Under week 10

CHAPTER EIGHTEEN

Python's Dictionary Implementation: Being All Things to All People

Andrew Kuchling

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

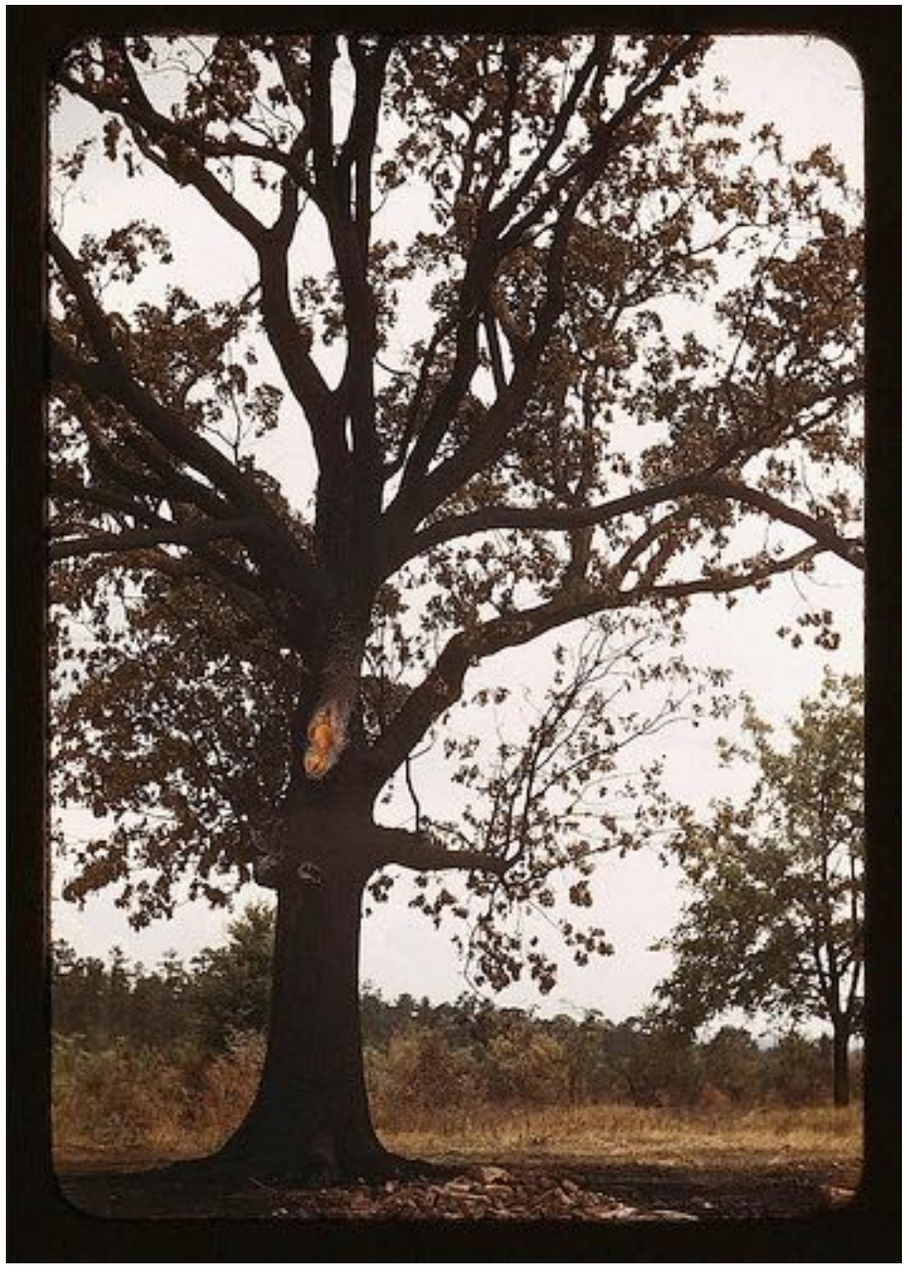
Lecture 32
Binary Trees

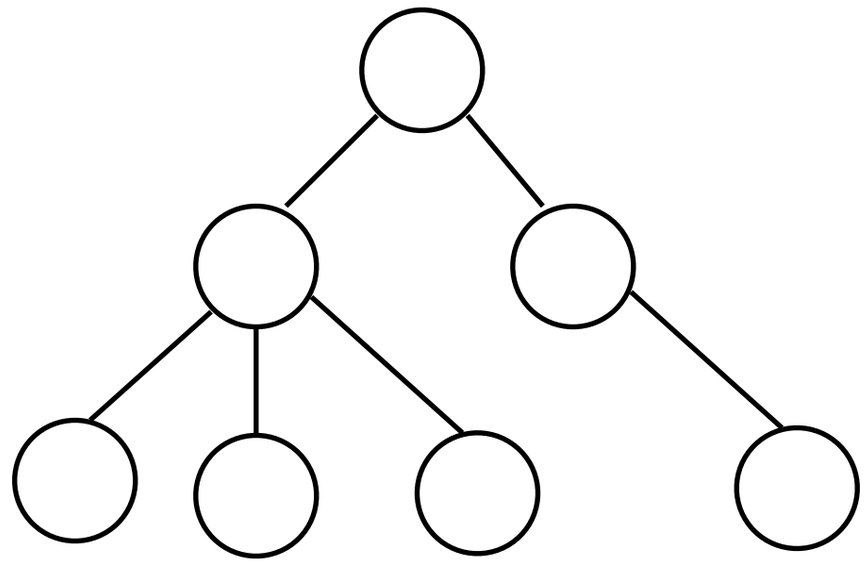
FIT 1008
Introduction to Computer Science

Objectives

- Revise **Trees**:
  - ➡ Concepts
  - ➡ Operations & Implementation
  - ➡ Complexity Ideas
  - ➡ Traversal

Trees

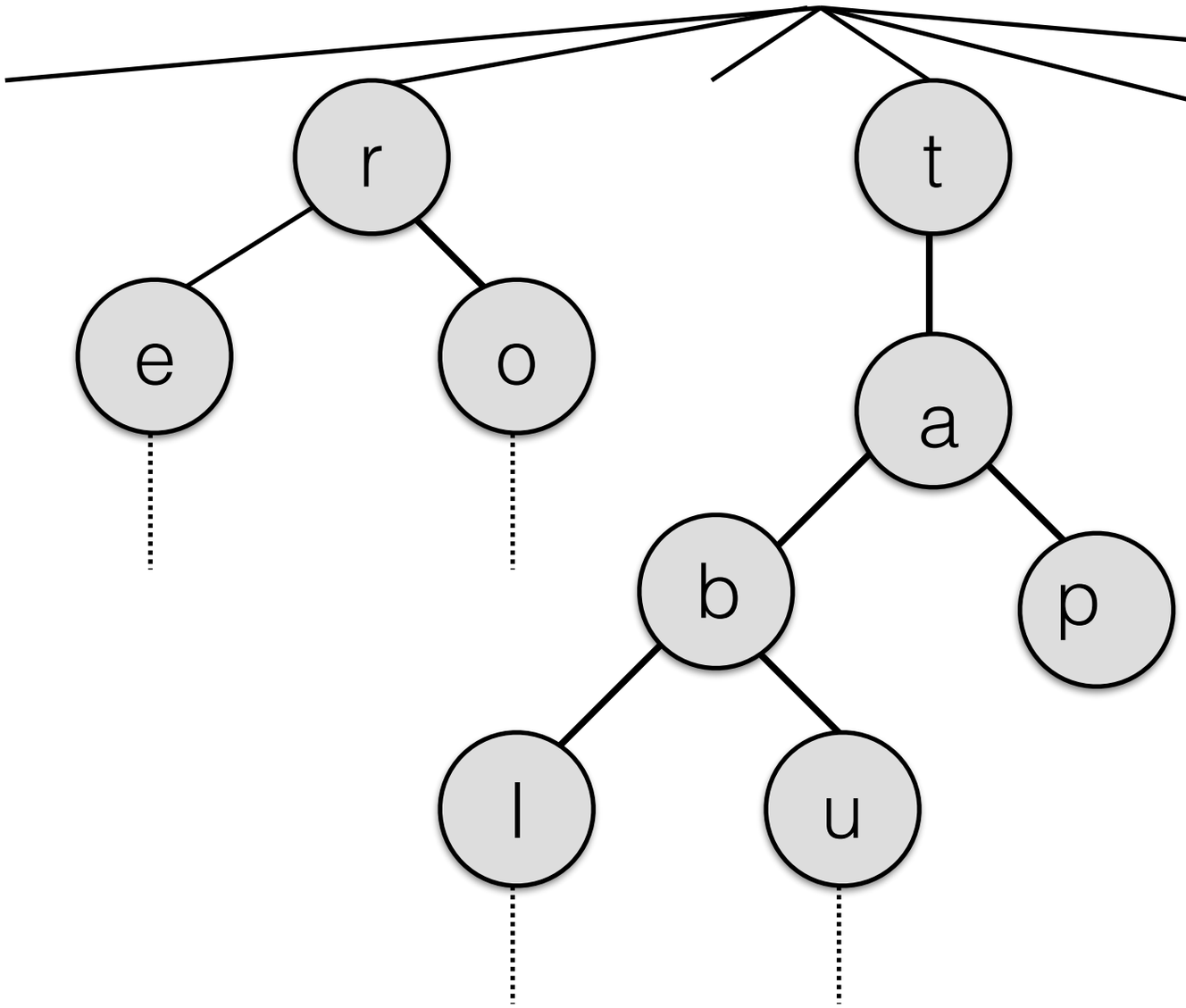




Trees

- Extremely useful.
- Natural way of modelling many things:
  - ➡ Family trees
  - ➡ Organisation structure charts
  - ➡ Structure of chapters and sections in a book
  - ➡ Execution/call tree (recall the one for fibonacci)
  - ➡ Object Oriented Class Hierarchies
- Particularly good for some operations (like search)
- Compact representation of data

Compact representation of data

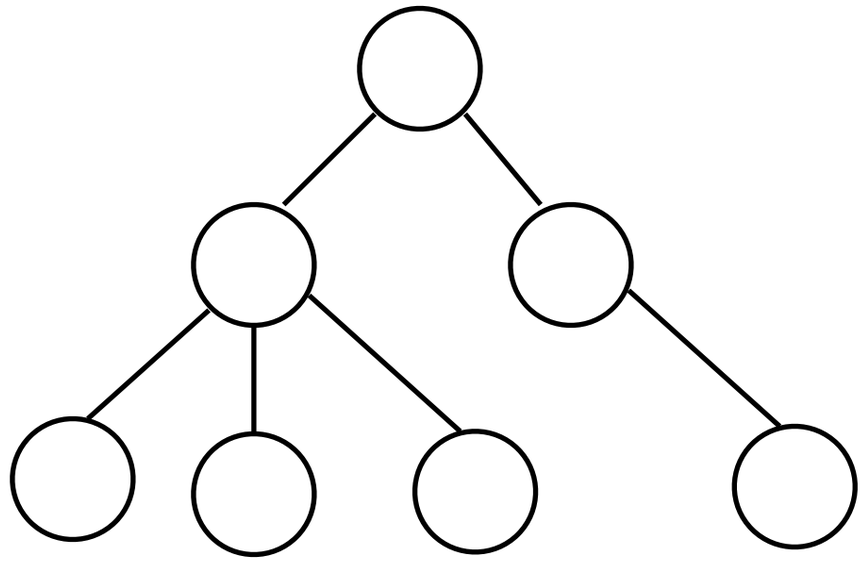


Represent a
list of strings

- table
- tap
- tabular
- rest
- restaurant
- remain
- route

Branches represent different strings.

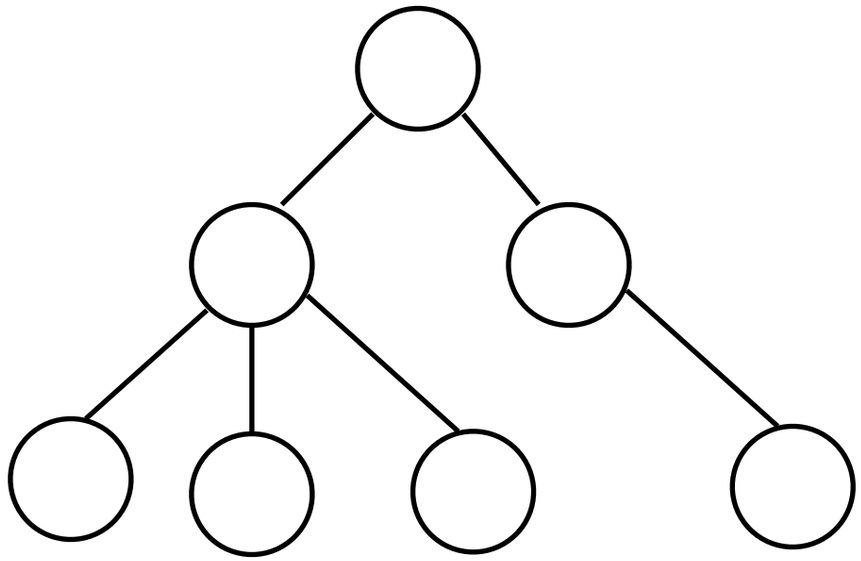
Trees



Trees

- Graphs which are:
  - ➡ Simple

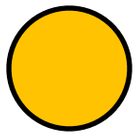
    no loops or multiple edges
  - ➡Connected
  - ➡No circuits.



Pre-reading:

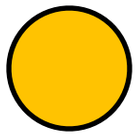
Binary Trees.

Perfect Binary Trees

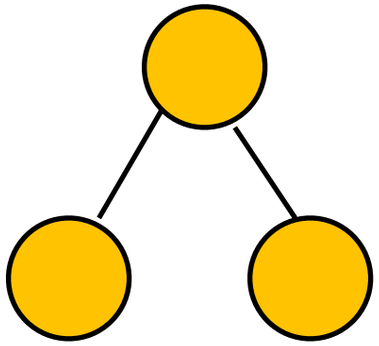


N =1 Height = 0

Perfect Binary Trees

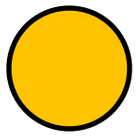


N =1 Height = 0

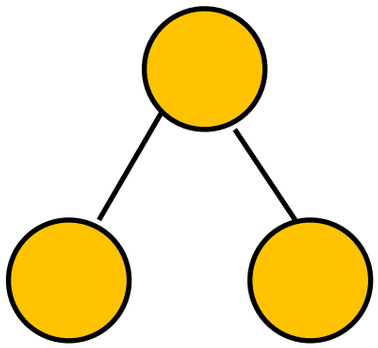


N =3 Height = 1

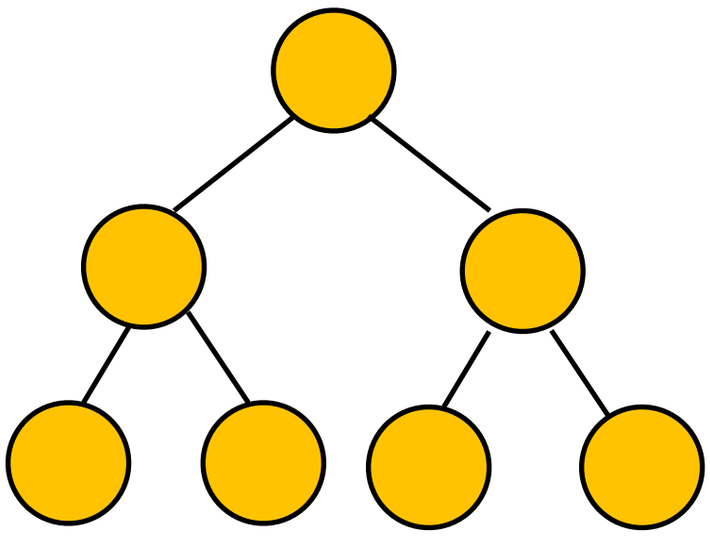
Perfect Binary Trees



N =1 Height = 0

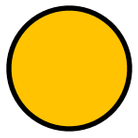


N =3 Height = 1

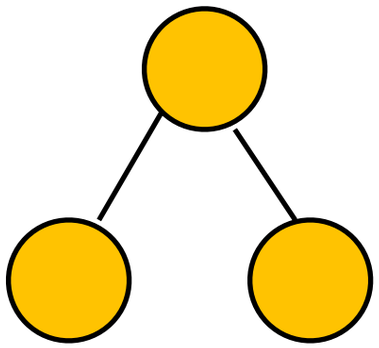


N =7 Height = 2

Perfect Binary Trees

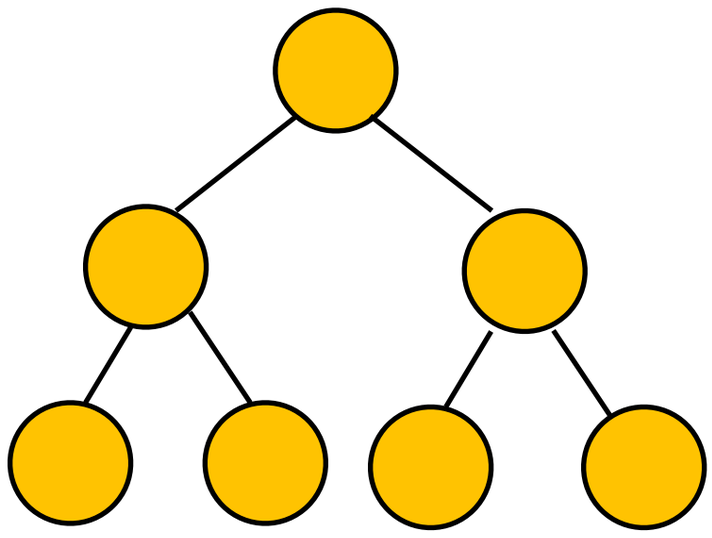


N =1 Height = 0



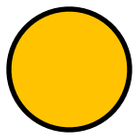
N =3 Height = 1

Each parent has two children

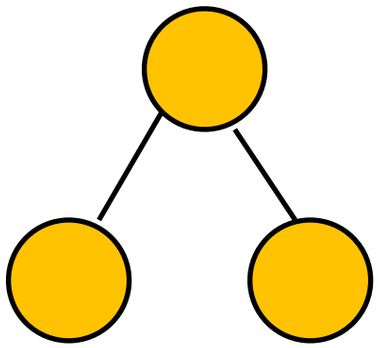


N =7 Height = 2

Perfect Binary Trees



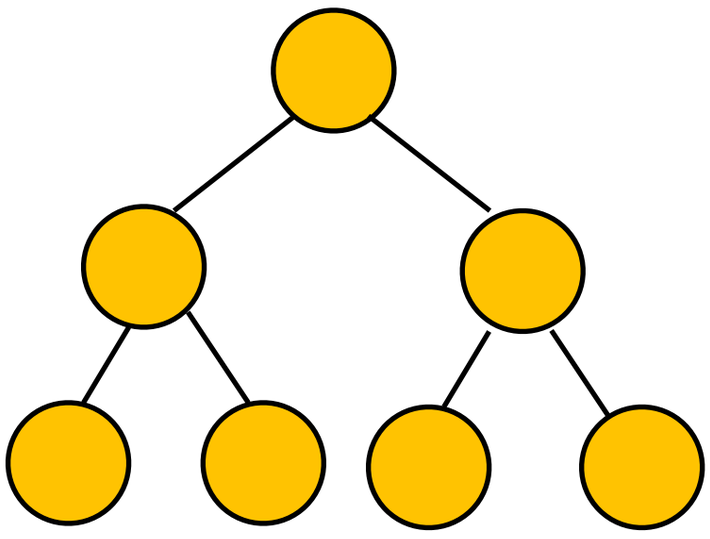
N =1 Height = 0



N =3 Height = 1

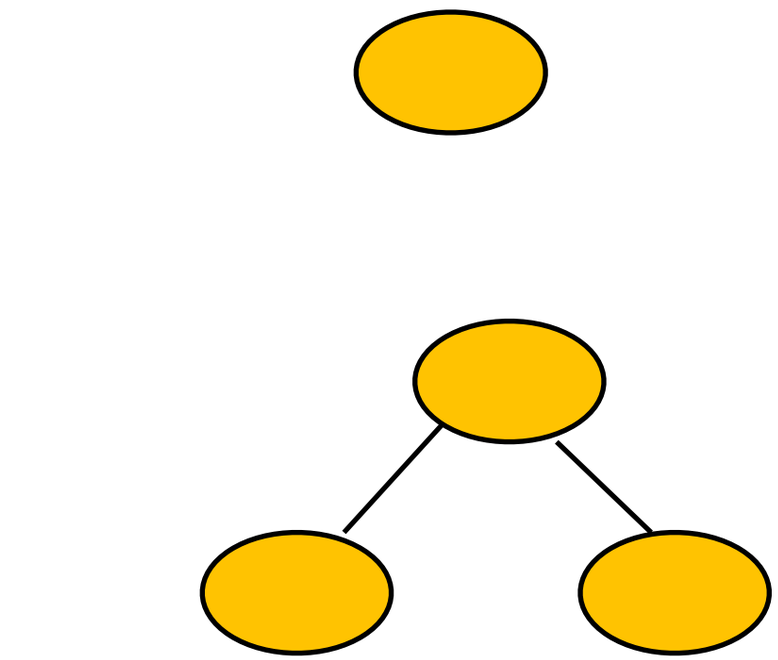
Each parent has two children

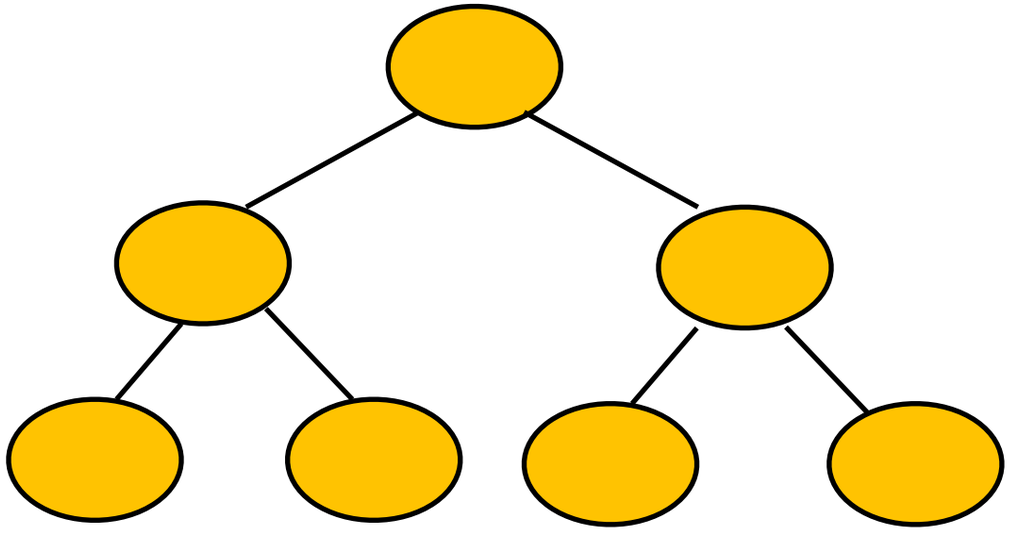
All leaves at same level



N =7 Height = 2

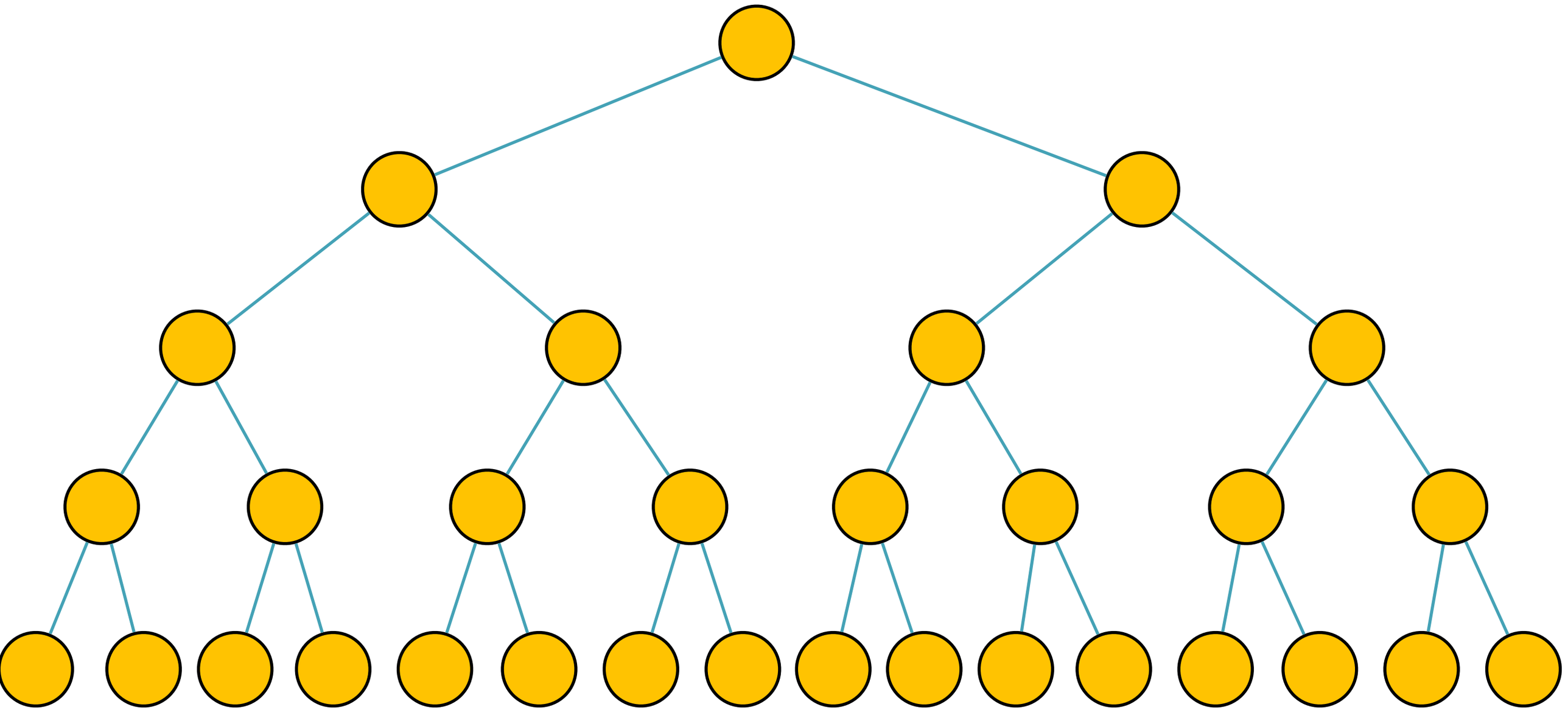
Perfect Binary Trees





height	leaves	nodes
0	1	1
1	2	3
2	4	7
3	8	15
k	**2^k**	**$2^{k+1}-1$**

$N = 2^{k+1}-1$ Height = k



Perfect Binary Trees

$$N = 2^{k+1}-1$$

$$N+1 = 2^{k+1}$$

$$\log_2(N+1) = k+1$$

$$\log_2(N+1)-1 = k$$

Perfect Binary Trees

$$N = 2^{k+1}-1$$

$$N+1 = 2^{k+1}$$

$$\log_2(N+1) = k+1$$

$$\log_2(N+1)-1 = k$$

In a perfect binary tree with N nodes, the height is O(logN)

