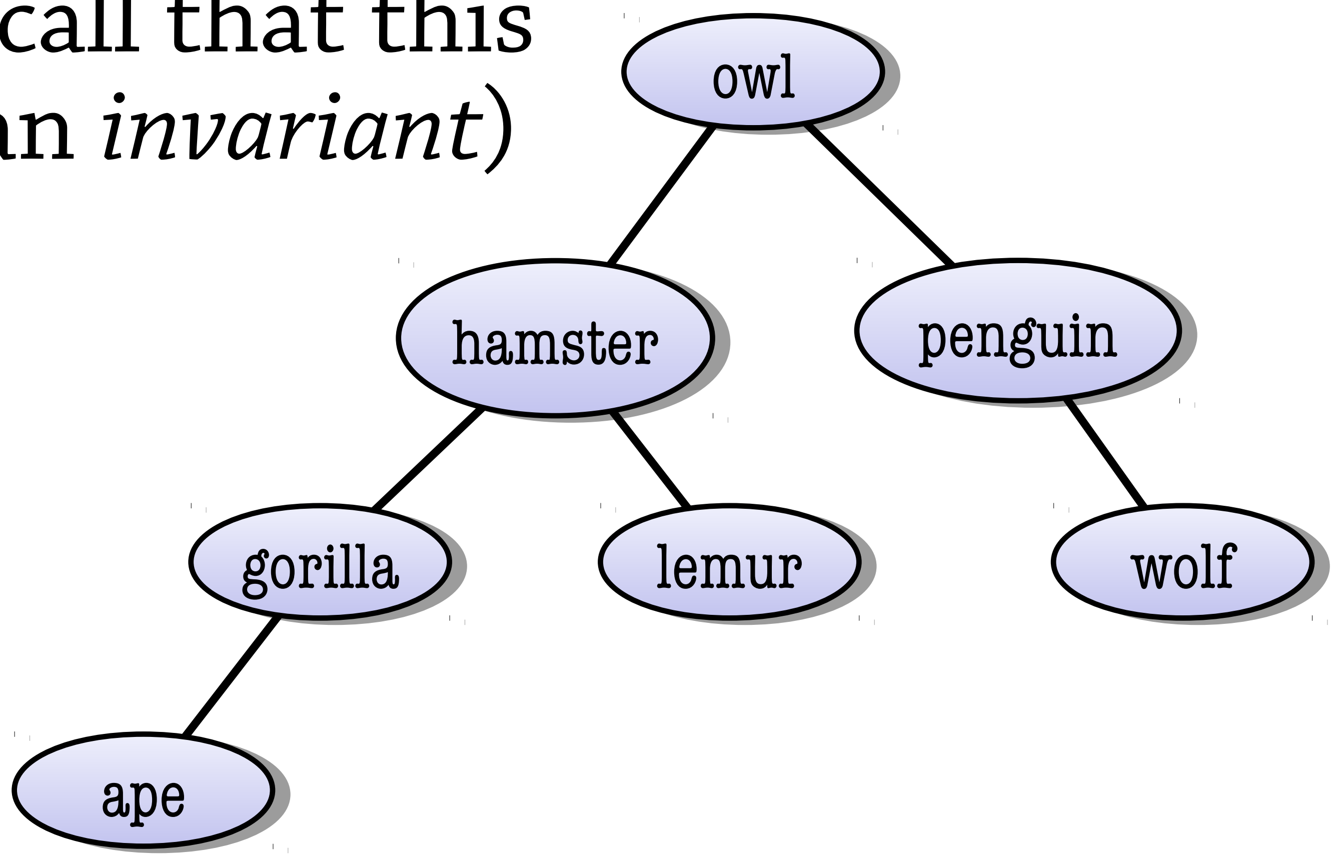
Binary search trees
(chapters 18.1 – 18.3)

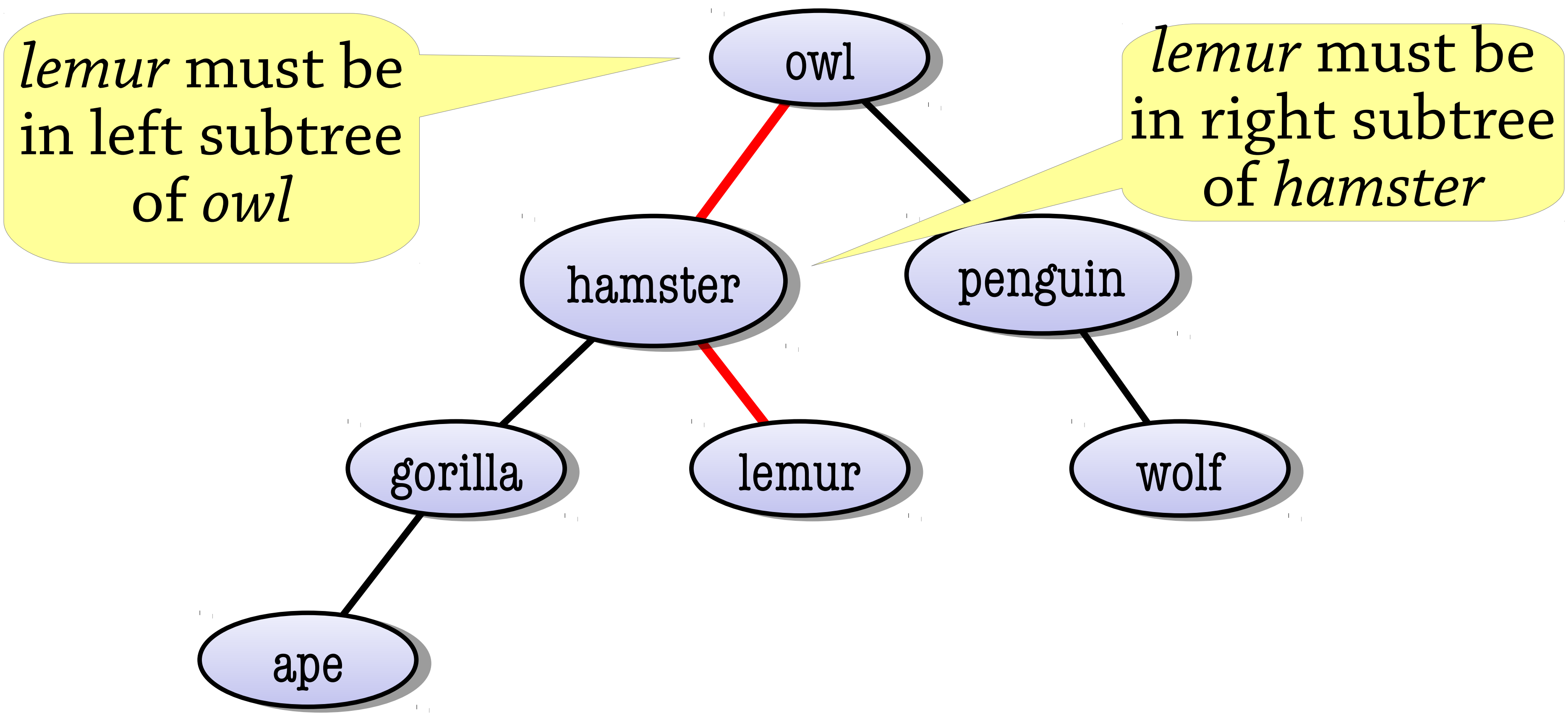
Binary search trees

In a *binary search tree* (BST), every node is greater than all its left descendants, and less than all its right descendants (recall that this is an *invariant*)



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:

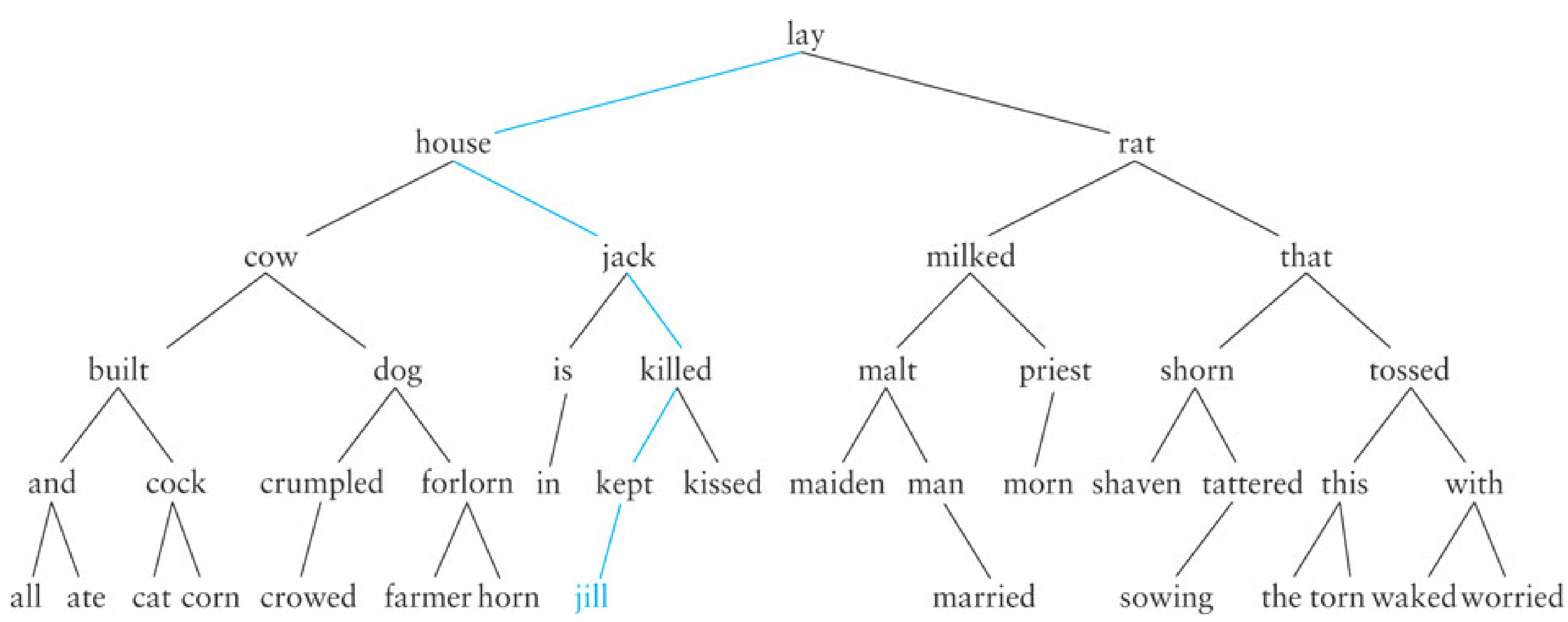
- If the target matches the root node's data, we've found it
- If the target is *less* than the root node's data, recursively search the left subtree
- If the target is *greater* than the root node's data, recursively search the right subtree
- If the tree is empty, fail

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

Inserting into a BST

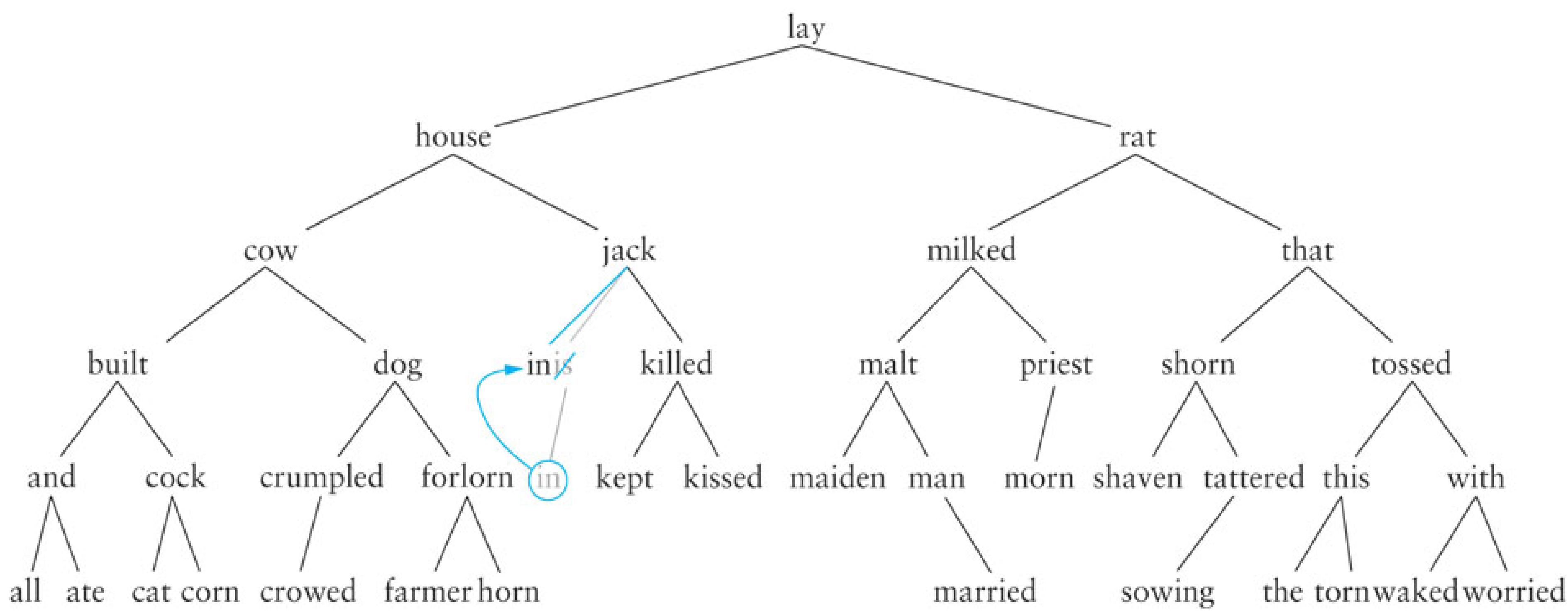
To insert a value into a BST:

- 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 a node with one child

Deleting "is", which has one child, "in" – we connect "in" to is's parent "jack"



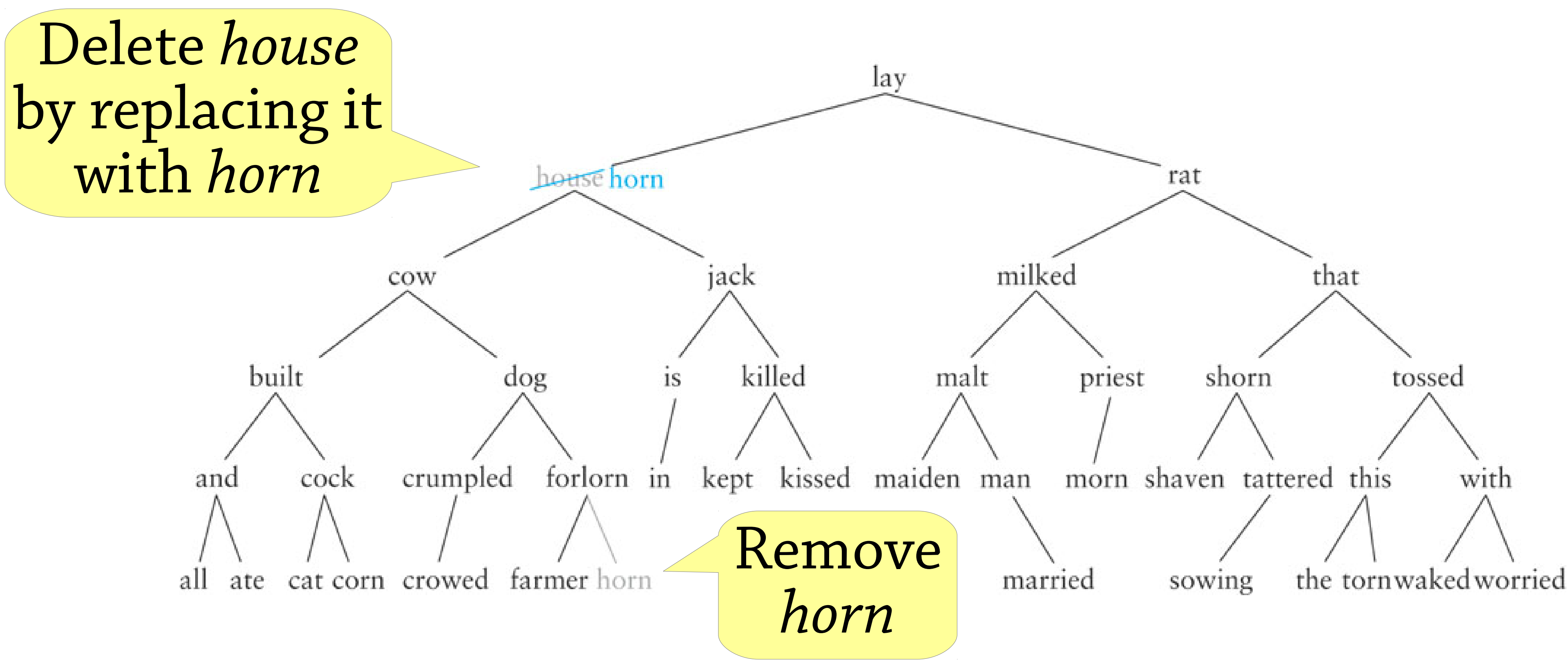
Deleting from a BST

To delete a value from a BST:

- Find the node and its parent
- If it has no children, just remove it from the tree (by disconnecting it from its parent)
- If it has one child, replace the node with its child (by making the node's parent point at the child)
- If it has two children…?

