Two things on course website

Haskell compendium – describes lots of data structures in Haskell

- Find under “Resources”

Last year's exam

- Find under “Exam”

Sets, maps and binary search trees (6.7 – 6.8, 18, 19.1 – 19.3)

Trees

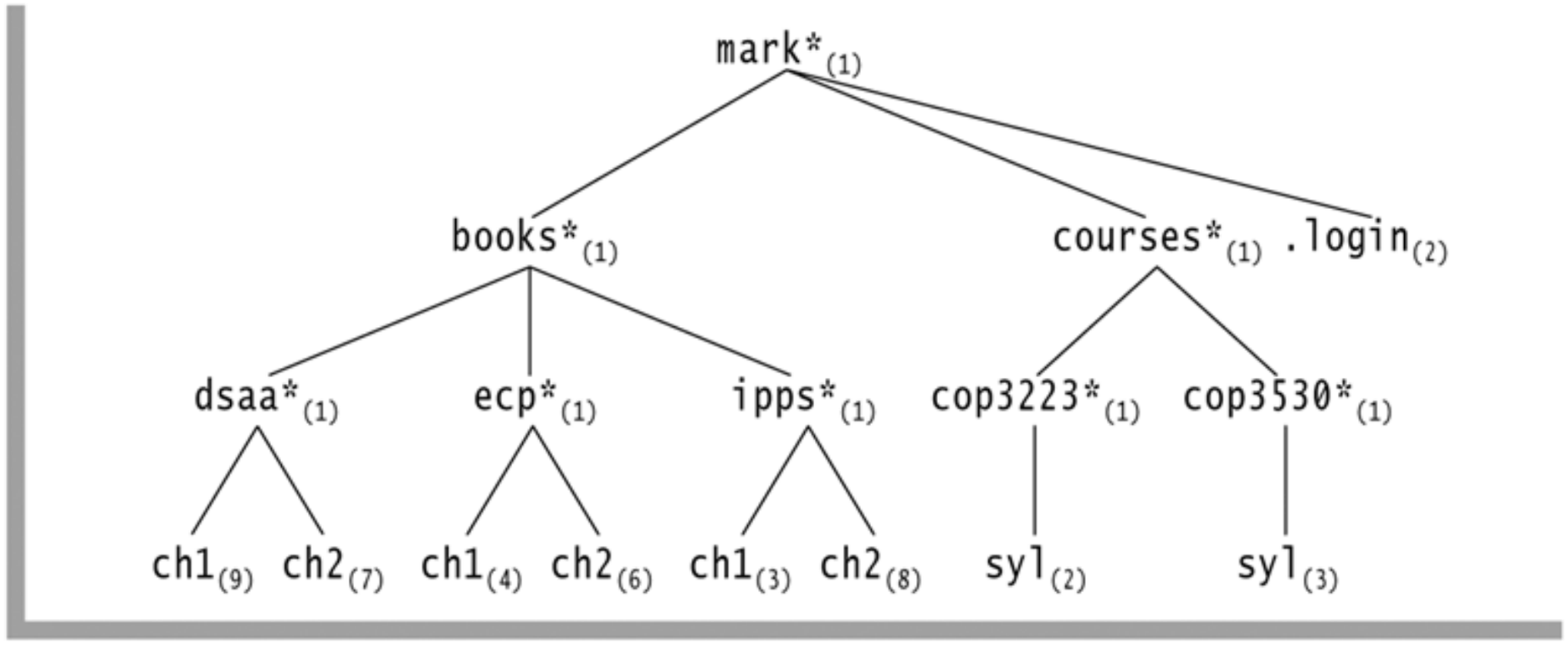
A *tree* is a hierarchical data structure

- Each node can have several *children* but only has one *parent*
- The *root* has no parents; there is only one root

Example: directory hierarchy

figure 18.7

The Unix directory with file sizes



Binary trees

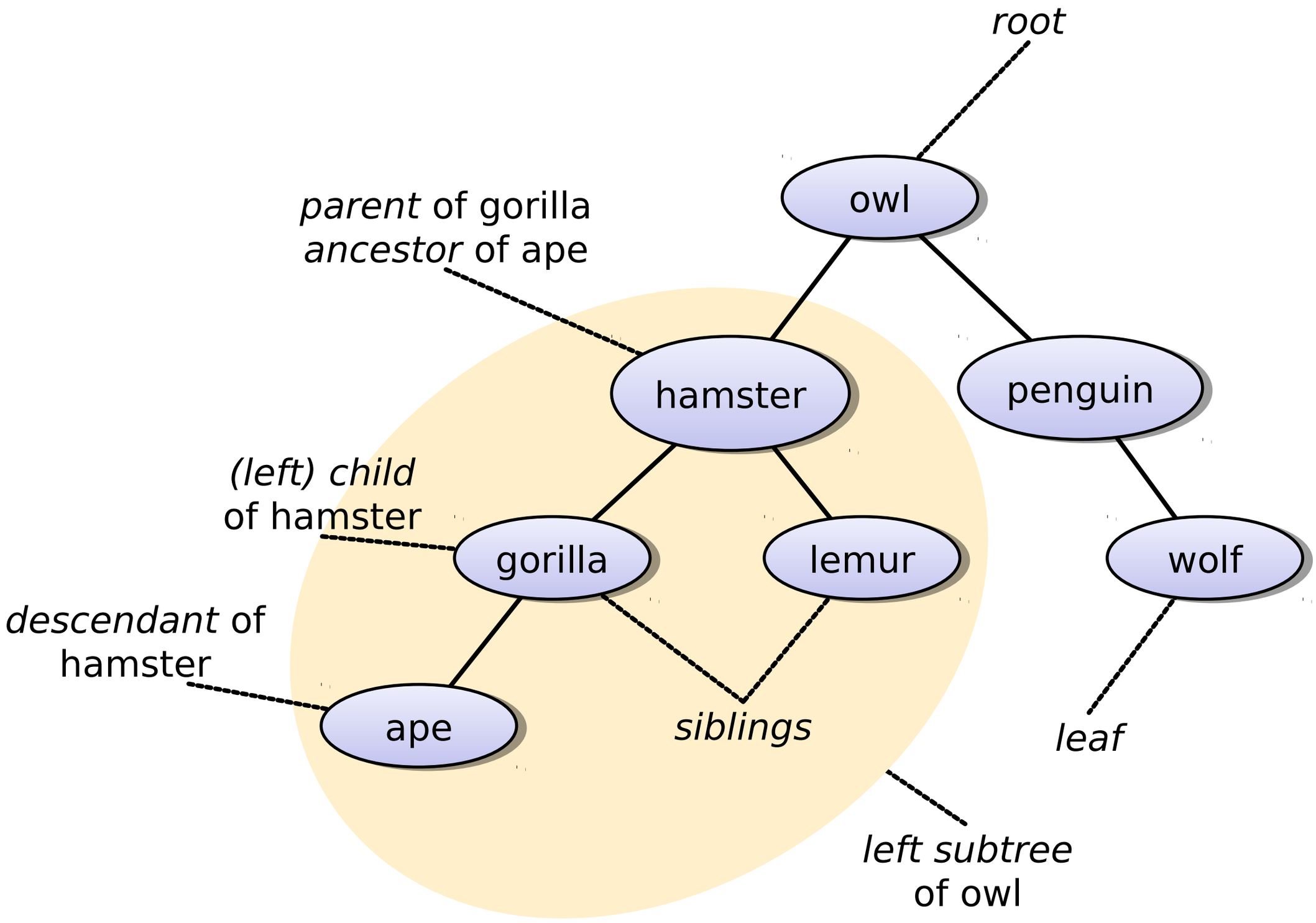
We will look at *binary trees*, where each node has at most two children

```
class Node<E> {
  E value;
  Node<E> left, right;
}
```

