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

Lecture 32
Binary Trees

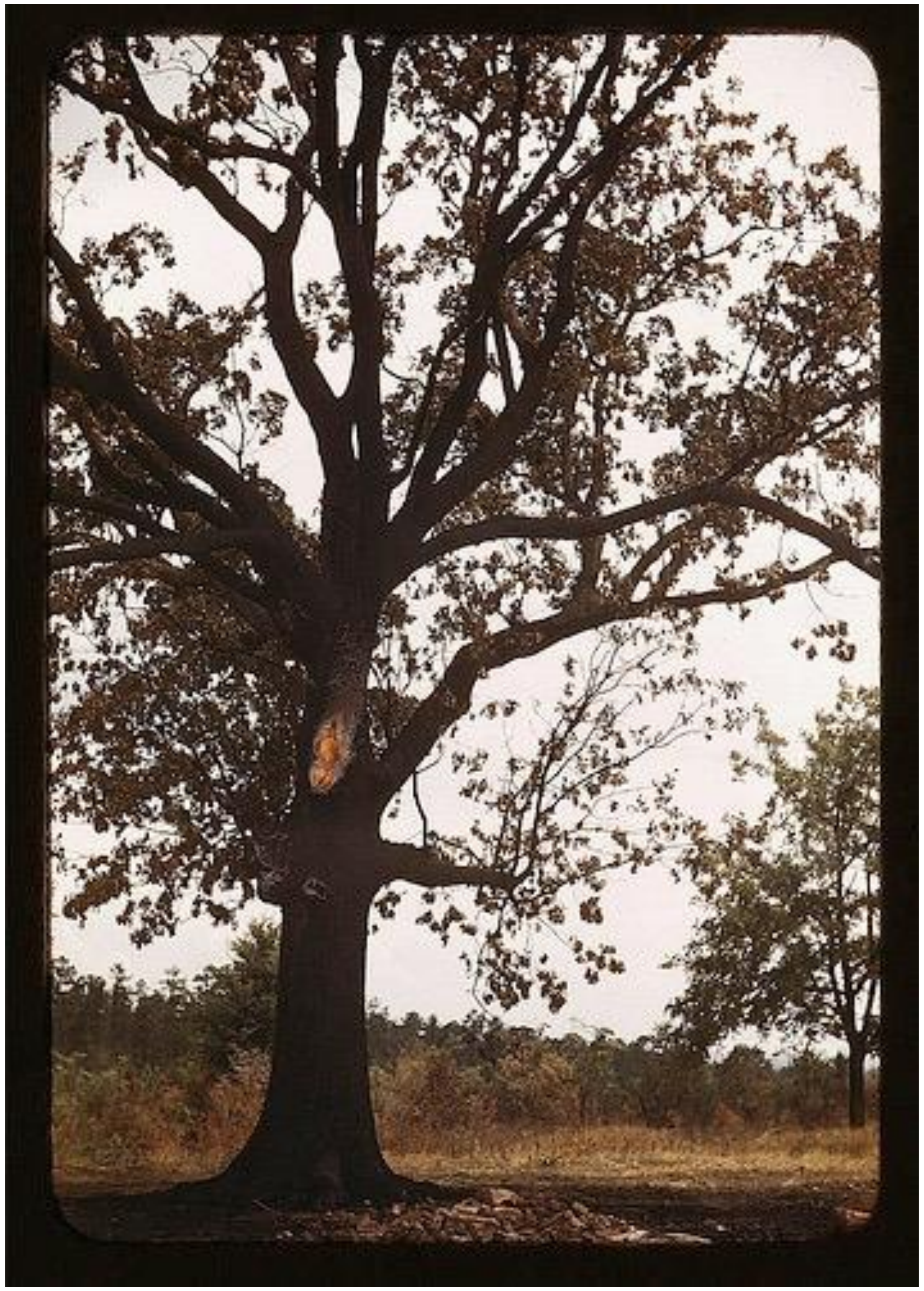
FIT 1008
Introduction to Computer Science

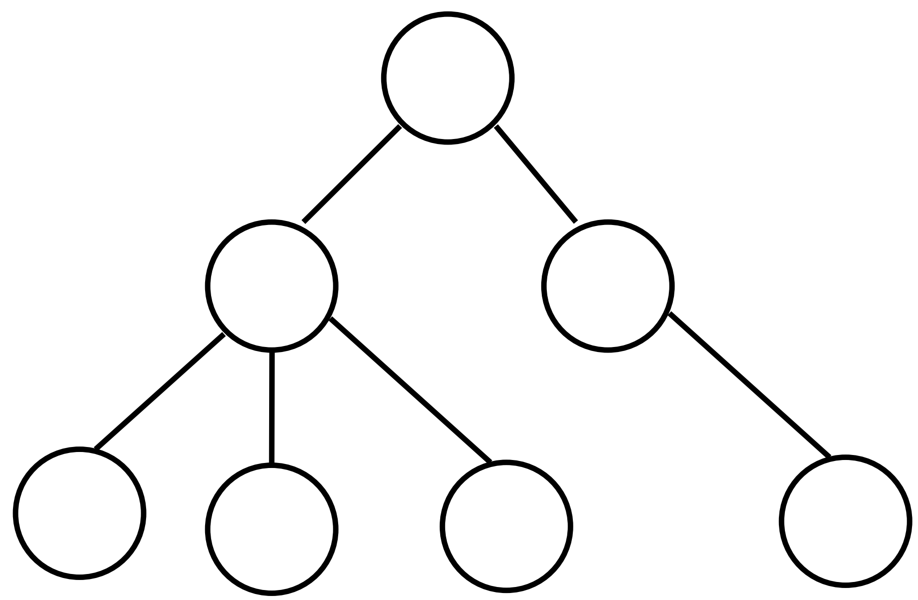
MONASH University
Information Technology

Objectives

- Revise Trees:
  - ➞ Concepts
  - ➞ Operations & Implementation
  - ➞ Complexity Ideas
  - ➞ Traversal

Trees

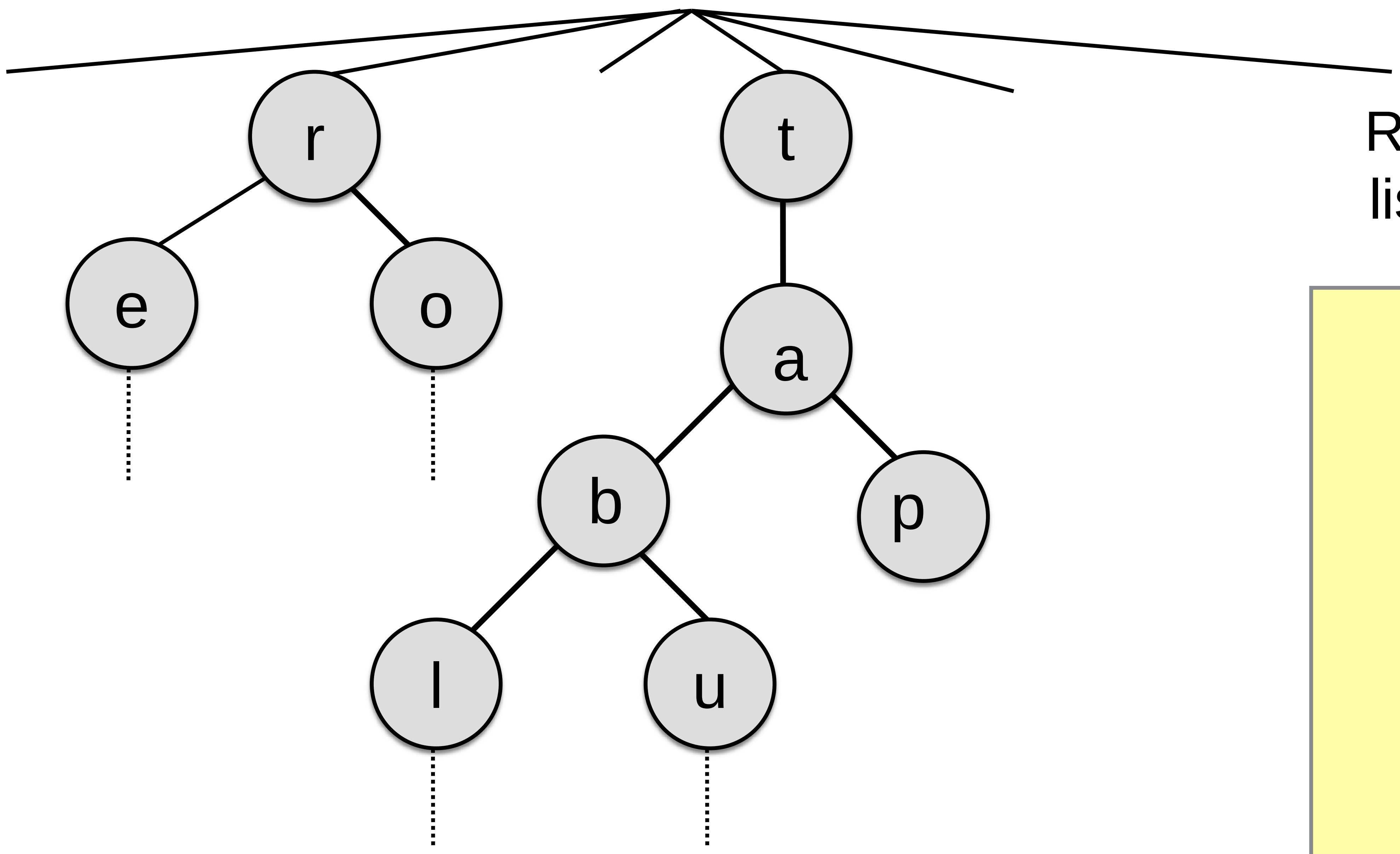




Trees

- 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

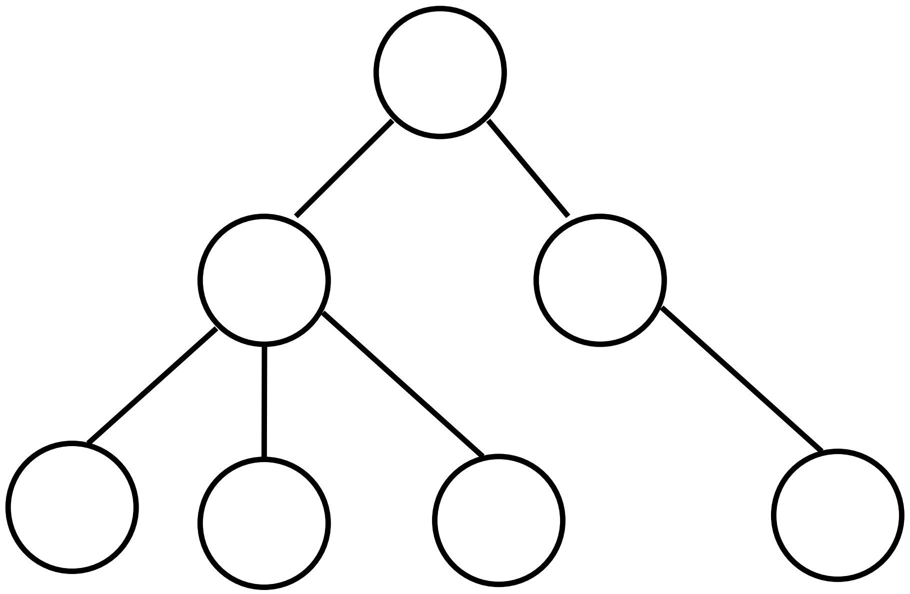
- table
- tap
- tabular
- rest
- restaurant
- remain
- route

Branches represent different strings.

