CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Tree Traversal and Rotations

Announcements

- PA3 Tests due ~~Sunday~~ Monday
  - Remember, no grace days
  - Implementation still due next Sunday, START EARLY
  - Implementation Autograder will be up shortly

Collection ADTs

Property	Seq	Set	Bag
Explicit Order	✓		
Enforced Uniqueness		✓	
Iterable	✓	✓	✓

BST Operations

Operation	Runtime
`find`	$O(d)$
`insert`	$O(d)$
`remove`	$O(d)$

*What is the runtime in terms of **n**?*

BST Operations

Operation	Runtime
`find`	***O*(*d*)**
`insert`	***O*(*d*)**
`remove`	***O*(*d*)**

*What is the runtime in terms of **n**? **O(n)***

BST Operations

Operation	Runtime
`find`	$O(d)$
`insert`	$O(d)$
`remove`	$O(d)$

*What is the runtime in terms of **n**?* $\boldsymbol{O(n)}$

Does it need to be that bad?

BST Operations

Operation	Runtime
`find`	$O(d)$
`insert`	$O(d)$
`remove`	$O(d)$

What is the runtime in terms of ***n****?* $\boldsymbol{O(n)}$

Does it need to be that bad? …hold that thought

Collection ADTs

Property	Seq	Set	Bag
Explicit Order	✓		
Enforced Uniqueness		✓	
$\boxed{\text{Iterable}}$	✓	✓	✓

Tree Traversals

Goal: Visit every element of a tree (in linear time?)

Pre-Order (top-down)

Visit the `root`, then the `left` subtree, then the `right` subtree

In-Order

Visit the `left` subtree, then the `root`, then the `right` subtree

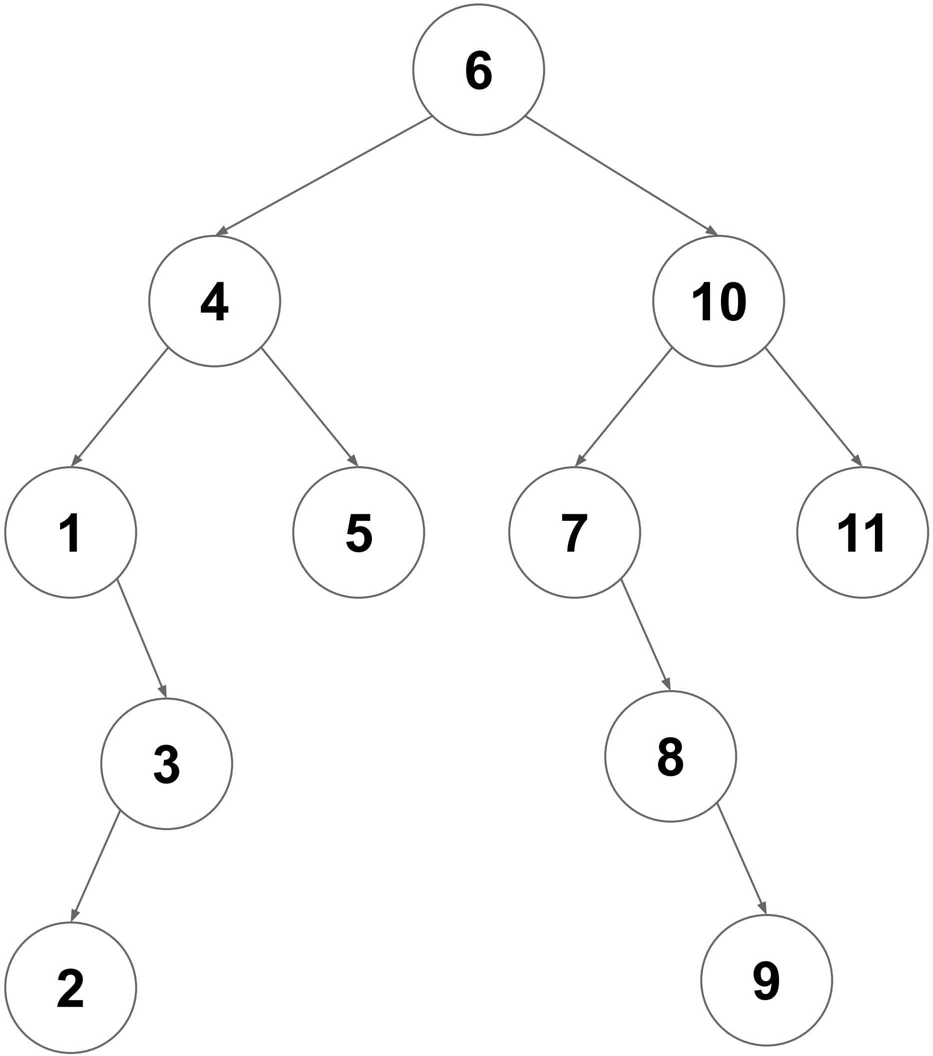
Post-Order (bottom-up)

Visit the `left` subtree, then the `right` subtree, then the `root`

Tree Traversal: In-Order

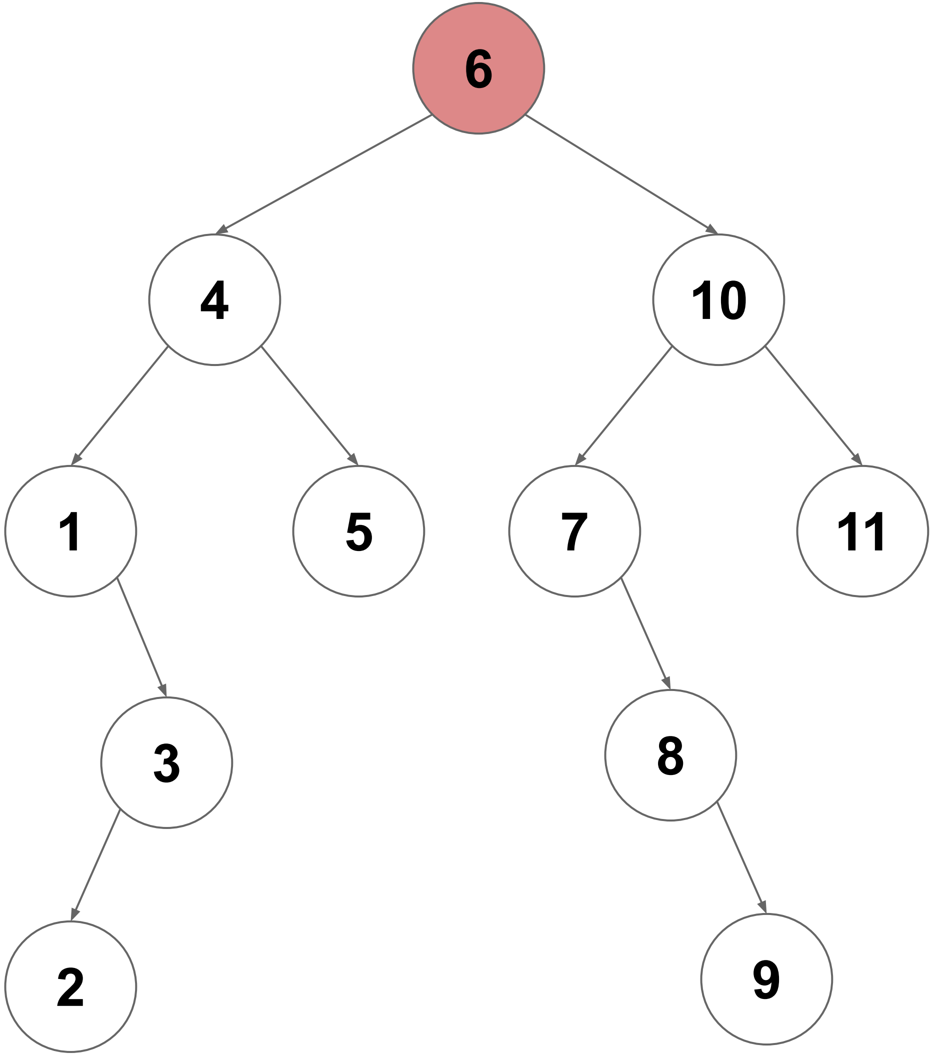
```
def inorderVisit[T](root: ImmutableTree[T], visit: ImmutableTree[T] => Unit) = {
  root match {
    case TreeNode(v, left, right) =>
      /* visit left */
      inorderVisit(left, visit)
      /* visit root */
      visit(v)
      /* visit right */
      inorderVisit(right, visit)

    case EmptyTree =>
      /* Do Nothing */
  }
}
```

In-Order Traversal on a BST



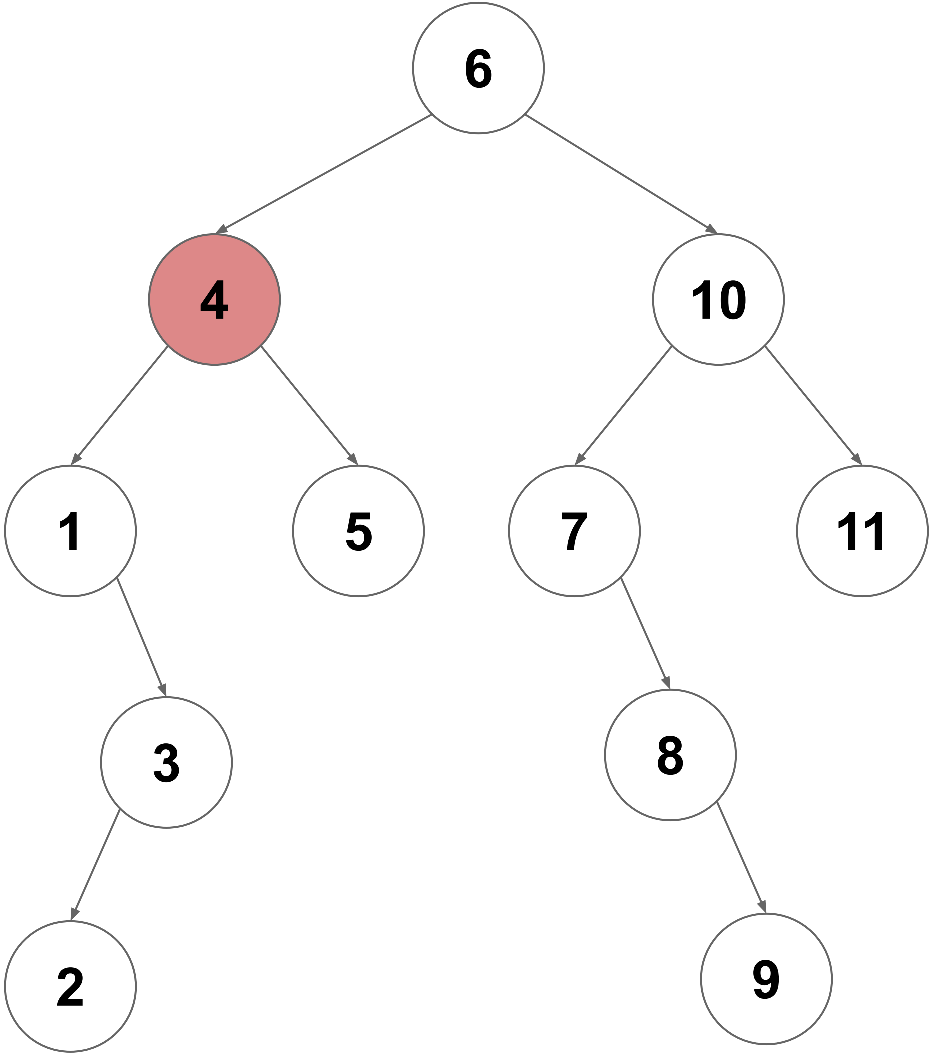
In-Order Traversal on a BST

`inorderVisit(6)`



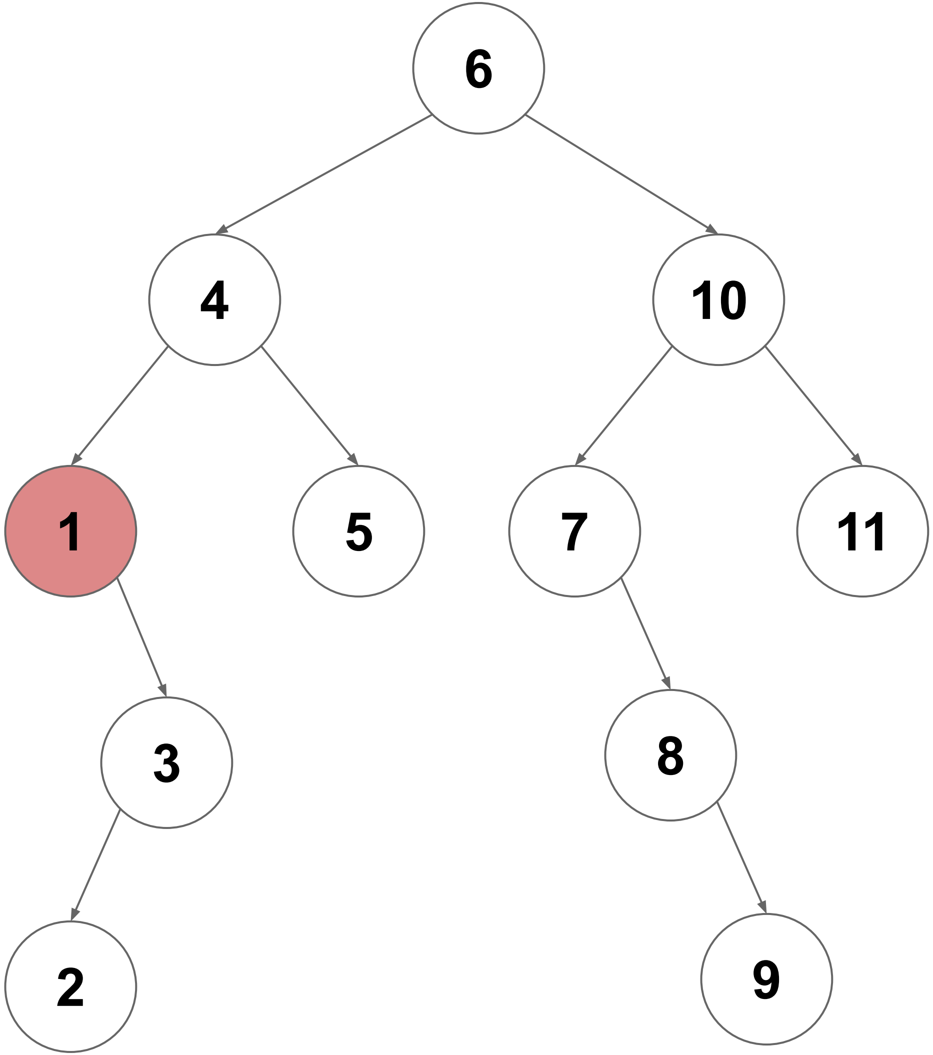
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(4)
```



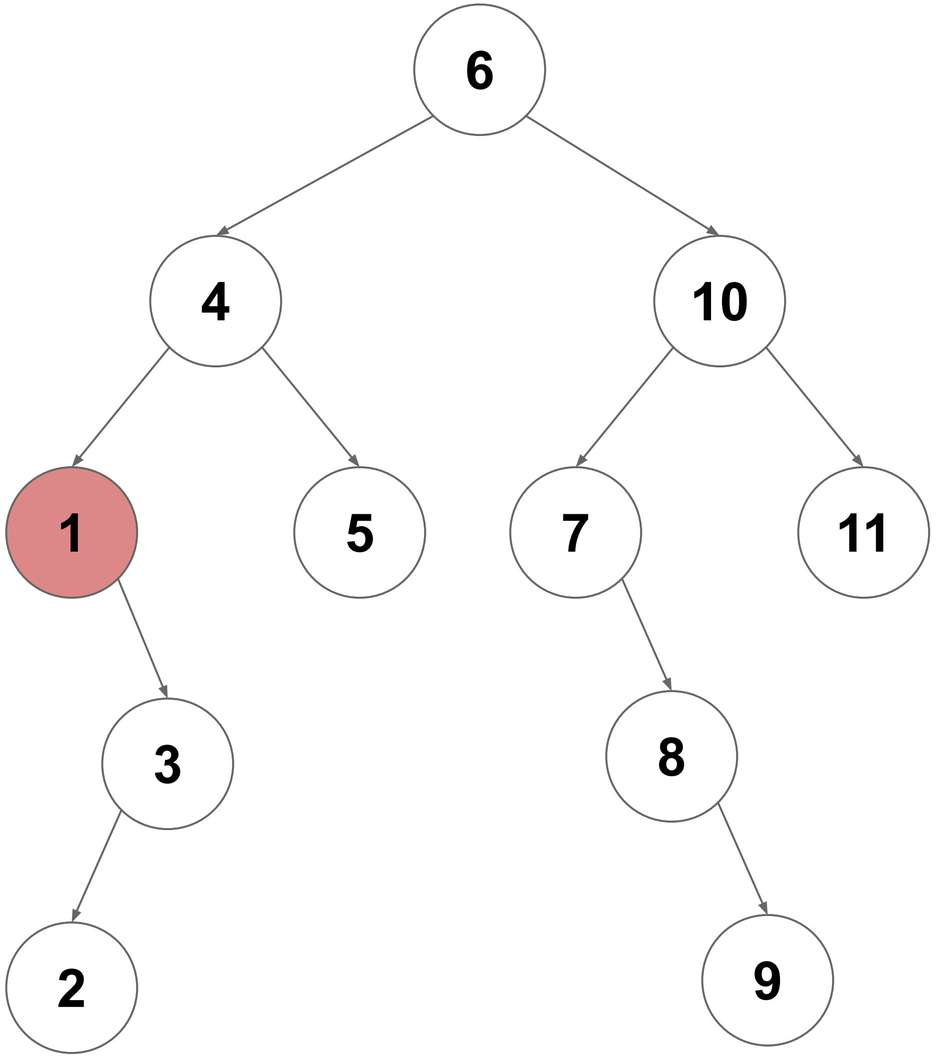
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(4)

inorderVisit(1)
```



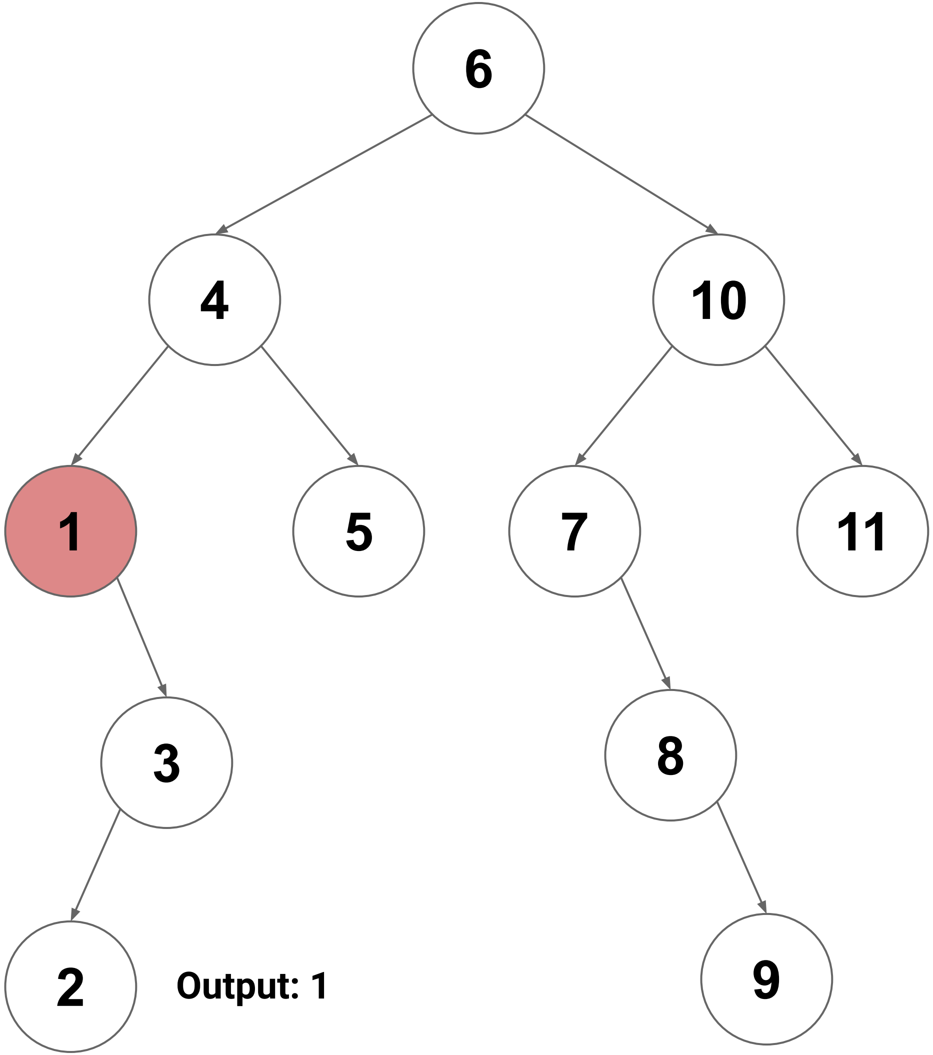
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(4)

inorderVisit(1)

inorderVisit(empty)
```



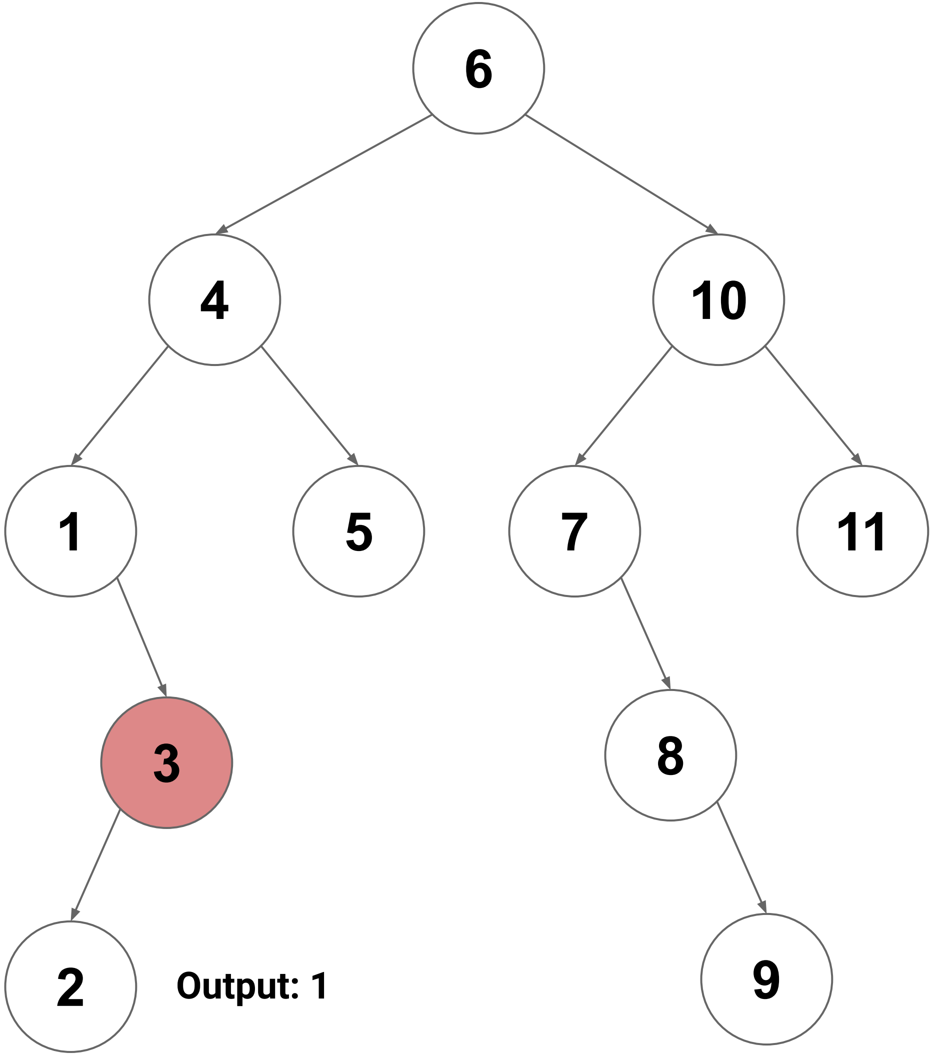
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(4)

inorderVisit(1)

visit(1)
```



