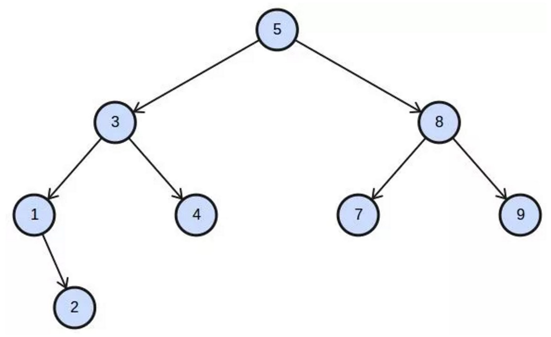
June12

11 June 2020 23:11

Trees

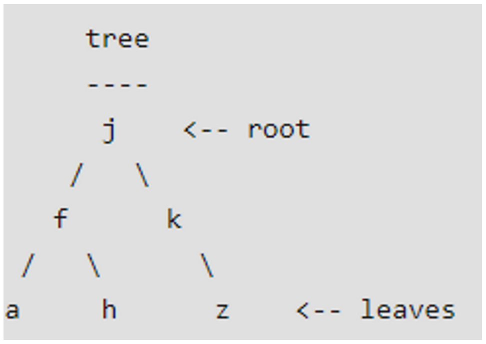
A **tree** is a data structure that is modelled after nature. Unlike trees in nature, the tree data structure is upside down: the root of the tree is on top. A tree consists of nodes and its connections are called edges. The bottom nodes are also named leaf nodes. A tree may not have a cycle.



A tree with eight nodes. The root of the tree (5) is on top.

- ❖ **Unlike Arrays, Linked Lists, Stack and queues, which are linear data structures, trees are hierarchical data structures.**

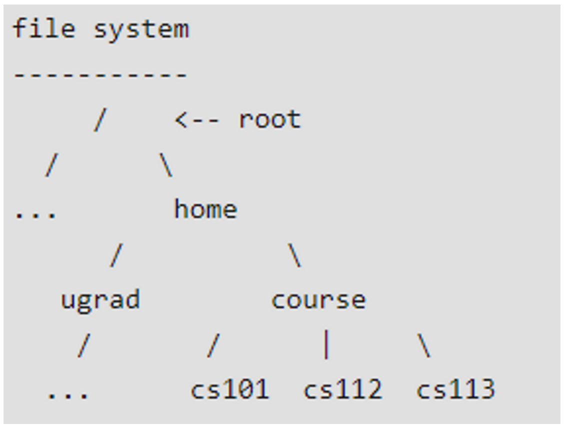
Tree Vocabulary: The topmost node is called root of the tree. The elements that are directly under an element are called its children. The element directly above something is called its parent. For example, ‘a’ is a child of ‘f’, and ‘f’ is the parent of ‘a’. Finally, elements with no children are called leaves.



```
     tree
     ----
      j    <-- root
    /   \
   f      k
 /   \      \
a     h      z    <-- leaves
```

Why Trees?

1. One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:



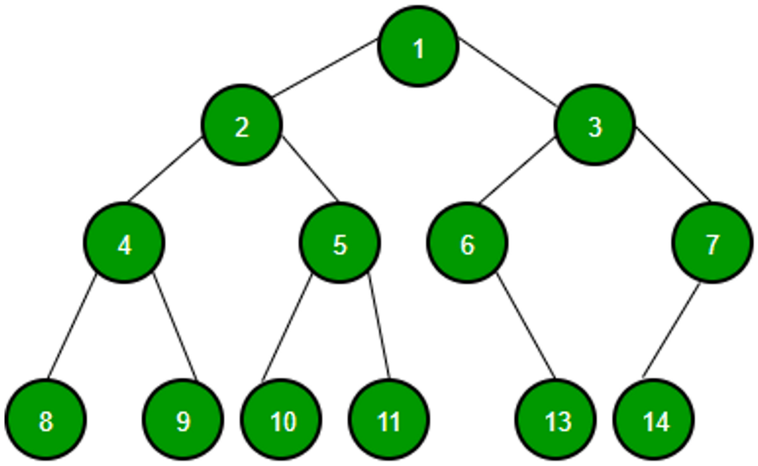
```
file system
-----------
     /    <-- root
  /      \
...       home
      /          \
   ugrad        course
    /       /      |     \
  ...      cs101  cs112  cs113
```

2. Trees (with some ordering e.g., BST) provide moderate access/search (quicker than Linked List and slower than arrays).
3. Trees provide moderate insertion/deletion (quicker than Arrays and slower than Unordered Linked Lists).
4. Like Linked Lists and unlike Arrays, Trees don't have an upper limit on number of nodes as nodes are linked using pointers.

