

Binary Trees and Traversals

Lecture Question

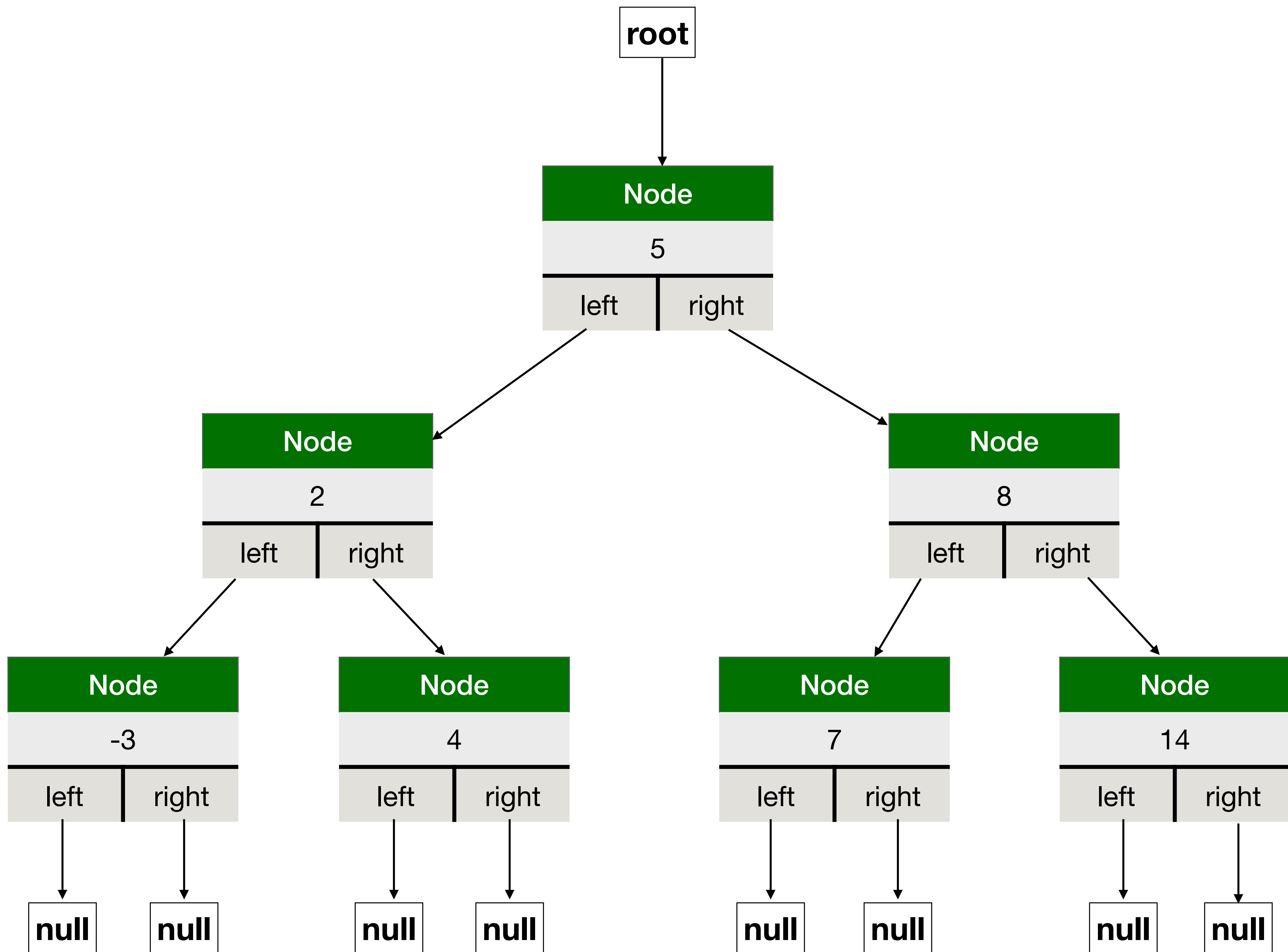
Task: Evaluate an expression tree

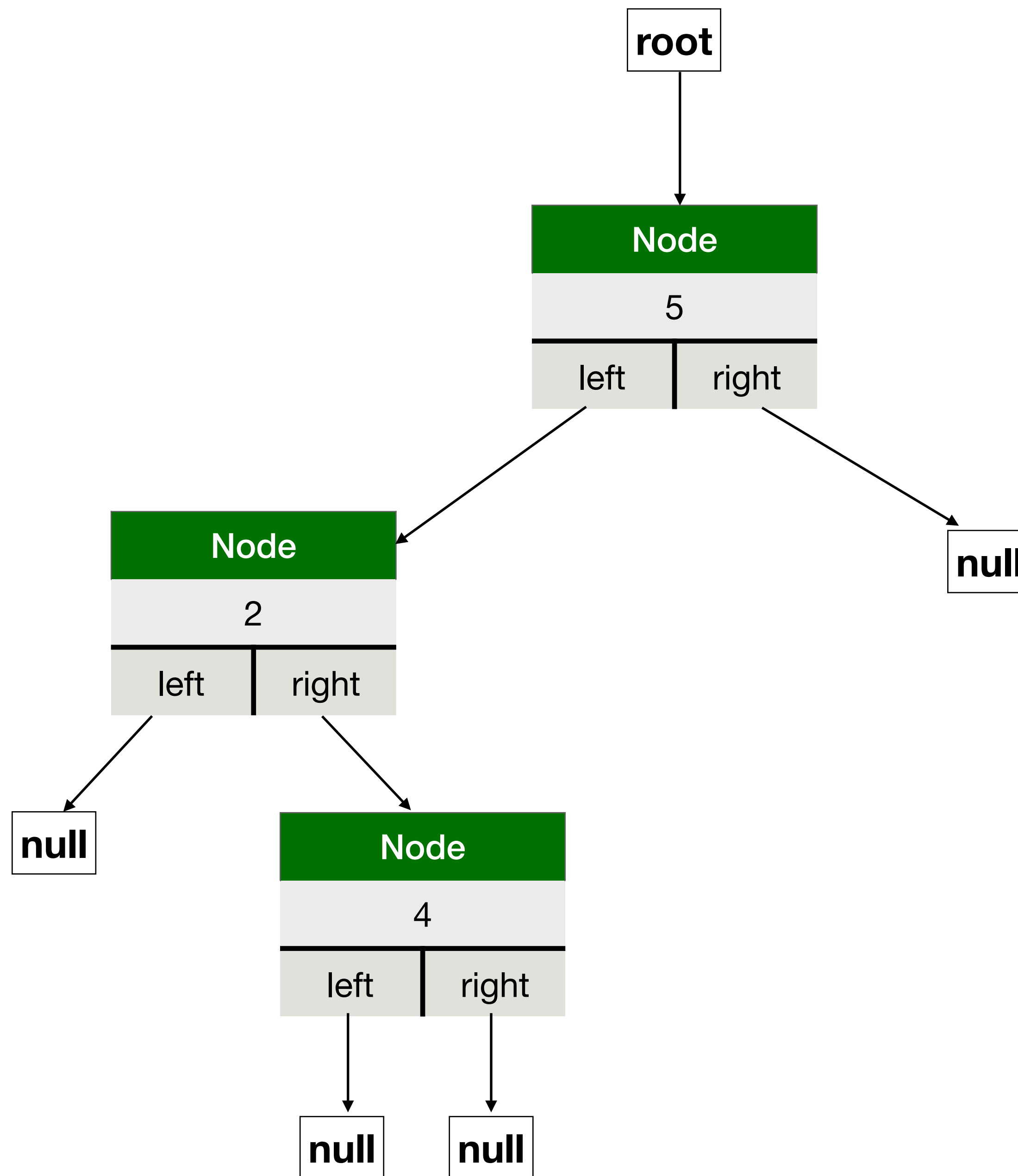
- In the `week8.trees.ExpressionTree` object, write a method named `evaluateTree` that takes the root of an expression tree (`BinaryTreeNode[String]`) as a parameter and returns the evaluation of the tree as a `Double`
- The operators can be `"*"`, `"/"`, `"+"`, and `"-"`
- You can assume that all Nodes storing numbers are well-formed (i.e. If a node's `String` is not one of the 4 operators, you can call `.toDouble` on it to convert it to a `Double`)

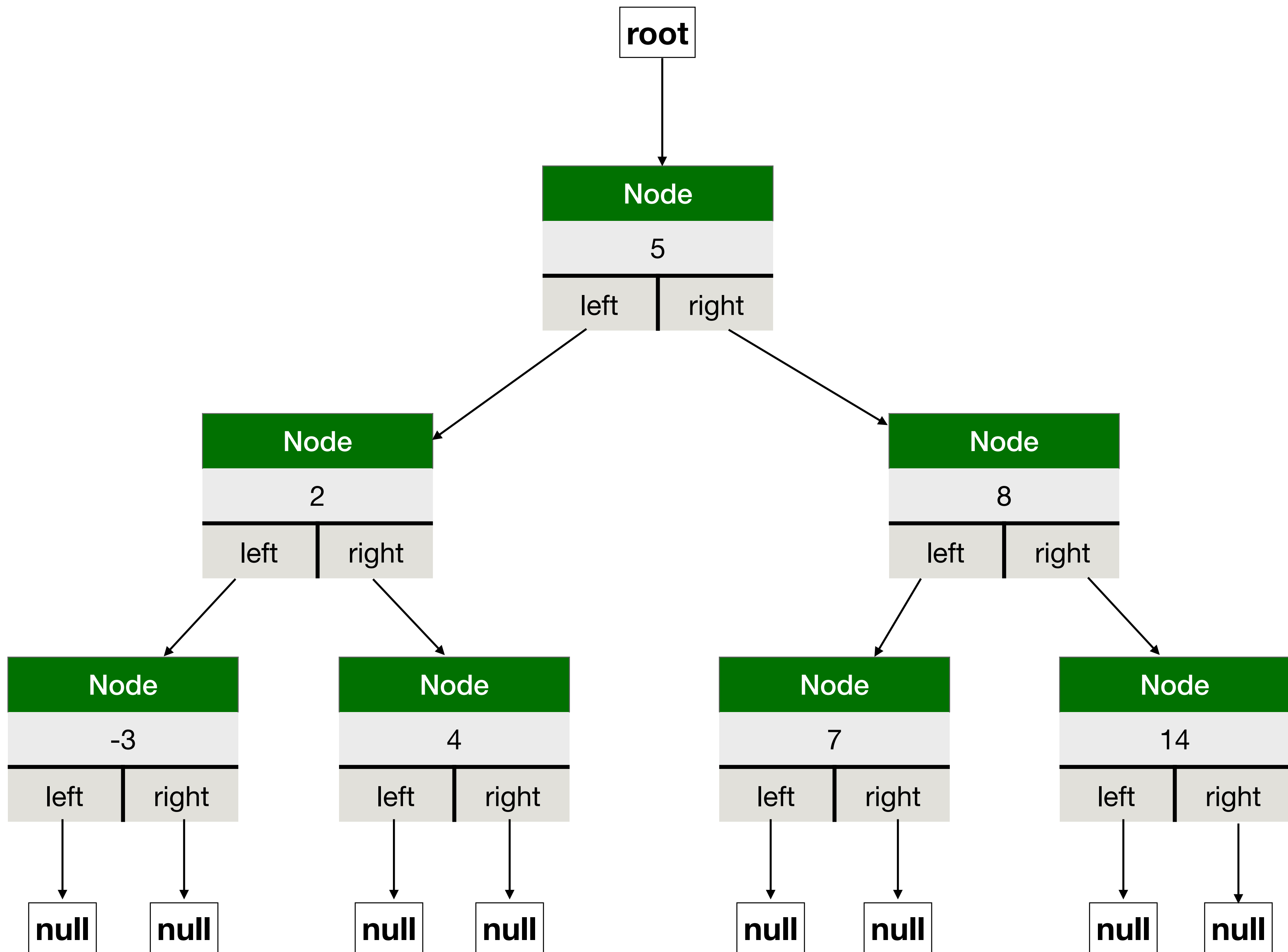
Testing: In a package named `"tests"` create a class named `"TestExpressionTree"` as a test suite that tests the functionality above (Only write tests with valid expression trees as input)

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 compute with trees?
 - With linked lists we wrote several methods that recursively visited the next node to visit every value
- With trees, how do we visit both children of each node?
 - Recursive call on both child nodes
- 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)  
  }  
}
```

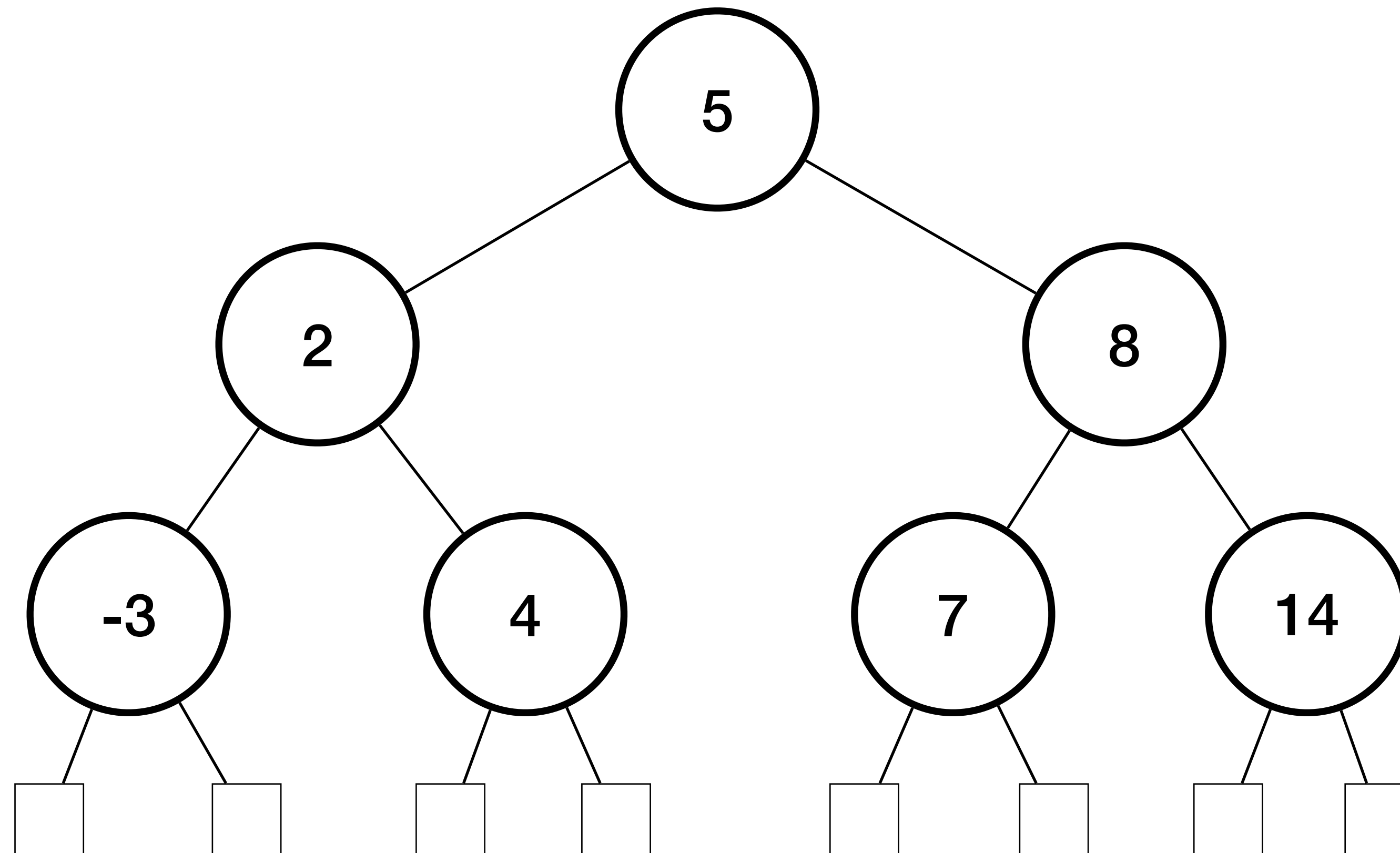
```
preOrderTraversal(root, println)
```

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)  
  }  
}
```

