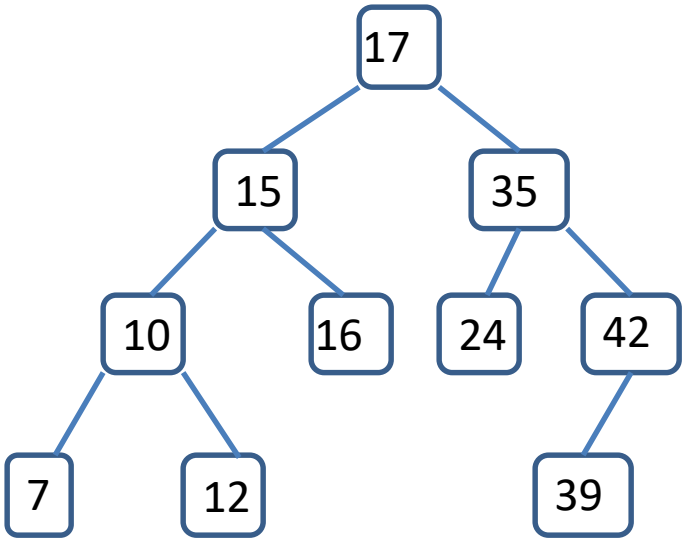
Binary Search Trees

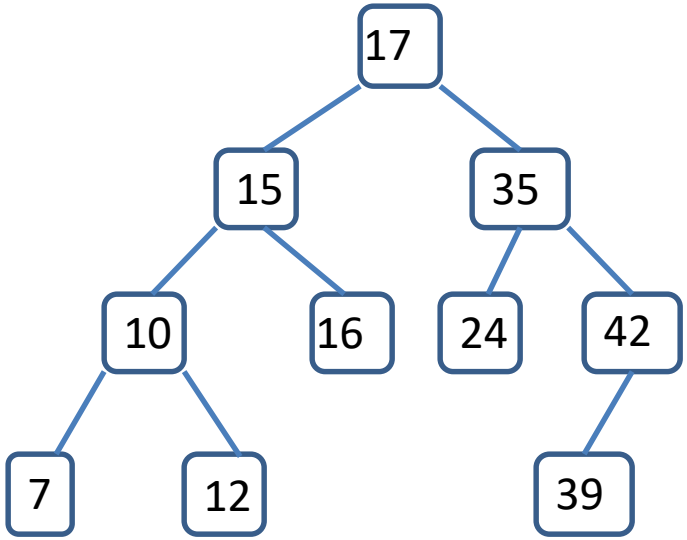
See Section 19.1 of the text, p 687-696.

Ultimately we are interested in maps, but all of the interesting ideas pertain to keys; the values in a key-value map just tag along. To keep our examples simple we will just consider the keys,as though the values mapped to the keys aren’t even there.

Consider the following *Binary Search Tree*

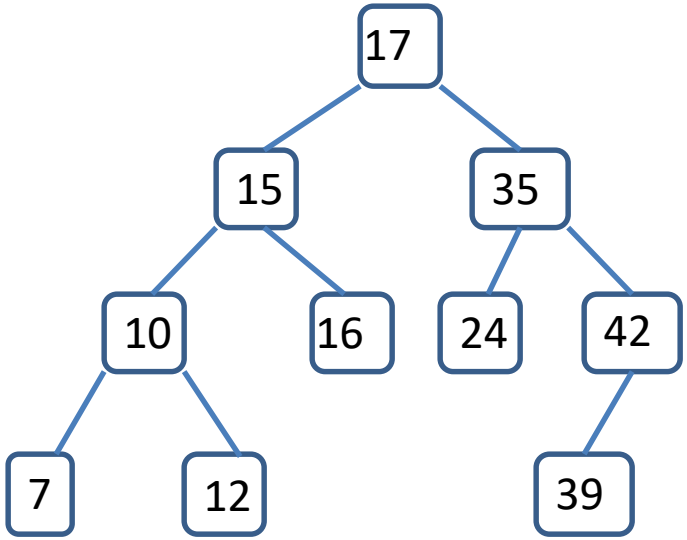


This tree has a nice property: for every node, all of the nodes in its left subtree have values less than the node's value; all of the nodes in its right subtree have values that are larger.



