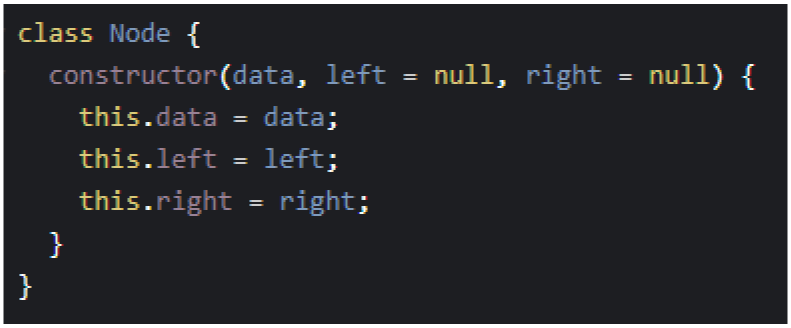
STEP ONE: we create 2 classes:

-class Node

-class BST



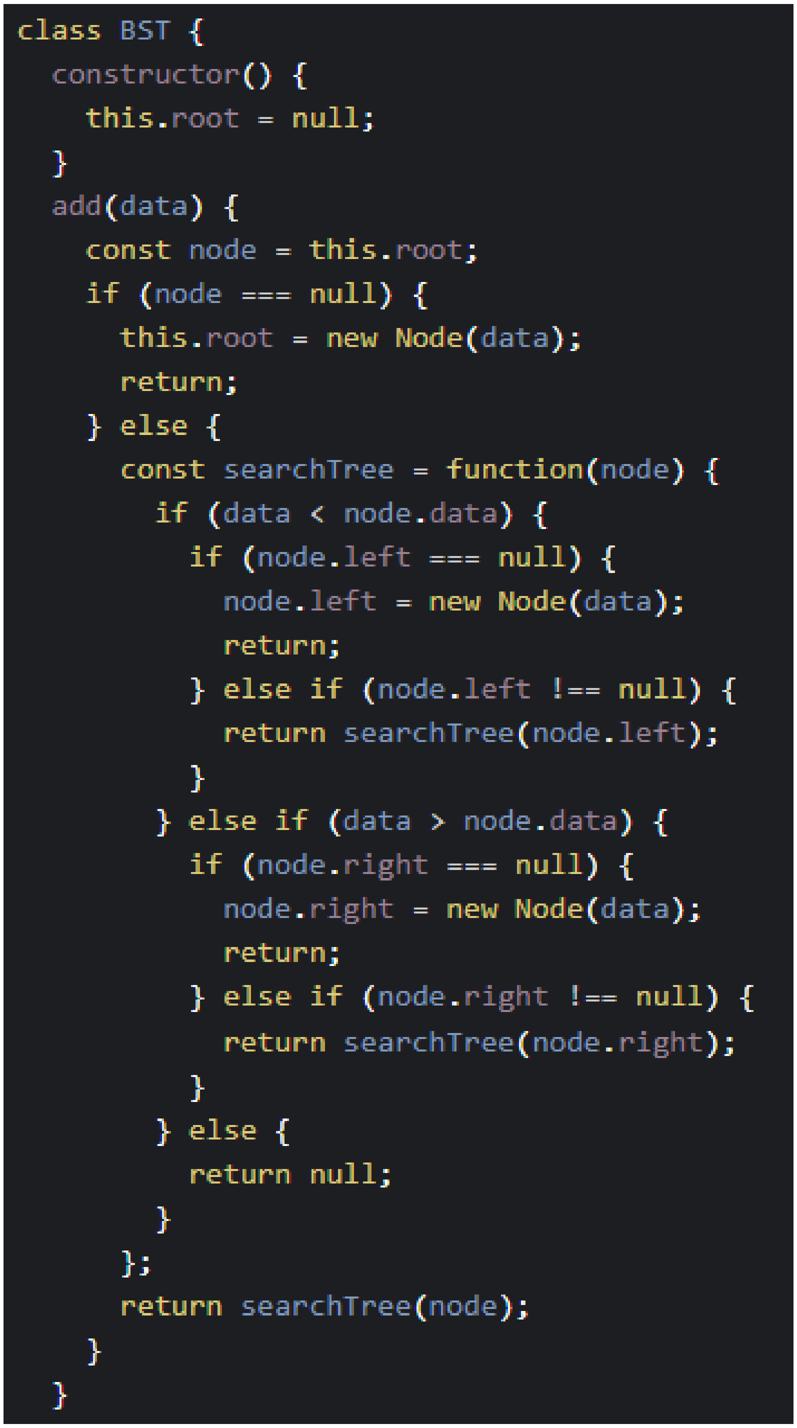
```
class Node {
  constructor(data, left = null, right = null) {
    this.data = data;
    this.left = left;
    this.right = right;
  }
}
```

