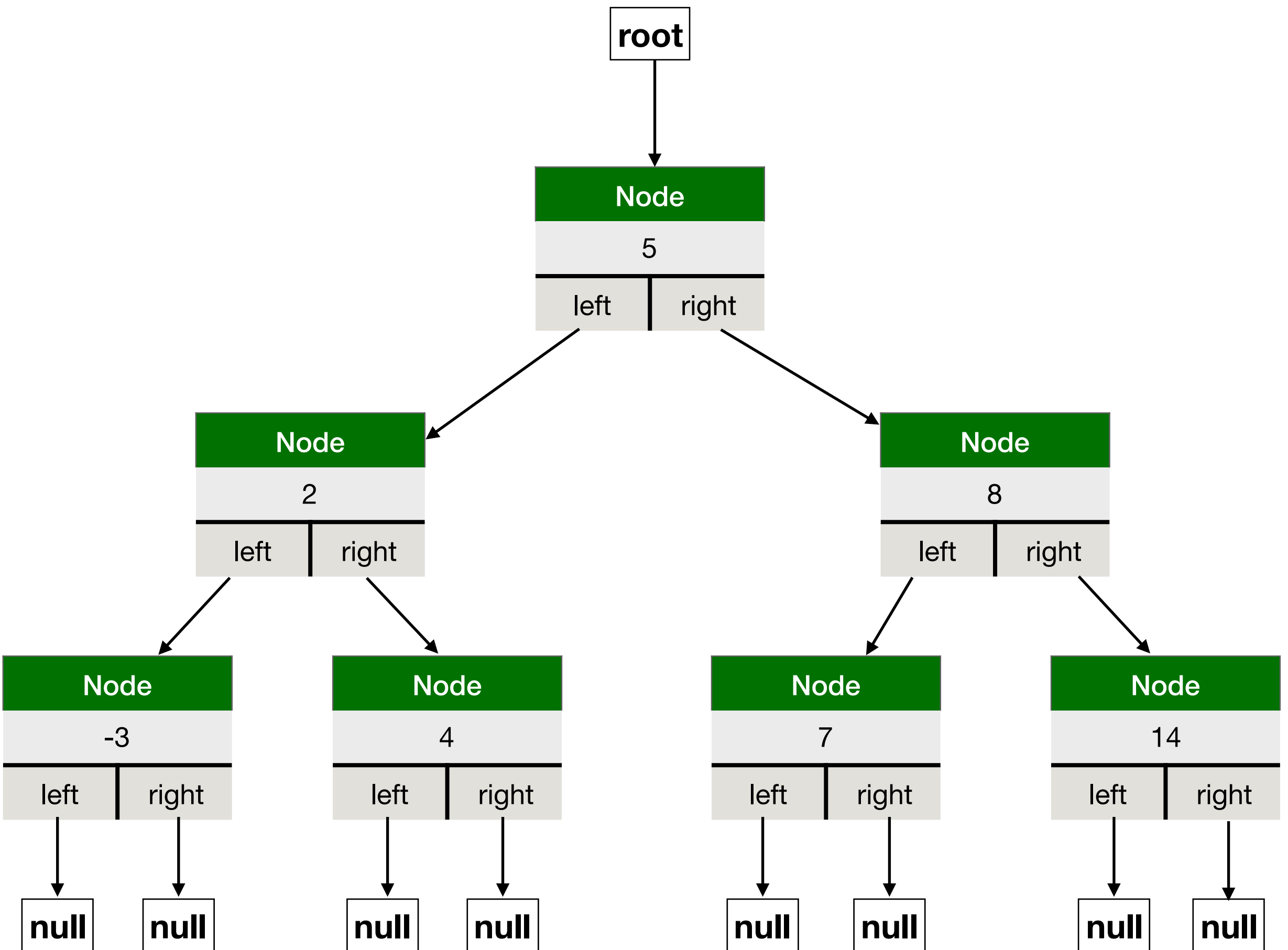


# Binary Trees and Traversals

# Binary Trees

- Similar in structure to Linked List
  - Consists of Nodes
  - A Tree is only a reference to the first node (Called the root node)
- Trees have 2 references to nodes
  - Each node has left and right reference
  - Vocab: These are called its child nodes
  - Vocab: The node is the parent to these children



# The Code

```
class BinaryTreeNode[A](var value: A, var left: BinaryTreeNode[A], var right: BinaryTreeNode[A]) {  
}
```

```
val root = new BinaryTreeNode[Int](5, null, null)  
root.left = new BinaryTreeNode[Int](2, null, null)  
root.right = new BinaryTreeNode[Int](8, null, null)  
root.left.left = new BinaryTreeNode[Int](-3, null, null)  
root.left.right = new BinaryTreeNode[Int](4, null, null)  
root.right.left = new BinaryTreeNode[Int](7, null, null)  
root.right.right = new BinaryTreeNode[Int](14, null, null)
```

- Binary Tree Nodes are very similar in structure to Linked List Nodes
- No simple prepend or append so we'll manually build a tree by setting left and right directly

# Tree Traversals

- How do we visit every node in a tree?
  - With linked lists we repeatedly visited next until the end of the list
- With trees, how do we visit both children of each node?
- We'll see 3 different approaches
  - Pre-Order Traversal
  - In-Order Traversal
  - Post-Order Traversal

# Tree Traversals

- Pre-Order Traversal
  - Visit the node's value
  - Call pre-order on the left child
  - Call pre-order on the right child

```
def preOrderTraversal[A](node: BinaryTreeNode[A], f: A => Unit): Unit = {  
  if (node != null) {  
    f(node.value)  
    preOrderTraversal(node.left, f)  
    preOrderTraversal(node.right, f)  
  }  
}
```

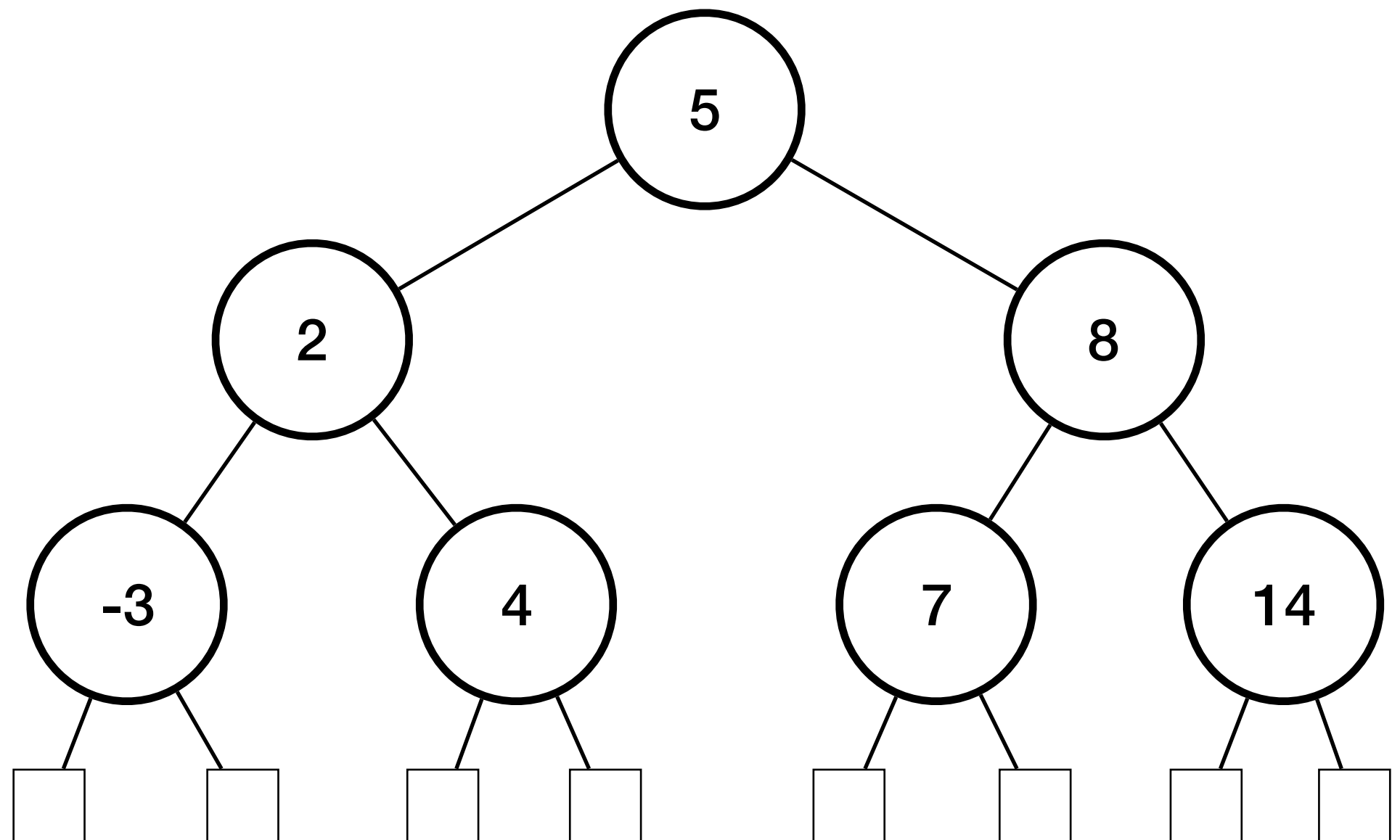
```
preOrderTraversal(root, println)
```

# Tree Traversals

```
preOrderTraversal(root, println)
```

**Printed:**

5  
2  
-3  
4  
8  
7  
14



# Tree Traversals

- Post-Order Traversal
  - Call post-order on the left child
  - Call post-order on the right child
  - Visit the node's value

```
def postOrderTraversal[A](node: BinaryTreeNode[A], f: A => Unit): Unit = {  
  if (node != null) {  
    postOrderTraversal(node.left, f)  
    postOrderTraversal(node.right, f)  
    f(node.value)  
  }  
}
```

```
postOrderTraversal(root, println)
```

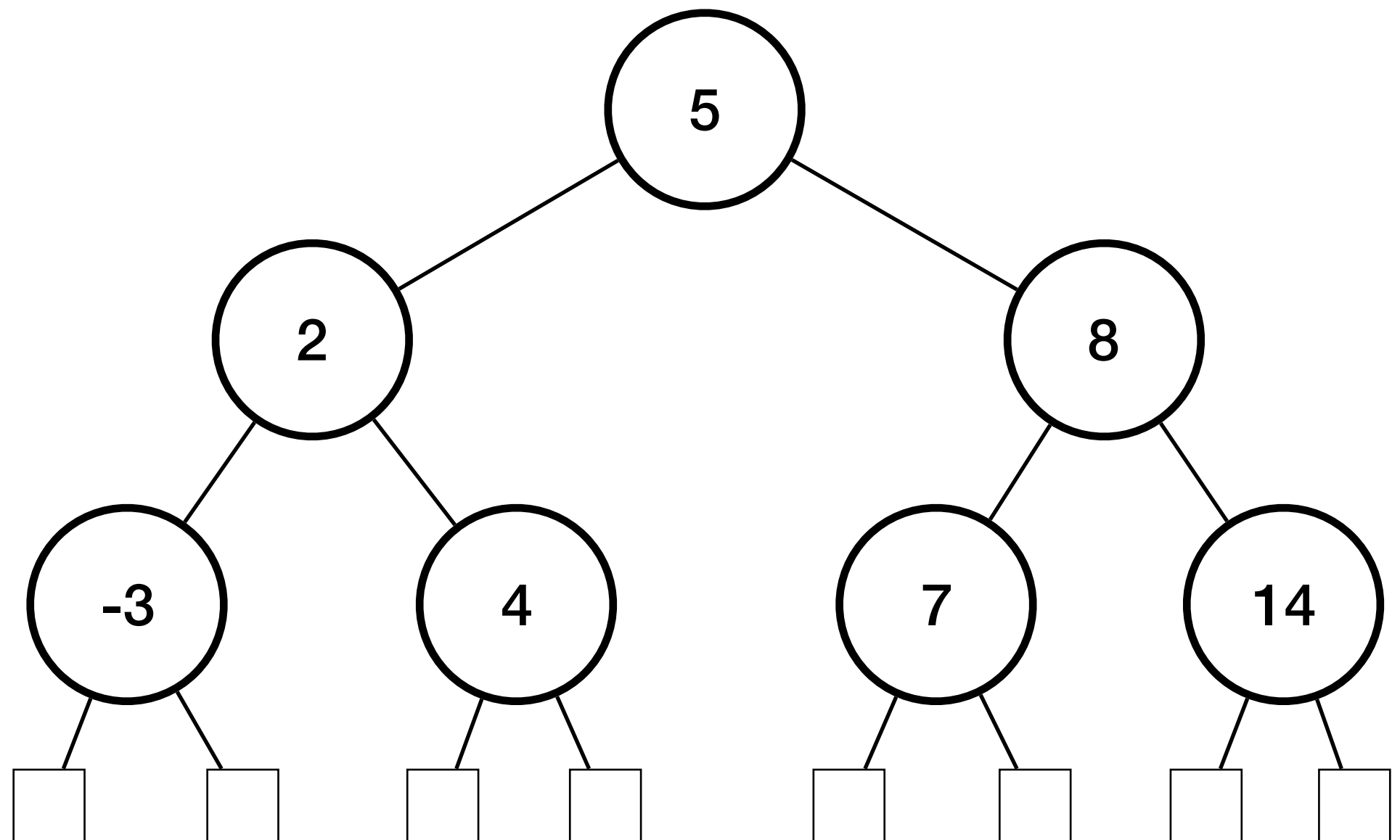


# Tree Traversals

```
postOrderTraversal(root, println)
```

**Printed:**

**-3  
4  
2  
7  
14  
8  
5**



# Tree Traversals

- In-Order Traversal
  - Call in-order on the left child
  - Visit the node's value
  - Call in-order on the right child

```
def inOrderTraversal[A](node: BinaryTreeNode[A], f: A => Unit): Unit = {  
  if (node != null) {  
    inOrderTraversal(node.left, f)  
    f(node.value)  
    inOrderTraversal(node.right, f)  
  }  
}
```

```
inOrderTraversal(root, println)
```

# The Code

```
def inOrderTraversal[A](node: BinaryTreeNode[A], f: A => Unit): Unit = {  
  if (node != null) {  
    inOrderTraversal(node.left, f)  
    f(node.value)  
    inOrderTraversal(node.right, f)  
  }  
}
```

```
def preOrderTraversal[A](node: BinaryTreeNode[A], f: A => Unit): Unit = {  
  if (node != null) {  
    f(node.value)  
    preOrderTraversal(node.left, f)  
    preOrderTraversal(node.right, f)  
  }  
}
```

```
def postOrderTraversal[A](node: BinaryTreeNode[A], f: A => Unit): Unit = {  
  if (node != null) {  
    postOrderTraversal(node.left, f)  
    postOrderTraversal(node.right, f)  
    f(node.value)  
  }  
}
```

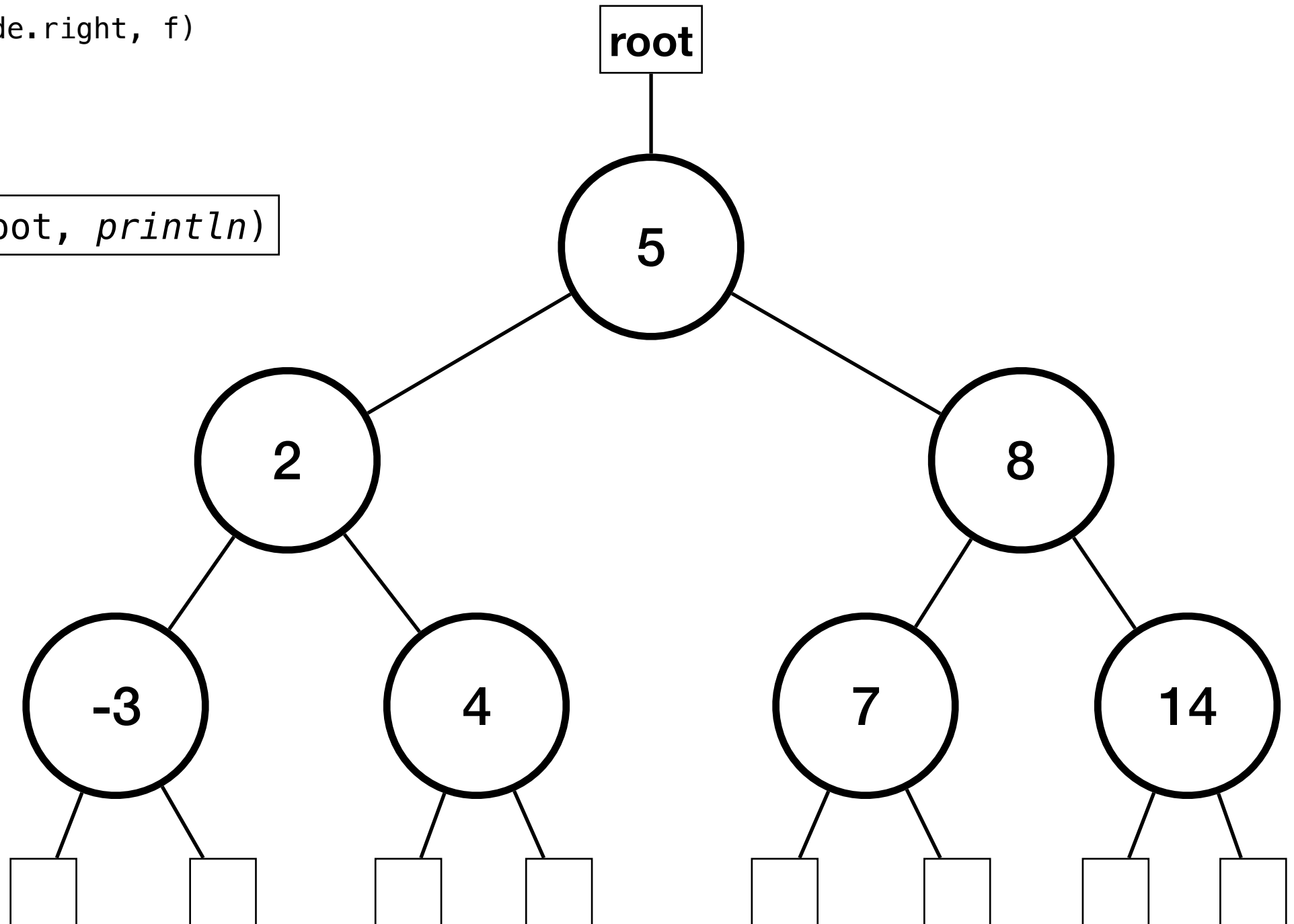
- **Challenge: Write these with loops and no recursion**

# Traversals

```
def inOrderTraversal[A](node: BinaryTreeNode[A], f: A => Unit): Unit = {  
  if (node != null) {  
    inOrderTraversal(node.left, f)  
    f(node.value)  
    inOrderTraversal(node.right, f)  
  }  
}
```

```
inOrderTraversal(root, println)
```

**Printed:**

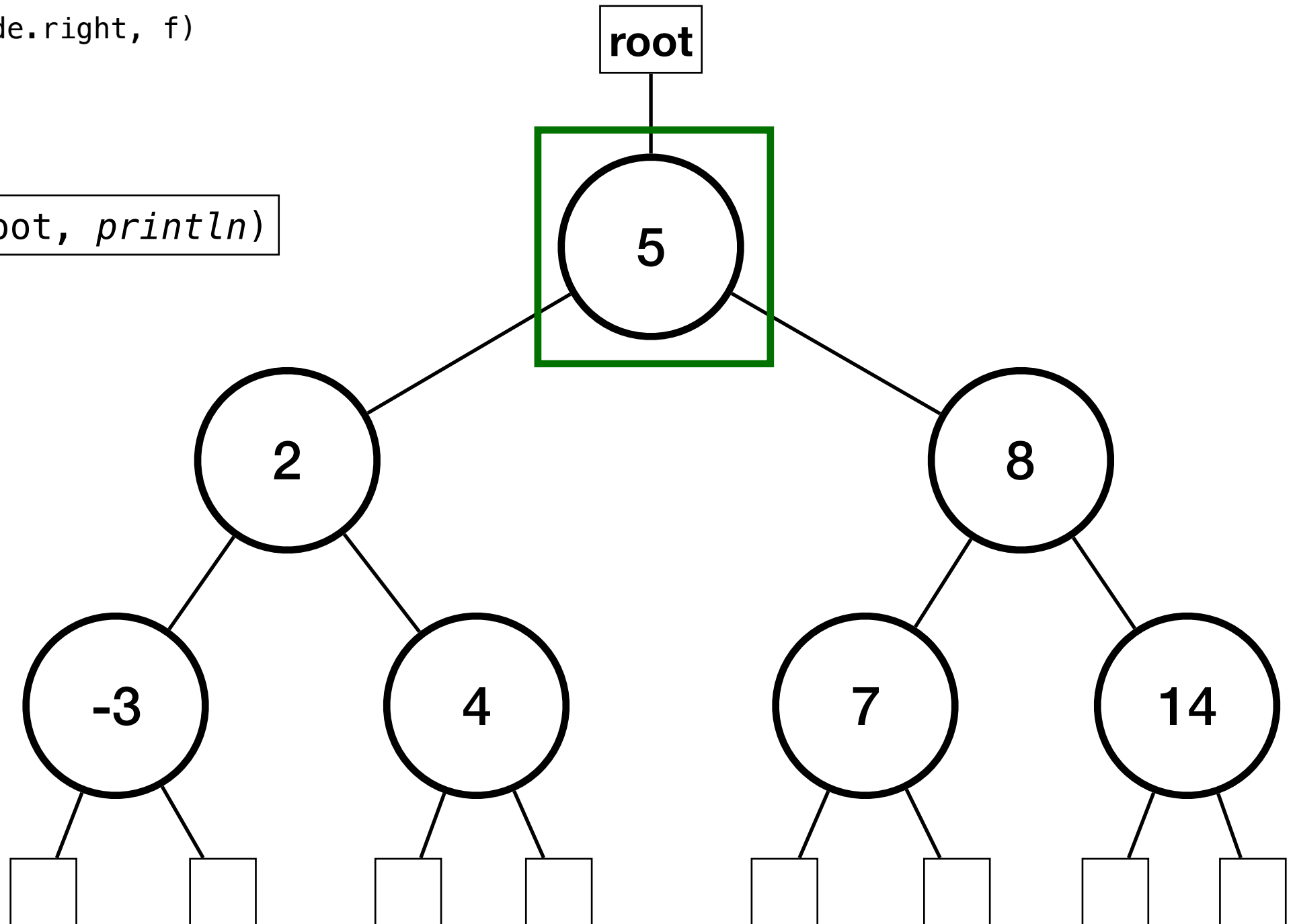


# Traversals

```
def inOrderTraversal[A](node: BinaryTreeNode[A], f: A => Unit): Unit = {  
  if (node != null) {  
    inOrderTraversal(node.left, f)  
    f(node.value)  
    inOrderTraversal(node.right, f)  
  }  
}
```

```
inOrderTraversal(root, println)
```

**Printed:**

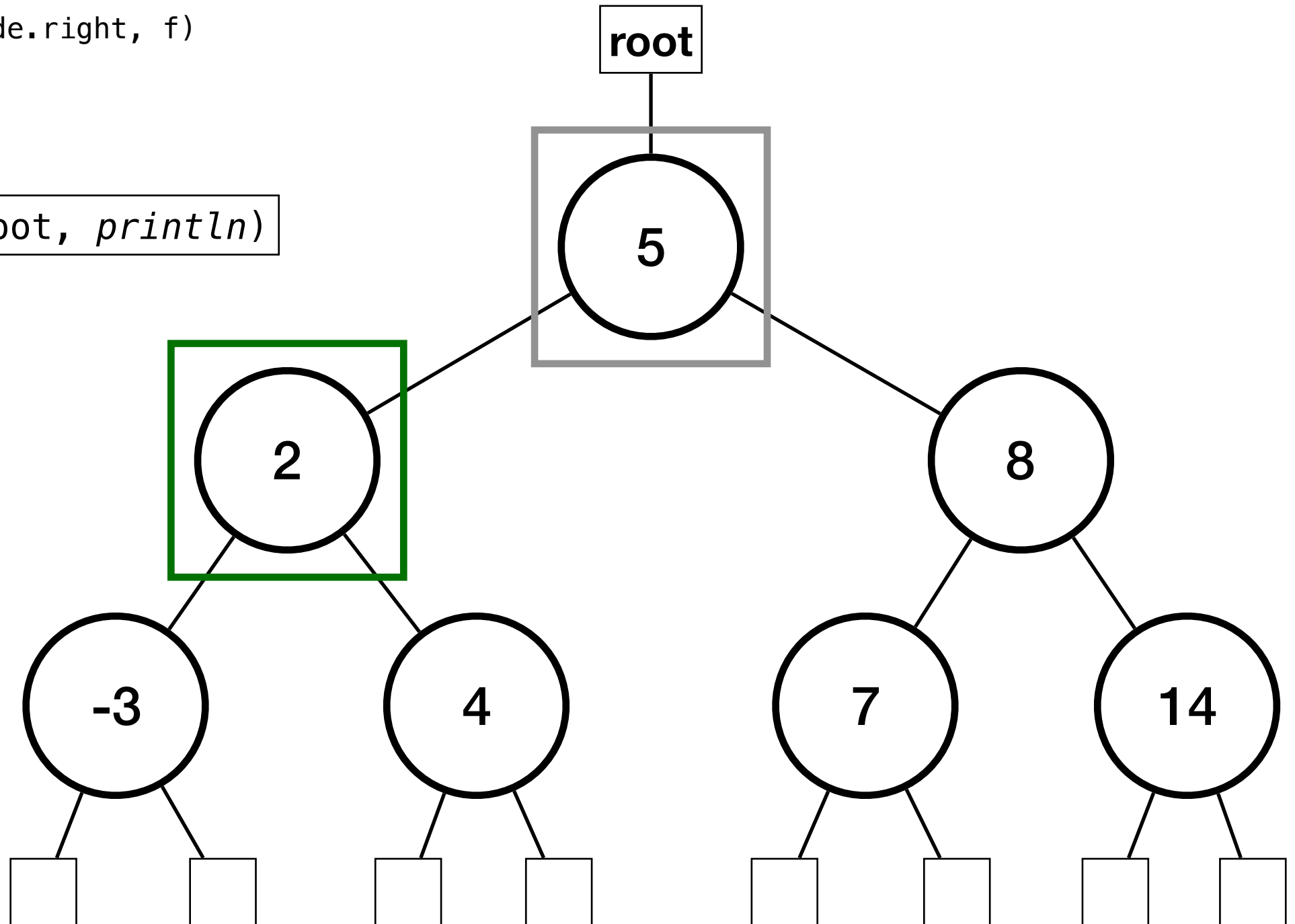


# Traversals

```
def inOrderTraversal[A](node: BinaryTreeNode[A], f: A => Unit): Unit = {  
  if (node != null) {  
    inOrderTraversal(node.left, f)  
    f(node.value)  
    inOrderTraversal(node.right, f)  
  }  
}
```

```
inOrderTraversal(root, println)
```

**Printed:**

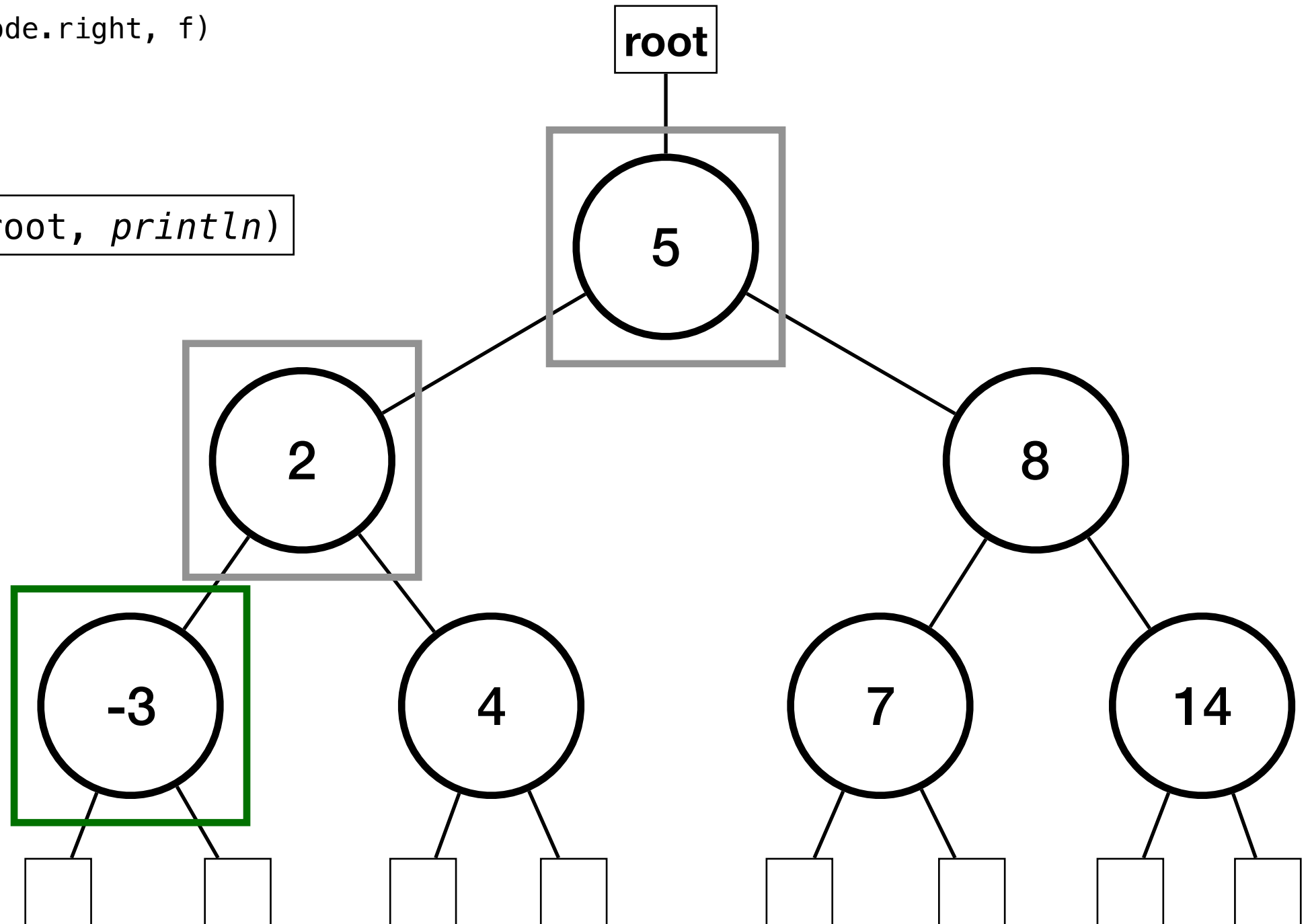


# Traversals

```
def inOrderTraversal[A](node: BinaryTreeNode[A], f: A => Unit): Unit = {  
  if (node != null) {  
    inOrderTraversal(node.left, f)  
    f(node.value)  
    inOrderTraversal(node.right, f)  
  }  
}
```

```
inOrderTraversal(root, println)
```

**Printed:**

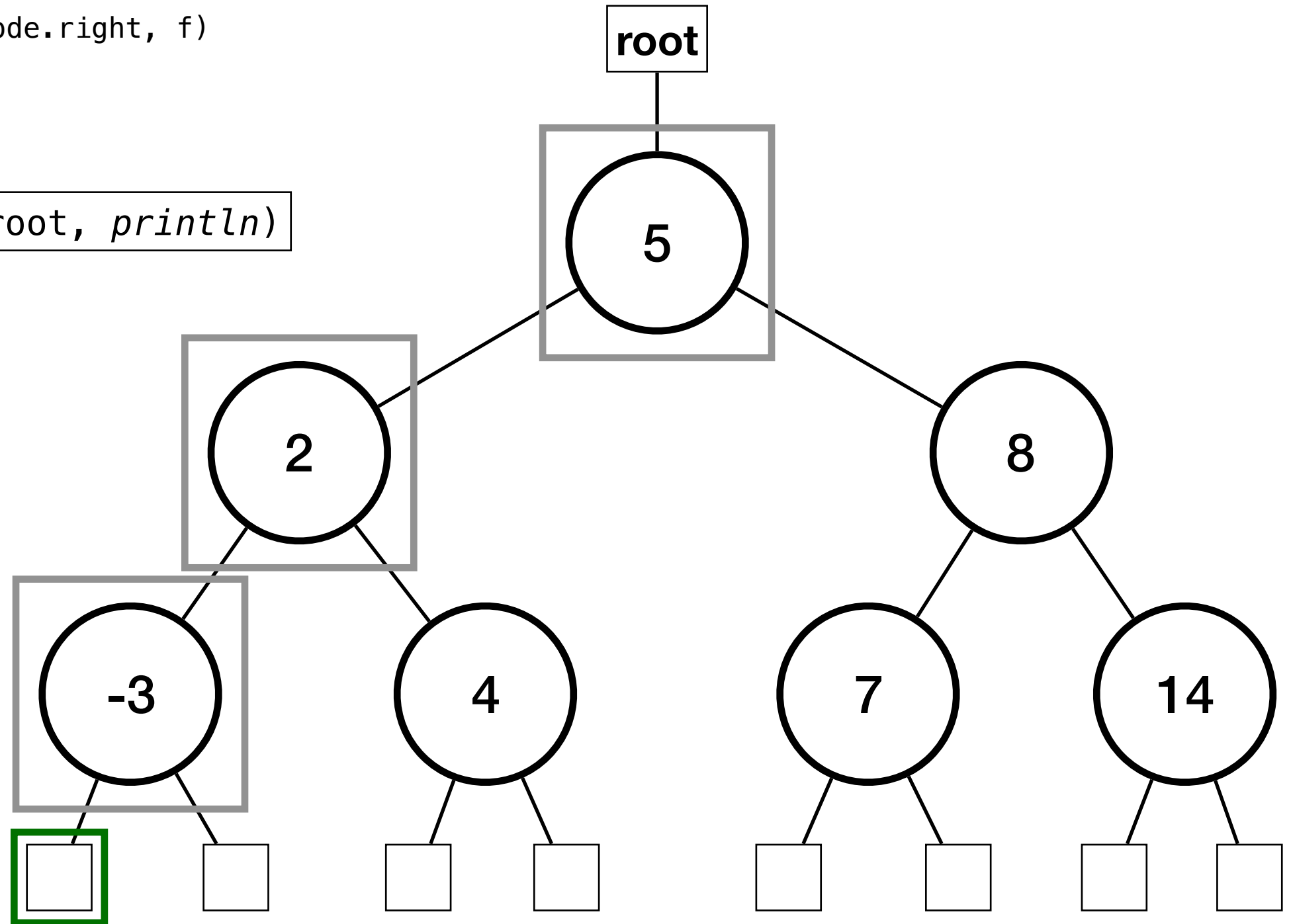


# Traversals

```
def inOrderTraversal[A](node: BinaryTreeNode[A], f: A => Unit): Unit = {  
  if (node != null) {  
    inOrderTraversal(node.left, f)  
    f(node.value)  
    inOrderTraversal(node.right, f)  
  }  
}
```

```
inOrderTraversal(root, println)
```

**Printed:**





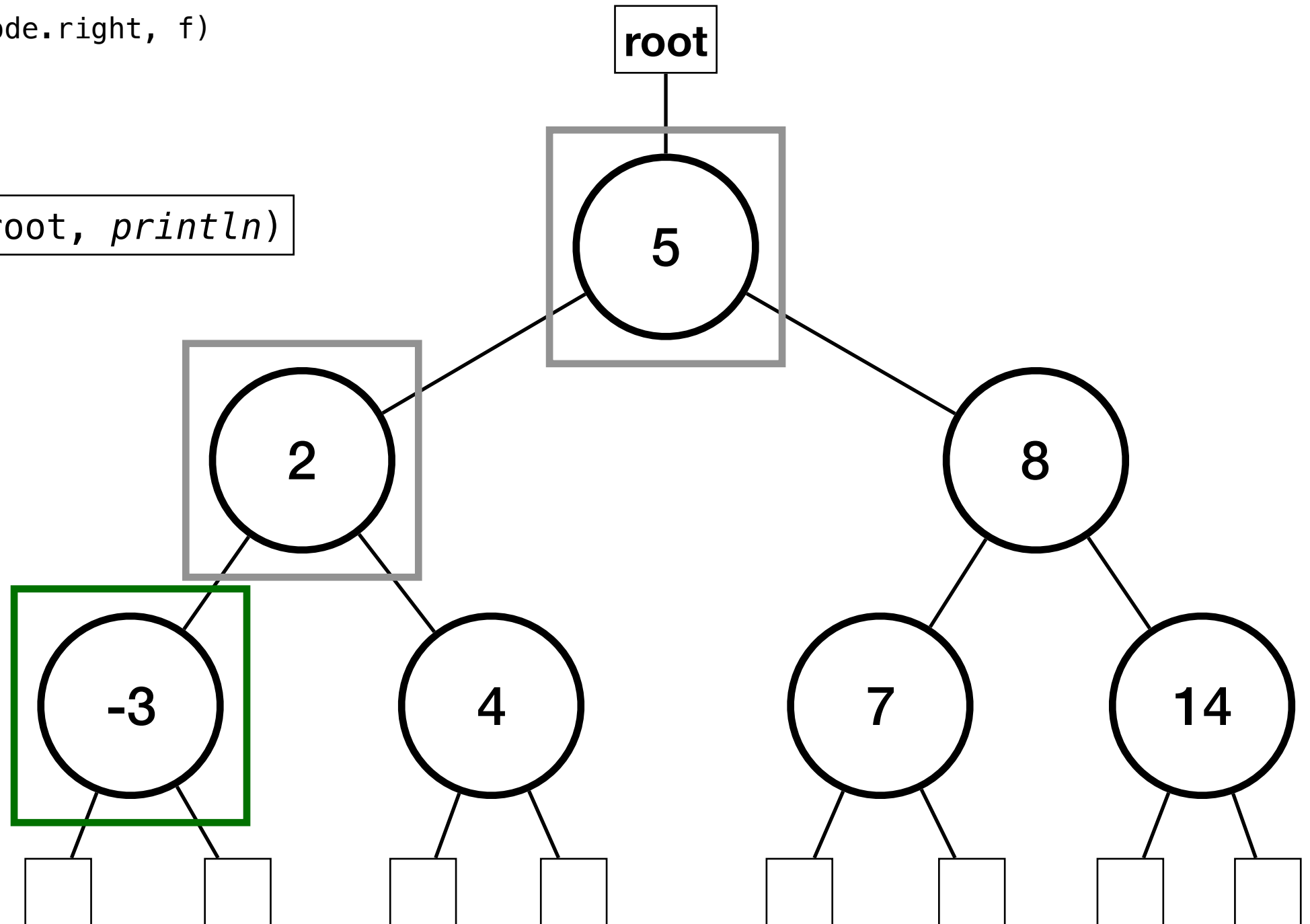
# Traversals

```
def inOrderTraversal[A](node: BinaryTreeNode[A], f: A => Unit): Unit = {  
  if (node != null) {  
    inOrderTraversal(node.left, f)  
    f(node.value)  
    inOrderTraversal(node.right, f)  
  }  
}
```

```
inOrderTraversal(root, println)
```

**Printed:**

**-3**



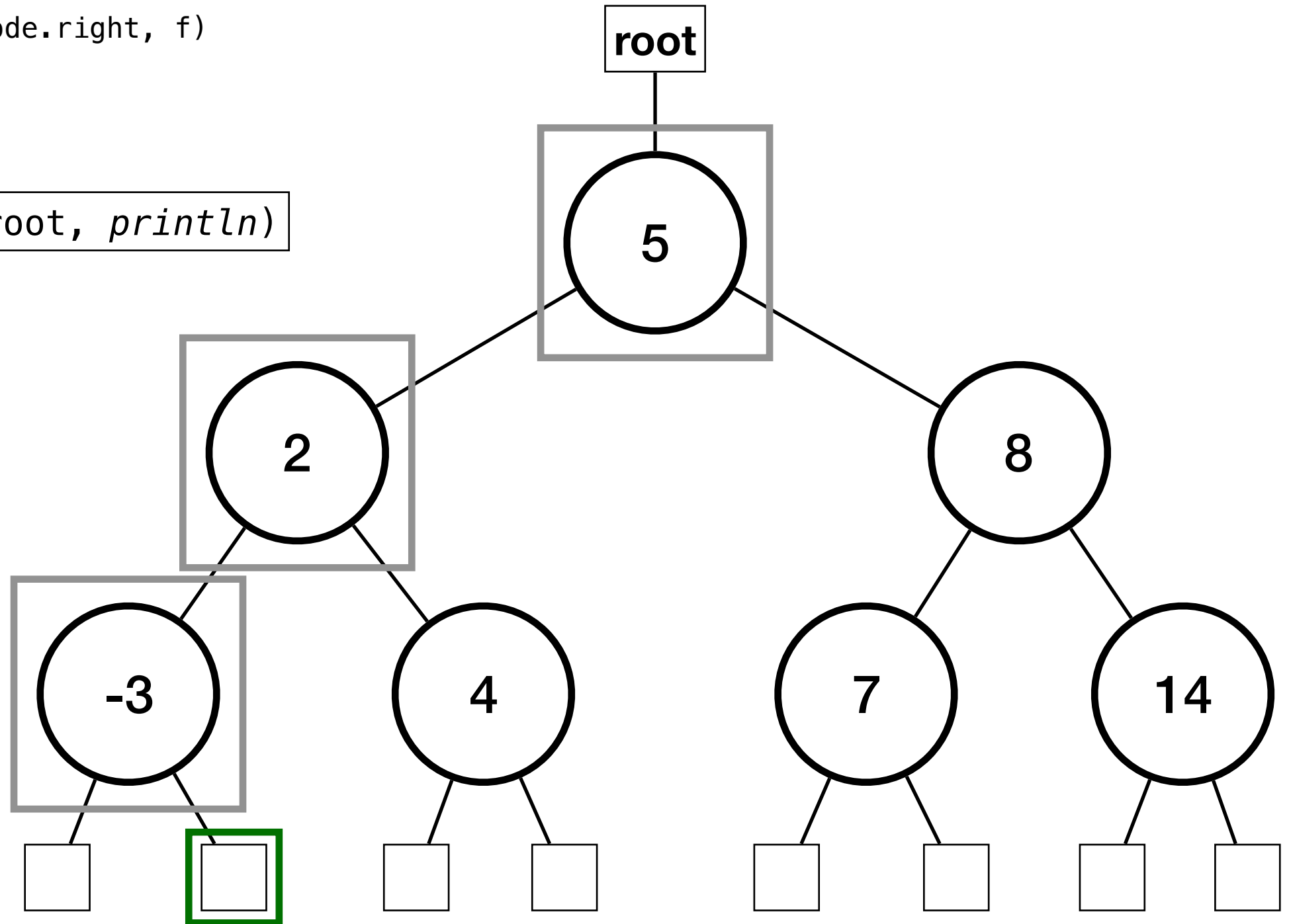
# Traversals

```
def inOrderTraversal[A](node: BinaryTreeNode[A], f: A => Unit): Unit = {  
  if (node != null) {  
    inOrderTraversal(node.left, f)  
    f(node.value)  
    inOrderTraversal(node.right, f)  
  }  
}
```

```
inOrderTraversal(root, println)
```

**Printed:**

**-3**



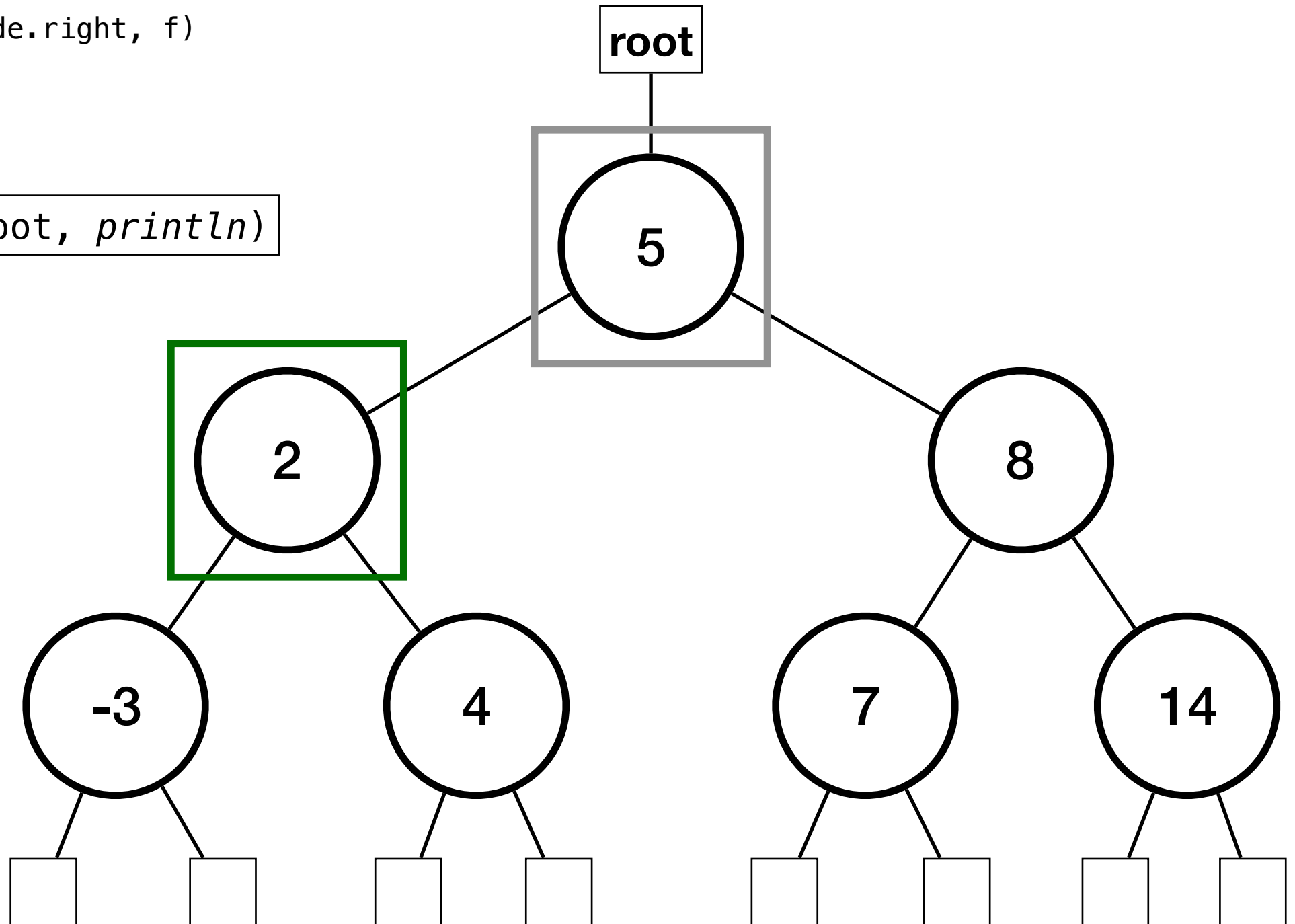
# Traversals

```
def inOrderTraversal[A](node: BinaryTreeNode[A], f: A => Unit): Unit = {  
  if (node != null) {  
    inOrderTraversal(node.left, f)  
    f(node.value)  
    inOrderTraversal(node.right, f)  
  }  
}
```

```
inOrderTraversal(root, println)
```

**Printed:**

-3  
2



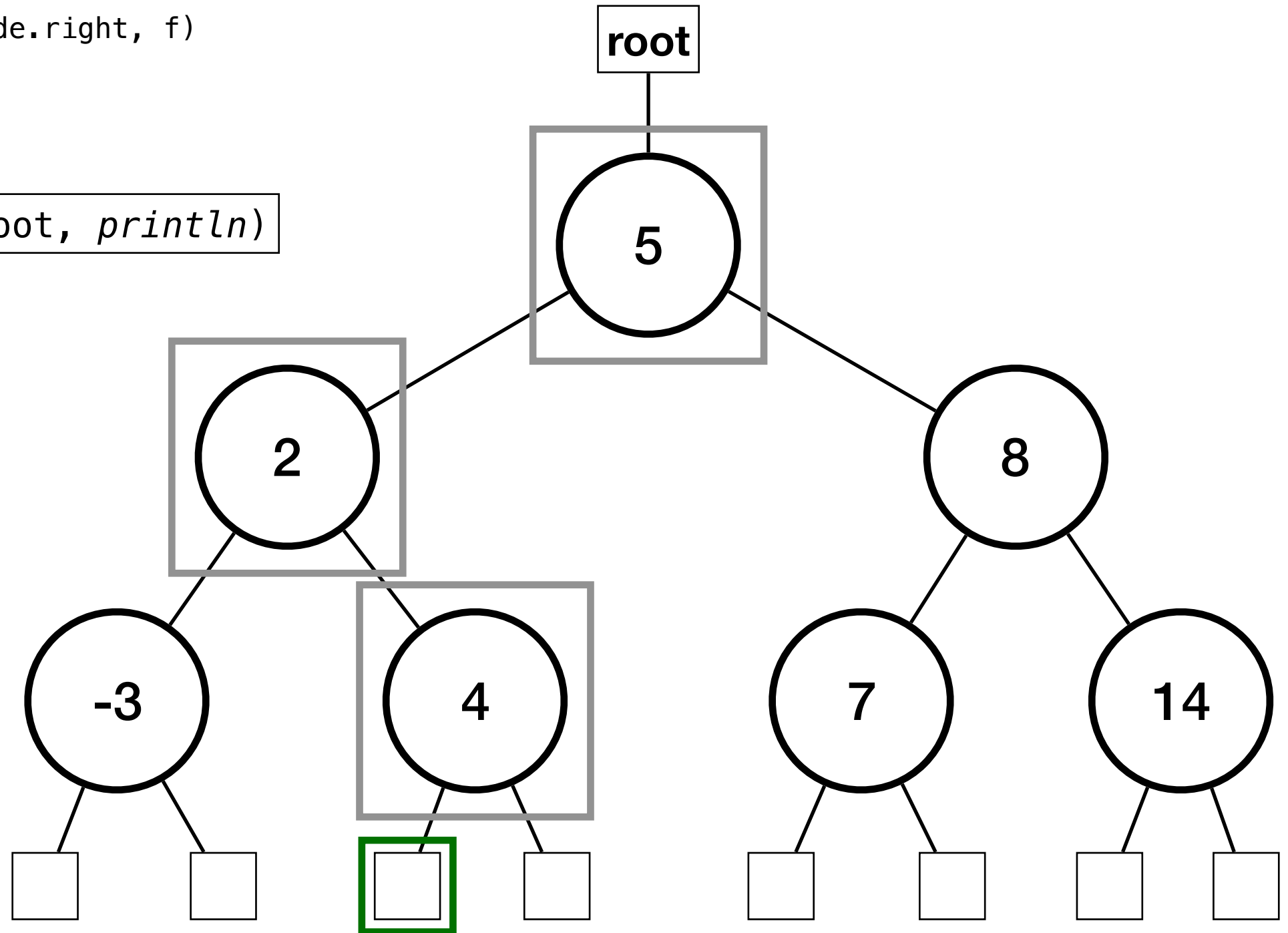
# Traversals

```
def inOrderTraversal[A](node: BinaryTreeNode[A], f: A => Unit): Unit = {  
  if (node != null) {  
    inOrderTraversal(node.left, f)  
    f(node.value)  
    inOrderTraversal(node.right, f)  
  }  
}
```

```
inOrderTraversal(root, println)
```

**Printed:**

-3  
2



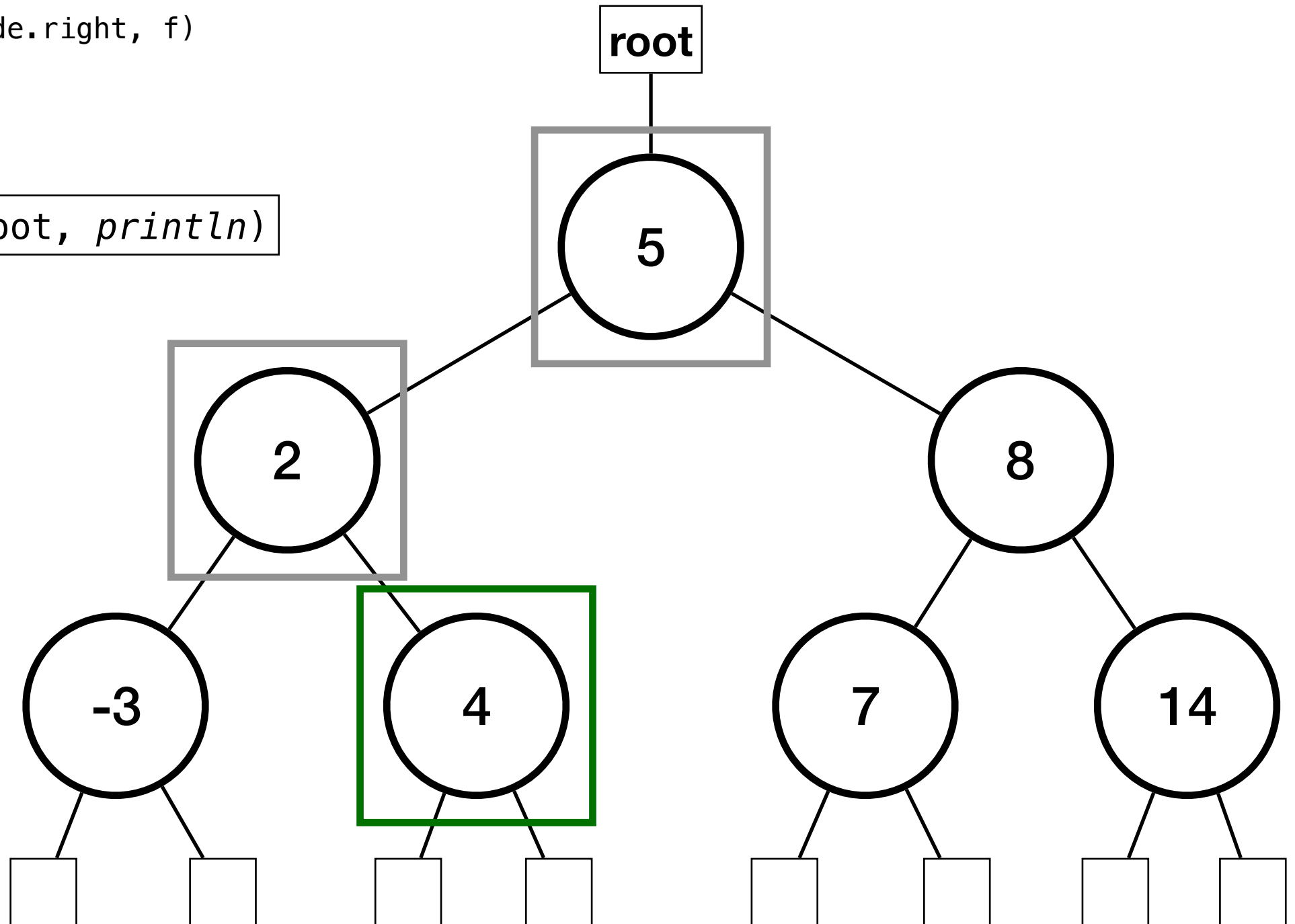
# Traversals

```
def inOrderTraversal[A](node: BinaryTreeNode[A], f: A => Unit): Unit = {  
  if (node != null) {  
    inOrderTraversal(node.left, f)  
    f(node.value)  
    inOrderTraversal(node.right, f)  
  }  
}
```

```
inOrderTraversal(root, println)
```

**Printed:**

-3  
2  
4



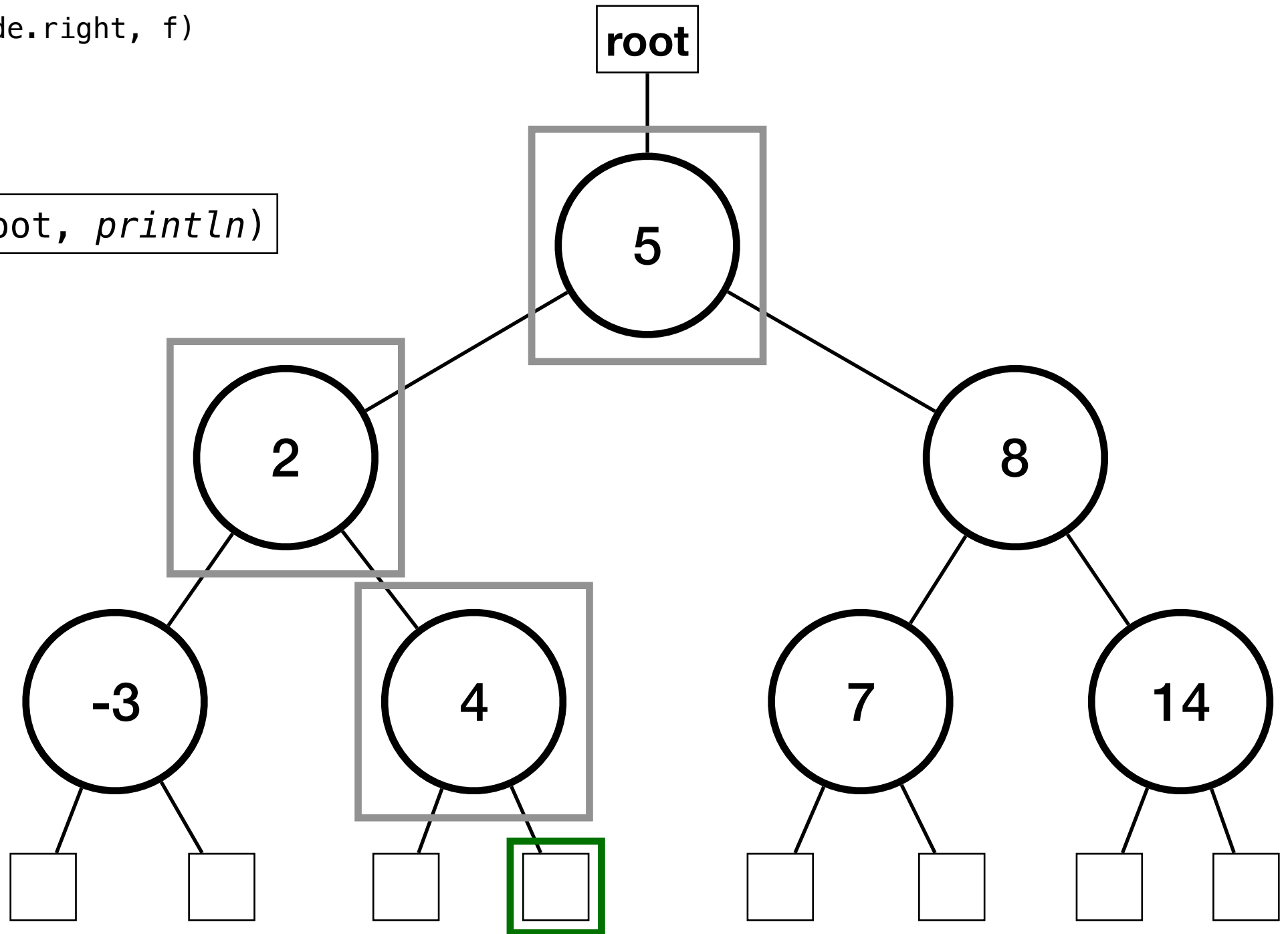
# Traversals

```
def inOrderTraversal[A](node: BinaryTreeNode[A], f: A => Unit): Unit = {  
  if (node != null) {  
    inOrderTraversal(node.left, f)  
    f(node.value)  
    inOrderTraversal(node.right, f)  
  }  
}
```

```
inOrderTraversal(root, println)
```

**Printed:**

-3  
2  
4



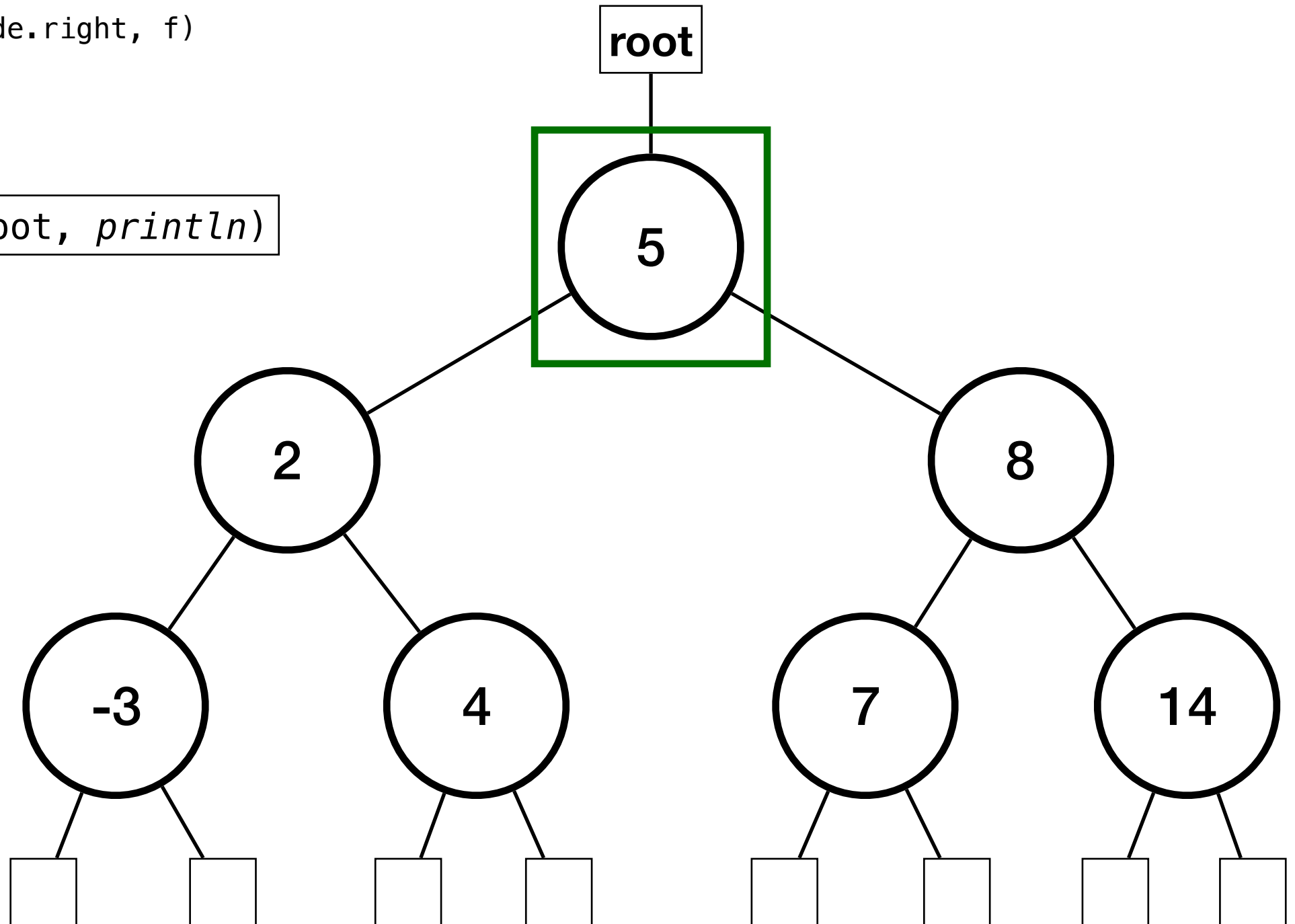
# Traversals

```
def inOrderTraversal[A](node: BinaryTreeNode[A], f: A => Unit): Unit = {  
  if (node != null) {  
    inOrderTraversal(node.left, f)  
    f(node.value)  
    inOrderTraversal(node.right, f)  
  }  
}
```

```
inOrderTraversal(root, println)
```

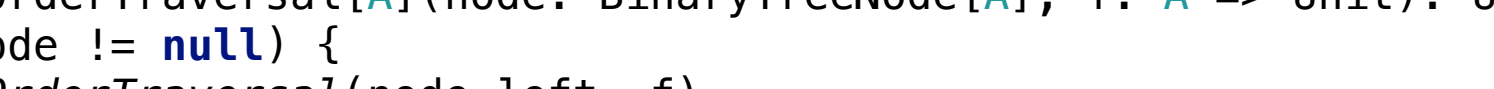
**Printed:**

-3  
2  
4  
5



# Traversals

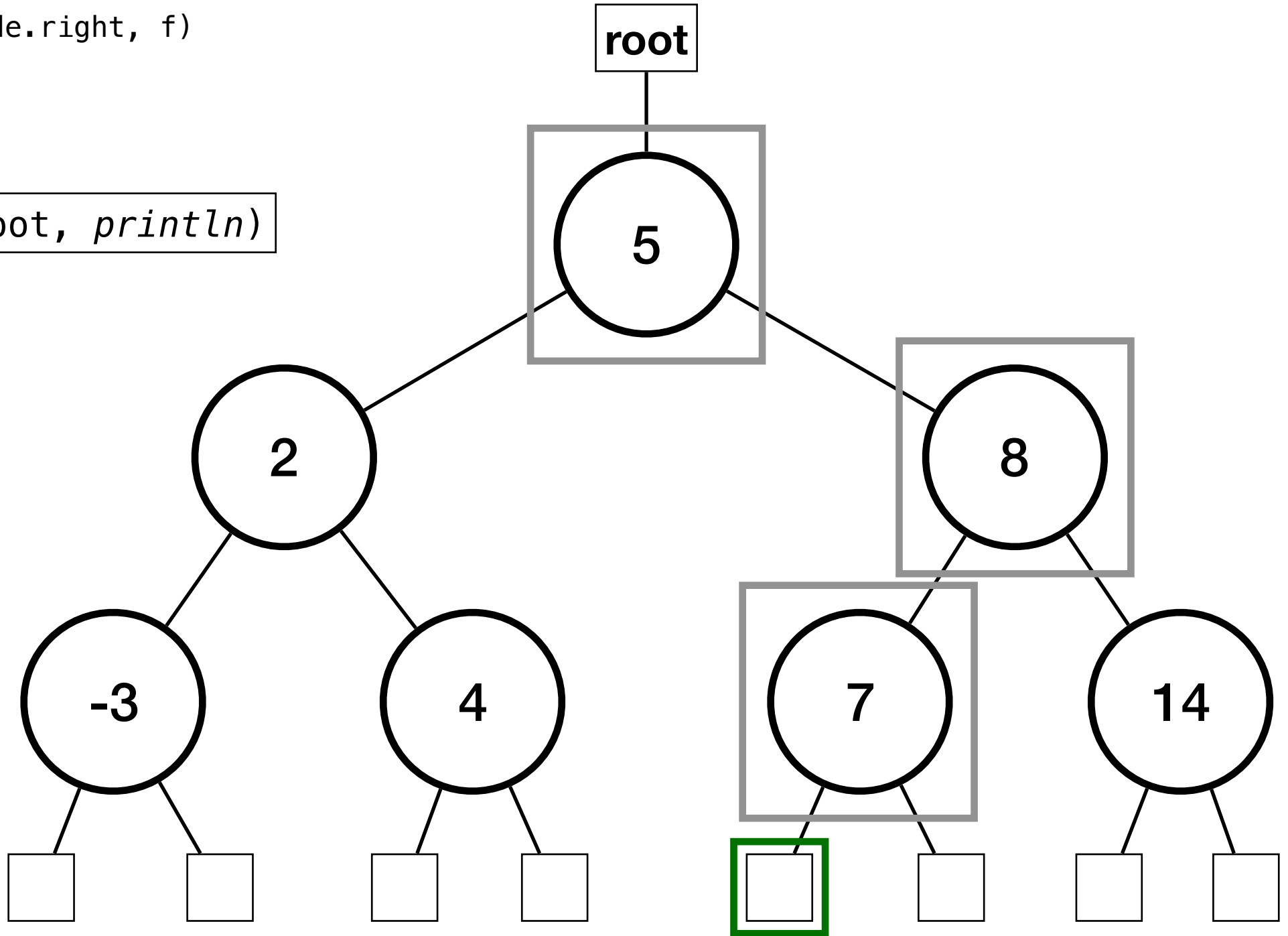
```
def inOrderTraversal[A](node: BinaryTreeNode[A], f: A => Unit): Unit = {
  if (node != null) {
    inOrderTraversal(node.left, f)
    f(node.value)
    inOrderTraversal(node.right, f)
  }
}
```



```
inOrderTraversal(root, println)
```

**Printed:**

**-3**  
**2**  
**4**  
**5**





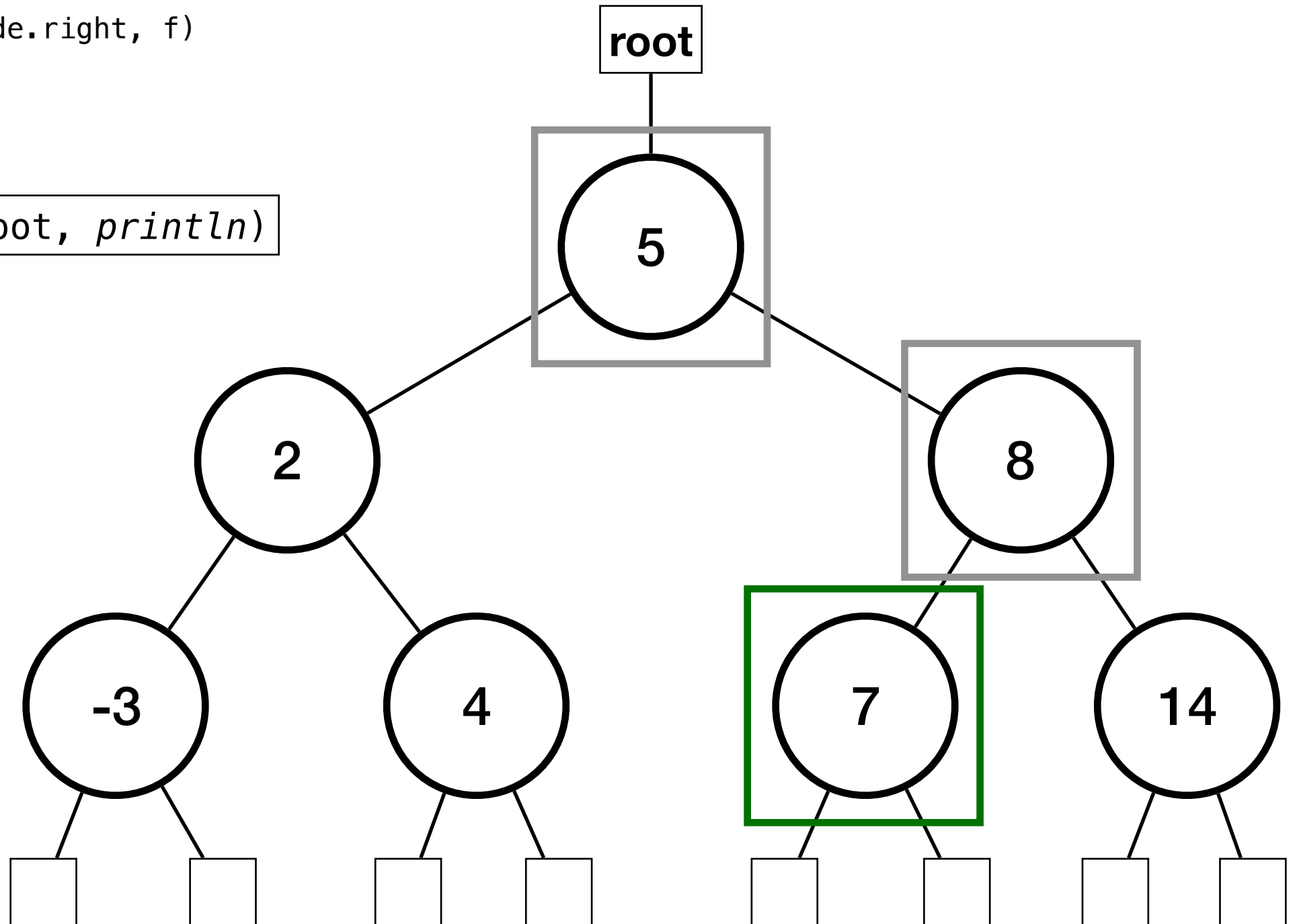
# Traversals

```
def inOrderTraversal[A](node: BinaryTreeNode[A], f: A => Unit): Unit = {  
  if (node != null) {  
    inOrderTraversal(node.left, f)  
    f(node.value)  
    inOrderTraversal(node.right, f)  
  }  
}
```

```
inOrderTraversal(root, println)
```

**Printed:**

-3  
2  
4  
5  
7



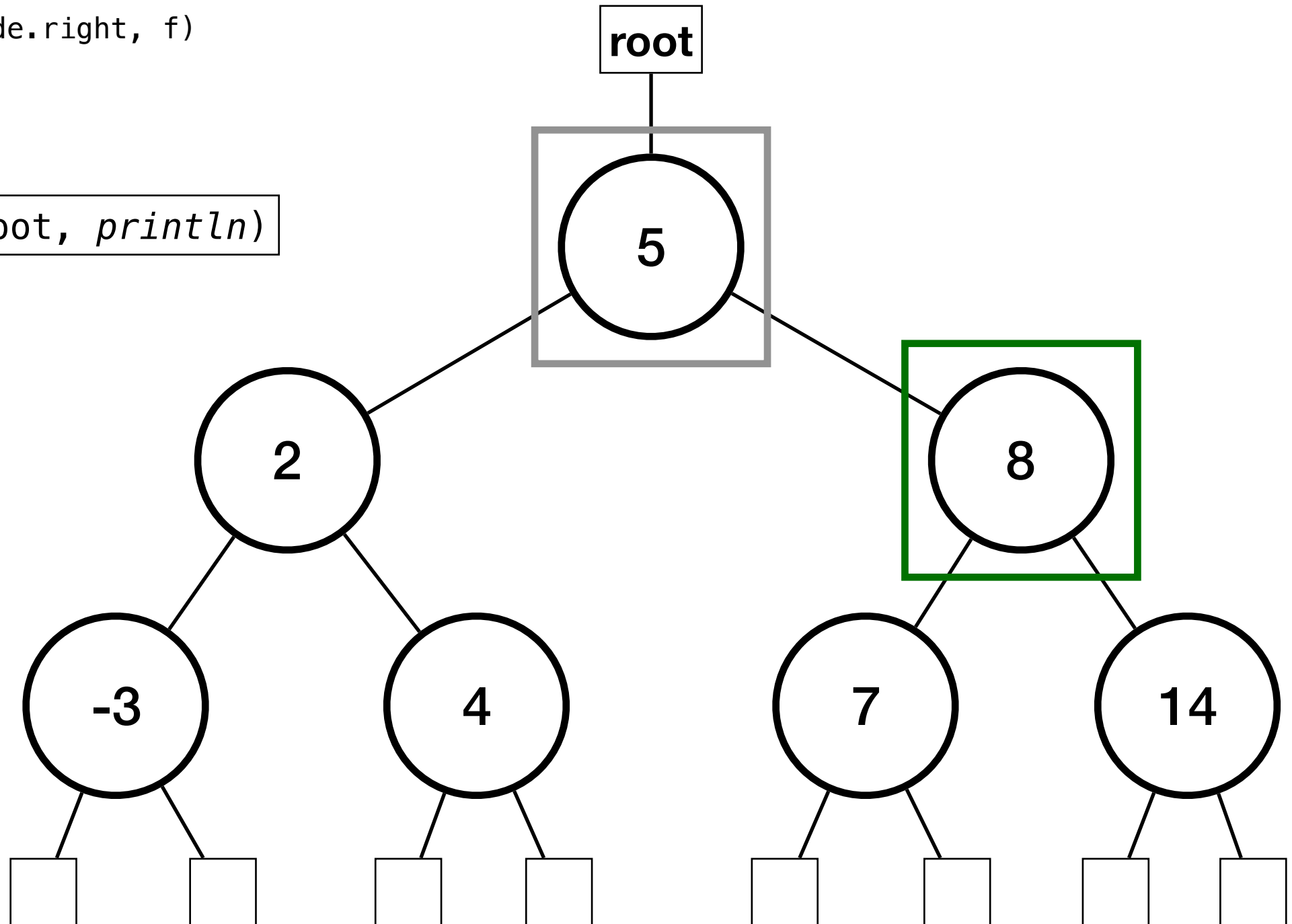
# Traversals

```
def inOrderTraversal[A](node: BinaryTreeNode[A], f: A => Unit): Unit = {  
  if (node != null) {  
    inOrderTraversal(node.left, f)  
    f(node.value)  
    inOrderTraversal(node.right, f)  
  }  
}
```

```
inOrderTraversal(root, println)
```

**Printed:**

-3  
2  
4  
5  
7  
8



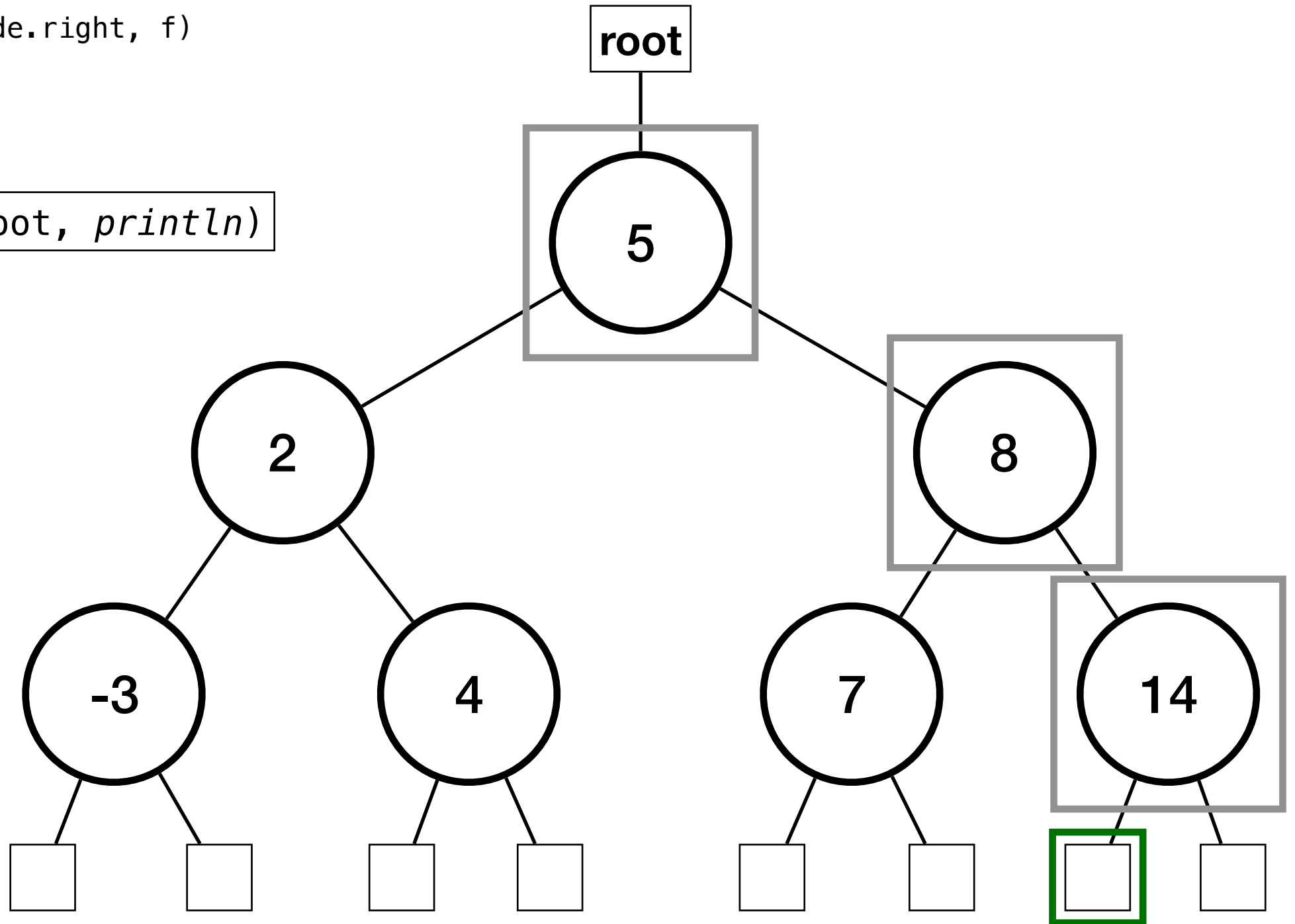
# Traversals

```
def inOrderTraversal[A](node: BinaryTreeNode[A], f: A => Unit): Unit = {  
  if (node != null) {  
    inOrderTraversal(node.left, f)  
    f(node.value)  
    inOrderTraversal(node.right, f)  
  }  
}
```

```
inOrderTraversal(root, println)
```

**Printed:**

-3  
2  
4  
5  
7  
8



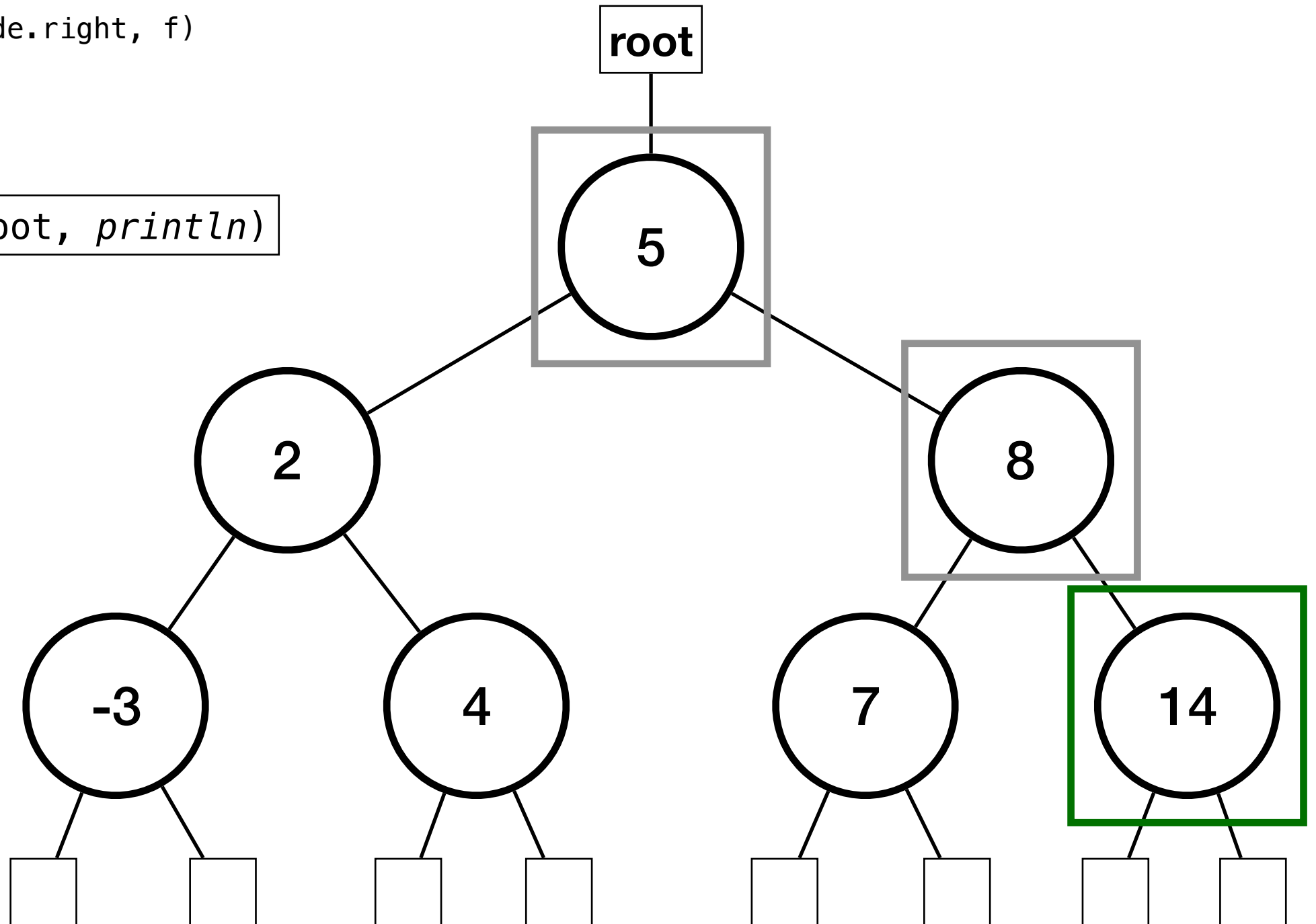
# Traversals

```
def inOrderTraversal[A](node: BinaryTreeNode[A], f: A => Unit): Unit = {  
  if (node != null) {  
    inOrderTraversal(node.left, f)  
    f(node.value)  
    inOrderTraversal(node.right, f)  
  }  
}
```

```
inOrderTraversal(root, println)
```

**Printed:**

-3  
2  
4  
5  
7  
8  
14



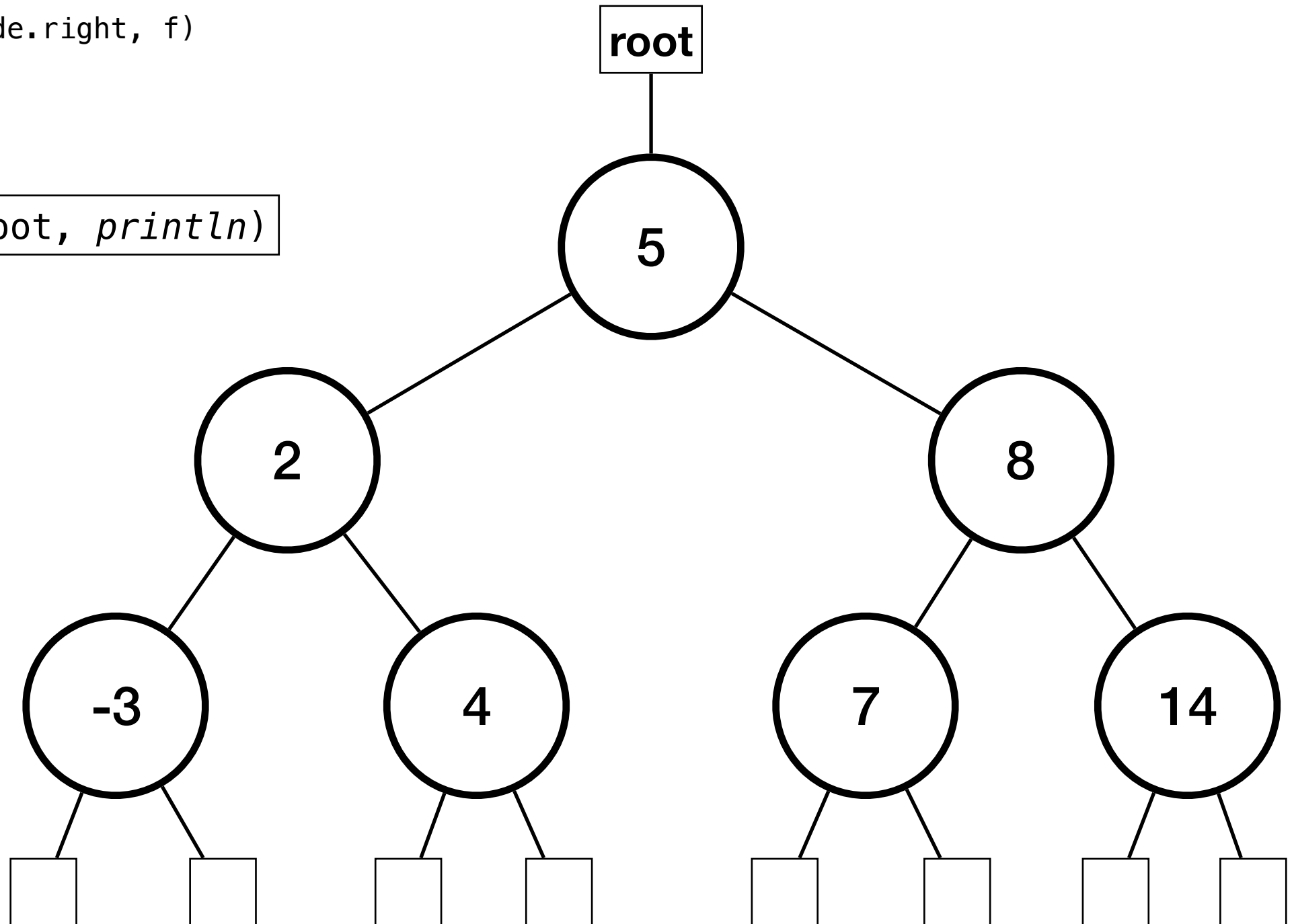
# Traversals

```
def inOrderTraversal[A](node: BinaryTreeNode[A], f: A => Unit): Unit = {  
  if (node != null) {  
    inOrderTraversal(node.left, f)  
    f(node.value)  
    inOrderTraversal(node.right, f)  
  }  
}
```

```
inOrderTraversal(root, println)
```

**Printed:**

-3  
2  
4  
5  
7  
8  
14



# Lecture Question

## Task: Find the size of a Tree

- In the `datastructures.BinaryTreeNode` class write a method named `size` that returns the number of values (as an `Int`) in the tree rooted at this node
- This method does not take any parameters, but remember you always have access to the calling object of a method by using the `"this"` keyword

\* This question will be open until midnight