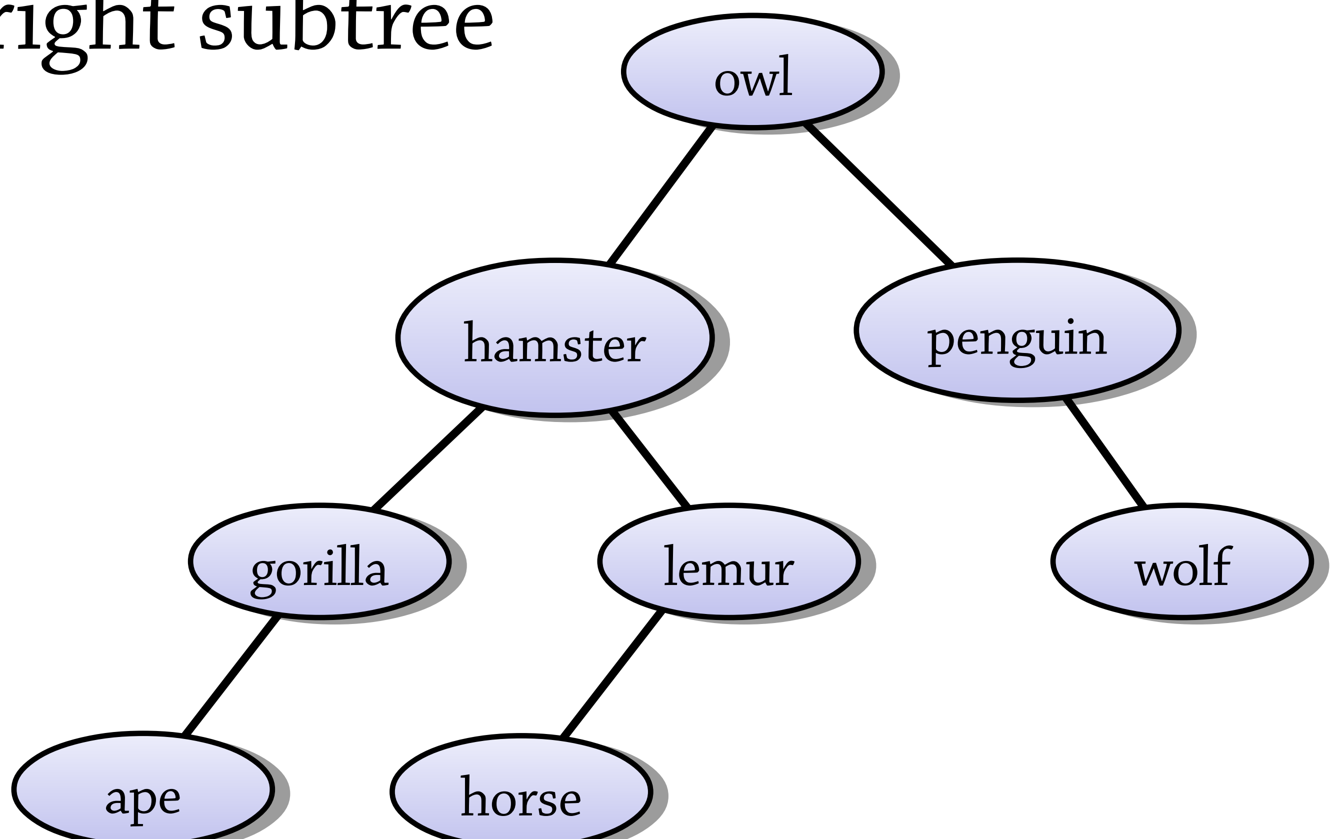
Binary search trees
(Weiss chapter 4.2-4.3)

