CS 240 – Data Structures and Data Management

Module 4: Dictionaries

Armin Jamshidpey, Éric Schost
Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Spring 2026

version 2026-06-15 16:57

Outline

4. Dictionaries and Balanced Search Trees
   - ADT Dictionary
   - Binary Search Trees
   - AVL Trees
   - Restructuring a BST: Rotations
   - Insertion in AVL Trees
   - Deletion in AVL Trees

Outline

4 Dictionaries and Balanced Search Trees
- ADT Dictionary
- Binary Search Trees
- AVL Trees
- Restructuring a BST: Rotations
- Insertion in AVL Trees
- Deletion in AVL Trees

Review: ADT Dictionary

Dictionary: A collection of items, each of which contains

- a *key*
- some *data* (the "value")

and is called a *key-value pair* (KVP). Keys can be compared and are assumed to be unique.

Operations:

- *search*(k)
- *insert*(k, v)
- *delete*(k)
- optional: *successor*, *merge*, *is-empty*, *size*, *etc.*

Examples: symbol table, license plate database

Review: Elementary realizations

Common assumptions:

- Dictionary has n KVPs
- Each KVP uses constant space
  (if not, the "value" could be a pointer)
- Keys can be compared in constant time

We commonly make one more assumption (to keep pseudo-code simple):

- Dictionary is non-empty both before and after operation.

  (In a real-life implementation you would have to treat these special cases.)

Exercise: easy realizations

	search	*insert*	*delete*
unsorted list/array	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$
sorted array	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
binary search tree	$\Theta(\mathit{height})$	$\Theta(\mathit{height})$	$\Theta(\mathit{height})$

Sorted array search uses *binary-search*, we'll review that in module 6

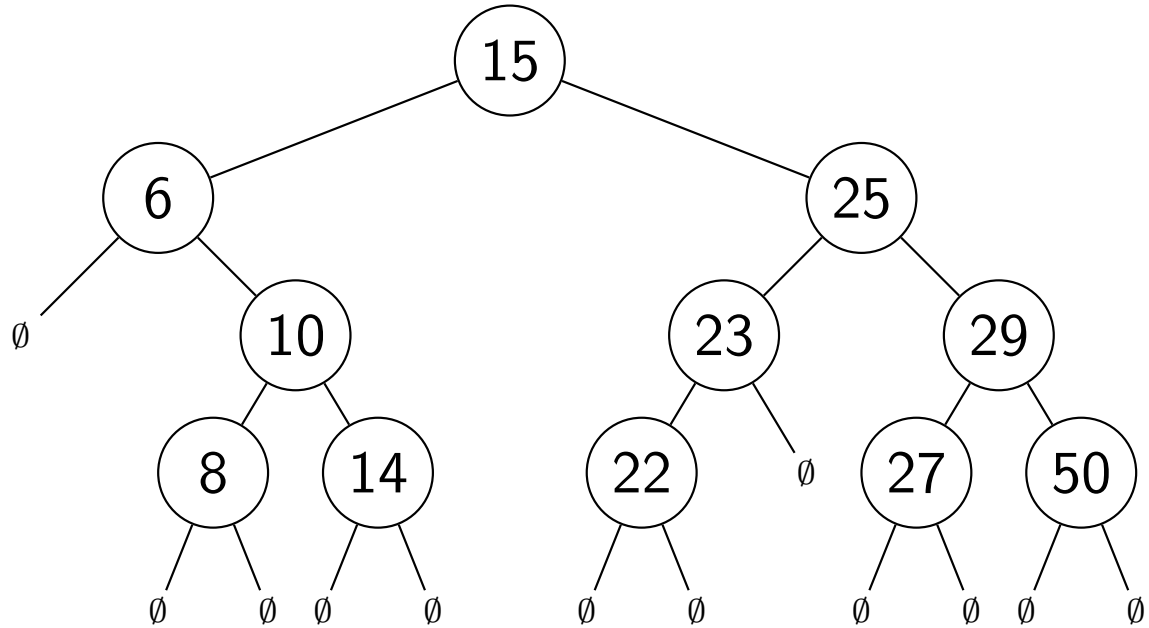
Outline

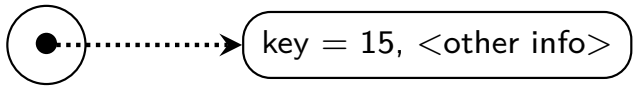
4. Dictionaries and Balanced Search Trees
   - ADT Dictionary
   - Binary Search Trees
   - AVL Trees
   - Restructuring a BST: Rotations
   - Insertion in AVL Trees
   - Deletion in AVL Trees

Review: Binary search trees

Structure Binary tree: all nodes have two (possibly empty) subtrees
Every node stores a KVP
Empty subtrees usually not shown

Ordering Every key k in $T.left$ is less than the root key.
Every key k in $T.right$ is greater than the root key.

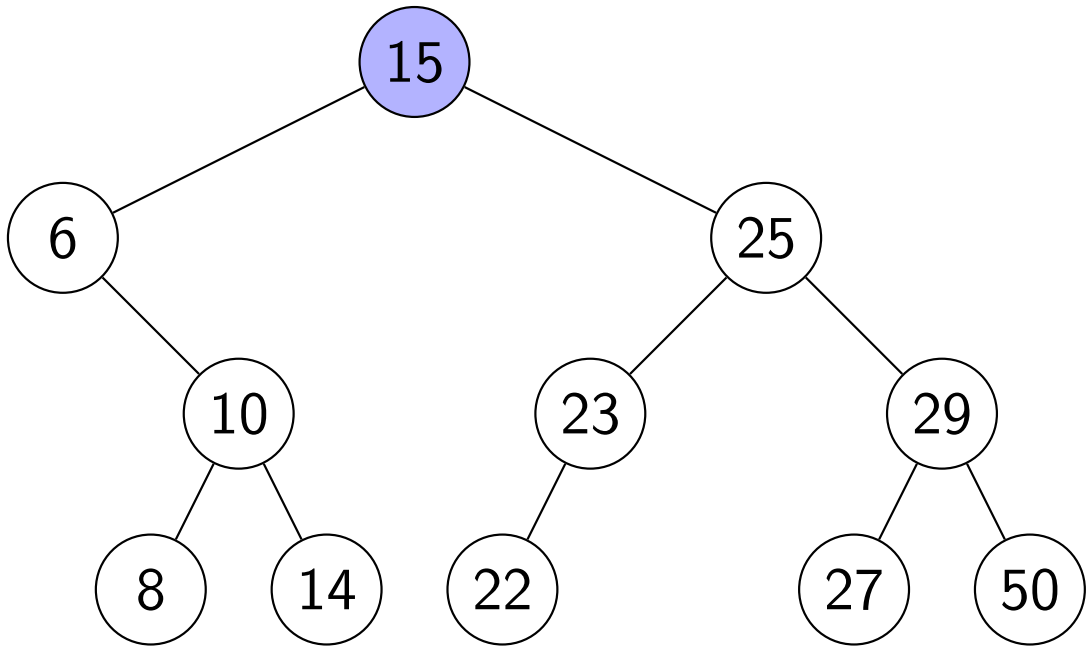


(In our examples we only show the keys, and we show them directly in the node. A more accurate picture would be )

Review: BST as realization of ADT Dictionary

BST::search(k) Start at root, compare k to current node's key.
Stop if found or subtree is empty, else recurse at subtree.

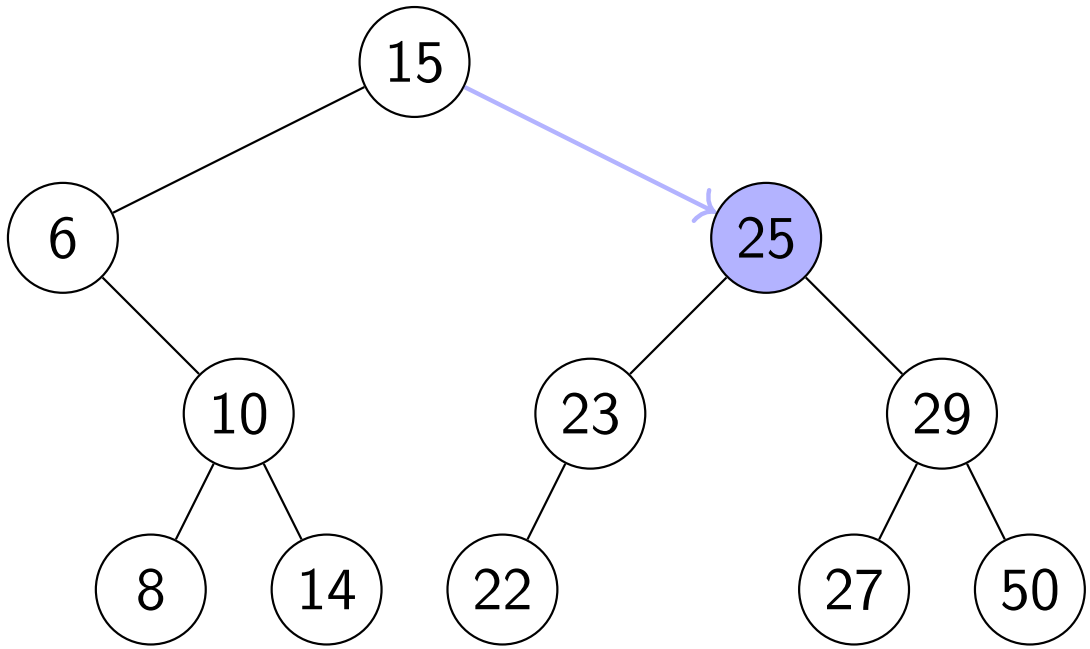
Example: *BST::search*(24)



Review: BST as realization of ADT Dictionary

BST::search(k) Start at root, compare k to current node's key.
Stop if found or subtree is empty, else recurse at subtree.

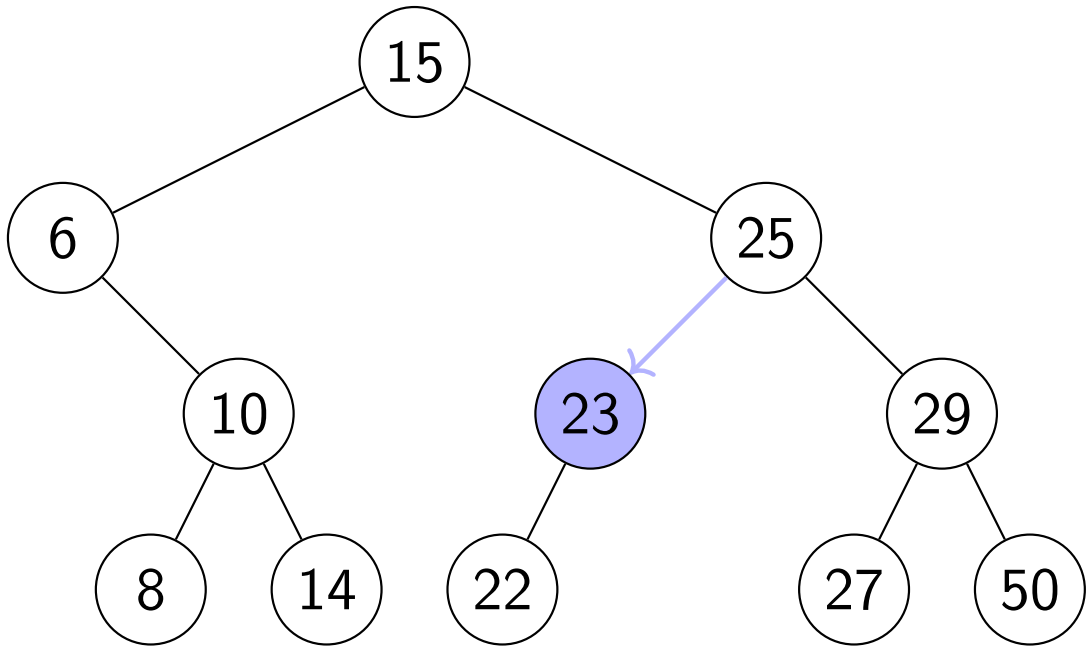
Example: *BST::search*(24)



Review: BST as realization of ADT Dictionary

BST::search(k) Start at root, compare k to current node's key.
Stop if found or subtree is empty, else recurse at subtree.

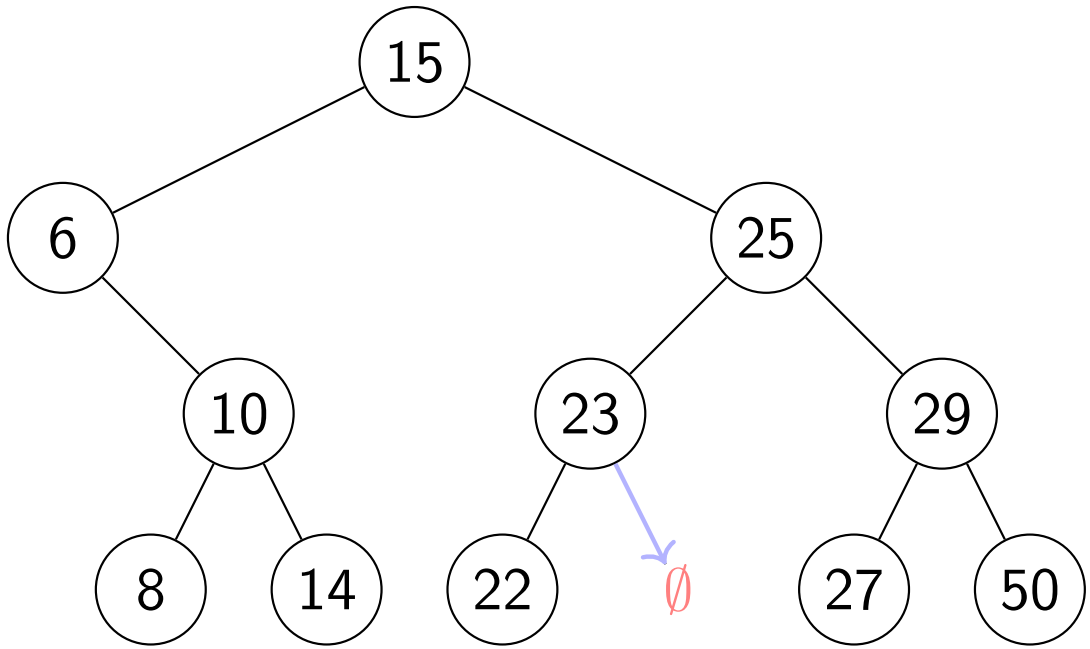
Example: *BST::search*(24)



Review: BST as realization of ADT Dictionary

BST::search(k) Start at root, compare k to current node's key.
Stop if found or subtree is empty, else recurse at subtree.

Example: *BST::search*(24)

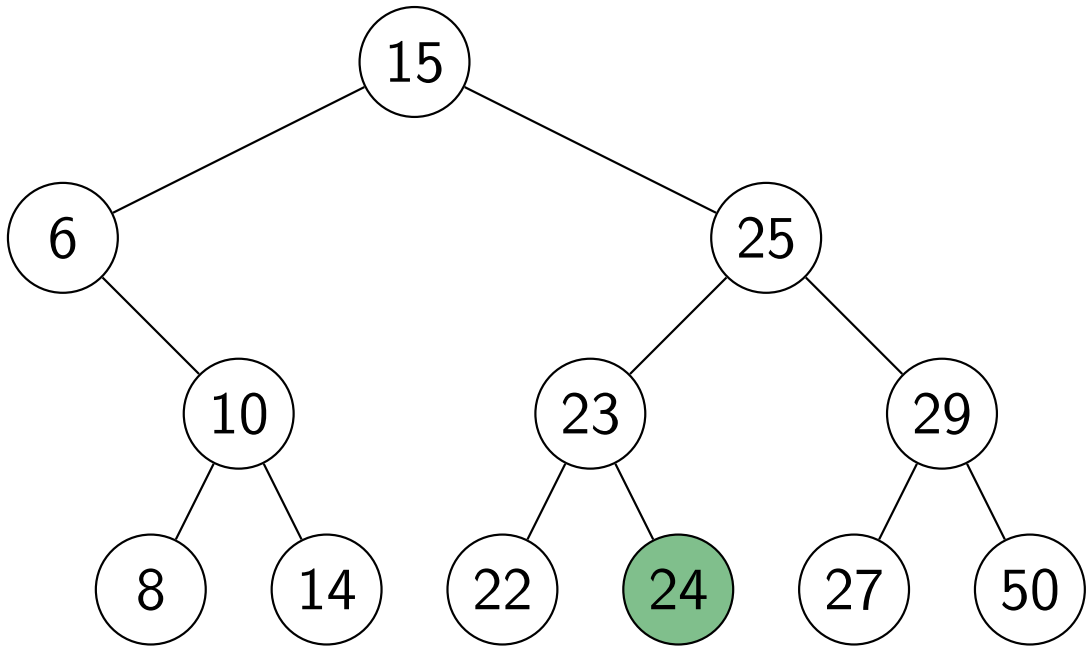


Review: BST as realization of ADT Dictionary

BST::search(k) Start at root, compare k to current node's key.
Stop if found or subtree is empty, else recurse at subtree.

