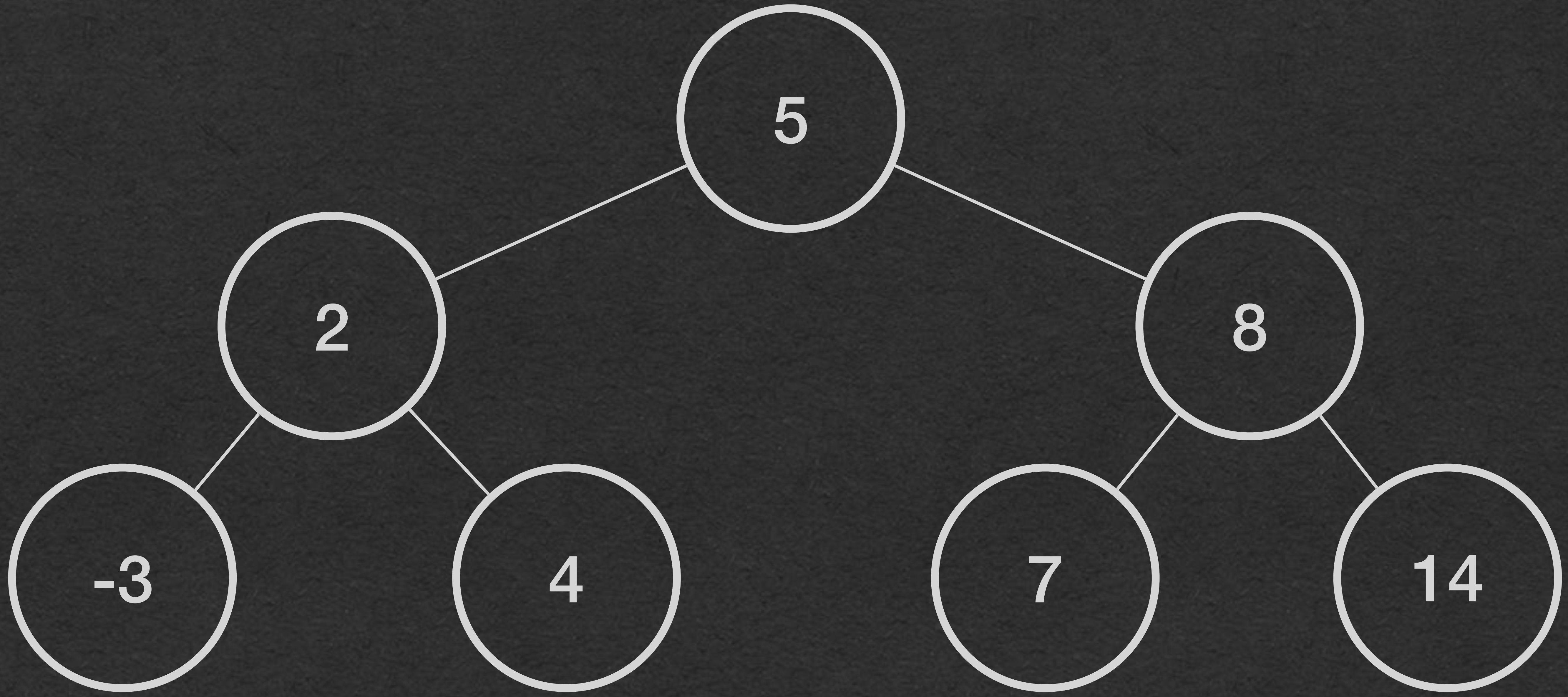
Binary Search Tree
(BST)

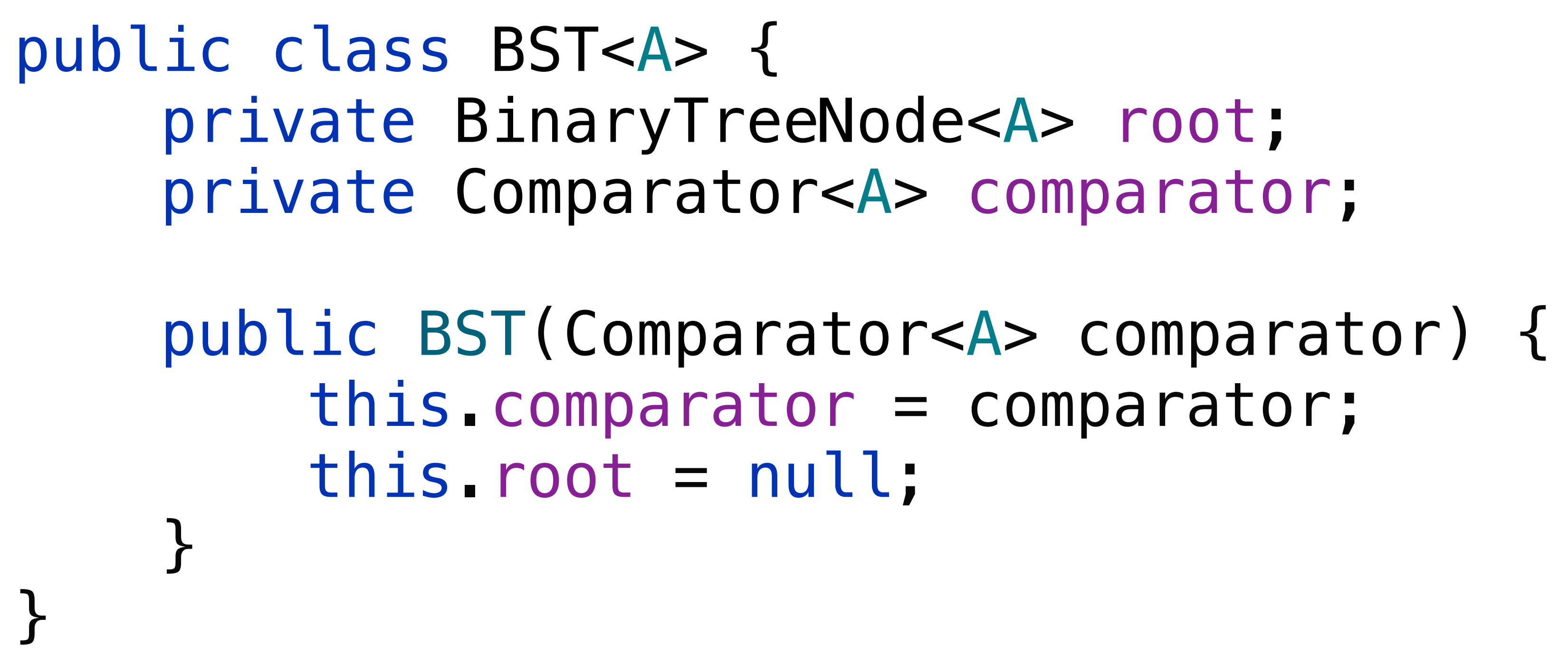
BST - Definition

- For each node:
  - All values in the left subtree come before the node's value
  - All values in the right subtree come after the node's value
  - Duplicate values handled differently based on implementation
    - Sometimes not allowed at all
  - All comparisons are made using a Comparator



BST - Code

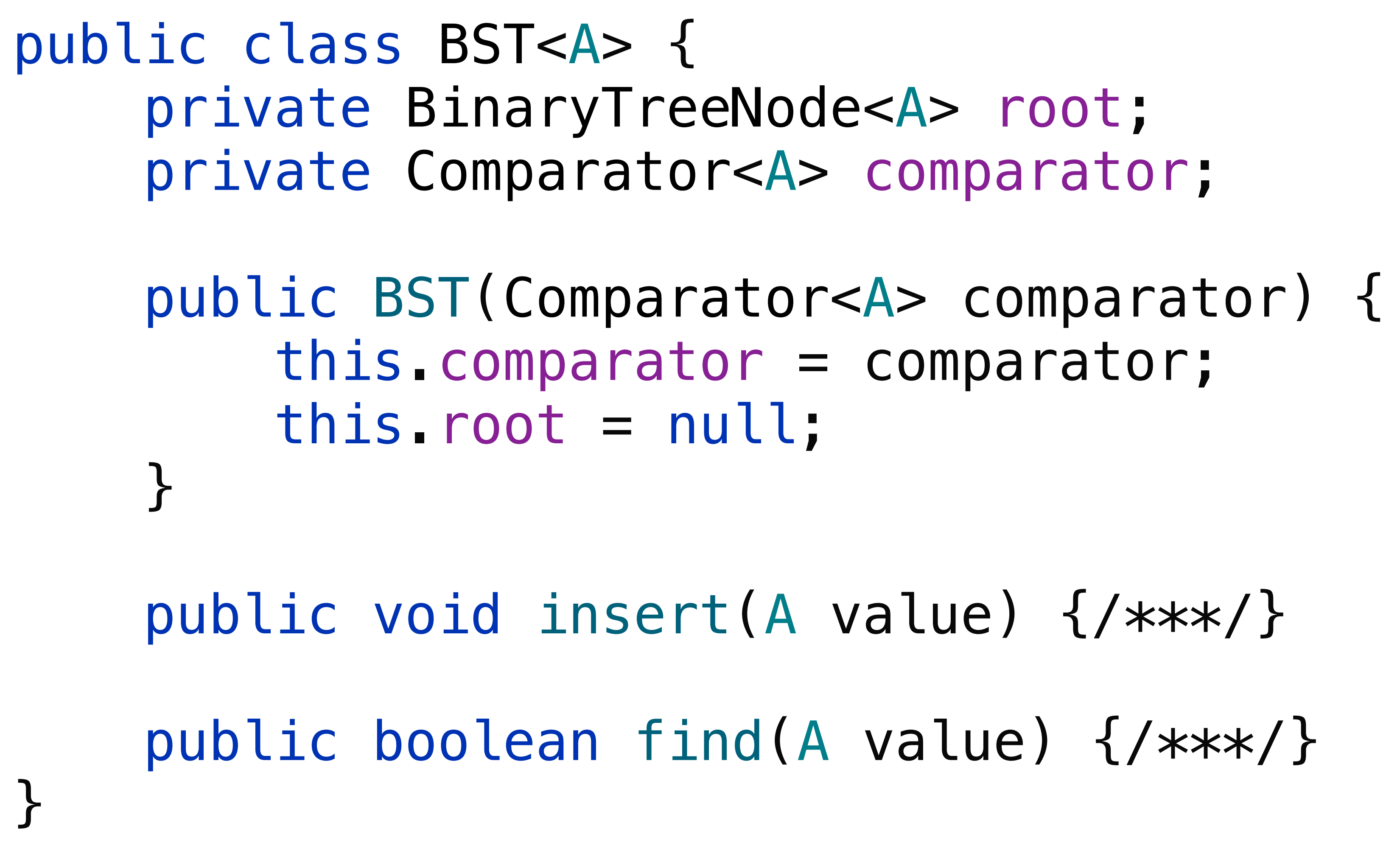
- BST takes a generic type
- BST constructor takes a Comparator of the generic type
  - Determines the sorted order
- Store a reference to the root node of the tree



```
public class BST<A> {
    private BinaryTreeNode<A> root;
    private Comparator<A> comparator;

    public BST(Comparator<A> comparator) {
        this.comparator = comparator;
        this.root = null;
    }
}
```

