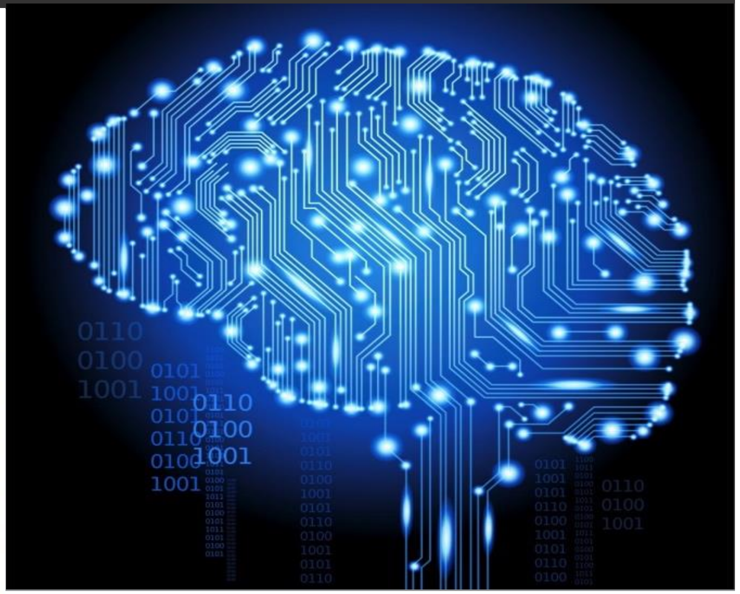
MONASH University

Information Technology

Tree Basics

Prepared by Maria Garcia de la Banda
Updated by Brendon Taylor



Objectives for this lecture

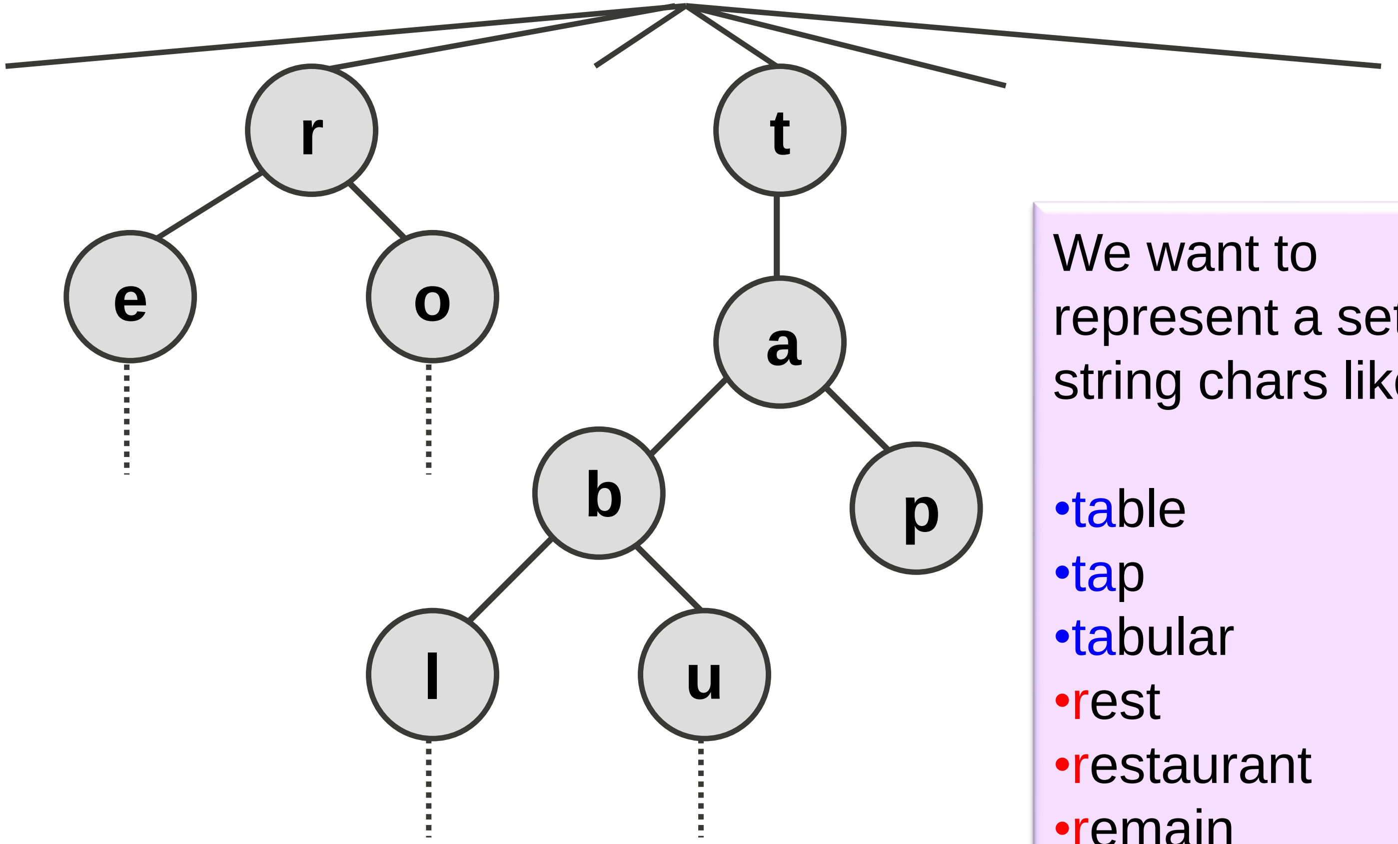
- **To review the basic concepts for Trees**
- **To understand**
  - The concept of binary tree
  - What an unbalanced binary tree is and its impact on tree operations
  - Three of the main basic traversals on binary trees
- **To be able to implement a basic binary tree, its traversals and a few simple operations**

Trees

Trees

- **Extremely useful in practice**
- **Natural way of modeling many things, such as:**
  - 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)**
- **And for compactly representing some data**

Compact representation of data

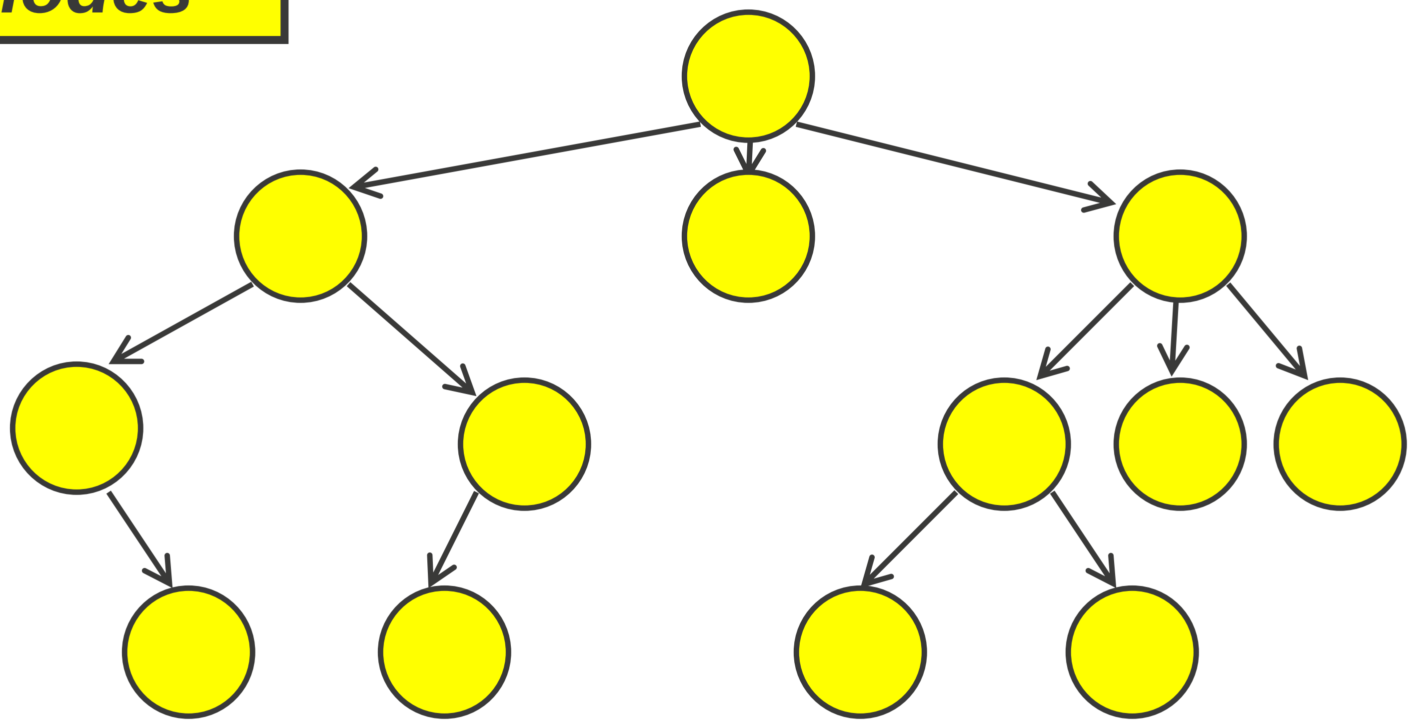


We want to represent a set of string chars like:

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

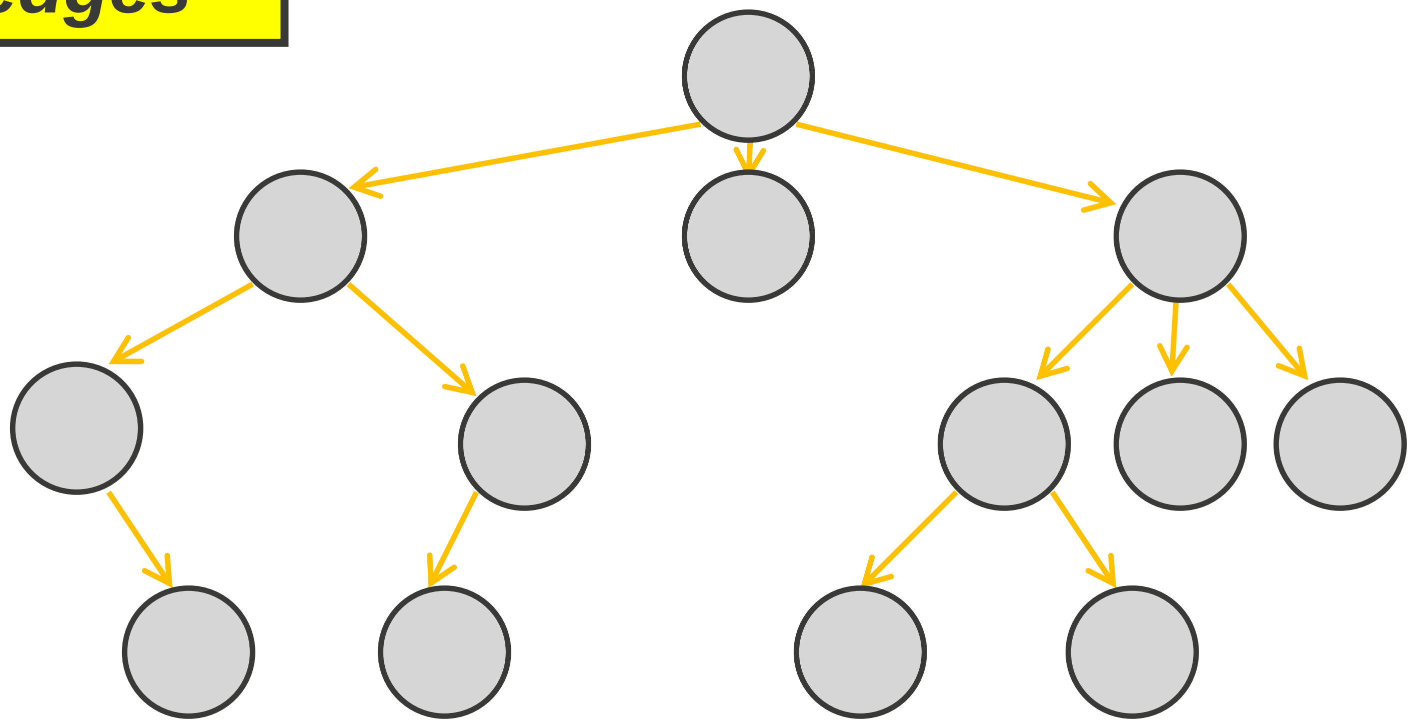
Tree Notation

nodes



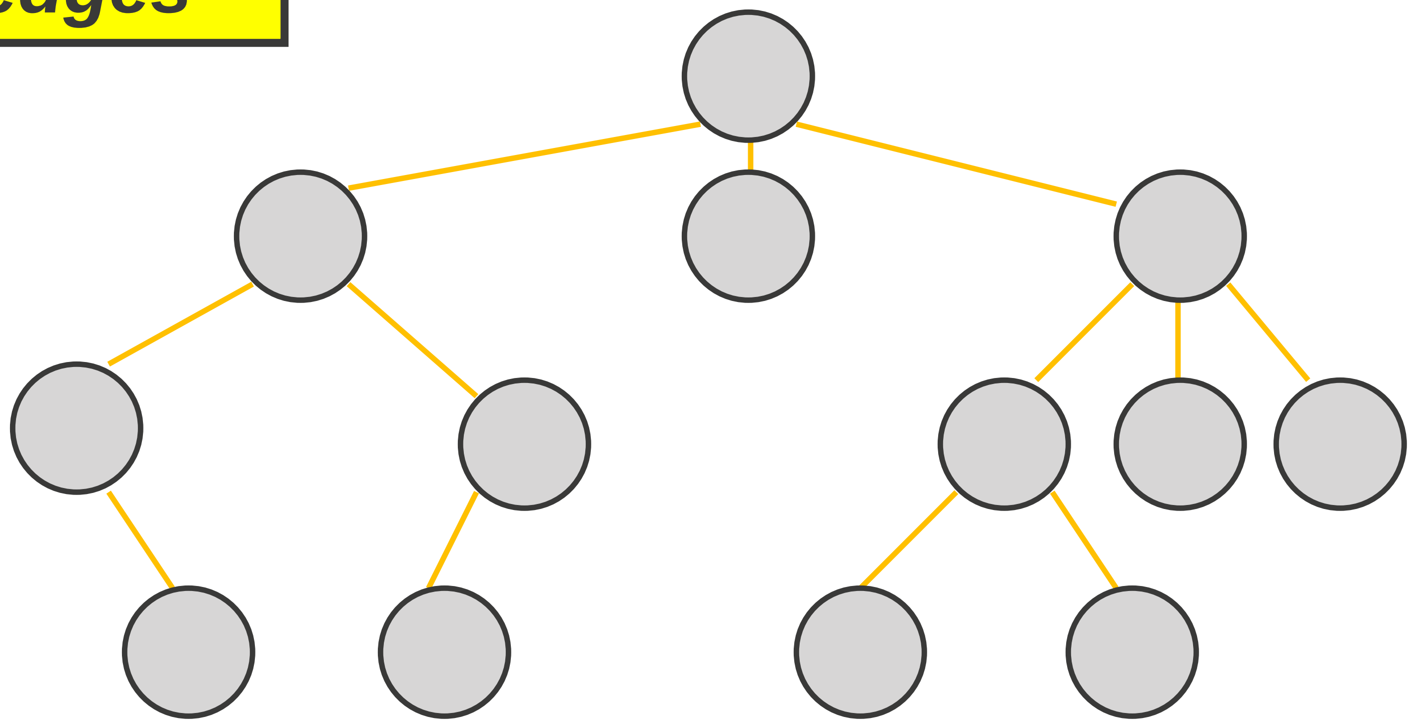
Tree Notation

edges



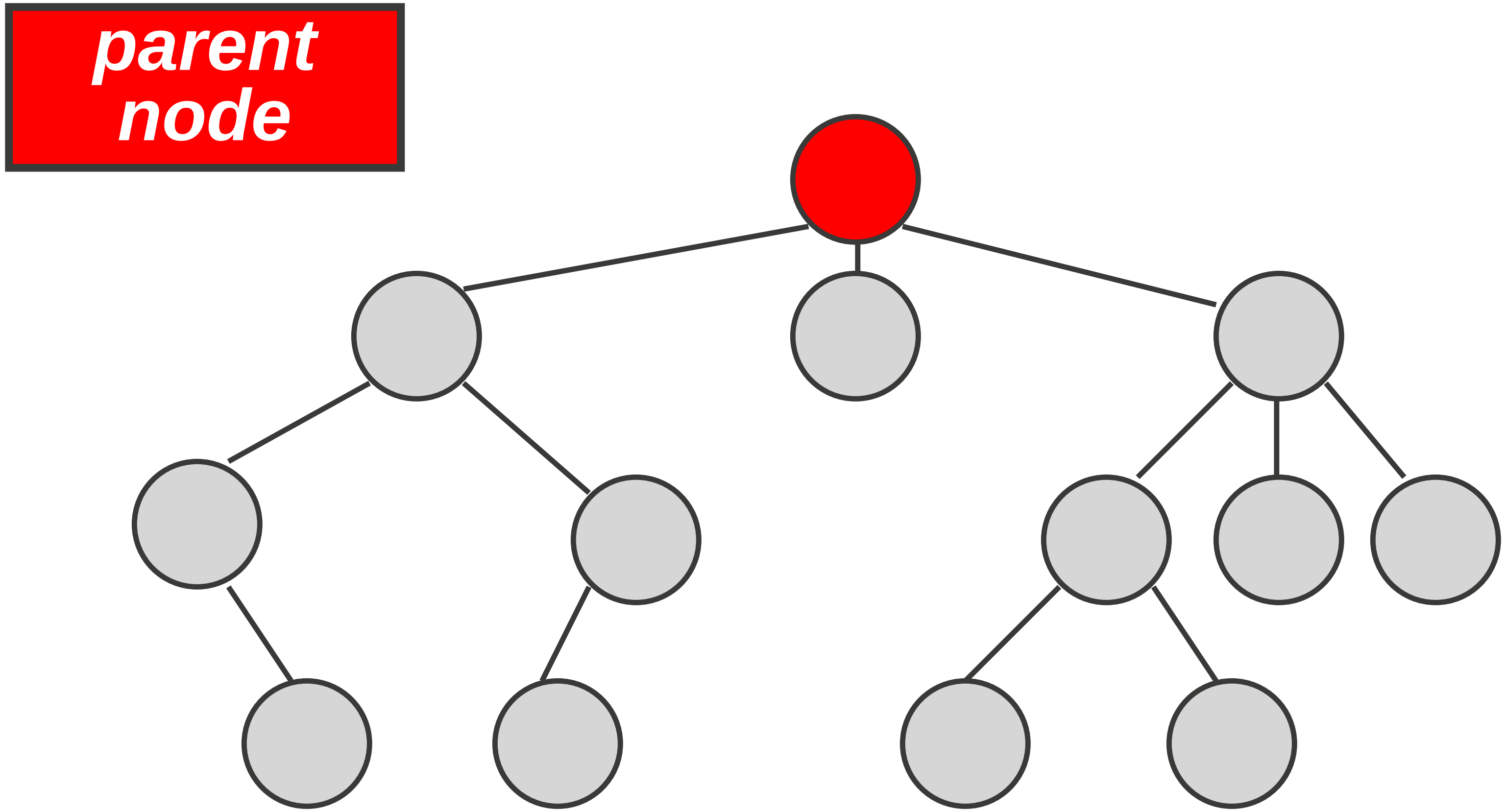
Tree Notation

edges

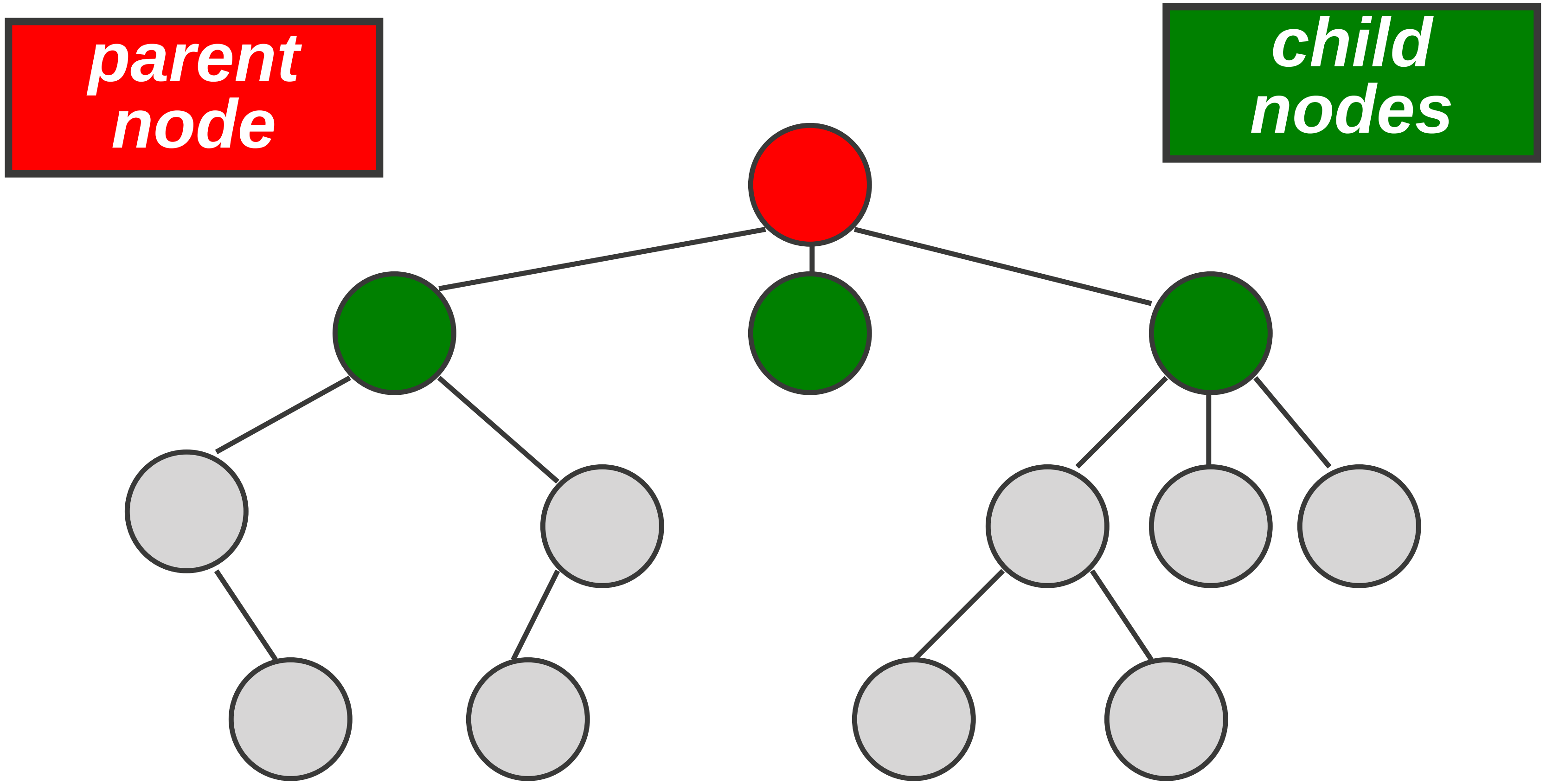


For simplicity we will draw undirected edges (no arrows) but in reality they are directed

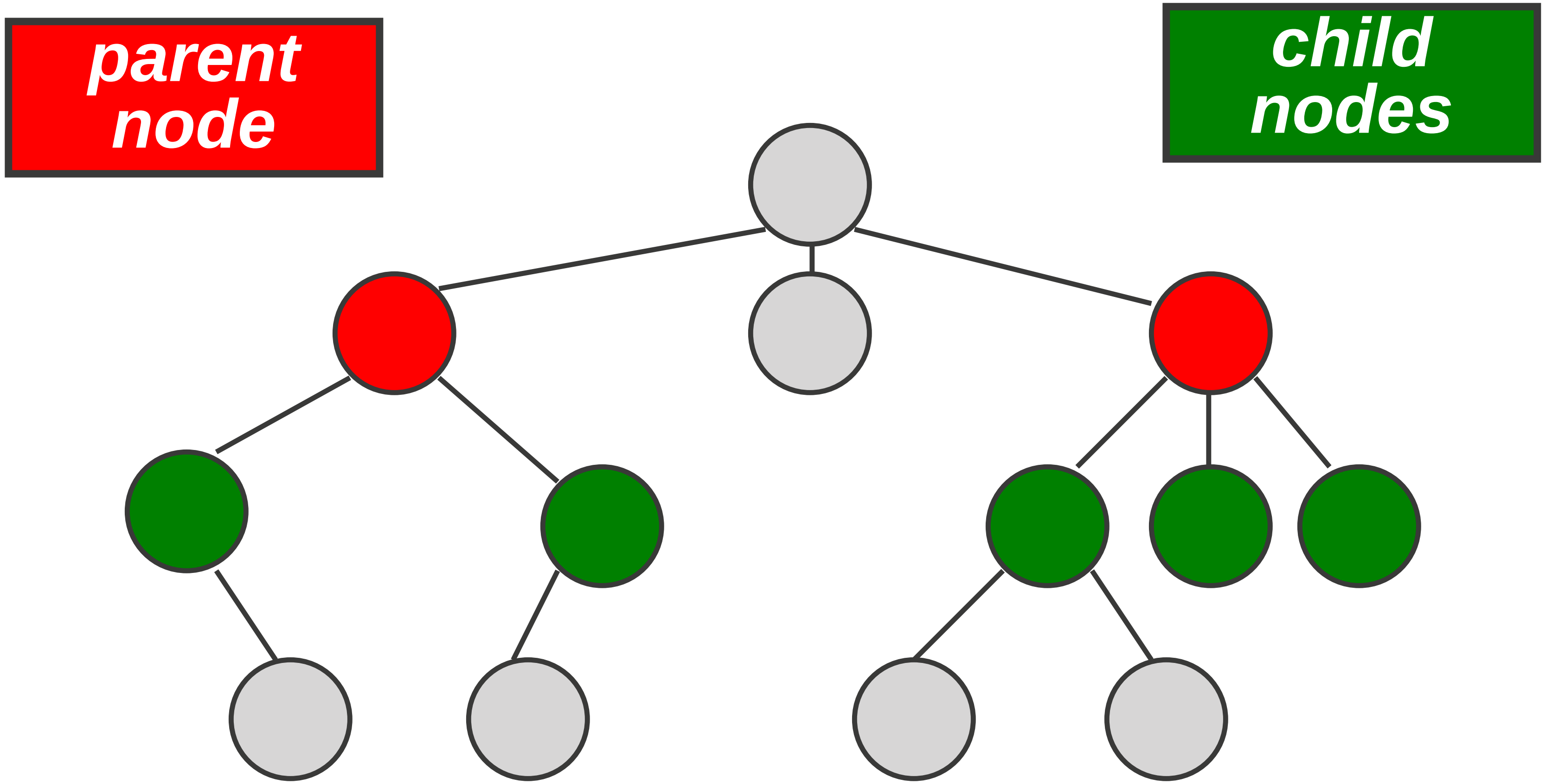
Tree Notation



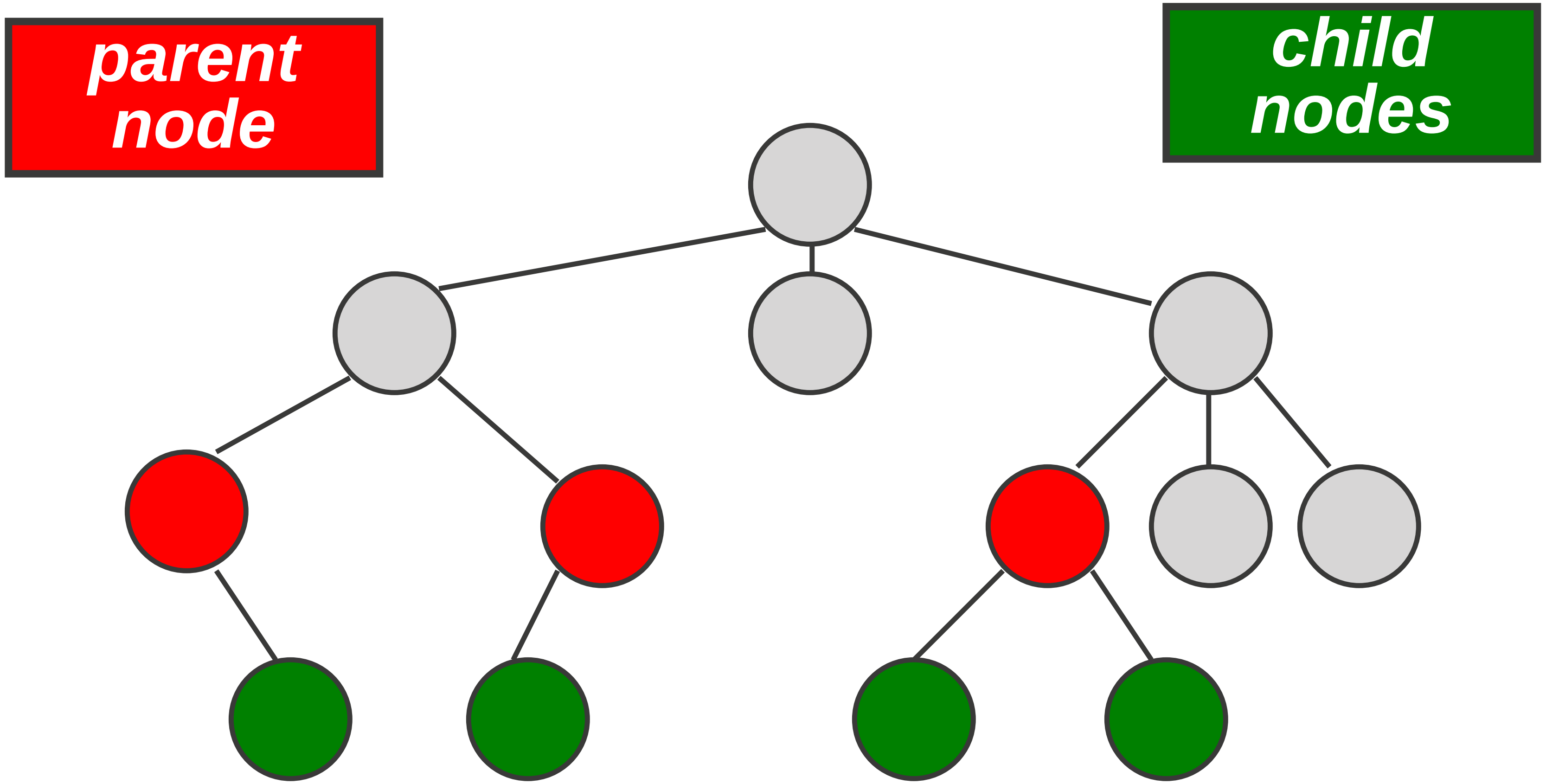
Tree Notation



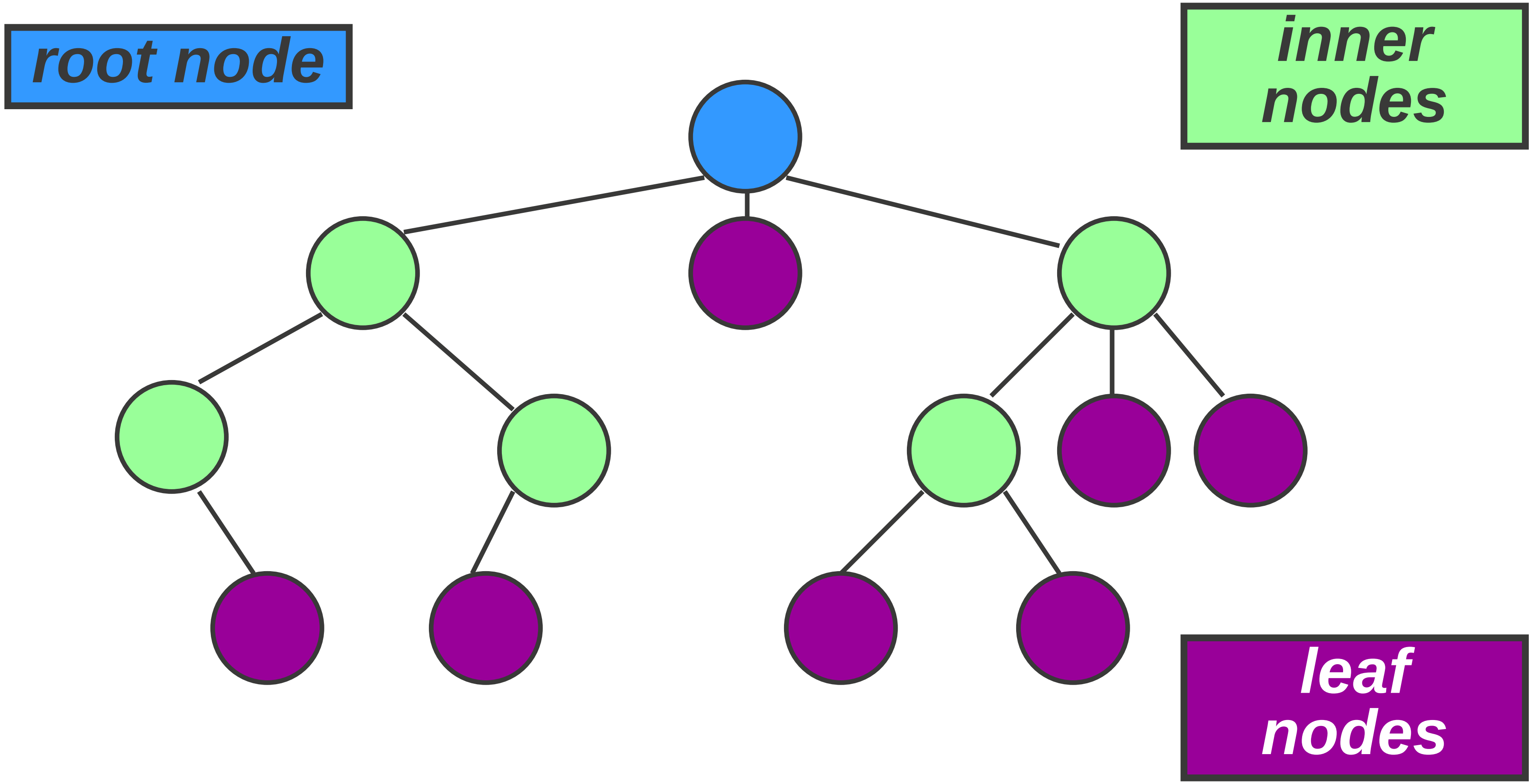
Tree Notation



Tree Notation

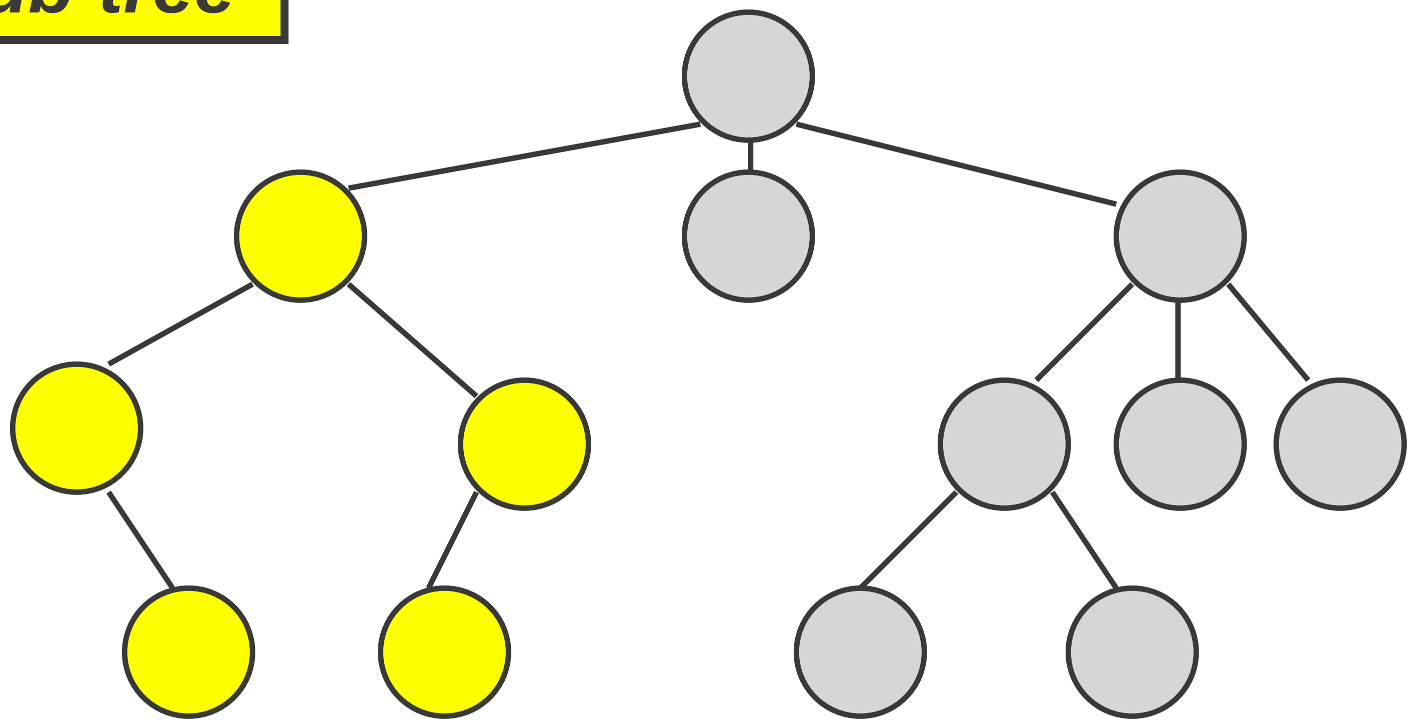


Tree Notation

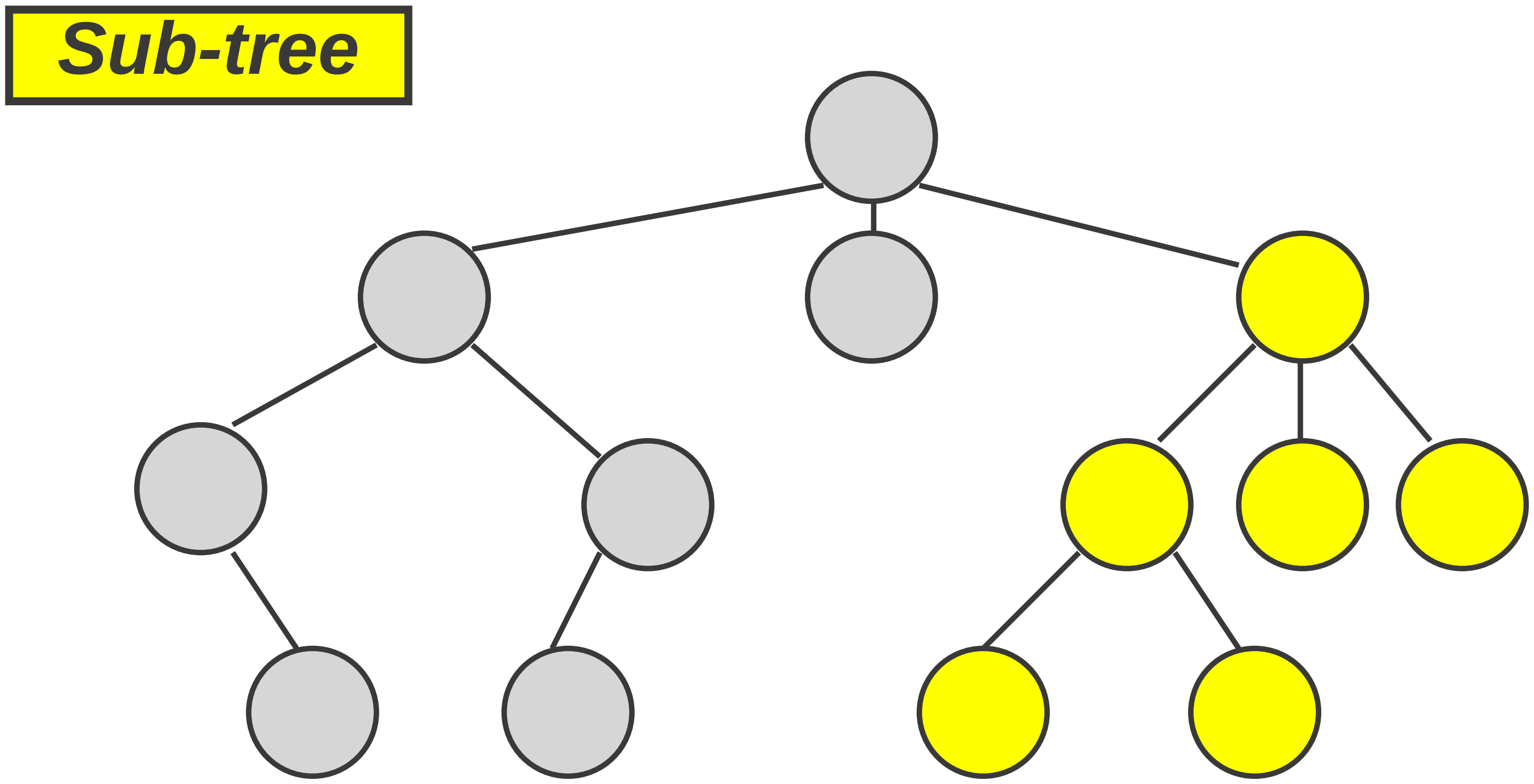


Tree Notation

Sub-tree

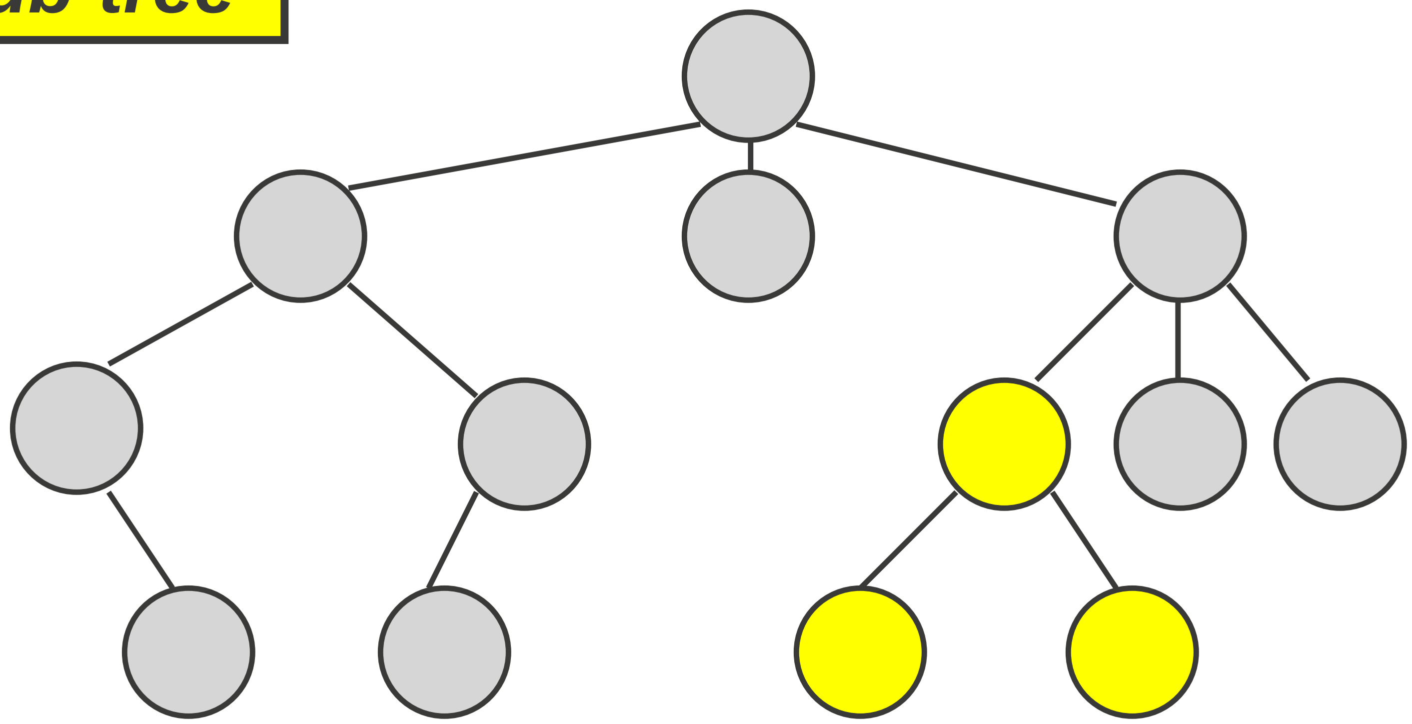


Tree Notation



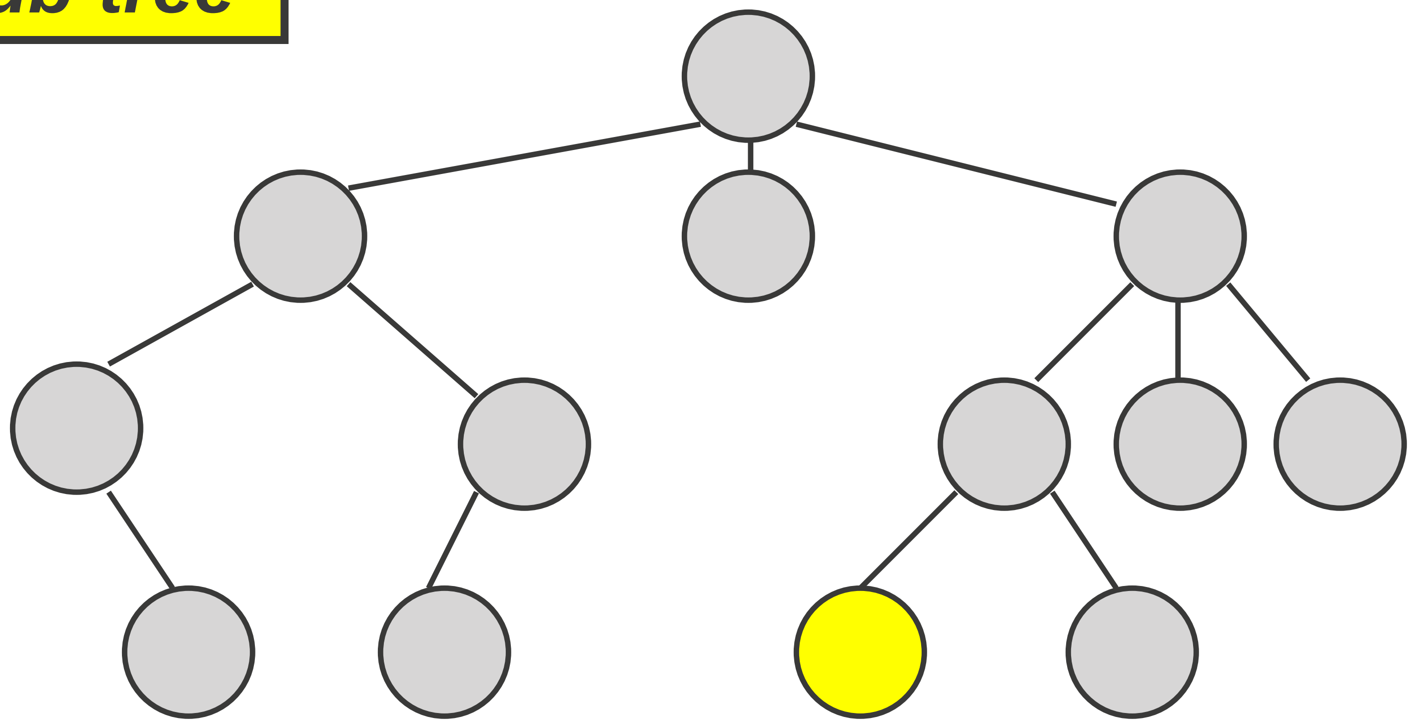
Tree Notation

Sub-tree



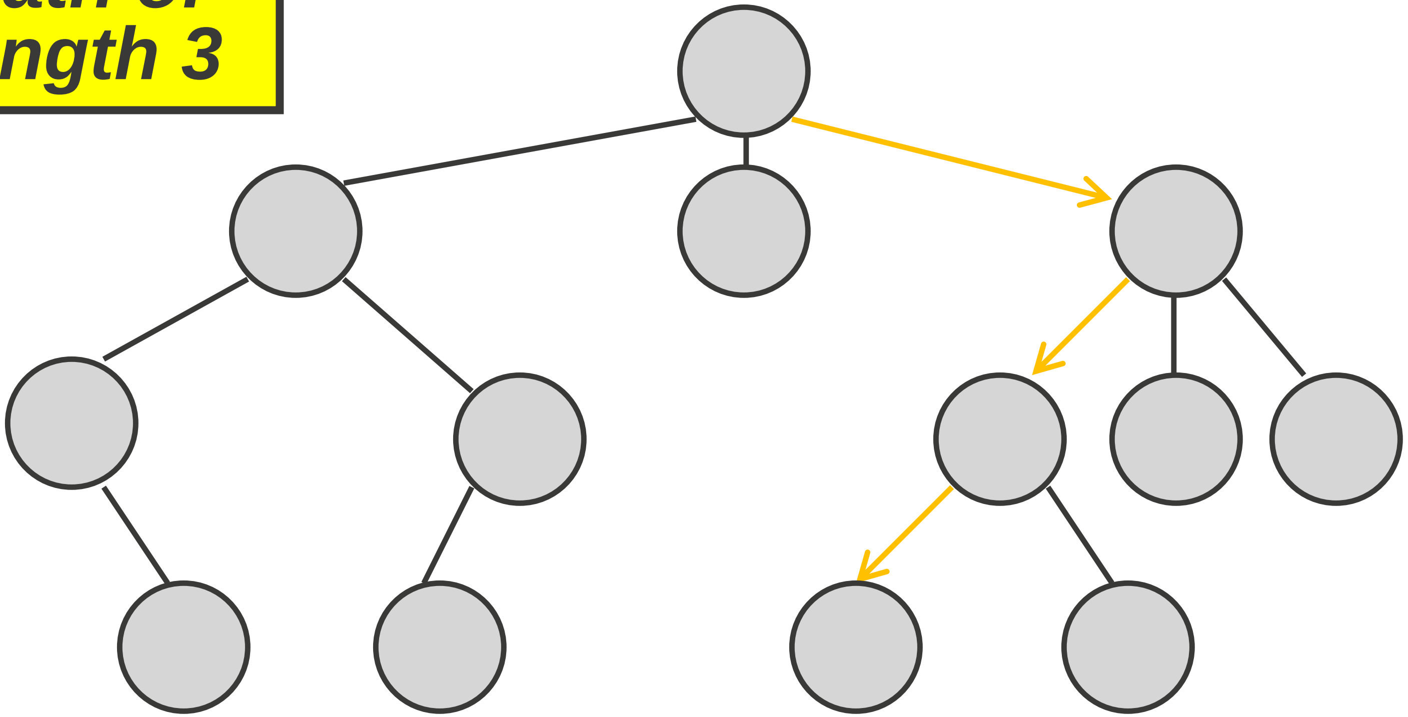
Tree Notation

Sub-tree



Tree Notation

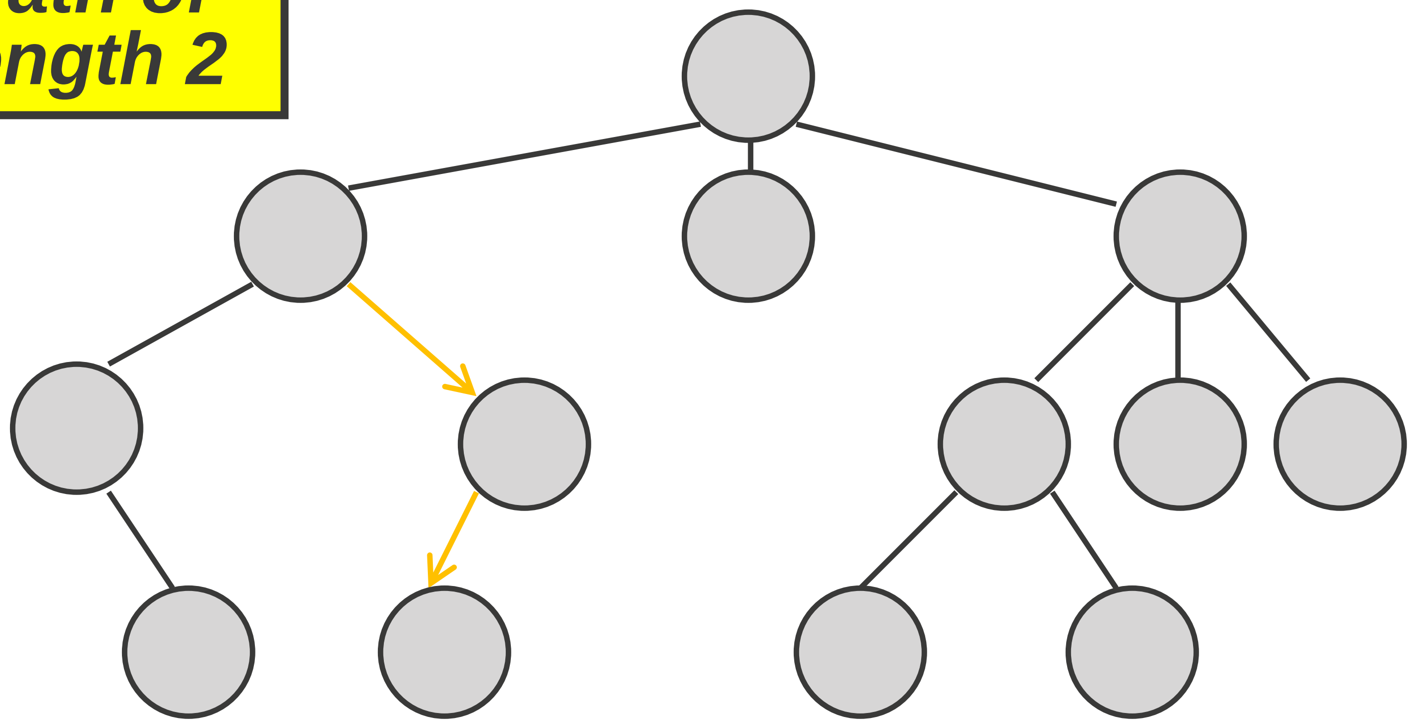
Path of length 3



The path is directed: follows the direction of the arrows (which I have again shown for clarity)

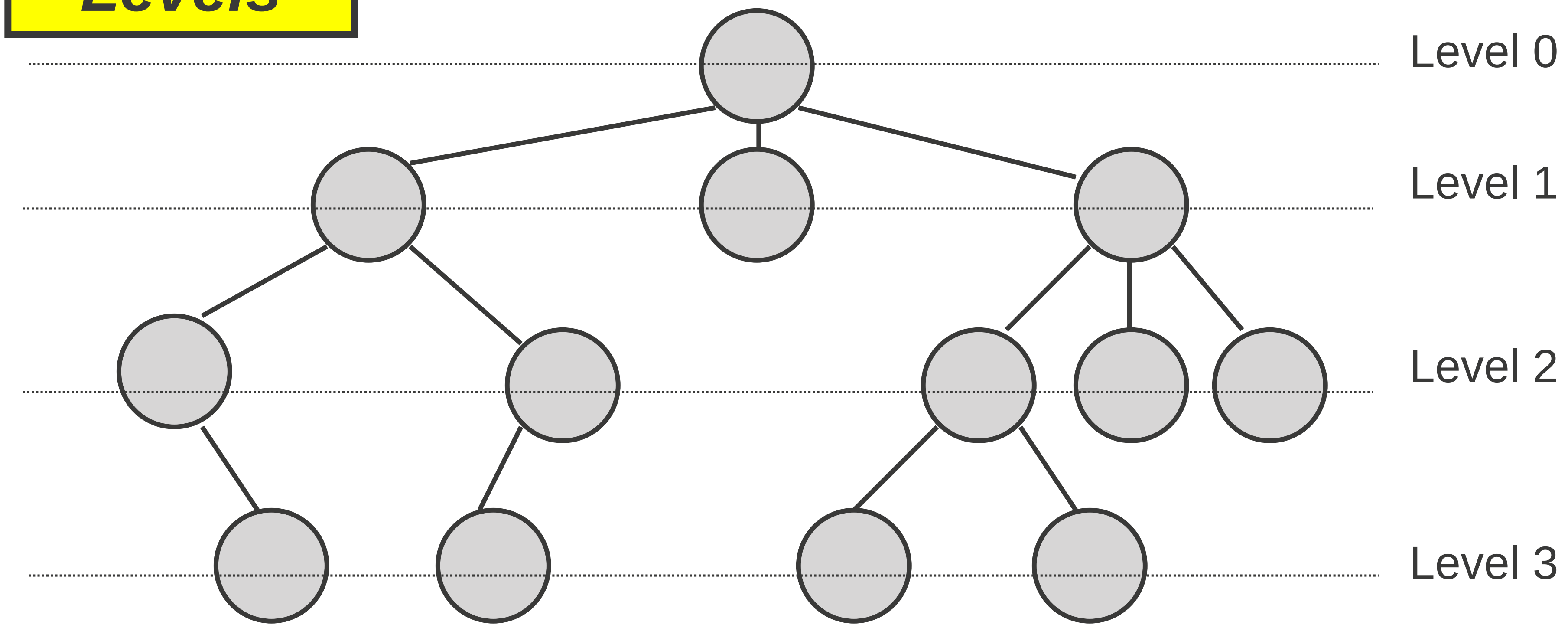
Tree Notation

Path of length 2



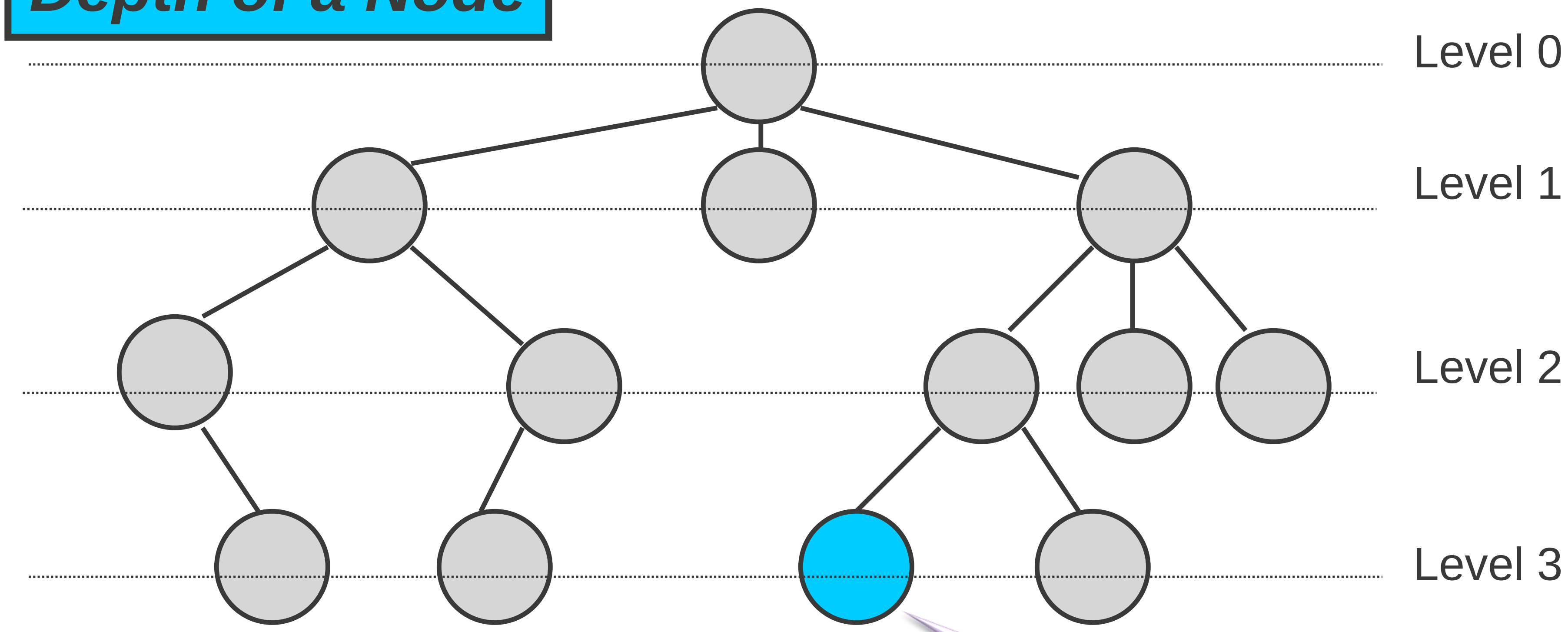
Tree Notation

Levels



Tree Notation

Depth of a Node

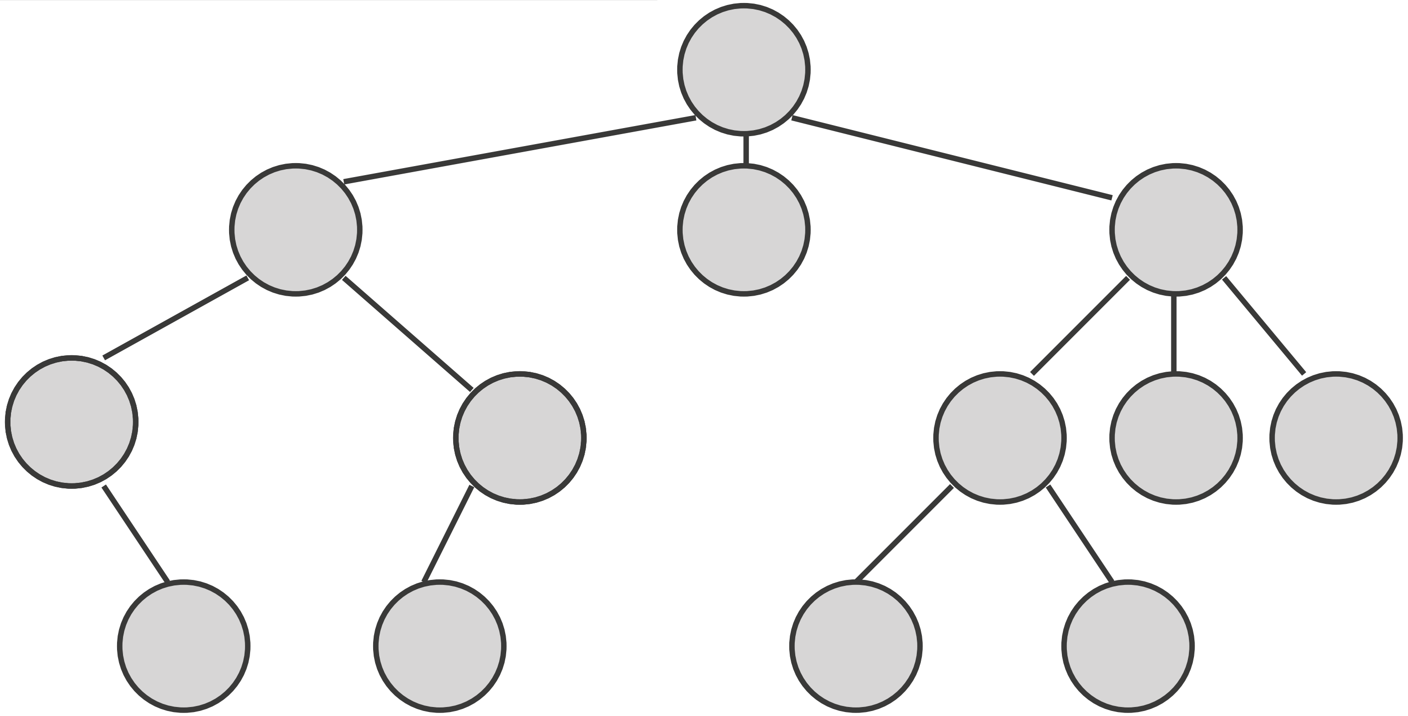


Equal to its level. Also defined as the distance (length of path) from the root.

In this case = 3

Tree Notation

Height/Depth of a tree

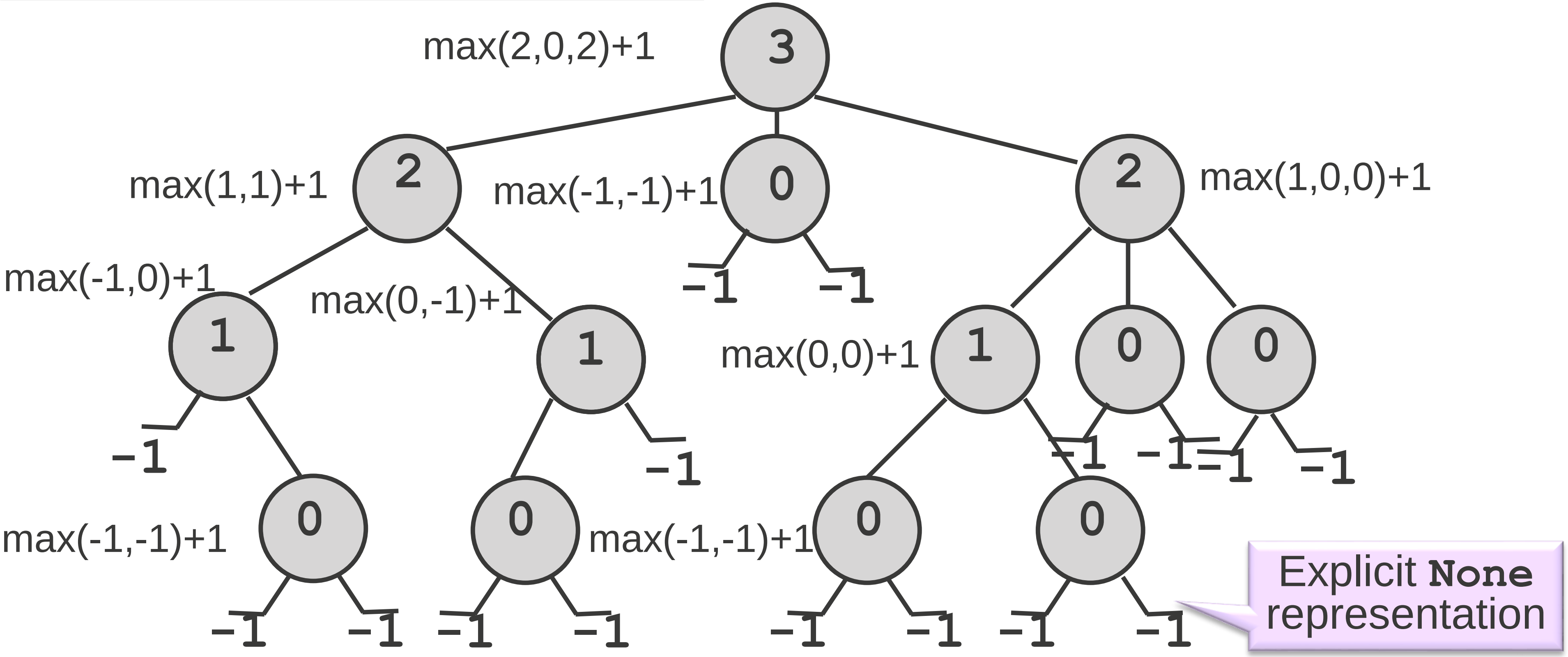


Max length of a path from the root. Recursively computed as the max height of its nodes.

Base case: -1 for empty (None) nodes. For the rest, we take the max height of its children + 1

Tree Notation

Height/Depth of a tree

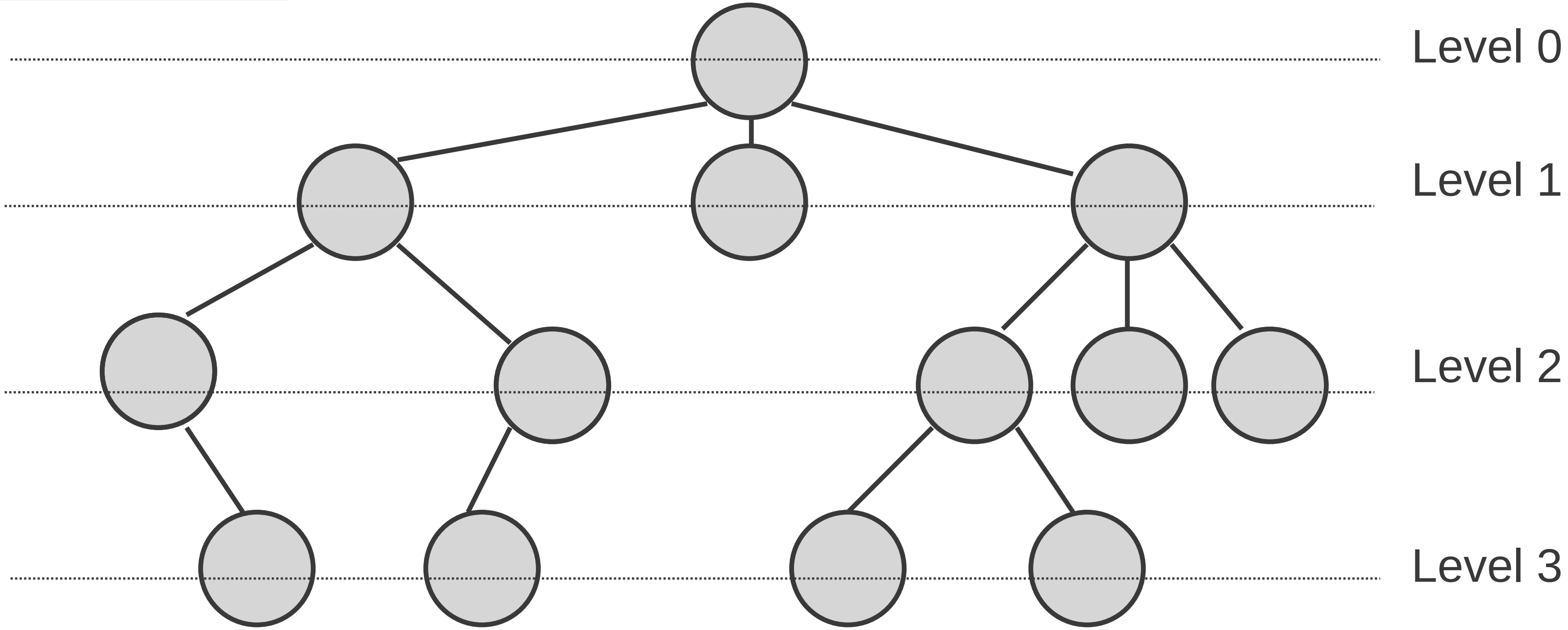


Max length of a path from the root. Recursively computed as the max height of its nodes.

Base case: -1 for empty (None) nodes. For the rest, we take the max height of its children + 1

Tree Notation

Width



Number of nodes in the level with the most nodes

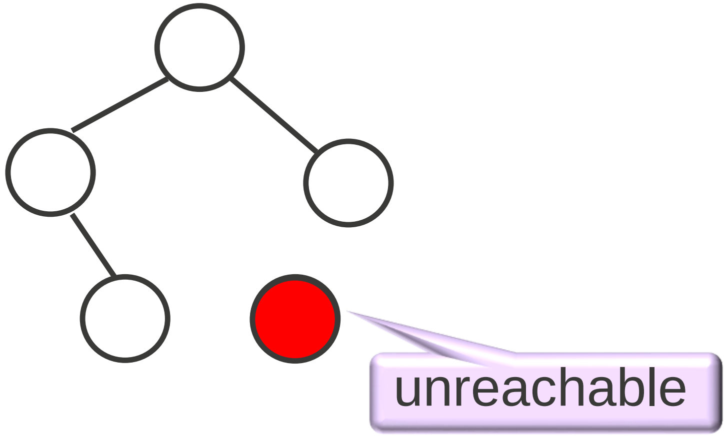
In this case 5

Summary of Tree Notation

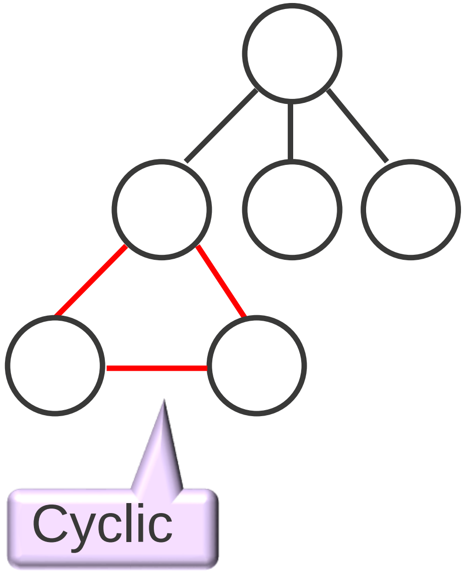
- The node with no parent is the **root** (one per tree)
- A node with no child is a **leaf**
- Each node is either an **inner** node, or it is a root and/or leaf
- Every node that is not a leaf is a **parent** node
- Every node is the root node of its **subtree**
- Every node except the root is a **child**
- **Height/Depth** of a tree is also its maximum level
- **Width**: number of nodes in the level with the highest number of nodes

Defining trees more formally

- **A graph is composed of a:**
  - Set V of vertices (or nodes)
  - Set E of edges, where each element of E is a pair of nodes in V
- **In a connected graph, there is a path between every pair of nodes (i.e., there are no unreachable nodes)**



- **In an acyclic graph there are no cycles (i.e., no path starts and ends at the same node)**



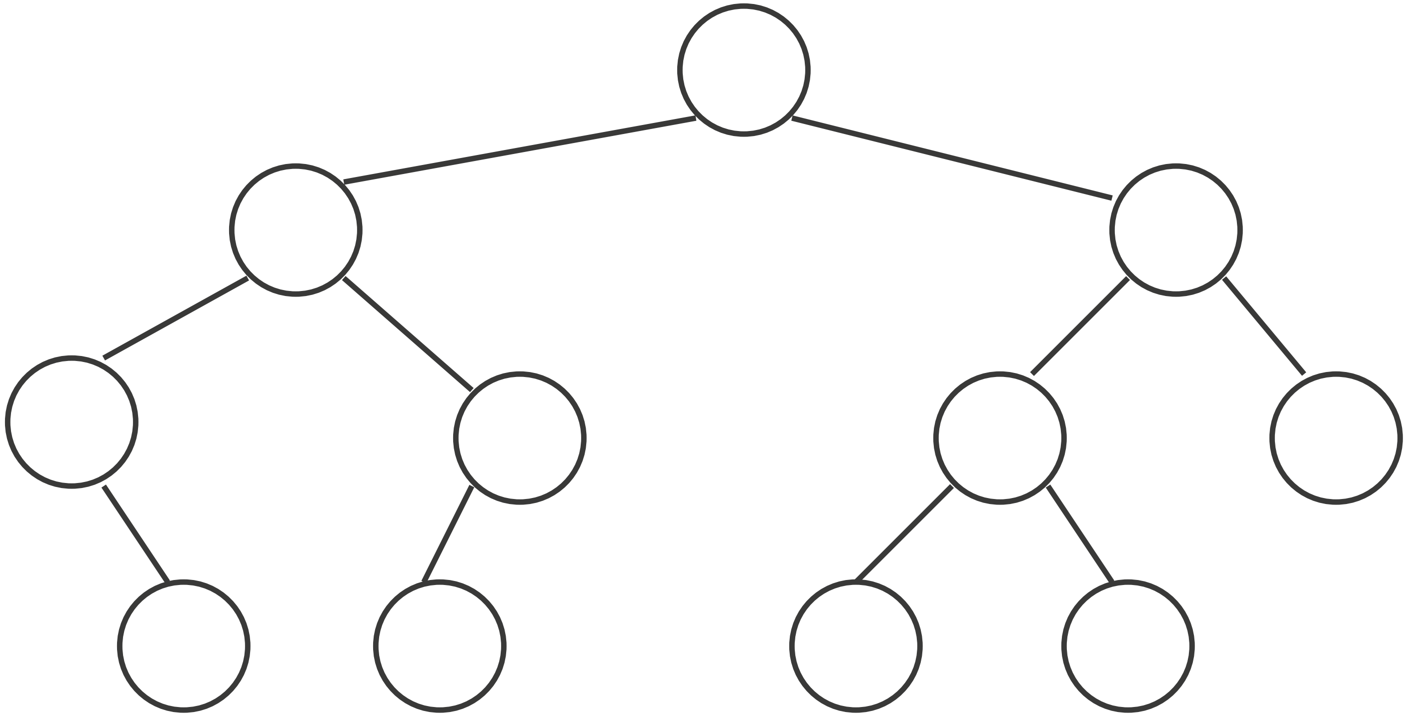
Defining trees more formally (cont)

- **In maths, a tree is any acyclic, connected graph**
- **In CS, we look at rooted trees, where one node is marked as the root and:**
  - Edges do have a direction (from parent to child)
  - Sibling nodes might also have an order (left-to-right)

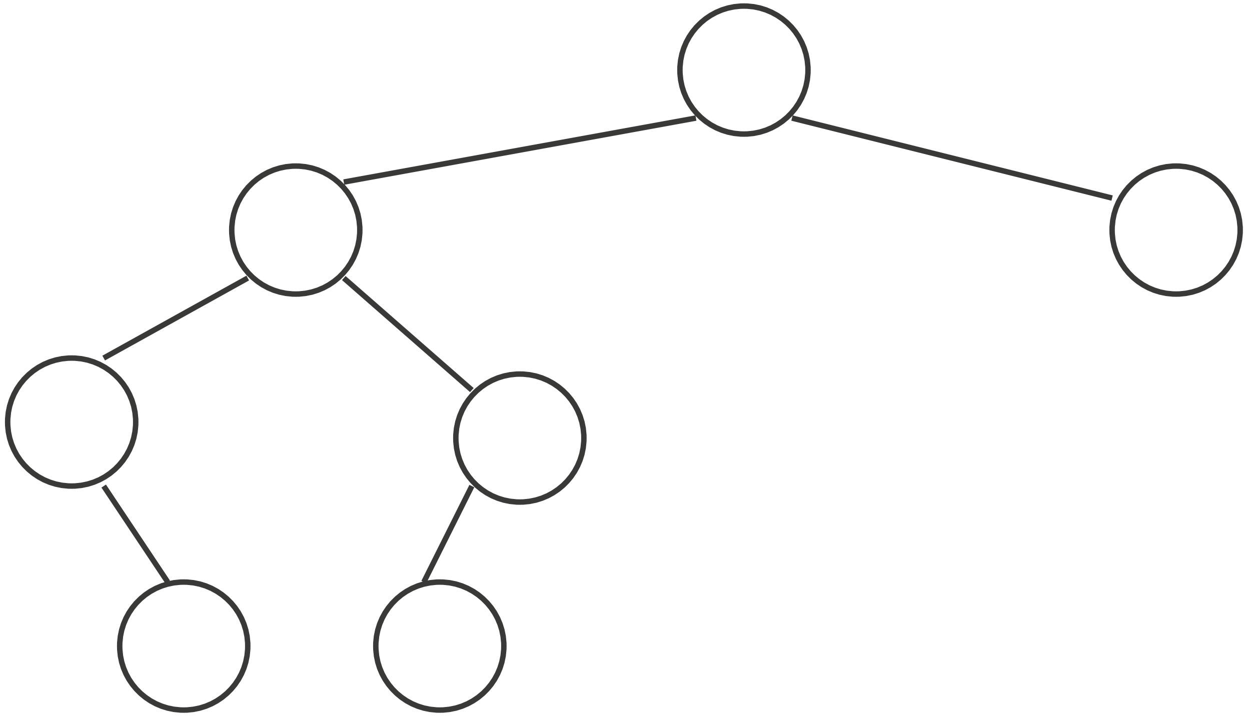
Binary Tree

Binary Tree

- **Each node can have at most 2 children**

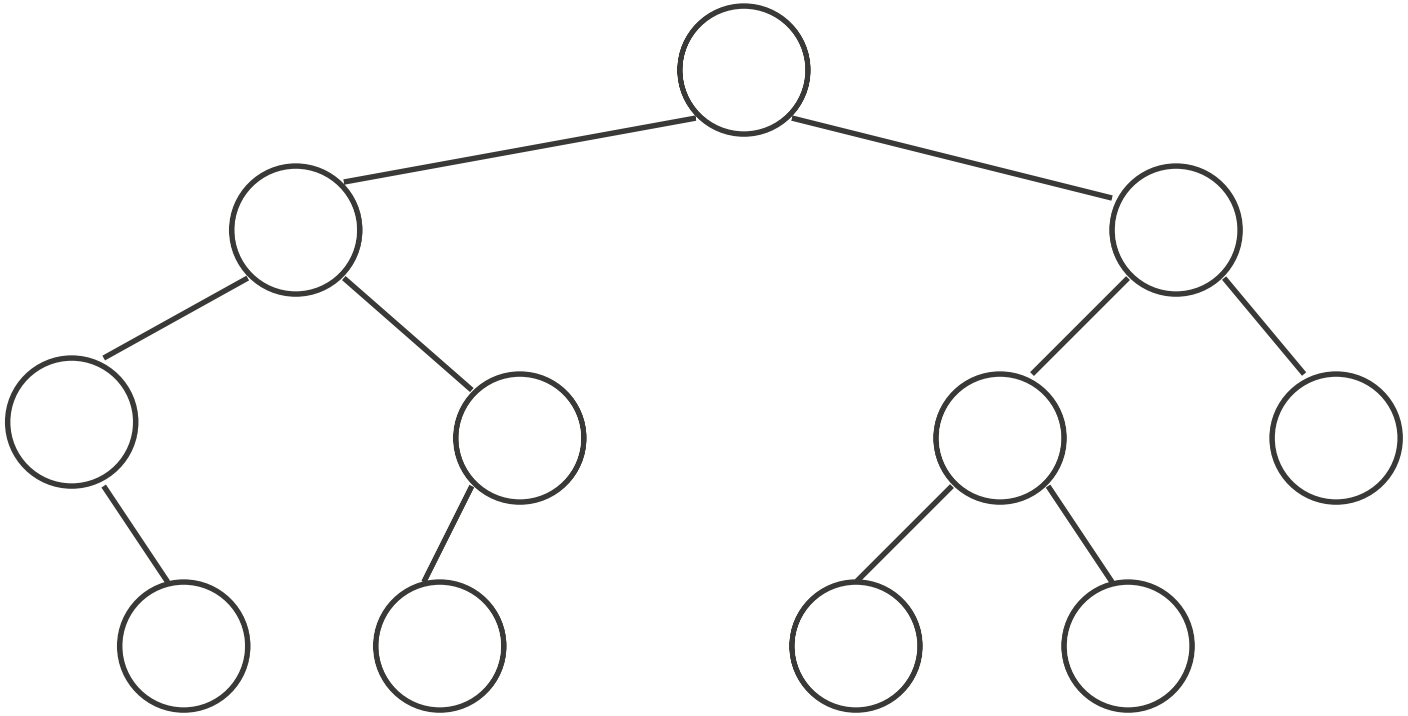


Unbalanced Binary Tree



A binary tree is (height) balanced if, for every node, the difference between the height of the left subtree and that of the right subtree is at most 1

Balanced Binary Tree

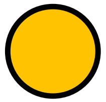


A binary tree is (height) balanced if, for every node, the difference between the height of the left subtree and that of the right subtree is at most 1

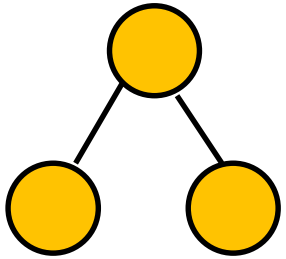
Balanced Trees

- **There are several definitions for a balanced tree**
- **The one we have seen is called height-balanced**
  - It is based on the height of the sub-trees
- **There are others like weight-balanced**
  - Based on the size (number of nodes) of the subtree
- **We will focus on height-balanced trees and simply called them balanced trees**
- **But do keep in mind there are other kinds of balanced trees**

Perfect Binary Trees



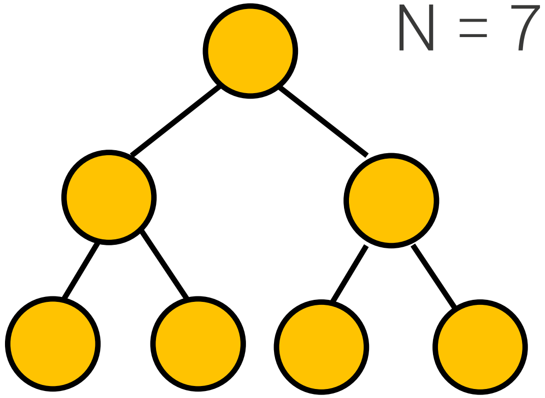
N = 1 Height = 0



N = 3 Height = 1

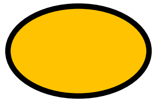
All parents have two children

All leaves are at the same level



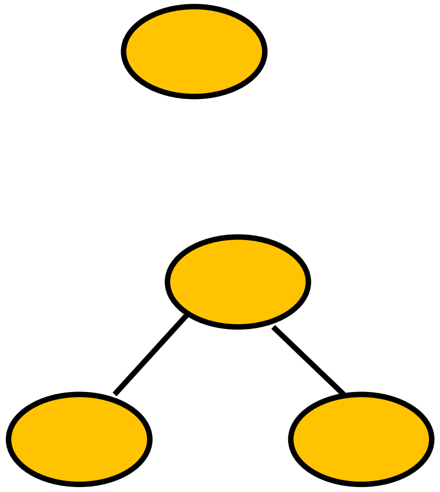
N = 7 Height = 2

Perfect Binary Trees



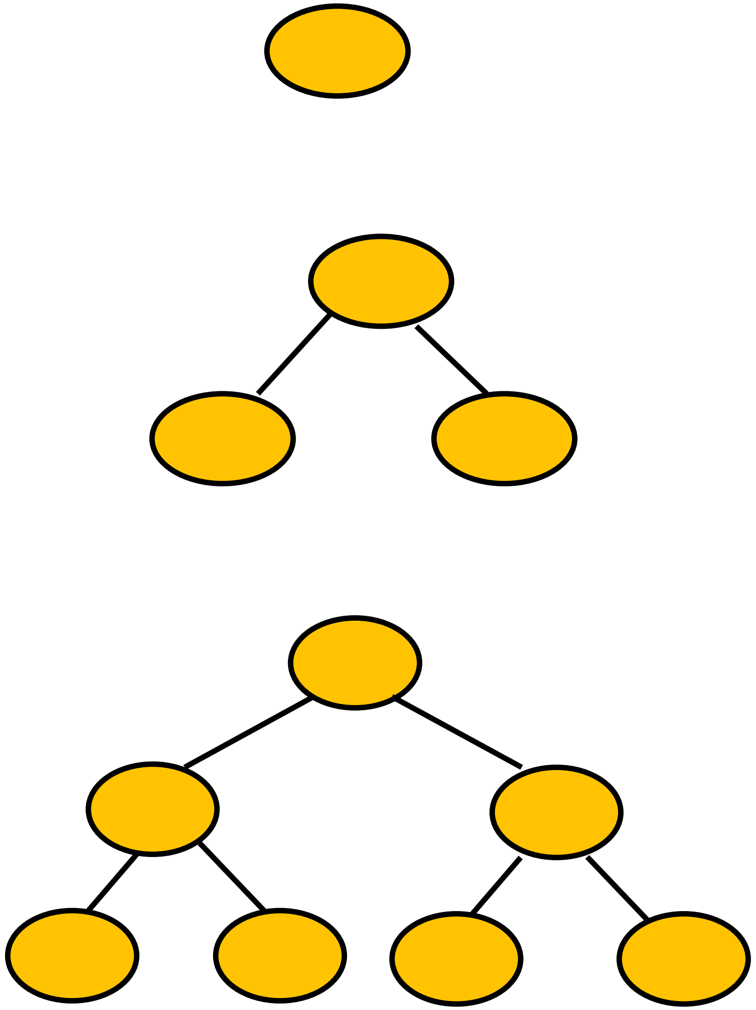
height	leaves
0	1

Perfect Binary Trees



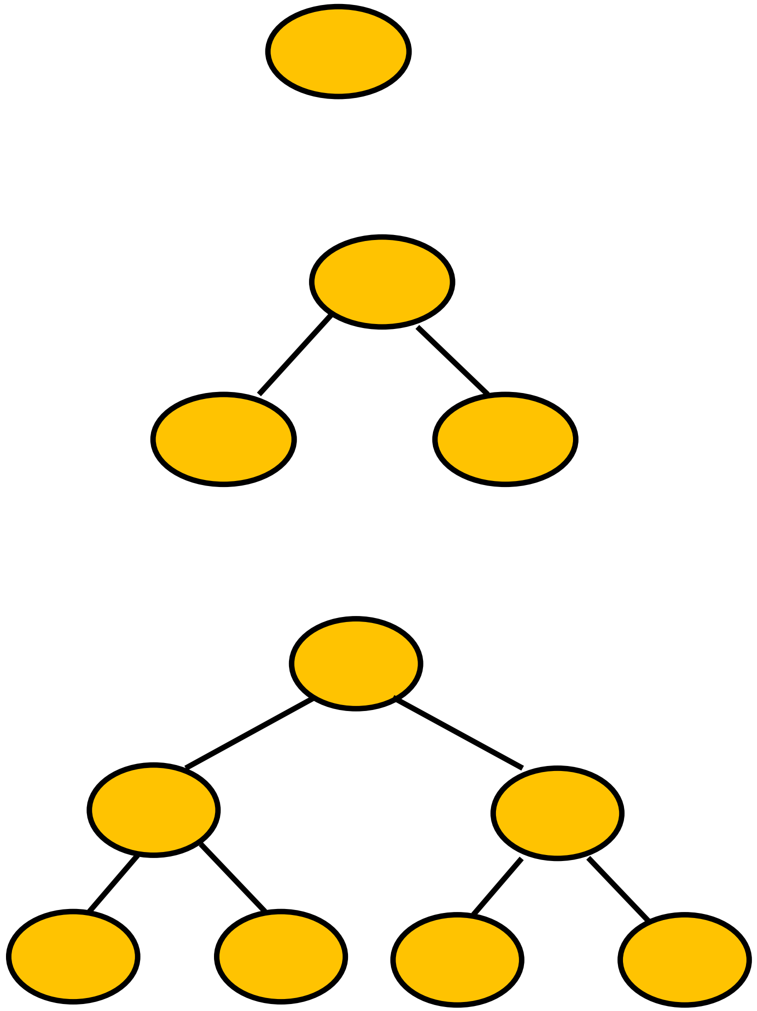
height	leaves
0	1
1	2

Perfect Binary Trees



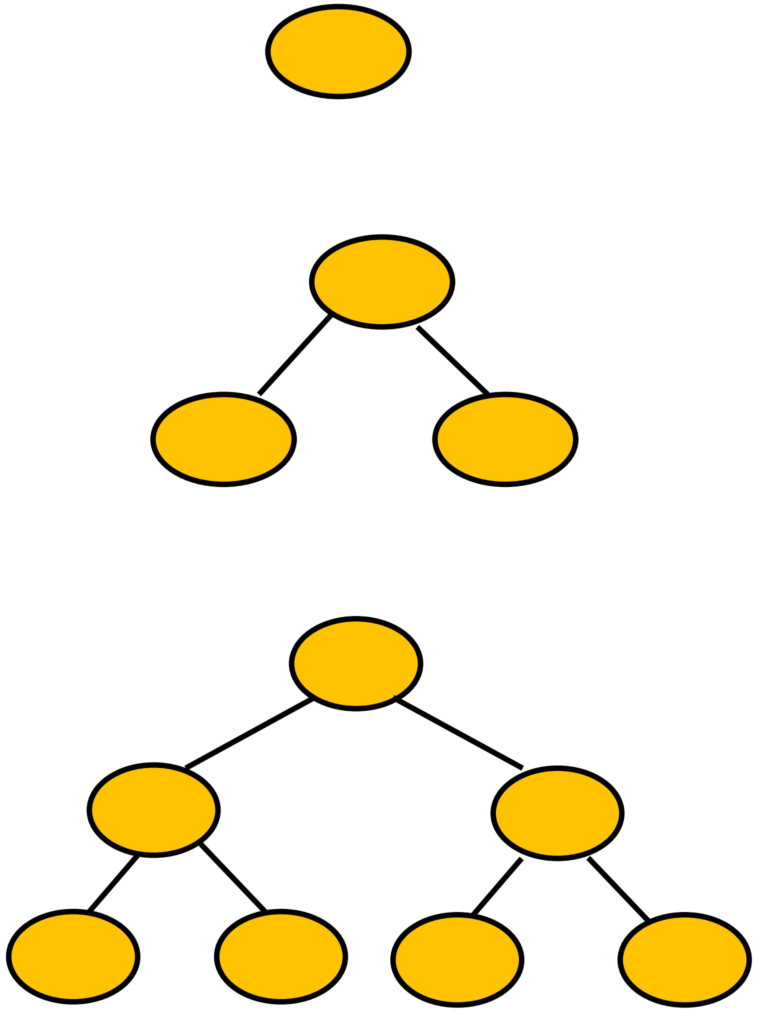
height	leaves
0	1
1	2
2	4

Perfect Binary Trees



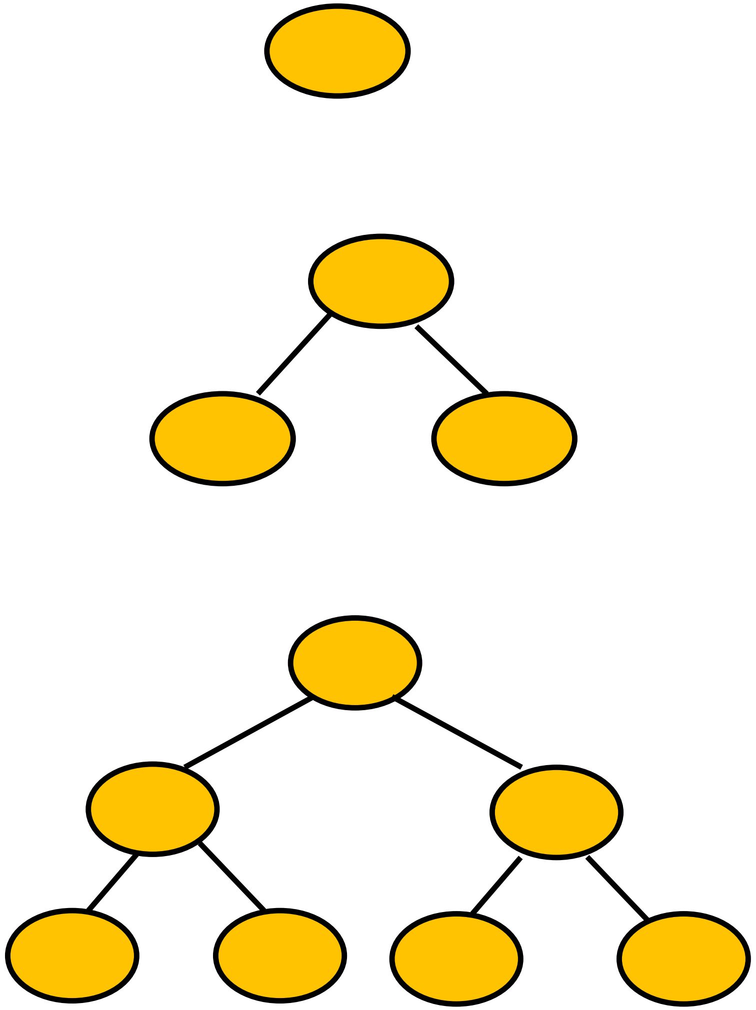
height	leaves
0	1
1	2
2	4
3	8

Perfect Binary Trees



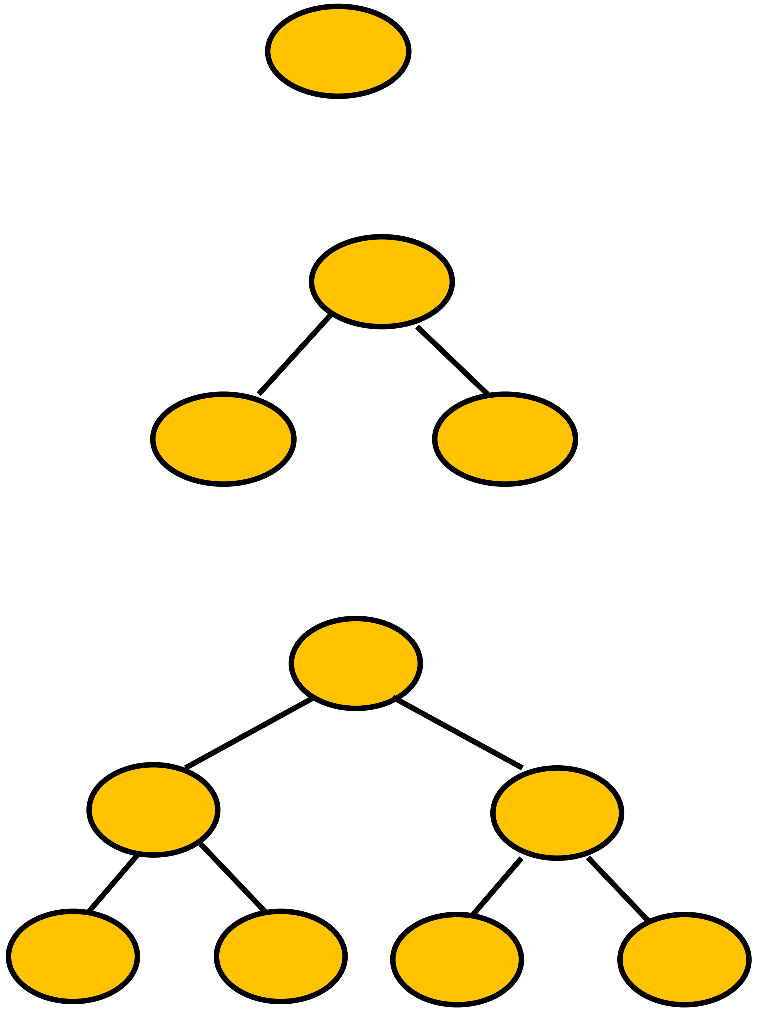
height	leaves
0	1
1	2
2	4
3	8
k	**2^k**

Perfect Binary Trees



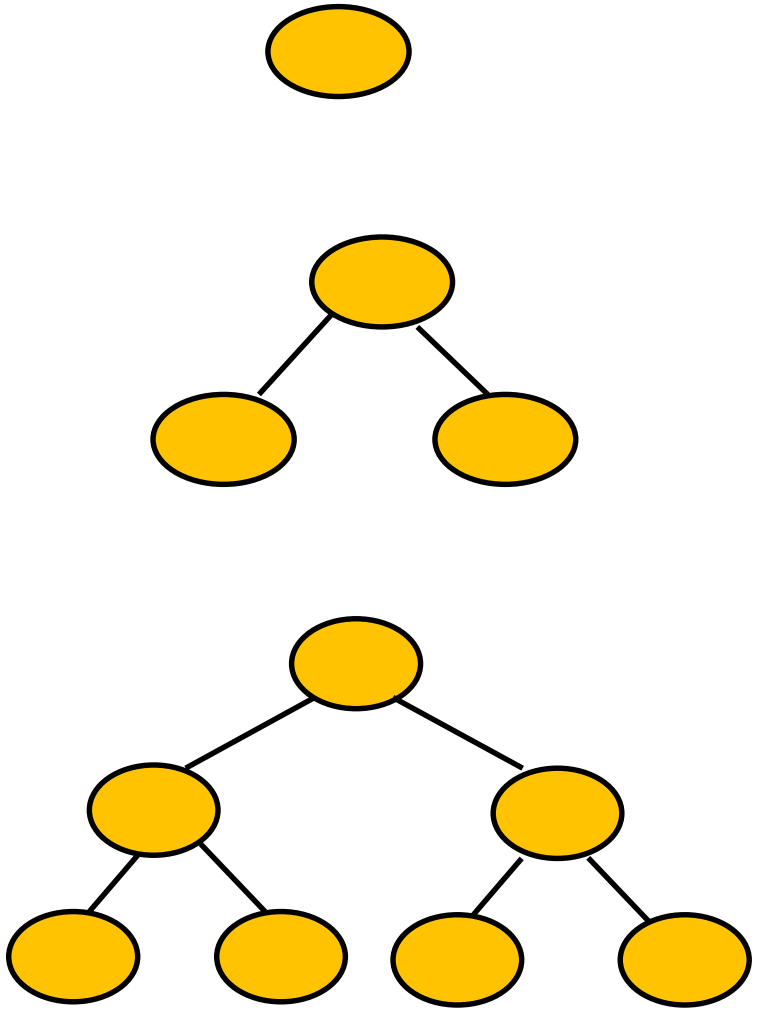
height	leaves	nodes
0	1	
1	2	
2	4	
3	8	
k	$\mathbf{2^k}$	

Perfect Binary Trees



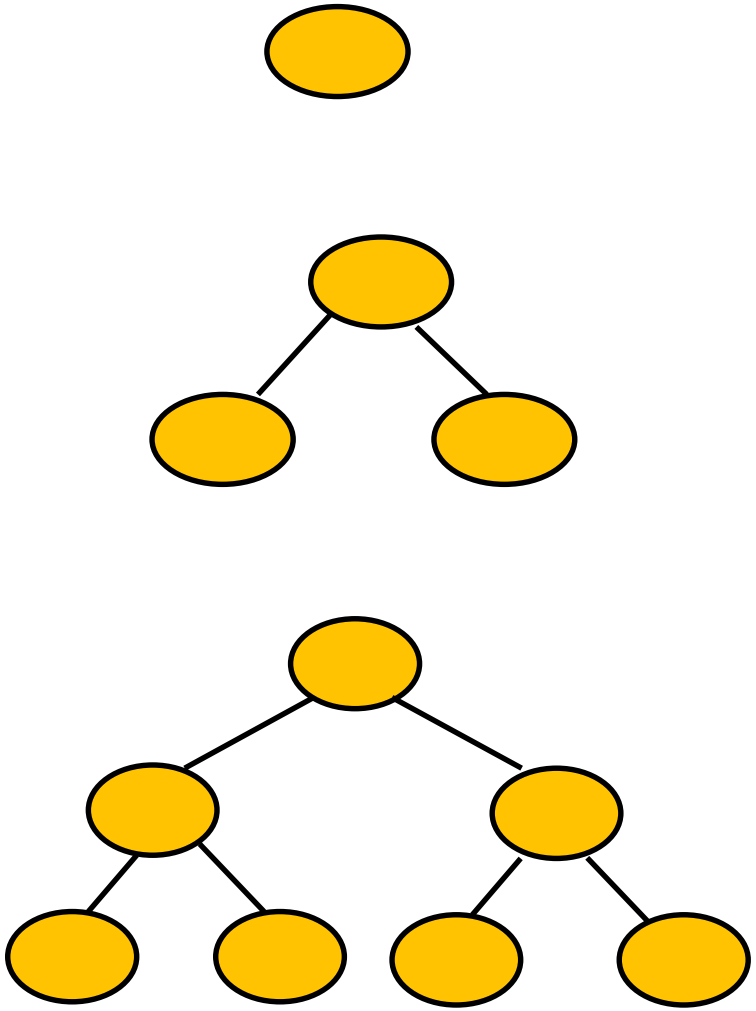
height	leaves	nodes
0	1	1
1	2	3
2	4	7
3	8	
k	**2^k**	

Perfect Binary Trees



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

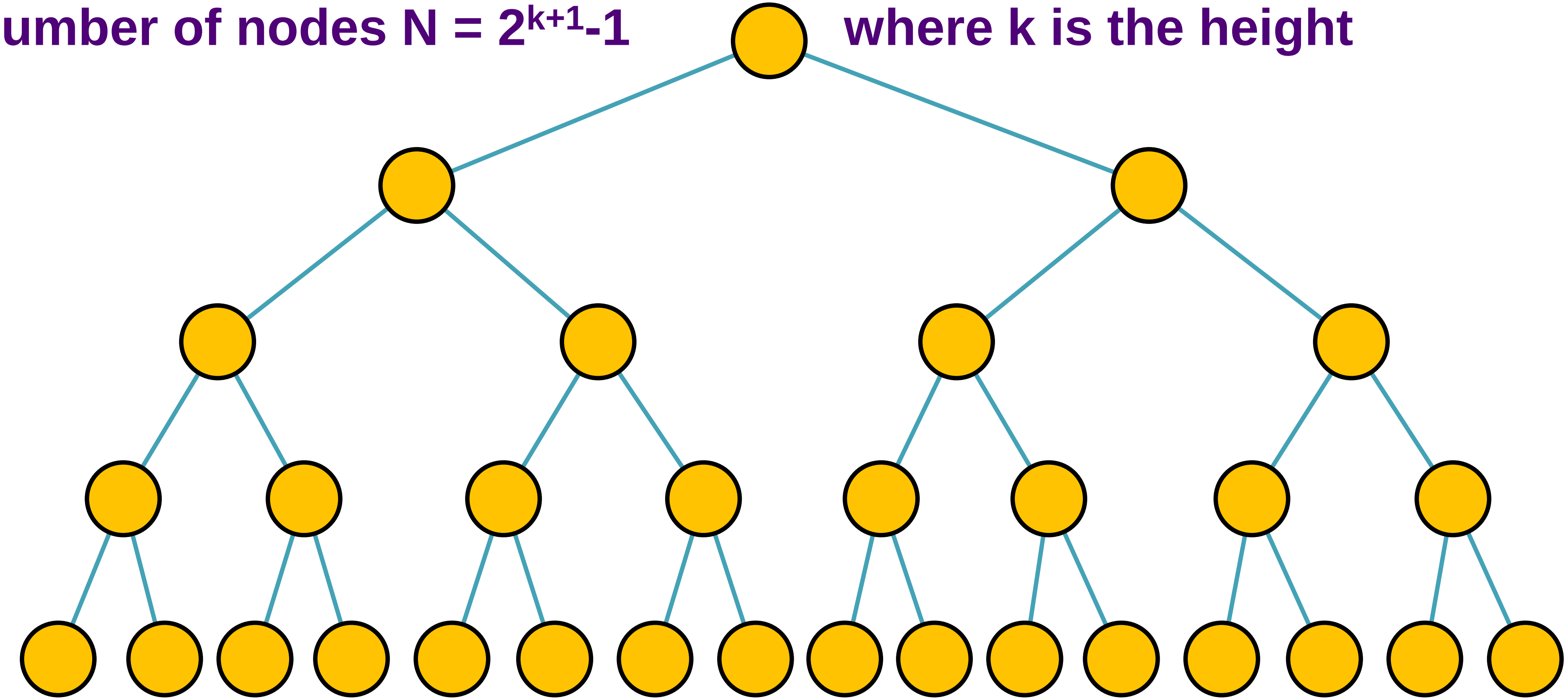
Perfect Binary Trees



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

Perfect Binary Trees

- **Number of nodes $N = 2^{k+1}-1$ where k is the height**



Perfect Binary Trees

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

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

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

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

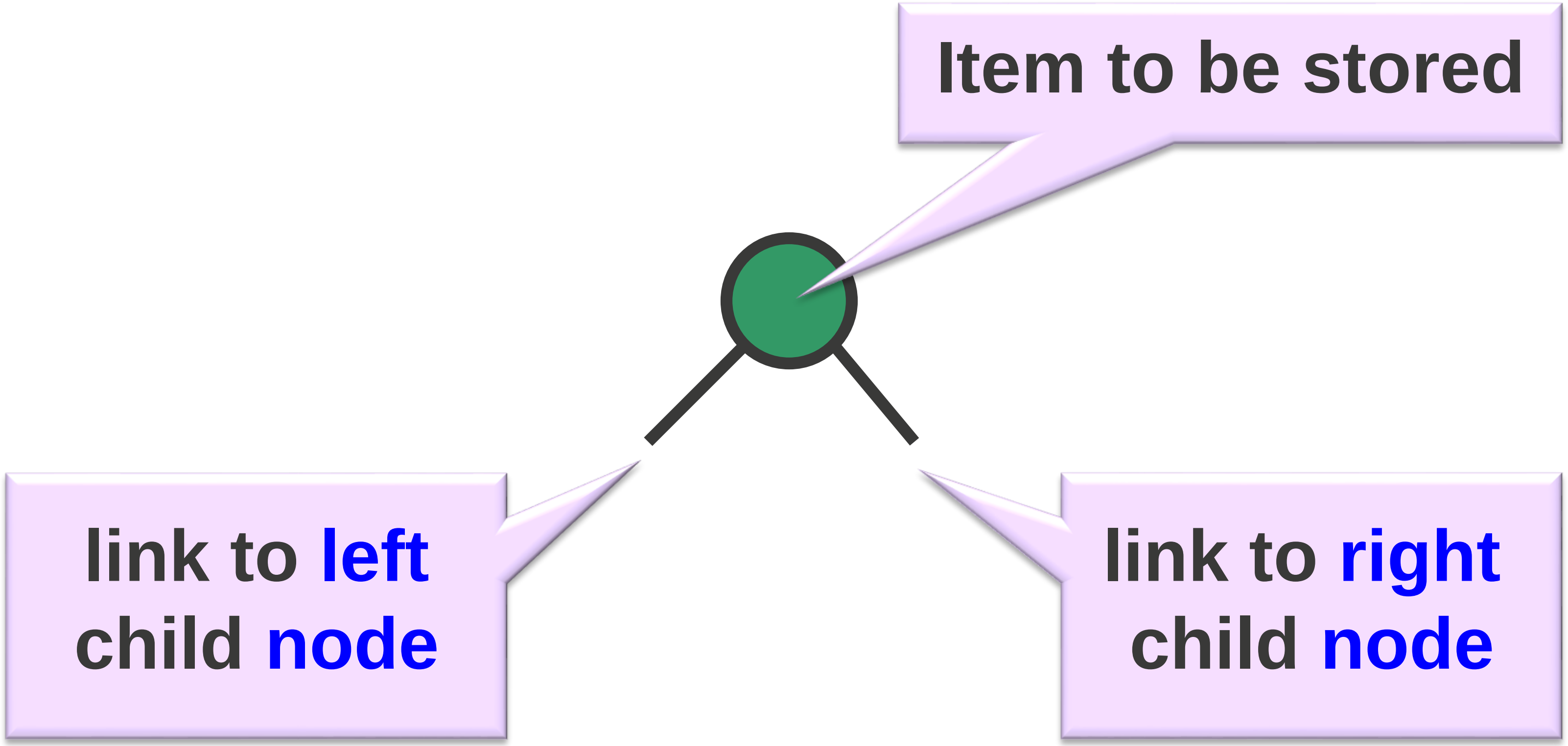
In a perfect binary tree with N nodes, the height is O(logN)

So when we talk about complexity…

For a balanced tree the height is **O(logN)**

For an unbalanced tree the height is **O(N)**

Representing a Binary Tree Node



Our representation: each link will point to a node

Possible class for Binary Trees

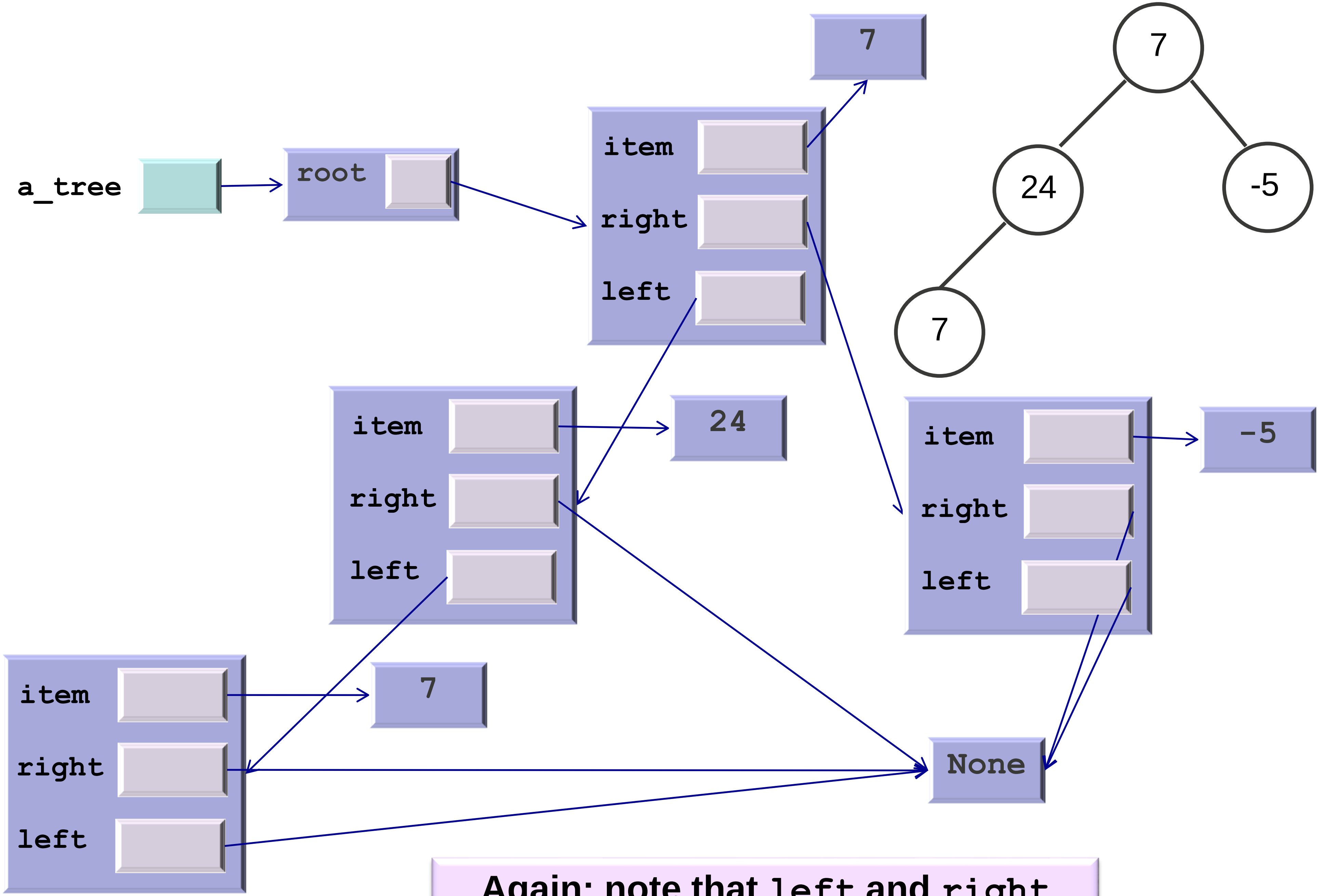
```
from typing import TypeVar, Generic, Callable
T = TypeVar('T')

class BinaryTreeNode(Generic[T]):
    def __init__(self, item: T = None) -> None:
        self.item = item
        self.left = None
        self.right = None

    def __str__(self) -> str:
        return str(self.item)

class BinaryTree(Generic[T]):
    def __init__(self) -> None:
        self.root = None

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

We will discuss later

First a class for binary tree nodes

Then one for the actual tree

The **`List`** class only contains a **`head`** reference to a node. Similarly, the **`BinaryTree`** class only contains a **`root`** reference to a node

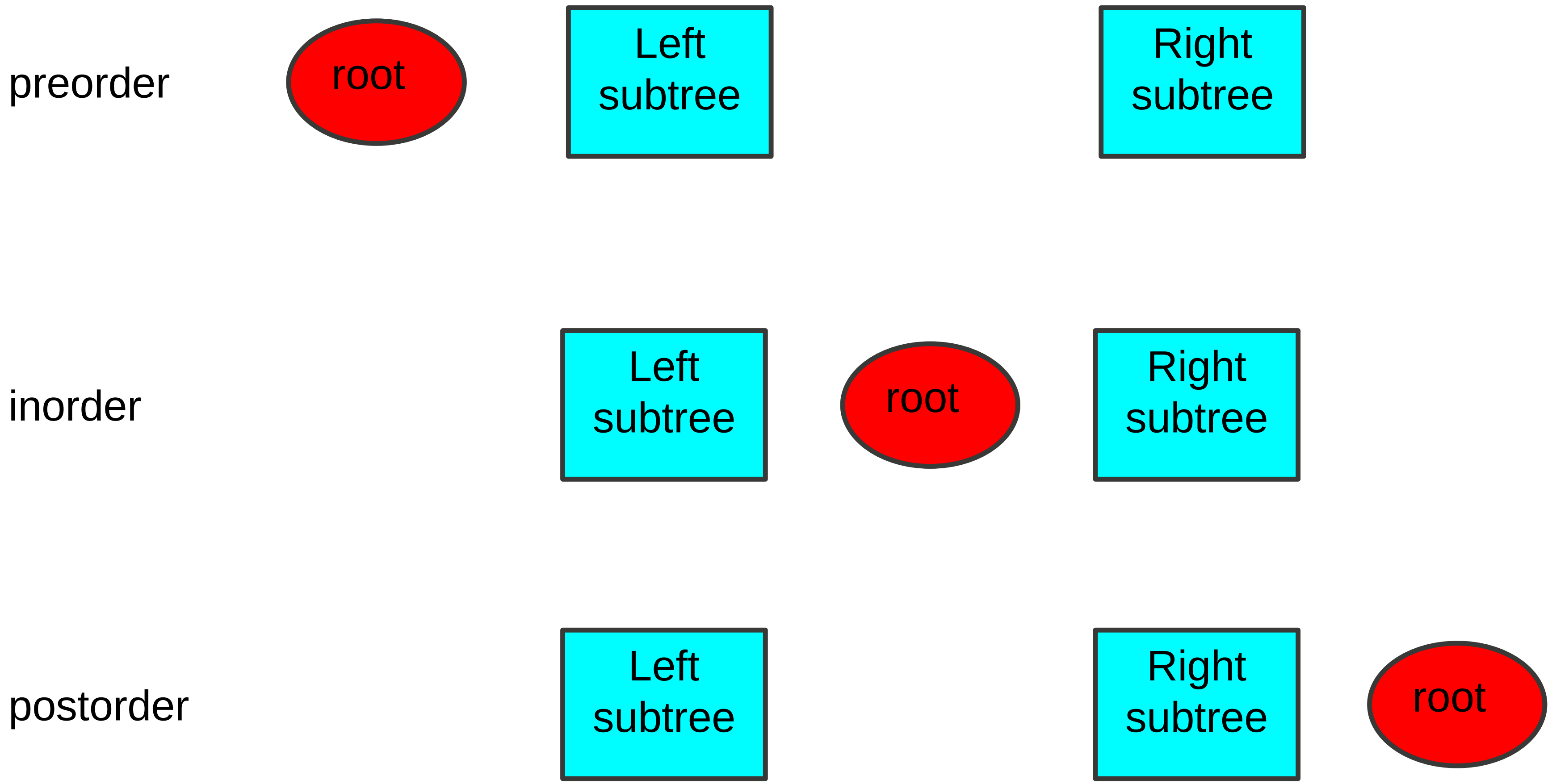


Again: note that `left` and `right` point to a node not to a sub-tree

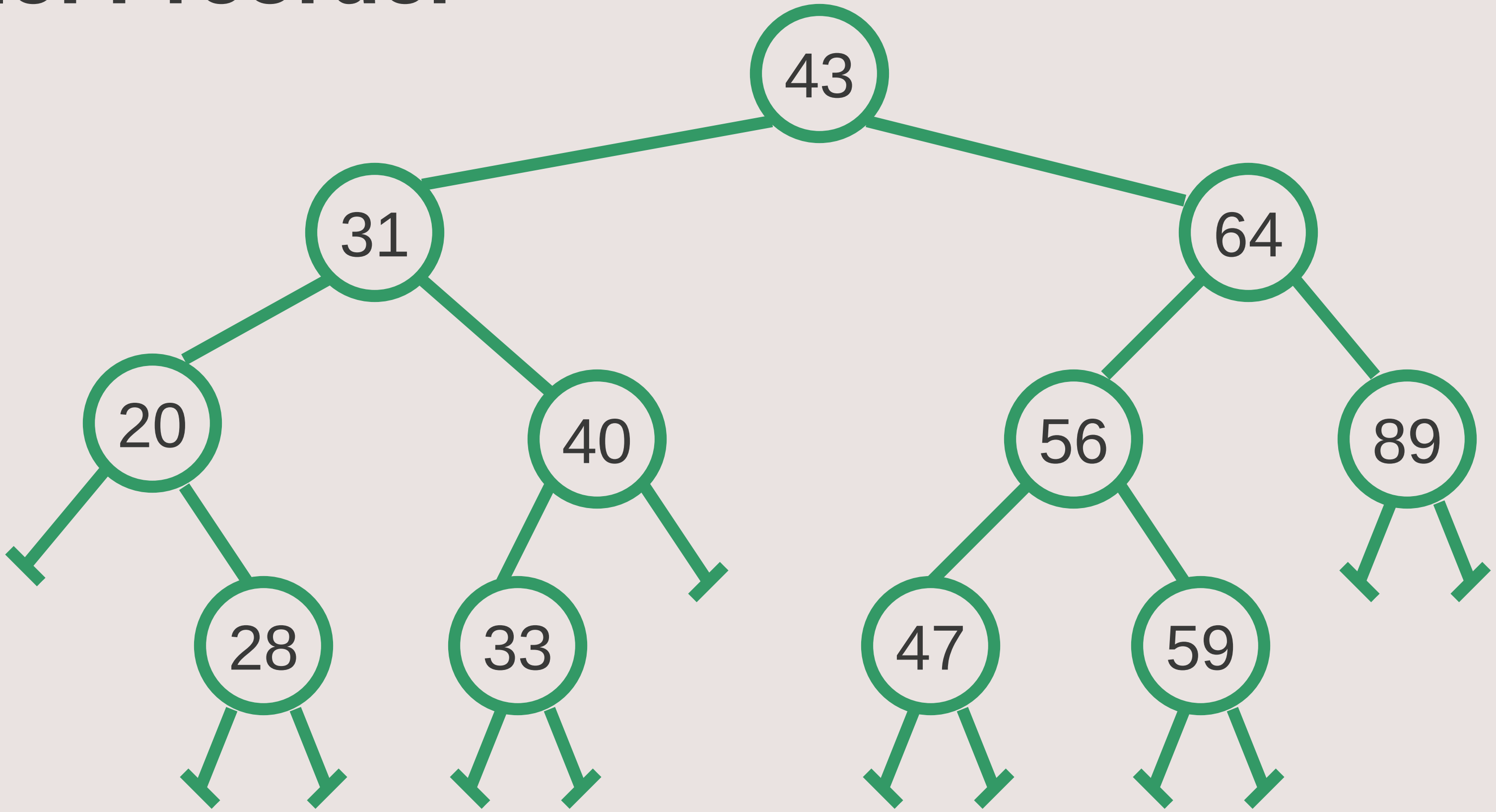
Binary Tree Traversal

Binary Tree Traversal

- **Systematic way of visiting/processing all the nodes**
- **Common methods:**
  - Preorder, Inorder, and Postorder
- **They all traverse the left subtree before the right subtree**
- **The name of the traversal method depends on when the root is processed**


preorder
root
Left subtree
Right subtree
inorder
Left subtree
root
Right subtree
postorder
Left subtree
Right subtree
root

Example: Preorder



43	31	20	28	40	33	64	56	47	59	89

Print Preorder Traversal

1) Print the root node
2) Print the left subtree in preorder
3) Print the right subtree in preorder

Note the recursive nature of the algorithm

Needed to pass the root node, rather than the tree

```
def print_preorder(self) -> None:
    self.print_preorder_aux(self.root)

def print_preorder_aux(self, current: BinaryTreeNode[T]) -> None:
    if current is not None: #if not a base case
        print(current)
        self.print_preorder_aux(current.left)
        self.print_preorder_aux(current.right)
```

The node has __str__

General Preorder Traversal

1) Process the root node
2) Process the left subtree in preorder
3) Process the right subtree in preorder

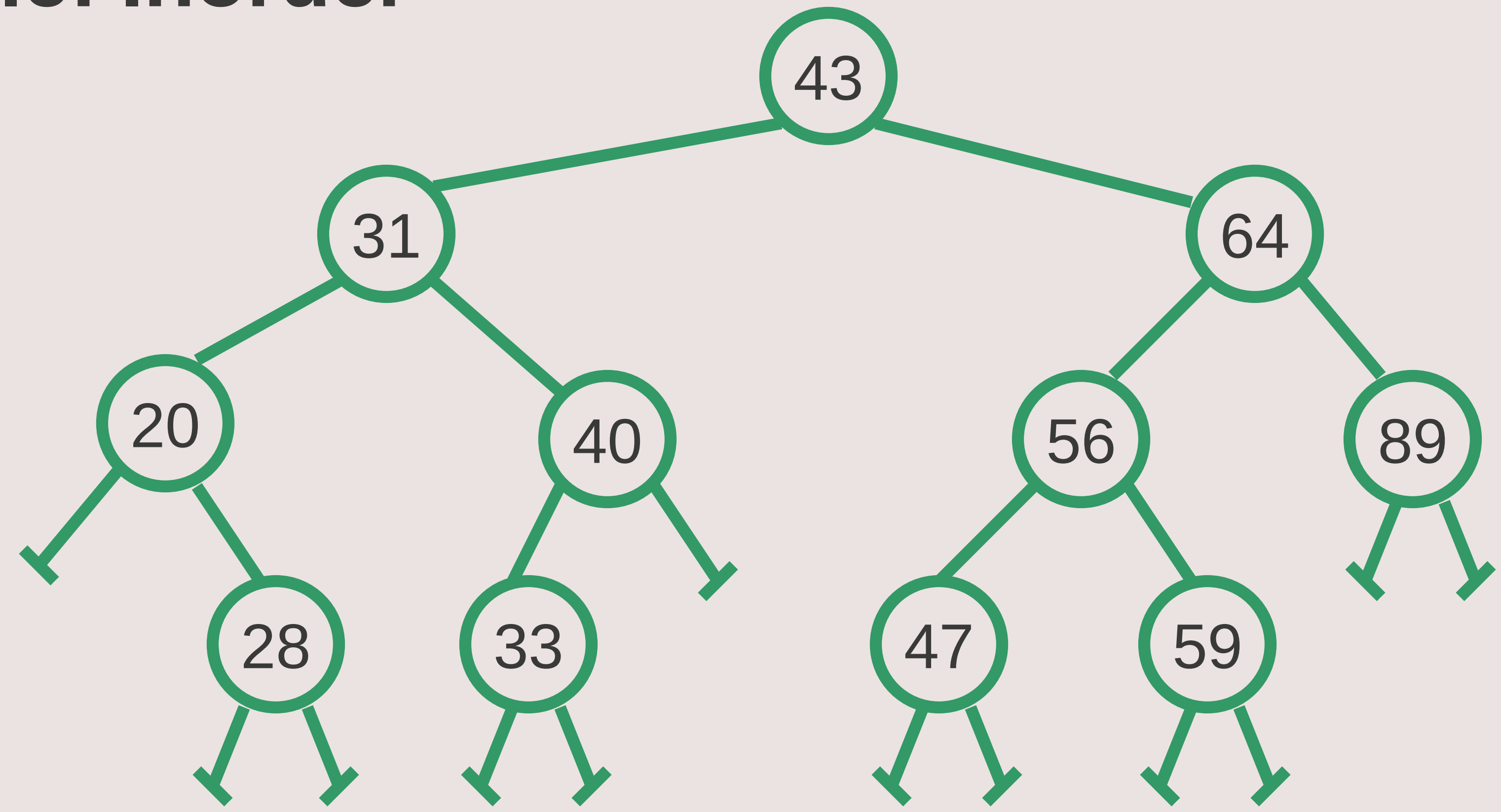
Use a general function f to process the nodes (could be print or anything else)

```
def preorder(self, f: Callable) -> None:
    self.preorder_aux(self.root, f)

def preorder_aux(self, current: BinaryTreeNode[T], f: Callable) -> None:
    if current is not None: #if not a base case
        f(current)
        self.preorder_aux(current.left, f)
        self.preorder_aux(current.right, f)
```

Complexity

- **Best case is equal to worse case**
  - We visit every node, regardless of the node's content
- **O(N)*Compf where**
  - N is the number of nodes in the tree
  - Compf is the complexity of method `f`
- **For example, if `f` is `print`, Compf will often be O(M) where M is the is the maximum size for an item**
  - Then, the complexity would be O(N*M)

Example: Inorder



20	28	31	33	40	43	47	56	59	64	89

Inorder Traversal

1) Process the left subtree in inorder
2) Process the root node
3) Process the right subtree in inorder

```
def inorder(self, f: Callable) -> None:
    self.inorder_aux(self.root, f)

def inorder_aux(self, current: BinaryTreeNode[T], f: Callable) -> None:
    if current is not None: #if not a base case
        self.inorder_aux(current.left, f)
        f(current.item)
        self.inorder_aux(current.right, f)
```

Postorder Traversal

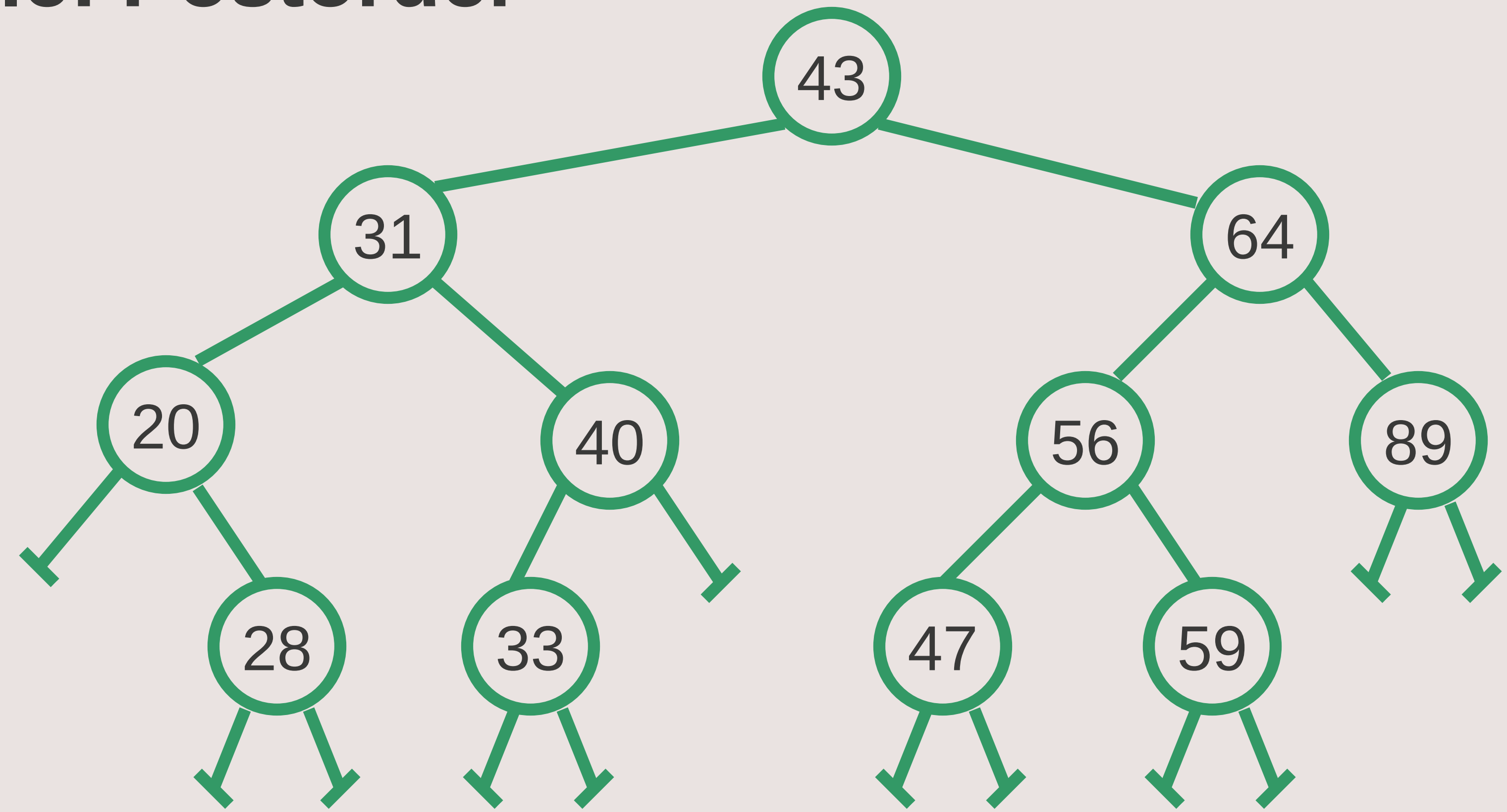
Complexity?

same as before

1) Process the left subtree in postorder
2) Process the right subtree in postorder
3) Process the root node

```
def postorder(self, f: Callable) -> None:
    self.postorder_aux(self.root, f)

def postorder_aux(self, current: BinaryTreeNode[T], f: Callable) -> None:
    if current is not None: #if not a base case
        self.postorder_aux(current.left, f)
        self.postorder_aux(current.right, f)
        f(current.item)
```

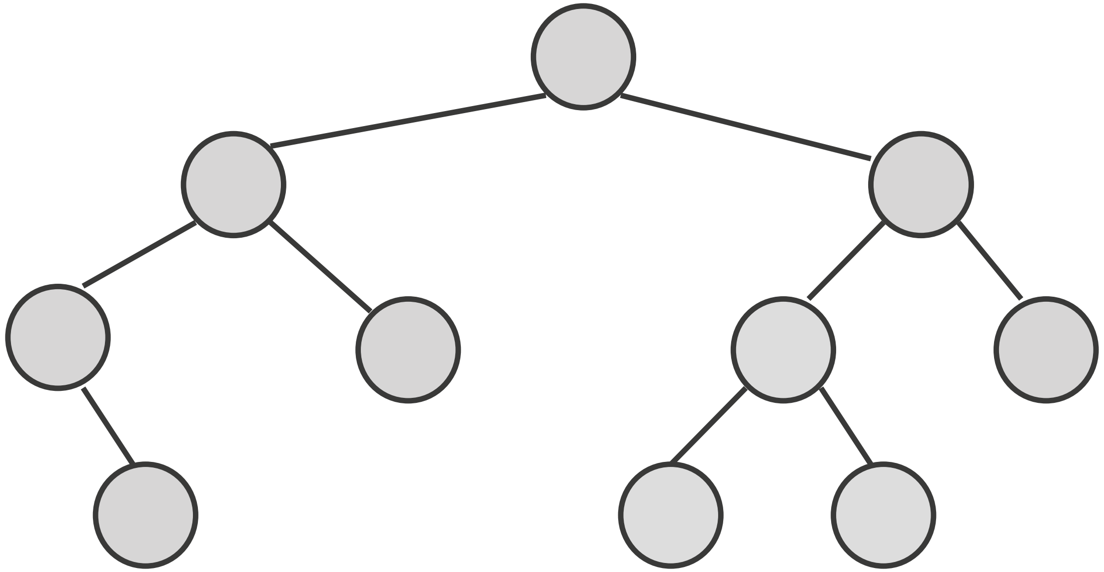
Example: Postorder



28	20	33	40	31	47	59	56	89	64	43

Example: computing the size of a tree

Add a recursive `__len__` method to `BinaryTree` that returns the number of nodes in the tree (without modifying the tree)



For example, the above tree has 10 nodes.
How to compute this recursively? It must use the size of smaller trees…
Convergence? (when using lists we passed a copy of head so now…)
Base case? (empty? Which returns 0)
Combination of solutions? (+)

Example: computing the size of a tree

```
def __len__(self) -> int:
    return self.len_aux(self.root)


def len_aux(self, current: BinaryTreeNode[T]) -> int:
    if current is None:
        return 0
    else:
        return 1+self.len_aux(current.left)
                +self.len_aux(current.right)
```

Summary

- **Tree concepts:**
  - Parent, child, root, leaf, and inner nodes
  - Subtree
  - Levels and maximum depth
  - Paths
  - Binary trees
  - Balanced/unbalanced binary trees
  - Perfect binary trees

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