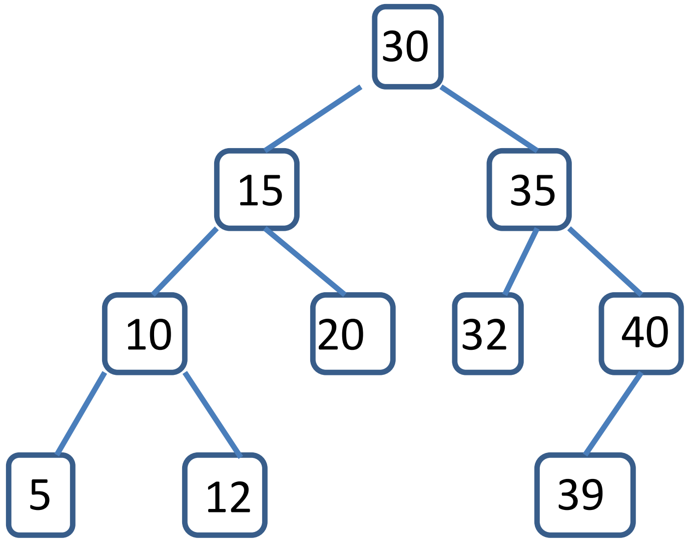
Binary Search Trees Revisited

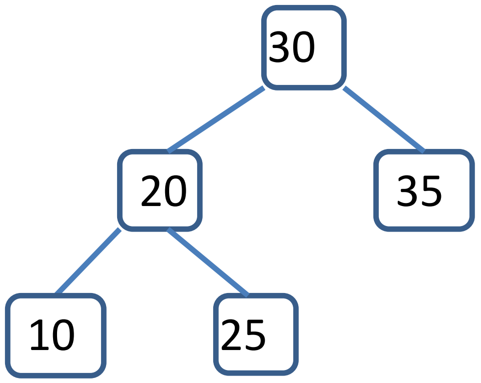
See Section 19.1 of the text, p 687-696.

We talked about Binary Search trees on Friday April 1, the day before Spring Break. This is a quick review of that. If you haven't read the April 1 notes you might want to.

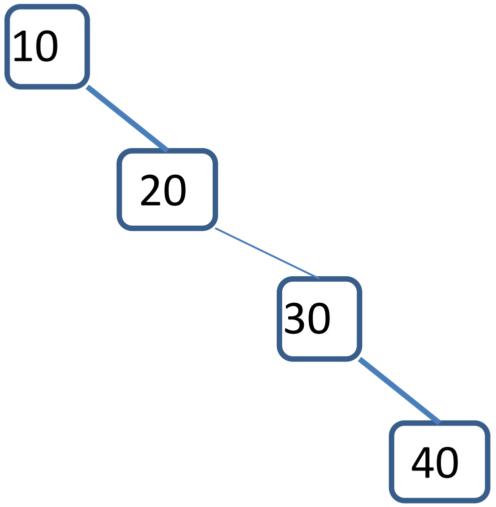
The *Binary Search Tree* property applies to trees that have values (or key-value pairs) at every node. The property says that for each node in the tree all of the nodes in its left subtree have values less than the node's value and all of the values in its right subtree are greater than the node's value. For example:



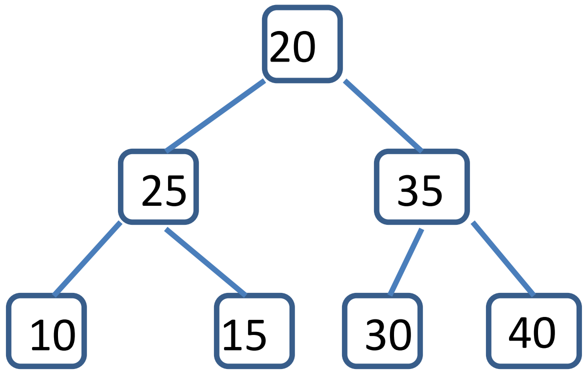
Here are two BSTs:



and



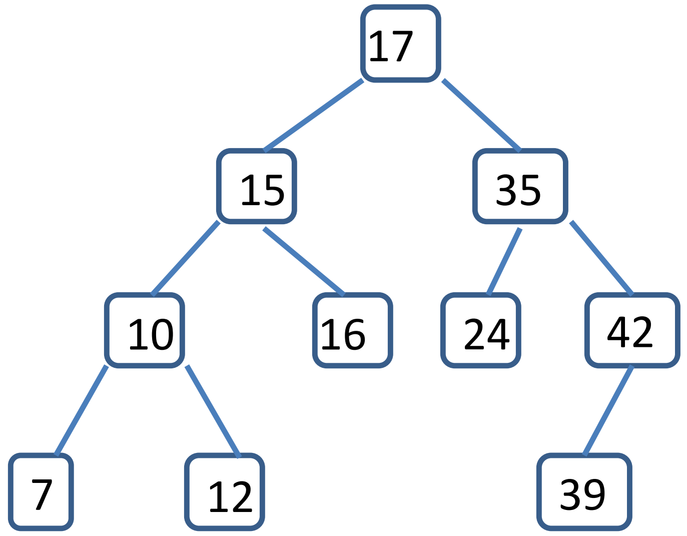
This is NOT a BST:



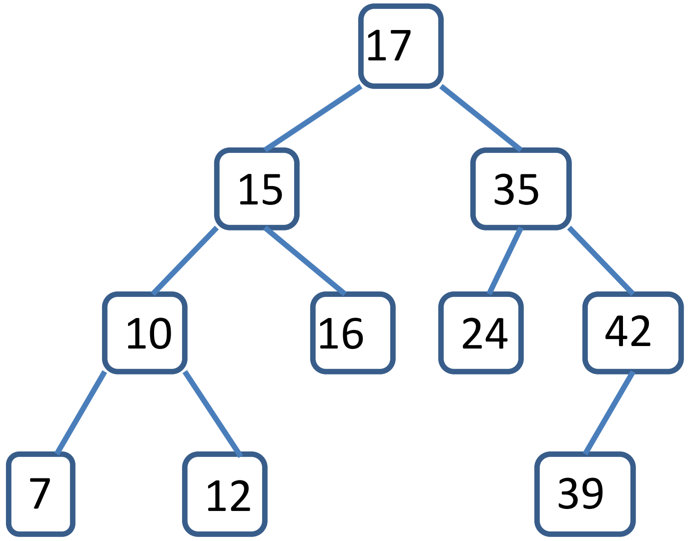
The left child of 20 has a larger value, 25

We want to implement Binary Search Trees with the following methods:

- find(v) finds the node with value v
- insert(v) adds a node with value v
- findMin() and findMax() return the tree nodes with the smallest and largest values
- removeMin() and removeMax() remove the smallest or largest values
- remove(v) removes the node with value v

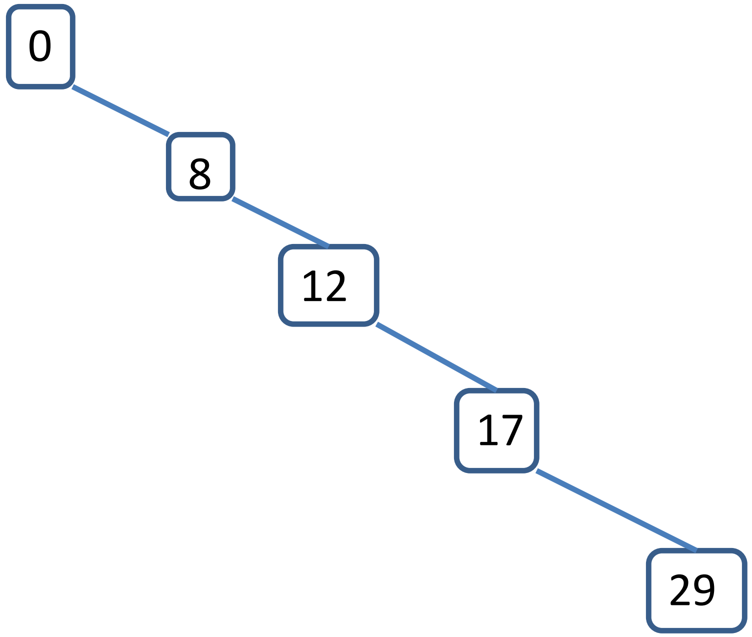


There is an easy algorithm for searching a BST to determine if it contains a node with value k: We start at the root. At each step if k is greater than the value of the node we move to the right child; if k is less we move to the left. This ends either when we find k or when we get to a null child.