BST - Code

- BSTs have 2 primary methods that we'll define
    - insert - Add a new value to the BST
    - find - Check if a value is in the BST



```java
public class BST<A> {
    private BinaryTreeNode<A> root;
    private Comparator<A> comparator;

    public BST(Comparator<A> comparator) {
        this.comparator = comparator;
        this.root = null;
    }

    public void insert(A value) {/***/}

    public boolean find(A value) {/***/}
}
```

BST - Usage

```java
public static void main(String[] args) {
    BST<Integer> bst = new BST<>(new IntIncreasing());

    bst.insert(5);
    bst.insert(2);
    bst.insert(8);
    bst.insert(-3);
    bst.insert(4);
    bst.insert(7);
    bst.insert(14);

    System.out.println(bst.find(7));
    System.out.println(bst.find(8));
    System.out.println(bst.find(3));

    System.out.println(bst.root.inOrderTraversal(bst.root));
}
```

```java
public class IntIncreasing extends Comparator<Integer> {

    @Override
    public boolean compare(Integer a, Integer b) {
        return a < b;
    }
}
```

BST - Insert

- Insert a value into a BST

- Must preserve the BST property when inserting
  - Values that come before a node's value are in it's left subtree
  - Values that come after a node's value are in it's right subtree
  - We'll break ties to the right in this implementation

```
public class BST<A> {
    private BinaryTreeNode<A> root;
    private Comparator<A> comparator;

    public BST(Comparator<A> comparator) {
        this.comparator = comparator;
        this.root = null;
    }

    public void insert(A value) {
        if (this.root == null) {
            this.root = new BinaryTreeNode<>(value, null, null);
        } else {
            this.insertHelper(this.root, value);
        }
    }

    private void insertHelper(BinaryTreeNode<A> node, A toInsert) {
        if (this.comparator.compare(toInsert, node.getValue())) {
            if (node.getLeft() == null) {
                node.setLeft(new BinaryTreeNode<>(toInsert, null, null));
            } else {
                insertHelper(node.getLeft(), toInsert);
            }
        } else {
           if (node.getRight() == null) {
                node.setRight(new BinaryTreeNode<>(toInsert, null, null));
            } else {
                insertHelper(node.getRight(), toInsert);
            }
        }
    }
}
```

BST - Insert

- If the BST is empty:
  - The value we're inserting is now the root
- Otherwise:
  - Start the recursion
  - Call a helper method so we can take the current node as a parameter

```java
public class BST<A> {
    private BinaryTreeNode<A> root;
    private Comparator<A> comparator;

    public BST(Comparator<A> comparator) {
        this.comparator = comparator;
        this.root = null;
    }

    public void insert(A value) {
        if (this.root == null) {
            this.root = new BinaryTreeNode<>(value, null, null);
        } else {
            this.insertHelper(this.root, value);
        }
    }

    private void insertHelper(BinaryTreeNode<A> node, A toInsert) {
        if (this.comparator.compare(toInsert, node.getValue())) {
            if (node.getLeft() == null) {
                node.setLeft(new BinaryTreeNode<>(toInsert, null, null));
            } else {
                insertHelper(node.getLeft(), toInsert);
            }
        } else {
           if (node.getRight() == null) {
                node.setRight(new BinaryTreeNode<>(toInsert, null, null));
            } else {
                insertHelper(node.getRight(), toInsert);
            }
        }
    }
}
```

BST - Insert

- Use the Comparator to determine where to move next
    - compare the value we're inserting with the value at the current node
    - Returns true if the value we're inserting comes before the value at this node
    - False otherwise - including ties

```
public class BST<A> {
    private BinaryTreeNode<A> root;
    private Comparator<A> comparator;

    public BST(Comparator<A> comparator) {
        this.comparator = comparator;
        this.root = null;
    }

    public void insert(A value) {
        if (this.root == null) {
            this.root = new BinaryTreeNode<>(value, null, null);
        } else {
            this.insertHelper(this.root, value);
        }
    }

    private void insertHelper(BinaryTreeNode<A> node, A toInsert) {
        if (this.comparator.compare(toInsert, node.getValue())) {
            if (node.getLeft() == null) {
                node.setLeft(new BinaryTreeNode<>(toInsert, null, null));
            } else {
                insertHelper(node.getLeft(), toInsert);
            }
        } else {
           if (node.getRight() == null) {
                node.setRight(new BinaryTreeNode<>(toInsert, null, null));
            } else {
                insertHelper(node.getRight(), toInsert);
            }
        }
    }
}
```

BST - Insert

- If the value to insert comes before the value at the current node:
    - Move to the left
    - If the left child is null, insert the value here
    - If there's a node to the left, make a recursive call to "move" to the left child

```
public class BST<A> {
    private BinaryTreeNode<A> root;
    private Comparator<A> comparator;

    public BST(Comparator<A> comparator) {
        this.comparator = comparator;
        this.root = null;
    }

    public void insert(A value) {
        if (this.root == null) {
            this.root = new BinaryTreeNode<>(value, null, null);
        } else {
            this.insertHelper(this.root, value);
        }
    }

    private void insertHelper(BinaryTreeNode<A> node, A toInsert) {
        if (this.comparator.compare(toInsert, node.getValue())) {
            if (node.getLeft() == null) {
                node.setLeft(new BinaryTreeNode<>(toInsert, null, null));
            } else {
                insertHelper(node.getLeft(), toInsert);
            }
        } else {
           if (node.getRight() == null) {
                node.setRight(new BinaryTreeNode<>(toInsert, null, null));
            } else {
                insertHelper(node.getRight(), toInsert);
            }
        }
    }
}
```

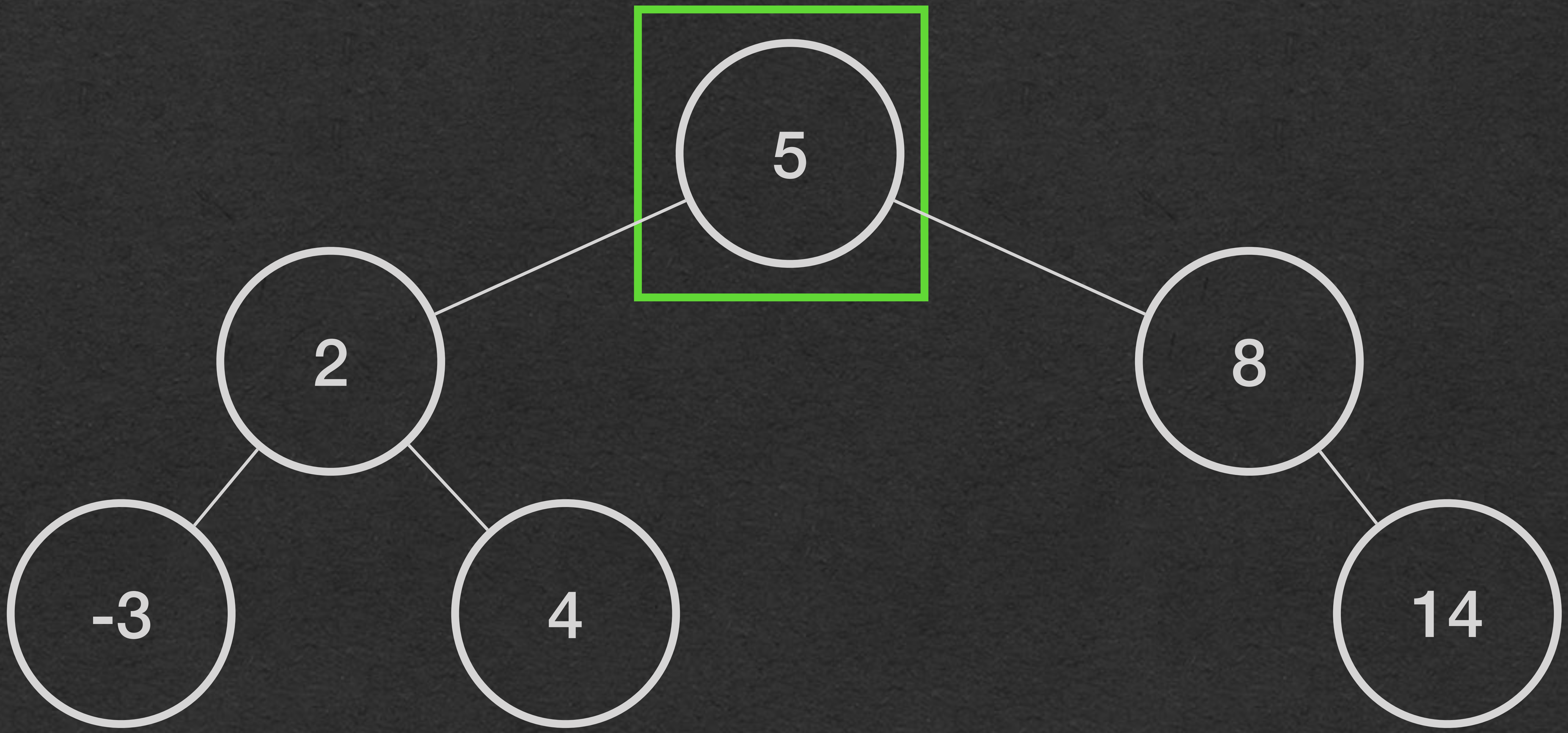
BST - Insert

- If the value to insert does not come before the value at this node:
  - Move to the right
  - Insert here if the right child is null

```java
public class BST<A> {
    private BinaryTreeNode<A> root;
    private Comparator<A> comparator;

    public BST(Comparator<A> comparator) {
        this.comparator = comparator;
        this.root = null;
    }

    public void insert(A value) {
        if (this.root == null) {
            this.root = new BinaryTreeNode<>(value, null, null);
        } else {
            this.insertHelper(this.root, value);
        }
    }

    private void insertHelper(BinaryTreeNode<A> node, A toInsert) {
        if (this.comparator.compare(toInsert, node.getValue())) {
            if (node.getLeft() == null) {
                node.setLeft(new BinaryTreeNode<>(toInsert, null, null));
            } else {
                insertHelper(node.getLeft(), toInsert);
            }
        } else {
           if (node.getRight() == null) {
                node.setRight(new BinaryTreeNode<>(toInsert, null, null));
            } else {
                insertHelper(node.getRight(), toInsert);
            }
        }
    }
}
```

