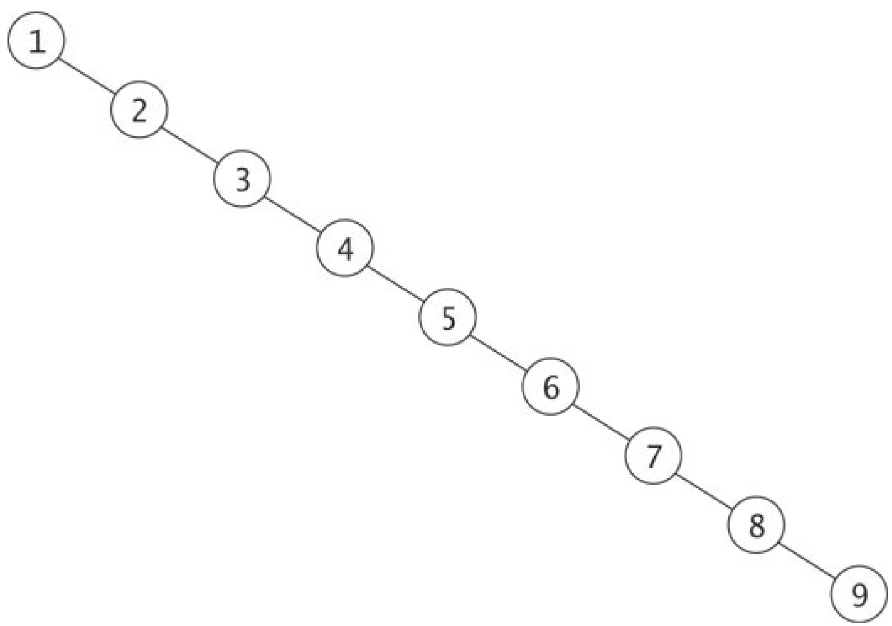
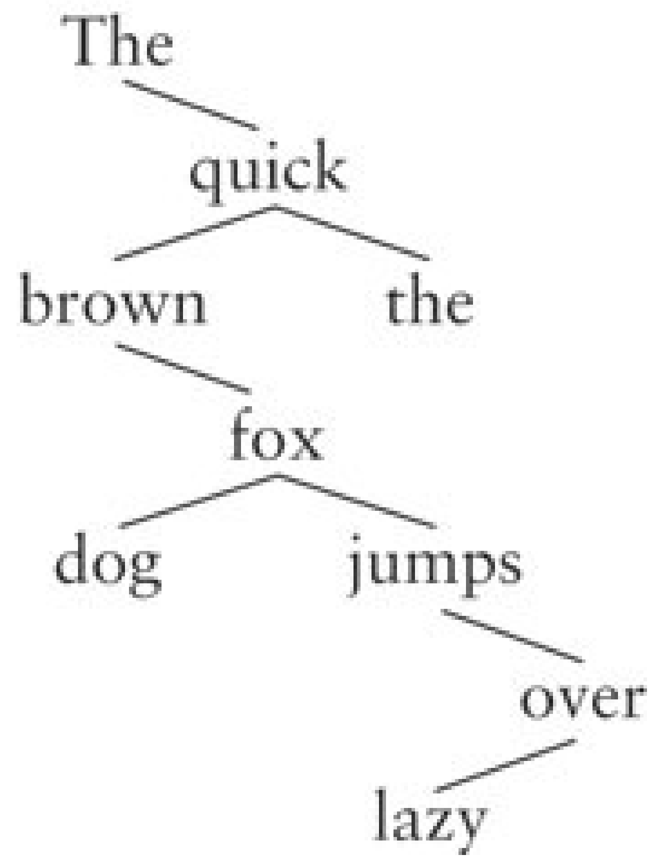
# AVL trees
(Weiss 4.4)

# Balanced BSTs: the problem

The BST operations take O(height of tree), so for unbalanced trees can take O(n) time



# Balanced BSTs: the solution

Take BSTs and add an extra invariant that makes sure that the tree is balanced

- Height of tree must be O(log n)
- Then all operations will take O(log n) time

One possible idea for an invariant:

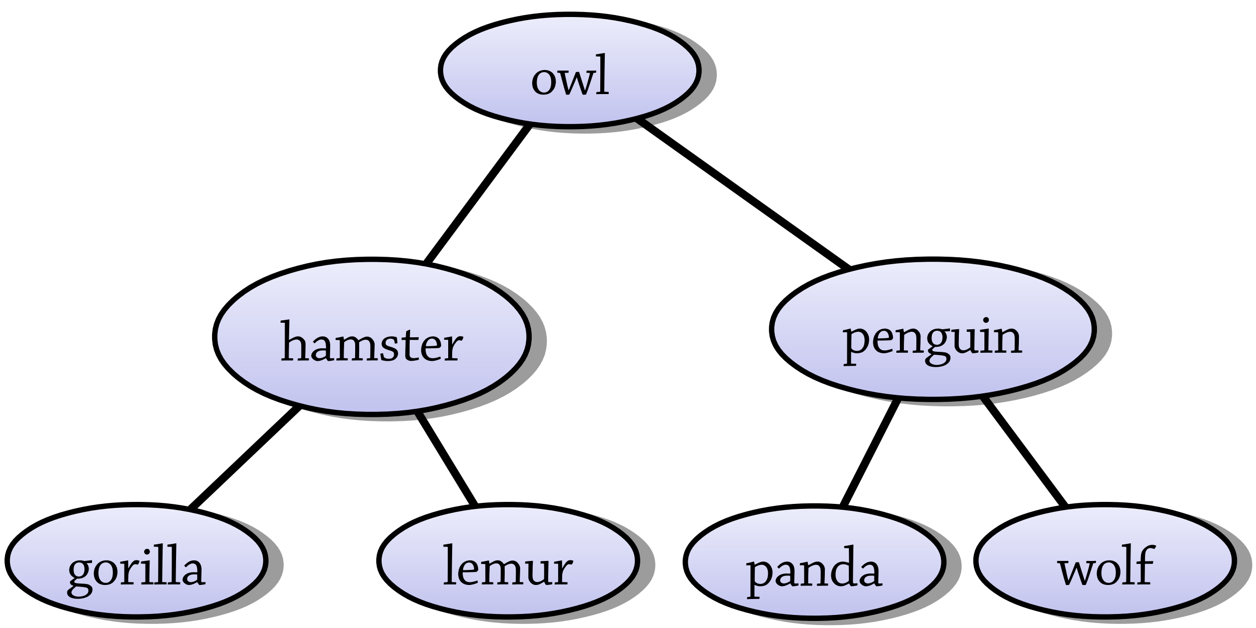
- Height of left child = height of right child (for all nodes in the tree)
- Tree would be sort of "perfectly balanced"

What's wrong with this idea?

# A too restrictive invariant

Perfect balance is too restrictive!

Number of nodes can only be 1, 3, 7, 15, 31, ...



# AVL trees – a less restrictive invariant

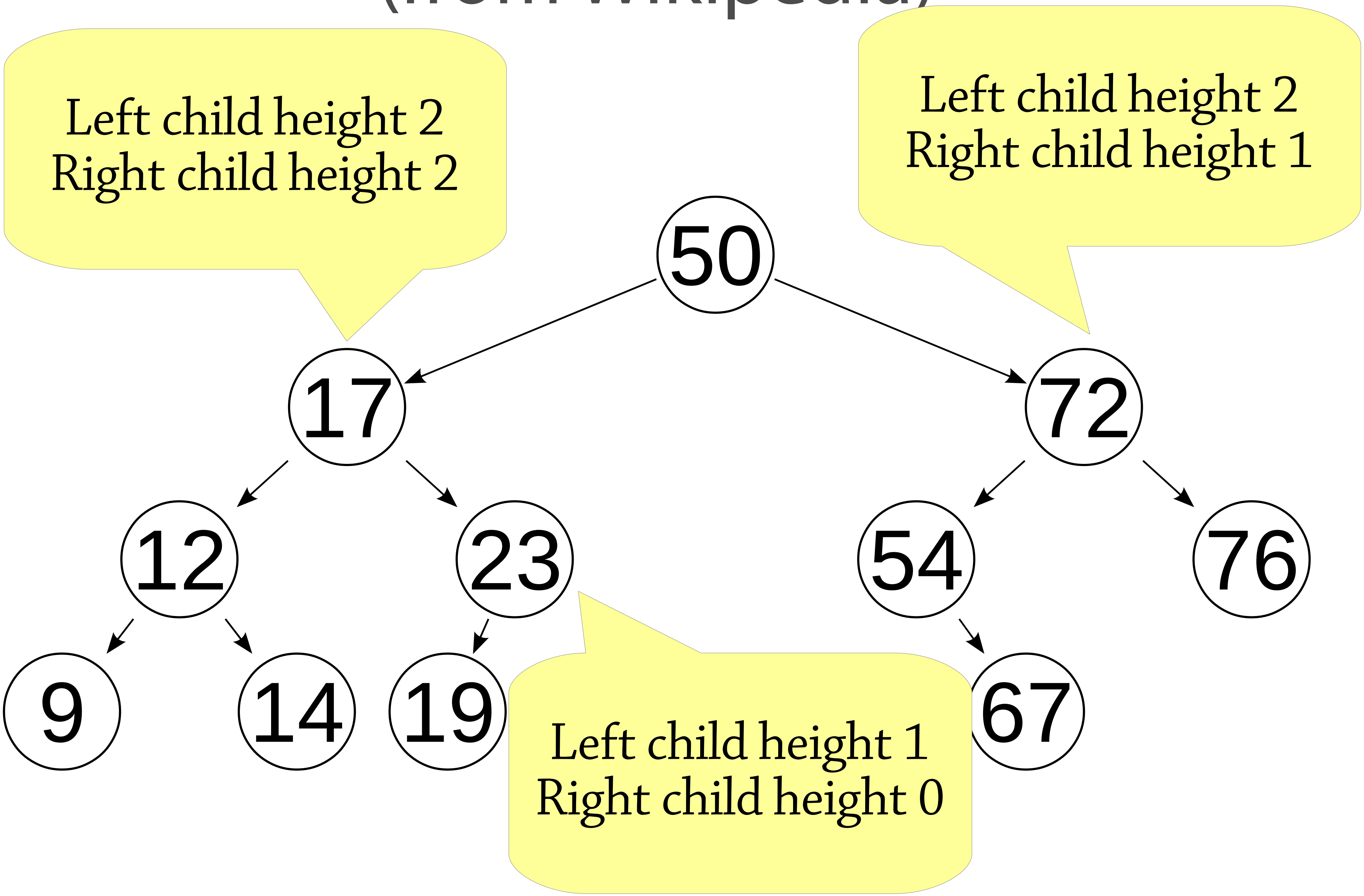
The AVL tree is the first balanced BST discovered (from 1962) – it's named after Adelson-Velsky and Landis

It's a BST with the following invariant:

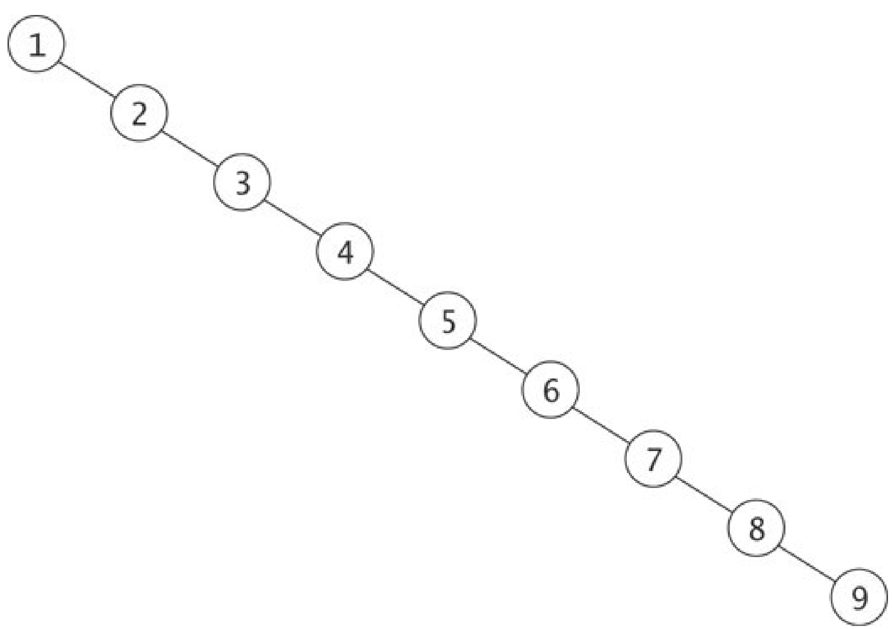
- The *difference in heights* between the left and right children of any node is at most 1
- (compared to 0 for a perfectly balanced tree)

