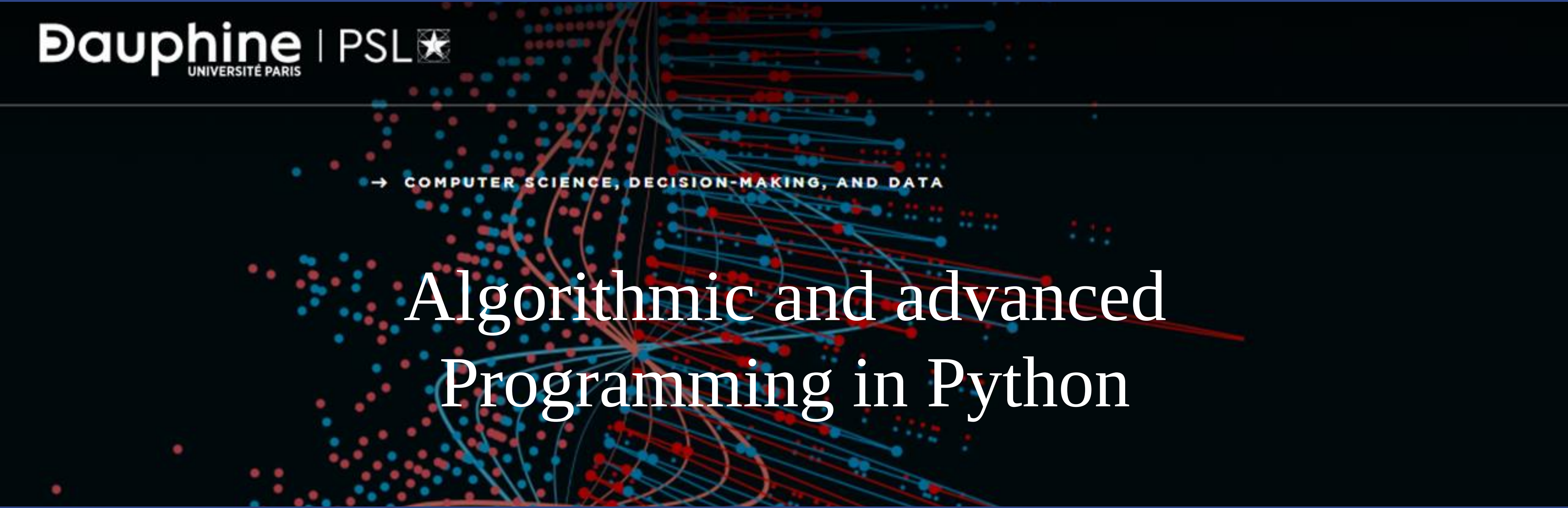


Algorithmic and advanced Programming in Python

Eric Benhamou eric.benhamou@dauphine.eu
Remy Belmonte remy.belmonte@dauphine.eu
Masterclass 5

Outline

Reminder of the objective of this course

- People often learn about data structures out of context
- But in this course you will learn foundational concepts by building a real application with python and Flask

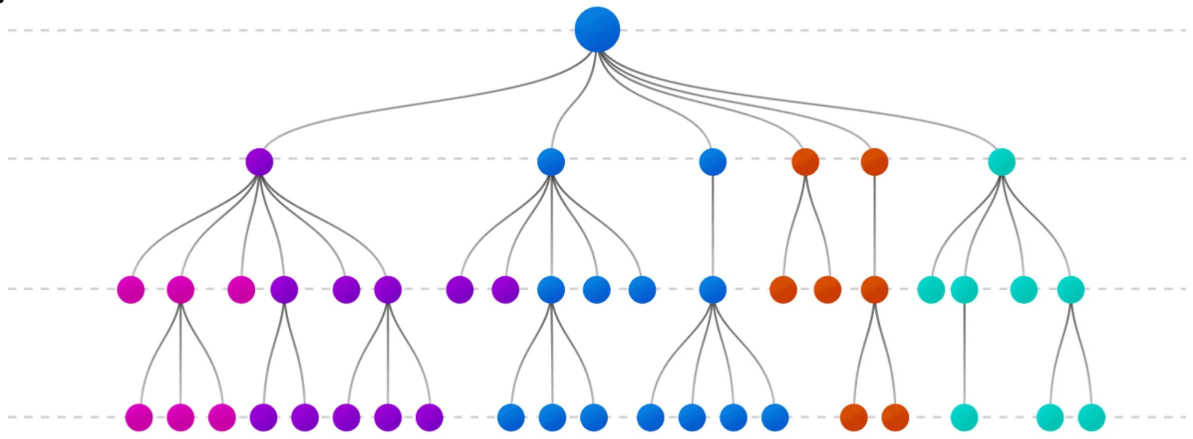
- To learn the ins and outs of the essential data structure, experiencing in practice has proved to be a much more powerful way to learn data structures

Reminder of previous session

- In Master class 4, we discuss about hashing
- Question: can you summarize what hashing is and how it works?

What is a Tree?

- A *tree* is a data structure similar to a linked list but instead of each node pointing simply to the next node in a linear fashion, each node points to a number of nodes.



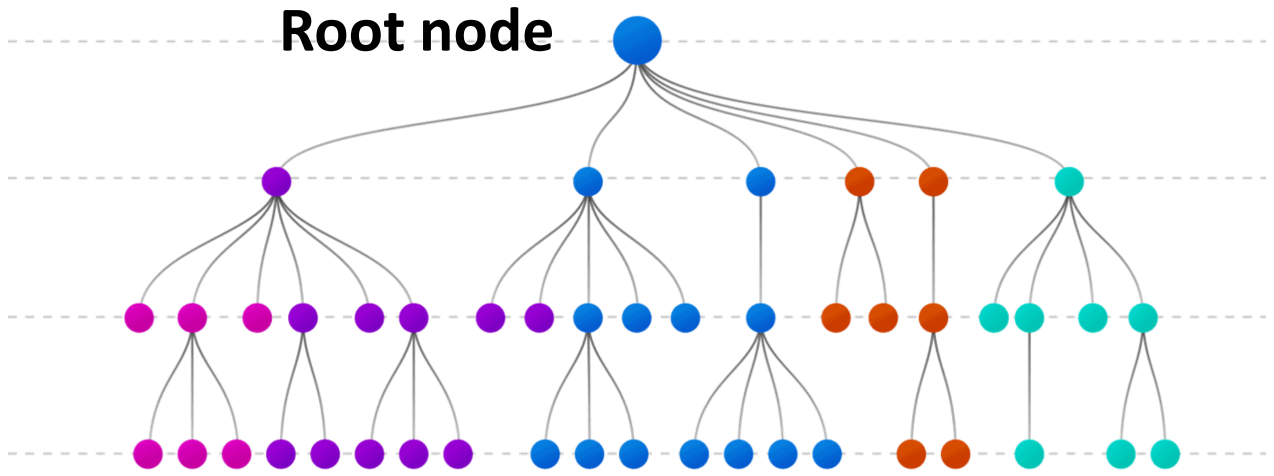
- Tree is an example of a non-linear data structure. A *tree* structure is a way of representing the hierarchical nature of a structure in a graphical form.

- Question: is the order important? can you cite some linear data structures?

Is the order important?

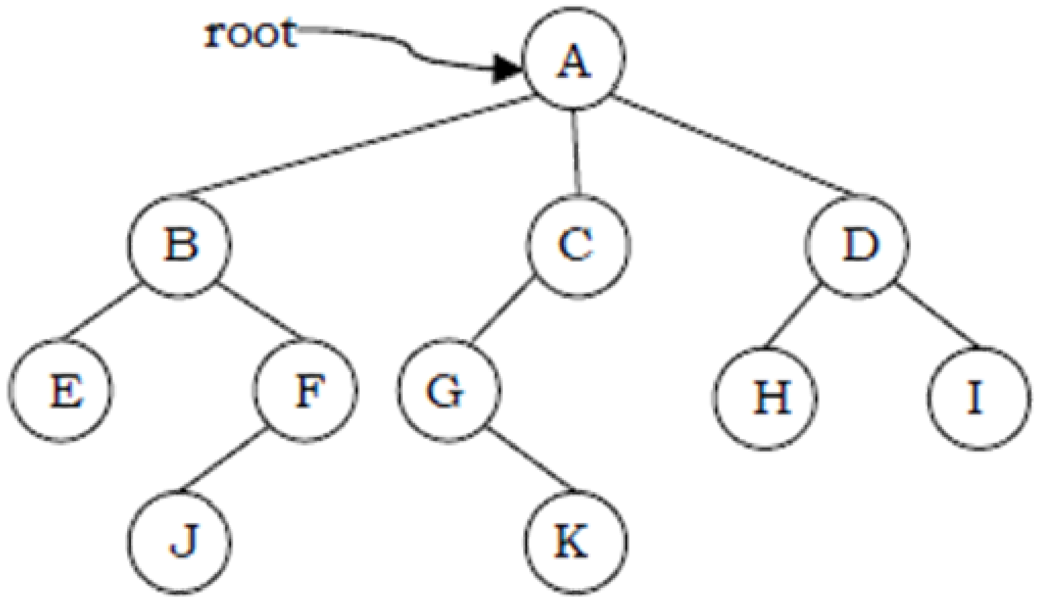
- In trees ADT (Abstract Data Type), the order of the elements is not important. If we need ordering information, linear data structures like linked lists, stacks, queues, etc. can be used.