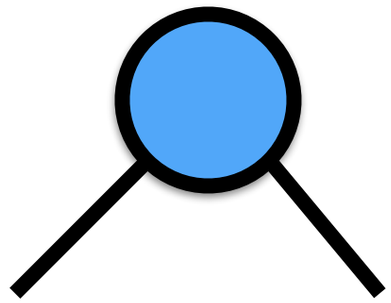
Balanced tree

the height is **O(logN)**

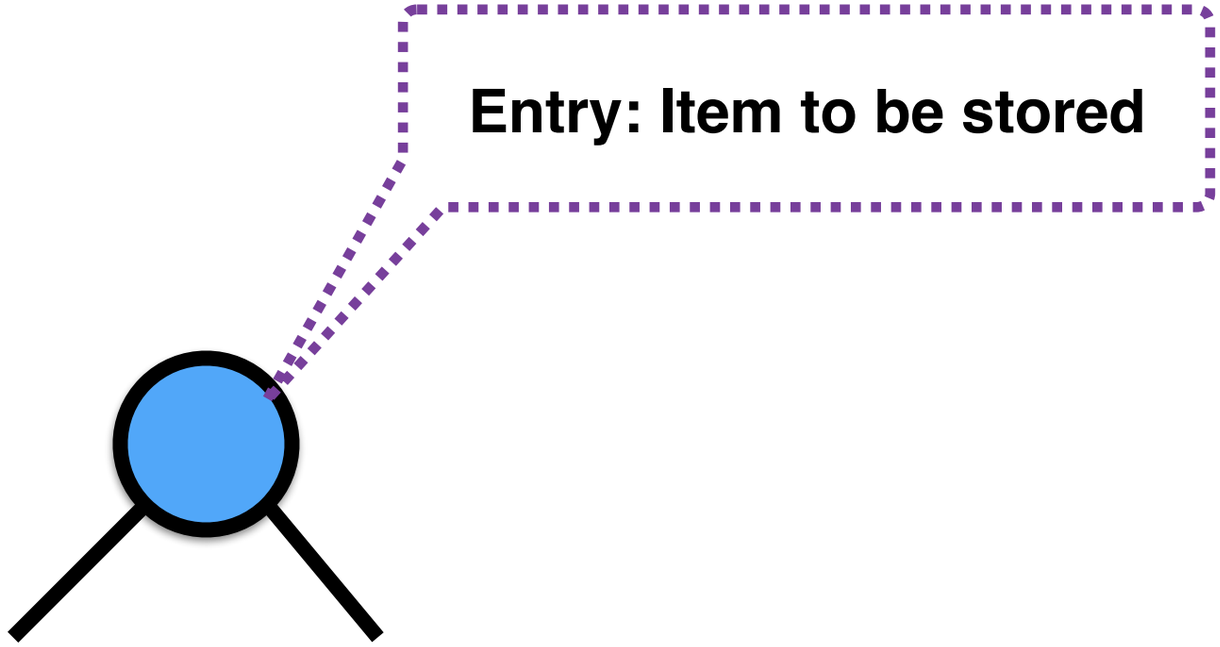
Unbalanced tree

the height is **O(N)**

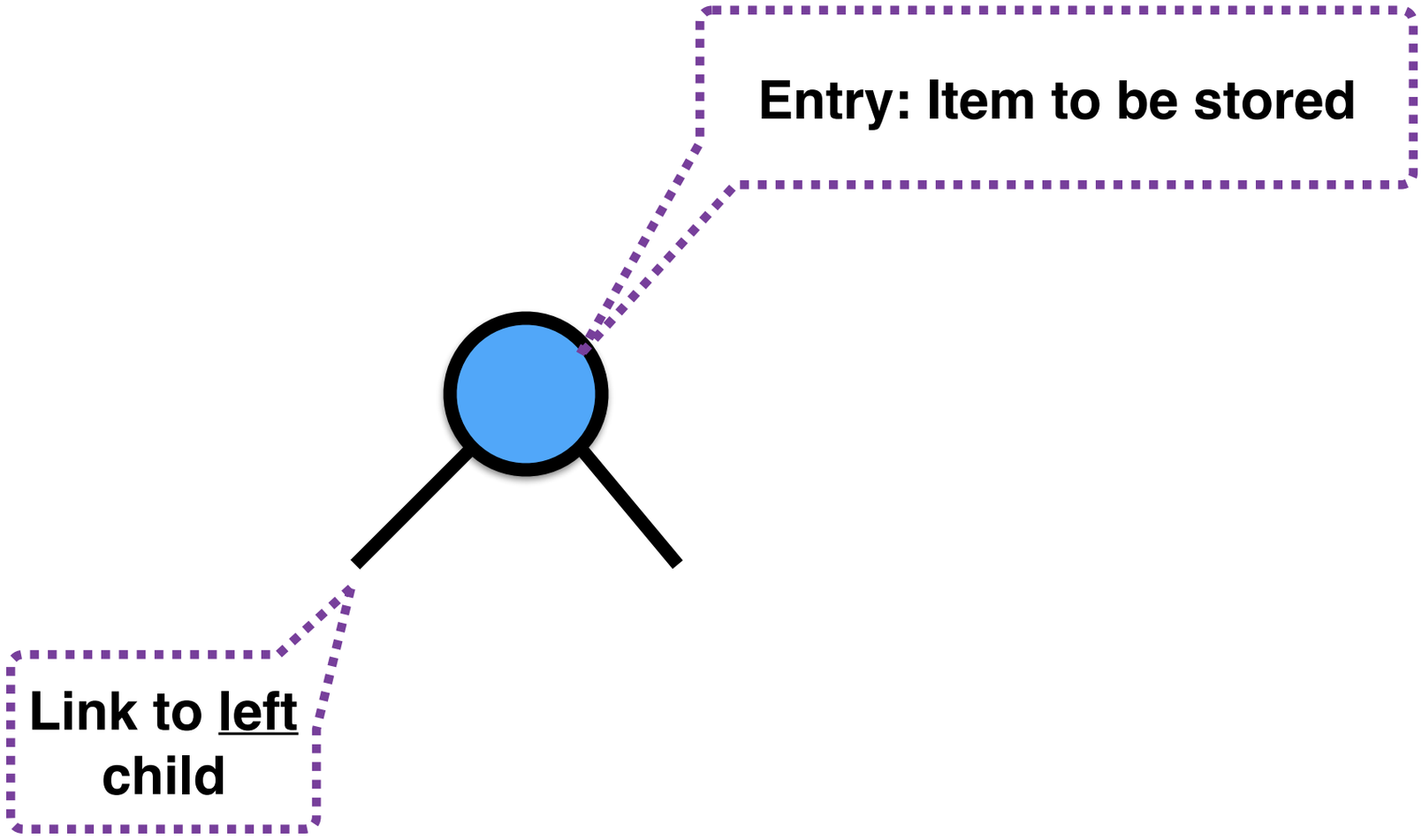
Representing a Binary Tree **Node**



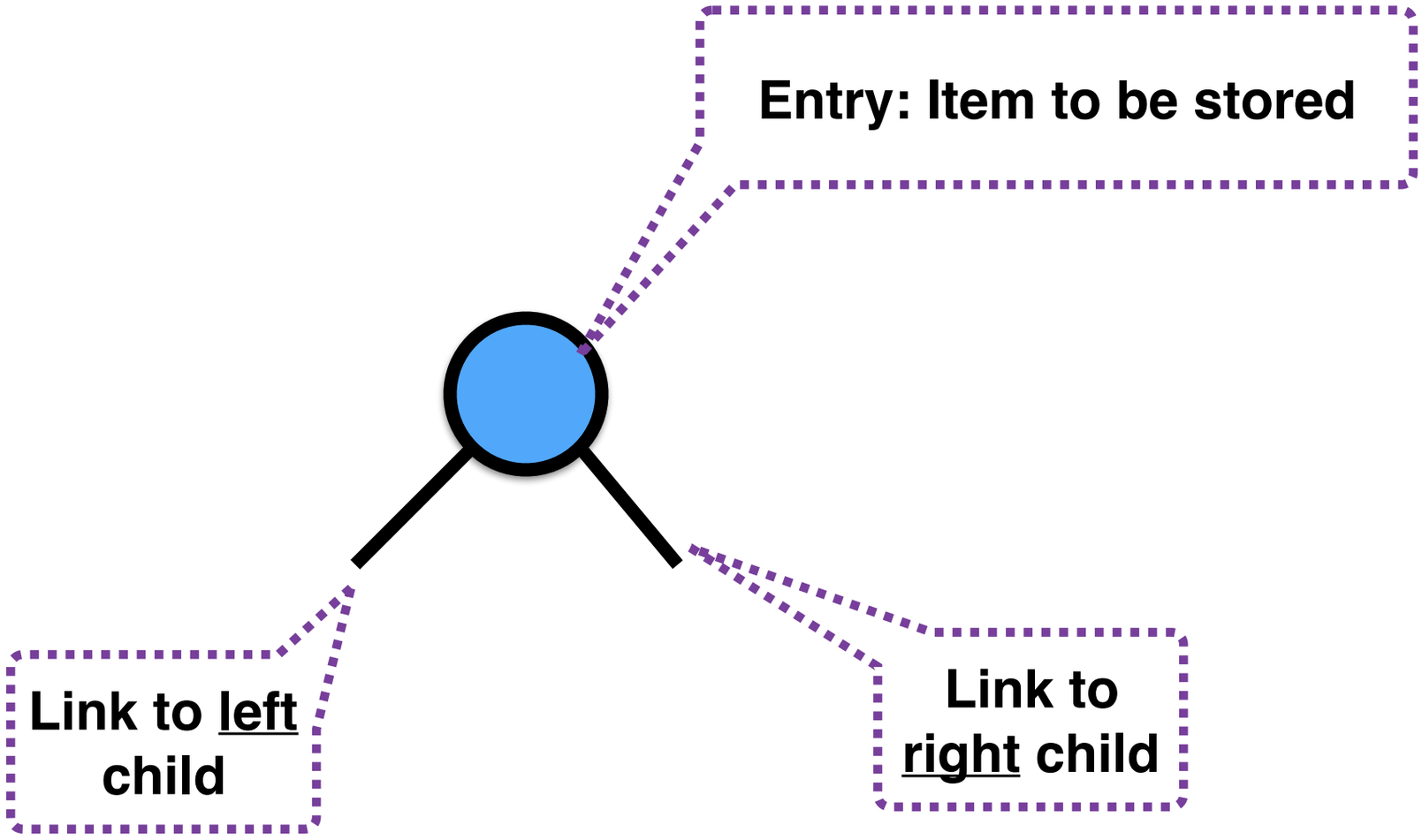
Representing a Binary Tree **Node**



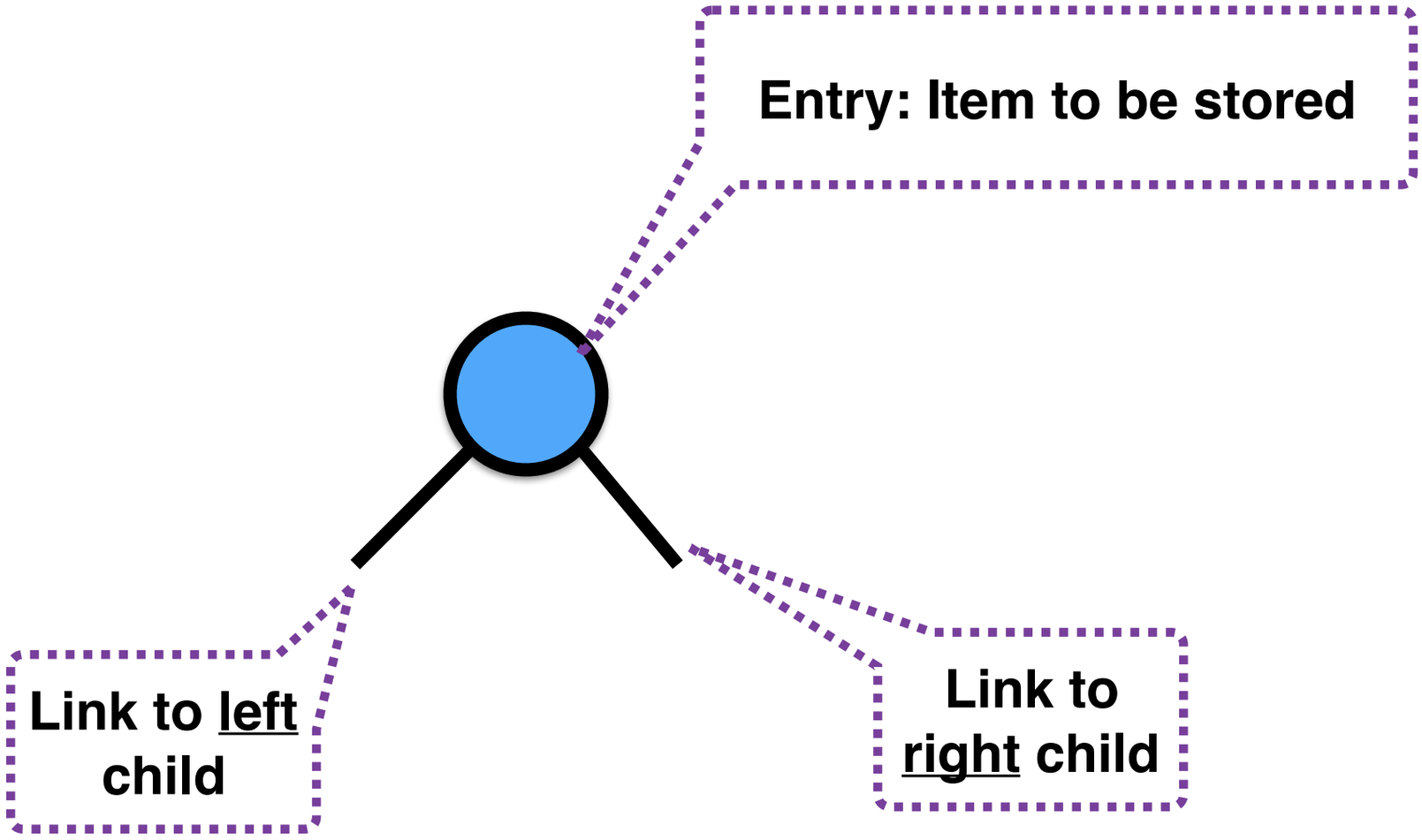
Representing a Binary Tree **Node**



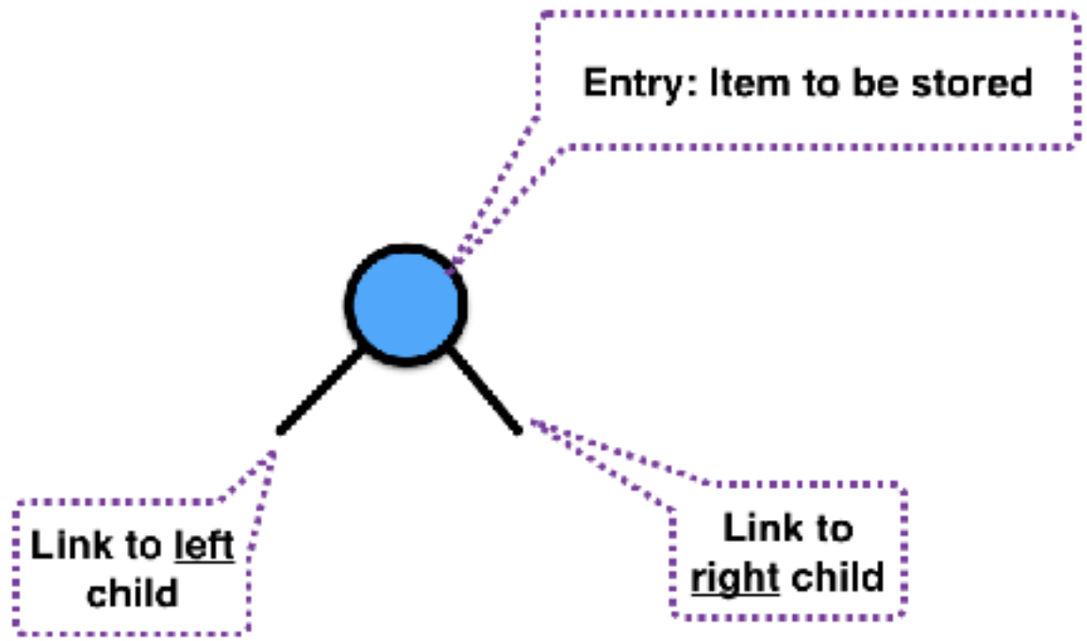
Representing a Binary Tree **Node**



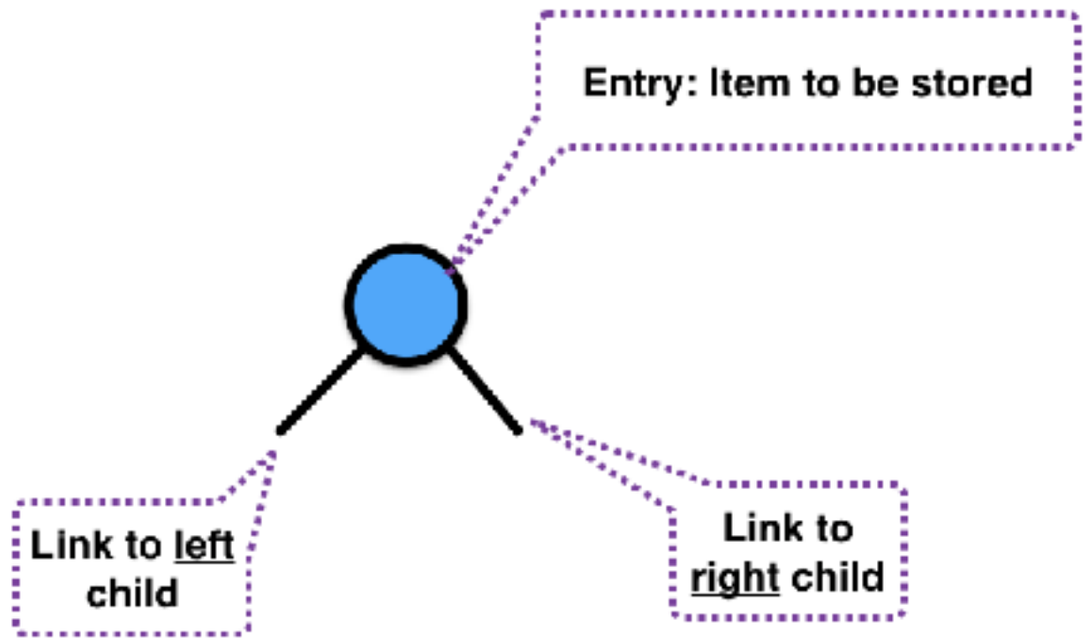
Representing a Binary Tree **Node**



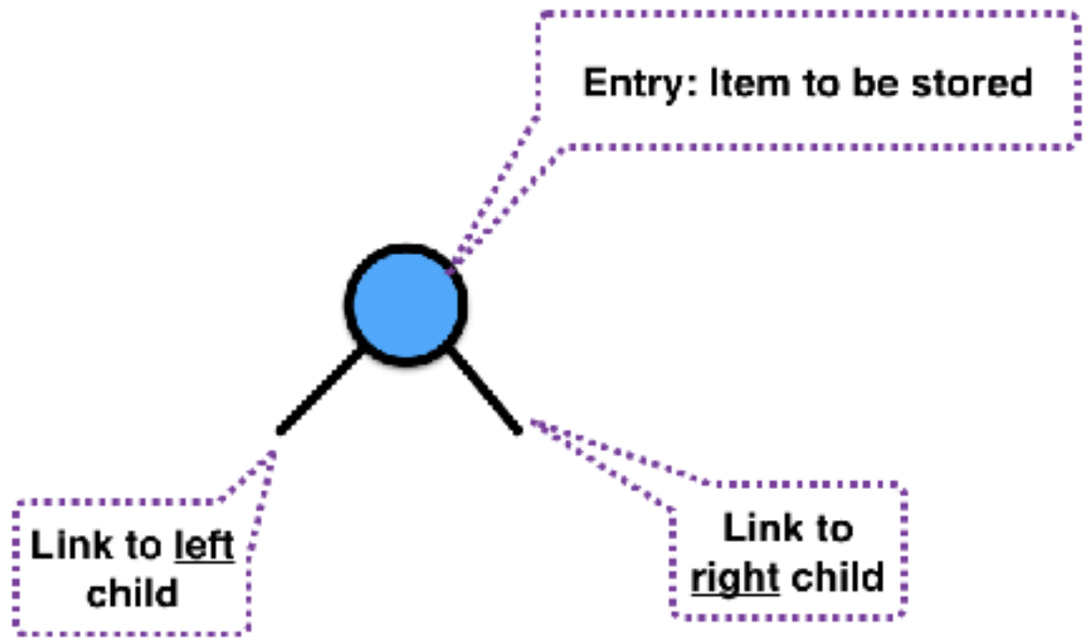
Our implementation: Each link points to a **Node**



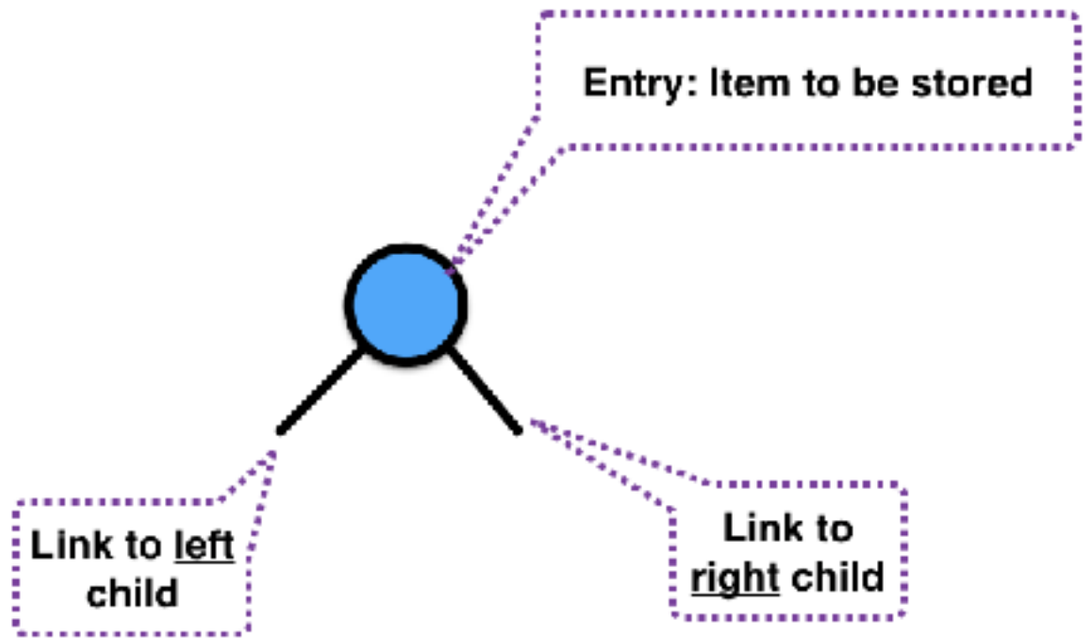

```
class TreeNode:
```




```
class TreeNode:

    def __init__(self,item=None,left=None,right=None):
```




```
class TreeNode:
    def __init__(self,item=None,left=None,right=None):
        self.item = item
        self.left = left
        self.right = right
```




```
class TreeNode:

    def __init__(self,item=None,left=None,right=None):
        self.item = item
        self.left = left
        self.right = right

    def __str__(self):
        return str(self.item)
```

```
class TreeNode:

    def __init__(self,item=None,left=None,right=None):
        self.item = item
        self.left = left
        self.right = right

    def __str__(self):
        return str(self.item)
```

```
class BinaryTree:
```

```
class TreeNode:

    def __init__(self,item=None,left=None,right=None):
        self.item = item
        self.left = left
        self.right = right

    def __str__(self):
        return str(self.item)
```

```
class BinaryTree:
    def __init__(self):
```

```python
class TreeNode:

    def __init__(self,item=None,left=None,right=None):
        self.item = item
        self.left = left
        self.right = right

    def __str__(self):
        return str(self.item)
```

```python
class BinaryTree:
    def __init__(self):
        self.root = None
```

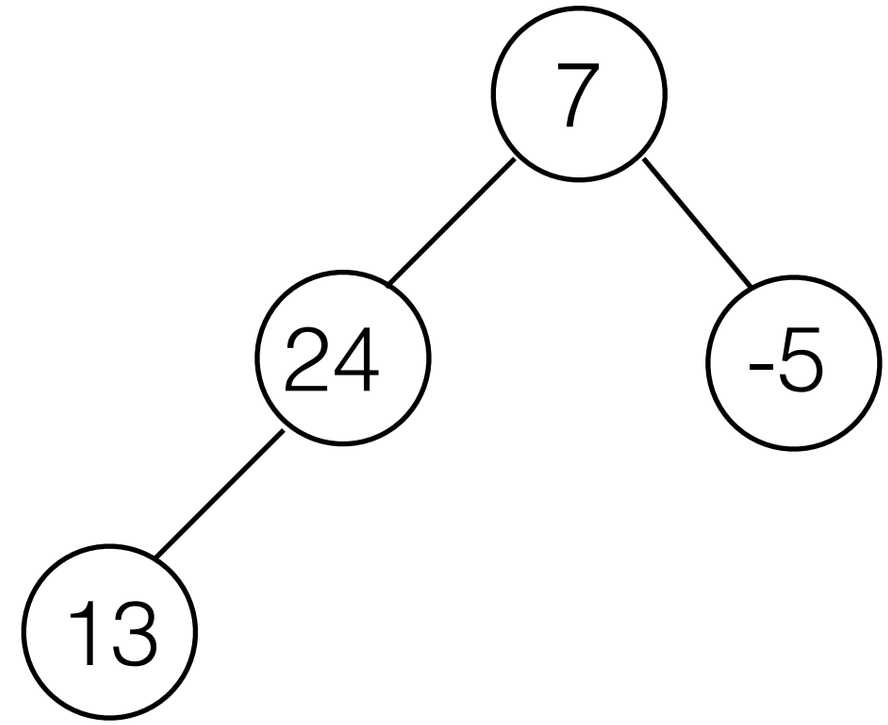
```
class TreeNode:

    def __init__(self,item=None,left=None,right=None):
        self.item = item
        self.left = left
        self.right = right

    def __str__(self):
        return str(self.item)
```

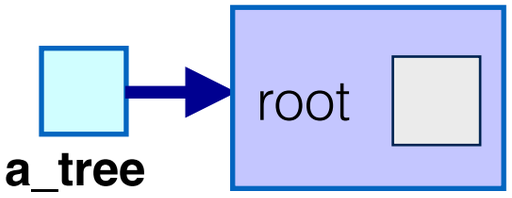
```
class BinaryTree:
    def __init__(self):
        self.root = None

    def is_empty(self):
        return self.root is None
```

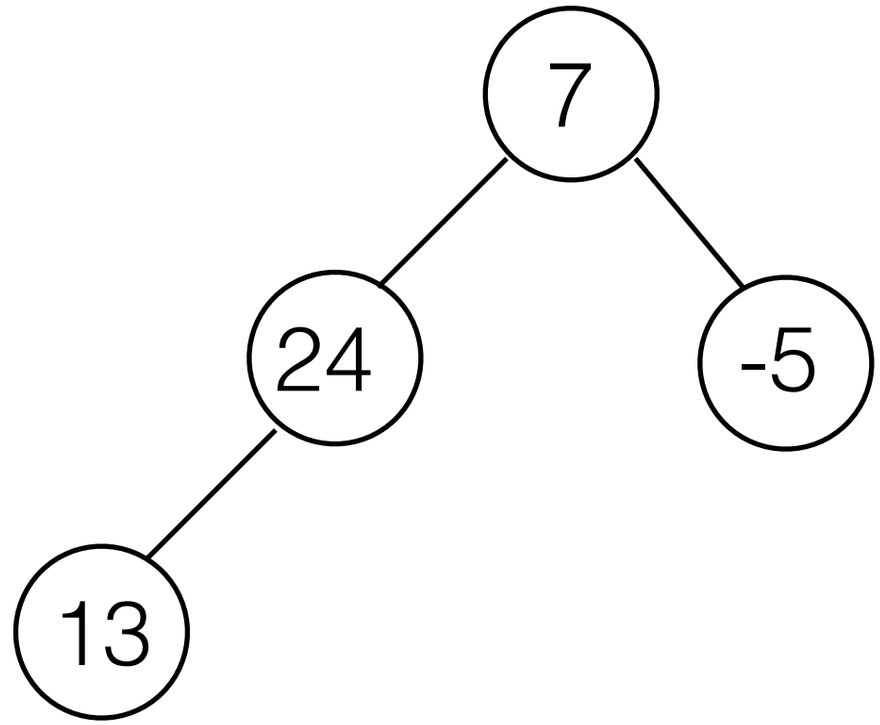
```python
class TreeNode:

    def __init__(self,item=None,left=None,right=None):
        self.item = item
        self.left = left
        self.right = right

    def __str__(self):
        return str(self.item)
```

