Binary Trees and Traversals

Lecture Task

- Enemy AI: Lecture Task 3 -

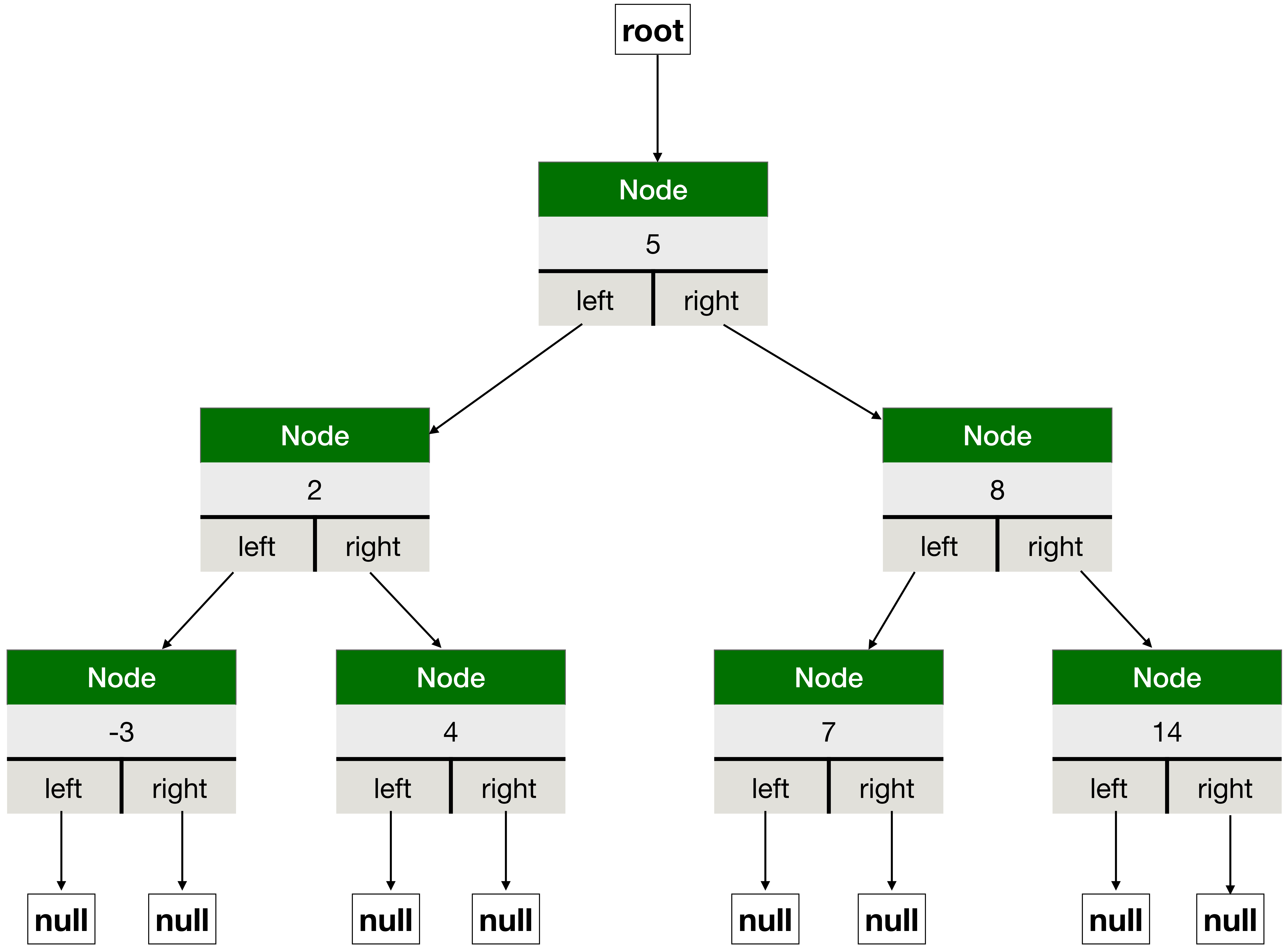
Functionality: In the game.enemyai.AIPlayer class, implement the following method:

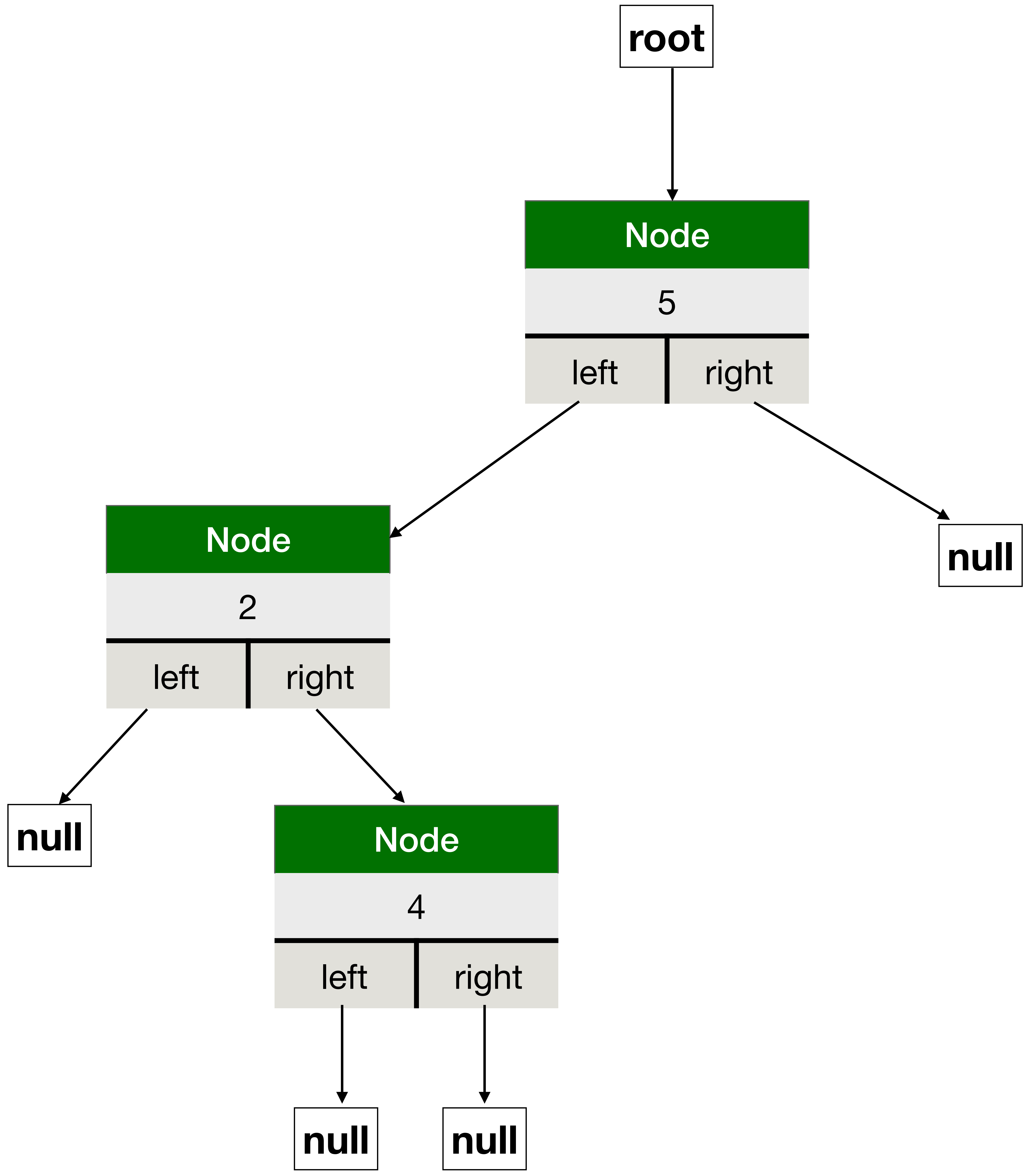
- A method named “computePath” with:
  - Two parameters of type GridLocation representing the start then end of the path to compute
  - Returns a path from the start to end locations as a LinkedListNode of GridLocations
    - Valid path connections only travel up, down, left, and right. Diagonal moves are not allowed
      - Ex. (1,3) -> (1,2) -> (2,2) is a valid path from (1,3) to (2,2)
    - The returned path must contain the minimal number of GridLoactions possible
      - For testing, keep in mind that there are many valid paths that can be returned. As long as the path is valid and has the minimal possible length, it should pass your tests

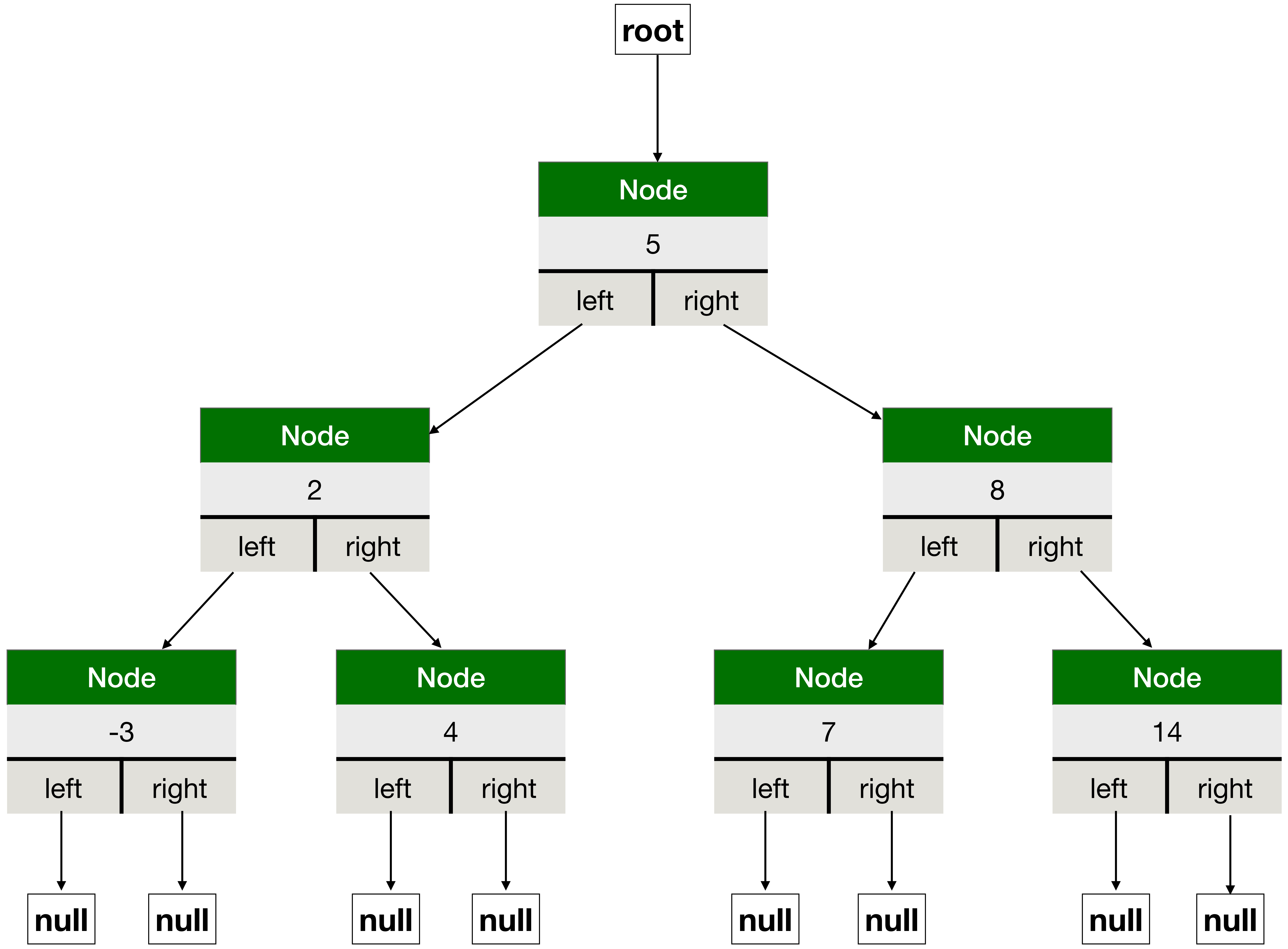
Testing: In the tests package, complete the test suite named LectureTask3 that tests this functionality.

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
2
left
right
Node
8
left
right
Node
-3
left
right
Node
4
left
right
Node
7
left
right
Node
14
left
right
null
null
null
null
null
null
null
null


root
Node
5
left
right
null
Node
2
left
right
null
Node
4
left
right
null
null


root
Node
5
left
right
Node
2
left
right
Node
8
left
right
Node
-3
left
right
Node
4
left
right
Node
7
left
right
Node
14
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