Glossary



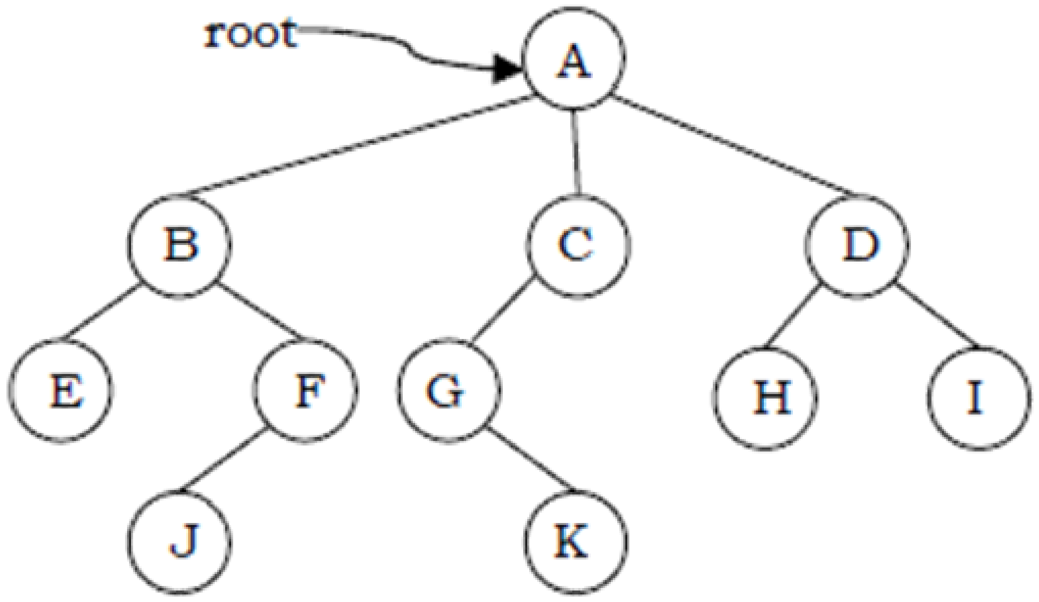
- The *root* of a tree is the node with no parents. There can be at most one root node in a tree (node A in the above example).
- An *edge* refers to the link from parent to child (all links in the figure).

Glossary



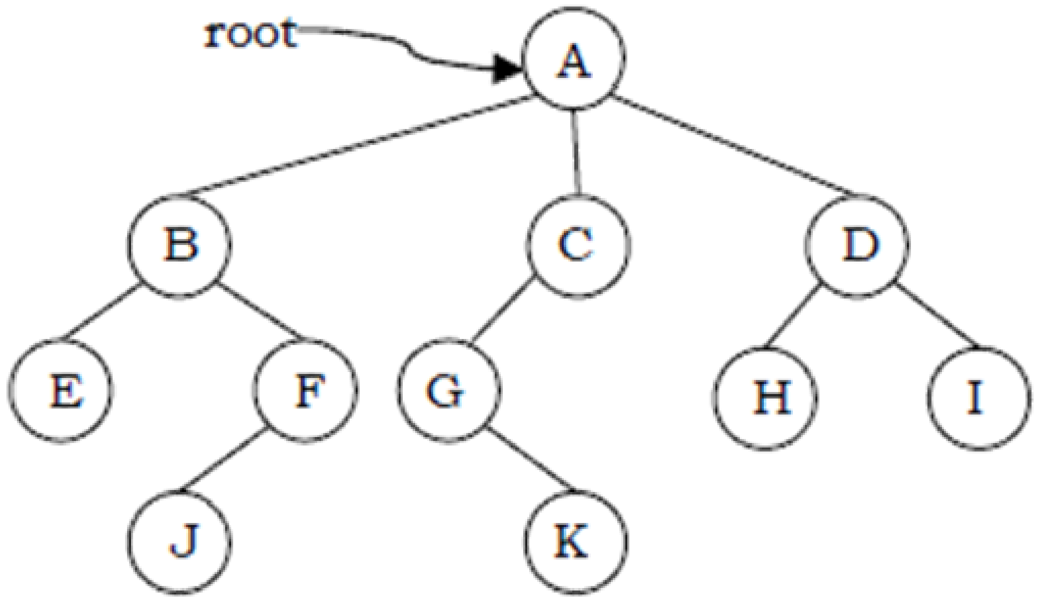
- A node with no children is called $leaf$ node.
- Question: can you cite all of them?

Glossary



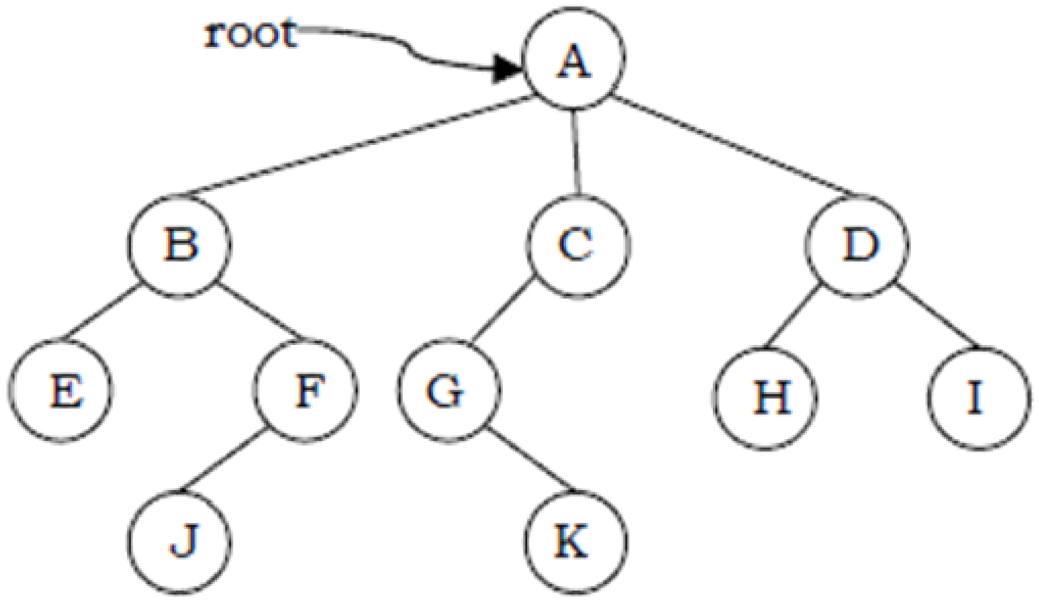
- A node with no children is called $leaf$ node.
- Question: can you cite all of them?
- (E, J, K, H and I).

Glossary



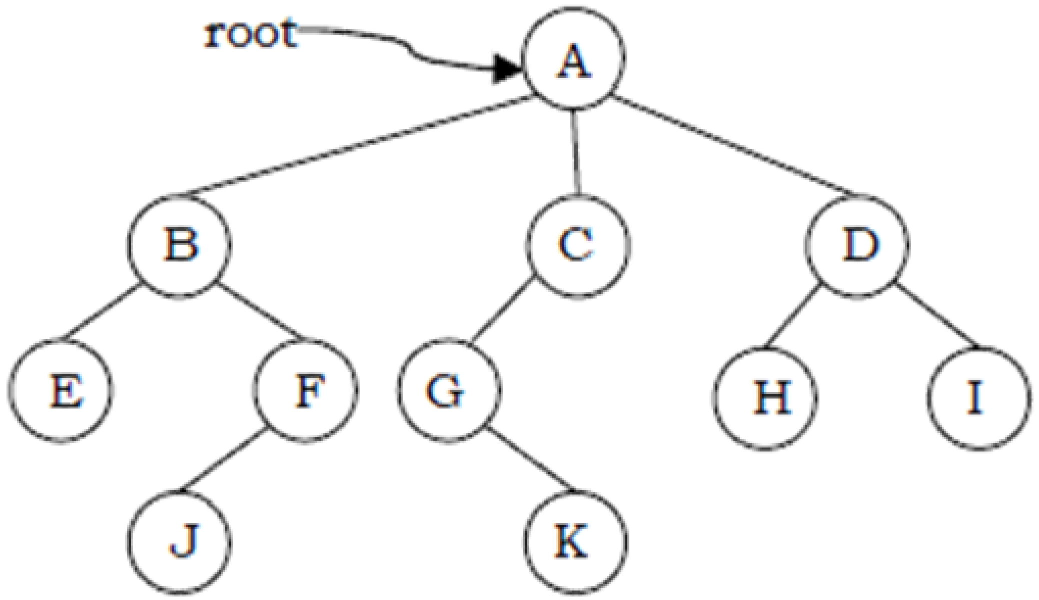
- A node with no children is called *leaf* node Children of same parent are called *siblings*
- Question: can you cite some?

Glossary



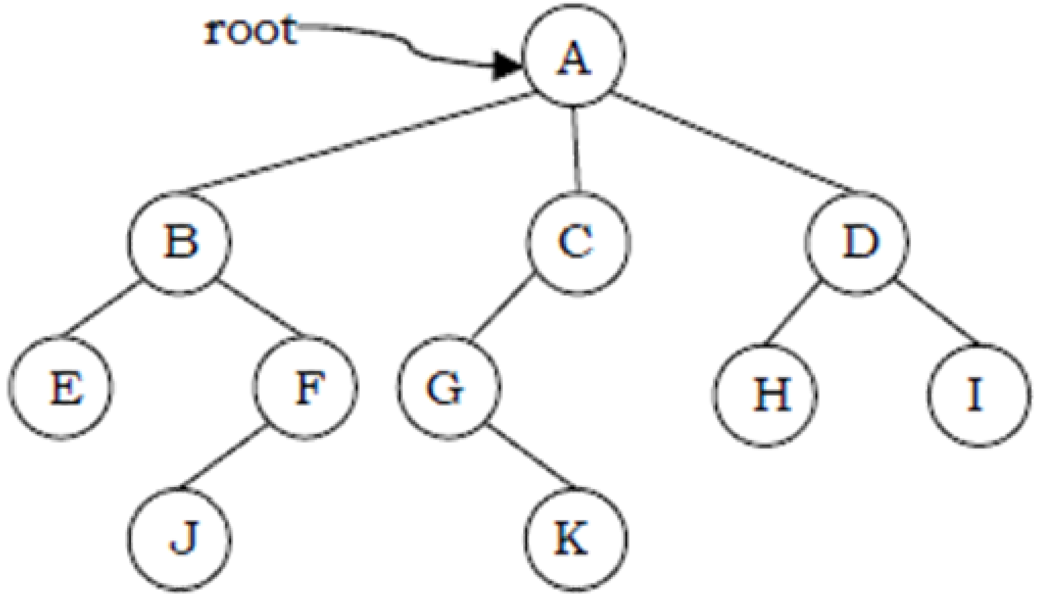
- A node with no children is called $leaf$ node Children of same parent are called $siblings$
- Question: can you cite some?
- (B, C, D are siblings of A, and E, F are the siblings of B).