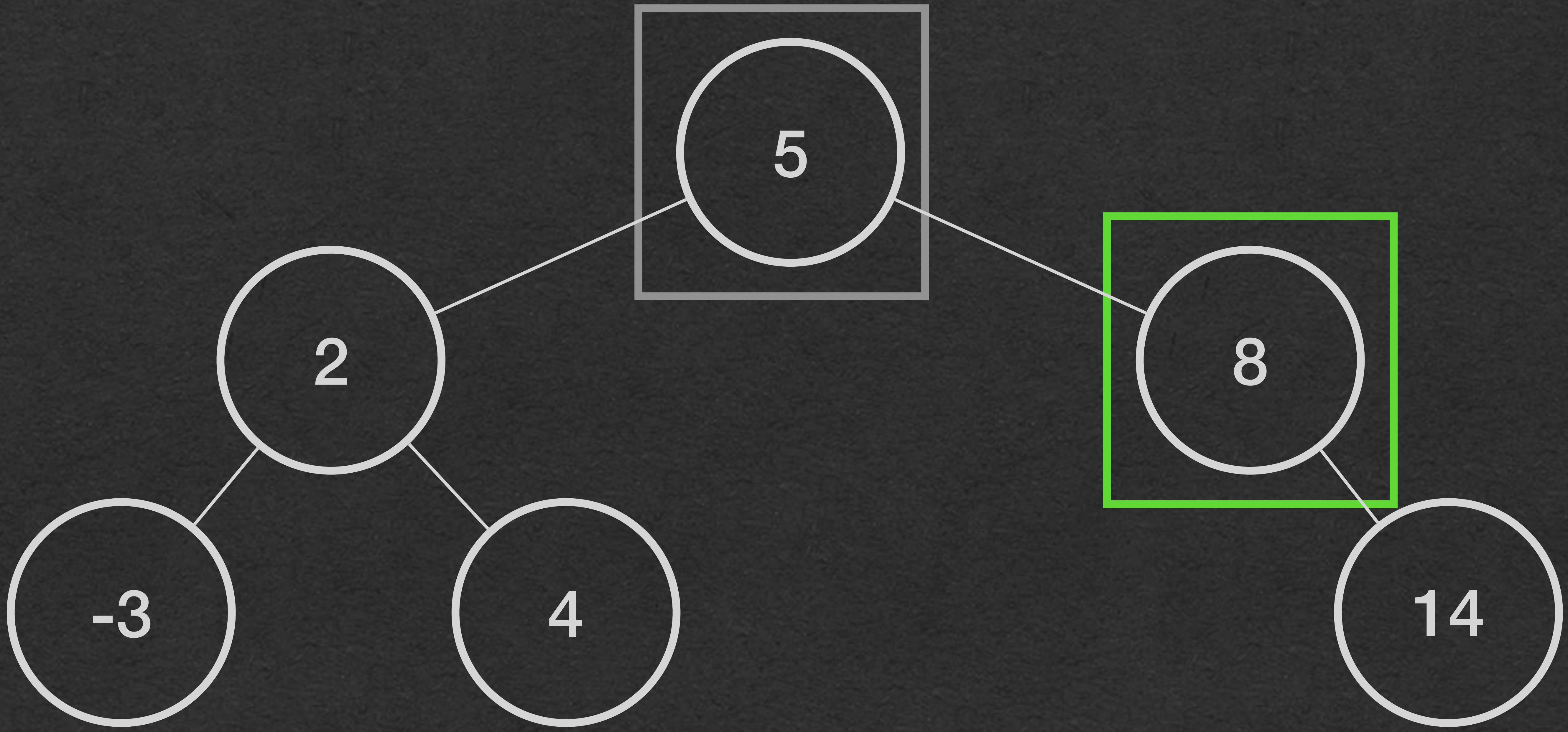
BST - Insert

- Insert 7



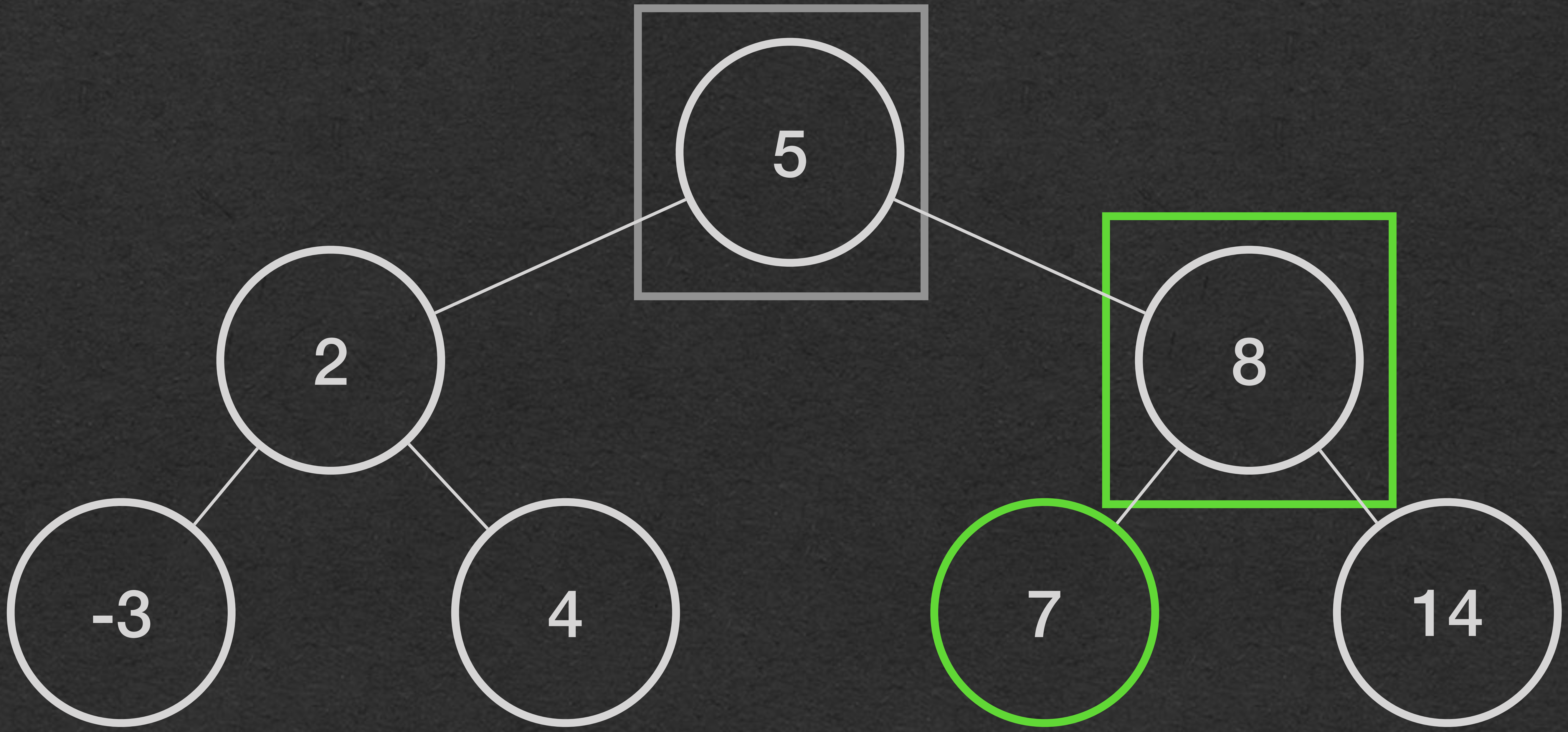
BST - Insert

- Insert 7
- 7 < 5 == false -> move right



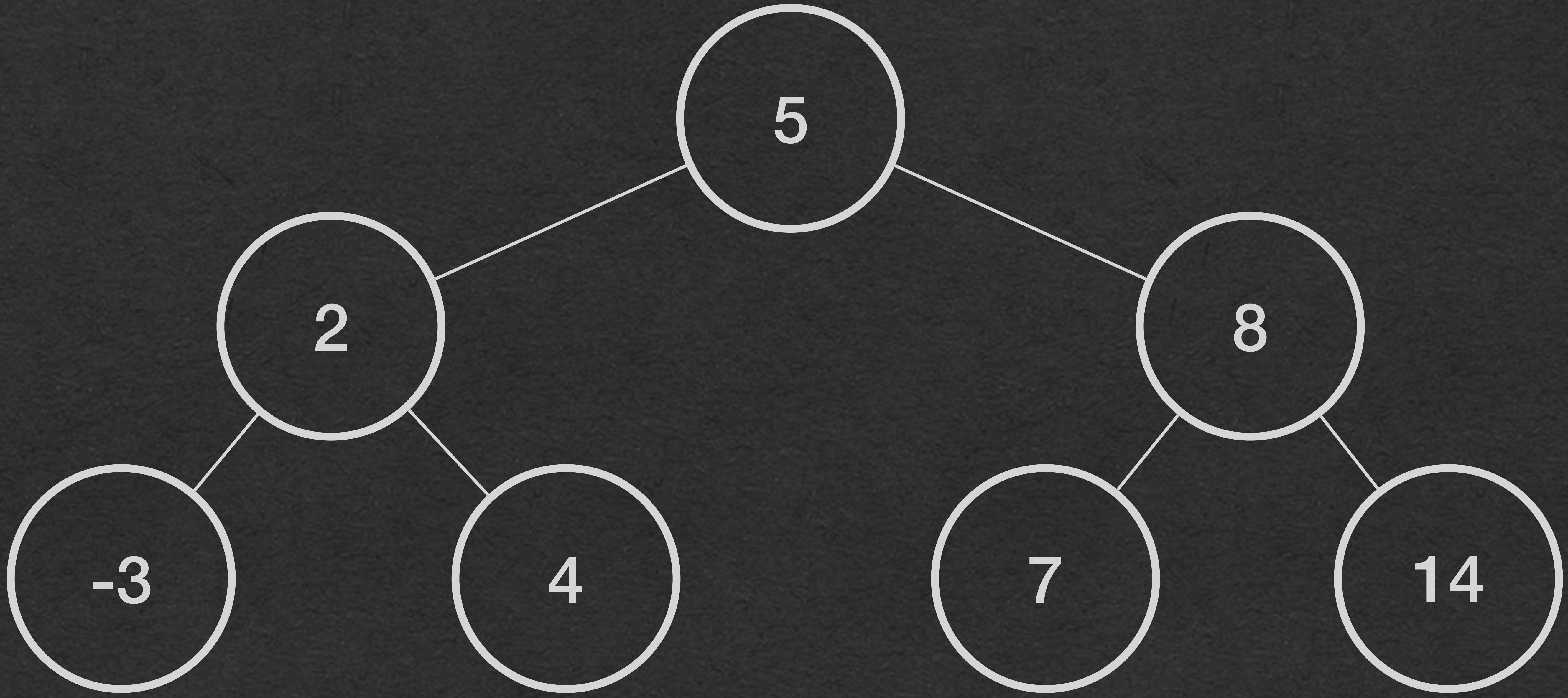
BST - Insert

- Insert 7
- 7 < 5 == false -> move right
- 7 < 8 == true -> move left
- Found null; insert here



