Binary Trees and Traversals
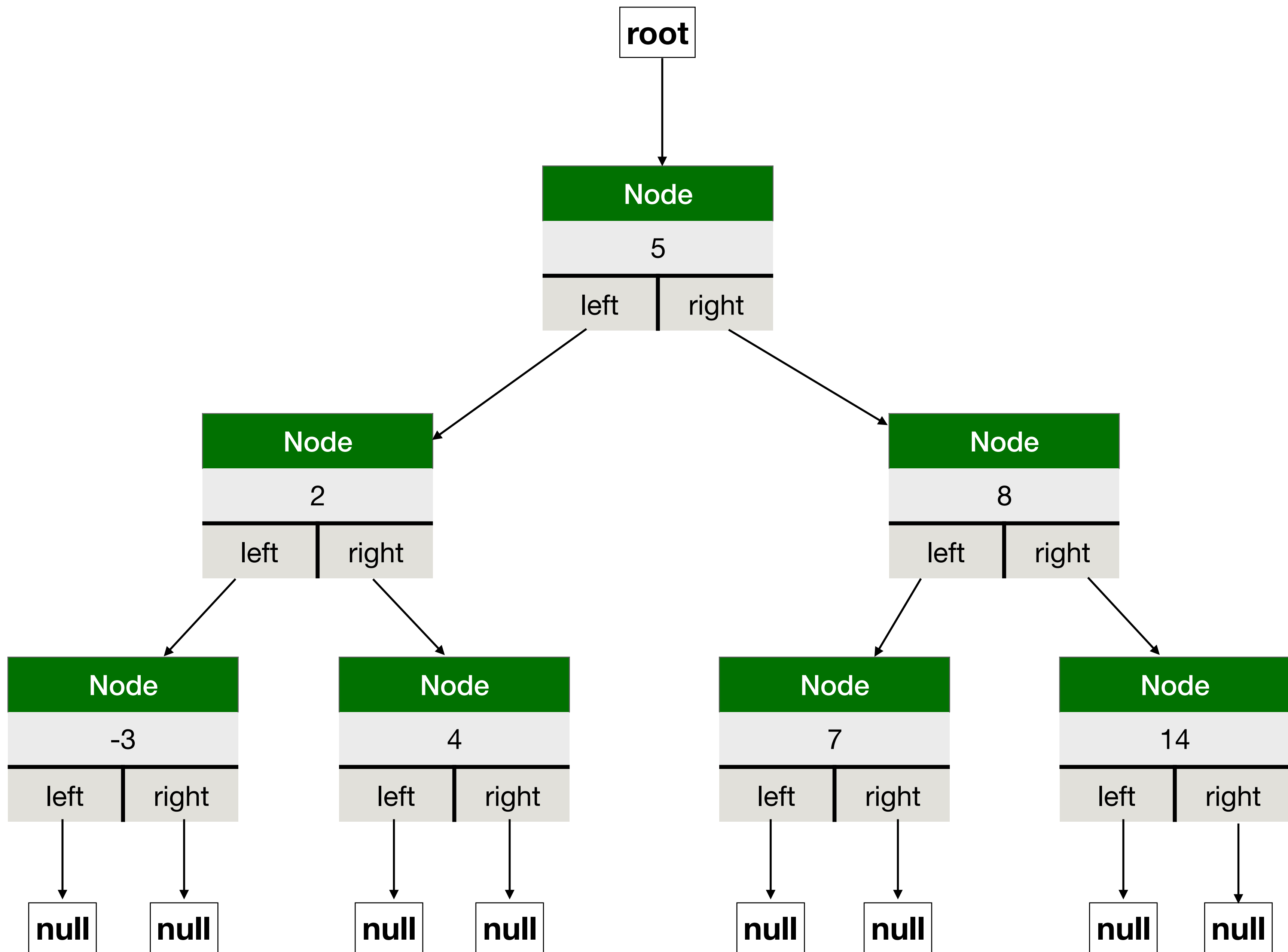
Lecture Question

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 -

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

```
                          ┌──────┐
                          │ root │
                          └──────┘
                              │
                              ▼
                    ┌──────────────────┐
                    │       Node       │
                    ├──────────────────┤
                    │        5         │
                    ├─────────┬────────┤
                    │  left   │  right │
                    └─────────┴────────┘
                   ╱                    ╲
                  ▼                      ▼
        ┌──────────────────┐    ┌──────────────────┐
        │       Node       │    │       Node       │
        ├──────────────────┤    ├──────────────────┤
        │        2         │    │        8         │
        ├─────────┬────────┤    ├─────────┬────────┤
        │  left   │ right  │    │  left   │ right  │
        └─────────┴────────┘    └─────────┴────────┘
       ╱              ╲          ╱              ╲
      ▼                ▼        ▼                ▼
┌────────────┐  ┌────────────┐  ┌────────────┐  ┌────────────┐
│    Node    │  │    Node    │  │    Node    │  │    Node    │
├────────────┤  ├────────────┤  ├────────────┤  ├────────────┤
│     -3     │  │     4      │  │     7      │  │     14     │
├──────┬─────┤  ├──────┬─────┤  ├──────┬─────┤  ├──────┬─────┤
│ left │right│  │ left │right│  │ left │right│  │ left │right│
└──────┴─────┘  └──────┴─────┘  └──────┴─────┘  └──────┴─────┘
   │      │        │      │        │      │        │      │
   ▼      ▼        ▼      ▼        ▼      ▼        ▼      ▼
 ┌────┐ ┌────┐  ┌────┐ ┌────┐  ┌────┐ ┌────┐  ┌────┐ ┌────┐
 │null│ │null│  │null│ │null│  │null│ │null│  │null│ │null│
 └────┘ └────┘  └────┘ └────┘  └────┘ └────┘  └────┘ └────┘
```

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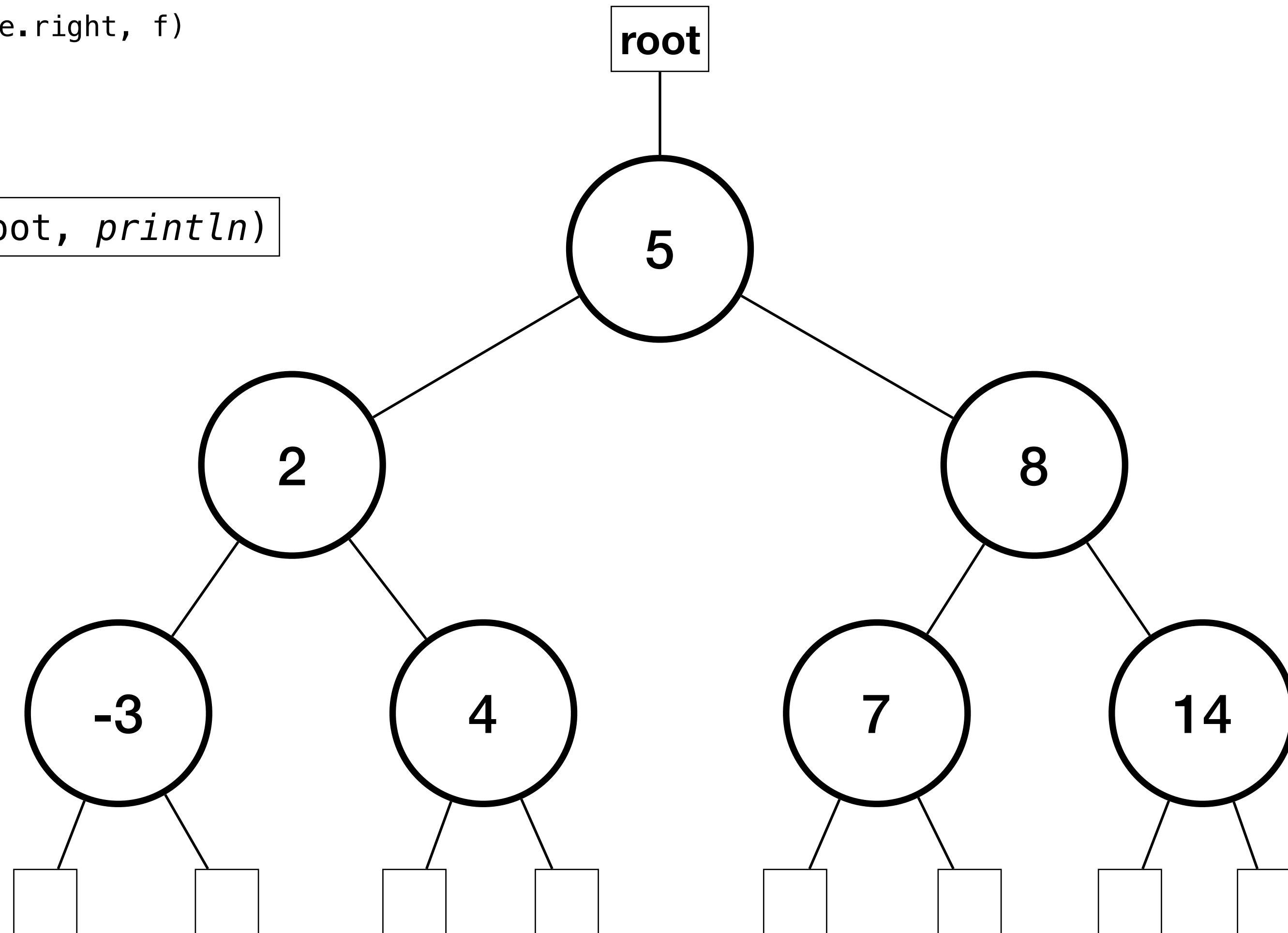
```
preOrderTraversal(root, println)
```

Tree Traversals

`preOrderTraversal(root, println)`

Printed:
5
2
-3
4
8
7
14
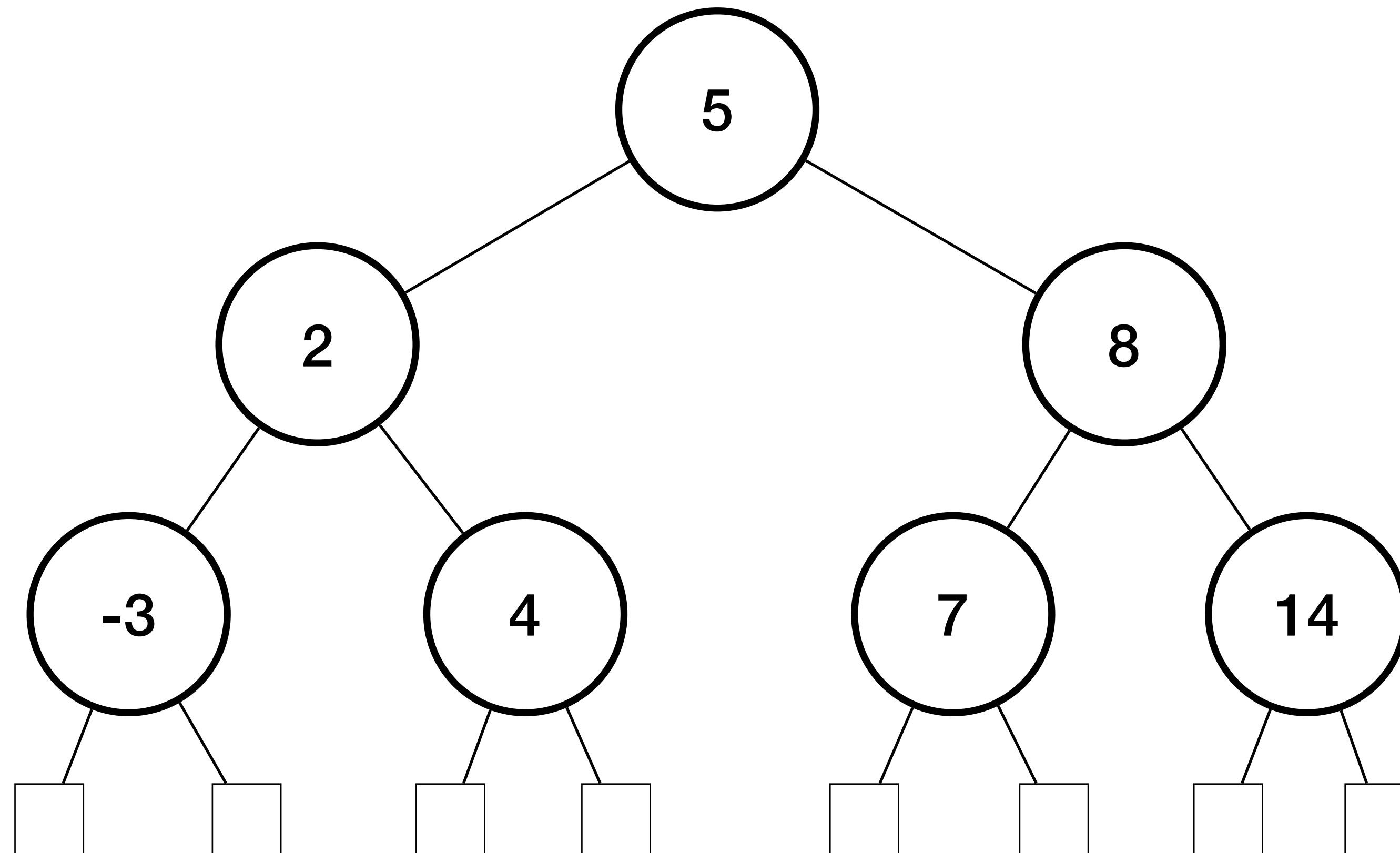
Tree Traversals