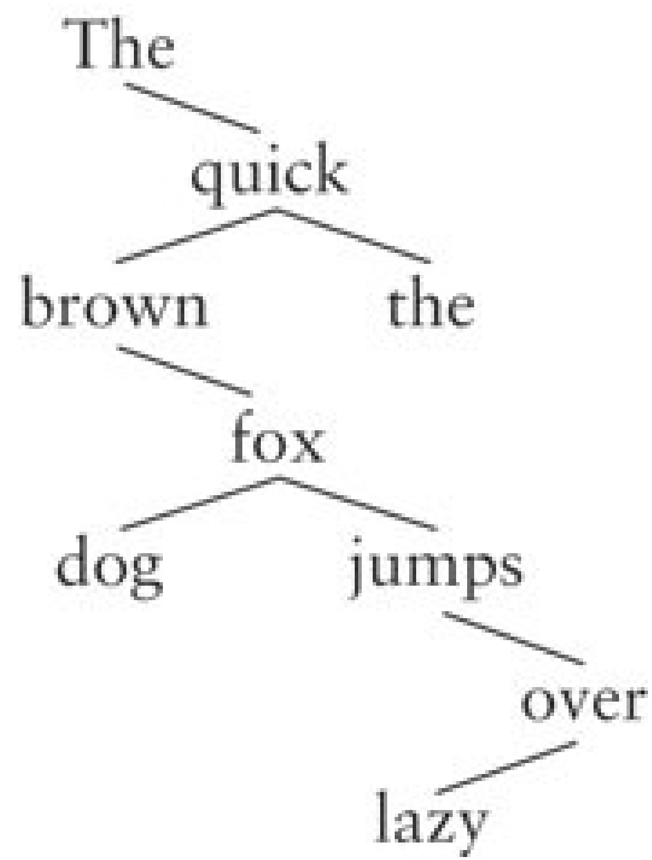
This makes the tree's height O(log n), so it's balanced

# Example of an AVL tree (from Wikipedia)



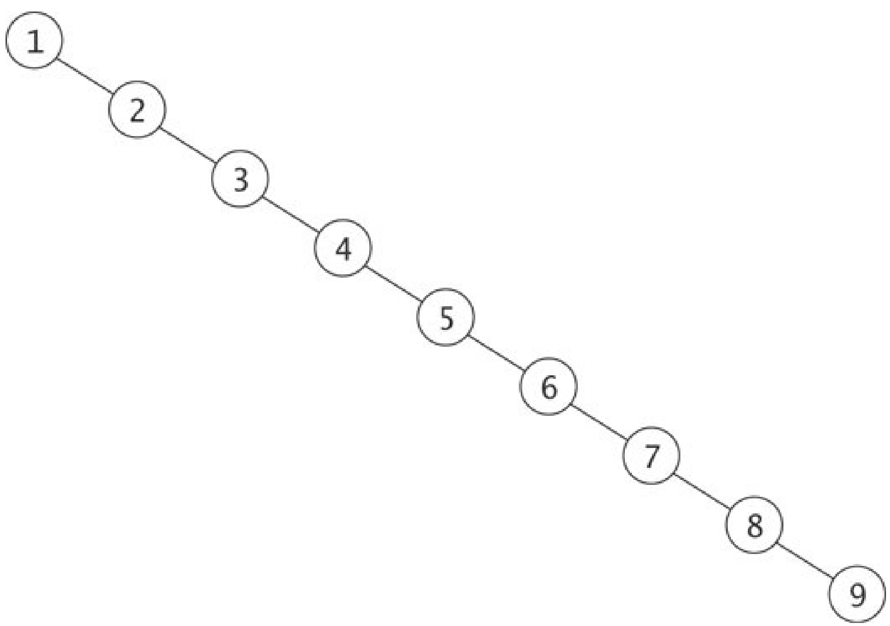
# Why are these not AVL trees?

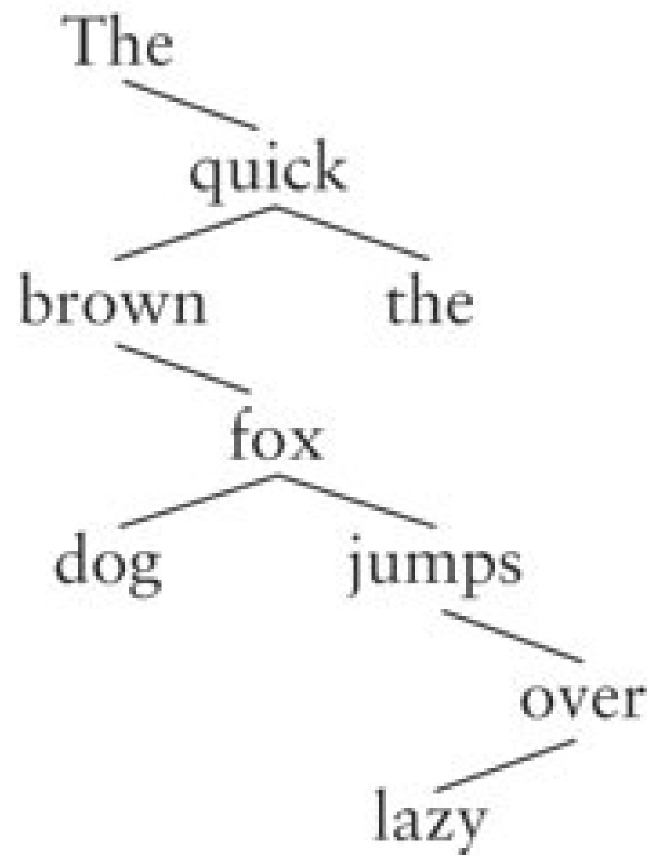


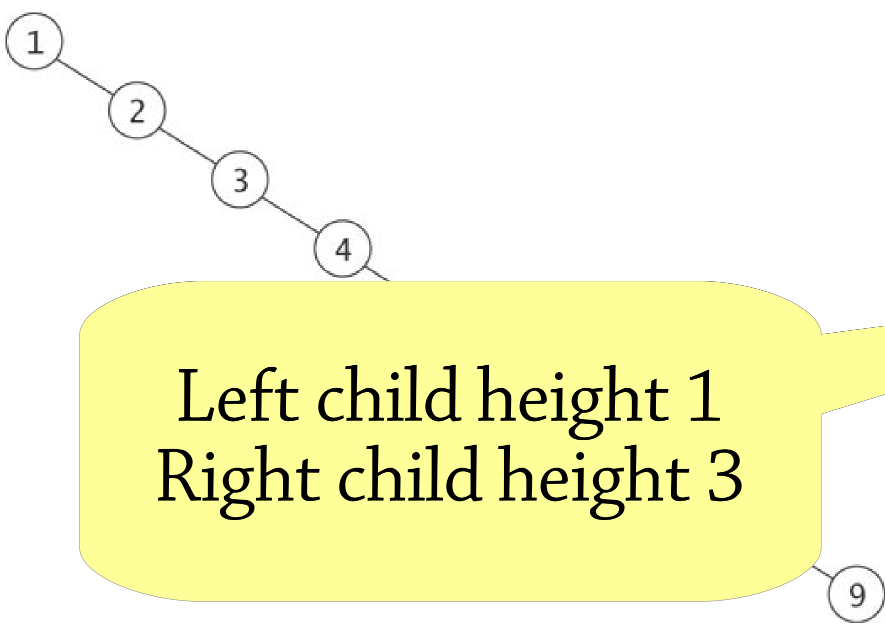
# Why are these not AVL trees?

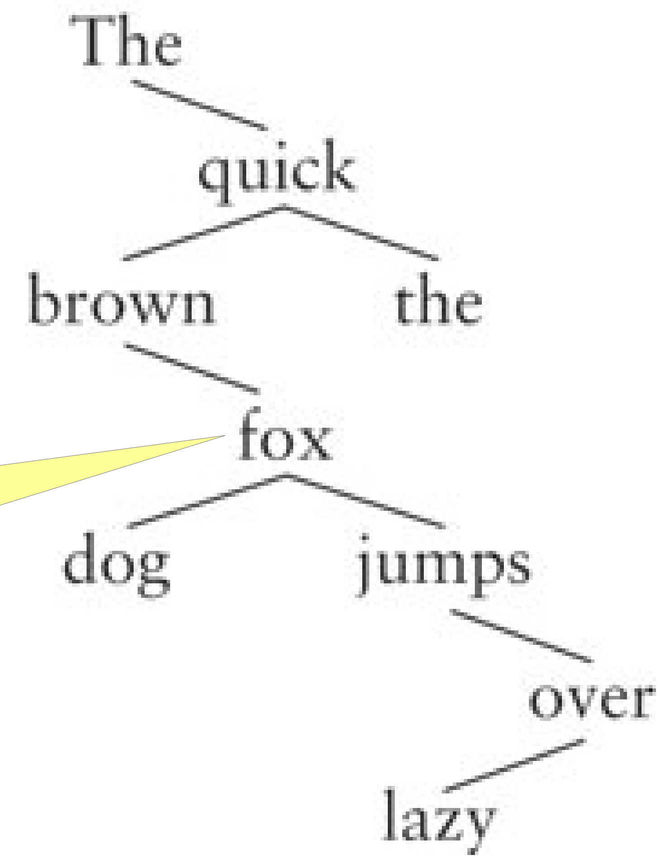
Left child height 0
Right child height 8