Question: How can we print the nodes of a Binary Search tree in increasing order?

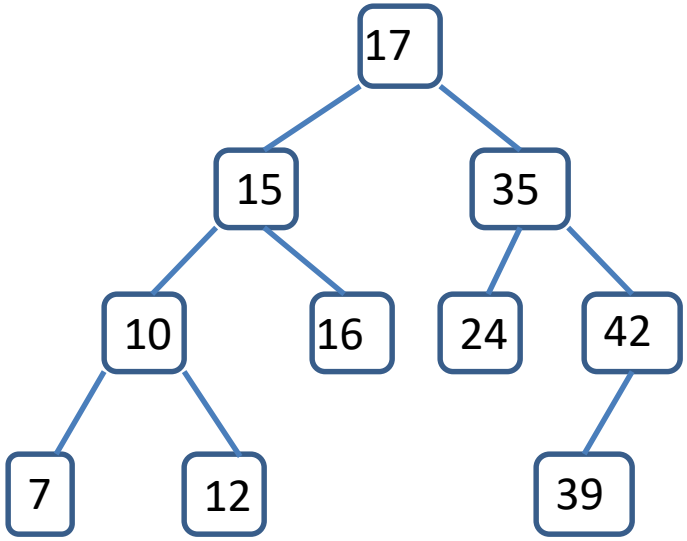
A. Preorder traversal (node, left child, right child)
B. Inorder traversal (left child, node, right child)
C. Postorder traversal (left child, right child, node)
D. None of the above



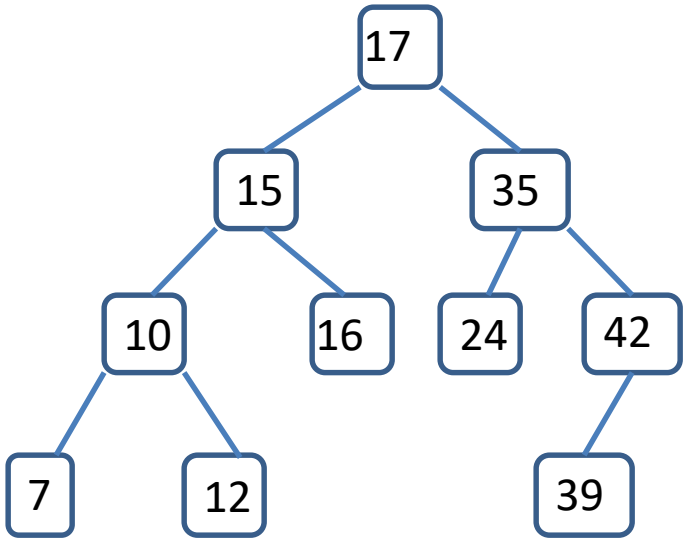
Question: How can we print the nodes of a Binary Search tree in increasing order?

Answer:

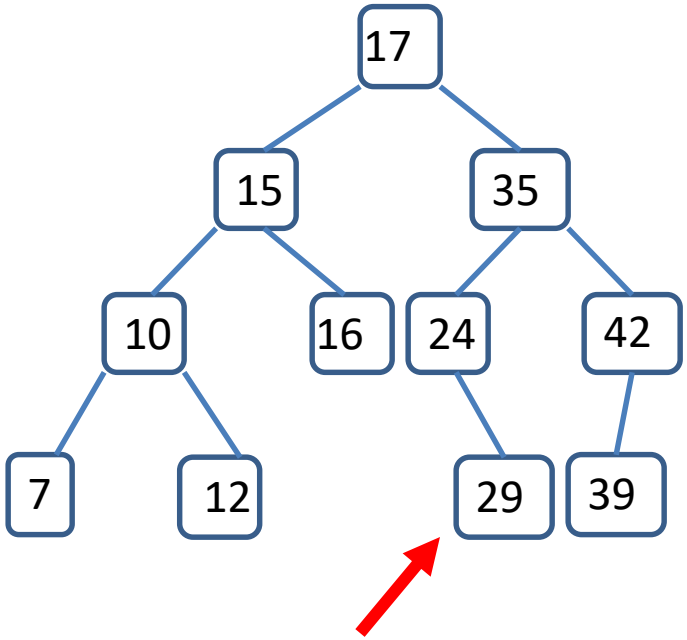
B. Inorder traversal (left child, node, right child)