Can be null

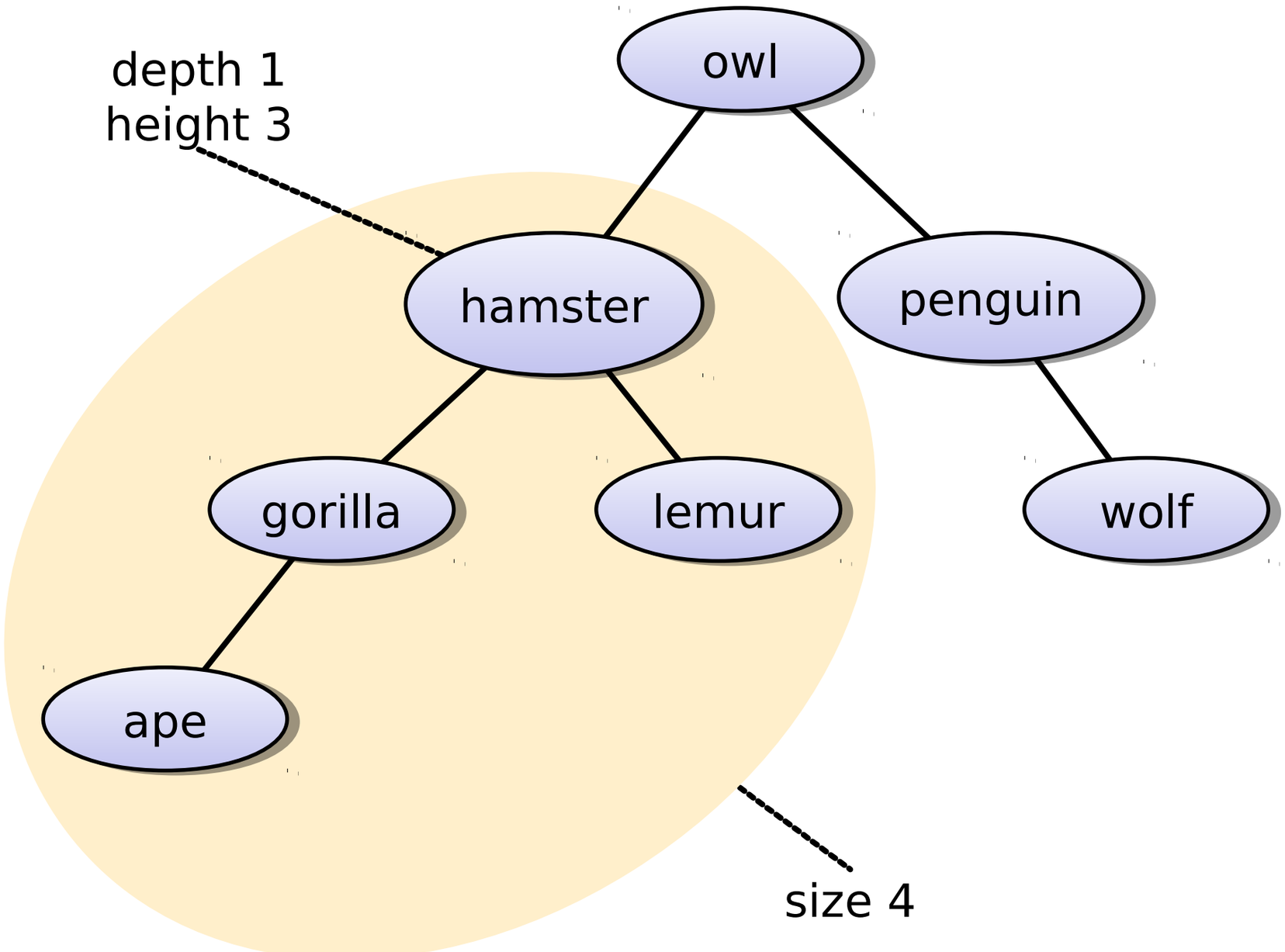
```
data Tree a
  = Node a (Tree a) (Tree a)
  | Nil
```

Terminology



Terminology

The *depth* of a node is the distance from the root
The *height* of a tree is the distance to the deepest leaf
The *size* of a tree is the number of nodes in it



