CS2040S
Data Structures and Algorithms

AVL Trees

Puzzle of the Week:

100 prisoners. Every so often, one is chosen at random to enter a room with a light bulb. You can turn the light bulb on or off.

- WIN if one prisoner announces correctly that all have visited the room.
- LOSE if announcement is incorrect.

What if, initially, the state of the light is unknown, either on or off?

Housekeeping

PS4 Release 12 Feb 15:15

- Due: 18 Feb 23:59

Implement Scapegoat Trees!

- **Beware: “It’s easy to introduce bugs”**
  - **One of your TAs**

Take care to make sure you’re careful when implementing it.

Todays Plan

On the importance of being balanced

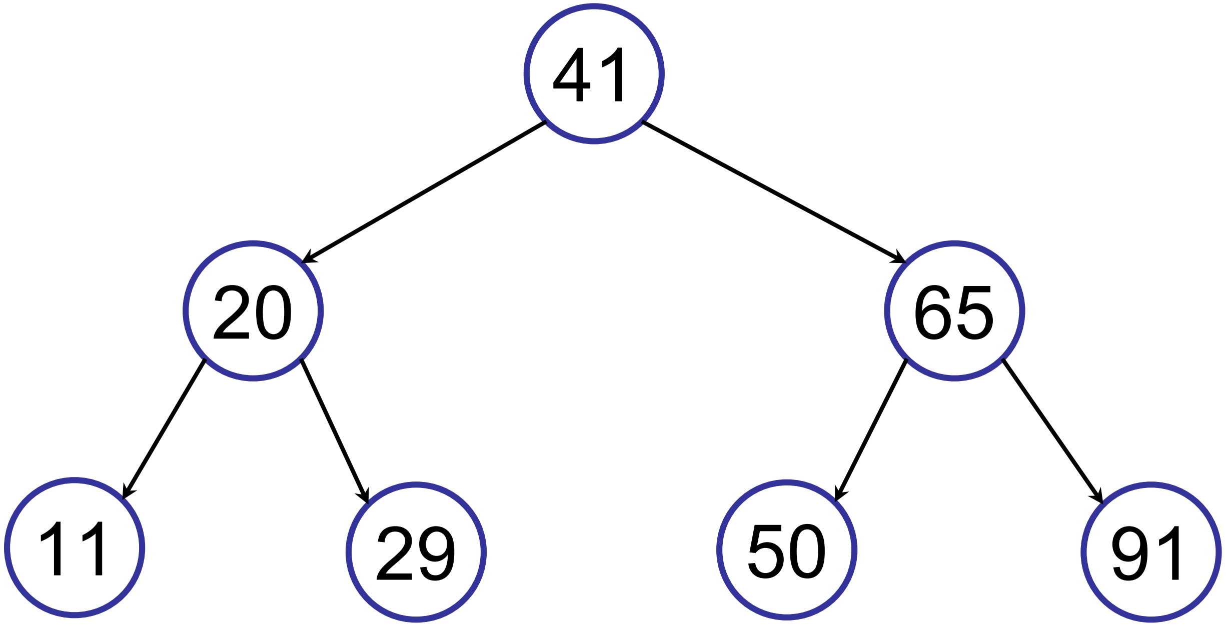
- Height-balanced binary search trees
- AVL trees
- Rotations
- Insertion Recap
- Deletion

Recap: Dictionary Interface

A collection of (key, value) pairs:

```
interface  IDictionary
--------------------------------------------------------------------------
     void  insert(Key k, Value v)       insert (k,v) into table
    Value  search(Key k)                get value paired with k
      Key  successor(Key k)             find next key > k
      Key  predecessor(Key k)           find next key < k
     void  delete(Key k)                remove key k (and value)
  boolean  contains(Key k)              is there a value for k?
      int  size()                       number of (k,v) pairs
```

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-tree

Binary Search Tree

Modifying Operations: O(h)

- insert
- delete

Query Operations: O(h)

- search
- predecessor, successor
- findMax, findMin

Traversals: O(n)

The Importance of Being Balanced

Operations take O(h) time

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

Key definition

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

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.

Invariant

AVL Trees [Adelson-Velskii & Landis 1962]

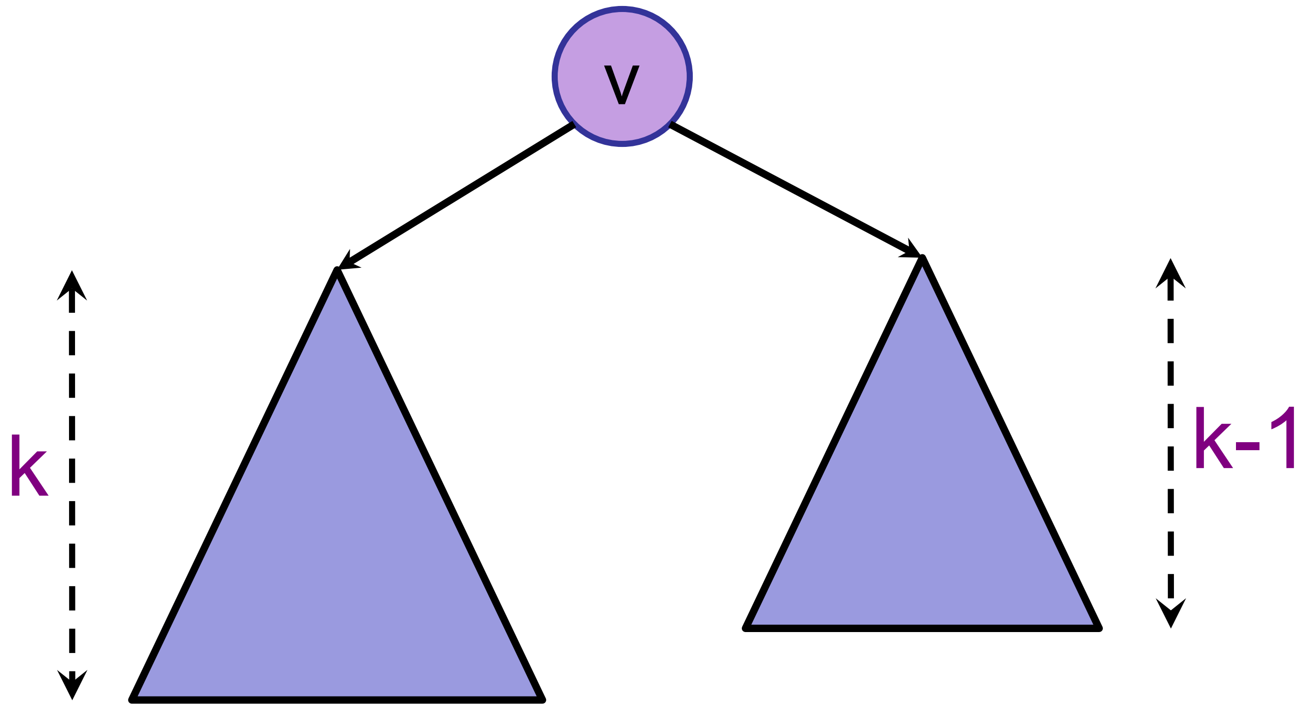
AVL Trees [Adelson-Velskii & Landis 1962]

Key definition

Step 1: Define Invariant

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

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



Height-Balanced Trees

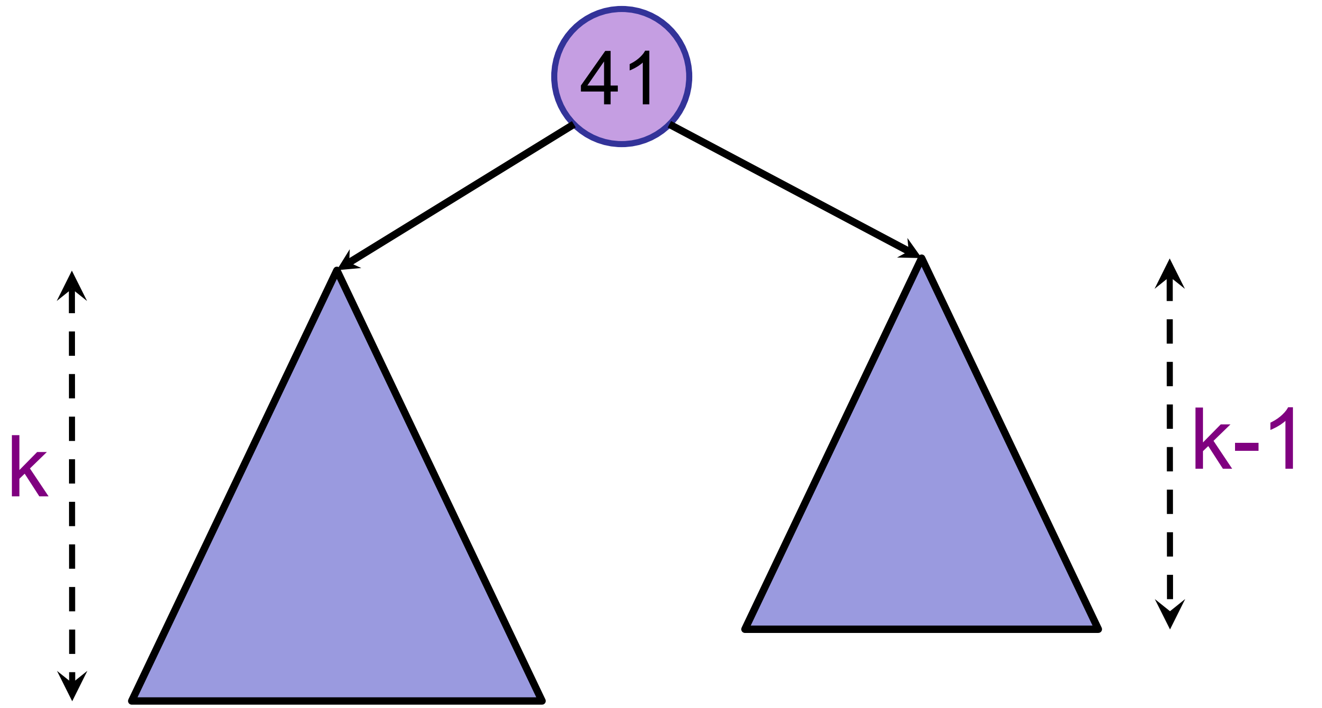
Theorem:

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

- A height-balanced tree is balanced.

AVL Trees [Adelson-Velskii & Landis 1962]

Step 2: Show how to maintain height-balance



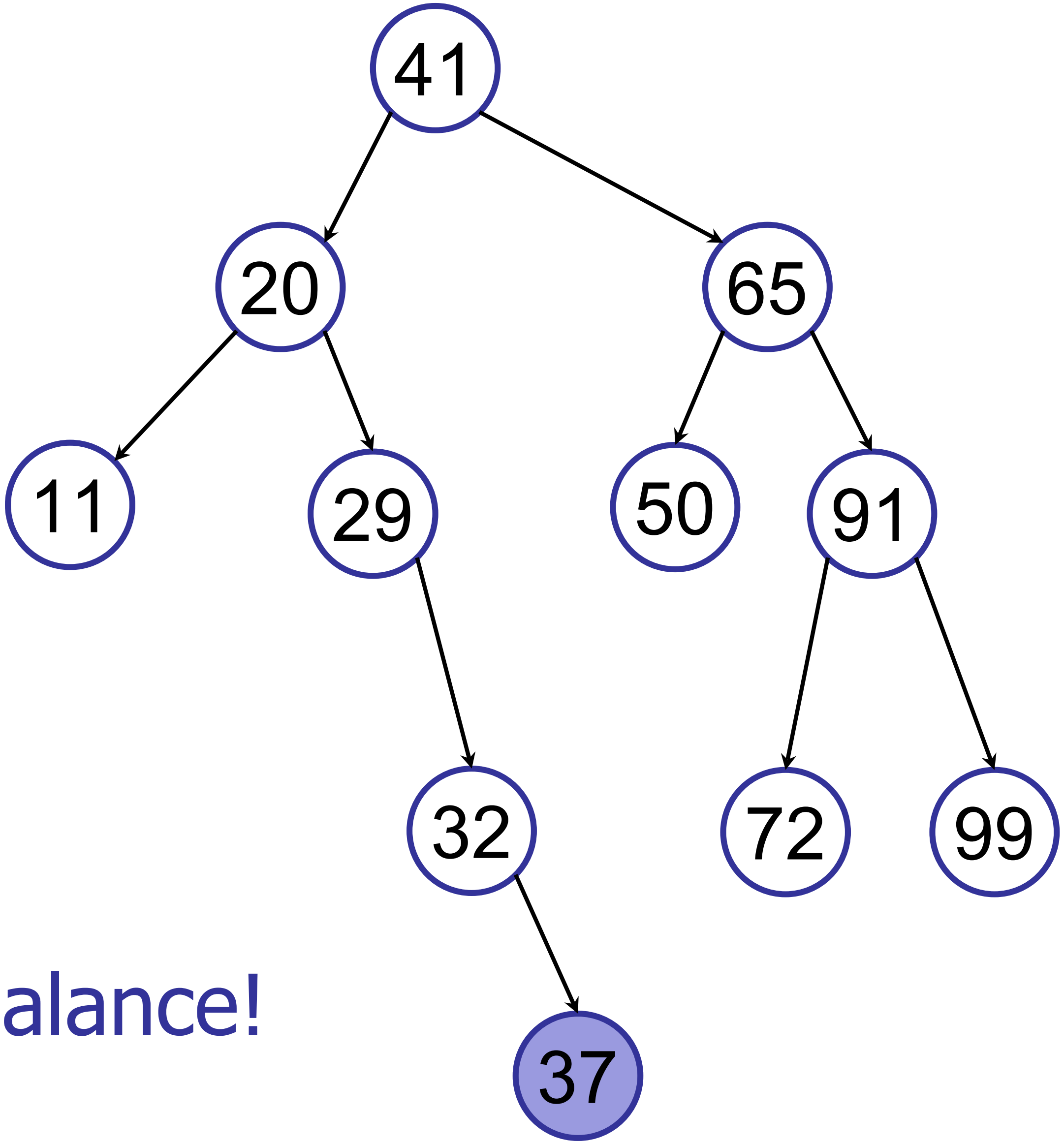
Inserting in an AVL Tree

Initially balanced

insert(37)

No longer balanced after insertion!

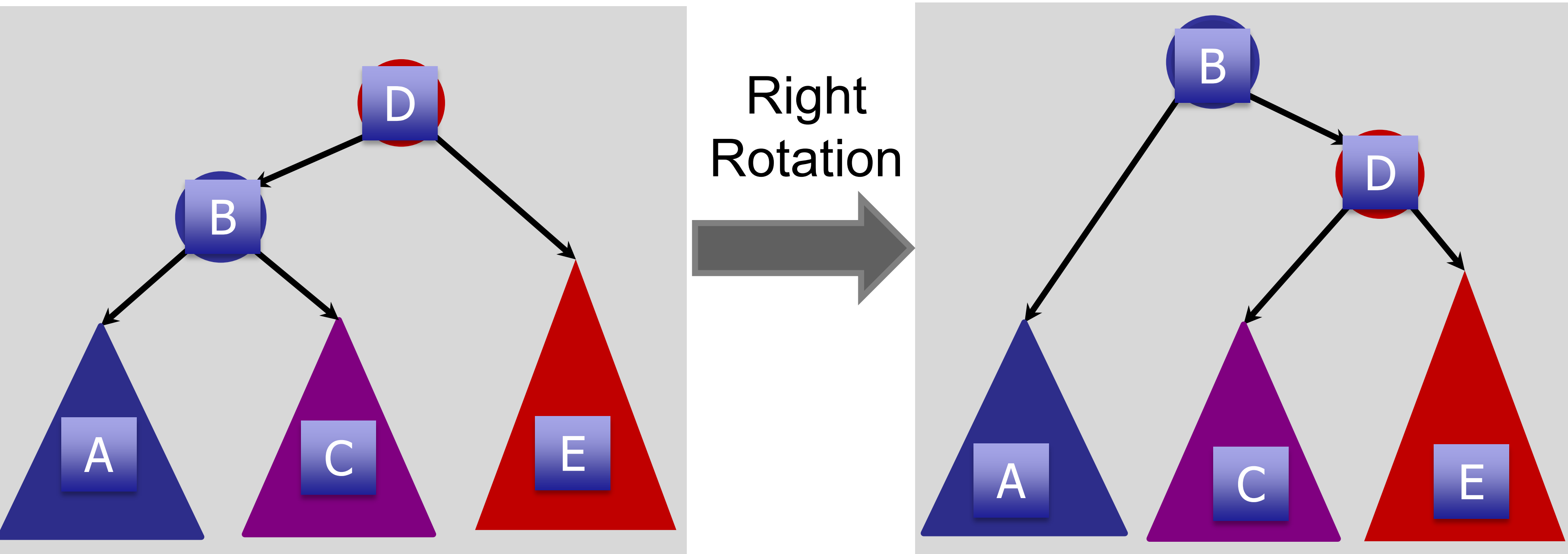
Use rotations to rebalance!