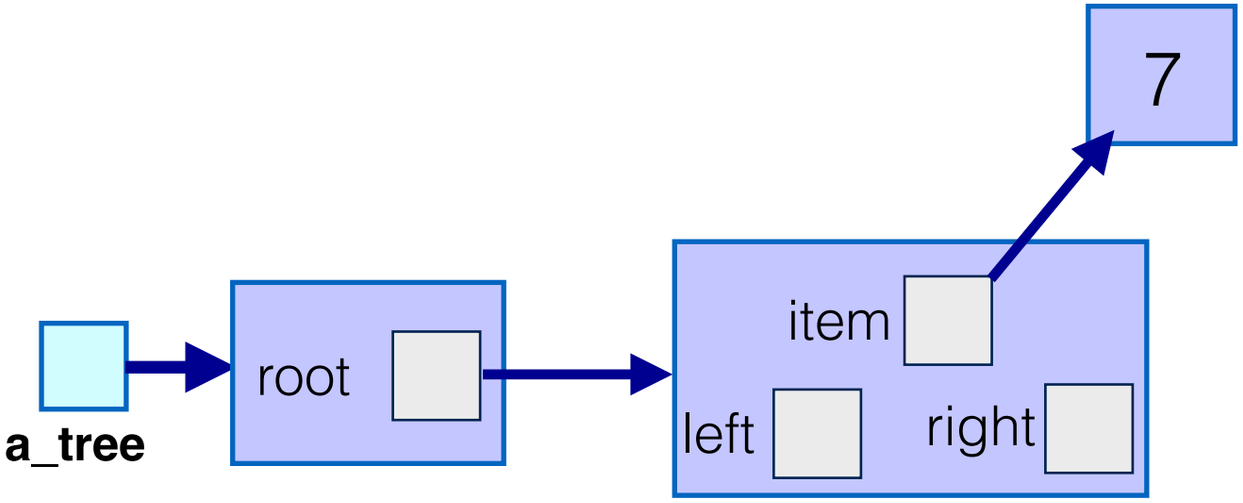
```python
class BinaryTree:
    def __init__(self):
        self.root = None

    def is_empty(self):
        return self.root is None
```

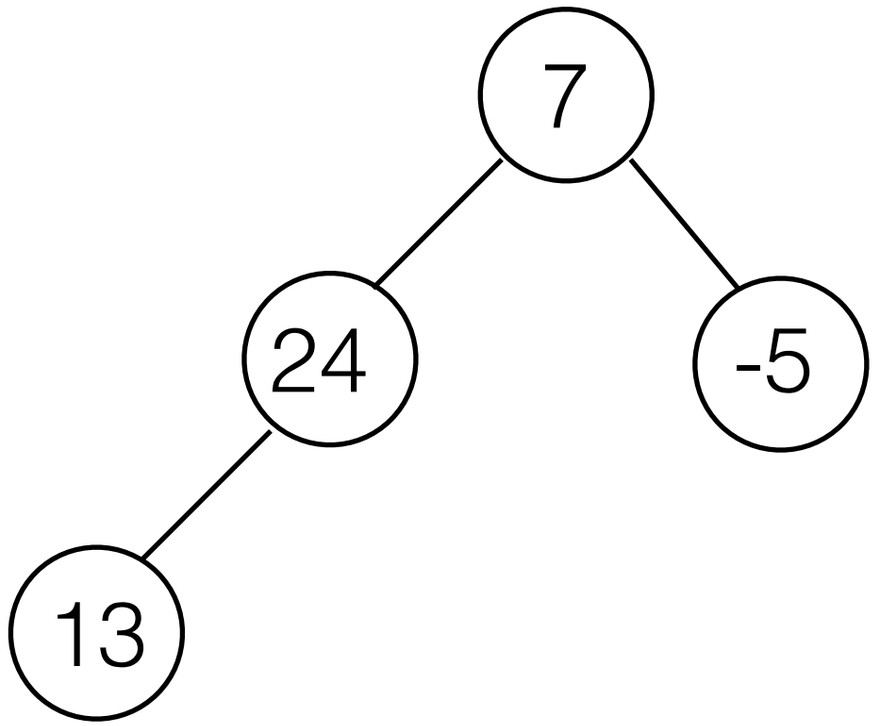
Only instance variable is a reference to the **`root`**

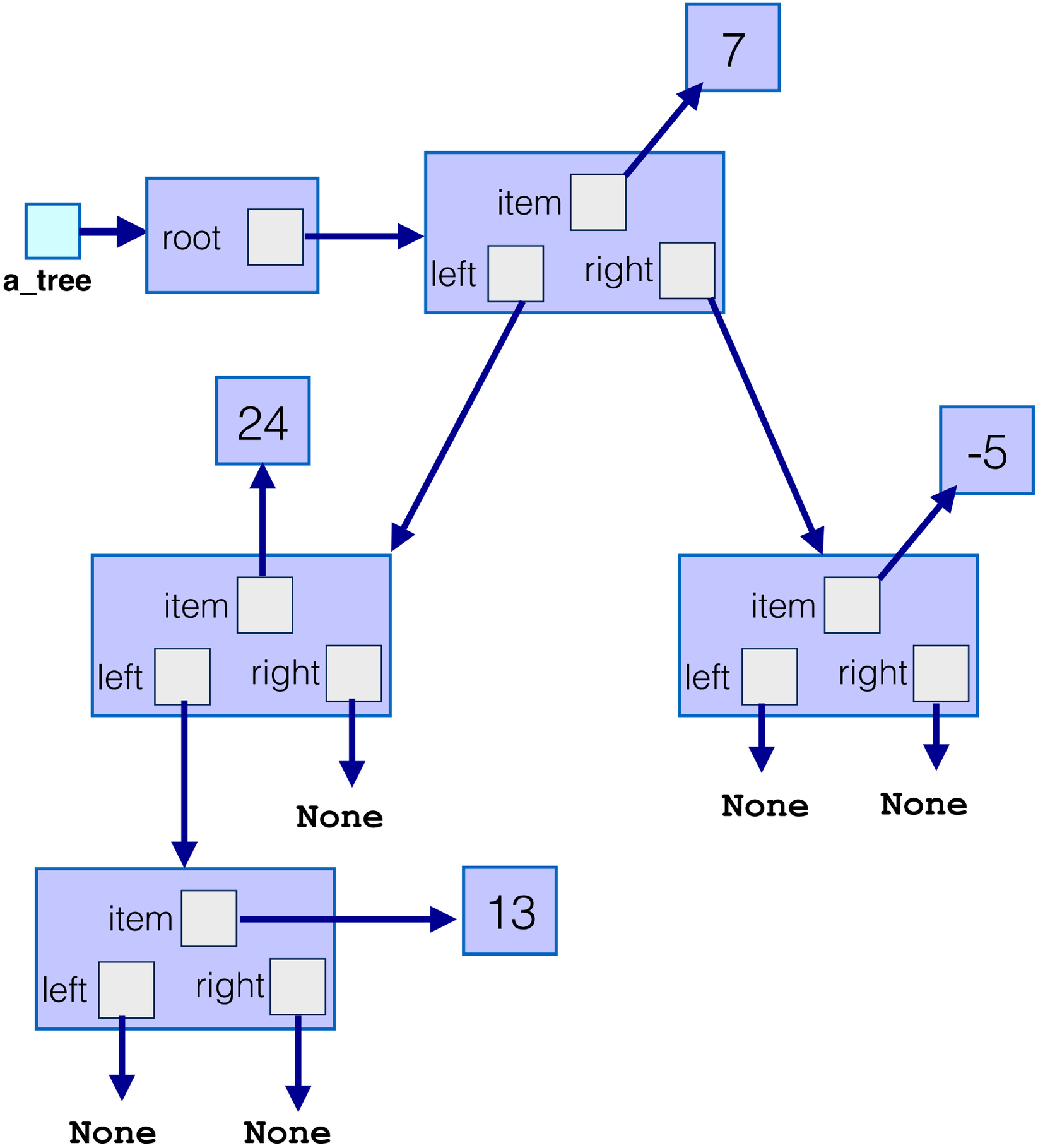

7
24
-5
13

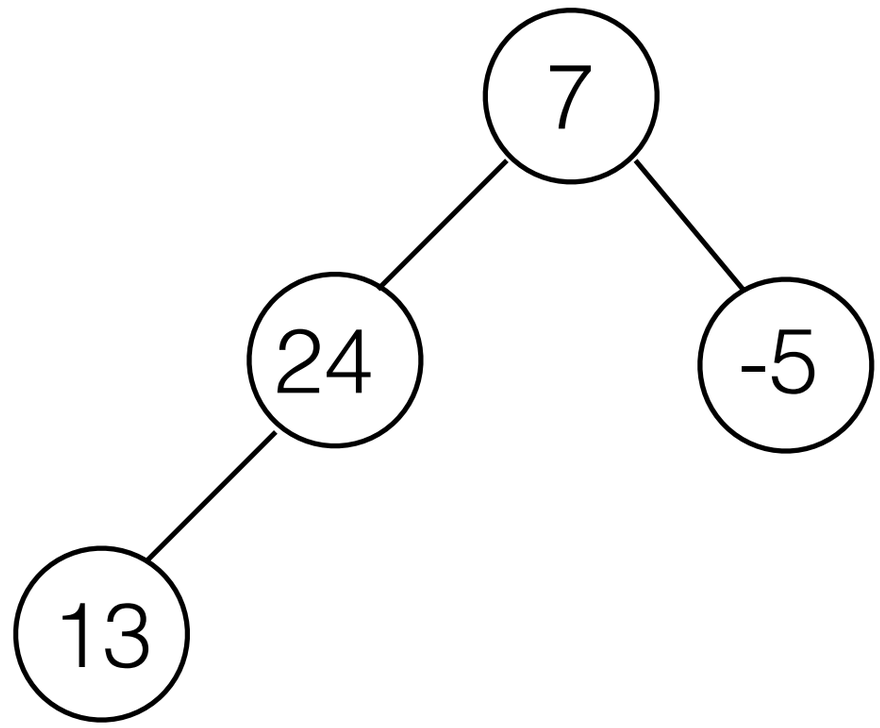

root
a_tree


7
24
-5
13


7
item
root
left
right
a_tree

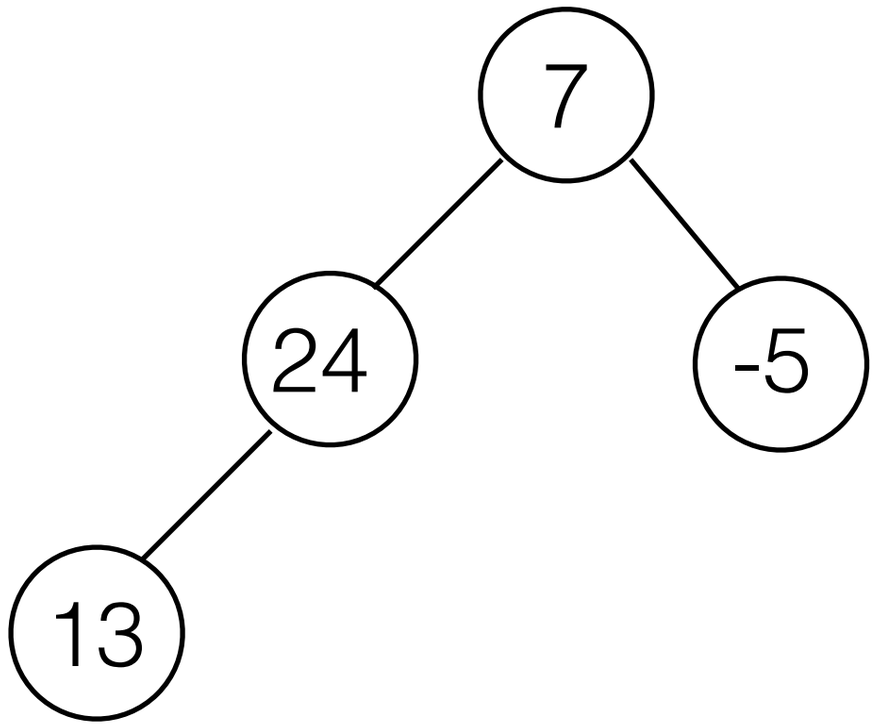

7
24
-5
13


7
item
root
a_tree
left
right
24
-5
item
item
left
right
left
right
None
None
None
None
item
13
left
right
None
None

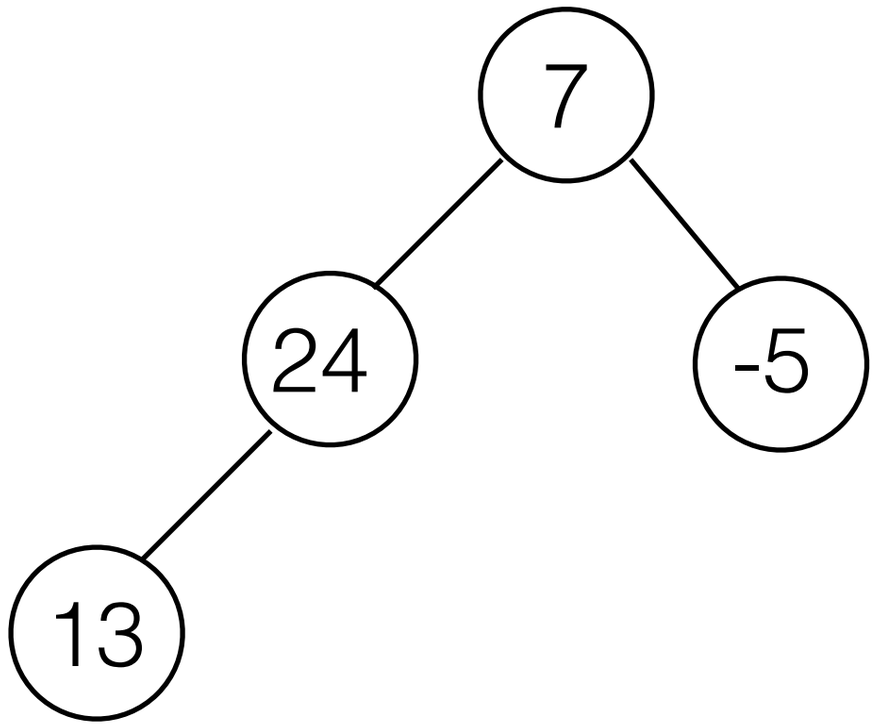

7
24
-5
13

Add an item.

Add 3

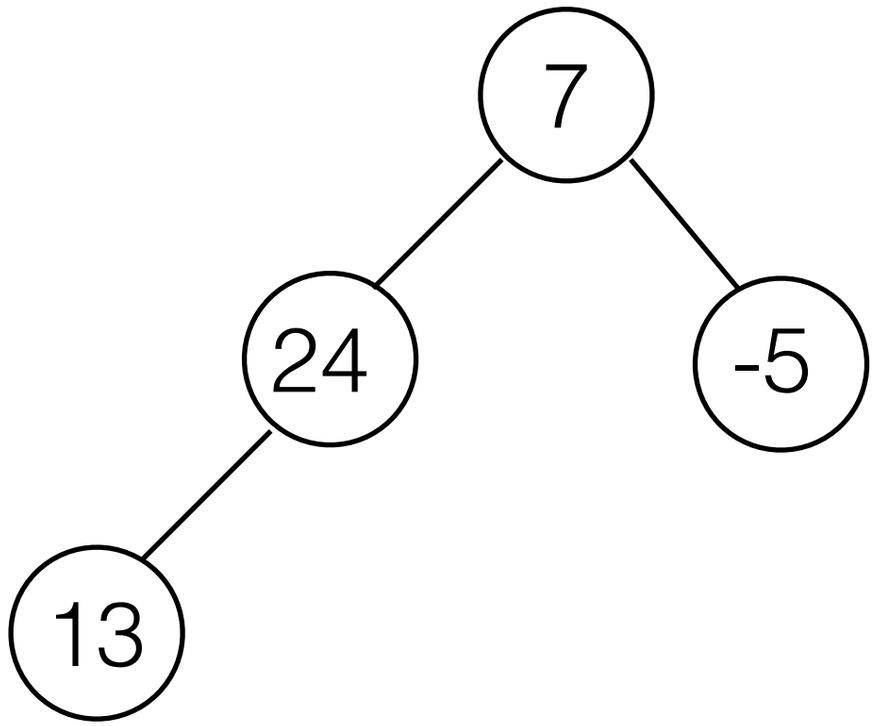


Add 3

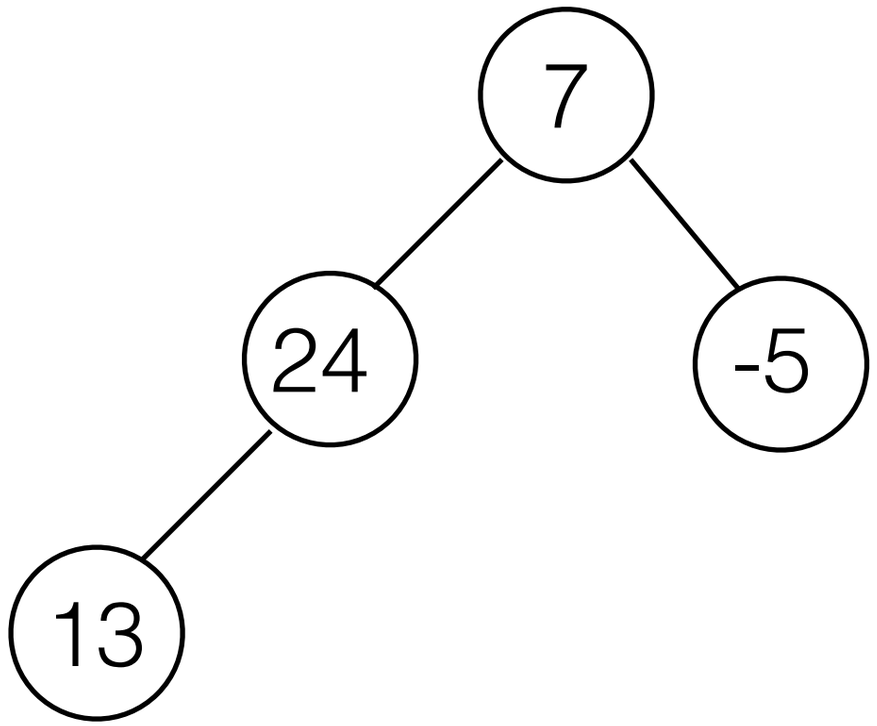


where?

