# CSE 250
## Data Structures

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

# Lec 26: More Tree Operations

# Announcements

- PA2 was due last night, submissions close Tuesday
- WA4 comes out today – Due Sunday
- Classes cancelled Monday for the Eclipse
  - Recitation next week is midterm review, no attendance required
  - If you have recitation on Monday but still want to attend, you may attend a Tuesday recitation (as long as there is space)
- TA hiring starting soon – If you want to join 250 course staff email me!

# Collection ADTs

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

# BST Operations

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

# BST Operations

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

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

*What about the lower bound?* $\Omega(\log(n))$

*Can we do better?* **TBD...**

# Collection ADTs

| Property | Seq | Set | Bag |
|---|---|---|---|
| Explicit Order | ✓ | | |
| Enforced Uniqueness | | ✓ | |
| 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

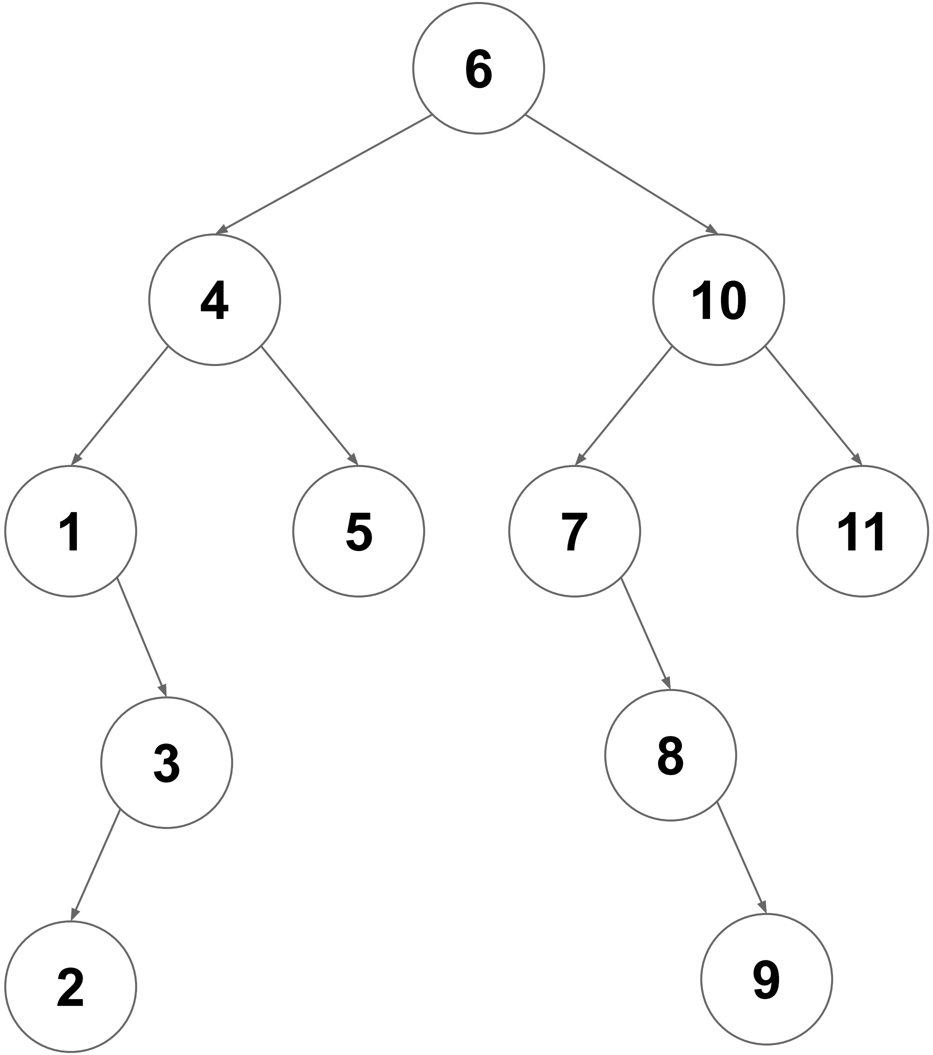
**Post-Order (bottom-up)**

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

# Tree Traversal: In-Order

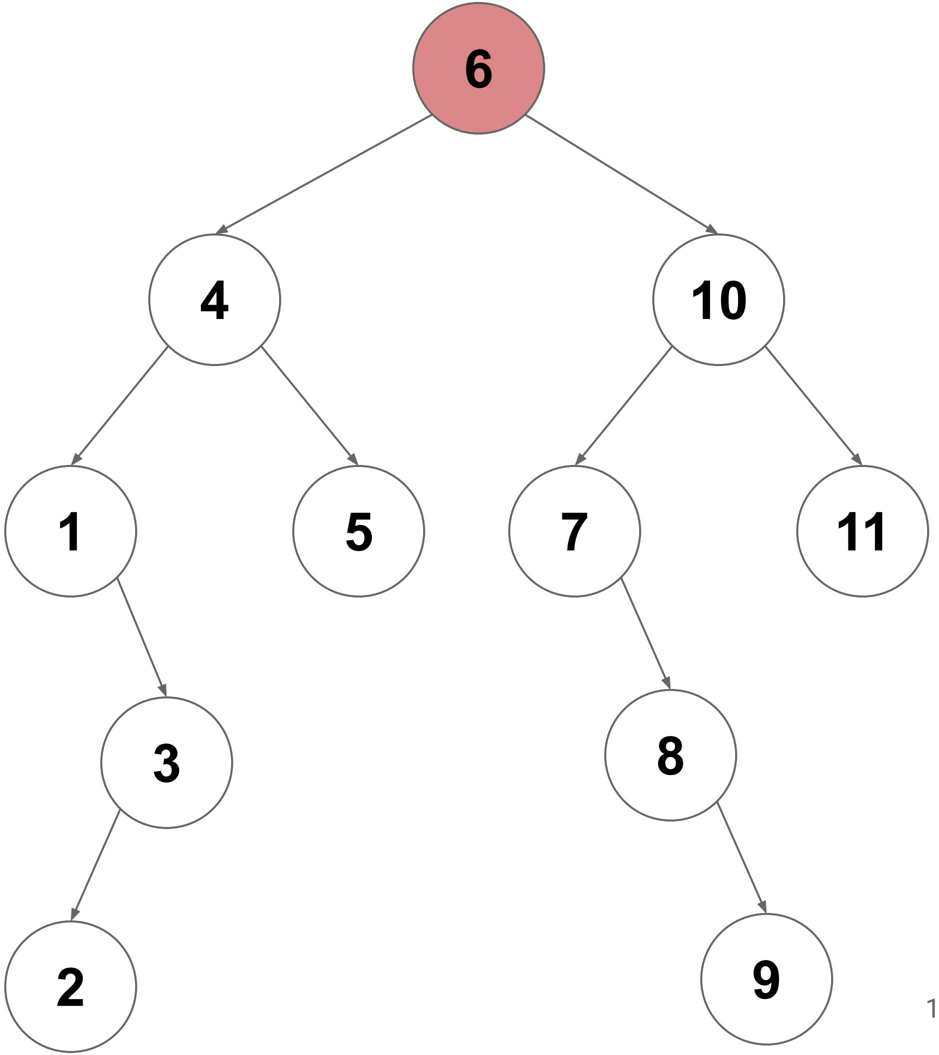
```java
void inOrderVisit(Optional<TreeNode<T>> root, Visitor v) {
  if (root.isPresent()) {
    inOrderVisit(root.get().leftChild, v);
    v.visit(root.get().value);
    inOrderVisit(root.get().rightChild, v);
  }
}
```

# In-Order Traversal on a BST



# In-Order Traversal on a BST

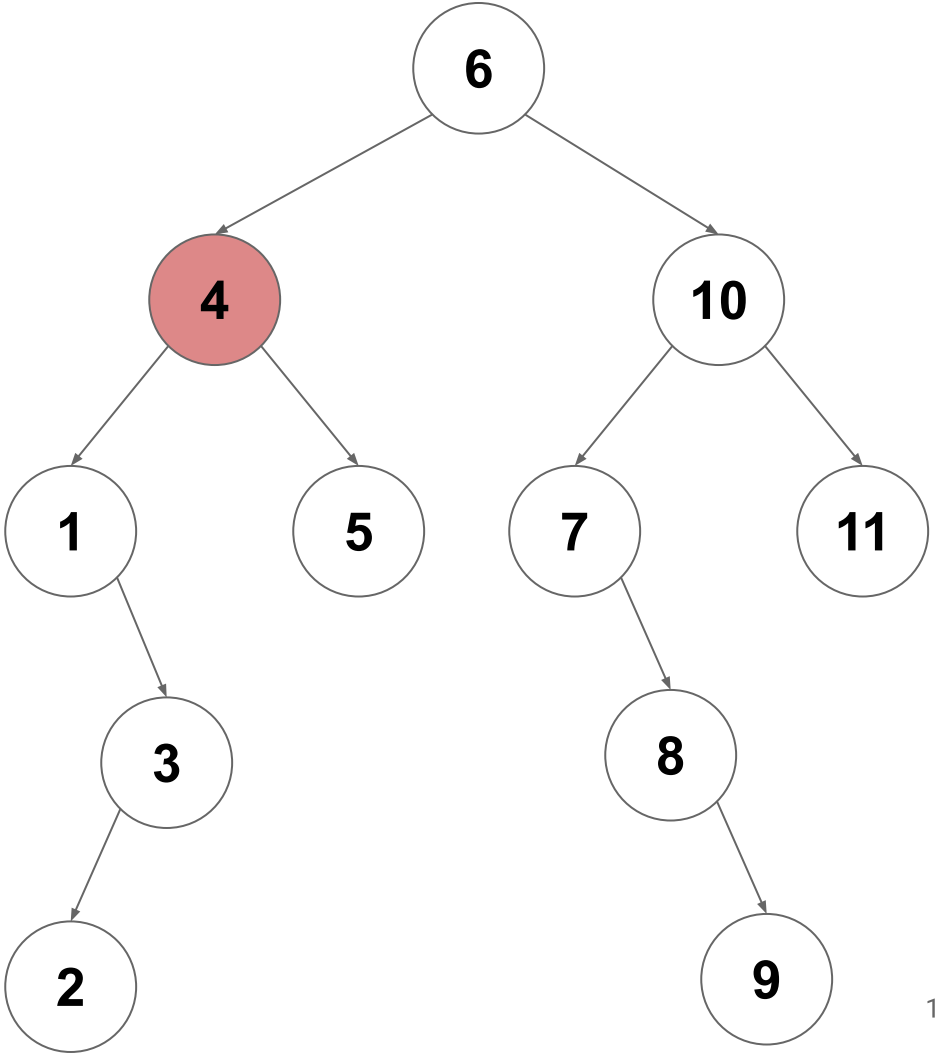
`inorderVisit(6)`



# In-Order Traversal on a BST

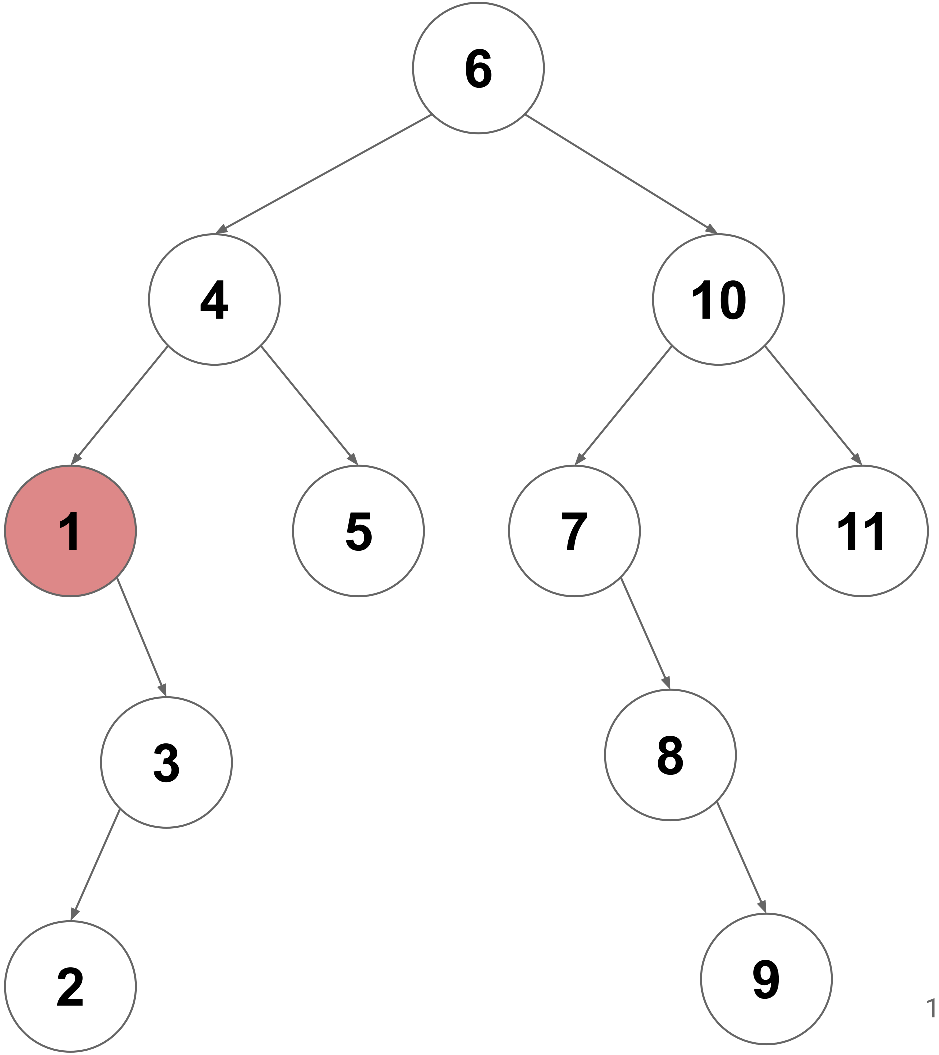
`inorderVisit(6)`

`inorderVisit(4)`



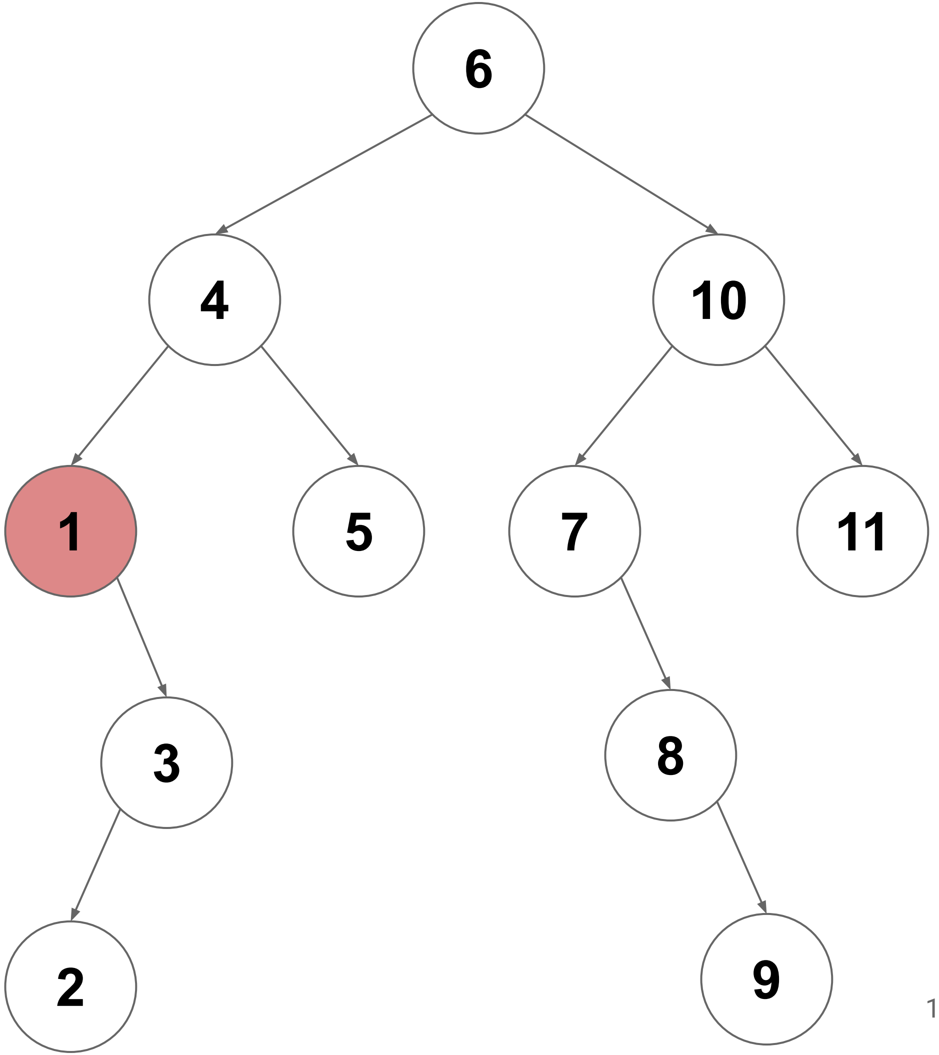
# In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(4)

inorderVisit(1)
```



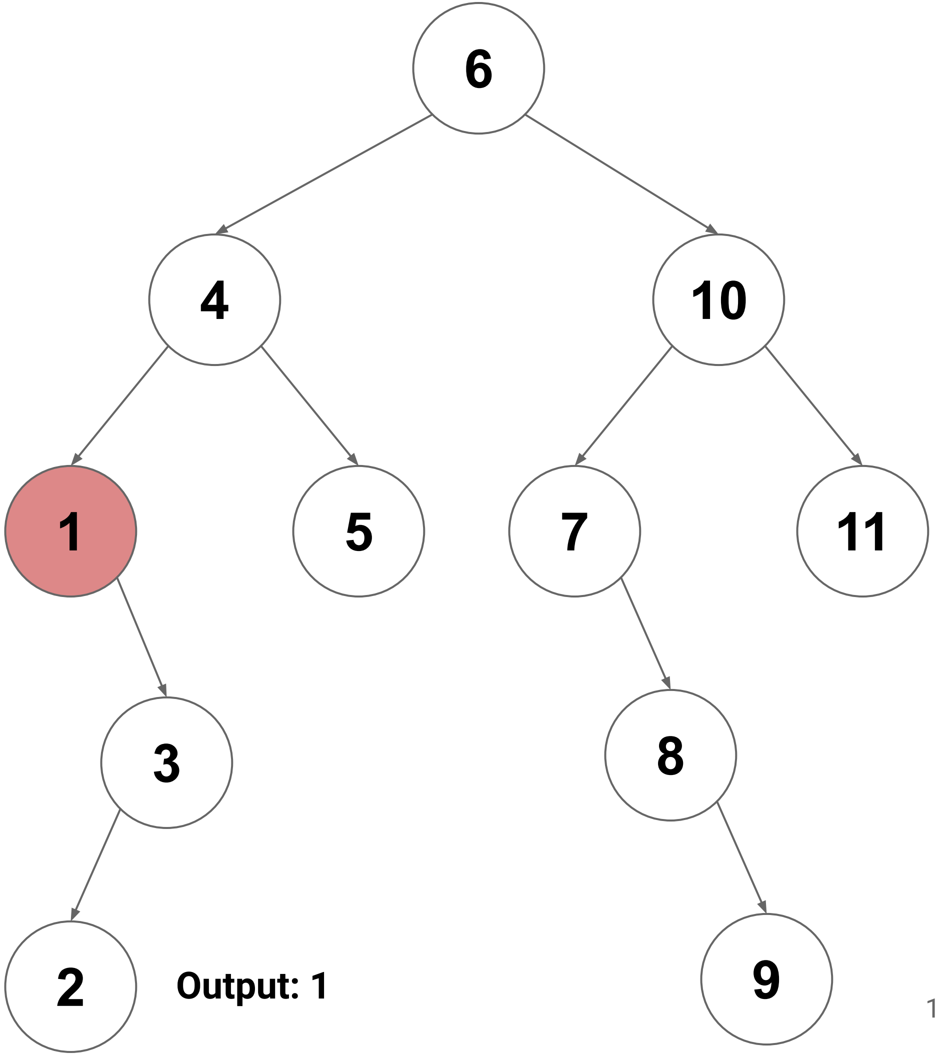
# In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(4)

inorderVisit(1)

inorderVisit(empty)
```



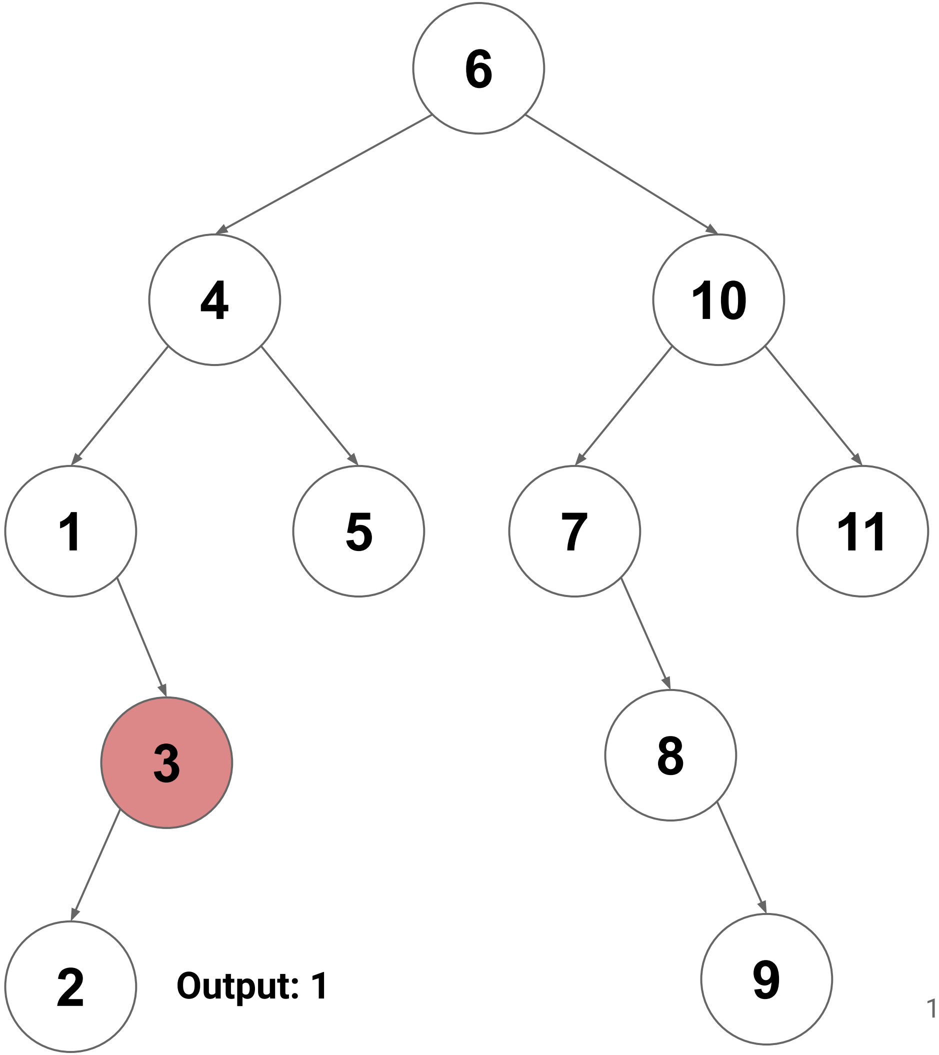
# In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(4)

inorderVisit(1)

visit(1)
```



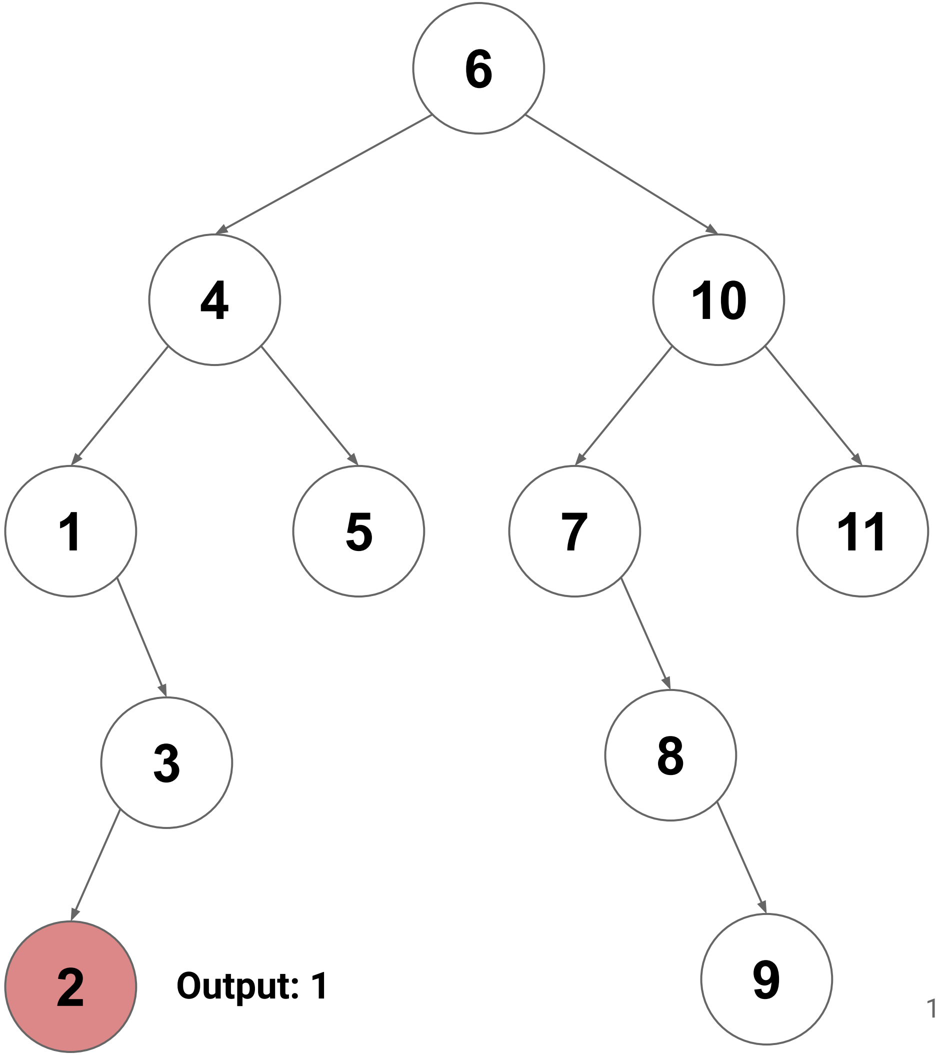
# In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(4)

inorderVisit(1)

inorderVisit(3)
```