- Post-Order Traversal

  - Call post-order on the left child

  - Call post-order on the right child

  - Visit the node's value

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

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

Tree Traversals

postOrderTraversal(root, *println*)

Printed:
-3
4
2
7
14
8
5
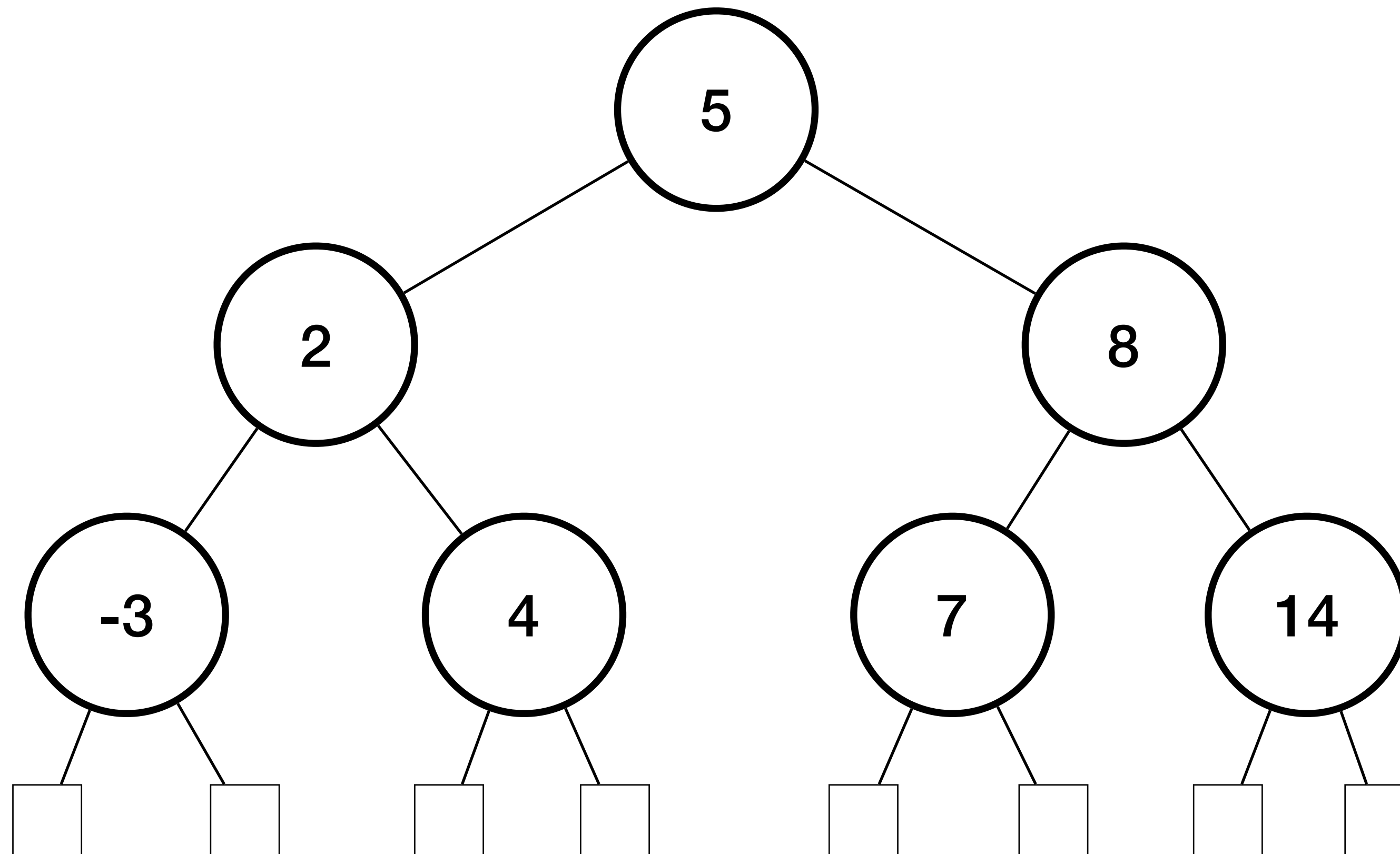
Tree Traversals

- In-Order Traversal

    - Call in-order on the left child

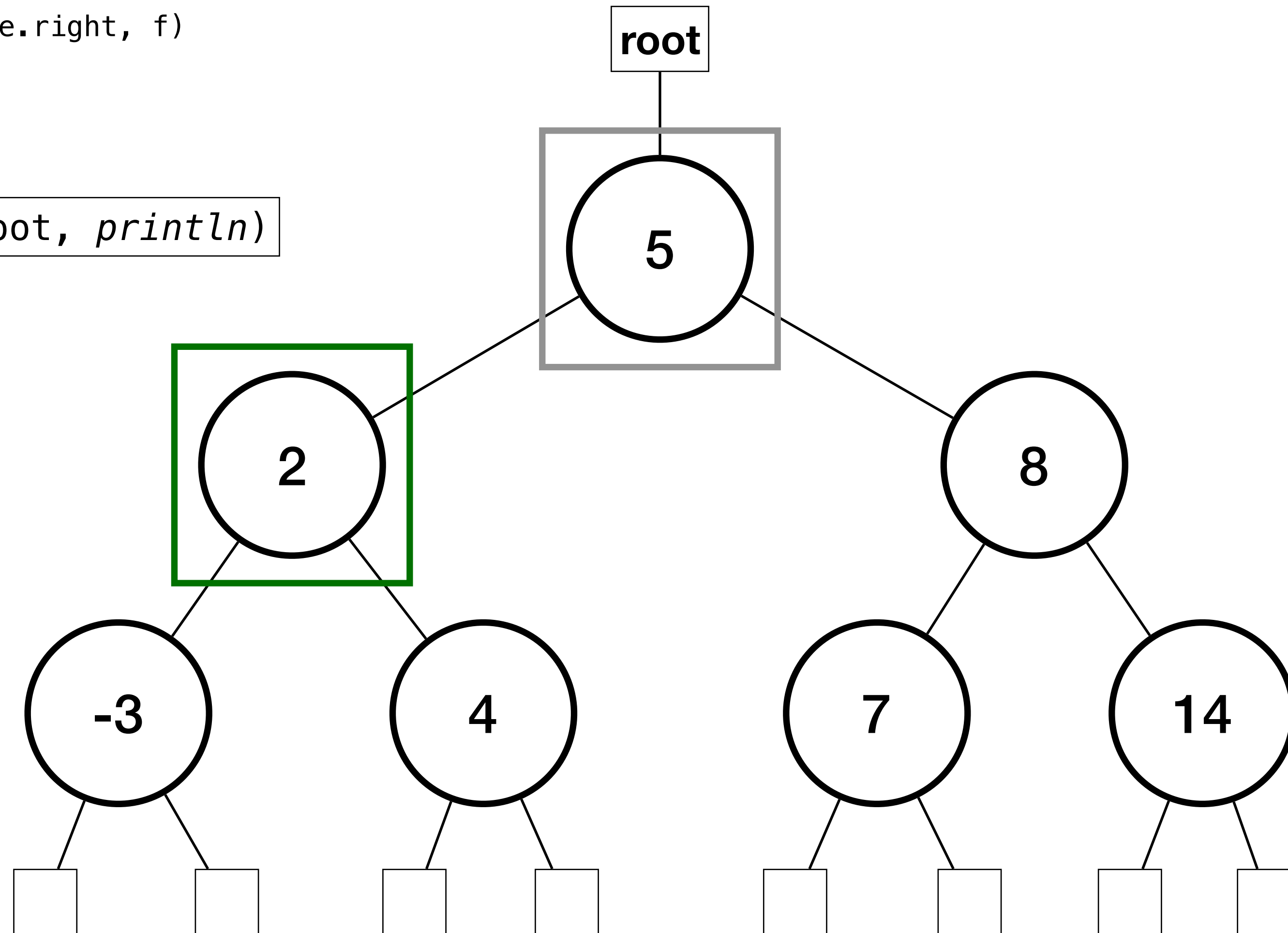   - Visit the node's value

    - Call in-order on the right child

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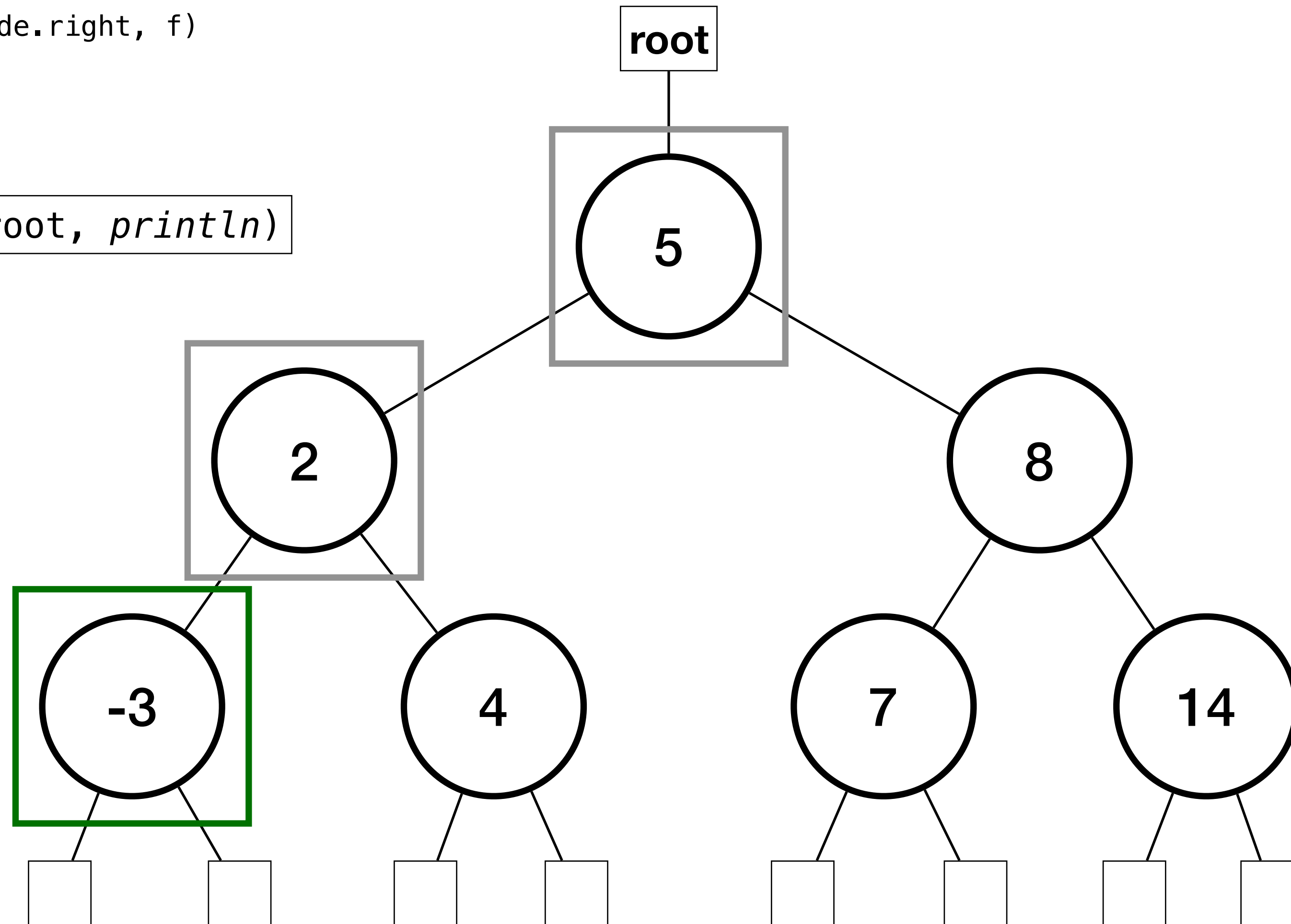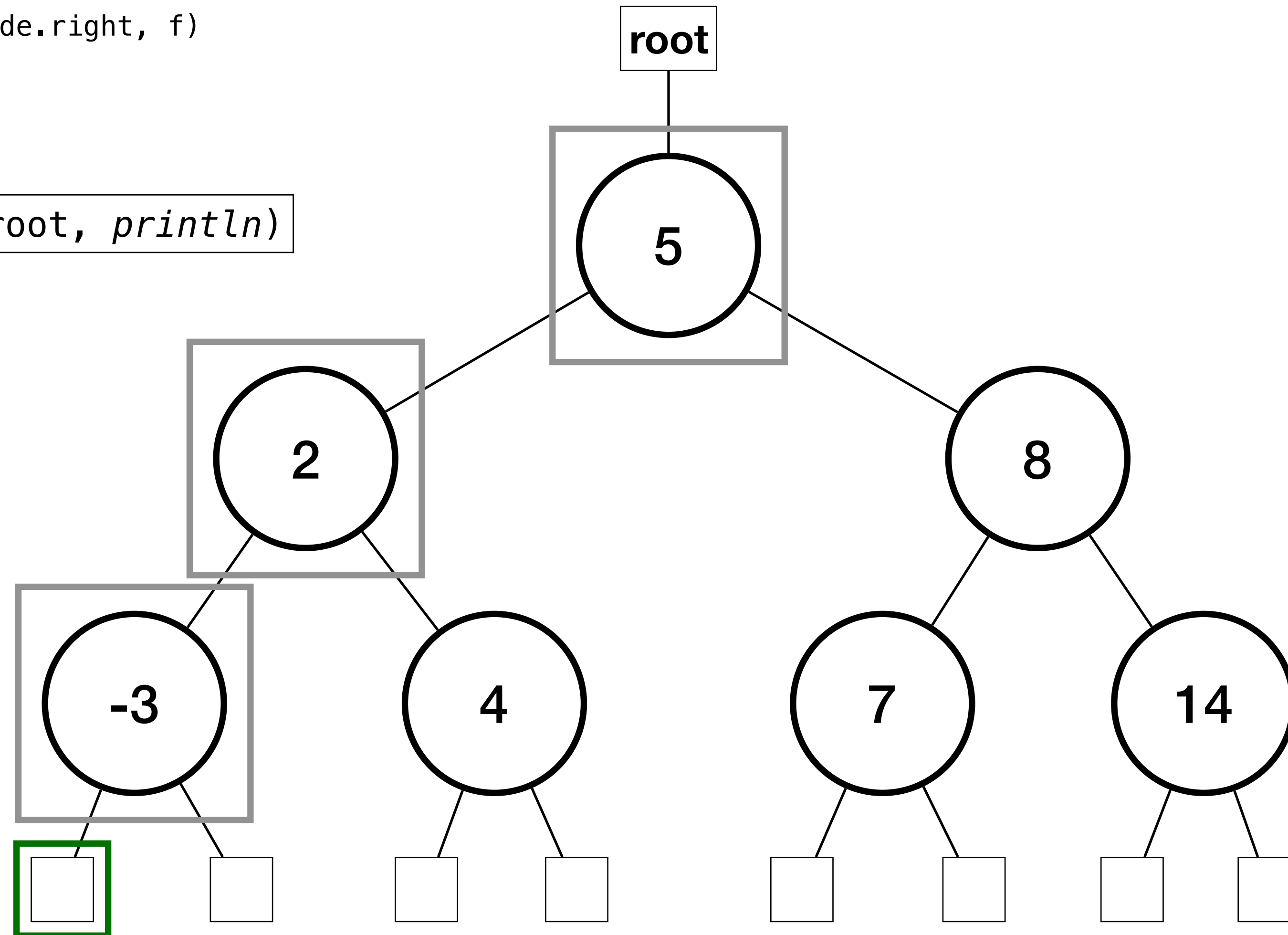
```scala
inOrderTraversal(root, println)
```

