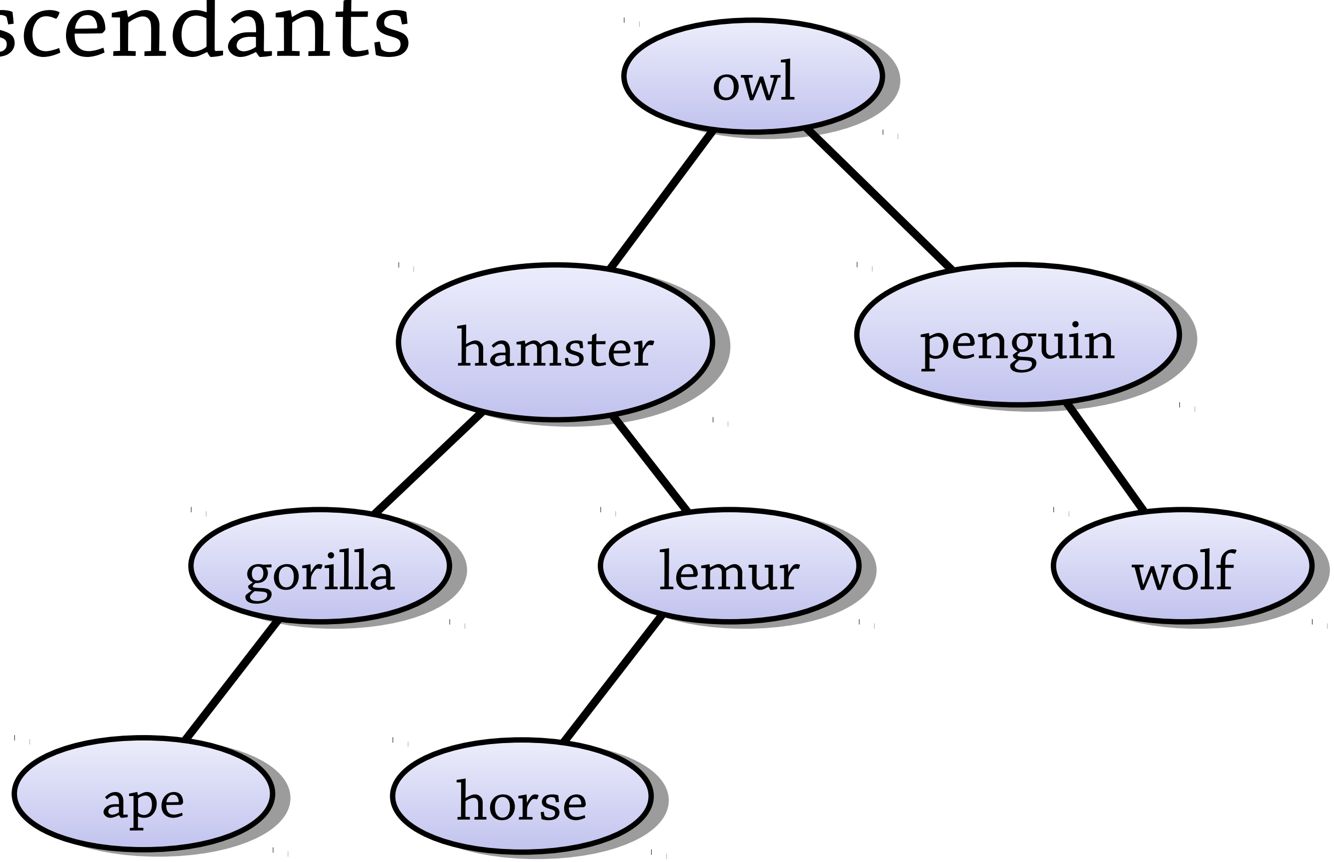
Binary search trees

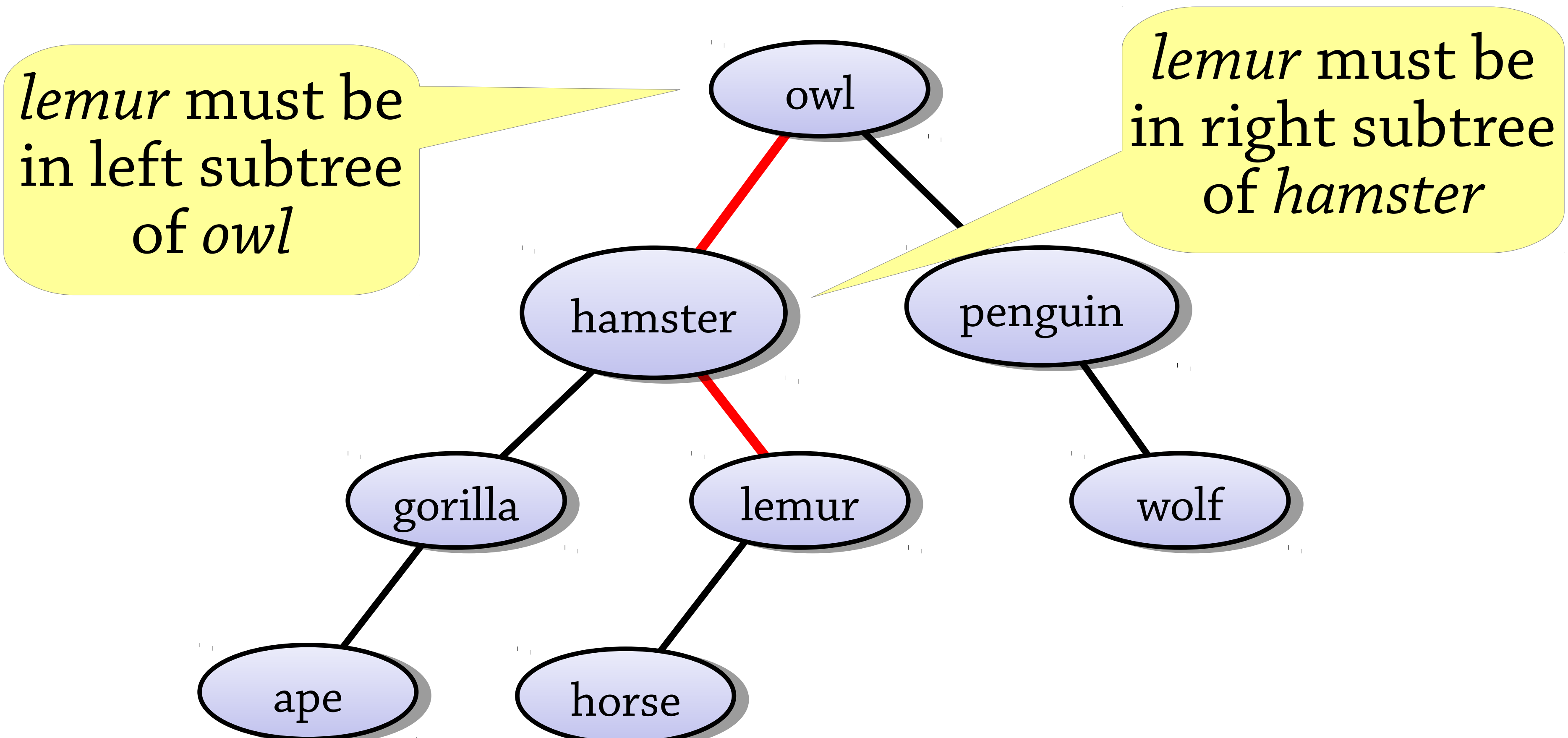
Binary search trees

A *binary search tree* (BST) is a binary tree where each node is greater than all its left descendants, and less than all its right descendants



Searching in a BST

Finding an element in a BST is easy, because by looking at the root you can tell which subtree the element is in



Searching in a binary search tree

To search for *target* in a BST:

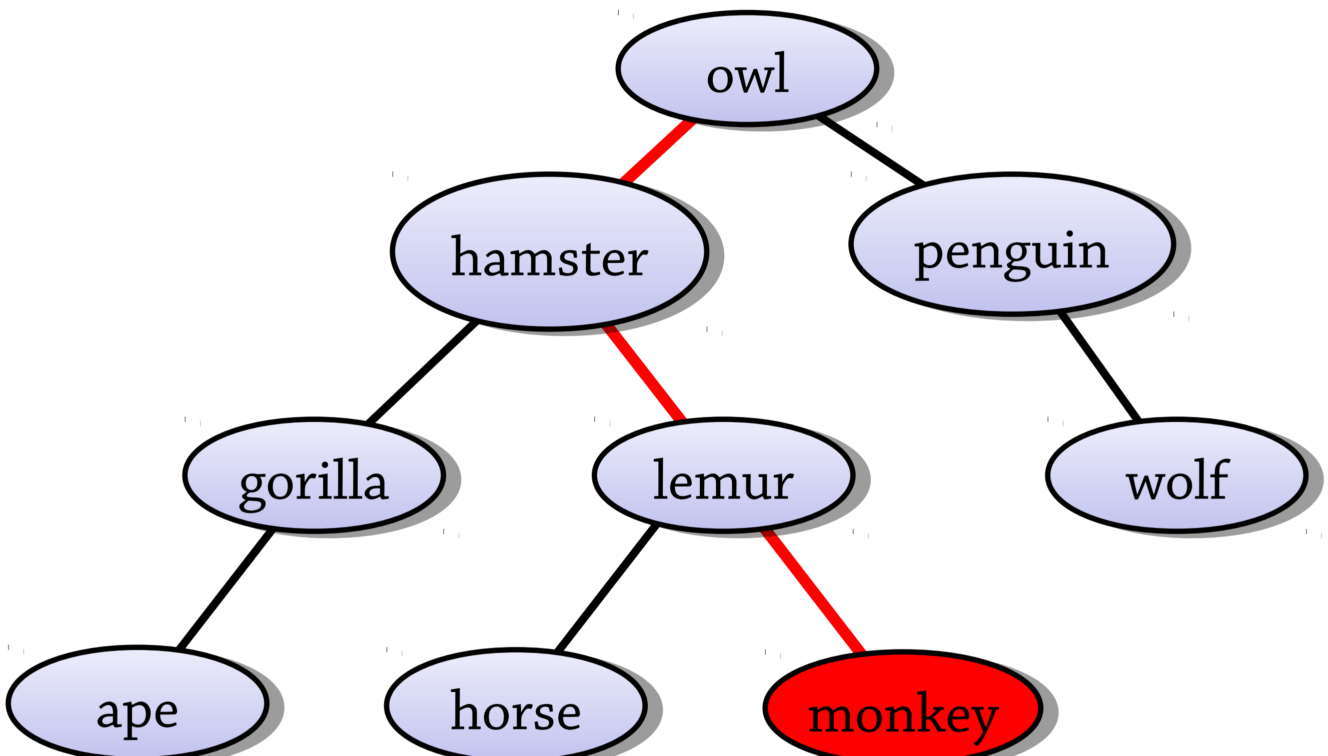
- If the target matches the root node's data, we've found it
- If the target is *less* than the root node's data, recursively search the left subtree
- If the target is *greater* than the root node's data, recursively search the right subtree
- If the tree is empty, fail

A BST can be used to implement a set, or a map from keys to values

Inserting into a BST

To insert a value into a BST:

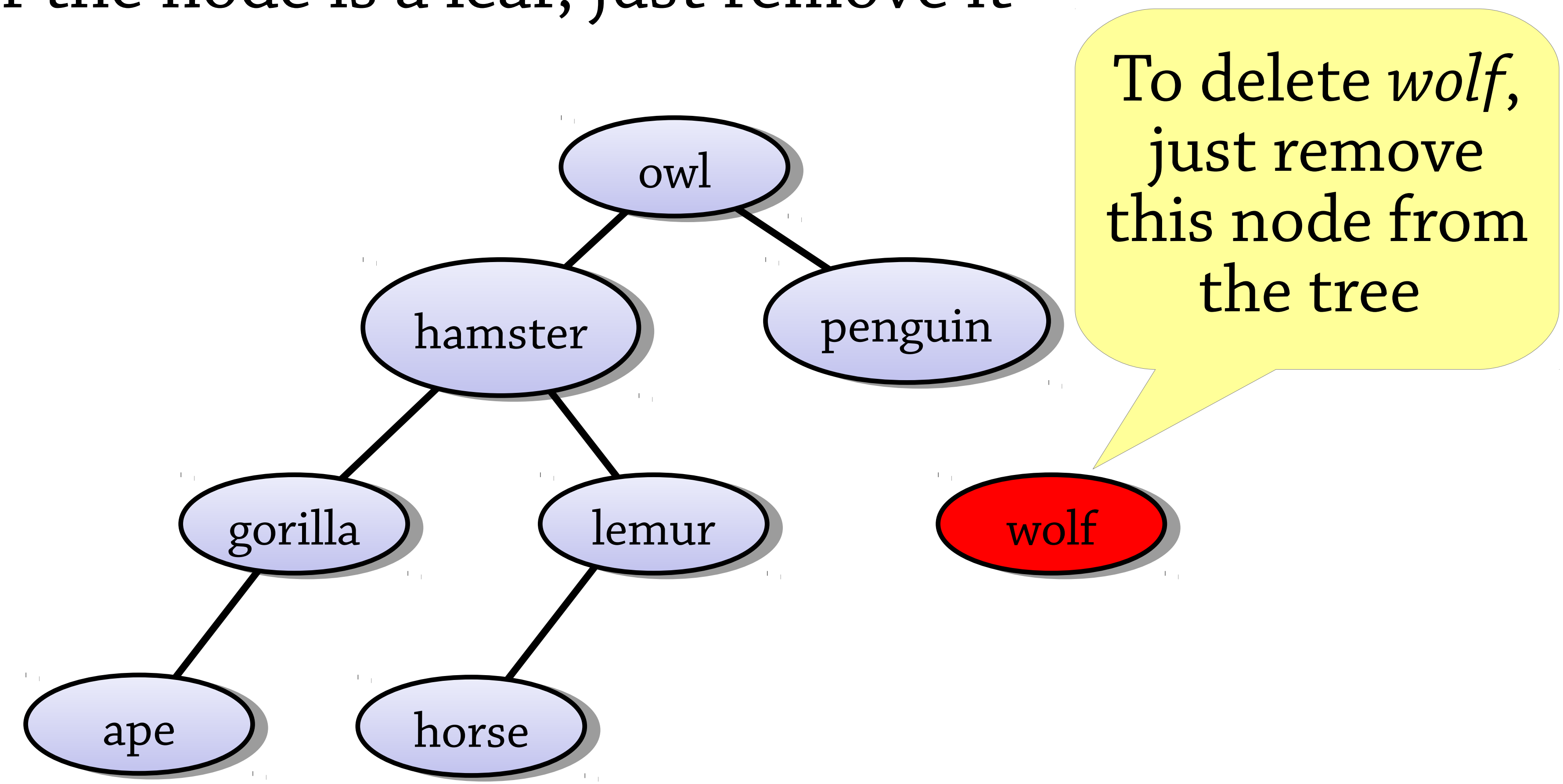
- Start by searching for the value
- But when you get to *null* (the empty tree), make a node for the value and place it there



Deleting from a BST

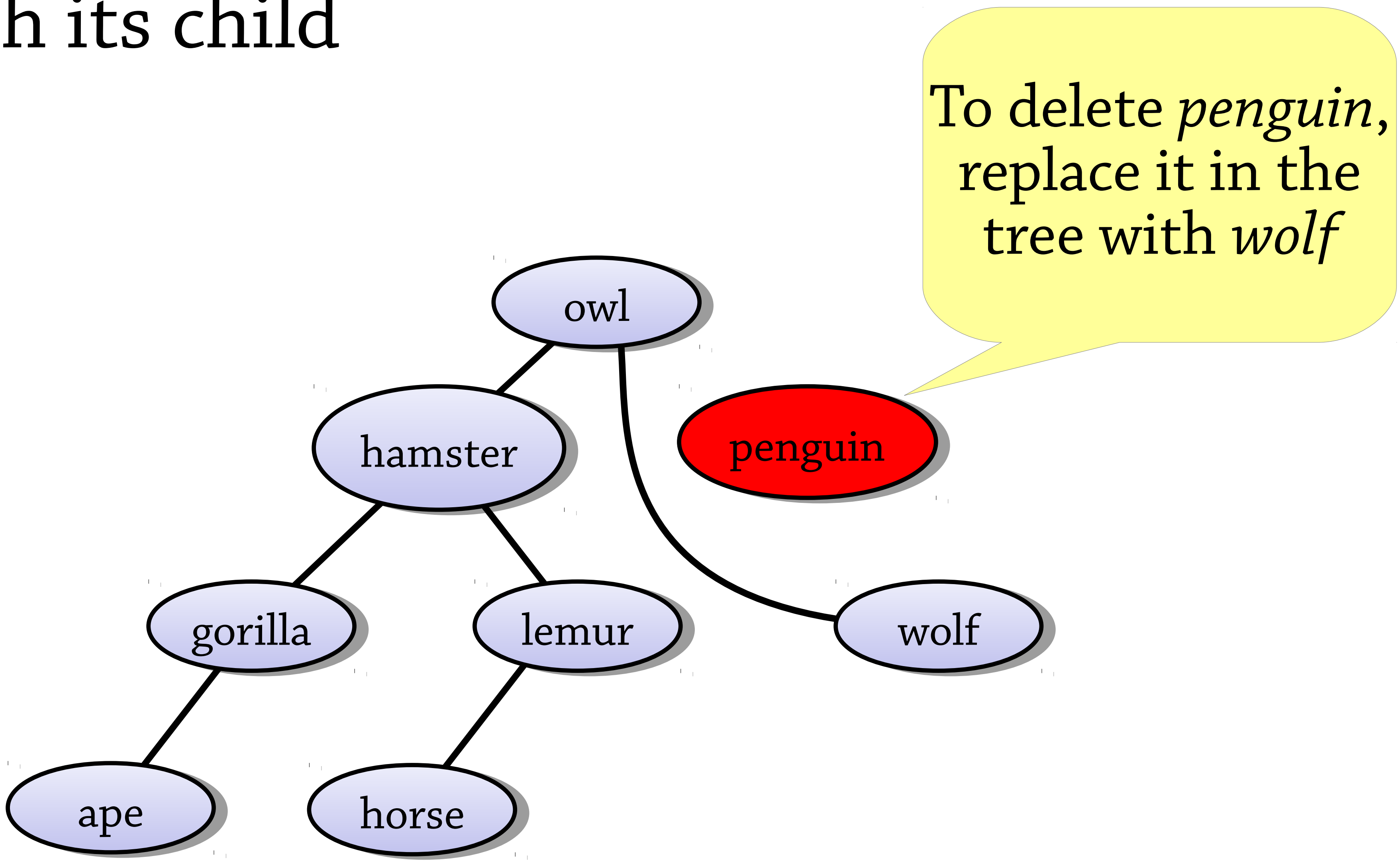
To delete a value into a BST:

- Find the node containing the value
- If the node is a leaf, just remove it



Deleting from a BST, continued

If the node has *one* child, replace the node with its child



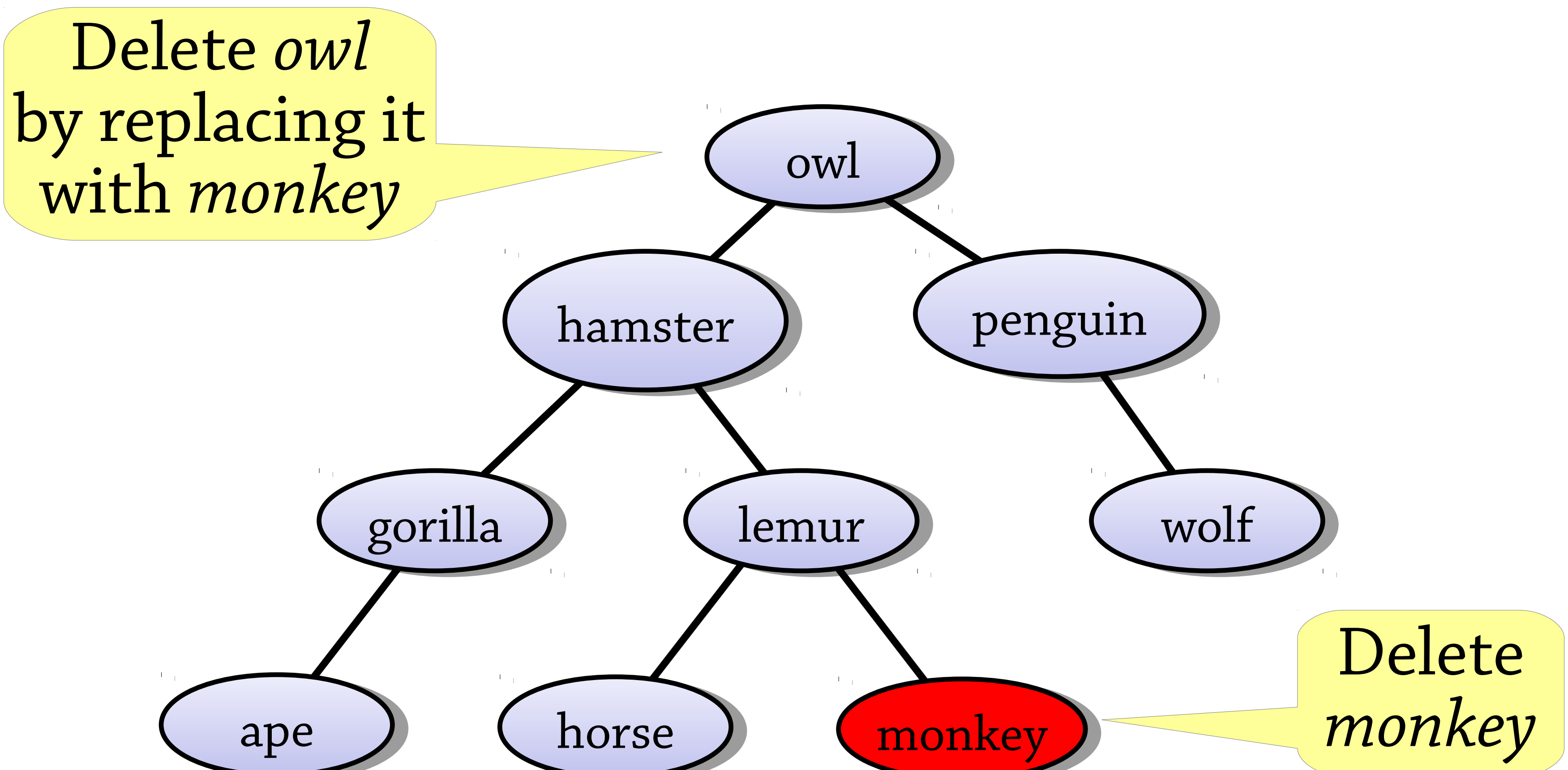
Deleting from a BST

To delete a value from a BST:

- Find the node
- If it has no children, just remove it from the tree
- If it has one child, replace the node with its child
- If it has two children...?
  Can't remove the node without removing its children too!

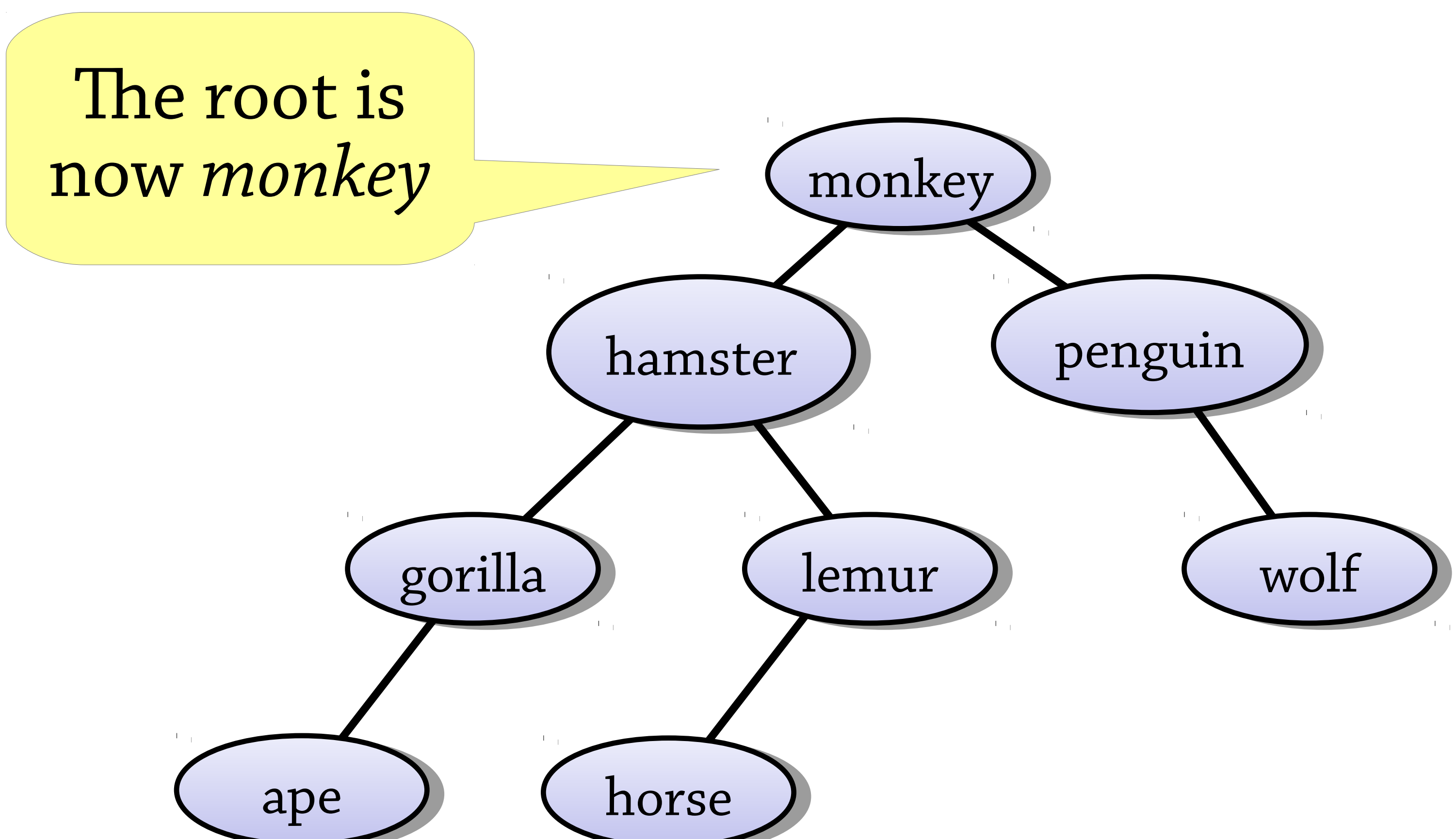
Deleting a node with two children

Delete the *biggest value from the node's left subtree* and put this value [why this one?] in place of the node we want to delete



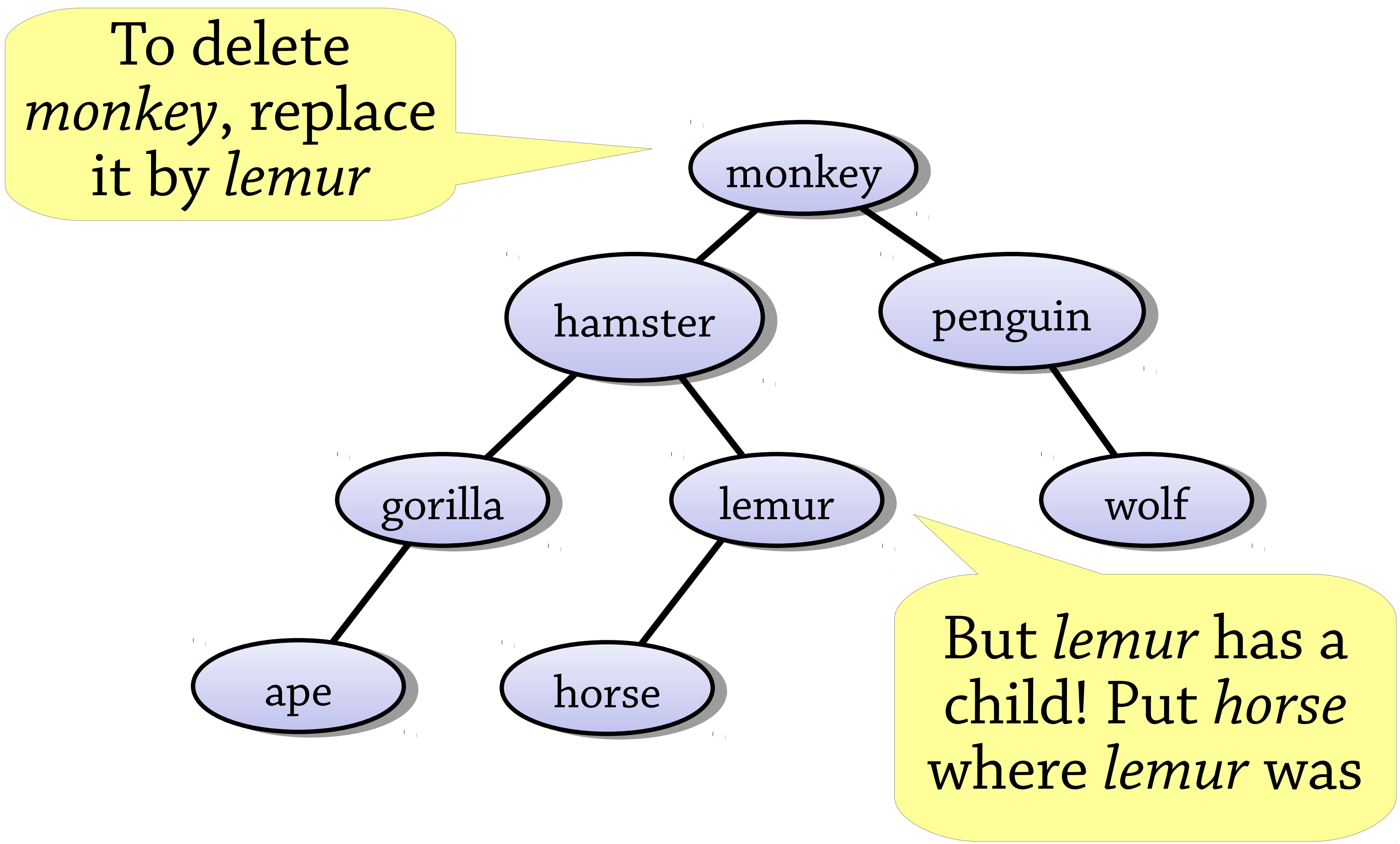
Deleting a node with two children

Delete the *biggest value from the node's left subtree* and put this value [why this one?] in place of the node we want to delete



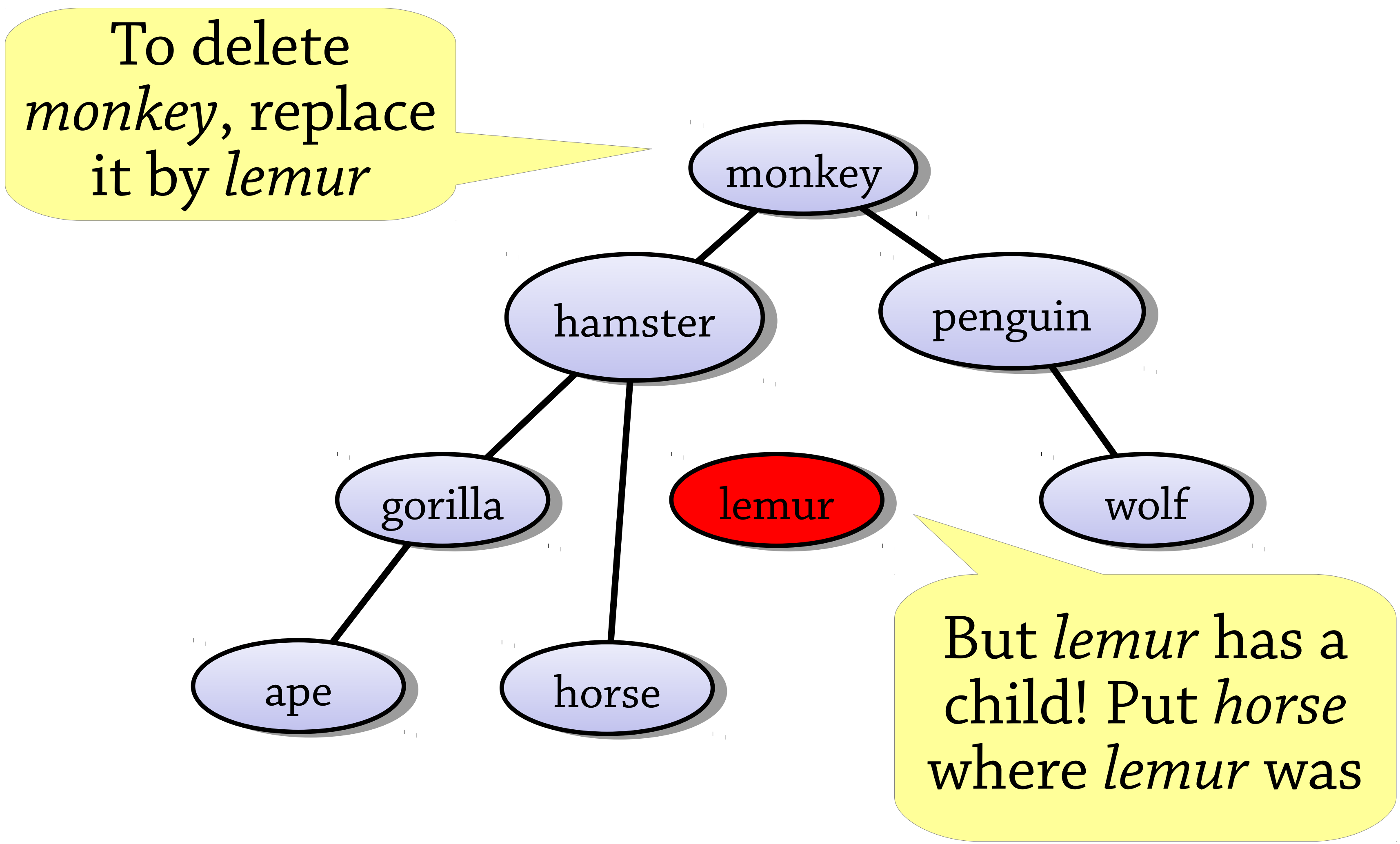
Deleting a node with two children

Here is the most complicated case:



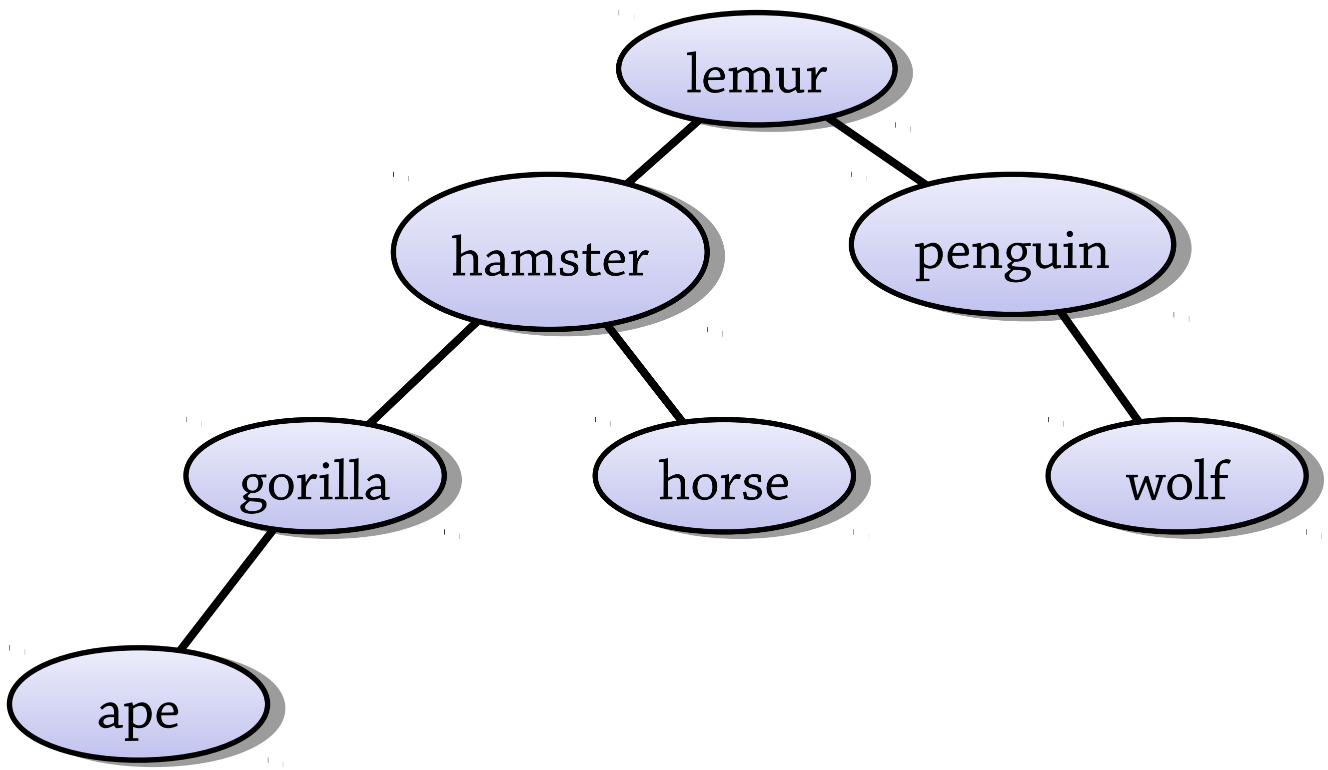
Deleting a node with two children

Here is the most complicated case:



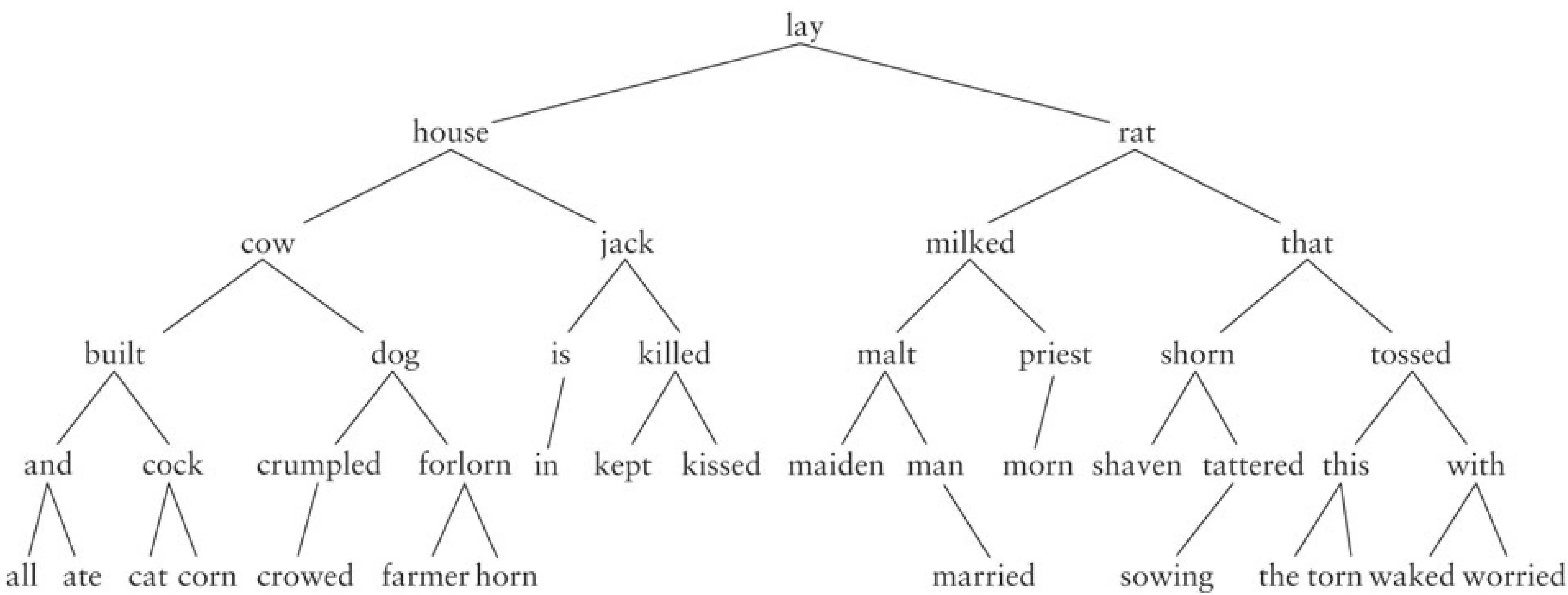
Deleting a node with two children

Here is the most complicated case:



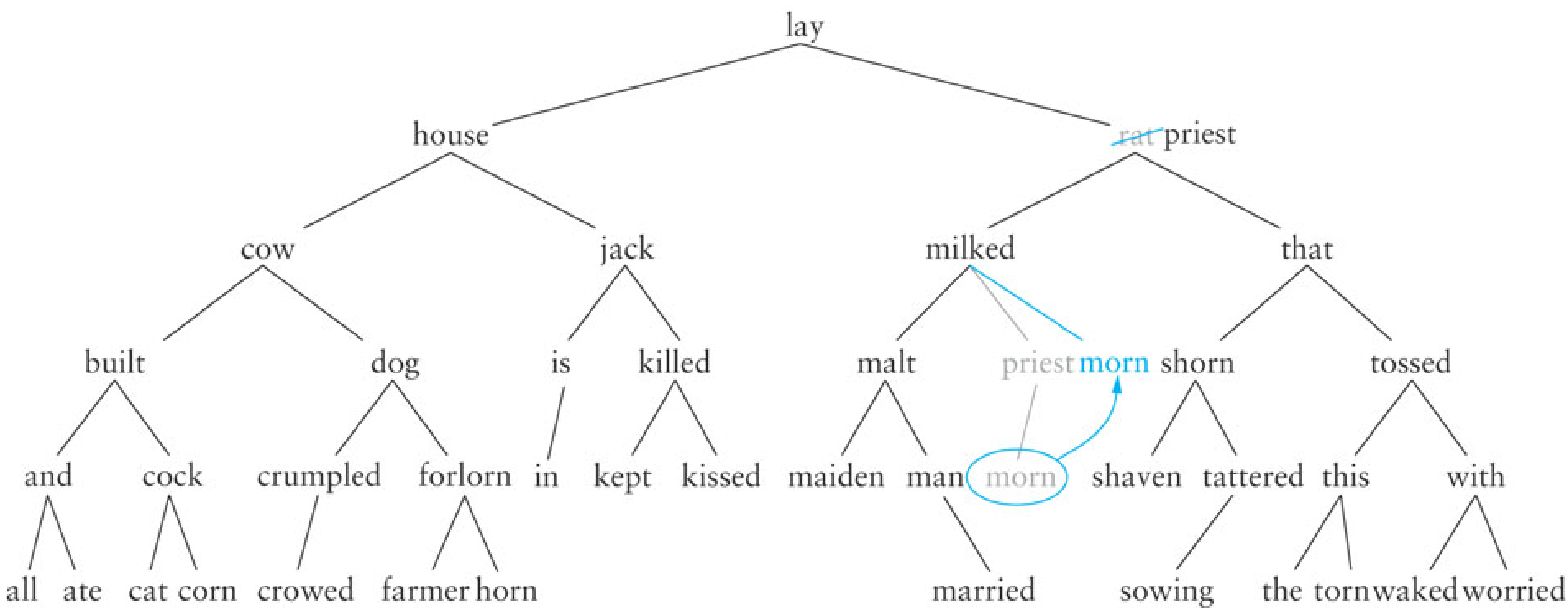
A bigger example

What happens if we delete
is? cow? rat?



Deleting a node with two children

Deleting *rat*, we replace it with *priest*; now we have to delete *priest* which has a child, *morn*



Deleting a node with two children

Find and delete the *biggest value* in the *left subtree* and put that value in the deleted node

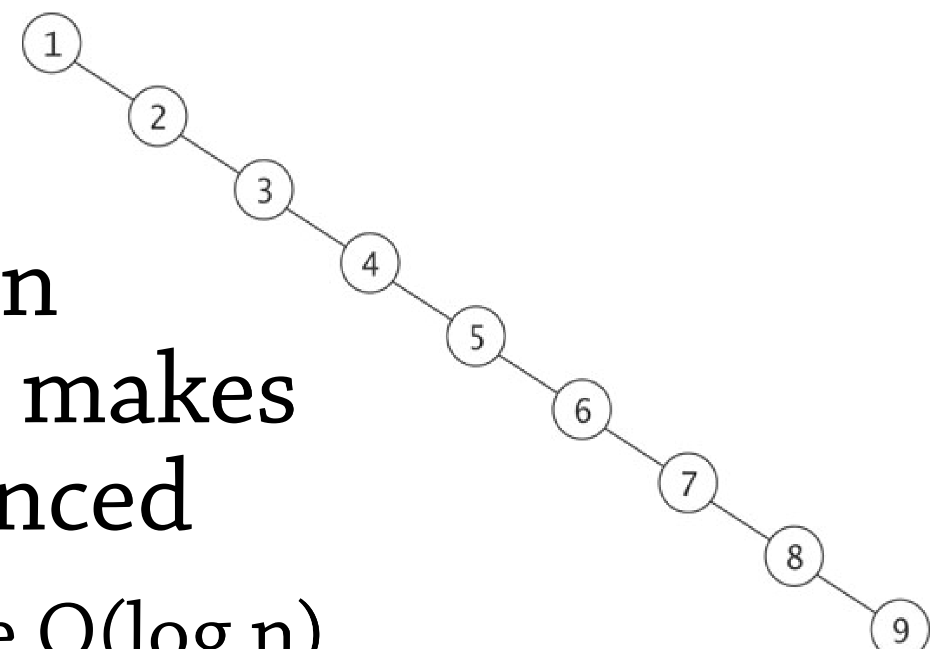
- Using the biggest value preserves the invariant (check you understand why)
- To find the biggest value: repeatedly descend into the right child until you find a node with no right child
- The biggest node can't have two children, so deleting it is easier

Complexity of BST operations

All our operations are O(height of tree)

This means O(log n) if the tree is balanced, but O(n) if it's unbalanced (like the tree on the right)

- how might we get this tree?



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

- then all operations are O(log n)

Summary of BSTs

Binary trees with *BST invariant*

Can be used to implement sets and maps

- lookup: can easily find a value in the tree
- insert: perform a lookup, then put the new value at the place where the lookup would stop
- delete: find the value, then remove its node from the tree – several cases depending on how many children the node has

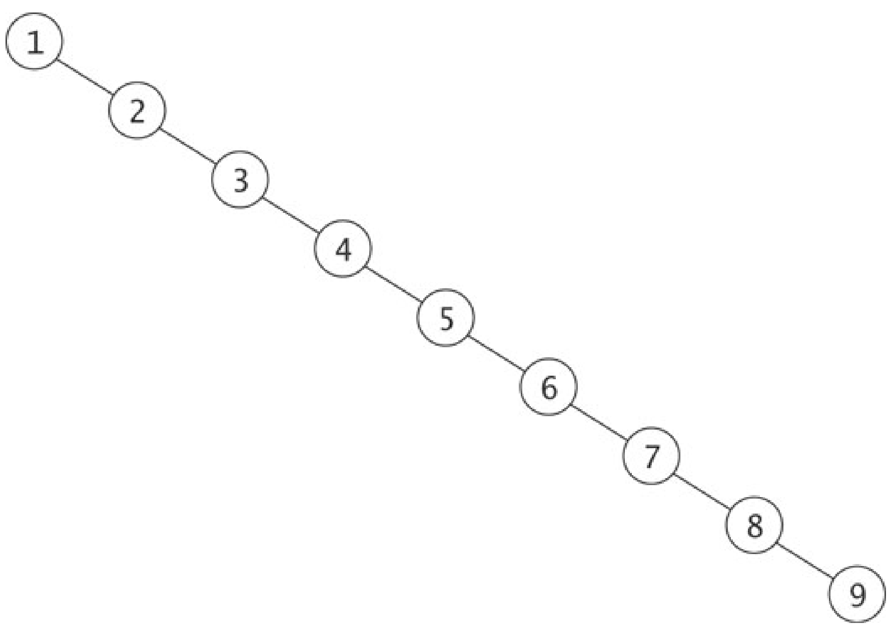
Complexity:

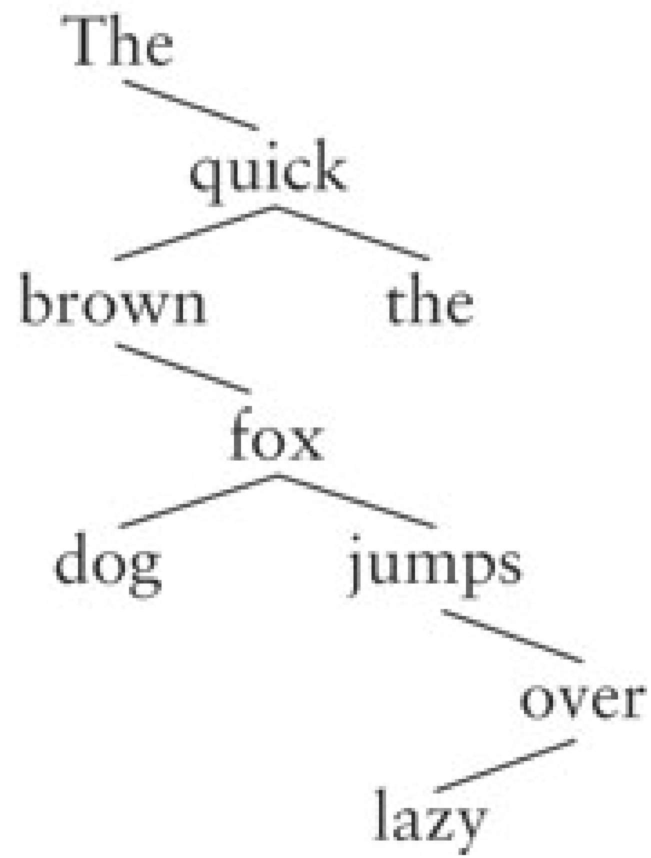
- all operations O(height of tree)
- that is, O(log n) if tree is balanced, O(n) if unbalanced
- inserting random data tends to give balanced trees, sequential data gives unbalanced ones

AVL trees

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:

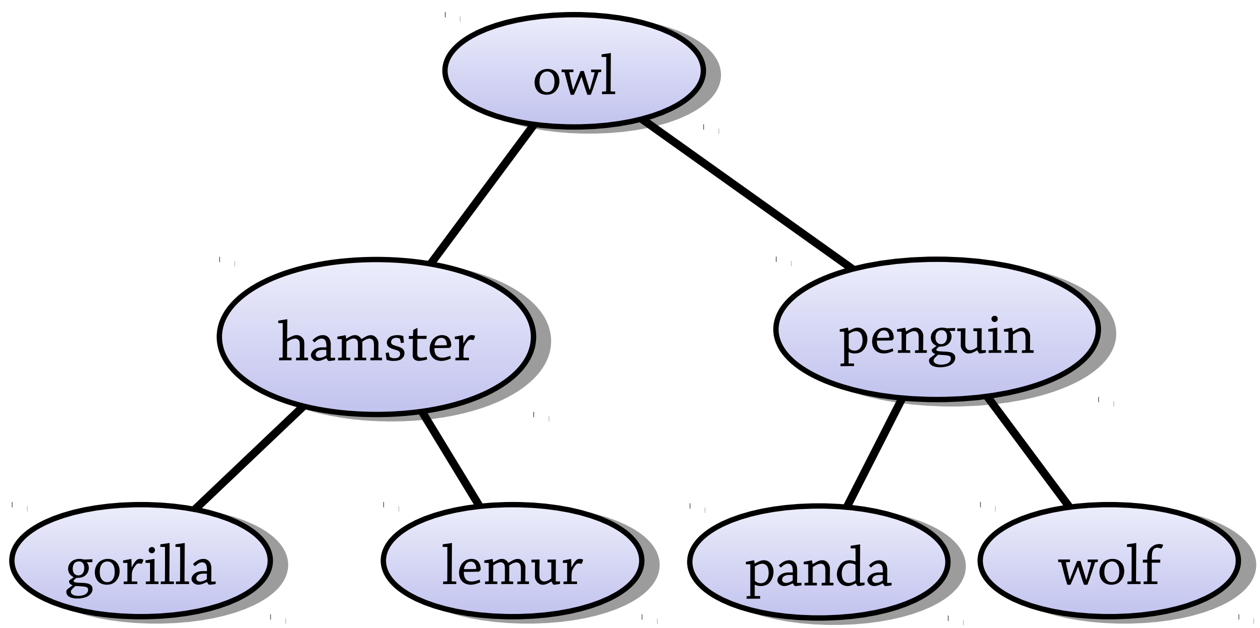
- Height of left child = height of right child (for all nodes in the tree)
- Tree would be sort of "perfectly balanced"

