# CSCI 2270

## Data Structures & Algorithms

Gabe Johnson

Lecture 6 Jan 28, 2013

## Recursion
## Binary Search Tree Operations

# Lecture Goals

1. Announcements
2. Recursion
3. Binary Search Trees
4. Traversal: pre-, in-, post-order
5. BST Operations

# Upcoming Homework Assignment

HW #2 **Due: Friday, Feb 1**

## Binary Search Trees

HW2 is out! This week I have provided a comprehensive driver. Don't get too used to this—you'll need to write more and more of this as time goes on. The main tricky thing this week is *recursion*.

# Announcements

1. **Email Policy:** I have one! Read the document at the top directory in the GitHub repository. Basically:

    - characterize your problem
    - try to debug it (print statements, debugger)
    - email the TA first if you think they can help
    - if you have code, mail a link to a Github Gist
    - use coherent English

# Yet More Announcements

2. **Linked List HW Results:** 1

   128 people got 15/15!

   Huzzah!

# Recursion

Recursion, *n*: See *recursion.*

# Recursion

# Recursion

A recursive function is one that calls itself. Different invocations of the function have their own local variable scope, so this is safe. As an example, consider summing all the numbers below (and including) some input number:

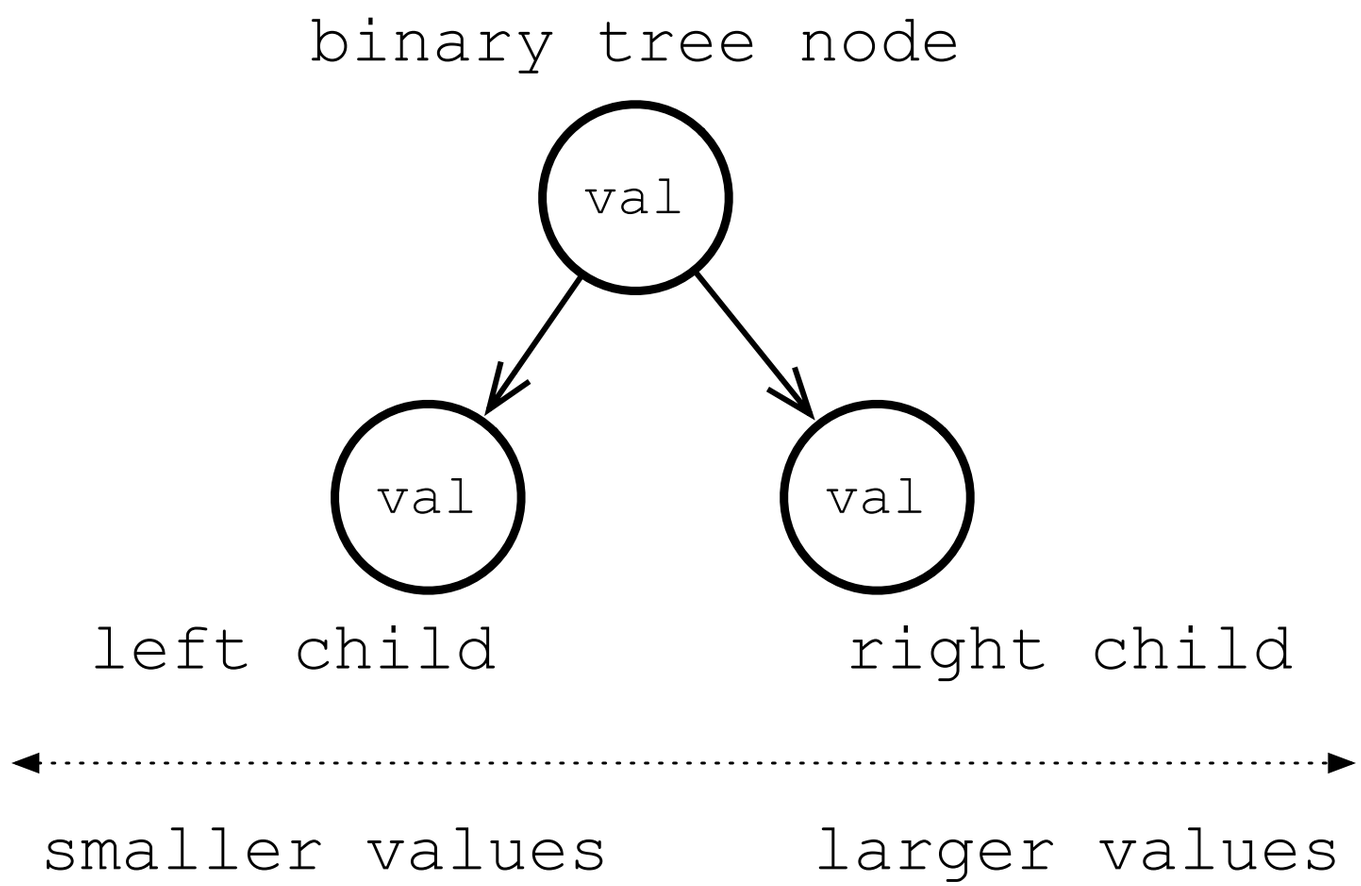
```
def add_it_up(num):
    if (num > 0):
        return num + add_it_up(num - 1)
    else:
        return 0
```

calling **add_it_up(6)** gives 21:

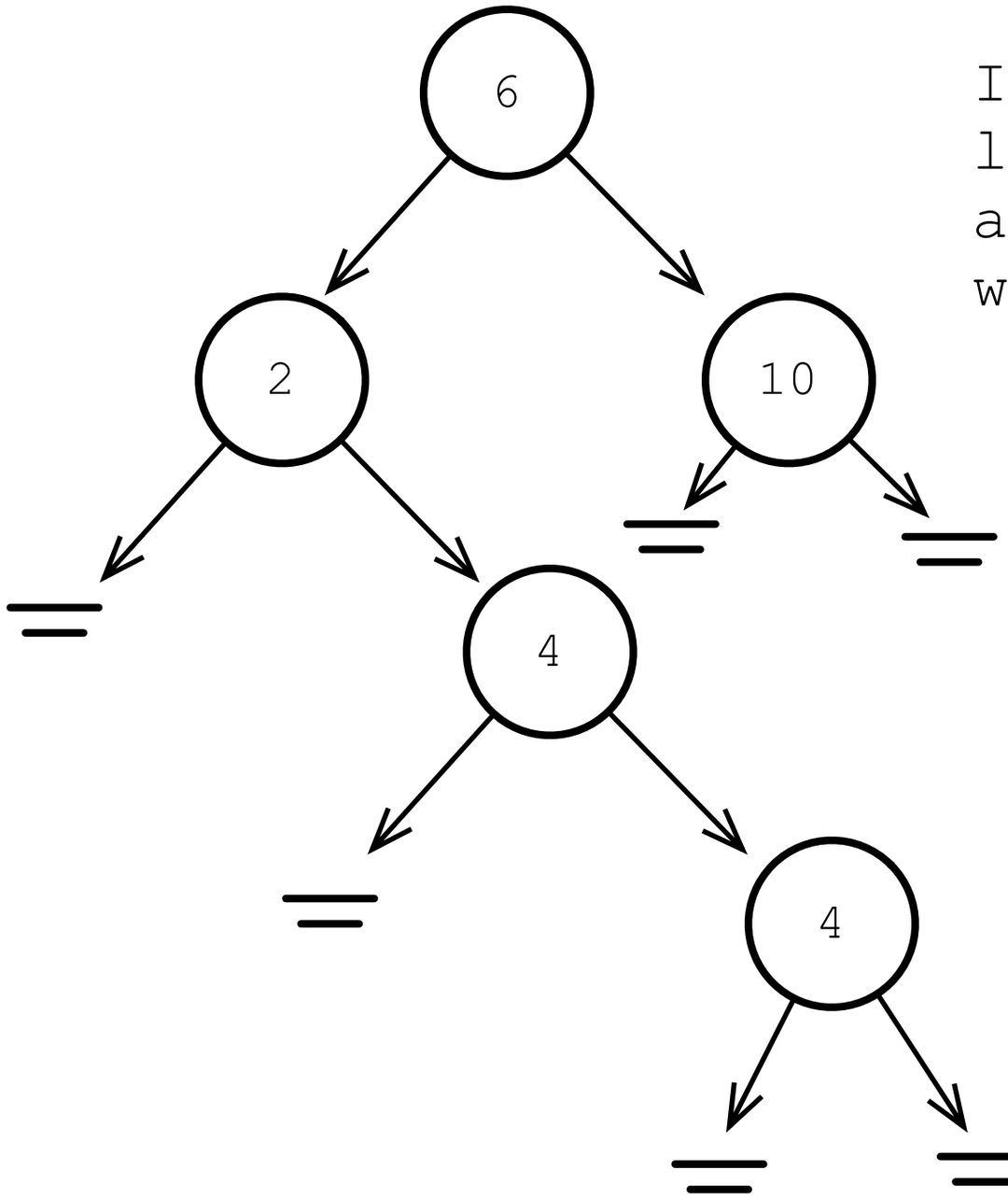
```
6 + 5 + 4 + 3 + 2 + 1 + 0
```

# Binary (Search) Tree Nodes

Binary trees are composed of nodes that have up to two children. Binary *search* trees are used to find values quickly, and are kept in a certain order (usually nondecreasing order).