Glossary



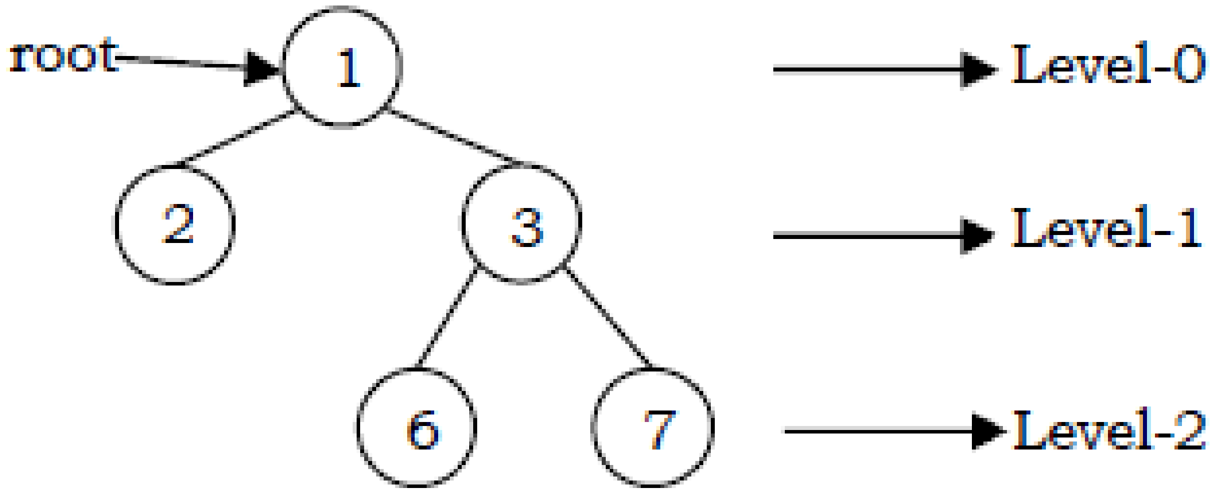
- A node p is an $ancestor$ of node q if there exists a path from $root$ to q and p appears on the path. The node q is called a $descendant$ of p. Question: what are the ancestors of K?

Glossary



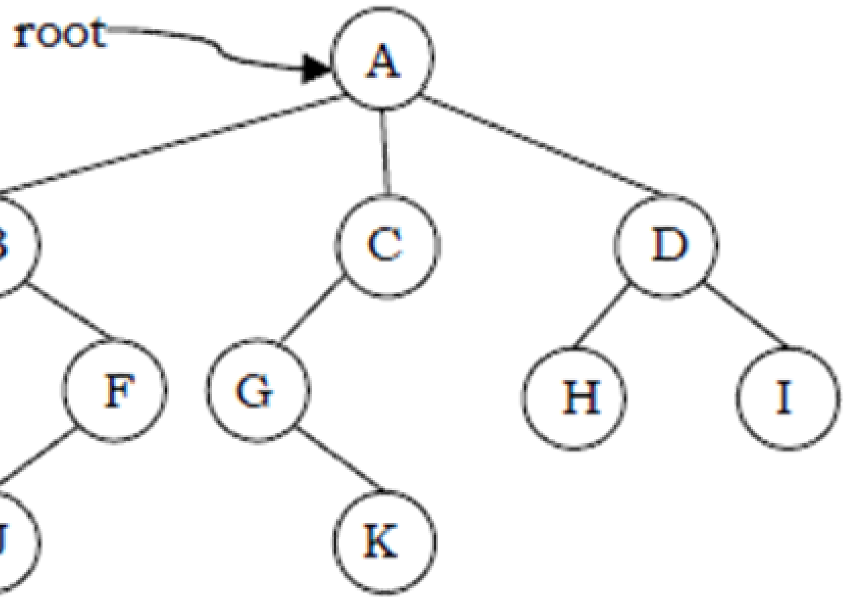
- A node p is an $ancestor$ of node q if there exists a path from $root$ to q and p appears on the path. The node q is called a $descendant$ of p. Question: what are the ancestors of K?
- A, C and G are the ancestors of K.

Glossary



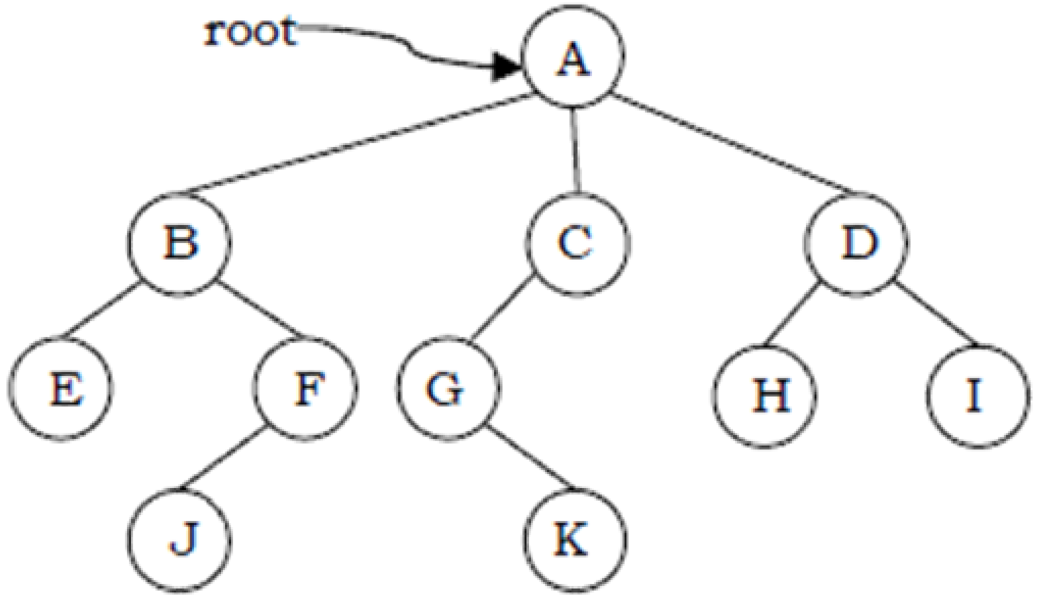
- The set of all nodes at a given depth is called the $level$ of the tree (B, C and D are the same level). The root node is at level zero.
- The $depth$ of a node is the length of the path from the root to the node

Glossary



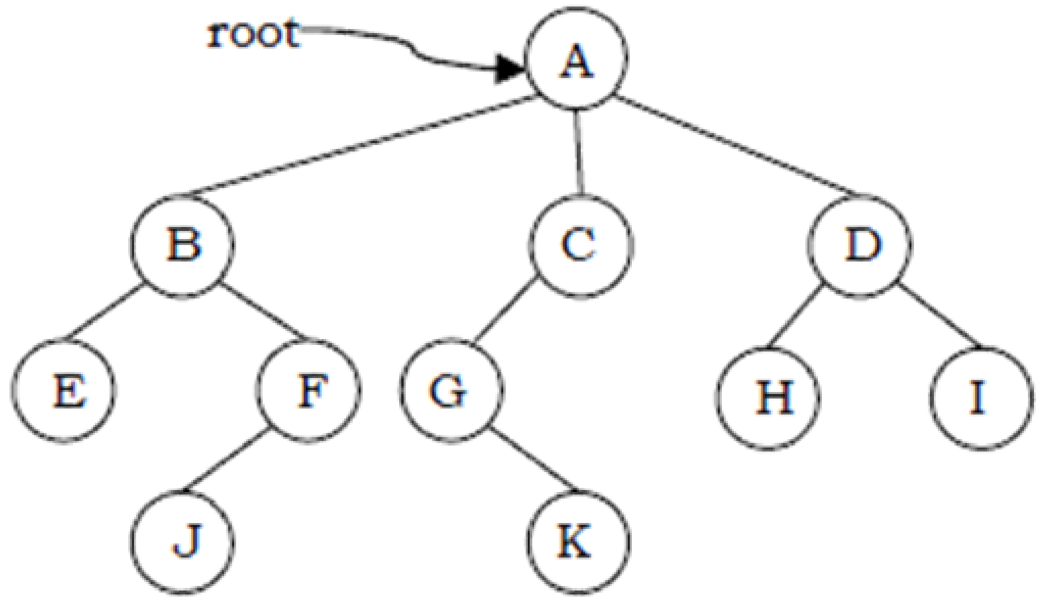
- The set of all nodes at a given depth is called the *level* of the tree (B, C and D are the same level). The root node is at level zero.
- The *depth* of a node is the length of the path from the root to the node
- Question: what is the depth of G?