Quick review: a rotation costs:

1. $O(1)$
2. $O(\log n)$
3. $O(n)$
4. $O(n^2)$
5. $O(2^n)$

Tree Rotations

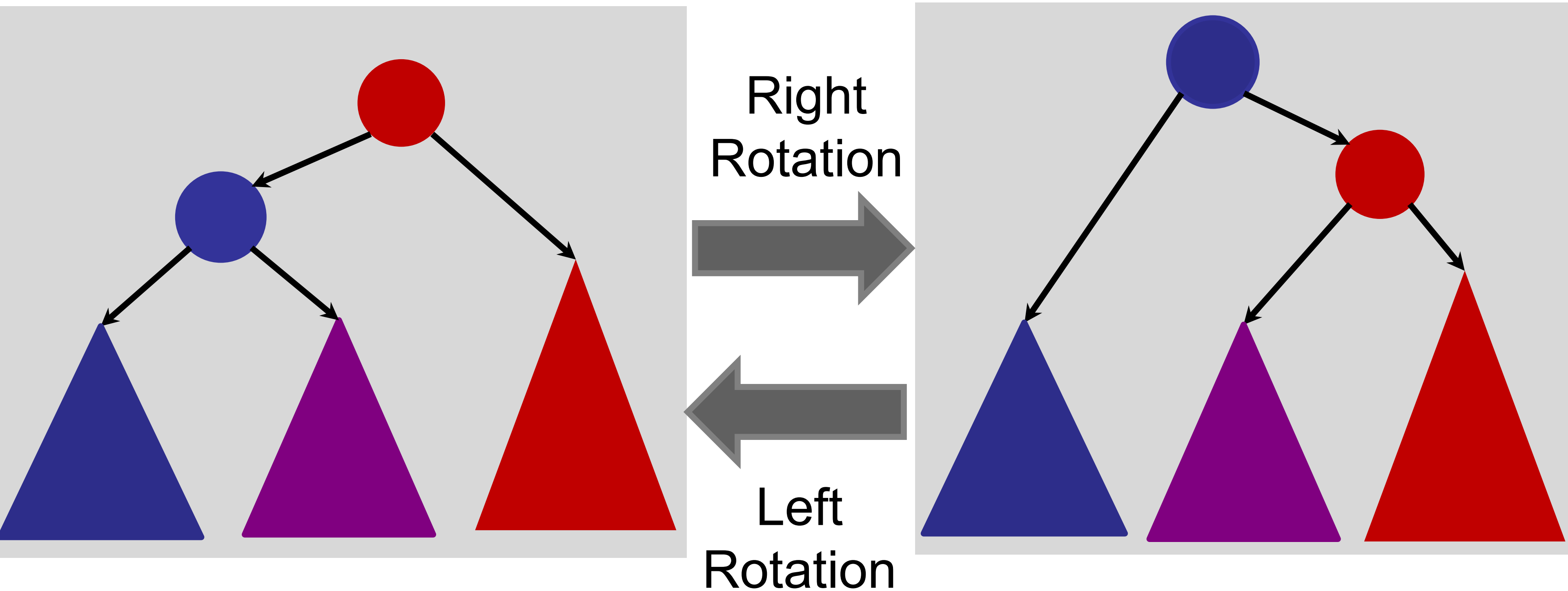


$A < B < C < D < E$

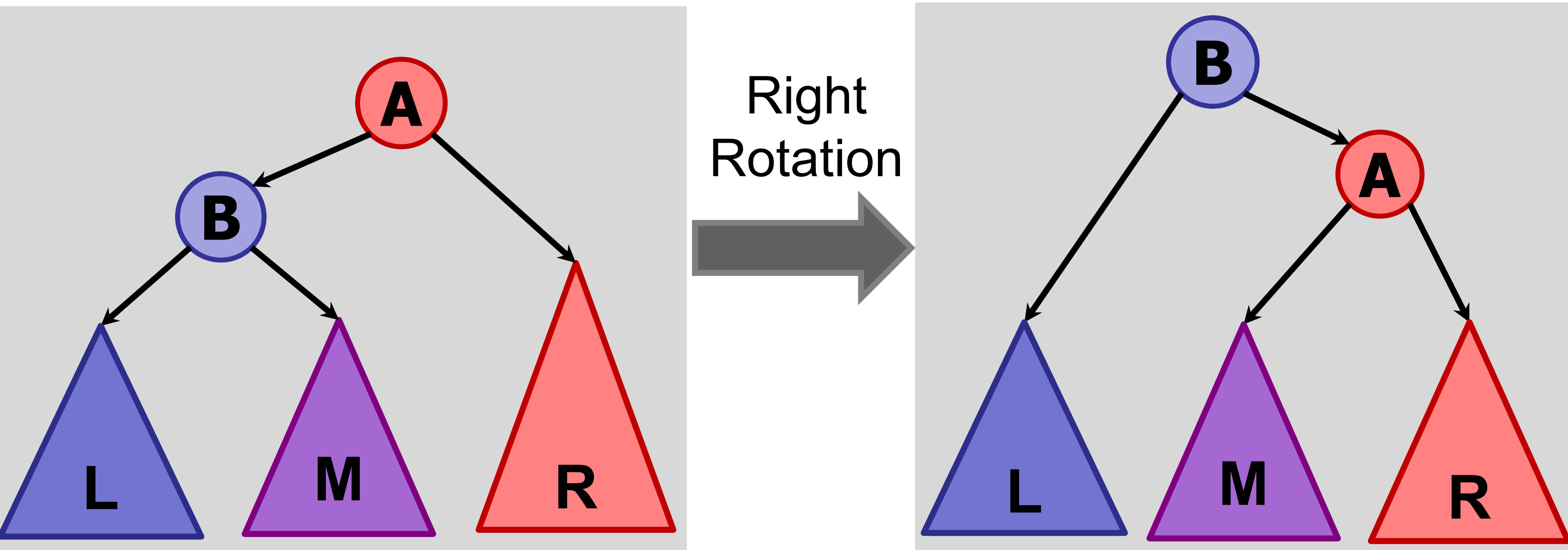
Rotations maintain ordering of keys.

$\Rightarrow$ Maintains BST property.

Tree Rotations



Tree Rotations

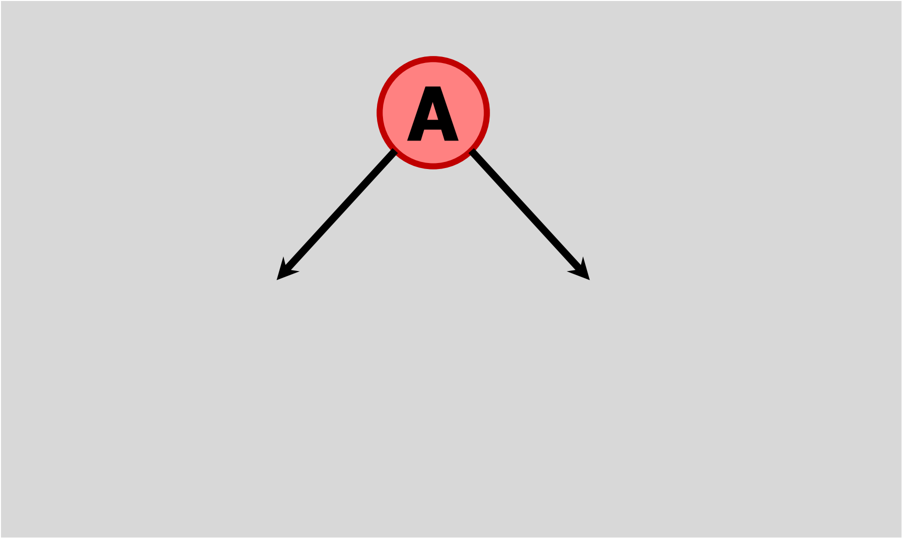


After insert:

Use tree rotations to restore balance.

Height is out-of-balance by 1

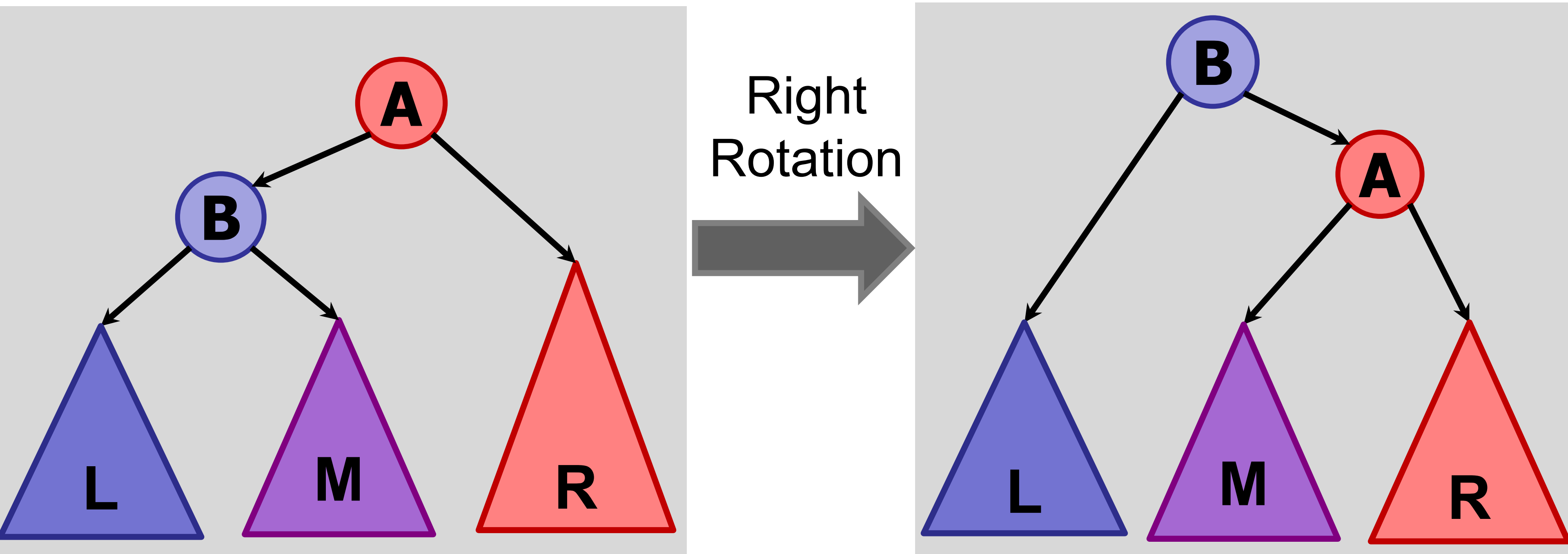
Tree Rotations



A is **LEFT-heavy** if left sub-tree has larger height than right sub-tree.

A is **RIGHT-heavy** if right sub-tree has larger height than left sub-tree.

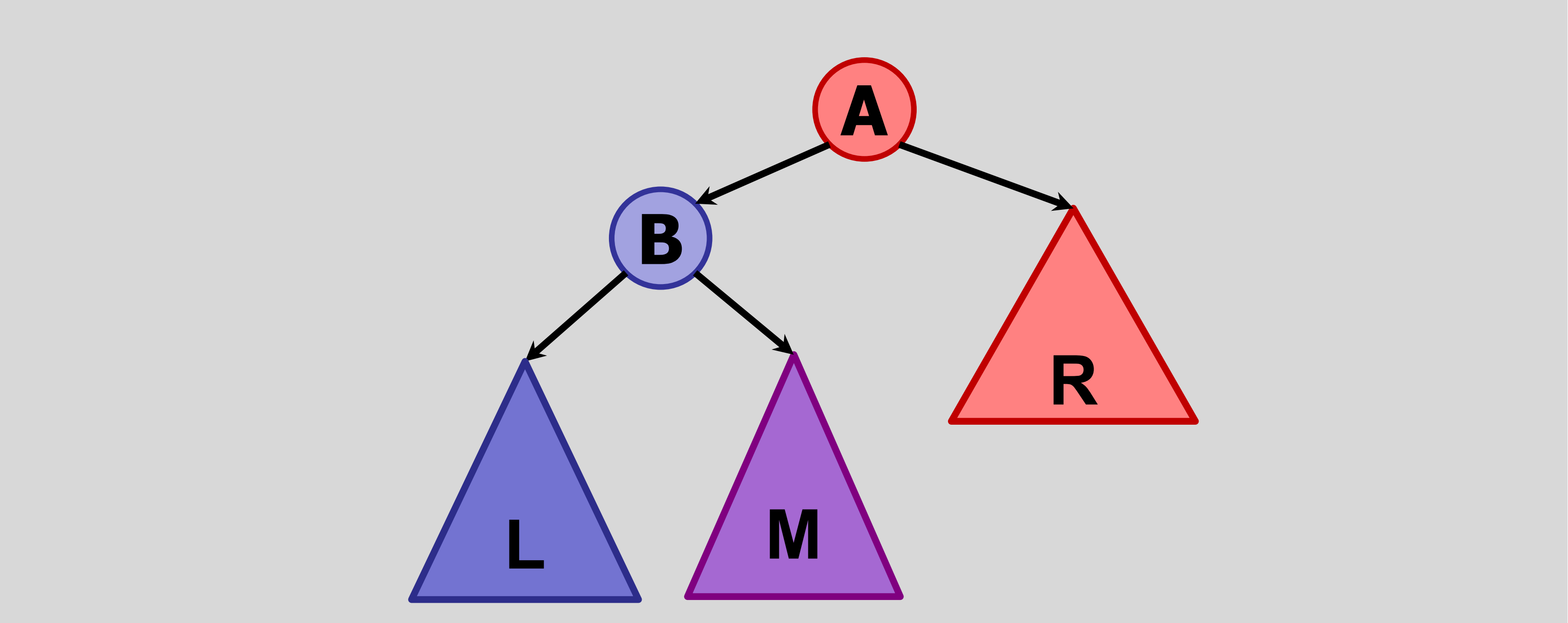
Tree Rotations



Use tree rotations to restore balance.

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

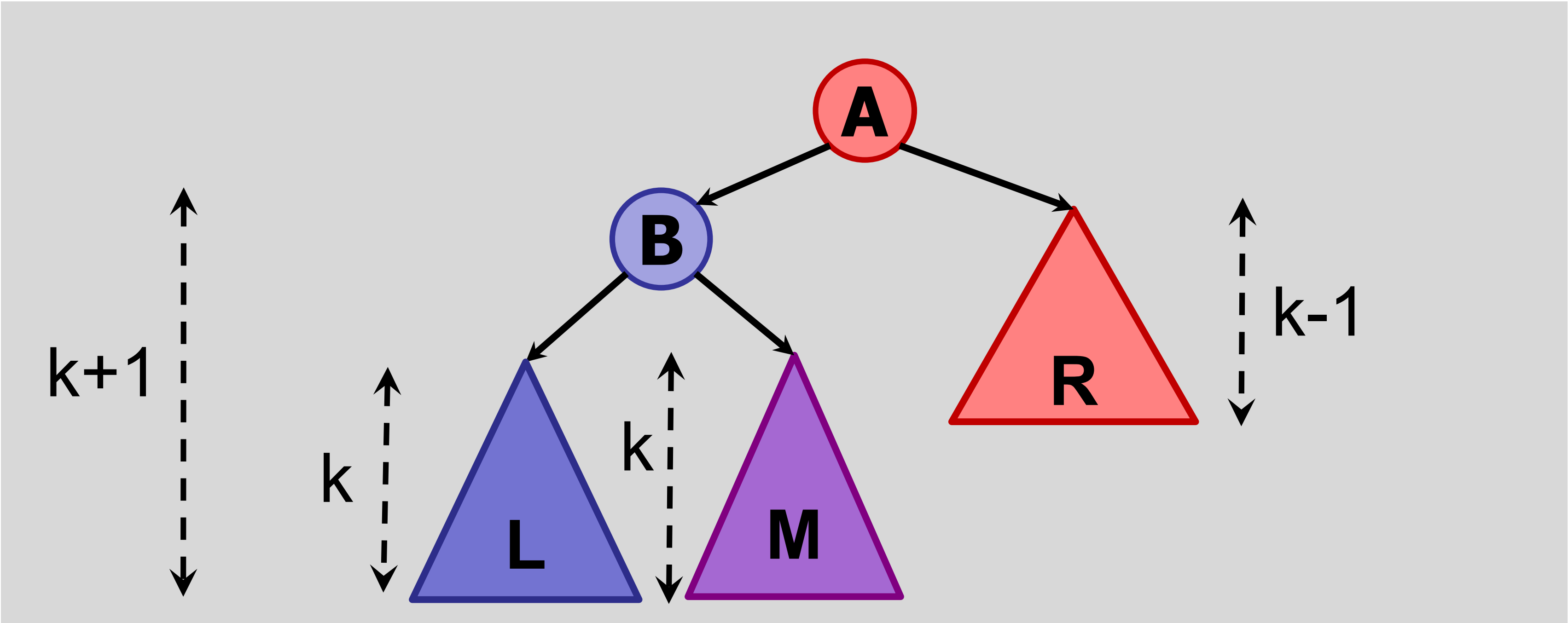
Tree Rotations



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

Assume A is **LEFT-heavy**.

Tree Rotations (Left Heavy)

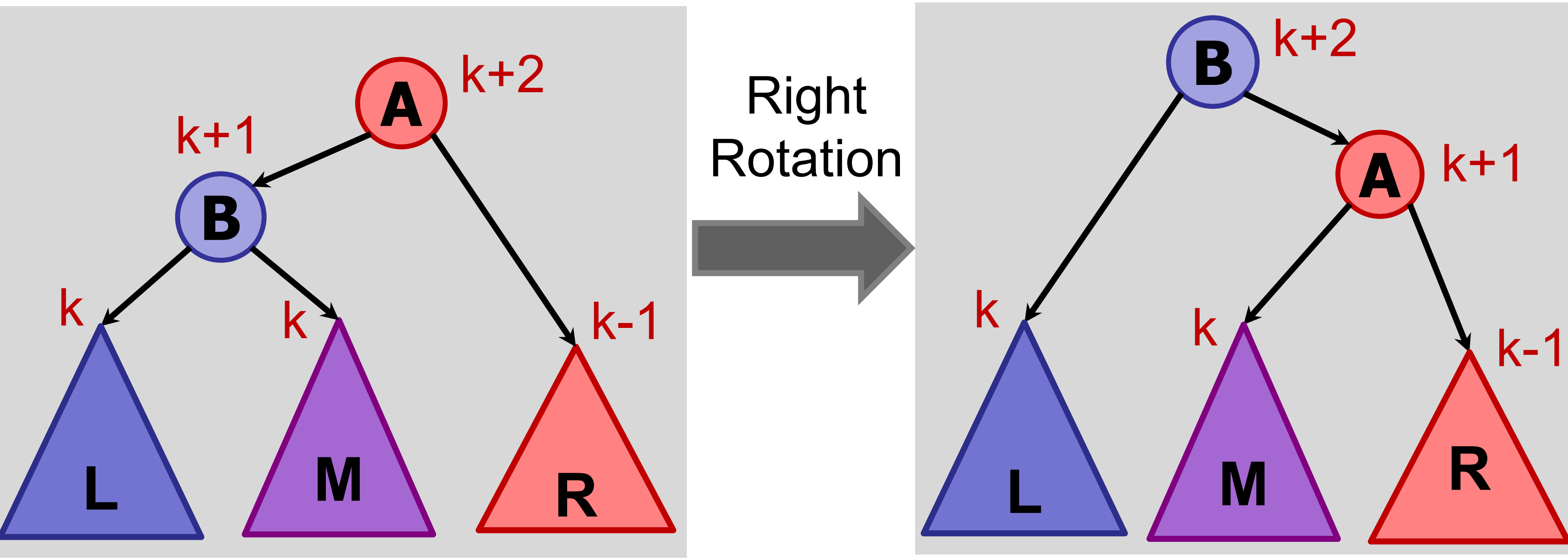


Assume **A** is the lowest node in the tree violating balance property.