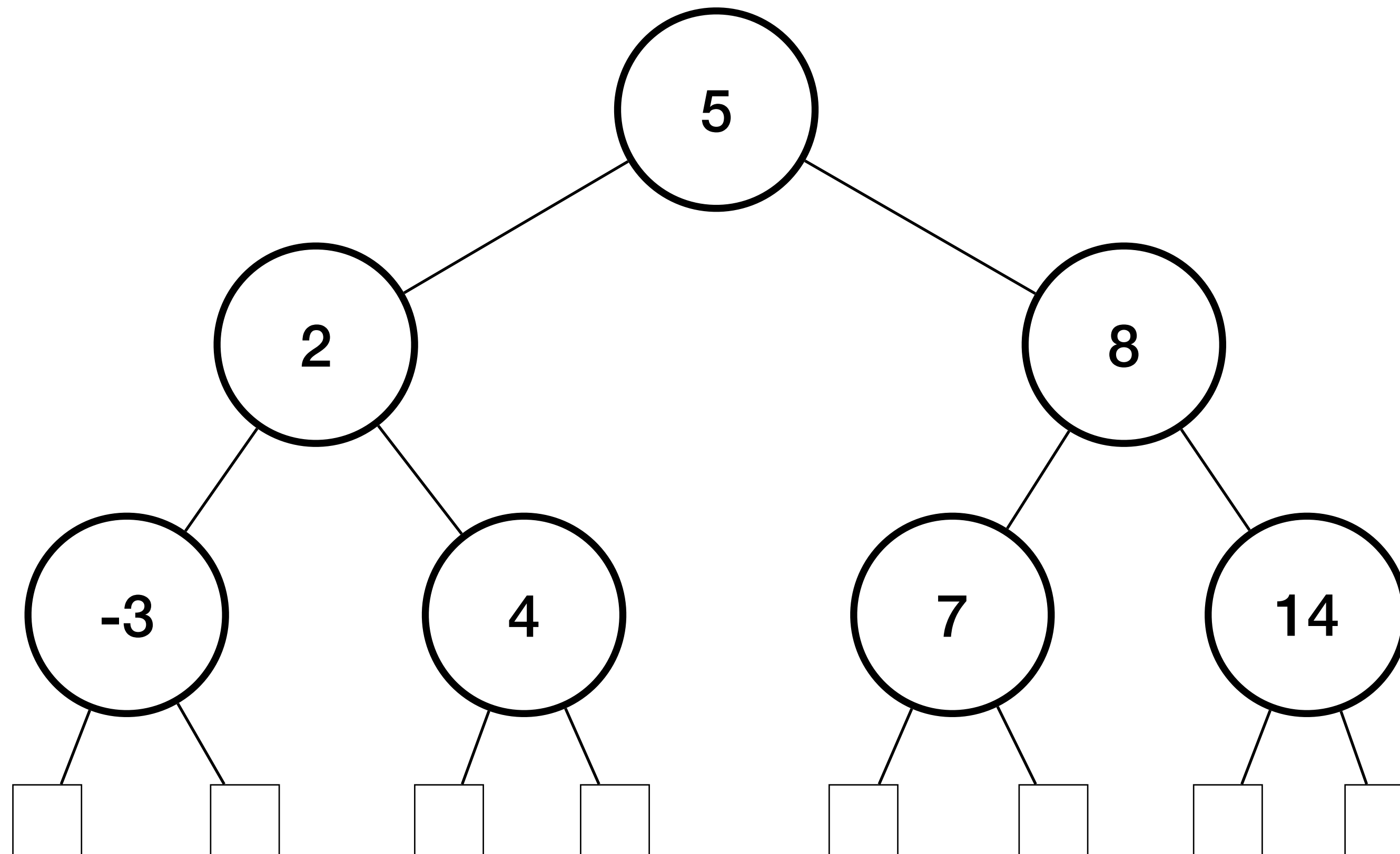
```
postOrderTraversal(root, println)
```

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

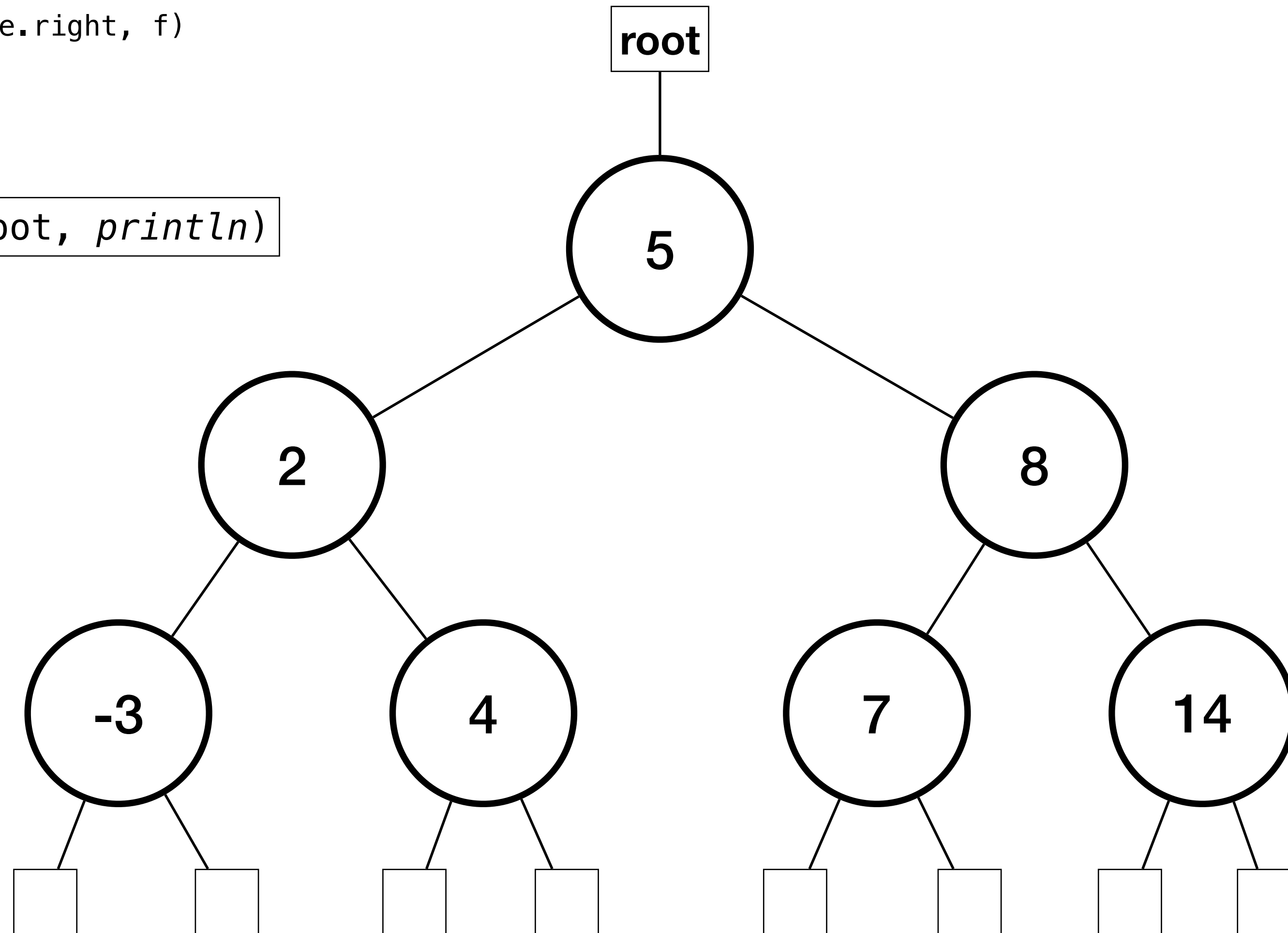
```
def inOrderTraversal[A](node: BinaryTreeNode[A], f: A => Unit): Unit = {  
  if (node != null) {  
    inOrderTraversal(node.left, f)  
    f(node.value)  
    inOrderTraversal(node.right, f)  
  }  
}  
  
def preOrderTraversal[A](node: BinaryTreeNode[A], f: A => Unit): Unit = {  
  if (node !=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

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:

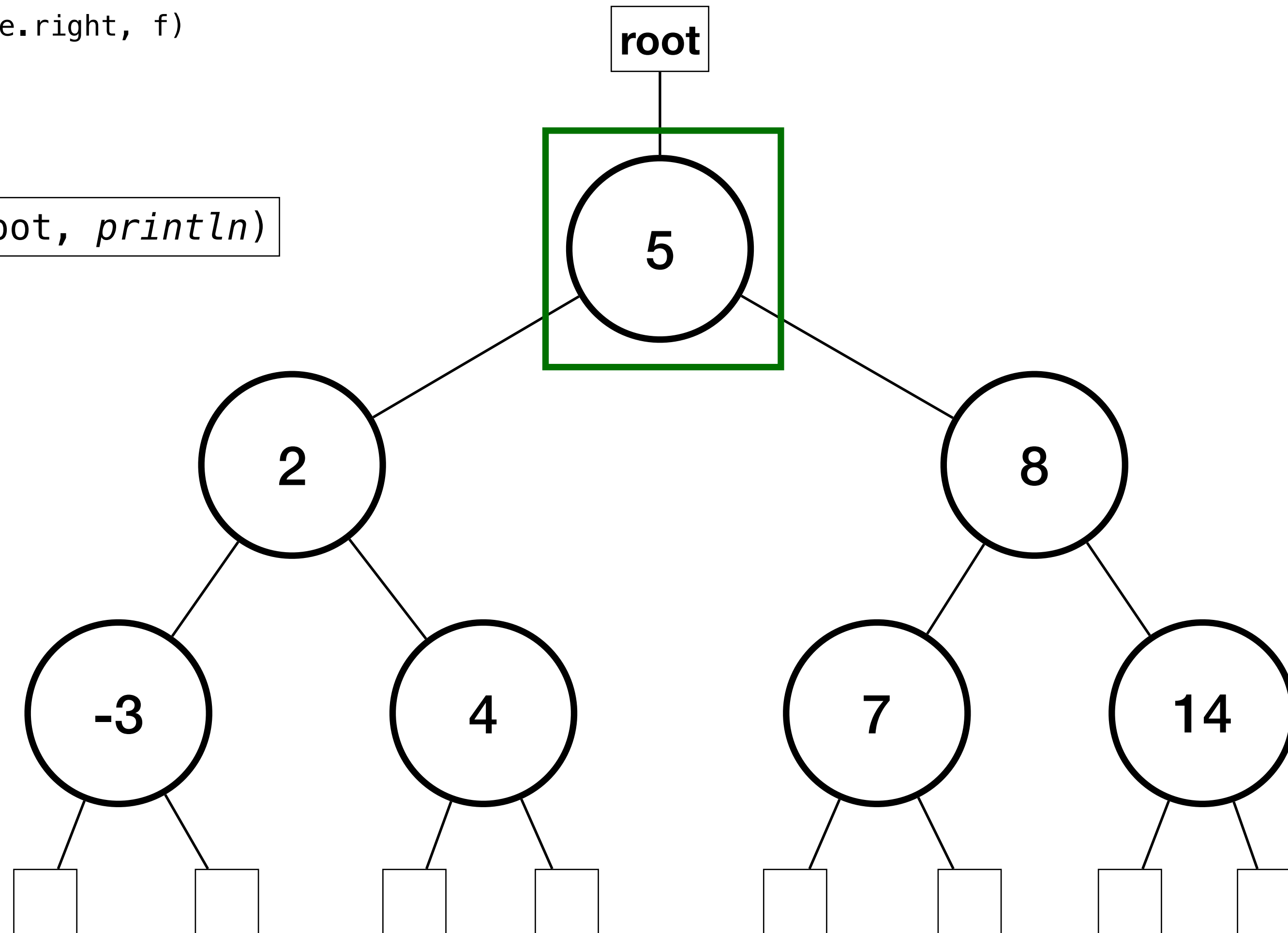


Traversals

```
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:

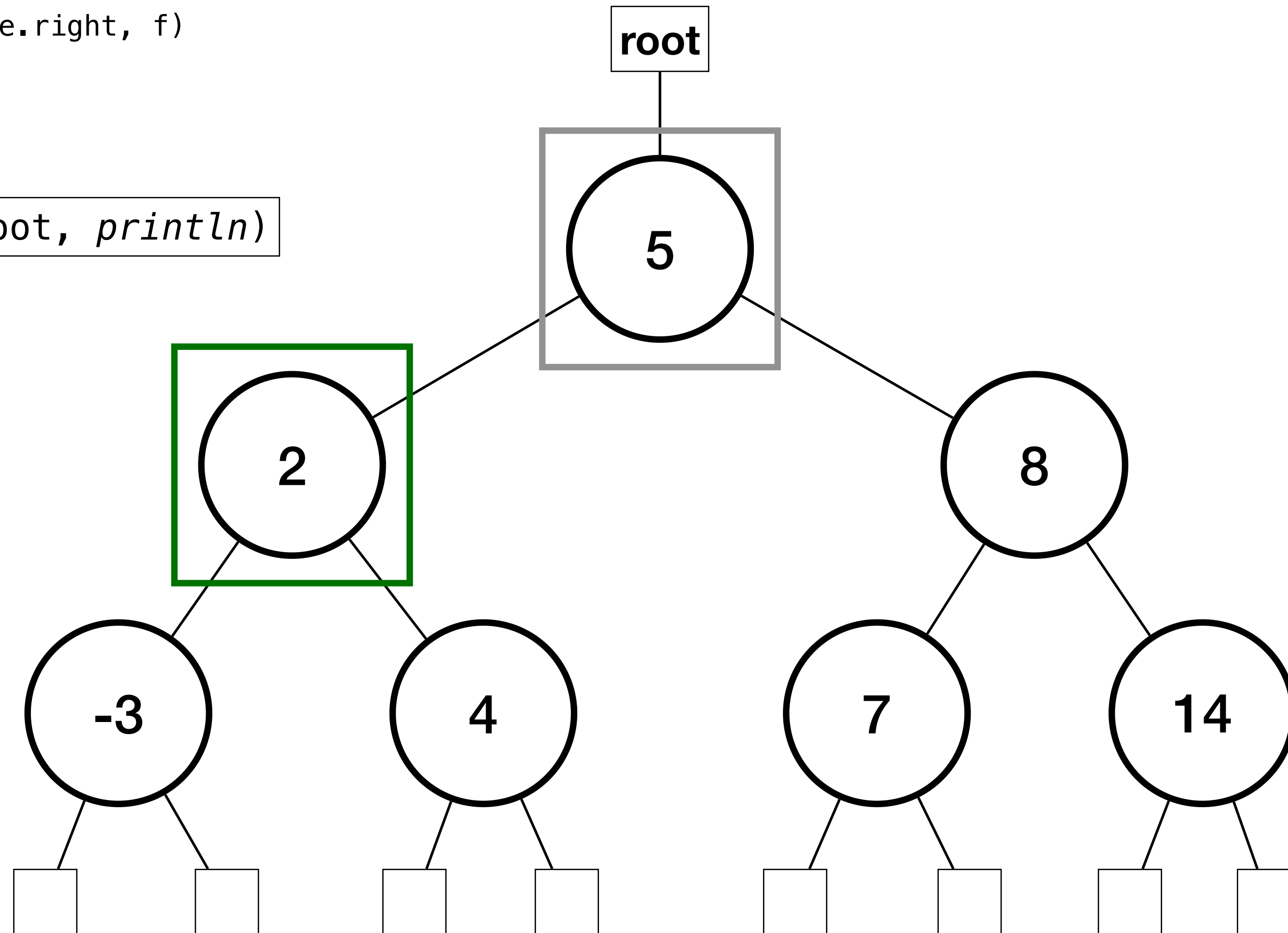


Traversals

```
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:

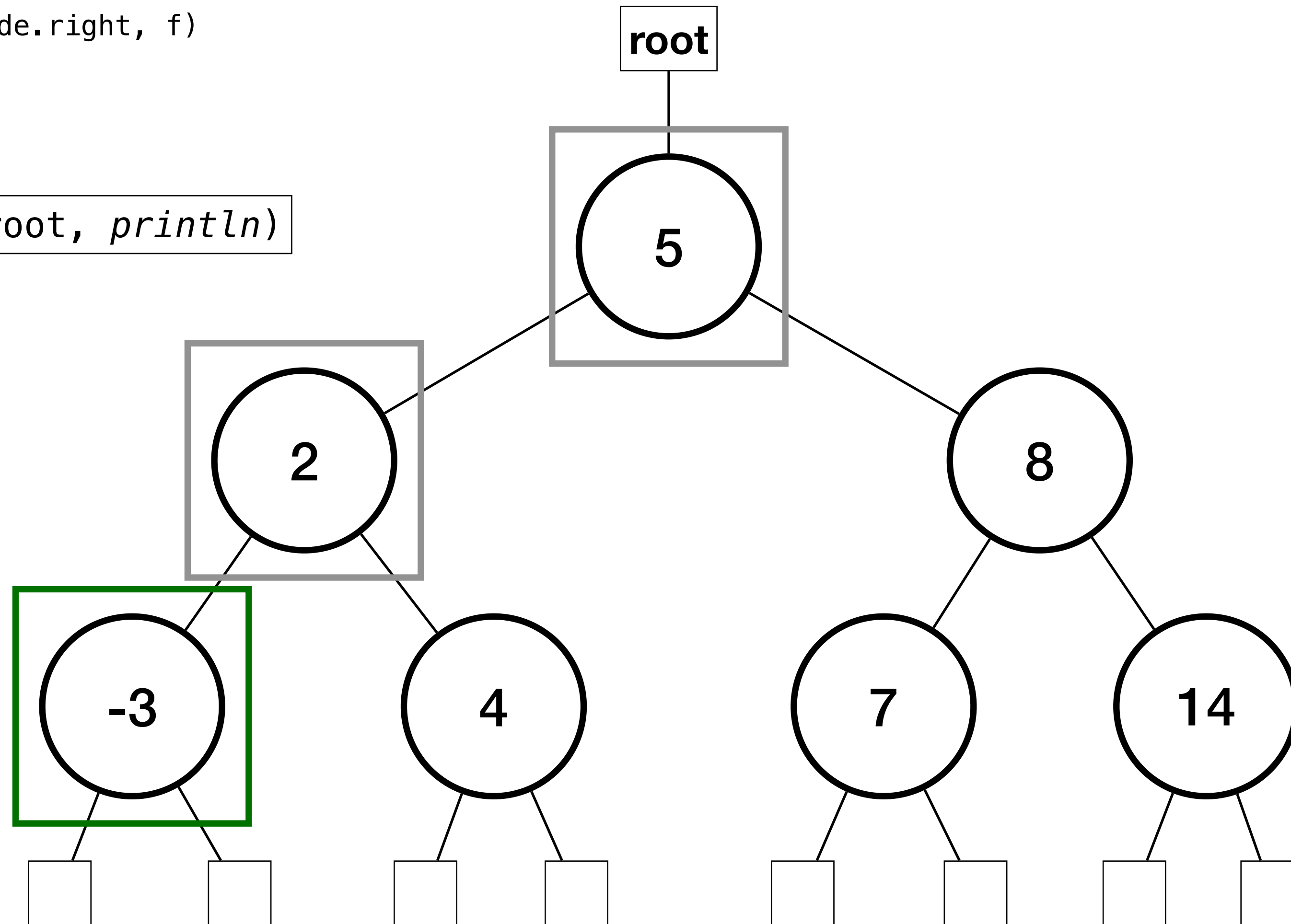


Traversals

```
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:

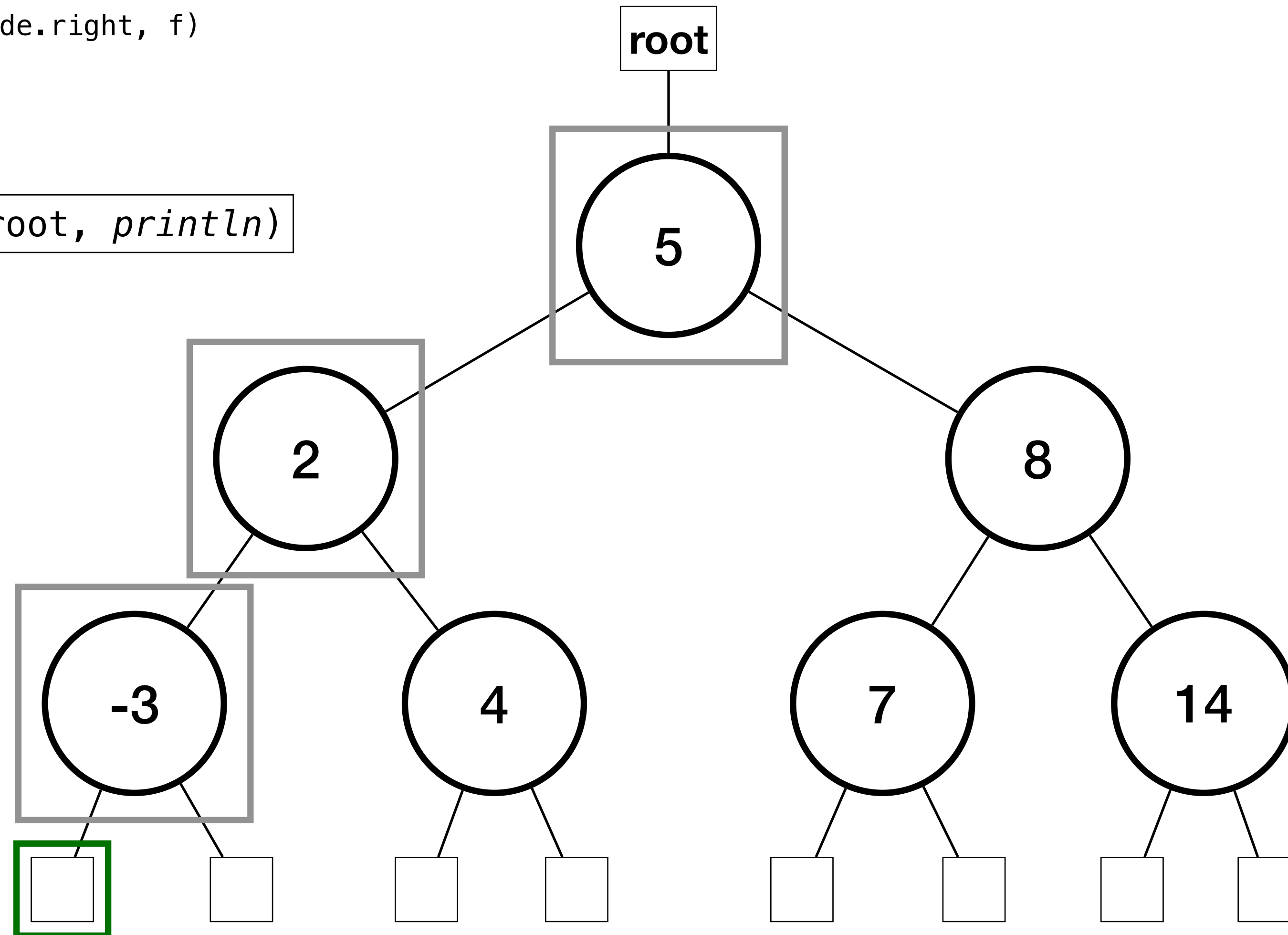


Traversals

```
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:

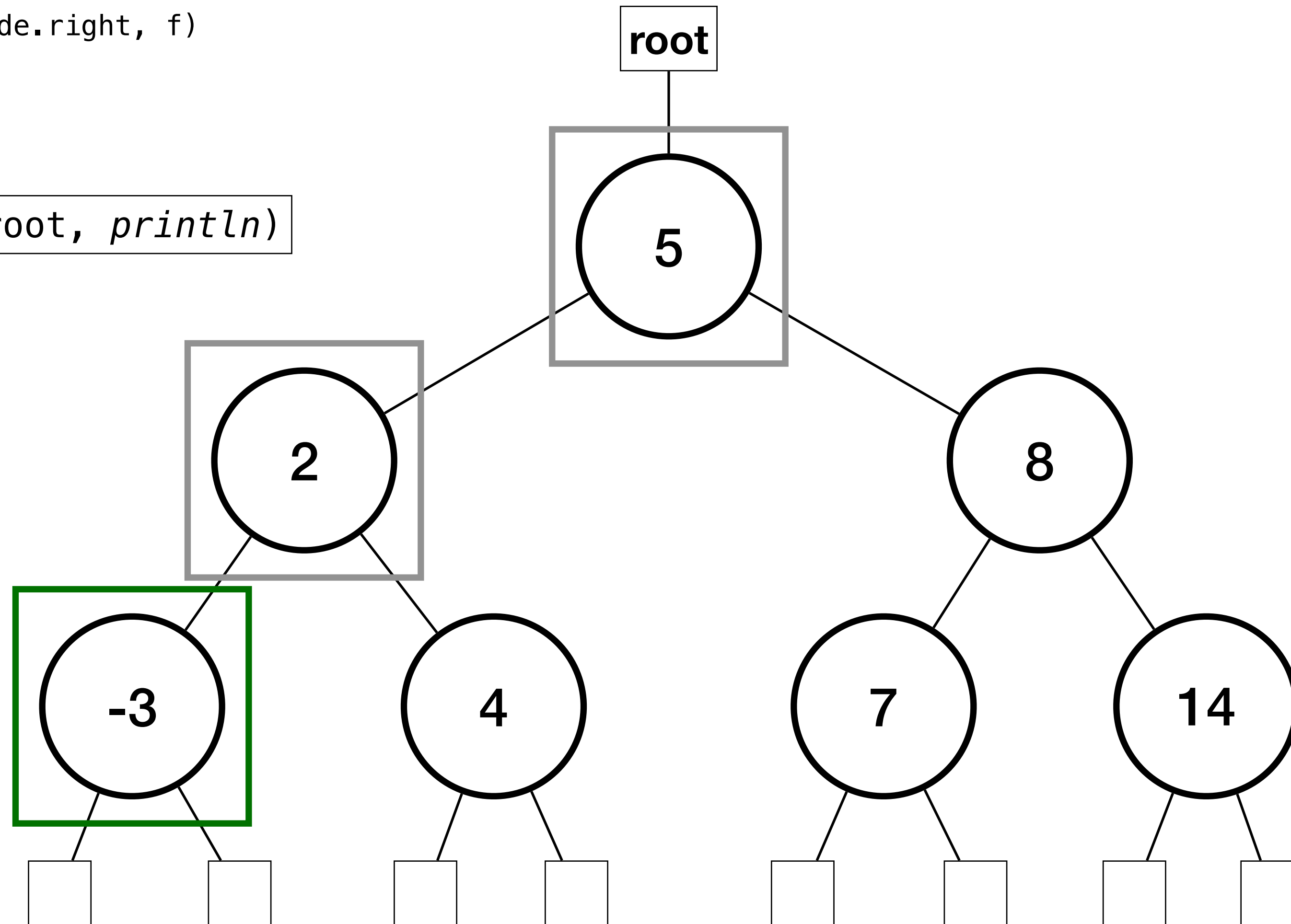


Traversals