Deleting a node with two children

Replace the deleted value with *the biggest value from its left subtree* (or the smallest from the right subtree) [why this one?]



Deleting a node with two children

Find the node to delete

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

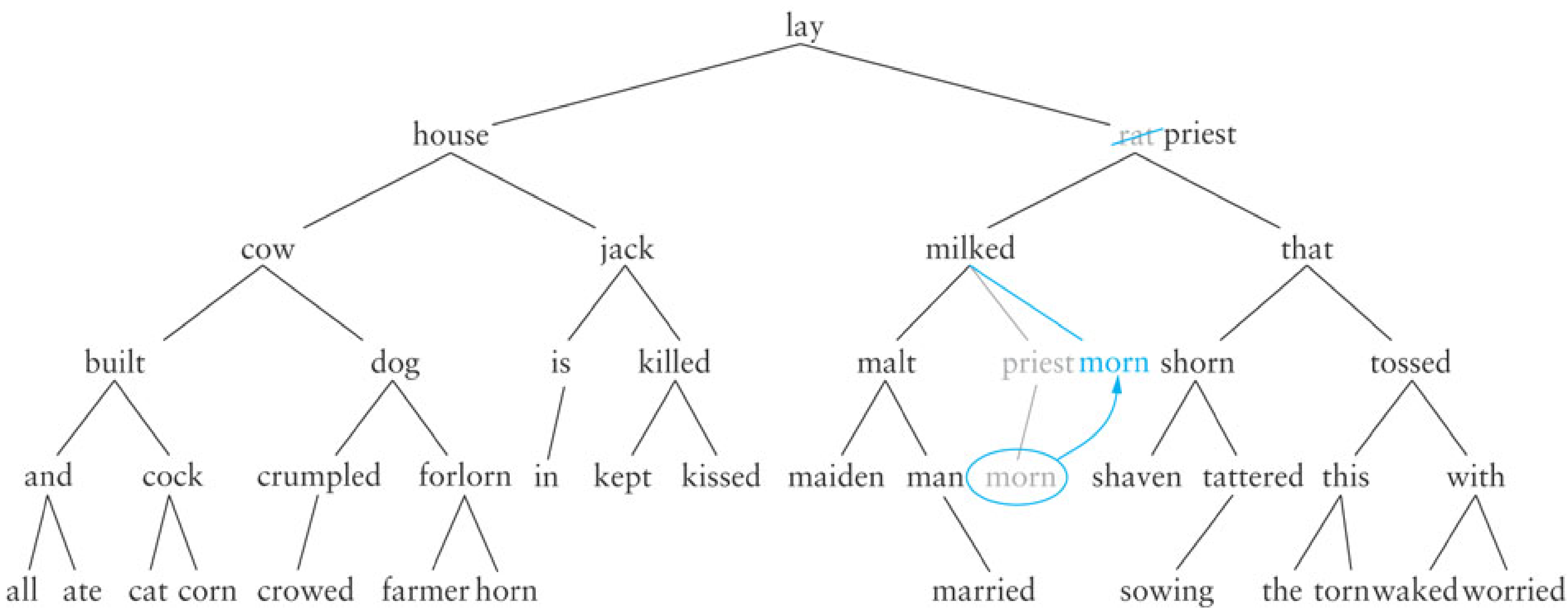
- Using the biggest value preserves the invariant (check you understand why)
- Biggest node = rightmost node

Finally, delete the biggest value from the left subtree

- This node can't have two children (no right child), so deleting it is much easier

Deleting a node with two children

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

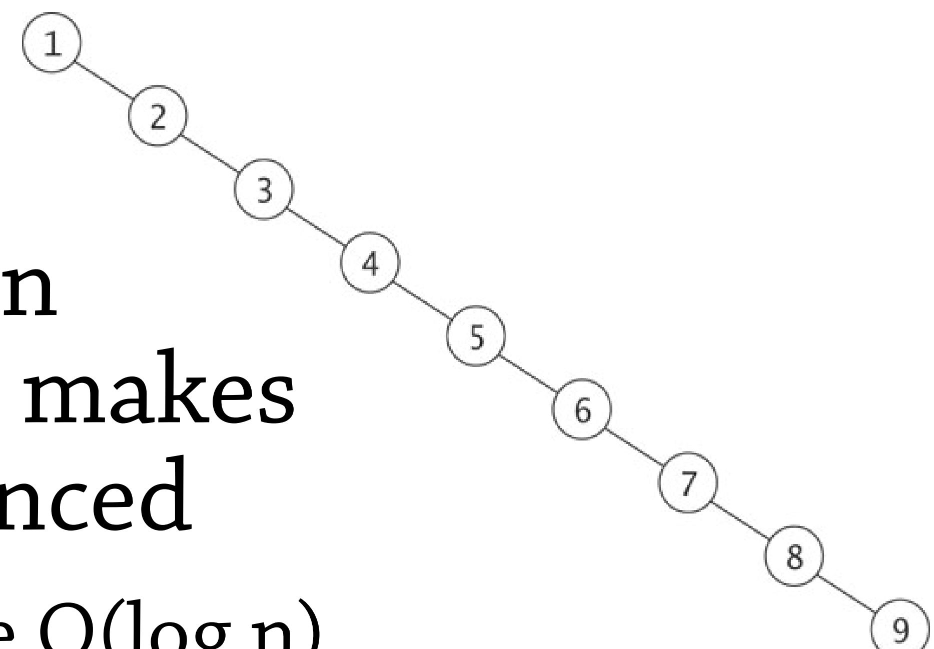


Complexity of BST operations

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 terminate
- delete: find the value, then 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

Tree traversal

Traversing a tree means visiting all its nodes in some order

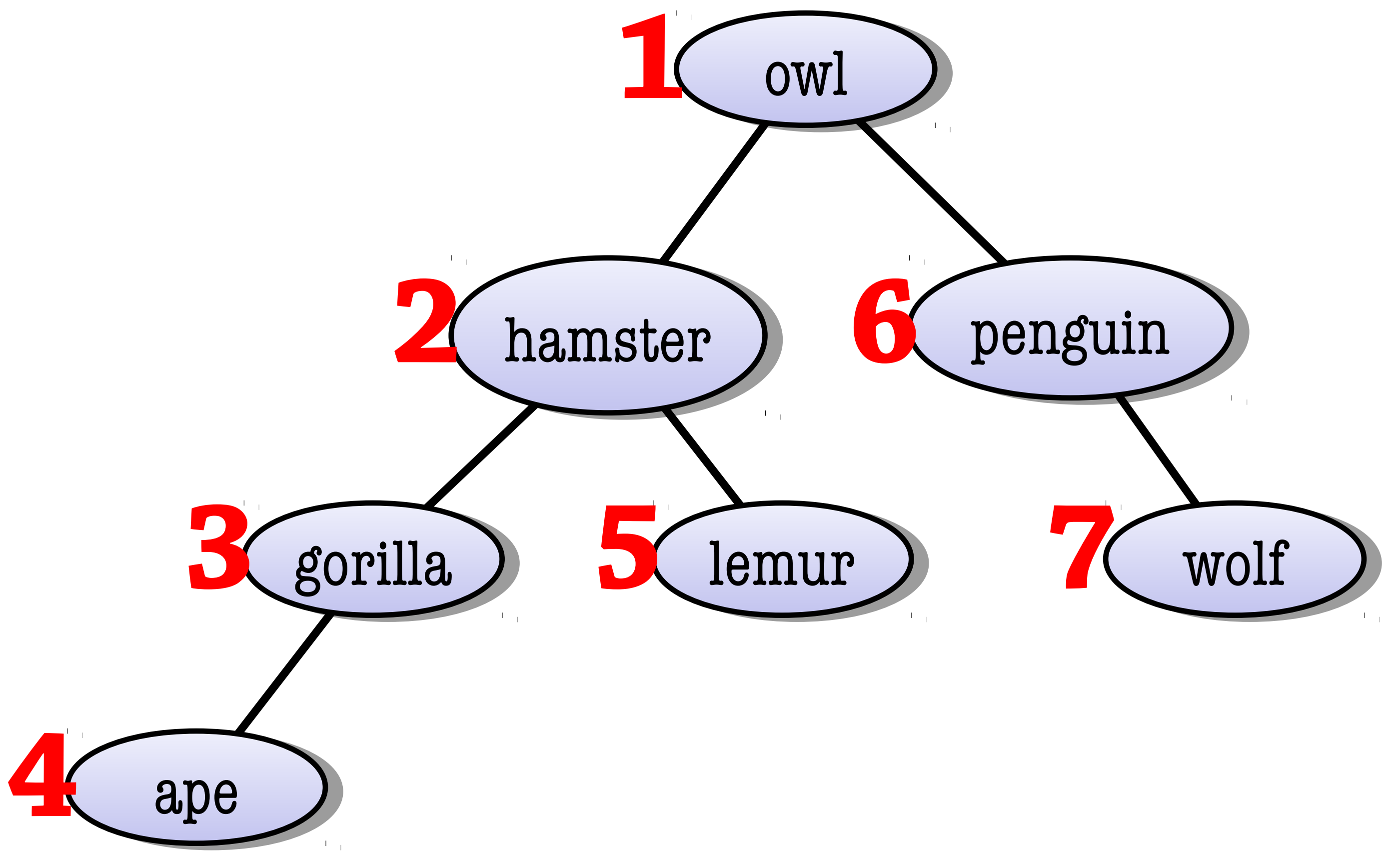
A *traversal* is a particular order that we visit the nodes in

Four common traversals: preorder, inorder, postorder, level-order

For each traversal, you can define an iterator that traverses the nodes in that order (see 17.4)

