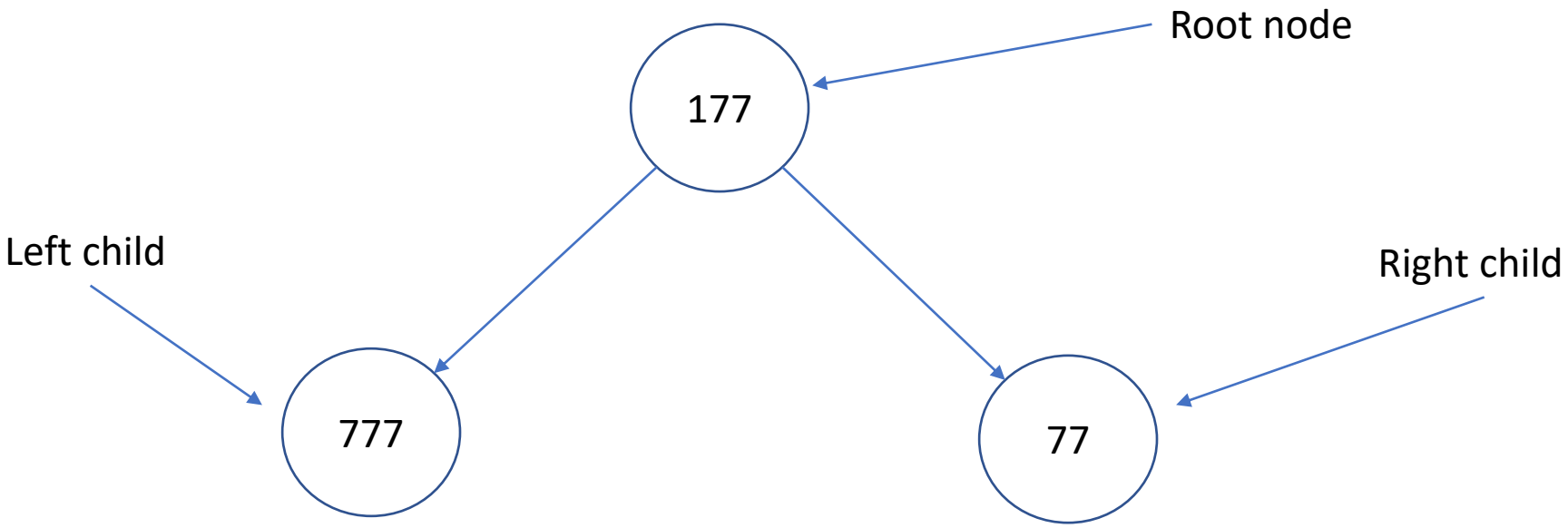
Binary Tree

Binary Tree

- A binary tree is a very commonly used data structure.
- Its pictorial representation :



Binary Tree

- So, a node in the tree could be represented by something like a struct which would look like:

```
class Node {
        int             value;      // assume int value
        Node            leftChild;
        Node            rightChild;
};
```

A binary tree has a root node which points to the root of the tree.

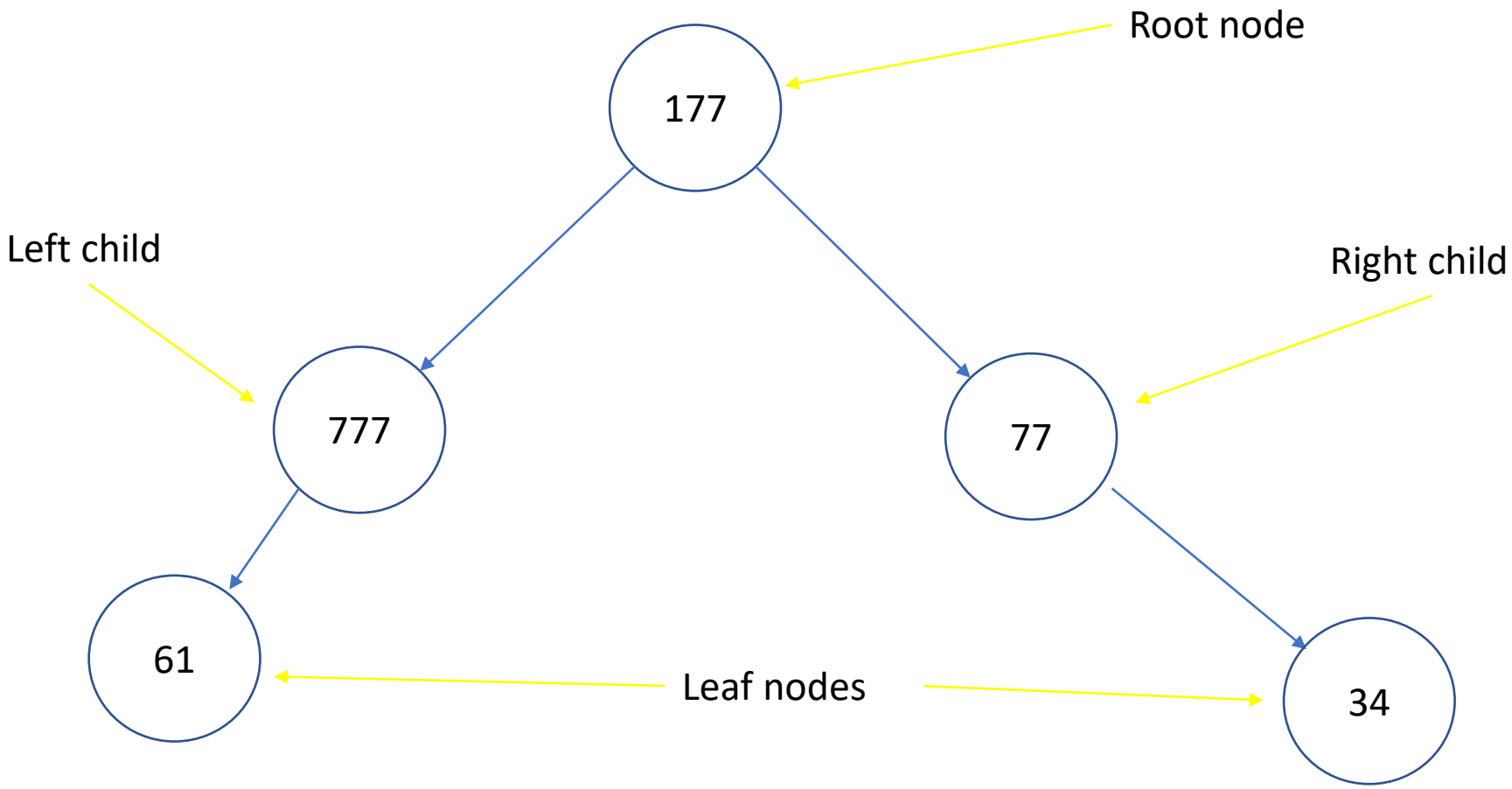
A binary tree may be empty, i.e., it has no nodes. In this case, the root would point to null.