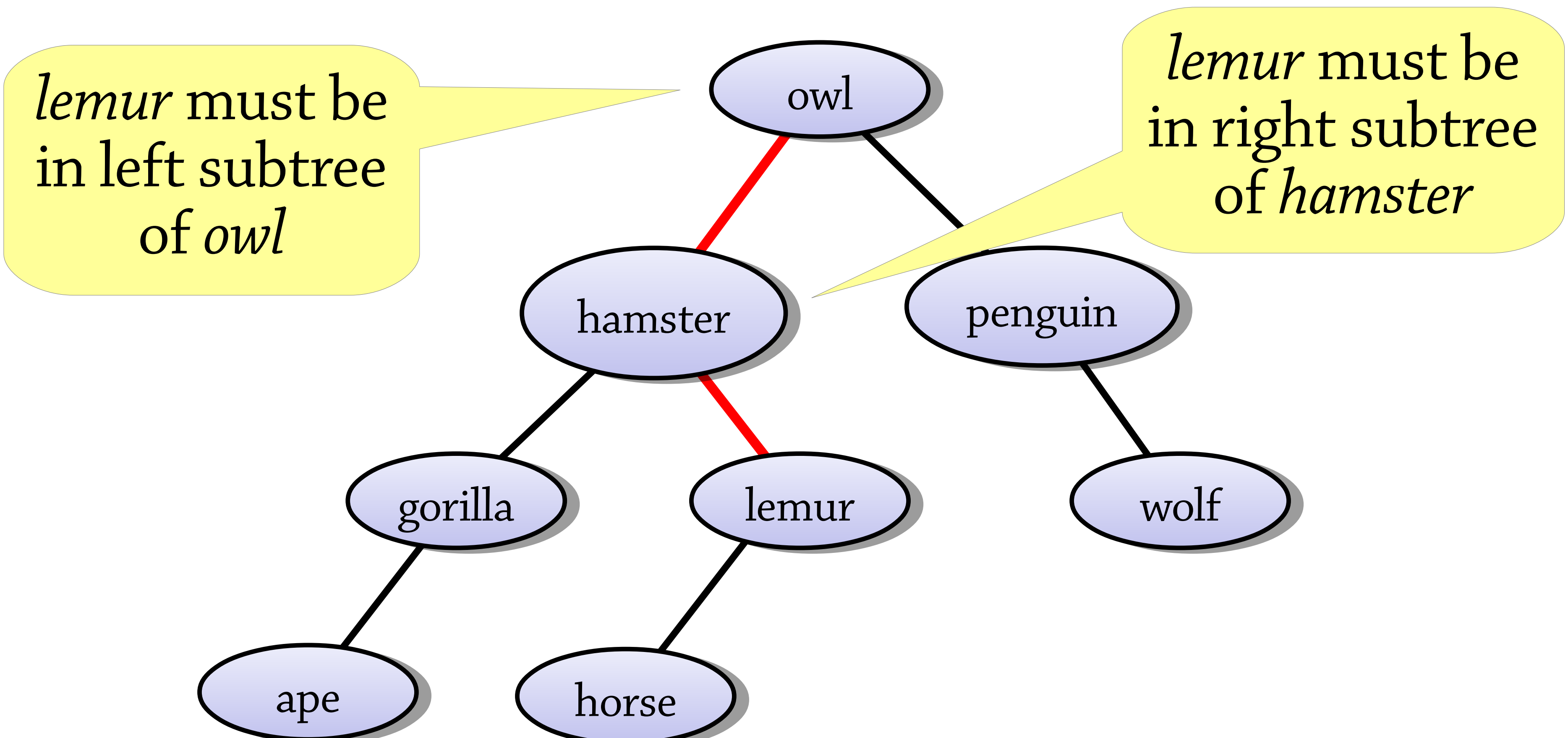
Binary search trees

A *binary search tree* (BST) is a binary tree where each node is greater than all the nodes in the left subtree, and less than all the nodes in the right subtree



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:

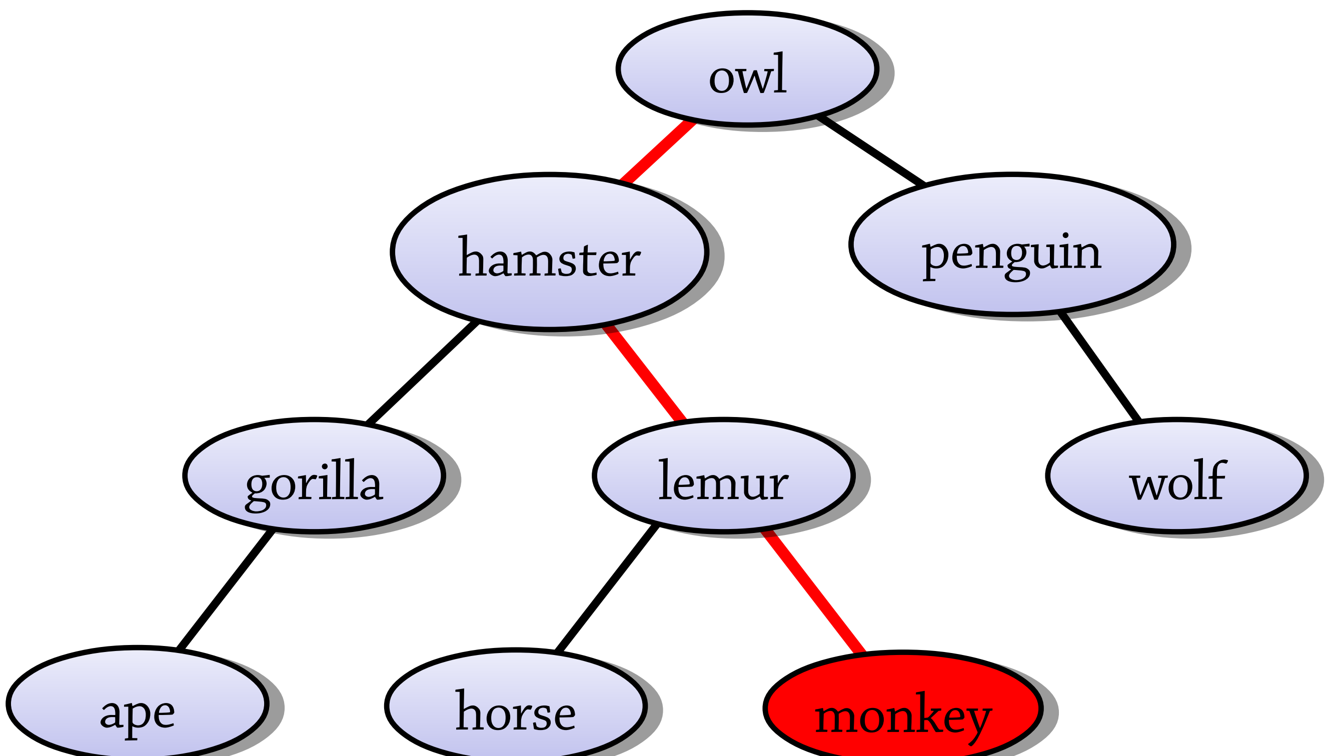
- If the target matches the root node's data, we've found it
- If the target is *less* than the root node's data, recursively search the left subtree
- If the target is *greater* than the root node's data, recursively search the right subtree
- If the tree is empty, fail

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:

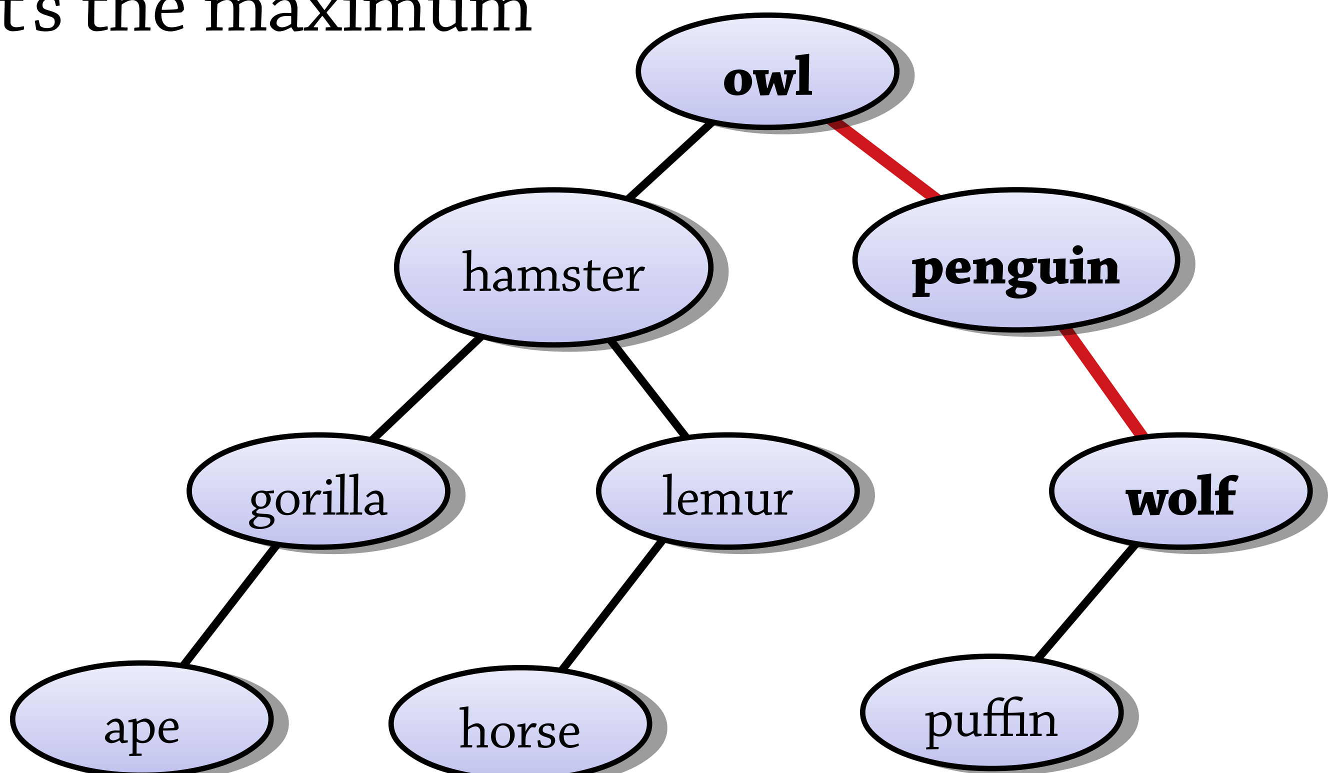
- Start by searching for the value
- But when you get to *null* (the empty tree), make a node for the value and place it there



Finding minimum/maximum in a BST

To find the maximum value in a BST:

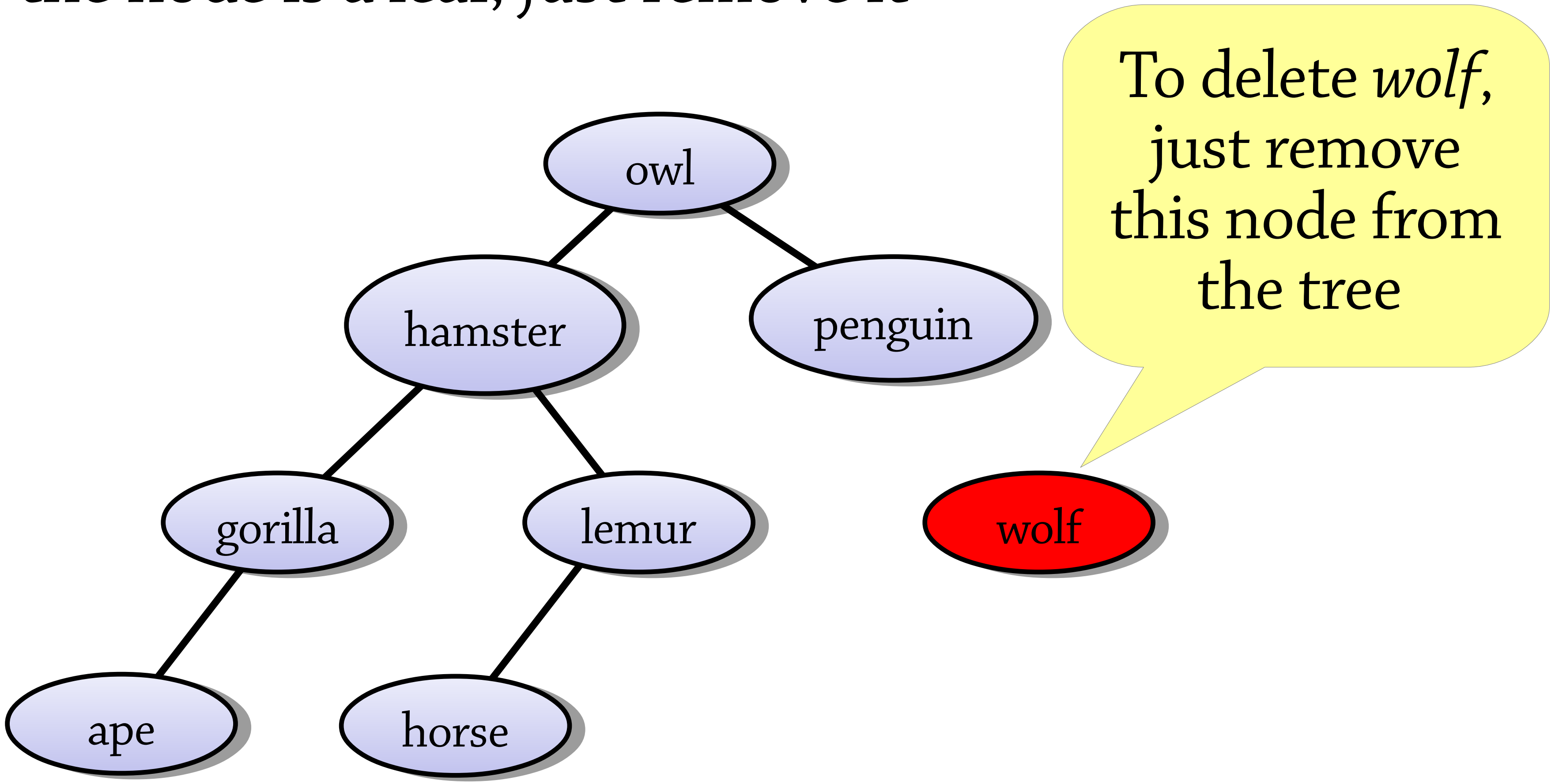
- Repeatedly go right from the root
- When you reach a node whose right child is empty, that's the maximum