Case 1: **B** is equi-height : h(**L**) = h(**M**)

h(**R**) = h(**M**) – 1

Tree Rotations

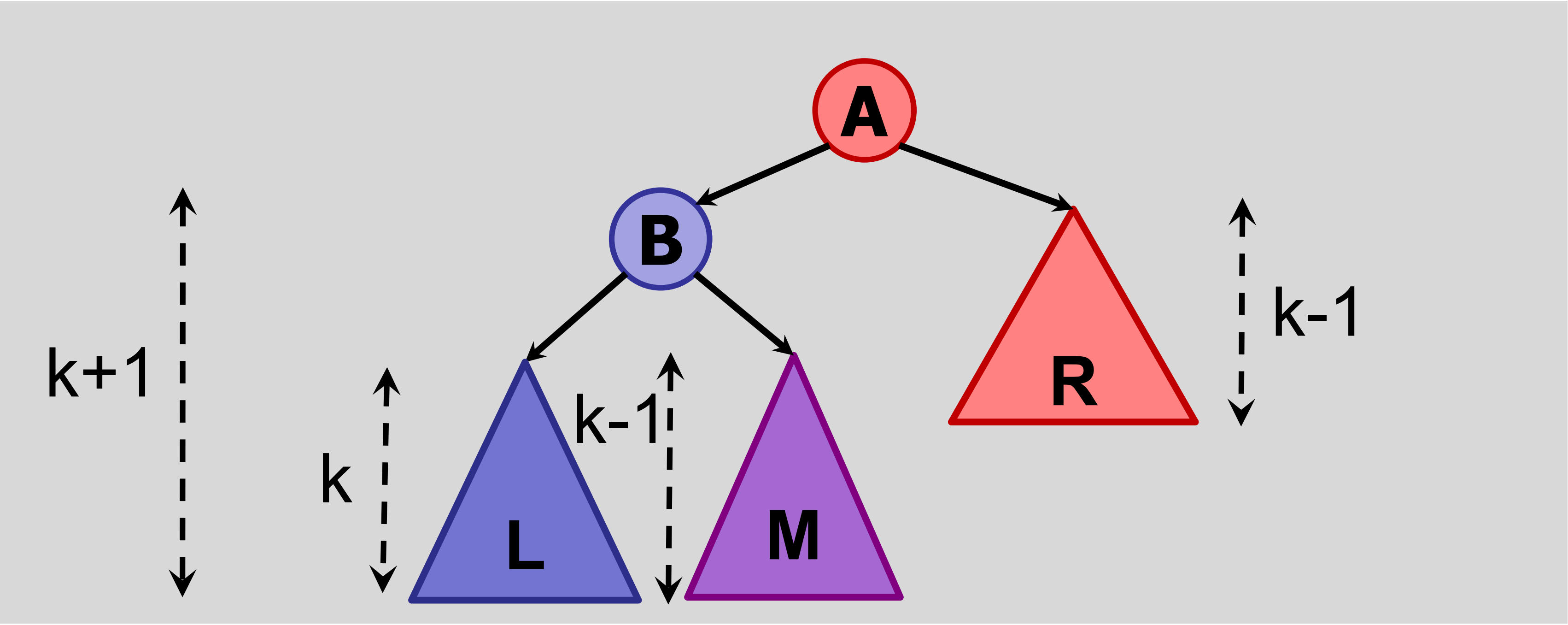


right-rotate:

Case 1: **B** is equi-height : h(**L**) = h(**M**)

h(**R**) = h(**M**) – 1

Tree Rotations (Left Heavy)

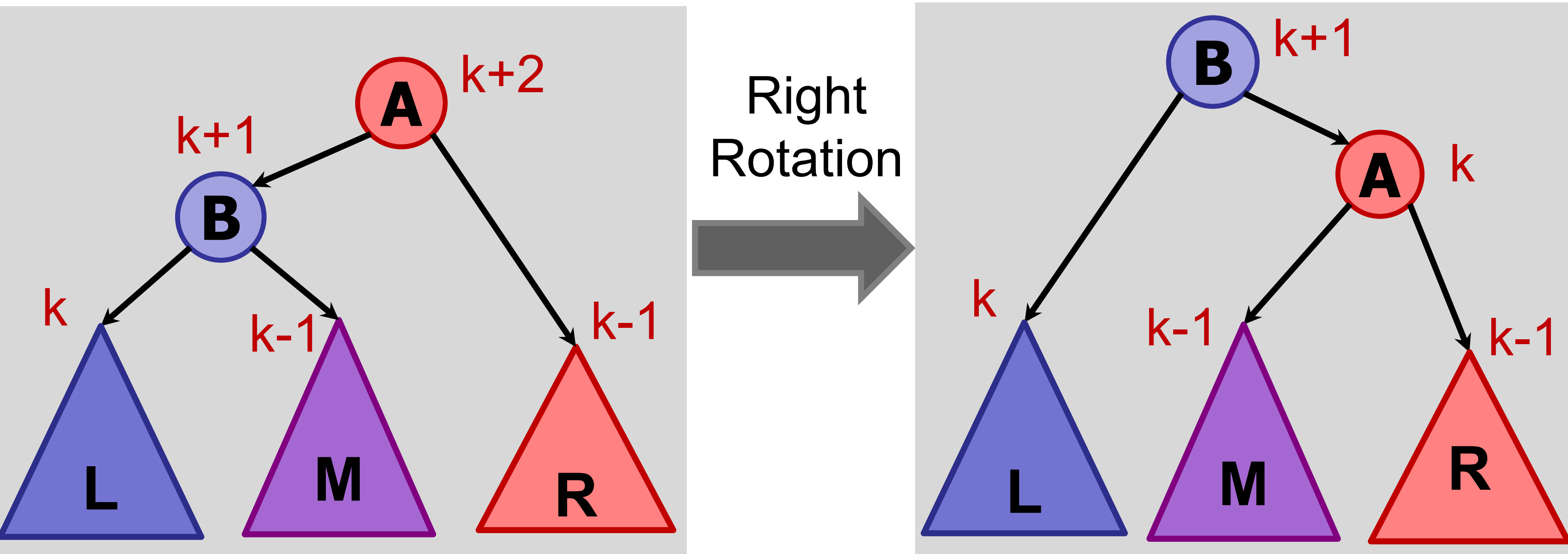


Assume **A** is the lowest node in the tree violating balance property.

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

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

Tree Rotations

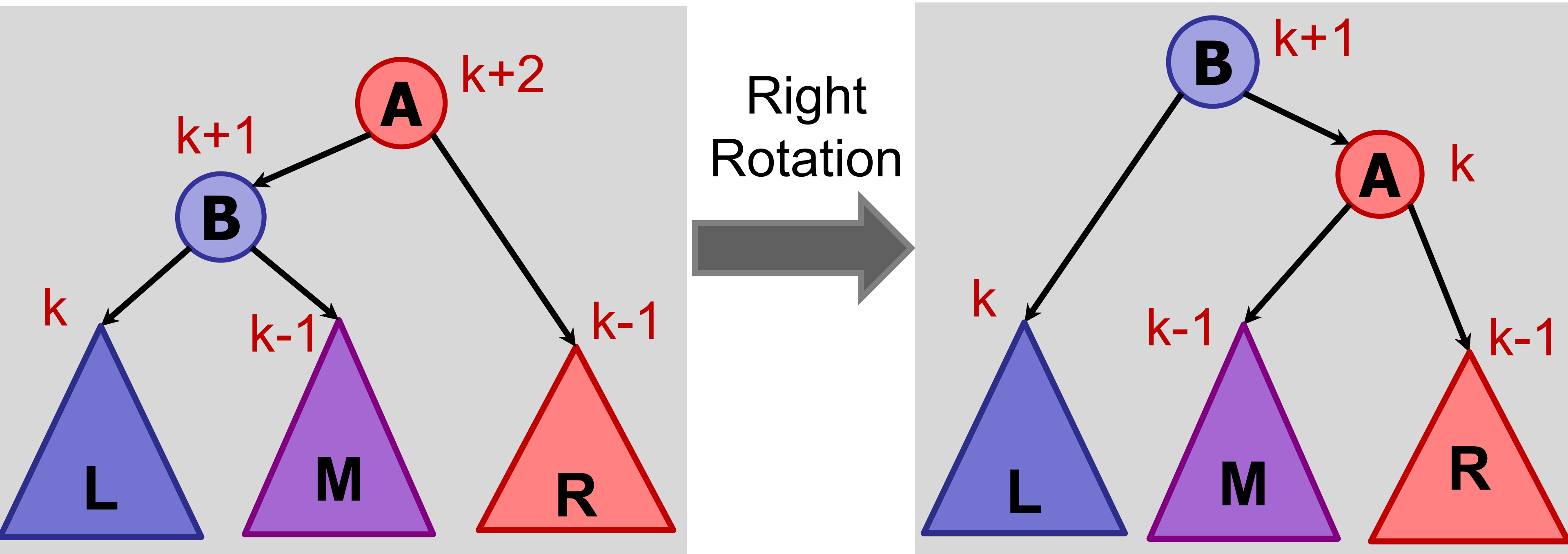


right-rotate:

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

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

Tree Rotations

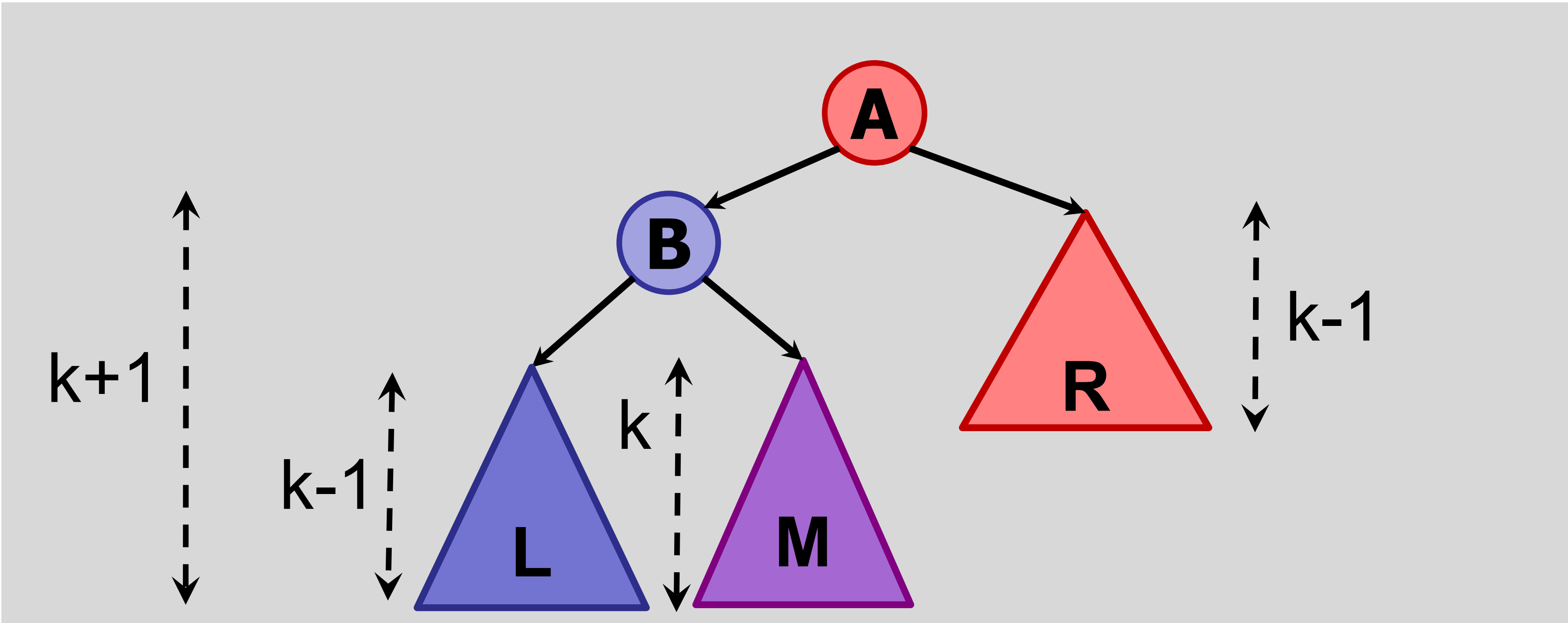


right-rotate:

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

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

Tree Rotations (Left Heavy)

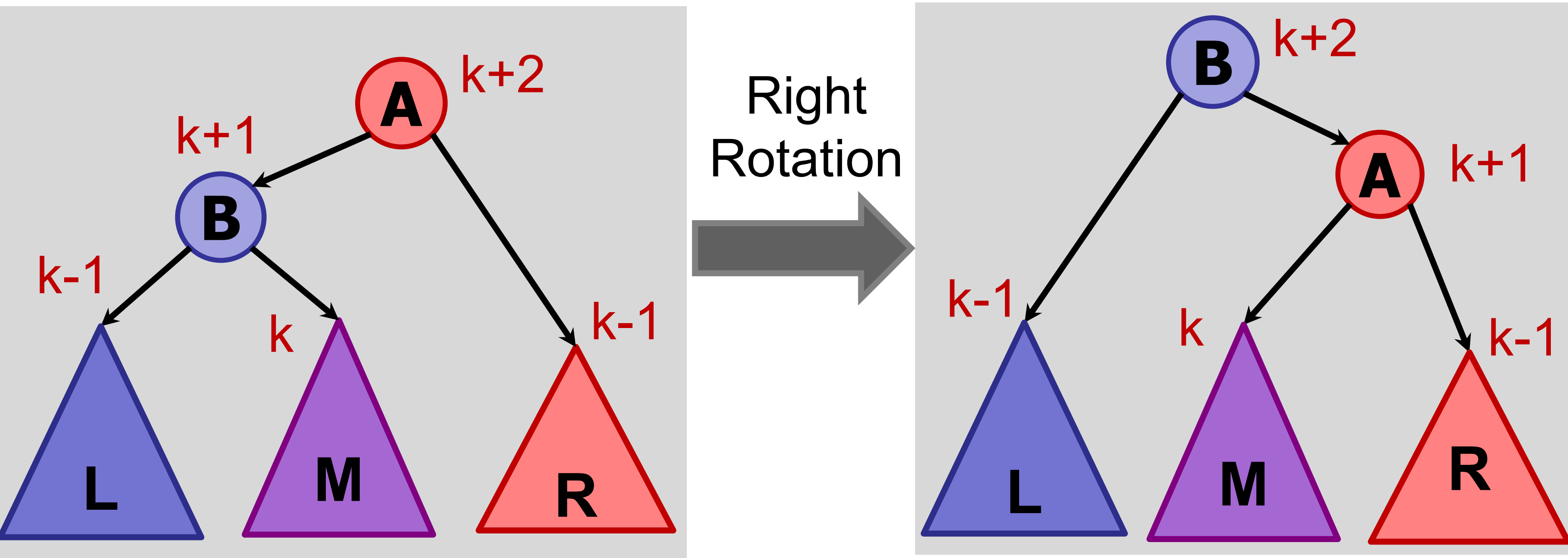


Assume **A** is the lowest node in the tree violating balance property.

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

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

Tree Rotations

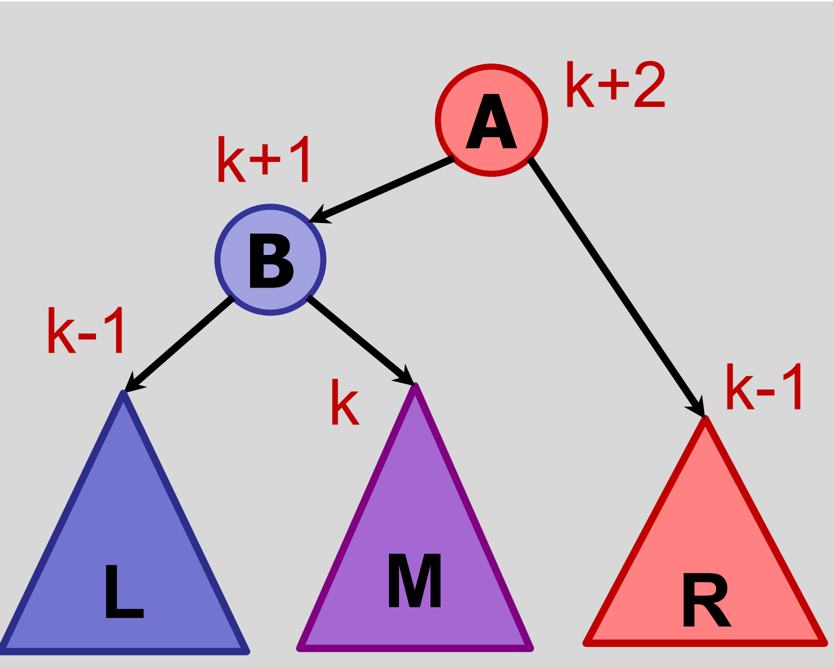


right-rotate:

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

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

Tree Rotations



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

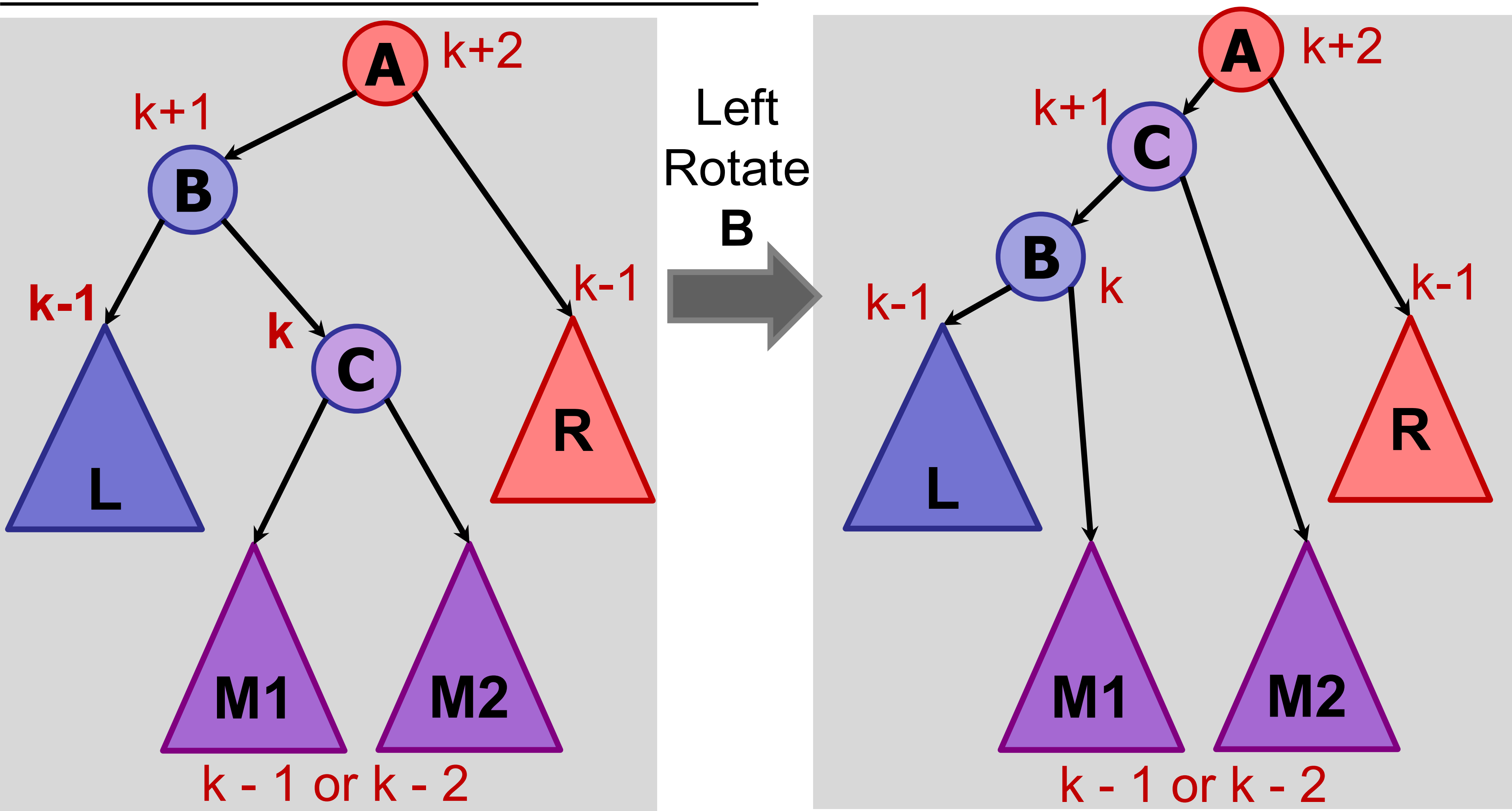
(Reduce it to a problem we have already solved!)

right-rotate:

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

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

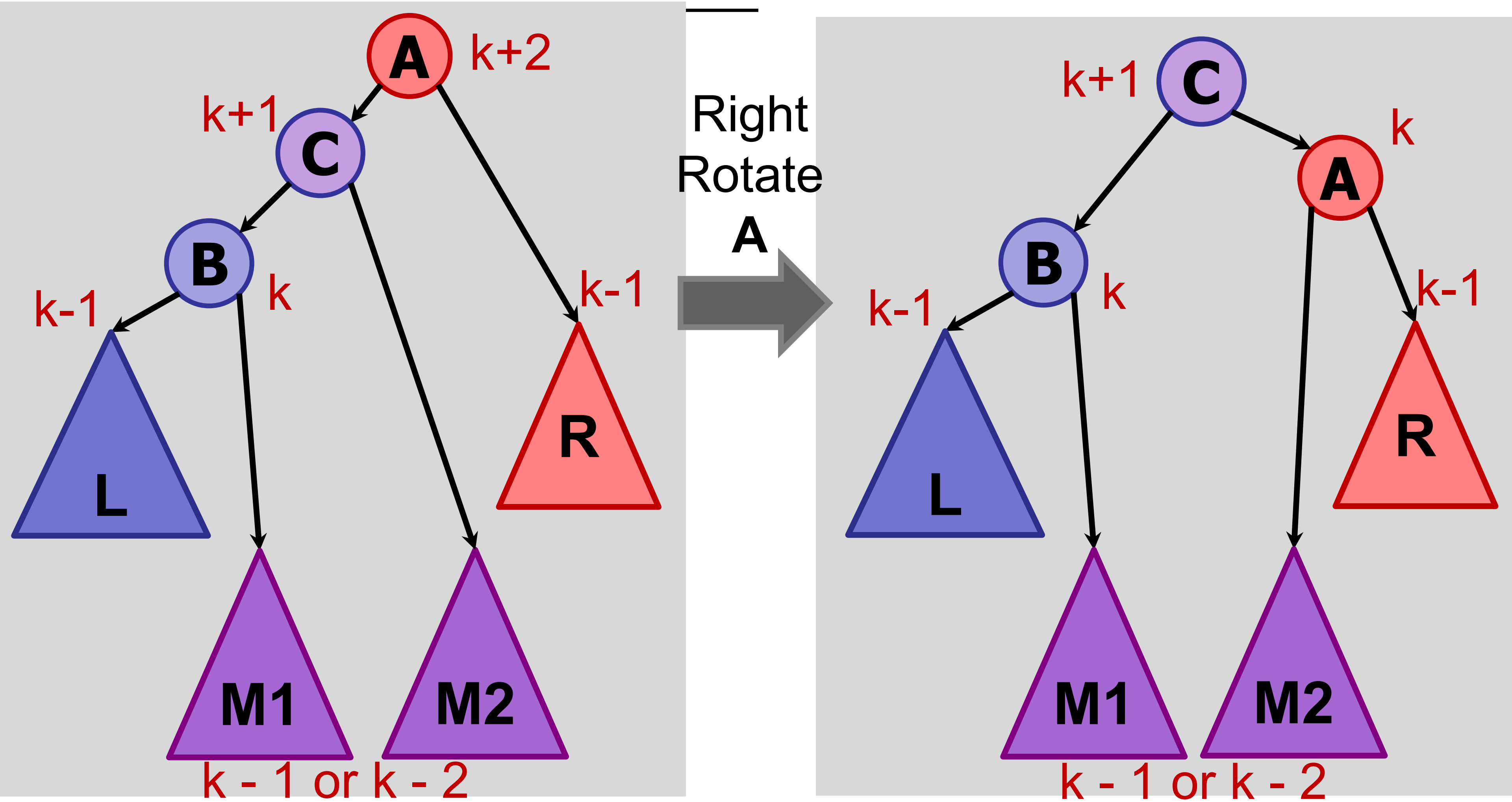
Tree Rotations



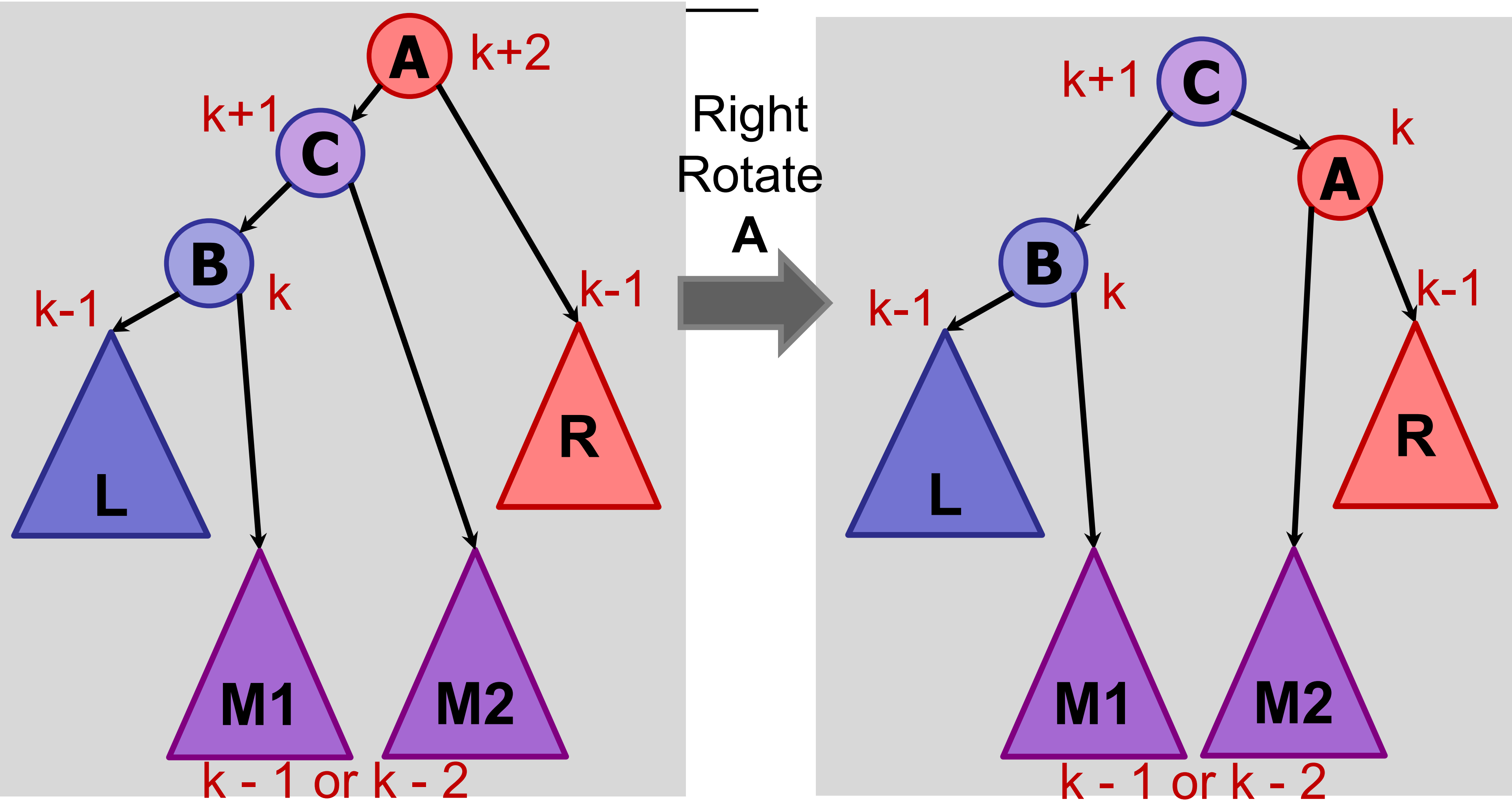
Left-rotate B

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

Tree Rotations



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....

How many rotations do you need after an insertion (in the worst case)?

1. 1
2. 2
3. 4
4. log(n)
5. 2log(n)
6. n

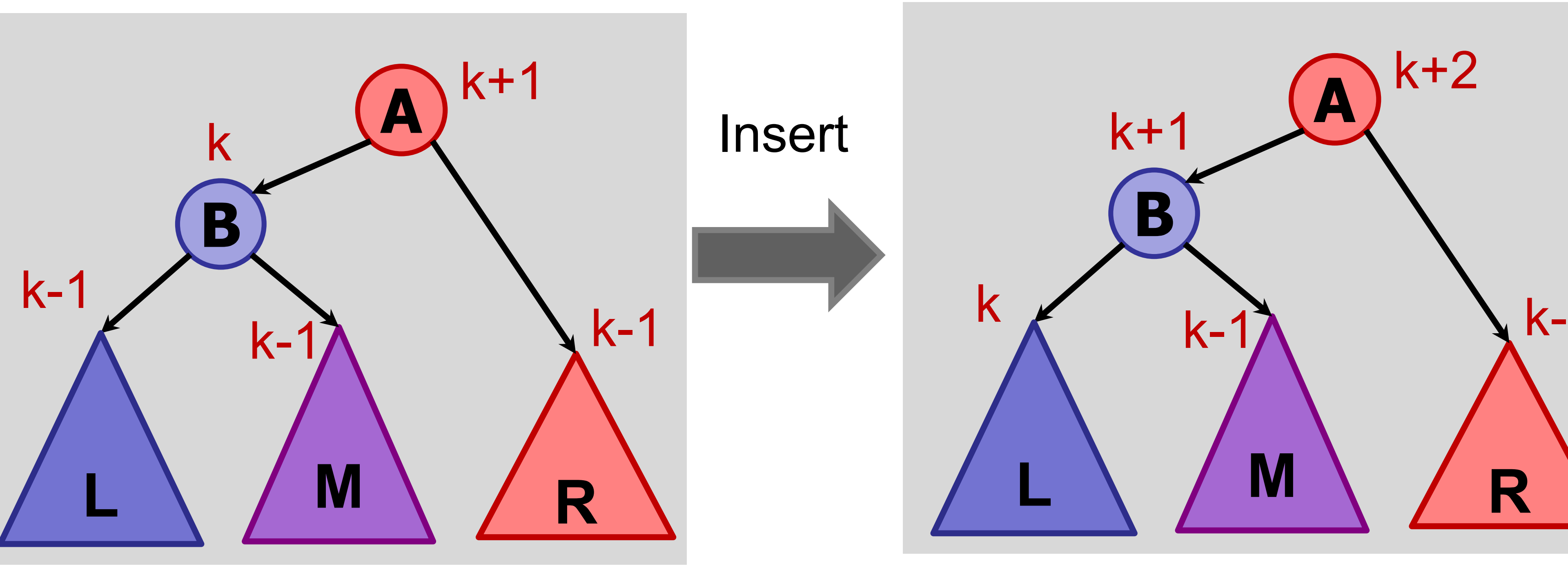
How many rotations do you need after an insertion (in the worst case)?

1. 1
2. 2 ✓
3. 4
4. log(n)
5. 2log(n)
6. n

Question:

Why isn't it 2log(n)?

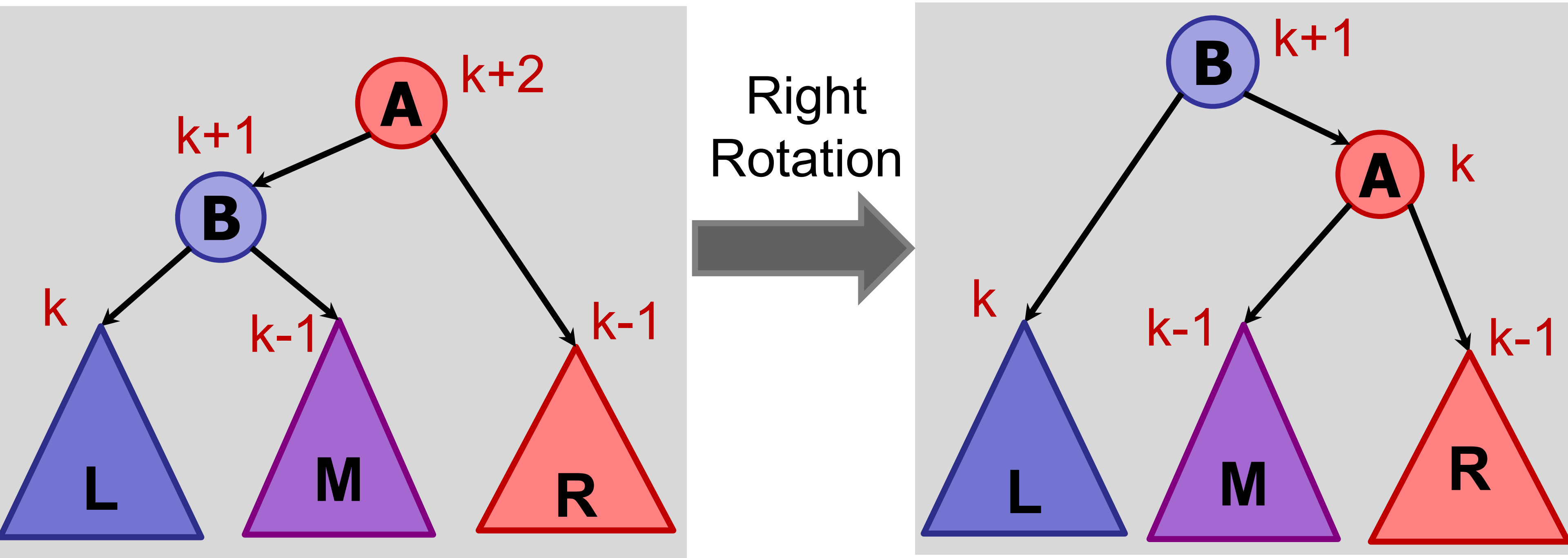
How many rotations?



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

Insert increased heights by 1.

How many rotations?

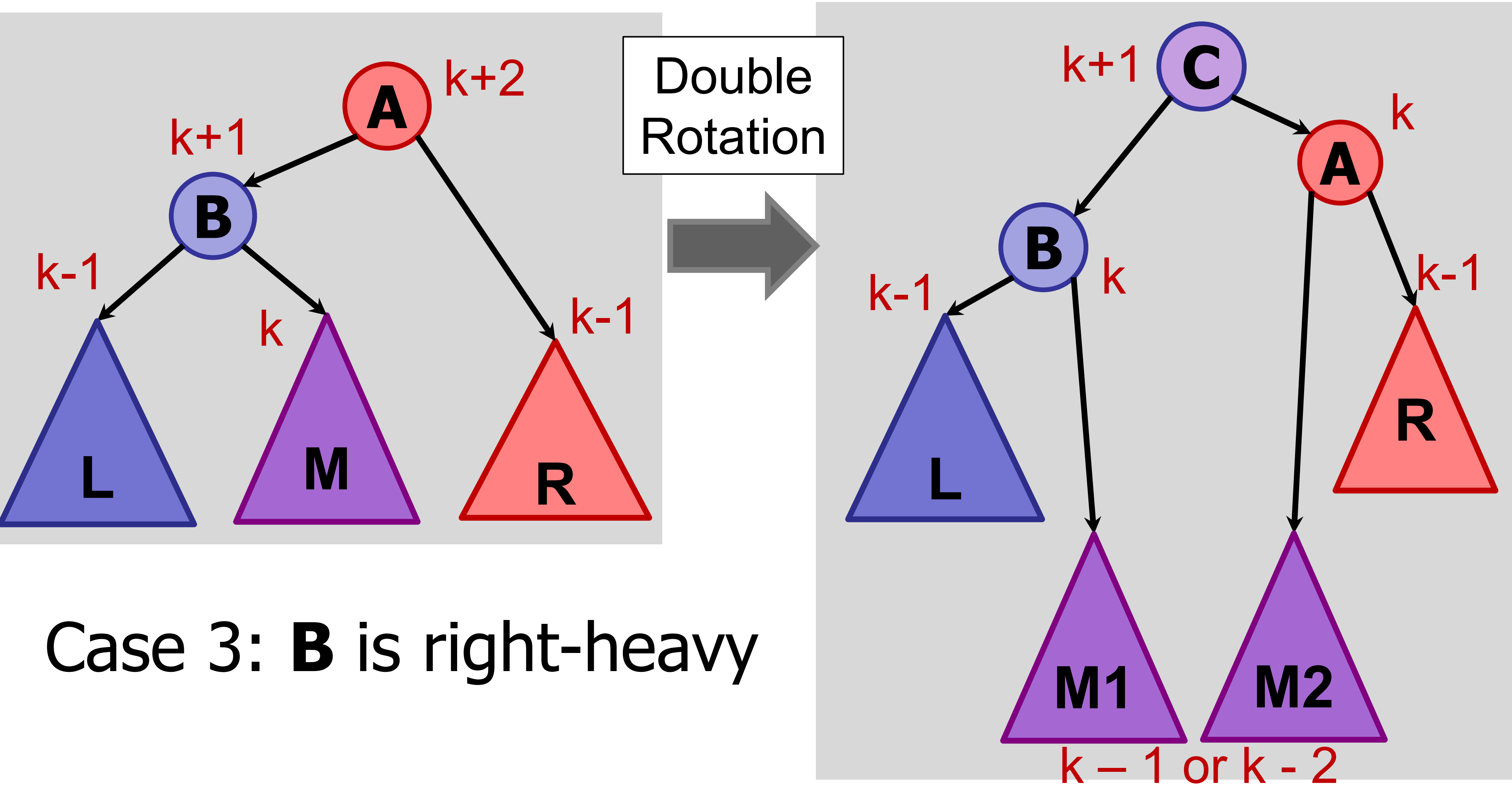


Case 2: **B** is left-heavy

Rotation reduces root height by 1.