BST - Insert

- Recursive calls return



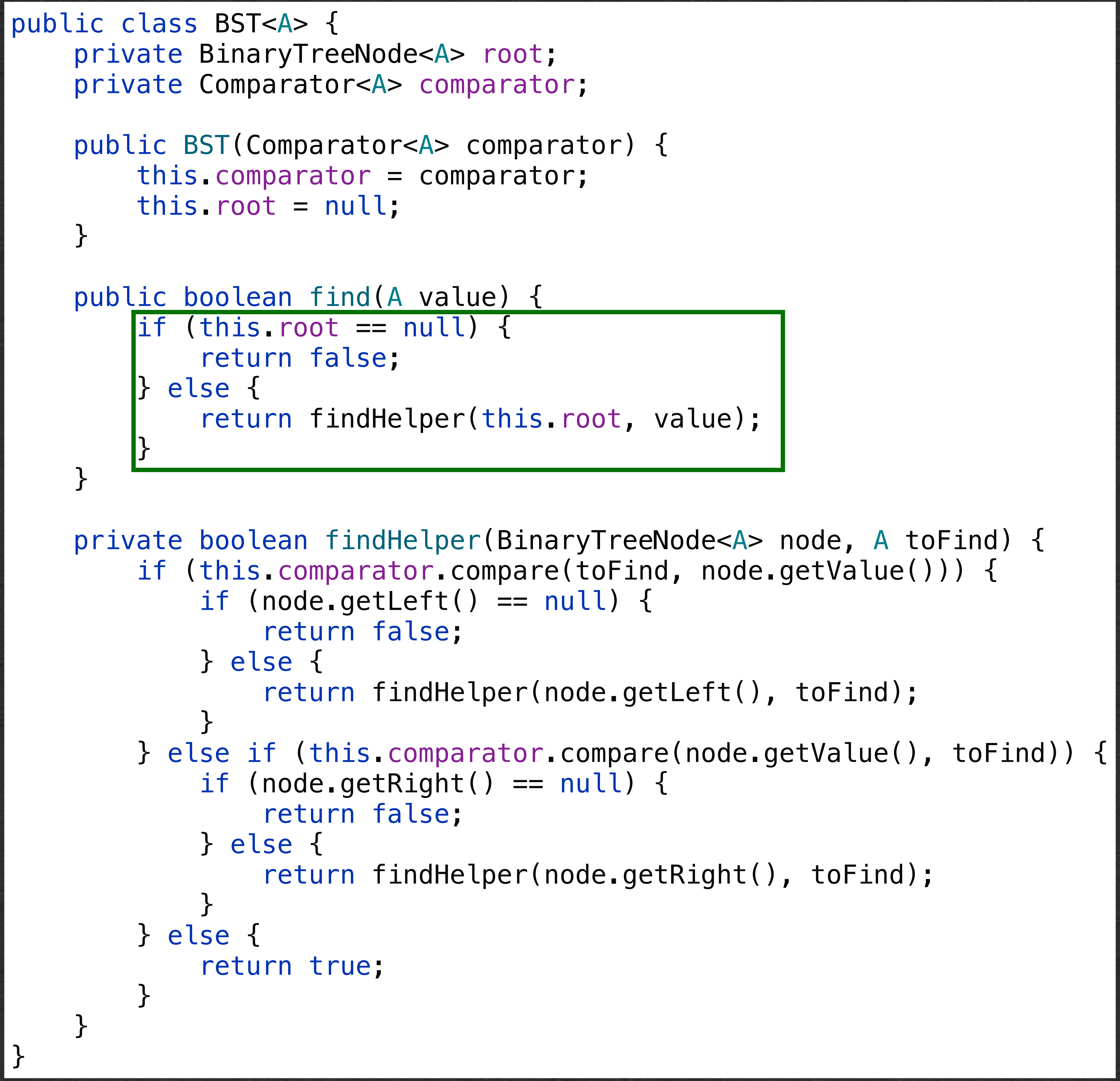
BST - Find

- Find a value in a BST
- Return true if the value is in the BST
- Return false if the value is not in the BST

```java
public class BST<A> {
    private BinaryTreeNode<A> root;
    private Comparator<A> comparator;

    public BST(Comparator<A> comparator) {
        this.comparator = comparator;
        this.root = null;
    }

    public boolean find(A value) {
        if (this.root == null) {
            return false;
        } else {
            return findHelper(this.root, value);
        }
    }

    private boolean findHelper(BinaryTreeNode<A> node, A toFind) {
        if (this.comparator.compare(toFind, node.getValue())) {
            if (node.getLeft() == null) {
                return false;
            } else {
                return findHelper(node.getLeft(), toFind);
            }
        } else if (this.comparator.compare(node.getValue(), toFind)) {
            if (node.getRight() == null) {
                return false;
            } else {
                return findHelper(node.getRight(), toFind);
            }
        } else {
            return true;
        }
    }
}
```

BST - Find

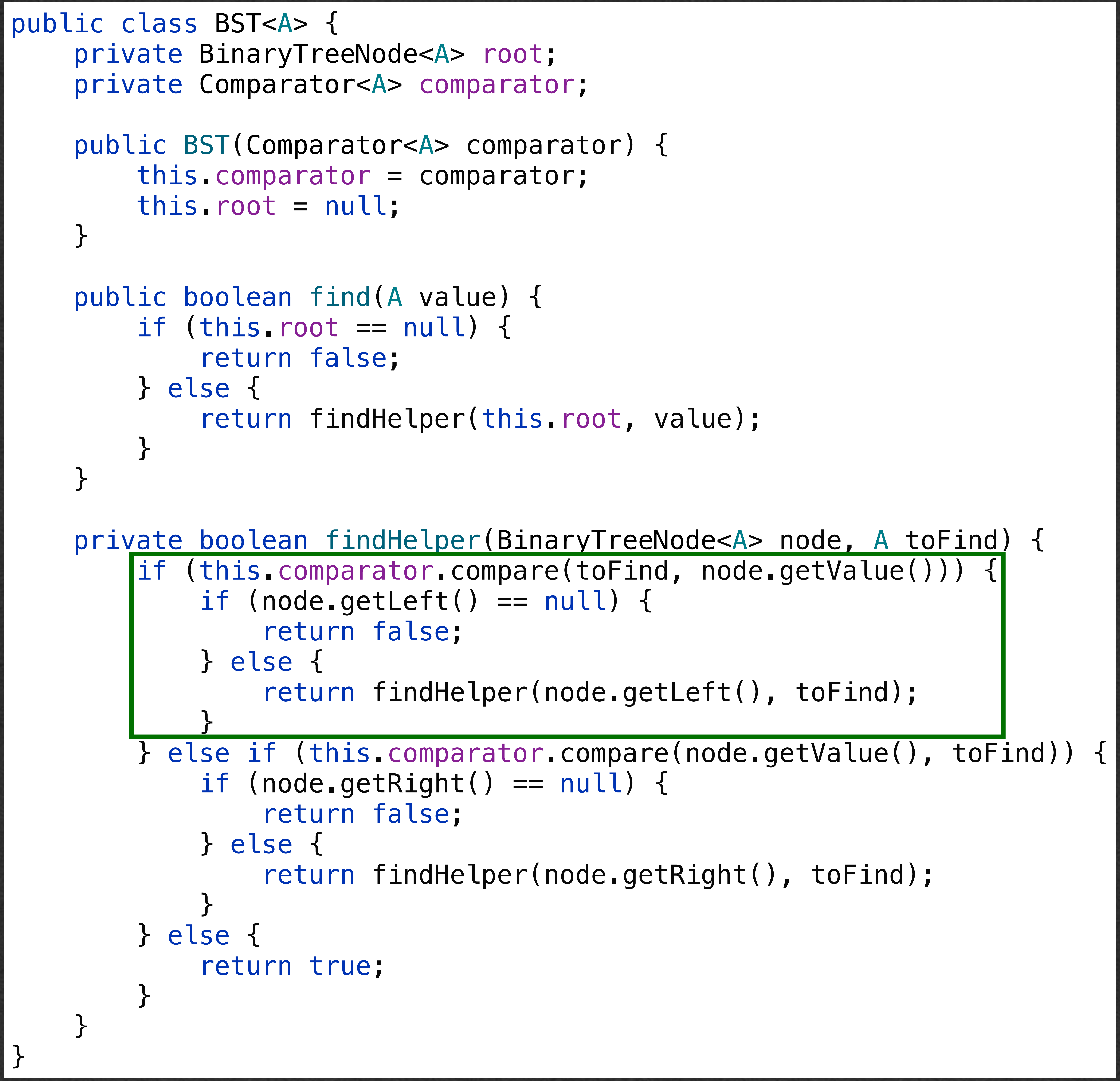
- If the BST is empty, return false
- Otherwise, start the recursion



```
public class BST<A> {
    private BinaryTreeNode<A> root;
    private Comparator<A> comparator;

    public BST(Comparator<A> comparator) {
        this.comparator = comparator;
        this.root = null;
    }

    public boolean find(A value) {
        if (this.root == null) {
            return false;
        } else {
            return findHelper(this.root, value);
        }
    }

    private boolean findHelper(BinaryTreeNode<A> node, A toFind) {
        if (this.comparator.compare(toFind, node.getValue())) {
            if (node.getLeft() == null) {
                return false;
            } else {
                return findHelper(node.getLeft(), toFind);
            }
        } else if (this.comparator.compare(node.getValue(), toFind)) {
            if (node.getRight() == null) {
                return false;
            } else {
                return findHelper(node.getRight(), toFind);
            }
        } else {
            return true;
        }
    }
}
```

BST - Find

- Compare the value to find with the value at the current node
- If this comparison is true:
    - The value we've looking for comes before the value at this node
    - Therefore, if the value to find is in this BST it MUST be in the left subtree
    - We eliminate the entire right subtree with one comparison



```java
public class BST<A> {
    private BinaryTreeNode<A> root;
    private Comparator<A> comparator;

    public BST(Comparator<A> comparator) {
        this.comparator = comparator;
        this.root = null;
    }

    public boolean find(A value) {
        if (this.root == null) {
            return false;
        } else {
            return findHelper(this.root, value);
        }
    }

    private boolean findHelper(BinaryTreeNode<A> node, A toFind) {
        if (this.comparator.compare(toFind, node.getValue())) {
            if (node.getLeft() == null) {
                return false;
            } else {
                return findHelper(node.getLeft(), toFind);
            }
        } else if (this.comparator.compare(node.getValue(), toFind)) {
            if (node.getRight() == null) {
                return false;
            } else {
                return findHelper(node.getRight(), toFind);
            }
        } else {
            return true;
        }
    }
}
```