...this defines every node as having:

-- a “center” data part,

--a left node,

--a right node.



```javascript
class BST {
  constructor() {
    this.root = null;
  }
  add(data) {
    const node = this.root;
    if (node === null) {
      this.root = new Node(data);
      return;
    } else {
      const searchTree = function(node) {
        if (data < node.data) {
          if (node.left === null) {
            node.left = new Node(data);
            return;
          } else if (node.left !== null) {
            return searchTree(node.left);
          }
        } else if (data > node.data) {
          if (node.right === null) {
            node.right = new Node(data);
            return;
          } else if (node.right !== null) {
            return searchTree(node.right);
          }
        } else {
          return null;
        }
      };
      return searchTree(node);
    }
  }
```

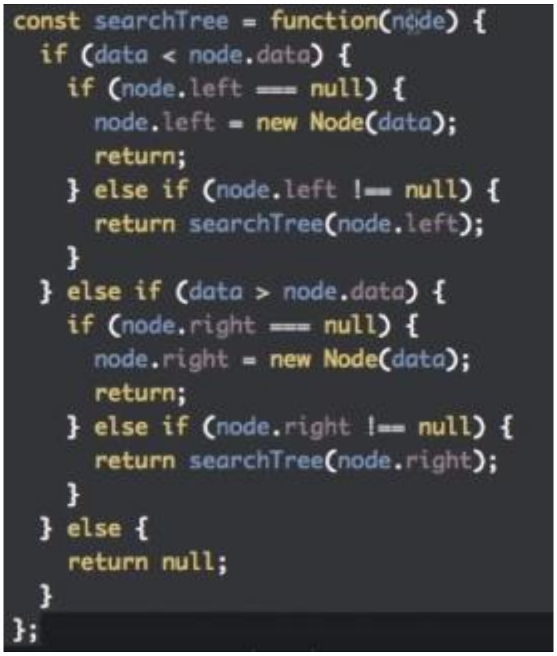
The class Node has parts:

--the **constructor** that creates the root node as starting out as null (not having children).

--the **add** function, which defines how we will add data to the tree:

--the root node, if null, will be the reference point to start adding data (new node data).

--else, we will need to search using **searchTree**, which is a recursive function:



```
const searchTree = function(node) {
  if (data < node.data) {
    if (node.left === null) {
      node.left = new Node(data);
      return;
    } else if (node.left !== null) {
      return searchTree(node.left);
    }
  } else if (data > node.data) {
    if (node.right === null) {
      node.right = new Node(data);
      return;
    } else if (node.right !== null) {
      return searchTree(node.right);
    }
  } else {
    return null;
  }
};
```

... if the **data** is smaller than the **node data**, and node data is null, move it to the left.

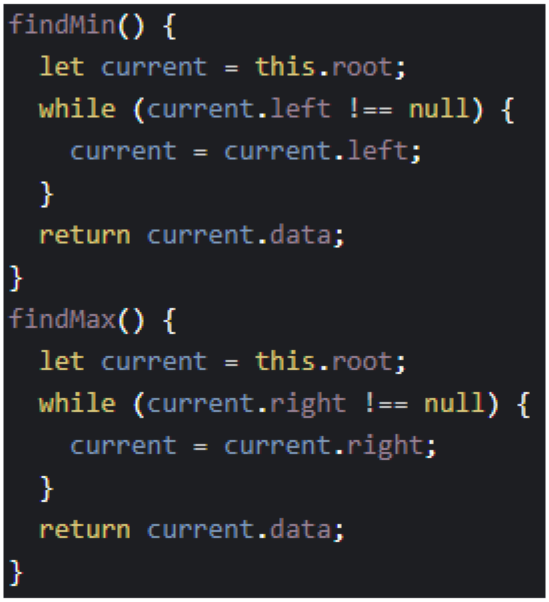
… else if **node data** is not null, search elsewhere for a null node data to place the **data**.