(Everything higher in tree is unchanged!)

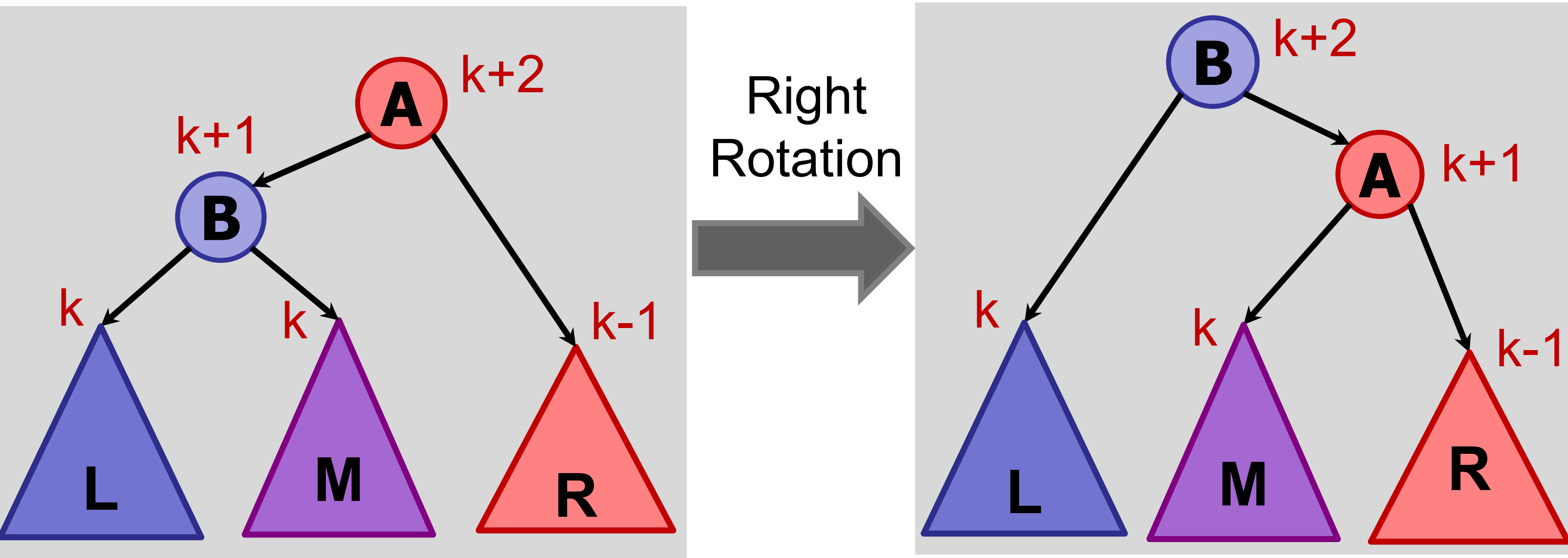
How many rotations?



Case 3: **B** is right-heavy

Rotation reduces root height by 1.

How many rotations?

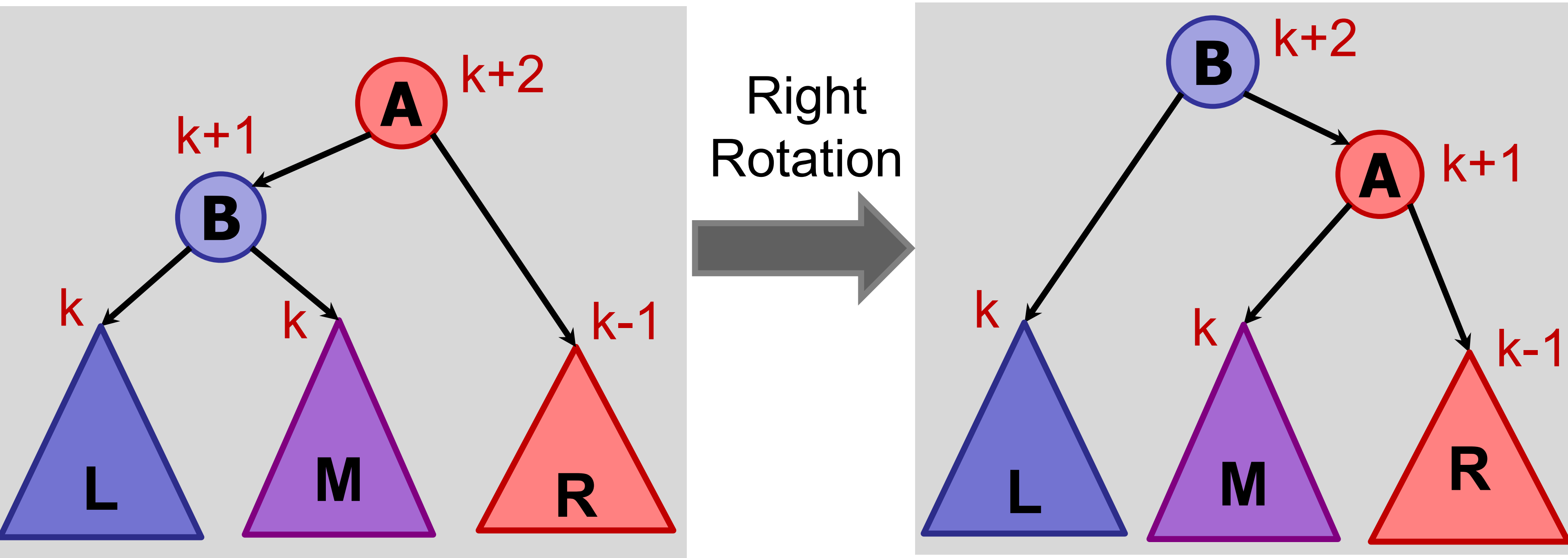


Case 1: **B** is balanced

Rotation does *not* reduce height by 1.

Challenge: figure out why this is okay!

Root height does not decrease!

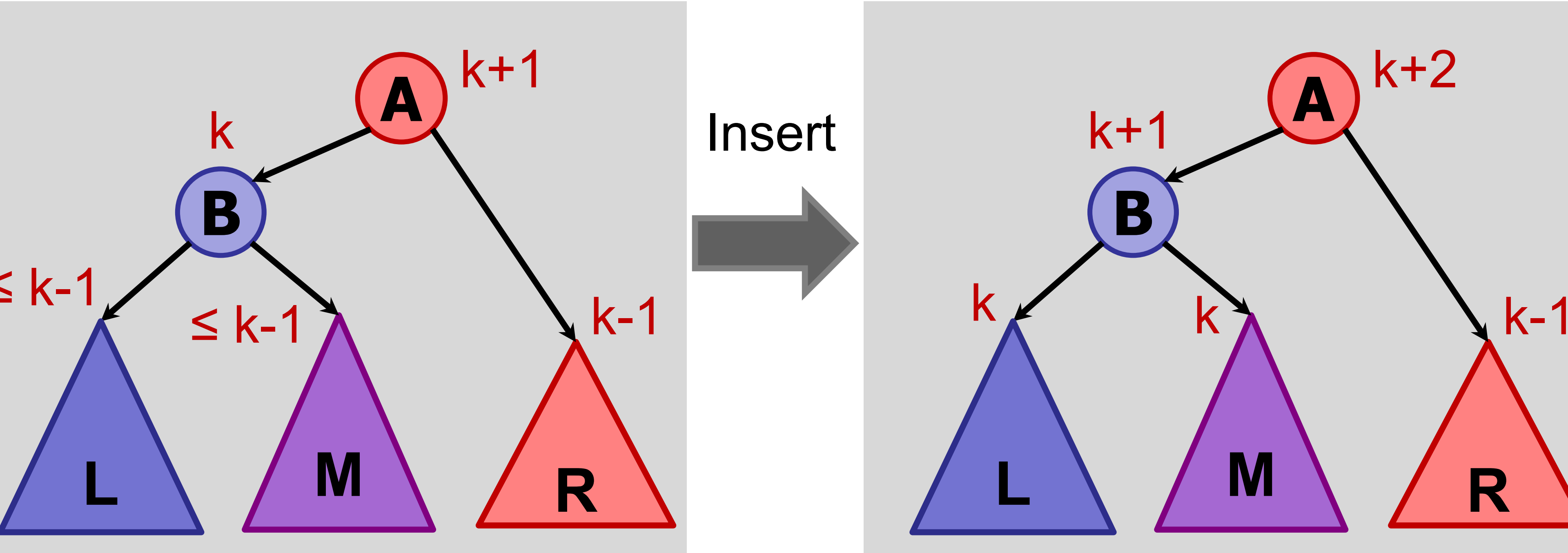


right-rotate:

Case 1: **B** is equi-height : h(**L**) = h(**M**)

h(**R**) = h(**M**) – 1

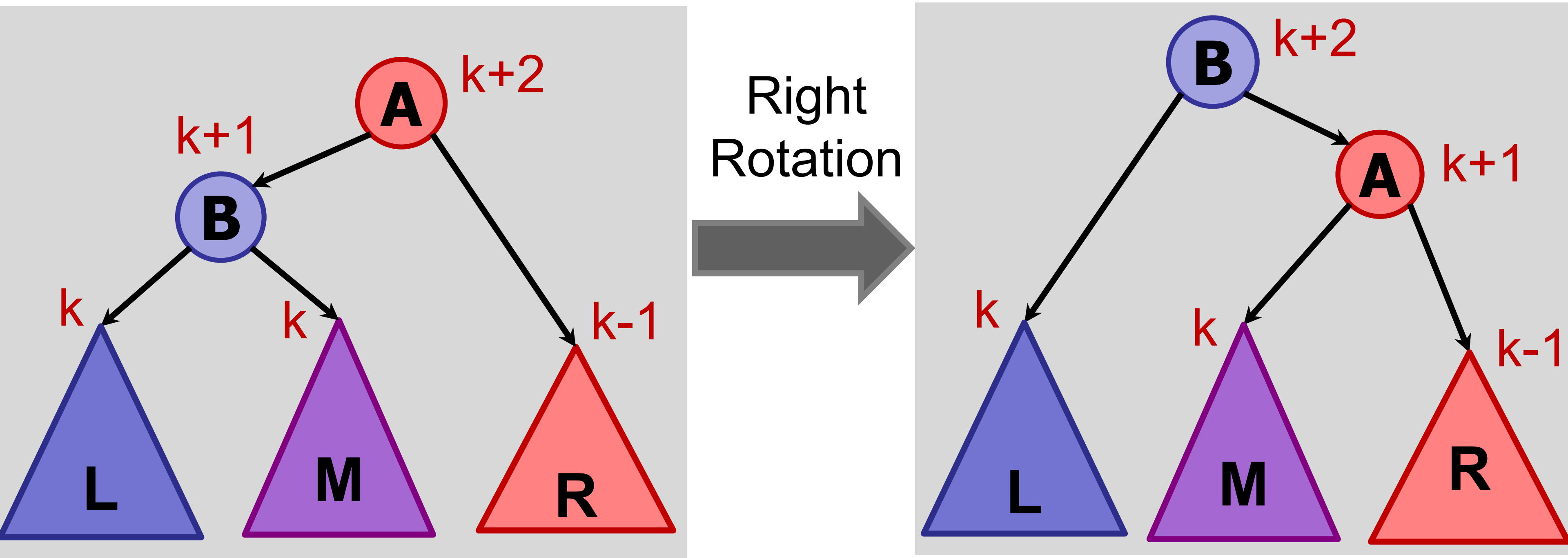
How did we get here?



If the tree was balanced before the insert…

… no possible insert could have increased the height of both L and M.

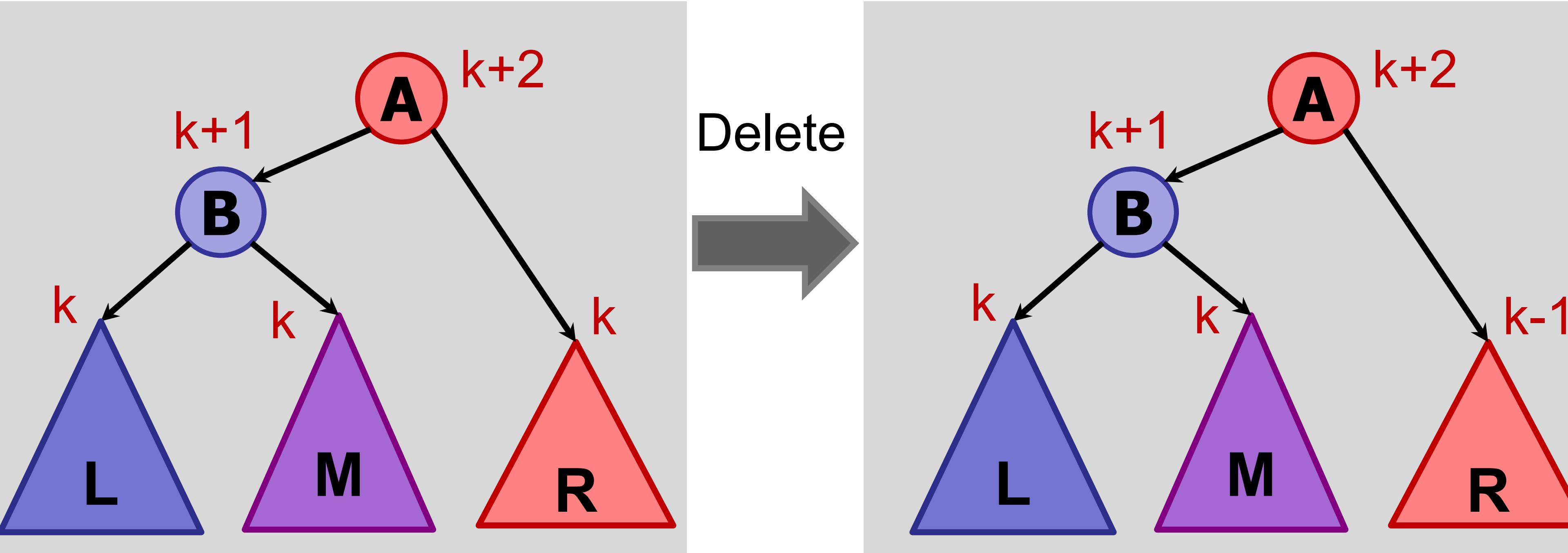
Root height does not decrease!



Why did we cover Case 1?

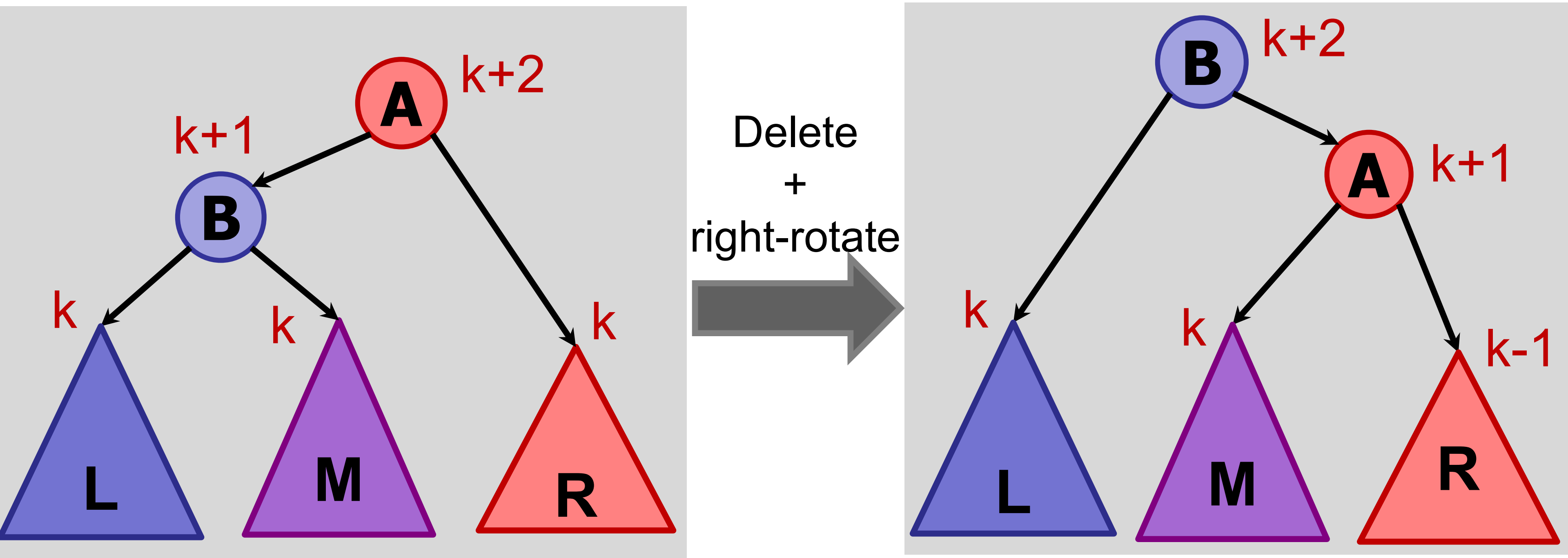
We need this case for deletes…

How did we get here?