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

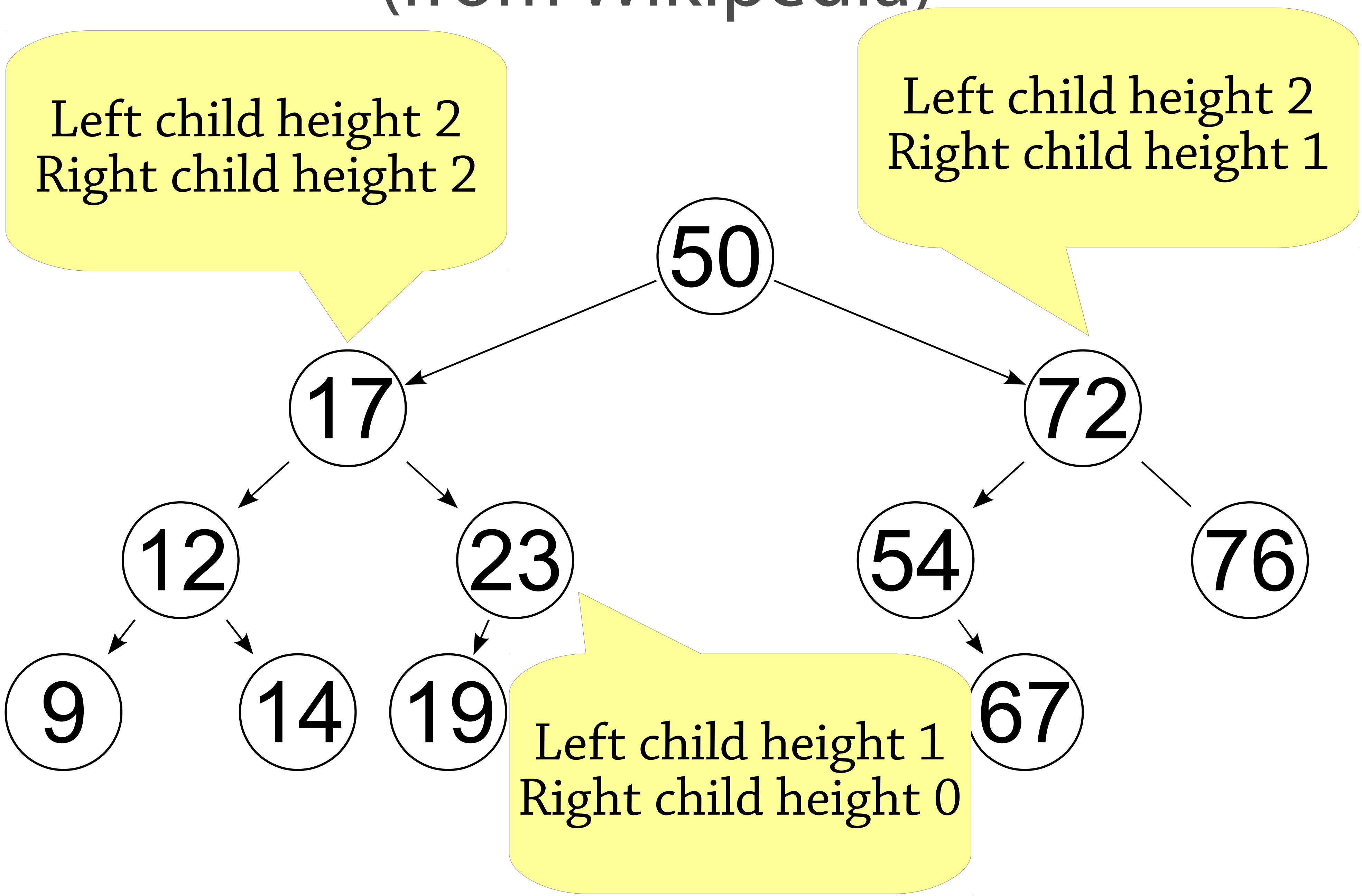
The AVL tree is the first balanced BST discovered (from 1962) – it's named after Adelson-Velsky and Landis

It's a BST with the following invariant:

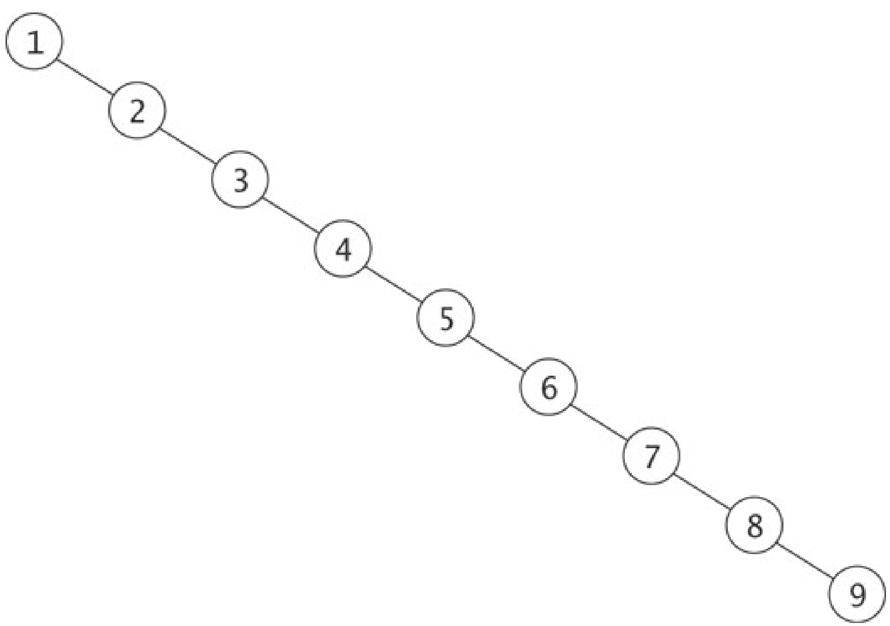
- The *difference in heights* between the left and right children of any node is at most 1
- (compared to 0 for a perfectly balanced tree)

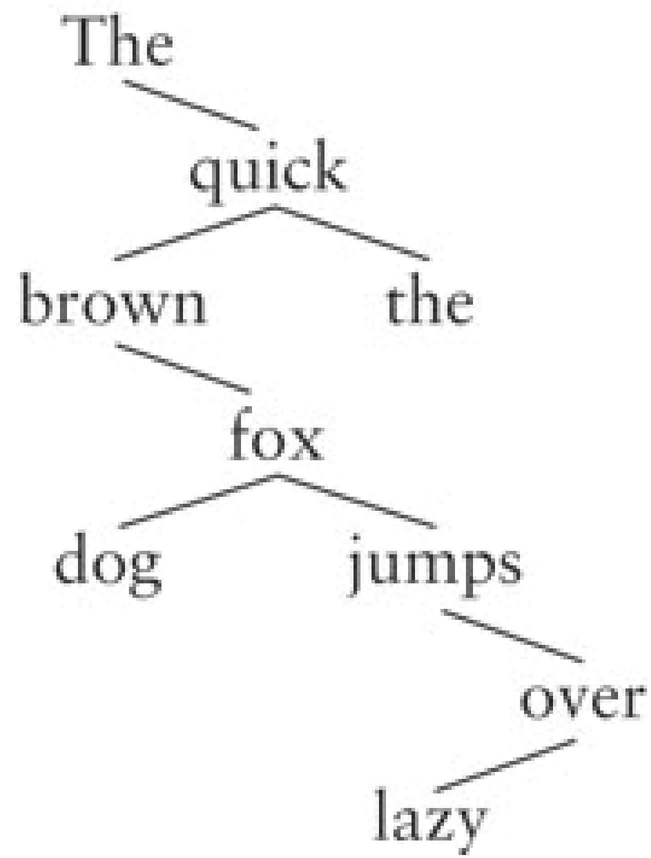
This makes the tree's height O(log n), so it's balanced

Example of an AVL tree (from Wikipedia)



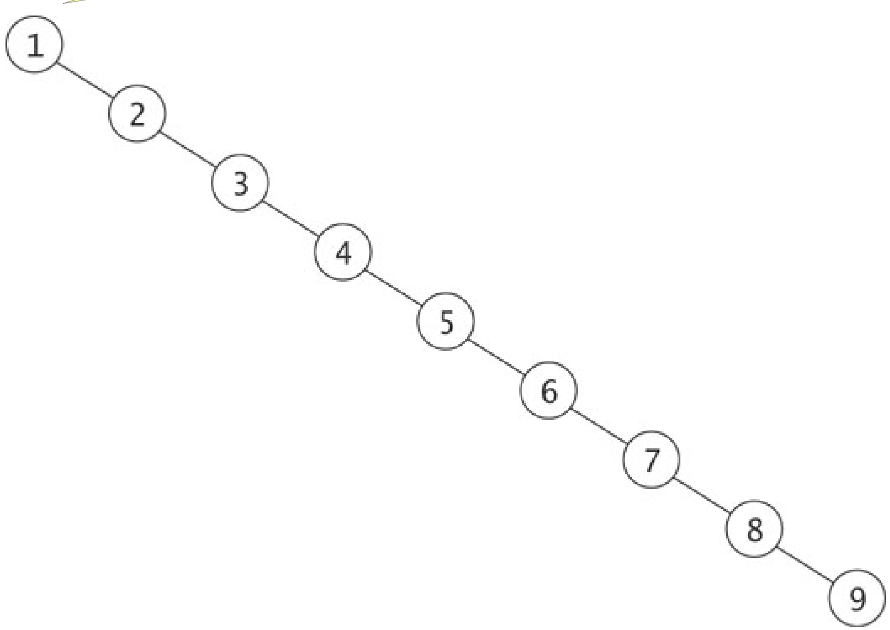
Why are these not AVL trees?

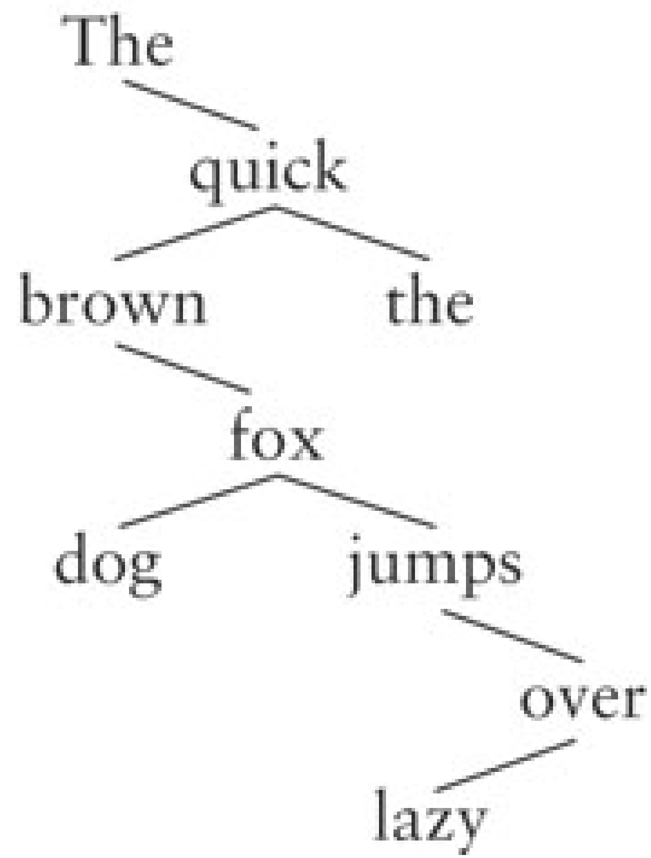




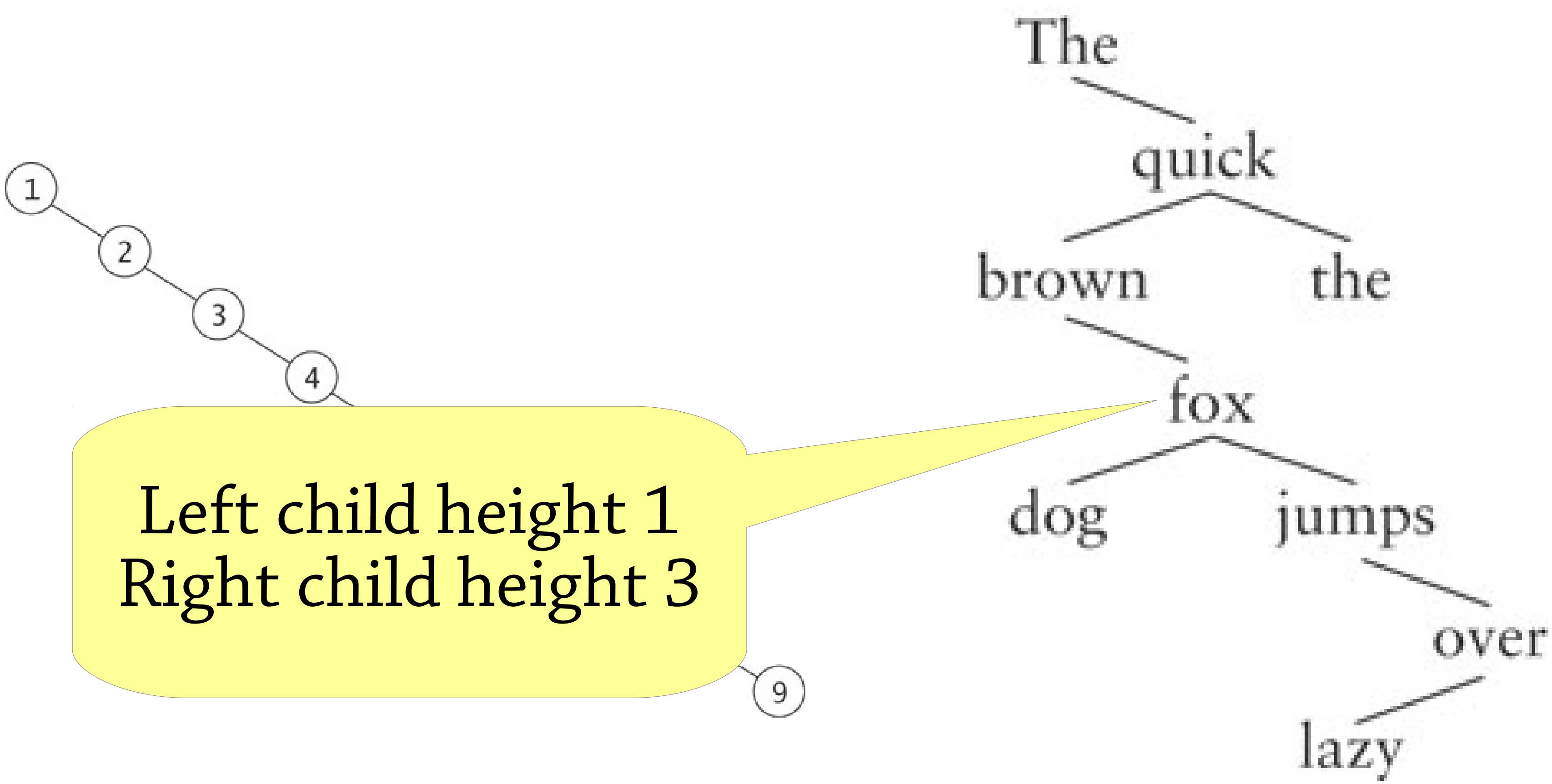
Why are these not AVL trees?

Left child height 0
Right child height 8



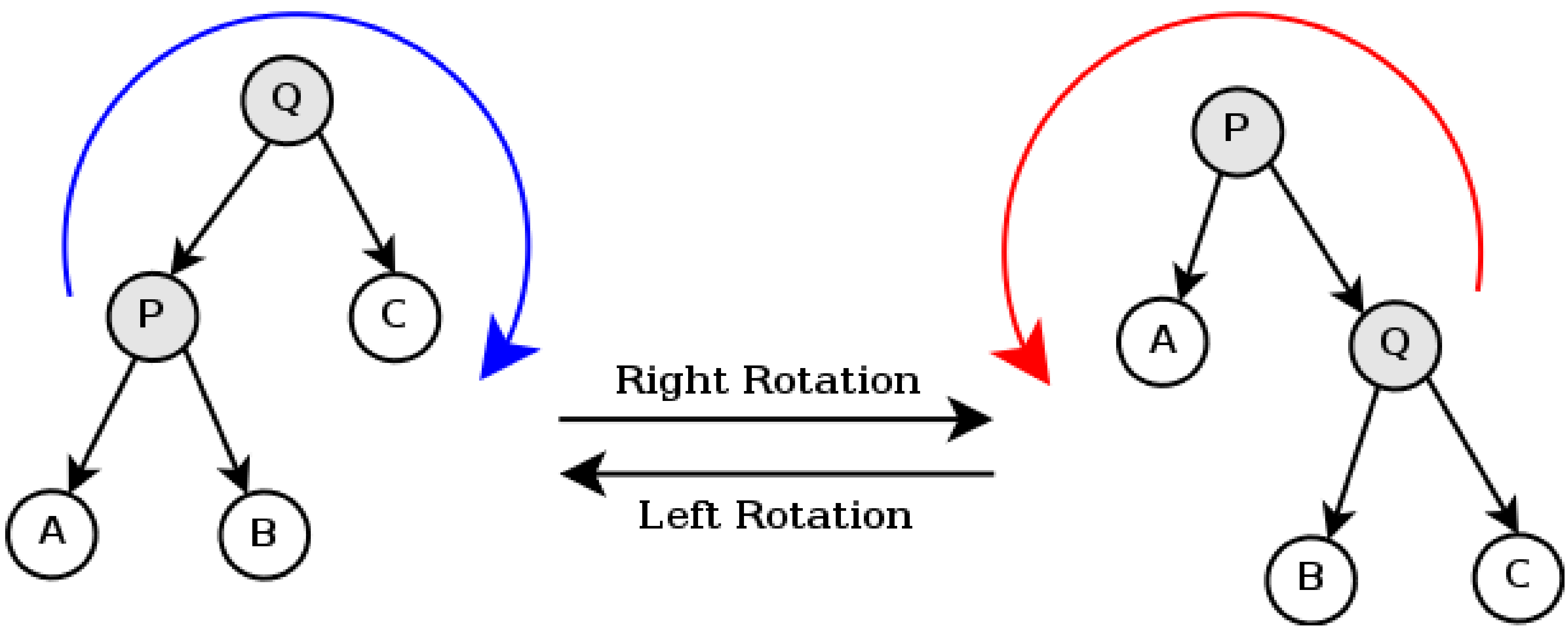


Why are these not AVL trees?



