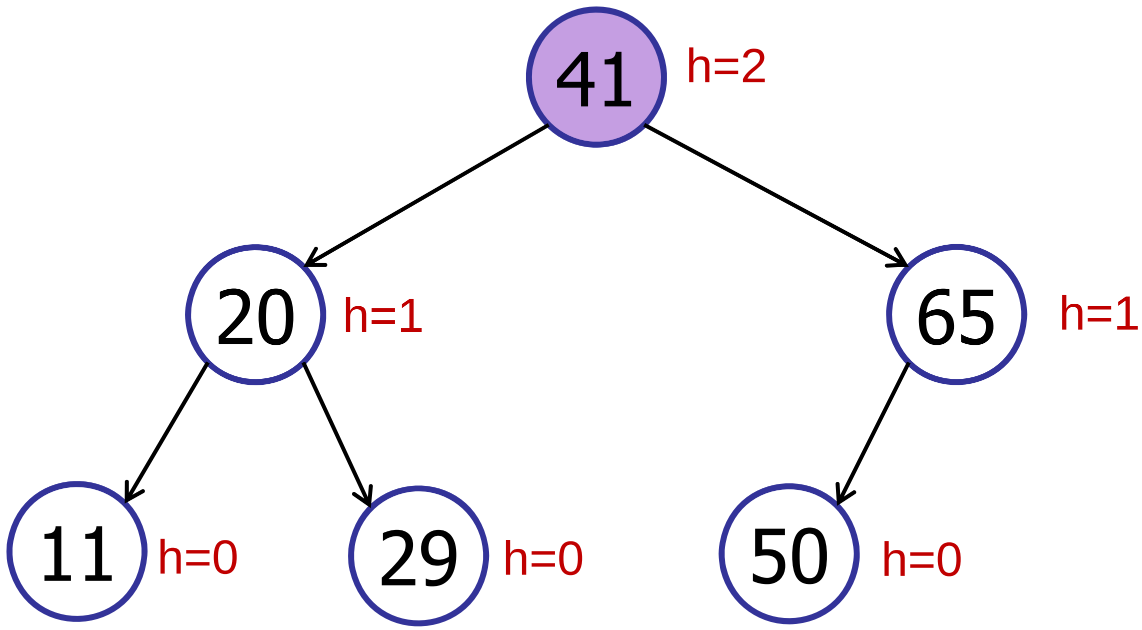
  v.height = h(v)

- On insert & delete update height:

```
insert(x)
    if (x < key)
        left.insert(x)
    else right.insert(x)
    height = max(left.height, right.height) + 1
```

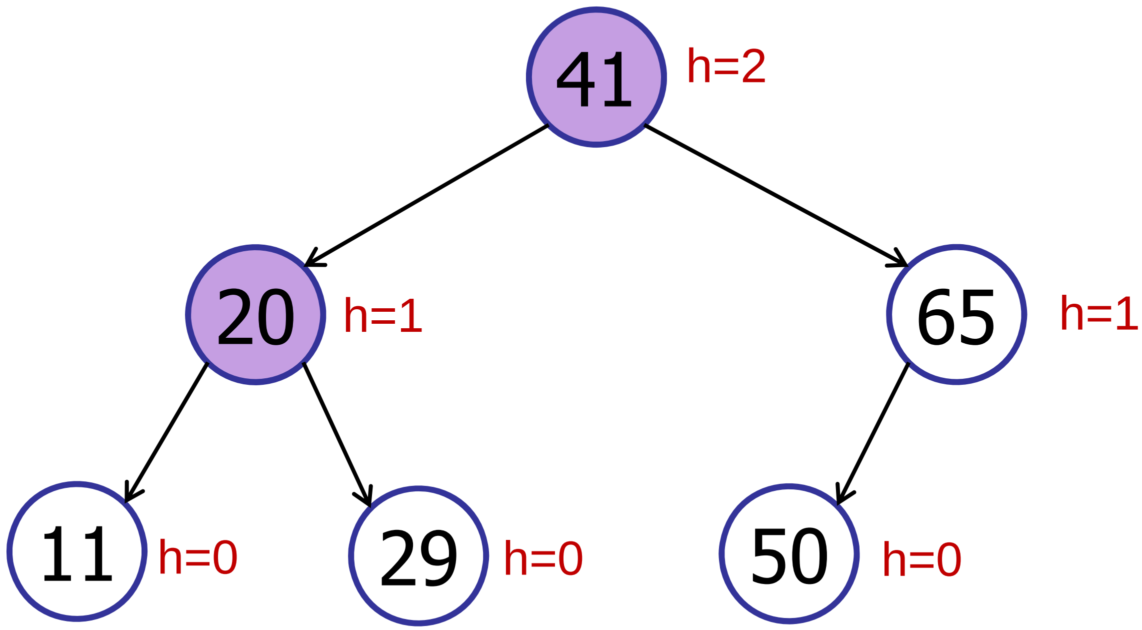
Binary Search Trees

insert(27)



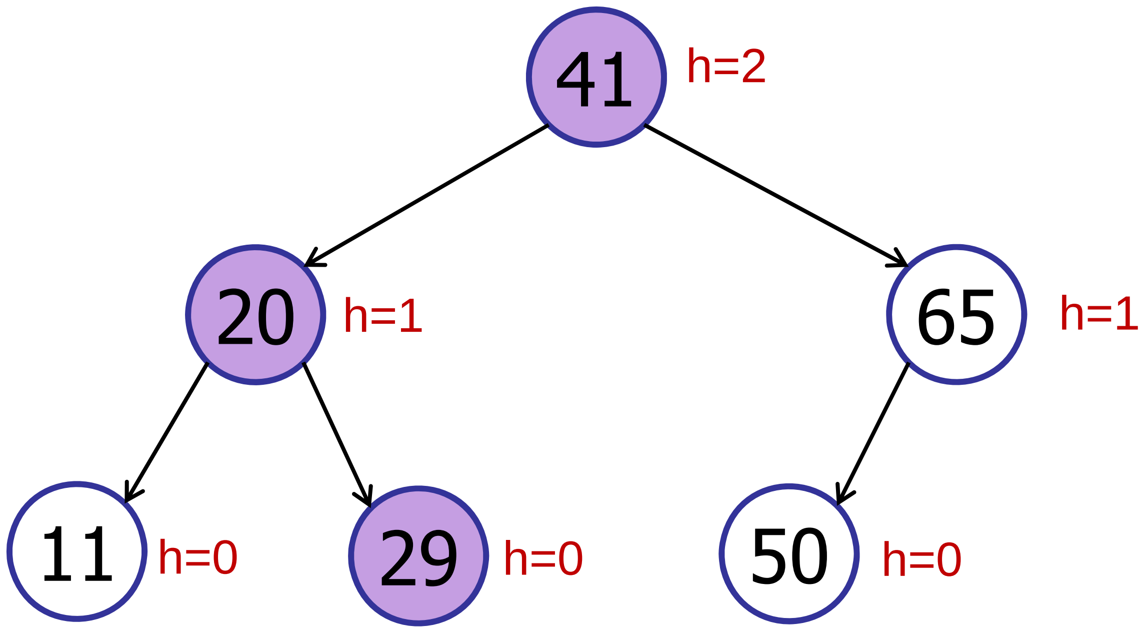
Binary Search Trees

insert(27)



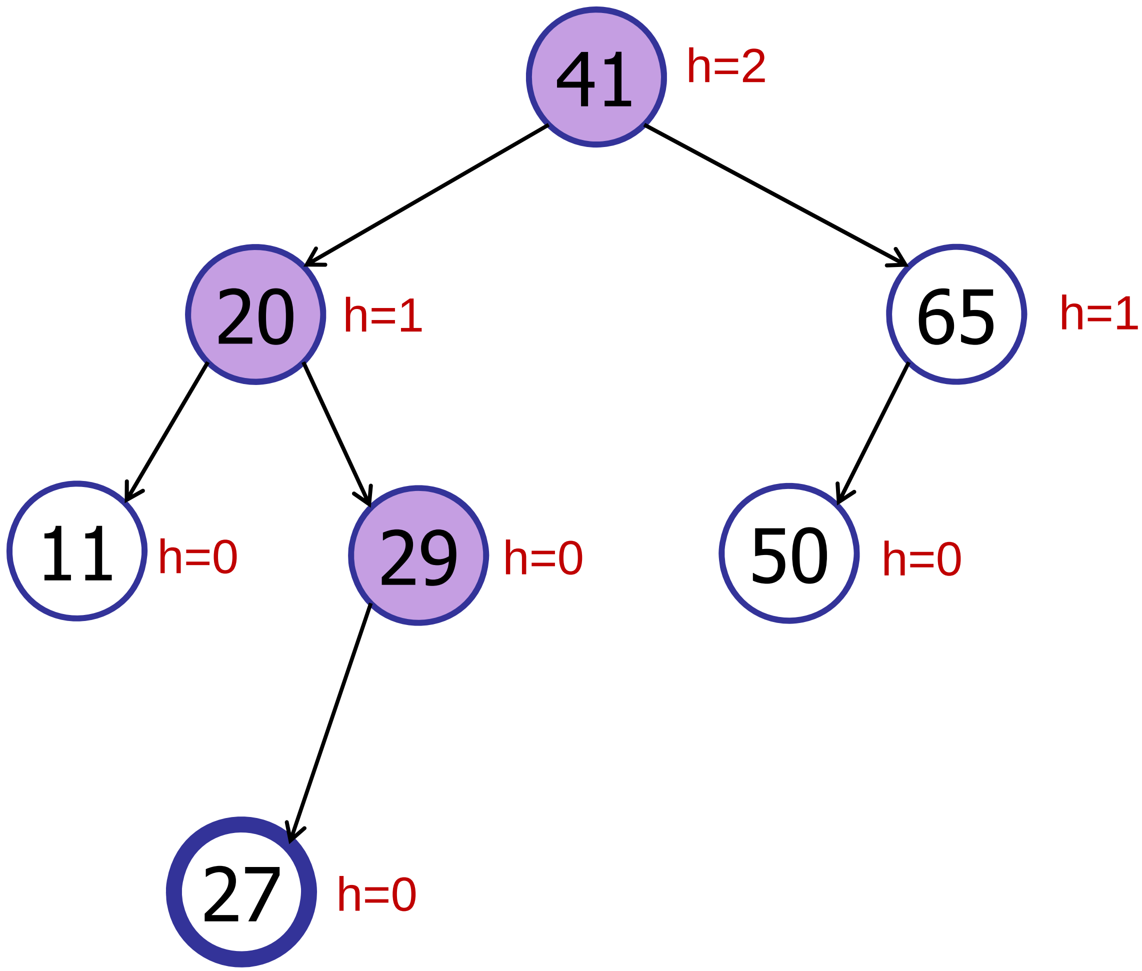
Binary Search Trees

insert(27)



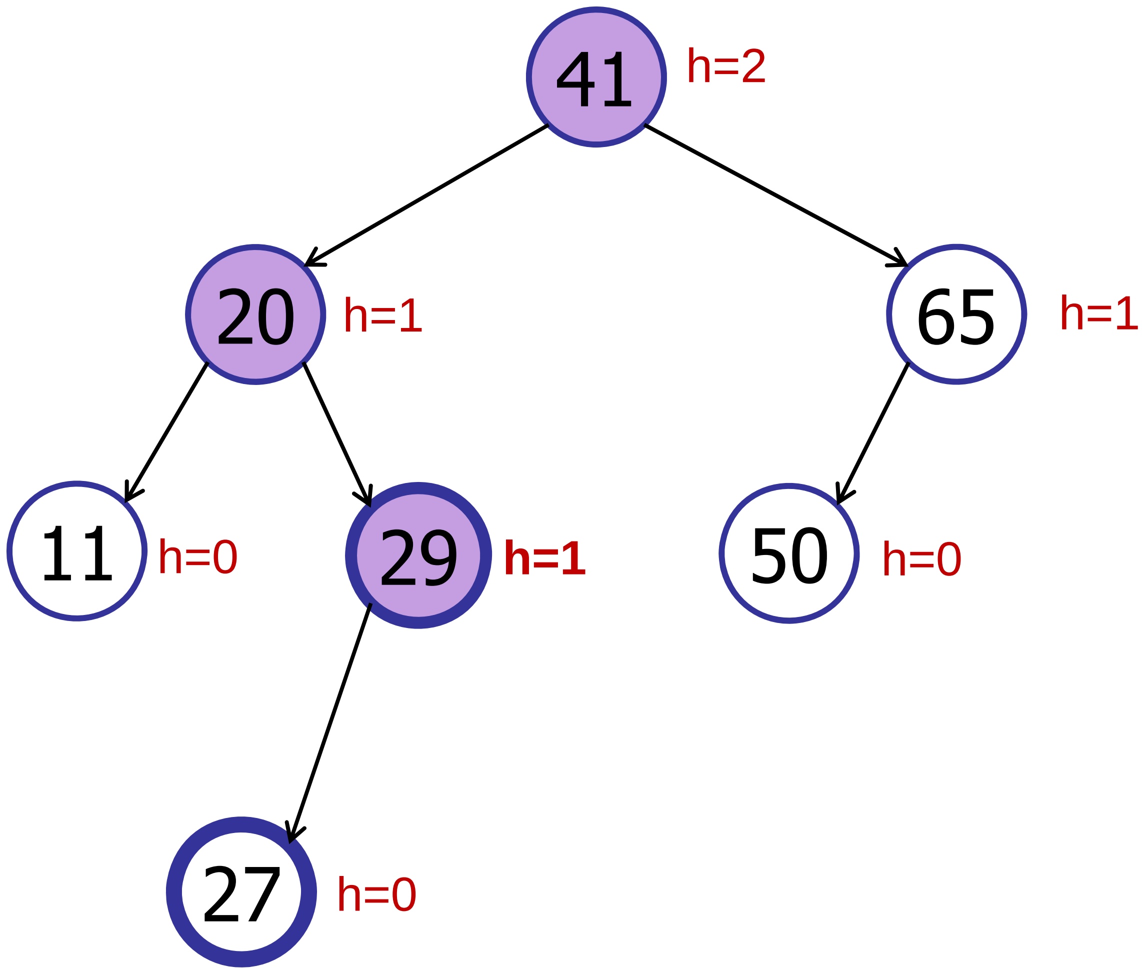
Binary Search Trees

insert(27)



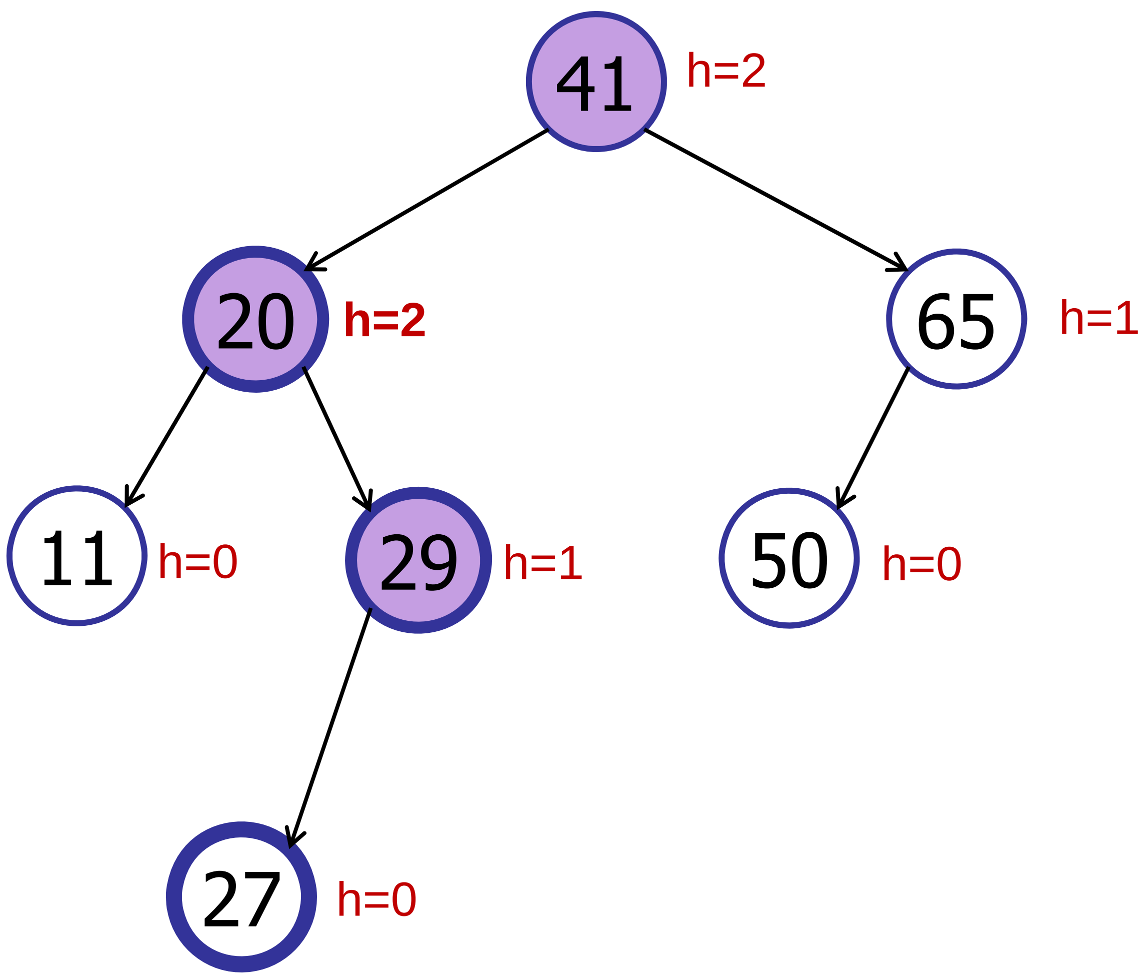
Binary Search Trees

insert(27)



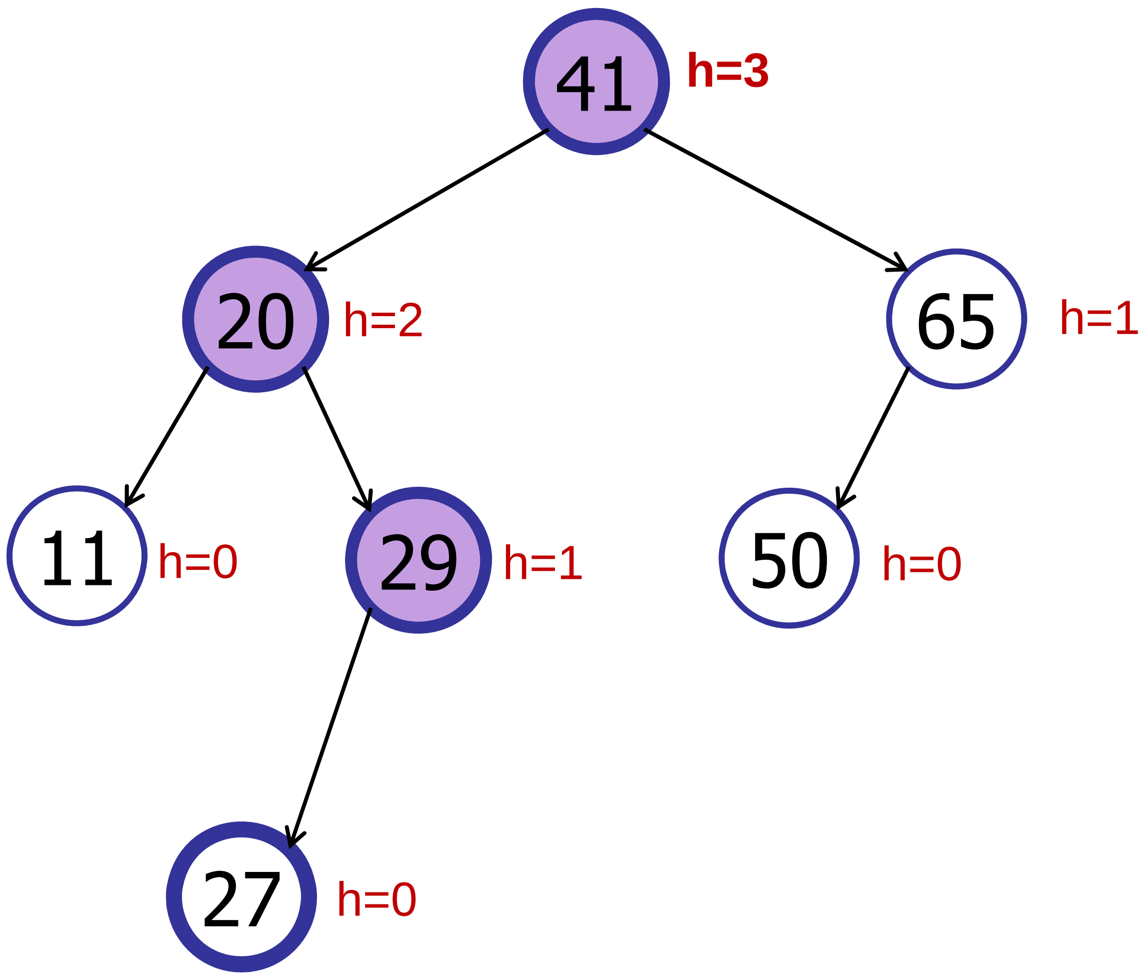
Binary Search Trees

insert(27)



Binary Search Trees

insert(27)



AVL Trees [Adelson-Velskii & Landis 1962]

Step 1: Augment

- In every node v, store height:

  v.height = h(v)

- On insert & delete update height:

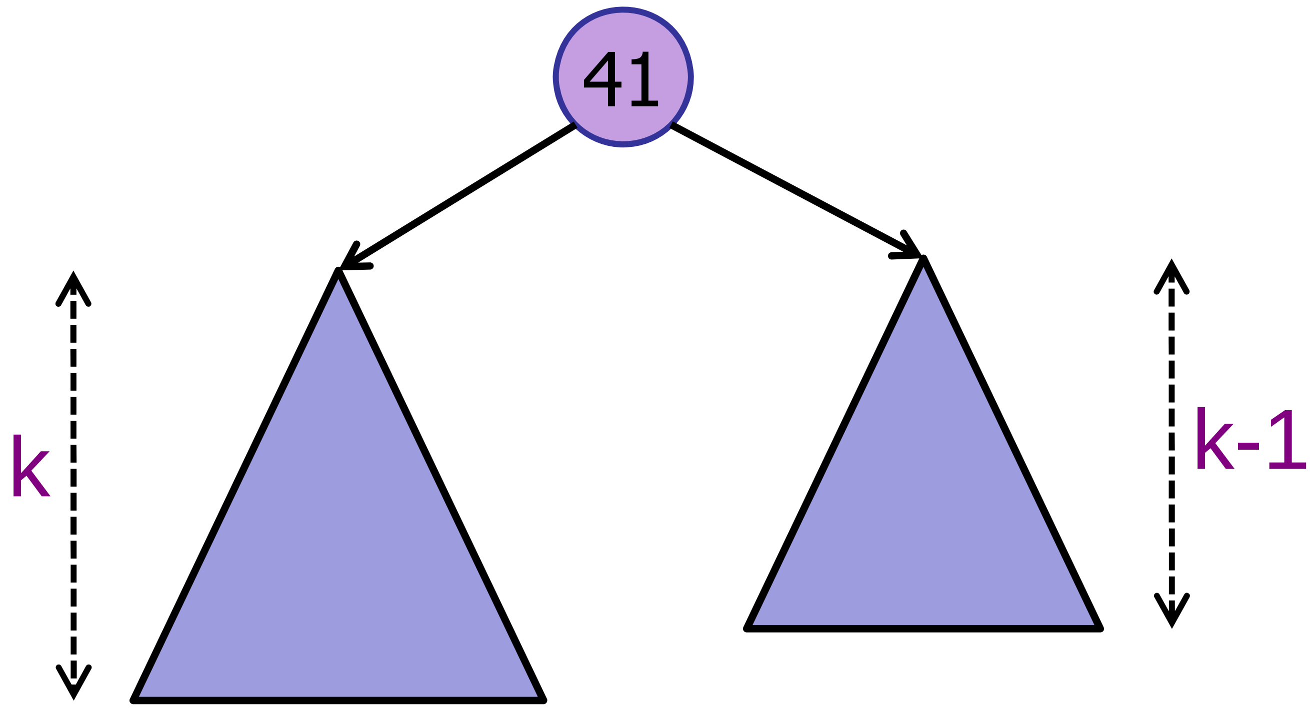
```
insert(x)
        if (x < key)
                left.insert(x)
        else right.insert(x)
        height = max(left.height, right.height) + 1
```

AVL Trees [Adelson-Velskii & Landis 1962]

Step 2: Define Invariant

A node v is <u>height-balanced</u> if:

$$|v.left.height - v.right.height| \leq 1$$



AVL Trees [Adelson-Velskii & Landis 1962]

Step 2: Define Invariant

- A node v is <u>height-balanced</u> if:

$$|v.left.height - v.right.height| \leq 1$$

- A binary search tree is <u>height balanced</u> if every node in the tree is height-balanced.

Height-Balanced Trees

Claim:

A height-balanced tree with n nodes has **at most** height $h < 2\log(n)$.

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

$\Leftrightarrow$ For a tree with height h, the tree can contain **at least** $n > 2^{h/2}$ nodes

Height-Balanced Trees

Proof:

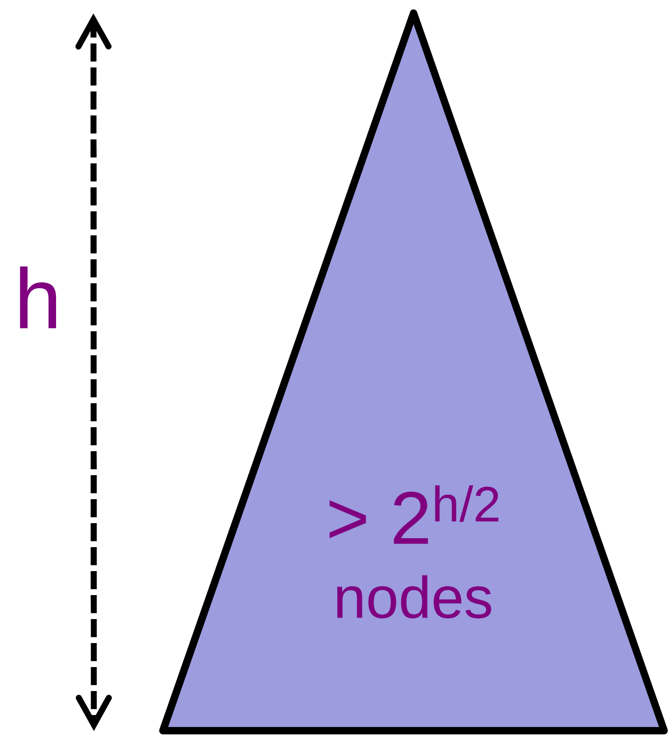
Let n_h be the minimum number of nodes in a height-balanced tree of height h.

Show:

$n_h > 2^{h/2}$

$\Rightarrow$

$2\log(n_h) > h$



Height-Balanced Trees

Proof:

Let n_h be the minimum number of nodes in a height-balanced tree of height h.

Show:

$n_h > 2^{h/2}$

$\Rightarrow$

If you give me a tree of height $h = 2\log(n)+1$, then it must have $> 2^{\log(n)+½} > n$ nodes.

Height-Balanced Trees

Proof:

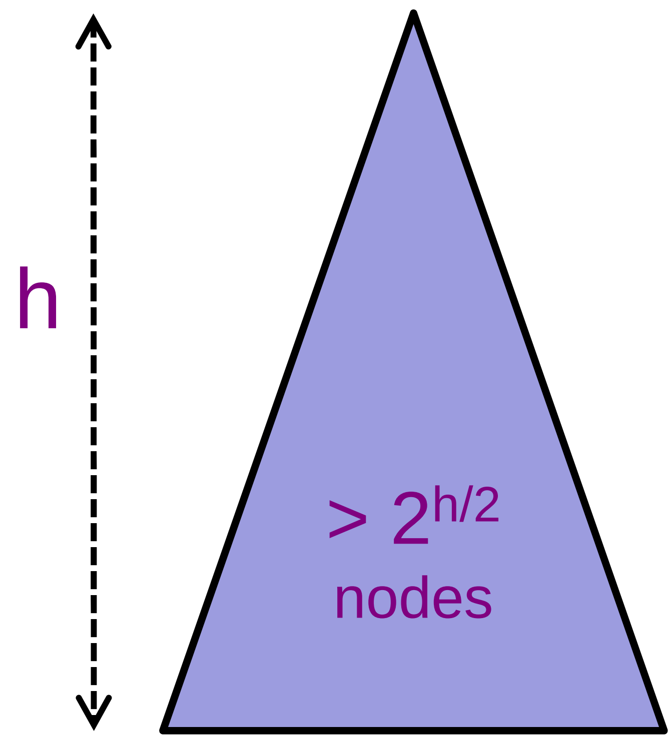
Let n_h be the minimum number of nodes in a height-balanced tree of height h.

Show:

$n_h > 2^{h/2}$

$\Rightarrow$

$2\log(n_h) > h$

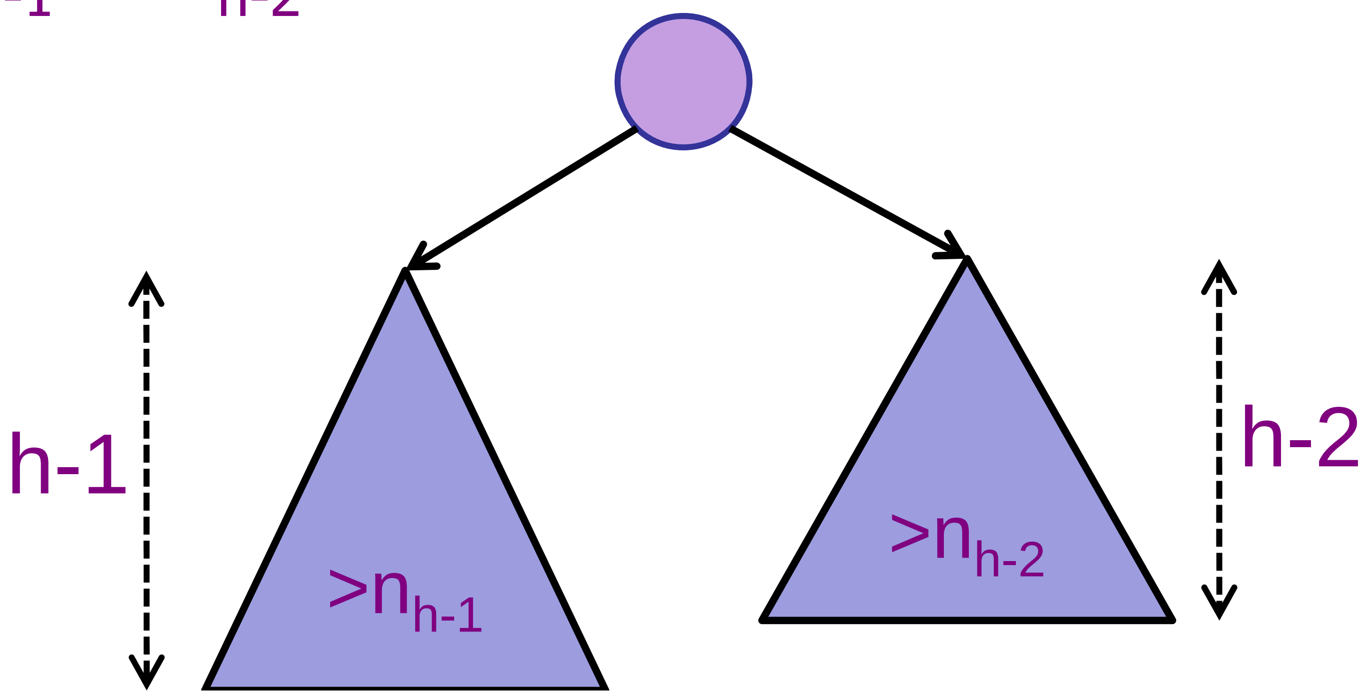


Height-Balanced Trees

Proof:

Let n_h be the minimum number of nodes in a height-balanced tree of height h.

$n_h \geq 1 + n_{h-1} + n_{h-2}$



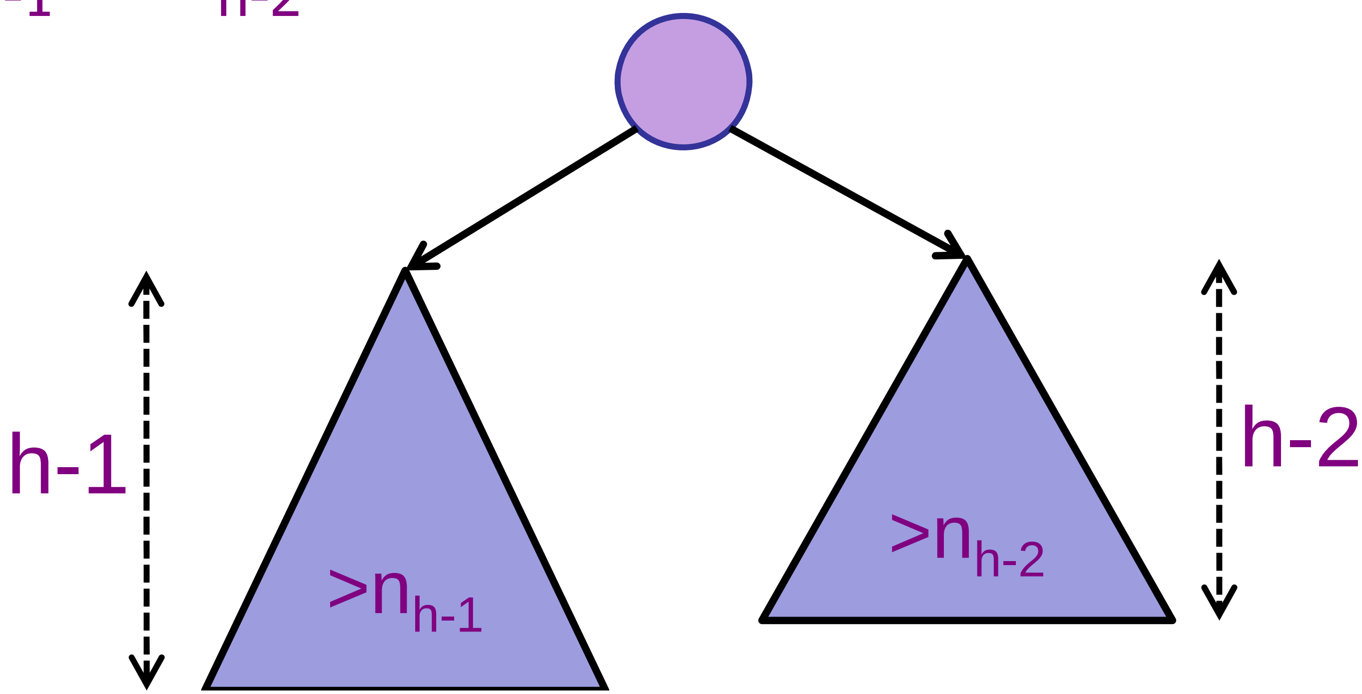
Height-Balanced Trees

Proof:

Let n_h be the minimum number of nodes in a height-balanced tree of height h.

$$n_h \geq 1 + n_{h-1} + n_{h-2}$$

$$\geq 2n_{h-2}$$



Height-Balanced Trees

Proof:

Let n_h be the minimum number of nodes in a height-balanced tree of height h.

$$n_h \geq 1 + n_{h-1} + n_{h-2}$$

$$\geq 2n_{h-2}$$

$$\geq 4n_{h-4}$$

$$\geq 8n_{h-6}$$

$$\geq \ldots$$

Height-Balanced Trees

Proof:

Let n_h be the minimum number of nodes in a height-balanced tree of height h.

$$n_h \geq 1 + n_{h-1} + n_{h-2}$$

$$\geq 2n_{h-2}$$

$$\geq 4n_{h-4}$$

$$\geq 8n_{h-6}$$

$$\geq \ldots$$

Base case:
$n_0 = 1$

Height-Balanced Trees

Proof:

Let n_h be the minimum number of nodes in a height-balanced tree of height h.

$$
\begin{aligned}
n_h &\geq 1 + n_{h-1} + n_{h-2} \\
&\geq 2n_{h-2} \\
&\geq 2^{h/2}\, n_0 \\
&\geq 2^{h/2}
\end{aligned}
$$

Base case:
$n_0 = 1$

Height-Balanced Trees

Claim:

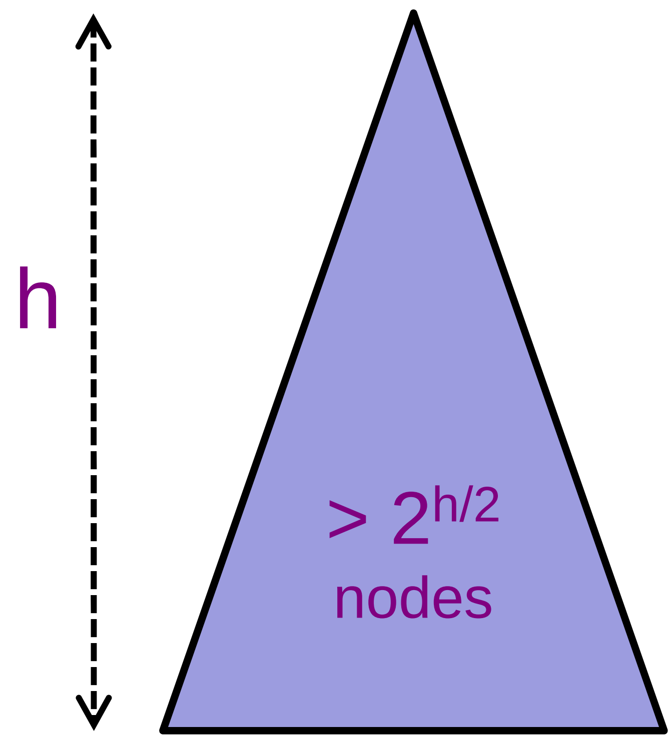
A height-balanced tree with n nodes has height $h < 2\log(n)$.