Delete in tree R unbalances the tree…

How did we get here?



Delete in tree R unbalances the tree…

And after the rotation to fix it…

Insert in AVL Tree

Summary:

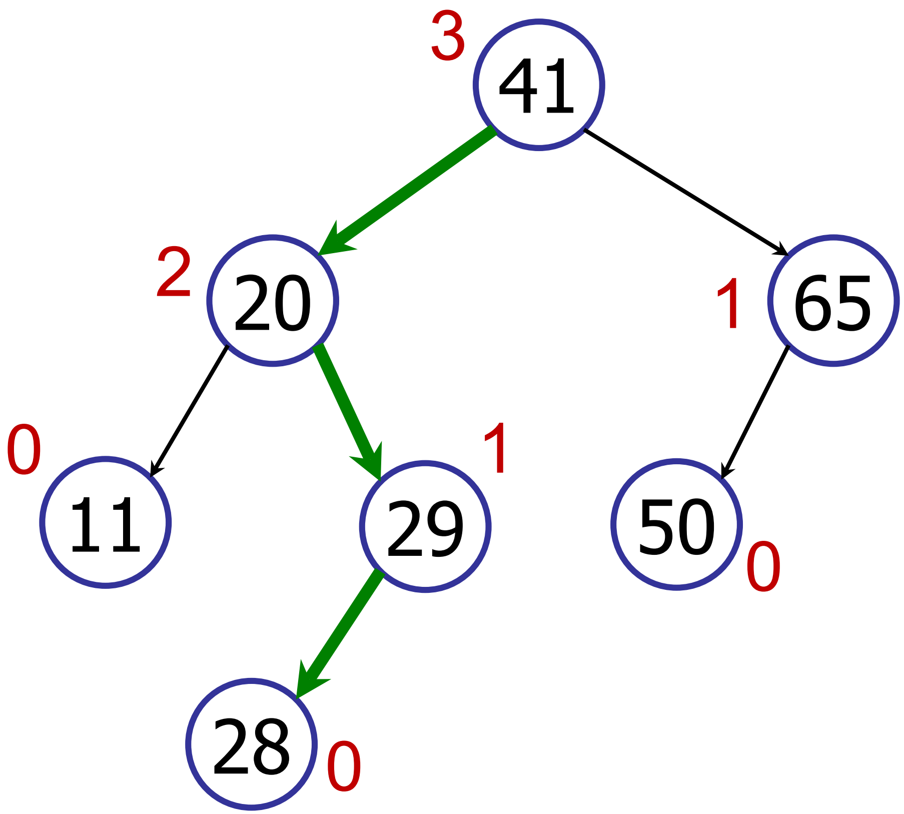
- Insert key in BST.
- Walk up tree:
  - At every step, check for balance.
  - If out-of-balance, use rotations to rebalance and return.

Key observation:

- Only need to fix *lowest* out-of-balance node.
- Only need at most two rotations to fix.

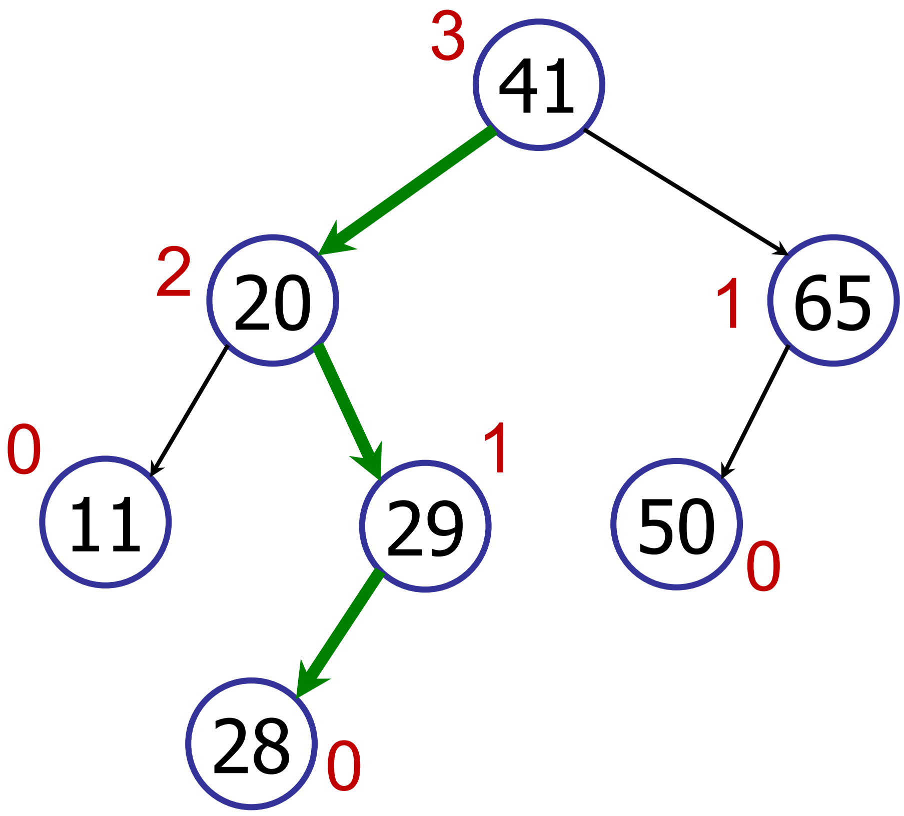
Example

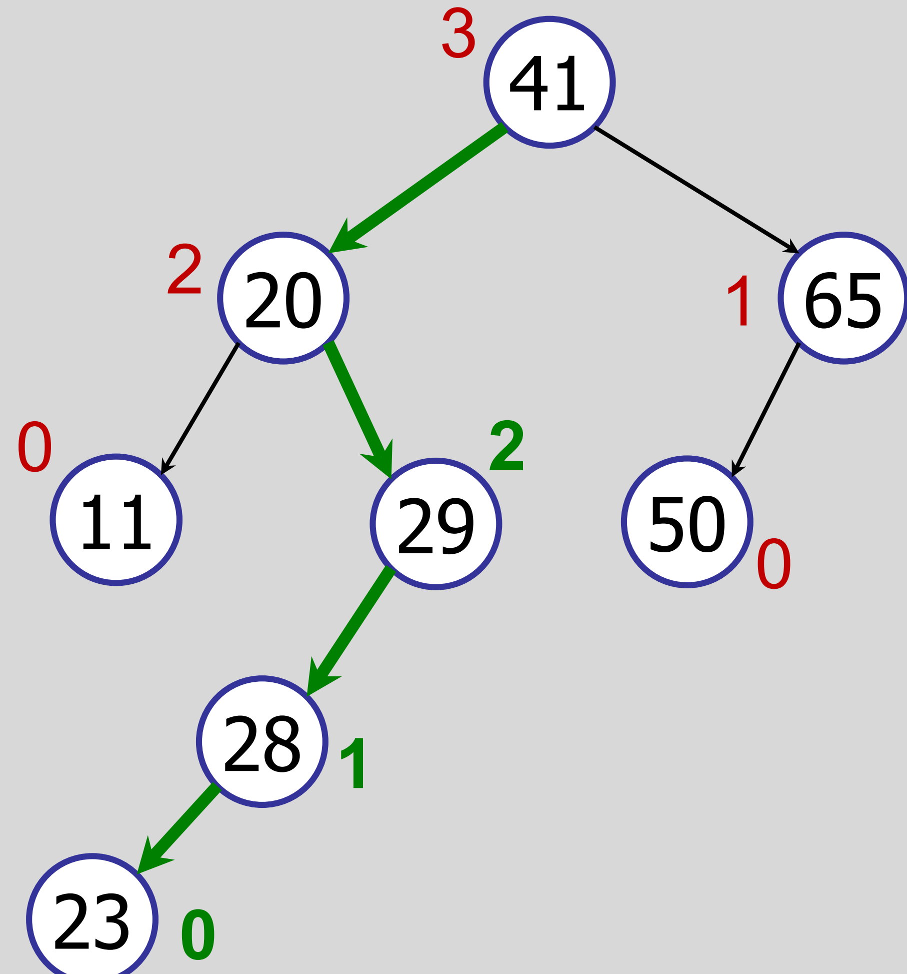
insert(23)



Example

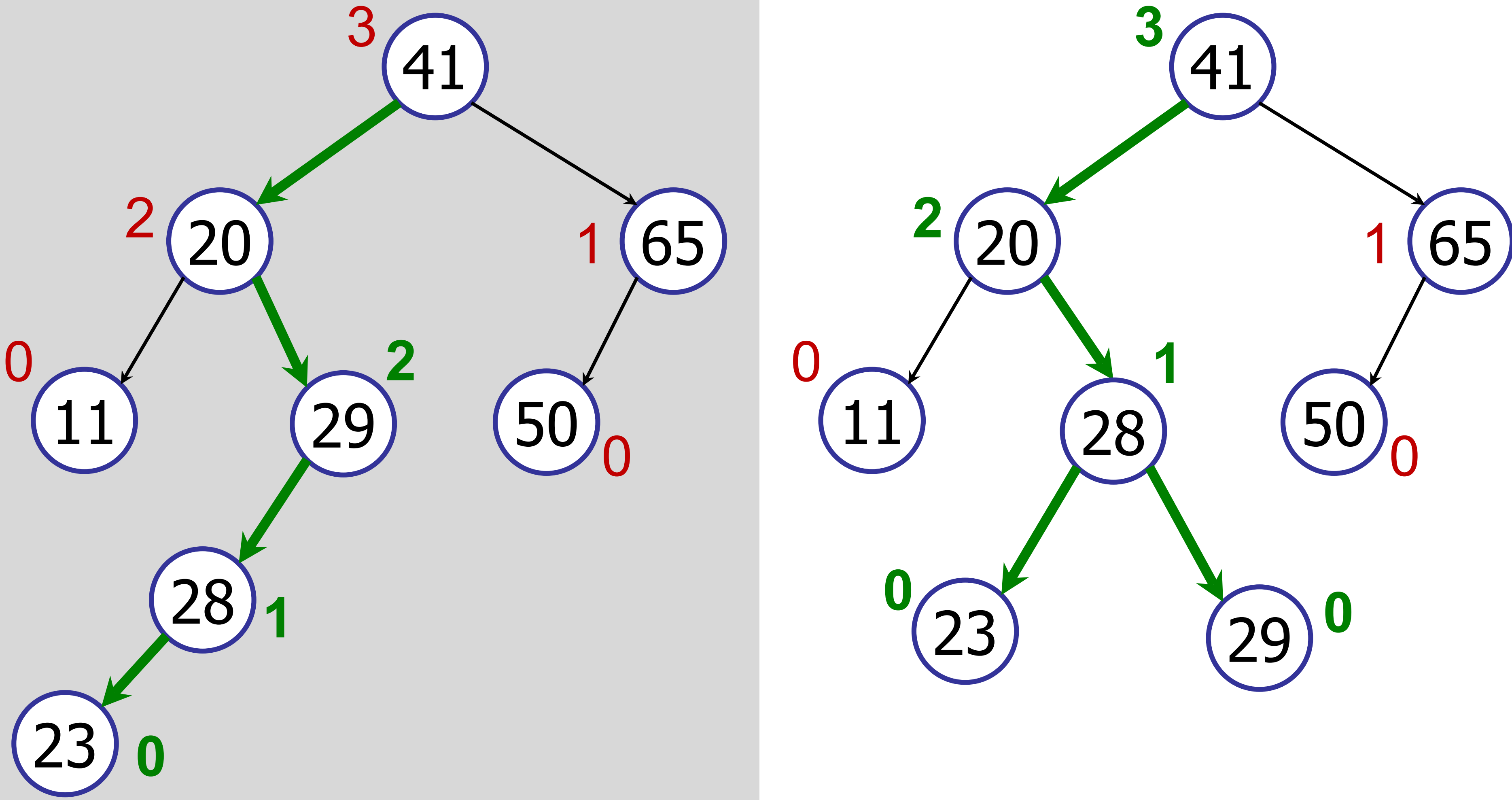
insert(23)





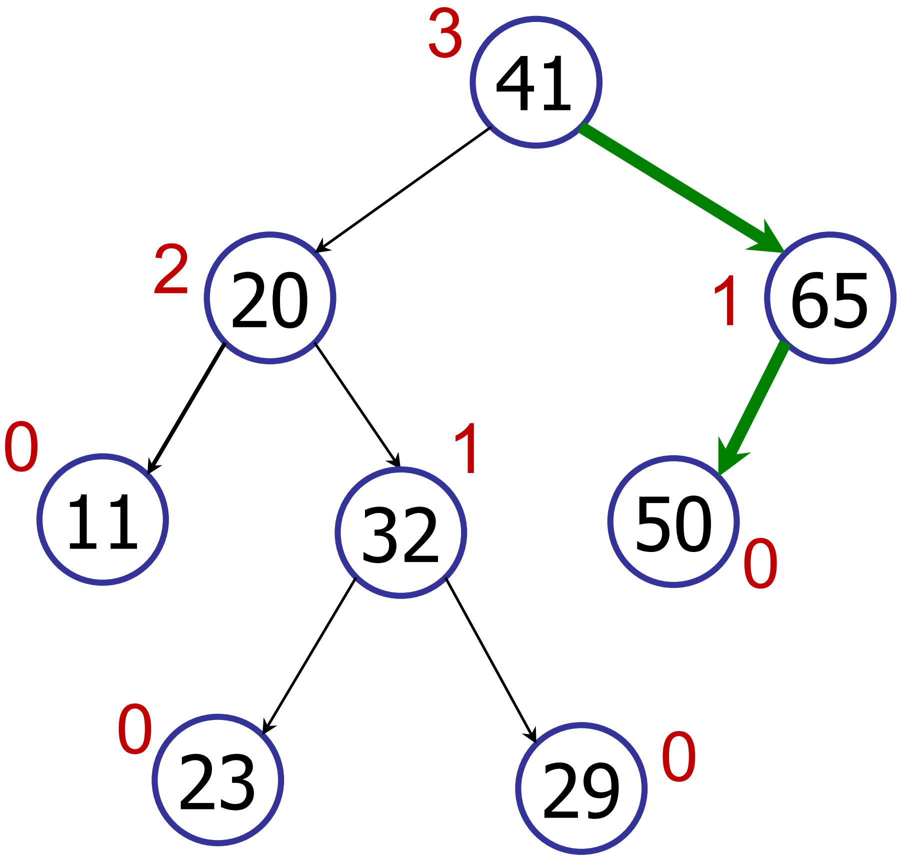
Example

right-rotate(29)



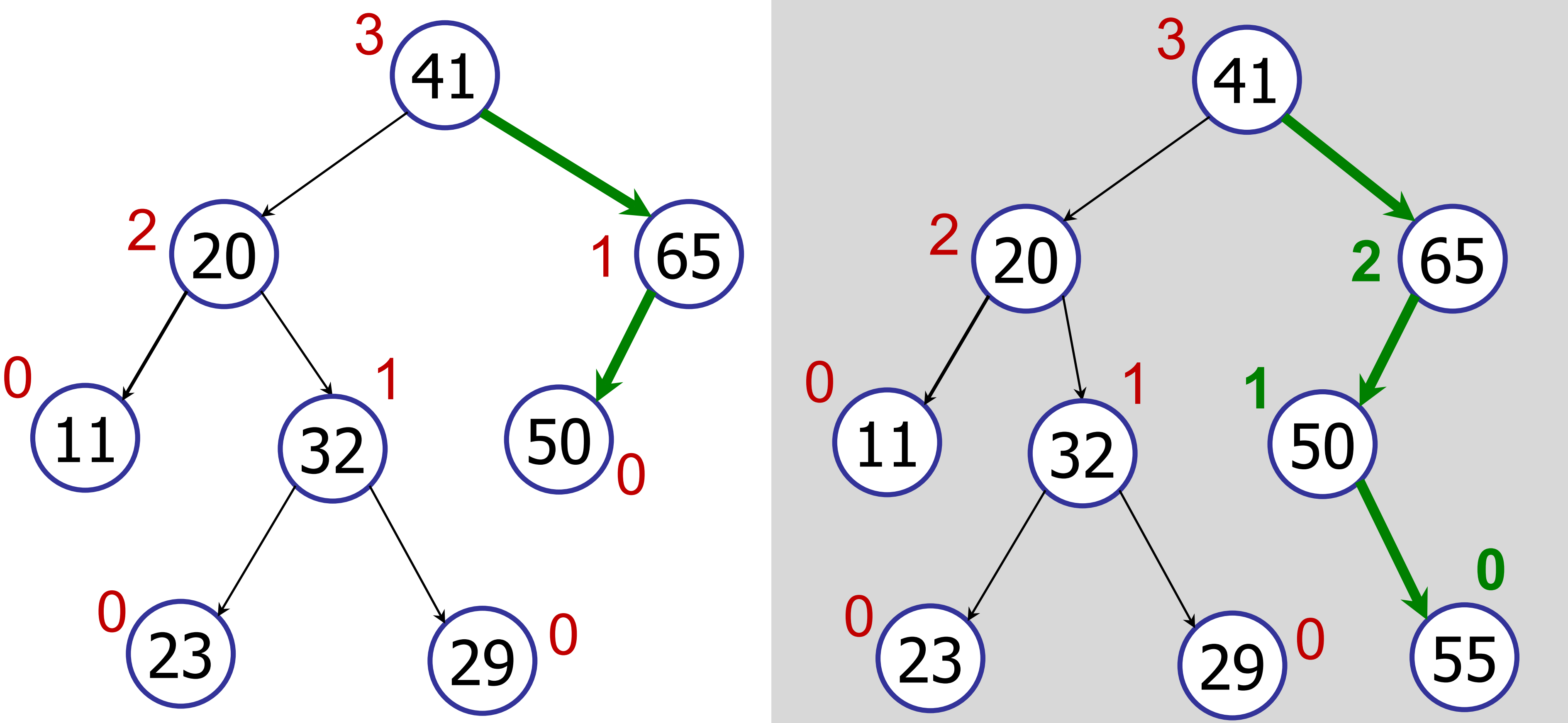
Example

insert(55)