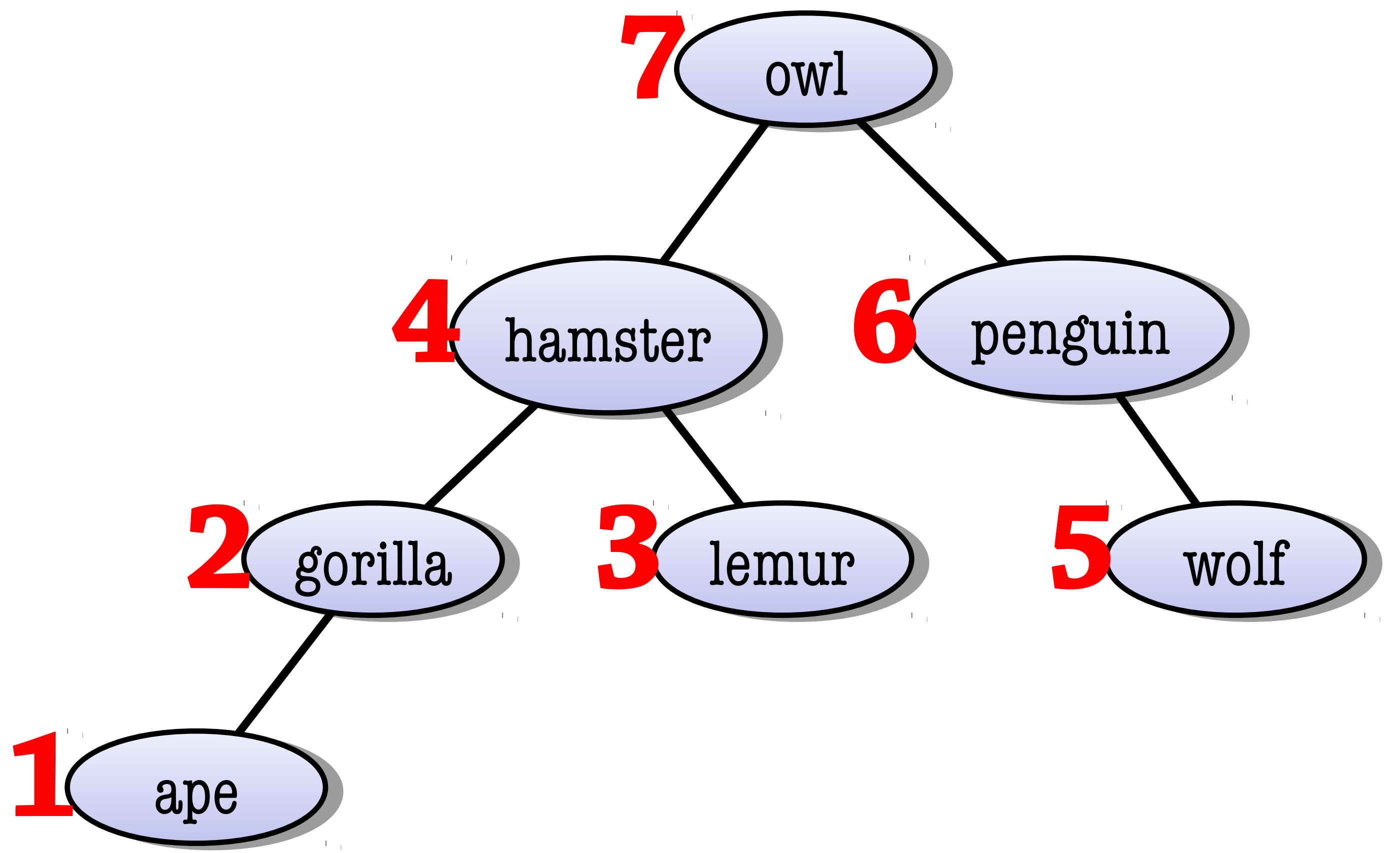
Preorder traversal

Visit root node, then left child, then right



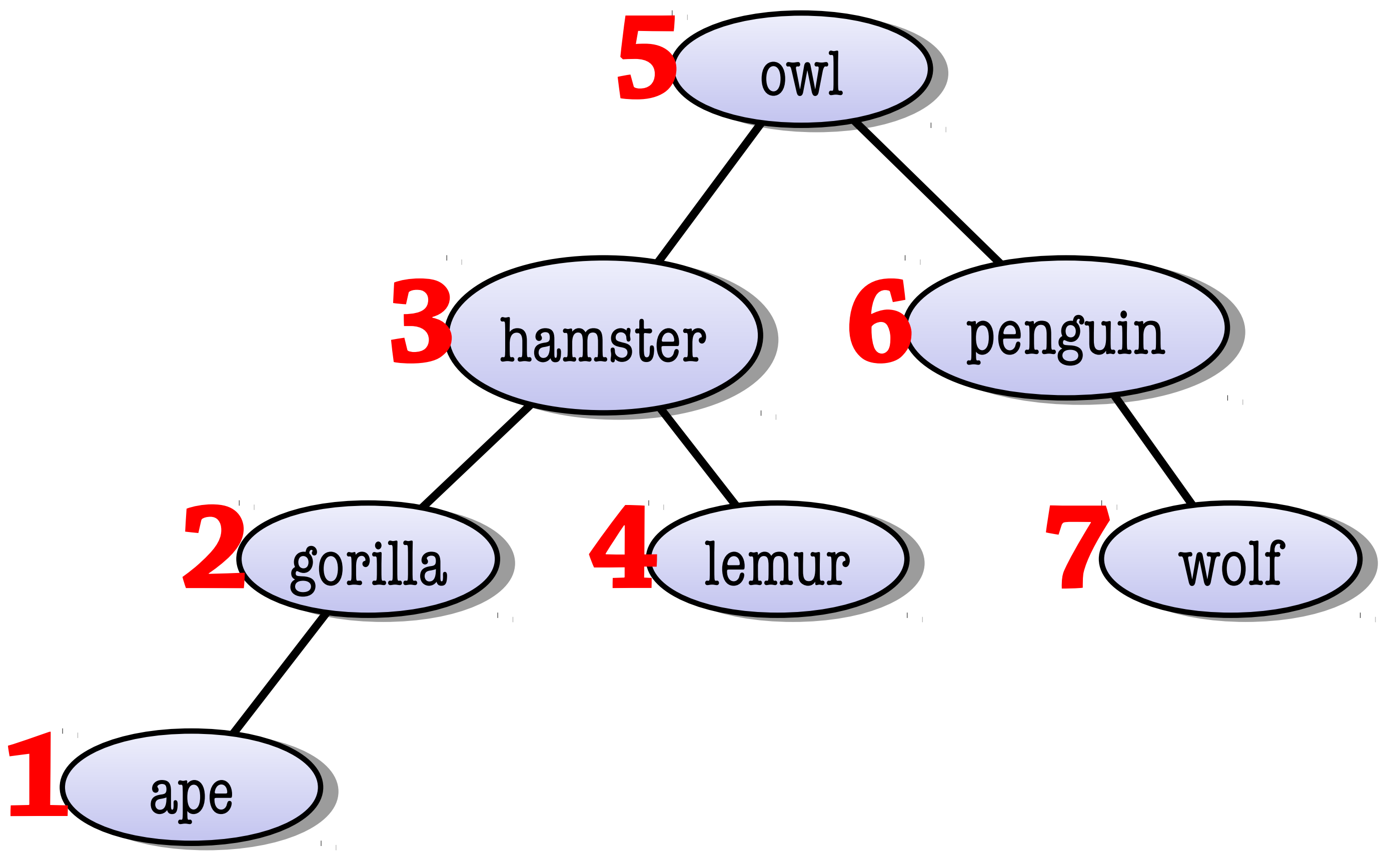
Postorder traversal

Visit left child, then right, then root node



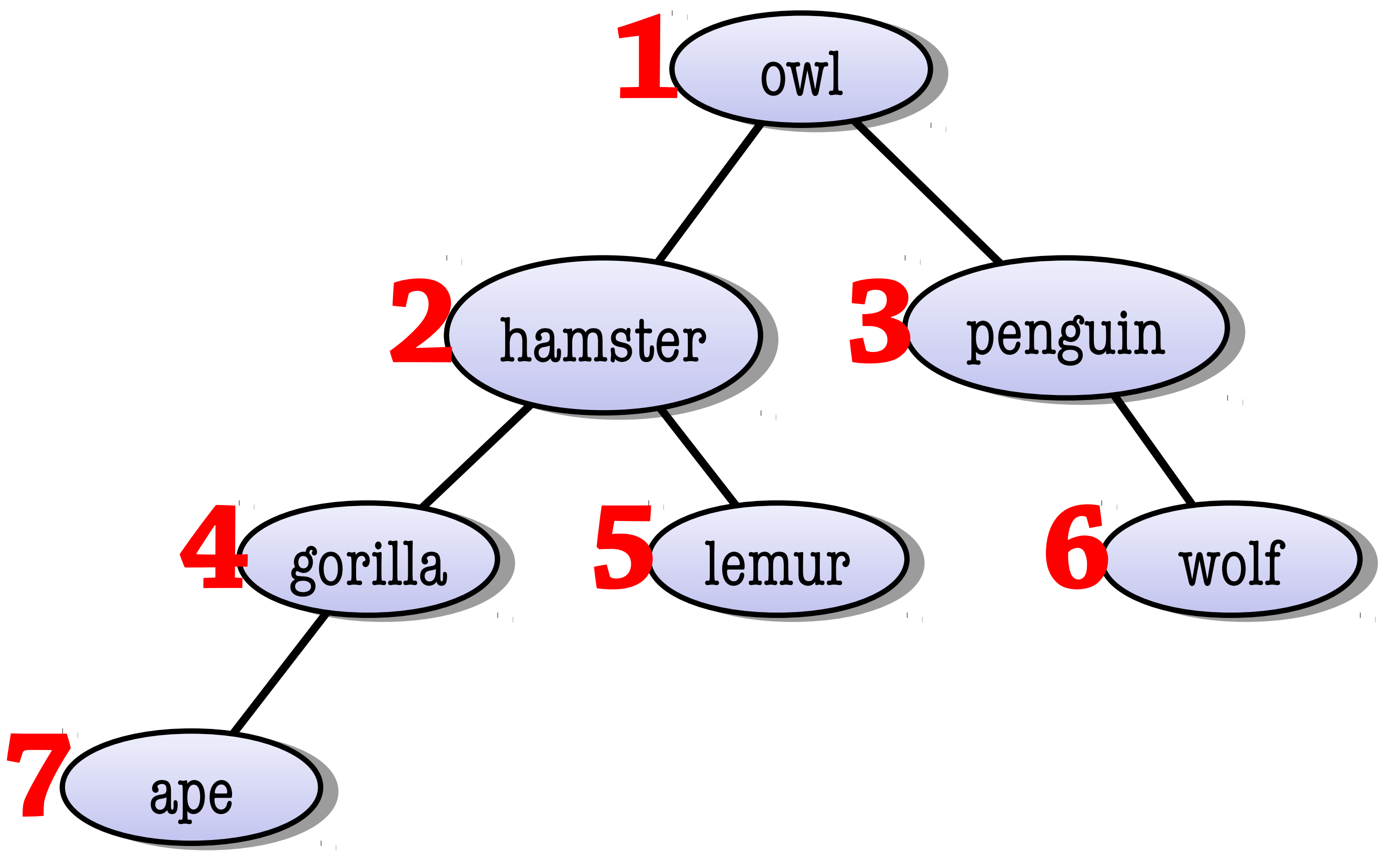
Inorder traversal

Visit left child, then root node, then right



Level-order traversal

Visit nodes left to right, top to bottom



In-order traversal – printing

```
void inorder(Node<E> node) {
  if (node == null) return;
  inorder(node.left);
  System.out.println(node.value);
  inorder(node.value);
}
```

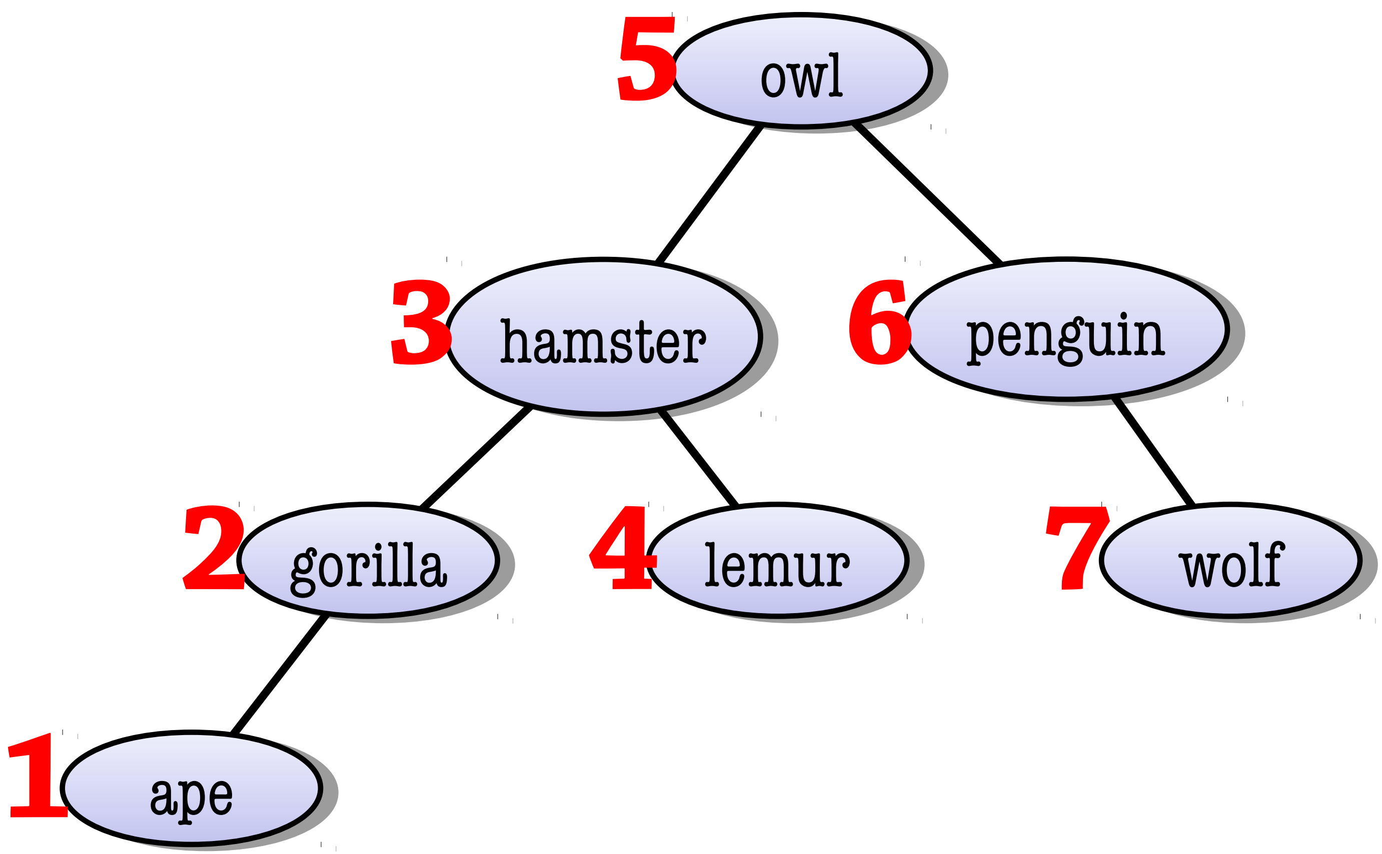
But nicer to define an iterator!

```
Iterator<Node<E>> inorder(Node<E>
node);
```

Level-order traversal is slightly trickier, and uses a queue – see 17.4.4

Sorting a binary search tree

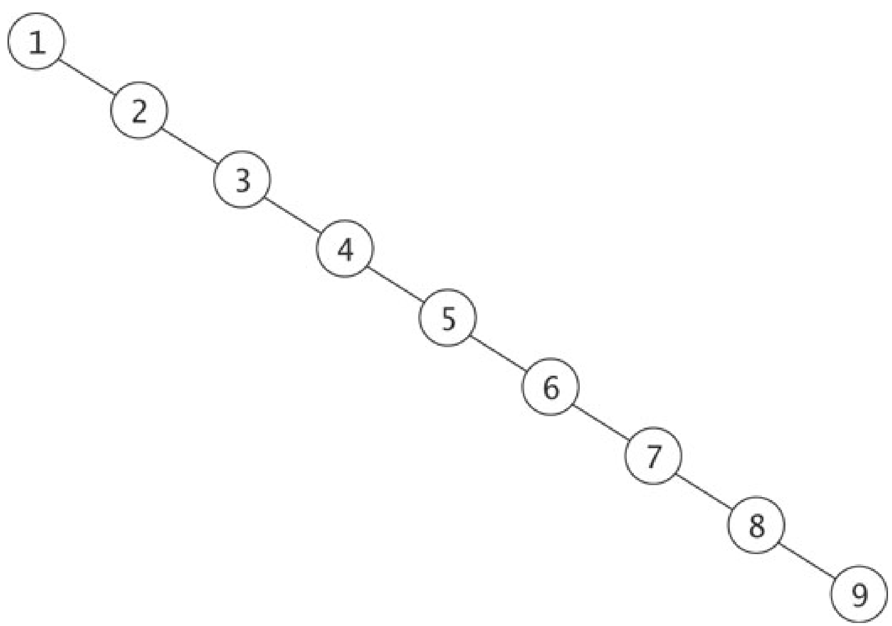
If we do an inorder traversal of a BST, we get its elements in sorted order!

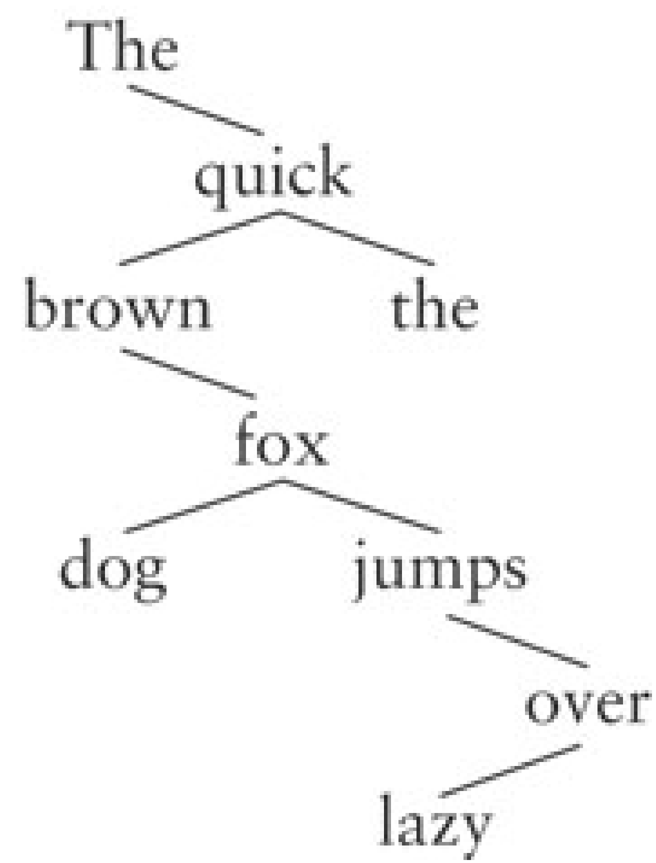


AVL trees
(chapter 18.4)

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:

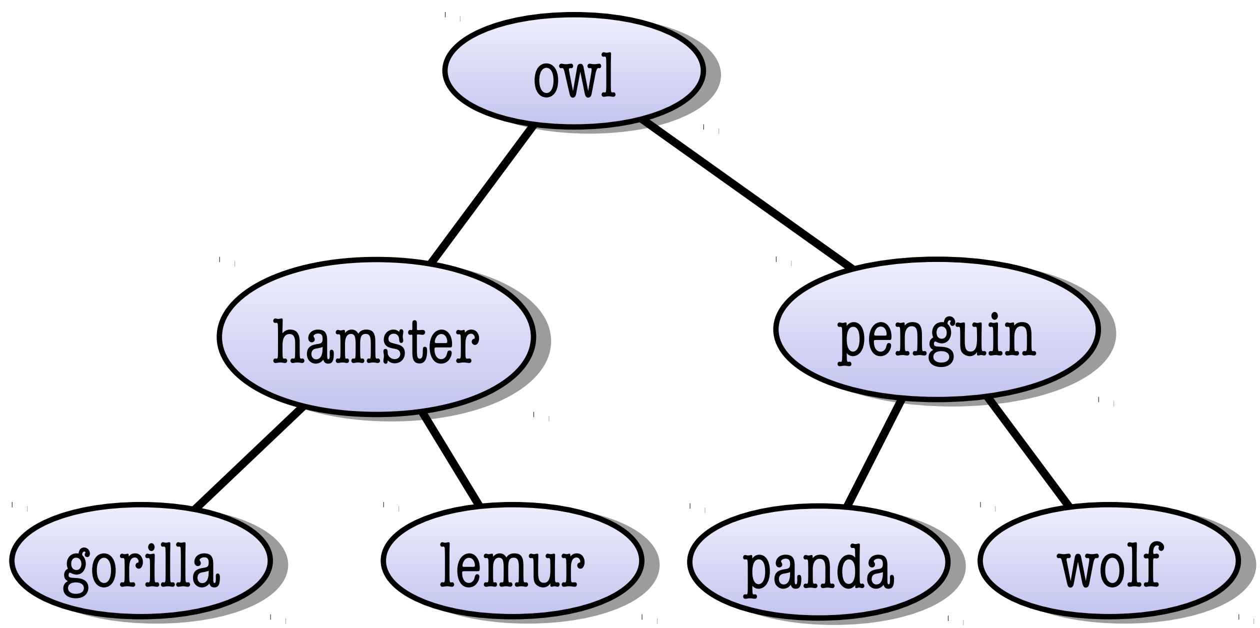
- Height of left child = height of right child (for all nodes in the tree)
- Tree would be sort of "perfectly balanced"

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

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

AVL trees

We call the quantity *right height – left height* of a node its *balance*

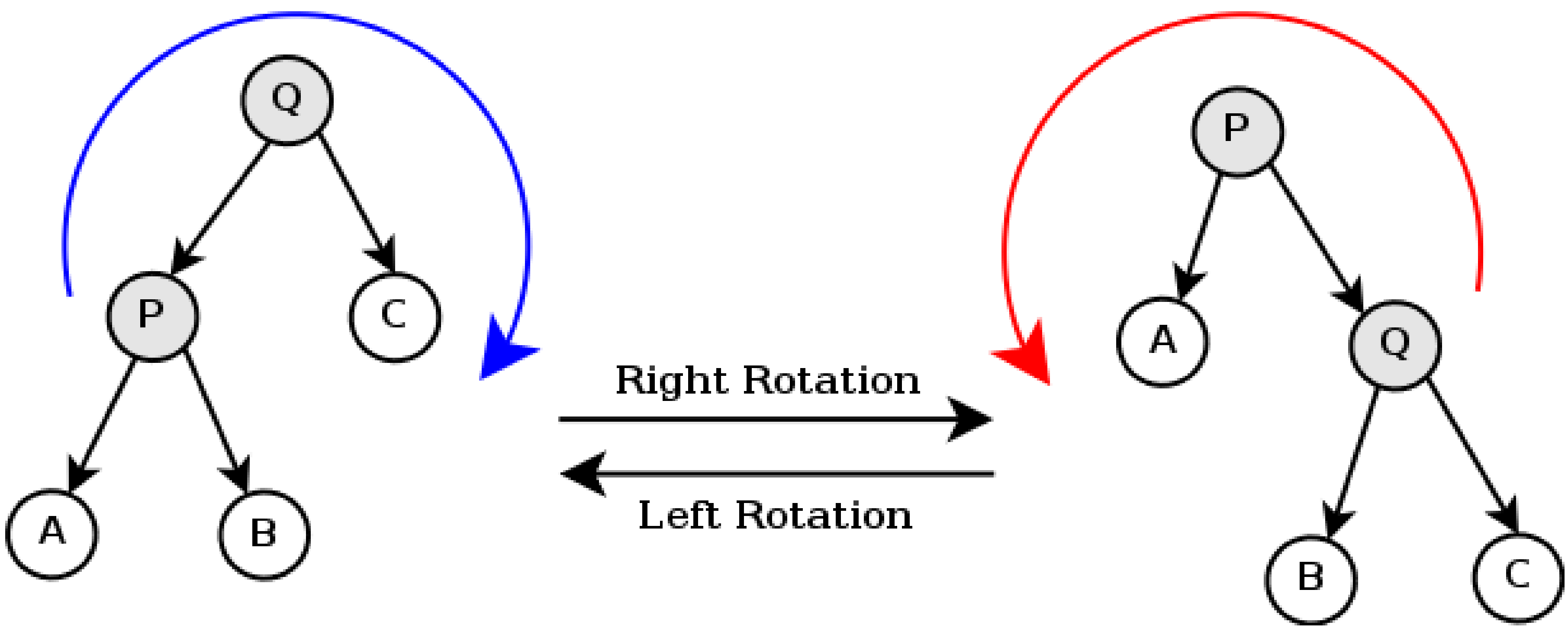
Thus the AVL invariant is: the balance of every node is -1, 0, or 1