Glossary



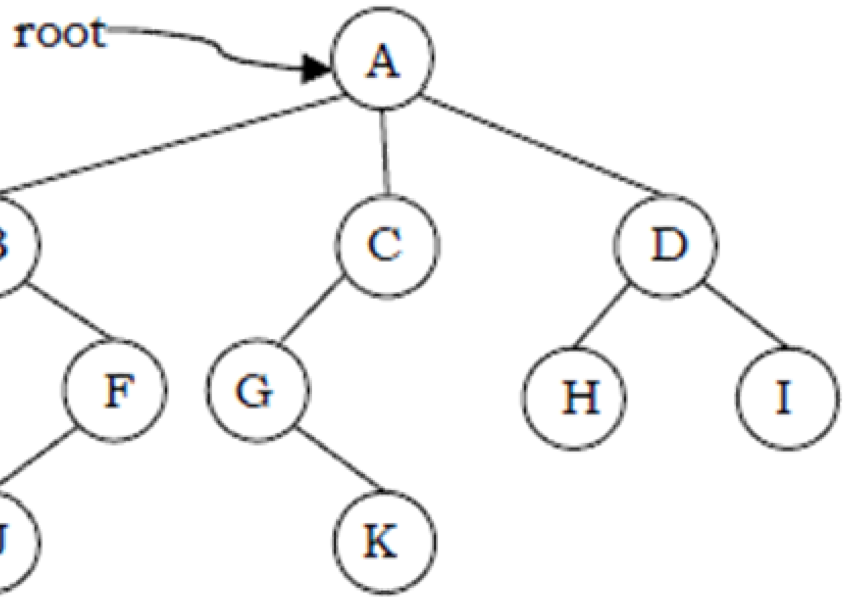
- The set of all nodes at a given depth is called the *level* of the tree (B, C and D are the same level). The root node is at level zero.
- The *depth* of a node is the length of the path from the root to the node
- Question: what is the depth of G?
- The depth of G is 2, $A - C - G$

Glossary



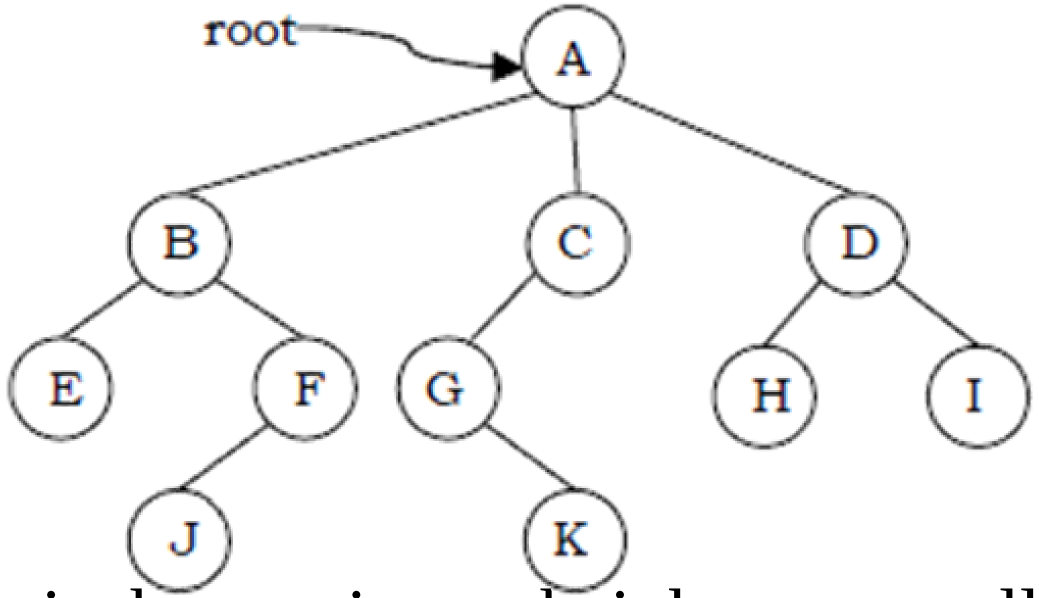
- The *height* of a node is the length of the path from that node to the deepest node. The height of a tree is the length of the path from the root to the deepest node in the tree. A (rooted) tree with only one node (the root) has a height of zero.
- Question: what is the height of *B* ?

Glossary



- The *height* of a node is the length of the path from that node to the deepest node. The height of a tree is the length of the path from the root to the deepest node in the tree. A (rooted) tree with only one node (the root) has a height of zero.
- Question: what is the height of B ?
- The height of B is 2 ($B - F - J$).

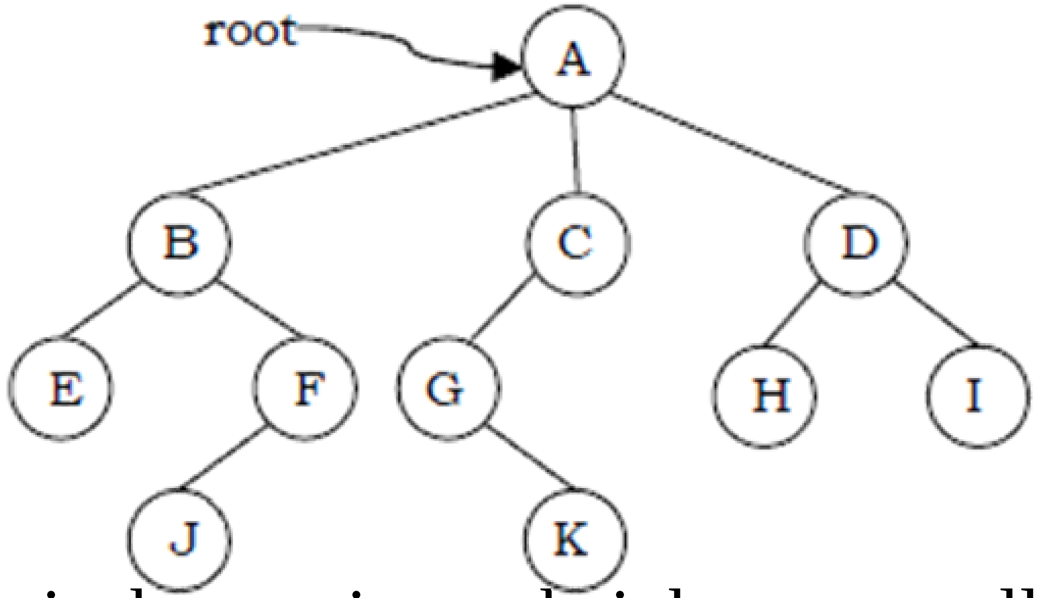
Glossary



- *Height of the tree* is the maximum height among all the nodes in the tree and *depth of the tree* is the maximum depth among all the nodes in the tree.
- For a given tree, depth and height returns the same value. But for individual nodes we may get different results.

- Question: give height and depth of B?

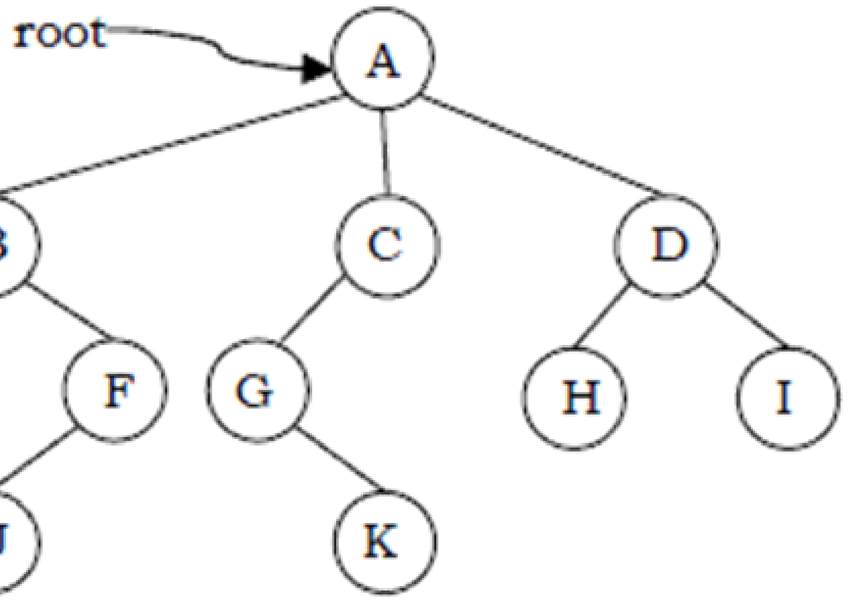
Glossary



- *Height of the tree* is the maximum height among all the nodes in the tree and *depth of the tree* is the maximum depth among all the nodes in the tree.
- For a given tree, depth and height returns the same value. But for individual nodes we may get different results.