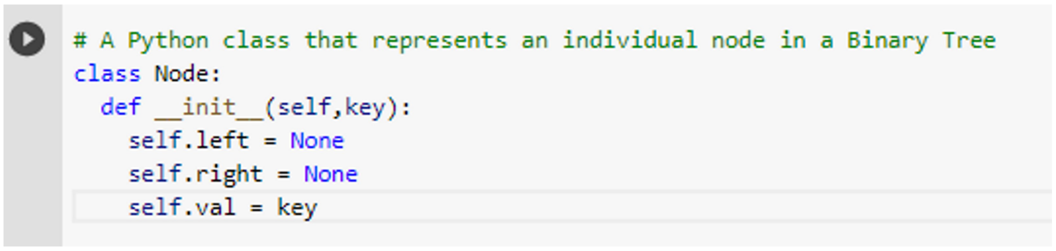
Python does not have built-in support for trees.

Binary Tree Data Structure

A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child. A binary tree is a data structure where every node has at most two children (left and right child). The root of a tree is on top. Every node below has a node above known as the parent node.

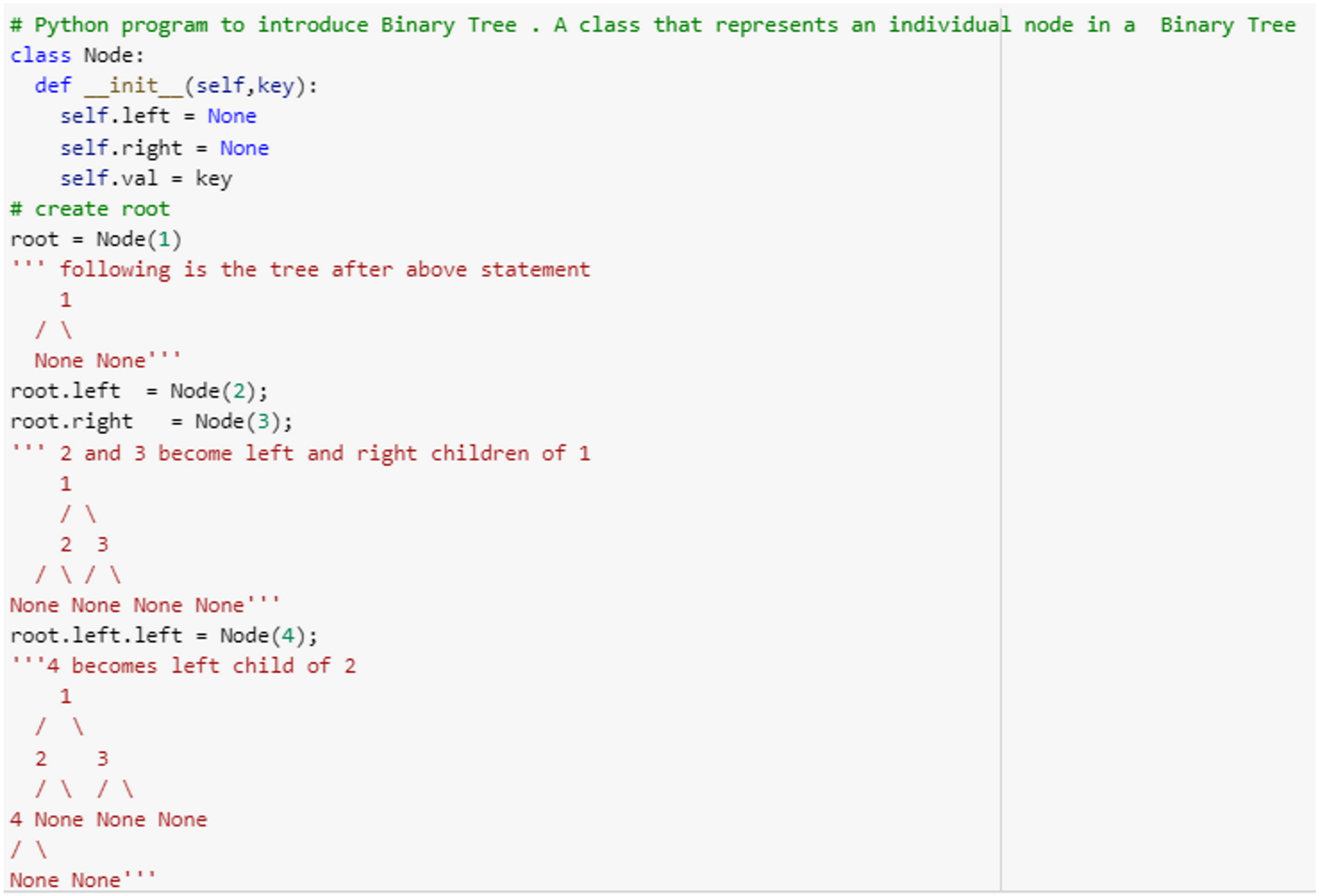


A Binary Tree node contains following parts.
1. Data
2. Pointer to left child
3. Pointer to right child



```python
# A Python class that represents an individual node in a Binary Tree
class Node:
  def __init__(self,key):
    self.left = None
    self.right = None
    self.val = key
```

Let us create a simple tree with 4 nodes



```python
# Python program to introduce Binary Tree . A class that represents an individual node in a  Binary Tree
class Node:
  def __init__(self,key):
    self.left = None
    self.right = None
    self.val = key
# create root
root = Node(1)
''' following is the tree after above statement
    1
  / \
  None None'''
root.left  = Node(2);
root.right   = Node(3);
''' 2 and 3 become left and right children of 1
    1
    / \
    2  3
  / \ / \
None None None None'''
root.left.left = Node(4);
'''4 becomes left child of 2
    1
  /  \
  2    3
  / \  / \
4 None None None
/ \
None None'''
```

Binary Tree Properties

- The maximum number of nodes at level 'l' will be 2^{l-1} . Here level is the number of nodes on path from root to the node, including the root itself. We are considering the level of root is 1.
- Maximum number of nodes present in binary tree of height h is $2^h - 1$. Here height is the max number of nodes on root to leaf path. Here we are considering height of a tree with one node is 1.
- In a binary tree with n nodes, minimum possible height or minimum number of levels are $\log_2(n+1)$. If we consider that the height of leaf node is considered as 0, then the formula will be $\log_2(n+1) - 1$
- A binary tree with 'L' leaves has at least $\log_2 L + 1$ number of levels
- If a binary tree has 0 or 2 children, then number of leaf nodes are always one more than nodes with two children.

Types of Binary Tree