Add 3

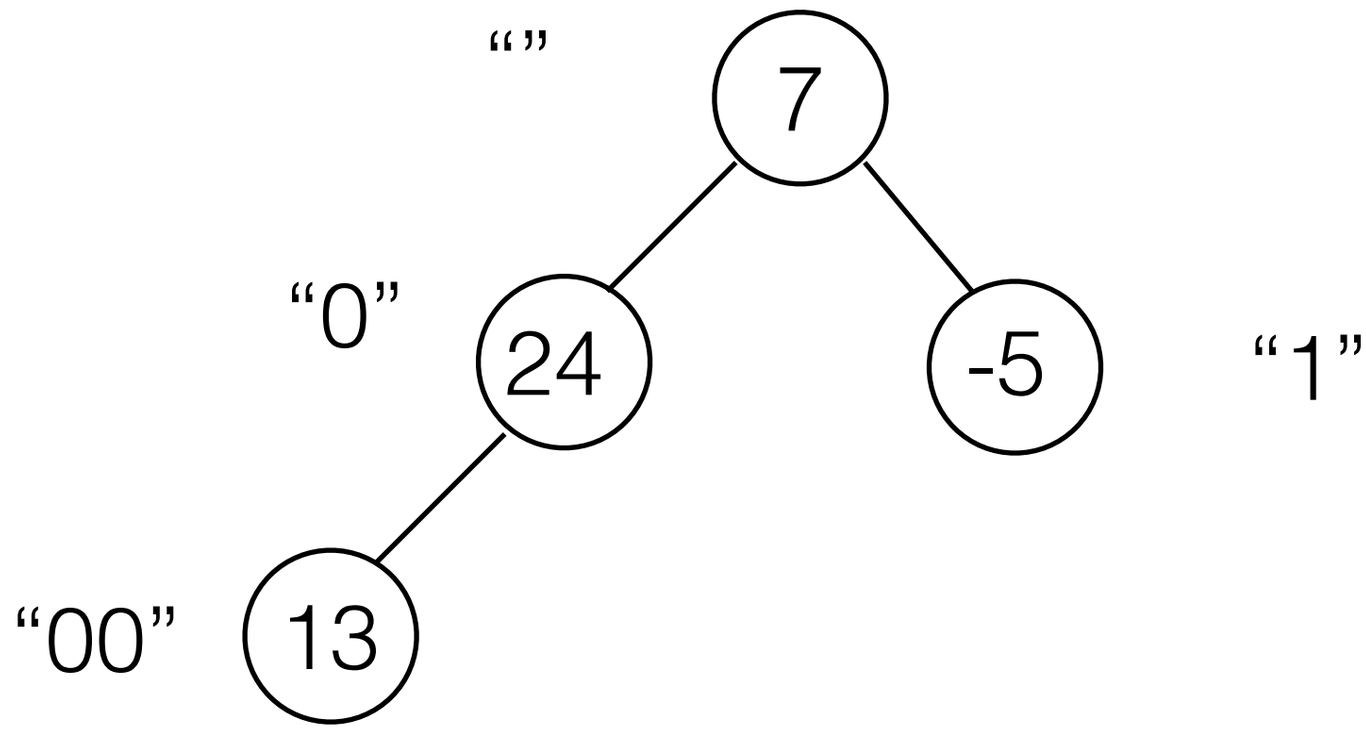


Add 3



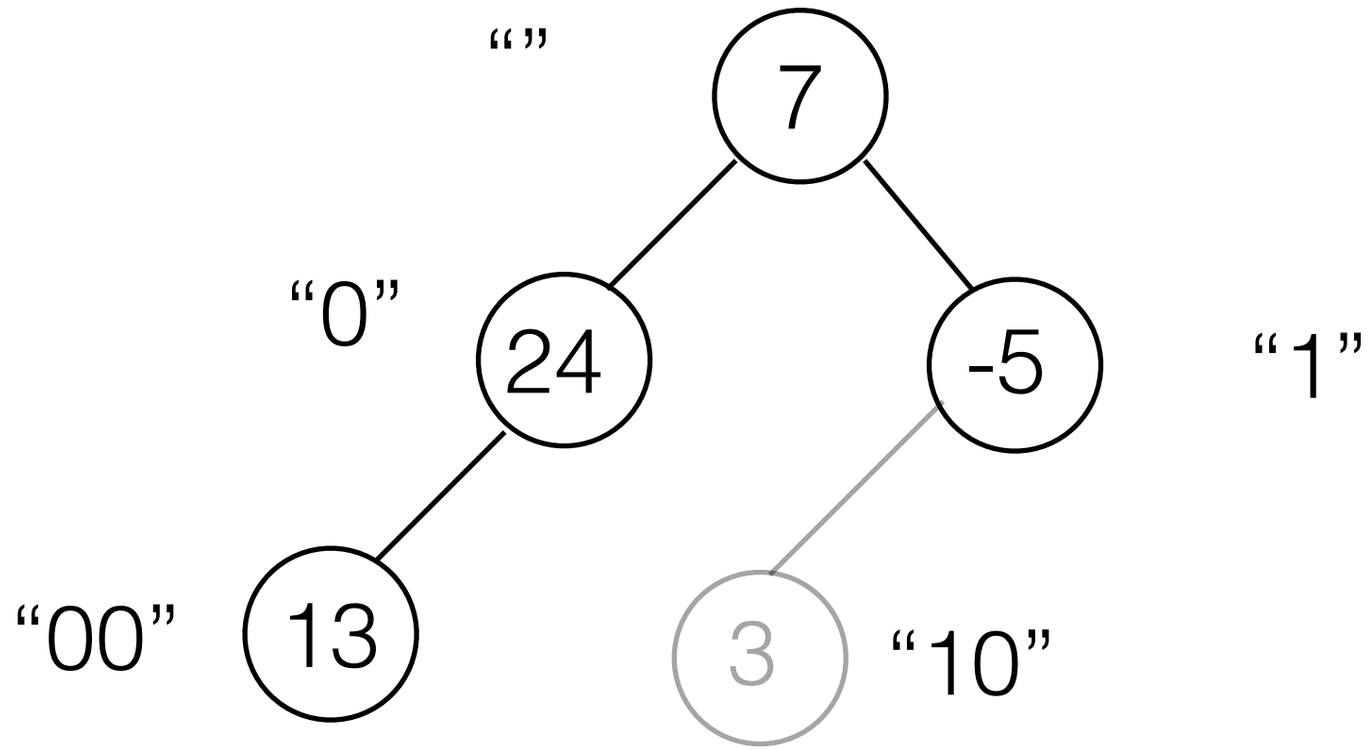
0: Go **left**
1: Go **right**

Add 3



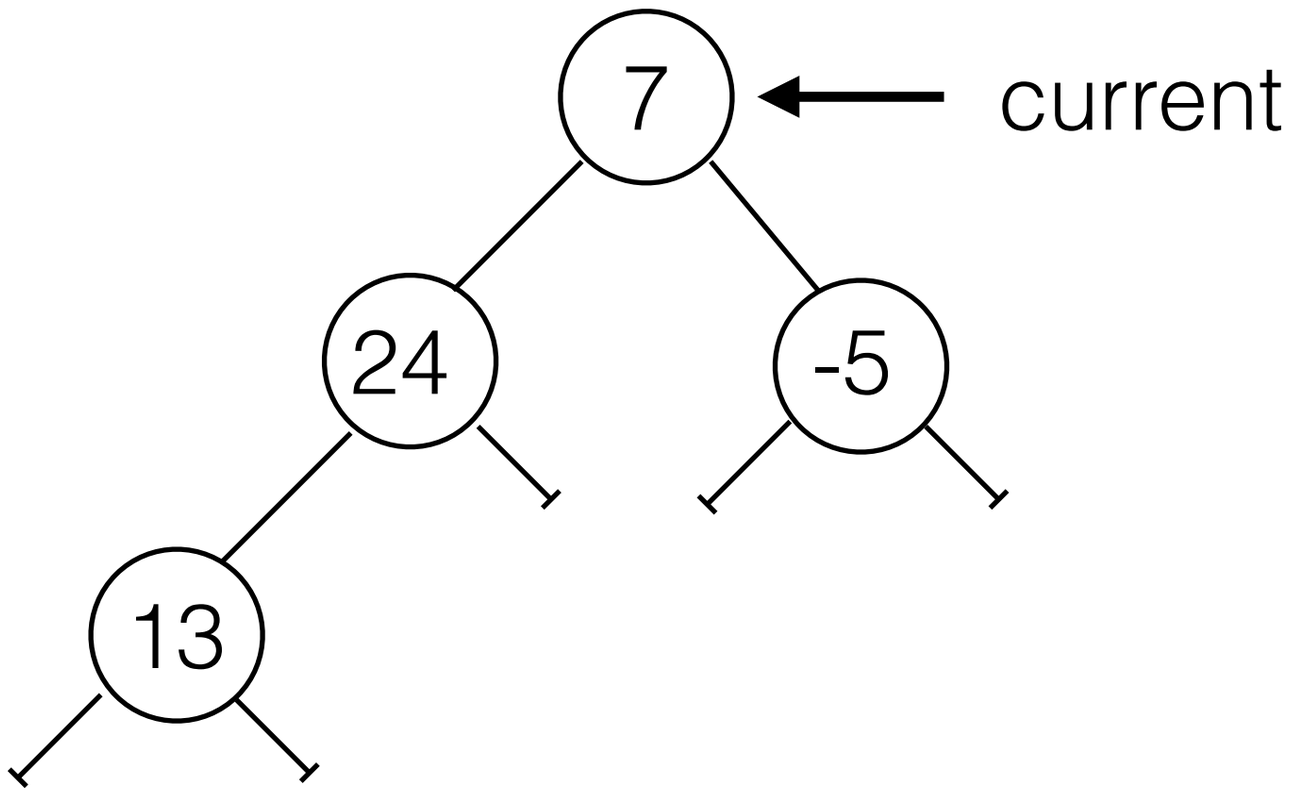
0: Go **left**
1: Go **right**

Add 3



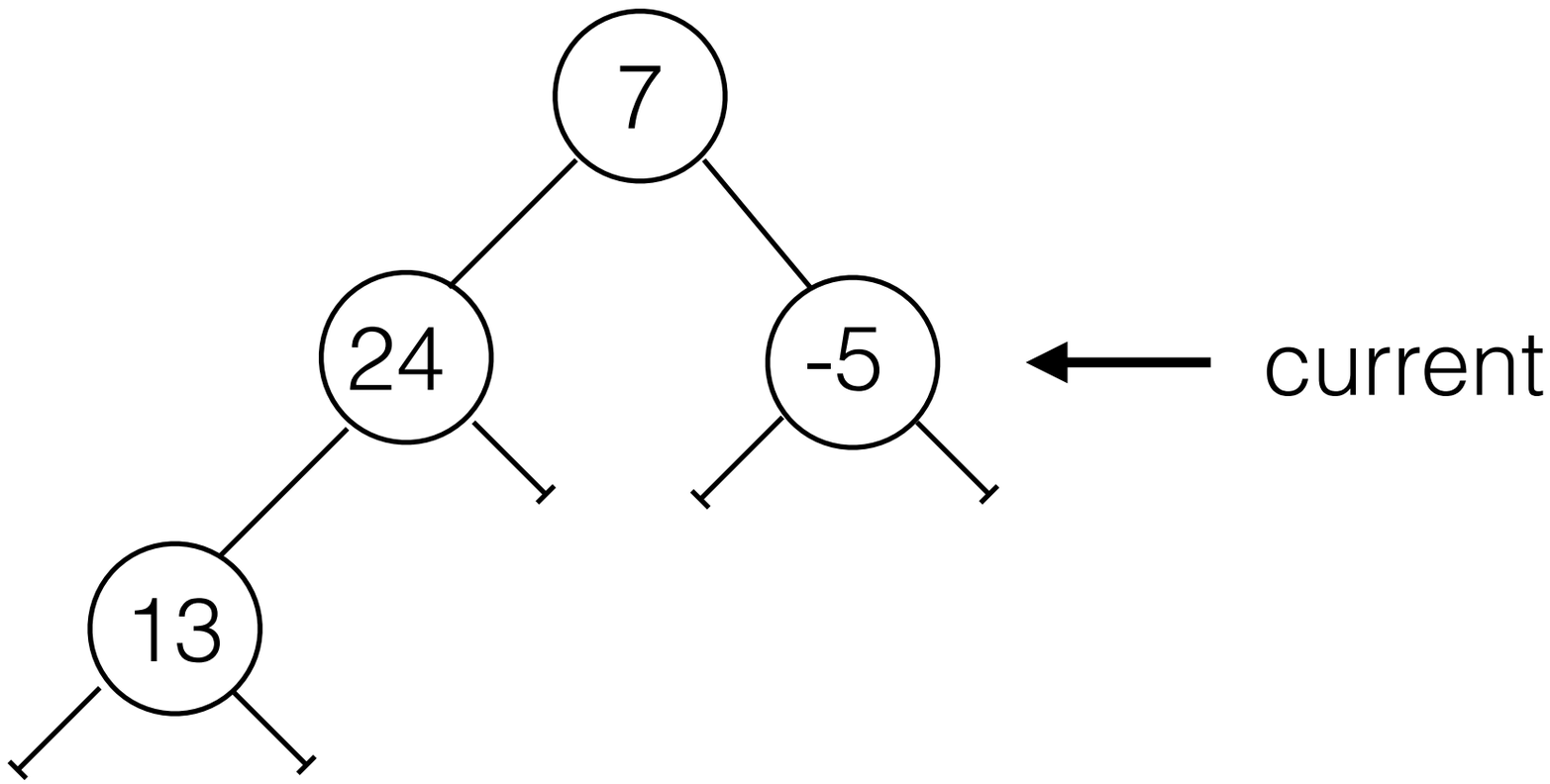
0: Go **left**
1: Go **right**

Add 3



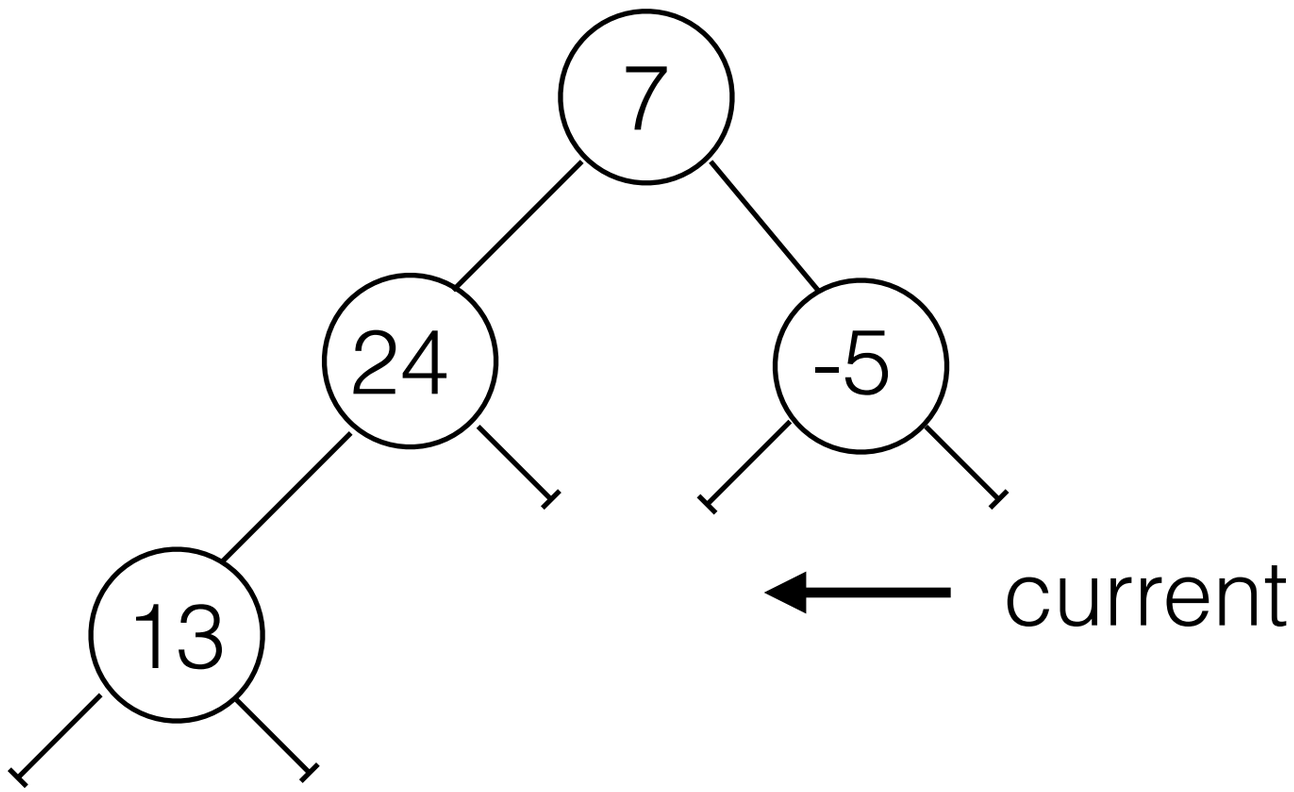
bitstring = “10”, **item**= 3

Add 3



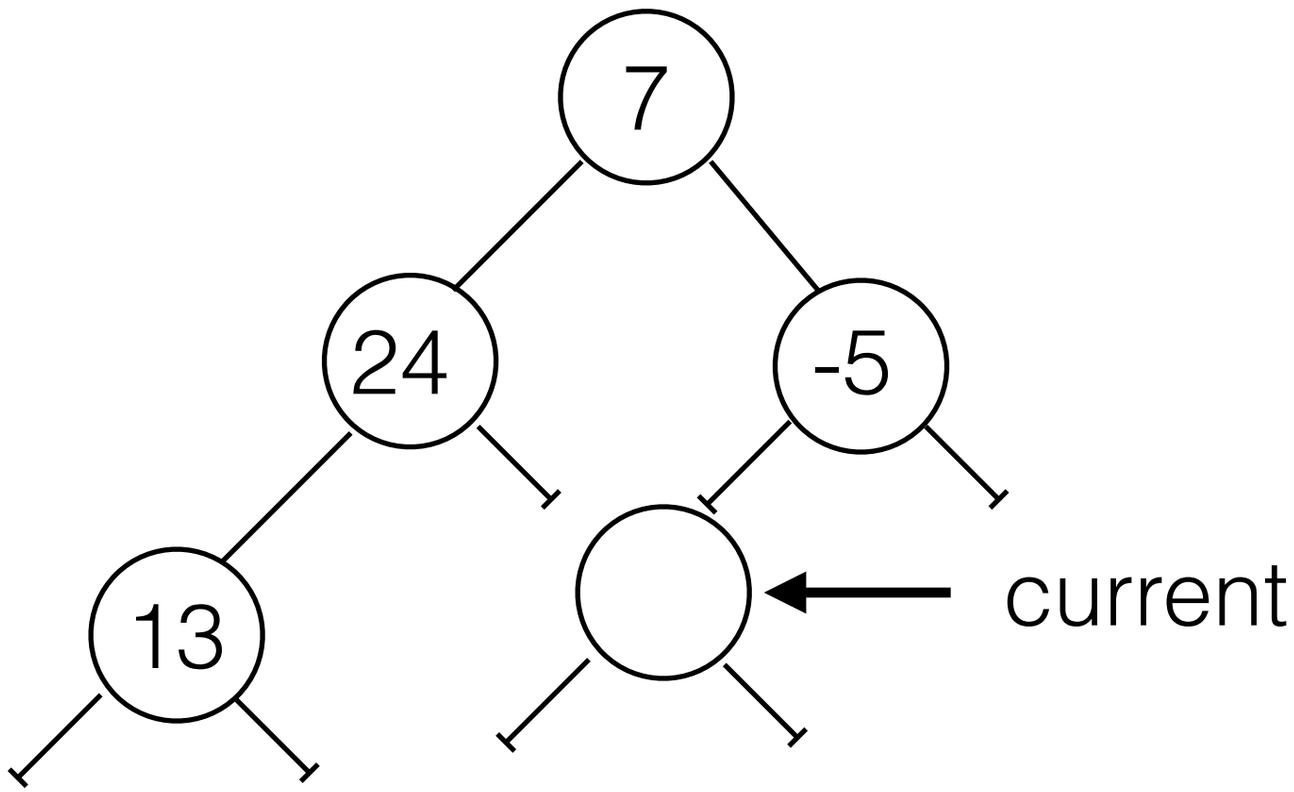
bitstring = “1̲0”, **item**= 3

Add 3



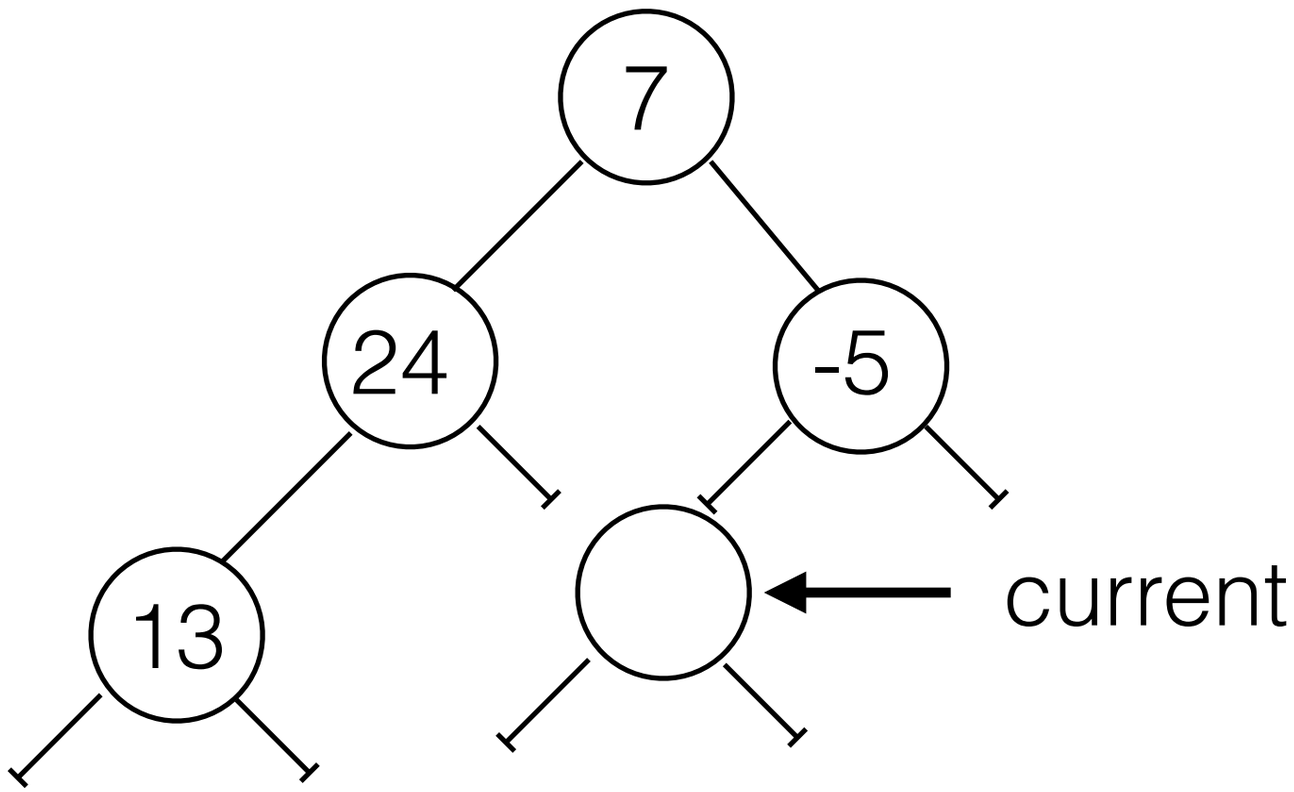
bitstring = “1<u>0</u>”, **item**= 3

Add 3



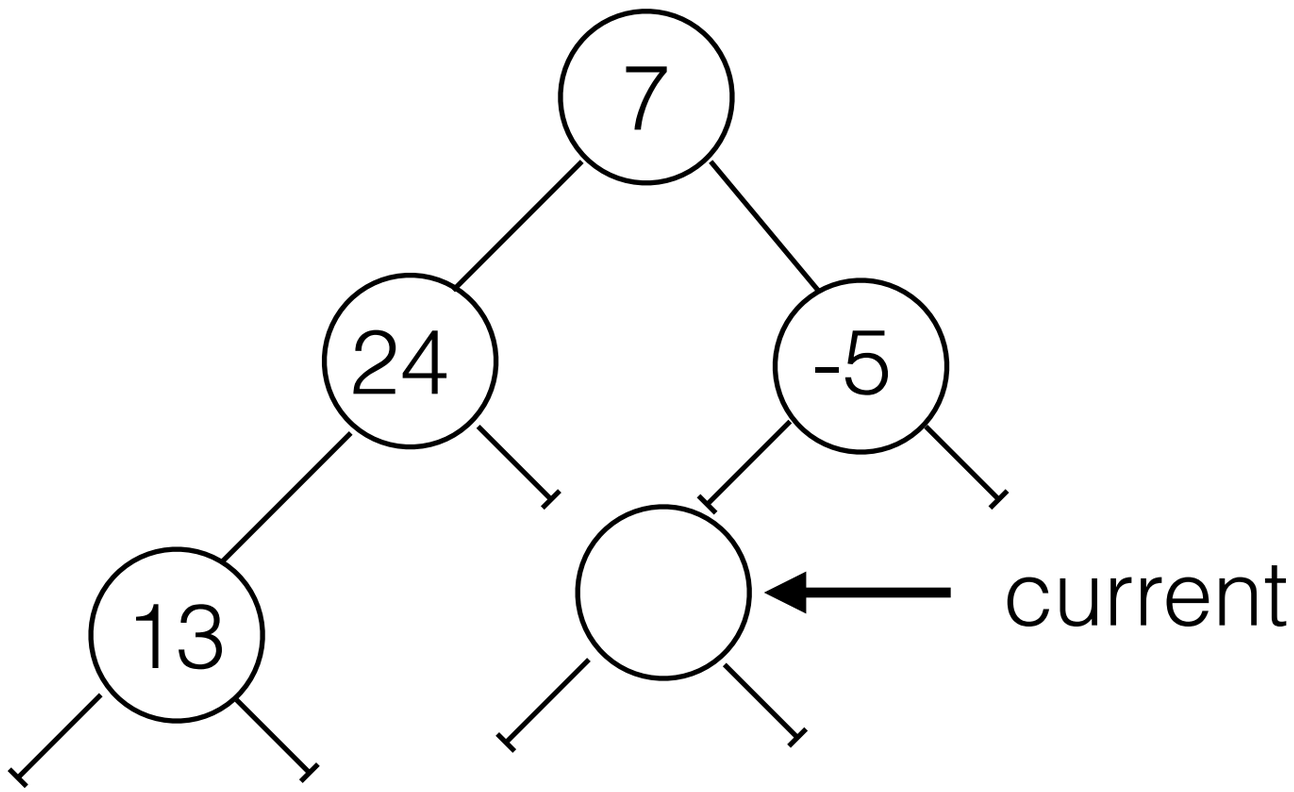
bitstring = “1<u>0</u>”, **item**= 3

Add 3



bitstring = “10”, **item**= 3

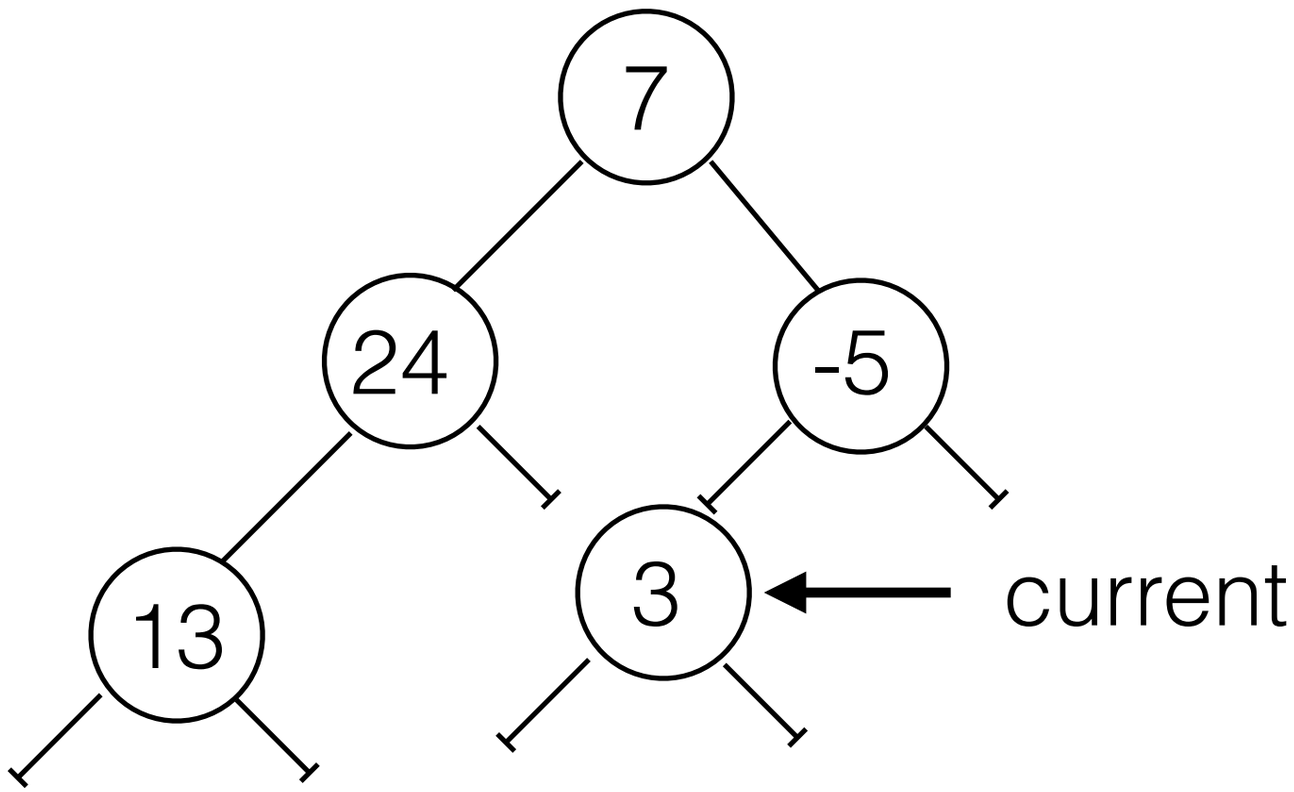
Add 3



bitstring = “10”, **item**= 3

Iteration ended, so this must be the place….

Add 3



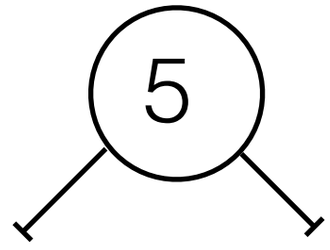
bitstring = “10”, **item**= 3

Examples

bitstring = “”, **item**= 5

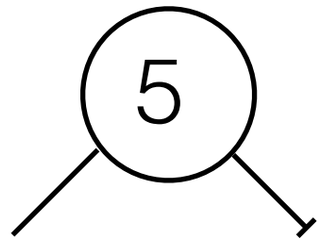
Examples

bitstring = “”, **item**= 5



Examples

bitstring = “”, **item**= 5

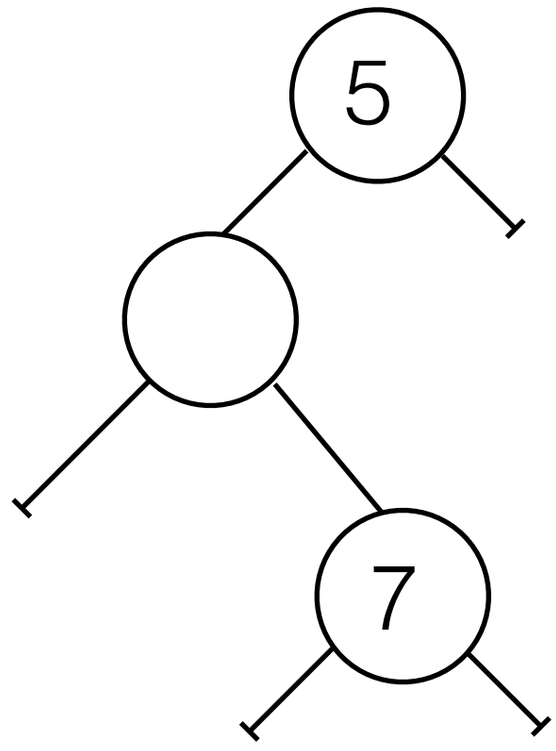


bitstring = “01”, **item**= -7

Examples

bitstring = “”, **item**= 5

bitstring = “01”, **item**= -7

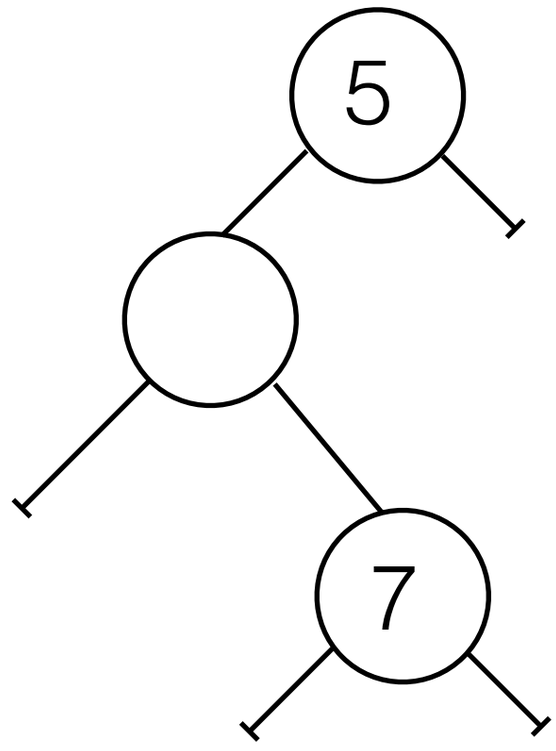


Examples

bitstring = “”, **item**= 5

bitstring = “01”, **item**= -7

bitstring = “ ”, **item**= 2

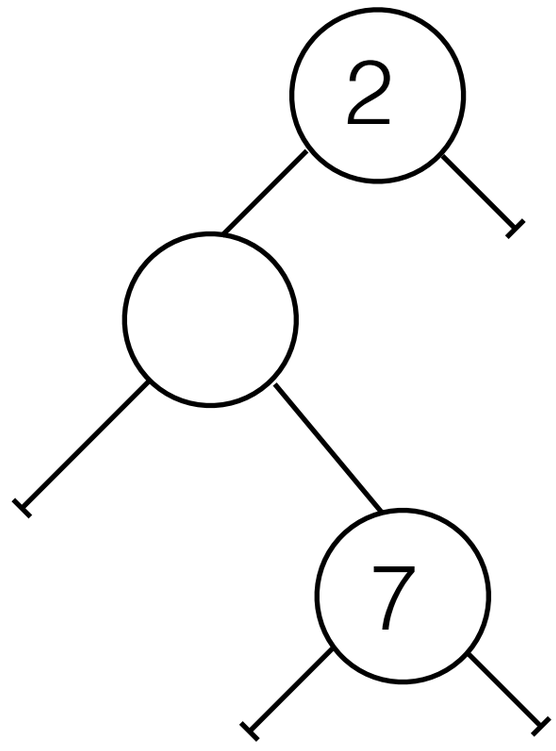


Examples

bitstring = “”, **item**= 5

bitstring = “01”, **item**= -7

bitstring = “ ”, **item**= 2

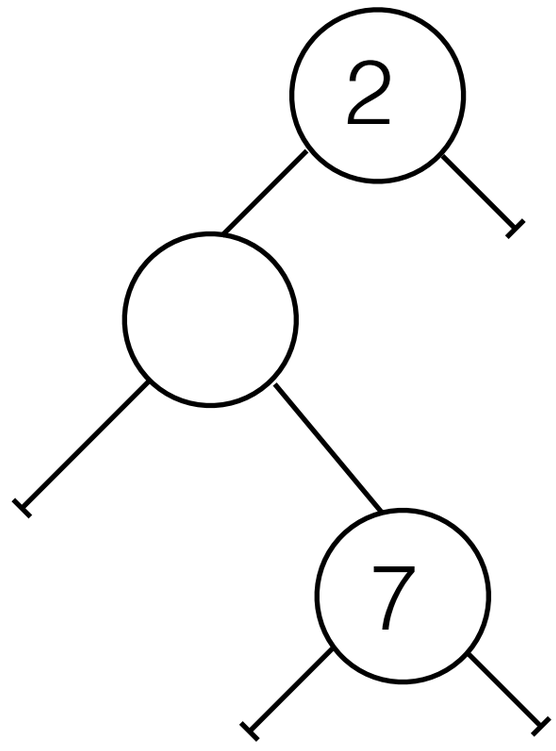


Examples

bitstring = “”, **item**= 5

bitstring = “01”, **item**= -7

bitstring = “ ”, **item**= 2



Recursively explore subtree
following “bitstring directions”

```
def add(self, item, position_bitstring):
```

