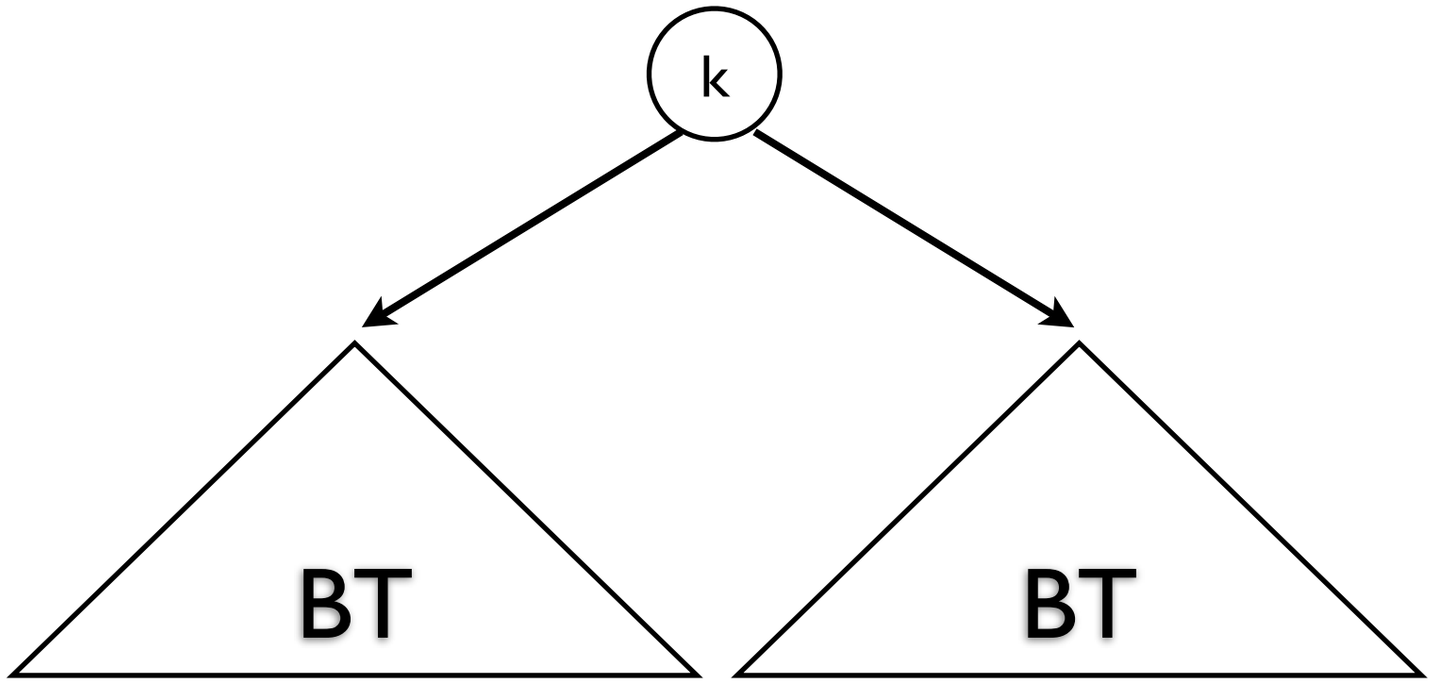
Binary Trees

Recursive Definition

A Binary Tree can also be defined recursively:

A binary tree is:

- empty or
- a node with a value k, a Binary Tree as its left child and a Binary Tree as its right child

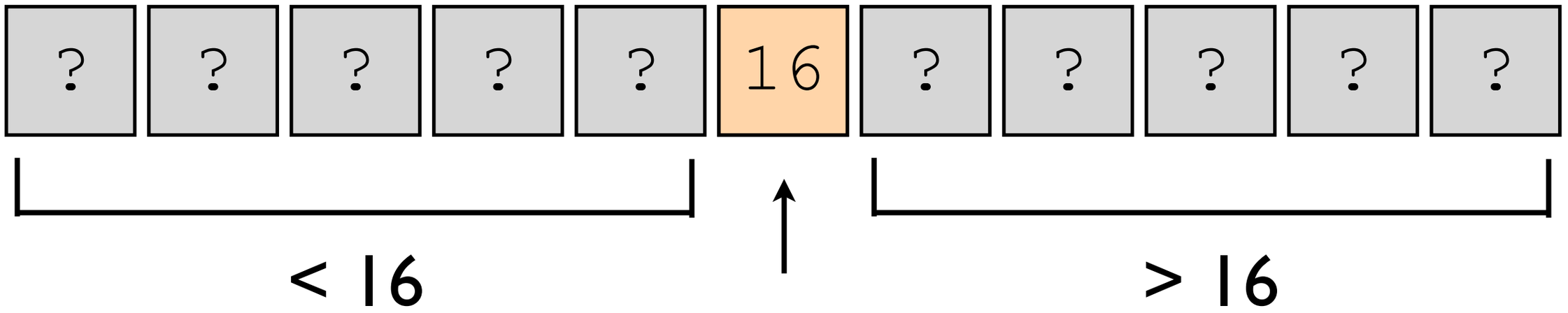


4

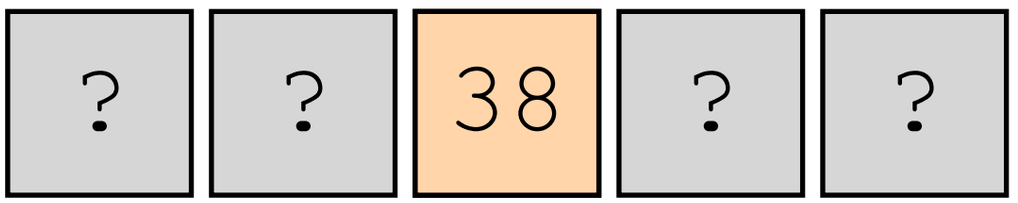
Binary Search

Binary Search

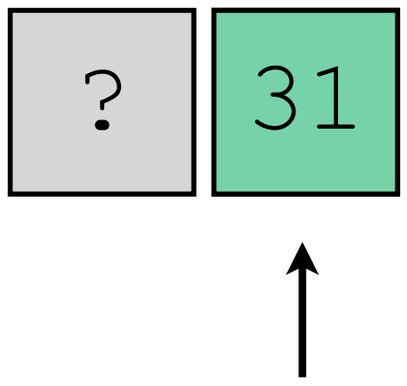
Question: is 31 in this sorted list?