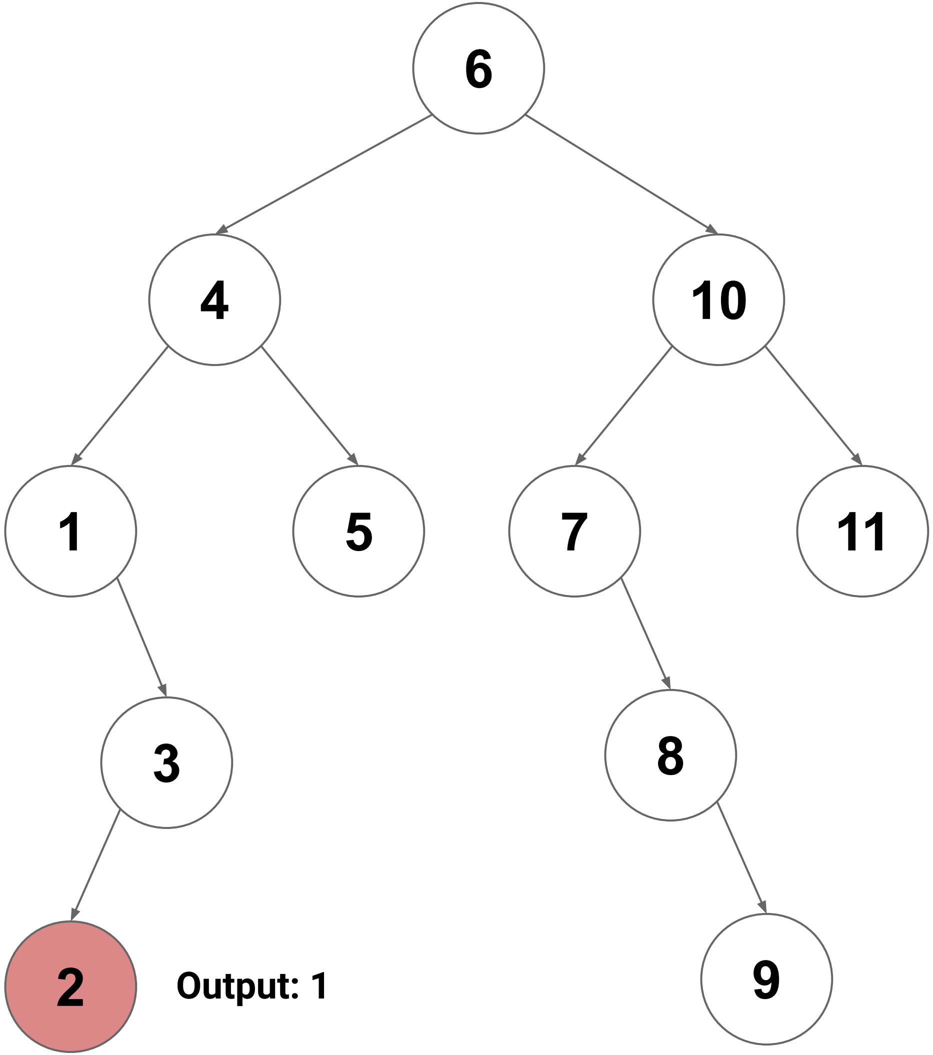
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(4)

inorderVisit(1)

inorderVisit(3)
```



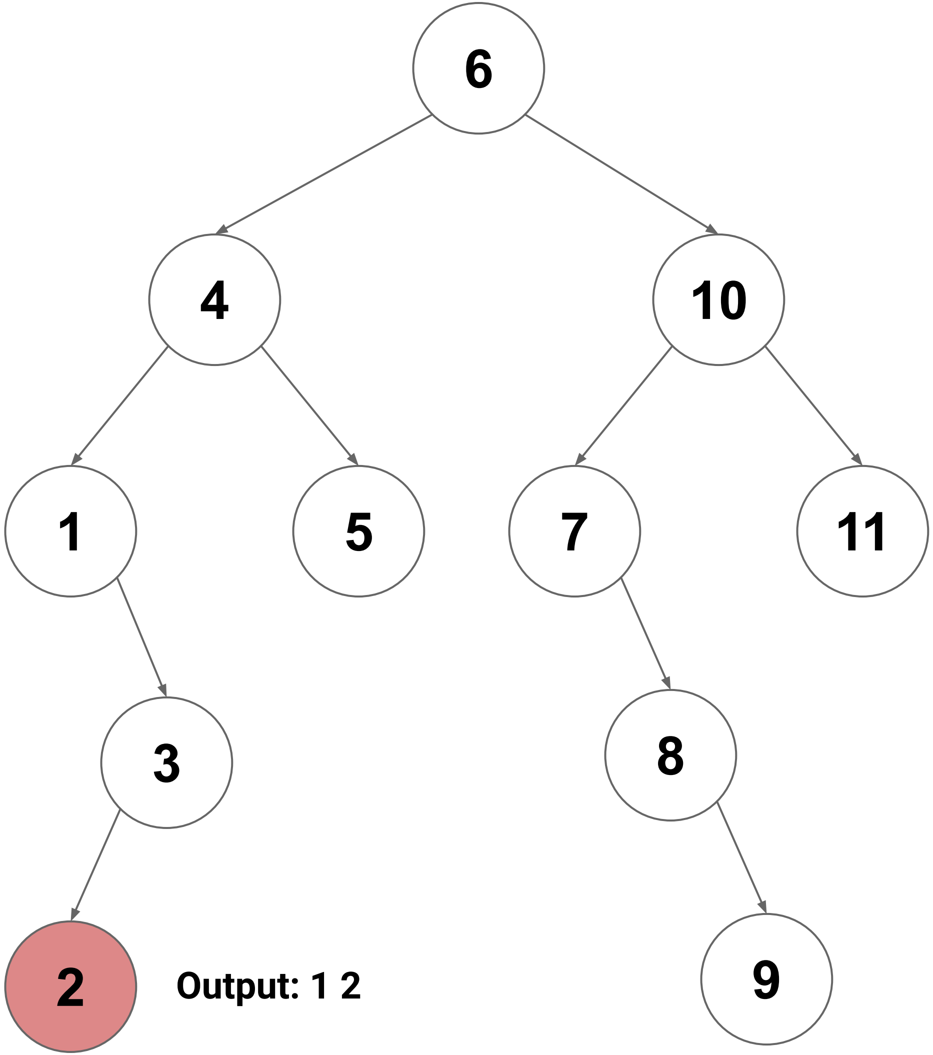
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(4)

inorderVisit(1)

inorderVisit(3)

inorderVisit(2)
```



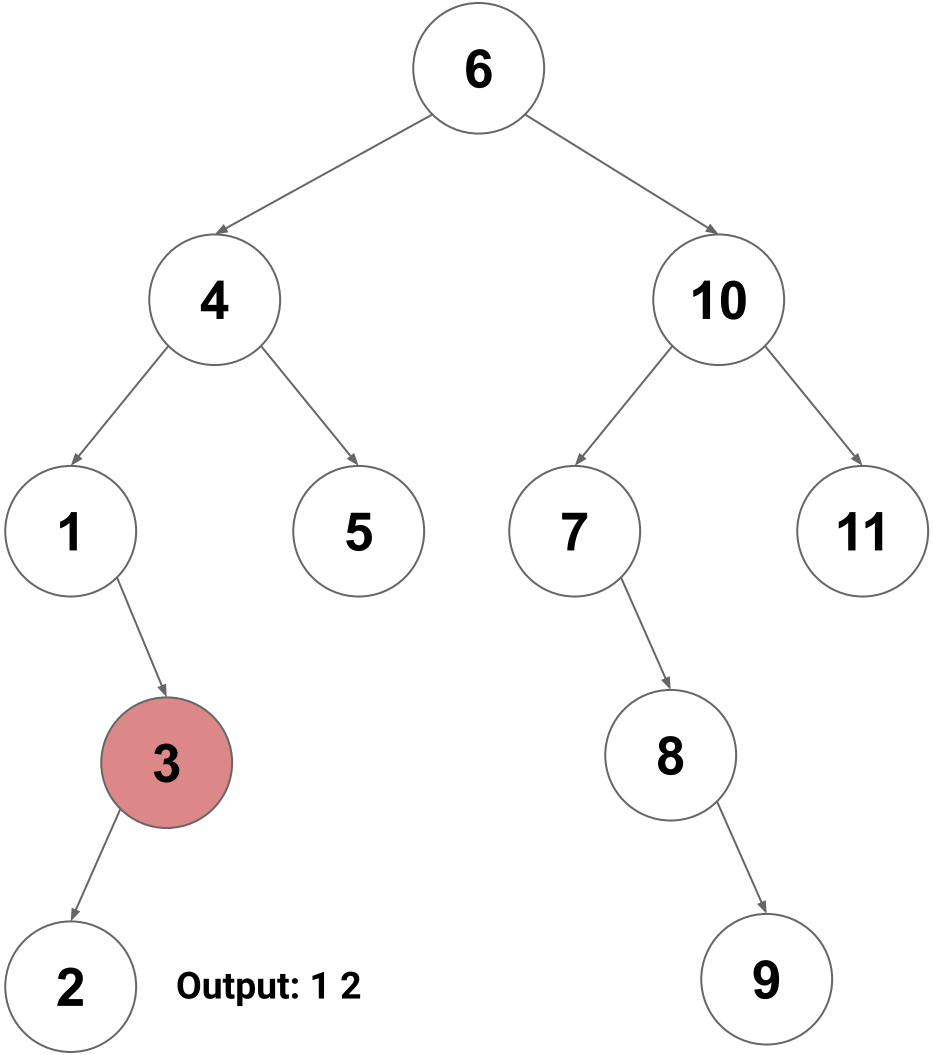
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(4)

inorderVisit(1)

inorderVisit(3)

visit(2)
```



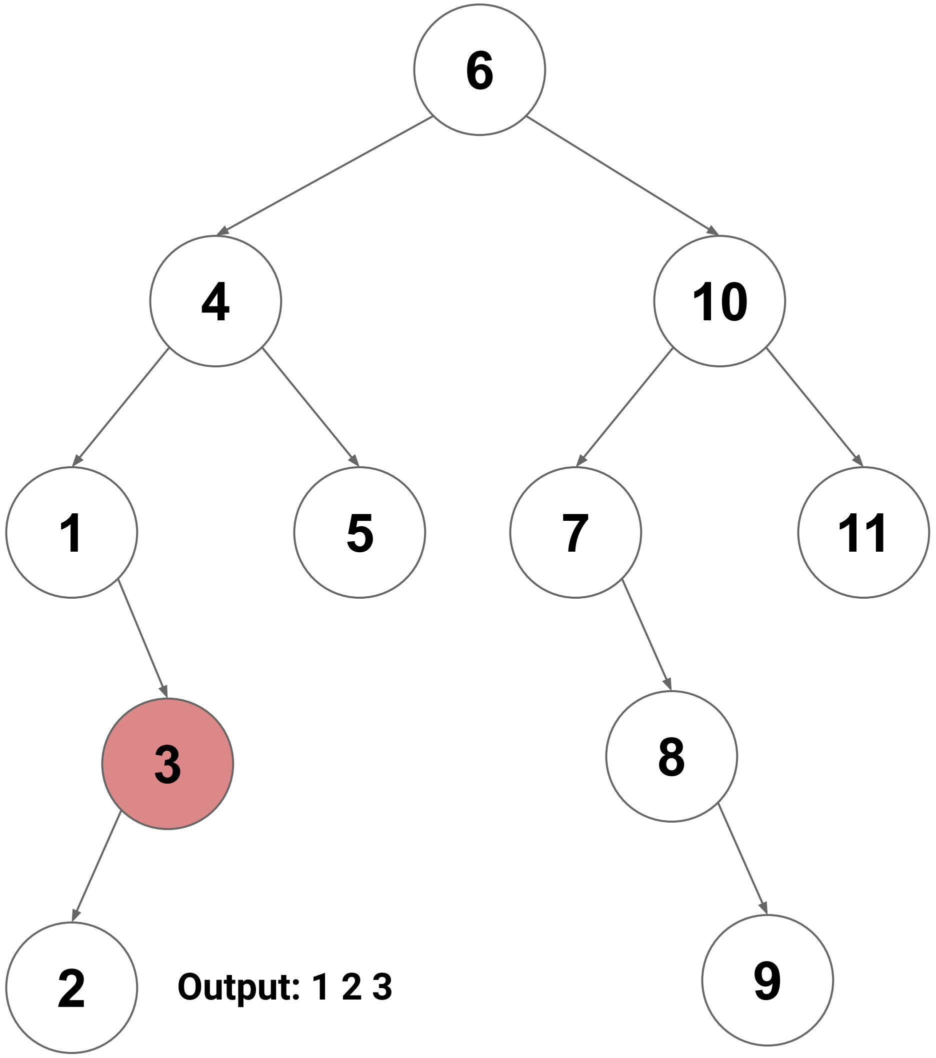
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(4)

inorderVisit(1)

inorderVisit(3)
```



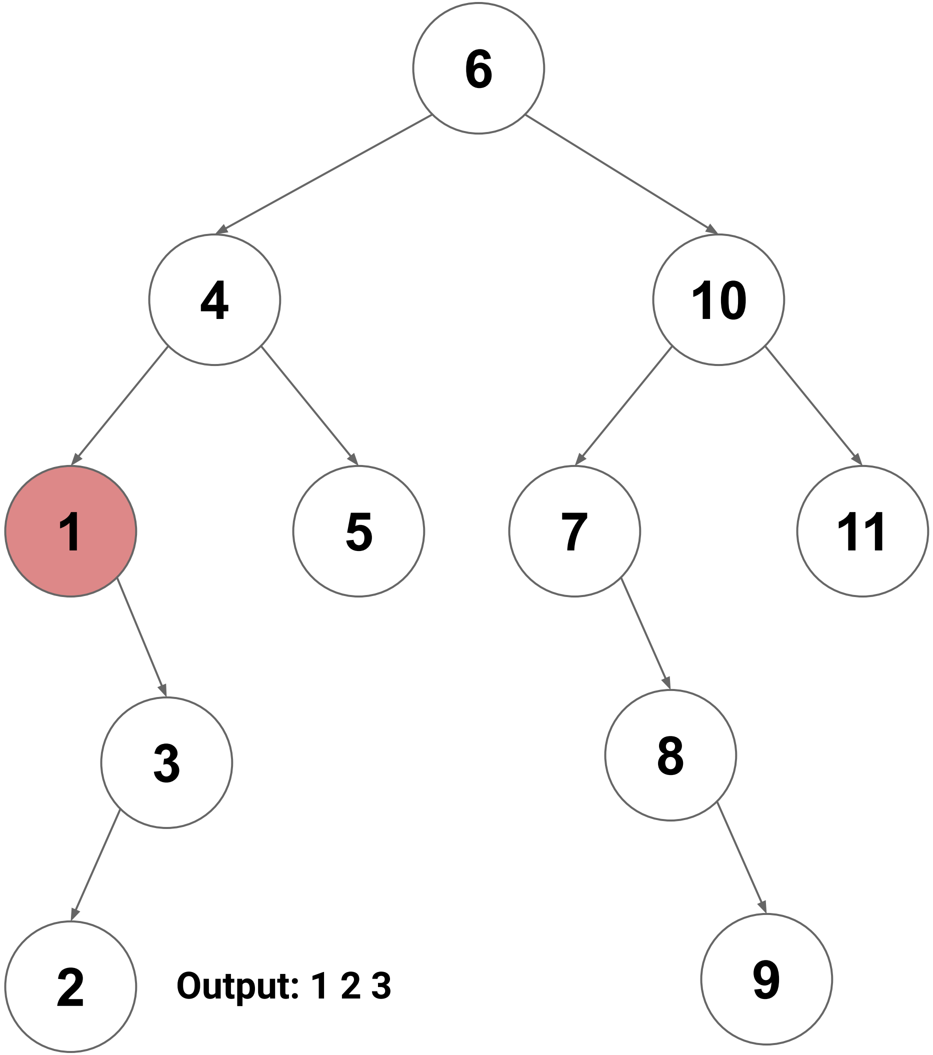
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(4)

inorderVisit(1)

visit(3)
```



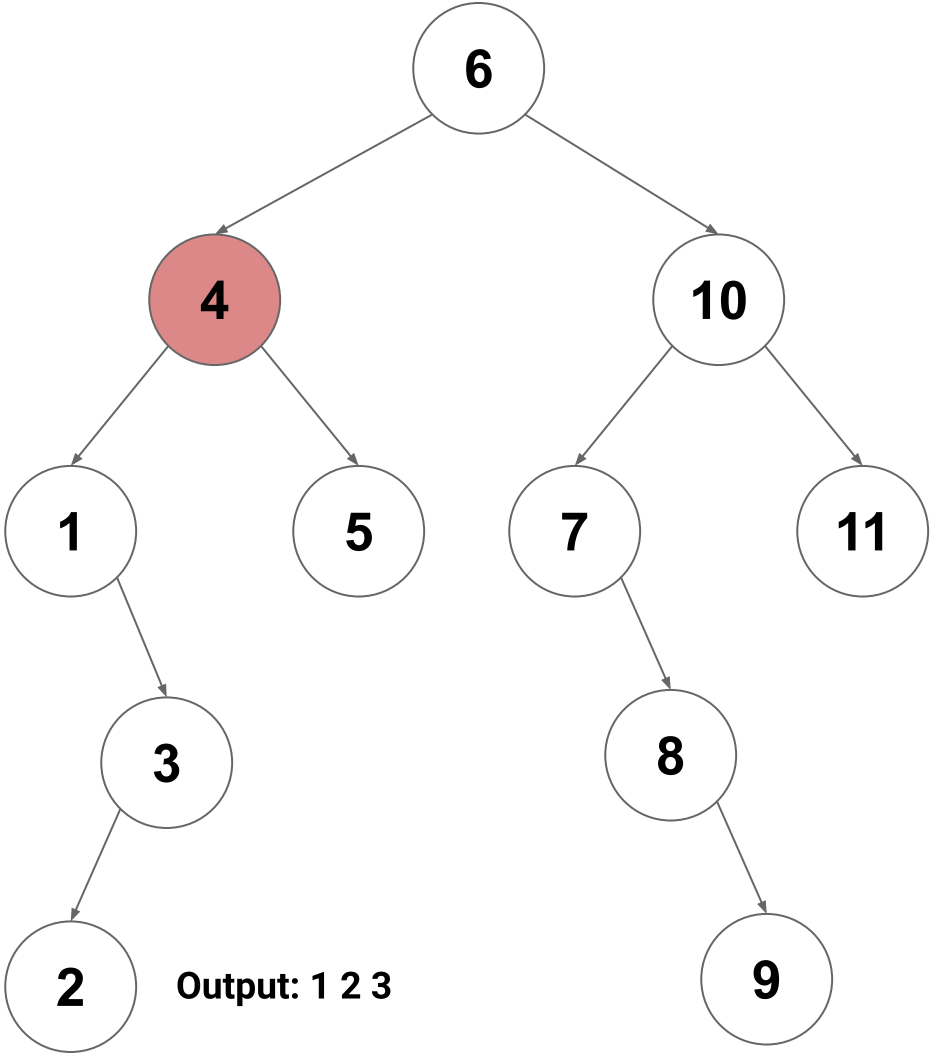
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(4)

inorderVisit(1)
```



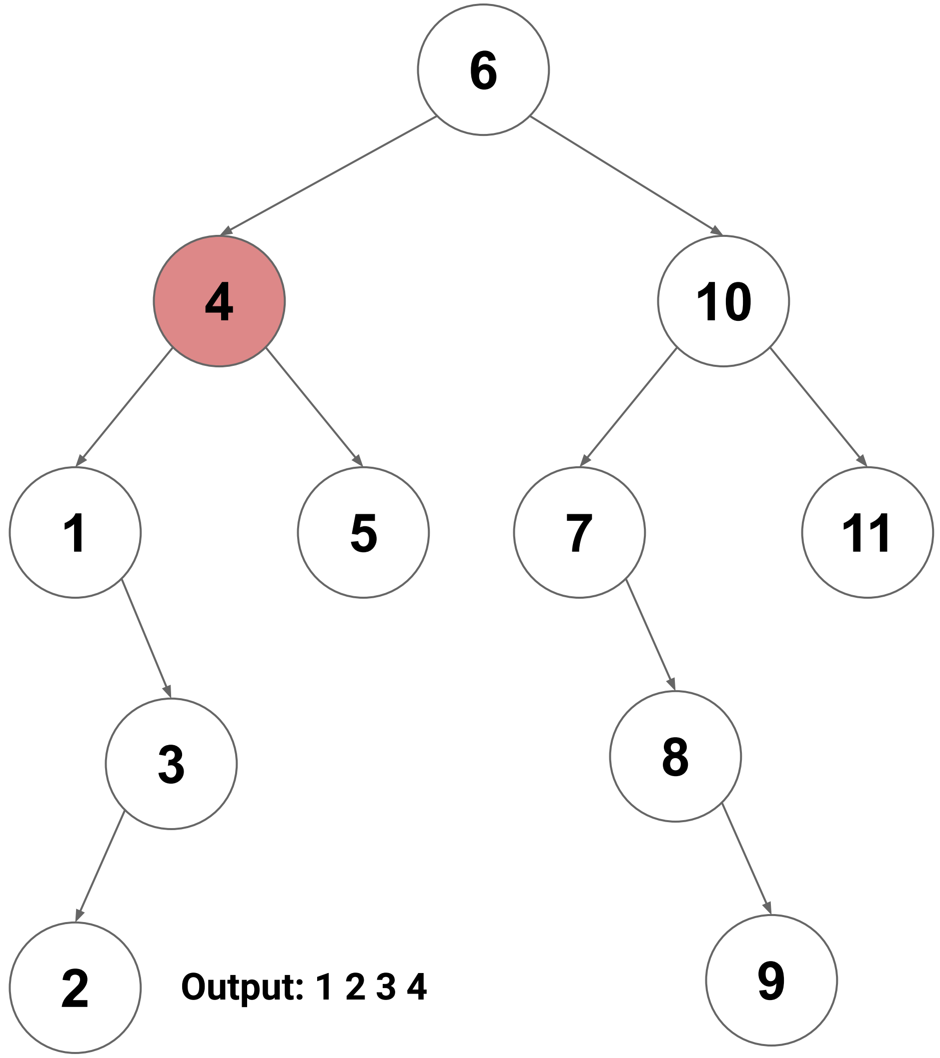
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(4)
```



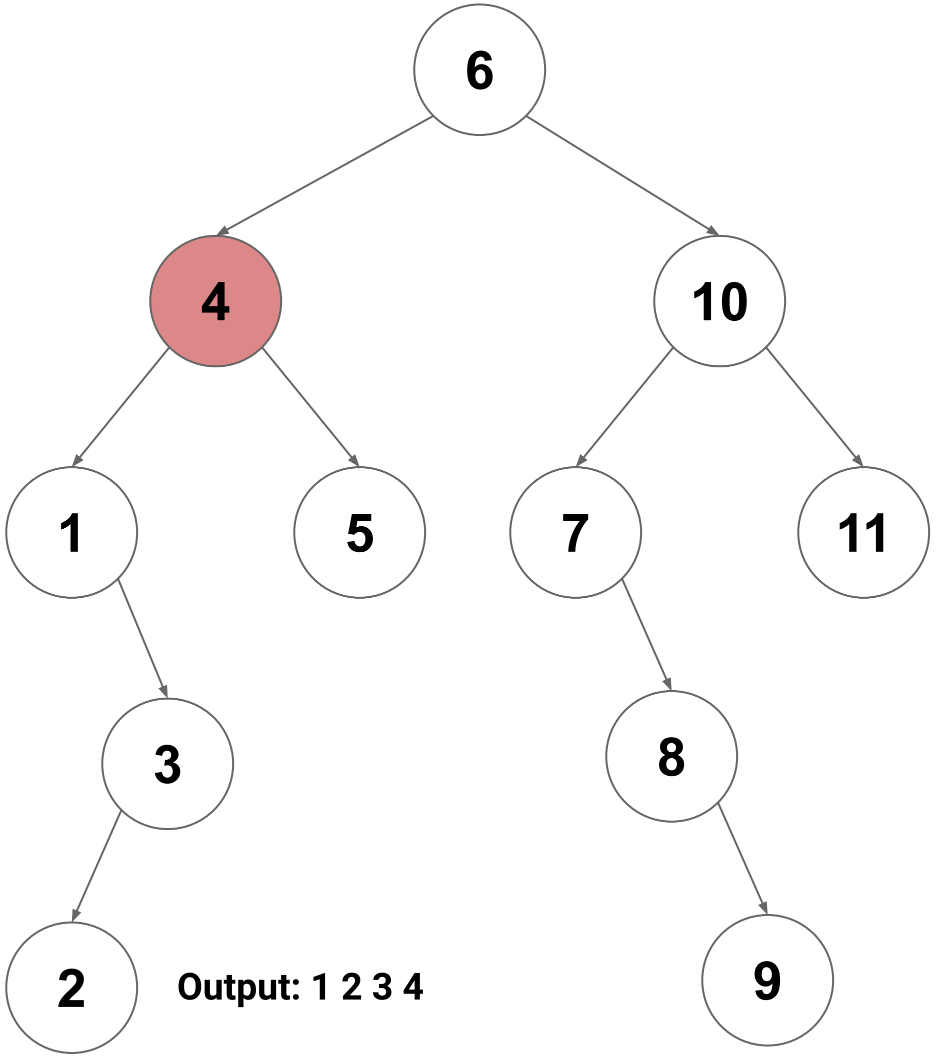
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(4)

visit(4)
```



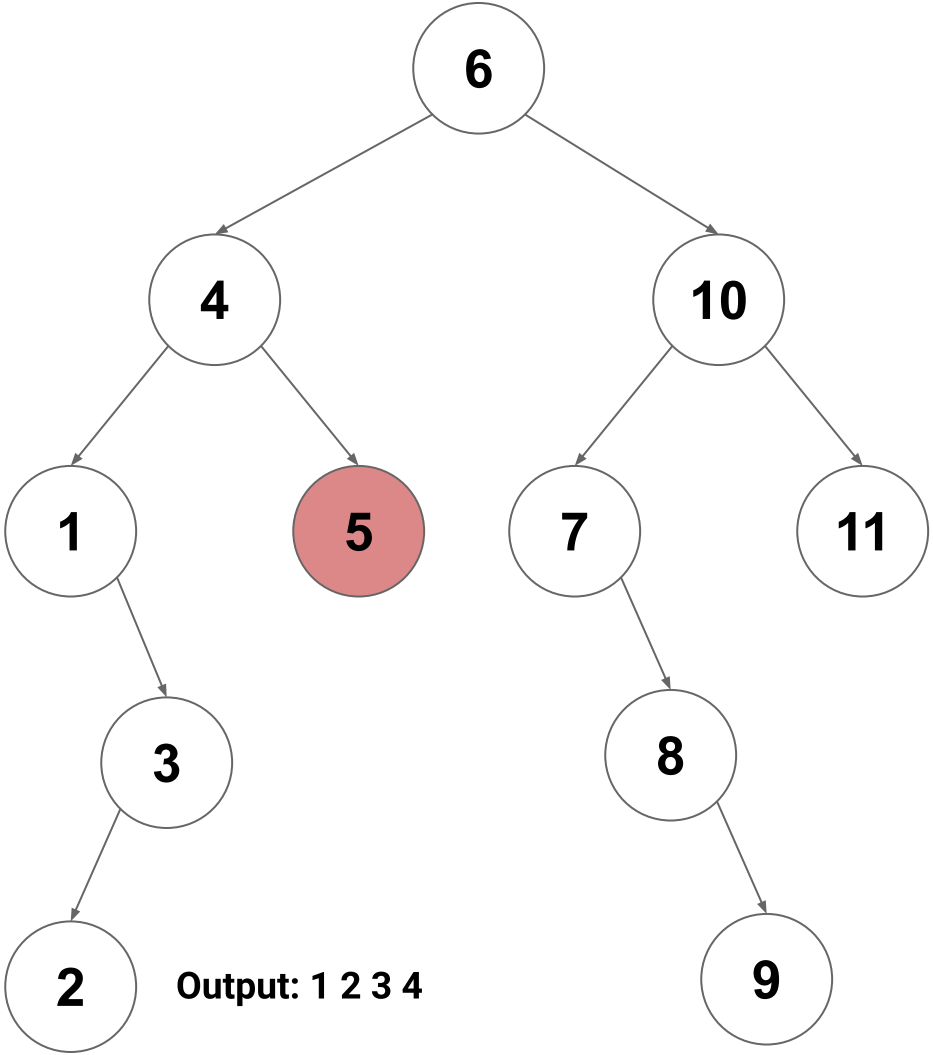
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(4)
```



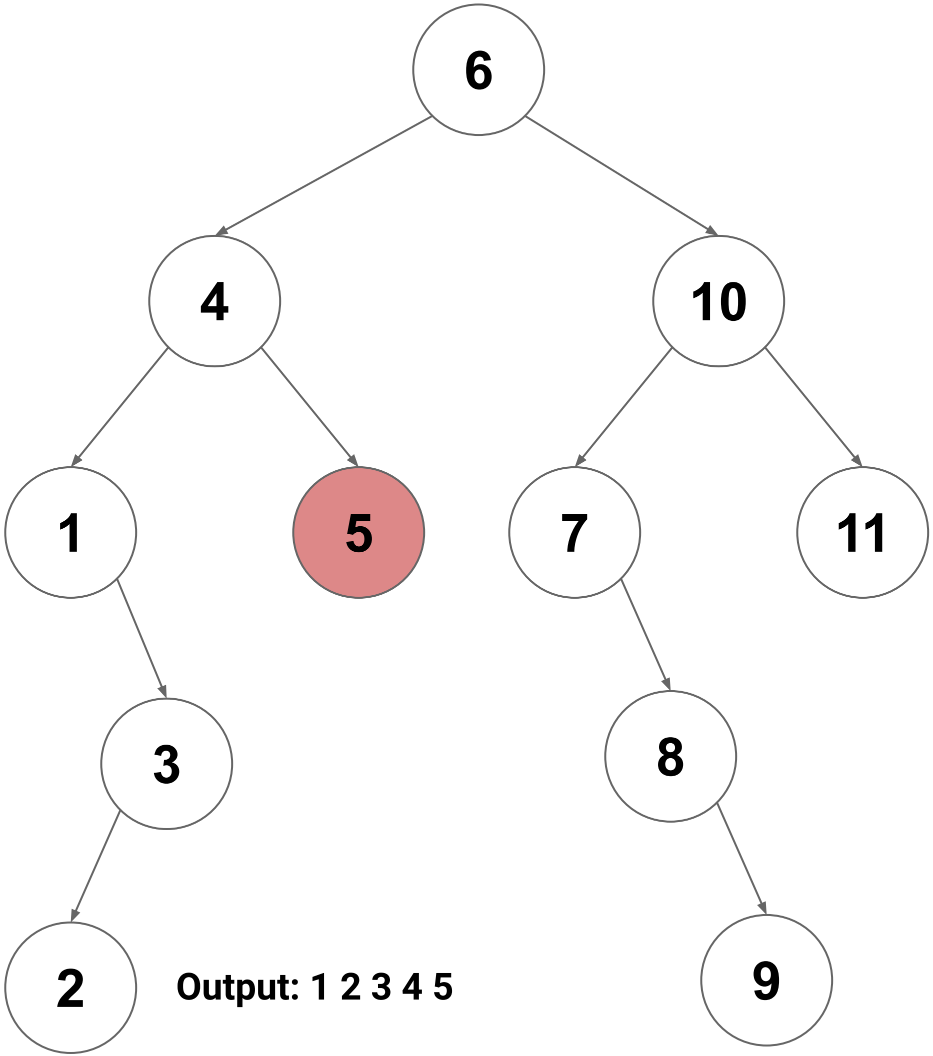
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(4)

inorderVisit(5)
```



