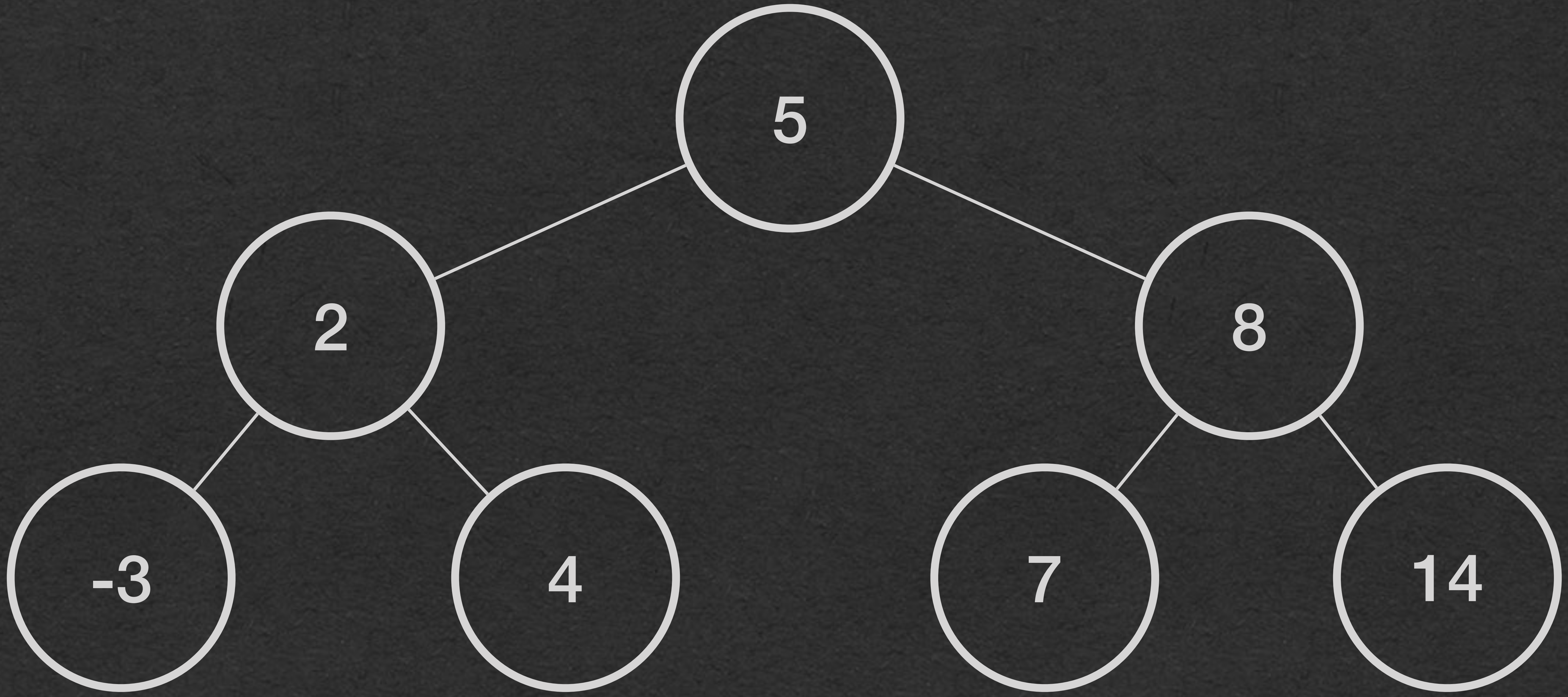
Binary Search Tree
(BST)

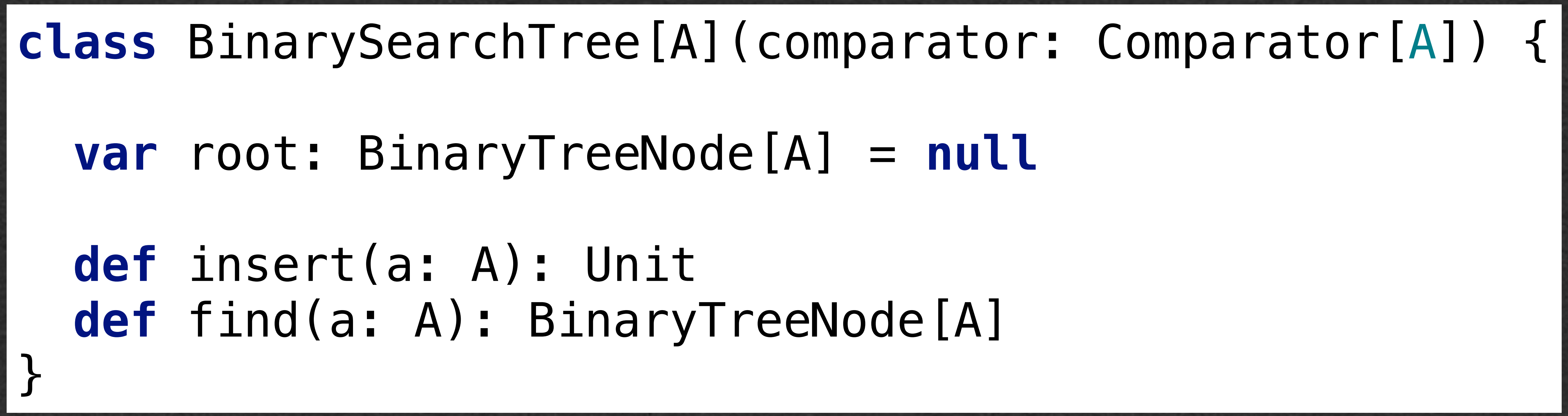
BST - Definition

- For each node:
  - All values in the left subtree are less than the node's value
  - All values in the right subtree are greater than the node's value
  - Duplicate values handled differently based on implementation
    - Sometimes not allowed at all



BST - Code

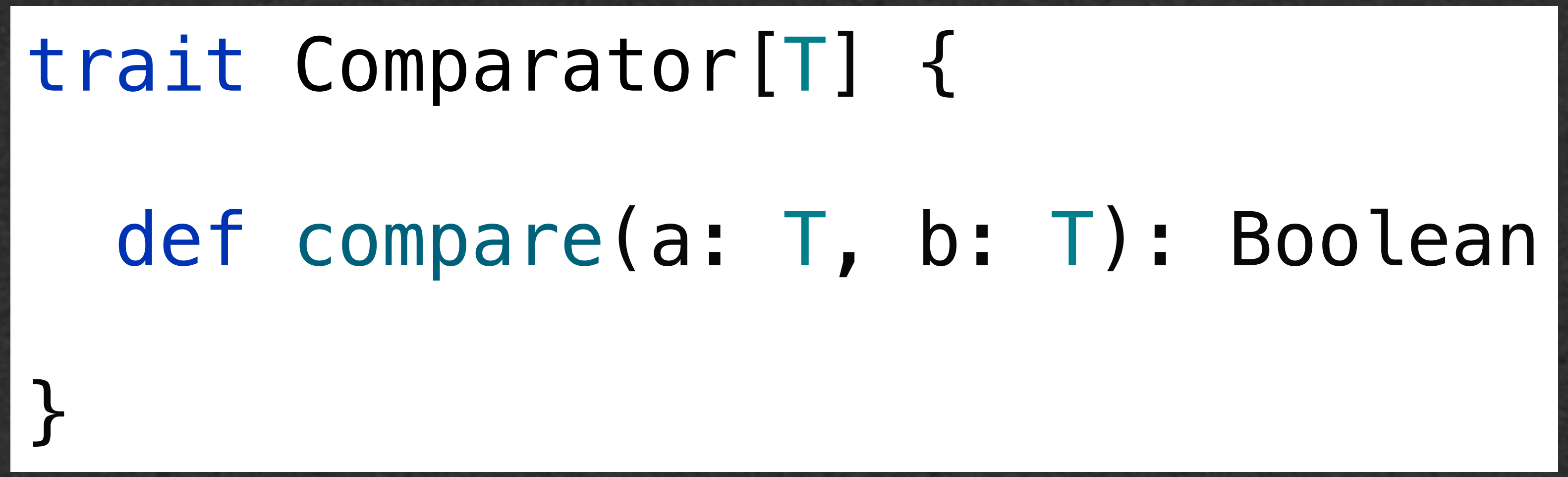
- To make the BST generic
  - Take a type parameter
  - Take a Comparator to decide the sorted order
    - [We define Comparator on the next slide]
- Store a reference to the root node



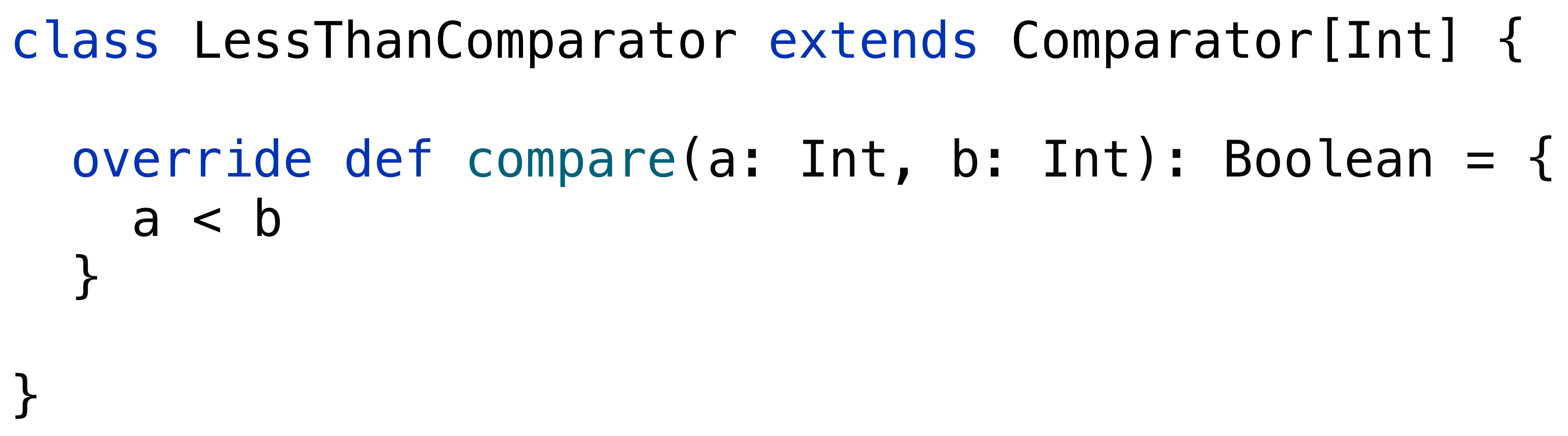
```
class BinarySearchTree[A](comparator: Comparator[A]) {

  var root: BinaryTreeNode[A] = null

  def insert(a: A): Unit
  def find(a: A): BinaryTreeNode[A]
}
```

BST - Code

- Use OOP to create a Comparator
- A trait is like an abstract class, but it cannot have constructor parameters
- Extend the trait and implement compare to determine the sorted order
  - Returns true if a should come before b, false otherwise



```scala
trait Comparator[T] {

  def compare(a: T, b: T): Boolean

}
```



```scala
class LessThanComparator extends Comparator[Int] {

  override def compare(a: Int, b: Int): Boolean = {
    a < b
  }

}
```

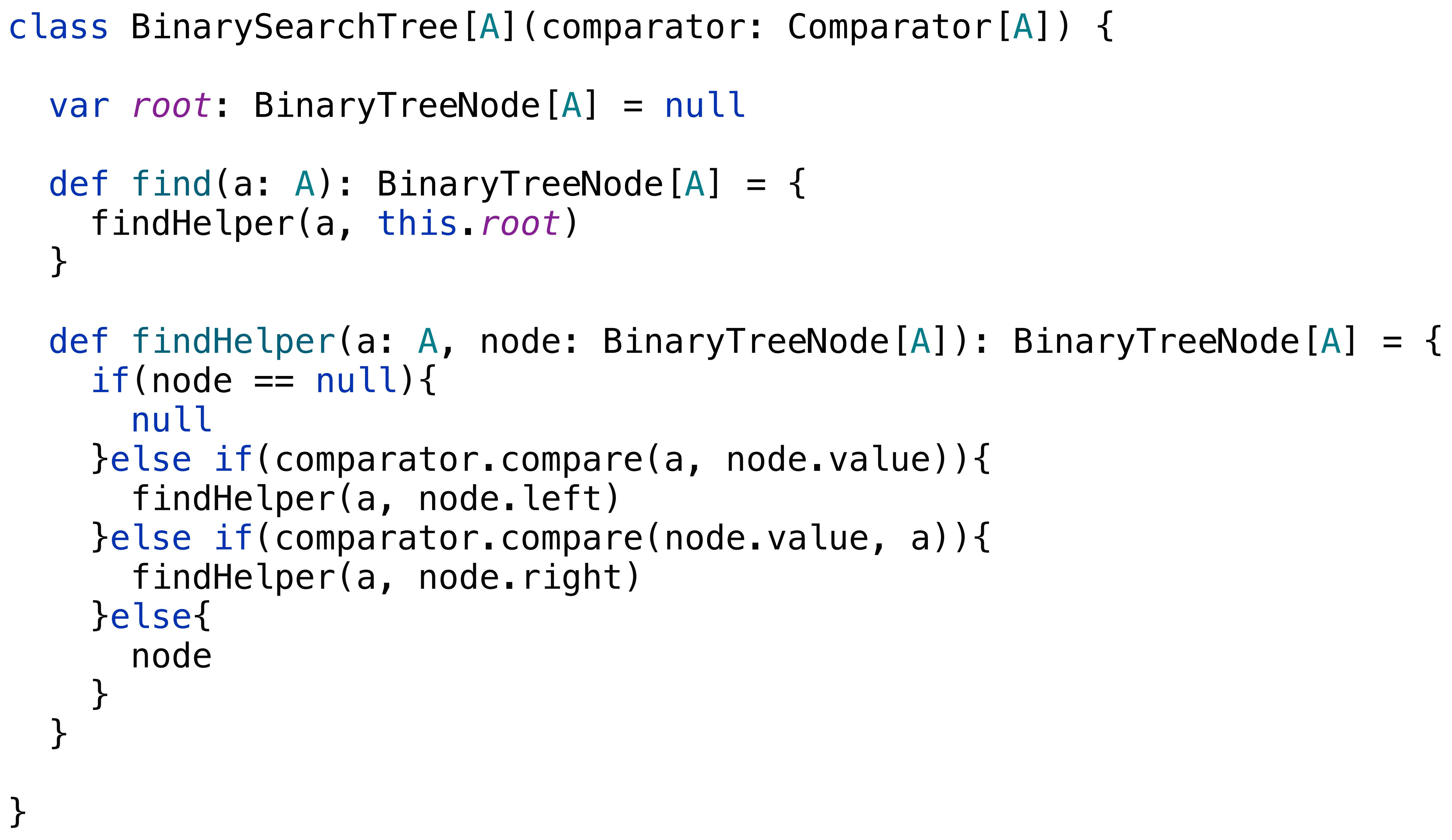
BST - Usage

```scala
class BinarySearchTree[A](comparator: Comparator[A]) {

  var root: BinaryTreeNode[A] = null

  def insert(a: A): Unit
  def find(a: A): BinaryTreeNode[A]
}
```

```scala
val bst = new BinarySearchTree[Int](new LessThanComparator())
bst.insert(5)
bst.insert(2)
bst.insert(8)
bst.insert(4)
bst.insert(7)
bst.insert(14)
bst.insert(-3)

val node = bst.find(4)
```

BST - Find

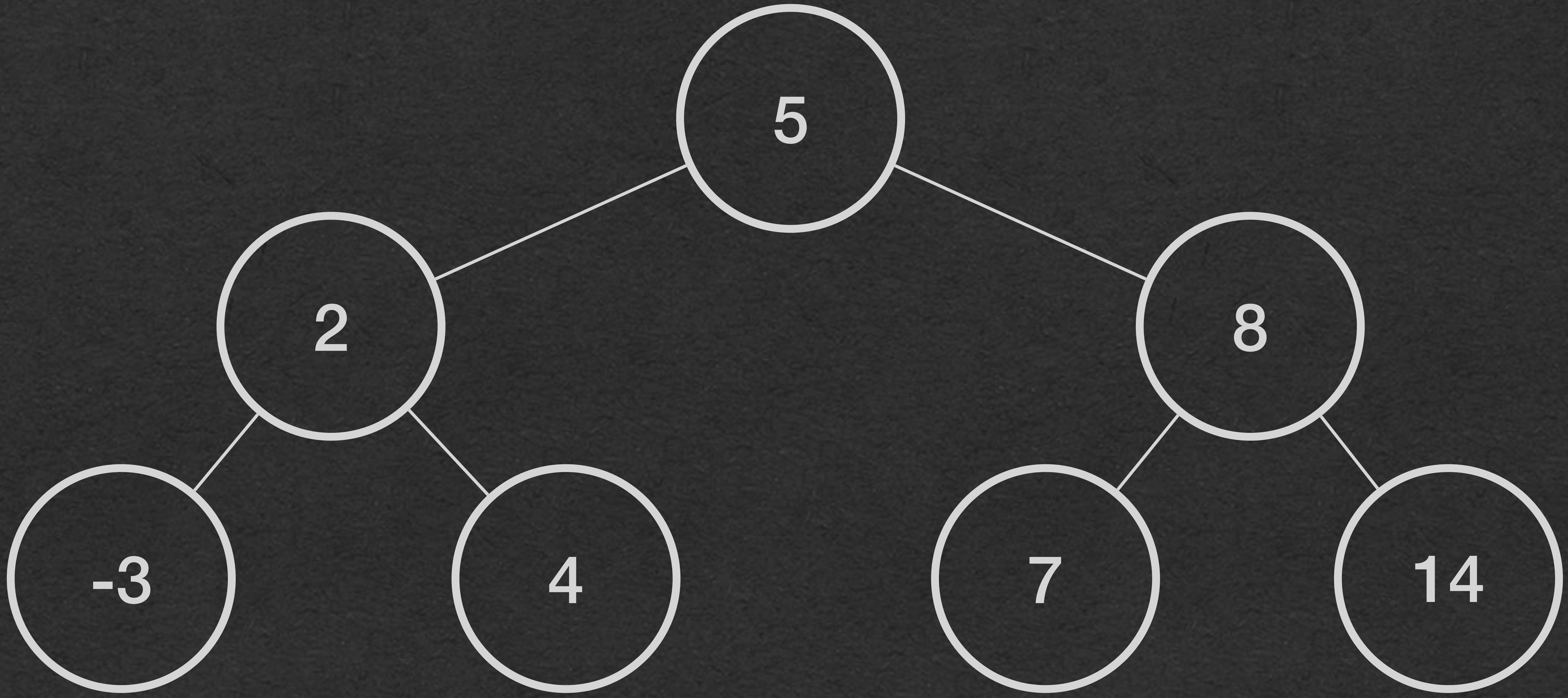
- If the value to find is less than the value of the node - Move to the left child
- If the value to find is greater than the value of the node - Move to right child
- If value is found - return this node
- If value is not found - return null



```
class BinarySearchTree[A](comparator: Comparator[A]) {

  var root: BinaryTreeNode[A] = null

  def find(a: A): BinaryTreeNode[A] = {
    findHelper(a, this.root)
  }

  def findHelper(a: A, node: BinaryTreeNode[A]): BinaryTreeNode[A] = {
    if(node == null){
      null
    }else if(comparator.compare(a, node.value)){
      findHelper(a, node.left)
    }else if(comparator.compare(node.value, a)){
      findHelper(a, node.right)
    }else{
      node
    }
  }

}
```

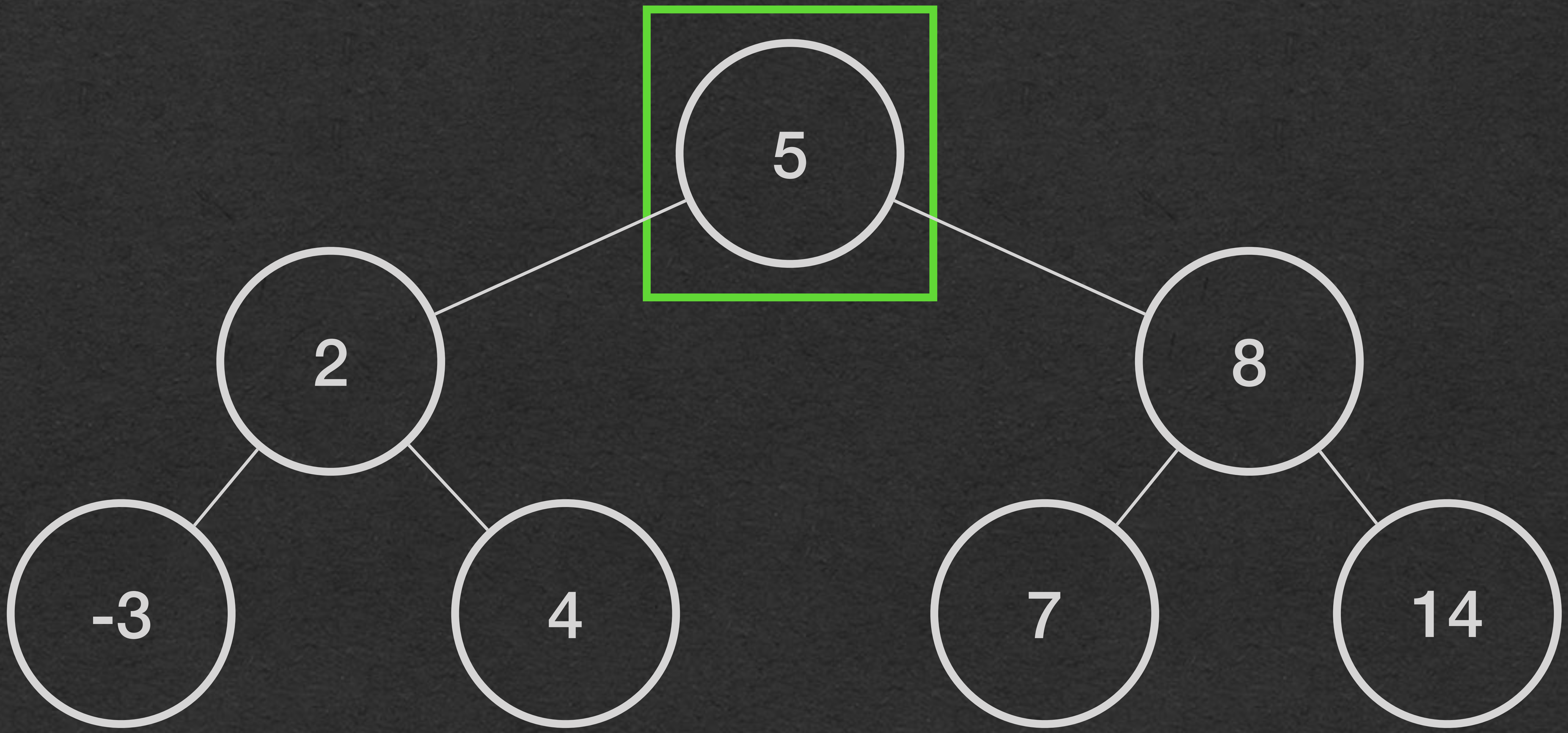
BST - Find

- Find the value 4



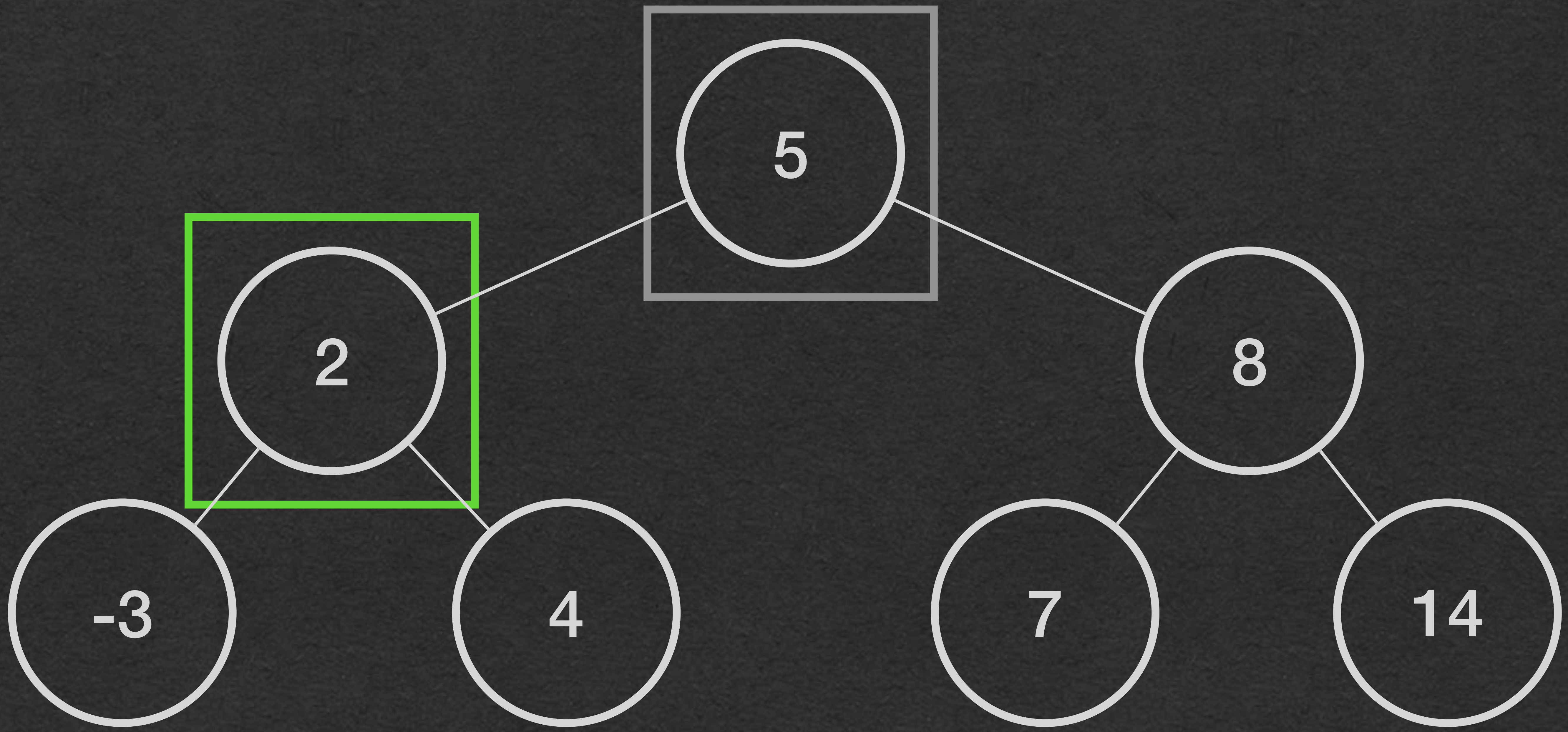
BST - Find

- Find the value 4
- 4 < 5



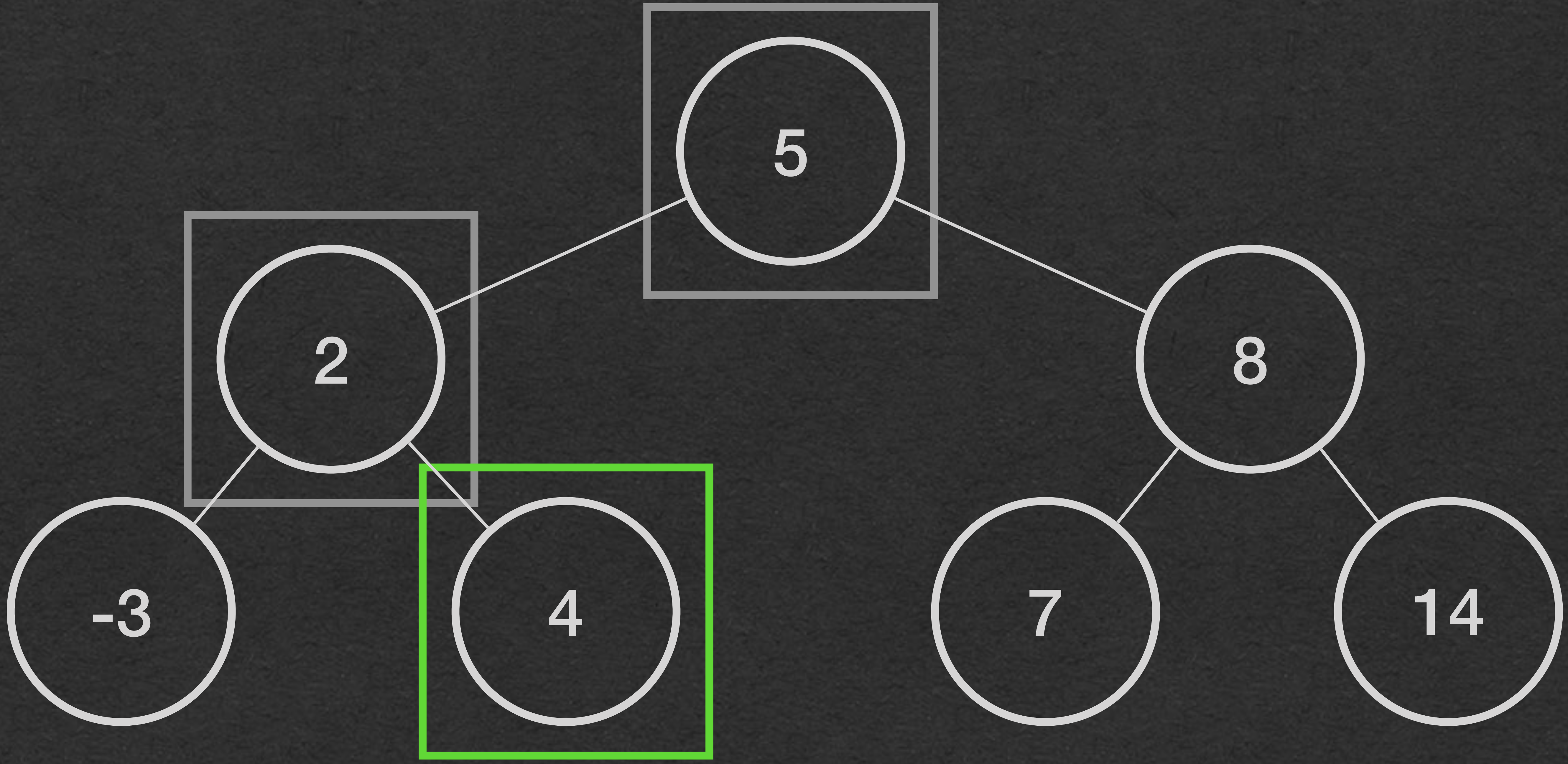
BST - Find

- Find the value 4
- 4 < 5
- 4 > 2



BST - Find

- Find the value 4
- 4 < 5
- 2 < 4
- 4 == 4 - return this node



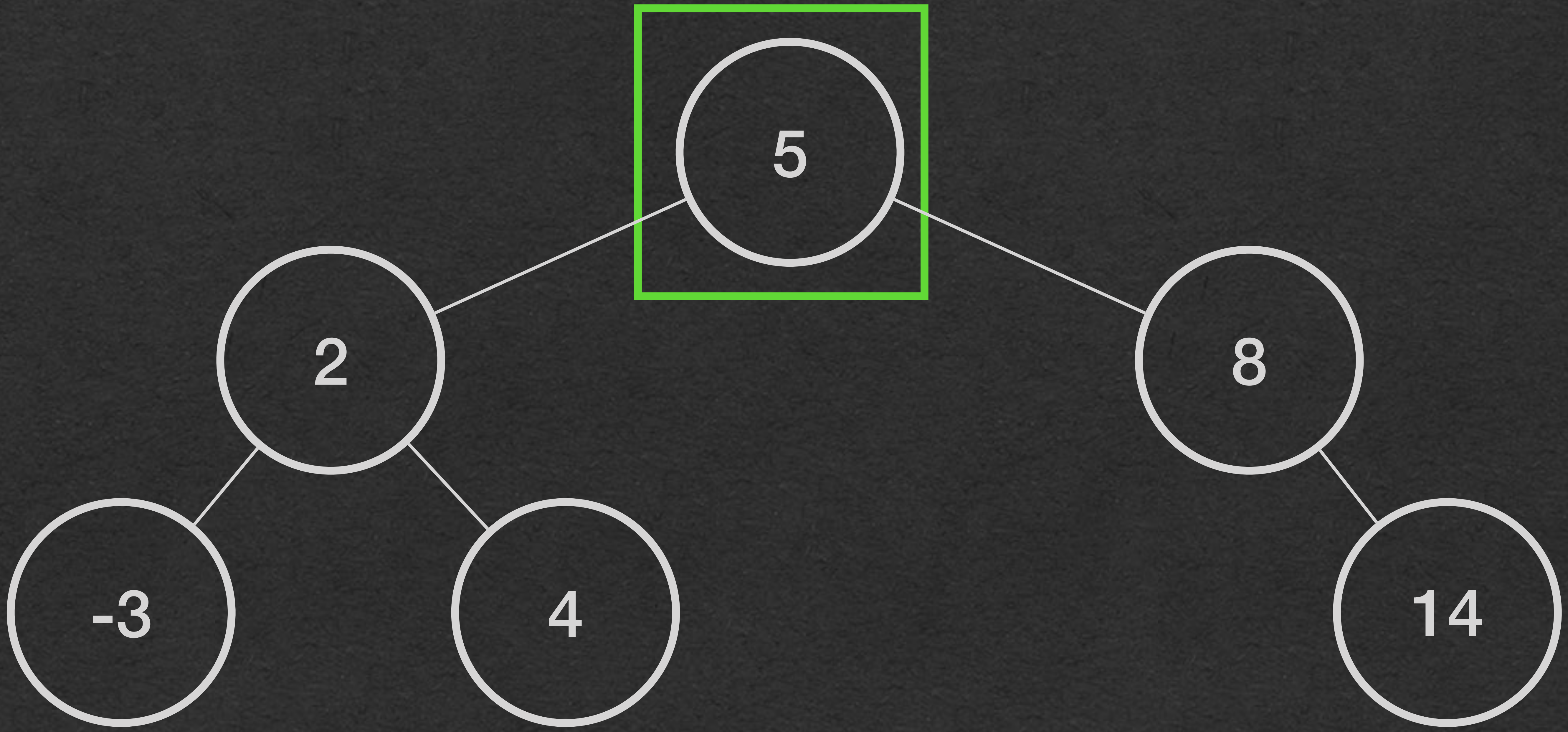
BST - Insert

- Run find until a null node is reached - insert new node here
- If value is a duplicate, move to the left (In this implementation)

```
class BinarySearchTree[A](comparator: Comparator[A]) {

  var root: BinaryTreeNode[A] = null

  def insert(a: A): Unit = {
    if(this.root == null){
      this.root = new BinaryTreeNode(a, null, null)
    }else{
      insertHelper(a, this.root)
    }
  }

  def insertHelper(a: A, node: BinaryTreeNode[A]): Unit = {
    if(comparator.compare(node.value, a)){
      if(node.right == null){
        node.right = new BinaryTreeNode[A](a, null, null)
      }else{
        insertHelper(a, node.right)
      }
    }else{
      if(node.left == null){
        node.left = new BinaryTreeNode[A](a, null, null)
      }else{
        insertHelper(a, node.left)
      }
    }
  }

}
```

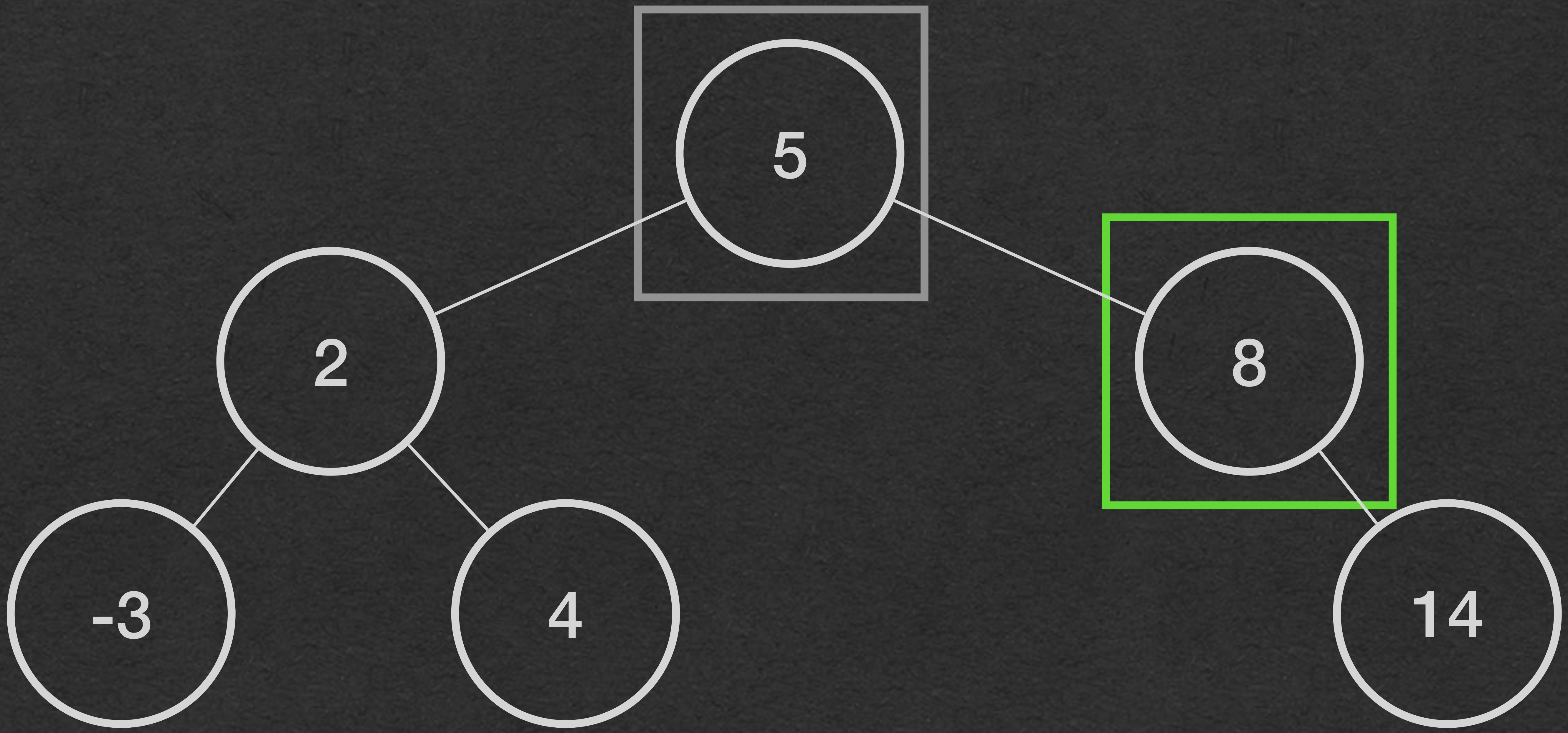
BST - Insert

- Insert 7



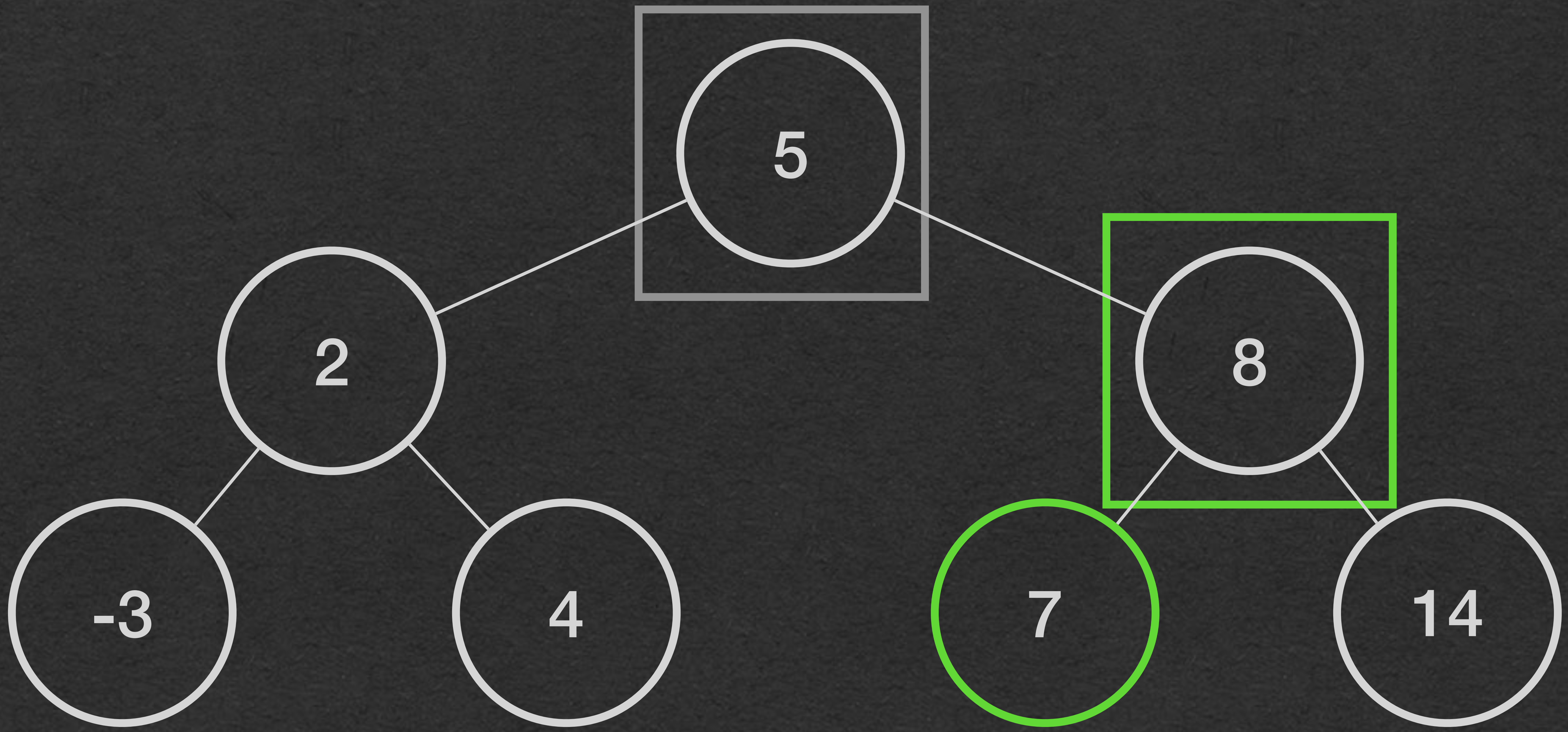
BST - Insert

- Insert 7
- 5 < 7



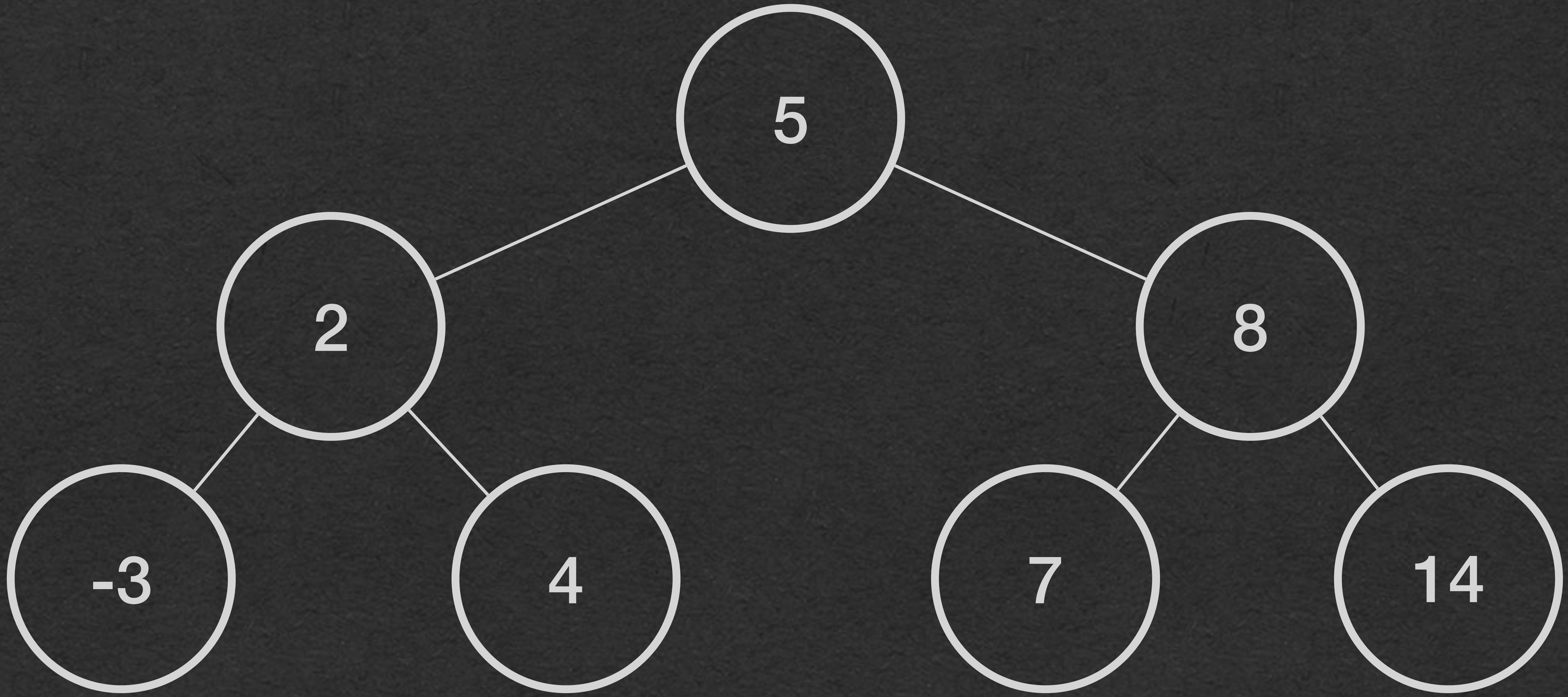
BST - Insert

- Insert 7
- 5 < 7
- 7 < 8 and left child is null - Insert here



BST - Insert

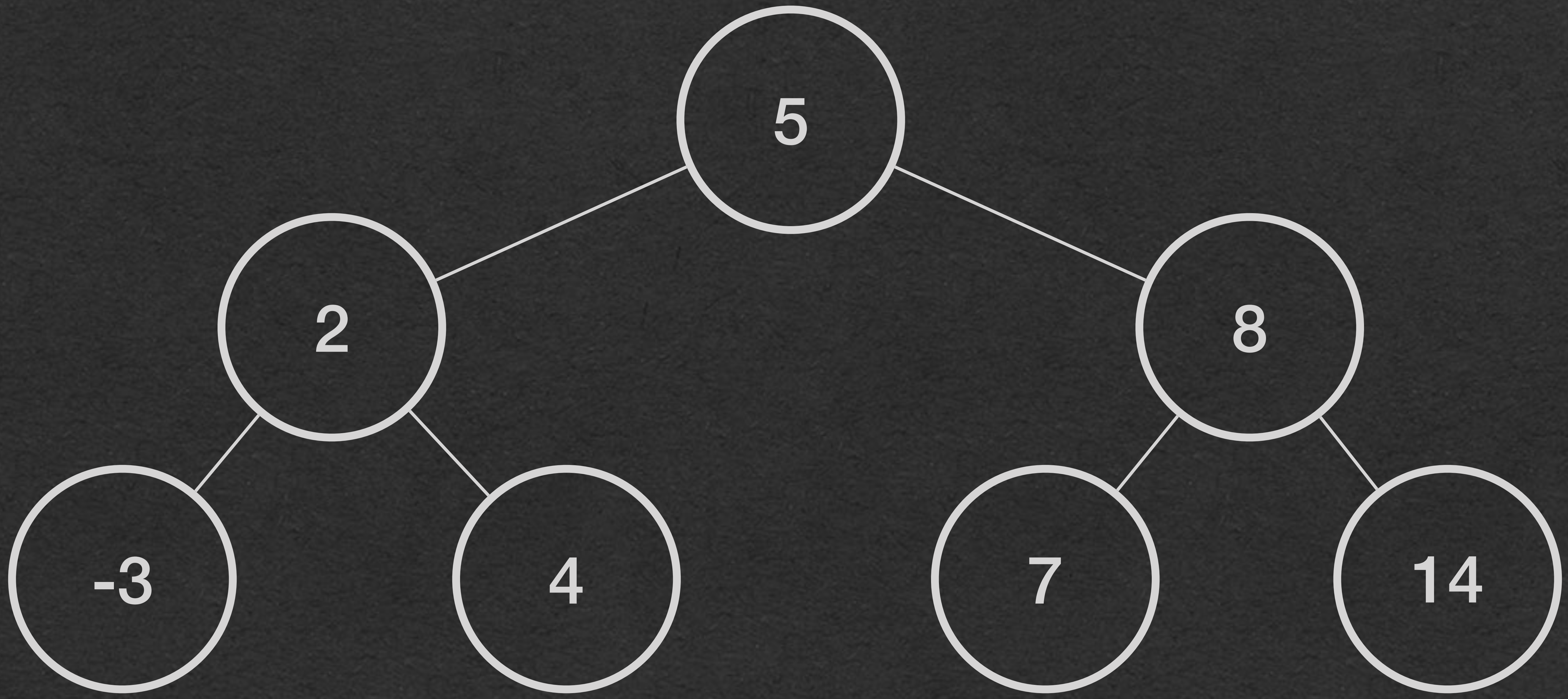
- Recursive calls return



In-Order Traversal

- In-Order traversal of a BST iterates over the values in sorted order
- Visit all elements of the left subtree
  - Elements less that the node's value
- Visit the nodes value
- Visit all elements of the right subtree
  - Elements greater than the node's value

Printed:
-3
2
4
5
7
8
14



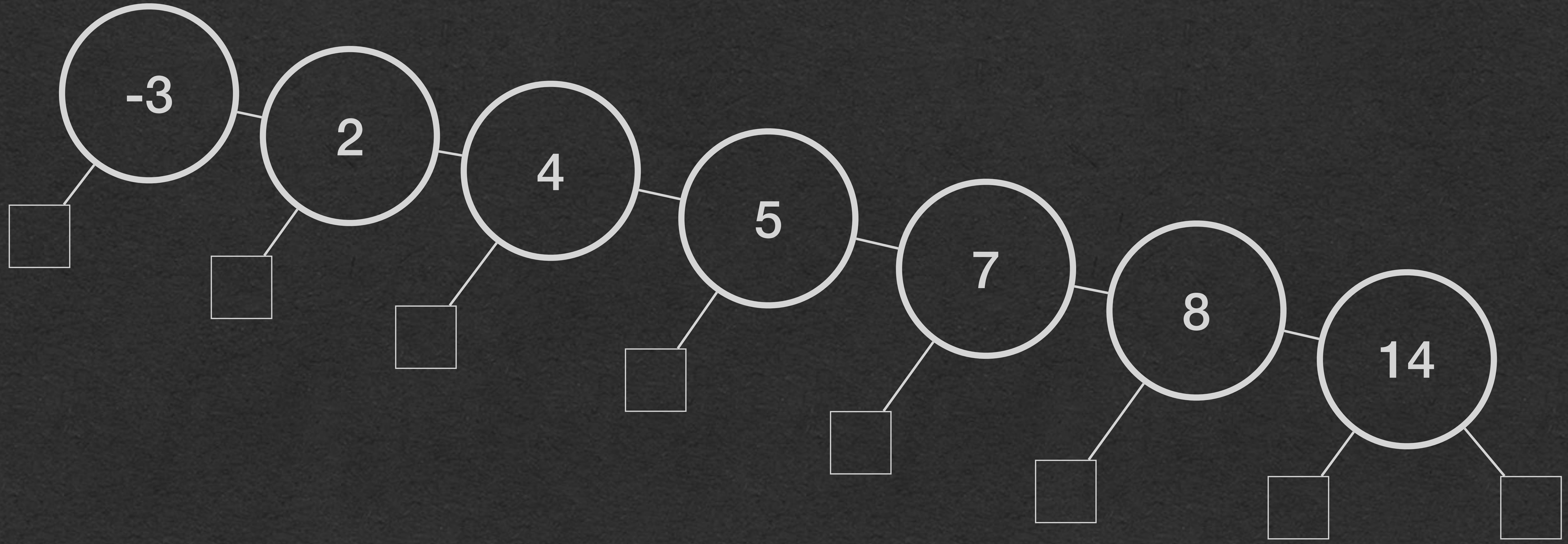
BST - Efficiency

- Vocab: A tree is balanced if each node has the same number of descendants in its left and right subtrees
- *** If a BST is balanced ***
  - The number of nodes from the root to a leaf - 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 - Inefficiency

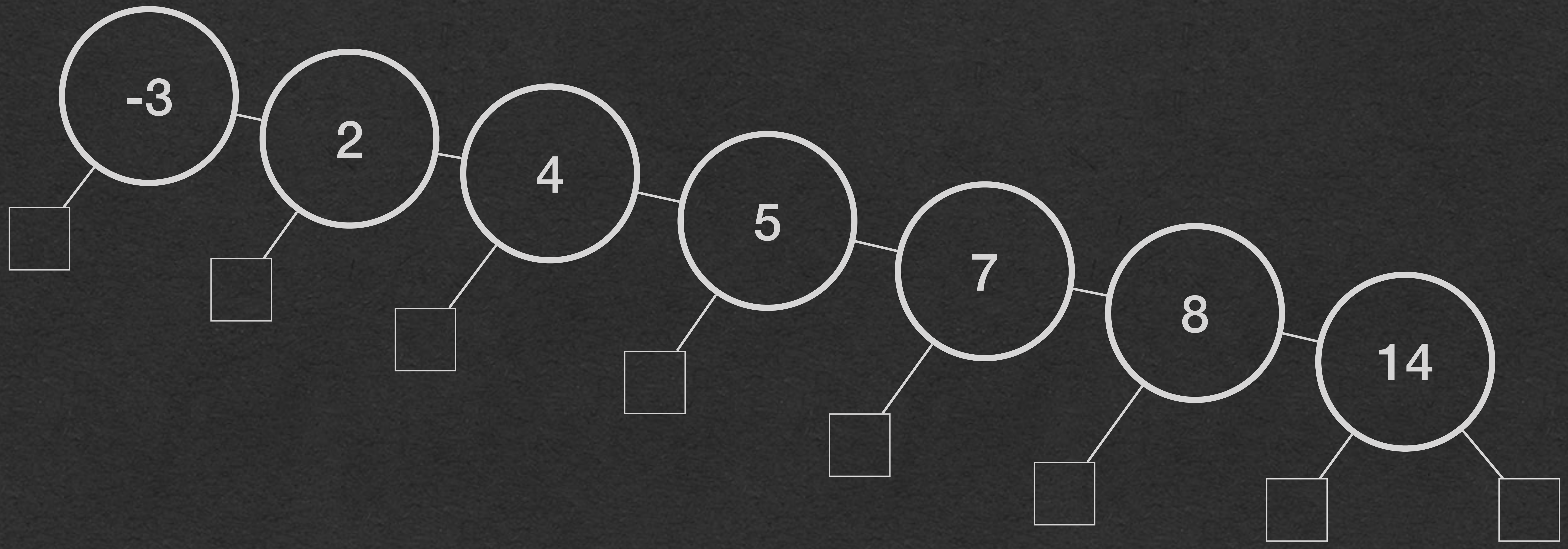
- What if the tree is not balanced?

```
val bst = new BinarySearchTree[Int](new LessThanComparator())
bst.insert(-3)
bst.insert(2)
bst.insert(4)
bst.insert(5)
bst.insert(8)
bst.insert(7)
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**