Recurse on half the list



Recurse on half the list

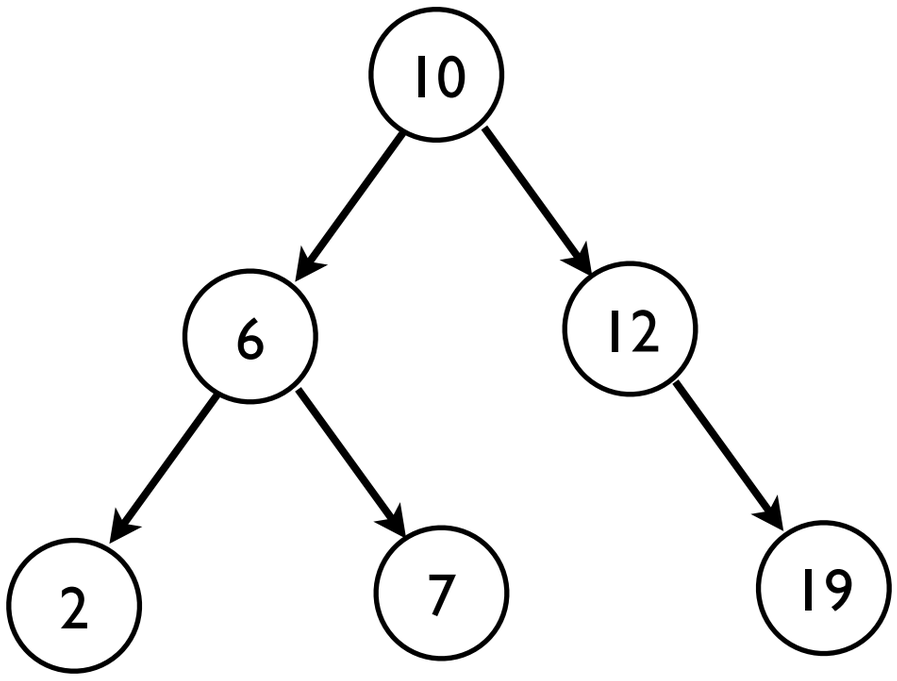


Binary Search Trees

Binary Search Tree

A **Binary Search Tree** (BST) is a special type of binary tree that adheres to the following property:

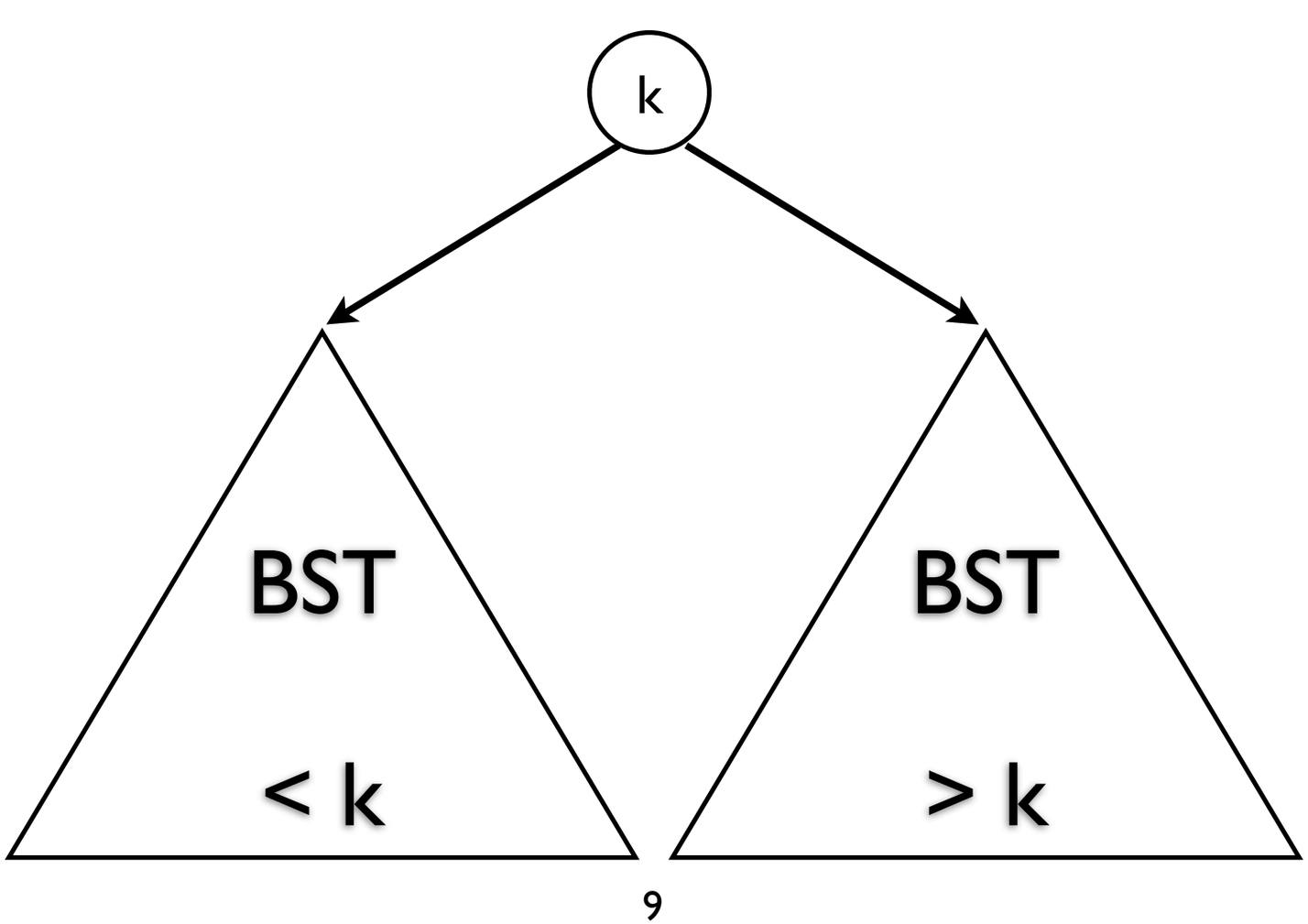
All nodes in the left subtree of node with value k have values that are less than k. All nodes in the right subtree of node with value k have values that are greater than k.



Binary Search Tree

A Binary Search Tree (BST) can also be defined recursively:

A BST is either empty, or it's composed of a root node with a BST of values less than the root value on the left branch and a BST of values greater than the root value on the right branch.



BST Operations

Common operations on BSTs include:

searching for a value

inserting a new node

removing a node

traversing the tree (visiting all nodes in some order and doing something to each)