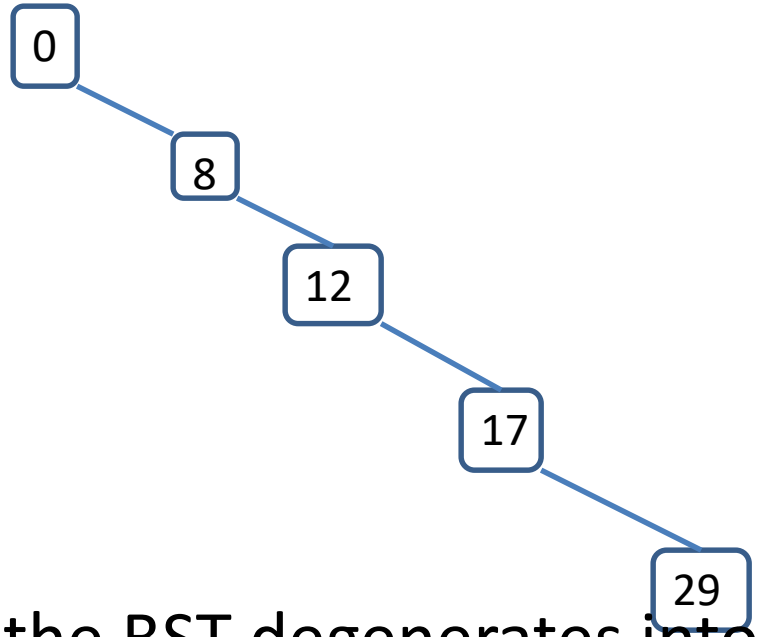
There is an easy algorithm for searching a BST to determine if it contains a node with key k: We start at the root. At each step if k is greater than the key of the node we move to the right child; if k is less we move to the left. This ends either when we find k or when we get to a null child.



To insert key 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 key 29 to our BST we notice that it is greater than 17 (the root), less than 35, and greater than 24. The node with key 24 does not have a right child, so we add 29 there:


17
15
35
10
16
24
42
7
12
29
39

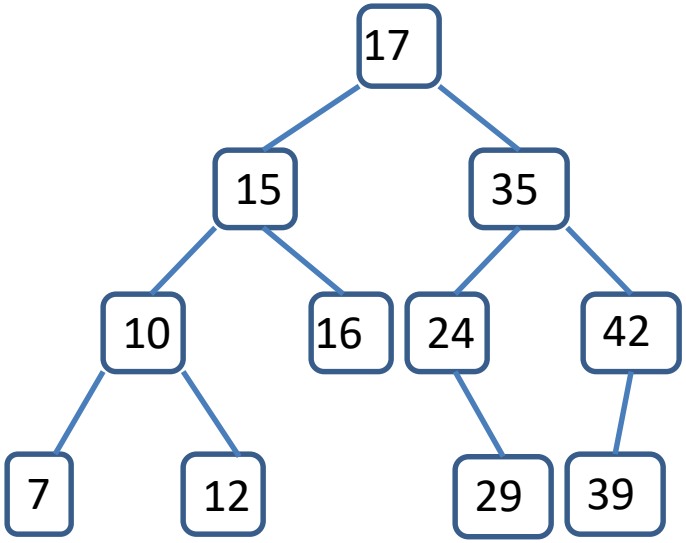
If we are lucky Binary Search Trees are balanced (each node has 2 children and those children have the same number of nodes) and each step of a search eliminates half of the nodes. However, we might not be so lucky. Consider the BST we would get if we start with root key 0 and then add, in order, the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



In this case searching the BST degenerates into a linear search.

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

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



Question: What is an algorithm for finding the maximum value in a BST?

A. Start at the root and go right as far as you can; when a node has no right child it is the largest.
B. Do an inorder traversal; the last value is the largest
C. Walk down the tree comparing the values of the children; move in the direction of the larger child.
D. None of the above.

Both A and B are correct answers. An in-order traversal of a BST lists the nodes in increasing order, so the last key it gets to is the largest. However, there is no reason to go through all of the nodes in a tree to find the largest one. Just start at the root and follow the right links until you get to a node that does not have a right child. That node's]key is the largest.