# Why are these not AVL trees?

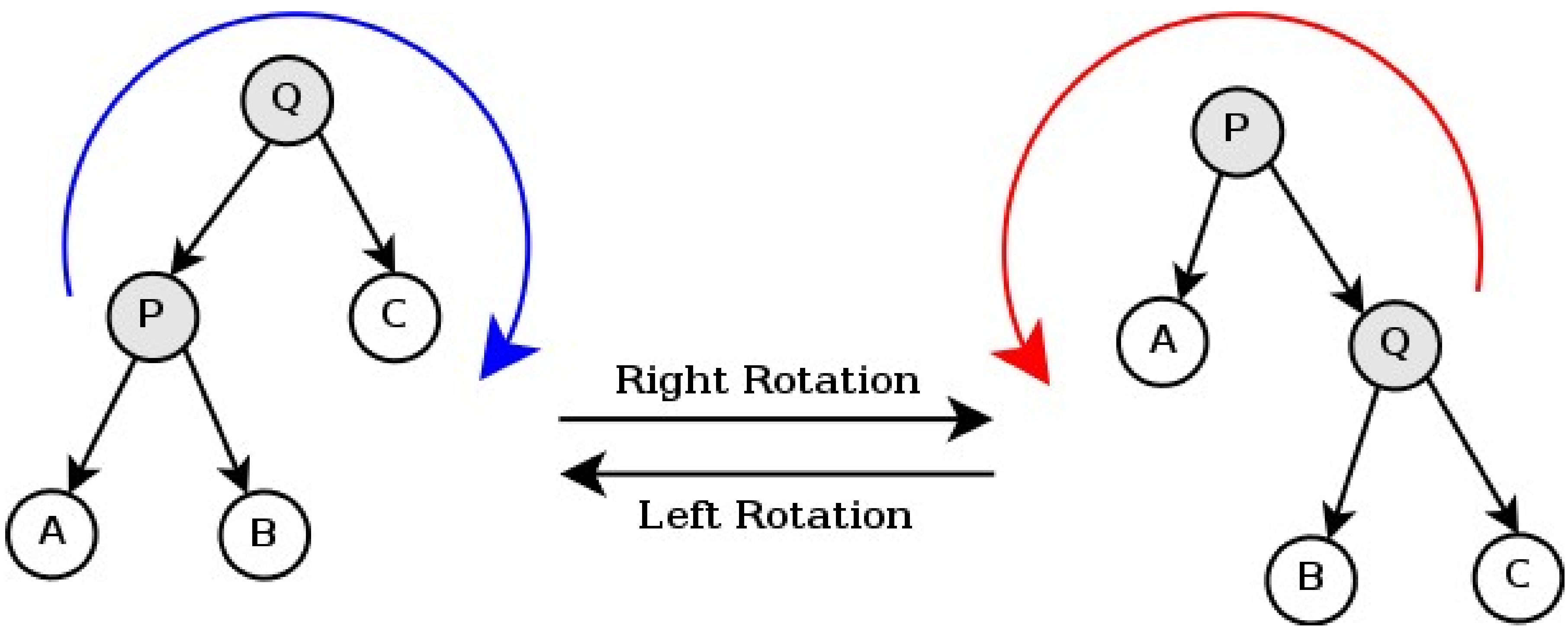



Left child height 1
Right child height 3

# Rotation

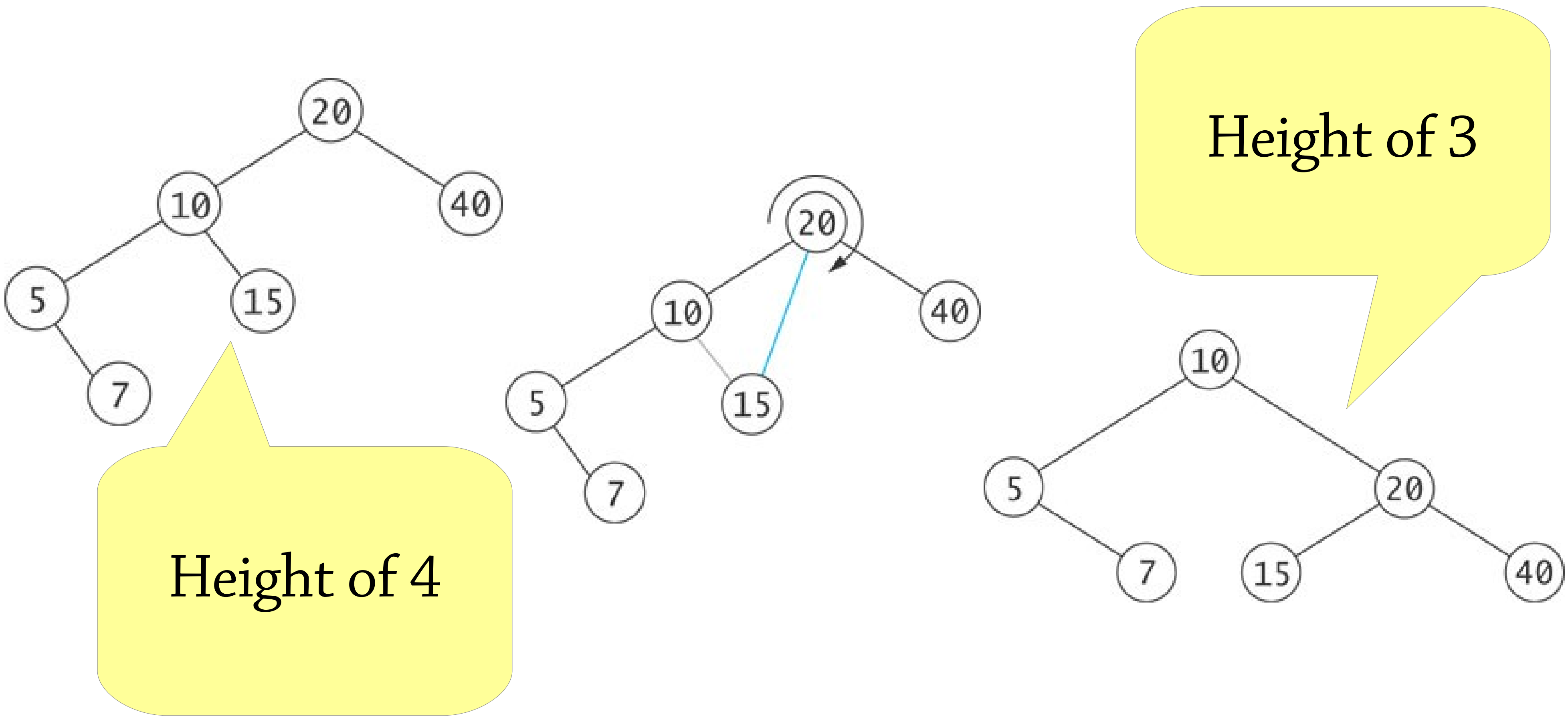
Rotation rearranges a BST by moving a different node to the root, without changing the BST's contents




(pic from Wikipedia)

# Rotation

We can strategically use rotations to rebalance an unbalanced tree.
This is what most balanced BST variants do!



# AVL insertion

Start by doing a BST insertion

- This might break the AVL (balance) invariant

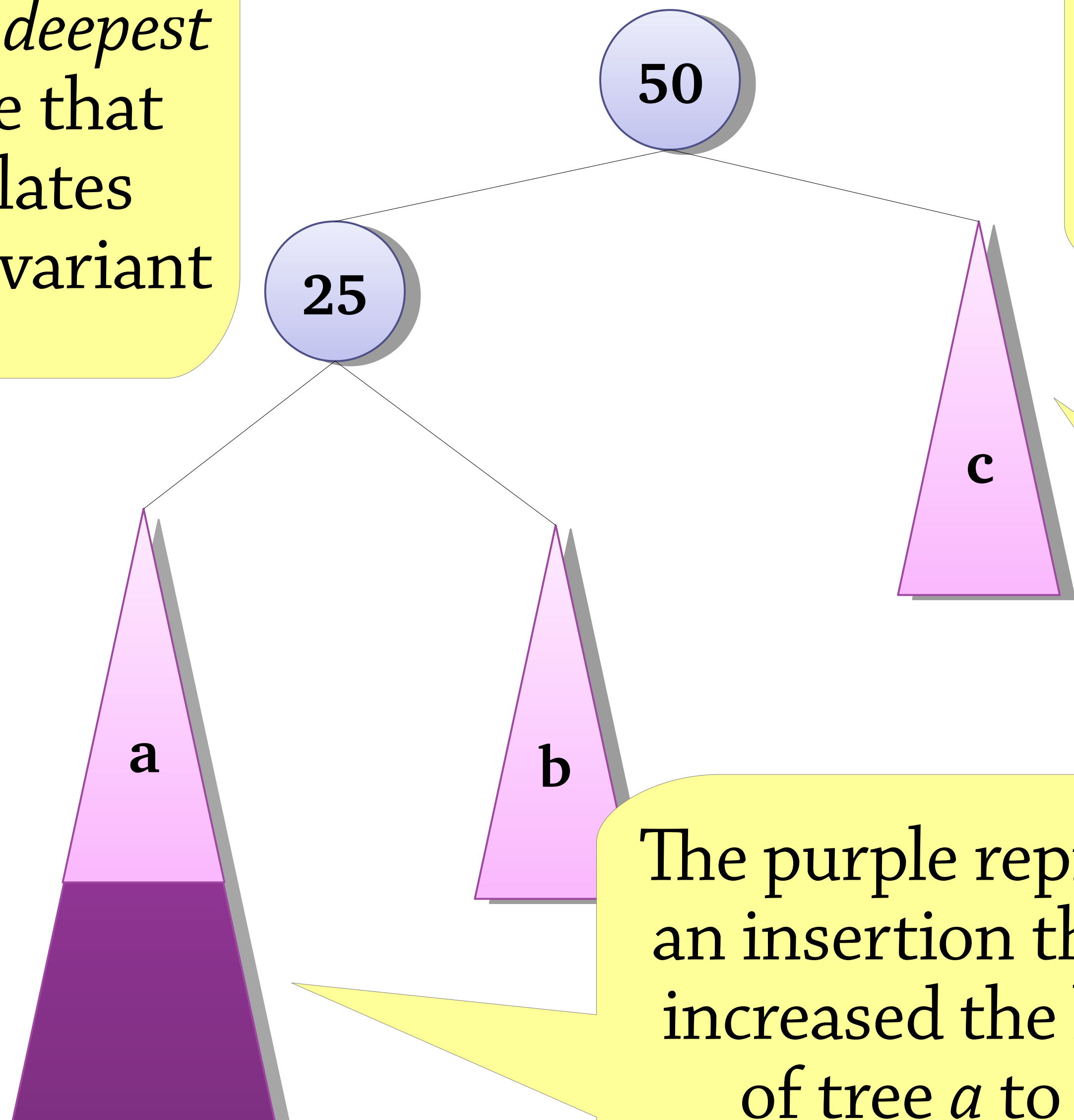
Then go upwards from the newly-inserted node, looking for nodes that break the invariant (unbalanced nodes)

