COMP 250

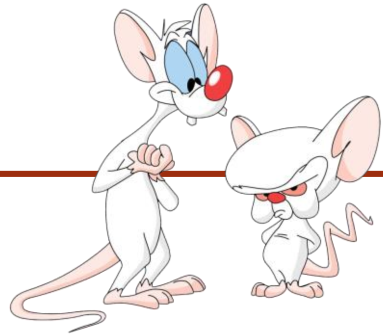
INTRODUCTION TO COMPUTER SCIENCE

Week 11-1 : Rooted Trees

Giulia Alberini, Fall 2020

Slides adapted from Michael Langer's

WHAT ARE WE GOING TO DO IN THIS VIDEO?



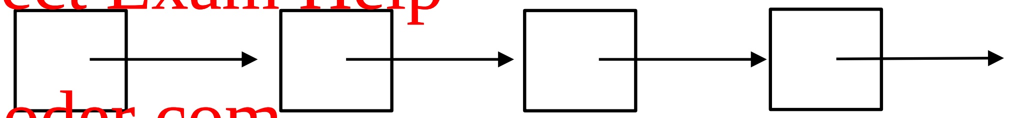
- Rooted Trees
  - Terminology
  - Implementation

DATA STRUCTURES

- Linear

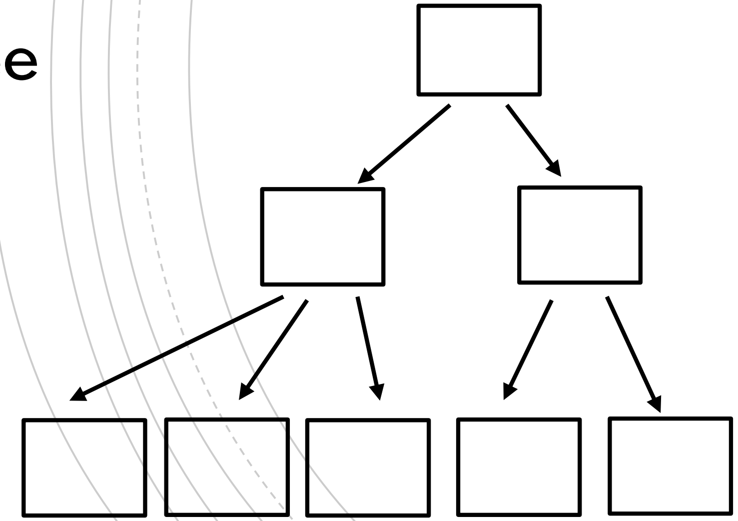
array

Linked list

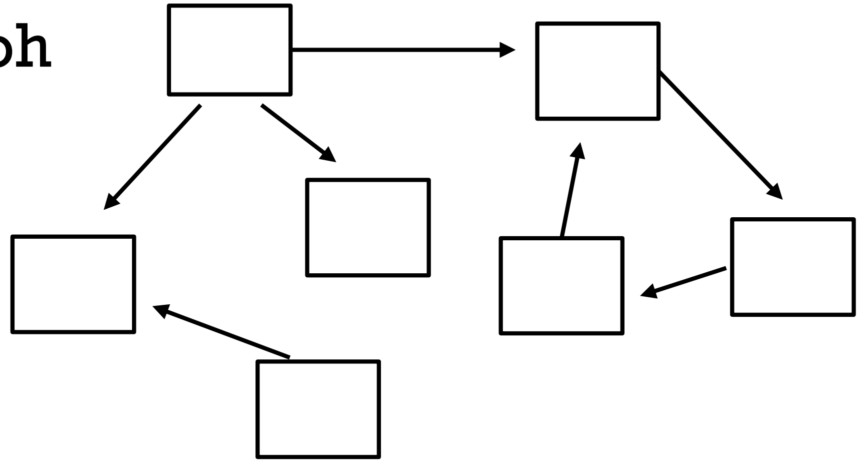


- Non-linear

tree

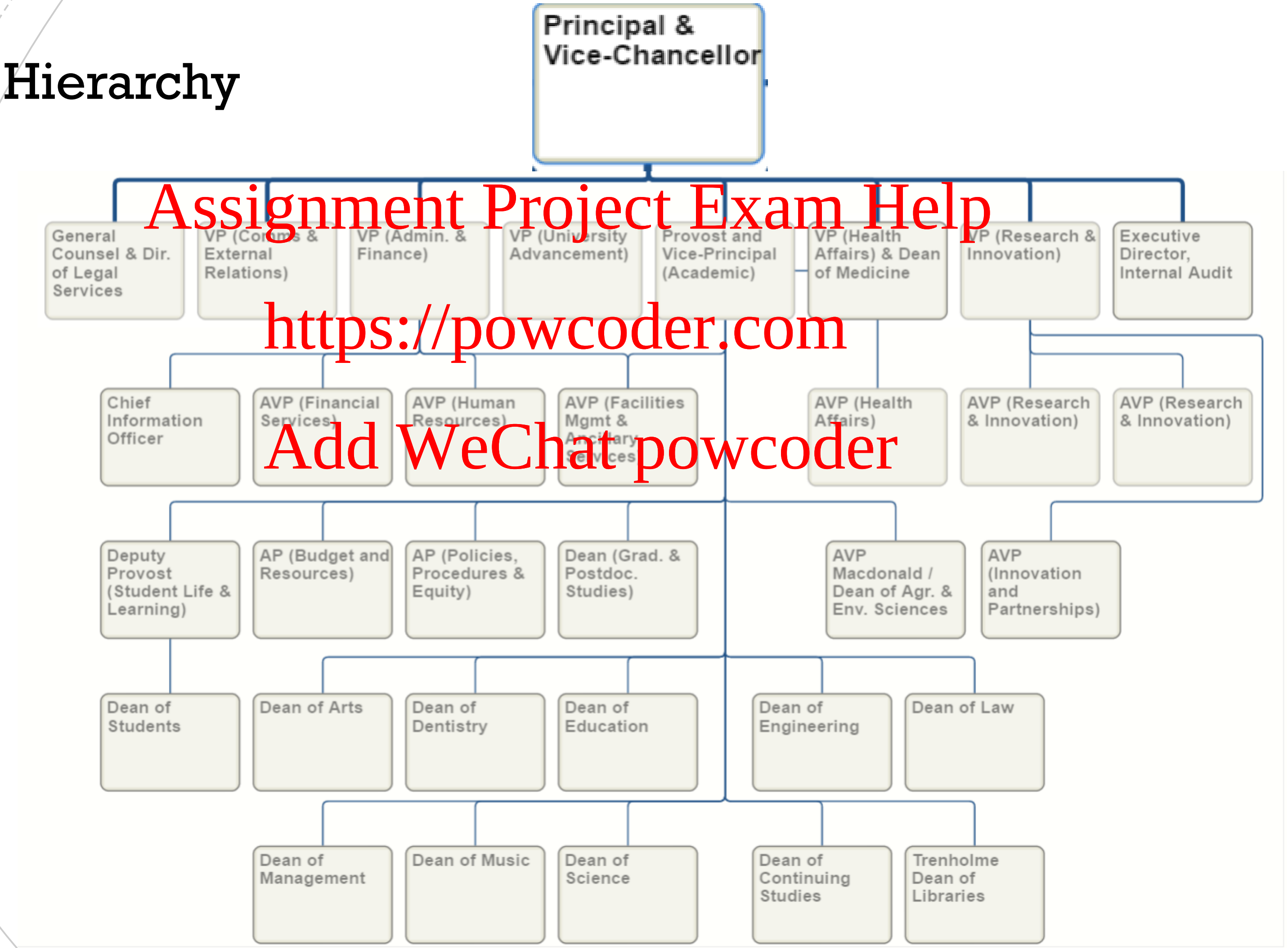


graph

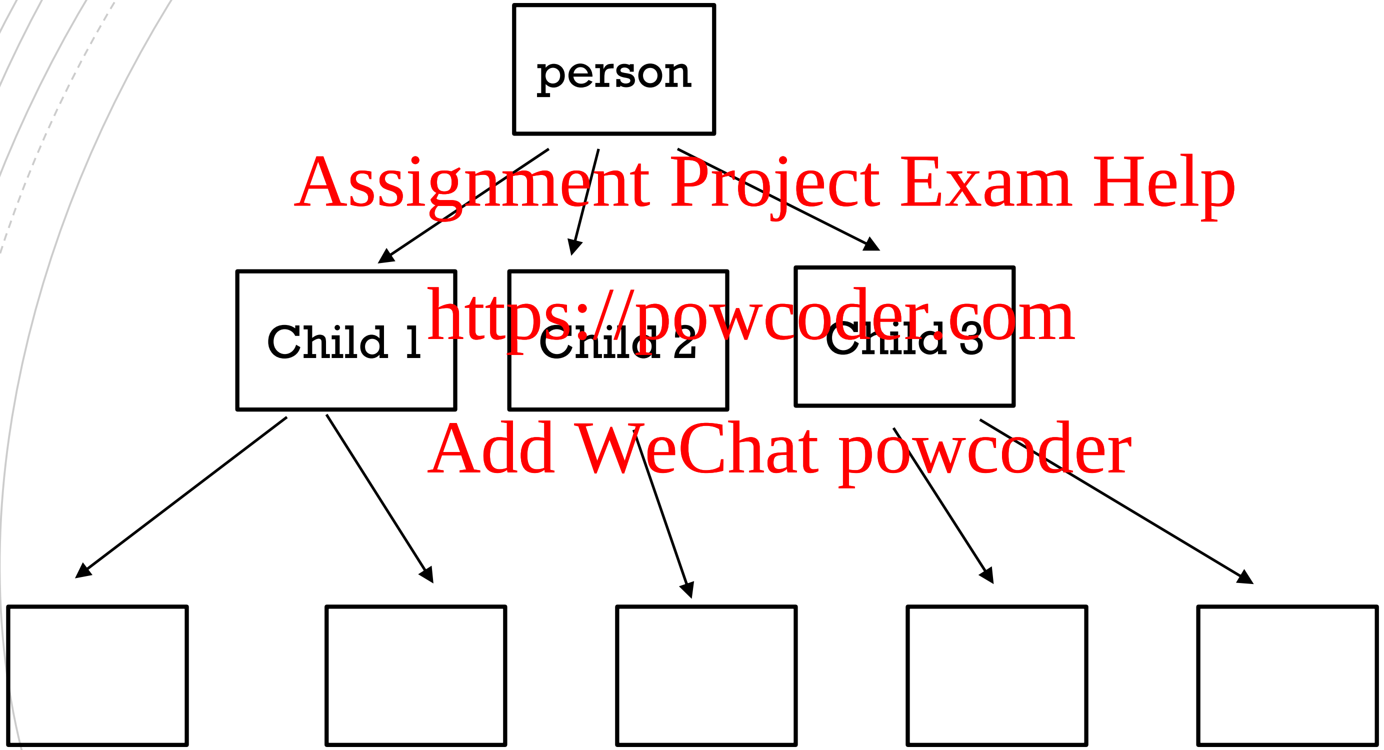


TREE - EXAMPLE

- Organizational Hierarchy (McGill)

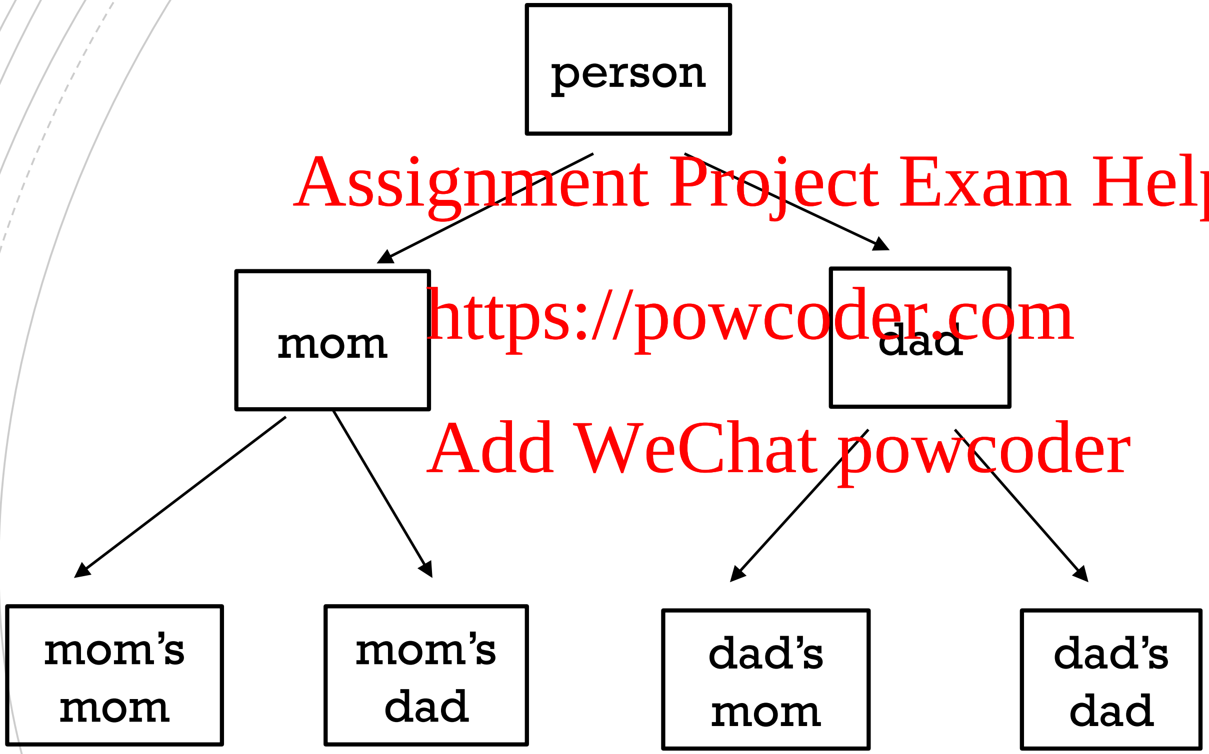


EXAMPLE 2: FAMILY TREE (DESCENDANTS)



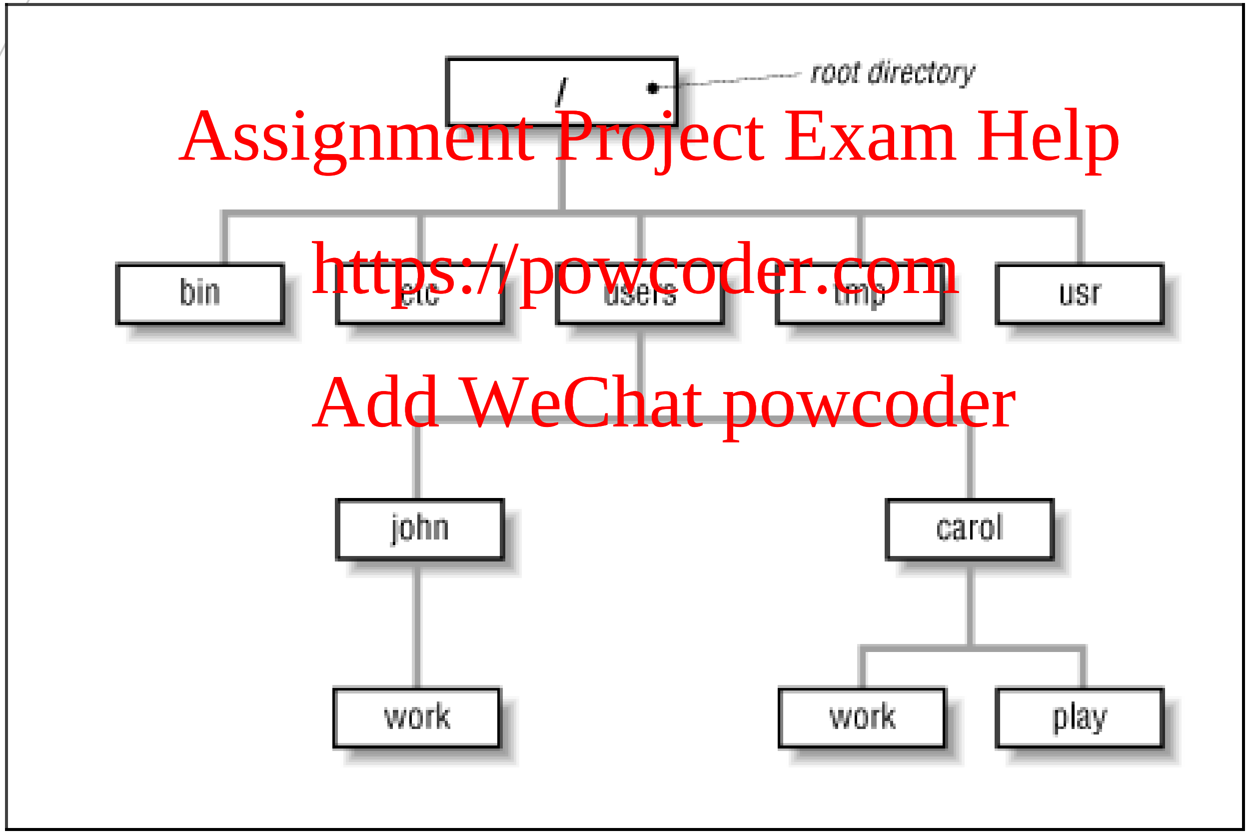
Here we ignore spouses (partner).