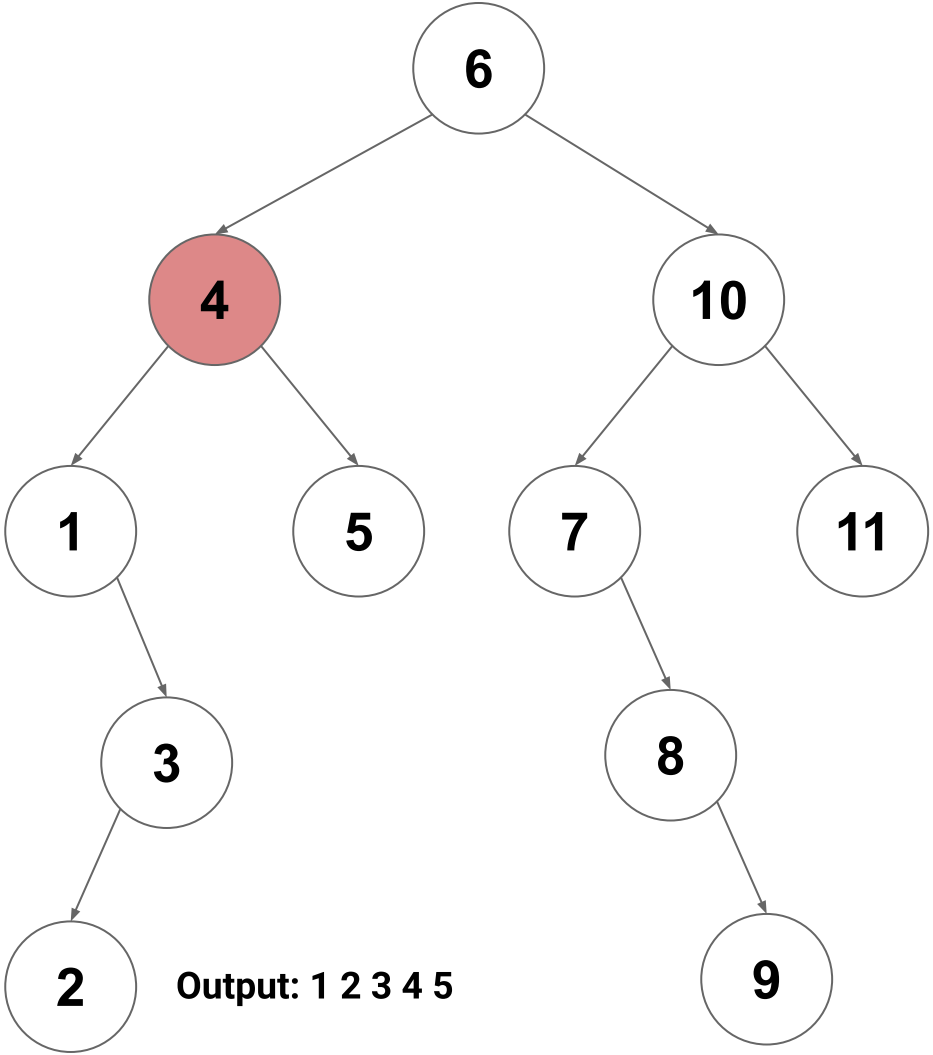
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(4)

visit(5)
```



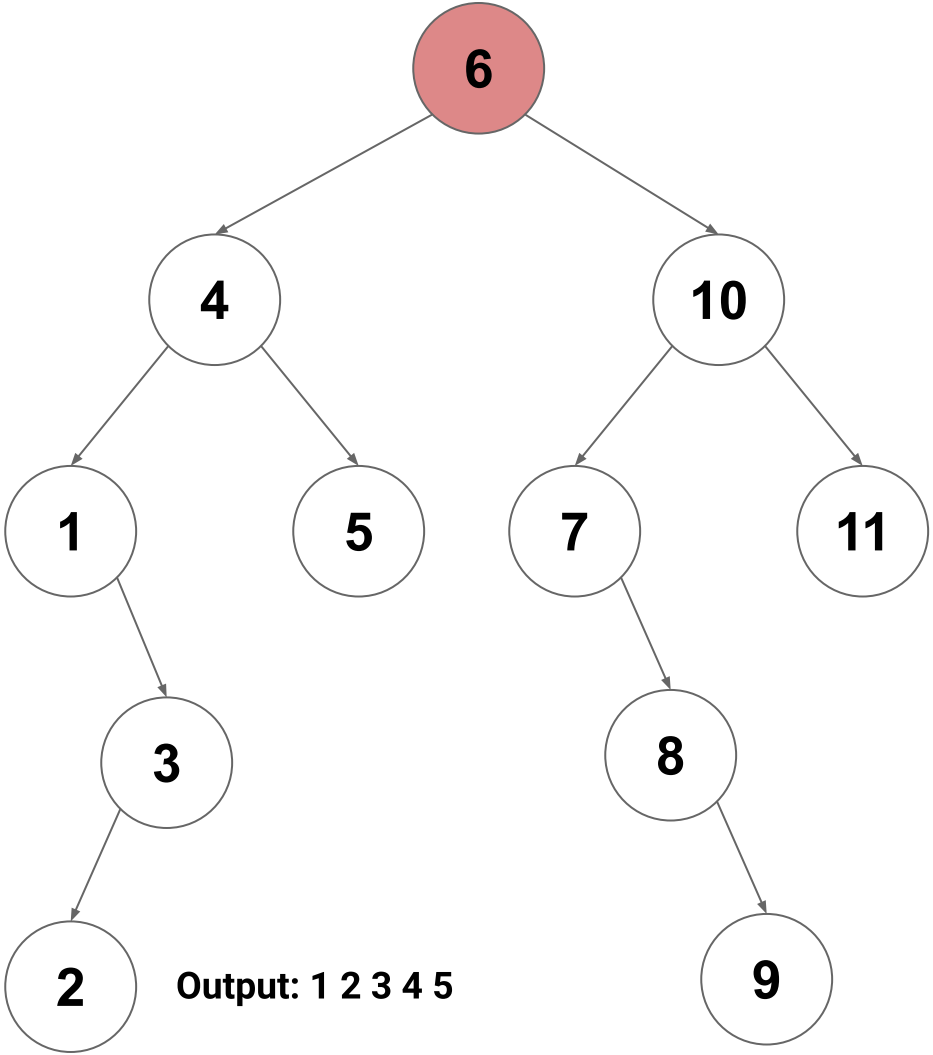
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(4)
```



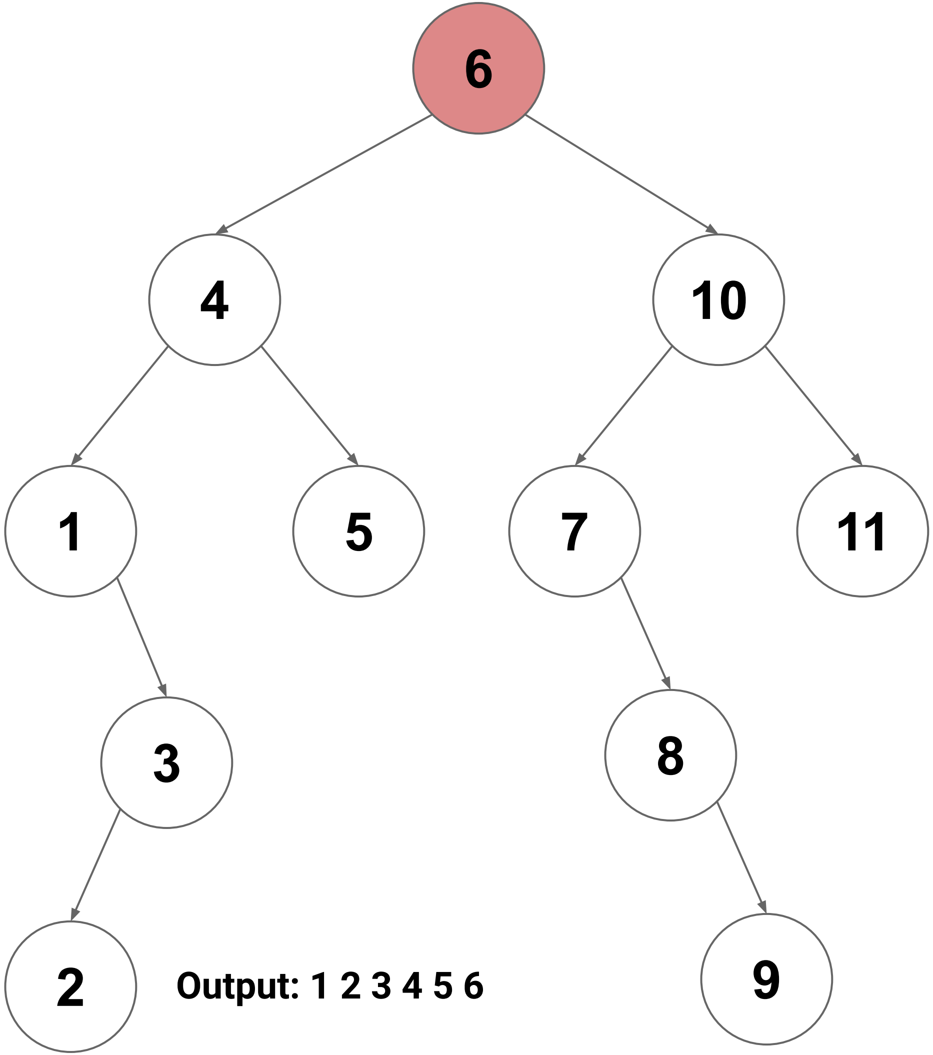
In-Order Traversal on a BST

`inorderVisit(6)`



In-Order Traversal on a BST

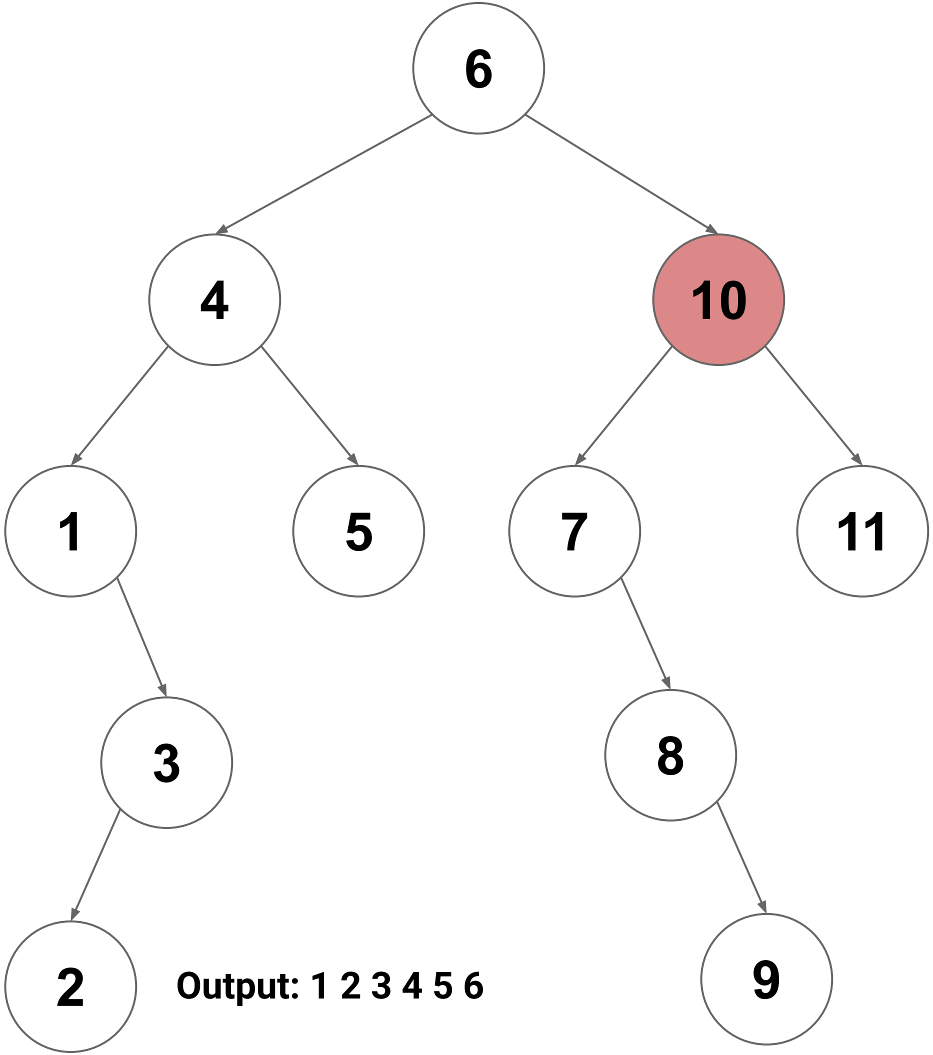
```
inorderVisit(6)

visit(6)
```



In-Order Traversal on a BST

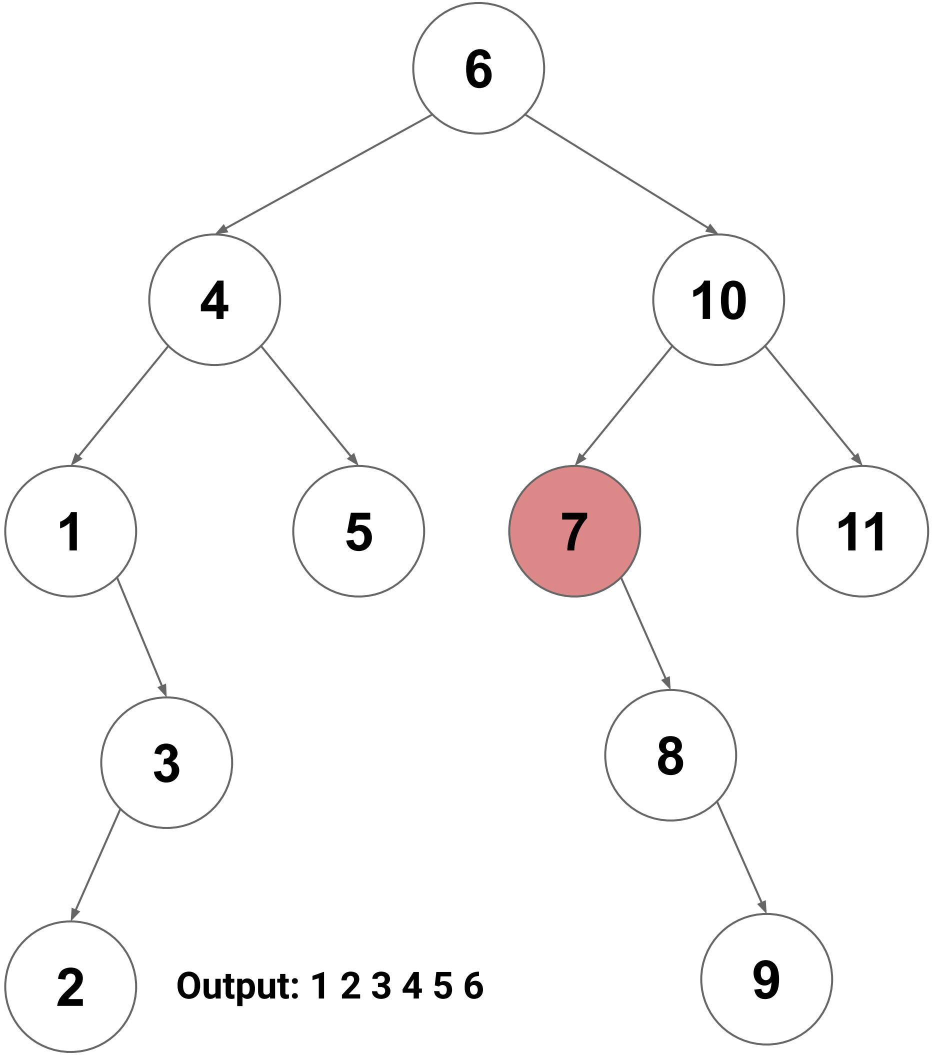
`inorderVisit(6)`

`inorderVisit(10)`



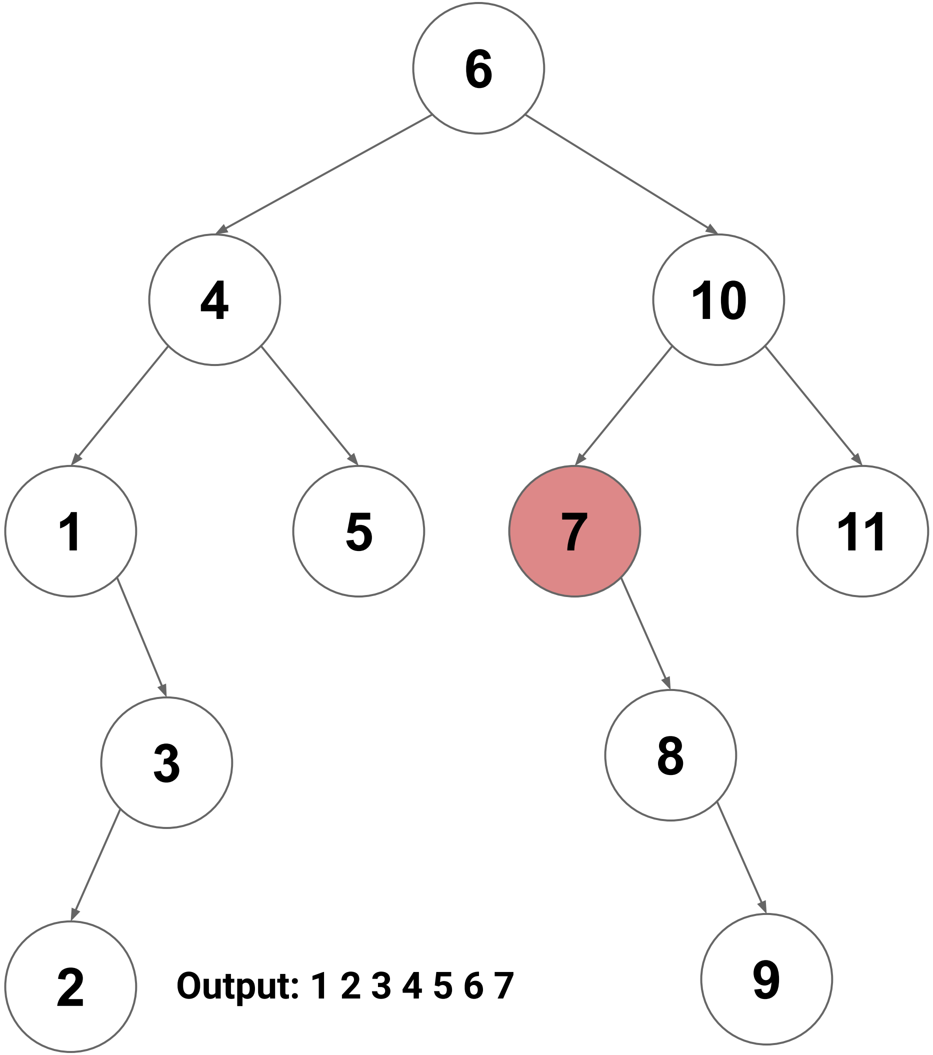
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(10)

inorderVisit(7)
```



In-Order Traversal on a BST

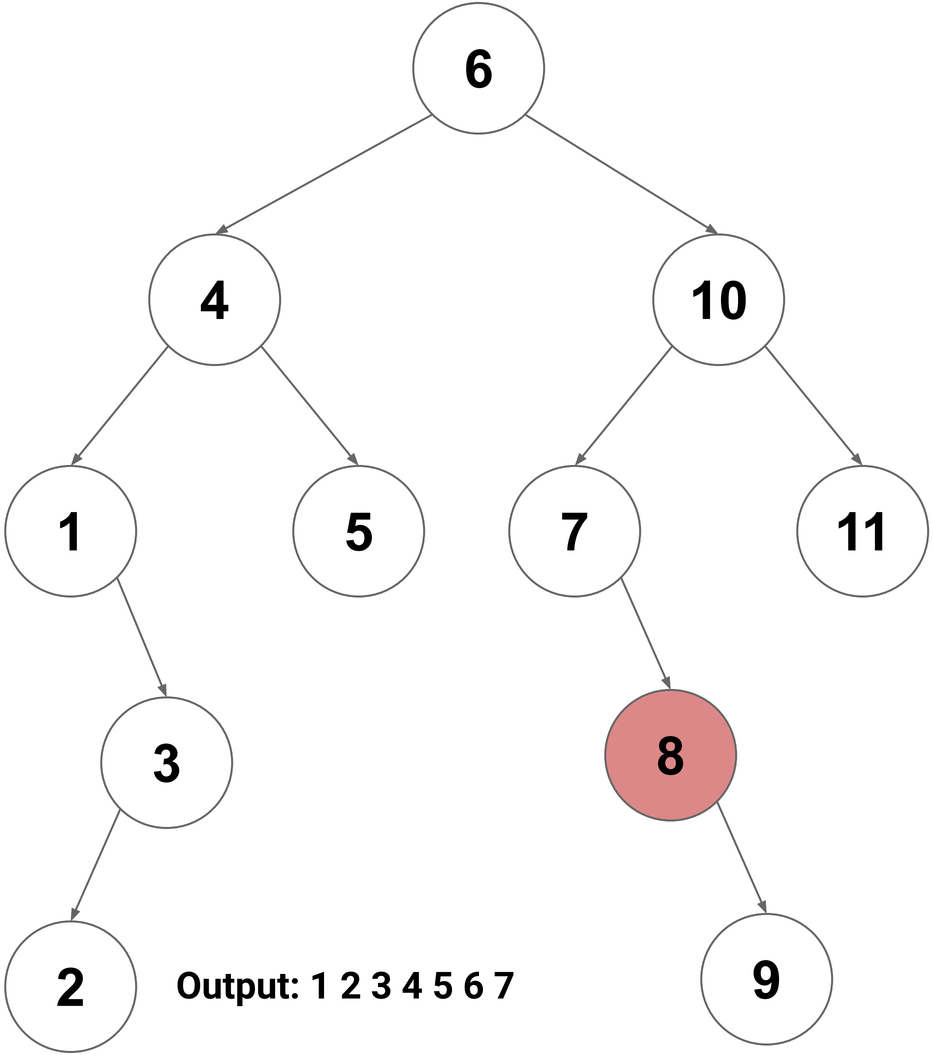
```
inorderVisit(6)

inorderVisit(10)

inorderVisit(7)

visit(7)
```



Output: 1 2 3 4 5 6 7

In-Order Traversal on a BST

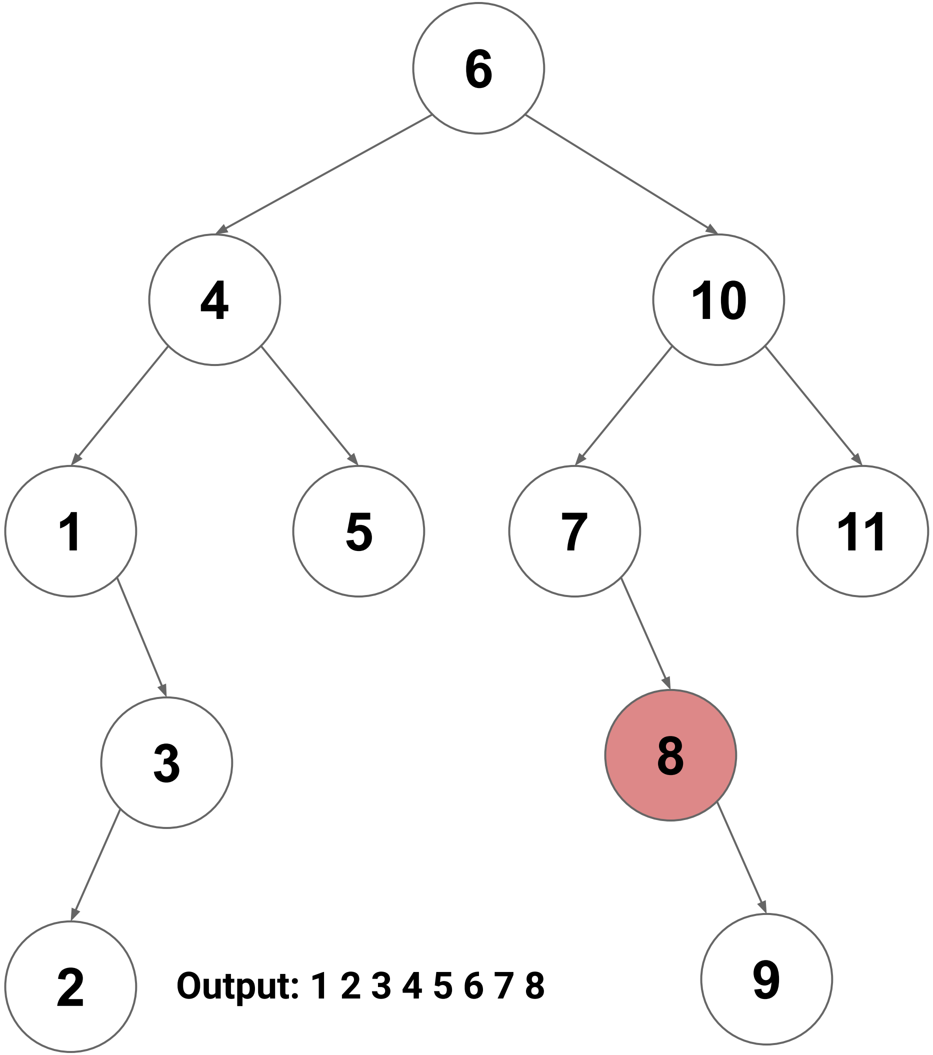
```
inorderVisit(6)