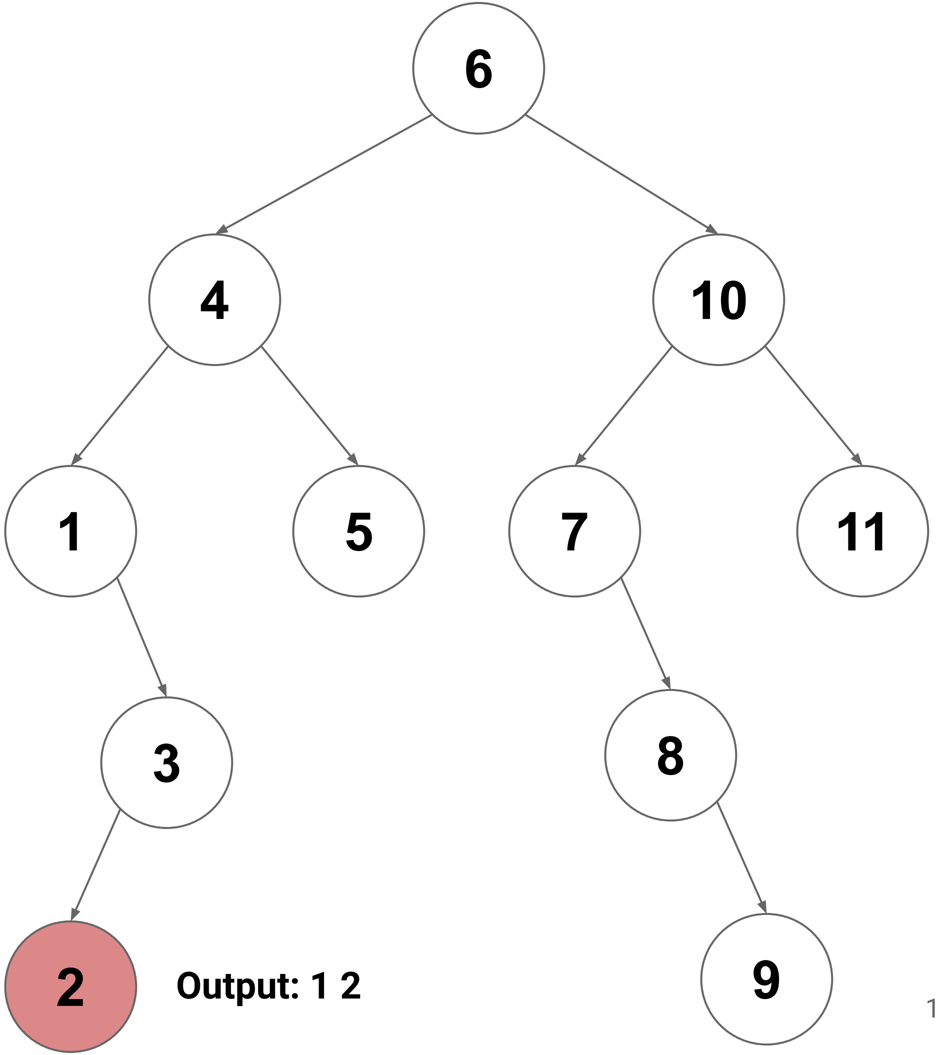
# In-Order Traversal on a BST

```
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)
```



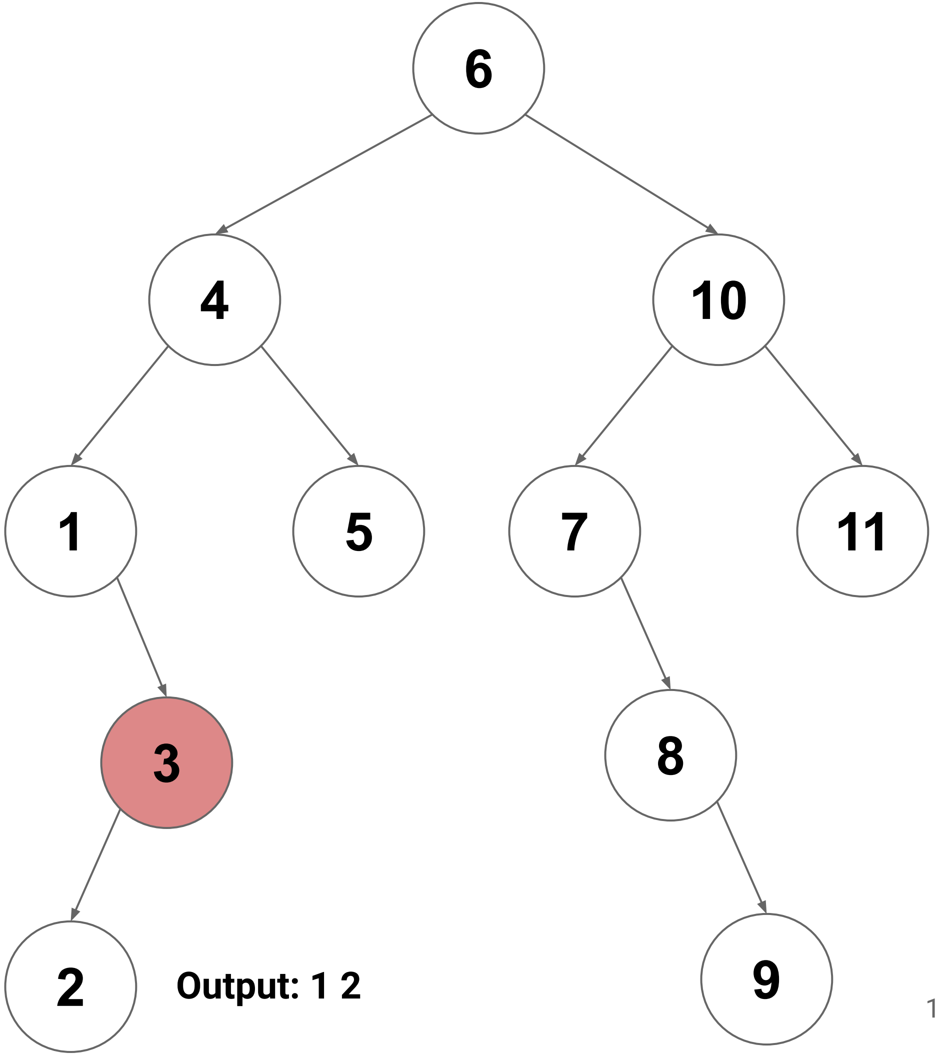
# In-Order Traversal on a BST

inorderVisit(6)

inorderVisit(4)

inorderVisit(1)

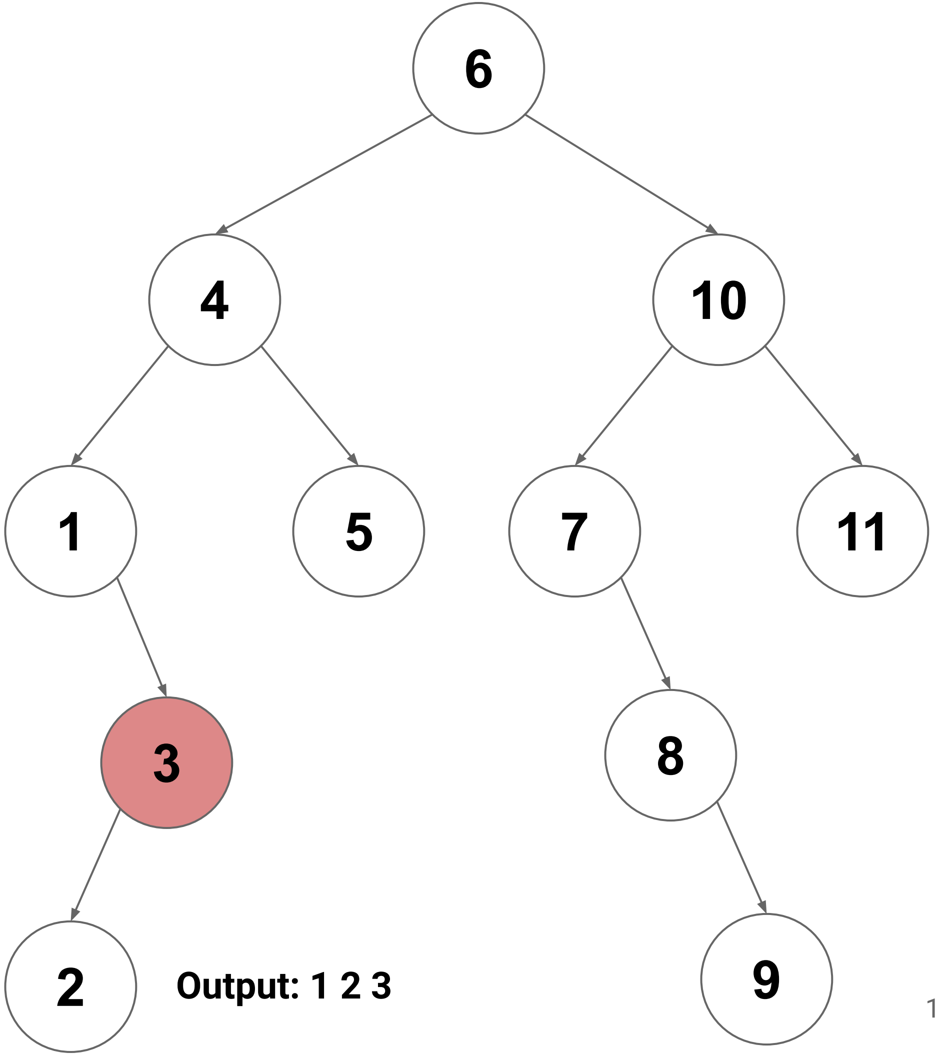
inorderVisit(3)




Output: 1 2

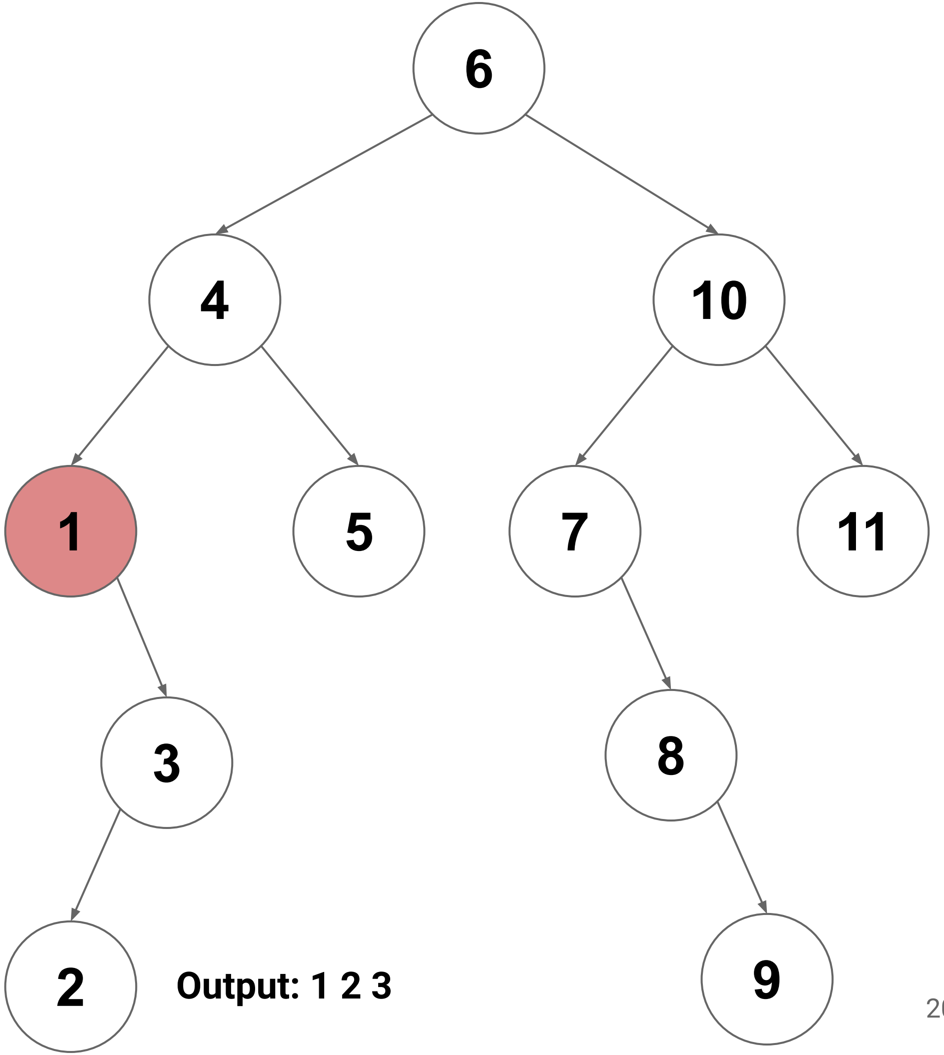
# In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(4)

inorderVisit(1)

visit(3)
```



# In-Order Traversal on a BST

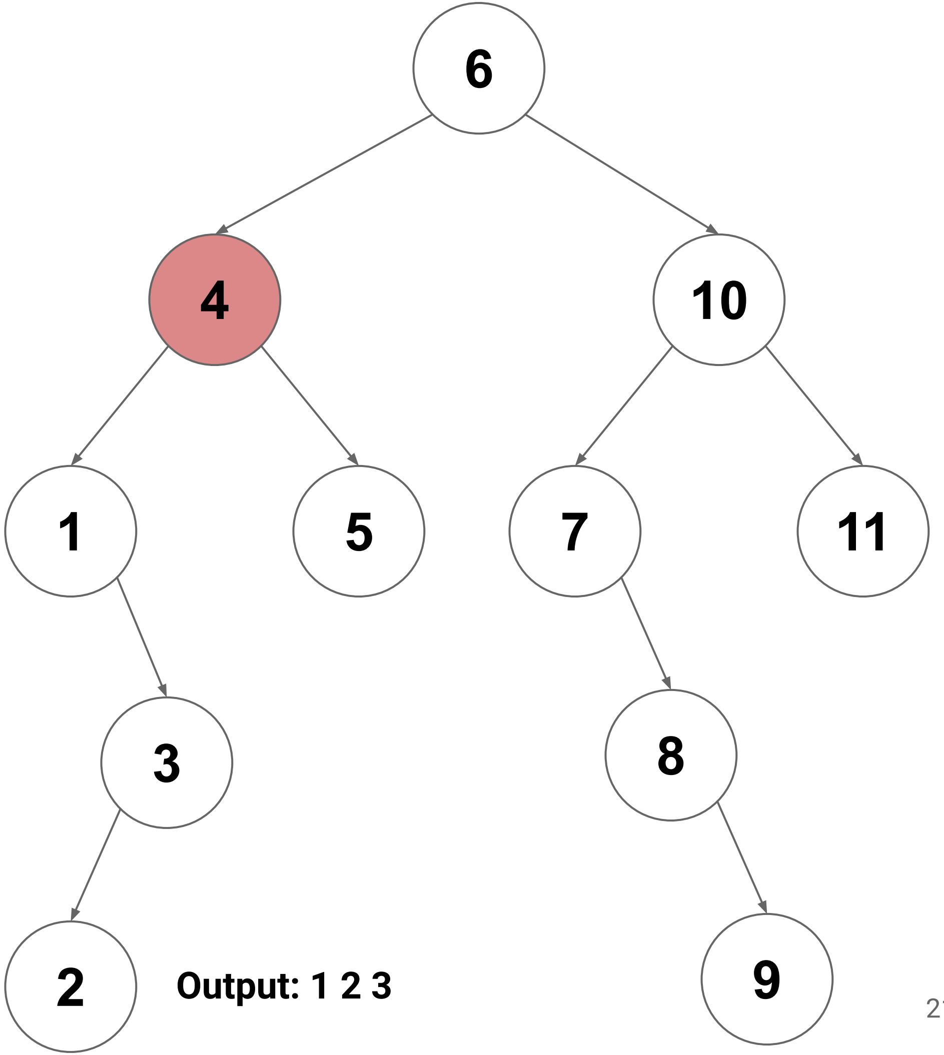
```
inorderVisit(6)

inorderVisit(4)

inorderVisit(1)
```



# In-Order Traversal on a BST

`inorderVisit(6)`

`inorderVisit(4)`



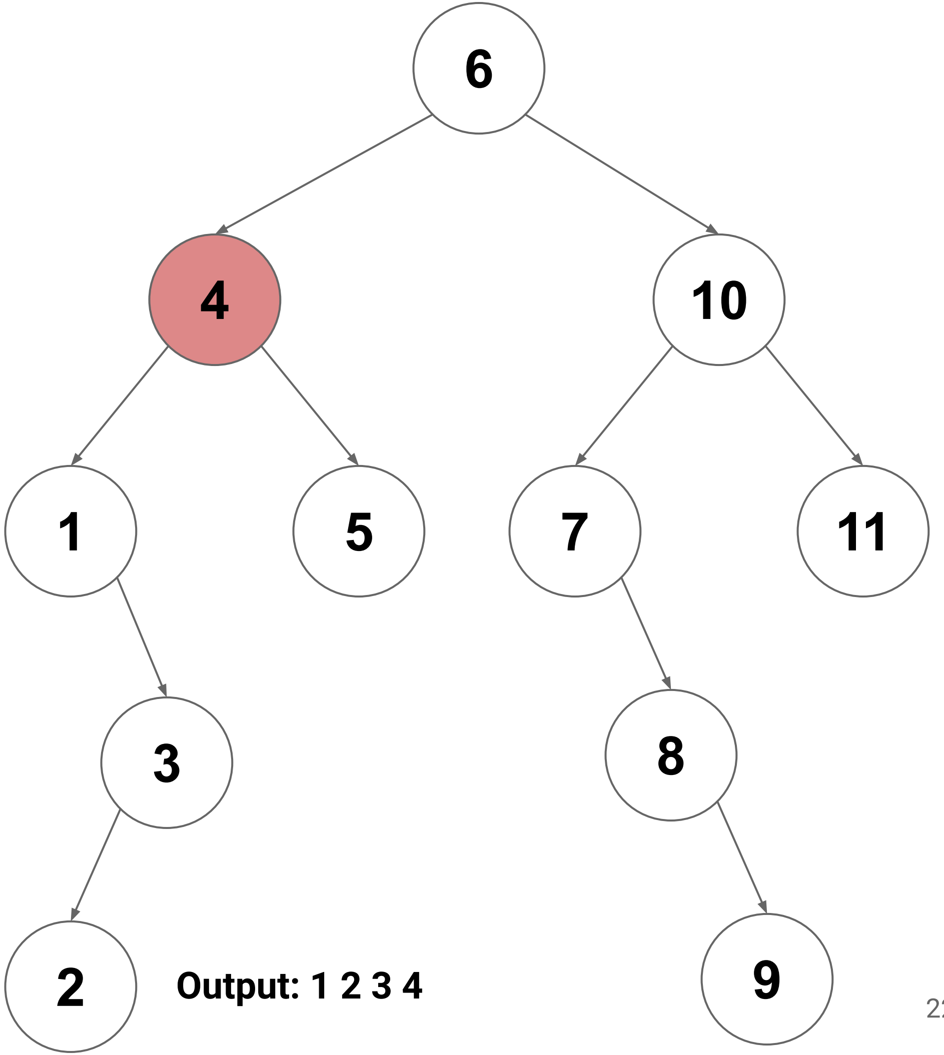

**Output: 1 2 3**

# In-Order Traversal on a BST

`inorderVisit(6)`

`inorderVisit(4)`

`visit(4)`

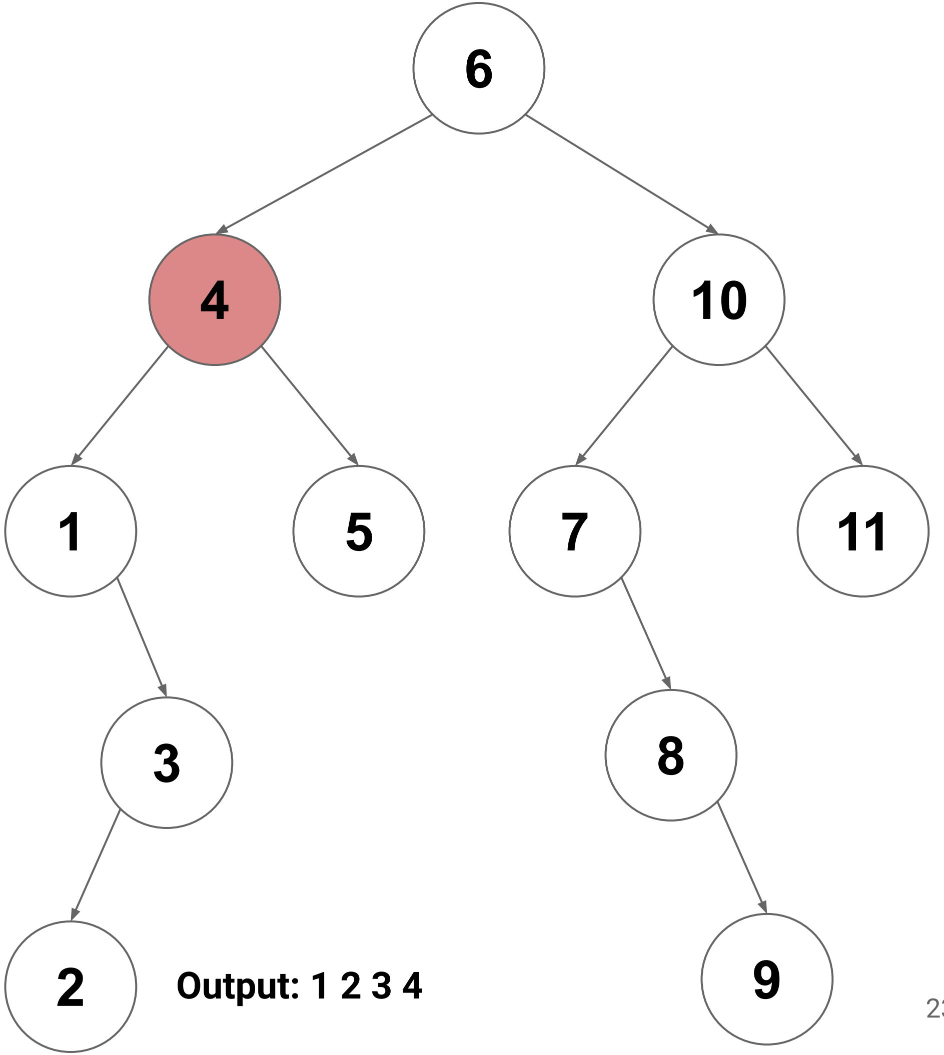



**Output: 1 2 3 4**

# In-Order Traversal on a BST

`inorderVisit(6)`

`inorderVisit(4)`



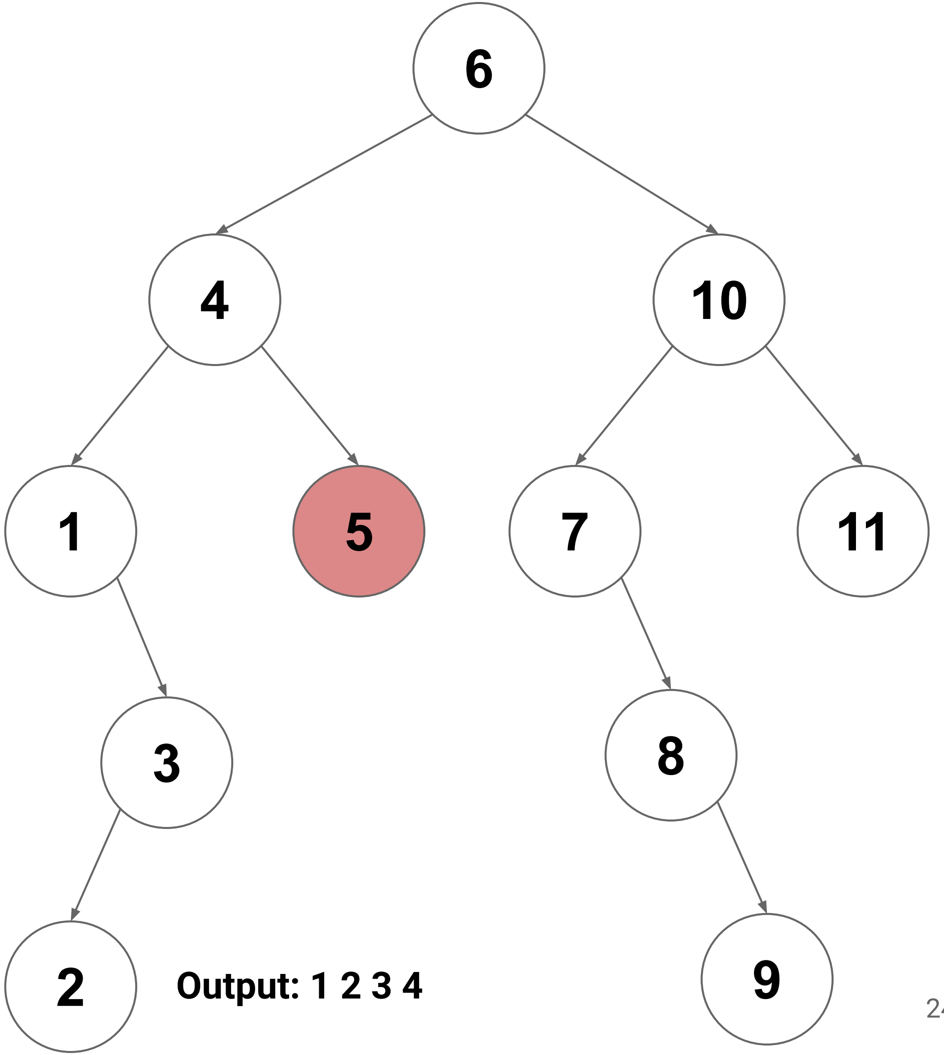

**Output: 1 2 3 4**

# In-Order Traversal on a BST

`inorderVisit(6)`

`inorderVisit(4)`

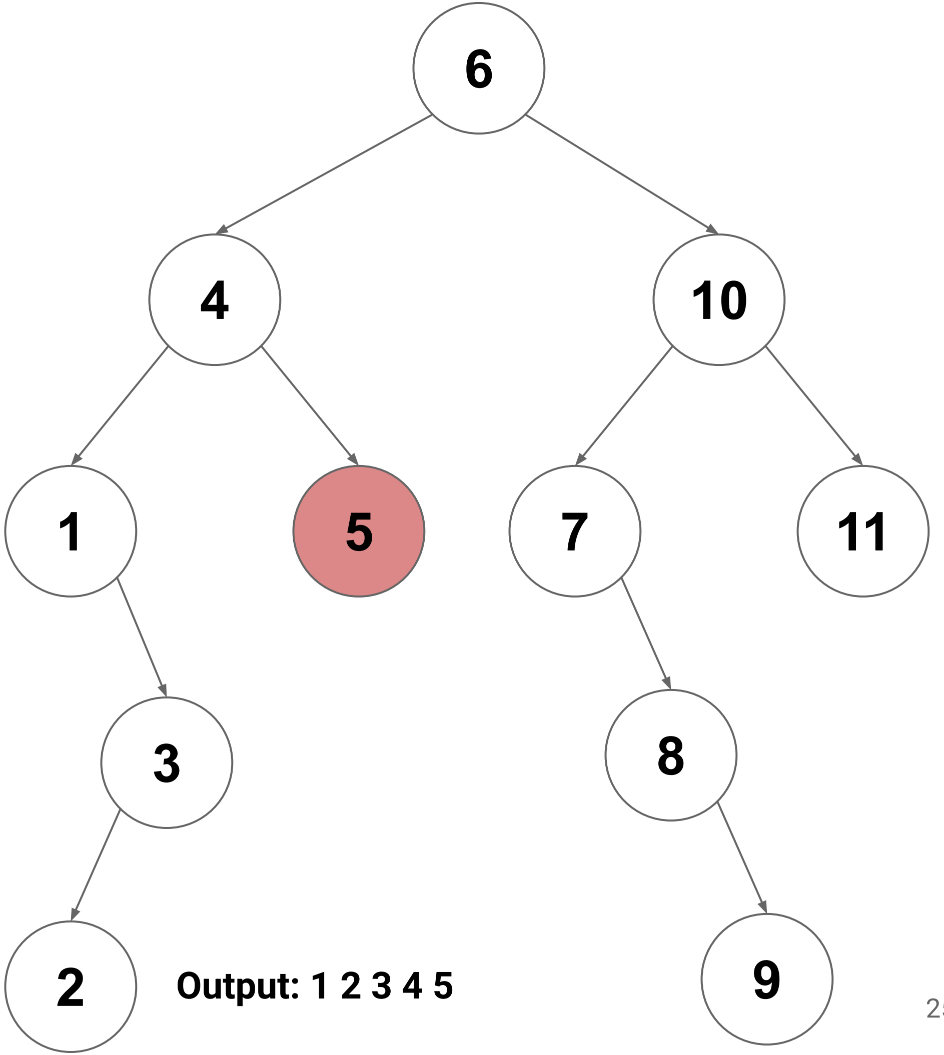
`inorderVisit(5)`




**Output: 1 2 3 4**

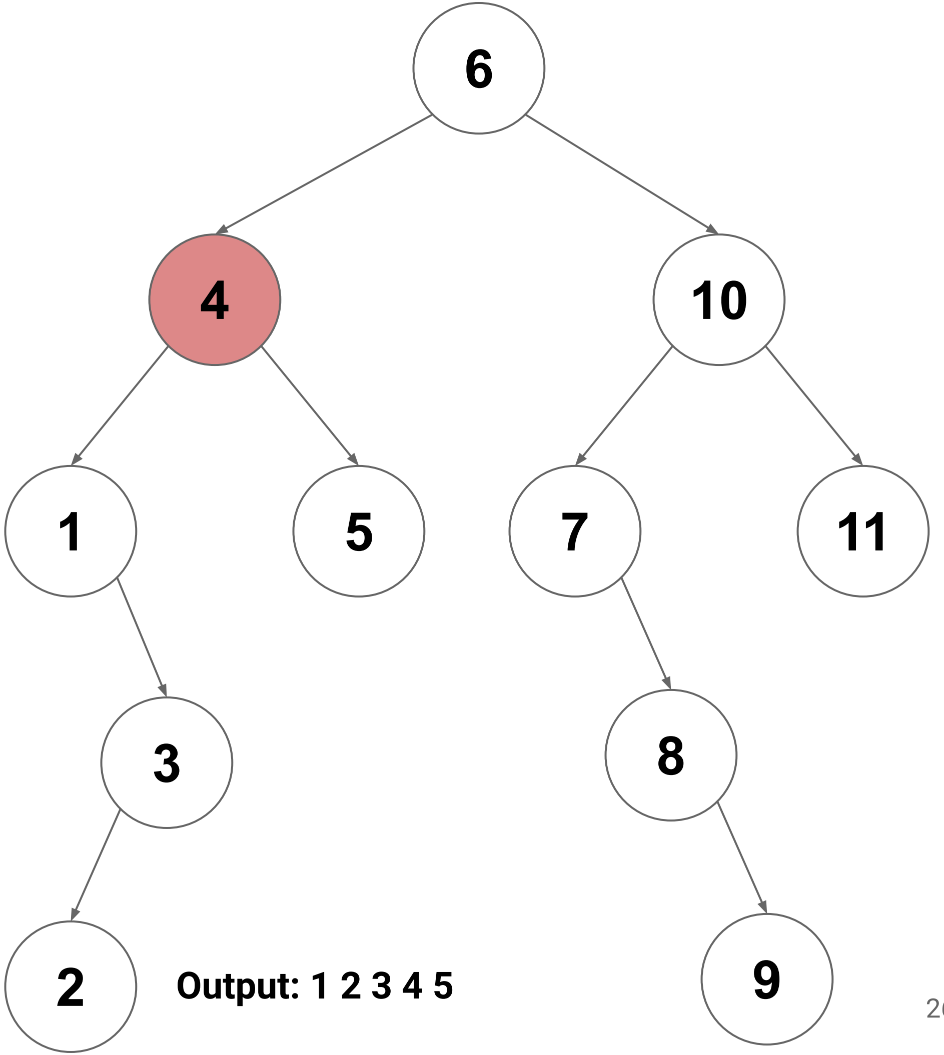
# In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(4)

visit(5)
```