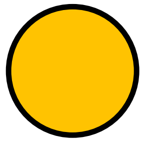
Trees

- Graphs which are:
  - Simple

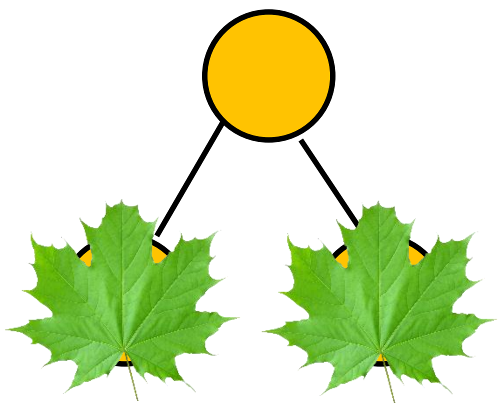
    no loops or multiple edges
  - Connected
  - No circuits.



Perfect Binary Trees



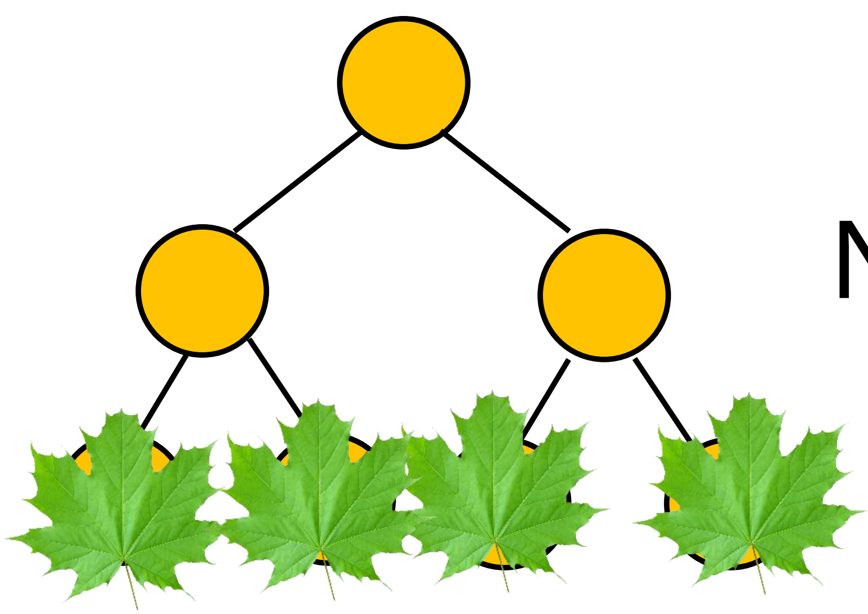
N =1 Height = 0



N =3 Height = 1

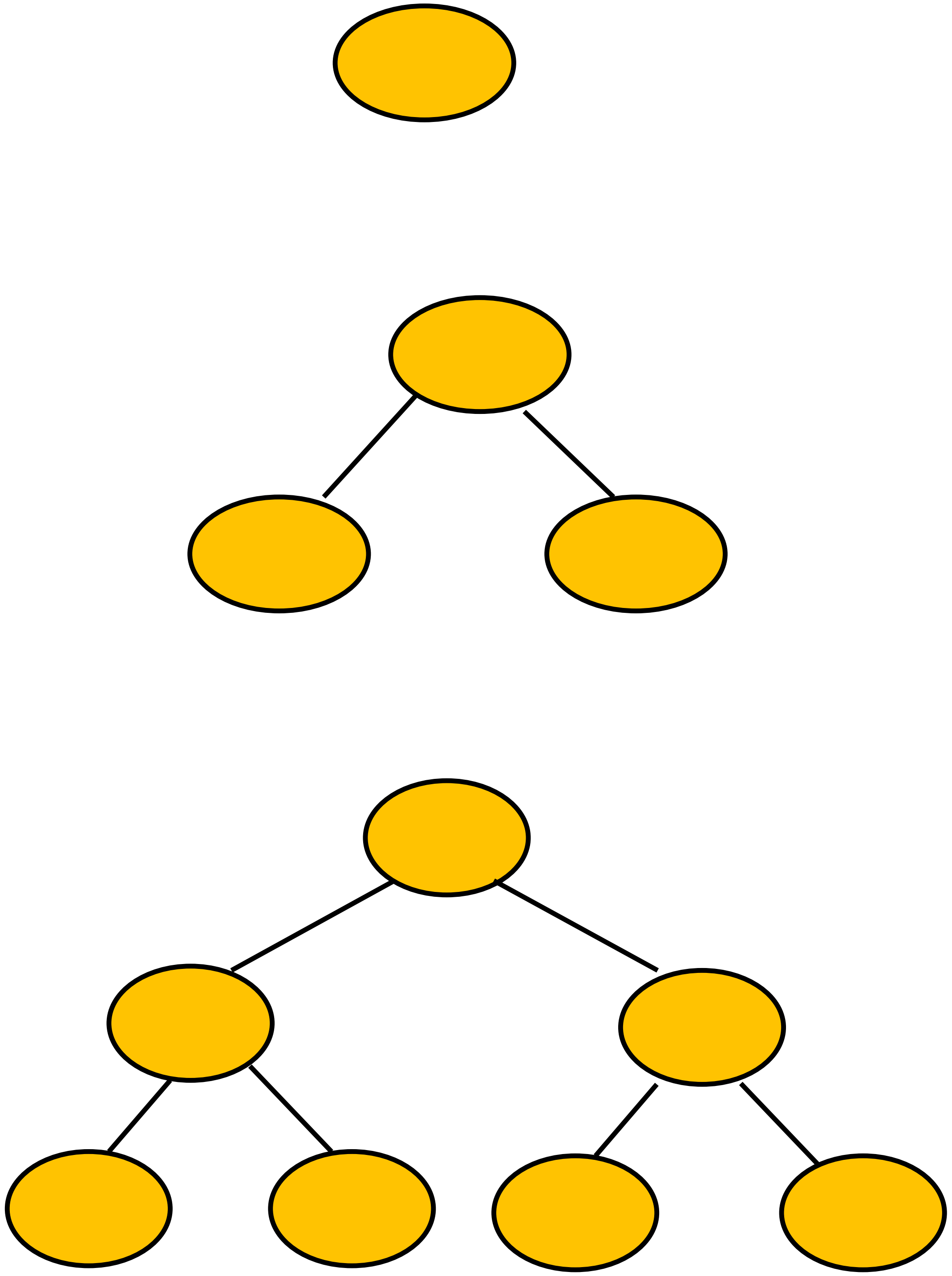
Each parent has two children

All leaves at same level



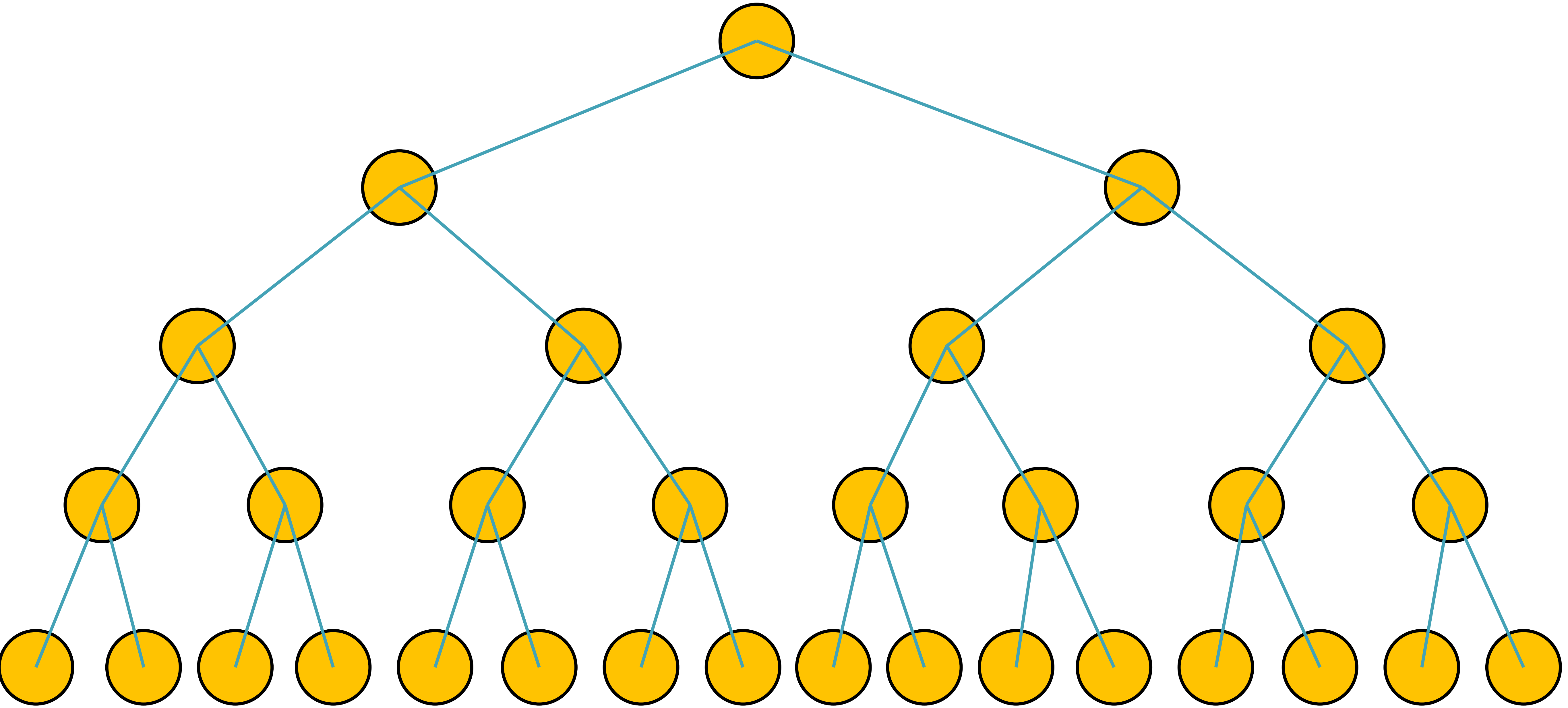
N =7 Height = 2

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

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