If you find one, rotate it to fix the balance

There are four cases depending on *how* the node became unbalanced

# Case 1: a *left-left* tree

Assumption: this tree is the *deepest* node that violates the invariant

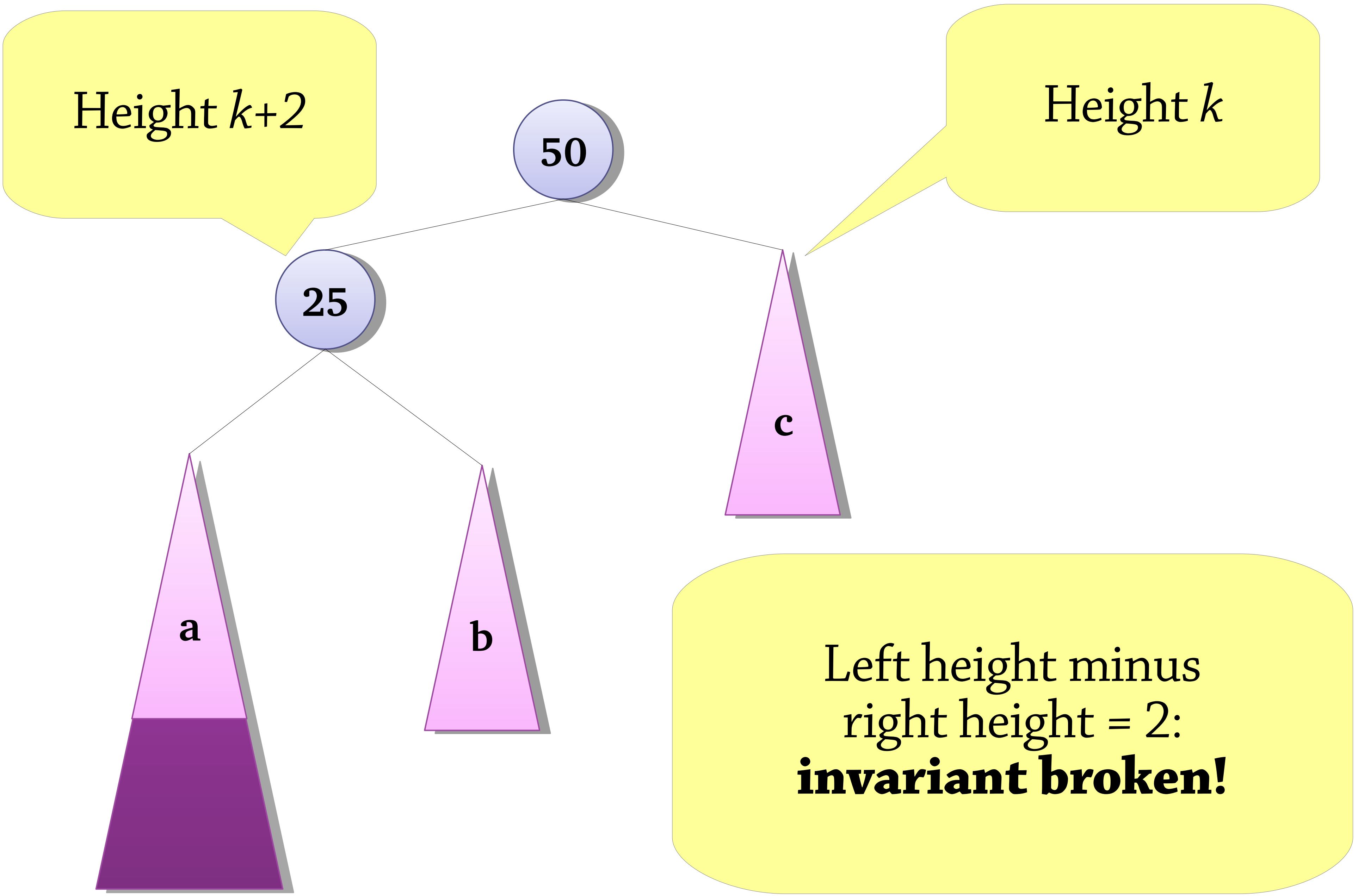



Notice that the tree was balanced before the purple bit was added

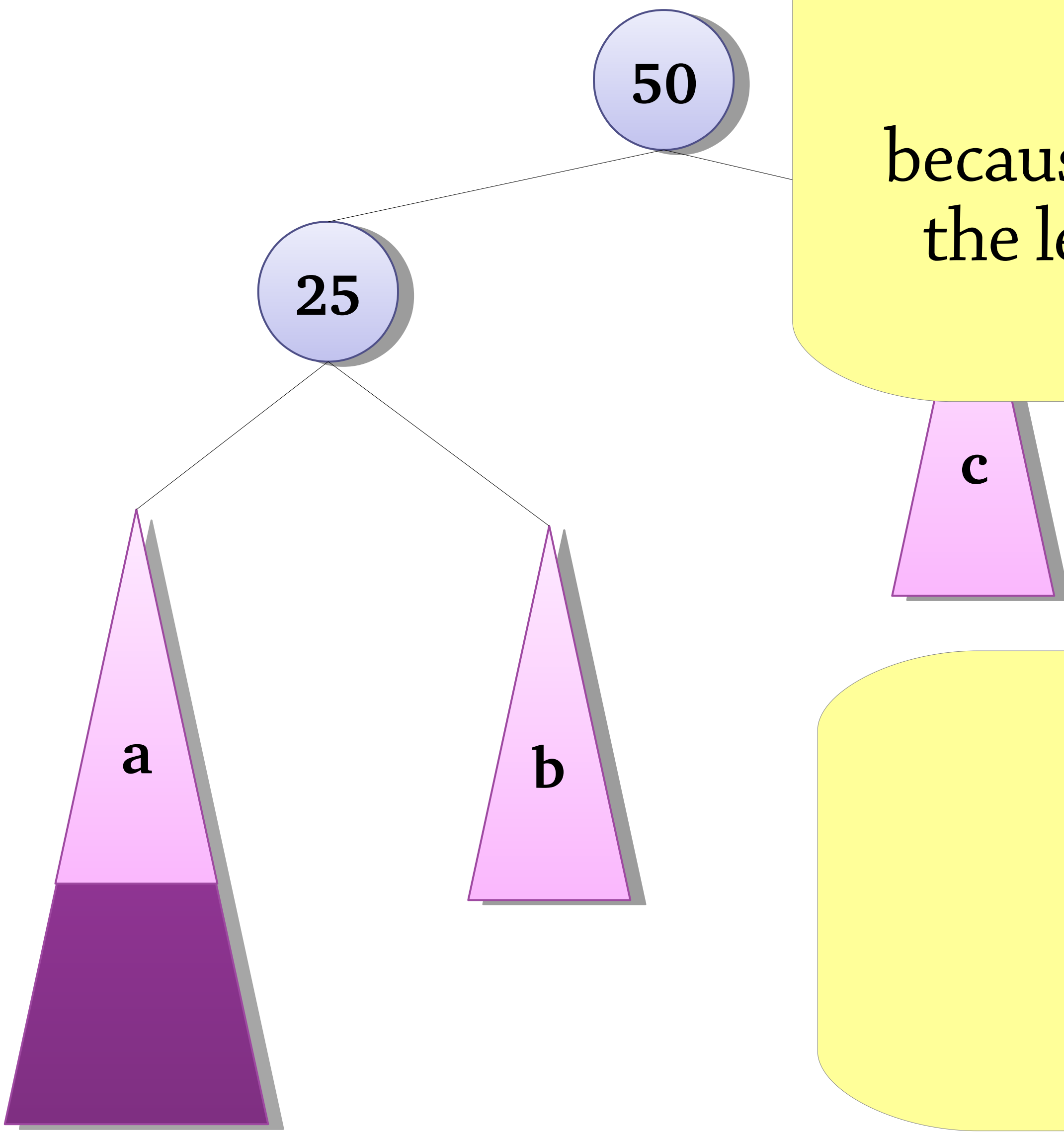
Each pink triangle represents an AVL tree with height *k*

The purple represents an insertion that has increased the height of tree *a* to *k+1*

# Case 1: a *left-left* tree



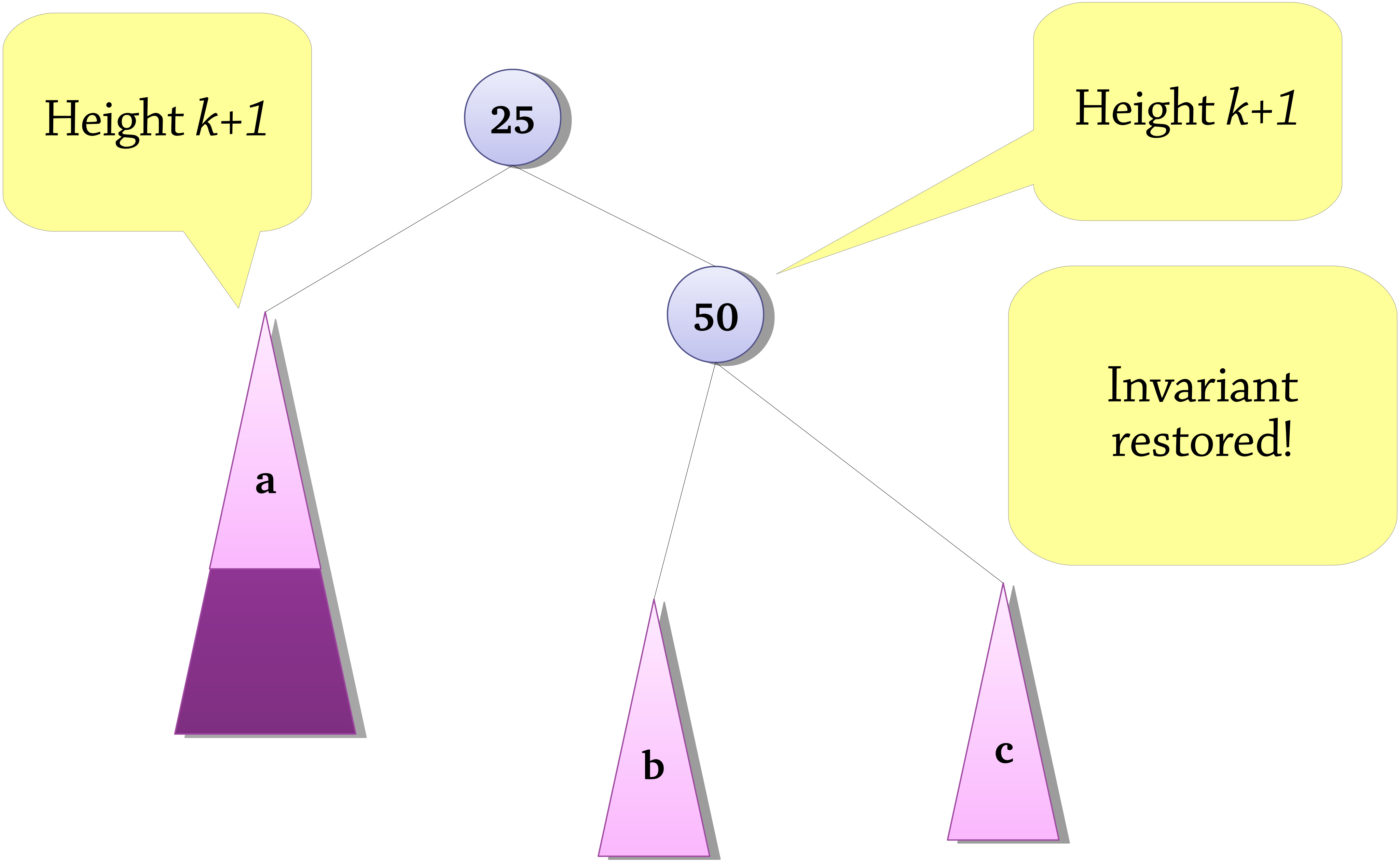
# Case 1: a *left-left* tree



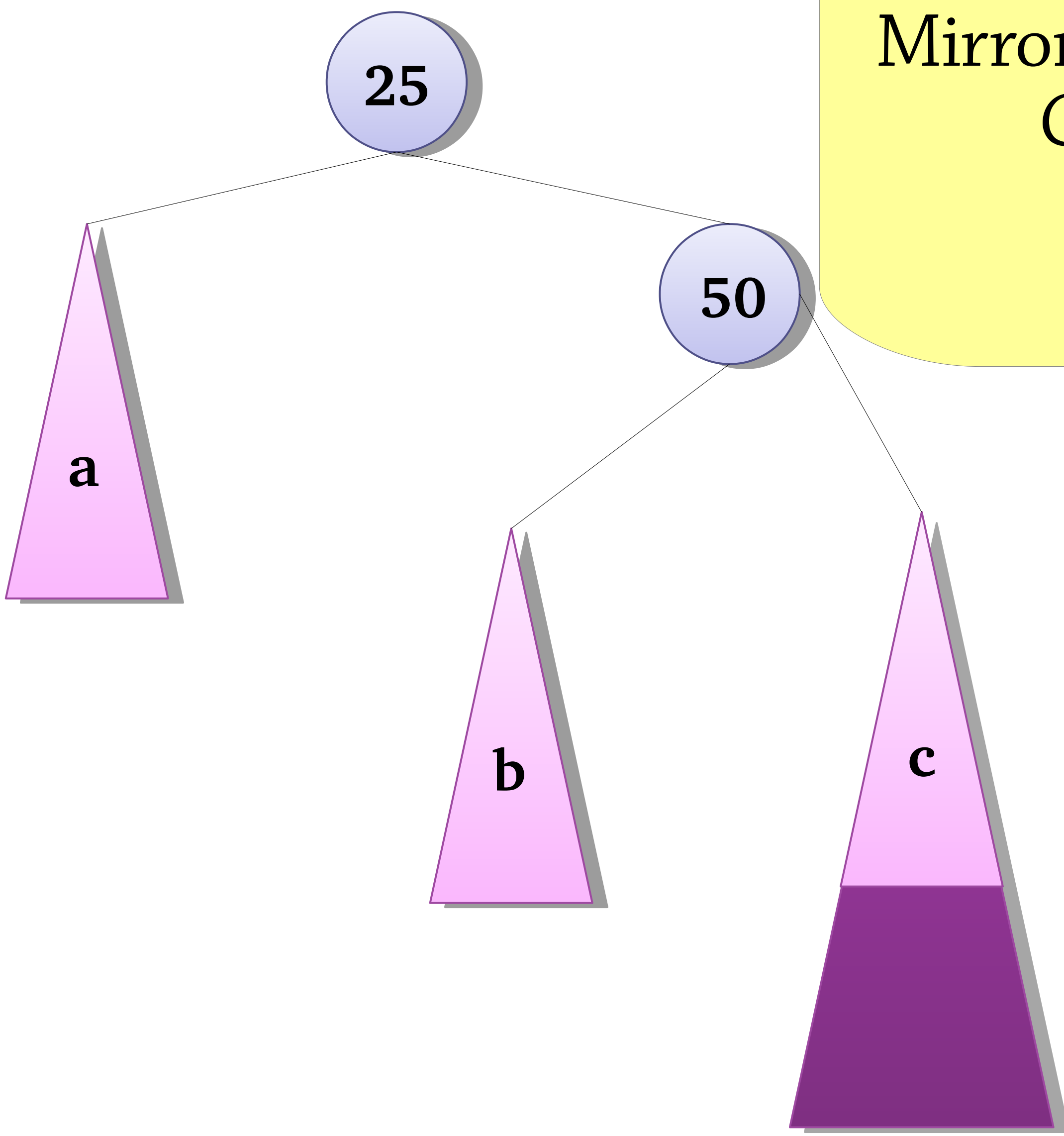

This is called a
*left-left tree*
because both the root and
the left child are deeper
on the left