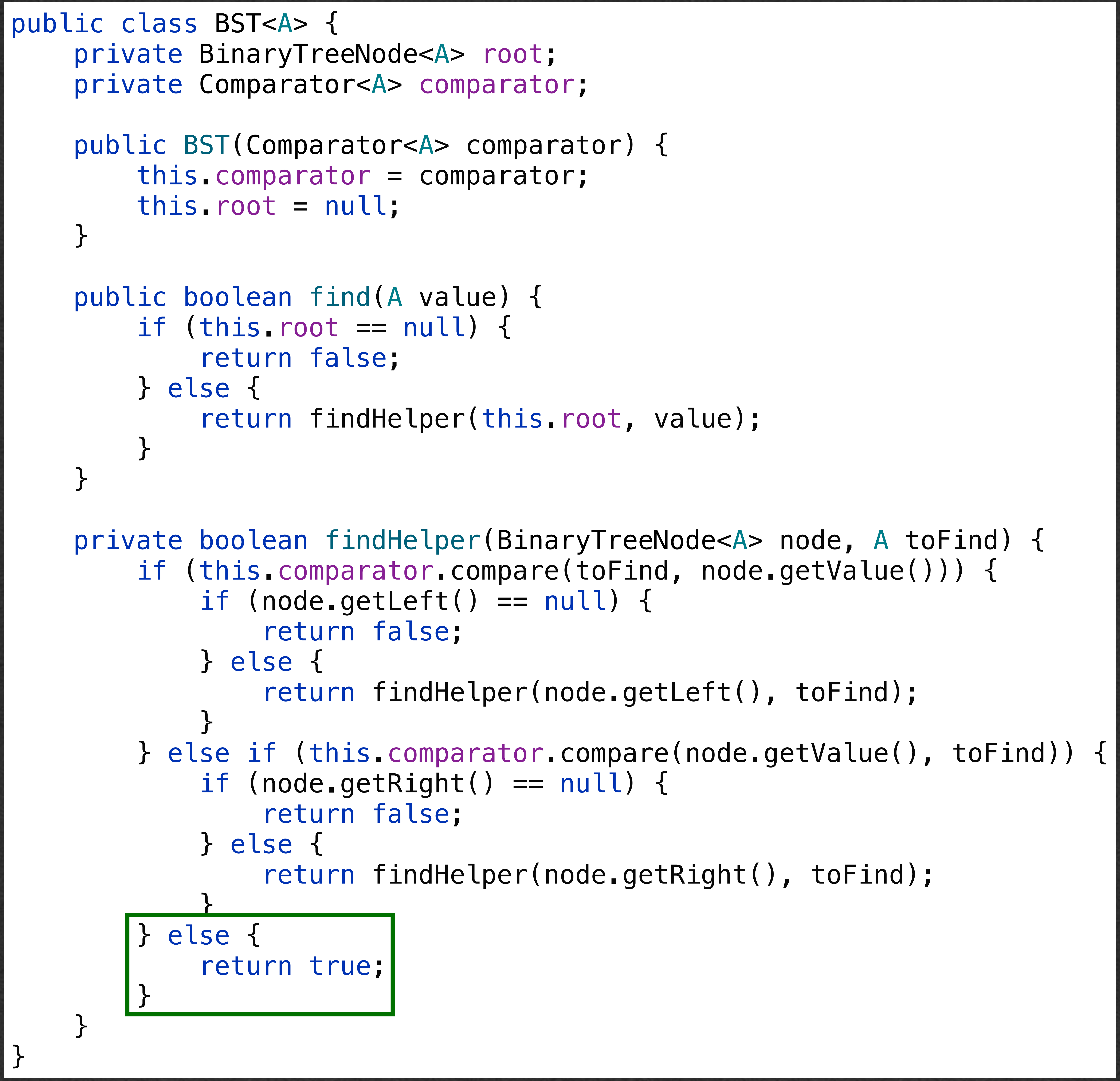
BST - Find

- If the first comparison is false, compare again in the opposite order
- If this comparison is true:
  - If the value we're looking for is in this BST, it MUST be in the right subtree
  - Eliminate the entire left subtree with one comparison

```java
public class BST<A> {
    private BinaryTreeNode<A> root;
    private Comparator<A> comparator;

    public BST(Comparator<A> comparator) {
        this.comparator = comparator;
        this.root = null;
    }

    public boolean find(A value) {
        if (this.root == null) {
            return false;
        } else {
            return findHelper(this.root, value);
        }
    }

    private boolean findHelper(BinaryTreeNode<A> node, A toFind) {
        if (this.comparator.compare(toFind, node.getValue())) {
            if (node.getLeft() == null) {
                return false;
            } else {
                return findHelper(node.getLeft(), toFind);
            }
        } else if (this.comparator.compare(node.getValue(), toFind)) {
            if (node.getRight() == null) {
                return false;
            } else {
                return findHelper(node.getRight(), toFind);
            }
        } else {
            return true;
        }
    }
}
```

BST - Find

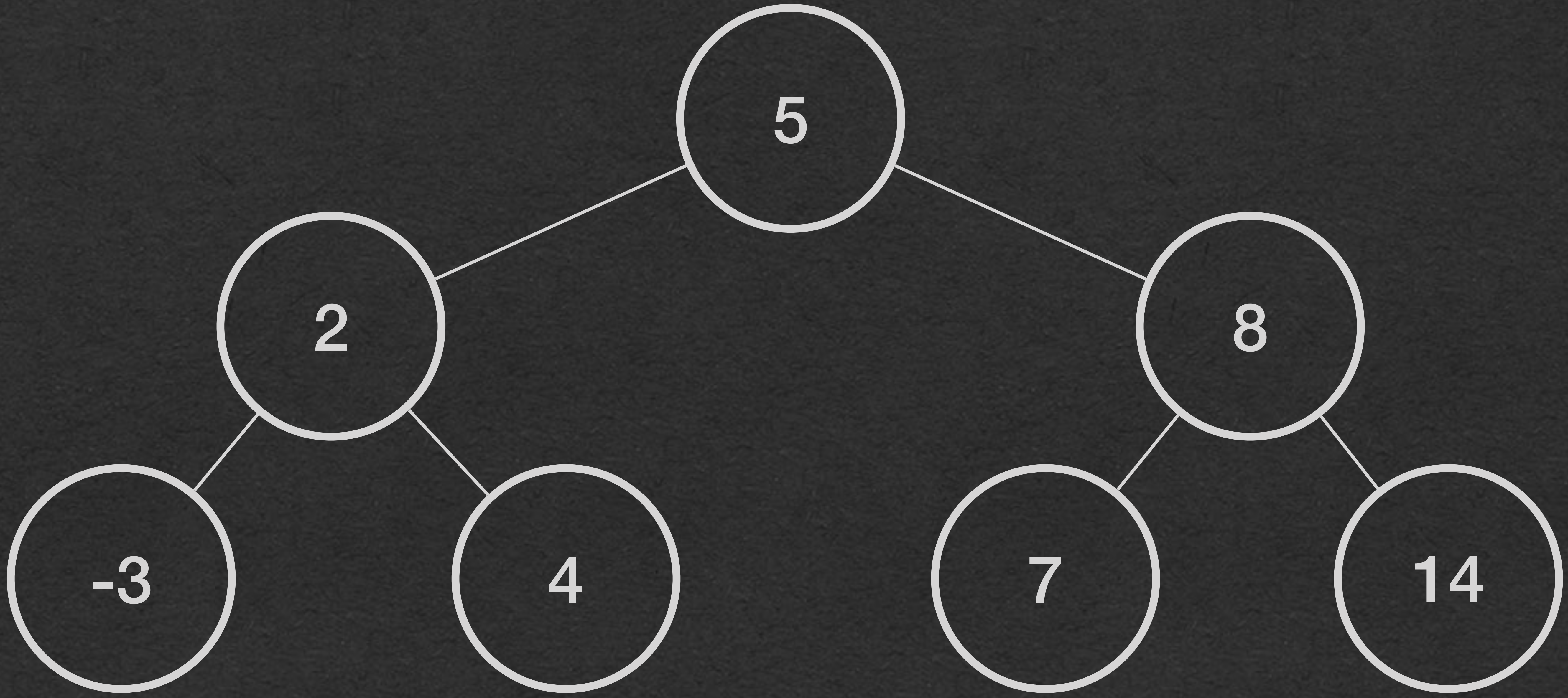
- If both comparisons return false:
  - The value we're looking for does not come before the value at this node
  - The value at this node does not come before the value to find
  - Therefore, these values are tied according to this comparator
    - We found the value!



```
public class BST<A> {
    private BinaryTreeNode<A> root;
    private Comparator<A> comparator;

    public BST(Comparator<A> comparator) {
        this.comparator = comparator;
        this.root = null;
    }

    public boolean find(A value) {
        if (this.root == null) {
            return false;
        } else {
            return findHelper(this.root, value);
        }
    }

    private boolean findHelper(BinaryTreeNode<A> node, A toFind) {
        if (this.comparator.compare(toFind, node.getValue())) {
            if (node.getLeft() == null) {
                return false;
            } else {
                return findHelper(node.getLeft(), toFind);
            }
        } else if (this.comparator.compare(node.getValue(), toFind)) {
            if (node.getRight() == null) {
                return false;
            } else {
                return findHelper(node.getRight(), toFind);
            }
        } else {
            return true;
        }
    }
}
```

