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

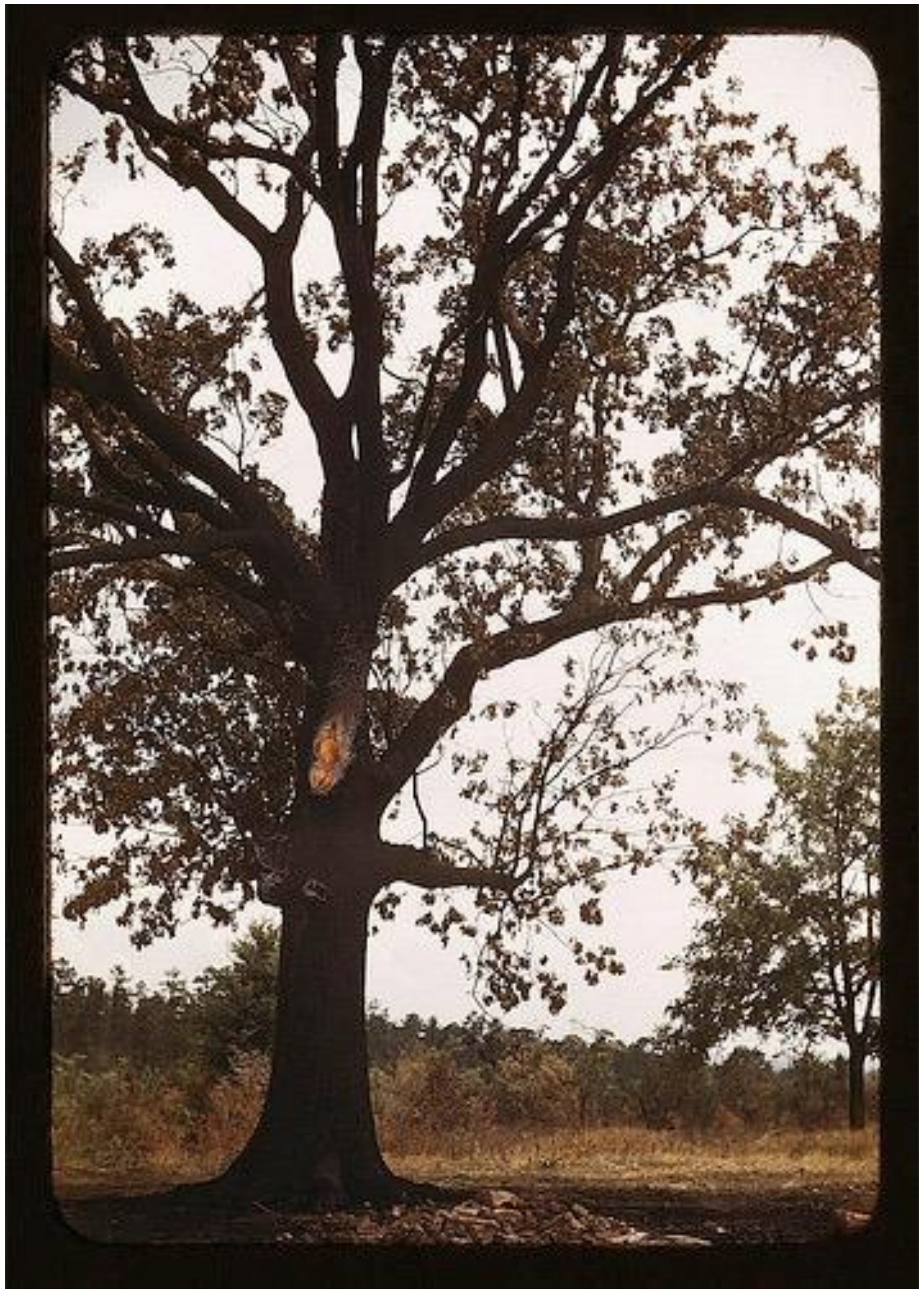
Lecture 30
Binary Trees

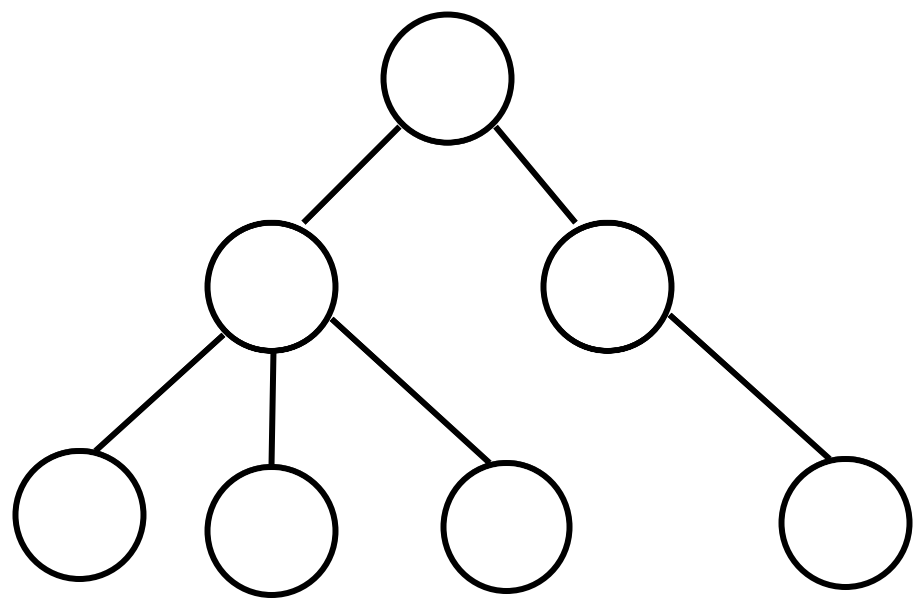
FIT 1008&2085
Introduction to Computer Science

Objectives

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

Trees

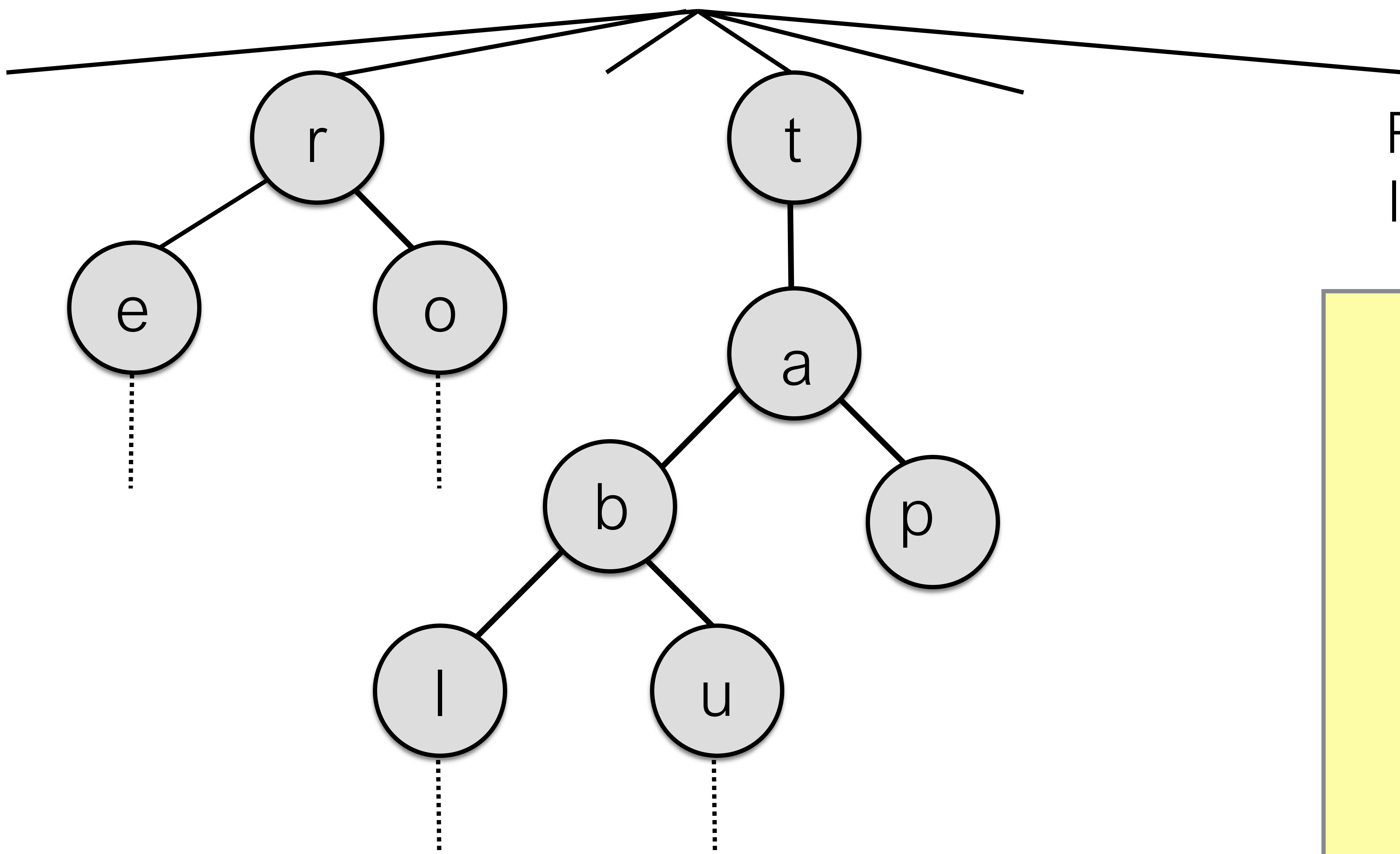




Trees

- 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



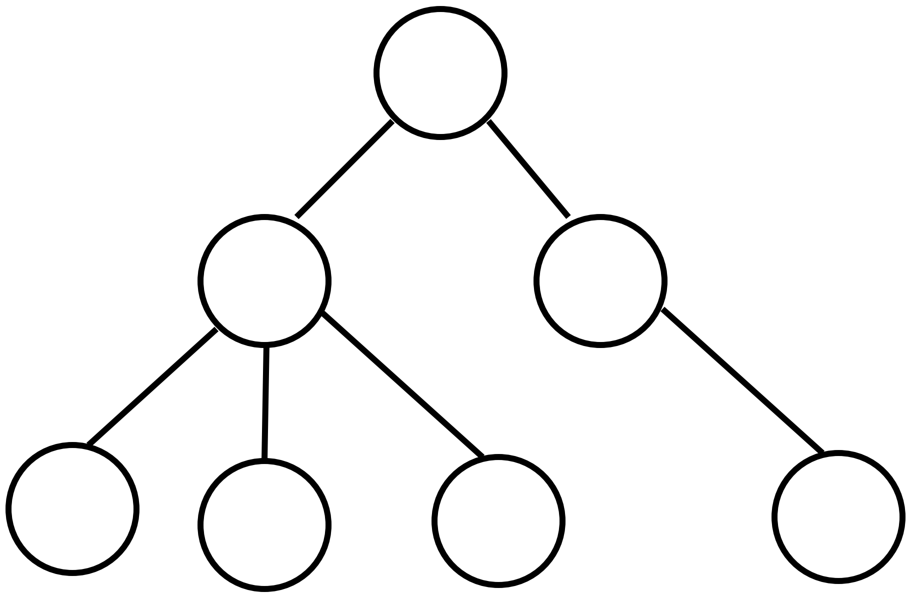
Represent a list of strings

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

Branches represent different strings.

Trees

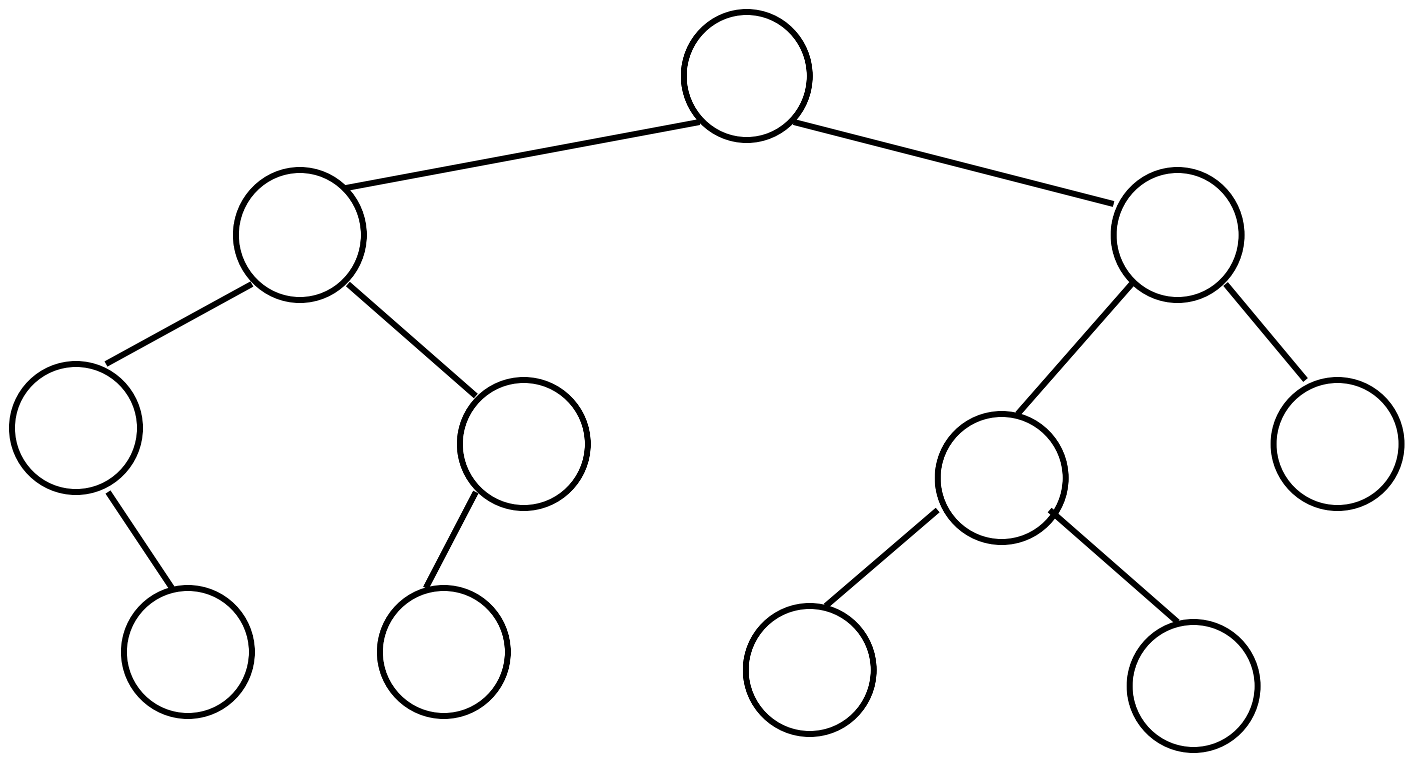
- Graphs which are:
  - Connected
  - No circuits.