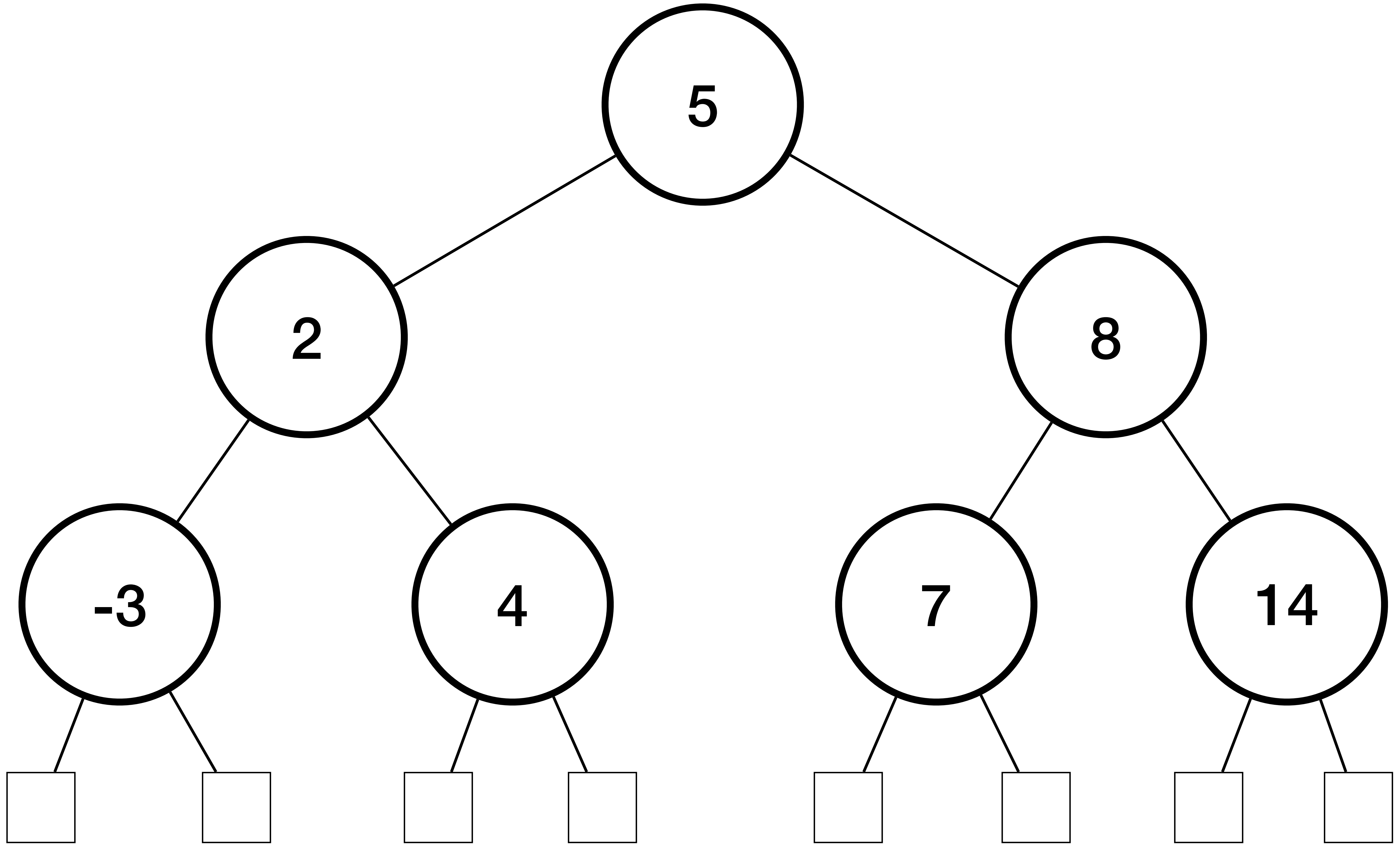
```
def preOrderTraversal[A](node: BinaryTreeNode[A], f: A => Unit): Unit = {
  if(node != null) {
    f(node.value)
    preOrderTraversal(node.left, f)
    preOrderTraversal(node.right, f)
  }
}
```

```
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

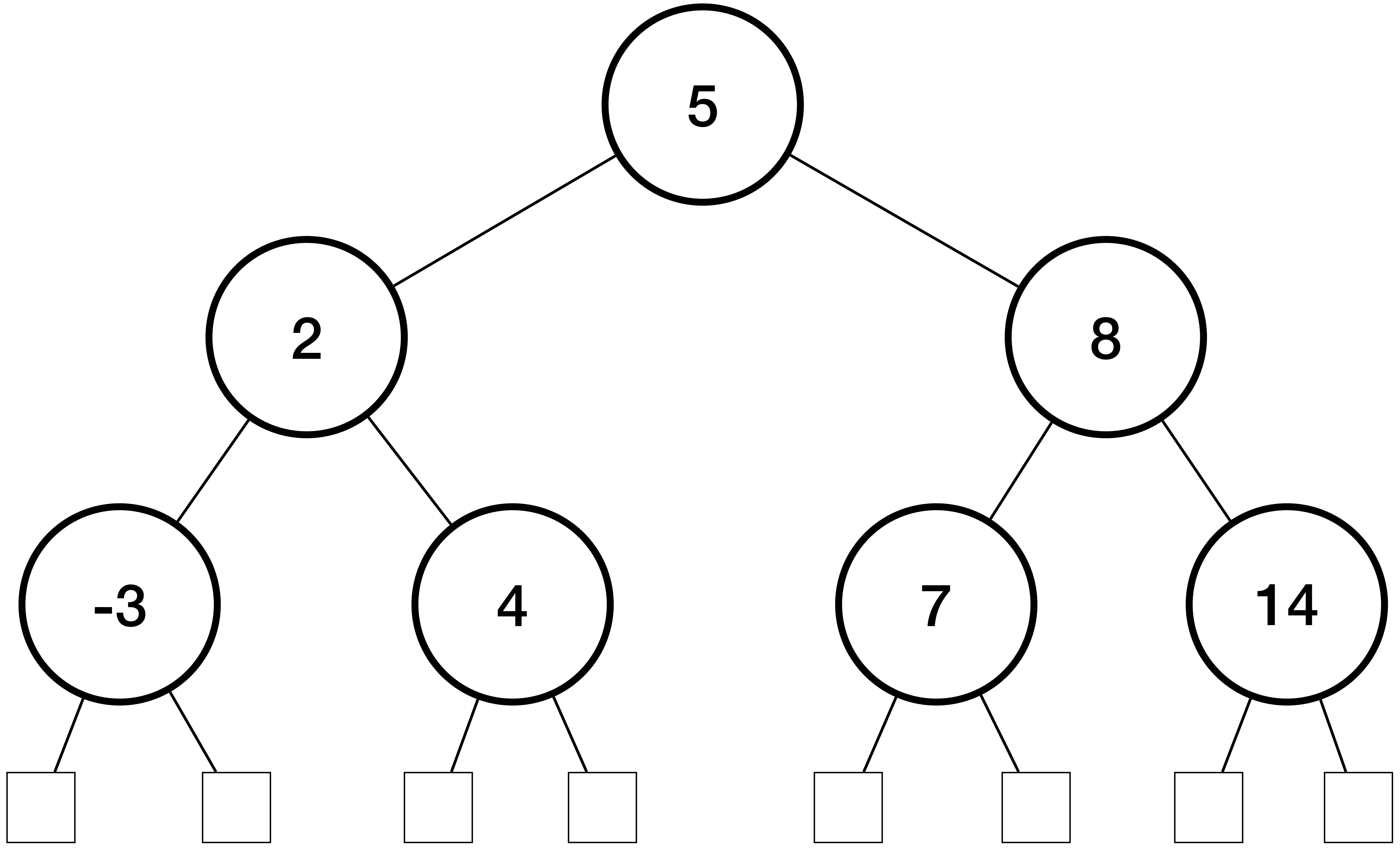
```
def postOrderTraversal[A](node: BinaryTreeNode[A], f: A => Unit): Unit = {
  if(node != null) {
    postOrderTraversal(node.left, f)
    postOrderTraversal(node.right, f)
    f(node.value)
  }
}
```

```
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

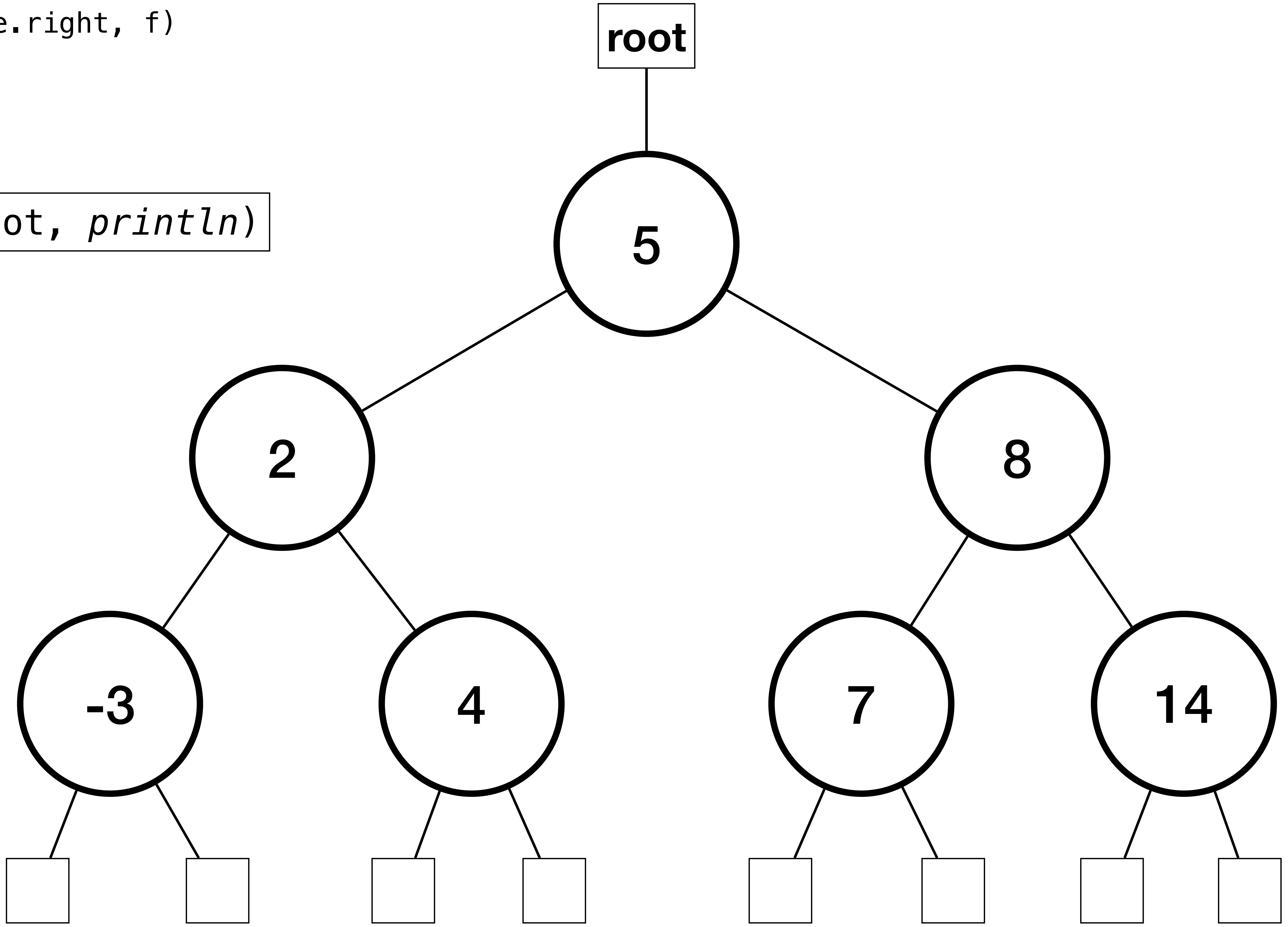
```scala
def inOrderTraversal[A](node: BinaryTreeNode[A], f: A => Unit): Unit = {
  if(node != null) {
    inOrderTraversal(node.left, f)
    f(node.value)
    inOrderTraversal(node.right, f)
  }
}

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)
  }
}
```

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:

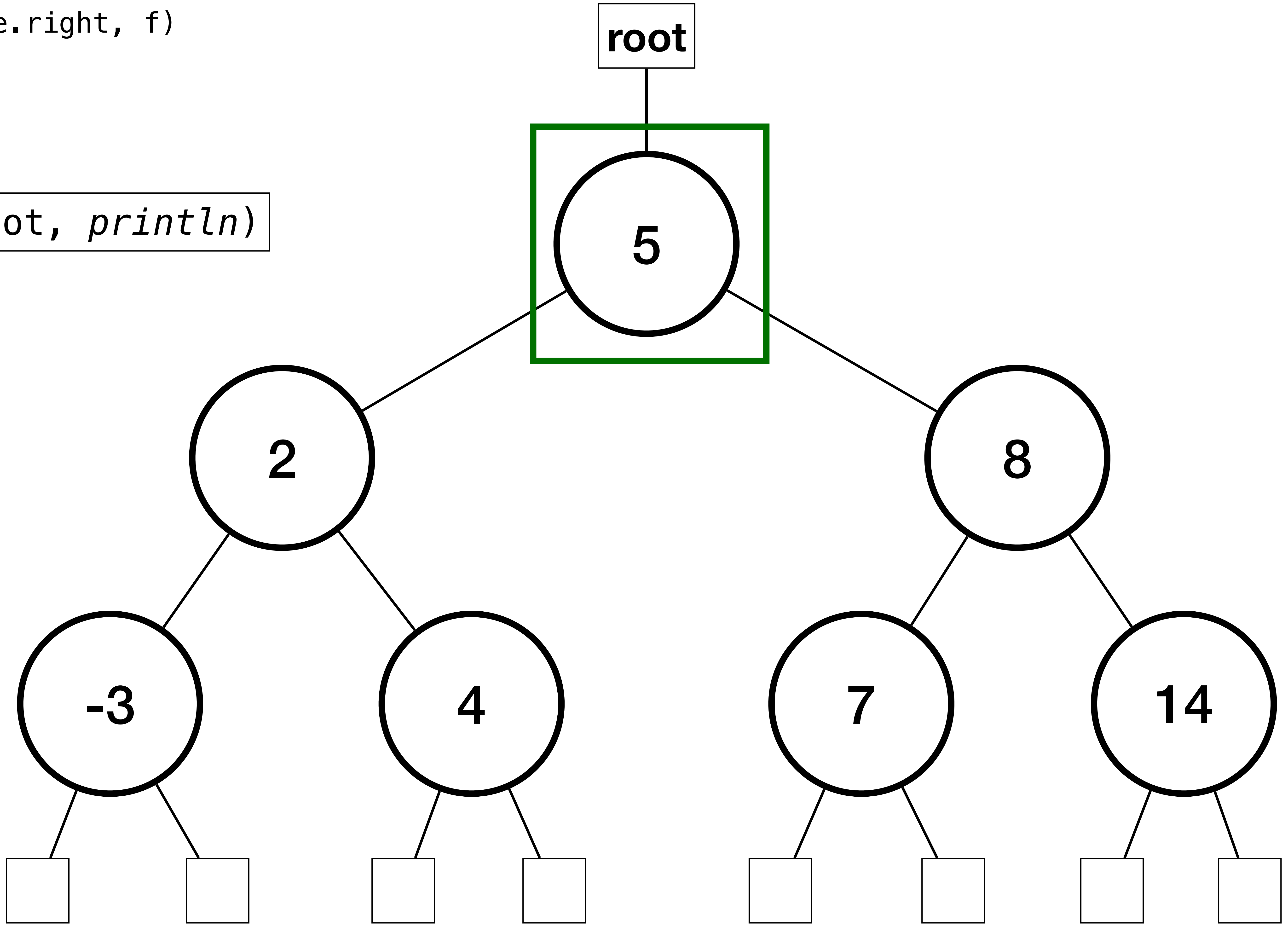


Traversals

```
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:

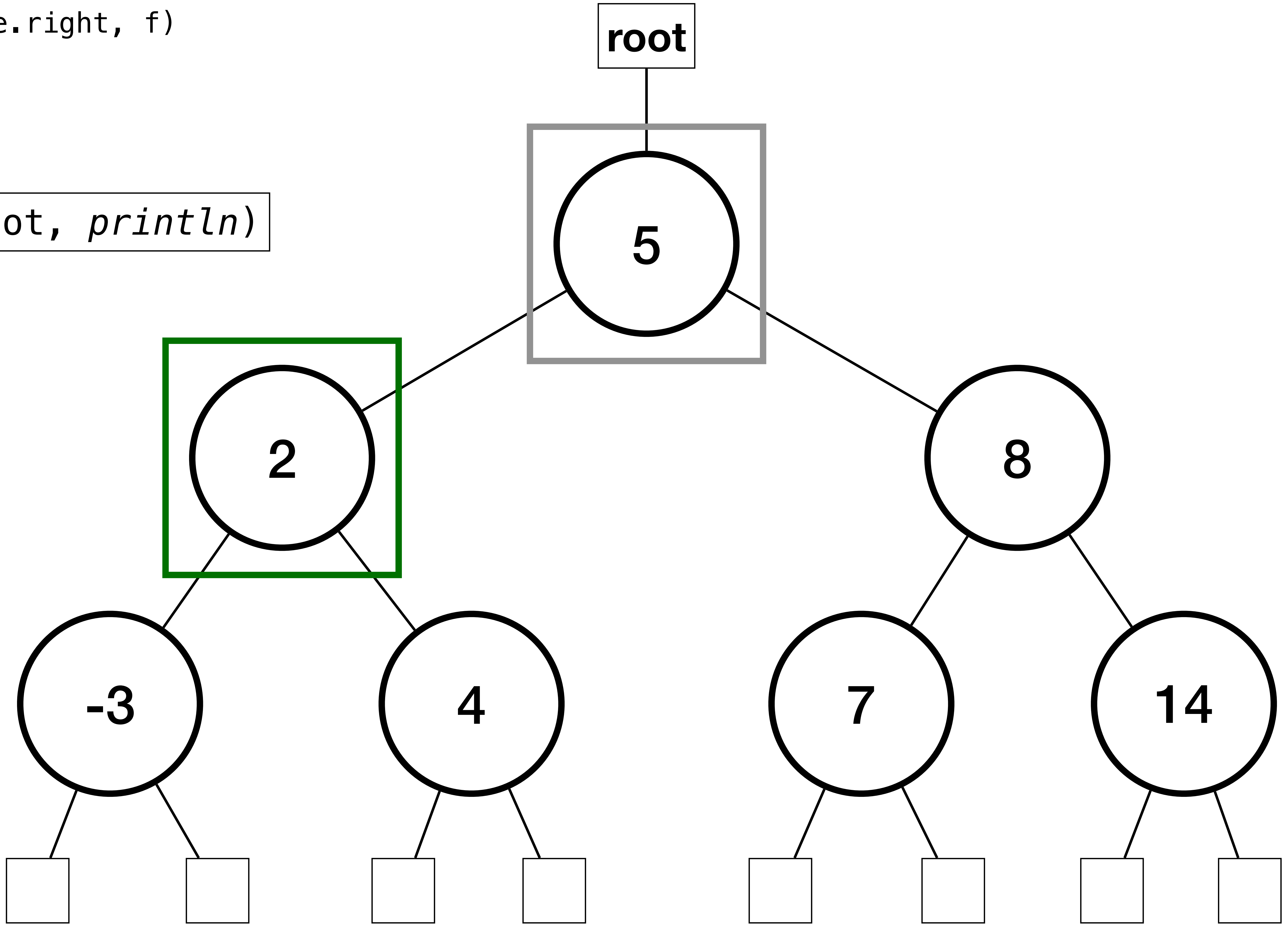


Traversals