1. **Full Binary Tree:** A Binary Tree is a full binary tree if every node has 0 or 2 children. The following are the examples of a full binary tree. We can also say a full binary tree is a binary tree in which all nodes except leaf nodes have two children.

   In a Full Binary Tree, number of leaf nodes is the number of internal nodes plus 1
   $L = I + 1$
   Where L = Number of leaf nodes, I = Number of internal nodes

```
             18
         /       \
       15         30
      /  \        /  \
    40    50    100   40

           18
         /    \
       15      20
      /  \
    40    50
   /   \
  30   50

             18
          /     \
        40       30
                /  \
              100   40
```

2. **Complete Binary Tree:** A Binary Tree is a complete Binary Tree if all the levels are completely filled except possibly the last level and the last level has all keys as left as possible

```
               18
           /       \
         15         30
        /  \        /  \
      40    50    100   40


               18
           /       \
         15         30
        /  \        /  \
      40    50    100   40
     /  \   /
    8   7  9
```

3. **Perfect Binary Tree** A Binary tree is a Perfect Binary Tree in which all the internal nodes have two children and all leaf nodes are at the same level.

```
               18
           /       \
         15         30
        /  \        /  \
      40    50    100   40


               18
           /       \
         15         30
```

A Perfect Binary Tree of height h (where height is the number of nodes on the path from the root to leaf) has 2h – 1 node.

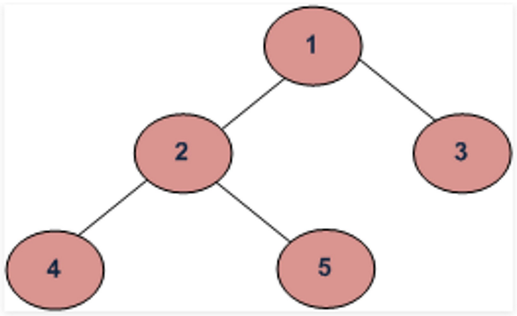
4. **Balanced Binary Tree :** A binary tree is balanced if the height of the tree is O(Log n) where n is the number of nodes. For Example, the AVL tree maintains O(Log n) height by making sure that the difference between the heights of the left and right subtrees is almost 1. Red-Black trees maintain O(Log n) height by making sure that the number of Black nodes on every root to leaf paths is the same and there are no adjacent red nodes. Balanced Binary Search trees are performance-wise good as they provide O(log n) time for search, insert and delete.