Deleting from a BST

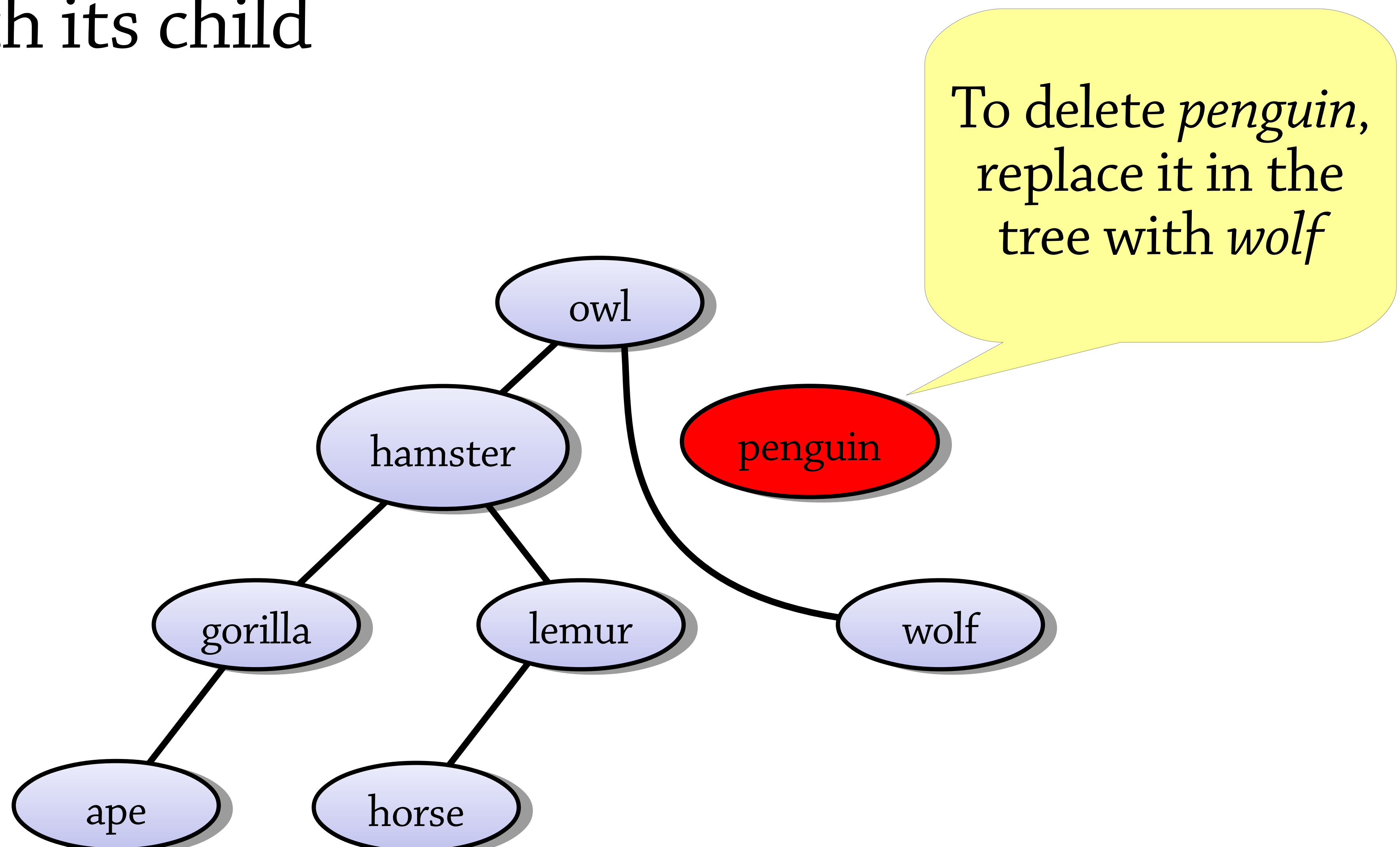
To delete a value from a BST:

- 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



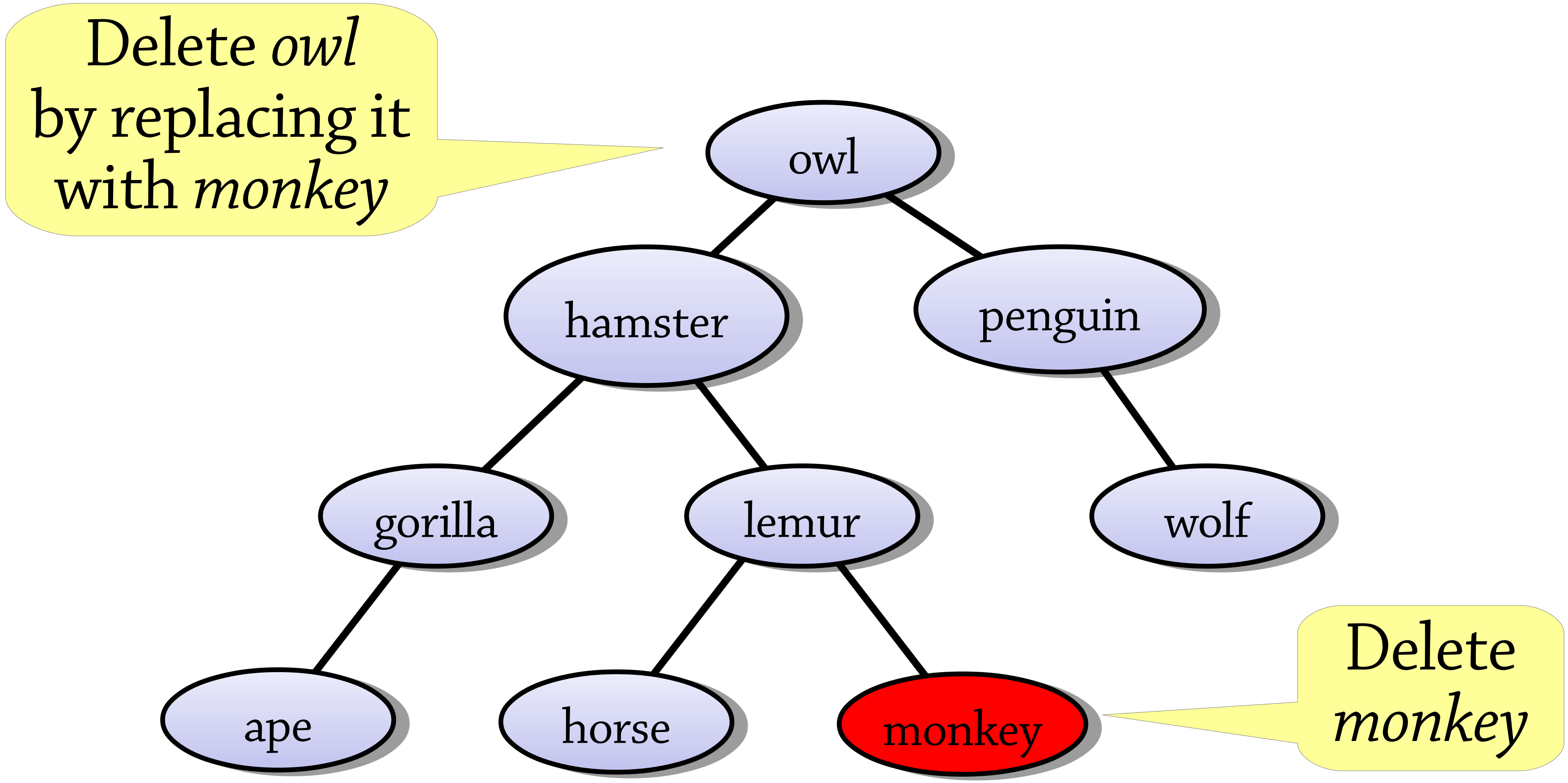
Deleting from a BST

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!