```
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

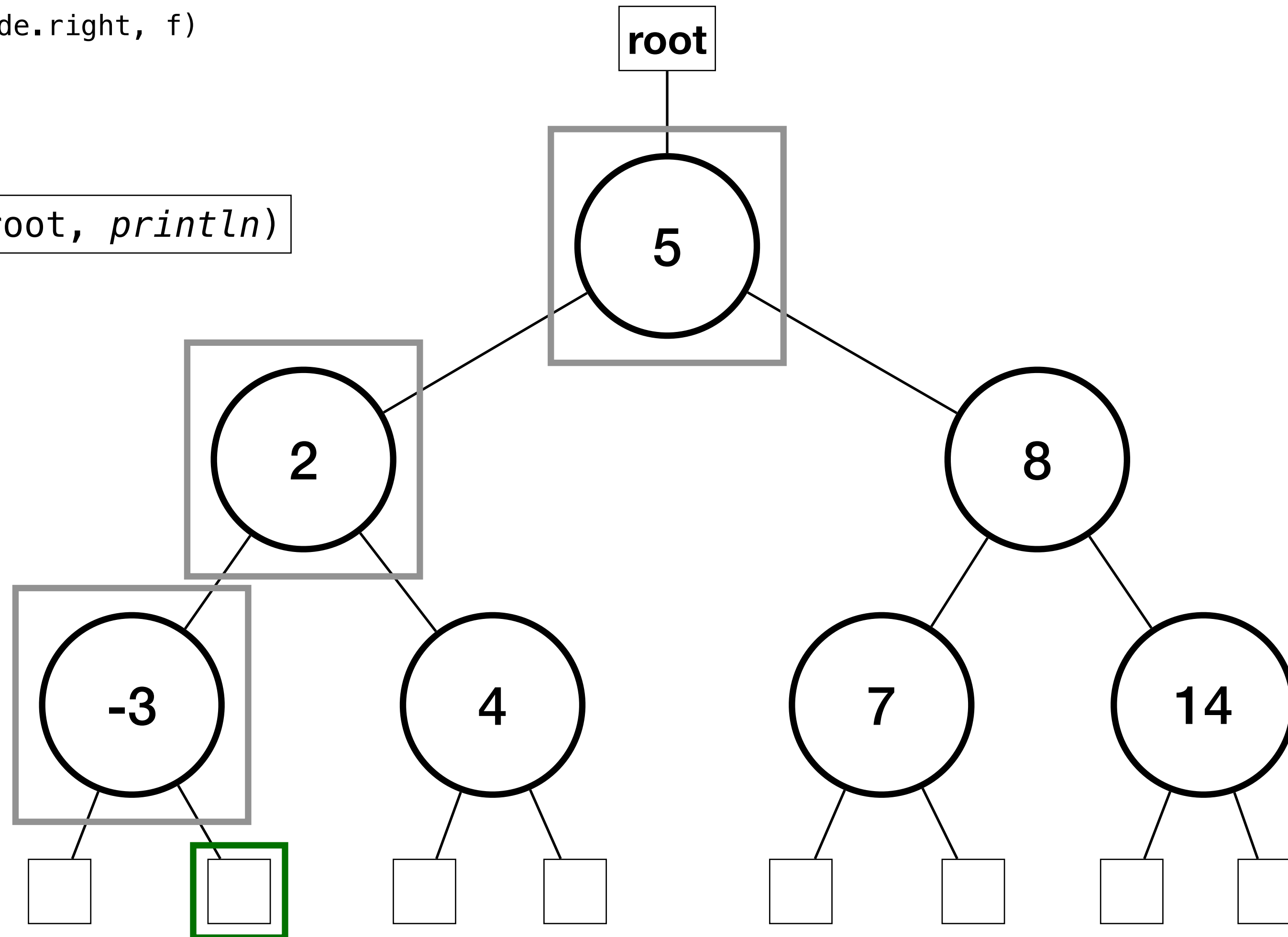


Traversals

```
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

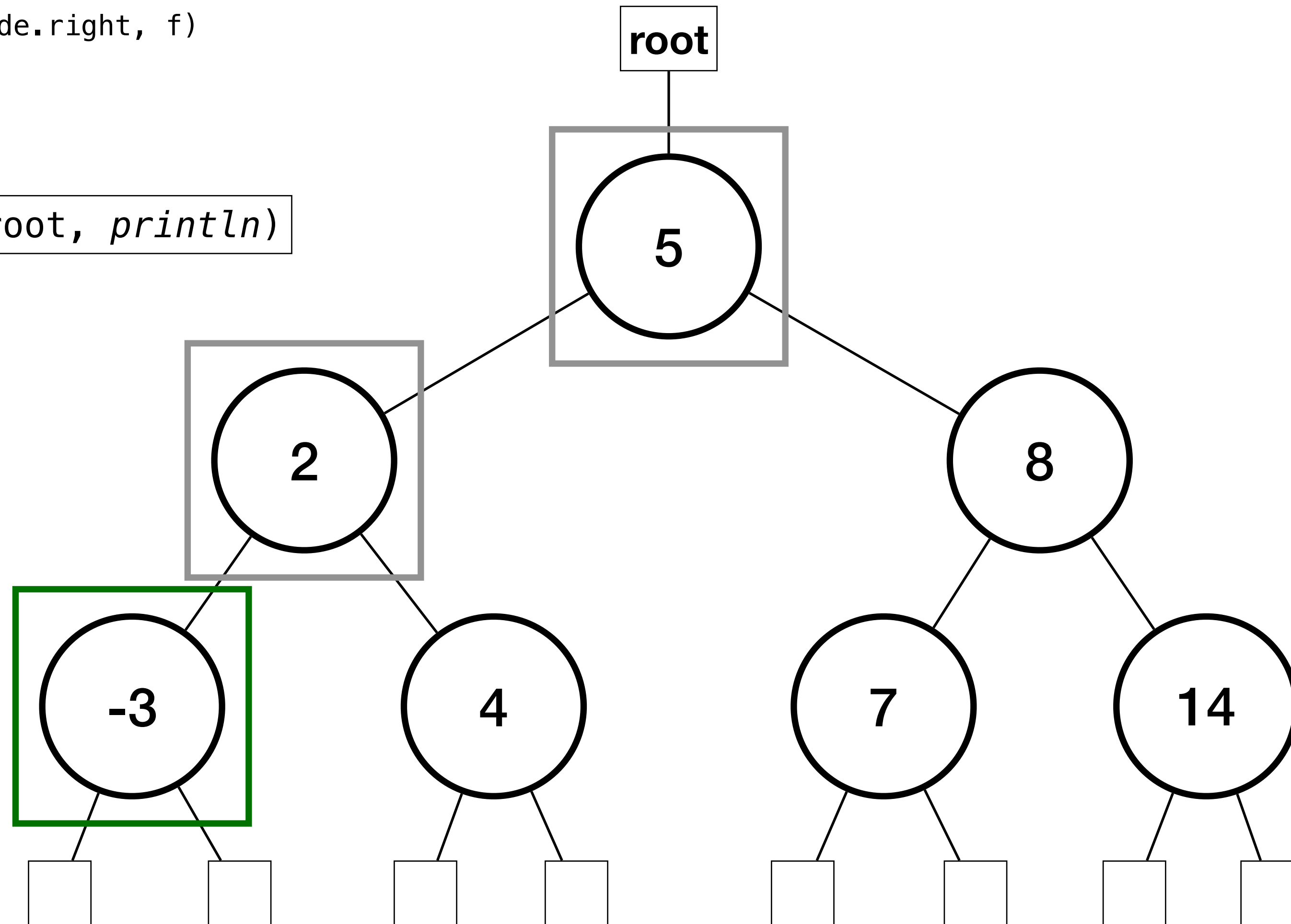


Traversals

```
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



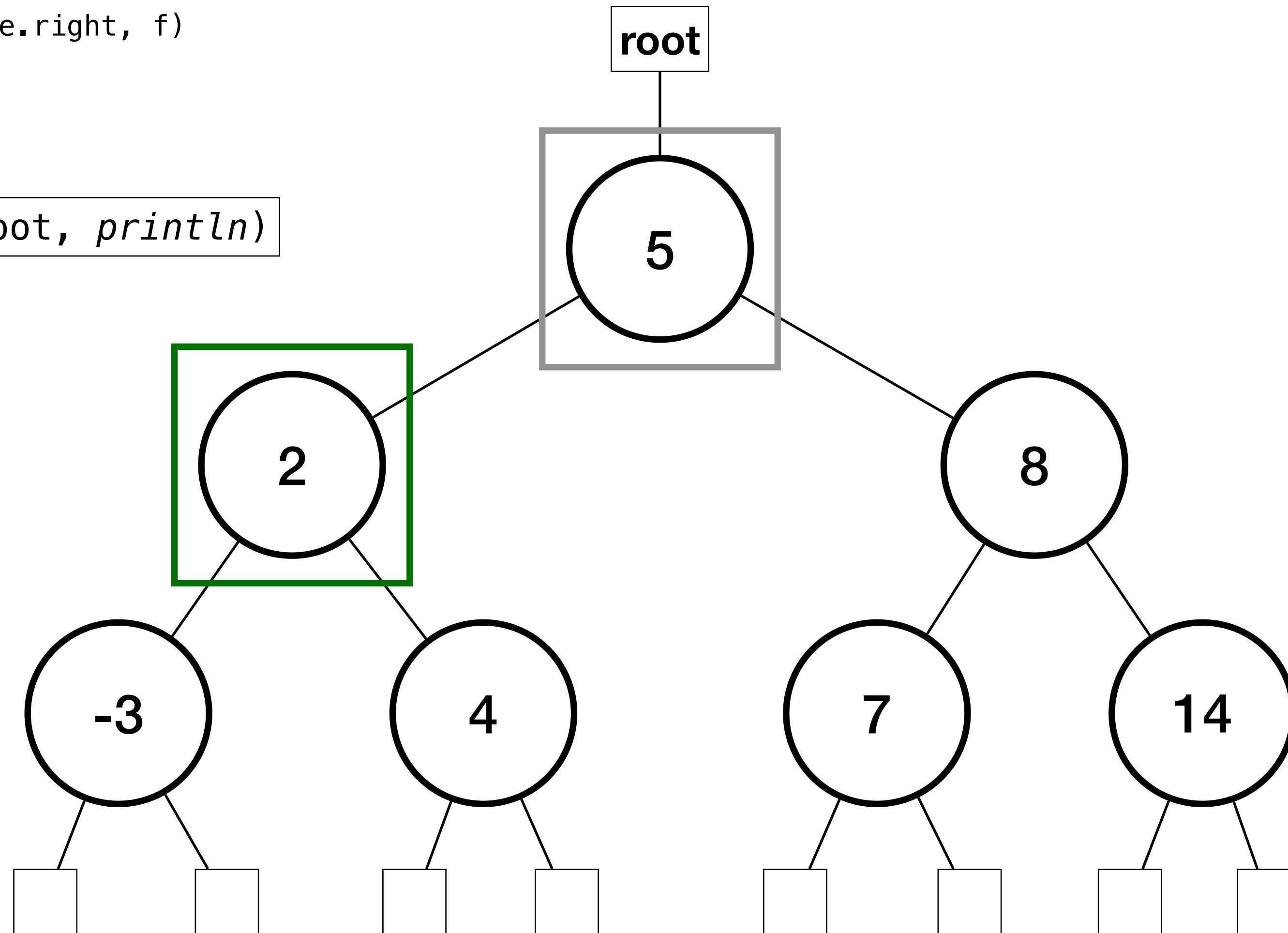
Traversals

```
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**



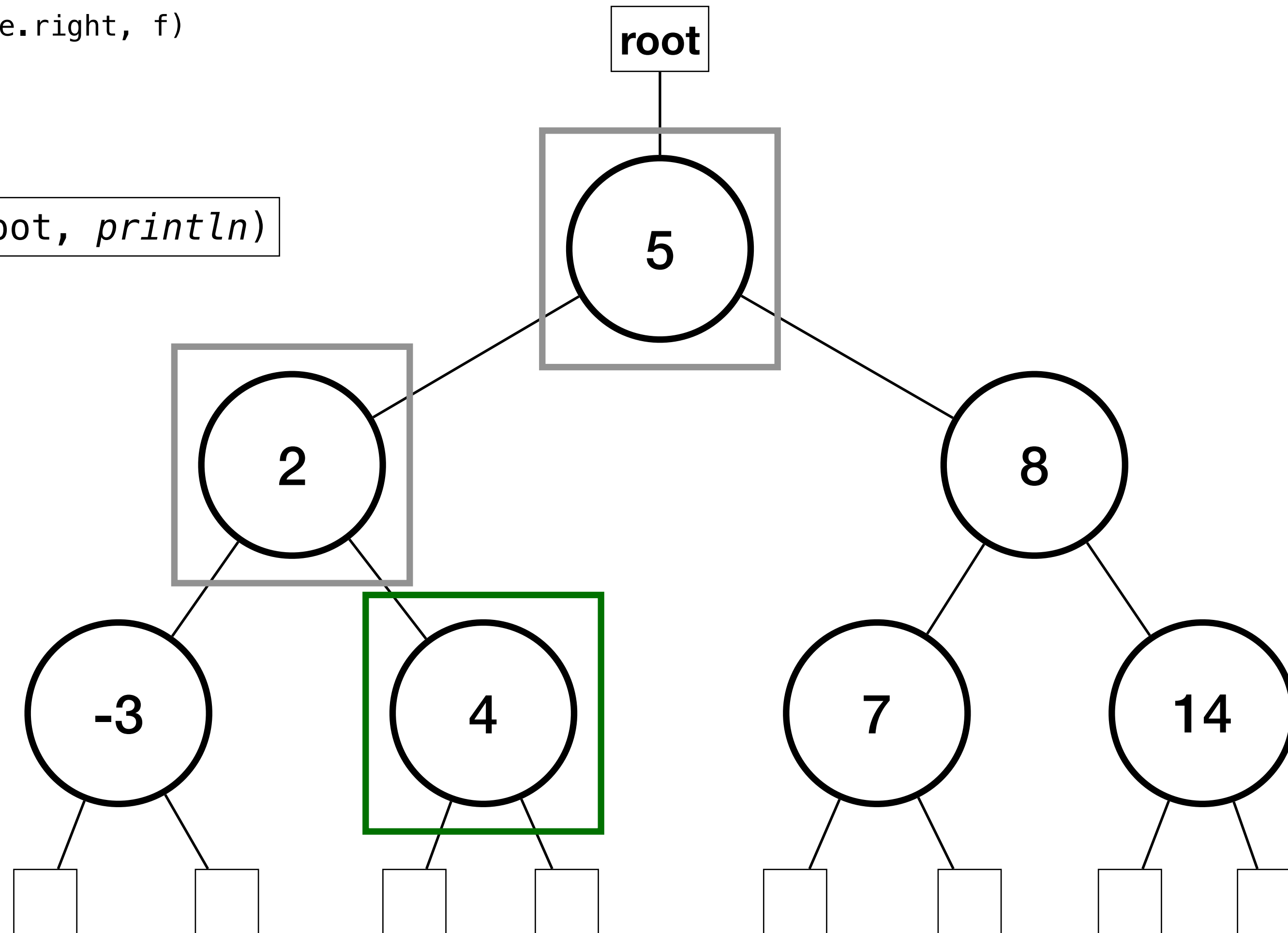
Traversals