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

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

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

add sets up the recursion starting at the root
and calls an auxiliary method

```
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):
```

```
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:
```

```
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()
```

```
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()
```

Add empty node if it does not exist

```
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)
```

```
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)
```

Explore left branch

```
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)
```

```
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)
```

Explore right branch

```
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
```

```
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
```

Bitstring is telling me I have arrived at the correct stop

```
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

Traversal

- Systematic way of **visiting**/processing **all the nodes**

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**.

Left subtree

Right subtree

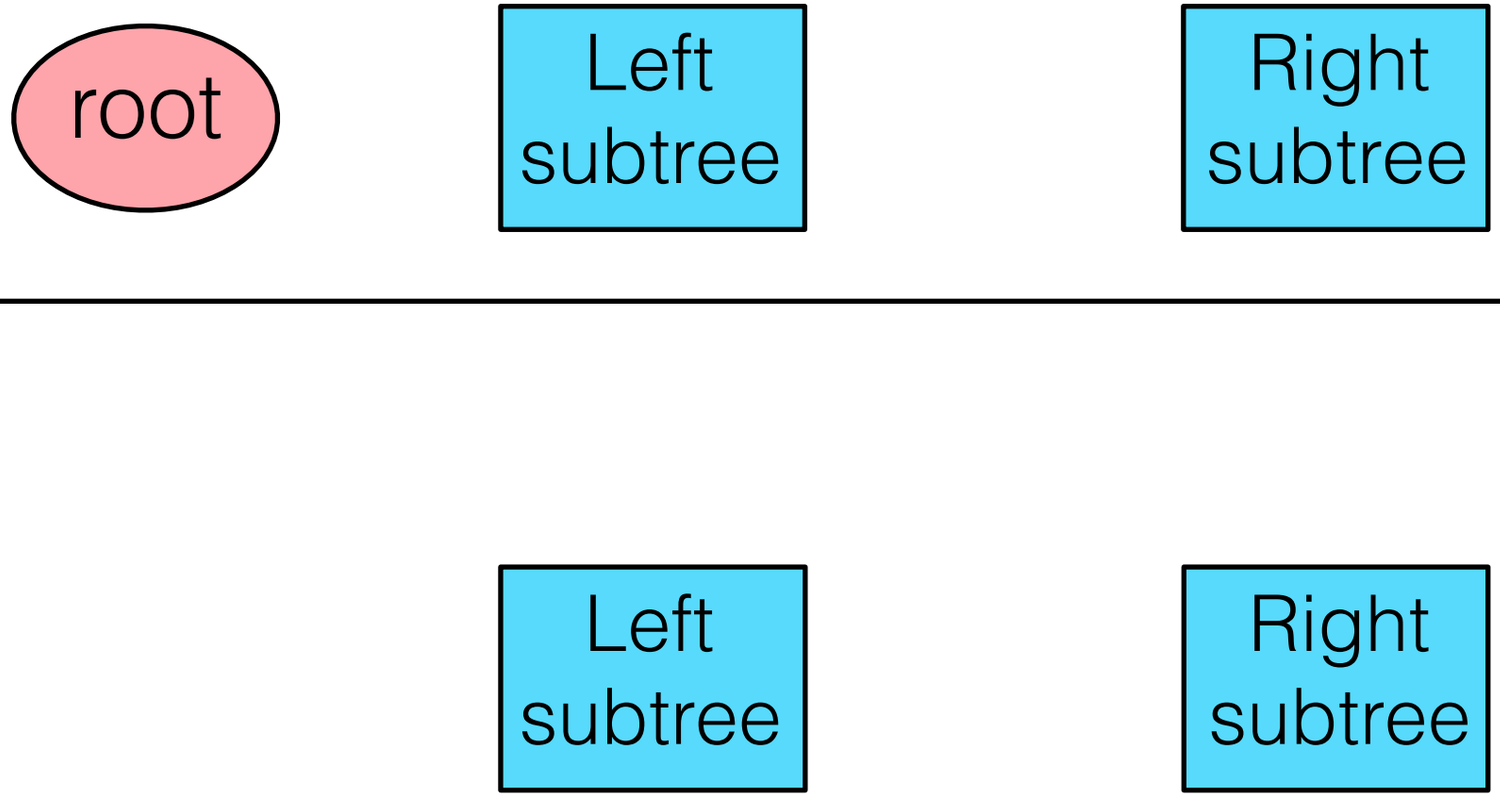
Left subtree

Right subtree

Left subtree

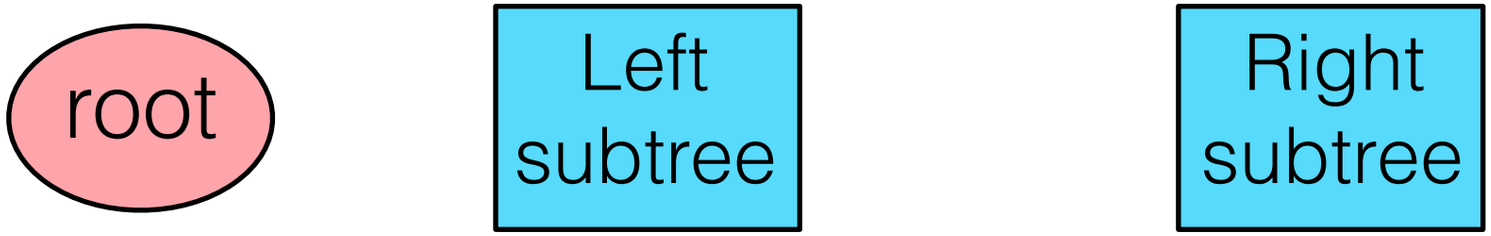
Right subtree

Preorder

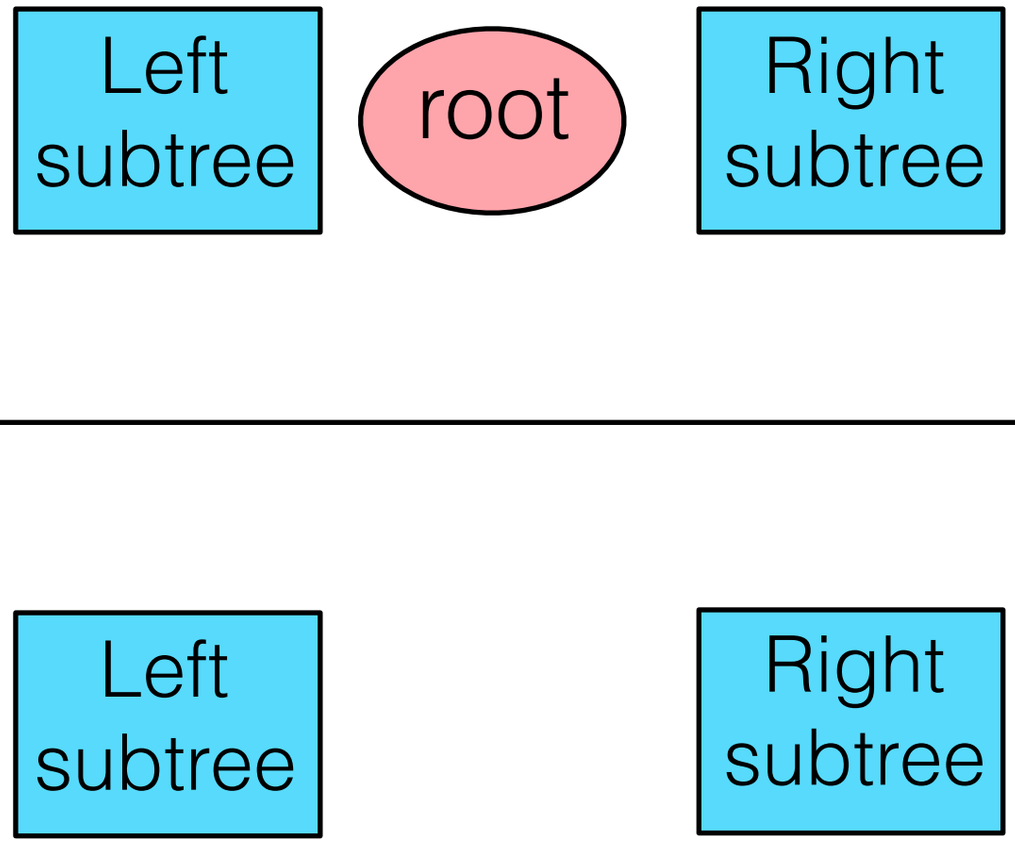

root
Left subtree
Right subtree

Left subtree
Right subtree

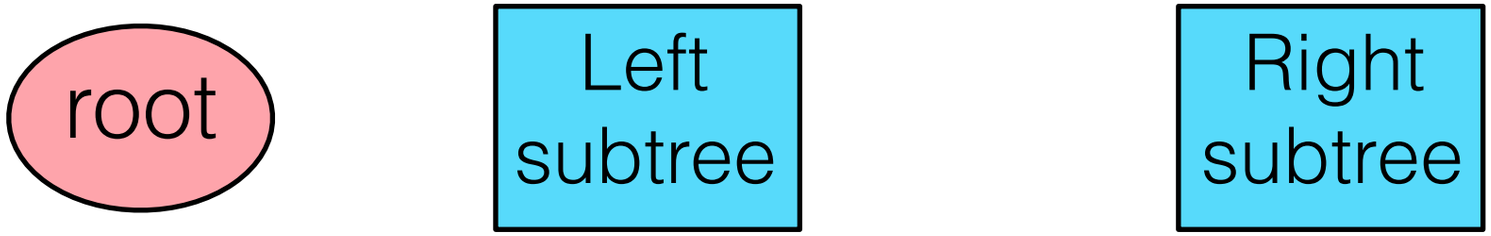
Preorder



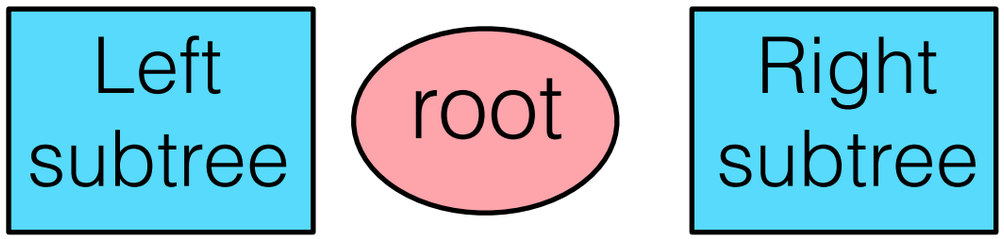
Inorder



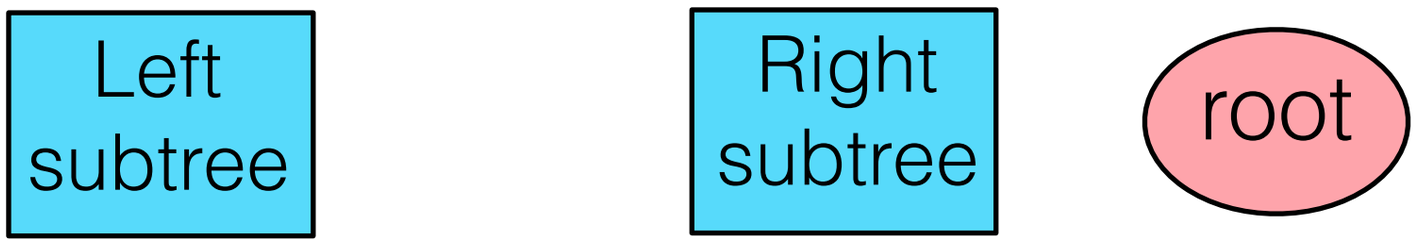
Preorder



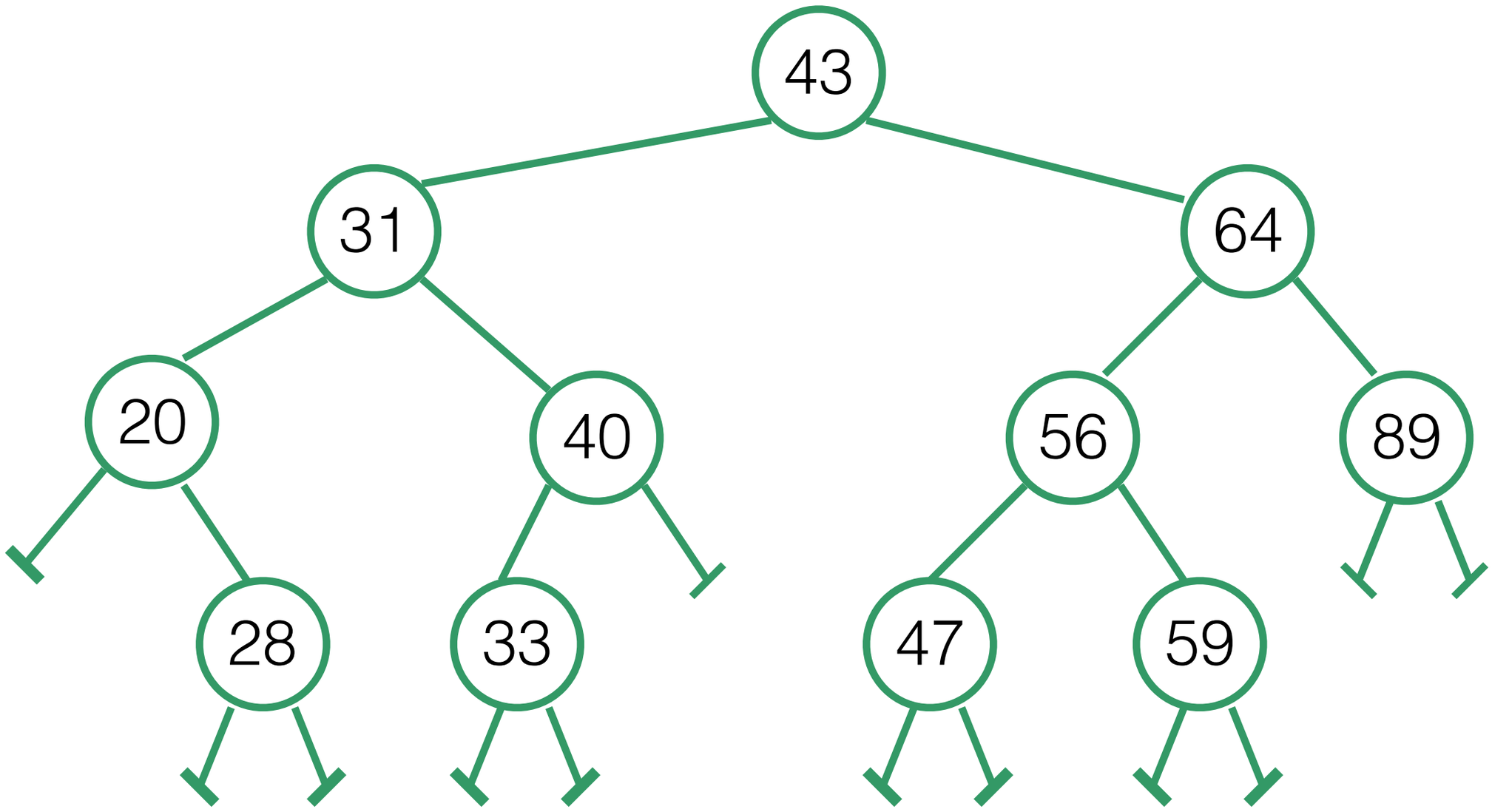
Inorder



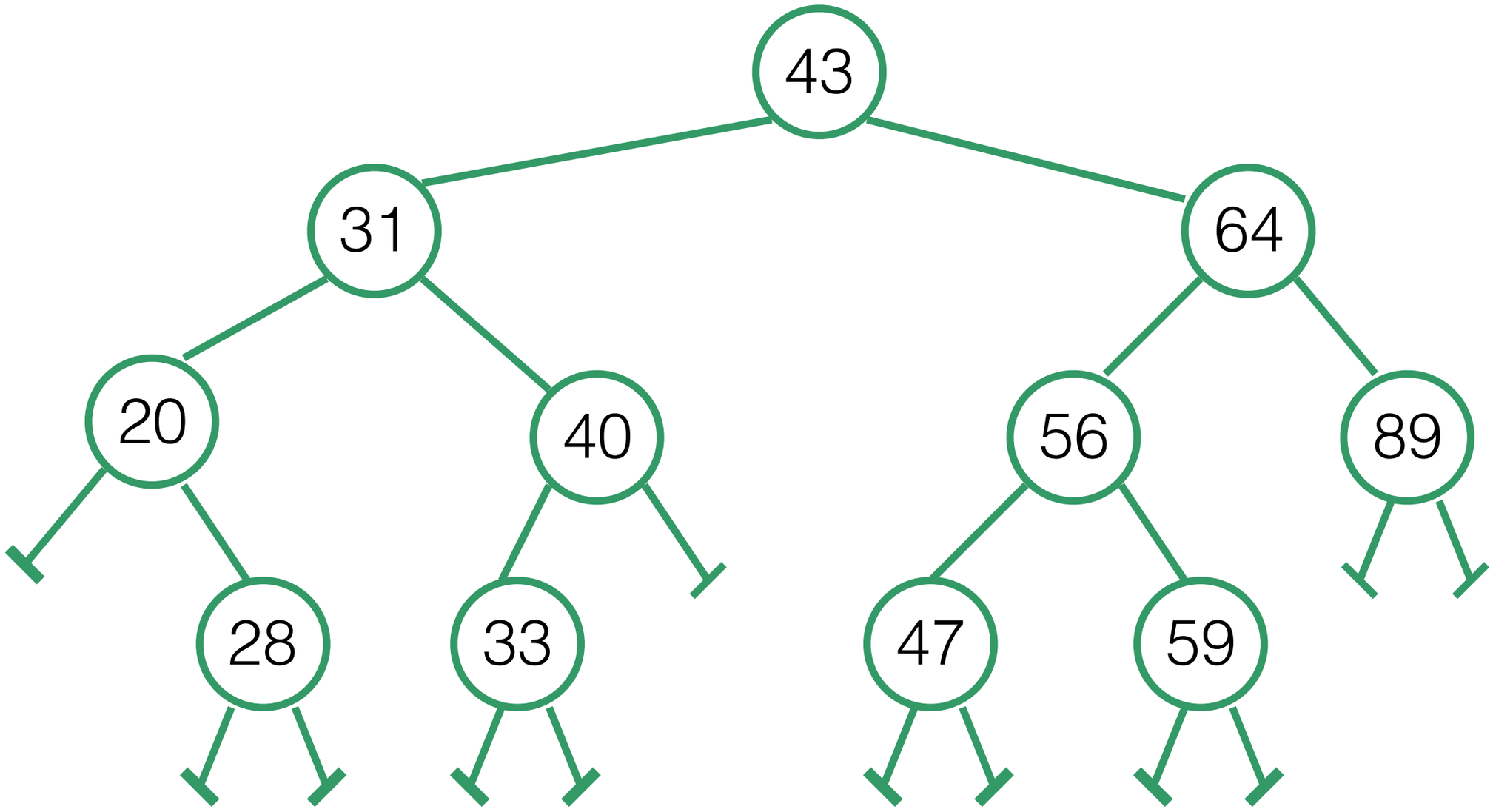
Postorder



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

1) Print the **root** node

2) Traverse the **left** subtree

3) Traverse the **right** subtree

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

Print Preorder Traversal

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

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

Auxiliary method receives a reference to the “next root”

Print Preorder Traversal

1) Print the **root** node

2) Traverse the **left** subtree

3) Traverse the **right** subtree

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

def _print_preorder_aux(self, current):
```

Print Preorder Traversal

1) Print the **root** node

2) Traverse the **left** subtree

3) Traverse the **right** subtree

```
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 Preorder Traversal

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

```
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
```

Work to do…

Print Preorder Traversal

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

```
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)
```

Print Preorder Traversal

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

```
class TreeNode:
    def __init__(self, item=None, left=None, right=None):
        self.item = item
        self.left = left
        self.right = right

    def __str__(self):
        return str(self.item)
```

```
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)
```

Print Preorder Traversal

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

```
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)
```

Print Preorder Traversal

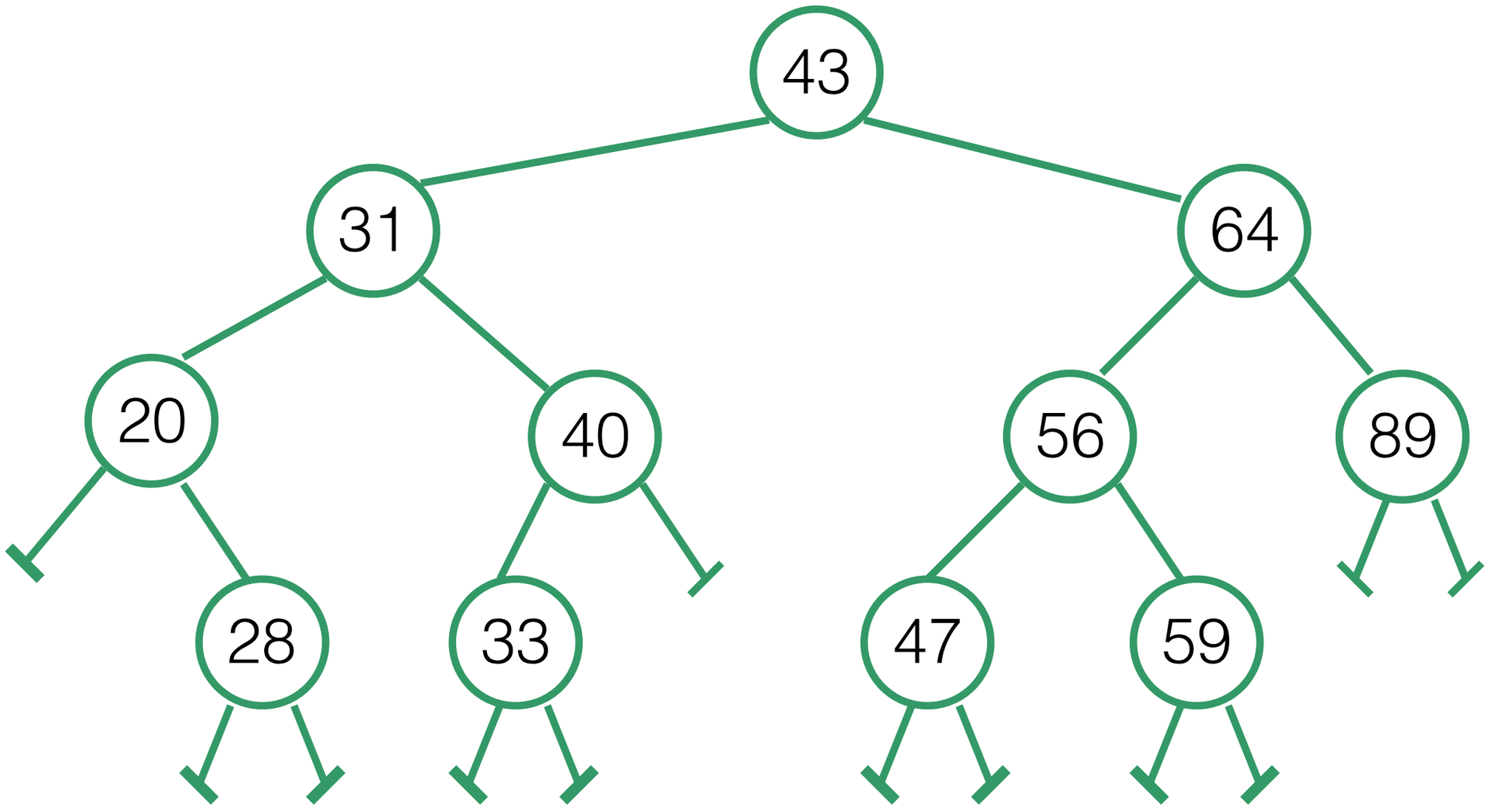
1) Print the **root** node
2) Traverse the **left** subtree
3) Traverse the **right** subtree

```
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)
```

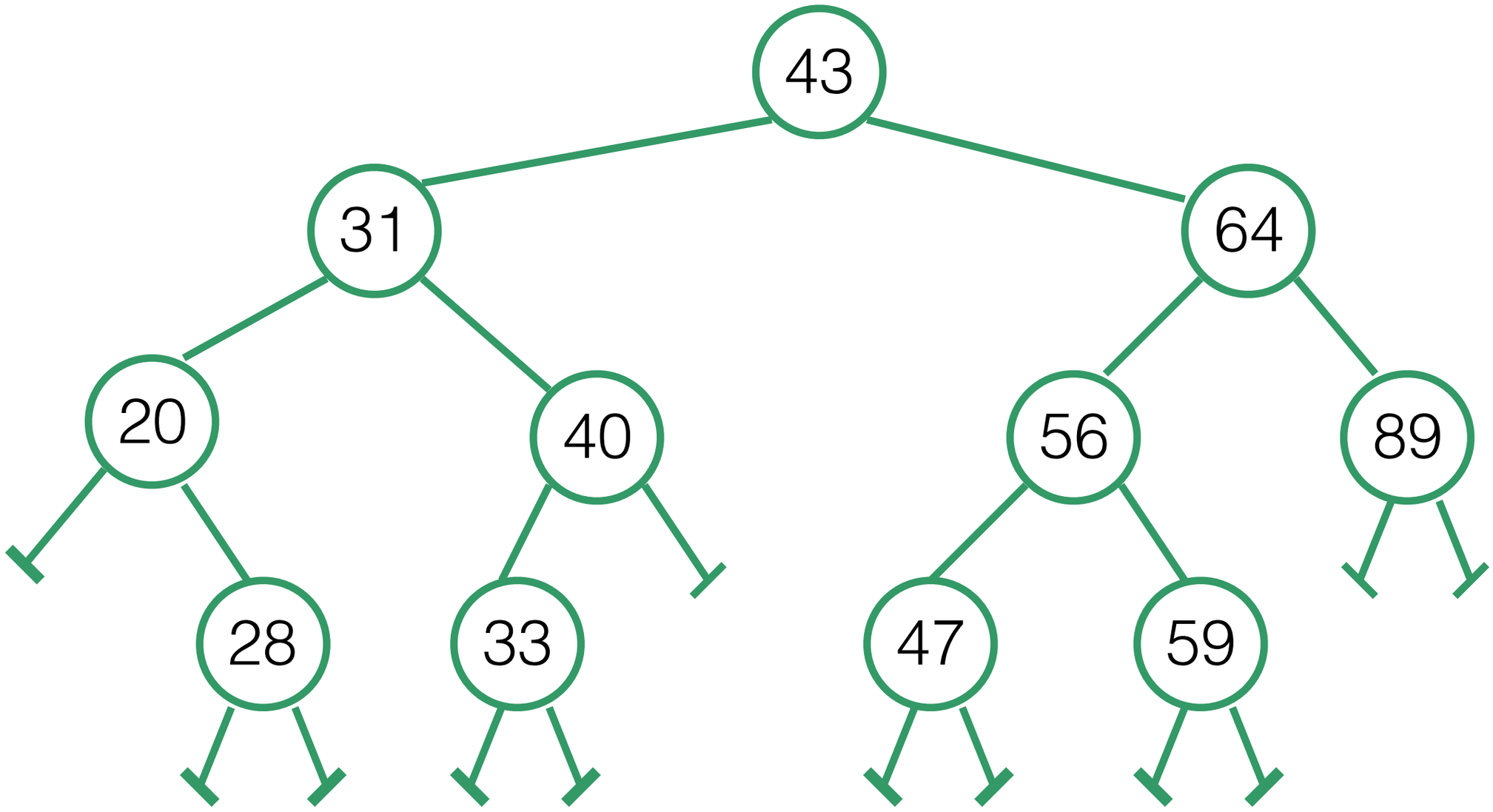
Print Preorder Traversal