A node can have a **max** of two children, a left and a right child.

If a node has no children, it is called a **leaf** node.

A root node can also be a leaf, and if so, the binary tree has only one node, which is the root node.

Another example

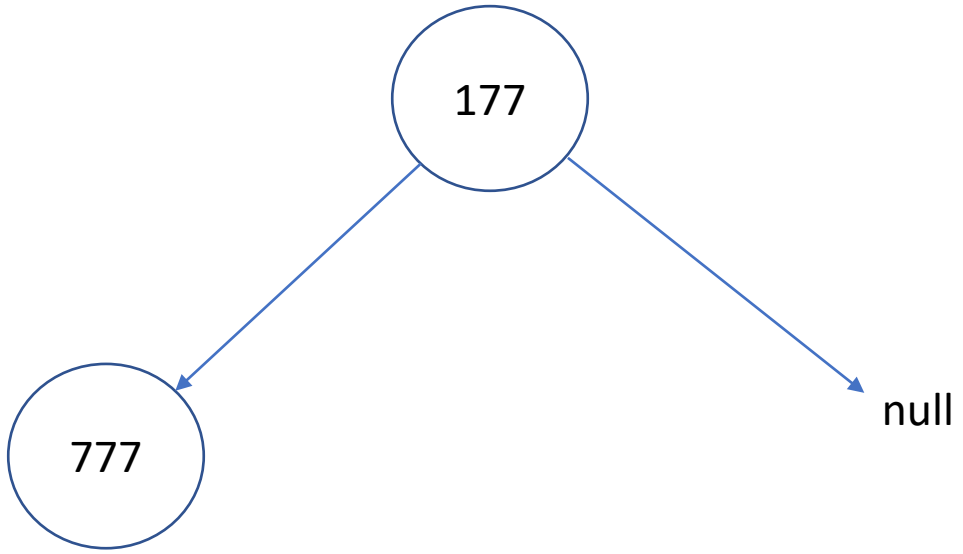


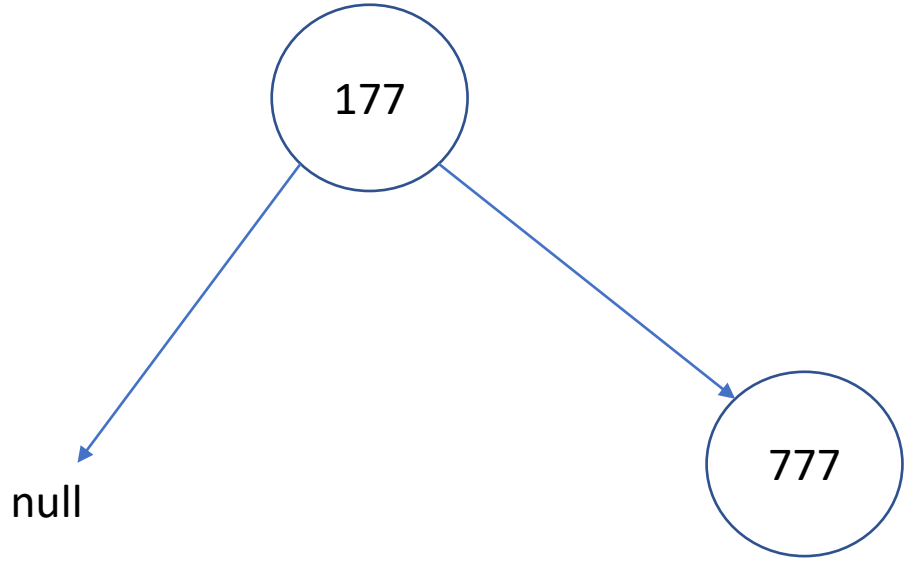
Another example

Tree example

- These two trees below are **different**, even though
  - they have same number of nodes and
  - And the nodes contain the same values.

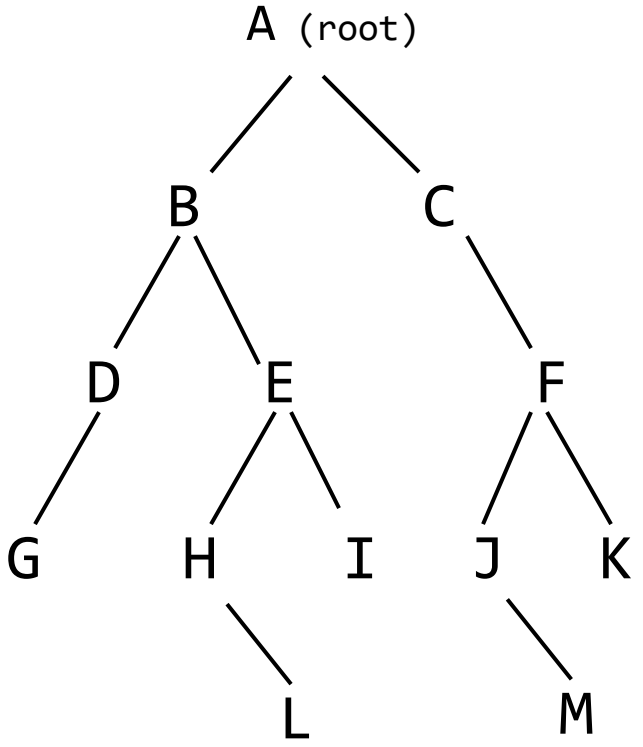
- The one on left has a root which has a **left child only.**
- The one on right has a root which has a **right child only.**





Some definitions

- A is the root.
- B is its left child, and C is its right child.
  - B and C are siblings
- G, I, K, L and M are leaf nodes.
- B and C are at the same level in the tree.
- DEF are at the same level, as are GHIJK nodes.



Binary Tree

A binary tree has a **root** node which points to the root of the tree.

A binary tree may be empty, i.e., it has no nodes. In this case, the root would point to null.

A node can have a max of two children, a **left** and a **right** child.

If a node has no children, it is called a **leaf** node.