Tree traversal

Traversing a tree means visiting all its nodes in some order

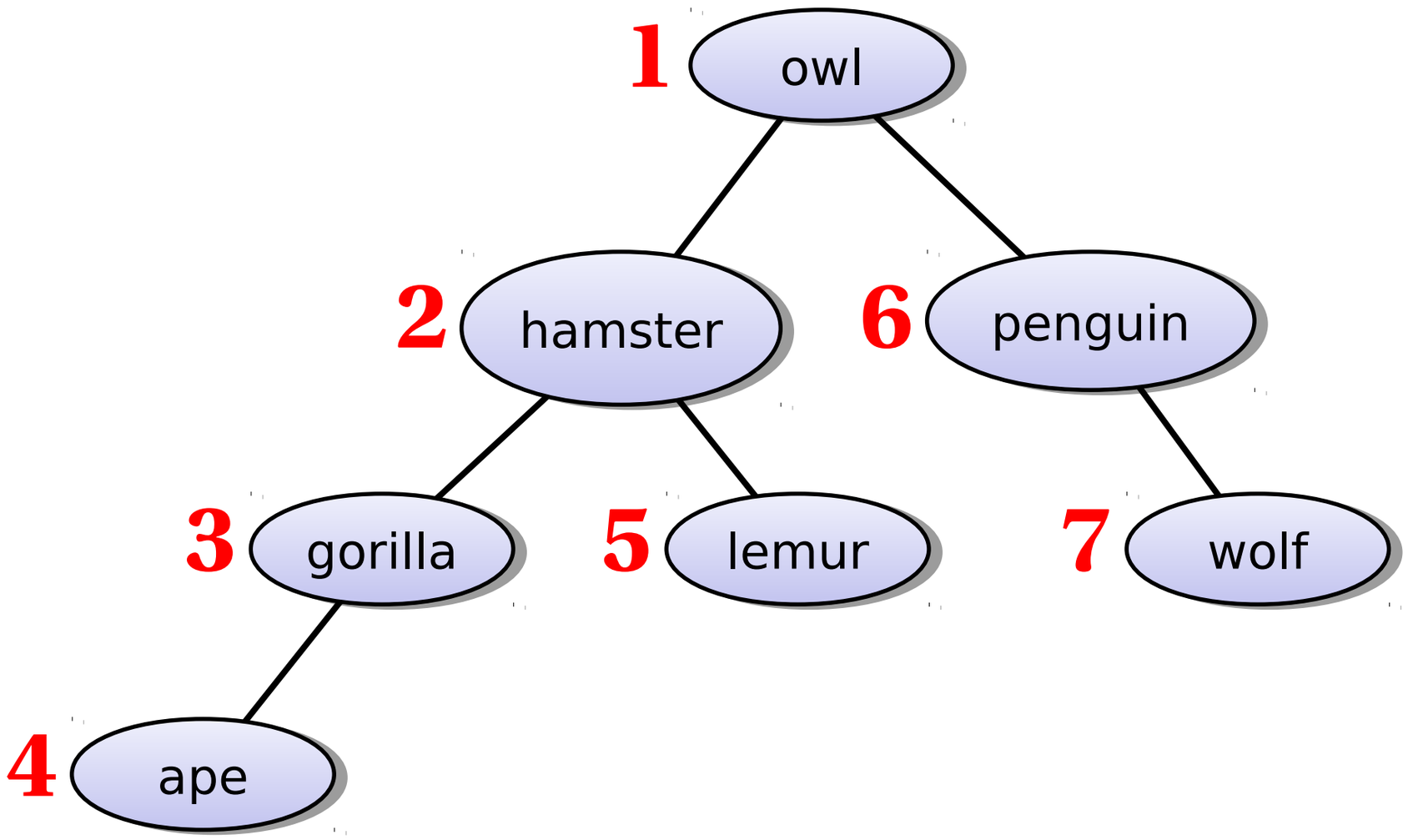
A *traversal* is an order to visit the nodes in

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

For each traversal, you can define an iterator that traverses the nodes in that order (see 18.4)