```
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)
  }
}
```

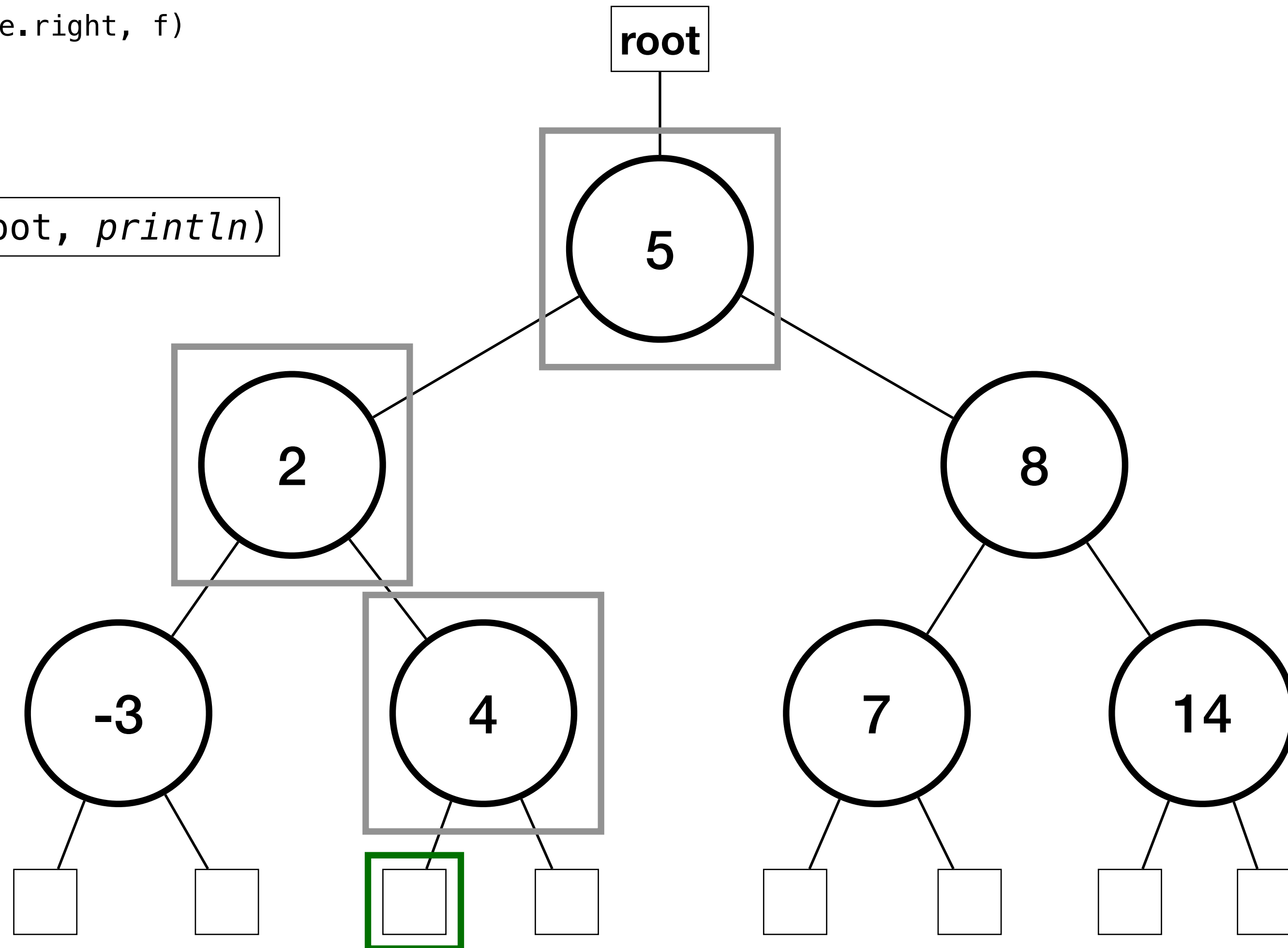


A diagram showing a single node labeled "root" in a box, with a vertical line extending downwards from the bottom center of the box, representing the start of a tree structure.

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

Printed:

-3
2



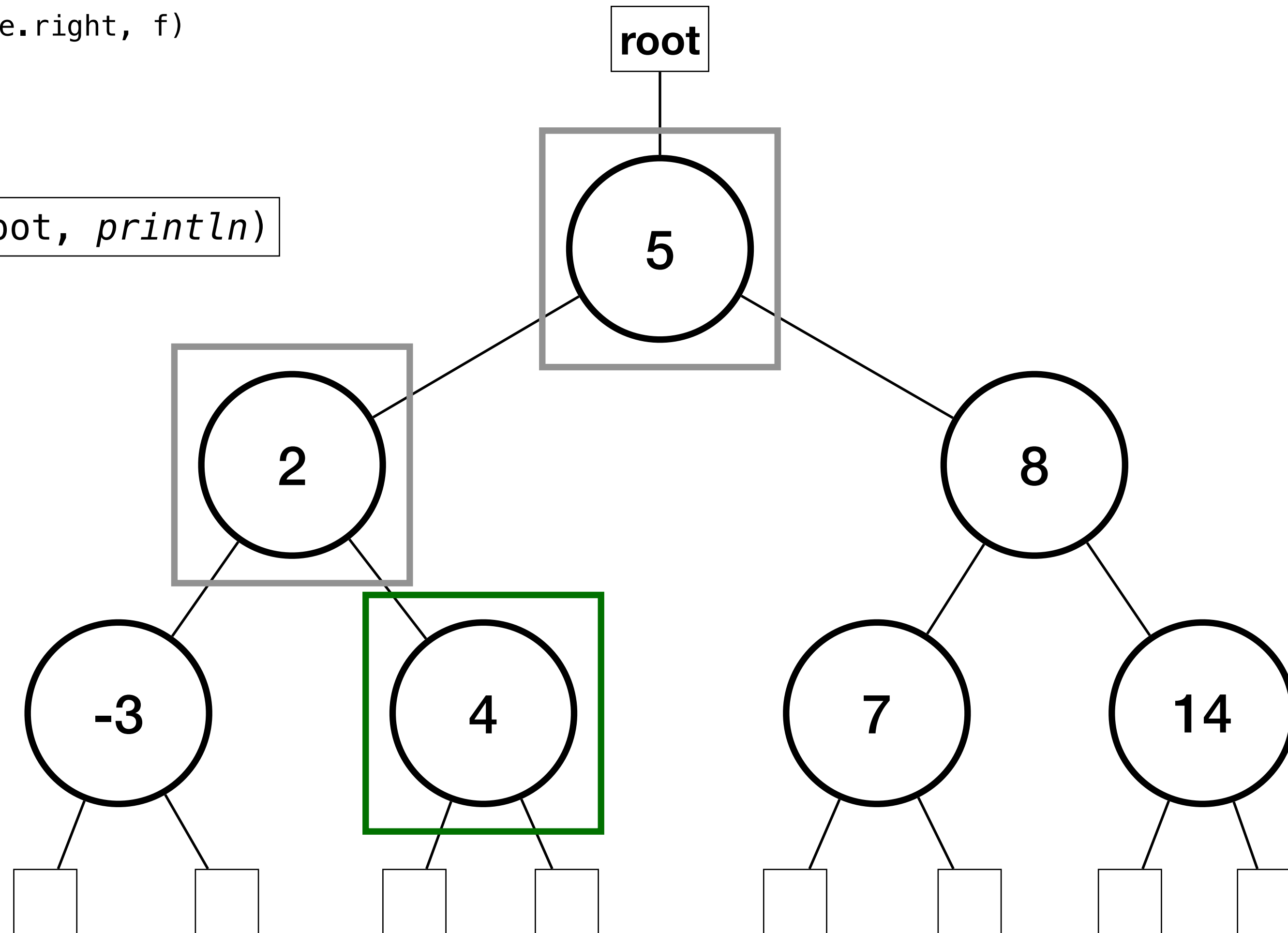
Traversals

```
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



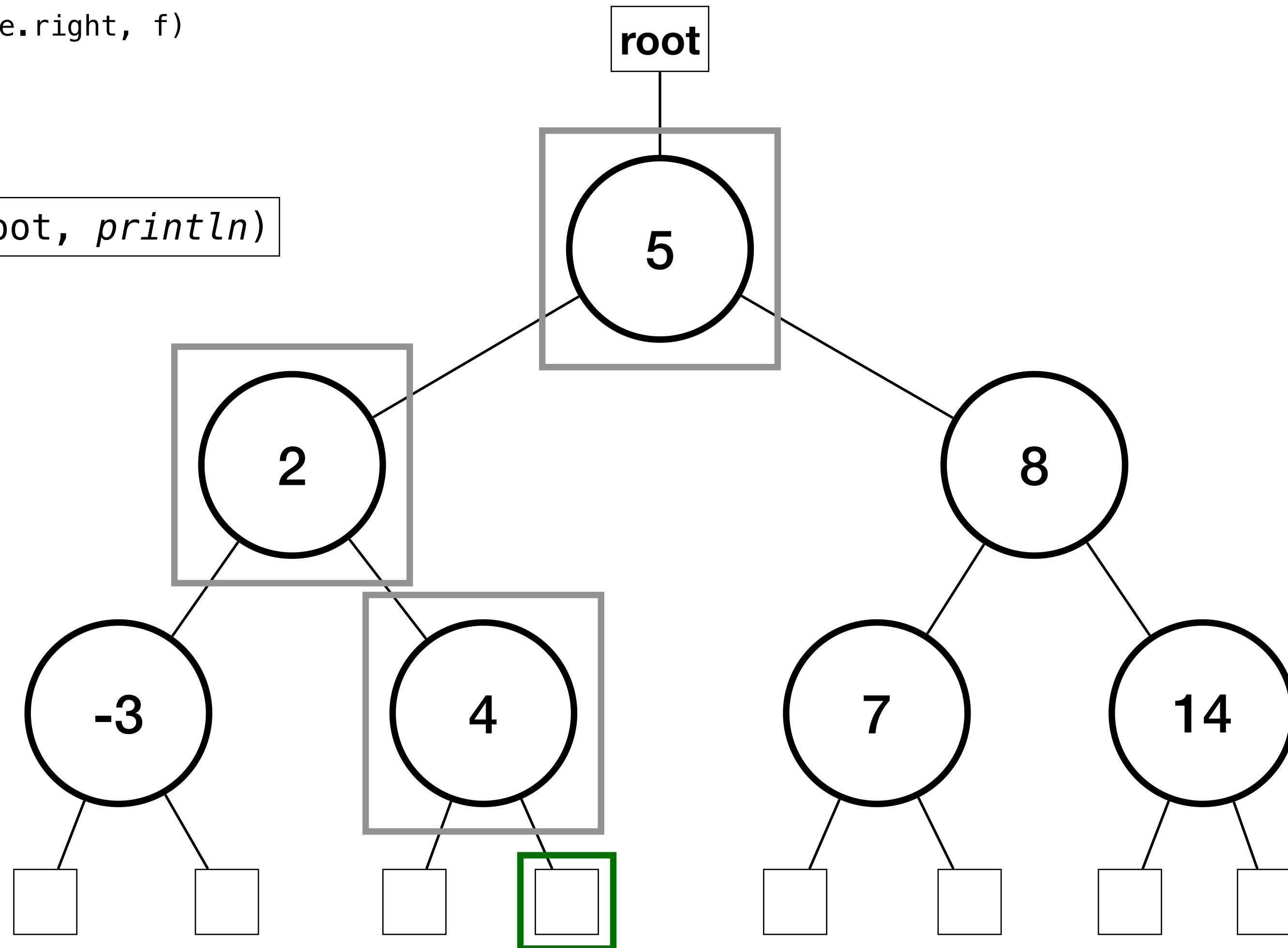
Traversals

```
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



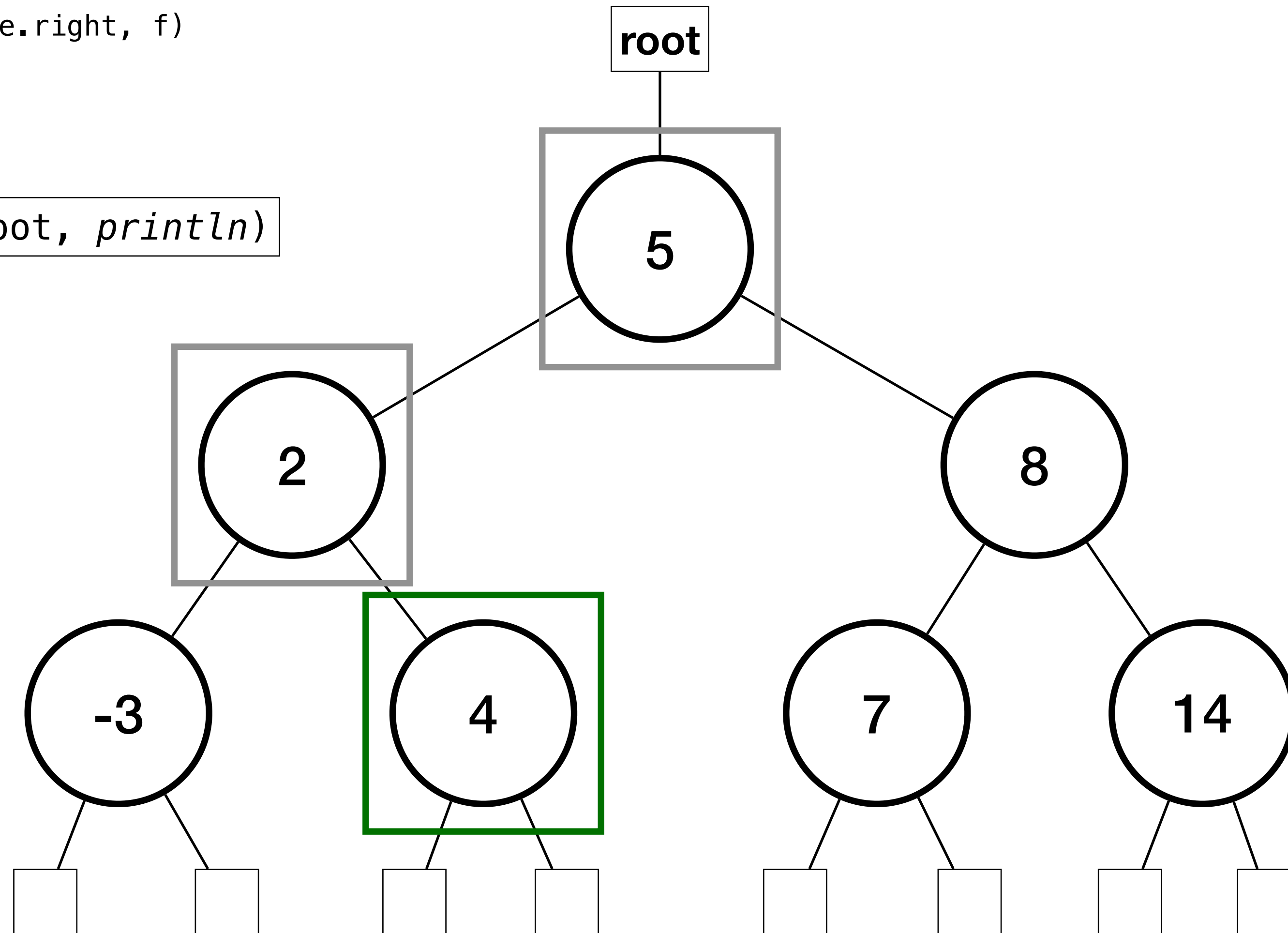
Traversals