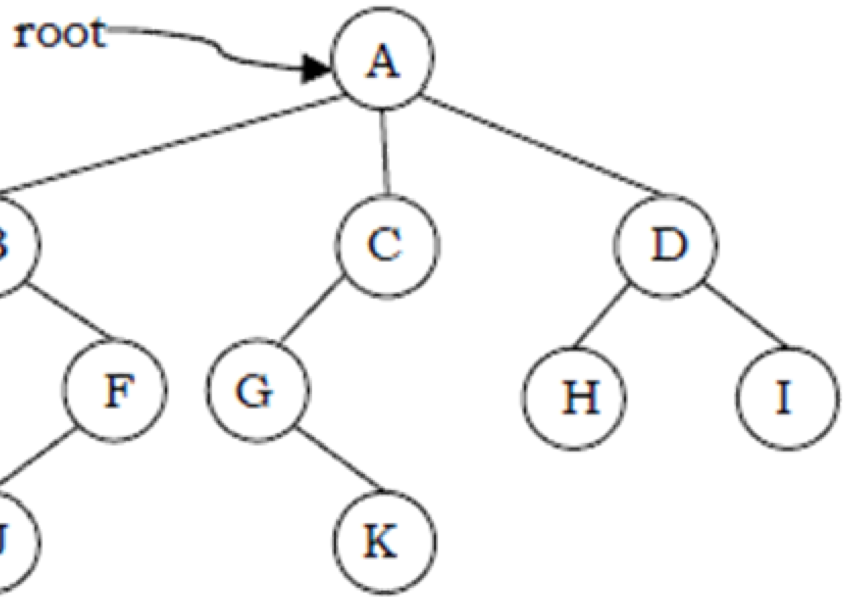
- Question: give height and depth of B? height of B 2, depth 1

Glossary



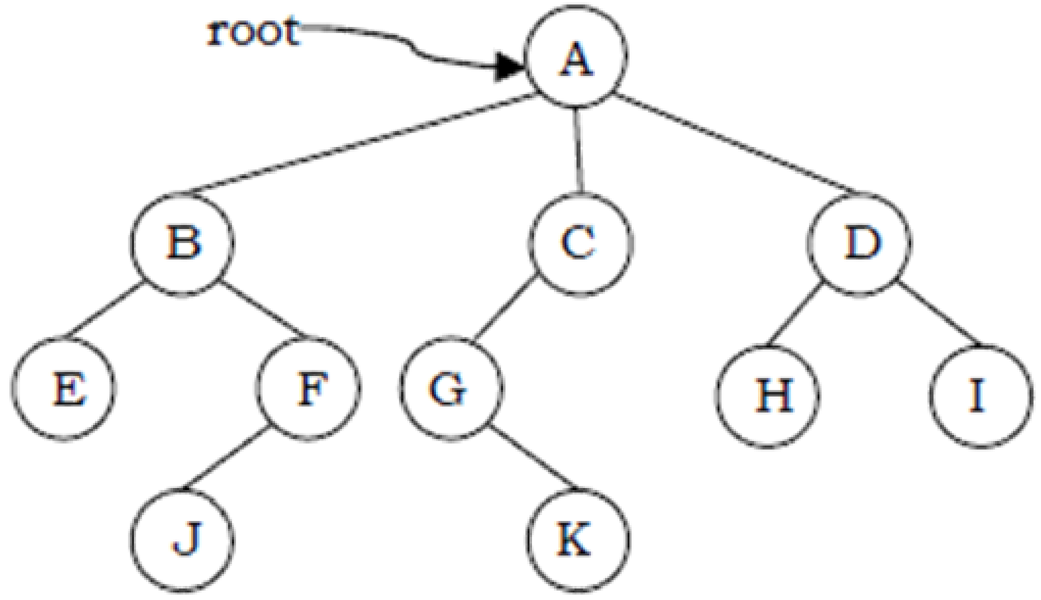
- Question: what is the relationship between height and depth of a node and the depth/height of a tree?

Glossary



- Question: what is the relationship between height and depth of a node and the depth/height of a tree?
- height + depth of a node = height/depth of the tree

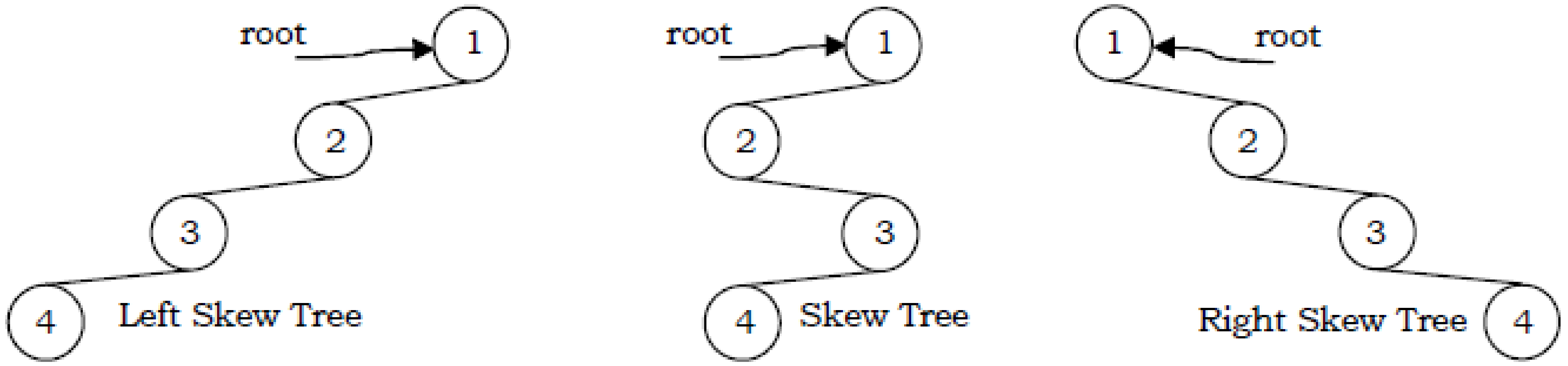
Glossary



- The size of a node is the number of descendants it has including itself (the size of the subtree C is 3).

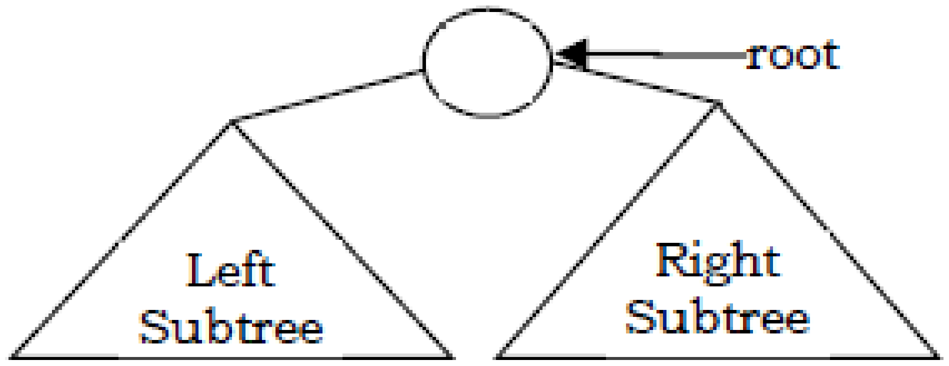
Glossary

- If every node in a tree has only one child (except leaf nodes) then we call such trees *skew trees*. If every node has only left child then we call them *left skew trees*. Similarly, if every node has only right child then we call them *right skew trees*.



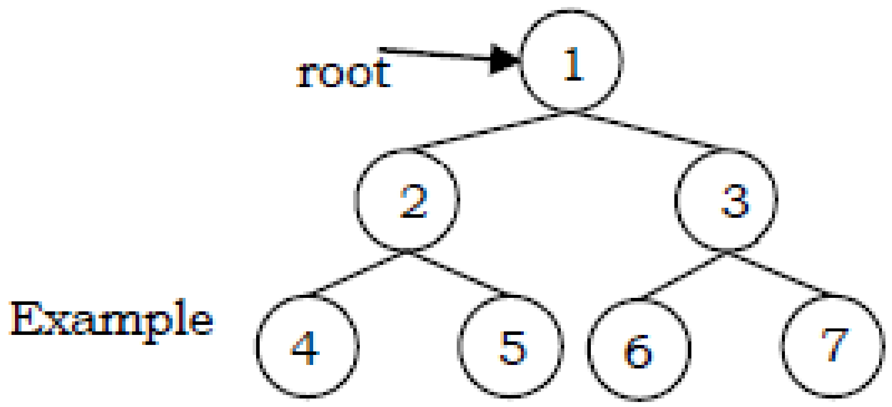
Binary trees

- A tree is called *binary tree* if each node has zero child, one child or two children. Empty tree is also a valid binary tree. We can visualize a binary tree as consisting of a root and two disjoint binary trees, called the left and right subtrees of the root.



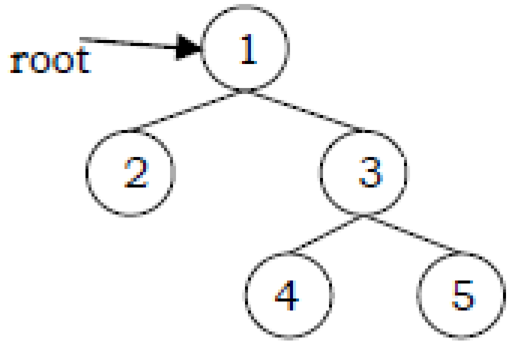
Generic binary trees

- A binary is balanced if it has symmetric right and left subtree



Type of Binary trees

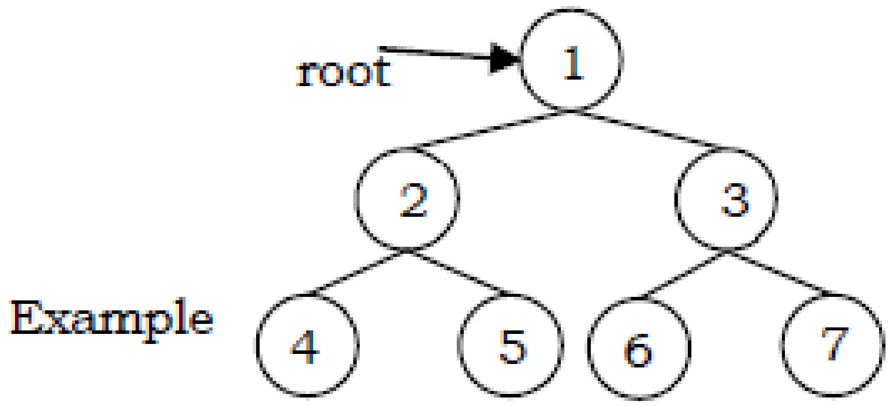
- Strict Binary Tree: A binary tree is called *strict binary tree* if each node has exactly two children or no children.



- Full Binary Tree: A binary tree is called *full binary tree* if each node has exactly two children and all leaf nodes are at the same level.
- Question: can you draw a full binary tree of depth 3?