To insert value k into a BST, we do a search and insert the new node when we come to a null child. For example if we want to add value 29 to our BST we notice that it is greater than 17 (the root), less than 35, and greater than 24. The node with value 24 does not have a right child, so we add 29 there:

If we are lucky Binary Search Trees are balanced (each node has 2 children) and each step of a search eliminates half of the nodes. However, we might not be so lucky. Consider the BST we would get if we start with root value 0 and then add, in order, the values 8, 12, 17 and 29:



In this case searching the BST degenerates into a linear search

There are two ways to handle the insert method. For an iterative method you can loop down from the root, following the BST algorithm. If you are ready to move to the left and there is no left child, put the new node there. If you are ready to move to the right and there is no right child, put the node there.

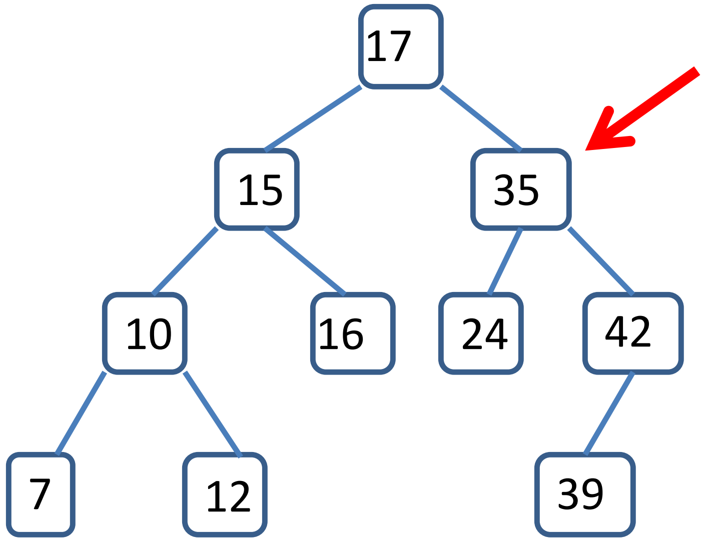
Alternatively you can recurse to do an insert. To insert the item in the tree rooted at node t, look to see if it should go in the left or right subtree. If the left subtree, replace t.left by the result of inserting the new item in t.left. In other words

```
        t.left = insert(x, t.left);
```

The same applies to the right. When you get to a null pointer, return a new node containing item x.

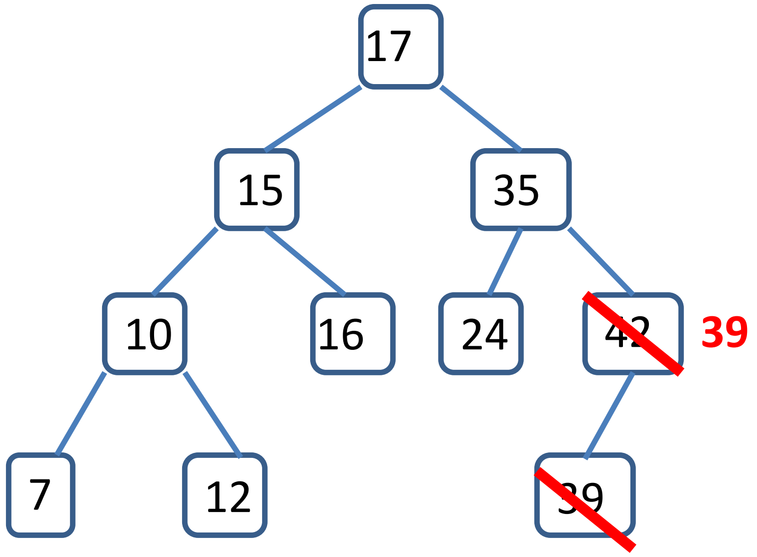
The only BST method that is tricky to implement is remove.

Consider our example tree:

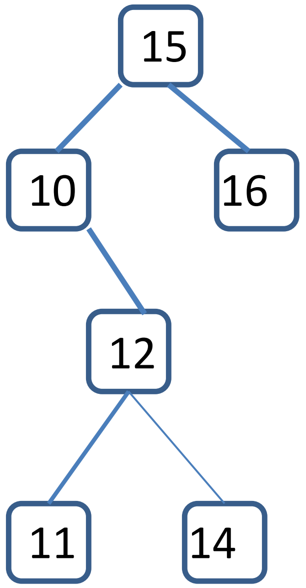


Suppose we want to remove the node with value 35. That is tricky. So we cheat and remove something that is easy.