We will only talk about binary trees

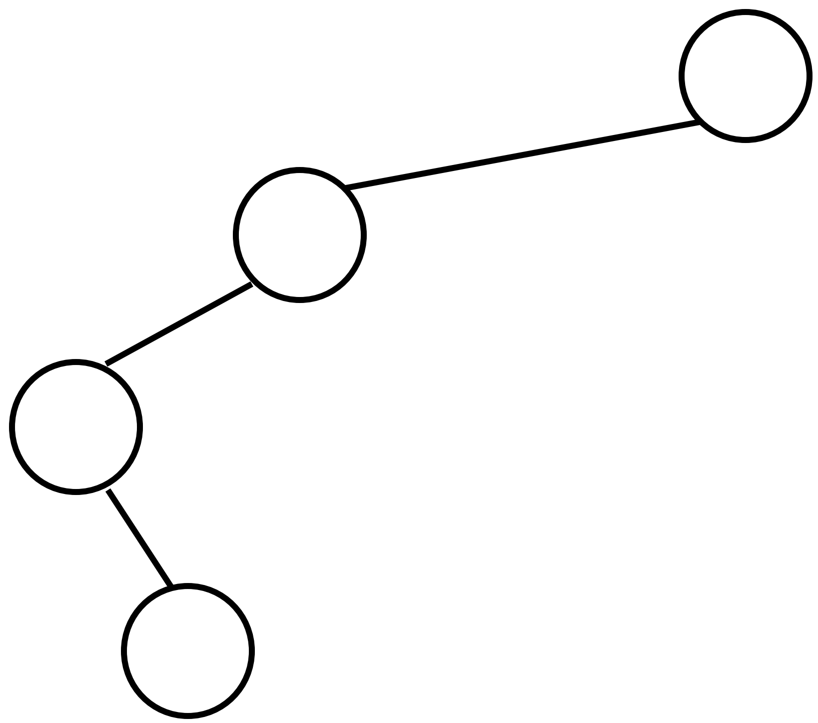
Binary tree

Every node has **at most** two children.

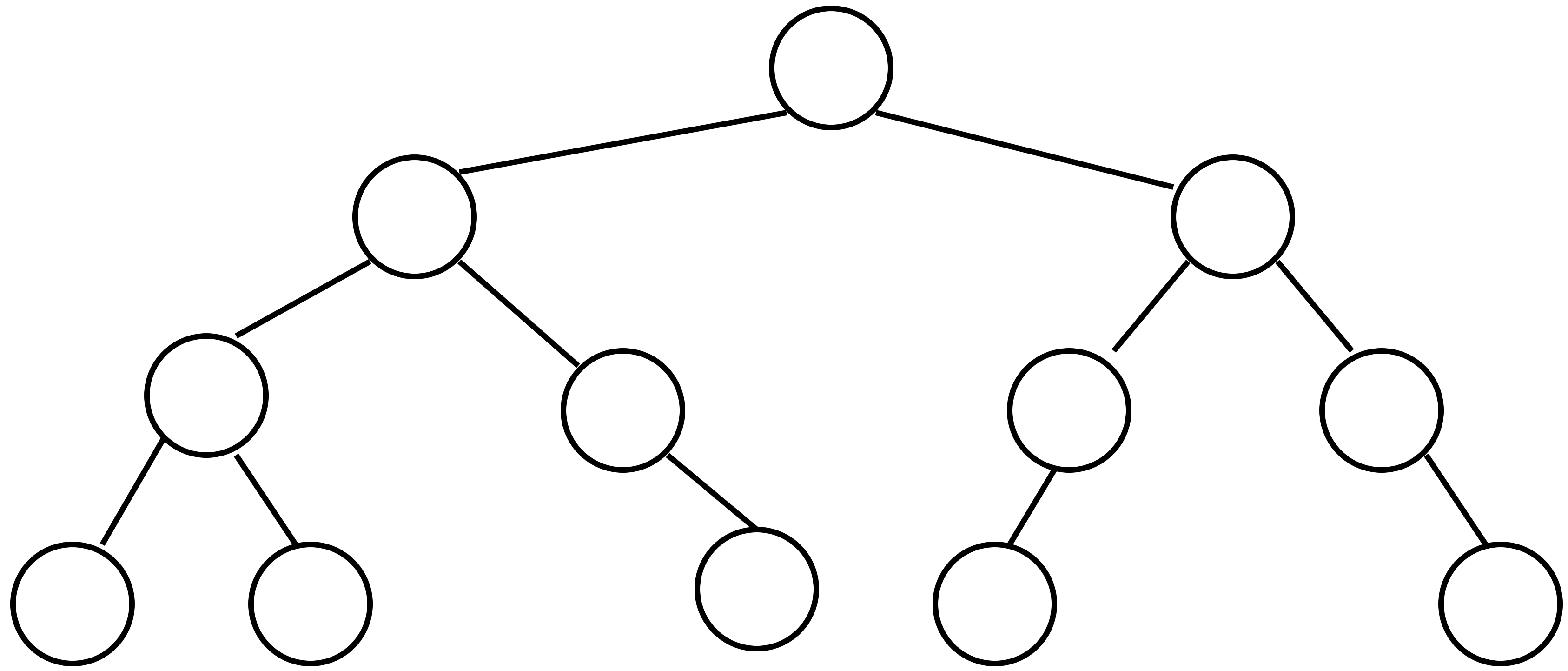


Note: Every subtree is a Binary Tree

Unbalanced Binary Tree



Balanced Binary Tree

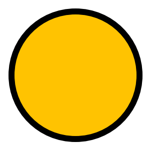


For every node

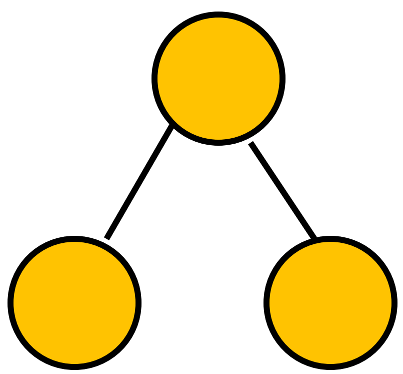
|height(left subtree) – height(right subtree)| ≤ 1