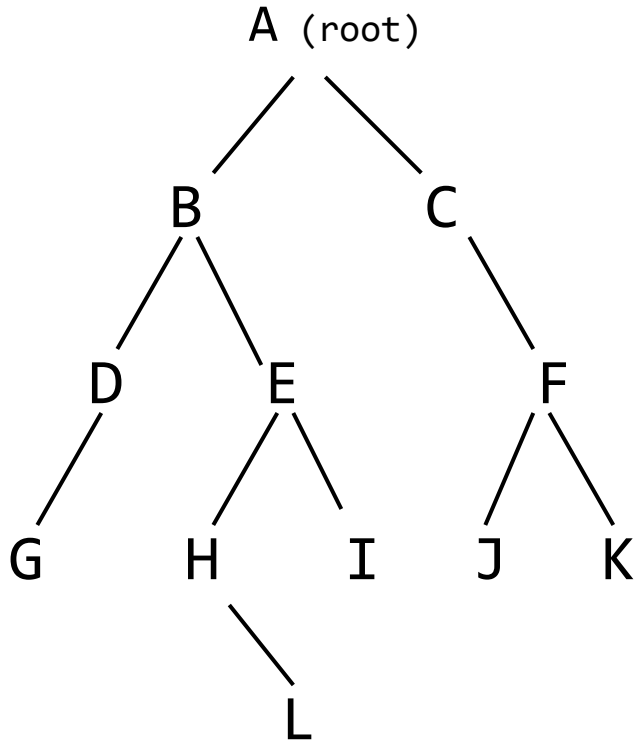
A root node can also be a leaf, and if so, the binary tree has only one node, which is the root node.

```
class Node {
    int         value;      // assume int value
    Node        leftChild;
    Node        rightChild;
};
```

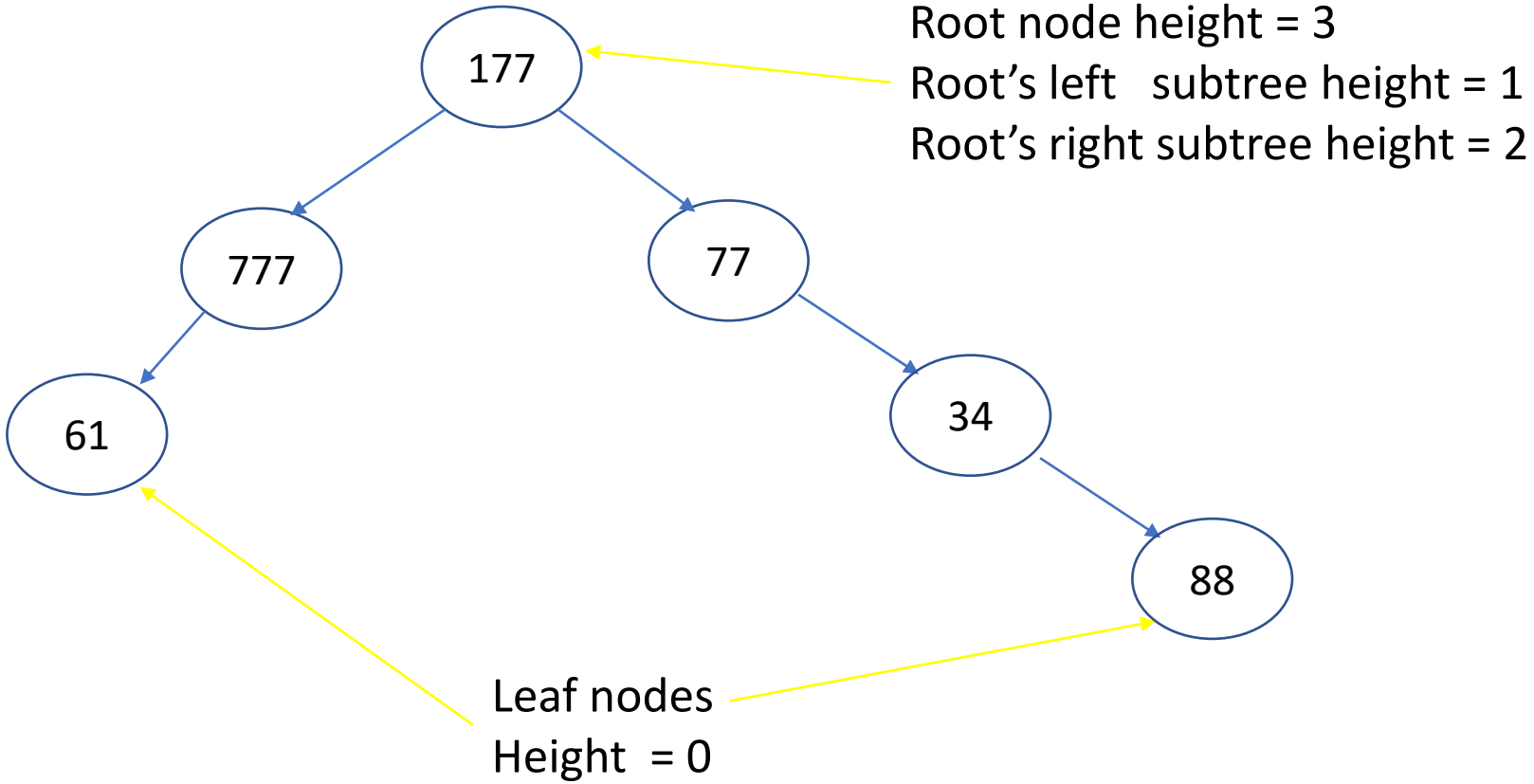
Height and Depth

- The height of a node is the number of edges from that node to the **deepest** leaf node.
- The height of a binary tree is == height of its root node.
- **The height of a leaf node is 0.**

- The depth of a node is the number of edges from that node to the root.
- **Depth of the root node is 0.**
- Depth of the deepest leaf node is the same as the height of the tree.



Height


177
777
77
61
34
88
Root node height = 3
Root's left subtree height = 1
Root's right subtree height = 2
Leaf nodes
Height = 0

Height

- Lets look at how to calculate the height of a binary tree.

- From the tree on the previous slide, we can say the following:

  - Height of a tree
    = height of its root node
    = 1 + height of root's left subtree
    OR
    = 1 + height of root's right subtree

  - Which one do we take ?

Height

- In an earlier slide, we had said the definition of height of a binary tree:
  - The height of a node is the number of edges from that node to the **deepest** leaf node.

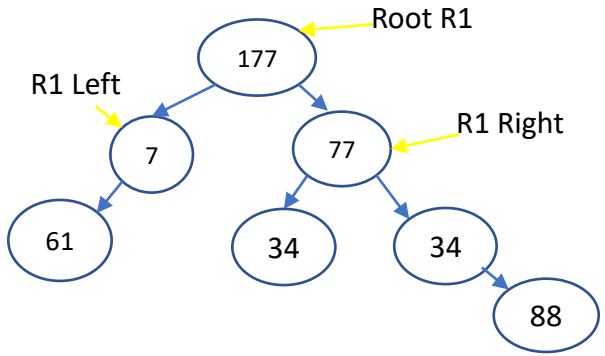
- So we really want the **max** of the following two:
  - height of root's left subtree
  - height of root's right subtree

- So, now we have:

  - Height of a tree
    = height of its root node
    = 1 + **MAX** (height of root's left subtree, height of root's right subtree)

Now, lets look at the next slide to figure out how to calculate height of the sub trees…

- To calculate height of a sub tree, we can again say the following:
  - Height of a sub tree
    = height of the sub tree's root node … **Call it R1**
    = 1 + **MAX** (height of R1's left subtree, height of R1's right subtree)



Now, on previous slide we said:

- Height of a tree
  = 1 + **MAX** (height of root's left subtree (call it **R1Left**), height of root's right subtree(call it **R1Right**))

- And then we have:
- Height of R1Left =
  - 1 + **MAX** (height of R1Left's *left* subtree, height of R1Left's *right* subtree)
  - And we will continue to determine heights of all subtrees similarly

- Height of R1Right will be computed in a manner similar to that of R1Left.