Rotation

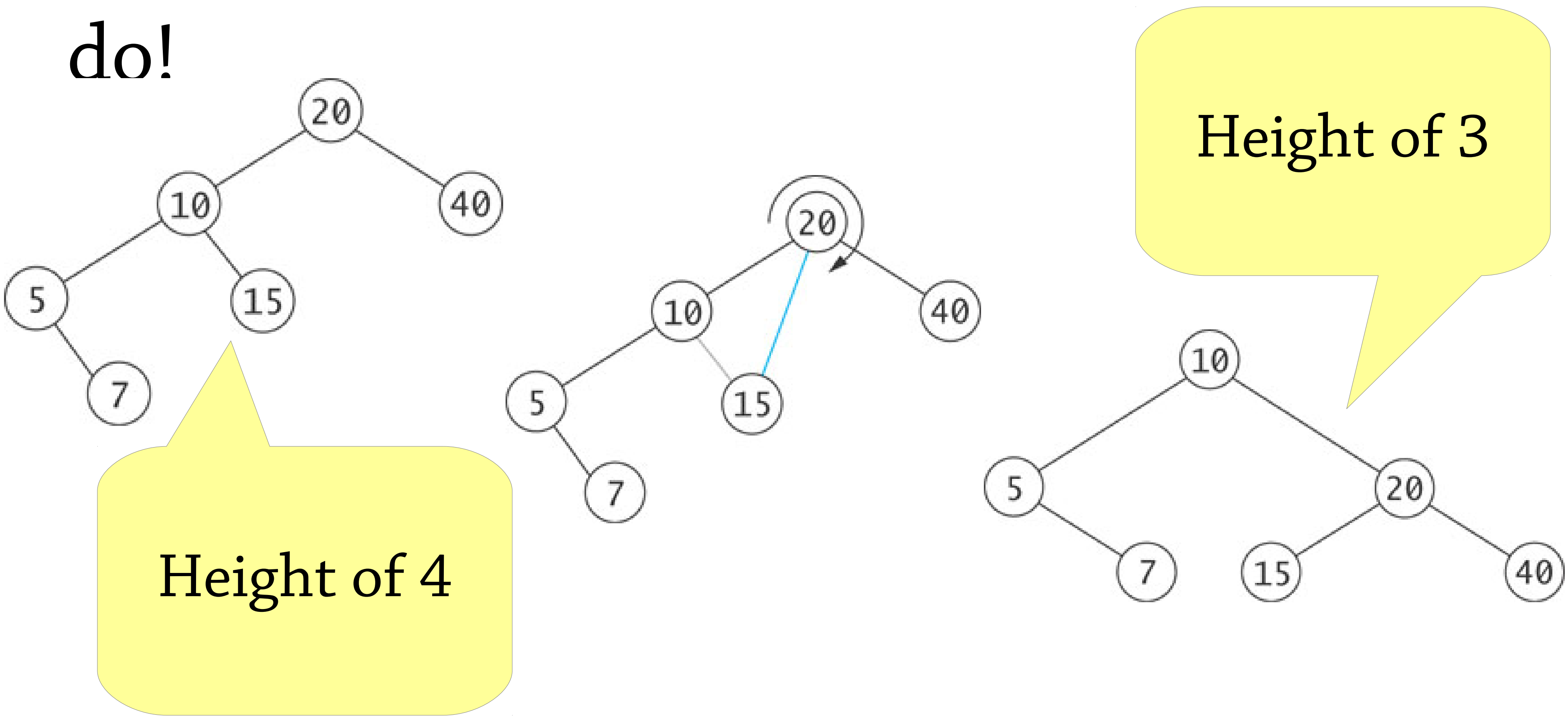
Rotation rearranges a BST by moving a different node to the root, without changing the BST's contents



(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

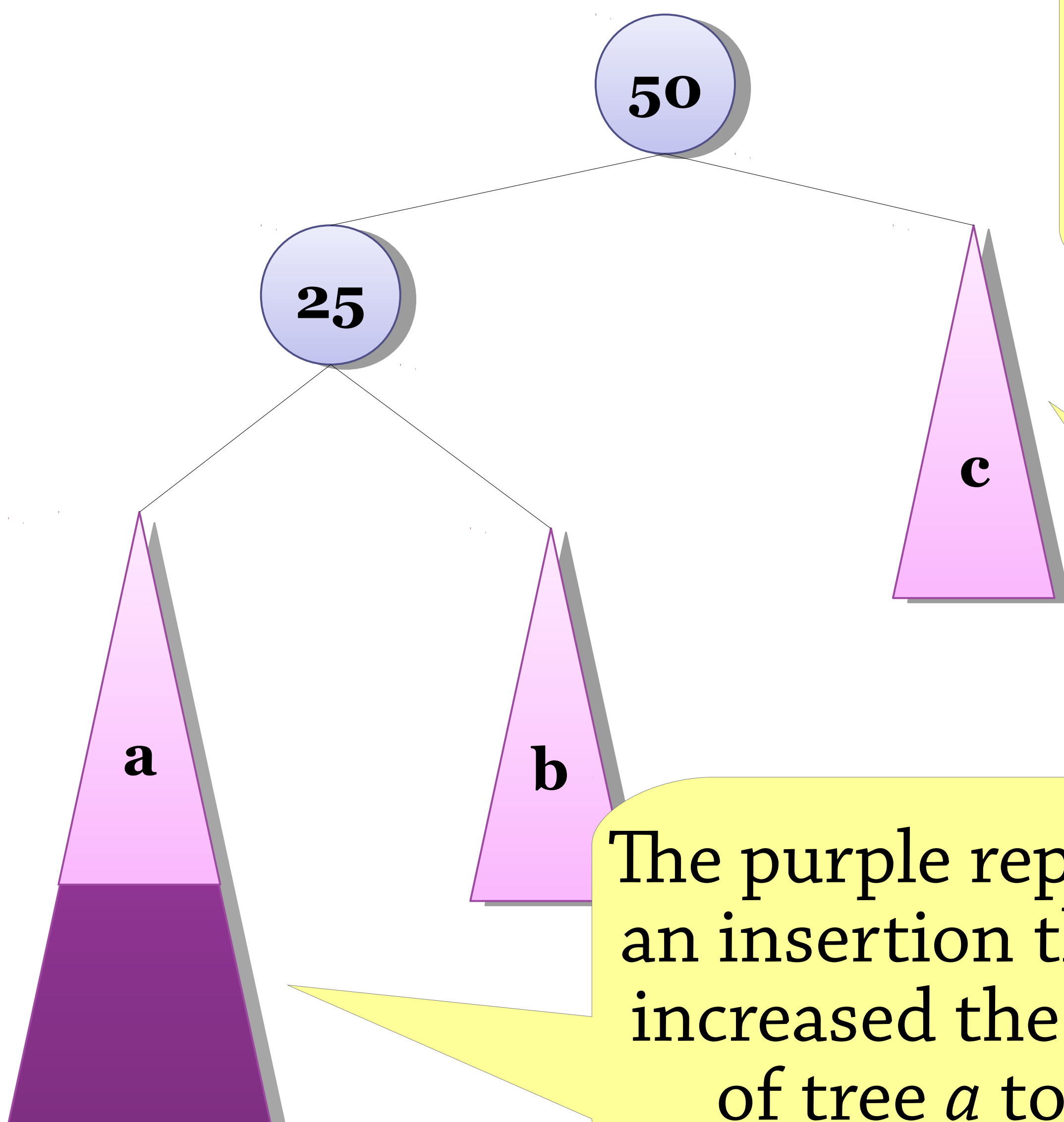
- This might break the AVL (balance) invariant

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

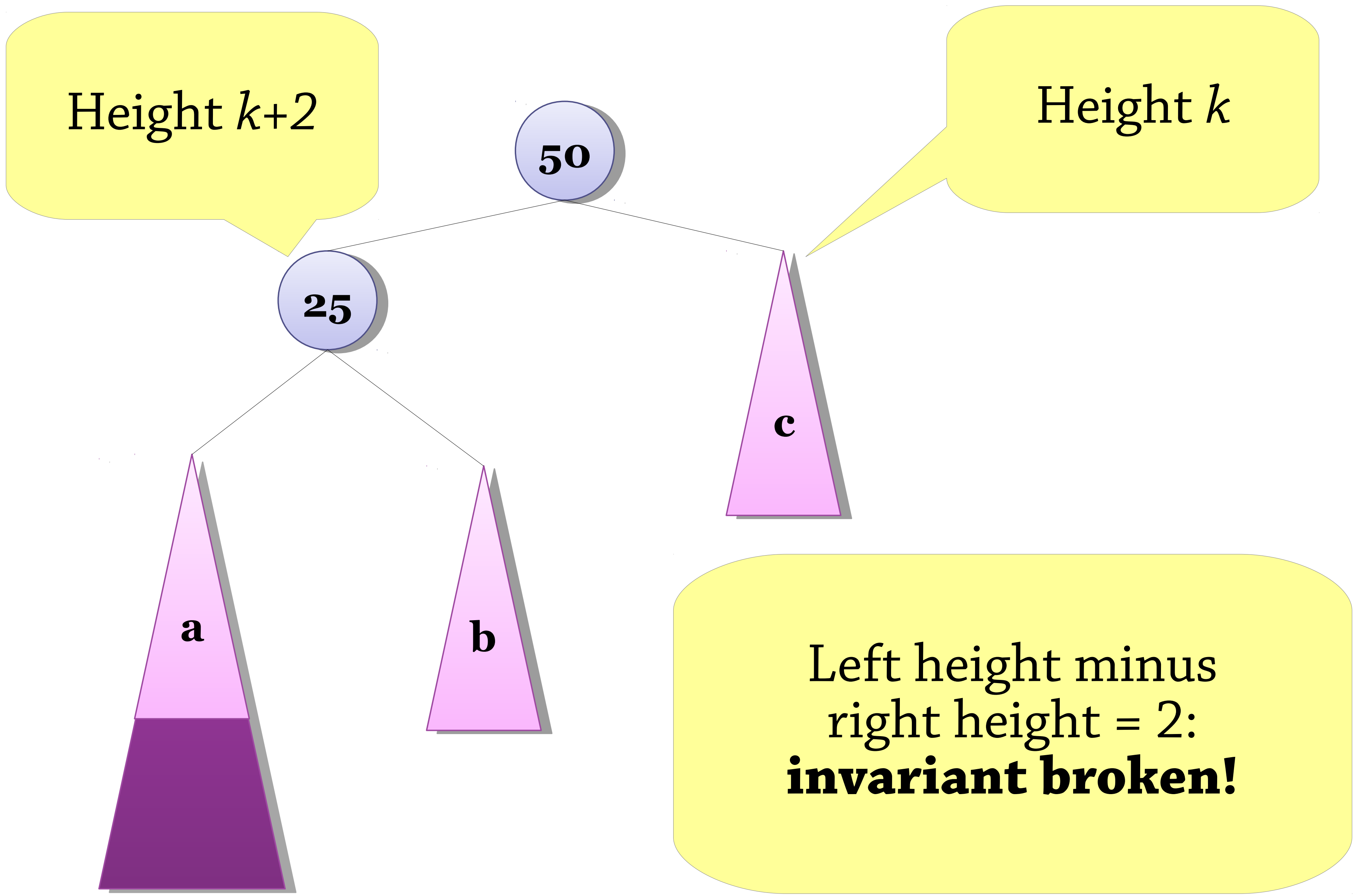


Notice that the tree was balanced before the purple bit was added

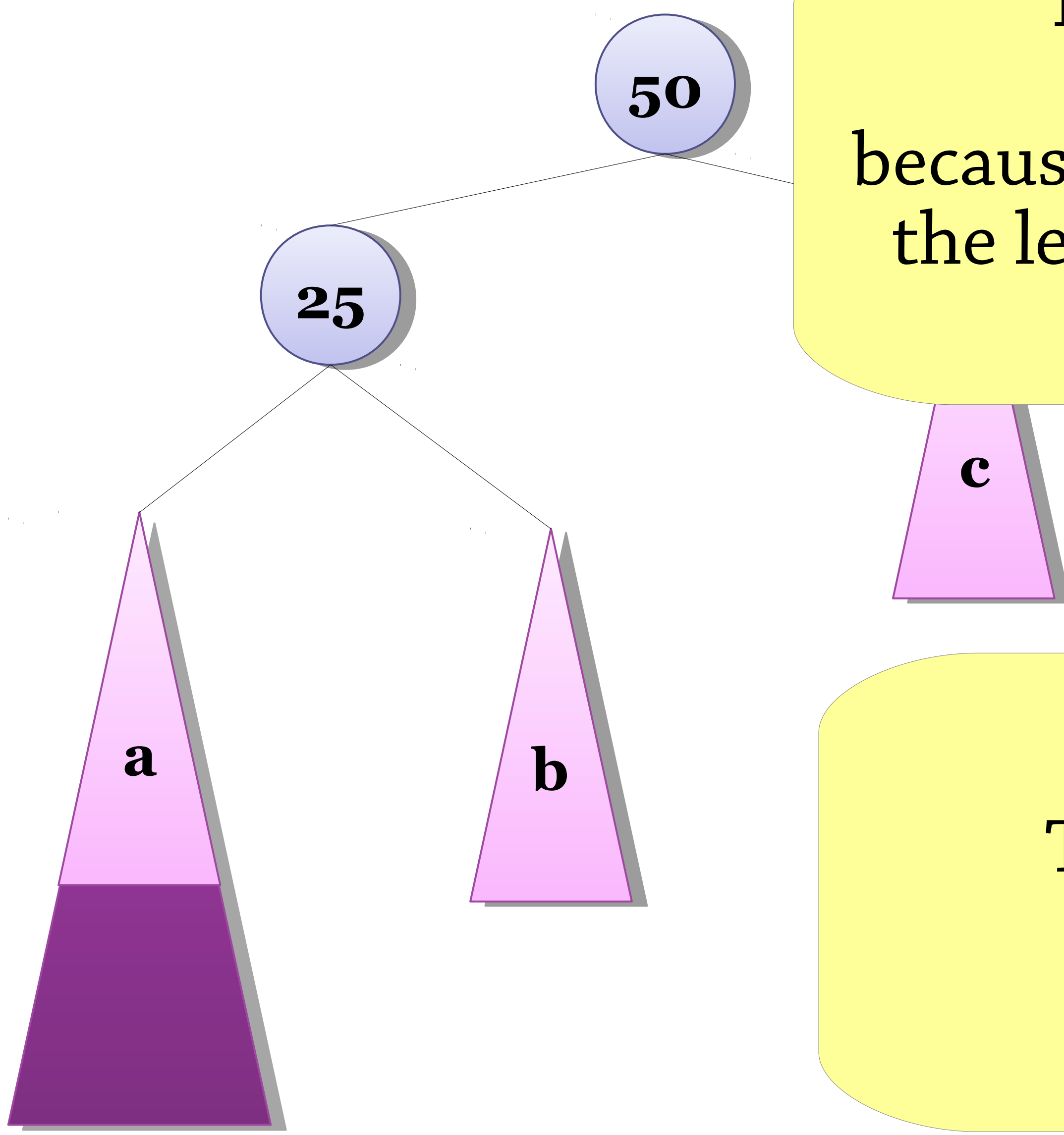
Each pink triangle represents an AVL tree with height *k*