```
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:

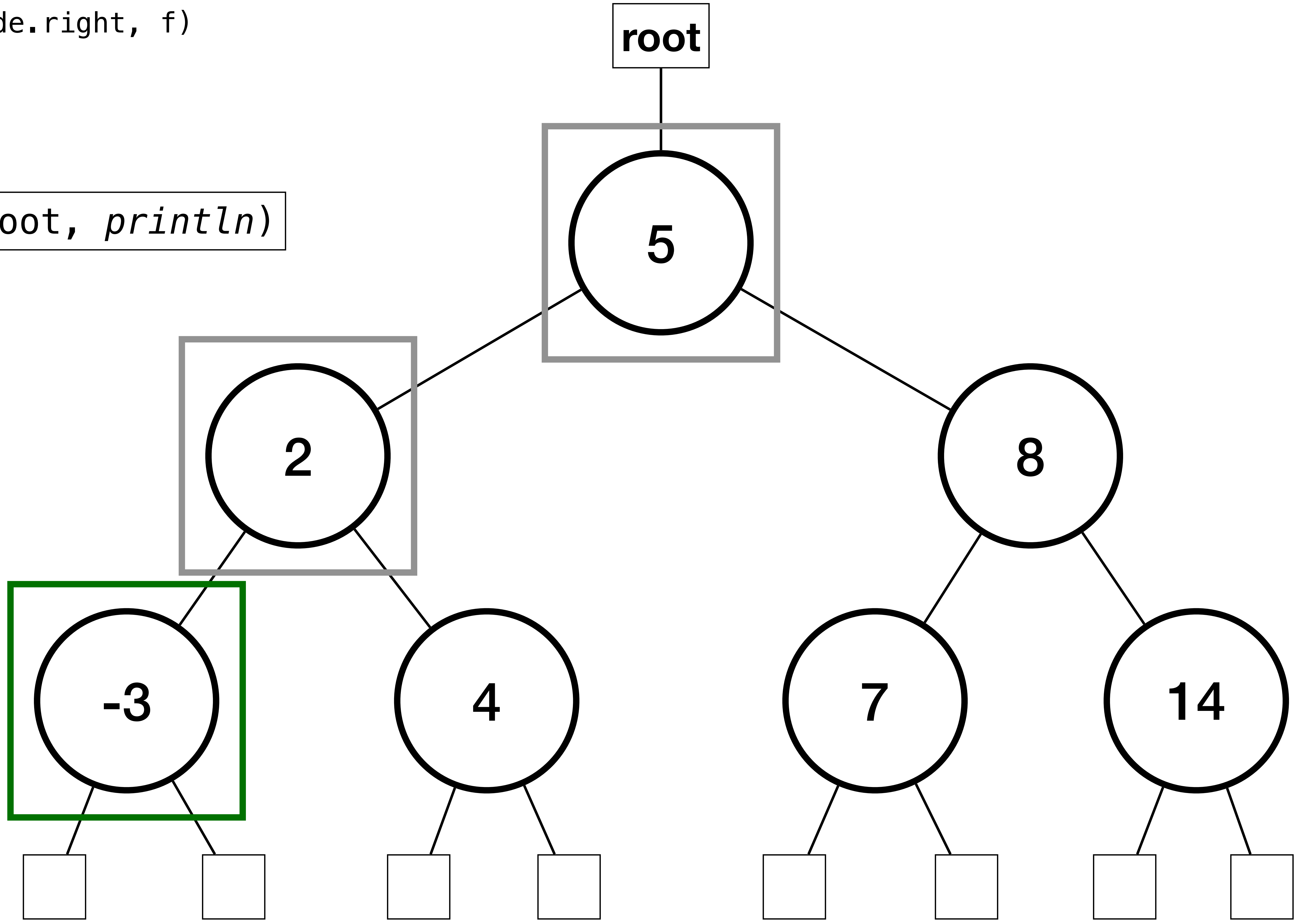


Traversals

```
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:

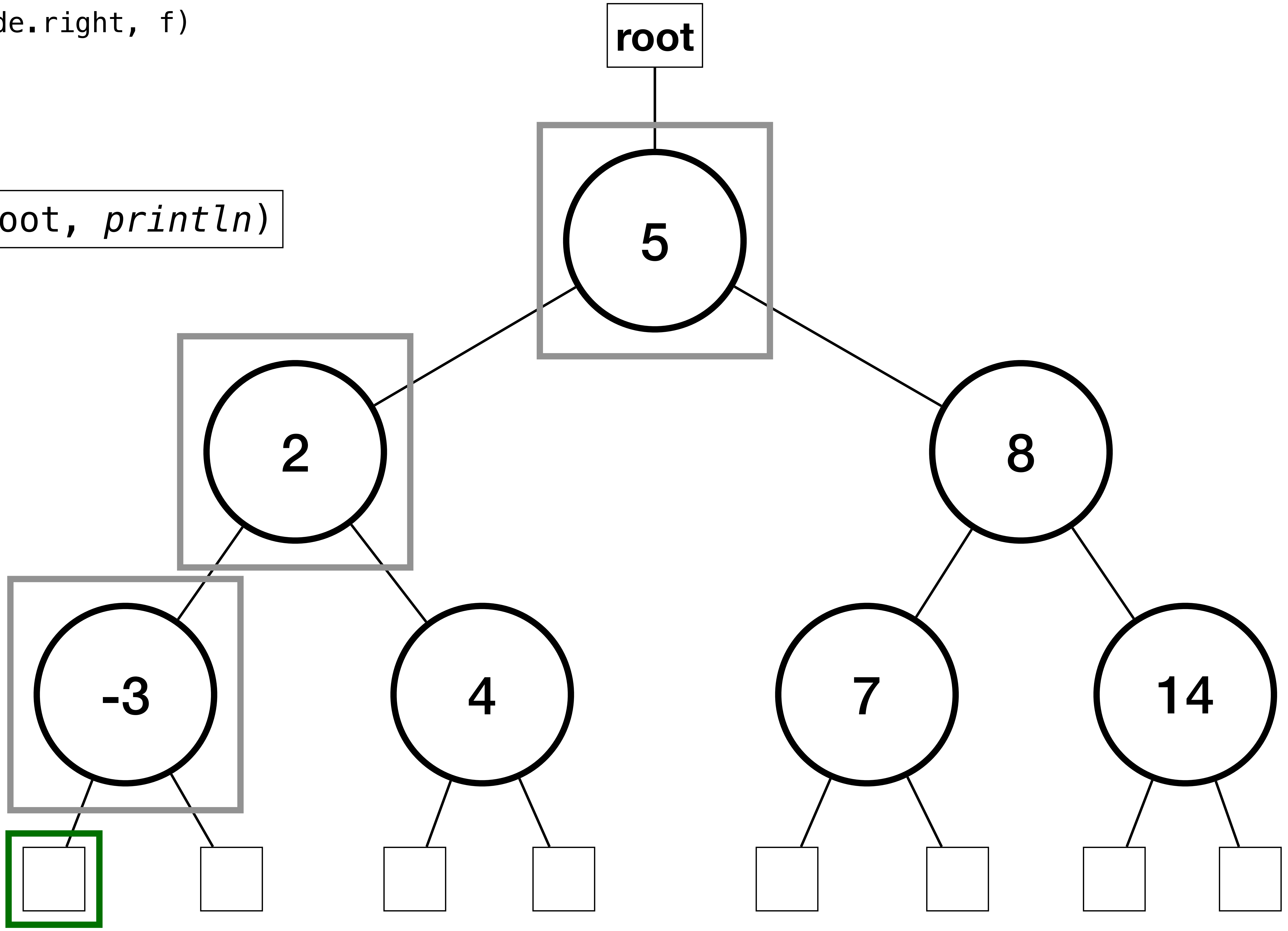


Traversals

```
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:

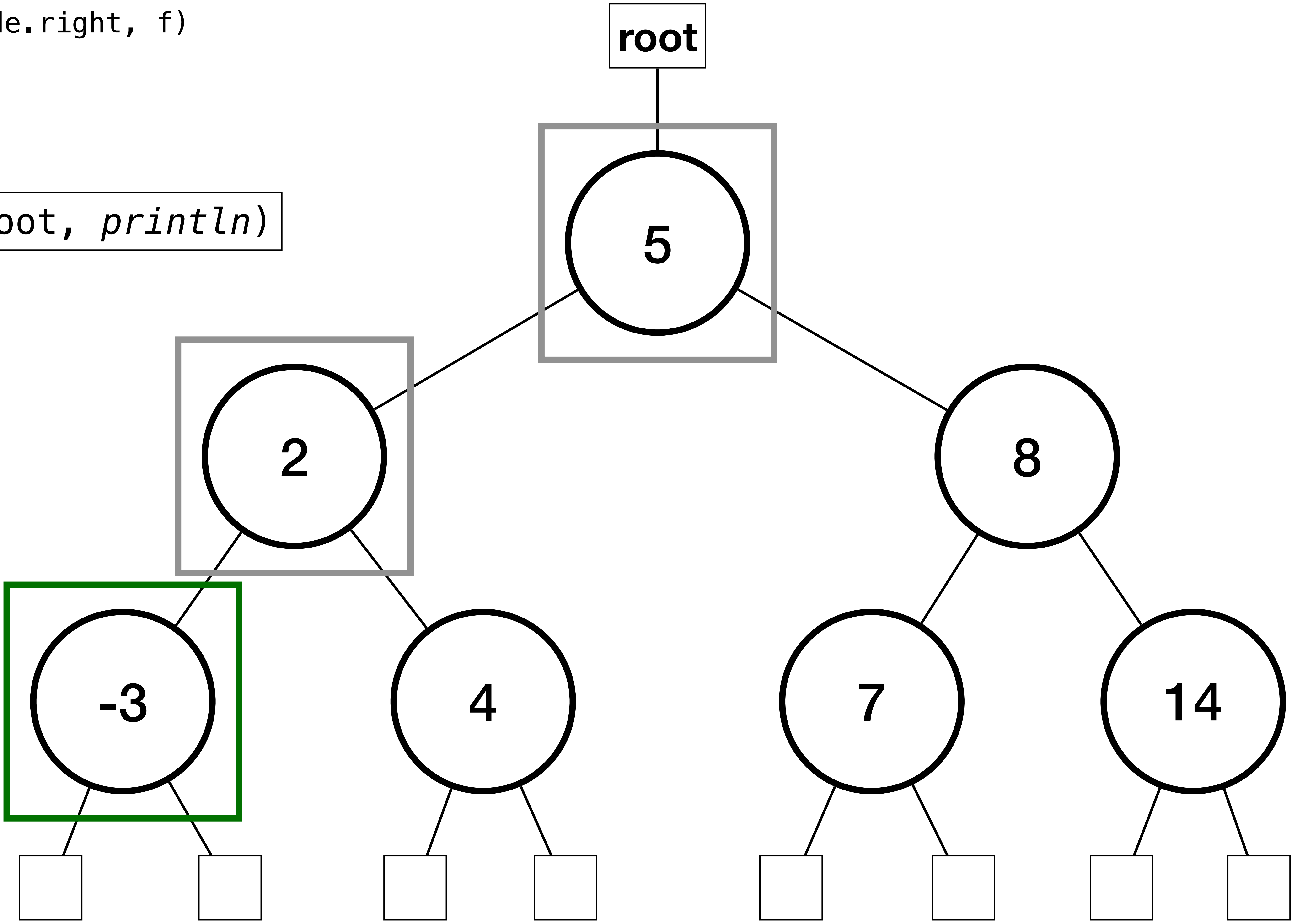


Traversals

```
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

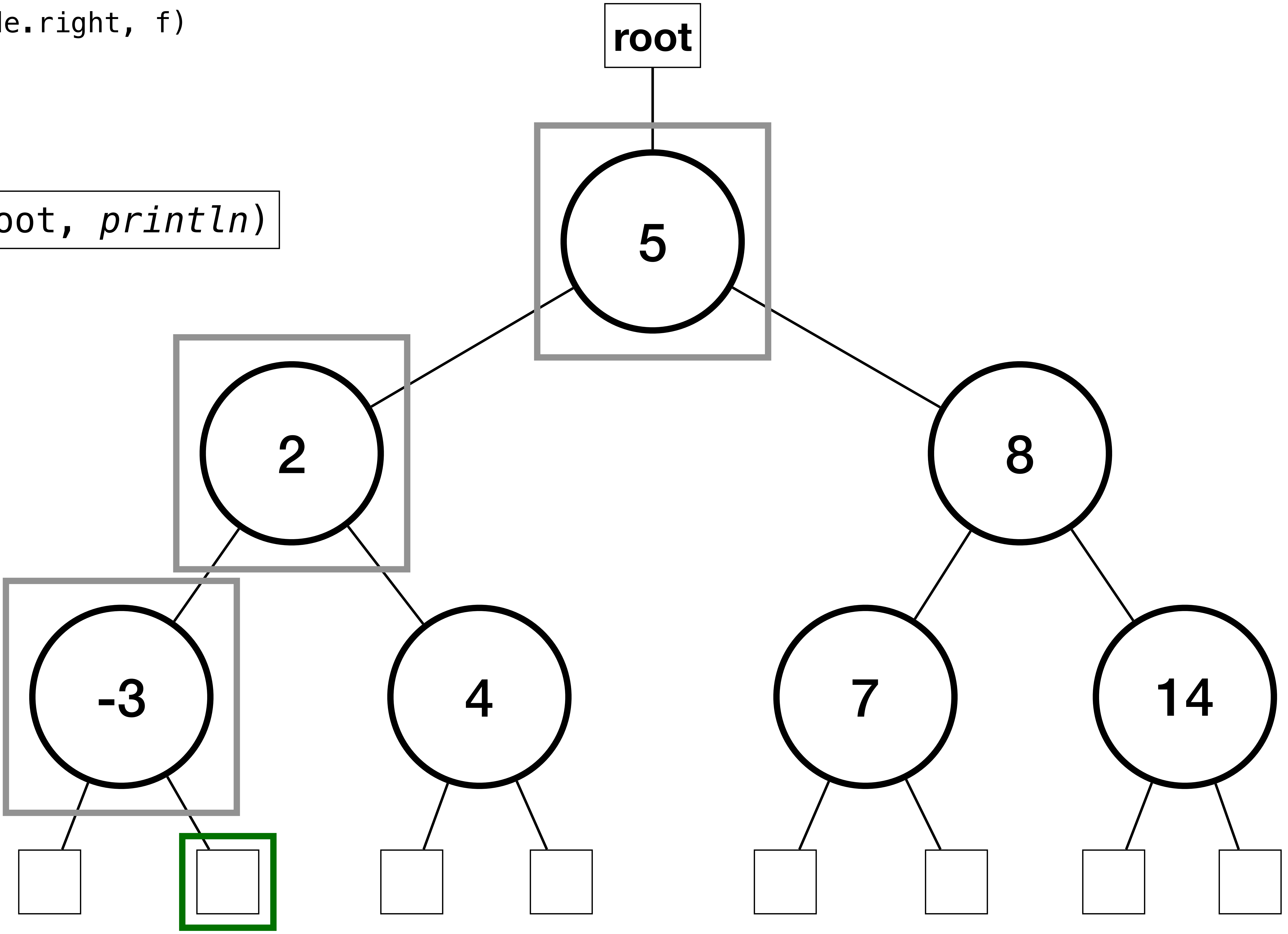


Traversals