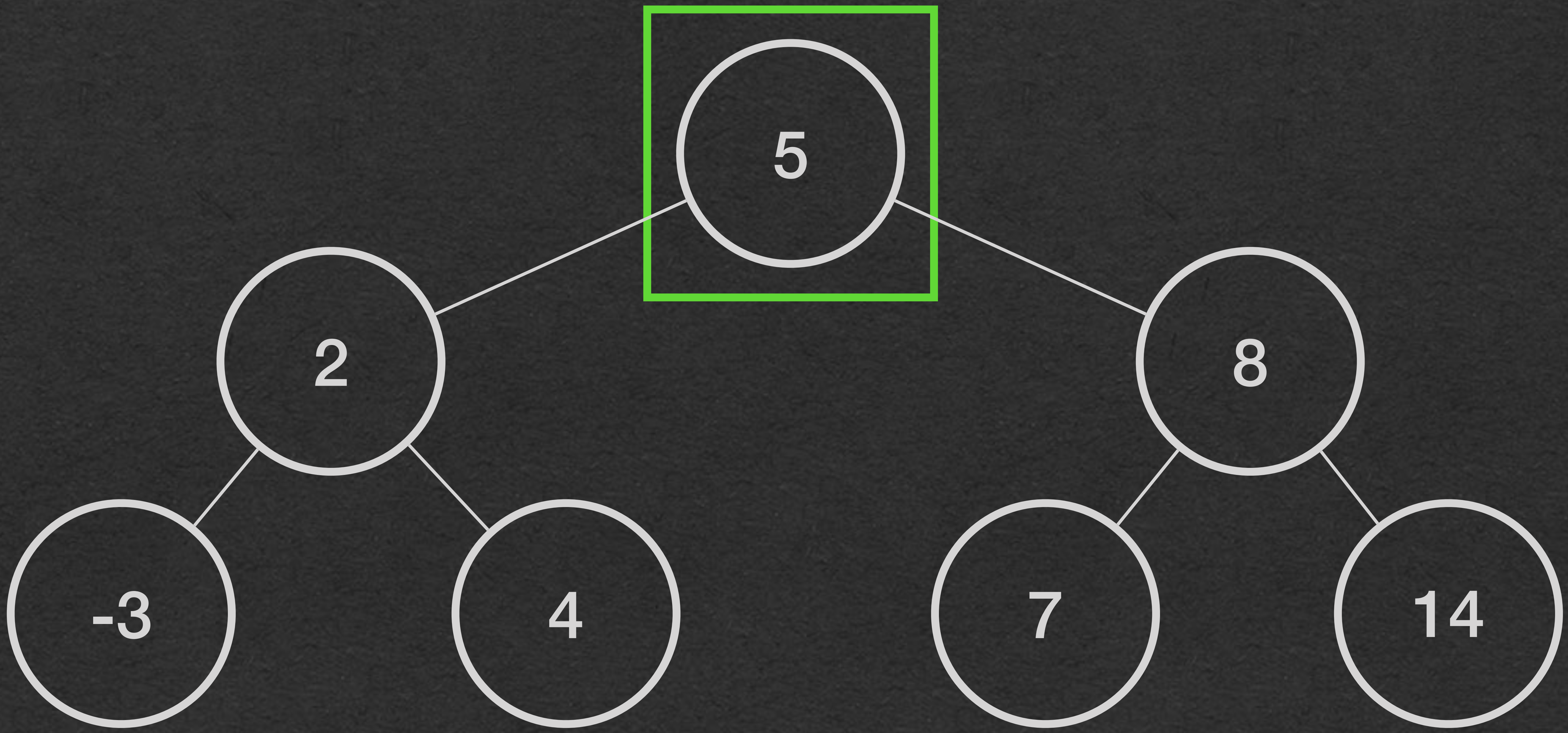
BST - Find

- Find the value 4



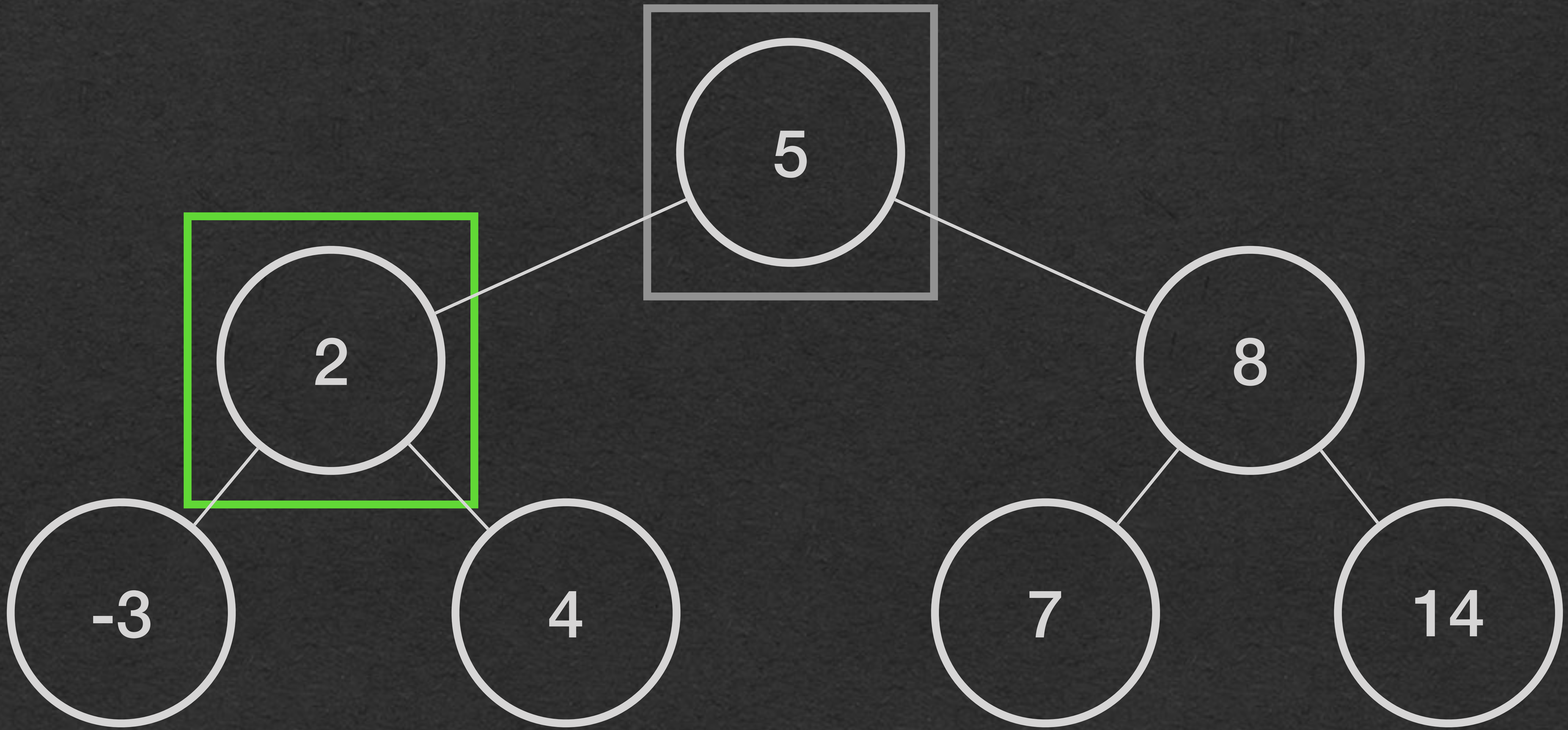
BST - Find

- Find the value 4
- 4 < 5 == true -> move left



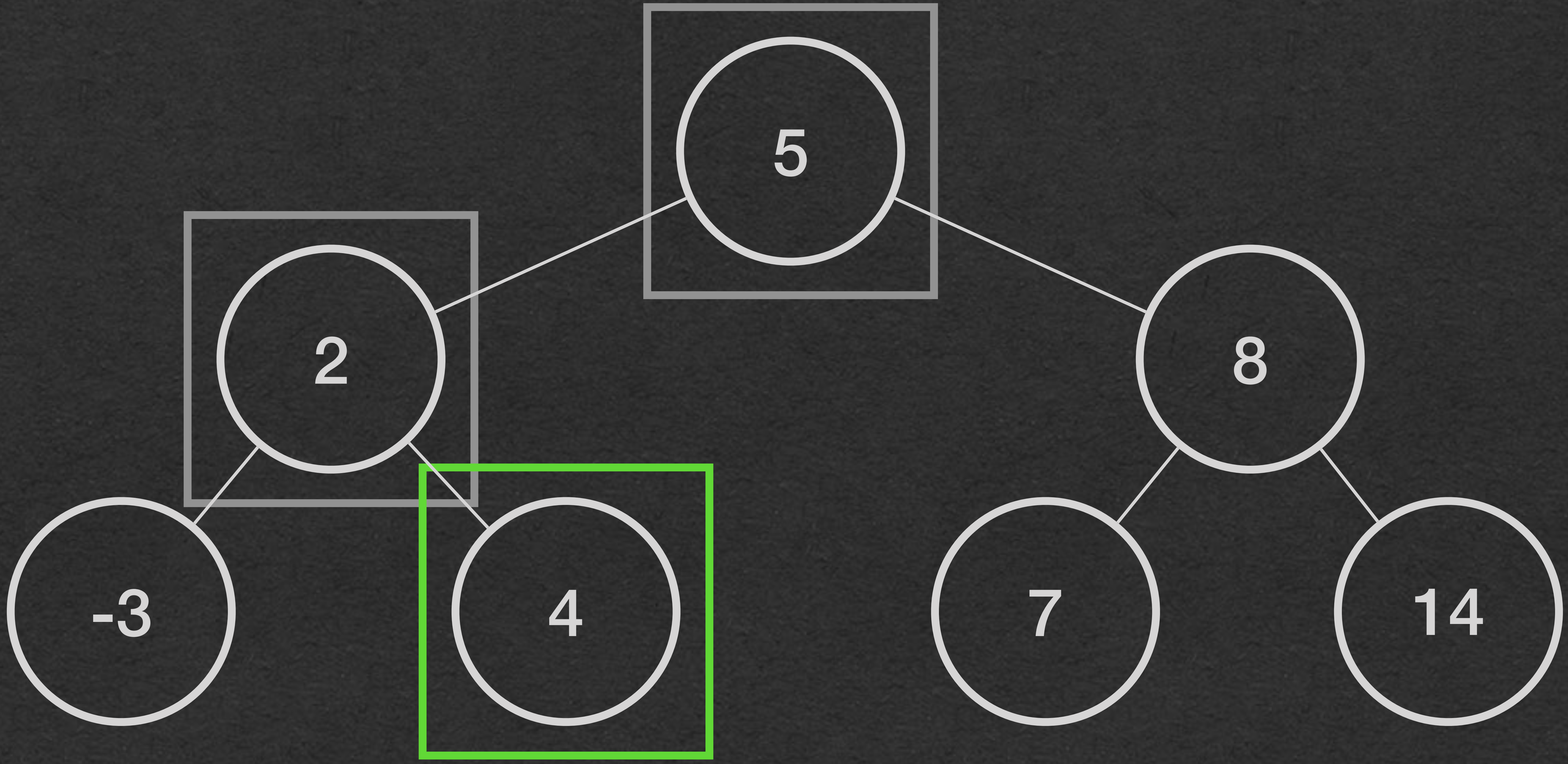
BST - Find

- Find the value 4
- 4 < 5 == true -> move left
- 4 < 2 == false -> 2 < 4 == true -> move right



BST - Find

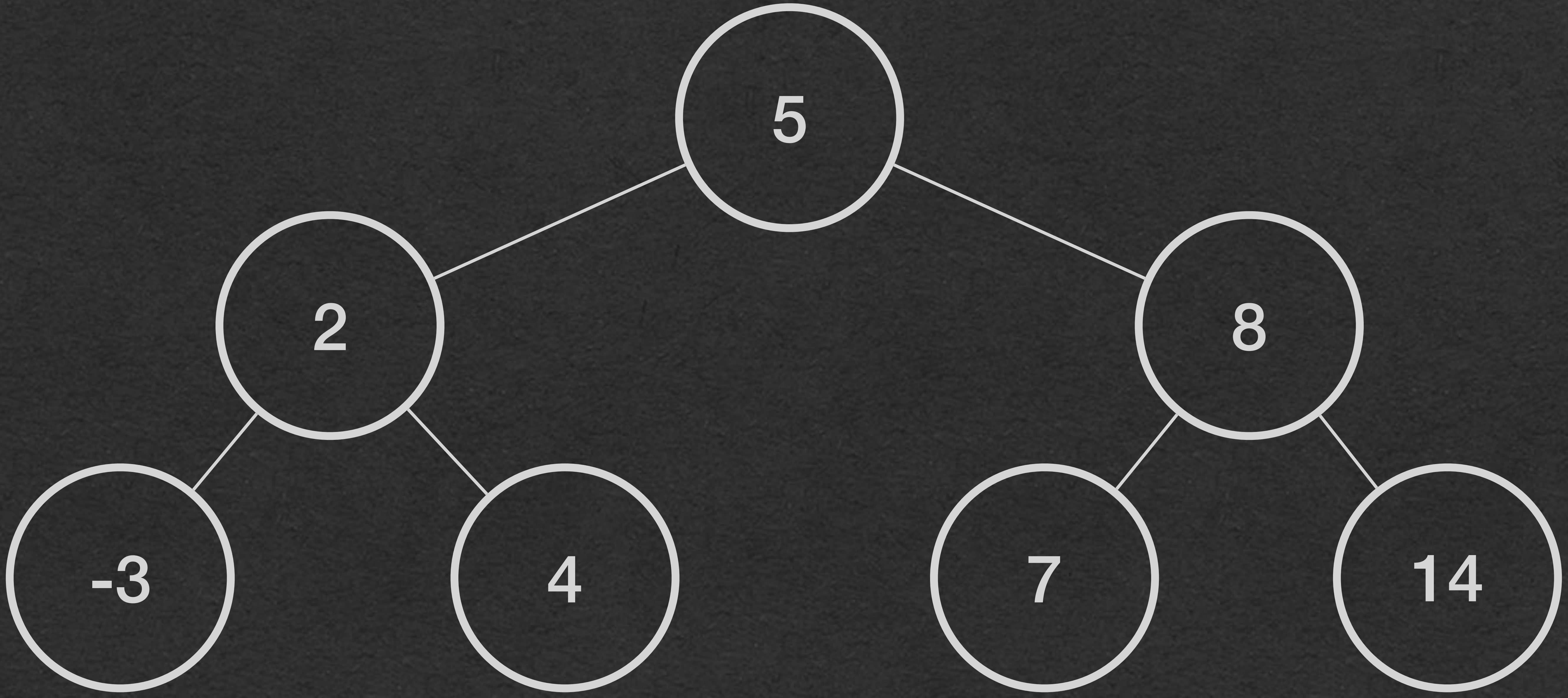
- Find the value 4
- 4 < 5 == true -> move left
- 4 < 2 == false -> 2 < 4 == true -> move right
- 4 < 4 == false -> 4 < 4 == false -> return true