The Code

```scala
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)
  }
}
```

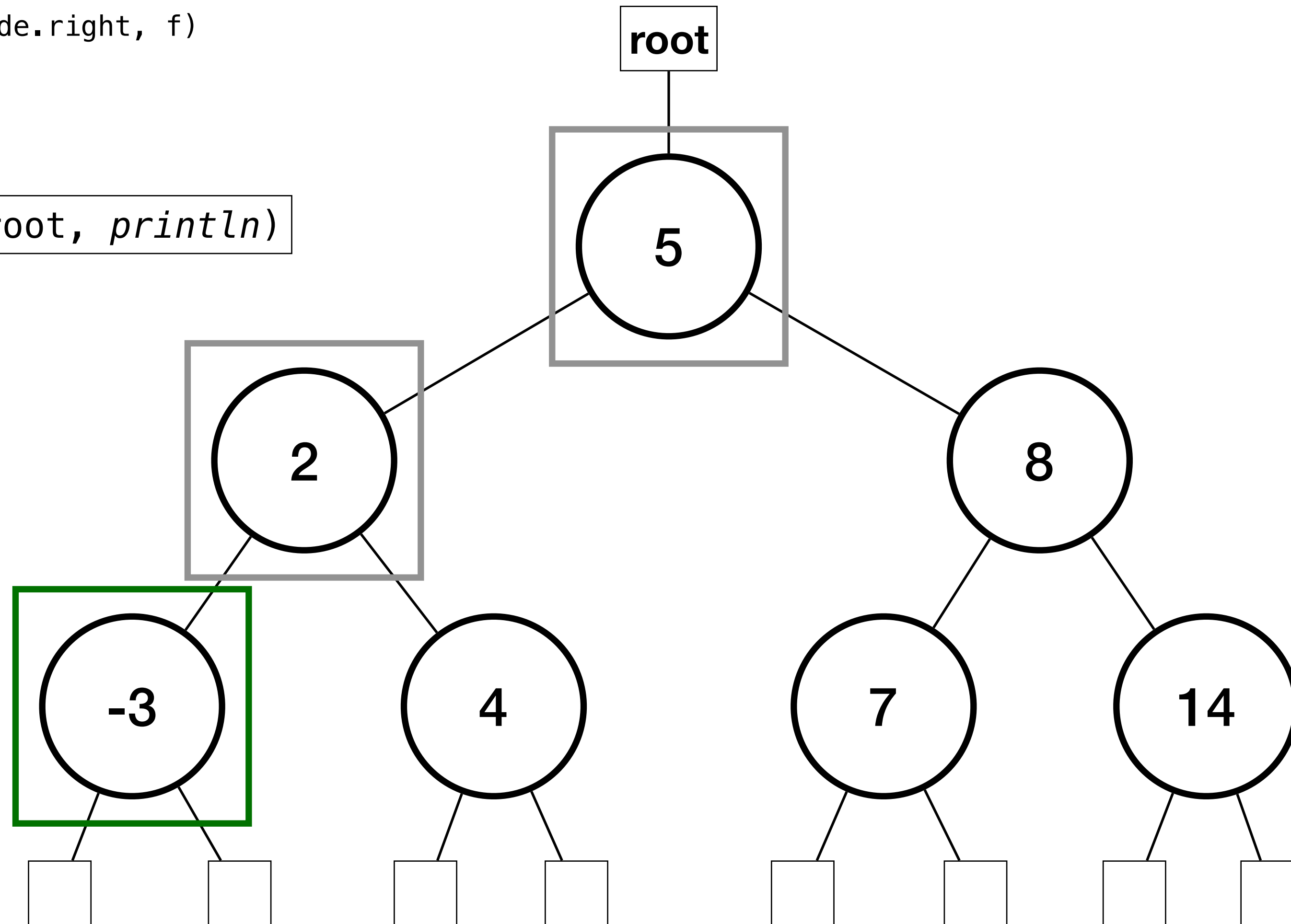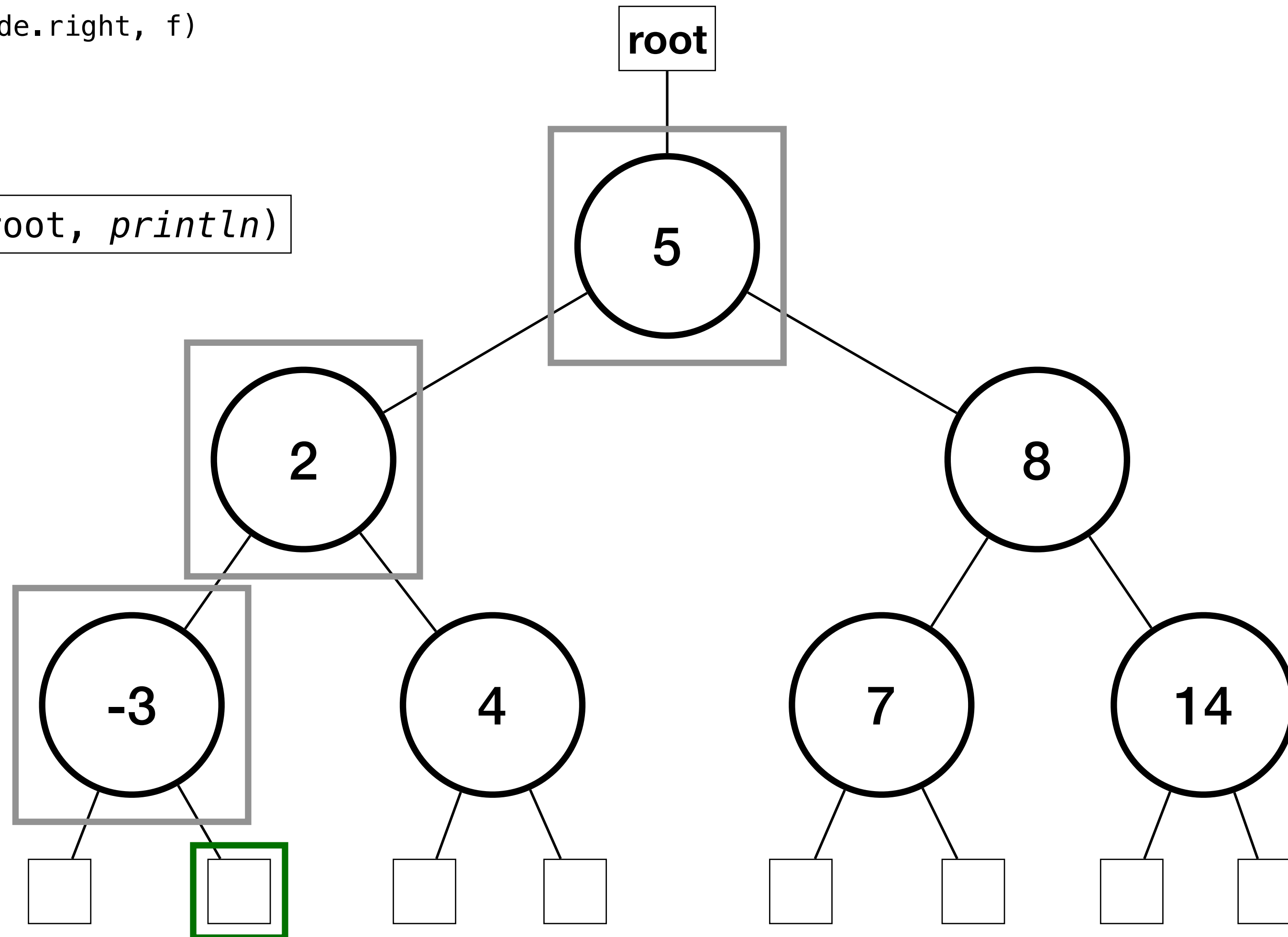- **Challenge: Write these with loops and no recursion**

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:

root

5

2

8

-3

4

7

14

Traversals

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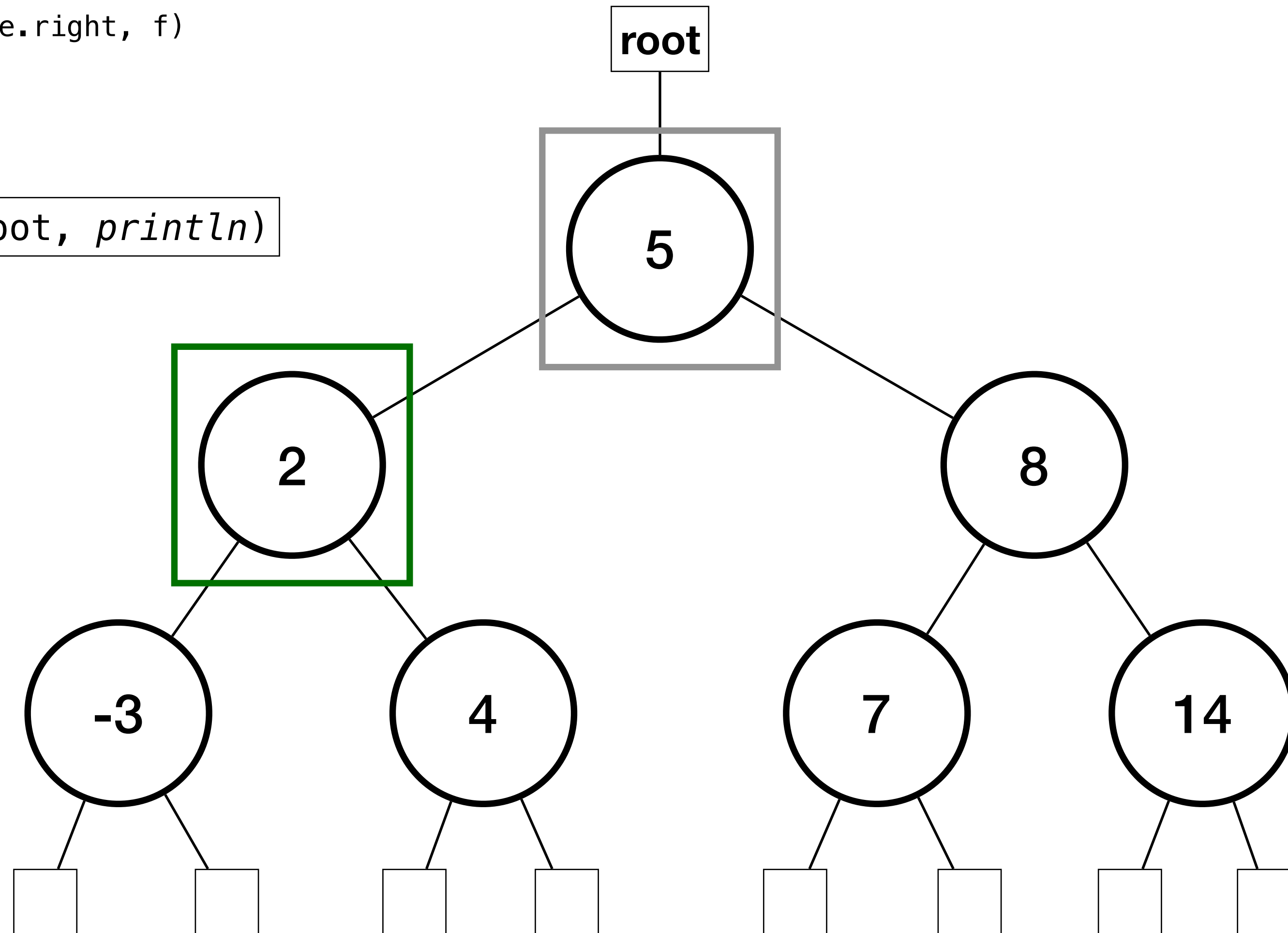
inOrderTraversal(root, println)