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

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

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

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

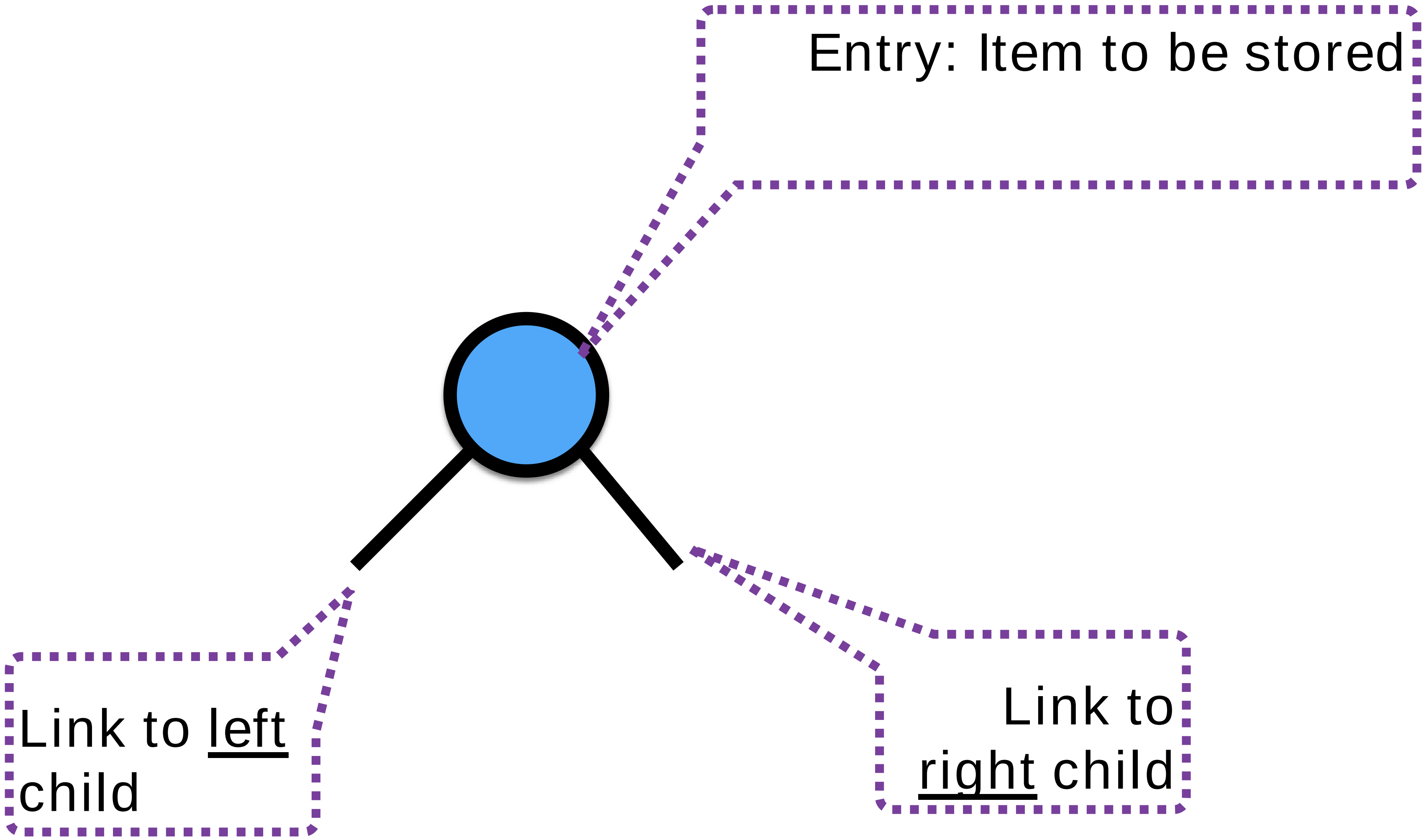
Balanced tree

the height is **O(logN)**

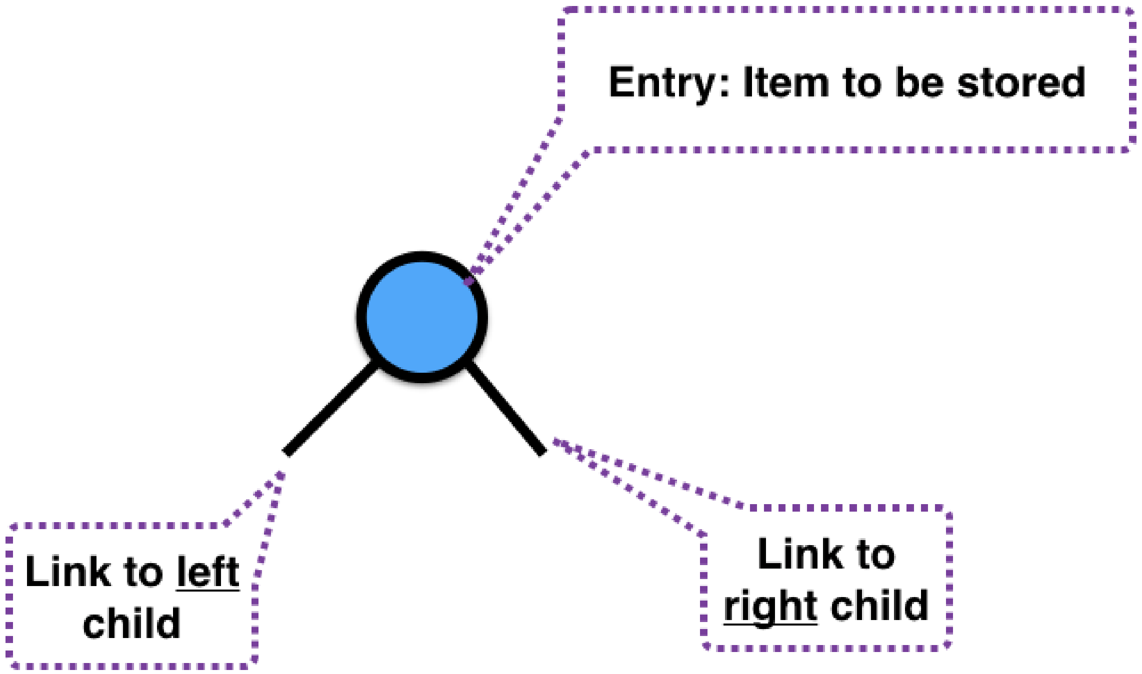
Unbalanced tree

the height is **O(N)**

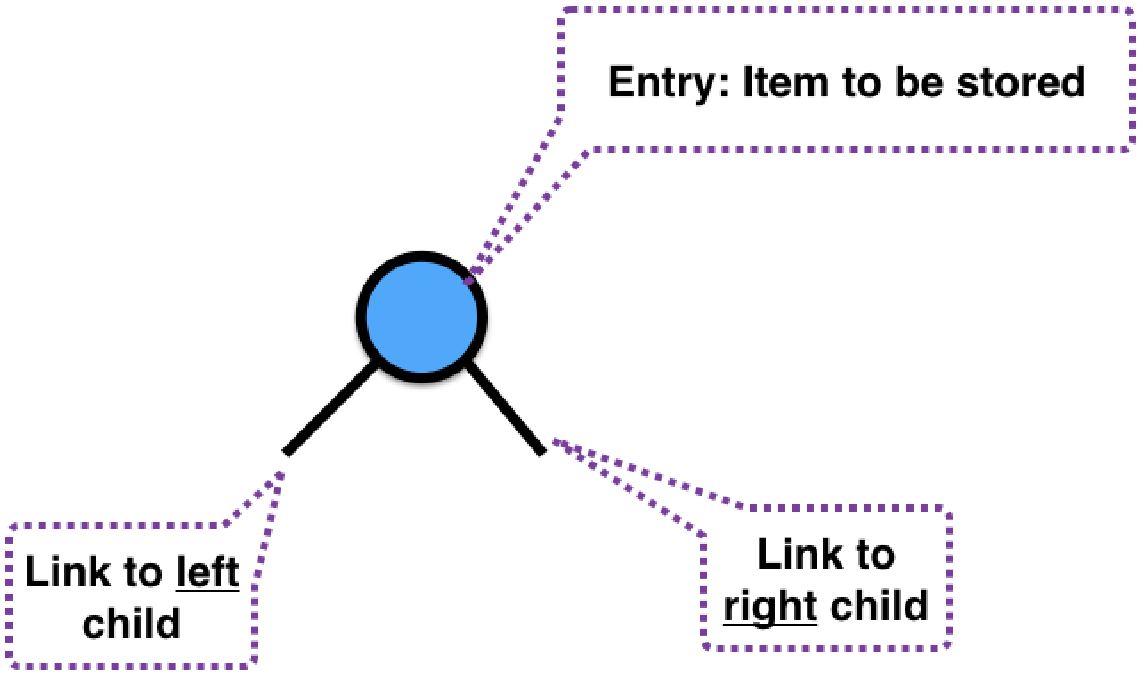
Representing a Binary Tree Node



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




```
class TreeNode:
```