… the **data** is larger than the **node data**, move it to the right.

… else if **node data** is not null, search elsewhere for a null node data to place the **data**.

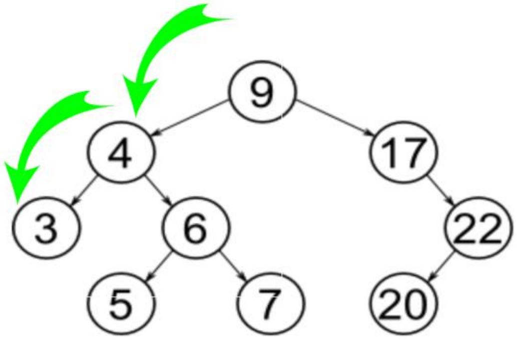
… and if **data** is not less than or greater than **node data**, then they must be equal, in which case, we return **null**… which means, we don't add the new data to the tree.

STEP TWO: Find the maximum height and minimum height:



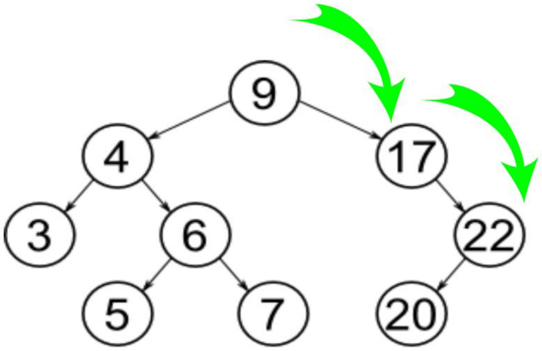
```
findMin() {
  let current = this.root;
  while (current.left !== null) {
    current = current.left;
  }
  return current.data;
}
findMax() {
  let current = this.root;
  while (current.right !== null) {
    current = current.right;
  }
  return current.data;
}
```

For findMin, we loop from **this**.**root** (9) towards the left, until we reach a null node:



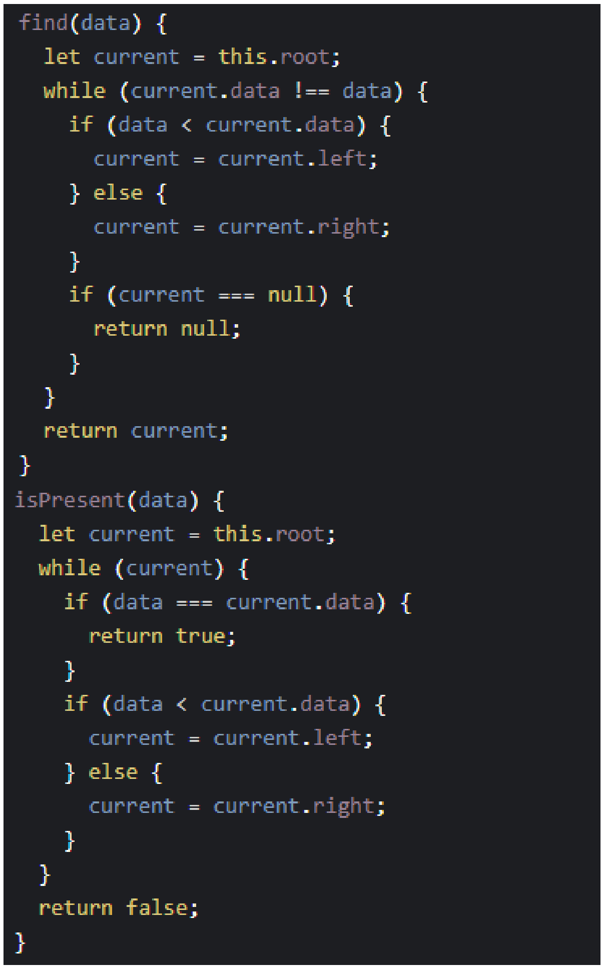
… 3 is null, because there‘s nothing to its left... thus, it is returned as the **current**.**data**.