inorderVisit(10)

inorderVisit(7)

inorderVisit(8)
```



Output: 1 2 3 4 5 6 7

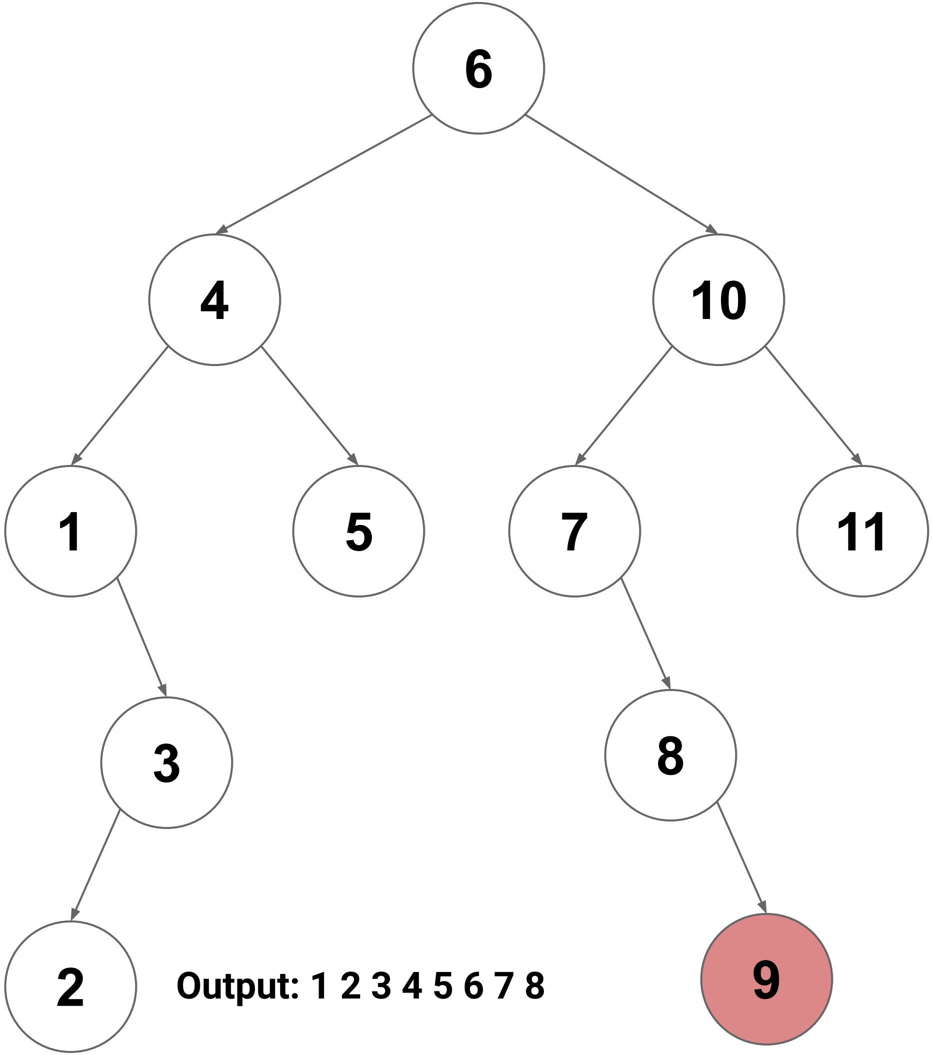
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(10)

inorderVisit(7)

inorderVisit(8)

visit(8)
```



In-Order Traversal on a BST

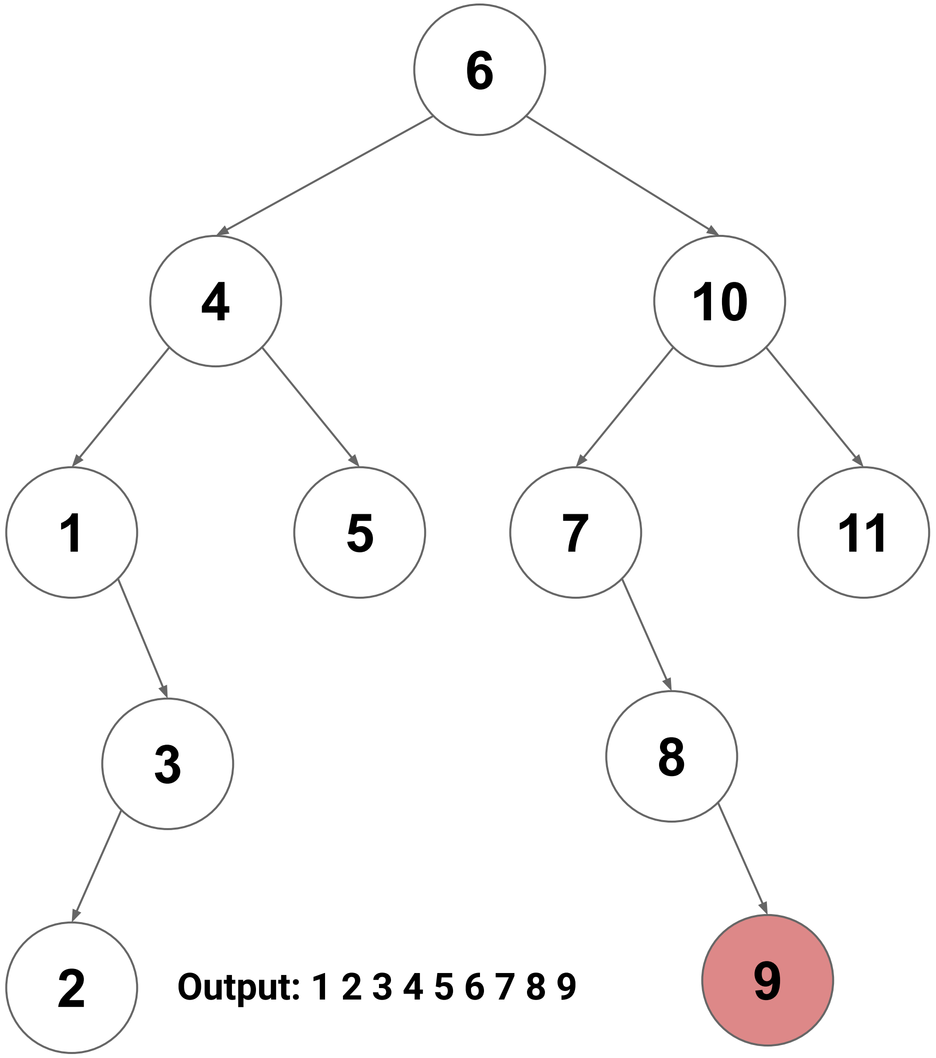
```
inorderVisit(6)

inorderVisit(10)

inorderVisit(7)

inorderVisit(8)

inorderVisit(9)
```



Output: 1 2 3 4 5 6 7 8

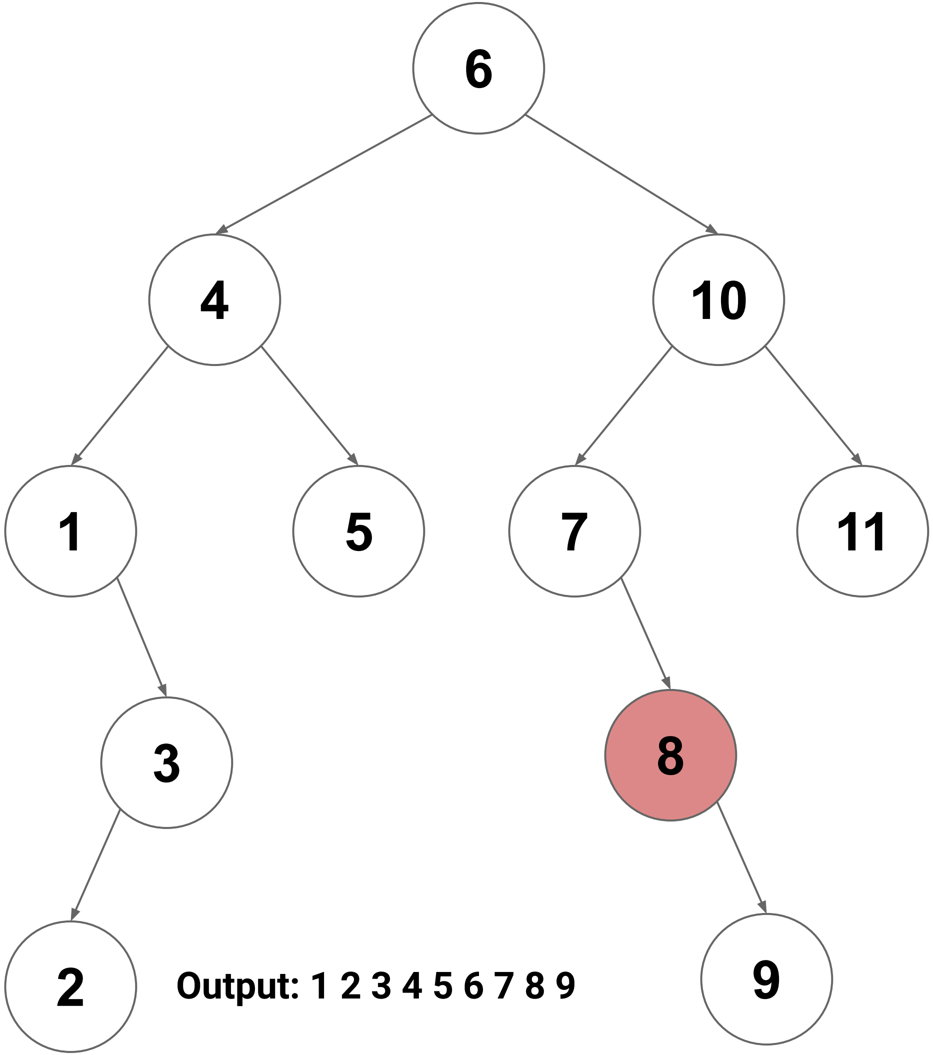
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(10)

inorderVisit(7)

inorderVisit(8)

visit(9)
```



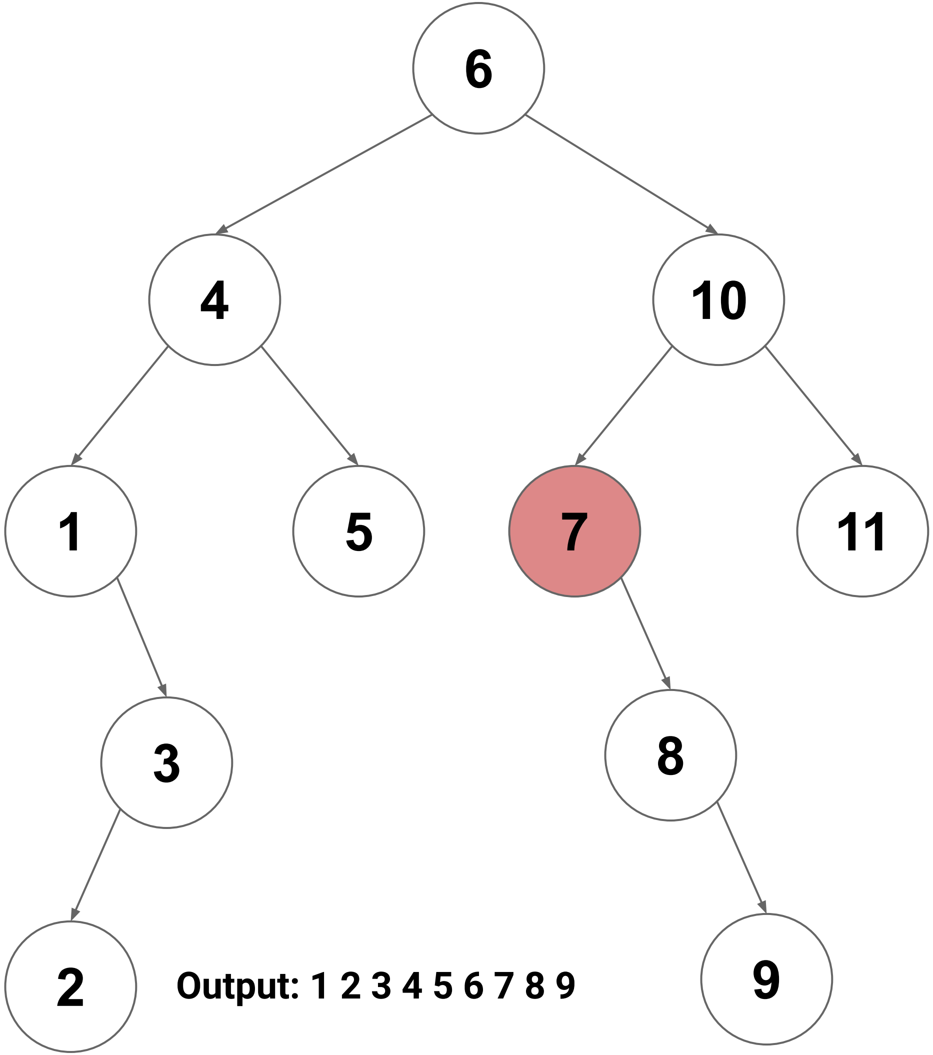
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(10)

inorderVisit(7)

inorderVisit(8)
```



In-Order Traversal on a BST

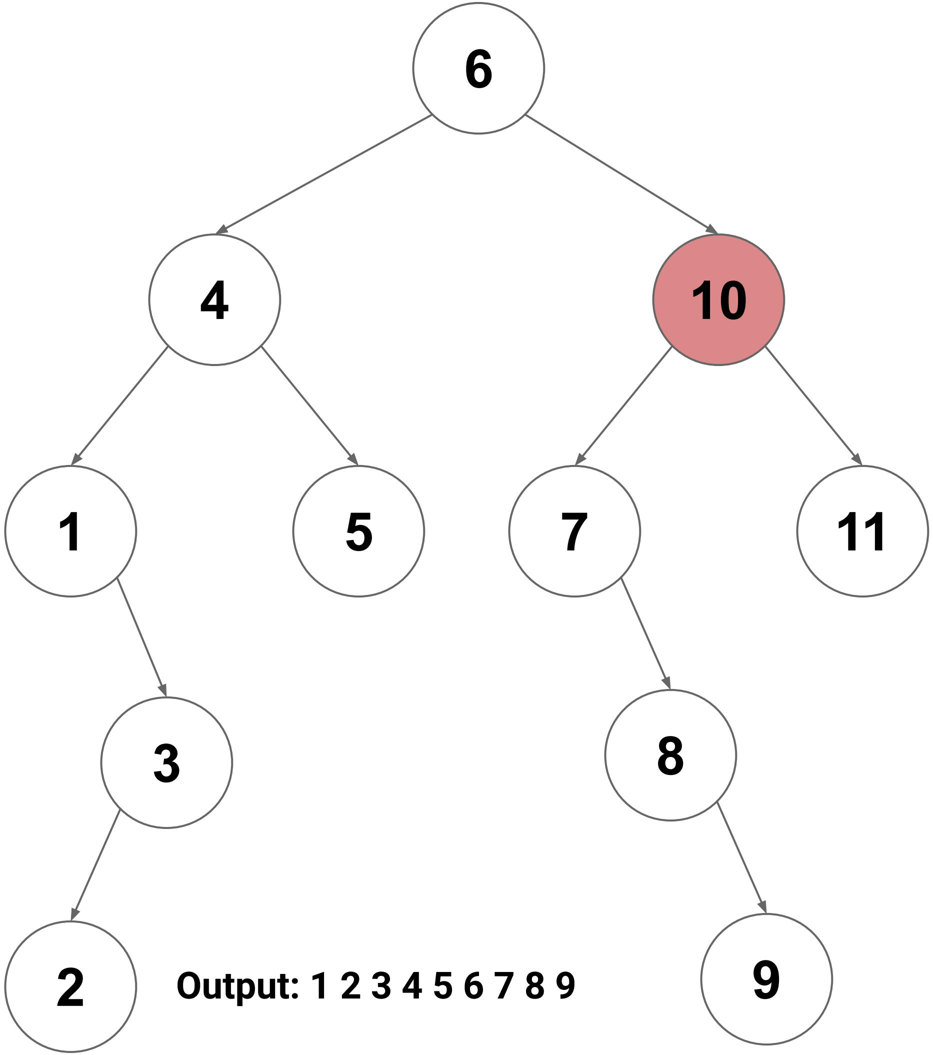
```
inorderVisit(6)

inorderVisit(10)

inorderVisit(7)
```



In-Order Traversal on a BST

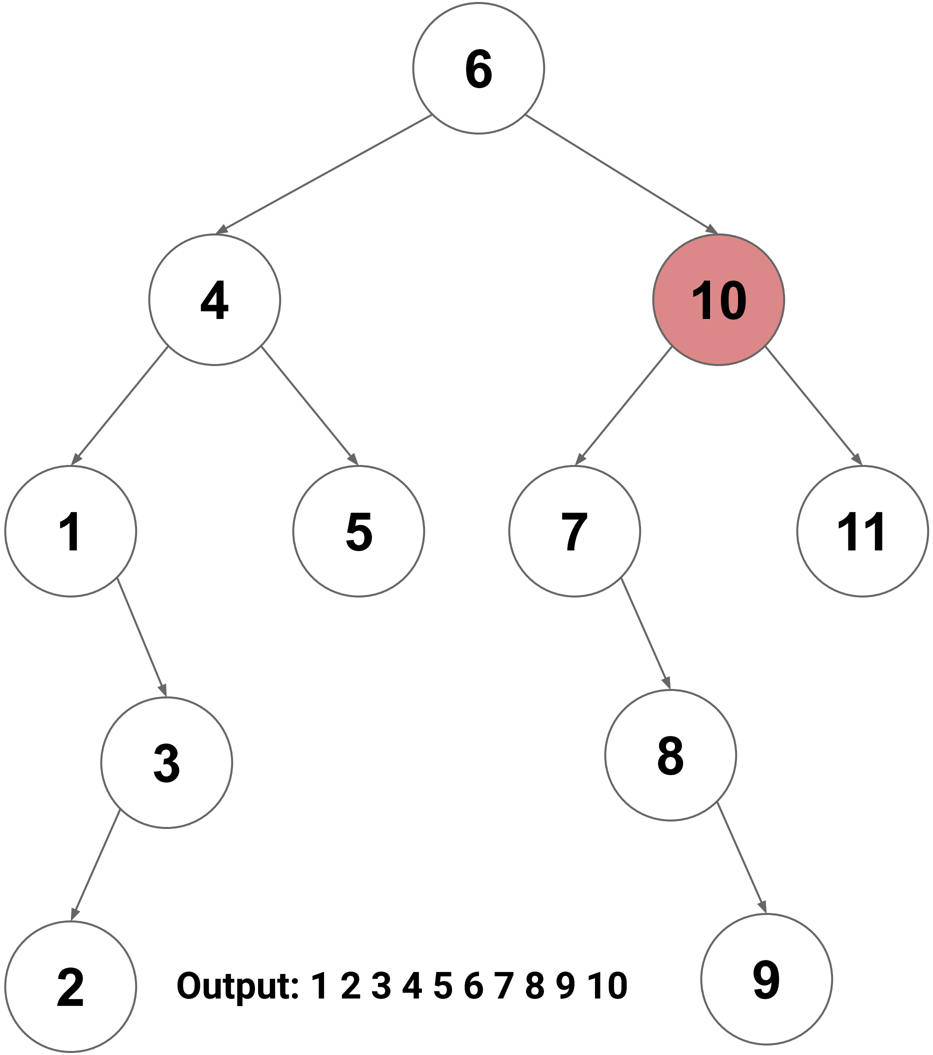
`inorderVisit(6)`

`inorderVisit(10)`



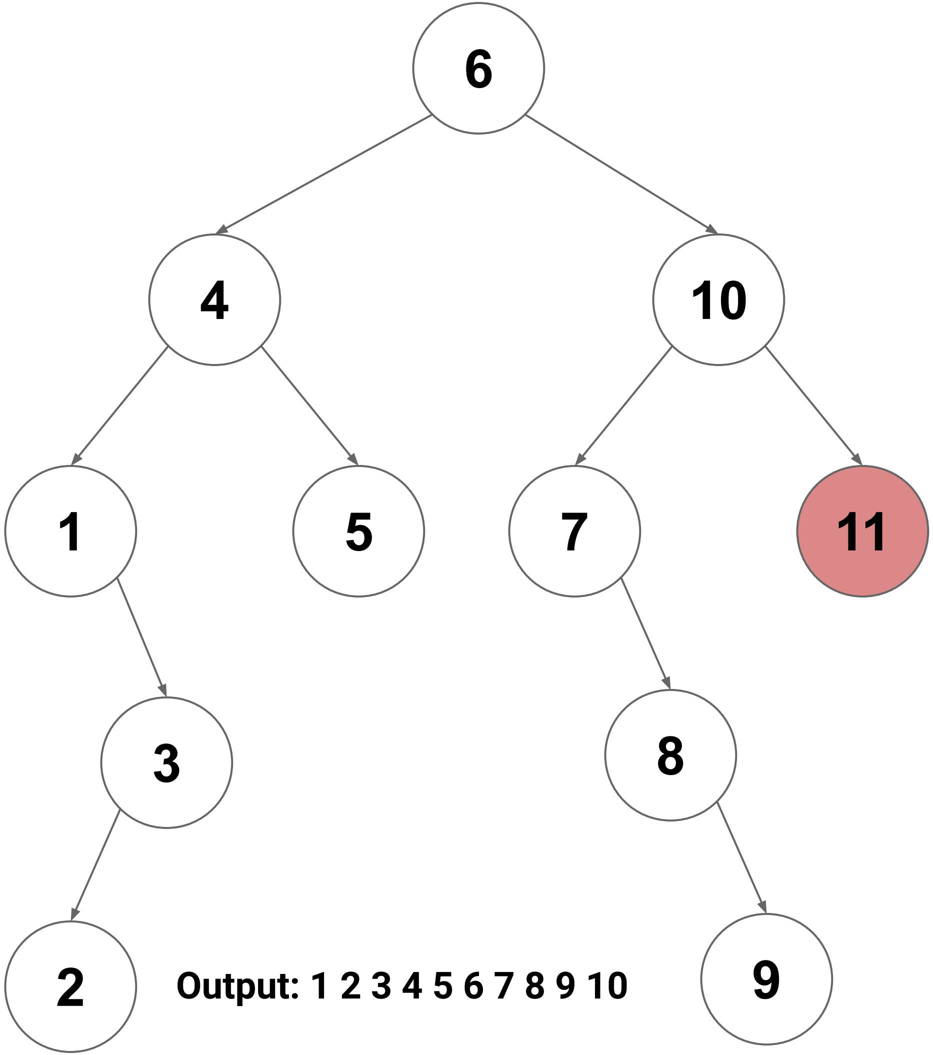
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(10)

visit(10)
```



In-Order Traversal on a BST

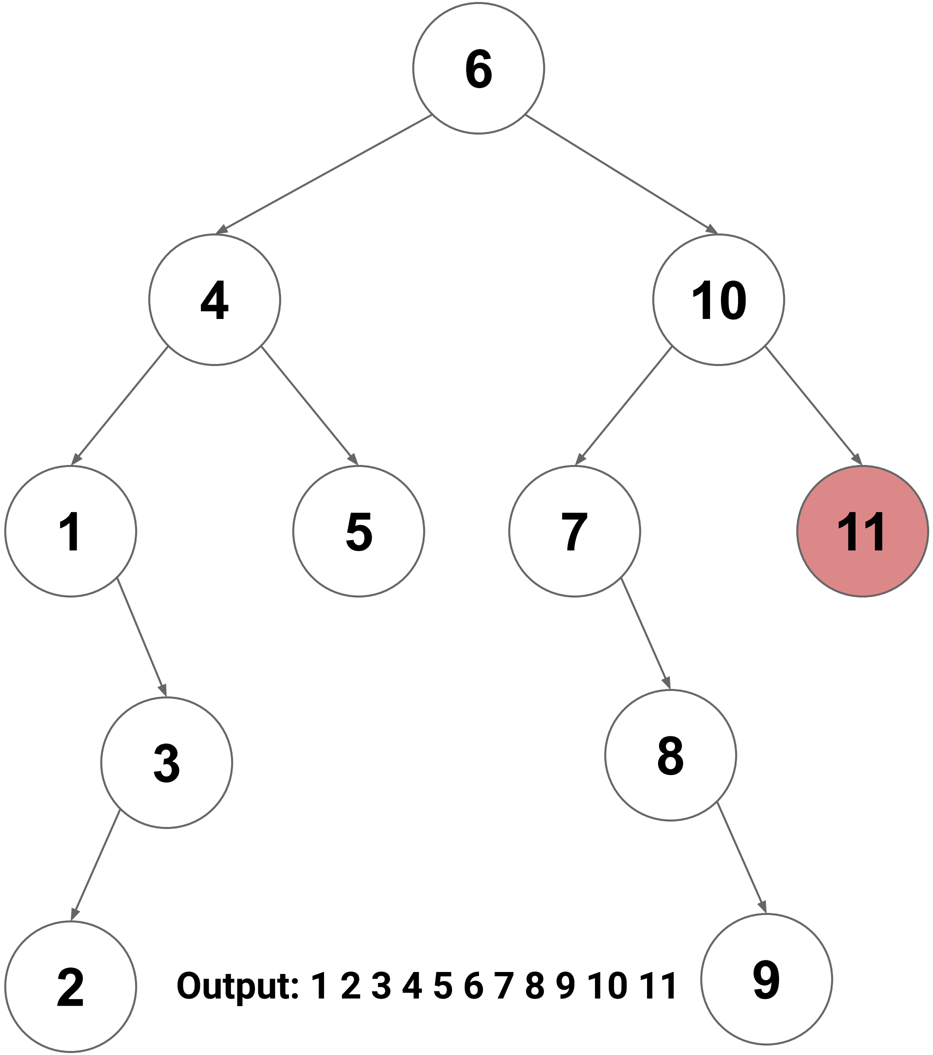
```
inorderVisit(6)

inorderVisit(10)

inorderVisit(11)
```



Output: 1 2 3 4 5 6 7 8 9 10

In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(10)

inorderVisit(11)

visit(11)
```