Show:

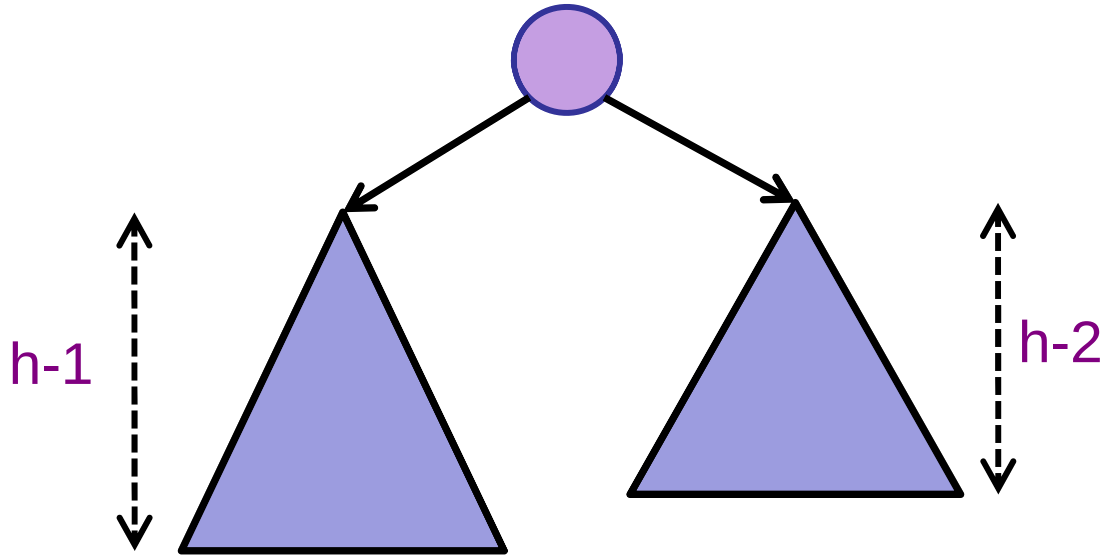
$n_h > 2^{h/2}$

$\Rightarrow$

$2\log(n_h) > h$



Height-Balanced Trees



Show (induction):

F_n = n^{th} Fibonacci number

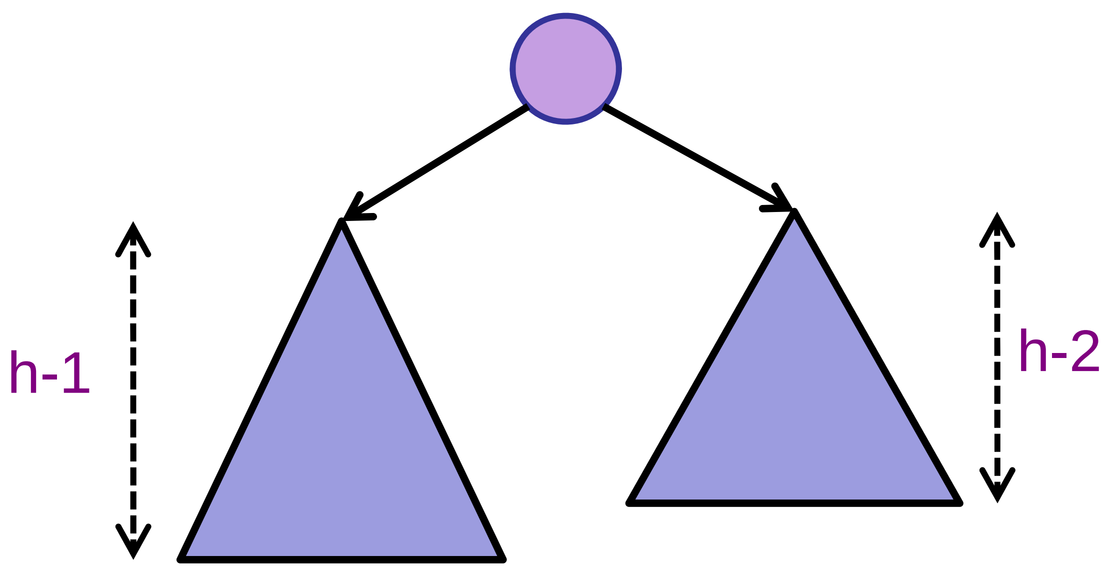
$n_h = F_{h+2} - 1 \cong \phi^{h+1}/\sqrt{5} - 1$ (rounded to nearest int)

$h \cong \log(n) / \log(\phi)$ $\qquad$ $\phi \cong 1.618$

Height-Balanced Trees

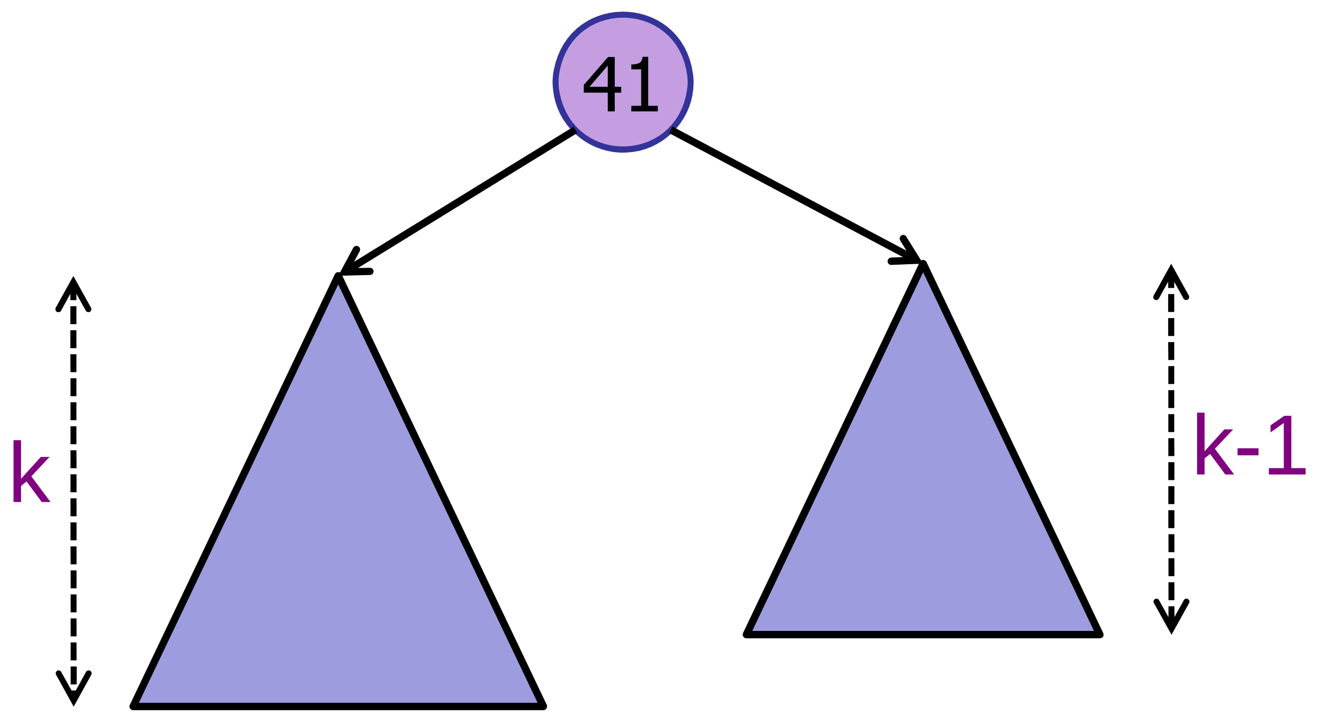
Claim:

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



AVL Trees [Adelson-Velskii & Landis 1962]

Step 3: Show how to maintain height-balance

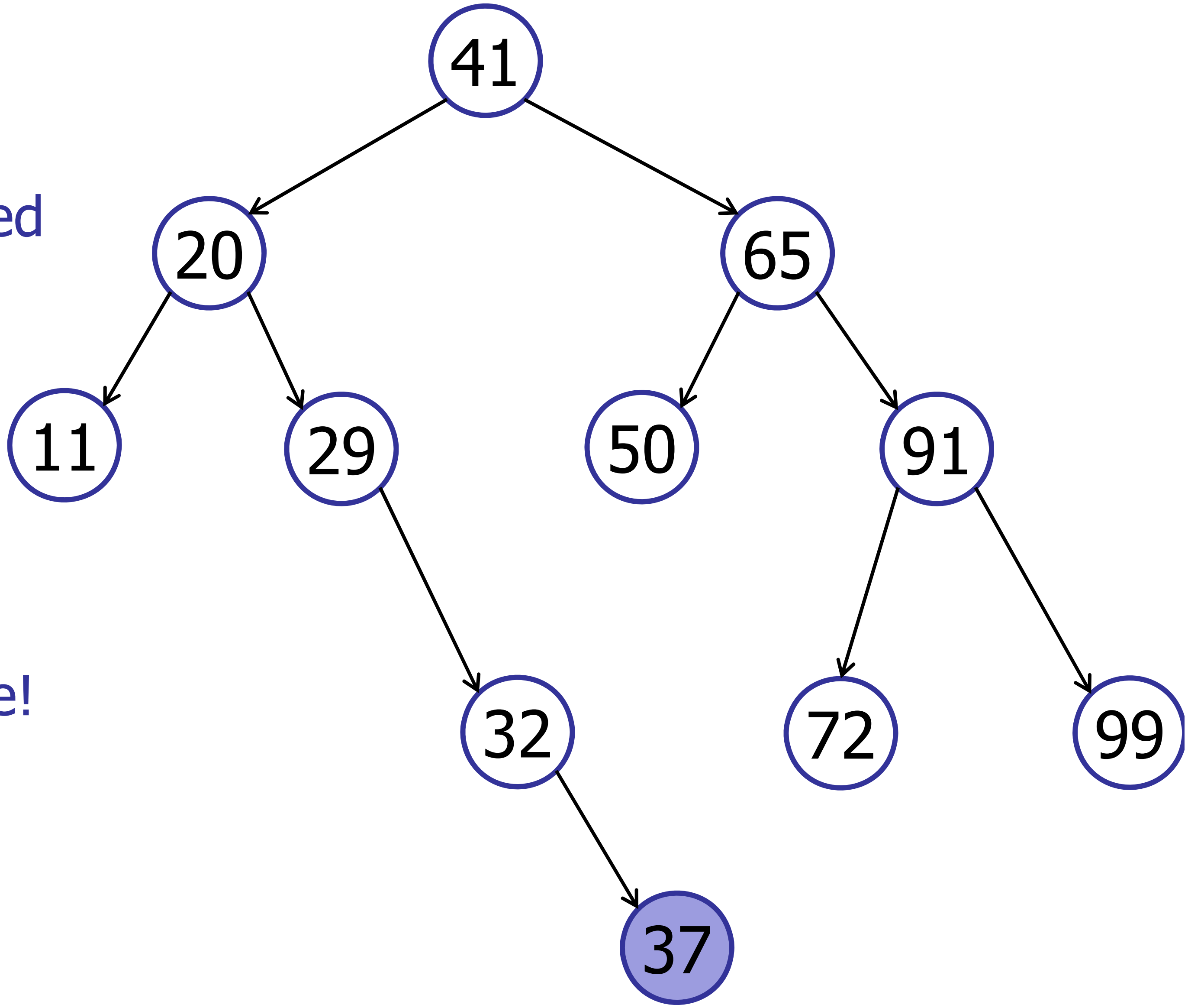


Inserting in an AVL Tree

insert(37)

No longer balanced after insertion!

Need to rebalance!

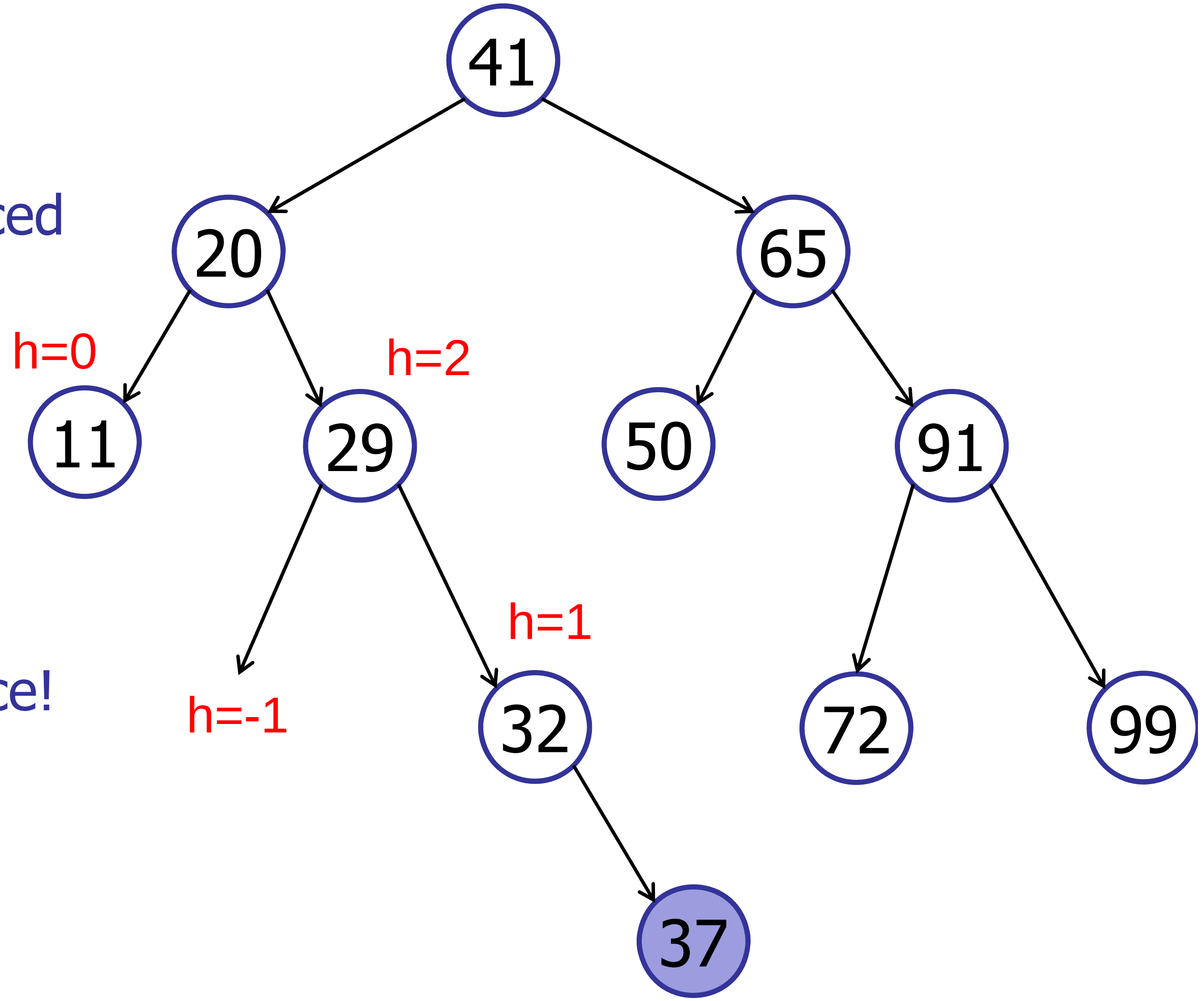


Inserting in an AVL Tree

insert(37)

No longer balanced after insertion!

Need to rebalance!

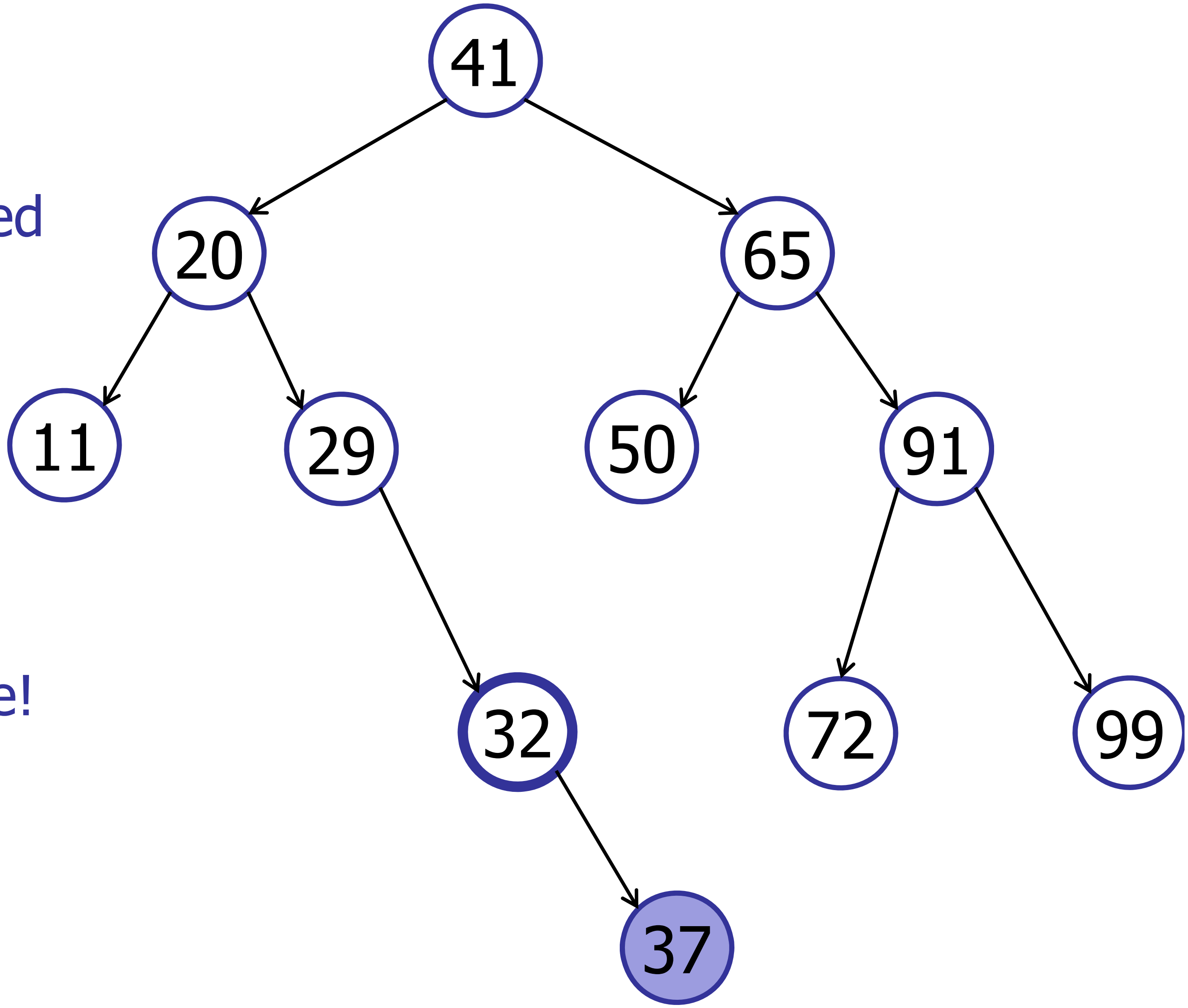


Inserting in an AVL Tree

insert(37)

No longer balanced after insertion!

Need to rebalance!

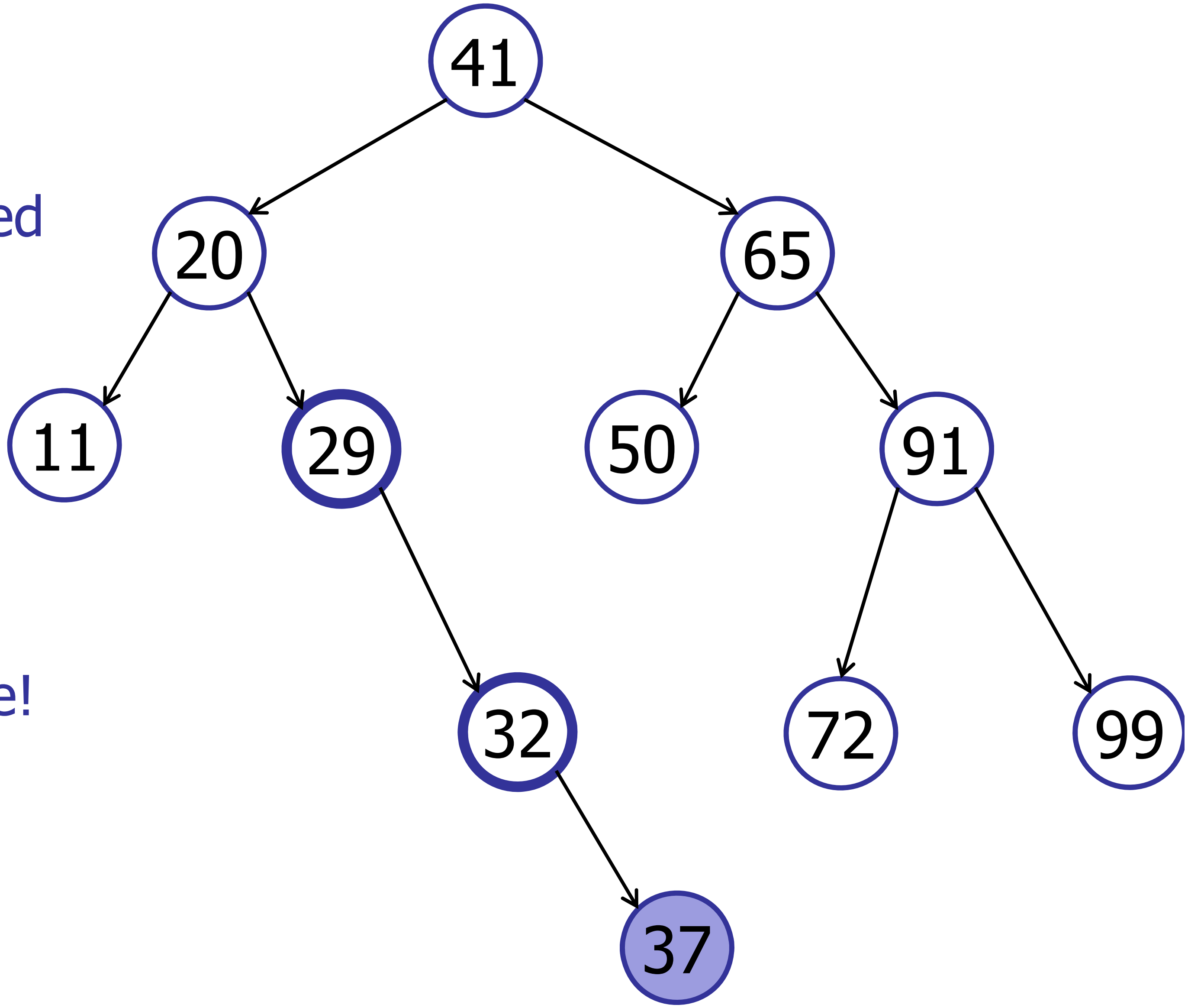


Inserting in an AVL Tree

insert(37)

No longer balanced after insertion!

Need to rebalance!

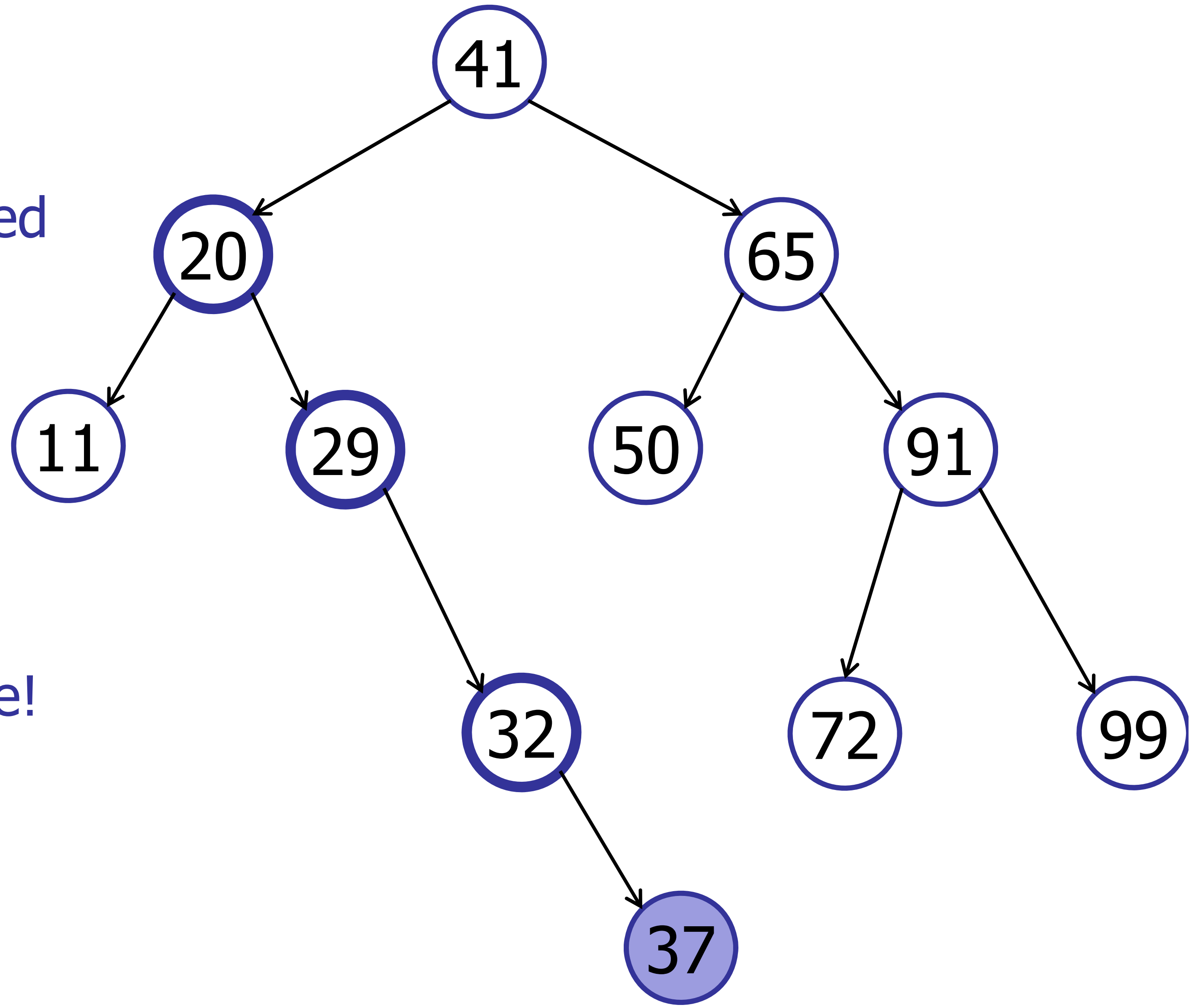


Inserting in an AVL Tree

insert(37)

No longer balanced after insertion!

Need to rebalance!

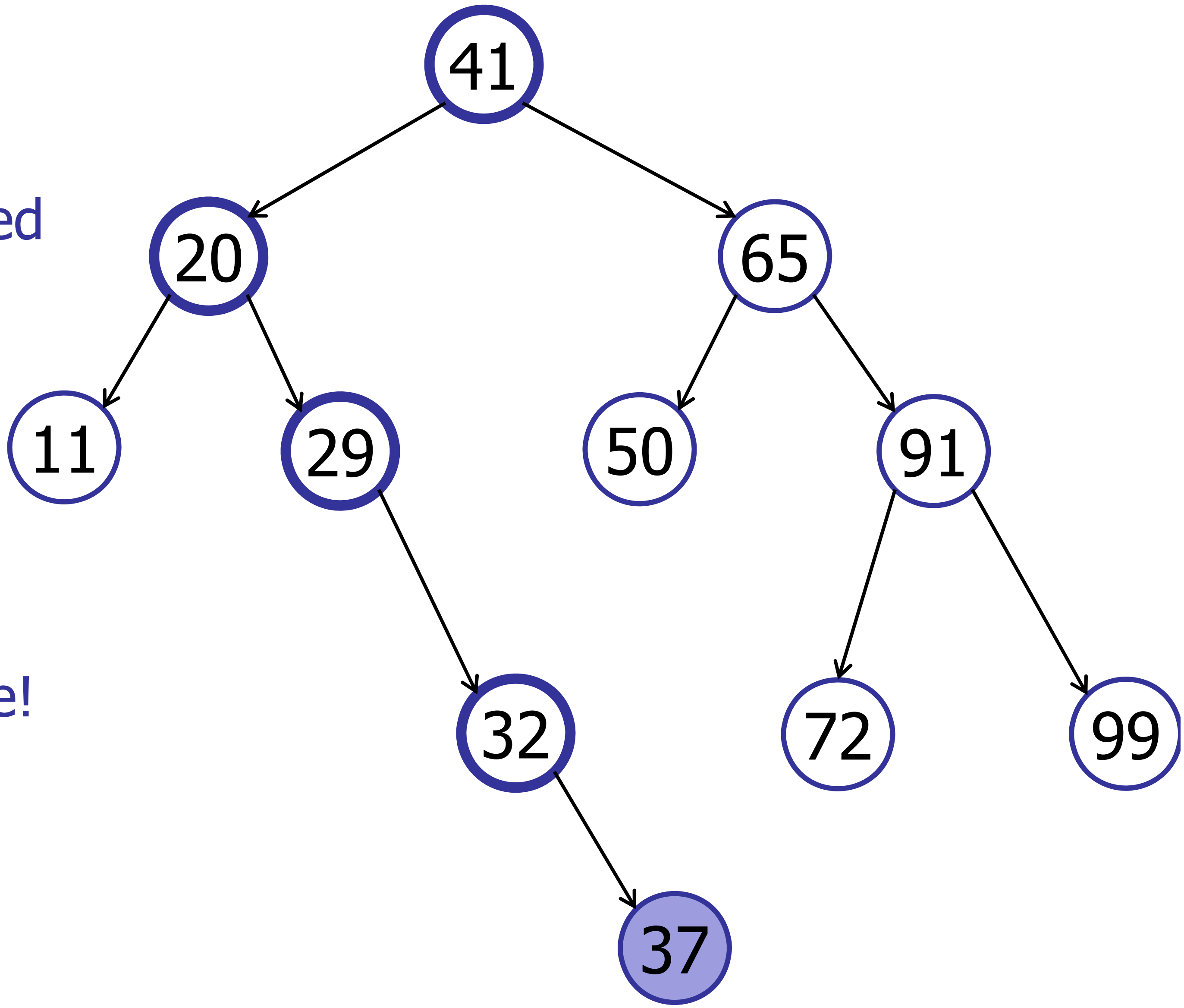


Inserting in an AVL Tree

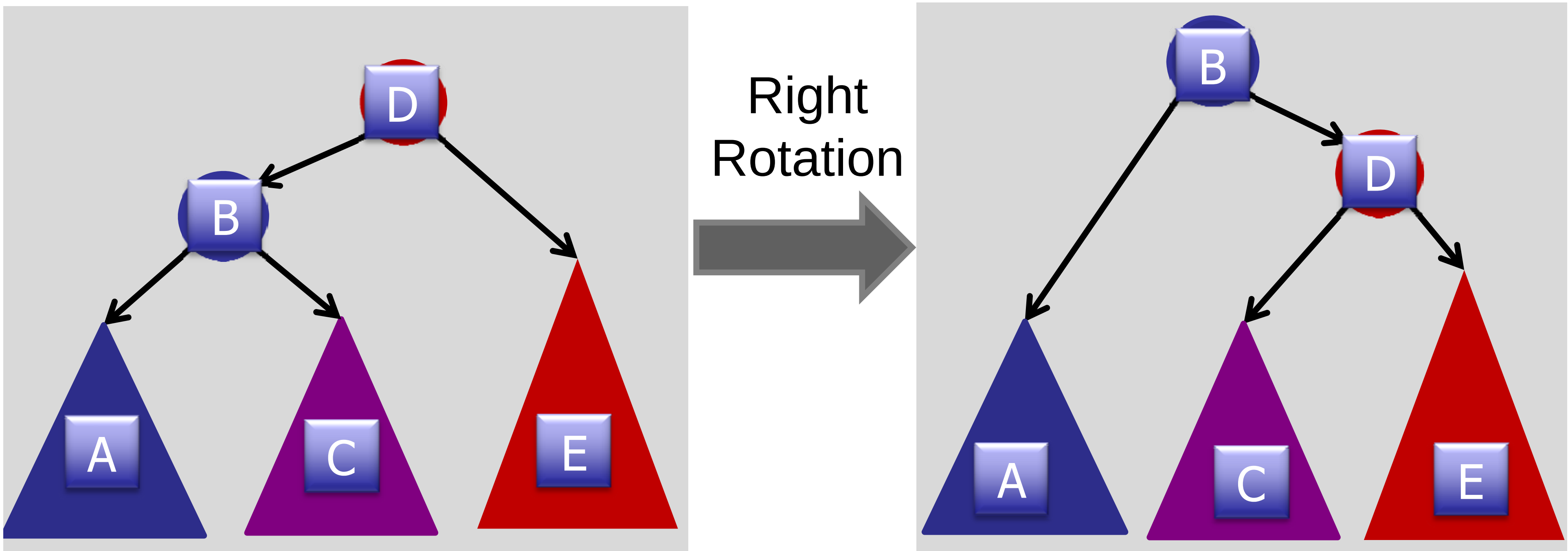
insert(37)

No longer balanced after insertion!

Need to rebalance!