# In-Order Traversal on a BST

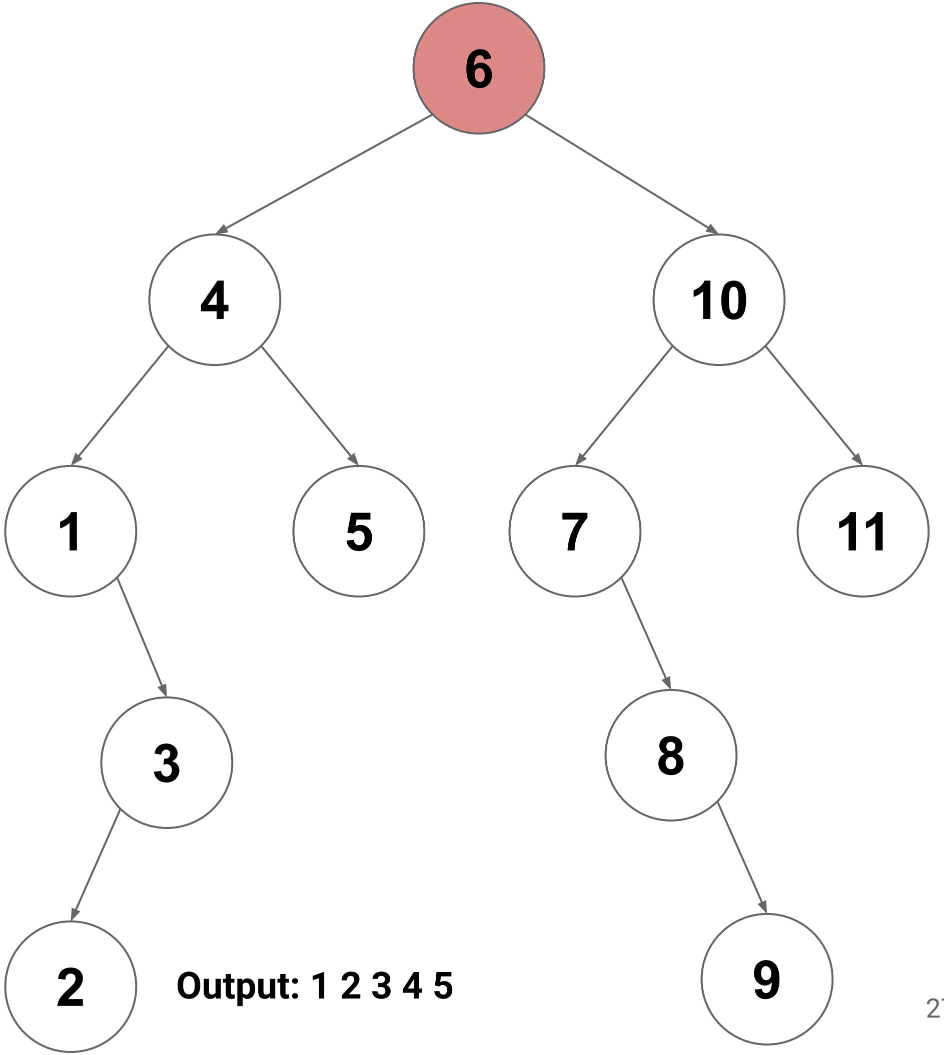
```
inorderVisit(6)

inorderVisit(4)
```




Output: 1 2 3 4 5

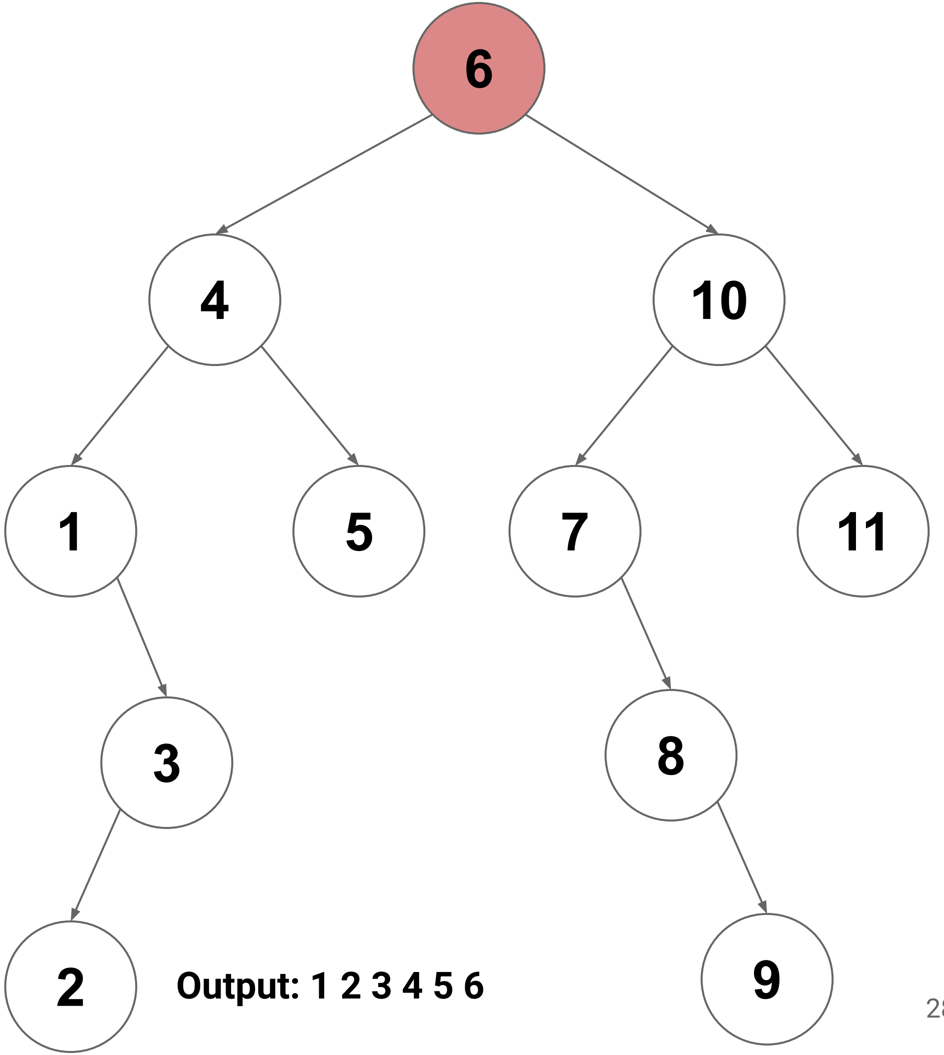
# In-Order Traversal on a BST

`inorderVisit(6)`



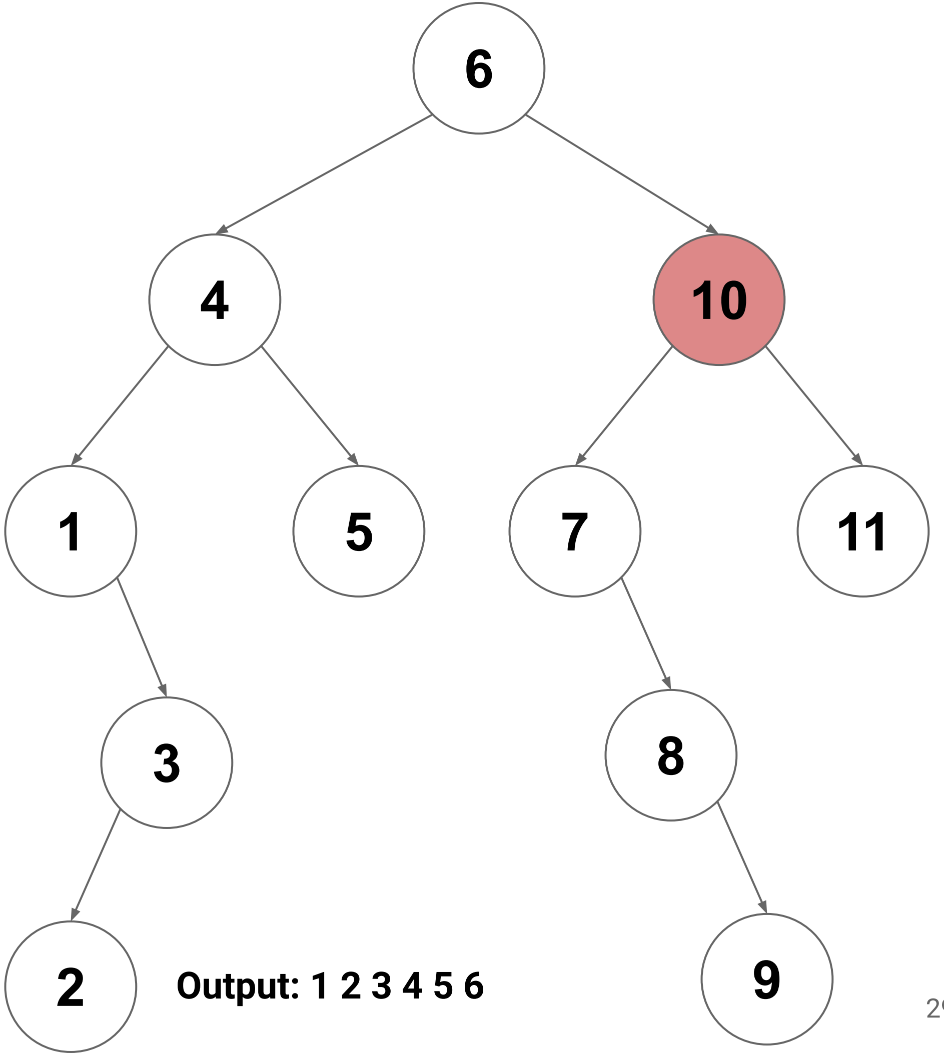
# In-Order Traversal on a BST

```
inorderVisit(6)

visit(6)
```



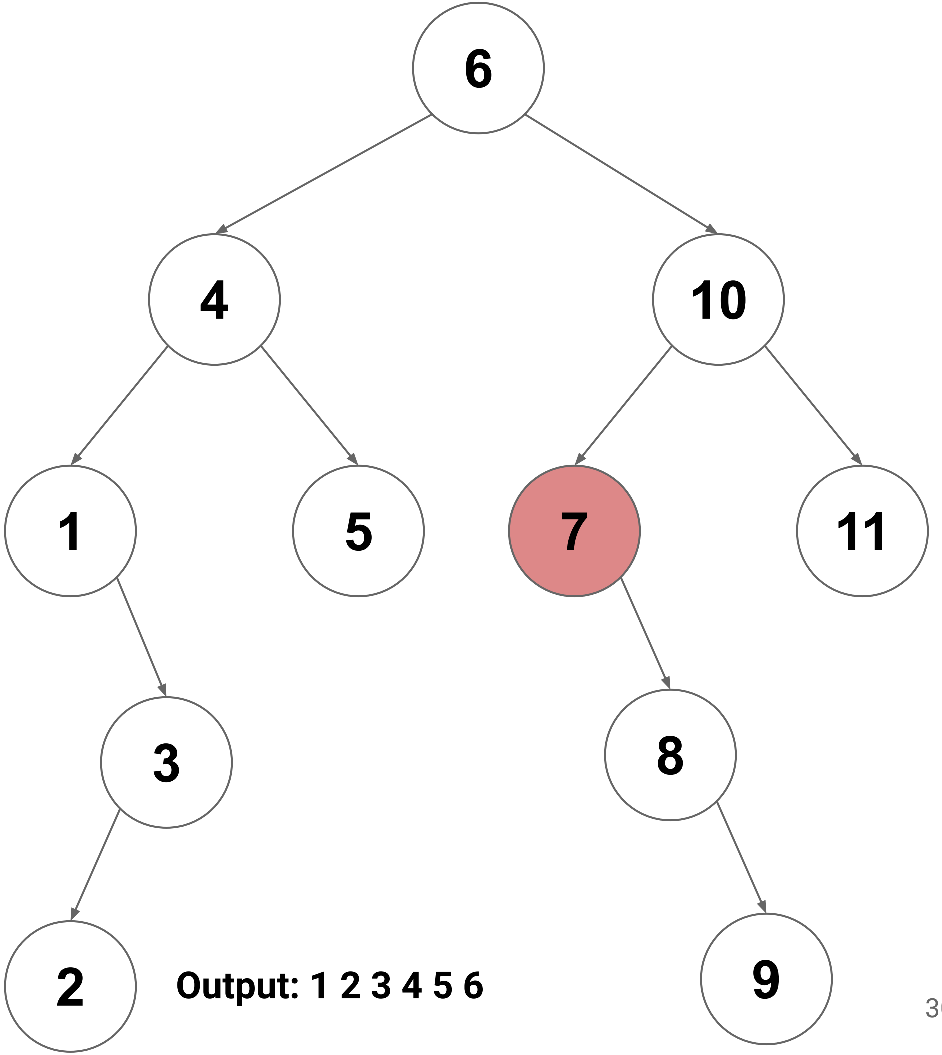
# In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(10)
```



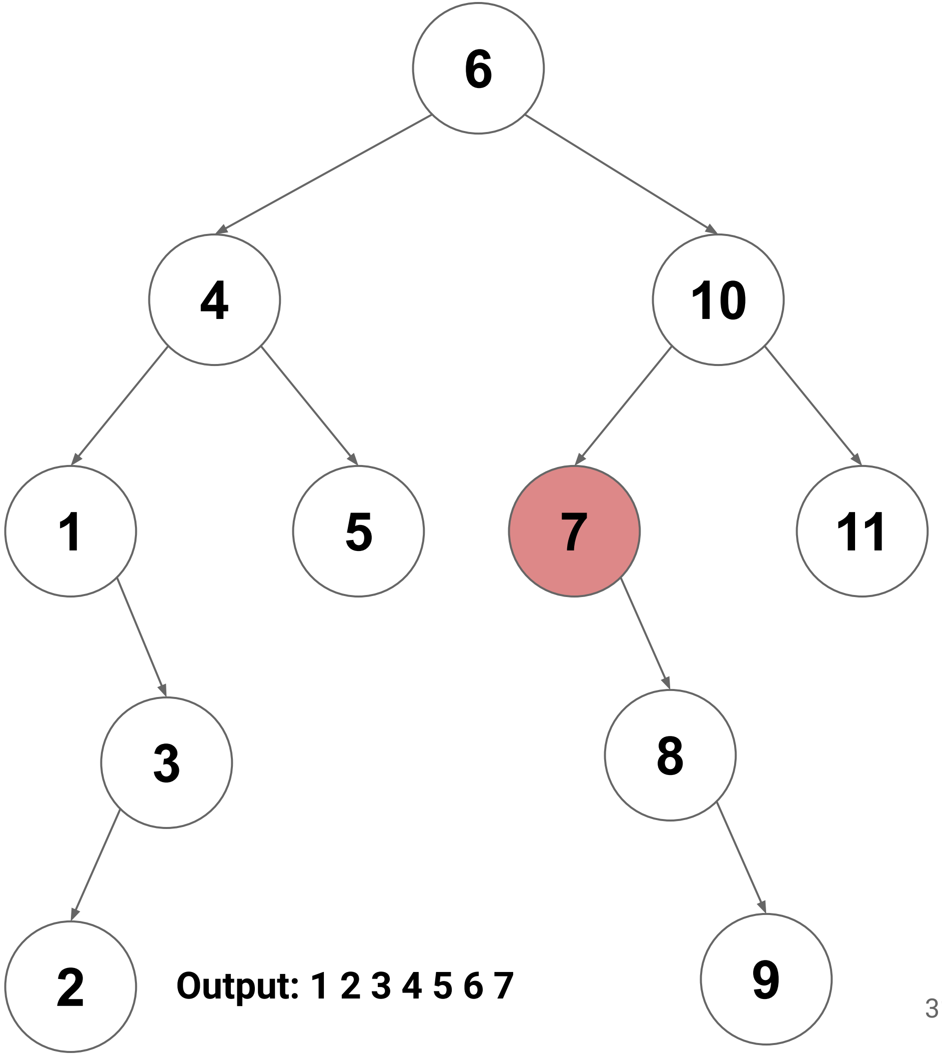
# In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(10)

inorderVisit(7)
```



