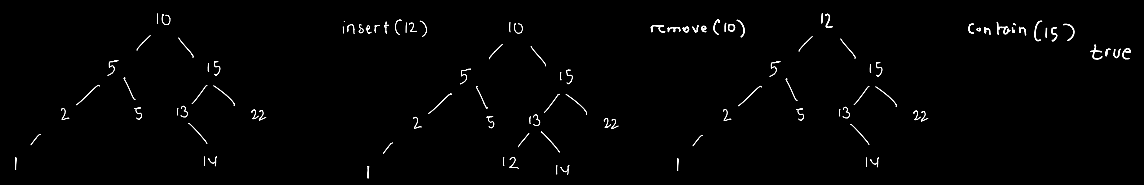
Write a BST class for a Binary Search Tree. The class should support:

1. Inserting values with the insert method
2. Removing values with the remove method; this method should only remove the first instance of a given value
3. Searching for values with the contains method.

Note that you can't remove values from a single-node tree. In other words, calling the REMOVE method on a single-node tree should simply not do anything

Each BST node has an integer VALUE, a LEFT, child node, and a RIGHT child node. A node is said to be a valid BST node if and only if it satisfies the BST property: its VALUE is strictly greater than the values of every node to its left; its VALUE is less than or equal to the values of every node to its right; and its children nodes are either valid BST nodes themselves or None/null




```
// Average: O(log(n)) time | O(log(n)) space
// Worst: O(n) time | O(n) space
insert(value) {
  if (value >= this.value) {
    if (this.right === null) {
      this.right = new BST(value);
    } else {
      this.right.insert(value);
    }
  } else {
    if (this.left === null) {
      this.left = new BST(value);
    } else {
      this.left.insert(value);
    }
  }
  return this;
}
```

Idea: Traverse the BST until we reach a null Node and then insert our new BST with the value where the null node was

AVG CASE:

Time: O(log n) (where n is the # of nodes in the BST) since we cut the tree in half at every iteration

Space: O(log n) since the recursive calls use frames on the call stack

WORST CASE:

O(n) for both since we could have a BST with only left or only right nodes therefore not cutting the tree in half each time

```
// Average: O(log(n)) time | O(log(n)) space
// Worst: O(n) time | O(n) space
contains(value) {
  if (value > this.value) {
    if (this.right === null) return false;
    return this.right.contains(value);
  } else if (value < this.value) {
    if (this.left === null) return false;
    return this.left.contains(value);
  } else {
    return true;
  }
}
```

Idea: traverse the tree until we find the value and return true. If after traversal the value is not found, return false or when we reach a null node

AVG / WORST CASE: same as above

```
class BST {
  constructor(value) {
    this.value = value;
    this.left = null;
    this.right = null;
  }

  // Average: O(log(n)) time | O(log(n)) space
  // Worst: O(n) time | O(n) space
  insert(value) {
    if (value >= this.value) {
      if (this.right === null) {
        this.right = new BST(value);
      } else {
        this.right.insert(value);
      }
    } else {
      if (this.left === null) {
        this.left = new BST(value);
      } else {
        this.left.insert(value);
      }
    }
    return this;
  }

  // Average: O(log(n)) time | O(log(n)) space
  // Worst: O(n) time | O(n) space
  contains(value) {
    if (value > this.value) {
      if (this.right === null) return false;
      return this.right.contains(value);
    } else if (value < this.value) {
      if (this.left === null) return false;
      return this.left.contains(value);
    } else {
      return true;
    }
  }

  // Average: O(log(n)) time | O(log(n)) space
  // Worst: O(n) time | O(n) space
  remove(value, parent = null) {
    if (value < this.value) {
      if (this.left !== null) {
        this.left.remove(value, this);
      }
    } else if (value > this.value) {
      if (this.right !== null) {
        this.right.remove(value, this);
      }
    } else {
      if (this.left !== null && this.right !== null) {
        this.value = this.right.getMinValue();
        this.right.remove(this.value, this);
      } else if (parent === null) {
        if (this.left !== null) {
          this.value = this.left.value;
          this.right = this.left.right;
          this.left = this.left.left;
        } else if (this.right !== null) {
          this.value = this.right.value;
          this.left = this.right.left;
          this.right = this.right.right;
        } else {
          // Single node tree, do nothing
        }
      } else if (parent.left === this) {
        parent.left = this.left !== null ? this.left : this.right;
      } else if (parent.right === this) {
        parent.right = this.left !== null ? this.left : this.right;
      }
    }
    return this;
  }

  getMinValue() {
    if (this.left === null) {
      return this.value;
    } else {
      return this.left.getMinValue();
    }
  }
}
```

LOOKING FOR NODE (step 1)

EDGE CASE 1

EDGE CASE 2

2.1

2.2

2.3

EDGE CASE 3

REMOVE NODE (step 2)

REMOVAL:

Steps:
1) Find node you're trying to remove
2) Remove it

Edge cases:

1. Node that has two children nodes
   ↳ Find smallest value in the right sub tree and replace it with the value we're trying to remove
2. When root node doesn't have a parent node
   2.1: If the left node is the only child node
   2.2: If the right node is the only child node
   2.3: Root node we want to remove has no children nodes
3. Node doesn't have two children nodes (one child node or none)
   ↳ Assign left child node to left node if exits, right child node if not
   ↳ Assign right child node to right node if exists, left child otherwise