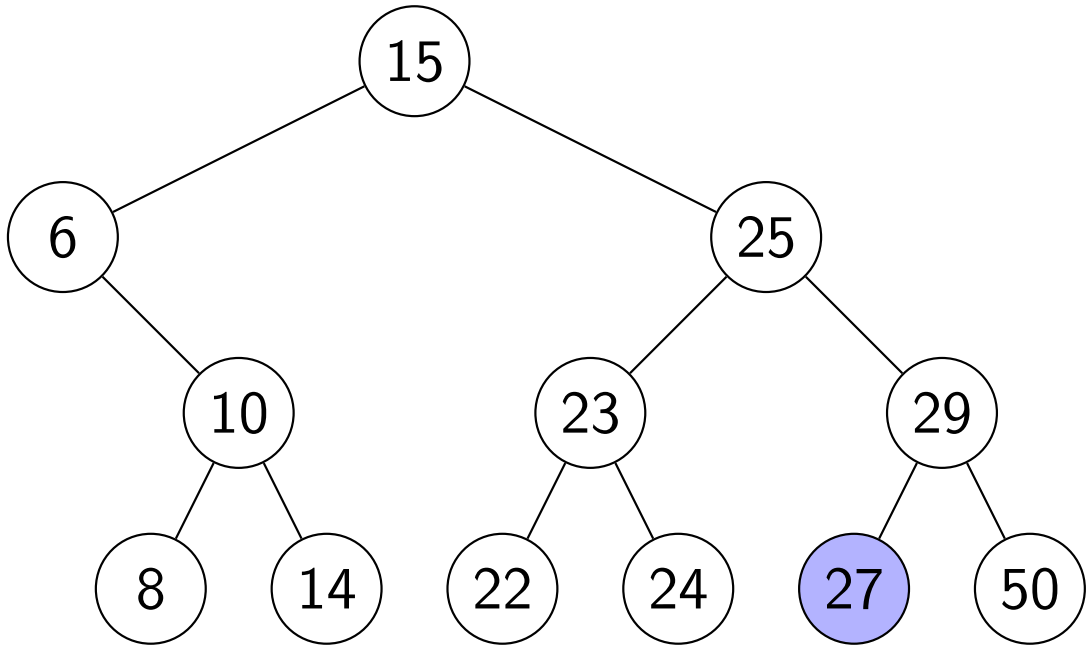
BST::insert(k, v) Search for k, then insert (k, v) as new node

Example: *BST::insert*$(24, v)$



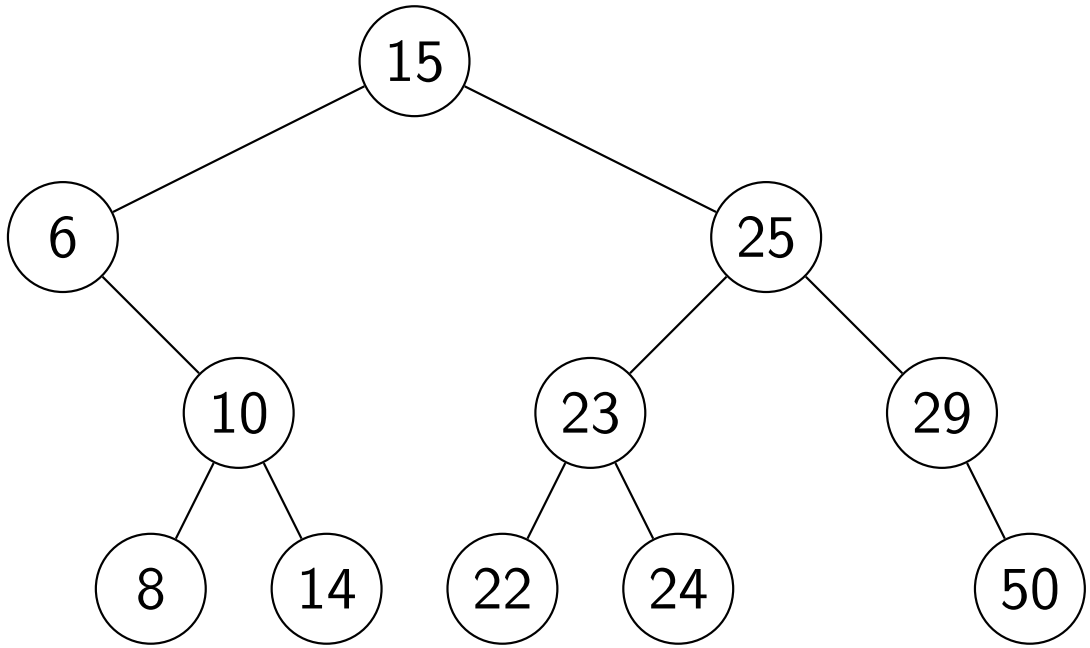
Deletion in a BST

- First search for the node x that contains the key.
- If x is a **leaf** (both subtrees are empty), delete it.



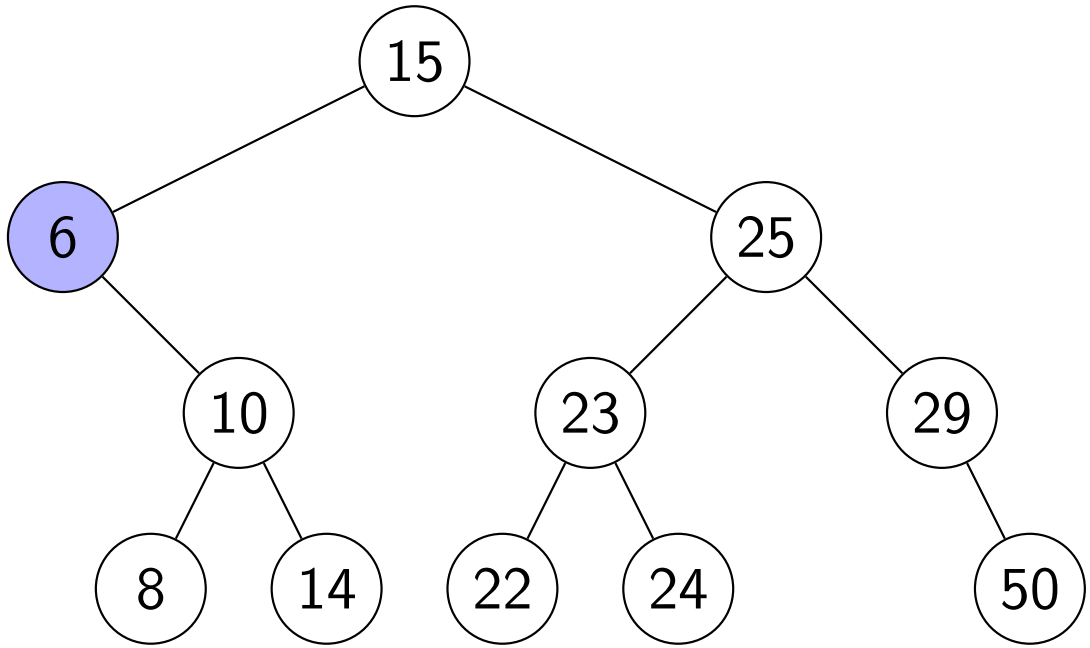
Deletion in a BST

- First search for the node x that contains the key.
- If x is a **leaf** (both subtrees are empty), delete it.



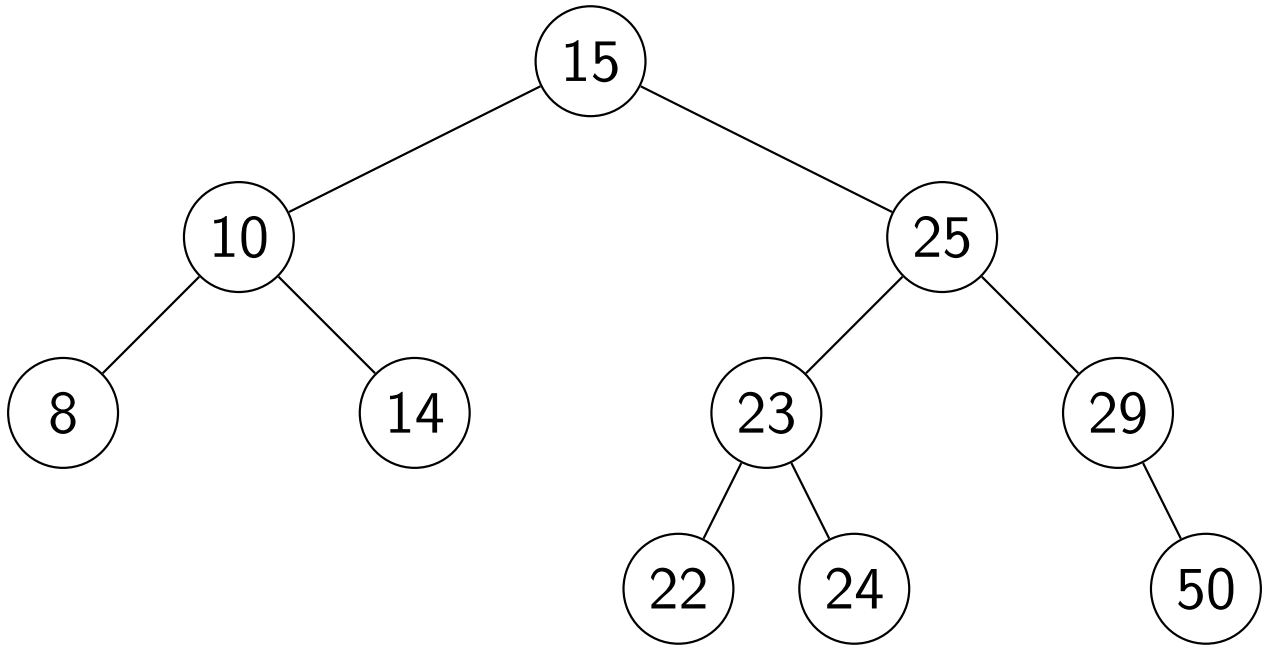
Deletion in a BST

- First search for the node x that contains the key.
- If x is a **leaf** (both subtrees are empty), delete it.
- If x has one non-empty subtree, move child up



Deletion in a BST

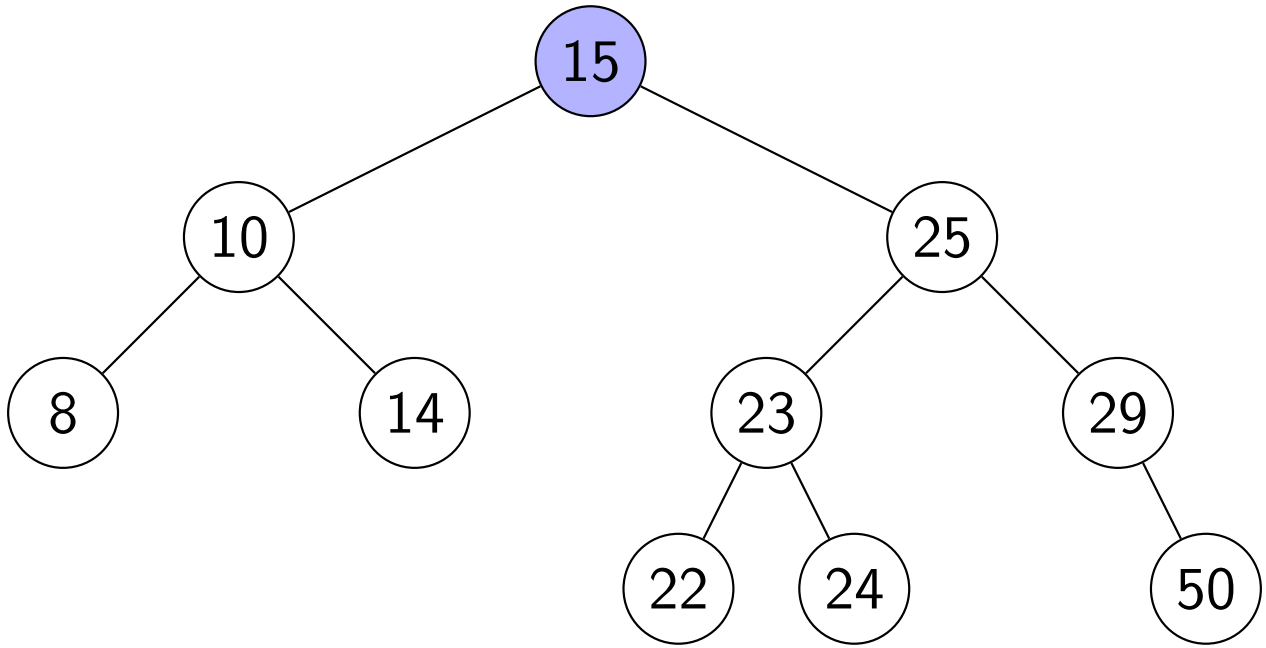
- First search for the node x that contains the key.
- If x is a **leaf** (both subtrees are empty), delete it.
- If x has one non-empty subtree, move child up



Deletion in a BST

- First search for the node x that contains the key.
- If x is a **leaf** (both subtrees are empty), delete it.
- If x has one non-empty subtree, move child up
- Else, swap key at x with key at **successor** node and then delete that node.

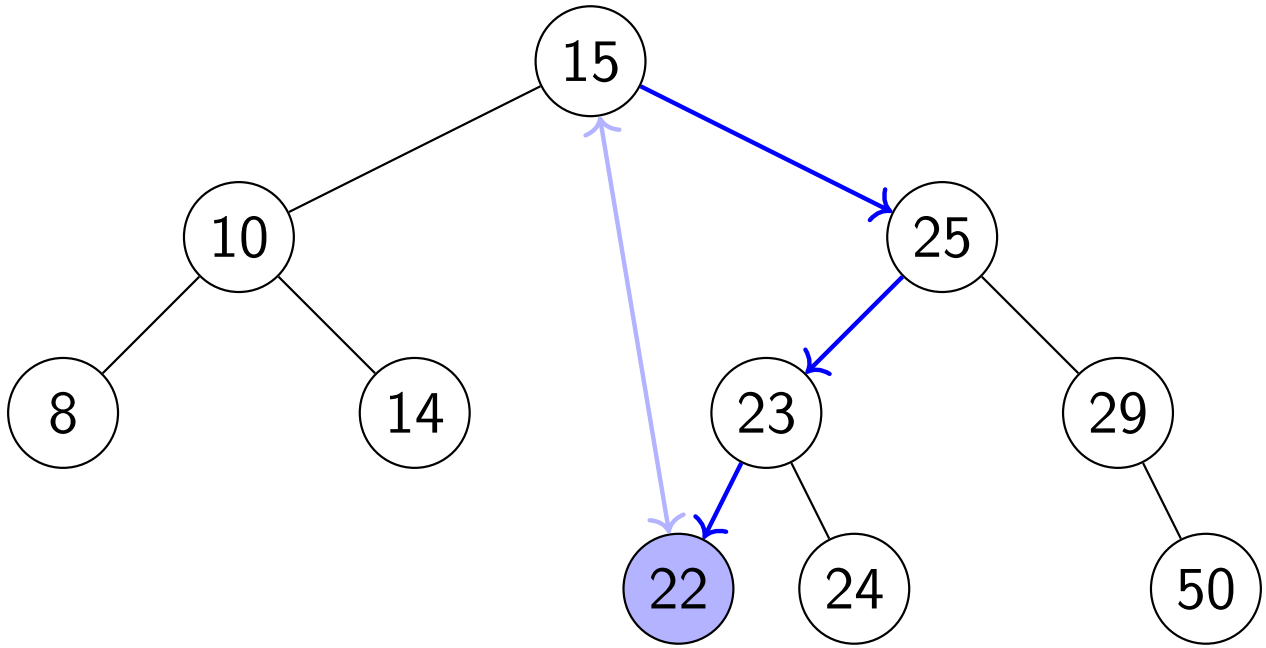
(Successor: next-smallest among all keys in the dictionary.)



Deletion in a BST

- First search for the node x that contains the key.
- If x is a **leaf** (both subtrees are empty), delete it.
- If x has one non-empty subtree, move child up
- Else, swap key at x with key at **successor** node and then delete that node.

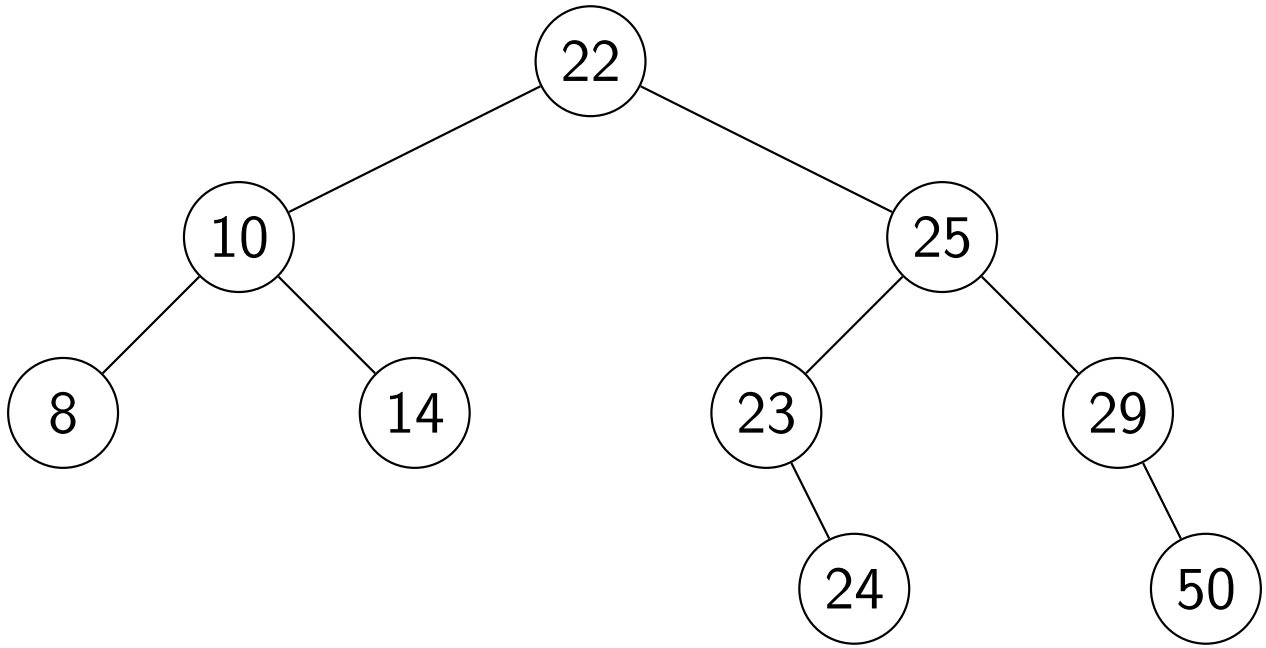
(Successor: next-smallest among all keys in the dictionary.)



Deletion in a BST

- First search for the node x that contains the key.
- If x is a **leaf** (both subtrees are empty), delete it.
- If x has one non-empty subtree, move child up
- Else, swap key at x with key at **successor** node and then delete that node.

(Successor: next-smallest among all keys in the dictionary.)



Height of a BST

BST::search, *BST::insert*, *BST::delete* all have cost $\Theta(h)$, where h = height of the tree = max. path length from root to leaf

- worst-case height: $n - 1 = \Theta(n)$
- so worst-case time $\Theta(n)$ for all operations

Goal: Create subclasses of BSTs where the height is *always* good.

- Impose a structural property.
- Argue that the property implies logarithmic height.
- Discuss how to maintain the property during operatons.

Outline

4. Dictionaries and Balanced Search Trees
   - ADT Dictionary
   - Binary Search Trees
   - AVL Trees
   - Restructuring a BST: Rotations
   - Insertion in AVL Trees
   - Deletion in AVL Trees

AVL trees

Introduced by Adel'son-Vel'skiĭ and Landis in 1962, an **AVL tree** is a BST with an additional **height-balance property** at every node:

The heights of the left and right subtree differ by at most 1.