The *find* method is easy to implement: just follow the definition of a Binary Search Tree. If you get to a null node, the key you seek isn't there and you can return null.

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.

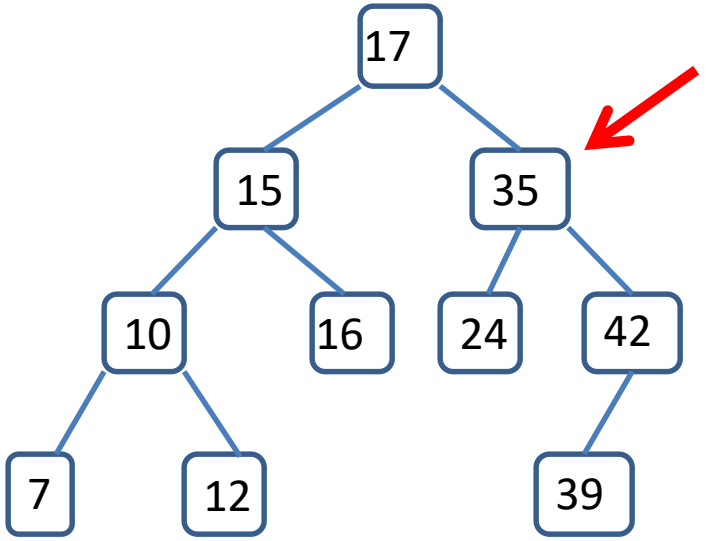
Alternatively you can recurse to do an insert. To insert the item in the tree rooted at node t, look to see if it should go in the left or right subtree. If the left subtree, replace t.left by the result of inserting the new item in t.left. In other words

```
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.

