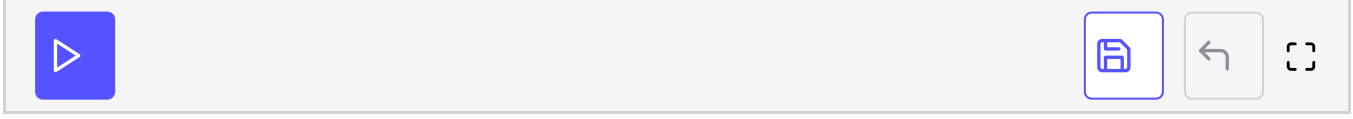
Deletion in Binary Search Tree

(Reading time: 5 minutes)

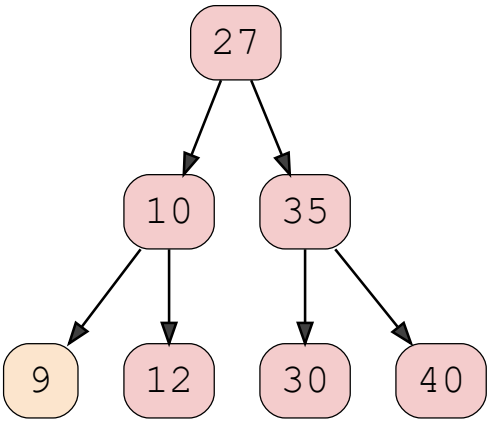
The function to remove a node from a binary search tree is as follows:

```
remove(data) {
  this.root = this.removeNode(this.root, data)
}

removeNode(node, data) {
  if (!node) {
    return null;
  }
  if (data < node.data) {
    node.left = this.removeNode(node.left, data);
    return node;
  } else if (data > node.data) {
    node.right = this.removeNode(node.right, data);
    return node;
  } else {
    if (!node.left && !node.right) {
      node = null;
      return node;
    }
    if (!node.left) {
      node = node.right;
      return node;
    }
    if (!node.right) {
      node = node.left;
      return node;
    }

    let min = this.findMinNode(node.right);
    node.data = min.data;
    node.right = this.removeNode(node.right, min.data);
    return node;
  }
}
```

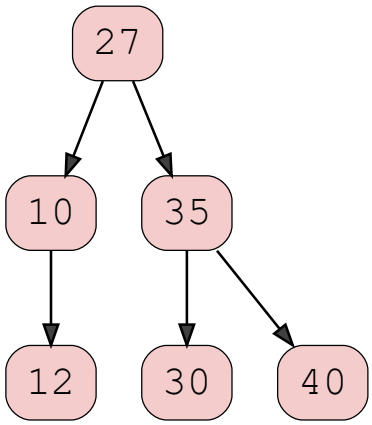


BST after removing '9'

Removing a leaf



1 of 2



2 of 2

First, we invoke the **remove** function. We set the root's value equal to whatever gets returned from the **removeNode** function. We pass two arguments to this function: the root, and the node's value that we want to delete, 9 in this case.

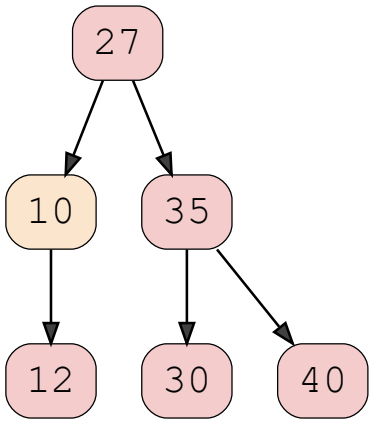
Inside the **removeNode** function, we get to the first if-statement. Nodes are present in the tree, so **!node** returns false. We get to the second if-statement, which returns true as data (9) is smaller than **node.data** (27). Now, we set the left node's value equal to whatever gets returned from the **removeNode** function, which we invoke again, only now with the value of the left node (10) as the first argument.

Again, we get to the first if-statement, which returns false. The second if-statement returns true again, as **data** (9) is smaller than **node.data** (10). Again, we set the left node's value equal to whatever gets returned from the

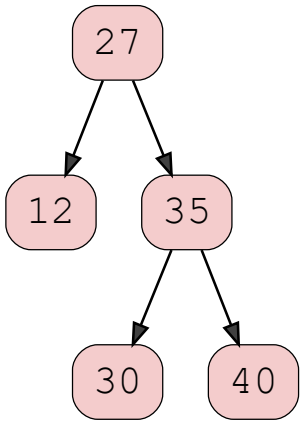
removeNode function, with the new left node (9) as the first argument.

The second if-statement now returns false, because **data** (9) is not smaller than **node.data** (9). The else-if statement also returns false, because 9 is not bigger than 9. We get into the else block, where we get our first if-statement. This one, **!node.left** && **!node.right**, returns true: the node is a leaf and doesn't have a left and right node. We set the node equal to null: the node gets deleted.

Removing a value with **one child** node:



1 of 2



2 of 2

First, we invoke the **remove** function. We set the root's value equal to whatever gets returned from the **removeNode** function. We pass two arguments to this function: the root, and the node's value that we want to delete, 10 in this case.