Printed:

Traversals

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

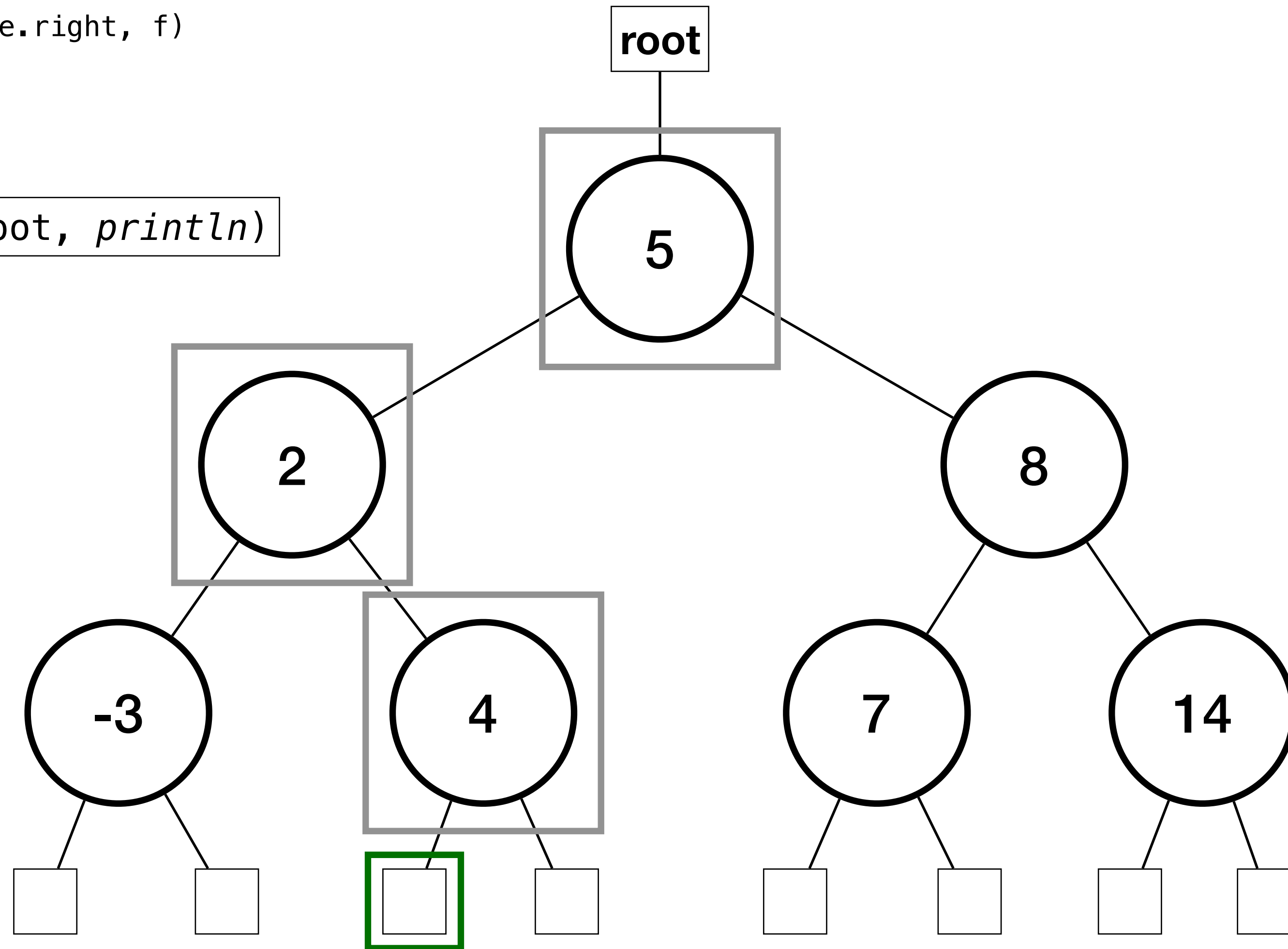`inOrderTraversal(root, println)`

Printed:

Traversals

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

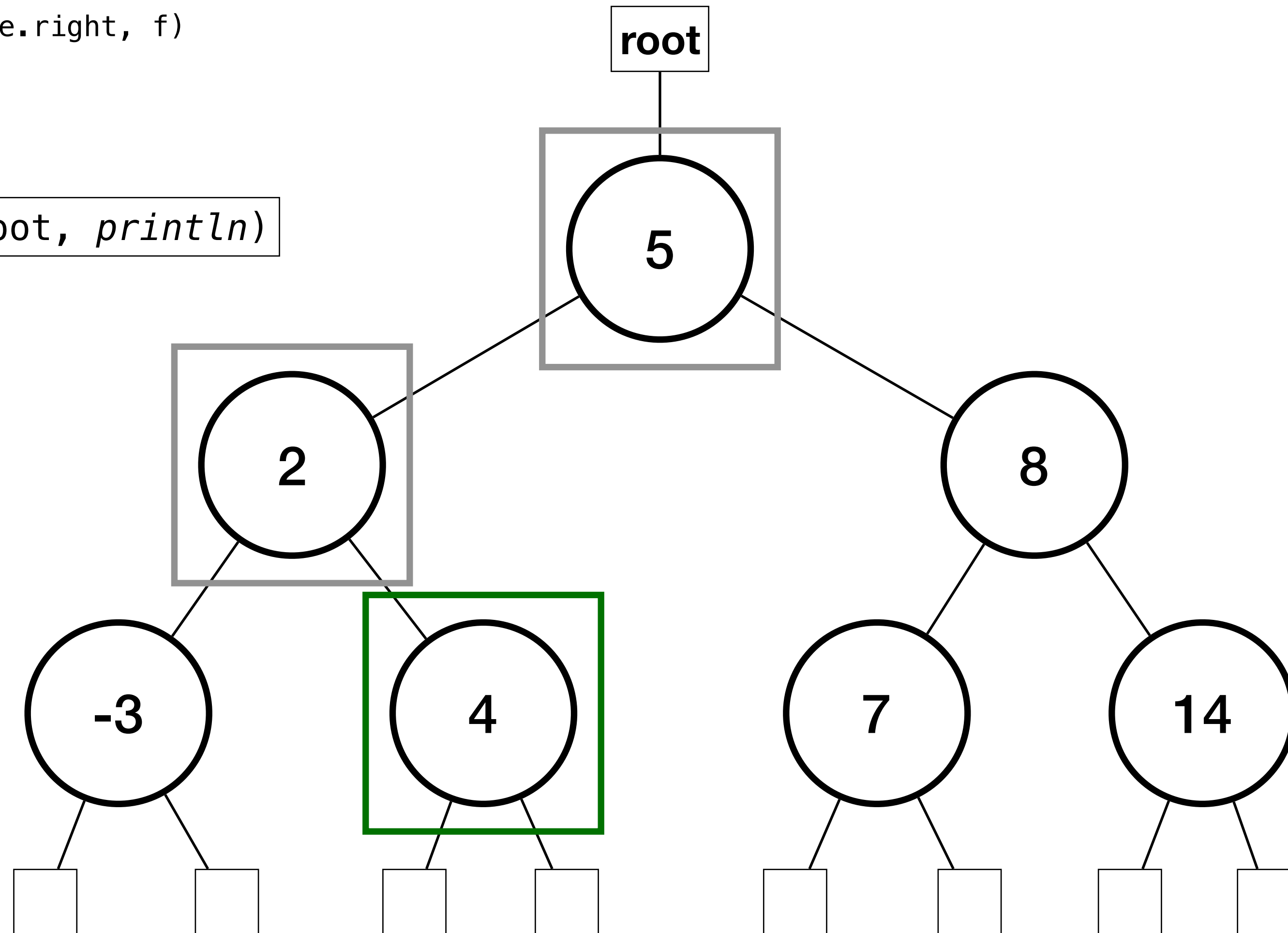`inOrderTraversal(root, println)`

Printed:

Traversals

```scala
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

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

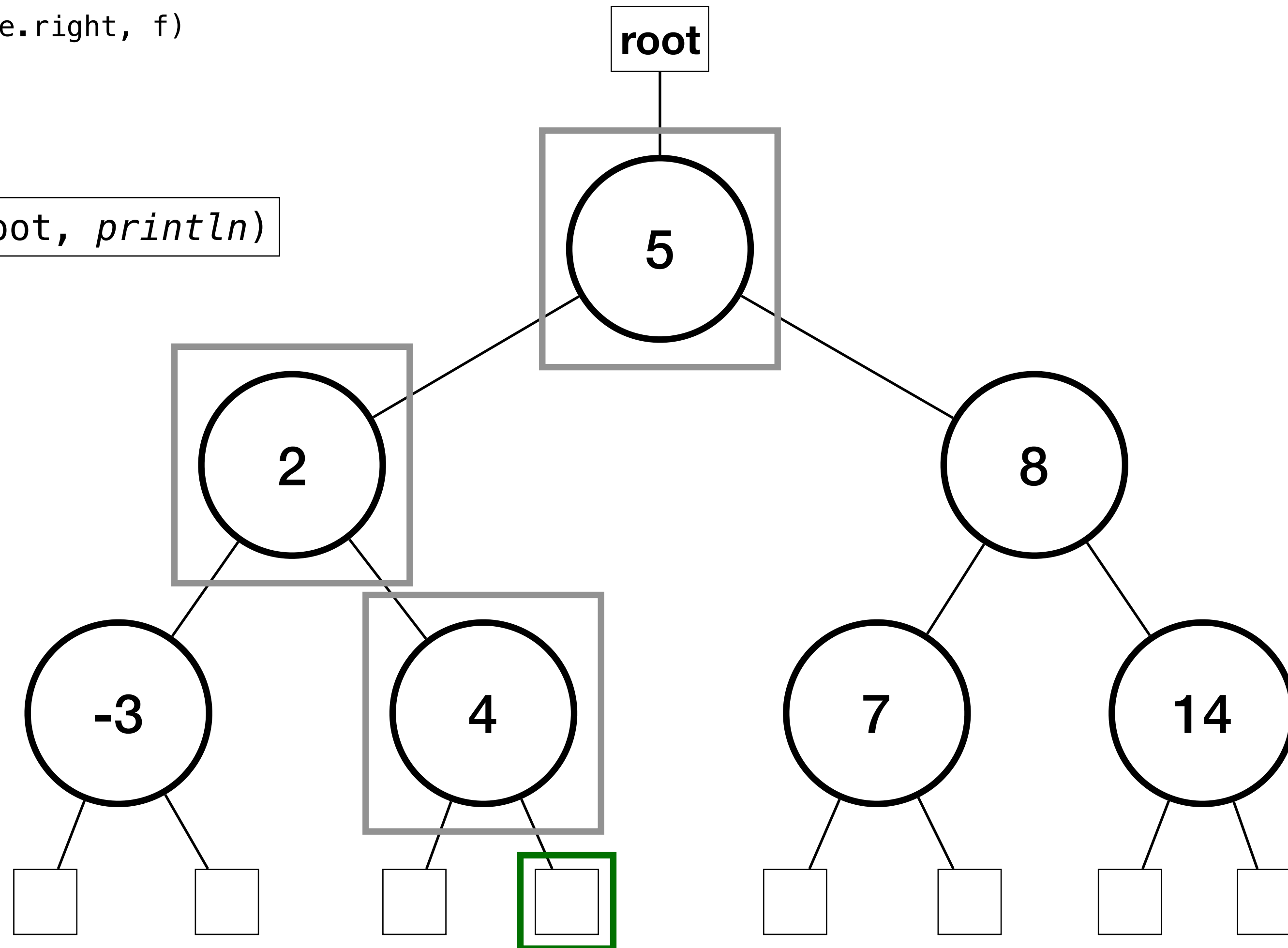`inOrderTraversal(root, println)`

Printed:
-3

Traversals

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

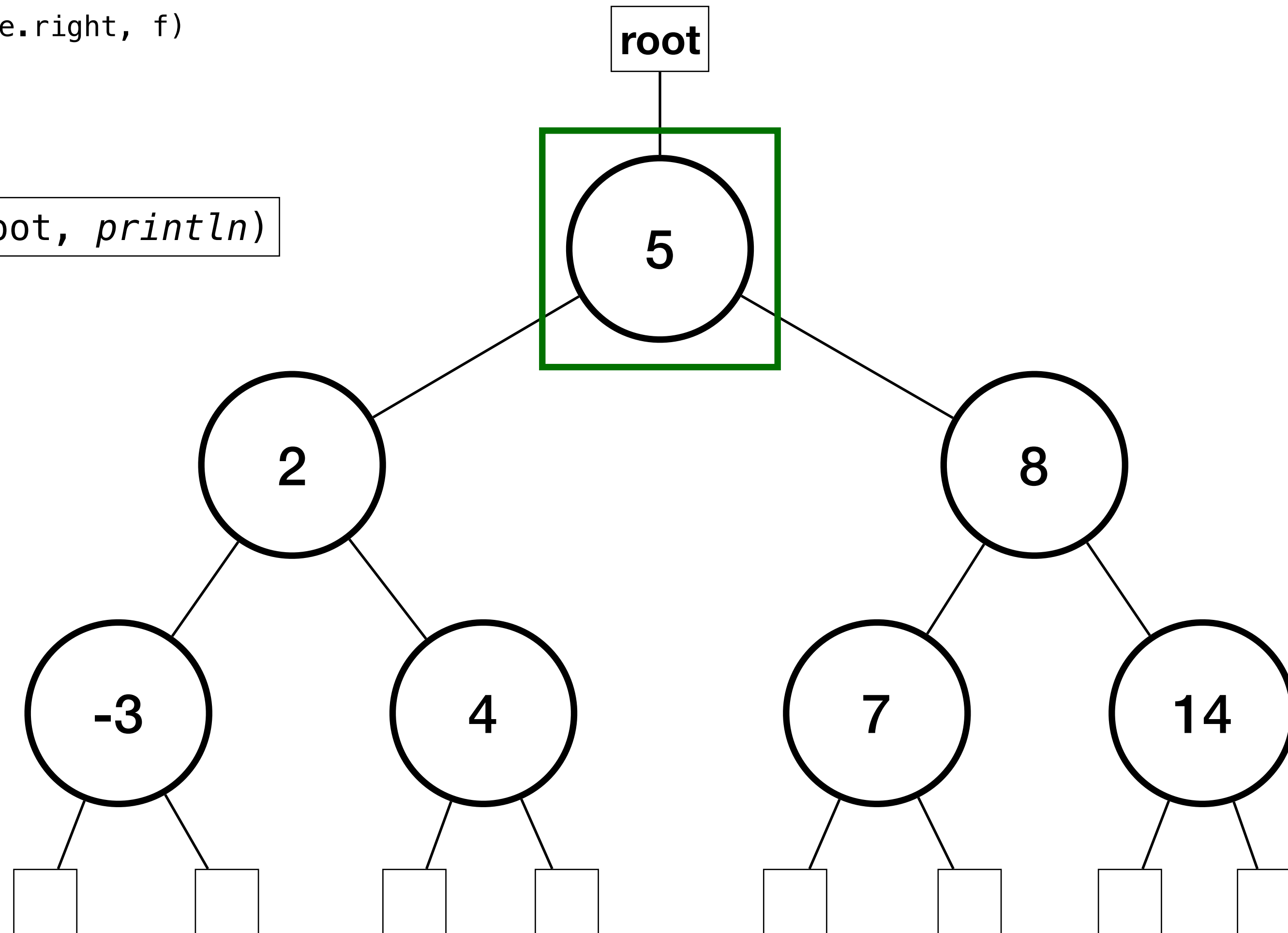inOrderTraversal(root, println)

Printed:
-3

Traversals

```scala
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

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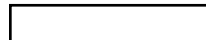
`inOrderTraversal(root, println)`

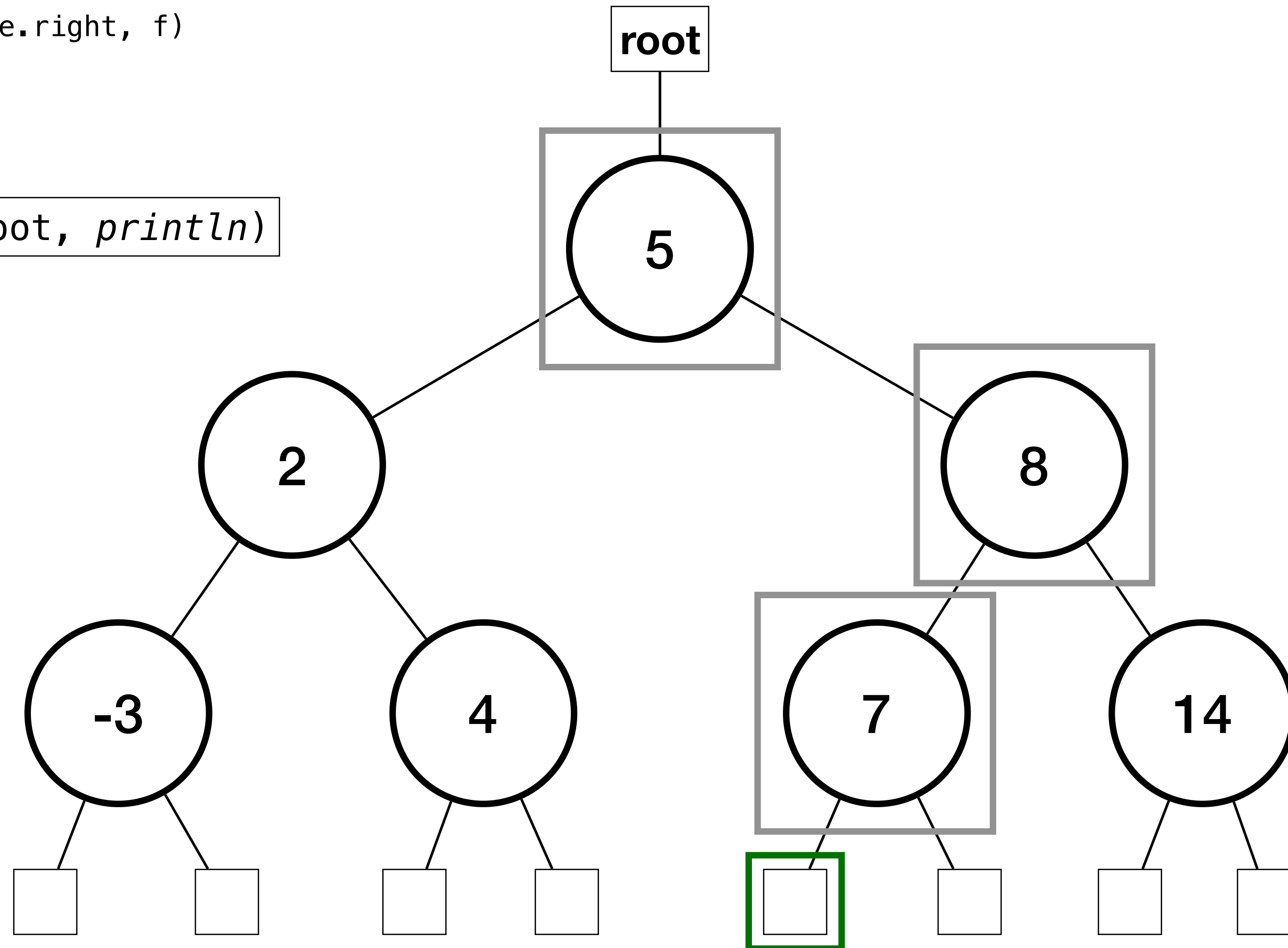**Printed:**
-3
2

Traversals

```scala
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:
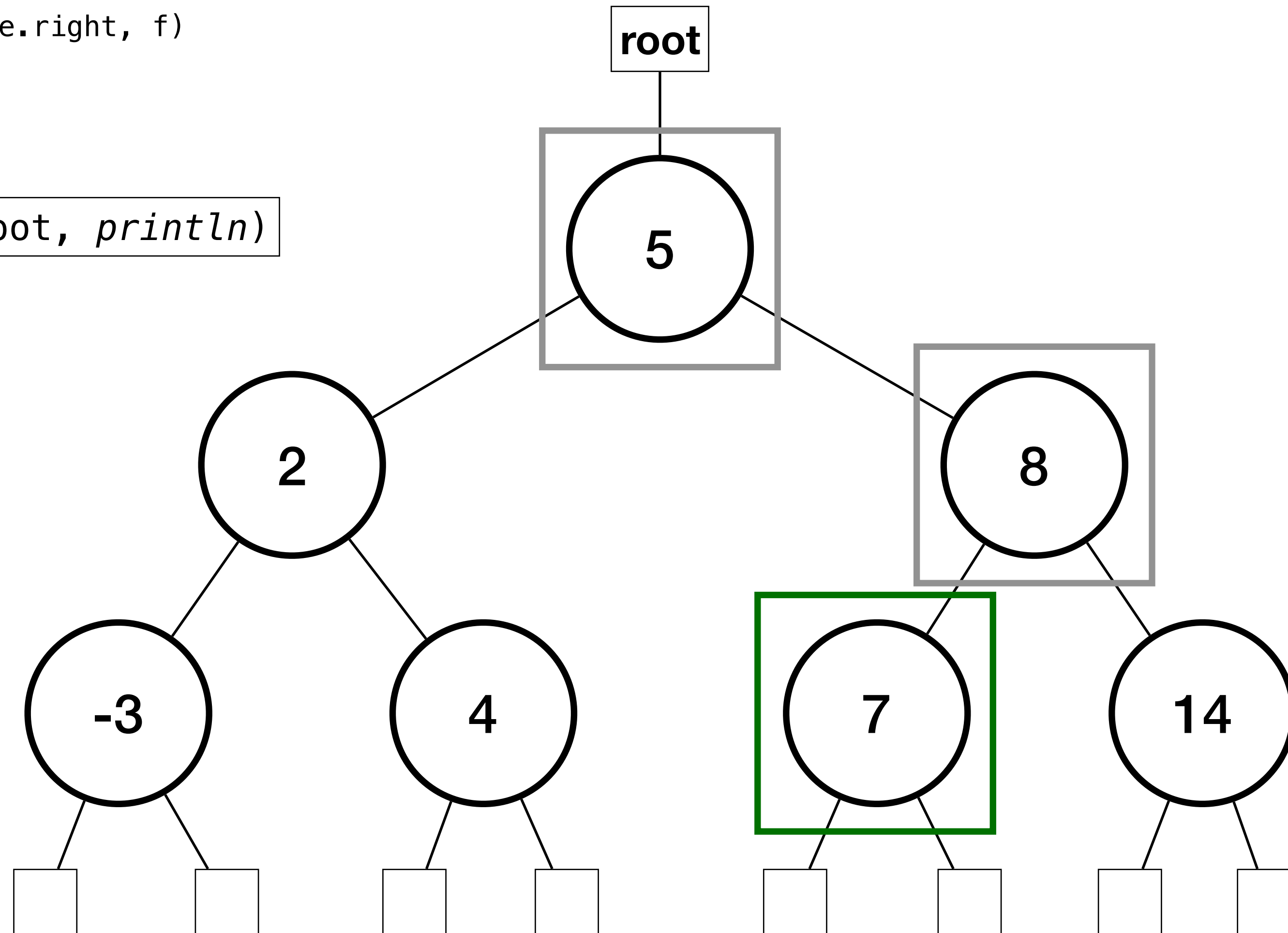**-3**
2
4

Traversals