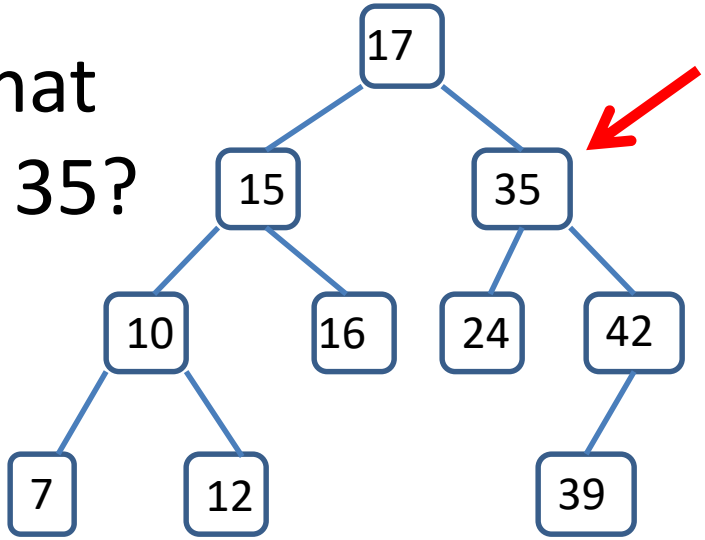
The only BST method that is tricky to implement is remove.

Consider our example tree:

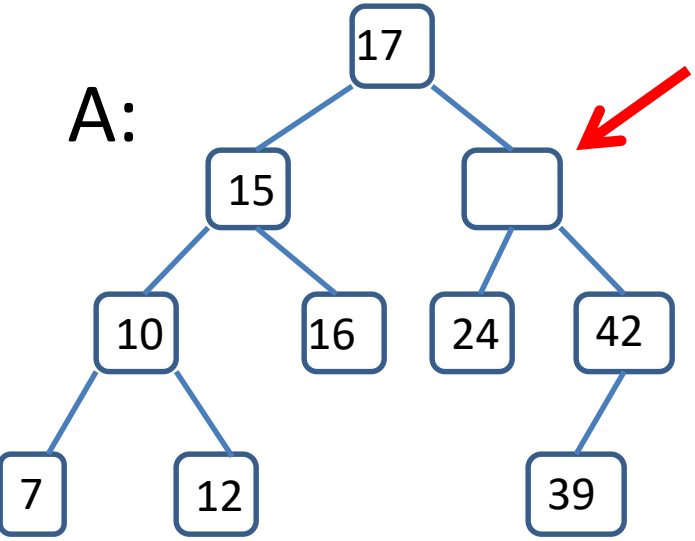


Suppose we want to remove the node with key 35. That is tricky. So we cheat and remove something that is easy.

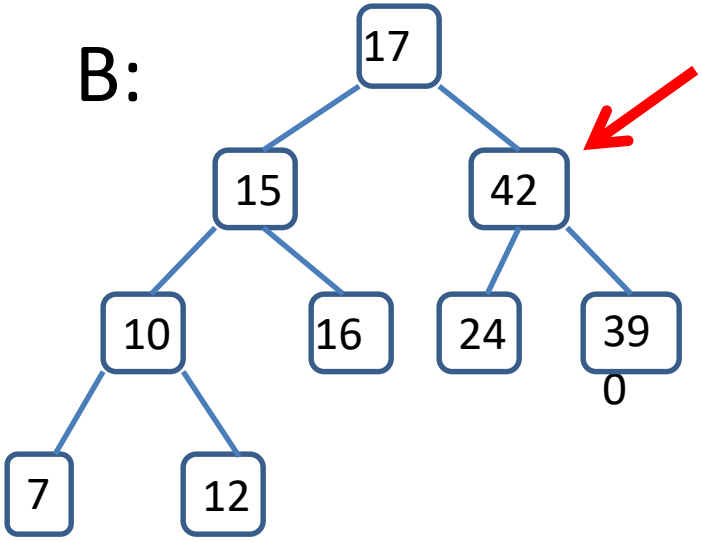
Question: What is a correct BST that might result from removing node 35?



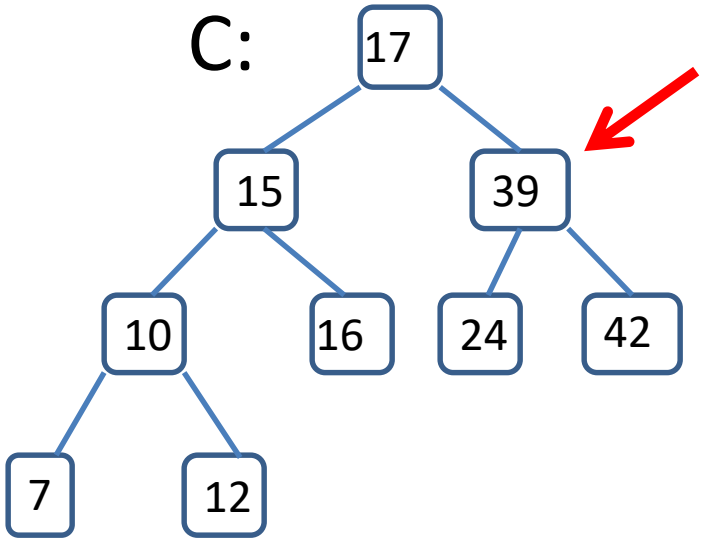
A:



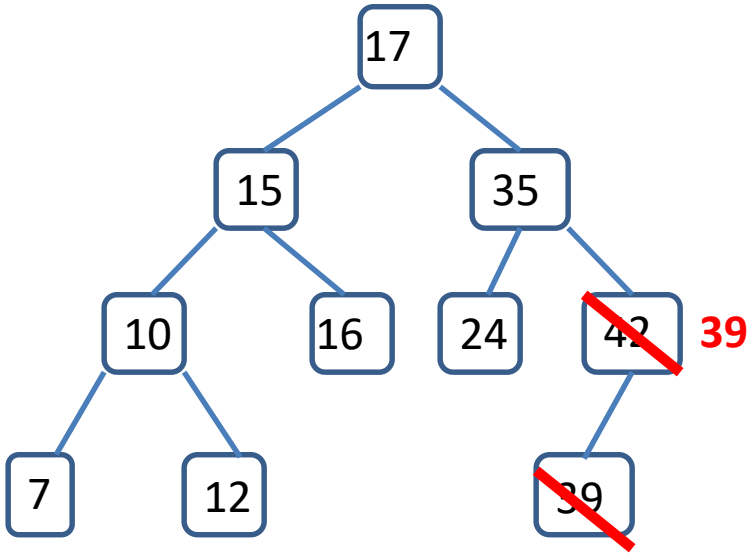
B:



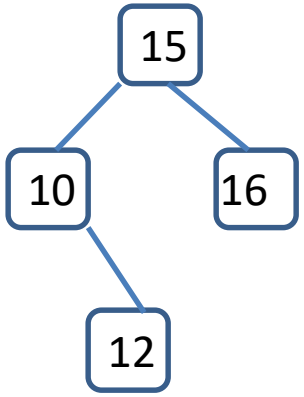
C:



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:



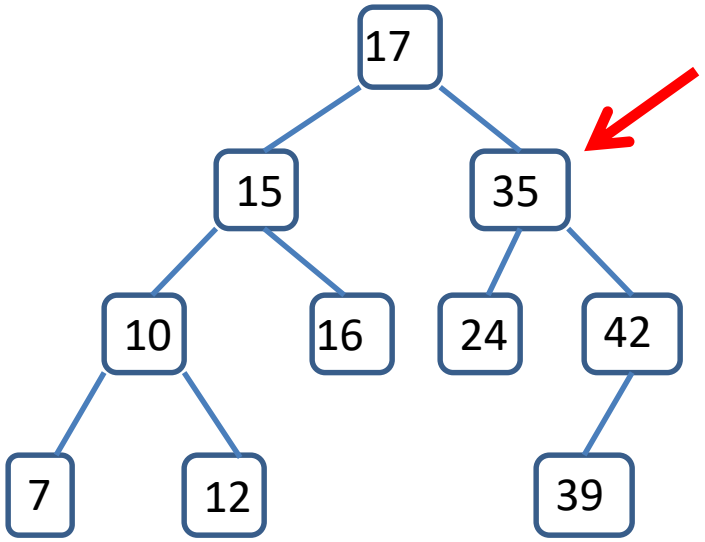
In this tree we could replace node 10 with 12 and the result would still be a BST.



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.

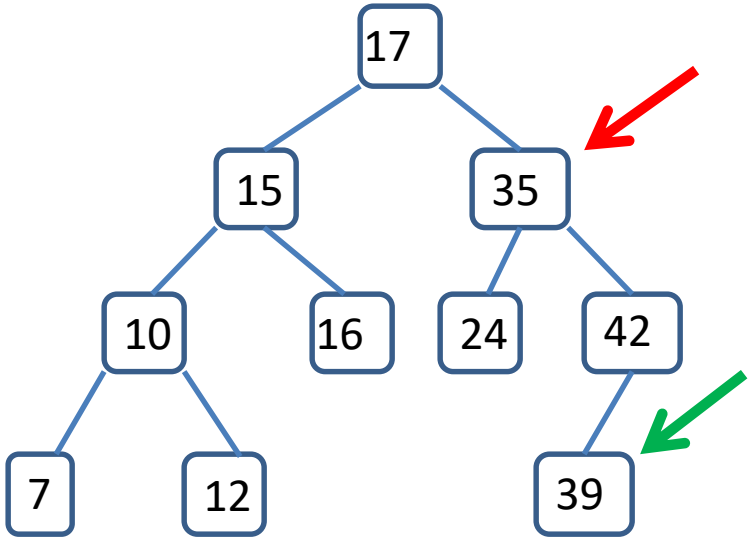
So let's go back to the problem of removing an interior node of a BST.

We want to remove node 35.

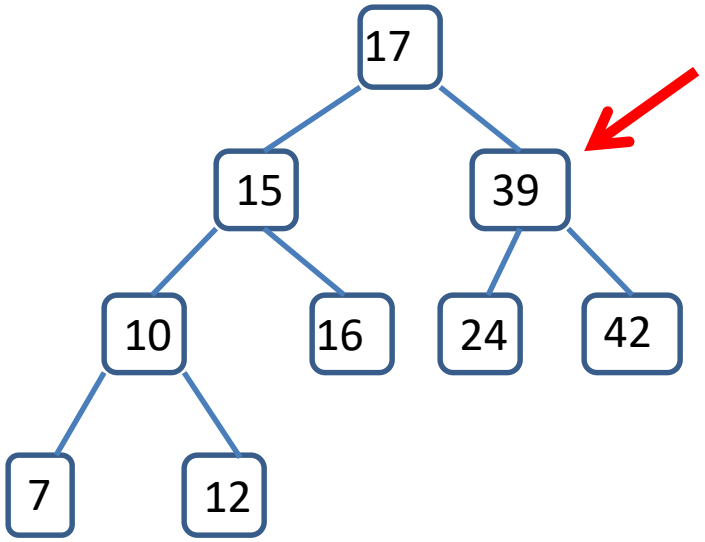


Since it has two children we go to its right child; all of the keys in this subtree are greater than 35.

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

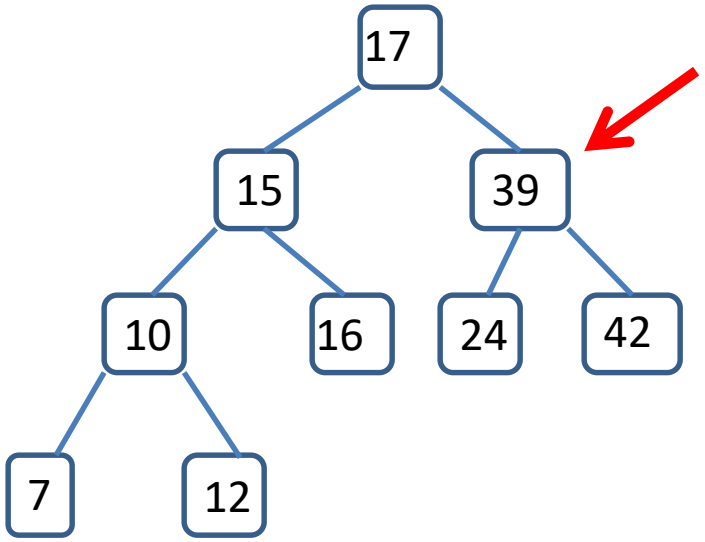