```
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)
```

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):
```

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
```

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
```

Work to do…

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 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)
```

Print In-order Traversal

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

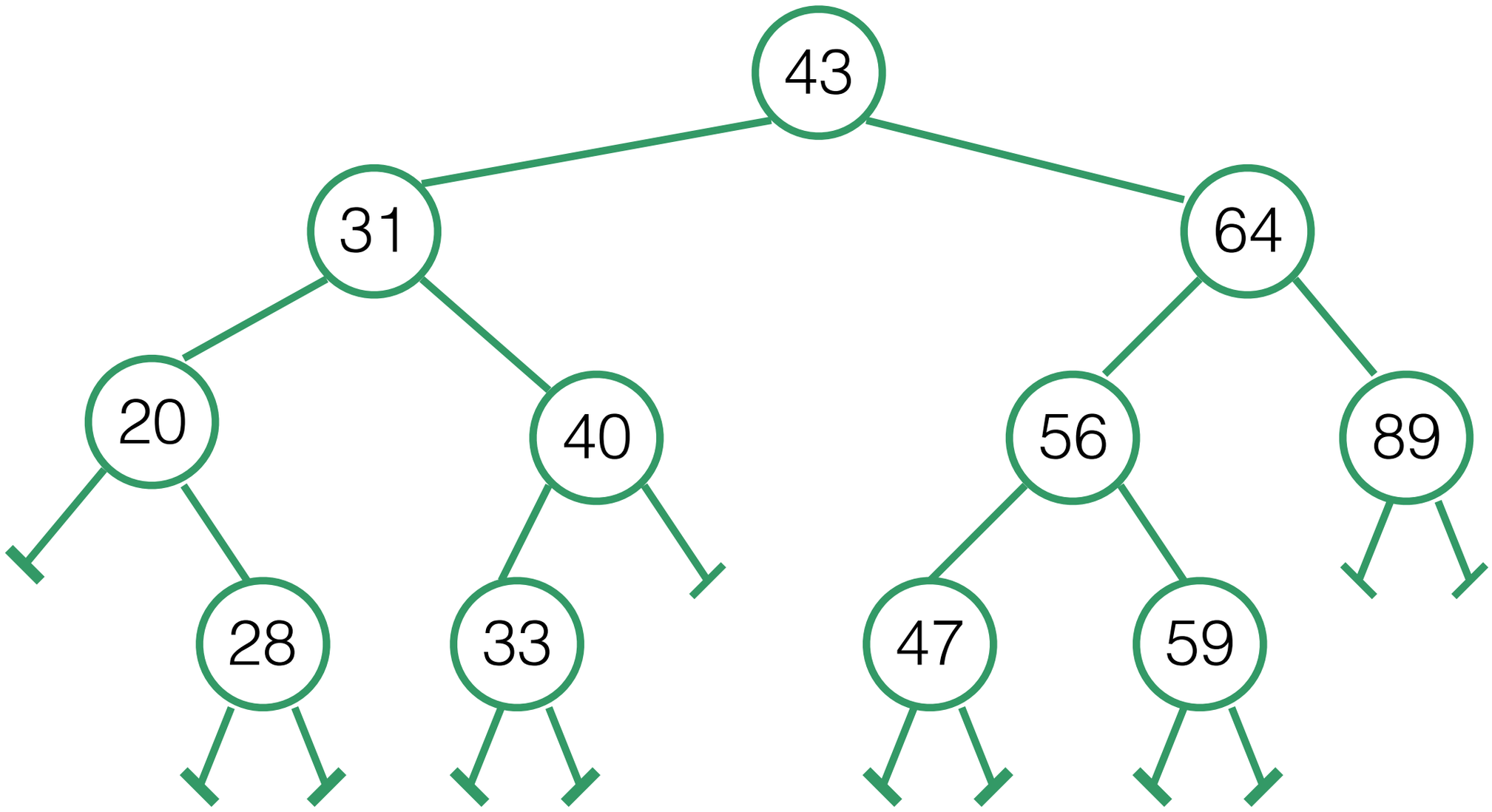
```
def print_inorder(self):
    self._print_inorder_aux(self.root)

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)
```

Print In-order Traversal

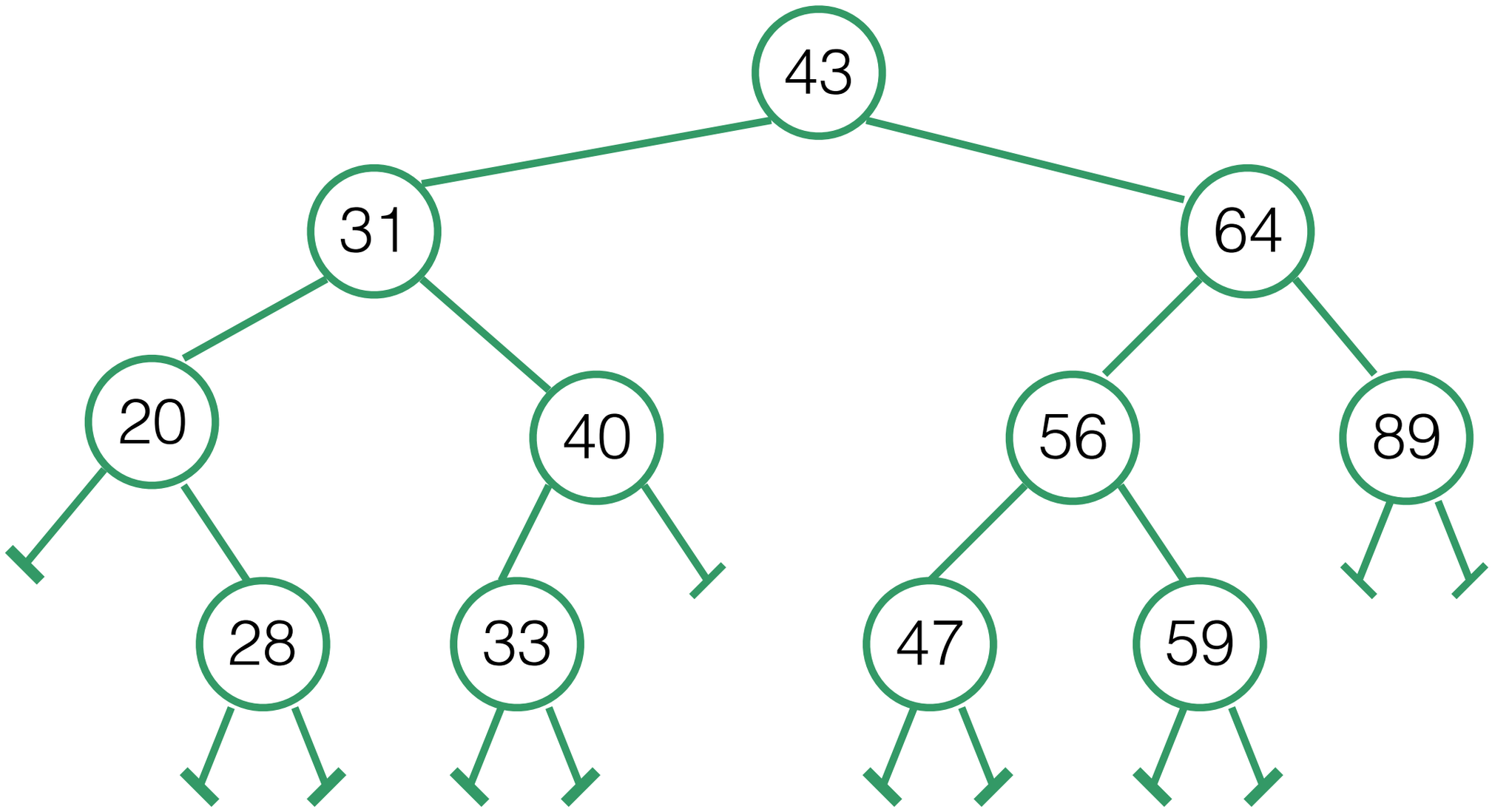
1) Traverse the **left** subtree
2) Print the **root** node
3) Traverse the **right** subtree

```
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

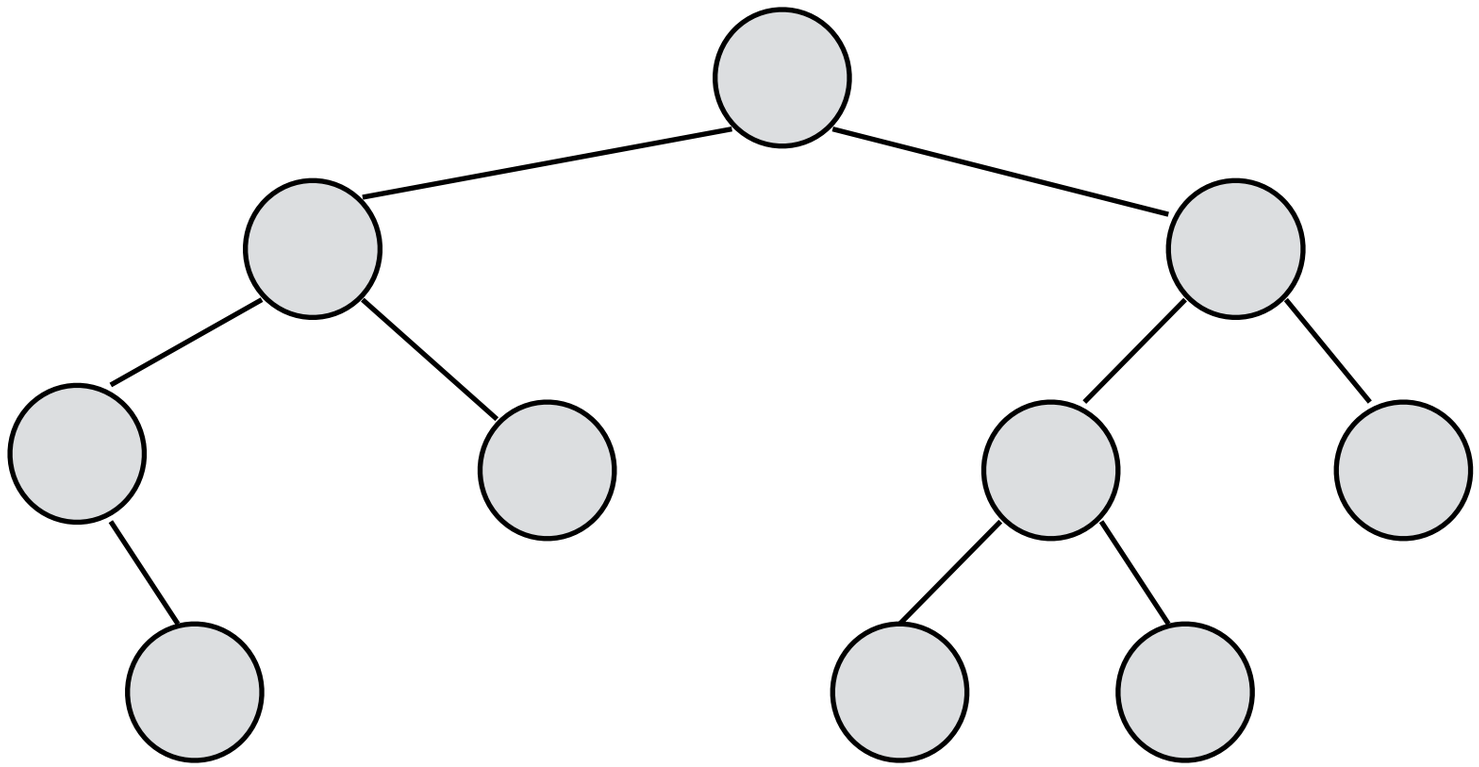
Print Post-order Traversal

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)
```

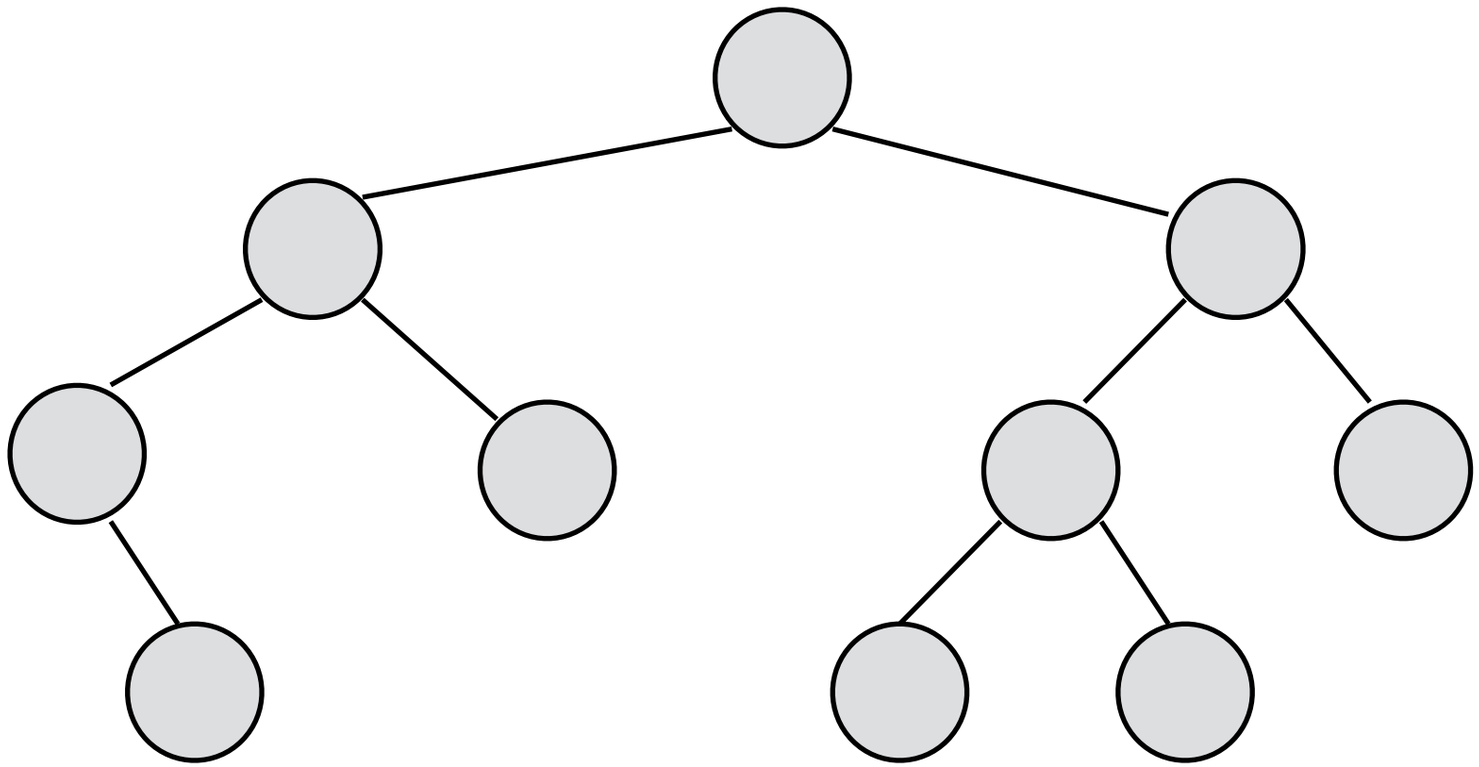
Computing the size of a tree

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



Computing the size of a tree

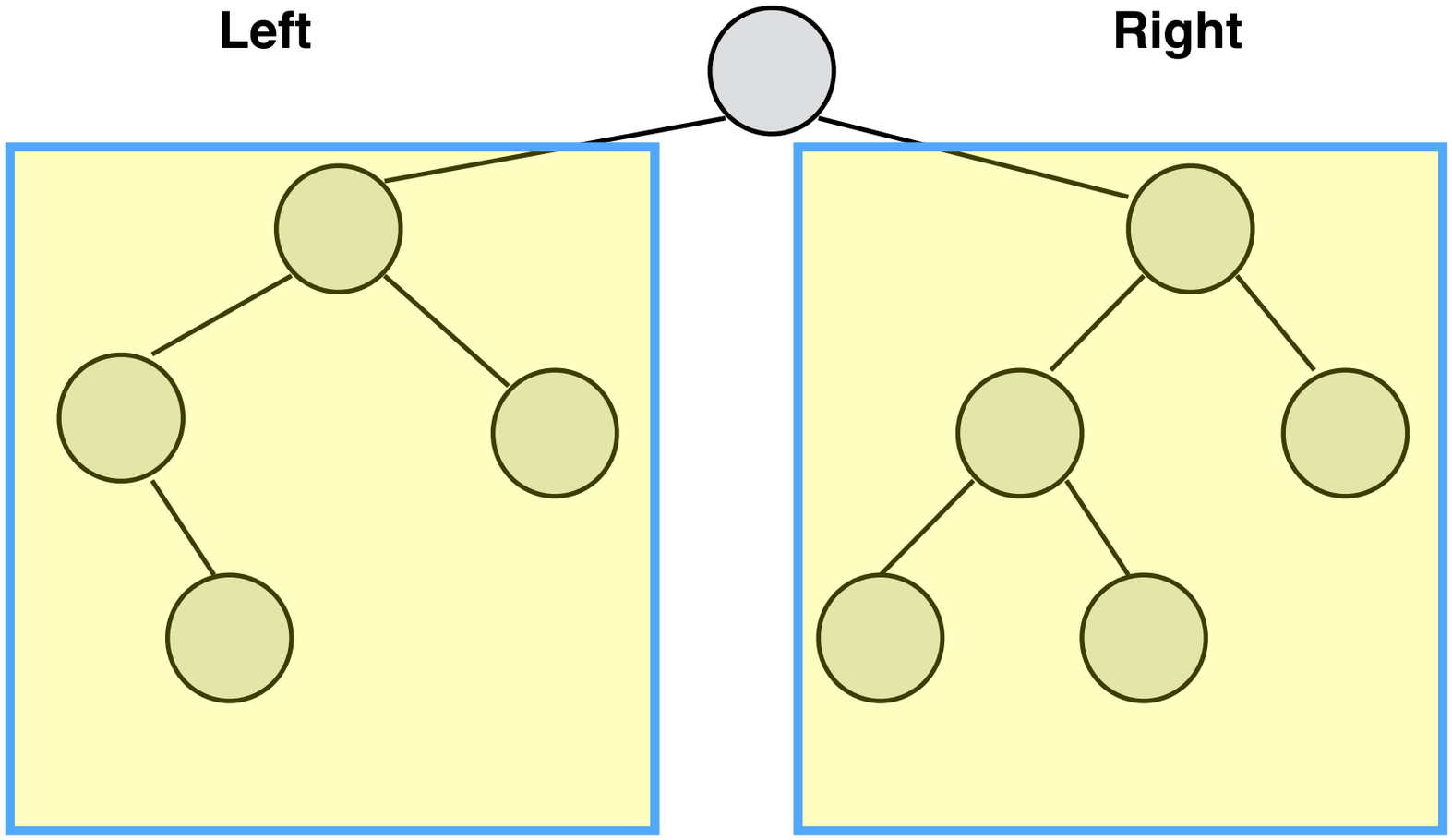
Returns the **number of nodes in the tree** (without modifying the tree)



10

Computing the size of a tree

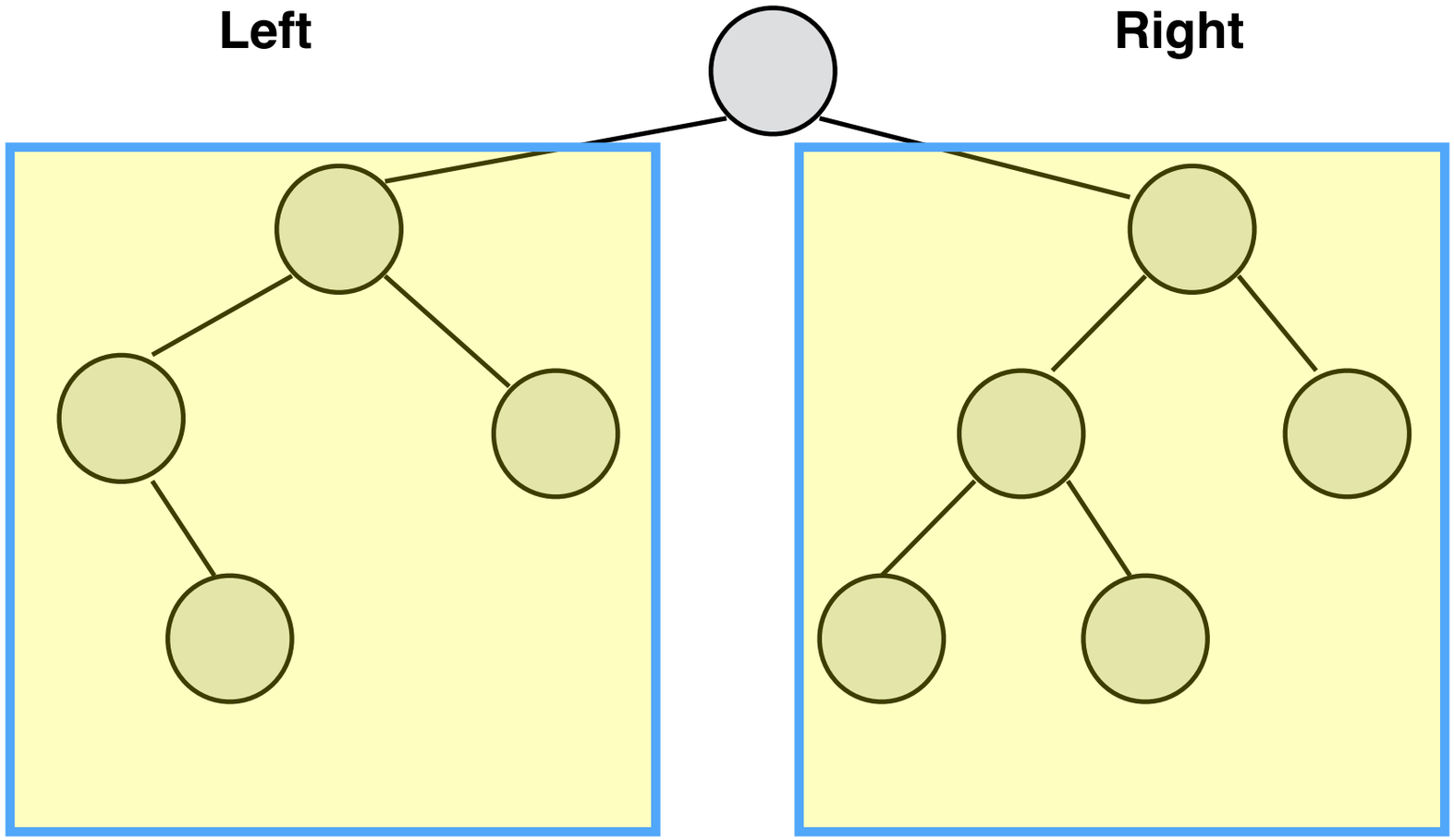
Returns the **number of nodes in the tree** (without modifying the tree)



10

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_aux(self, current):
```

Computing the size of a tree

```
def len_aux(self, current):
    if current is None:
```

Computing the size of a tree

```
def len_aux(self, current):
    if current is None:
        return 0
```

Computing the size of a tree

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

Computing the size of a tree

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

Can we implement an Iterator to traverse a Binary Tree?

A) Yes

B) No

C) I have no idea.

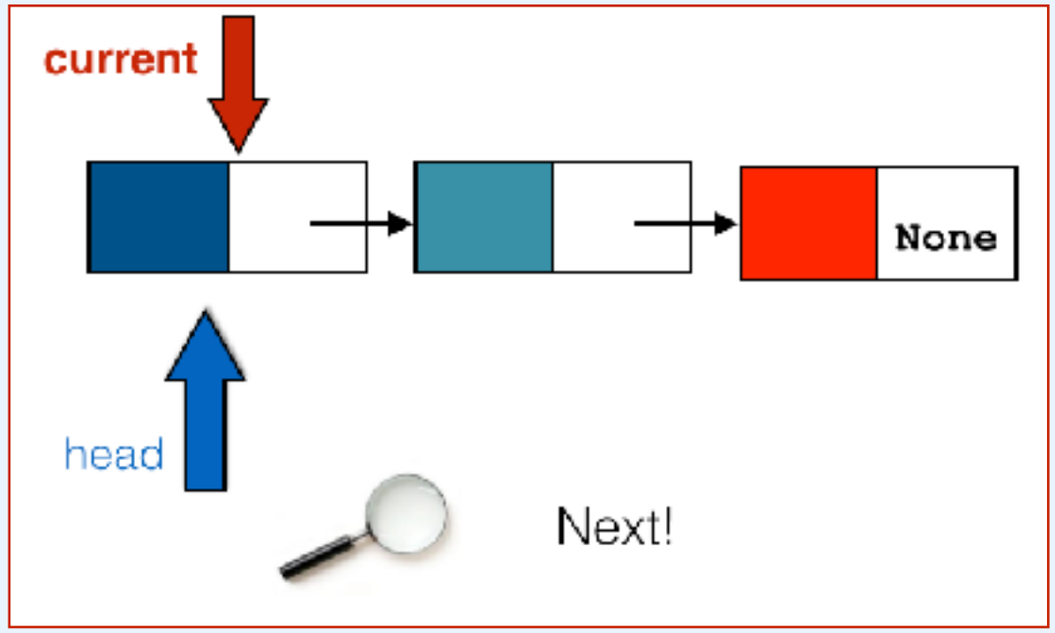
Can we implement an Iterator to traverse a Binary Tree?