Inside the **removeNode** function, we get to the first if-statement. There are

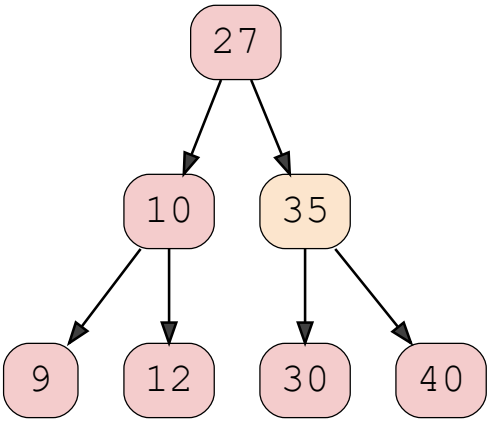
nodes, so **!node** returns false. We get to the second if-statement, which returns

true as **data** (10) is smaller than **node.data** (27). Now, we set the left node's value equal to whatever gets returned from the **removeNode** function, which we invoke again, only now with the value of the left node (10) as the first argument.

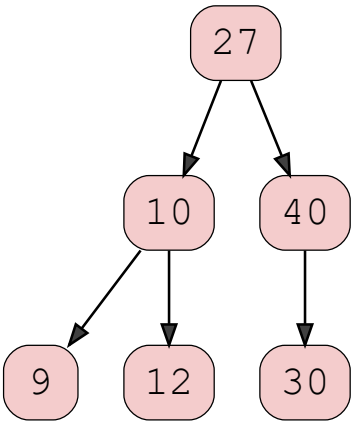
Again, we get to the first if-statement, which returns false. The second if-statement returns false, as **data** (10) is not smaller than **node.data** (10). Also, the else-if statement returns false, as 10 is not bigger than 10. We get into the else block, where we get to the first if-statement. This one, **!node.left** && **!node.right** returns false, as the node we want to delete has a node.right, namely the one with the value 12. The second if-statement in the else-block, !node.left, returns true: there is no left node! We now replace the current node with its node on the right. The node has now been deleted and replaced.

Removing a value with **two child** nodes:

EXAMPLE 1:



1 of 2



2 of 2

First, we invoke the **remove** function. We set the root's value equal to whatever gets returned from the **removeNode** function. We pass two arguments to this function: the root, and the node's value that we want to delete, 35 in this case.

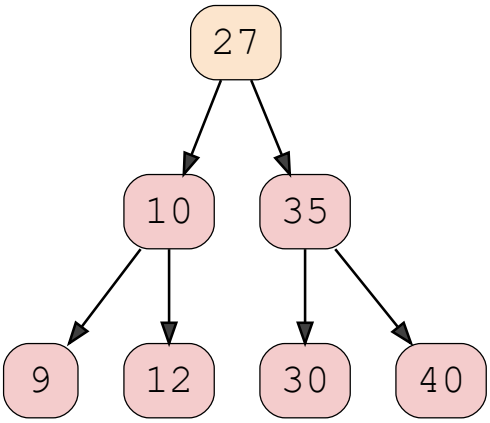
Inside the **removeNode** function, we get to the first if-statement. There are nodes, so **!node** returns false. We get to the second if-statement, which returns false as **data** (35) is bigger than **node.data** (27). Now, we set the right node's value equal to whatever gets returned from the **removeNode** function, which we invoke again, only now with the value of the right node (35) as the first argument.

Again, we get to the first if-statement, which returns false. The second if-statement returns false as well, as **data** (35) is not smaller than **node.data** (35). Also, the else-if statement returns false, as 35 is not bigger than 35. We get into the else block, where all if-statements return false: there is a left node and a right node. This means that we get to the last part:

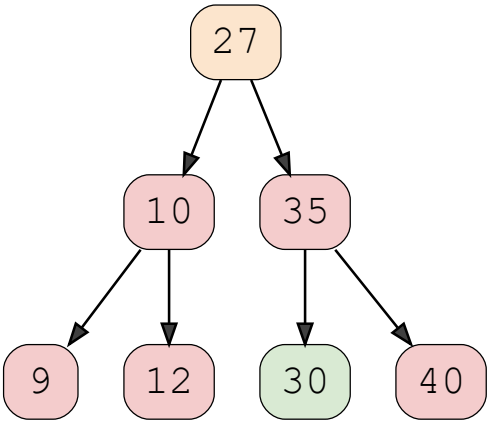
```
let min = this.findMinNode(node.right);
node.data = min.data;
node.right = this.removeNode(node.right, min.data);
return node;
```

The **findMinNode** is a function we haven't implemented yet: however it finds the minimum value of the right subtree. It needs to be the right subtree because the minimum value means that it's at least always bigger than every value on the left subtree, and always smaller than any value on the right subtree! After we find that minimum node, we replace the current node with the minimum node, by setting its value equal to the minimum node's value. We then remove the node that's on the right, as we replaced the current node with that node (otherwise they would be duplicated). The node has now been deleted and replaced!

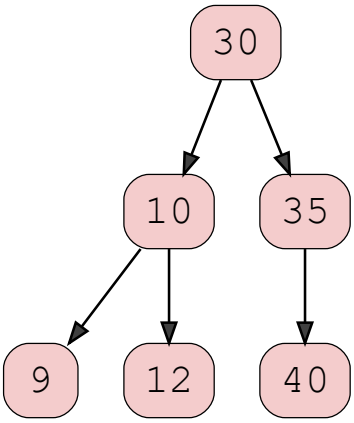
EXAMPLE 2:



1 of 3



2 of 3



3 of 3

As the node we want to delete is the root node, we immediately go to the last part. The minimum value on the right subtree is 30, so the current node gets replaced with the minimum node's value.

In the next lesson, I'll discuss traversal of a binary search tree.