For findMax, we move loop from **this**.**root** (9) towards the right, until we reach a null node:



… 3 is null, because there‘s nothing to its right... thus, it isreturned as the **current**.**data**.

STEP THREE: The **find()** function and **isPresent()** function are very similar...



```
find(data) {
  let current = this.root;
  while (current.data !== data) {
    if (data < current.data) {
      current = current.left;
    } else {
      current = current.right;
    }
    if (current === null) {
      return null;
    }
  }
  return current;
}
isPresent(data) {
  let current = this.root;
  while (current) {
    if (data === current.data) {
      return true;
    }
    if (data < current.data) {
      current = current.left;
    } else {
      current = current.right;
    }
  }
  return false;
}
```

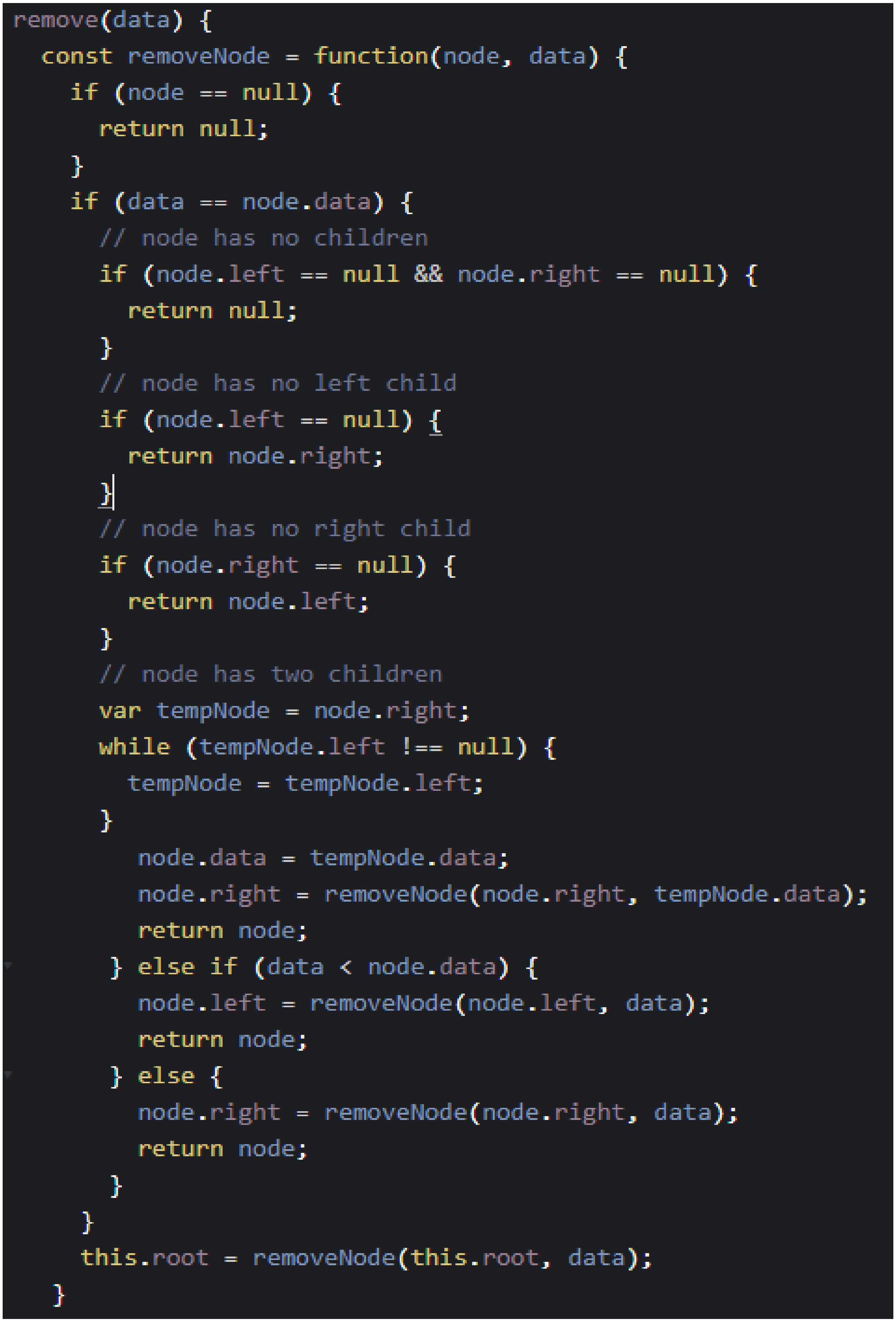
… except:

find() returns the node of the data

and

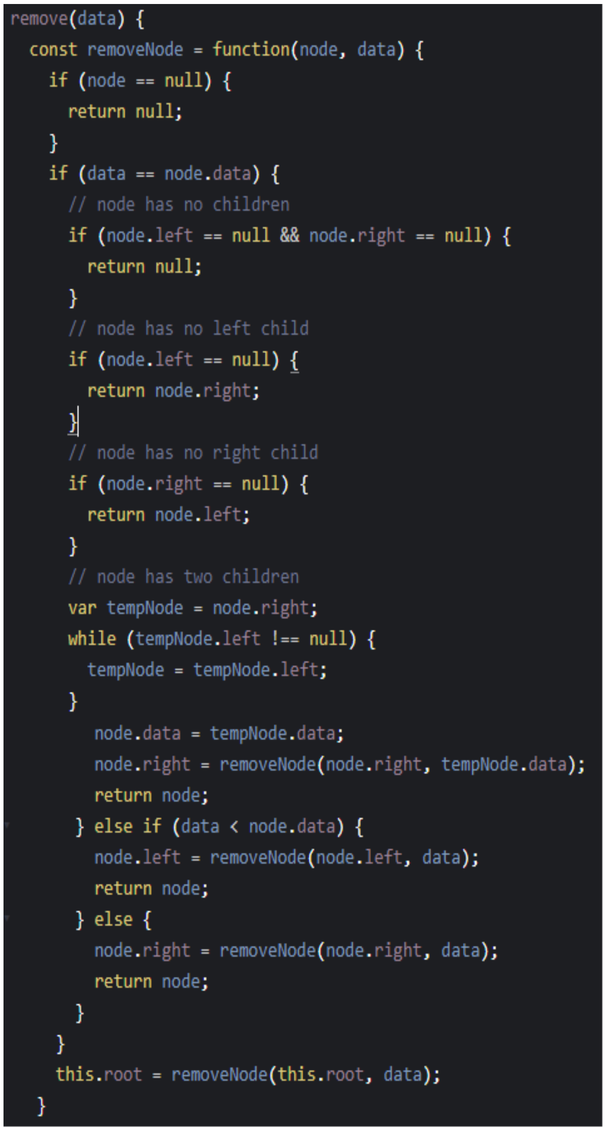
isPresent() returns “True”or ”False” for whether or not the data is present

To explain the script:



```
remove(data) {
  const removeNode = function(node, data) {
    if (node == null) {
      return null;
    }
    if (data == node.data) {
      // node has no children
      if (node.left == null && node.right == null) {
        return null;
      }
      // node has no left child
      if (node.left == null) {
        return node.right;
      }
      // node has no right child
      if (node.right == null) {
        return node.left;
      }
      // node has two children
      var tempNode = node.right;
      while (tempNode.left !== null) {
        tempNode = tempNode.left;
      }
         node.data = tempNode.data;
         node.right = removeNode(node.right, tempNode.data);
         return node;
       } else if (data < node.data) {
         node.left = removeNode(node.left, data);
         return node;
       } else {
         node.right = removeNode(node.right, data);
         return node;
       }
     }
     this.root = removeNode(this.root, data);
   }
```