```scala
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)`
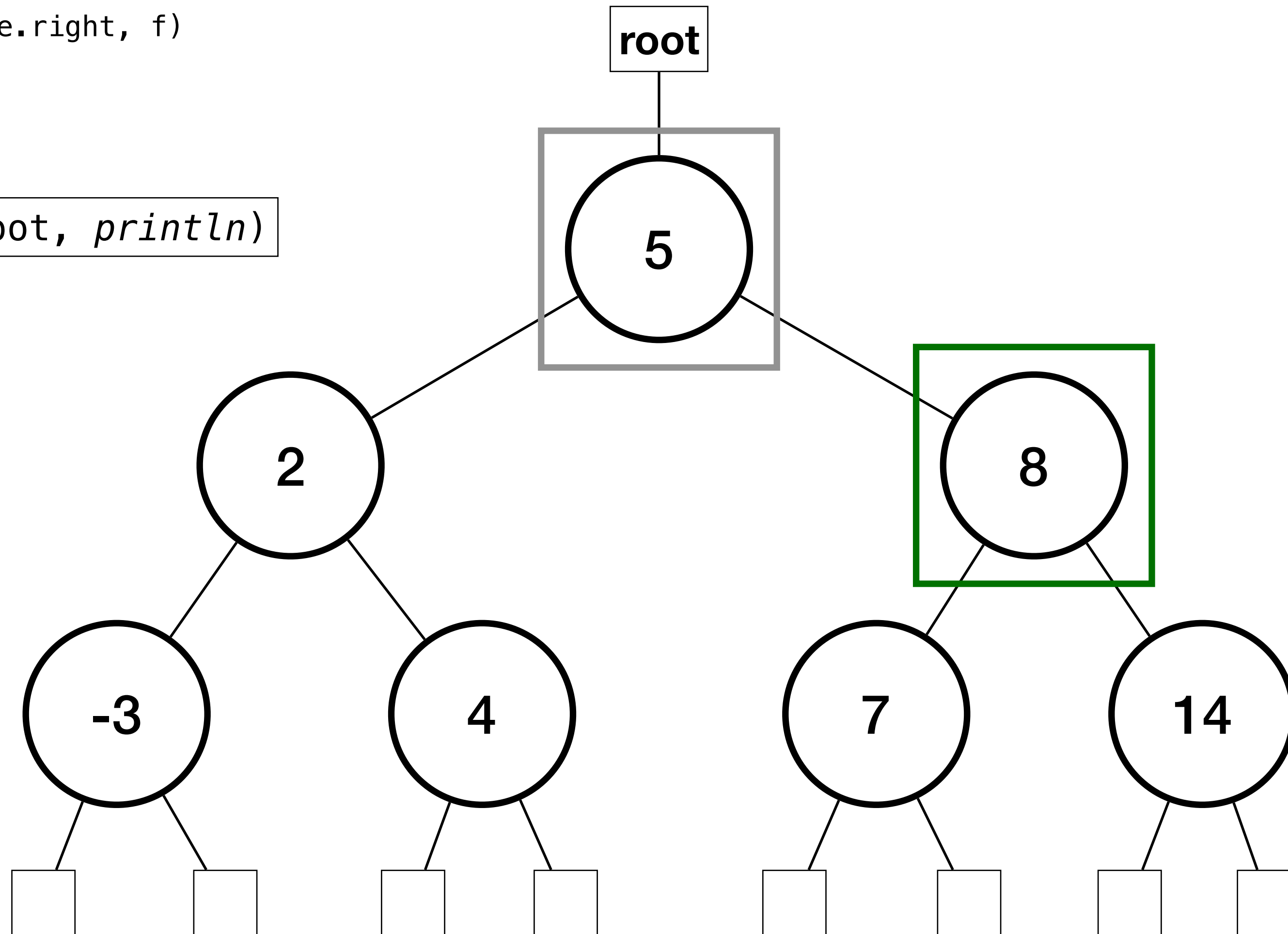
Printed:
-3
2
4

Traversals

```scala
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

root

5
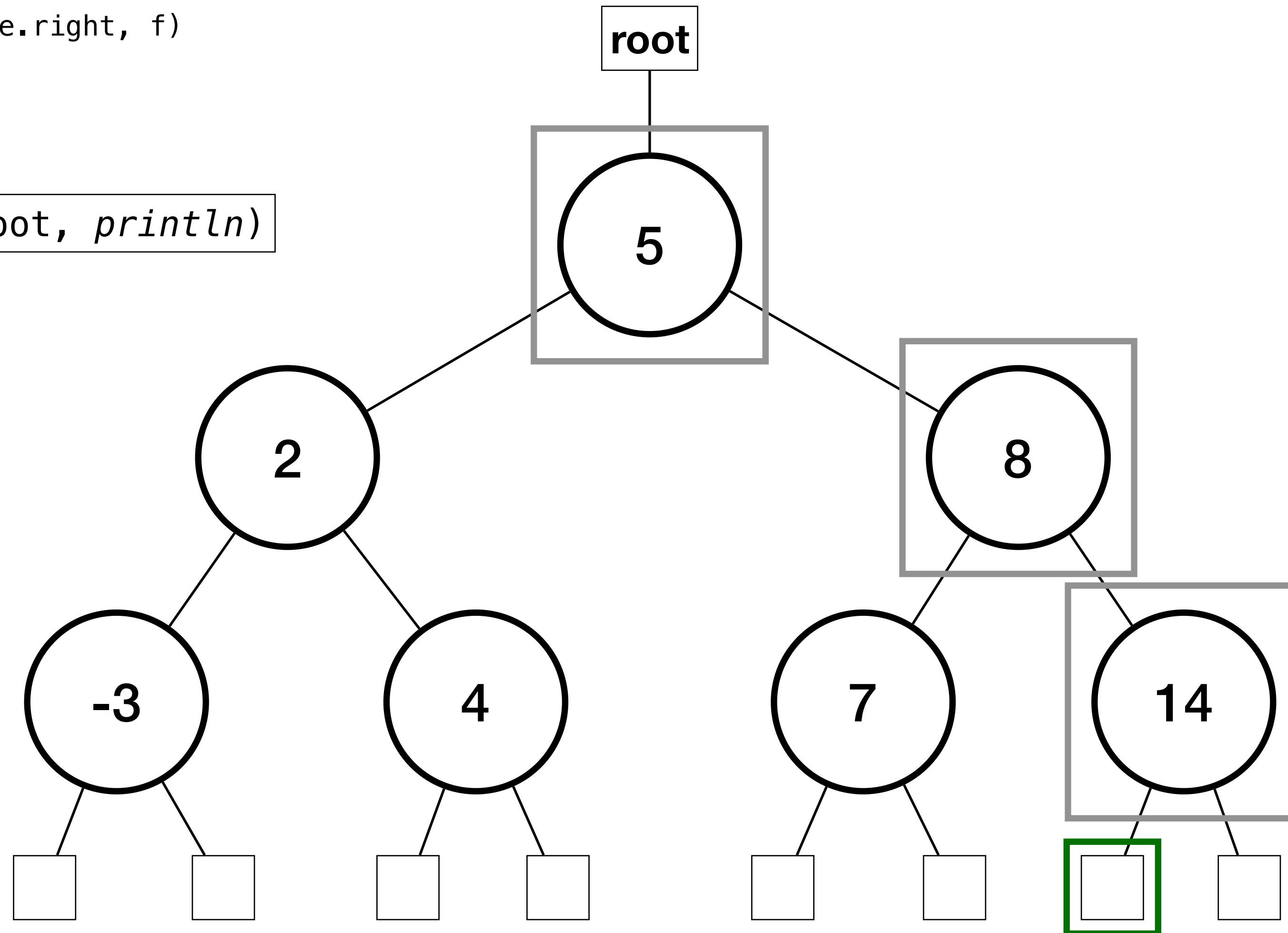
2   8

-3   4   7   14

Traversals

```scala
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

```scala
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
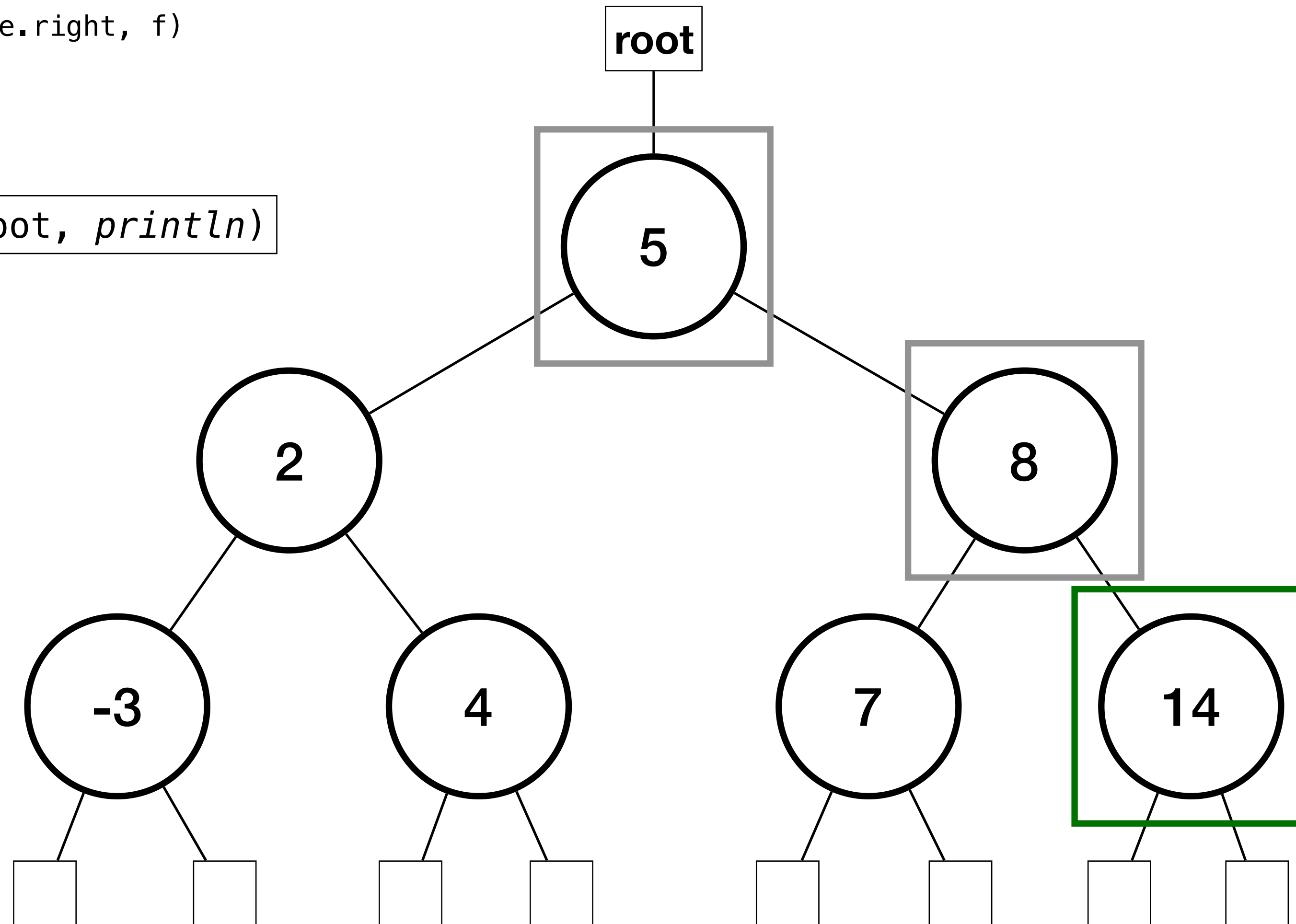
root

5

2

8

-3

4

7

14

Traversals

```scala
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:
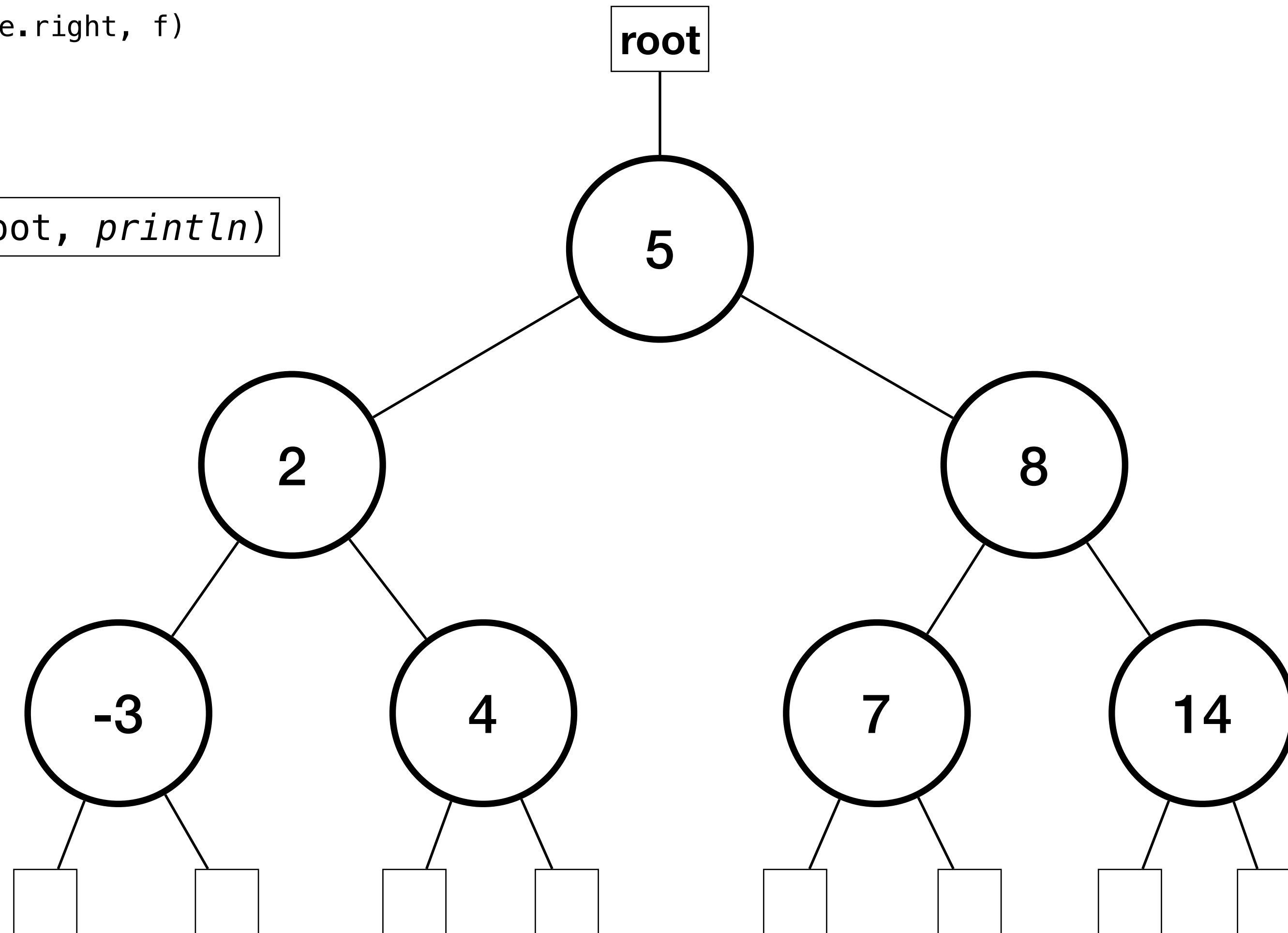
-3