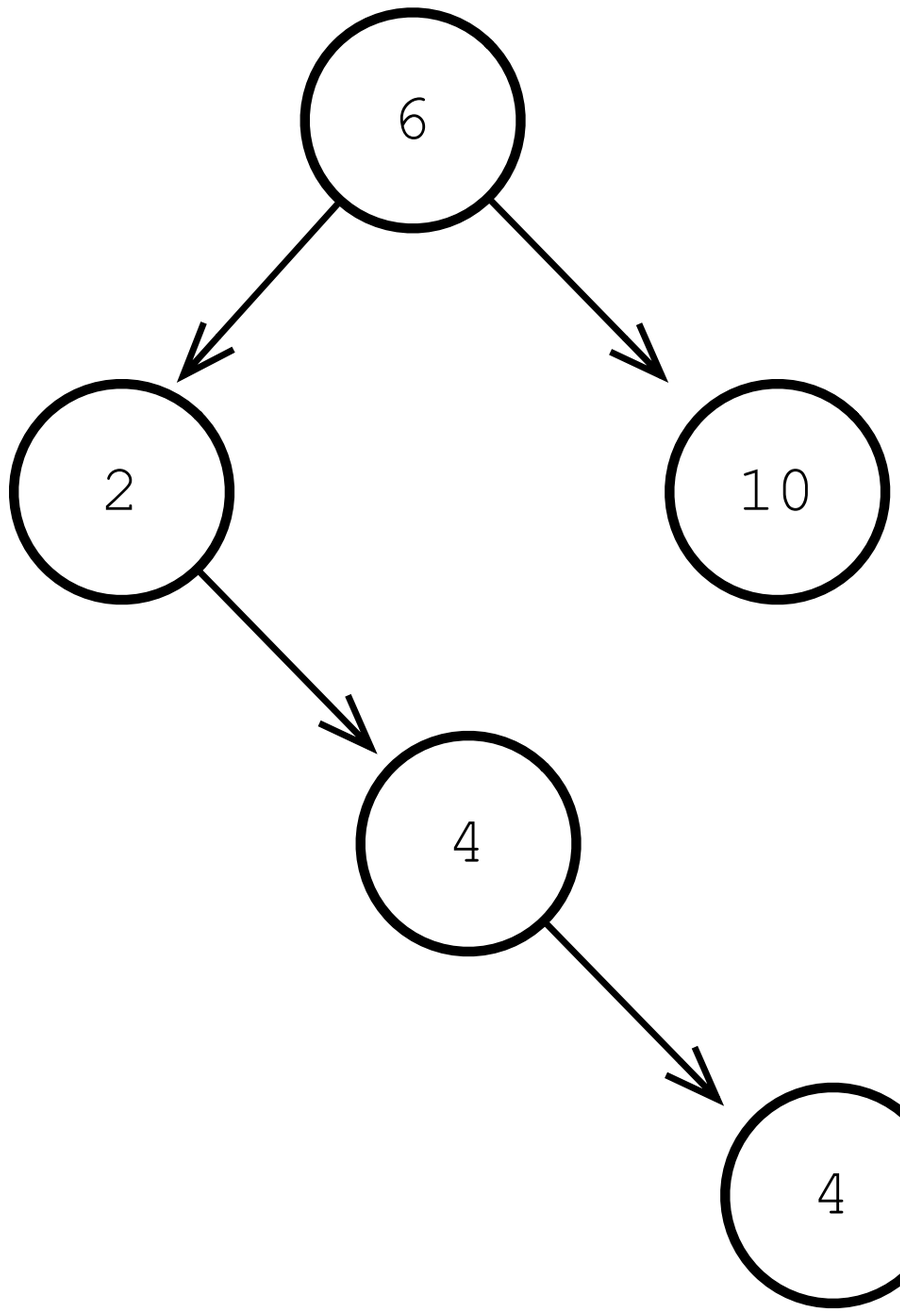


# How To Draw I



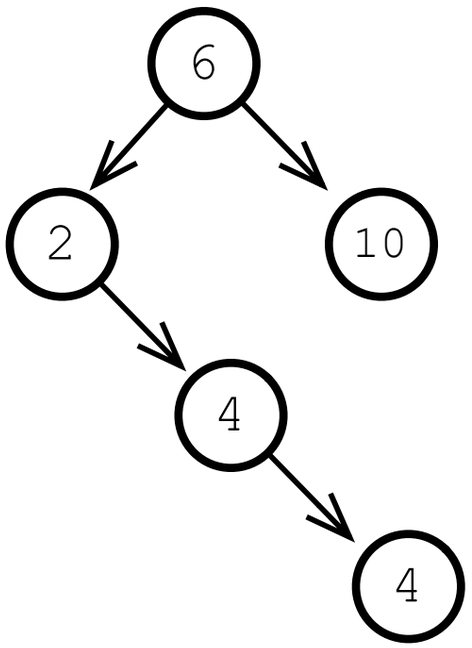

If we drew the null links between all parent and child nodes, it would be a little messy.

# How To Draw II




Binary trees are almost always drawn without the null links. It is assumed, for example, that the '2' node has a null left child, and that the lower '4' node has two null children.

# Traversing




Nodes have the following order:

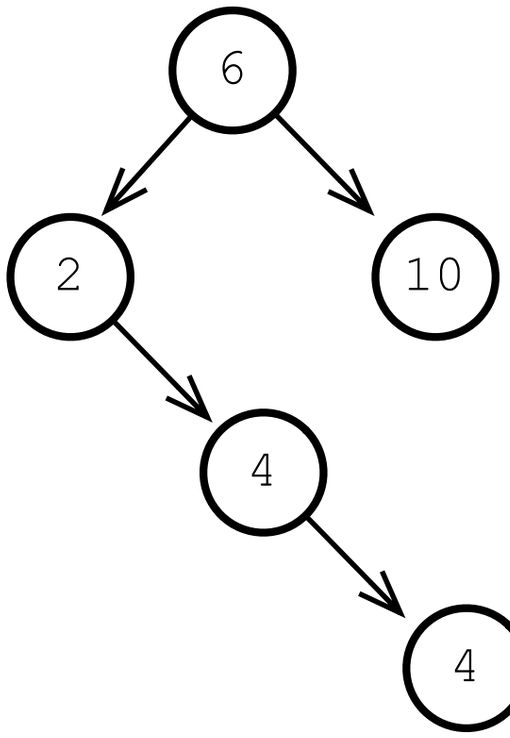
left children are all **less** than the parent node.

right children are all **greater than or equal to** the parent node.

Because nodes have this structure and ordering rule, we can confidently apply **traversal** algorithms to them. We can traverse in three ways, but for our purposes, only one of them is useful. They are:

pre-order walk: visit, go left, go right
in-order walk: go left, visit, go right
post-order walk: go left, go right, visit

# Walk This Way (in-order)




In-Order is the useful one for the BST assignment, because it gives us the sort order we want. Here's the code outline:

```
def in_order(node):
    if (node is not None):
        in_order(node.left)
        visit(node)
        in_order(node.right)
```

Notes:
Sanity checking (is it null?) happens inside the function.

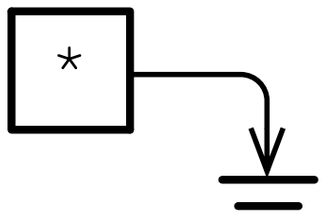
It is recursive because it calls itself.

The 'visit' line can be replaced with whatever you need, like printing out the node, or adding it to a list of values that are in the proper sort order.