To fix it we do a
*right rotation*

# Balancing a left-left tree, afterwards

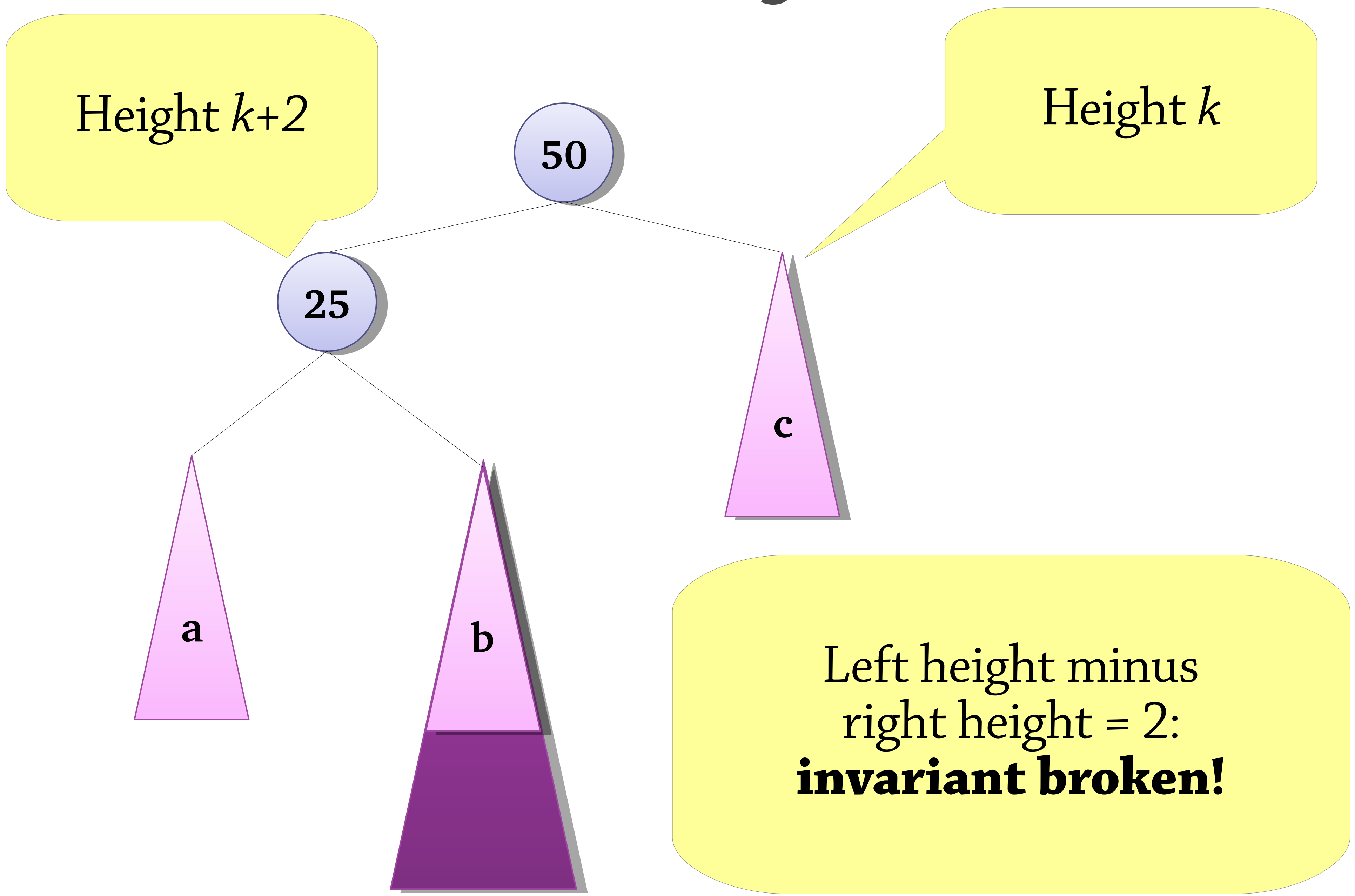


# Case 2: a *right-right* tree

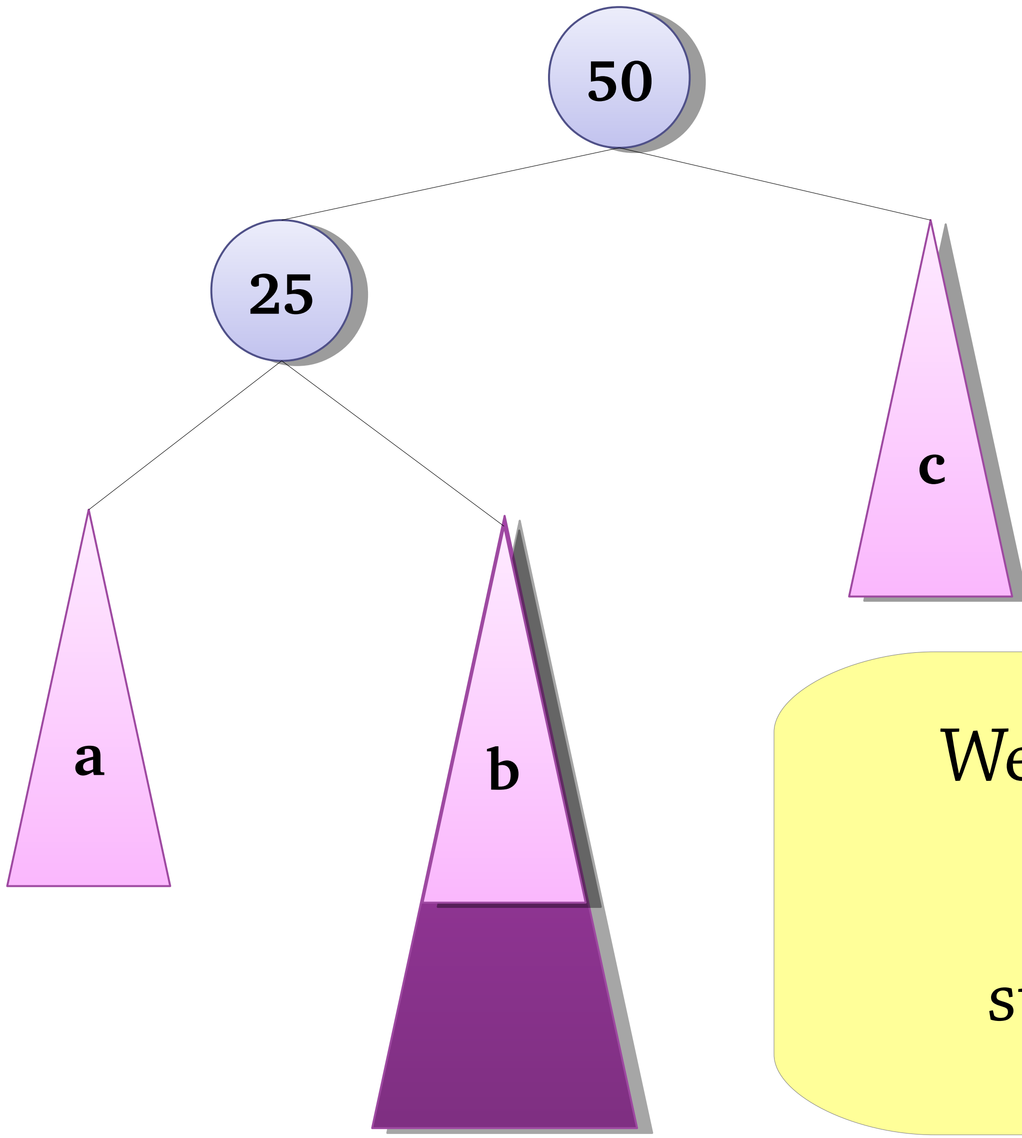



Mirror image of left-left tree
Can be fixed with
*left rotation*

# Case 3: a *left-right* tree

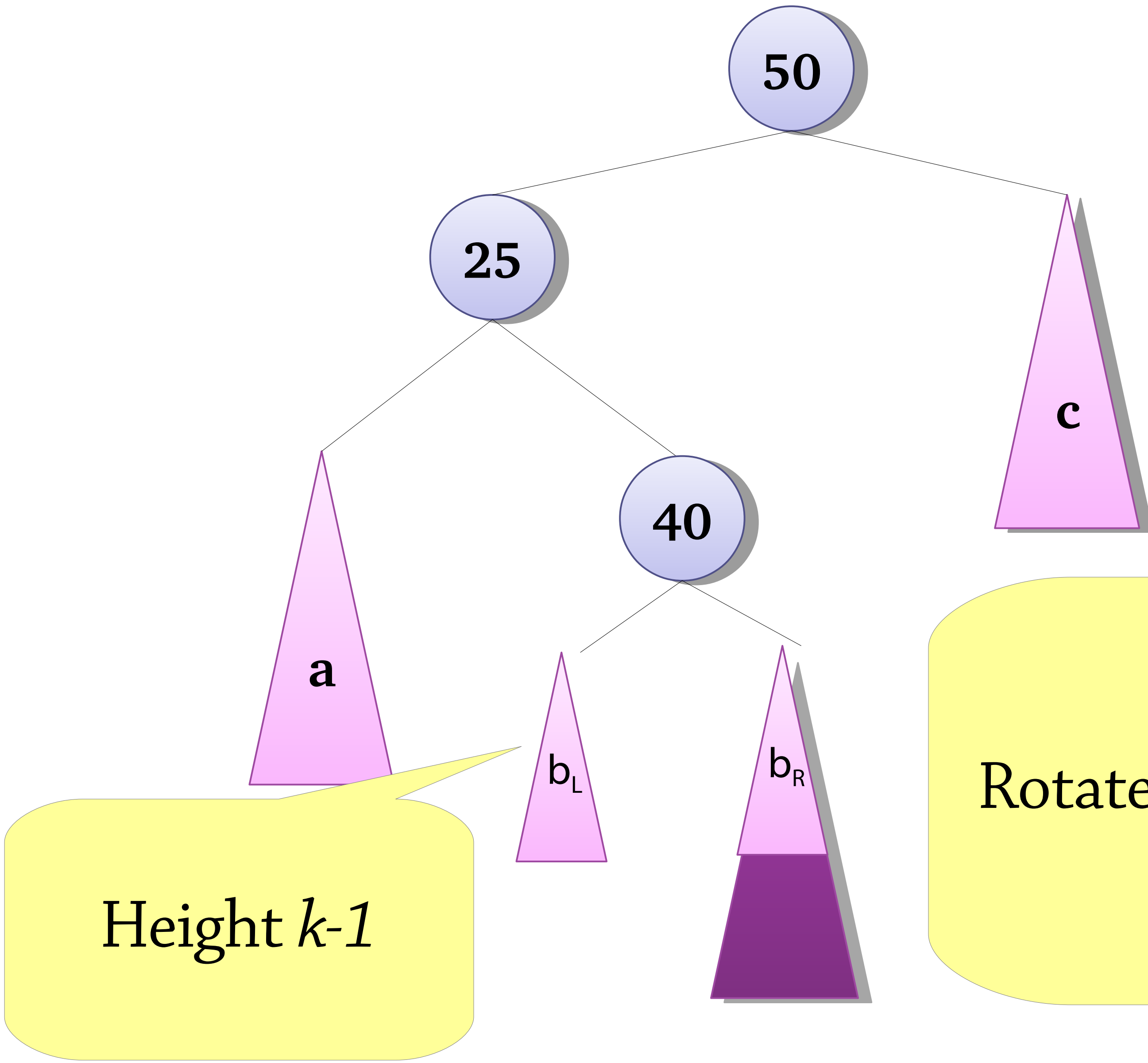


# Case 3: a *left-right* tree



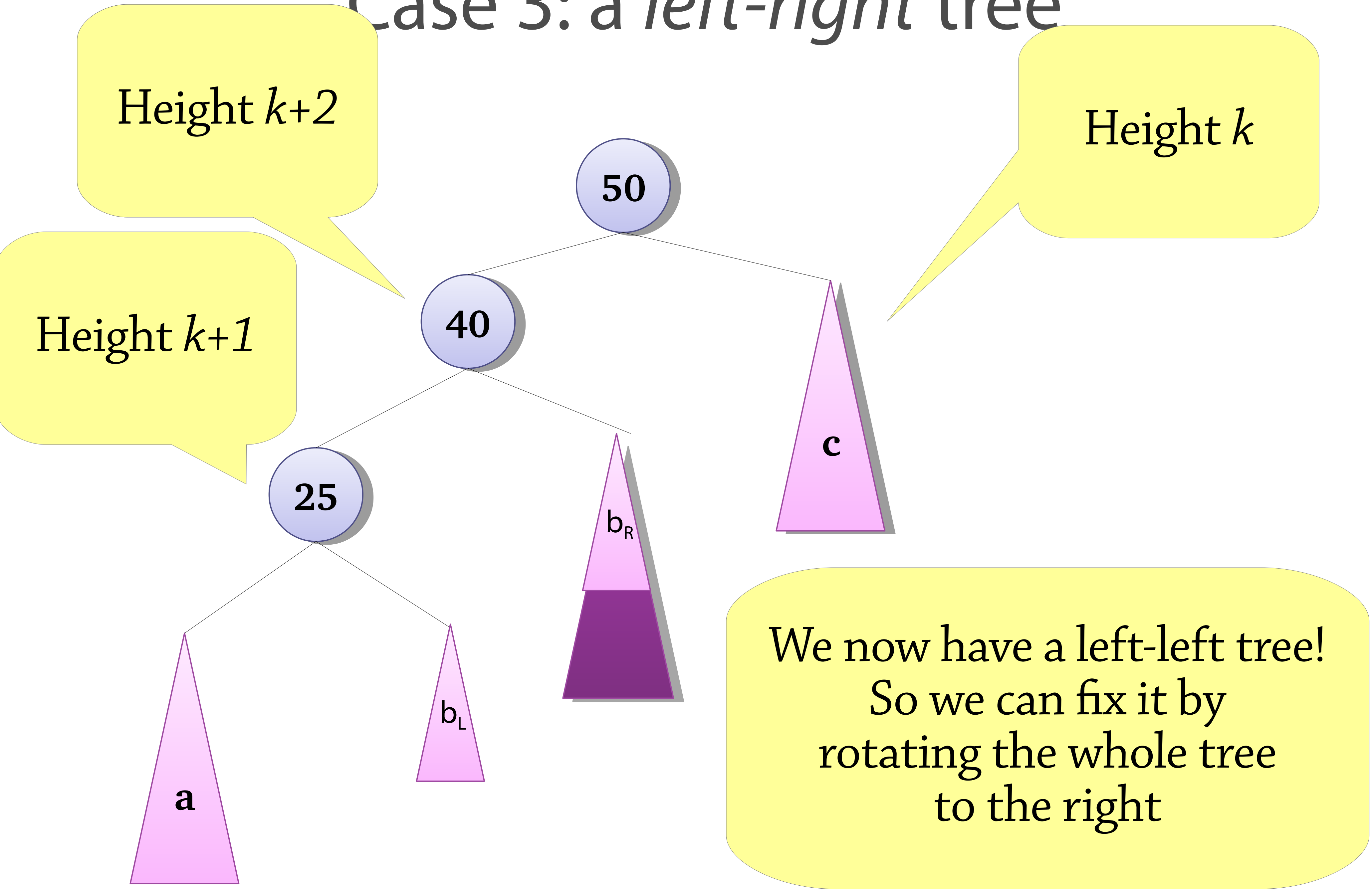

We can't fix this with one rotation
Let's look at b's subtrees $b_L$ and $b_R$

# Case 3: a *left-right* tree



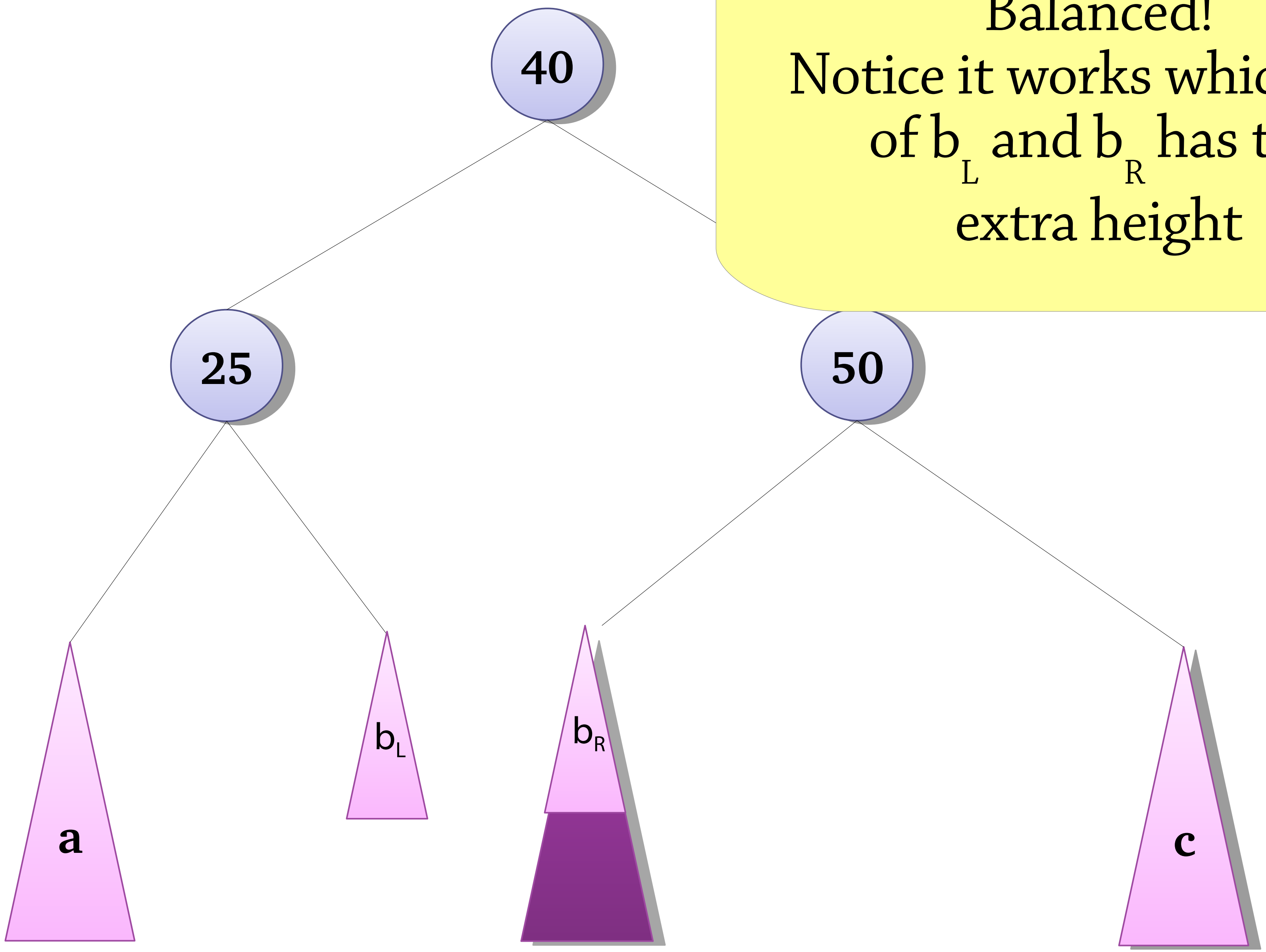

Rotate 25-subtree to the left

# Case 3: a *left-right* tree

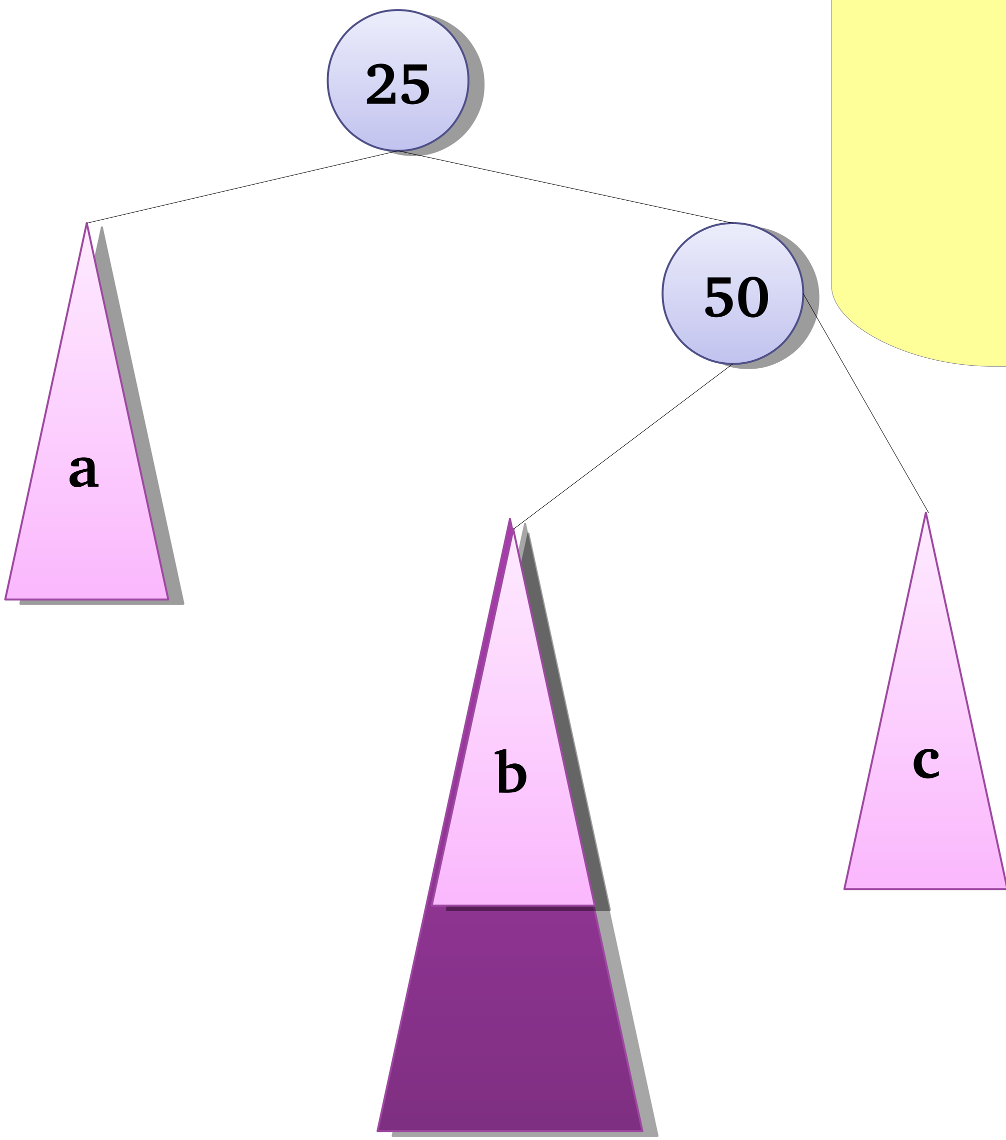


# Case 3: a *left-right* tree

Balanced!
Notice it works whichever of $b_L$ and $b_R$ has the extra height



# Case 4: a *right-left* tree




Mirror image of
left-right tree

# How to identify the cases

## Left-left (extra height in left-left grandchild):

- height of left-left grandchild = k+1
  height of left child = k+2
  height of right child = k
- Rotate the whole tree to the right

## Left-right (extra height in left-right grandchild):

- height of left-right grandchild = k+1
  height of left child = k+2
  height of right child = k
- First rotate the left child to the left
- Then rotate the whole tree to the right