Full Binary Tree

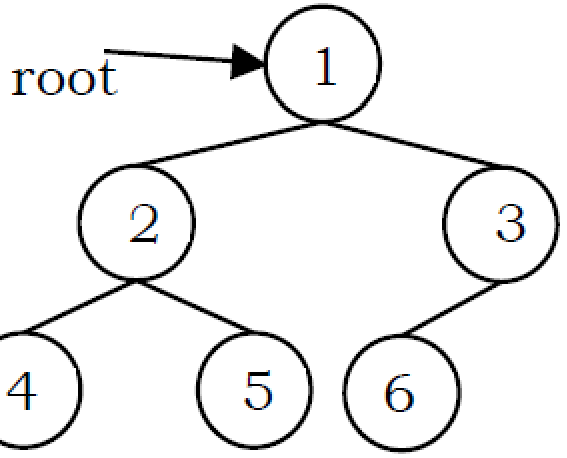
- Question: can you draw a full binary tree of depth 3?
- Yes, here we go!



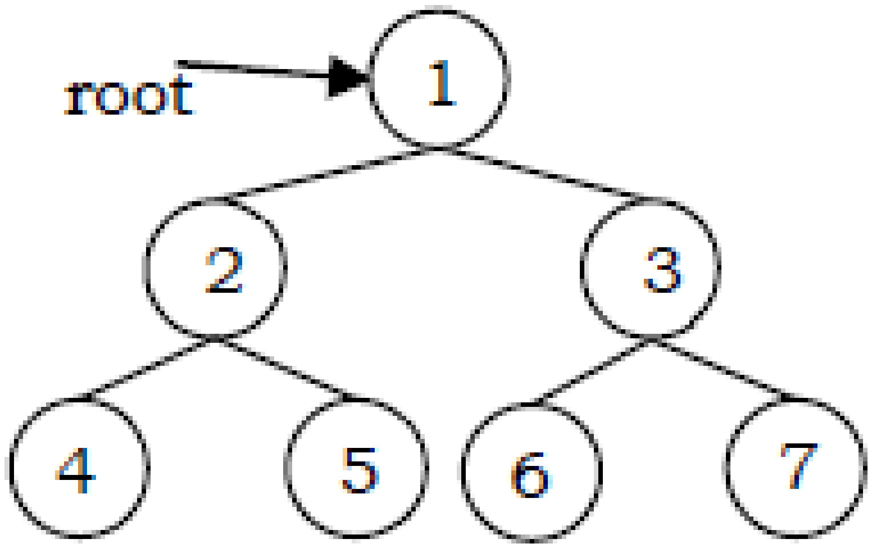
Complete binary tree

- Before defining the $complete\ binary\ tree$, let us assume that the height of the binary tree is h. In complete binary trees, if we give numbering for the nodes by starting at the root (let us say the root node has 1) then we get a complete sequence from 1 to the number of nodes in the tree. While traversing we should give numbering for nil pointers also. A binary tree is called $complete\ binary\ tree$ if all leaf nodes are at height h or $h - 1$ and also without any missing number in the sequence.

Please complete?

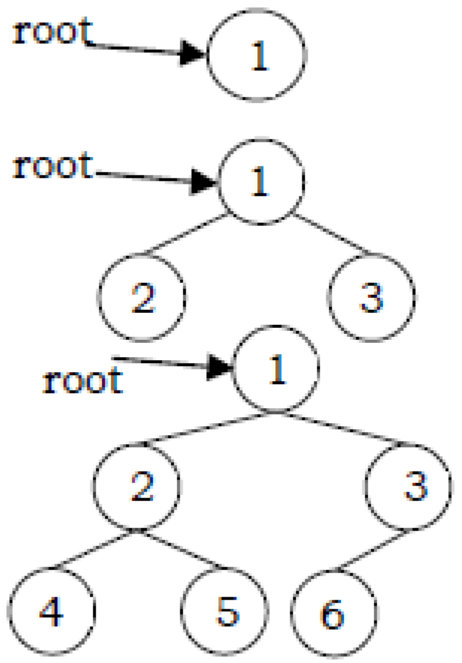


Please complete?



Properties of binary trees

- For the following properties, let us assume that the height of the tree is h. Also, assume that root node is at height zero.



Height	Number of nodes at level h
$h = 0$	$2^0 = 1$
$h = 1$	$2^1 = 2$
$h = 2$	$2^2 = 4$

Properties

- It is easy to get the number of nodes in a full binary tree of depth n
- Question: what is it?

Properties

- It is easy to get the number of nodes in a full binary tree of depth n
- Question: what is it?
- $2^0 + 2^1 + 2^2 + \cdots + 2^n = 2^{n+1} - 1$

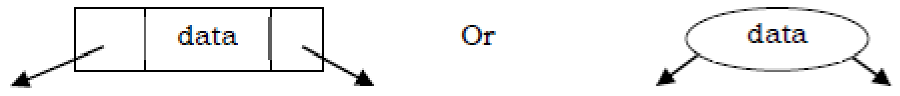
Properties

- From the diagram we can infer the following properties:
  - The number of nodes n in a full binary tree is $2^{h+1} – 1$
  - The number of nodes n in a complete binary tree is between 2^{h} (minimum) and $2^{h+1} – 1$ (maximum).
  - The number of leaf nodes in a full binary tree is 2^{h}
  - The number of None links (wasted pointers) in a complete binary tree of n nodes is $n + 1$.

Structure of Binary Trees

- Now let us define structure of the binary tree. For simplicity, assume that the data of the nodes are integers. One way to represent a node (which contains data) is to have two links which point to left and right children along with data fields as shown below:

Structure



```python
'''Binary Tree Class and its methods'''
class BinaryTree:
    def __init__(self, data):
        self.data = data  # root node
        self.left = None  # left child
        self.right = None  # right child
    # set data
    def setData(self, data):
        self.data = data
    # get data
    def getData(self):
        return self.data
    # get left child of a node
    def getLeft(self):
        return self.left
    # get right child of a node
    def getRight(self):
        return self.right
    # get left child of a node
    def setLeft(self, left):
        self.left = left
    # get right child of a node
    def setRight(self, right):
        self.right = right
```

- Note: In trees, the default flow is from parent to children and it is not mandatory to show directed branches. For our discussion, we assume both the representations shown below are the same.

Operations on Binary Trees

- Basic Operations
  - Inserting an element into a tree
  - Deleting an element from a tree
  - Searching for an element
  - Traversing the tree
- Auxiliary Operations
  - Finding the size of the tree, node
  - Finding the height of a tree, node
  - Finding the level which has maximum sum
  - Finding the least common ancestor (LCA) for a given pair of nodes, and many more.

Applications of Binary Trees

- Following are the some of the applications where *binary trees* play an important role:
  - Expression trees are used in compilers.
  - XML, json, etc.. Parsing
  - Huffman coding trees that are used in data compression algorithms.
  - Binary Search Tree (BST), which supports search, insertion and deletion on a collection of items in O($logn$) (average).
  - Priority Queue (PQ), which supports search and deletion of minimum (or maximum) on a collection of items in logarithmic time (in worst case).

  - In machine learning, GBDT

Binary Tree Traversals

- In order to process trees, we need a mechanism for traversing them, and that forms the subject of this section. The process of visiting all nodes of a tree is called *tree traversal*.
- Each node is processed only once but it may be visited more than once.
- As we have already seen in linear data structures (like linked lists, stacks, queues, etc.), the elements are visited in sequential order.

- But, in tree structures there are many different ways.
- Question: can you imagine some traversal possibilities?

Binary Tree Traversals

- Tree traversal is like searching the tree, except that in traversal the goal is to move through the tree in a particular order.
- In addition, all nodes are processed in the $traversal\ but\ searching$ stops when the required node is found.

Traversal Possibilities

- Starting at the root of a binary tree, there are three main steps that can be performed and the order in which they are performed defines the traversal type. These steps are: performing an action on the current node (referred to as "visiting" the node and denoted with “*D*”), traversing to the left child node (denoted with “*L*”), and traversing to the right child node (denoted with “*R*”). This process can be easily described through recursion. Based on the above definition there are 6 possibilities:

Traversal Possibilities

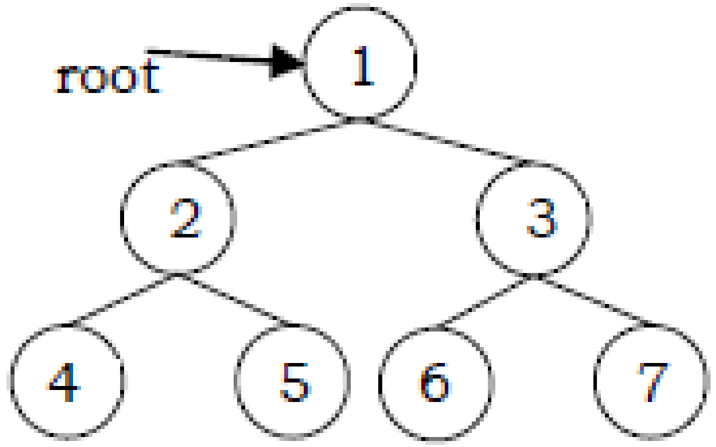
1. LDR: Process left subtree, process the current node data and then process right subtree
2. LRD: Process left subtree, process right subtree and then process the current node data
3. DLR: Process the current node data, process left subtree and then process right subtree
4. DRL: Process the current node data, process right subtree and then process left subtree
5. RDL: Process right subtree, process the current node data and then process left subtree
6. RLD: Process right subtree, process left subtree and then process the current node data

Classifying the Traversals

- The sequence in which these entities (nodes) are processed defines a particular traversal method. The classification is based on the order in which current node is processed. That means, if we are classifying based on current node (D) and if D comes in the middle then it does not matter whether L is on left side of D or R is on left side of D.
- Similarly, it does not matter whether L is on right side of D or R is on right side of D. Due to this, the total 6 possibilities are reduced to 3 and these are:
  - PreOrder (DLR) Traversal
  - InOrder (LDR) Traversal
  - PostOrder (LRD) Traversal

Classifying the Traversals

- There is another traversal method which does not depend on the above orders and it is:
  - Level Order Traversal: This method is inspired from Breadth First Traversal (BFS of Graph algorithms).
- Let us use the diagram below for the remaining discussion.