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

Case 1: a *left-left* tree



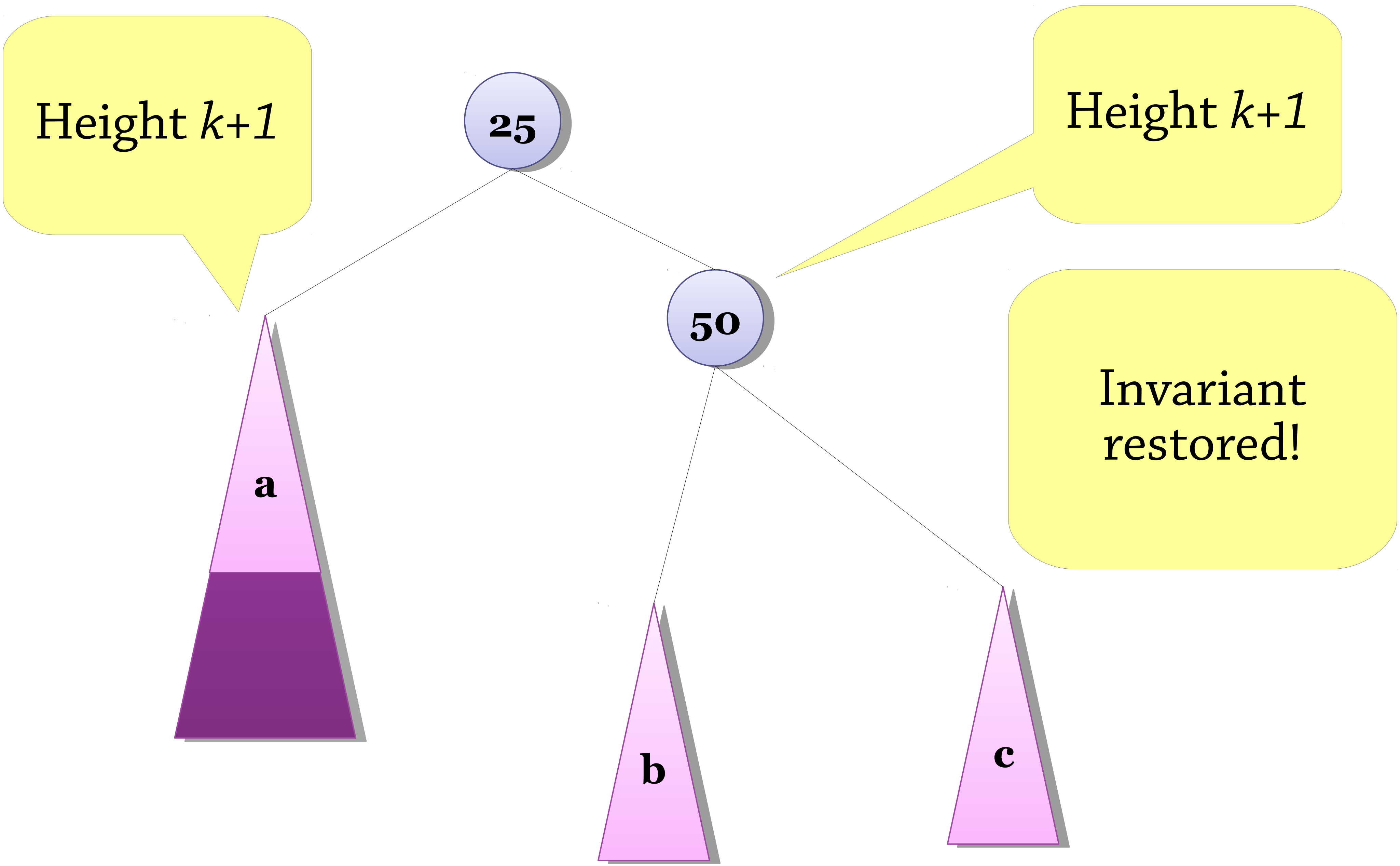
Case 1: a *left-left* tree



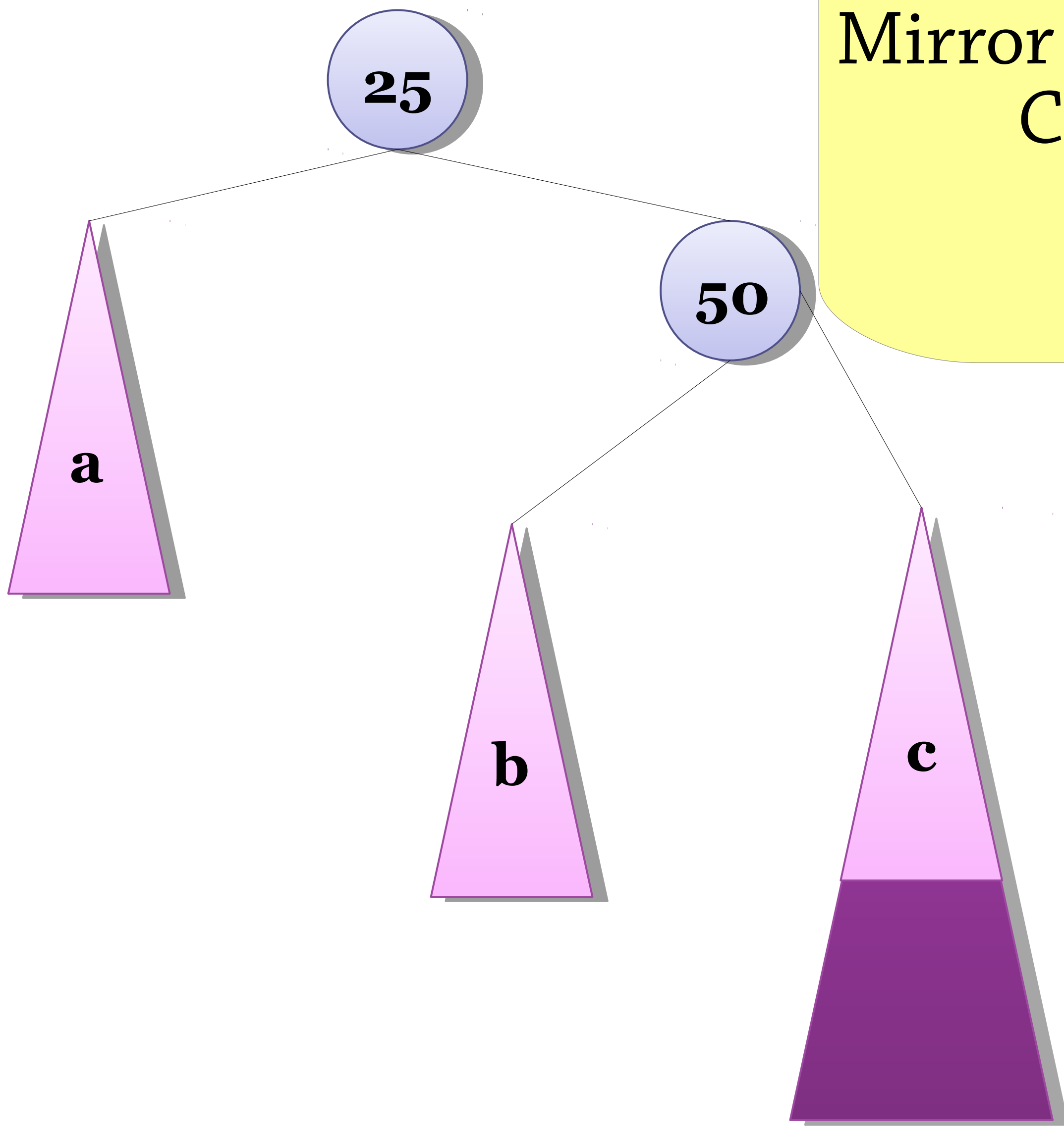
This is called a
left-left tree
because both the root and
the left child are deeper
on the left