Whenever a node gets out of balance, we fix it with so-called *tree rotations* (next)

(Implementation: store the balance of each node as a field in the node, and remember to update it when updating the tree)

Rotation

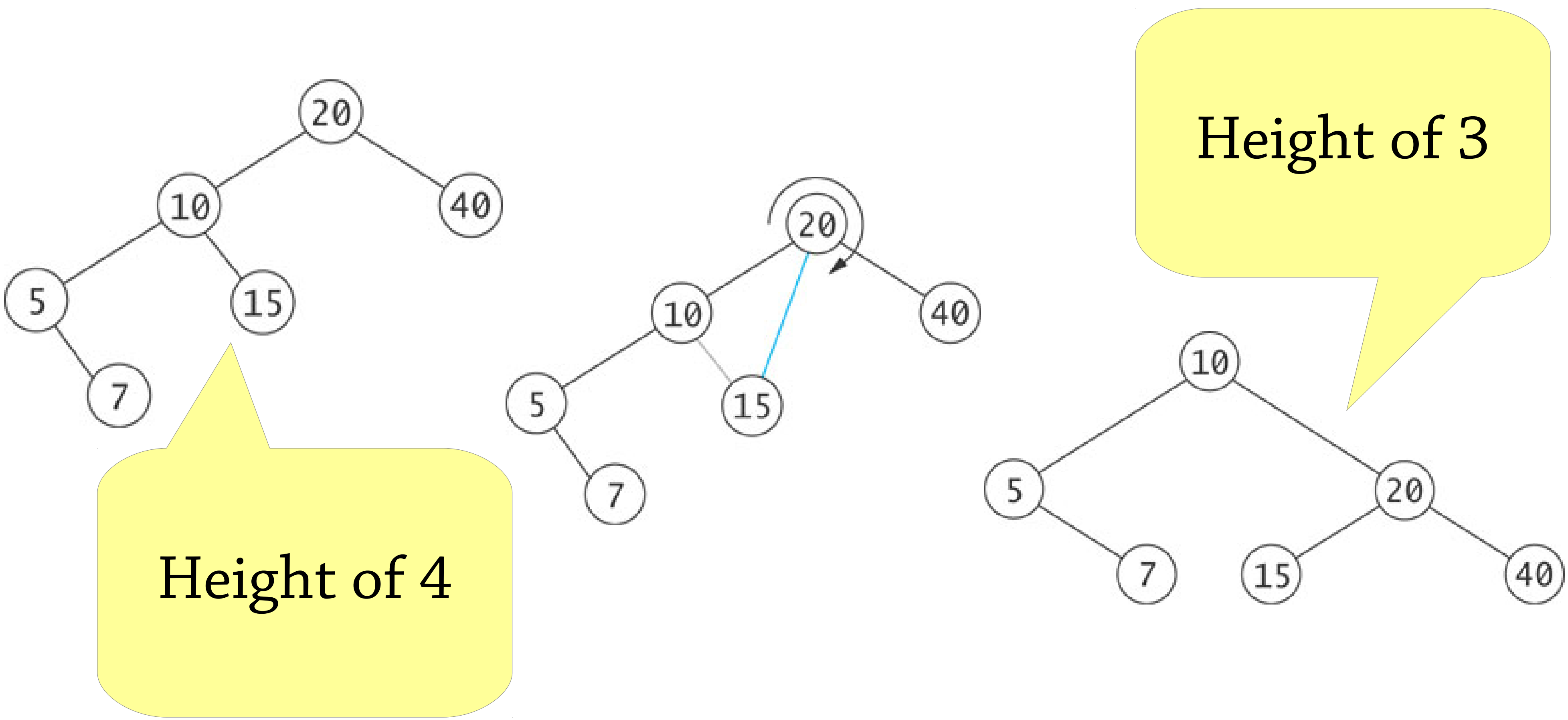
Rotation rearranges a BST by moving a different node to the root, without changing the BST's contents



(pic from Wikipedia)

Rotation

We can use rotations to adjust the relative height of the left and right branches of a tree



AVL insertion

Start by doing a BST insertion

- This might break the AVL (balance) invariant

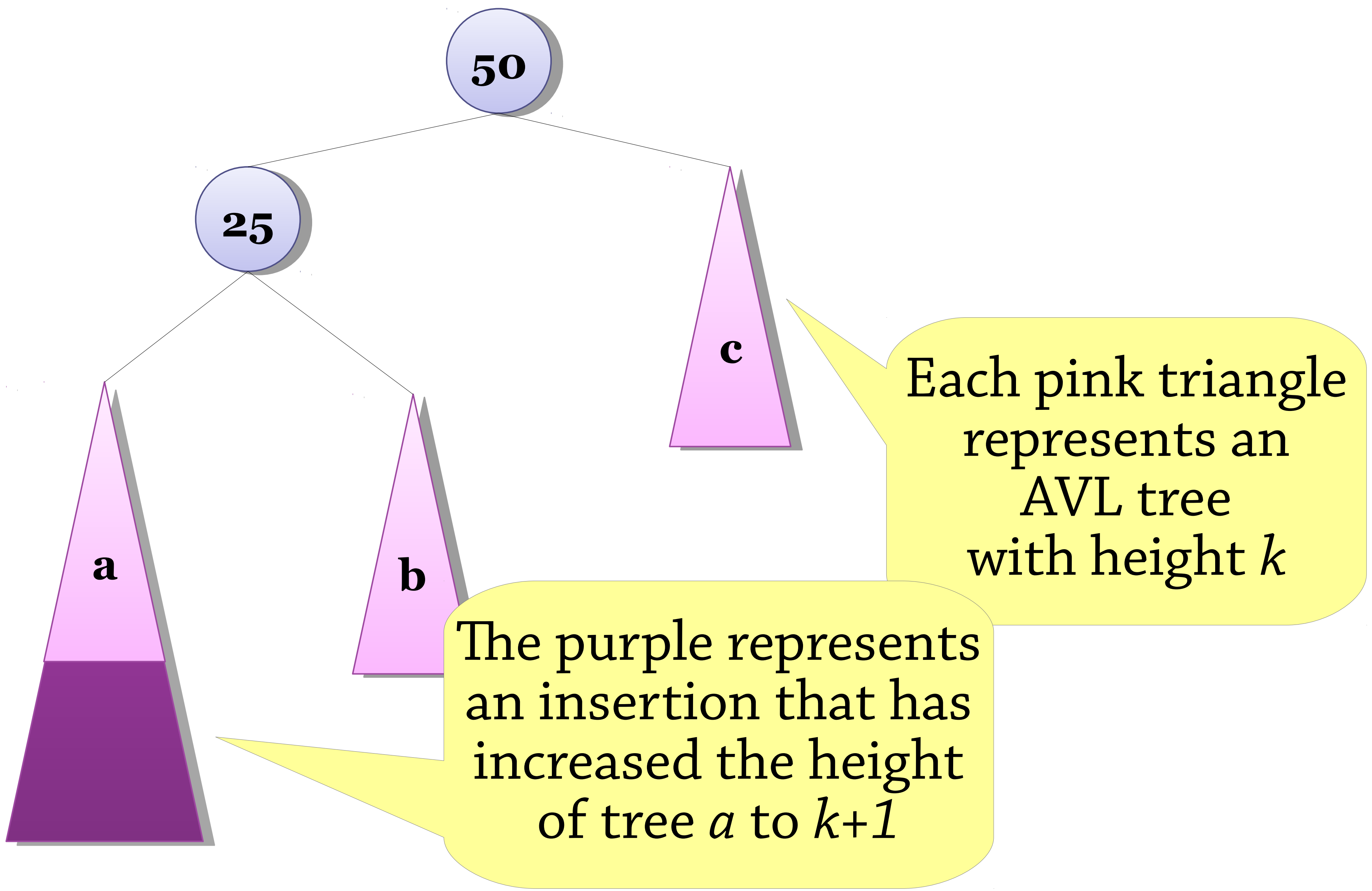
Then go upwards from the newly-inserted node, looking for nodes that break the invariant (unbalanced nodes)