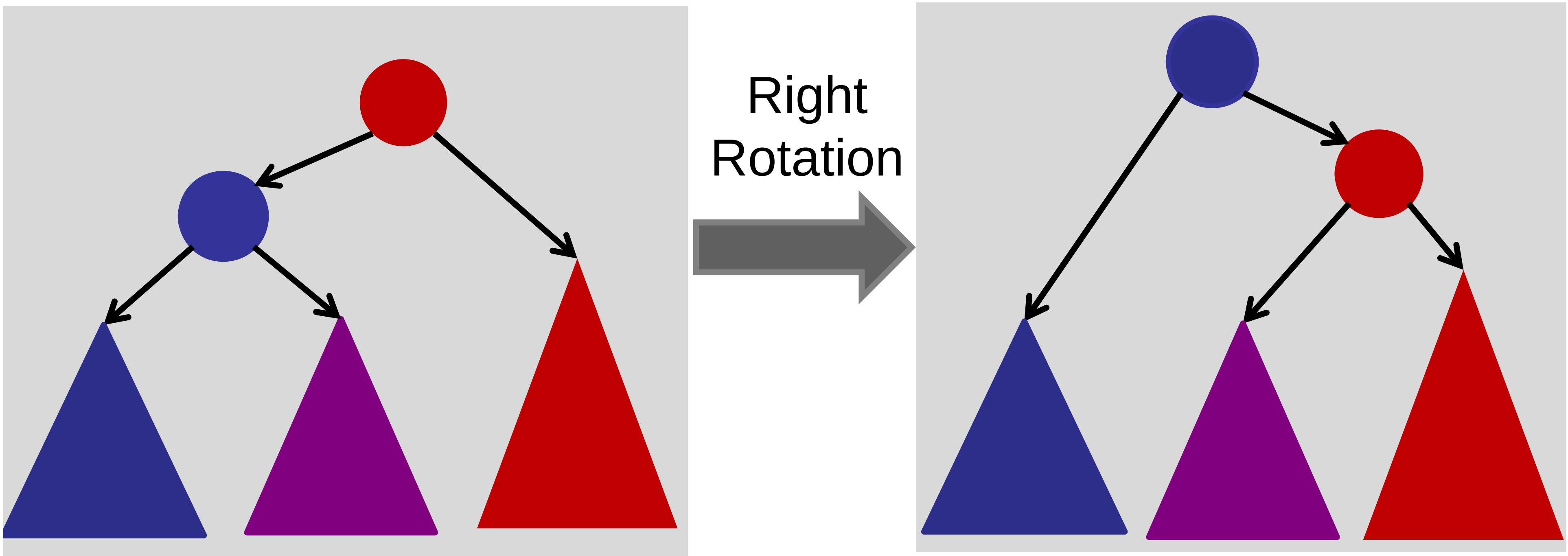


Tree Rotations



$A < B < C < D < E$

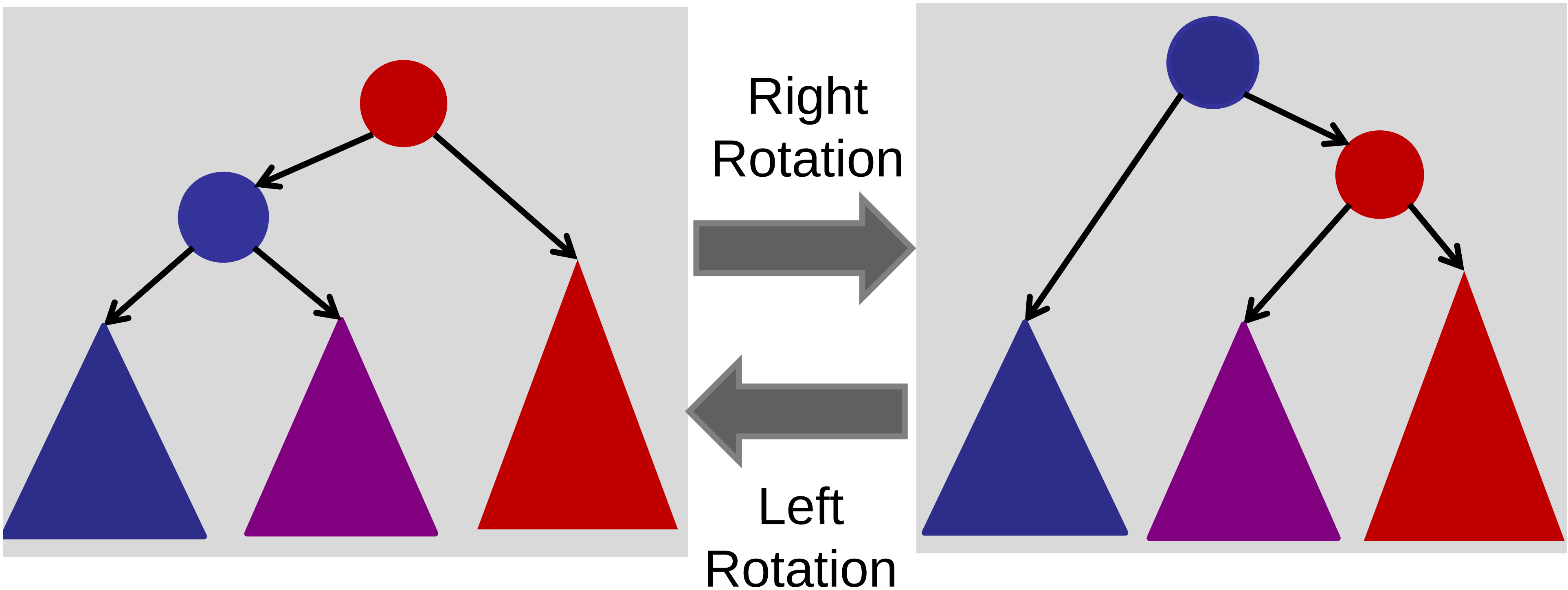
Tree Rotations

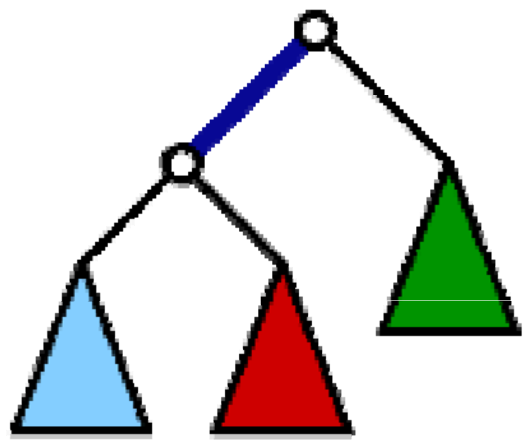


Rotations maintain ordering of keys.

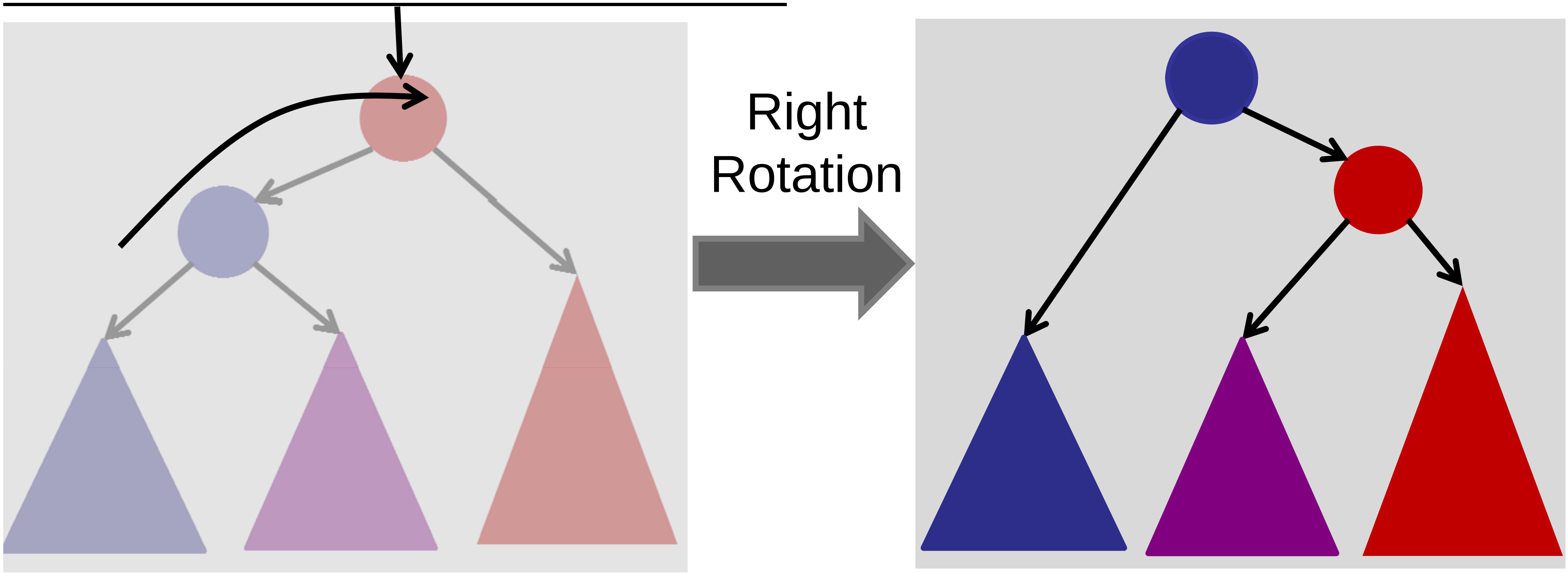
$\Rightarrow$ Maintains BST property.

Tree Rotations



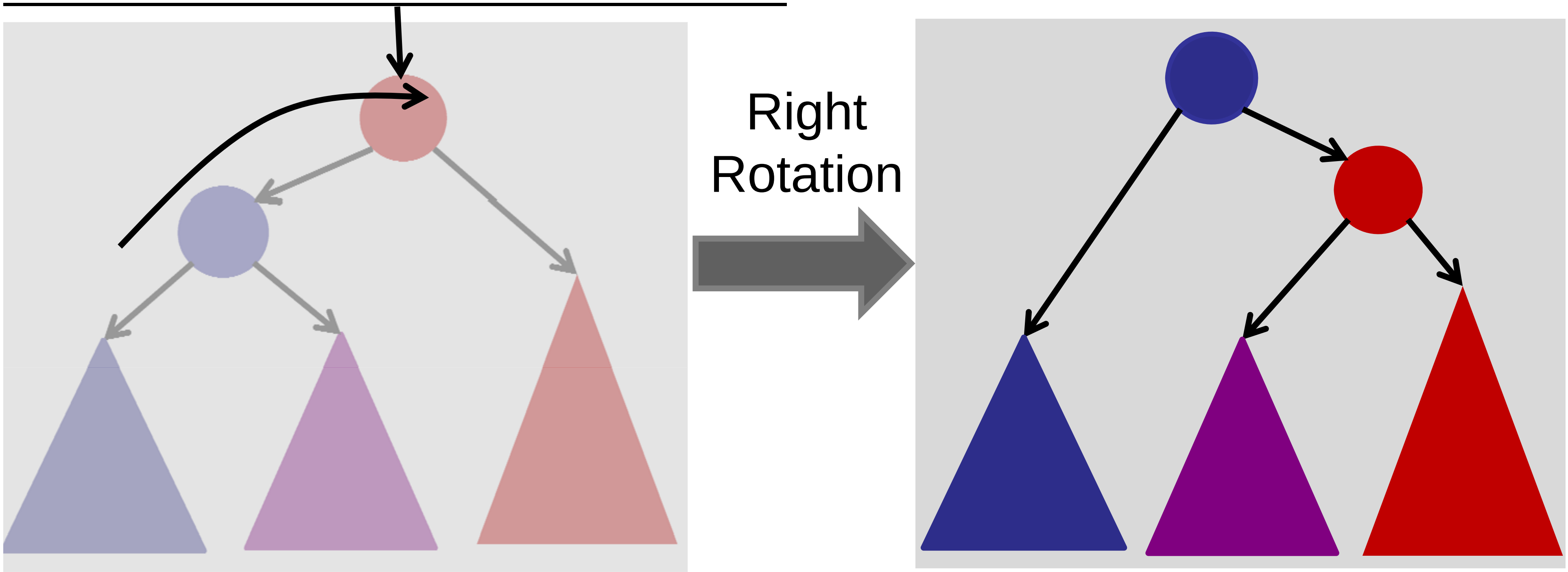


Tree Rotations



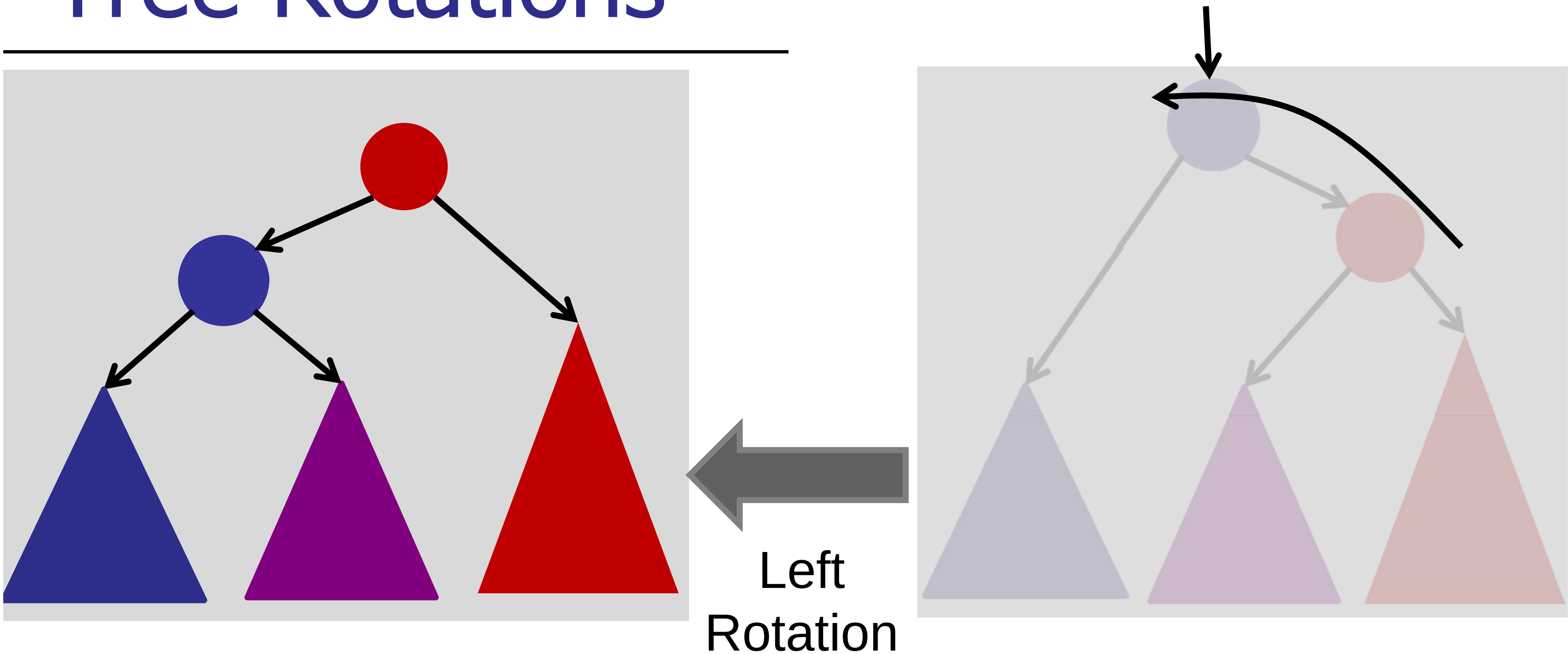
The root of the subtree moves right

Tree Rotations



The root of the subtree moves right

Tree Rotations

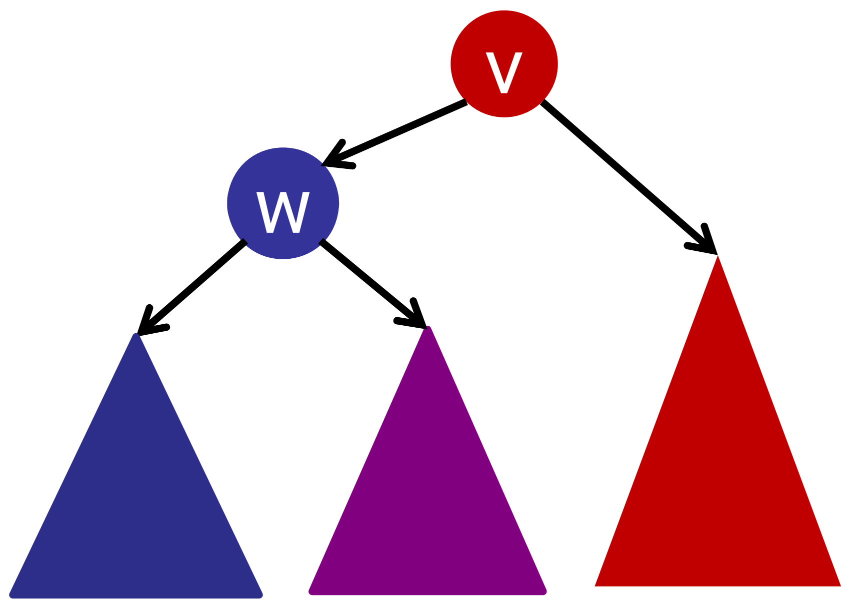


The root of the subtree moves left

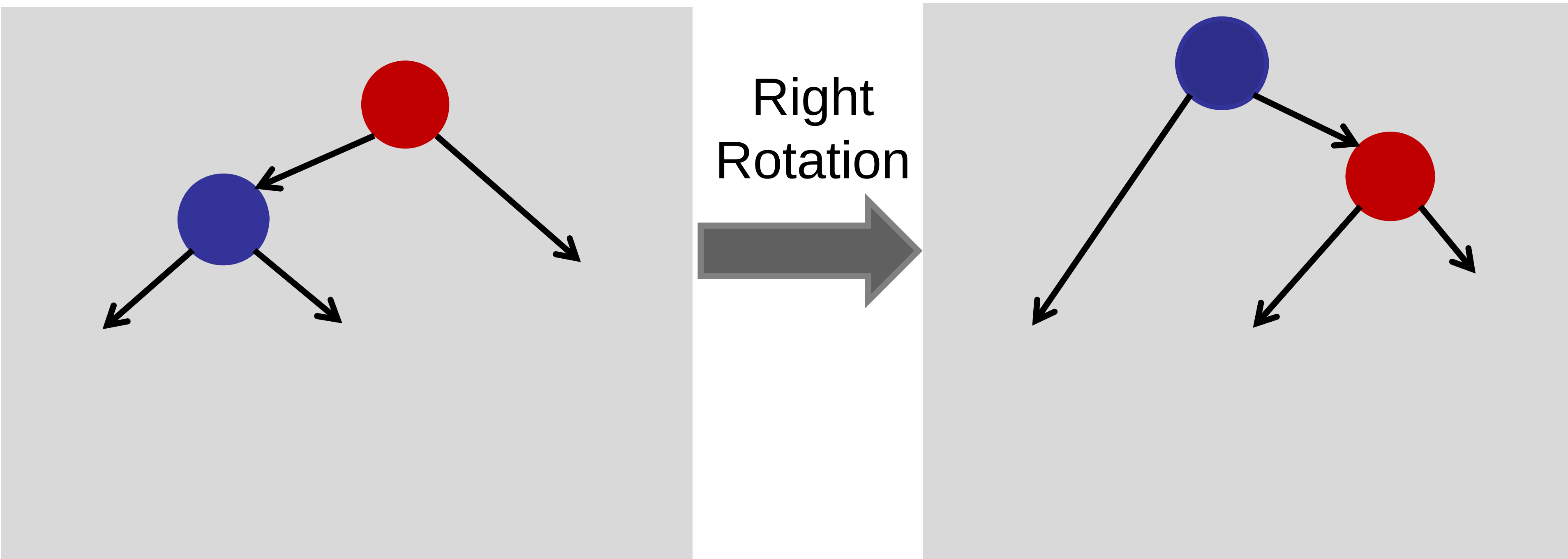
Rotations

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

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



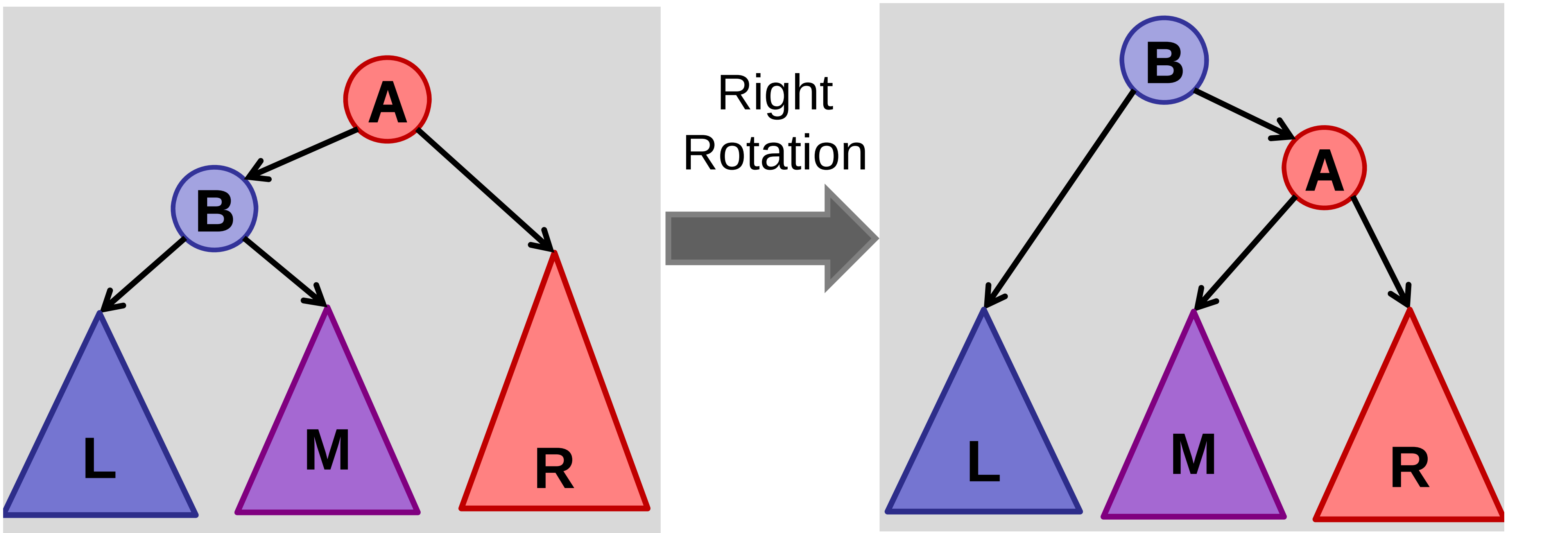
Tree Rotations



rotate-right requires a left child

rotate-left requires a right child

Tree Rotations



After insert:

Use tree rotations to restore balance.

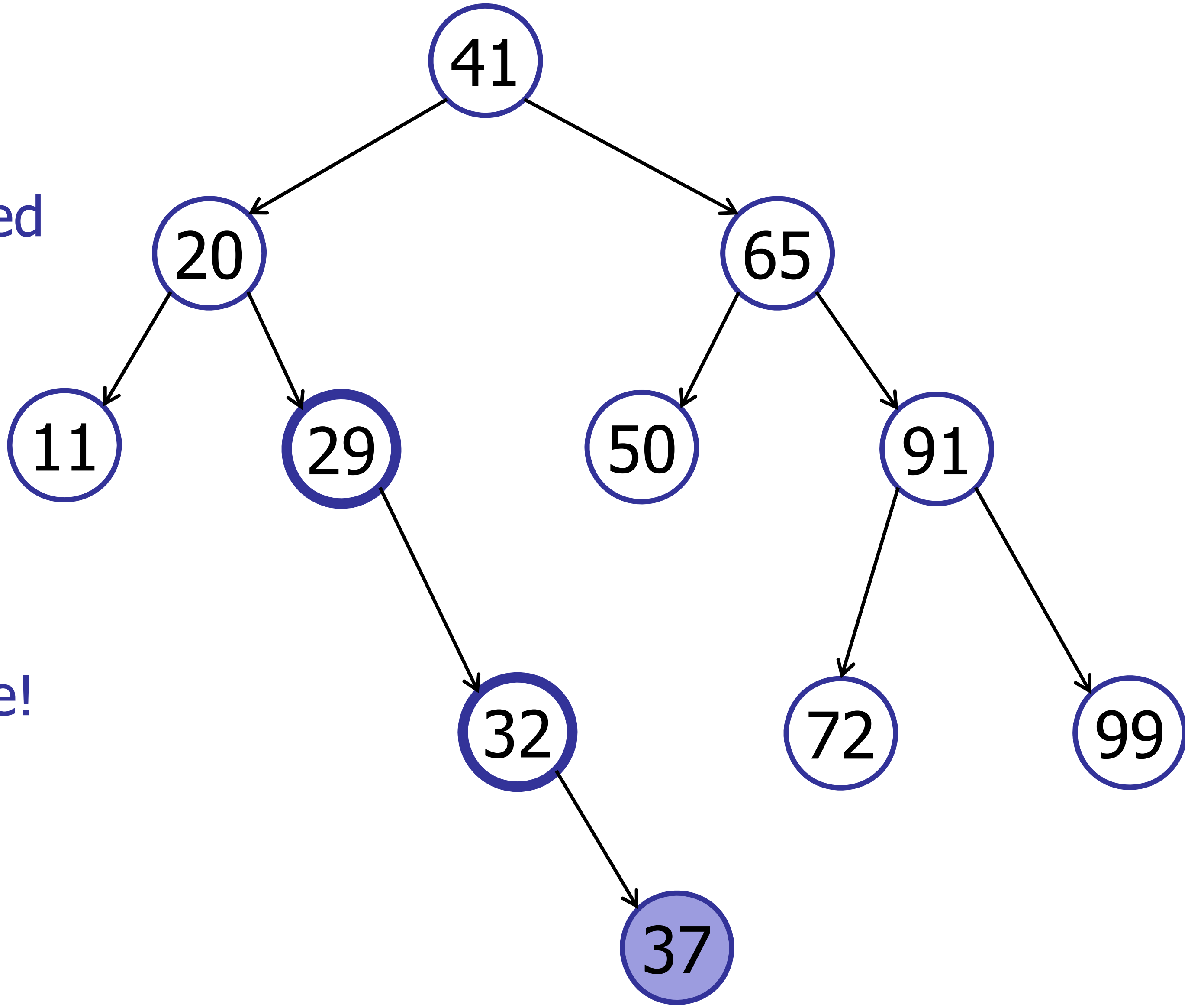
Height is out-of-balance by 1

Inserting in an AVL Tree

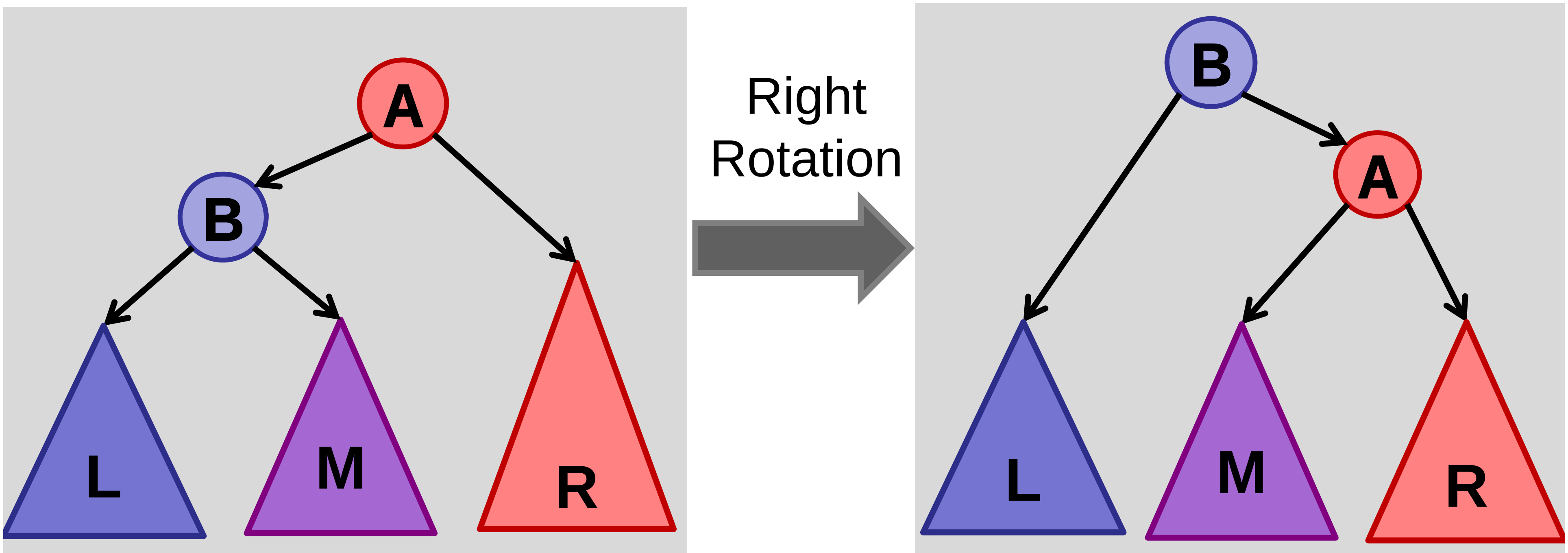
insert(37)

No longer balanced after insertion!

Need to rebalance!



Tree Rotations

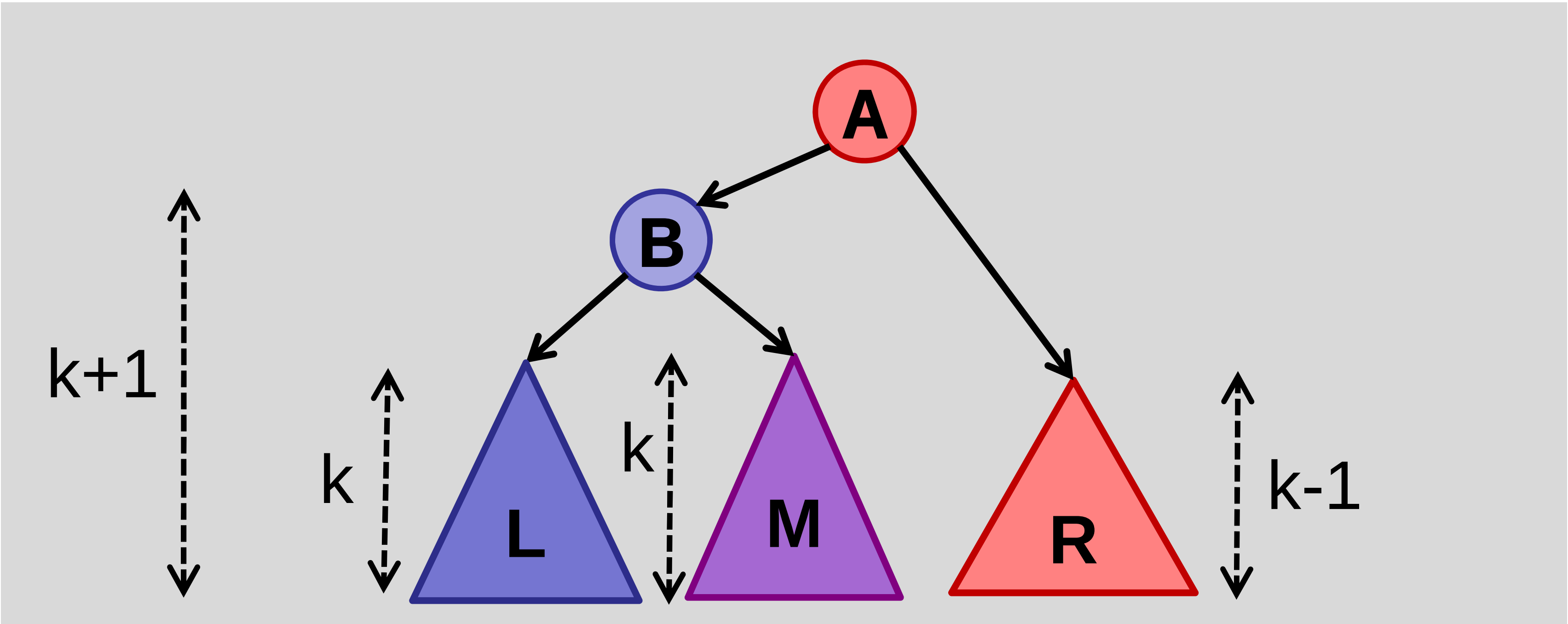


Use tree rotations to restore balance.

After insert, start at bottom, work your way up.

Assume tree is LEFT-heavy.

Tree Rotations

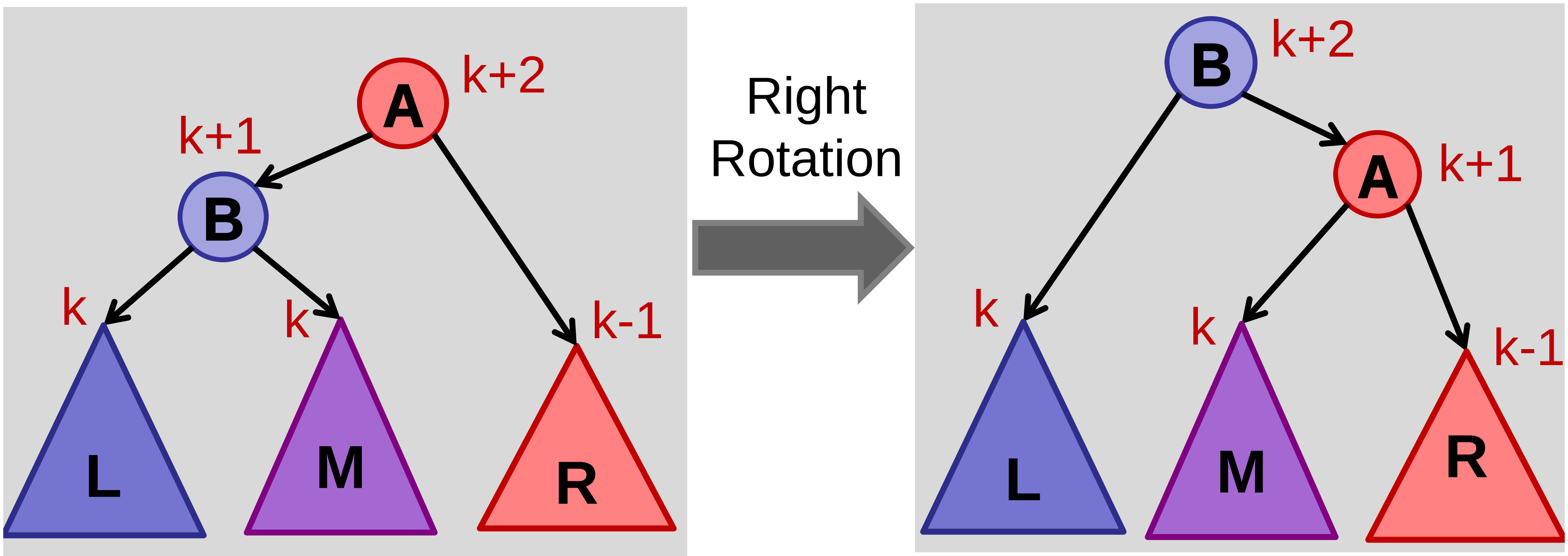


Assume **A** is the lowest node in the tree violating balance property.