To explain the script:



```
remove(data) {
  const removeNode = function(node, data) {
    if (node == null) {
      return null;
    }
    if (data == node.data) {
      // node has no children
      if (node.left == null && node.right == null) {
        return null;
      }
      // node has no left child
      if (node.left == null) {
        return node.right;
      }
      // node has no right child
      if (node.right == null) {
        return node.left;
      }
      // node has two children
      var tempNode = node.right;
      while (tempNode.left !== null) {
        tempNode = tempNode.left;
      }
         node.data = tempNode.data;
         node.right = removeNode(node.right, tempNode.data);
         return node;
       } else if (data < node.data) {
         node.left = removeNode(node.left, data);
         return node;
       } else {
         node.right = removeNode(node.right, data);
         return node;
       }
     }
     this.root = removeNode(this.root, data);
   }
```

We check to see if we have an empty tree (ie, if **node** equals **null**).

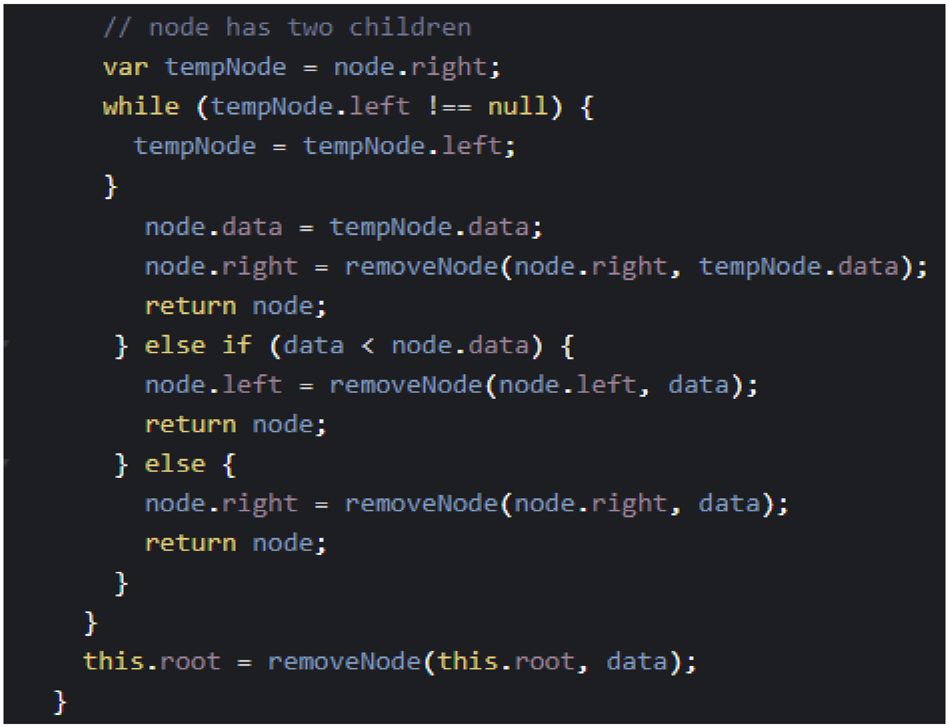
We try to see if we found the data in the tree (ie, if **data** equals **node.data**).
-If we found the node with the data, we have 4 options:

--node has no children.
--node has one child (no left child).
--node has one child (no right child).
--node has two children.

- **this.root** will be passed as the value retrieved by the **removeNode** function.