```
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



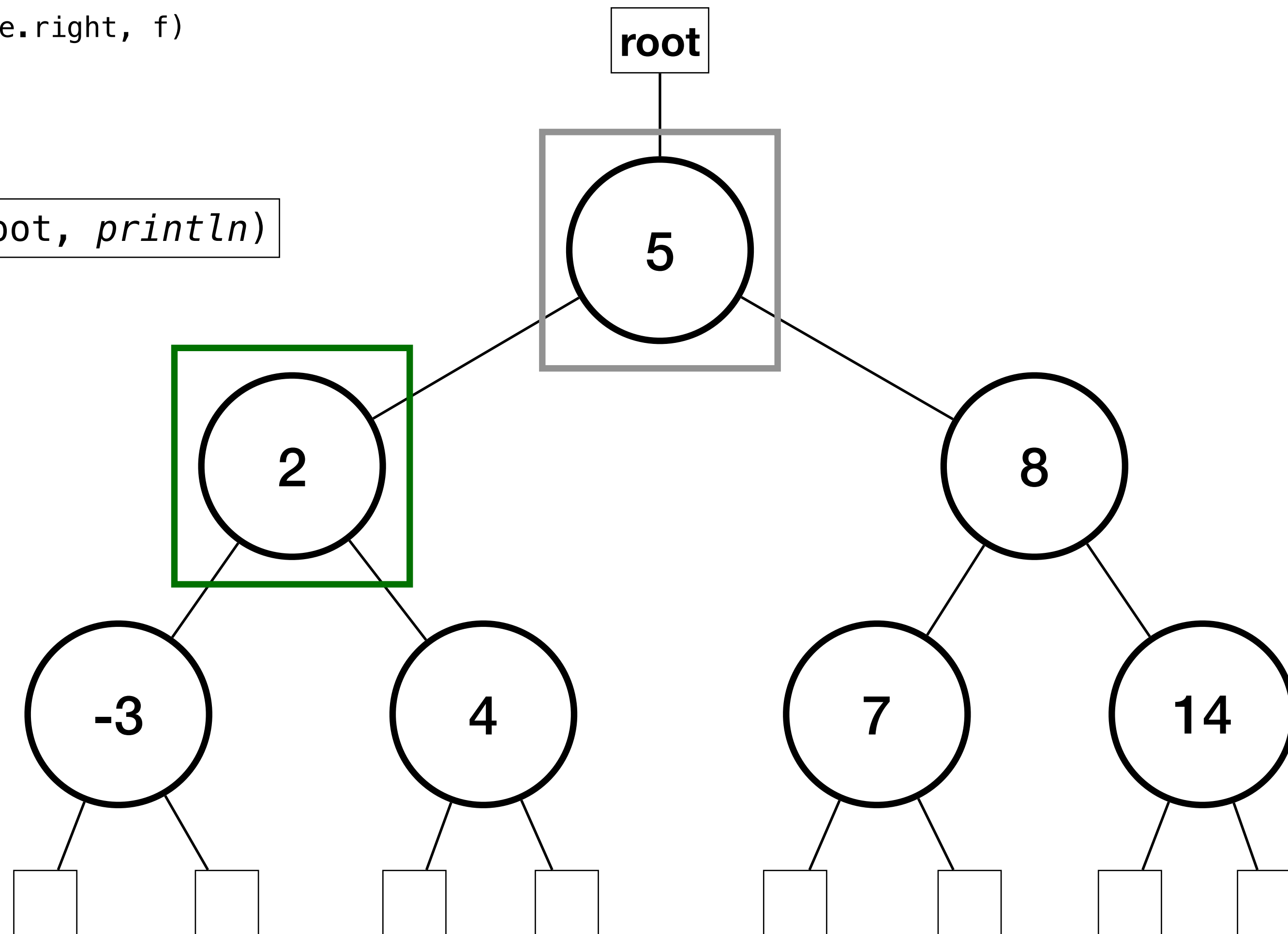
Traversals

```
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



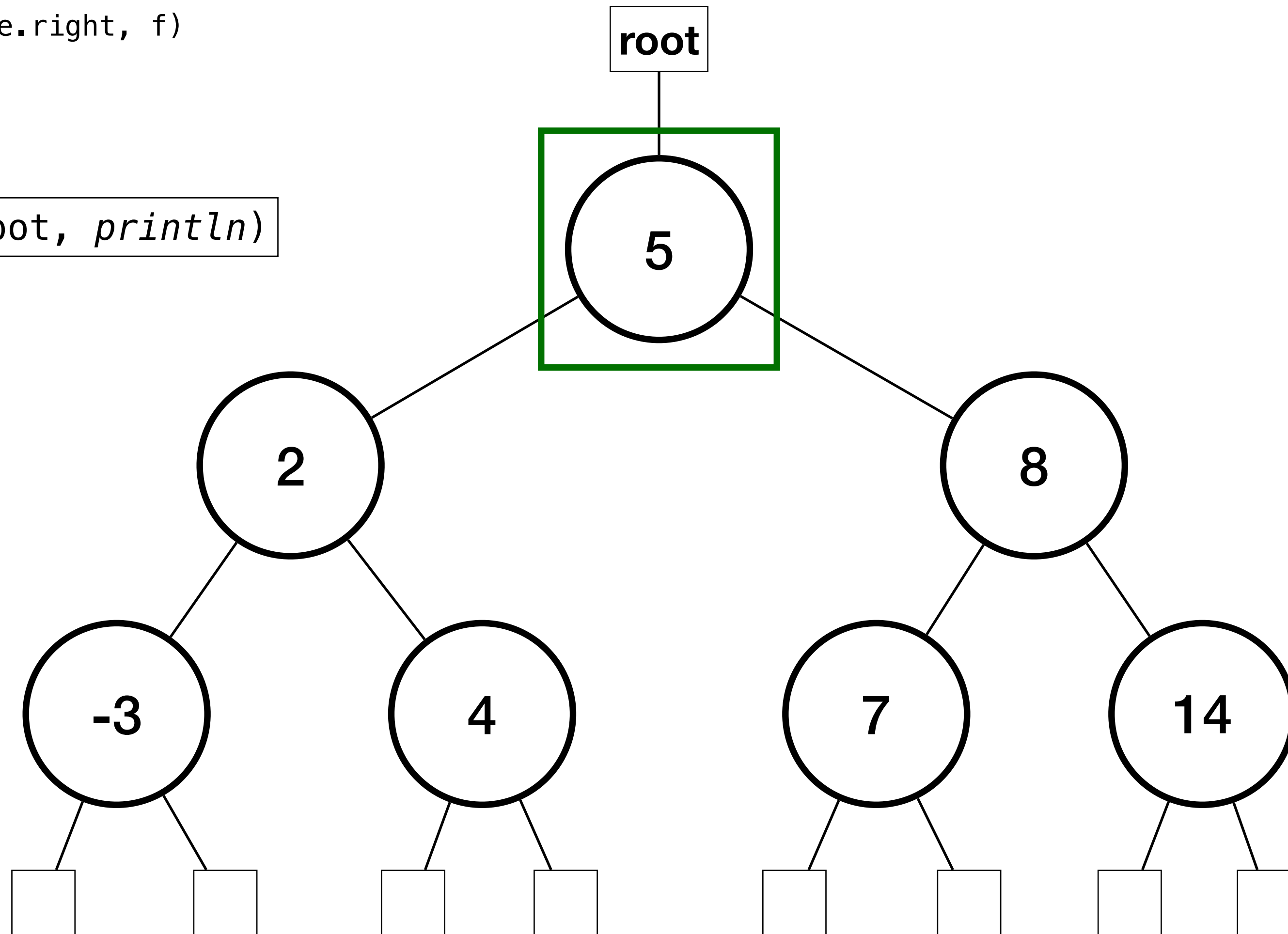
Traversals

```
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



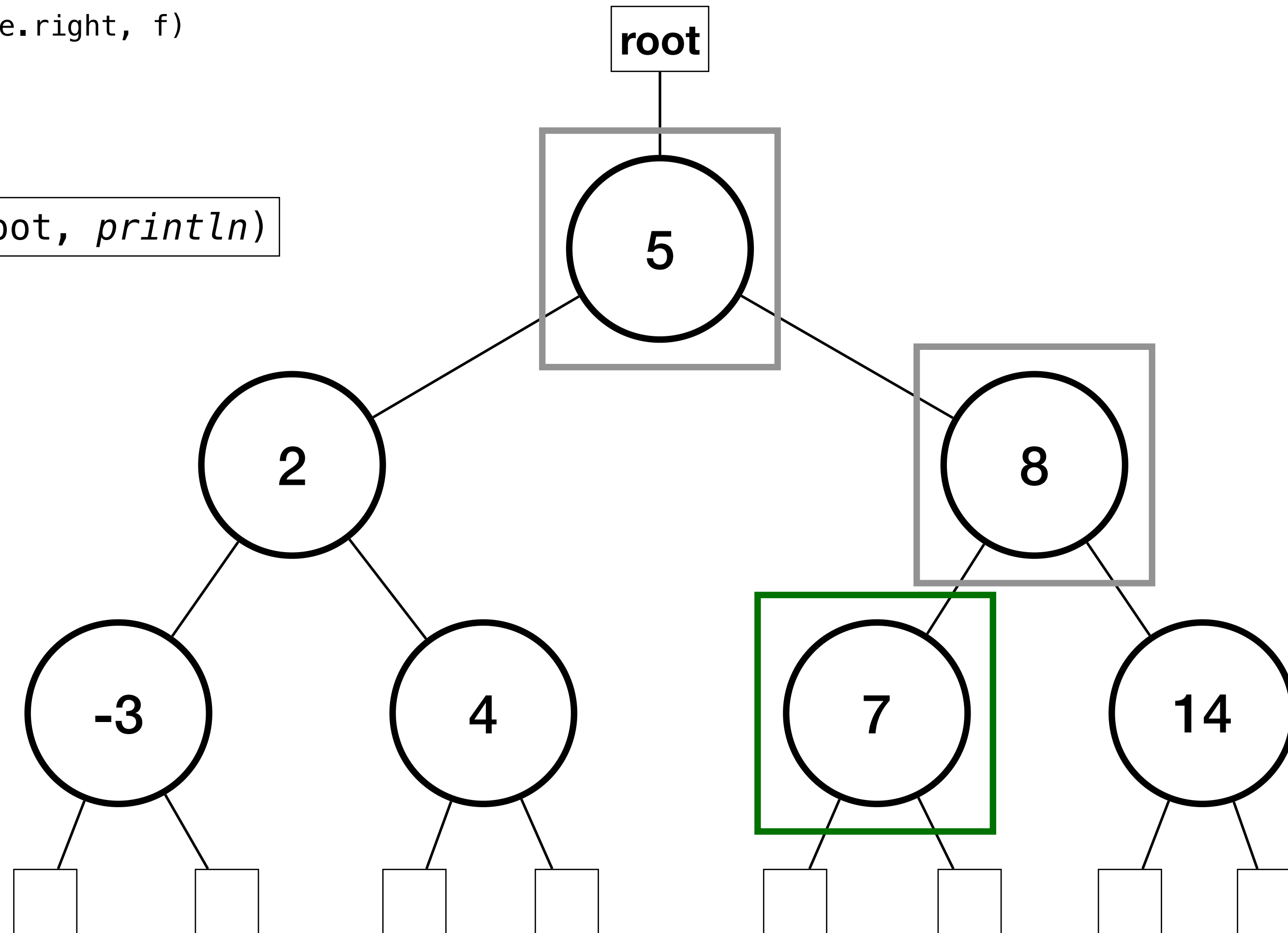
Traversals

```
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
7
14



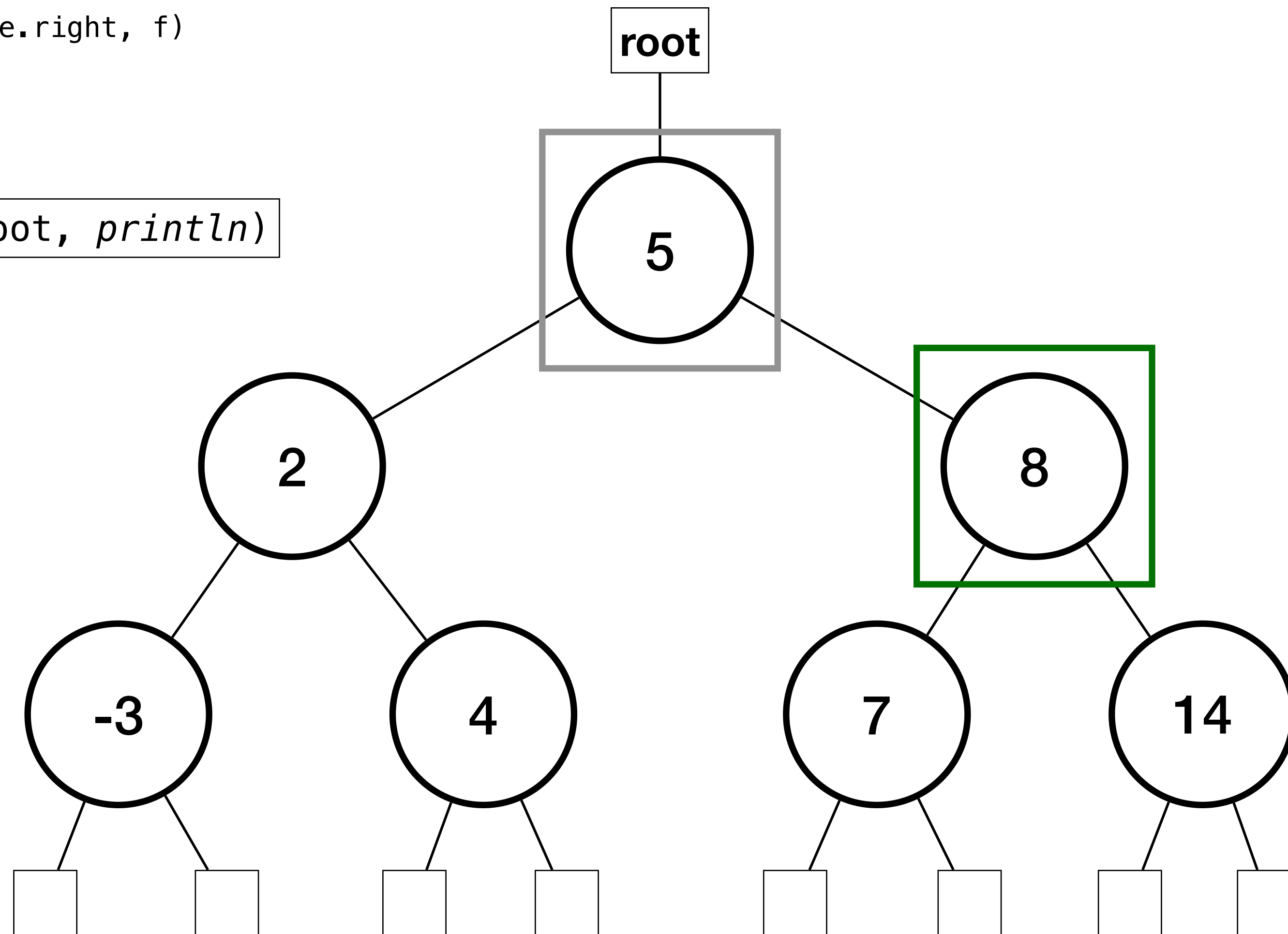
Traversals