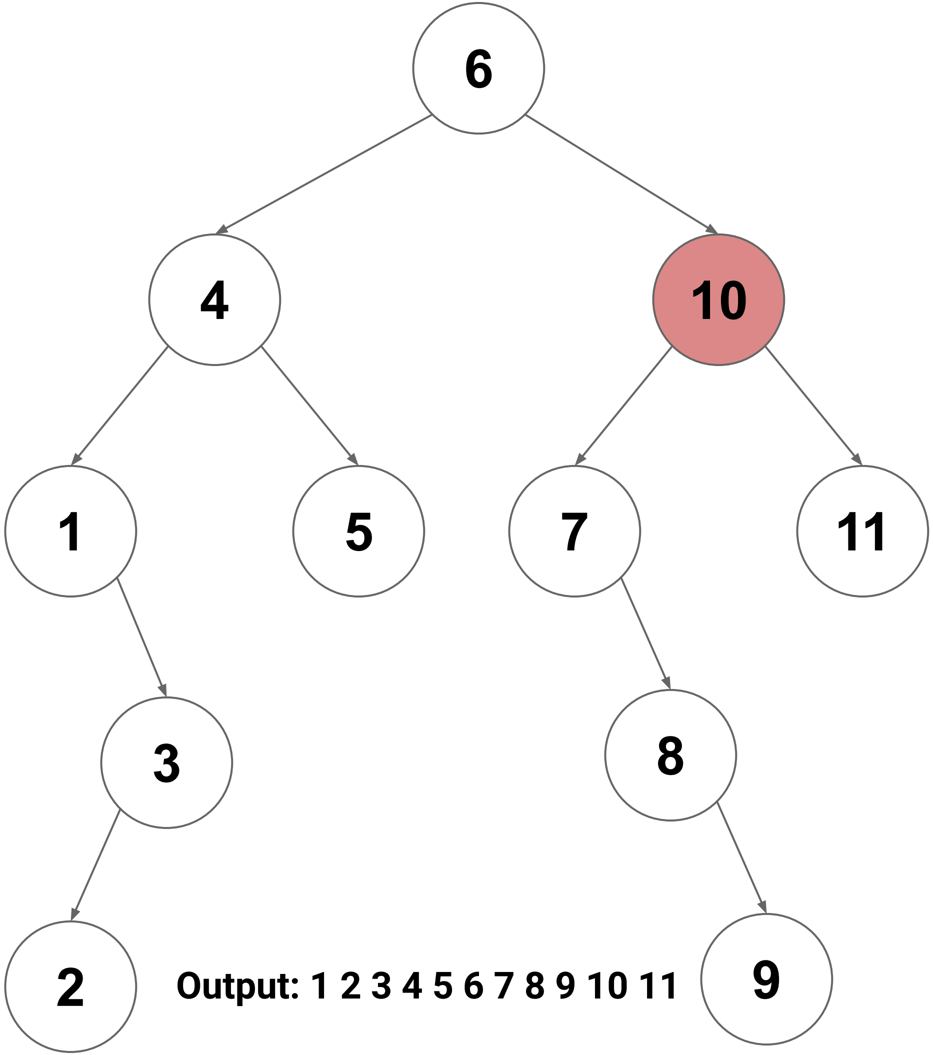


Output: 1 2 3 4 5 6 7 8 9 10 11

In-Order Traversal on a BST

`inorderVisit(6)`

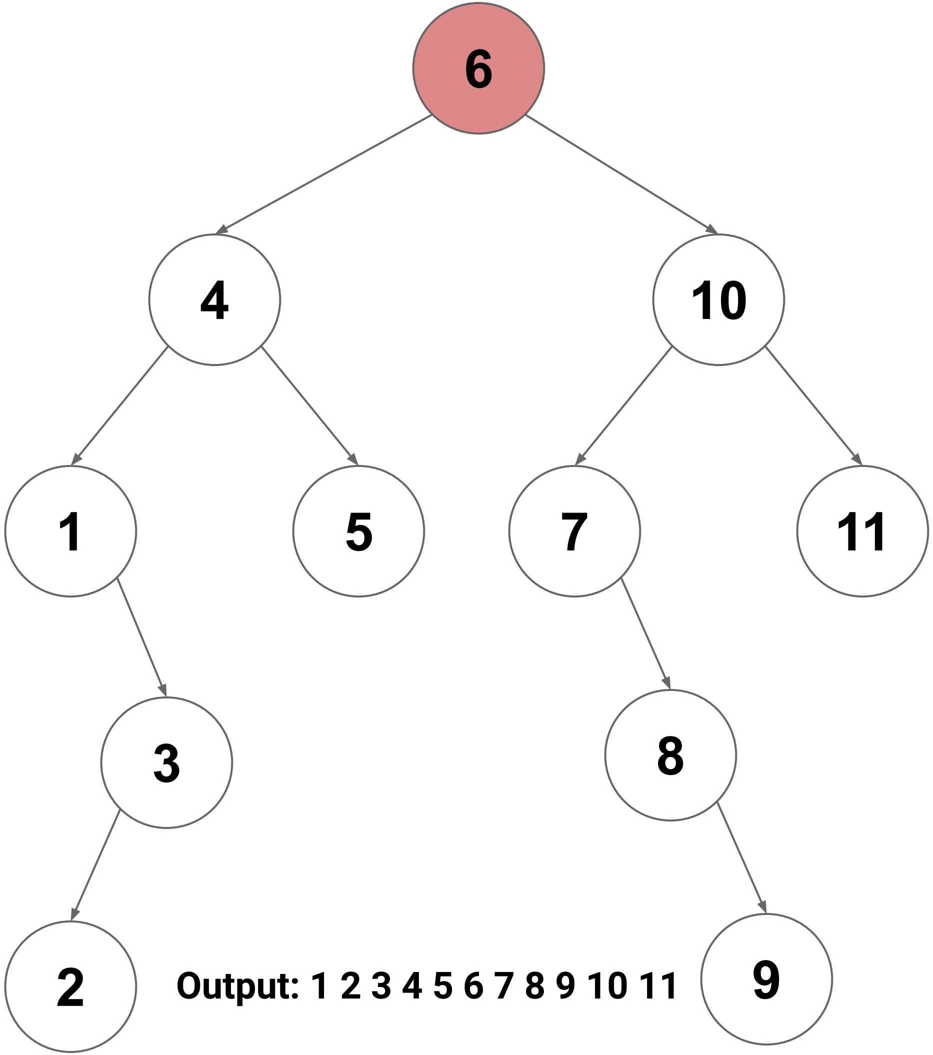
`inorderVisit(10)`



Output: 1 2 3 4 5 6 7 8 9 10 11

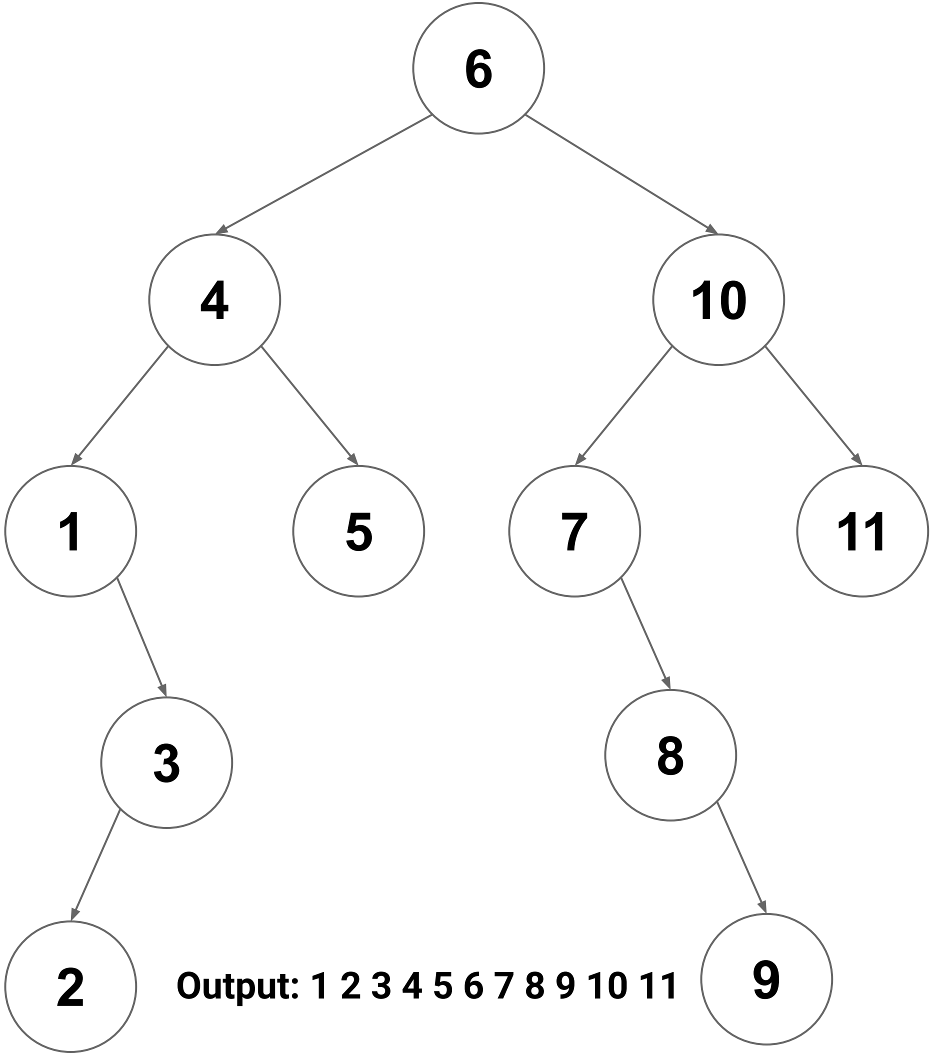
In-Order Traversal on a BST

`inorderVisit(6)`



Output: 1 2 3 4 5 6 7 8 9 10 11

In-Order Traversal on a BST



Output: 1 2 3 4 5 6 7 8 9 10 11

Tree Traversal: In-Order Iterator

```
class ImmutableTreeIterator[T](root: ImmutableTree[T]) {
  /*** Initialize the Iterator ***/
  val toVisit = mutable.Stack[ImmutableTree[T]]
  pushLeft(root)

  def pushLeft(node: ImmutableTree[T]): Unit =
    node match {
      case EmptyTree => ()
      case t: ImmutableTree =>
        toVisit.push(t)
        pushLeft(t.left)
    }
  …
```

Tree Traversal: In-Order Iterator

```
class ImmutableTreeIterator[T](root: ImmutableTree[T]) {
  /*** Initialize the Iterator ***/
  val toVisit = mutable.Stack[ImmutableTree[T]]
  pushLeft(root)

  def pushLeft(node: ImmutableTree[T]): Unit =
    node match {
      case EmptyTree => ()
      case t: ImmutableTree =>
        toVisit.push(t)
        pushLeft(t.left)
    }
  ...
```

Initialize our iterator by recursively pushing the left trees (we know the FIRST element in an in-order traversal is the left-most

Tree Traversal: In-Order Iterator

```scala
class ImmutableTreeIterator[T](root: ImmutableTree[T]) {
  /*** Initialize the Iterator ***/
  val toVisit = mutable.Stack[ImmutableTree[T]]
  pushLeft(root)

  def pushLeft(node: ImmutableTree[T]): Unit =
    node match {
      case EmptyTree => ()
      case t: ImmutableTree =>
        toVisit.push(t)
        pushLeft(t.left)
    }
  …
```

Initialize our iterator by recursively pushing the left trees (we know the FIRST element in an in-order traversal is the left-most

This pushes nodes onto our toVisit Stack, followed by their left trees (LIFO!)

Tree Traversal: In-Order Iterator

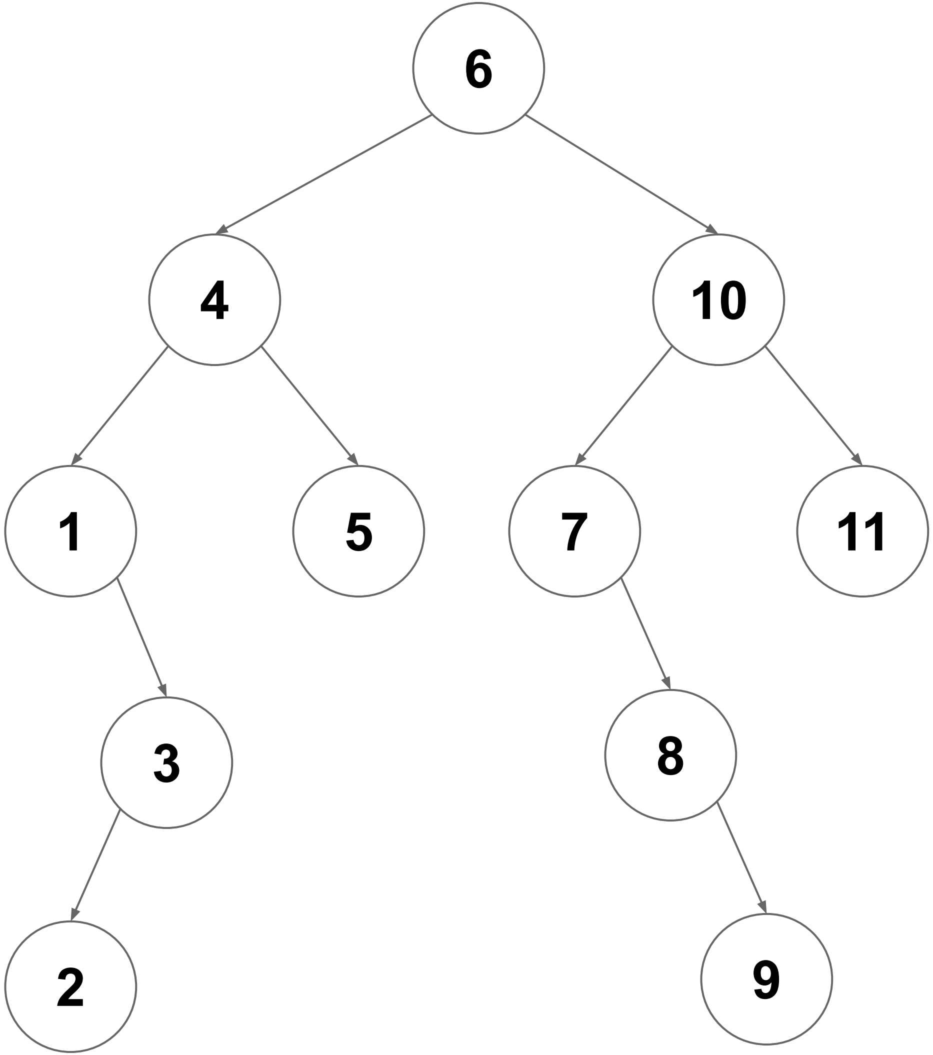
```
class ImmutableTreeIterator[T](root: ImmutableTree[T]) {

  ...

  def isEmpty = toVisit.isEmpty

  def next: T = {
    val nextNode = toVisit.pop
    pushLeft(nextNode.right)
    return nextNode.value
  }
}
```

Tree Traversal: In-Order Iterator

```
class ImmutableTreeIterator[T](root: ImmutableTree[T]) {

  ...

  def isEmpty = toVisit.isEmpty

  def next: T = {
    val nextNode = toVisit.pop
    pushLeft(nextNode.right)
    return nextNode.value
  }
}
```

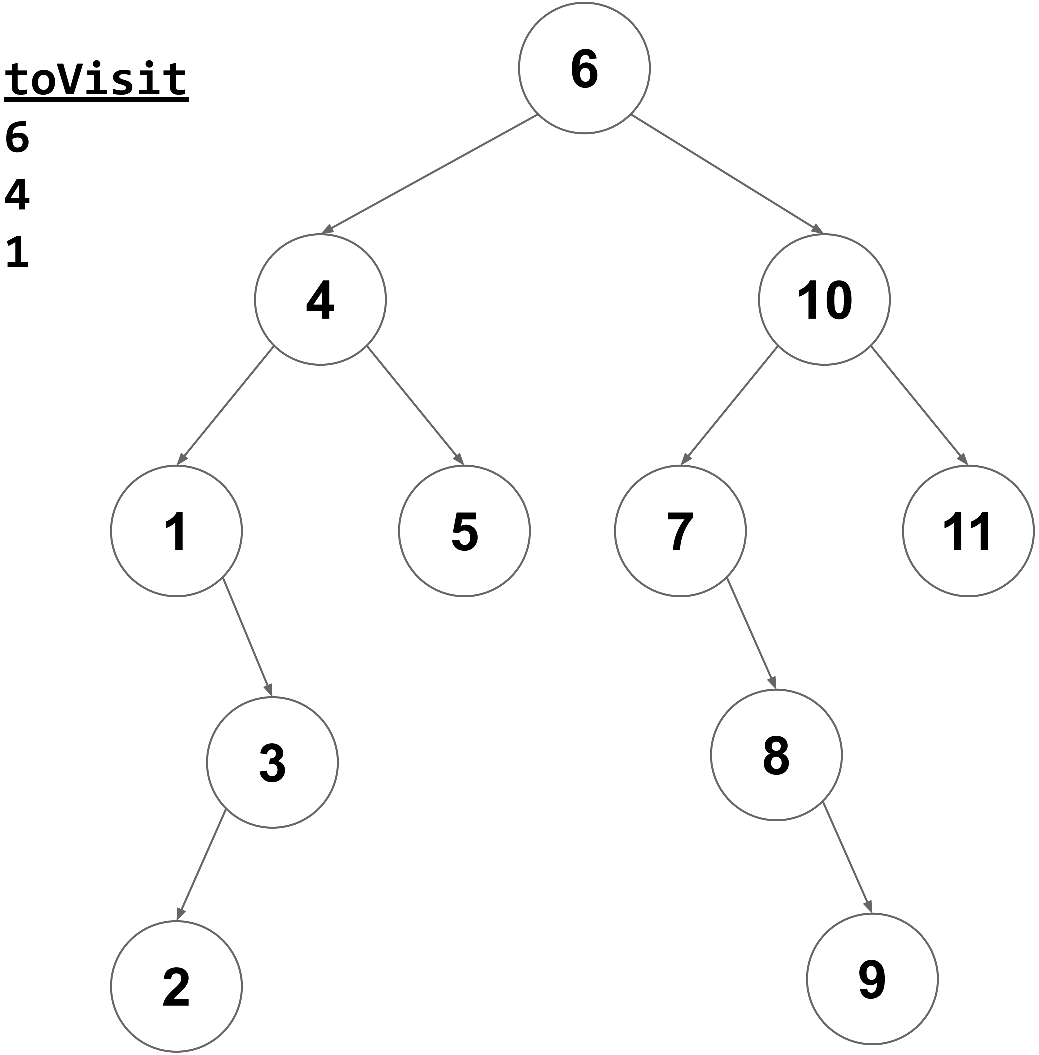
next pops the next node from our stack, and pushes it's right subtree, then returns it

In-Order Traversal with an Iterator



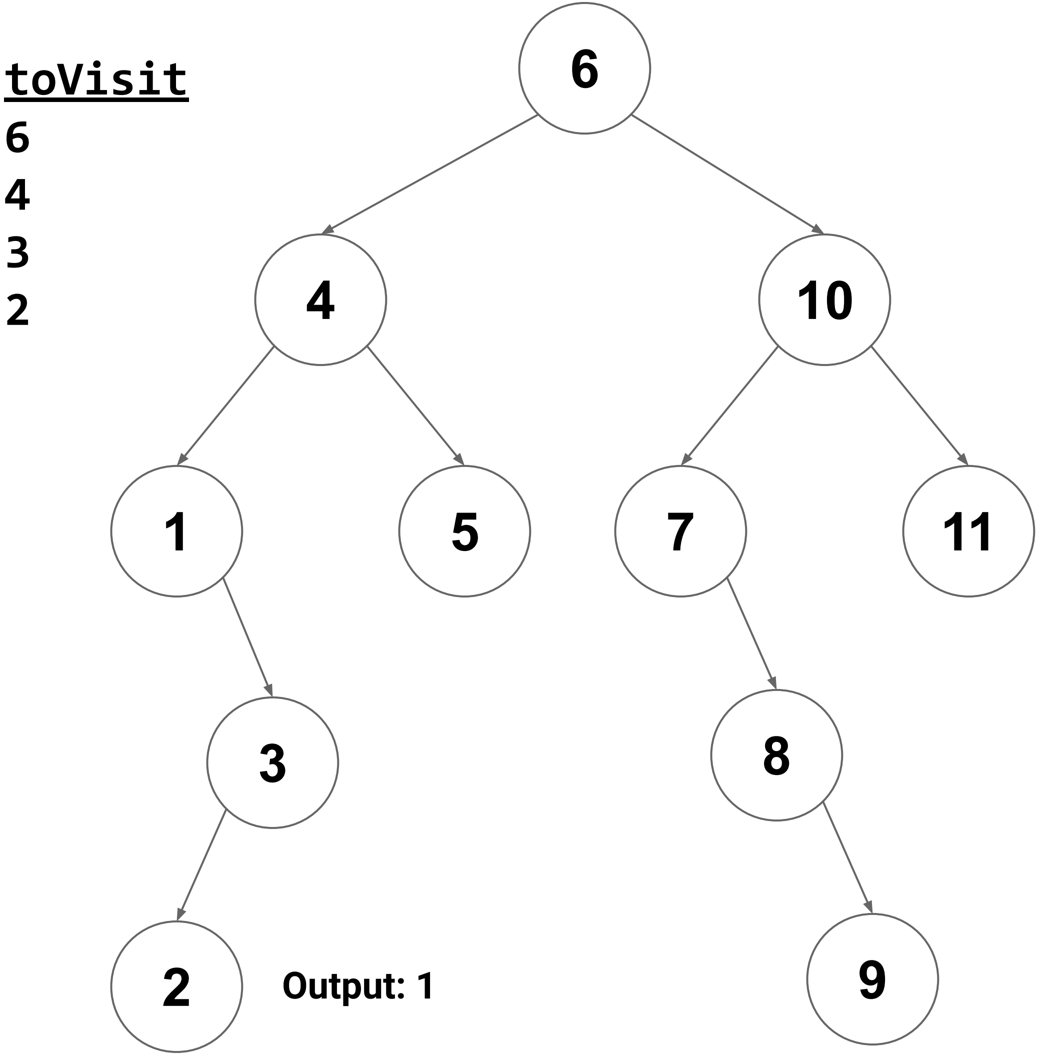
In-Order Traversal with an Iterator

When we create the iterator, the **`toVisit`** stack is initialized



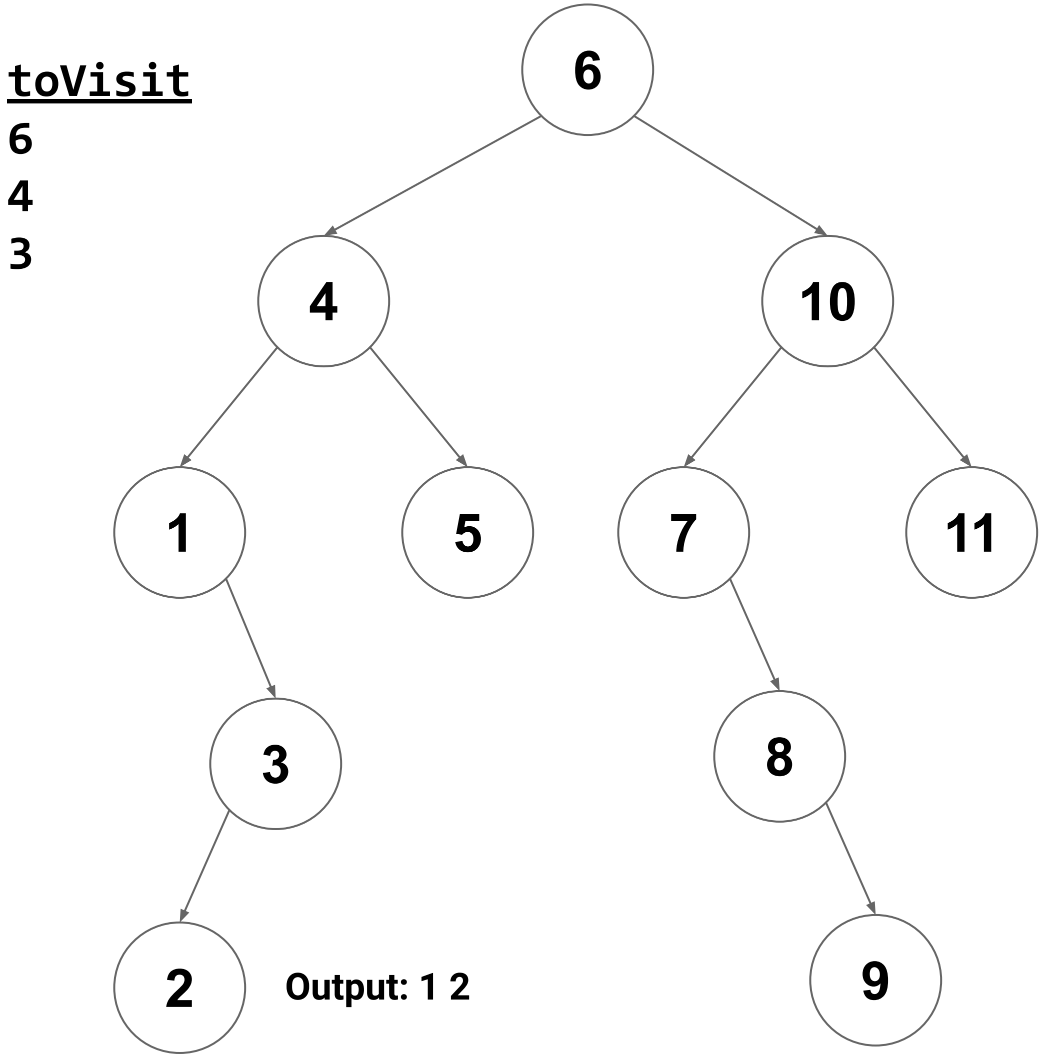
In-Order Traversal with an Iterator

`next` pops the stack (1), and calls `pushLeft` on the right subtree of 1



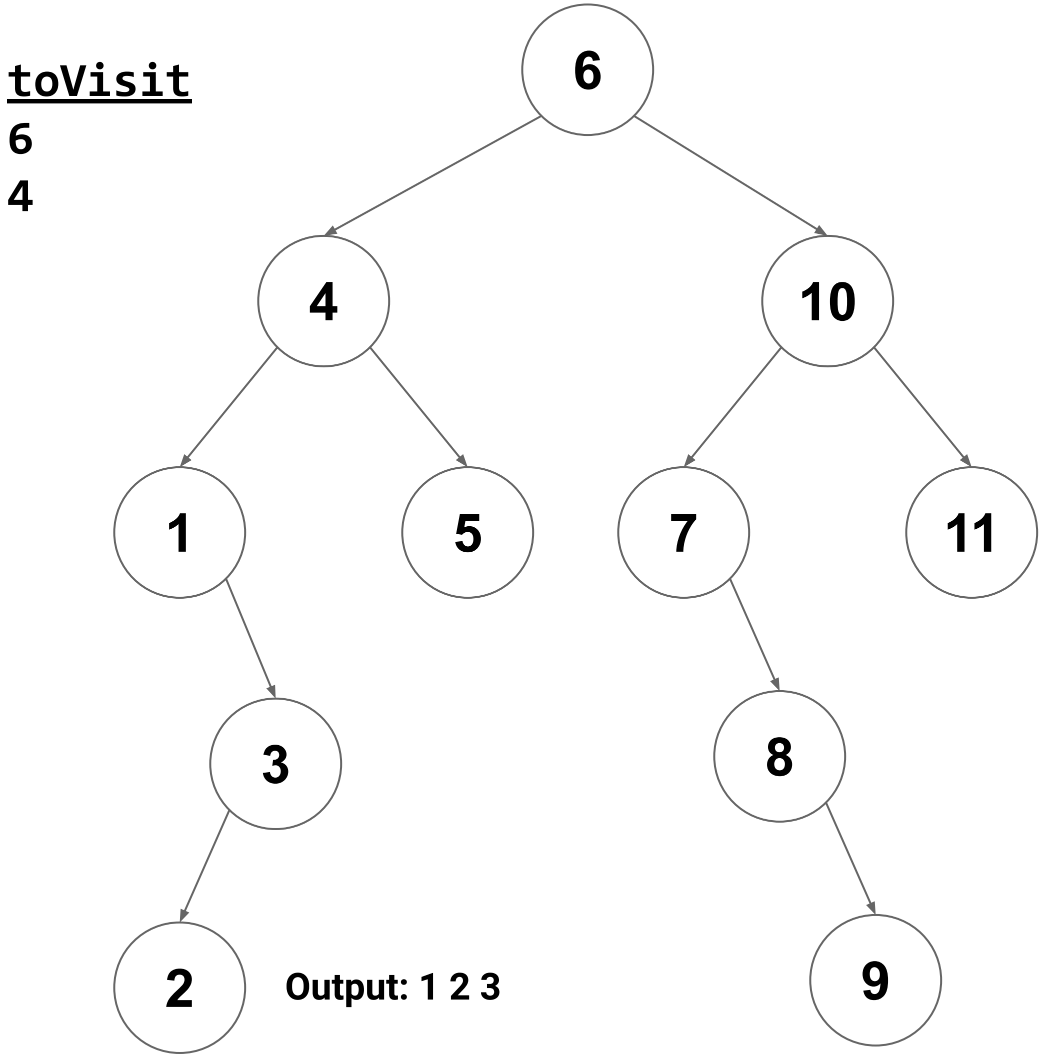
In-Order Traversal with an Iterator

next pops the stack (2) and pushes the right subtree (nothing)