What does this technique remind you of (something we have talked about many times earlier classes)?

LAB 1 of 2

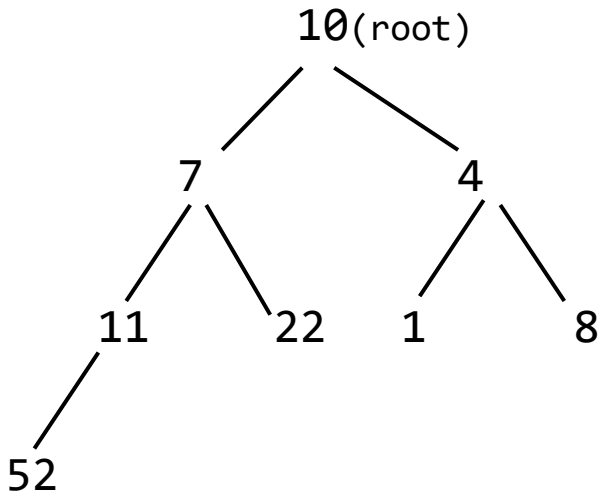
- Write code to

    1. Construct the binary tree below
    2. Then compute and print the height of its root node.
    - If you can, write the two in separate functions (not necessary, but recommended)

    - So, your functions could look something like:

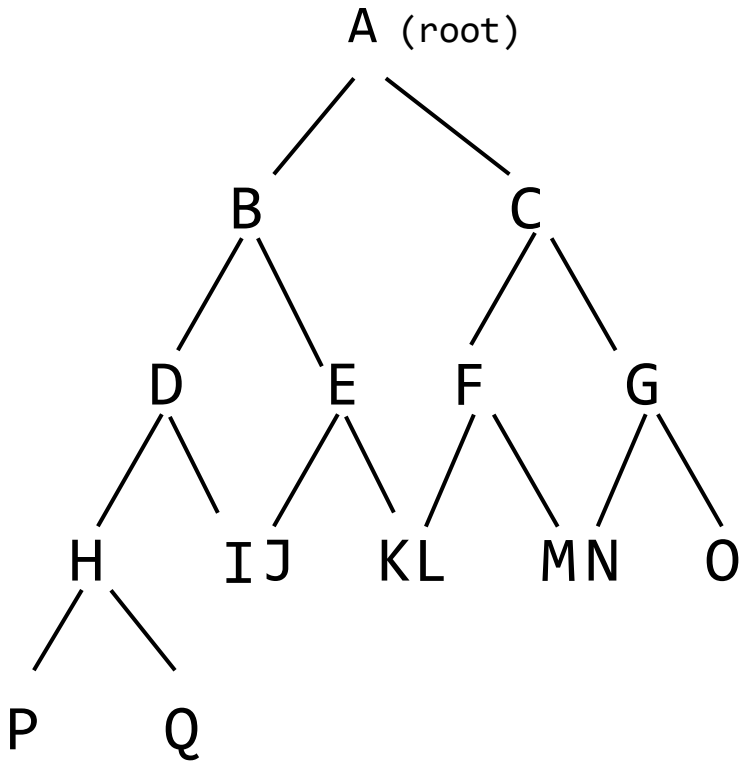
        - Node root = CreateMyTree()
        - int rootHeight = ComputeTreeHeight(root)

    - What is the height ur function computed?

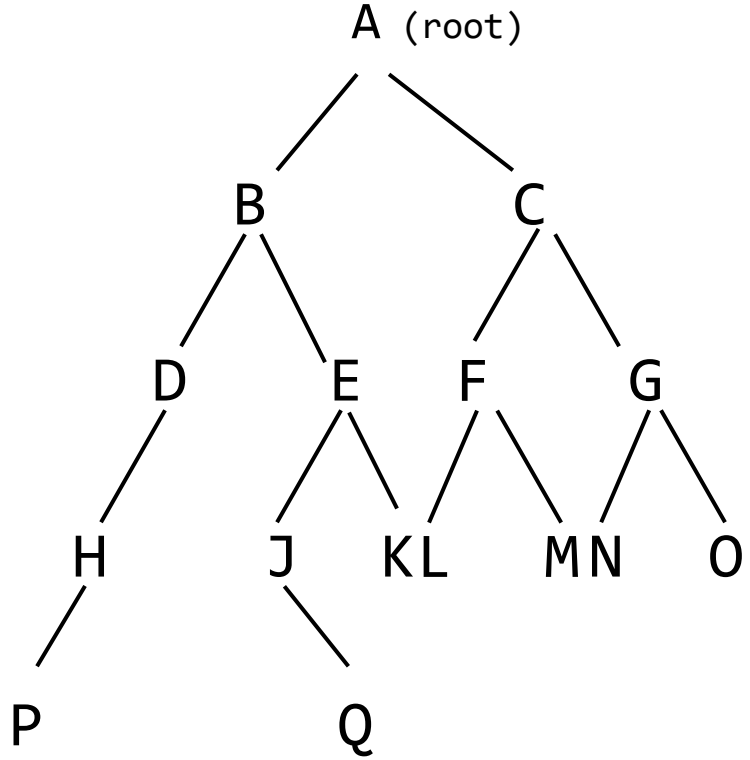


Complete binary tree

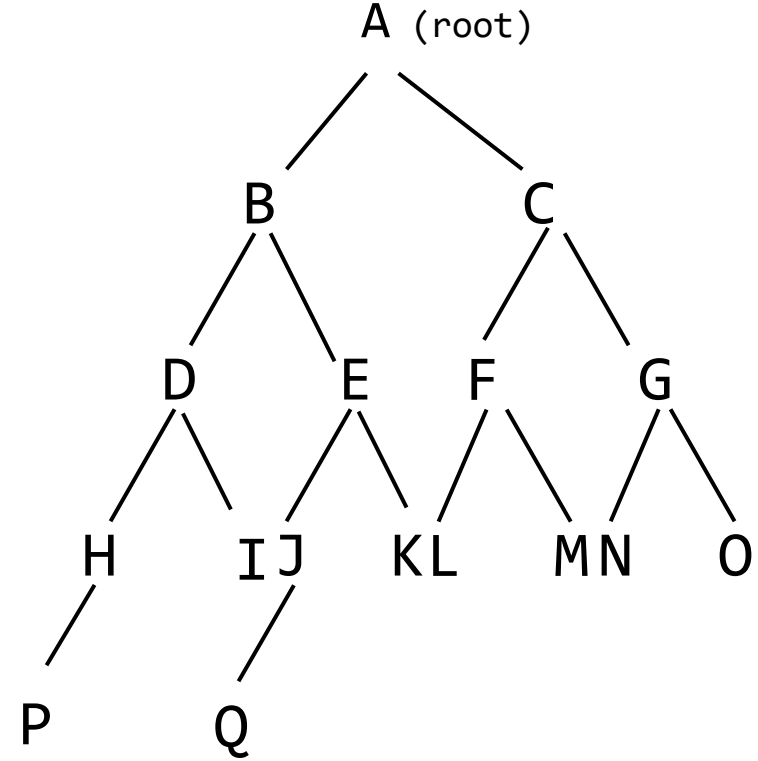
- A binary tree is *complete* if
  - all its nodes have two children,
  - with **the exception** of the deepest level,
    - which would be filled from left to right.
- In other words, any empty spots are only on the right side of the bottom level.
- All levels above the bottom level are fully occupied.