```
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

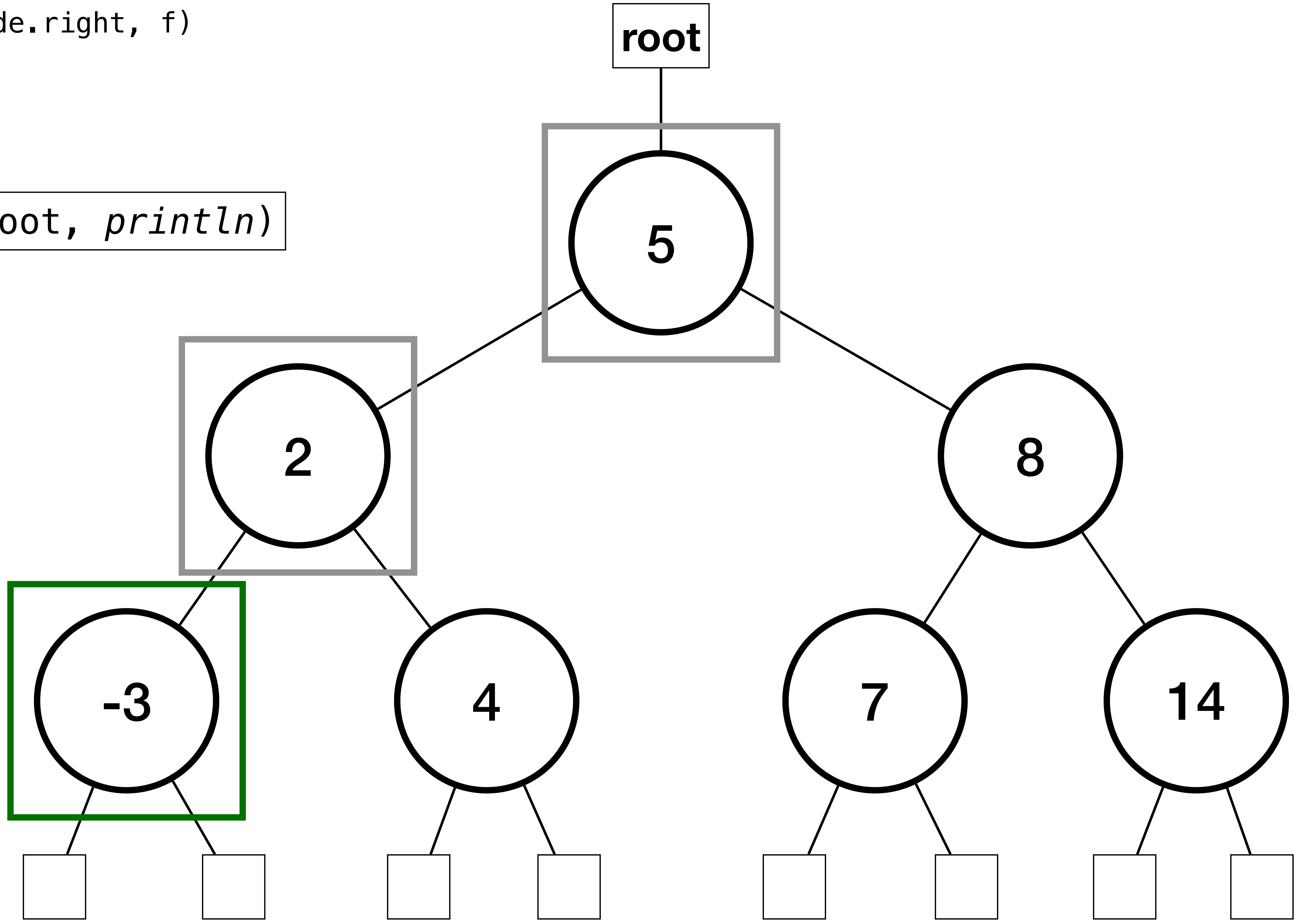


Traversals

```
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

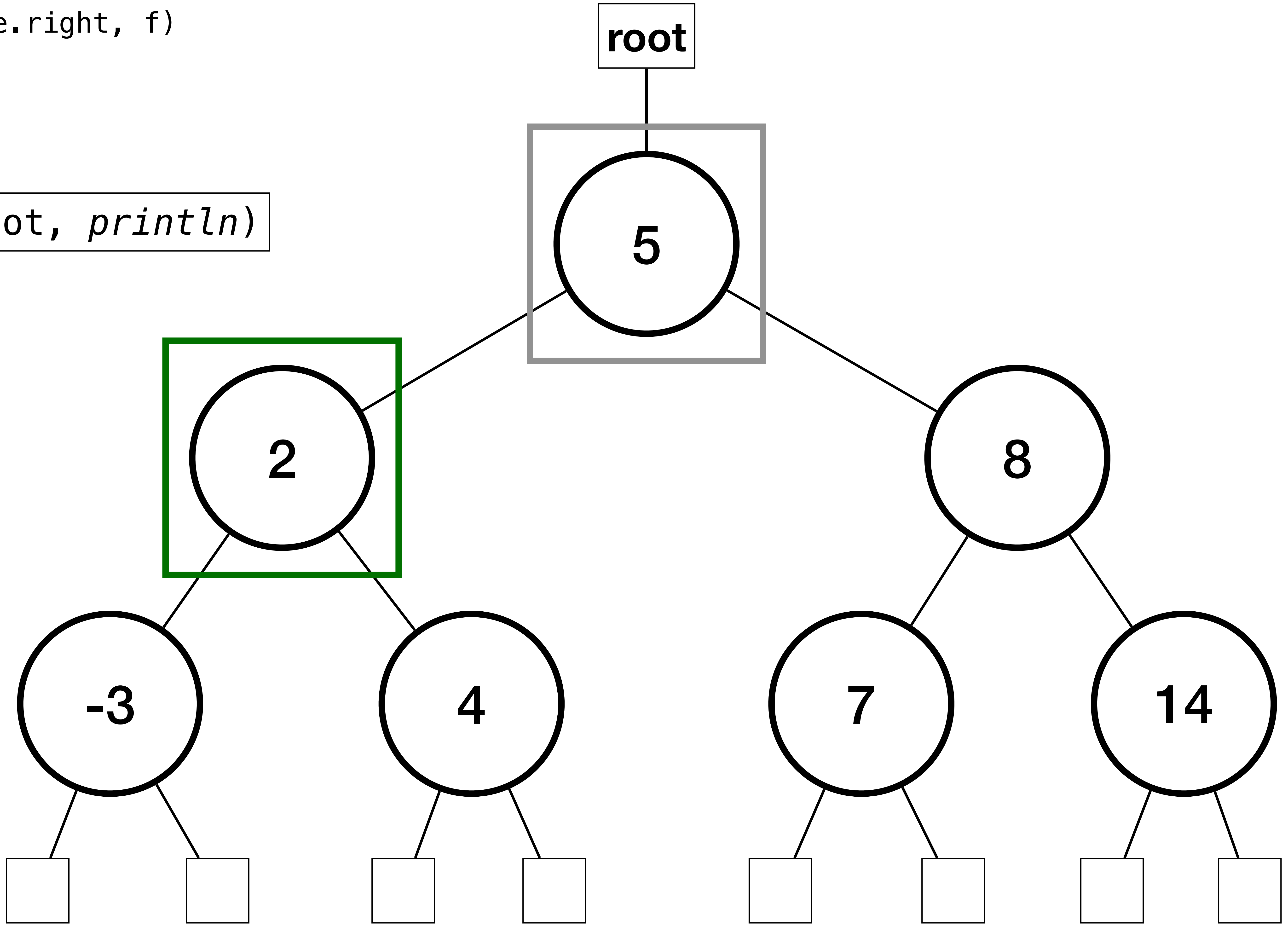


Traversals