Case 1: **B** is balanced : h(**L**) = h(**M**)

h(**R**) = h(**M**) – 1

Tree Rotations

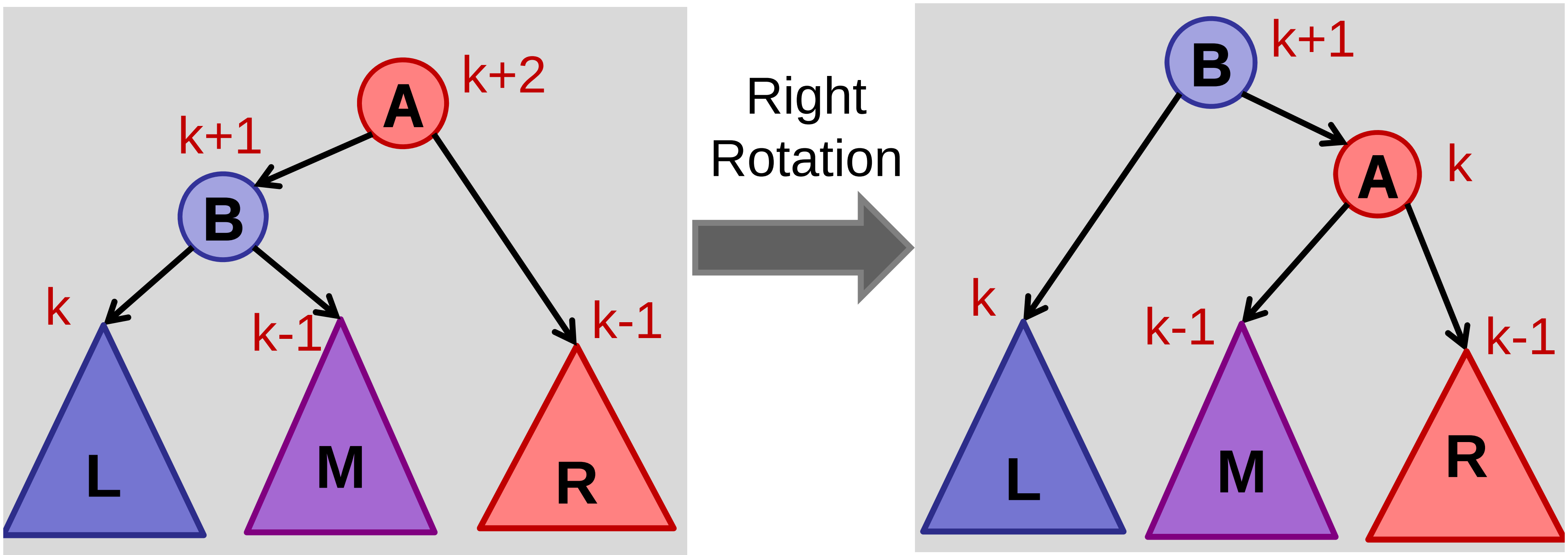


right-rotate:

Case 1: **B** is balanced : h(**L**) = h(**M**)

h(**R**) = h(**M**) – 1

Tree Rotations

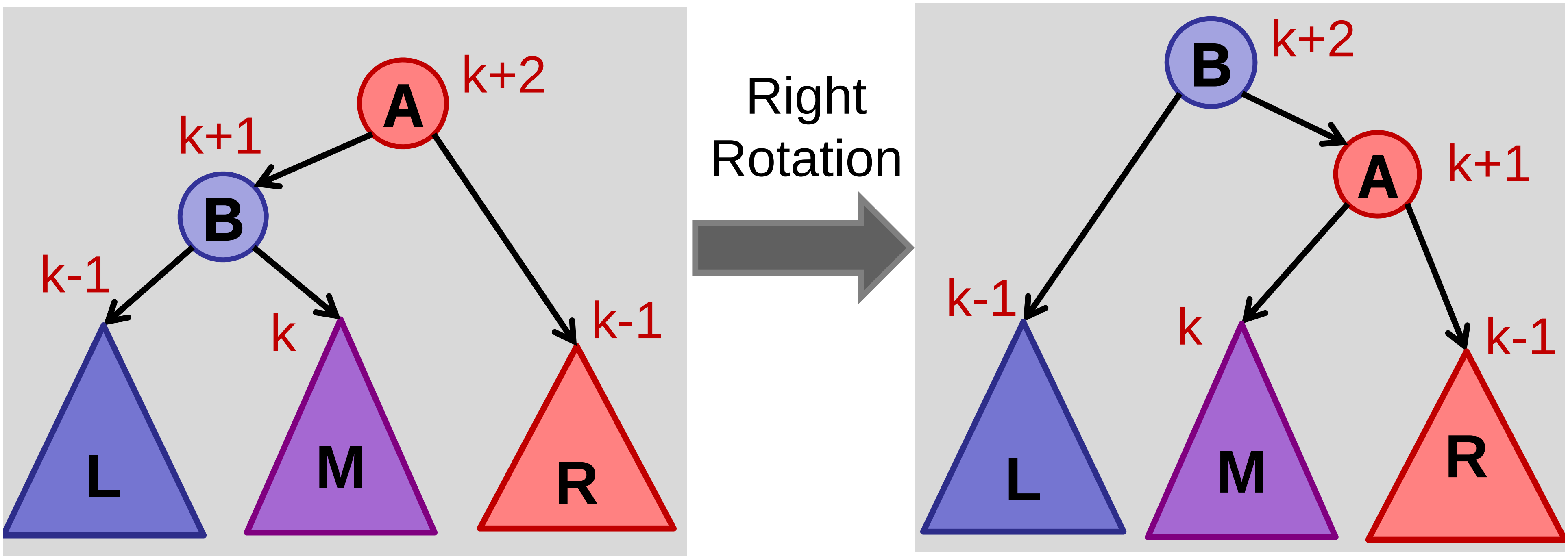


right-rotate:

Case 2: **B** is left-heavy: h(**L**) = h(**M**) +1

h(**R**) = h(**M**)

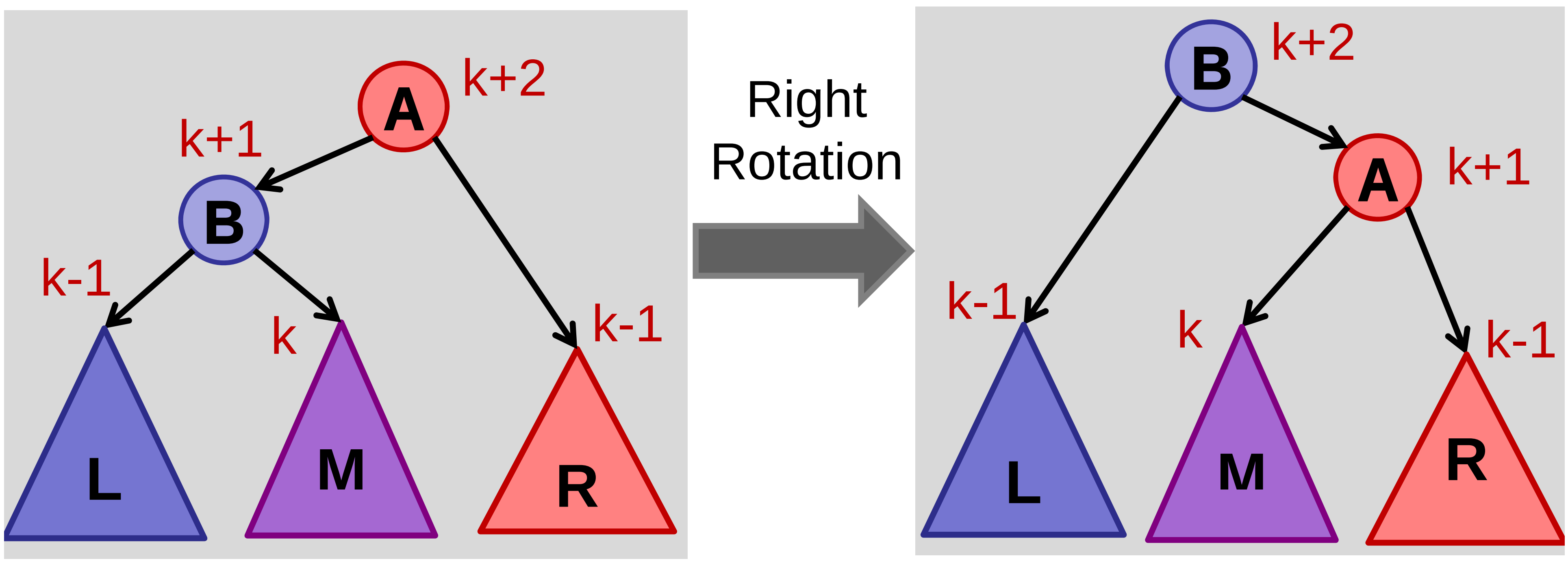
Tree Rotations



right-rotate:

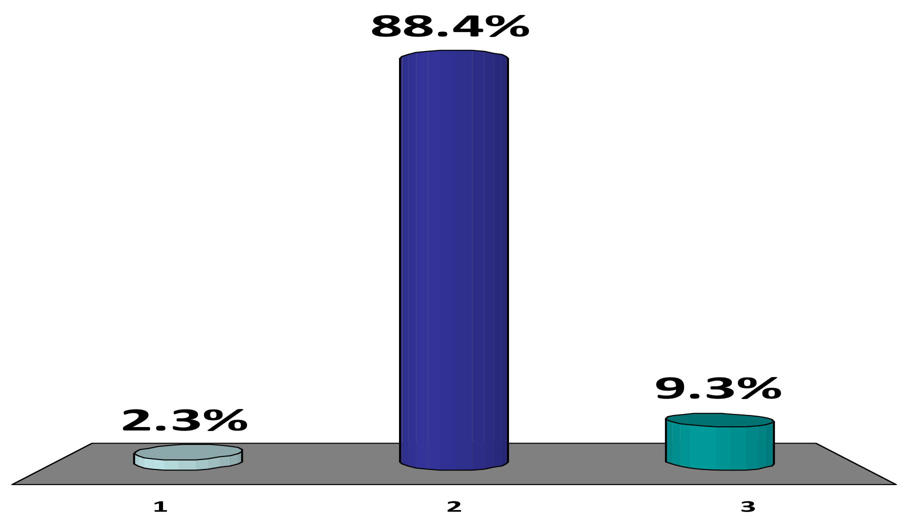
Case 3: **B** is right-heavy: h(**L**) = h(**M**) – 1

h(**R**) = h(**L**)

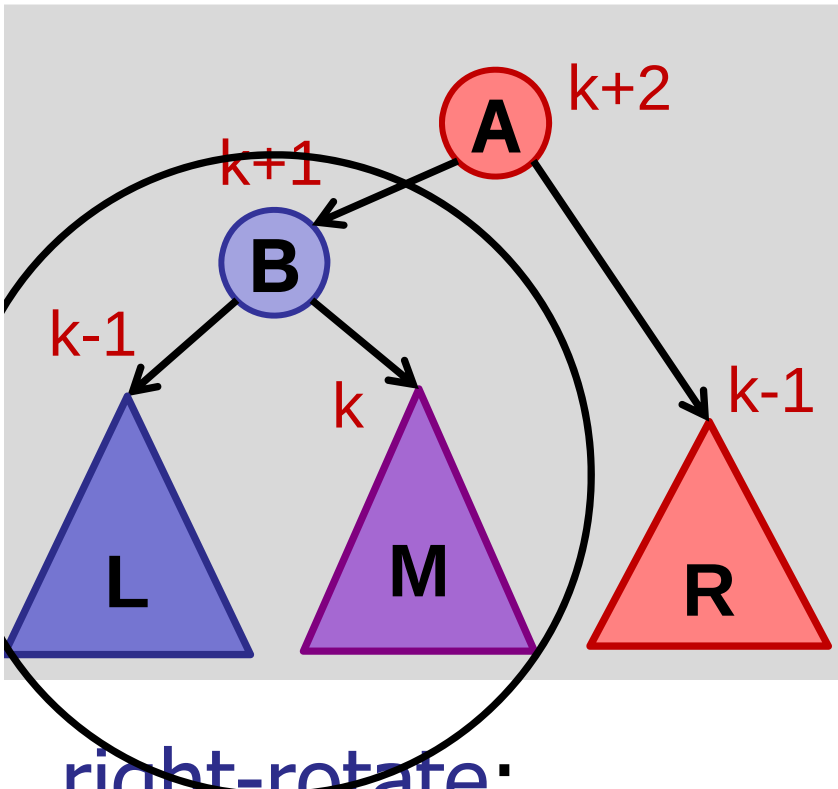


Are we done?

1. Yes.
2. No.
3. Maybe.



Tree Rotations



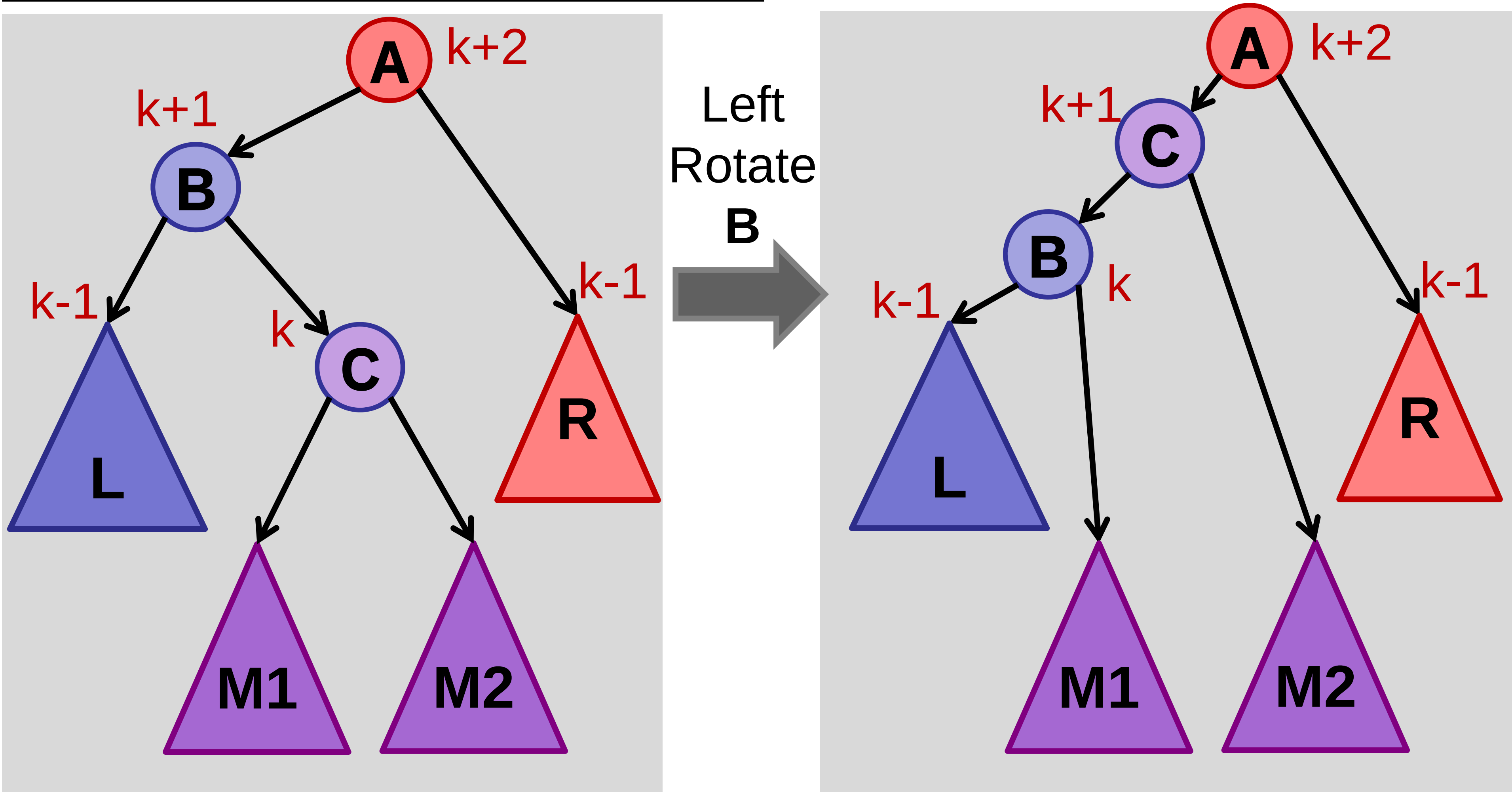
Let’s do something first before we right-rotate(A)

right-rotate:

Case 3: **B** is right-heavy: h(**L**) = h(**M**) – 1

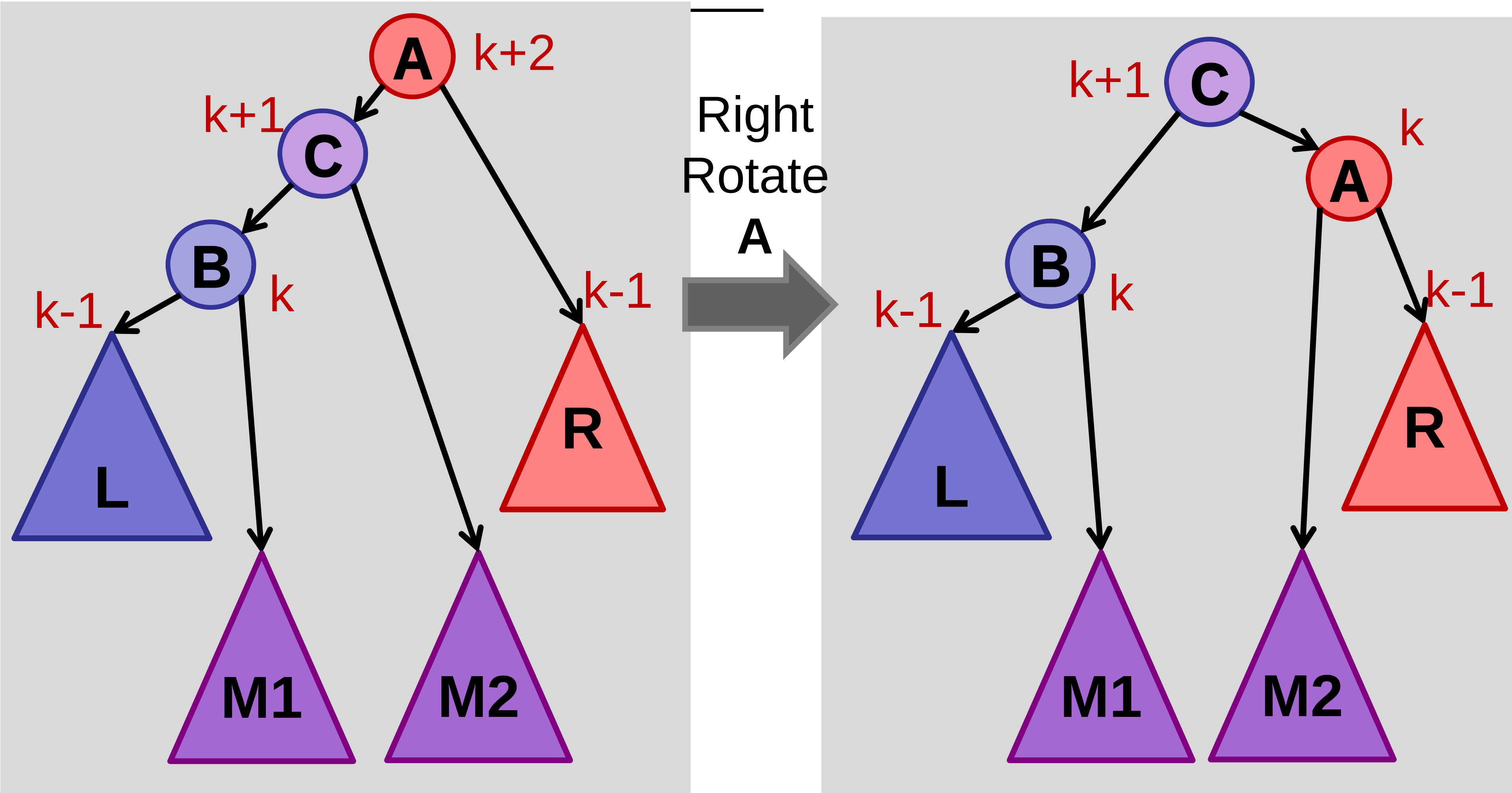
h(**R**) = h(**L**)

Tree Rotations



After left-rotate B: **A** and **C** still out of balance.

Tree Rotations



After right-rotate A: all in balance.

Rotations

Summary:

If v is out of balance and left heavy:

1. v.left is balanced: right-rotate(v)
2. v.left is left-heavy: right-rotate(v)
3. v.left is right-heavy: left-rotate(v.left)
   right-rotate(v)

If v is out of balance and right heavy:

Symmetric three cases….

Insert in AVL Tree

Summary:

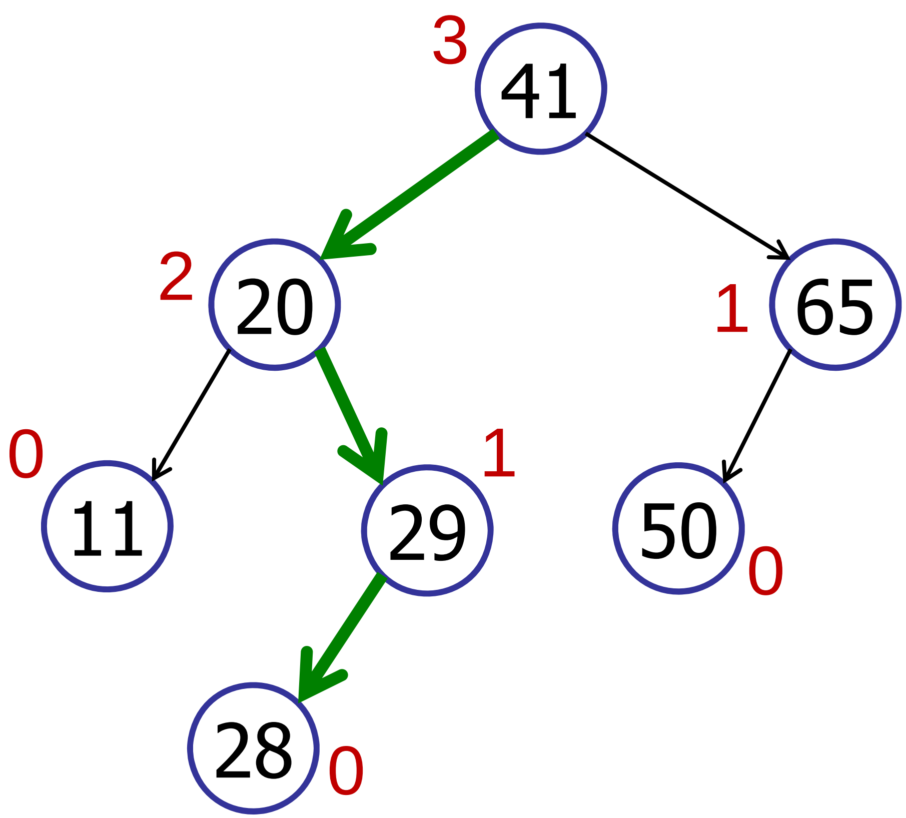
- Insert key in BST.
- Walk up tree:
  - At every step, check for balance.
  - If out-of-balance, use rotations to rebalance.

Note: only need to perform two rotations

- Why?
- In each case, reduce height of sub-tree by 1
- What about Case 1, above?

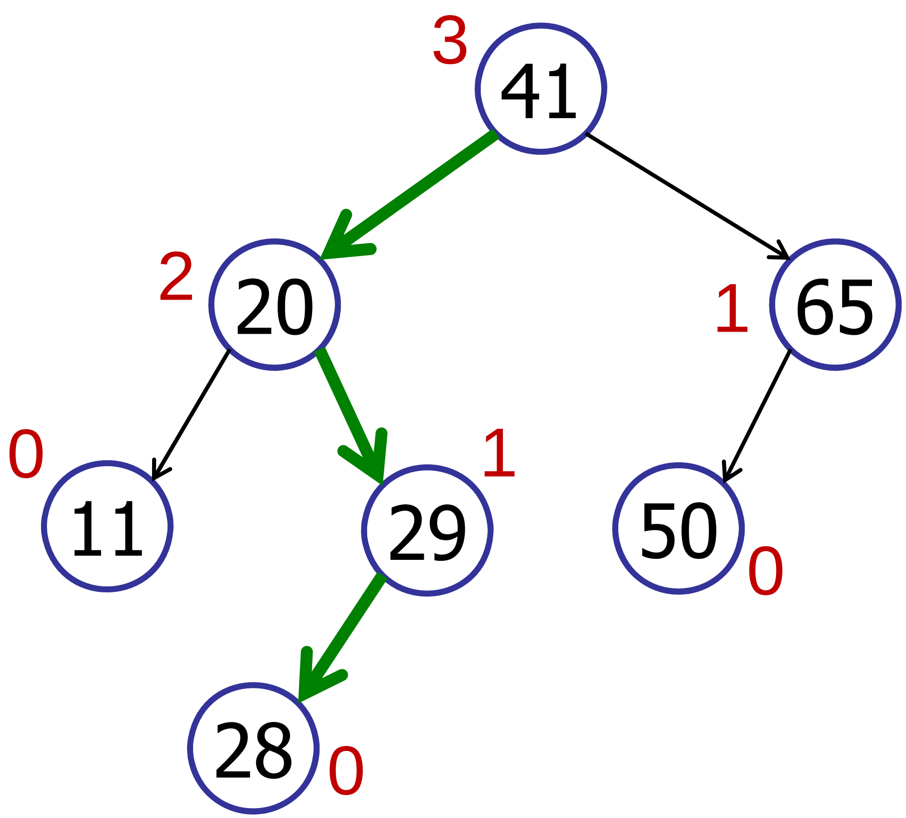
Example

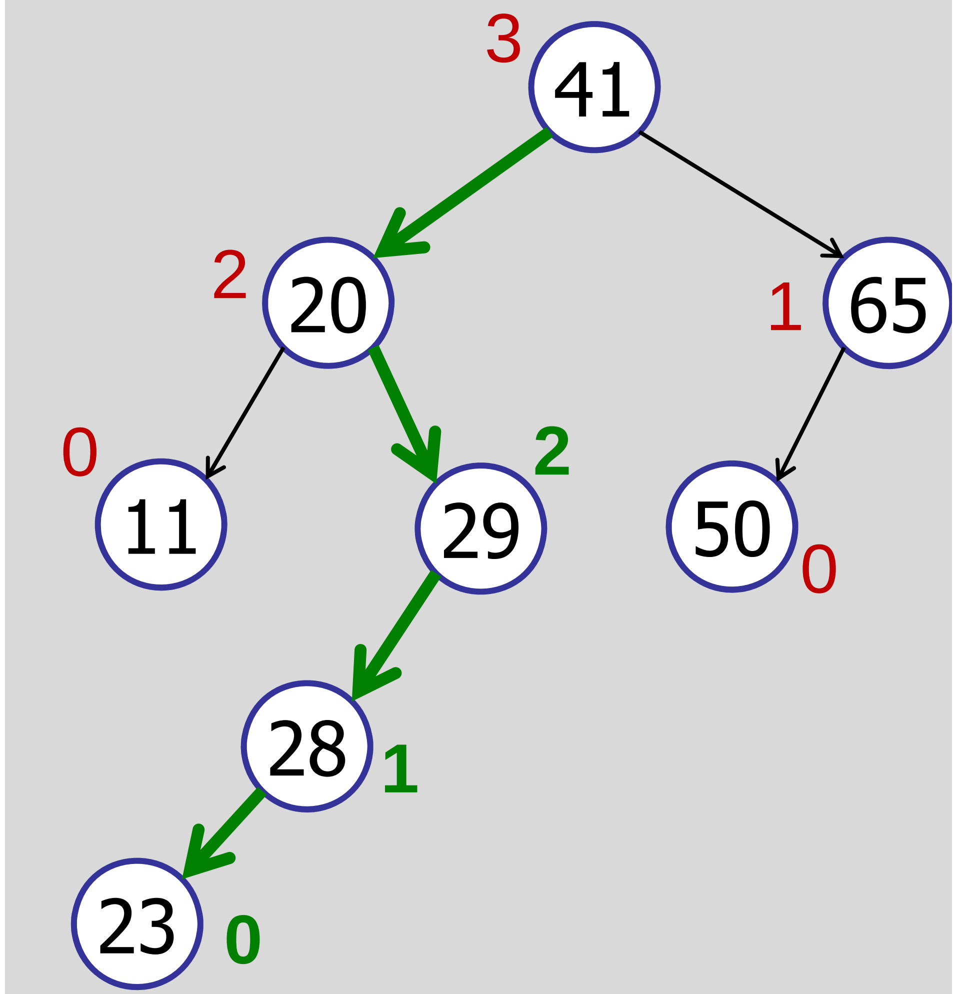
insert(23)



Example

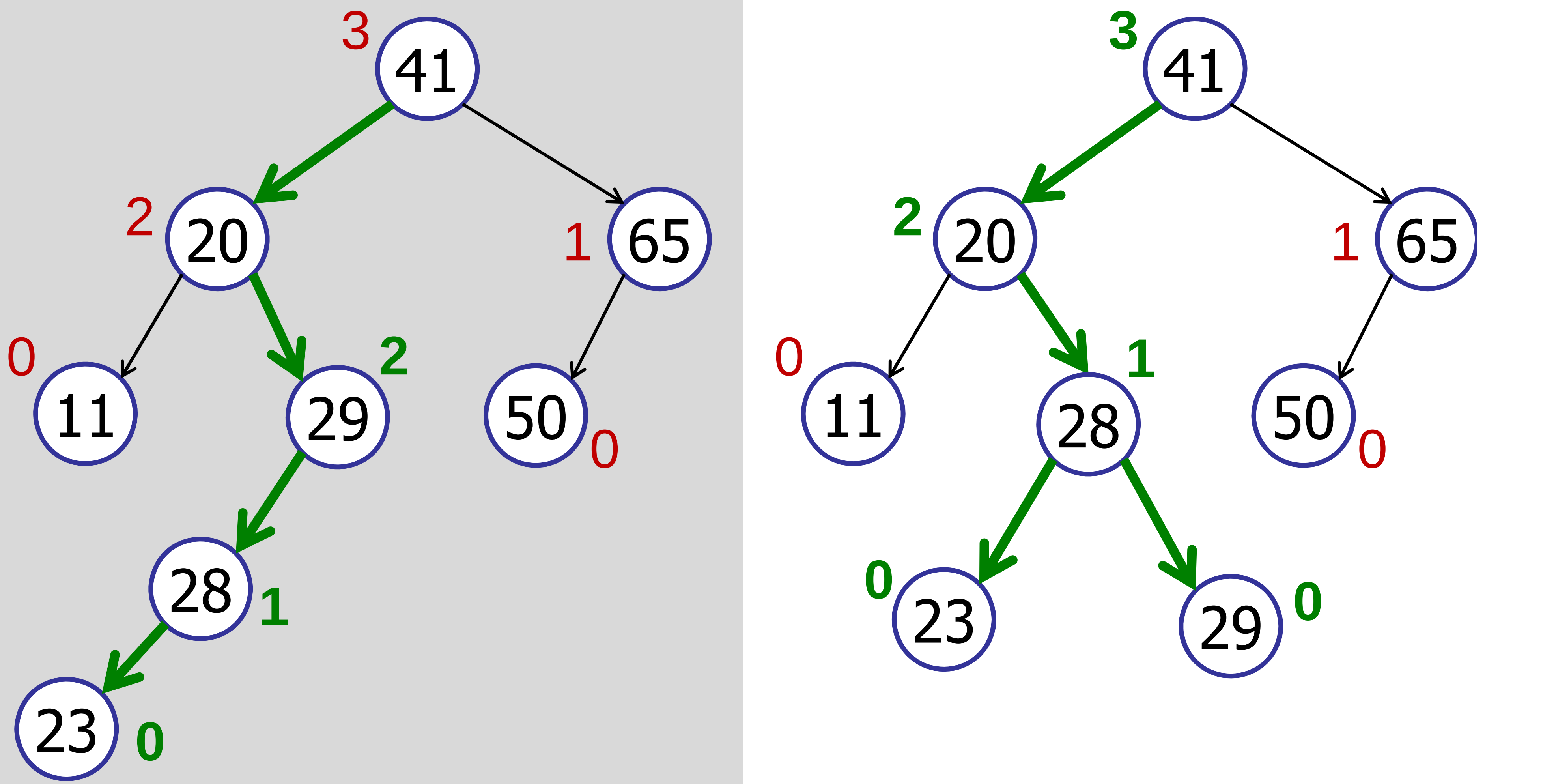
insert(23)





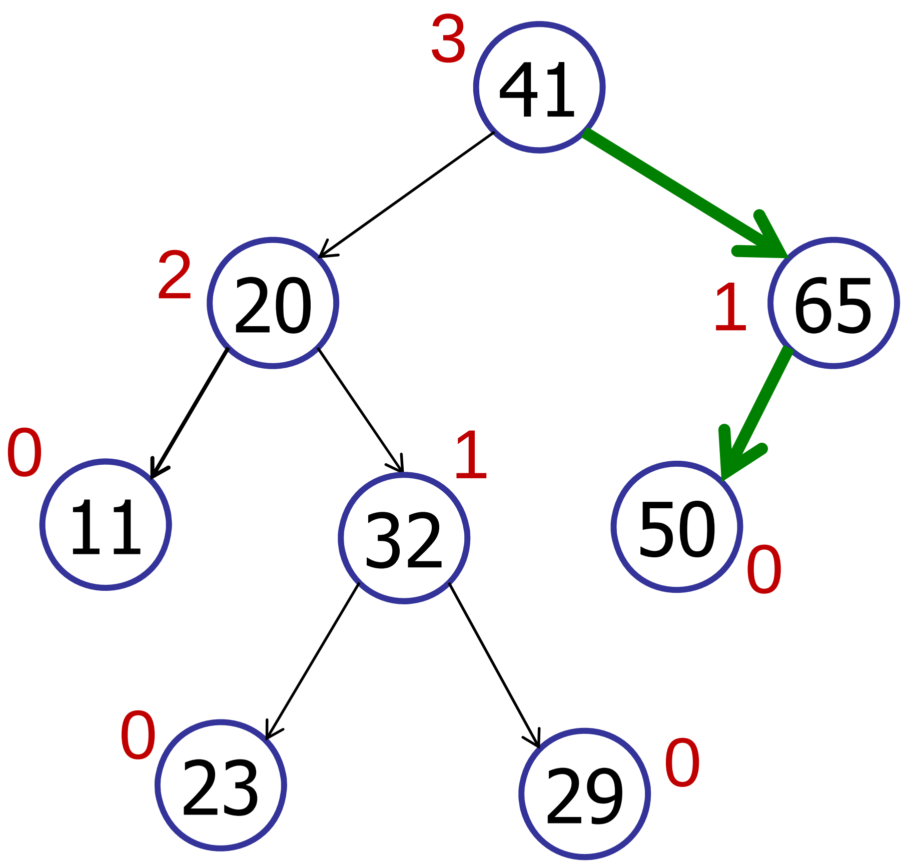
Example

right-rotate(29)



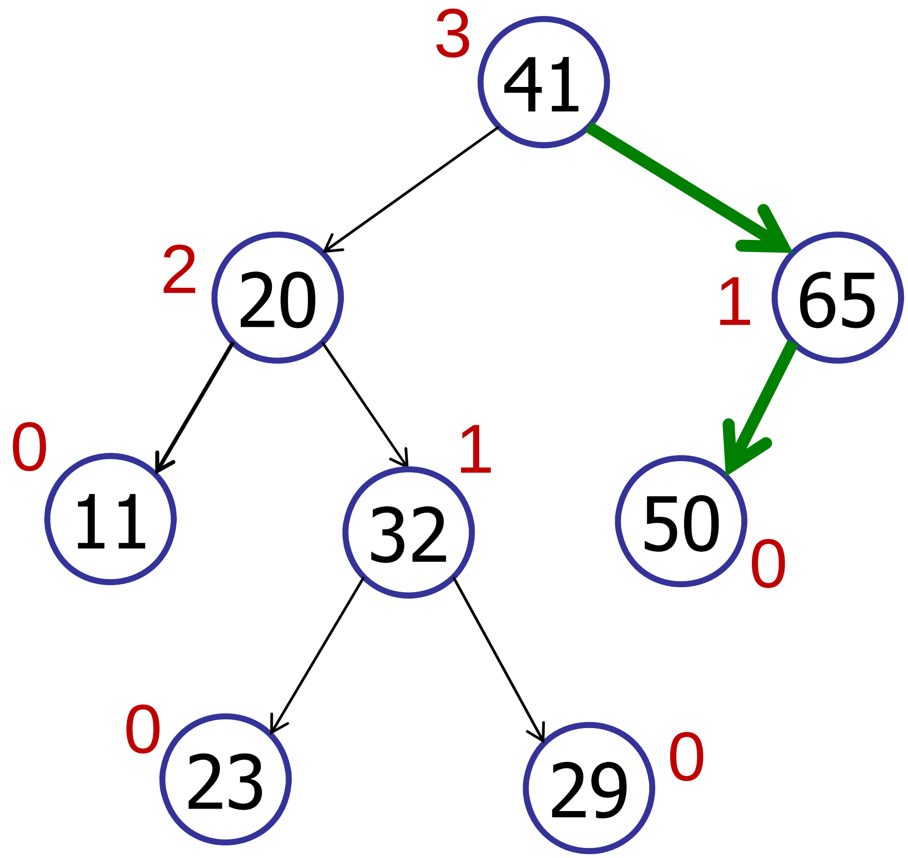
Example

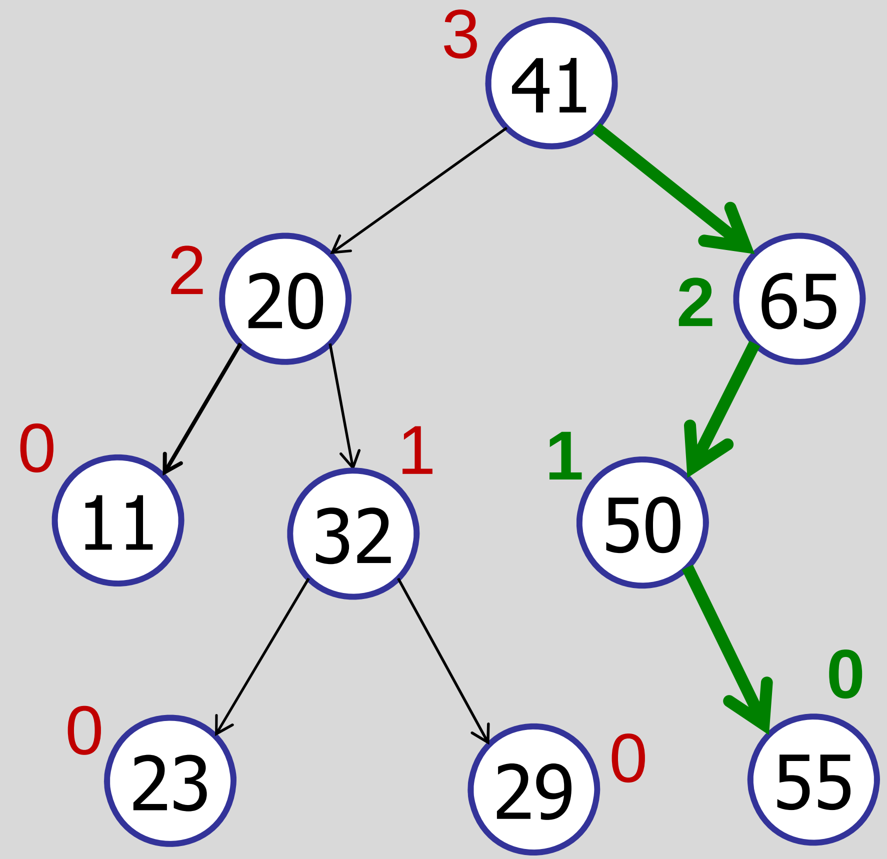
insert(55)