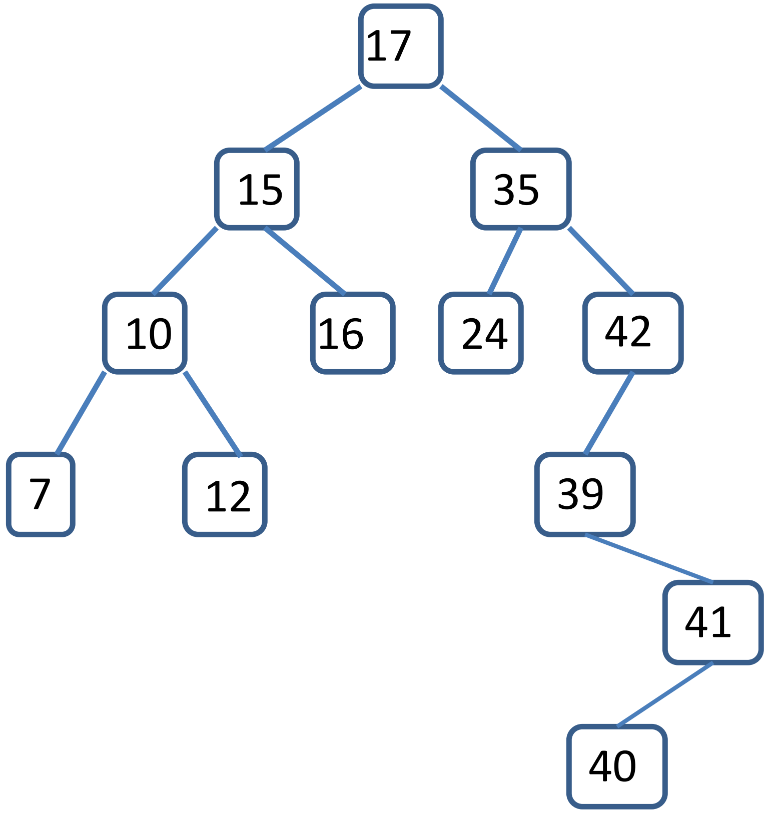
A node with at most one child is easy to remove: we just replace it by its child. For example, node 42 could be replaced by node 39 and this would still be a BST:



In this tree we could replace node 10 with its right subtree and the result would still be a BST.



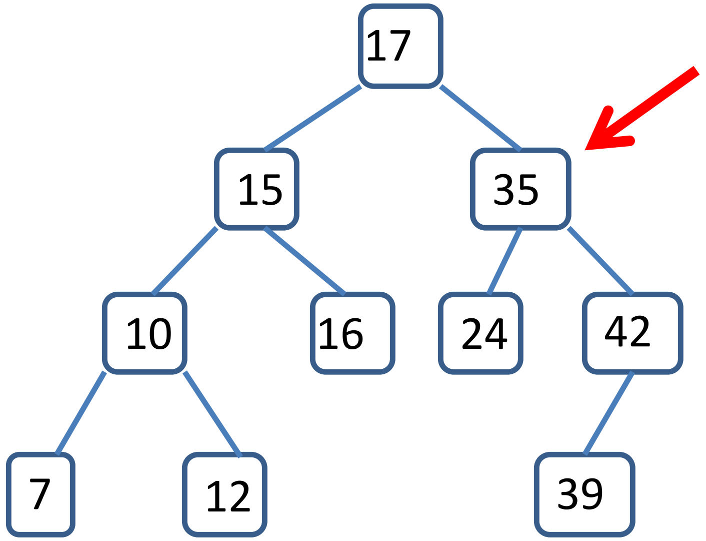
One way to get a node with at most one child is to find the minimum or maximum value beneath any node. We get to the minimum by following left pointers until there is no left child; the minimum node might have a right child but no left child.



One way to get a node with at most one child is to find the minimum or maximum value beneath any node. We get to the minimum by following left pointers until there is no left child; the minimum node might have a right child but no left child.

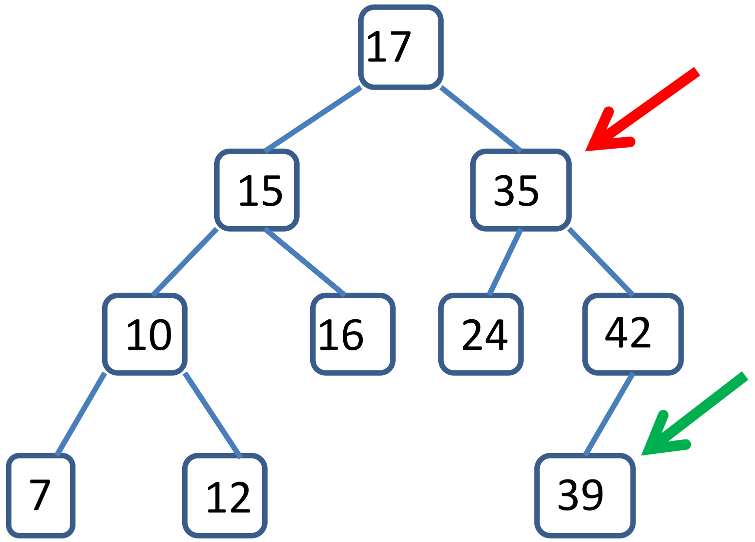
So let's go back to the problem of removing an interior node of a BST.