```
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

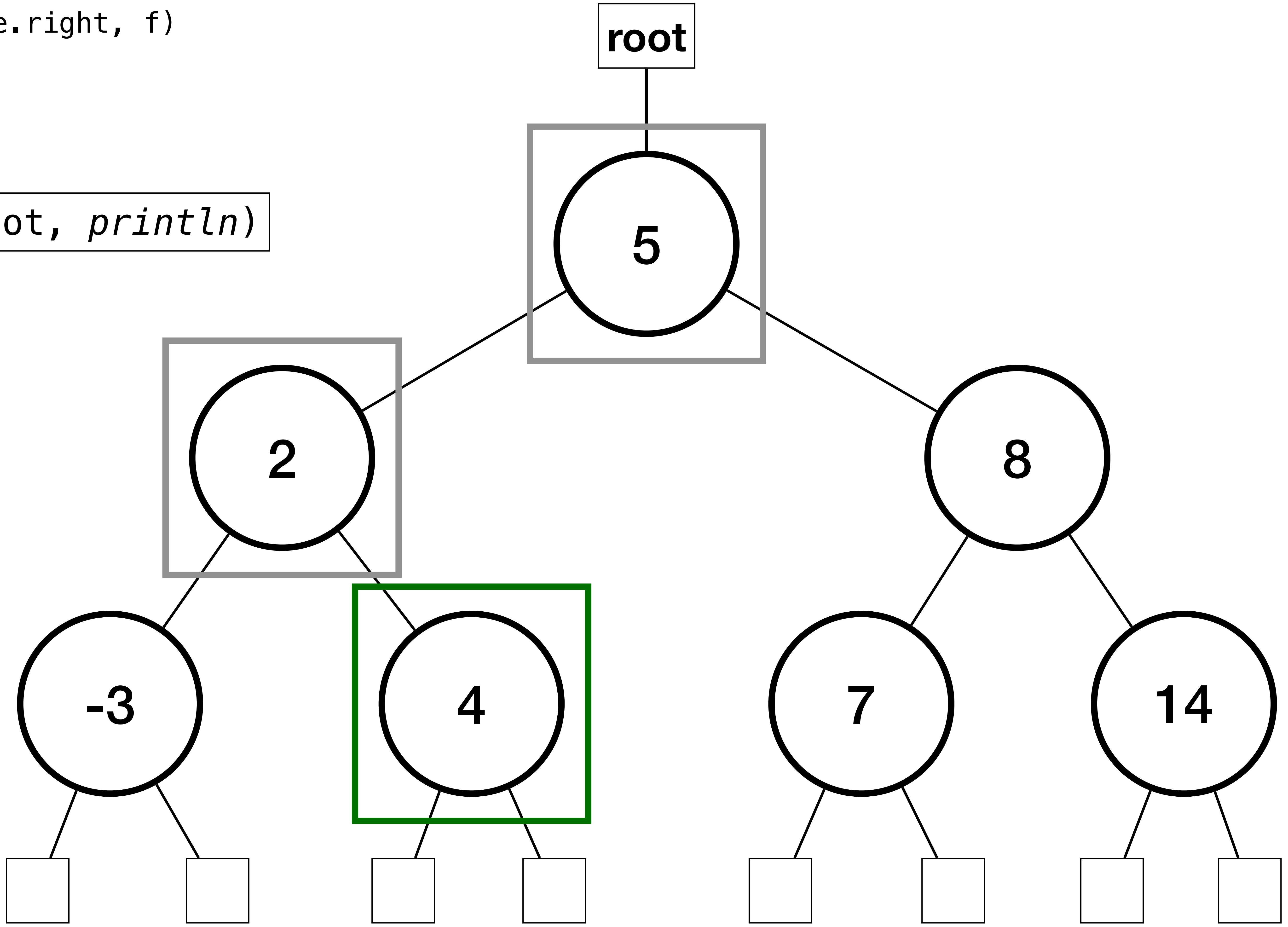


Traversals

```
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

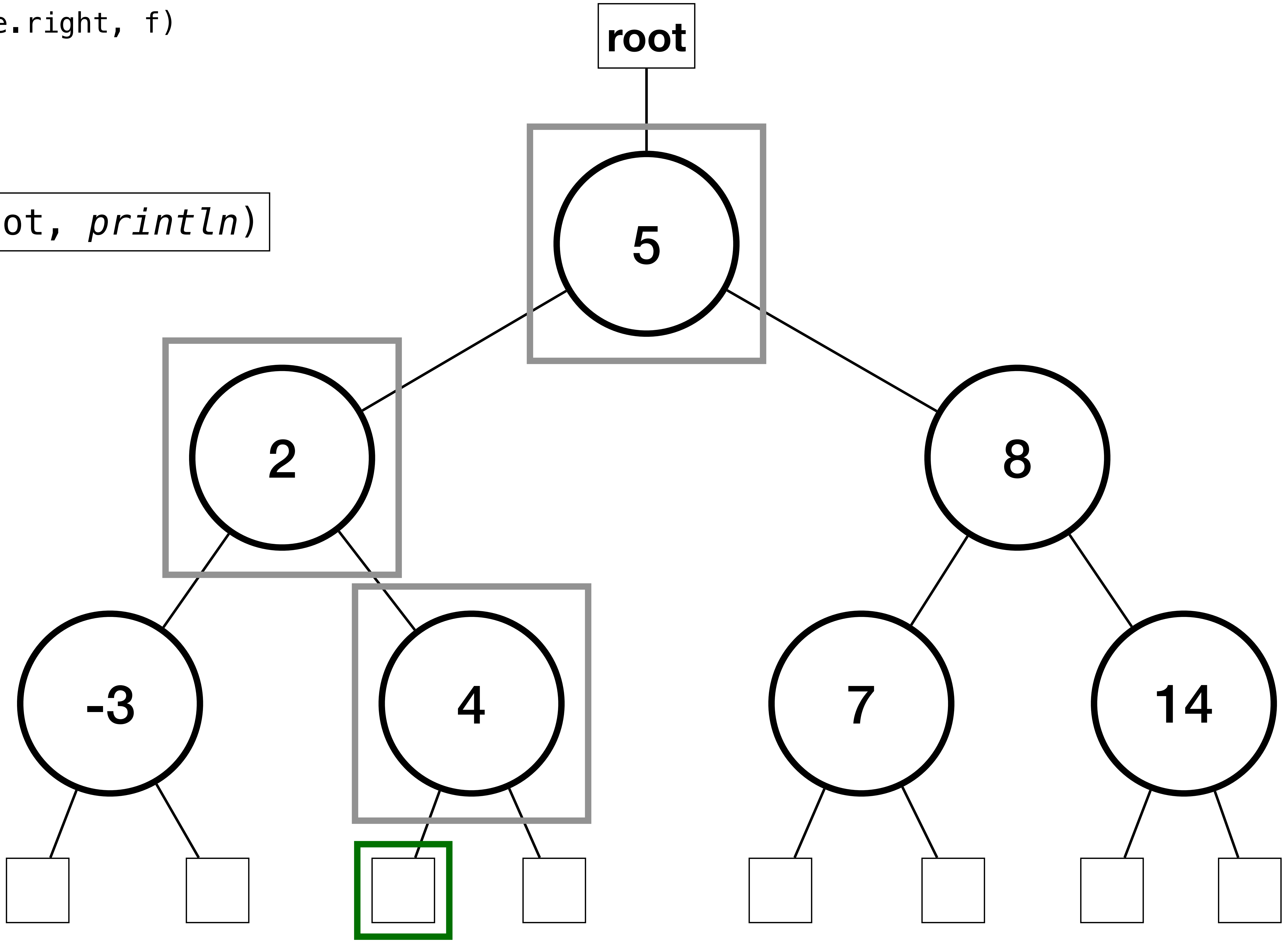


Traversals

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

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

Printed:
-3
2

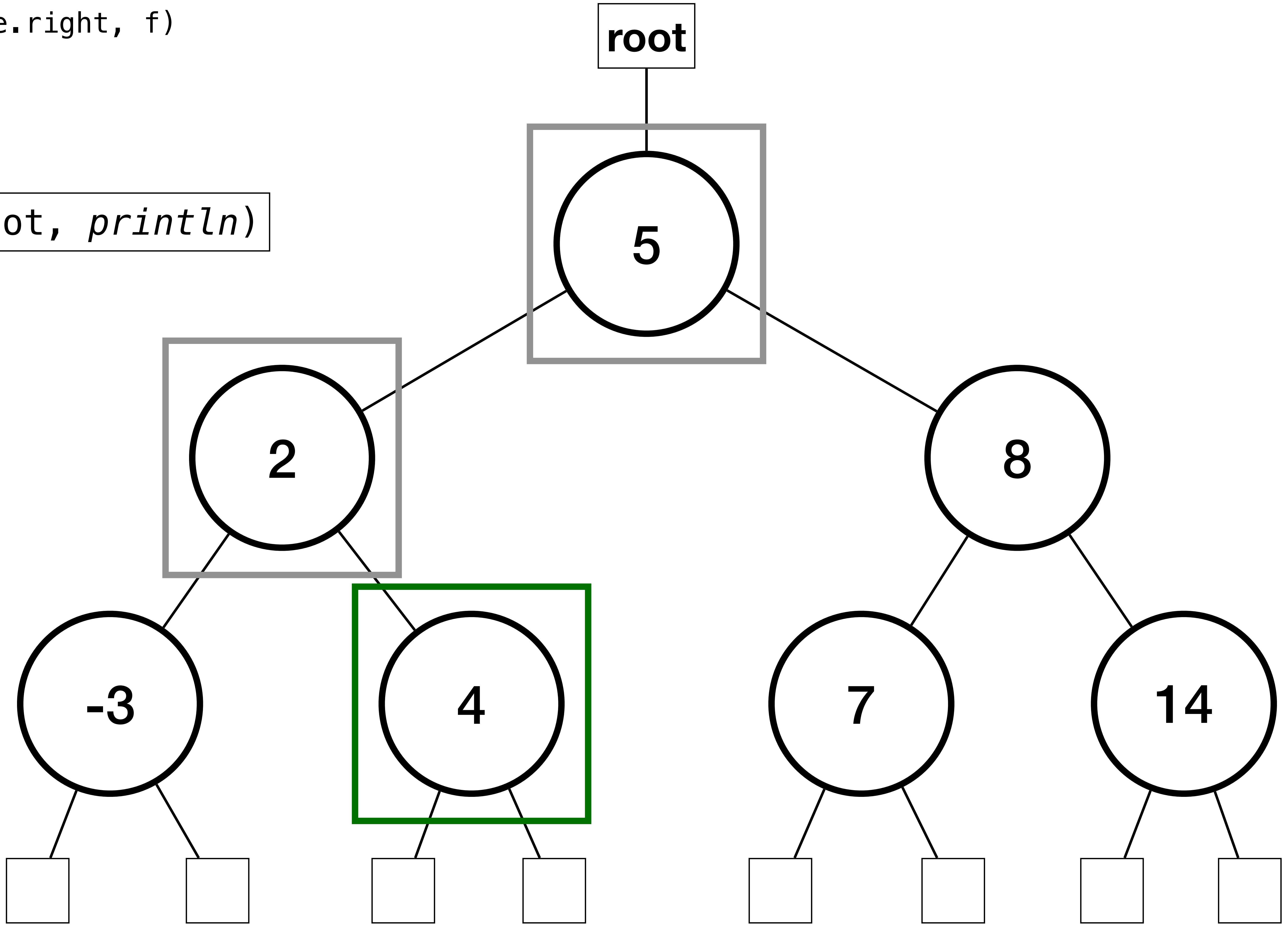


Traversals

```
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

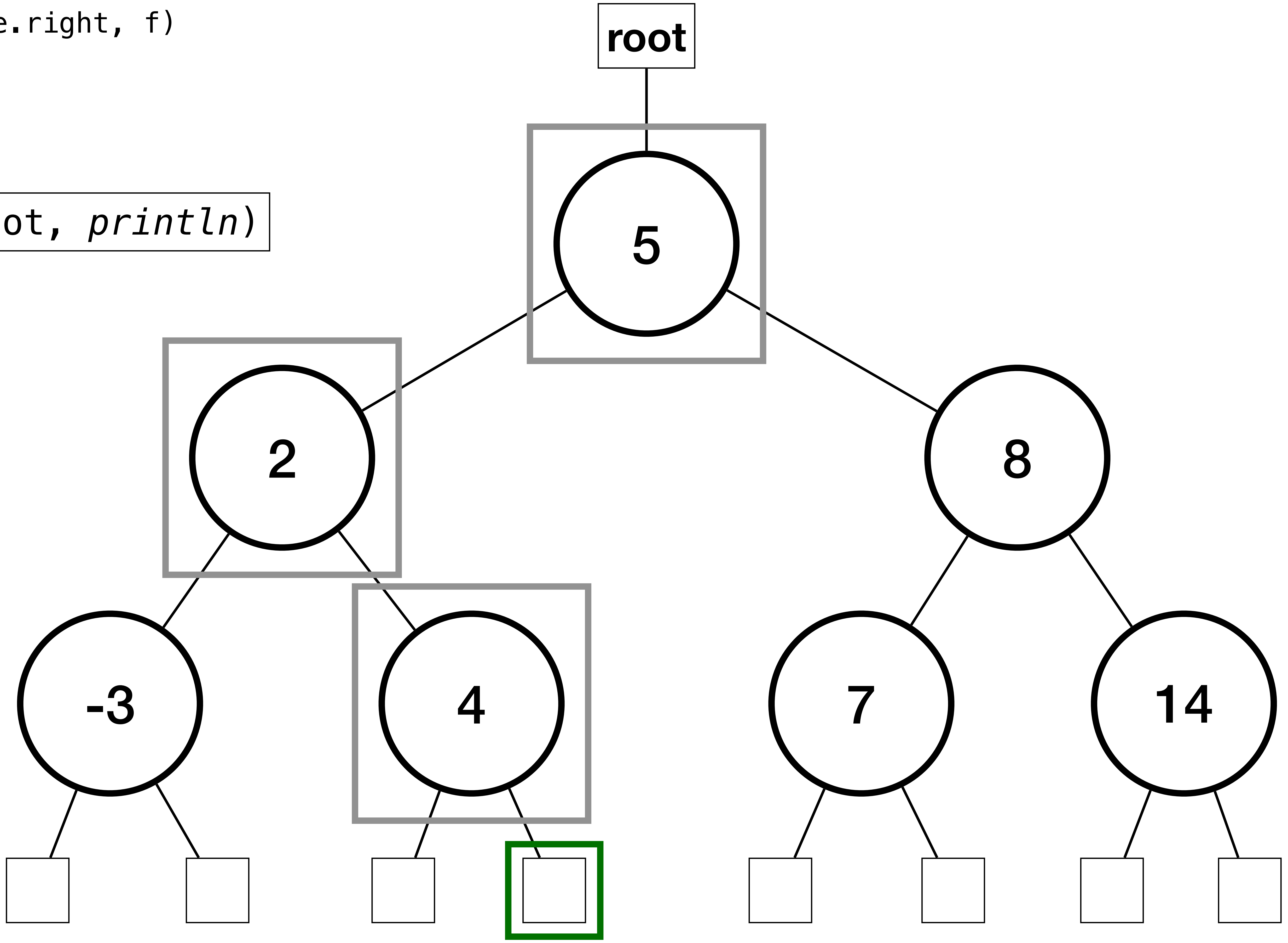


Traversals

```
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

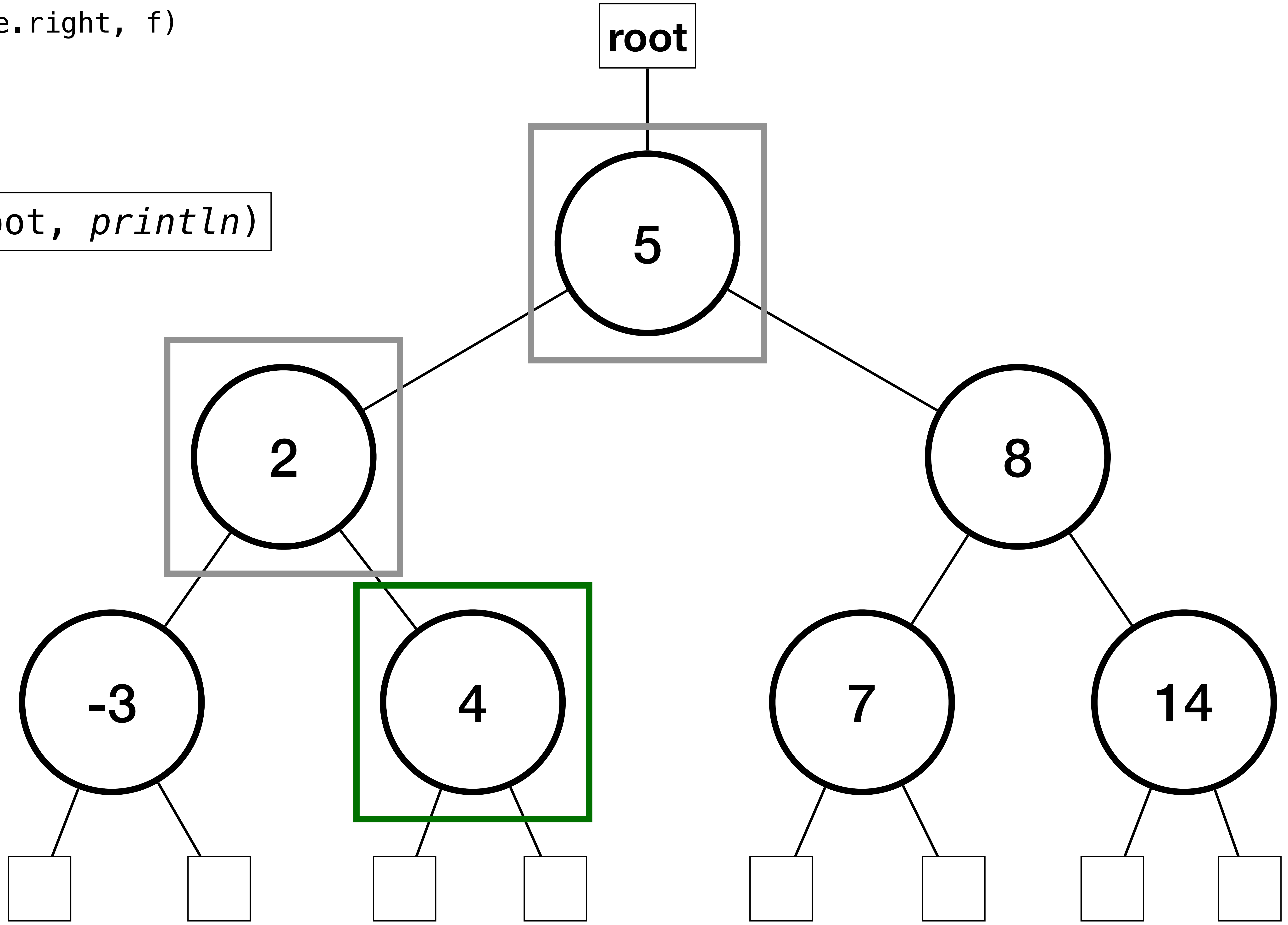


Traversals