attaching a subtree at a node

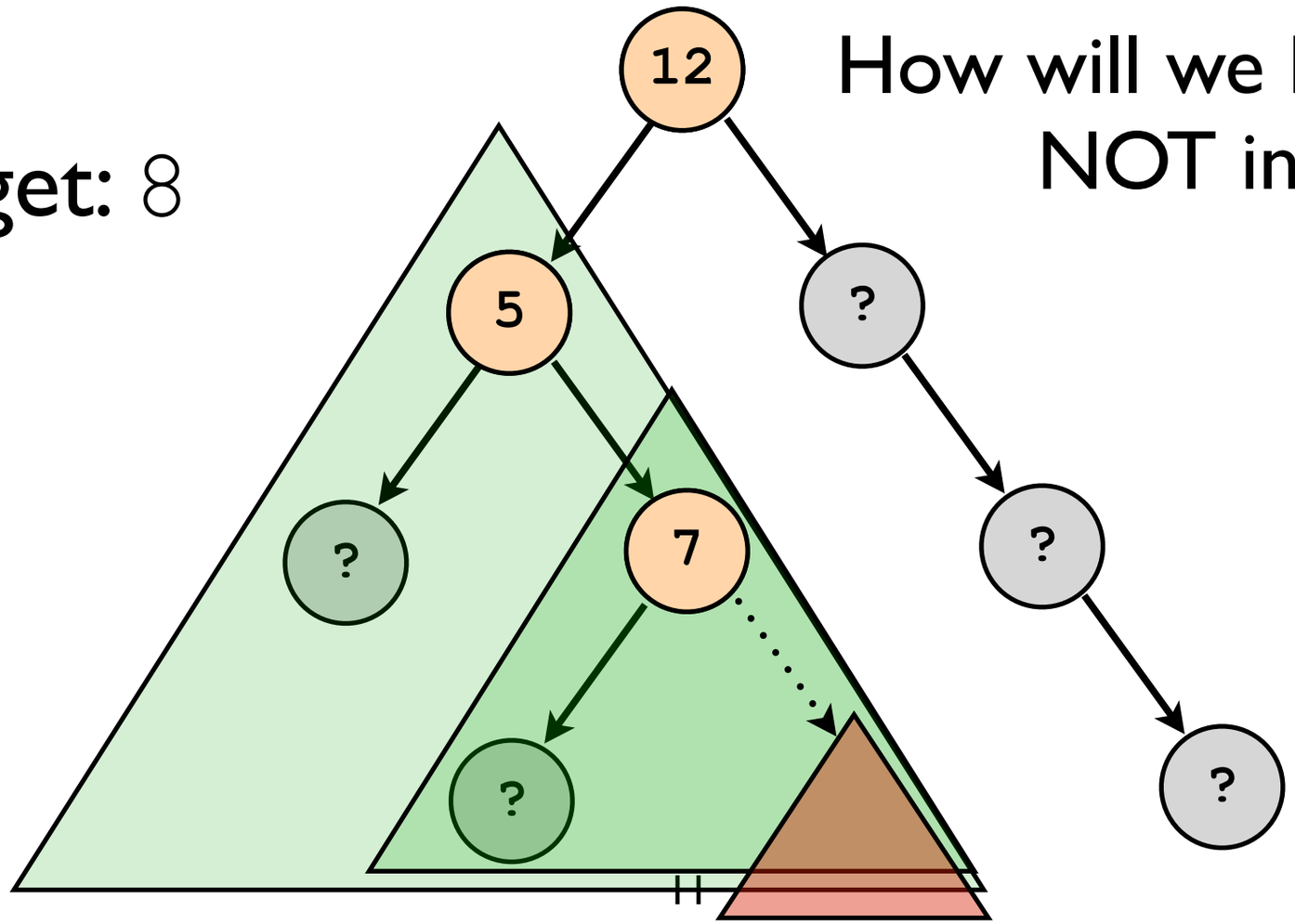
removing a subtree

BST Search

Searching in a BST follows the binary search pattern we saw at the start.

We compare the target value to the root's value and use it to decide which branch of the root to recurse on.

Target: 8



How will we know an item is NOT in the tree?

BST Insert

Inserting into a BST has a very specific secondary purpose:

Do **NOT** mess up the BST property of the resulting tree.

So, we should insert a new Node where we would've expected to find that value.

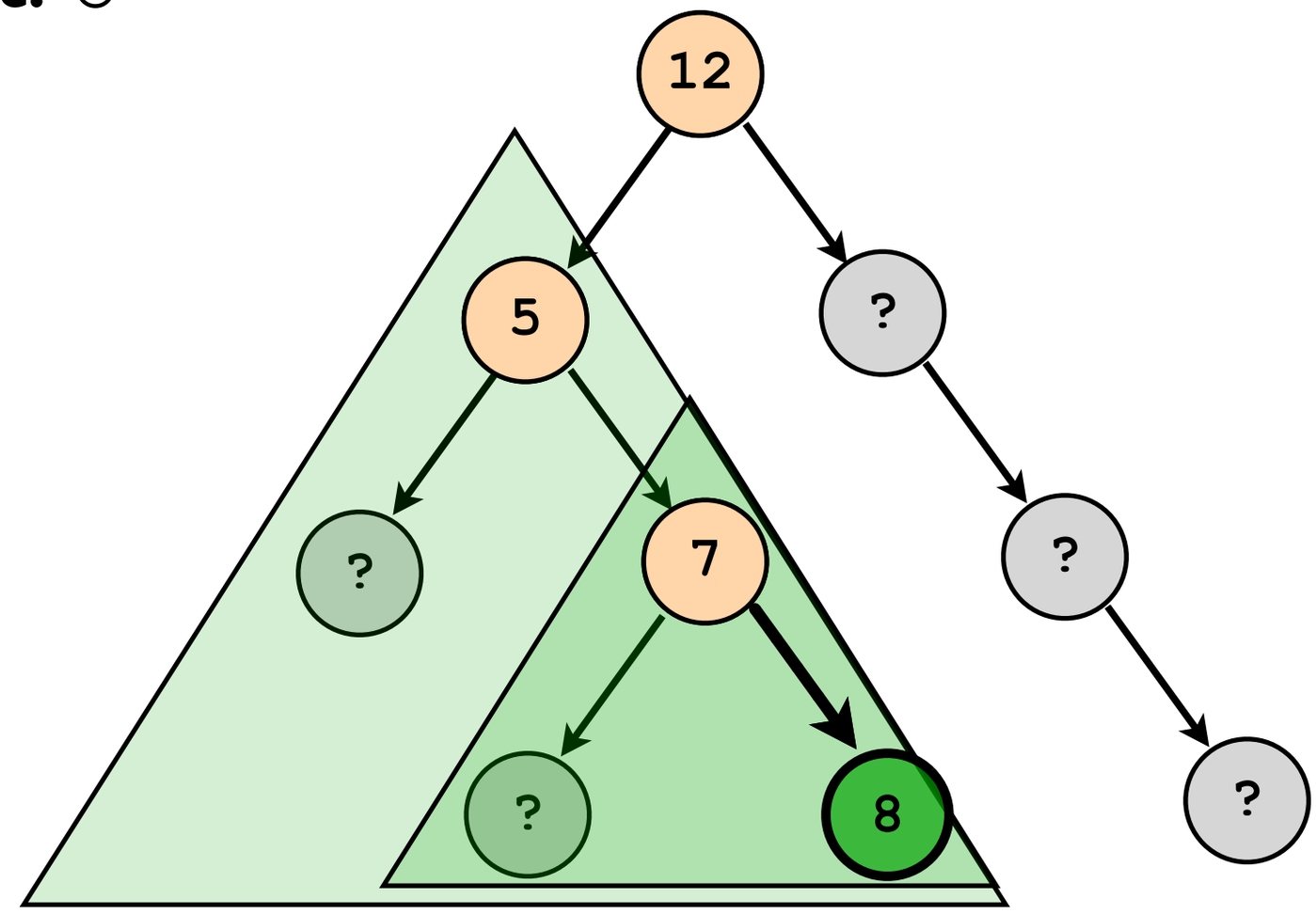
In fact, a BST insert requires a BST search.

It also turns out that for every new value, there is exactly one spot in the BST where it correctly belongs.

(By the way, our BST's will only allow unique values)

BST Insert

Target: 8



BST Deletion

BST Delete

Deletion: removing the node with a particular value from the tree, if such a node exists.