We want to remove node 35.

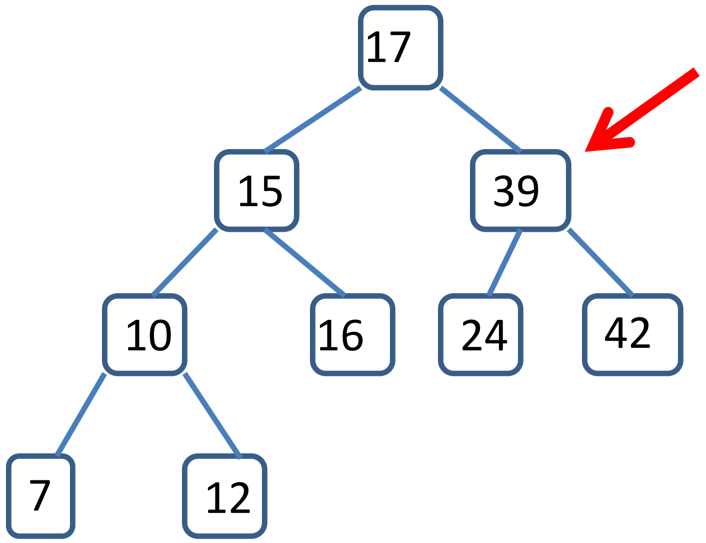


Since it has two children we go to its right child; all of the values in this subtree are greater than 35.

The minimum node in the right subtree of 35 is a node we know how to delete. The value of this node is 39.



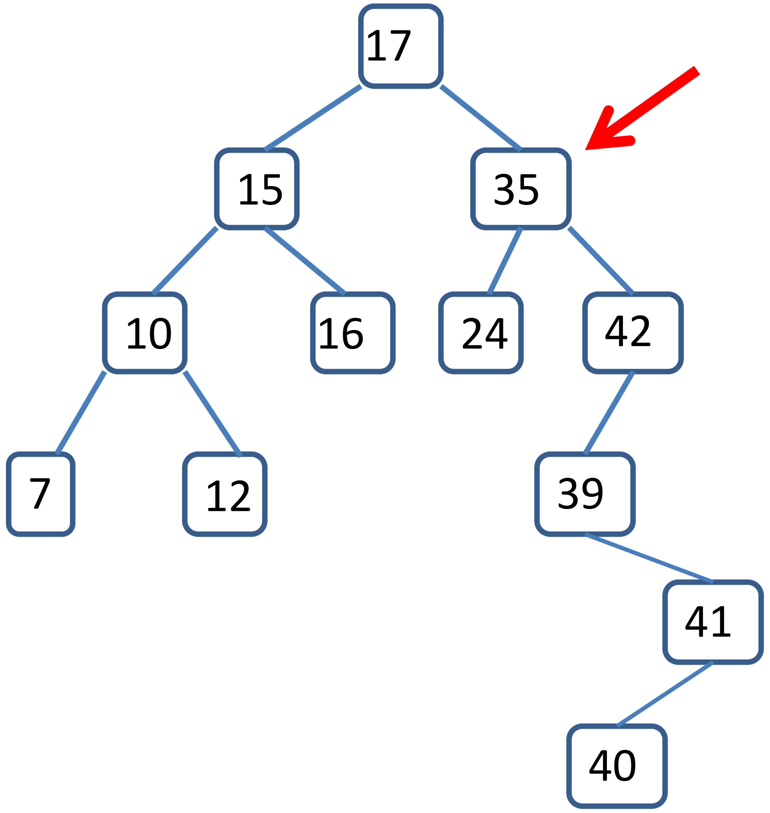
Note that 39 is less than every other node in the right subtree of 35, and greater than every node in the left subtree. We can switch it with 35 and delete the node that currently contains 39. The result is a BST that has all of the original values except 35:



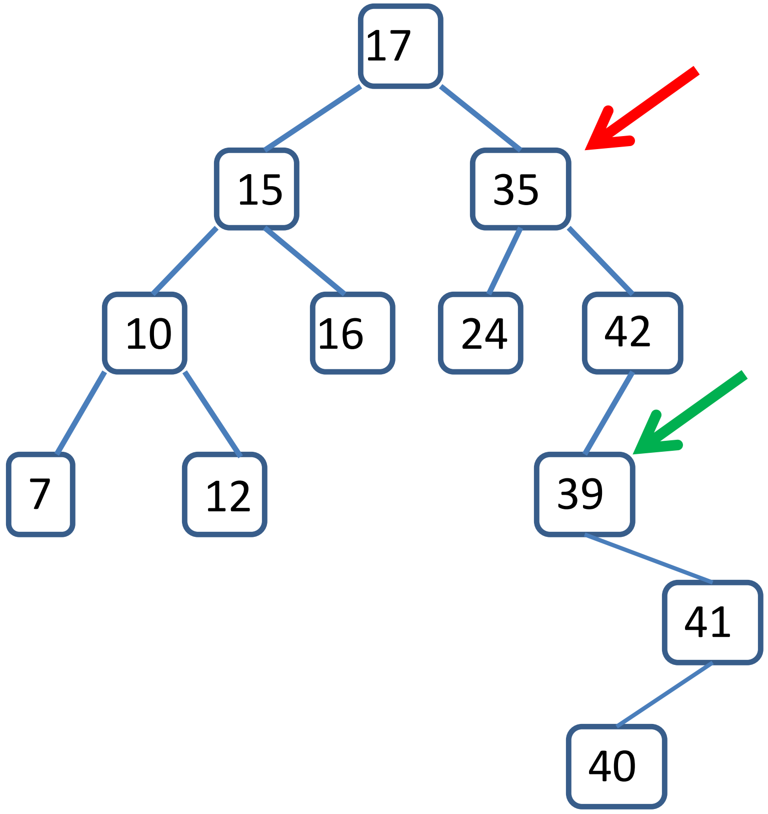
Isn't that clever!

To implement this we make a method that starts at a given node and returns the smallest data value below it, and another method that starts at a node and removes the node with the smallest value below it.

Here is another example with a slightly different tree. Again we want to remove 35.

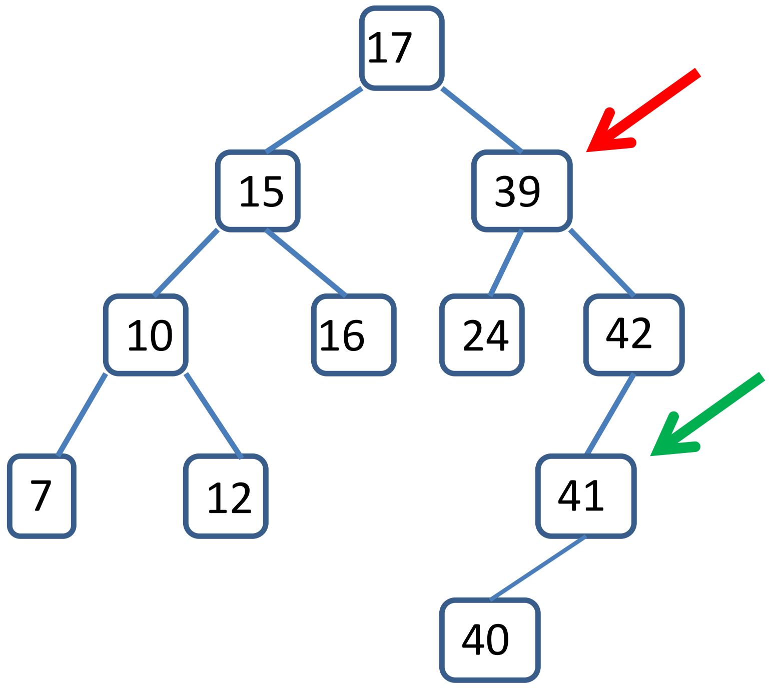


We go to the right from 35, then find the smallest node in 35's right subtree.



Again it is 39, but this time 39 isn't a leaf; it has a child.

The same algorithm applies. Since 39 has only one child, we remove the 39 node by replacing it with its child, and then replace the value 35 by 39:


17
15
39
10
16
24
42
7
12
41
40