Rephrase: If node z has left subtree L and right subtree R, then

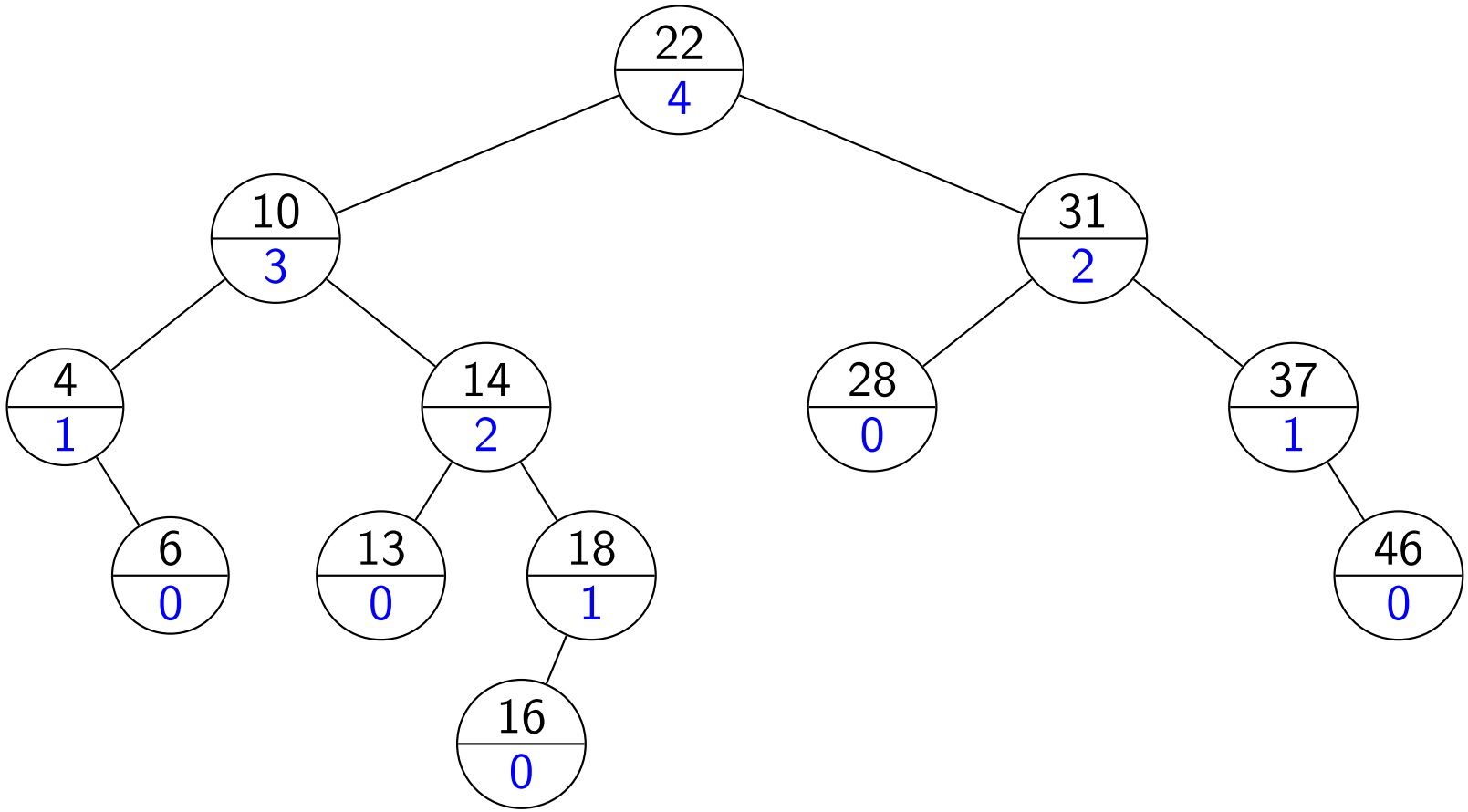
$$balance(z) = height(R) - height(L) \text{ must be in } \{-1, 0, 1\}$$

Things to show:

- This *structural condition* implies logarithmic height.
- After *insert* and *delete*, we can restore the structural condition within logarithmic time.
  - For this, we need to store at each node z the height of the subtree rooted at it.

AVL tree example

(The lower numbers indicate the height of the subtree.)



AVL trees: Height

Theorem: An AVL tree on n nodes has $\Theta(\log n)$ height.
$\Rightarrow$ *search*, *BST::insert*, *BST::delete* all cost $\Theta(\log n)$ in the *worst case!*

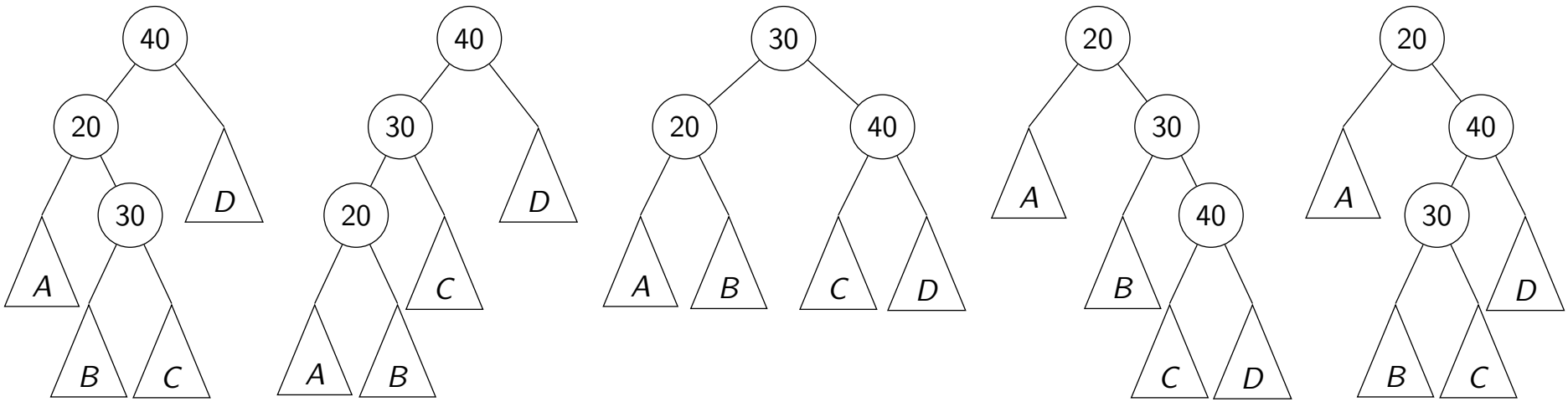
Proof:

- Define $N(h)$ to be the *least* number of nodes in a height-h AVL tree.
- What is a recurrence relation for $N(h)$?
- What does this recurrence relation resolve to?

Outline

4. Dictionaries and Balanced Search Trees
   - ADT Dictionary
   - Binary Search Trees
   - AVL Trees
   - Restructuring a BST: Rotations
   - Insertion in AVL Trees
   - Deletion in AVL Trees

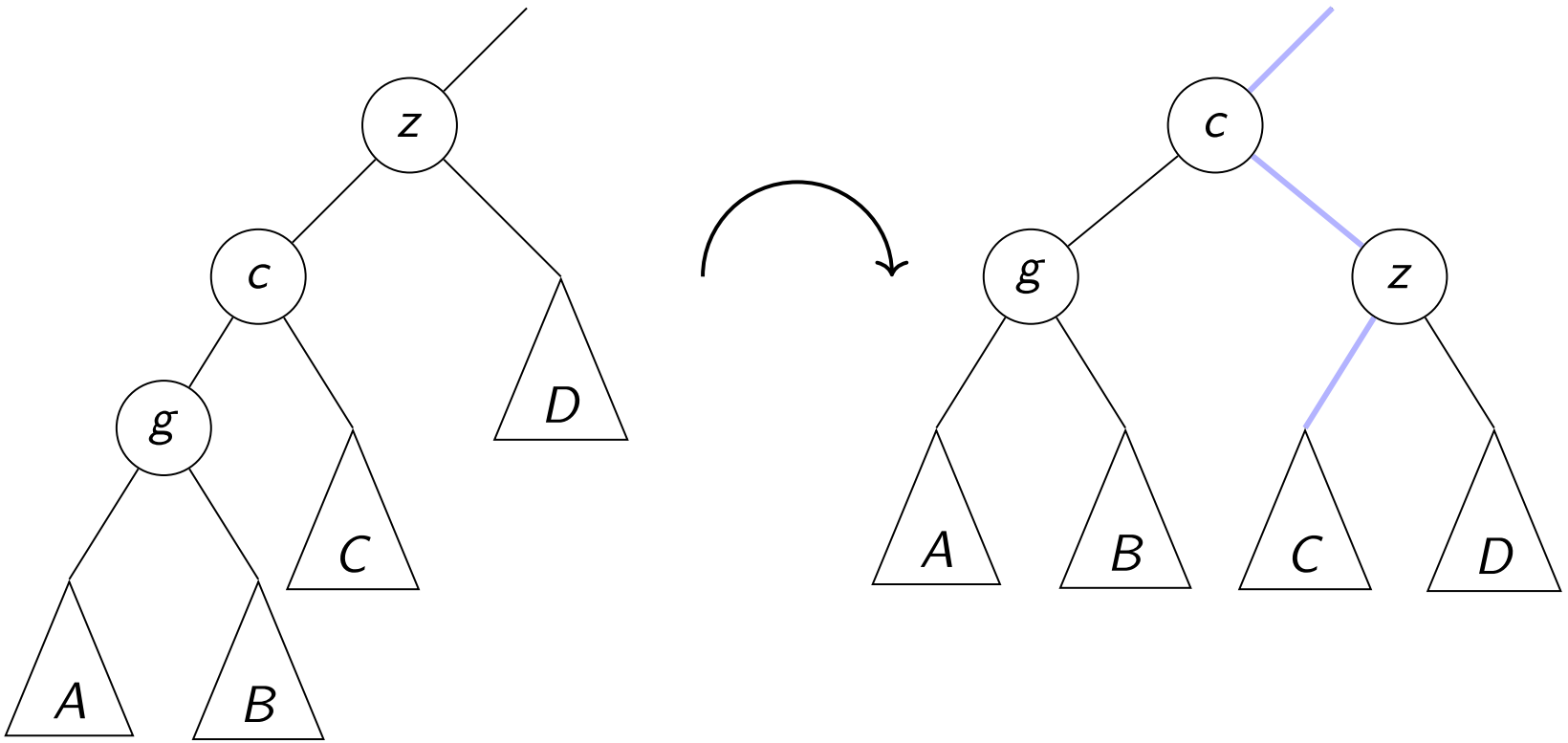
Changing structure without changing order



Goal: Change the *structure* locally while preserving the order property, and improving balances

Right Rotation

This is a **right rotation** on node z:



Note: Only $O(1)$ links are changed. Useful to fix left-left imbalance.

Right Rotation: Pseudocode

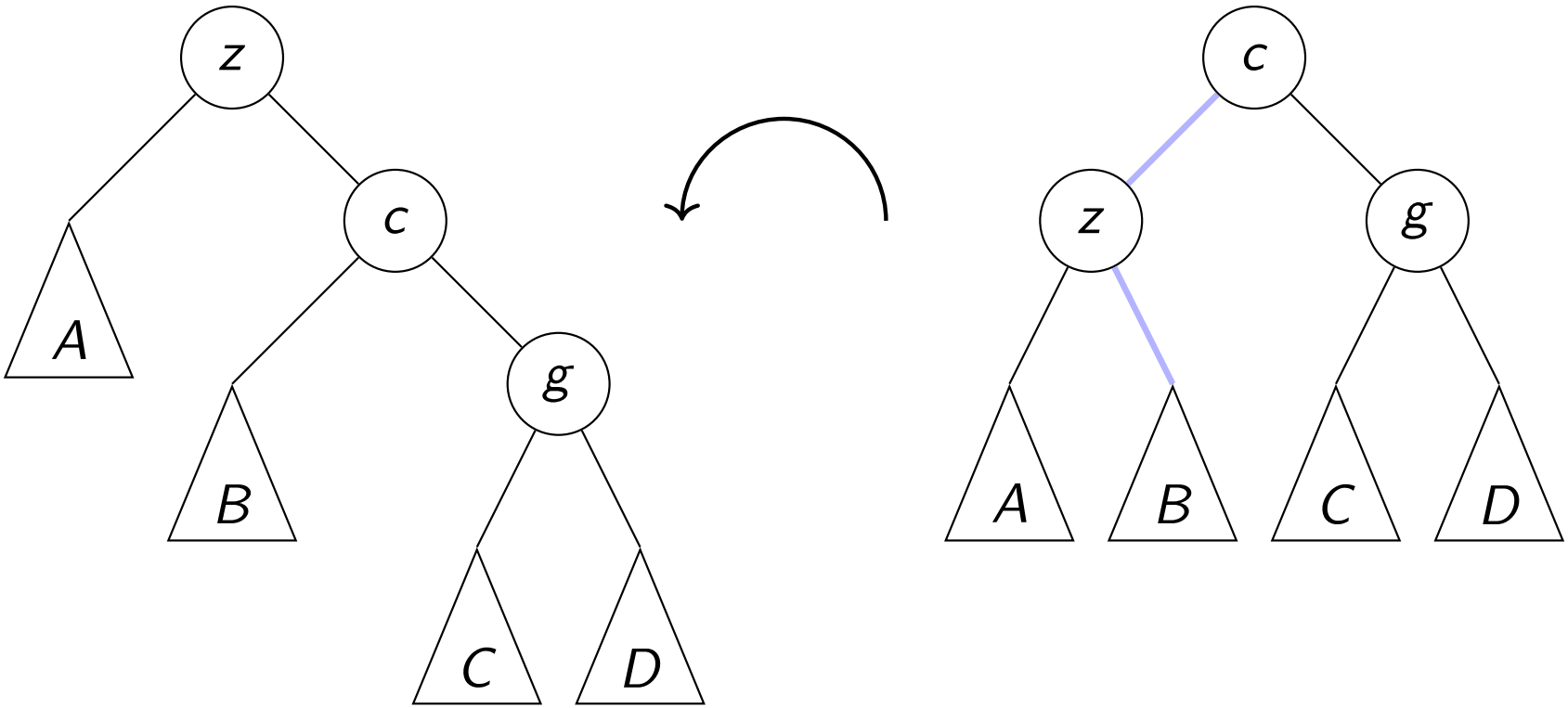
```
rotate-right(z)
1.  c ← z.left

2.  // fix links connecting to above
3.  c.parent ← (p ← z.parent)
4.  if p = NULL then root ← c else
5.        if p.left = z then p.left ← c else p.right ← c

6.  // actual rotation
7.  z.left ← c.right, c.right.parent ← z
8.  c.right ← z, z.parent ← c

9.  set-height-from-subtrees(z), set-height-from-subtrees(c)
10. return c      // returns new root of subtree
```

```
set-height-from-subtrees(u)
1.  u.height ← 1 + max{u.left.height, u.right.height}
```

Run-time: $O(1)$, but the heights of the ancestors of z may not be correct

Left Rotation

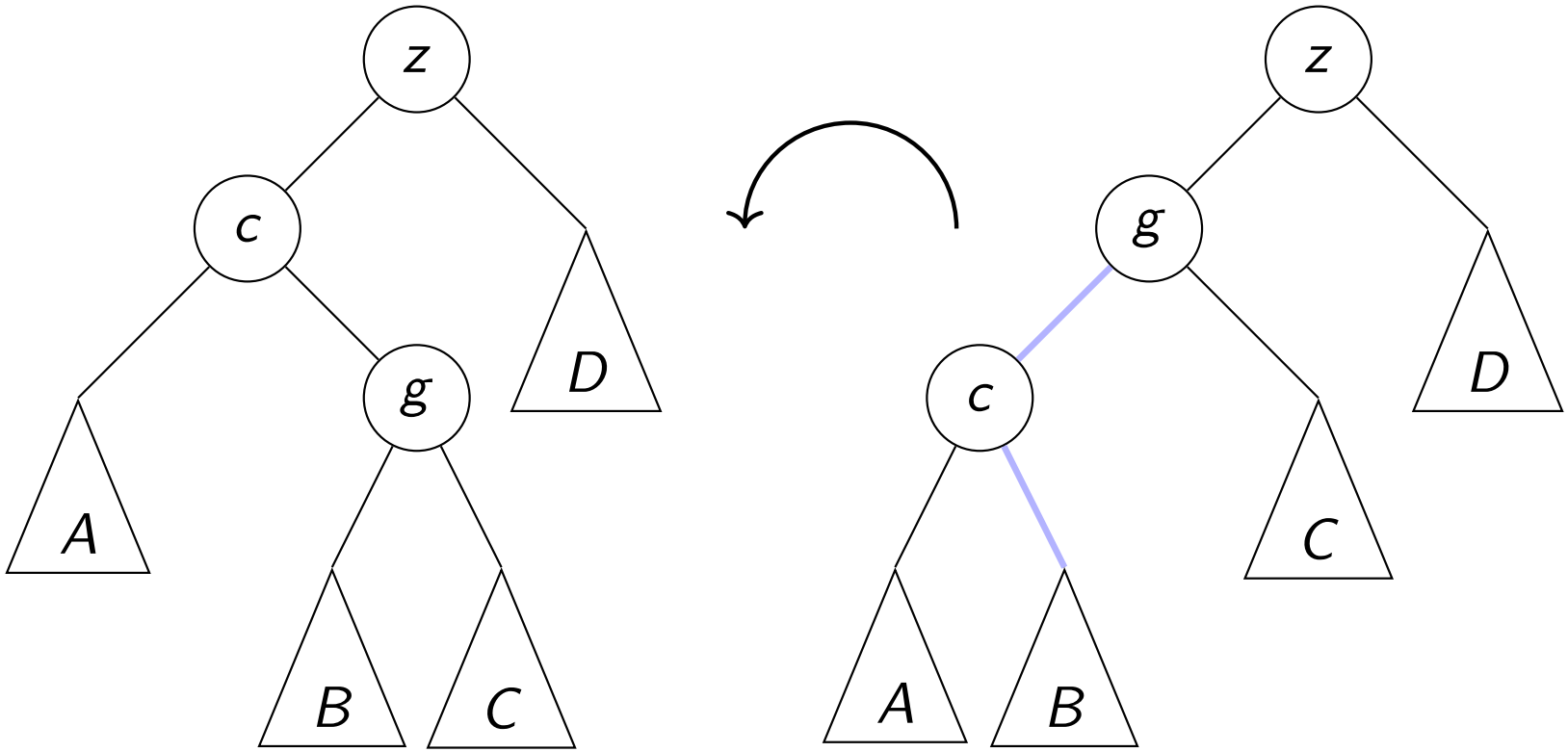
Symmetrically, this is a **left rotation** on node z:



Again, only $O(1)$ links need to be changed.
Useful to fix right-right imbalance.