Example

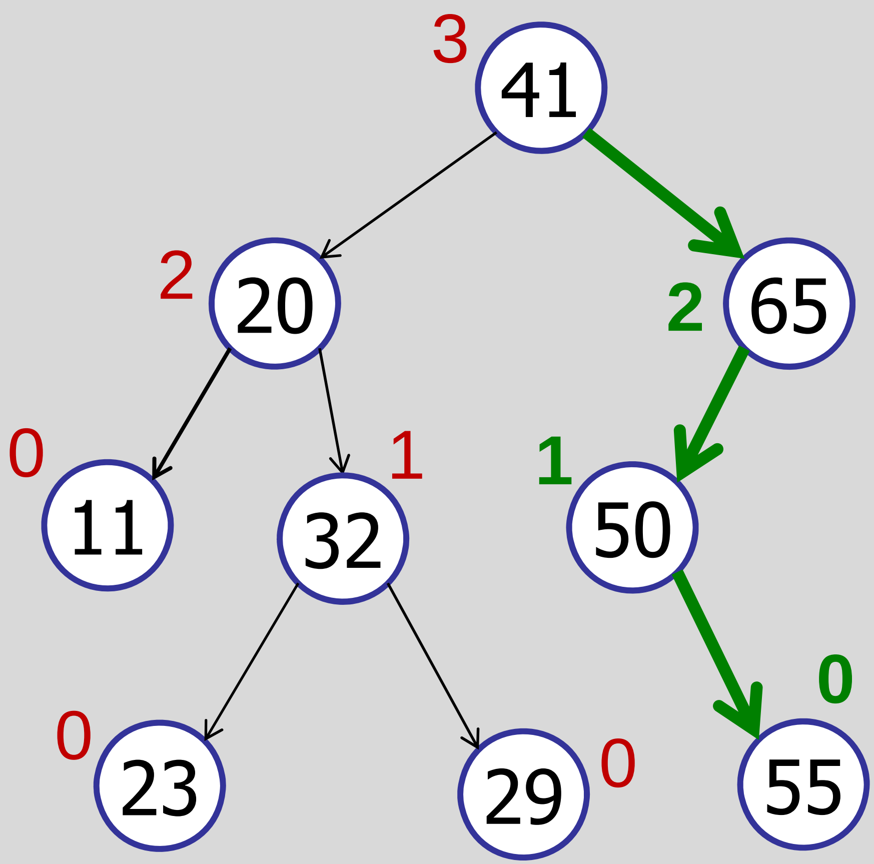
insert(55)

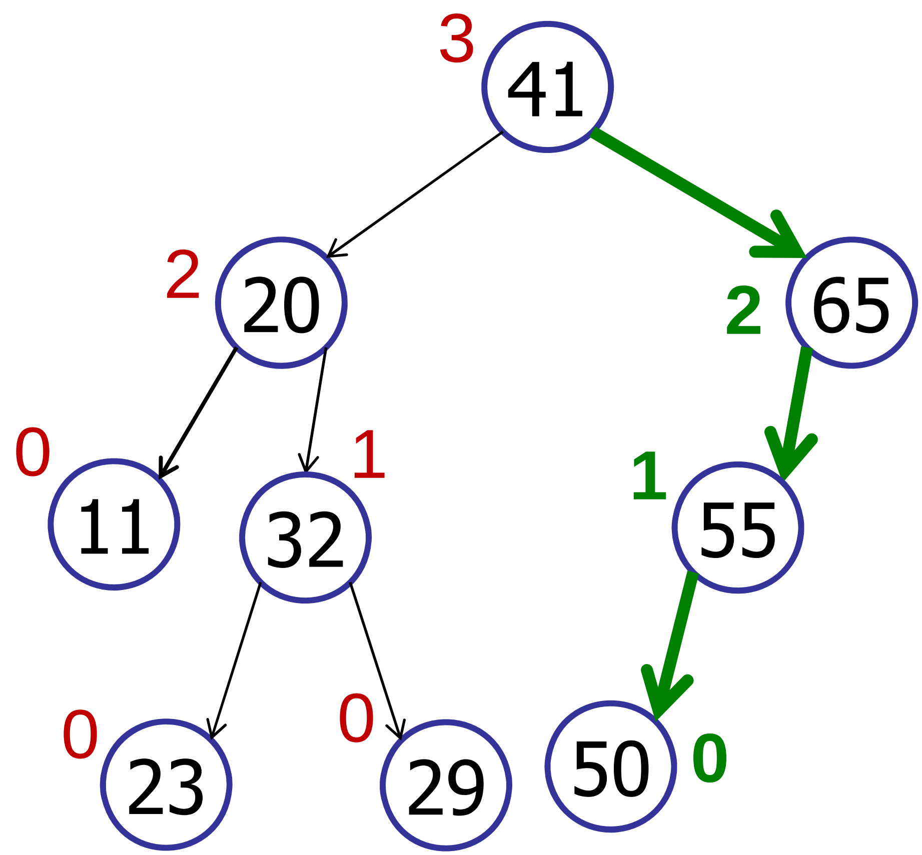




Example

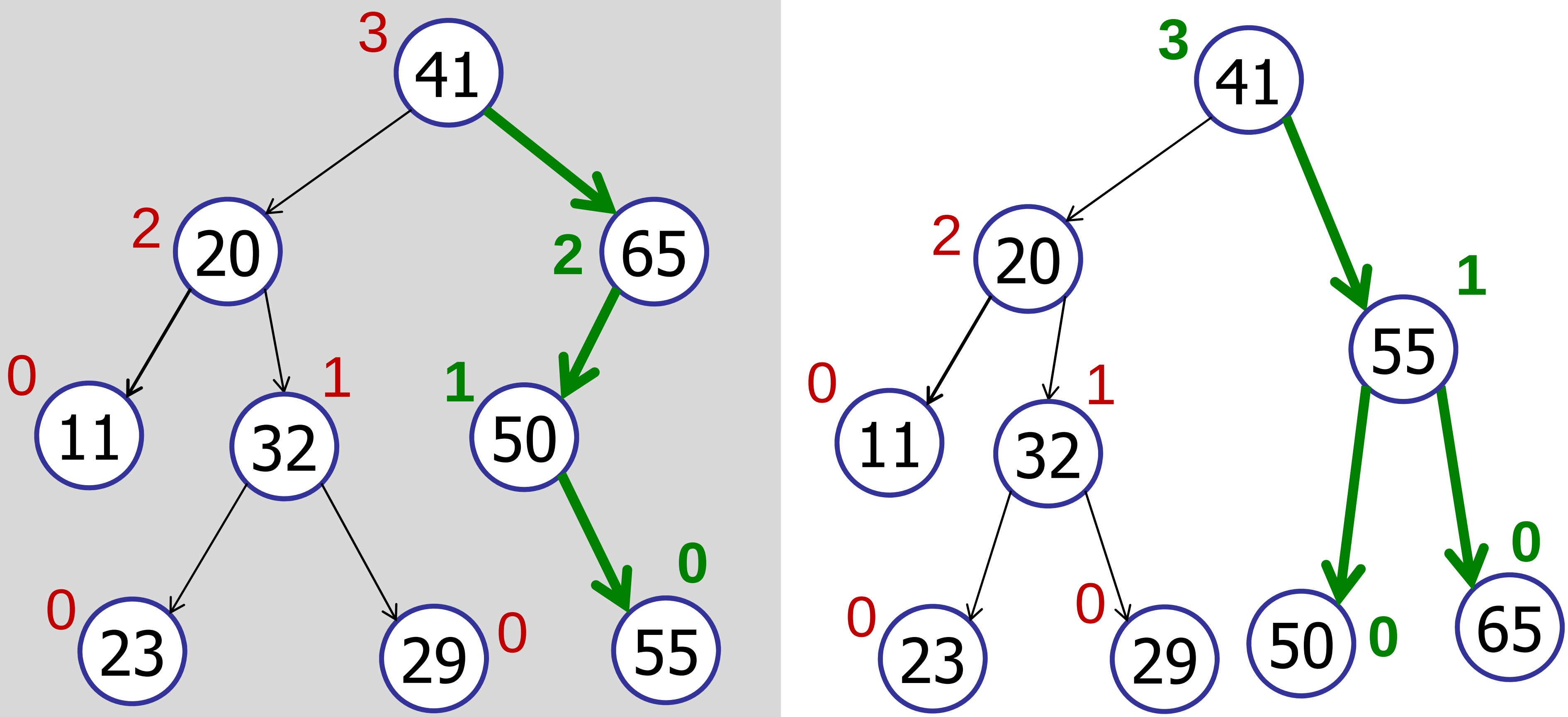
left-rotate(50)





Example

right-rotate(65)

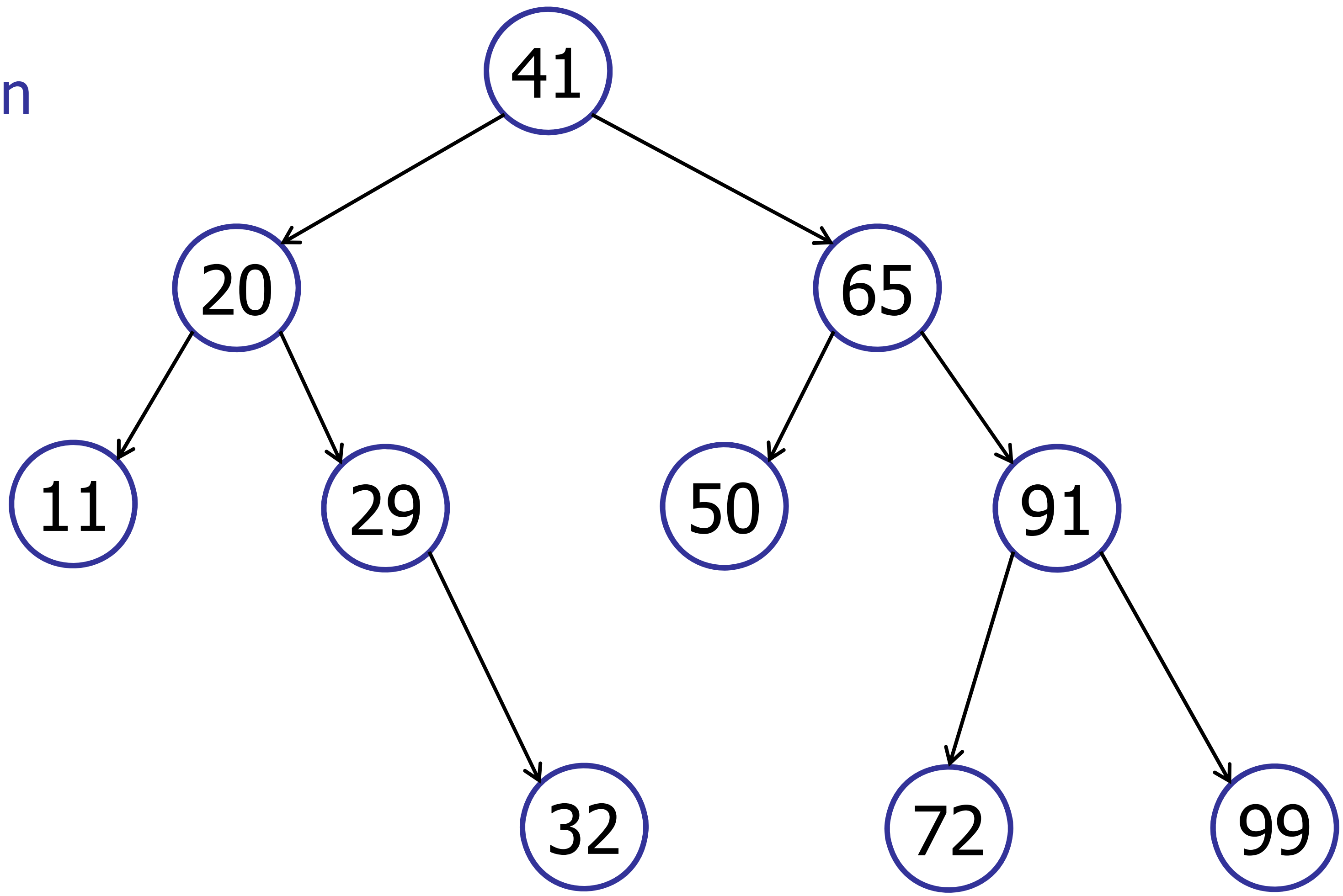


Binary Search Tree

delete(v)

Three cases:

1. No children
2. 1 child
3. 2 children



Binary Search Tree

delete(v)

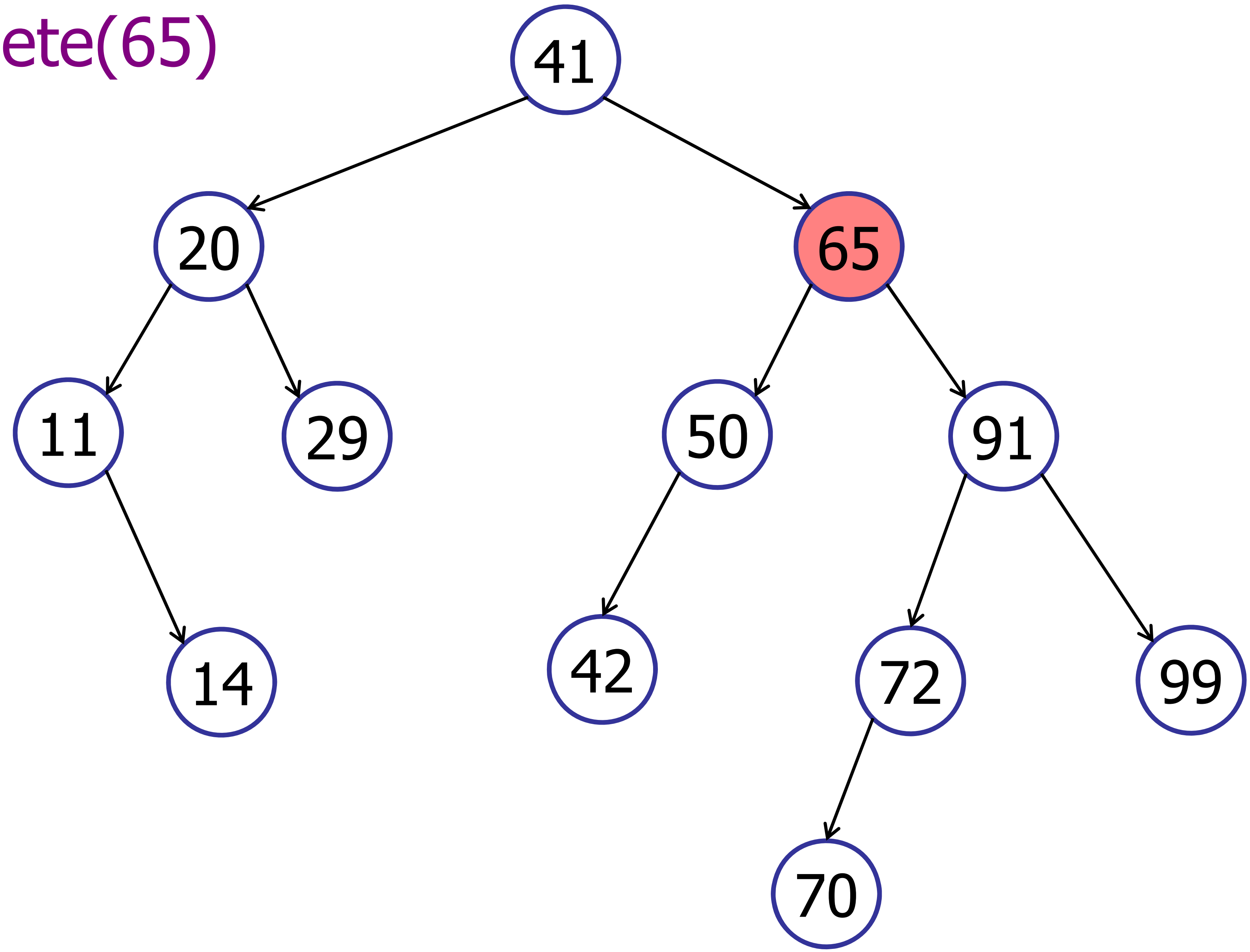
Three cases:

1. No children:
   - remove v
2. 1 child:
   - remove v
   - connect child(v) to parent(v)
3. 2 children
   - Swap v with x = successor(v)
   - delete(v)
     - (which is in the original position of the successor)

Will this cause more calls for the function delete()?

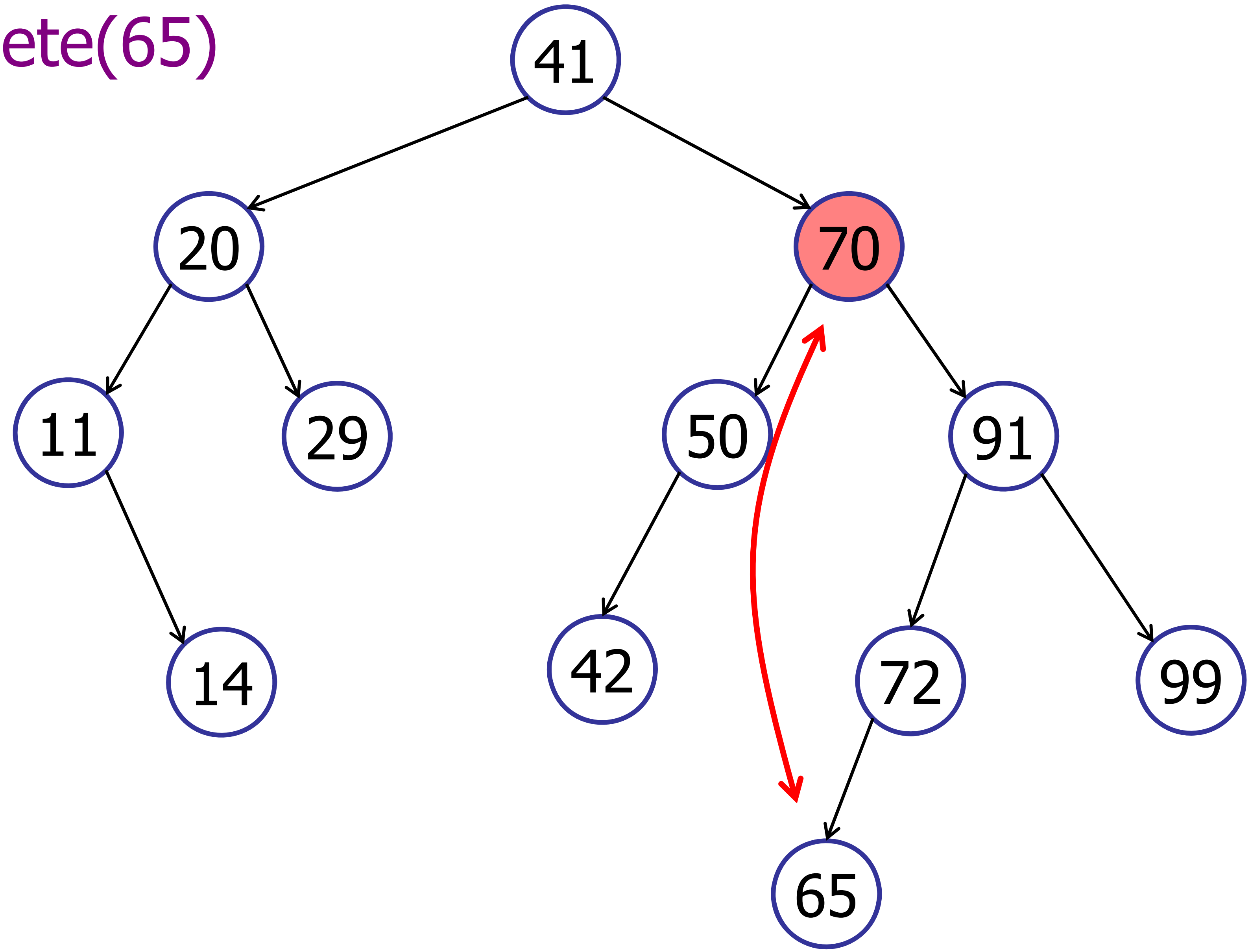
Binary Search Tree

delete(65)



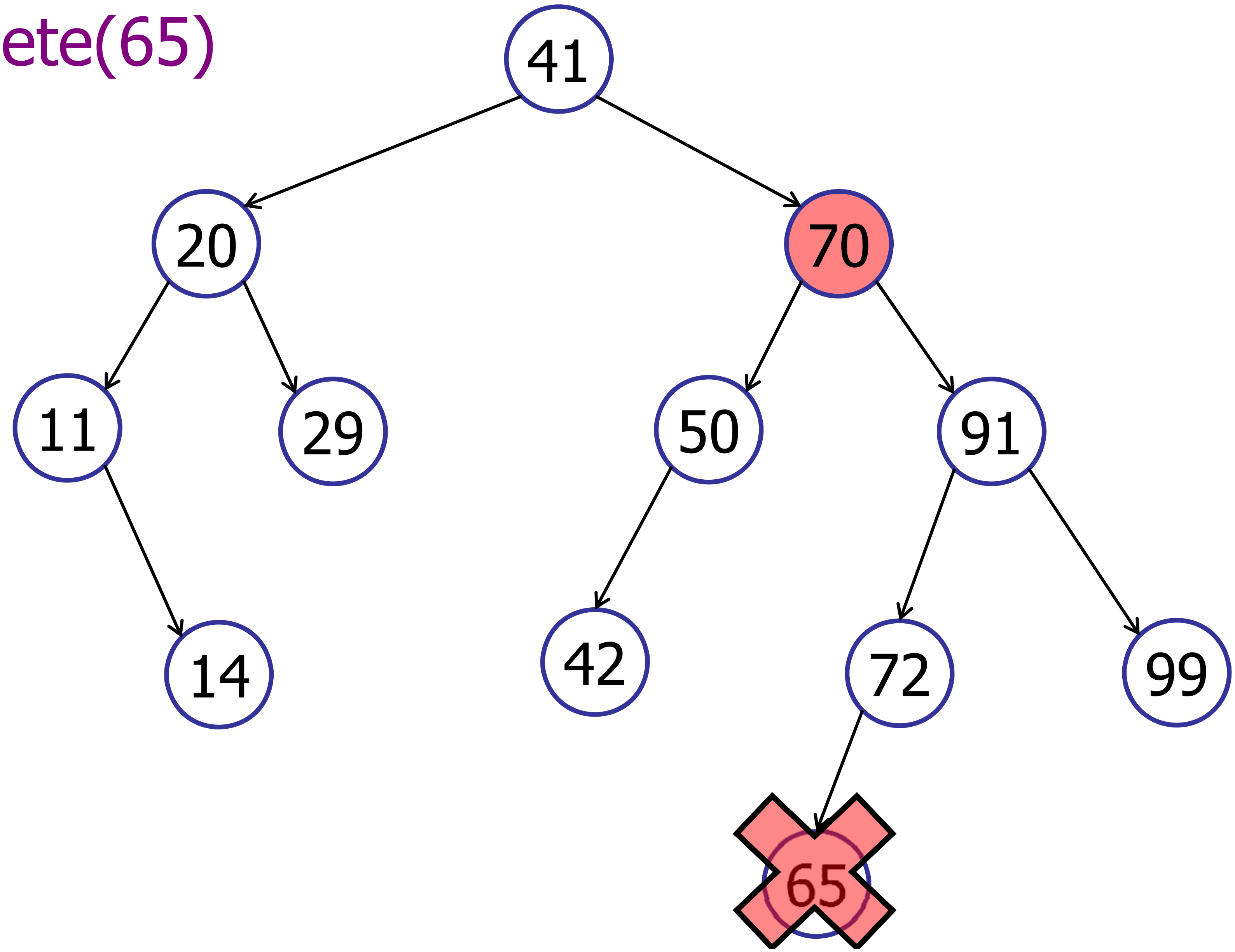
Binary Search Tree

delete(65)



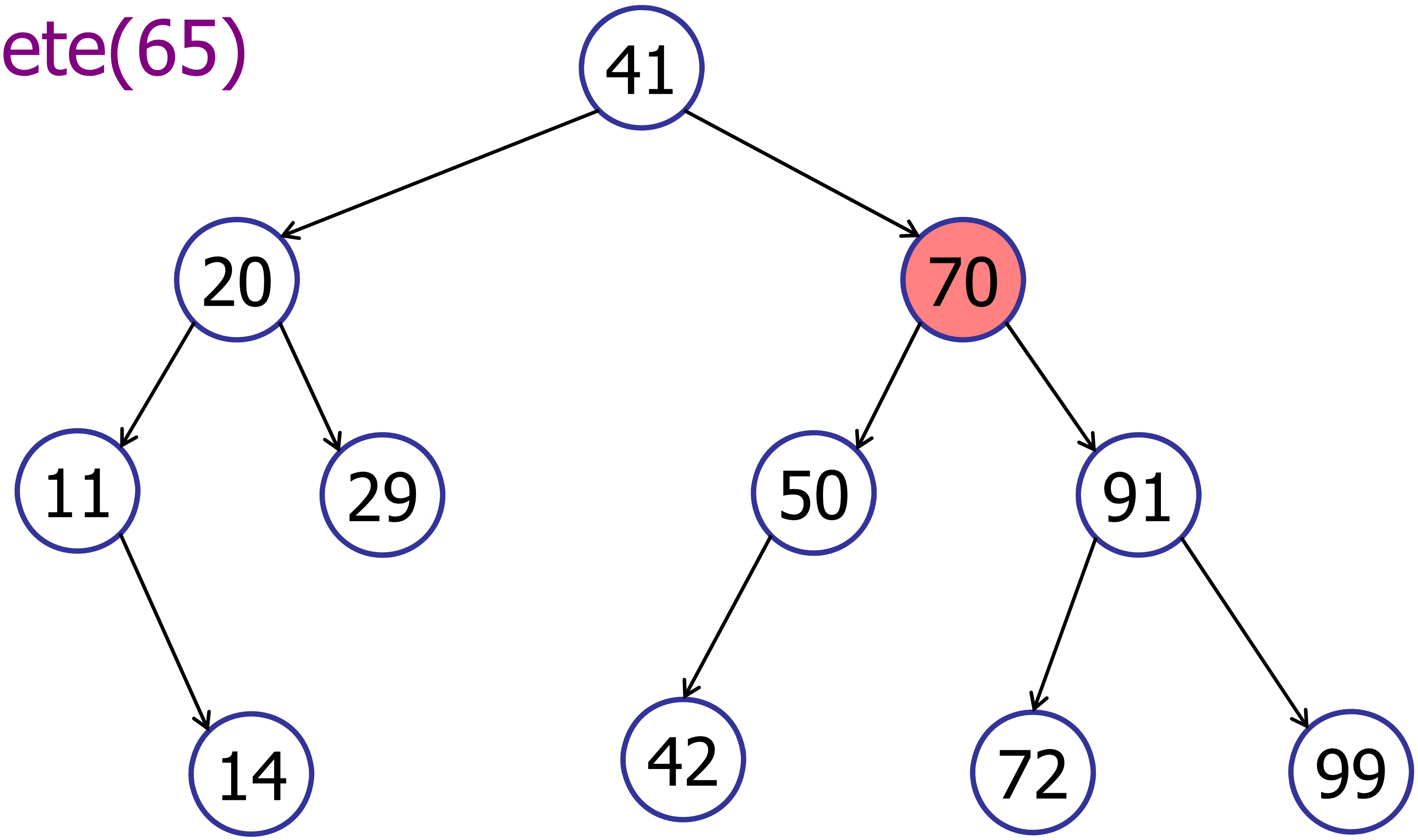
Binary Search Tree

delete(65)



Binary Search Tree

delete(65)



Binary Search Tree

delete(v)

1. If v has two children, swap it with its successor.

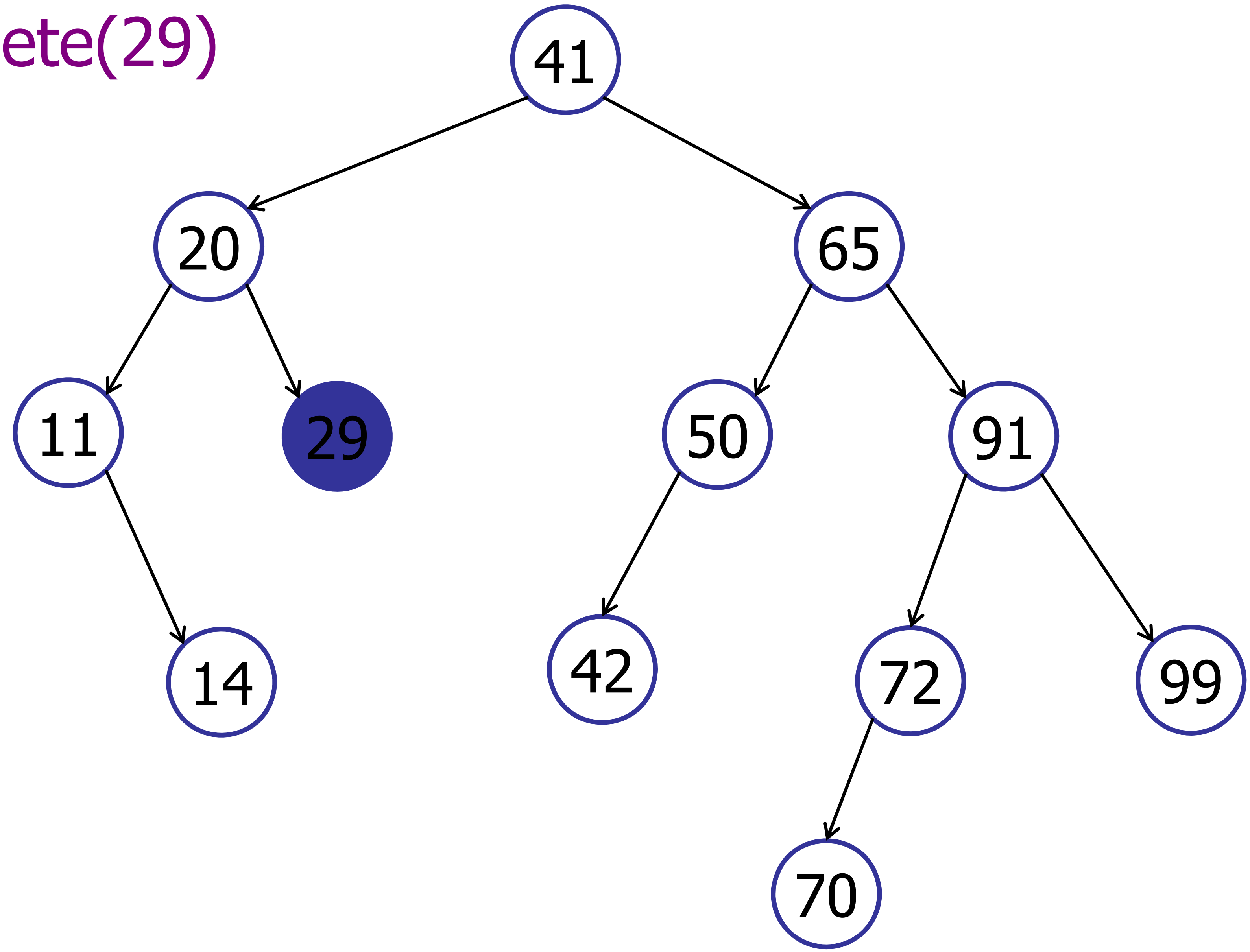
2. Delete node v from binary tree (and reconnect children).

3. For every ancestor of the deleted node:
   - Check if it is height-balanced.
   - If not, perform a rotation.
   - Continue to the root.

Deletion may take up to log(n) rotations.