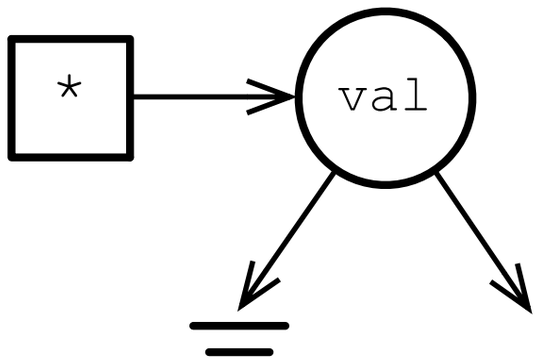
# Insert

Insert has three possible conditions:

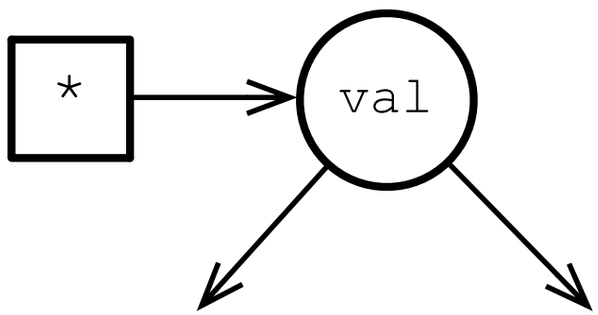
The tree is empty.



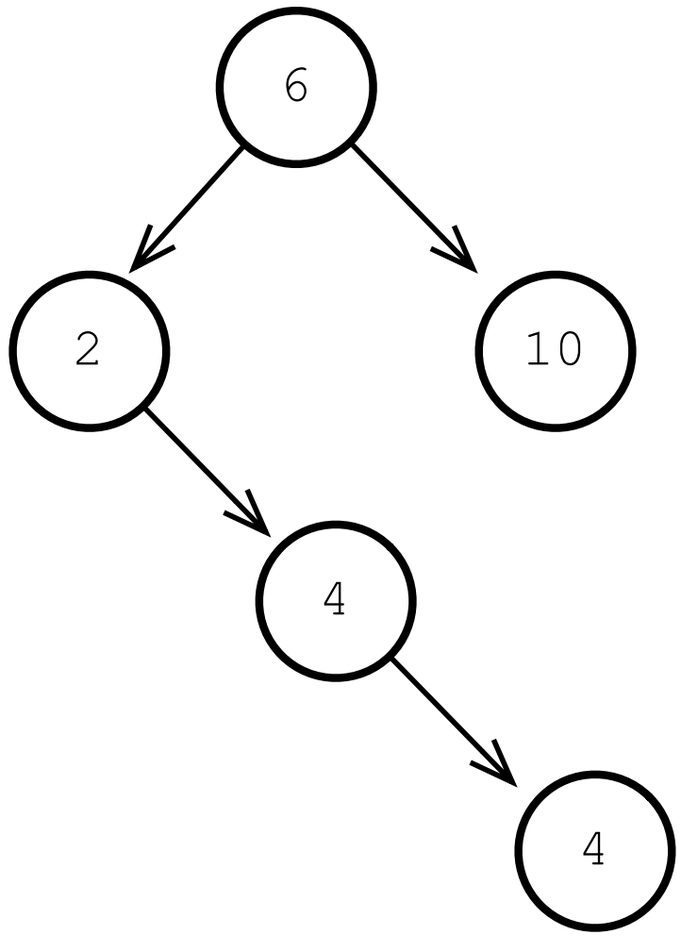

The node you are examining has an empty slot where you want to insert the node (shown as the null child).




The node you are examining has two valid child nodes, so you the new node has to be inserted somewhere downstream.



# Find I




To see if the tree contains a node with a value, find it using a recursive pre-order search:

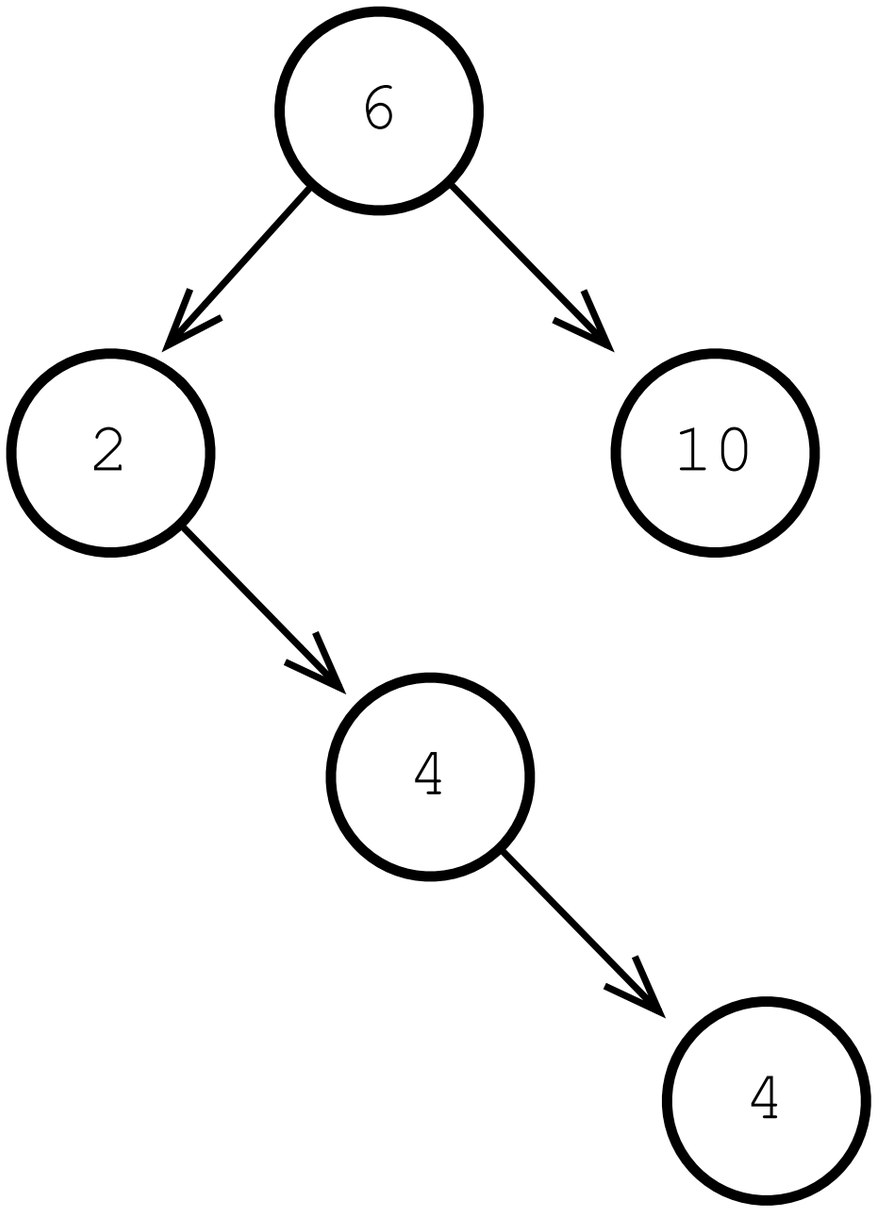
Is this node null? If so, return false.