Double Right Rotation

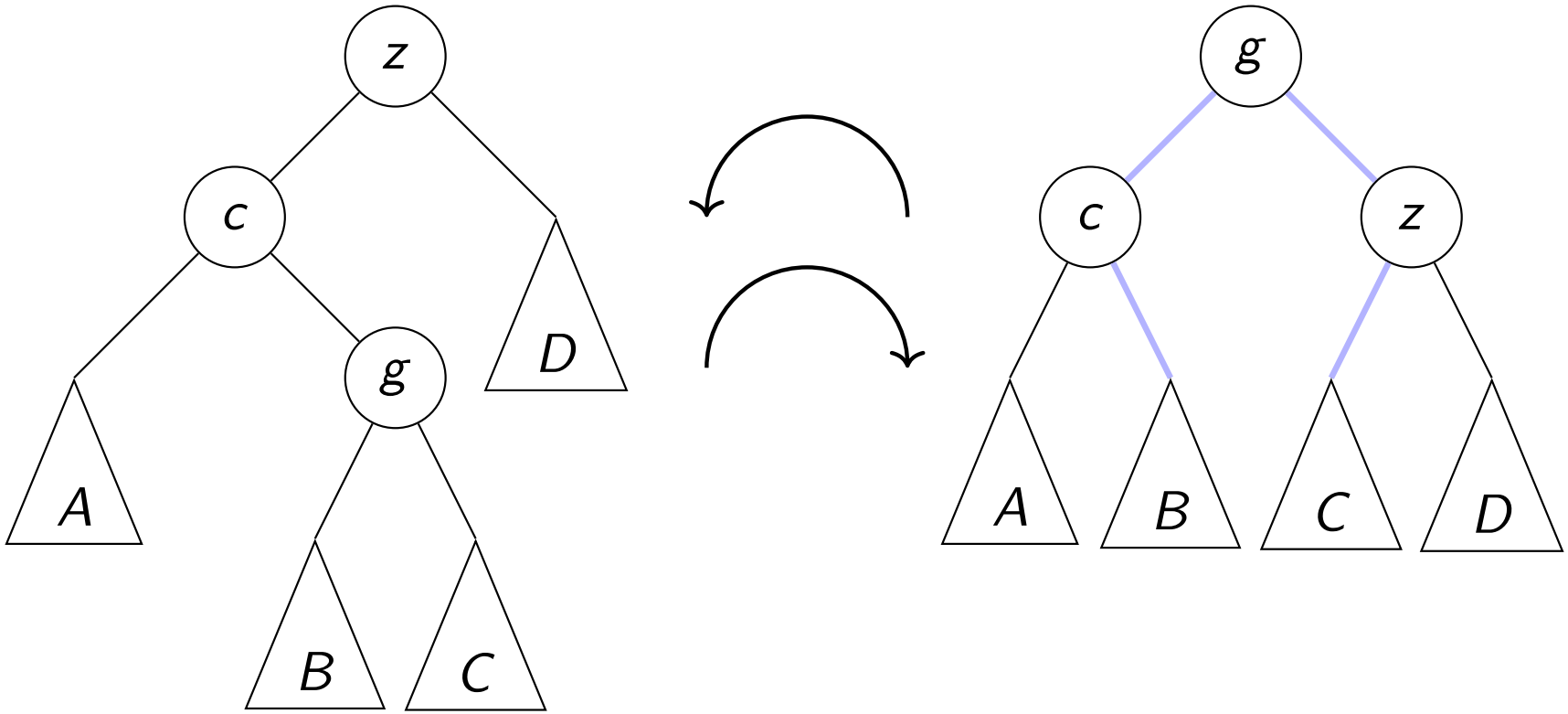
This is a **double right rotation** on node z:



First, a left rotation at c.

Double Right Rotation

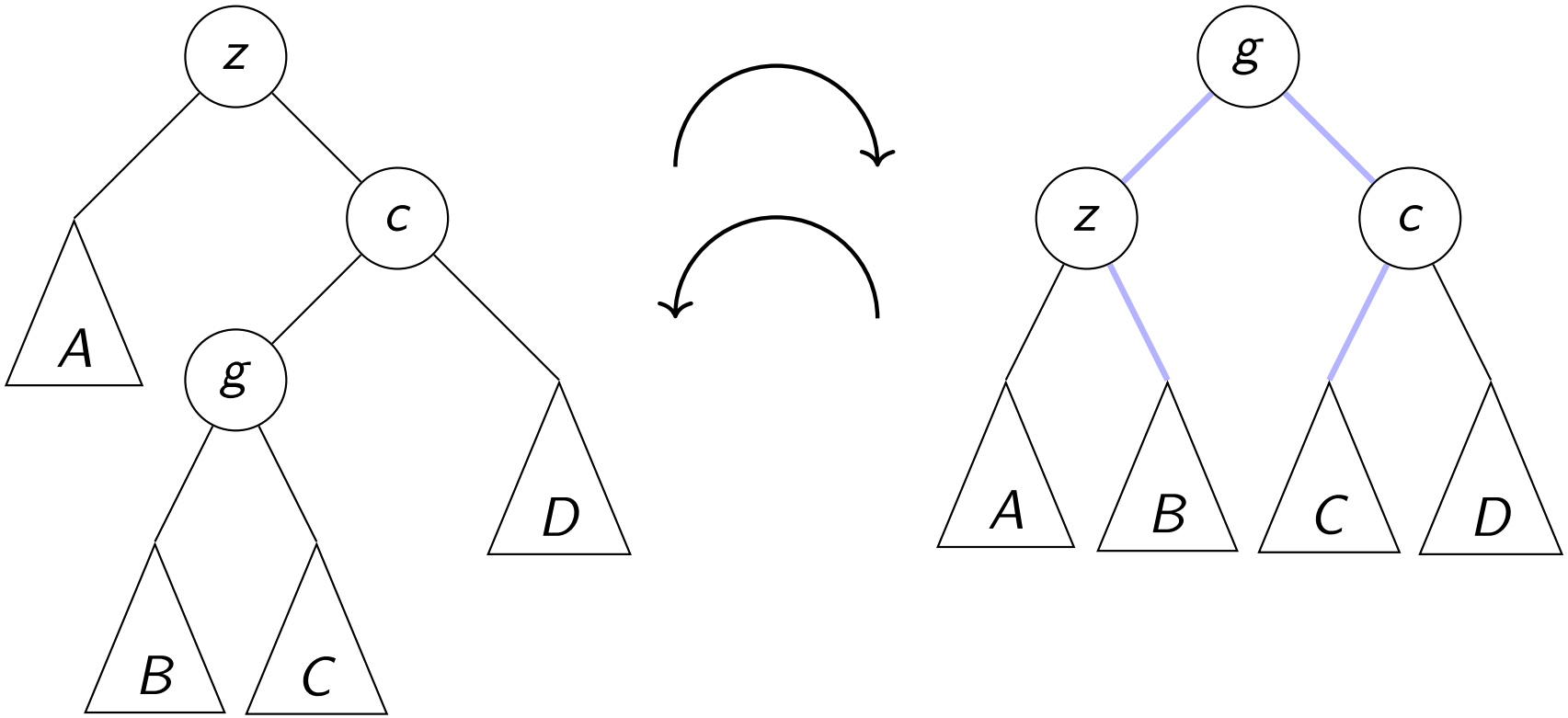
This is a **double right rotation** on node z:



First, a left rotation at c.
Second, a right rotation at z.

Double Left Rotation

Symmetrically, there is a **double left rotation** on node z:



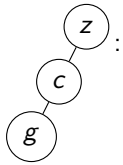
First, a right rotation at c.
Second, a left rotation at z.

Fixing a slightly-unbalanced AVL tree

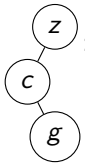
z unbalanced, c taller child of z (no tie possible), g taller child of c.
If tie for g: choose the one that does single rotation

restructure(z) (returns reference to new root)

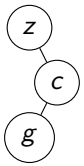
case



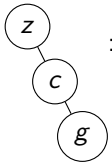
```
// Right rotation
return rotate-right(z)
```



```
// Double-right rotation
rotate-left(c)
return rotate-right(z)
```



```
// Double-left rotation
rotate-right(c)
return rotate-left(z)
```



```
// Left rotation
return rotate-left(z)
```

Rule: The middle key of g, c, z becomes the new root.

Outline

4. Dictionaries and Balanced Search Trees
   - ADT Dictionary
   - Binary Search Trees
   - AVL Trees
   - Restructuring a BST: Rotations
   - Insertion in AVL Trees
   - Deletion in AVL Trees

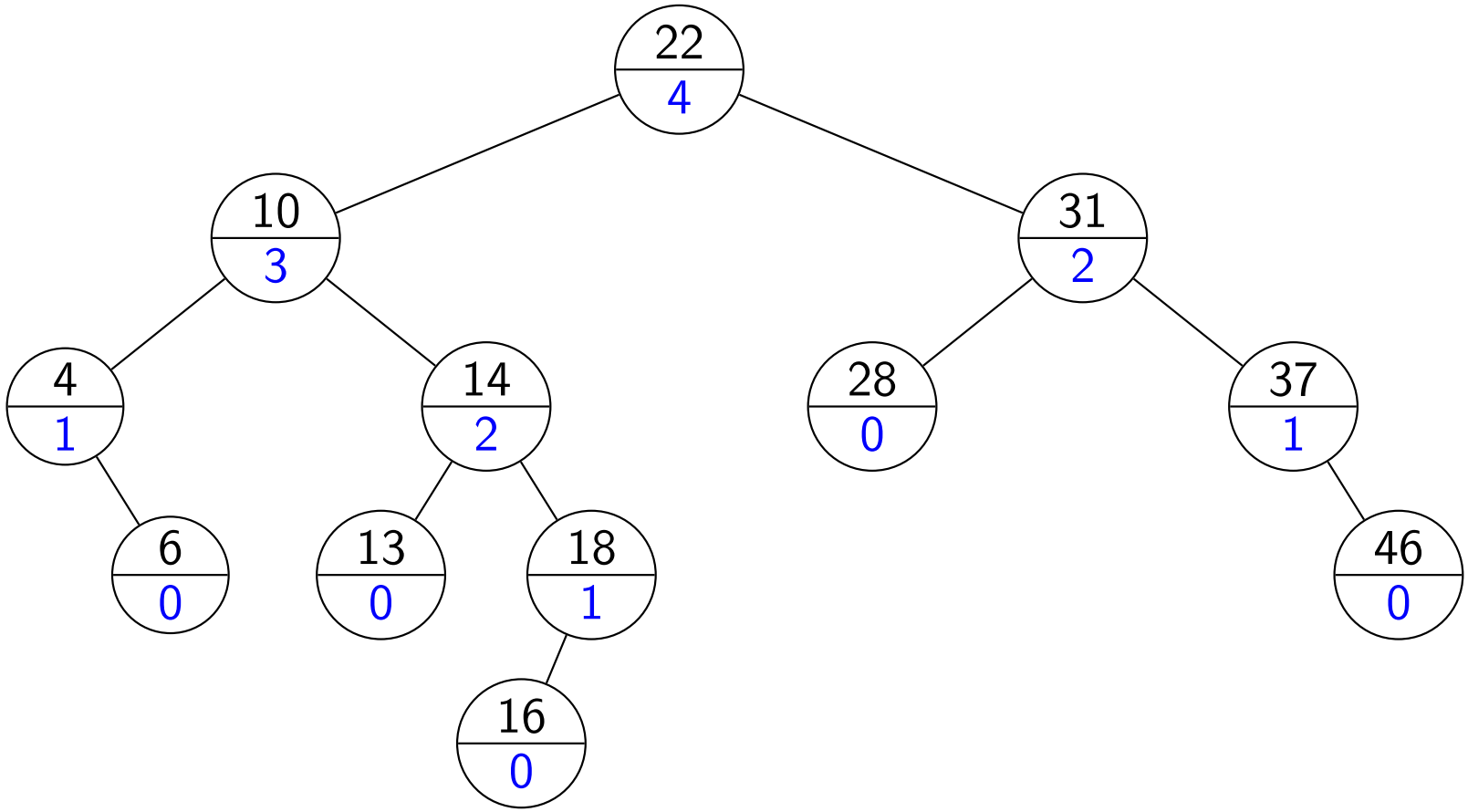
AVL trees: Insertion

To perform *AVL::insert*(k, v):

- First, insert (k, v) with the usual BST insertion.
- We assume that this returns the new leaf z where the key was stored.
- Then, **move up** the tree from z
  (we assume for this that we have parent-links)
- Update height (constant time)
- If the height difference becomes ± 2 at node z, then z is **unbalanced**.
  Must re-structure the tree to rebalance.

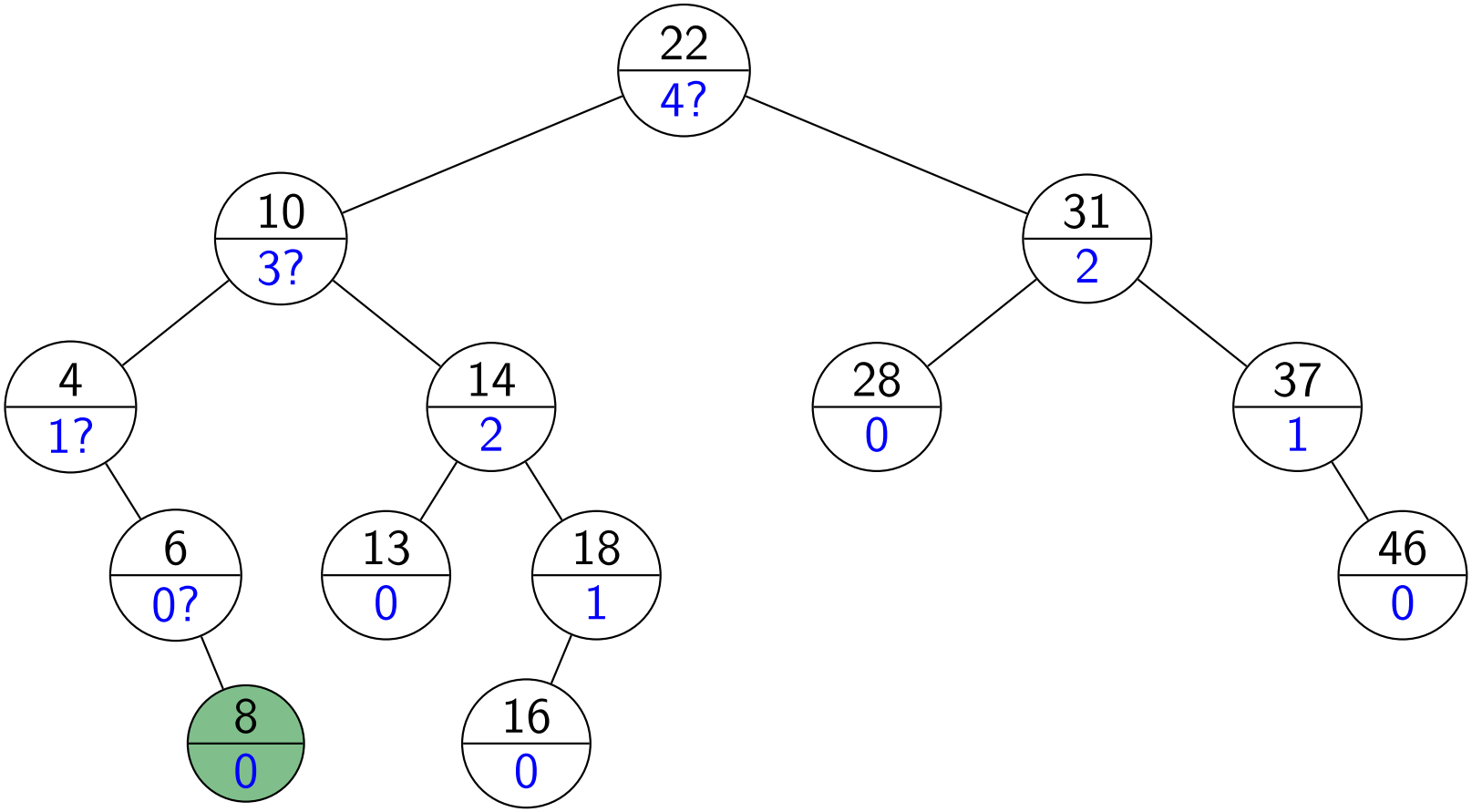
AVL tree insertion: Example

Example: *AVL::insert*(8)