To fix it we do a
right rotation

Balancing a left-left tree, afterwards

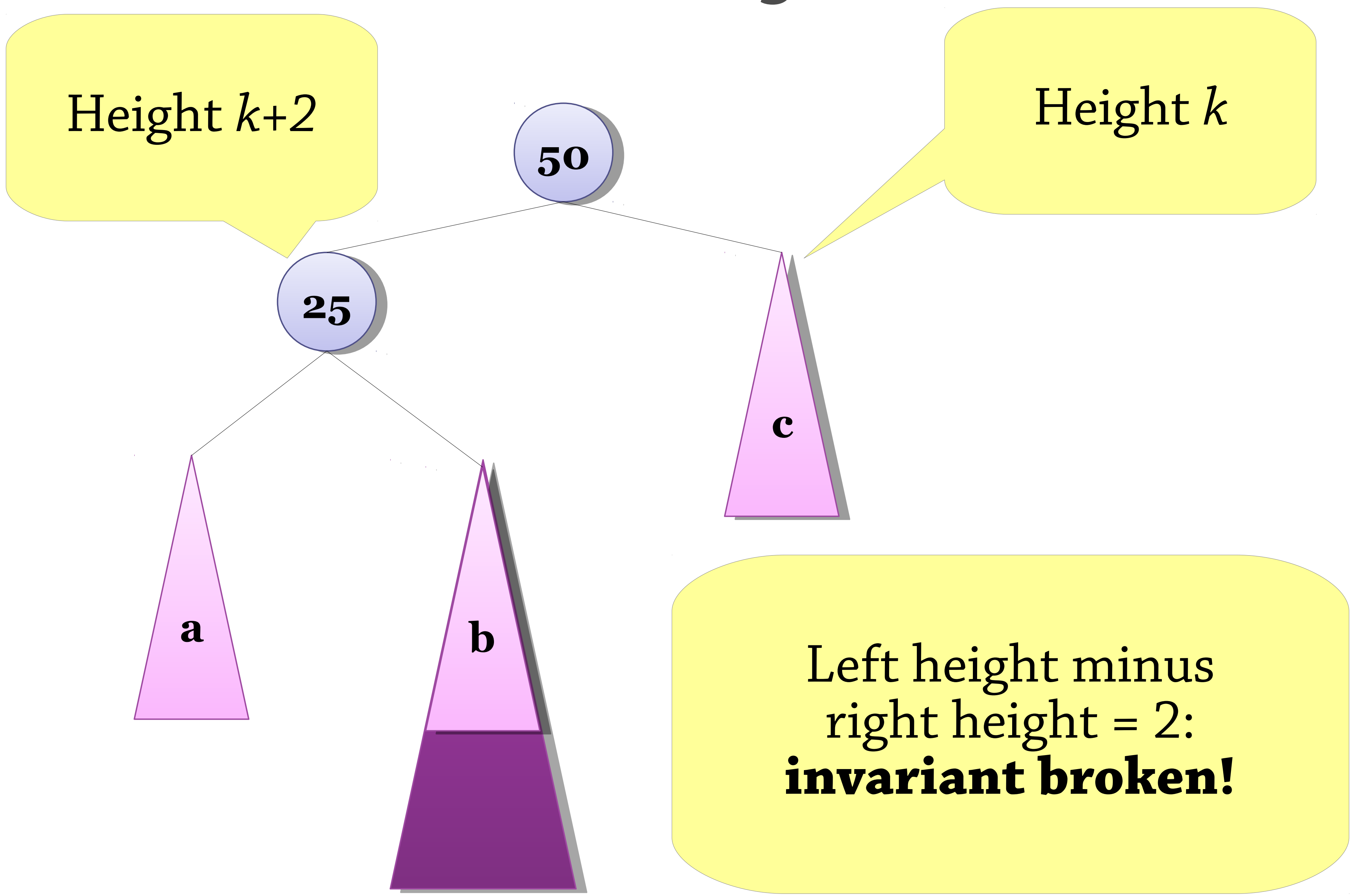


Case 2: a *right-right* tree

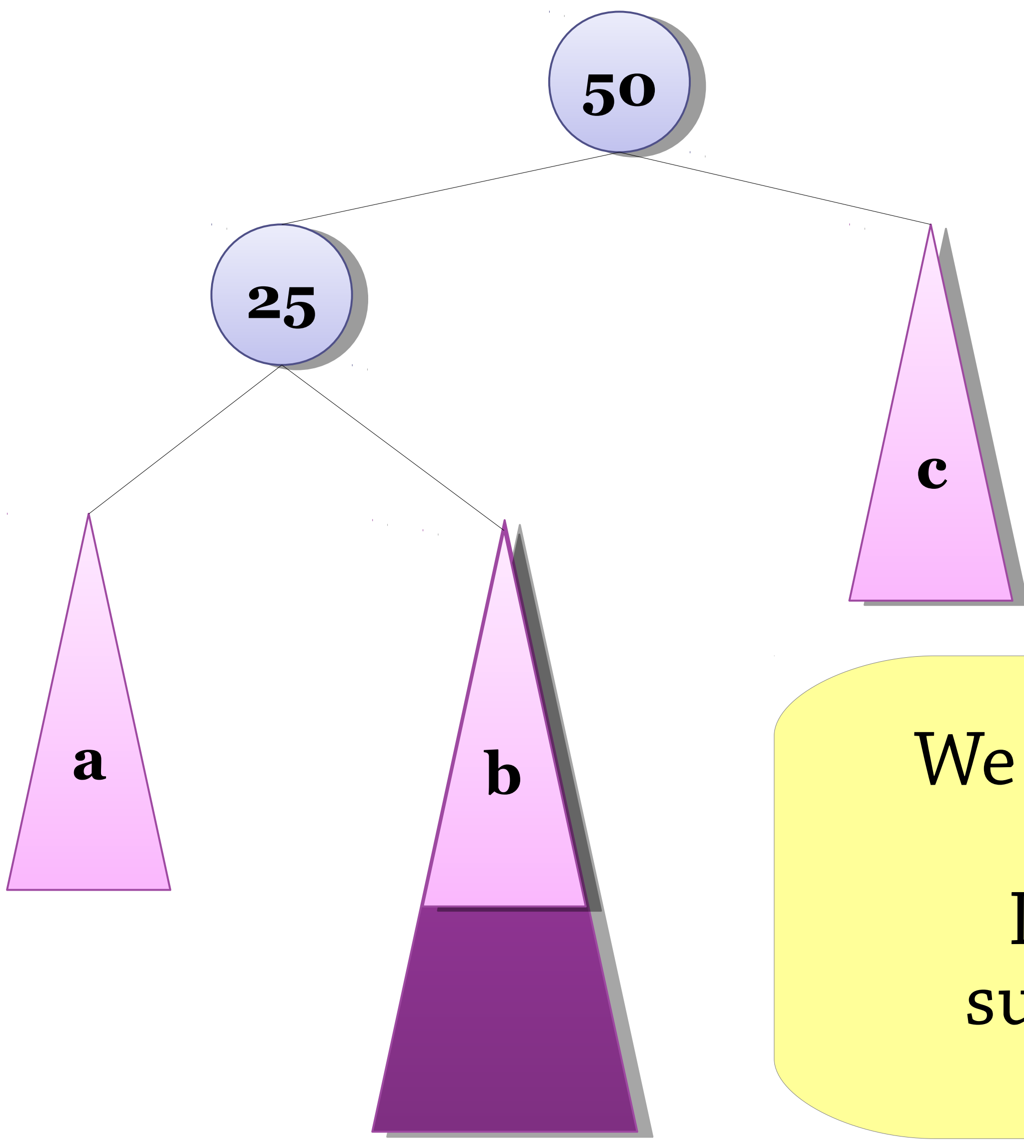


Mirror image of left-left tree
Can be fixed with
left rotation

Case 3: a *left-right* tree

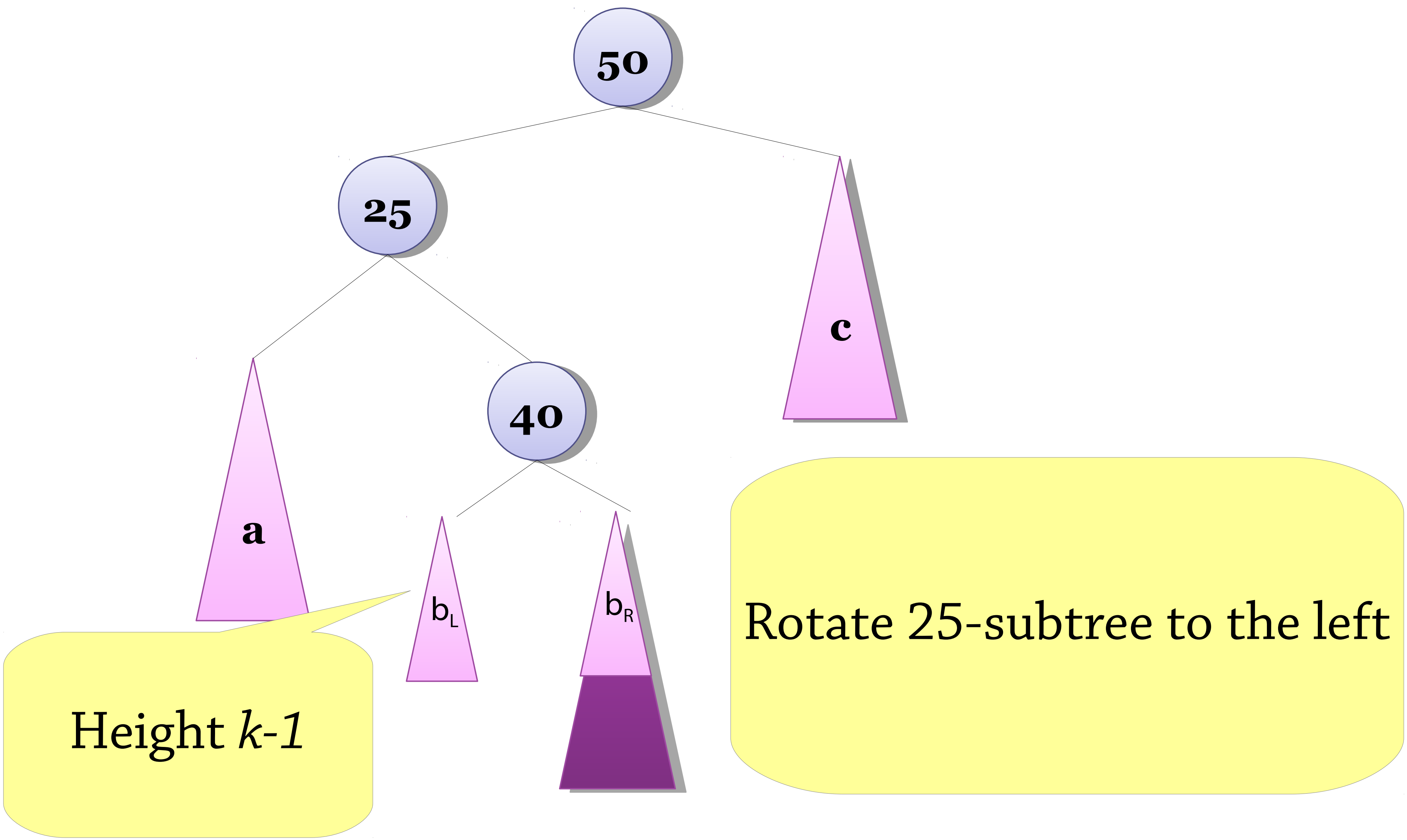


Case 3: a *left-right* tree

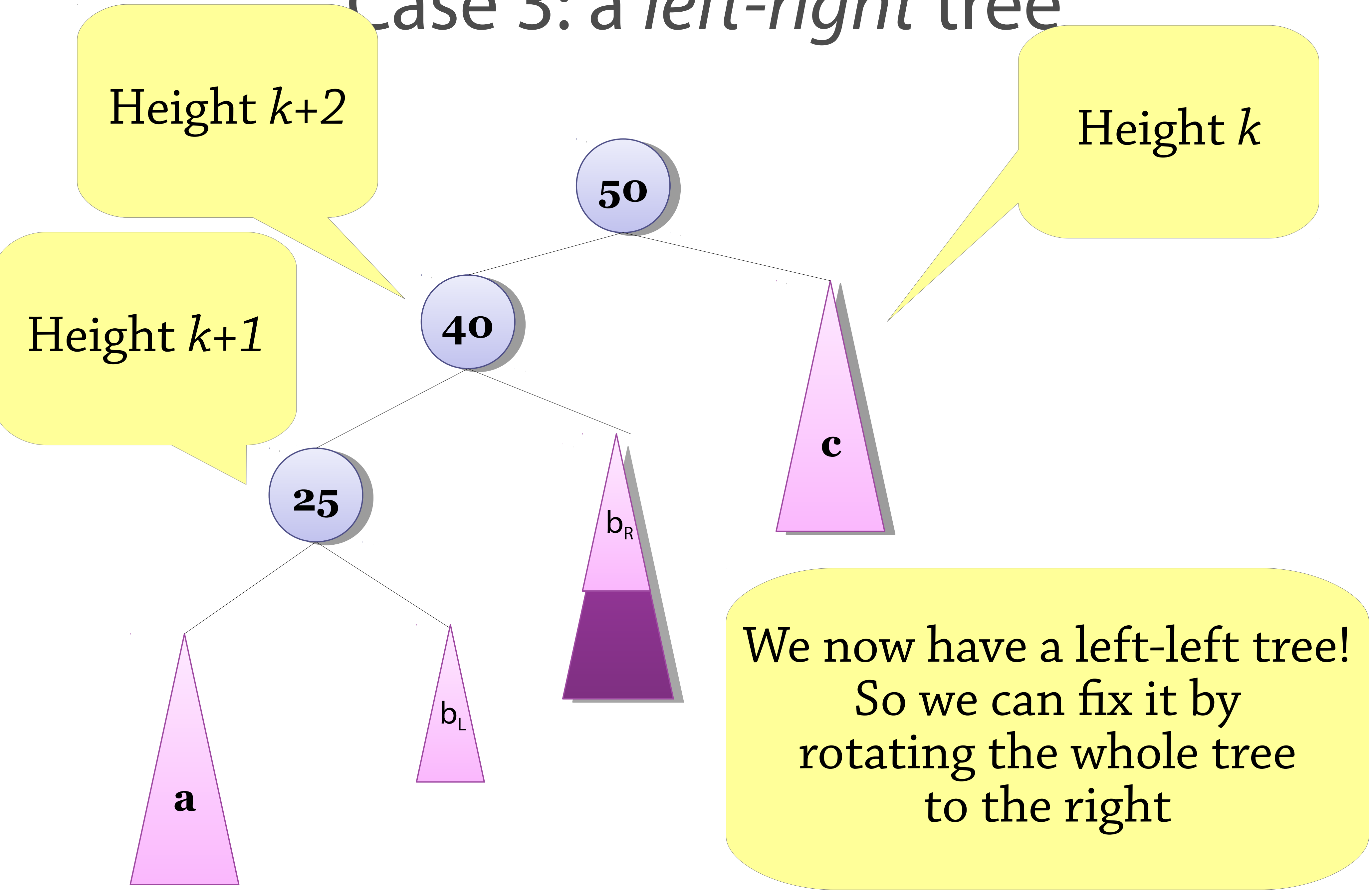


We can't fix this with one rotation
Let's look at b's subtrees b_L and b_R

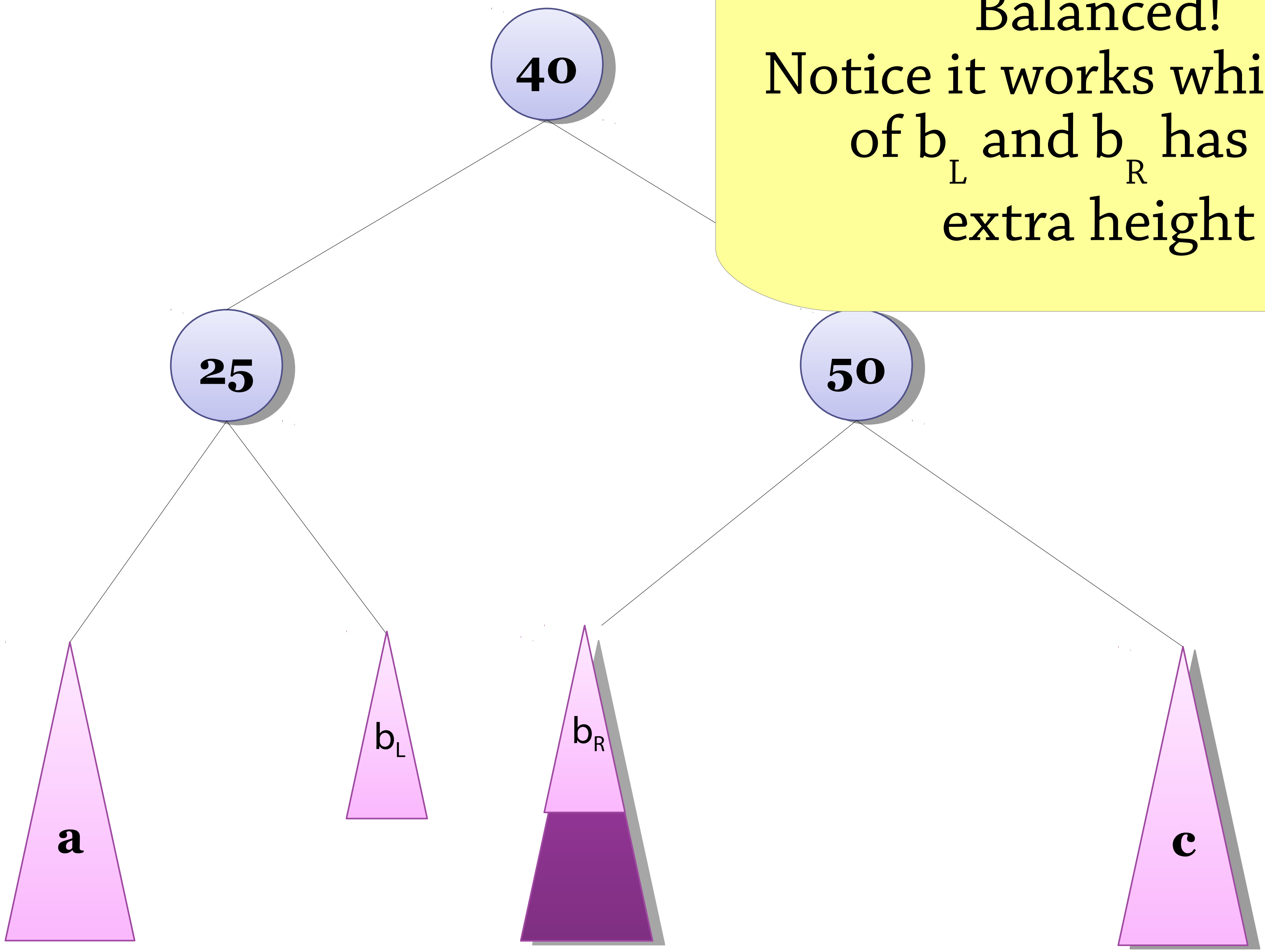
Case 3: a *left-right* tree



Case 3: a *left-right* tree

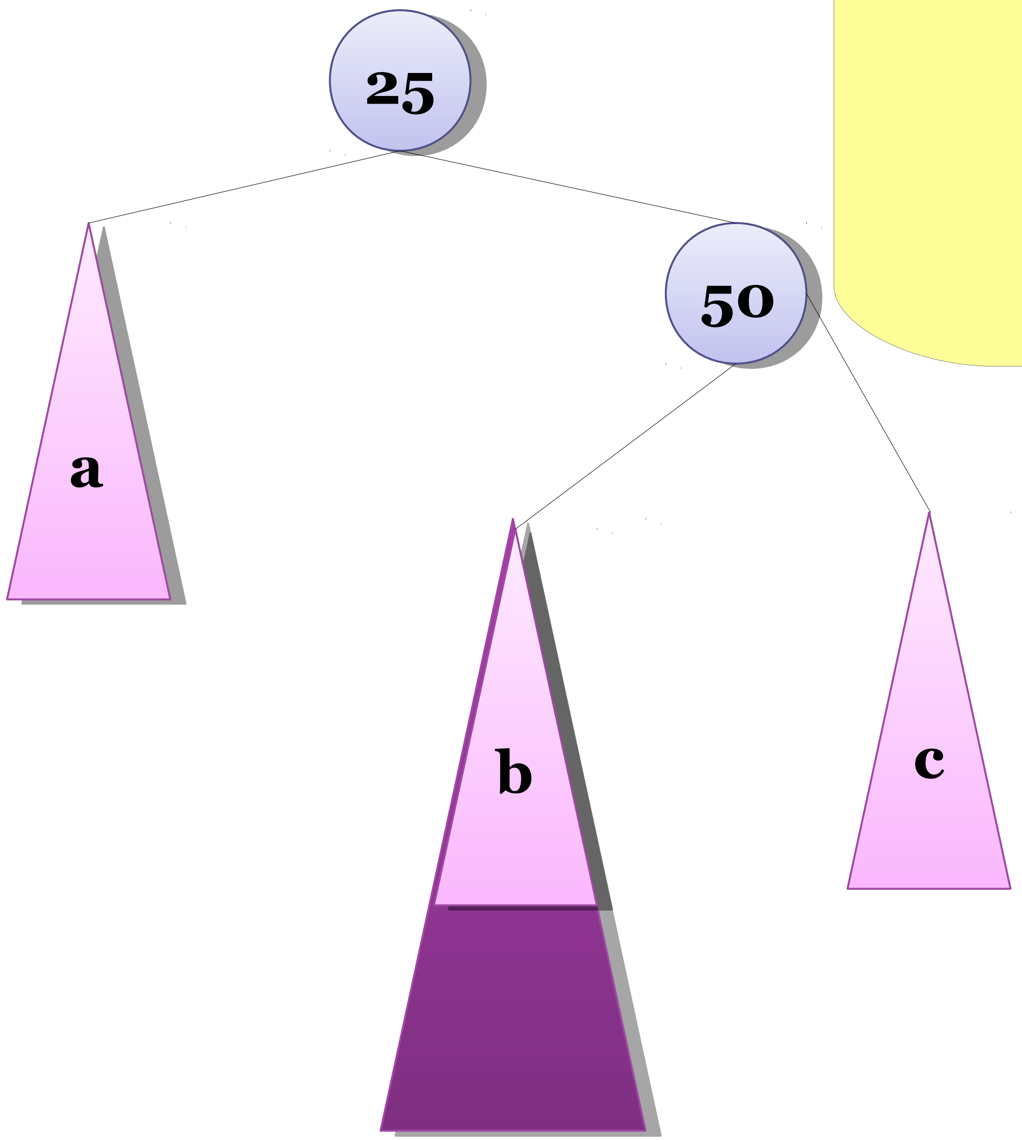


Case 3: a *left-right* tree



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

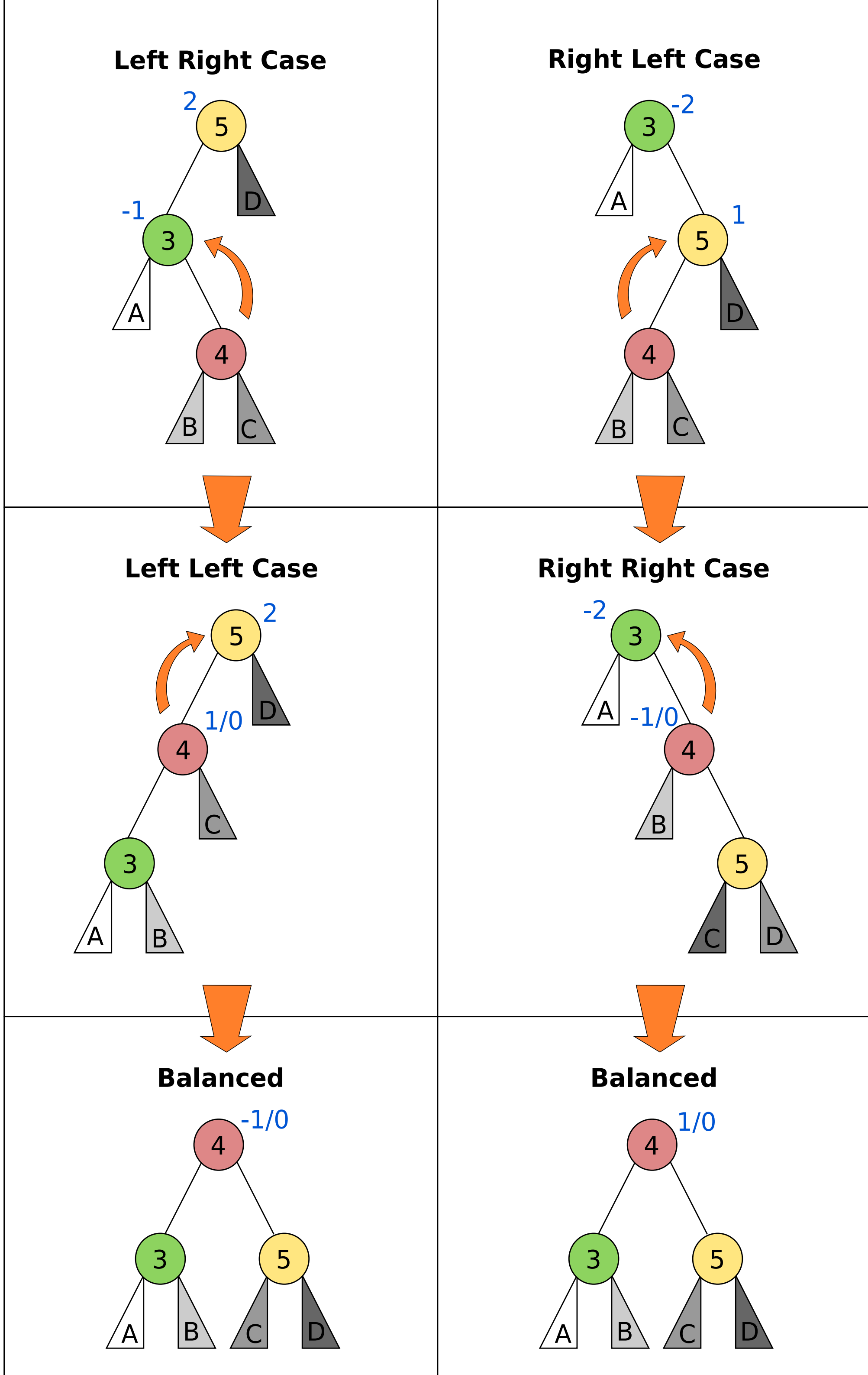
Right-left and right-right: symmetric

The four cases

(picture from Wikipedia)

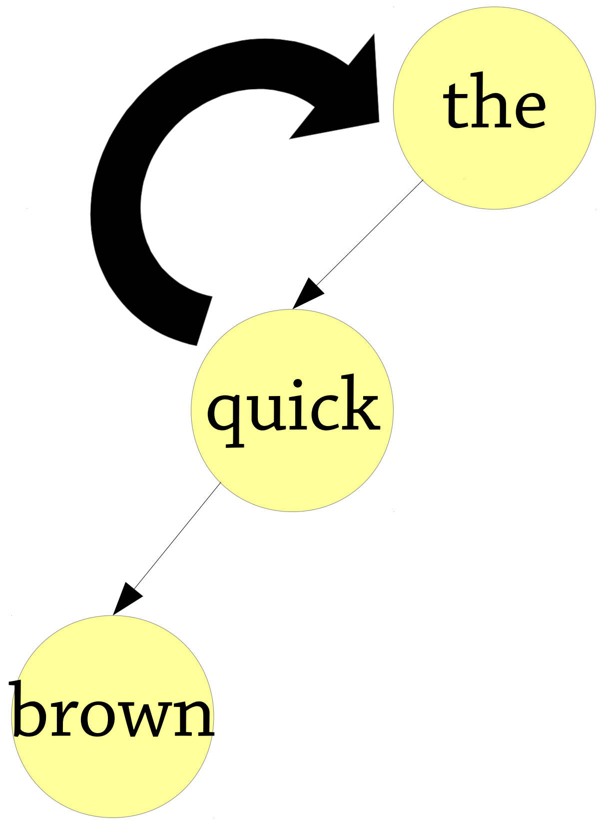
The numbers in the diagram show the *balance* of the tree: left height minus right height

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

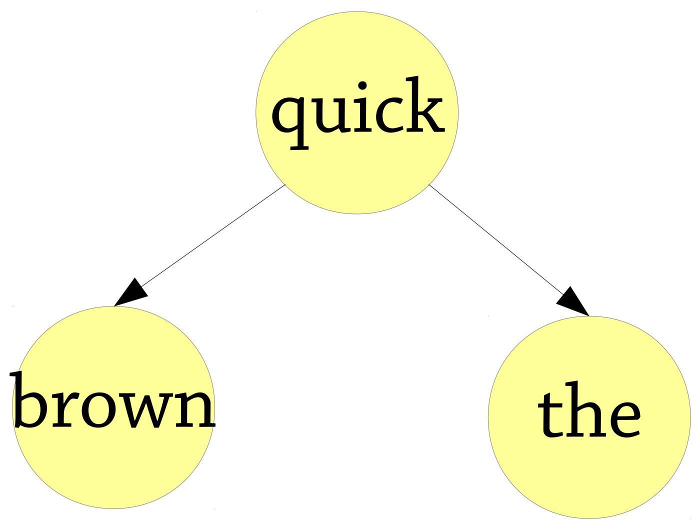
Insert “brown” into “the quick”



Left-left tree!
Rotate right

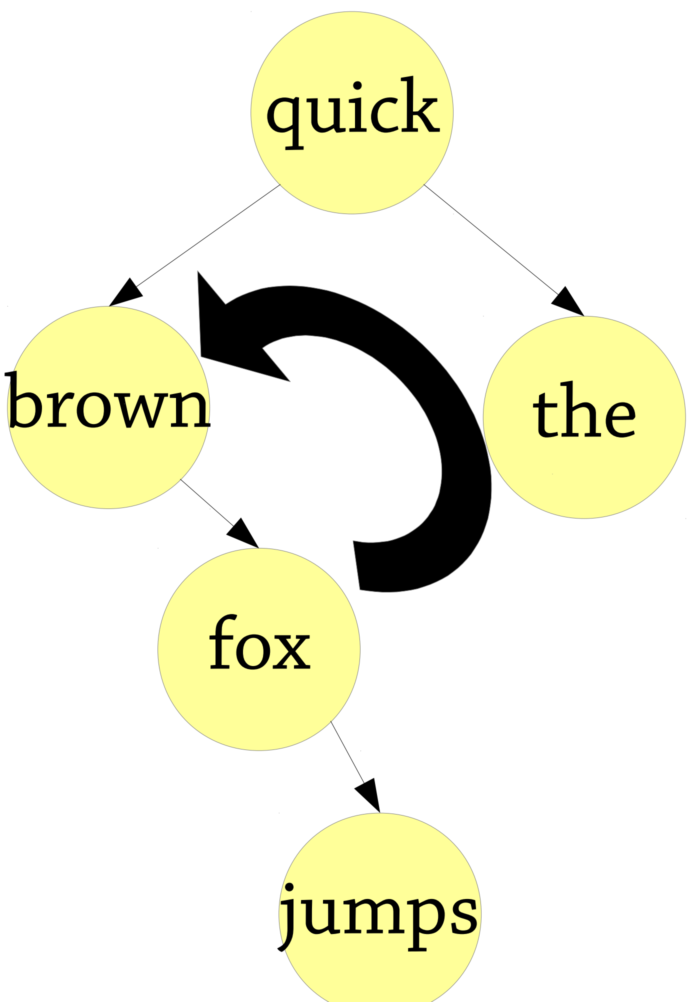
Example: the quick brown fox jumps over a lazy dog

Insert “brown” into “the quick”