Algorithm uses heights of subtrees to determine case

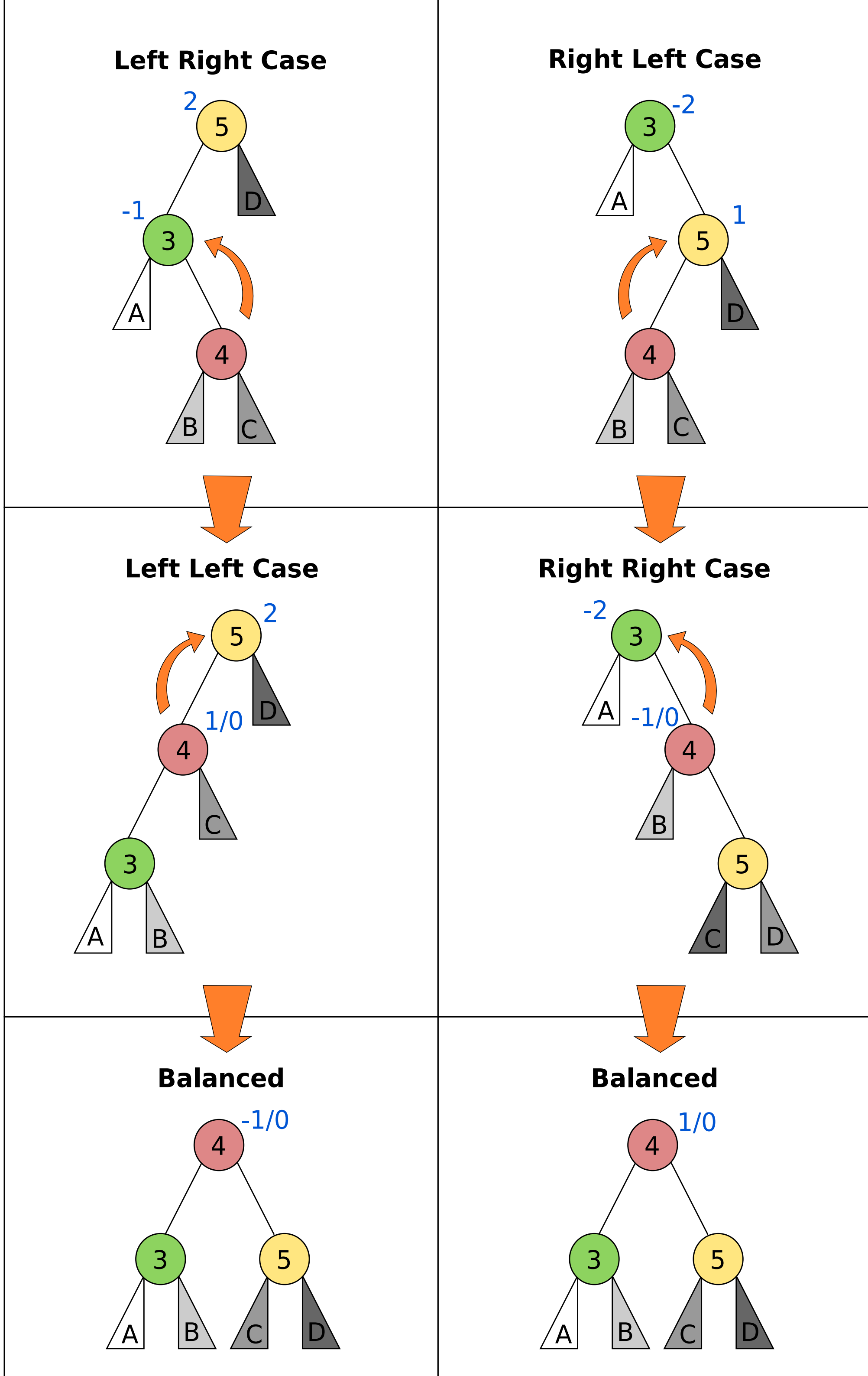
## Right-left and right-right: symmetric

# The four cases

(picture from Wikipedia)

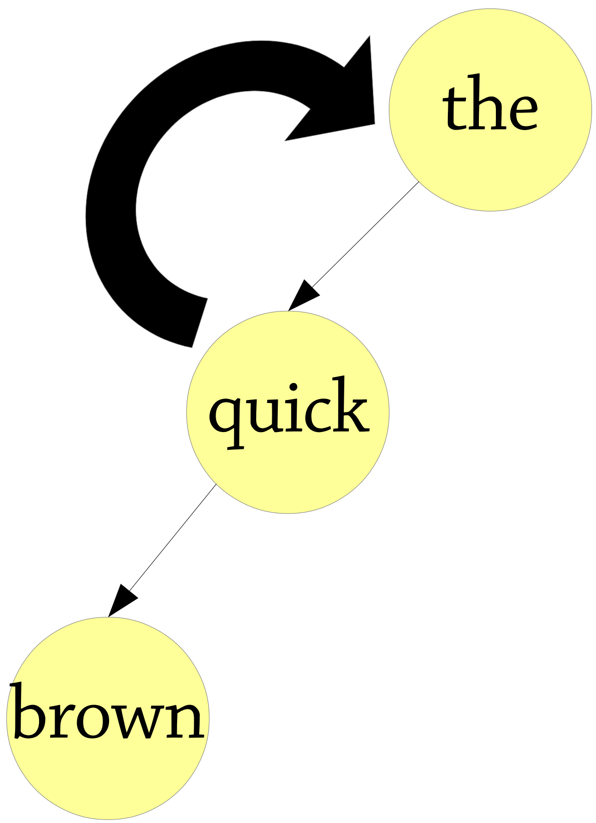
The numbers in the diagram show the *balance* of the tree: left height minus right height

To implement this efficiently, record the balance in the nodes and look at it to work out which case you're in



# Example: the quick brown fox jumps over a lazy dog

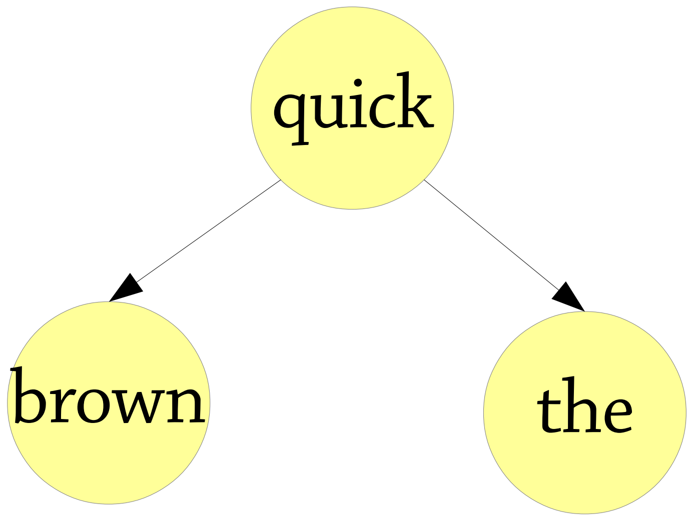
Insert "brown" into "the quick"




Left-left tree!
Rotate right

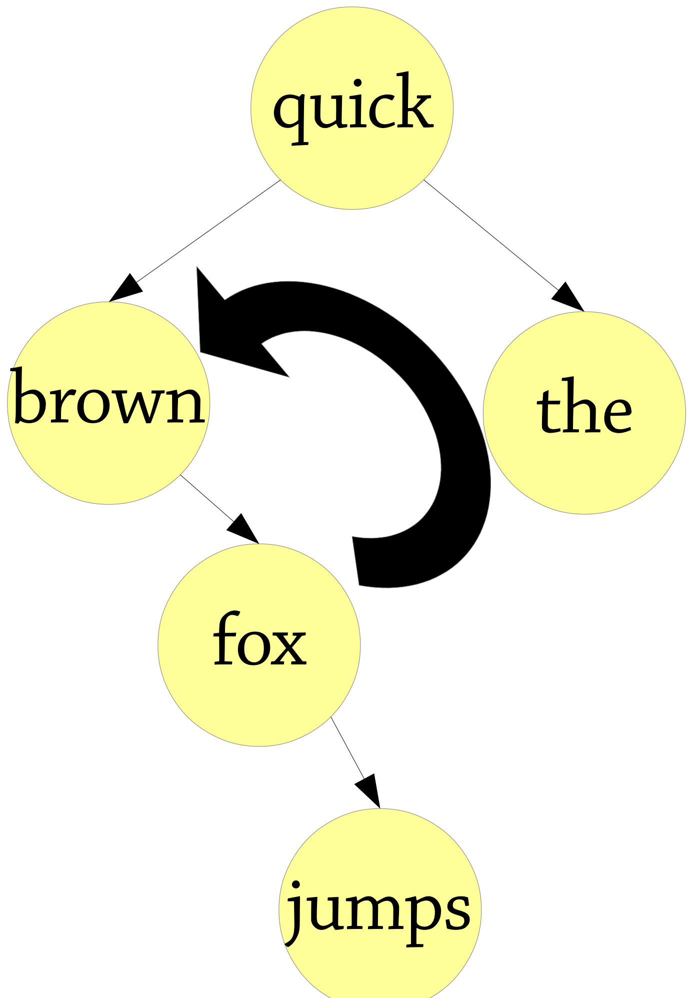
# Example: the quick brown fox jumps over a lazy dog

Insert “brown” into “the quick”



# Example: the quick brown fox jumps over a lazy dog

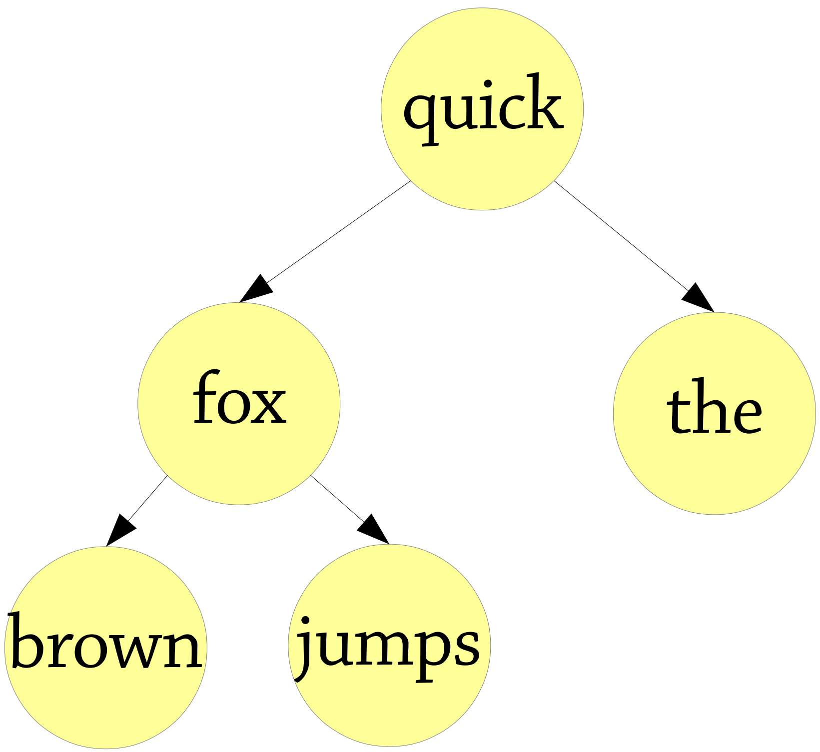
Insert "jumps" into "the quick brown fox"




Right-right tree!
(What node?)
Rotate left

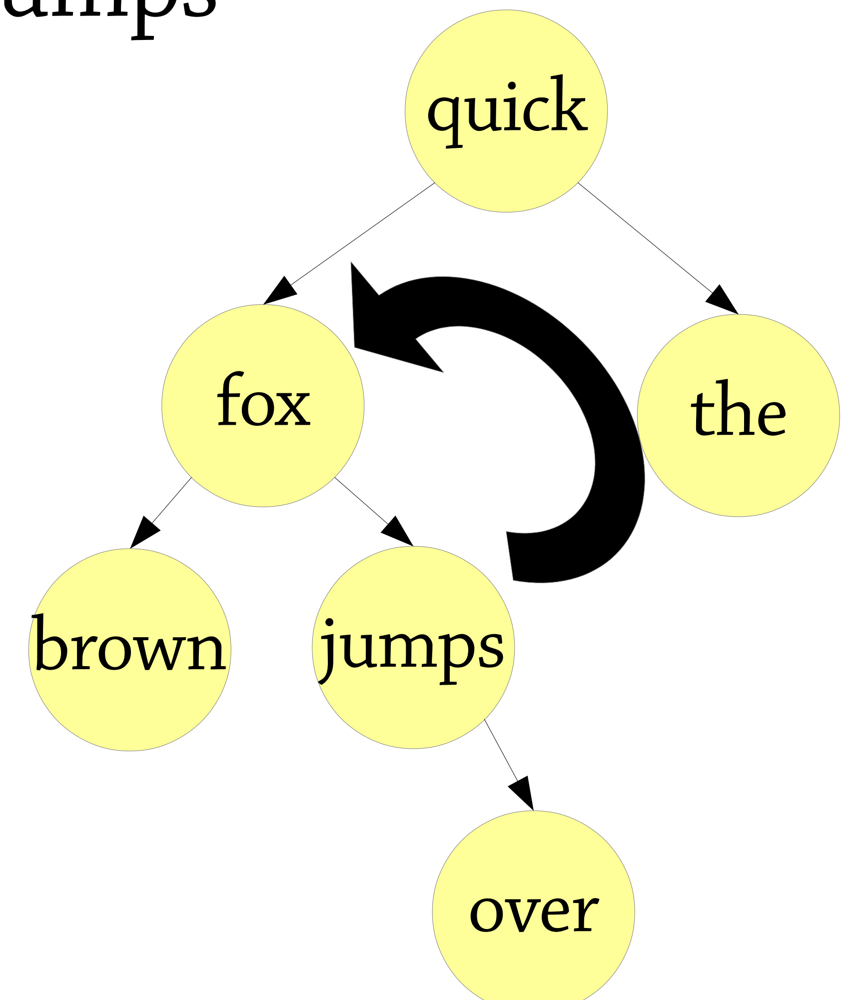
# Example: the quick brown fox jumps over a lazy dog