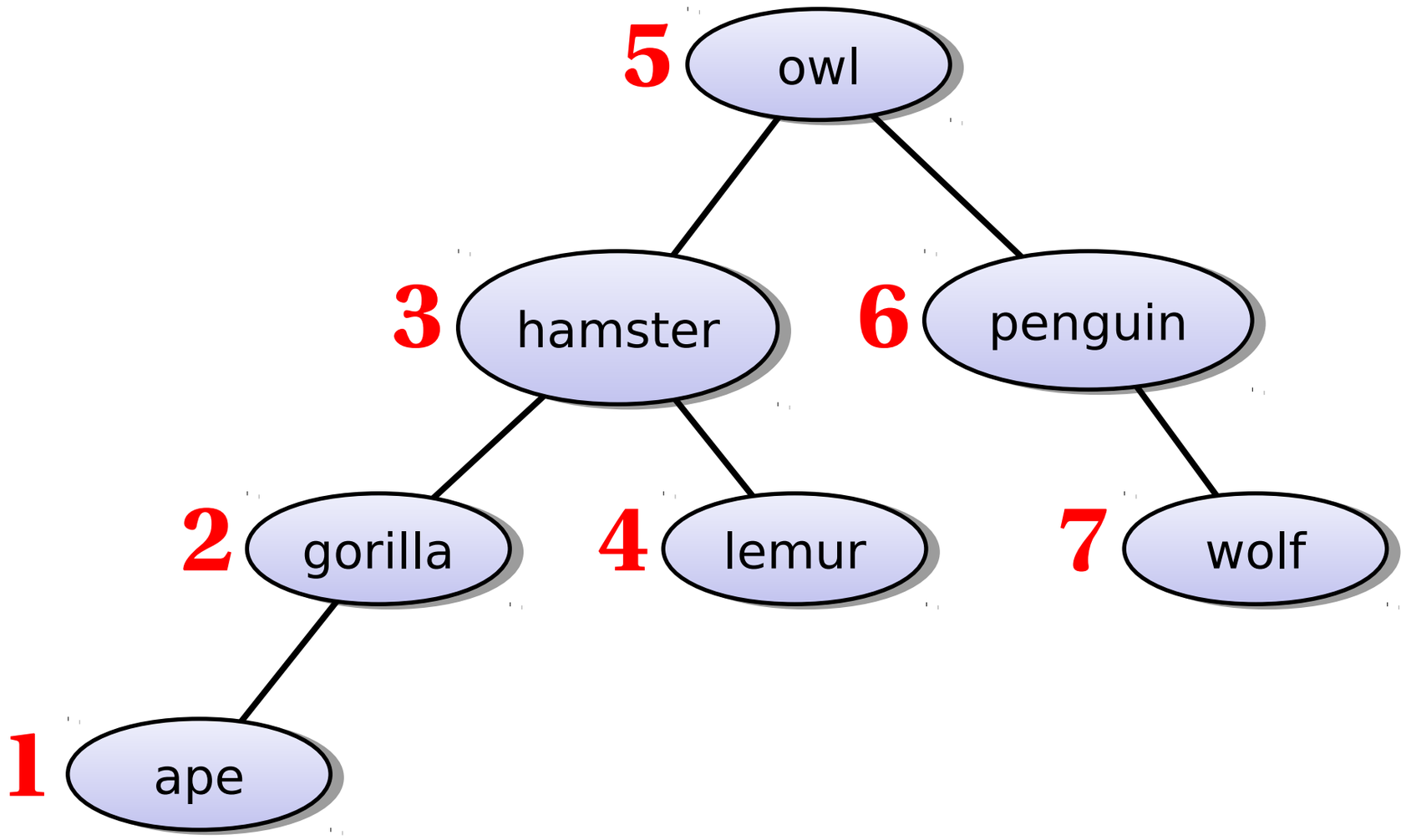
Preorder traversal

Visit root node, then left child, then right



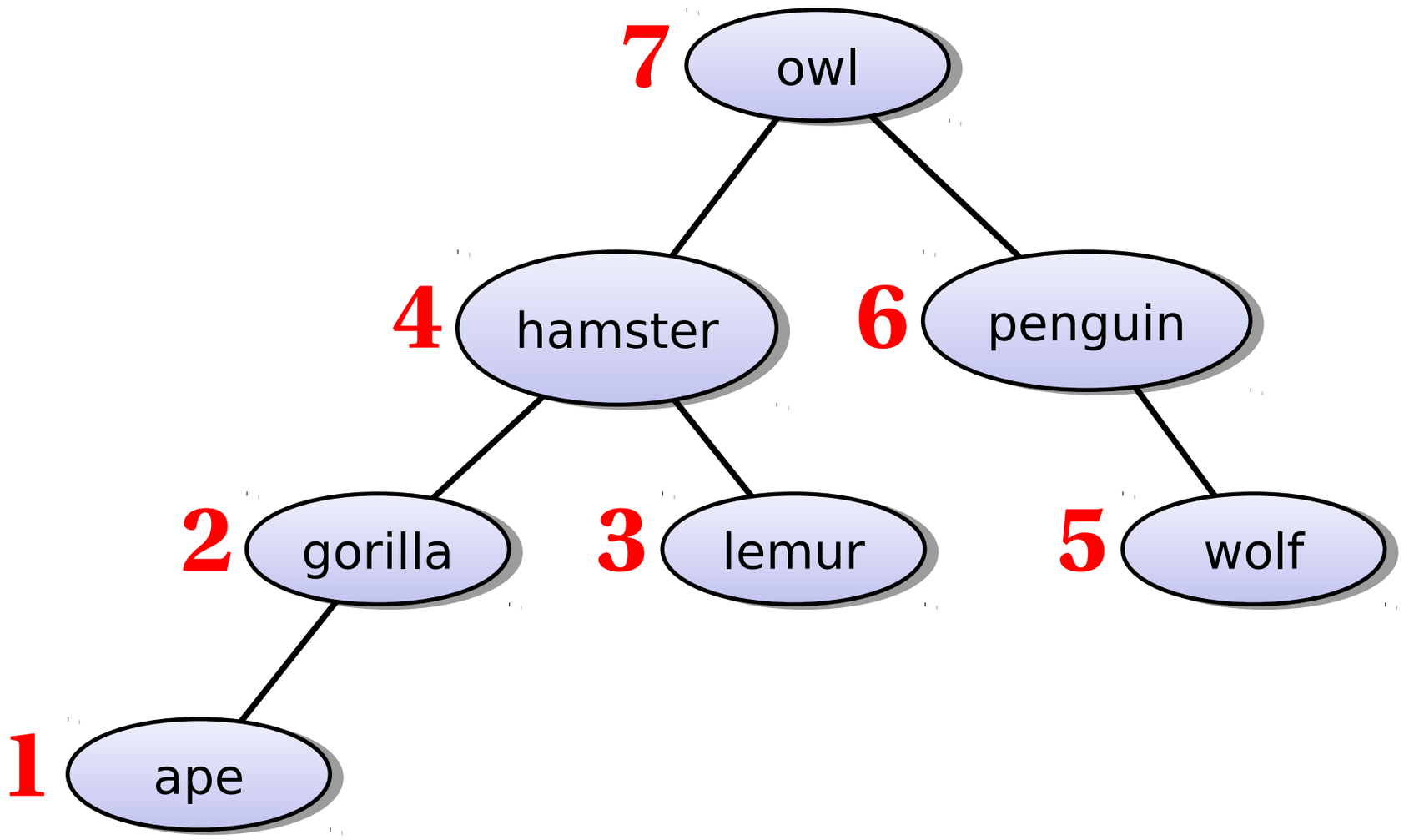
Inorder traversal

Visit left child, then root node, then right



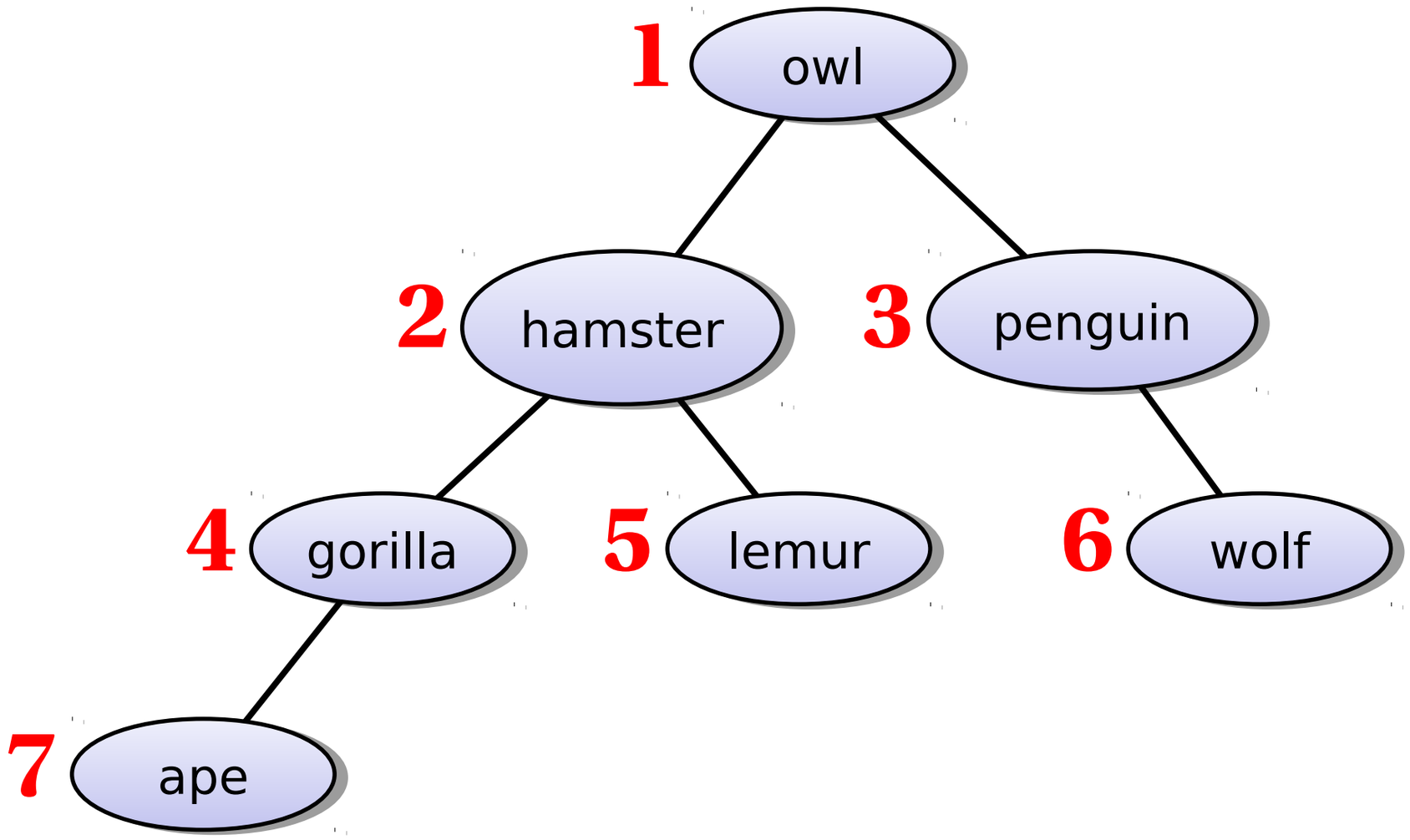
Postorder traversal

Visit left child, then right, then root node



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);
}
```

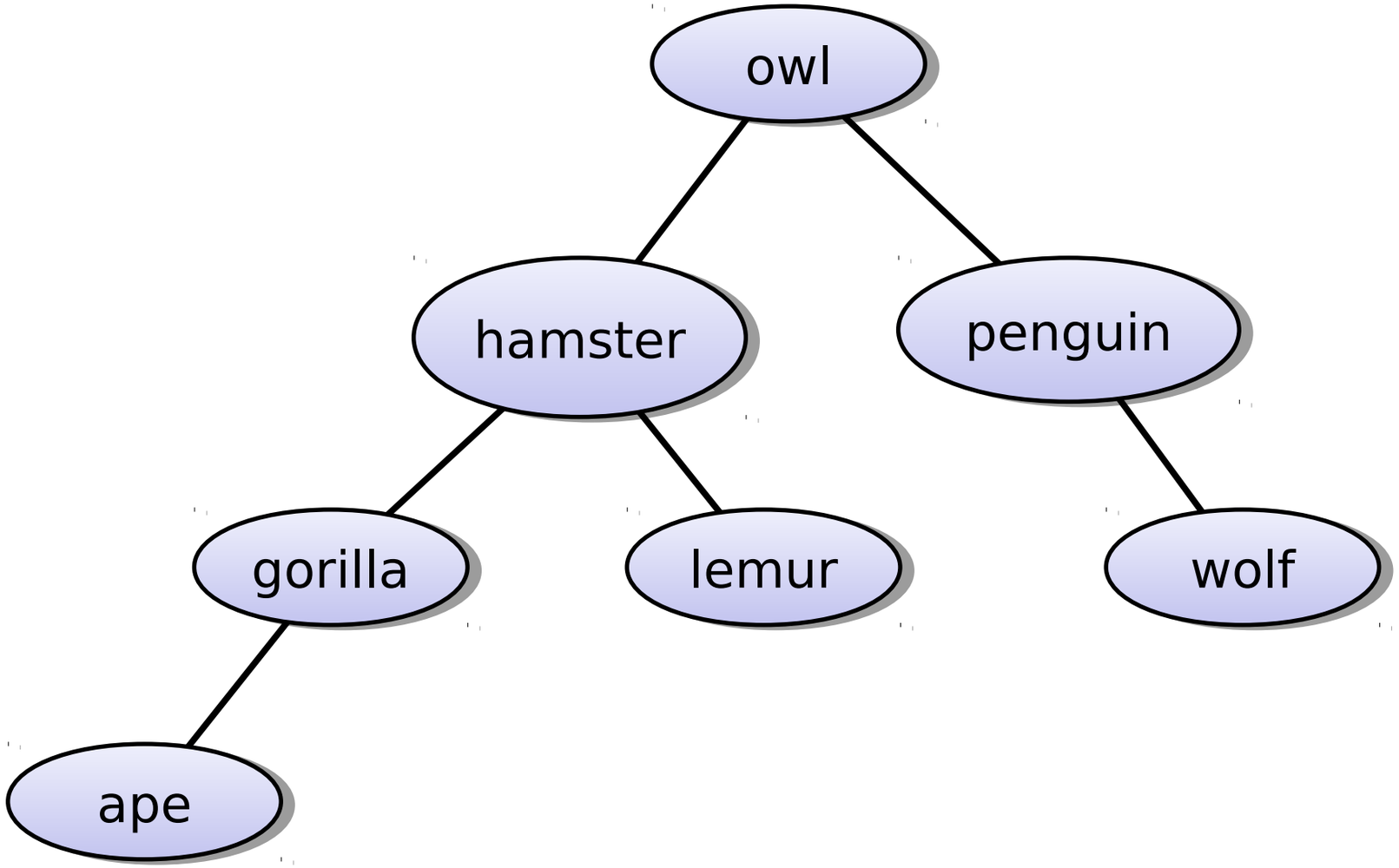
But nicer to define an iterator!

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

Level-order traversal is slightly trickier, and uses a queue – see 18.4.4

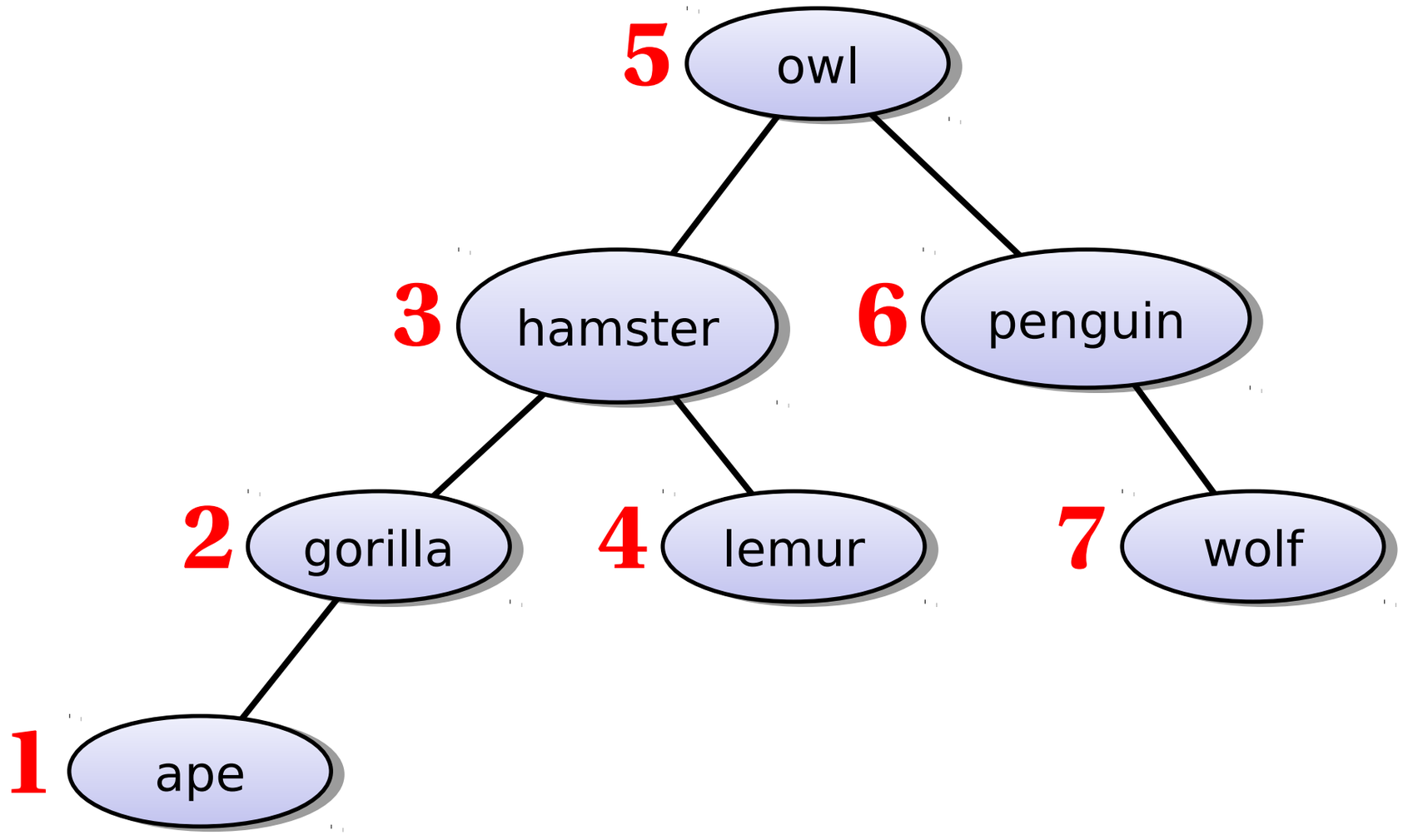
Binary search trees

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



Sorting a binary search tree

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



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

Invariants

"Every node is greater than all its left descendants, and less than all its right descendants": this is an *invariant*

- It holds of every binary search tree
- When using the BST, we can assume the invariant holds
- But when updating the BST, we must make sure to *preserve* the invariant: it should still hold afterwards

Invariants

When designing a complex data structure, the first thing you should decide is the invariant!

- If there is an invariant and you don't know what it is, you will probably end up with subtle bugs
- If you break the invariant, the program might not crash, it might just go wrong in mysterious ways – e.g., if you insert an item into the wrong place in a BST, you just won't be able to find it later

Checking the invariant

Write a method `boolean invariant()` that checks whether your data structure's invariant holds.

Then before and after every operation, write `assert invariant()` - this will throw an error if the invariant doesn't hold

This finds many tricky data structure bugs!

Almost all languages support assertions. **Use them!** There is normally an option not to check assertions – in Java you have to run with `-ea` to check them.

BST invariant (sketch)