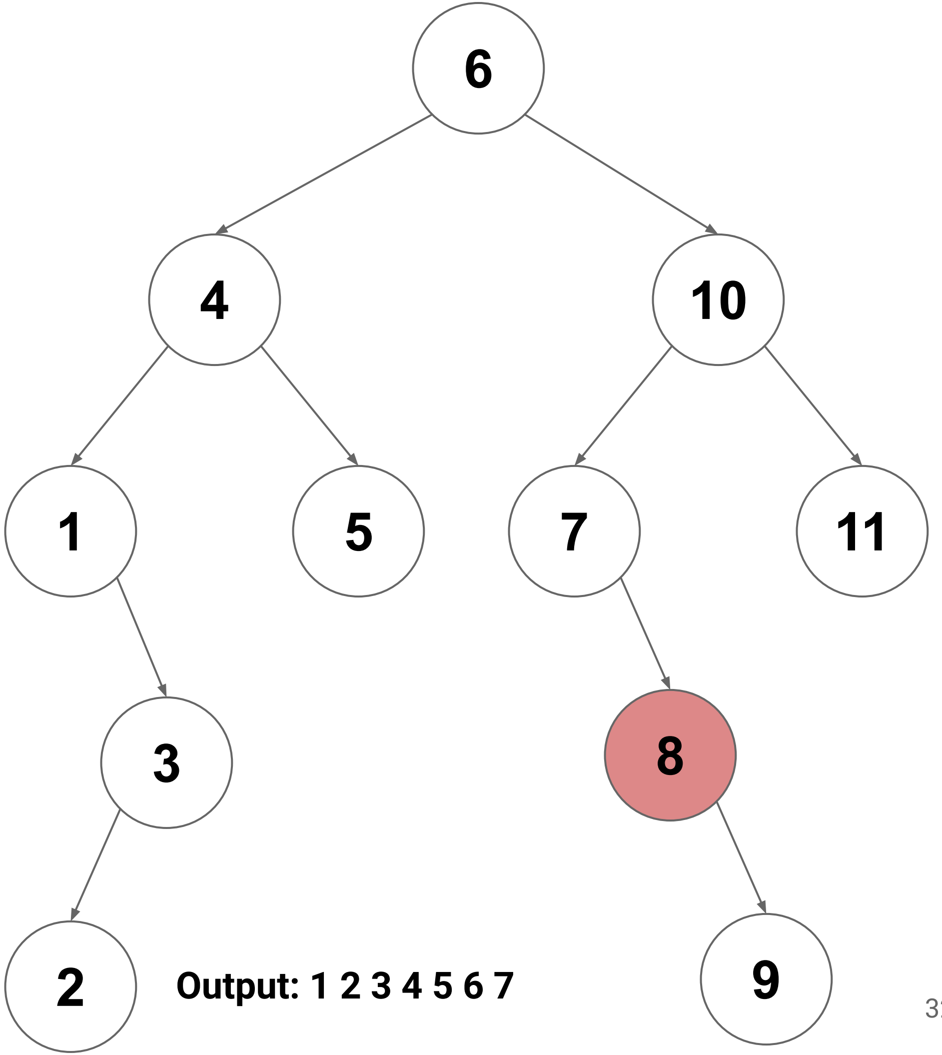
# In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(10)

inorderVisit(7)

visit(7)
```



# In-Order Traversal on a BST

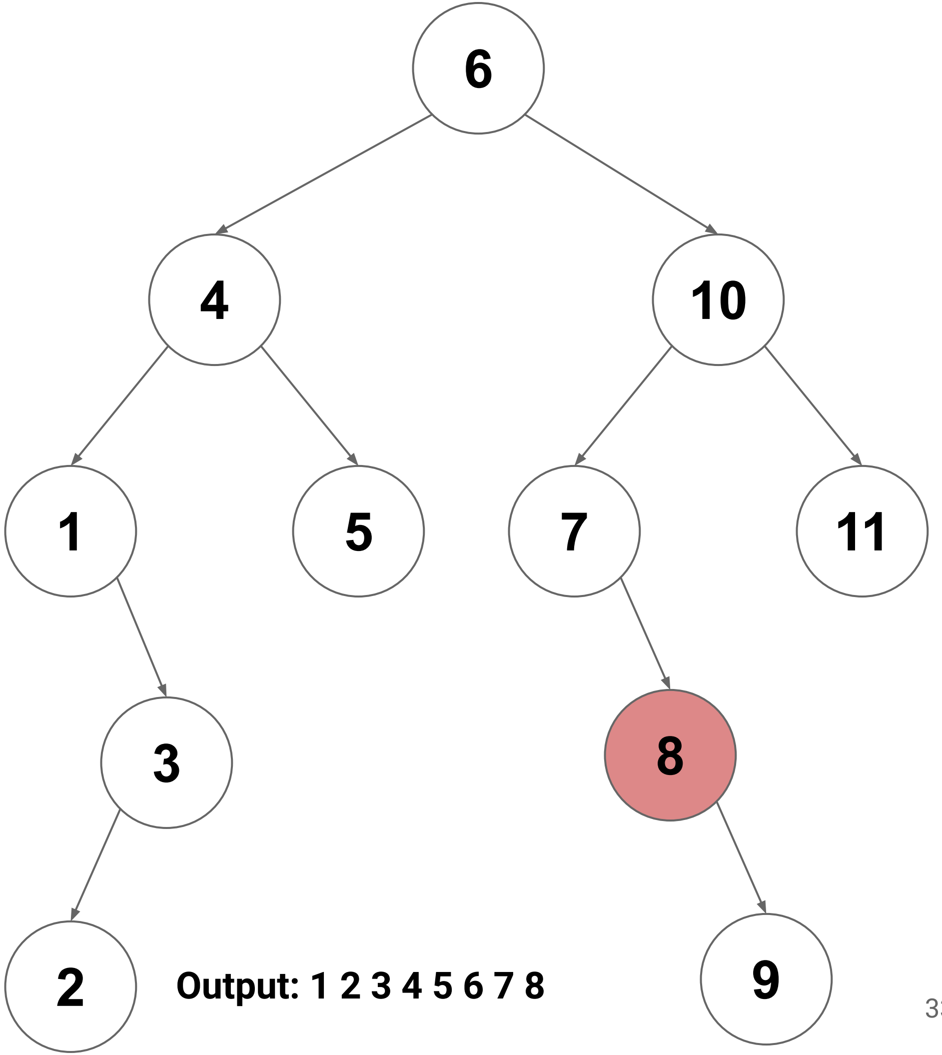
```
inorderVisit(6)

inorderVisit(10)

inorderVisit(7)

inorderVisit(8)
```




Output: 1 2 3 4 5 6 7

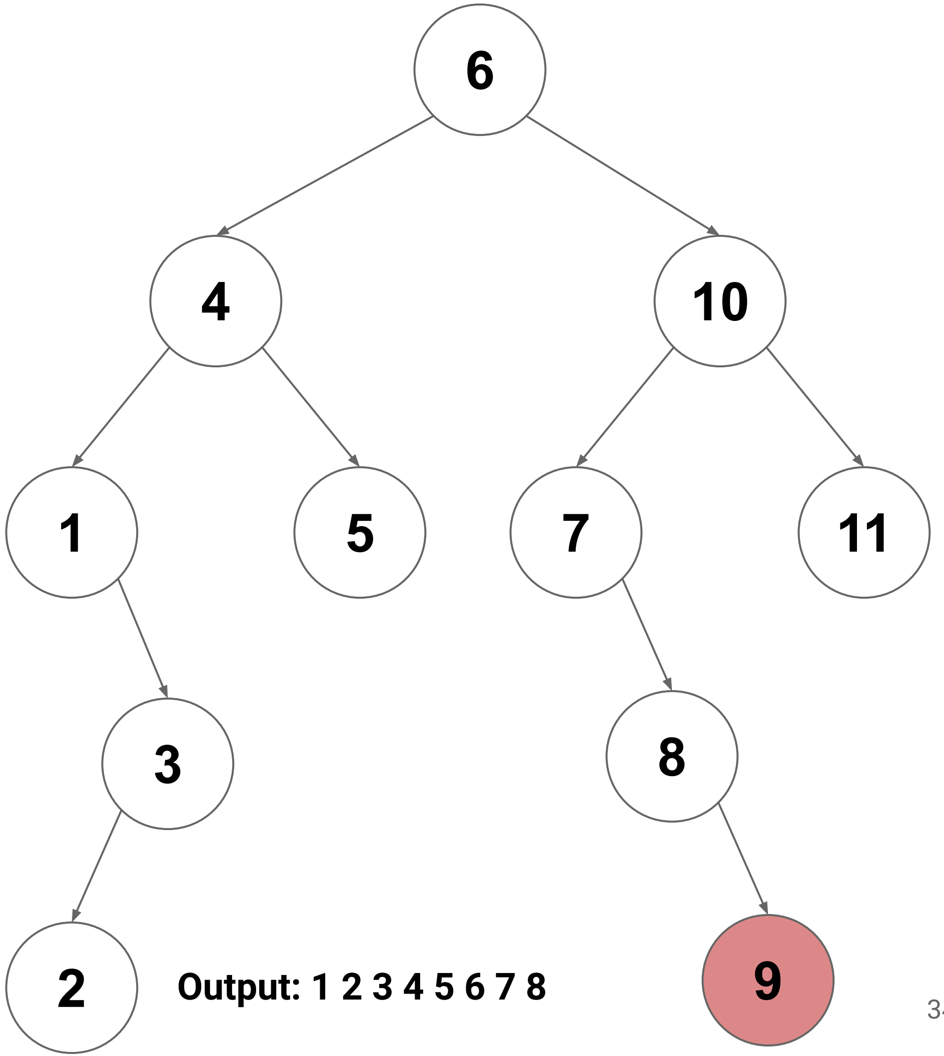
# In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(10)

inorderVisit(7)

inorderVisit(8)

visit(8)
```



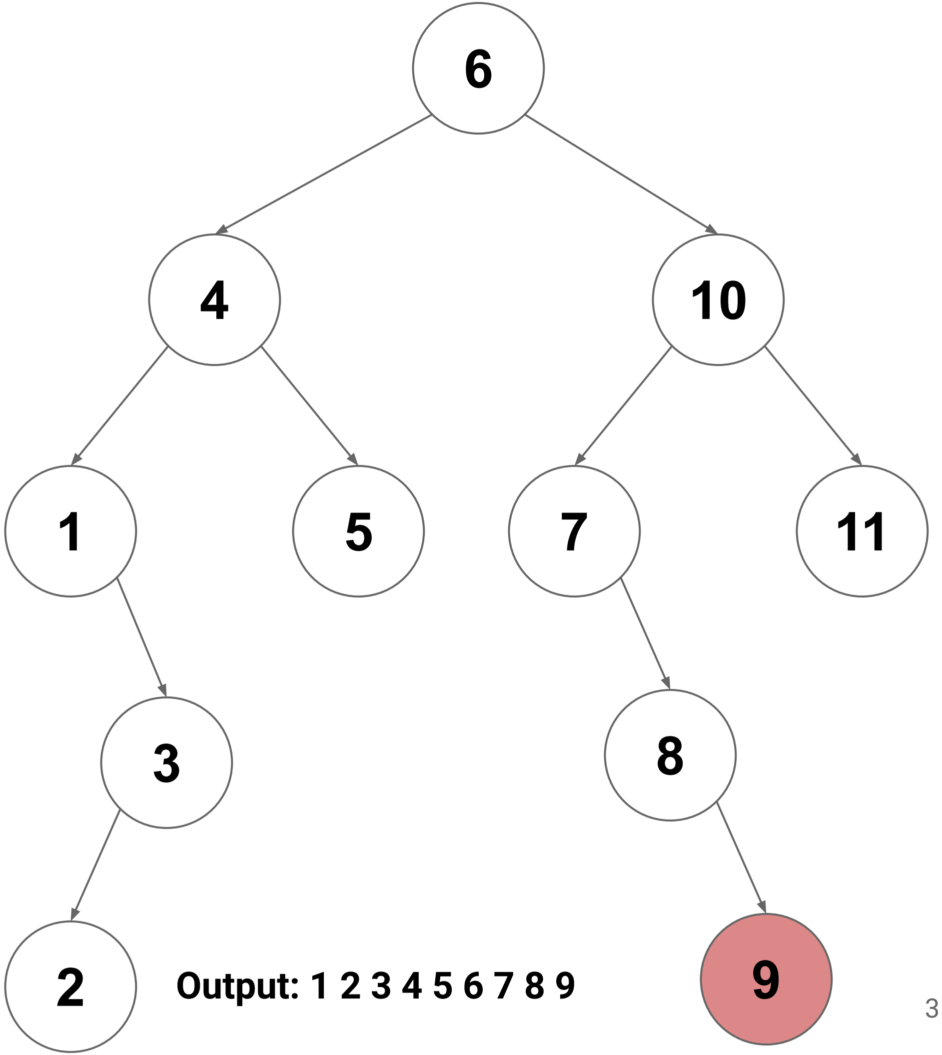
# In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(10)

inorderVisit(7)

inorderVisit(8)

inorderVisit(9)
```



# In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(10)

inorderVisit(7)

inorderVisit(8)

visit(9)
```