```
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

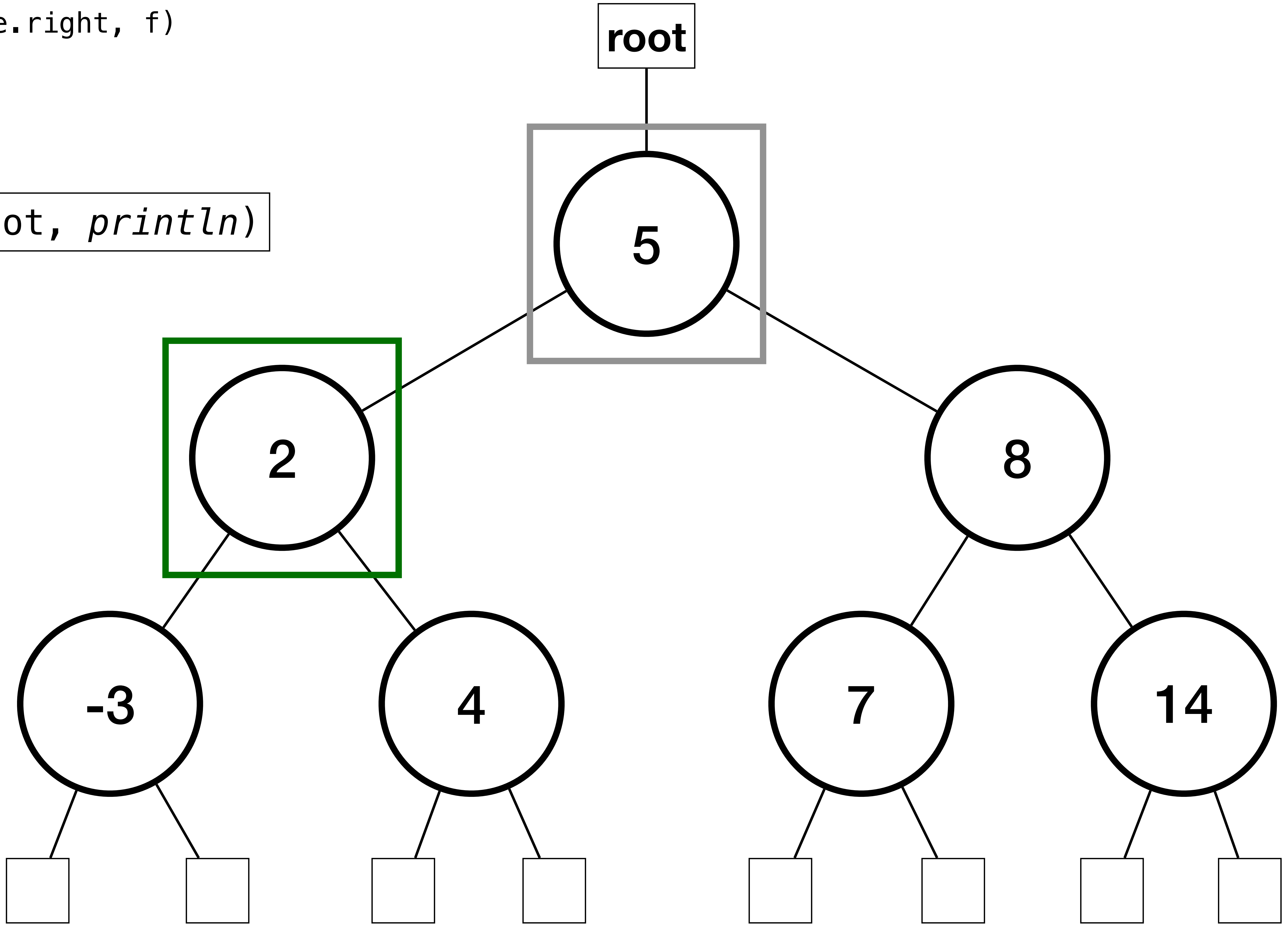


Traversals

```
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

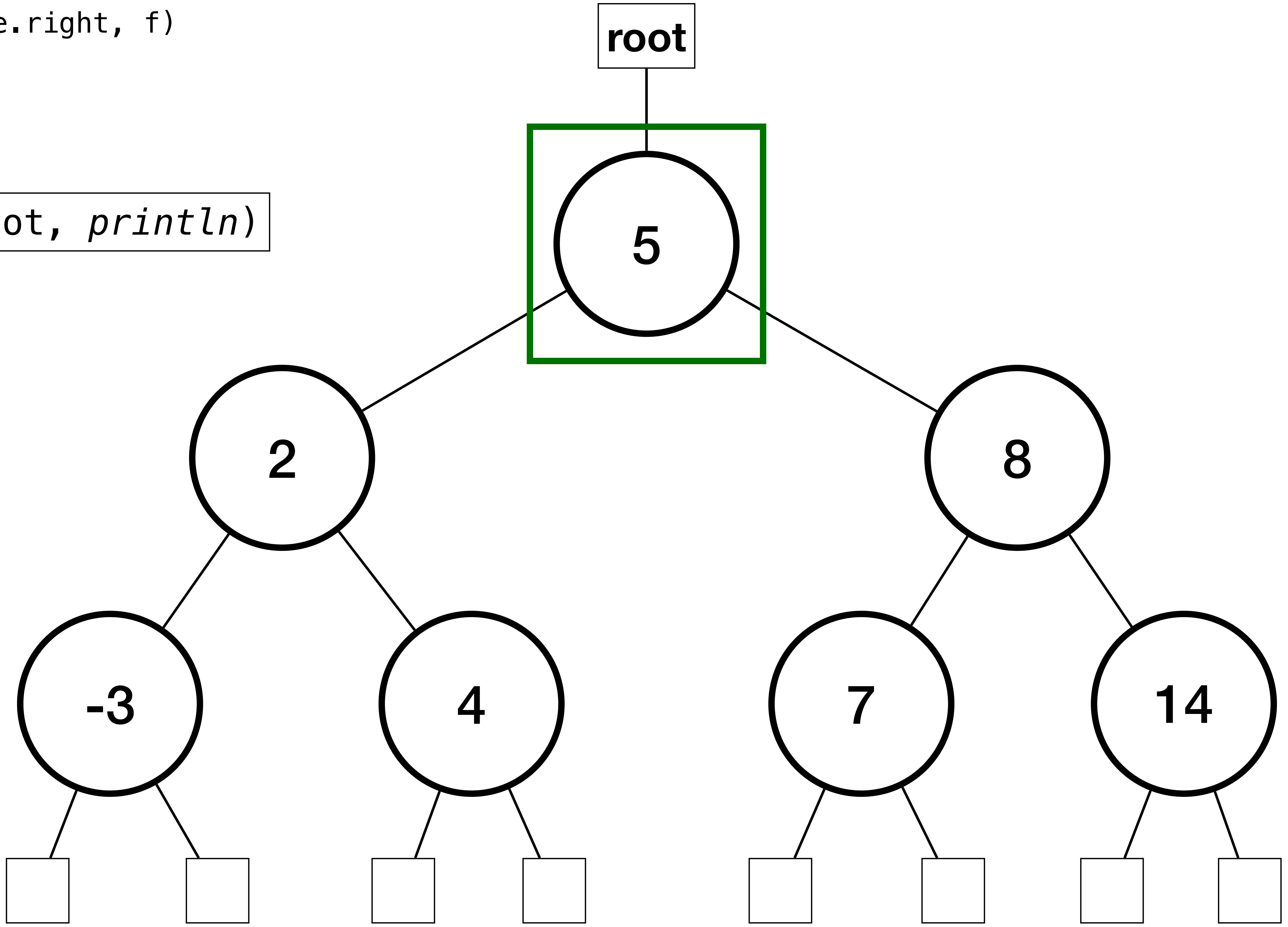


Traversals

```
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

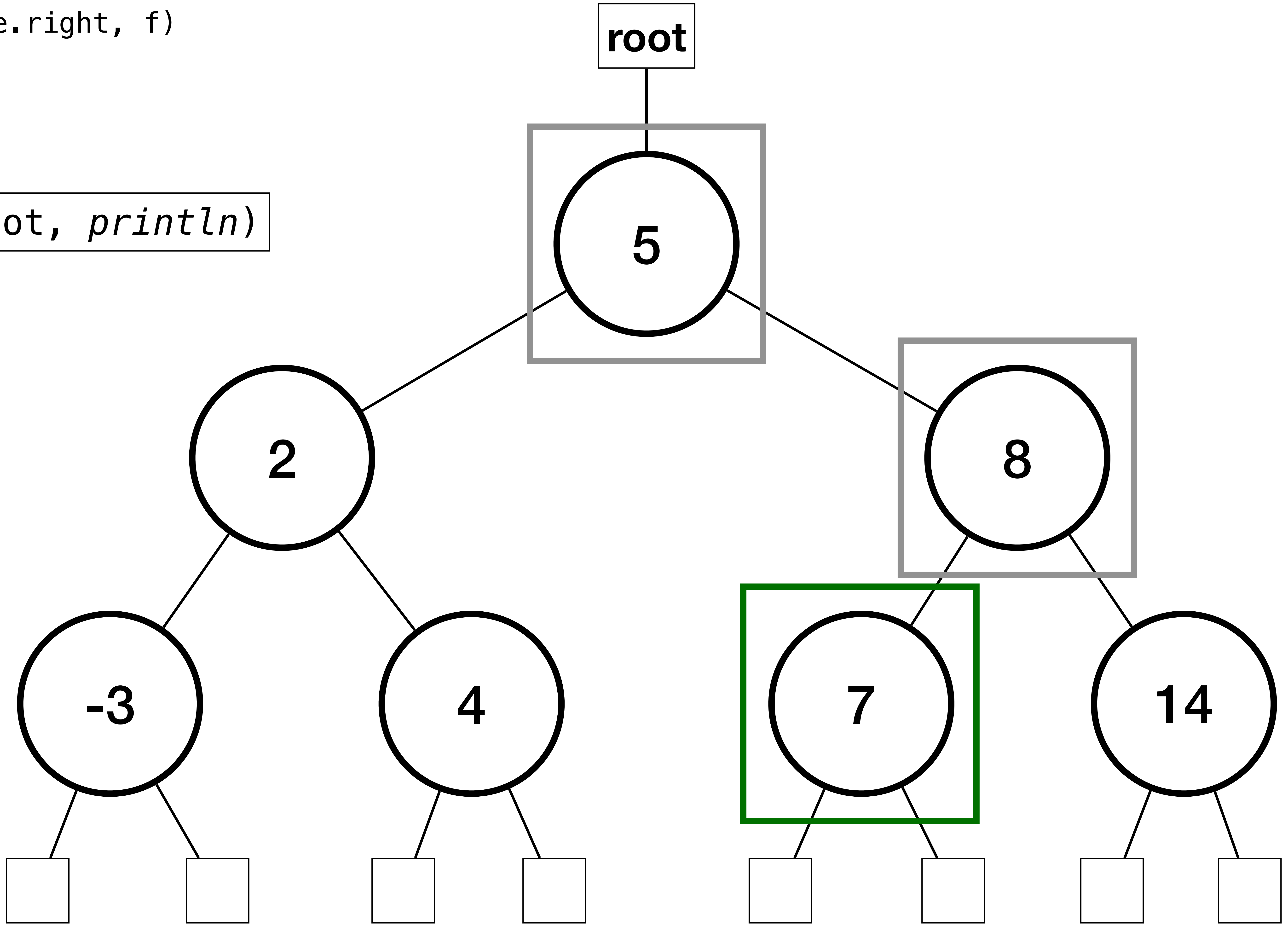


Traversals

```
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

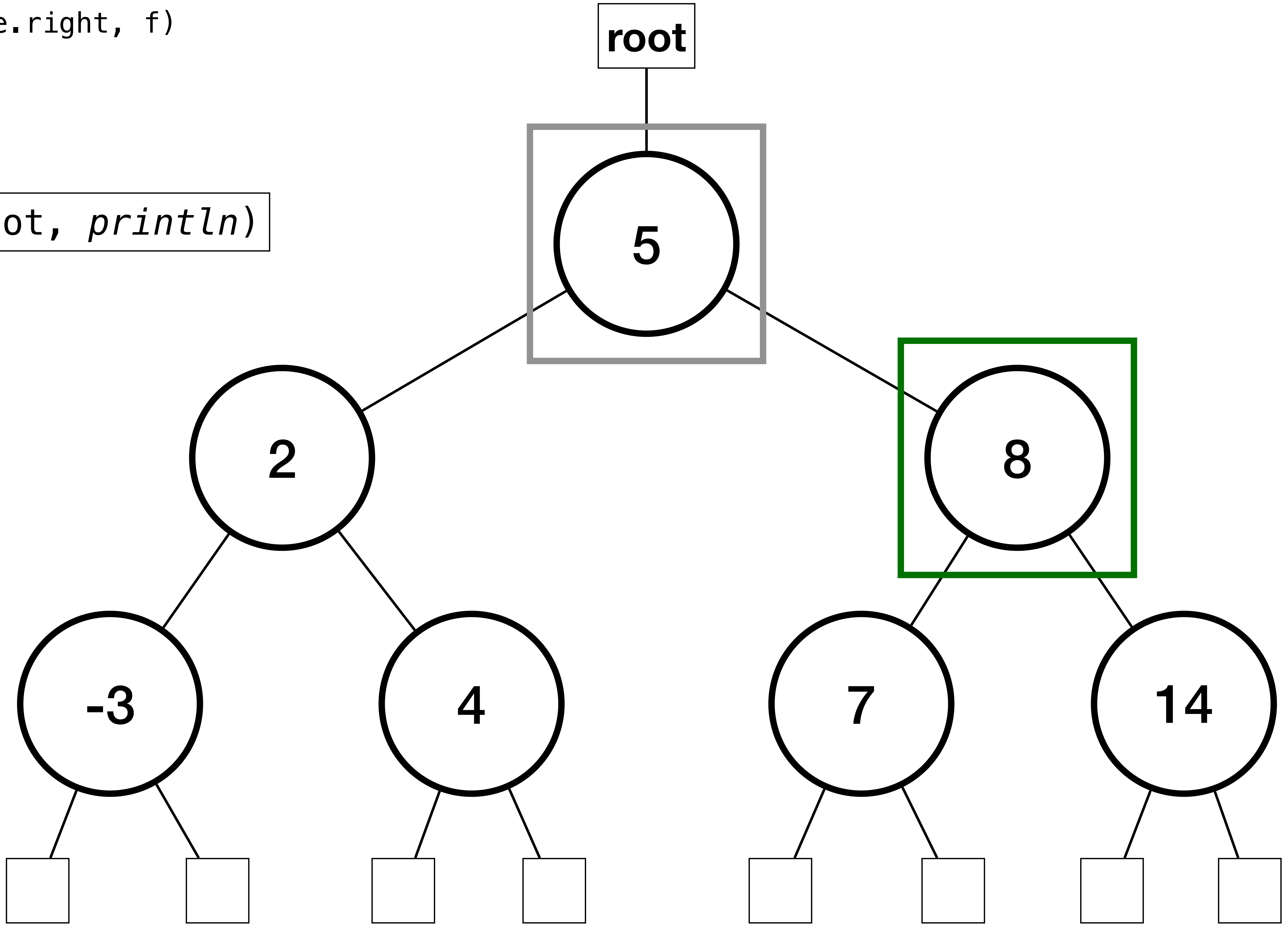


Traversals