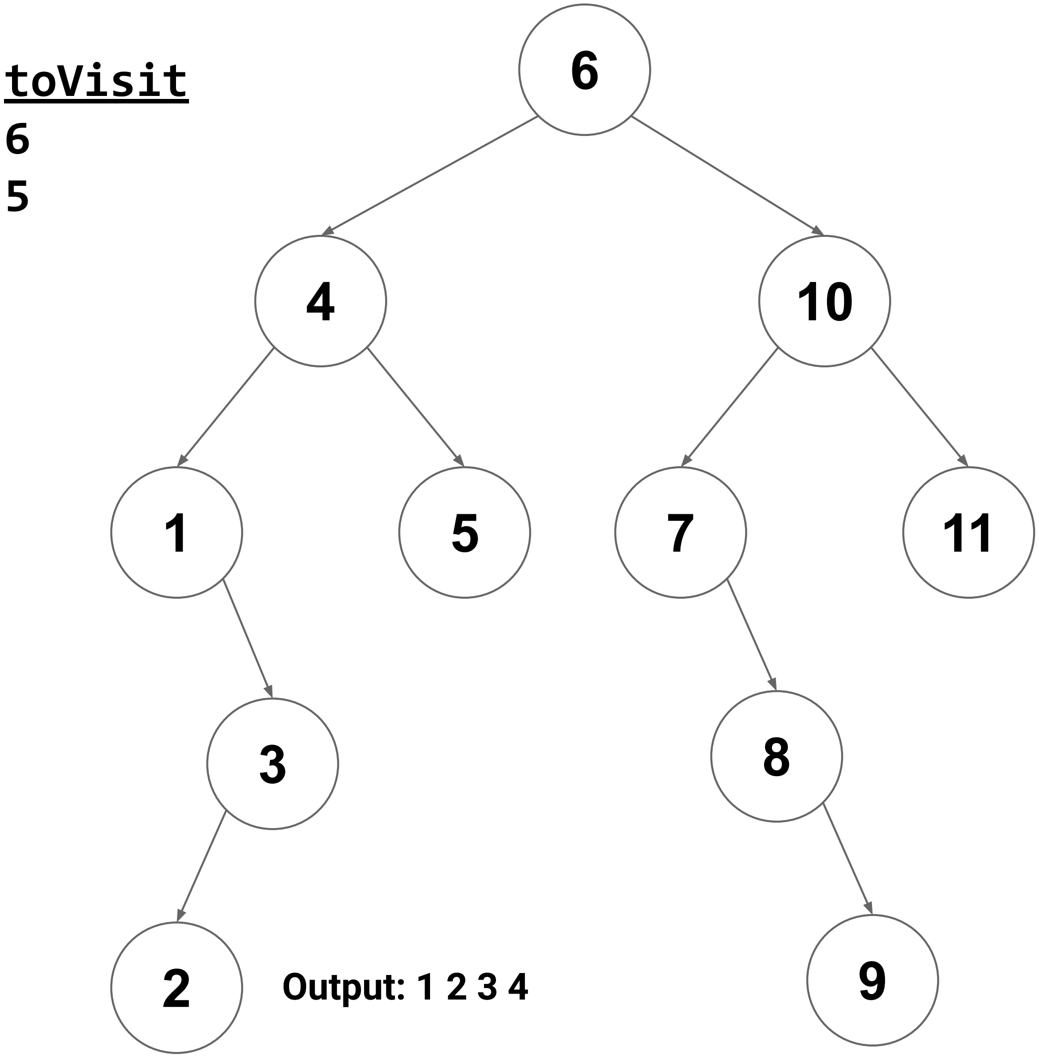
In-Order Traversal with an Iterator

next pops the stack (3) and pushes the right subtree (nothing)



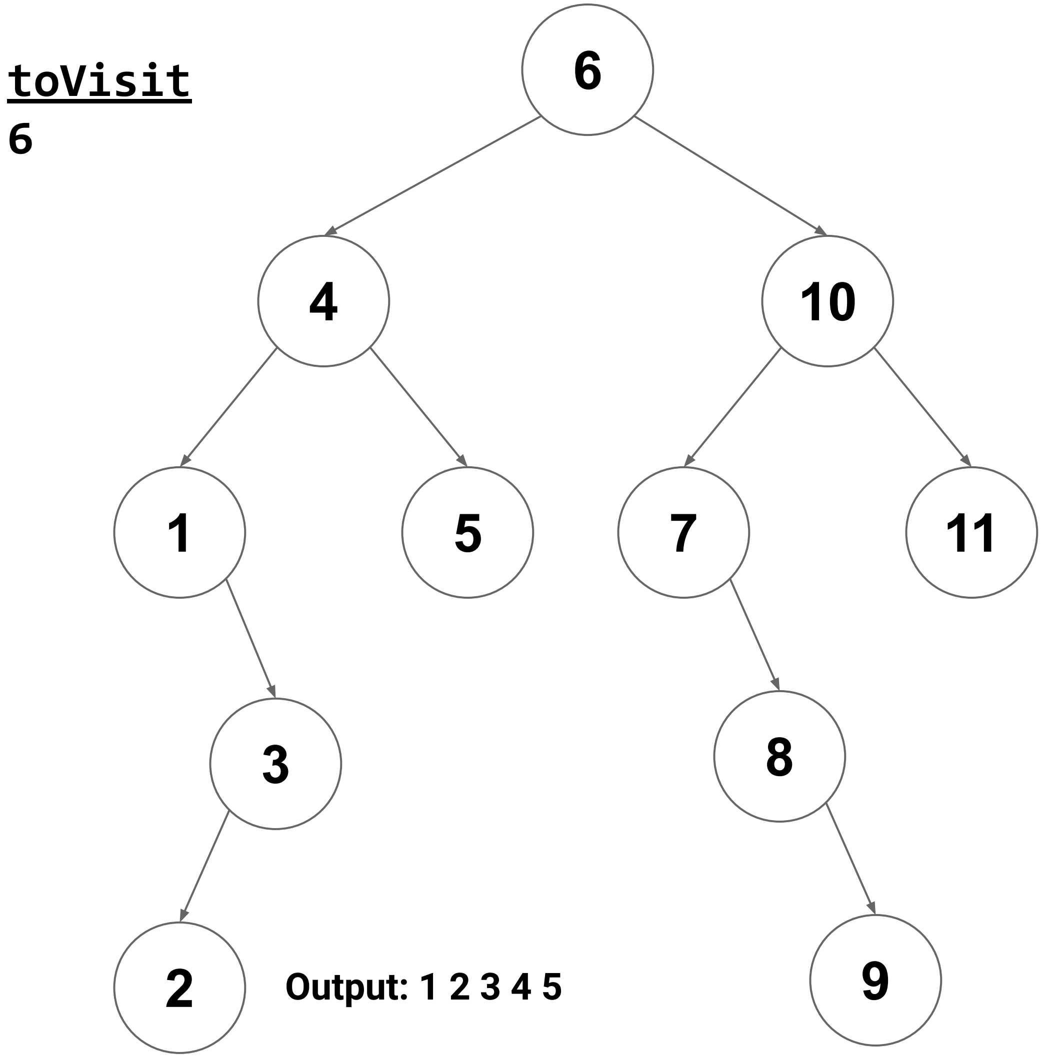
In-Order Traversal with an Iterator

next pops the stack (4) and pushes the right subtree



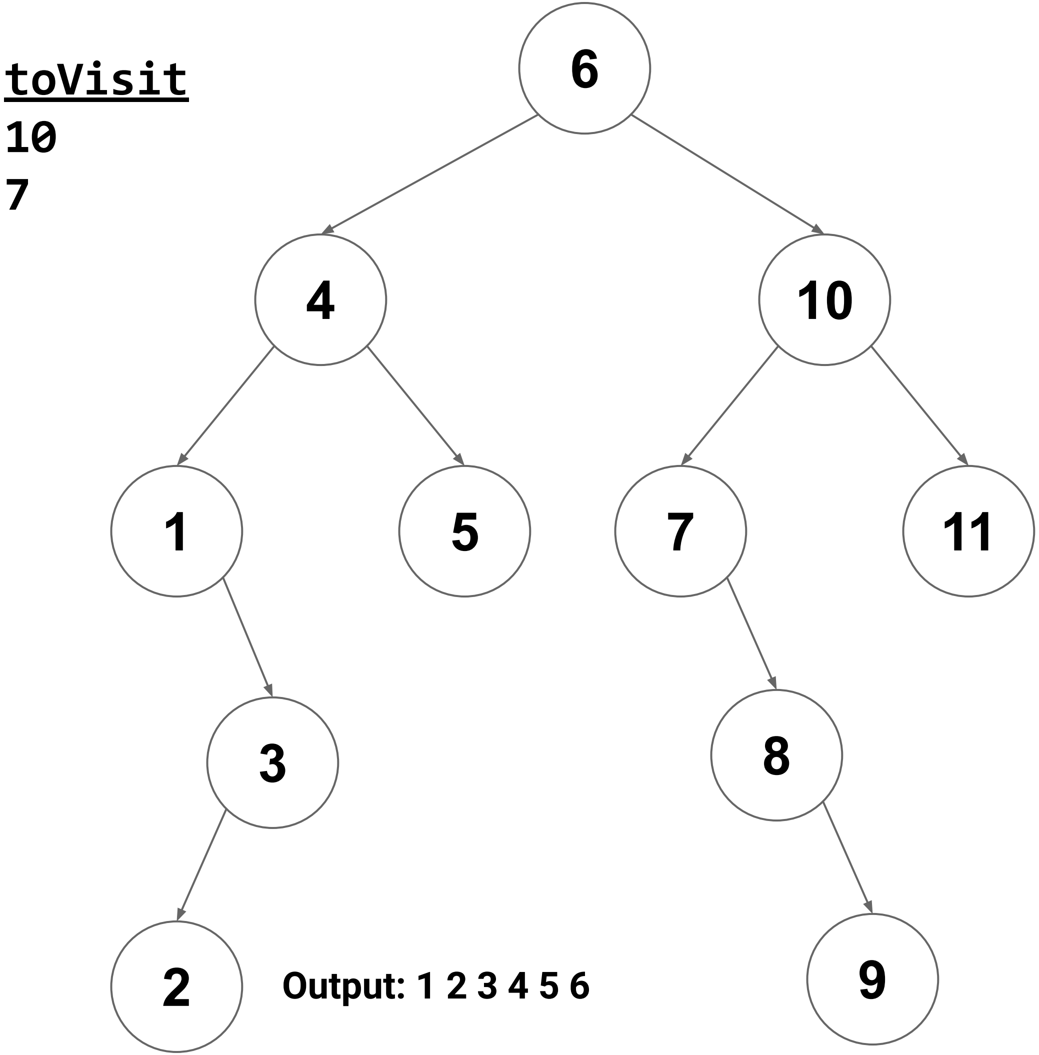
In-Order Traversal with an Iterator

next pops the stack (5) and pushes the right subtree (nothing)



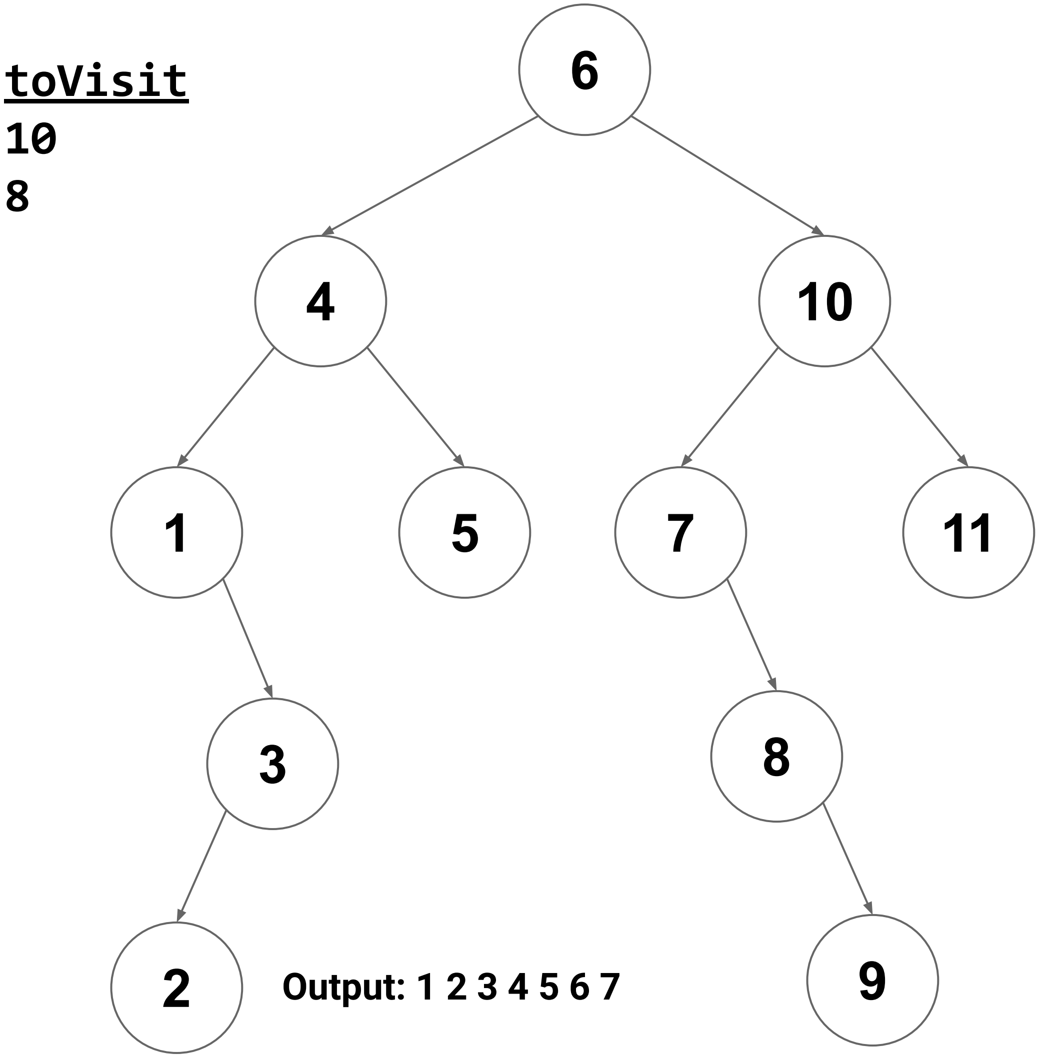
In-Order Traversal with an Iterator

next pops the stack (6) and pushes the right subtree (10 7)



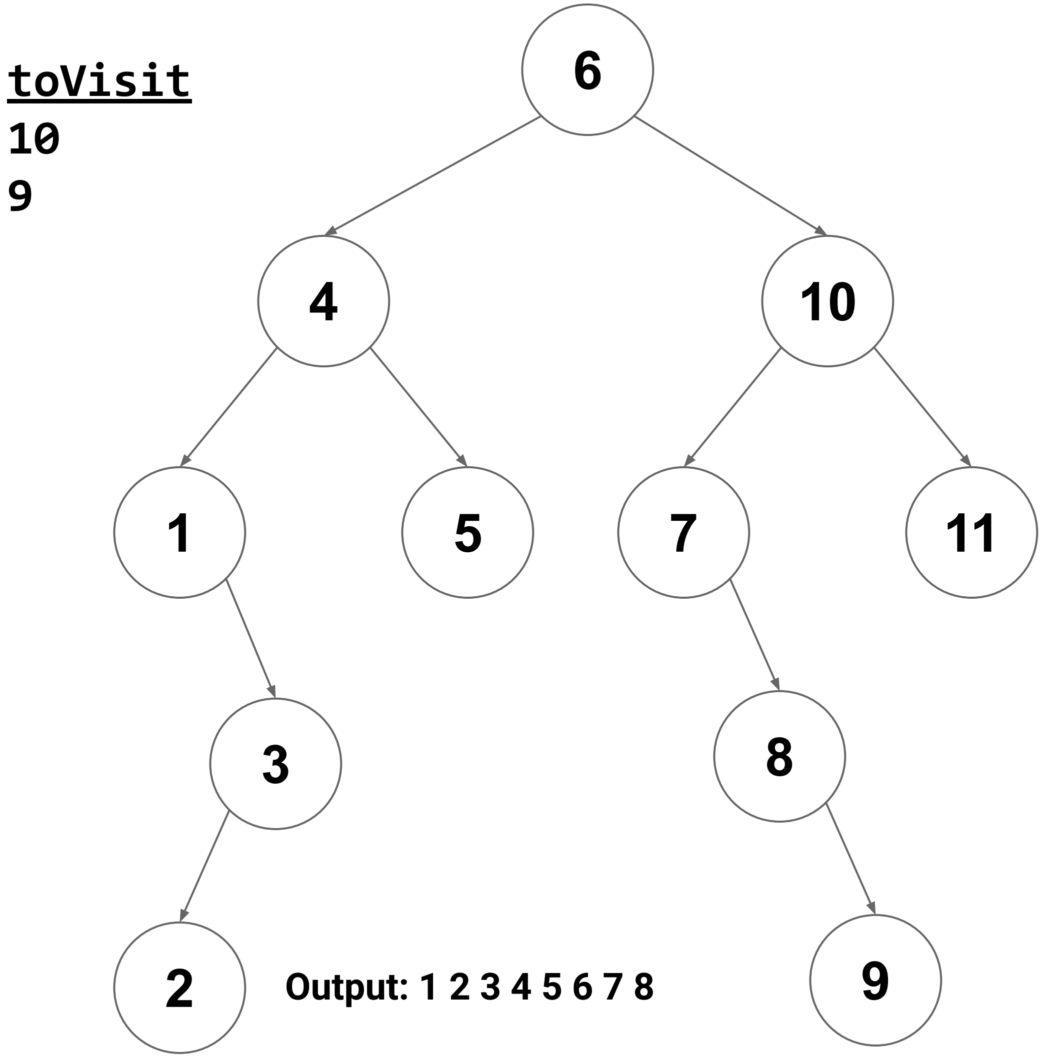
In-Order Traversal with an Iterator

next pops the stack (7) and pushes the right subtree (8)



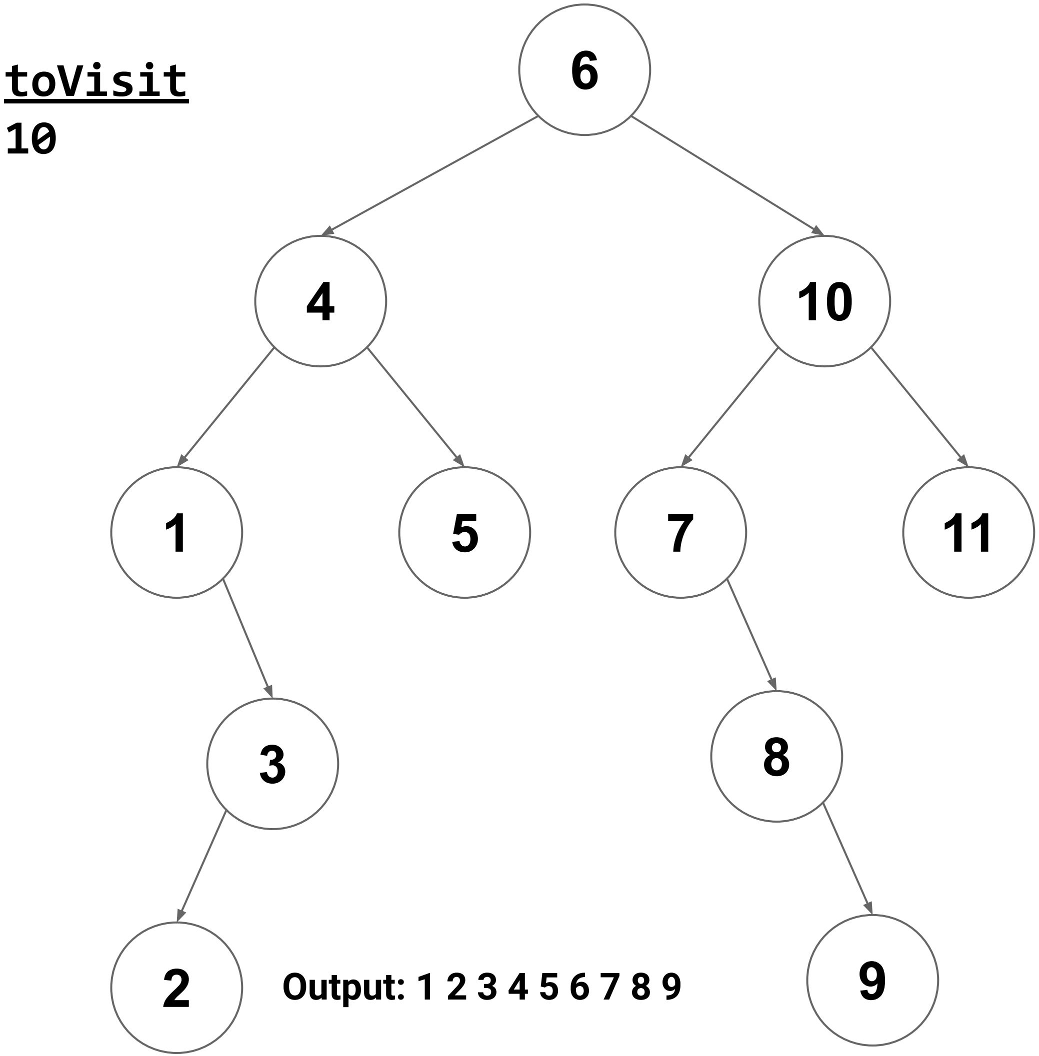
In-Order Traversal with an Iterator

next pops the stack (8) and pushes the right subtree (9)



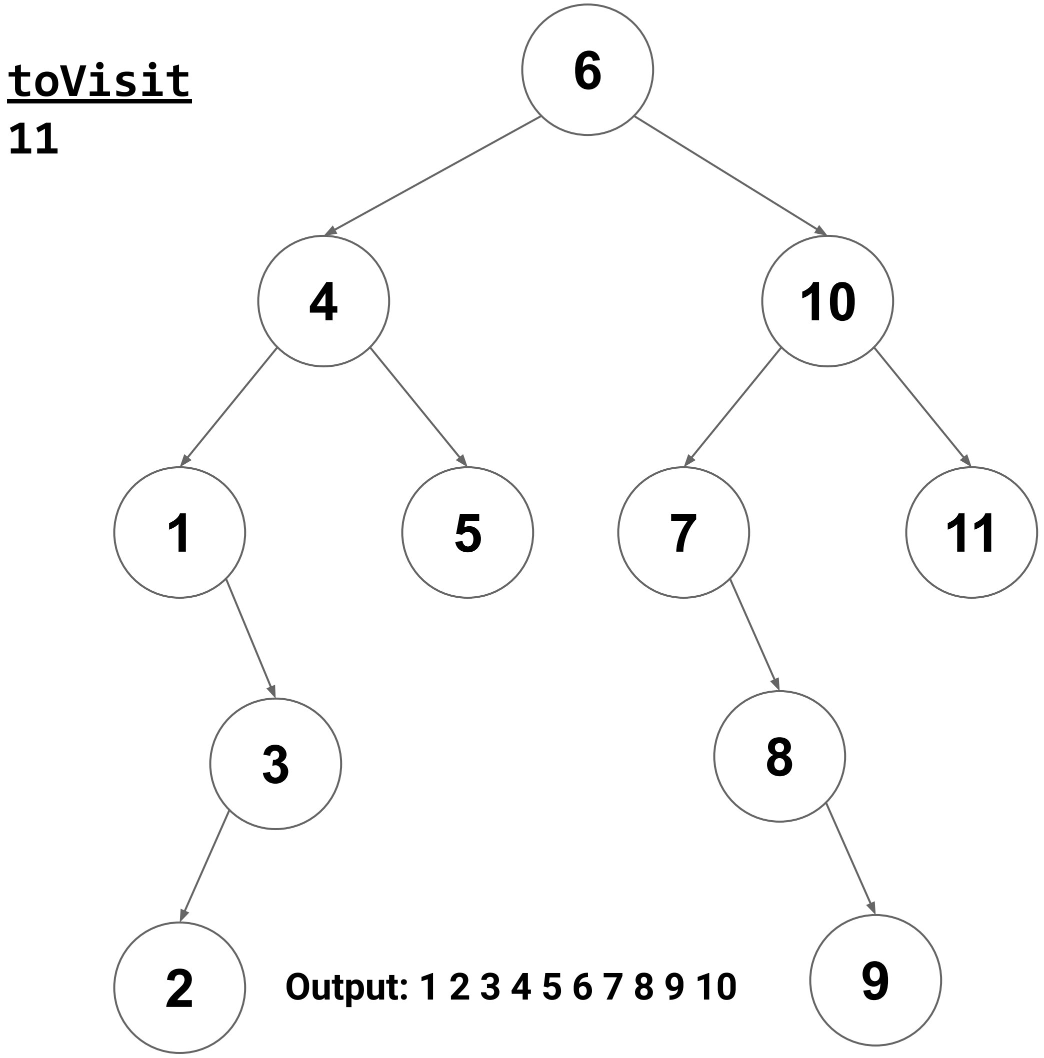
In-Order Traversal with an Iterator

next pops the stack (9) and pushes the right subtree (nothing)