```
boolean invariant() {
  return checkNode(root);
}
private boolean checkNode(Node<E> node) {
  if (node == null) return true;

  if (!checkNode(node.left)) return false;
  if (!checkNode(node.right)) return false;

  for (E x : allDataValues(node.left))
    if (x ≥ node.data) return false;
  for (E x : allDataValues(node.right))
    if (x ≤ node.data) return false;
  return true;
}
```

A bit of work to write, **but worth it** when it finds bugs!

BST invariant (Haskell)

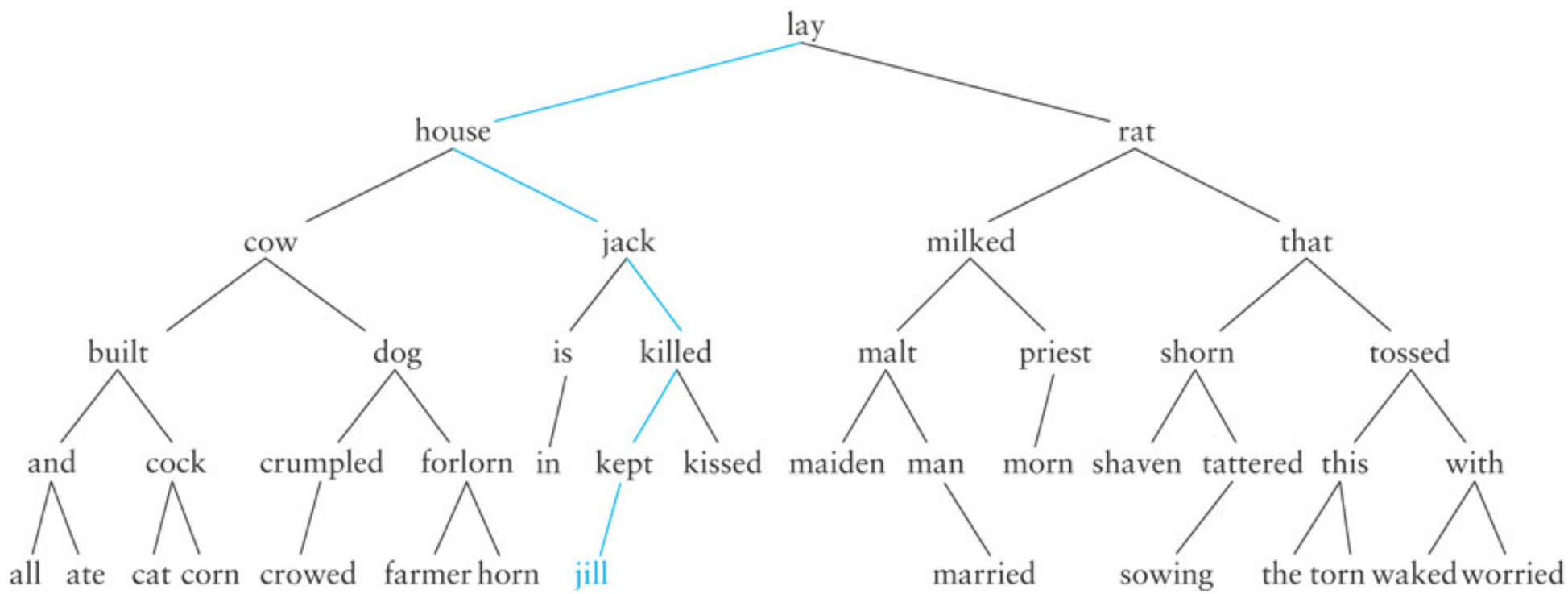
```
invariant :: Ord a => Tree a → Bool
invariant Nil = True
invariant (Node x l r) =
  all (< x) (values l) && all (> x) (values r)