Insert "jumps" into "the quick brown fox"



# Example: the quick brown fox jumps over a lazy dog

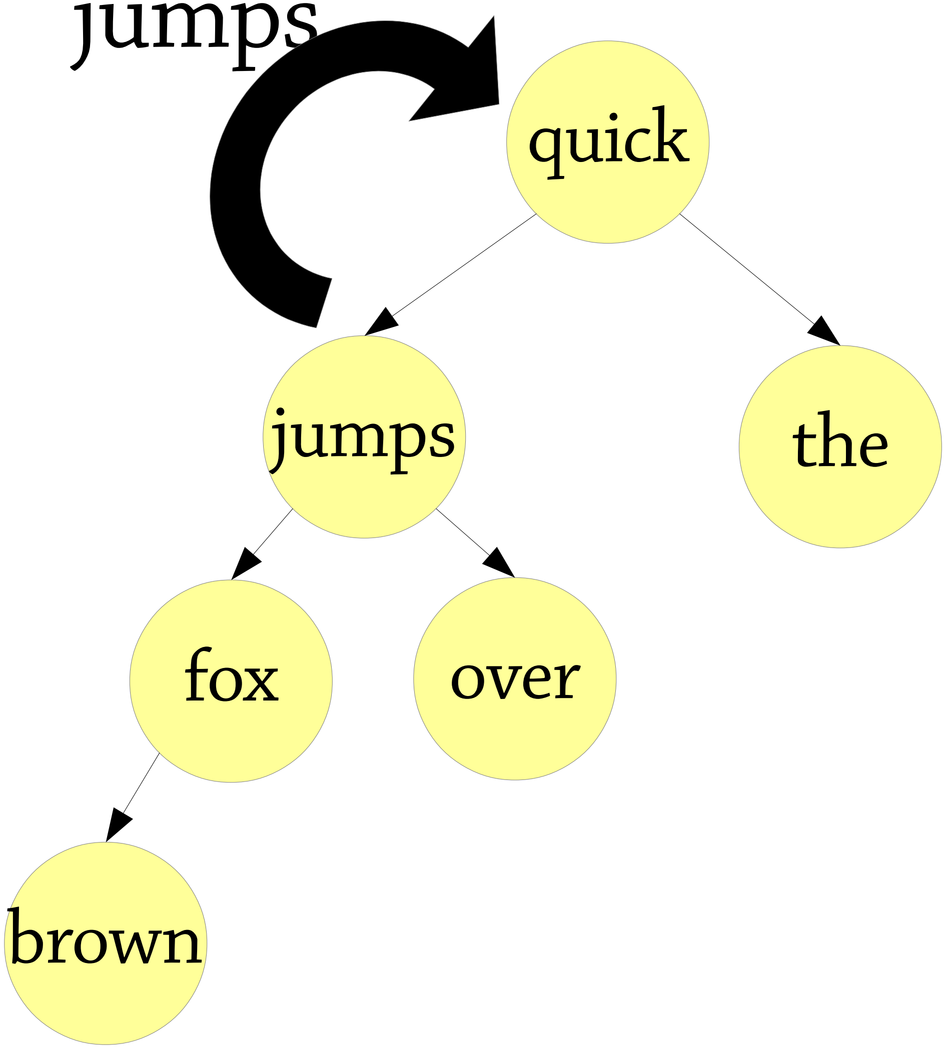
Insert “over” into “the quick brown fox jumps”




Left-right tree!
(quick → fox → jumps)
Rotate fox left...

# Example: the quick brown fox jumps over a lazy dog

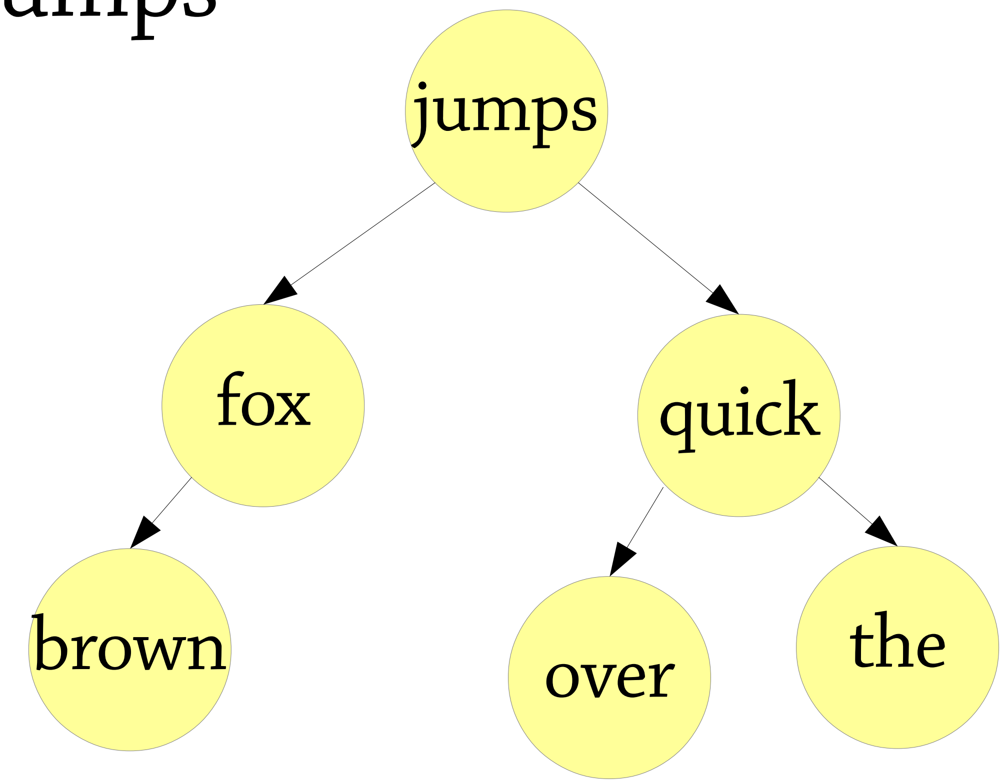
Insert "over" into "the quick brown fox jumps"




...then rotate quick right

# Example: the quick brown fox jumps over a lazy dog

Insert “over” into “the quick brown fox jumps”



# Deletion in an AVL tree

First do the normal BST deletion

Then go up the tree, finding nodes that break the invariant and fixing them using rotations

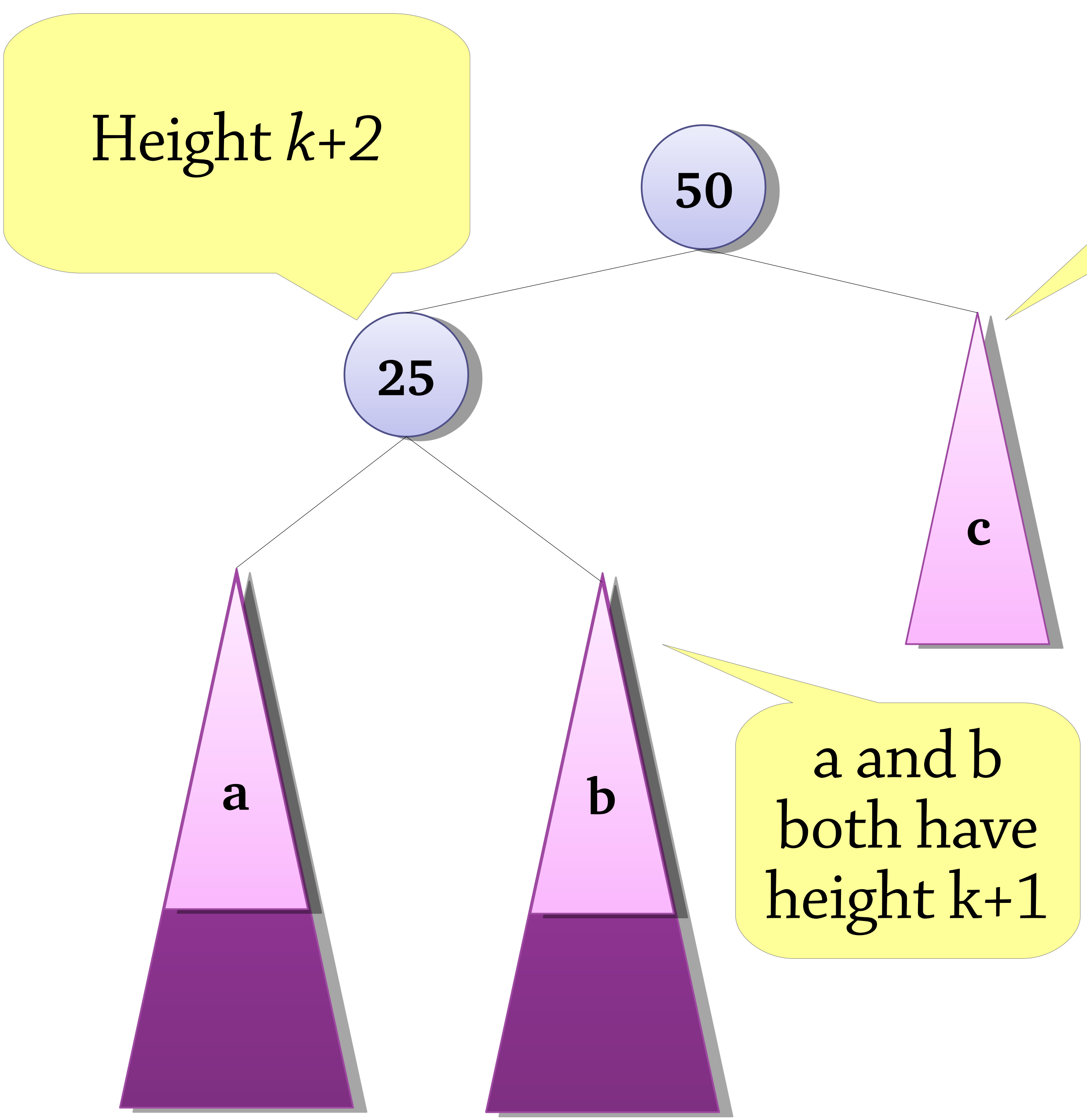
- Start from the node that was removed from the tree (recall that in the case that the value to be deleted was a node with two children, this was the biggest value in the left subtree)

The cases you need to consider are **exactly the same** as for insertion!

- When implementing AVL trees, you need only implement the balancing code once
- In general, the balancing algorithm works for any node where the left and right children satisfy the AVL invariant, but their heights differ by 2. Nothing specific to insertion...

There is one subtle point, see next slide...

# An extra case in deletion




This case doesn't appear in insertion (think about why).

But it works fine when treated as a **left-left tree**!

# How balanced are AVL trees?

Consider the smallest AVL tree with height *h* ($h \geq 2$). It must have two children:

- One of height *h-1*, so that the tree to have height *h*
- One of height *h-2*, so that the tree is as small as possible

Thus, if F(h) is the size of the smallest AVL tree of height *h*, we have:

- $F(0) = 0$, $F(1) = 1$, $F(h) = F(h-1)+F(h-2)$ if $h \geq 2$

Thus F(h) is the *h*th Fibonacci number!

- $F(h) \sim \varphi^h$, where $\varphi$ is the golden ratio
- If an AVL tree has size n and height h, then $n \geq \varphi^h$
- Taking logs of both sides, $h \leq \log_\varphi n = \log_2 n / \log_2 \varphi \sim 1.44 \log_2 n$

So: an AVL tree of n nodes has height at most $1.44 \log_2 n$

# AVL trees

Use *rotation* to keep the tree balanced

- Worst case height 1.44 $\log_2 n$, normally close to $\log_2 n$ – so lookups are quick

Insertion/deletion – BST insertion/deletion, then rotate to repair the invariant

Visualisation:

- http://visualgo.net/
- https://www.cs.usfca.edu/~galles/visualization/AVLtree.html