Does this node have the right value? If so, return true!

If this node is larger than the target node, recurse left and return whatever it gives you. If not, recurse right.

If this node has no children, return false.

# Find II




Example: say we are looking for '4'.

Is 6 it? No. Recurse left, because 4 < 6!

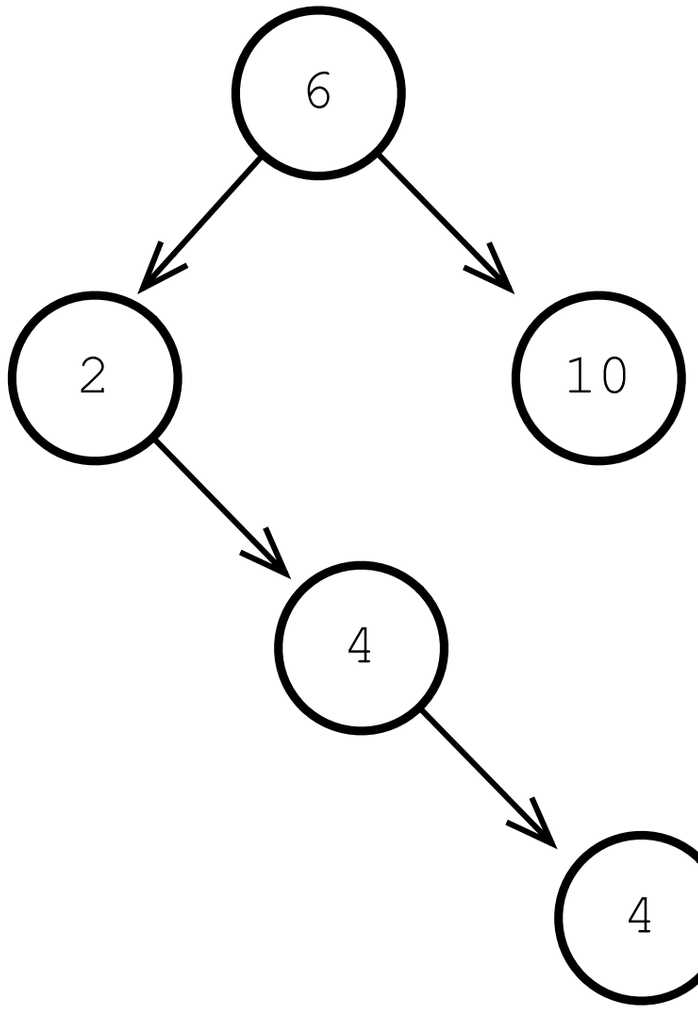
Is 2 it? No. Recurse right, because 4 >= 2!

Is it 4? Yep. Return true.

Return true.

Return true.

# Size



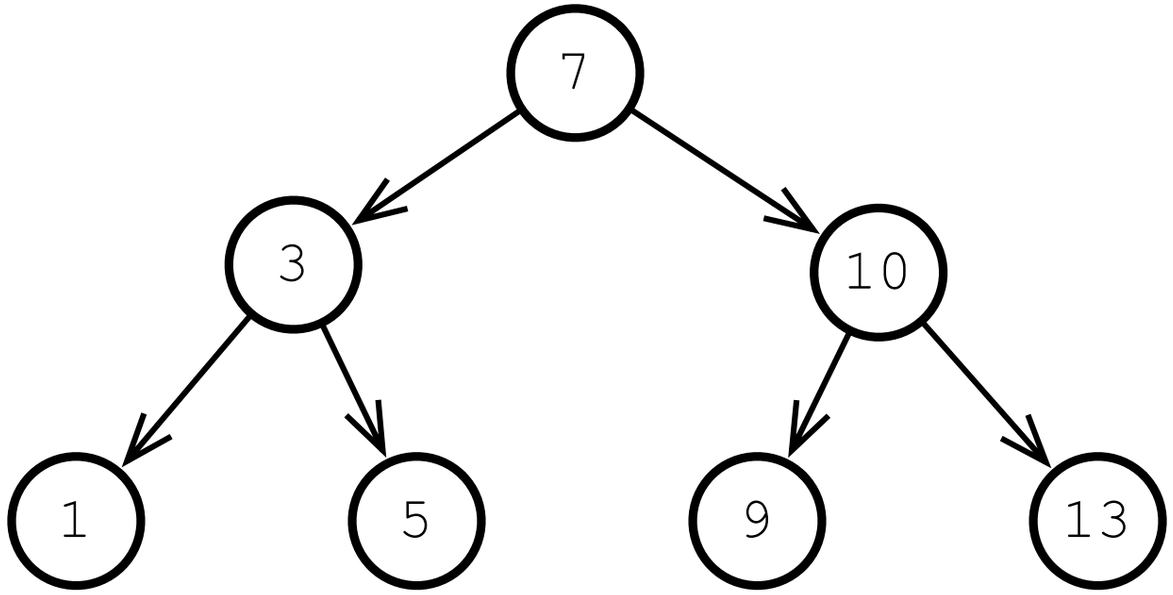

To find the **size** of the tree, simply perform a walk (of any kind) and increment a counter when you visit a node.