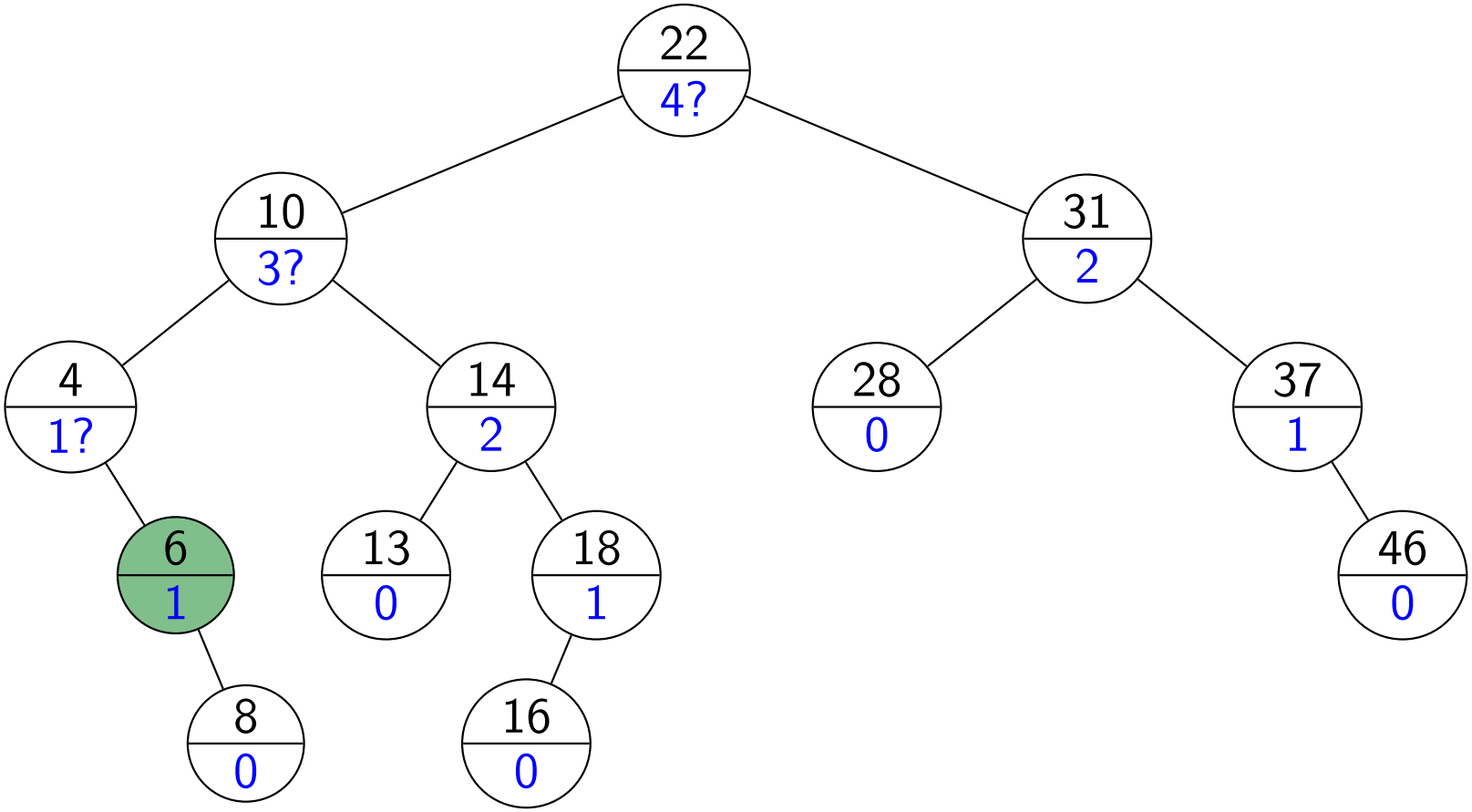
AVL tree insertion: Example

Example: *AVL::insert*(8)



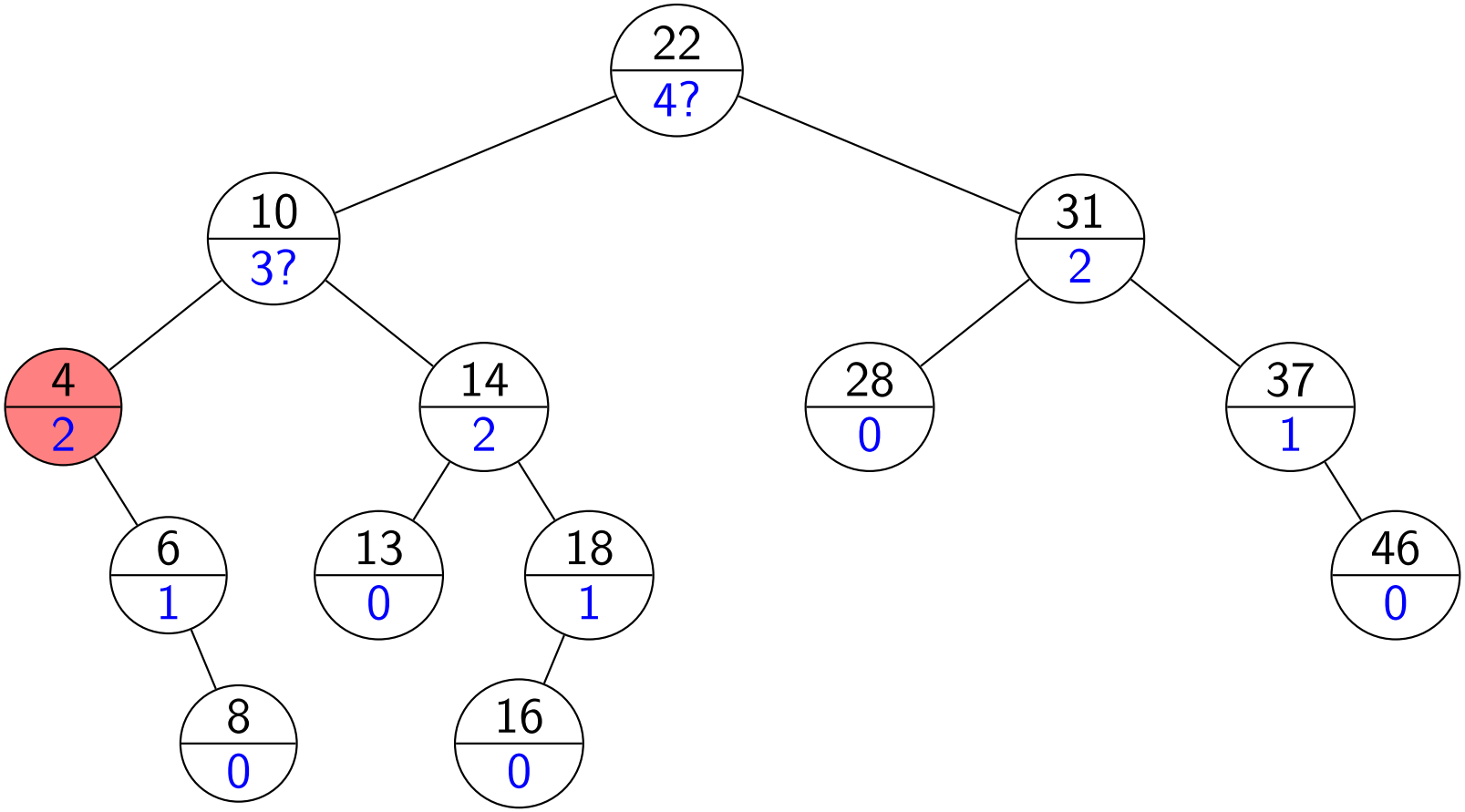
AVL tree insertion: Example

Example: *AVL::insert*(8)



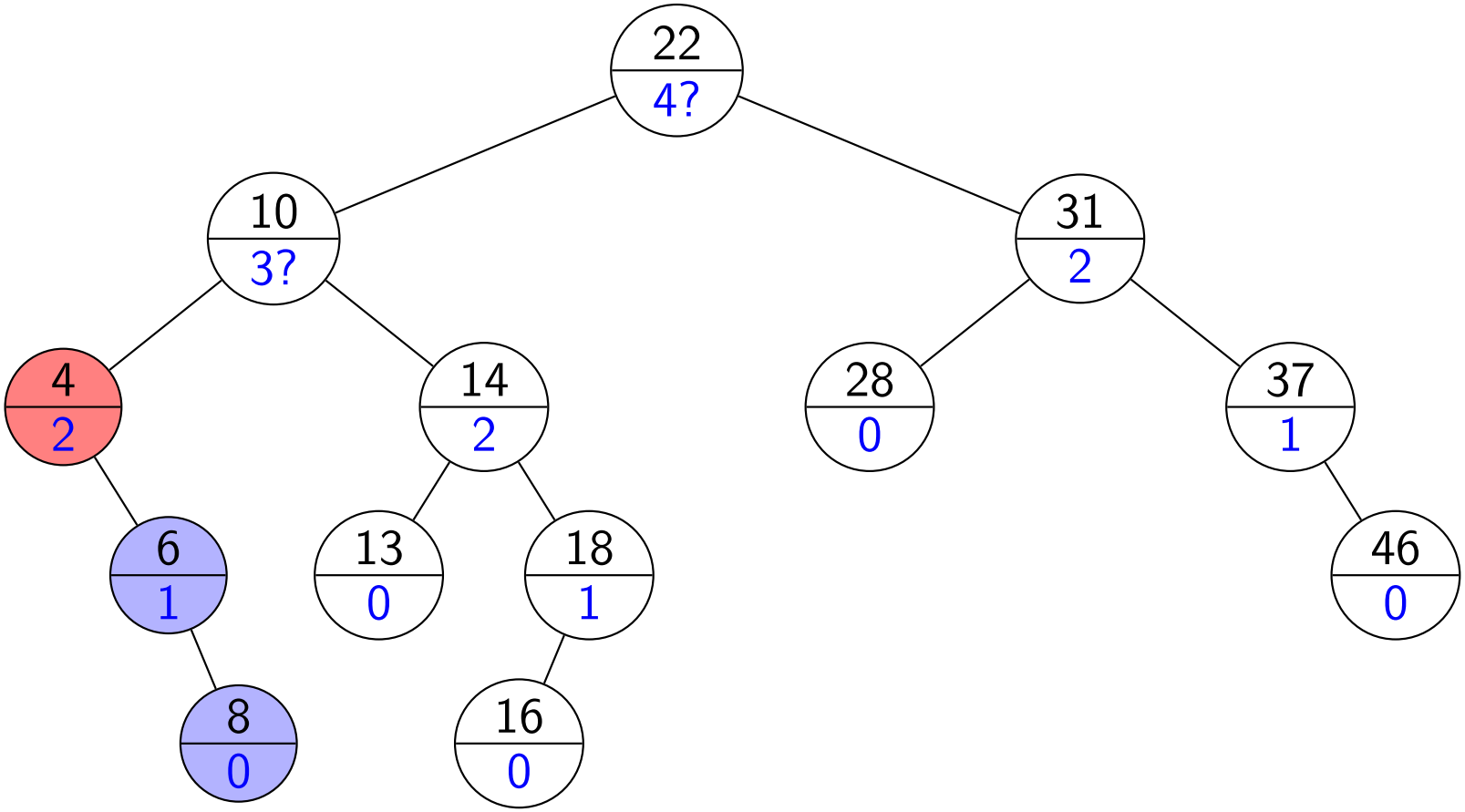
AVL tree insertion: Example

Example: *AVL::insert*(8)



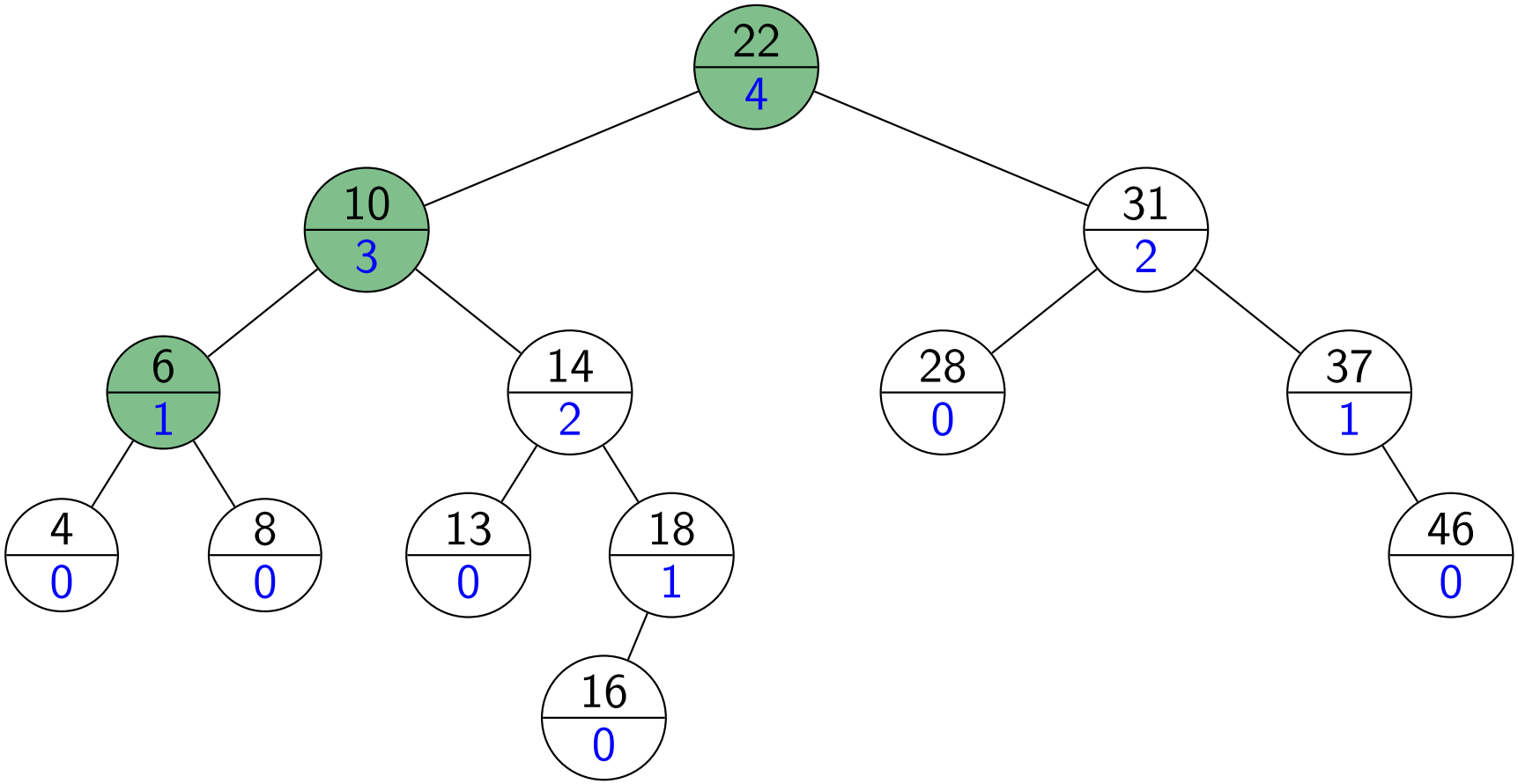
AVL tree insertion: Example

Example: *AVL::insert*(8)



AVL tree insertion: Example

Example: *AVL::insert*(8)



AVL tree insertion

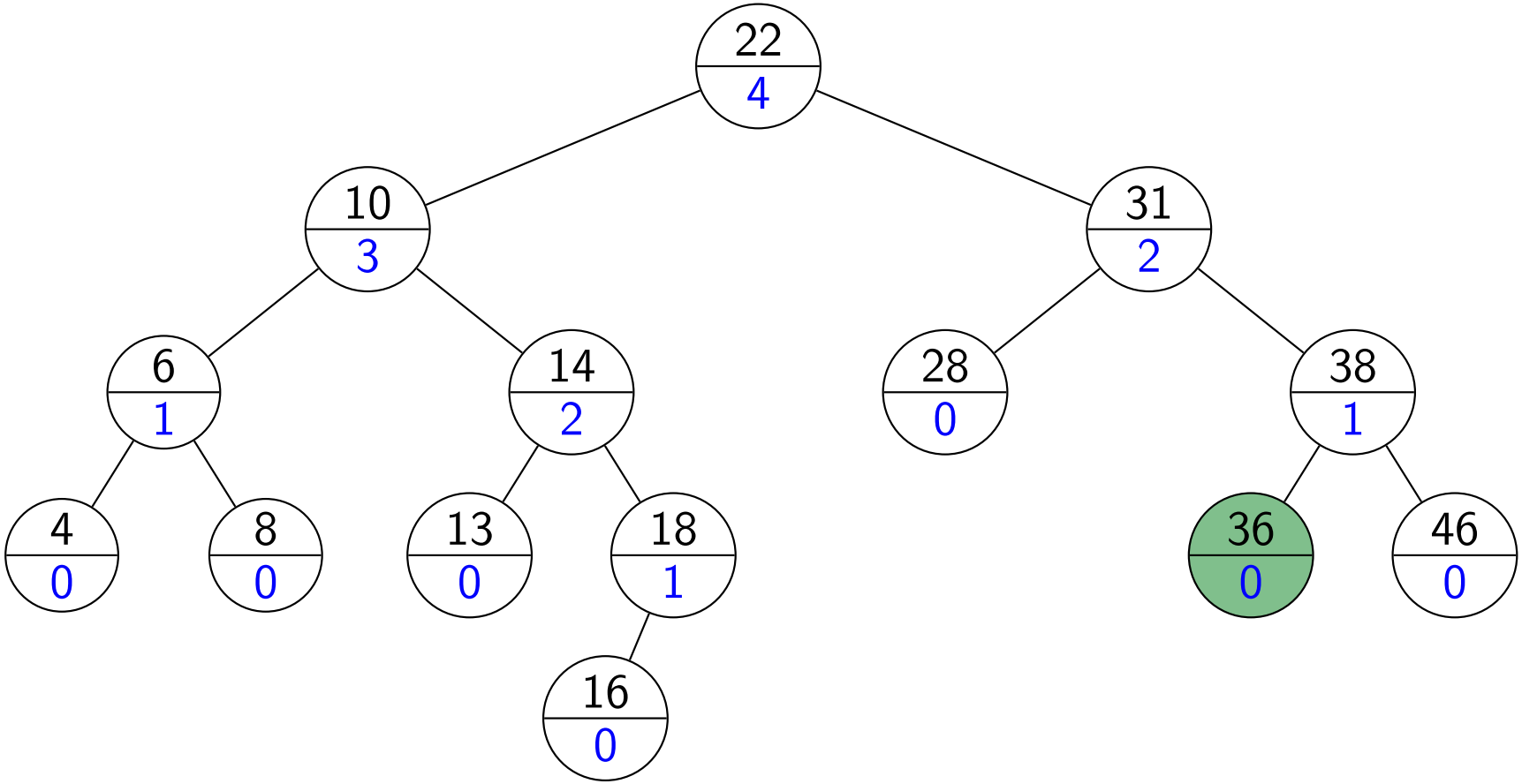
```
AVL::insert(k, v)
1.  z ← BST::insert(k, v)              // new leaf with k
2.  while (z is not NULL)
3.      if (|z.left.height − z.right.height| > 1) then
4.          z ← restructure(z)
5.          break          // can argue that we are done
6.      set-height-from-subtrees(z)
7.      z ← z.parent
```

Can prove: For insertion *one* rotation restores all heights of subtrees.

$\Rightarrow$ No further imbalances, can stop checking.