5. **A degenerate (or pathological) tree :**A Tree where every internal node has one child. Such trees are performance-wise same as linked list.

```
    10
   /
 20
  \
   30
    \
    40
```

Tree Traversals (In-order, Pre-order and Post-order)

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.



Depth First Traversals:

(a) In-order (Left, Root, Right) : 4 2 5 1 3

```
Algorithm Inorder(tree)
   1. Traverse the left subtree, i.e., call Inorder(left-subtree)
   2. Visit the root.
   3. Traverse the right subtree, i.e., call Inorder(right-subtree)
```

(b) Pre-order (Root, Left, Right) : 1 2 4 5 3

```
Algorithm Preorder(tree)
   1. Visit the root.
   2. Traverse the left subtree, i.e., call Preorder(left-subtree)
   3. Traverse the right subtree, i.e., call Preorder(right-subtree)
```

(c) Post-order (Left, Right, Root) : 4 5 2 3 1

```
Algorithm Postorder(tree)
   1. Traverse the left subtree, i.e., call Postorder(left-subtree)
   2. Traverse the right subtree, i.e., call Postorder(right-subtree)
   3. Visit the root.
```

Count leaf nodes in a binary tree

A node is a leaf node if both left and right child nodes of it are NULL.

```
getLeafCount(node)
1) If node is NULL then return 0.
2) Else If left and right child nodes are NULL return 1.
3) Else recursively calculate leaf count of the tree using below formula.
    Leaf count of a tree = Leaf count of left subtree +
                                 Leaf count of right subtree
```

Find the Maximum Depth or Height of a Tree

```
 maxDepth()
1. If tree is empty then return 0
2. Else
     (a) Get the max depth of left subtree recursively  i.e.,
          call maxDepth( tree->left-subtree)
     (a) Get the max depth of right subtree recursively  i.e.,
          call maxDepth( tree->right-subtree)
     (c) Get the max of max depths of left and right
          subtrees and add 1 to it for the current node.
         max_depth = max(max dept of left subtree,
                             max depth of right subtree)
                             + 1
     (d) Return max_depth
```

Construct a Binary Tree from Post-order and In-order

```
Input :
in[]   = {4, 8, 2, 5, 1, 6, 3, 7}
post[] = {8, 4, 5, 2, 6, 7, 3, 1}

Output : Root of below tree
          1
       /     \
     2         3
   /    \   /    \
  4      5  6     7
    \
      8
```

Let us see the process of constructing tree from in[] = {4, 8, 2, 5, 1, 6, 3, 7} and post[] = {8, 4, 5, 2, 6, 7, 3, 1}
Algorithm
1) We first find the last node in post[]. The last node is "1", we know this value is root as root always appear in the end of post-order traversal.

2) We search "1" in in[] to find left and right subtrees of root. Everything on left of "1" in in[] is in left subtree and everything on right is in right subtree

```
         1
       /    \
[4, 8, 2, 5]   [6, 3, 7]
```

3) We recur the above process for following two.
….b) Recur for in[] = {6, 3, 7} and post[] = {6, 7, 3}
…….Make the created tree as right child of root.
….a) Recur for in[] = {4, 8, 2, 5} and post[] = {8, 4, 5, 2}.
…….Make the created tree as left child of root.