2

4

5

7

8

Traversals

```scala
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

```scala
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

root

5

2

8

-3

4

7

14

Traversals

```scala
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
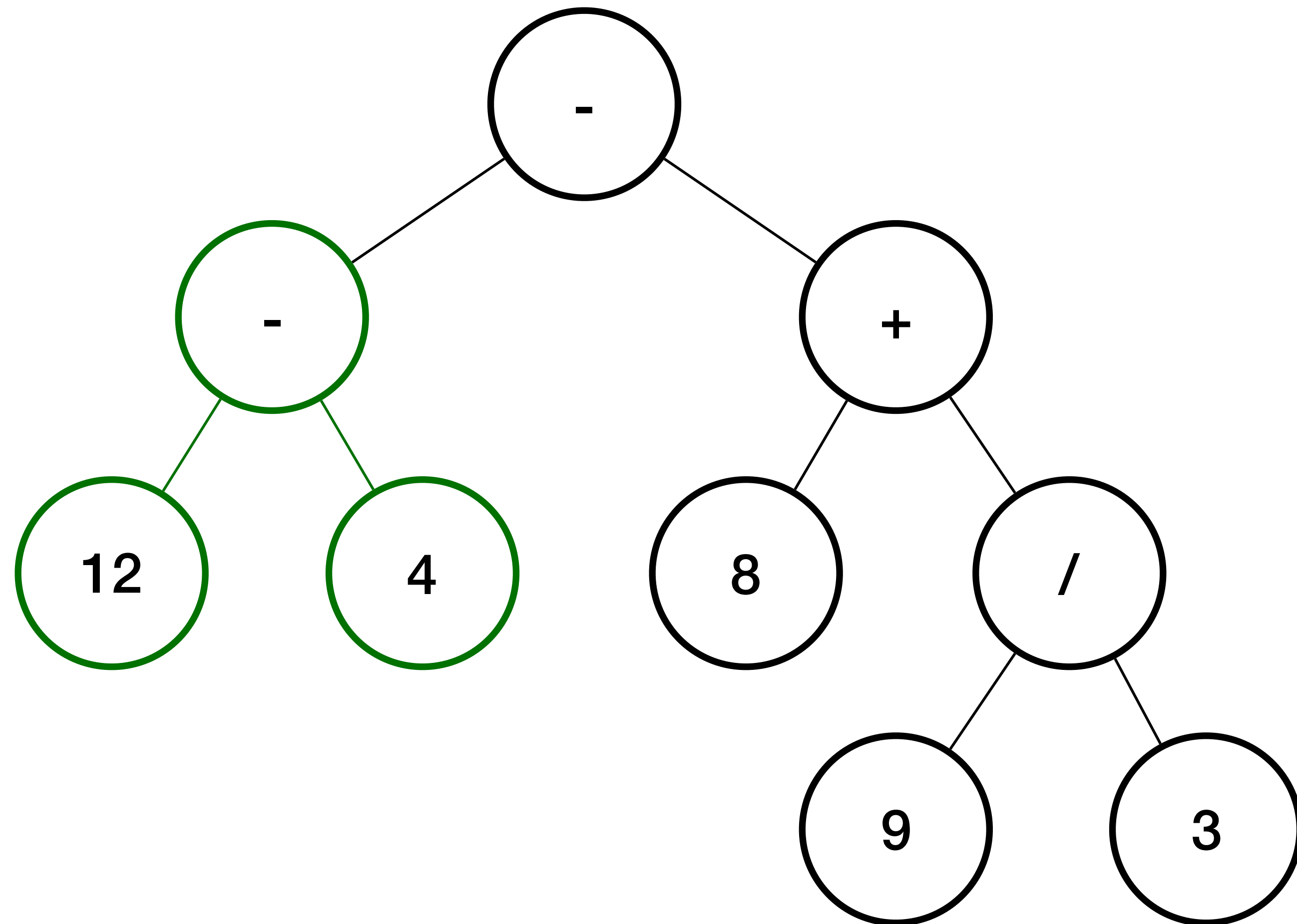**4**
5
7
8
14

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

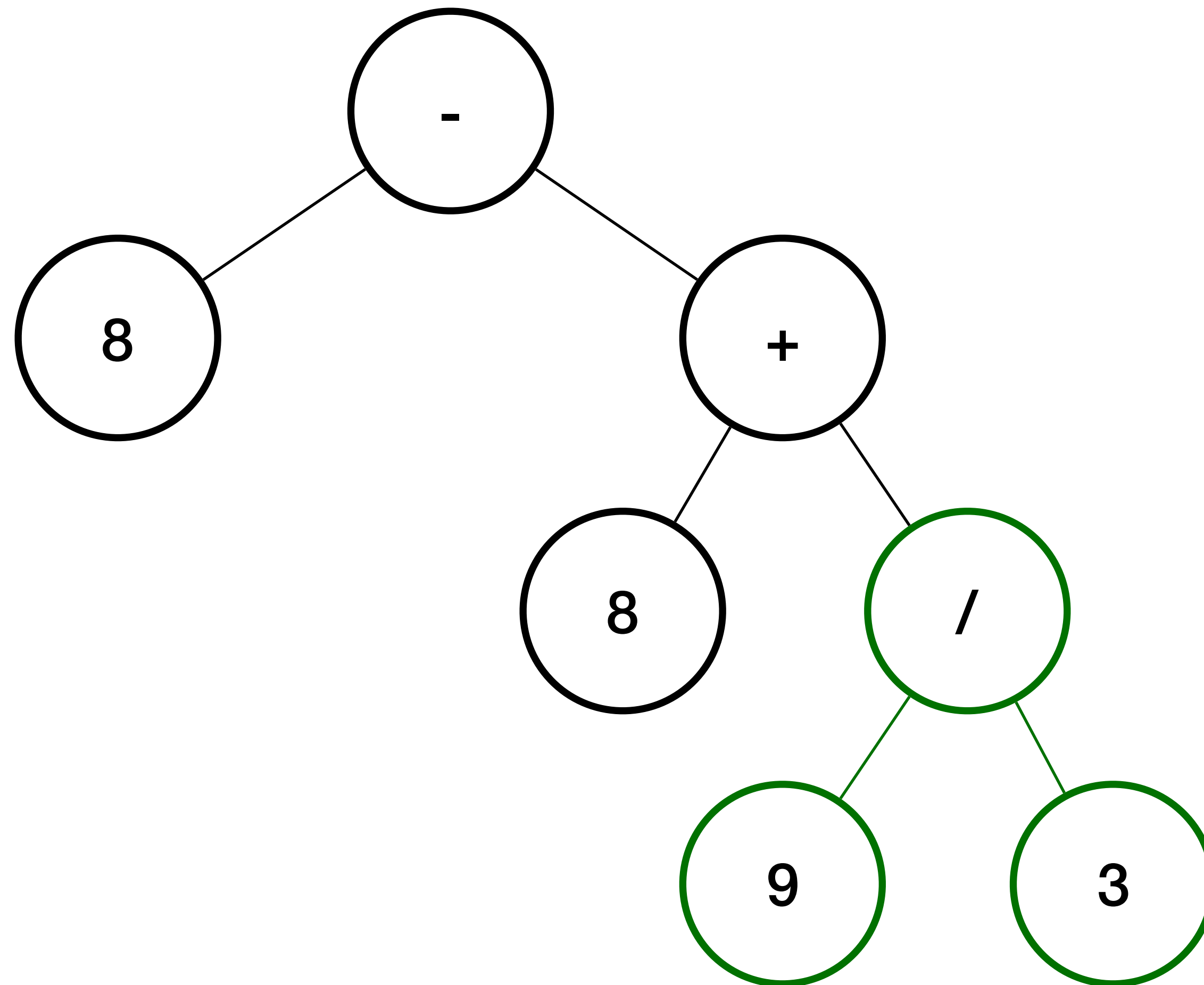 - Operands can be operators

Expression Tree
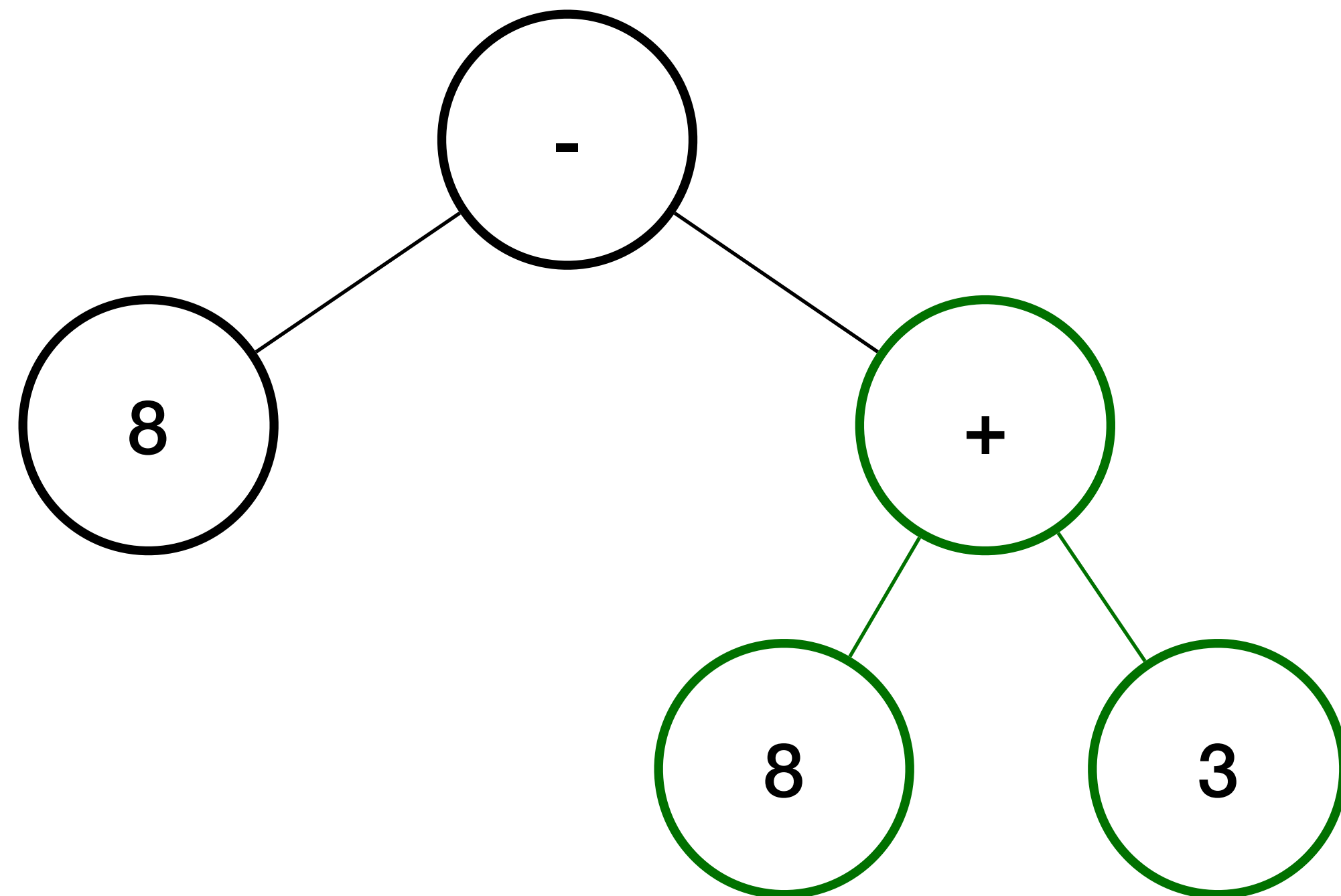
- (12-4) - (8+9/3) as an expression tree
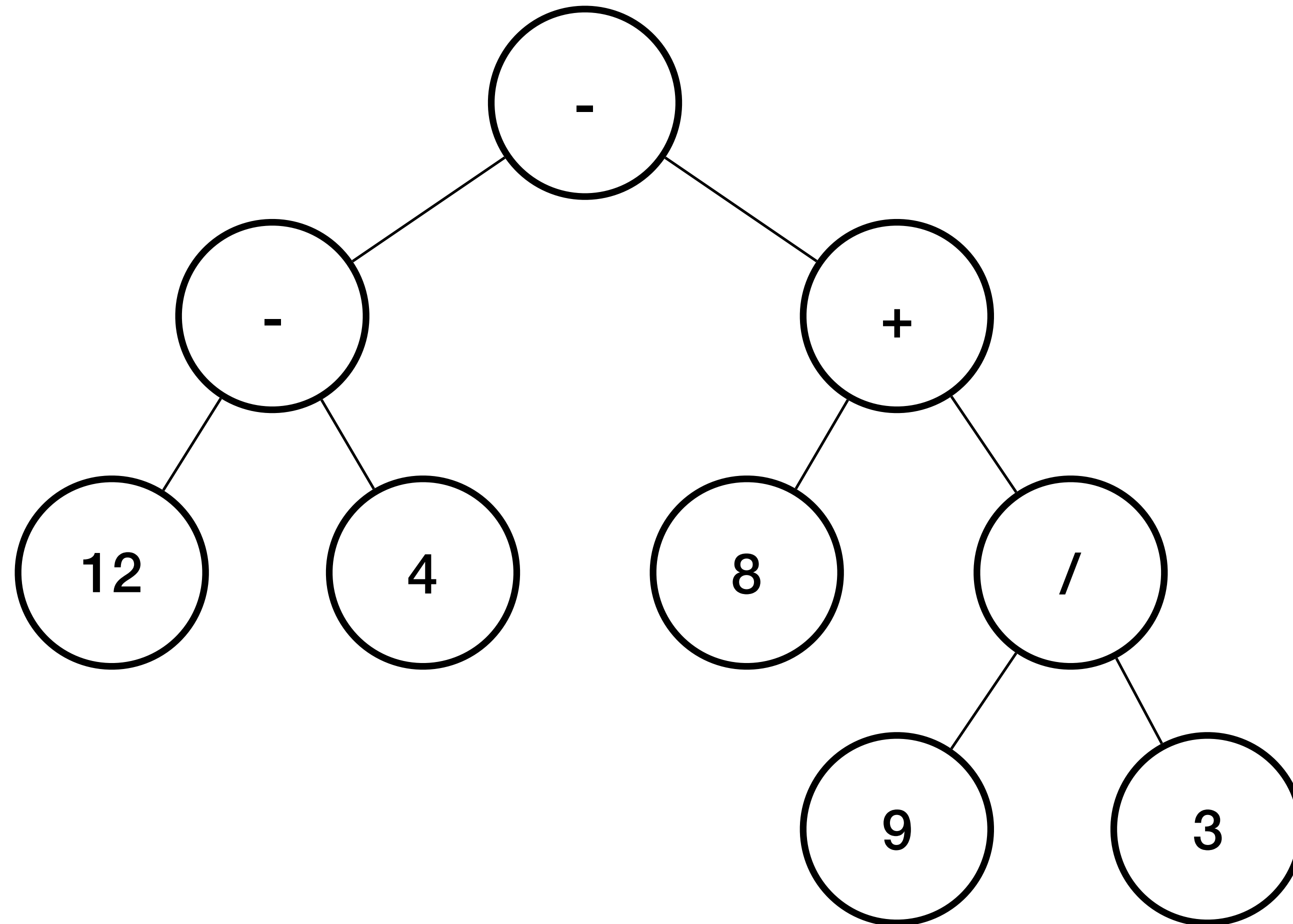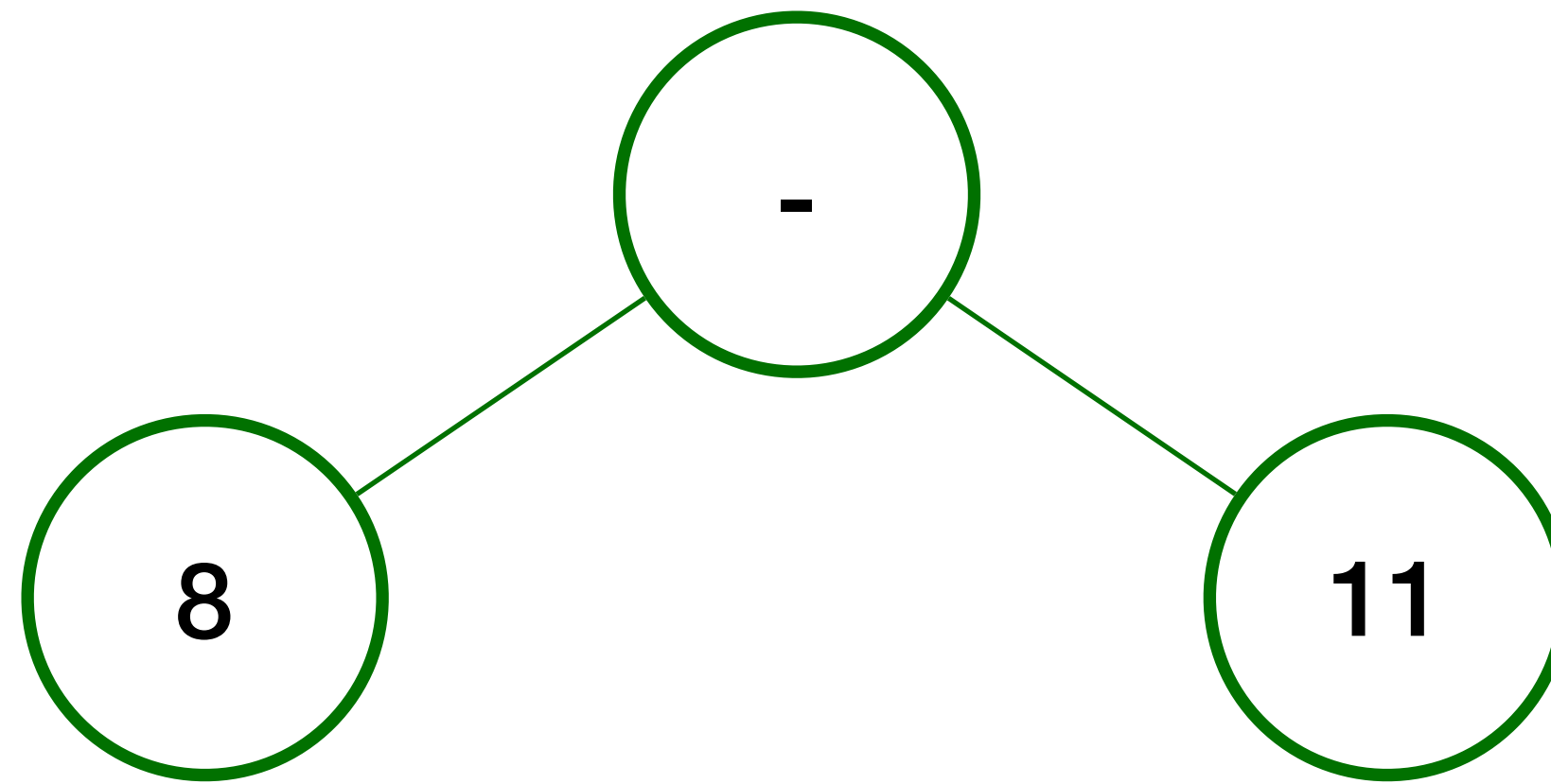
Evaluating Expression Tree
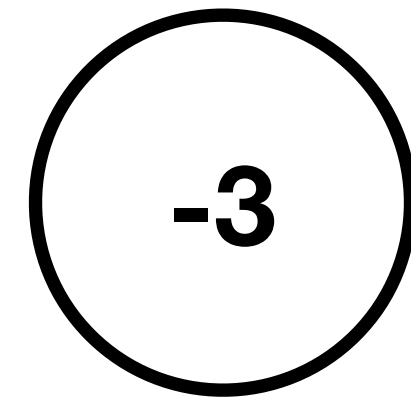
Evaluating Expression Tree

Evaluating Expression Tree

Evaluating Expression Tree

Evaluating Expression Tree

-3

Expression Tree Traversals

- Modified in-order traversal that adds parentheses around each operator

- Generates a fully parenthesized infix expression

- ((12-4)-(8+(9/3)))