A)Yes? You may need to use a Stack

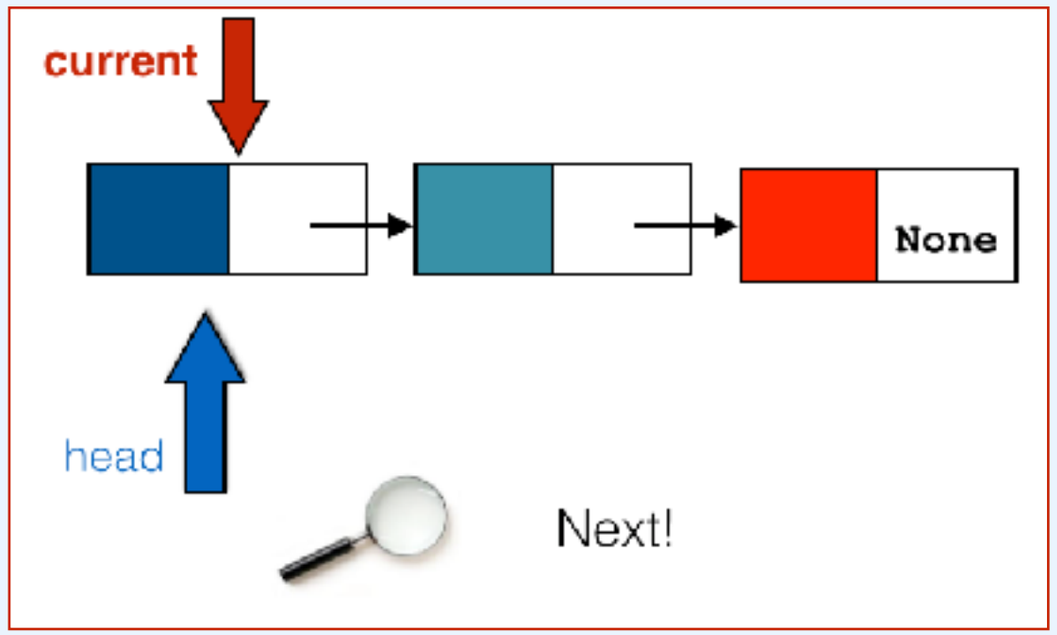
B) No

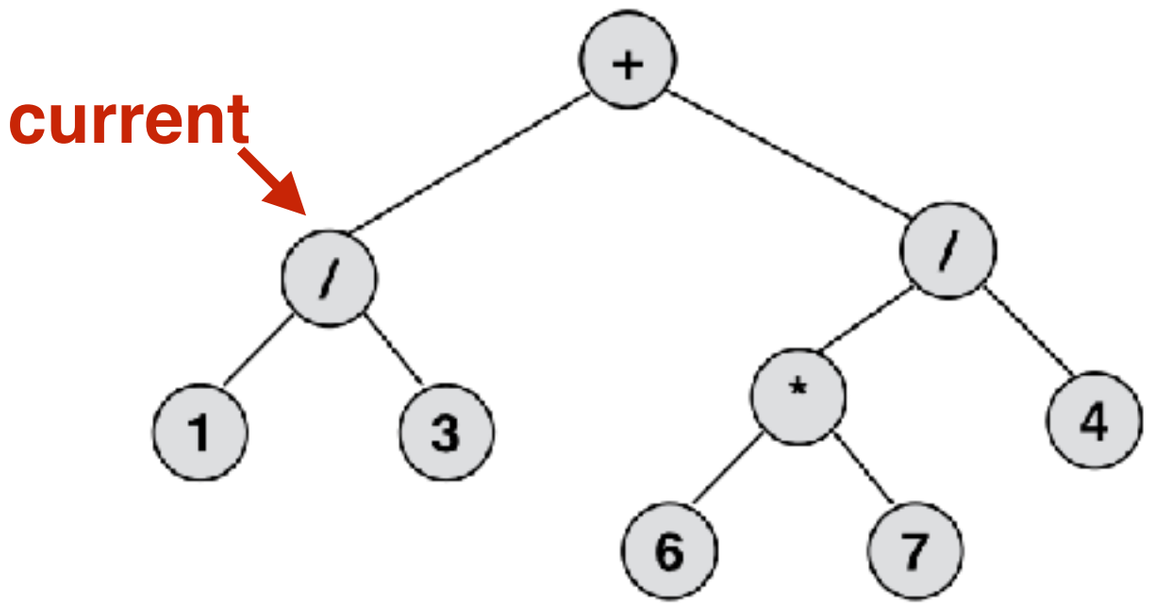
C) I have no idea.

```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
```

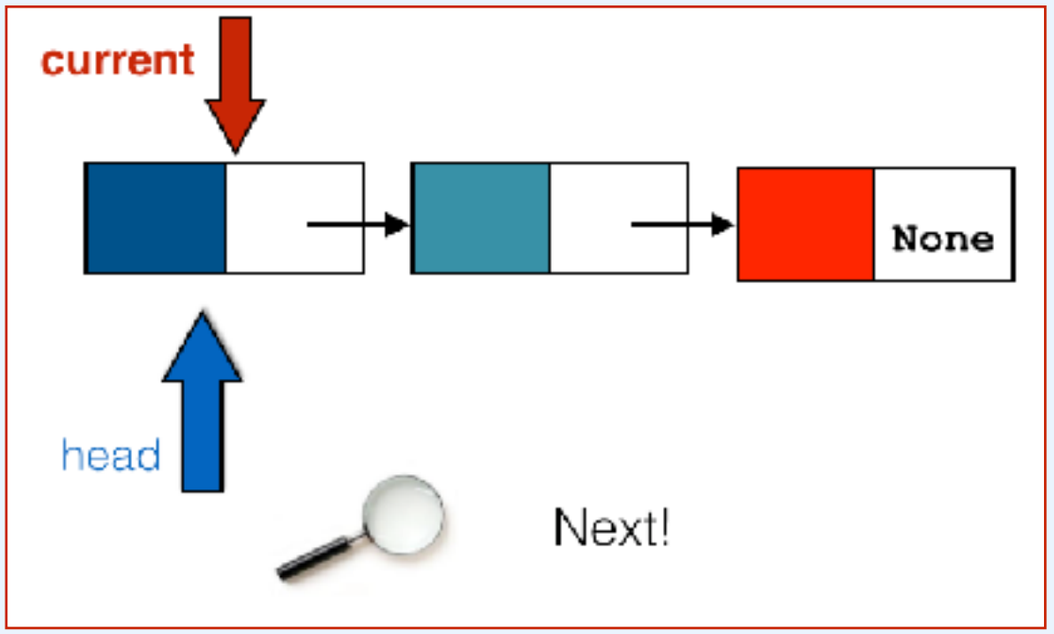


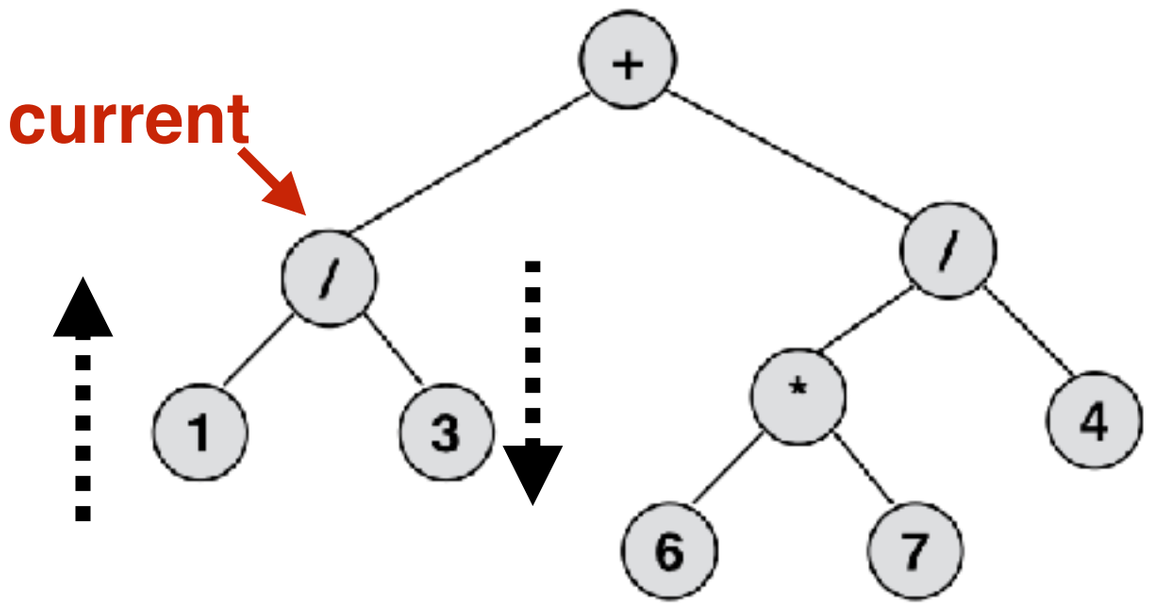
```python
class ListIterator:
    def __init__(self,head):
        self.current = head

    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
```



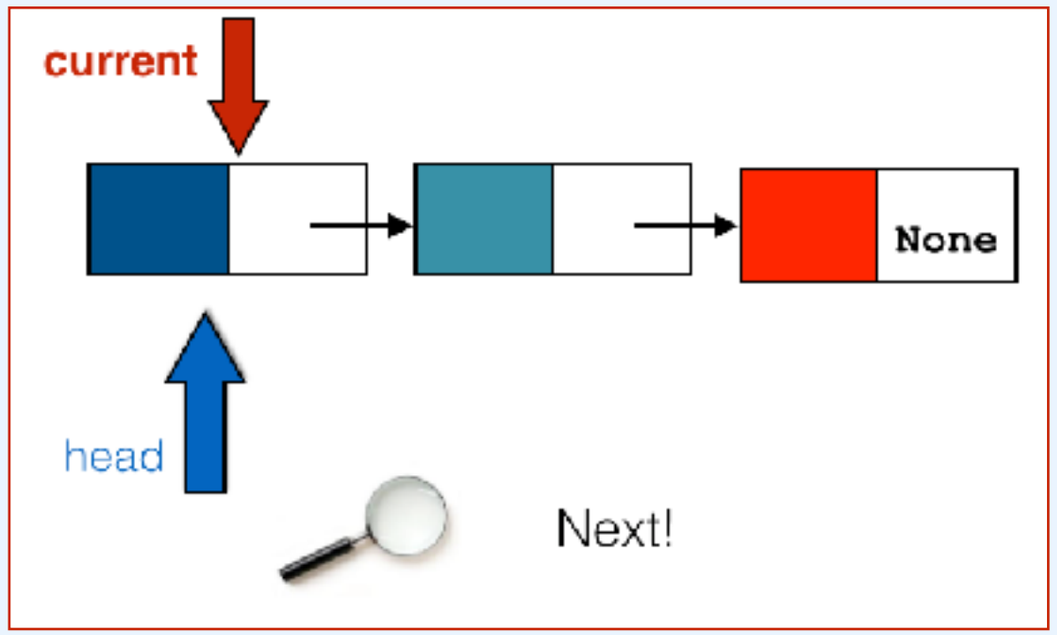


```python
class ListIterator:
    def __init__(self,head):
        self.current = head

    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
```

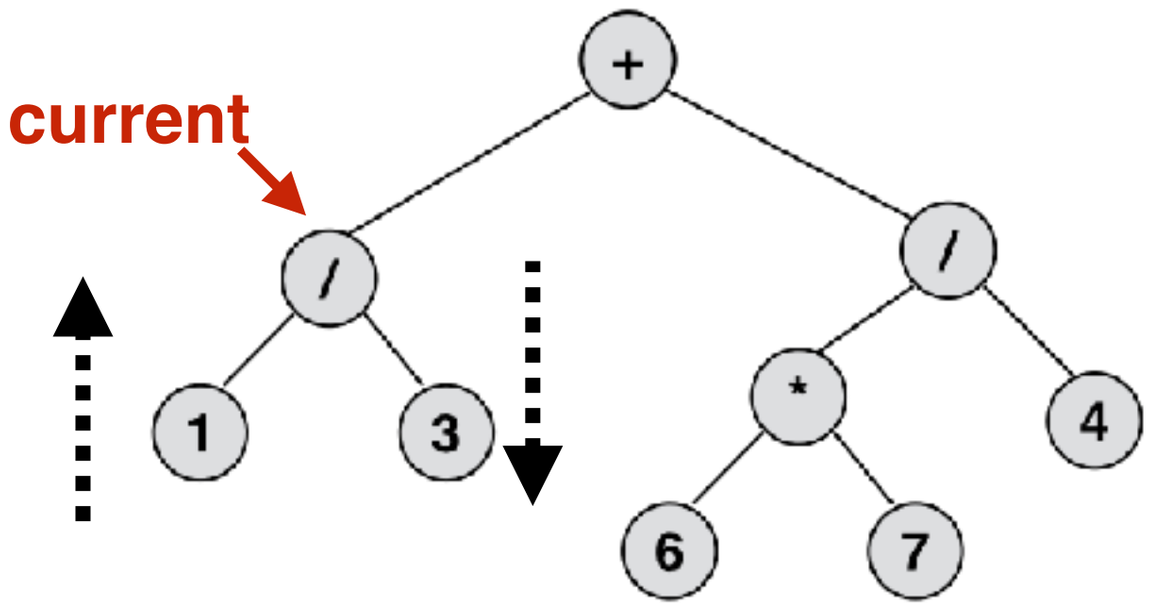




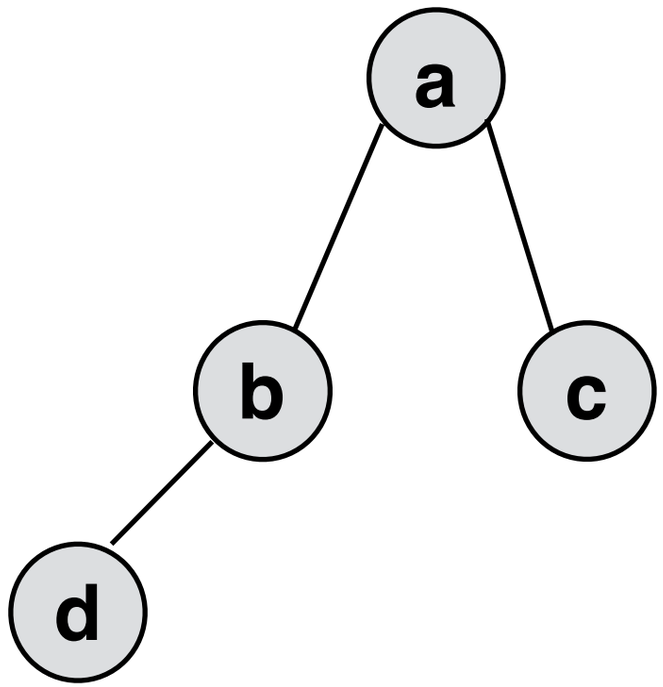
```python
class ListIterator:
    def __init__(self,head):
        self.current = head

    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
```



?

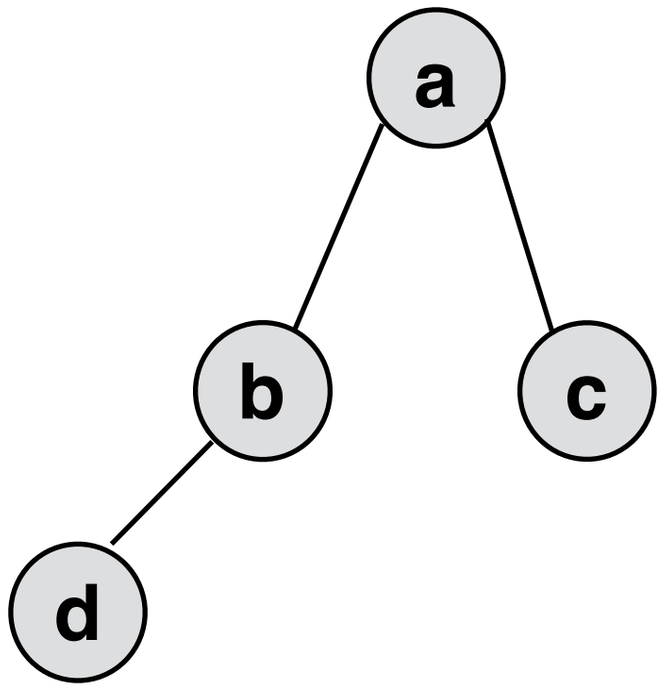


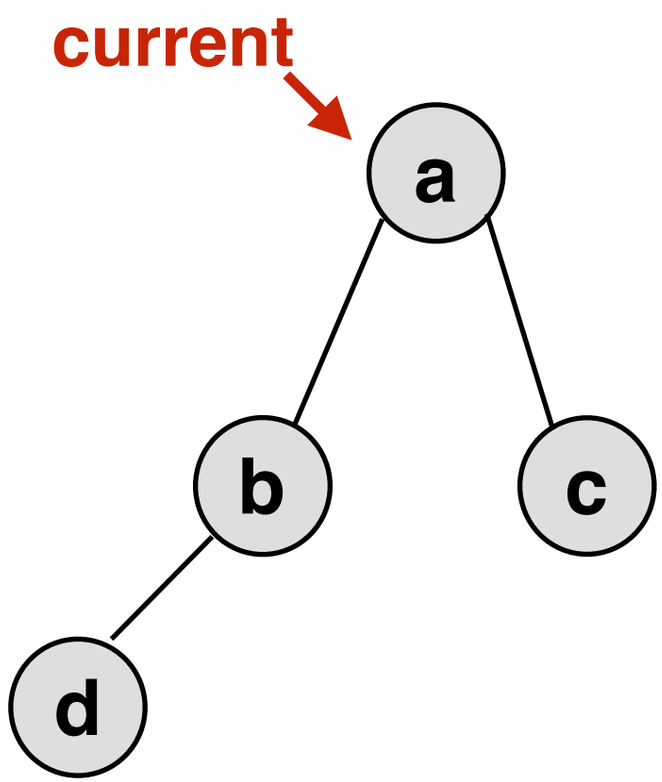
State of the **Iterator** on creation

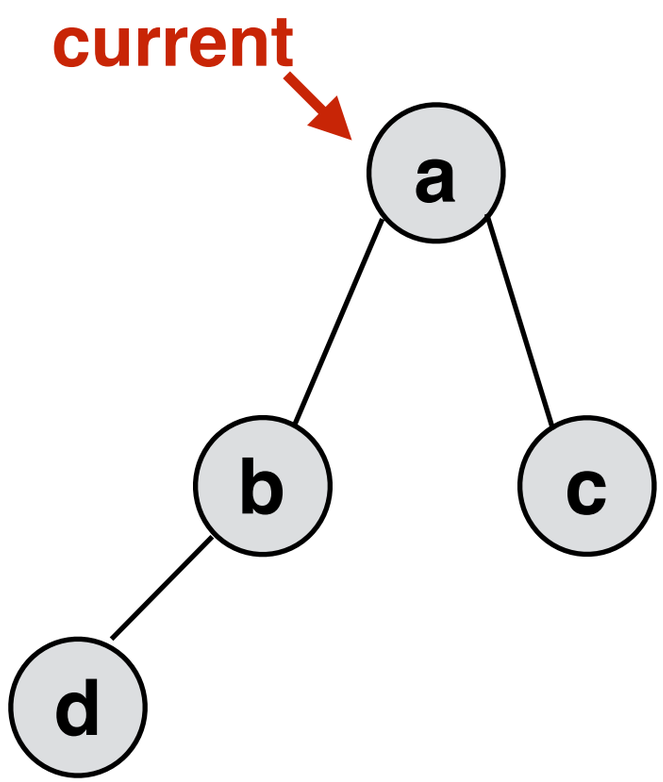


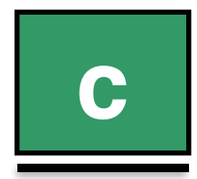
self.stack

Next!

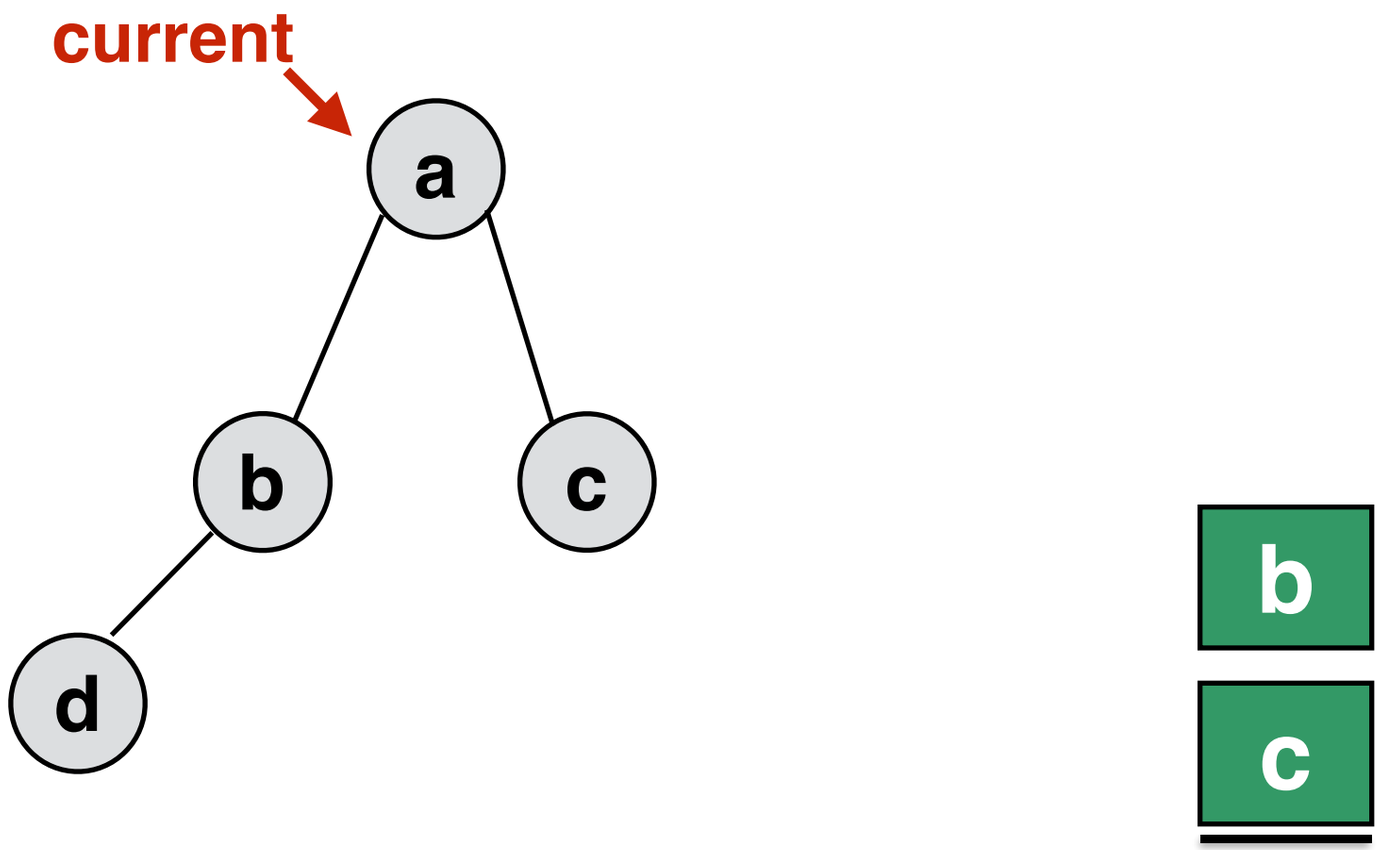



current
a
b
c
d



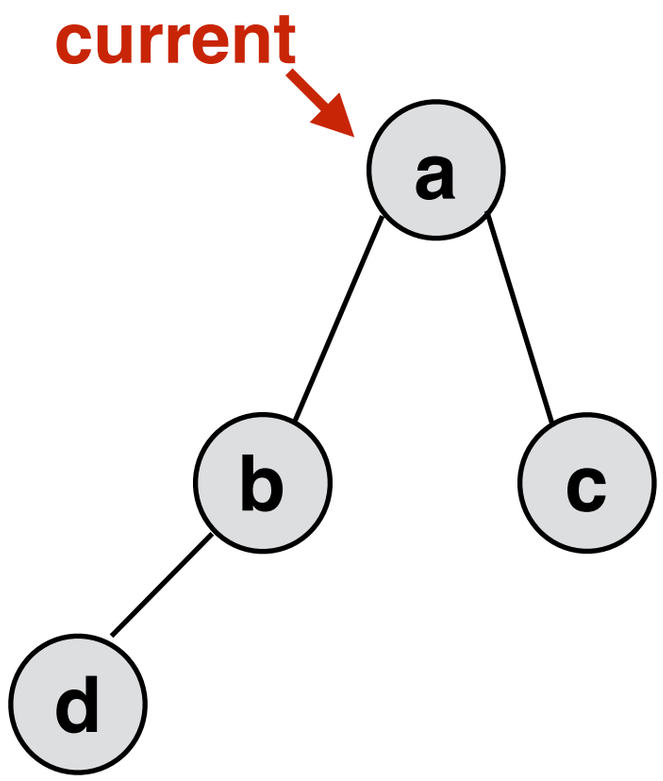


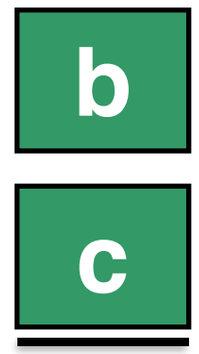
Push what is to the right of current.



Push what is to the left of current.

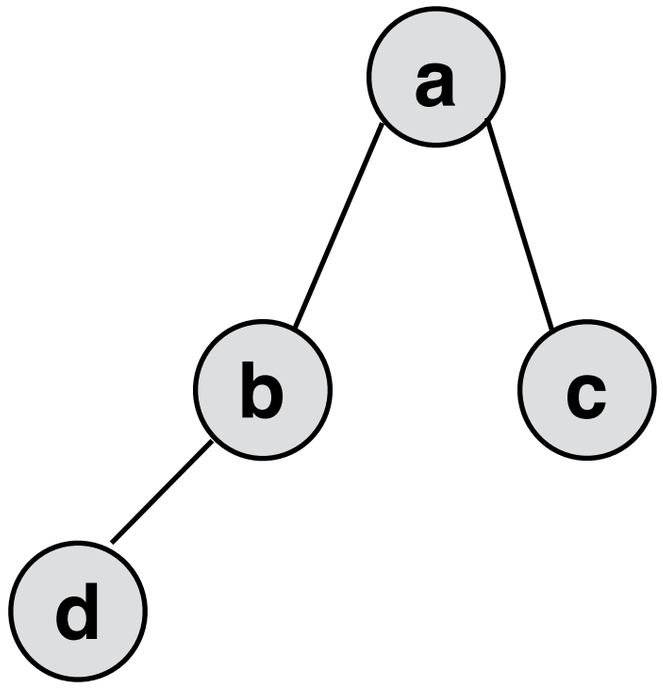
return current.item

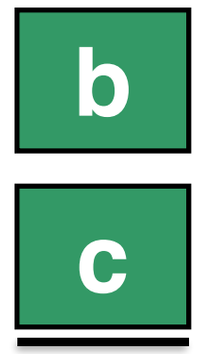




a

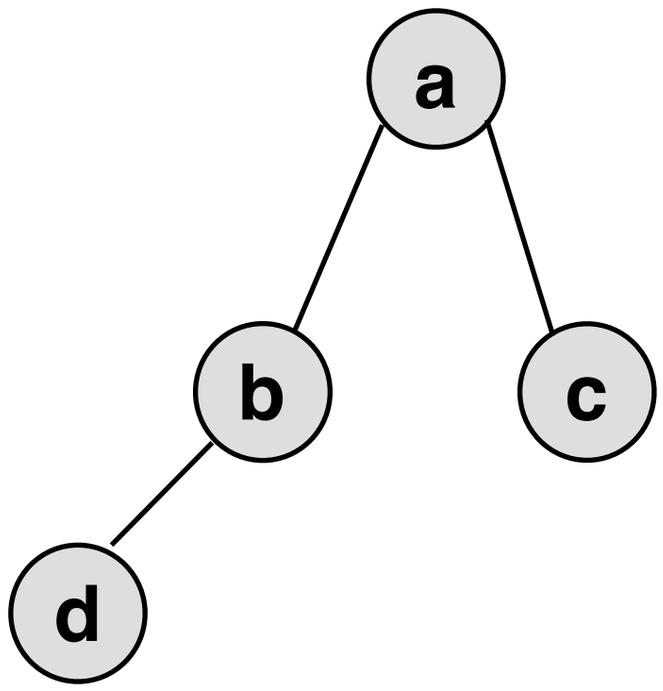
return current.item

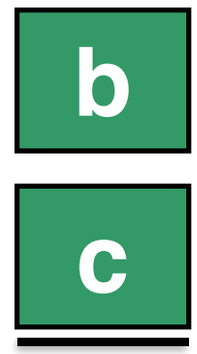




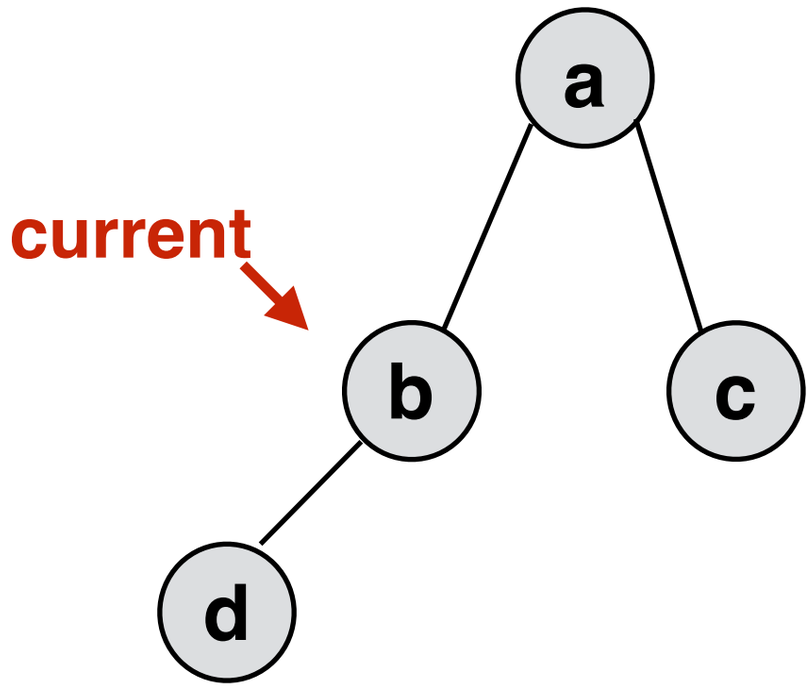
a

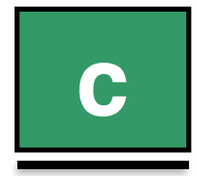
Next!