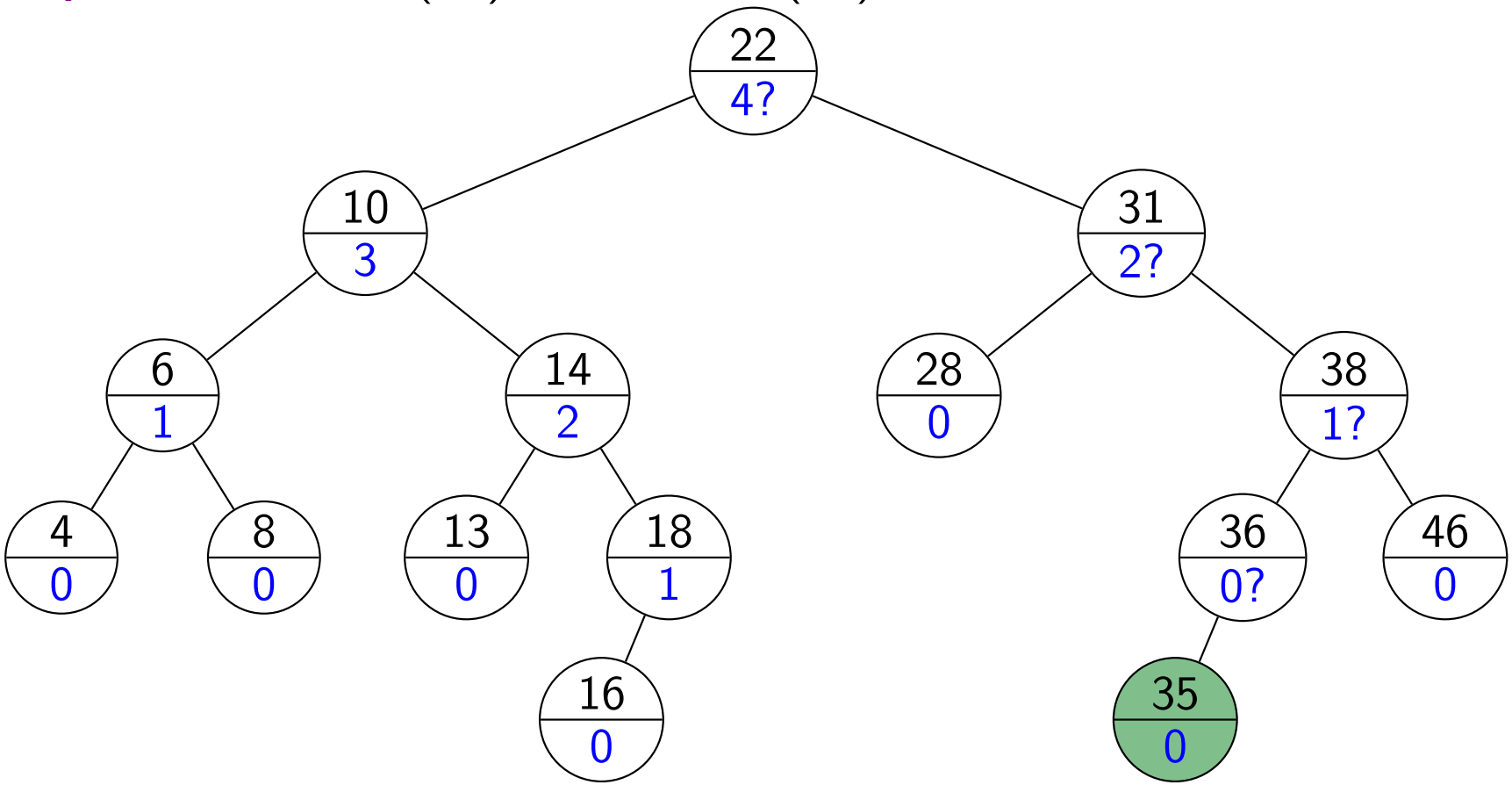
AVL tree insertion: Second example

Example: *AVL::insert*(36), *AVL::insert*(35)



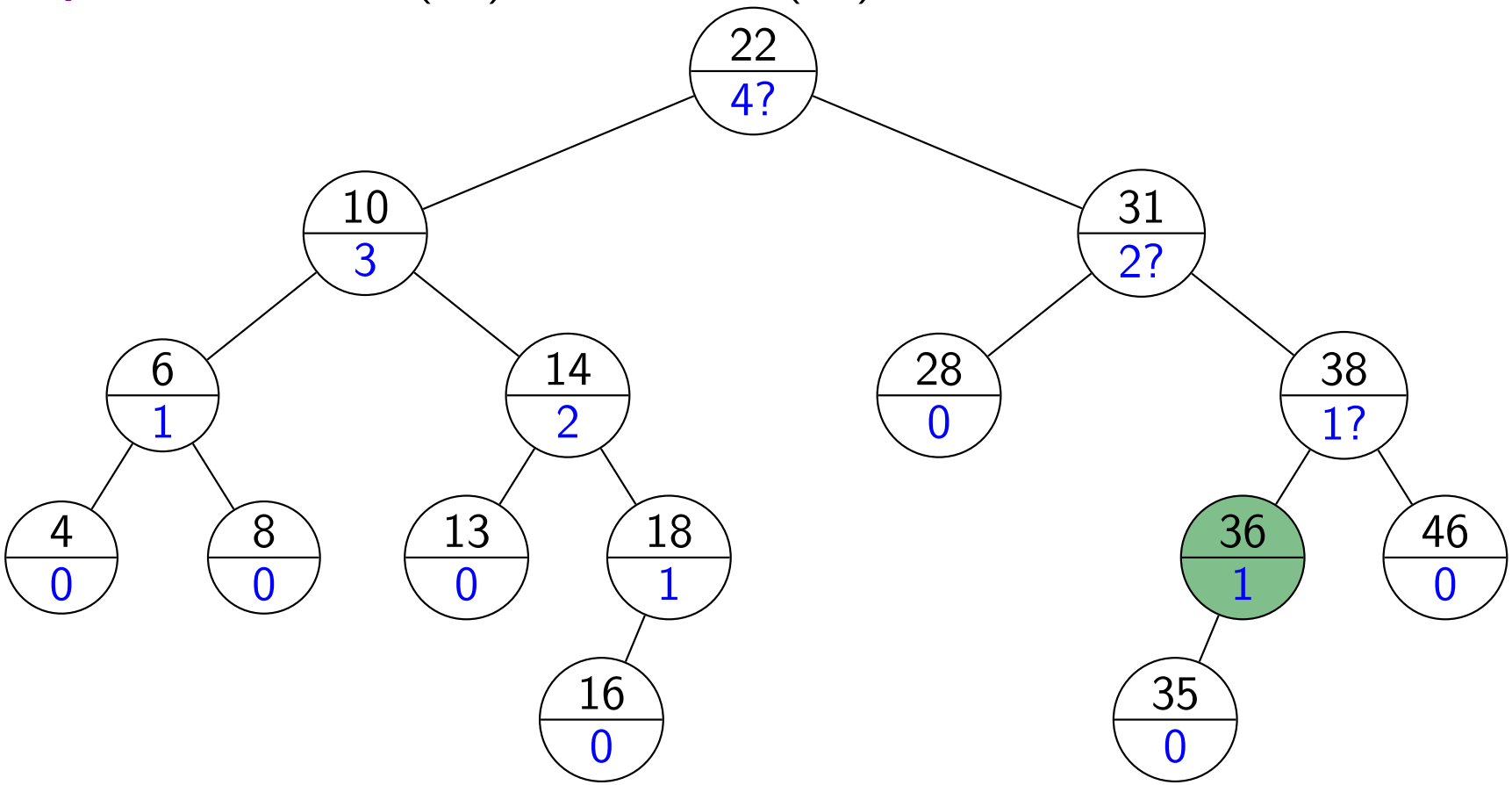
AVL tree insertion: Second example

Example: *AVL::insert*(36), *AVL::insert*(35)



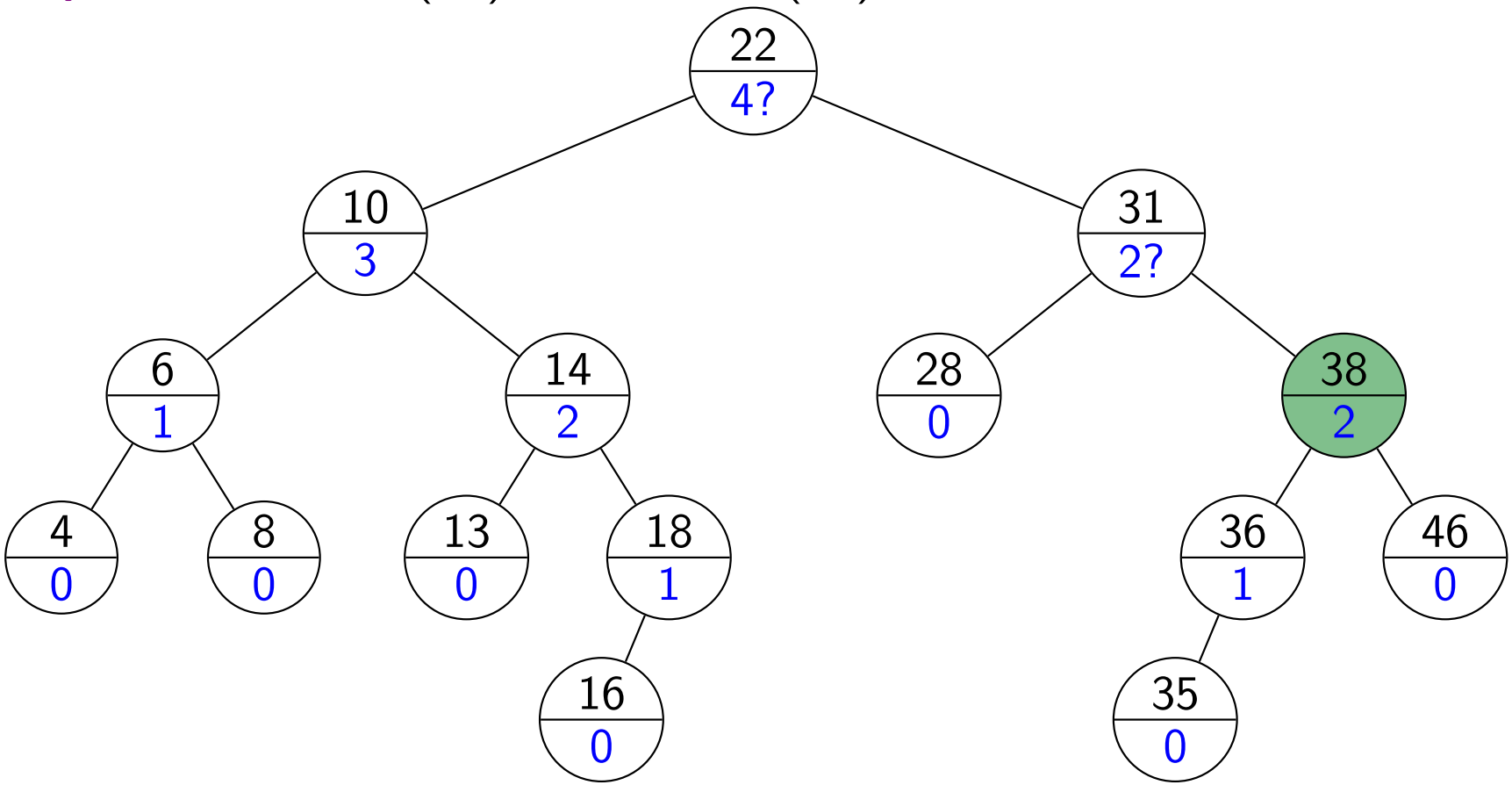
AVL tree insertion: Second example

Example: *AVL::insert*(36), *AVL::insert*(35)



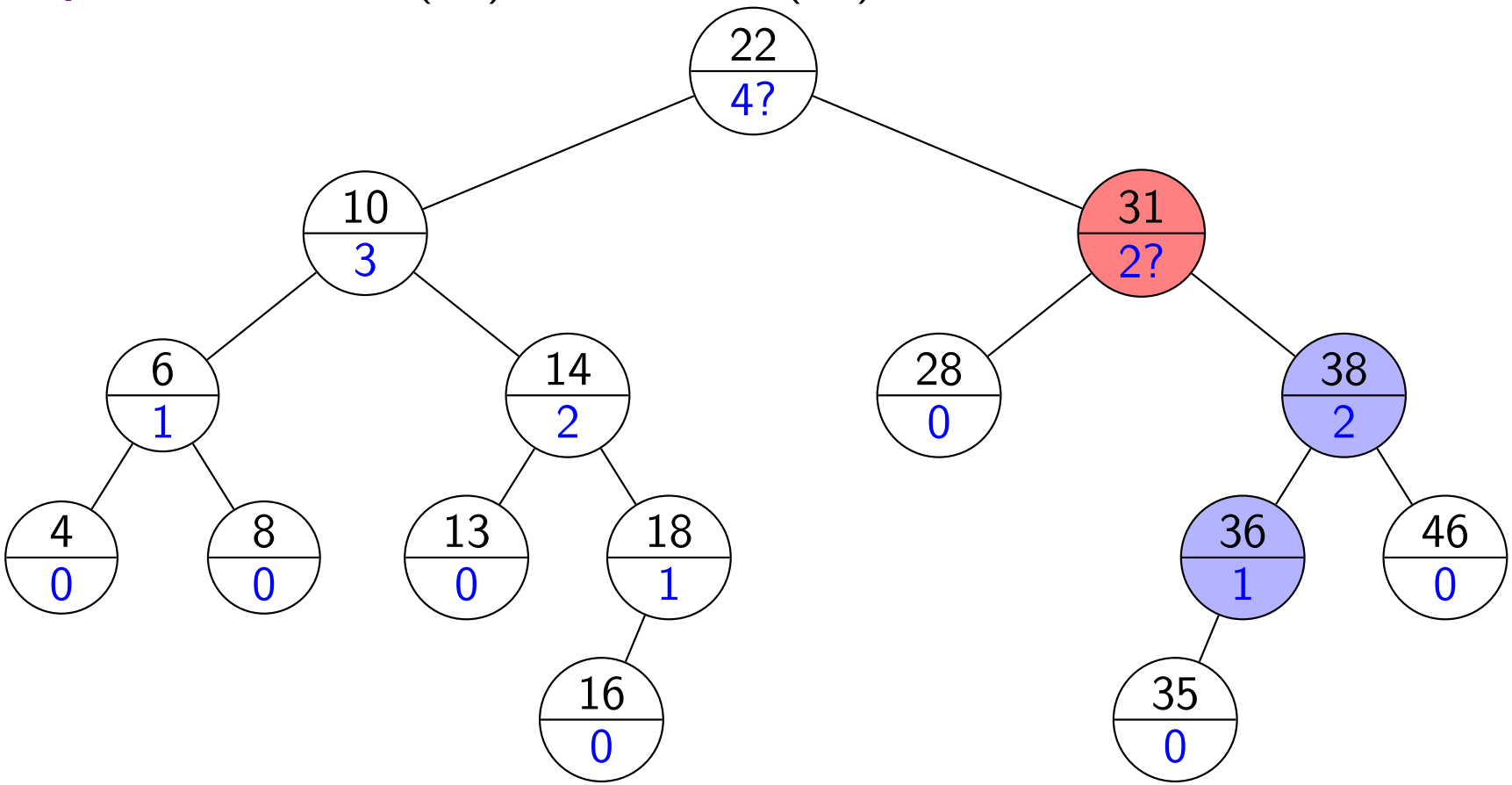
AVL tree insertion: Second example

Example: *AVL::insert*(36), *AVL::insert*(35)



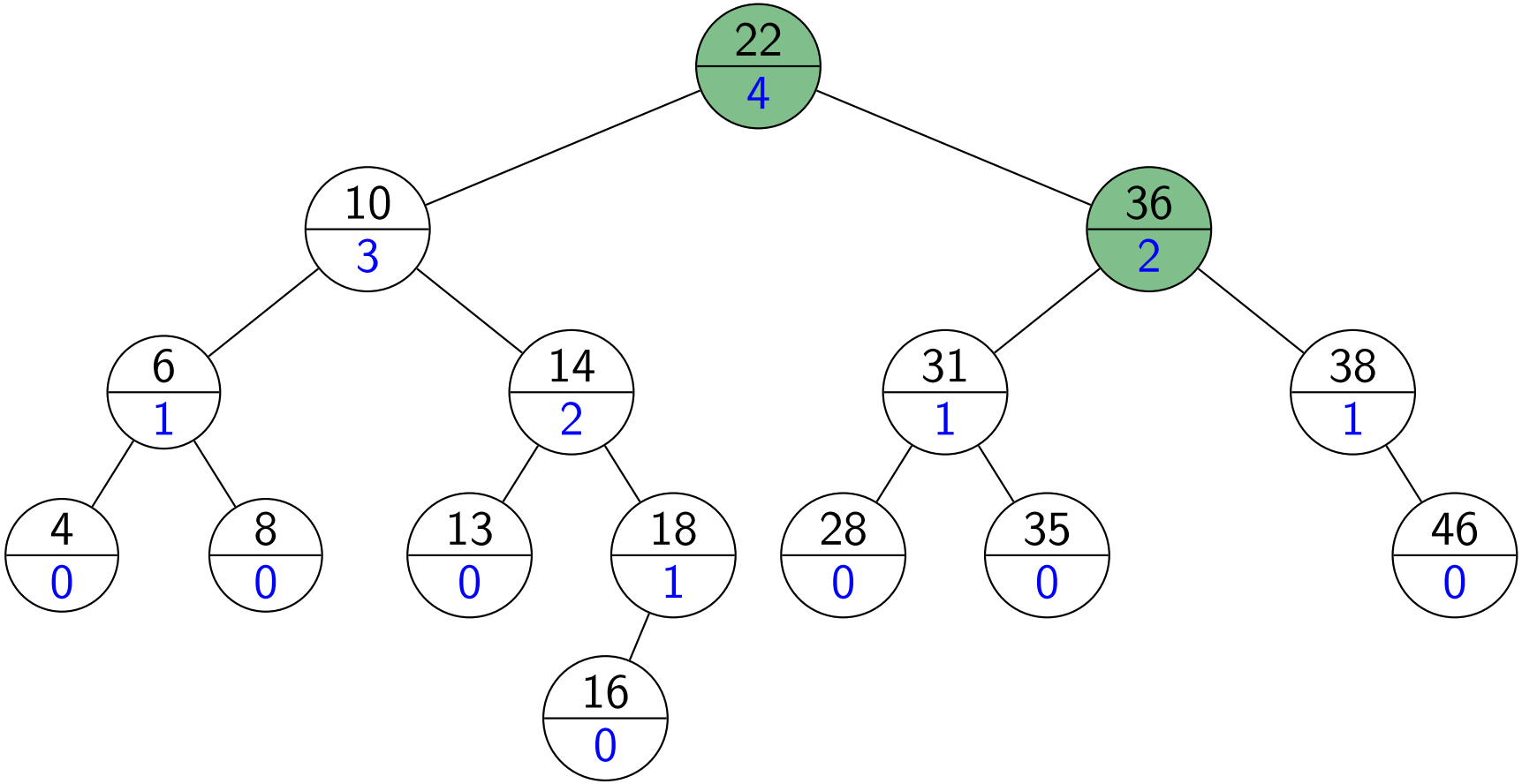
AVL tree insertion: Second example

Example: *AVL::insert*(36), *AVL::insert*(35)