Only tricks here are to be careful about making sure you're not off by one, and having a way to pass along your count using a recursive function.

e.g.,

```
return 1 + left_size() + right_size()
```

# Remove I



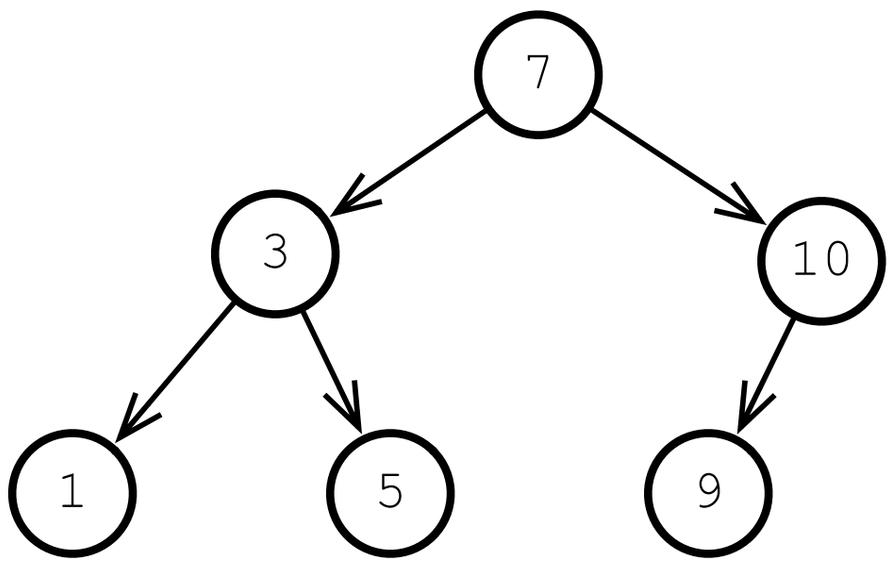

Removing a node is not completely obvious.

Whatever we do, we have to maintain three rules:

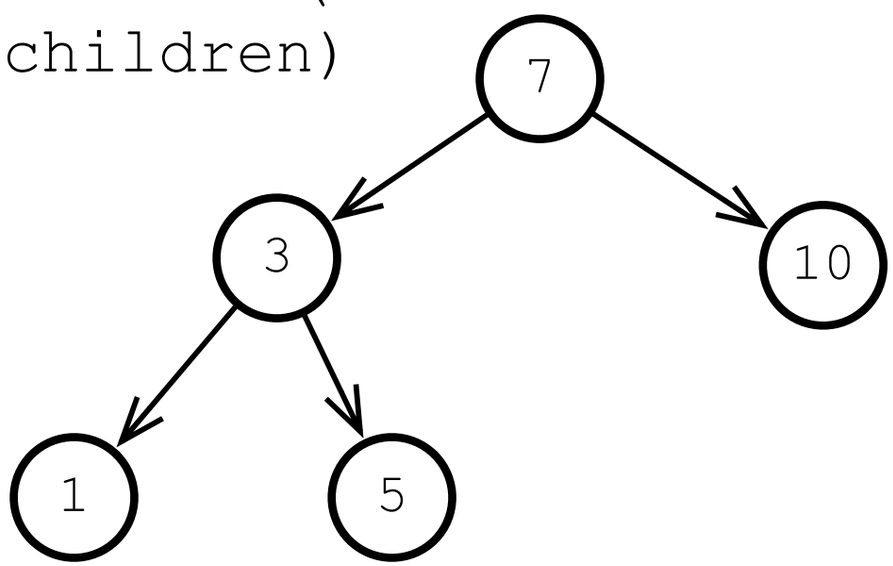
1. On exit, the doomed node is not in the tree.
2. On exit, it is still one tree—meaning we didn't orphan anything.
3. On exit, the sort order is still intact.

# Remove II

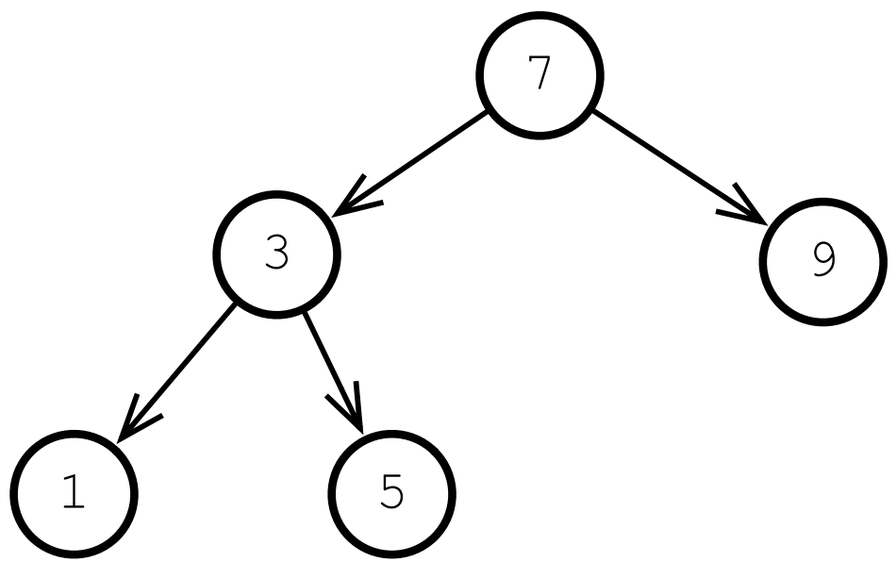
initial tree




delete 9 (no children)




or delete 10 (one child)