Complete tree



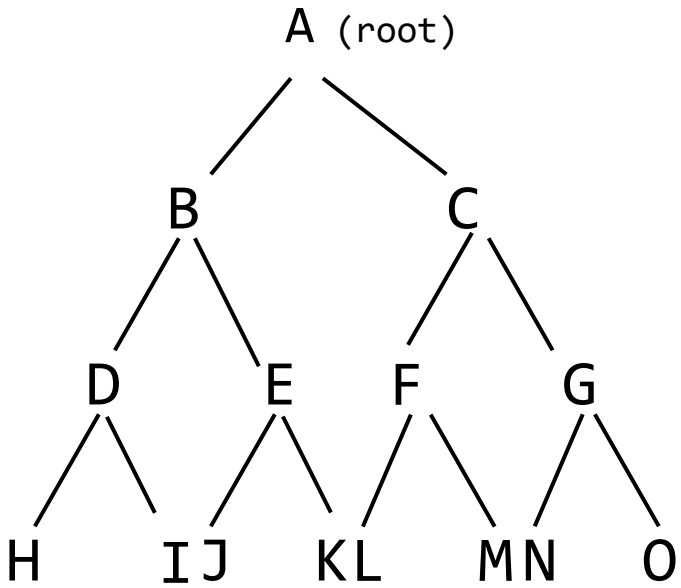
Not complete



Not complete

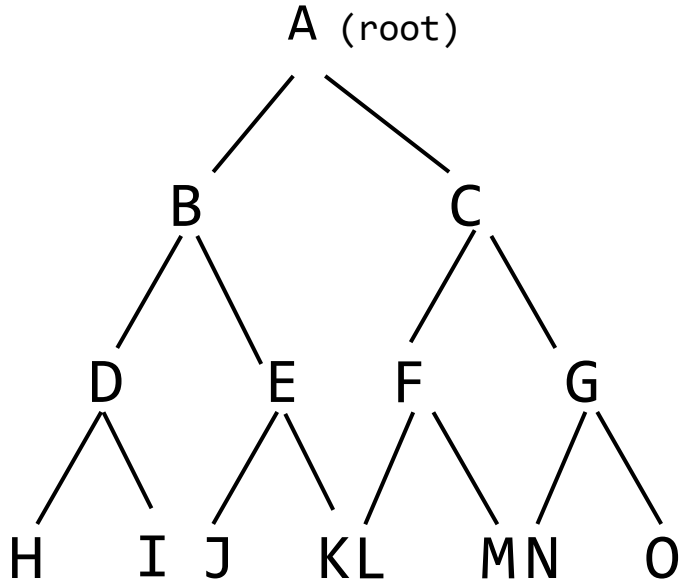
Number of nodes in a binary tree

- If there are N nodes in a binary tree (**assume** a complete and full tree).
- Then, the height of the tree is $\log_2 N$
- Another way to look at it:
  - If the height of a tree is h.
  - The number of nodes in the tree is $2^{(h+1)} - 1$
  - Examples:
    - 1
      - Binary tree with only root node.
      - h = 0
      - Number nodes = $2^{(0+1)} – 1 = 2 – 1 = 1$
    - 2
      - Binary tree with root node that has both children
      - h = 1
      - Number nodes = $2^{(1+1)} – 1 = 2^{(2)} – 1 = 3$



Interesting fact

- This is a full complete tree.
- Height = 3



- So, total number of nodes

  $N = 2^{(h+1)} - 1$

  $= 2^{(3+1)} - 1$

  $= 2^{4} - 1$

  $= 16 - 1$

  $= 15$

- Now, number of nodes in bottom level = 8, which is approximately N/2
- And, number of nodes in level just above bottom = 4, which is N/4
- And the level above that has N/8 nodes.

Interesting fact

- So, the number of nodes in the last 3 levels is
  - N/2 ← last level
  - N/4 ← 2nd last level
  - N/8 ← 3rd last level

- This is true for **any** full complete binary tree.
- Adding these :

  N/2 + N/4 + N/8 = 7N/8 = 0.875 * N

- So, the last 3 levels contain **87.5% (approx. 90%)** of all the nodes in the tree !!!

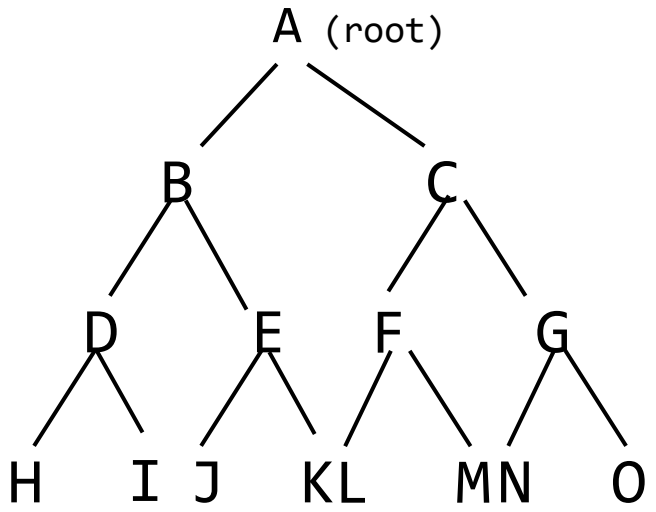
- This is true for a **complete binary tree with any number of levels**.

- Question:
  - Approximately how many total nodes in a **full complete** binary tree of height 30?
  - And how many nodes in the last 3 levels?