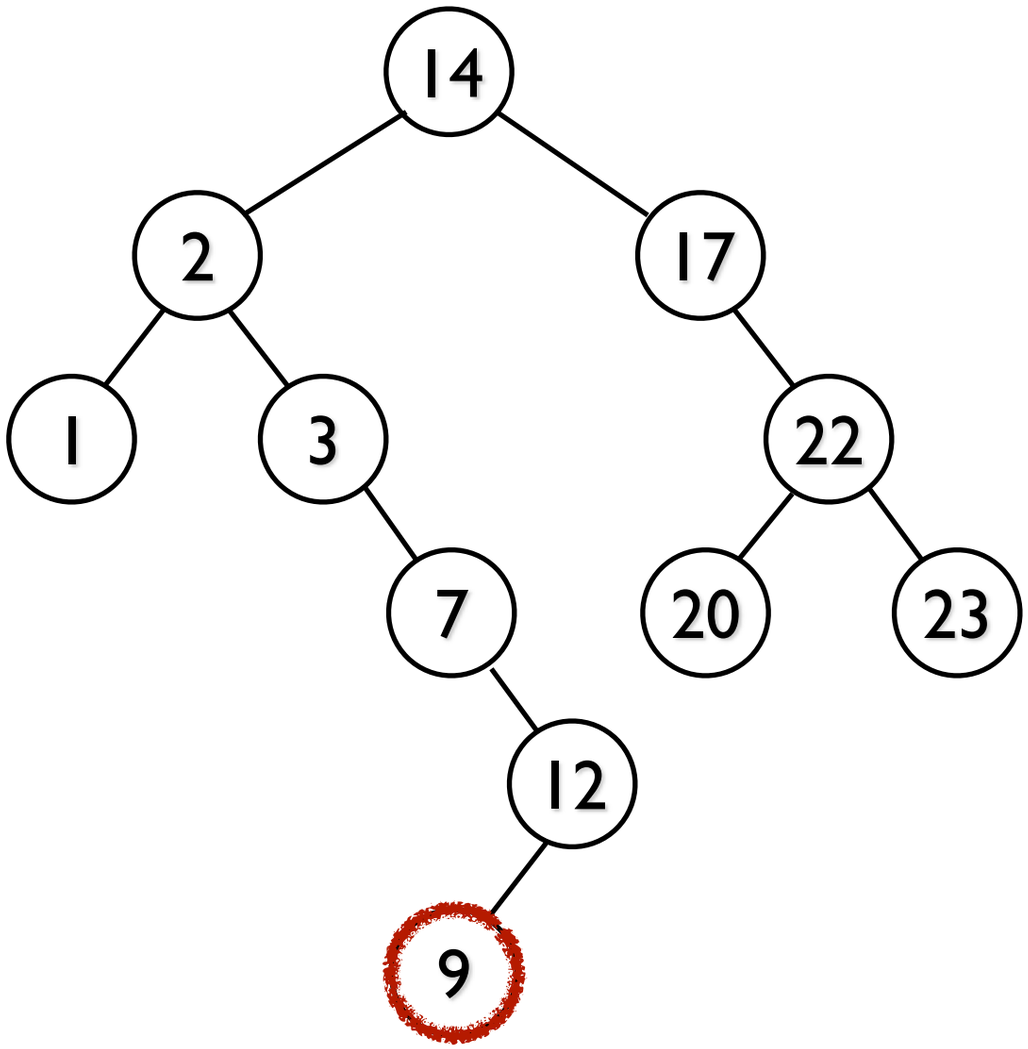
Like inserting, deleting from a BST has to respect and not break the BST property of the tree.

However, deletion will require more work and more caution to preserve the BST property.

Also, since we're changing the topology of the tree, we will need to pass information about that changing topology all the way up to the root of the tree.