In-Order Traversal

- **In-Order traversal of a BST iterates over the values in sorted order**
- Visit all elements of the left subtree
  - Elements that come before node's value
- Visit the node's value
- Visit all elements of the right subtree
  - Elements that come after the node's value

Printed:
-3
2
4
5
7
8
14



BST - Efficiency

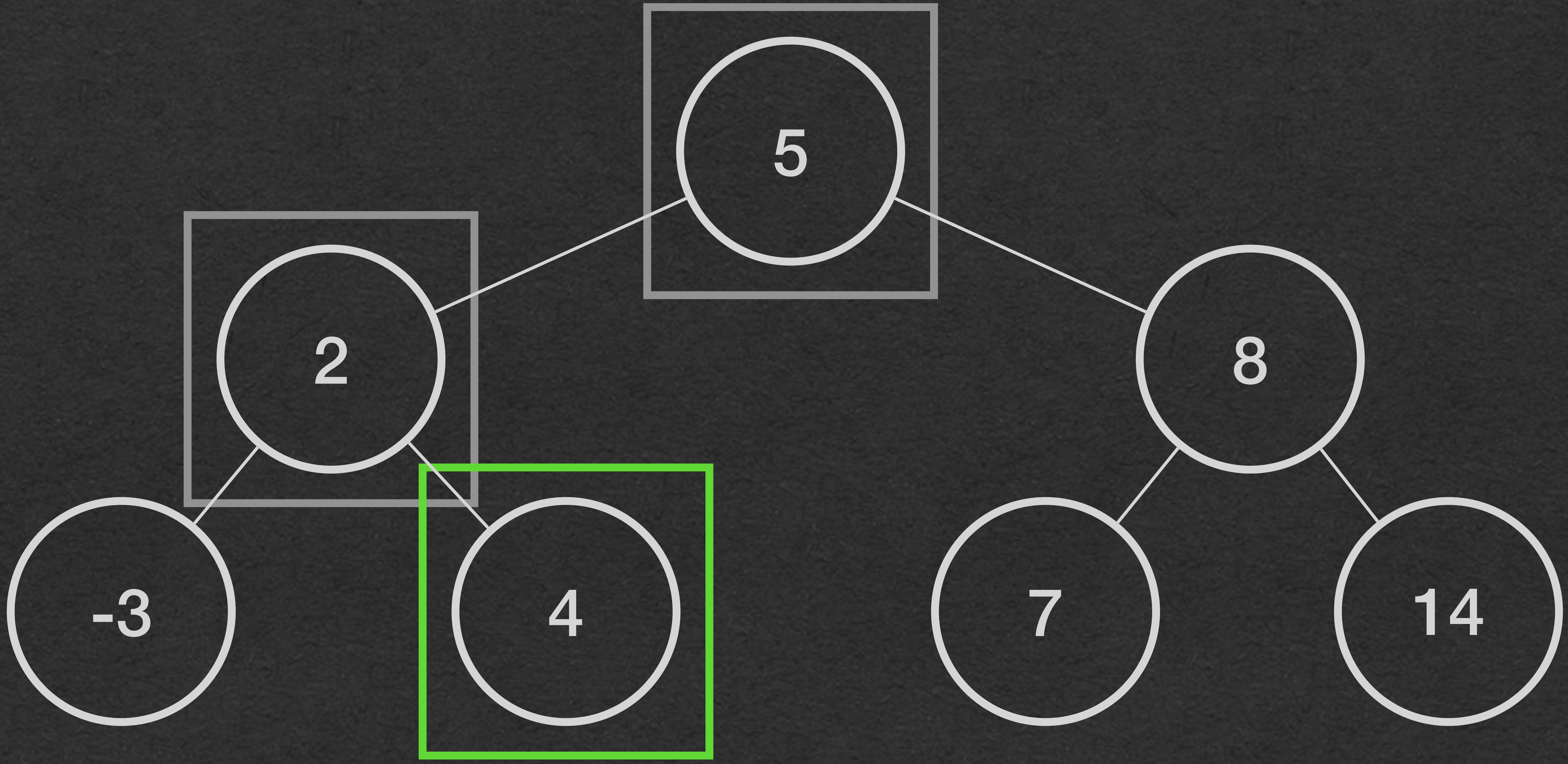
- Vocab: A tree is balanced if each node has the same number of descendants in its left and right subtrees
- The tree we used in today's example was balanced

BST - Efficiency

- *** If a BST is balanced ***
  - The number of nodes from the root to any null - the *height* of the tree - is O(log(n))
  - Insert and find take O(log(n)) time
  - Inserting n elements effectively sorts in O(n*log(n)) time
- Advantage: Sorted order is efficiently maintained as new elements are added in O(log(n))
  - Array takes O(n) to insert
  - Linked list takes O(n) to find where to insert

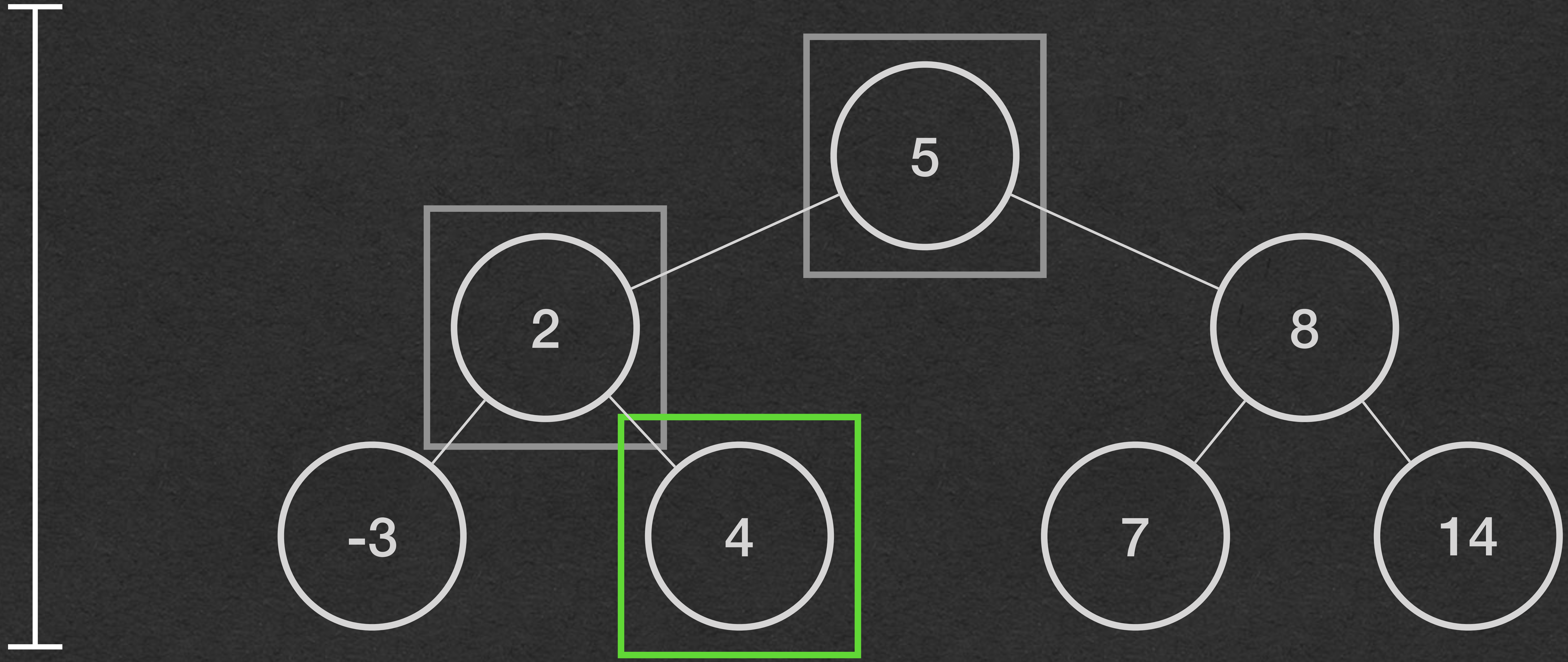
BST - Efficiency

- Notice that we checked very few nodes in our algorithms
- eg. In other data structures, we would have to check every node to find a value (Including Binary Trees that are not BST's)



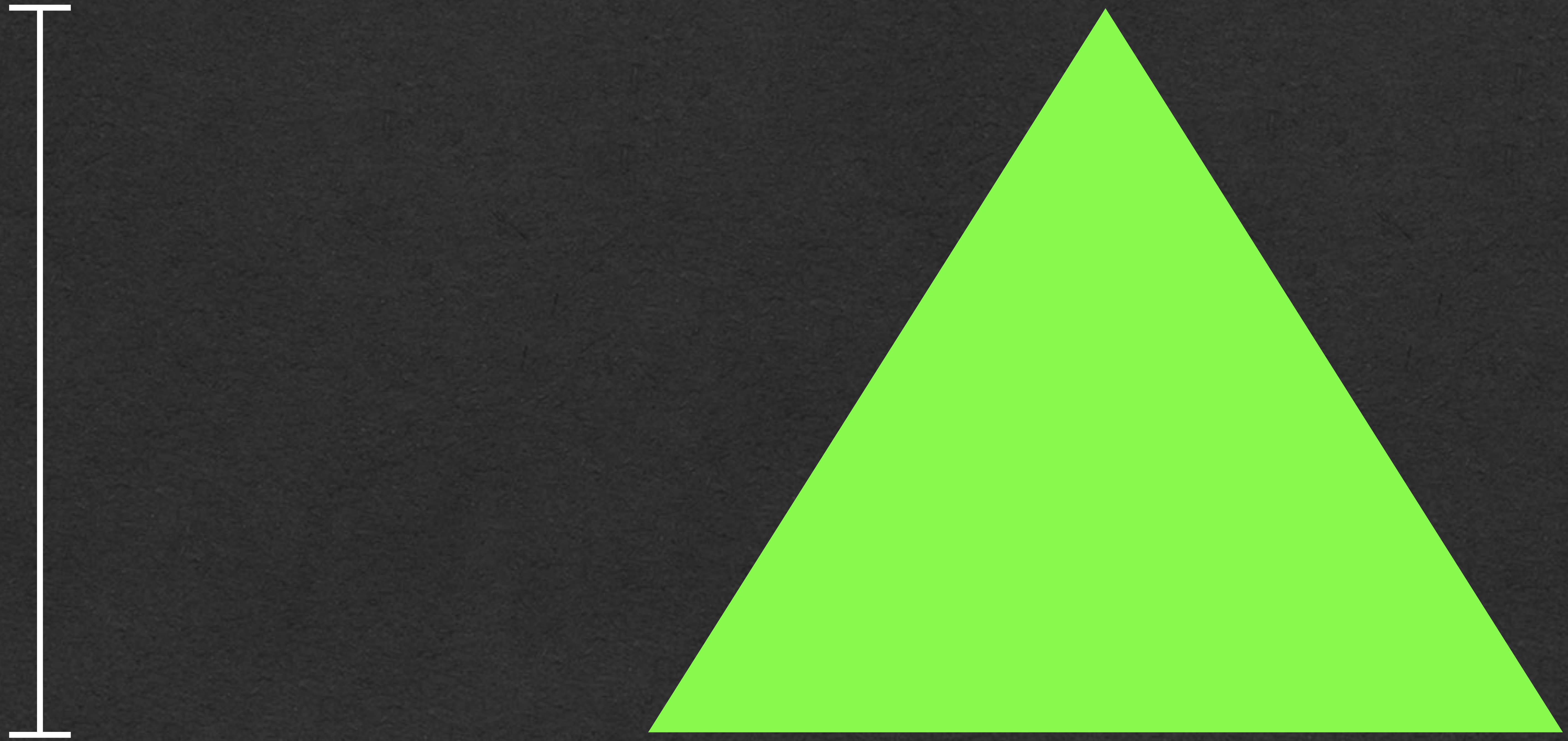
BST - Efficiency

- With BST's, we checked a number of nodes equal to the height of the tree
- This tree has height 3 and we checked 3 nodes instead of all 7