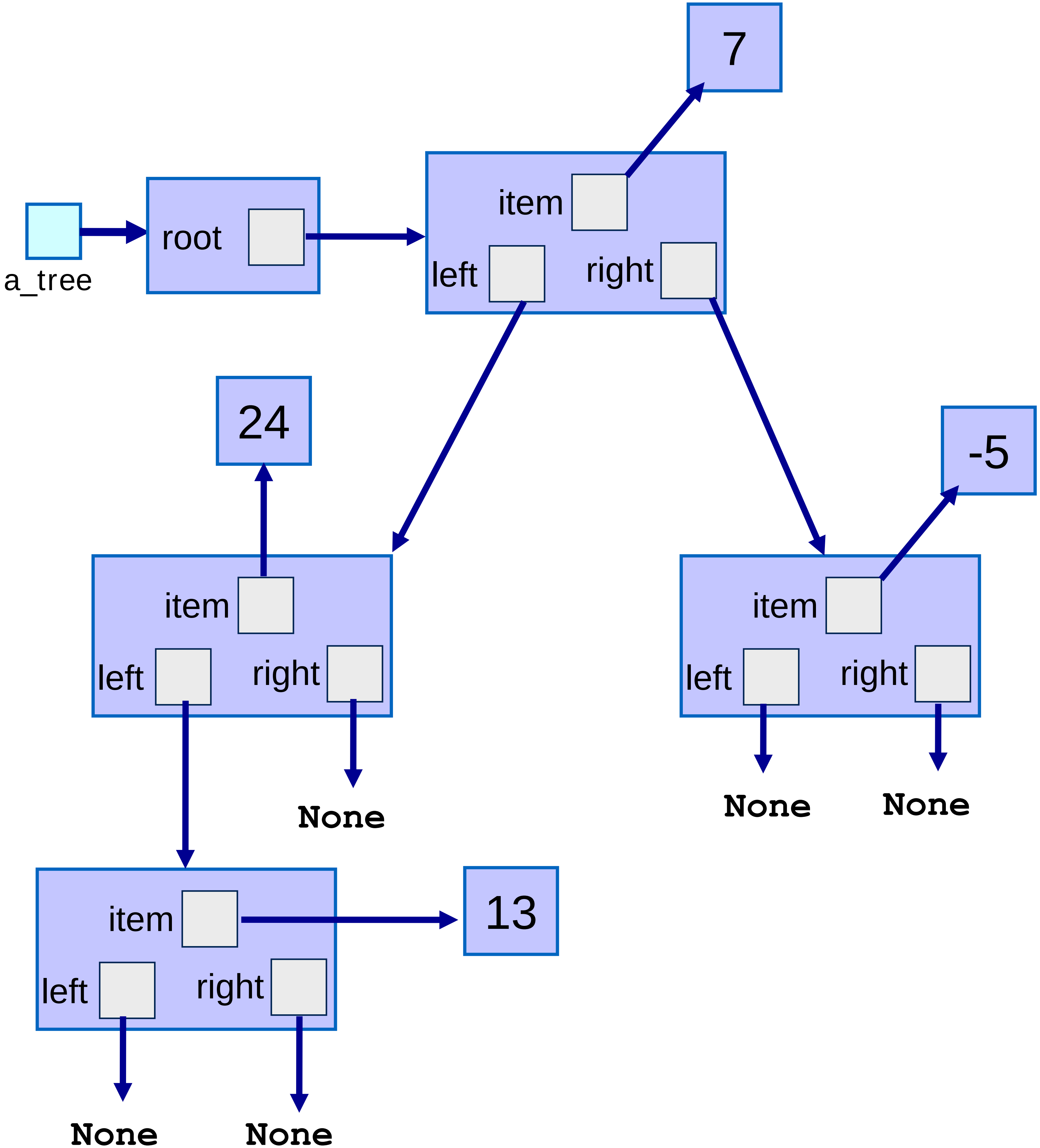
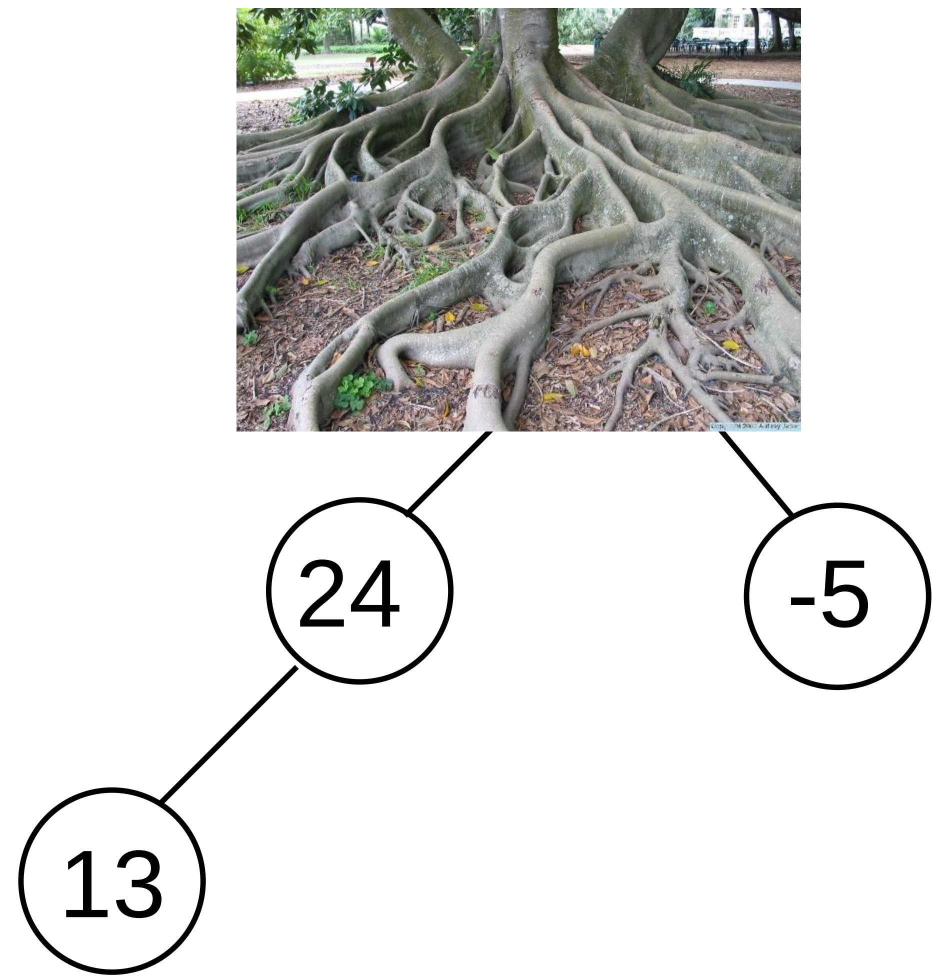



```python
class TreeNode:

    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 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)
```

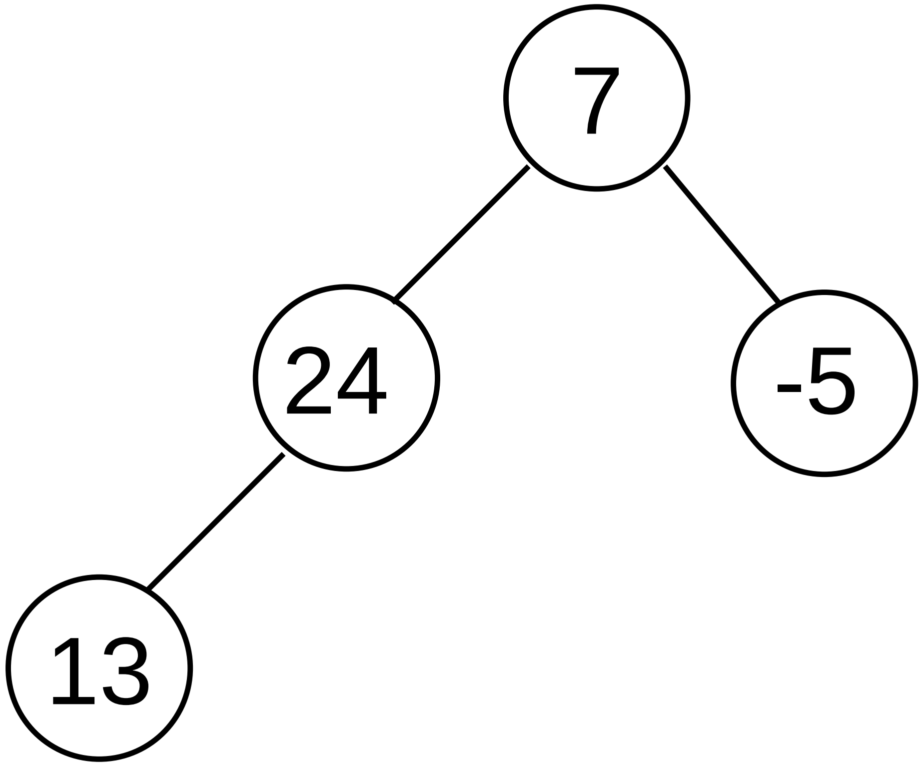
```python
class BinaryTree:
    def __init__(self):
        self.root = None

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

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


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

24
-5
13

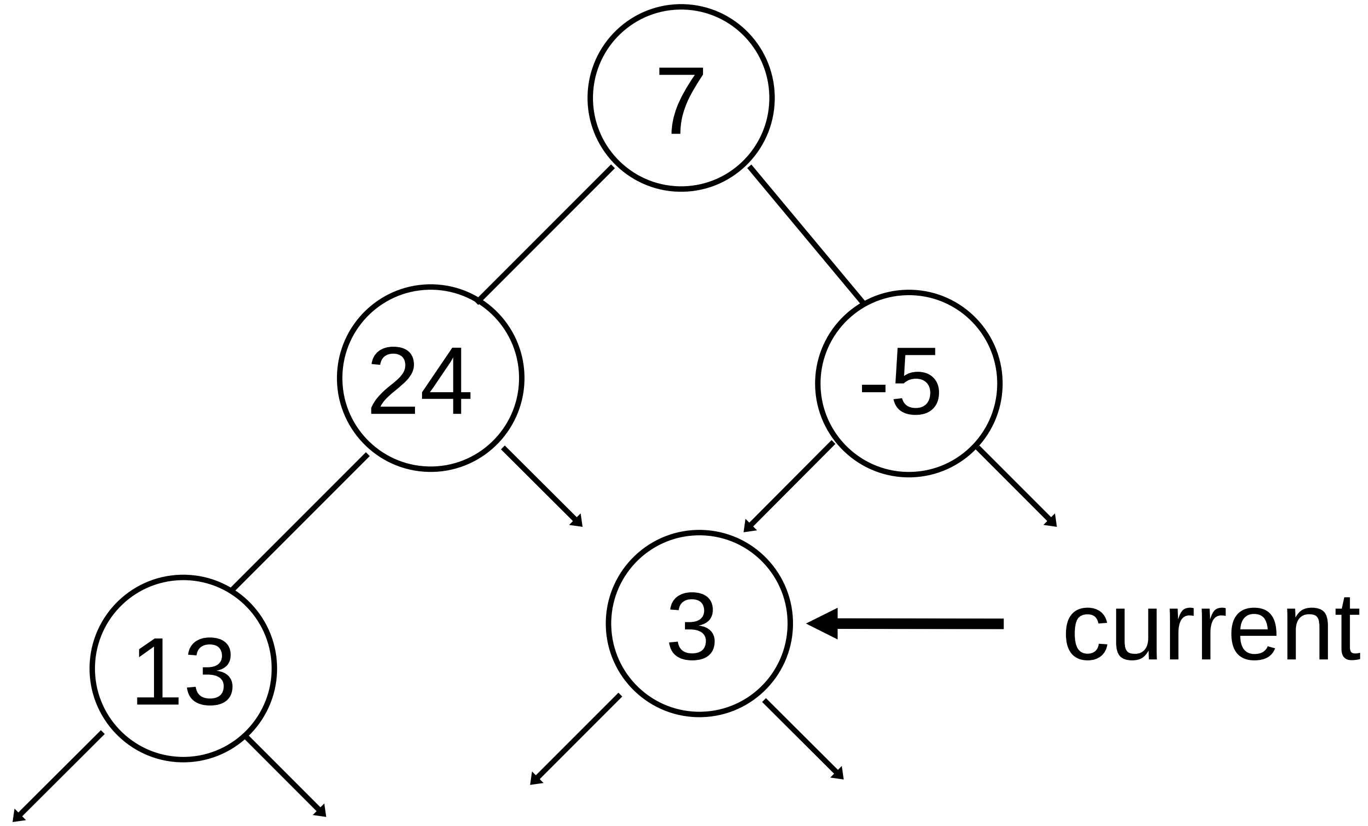
Add an item.

Add 3



where?

Add 3



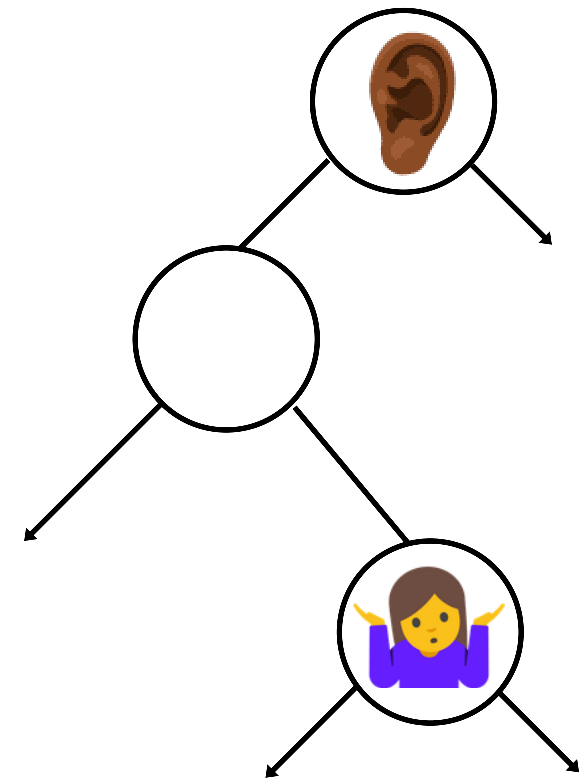
bitstring = “10”, item= 3

Examples

bitstring = “”, item=👂🏾

bitstring = “01”, item=🤷‍♀️

bitstring = “ ”, item=💵

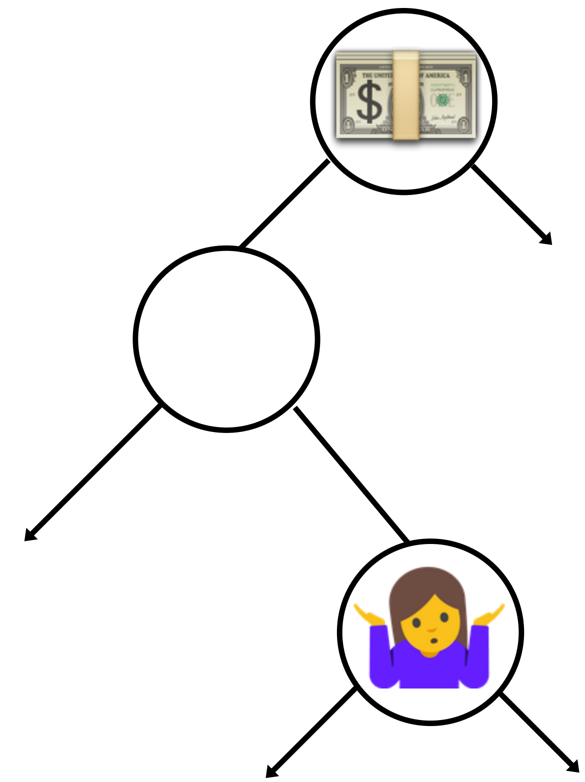


Examples

bitstring = “”, item=👂🏾

bitstring = “01”, item=🤷‍♀️

bitstring = “ ”, item=💵

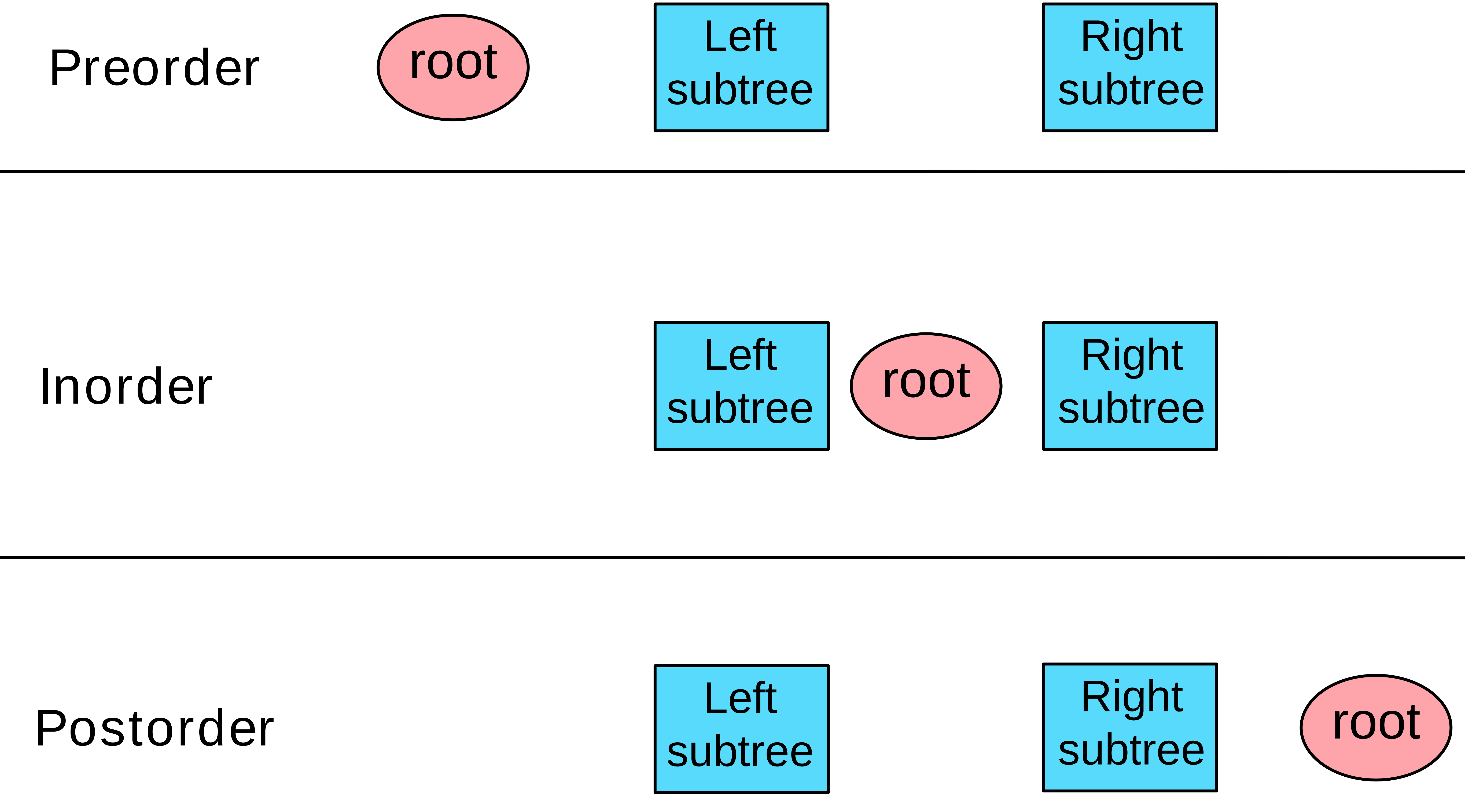


Recursively explore subtree
following “bitstring directions”

```python
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
root
Left subtree
Right subtree
Inorder
Left subtree
root
Right subtree
Postorder
Left subtree
Right subtree
root

Print Preorder Traversal

1) Print the root node

2) Traverse the left subtree

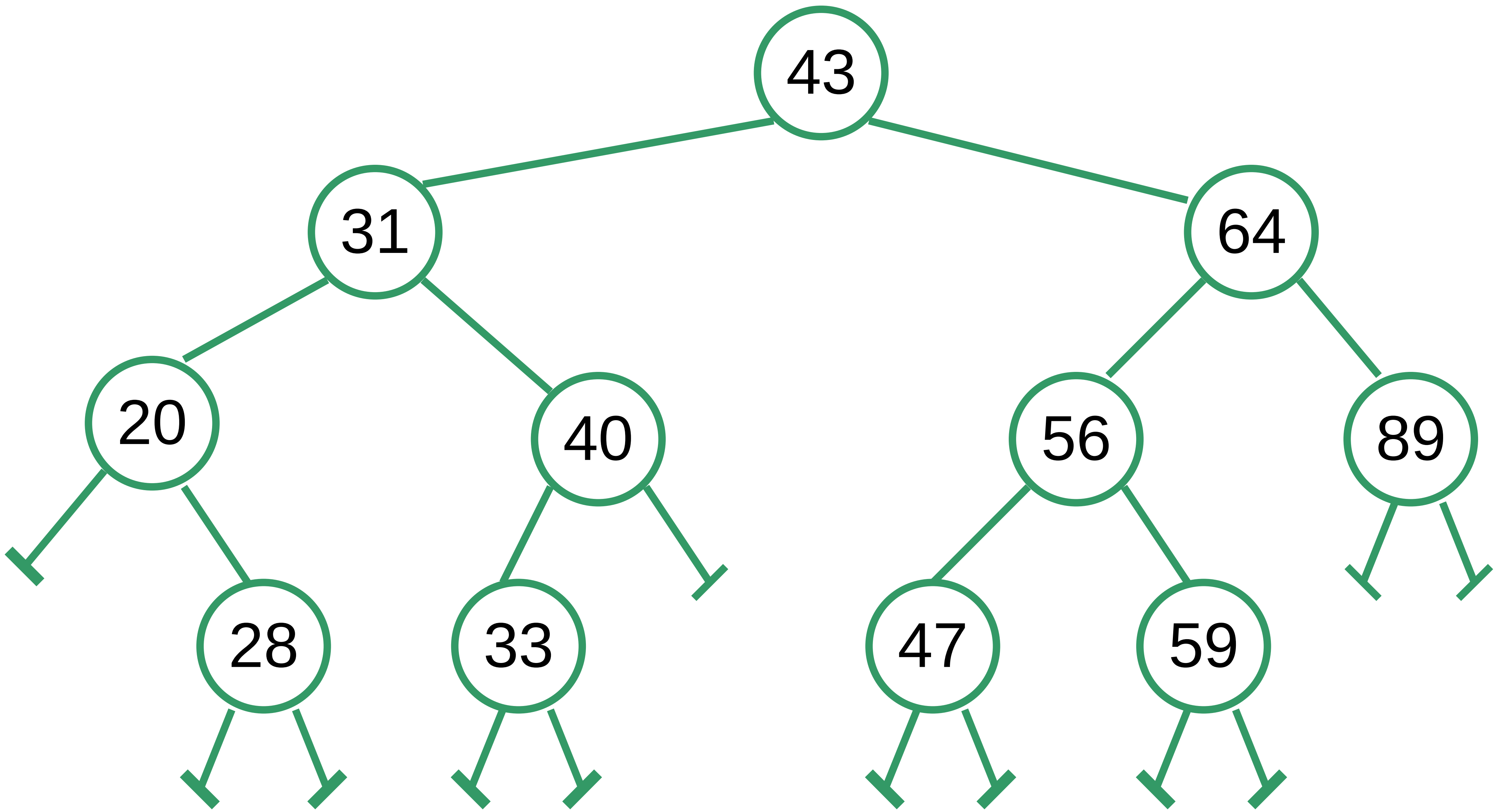
3) Traverse the right subtree

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

Print Preorder Traversal

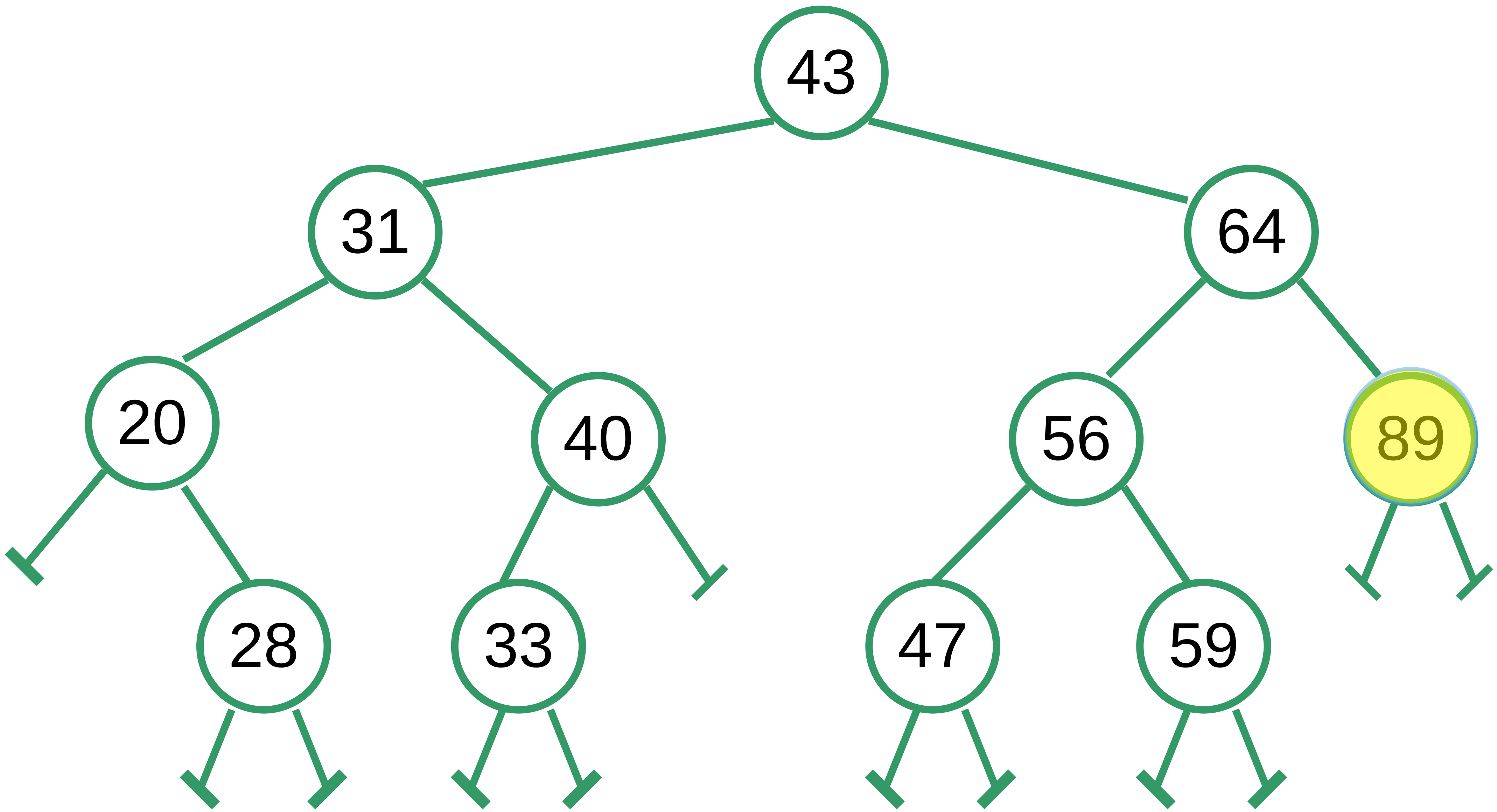
```python
def print_preorder(self):
    self._print_preorder_aux(self.root)

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: Preorder



43	31	20	28	40	33	64	56	47	59	89

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

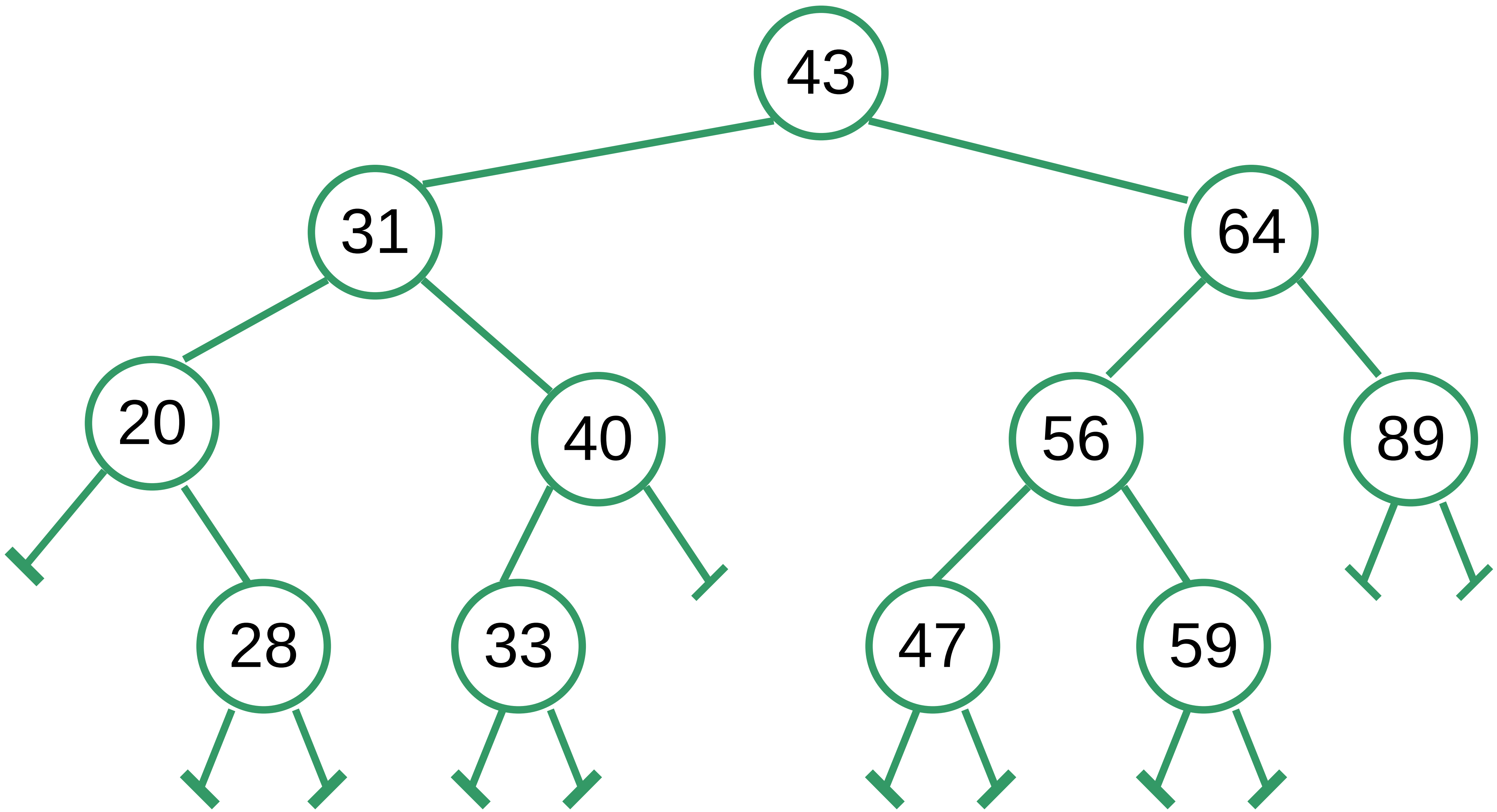
```
def print_inorder(self):
    self._print_inorder_aux(self.root)

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)
```

Print Post-order Traversal

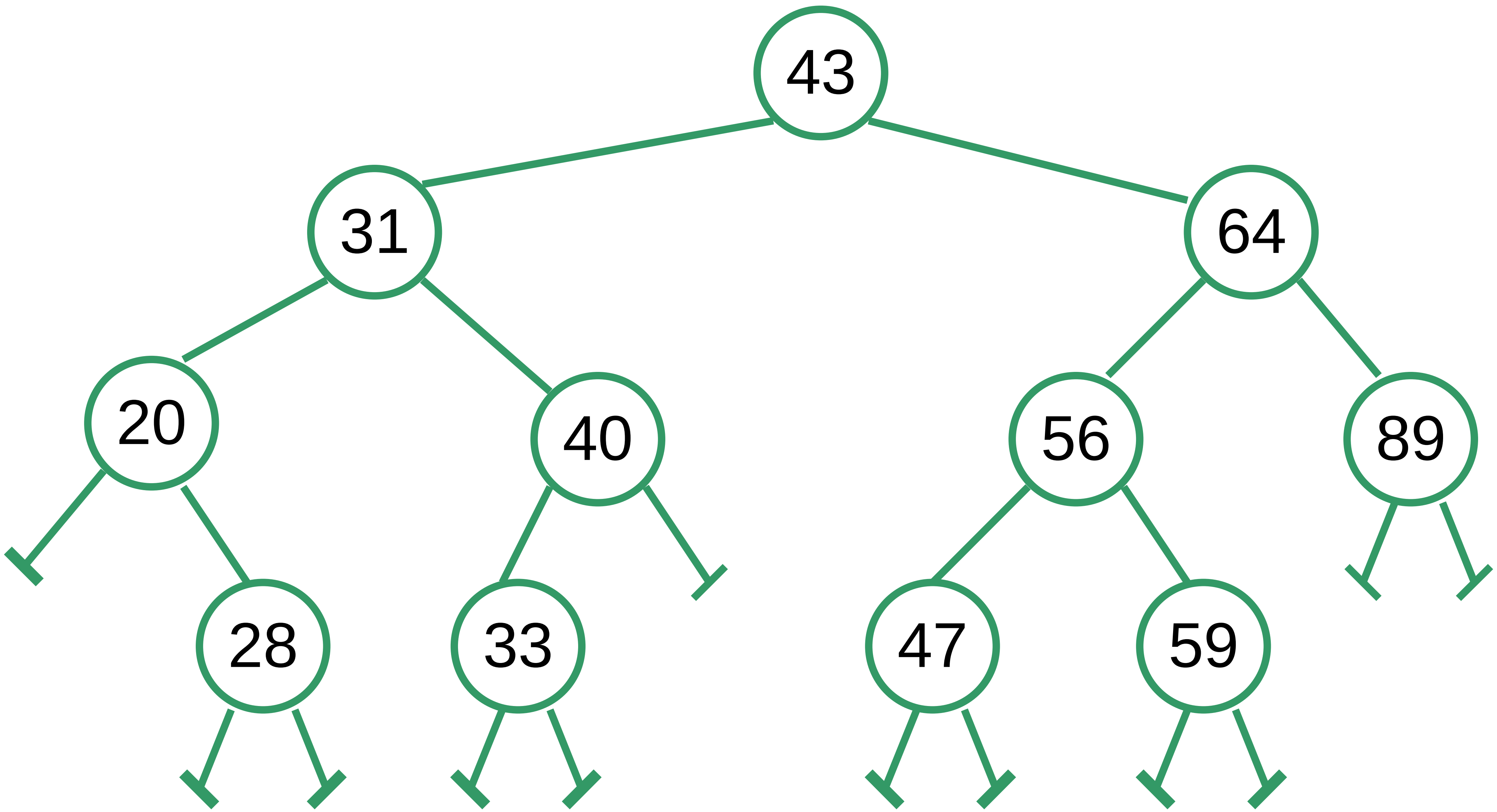
1) Traverse the left subtree
2) Traverse the right subtree
3) Print the root node

```
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)
```

Example: Postorder

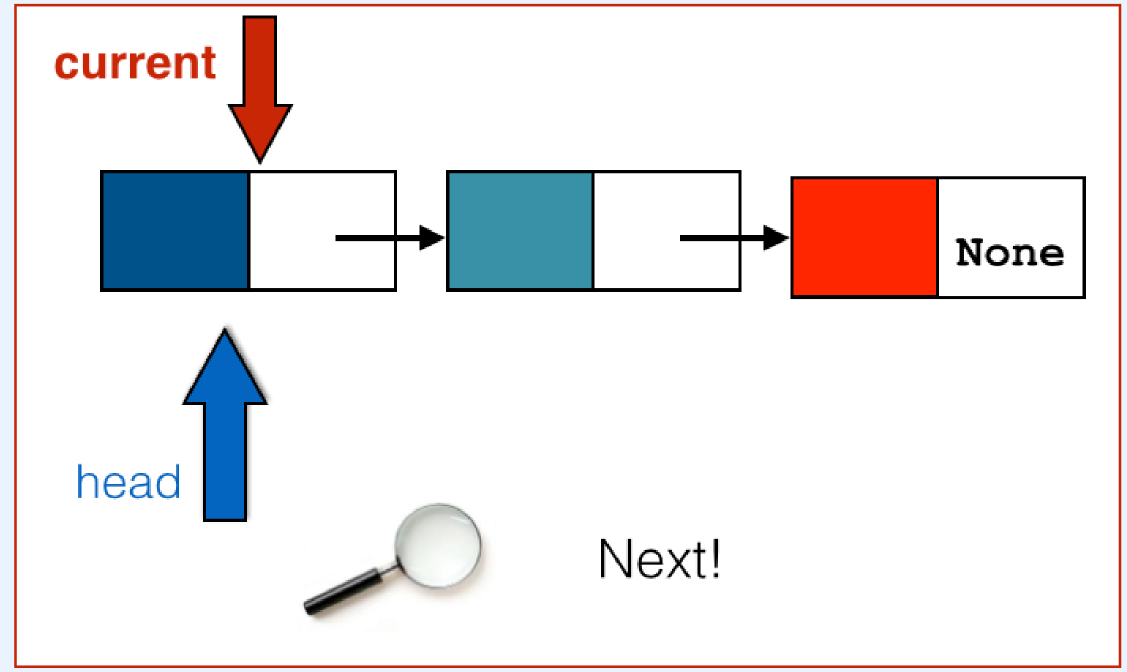


Example: Postorder

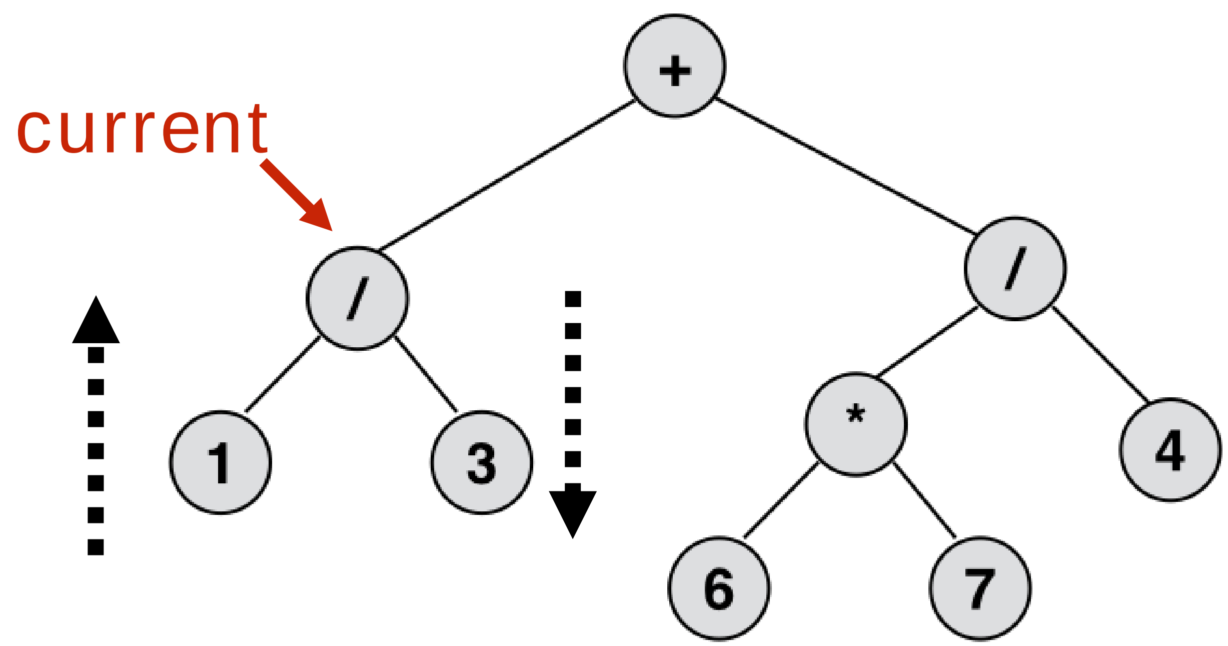


28	20	33	40	31	47	59	56	89	64	43

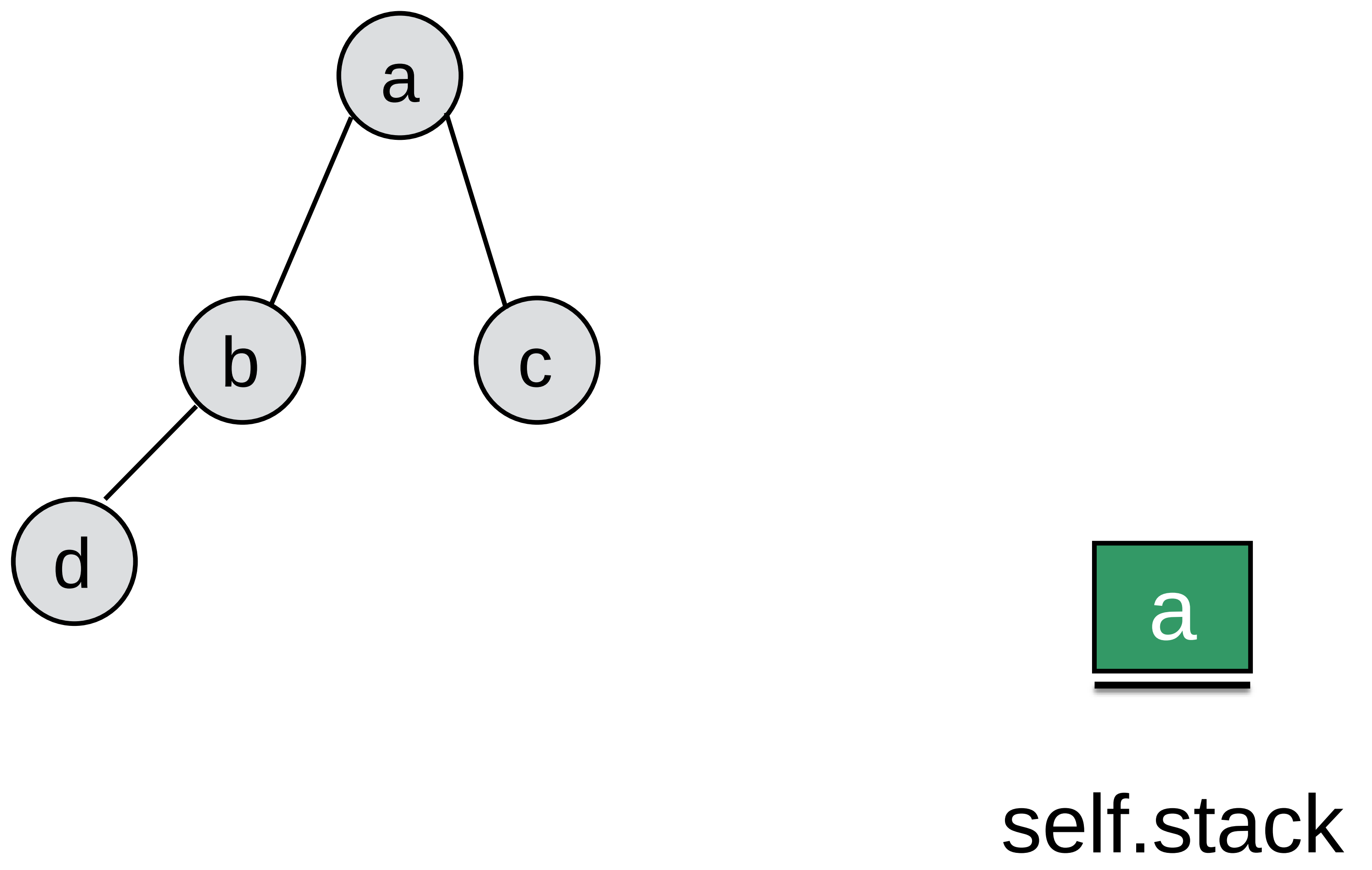
```python
class ListIterator:
    def __init__(self,head):
        self.current = head

    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
```

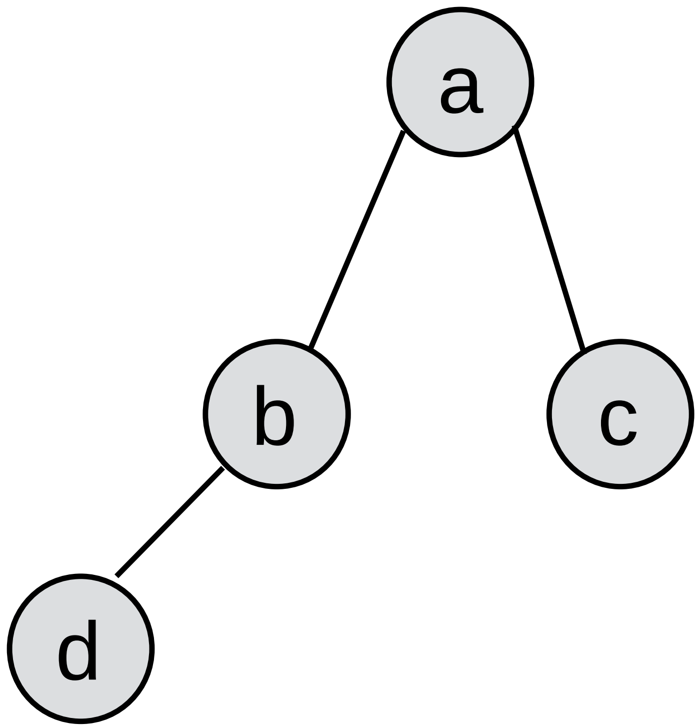


?



State of the Iterator on creation





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
```

```python
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:

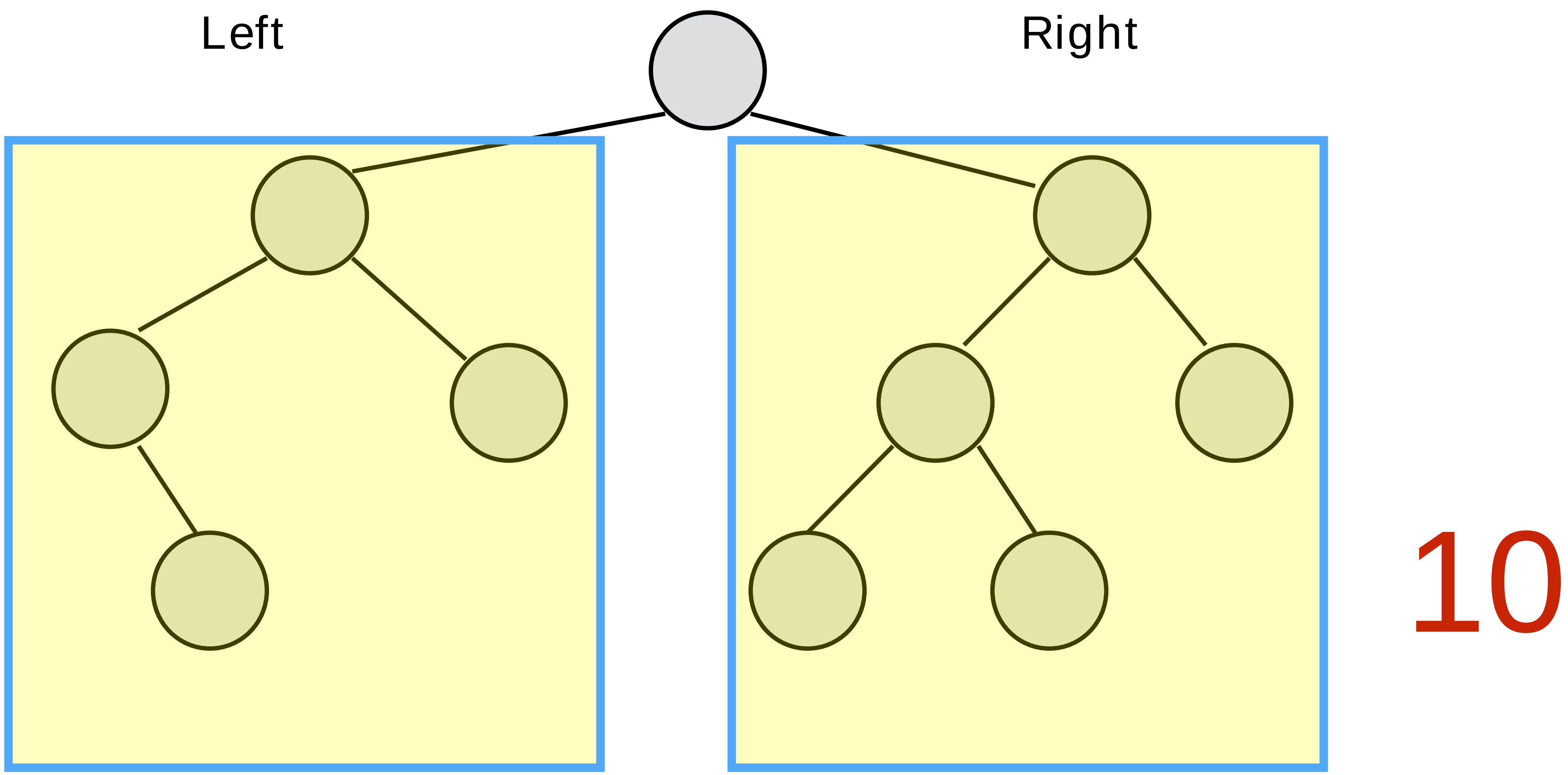
```
def __iter__(self):
    return PreOrderIteratorStack(self.root)
```

What about without a stack?

hint: find out about python generators…
and yield

Computing the size of a tree

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



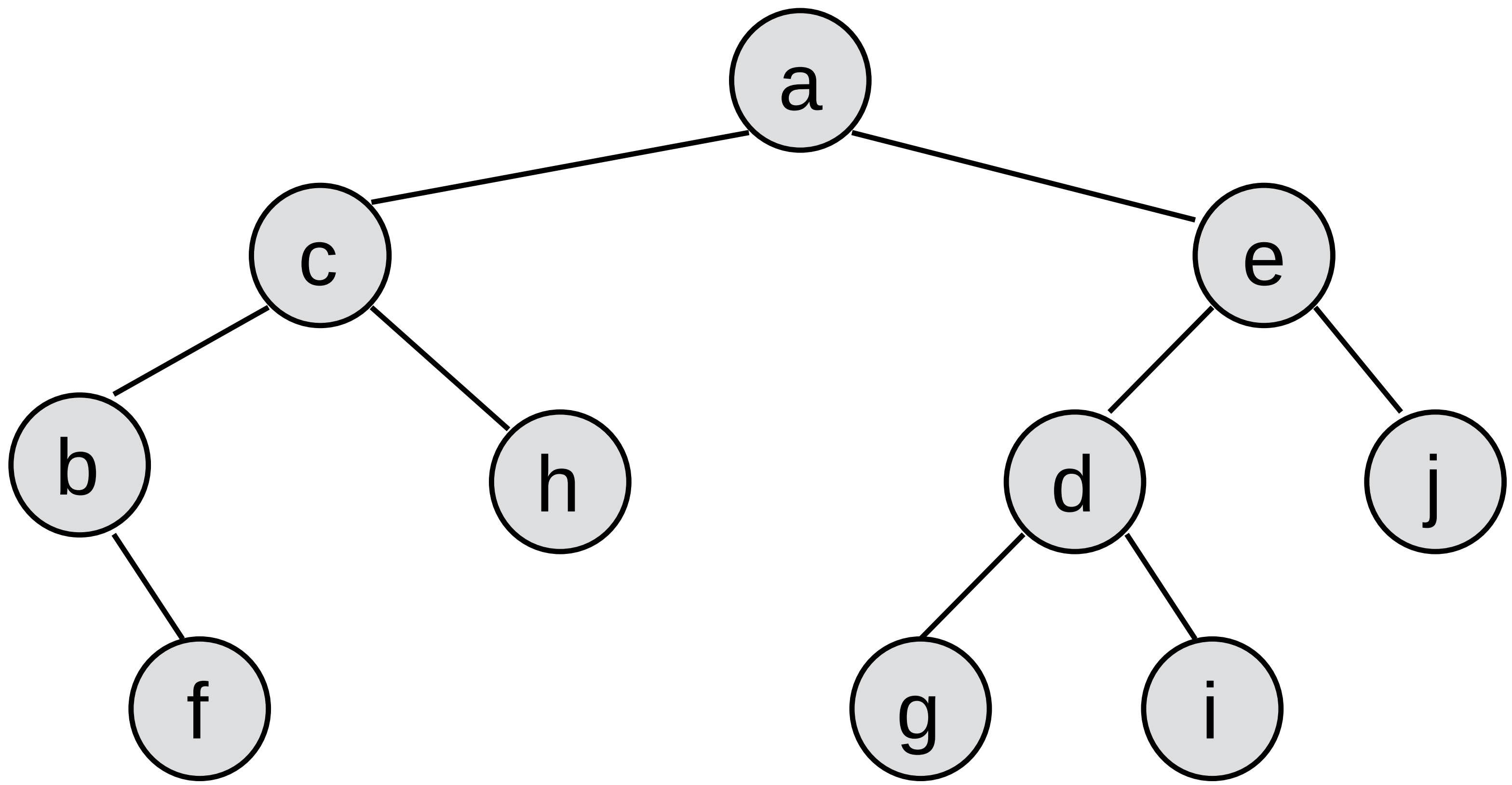

```
size(self)  = size(left) + 1 + size(right)
```

Computing the size of a tree

```python
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
- Expression trees: prefix, infix, postfix