Case 1: Leaf Node

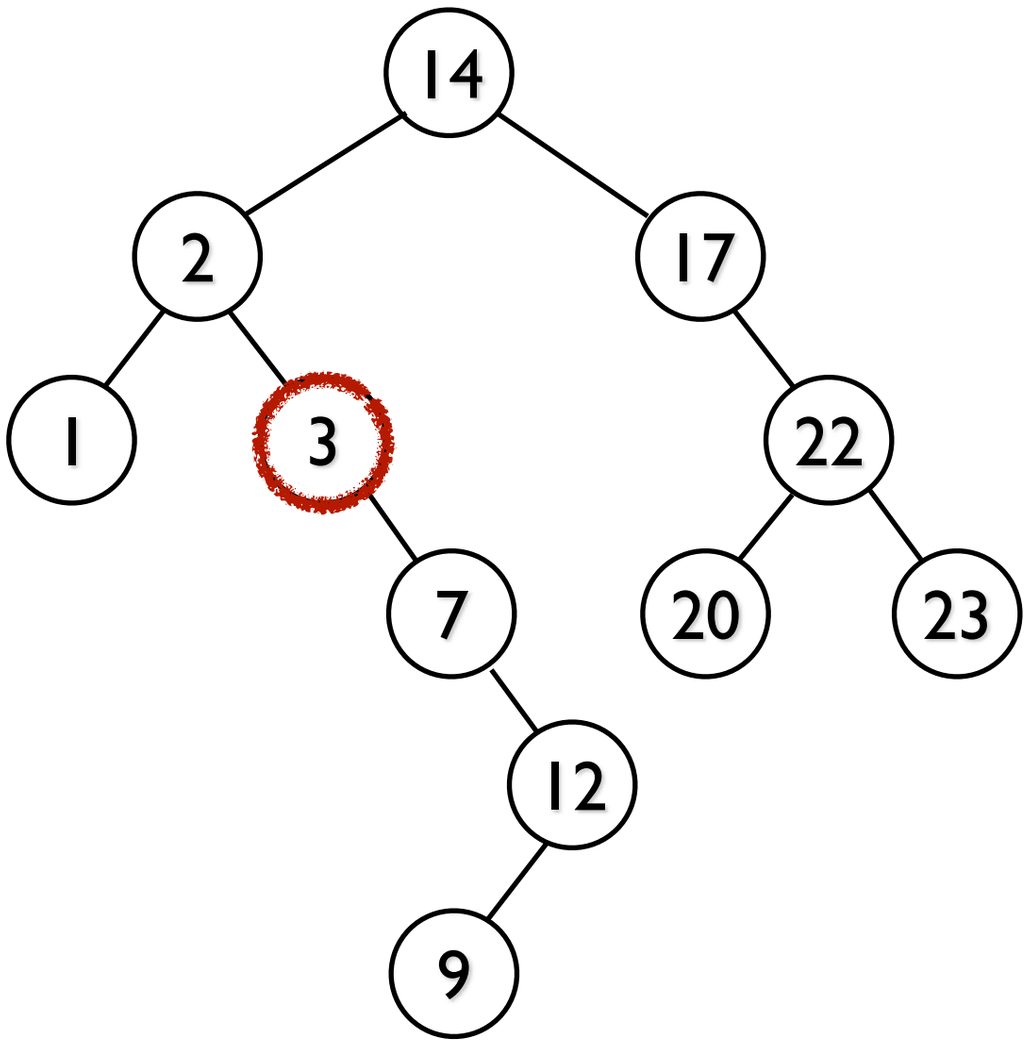
Target: 9



Simple: delete the node

Case 2: One Subtree

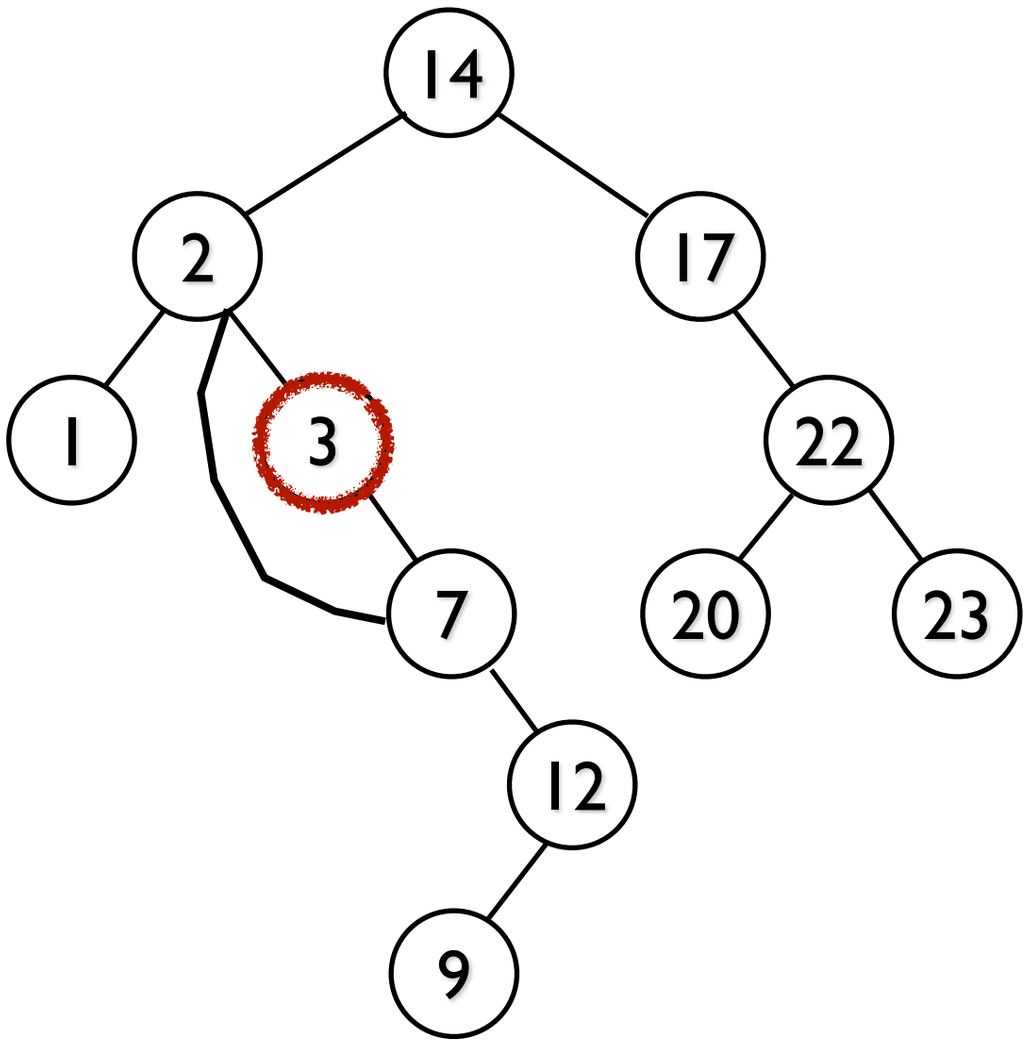
Target: 3



Replace node with the root of its subtree.

Case 2: One Subtree

Target: 3



Replace node with the root of its subtree.

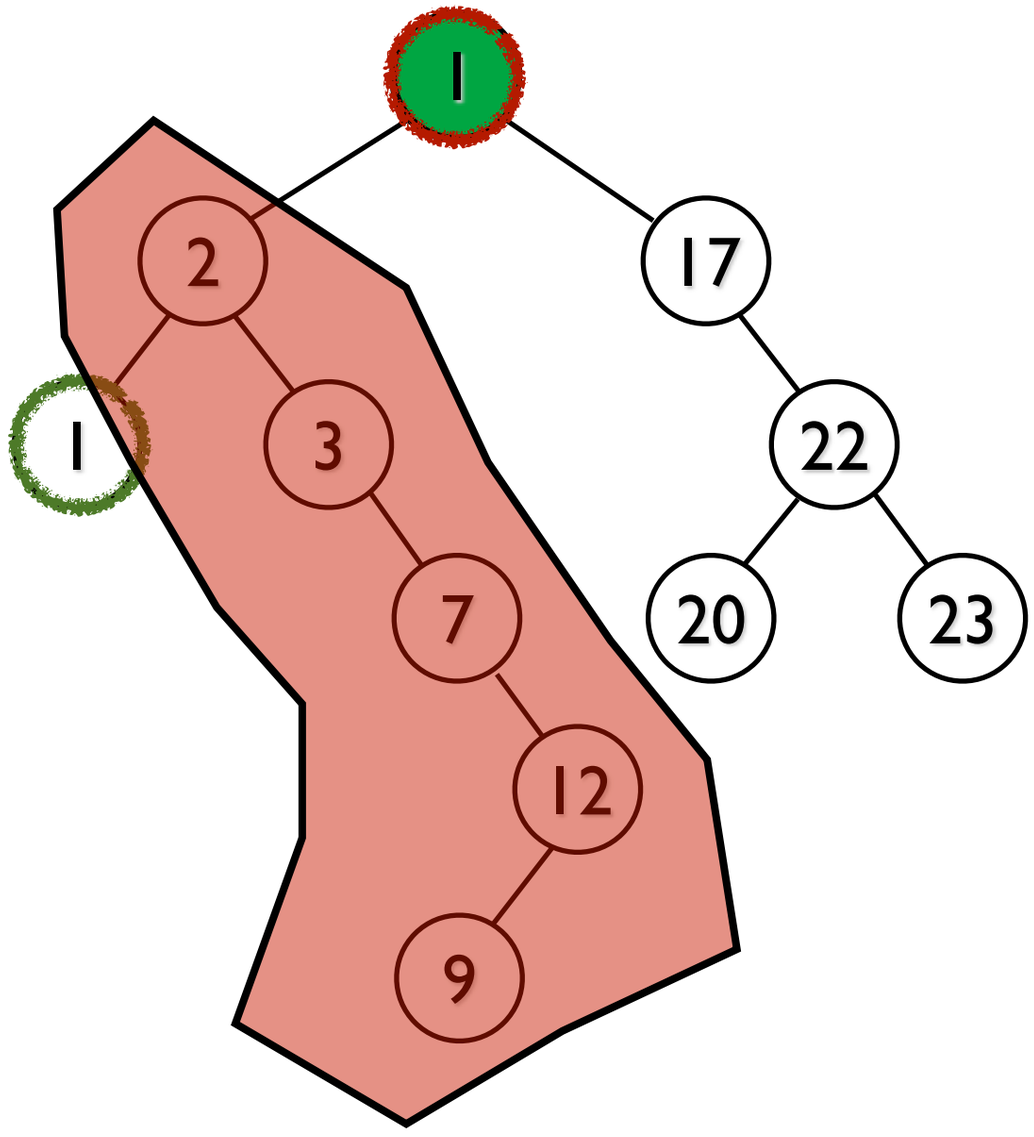
Case 3: Two Subtrees

Target: 14

We should swap the node's value with another node in its subtree that would be easier to delete.

The node we choose should be such that we have to do the least amount of work to restore BST properties.

E.g. If we swap with 1, we have five "problem nodes" to relocate or reshuffle.