```
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



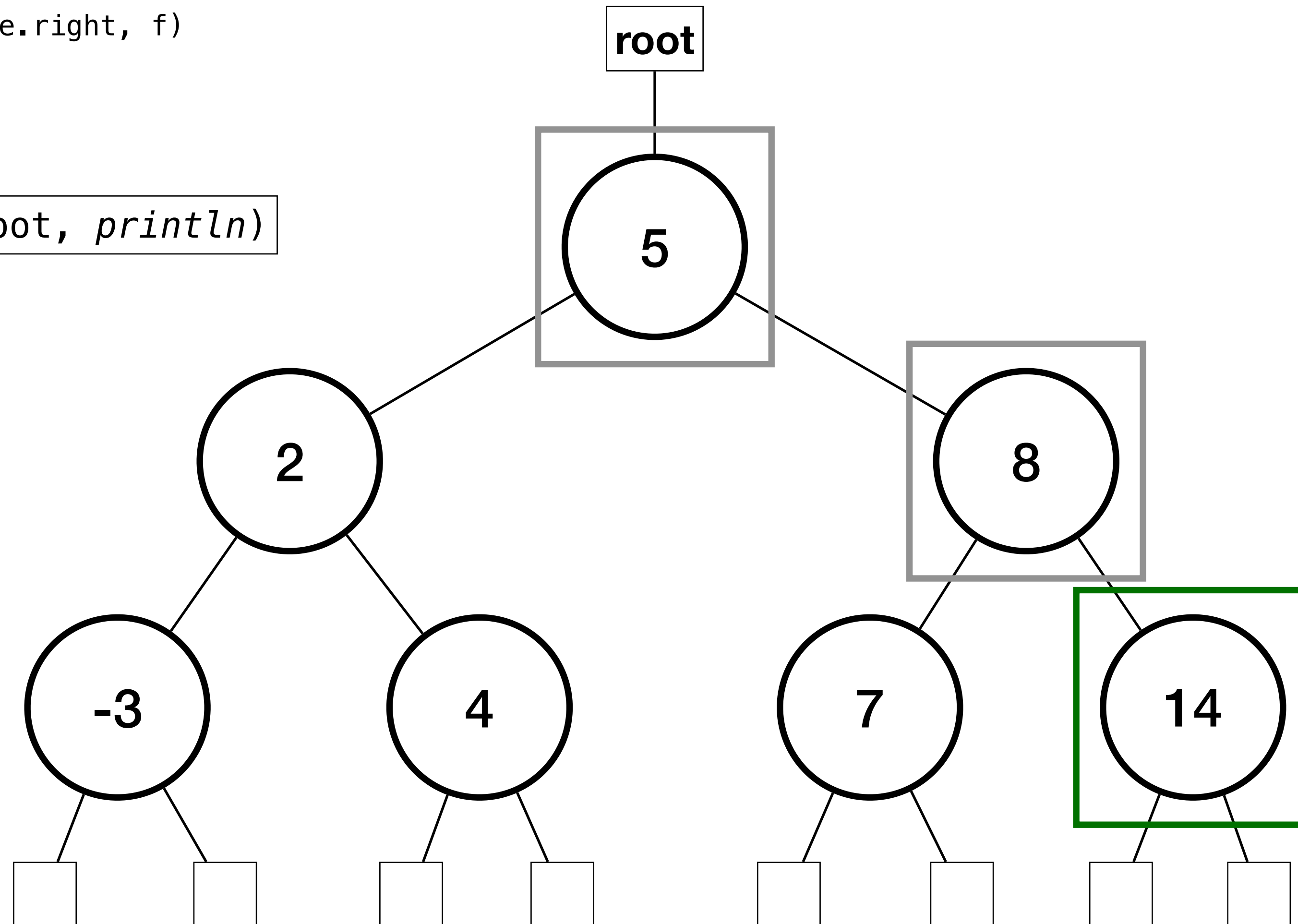
Traversals

```
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



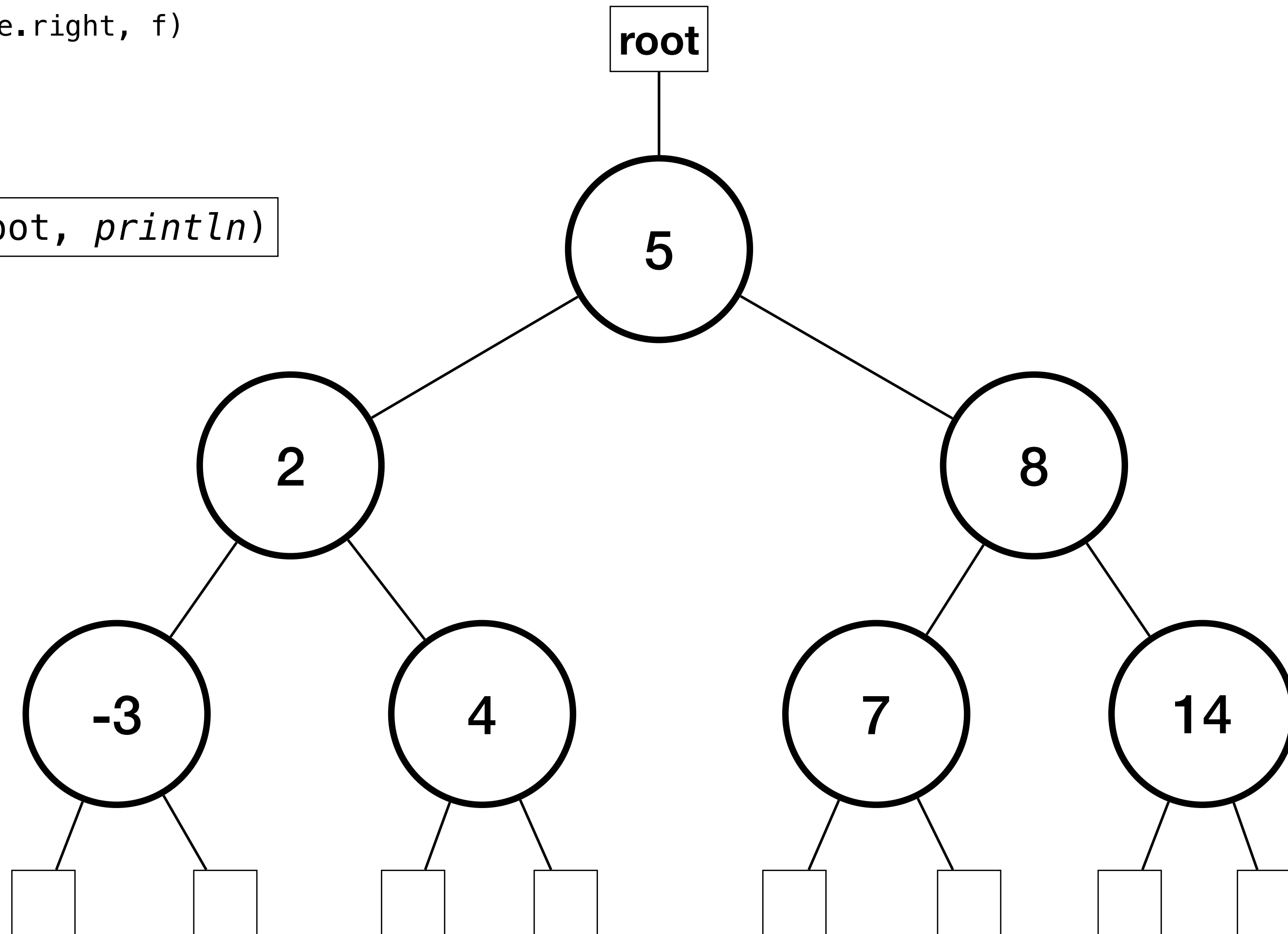
Traversals

```
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**



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**

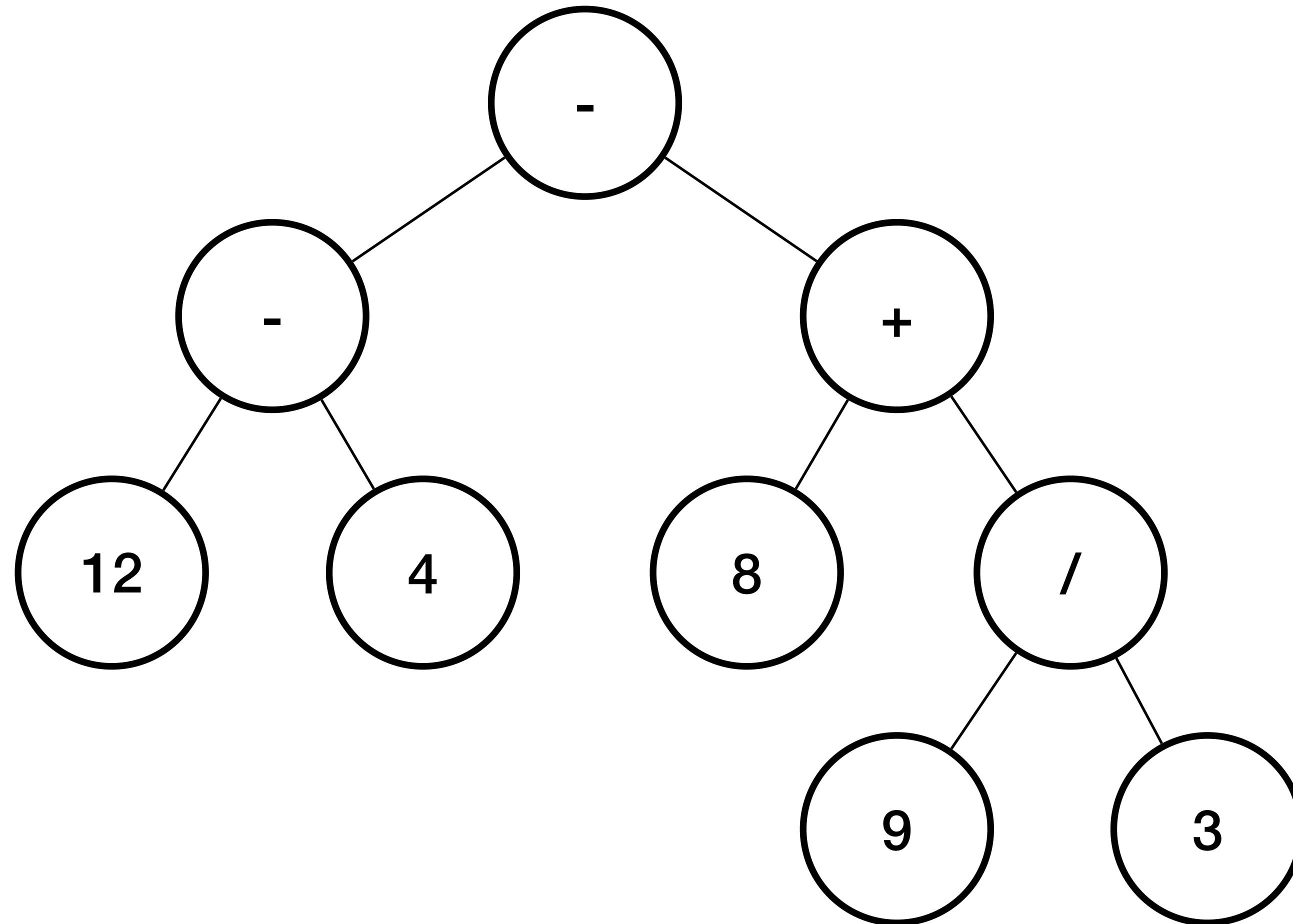
Expression Trees

Expression Trees

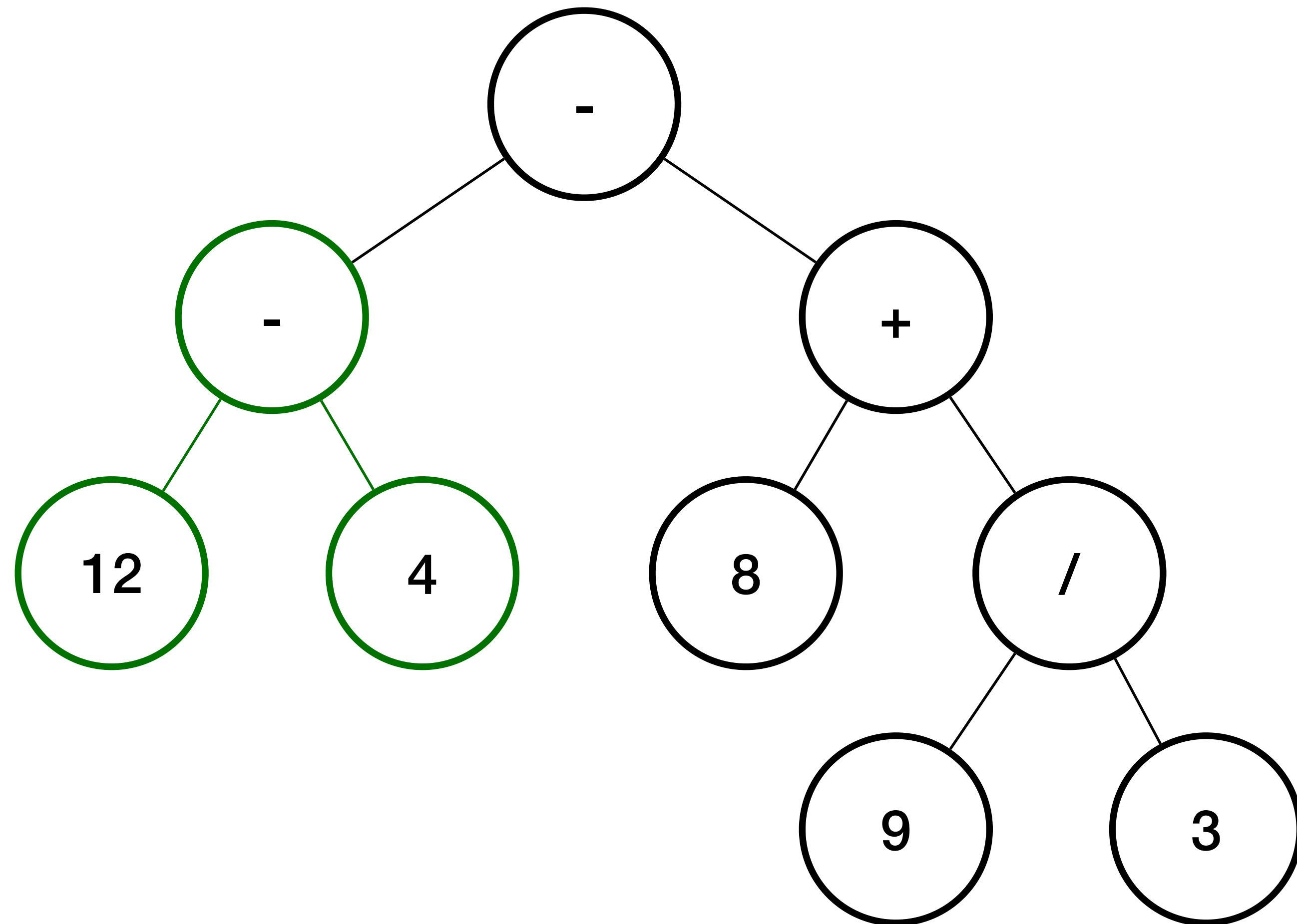
- Represent an expression as a binary tree
- Nodes can be
 - Operands
 - Operators
- An operand is a literal value
- An operator is evaluated by using its left and right children as operands
 - Operands can be operators

Expression Tree

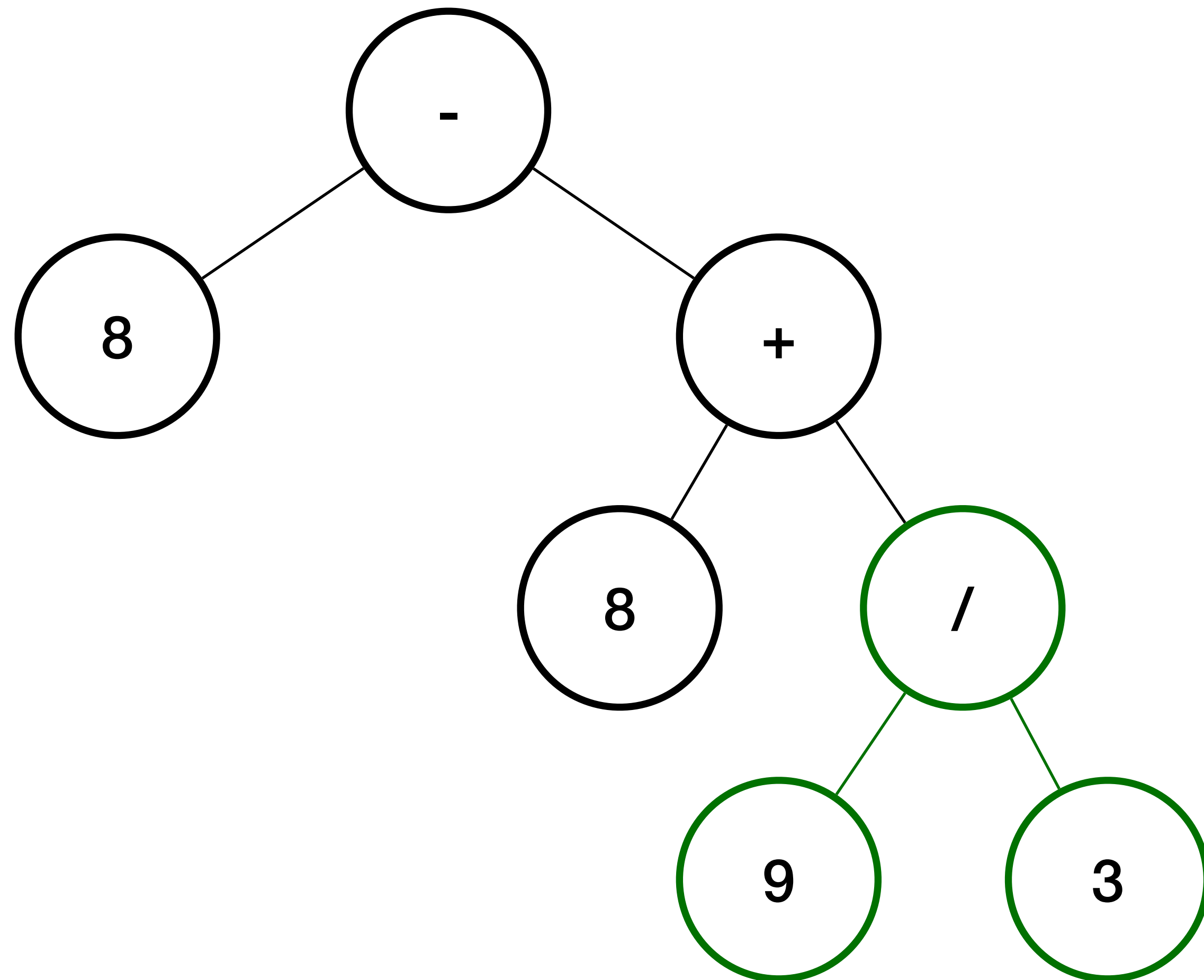
- $(12-4) - (8+9/3)$ as an expression tree