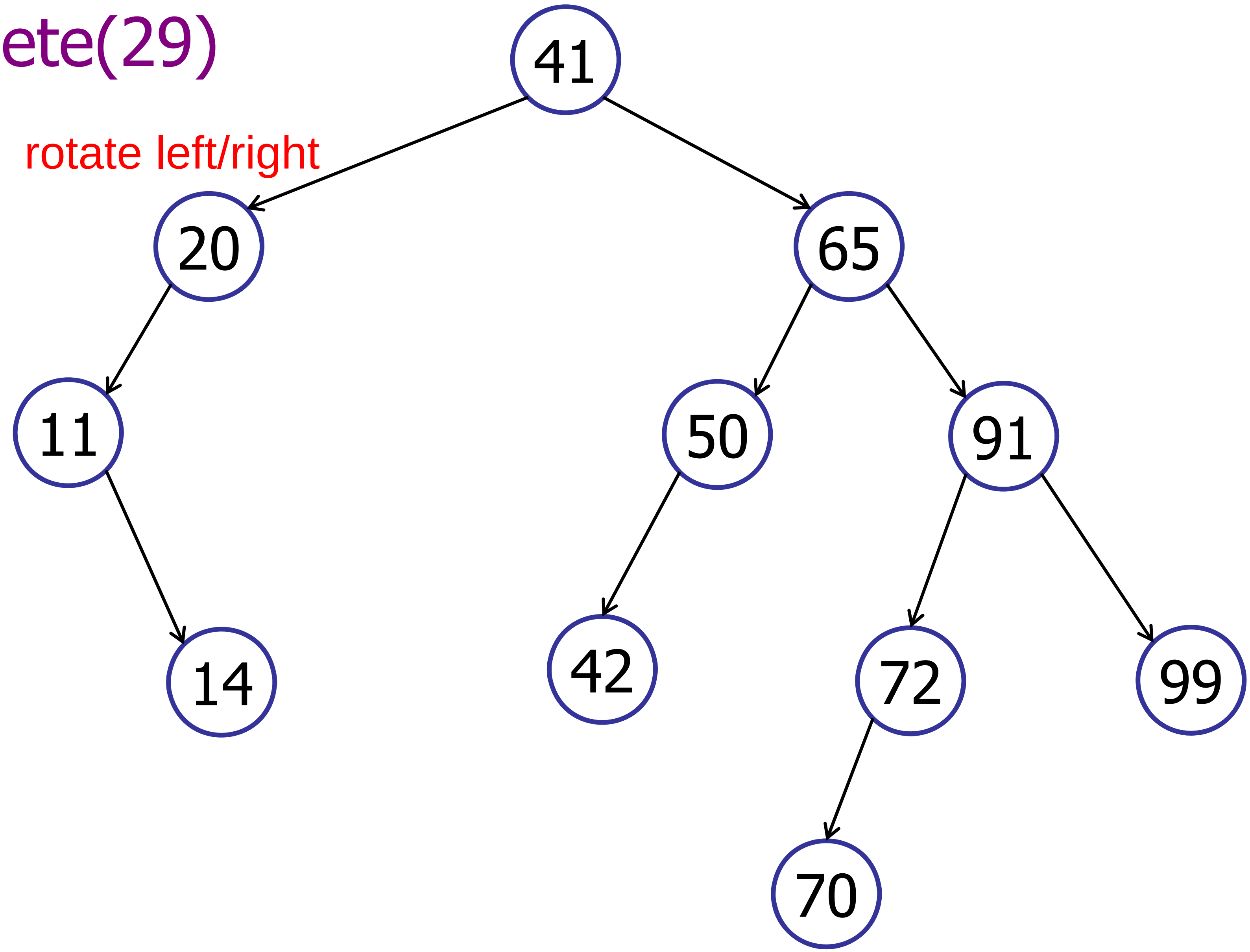
Binary Search Tree

delete(29)



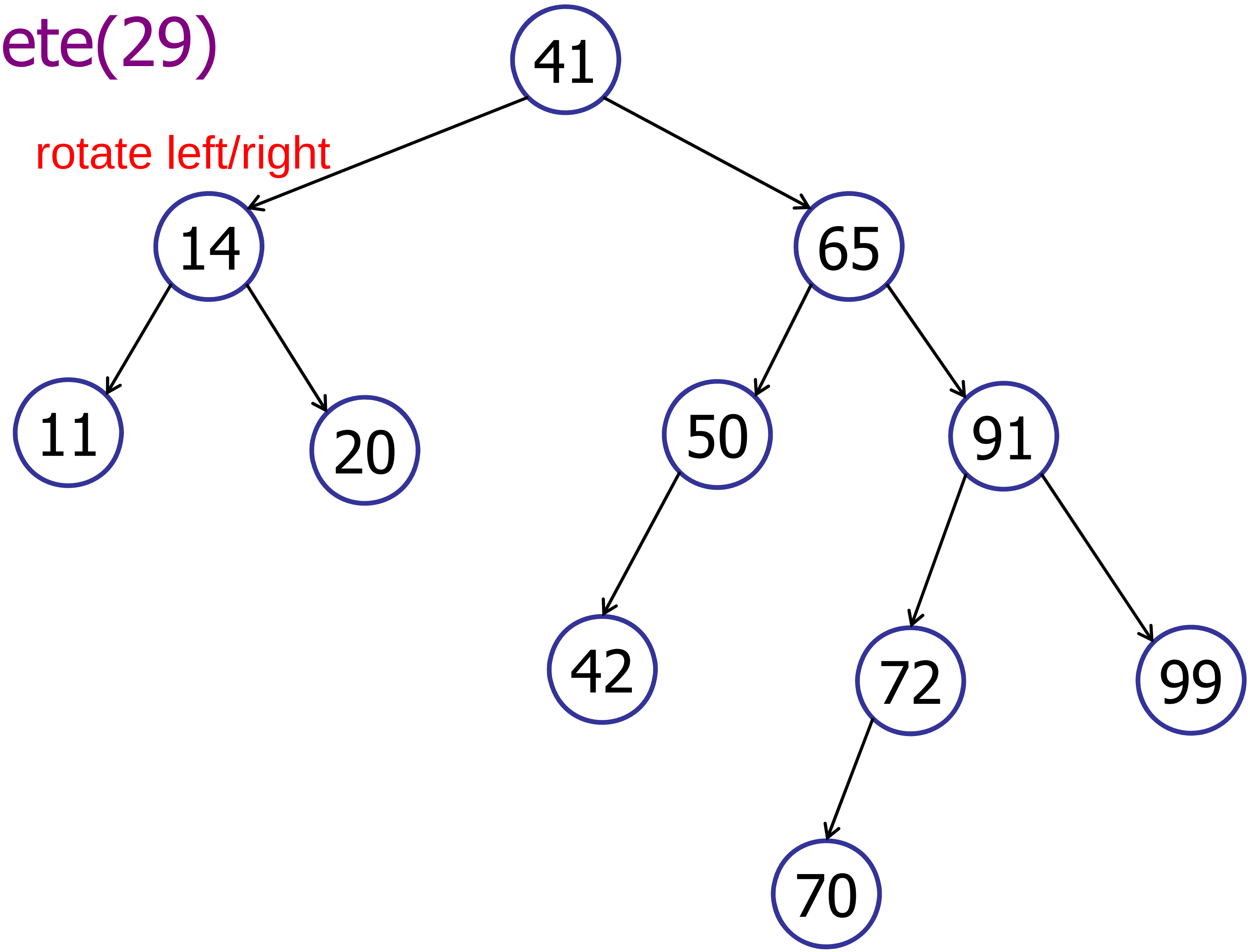
Binary Search Tree

delete(29)



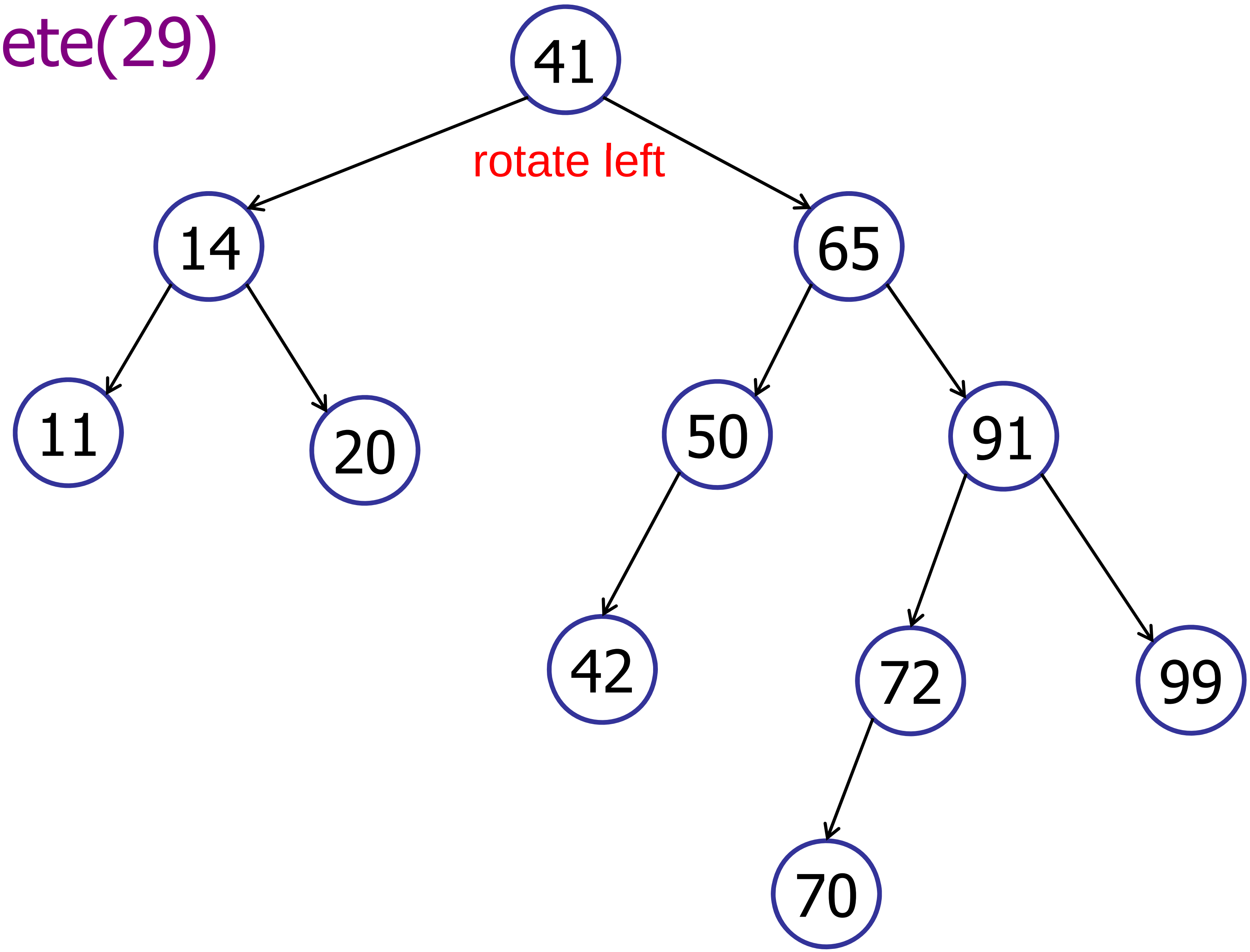
Binary Search Tree

delete(29)



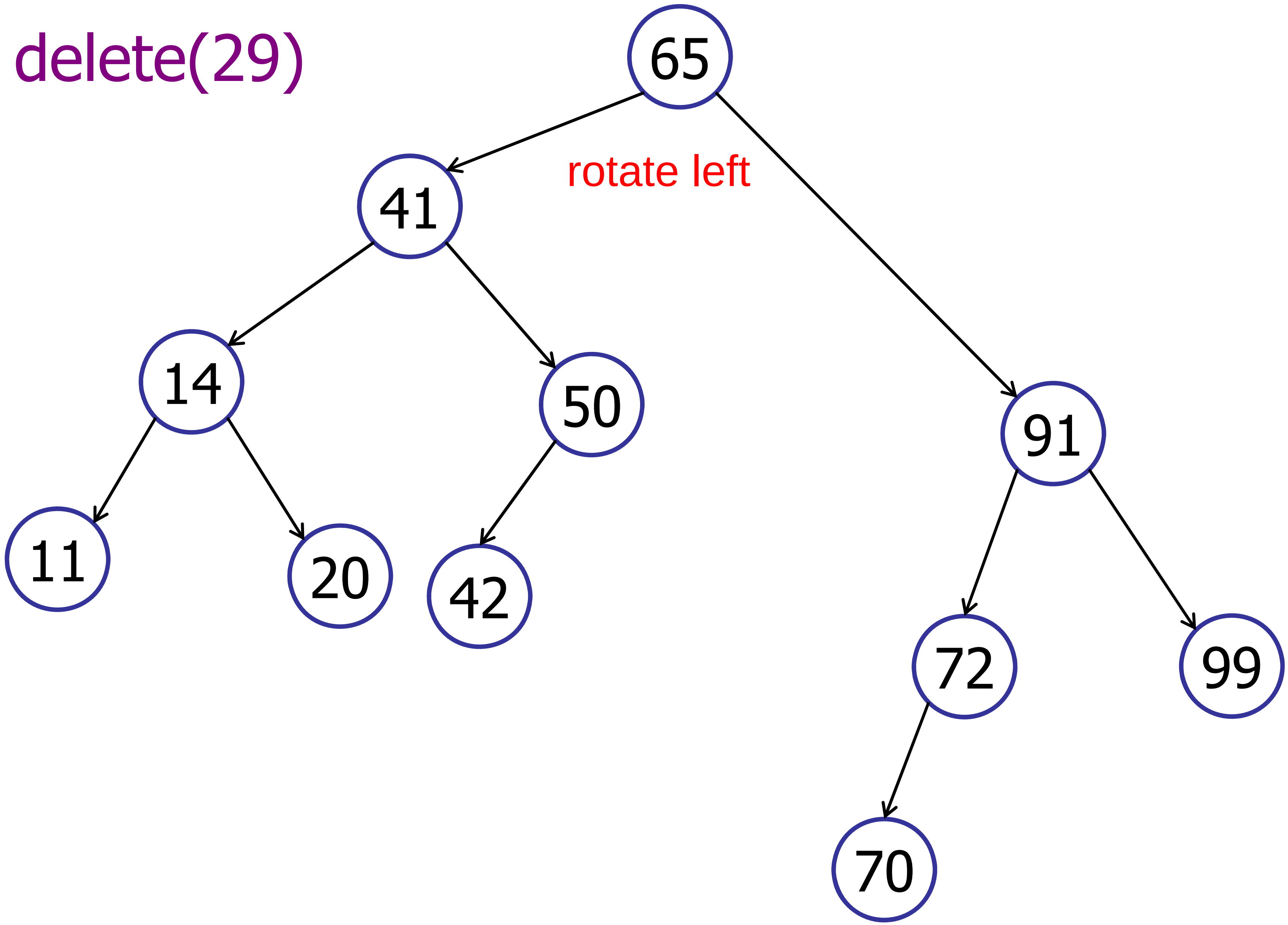
Binary Search Tree

delete(29)



Binary Search Tree

delete(29)



AVL Trees

What if you do not remove deleted nodes?

- Mark a node “deleted” and leave it in the tree.

Logical deletes:

- Performance degrades over time.
- Clean up later? (Amortized performance…)

AVL Trees

What if you do not want to store the height in every node?

– Only store difference in height from parent.

Balanced Search Trees

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)

Balanced Search Trees

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

Balanced Search Trees

Skip Lists and Treaps

- Randomized data structures
- Random insertions => balanced tree
- Use randomness on insertion to maintain balance

The Importance of Being Balanced

Is it really important?

The 90-10 Rule

90% of your queries access 10% of your data.

Example:

search(70)

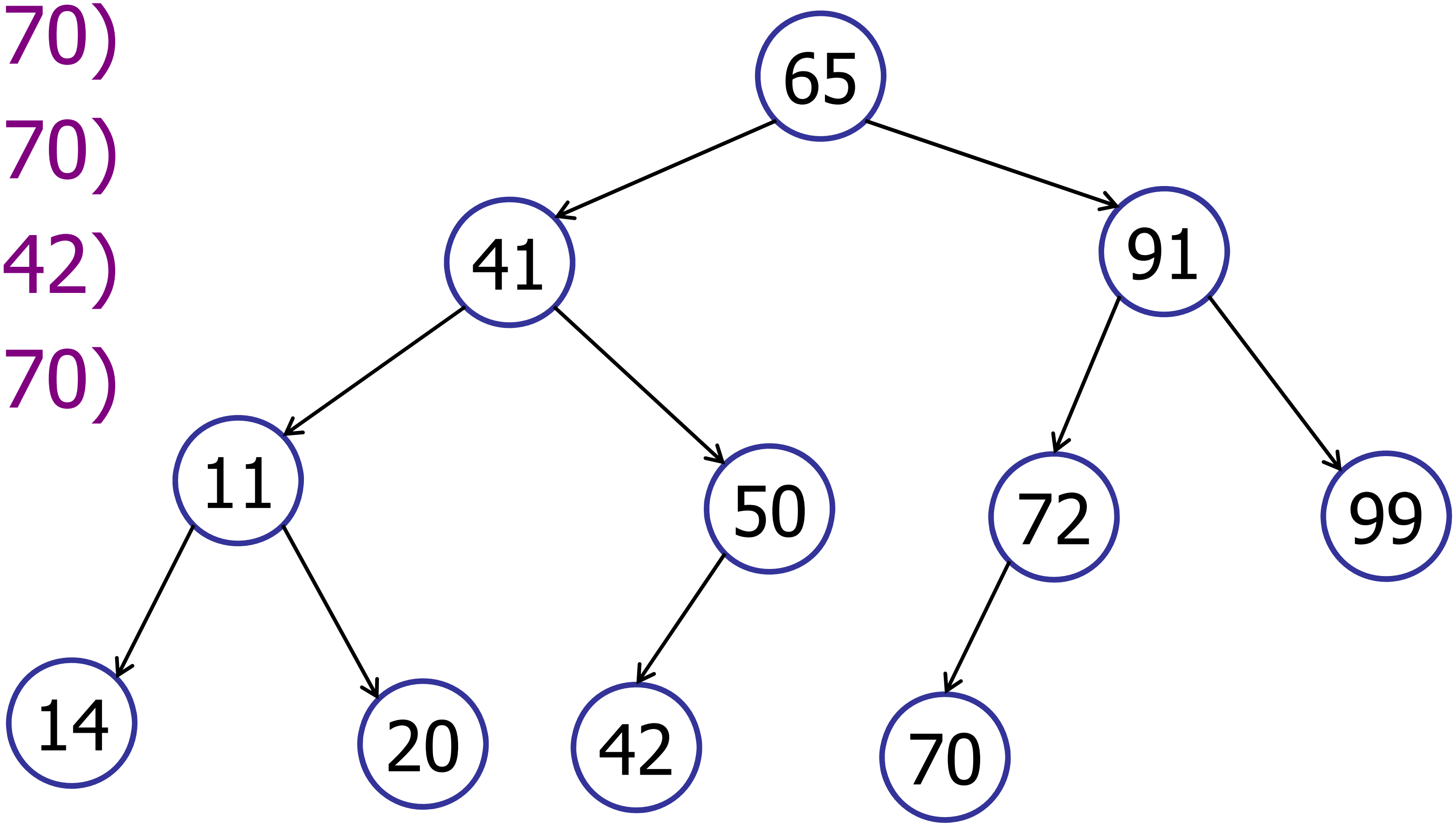
search(70)

search(70)

search(70)

search(42)

search(70)



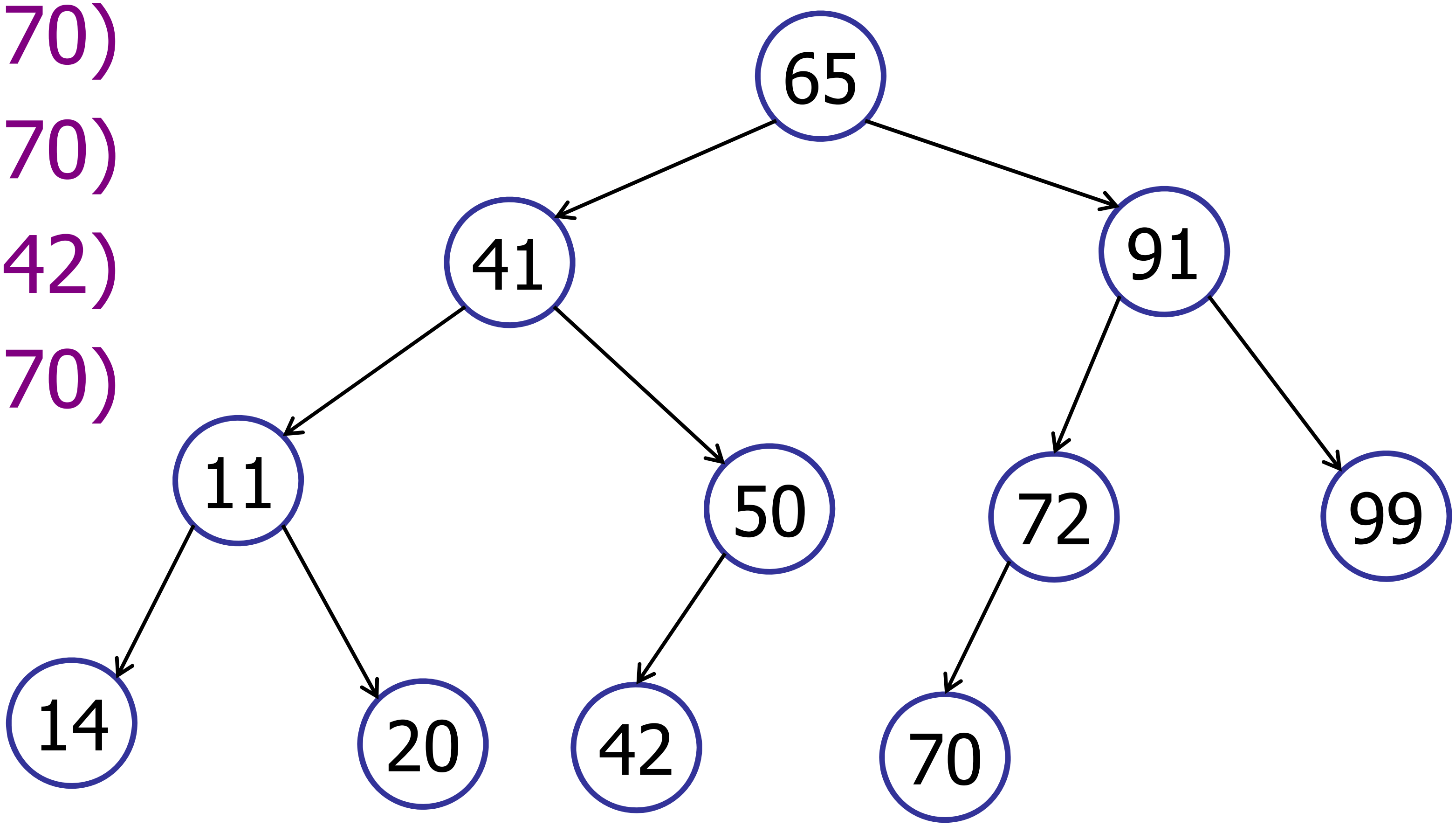
Remember Problem Set 2?

Move-to-Front List

- Whenever you search for an item, move it to the front of the list.
- Recently used items stay at the front of the list.

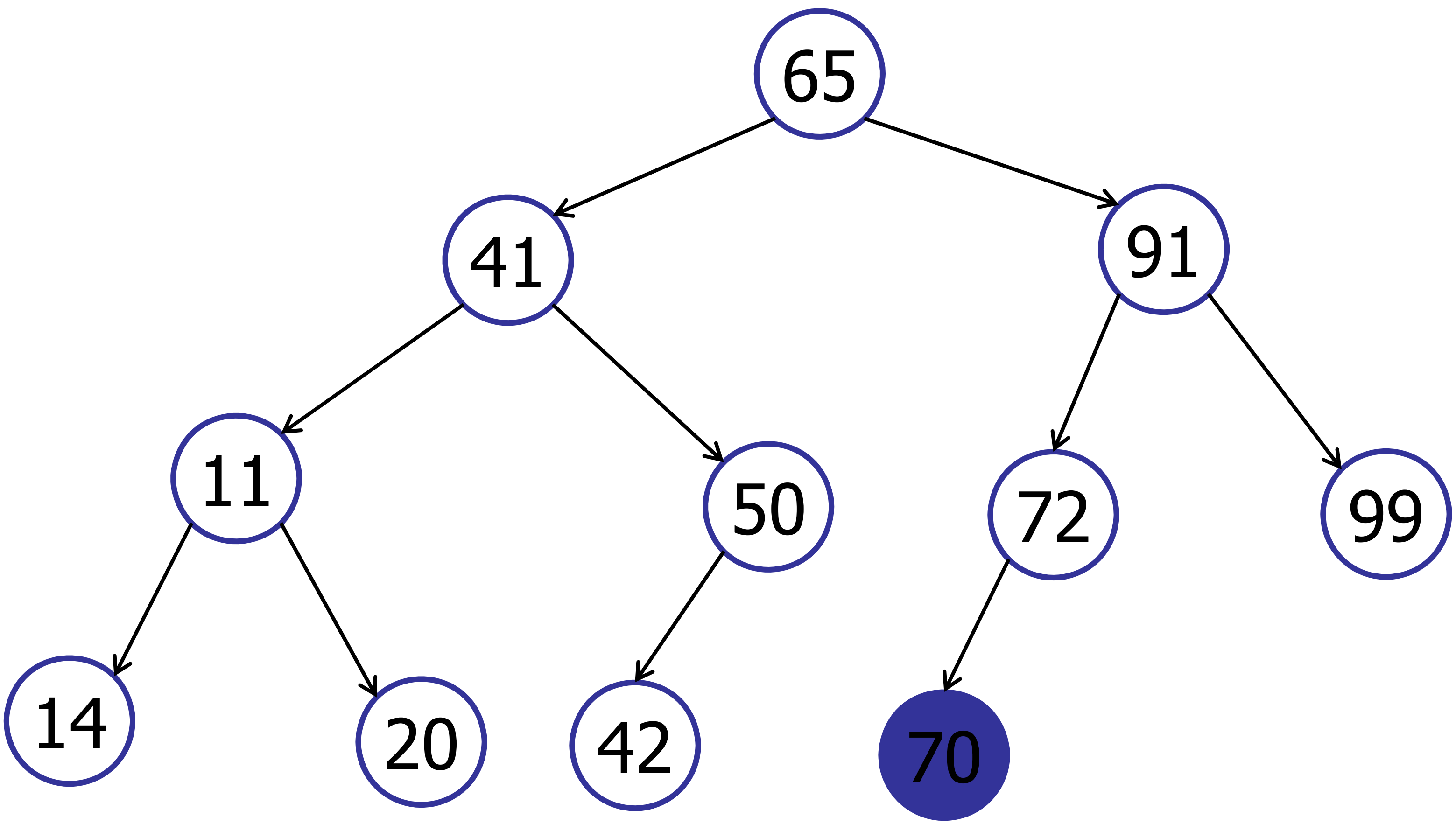
Move-to-Front Tree?

search(70)
search(70)
search(70)
search(70)
search(42)
search(70)



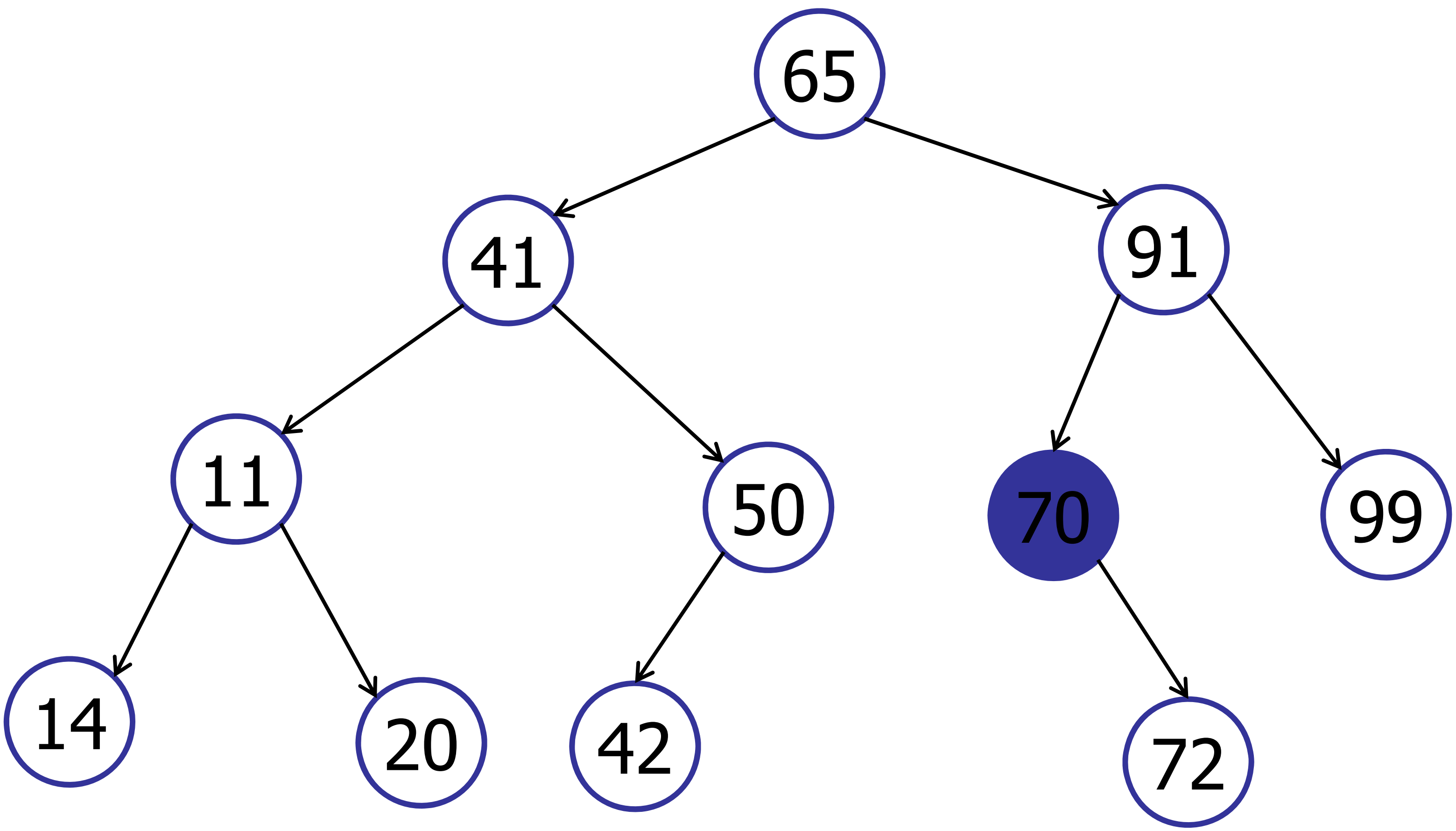
Move-to-Root Tree

search(70)



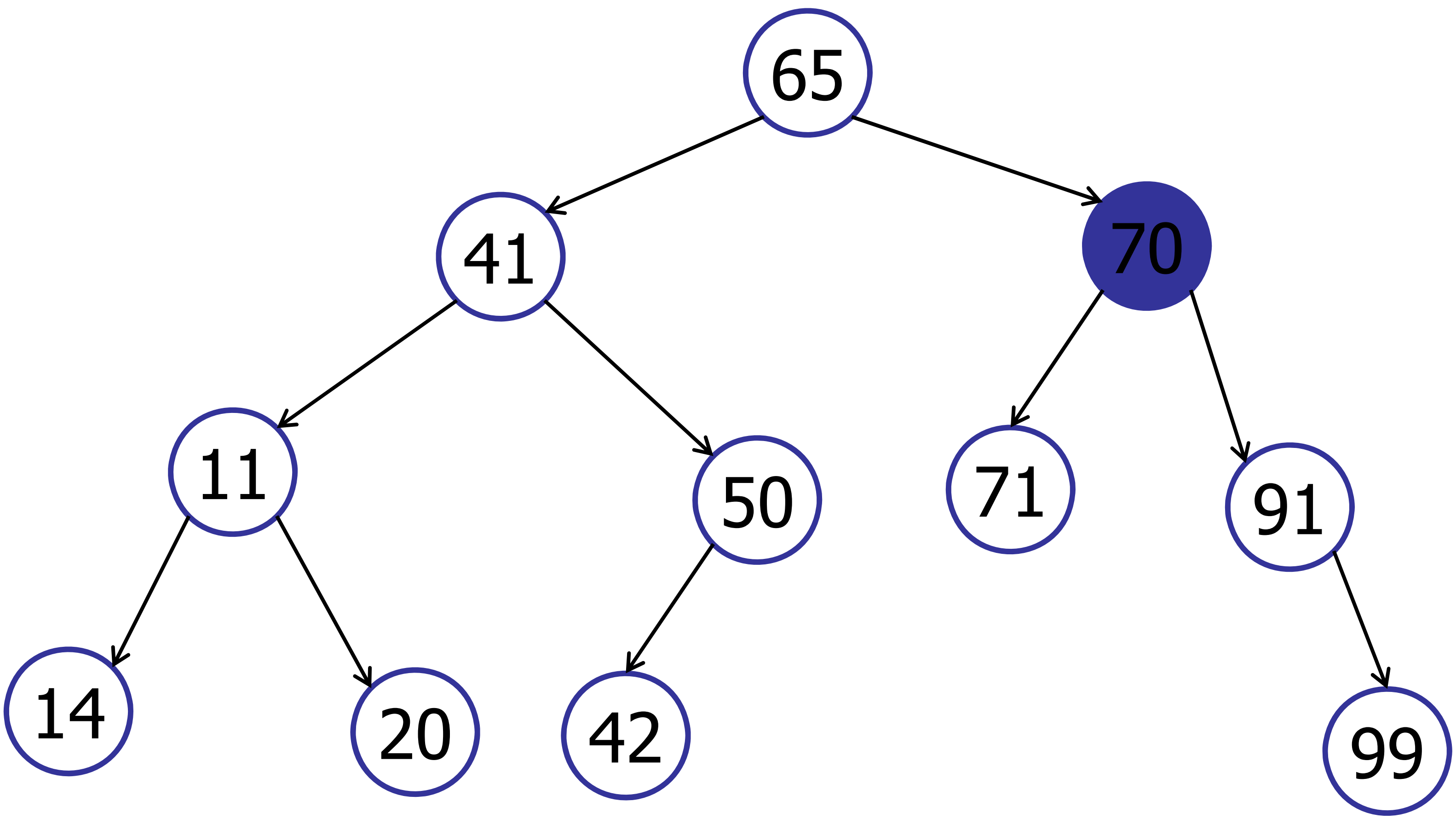
Move-to-Root Tree

search(70)



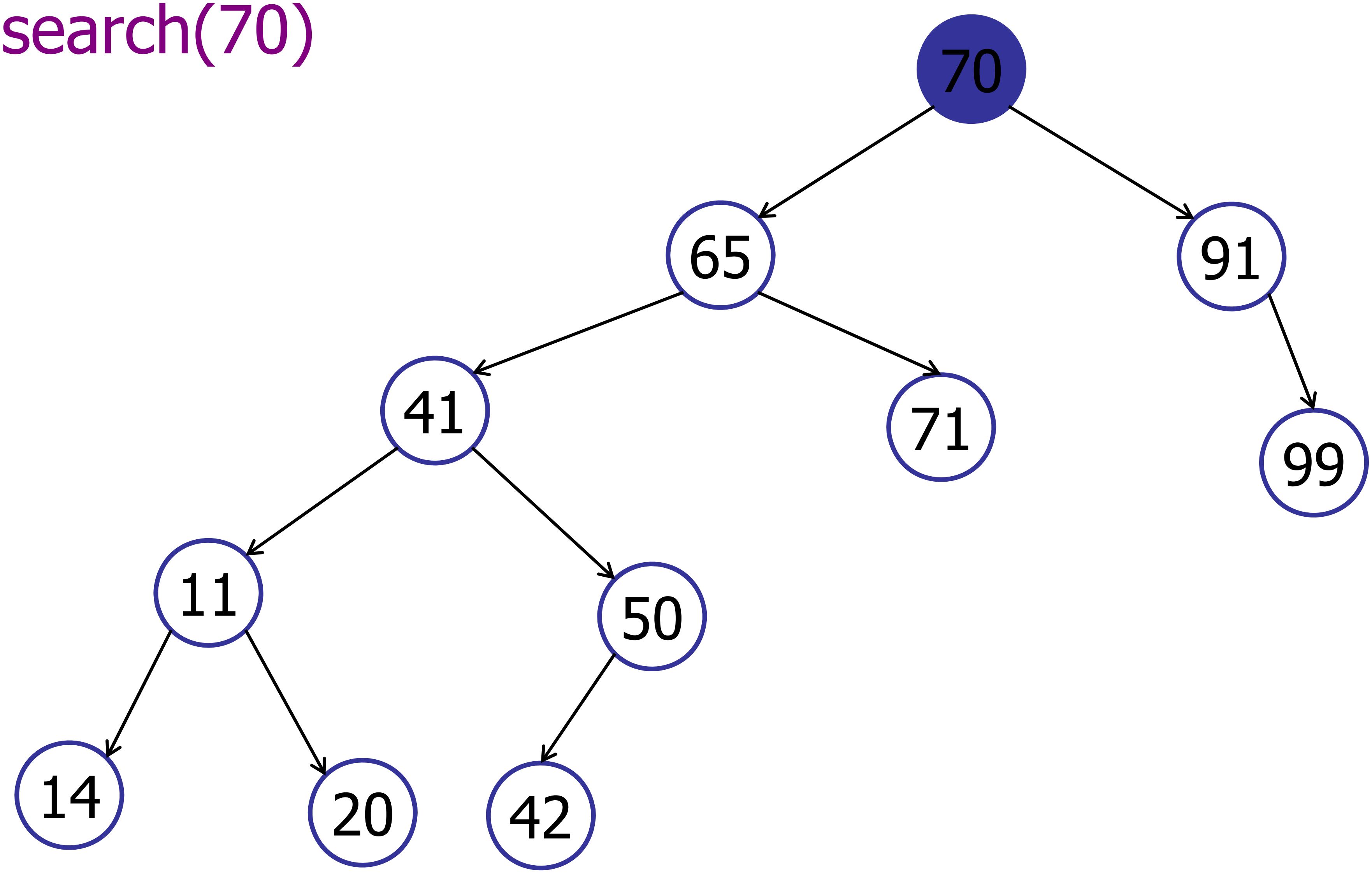
Move-to-Root Tree

search(70)