Case 3: Two Subtrees

Target: 14

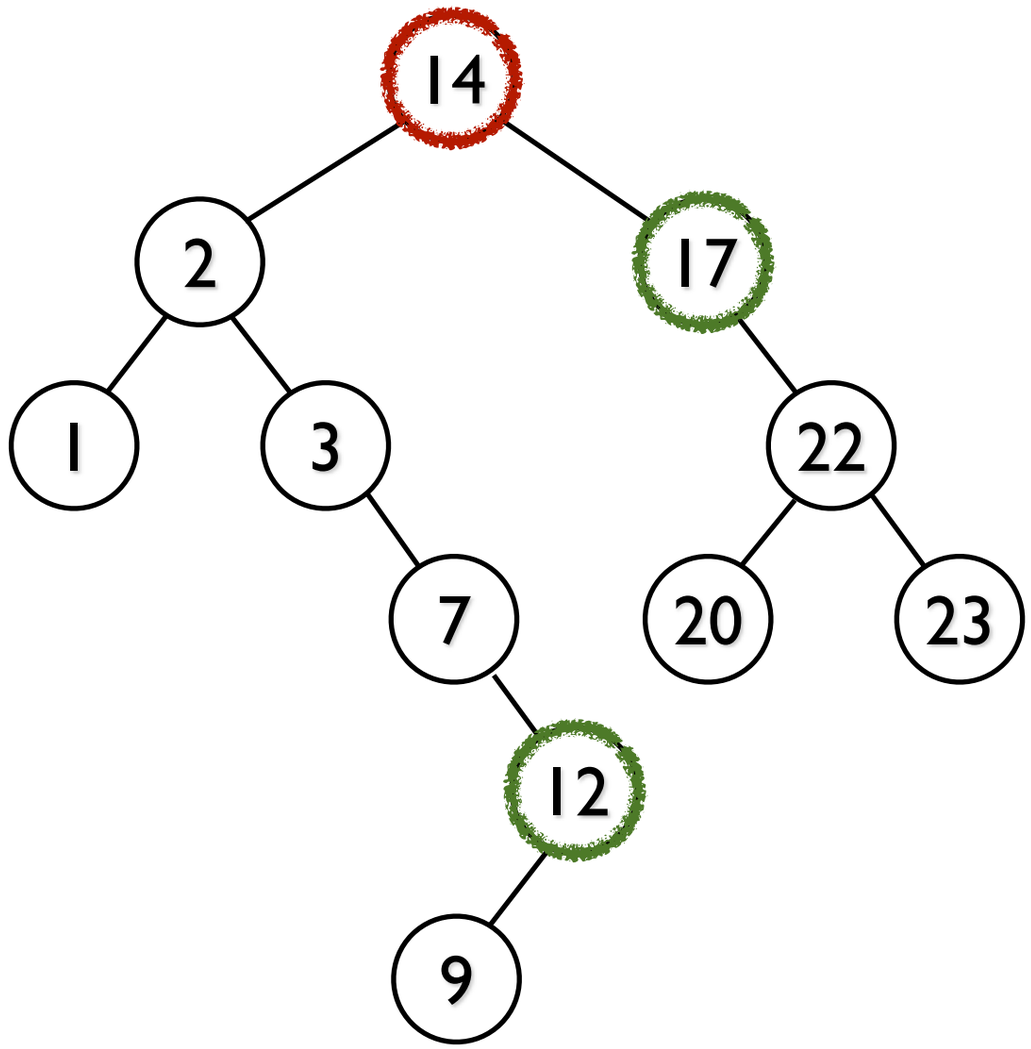
We should swap the node's value with another node in its subtree.

What's the best node to do this with?

In the left subtree,
we should pick the largest node.

In the right subtree,
we should pick the smallest node.

These are called the in-order predecessor and the in-order successor.



Question: at most how many subtrees do either of these nodes have? Why?

Case 3: Two Subtrees

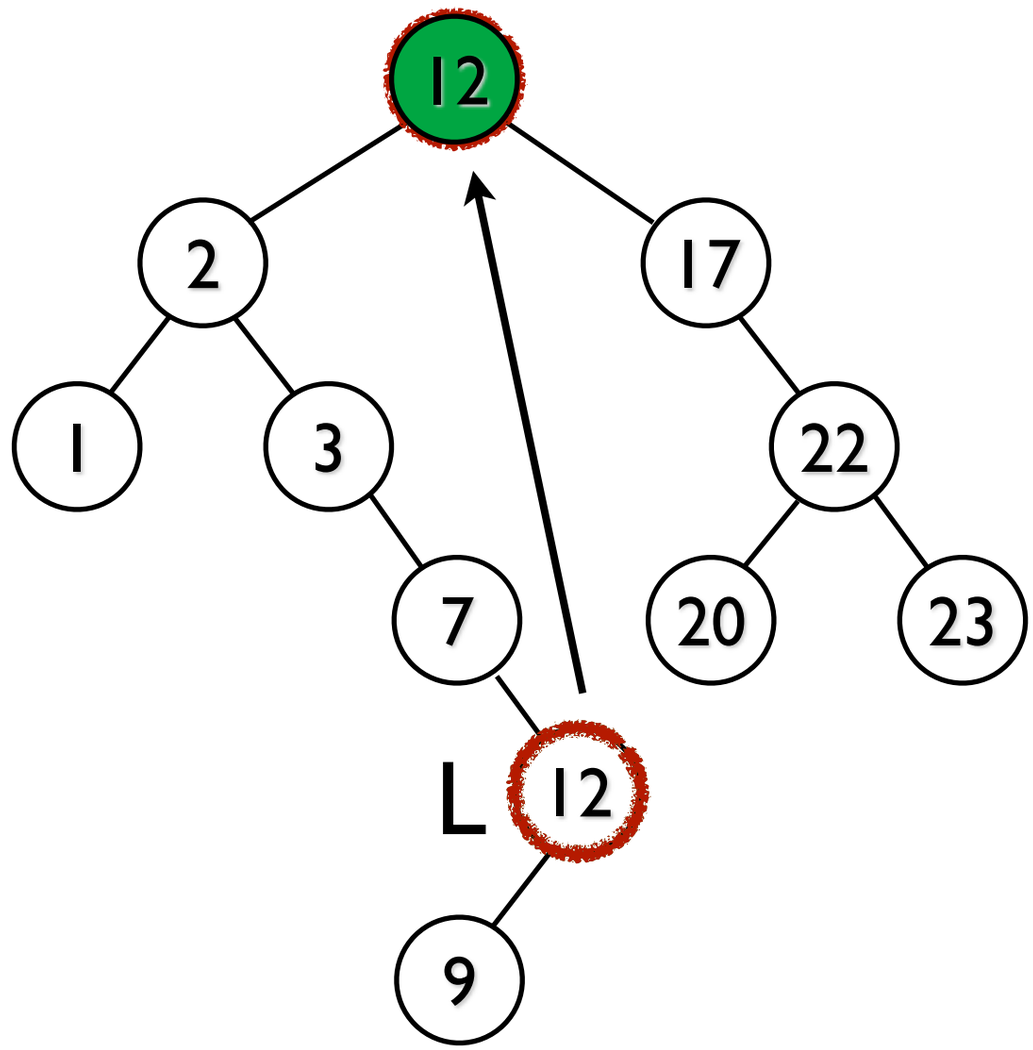
Target: 14

Case 3.1. In-order Predecessor

Find L, the largest node in the left subtree.

Copy L's value into the node to delete.