Whenever you find one, rotate it

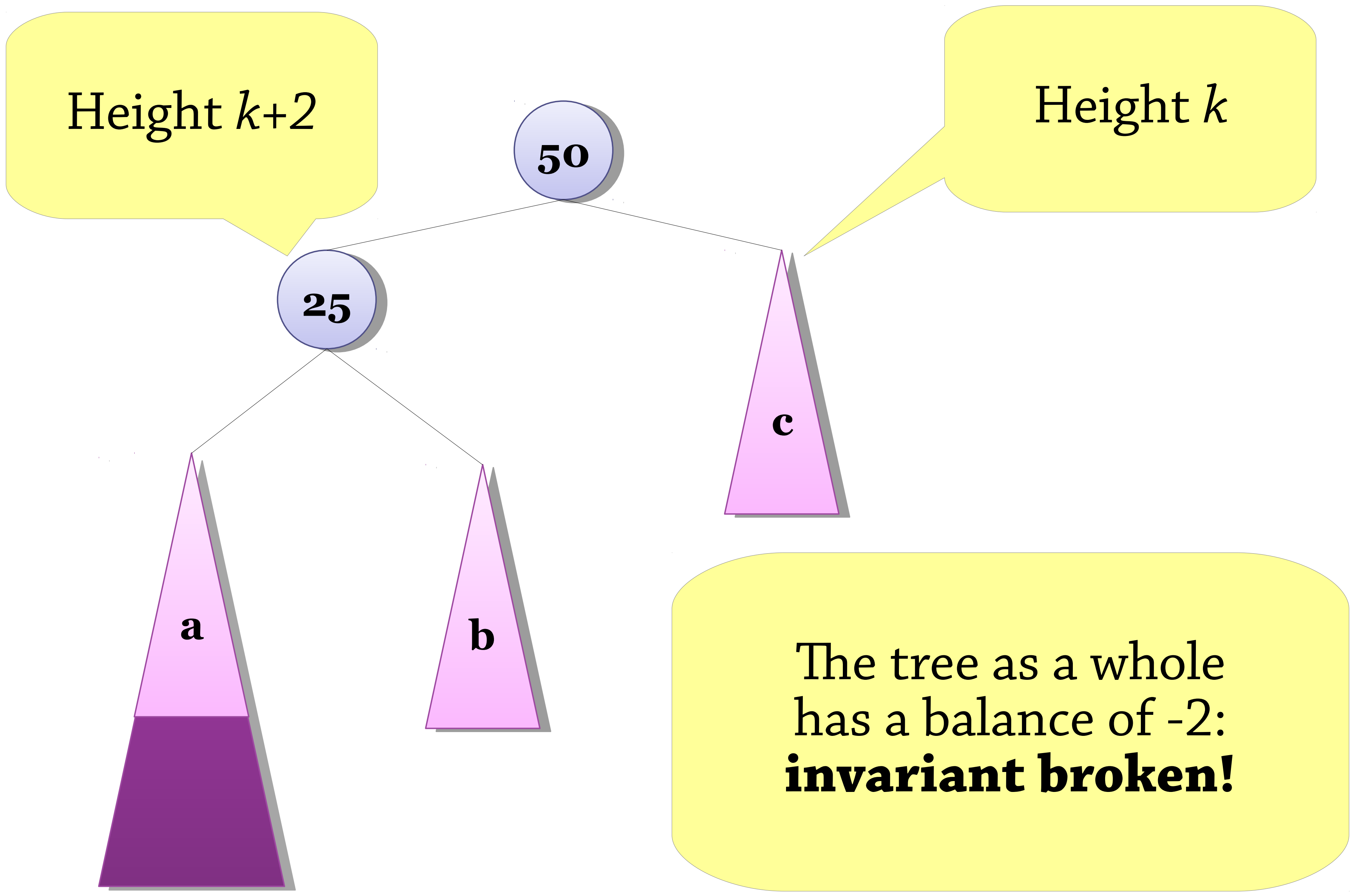
- Then continue upwards in the tree

There are several cases depending on *how* the node is unbalanced

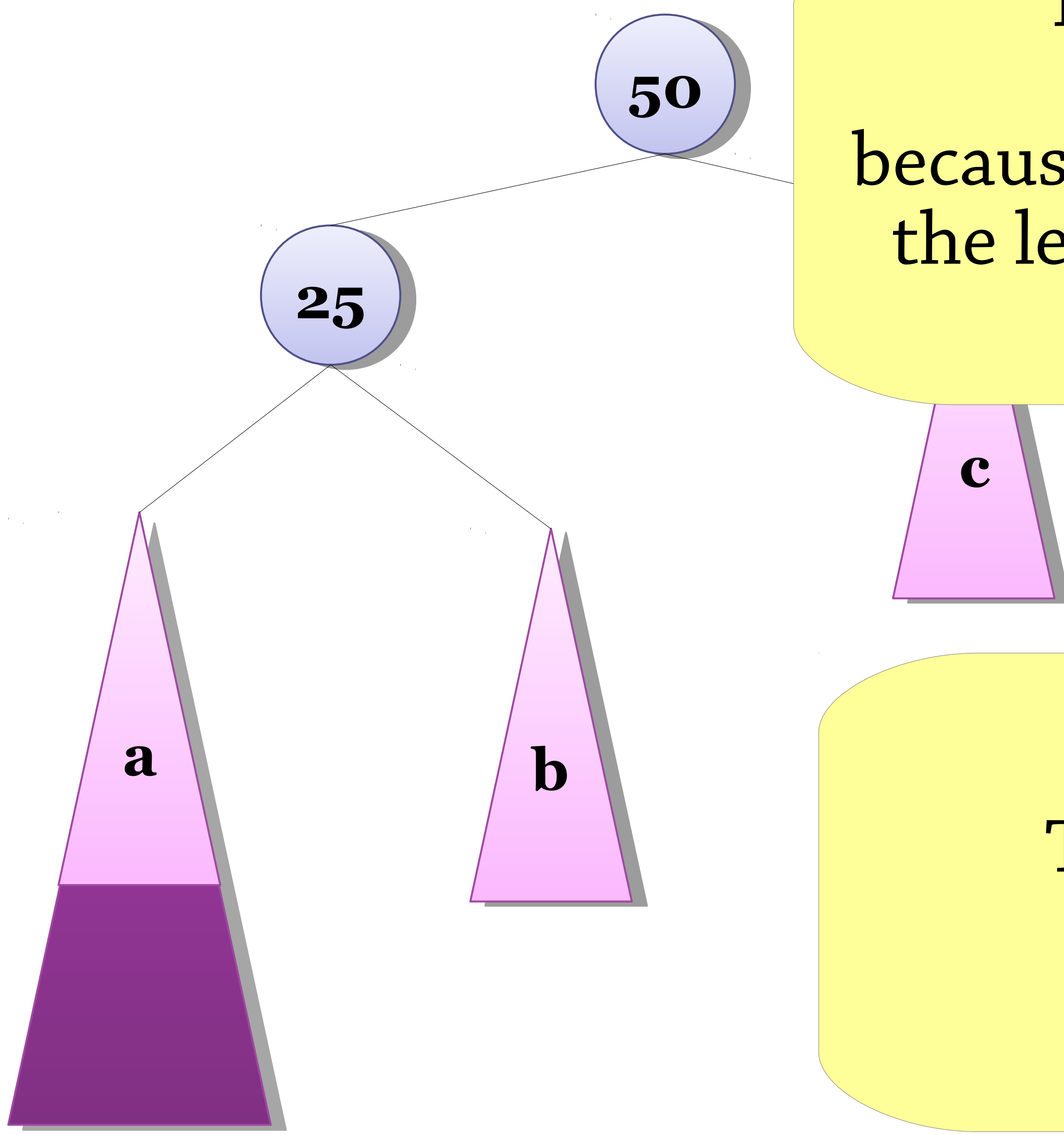
Case 1: a *left-left* tree



Case 1: a *left-left* tree



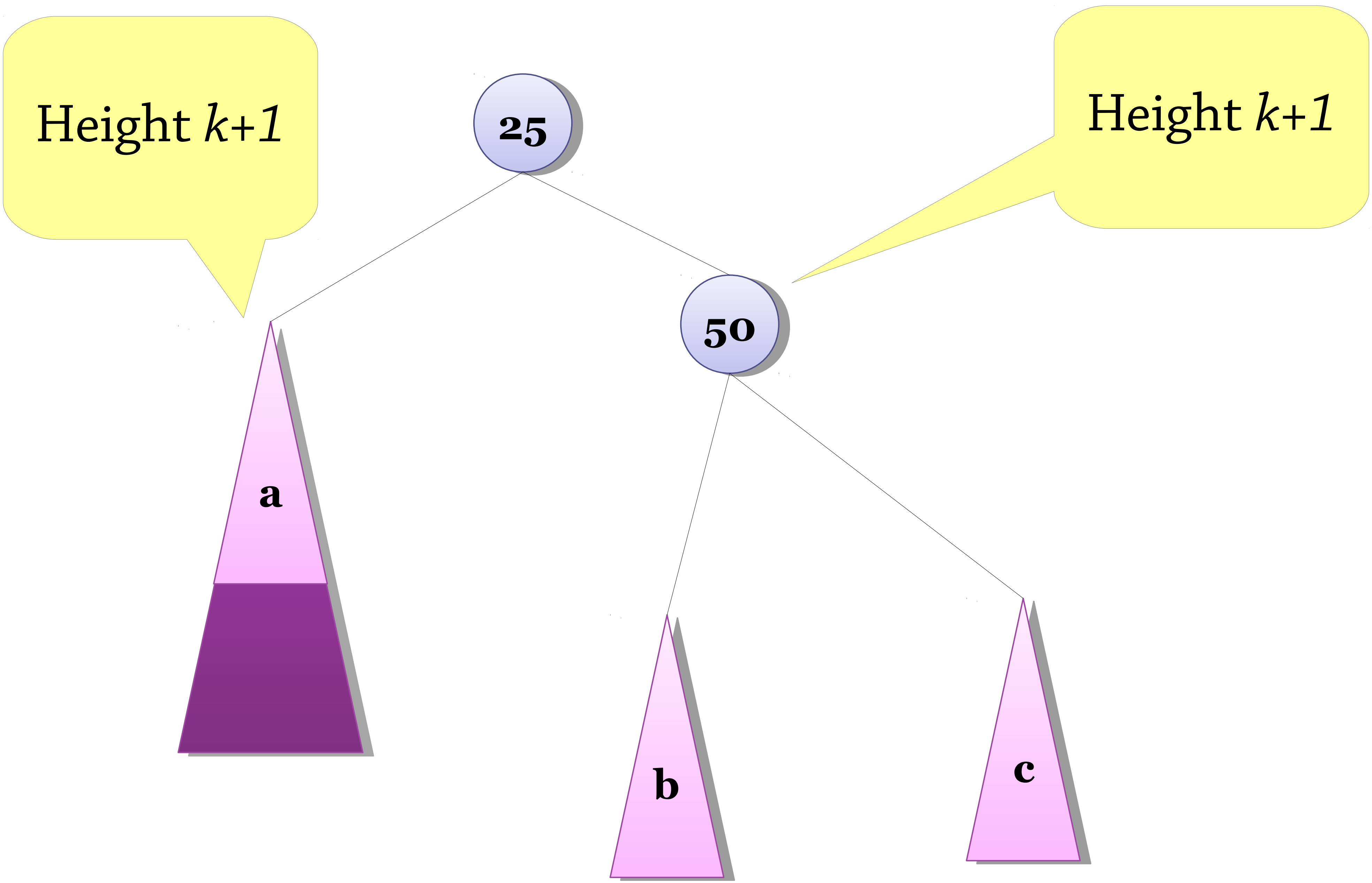
Case 1: a *left-left* tree



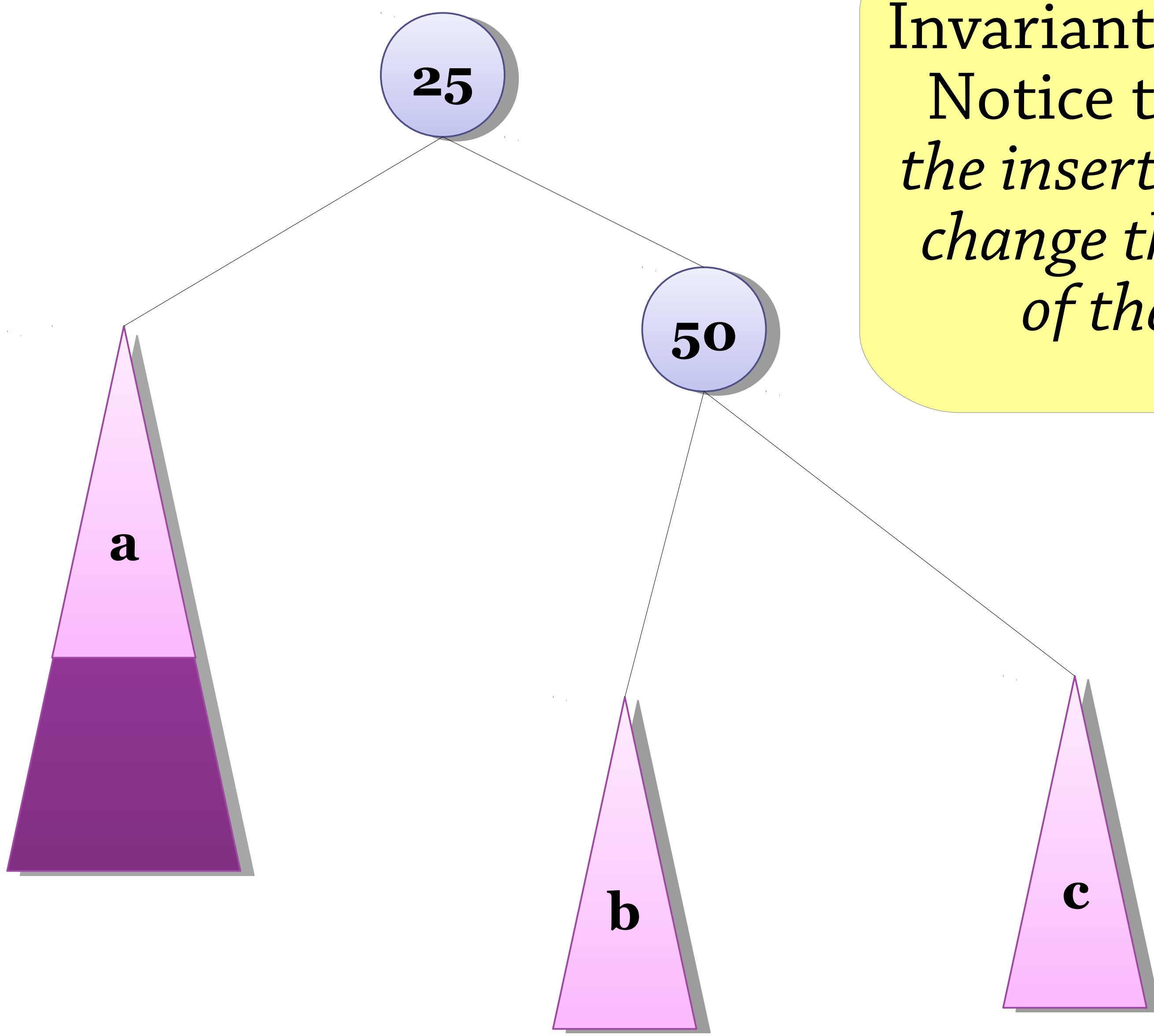
This is called a
left-left tree
because both the root and the left child are deeper on the left

To fix it we do a
right rotation

Balancing a left-left tree, afterwards

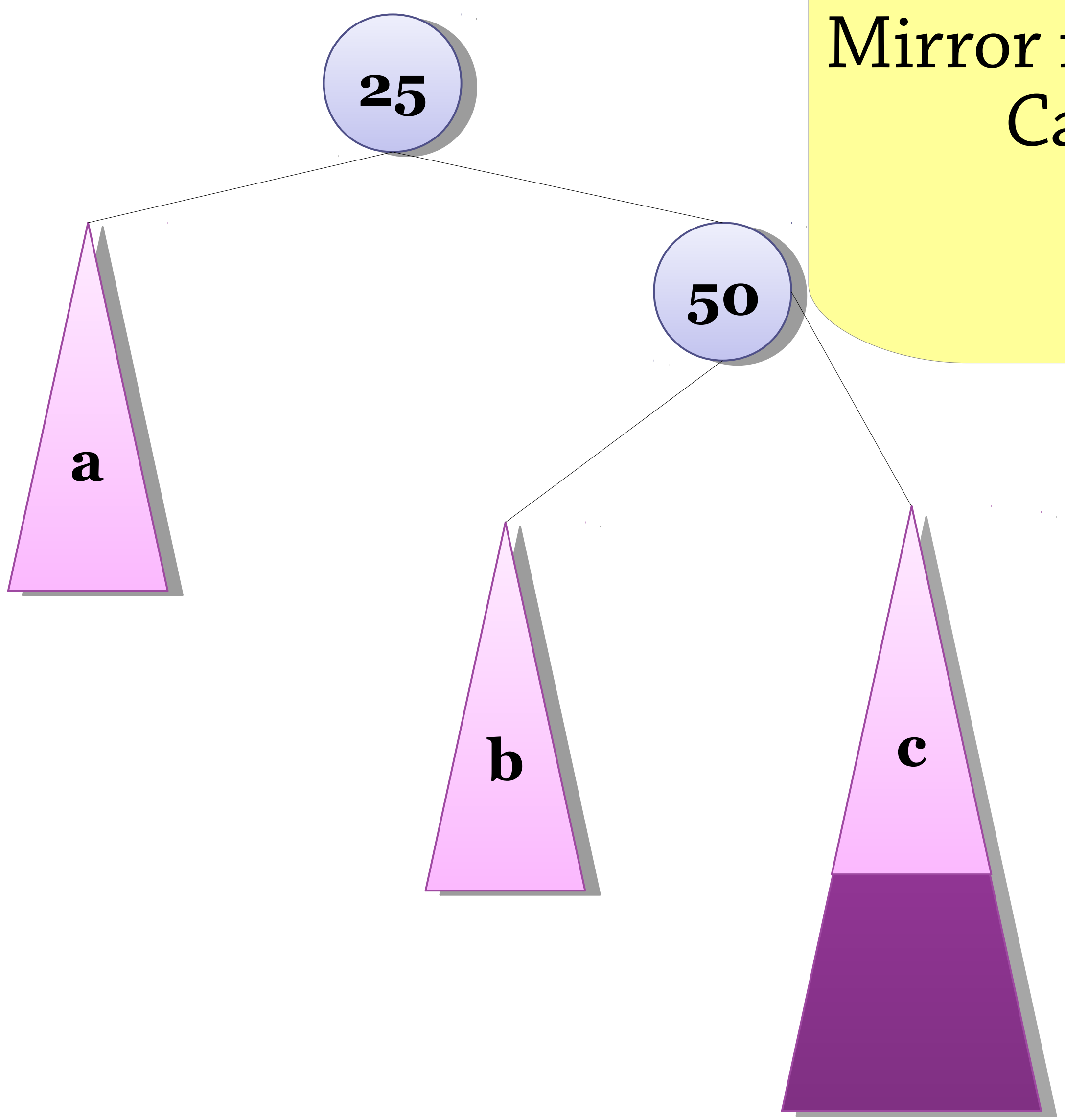


Balancing a left-left tree, afterwards



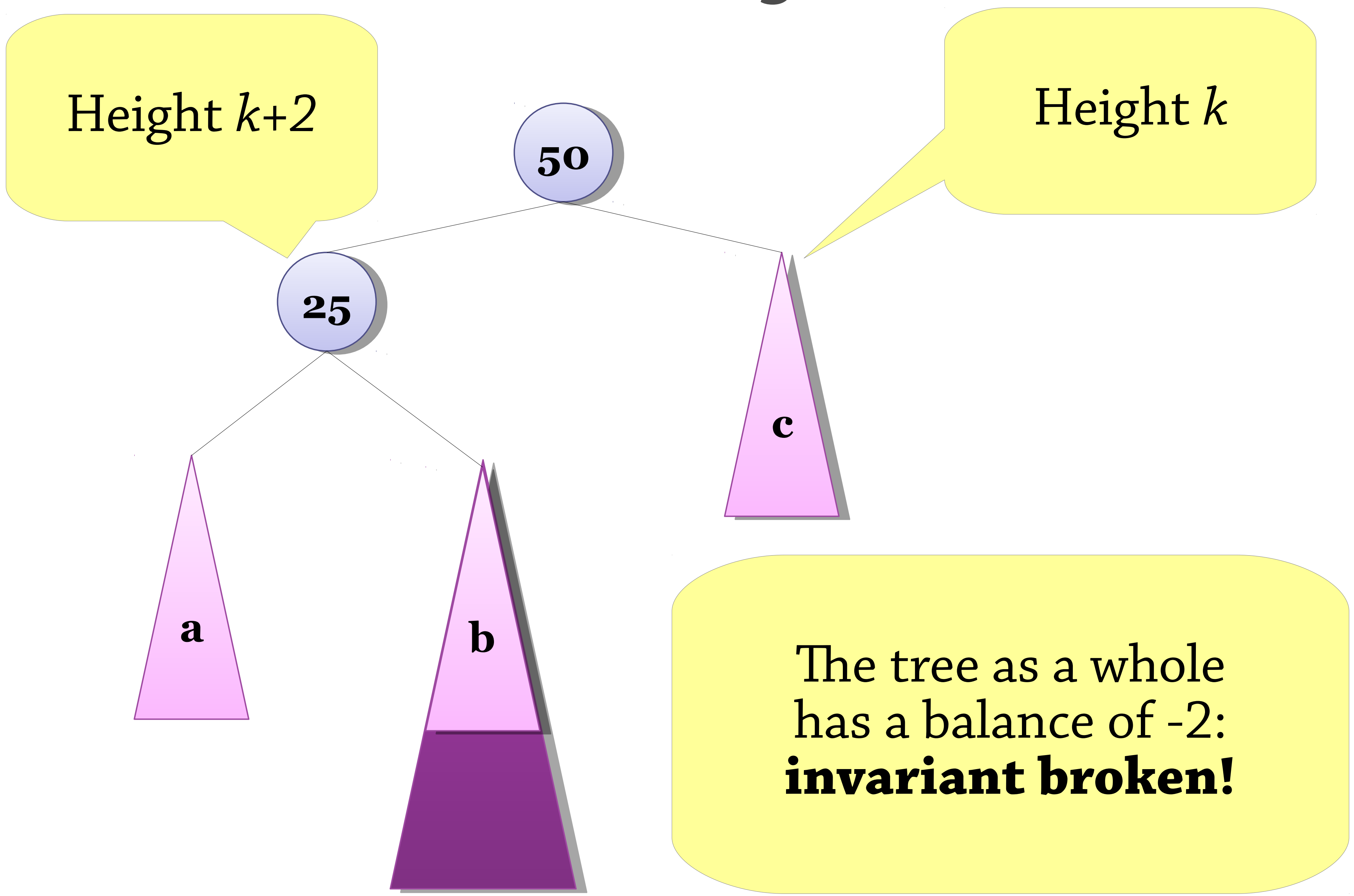
Invariant restored!
Notice that now
the insertion didn't change the height of the tree