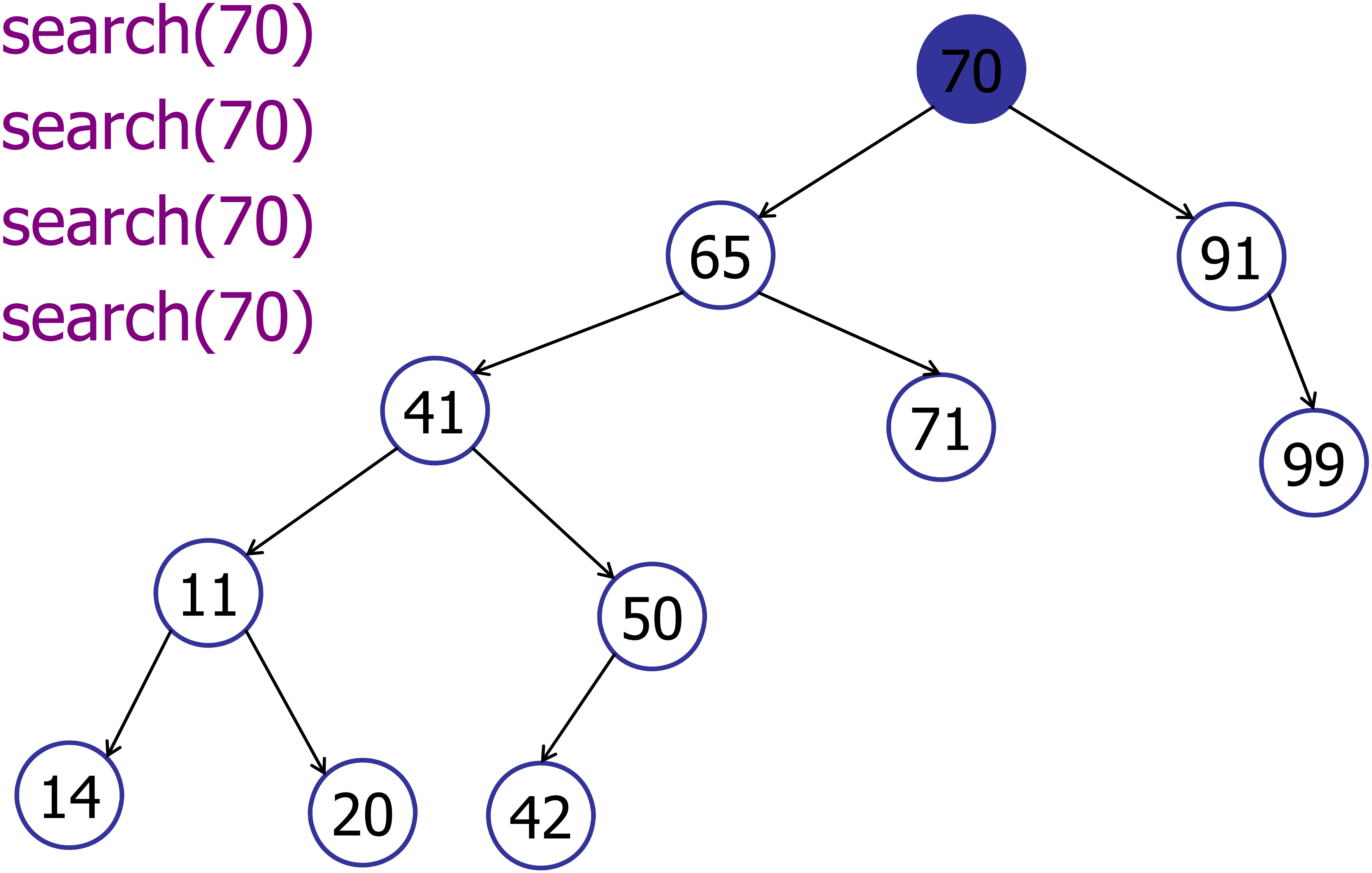
Move-to-Root Tree

search(70)



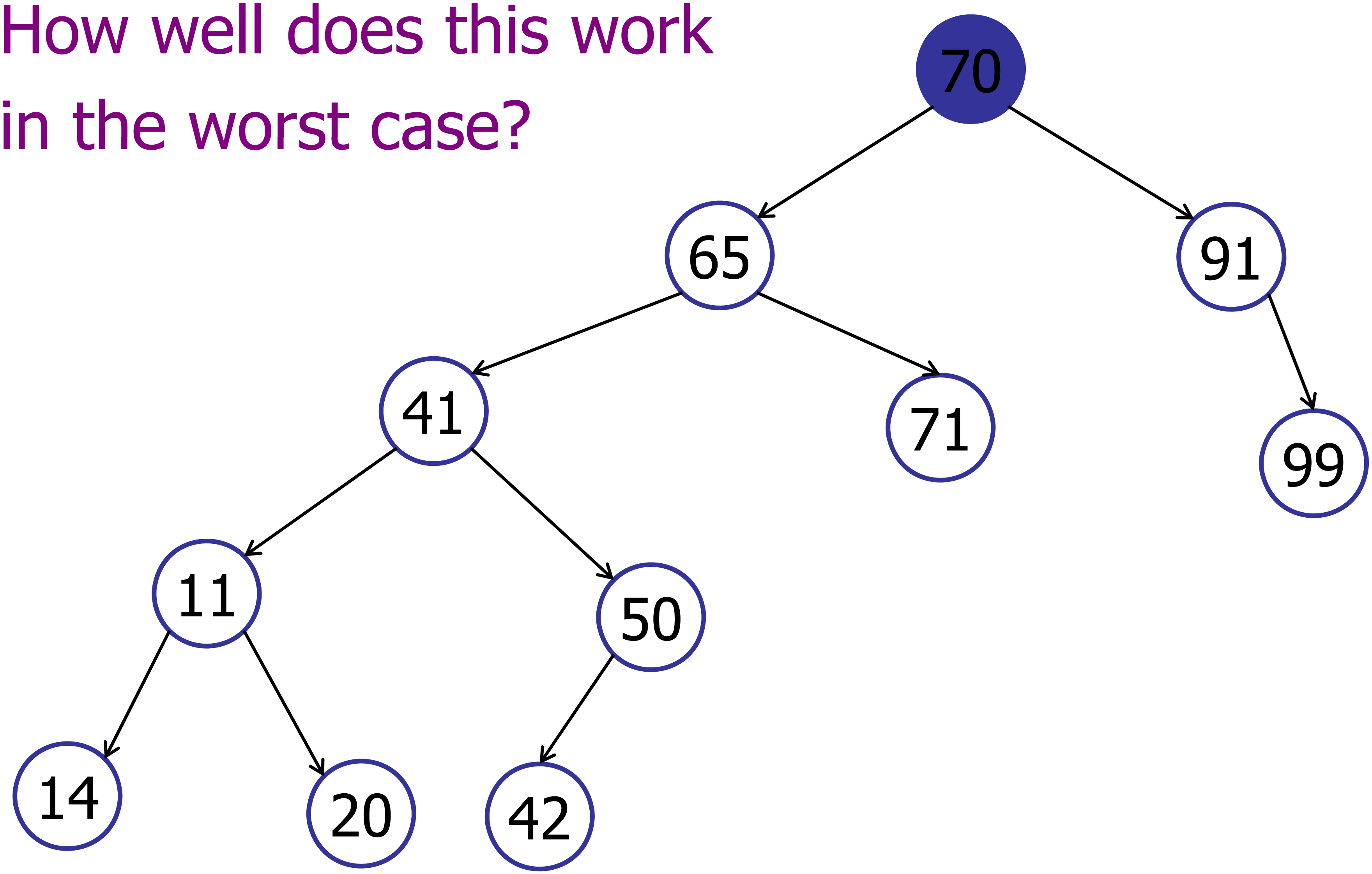
Move-to-Root Tree

search(70)
search(70)
search(70)
search(70)



Move-to-Root Tree

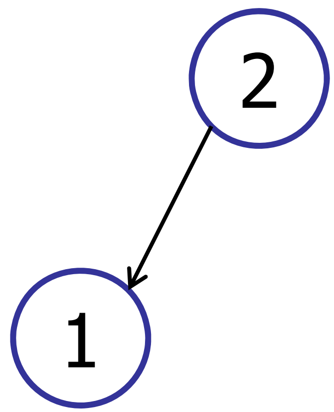
How well does this work in the worst case?



Move-to-Root Tree

insert(1)

insert(2)

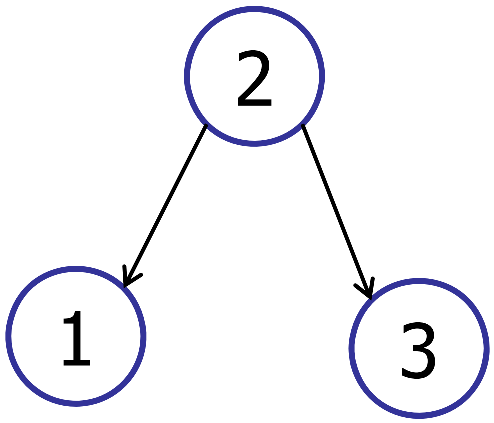


Move-to-Root Tree

insert(1)

insert(2)

insert(3)

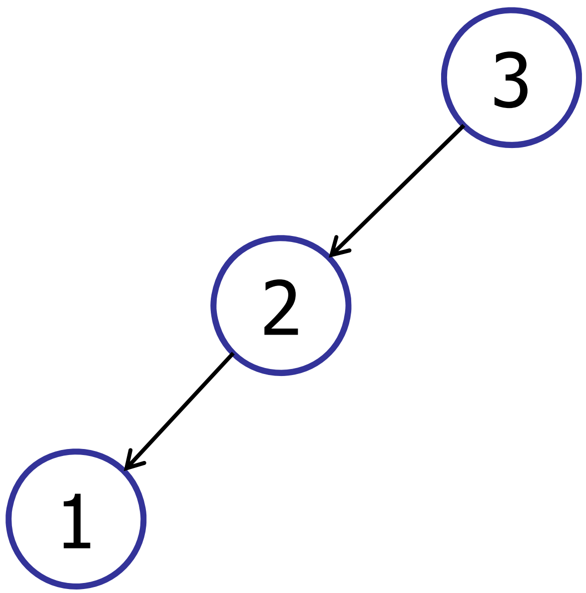


Move-to-Root Tree

insert(1)

insert(2)

insert(3)



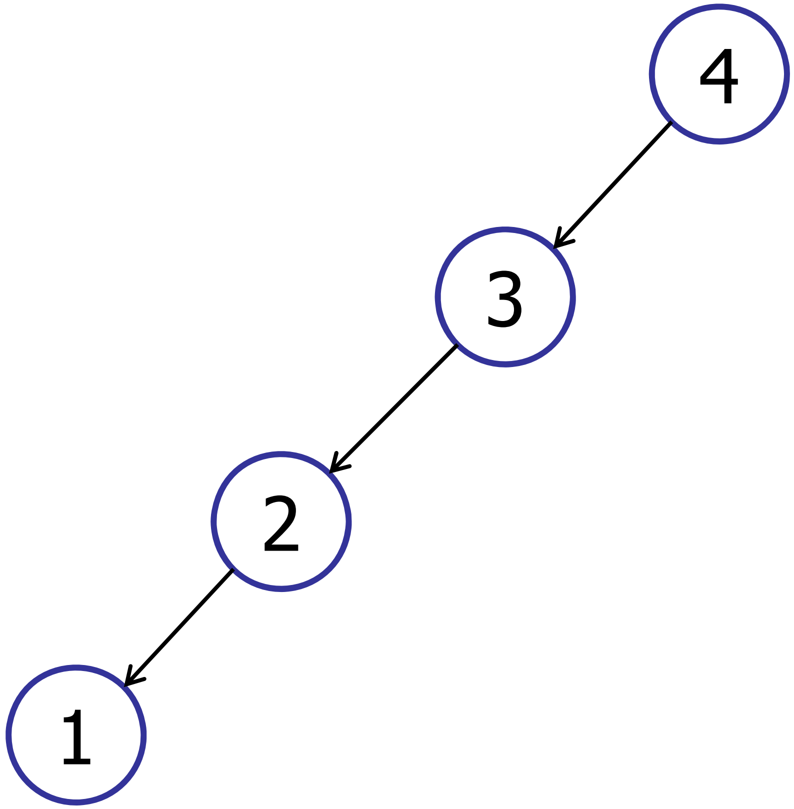
Move-to-Root Tree

insert(1)

insert(2)

insert(3)

insert(4)



Move-to-Root Tree

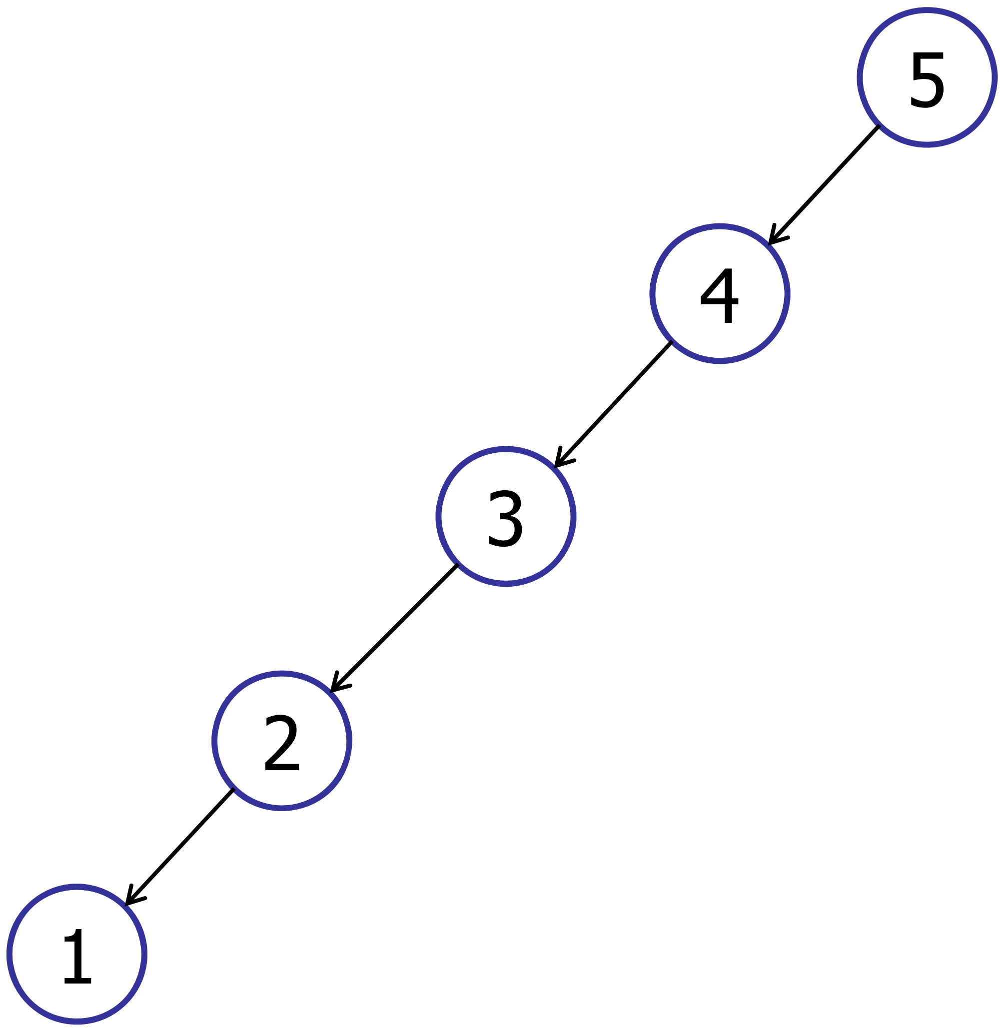
insert(1)

insert(2)

insert(3)

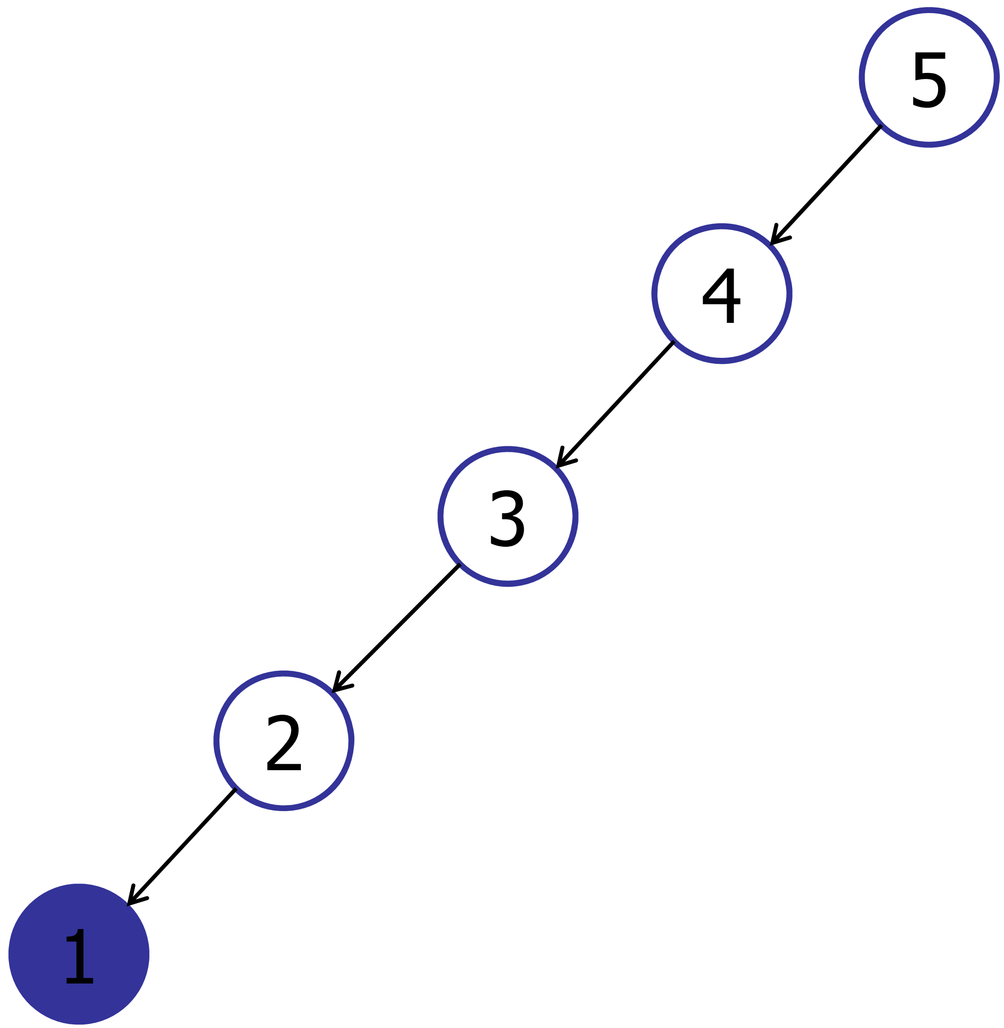
insert(4)

insert(5)



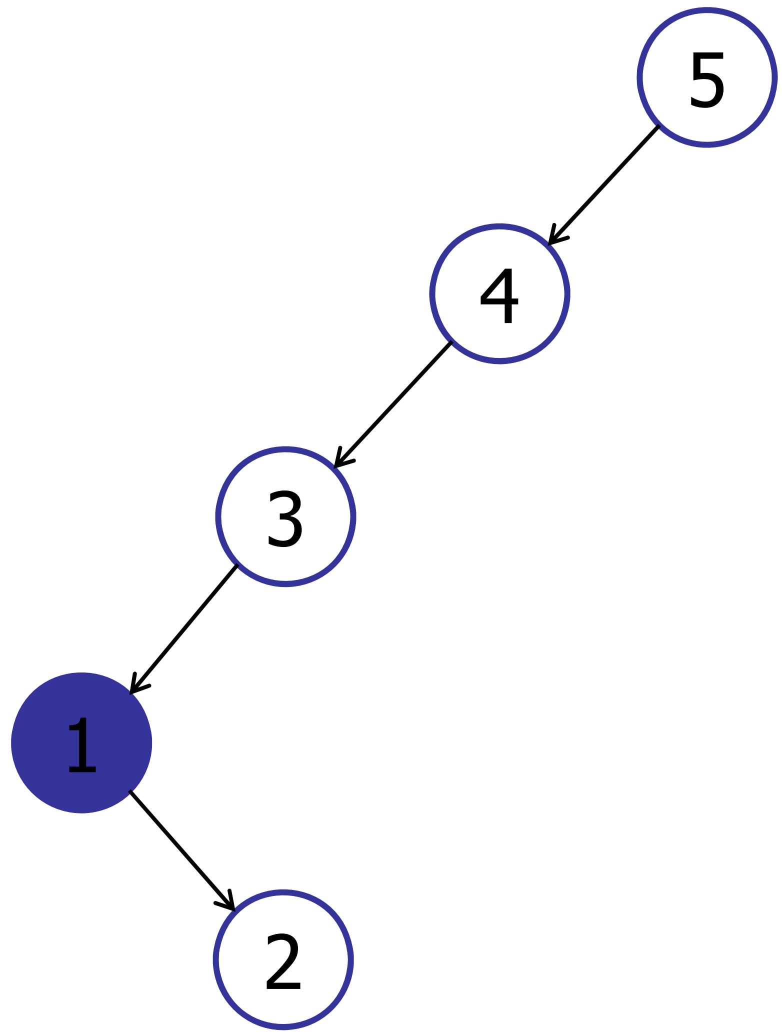
Move-to-Root Tree

search(1)



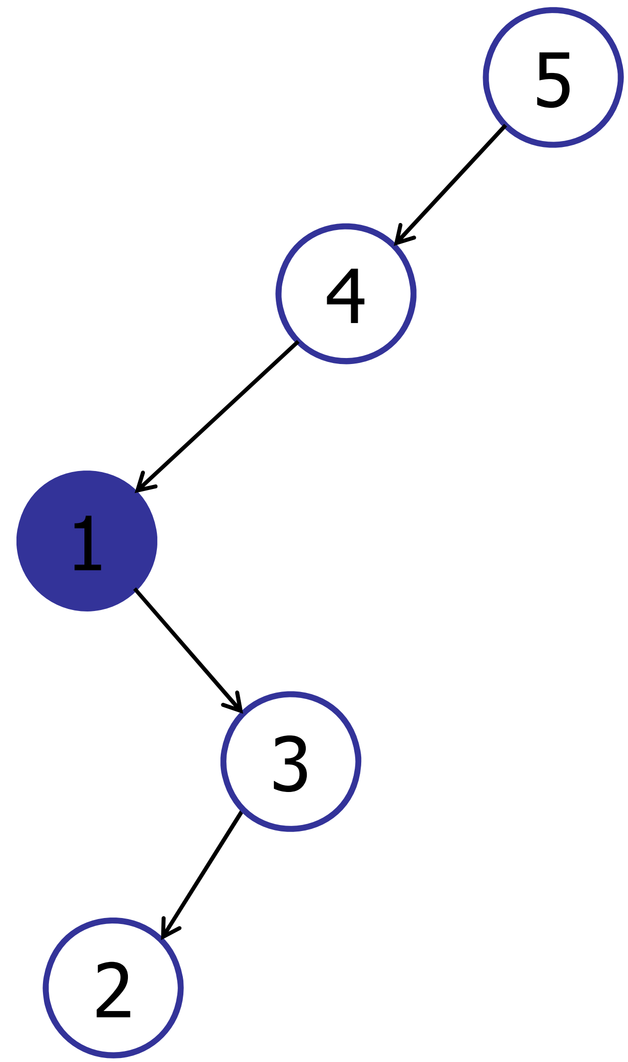
Move-to-Root Tree

search(1)



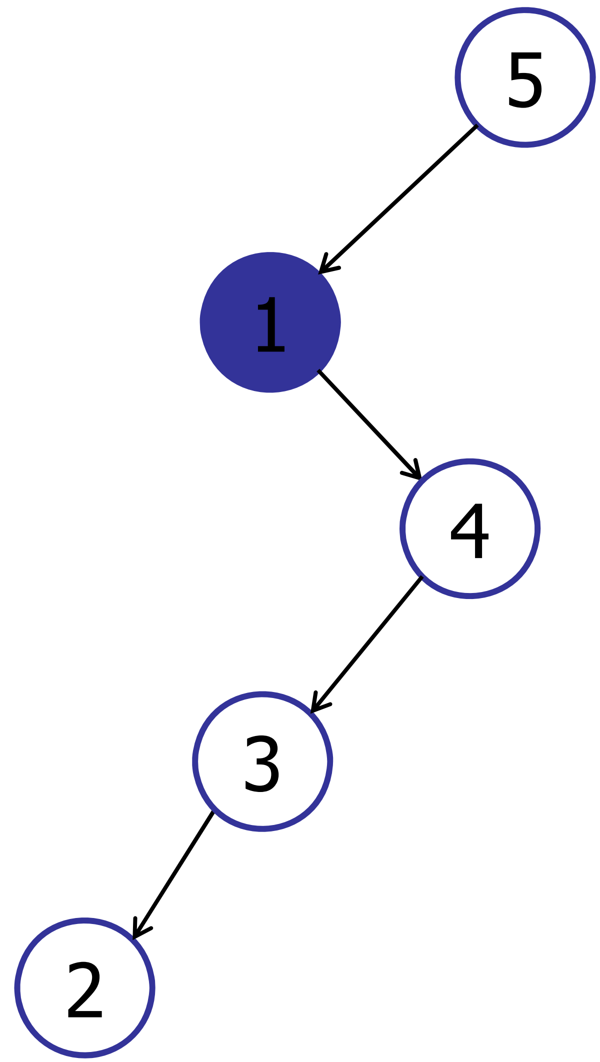
Move-to-Root Tree

search(1)



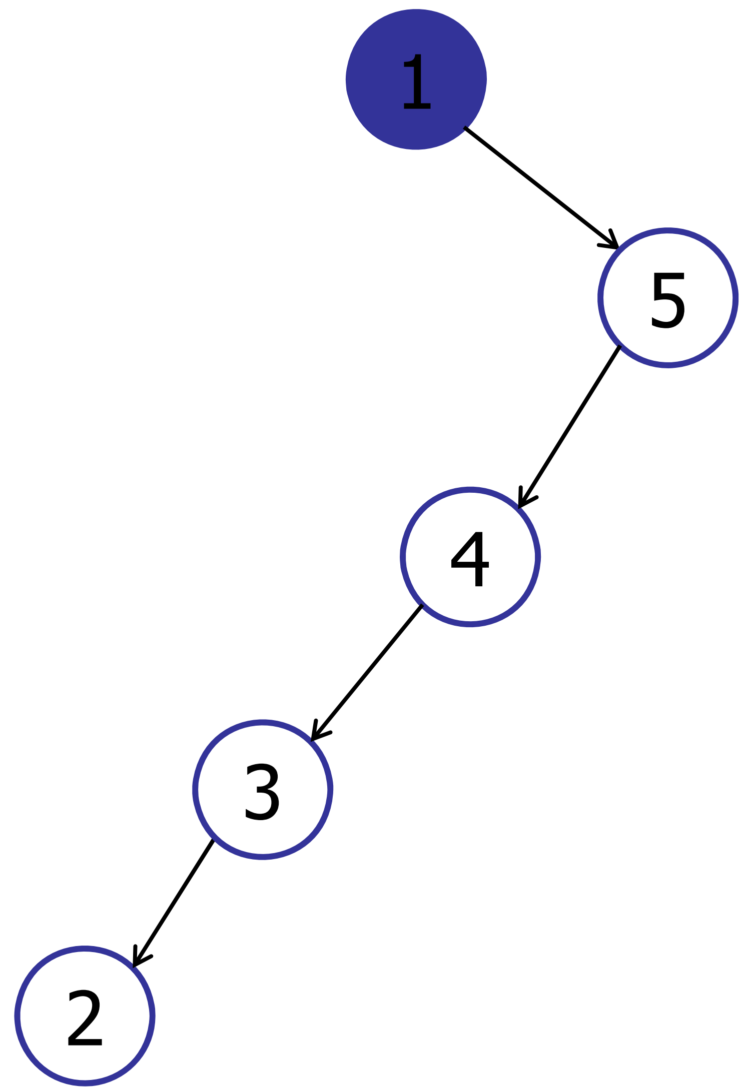
Move-to-Root Tree

search(1)



Move-to-Root Tree

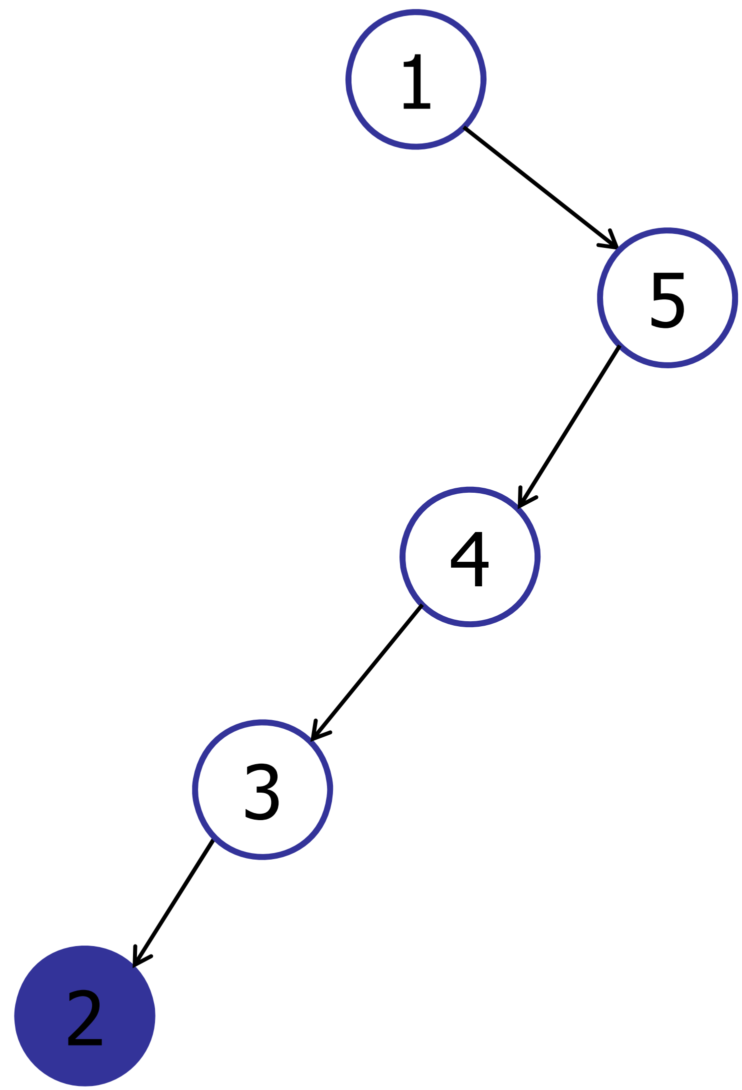
search(1)



Move-to-Root Tree

search(1)

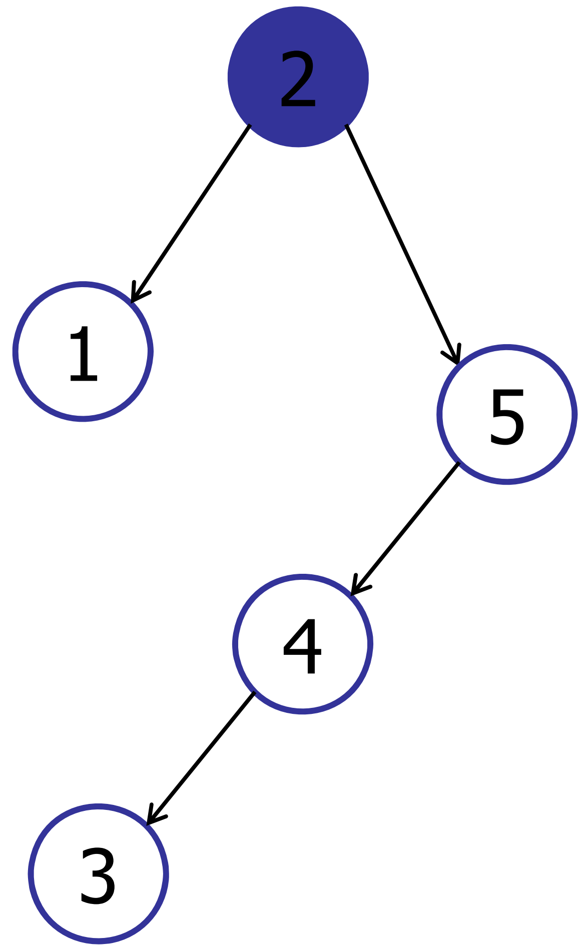
search(2)



Move-to-Root Tree

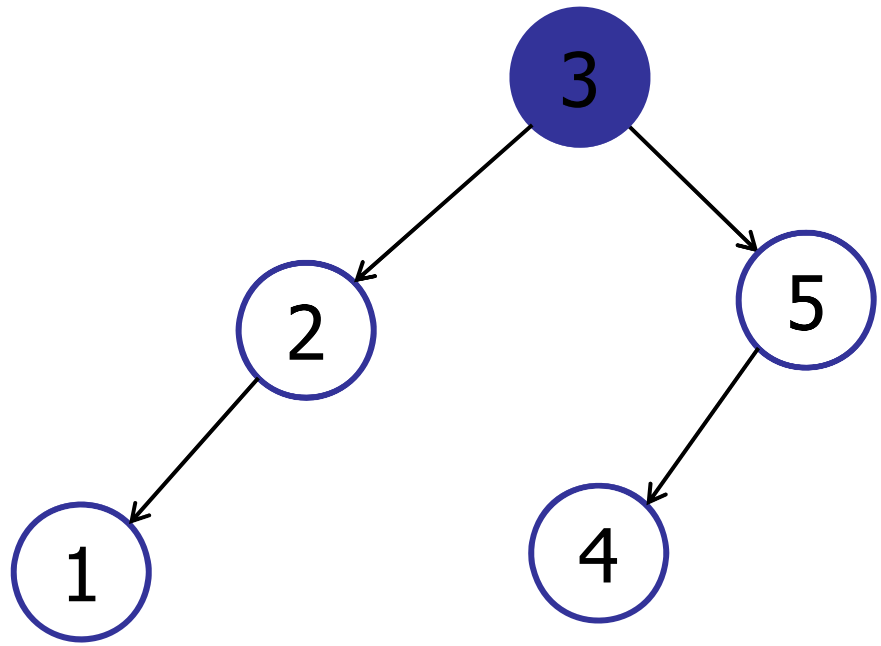
search(1)

search(2)



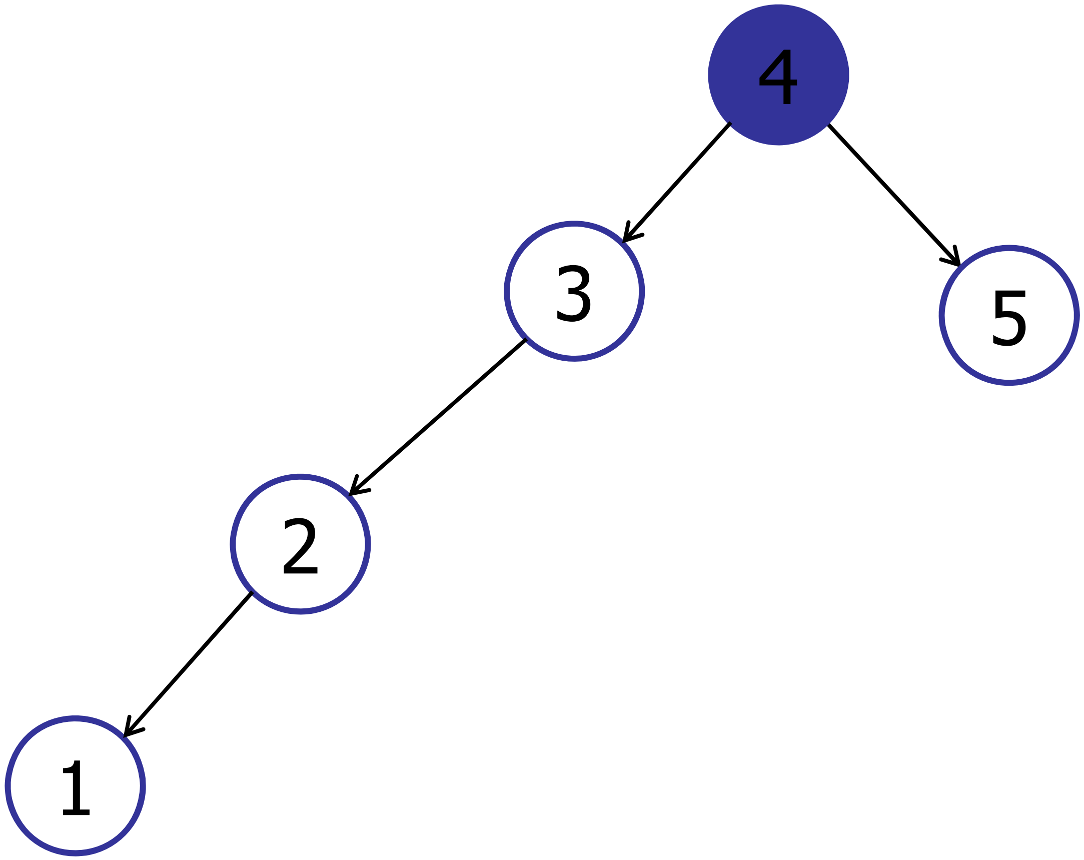
Move-to-Root Tree

search(1)
search(2)
search(3)