There are structures which maintain this, you'll see them in other units

Perfect Binary Trees



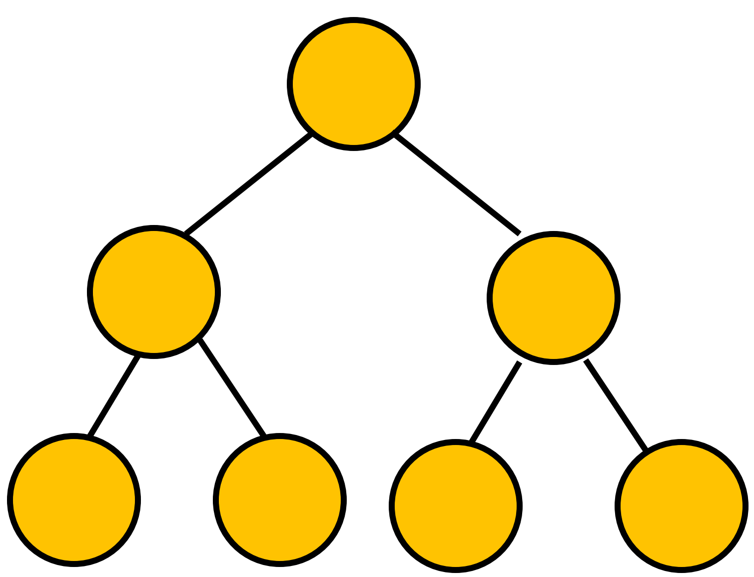
N =1 Height = 0



N =3 Height = 1

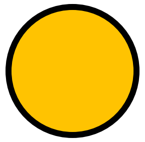
Each parent has two children

All leaves at same level



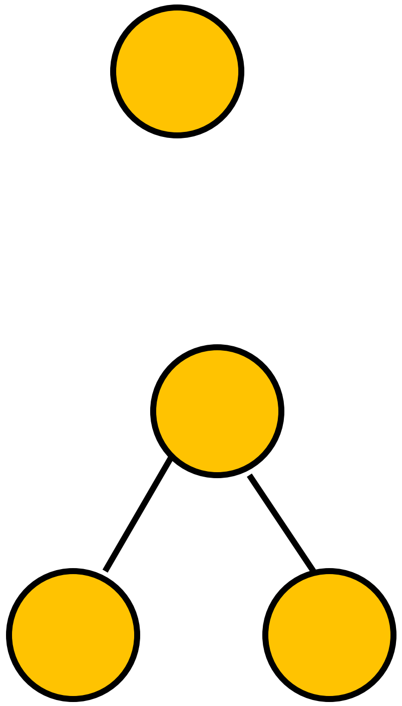
N =7 Height = 2

Perfect Binary Trees



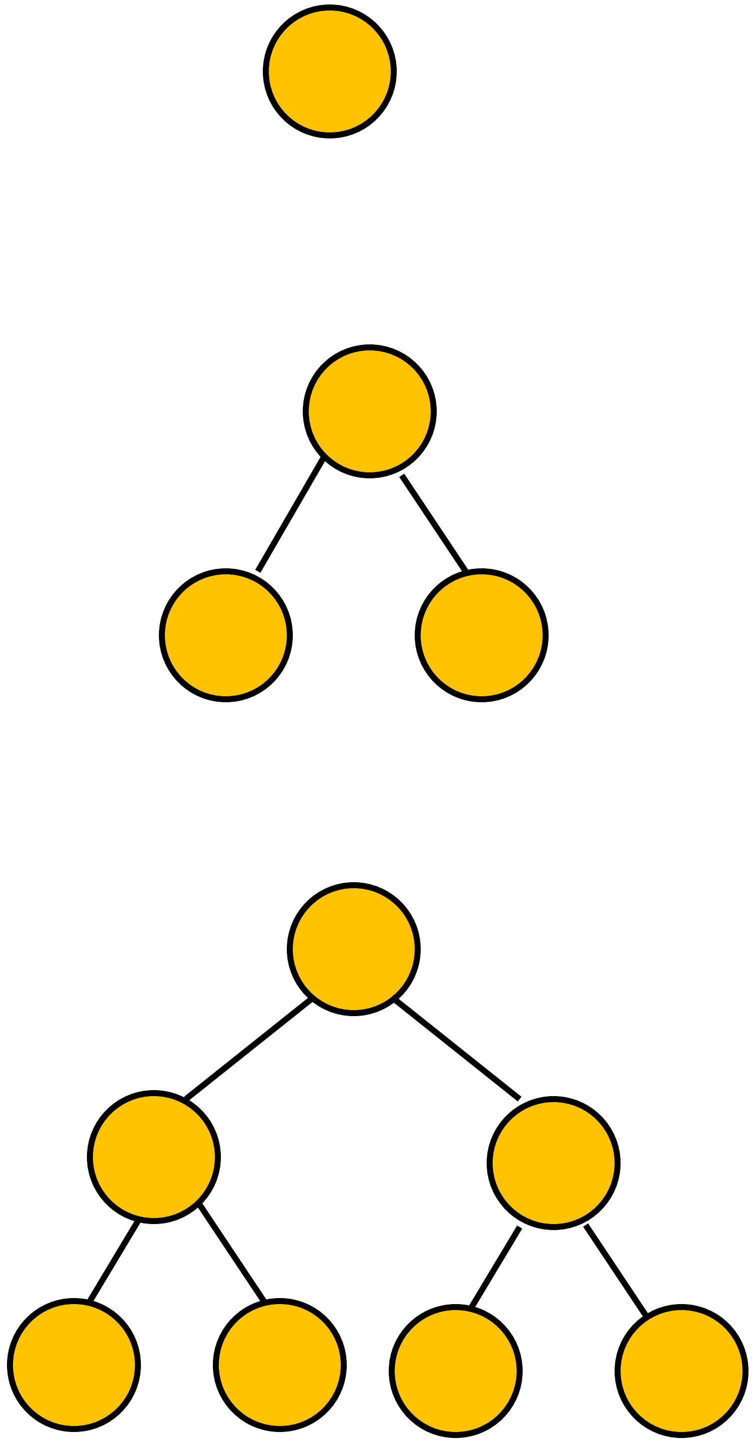
height	leaves
0	1

Perfect Binary Trees



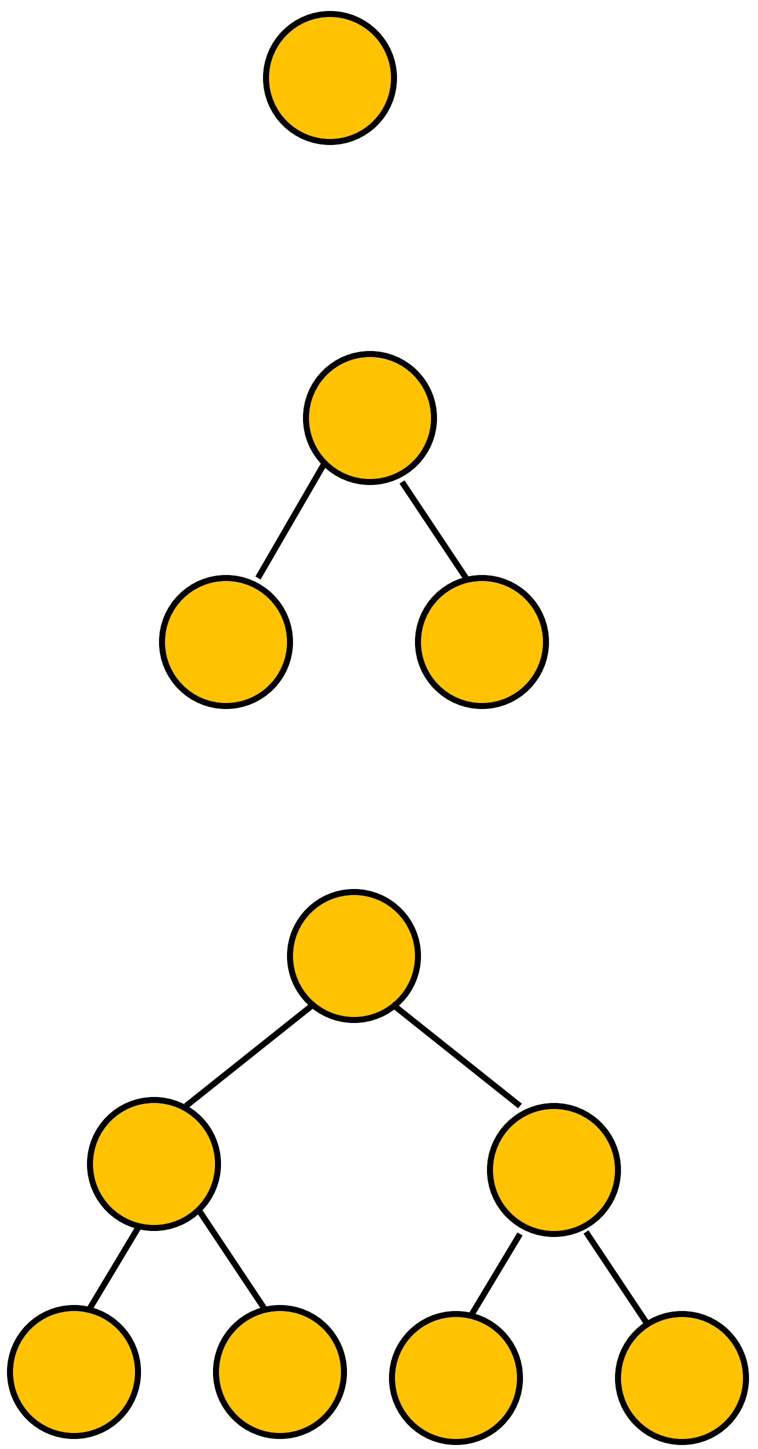
height	leaves
0	1
1	2

Perfect Binary Trees



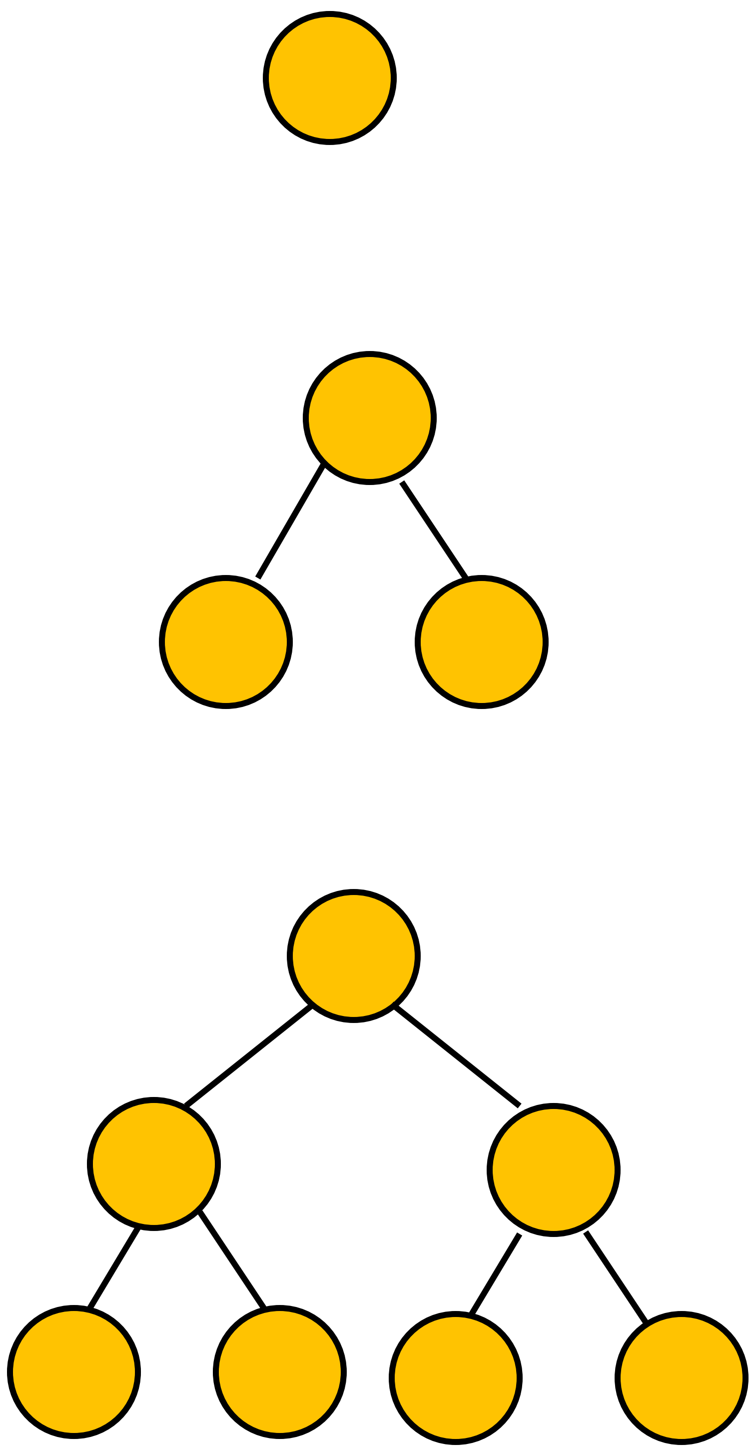
height	leaves
0	1
1	2
2	4

Perfect Binary Trees



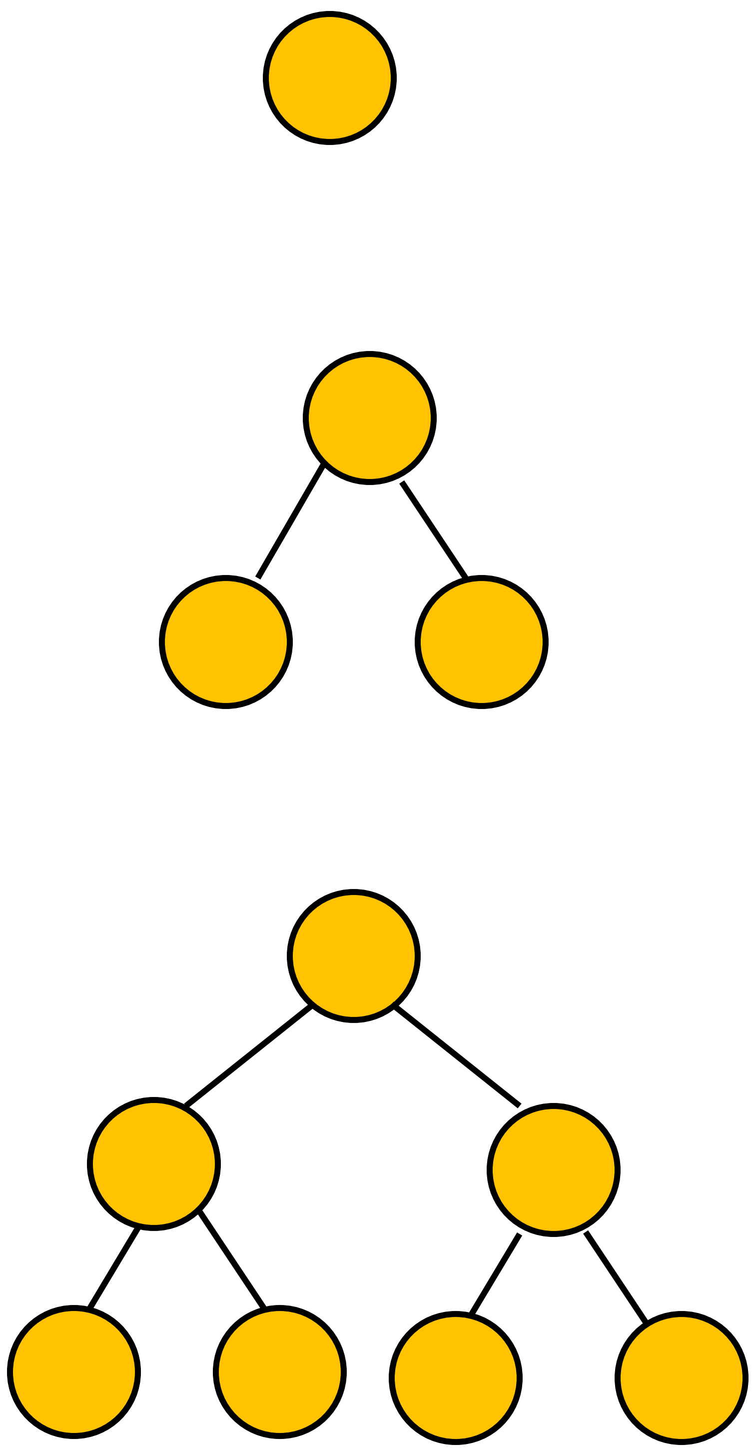
height	leaves
0	1
1	2
2	4
3	8

Perfect Binary Trees



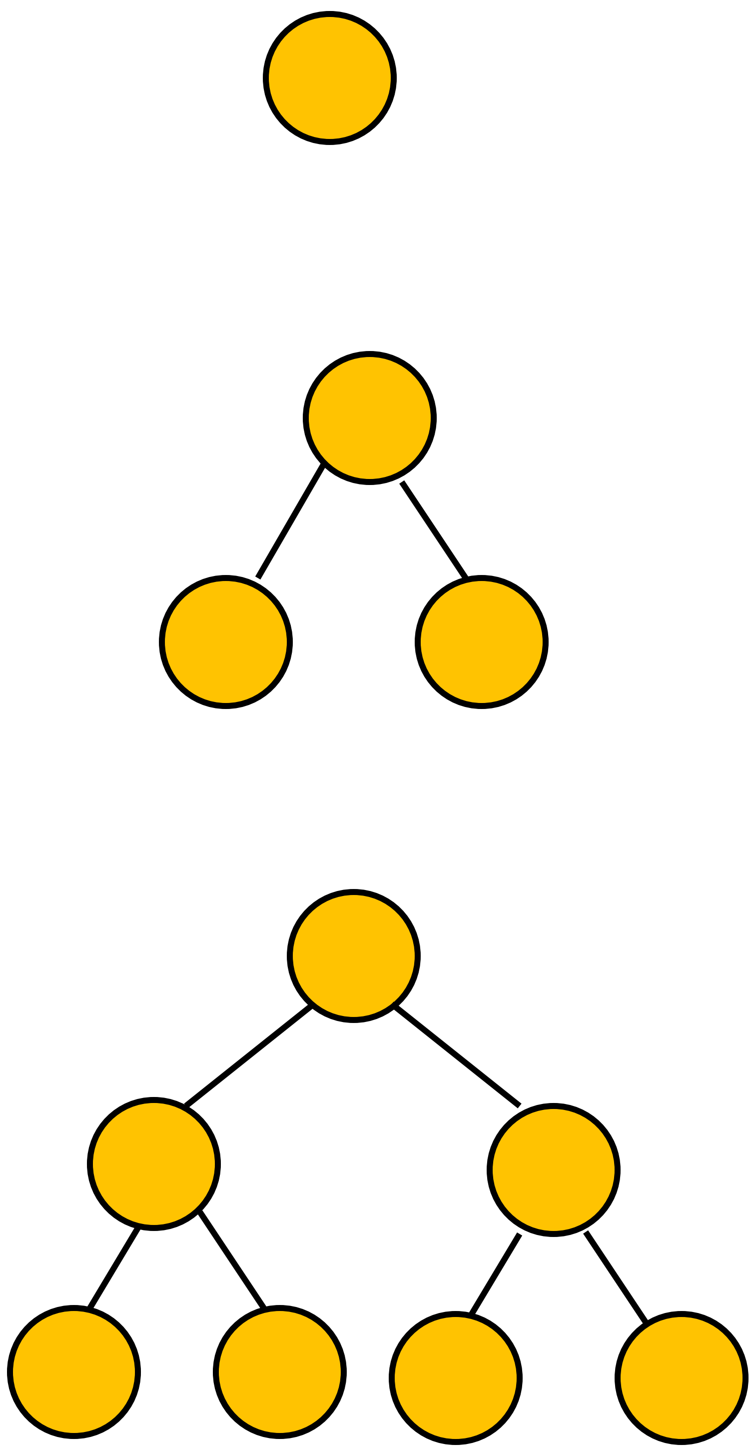
height	leaves
0	1
1	2
2	4
3	8
k	**2^k**

Perfect Binary Trees



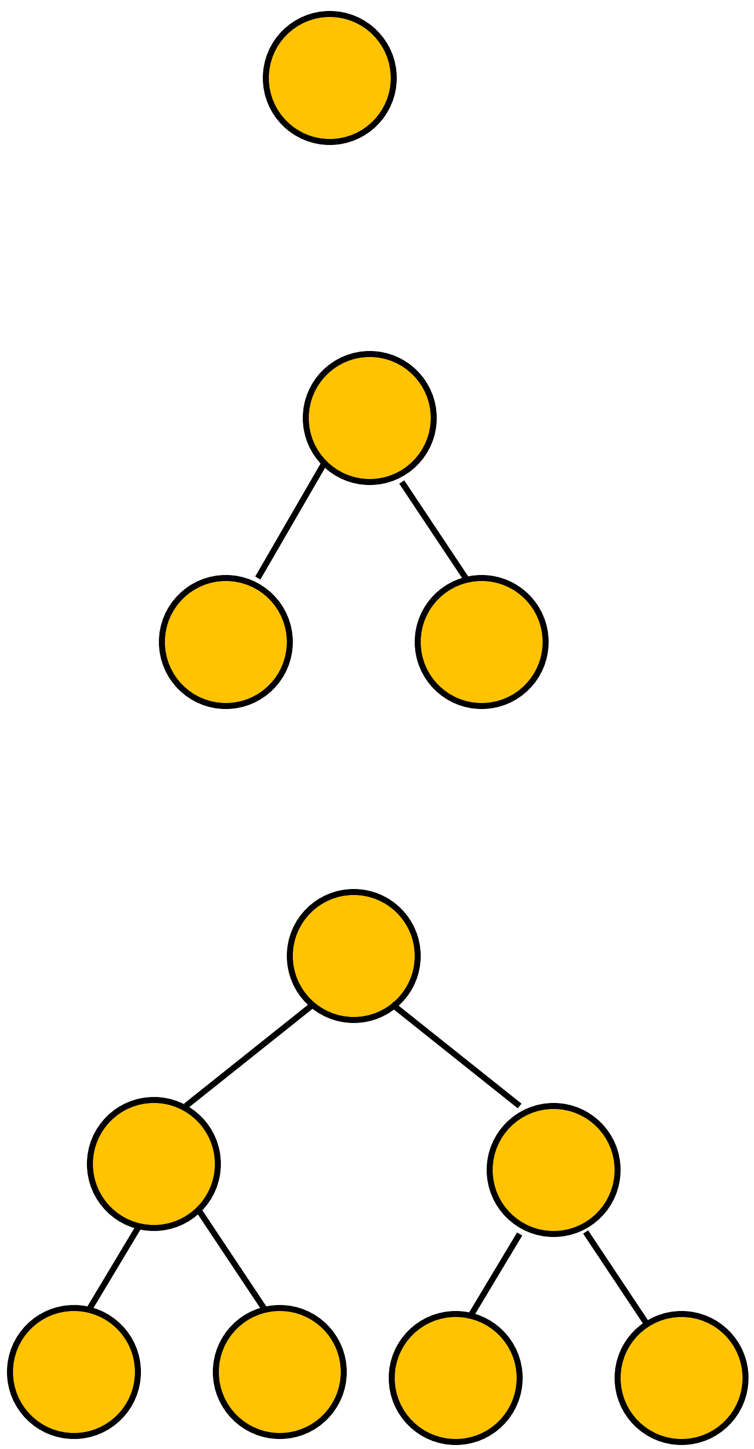
height	leaves	nodes
0	1	
1	2	
2	4	
3	8	
k	$\mathbf{2^k}$	

Perfect Binary Trees



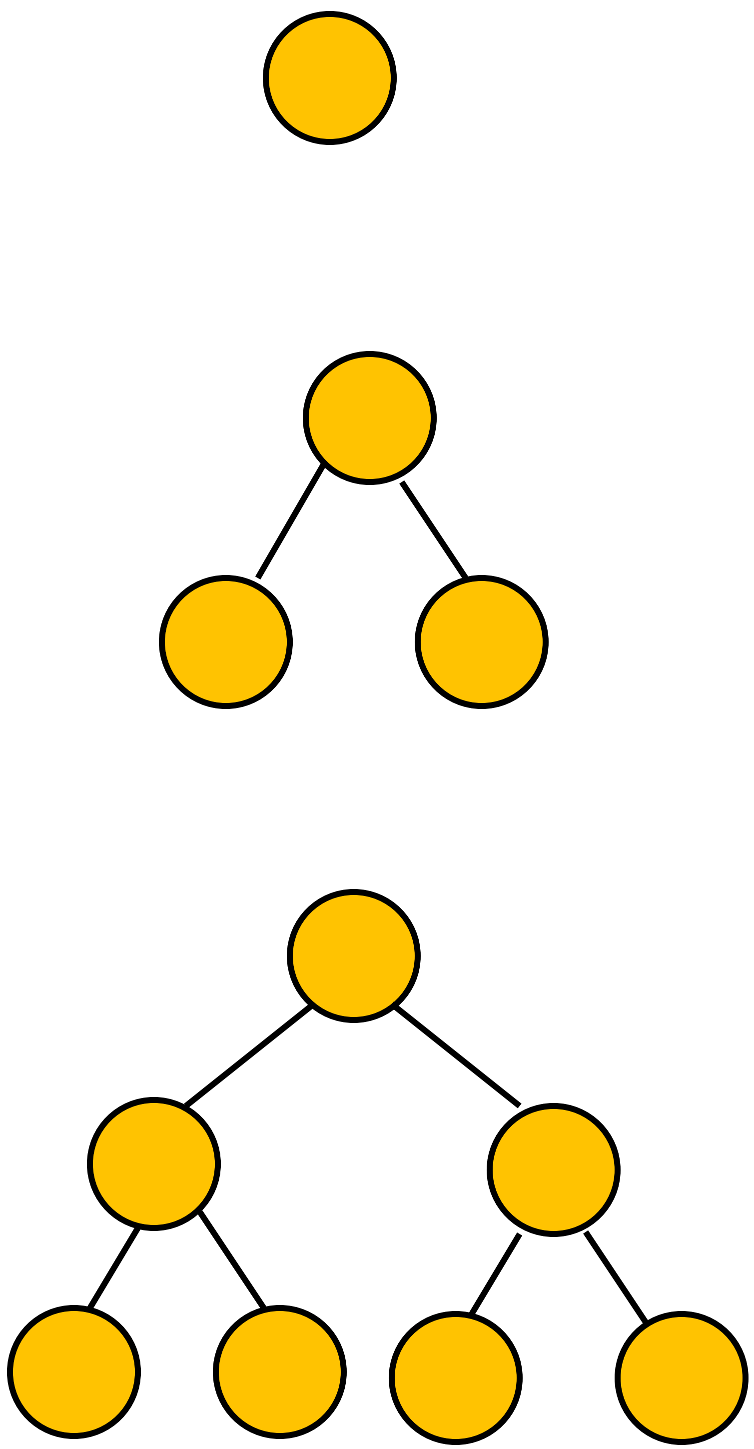
height	leaves	nodes
0	1	1
1	2	3
2	4	7
3	8	
k	**2^k**	

Perfect Binary Trees



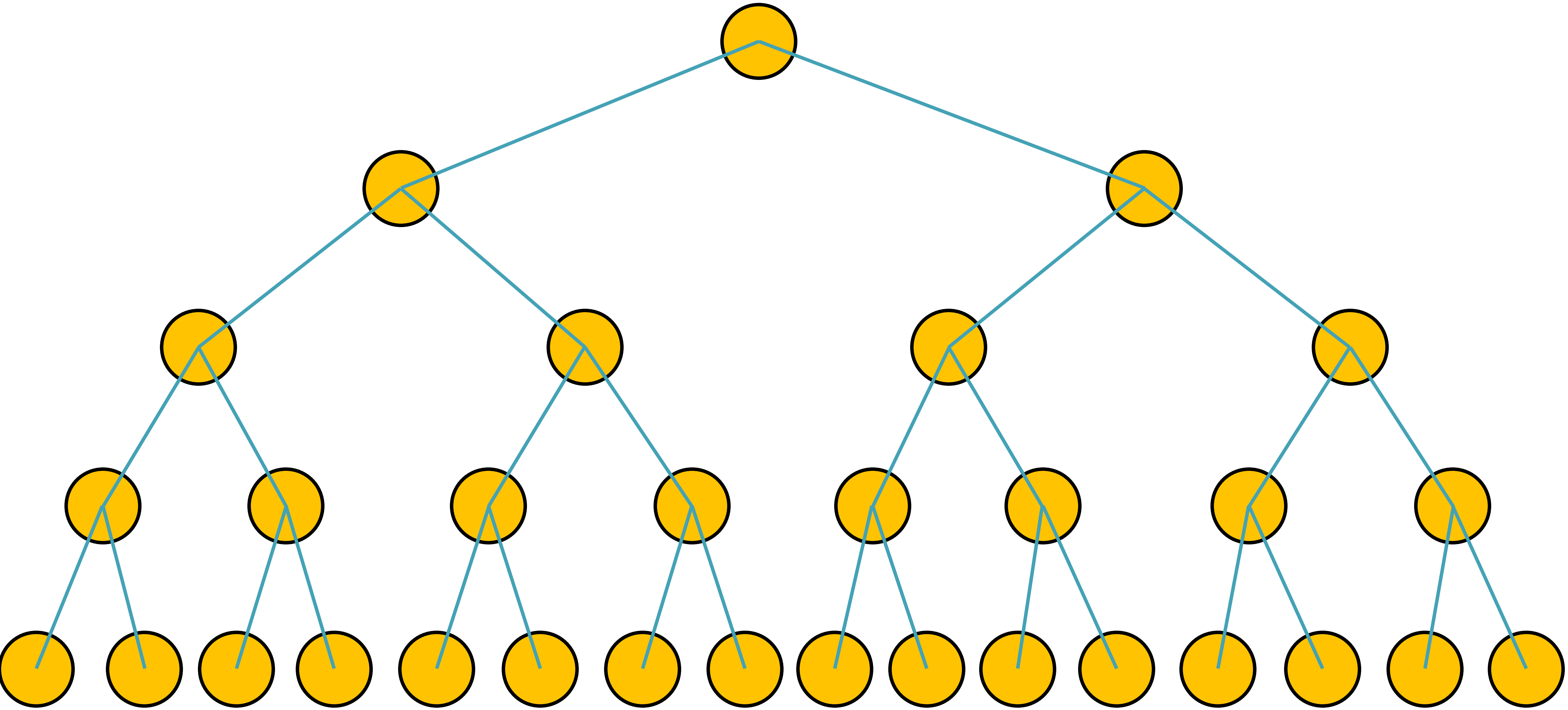
height	leaves	nodes
0	1	1
1	2	3
2	4	7
3	8	15
k	**2^k**	

Perfect Binary Trees



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

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

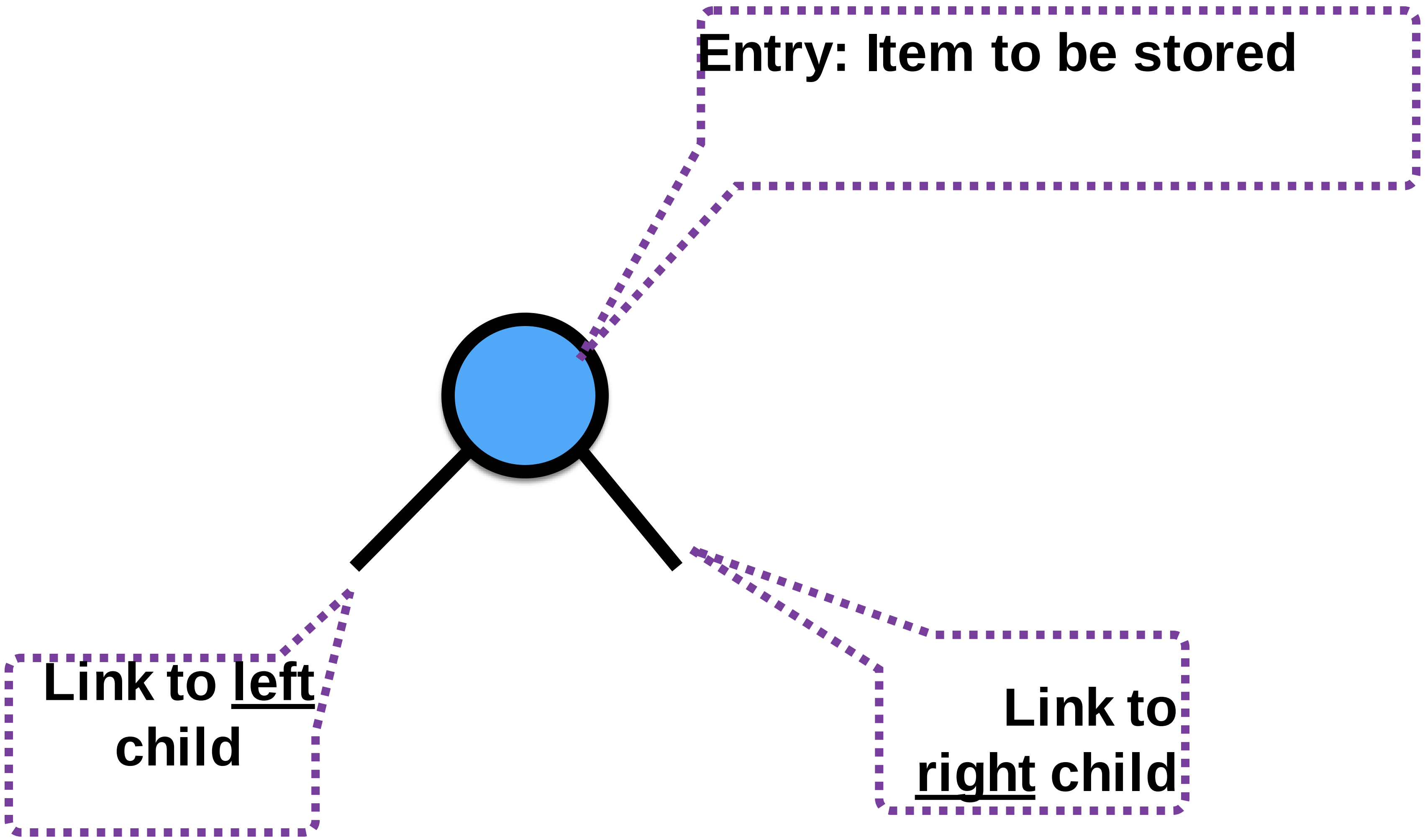
Balanced tree

the height is **O(logN)**

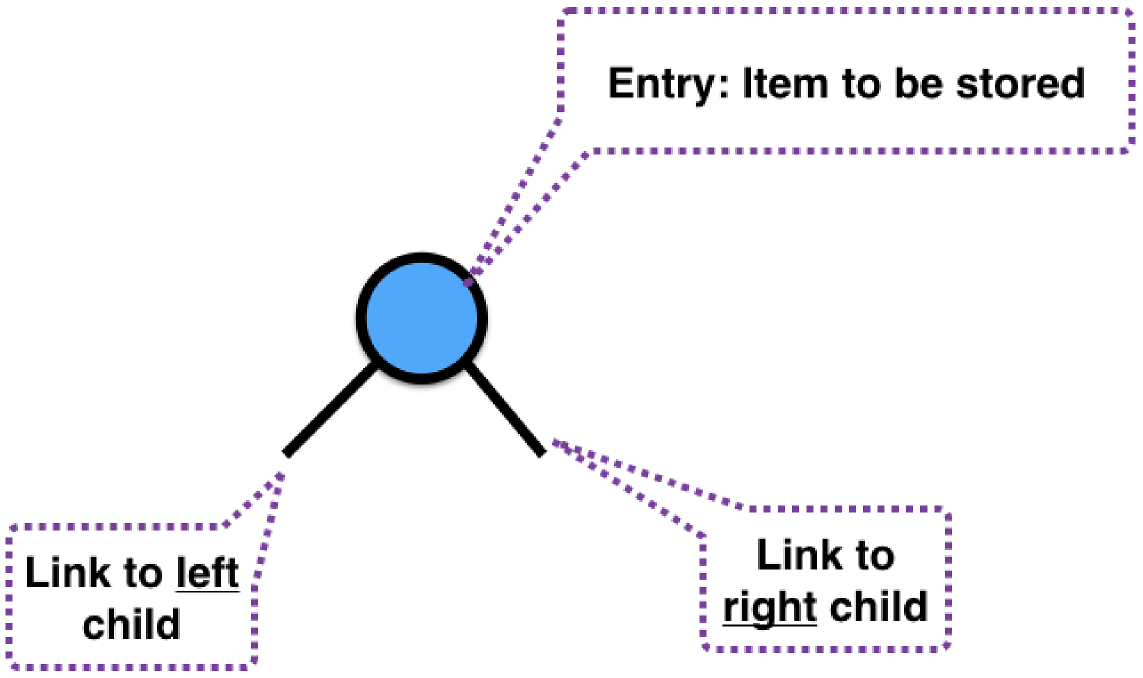
Unbalanced tree

the height is **O(N)**

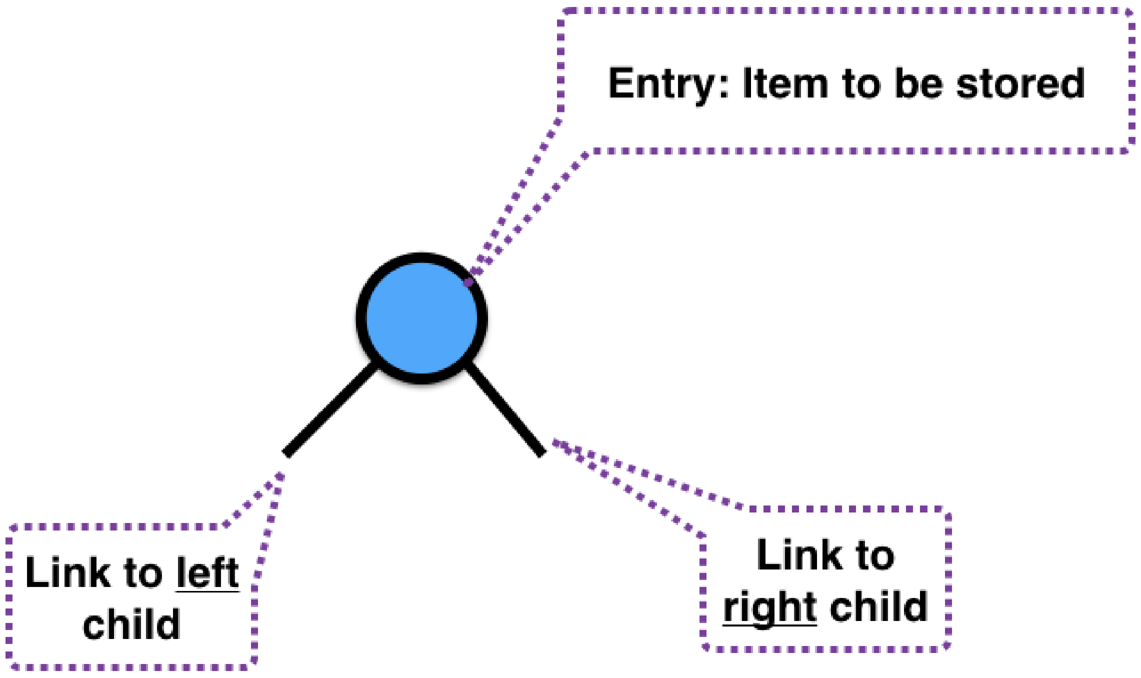
Representing a Binary Tree **Node**



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




```python
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)
```

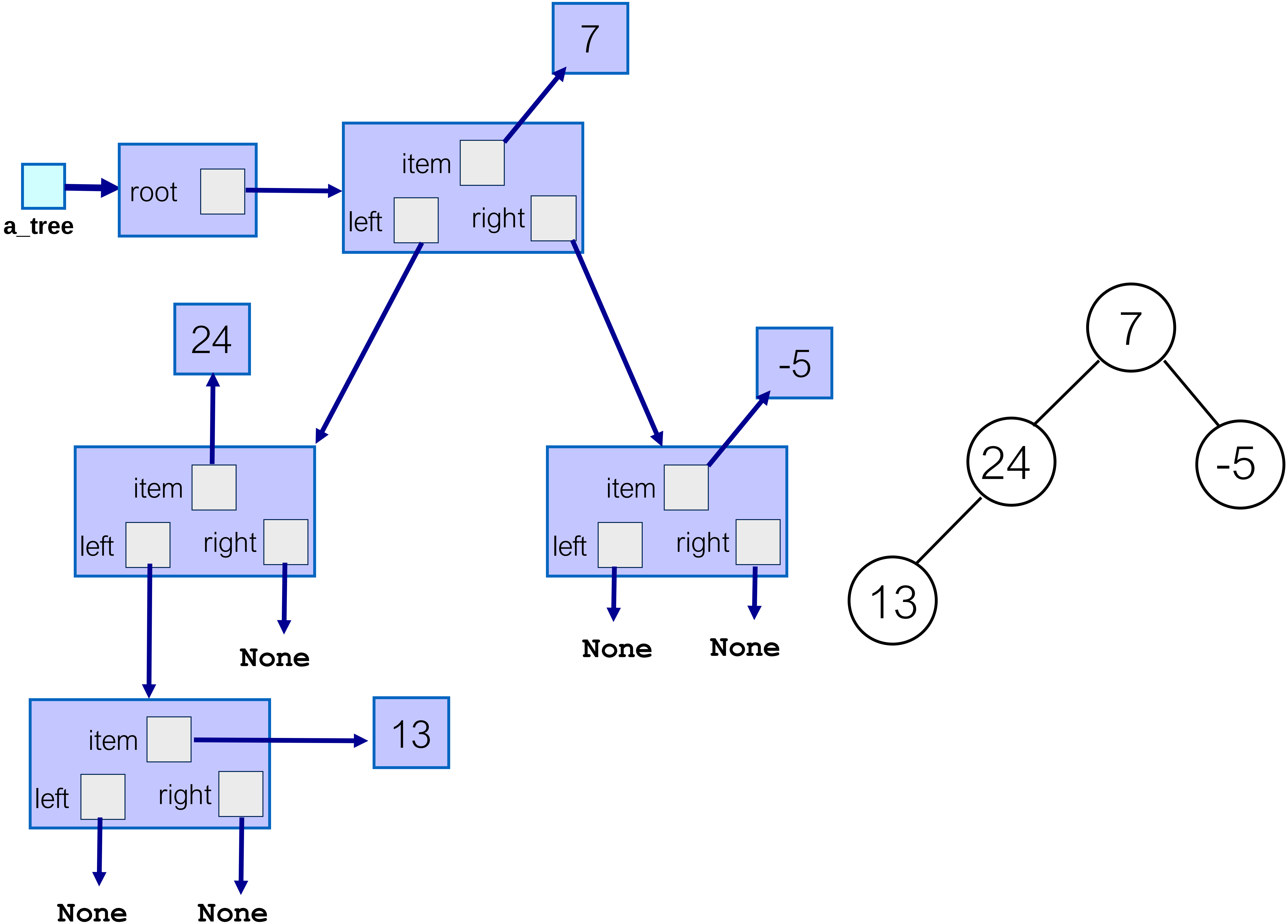
```python
class TreeNode:

    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)



class BinaryTree:
```

```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
```

As with linked lists, we only need the start element to access all others

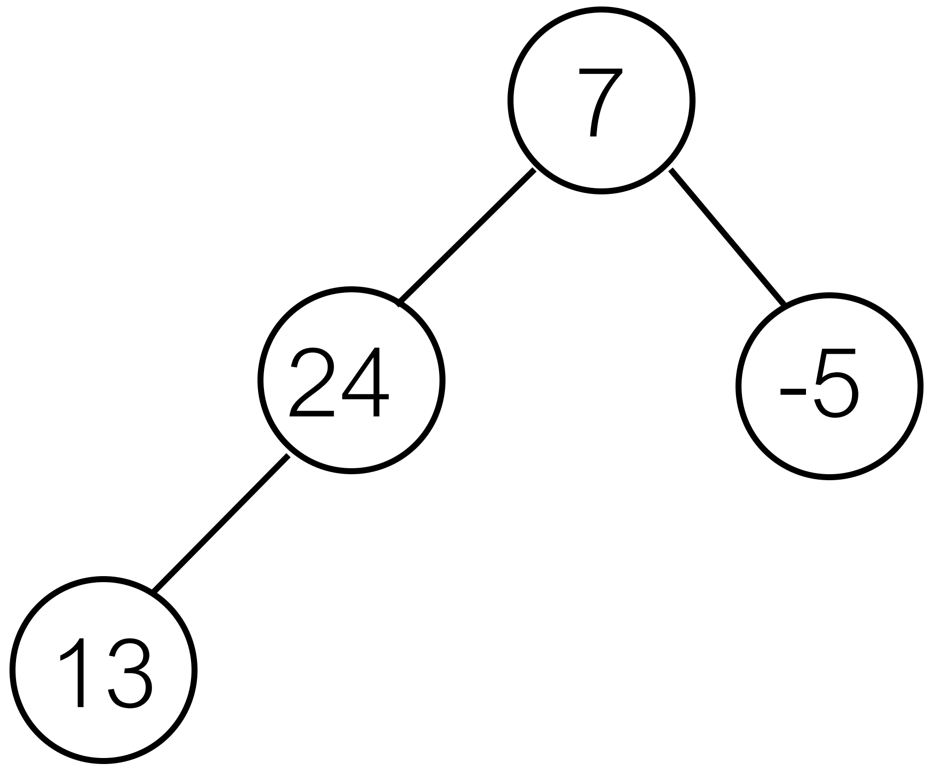
Only instance variable is a reference to the `root`



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

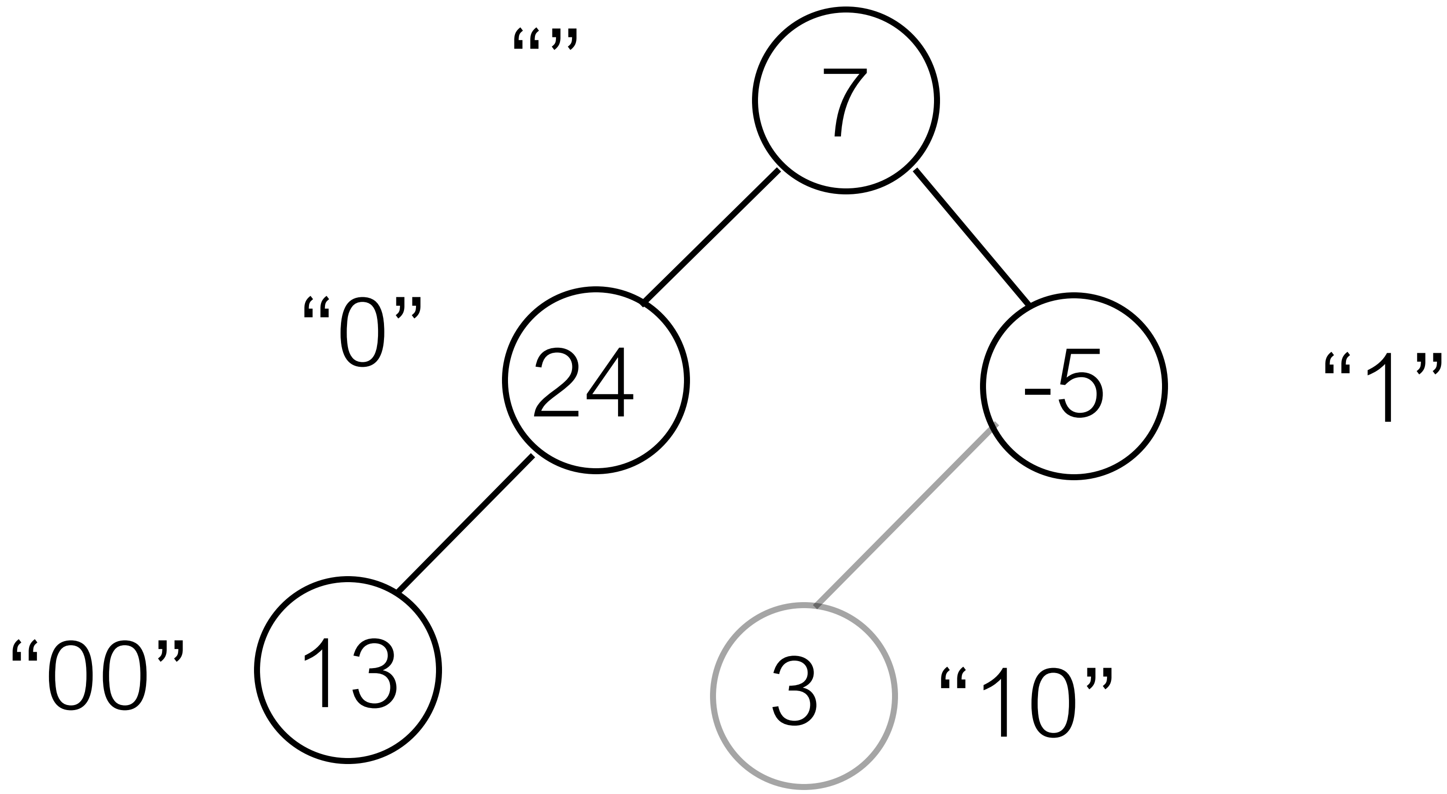
Add an item.

Add 3



where?

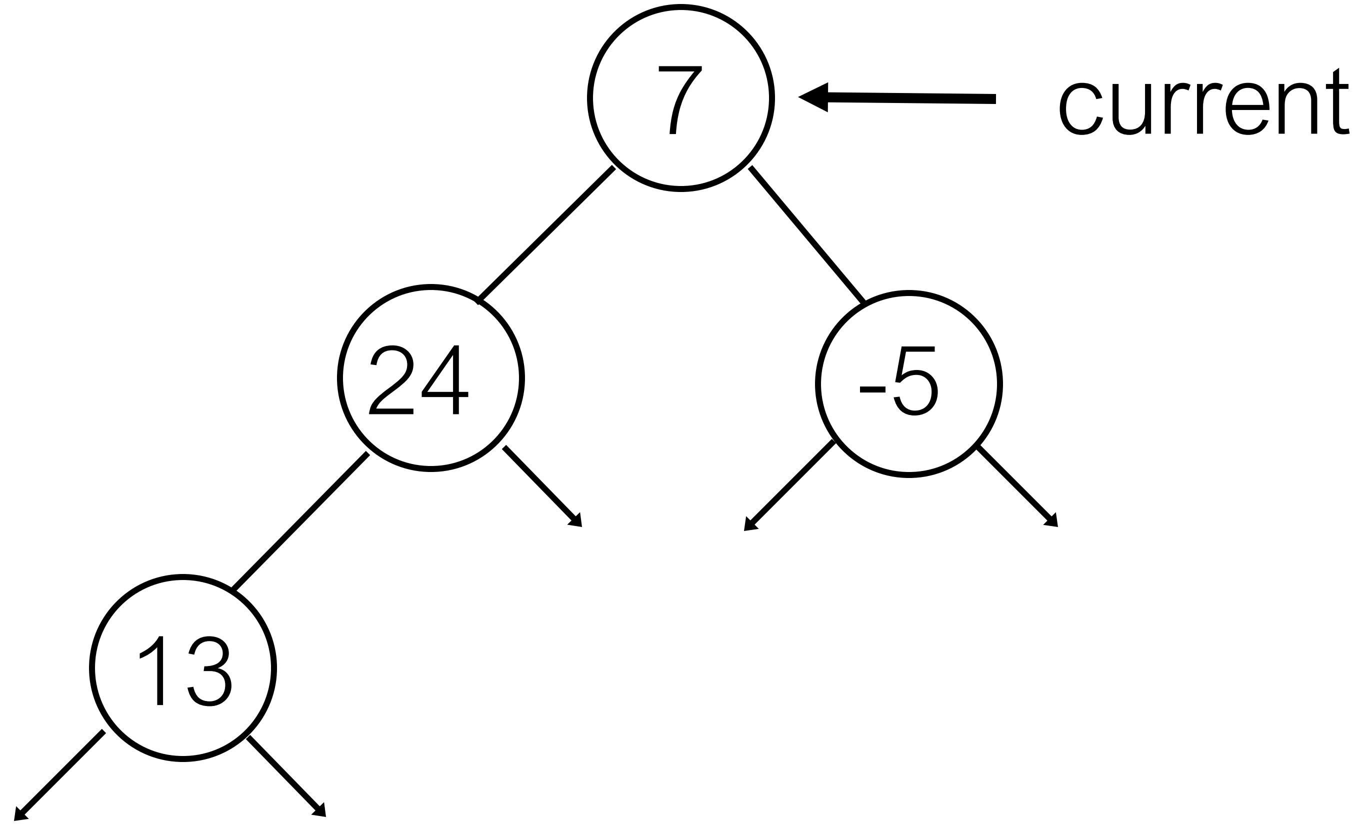
Add 3



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

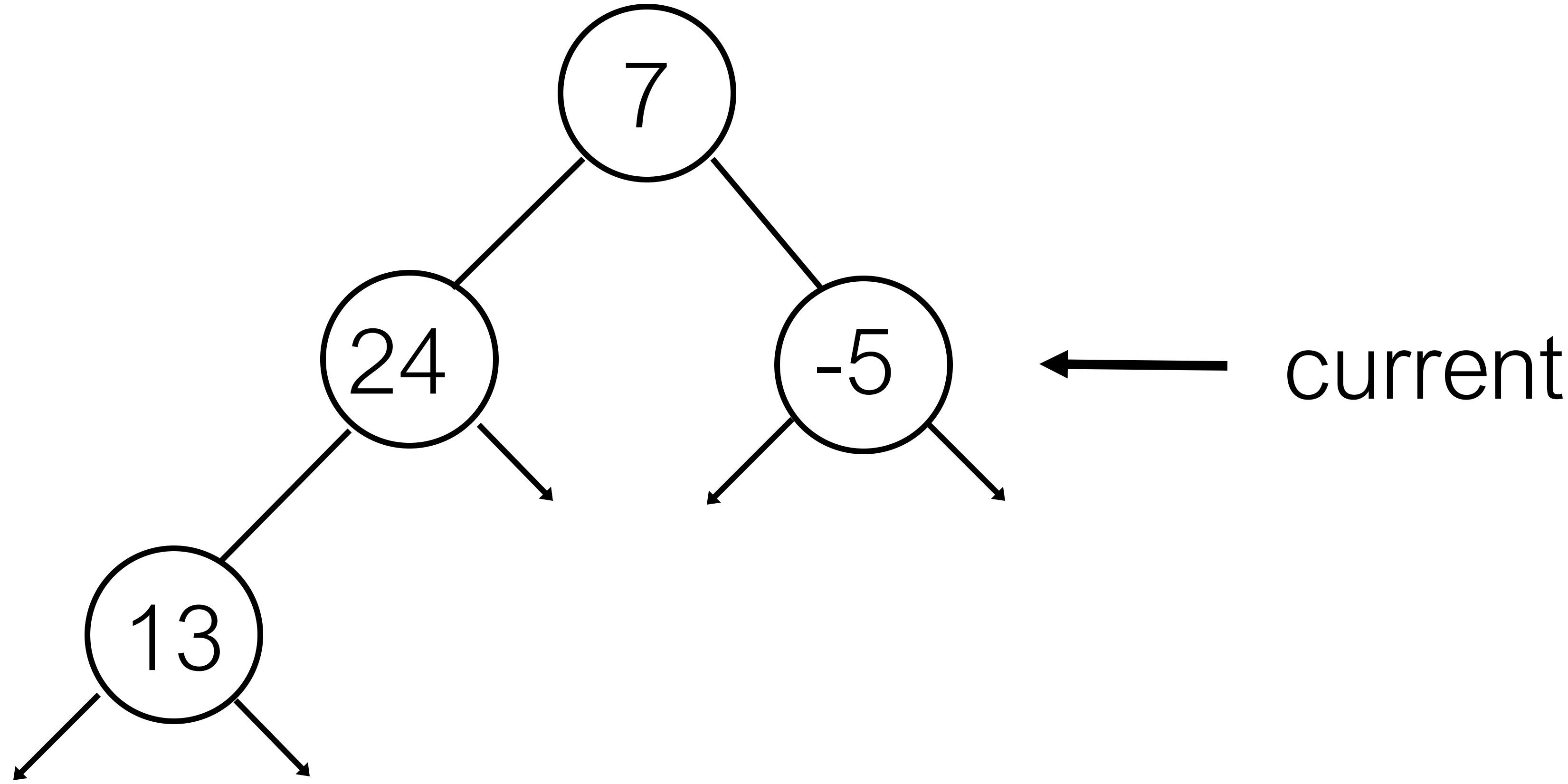
Here we use a string to decide where to place things, a good start, we’ll extend this in later lectures

Add 3



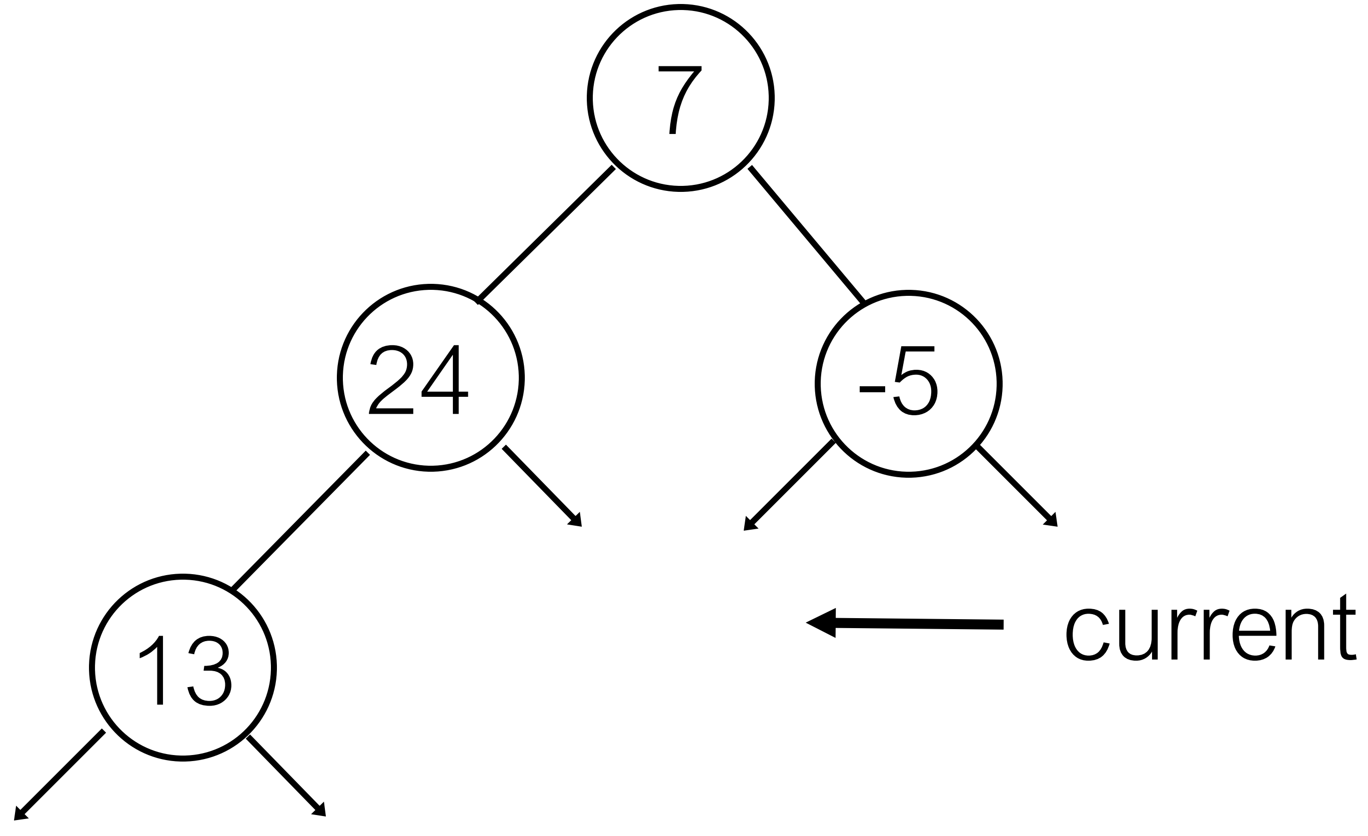
bitstring = “10”, **item**= 3

Add 3



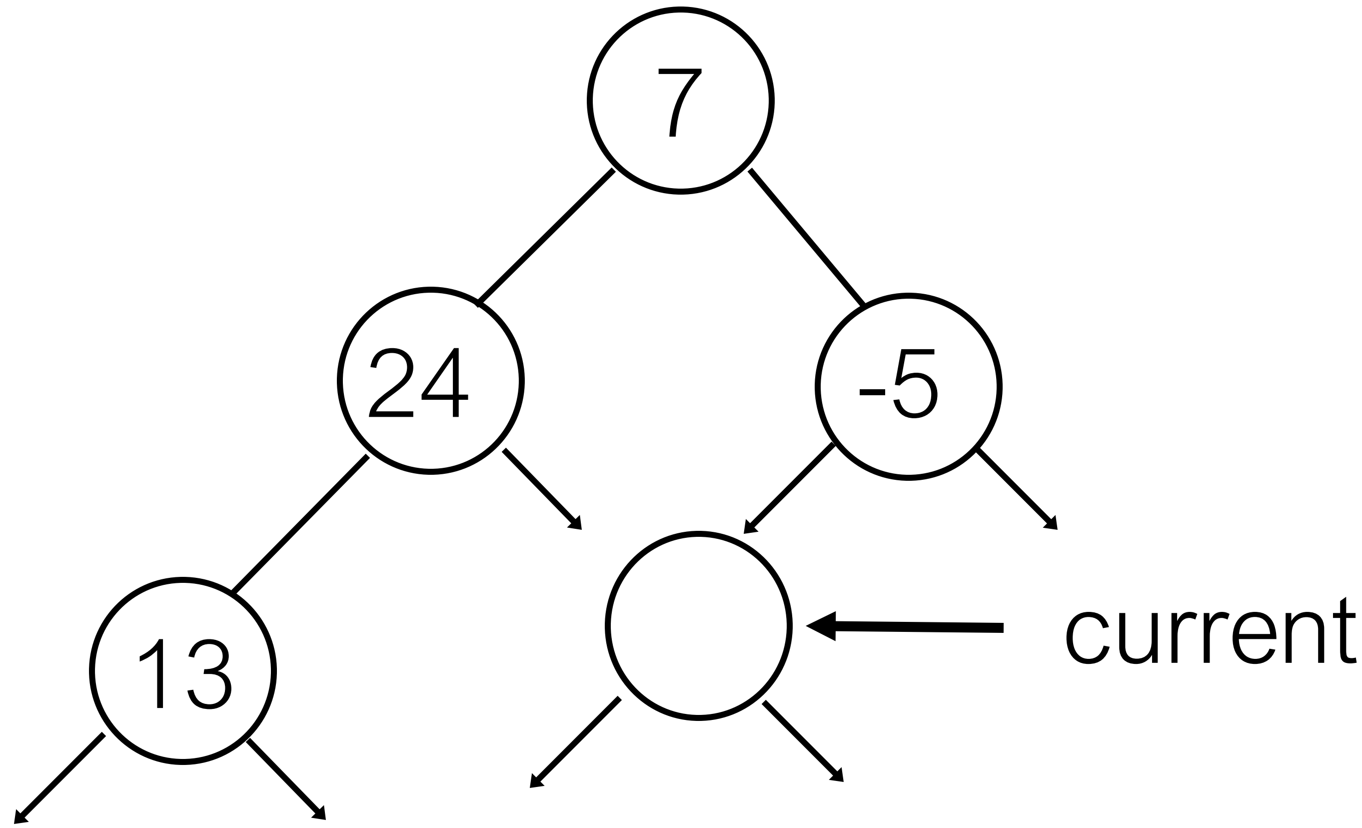
bitstring = “1̲0”, **item**= 3

Add 3



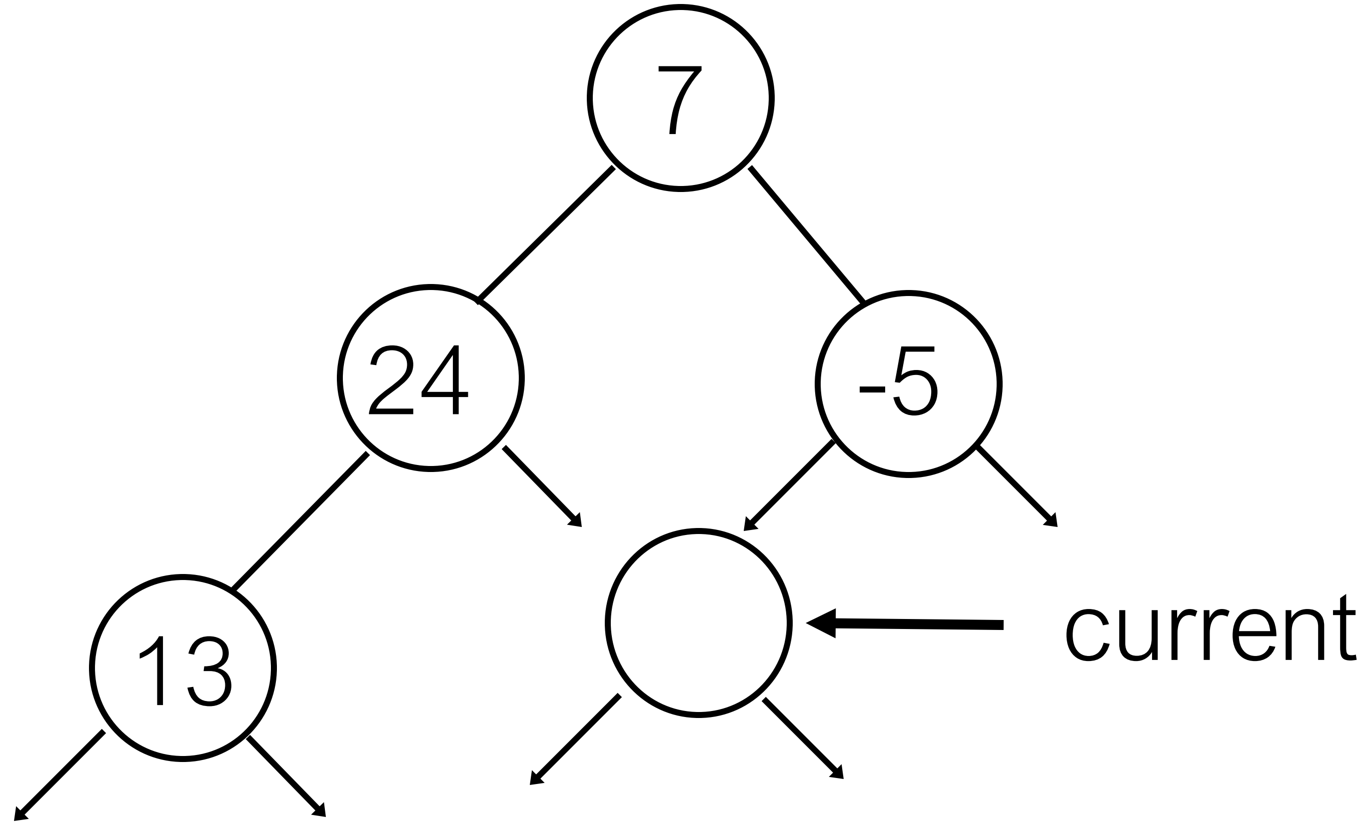
bitstring = “1<u>0</u>”, **item**= 3

Add 3



bitstring = “1<u>0</u>”, **item**= 3

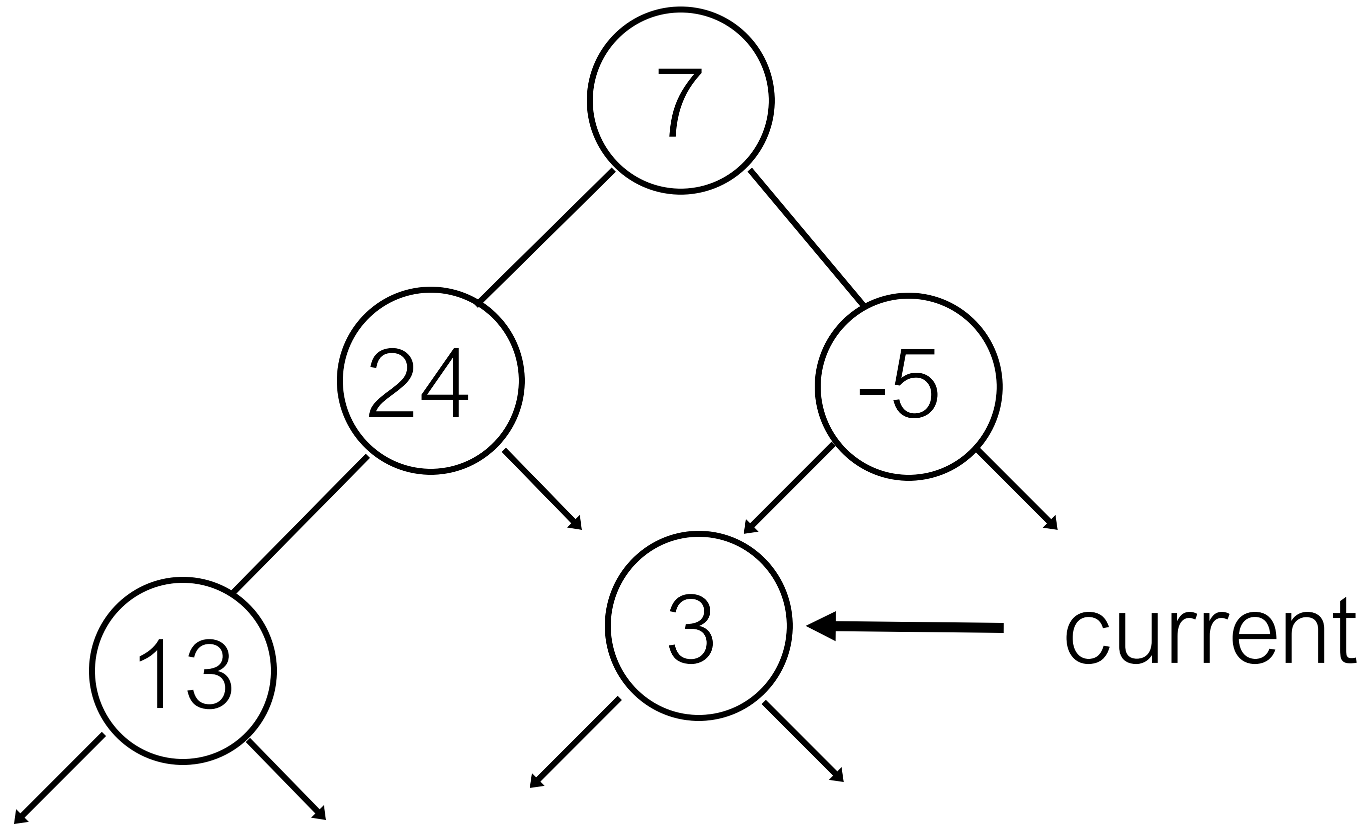
Add 3



bitstring = “10”, **item**= 3

Iteration ended, so this must be the place….

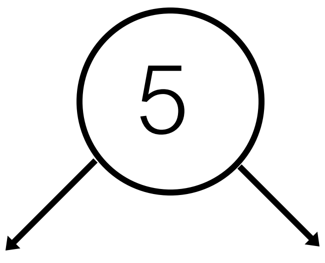
Add 3



bitstring = “10”, **item**= 3

Examples

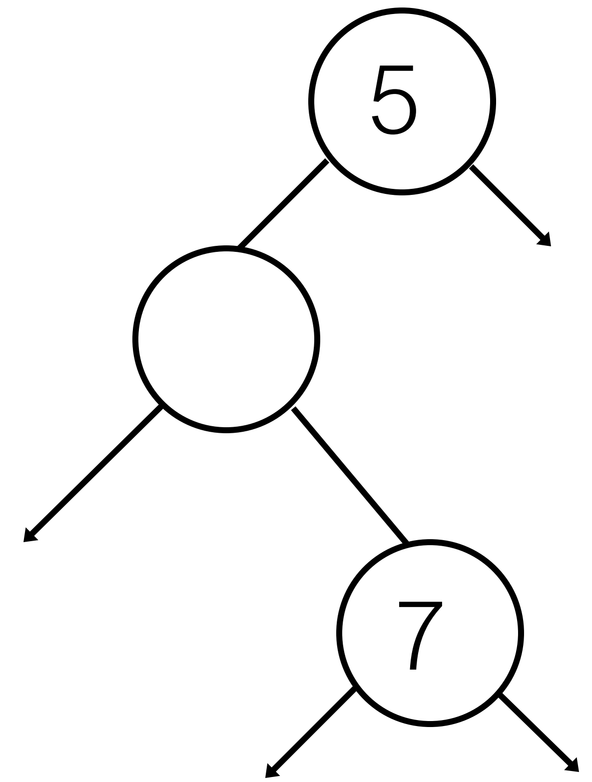
bitstring = “”, **item**= 5



Examples

bitstring = “”, **item**= 5

bitstring = “01”, **item**= 7

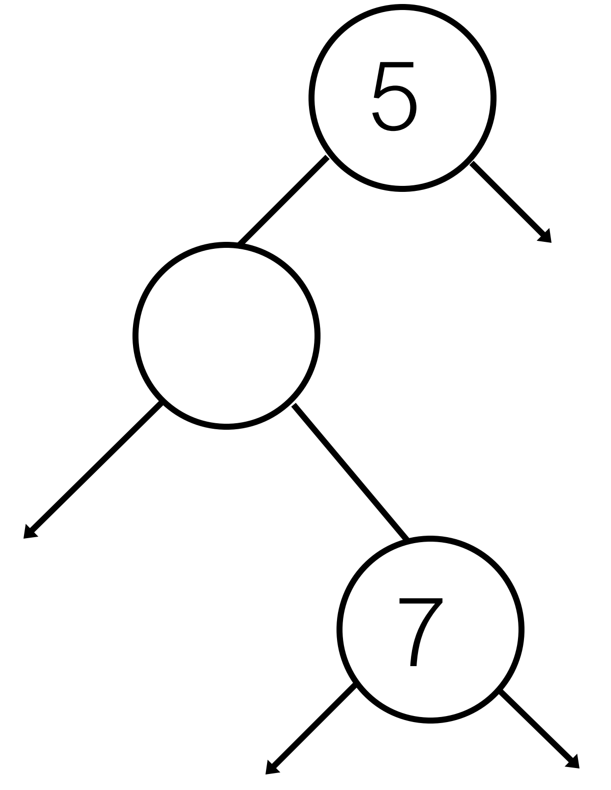


Examples

bitstring = “”, **item**= 5

bitstring = “01”, **item**= 7

bitstring = “ ”, **item**= 2

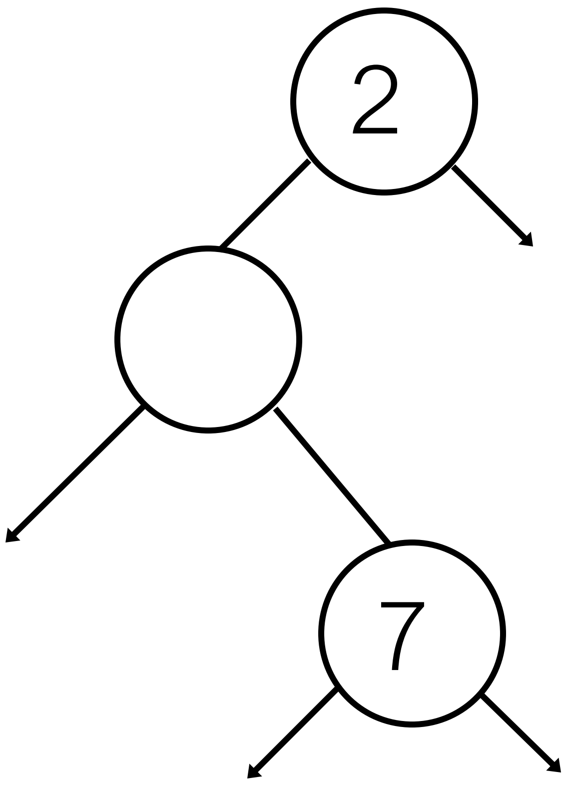


Examples

bitstring = “”, **item**= 5

bitstring = “01”, **item**= 7

bitstring = “ ”, **item**= 2



Recursively explore subtree
following “bitstring directions”

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

```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
```

Add empty node if it does not exist

Explore left branch

Explore right branch

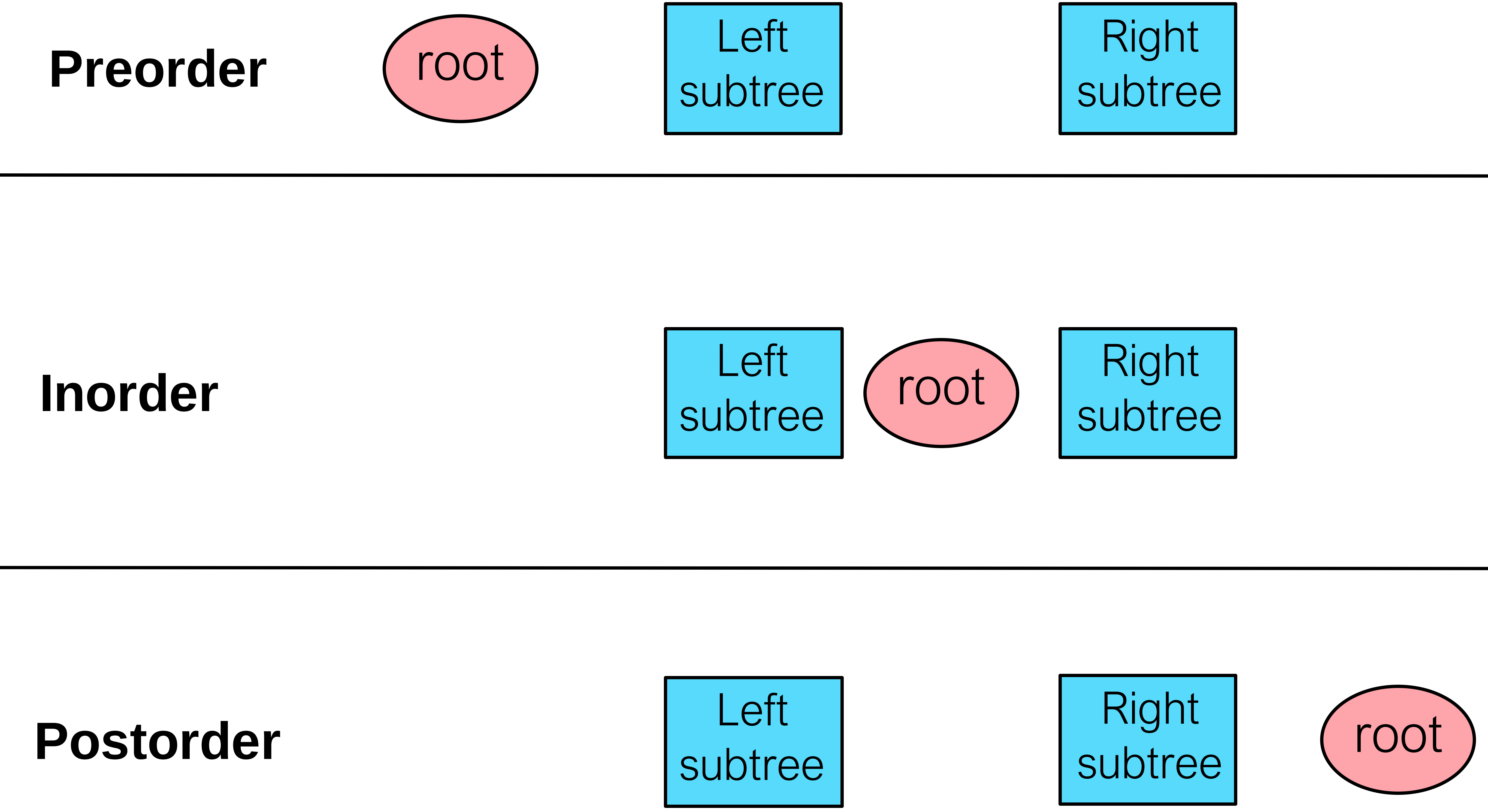
Bitstring is telling me I have arrived at the correct stop

Assigning to left or right to include the created node as a child

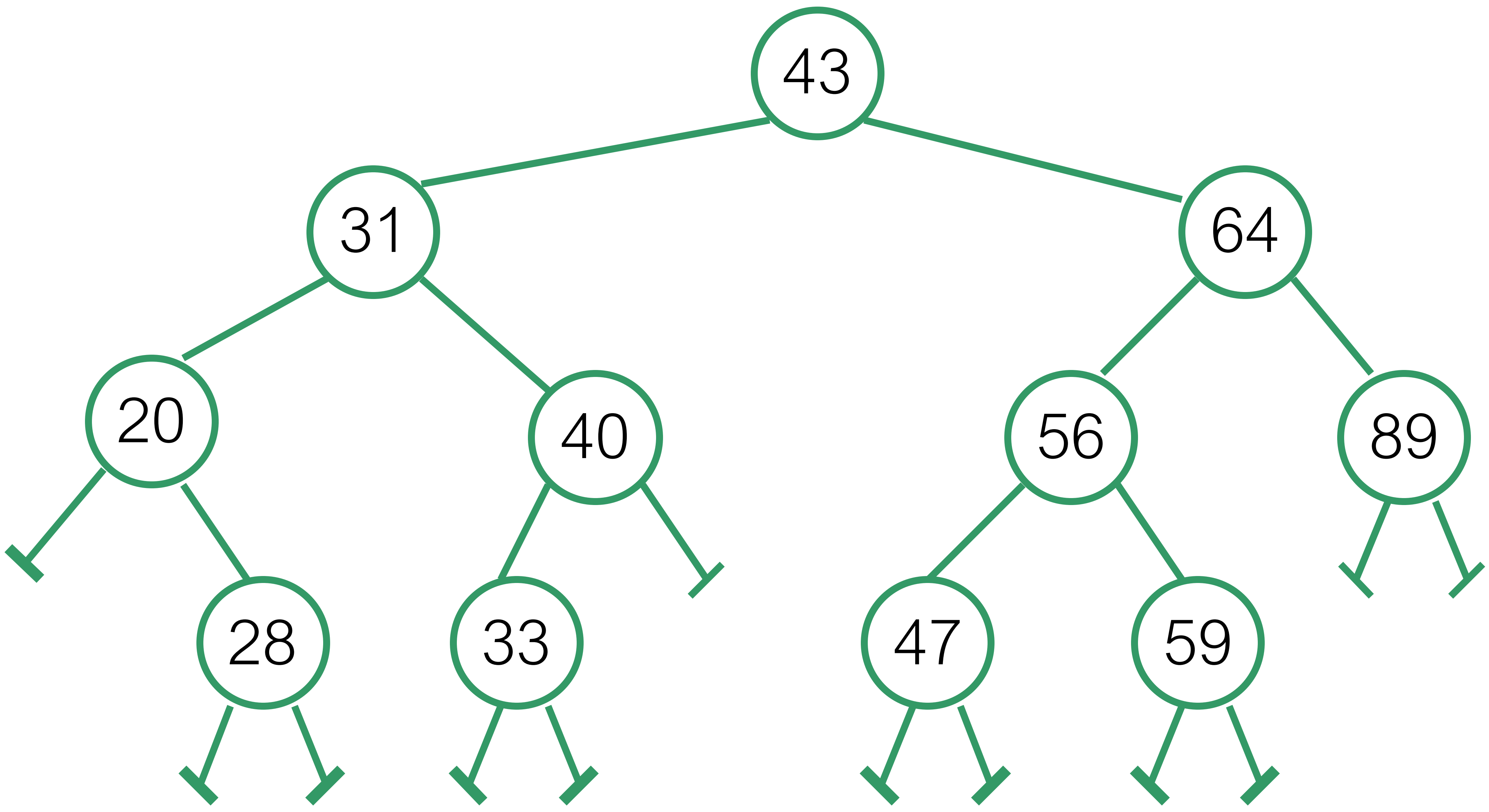
As we continue returning current from there, at higher levels are we just reassigning something to Itself. We could equally just stop when the child is None and assign something there – nothing to return

Traversal

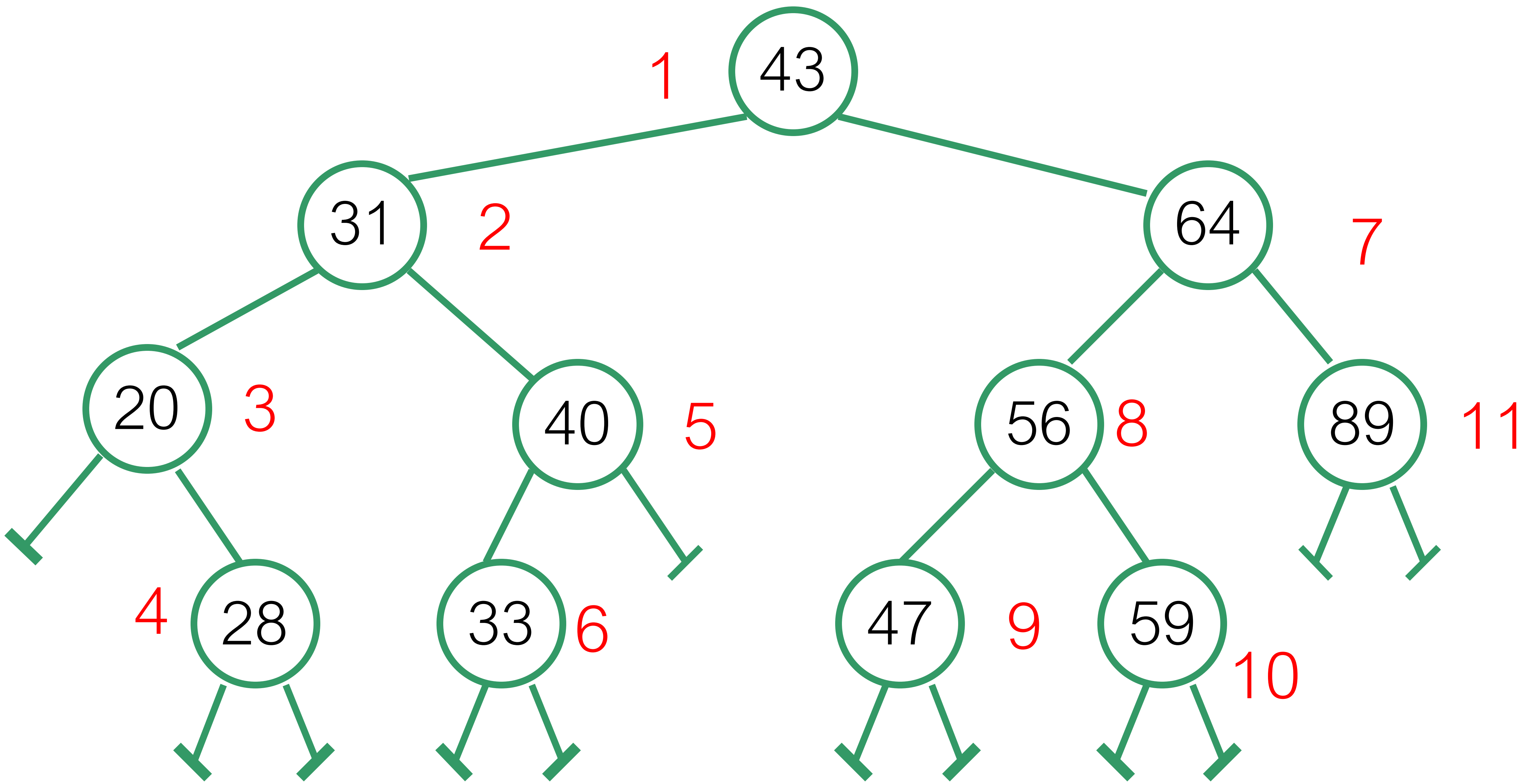
- Systematic way of **visiting**/processing **all the nodes**
- **Methods**: Preorder, Inorder, and Postorder
- They **all** traverse the left subtree before the right subtree. It's all about the **position of the root**.


Preorder
root
Left subtree
Right subtree
Inorder
Left subtree
root
Right subtree
Postorder
Left subtree
Right subtree
root

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)
```

Summary

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