Delete L (reduced to case 1 or 2)



Case 3: Two Subtrees

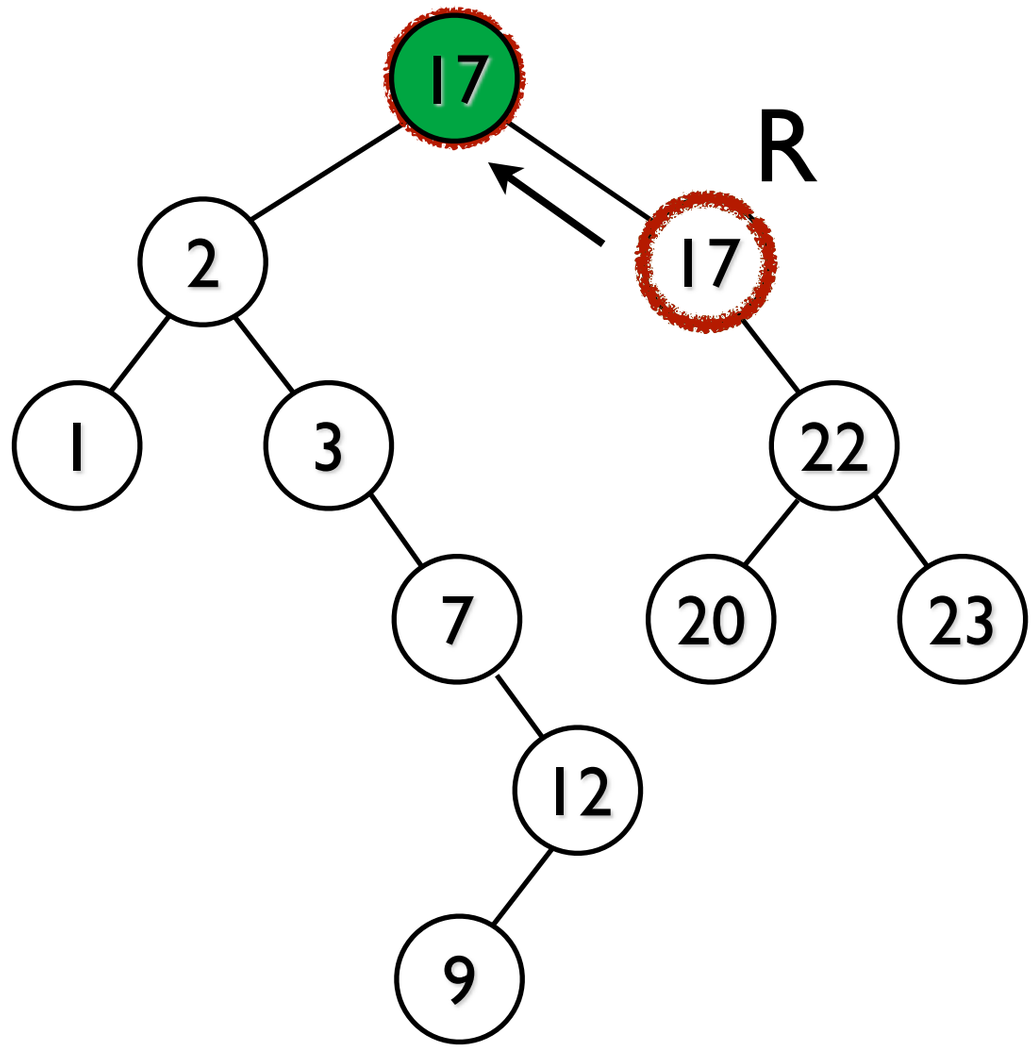
Target: 14

Case 3.2. In-order Successor

Find R, the smallest node in the right subtree.

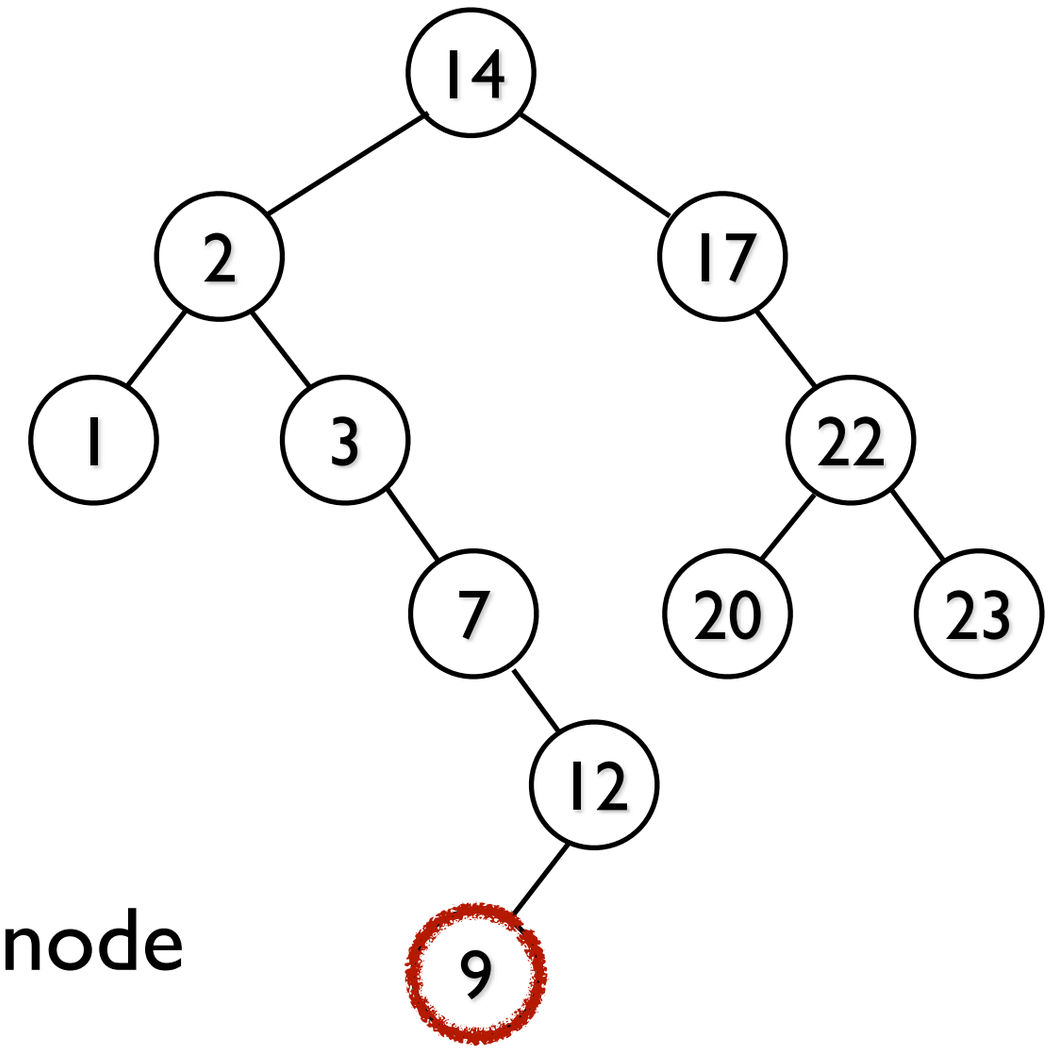
Copy R's value into the node to delete.

Delete R (reduced to case 1 or 2)



Updating BST Topology

Target: 9



Simple: delete the node

But really, tell the node's parent that its left child should be `None`.

How do we actually do that in code, given that we have no `parent` links?