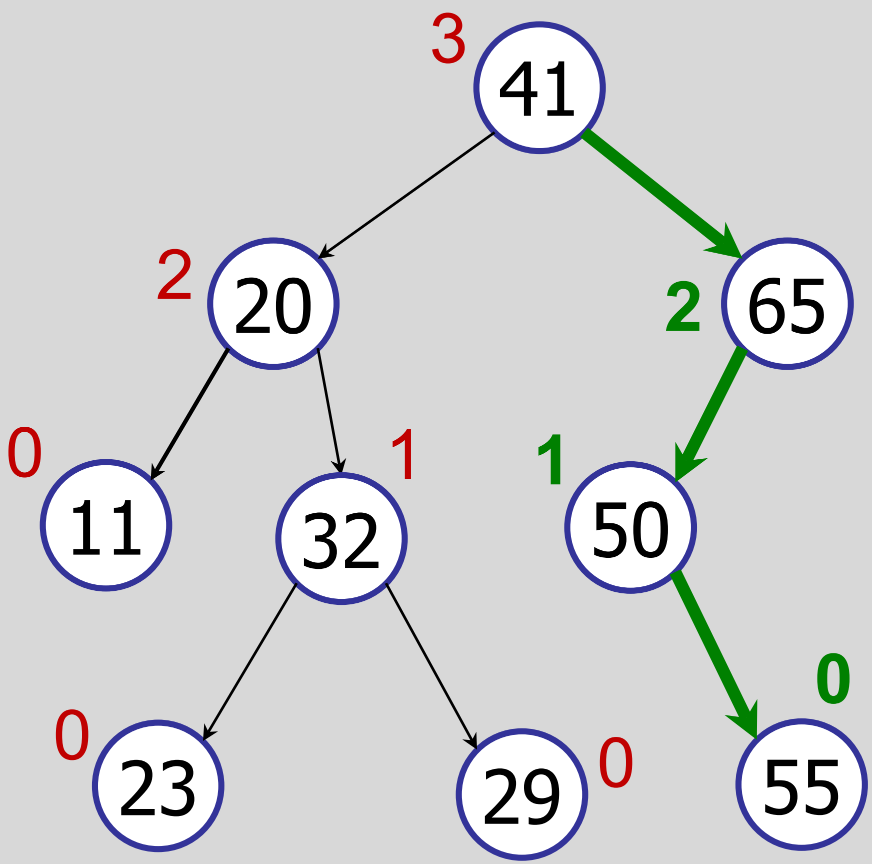
Example

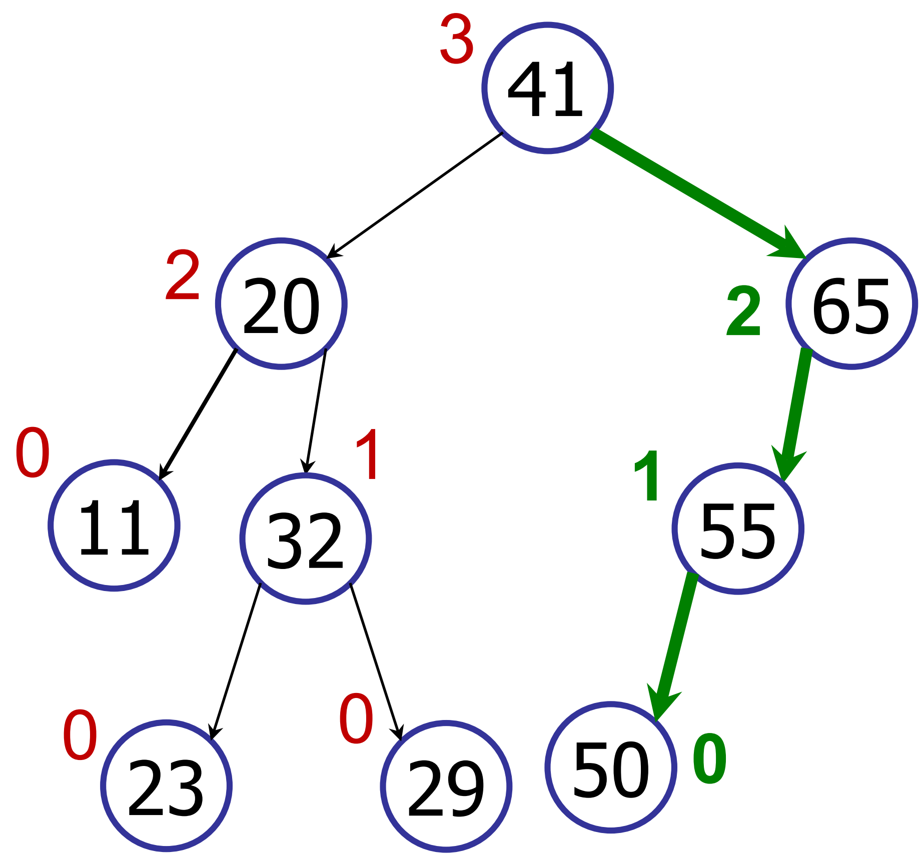
insert(55)



Example

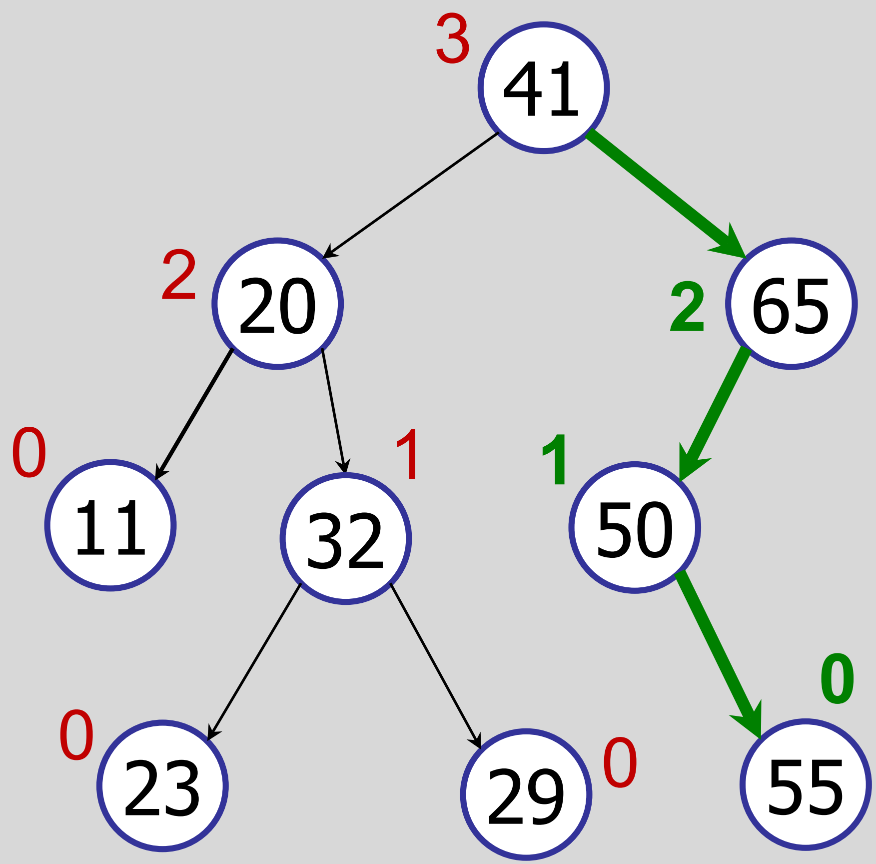
left-rotate(50)

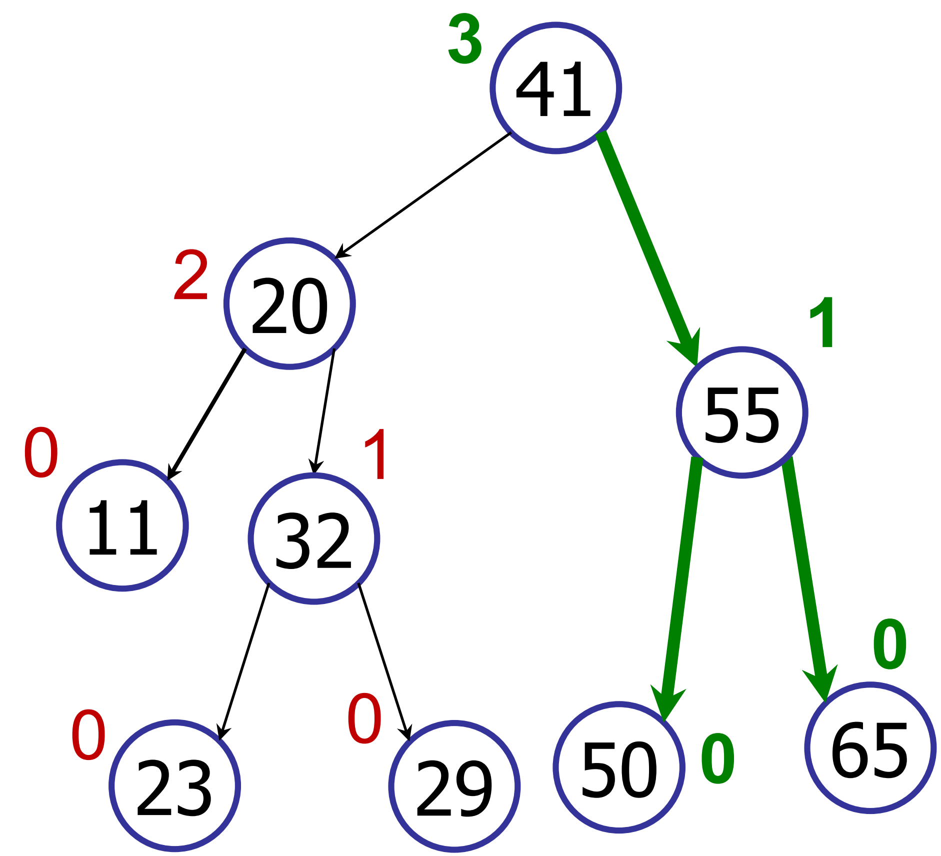




Example

right-rotate(65)



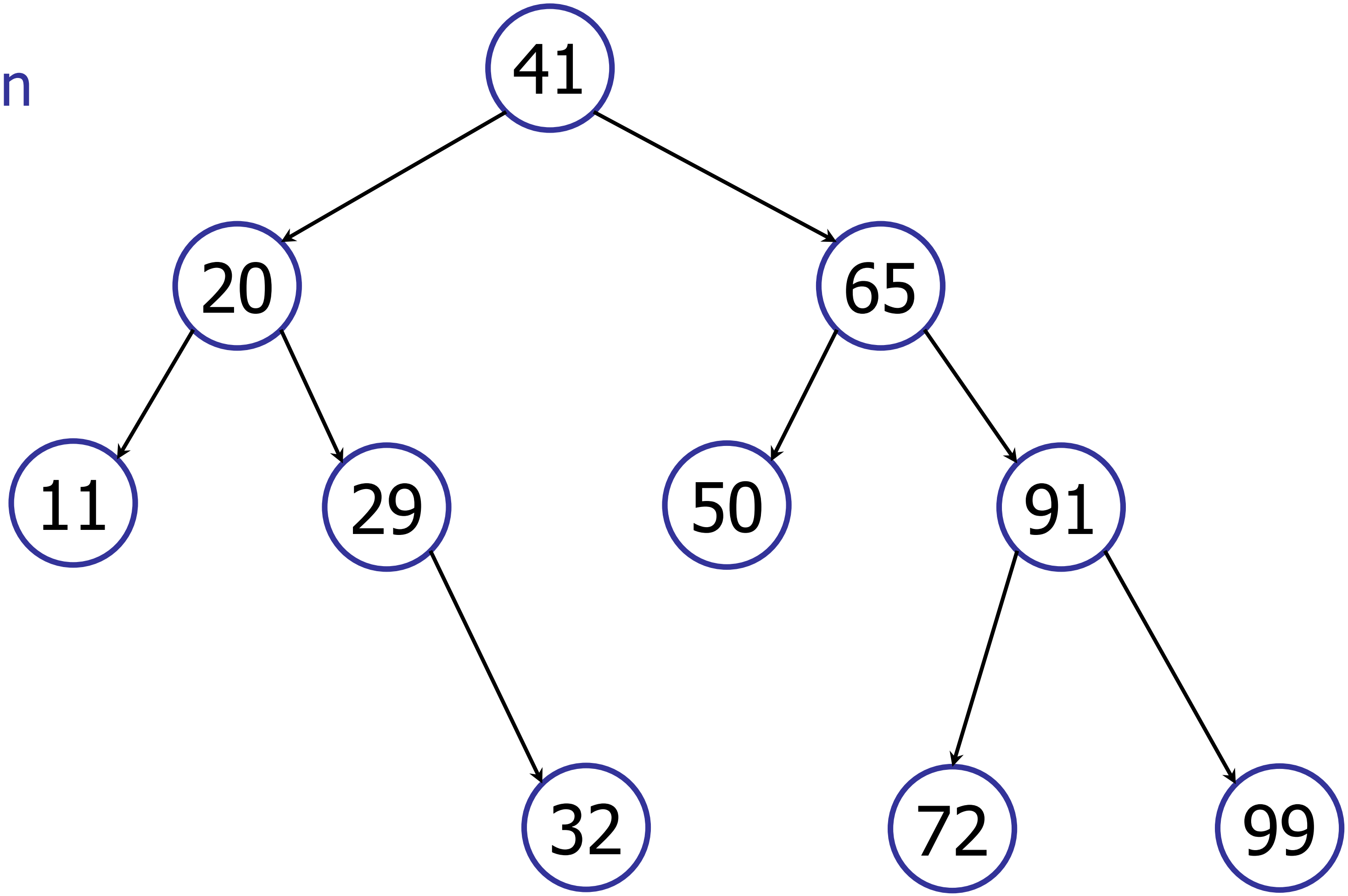


Binary Search Tree

delete(v)

Three cases:

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



Binary Search Tree

delete(v)

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

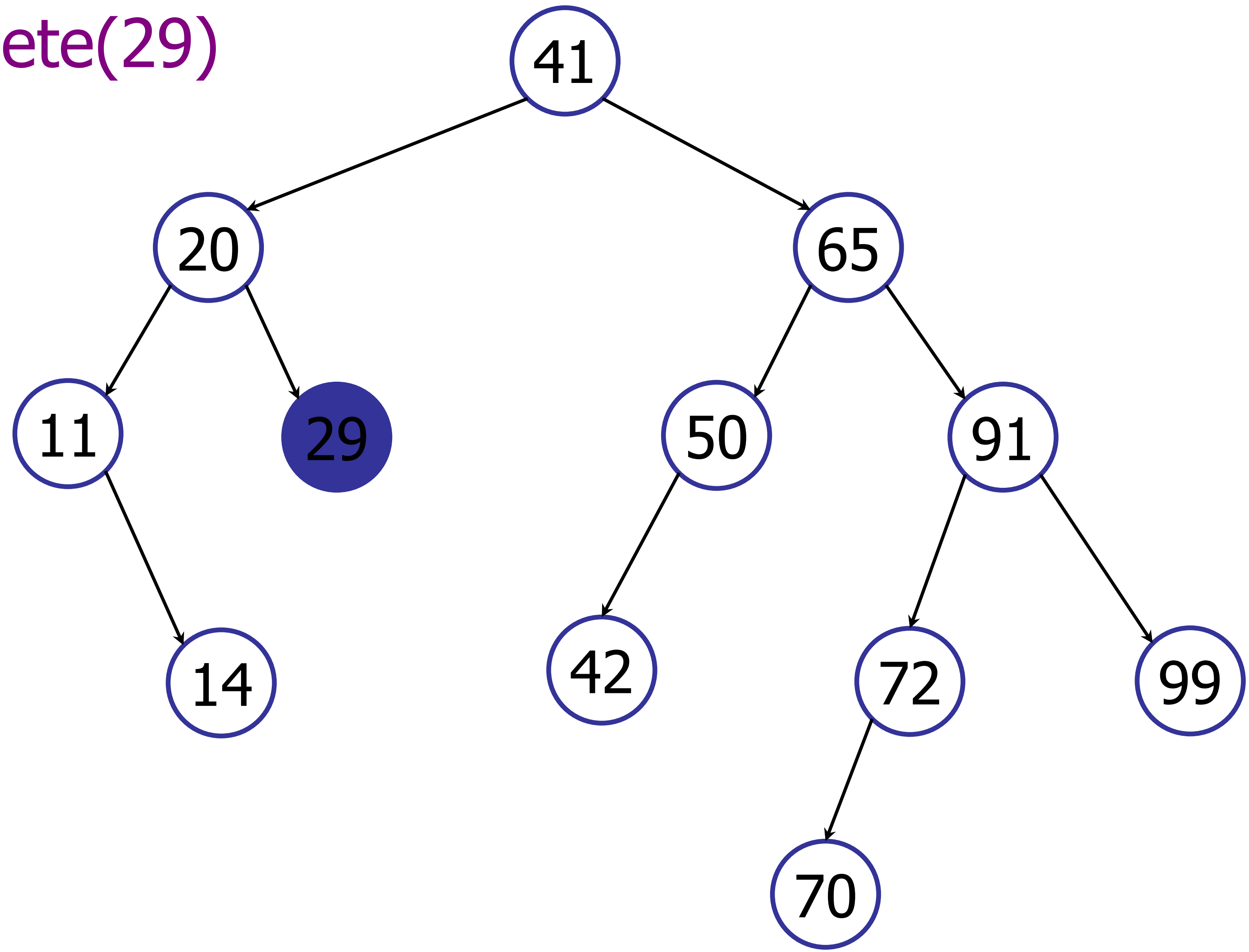
2. Delete node v from binary tree (and reconnect children).

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 O(log(n)) rotations.