```
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

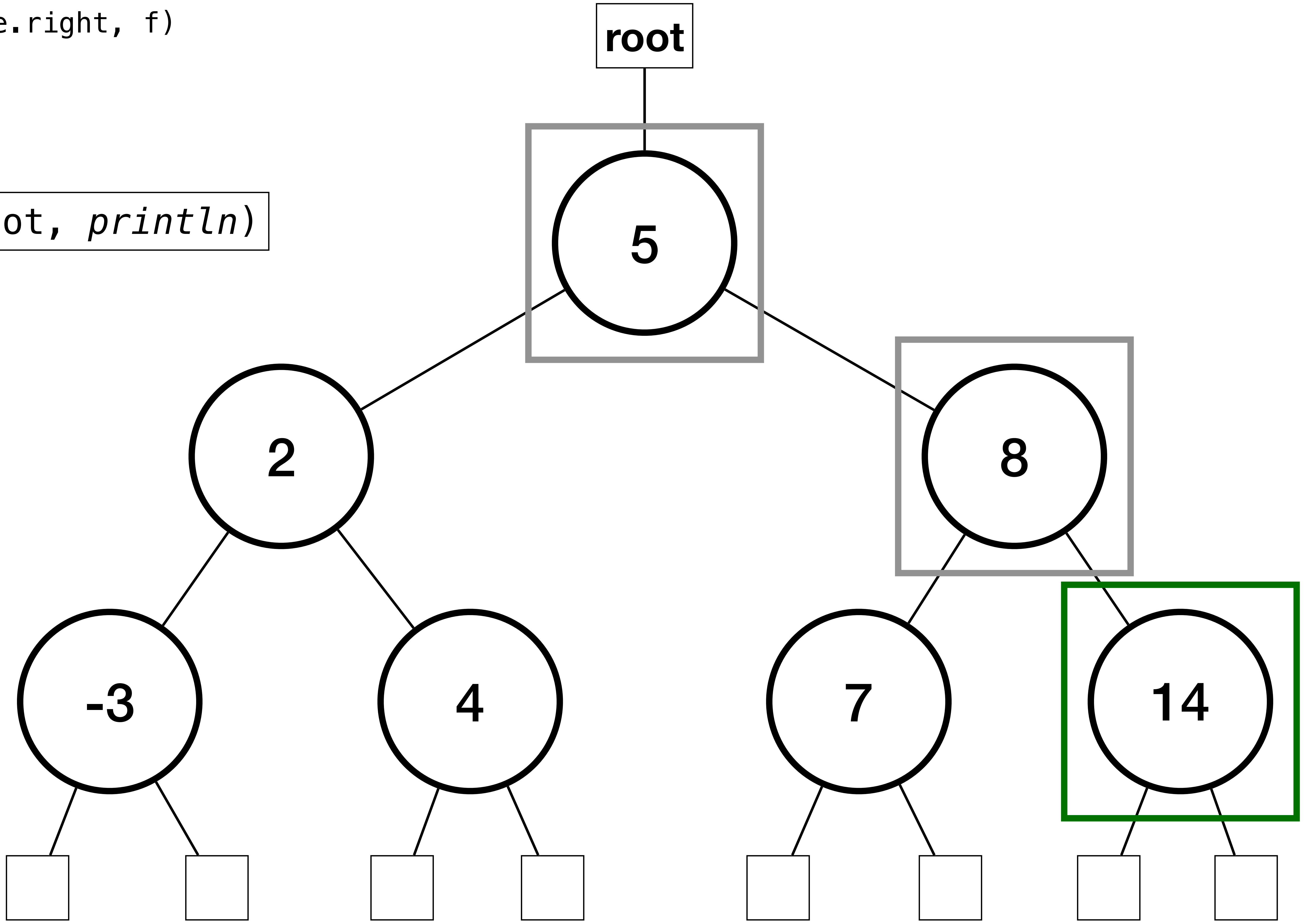


Traversals

```
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

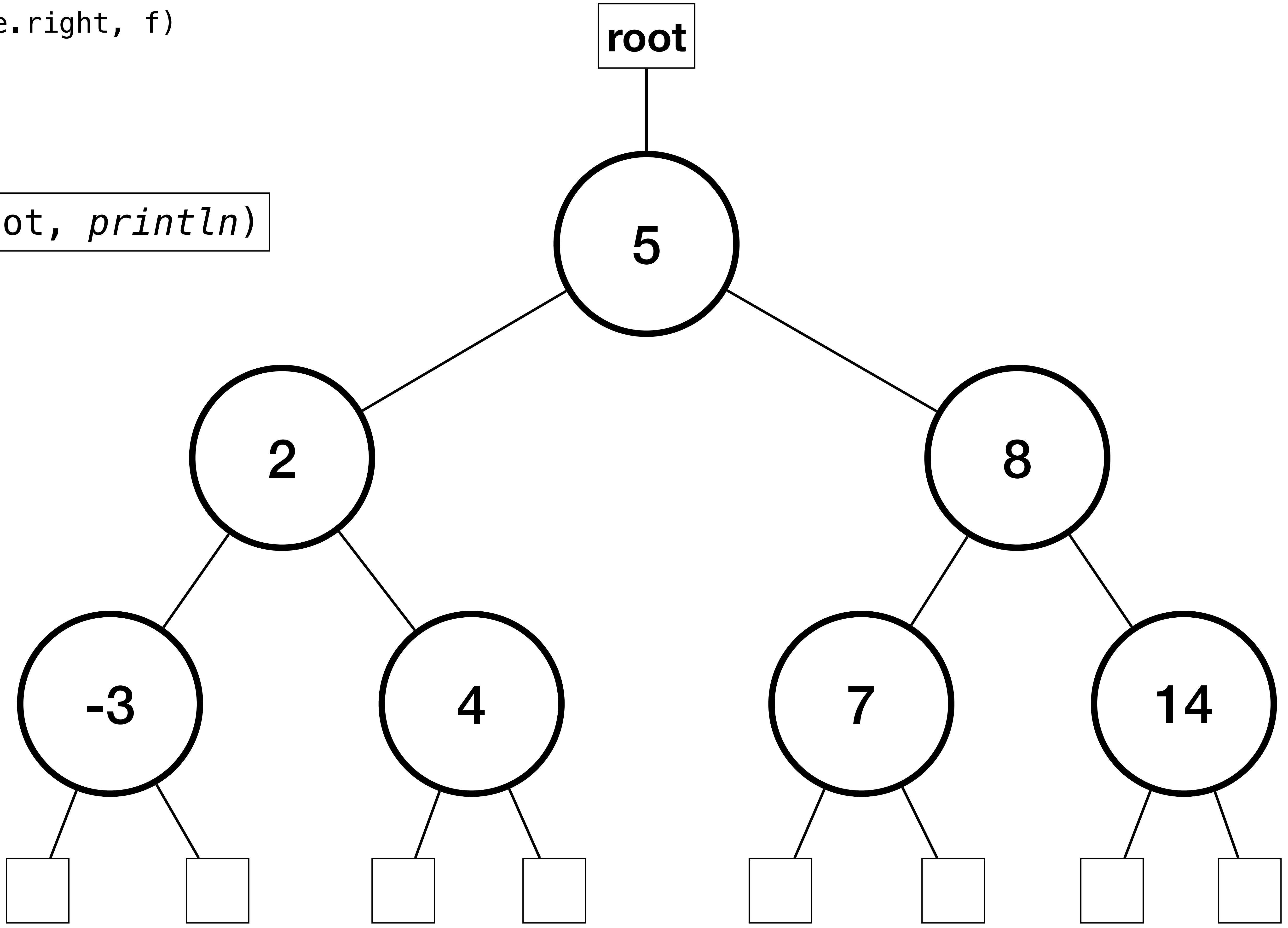


Traversals