Case 2: a *right-right* tree

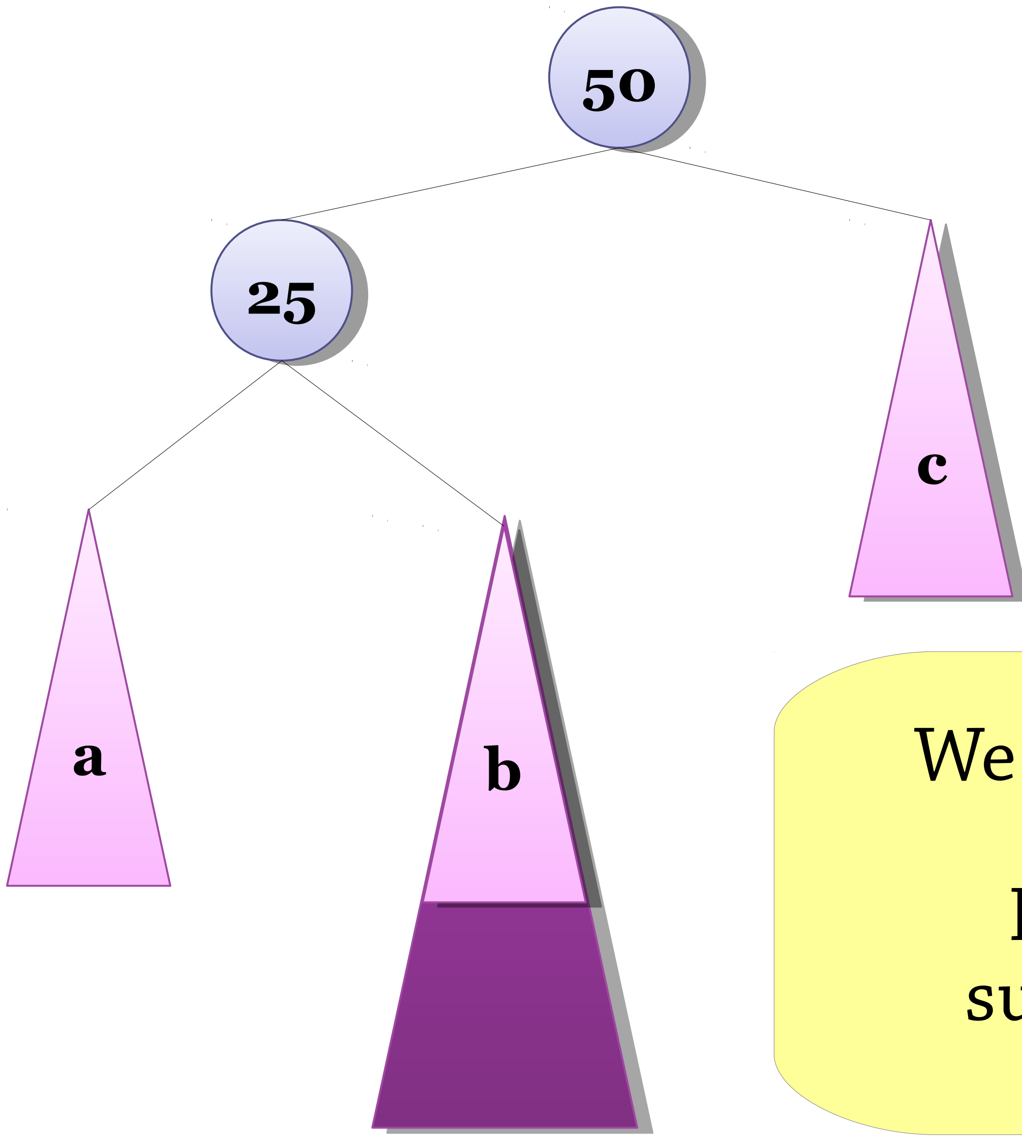


Mirror image of left-left tree
Can be fixed with
left rotation

Case 3: a *left-right* tree

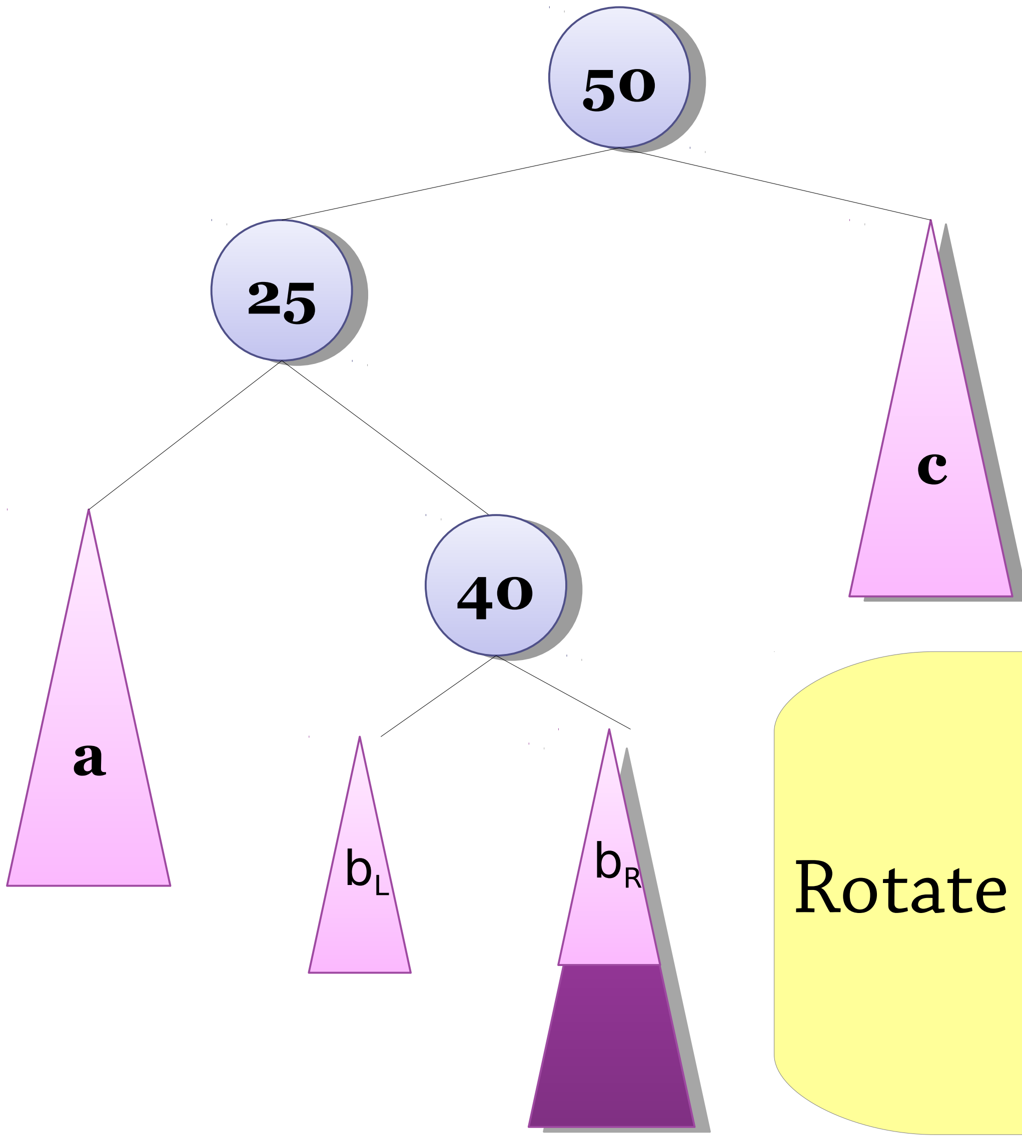


Case 3: a *left-right* tree



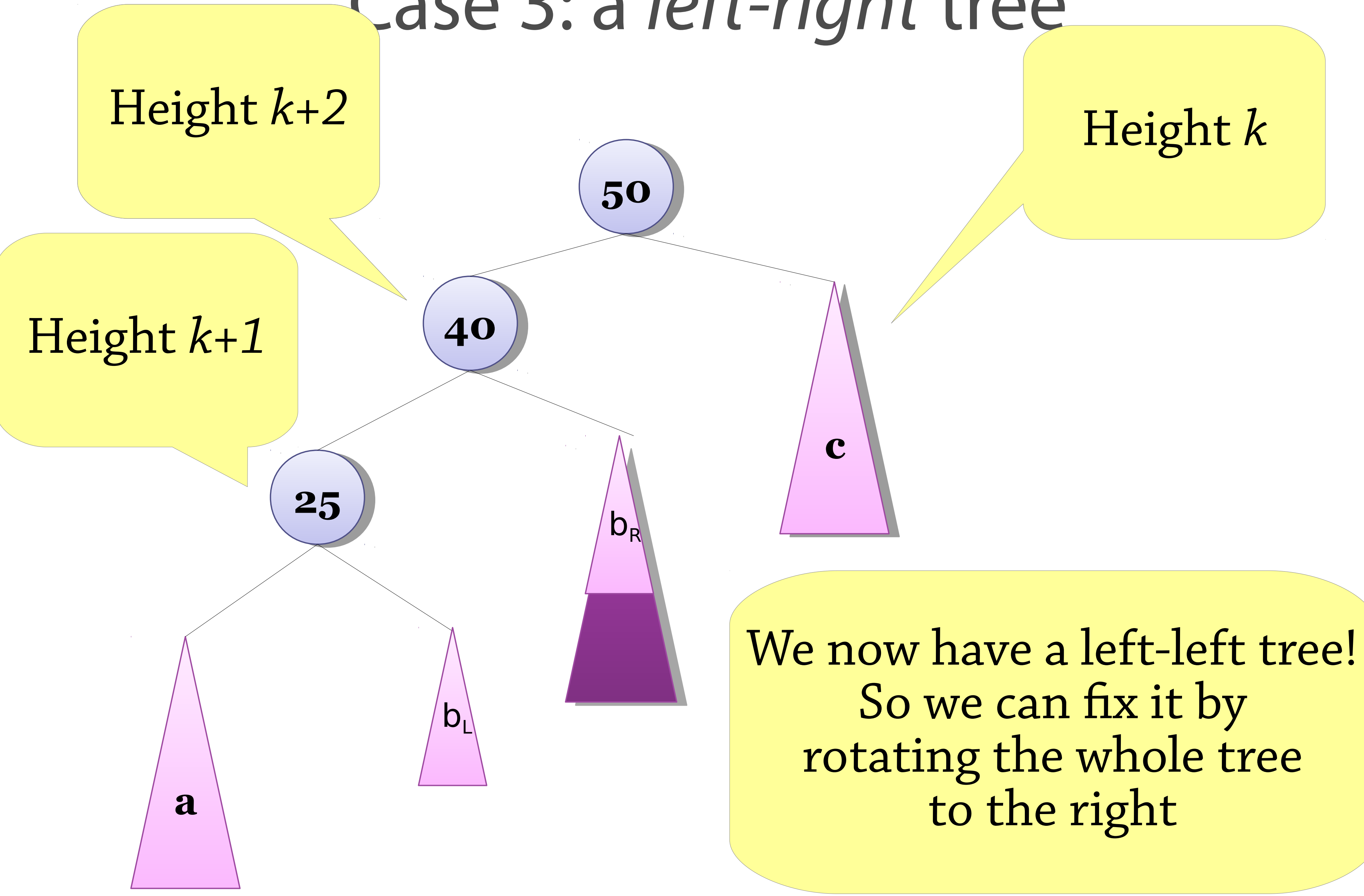
We can't fix this with one rotation
Let's look at b's subtrees b_L and b_R

Case 3: a *left-right* tree



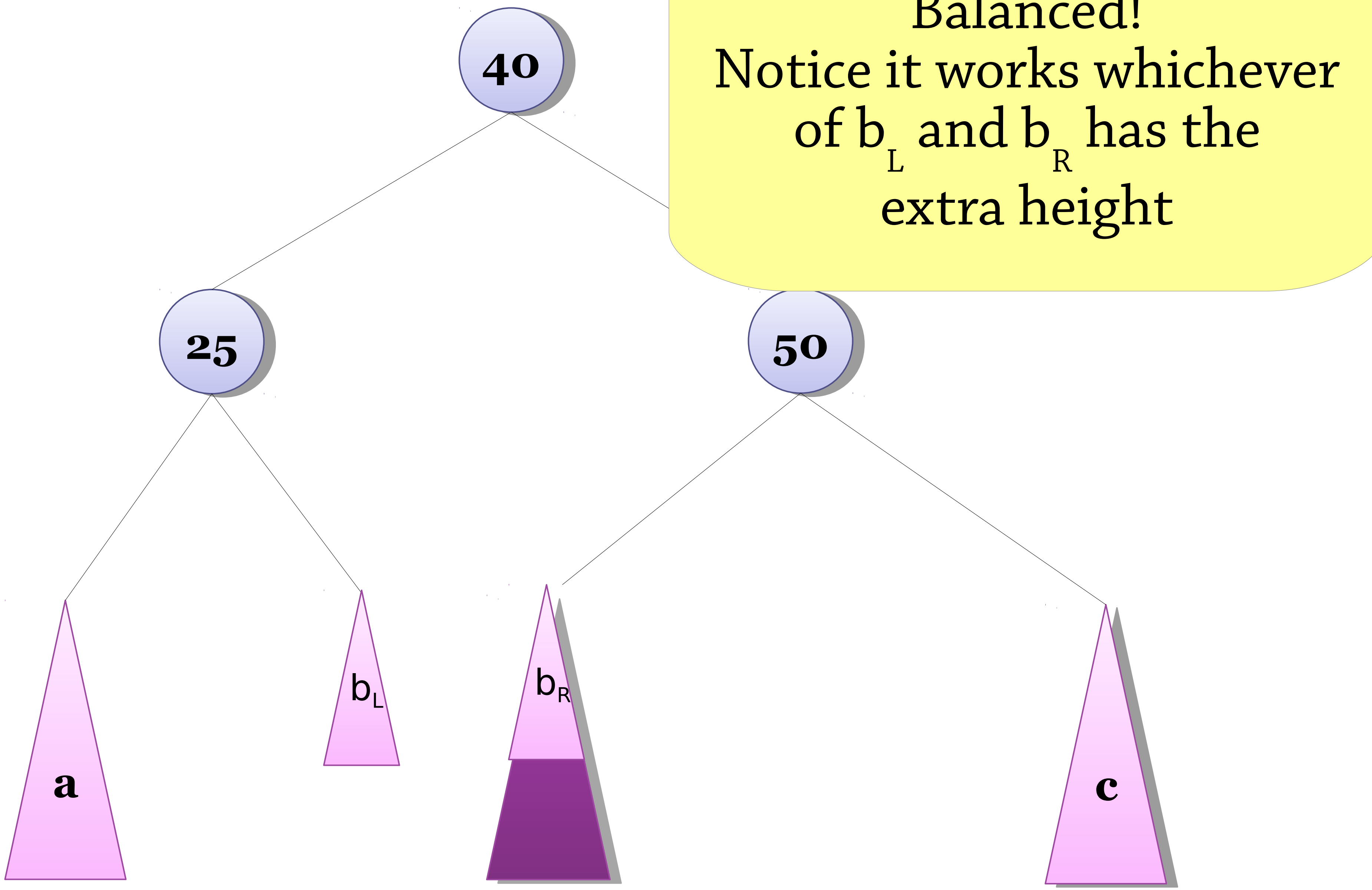
Rotate 25-subtree to the left

Case 3: a *left-right* tree

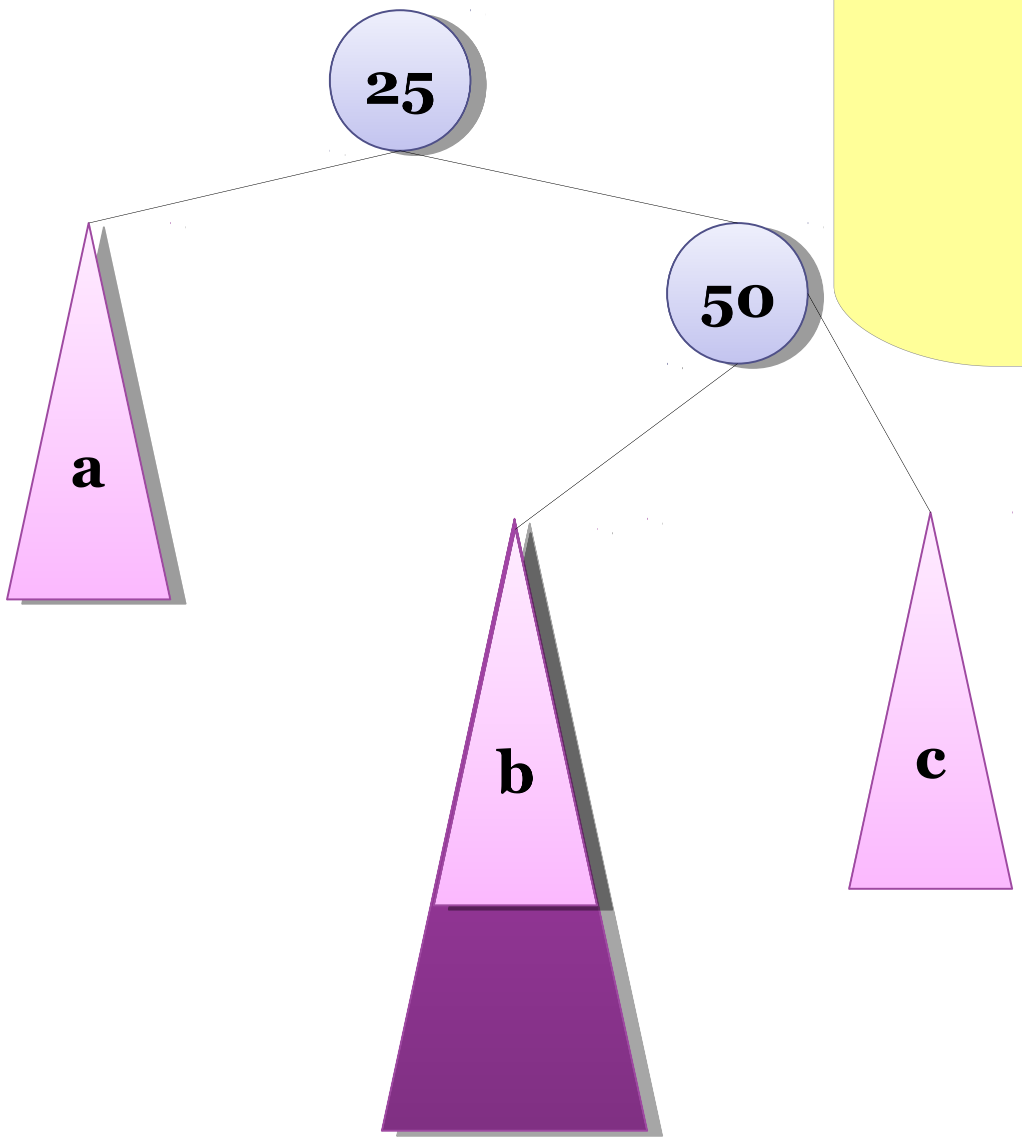


Case 3: a *left-right* tree

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

Four sorts of unbalanced trees

Left-left (root's balance is -2, left child's balance ≤ 0)

