CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu

Dr. Oliver Kennedy
okennedy@buffalo.edu

212 Capen Hall

Day 25
Traversing and Balancing Trees

Sets

A **Set** is an **unordered** collection of **unique** elements.

(order doesn't matter, and at most one copy of each item)

Sets

A **Set** is an **unordered** collection of **unique** elements.

(order doesn't matter, and at most one copy of each ~~item~~ key)

The `mutable.Set[T]` ADT

`add(element: T): Unit`

Store one copy of **`element`** if not already present

`apply(element: T): Boolean`

Return true if **`element`** is present in the set

`remove(element: T): Boolean`

Remove **`element`** if present, or return false if not

Bags

A **Bag** is an **unordered** collection of **non-unique** elements.

(order doesn't matter, and multiple copies with the same key is OK)

The `mutable.Bag[T]` ADT

`add(element: T): Unit`

Register the presence of a new (copy of) **`element`**

`apply(element: T): Boolean`

Return the number of copies of **`element`** in the bag

`remove(element: T): Boolean`

Remove one copy of **`element`** if present, or return false if not

Collection ADTs

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

(Rooted) Trees

(Even More) Tree Terminology

Rooted, Directed Tree - Has a single root node (node with no parents)

Parent of node X - A node with an out-edge to X (max 1 parent per node)

Child of node X - A node with an in-edge from X

Leaf - A node with no children

Depth of node X - The number of edges in the path from the root to X

Height of node X - The number of edges in the path from X to the deepest leaf

(Even More) Tree Terminology

Level of a node - Depth of the node + 1

Size of a tree (*n*) - The number of nodes in the tree

Height/Depth of a tree (*d*) - Height of the root/depth of the deepest leaf

(Even More) Tree Terminology

Binary Tree - Every vertex has at most 2 children

Complete Binary Tree - All leaves are in the deepest two levels

Full Binary Tree - All leaves are at the deepest level, therefore every vertex has exactly 0 or 2 children, and $\boldsymbol{d = \log(n)}$

Computing Tree Height

The height of a tree is the height of the root

The children of the root are each roots of the left and right subtrees

So we can compute height recursively:

$$h(root) = \begin{cases} 0 & \textbf{if the tree is empty} \\ 1 + max(h(\texttt{root.left}), h(\texttt{root.right})) & \textbf{otherwise} \end{cases}$$

Computing Tree Height

```scala
def height[T](root: Tree[T]): Int = {
  root match {
    case EmptyTree =>
      0

    case TreeNode(v, left, right) =>
      1 + Math.max( height(left), height(right) )
}
```

$$h(root) = \begin{cases} 0 & \text{if the tree is empty} \\ 1 + max(h(\texttt{root.left}), h(\texttt{root.right})) & \text{otherwise} \end{cases}$$

Computing Tree Height

```scala
def height[T](root: Tree[T]): Int = {
  root match {
    case EmptyTree =>
      0

    case TreeNode(v, left, right) =>
      1 + Math.max( height(left), height(right) )
}
```

Case classes have a nice mapping onto functions with multiple cases

$$h(root) = \begin{cases} 0 & \textbf{if the tree is empty} \\ 1 + max(h(\texttt{root.left}), h(\texttt{root.right})) & \textbf{otherwise} \end{cases}$$

Binary Search Tree

A **Binary Search Tree** is a **Binary Tree** in which each node stores a unique key, and the keys are ordered.

Binary Search Tree

A **Binary Search Tree** is a **Binary Tree** in which each node stores a unique key, and the keys are ordered.

Constraints

Binary Search Tree

A **Binary Search Tree** is a **Binary Tree** in which each node stores a unique key, and the keys are ordered.

Constraints

- No duplicate keys

Binary Search Tree

A **Binary Search Tree** is a **Binary Tree** in which each node stores a unique key, and the keys are ordered.

Constraints

- No duplicate keys
- For every node $\boldsymbol{X_L}$ in the left subtree of node $\boldsymbol{X}$: $\boldsymbol{X_L}$`.key` < $\boldsymbol{X}$`.key`

Binary Search Tree

A **Binary Search Tree** is a **Binary Tree** in which each node stores a unique key, and the keys are ordered.

Constraints

- No duplicate keys
- For every node $\boldsymbol{X_L}$ in the left subtree of node $\boldsymbol{X}$: $\boldsymbol{X_L}$`.key` < $\boldsymbol{X}$`.key`
- For every node $\boldsymbol{X_R}$ in the right subtree of node $\boldsymbol{X}$: $\boldsymbol{X_R}$`.key` > $\boldsymbol{X}$`.key`

Binary Search Tree

A **Binary Search Tree** is a **Binary Tree** in which each node stores a unique key, and the keys are ordered.

Constraints

- No duplicate keys
- For every node $\boldsymbol{X_L}$ in the left subtree of node $\boldsymbol{X}$: $\boldsymbol{X_L}$`.key` < $\boldsymbol{X}$`.key`
- For every node $\boldsymbol{X_R}$ in the right subtree of node $\boldsymbol{X}$: $\boldsymbol{X_R}$`.key` > $\boldsymbol{X}$`.key`

X **partitions** its children

Finding an Item

Goal: Find an item with key ***k*** in a BST rooted at **`root`**

Finding an Item

Goal: Find an item with key ***k*** in a BST rooted at `root`

1. Is `root` empty? (if yes, then the item is not here)

Finding an Item

Goal: Find an item with key ***k*** in a BST rooted at **`root`**

1. Is **`root`** empty? (if yes, then the item is not here)
2. Does **`root.value`** have key ***k***? (if yes, done!)

Finding an Item

Goal: Find an item with key ***k*** in a BST rooted at **`root`**

1. Is **`root`** empty? (if yes, then the item is not here)
2. Does **`root.value`** have key ***k***? (if yes, done!)
3. Is ***k*** less than **`root.value`**'s key? (if yes, search left subtree)

Finding an Item

Goal: Find an item with key ***k*** in a BST rooted at **`root`**

1. Is **`root`** empty? (if yes, then the item is not here)
2. Does **`root.value`** have key ***k***? (if yes, done!)
3. Is ***k*** less than **`root.value`**'s key? (if yes, search left subtree)
4. Is ***k*** greater than **`root.value`**'s key? (If yes, search the right subtree)

find

```
def find[V: Ordering](root: BST[V], target: V): Option[V] =
  root match {
    case TreeNode(v, left, right) =>
      if(Ordering[V].lt( target, v ))      { return find(left, target) }
      else if(Ordering[V].lt( v, target )){ return find(right, target) }
      else                                 { return Some(v) }

    case EmptyTree =>
      return None
  }
```

find

```
def find[V: Ordering](root: BST[V], target: V): Option[V] =
  root match {
    case TreeNode(v, left, right) =>
      if(Ordering[V].lt( target, v ))       { return find(left, target) }
      else if(Ordering[V].lt( v, target )){ return find(right, target) }
      else                                 { return Some(v) }

    case EmptyTree =>
      return None
  }
```

What's the complexity?

find

```
def find[V: Ordering](root: BST[V], target: V): Option[V] =
  root match {
    case TreeNode(v, left, right) =>
      if(Ordering[V].lt( target, v ))      { return find(left, target) }
      else if(Ordering[V].lt( v, target )){ return find(right, target) }
      else                                 { return Some(v) }

    case EmptyTree =>
      return None
  }
```

What's the complexity? (how many times do we call **`find`***)?*

find

```scala
def find[V: Ordering](root: BST[V], target: V): Option[V] =
  root match {
    case TreeNode(v, left, right) =>
      if(Ordering[V].lt( target, v ))      { return find(left, target) }
      else if(Ordering[V].lt( v, target )){ return find(right, target) }
      else                                 { return Some(v) }

    case EmptyTree =>
      return None
  }
```

What's the complexity? (how many times do we call `find`)? ***O(d)***

Inserting an Item

Goal: Insert a new tem with key ***k*** in a BST rooted at `root`

Inserting an Item

Goal: Insert a new tem with key ***k*** in a BST rooted at **`root`**

1. Is **`root`** empty? (insert here)

Inserting an Item

Goal: Insert a new tem with key ***k*** in a BST rooted at **`root`**

1. Is **`root`** empty? (insert here)
2. Does **`root.value`** have key ***k***? (already present! don't insert)

Inserting an Item

Goal: Insert a new tem with key ***k*** in a BST rooted at **`root`**

1. Is **`root`** empty? (insert here)
2. Does **`root.value`** have key ***k***? (already present! don't insert)
3. Is ***k*** less than **`root.value`**'s key? (call insert on left subtree)

Inserting an Item

Goal: Insert a new tem with key ***k*** in a BST rooted at `root`

1. Is `root` empty? (insert here)
2. Does `root.value` have key ***k***? (already present! don't insert)
3. Is ***k*** less than `root.value`'s key? (call insert on left subtree)
4. Is ***k*** greater than `root.value`'s key? (call insert on right subtree)

insert

```
def insert[V: Ordering](root: BST[V], value: V): BST[V] =
  node match {
    case TreeNode(v, left, right) =>
      if(Ordering[V].lt( target, v ) ){
        return TreeNode(v, insert(left, target), right)
      } else if(Ordering[V].lt( v, target ) ){
        return TreeNode(v, left, insert(right, target))
      } else {
        return node // already present
      }

    case EmptyTree =>
      return TreeNode(value, EmptyTree, EmptyTree)
  }
```

insert

```
def insert[V: Ordering](root: BST[V], value: V): BST[V] =
  node match {
    case TreeNode(v, left, right) =>
      if(Ordering[V].lt( target, v ) ){
        return TreeNode(v, insert(left, target), right)
      } else if(Ordering[V].lt( v, target ) ){
        return TreeNode(v, left, insert(right, target))
      } else {
        return node // already present
      }

    case EmptyTree =>
      return TreeNode(value, EmptyTree, EmptyTree)
  }
```

What is the complexity?
(how many calls to `insert`)?

insert

```
def insert[V: Ordering](root: BST[V], value: V): BST[V] =
  node match {
    case TreeNode(v, left, right) =>
      if(Ordering[V].lt( target, v ) ){
        return TreeNode(v, insert(left, target), right)
      } else if(Ordering[V].lt( v, target ) ){
        return TreeNode(v, left, insert(right, target))
      } else {
        return node // already present
      }

    case EmptyTree =>
      return TreeNode(value, EmptyTree, EmptyTree)
  }
```

What is the complexity?
(how many calls to `insert`)? ***O*(*d*)**

Remove

Goal: Remove the item with key ***k*** from a BST rooted at **root**

1. `find` the item
2. Replace the found node with the right subtree
3. Insert the left subtree under the right

We'll look at this in more detail later, but for now…

What's the complexity? ***O*****(*****d*****)**

Sets and Bags

So we could use this specification of a BST to implement a Set

What about bags? How could we change our BST to implement a Bag?

Sets and Bags

So we could use this specification of a BST to implement a Set

What about bags? How could we change our BST to implement a Bag?

Idea 1: Allow multiple copies ($X_L \leq X$ instead of **<**)

Sets and Bags

So we could use this specification of a BST to implement a Set

What about bags? How could we change our BST to implement a Bag?

Idea 1: Allow multiple copies ($X_L \leq X$ instead of **<**)

Idea 2: Only store one copy of each element, but also store a count

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****?*

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*?* $O(n)$

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*?* $O(n)$

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

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

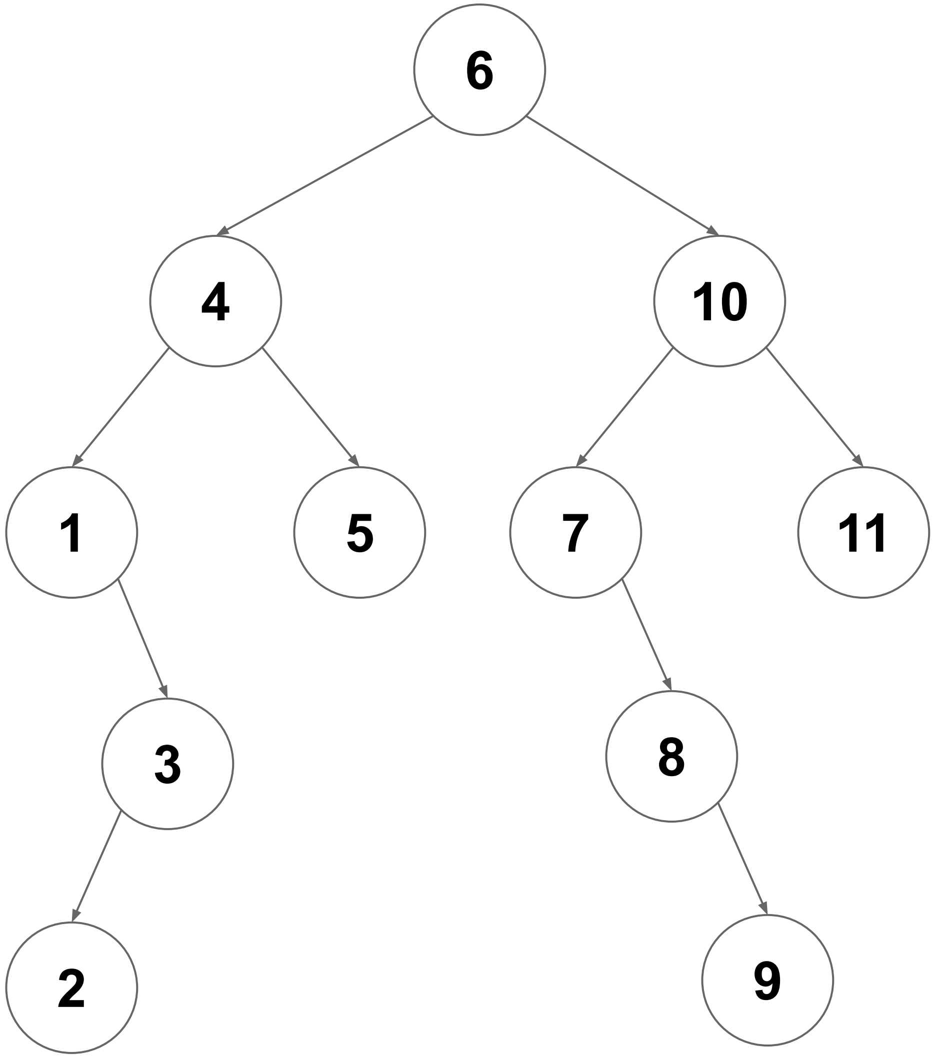
Post-Order (bottom-up)

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

Tree Traversal: In-Order

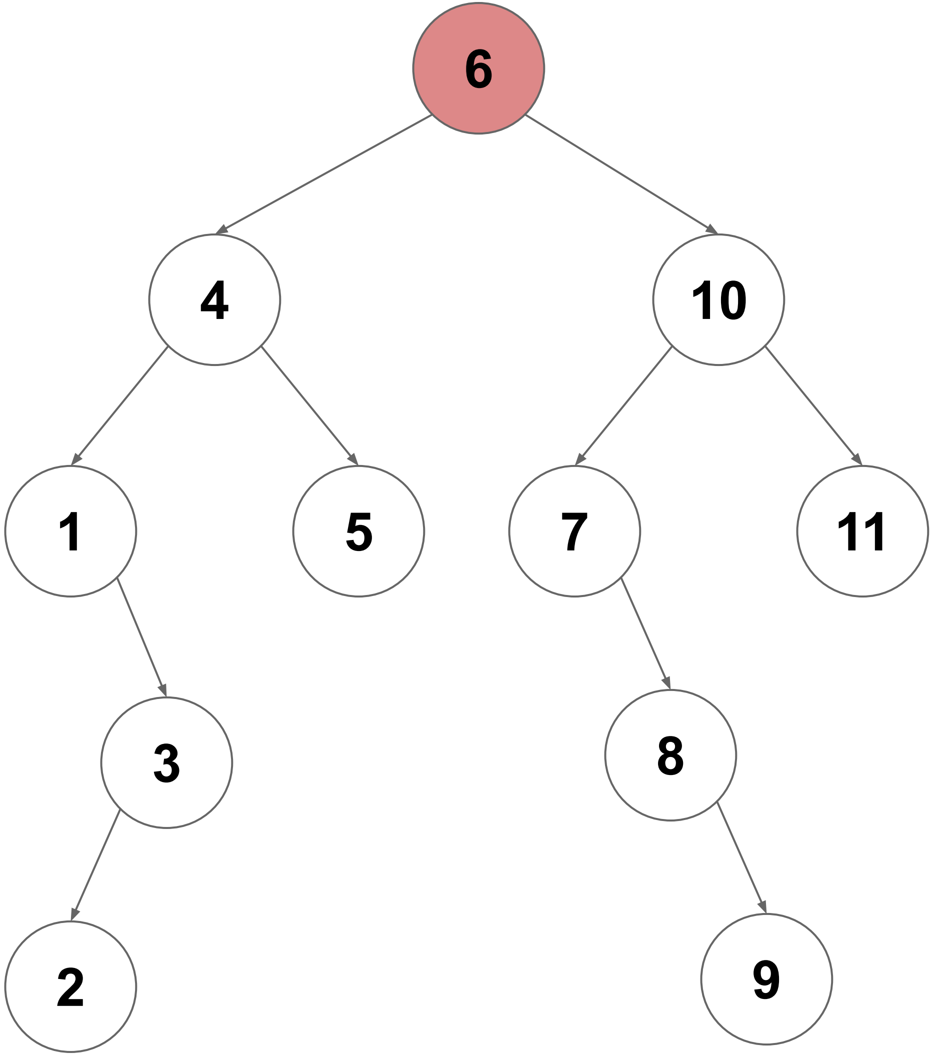
```
def inorderVisit[T](root: ImmutableTree[T], visit: ImmutableTree[T] => Unit) = {
  root match {
    case TreeNode(v, left, right) =>
      /* visit left */
      inorderVisit(left, visit)
      /* visit root */
      visit(v)
      /* visit right */
      inorderVisit(right, visit)

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

In-Order Traversal on a BST



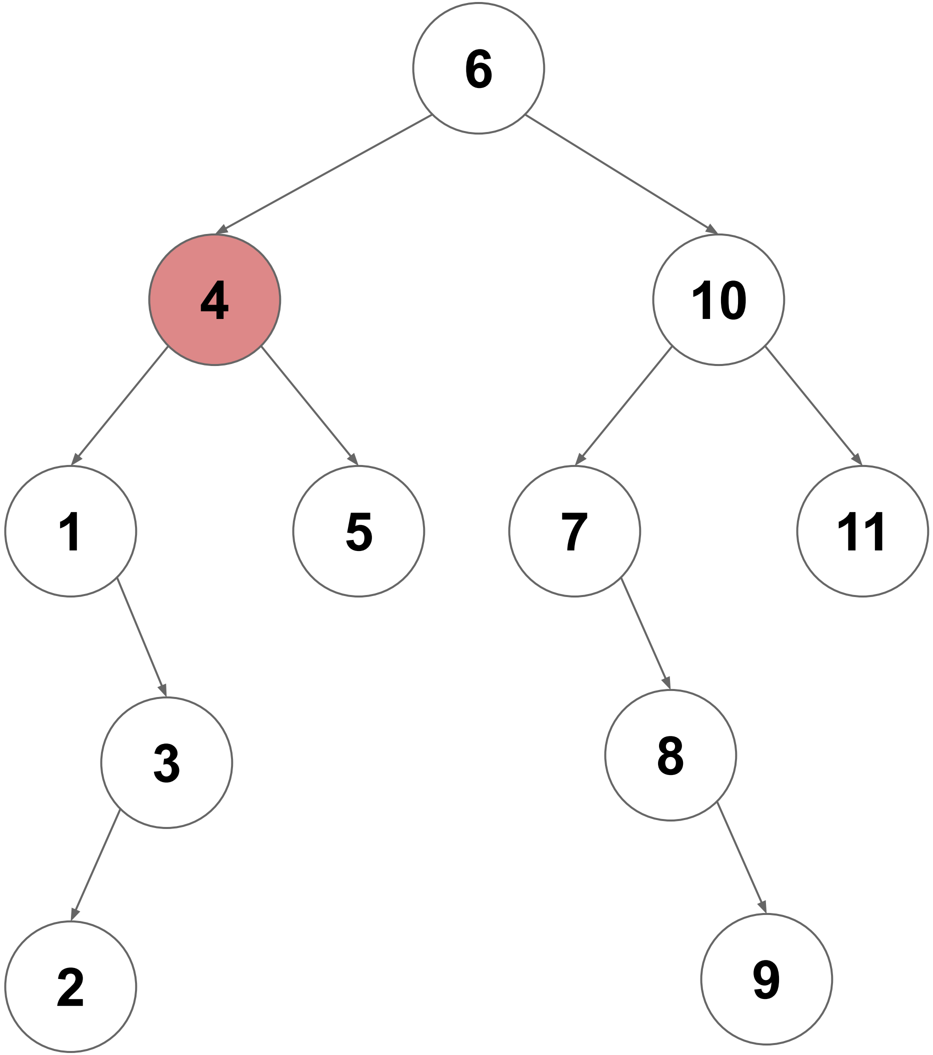
In-Order Traversal on a BST

`inorderVisit(6)`



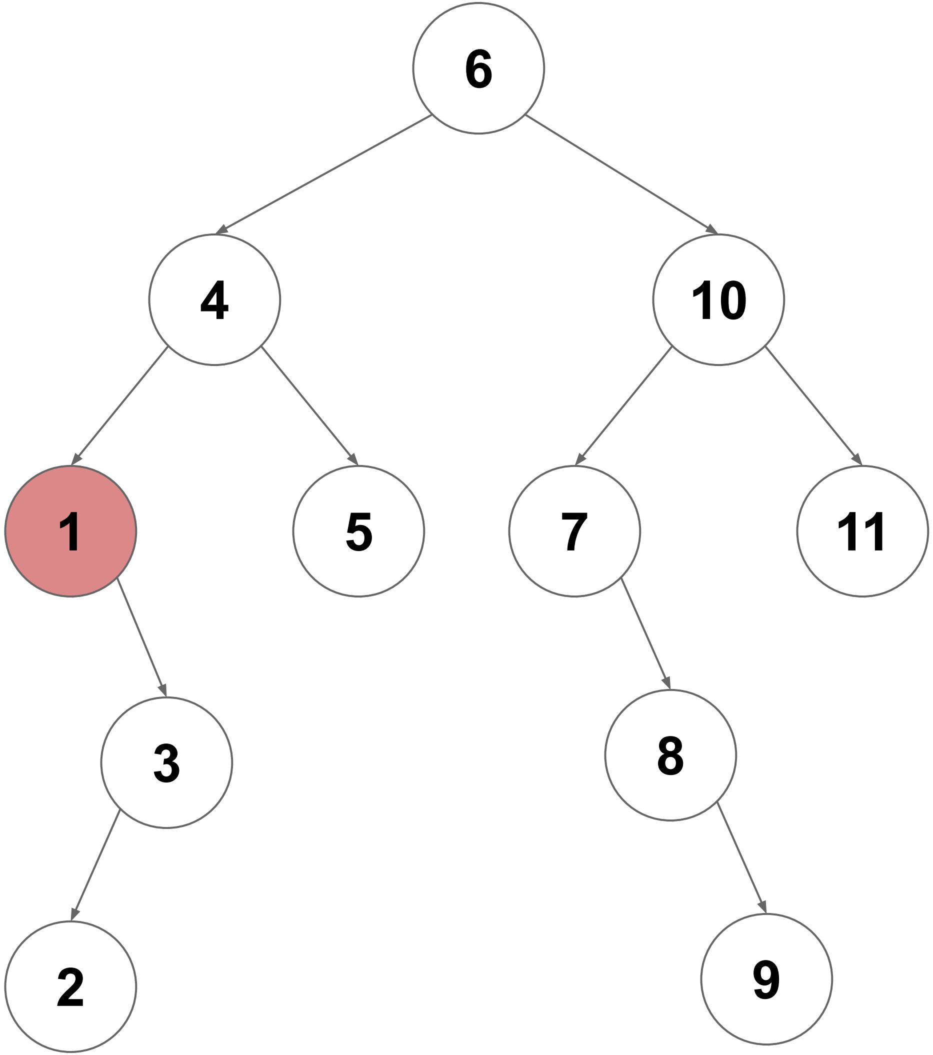
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(4)
```



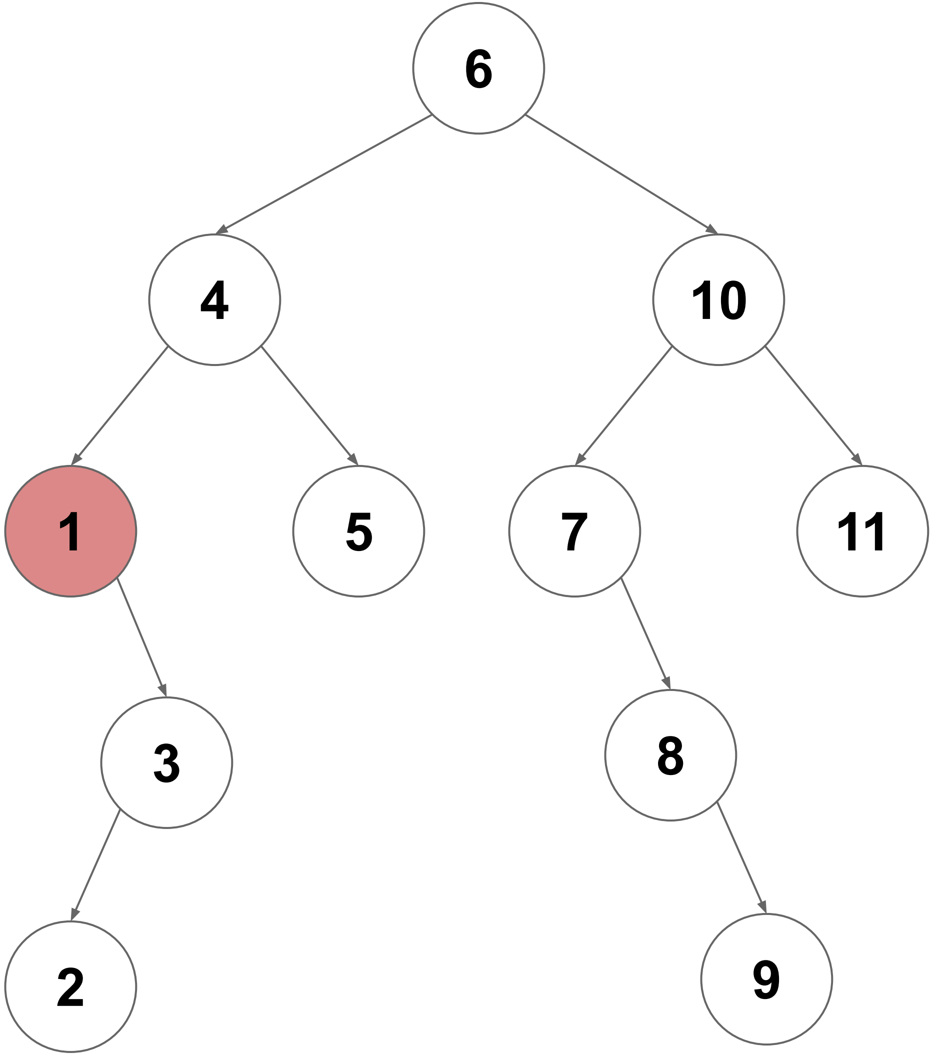
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(4)

inorderVisit(1)
```



In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(4)

inorderVisit(1)

inorderVisit(empty)
```



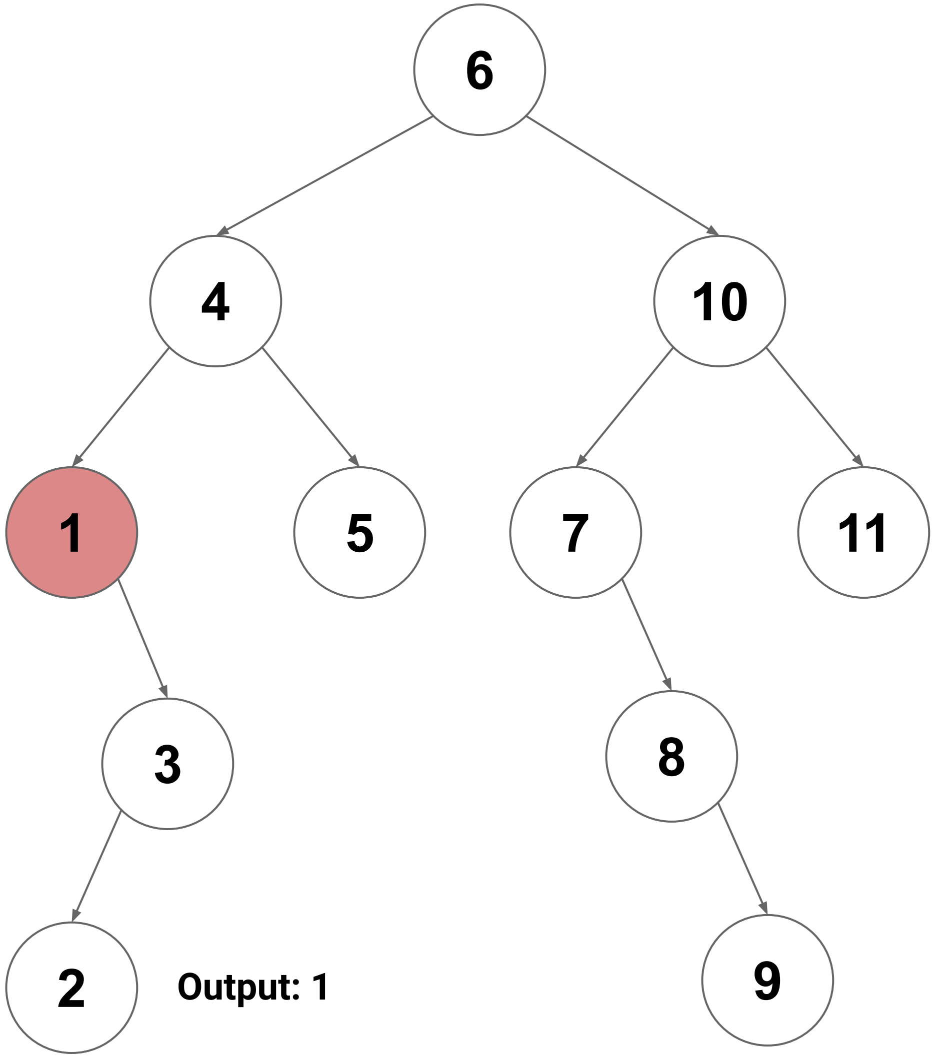
In-Order Traversal on a BST

`inorderVisit(6)`

`inorderVisit(4)`

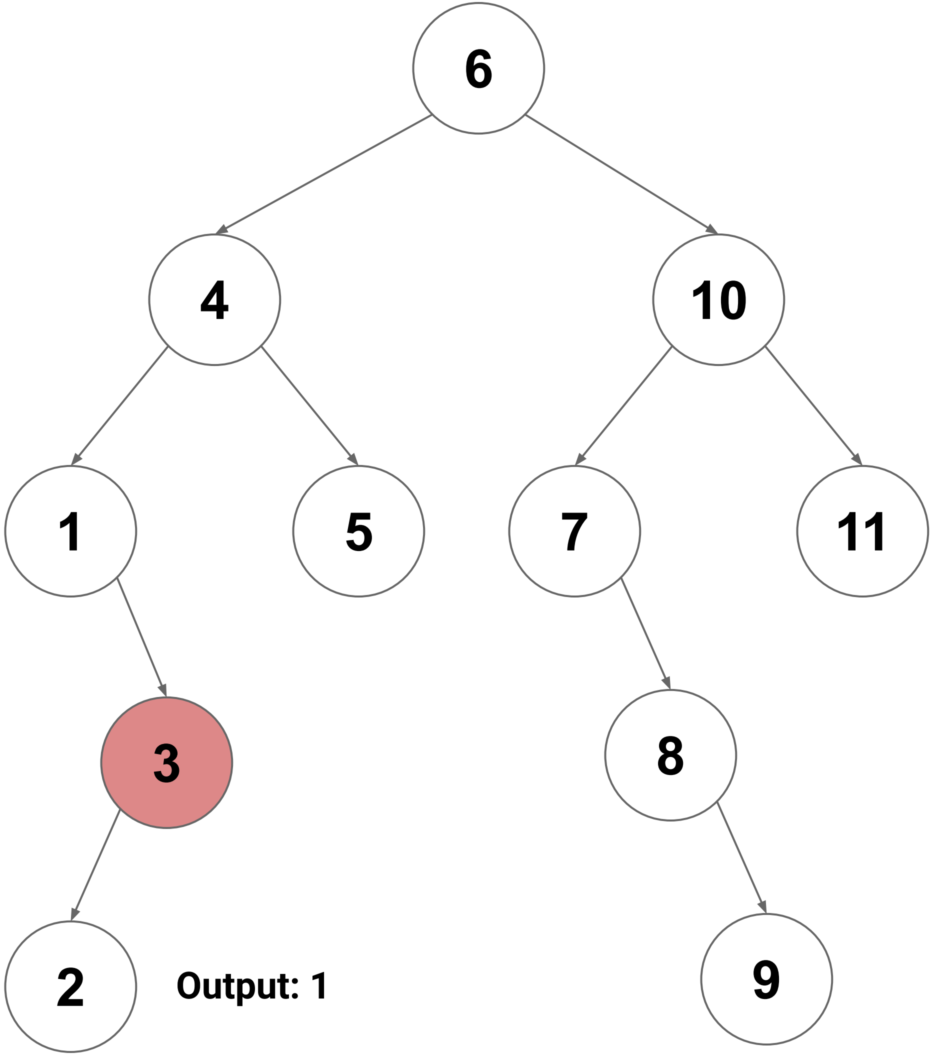
`inorderVisit(1)`

`visit(1)`



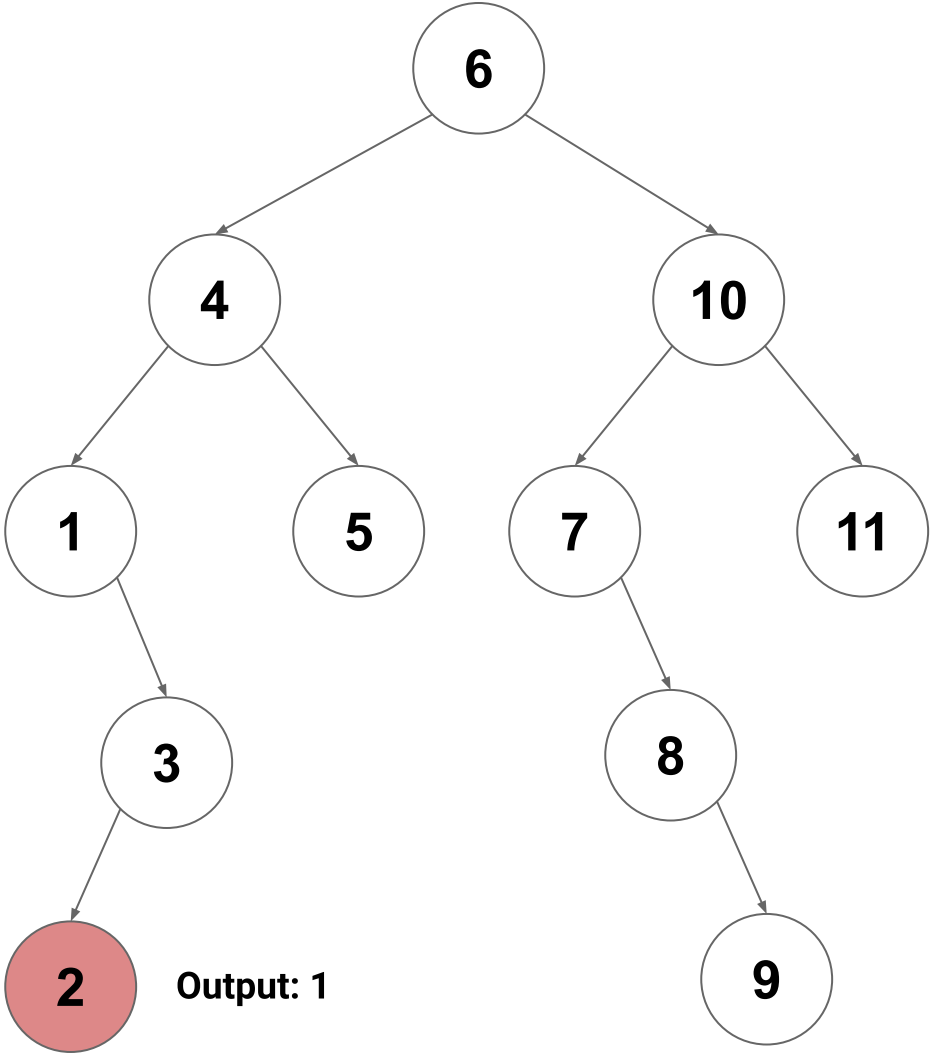
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(4)

inorderVisit(1)

inorderVisit(3)
```



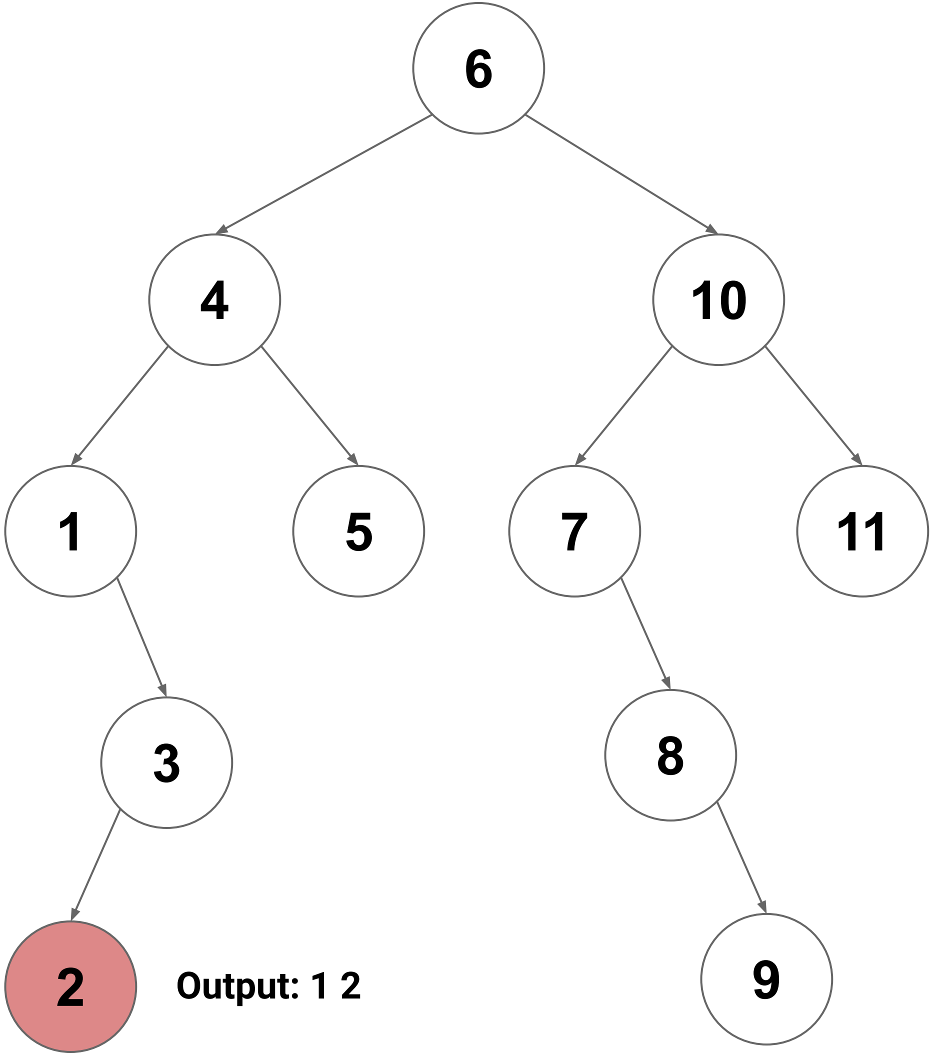
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(4)

inorderVisit(1)

inorderVisit(3)

inorderVisit(2)
```



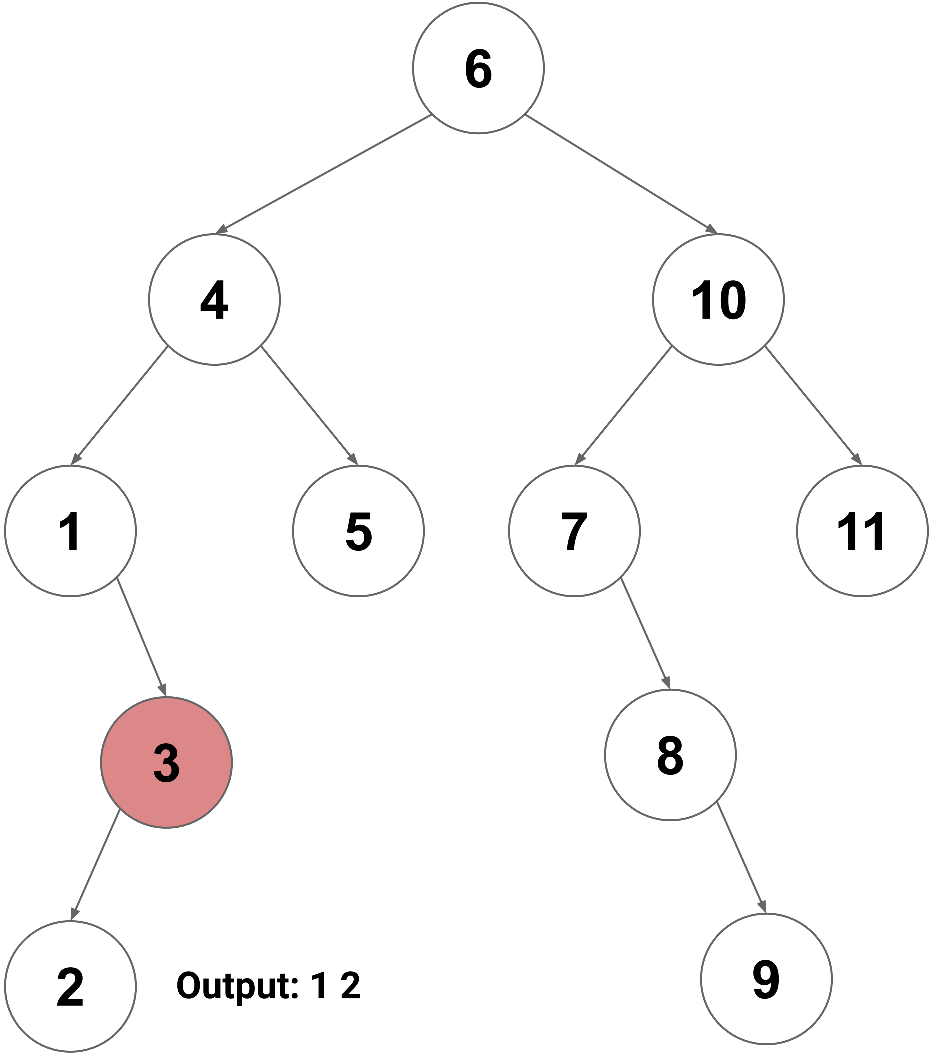
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(4)

inorderVisit(1)

inorderVisit(3)

visit(2)
```



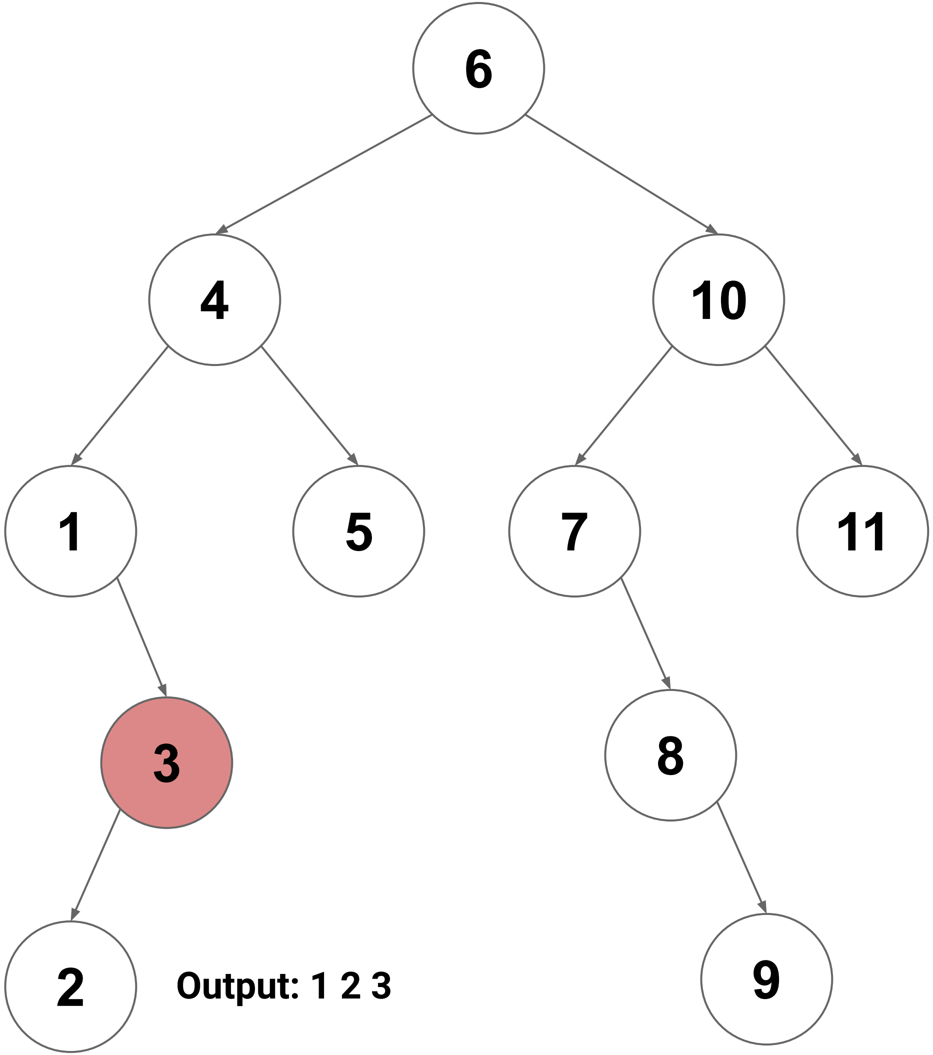
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(4)

inorderVisit(1)

inorderVisit(3)
```



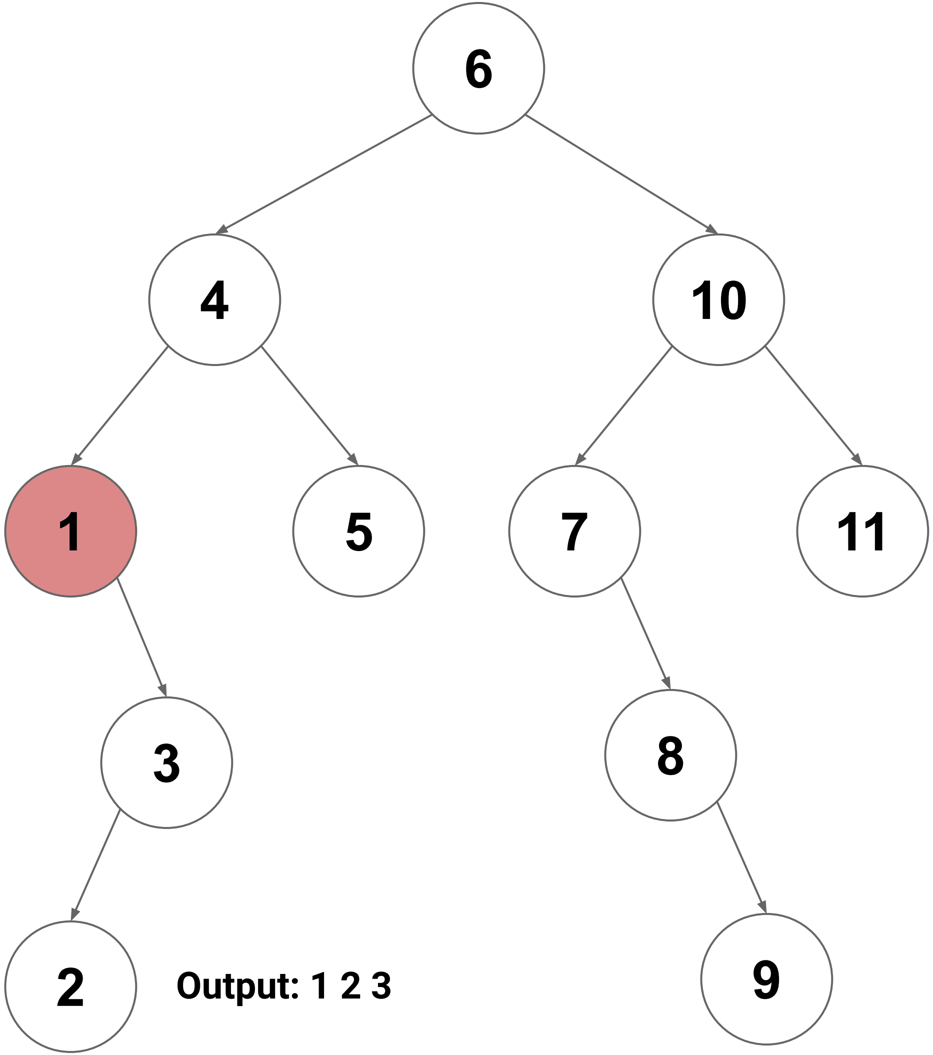
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(4)

inorderVisit(1)

visit(3)
```



In-Order Traversal on a BST

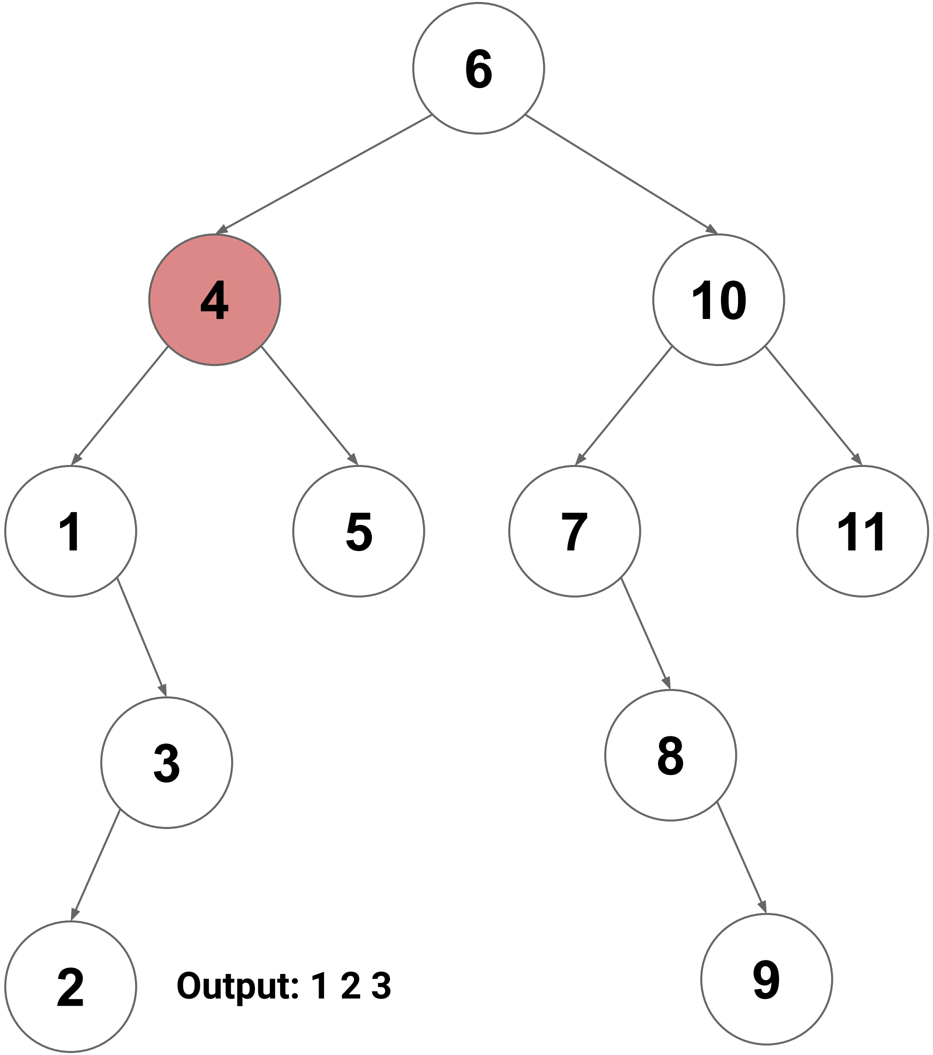
```
inorderVisit(6)

inorderVisit(4)

inorderVisit(1)
```



In-Order Traversal on a BST

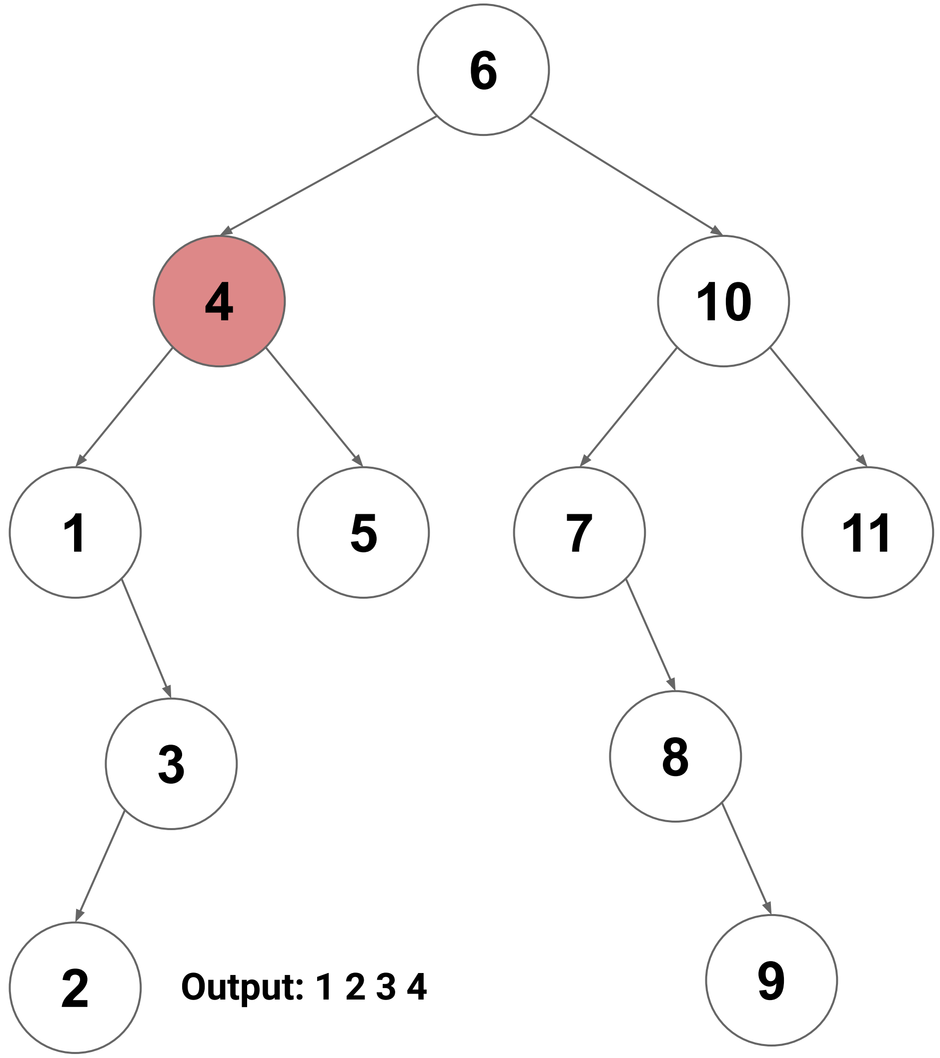
`inorderVisit(6)`

`inorderVisit(4)`



In-Order Traversal on a BST

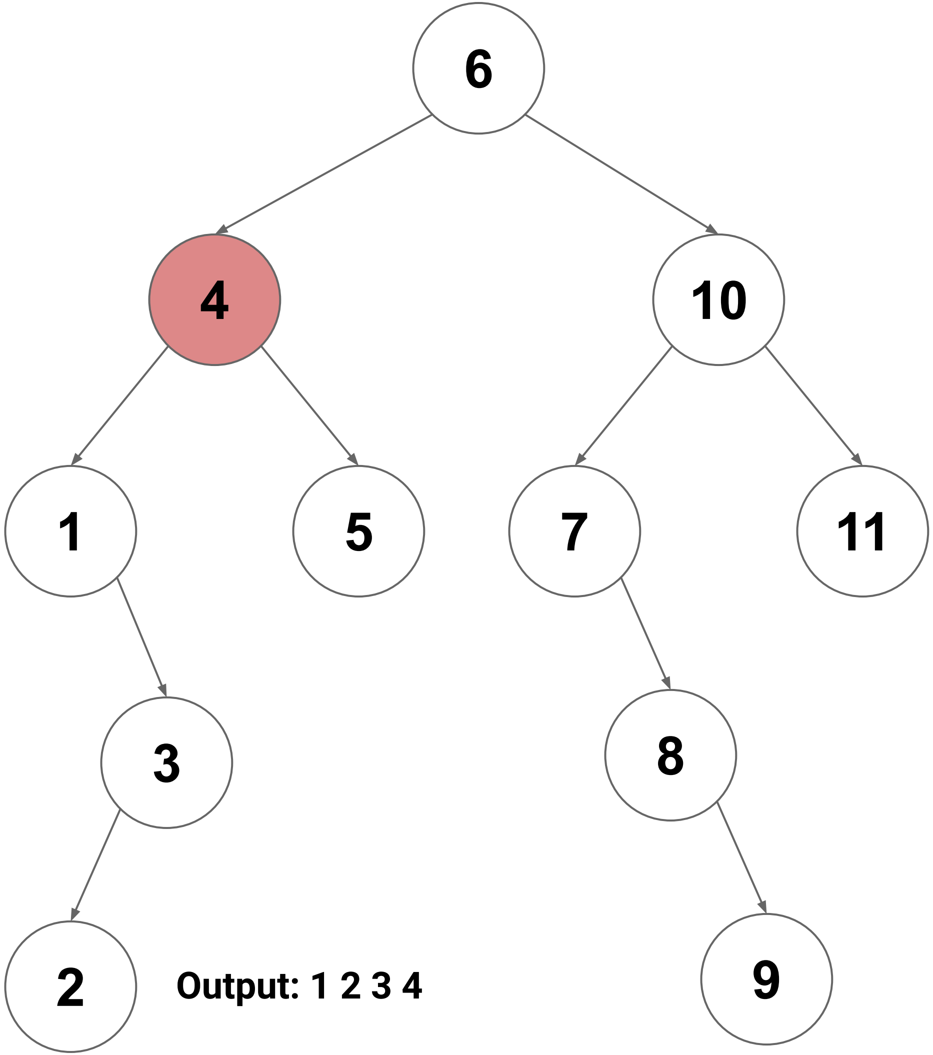
```
inorderVisit(6)

inorderVisit(4)

visit(4)
```



In-Order Traversal on a BST

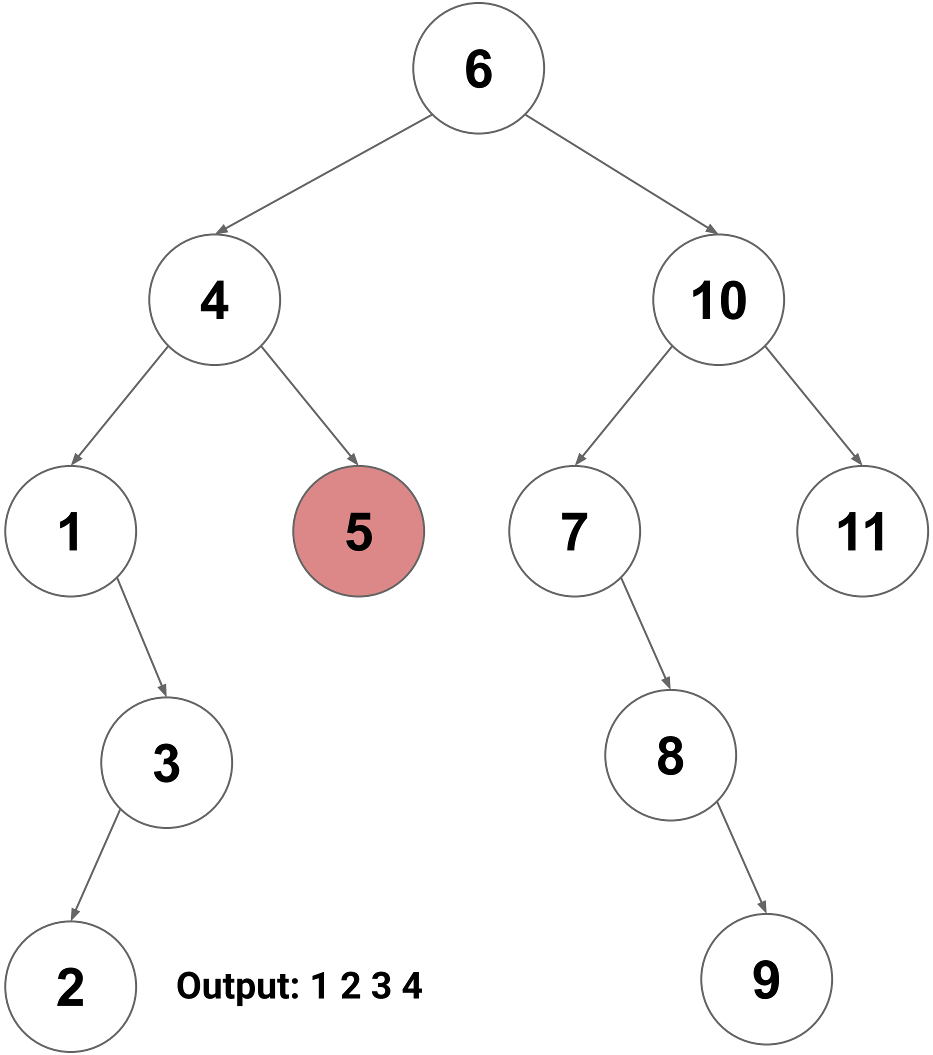
`inorderVisit(6)`

`inorderVisit(4)`



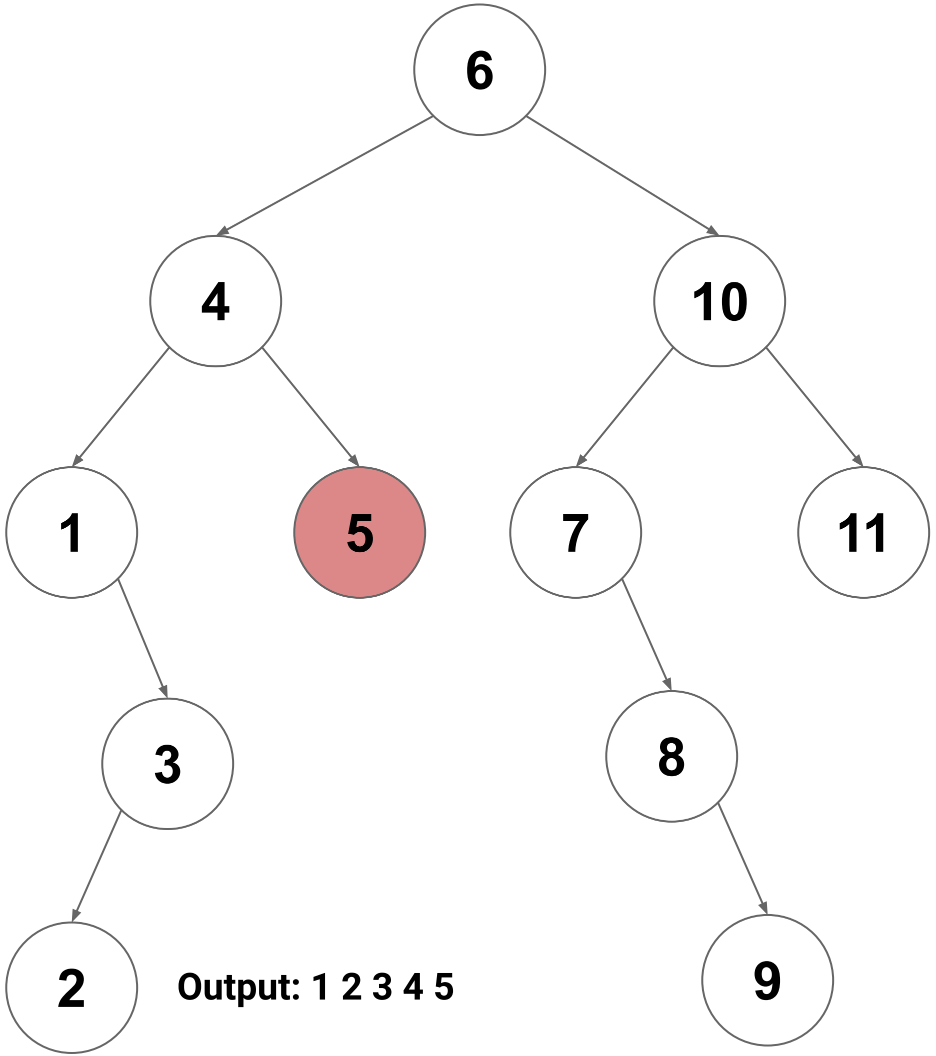
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(4)

inorderVisit(5)
```



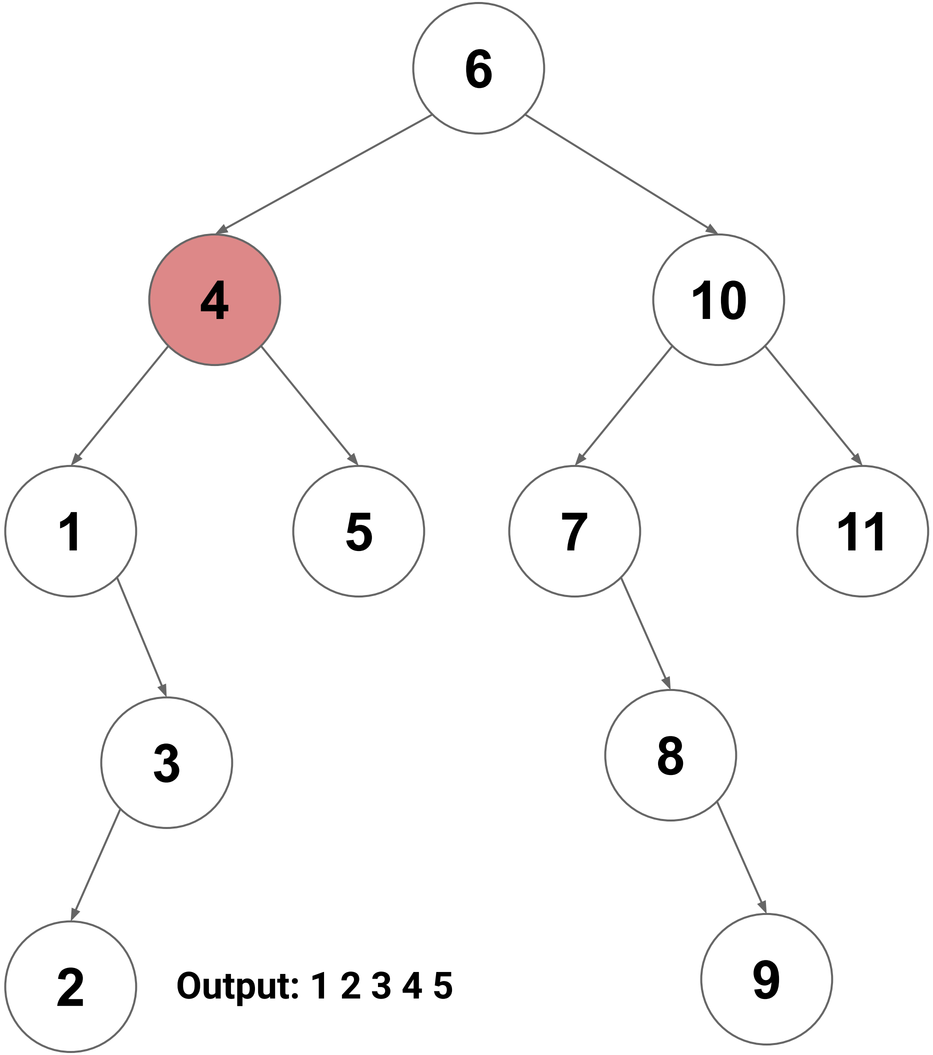
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(4)

visit(5)
```



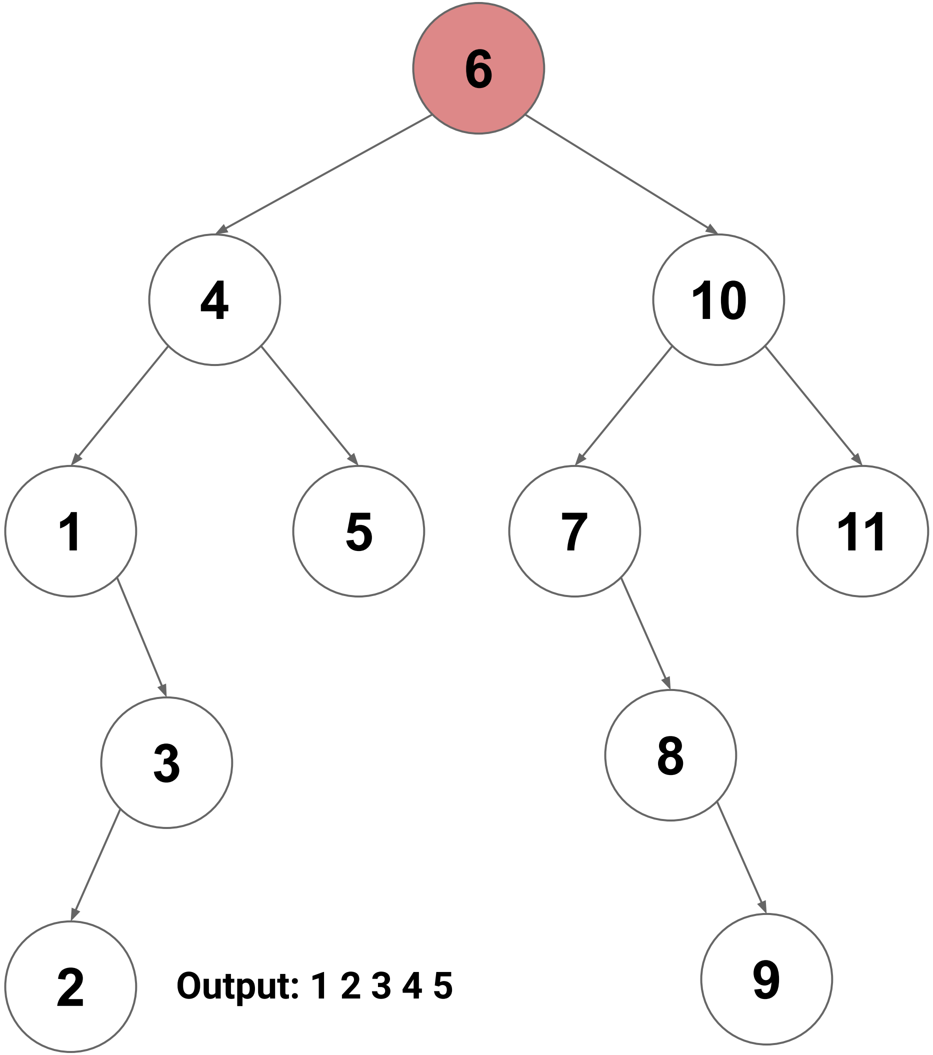
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(4)
```



In-Order Traversal on a BST

`inorderVisit(6)`

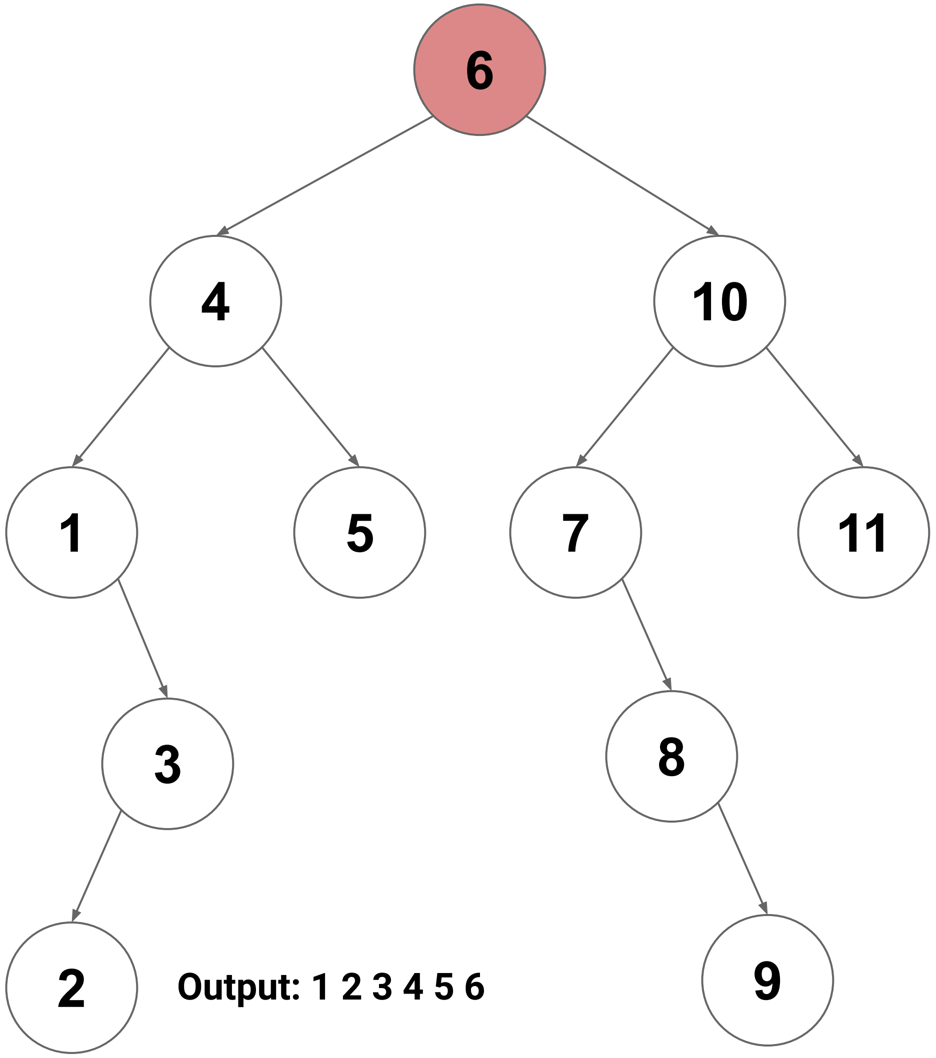


Output: 1 2 3 4 5

In-Order Traversal on a BST

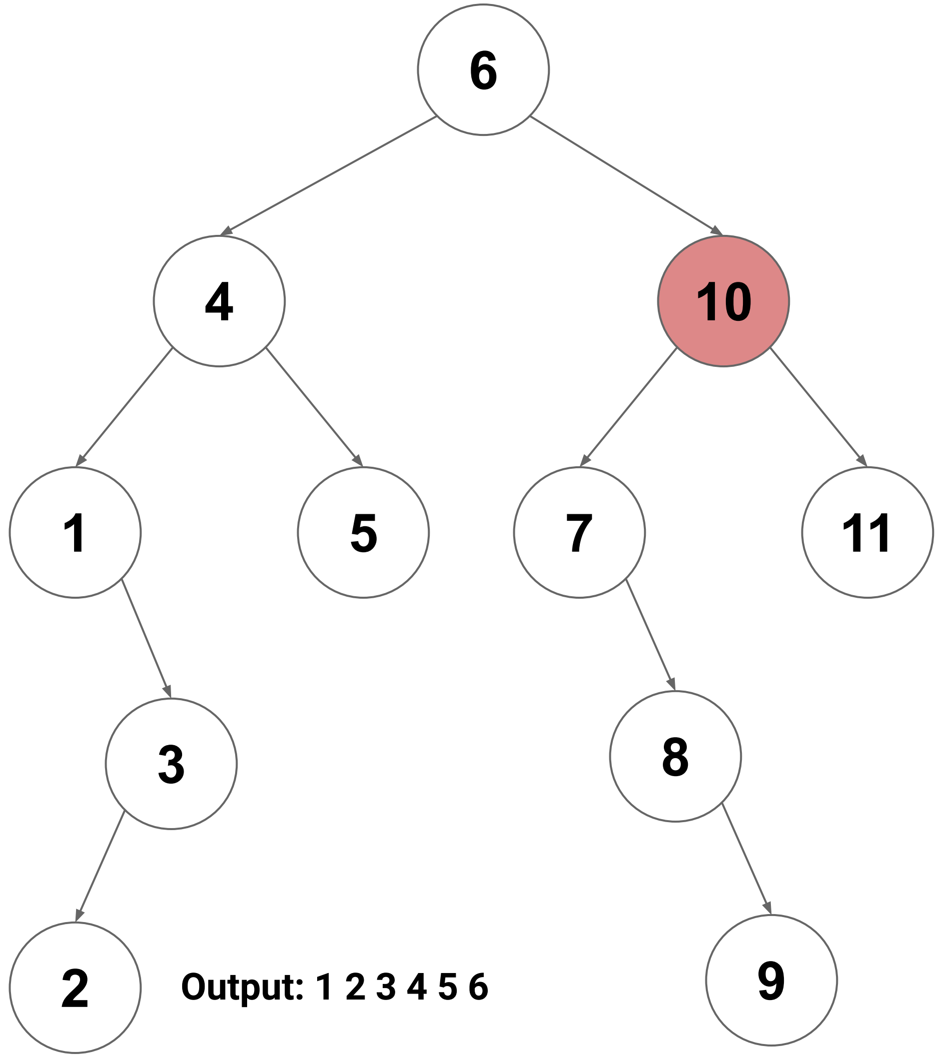
`inorderVisit(6)`

`visit(6)`



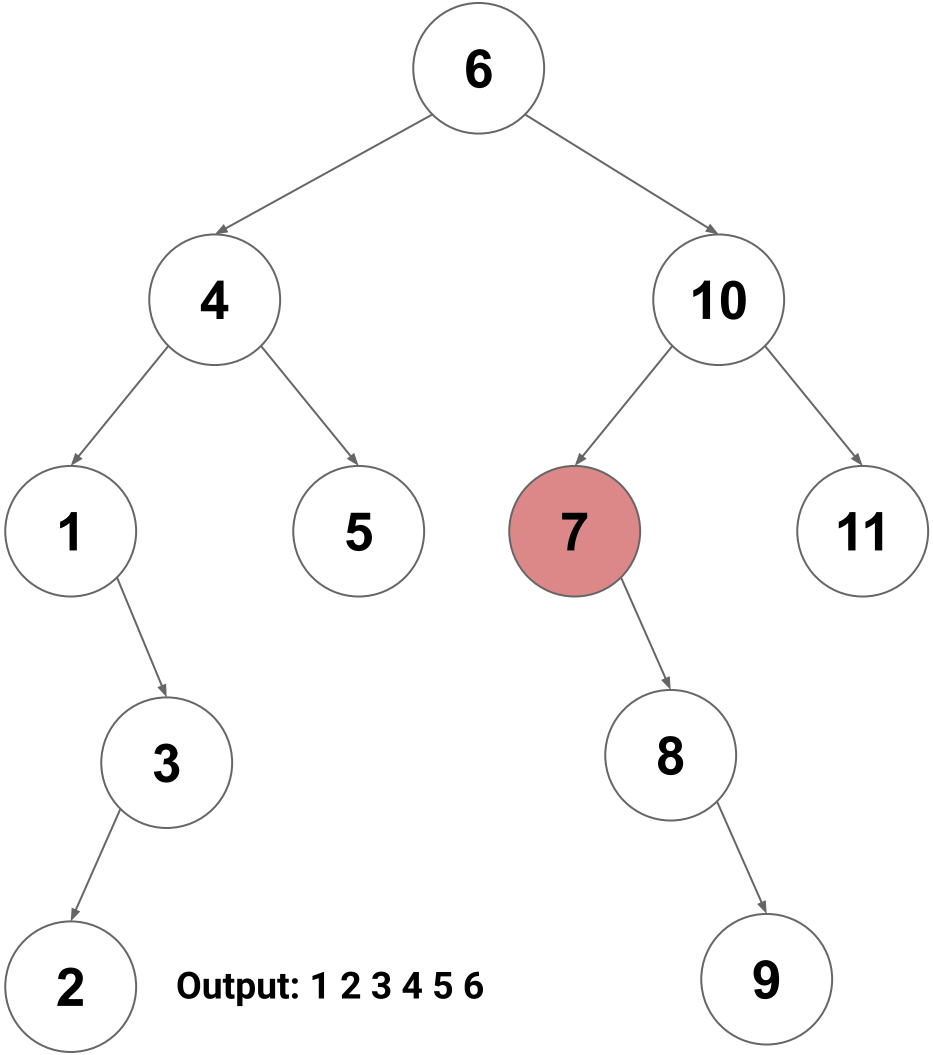
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(10)
```



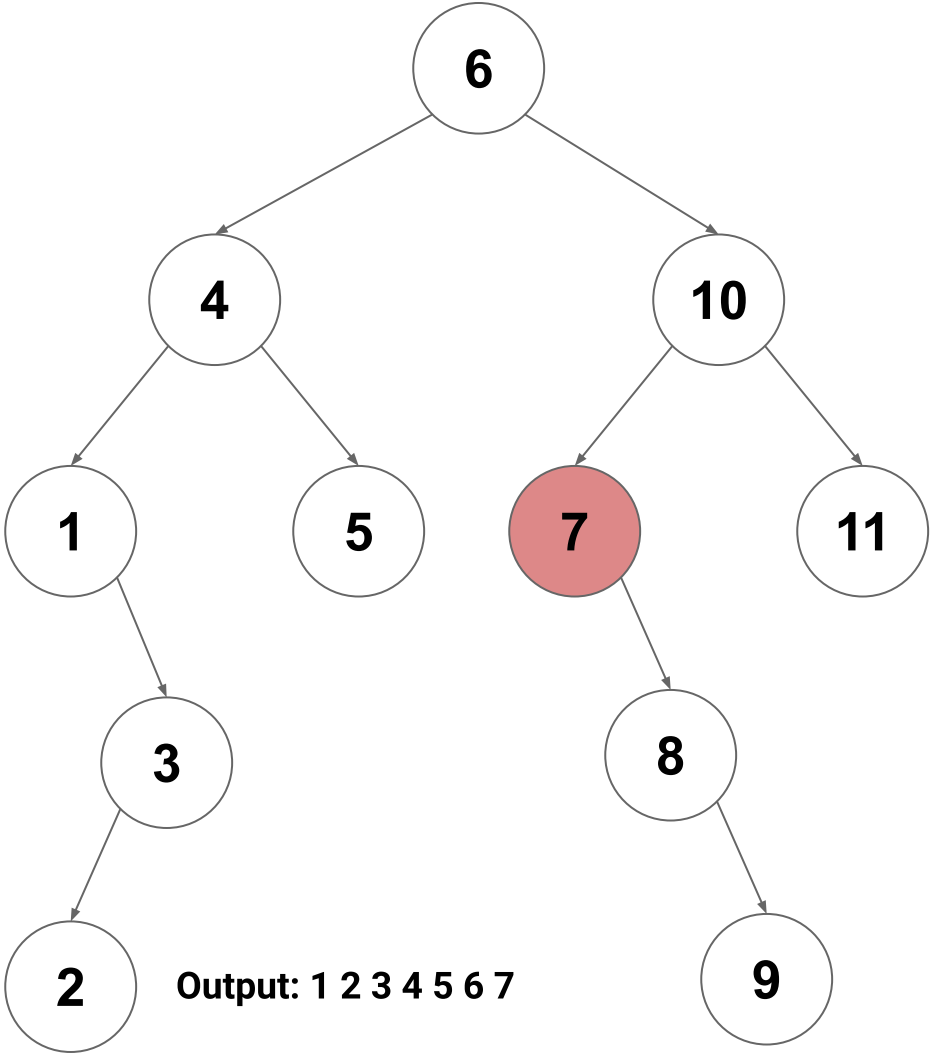
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(10)

inorderVisit(7)
```



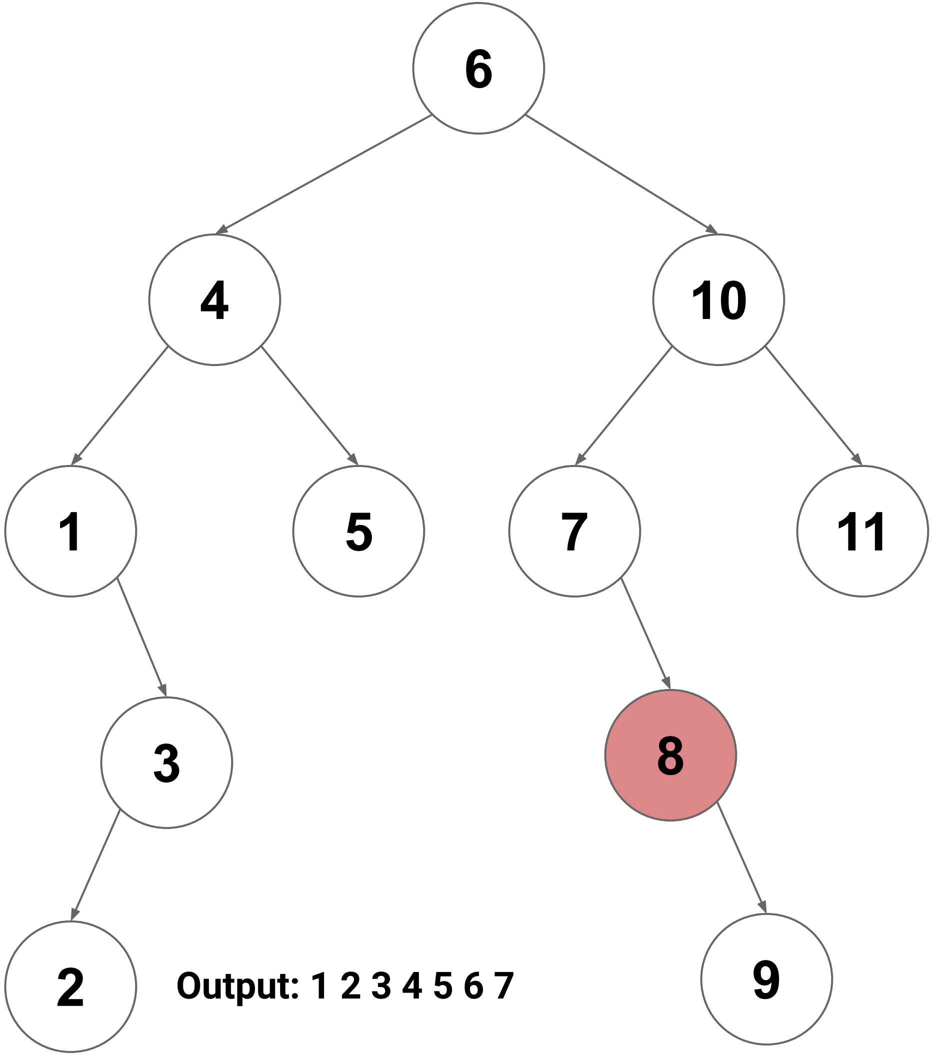
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(10)

inorderVisit(7)

visit(7)
```



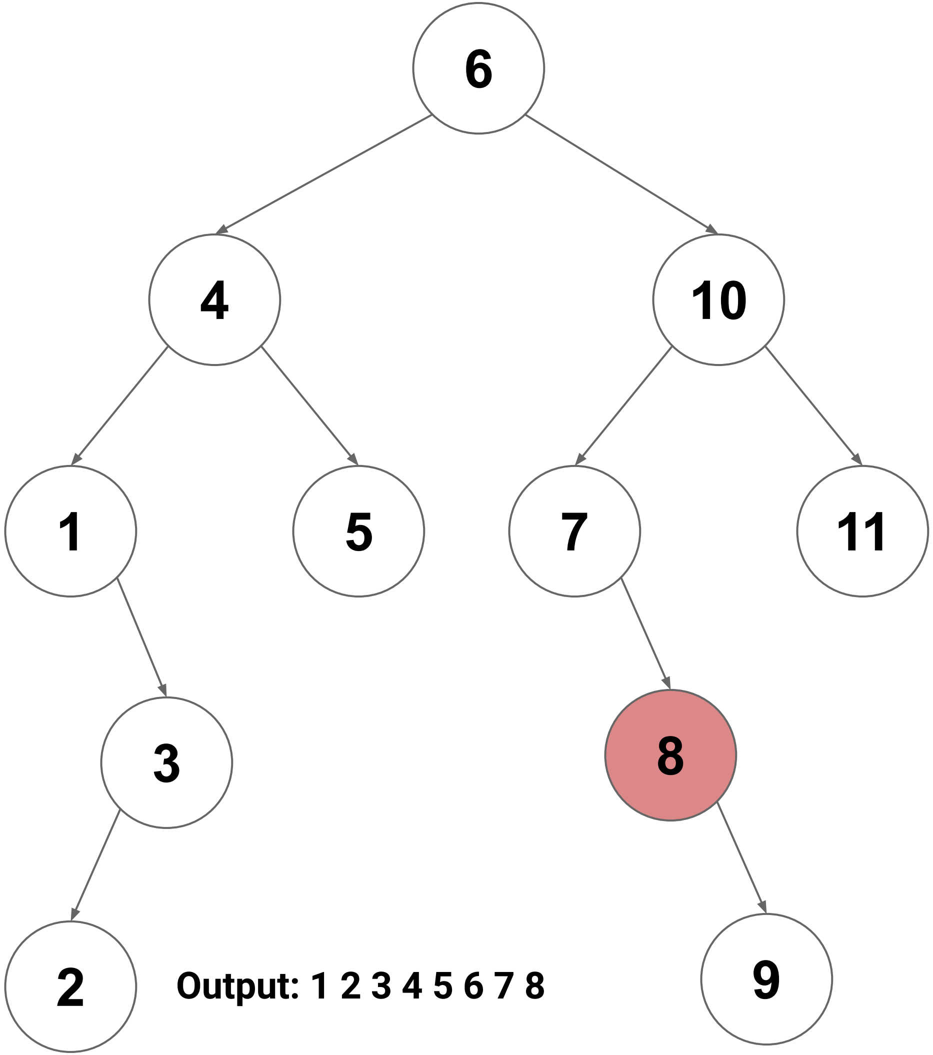
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(10)

inorderVisit(7)

inorderVisit(8)
```



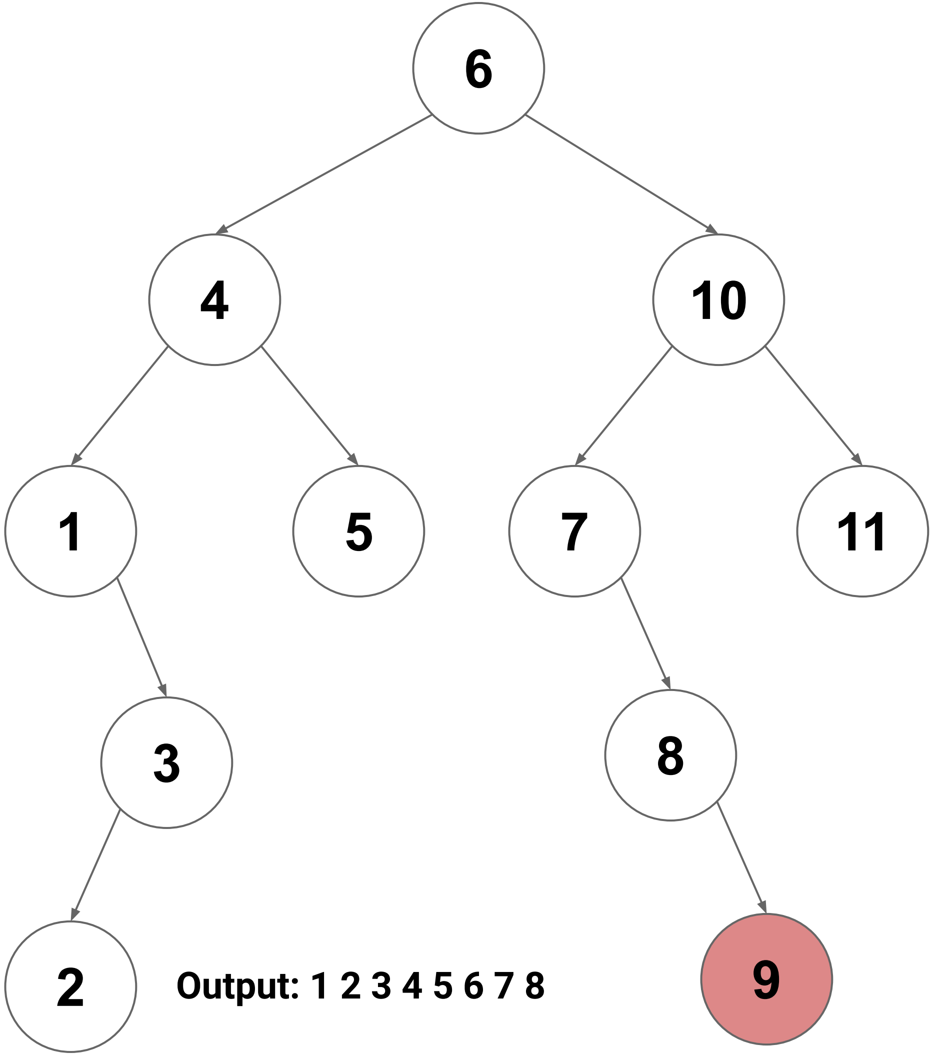
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(10)

inorderVisit(7)

inorderVisit(8)

visit(8)
```



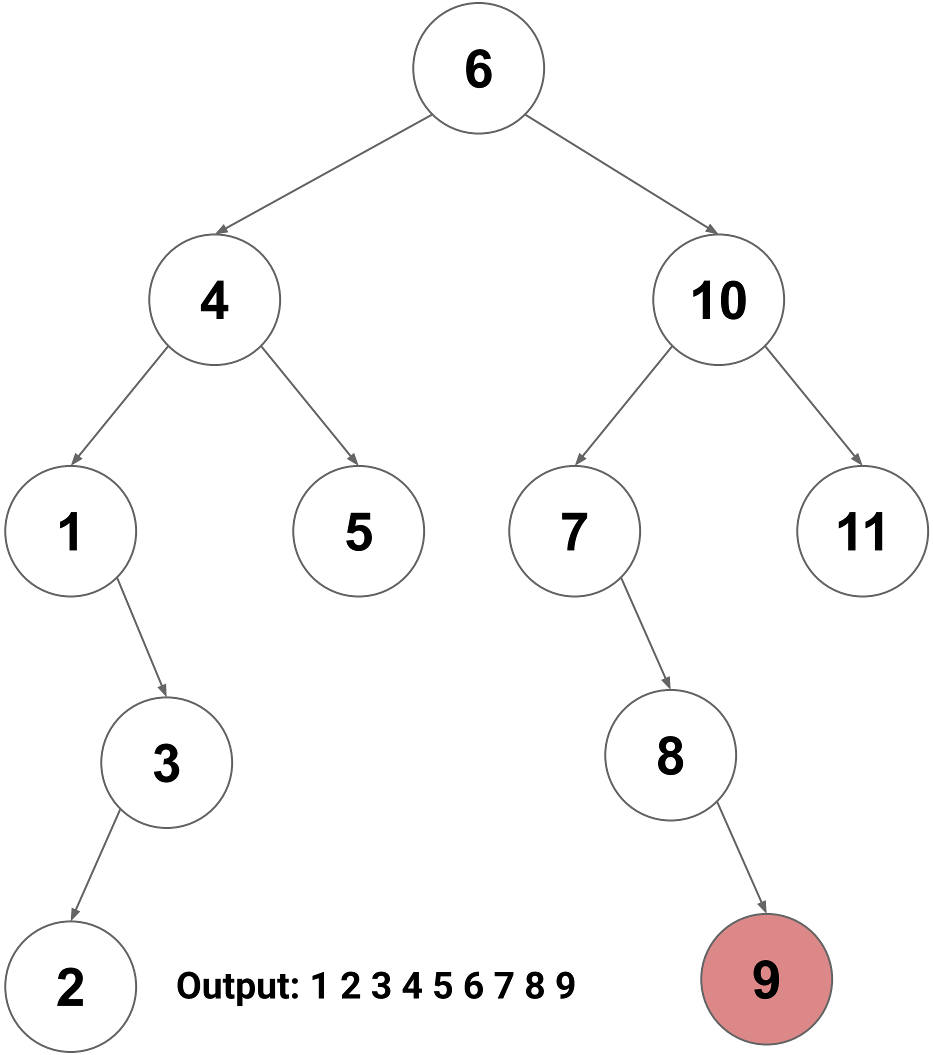
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(10)

inorderVisit(7)

inorderVisit(8)

inorderVisit(9)
```



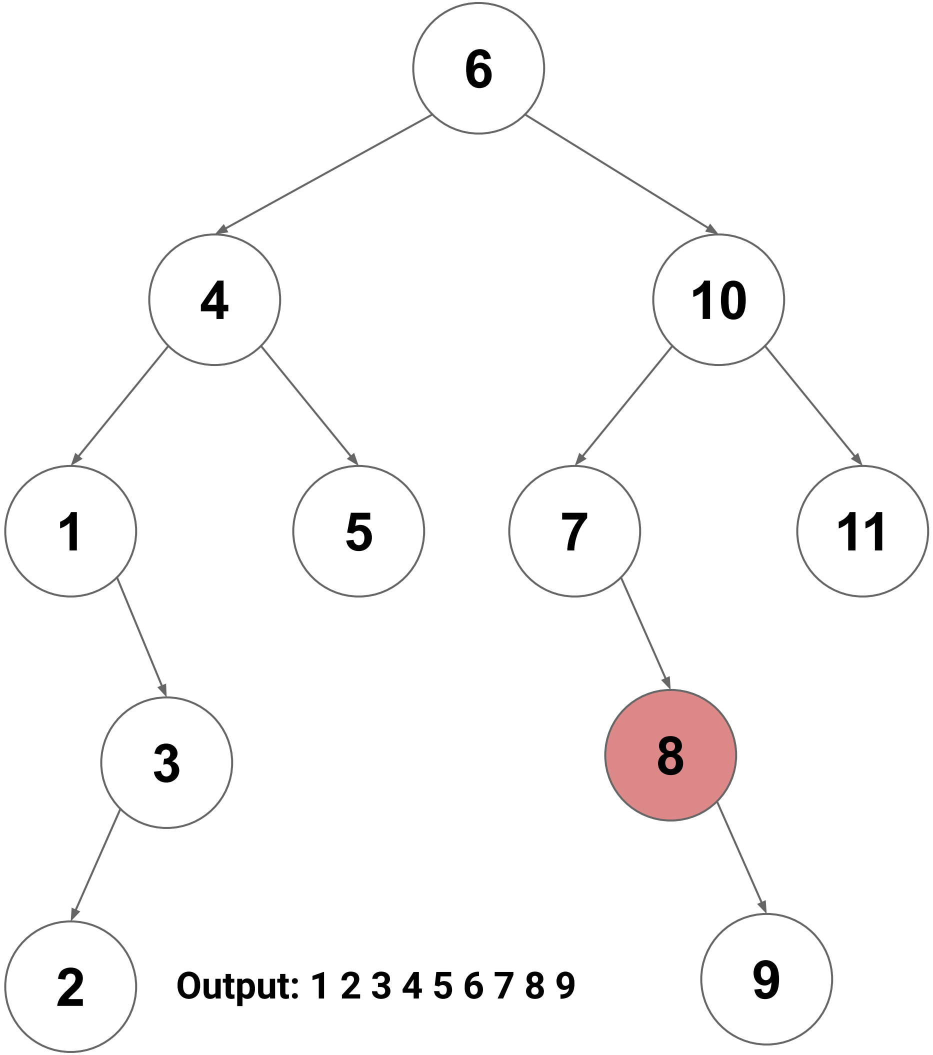
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(10)

inorderVisit(7)

inorderVisit(8)

visit(9)
```



In-Order Traversal on a BST

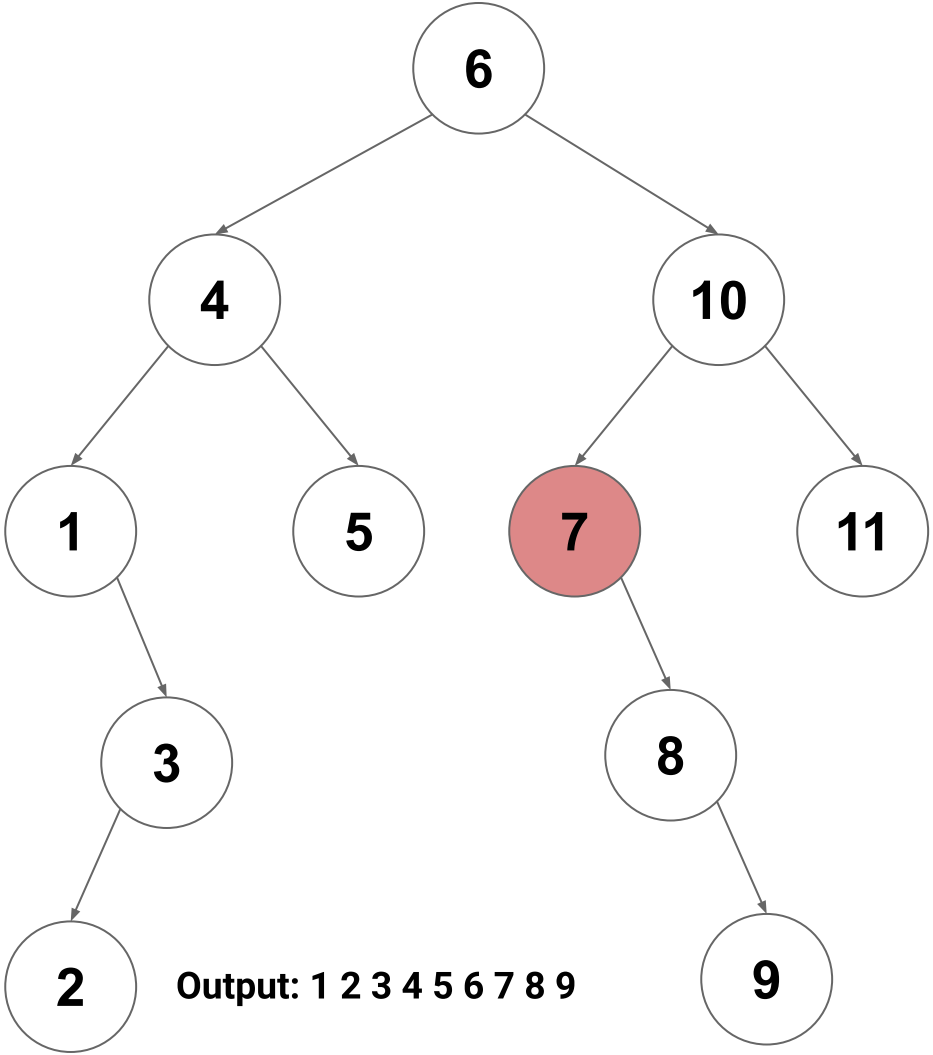
```
inorderVisit(6)

inorderVisit(10)

inorderVisit(7)

inorderVisit(8)
```



Output: 1 2 3 4 5 6 7 8 9

In-Order Traversal on a BST

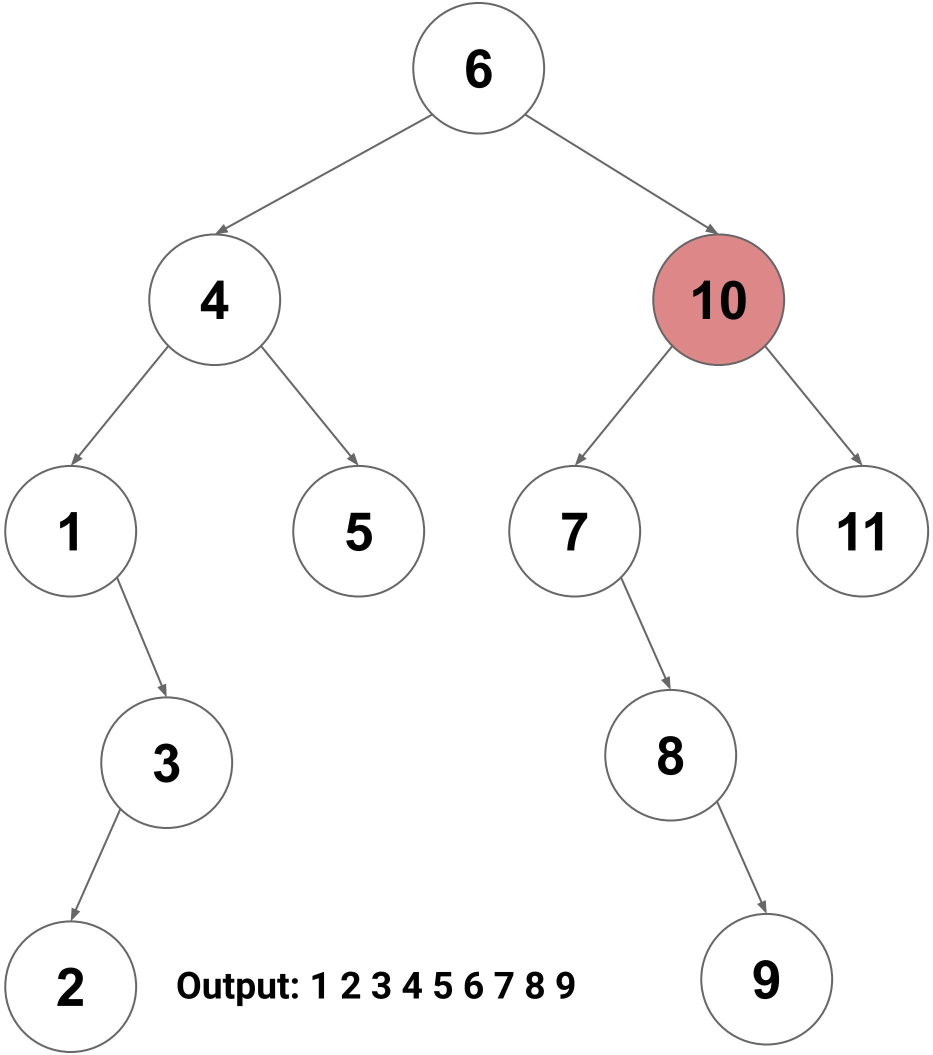
```
inorderVisit(6)

inorderVisit(10)

inorderVisit(7)
```



In-Order Traversal on a BST

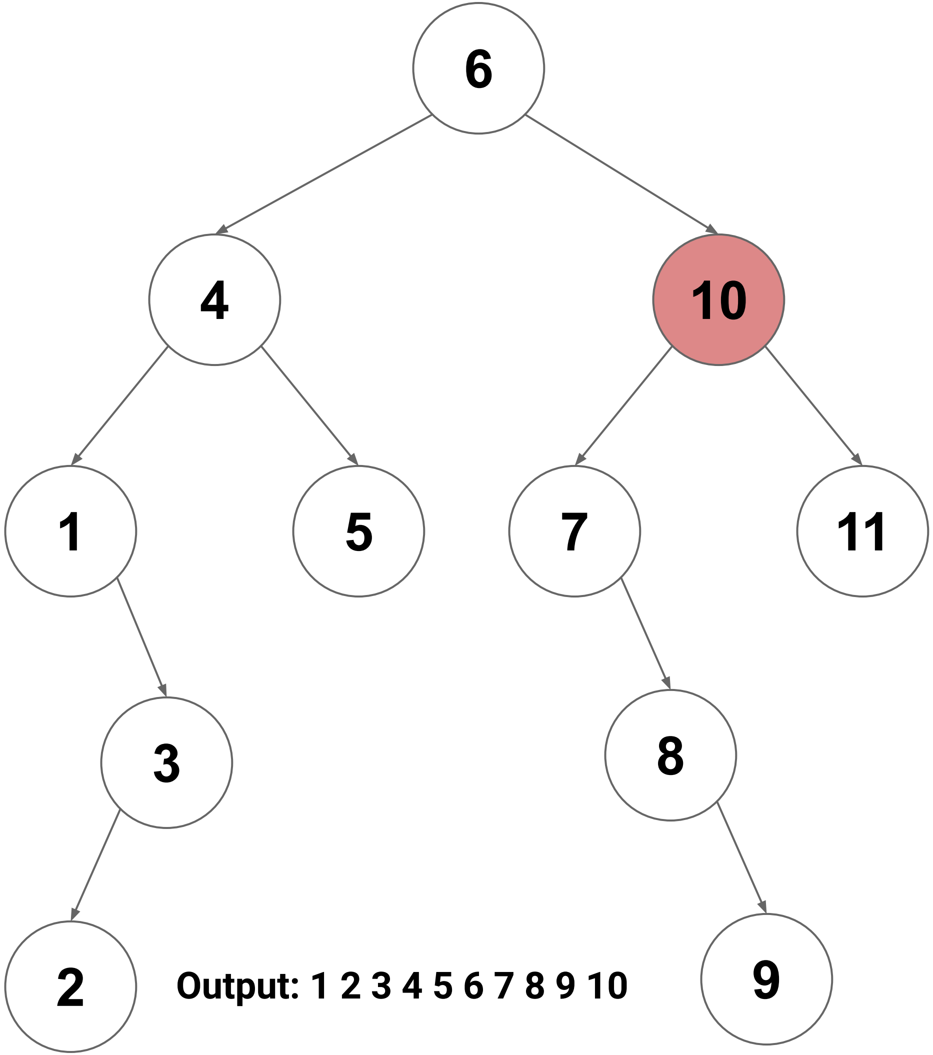
`inorderVisit(6)`

`inorderVisit(10)`



In-Order Traversal on a BST

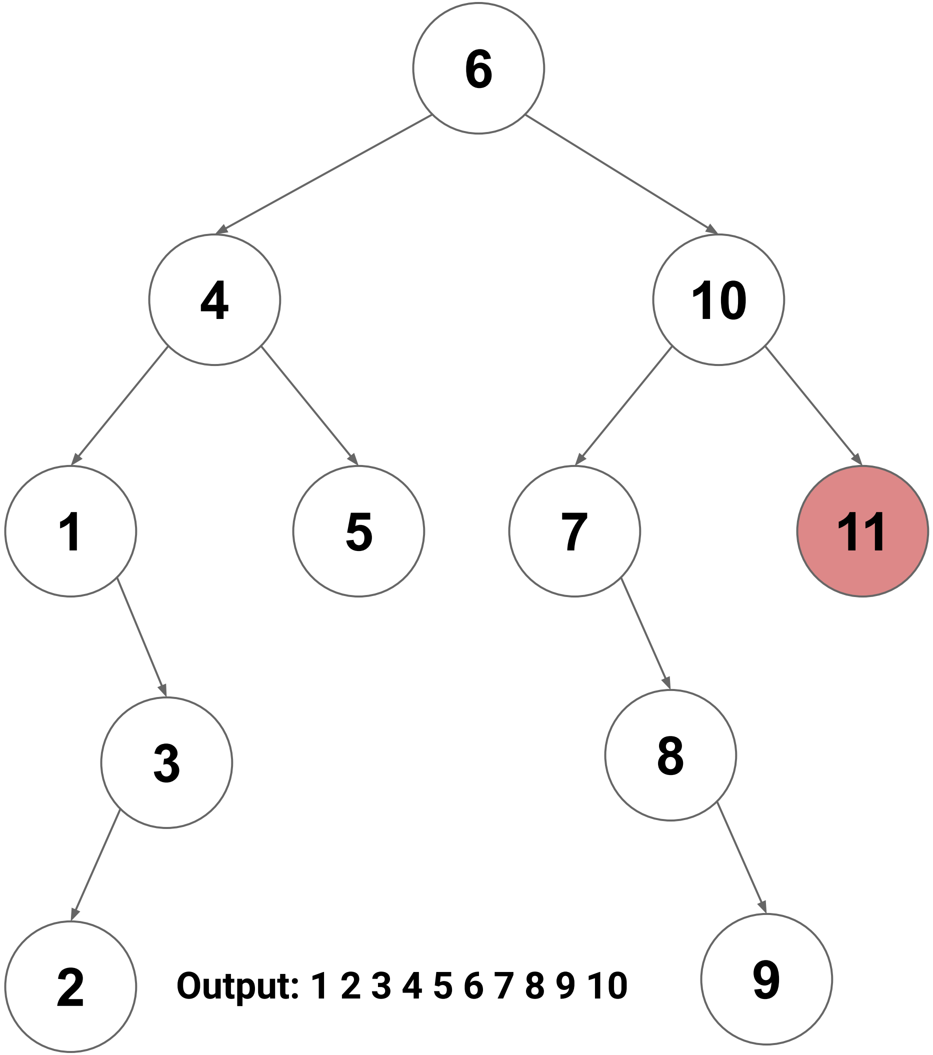
```
inorderVisit(6)

inorderVisit(10)

visit(10)
```



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

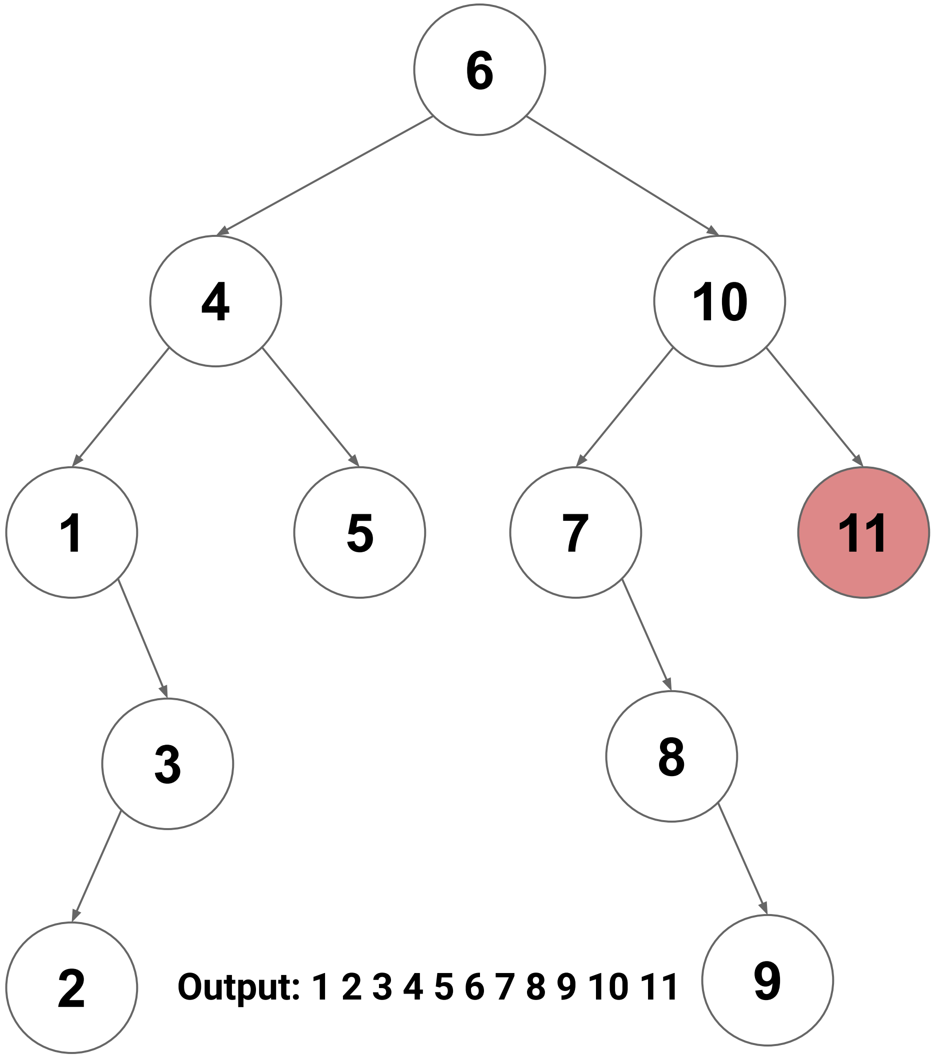
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(10)

inorderVisit(11)
```



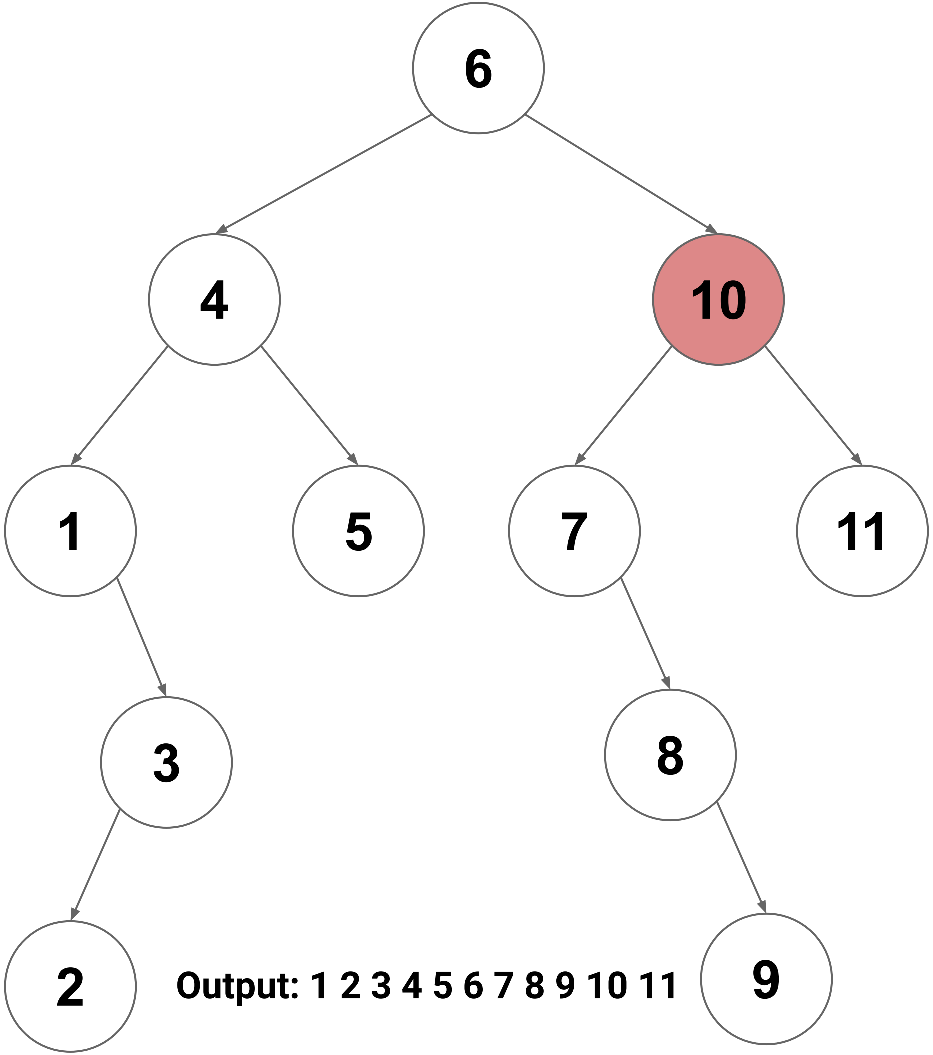
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(10)

inorderVisit(11)

visit(11)
```



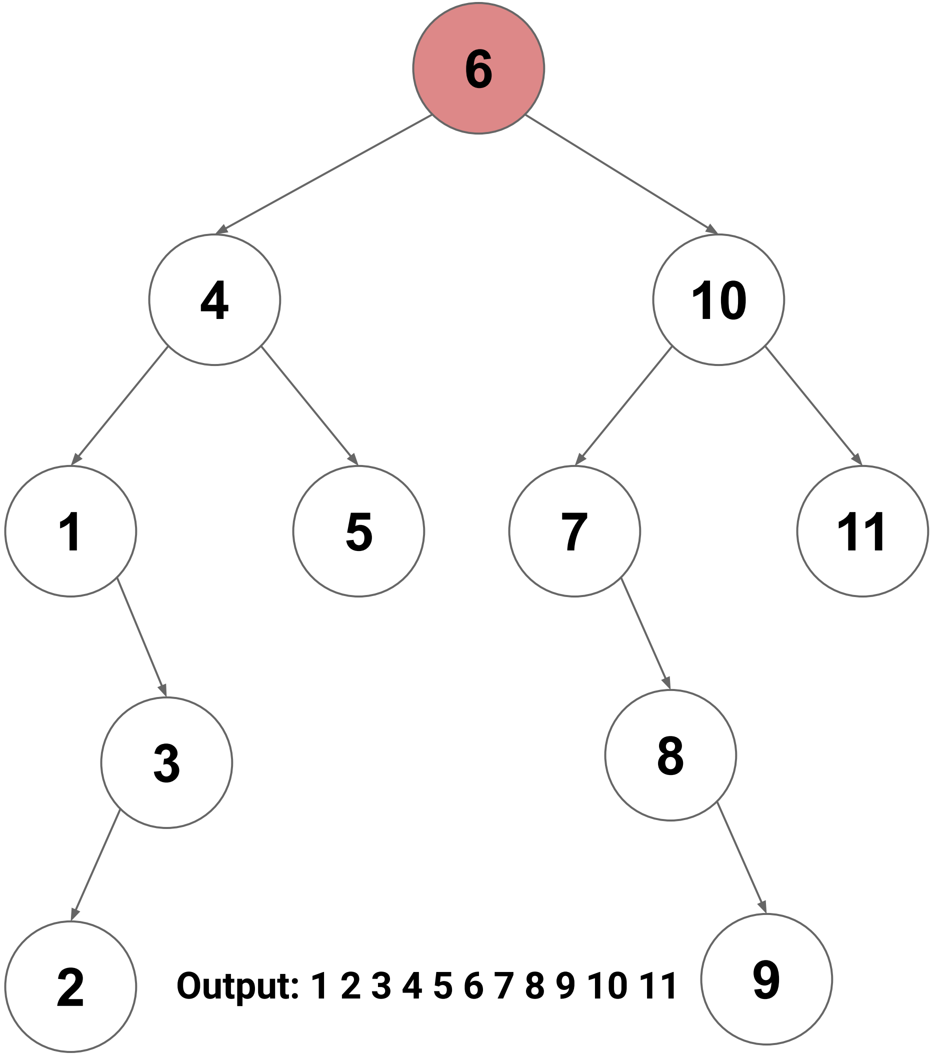
In-Order Traversal on a BST

```
inorderVisit(6)

inorderVisit(10)
```

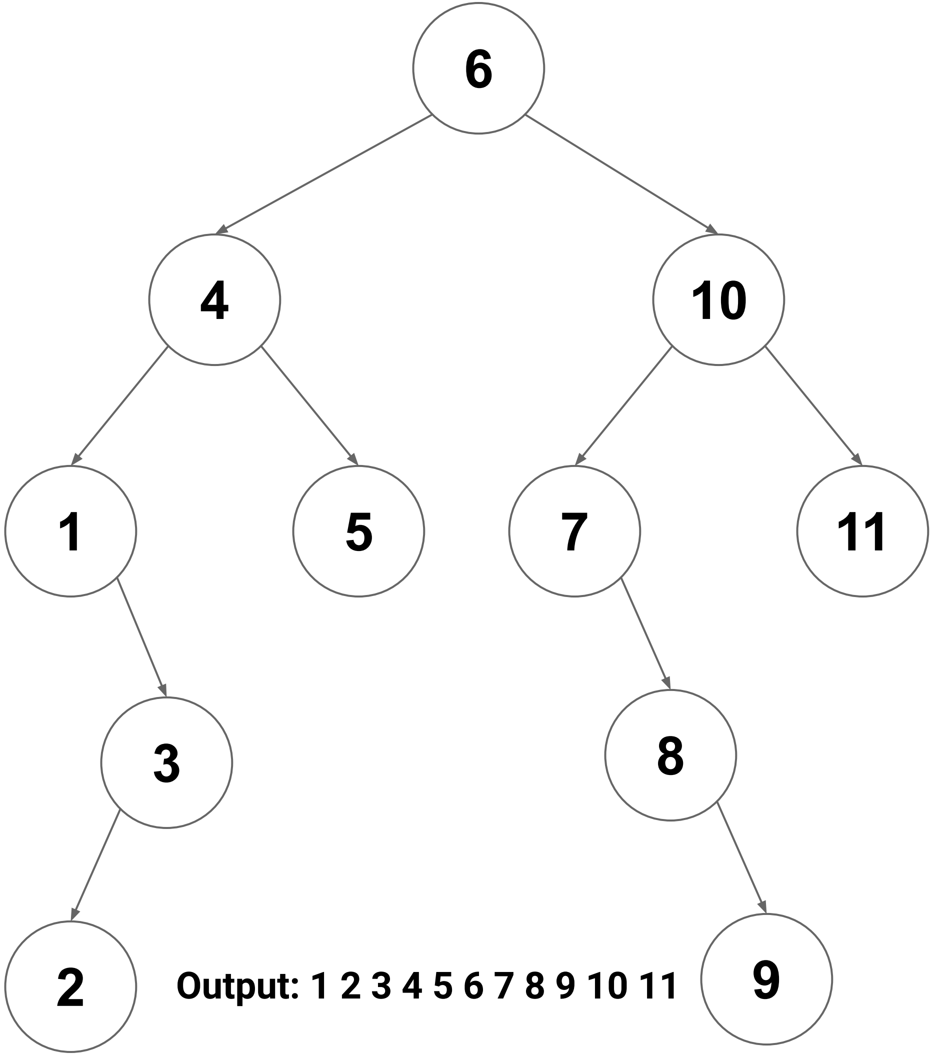


In-Order Traversal on a BST

`inorderVisit(6)`



In-Order Traversal on a BST



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

```
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

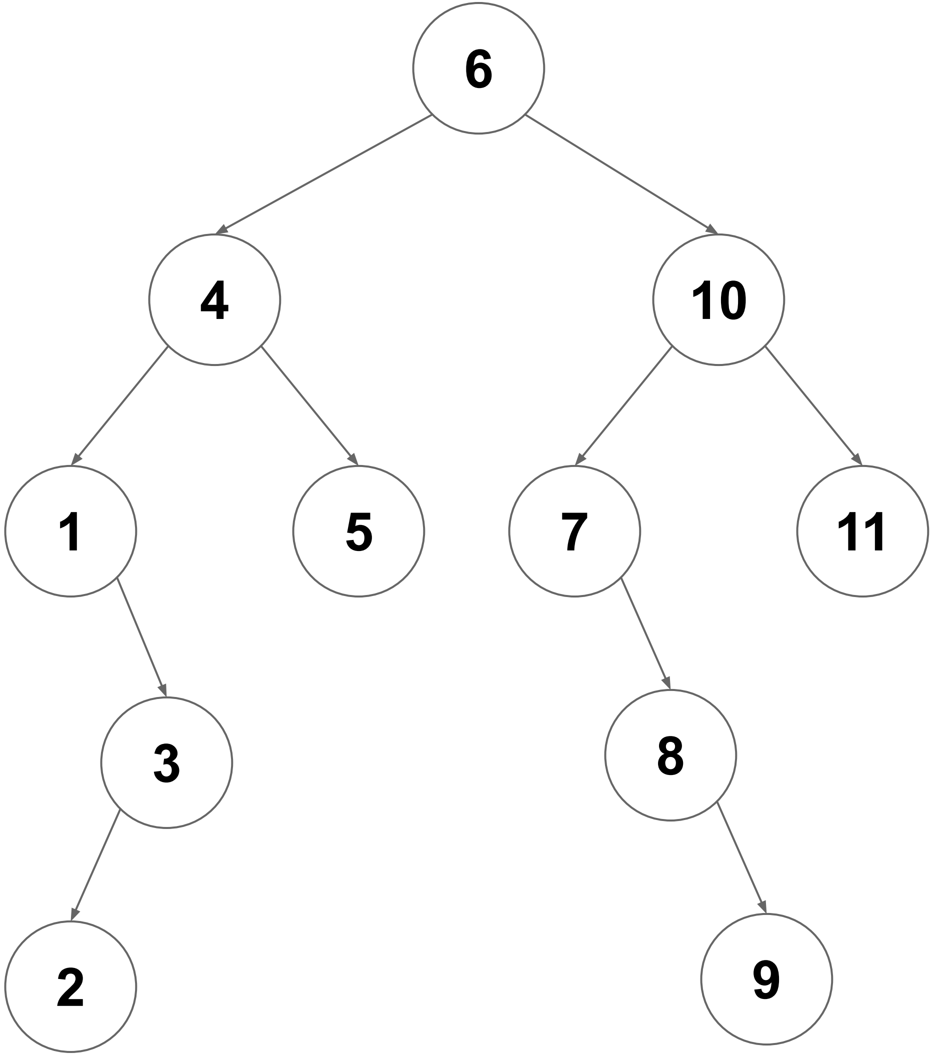
```
class ImmutableTreeIterator[T](root: ImmutableTree[T]) {

  ...

  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

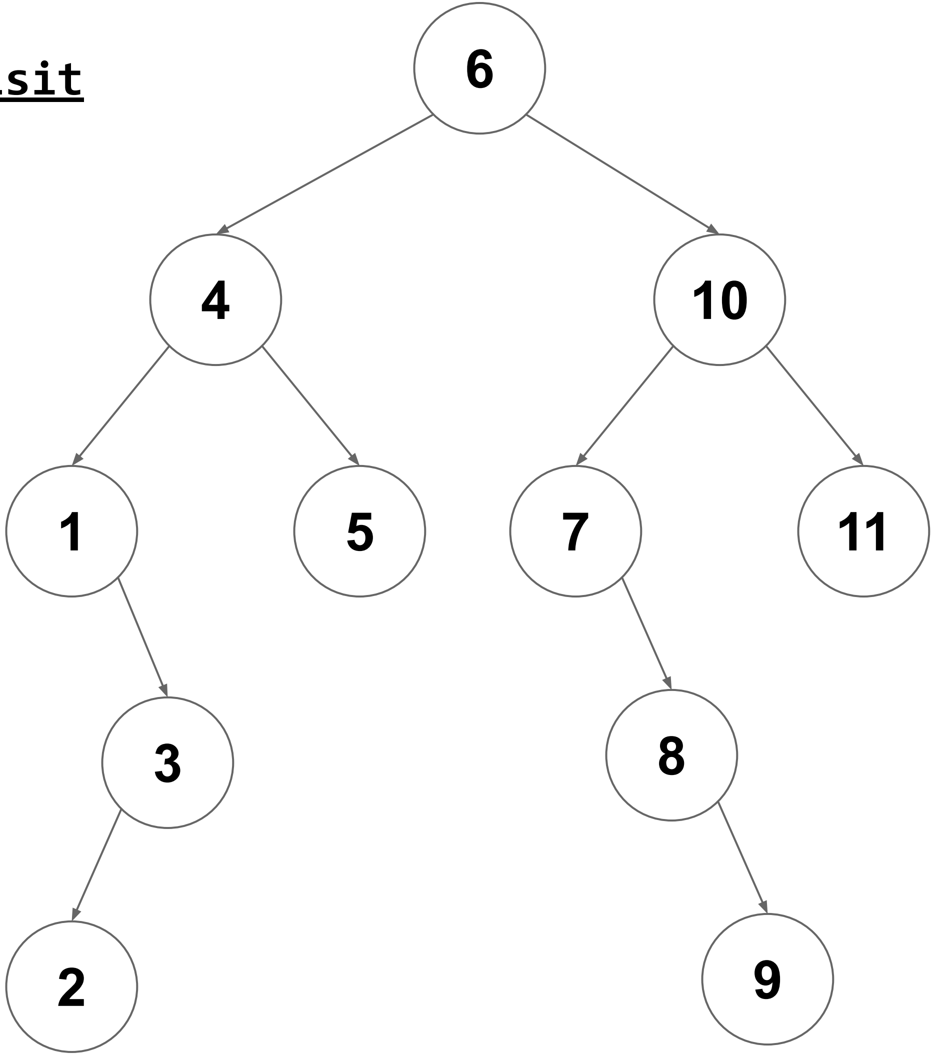


In-Order Traversal with an Iterator

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

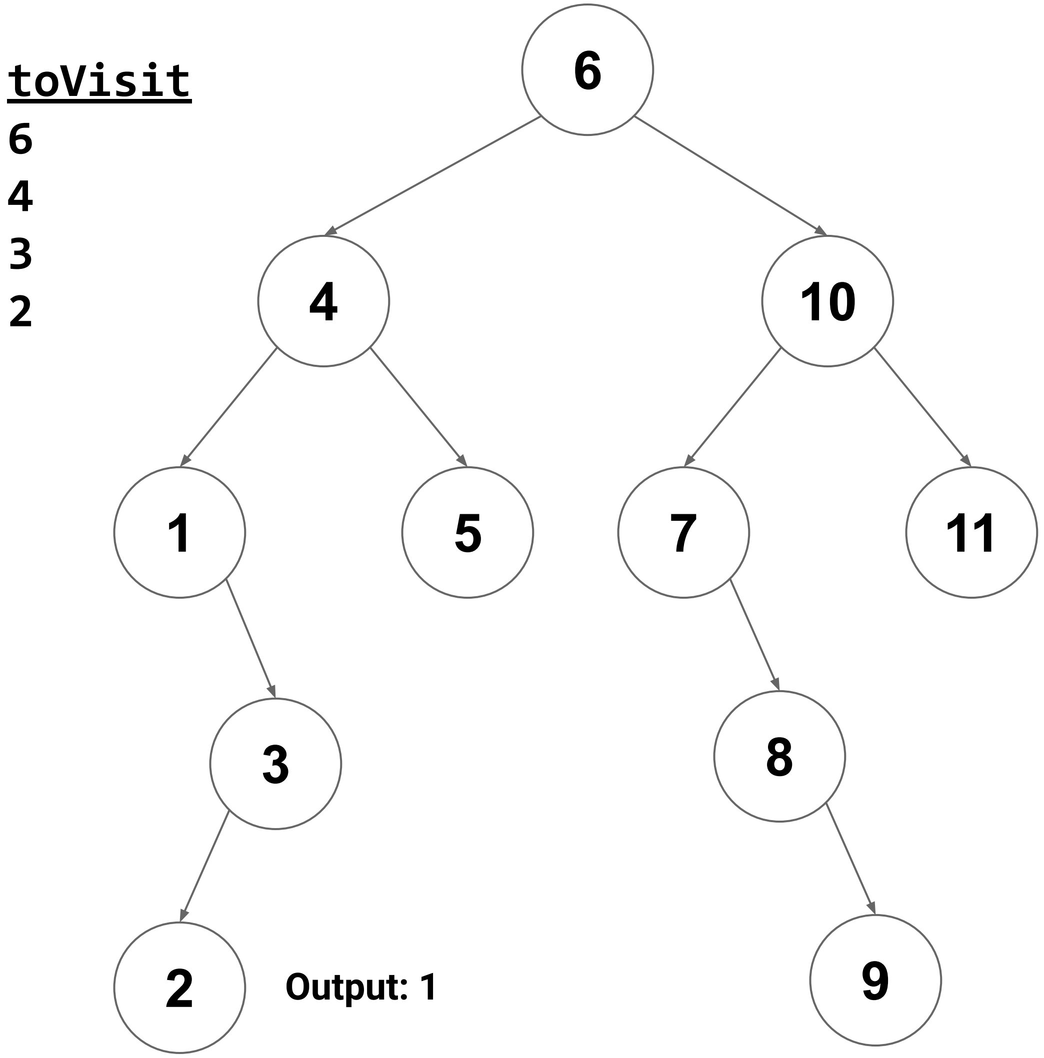
<u>`toVisit`</u>

`6`
`4`
`1`



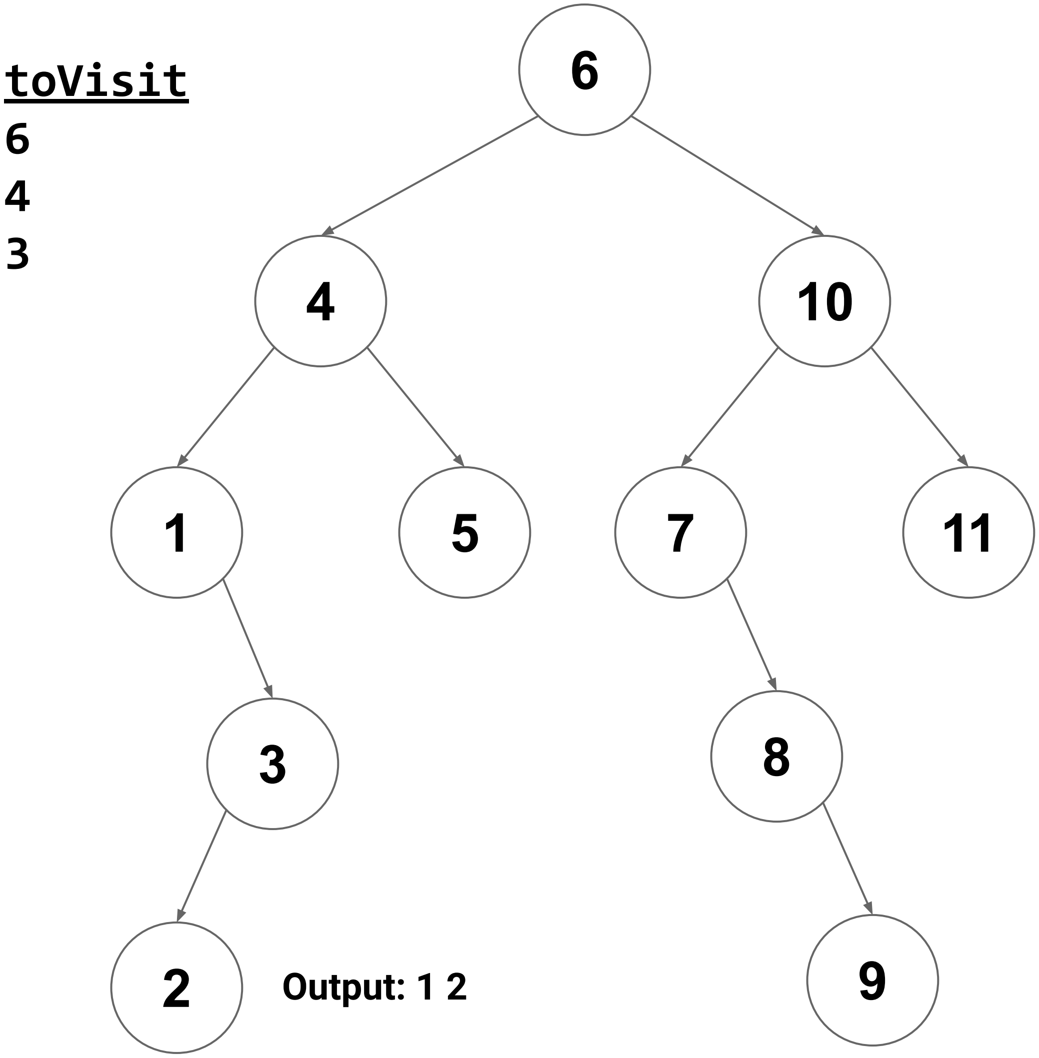
In-Order Traversal with an Iterator

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



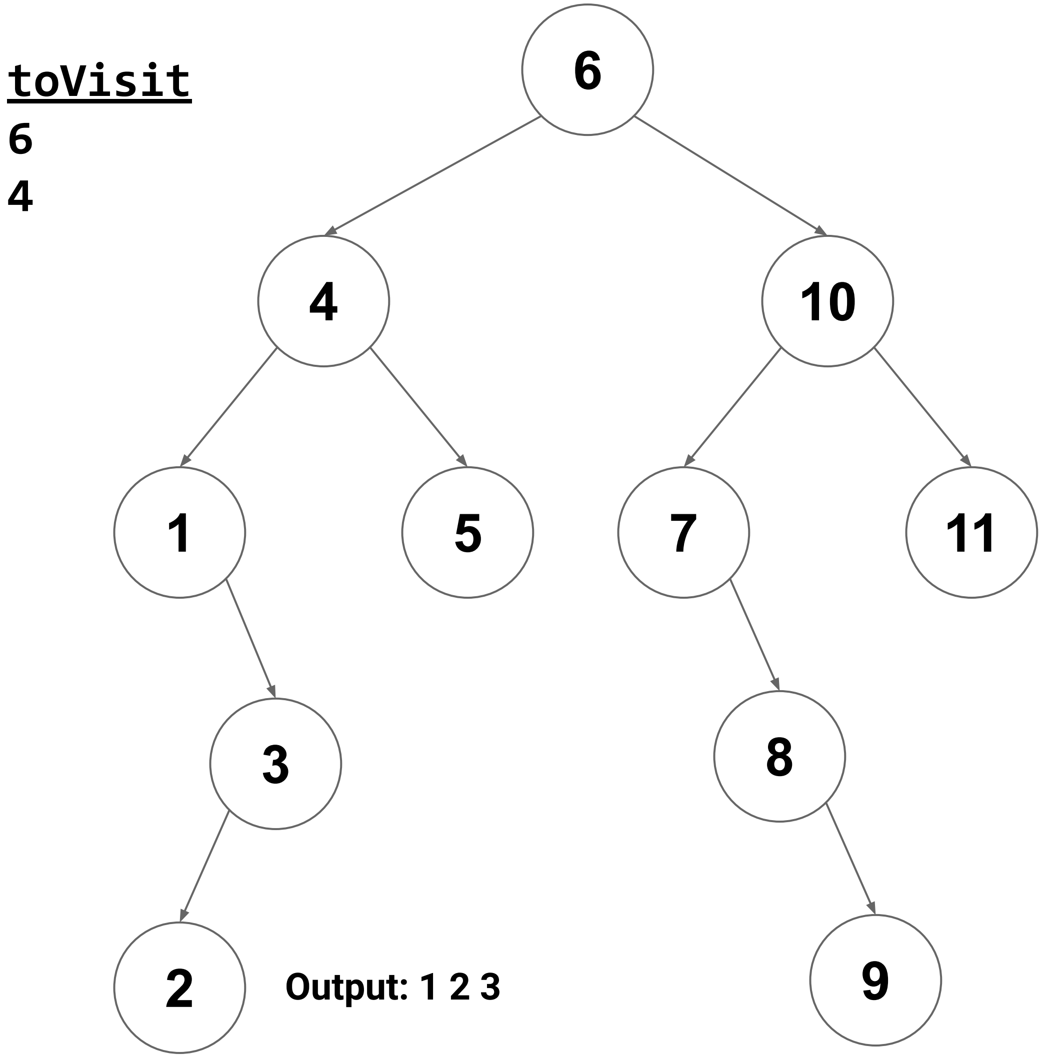
In-Order Traversal with an Iterator

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



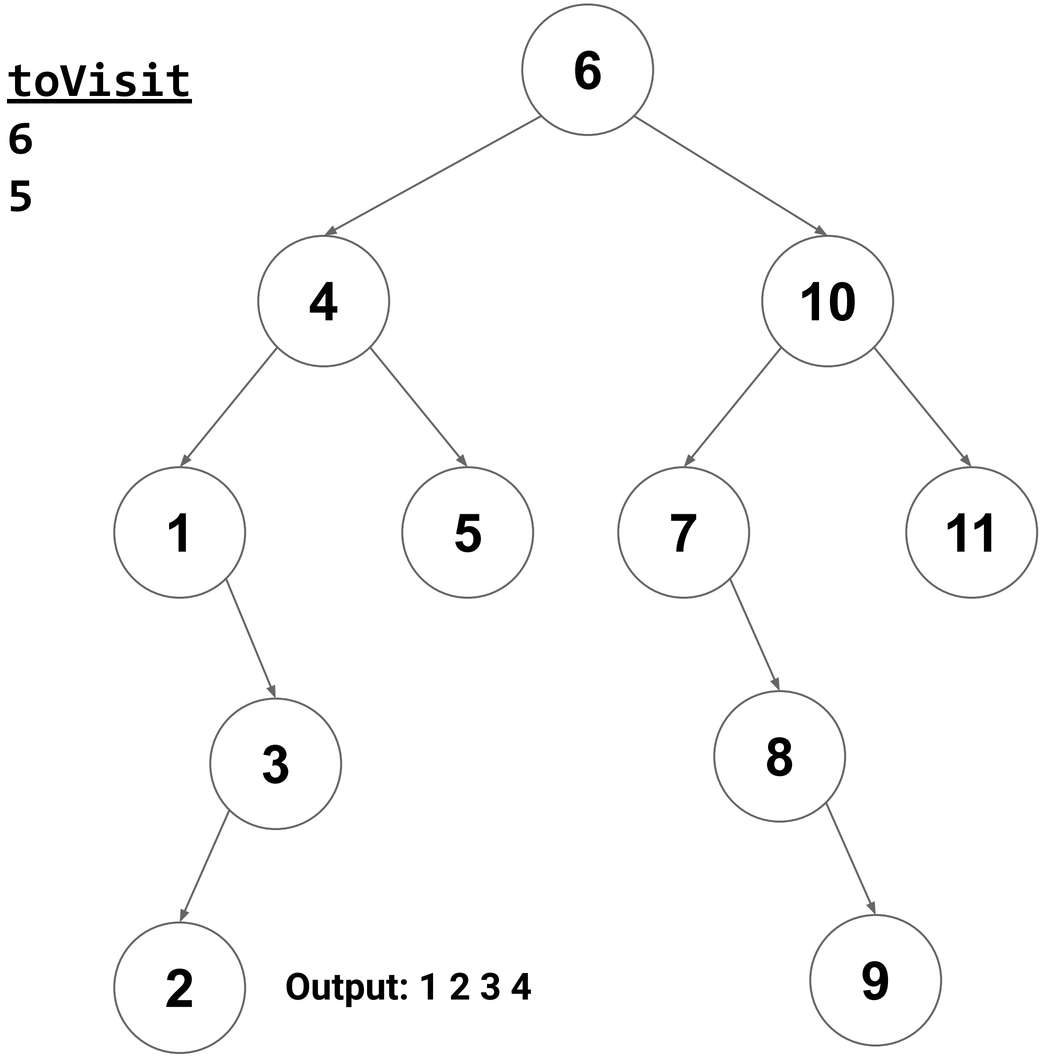
In-Order Traversal with an Iterator

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



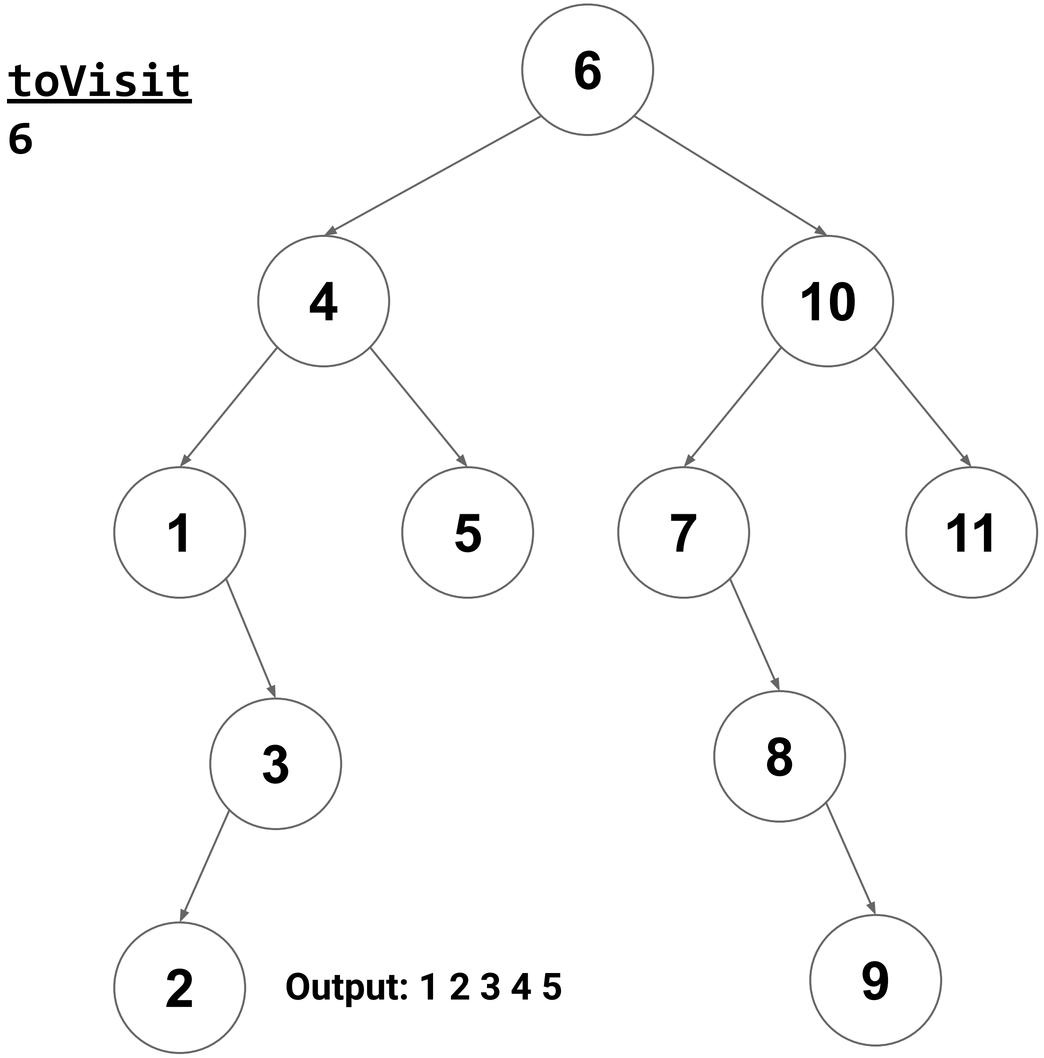
In-Order Traversal with an Iterator

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



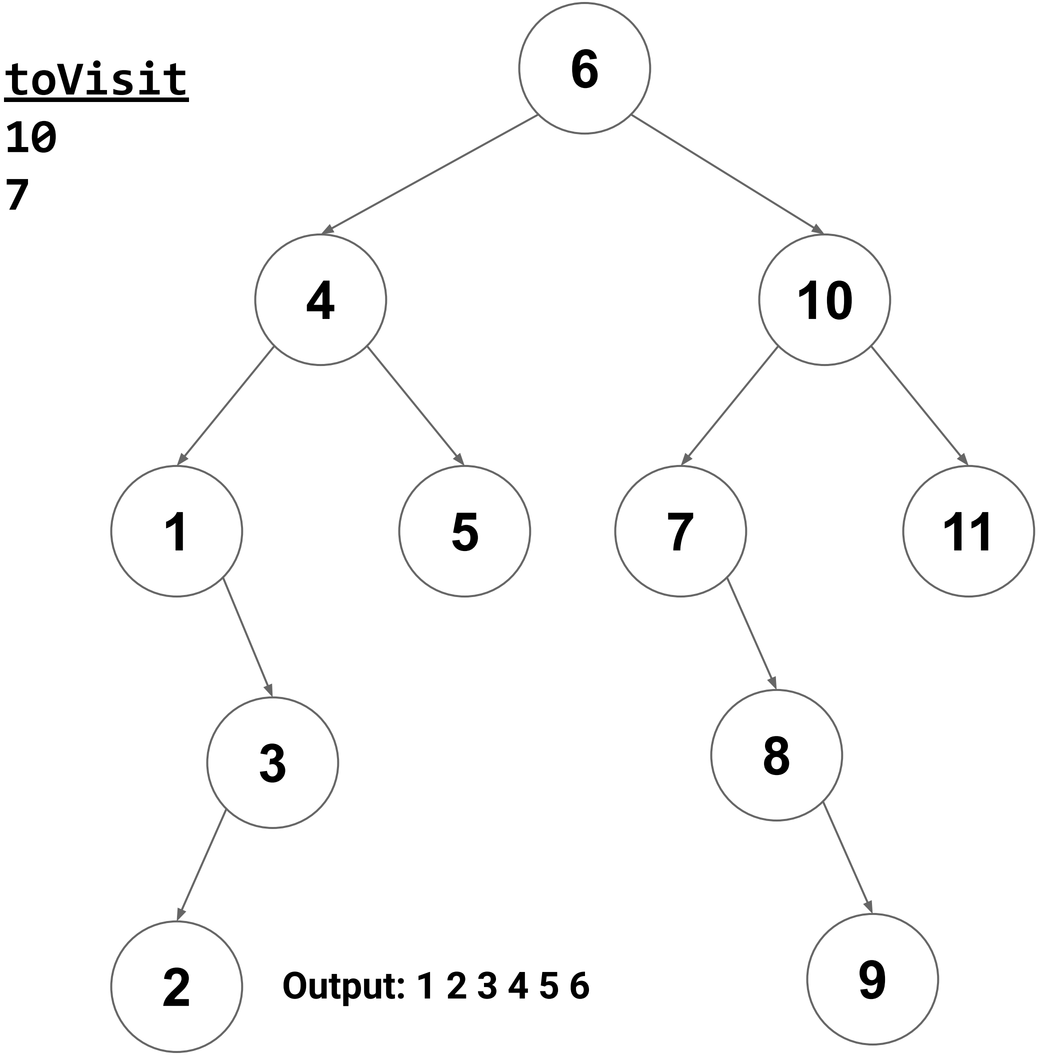
In-Order Traversal with an Iterator

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



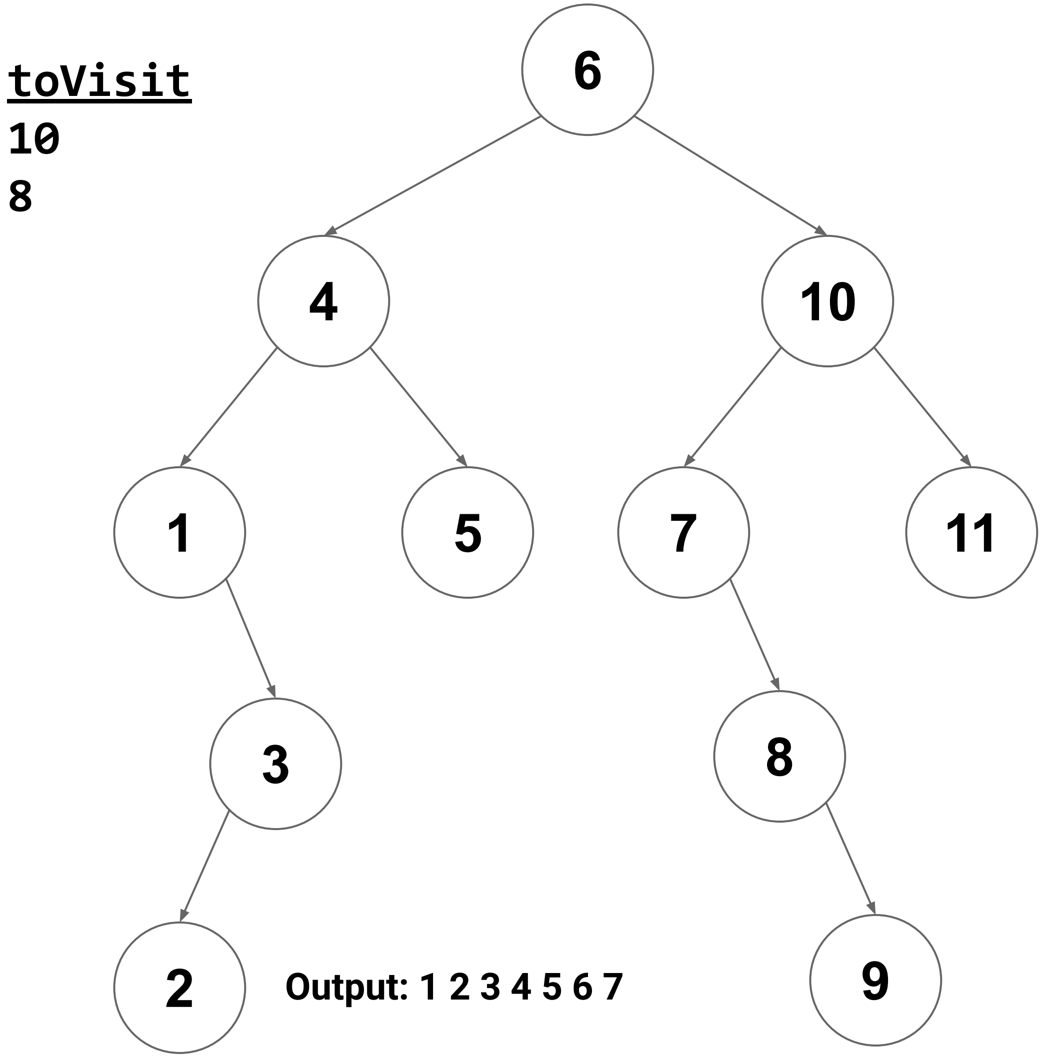
In-Order Traversal with an Iterator

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



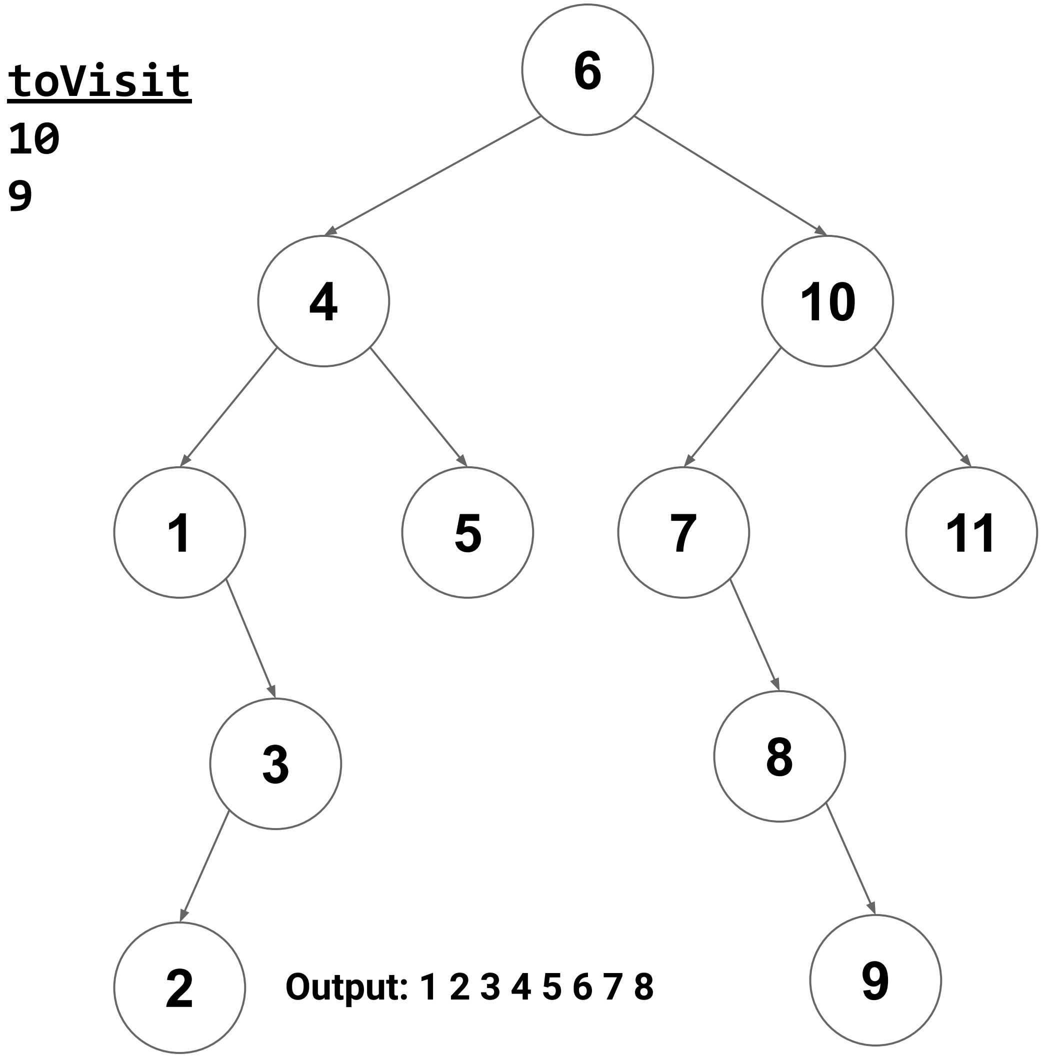
In-Order Traversal with an Iterator

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



In-Order Traversal with an Iterator

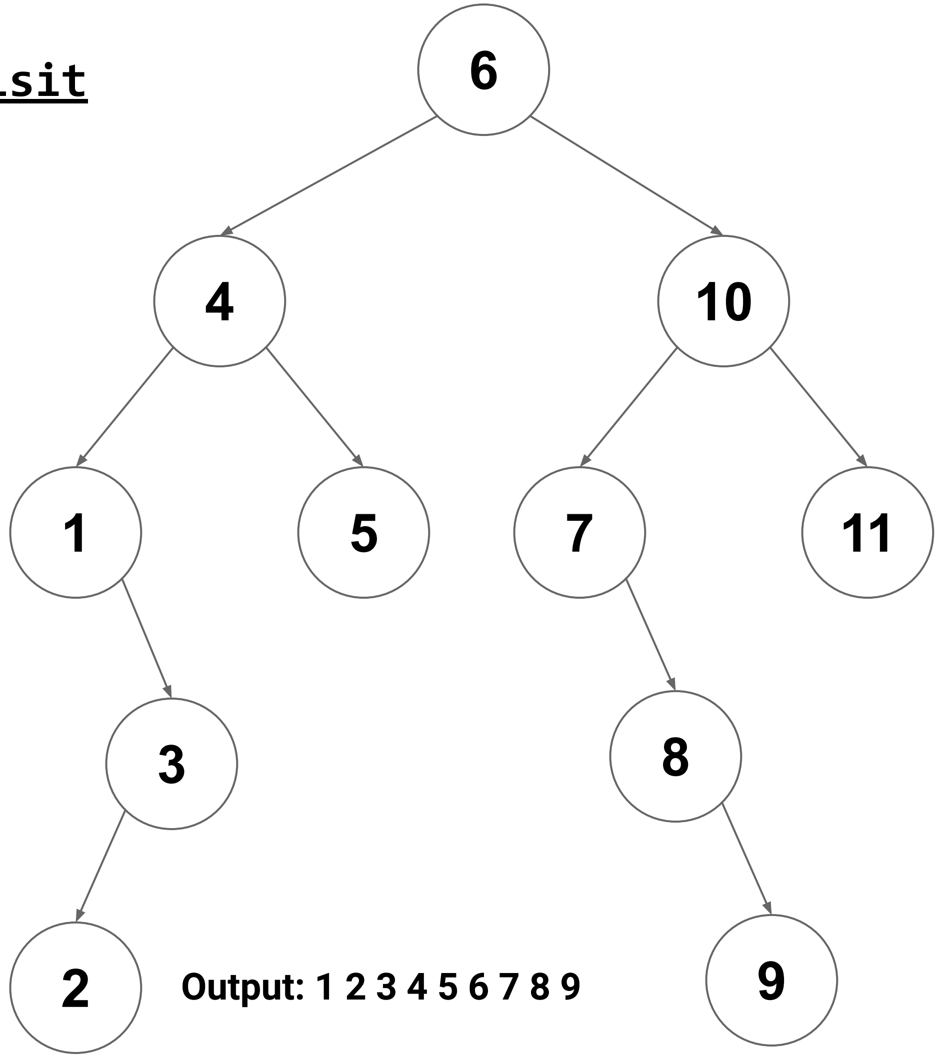
next pops the stack (8) and pushes the right subtree (9)



In-Order Traversal with an Iterator

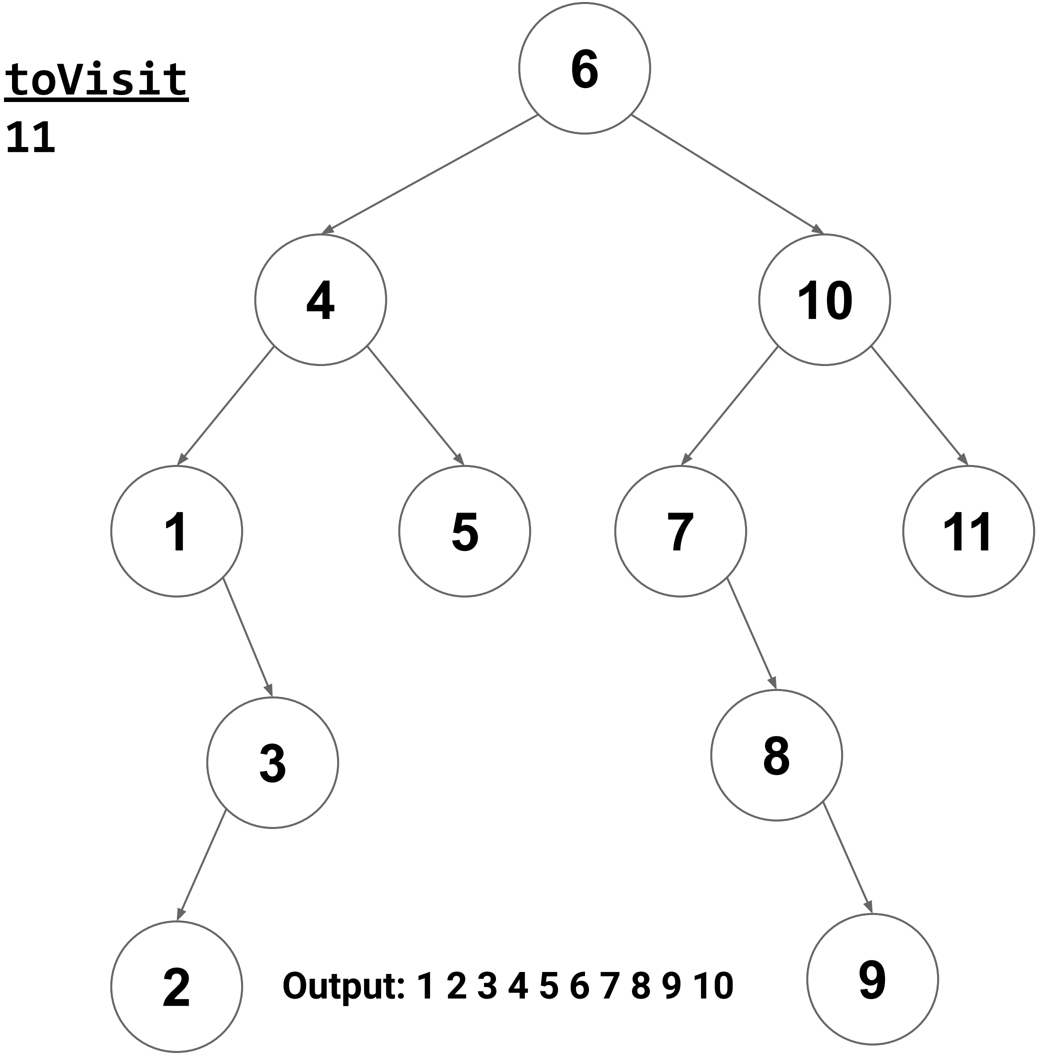
next pops the stack (9) and pushes the right subtree (nothing)

toVisit
10



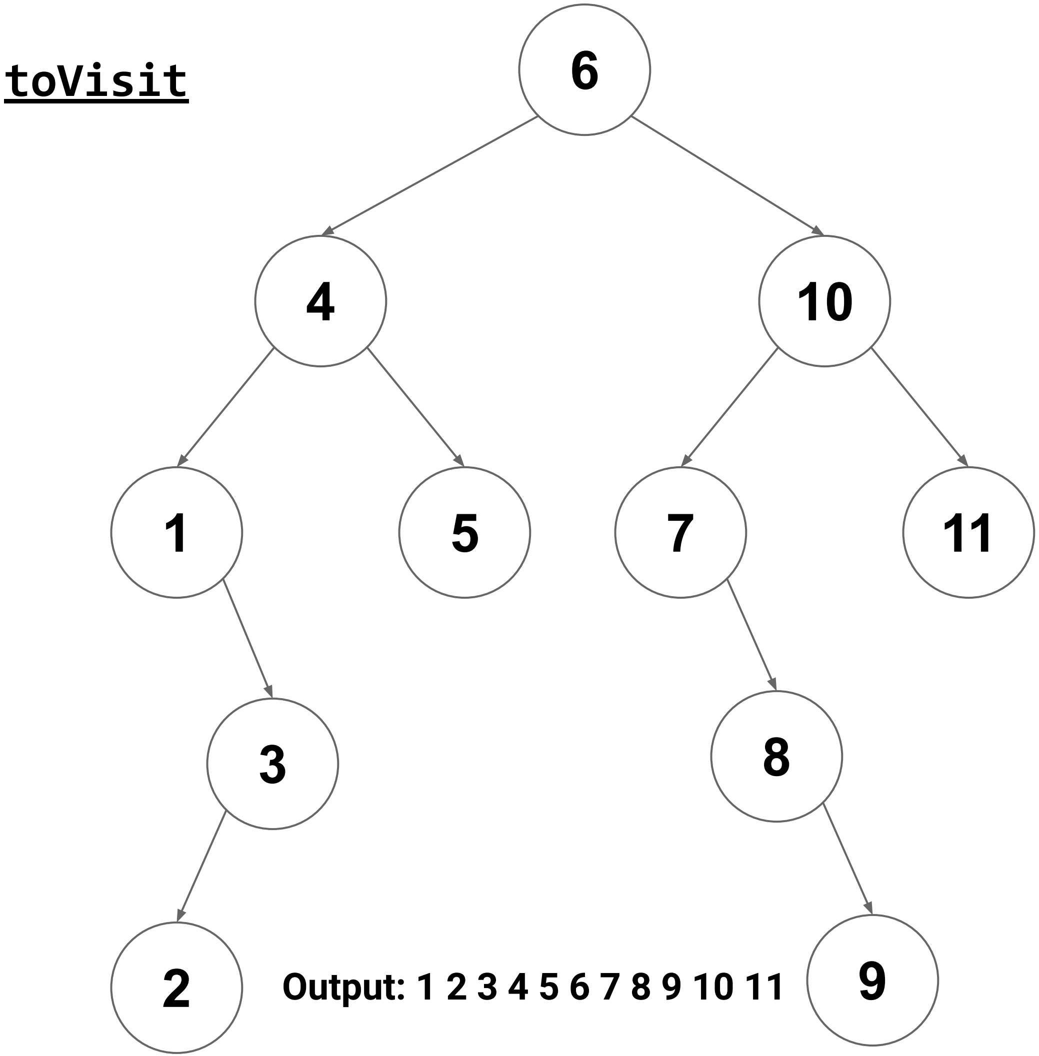
In-Order Traversal with an Iterator

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



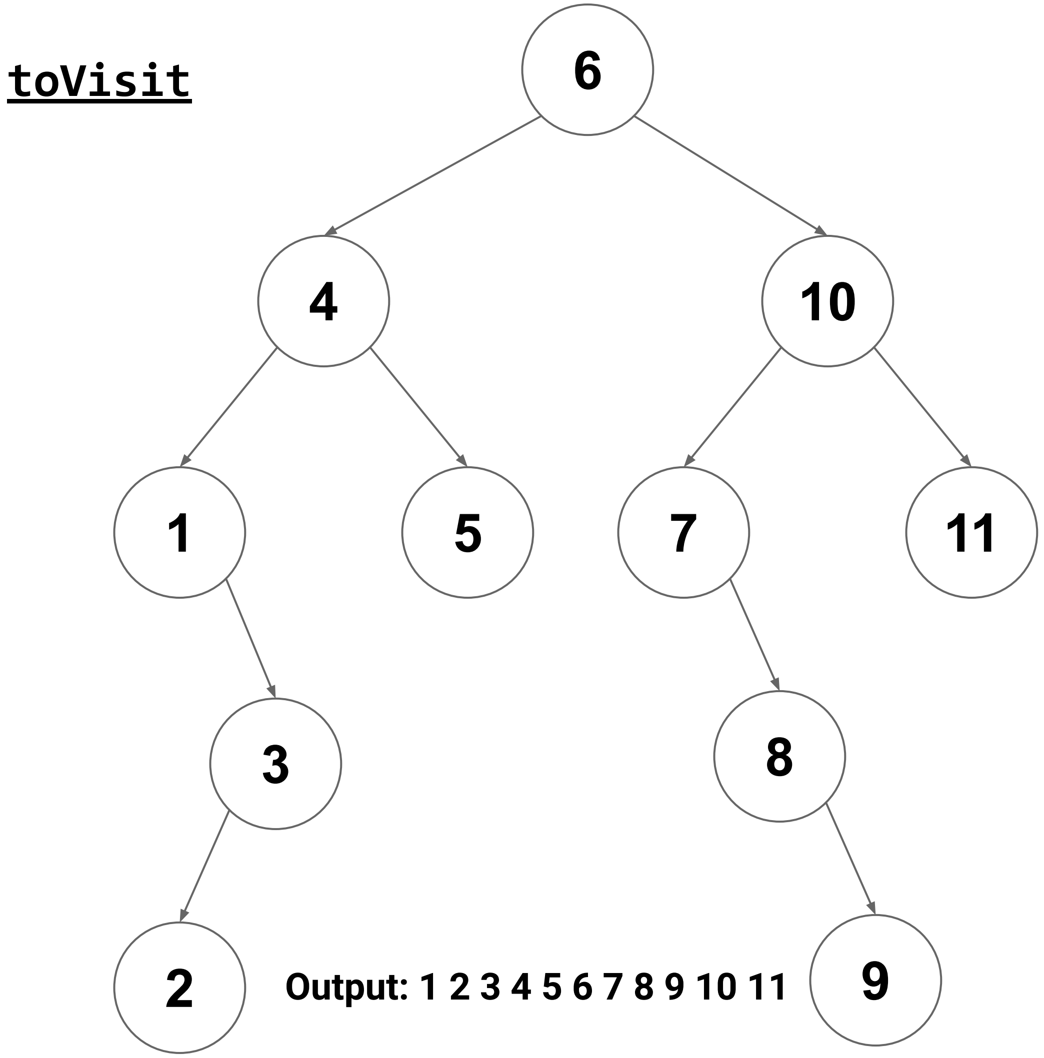
In-Order Traversal with an Iterator

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 $\boldsymbol{O}(\mathbf{1})$

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

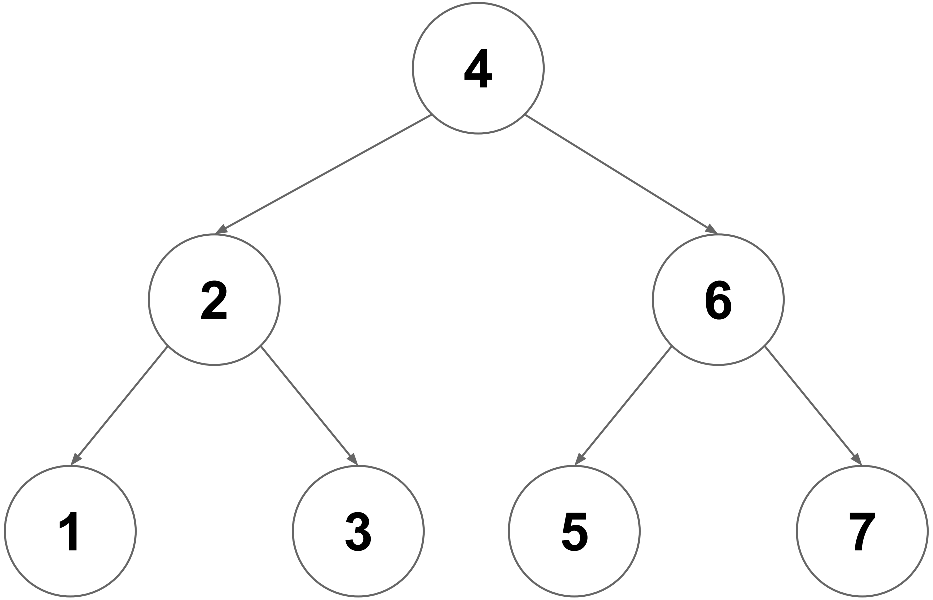
Operation	Runtime
`find`	$O(d)$
`insert`	$O(d)$
`remove`	$O(d)$

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

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

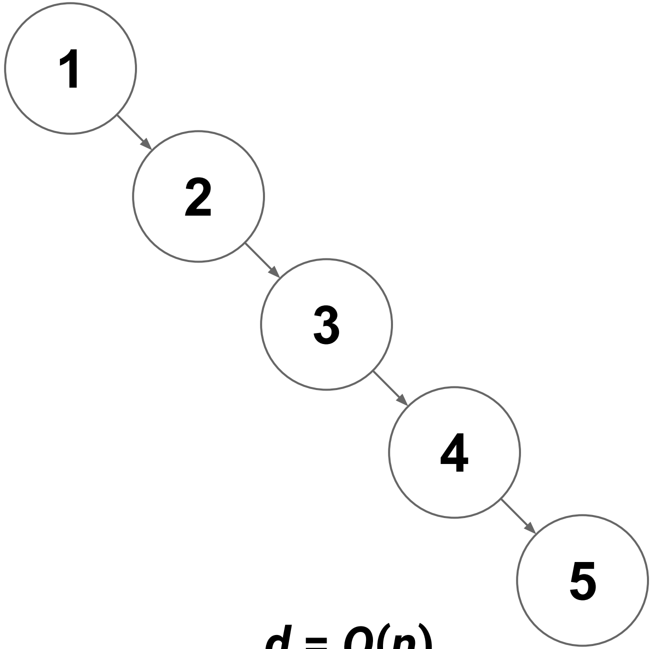
Tree Depth vs Size

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? **$|\text{height(left)} - \text{height(right)}| \leq 1$**

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")

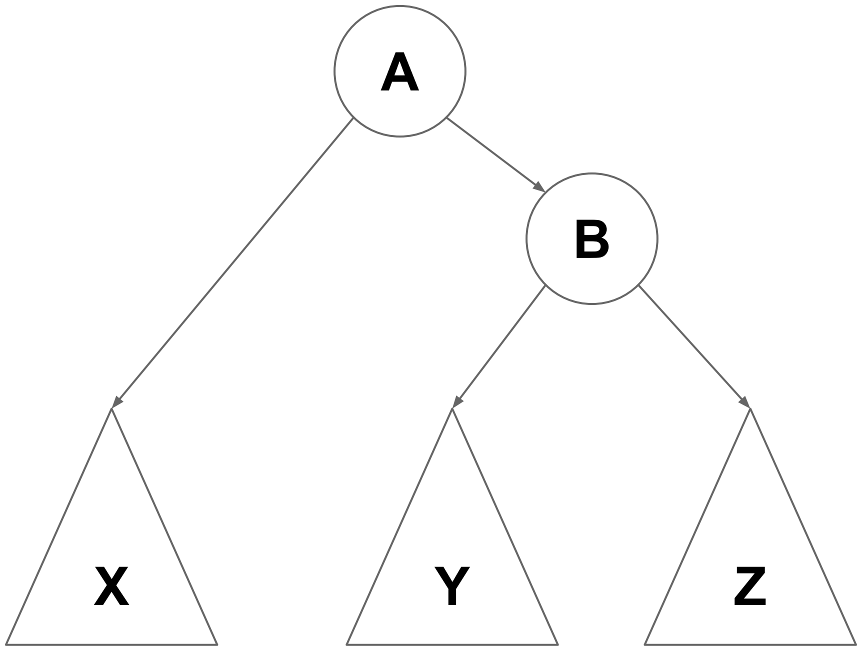
Option 2

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

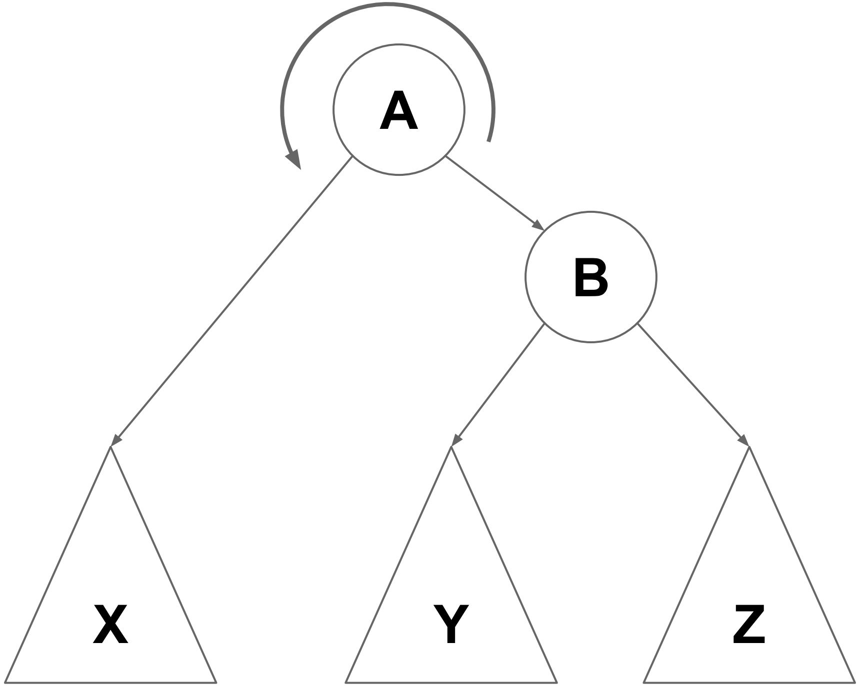
(Add a color to each node marking it as "red" or "black")

Ok…but how do we enforce this…?

Rebalancing Trees (rotations)

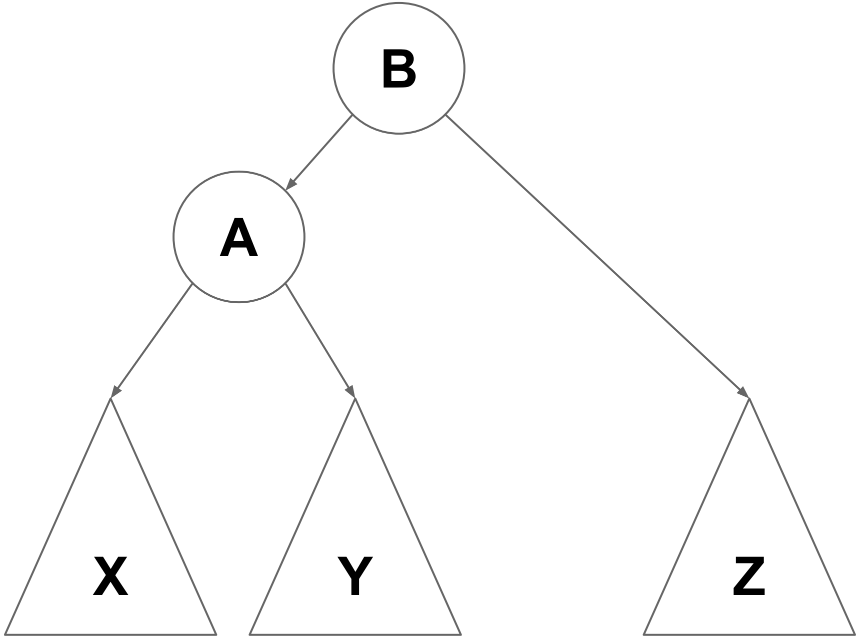


Rebalancing Trees (rotations)



Rotate(A, B)

Rebalancing Trees (rotations)

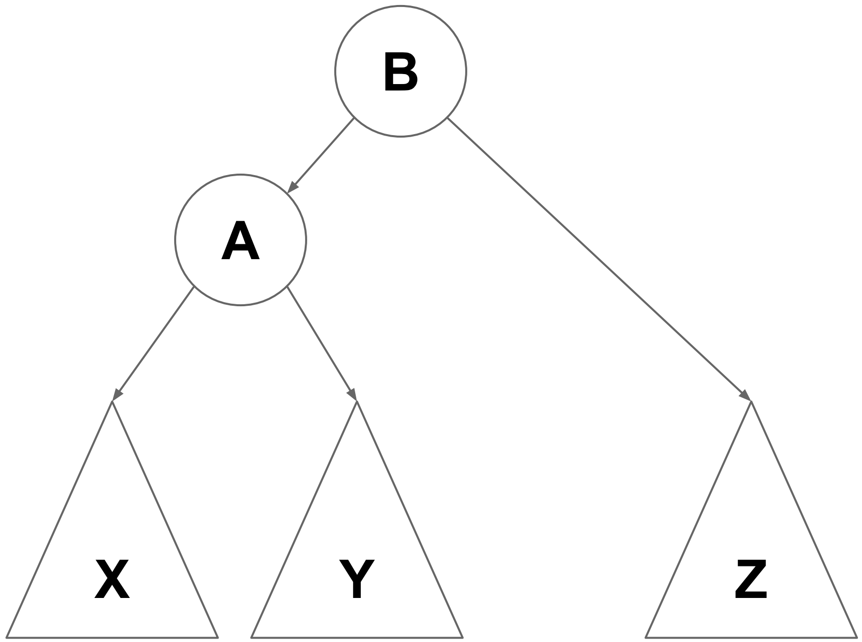


Rotate(A, B)

Rebalancing Trees (rotations)

A became **B**'s left child

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



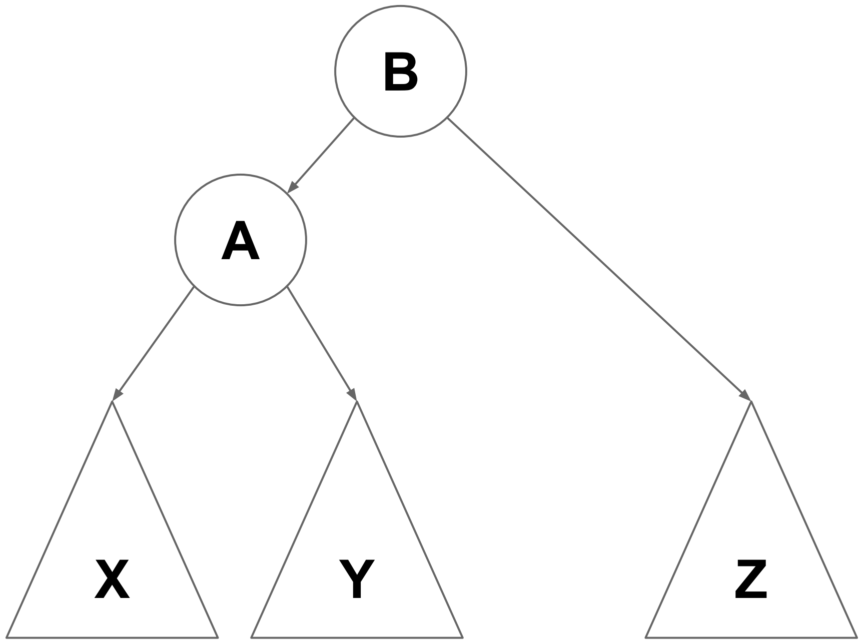
Rotate(A, B)

Rebalancing Trees (rotations)

A became **B**'s left child

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

Is ordering maintained?



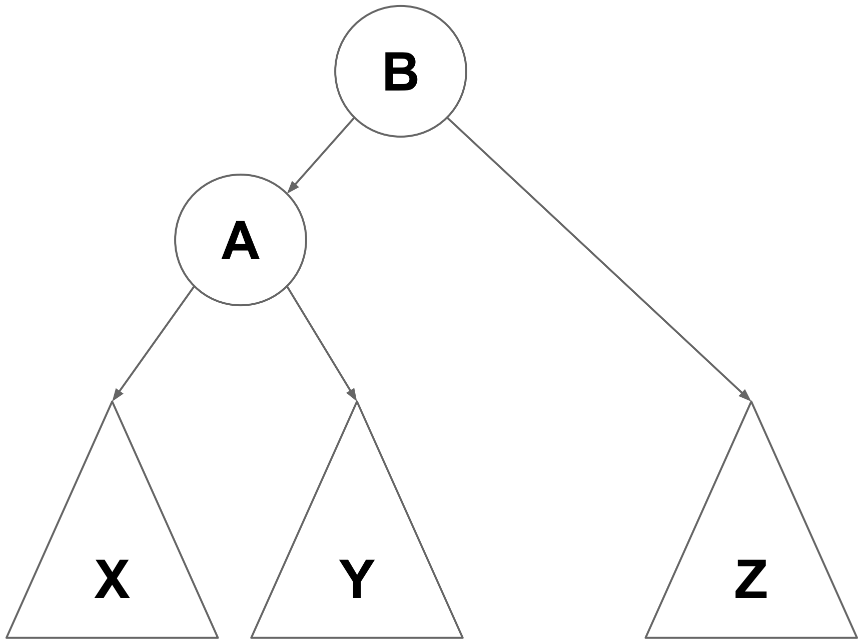
Rotate(A, B)

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)

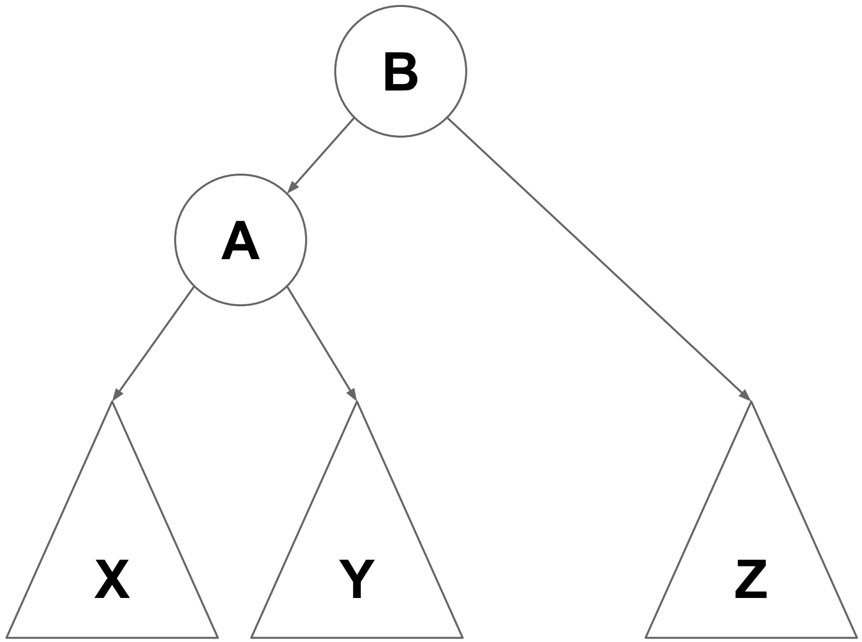
Rebalancing Trees (rotations)

A became **B**'s left child

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

Is ordering maintained? **Yes!**

Complexity?



Rotate(A, B)

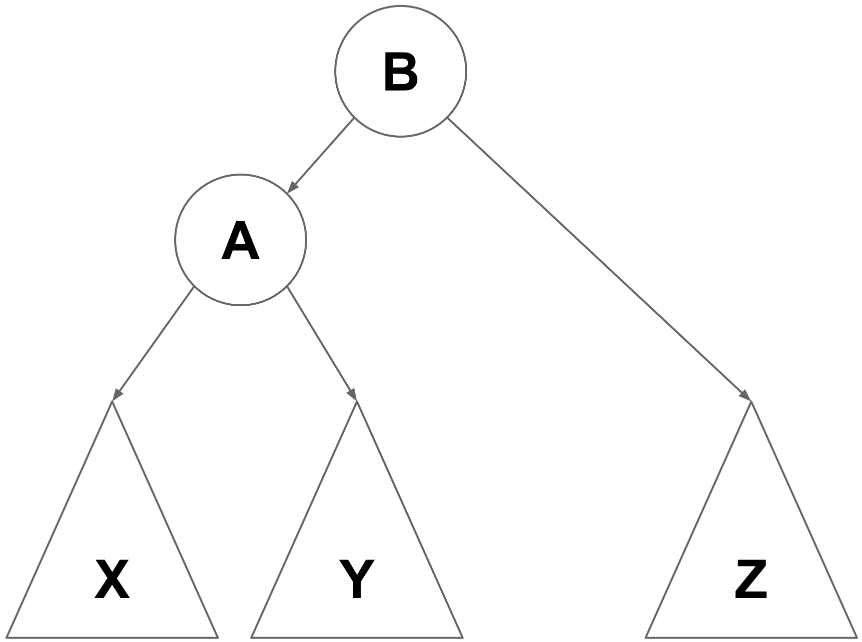
Rebalancing Trees (rotations)

A became **B**'s left child

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

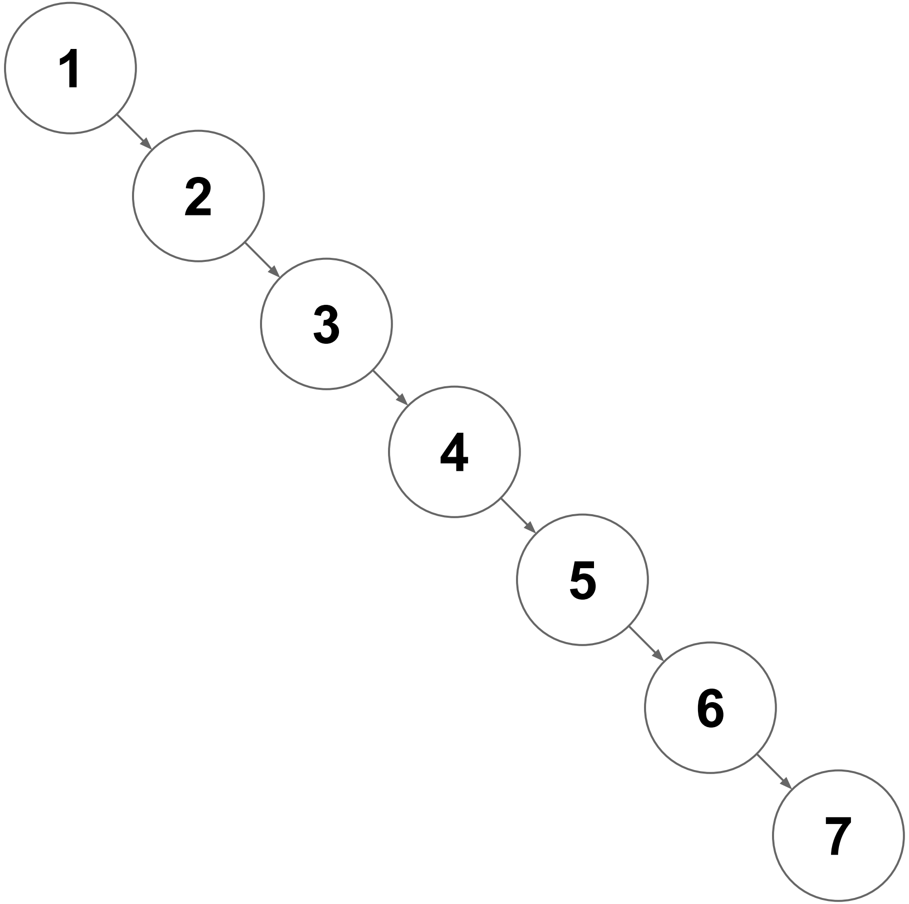
Is ordering maintained? **Yes!**

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



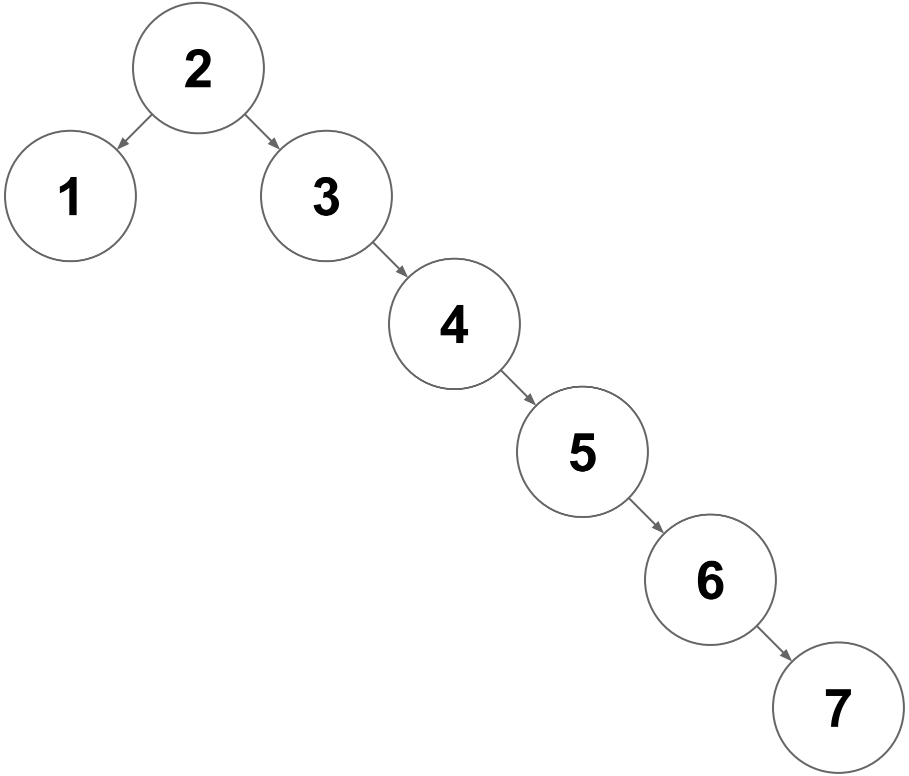
Rotate(A, B)

Rebalancing Trees



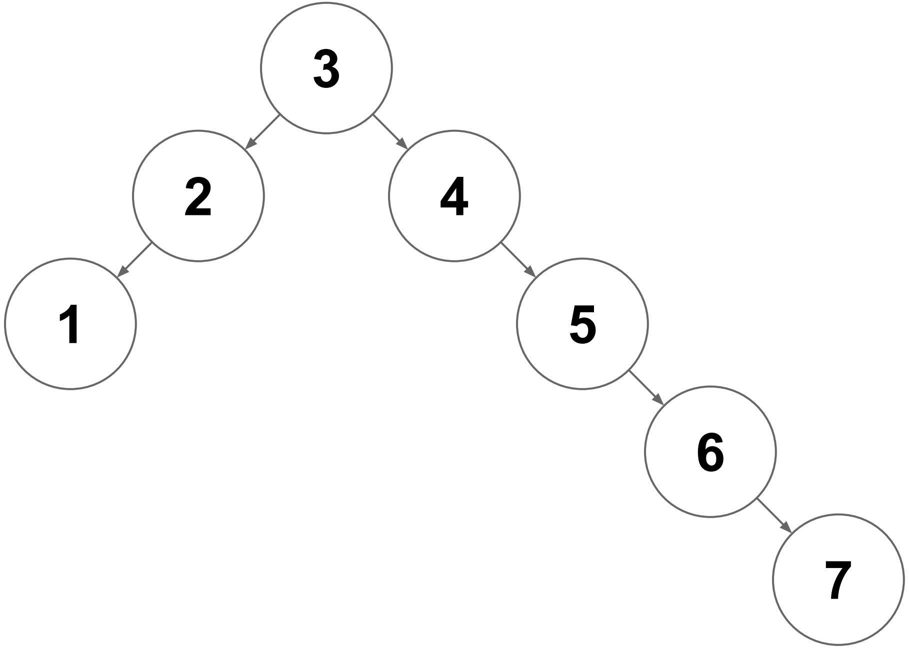
Rebalancing Trees

`Rotate(1,2)`



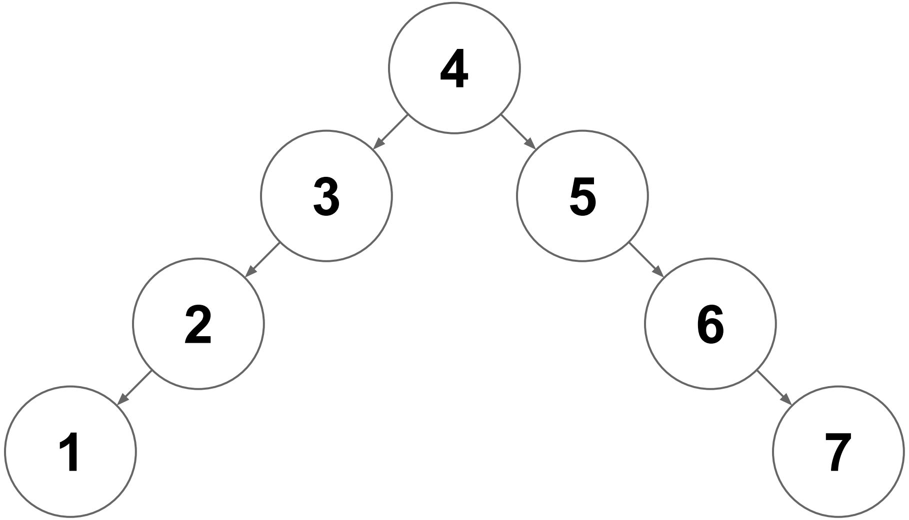
Rebalancing Trees

`Rotate(2,3)`



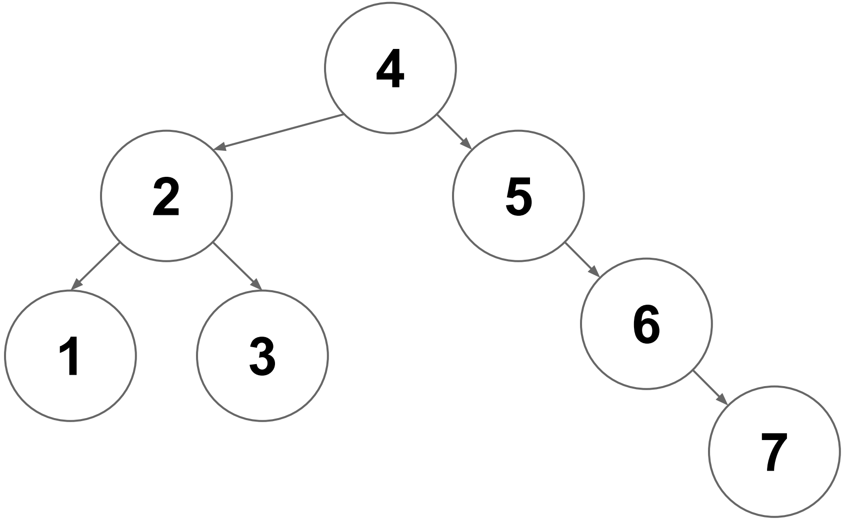
Rebalancing Trees

`Rotate(3,4)`



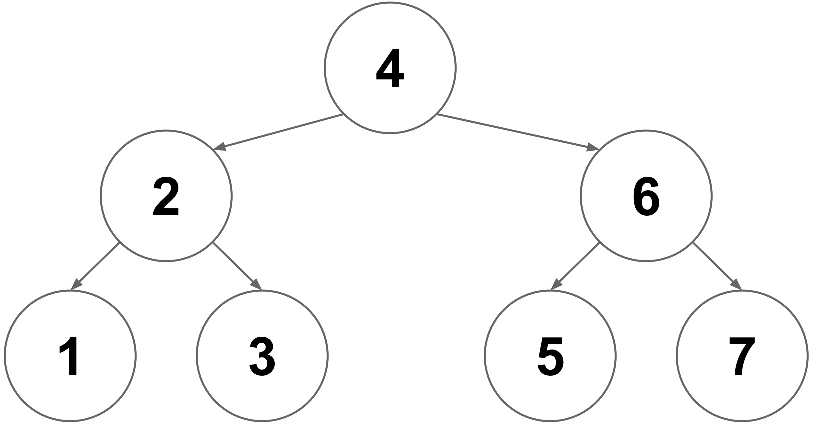
Rebalancing Trees

`Rotate(3,2)`



Rebalancing Trees

`Rotate(5,6)`



Next Time…

Enforcing Balance with AVL Trees…