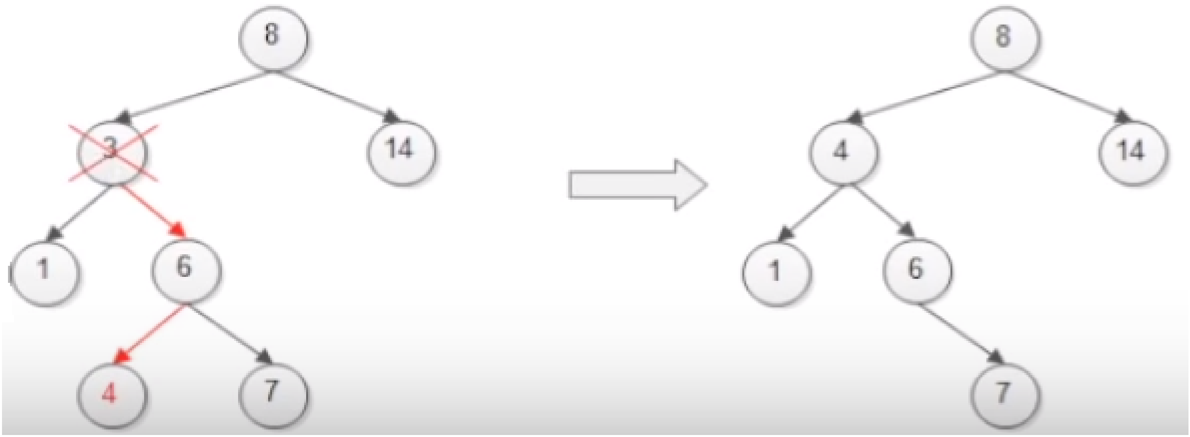
Even more complex, we have the remove function:

The node with 2 children is a little complicated:



```
      // node has two children
      var tempNode = node.right;
      while (tempNode.left !== null) {
        tempNode = tempNode.left;
      }
        node.data = tempNode.data;
        node.right = removeNode(node.right, tempNode.data);
        return node;
      } else if (data < node.data) {
        node.left = removeNode(node.left, data);
        return node;
      } else {
        node.right = removeNode(node.right, data);
        return node;
      }
    }
    this.root = removeNode(this.root, data);
  }
```

Let's say we want to remove the node 3:
-To do so, we need to replace it with another node... 4.

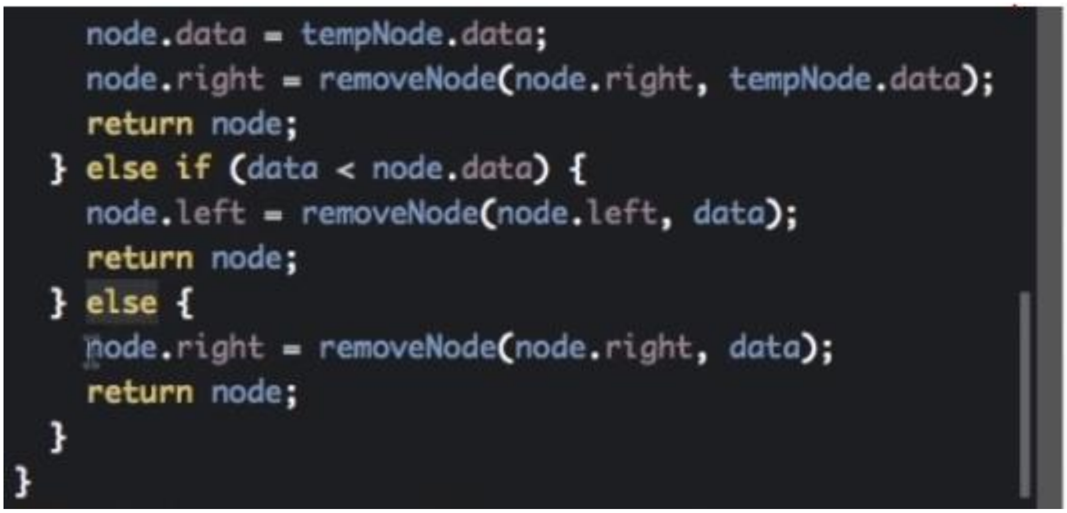


STEP ONE: We delete the 3, then go right, where right.node (6) becomes the tempNode

STEP TWO: We go to the last left node (in this case 4), and it becomes the tempNode

STEP THREE: Set **node.data** to **tempNode.data** (ie, the node **3** gets set to **4**).

STEP FOUR: Set **node.right** to **removeNode(node.right, tempNode.data)...** here it gets recursive:



```
    node.data = tempNode.data;
    node.right = removeNode(node.right, tempNode.data);
    return node;
  } else if (data < node.data) {
    node.left = removeNode(node.left, data);
    return node;
  } else {
    node.right = removeNode(node.right, data);
    return node;
  }
}
```

...if data is less than node, search from left side

…else

...if data is more than node, search from right side