AVL tree insertion: Second example

Example: *AVL::insert*(36), *AVL::insert*(35)



Outline

4. Dictionaries and Balanced Search Trees
   - ADT Dictionary
   - Binary Search Trees
   - AVL Trees
   - Restructuring a BST: Rotations
   - Insertion in AVL Trees
   - Deletion in AVL Trees

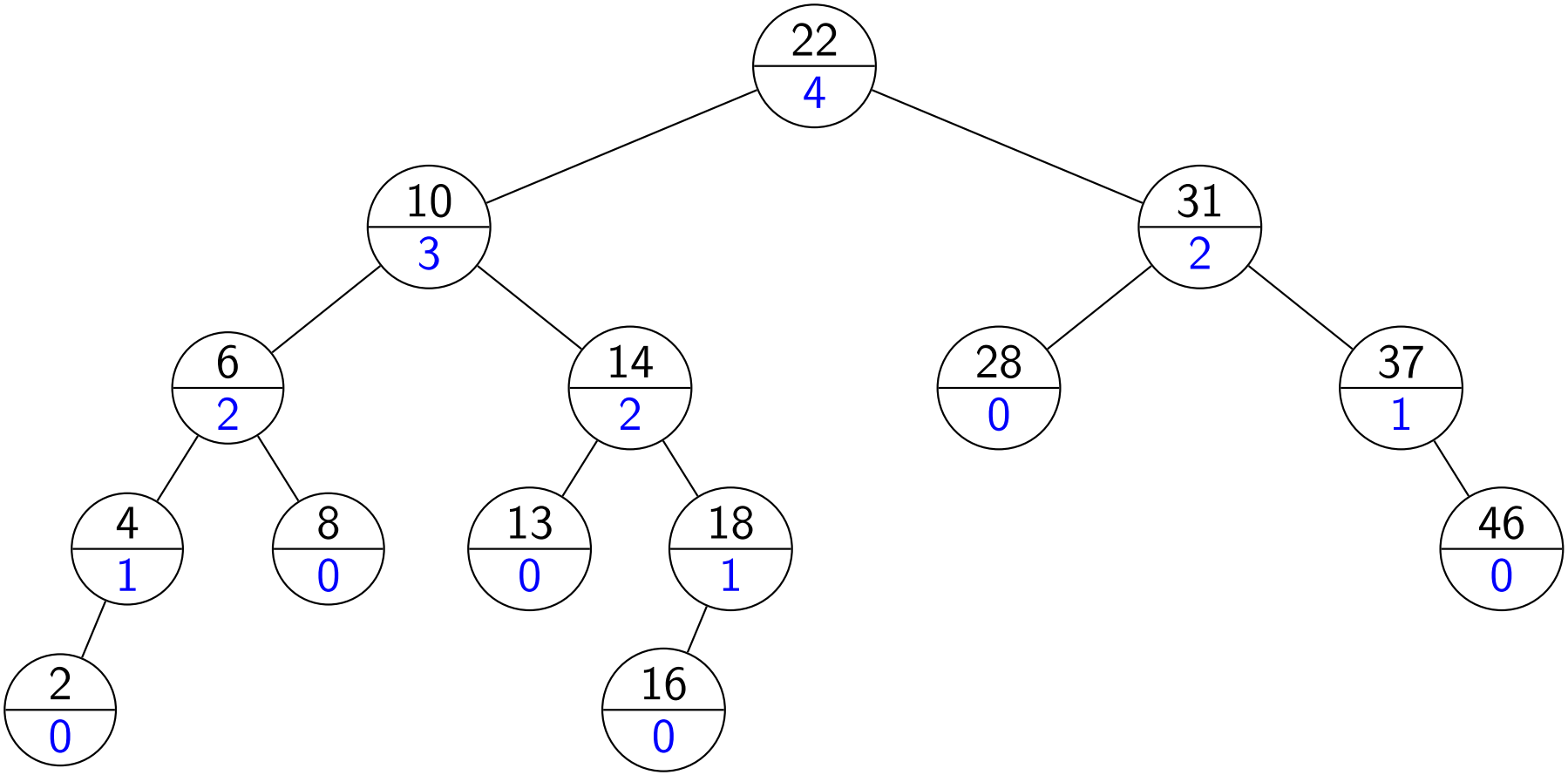
AVL trees: Deletion

Remove the key k with *BST::delete*.
Find node where *structural* change happened.
(This is not necessarily near the node that had k.)
Go back up to root, update heights, and rotate if needed.

```
AVL::delete(k)
1.  z ← BST::delete(k)
2.  // Assume z is the parent of the BST node that was removed
3.  while (z is not NULL)
4.      if (|z.left.height − z.right.height| > 1) then
5.          z ← restructure(z)
6.      // Always continue up the path
7.      set-height-from-subtrees(z)
8.      z ← z.parent
```

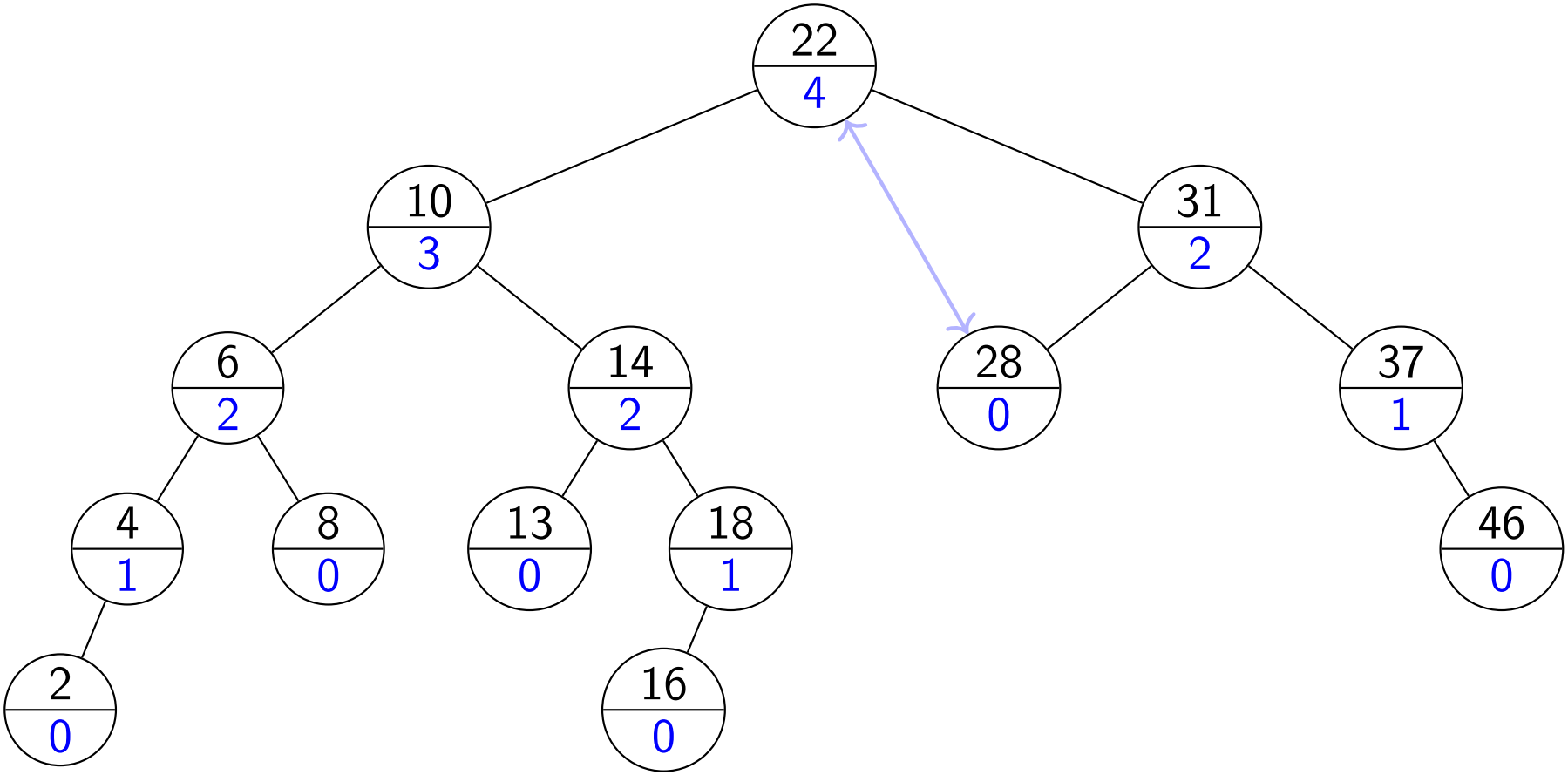
AVL trees deletion: Example

Example: *AVL::delete*(22)



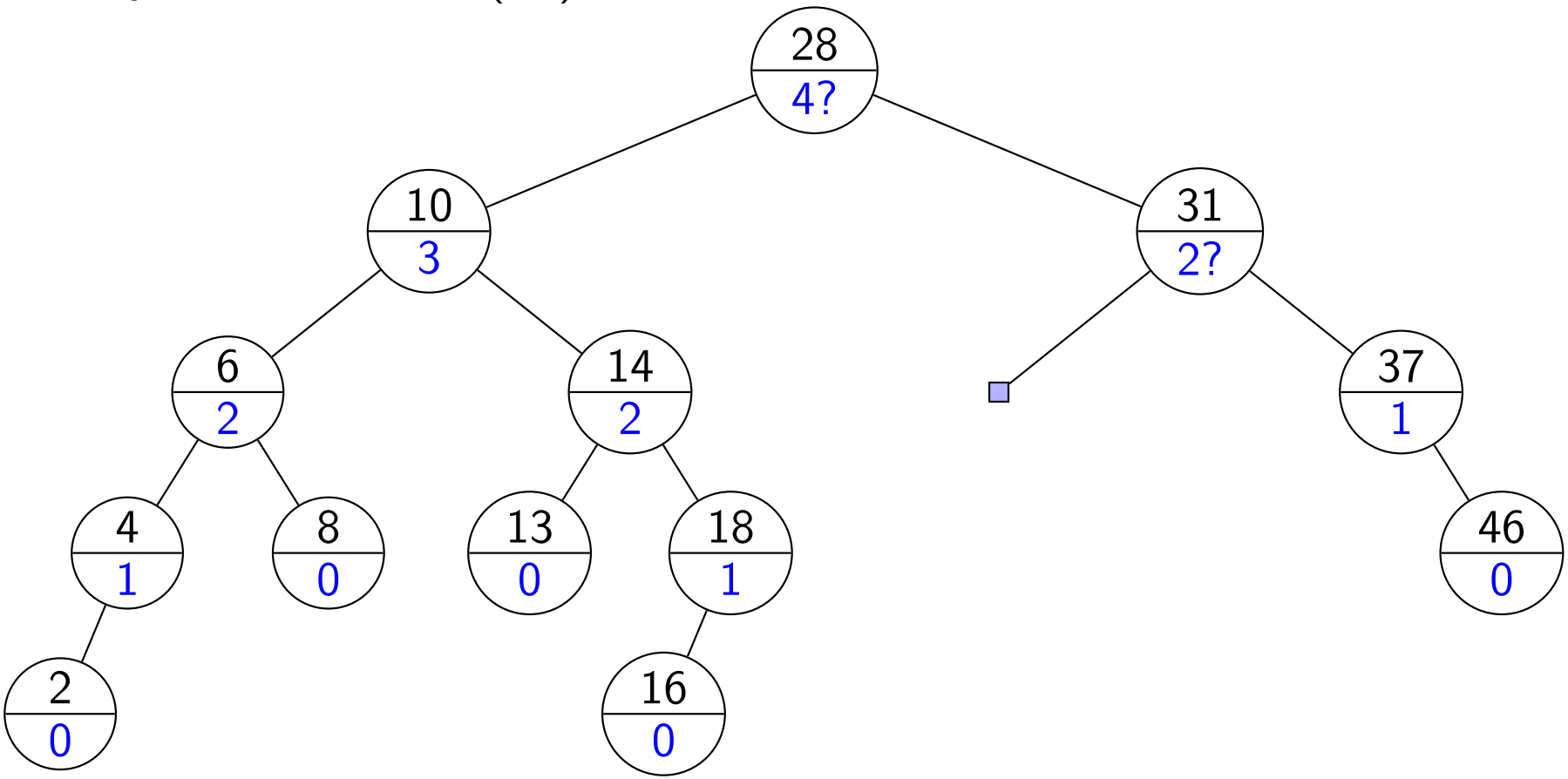
AVL trees deletion: Example

Example: *AVL::delete*(22)



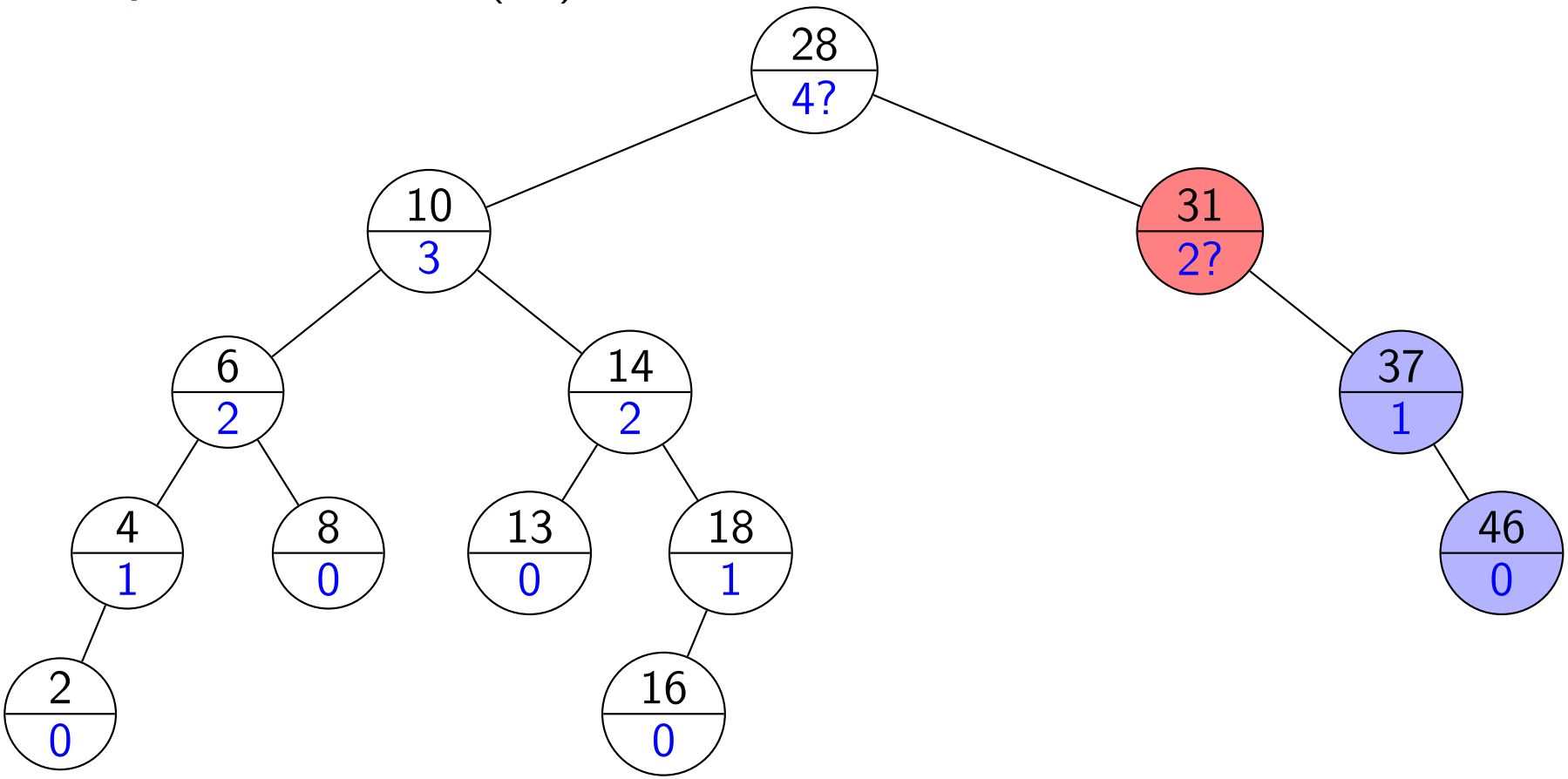
AVL trees deletion: Example

Example: *AVL::delete*(22)