- Rotate the whole tree to the right

Left-right (root's balance is -2, left child's balance > 0)

- First rotate the left child to the left
- Then rotate the whole tree to the right

Right-left (root's balance is 2, right child's balance < 0)

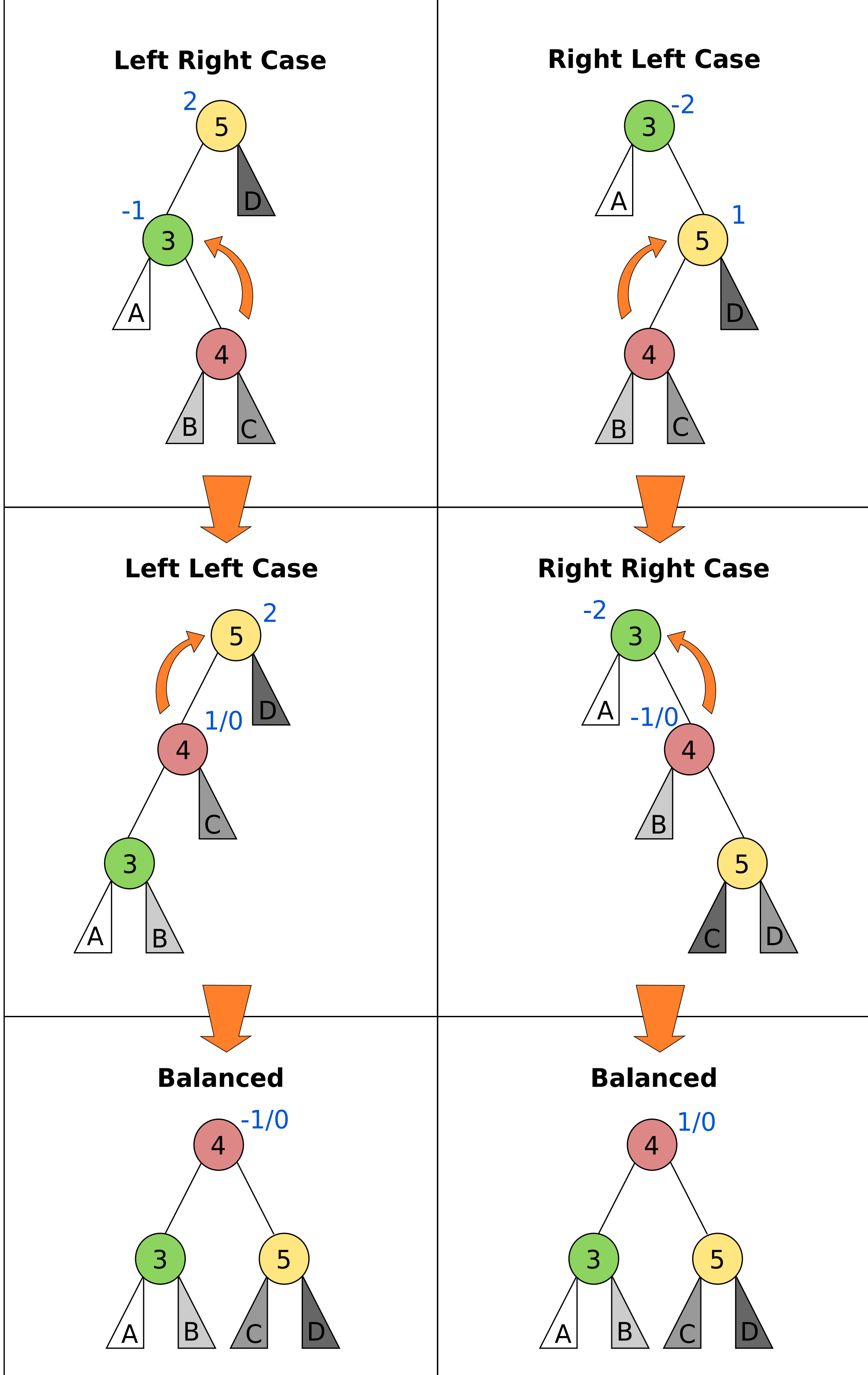
- First rotate the right child to the right
- Then rotate the whole tree to the left

Right-right (root's balance is 2, right child's balance ≥ 0)

- Rotate the whole tree to the left

The four cases

(picture from Wikipedia)

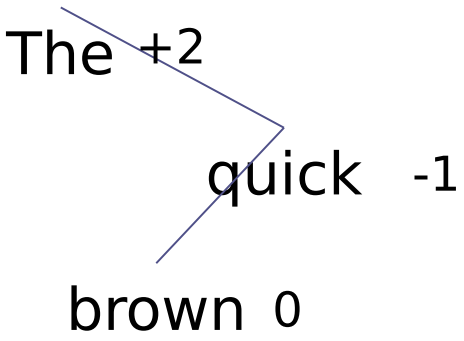


A bigger example
(slides from Peter Ljunglöf)

Let's build an AVL tree for the words in

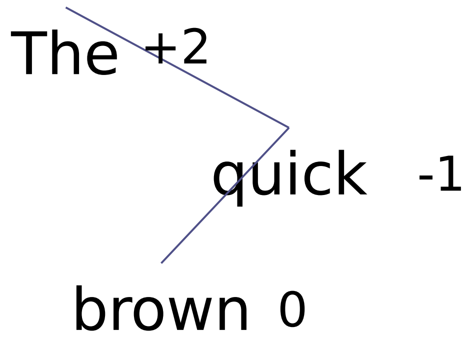
”The quick brown fox jumps over the lazy dog”

The quick brown…



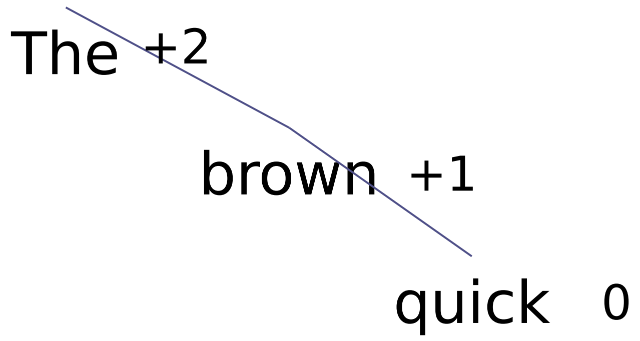
The overall tree is right-heavy (Right-Left)
parent balance = +2
right child balance = -1

The quick brown…



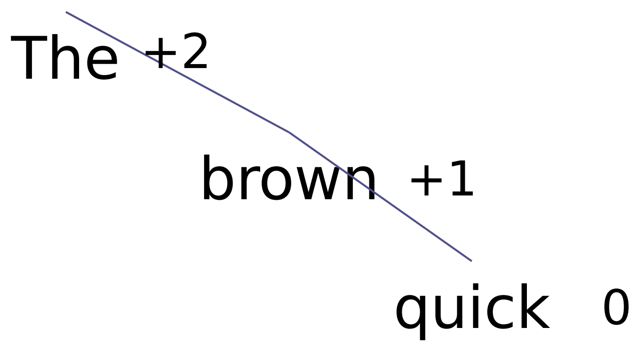
1. Rotate right around the child

The quick brown…



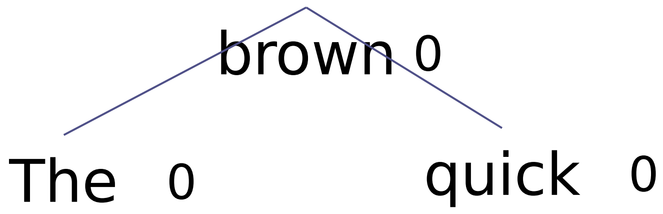
1. Rotate right around the child

The quick brown…



1. **Rotate right around the child**
2. **Rotate left around the parent**

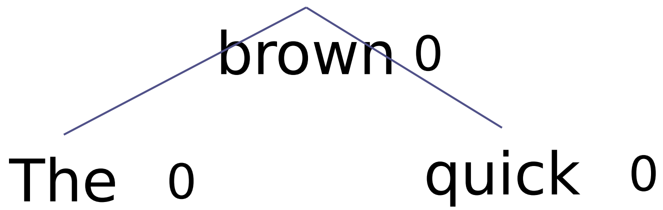
The quick brown…



1. **Rotate right around the child**

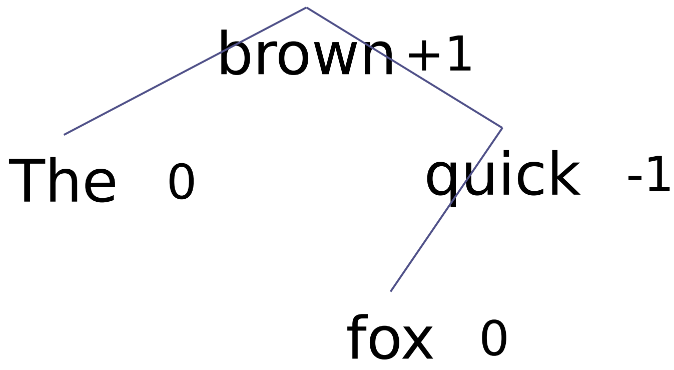
2. **Rotate left around the parent**

The quick brown fox…



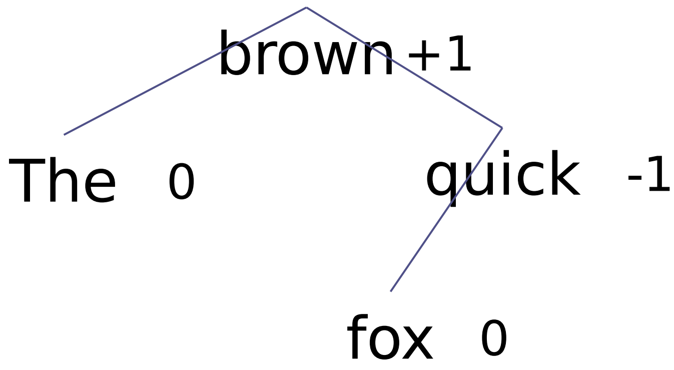
Insert *fox*

The quick brown fox…



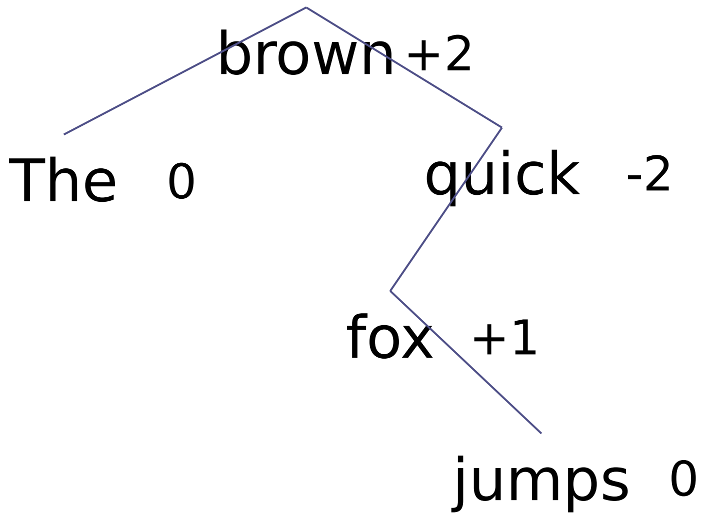
Insert *fox*

The quick brown fox jumps…



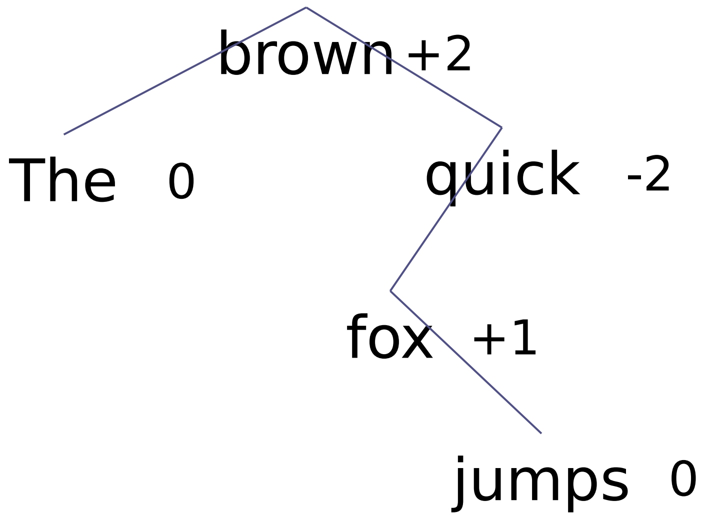
Insert *jumps*

The quick brown fox jumps…



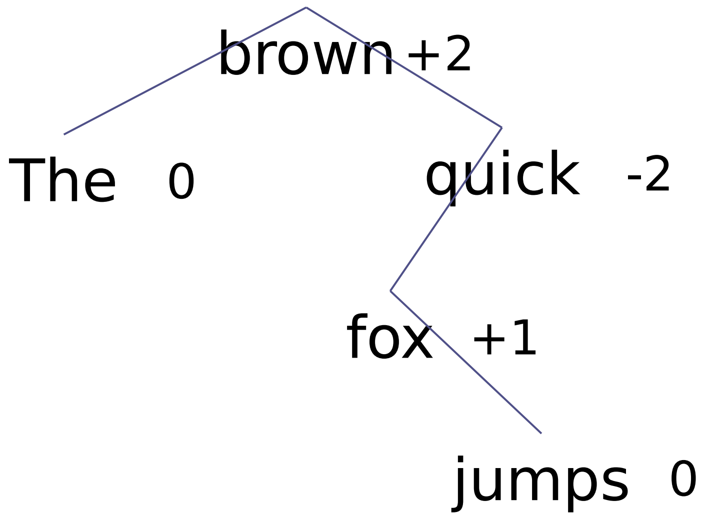
Insert *jumps*

The quick brown fox jumps…



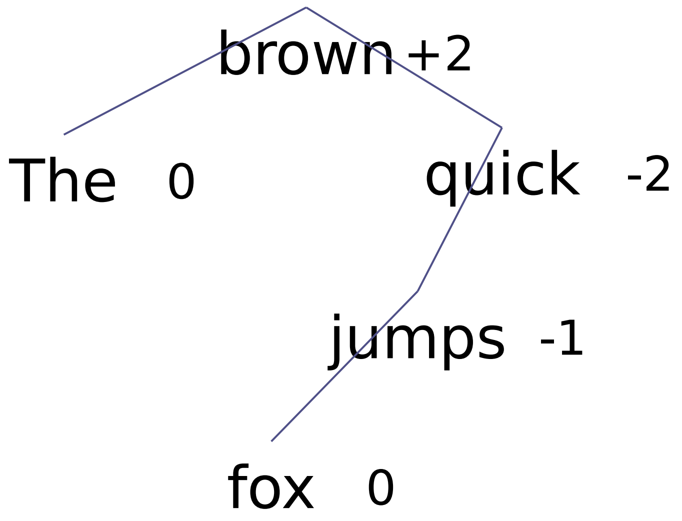
The tree is now left-heavy about *quick* (Left-Right case)

The quick brown fox jumps…



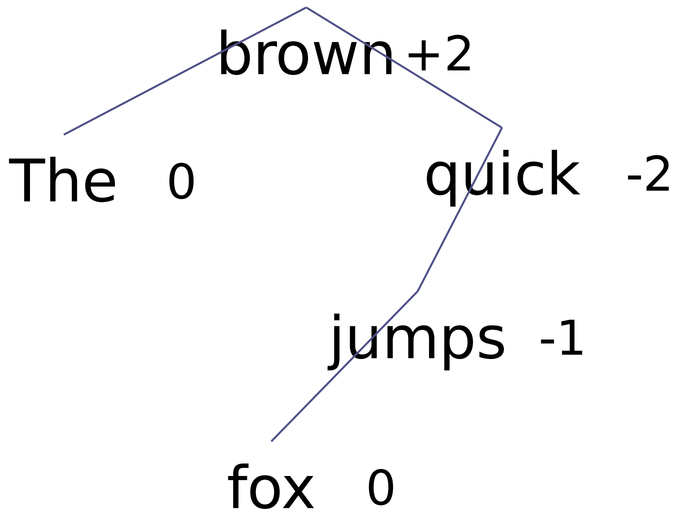
1. **Rotate left around the child**

The quick brown fox jumps…



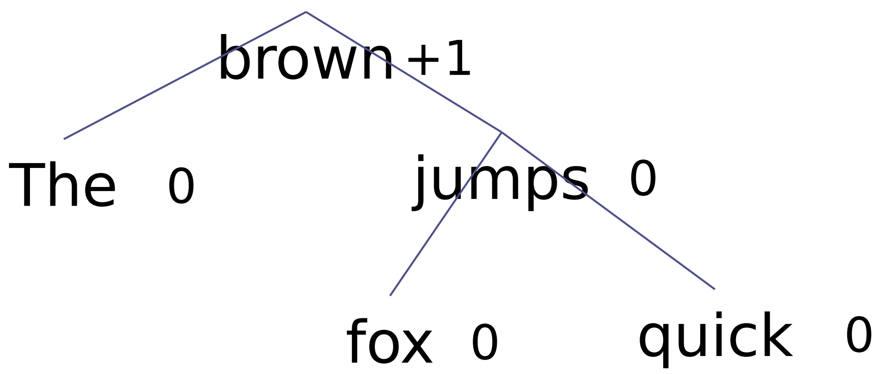
1. Rotate left around the child

The quick brown fox jumps…



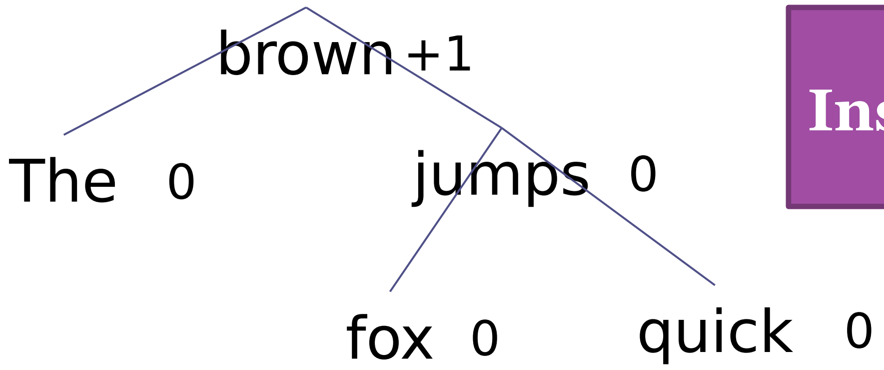
1. **Rotate left around the child**
2. **Rotate right around the parent**