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

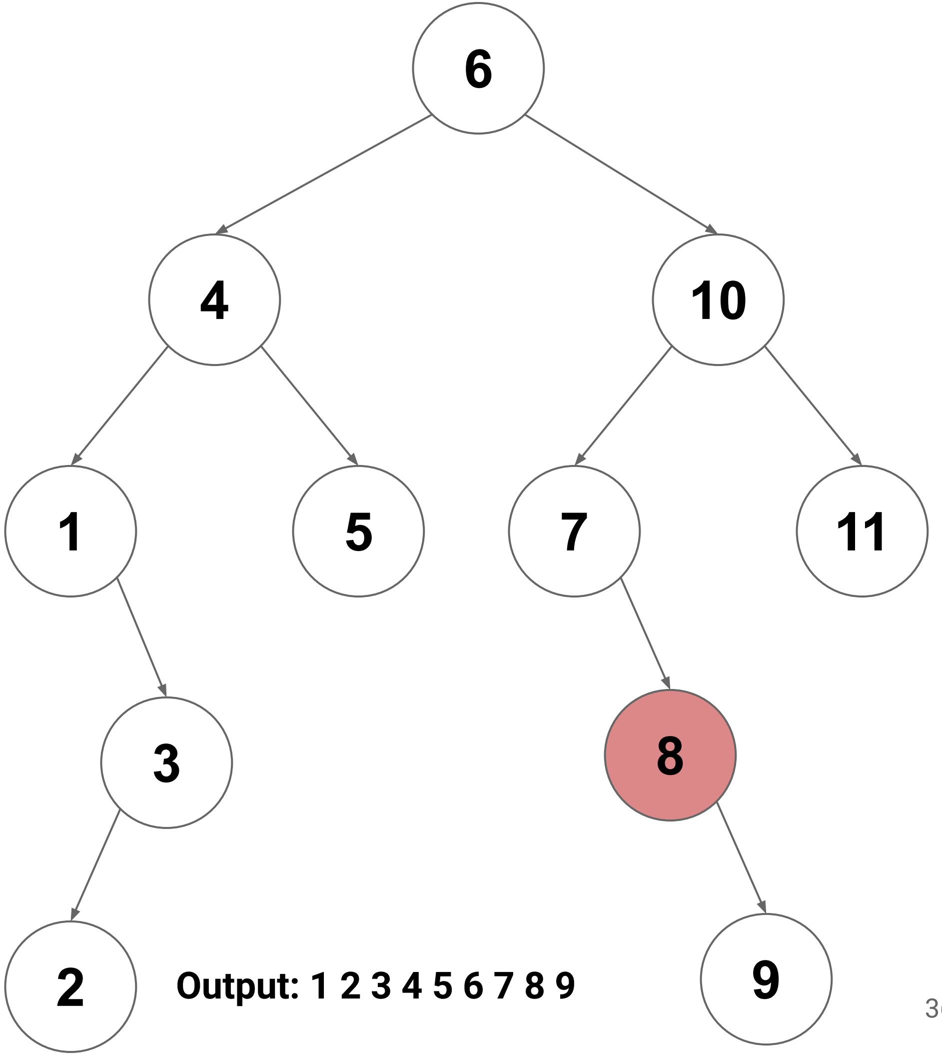
# In-Order Traversal on a BST

`inorderVisit(6)`

`inorderVisit(10)`

`inorderVisit(7)`

`inorderVisit(8)`



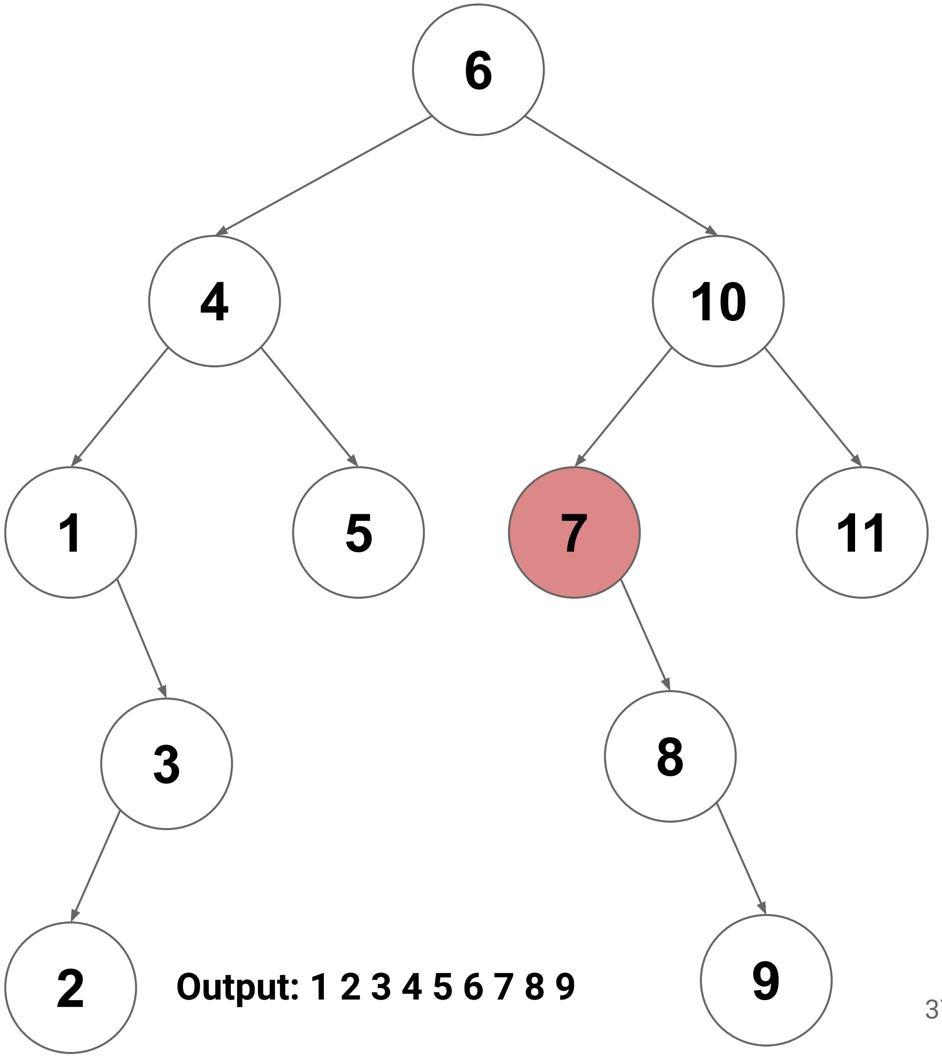

**Output: 1 2 3 4 5 6 7 8 9**

# In-Order Traversal on a BST

`inorderVisit(6)`

`inorderVisit(10)`

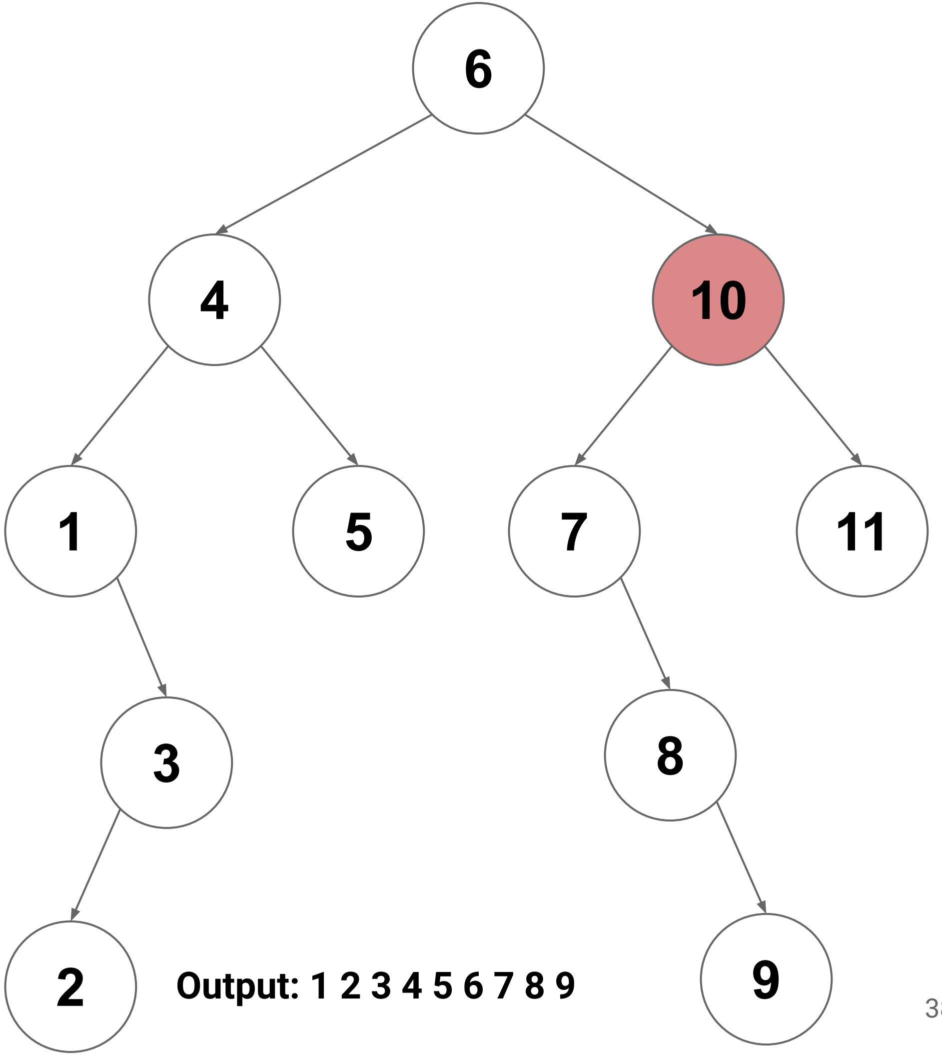
`inorderVisit(7)`



# In-Order Traversal on a BST

`inorderVisit(6)`

`inorderVisit(10)`



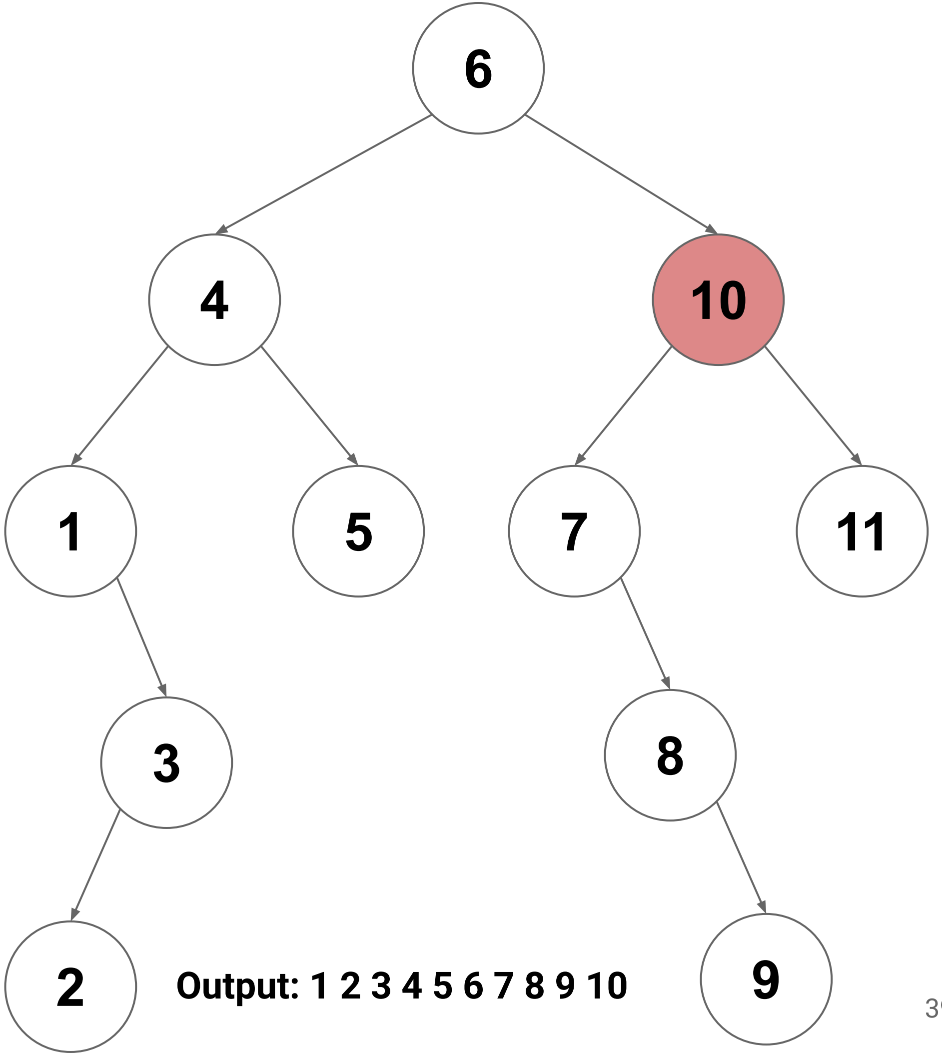

Output: 1 2 3 4 5 6 7 8 9

# In-Order Traversal on a BST

`inorderVisit(6)`

`inorderVisit(10)`

`visit(10)`

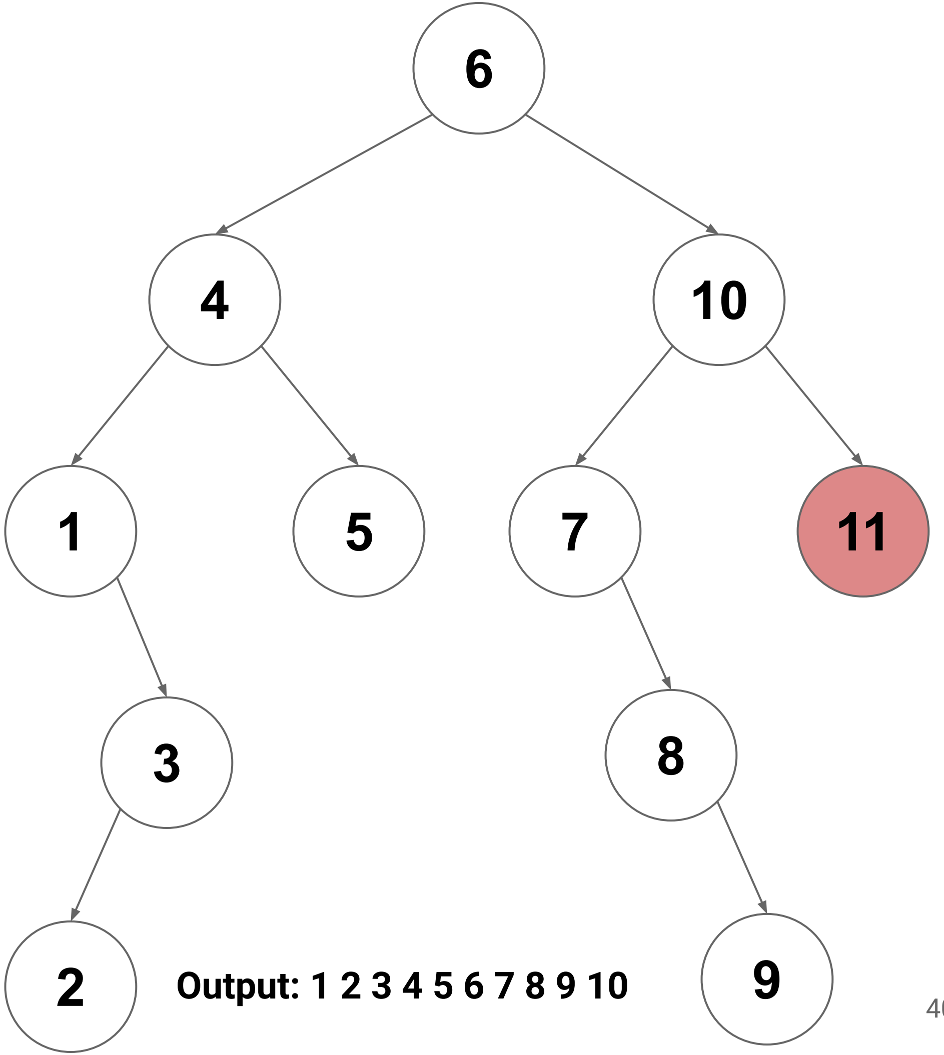


# In-Order Traversal on a BST

`inorderVisit(6)`

`inorderVisit(10)`

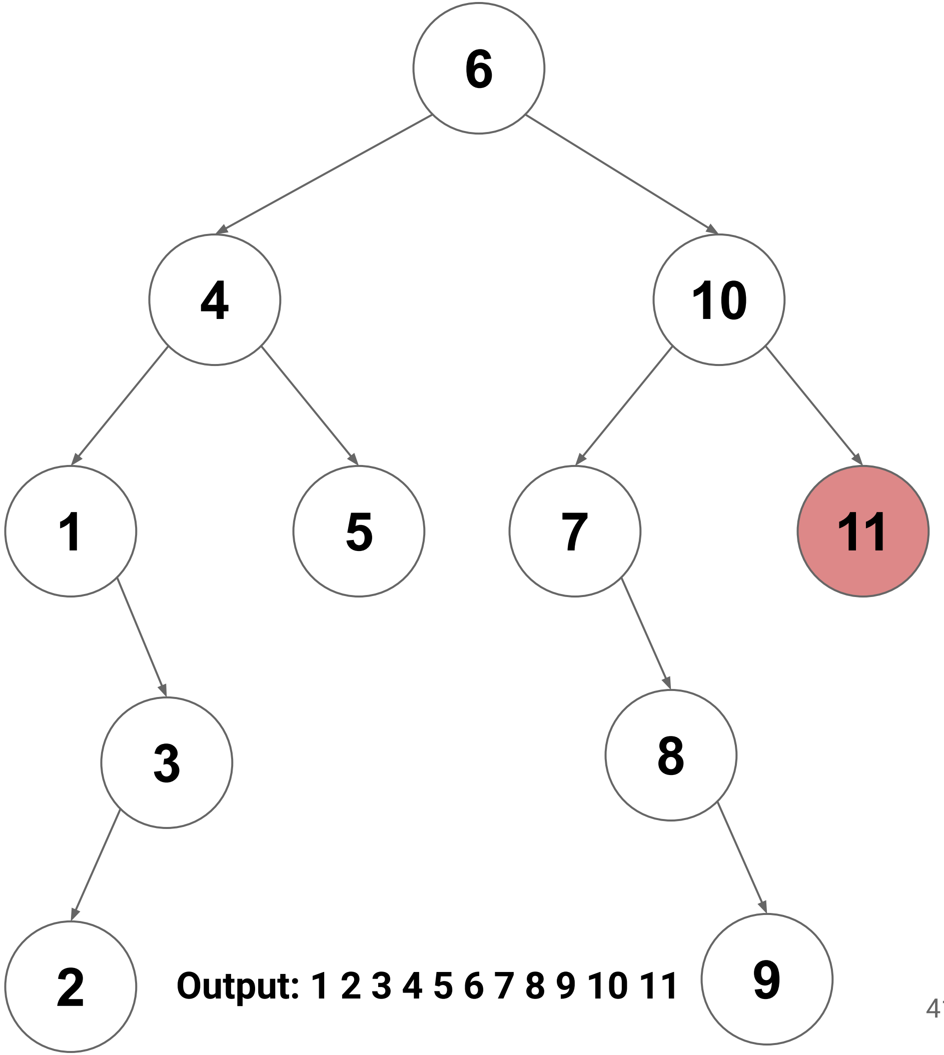
`inorderVisit(11)`




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

# In-Order Traversal on a BST

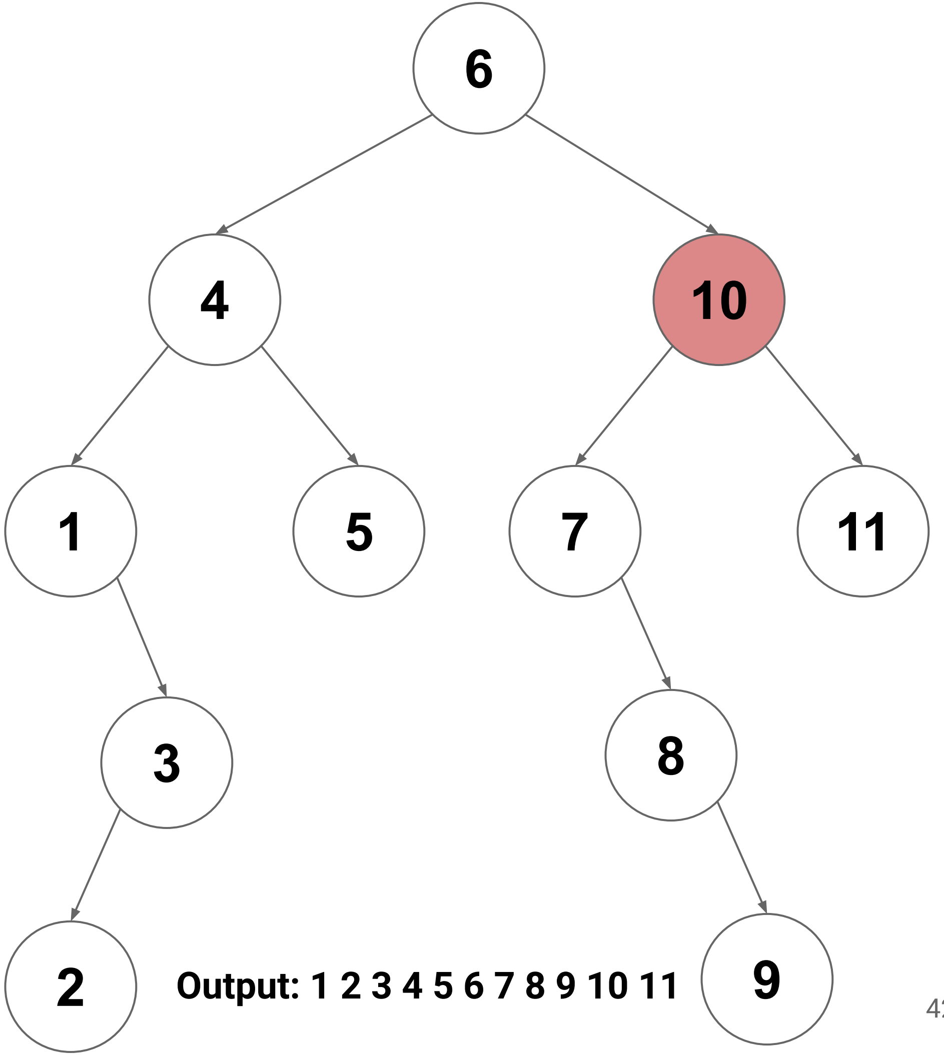
```
inorderVisit(6)

inorderVisit(10)

inorderVisit(11)

visit(11)
```



# In-Order Traversal on a BST

`inorderVisit(6)`

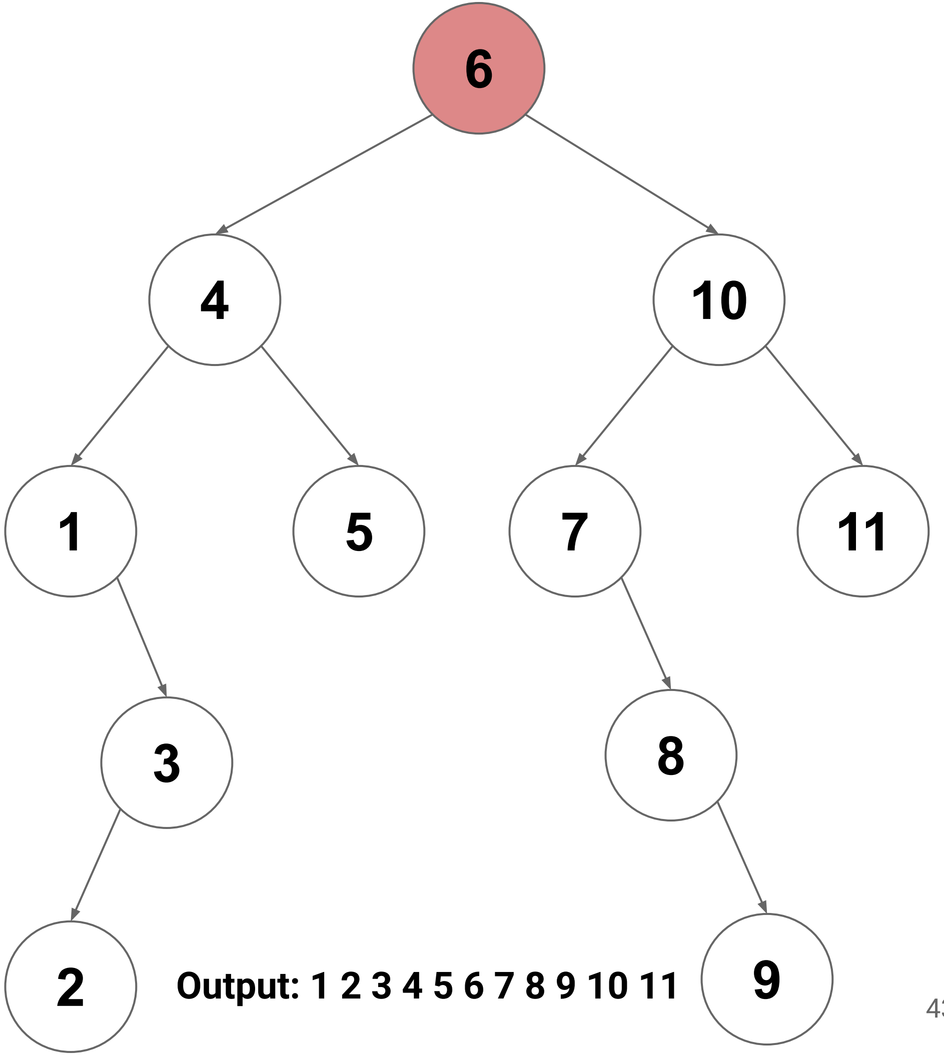
`inorderVisit(10)`



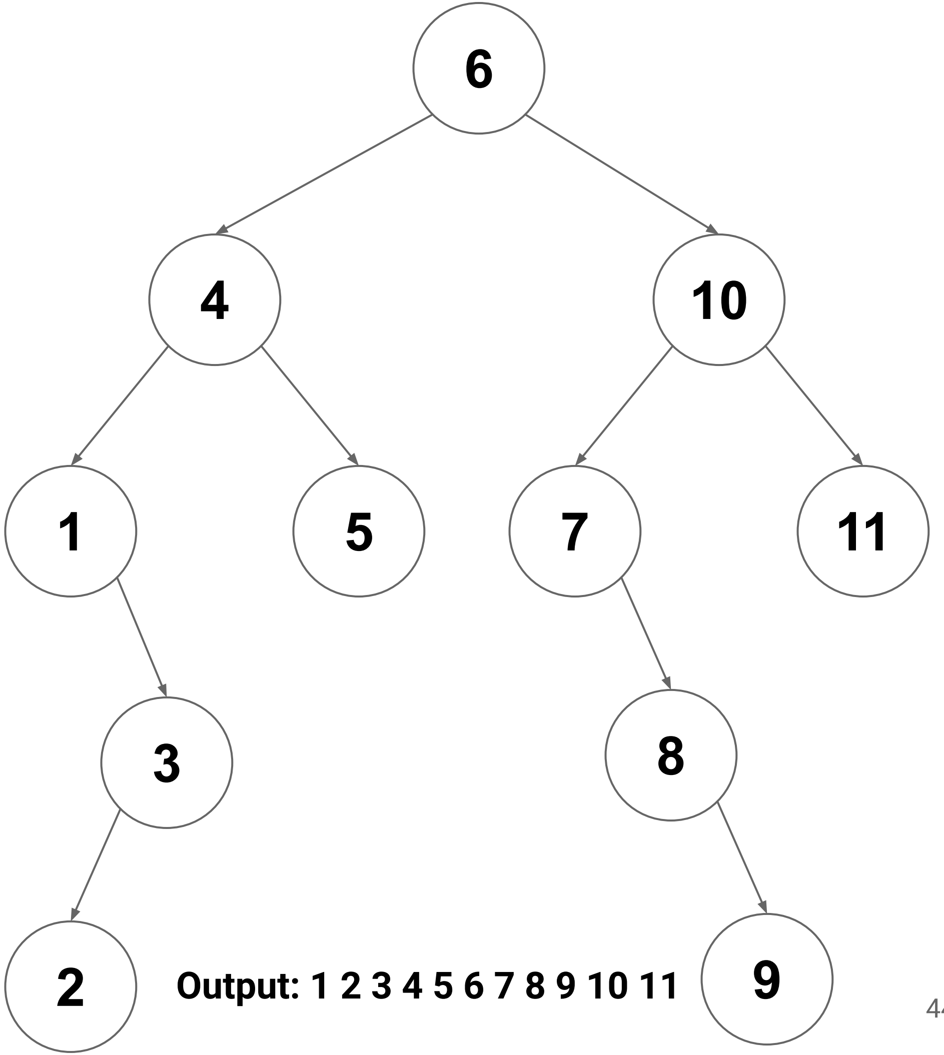

**Output: 1 2 3 4 5 6 7 8 9 10 11**

# In-Order Traversal on a BST

`inorderVisit(6)`



# In-Order Traversal on a BST




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

# Tree Traversal: In-Order Iterator

```java
class TreeIterator<T> implements Iterator<T> {
  Stack<TreeNode<T>> toVisit;
  TreeIterator() {
    toVisit = new Stack<>();
    pushLeft(root);
  }
  void pushLeft(Optional<TreeNode<T>> node) {
    if (node.isPresent()) {
      toVisit.push(node.get());
      pushLeft(node.get().leftChild());
    }
  }
  /* ... */
}
```

# Tree Traversal: In-Order Iterator

```
class TreeIterator<T> implements Iterator<T> {
  Stack<TreeNode<T>> toVisit;
  TreeIterator() {
    toVisit = new Stack<>();
    pushLeft(root);
  }
  void pushLeft(Optional<TreeNode<T>> node) {
    if (node.isPresent()) {
      toVisit.push(node.get());
      pushLeft(node.get().leftChild());
    }
  }
  /* ... */
}
```

Keep track of what we need to visit in a stack

Recursively push all the left nodes (they are the smallest and therefore should be visited first)

# Tree Traversal: In-Order Iterator

```
class TreeIterator<T> implements Iterator<T> {
  Stack<TreeNode<T>> toVisit;
  TreeIterator() {
    toVisit = new Stack<>();
    pushLeft(root);
  }
  void pushLeft(Optional<TreeNode<T>> node) {
    if (node.isPresent()) {
      toVisit.push(node.get());
      pushLeft(node.get().leftChild());
    }
  }
  /* ... */
}
```

Push the node, and then recursively push it's left children

# Tree Traversal: In-Order Iterator

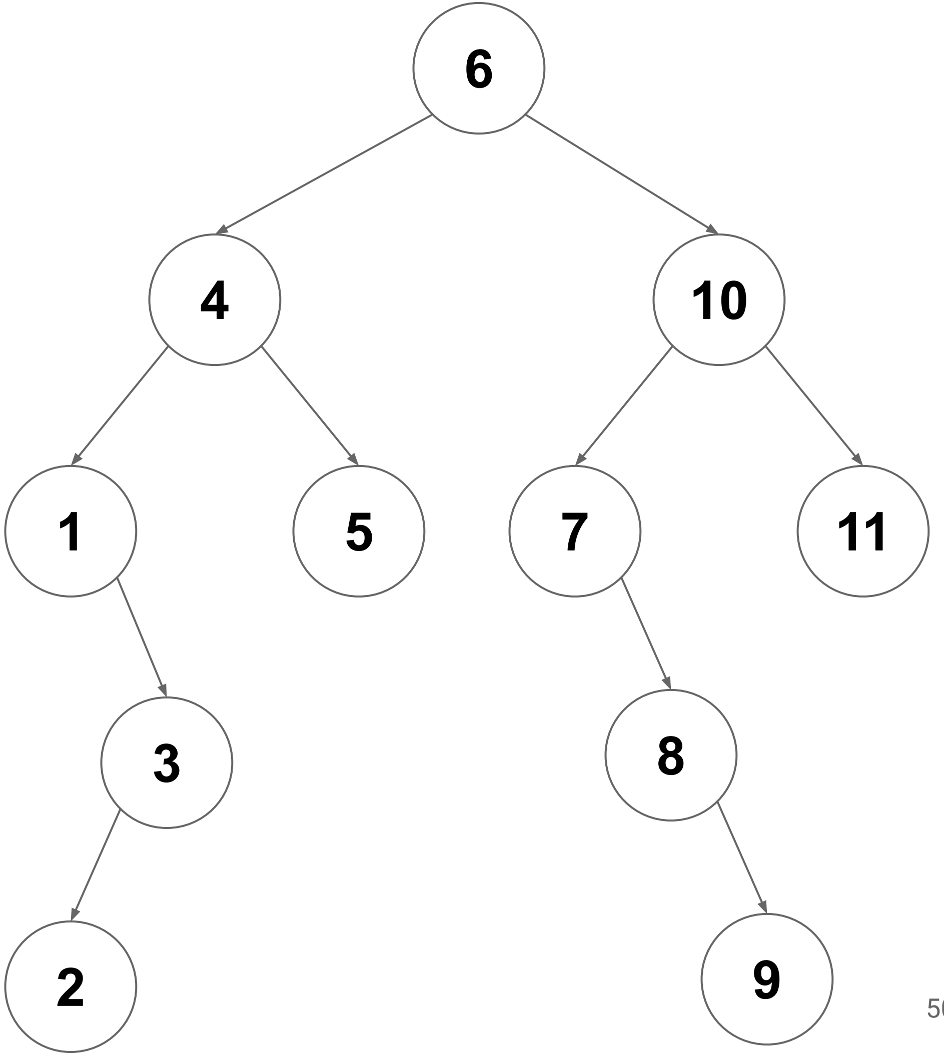
```
class TreeIterator<T> implements Iterator<T> {
  /* ... */

  boolean hasNext() { return !toVisit.isEmpty(); }

  T next() {
    TreeNode<T> nextNode = toVisit.pop();
    pushLeft(nextNode.rightChild);
    return nextNode.value;
  }
}
```

# Tree Traversal: In-Order Iterator

```java
class TreeIterator<T> implements Iterator<T> {
  /* ... */

  boolean hasNext() { return !toVisit.isEmpty(); }

  T next() {
    TreeNode<T> nextNode = toVisit.pop();
    pushLeft(nextNode.rightChild);
    return nextNode.value;
  }
}
```

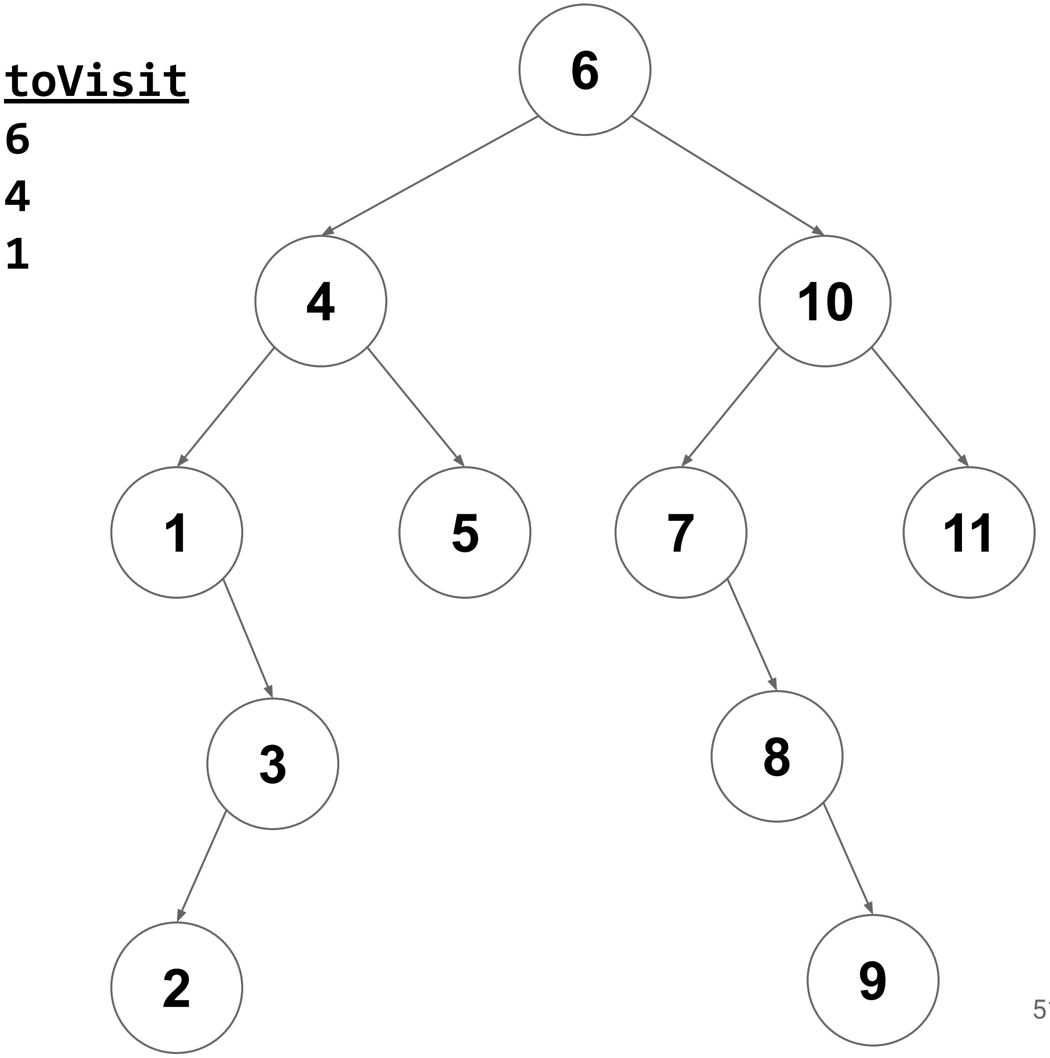
Pop the next node, then push the left children of it's right subtree

# In-Order Traversal with an Iterator



# In-Order Traversal with an Iterator

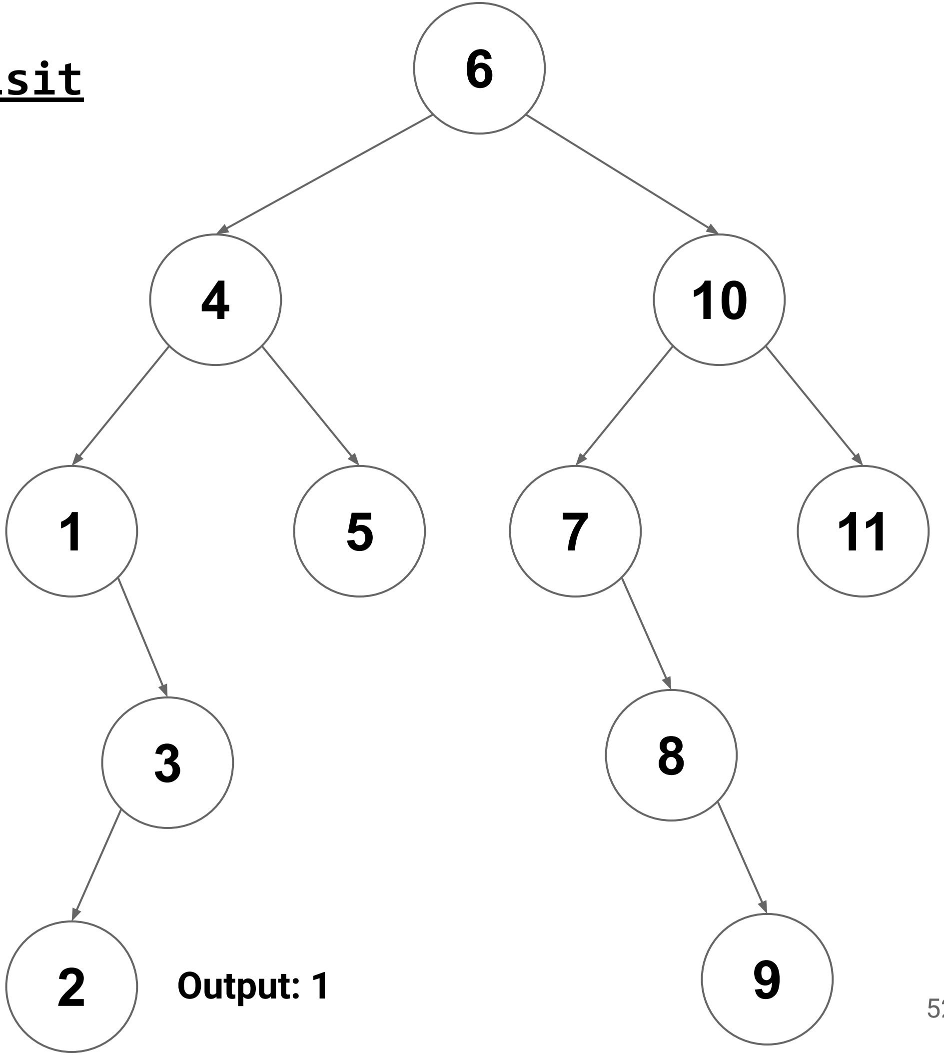
When we create the iterator, the **`toVisit`** stack is initialized



# In-Order Traversal with an Iterator

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

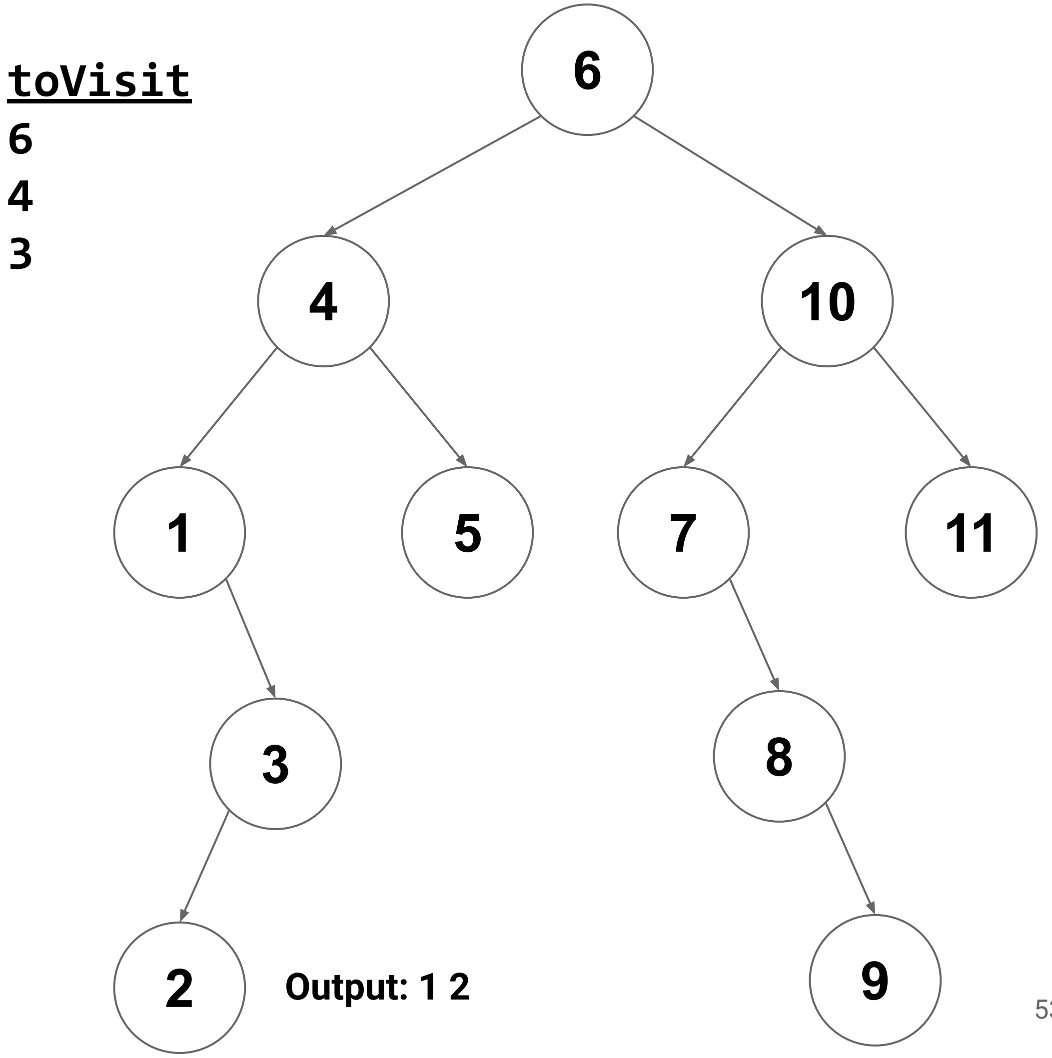
**toVisit**
6
4
3
2




**Output: 1**

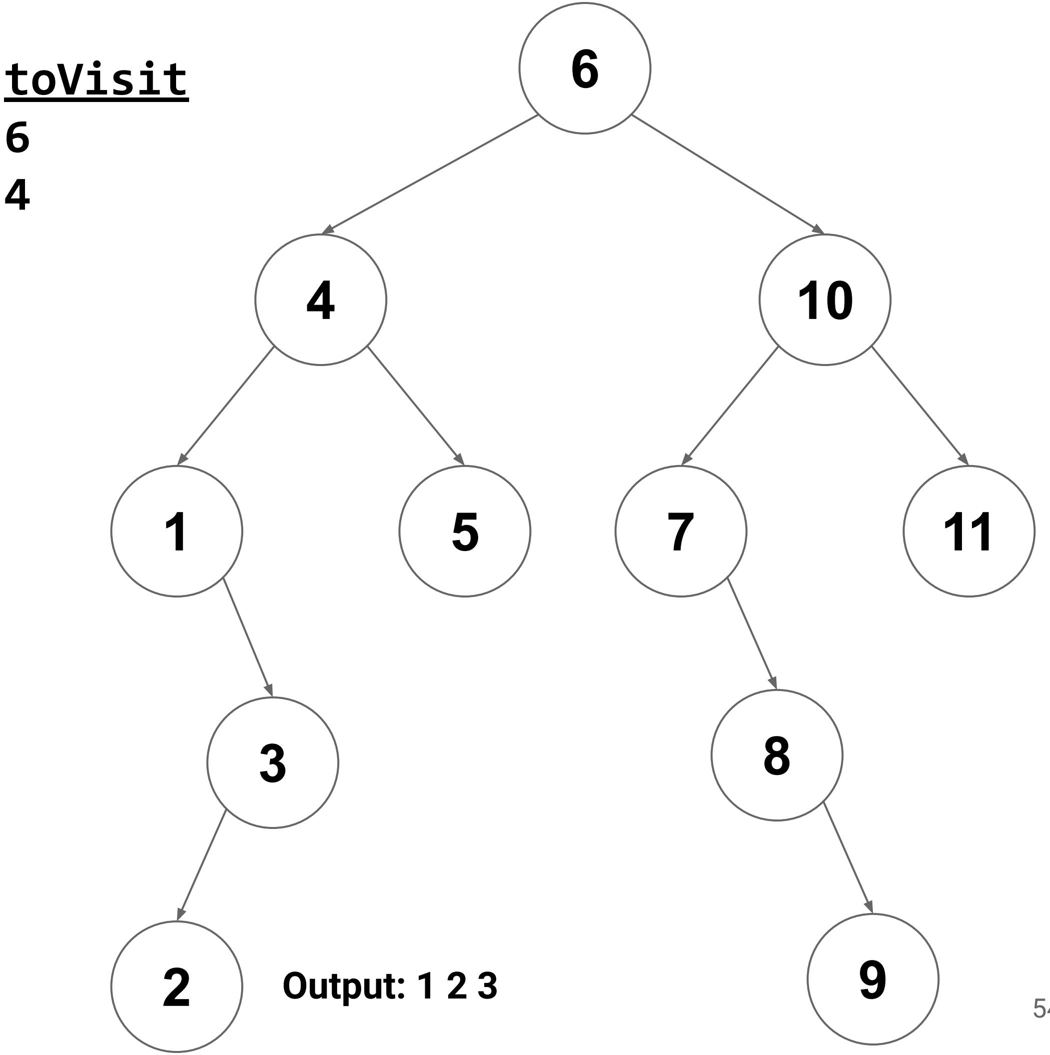
# In-Order Traversal with an Iterator

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



# In-Order Traversal with an Iterator

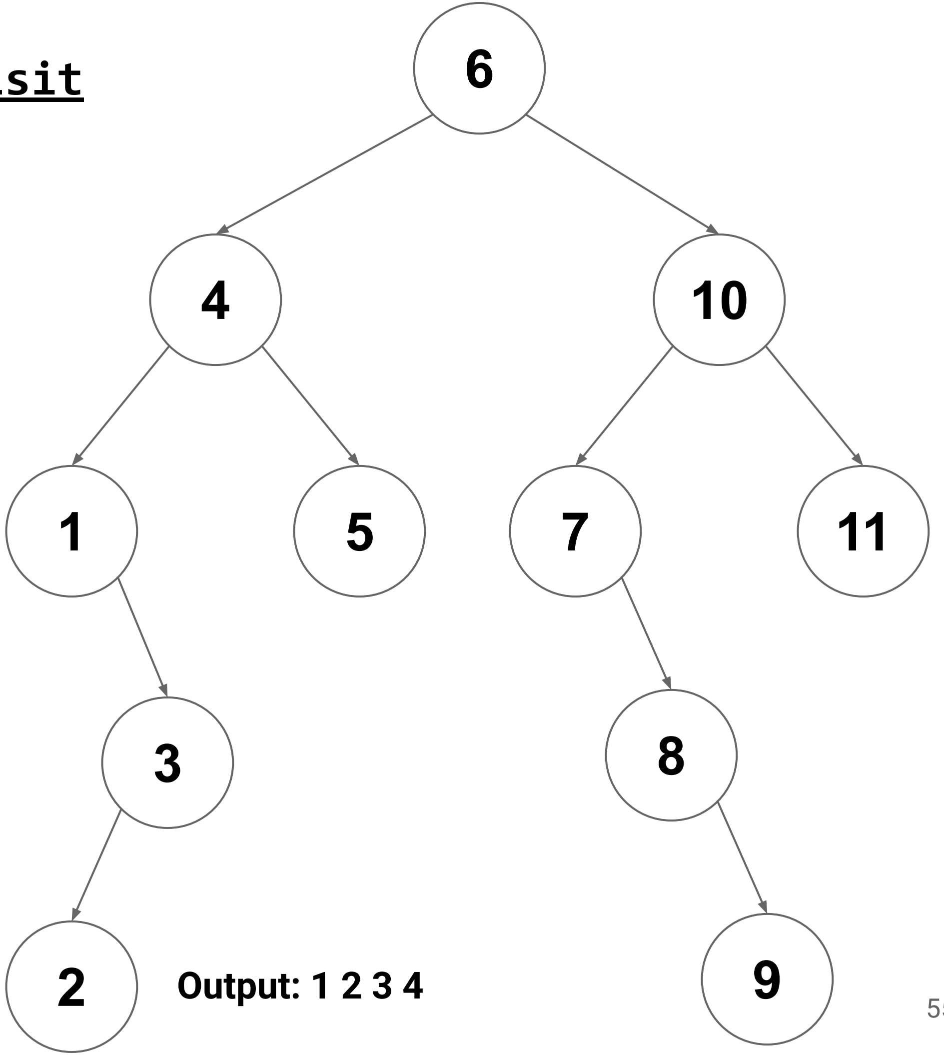
**next** pops the stack (3) and pushes the right subtree (nothing)



# In-Order Traversal with an Iterator

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

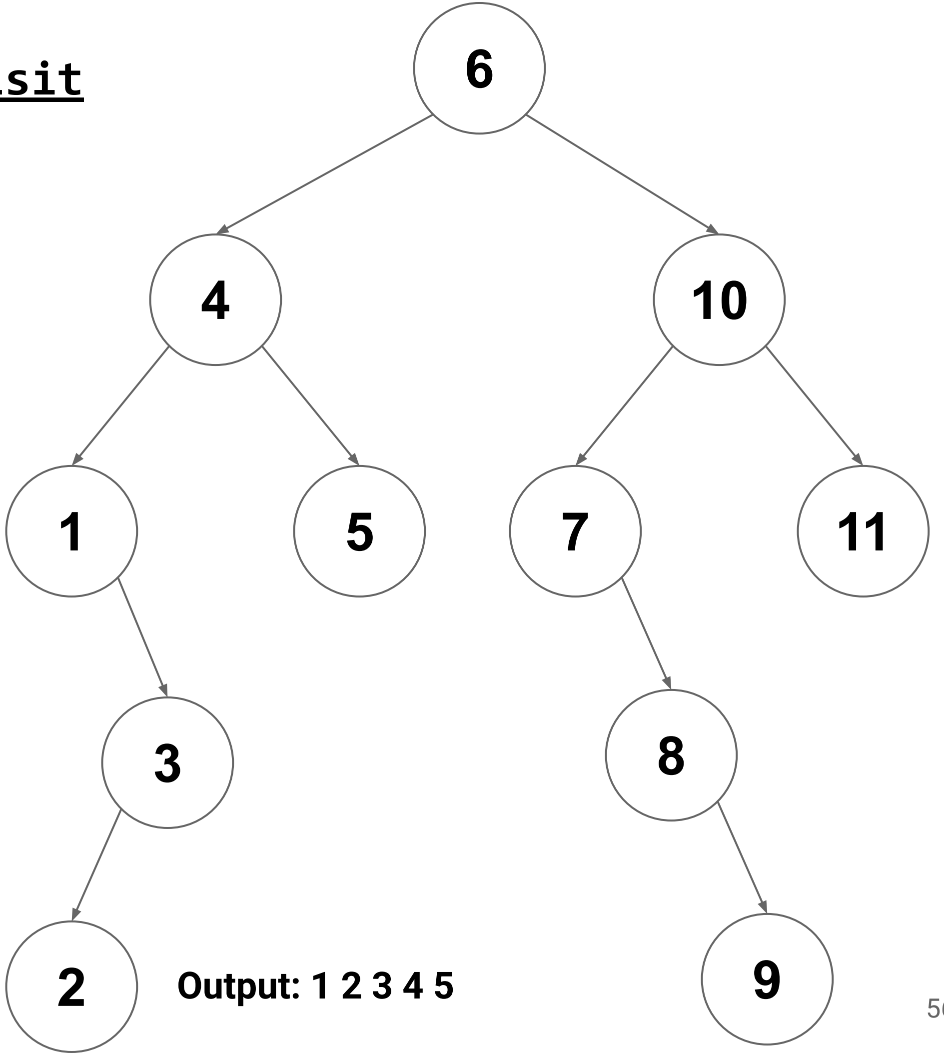
toVisit
6
5



# In-Order Traversal with an Iterator

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

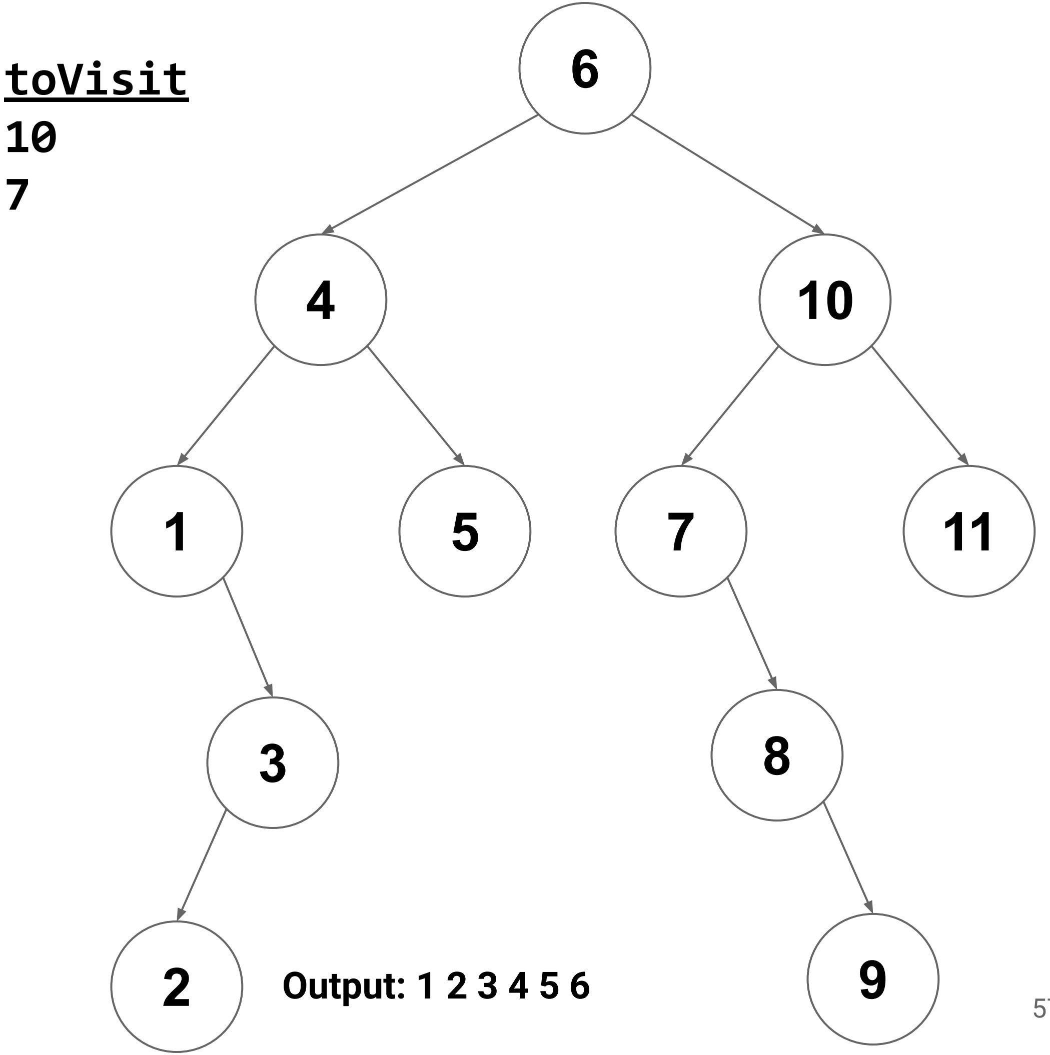
**toVisit**
6




**Output: 1 2 3 4 5**

# In-Order Traversal with an Iterator

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



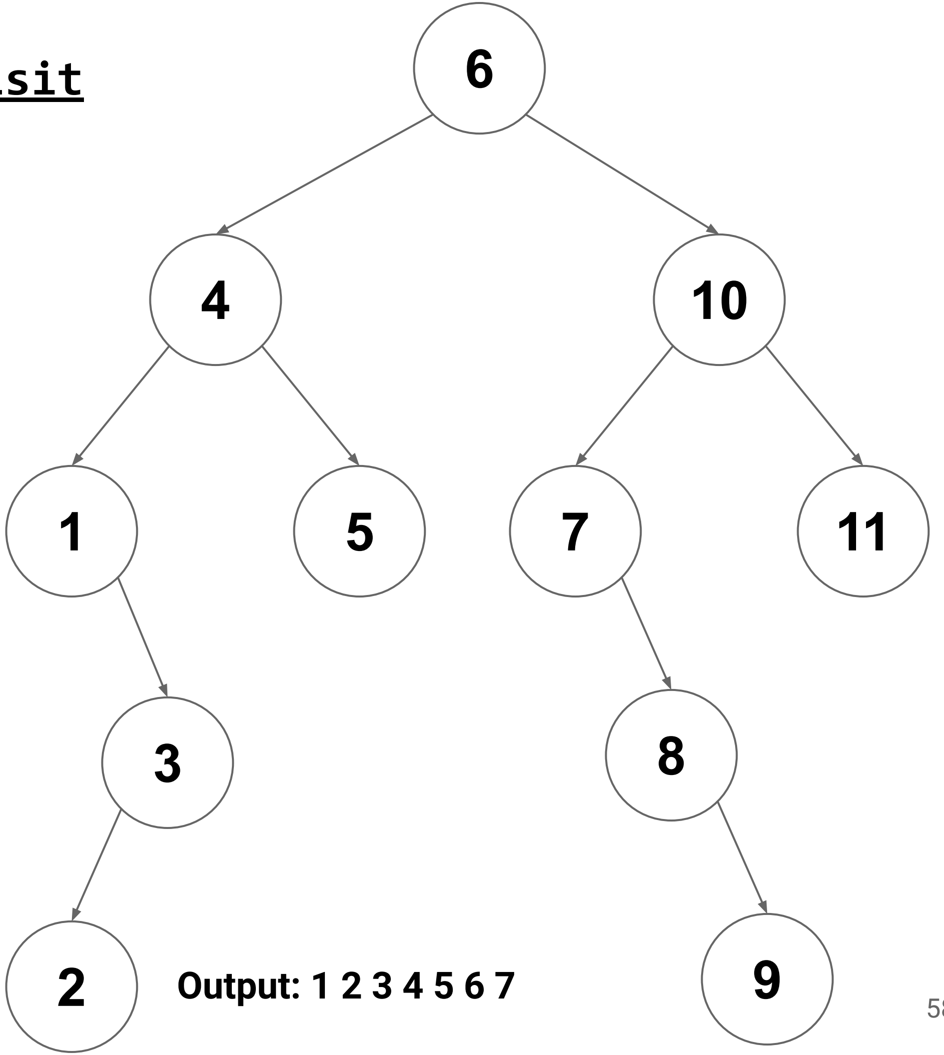
# In-Order Traversal with an Iterator

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

**toVisit**

10

8

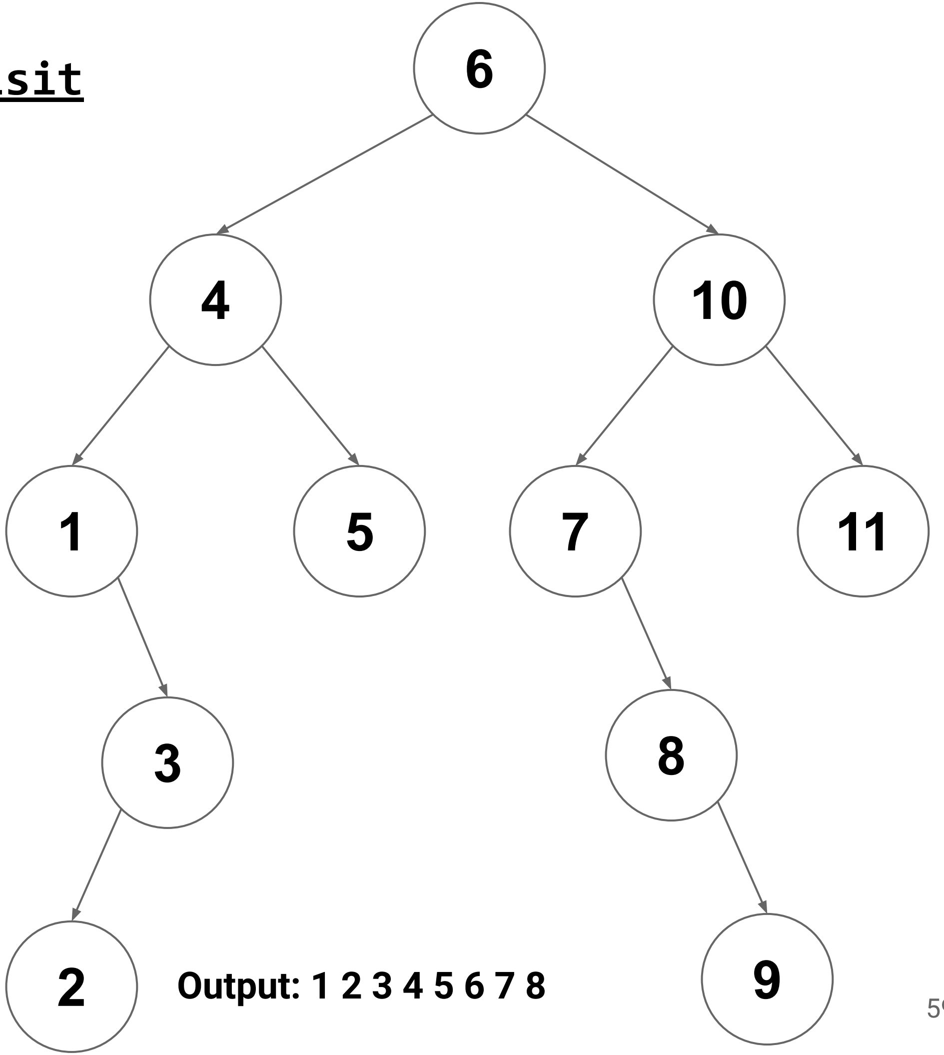



**Output: 1 2 3 4 5 6 7**

# In-Order Traversal with an Iterator

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

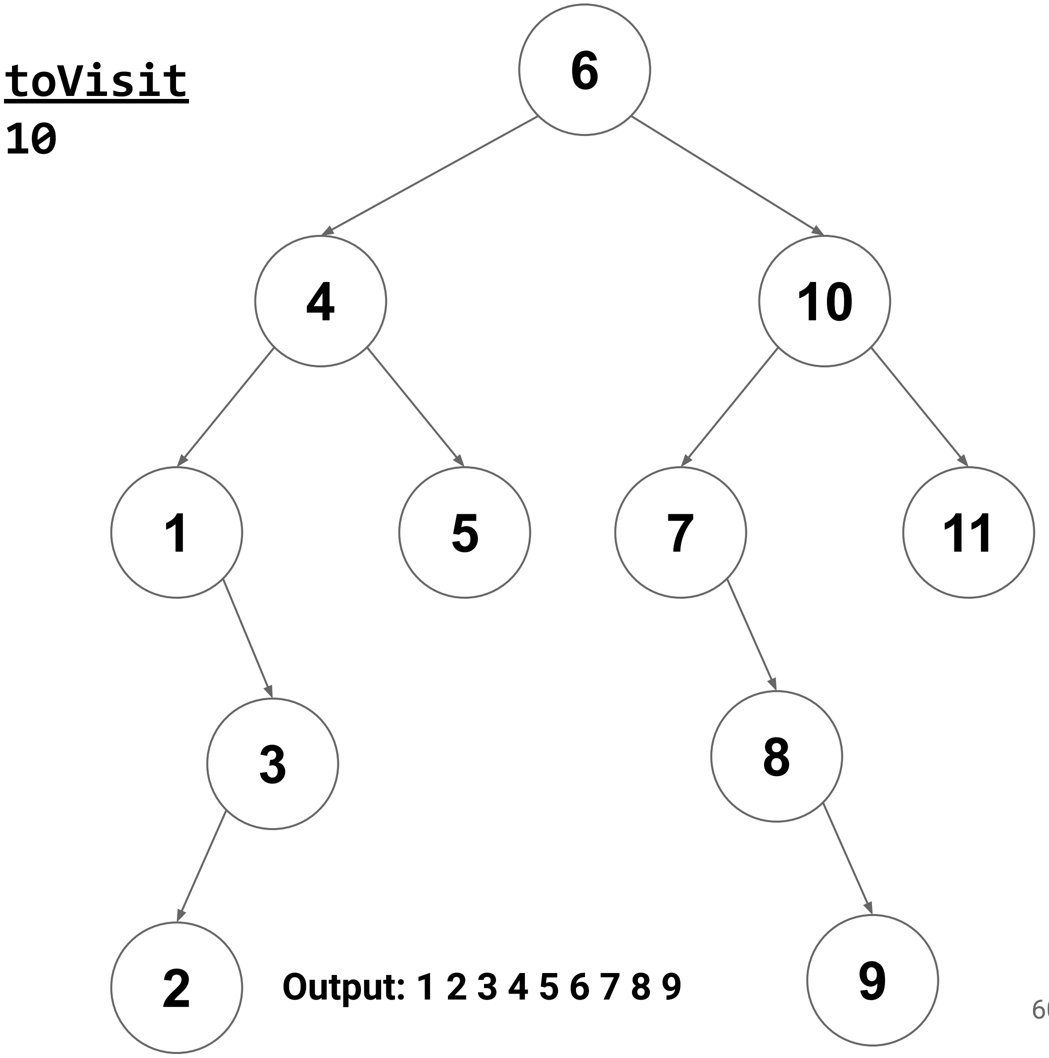
**toVisit**
```
10
9
```




**Output: 1 2 3 4 5 6 7 8**

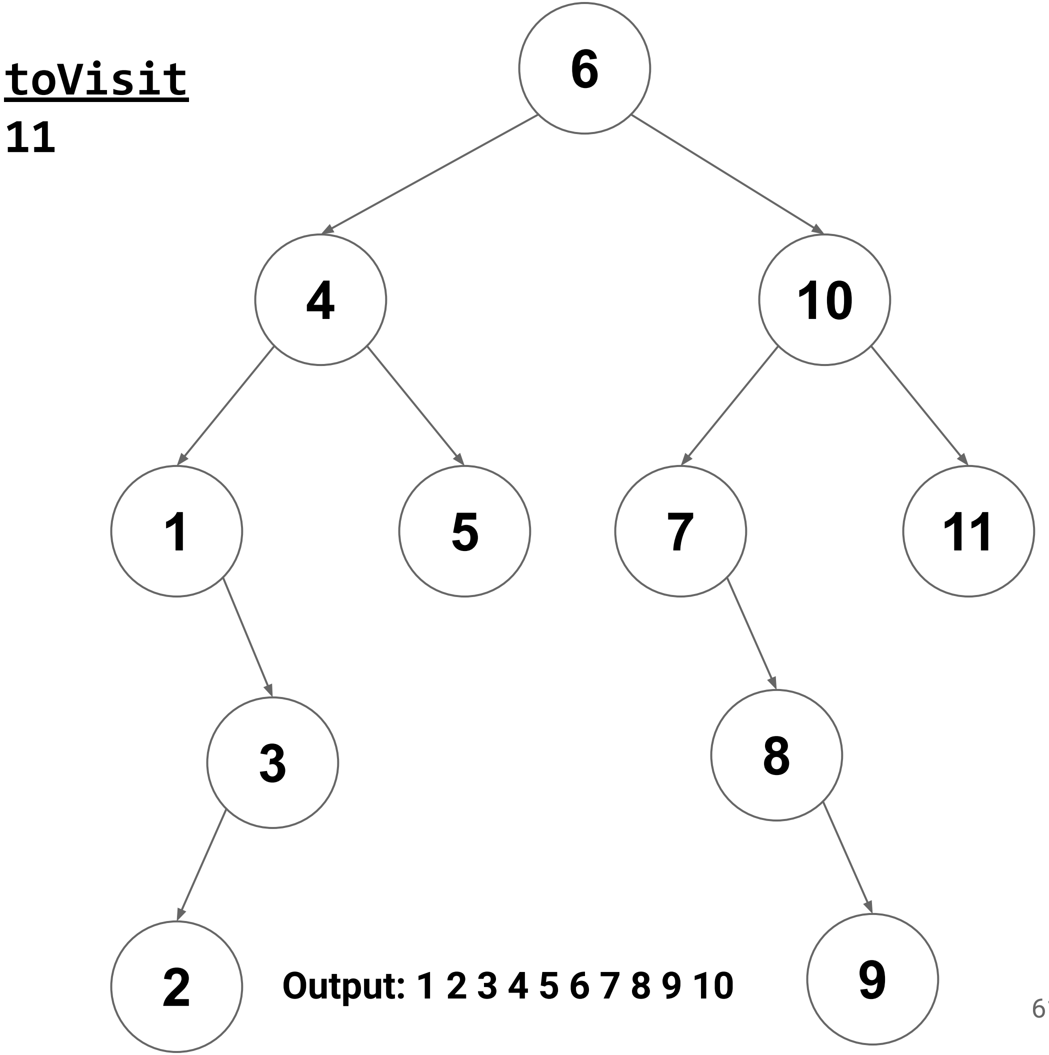
# In-Order Traversal with an Iterator

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



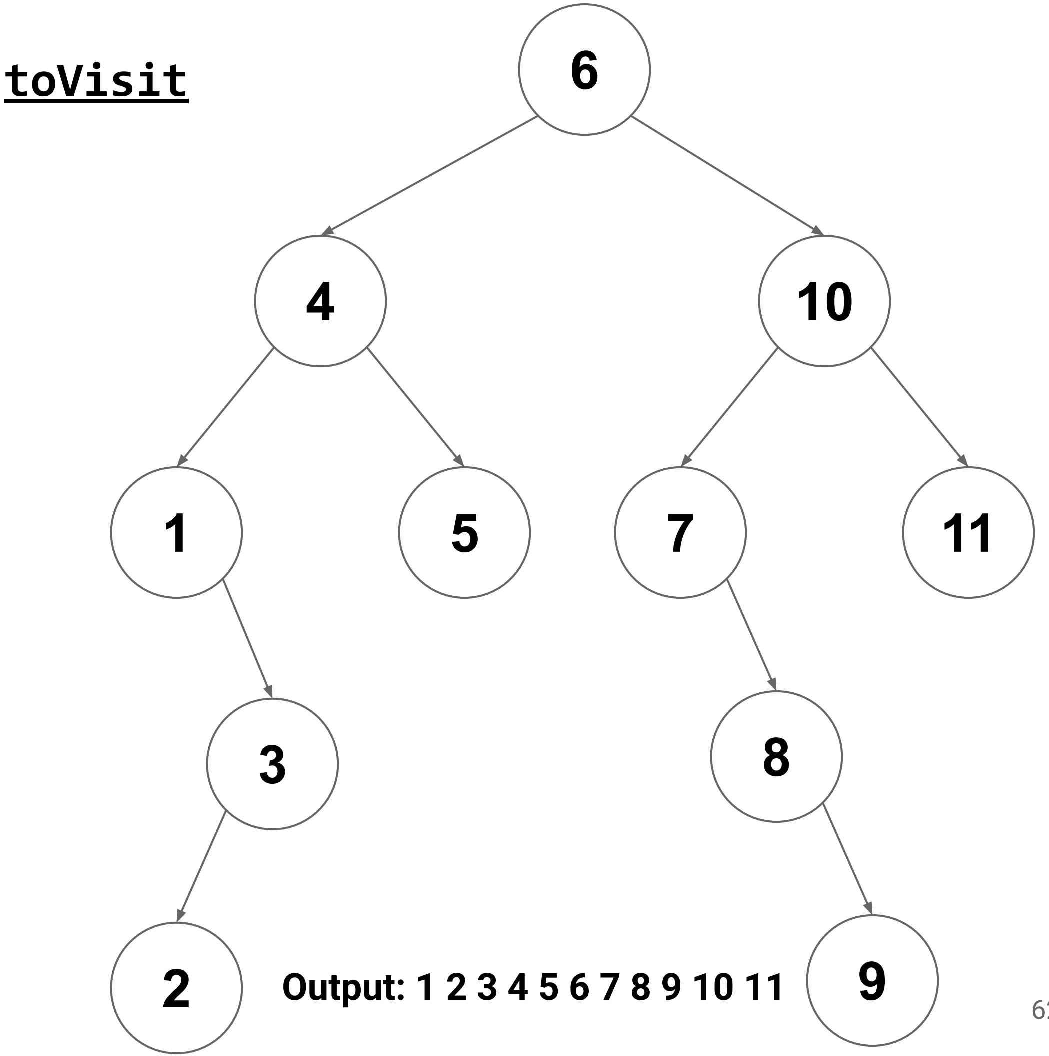
# In-Order Traversal with an Iterator

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



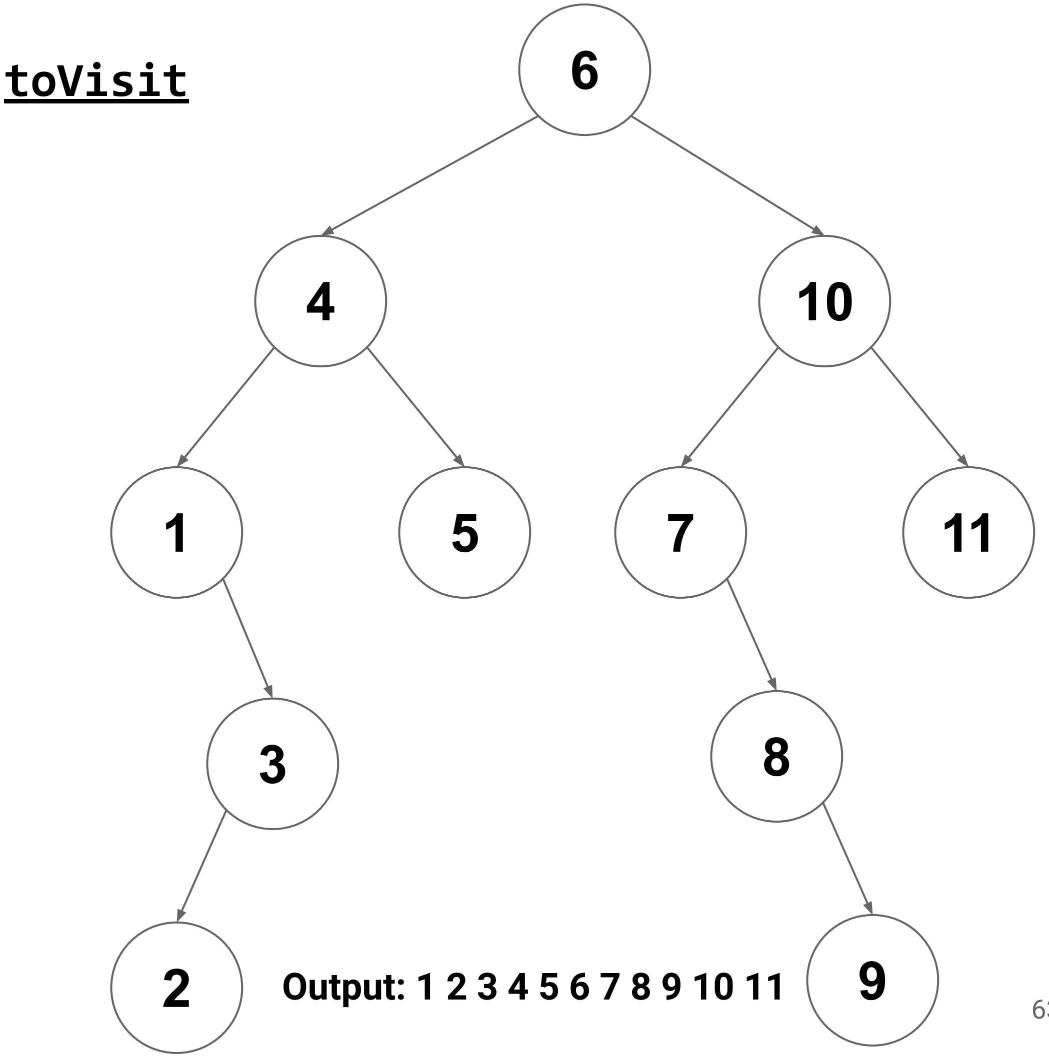
# In-Order Traversal with an Iterator

**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

```
TreeIterator() {
  toVisit = new Stack<>();
  pushLeft(root);
}
```

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

# Complexity

```java
TreeIterator() {
  toVisit = new Stack<>();
  pushLeft(root);
}
```

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

# Complexity

```
TreeIterator() {
  toVisit = new Stack<>();
  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

```
T next() {
  TreeNode<T> nextNode = toVisit.pop();
  pushLeft(nextNode.rightChild);
  return nextNode.value;
}
```

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

# Complexity

```
T next() {
  TreeNode<T> nextNode = toVisit.pop();
  pushLeft(nextNode.rightChild);
  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 $\boldsymbol{O}(1)$

# 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

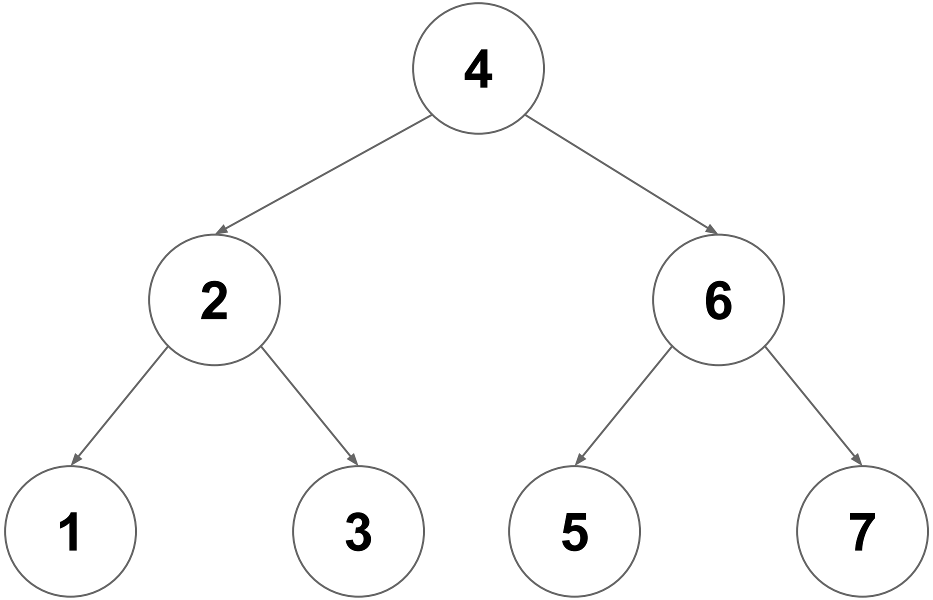
| Operation | Runtime |
|---|---|
| `find` | $O(d)$ |
| `insert` | $O(d)$ |
| `remove` | $O(d)$ |

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

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

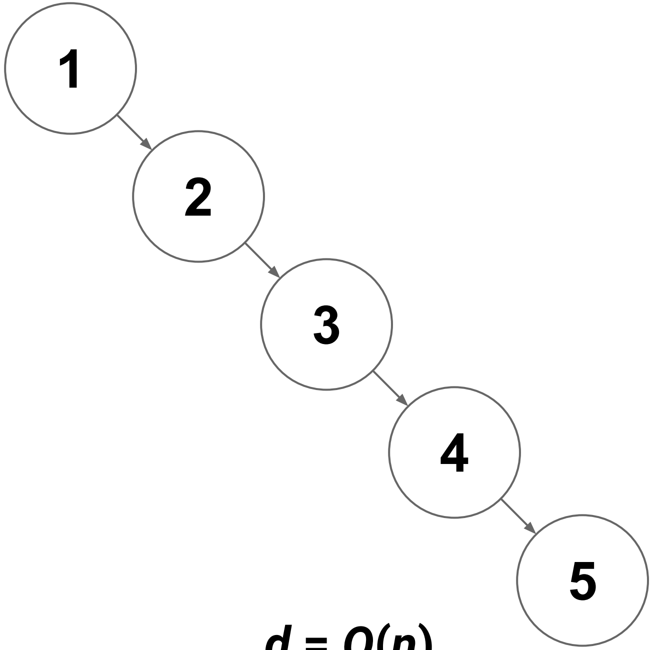
# Tree Depth vs Size

**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(right) - height(left)| ≤ 1**

# Balanced Trees

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

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

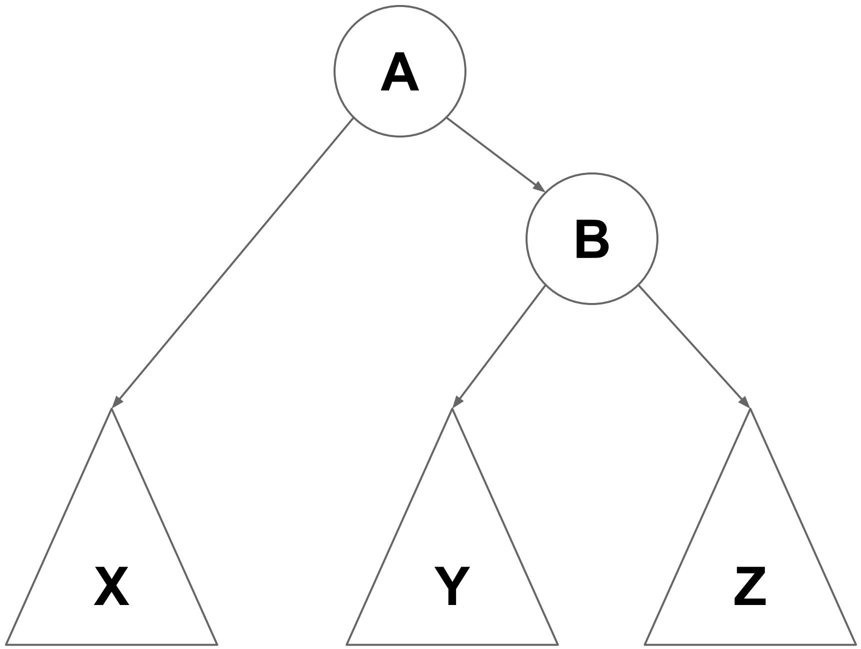
**Option 2**

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

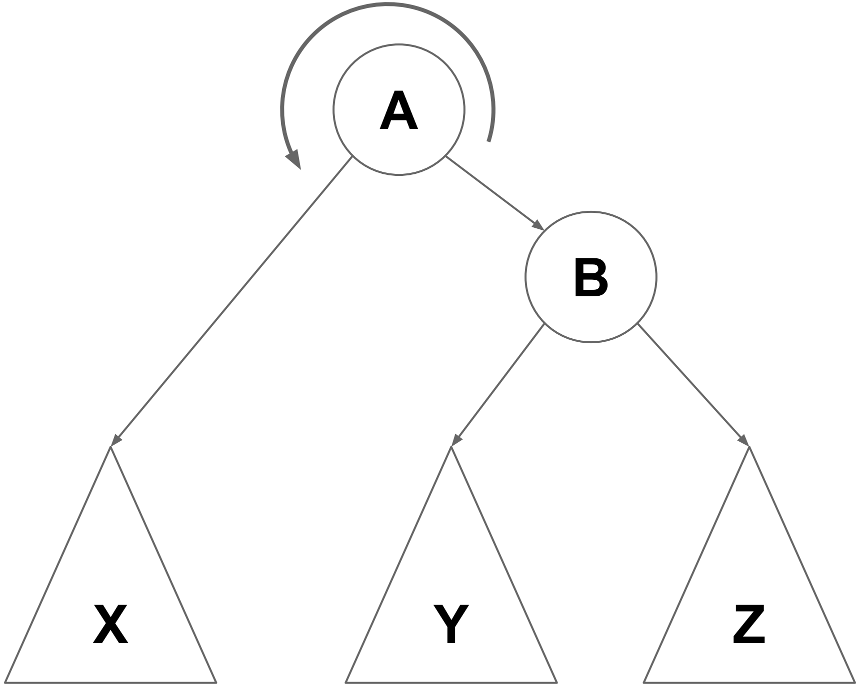
(Add a color to each node marking it as "red" or "black")

# Ok…but how do we enforce this…?

# Rebalancing Trees (rotations)

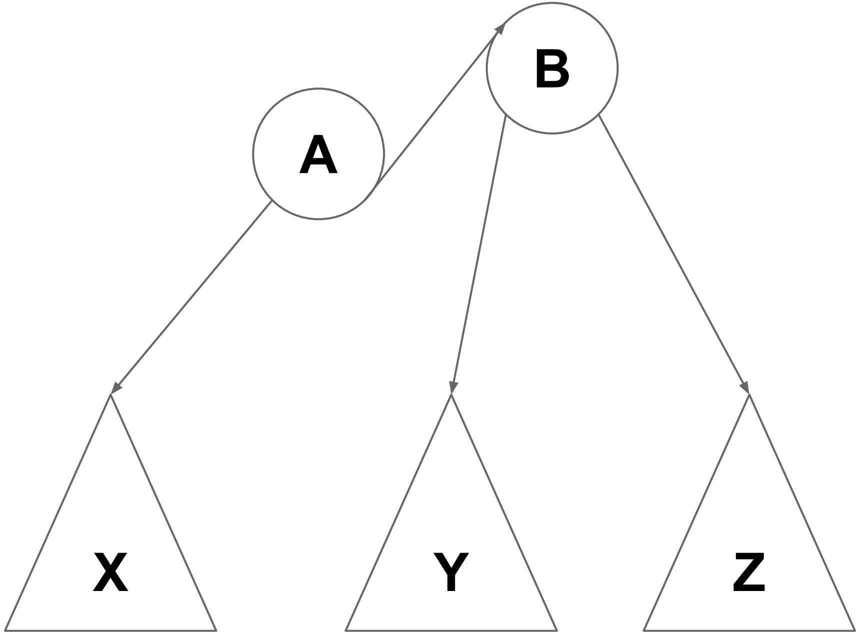


# Rebalancing Trees (rotations)




**Rotate(A, B)**

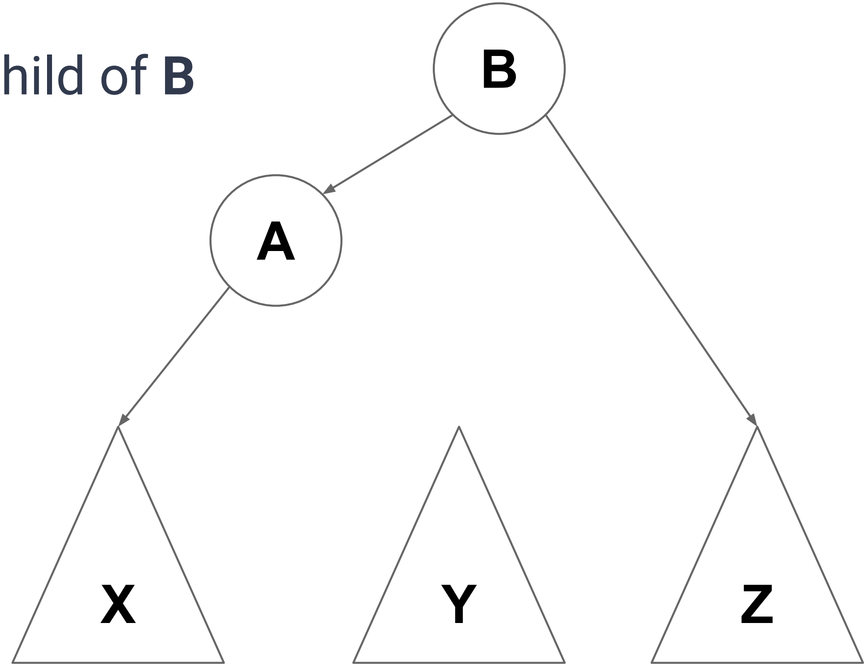
# Rebalancing Trees (rotations)




**Rotate(A, B)**

# Rebalancing Trees (rotations)

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

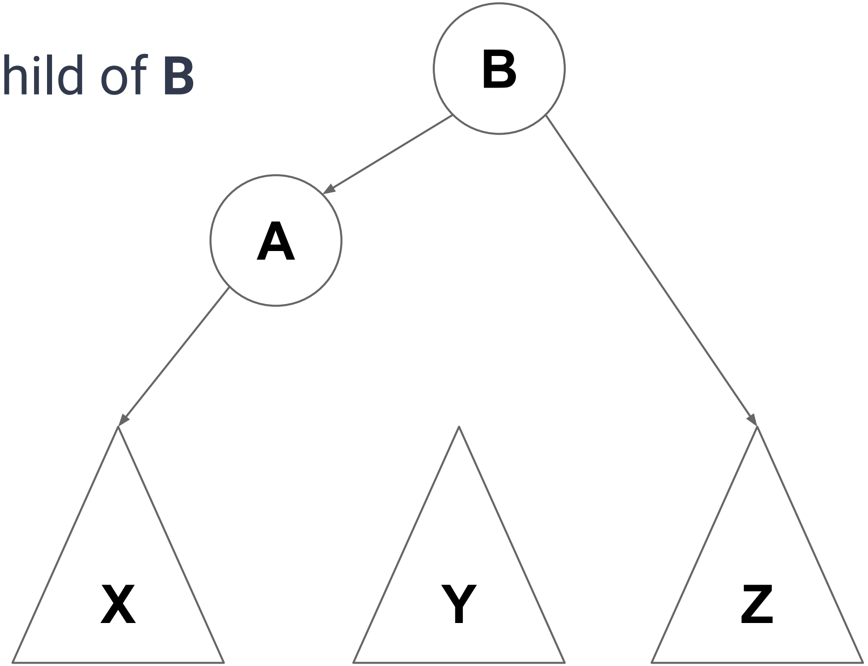



**Rotate(A, B)**

# Rebalancing Trees (rotations)

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

What about **Y**?



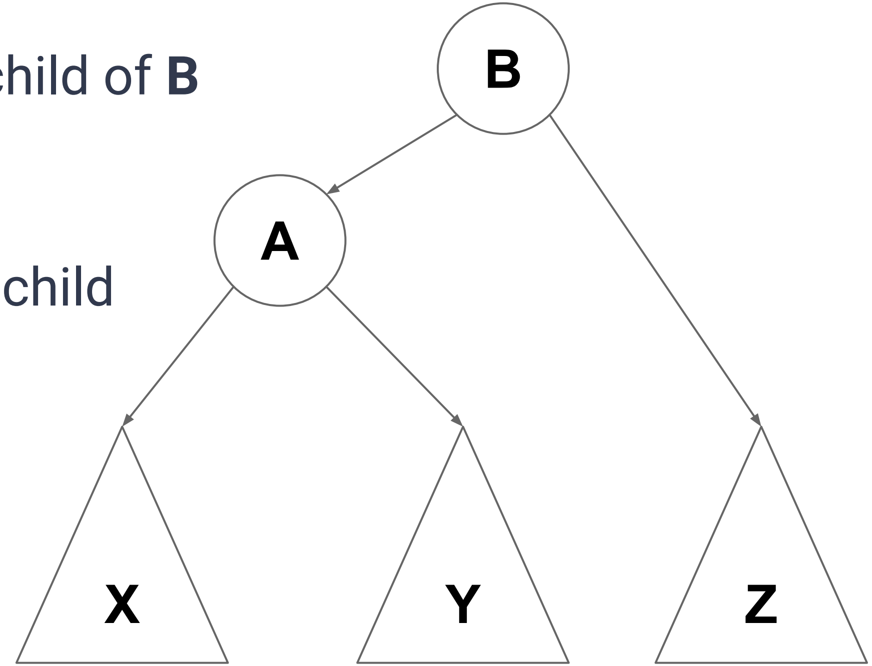

**Rotate(A, B)**

# Rebalancing Trees (rotations)

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

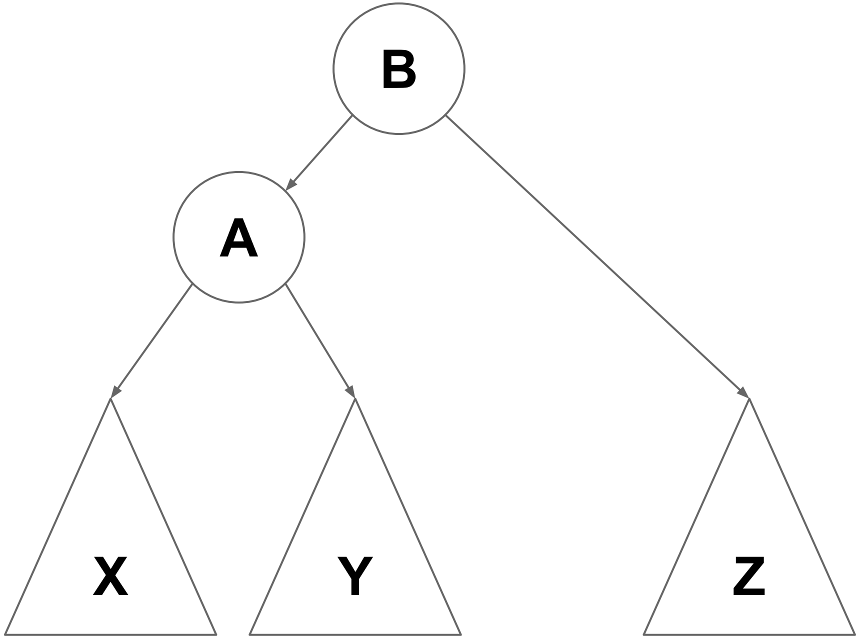
What about **Y**?

Make it the right child of **A**




**Rotate(A, B)**

# Rebalancing Trees (rotations)

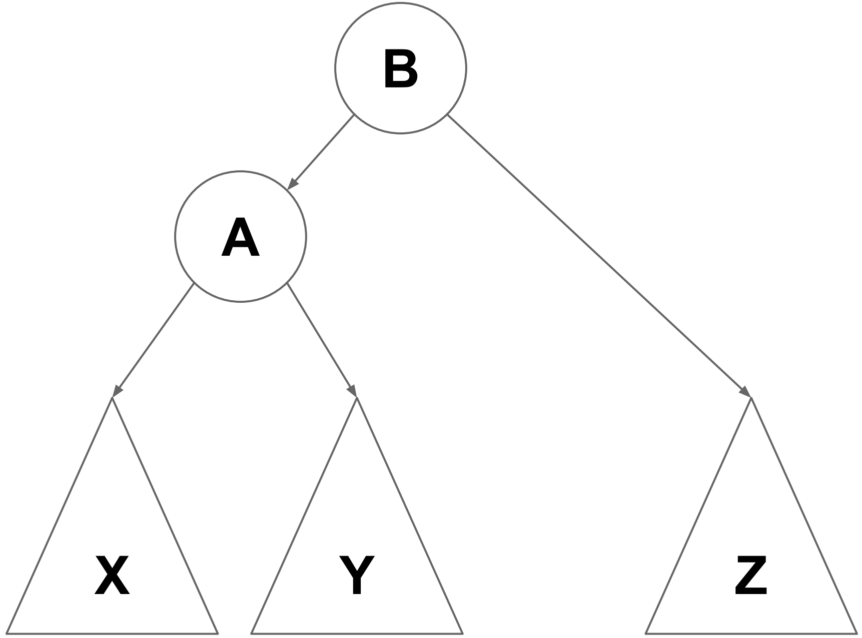



**Rotate(A, B)**

# Rebalancing Trees (rotations)

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

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



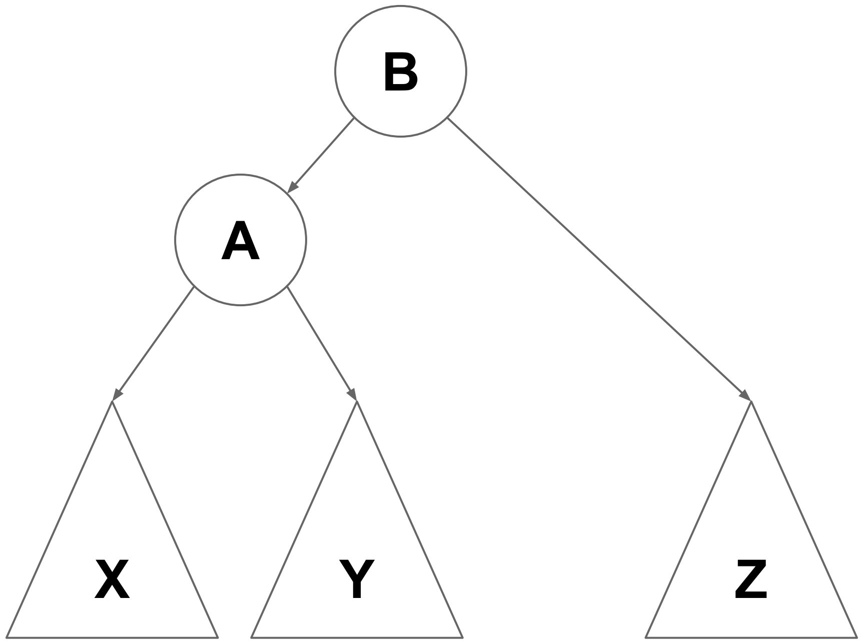

**Rotate(A, B)**

# Rebalancing Trees (rotations)

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

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

*Is ordering maintained?*



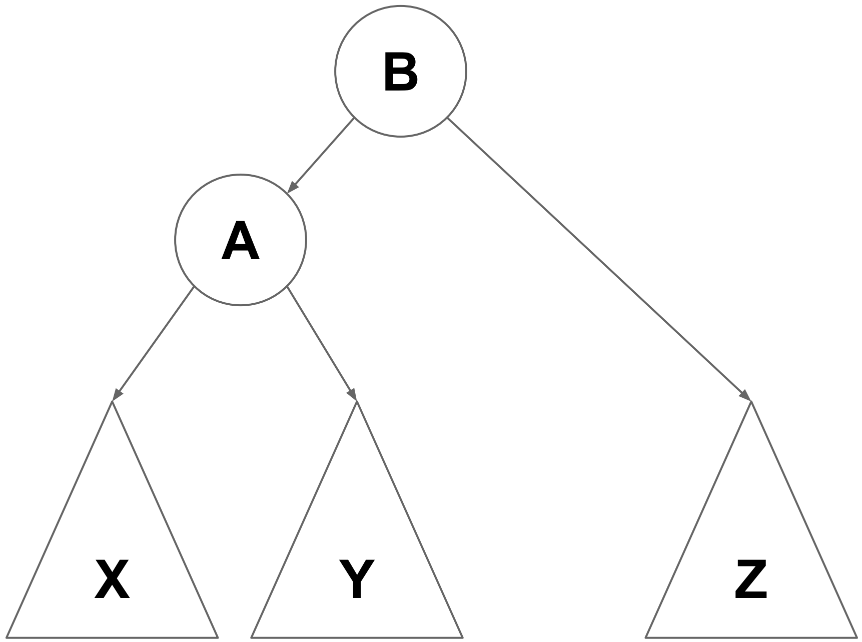

**Rotate(A, B)**

# 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

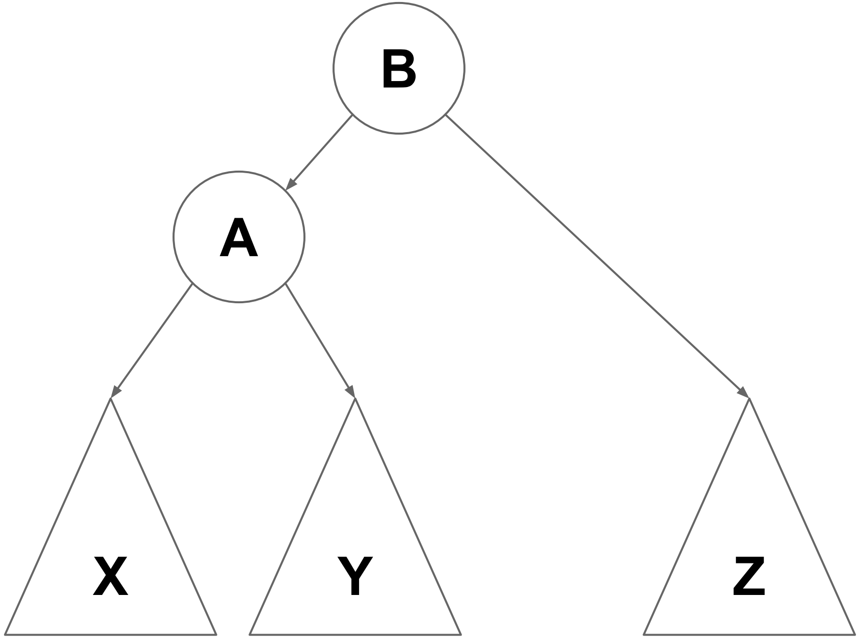
**B**'s left child became **A**'s right child

*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

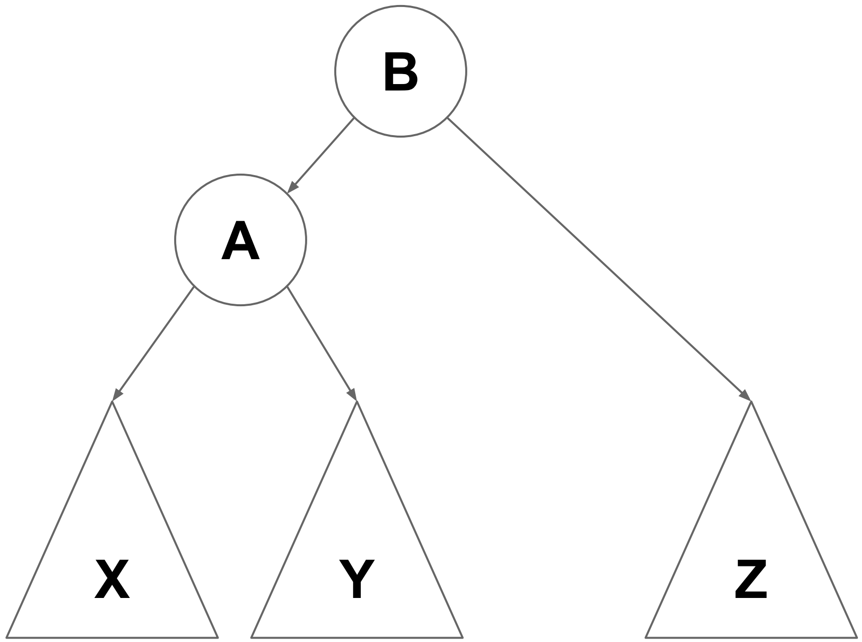
**B**'s left child became **A**'s right child

*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)

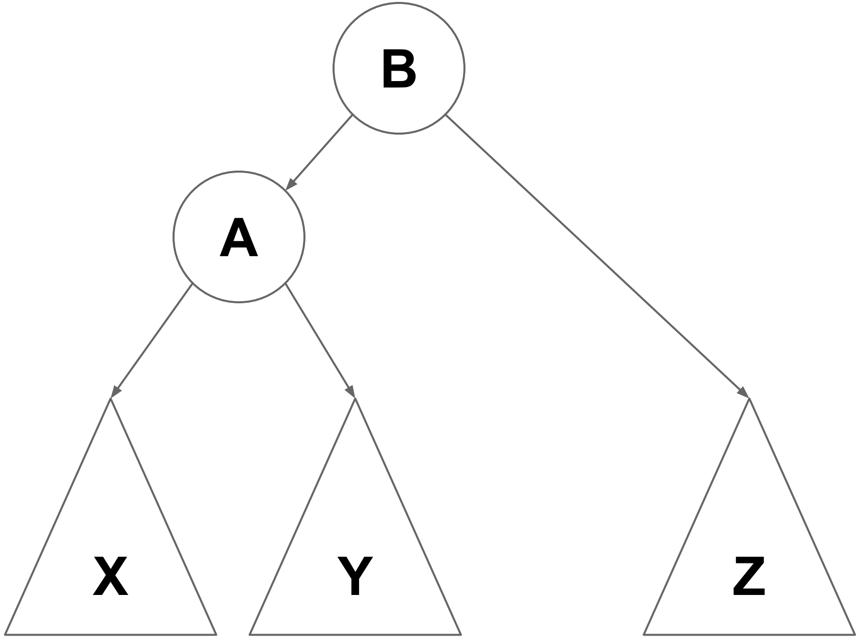
**A** became **B**'s left child

**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)**

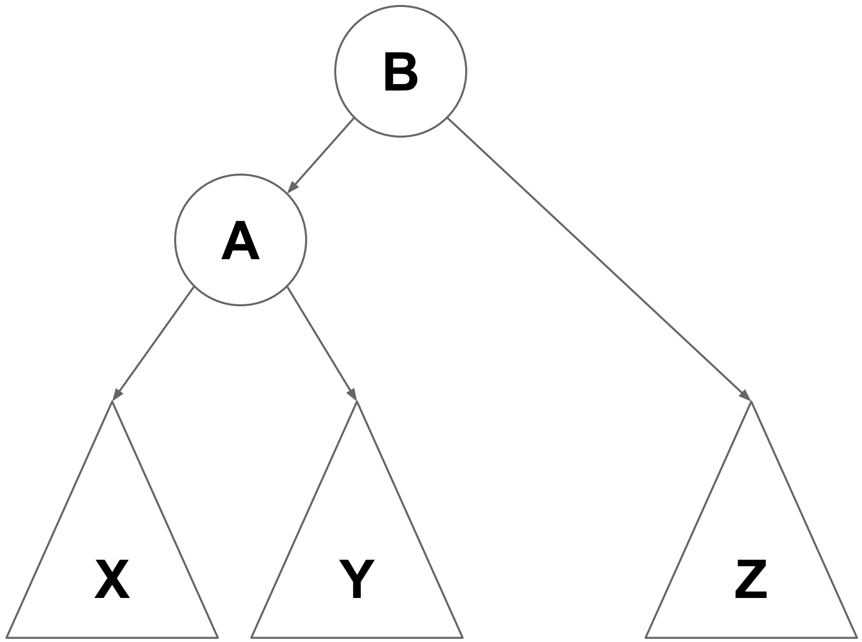
# Rebalancing Trees (rotations)

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

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

*Is ordering maintained?* **Yes!**

*Complexity?*




**Rotate(A, B)**

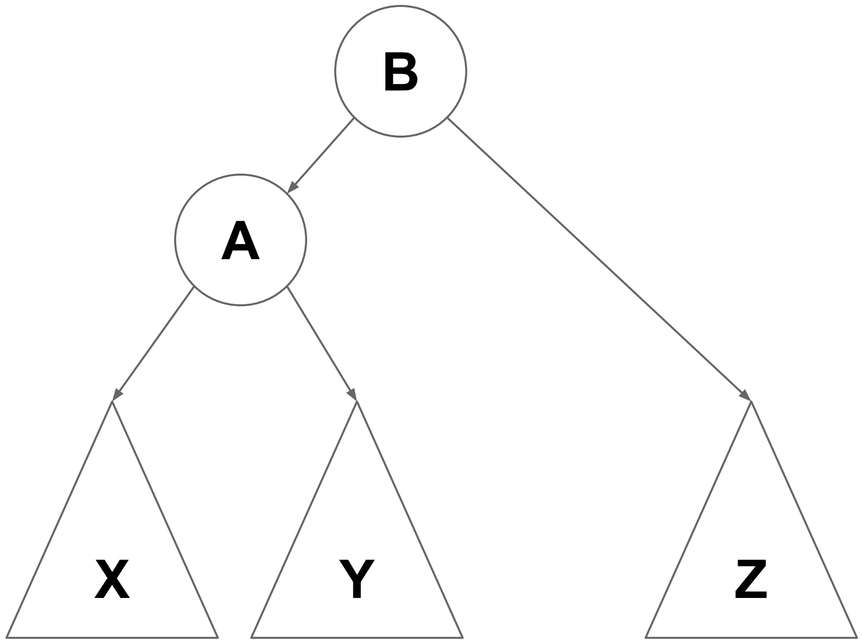
# Rebalancing Trees (rotations)

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

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

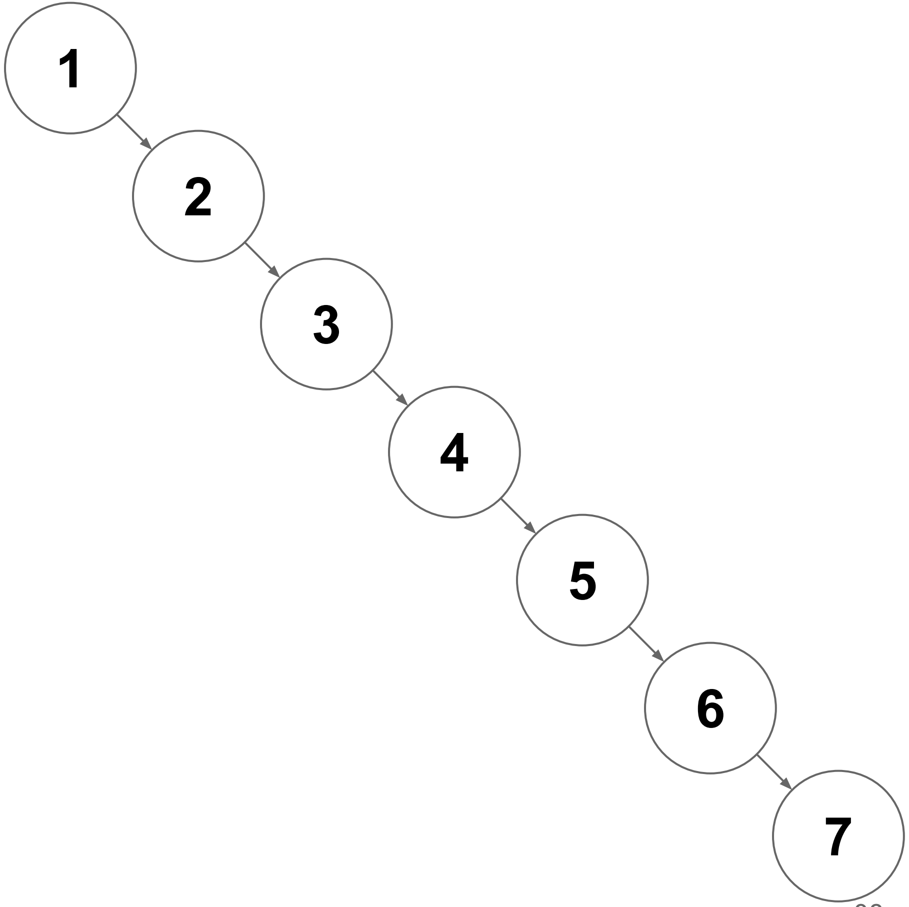
*Is ordering maintained?* **Yes!**

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



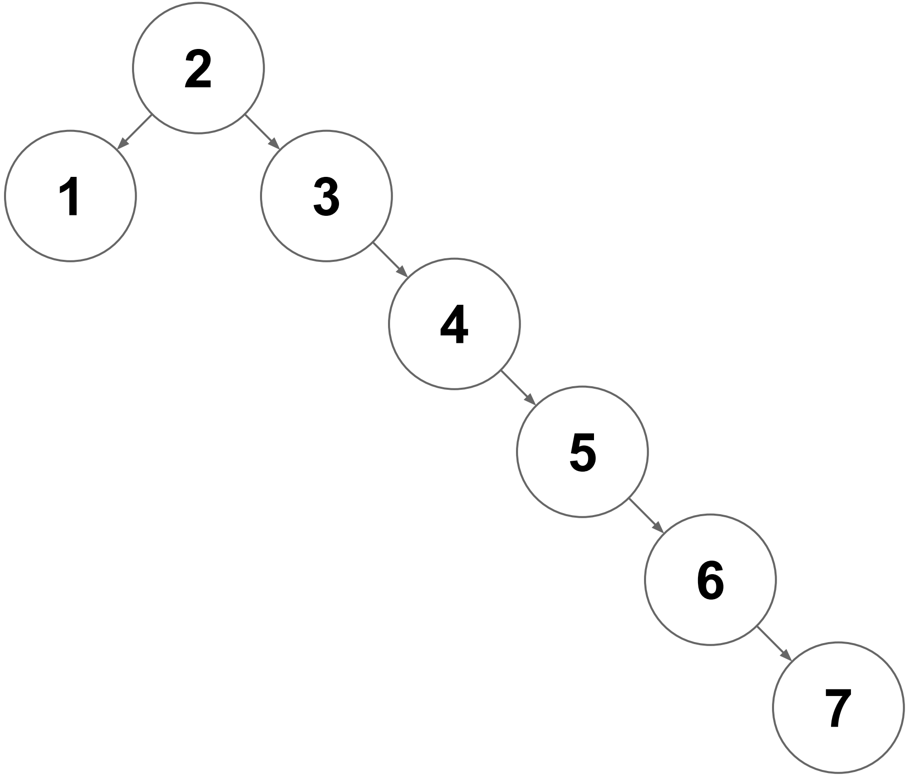

**Rotate(A, B)**

# Rebalancing Trees



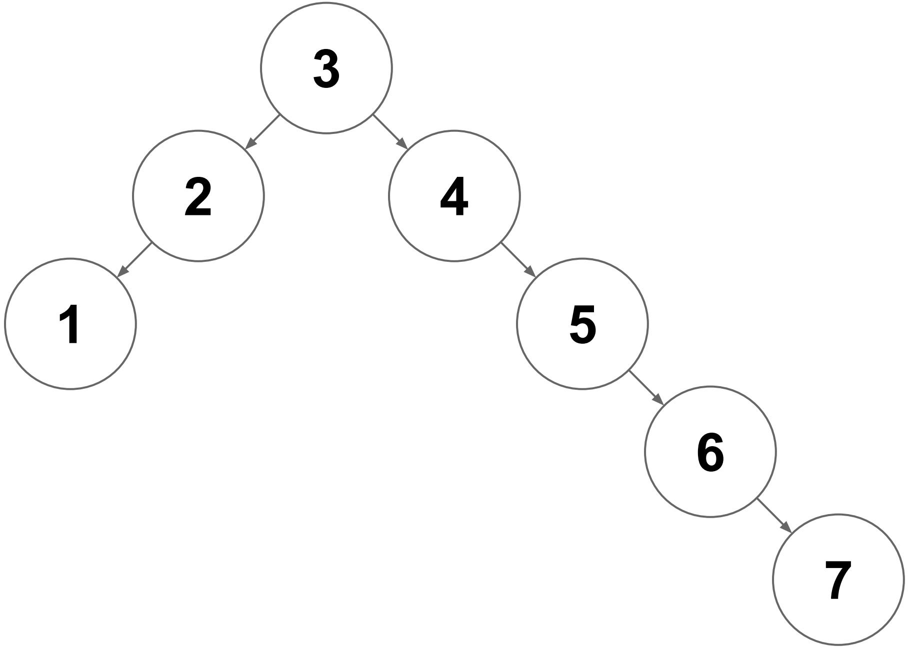
# Rebalancing Trees

`Rotate(1,2)`



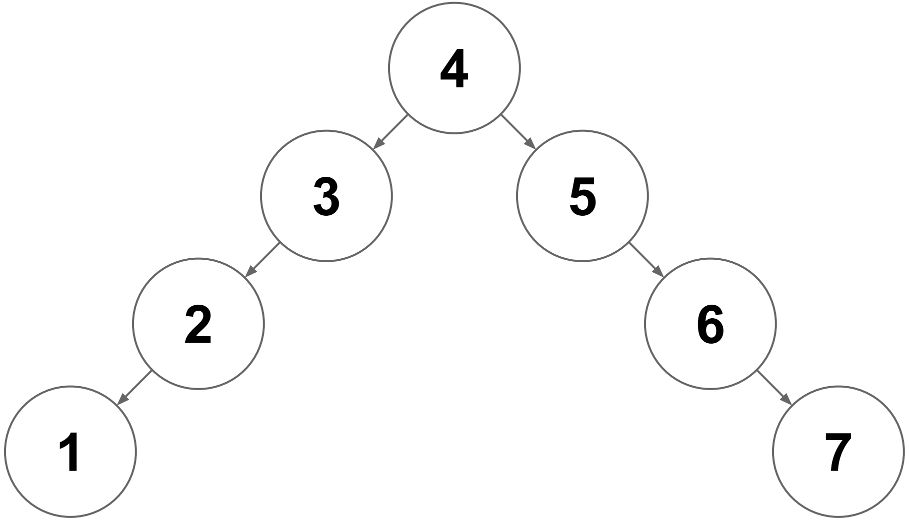
# Rebalancing Trees

`Rotate(2,3)`



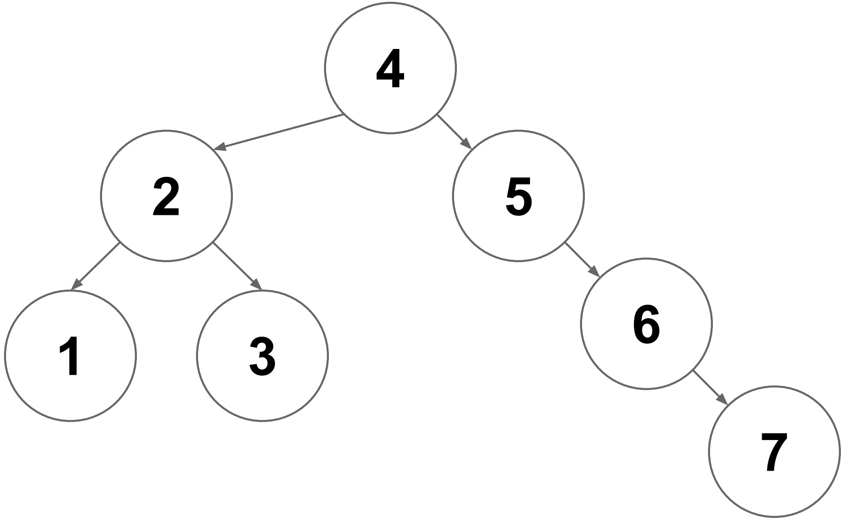
# Rebalancing Trees

`Rotate(3,4)`



# Rebalancing Trees

`Rotate(3,2)`



# Rebalancing Trees

`Rotate(5,6)`