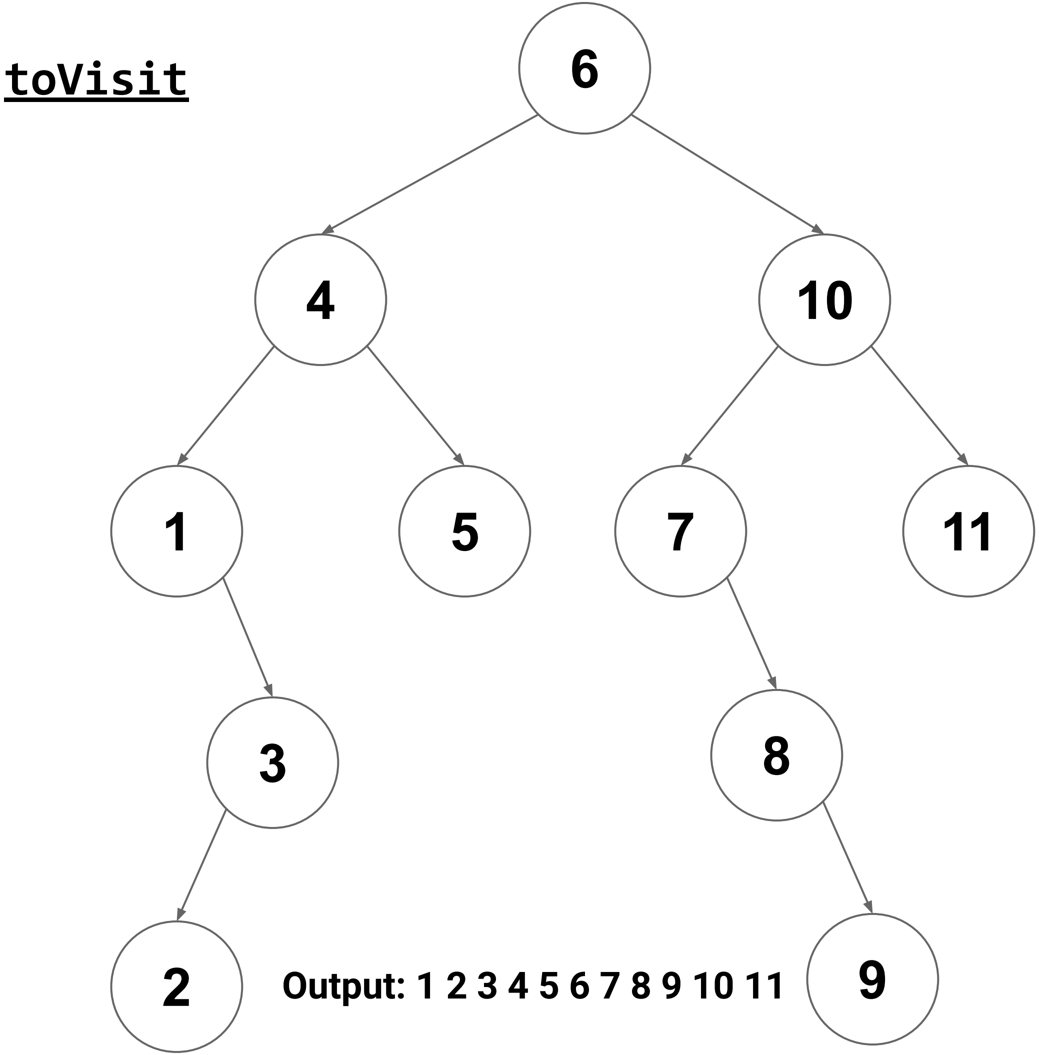
In-Order Traversal with an Iterator

next pops the stack (10) and pushes the right subtree (11)



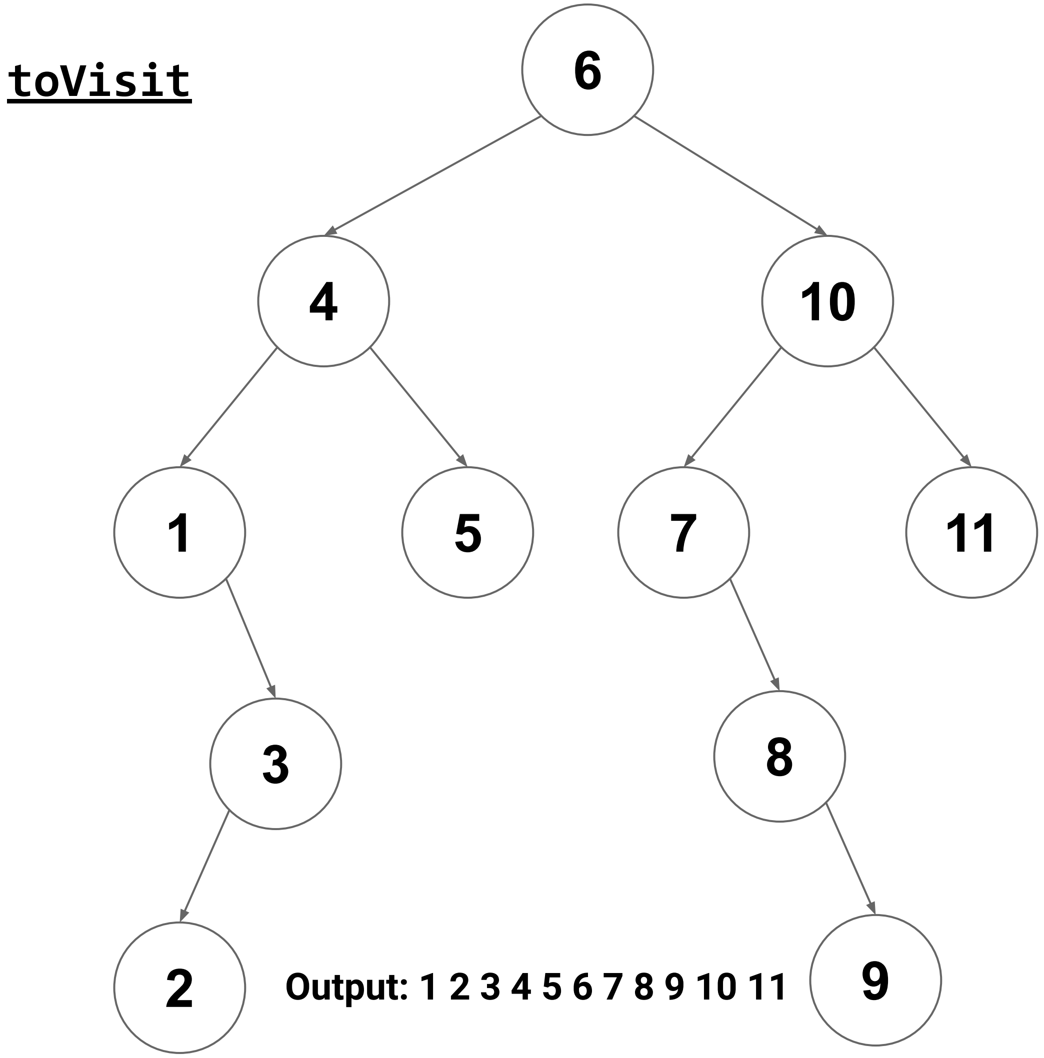
In-Order Traversal with an Iterator

next pops the stack (11) and pushes the right subtree (nothing)



In-Order Traversal with an Iterator

Our **`toVisit`** stack is empty, so **`isEmpty`** will now be true



Complexity

```
val toVisit = mutable.Stack[ImmutableTree[T]]
pushLeft(root)
```

What is our worst-case runtime to initialize the iterator?

Complexity

```
val toVisit = mutable.Stack[ImmutableTree[T]]
pushLeft(root)
```

What is our worst-case runtime to initialize the iterator? ***O(d)***

Complexity

```
val toVisit = mutable.Stack[ImmutableTree[T]]
pushLeft(root)
```

What is our worst-case runtime to initialize the iterator? ***O(d)***

(we may have to push as many as ***d*** *nodes onto the stack)*

Complexity

```
def next: T = {
  val nextNode = toVisit.pop
  pushLeft(nextNode.right)
  return nextNode.value
}
```

What is our worst-case runtime to call ***next****?*

Complexity

```
def next: T = {
  val nextNode = toVisit.pop
  pushLeft(nextNode.right)
  return nextNode.value
}
```

What is our worst-case runtime to call **next**? ***O*(*d*)**

(we may have to push as many as ***d*** *nodes onto the stack)*

Complexity

*What is the worst-case complexity to visit ALL **n** nodes?*

Complexity

*What is the worst-case complexity to visit ALL **n** nodes?*

Each node is at the top of the stack exactly once:

Complexity

What is the worst-case complexity to visit ALL ***n*** *nodes?*

Each node is at the top of the stack exactly once:

- One push ***O*****(1)**

Complexity

What is the worst-case complexity to visit ALL ***n*** *nodes?*

Each node is at the top of the stack exactly once:

- One push ***O*****(1)**
- One pop ***O*****(1)**

Complexity

*What is the worst-case complexity to visit ALL **n** nodes?*

Each node is at the top of the stack exactly once:

- One push $O(1)$
- One pop $O(1)$

Total: $O(n)$

Balancing Trees

BST Operations

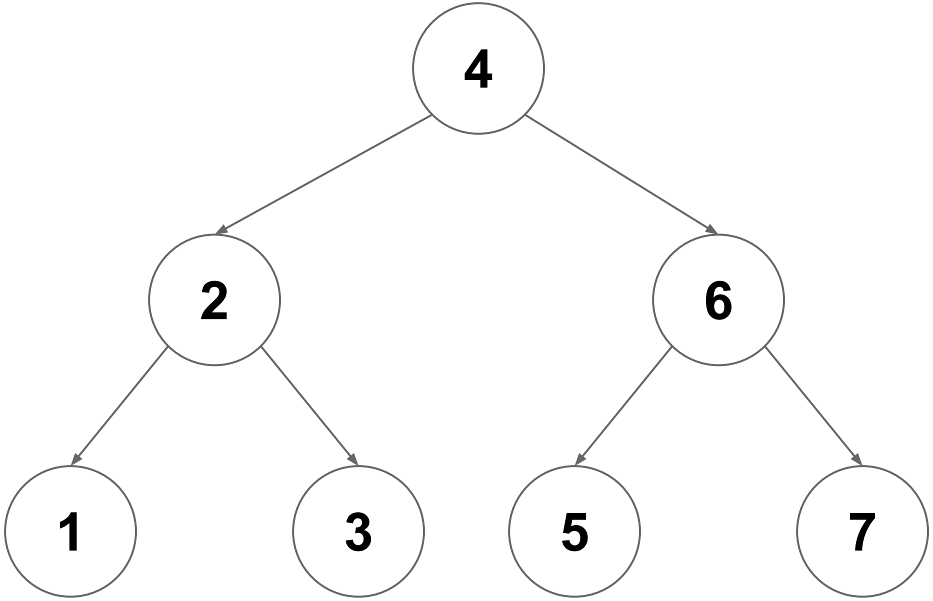
Operation	Runtime
`find`	$O(d)$
`insert`	$O(d)$
`remove`	$O(d)$

What is the runtime in terms of $\boldsymbol{n}$*?* $\boldsymbol{O(n)}$

$$\log(n) \leq d \leq n$$

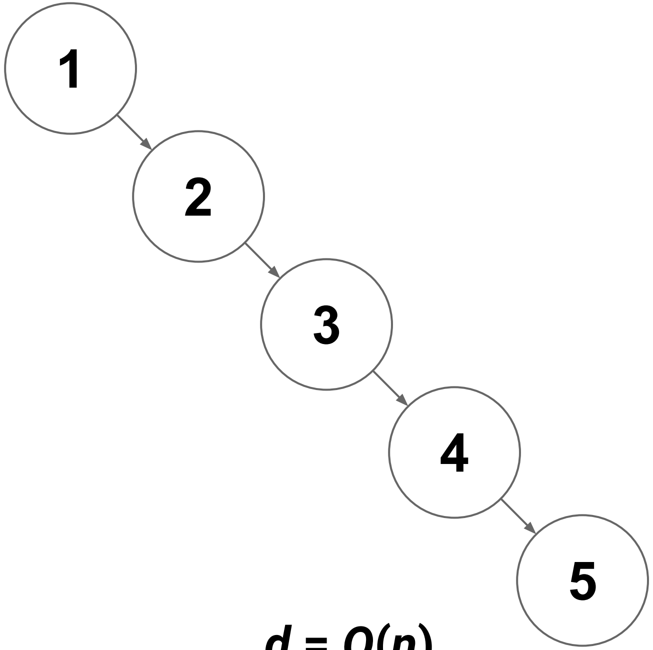
Tree Depth vs Size

If height(left) ≈ height(right)



***d* = *O*(log(*n*))**

If height(left) ≪ height(right)



***d* = *O*(*n*)**

Balanced Trees

Balanced Trees are good: Faster `find`, `insert`, `remove`

Balanced Trees

Balanced Trees are good: Faster **`find`**, **`insert`**, **`remove`**

What do we mean by balanced?

Balanced Trees

Balanced Trees are good: Faster `find`, `insert`, `remove`

What do we mean by balanced? **|height(left) - height(right)| ≤ 1**

Balanced Trees

Balanced Trees are good: Faster `find`, `insert`, `remove`

What do we mean by balanced? **|height(left) - height(right)| ≤ 1**

How do we keep a tree balanced?

Balanced Trees - Two Approaches

Option 1

Keep left/right subtrees within **+/-1** of each other in height

(add a field to track amount of "imbalance")

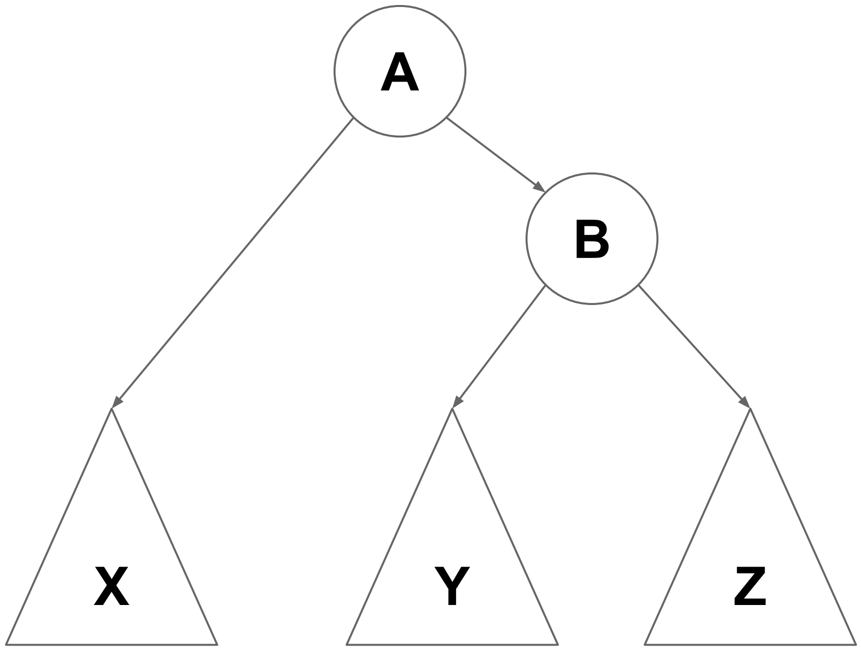
Option 2

Keep leaves at some minimum depth (***d*/2**)

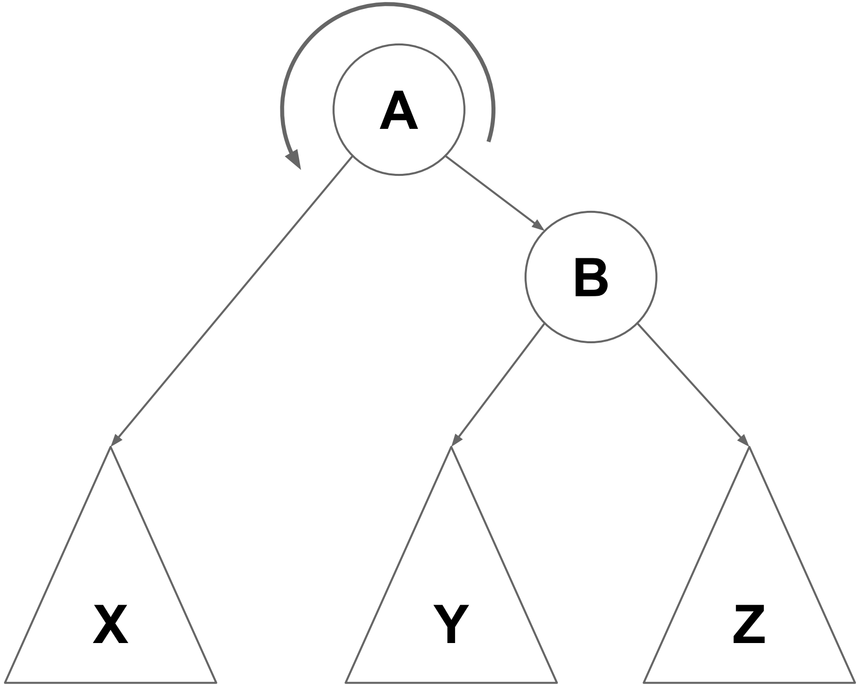
(Add a color to each node marking it as "red" or "black")

Ok…but how do we enforce this…?

Rebalancing Trees (rotations)

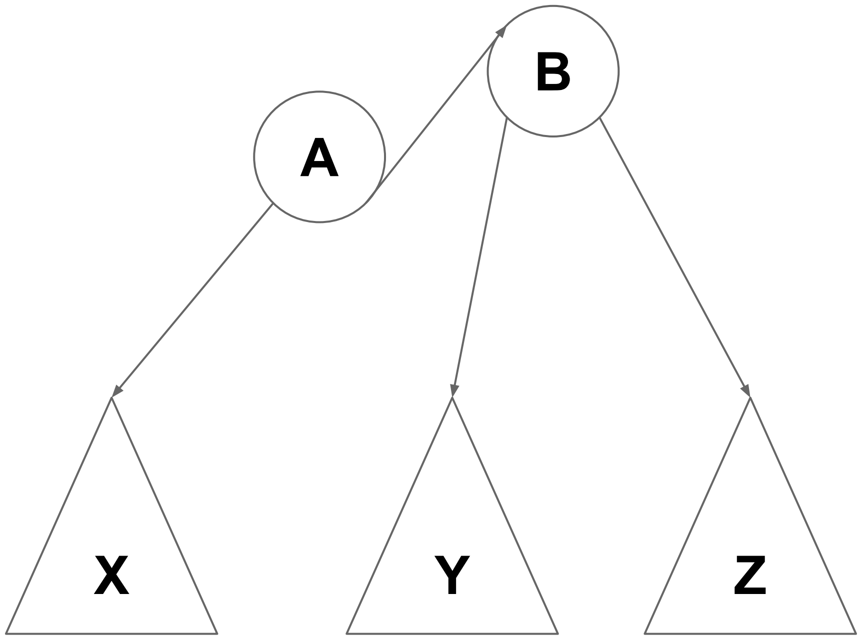


Rebalancing Trees (rotations)



Rotate(A, B)

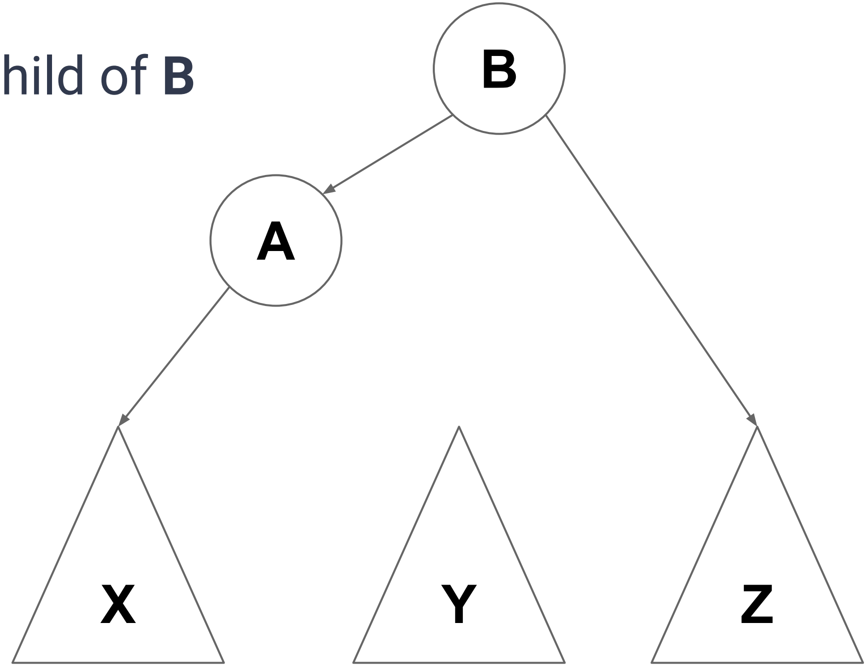
Rebalancing Trees (rotations)



Rotate(A, B)

Rebalancing Trees (rotations)

Make **A** the left child of **B**

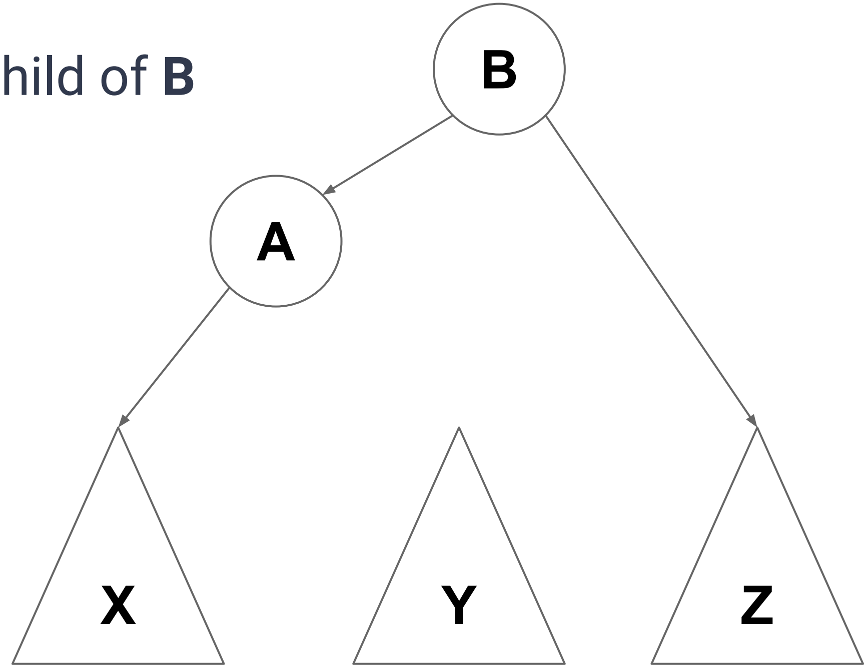


Rotate(A, B)

Rebalancing Trees (rotations)

Make **A** the left child of **B**

What about **Y**?



Rotate(A, B)

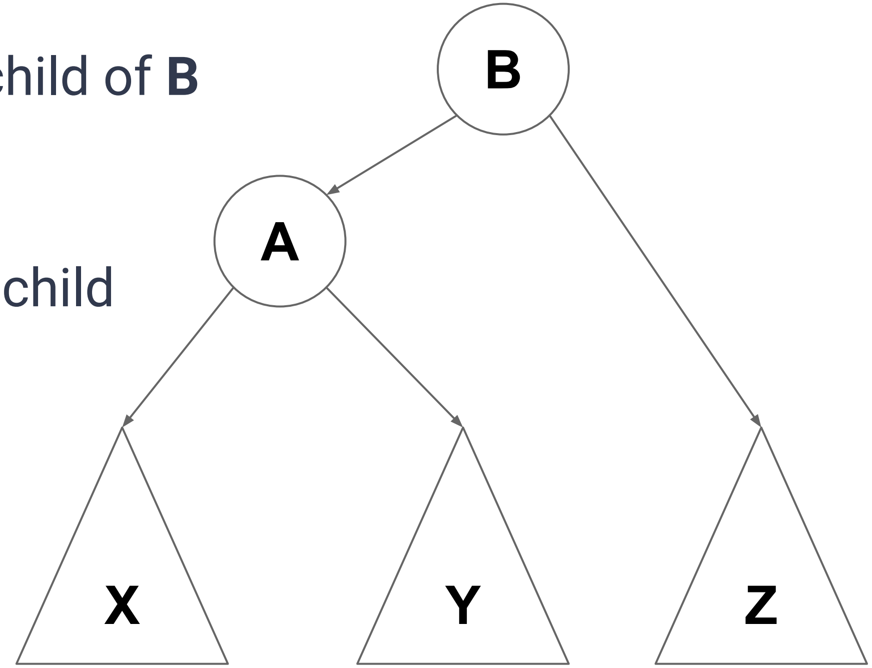
Rebalancing Trees (rotations)

Make **A** the left child of **B**

What about **Y**?