PreOrder Traversal

- In preorder traversal, each node is processed before (pre) either of its subtrees. This is the simplest traversal to understand. However, even though each node is processed before the subtrees, it still requires that some information must be maintained while moving down the tree. In the example above, 1 is processed first, then the left subtree, and this is followed by the right subtree.

- Therefore, processing must return to the right subtree after finishing the processing of the left subtree. To move to the right subtree after processing the left subtree, we must maintain the root information.

- Question: what is the obvious ADT for such an information?

PreOrder Traversal

- The obvious ADT for such information is a stack. Because of its LIFO structure, it is possible to get the information about the right subtrees back in the reverse order.   Preorder traversal is defined as follows:
  - Visit the root.
  - Traverse the left subtree in Preorder.
  - Traverse the right subtree in Preorder.

- The nodes of tree would be visited in the order: 1 2 4 5 3 6 7

PreOrder Traversal

```
# Pre-order recursive traversal. The nodes' values are appended to the result list
in traversal order
def preorderRecursive(root, result):
    if not root:
        return

    result.append(root.data)
    preorderRecursive(root.left, result)
    preorderRecursive(root.right, result)
```

- Question: What is the time and space complexity?

PreOrder Traversal

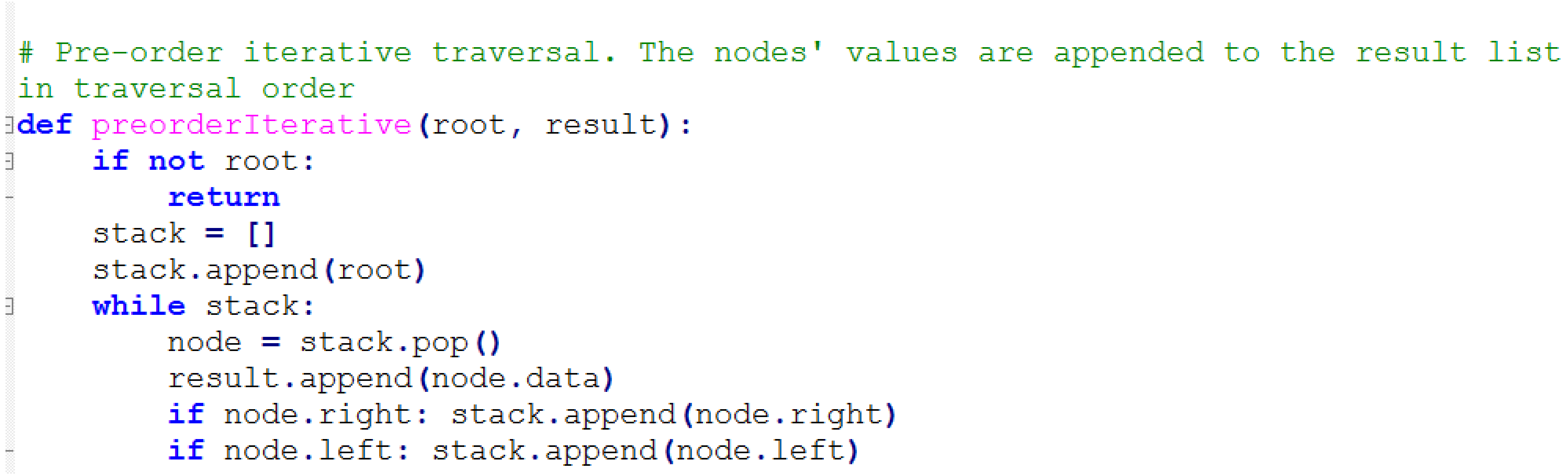
```
# Pre-order recursive traversal. The nodes' values are appended to the result list
in traversal order
def preorderRecursive(root, result):
    if not root:
        return

    result.append(root.data)
    preorderRecursive(root.left, result)
    preorderRecursive(root.right, result)
```

- Question: What is the time and space complexity?
- Time complexity O(n) and space complexity O(n)

Non-Recursive PreOrder Traversal

- In the recursive version, a stack is required as we need to remember the current node so that after completing the left subtree we can go to the right subtree. To simulate the same, first we process the current node and before going to the left subtree, we store the current node on stack. After completing the left subtree processing, *pop* the element and go to its right subtree. Continue this process until stack is nonempty.

Non-Recursive PreOrder Traversal



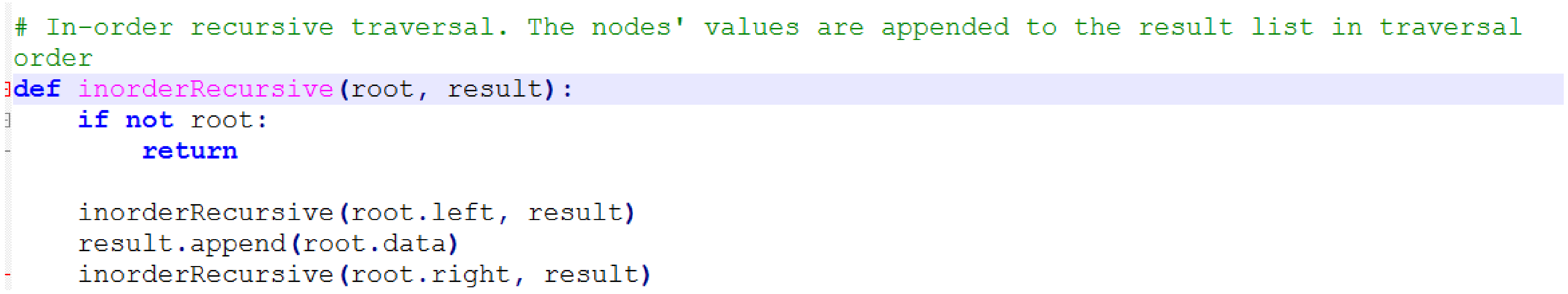
```
# Pre-order iterative traversal. The nodes' values are appended to the result list
in traversal order
def preorderIterative(root, result):
    if not root:
        return
    stack = []
    stack.append(root)
    while stack:
        node = stack.pop()
        result.append(node.data)
        if node.right: stack.append(node.right)
        if node.left: stack.append(node.left)
```

In order traversal

- In Inorder Traversal the root is visited between the subtrees. Inorder traversal is defined as follows:
  - Traverse the left subtree in Inorder.
  - Visit the root.
  - Traverse the right subtree in Inorder.

- The nodes of tree would be visited in the order: 4 2 5 1 6 3 7

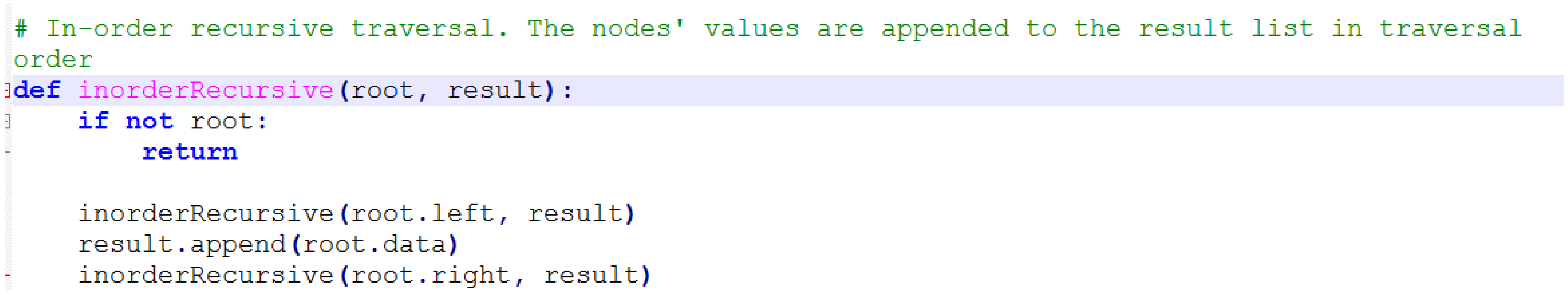
Code



```python
# In-order recursive traversal. The nodes' values are appended to the result list in traversal
order
def inorderRecursive(root, result):
    if not root:
        return

    inorderRecursive(root.left, result)
    result.append(root.data)
    inorderRecursive(root.right, result)
```

- Question: What is the time and space complexity?