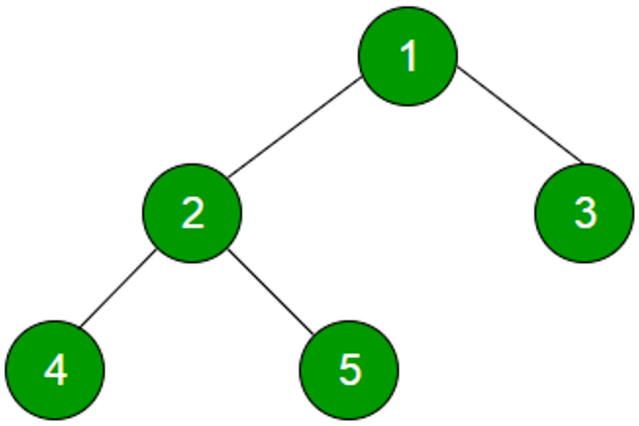
Check whether a binary tree is a full binary tree or not

Algorithm:

1) If a binary tree node is NULL then it is a full binary tree.
2) If a binary tree node does have empty left and right sub-trees, then it is a full binary tree by definition.
3) If a binary tree node has left and right sub-trees, then it is a part of a full binary tree by definition. In this case recursively check if the left and right sub-trees are also binary trees themselves.
4) In all other combinations of right and left sub-trees, the binary tree is not a full binary tree.

Level Order Tree Traversal

Level order traversal of a tree is breadth first traversal for the tree.



Level order traversal of the above tree is 1 2 3 4 5

Method 1 (Use function to print a given level)
Algorithm:
There are basically two functions in this method. One is to print all nodes at a given level (printGivenLevel), and other is to print level order traversal of the tree (printLevelorder). printLevelorder makes use of printGivenLevel to print nodes at all levels one by one starting from root.

```
/*Function to print level order traversal of tree*/
printLevelorder(tree)
for d = 1 to height(tree)
   printGivenLevel(tree, d);

/*Function to print all nodes at a given level*/
printGivenLevel(tree, level)
if tree is NULL then return;
if level is 1, then
    print(tree->data);
else if level greater than 1, then
    printGivenLevel(tree->left, level-1);
    printGivenLevel(tree->right, level-1);
```

Time Complexity: O(n^2) in worst case. For a skewed tree, printGivenLevel() takes O(n) time where n is the number of nodes in the skewed tree. So time complexity of printLevelOrder() is O(n) + O(n-1) + O(n-2) + .. + O(1) which is O(n^2).
Space Complexity: O(n) in worst case. For a skewed tree, printGivenLevel() uses O(n) space for call stack. For a Balanced tree, call stack uses O(log n) space, (i.e., height of the balanced tree).

Method 2 (Using queue)
Algorithm:
For each node, first the node is visited and then it's child nodes are put in a FIFO queue

```
printLevelorder(tree)
1) Create an empty queue q
2) temp_node = root /*start from root*/
3) Loop while temp_node is not NULL
    a) print temp_node->data.
    b) Enqueue temp_node's children (first left then right children) to q
    c) Dequeue a node from q and assign it's value to temp_node
```

Diameter of a Binary Tree
The diameter of a tree (sometimes called the width) is the number of nodes on the longest path between two end nodes. The diagram below shows two trees each with diameter nine, the leaves that form the ends of a longest path are shaded (note that there is more than one path in each tree of length nine, but no path longer than nine nodes).

Vertical Sum in a given Binary Tree

Given a Binary Tree, find the vertical sum of the nodes that are in the same vertical line. Print all sums through different vertical lines.

```
      1
    /   \
  2       3
 / \     / \
4   5   6   7
```

The tree has 5 vertical lines

Vertical-Line-1 has only one node 4 => vertical sum is 4

Vertical-Line-2: has only one node 2=> vertical sum is 2

Vertical-Line-3: has three nodes: 1,5,6 => vertical sum is 1+5+6 = 12

Vertical-Line-4: has only one node 3 => vertical sum is 3

Vertical-Line-5: has only one node 7 => vertical sum is 7

So expected output is 4, 2, 12, 3 and 7

Questions to implement

1. **Count the number of leaf nodes**
2. **Construct a Binary Tree and print the following results:**
   (a) Is it Complete binary search true
   (b) Is it strict binary search tree
   (c) height of the Binary tree
   (d) pre order

(e) post order
(f) in order
(g) level order

3. **Give an algorithm for finding the diameter of the binary tree. The diameter of a tree (sometimes called the width) is the number of nodes on the longest path between two leaves in the tree.**
4. **Give an algorithm for finding the vertical sum of a binary tree. For example,**
   The tree has 5 vertical lines
   Vertical-1: nodes-4 =>vertical sum is 4
   Vertical-2: nodes-2 =>vertical sum is 2
   Vertical -3: nodes- 1,5,6 => vertical sum is 1 + 5 + 6 =12
   Vertical-4: nodes-3 => vertical sum is 3
   Vertical-5: nodes-7 =>vertical sum is 7
   We need to output: 4 2 12 3 7