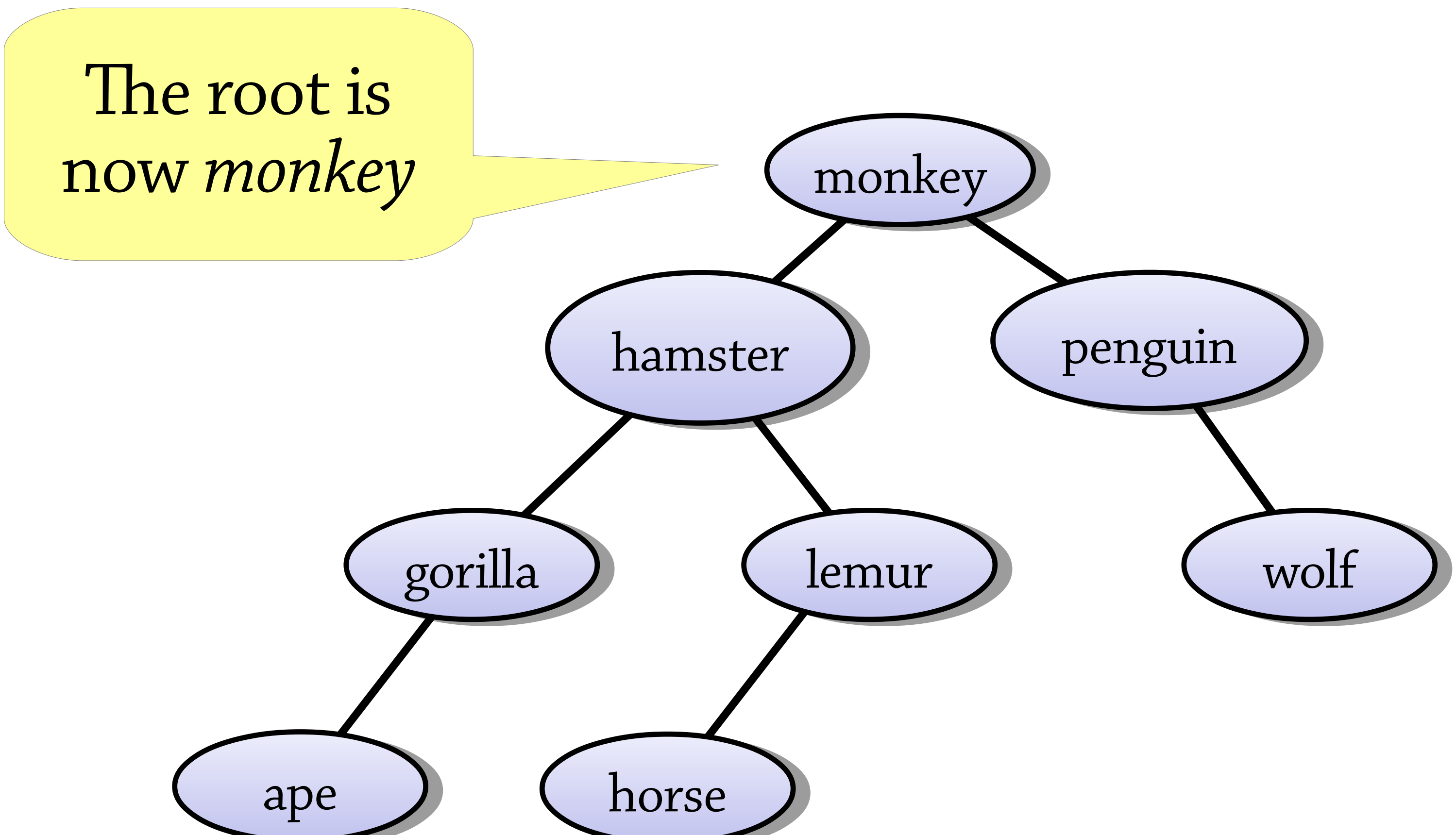
Deleting a node with two children

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



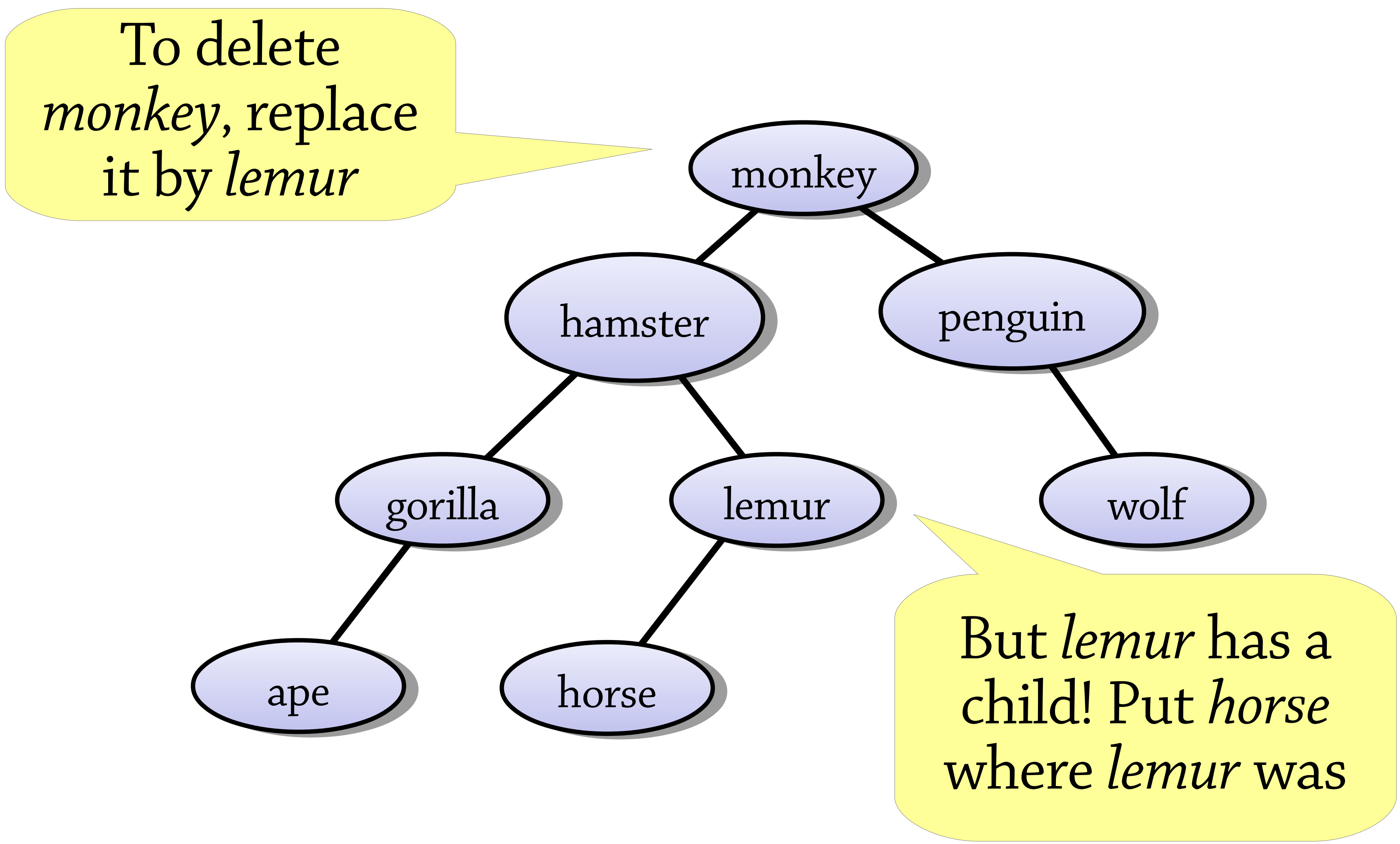
Deleting a node with two children

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



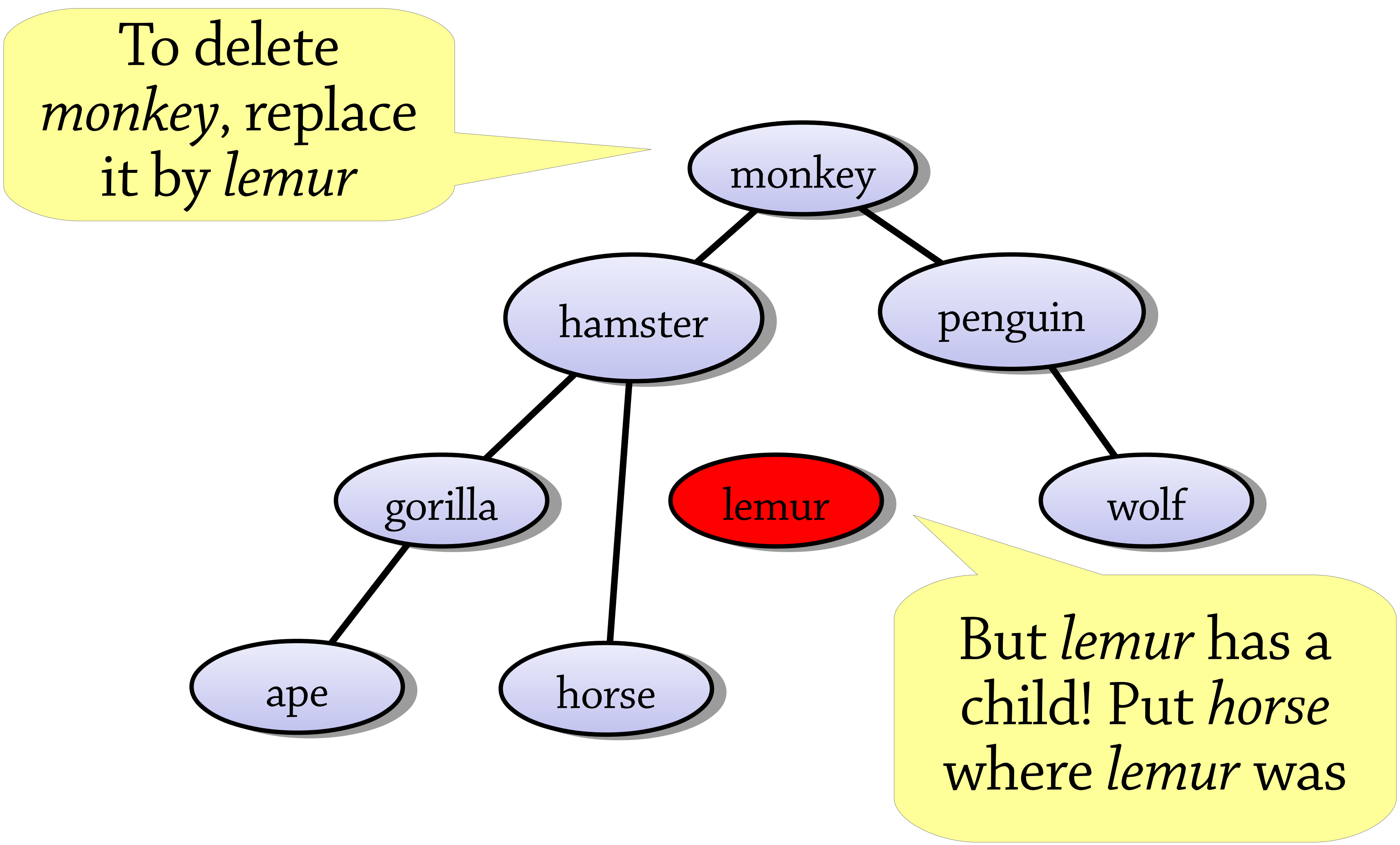
Deleting a node with two children

Here is the most complicated case:



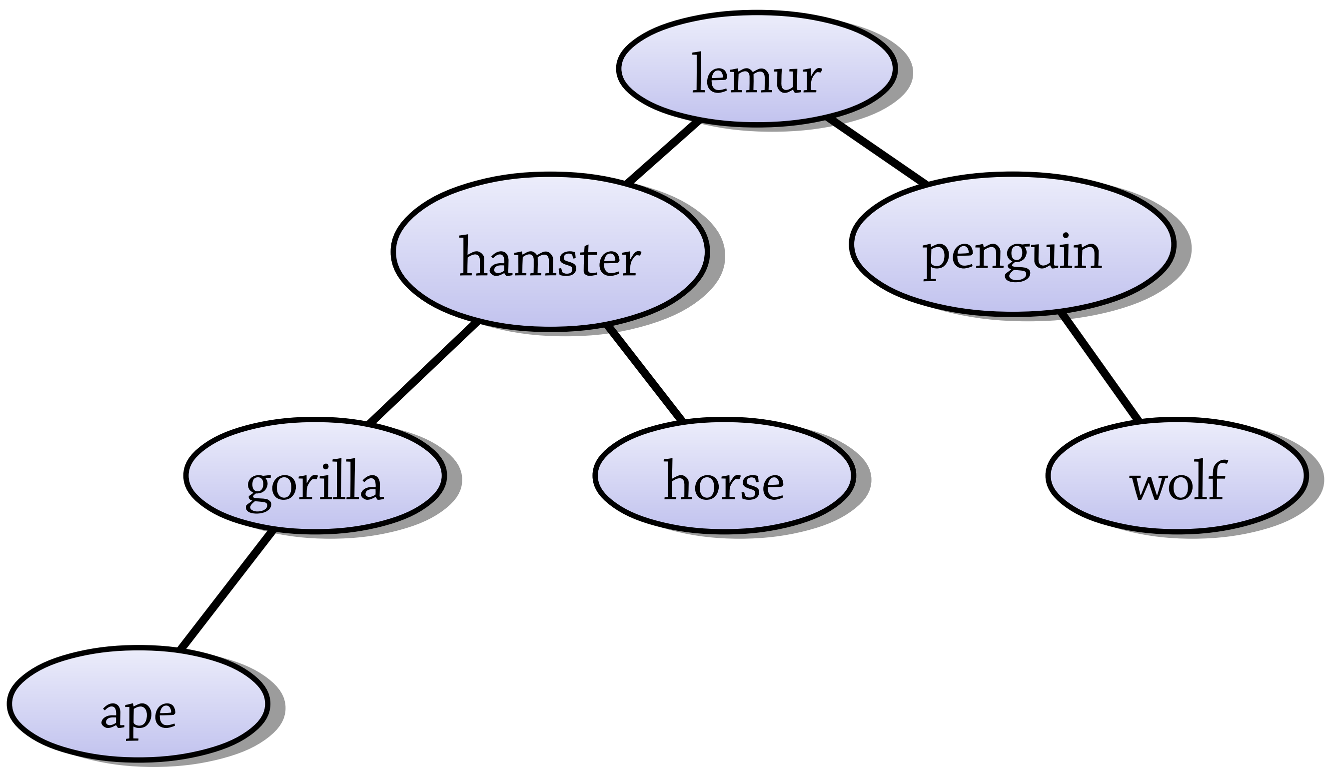
Deleting a node with two children

Here is the most complicated case:



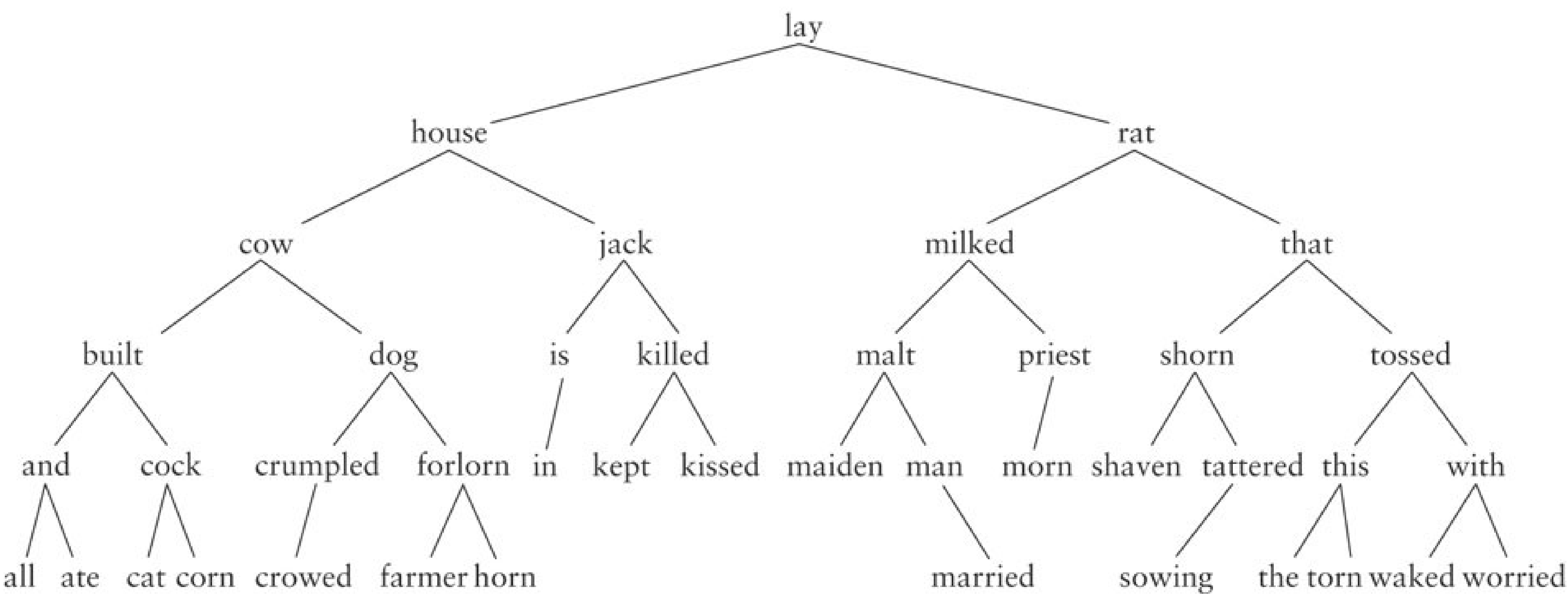
Deleting a node with two children

Here is the most complicated case:



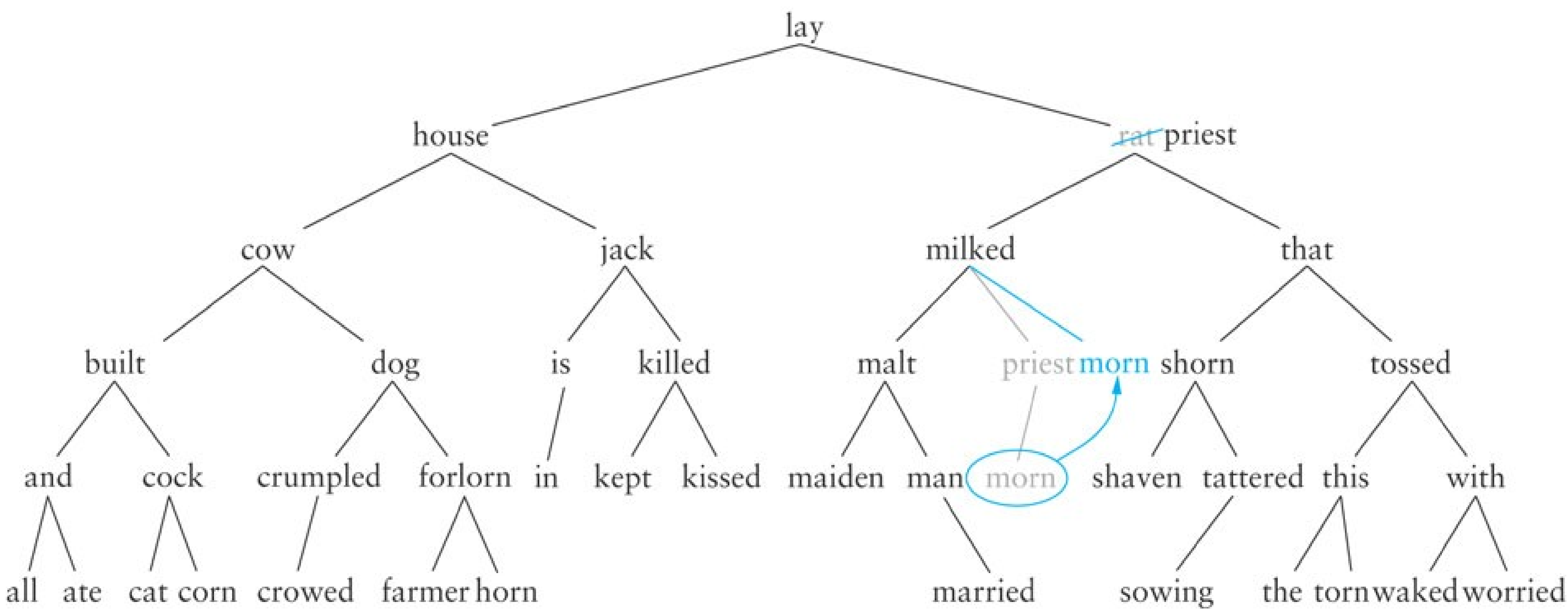
A bigger example

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

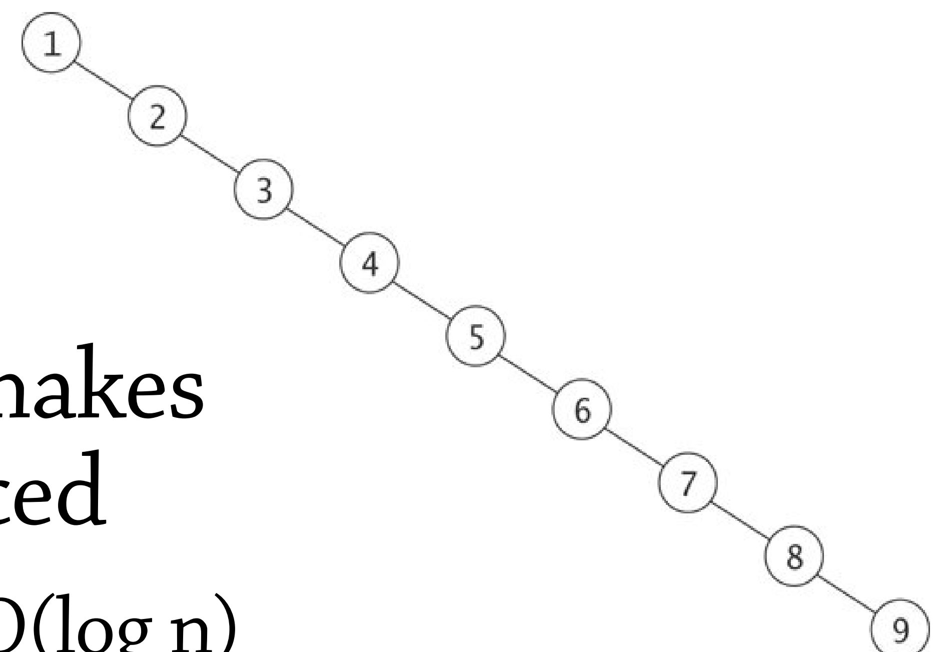
- Using the biggest value preserves the invariant (check you understand why)
- To find the biggest value: repeatedly descend into the right child until you find a node with no right child
- The biggest node can't have two children, so deleting it is easier

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