Counting nodes in a binary tree

Psuedocode for computing number of nodes (recursive)

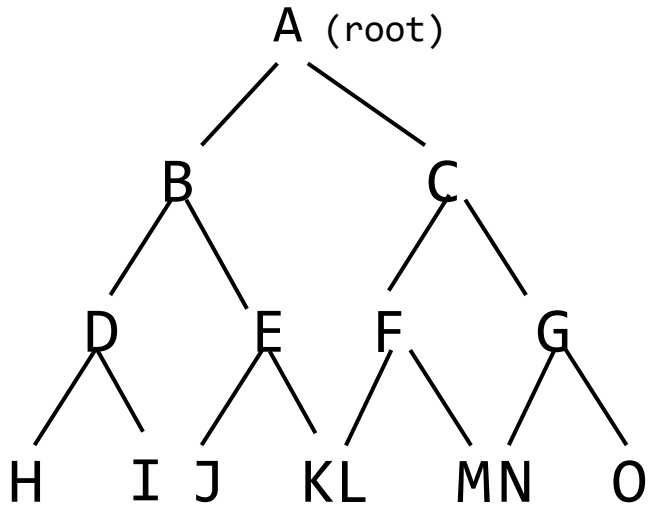
```
int GetNumberOfNodes( Node node )
{           ///  Question:  Do u see any bug(s) in this pseudo code?

    int numberOfNodes  = 1
                        + GetNumberOfNodes( node.left );
                        + GetNumberOfNodes( node.right );

   return numberOfNodes;
}
```



Counting nodes in a binary tree

Pseudocode for computing number of nodes (recursive)

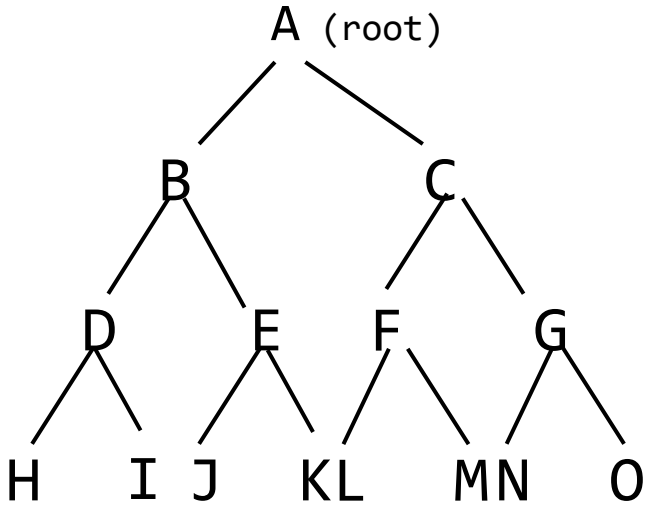
```
int GetNumberOfNodes( Node node )
{
   if (node == null)
      return 0;
   else
      return   1
               + GetNumberOfNodes( node >left )
               + GetNumberOfNodes( node > right );
}
```



Counting *leaf* nodes in a binary tree

Pseudocode for computing number of leaf nodes (recursive)

```
int GetNumberOfLeafNodes( Node node )
{
   if (node == null)
            return 0;
   if ( IsLeaf(node) )
            return 1;
   else
            return   GetNumberOfLeafNodes( node >left )   +   GetNumberOfLeafNodes( node > right );
}

bool IsLeaf( Node n )                    // Precondition:  n is not null
{
            return ( n->left == null )  && ( n->right == null );
}
```



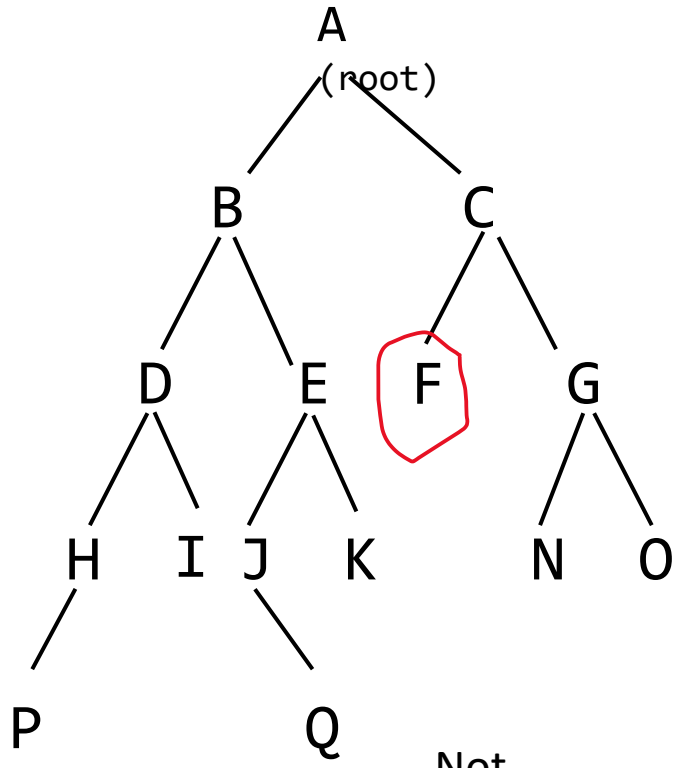
Note: Above condition is same as:

```
if ( ( n->left == null )  && ( n->right == null ))
        return true;
Else
        return false;
```

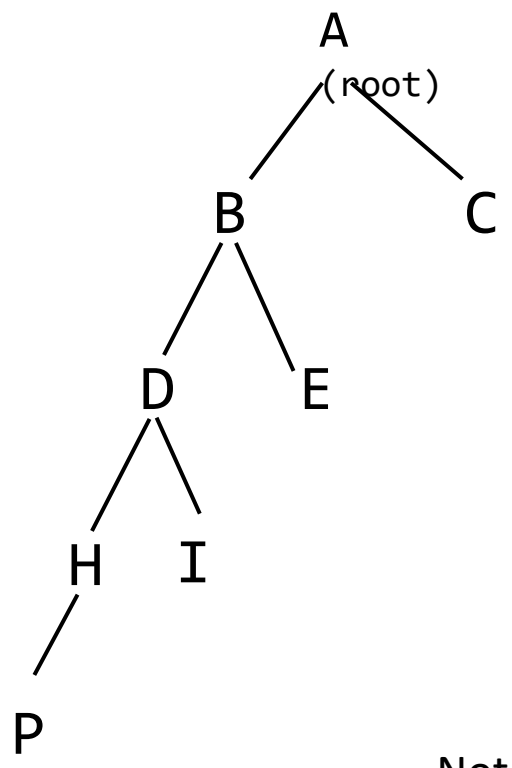
Balanced binary

- A binary tree is **balanced** if the maximum height **difference** of any node's right and left subtrees is 1.