Note that 39 is less than every other node in the right subtree of 35, and greater than every node in the left subtree. We can switch it with 35 and delete the node that currently contains 39. The result is a BST that has all of the original values except 35:



Isn't that clever!

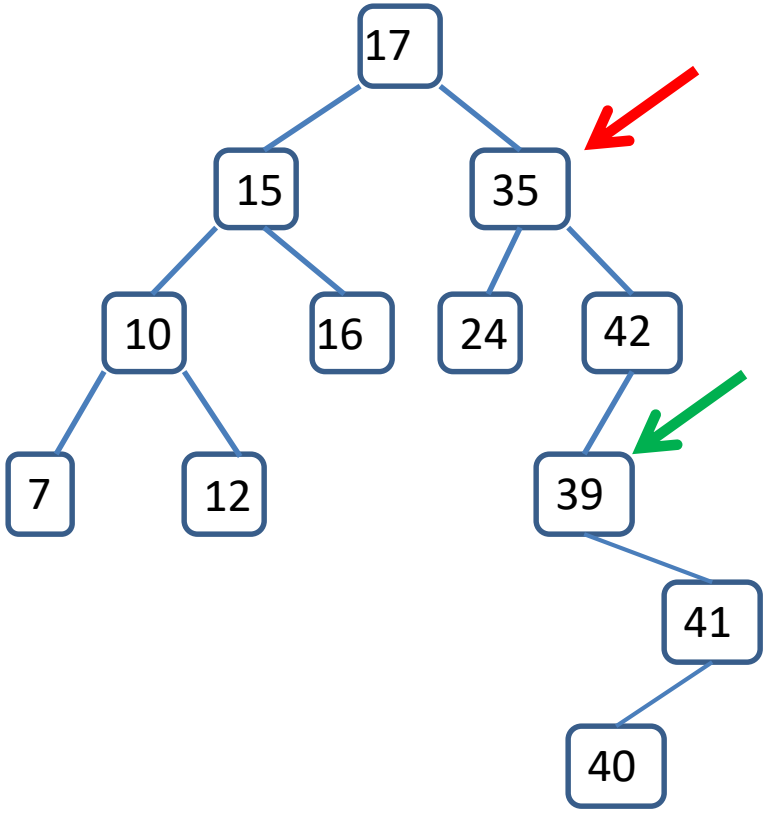
To implement this we make a method that starts at a given node and returns the smallest key below it, and another method that starts at a node and removes the node with the smallest key below it.



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.

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 key 35 by 39: