Prepared by Julian García, based on material by
David Albrecht and María García de la Banda

Lecture 31
More Binary Trees

FIT 1008
Introduction to Computer Science

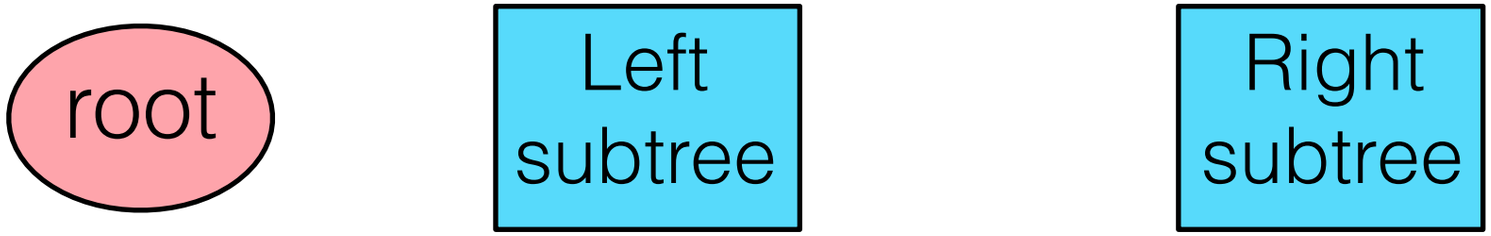
```
def add(self, item, position_bitstring):
    bitstring_iterator = iter(position_bitstring)
    self.root = self._add_aux(self.root, item, bitstring_iterator)

def _add_aux(self, current, item, bitstring_iterator):
    if current is None:
        current = TreeNode()
    try:
        bit = next(bitstring_iterator)
        if bit == "0":
            current.left = self._add_aux(current.left, item, bitstring_iterator)
        elif bit == "1":
            current.right = self._add_aux(current.right, item, bitstring_iterator)
    except StopIteration:
        current.item = item
    return current
```

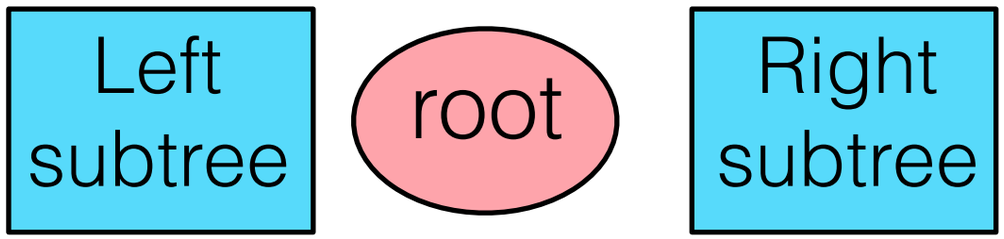
Traversal

- Systematic way of **visiting**/processing **all the nodes**
- **Methods**: Preorder, Inorder, and Postorder
- They **all** traverse the left subtree before the right subtree. It's all about the **position of the root**.

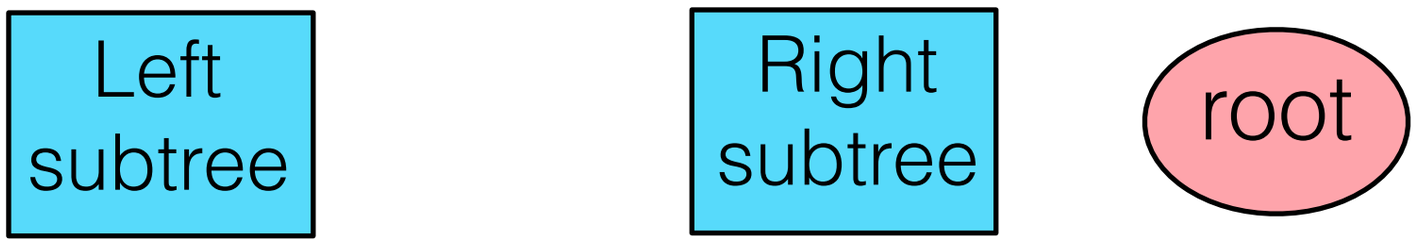
Preorder



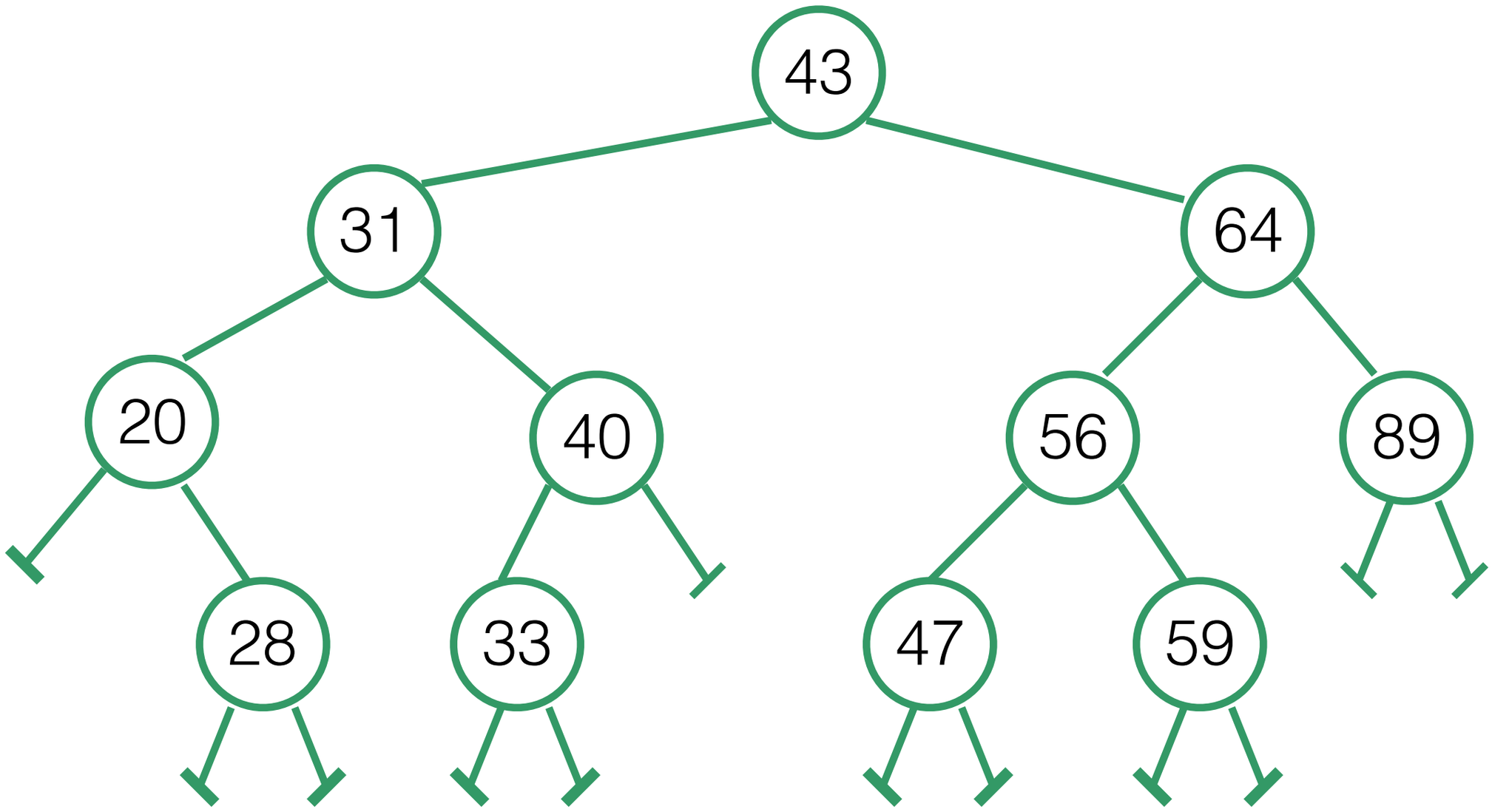
Inorder



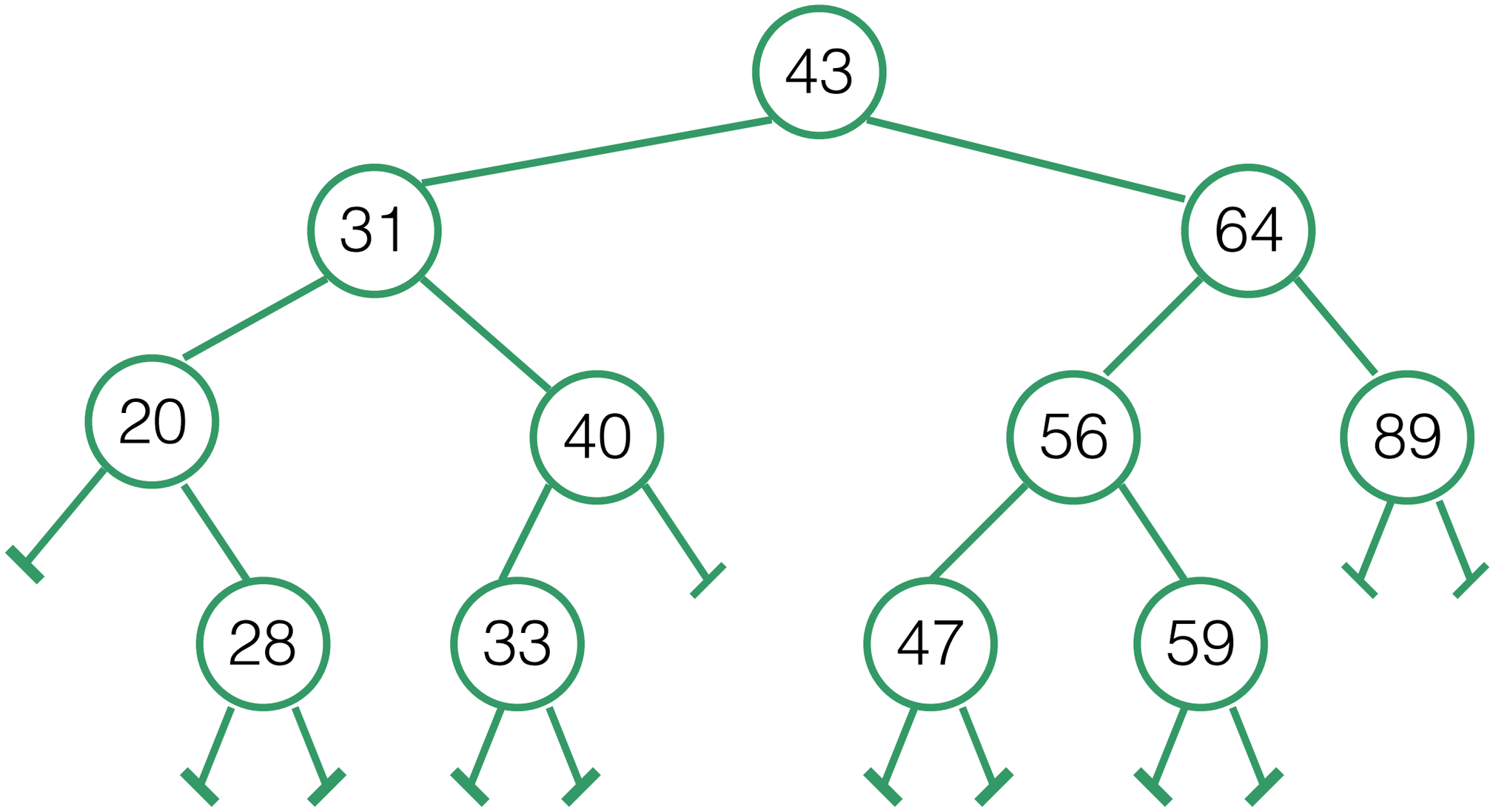
Postorder



Example: Preorder



Example: Preorder



43	31	20	28	40	33	64	56	47	59	89

Print Preorder Traversal

1) Print the **root** node

2) Traverse the **left** subtree

3) Traverse the **right** subtree

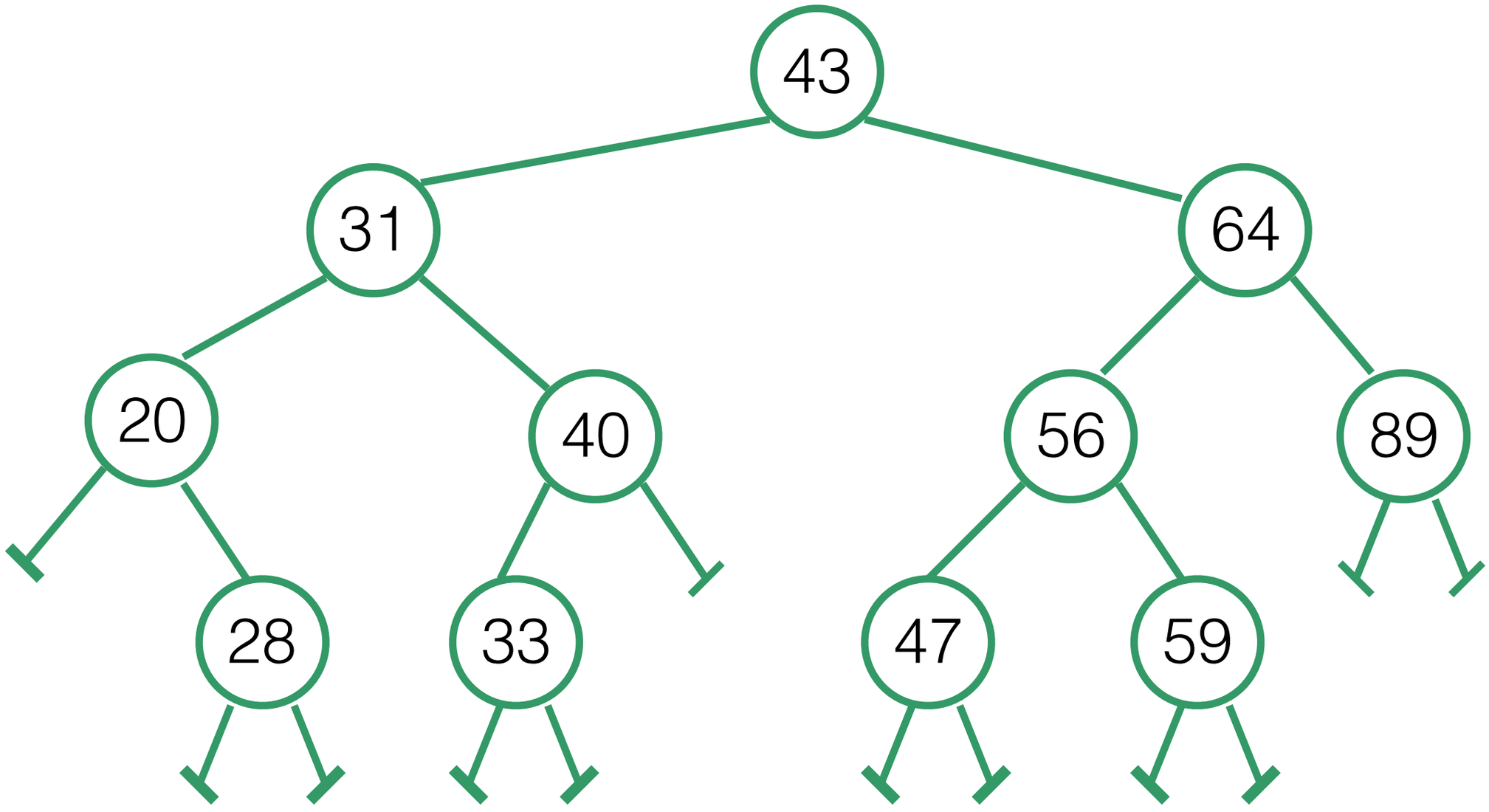
```
def print_preorder(self):
```

Print Preorder Traversal

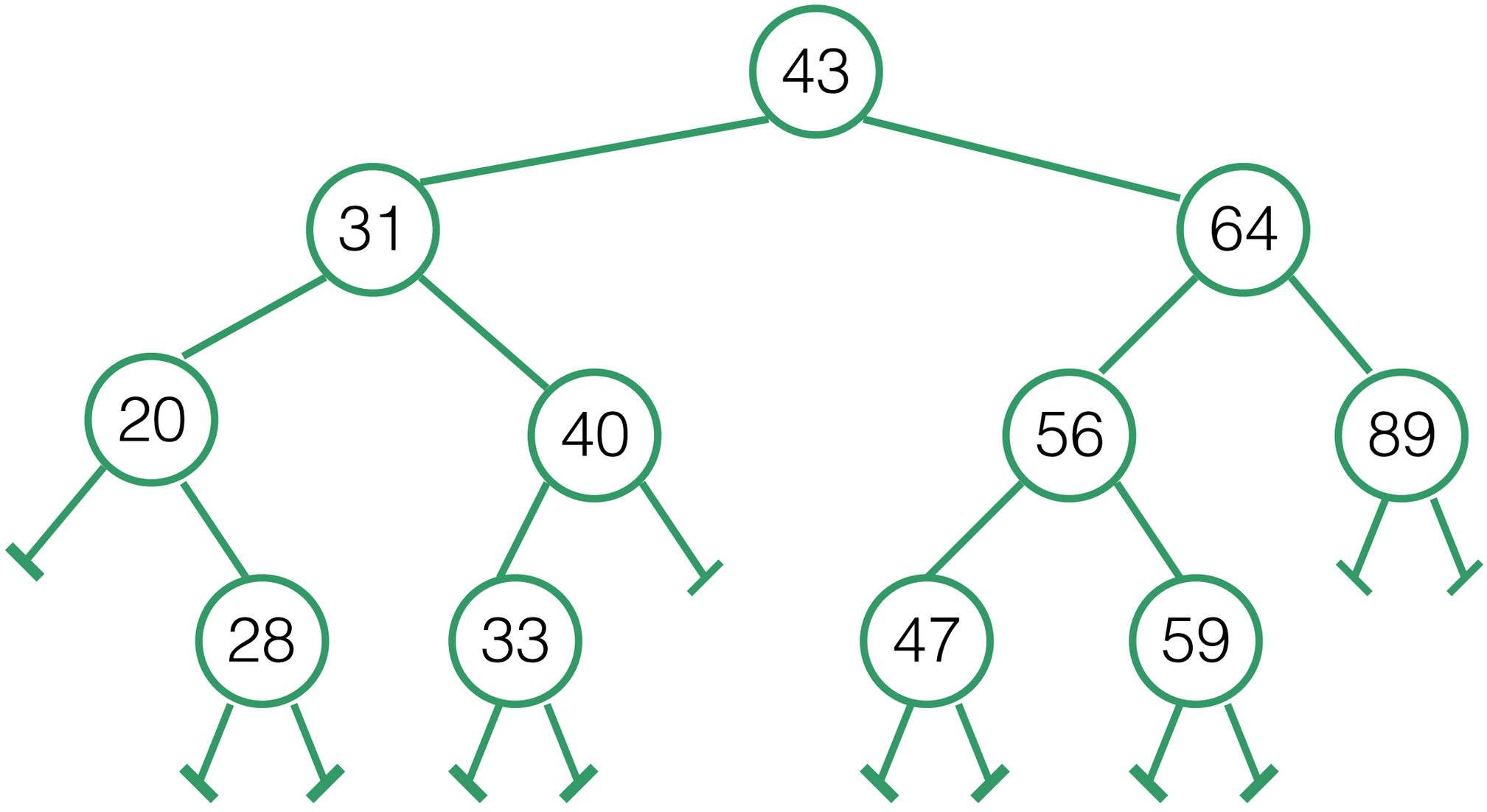
```
def print_preorder(self):
    self._print_preorder_aux(self.root)

def _print_preorder_aux(self, current):
    if current is not None:  # if not a base case
        print(current)
        self._print_preorder_aux(current.left)
        self._print_preorder_aux(current.right)
```

Example: Inorder



Example: Inorder



20	28	31	33	40	43	47	56	59	64	89

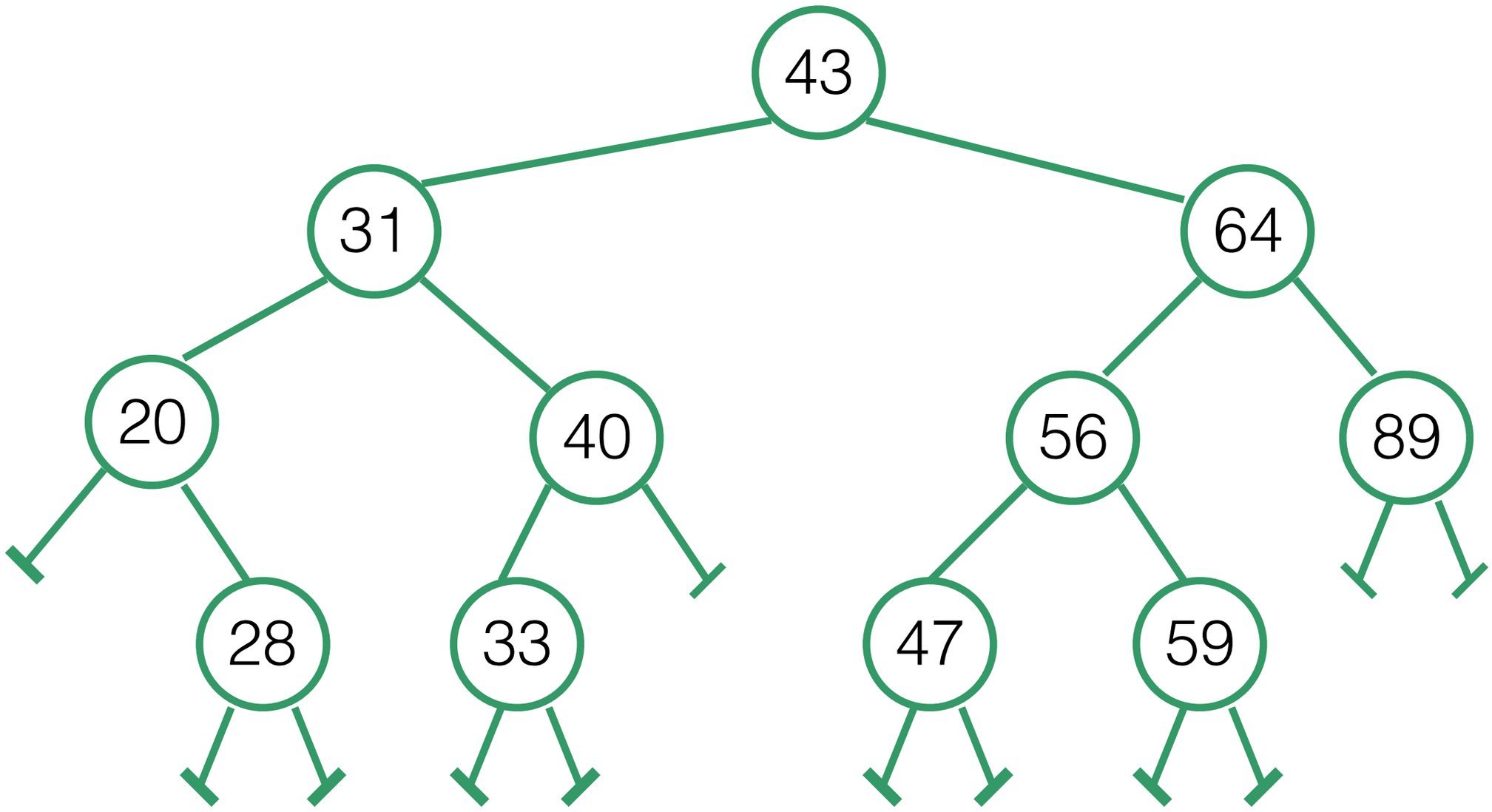
Print In-order Traversal

1) Traverse the **left** subtree
2) Print the **root** node
3) Traverse the **right** subtree

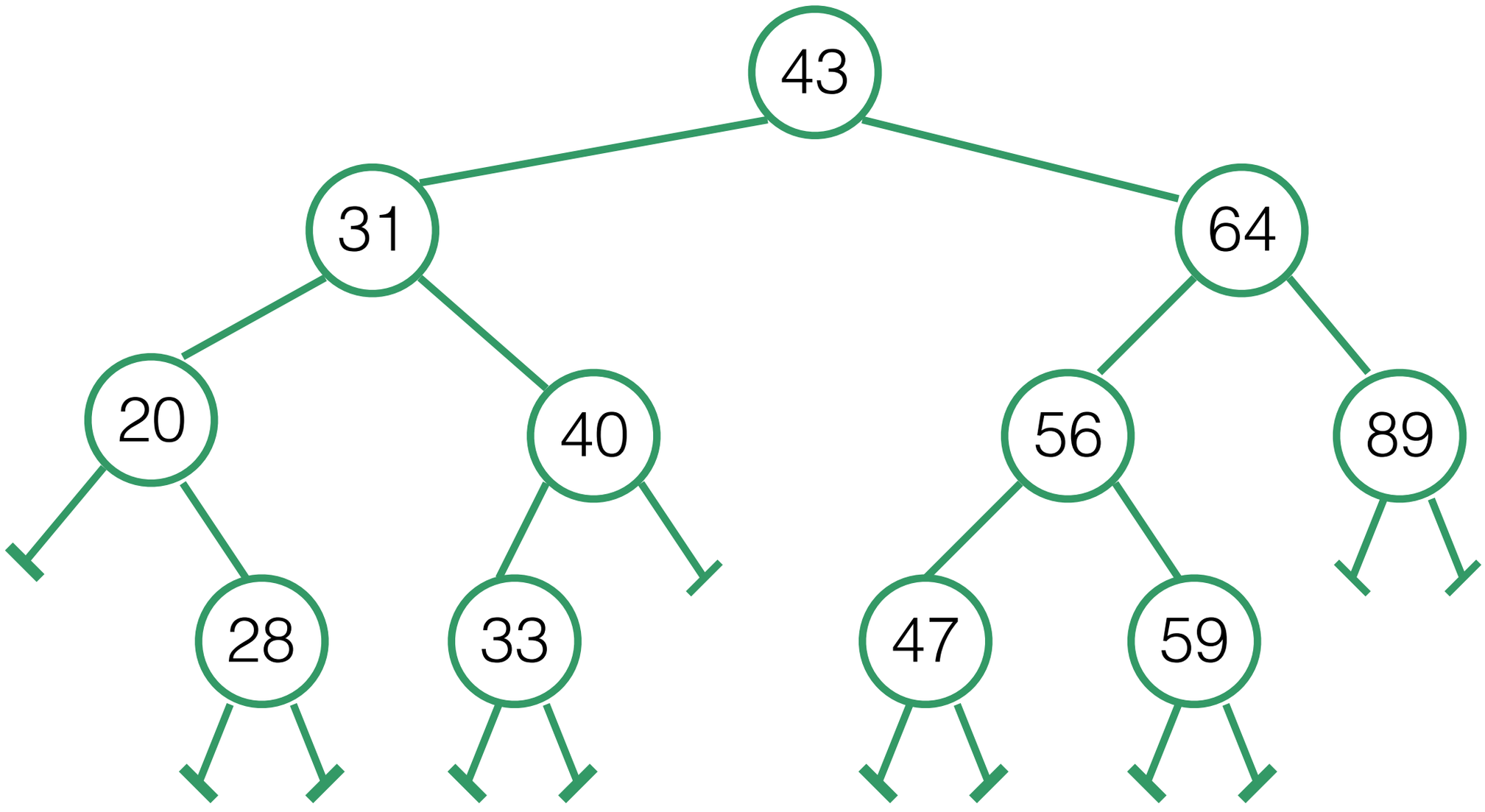
```
def print_inorder(self):
    self._print_inorder_aux(self.root)

def _print_inorder_aux(self, current):
    if current is not None:  # if not a base case
        self._print_inorder_aux(current.left)
        print(current)
        self._print_inorder_aux(current.right)
```

Example: Postorder



Example: Postorder



28	20	33	40	31	47	59	56	89	64	43

Print Post-order Traversal

1) Traverse the **left** subtree
2) Traverse the **right** subtree
3) Print the **root** node

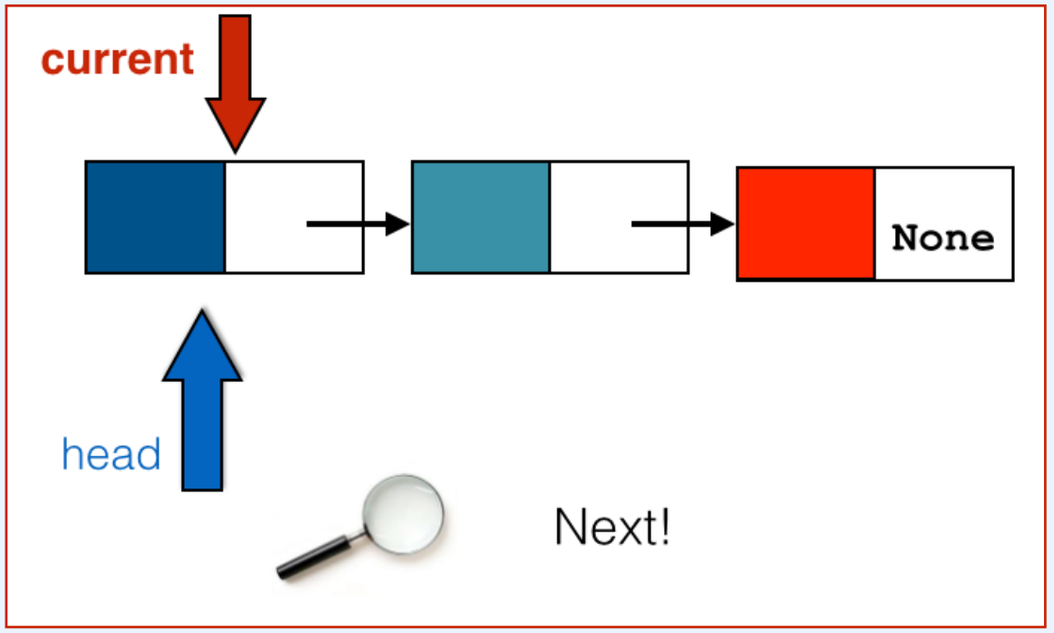
```
def print_postorder(self):
    self._print_postorder_aux(self.root)

def _print_postorder_aux(self, current):
    if current is not None:  # if not a base case
        self._print_postorder_aux(current.left)
        self._print_postorder_aux(current.right)
        print(current)
```

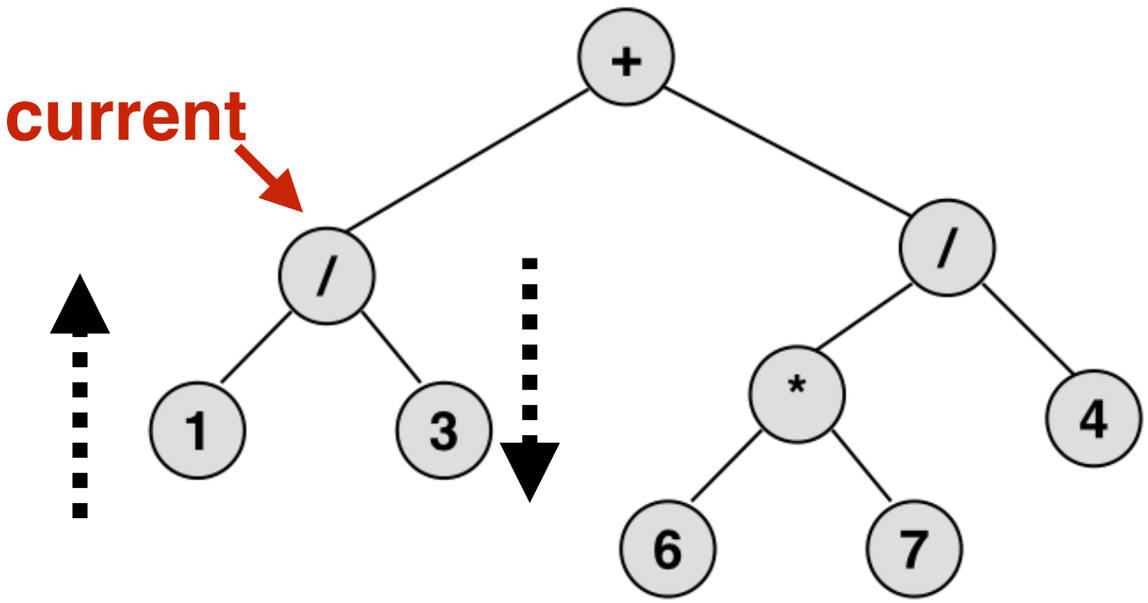
```python
class ListIterator:
    def __init__(self,head):
        self.current = head

    def __iter__(self):
        return self

    def __next__(self):
        if self.current is None:
            raise StopIteration
        else:
            item_required = self.current.item
            self.current = self.current.next
            return item_required
```

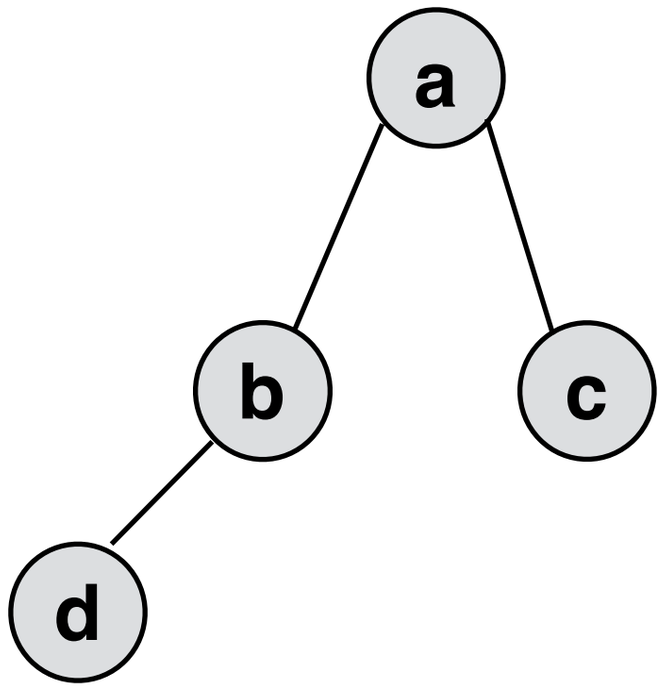


?



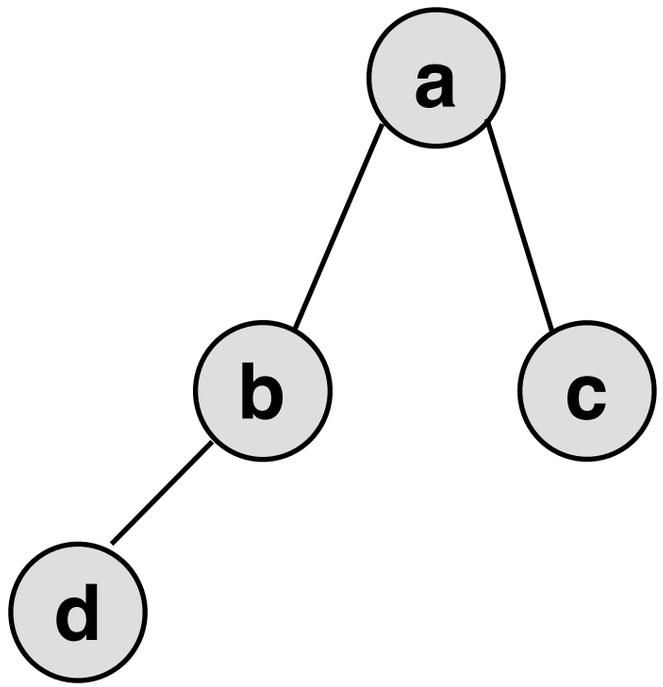
Pre-order Iterator

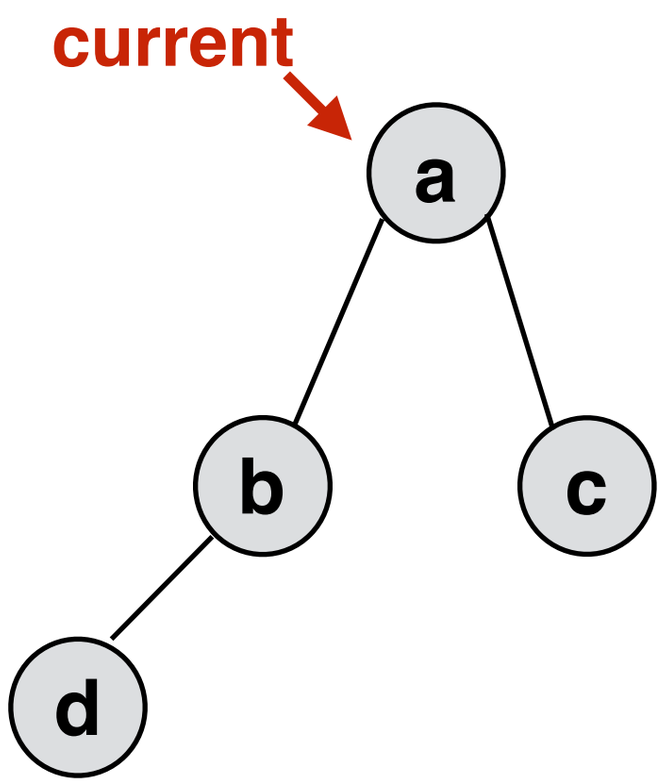
State of the **Iterator** on creation

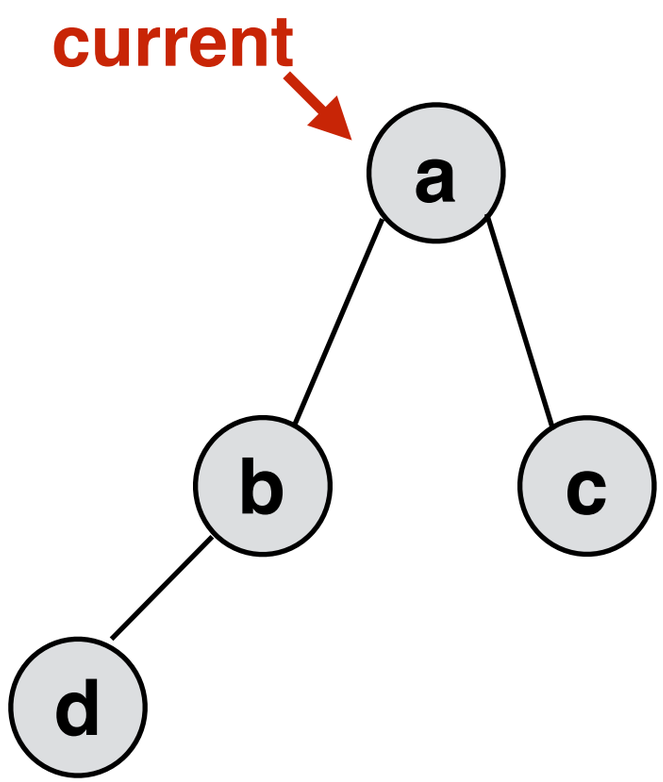


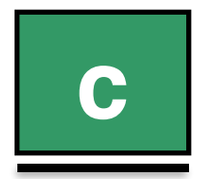
self.stack

Next!

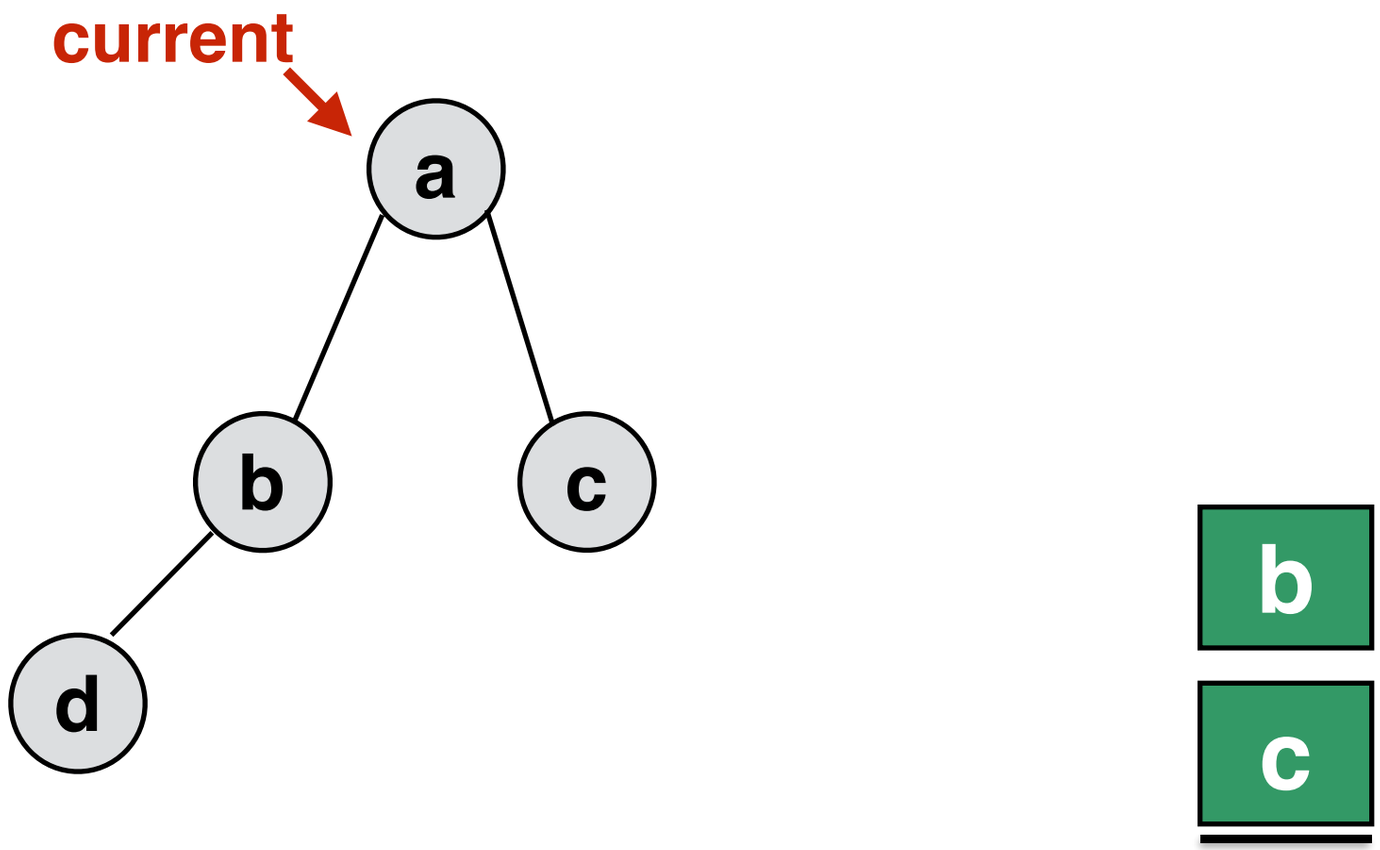



current
a
b
c
d



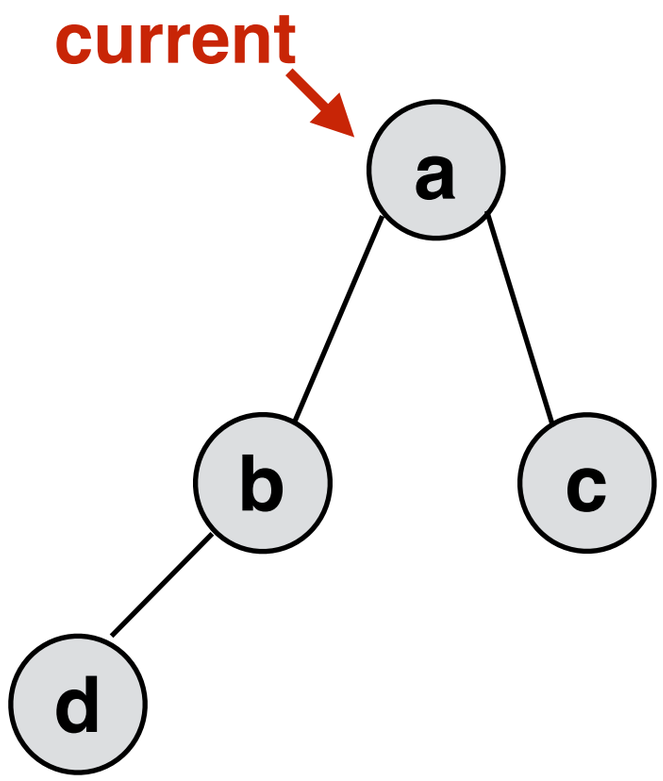


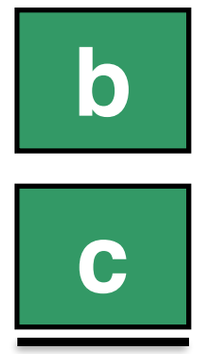
Push what is to the right of current.



Push what is to the left of current.

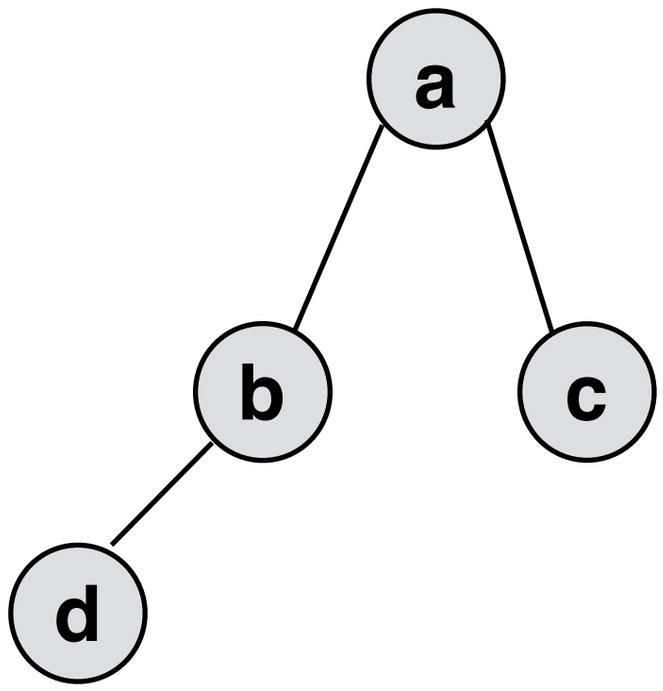
return current.item

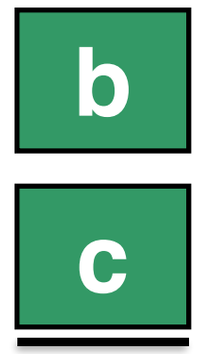




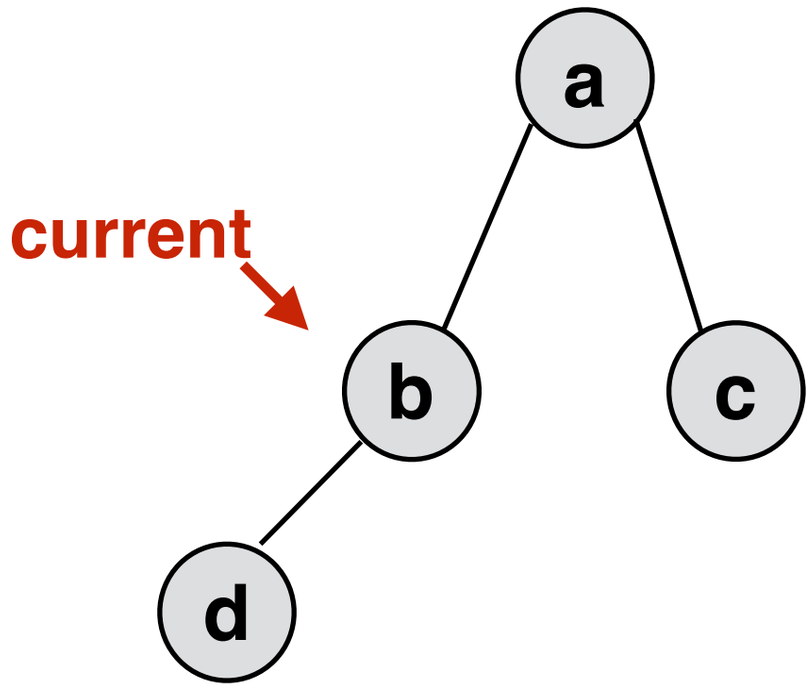
a

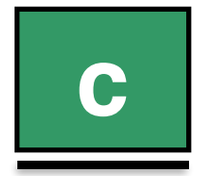
Next!





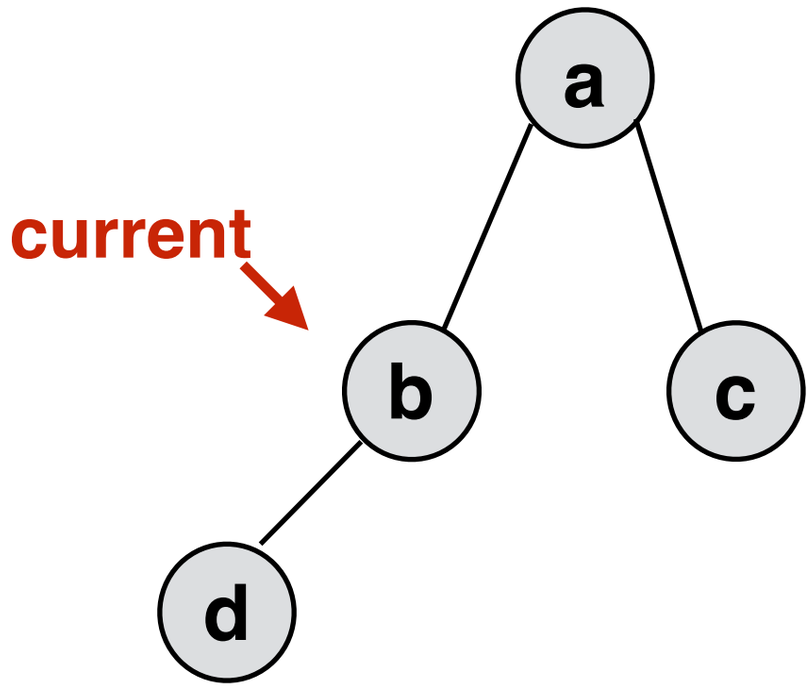
a

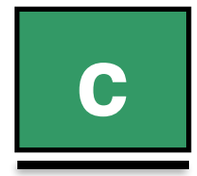




a

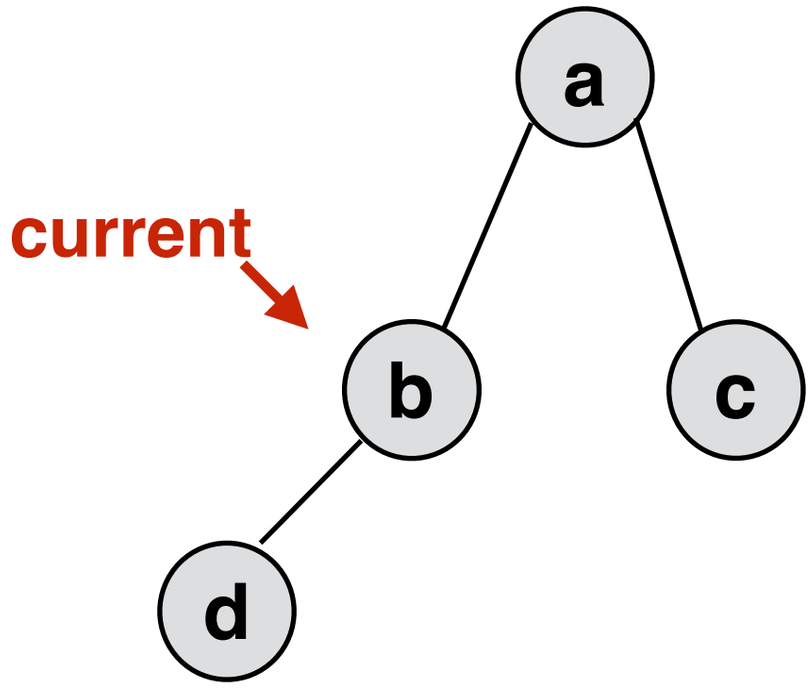
Nothing to push on right

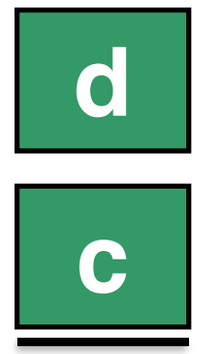




a

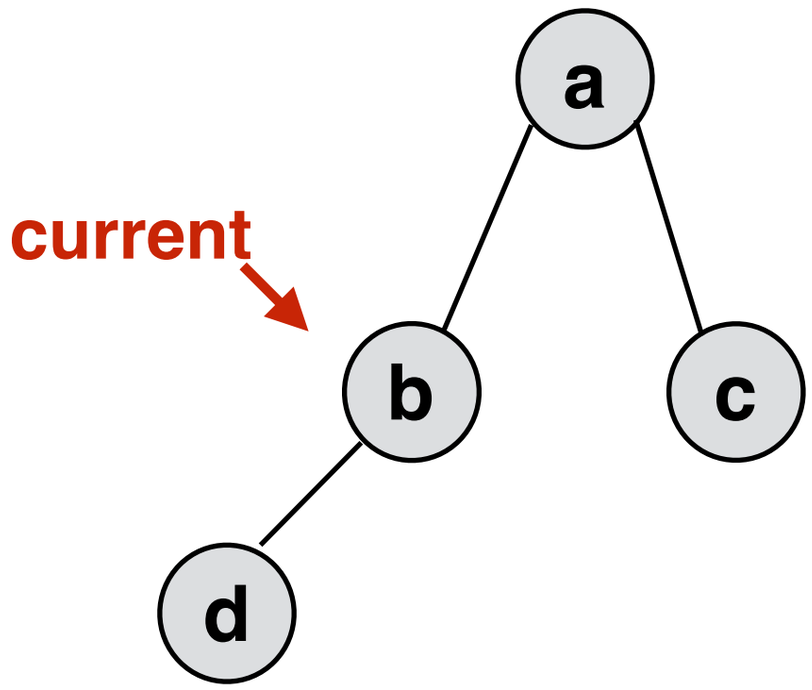
Push what is to the left of current.

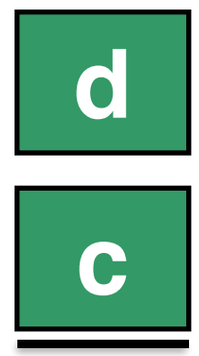




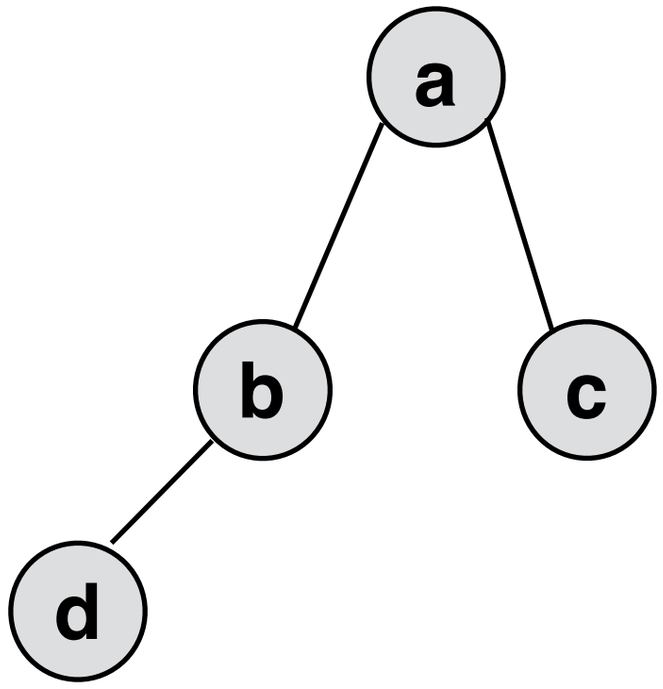
a

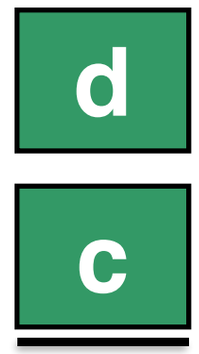
return current.item





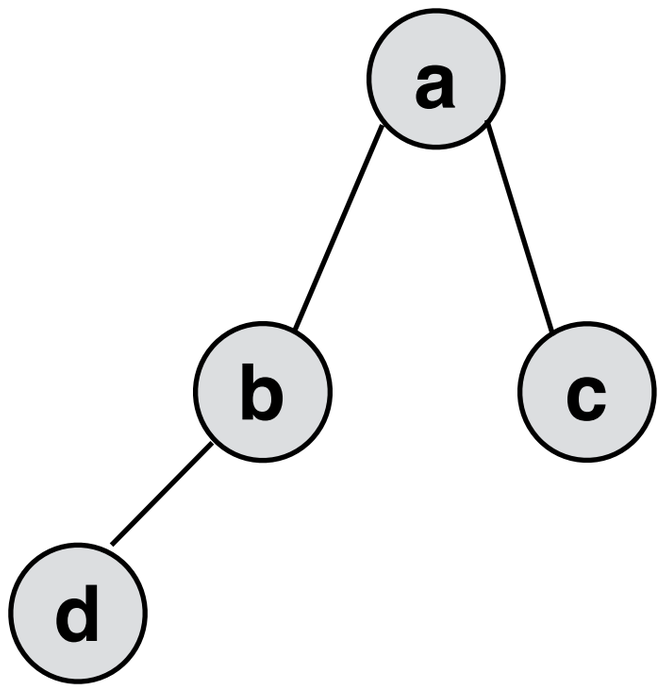
a b

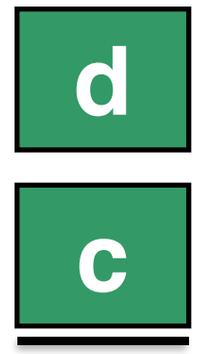




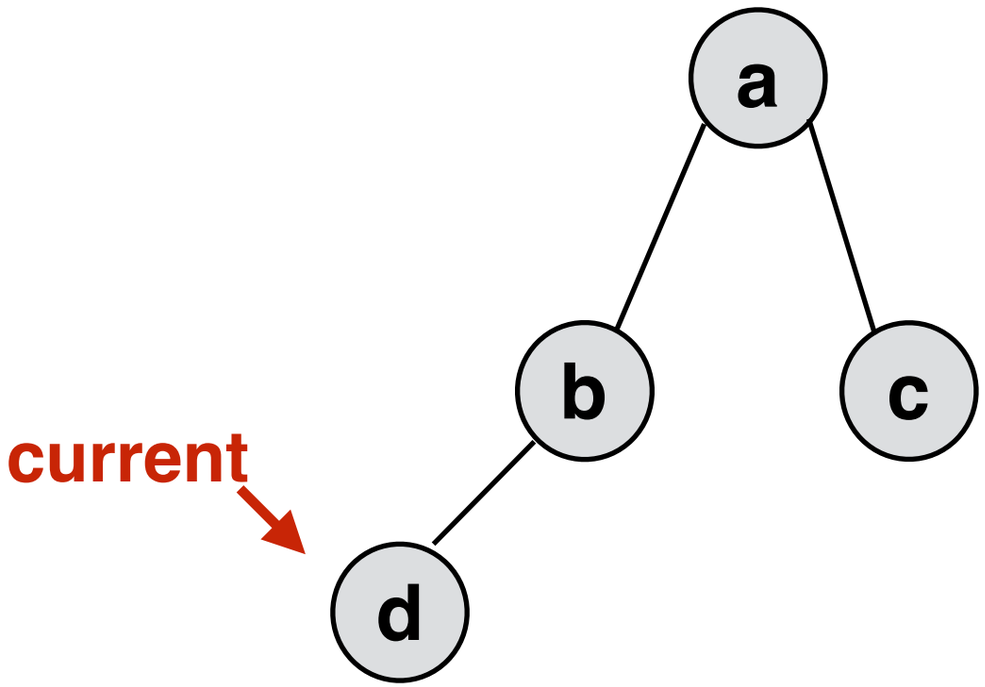
a b

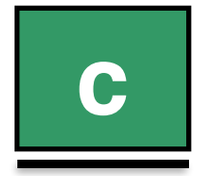
Next!



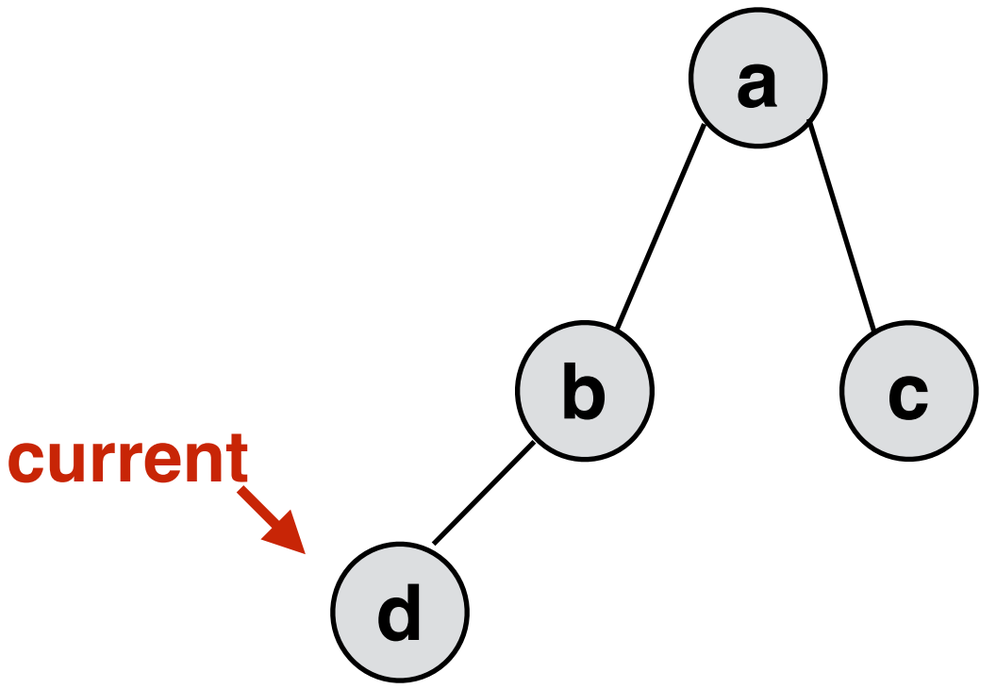


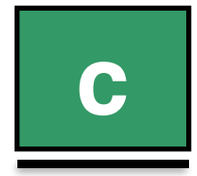
a b



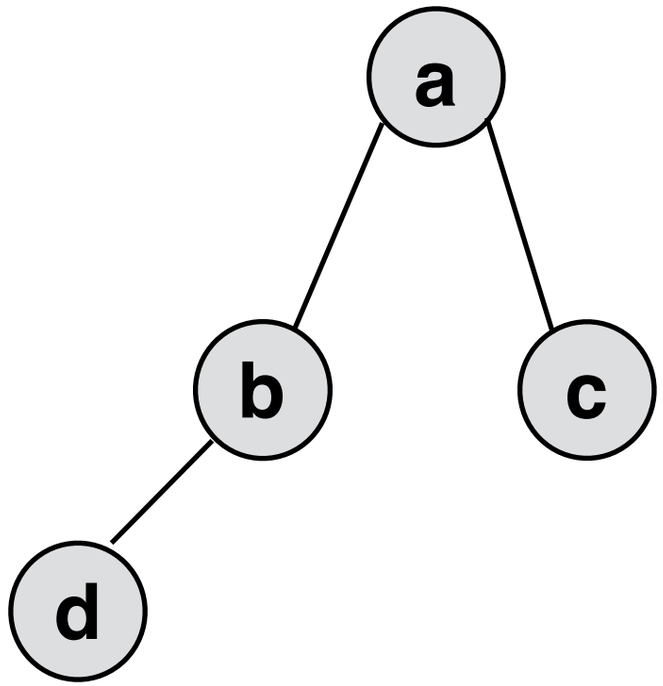


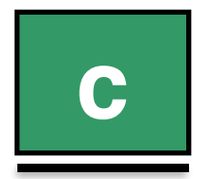
a b





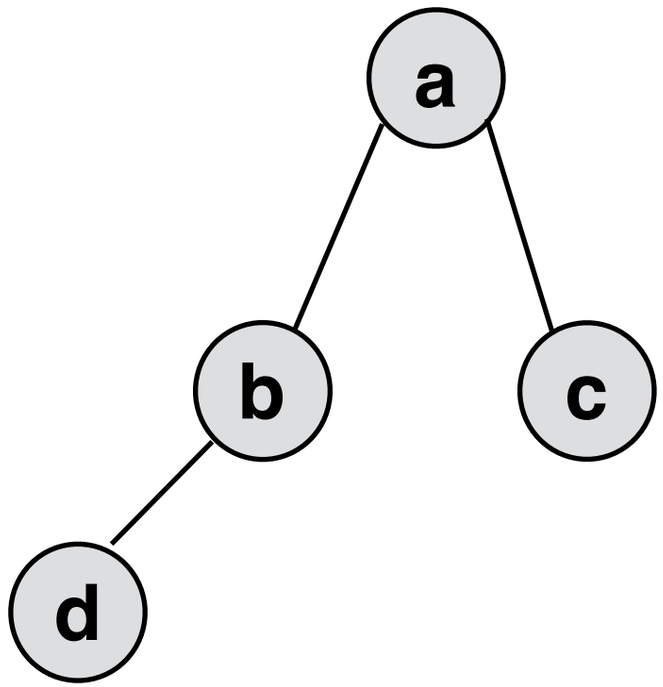
a b d





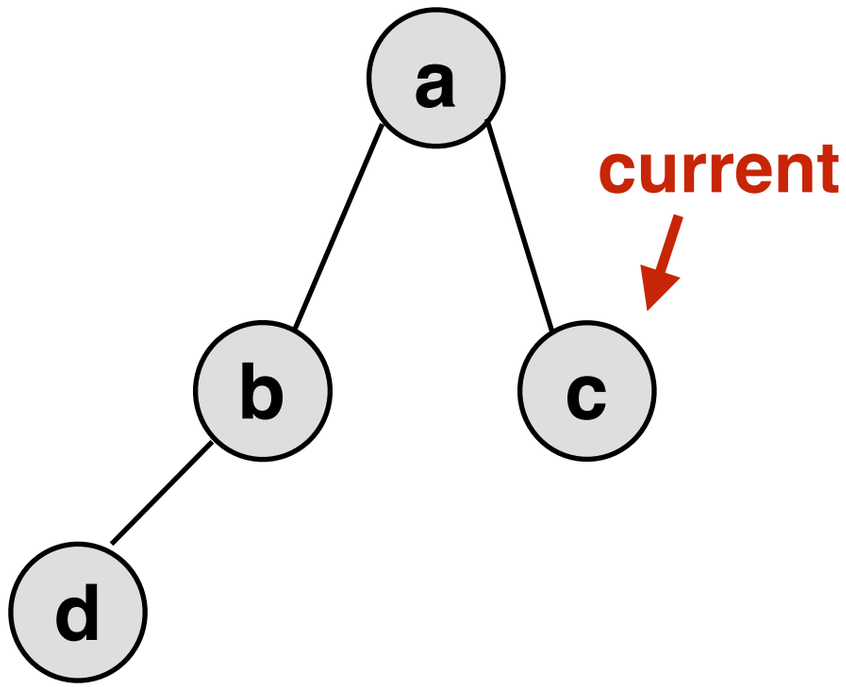
a b d

Next!



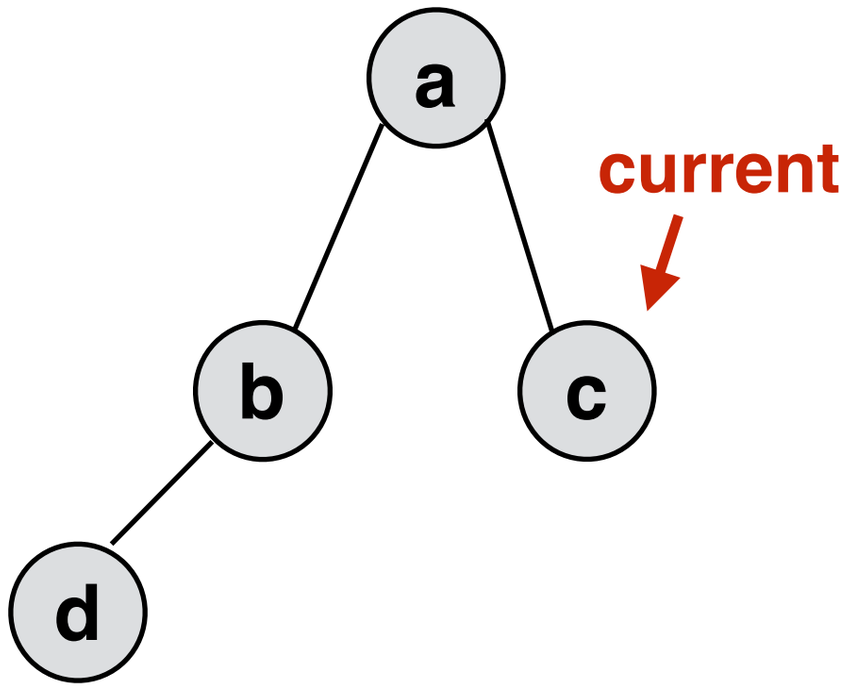
c

a b d

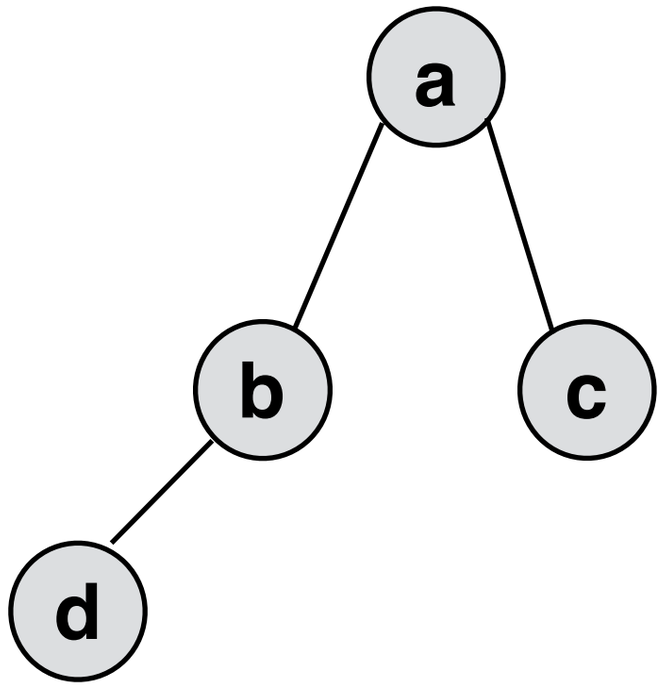


a b d

return current.item

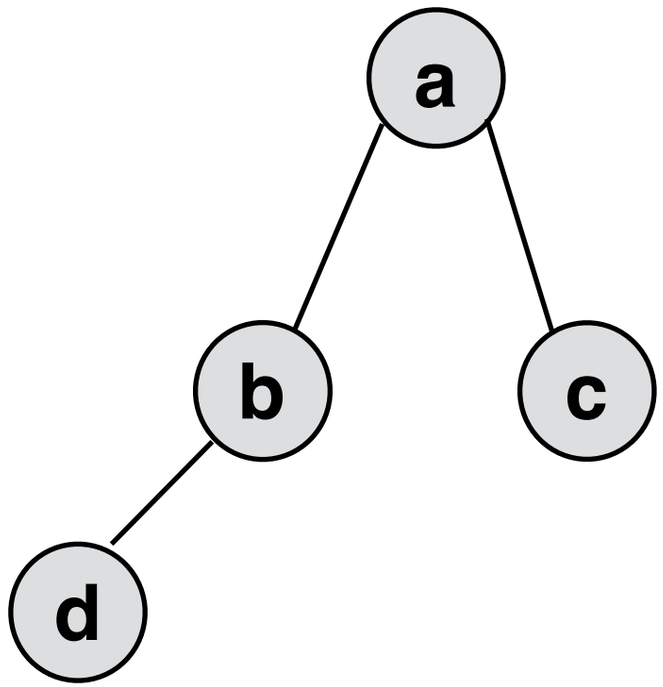


a b d c

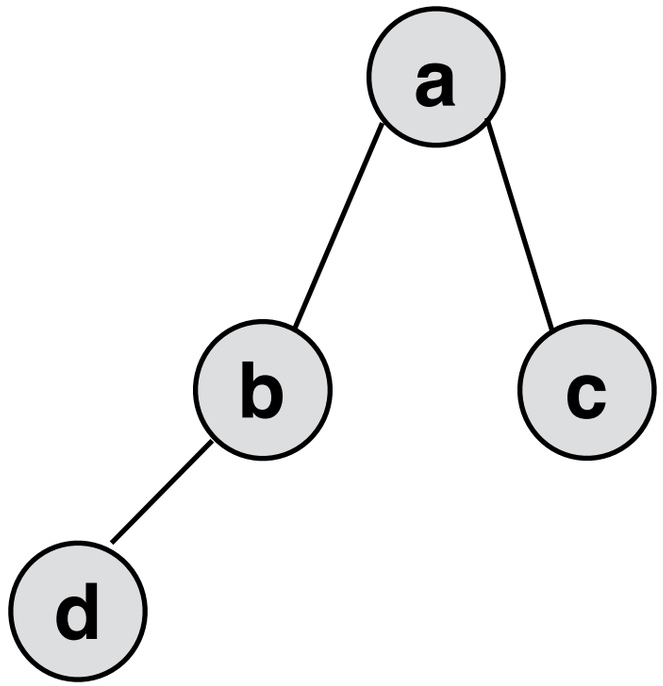


a b d c

Next!