Example: the quick brown fox jumps over a lazy dog

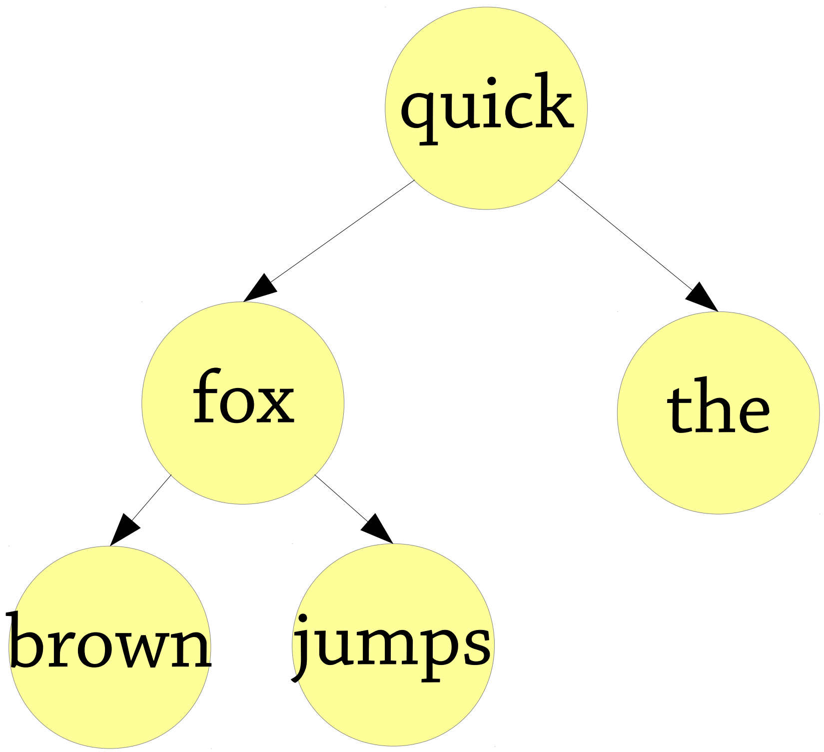
Insert “jumps” into “the quick brown fox”



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

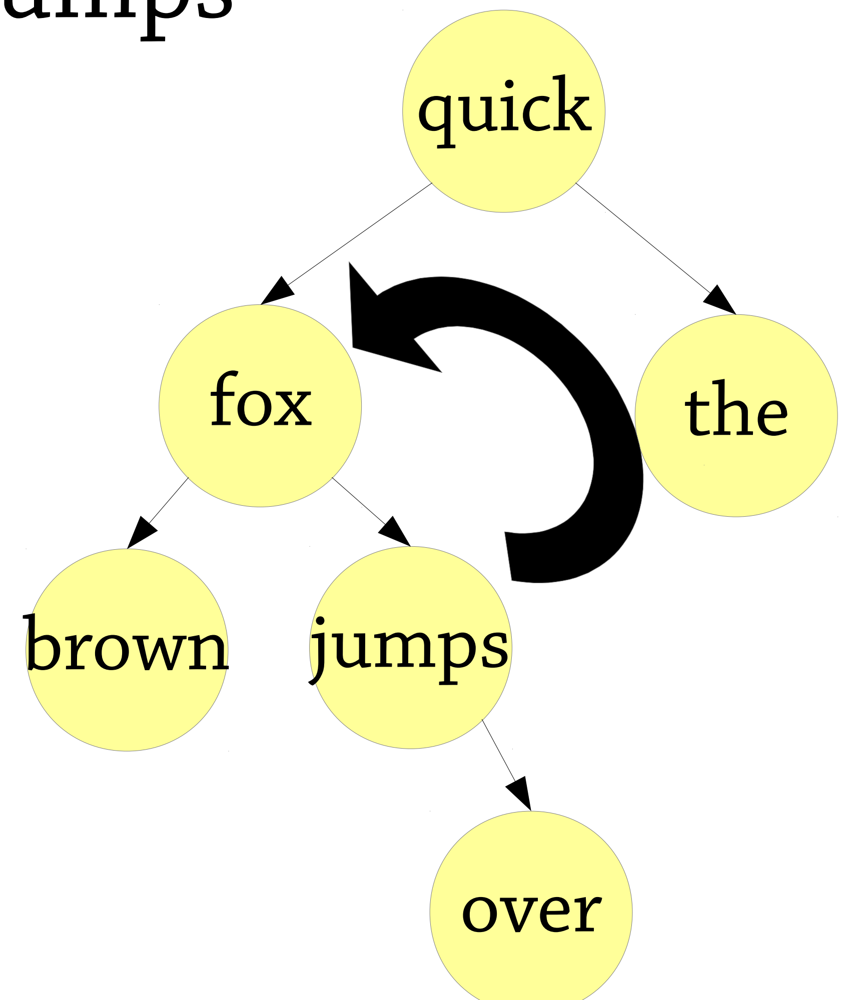
Example: the quick brown fox jumps over a lazy dog

Insert “jumps” into “the quick brown fox”



Example: the quick brown fox jumps over a lazy dog

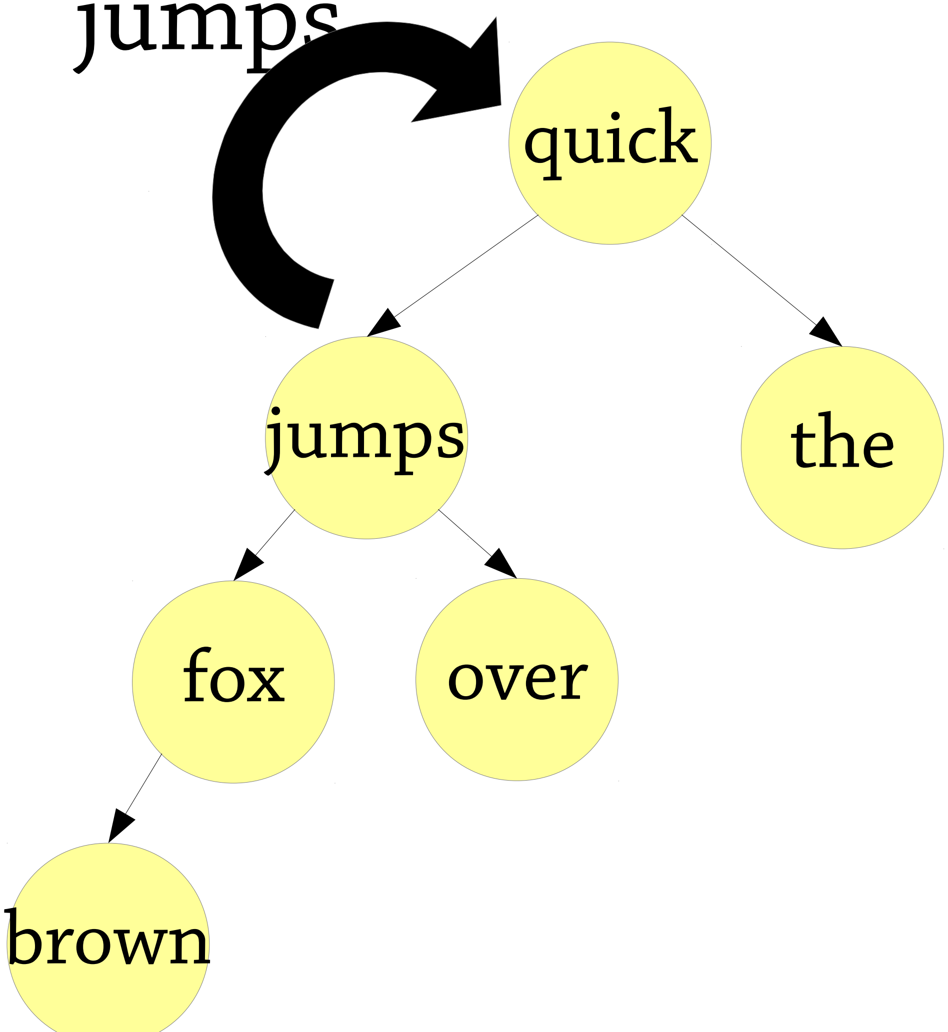
Insert "over" into "the quick brown fox jumps"



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

Example: the quick brown fox jumps over a lazy dog

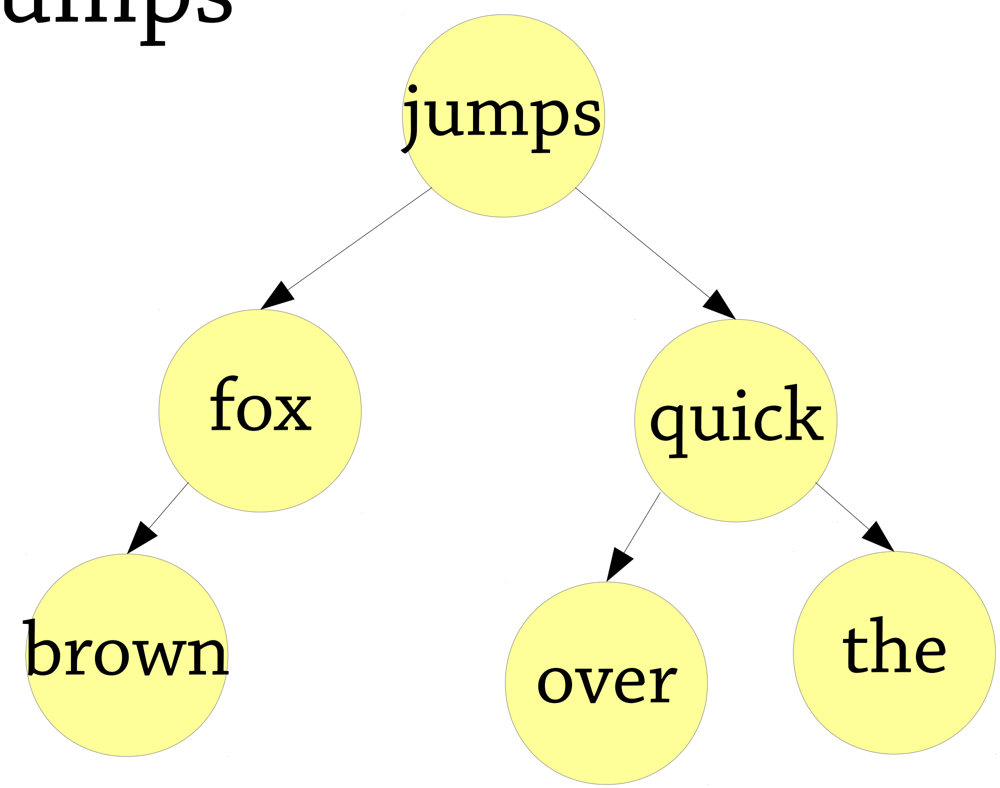
Insert "over" into "the quick brown fox jumps"



...then rotate quick right

Example: the quick brown fox jumps over a lazy dog

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



Lazy deletion **(not on exam)**

Deleting from a BST is quite hard…

deleting from an AVL tree is really complicated!

- Loads of cases, super annoying

Alternative: *lazy deletion*

Keep the node, mark it as deleted!

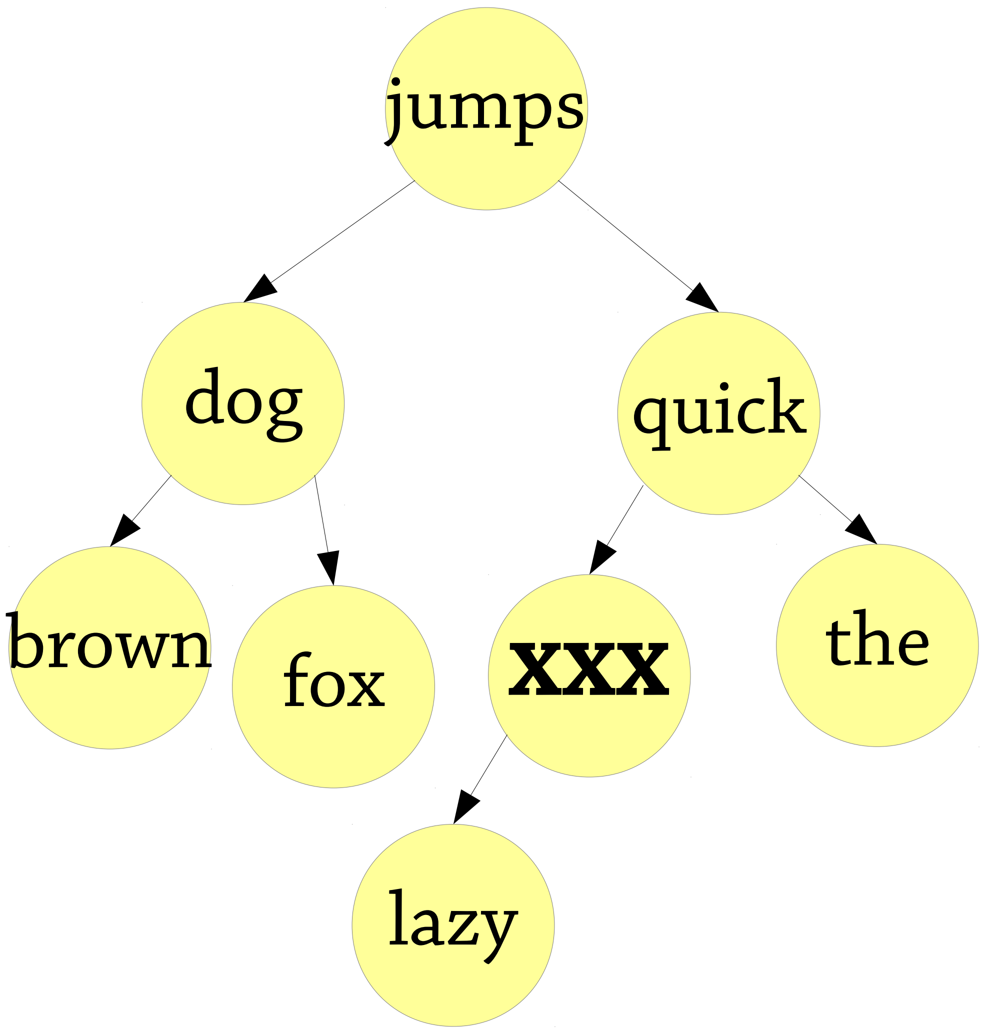
- Search simply skips over the node
- Opportunistically re-use the node in insertion

If you mark a node with two children as deleted, searching can become expensive

- Need to search both children
- Use same trick as from BST deletion to handle this case!

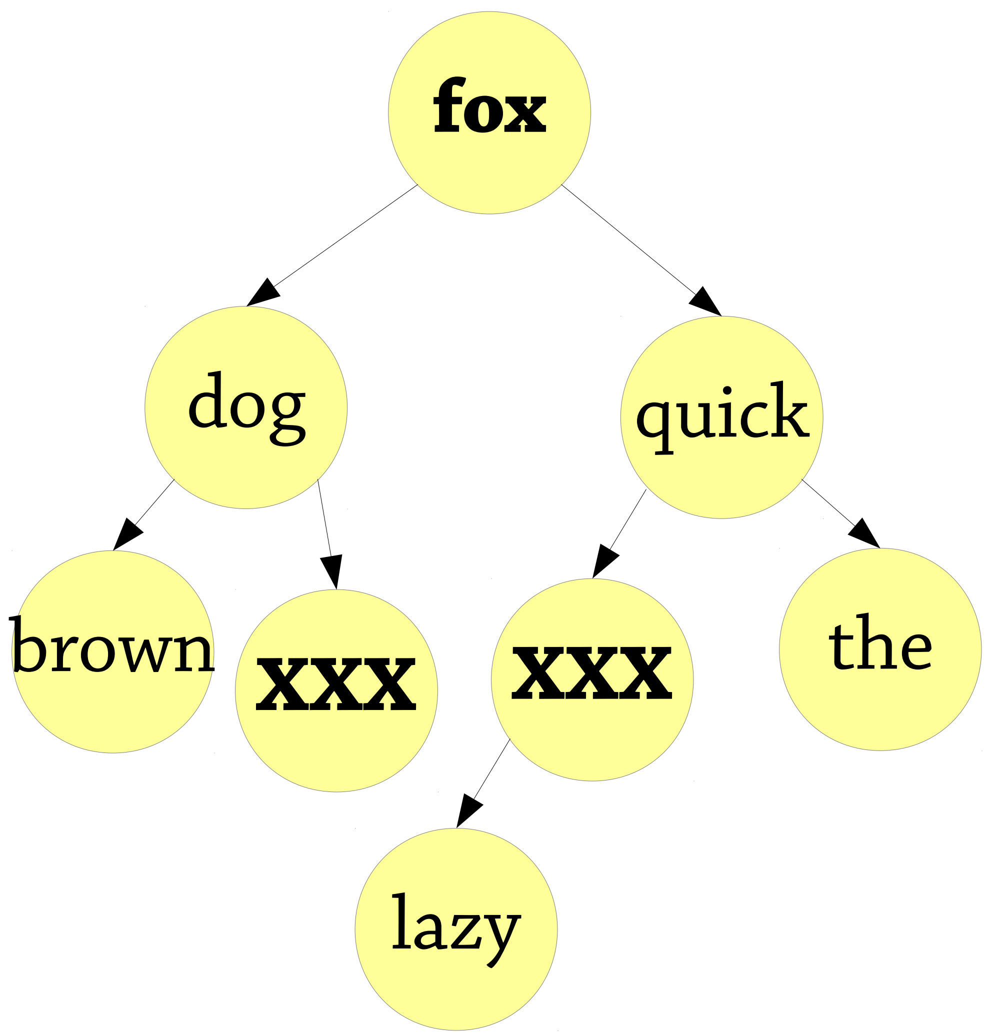
Example: the quick brown fox jumps over a lazy dog **(not on exam)**

Deleting "over":



Example: the quick brown fox jumps over a lazy dog **(not on exam)**

Deleting "jumps":



Lazy deletion **(not on exam)**

In the lecture we discovered that doing lazy deletion just like this *doesn't work*!

- E.g., deleting *fox* will now no longer preserve the invariant

I think the correct invariant is: a deleted node's children must be deleted

Exercise: work out how to do insertion and deletion to preserve this invariant!

AVL trees

Use *rotation* to keep the tree balanced

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

Insertion – BST insertion, then rotate to repair the invariant

Deletion (see Wikipedia if you're interested) – similar idea but a bit harder (more cases)

- or use lazy deletion

Implementation – see Haskell compendium on course website!

Visualisation:

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