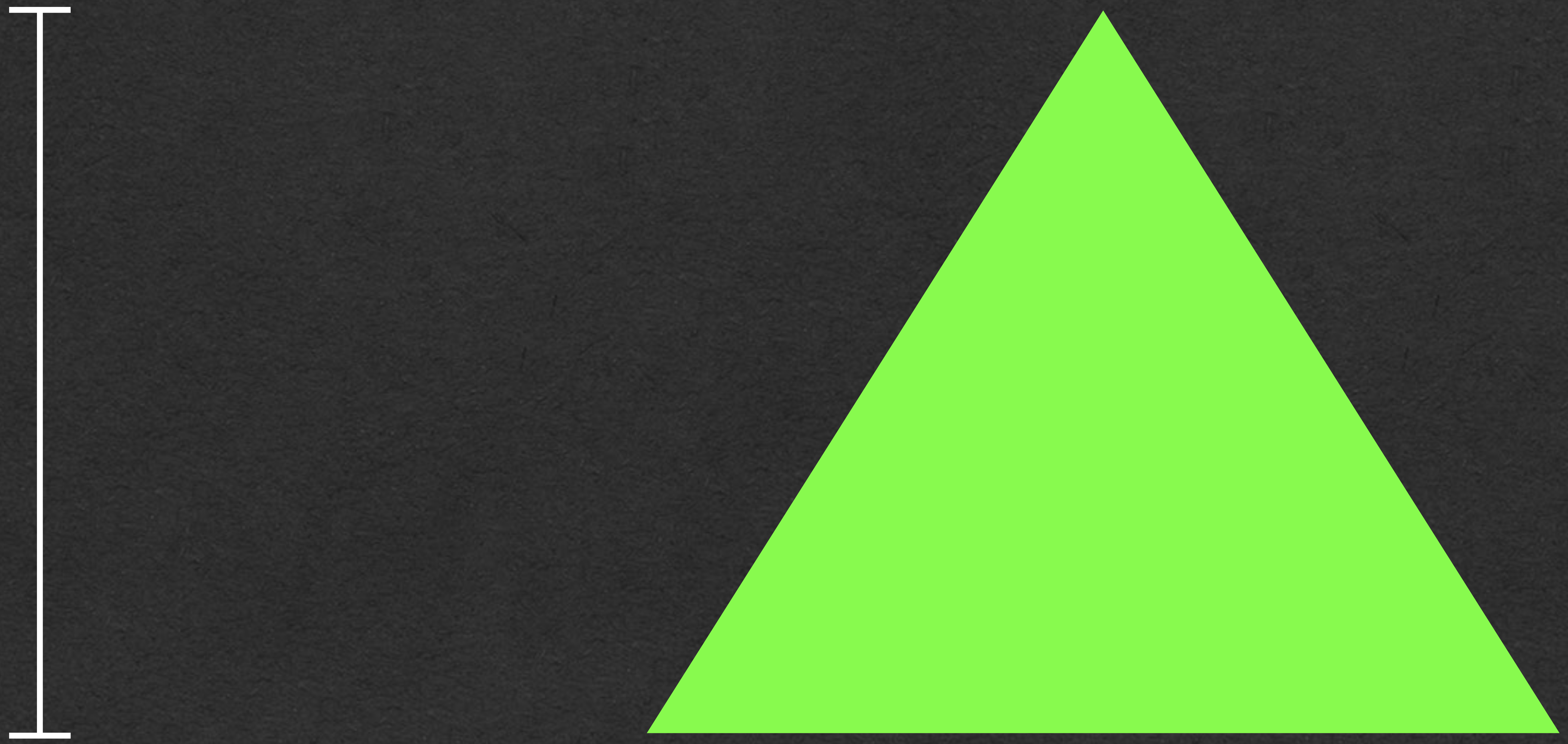
BST - Efficiency

- What's the height of a balanced BST with 1,000,000 nodes?



BST - Efficiency

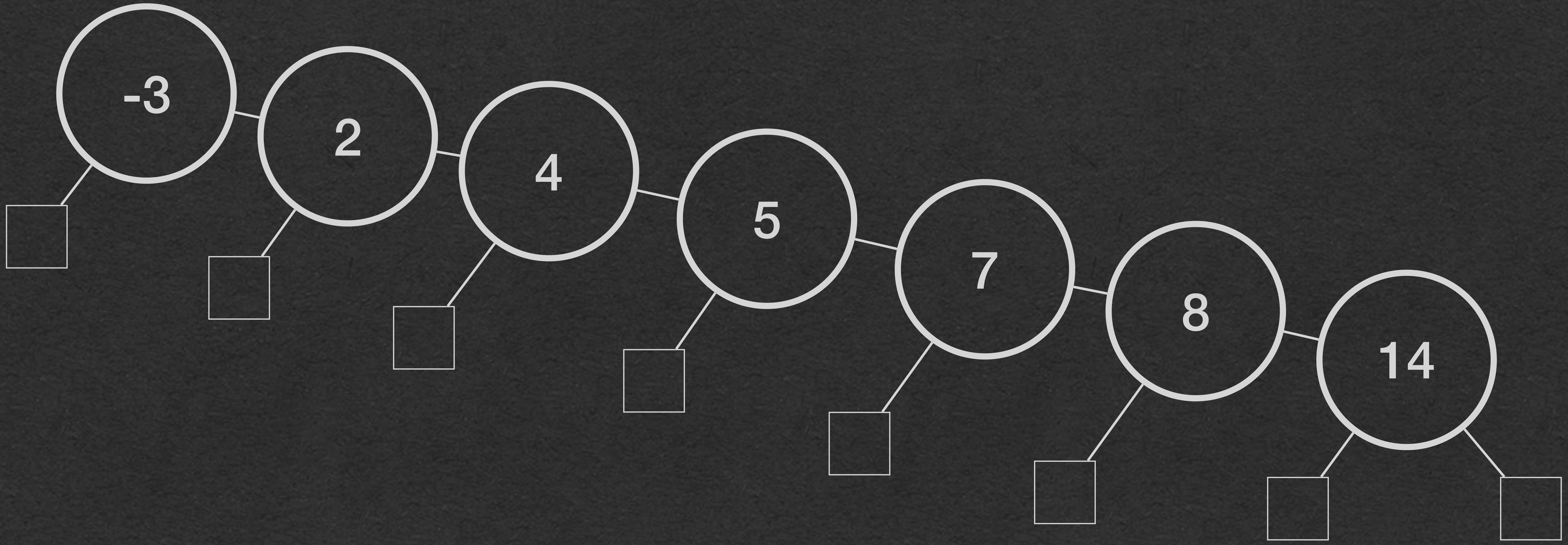
- What's the height of a balanced BST with 1,000,000 nodes?
- log_2(1,000,000) ≈ 20
- Only check 20 nodes instead of 1,000,000



BST - Inefficiency

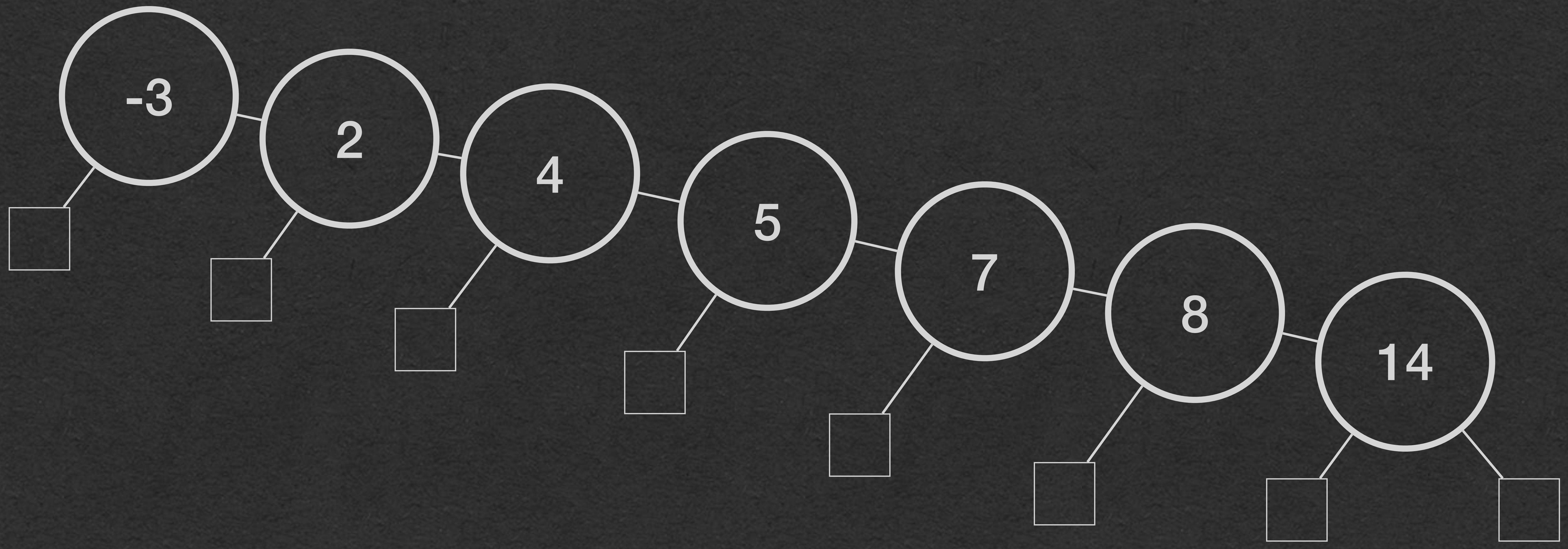
- What if the tree is not balanced?

```
BST<Integer> bst = new BST<>(new IntIncreasing());

bst.insert(-3);
bst.insert(2);
bst.insert(4);
bst.insert(5);
bst.insert(7);
bst.insert(8);
bst.insert(14);
```



BST - Inefficiency

- If elements are inserted in sorted order
- Tree effectively becomes a linked list
  - O(n) insert and find



BST for Thought

- How do we keep the tree balanced and still insert in O(log(n)) time
- How would we remove a node while maintaining sorted order?
- How do we handle duplicate values?
  - Should duplicates even be allowed?

- Answers to these questions and more..
  - In **CSE250**