

Input: tree =

```

      10
     /  \
    5    15
   / \  / \
  2  5 13 12
 / \ / \
1  4 14

```

A potentially invalid BST

Output: true  
A boolean representing whether the BST is valid

```

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

// O(n) time | O(d) space
function validateBst(tree) {
  return validateBstHelper(tree, -Infinity, Infinity);
}

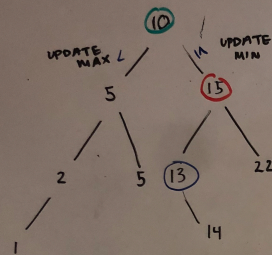
function validateBstHelper(tree, minValue, maxValue) {
  if (tree === null) return true;
  if (tree.value < minValue || tree.value >= maxValue) return false;
  const isValid = validateBstHelper(tree.left, minValue, tree.value);
  return isValid && validateBstHelper(tree.right, tree.value, maxValue);
}

```

Time:  $O(n)$  (where  $n$  is the # of nodes) since we touch every node in the tree to check its validity. Every other operation is  $O(1)$

Space:  $O(d)$  (where  $d$  is the depth of the BST) since we are using frames on the call stack bc of our recursive calls ( $d$  is 4 in this case). If only one branch,  $O(n)$

## Validate BST



13 is a valid BST node because:

$$13 \geq 10 \rightarrow \text{minV}$$

$$13 < 15 \rightarrow \text{maxV}$$

→ if we go left, we set the maxV as the current node's value  
→ if we go right, we set the minV as the current node's value

→ Important to remember BST property:

1. All nodes to the left are strictly less than the current node's value
2. All nodes to the right are greater than or equal to the current node's value
3. All nodes are BST node's themselves or null

```

function validateBST(tree, minV = -Infinity, maxV = Infinity) {
  if (tree === null) return true;
  if (tree.value < minV || tree.value >= maxV) return false;
  const isValidLeft = validateBST(tree.left, minV, tree.value);
  const isValidRight = validateBST(tree.right, tree.value, maxV);
  return isValidLeft && isValidRight;
}

```