Code

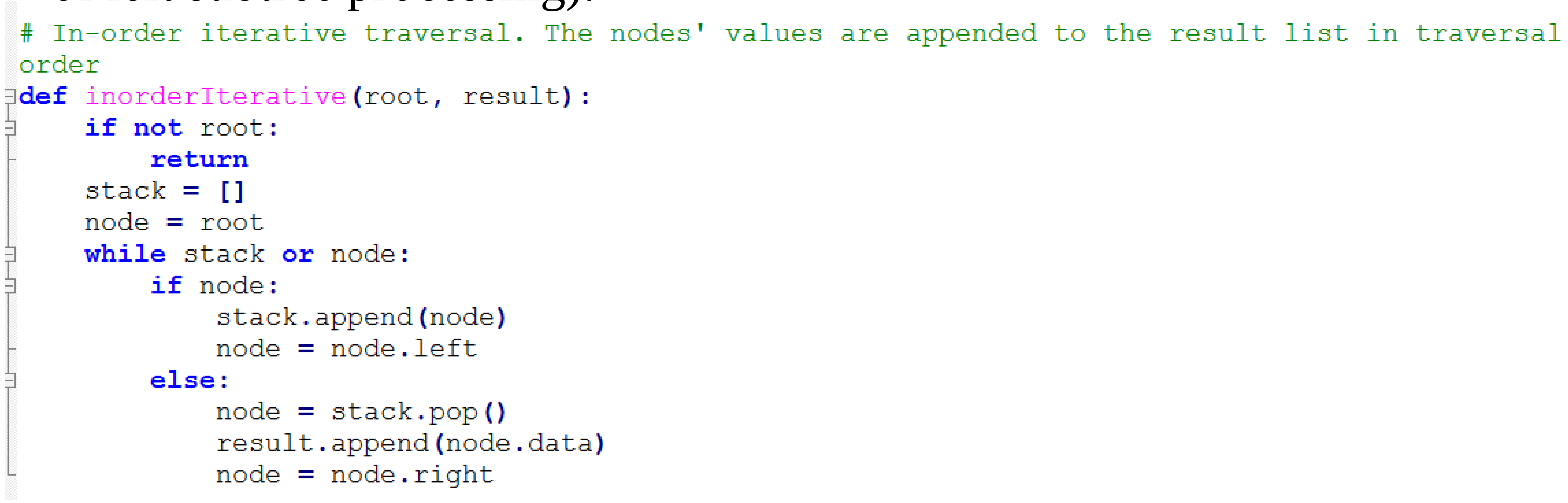


```
# In-order recursive traversal. The nodes' values are appended to the result list in traversal
order
def inorderRecursive(root, result):
    if not root:
        return

    inorderRecursive(root.left, result)
    result.append(root.data)
    inorderRecursive(root.right, result)
```

- Question: What is the time and space complexity?
- Time complexity O(n) and space complexity O(n)

Non-Recursive Inorder Traversal

- The Non-recursive version of Inorder traversal is similar to Preorder. The only change is, instead of processing the node before going to left subtree, process it after popping (which is indicated after completion of left subtree processing).



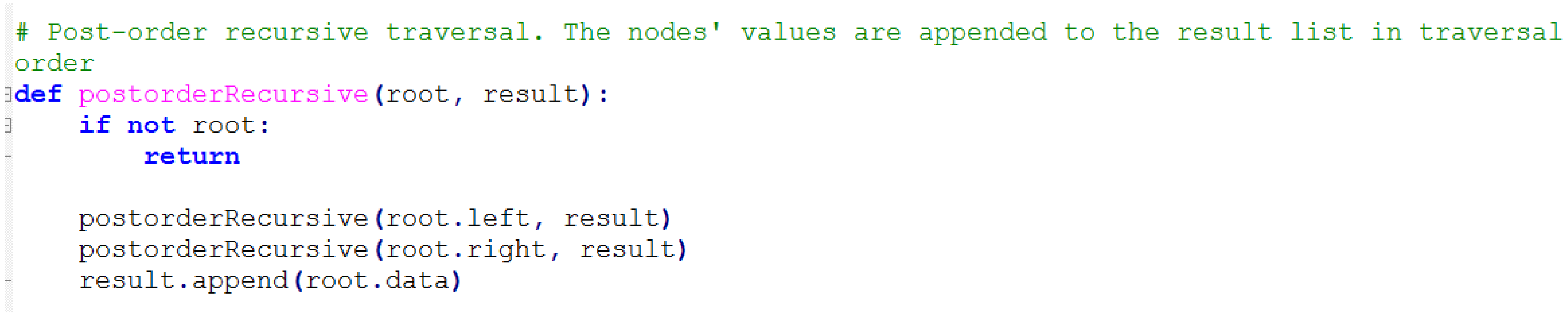
```
# In-order iterative traversal. The nodes' values are appended to the result list in traversal
order
def inorderIterative(root, result):
    if not root:
        return
    stack = []
    node = root
    while stack or node:
        if node:
            stack.append(node)
            node = node.left
        else:
            node = stack.pop()
            result.append(node.data)
            node = node.right
```

PostOrder Traversal

- In post order traversal, the root is visited after both subtrees. PostOrder traversal is defined as follows:
  - Traverse the left subtree in PostOrder.
  - Traverse the right subtree in PostOrder.
  - Visit the root.

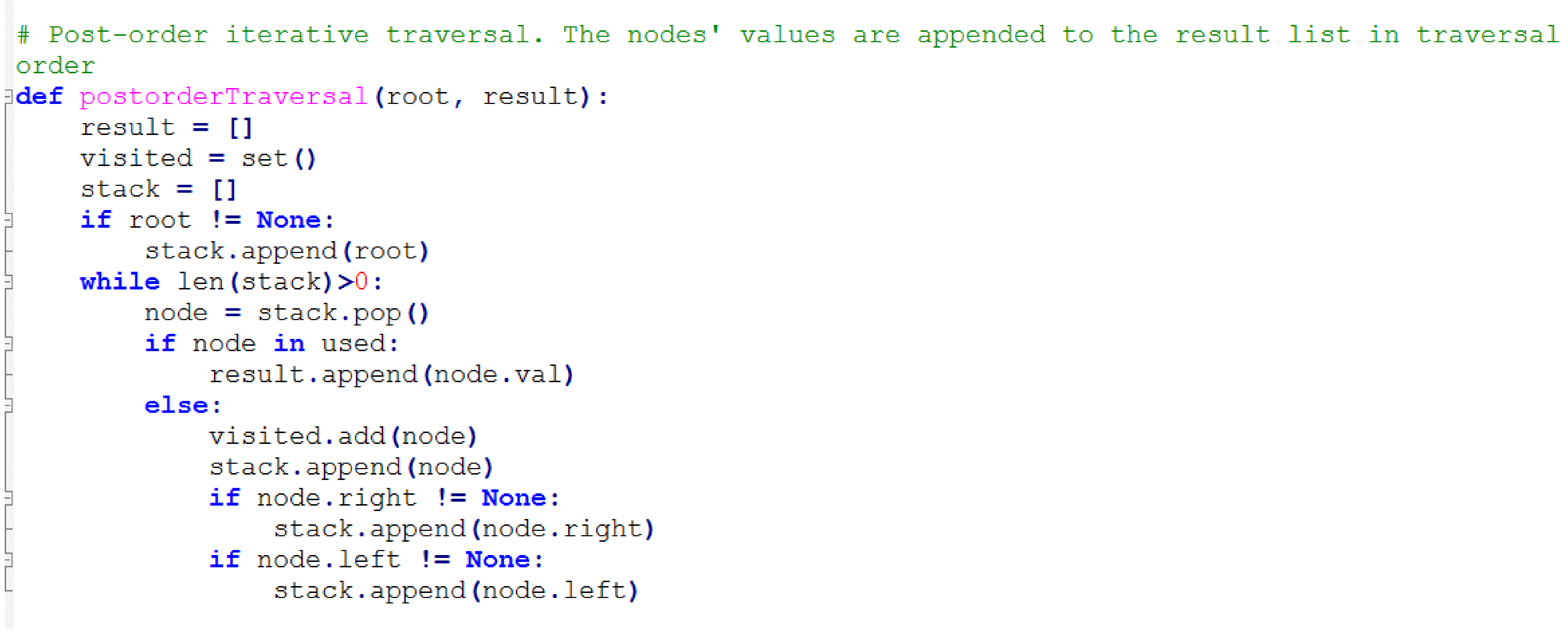
- The nodes of the tree would be visited in the order: 4 5 2 6 7 3 1

Code recursive



```
# Post-order recursive traversal. The nodes' values are appended to the result list in traversal
order
def postorderRecursive(root, result):
    if not root:
        return

    postorderRecursive(root.left, result)
    postorderRecursive(root.right, result)
    result.append(root.data)
```

Code iterative



```
# Post-order iterative traversal. The nodes' values are appended to the result list in traversal
order
def postorderTraversal(root, result):
    result = []
    visited = set()
    stack = []
    if root != None:
        stack.append(root)
    while len(stack)>0:
        node = stack.pop()
        if node in used:
            result.append(node.val)
        else:
            visited.add(node)
            stack.append(node)
            if node.right != None:
                stack.append(node.right)
            if node.left != None:
                stack.append(node.left)
```

Analysis

- In preorder and inorder traversals, after popping the stack element we do not need to visit the same vertex again. But in post order traversal, each node is visited twice. That means, after processing the left subtree we will visit the current node and after processing the right subtree we will visit the same current node. But we should be processing the node during the second visit. Here the problem is how to differentiate whether we are returning from the left subtree or the right subtree.

Analysis

- We use a $previous$ variable to keep track of the earlier traversed node. Let’s assume $current$ is the current node that is on top of the stack. When p $revious$ is $current's$ parent, we are traversing down the tree. In this case, we try to traverse to $current's$ left child if available (i.e., push left child to the stack). If it is not available, we look at $current's$ right child. If both left and right child do not exist (ie, $current$ is a leaf node), we print $current's$ value and pop it off the stack.

- If prev is $current's$ left child, we are traversing up the tree from the left. We look at $current's$ right child. If it is available, then traverse down the right child (i.e., push right child to the stack); otherwise print $current's$ value and pop it off the stack. If $previous$ is $current's$ right child, we are traversing up the tree from the right. In this case, we print $current's$ value and pop it off the stack.

Level Order Traversal

- Level order traversal is defined as follows:
  - Visit the root.
  - While traversing level l, keep all the elements at level $l + 1$ in queue.
  - Go to the next level and visit all the nodes at that level.
  - Repeat this until all levels are completed.

- The nodes of the tree are visited in the order: [1] [2 3] [4 5 6 7]

- Time Complexity: O(n). Space Complexity: O(n). In the worst case, all the nodes on the entire last level could be in the queue.

In Lab session

- You will play with the concepts and starts getting more and more familiar with trees
- This can be useful for your project

- Lab is done by Remy Belmonte