```scala
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(")")
    }
  }
}
```
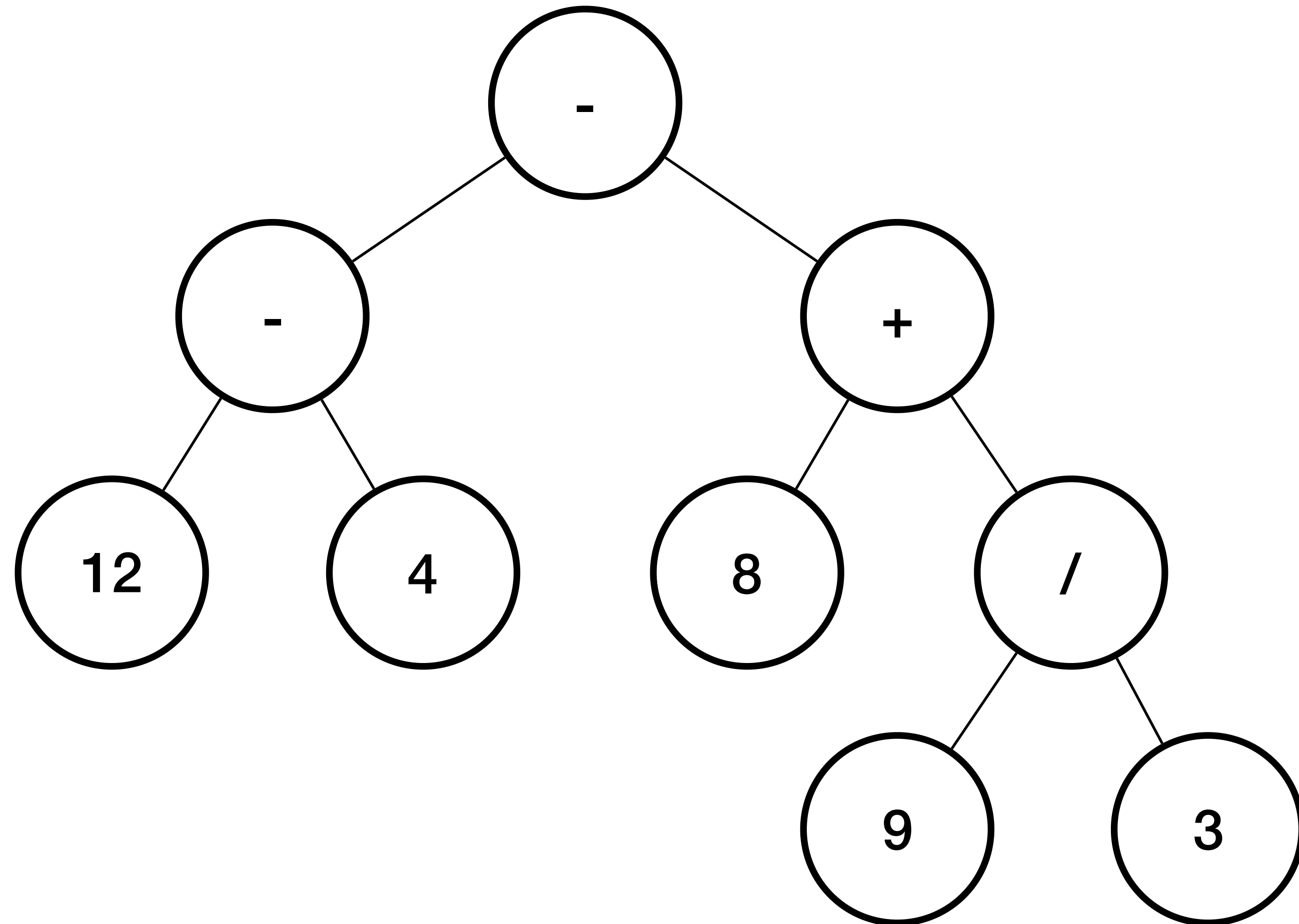
Expression Tree Traversals

- Unmodified post-order traversal generates a postfix express (Reverse Polish Notation)

- 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 -