The quick brown fox jumps…



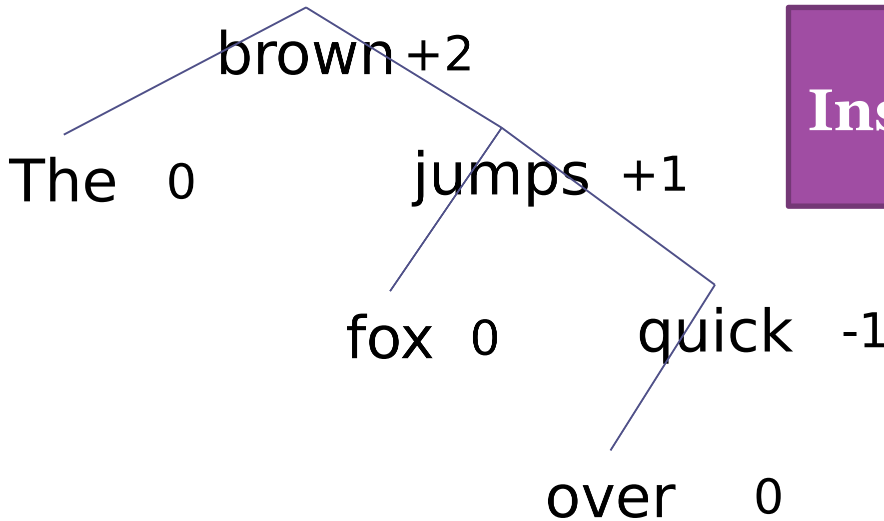
1. **Rotate left around the child**
2. **Rotate right around the parent**

The quick brown fox jumps over…



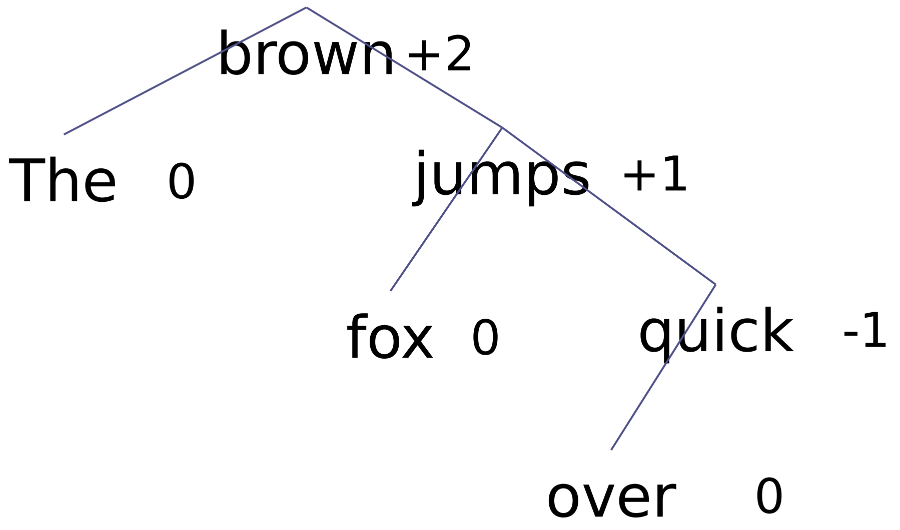
Insert *over*

The quick brown fox jumps over…



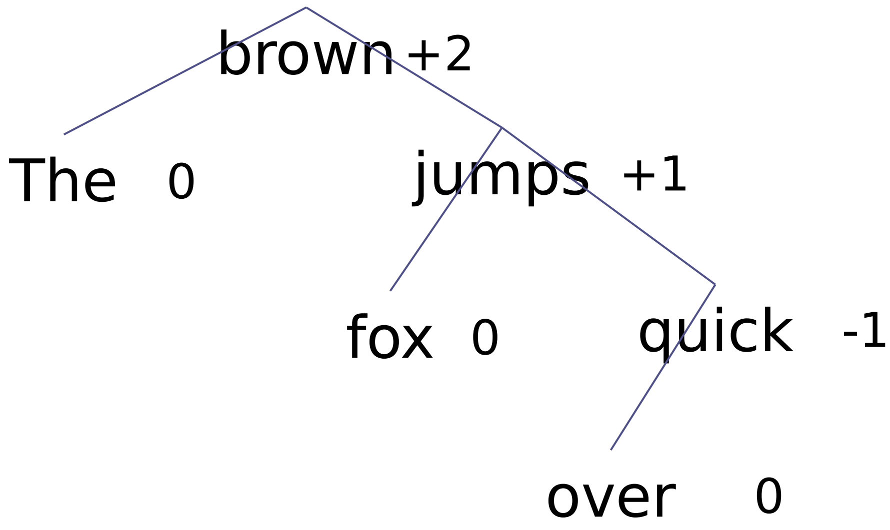
Insert *over*

The quick brown fox jumps over…



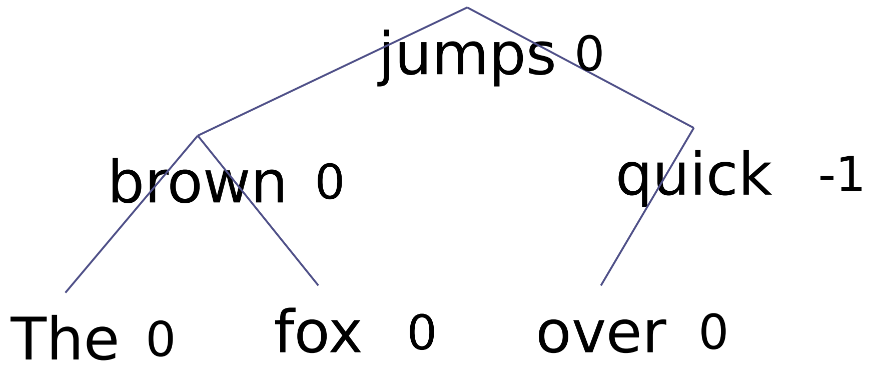
We now have a Right-Right imbalance

The quick brown fox jumps over…



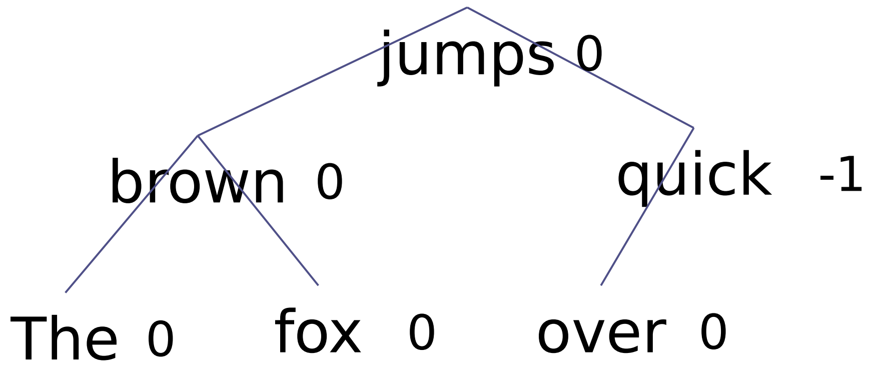
1. Rotate left around the parent

The quick brown fox jumps over…



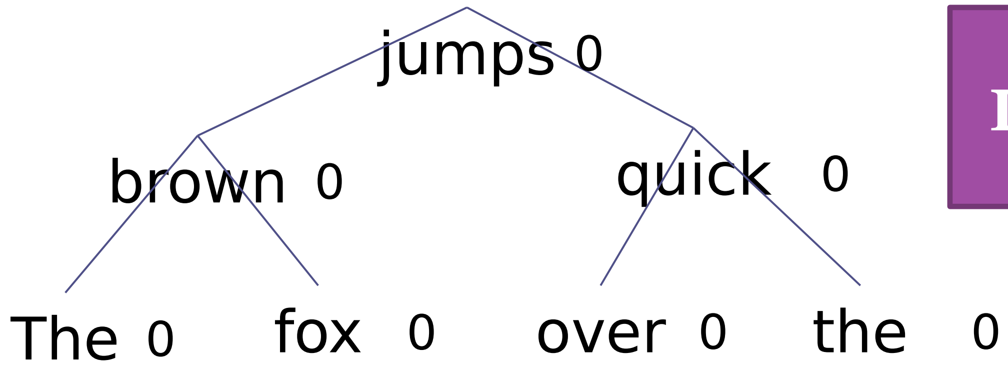
1. Rotate left around the parent

The quick brown fox jumps over the…



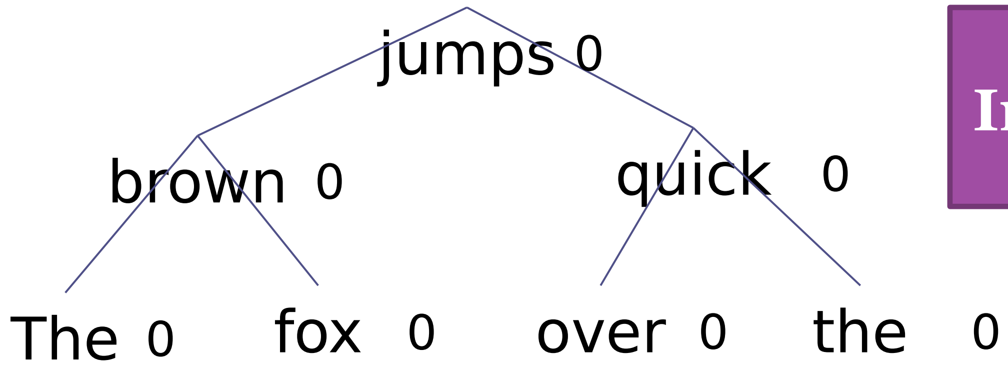
Insert *the*

The quick brown fox jumps over the…



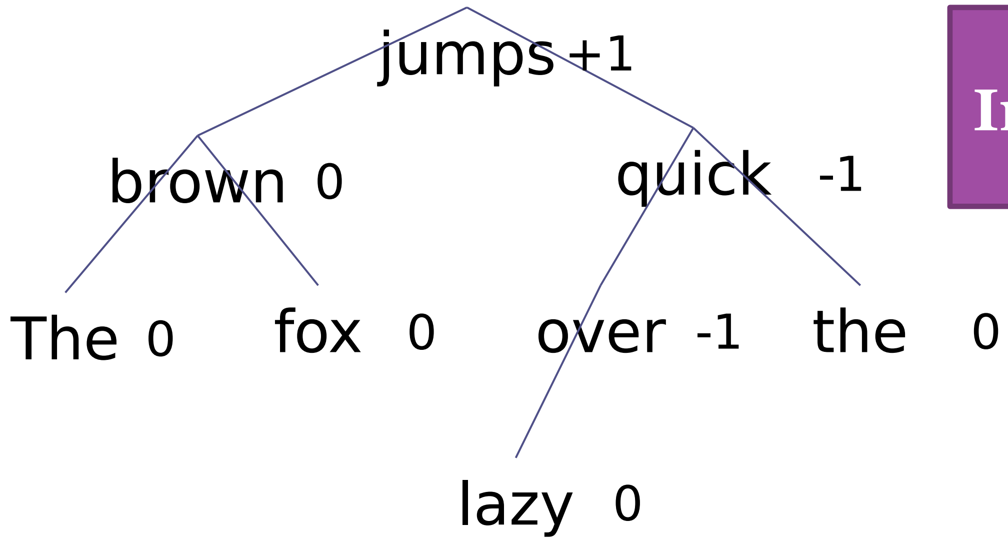
Insert *the*

The quick brown fox jumps over the lazy…



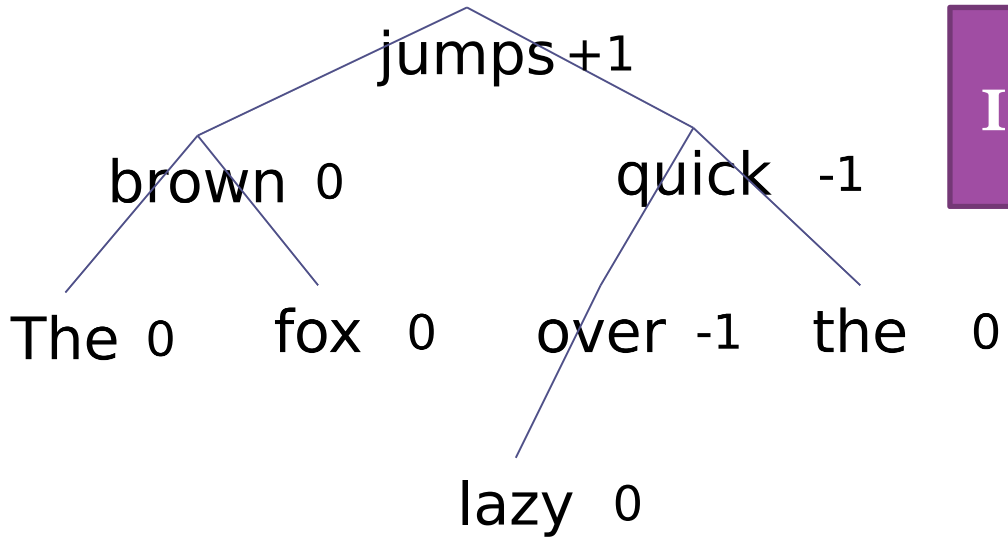
Insert *lazy*

The quick brown fox jumps over the lazy…



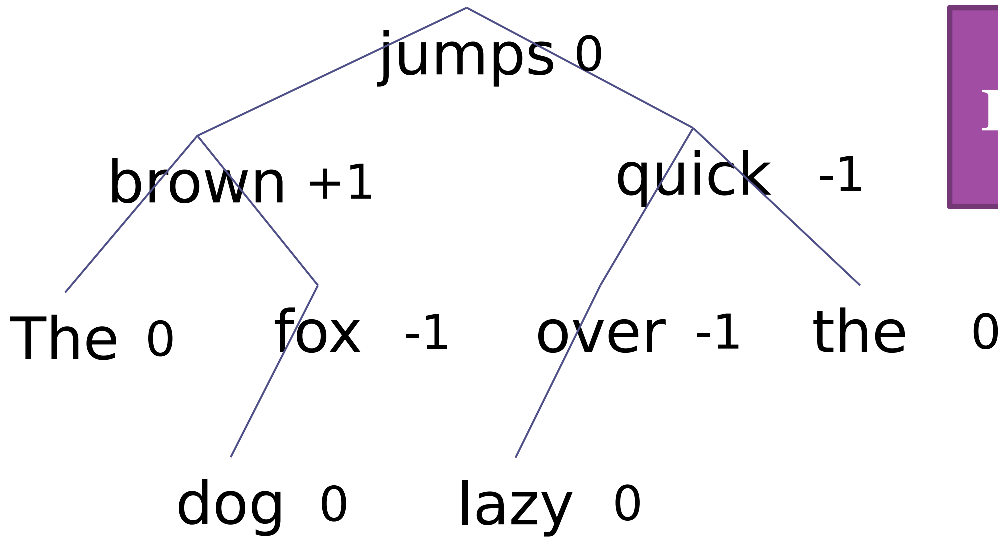
Insert *lazy*

e quick brown fox jumps over the lazy dog



Insert *dog*

quick brown fox jumps over the lazy dog!



Insert *dog*

AVL trees

A balanced BST that maintains balance by *rotating* the tree

- Insertion: insert as in a BST and move upwards from the inserted node, rotating unbalanced nodes
- Deletion (in book if you're interested): delete as in a BST and move upwards from the node that disappeared, rotating unbalanced nodes

Worst-case (it turns out) 1.44log n, typical log n comparisons for any operation – very balanced. This means lookups are quick.

- Insertion and deletion can be slower than in a naïve BST, because you have to do a bit of work to repair the invariant

Look in Haskell compendium (course website) for implementation