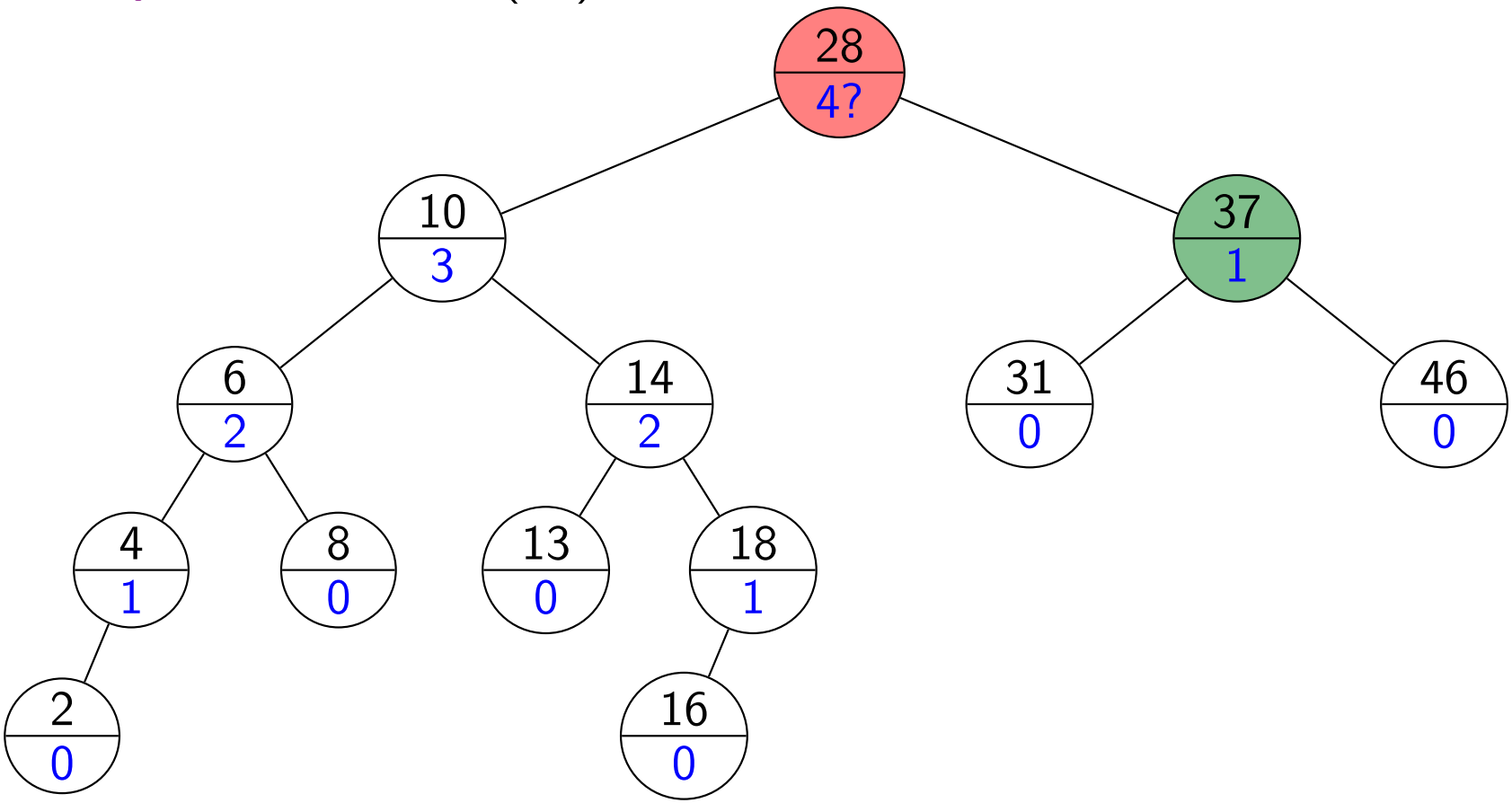
AVL trees deletion: Example

Example: *AVL::delete*(22)



AVL trees deletion: Example

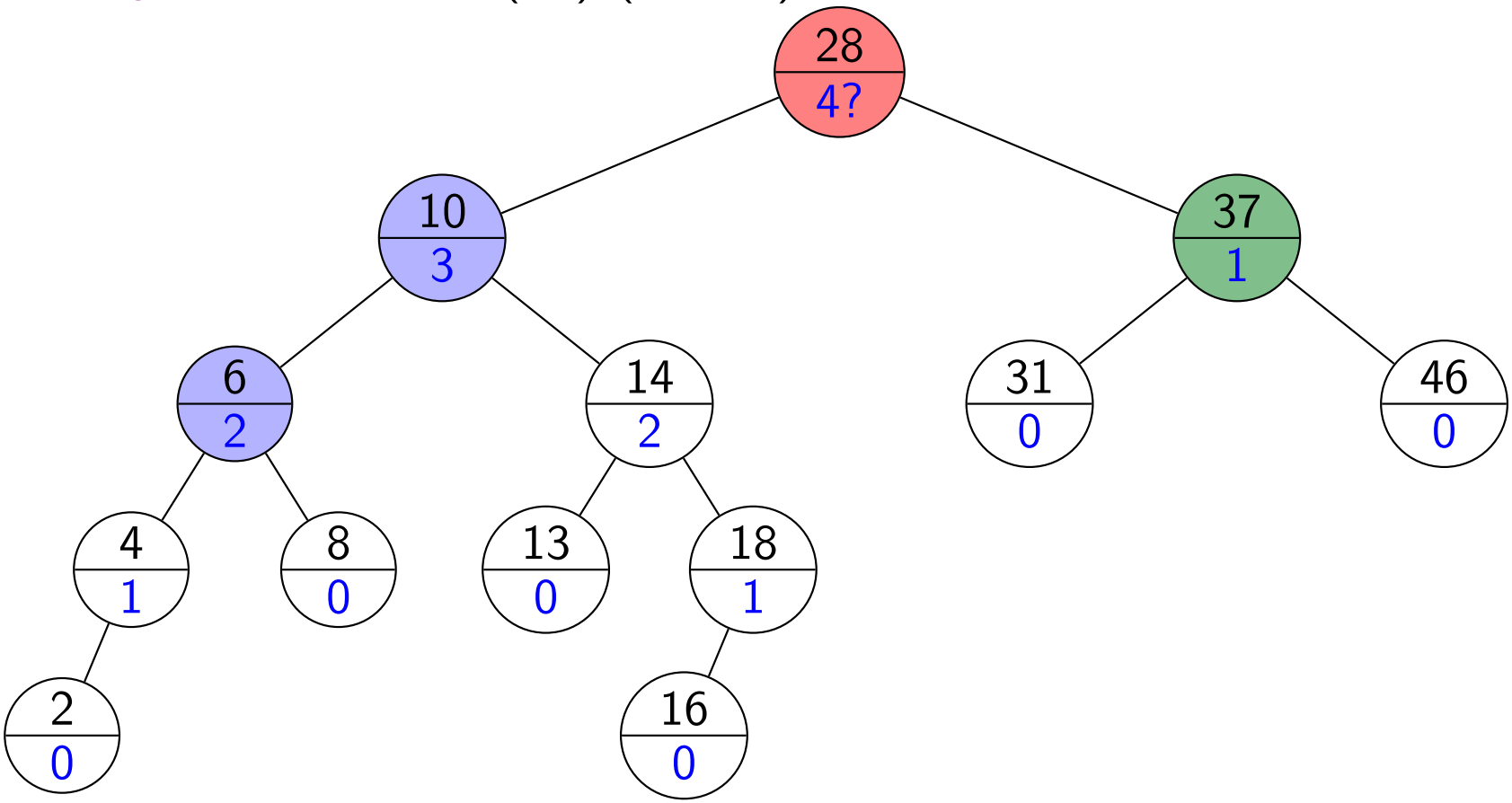
Example: *AVL::delete*(22)



A single *restructure* is not enough to restore all balances.

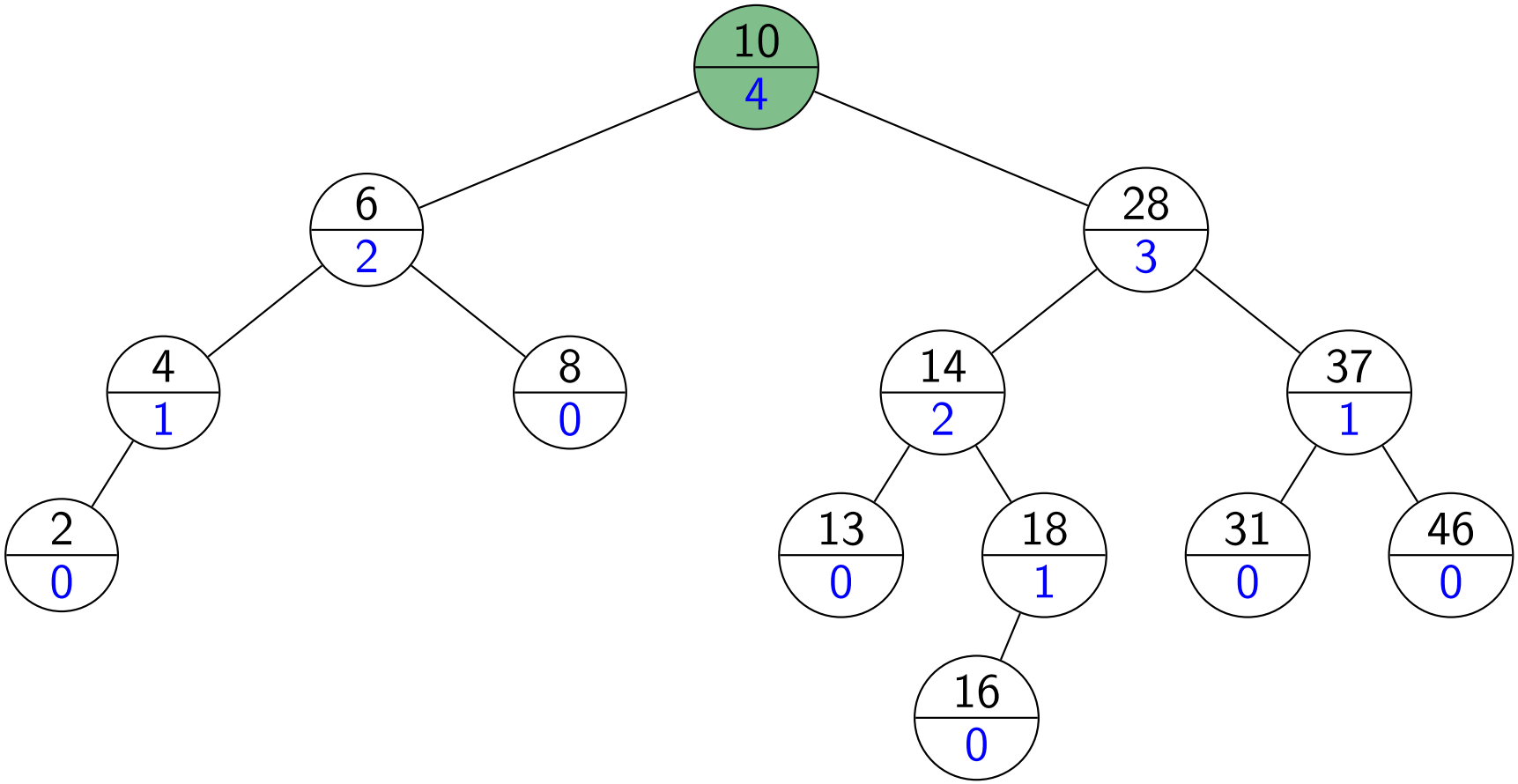
AVL trees deletion: Example

Example: *AVL::delete*(22) (cont'd)



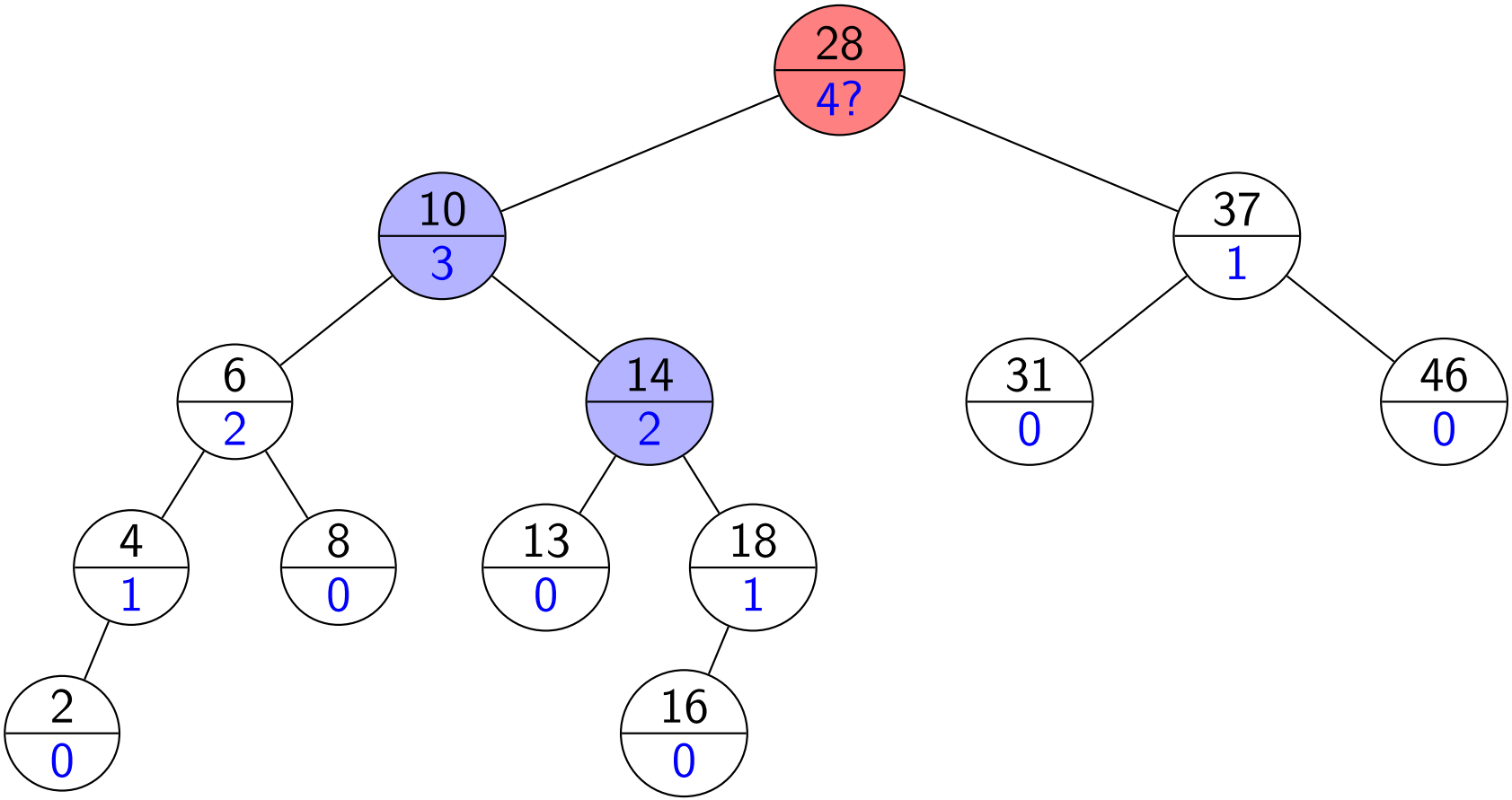
AVL trees deletion: Example

Example: *AVL::delete*(22) (cont'd)



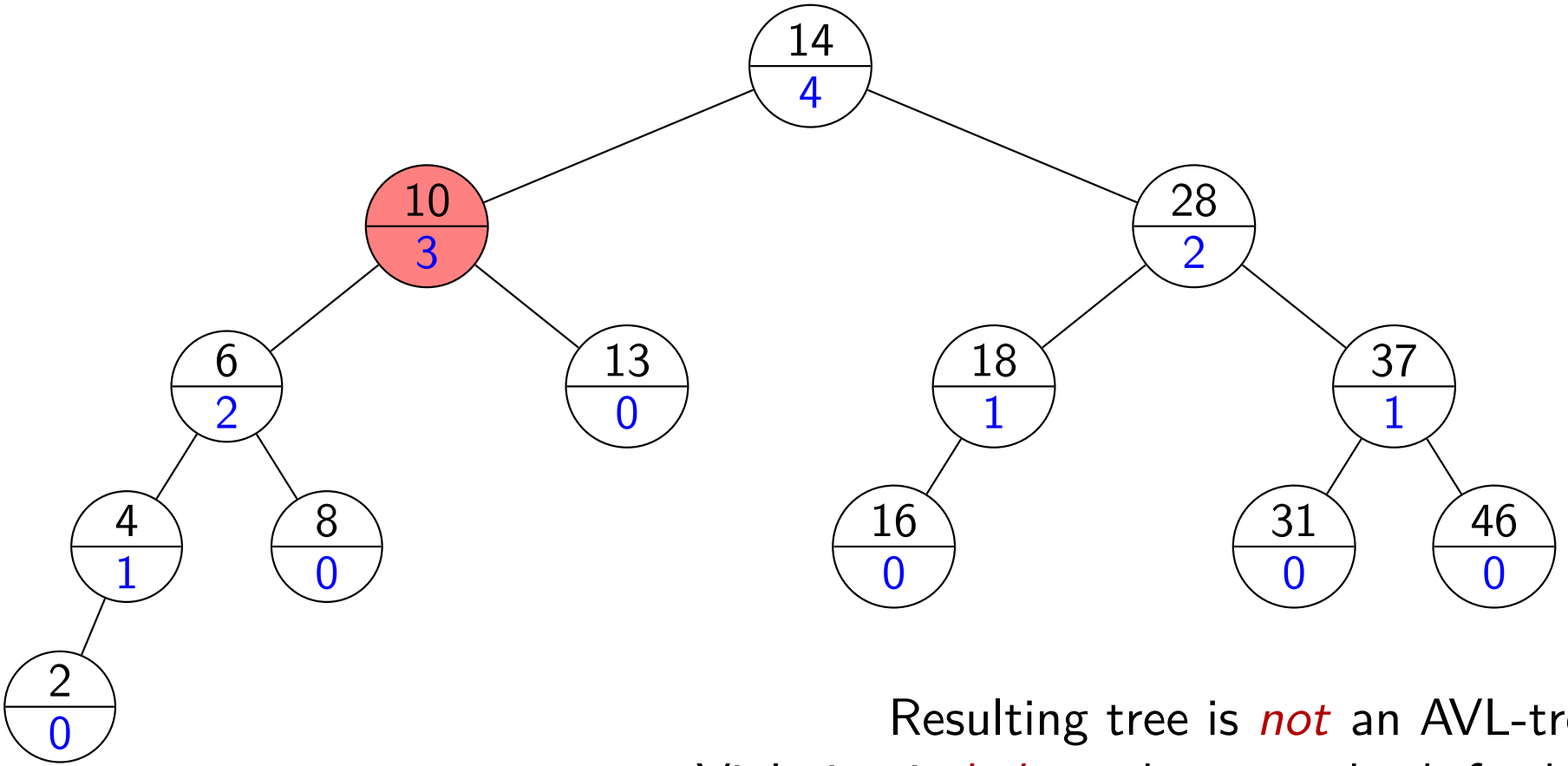
AVL trees deletion: Example

Important: Ties *must* be broken to avoid double rotation.
Consider again the above example. If we applied double-rotation:



AVL trees deletion: Example

Important: Ties *must* be broken to avoid double rotation.
Consider again the above example. If we applied double-rotation:



Resulting tree is *not* an AVL-tree.
Violation is *below* where we check further.

AVL trees: Summary

search: Just like in BSTs, costs $\Theta(\mathit{height})$

insert: *BST::insert*, then check & update along path to new leaf

- total cost $\Theta(\mathit{height})$
- *restructure* will be called *at most once*.

delete: *BST::delete*, then check & update along path to deleted node

- total cost $\Theta(\mathit{height})$
- *restructure* may be called $\Theta(\mathit{height})$ times.

Worst-case cost for all operations is $\Theta(\mathit{height}) = \Theta(\log n)$.

- In practice, the constant is quite large.
- Other realizations of ADT Dictionary are better in practice ($\rightarrow$ later)