```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)
```

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**

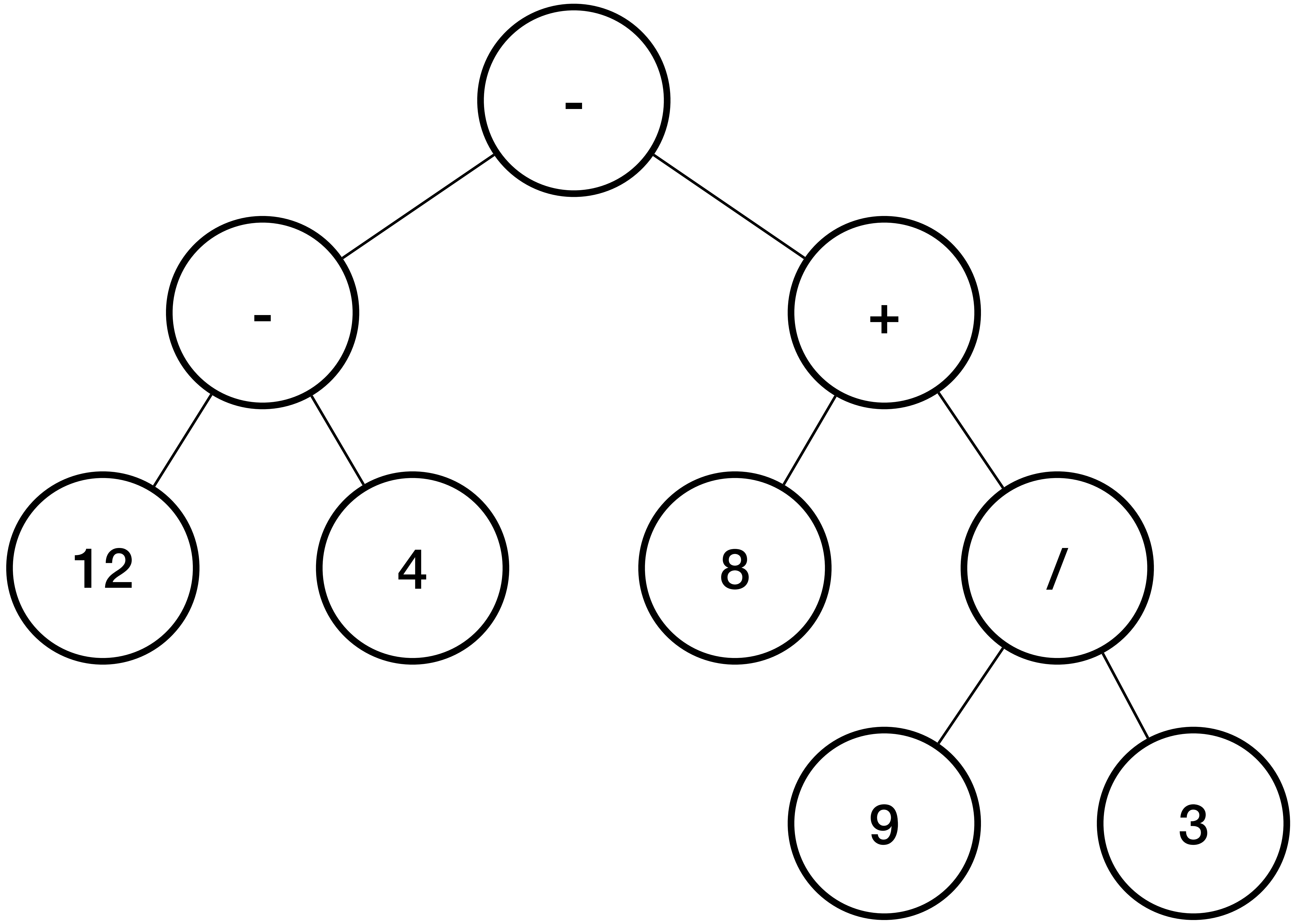
Expression Trees

Expression Trees

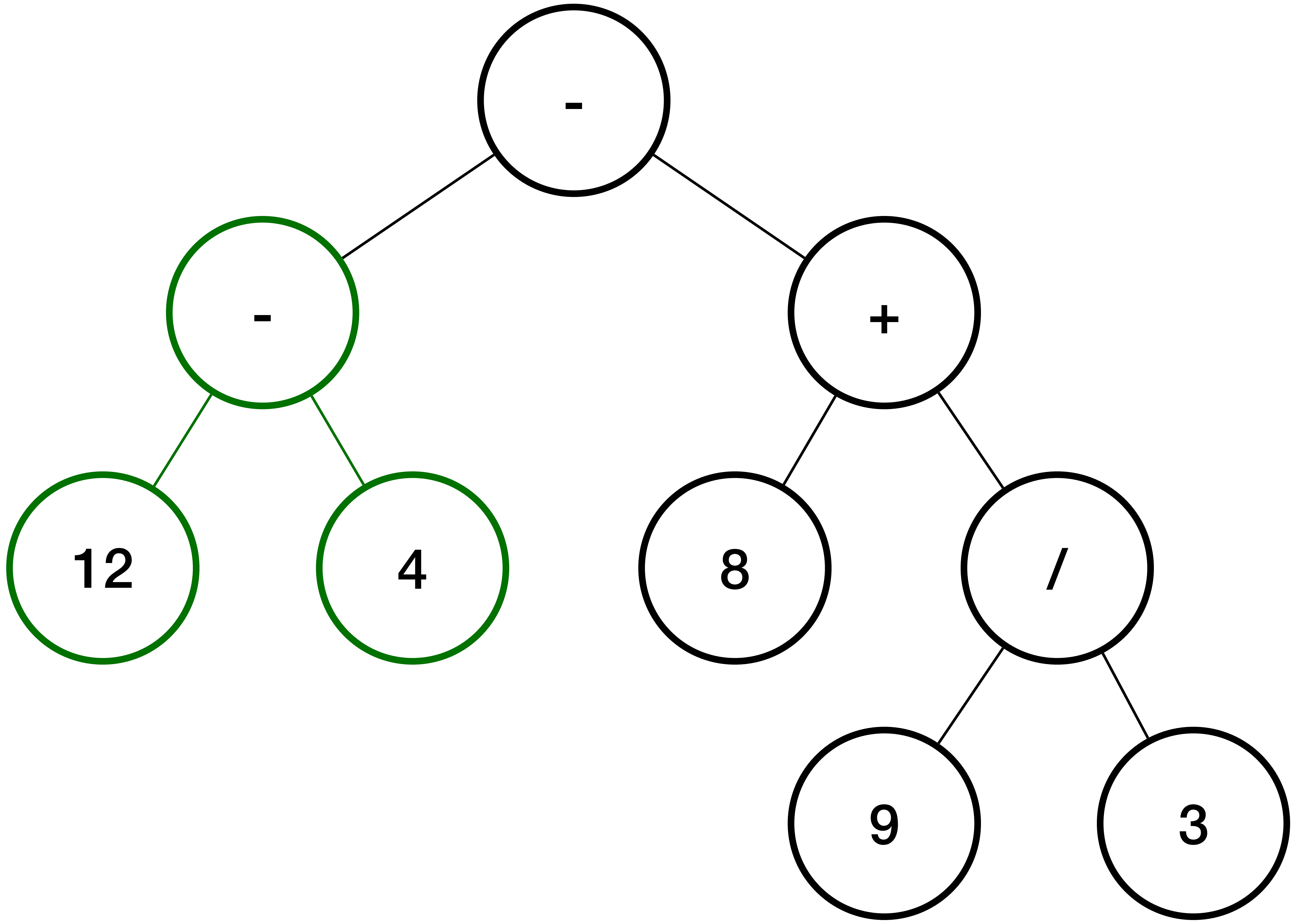
- Represent an expression as a binary tree
- Nodes can be
  - Operands
  - Operators
- An operand is a literal value
- An operator is evaluated by using its left and right children as operands
  - Operands can be operators

Expression Tree

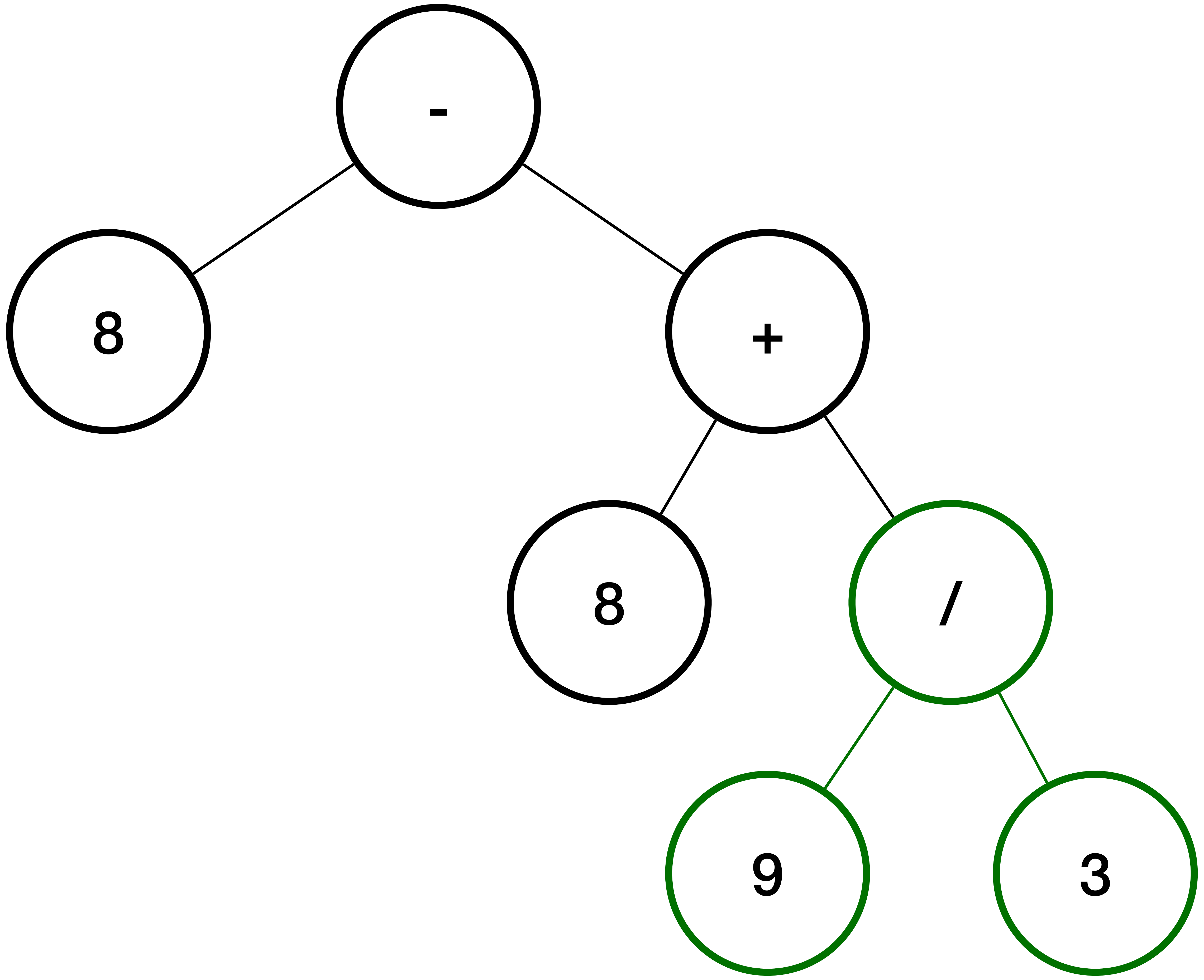
- (12-4) - (8+9/3) as an expression tree



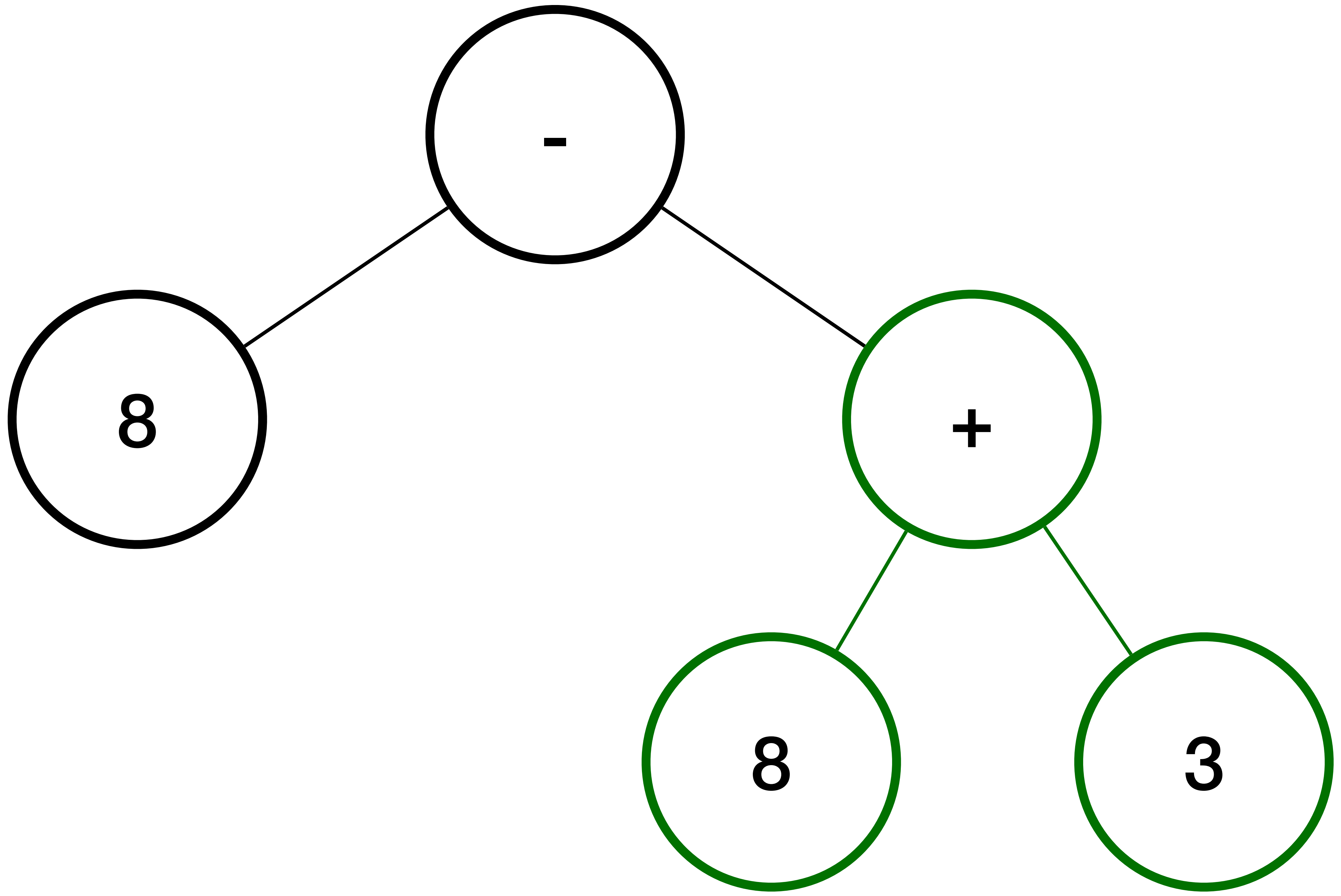
Evaluating Expression Tree



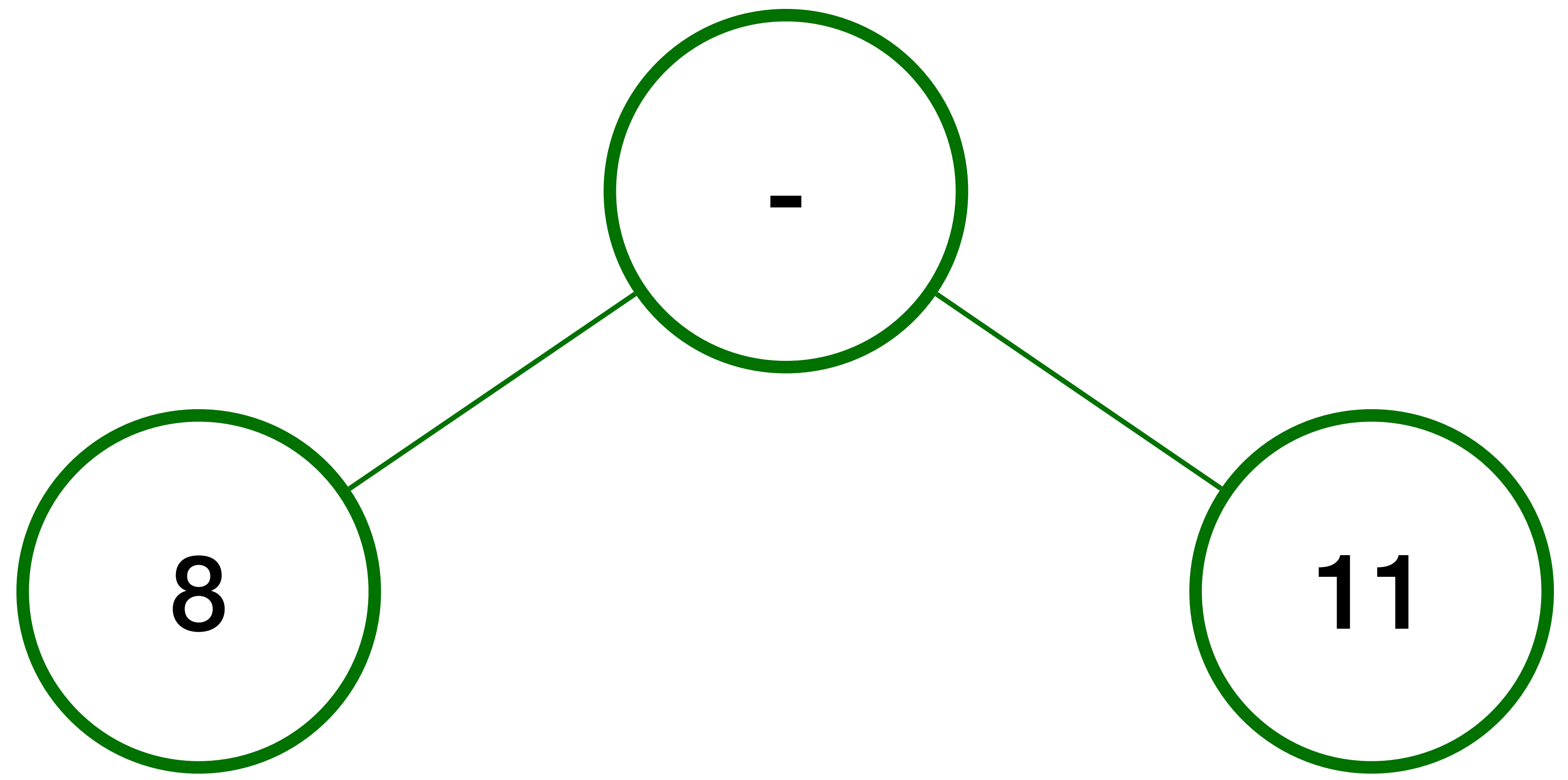
Evaluating Expression Tree



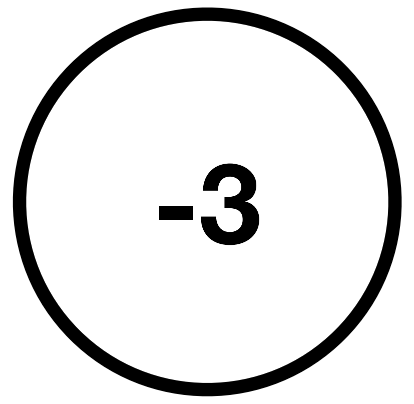
Evaluating Expression Tree



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(")")
    }
  }
}
```

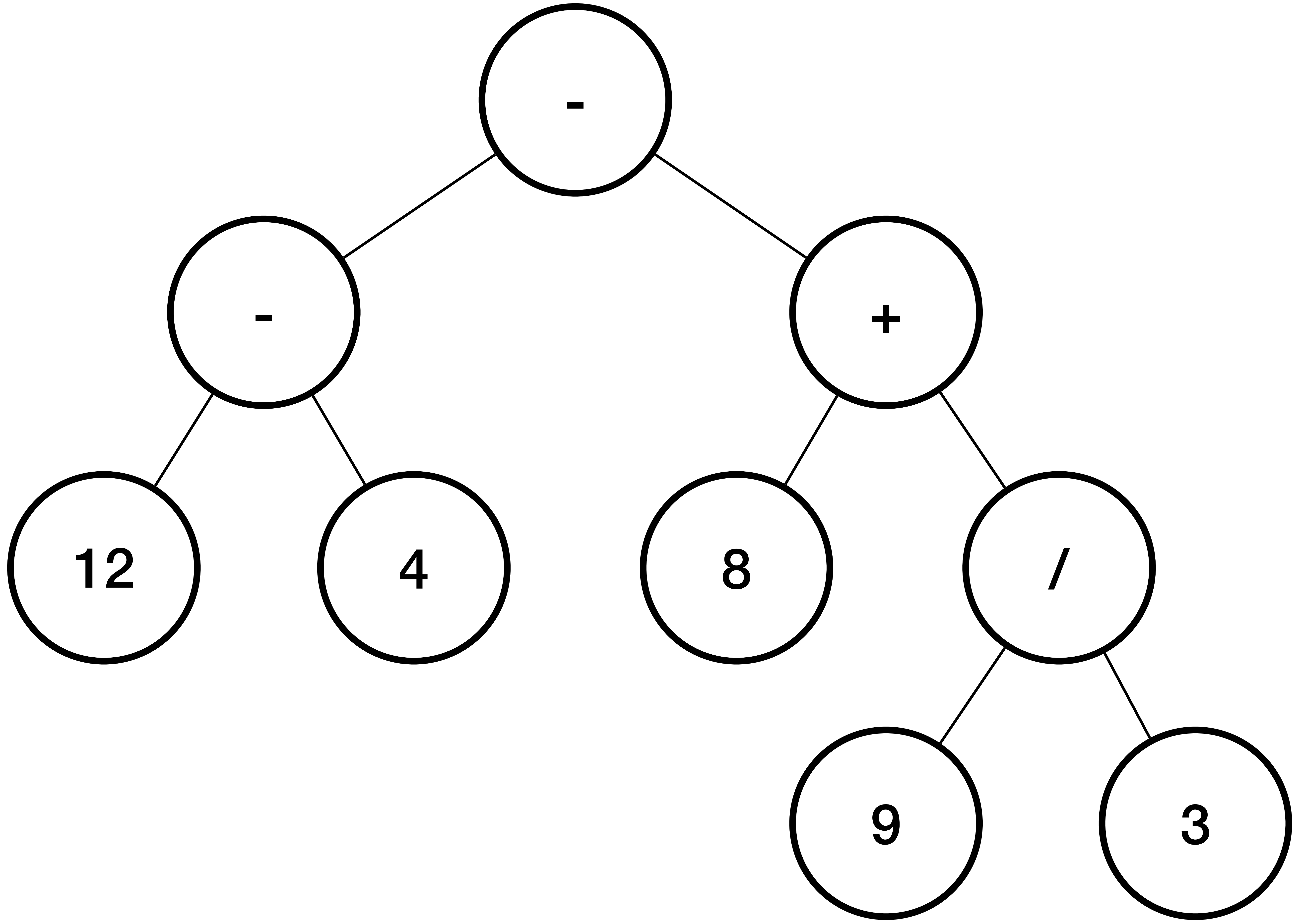
Expression Tree Traversals

- 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 Task

- Enemy AI: Lecture Task 3 -

Functionality: In the game.enemyai.AIPlayer class, implement the following method:

- A method named “computePath” with:
  - Two parameters of type GridLocation representing the start then end of the path to compute
  - Returns a path from the start to end locations as a LinkedListNode of GridLocations
    - Valid path connections only travel up, down, left, and right. Diagonal moves are not allowed
      - Ex. (1,3) -> (1,2) -> (2,2) is a valid path from (1,3) to (2,2)
    - The returned path must contain the minimal number of GridLoactions possible
      - For testing, keep in mind that there are many valid paths that can be returned. As long as the path is valid and has the minimal possible length, it should pass your tests

Testing: In the tests package, complete the test suite named LectureTask3 that tests this functionality.