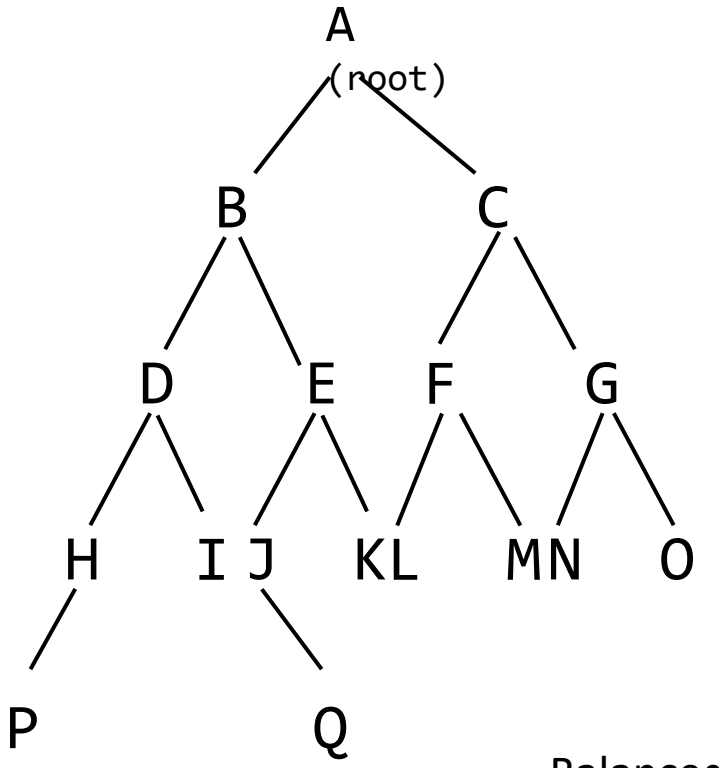
- Another way to look at it is that the maximum difference in depth between any two leaf nodes is 1.



Not balanced



Not balanced



Balanced

Tree traversal

- At a high level, there are these traversal techniques:
    - Depth first traversal
        - Pre order
        - In order
        - Post order
    - Breadth first traversal
        - Level order

Tree traversal

- Depth first traversal
    1. Think of a maze with an Entry point and an Exit point.
    2. We start at entry.
    3. We look at what options for next step.
    4. We take one of the options.
    5. Repeat from step 3

- Breadth first traversal
    - Level order

Traversal

Pre order traversal

- **Root**, Left child, right child
- A, B, D, E, C, F, G

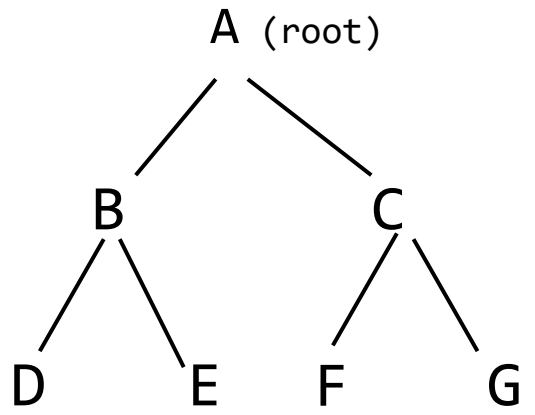
You can think of this as, starting at root node, you apply the (root, left, right) sequence to all nodes. So, root says I need to print myself first, then do pre order traversal on left child, and then pre order traversal on right child. So, we print A, then pre order on its left child B, which prints B, and then does pre order traversal on B's left child D, and so on.

In order traversal

- Left, **root**, right
- D, B, E, A, F, C, G

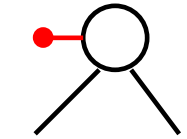
Post order traversal

- Left, right, **root**
- D, E, B, F, G, C, A

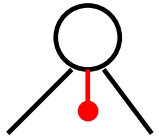


Traversal

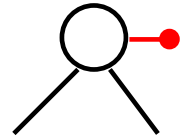
- Here's a trick to do the traversals easily (I learnt it from the internet).
- Imagine there is a pin attached to the nodes.
- Where the pin is attached depends on the traversal you want to do.



preorder



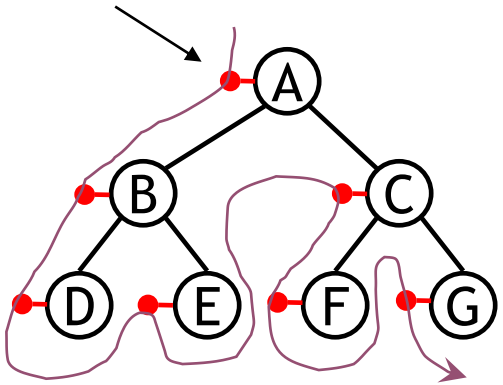
inorder



postorder

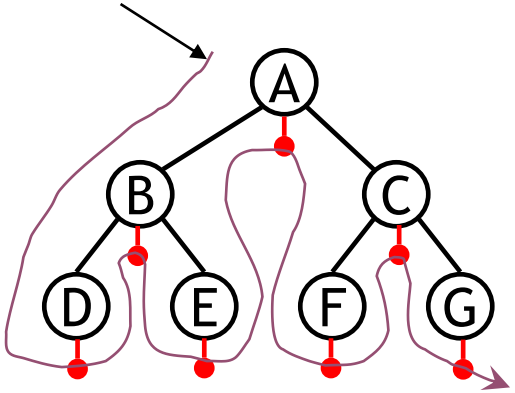
Now, you start at the root and traverse down its left side and collect the pins.

Start here



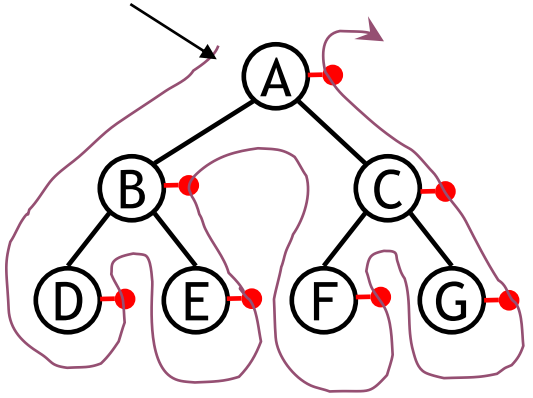
A B D E C F G

Start here



D B E A F C G

Start here



D E B F G C A

Traversal

```
class Node {
        int             value;      // assume int value
        Node            leftChild;
        Node            rightChild;
};

void InOrder( Node node ) {                 // print left child, then  root, then right child
                if (node == null)
                        return;

                InOrder( node.leftChild );
                print ( node.value );
                InOrder( node.rightChild );
};
```

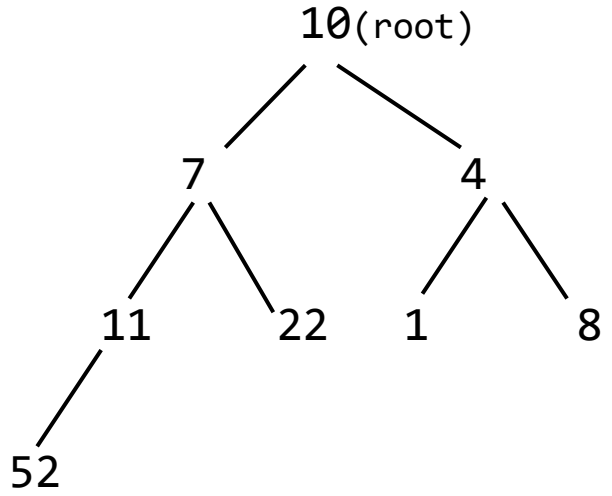
Traversal

```
void PreOrder( Node node ) {                    // print root, then left child, then right child
        if (node == null)
                return;

        print ( node.value );
         PreOrder( node.leftChild );
         PreOrder( node.rightChild );
};
```

LAB 2 of 2

- Write code to
  - construct the binary tree below
    - u can use the function from Lab1
  - pre order and post order traversal of a binary tree.