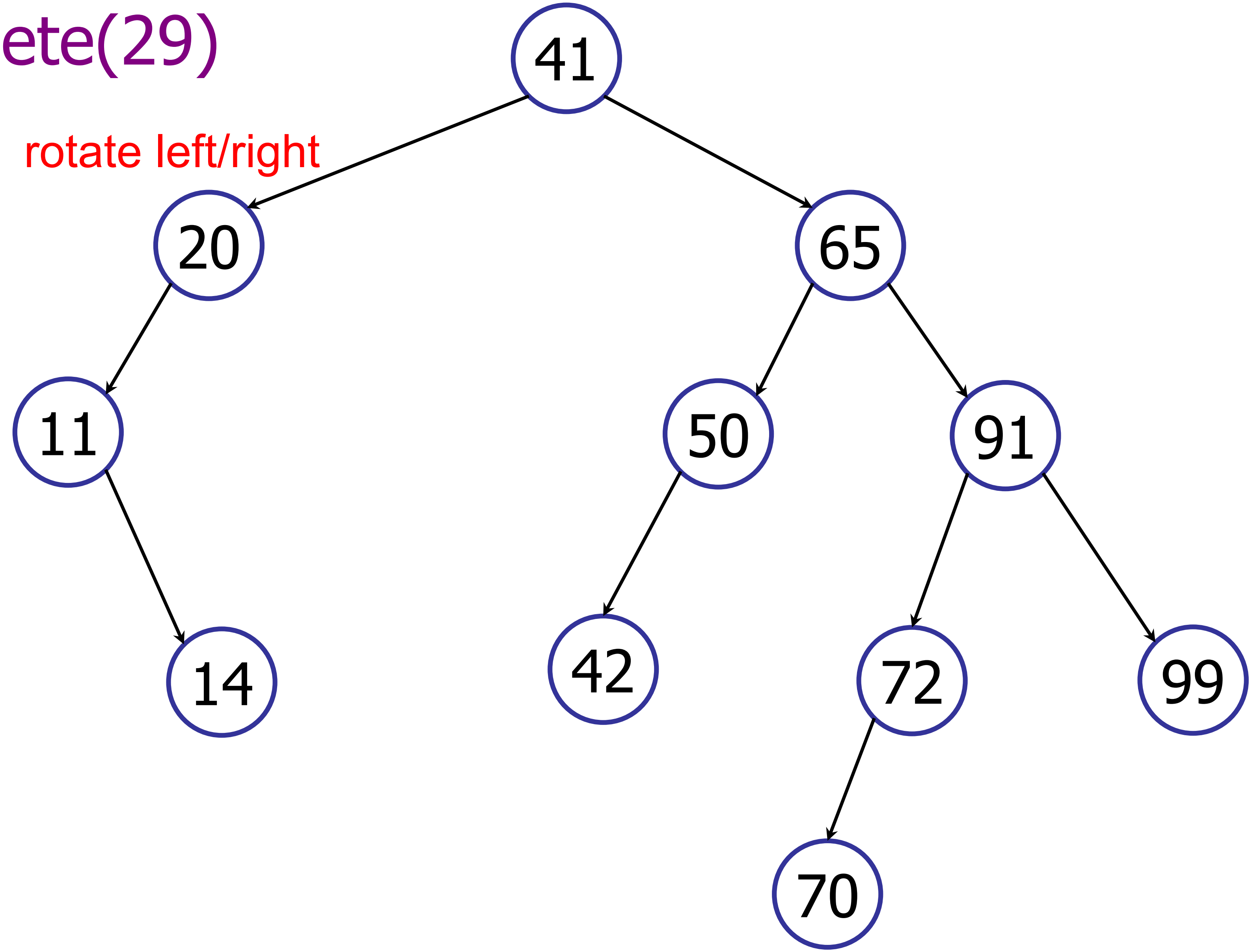
Binary Search Tree

delete(29)



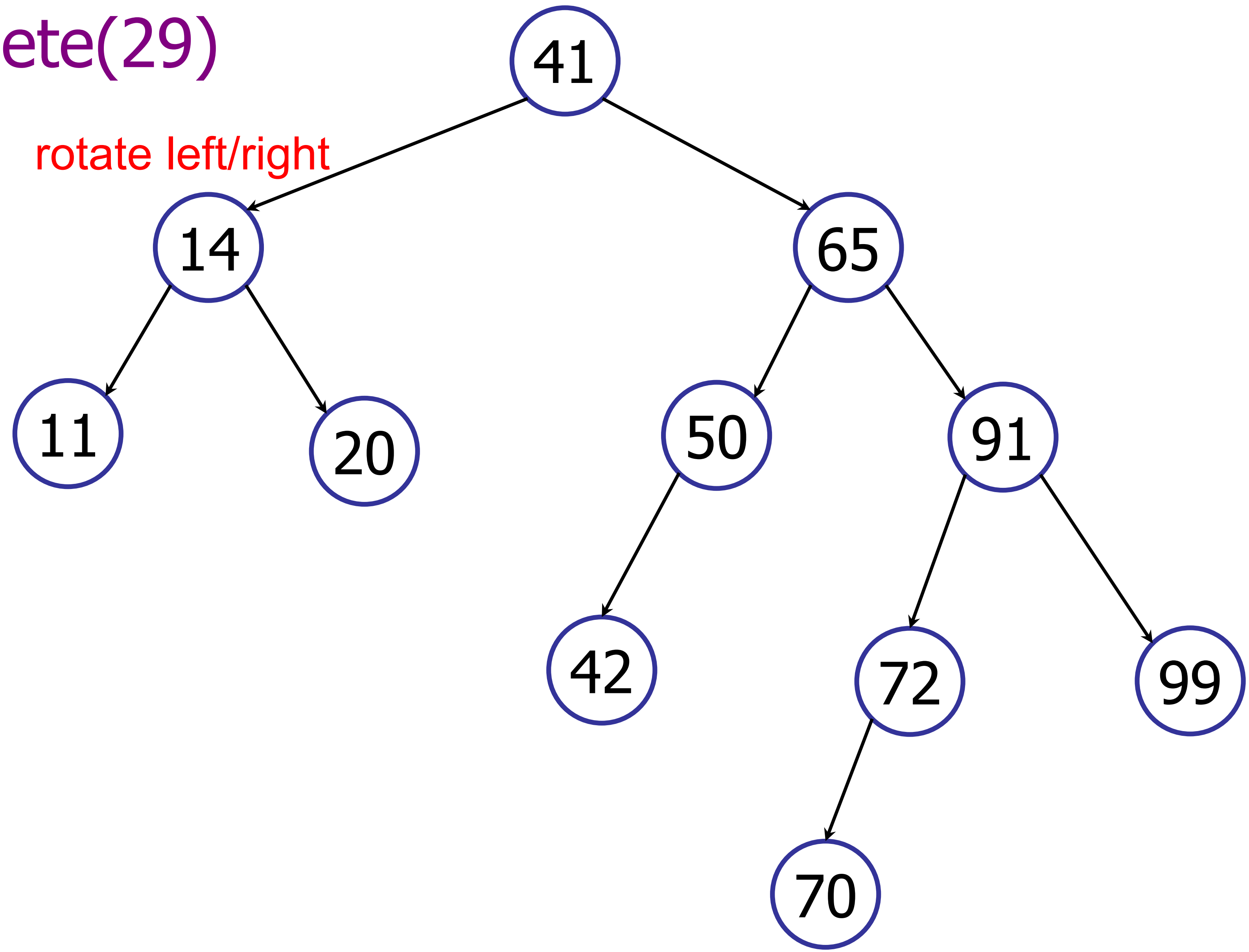
Binary Search Tree

delete(29)



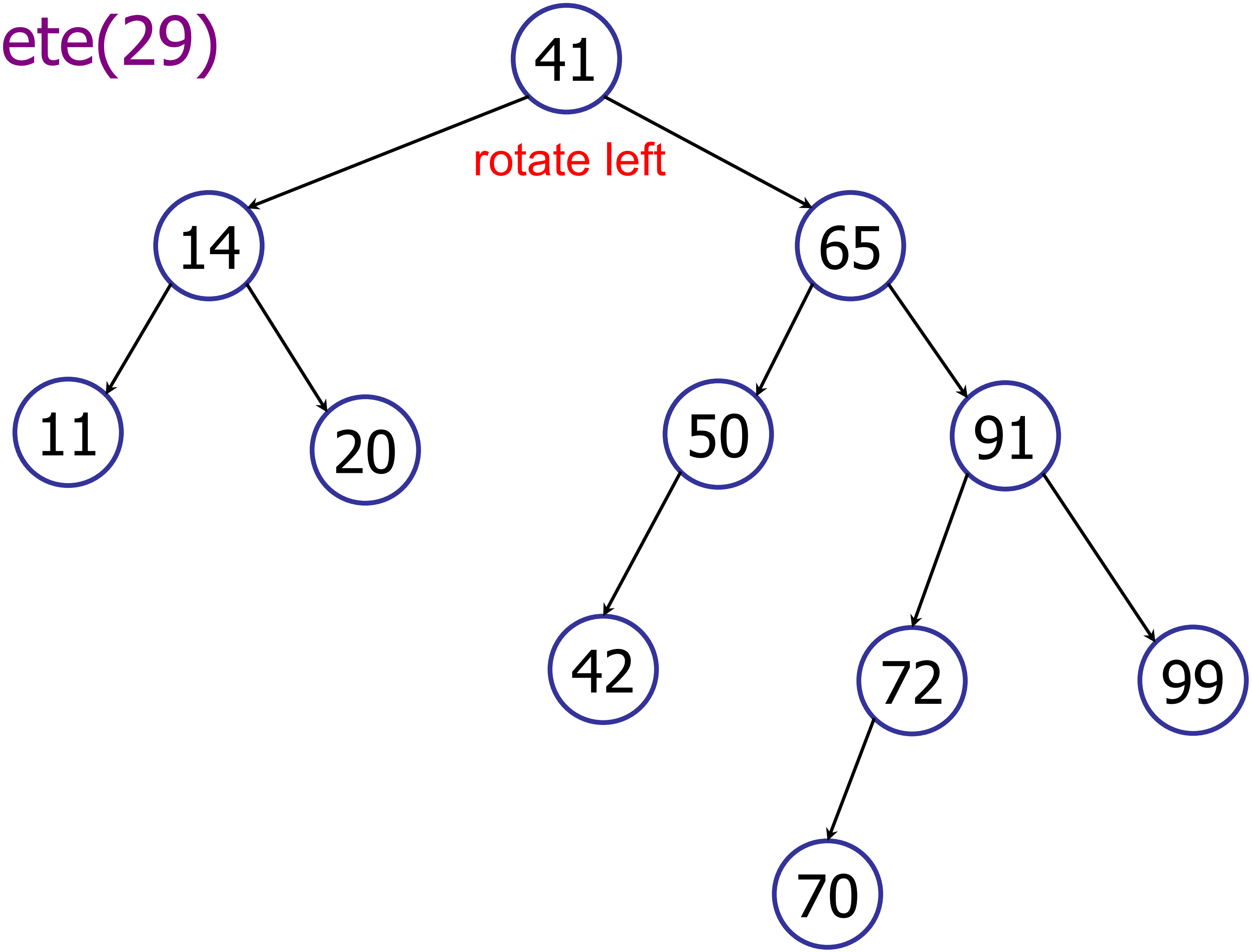
Binary Search Tree

delete(29)



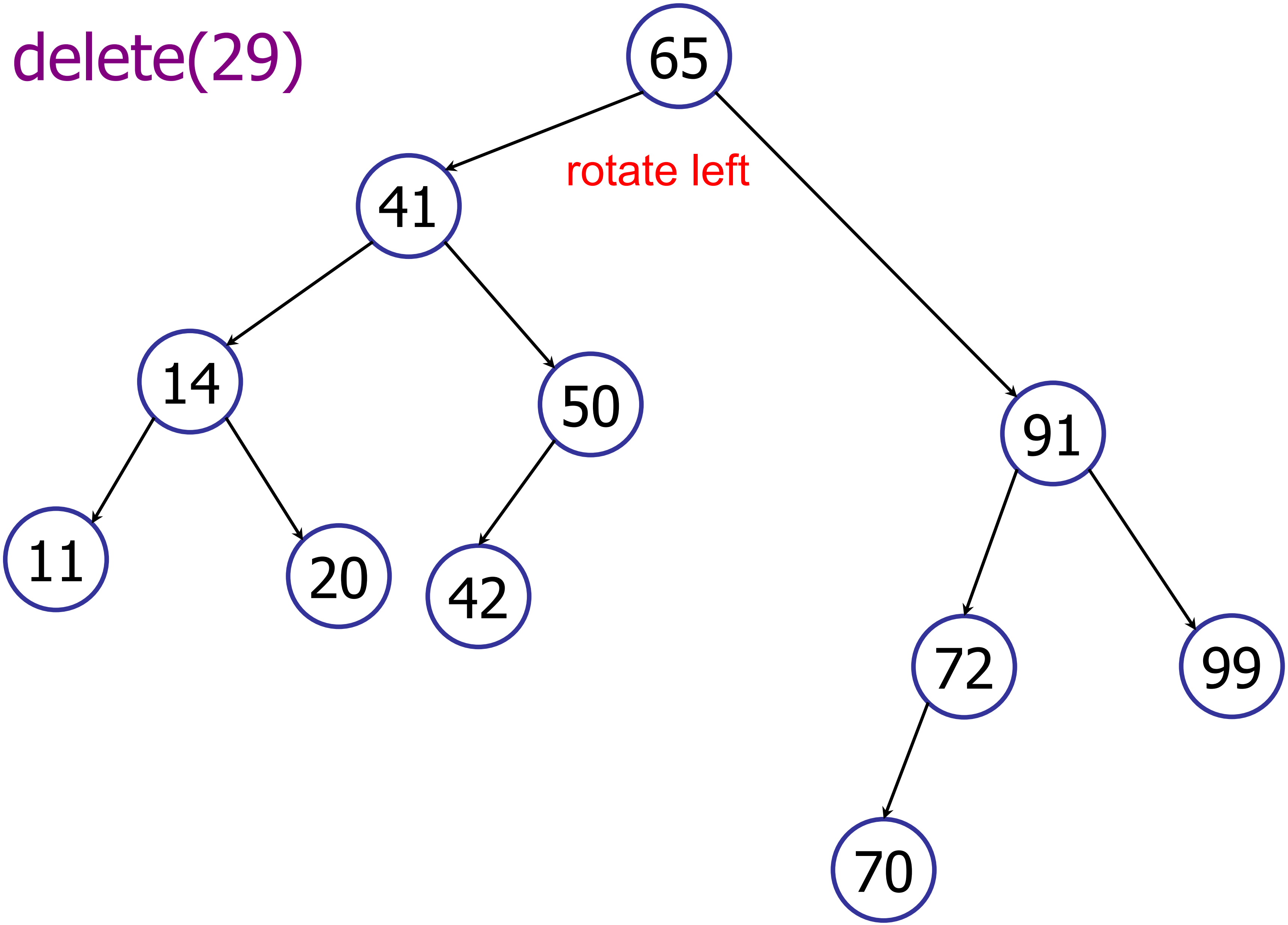
Binary Search Tree

delete(29)



Binary Search Tree

delete(29)



How many rebalances?

Why are two rotations not enough?

- Delete reduced height.
- Rotations (to rebalance) reduce height!

Key observation:

- Rebalancing does not “undo” the change in height caused by deletion.

Delete in AVL Tree

Summary:

- Delete key from BST.
- Walk up tree:
  - At every step, check for balance.
  - If out-of-balance, use rotations to rebalance.
  - Continue to root.

Key observation:

- It is *not* sufficient to only fix lowest out-of-balance node in tree.

Every insertion requires 1 or 2 rotations?

1. Yes
2. No
3. I don't know

A tree is **balanced** if every node's children differ in height be at most 1?

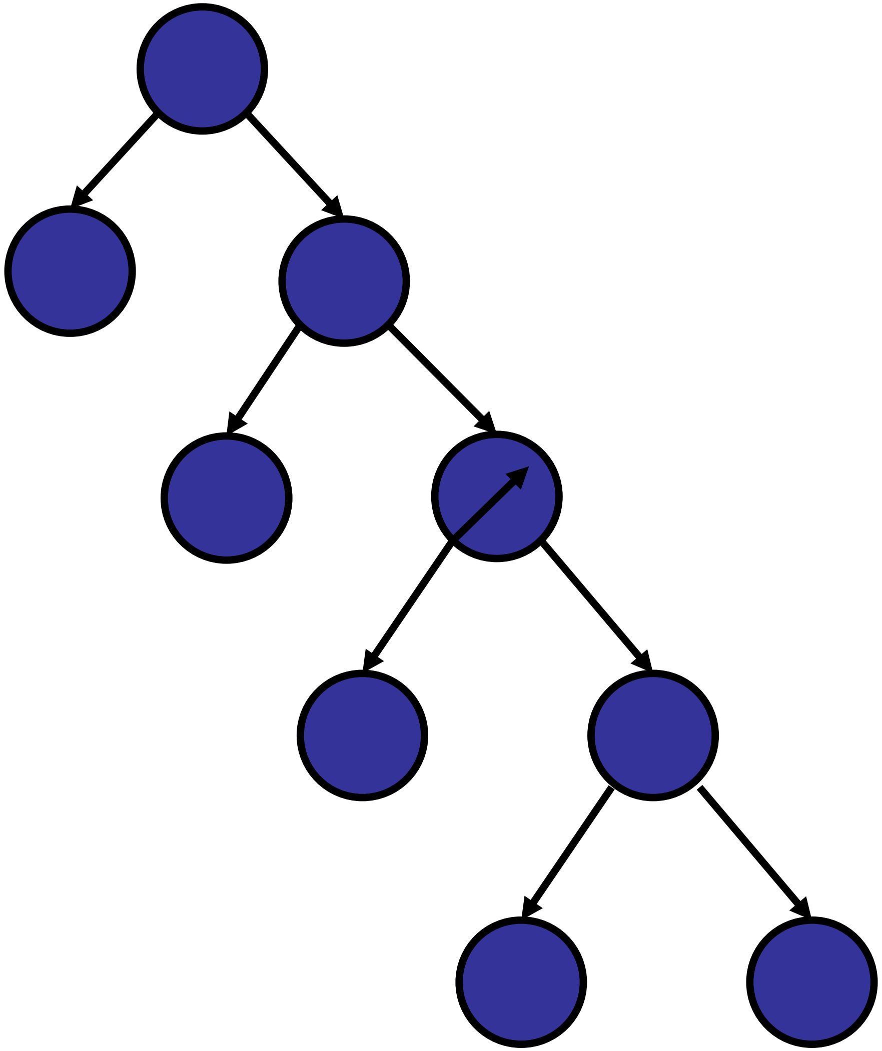
1. Yes
2. No
3. I don't know

A tree is **balanced** if every node either has two children or zero children?

1. Yes
2. No
3. I don't know

A tree is balanced if every node either has two children or zero children?

1. Yes
2. No
3. I don’t know



A tree is height-balanced if:
For every node, the number of keys in its heavier sub-tree is at most twice the number of keys in its lighter sub-tree.

1. Yes
2. No, but it is balanced.
3. No.
4. I don't know

Using rotations, you can create every possible "tree shape."

1. True
2. False
3. I don't know

AVL Trees: Other potential modifications

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: Other potential modifications

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

- Only store difference in height from parent.

Next Week

Even more trees! (Monday)

- Other augmentations
- Examples of other forms of trees

Hashing! (Wednesday)

- Introduction to hashing