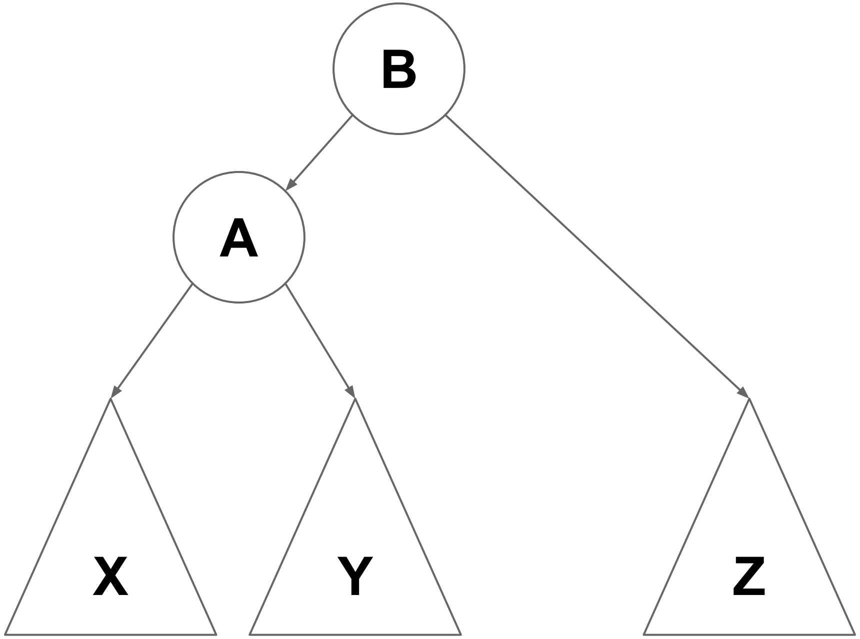
Make it the right child

of **A**



Rotate(A, B)

Rebalancing Trees (rotations)

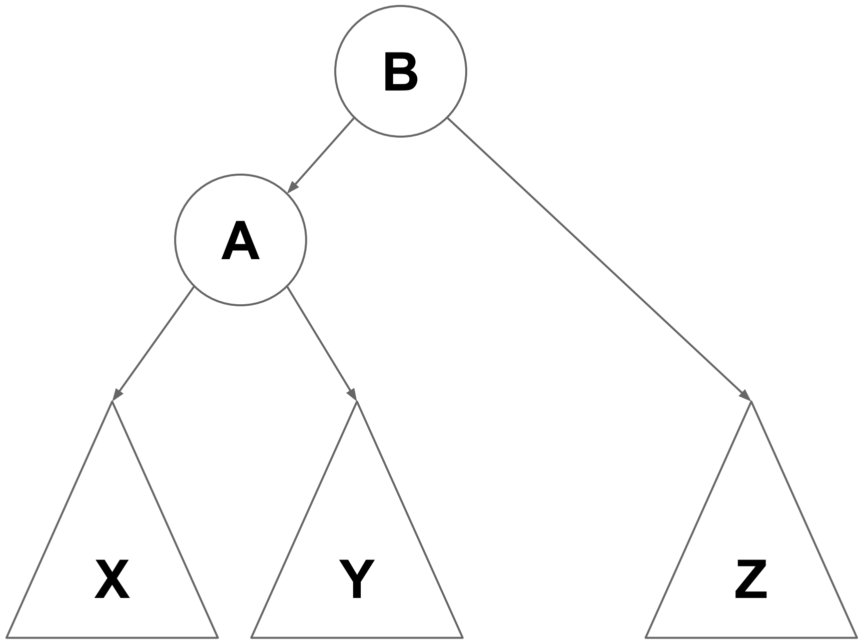


Rotate(A, B)

Rebalancing Trees (rotations)

A became **B**'s left child

B's left child became **A**'s right child



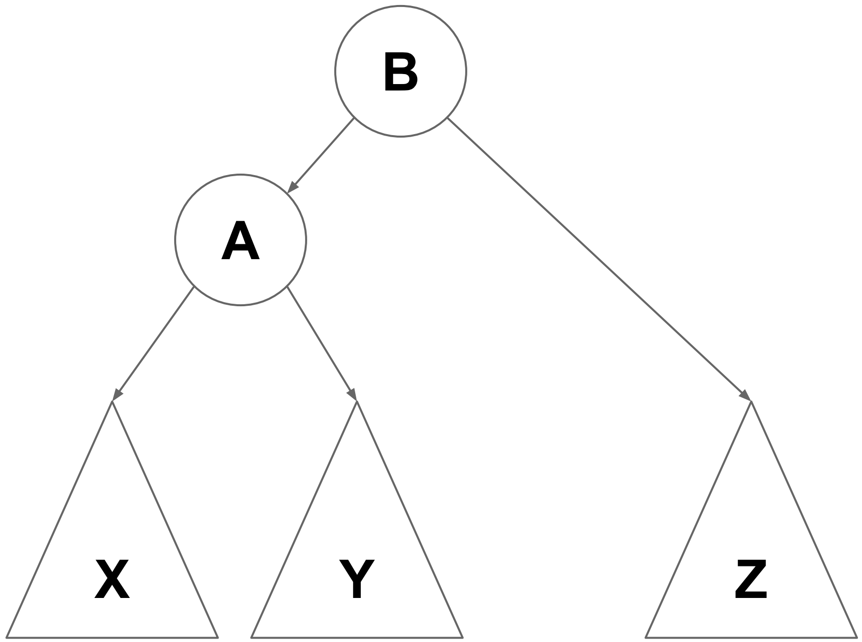
Rotate(A, B)

Rebalancing Trees (rotations)

A became **B**'s left child

B's left child became **A**'s right child

Is ordering maintained?



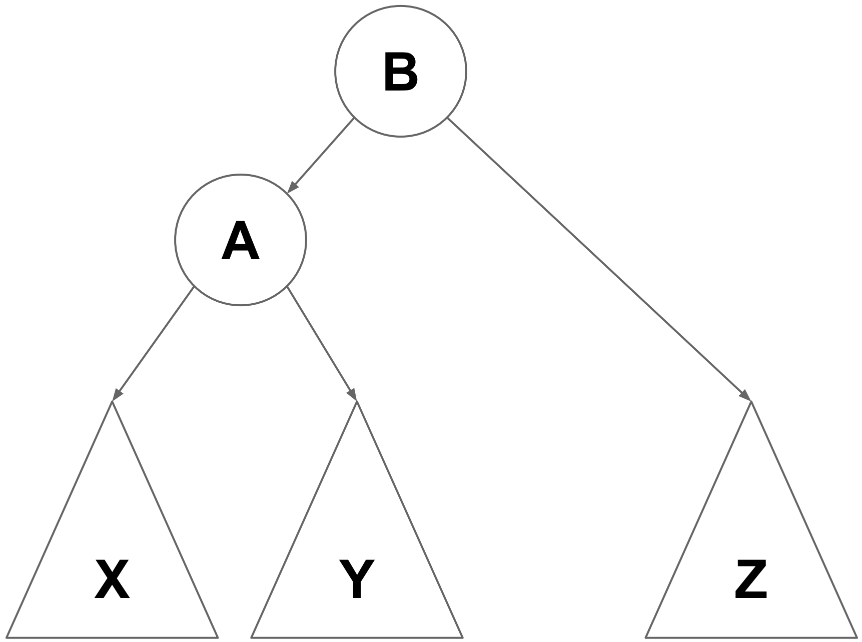
Rotate(A, B)

Rebalancing Trees (rotations)

A became **B**'s left child

B's left child became **A**'s right child

Is ordering maintained? **Yes!**



Rotate(A, B)

Rebalancing Trees (rotations)

A became **B**'s left child

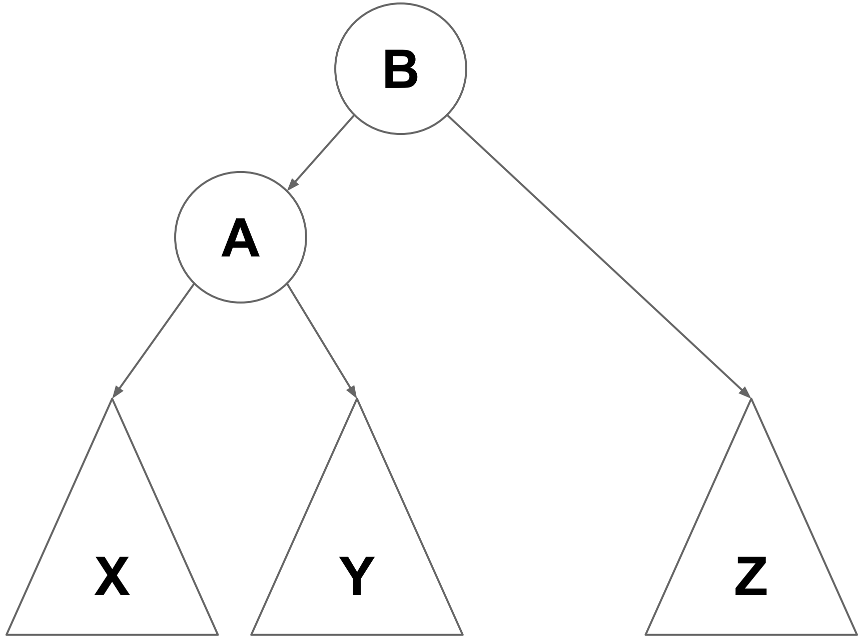
B's left child became **A**'s right child

Is ordering maintained? **Yes!**

B used to be the right child of **A**

Therefore **B** is bigger than **A**

Therefore **A** is smaller than **B** ✓



Rotate(A, B)

Rebalancing Trees (rotations)

A became **B**'s left child

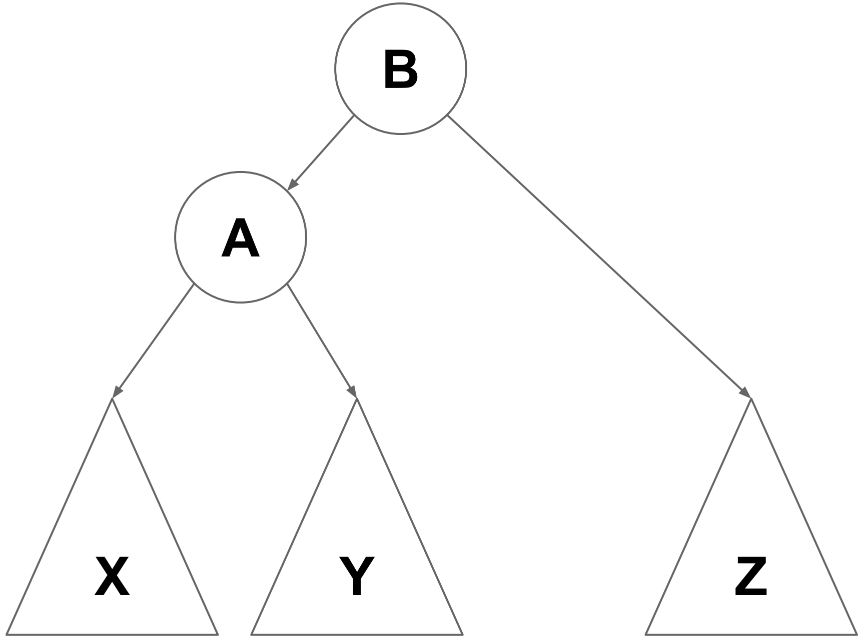
B's left child became **A**'s right child

Is ordering maintained? **Yes!**

Y used to be in the left subtree of **B**

Therefore **Y** is smaller than **B**

It is still left of **B** ✓



Rotate(A, B)

Rebalancing Trees (rotations)

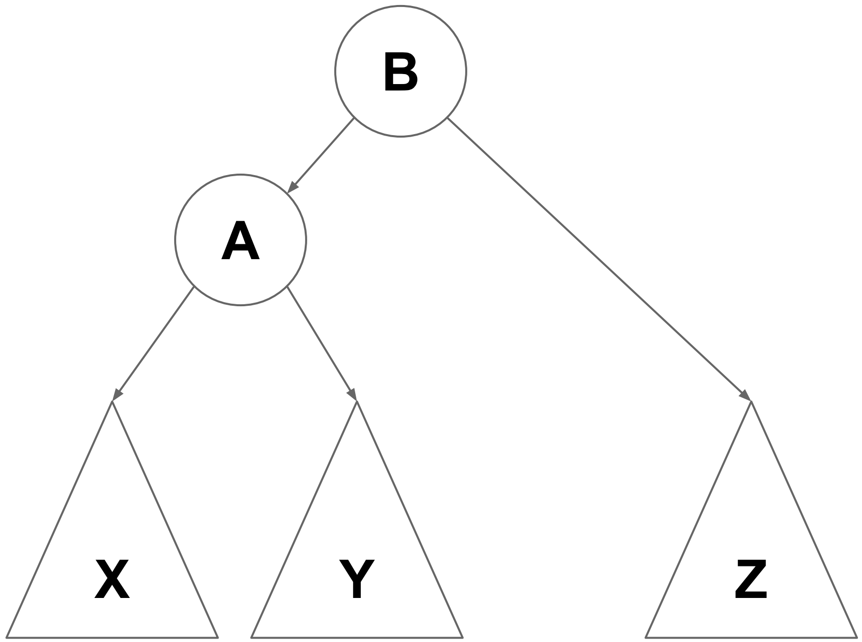
A became **B**'s left child

B's left child became **A**'s right child

Is ordering maintained? **Yes!**

Y used to be in the right subtree of **A**

It is still in the right subtree of **A** ✓



Rotate(A, B)

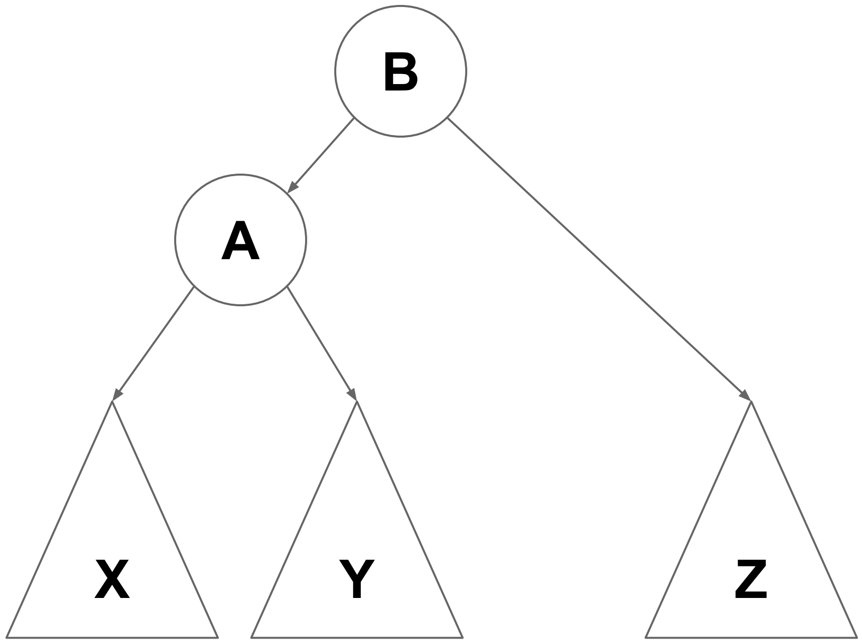
Rebalancing Trees (rotations)

A became **B**'s left child

B's left child became **A**'s right child

Is ordering maintained? **Yes!**

Complexity?



Rotate(A, B)

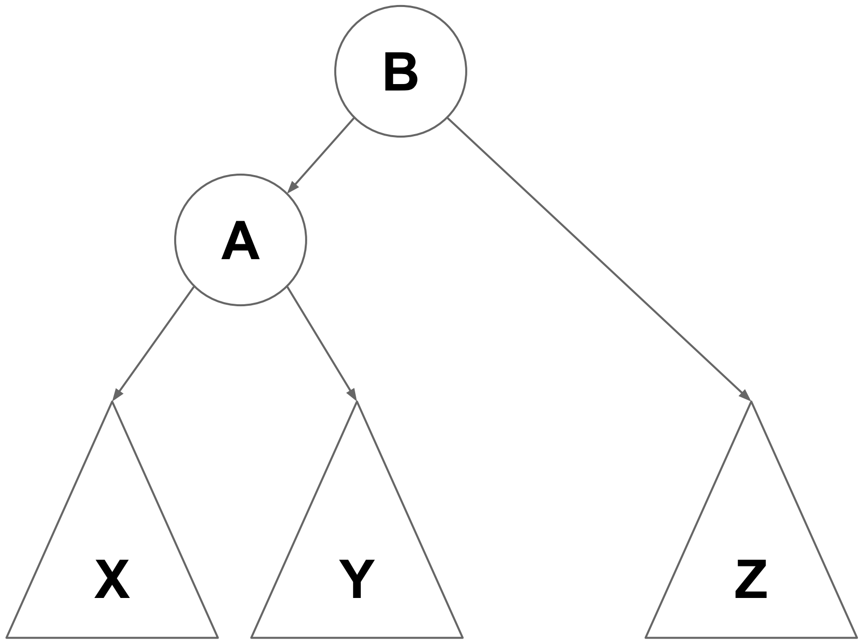
Rebalancing Trees (rotations)

A became **B**'s left child

B's left child became **A**'s right child

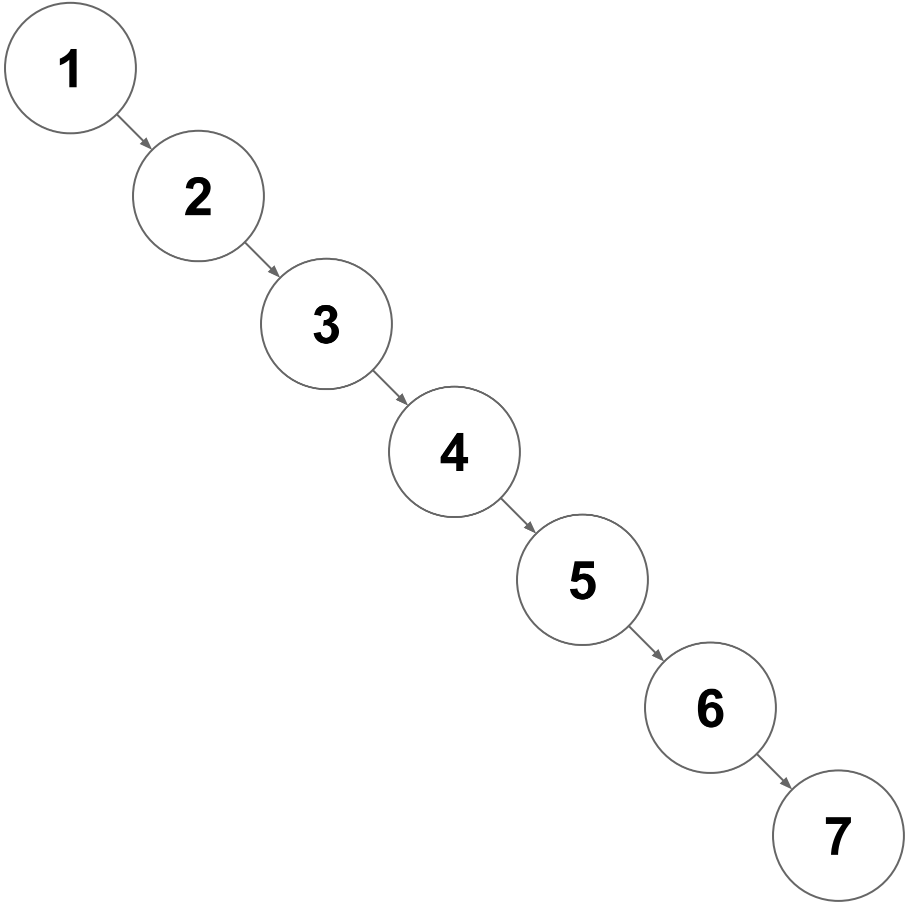
Is ordering maintained? **Yes!**

Complexity? ***O*(1)**



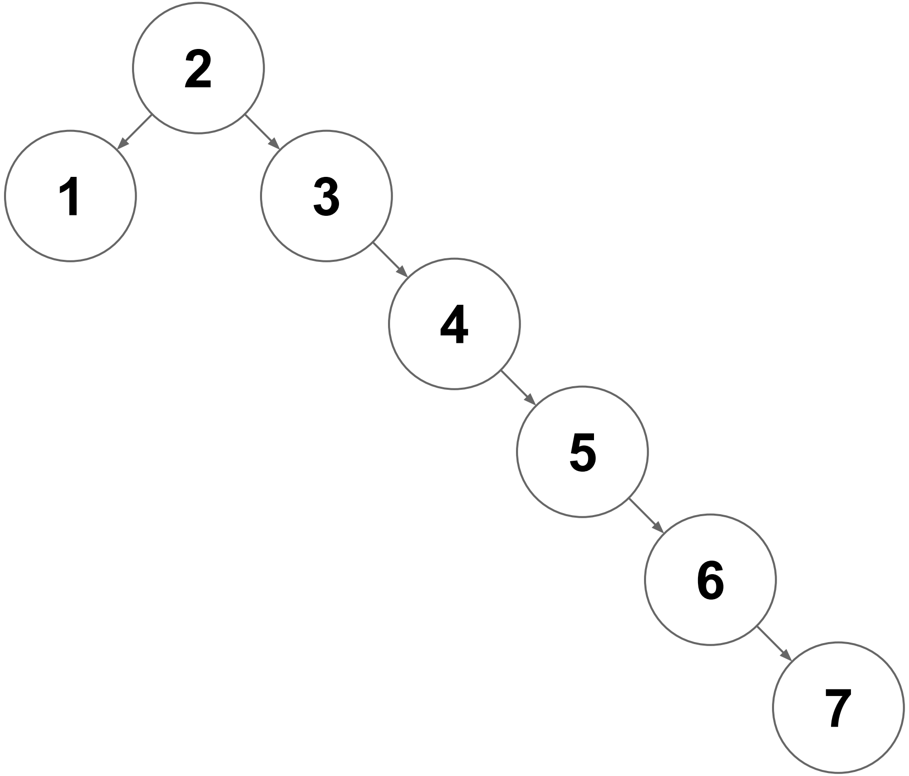
Rotate(A, B)

Rebalancing Trees



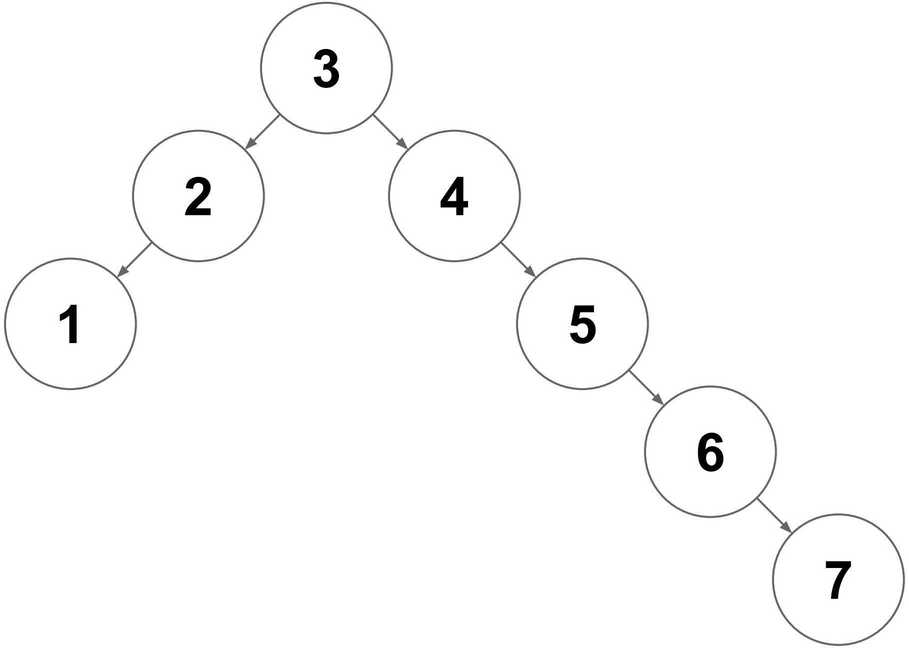
Rebalancing Trees

`Rotate(1,2)`



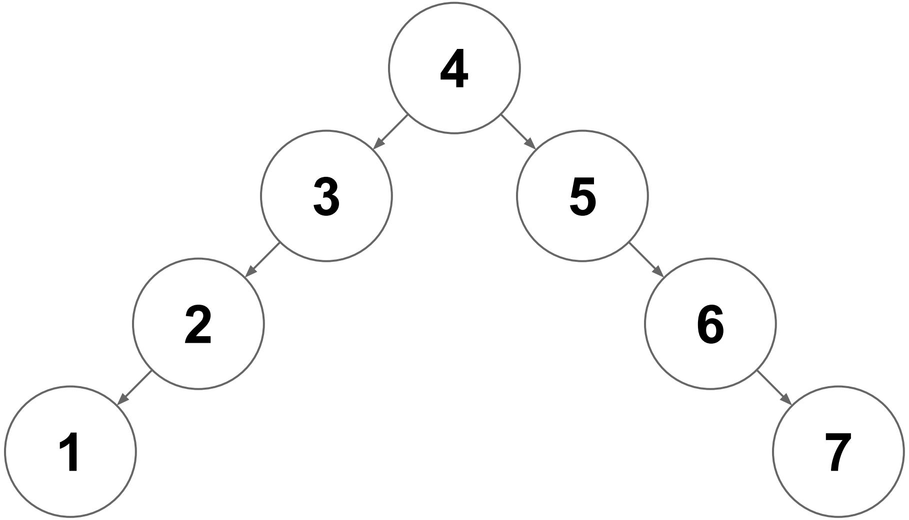
Rebalancing Trees

`Rotate(2,3)`



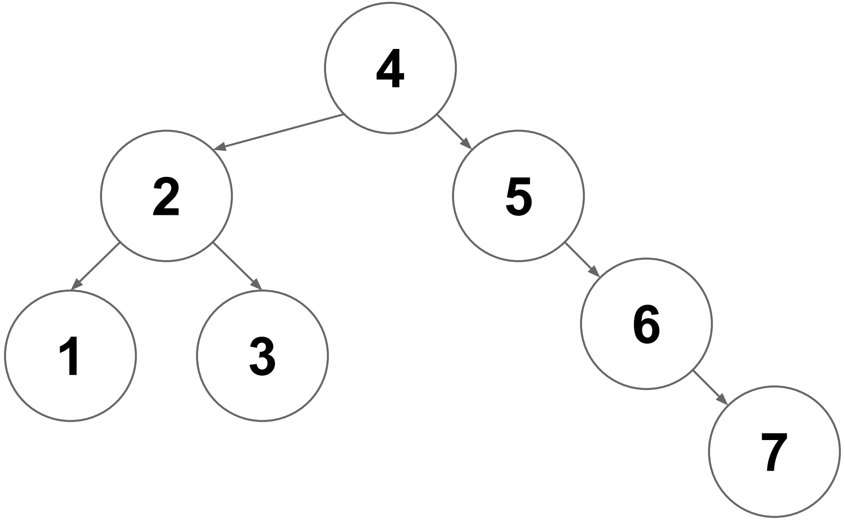
Rebalancing Trees

`Rotate(3,4)`



Rebalancing Trees

`Rotate(3,2)`



Rebalancing Trees

`Rotate(5,6)`