- Then call your post and pre order traversal functions on the tree you created.



LAB 1 of 2

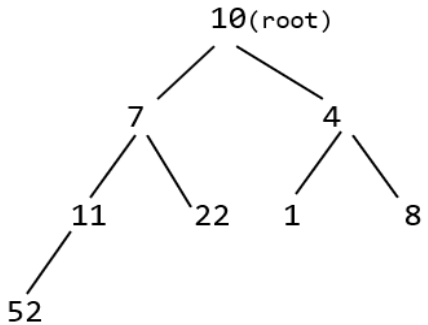
de to

onstruct the binary tree below
hen compute and print the height of its root node.
ou can, write the two in separate functions (not necessary, but recommended)

your functions could look something like:

```
Node root = CreateMyTree()
int rootHeight = ComputeTreeHeight( root )
```

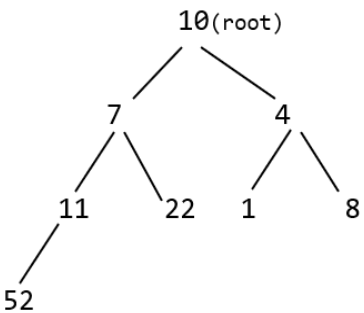
at is the height ur function computed?



LAB 2 of 2

- Write code to
  - construct the binary tree below
    - u can use the function from Lab1
  - pre order and post order traversal of a binary tree.

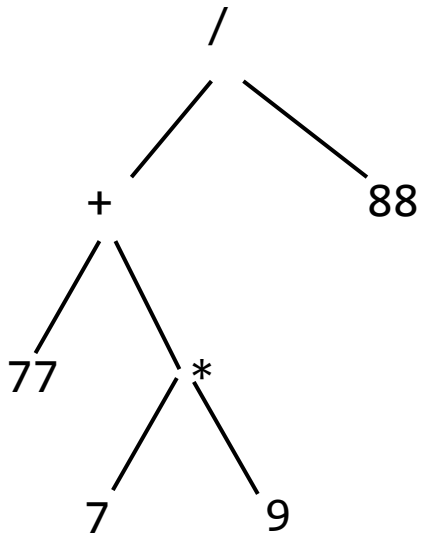
- Then call your post and pre order traversal functions on the tree you created.



Arithmetic expressions

- Binary trees can be used to represent arithmetic expressions.

- Lets look at this expression and the tree below: (77 + (7 * 9)) / 88



- What does an in order traversal of this tree output?

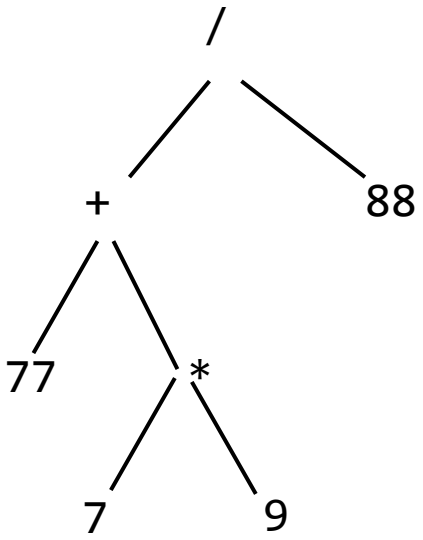
Arithmetic expressions

Expression: (77 + (7 * 9)) / 88

What does the post order traversal of this tree output?

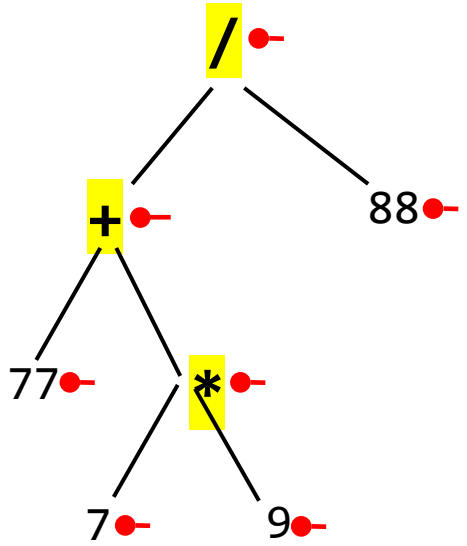
77 7 9 * + 88 /

Look at next slide for pins



Post fix notation

Expression: (77 + (7 * 9)) / 88



What does the post order traversal of this tree output?

Lets put the pins for easily getting the post order:

77 7 9 * + 88 /

This is a **post fix** representation of the given arithmetic expression.

Lets look at how we can evaluate this post fix representation to get the value of the expression.

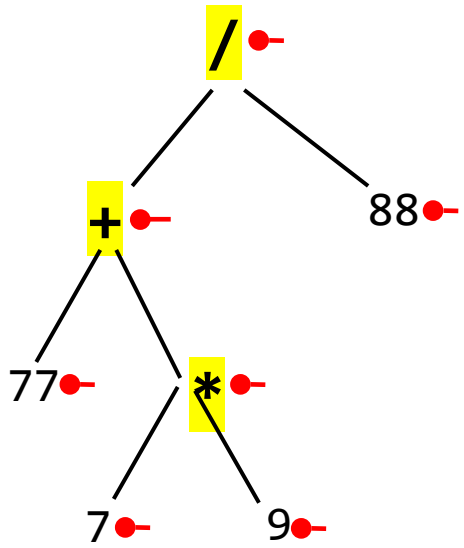
Post fix notation

Original expression:

(77 + (7 * 9)) / 88

Post fix:

77 7 9 * + 88 /



Algorithm for evaluating a post fix expression:

Note the expression contains values (operands) and the mathematical operator.

1. Create a stack.
2. Read the expression:
   A. If it is a value (operand), push it onto the stack.
   B. If it is an operator, pop two values from the stack and do the mathematical operation.
   C. Push the result back onto the stack.
   D. Loop back to step #2.

Post fix notation

Original expression: (77 + (7 * 9)) / 88

Post fix: 77 7 9 * + 88 /

Push 77, then Push 7, then Push 9

(read *, so need to pop twice)

Pop ← get 9

Pop ← get 7

Multiply 9 and 7 and push result (63) back onto the stack // 77 63

(read +, so need to pop twice)

Pop ← get 63

Pop ← get 77

Add 63 and 77 and push result (140) back onto the stack // 140

Push 88 // 140 88

(read /, so need to pop twice)

Pop ← get 88

Pop ← get 140

Divide 140 and 88 and push result back onto the stack // 1 (integer division assumed)

Input expression is finished.

Final result is what is on the stack. // 1