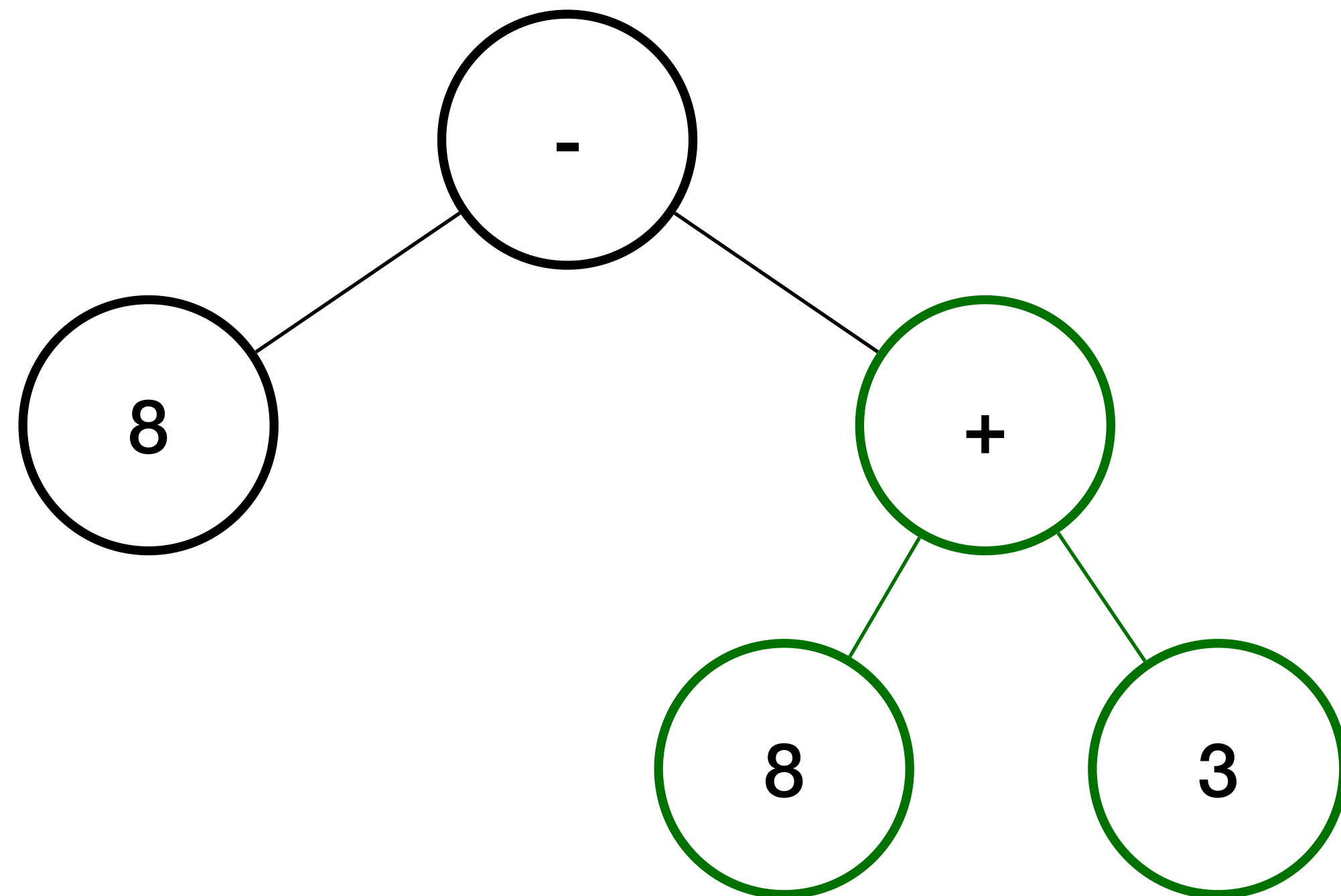
Evaluating Expression Tree



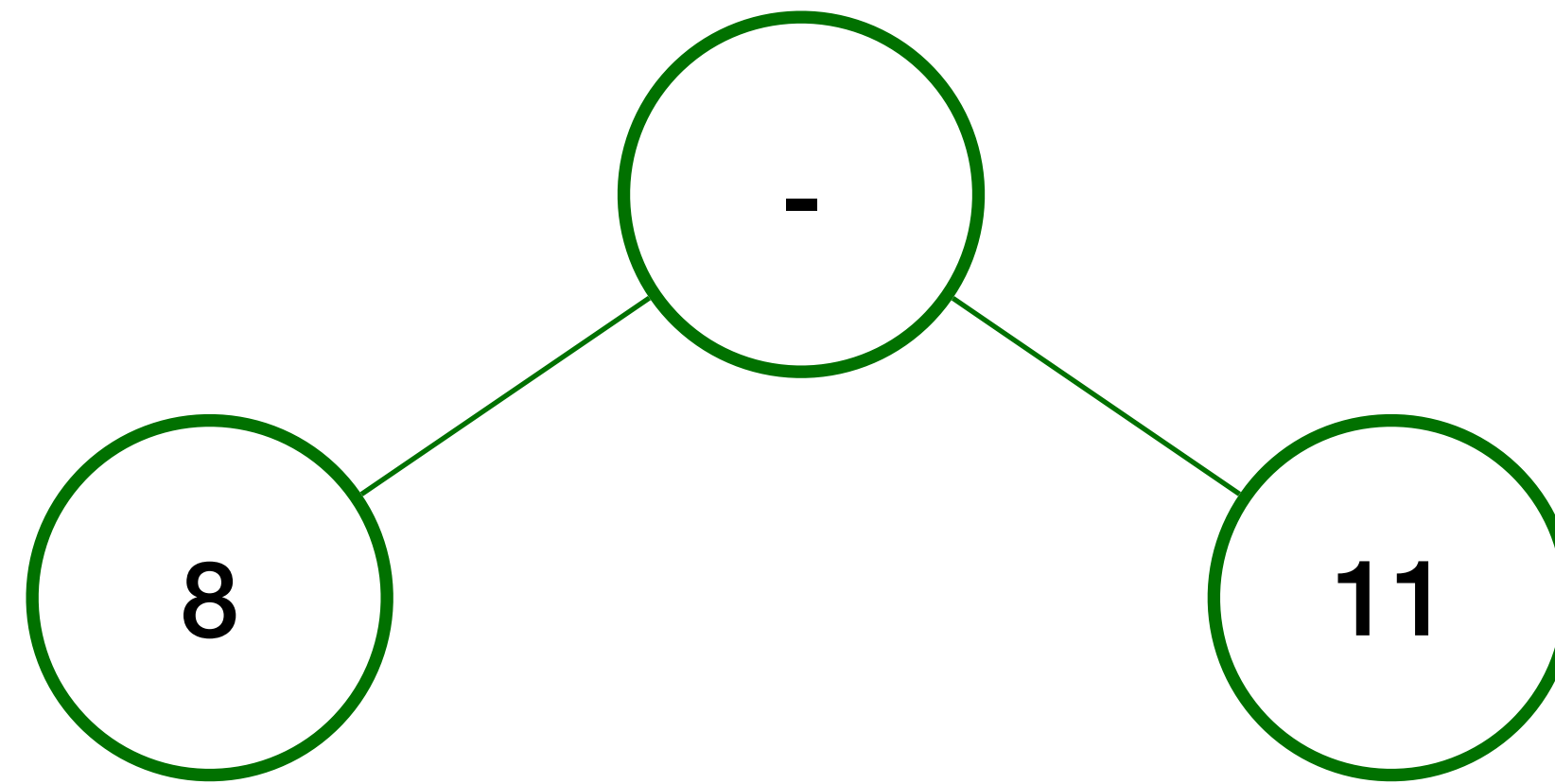
Evaluating Expression Tree



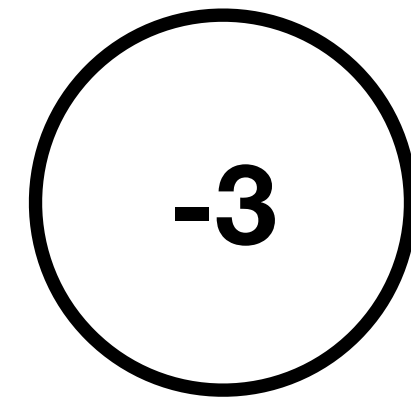
Evaluating Expression Tree



Evaluating Expression Tree



Evaluating Expression Tree



Expression Tree Traversals

- Modified in-order traversal that adds parentheses around each operator
- Generates a fully parenthesized infix expression
- $((12-4)-(8+(9/3)))$

```
def fullyParenthesizedInOrderTraversal[A](node: BinaryTreeNode[A], f: A => Unit): Unit = {  
  if (node != null) {  
    val operator = List("^", "*", "/", "+", "-").contains(node.value)  
    if (operator) {  
      print("(")  
    }  
    fullyParenthesizedInOrderTraversal(node.left, f)  
    f(node.value)  
    fullyParenthesizedInOrderTraversal(node.right, f)  
    if (operator) {  
      print(")")  
    }  
  }  
}
```

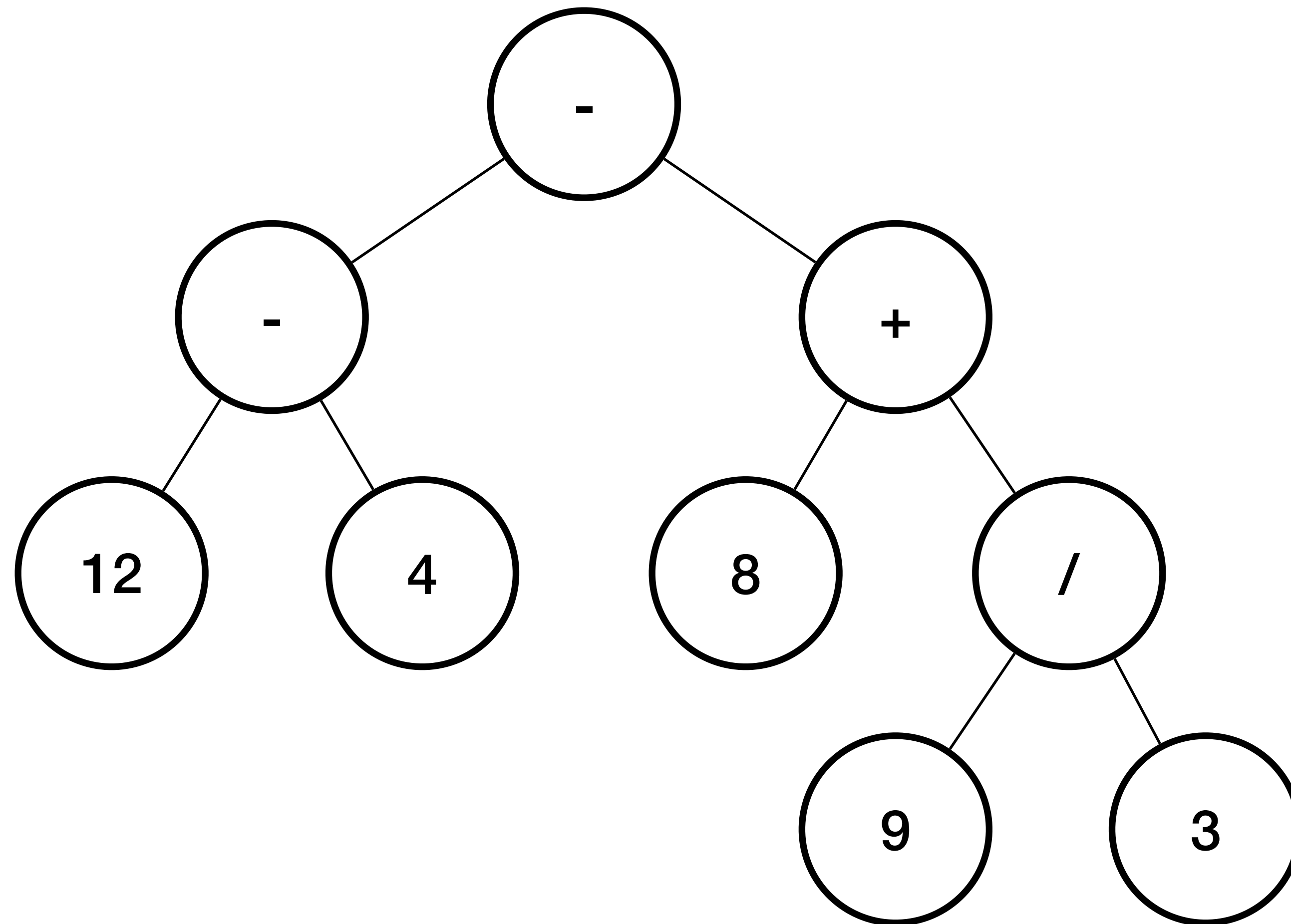
Expression Tree Traversals

- Unmodified post-order traversal generates a postfix express
- 12 4 - 8 9 3 / + -

```
postOrderTraversal(root, (token: String) => print(token + " "))
```

Expression Tree Traversals

- 12 4 - 8 9 3 / + -



Lecture Question

Task: Evaluate an expression tree

- In the `week8.trees.ExpressionTree` object, write a method named `evaluateTree` that takes the root of an expression tree (`BinaryTreeNode[String]`) as a parameter and returns the evaluation of the tree as a `Double`
- The operators can be `"*"`, `"/"`, `"+"`, and `"-"`
- You can assume that all Nodes storing numbers are well-formed (i.e. If a node's `String` is not one of the 4 operators, you can call `.toDouble` on it to convert it to a `Double`)

Testing: In a package named `"tests"` create a class named `"TestExpressionTree"` as a test suite that tests the functionality above (Only write tests with valid expression trees as input)