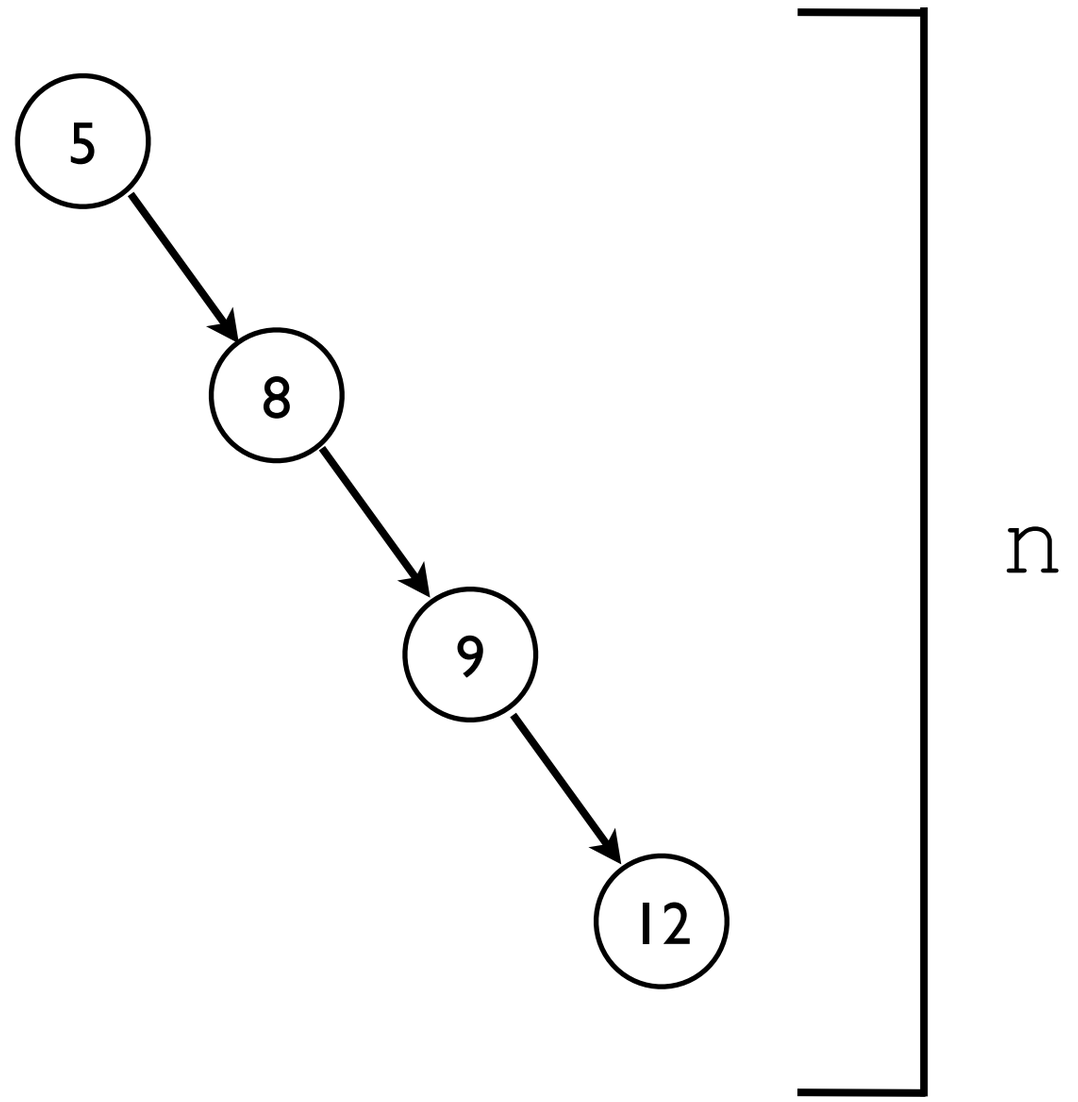
Optimal BST Height

Worst Binary Search Tree

The worst binary search tree is the one where no nodes can be eliminated when choosing a path.

This tree is created by inserting items into a BST in ascending (or descending) order.

For n items, this tree has a height of n.



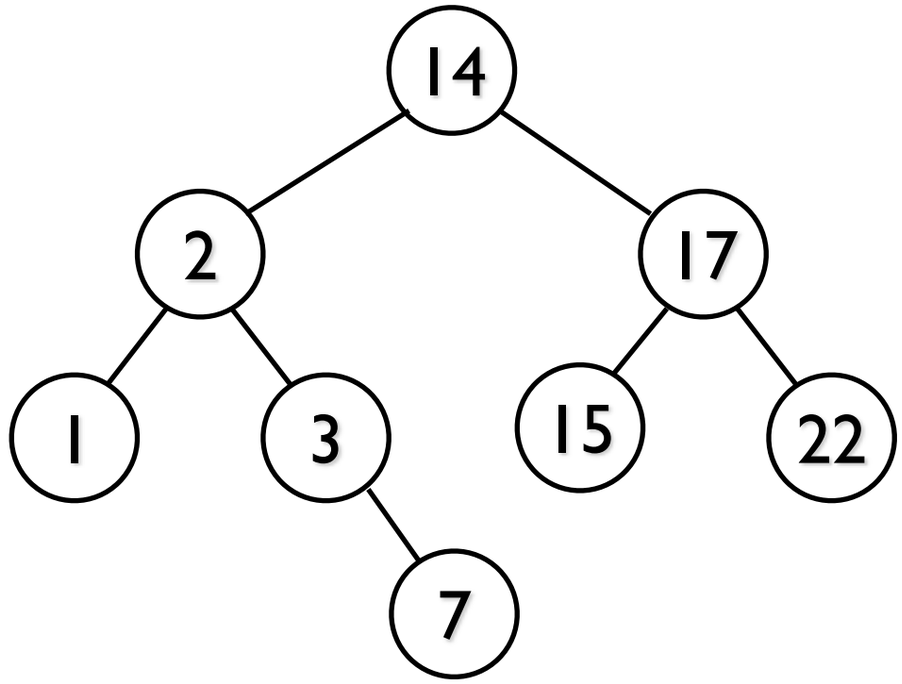
Optimal Binary Search Tree

The most optimal BST is one where as close to half of all possibilities are eliminated at every fork.

Typically, this means that every node except for the leaves has 2 children.

Such trees are called **full**.

A full tree with all of its leaf nodes on the same level is called a **perfect** tree.



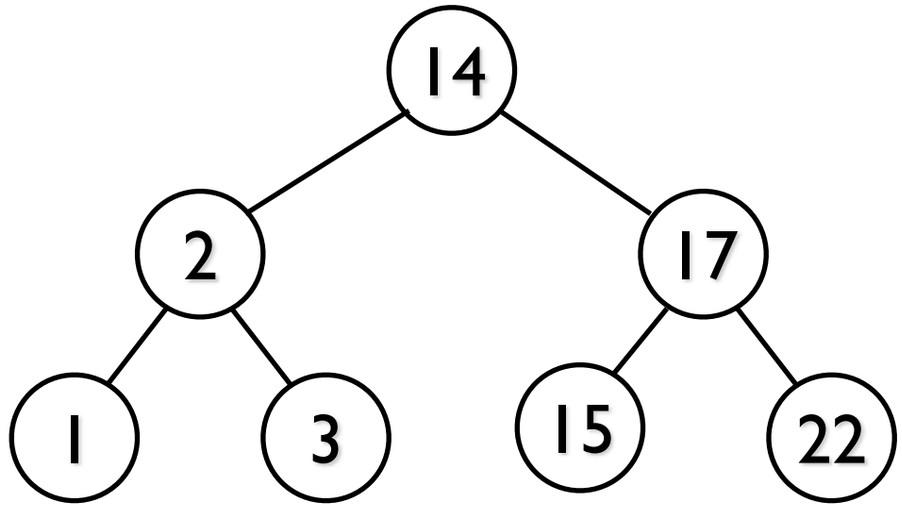
Optimal Binary Search Tree

In a perfect tree, the top level has 1 node (or 2^0 nodes).

The next level has 2^1 nodes.

The next level has 2^2 nodes, and so on.

The last level has 2^{h-1} nodes where **h** is the tree's height.



Which operations care about the height of a tree?

The total number of nodes is:

$$2^0 + 2^1 + 2^2 + \ldots + 2^{h-1} = 2^h - 1 = n$$

So, in a perfect tree: $h = \log_2(n + 1)$