values Nil = []
values (Node x l r) = values l ++ [x] ++ values r
```

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



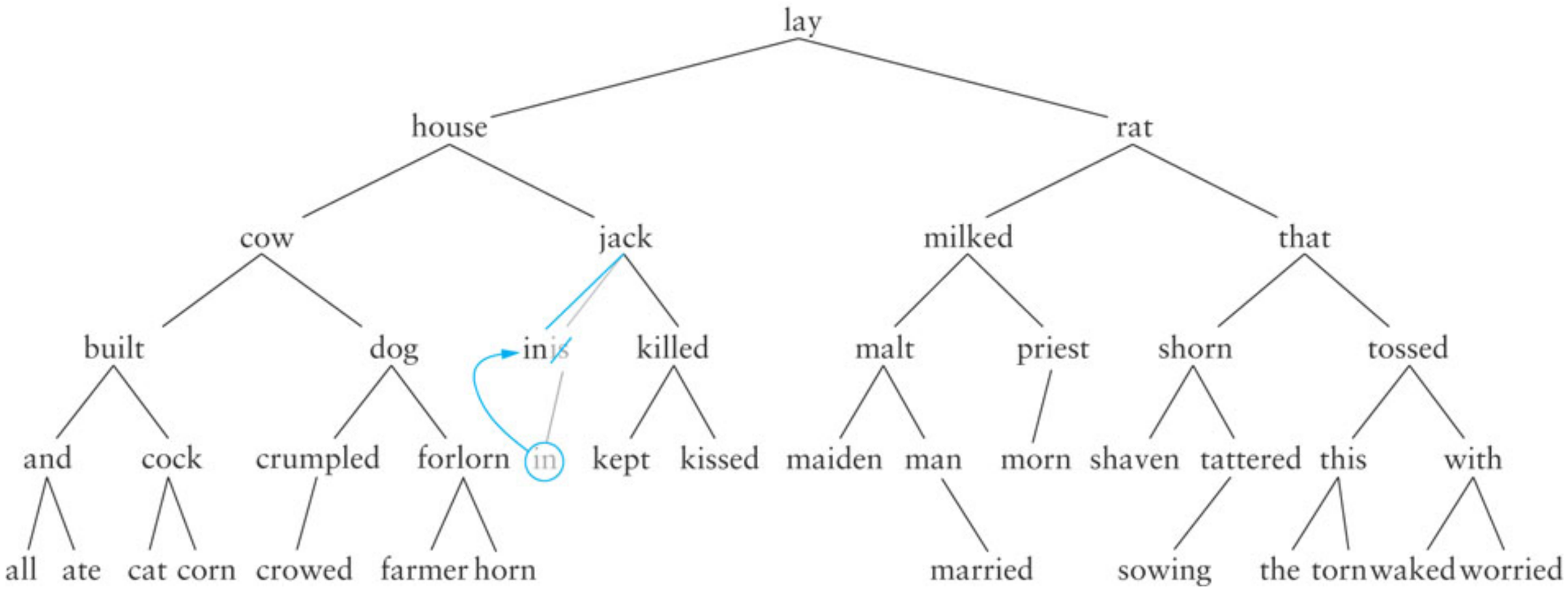
Deleting from a BST

To delete a value 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...?

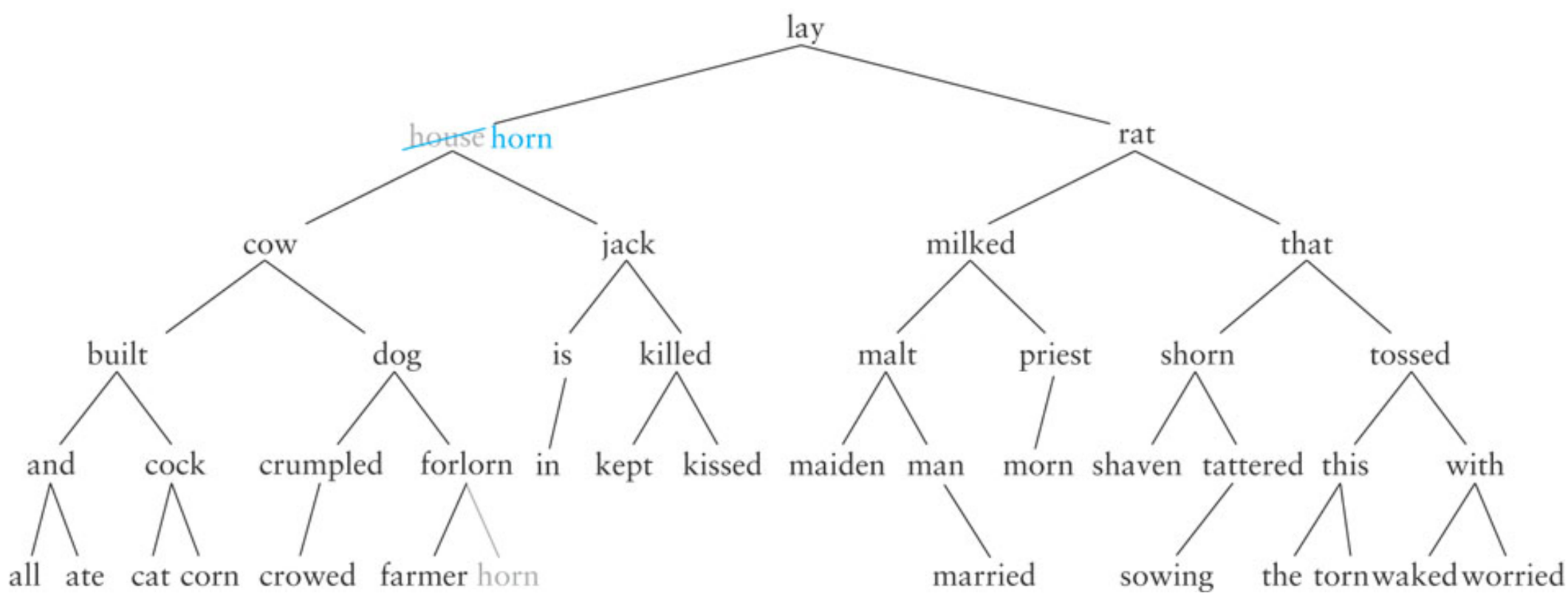
Deleting a node with one child

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



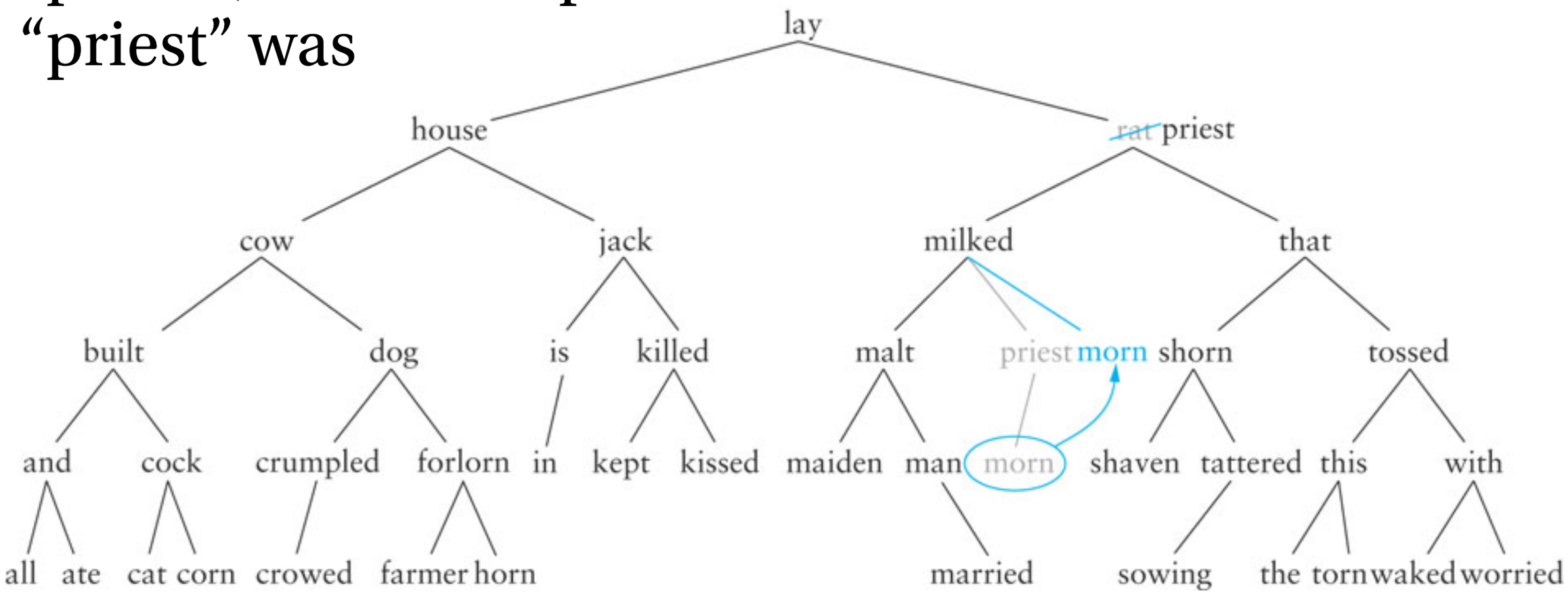
Deleting a node with two children

Replace the node with *the biggest (rightmost) node from its left subtree* (or the smallest from the right subtree) – there is no node between these two in order, so we won't break the invariant



Deleting a node with two children

The rightmost node of the left subtree might have a child! In that case, we connect that child where the rightmost node was. Here, we replace “rat” with “priest”, and move priest's left child “morn” where “priest” was



Deleting a node with two children

Look at the left subtree and find the rightmost (greatest) node

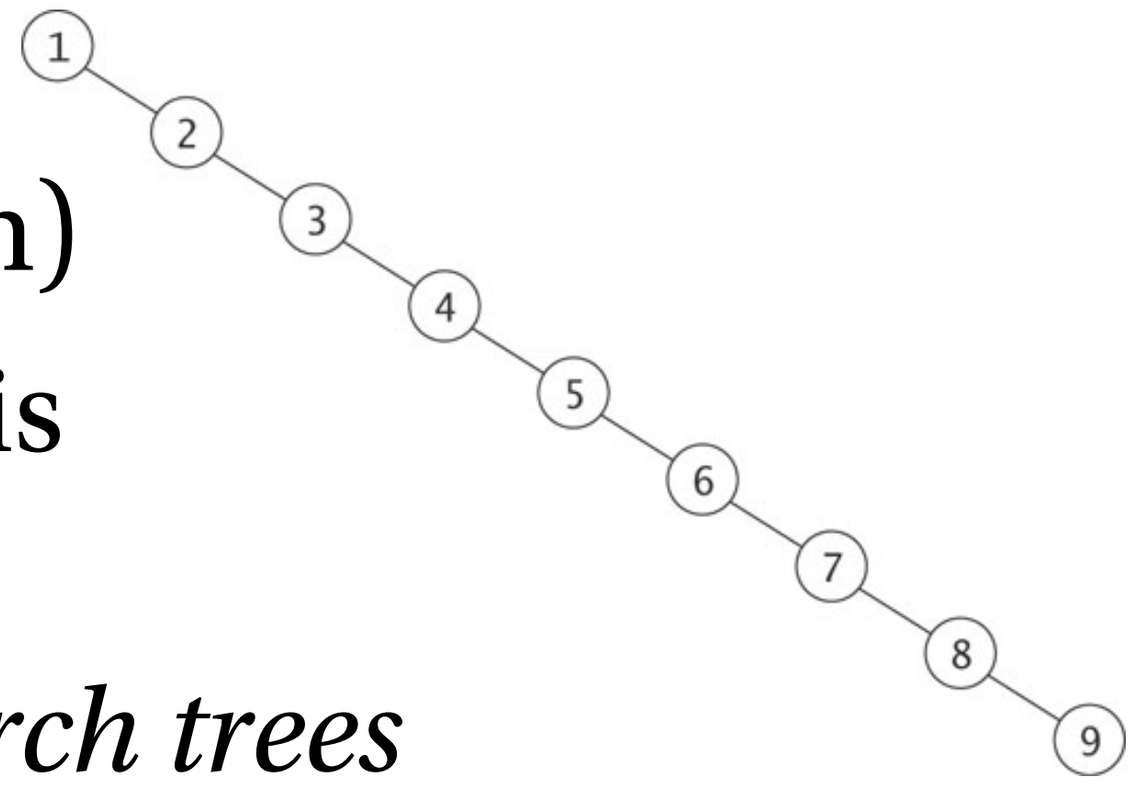
Delete that node as before (it can't have two children because it's rightmost)

Replace the node we're deleting with that rightmost node

Complexity of BST operations

A BST can be severely *unbalanced* (when?) – then finding an element is O(n)

If it is balanced, the complexity is O(log n)



General complexity is O(height of tree)

Balanced binary search trees (later) make sure the tree is balanced so complexity is O(log n)

Set ADT

The set ADT looks like this in Java:

```
interface Set<E> extends Collection<E> { }
```

What!

Well, `Collection` already contains all the set operations: add, remove, member, etc.

The difference between `Set` and `Collection`: if you add duplicate elements to a `Set`, they're ignored – a `Collection` might let you add duplicates

Map ADT

A map is a collection of key/value pairs.

Important methods of `Map<K,V>`:

```
boolean containsKey(K key);
V get(K key);
void put(K key, V value);
V remove(K key);
Set<K> keySet();
Collection<V> valueSet();
Set<Entry<K,V>> entrySet();
// Entry<K,V> has methods
// getKey, getValue, setValue
```

Pretty well every language has something similar.

Summary

Binary trees

- Hierarchical data structure
- Much standard terminology
- Traversals: preorder, inorder, postorder, level-order

Binary search trees

- O(log n) insert, delete, lookup *if balanced*
- We will see later how to keep a binary search tree balanced
- Java code in book (chapter 19.1), Haskell code in compendium (file BinarySearchTree.hs)

Data structure invariants

Sets and maps