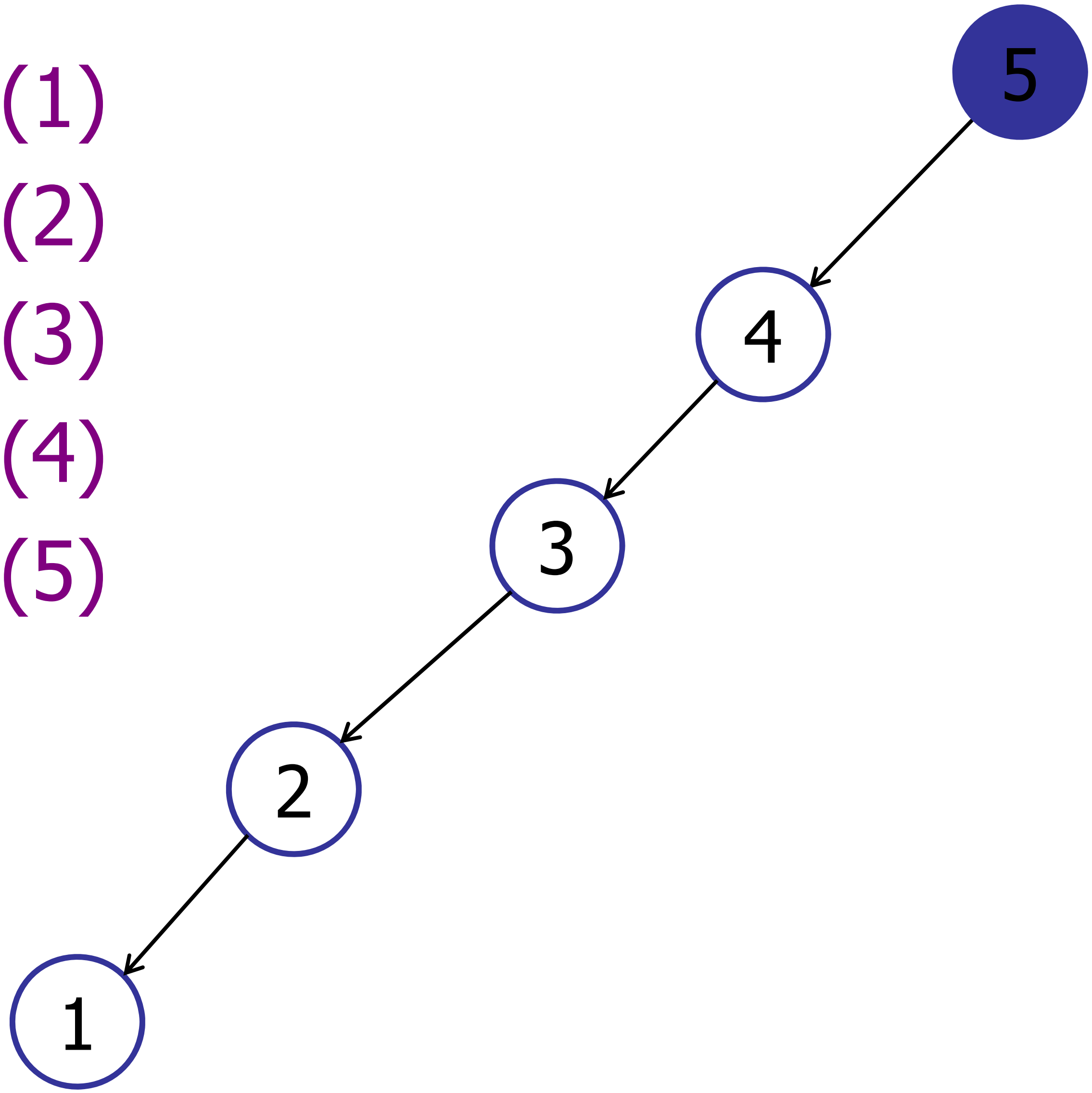
Move-to-Root Tree

search(1)
search(2)
search(3)
search(4)



Move-to-Root Tree

search(1)
search(2)
search(3)
search(4)
search(5)



Splay Trees

On search/insert/delete:

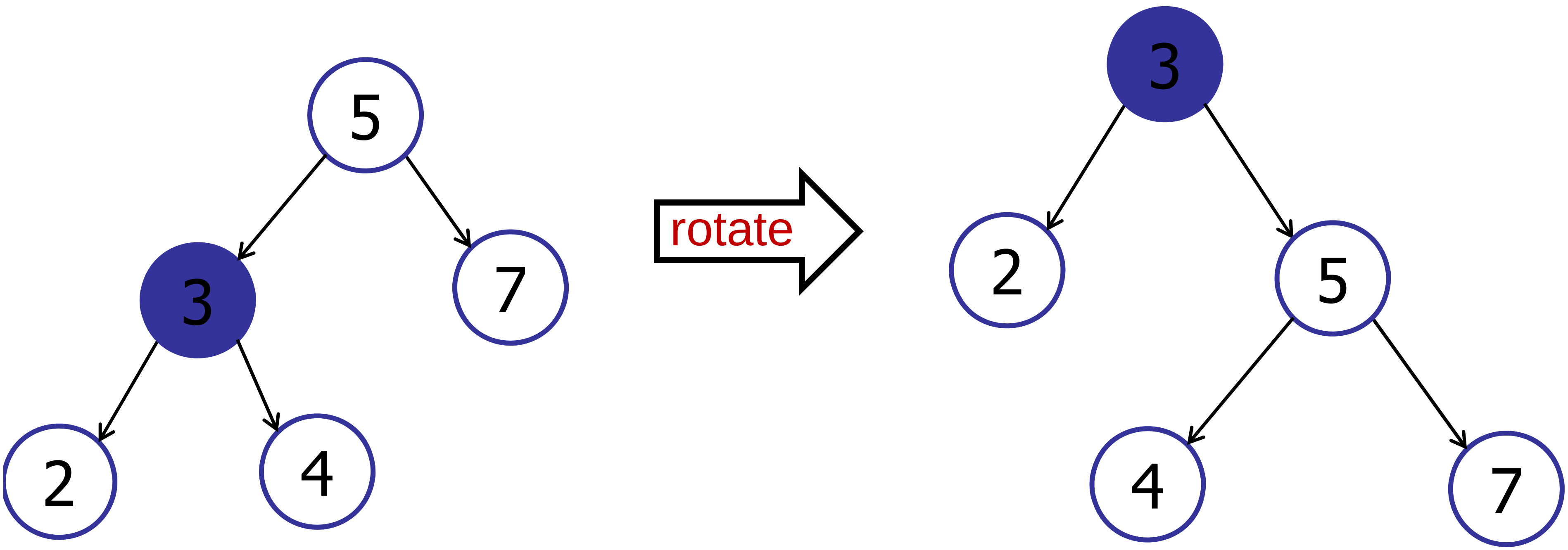
- Move to root, not rotate-to-root.
- Balance more along the way.

Three cases:

- Zig: Parent is root.
- Zig-Zag: Parent is left, grandparent is right.
- Zig-Zig: Parent and grandparent are left children.

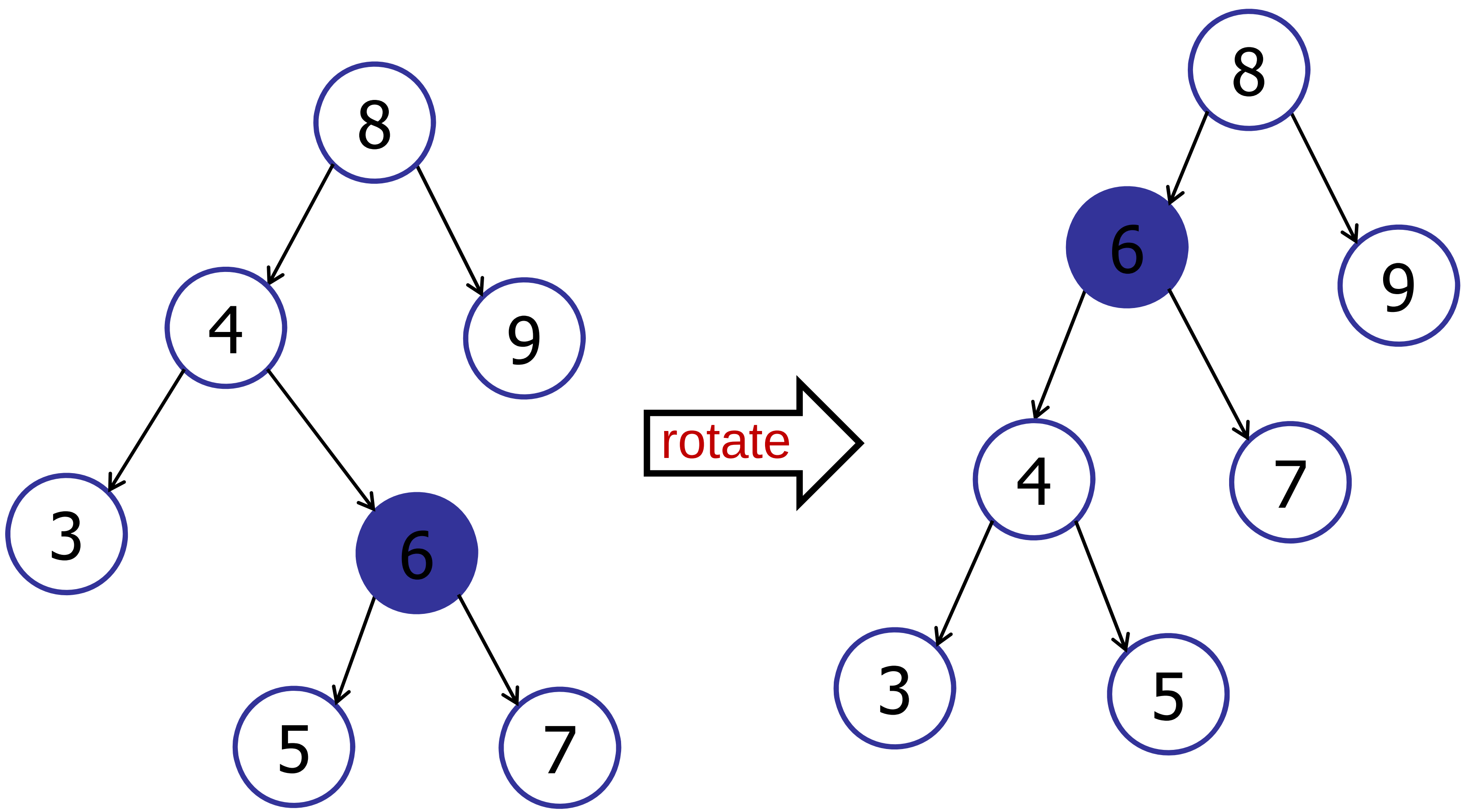
Splay Trees

Zig: Parent is root.



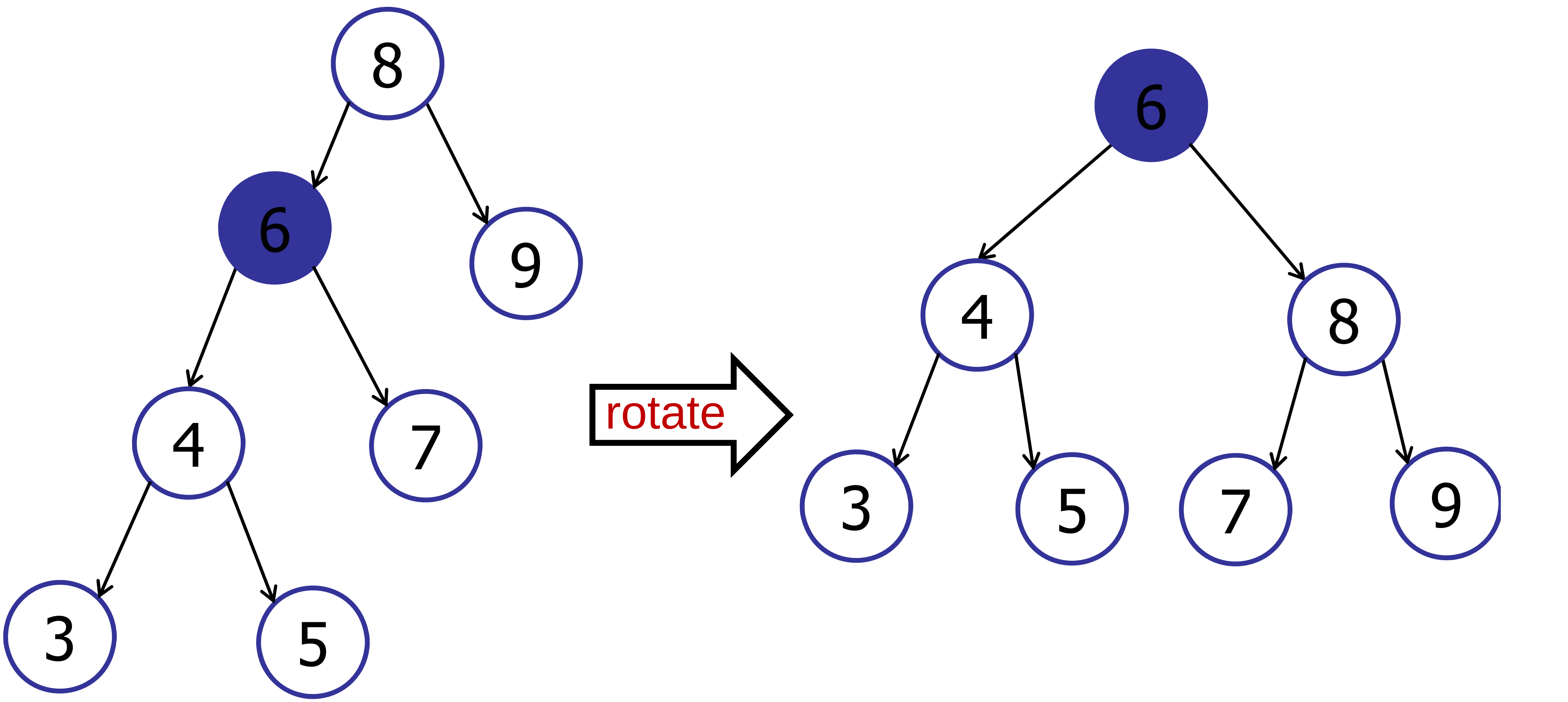
Splay Trees

Zig-Zag: Parent is left, grandparent is right.



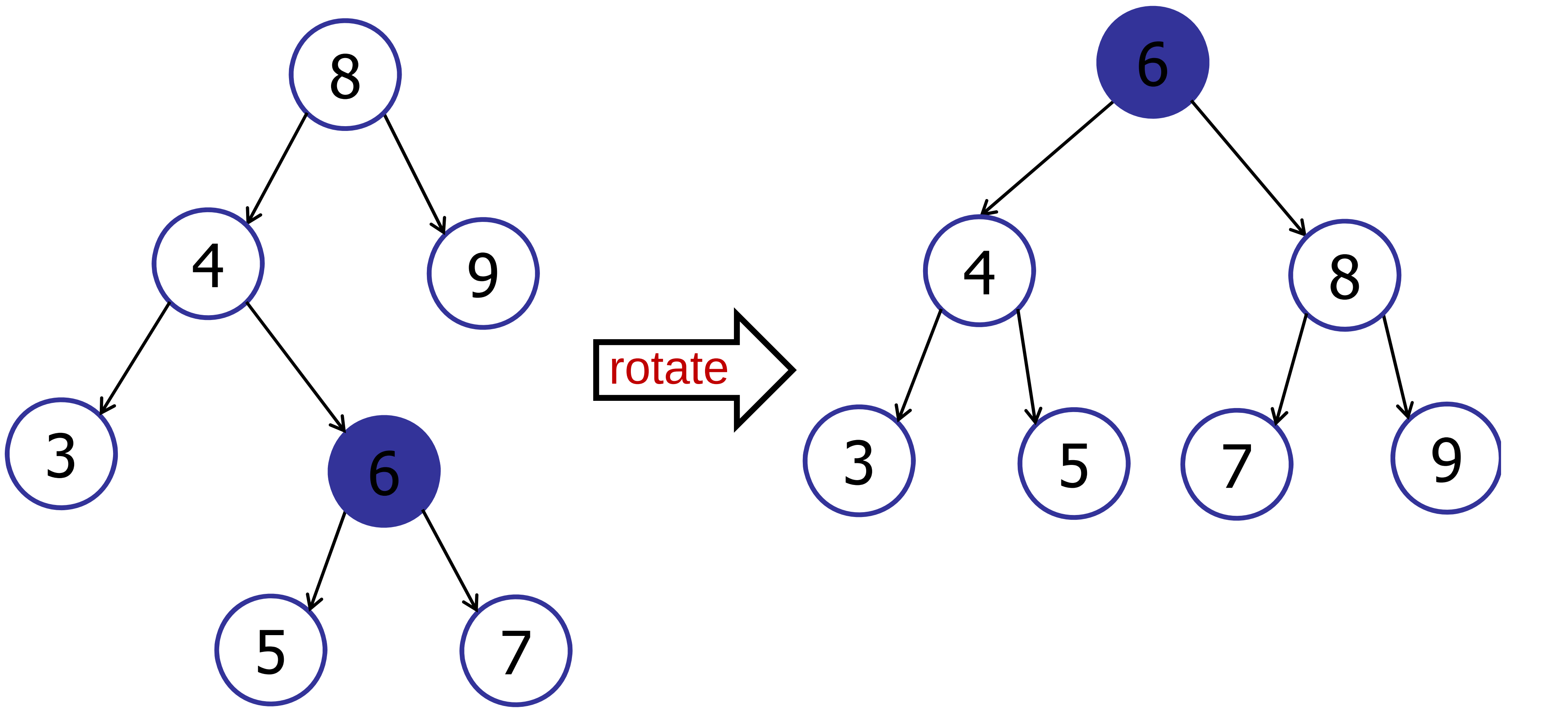
Splay Trees

Zig-Zag: Parent is left, grandparent is right.



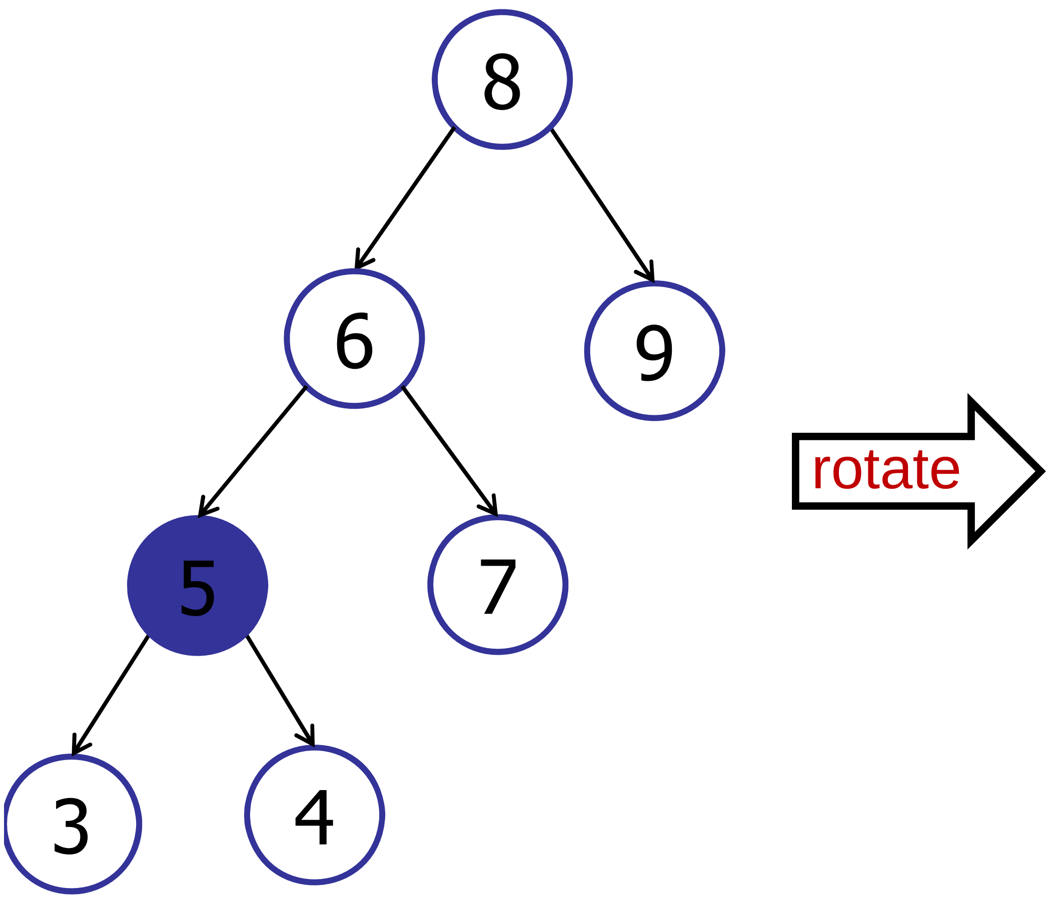
Splay Trees

Zig-Zag: Parent is left, grandparent is right.



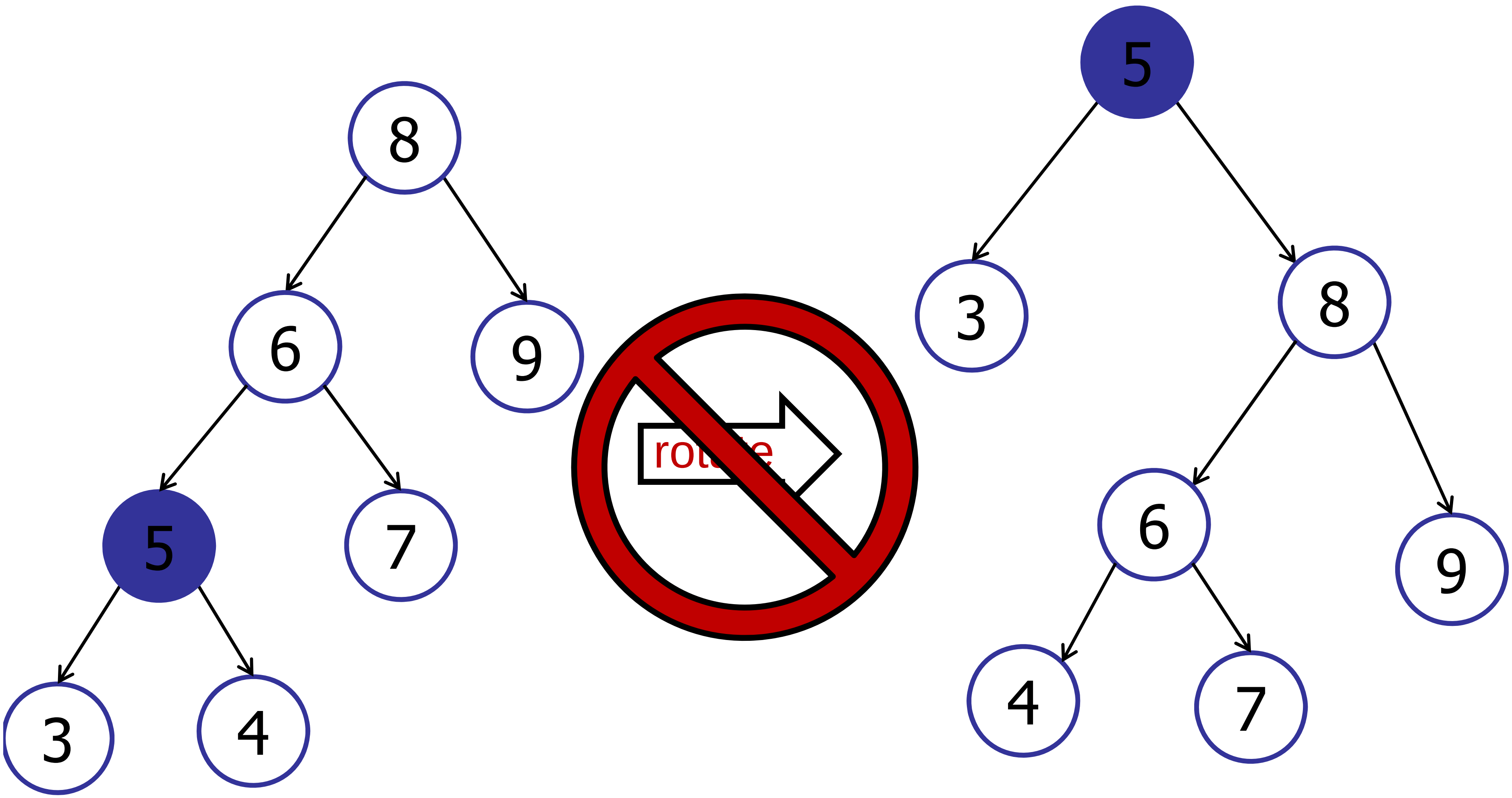
Splay Trees

Zig-Zig: Parent is right, grandparent is right.



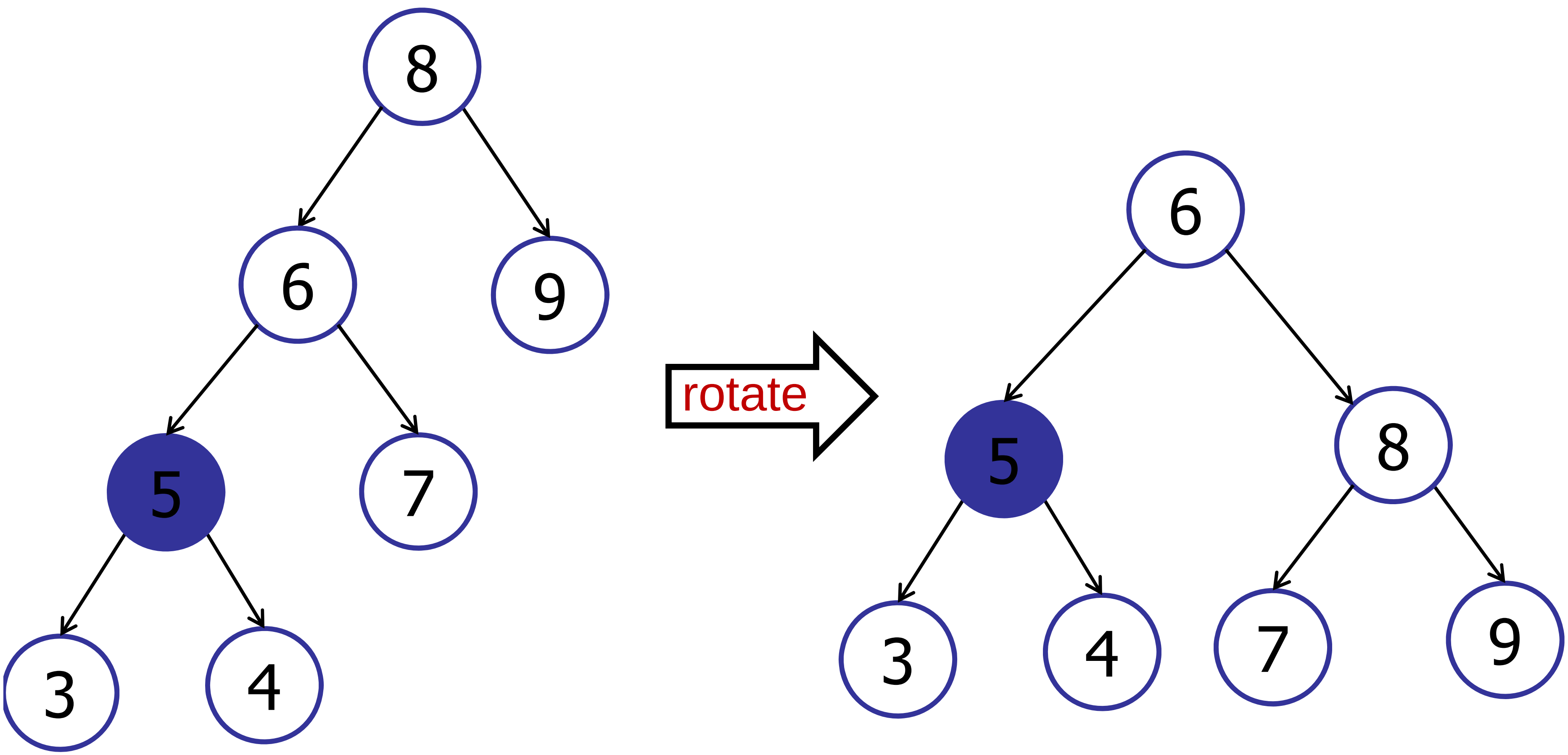
Splay Trees

Zig-Zig: Parent is left, grandparent is left.



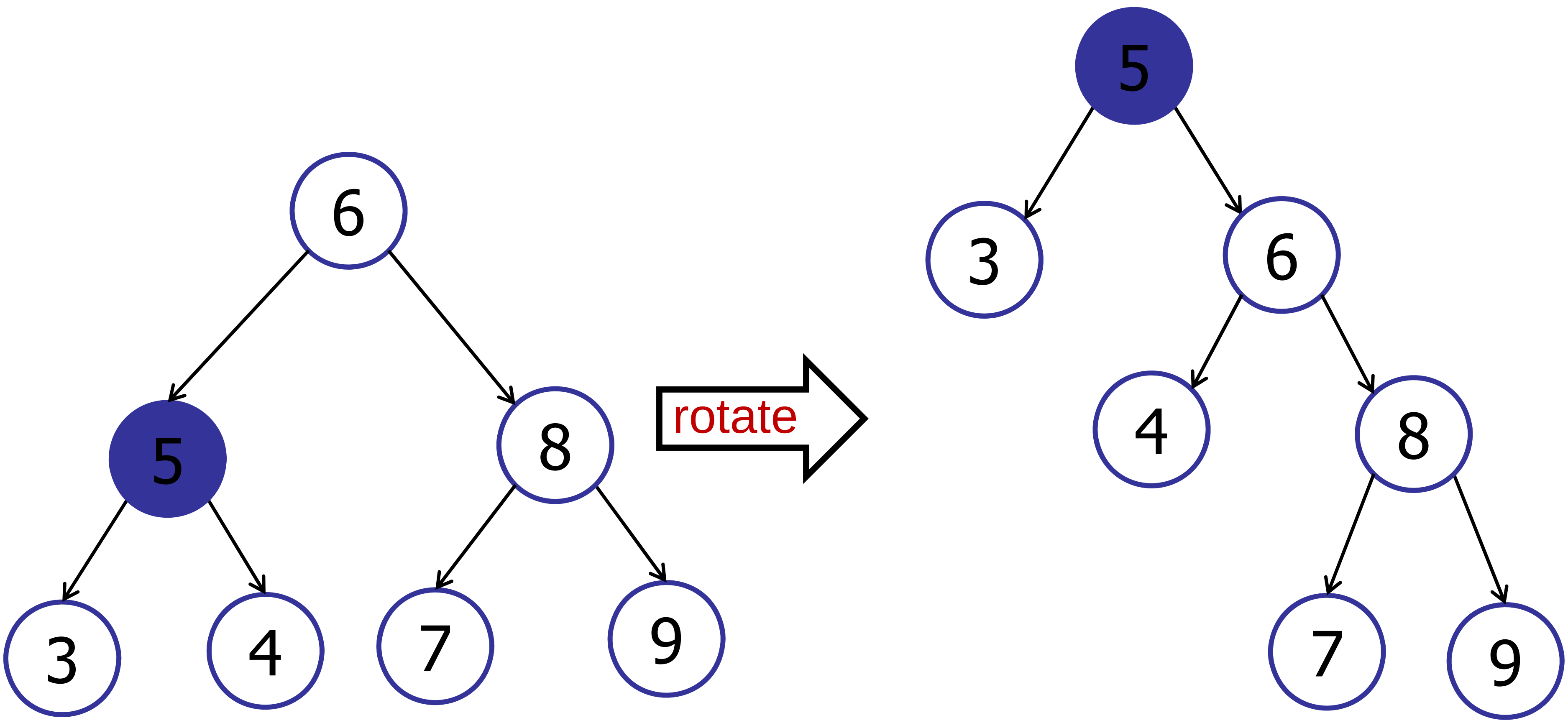
Splay Trees

Zig-Zig: Parent is left, grandparent is left.



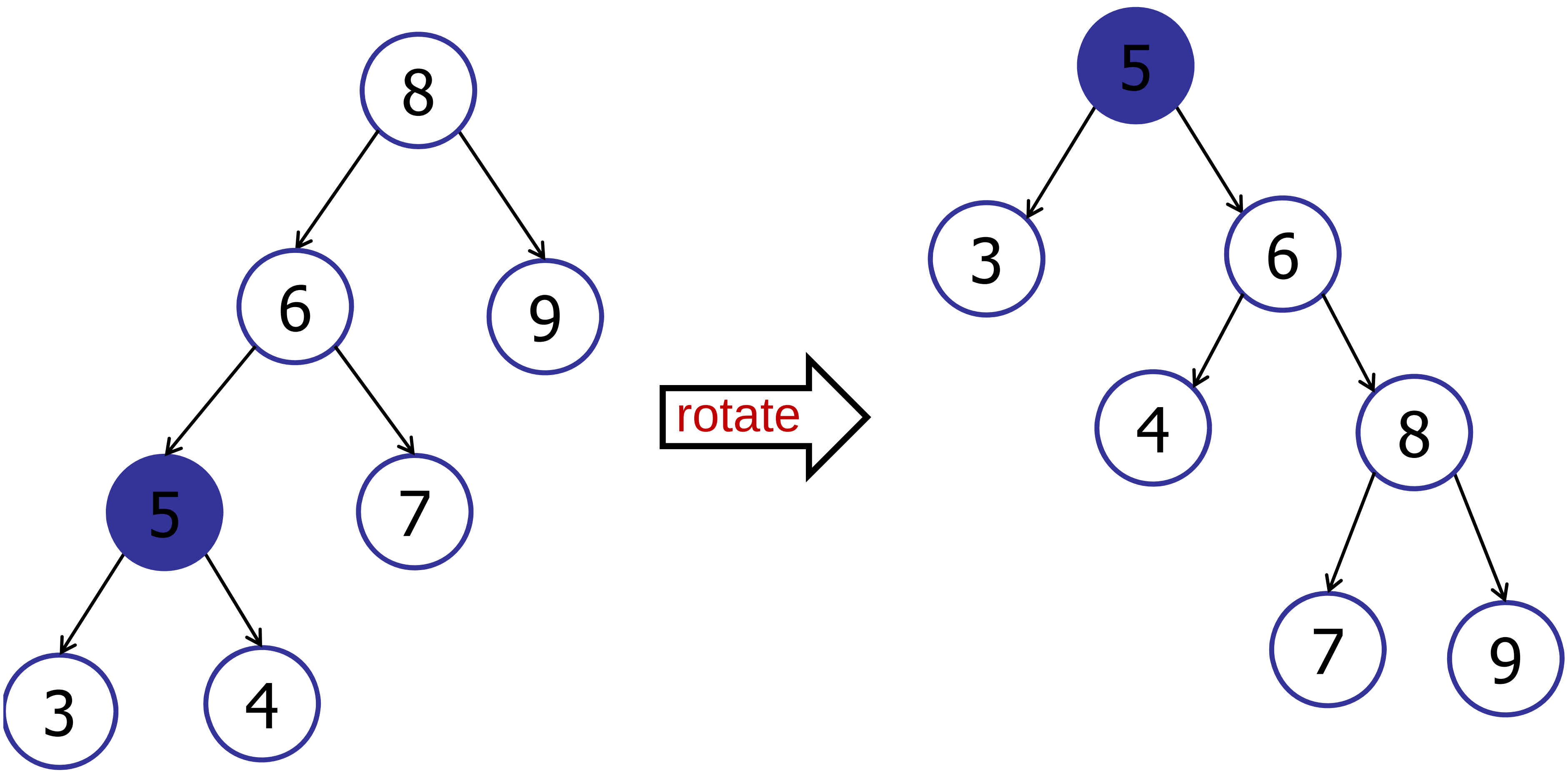
Splay Trees

Zig-Zig: Parent is left, grandparent is left.



Splay Trees

Zig-Zig: Parent is left, grandparent is left.



Splay Trees

On search/insert delete:

- Move to root, not rotate-to-root.
- Balance more along the way.
- Two levels at a time.

Mirror images are the same

Three cases:

- Zig: Parent is root.
- Zig-Zag: Parent is left, grandparent is right.
- Zig-Zig: Parent and grandparent are left children.

Only different from “Rotate-to-Root”

Splay Trees

Balance Theorem:

- Assume tree T has n nodes.
- Assume there are m operations.

The total cost is: $O((m+n) \log n)$

Splay Trees

Balance Theorem:

- Assume initially empty tree T.
- Assume there are m insert/search operations.
- Assume there are at most n inserts total.

The total cost is: $O(m \log n)$

Splay Trees

Scanning Theorem:

- Accessing all n elements in a splay tree, in order, costs $O(n)$.

Splay Trees

Static Optimality Theorem:

- Let q_i be the number of times i is accessed.
- Assume there are m searches and n nodes.

The total cost is: $O(m + \Sigma\, q_i \log(m/q_i))$

Balanced Search Trees

Summary:

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