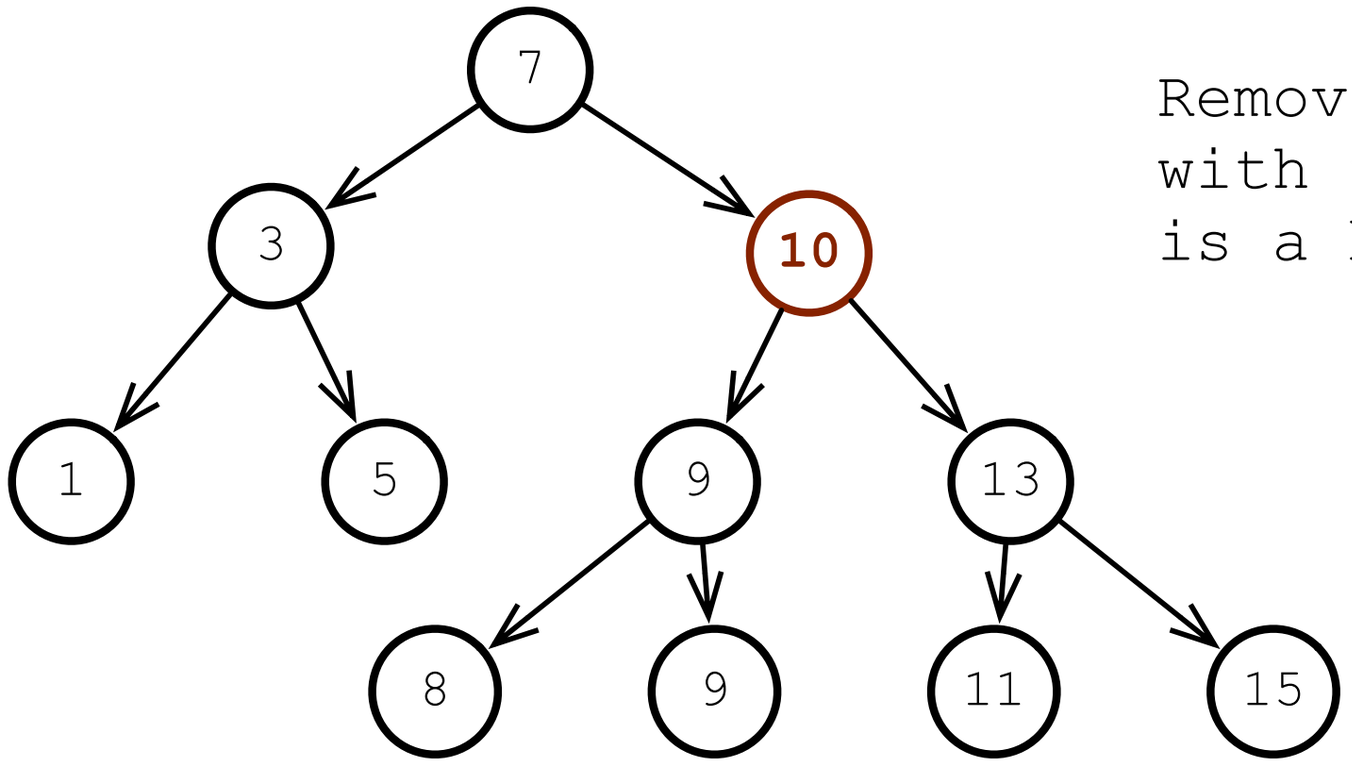



Removed nodes could have 0, 1, or 2 children.

Zero: set the parent pointer to null.

One: replace the doomed node with its child.