a b d c



StopIteration

a b d c

preorder!

```
self.current = self.stack.pop()
self.stack.push(self.current.right)
self.stack.push(self.current.left)
return current
```

```
class PreOrderIteratorStack:

    def __init__(self, root):
        self.current = root
        self.stack = Stack()
        self.stack.push(root)

    def __iter__(self):
        return self

    def __next__(self):
        if self.stack.is_empty():
            raise StopIteration
        current = self.stack.pop()
        if current.right is not None:
            self.stack.push(current.right)
        if current.left is not None:
            self.stack.push(current.left)
        return current.item
```

```
my_tree.print_preorder()
```

```
2
5
3
```

```
for i in my_tree:
    print(i)
```

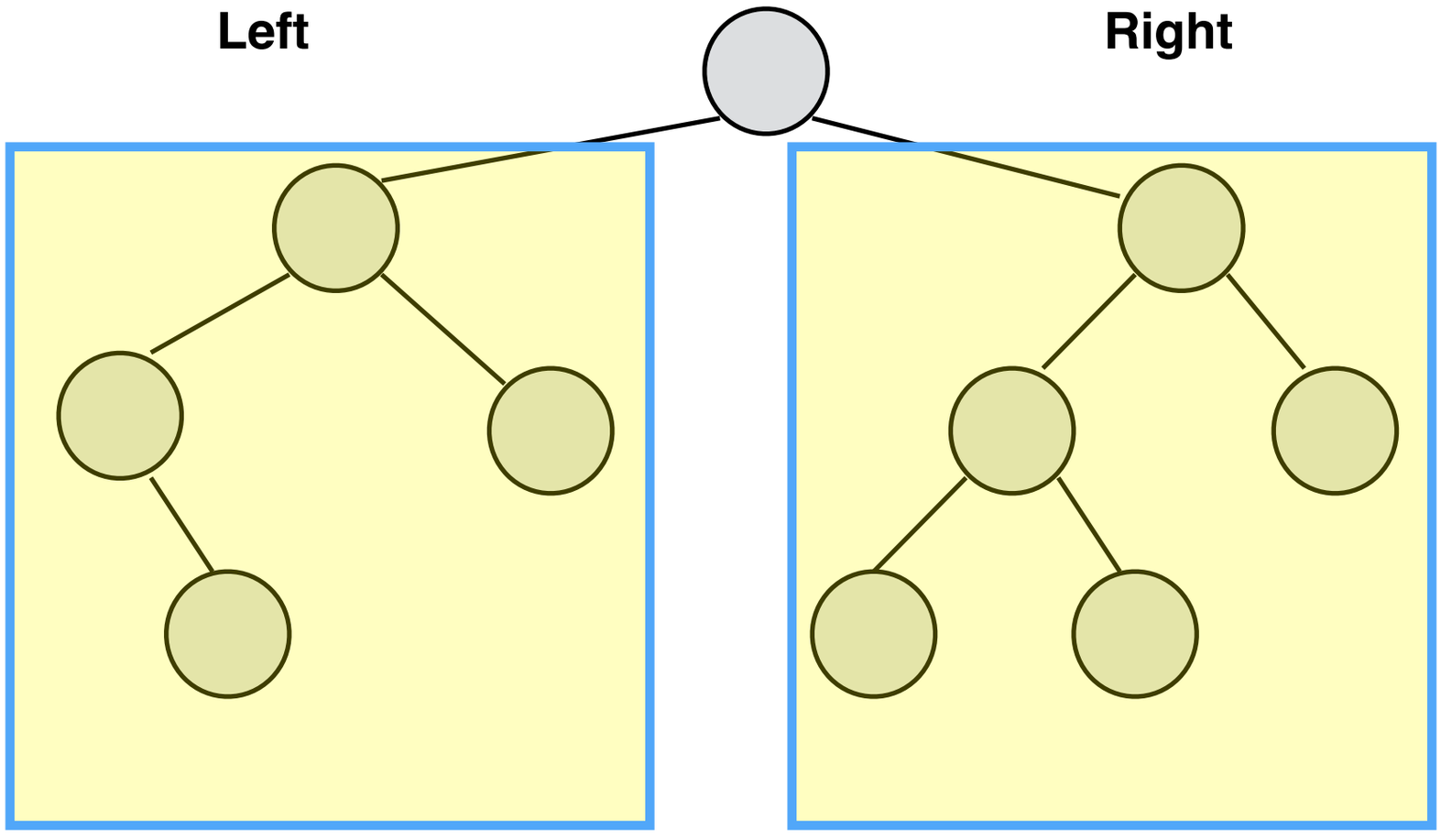
```
2
5
3
```

In BinaryTree:

```
def __iter__(self):
    return PreOrderIteratorStack(self.root)
```

Computing the size of a tree

Returns the **number of nodes in the tree** (without modifying the tree)



10

size(self) = size(left) + 1 + size(right)

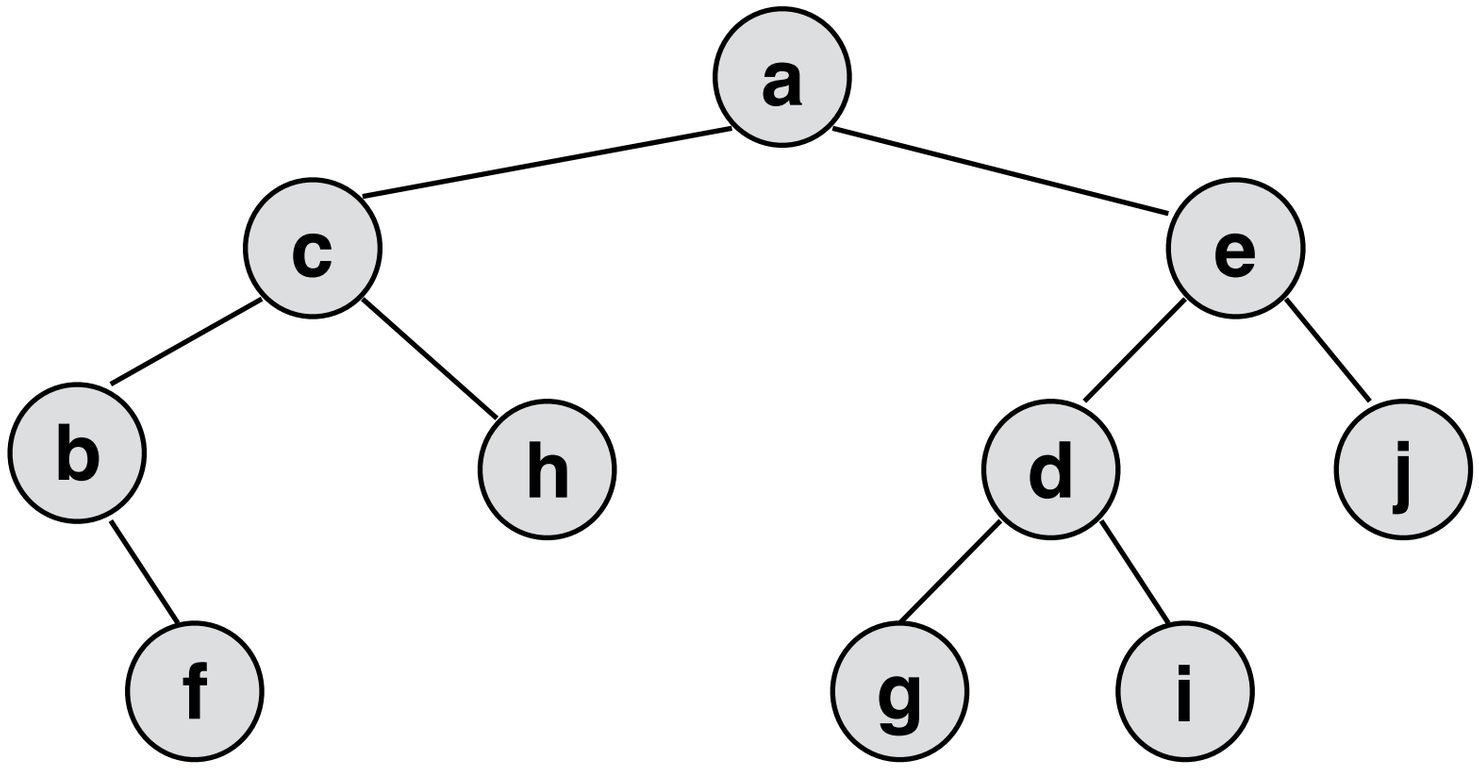
Computing the size of a tree

```
def __len__(self):
    return self.len_aux(self.root)

def len_aux(self, current):
    if current is None:
        return 0
    else:
        return 1 + self.len_aux(current.left) + self.len_aux(current.right)
```

Collecting the leaves of a tree

Returns the **a list of the leaves** (left to right)



[f, h, g, i, j]

traverse, when finding a leaf (no children) add to **list**…

[pass the **list** as an accumulator]

Collecting the leaves of a tree

```
def get_leaves(self):
    a_list = []
    self.get_leaves_aux(self.root, a_list)
    return a_list


def get_leaves_aux(self, current, a_list):
    if current is not None:
        if self.is_leaf(current):
            a_list.append(current.item)
        else:
            self.get_leaves_aux(current.left, a_list)
            self.get_leaves_aux(current.right, a_list)

def is_leaf(self, current):
    return current.left is None and current.right is None
```

```
>>> from lecture_31 import BinaryTree
>>> my_tree = BinaryTree()
>>> my_tree.add(1, '')
>>> my_tree.add(2, '1')
>>> my_tree.add(3, '0')
>>>
>>> my_tree.get_leaves()
[3, 2]
>>> my_tree.add(4, '01')
>>> my_tree.get_leaves()
[4, 2]
>>>
```

Summary

- **Tree traversal**: inorder, postorder, preorder