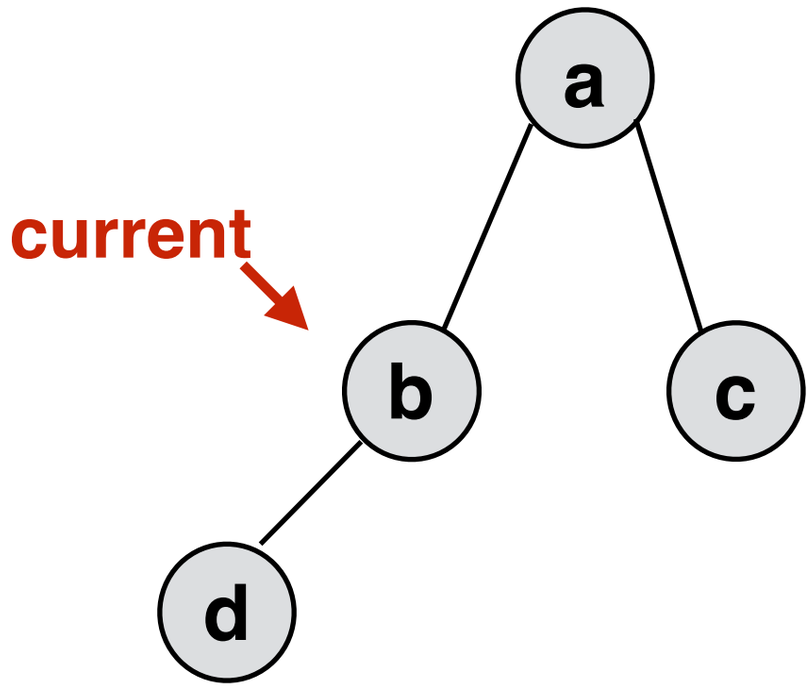
a

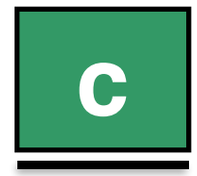




a

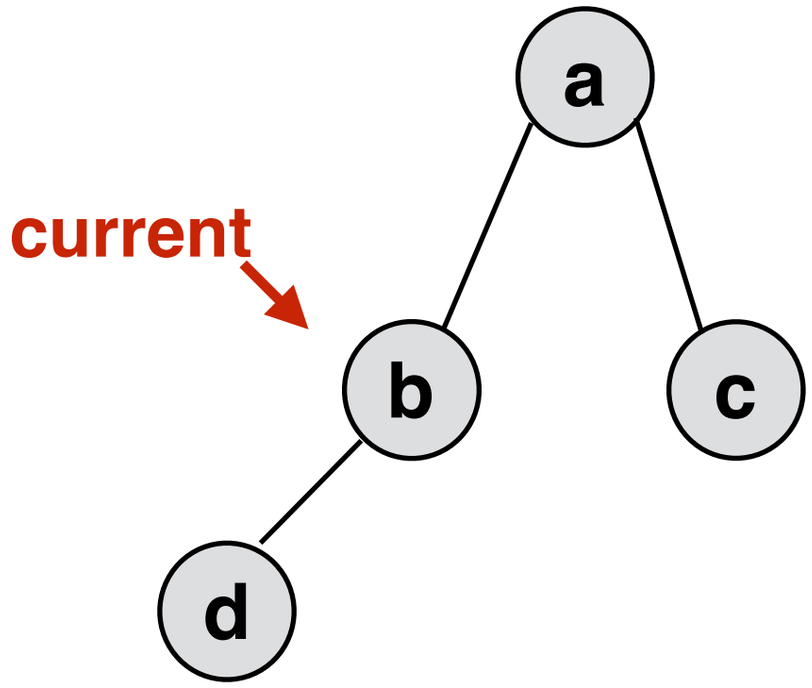
Nothing to push on right

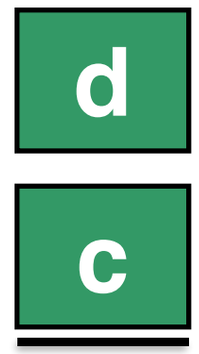




a

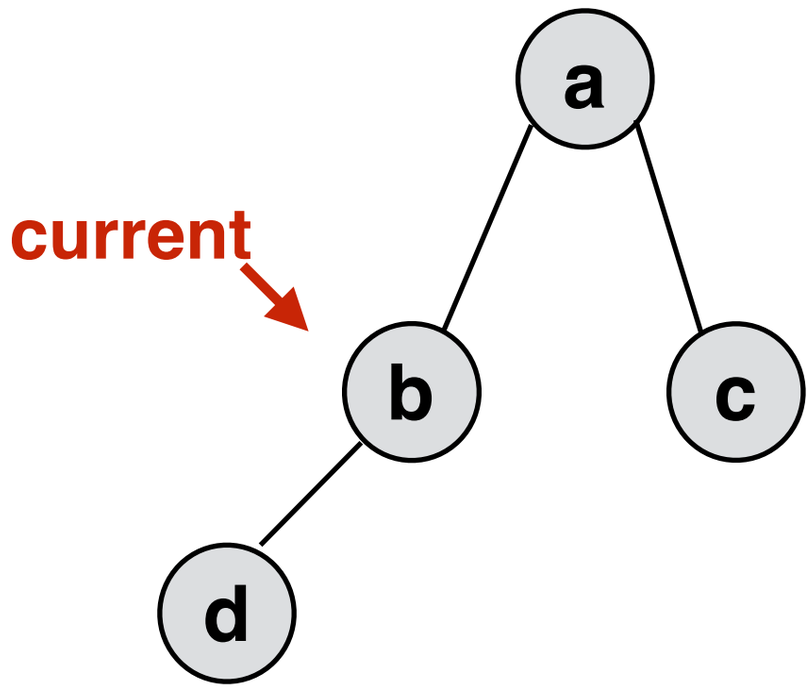
Push what is to the left of current.

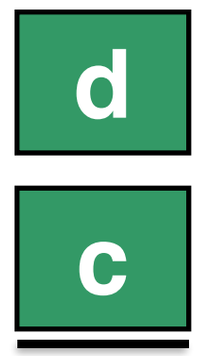




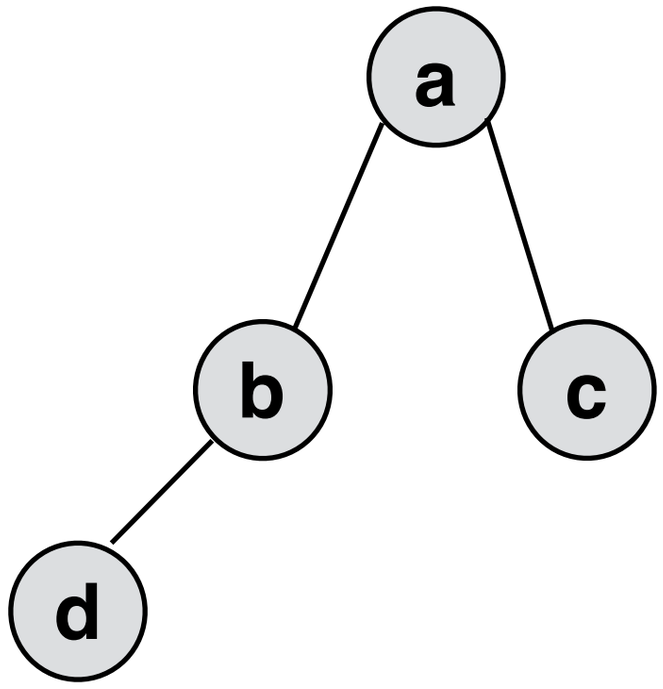
a

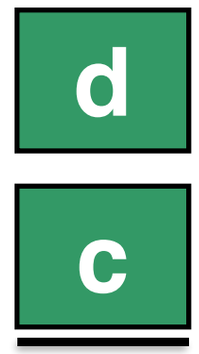
return current.item





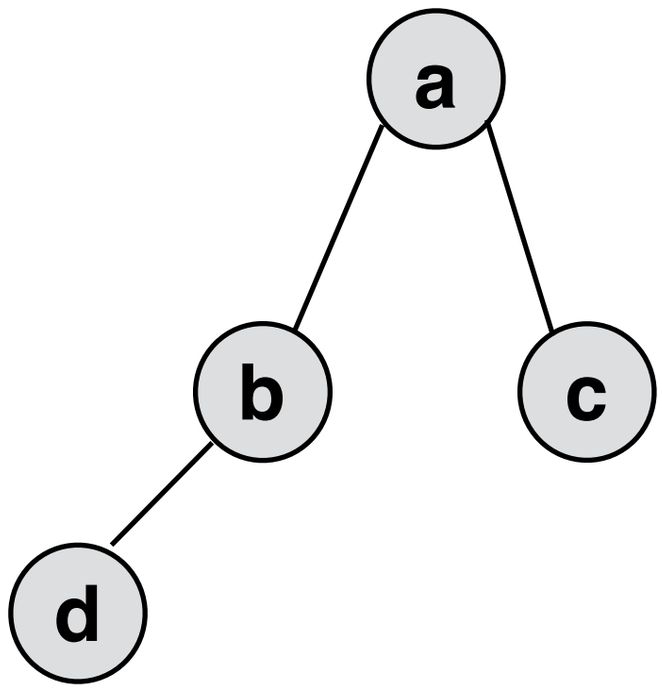
a b

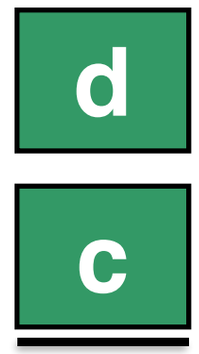




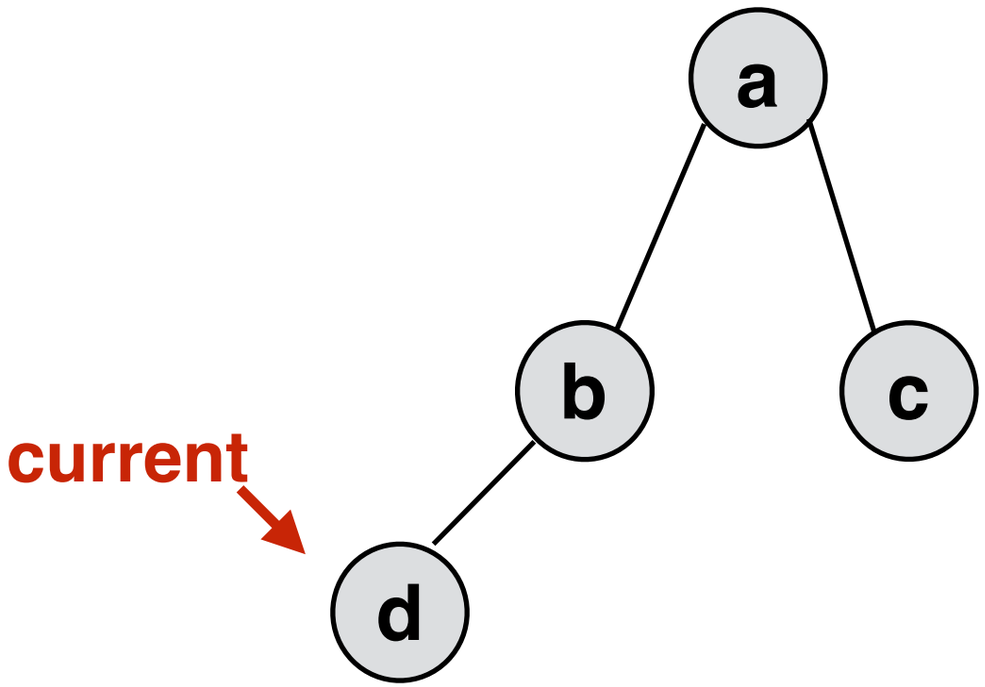
a b

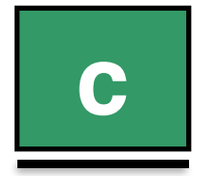
Next!



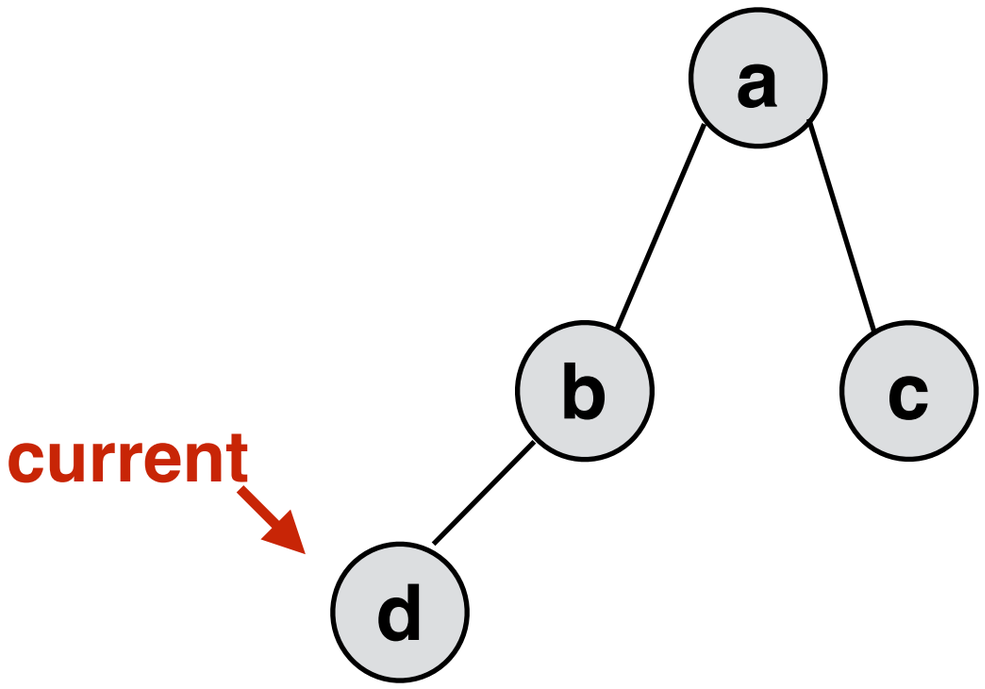


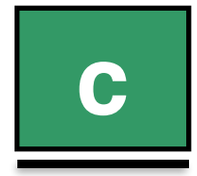
a b



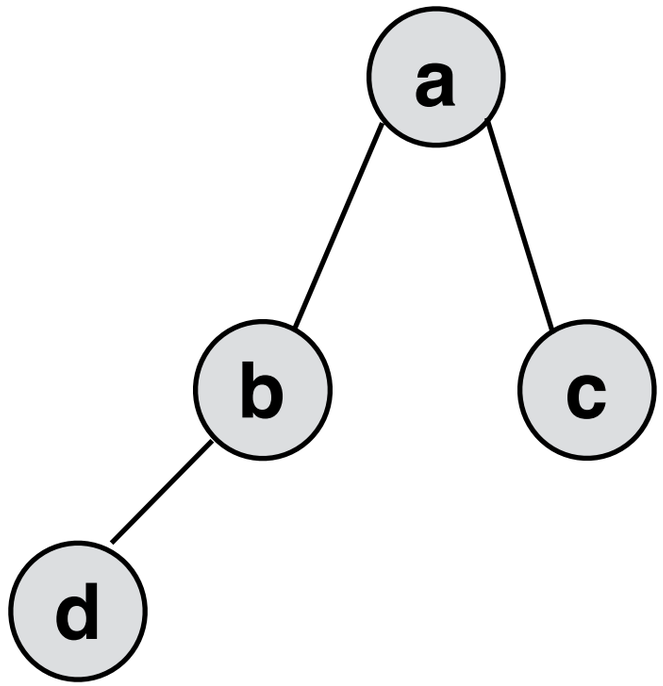


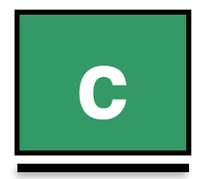
a b





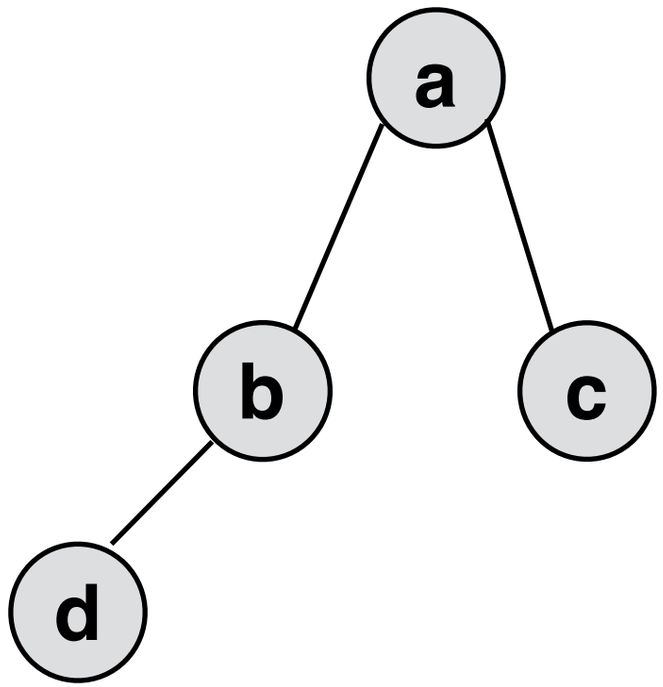
a b d

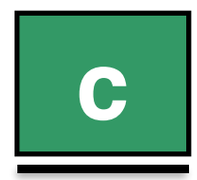




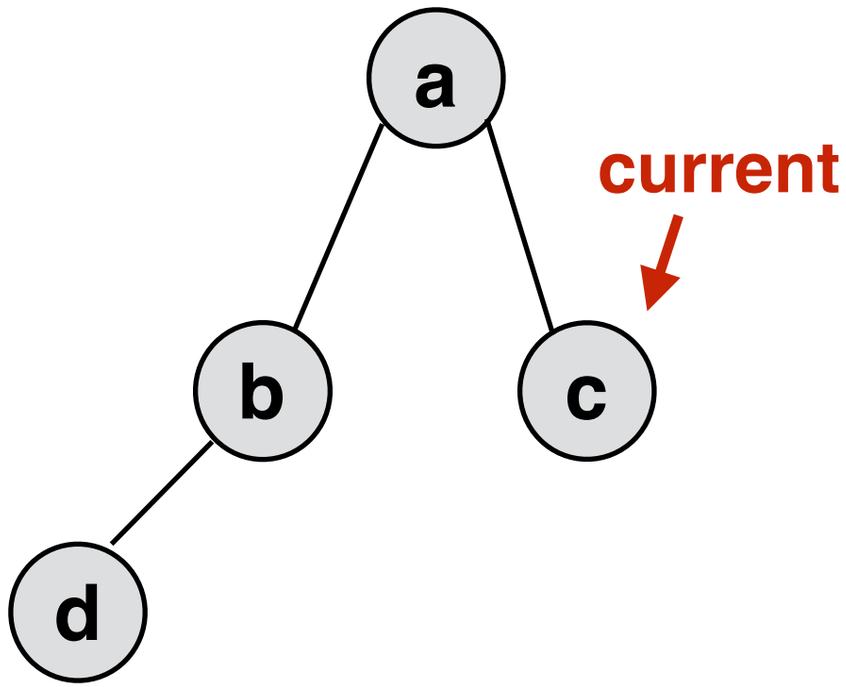
a b d

Next!



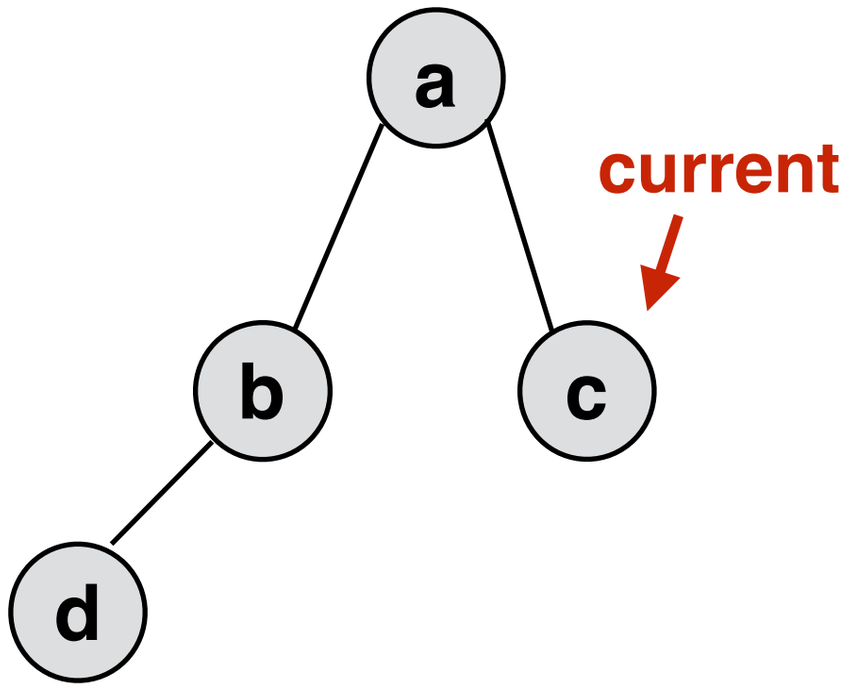


a b d

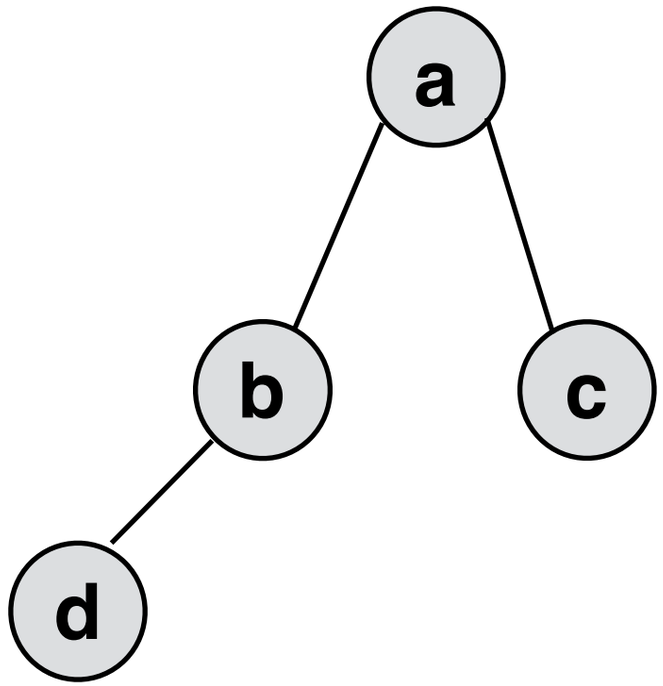


a b d

return current.item

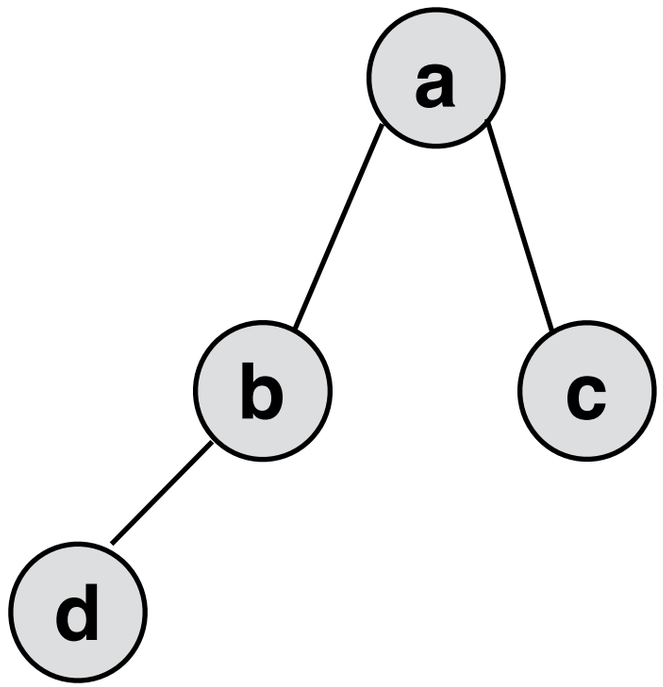


a b d c

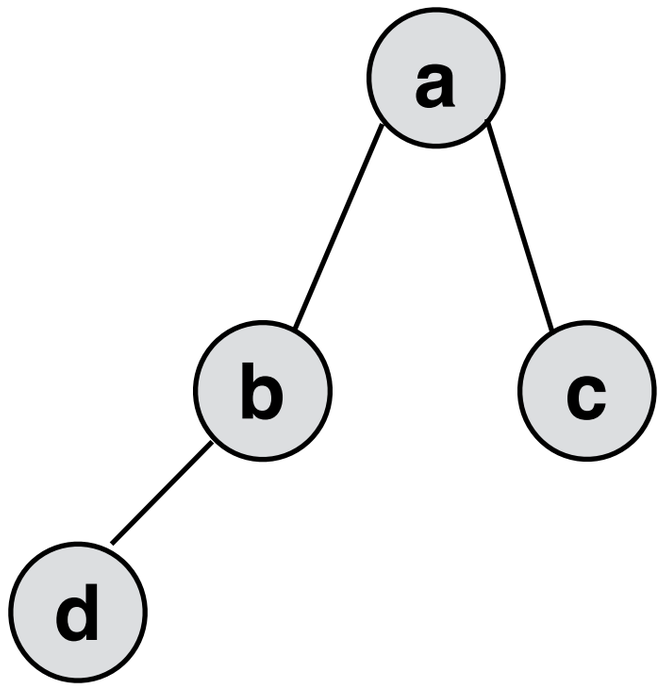


a b d c

Next!

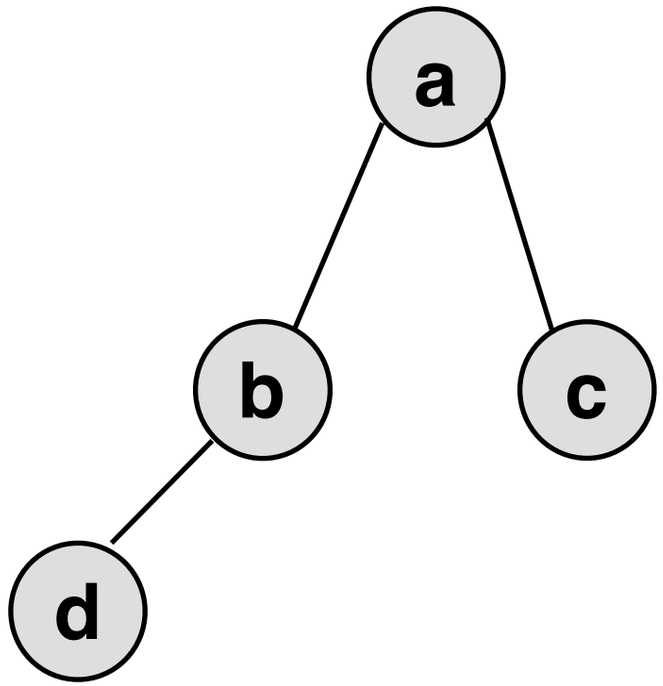


a b d c



StopIteration

a b d c



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)
```

What about without a stack?

hint: find out about python generators…
and **yield**

Summary

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