EXAMPLE 3: FAMILY TREE (ANCESTORS)

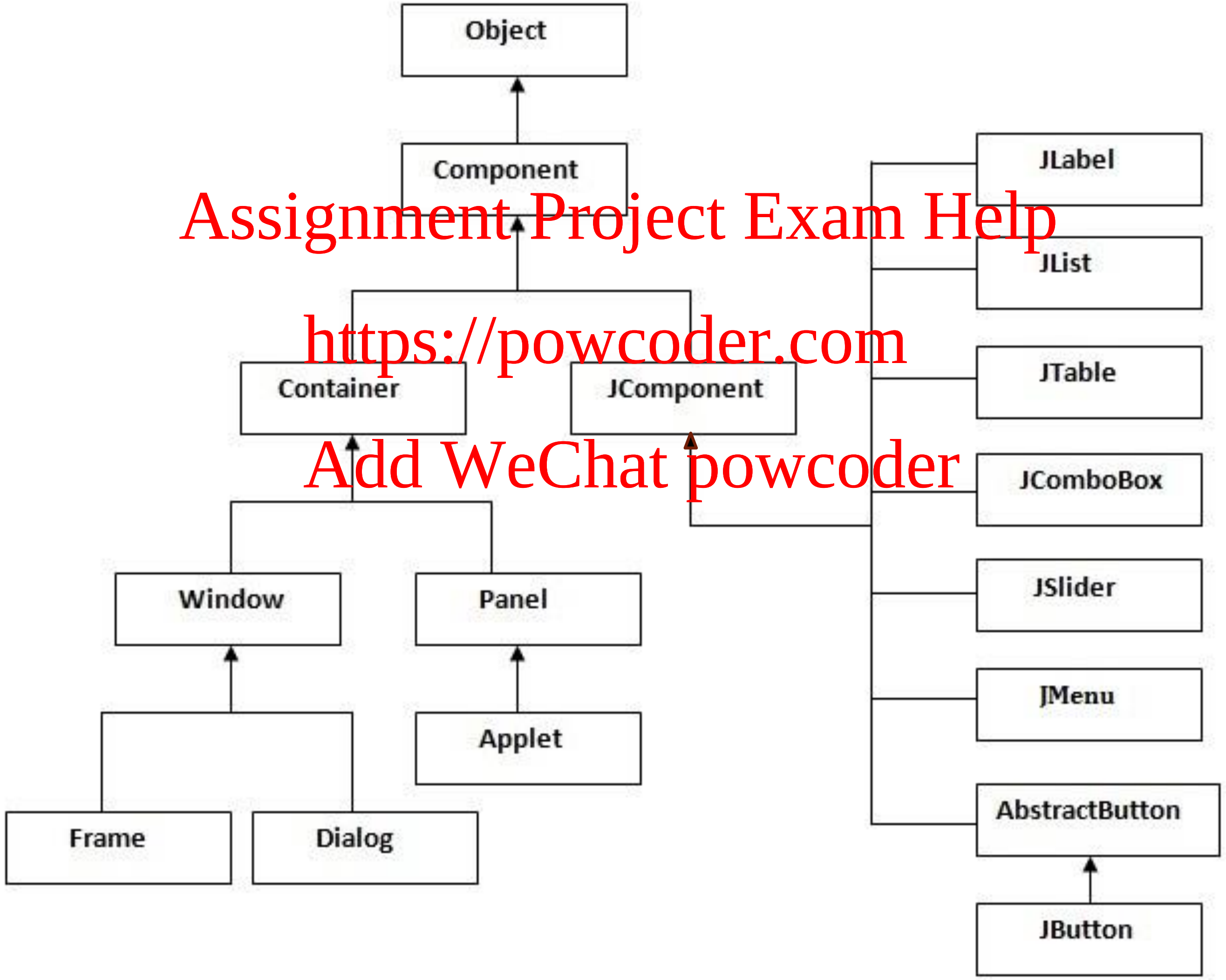


This is an example of a binary tree.

EXAMPLE 4: UNIX FILE SYSTEM

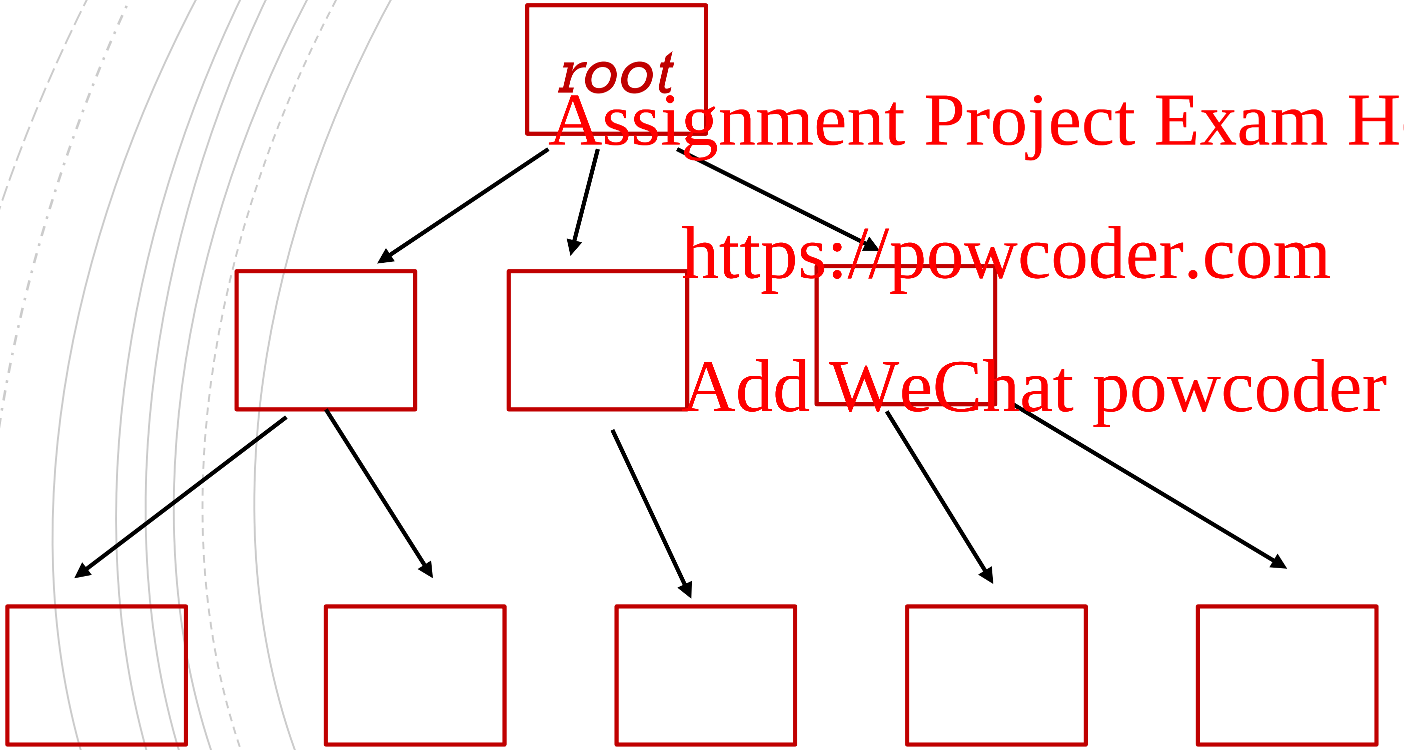


EXAMPLE 5: JAVA CLASSES E.G. GUI



DEFINITIONS

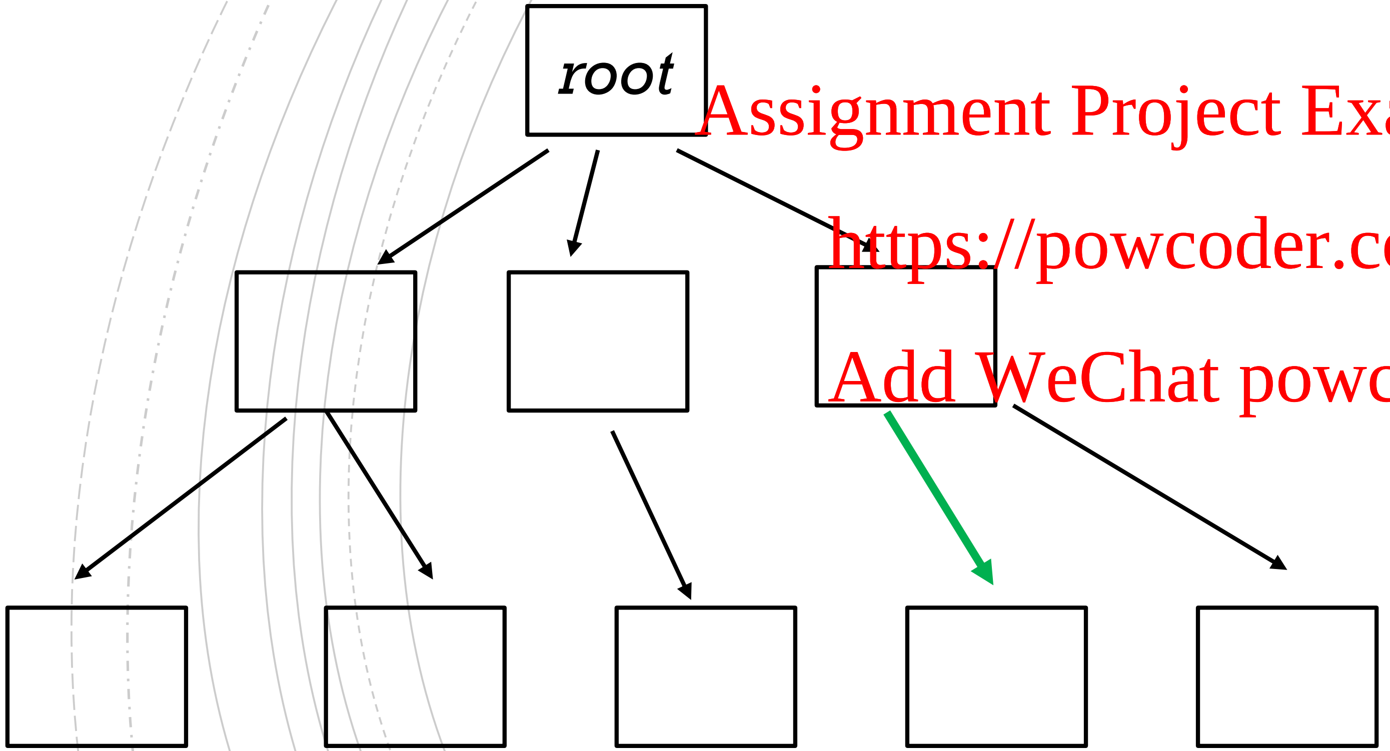
(ROOTED) TREE TERMINOLOGY



A tree is a collection of *nodes* (*vertexes*)

The *root* is the top node in a tree

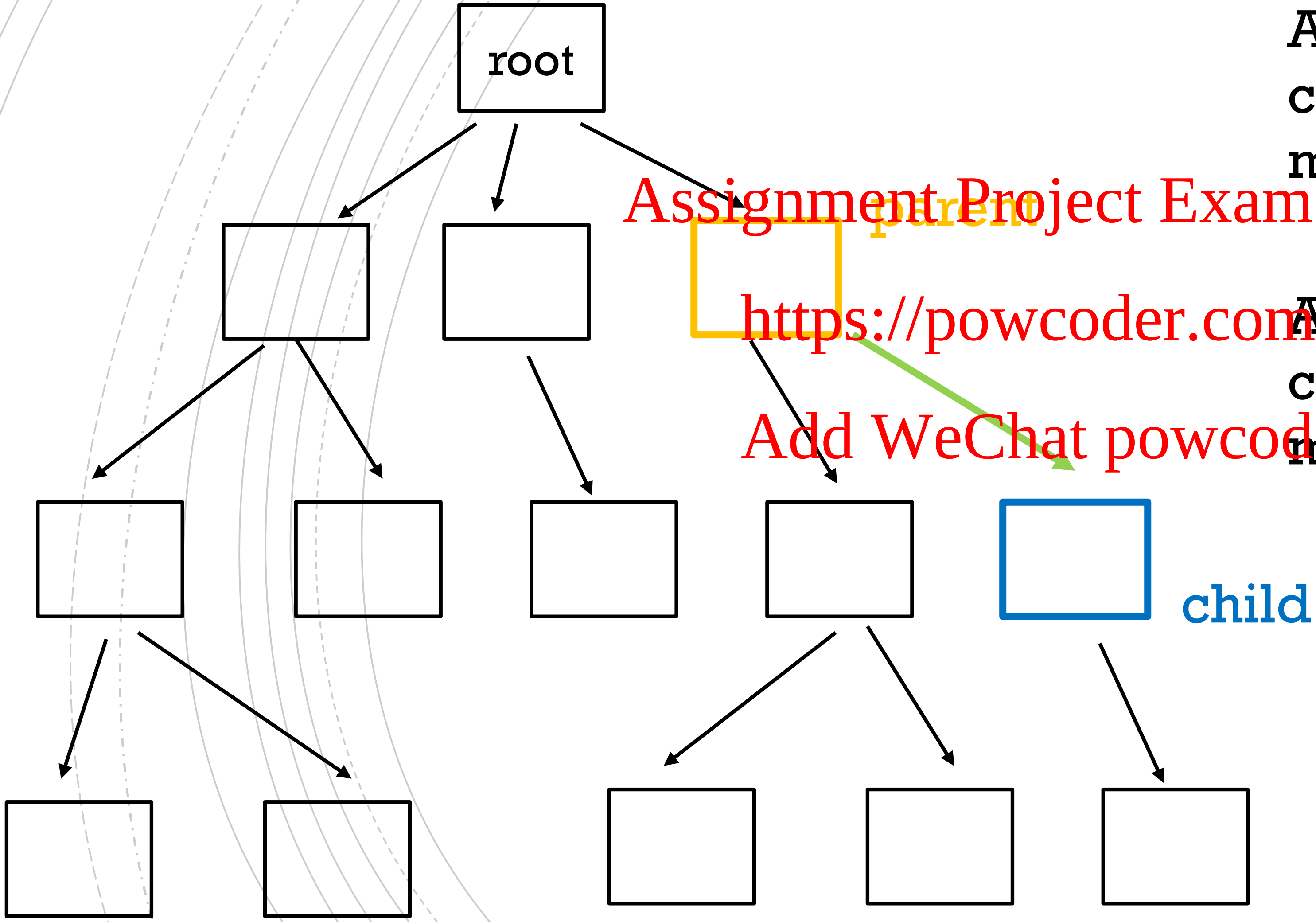
(ROOTED) TREE TERMINOLOGY



A *directed edge* is ordered pair of nodes (v_i, v_j) $(\text{from},\ \text{to})$.

Trees can be *undirected* or *directed*. If directed, the edges are either all pointing away from the root or all pointing towards the root.

(ROOTED) TREE TERMINOLOGY



A *child* is a node directly connected to another node when moving away from the root.

A *parent* is a node directly connected to another node when moving towards the root.

Every node except the root is a *child*, and has exactly one *parent*.

EDGE DIRECTION

For some trees,

- edges are directed from parent to child
- edges are directed from child to parent
- edges are directed both from parent to child and child to parent.
- edge direction is ignored e.g. common with non-rooted trees (see next slide)

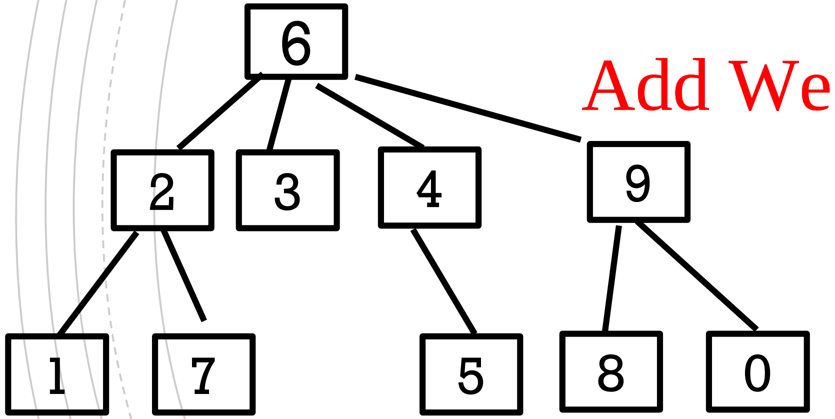
Most of definitions today will assume edges are from parent to child.

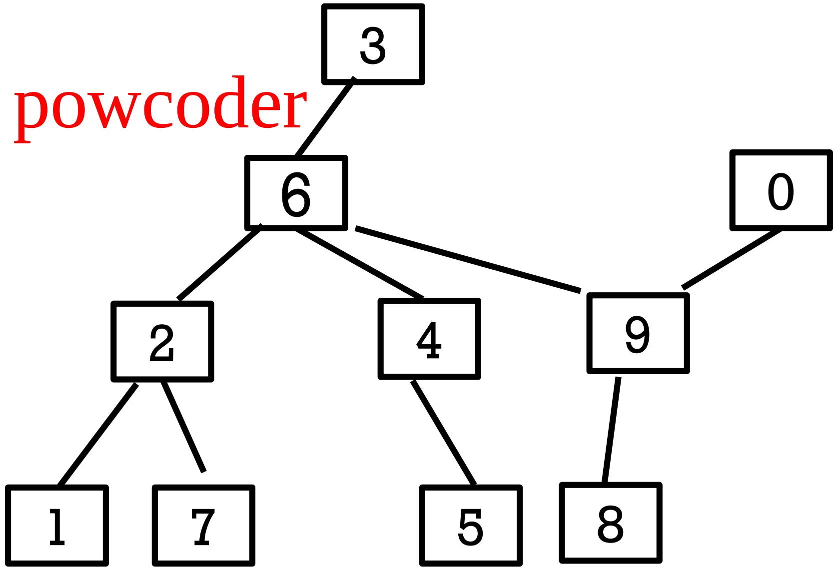
ASIDE: NON-ROOTED TREES

You will see non-rooted trees most commonly when edges are undirected, and there is no natural way to define the 'root'.

You will see examples in COMP 251.

e.g. the tree on the left is only rooted because I drew it that way. It is actually the same (non-rooted) tree as the one on the right.





NUMBER OF EDGES

- Q: If a (rooted) tree has n nodes, then how many edges does it have?

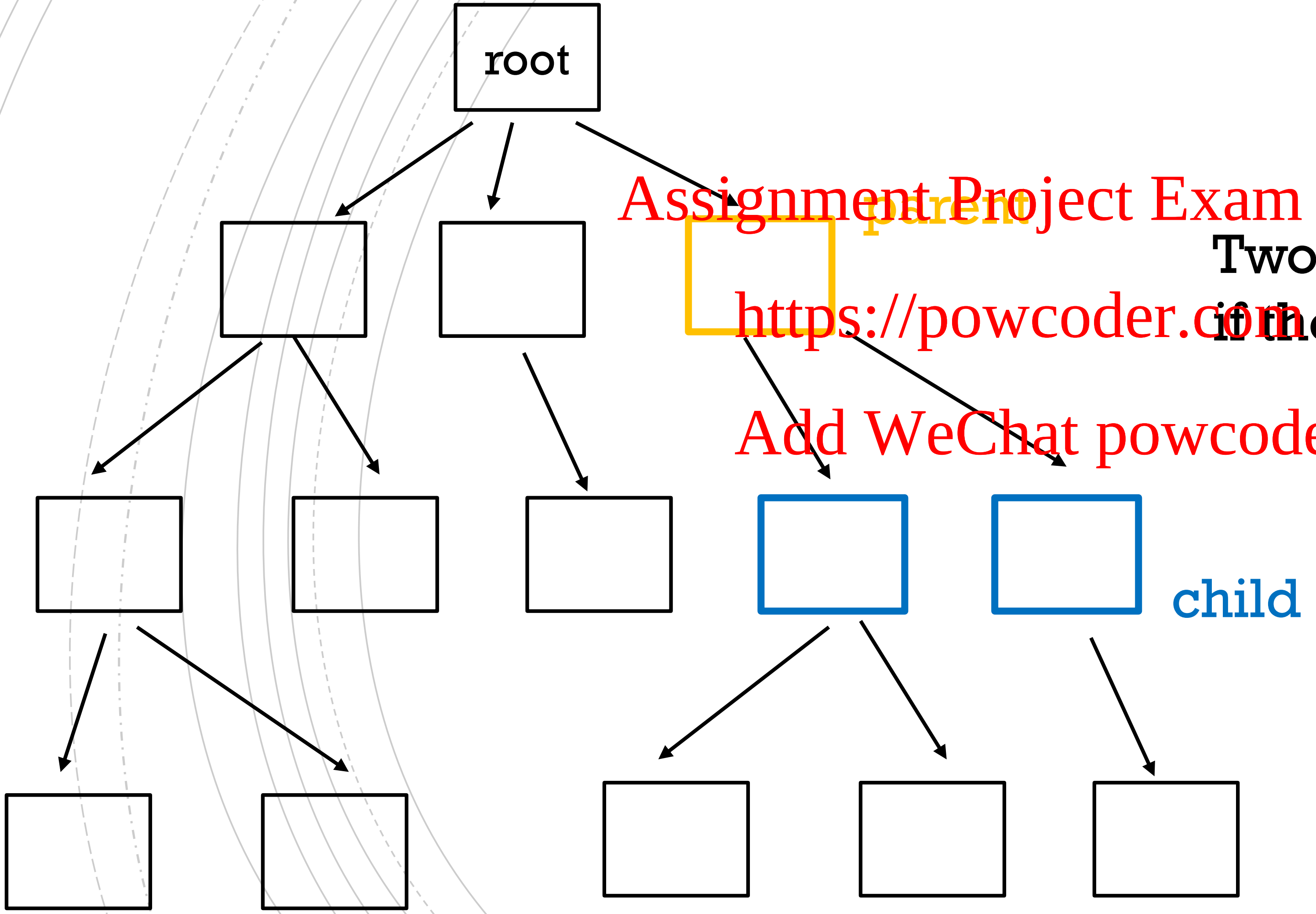
NUMBER OF EDGES

- Q: If a (rooted) tree has n nodes, then how many edges does it have?

- A: $n - 1$

Since every edge is of the form (parent, child), and each node except the root is a child and each child has exactly one parent.

(ROOTED) TREE TERMINOLOGY

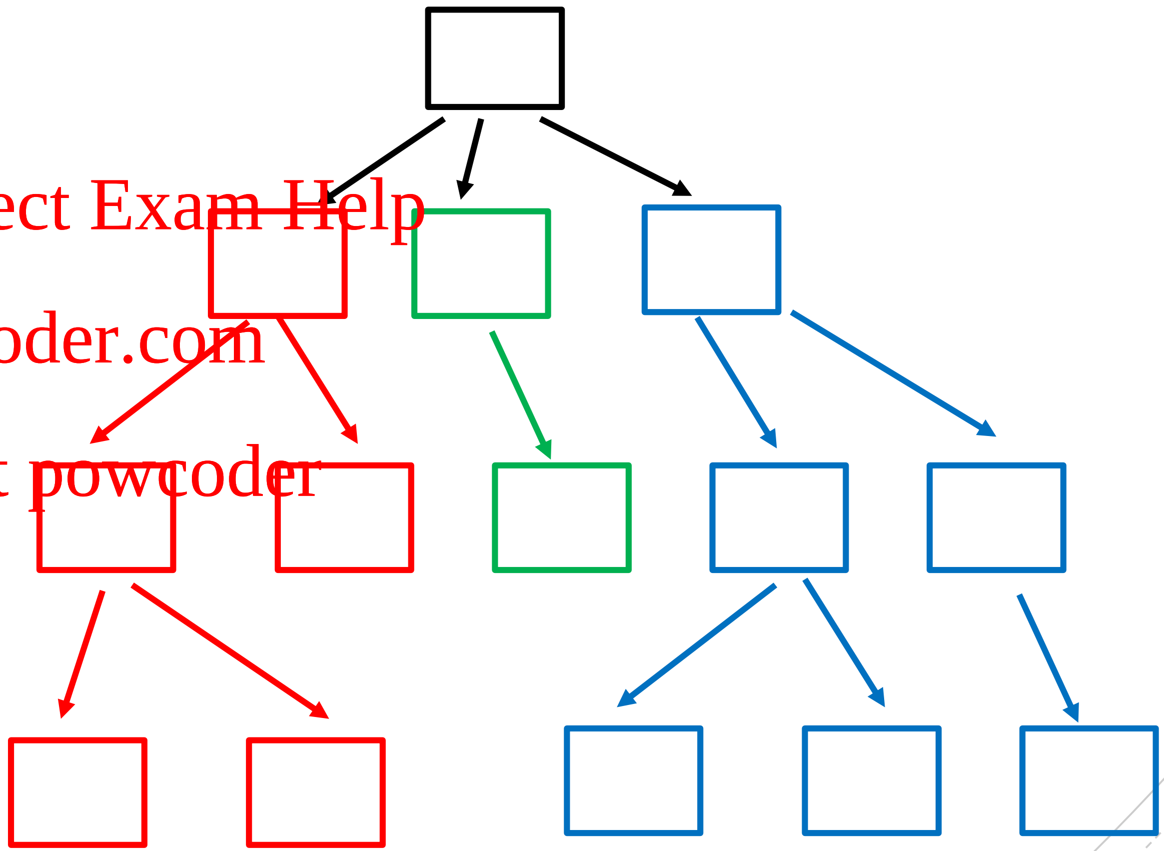


Two nodes are said to be siblings if they have the same parent.

RECURSIVE DEFINITION OF ROOTED TREE

A tree T is a finite (& possibly empty) set of n nodes such that:

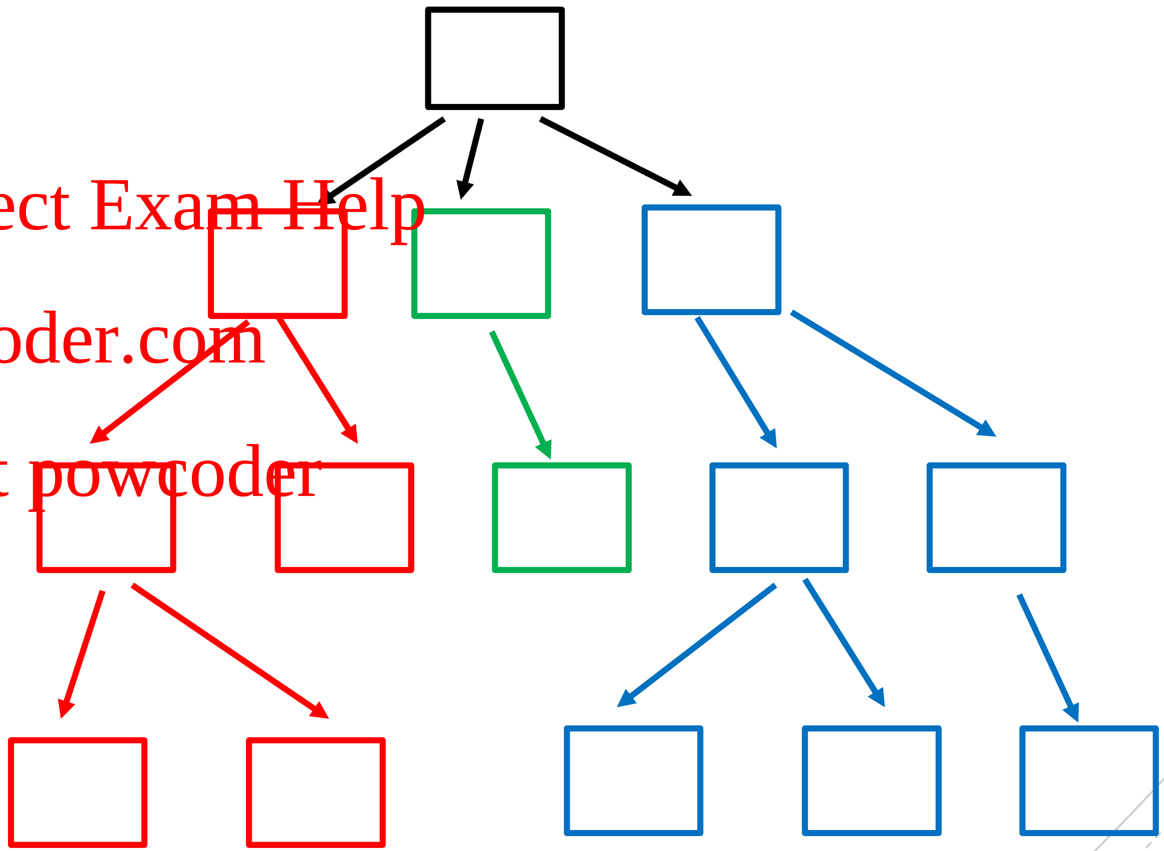
?



RECURSIVE DEFINITION OF ROOTED TREE

A tree T is a finite (& possibly empty) set of n nodes such that:

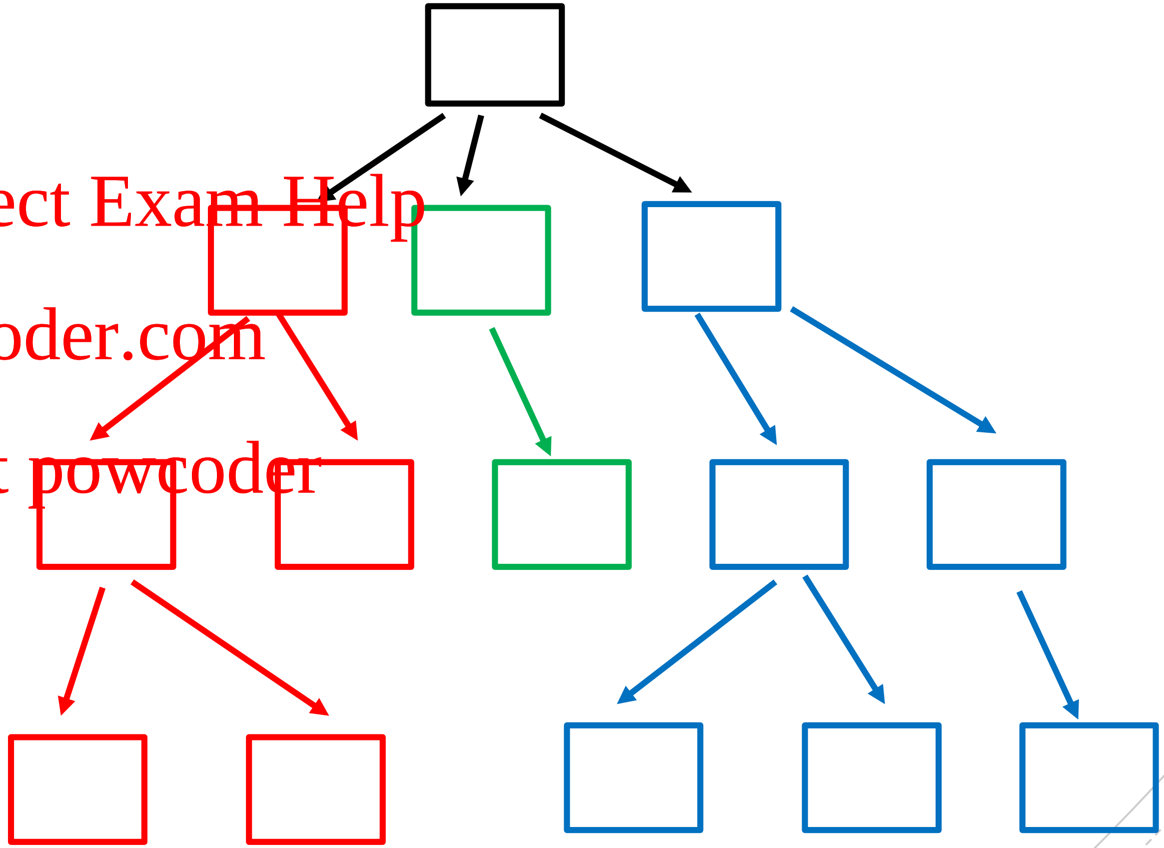
- if $n > 0$ then one of the nodes is the root r
- ?



RECURSIVE DEFINITION OF ROOTED TREE

A tree T is a finite (& possibly empty) set of n nodes such that:

- if $n > 0$ then one of the nodes is the root r
- if $n > 1$ then the $n - 1$ non-root nodes are partitioned into (non-empty) subsets $T_1, T_2, \ldots, T_k$, each of which is a tree (called a "subtree"), and the roots of the subtrees are the children of root r.



This definition assumes directed edges ("…children…") but we could change the wording so that it does not assume directed edges.

ANOTHER DEFINITION

A recursive definition for tree can also be given using lists as follows:


```
tree    =     root  |   ( root listOfSubTrees )
```



```
listOfSubTrees  =  tree | tree listOfSubTrees
```


Note that `listOfSubTrees` cannot be empty.

TRY IT!

A recursive definition for tree can also be given using lists as follows:


```
tree = root | ( root listOfSubTrees )
```



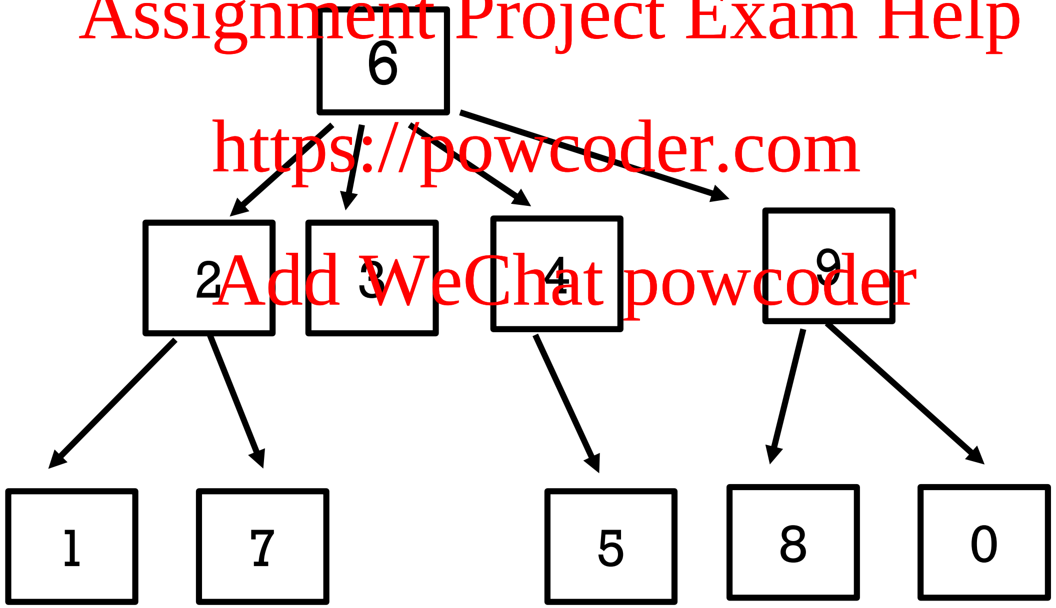
```
listOfSubTrees = tree | tree listOfSubTrees
```


- Draw the tree that corresponds to the following list, where the root elements are single digits.

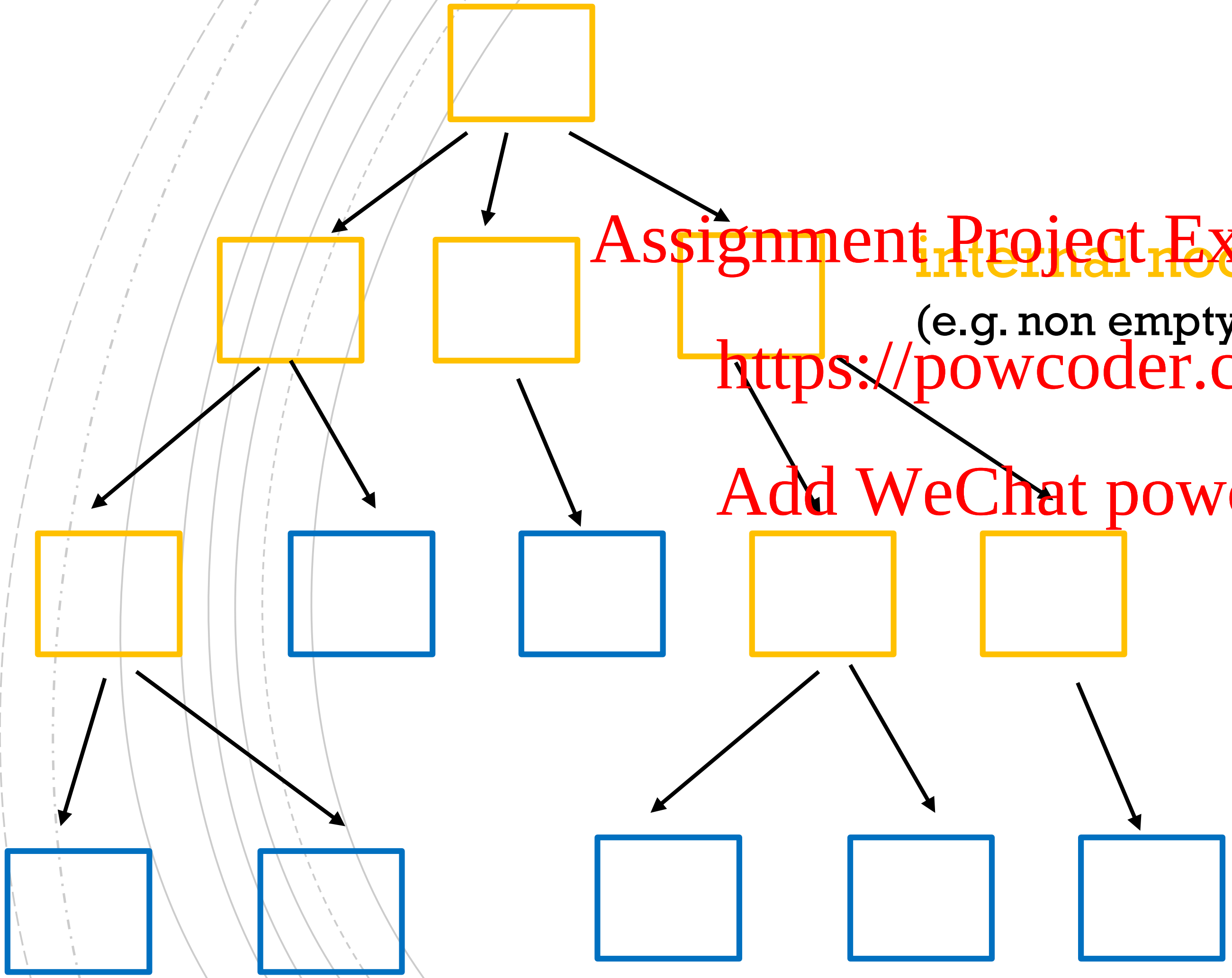
```
(6 (217) 3 (45) (980))
```

SOLUTION

`( 6 ( 2 1 7 ) 3 ( 4 5 ) ( 9 8 0 ) )` represents the following tree:



(ROOTED) TREE TERMINOLOGY



internal nodes

(e.g. non empty file directories)

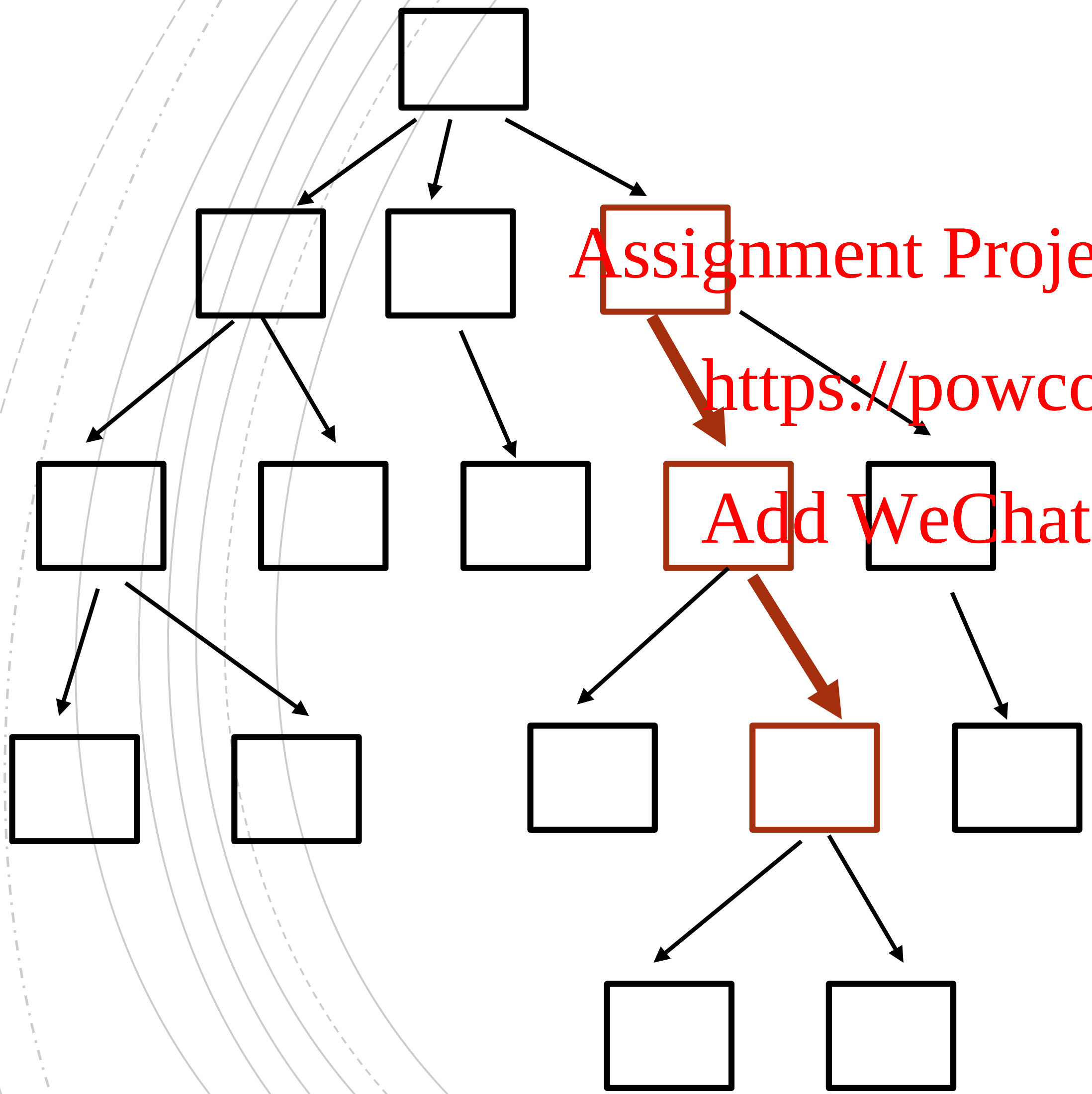
leaves (external nodes)

(e.g. files or empty directories)

An *internal node* is a node with at least one child.

An *leaf (or external node)* is a node with no children.

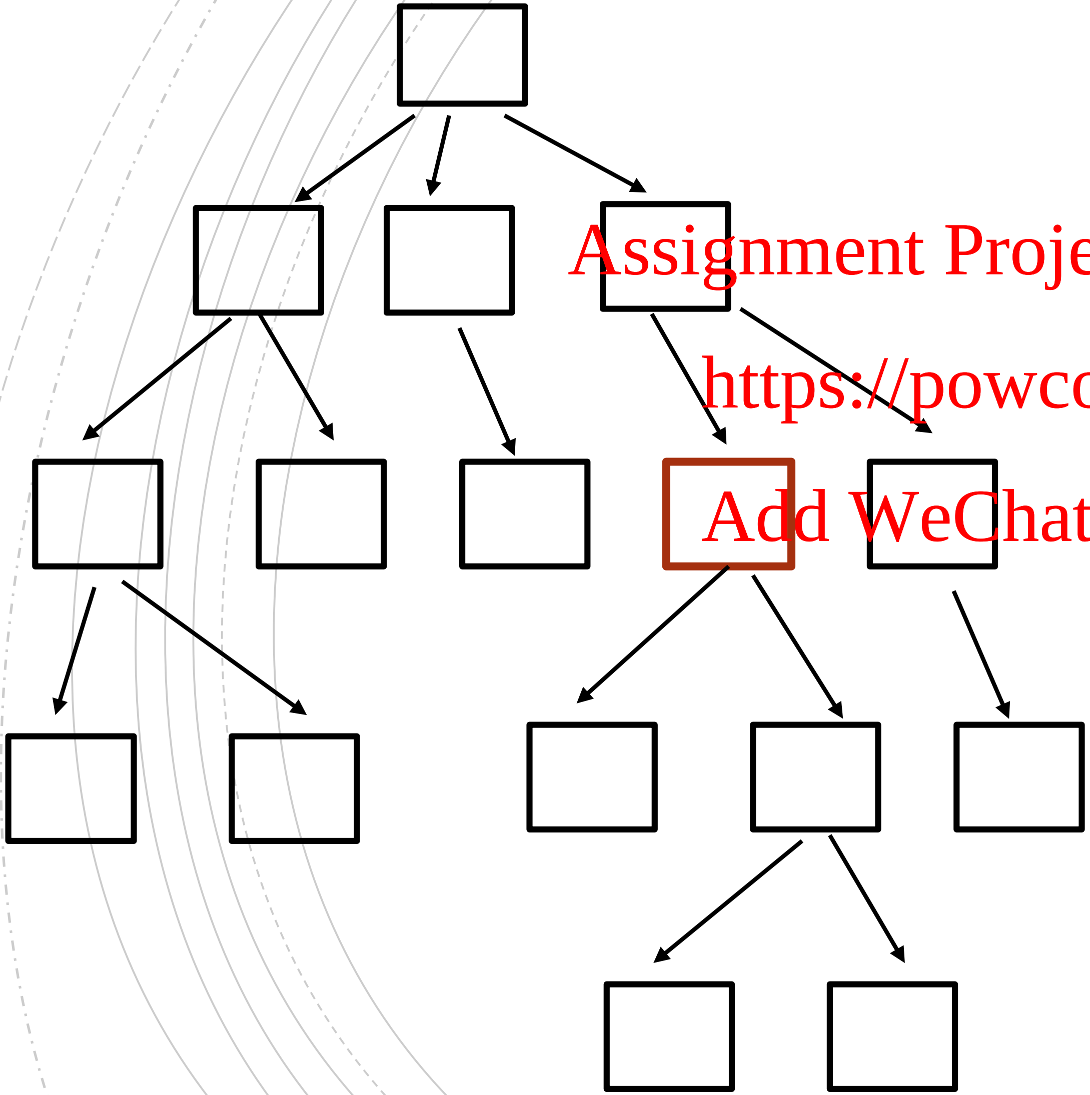
TREE TERMINOLOGY



A *path* in a tree is a sequence of nodes $(v_1, v_2, \ldots, v_k)$ such that (v_i, v_{i+1}) is an edge.

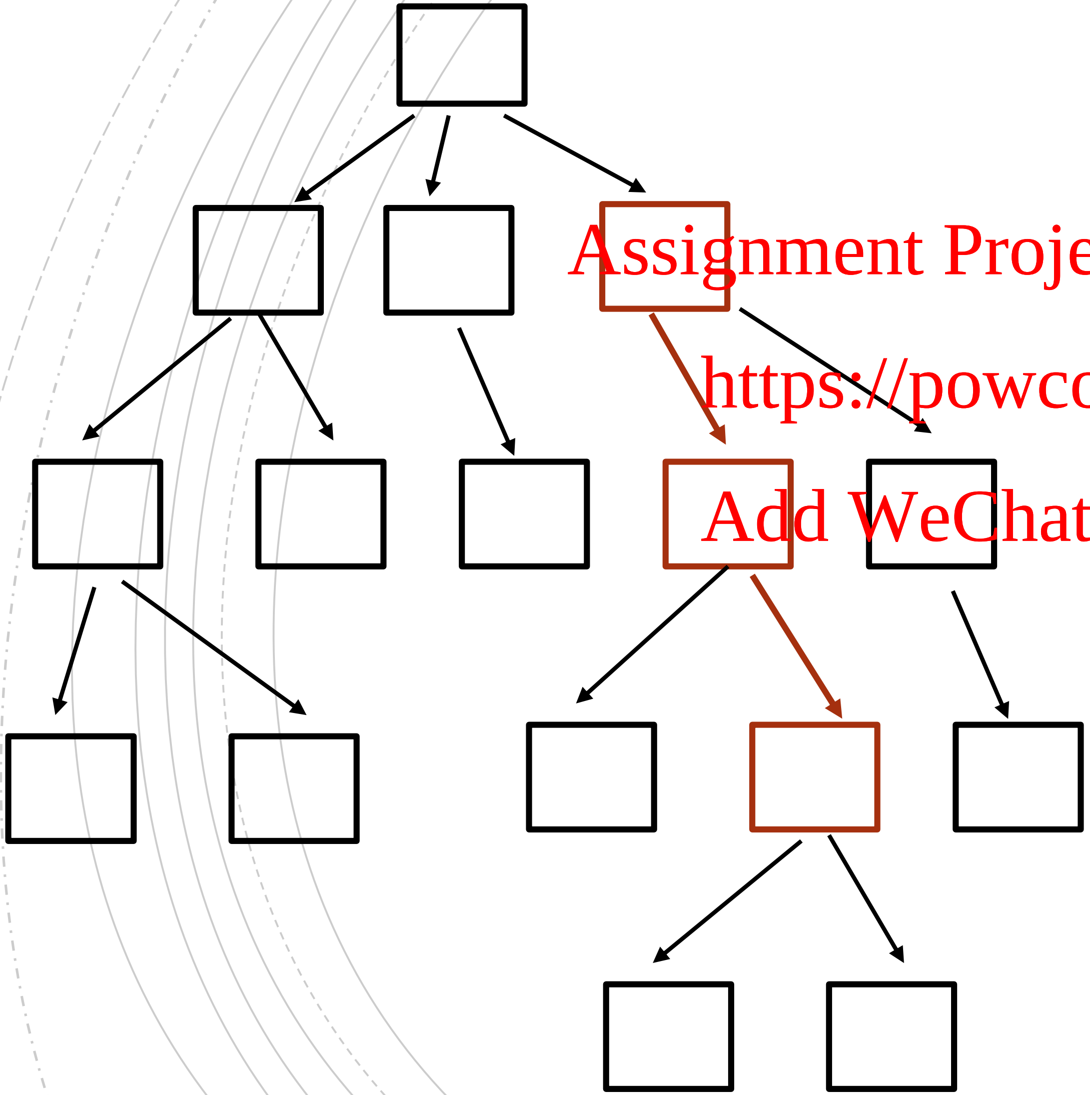
The *length* of a path is *the number of edges in the path* (number of nodes in the path – 1)

TREE TERMINOLOGY



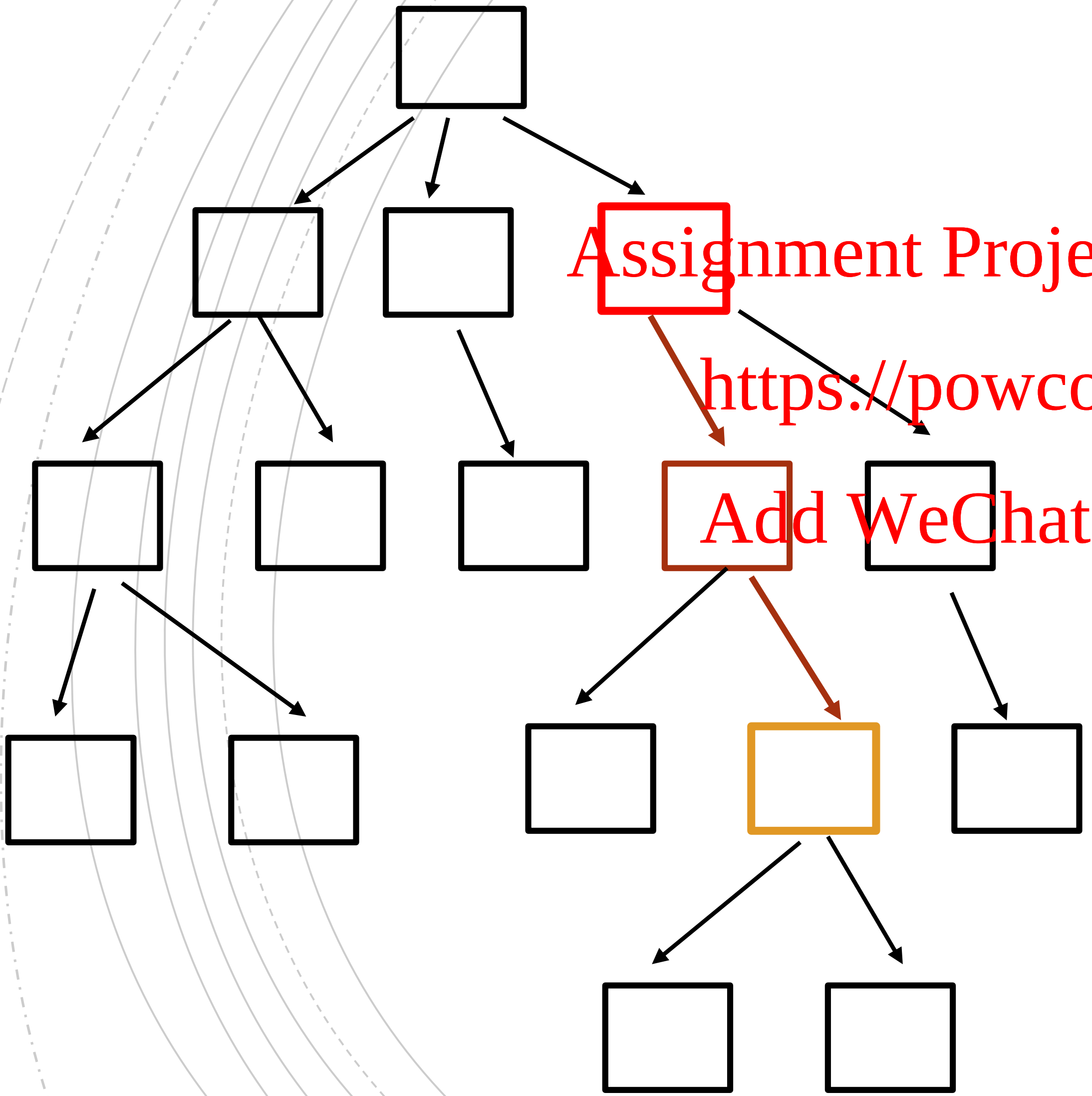
A **path** with just one node (v_1) has **length** $= 0$, since it has no edges.

QUICK QUESTION



What is the path length?

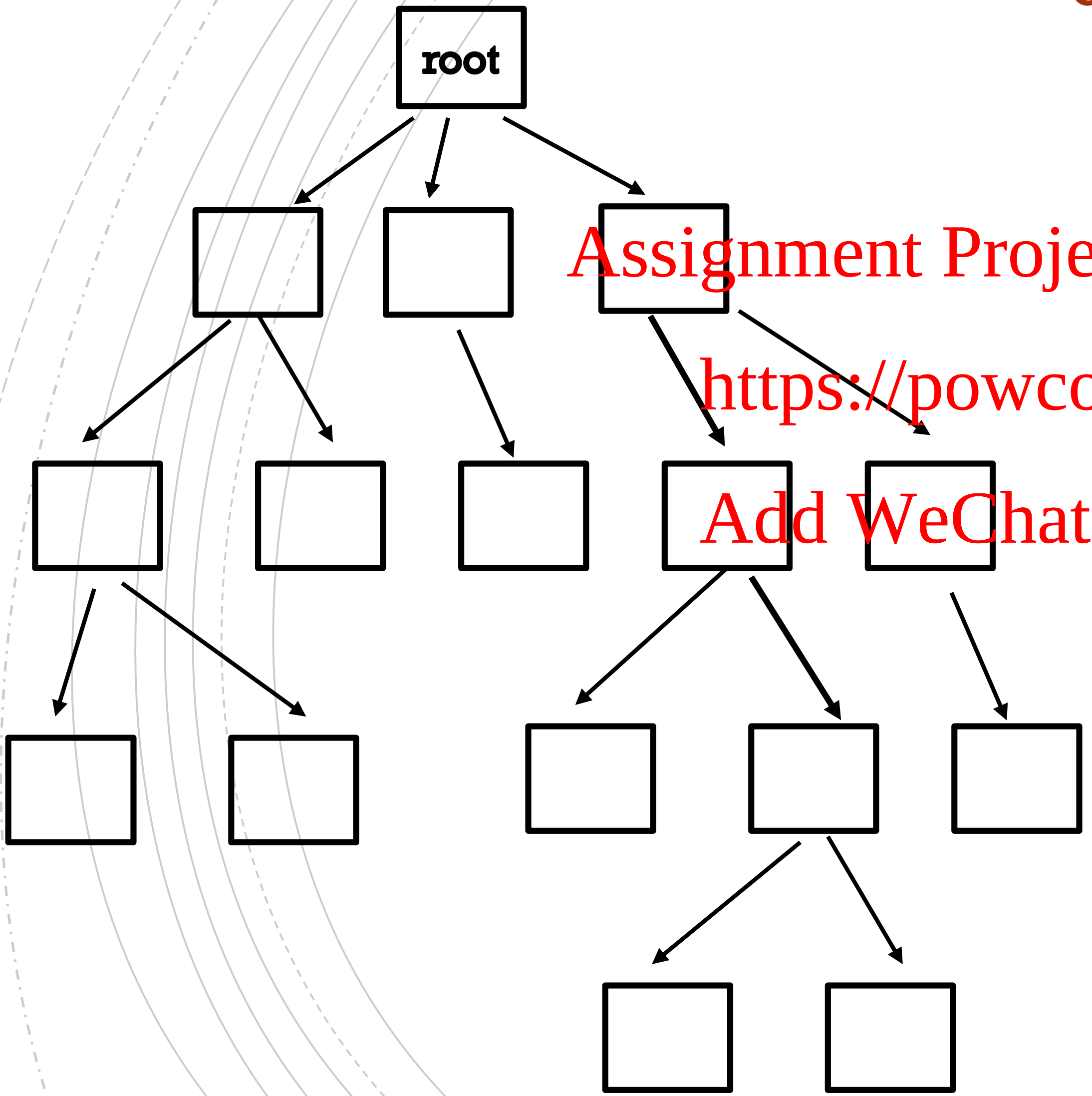
TREE TERMINOLOGY



Node v is an *ancestor* of node w if there is a path from v to w. In such case, we also say that w is a *descendent* of node v.

(ROOTED) TREE TERMINOLOGY

depth (level)



0

1

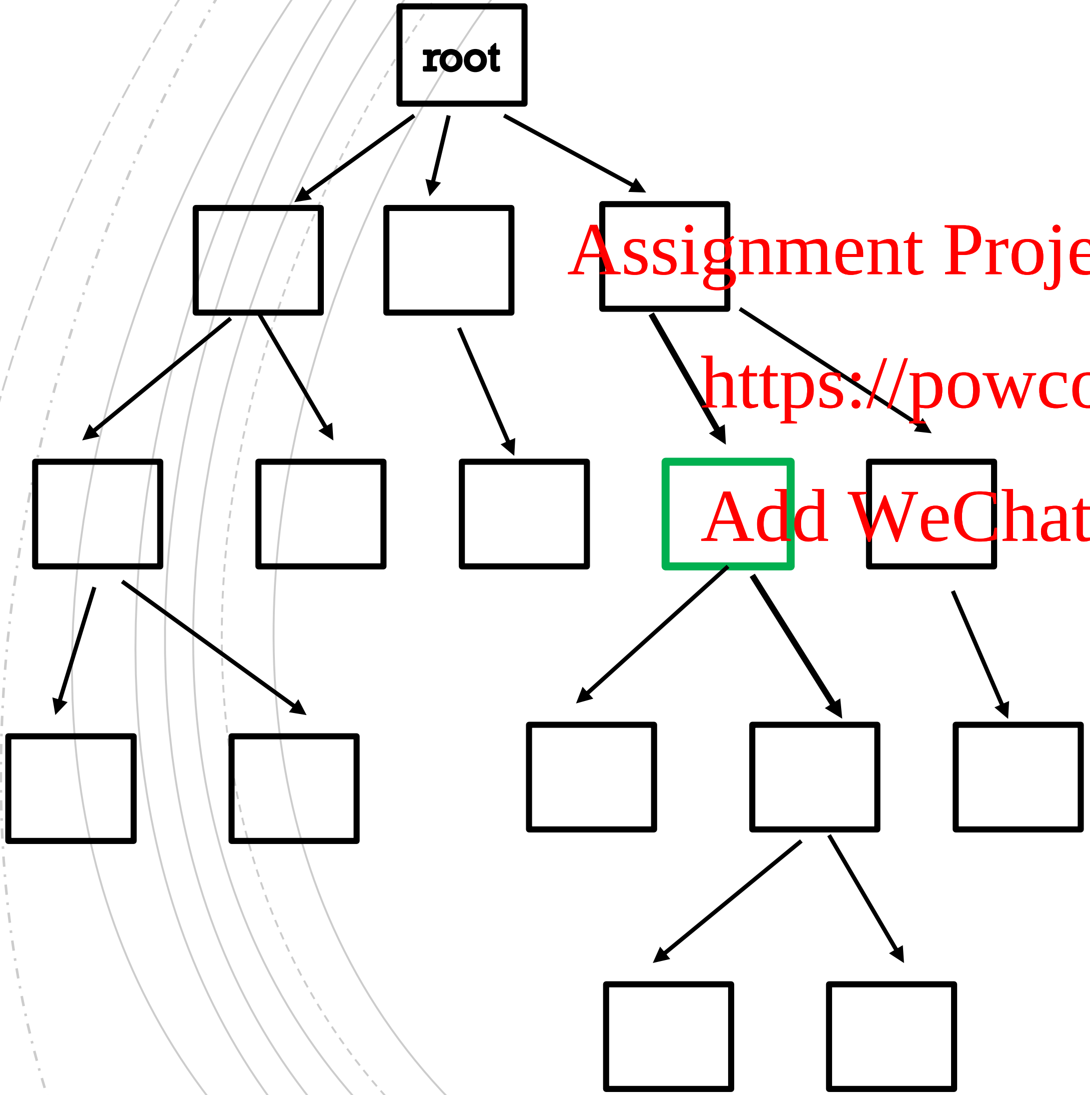
2

3

4

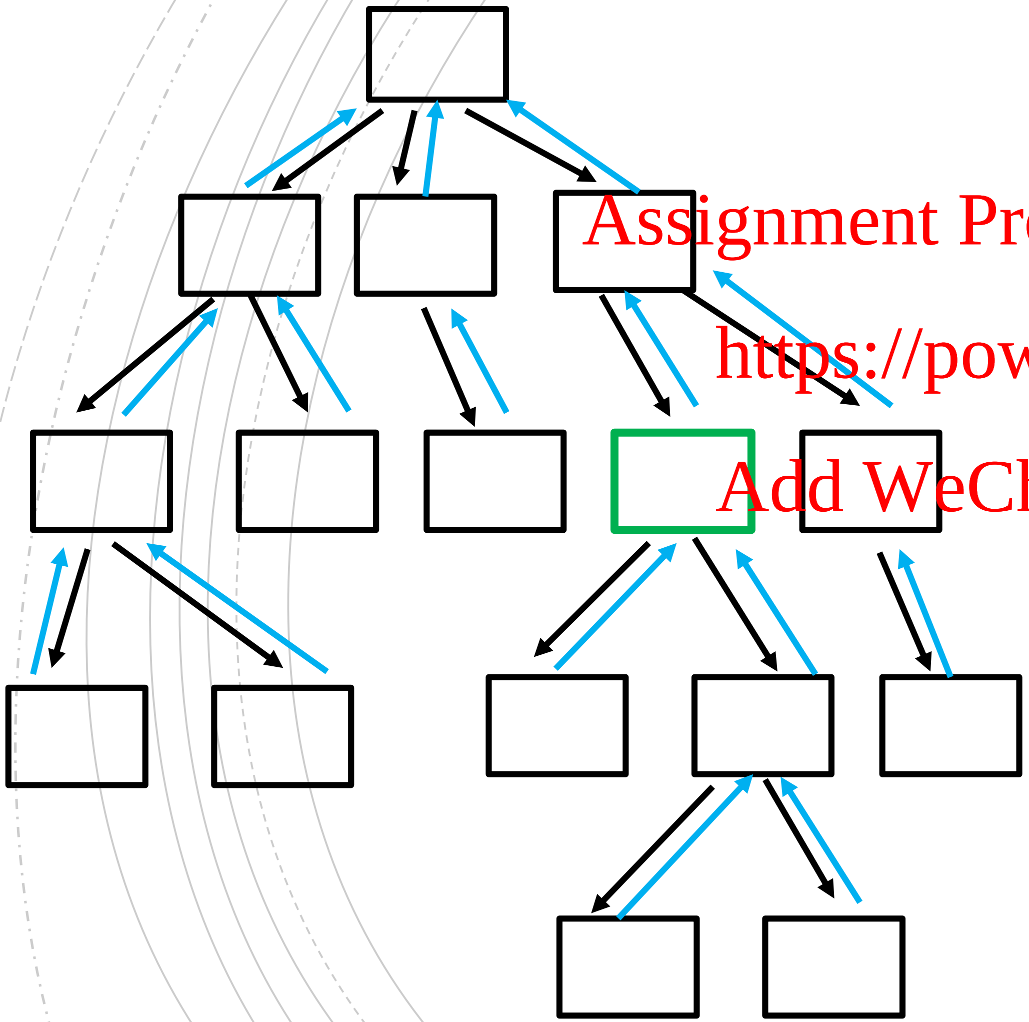
The *depth* (or *level*) of a node is the length of the path *from the root to the node*.

(ROOTED) TREE TERMINOLOGY



How can we compute the depth of a node **v**?

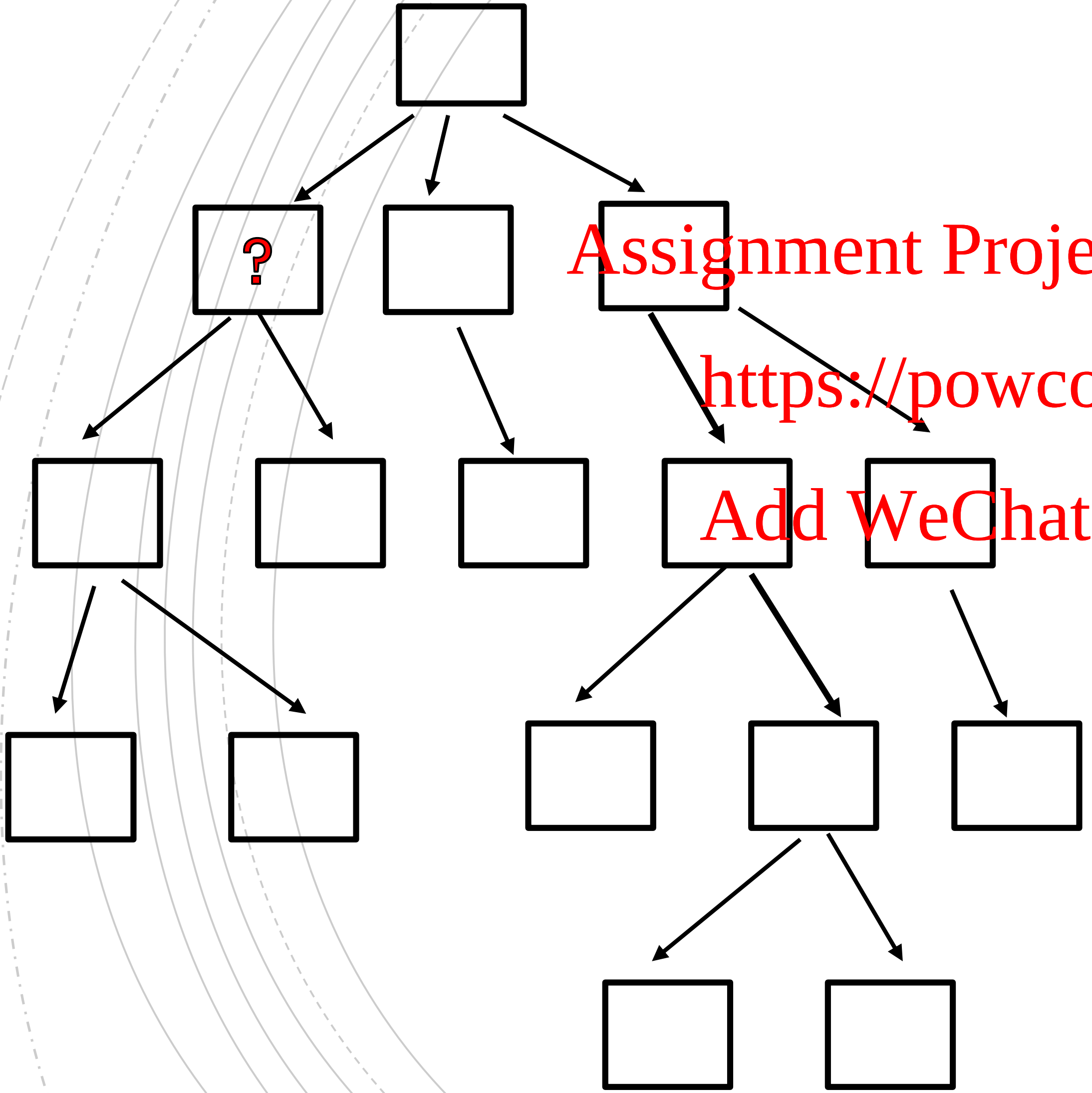
depth(v)



To do this efficiently we require nodes to have a parent link. This is analogous to a ‘prev’ link in a doubly linked list.

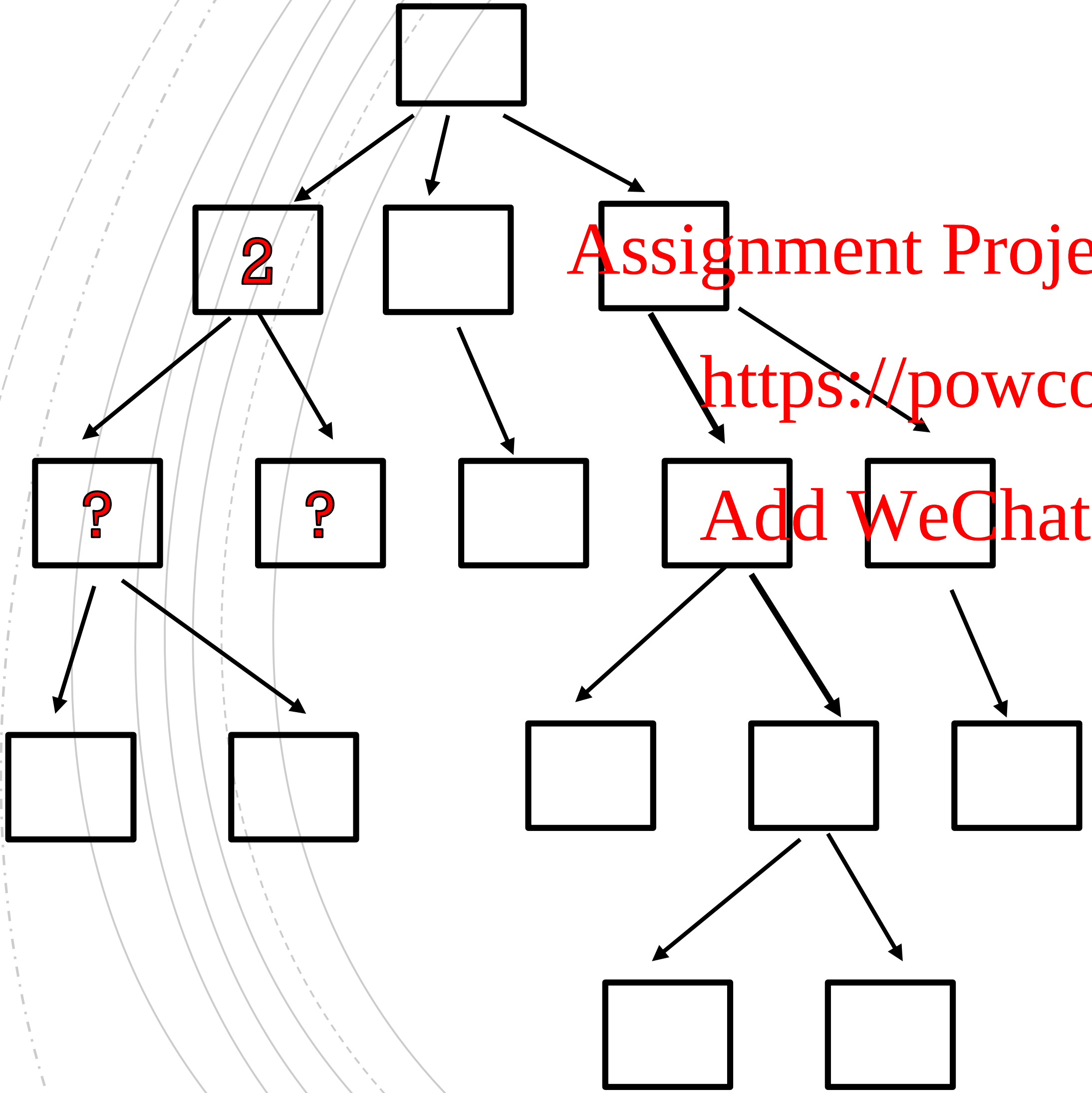
```
depth( v ){
   if(v.parent == null) //root
      return 0
   else
      return 1 + depth(v.parent)
}
```

(ROOTED) TREE TERMINOLOGY



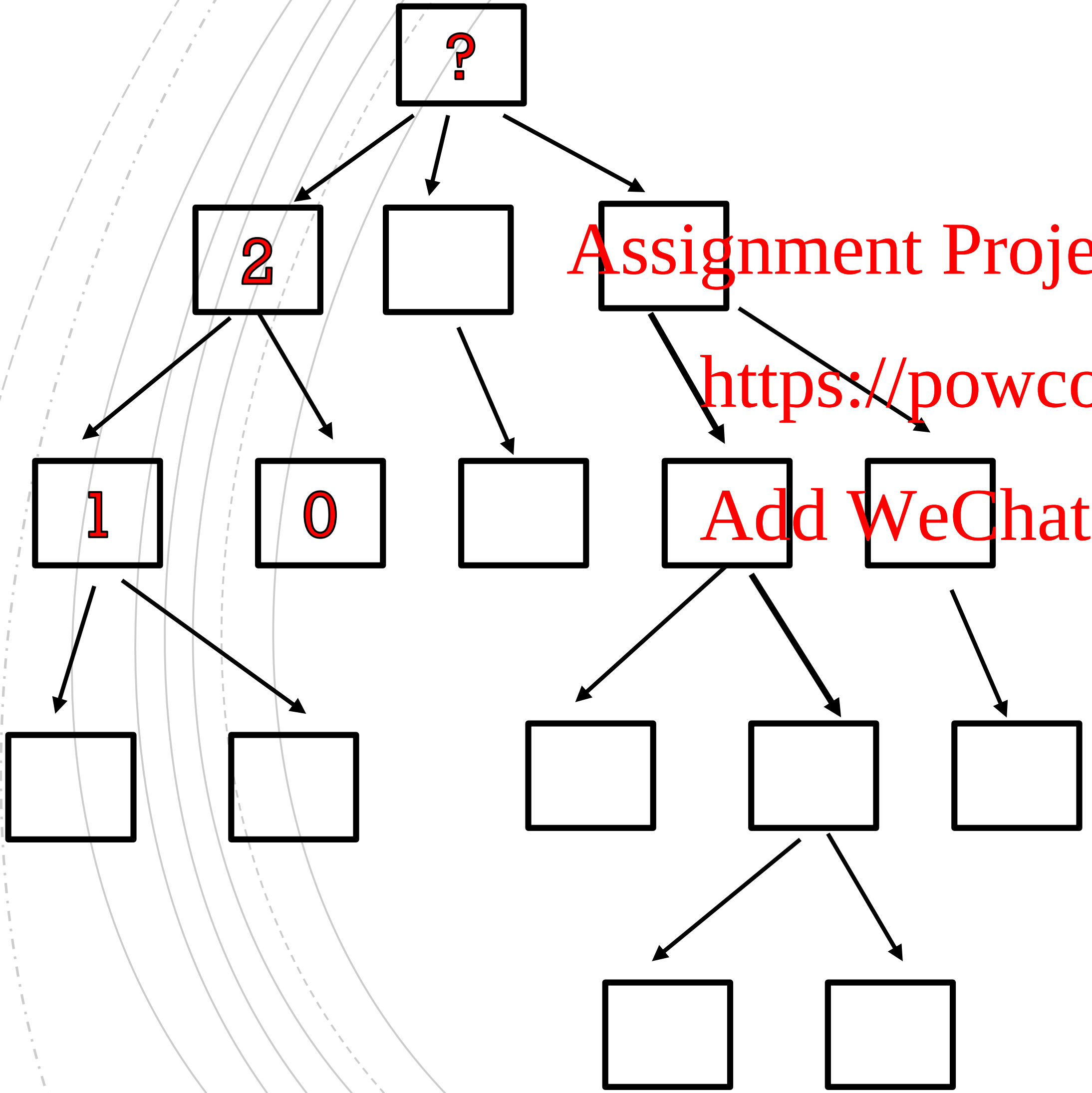
The *height* of a node is the maximum length of a path from that node to a leaf.

TREE TERMINOLOGY



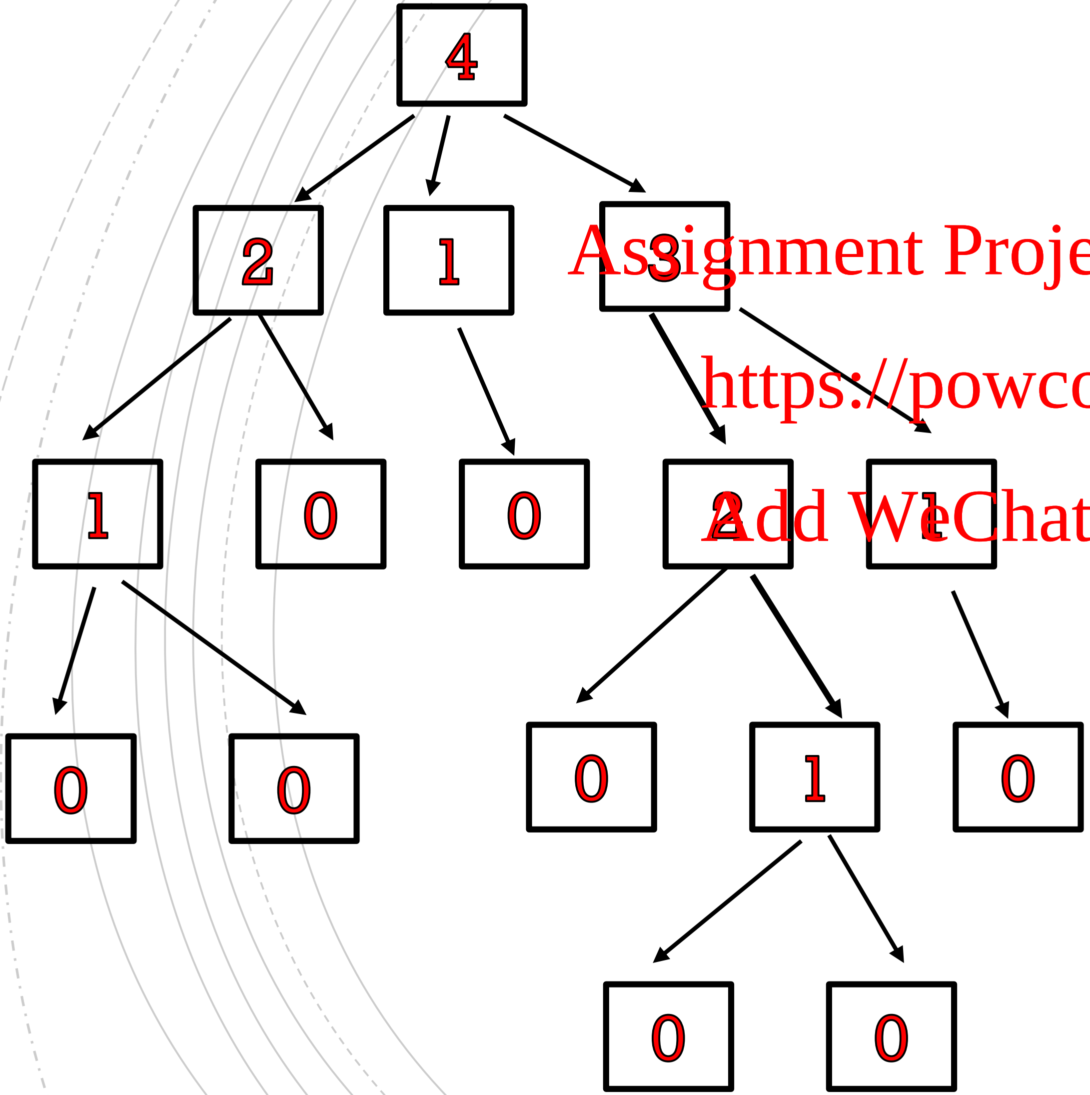
The *height* of a node is the maximum length of a path from that node to a leaf.

TREE TERMINOLOGY



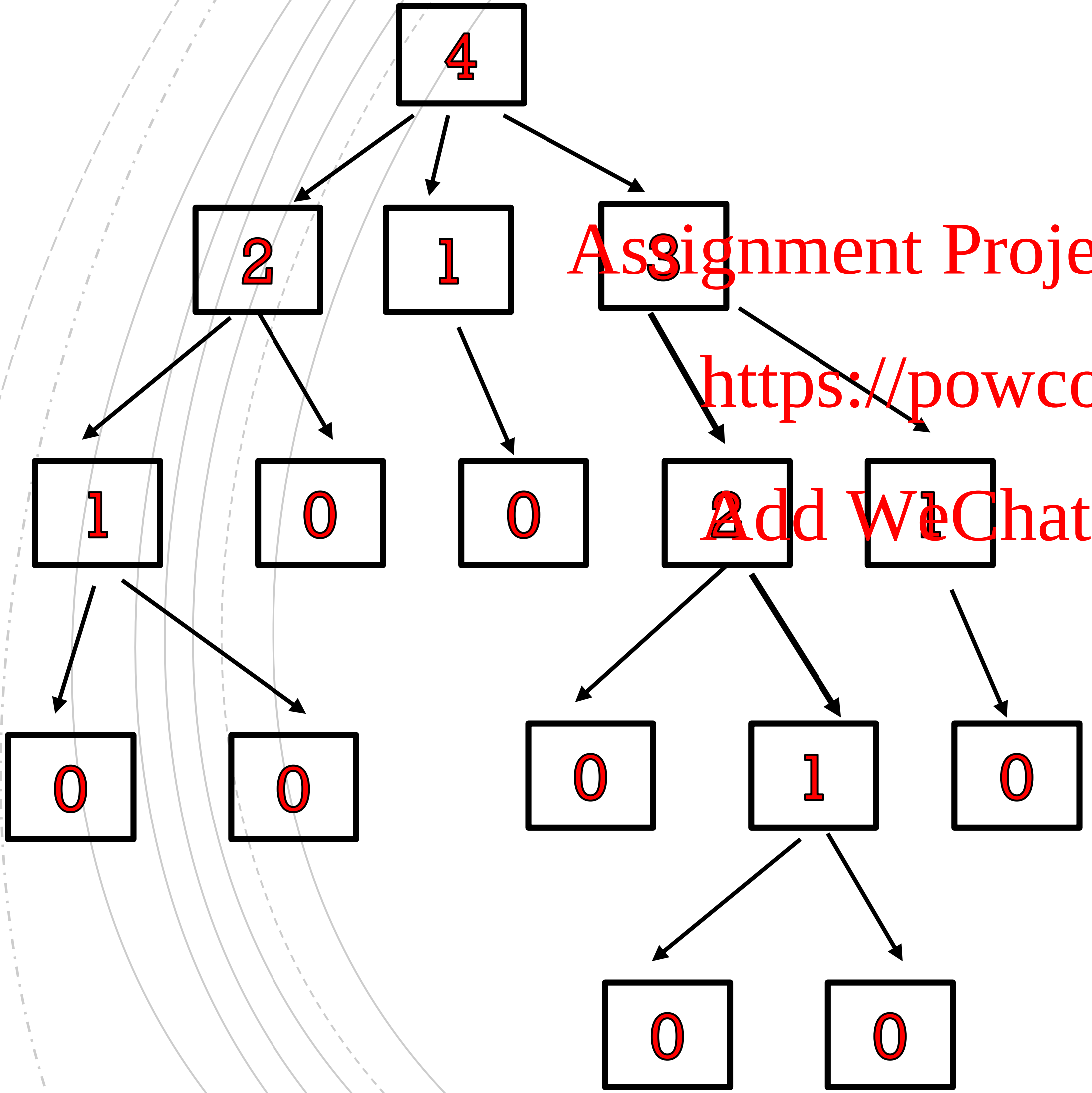
The *height* of a node is the maximum length of a path from that node to a leaf.

TREE TERMINOLOGY



How can we compute the height of a node v?

height(v)




```
height(v){
    if(v is a leaf)
        return 0
    else{
        h = 0
        for each child w of v
            h = max(h, height(w))
        return 1 + h
    }
}
```

IMPLEMENTATIONS

HOW TO IMPLEMENT A TREE IN JAVA?

Same idea as with linked lists:

- Create a data type to represent tree nodes.
- Represent a tree with a pointer to the root node.


```
class TreeNode<T>{

    T element;




}
```

HOW TO IMPLEMENT A TREE IN JAVA?

Same idea as with linked lists:

- Create a data type to represent tree nodes.
- Represent a tree with a pointer to the root node.

```
class TreeNode<T>{
    T element;
    ArrayList<TreeNode<T>> children;
    TreeNode<T> parent; // optional
}
```

HOW TO IMPLEMENT A TREE IN JAVA?

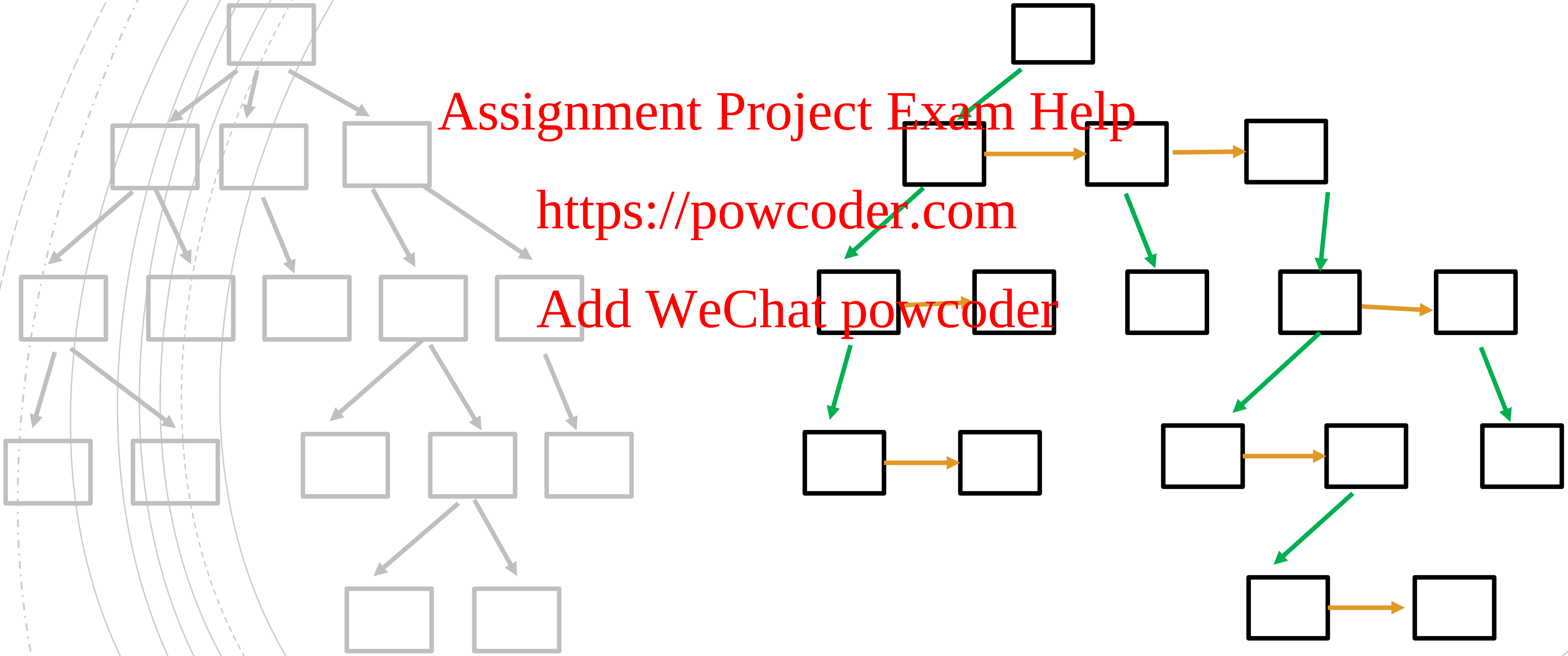
Same idea as with linked lists:

- Create a data type to represent tree nodes.
- Represent a tree with a pointer to the root node.

```
class Tree<T>{
   TreeNode<T> root;
       :

   class TreeNode<T>{
       T element;
       ArrayList<TreeNode<T>> children;
       TreeNode<T> parent; // optional
   }
}
```

ANOTHER COMMON IMPLEMENTATION: `first child, next sibling'

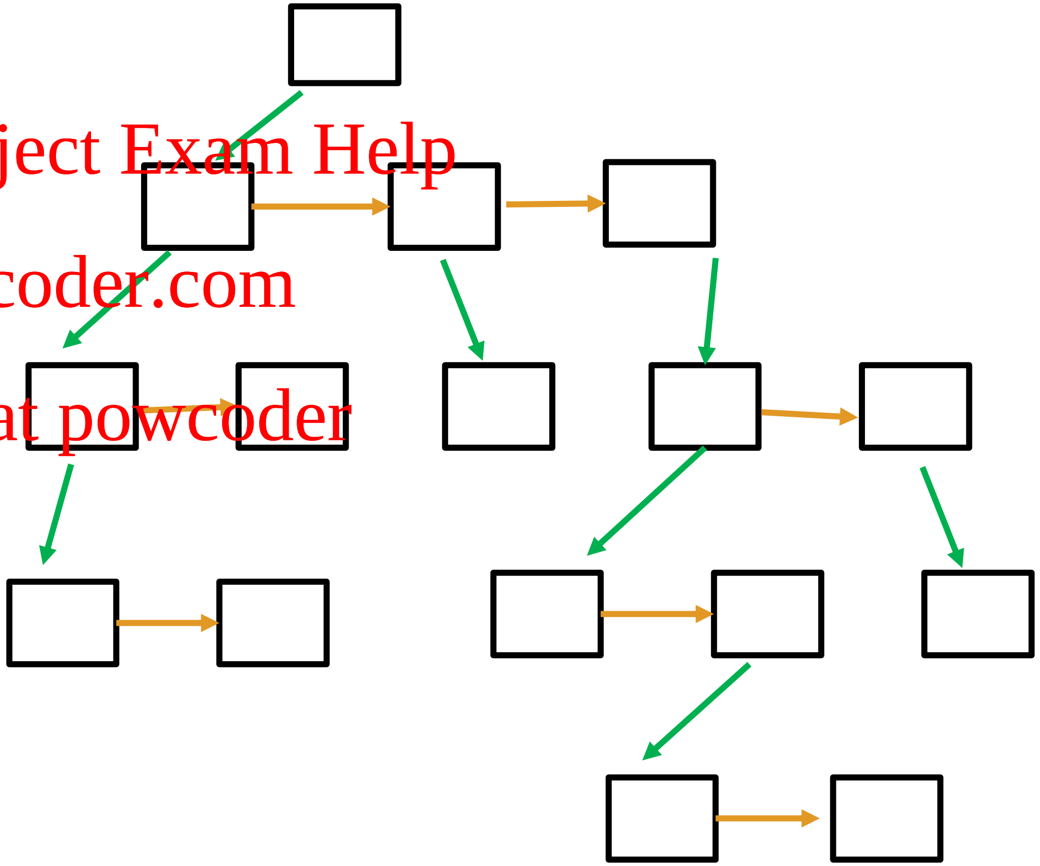


ANOTHER COMMON IMPLEMENTATION: `first child, next sibling'

(similar to singly linked lists)

```
class Tree<T>{
    TreeNode<T> root;
        :

    class TreeNode<T>{
        T element;
        TreeNode<T> firstChild;
        TreeNode<T> nextSibling;
            :
    }
}
```

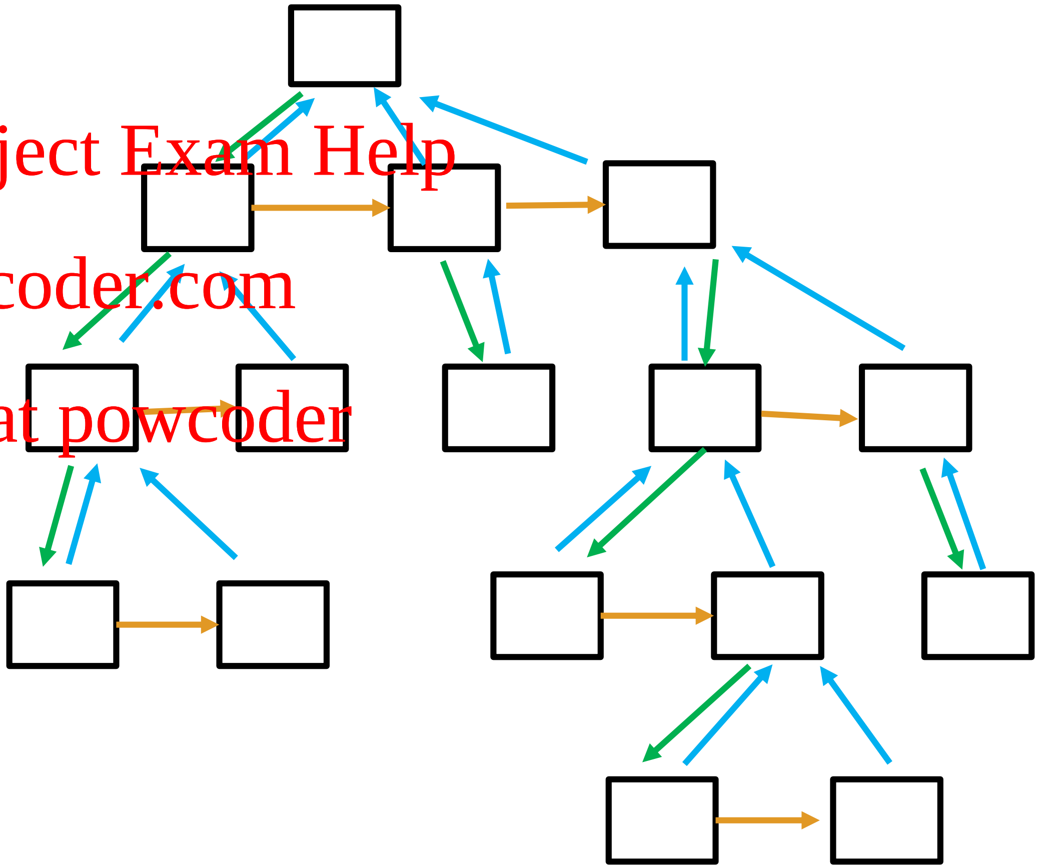



ANOTHER COMMON IMPLEMENTATION: `first child, next sibling'

(similar to singly linked lists)

```
class Tree<T>{
   TreeNode<T> root;
      :

   class TreeNode<T>{
      T element;
      TreeNode<T> firstChild;
      TreeNode<T> nextSibling;
      TreeNode<T> parent;
   }
}
```

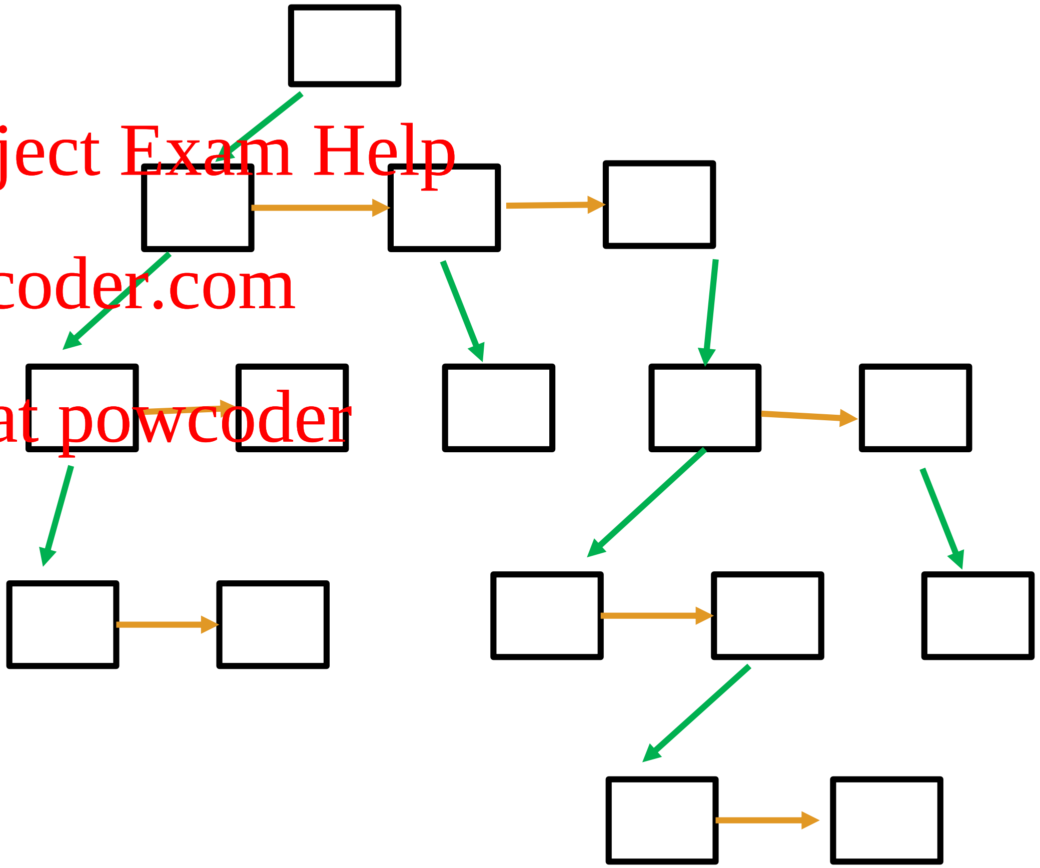



A TREE OF WHAT? EACH NODE HAS AN ELEMENT!

(NOT ILLUSTRATED ON THE RIGHT)

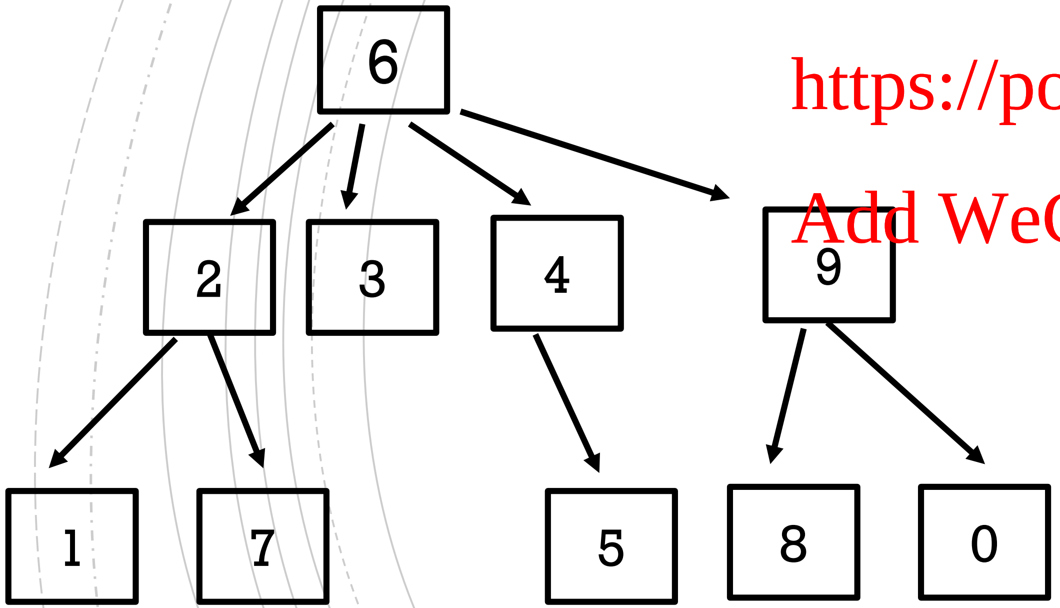
```
class Tree<T>{
    TreeNode<T> root;
        :

    class TreeNode<T>{
        T element;
        TreeNode<T> firstChild;
        TreeNode<T> nextSibling;
            :
    }
}
```



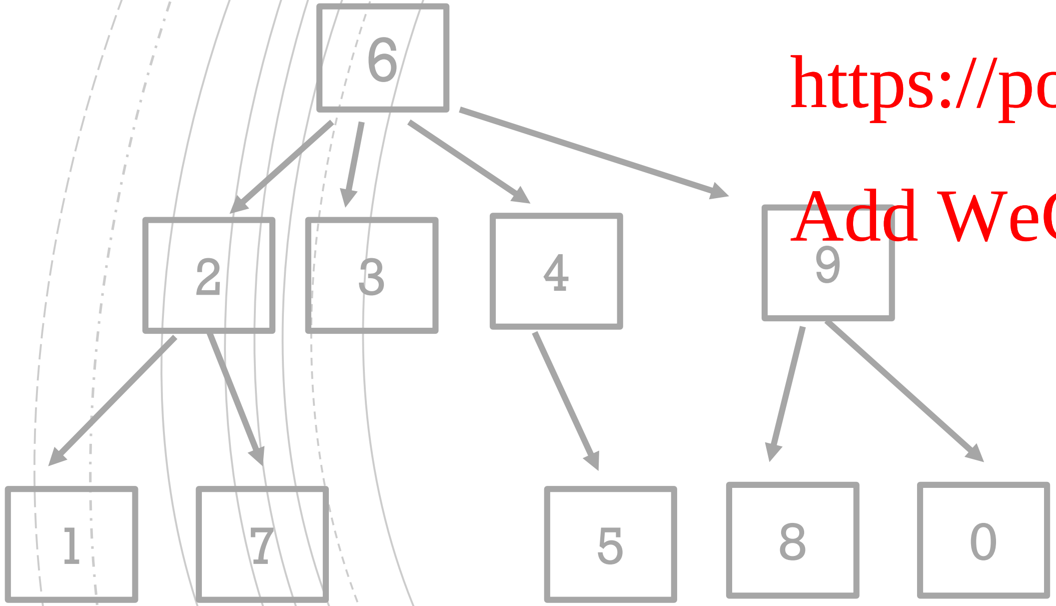
ANOTHER EXERCISES

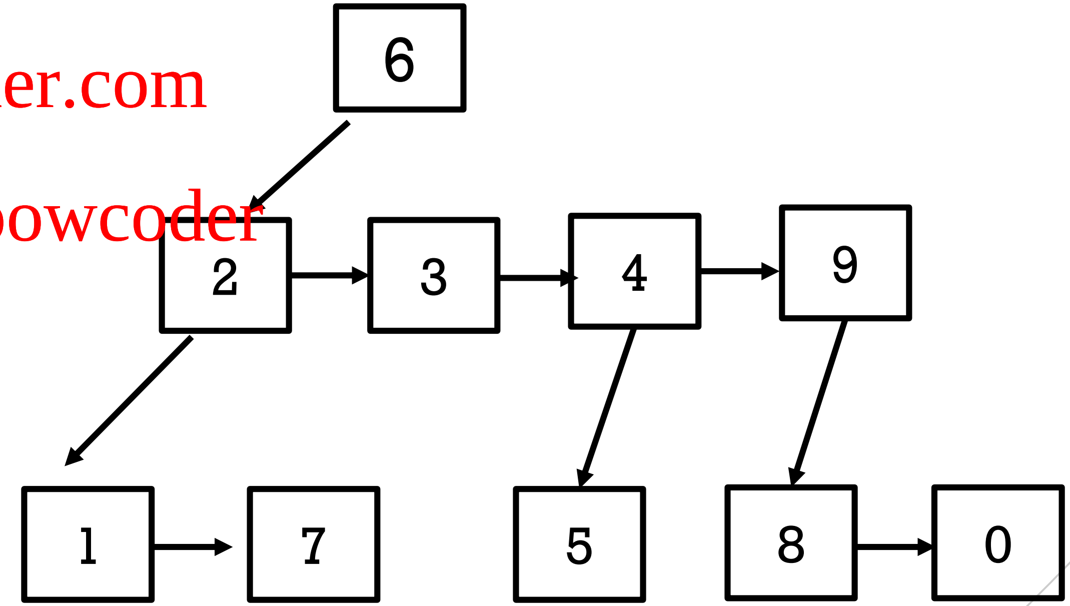
Write this tree using the first child, next sibling representation.

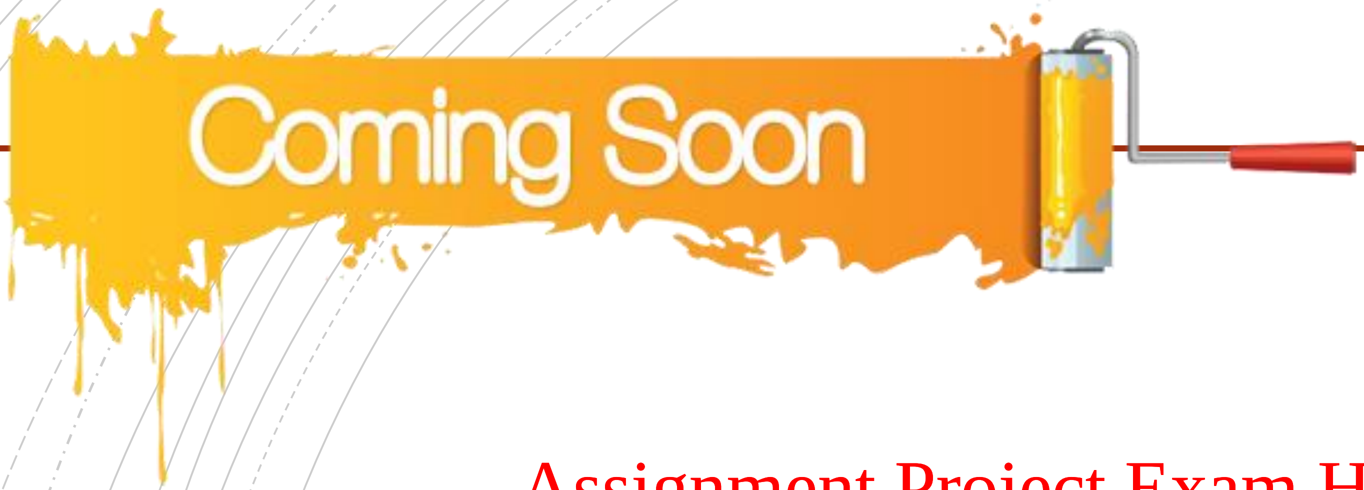


SOLUTION

Write this tree using the first child, next sibling representation.







In the next video:

- Tree Traversals