# Remove III

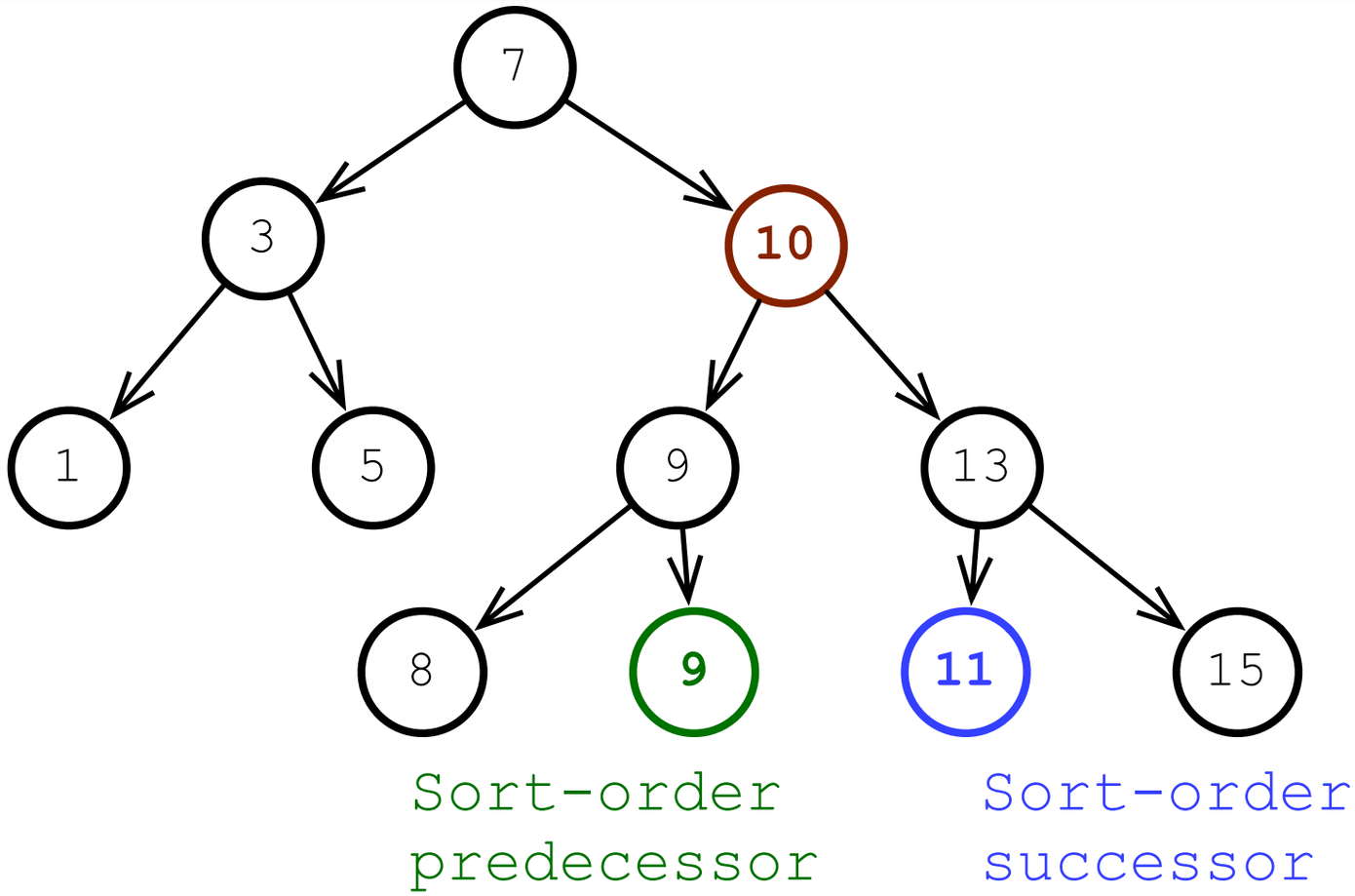



Removing nodes with two children is a bit tougher.

The committee has determined that node 10 must go. There are a number of ways to perform this operation, and we won't require that you use any specific one. All that matters is that the tree obey the rules enumerated earlier.

Study this tree and imagine what it could look like with the 10 node removed.

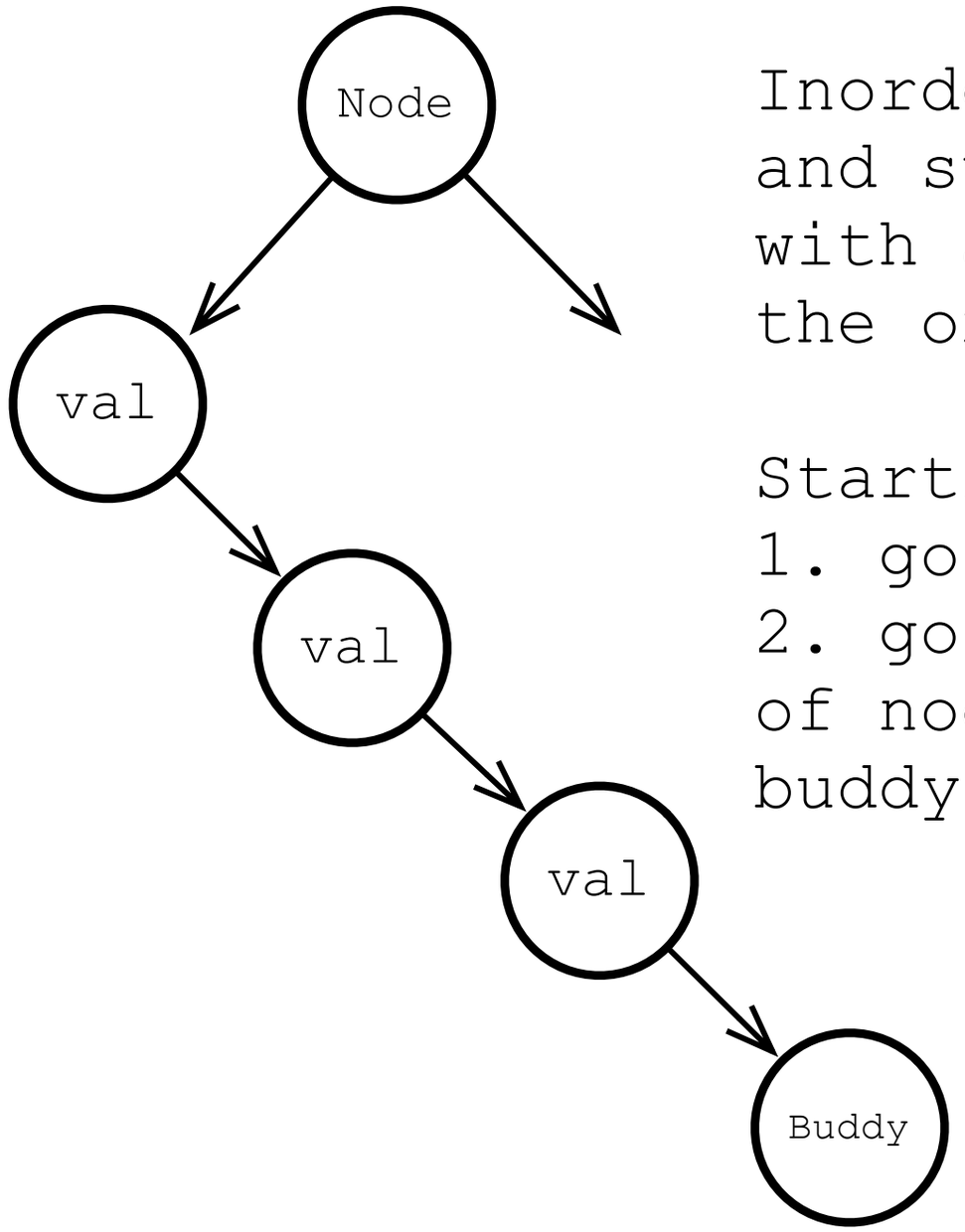
# Remove IV




One way: if the doomed node has two children,
find its in-order predecessor ('buddy'),
and *replace* the doomed value with buddy's value,
then *delete buddy*.

Here, 'buddy' is either the green or blue one.

# Inorder Neighbors




Inorder neighbors (predecessors and successors) can be found with a short algorithm. Here's the one for predecessors:

Starting from Node,
1. go left one time
2. go right until you run out of nodes. the last one is your buddy.

The procedure for finding the in-order successor is similar; just switch left and right.