educative

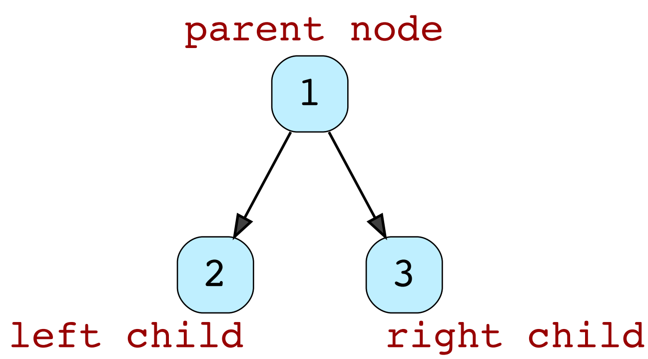
Introduction

In this lesson, you will be introduced to Binary Trees and their implementation in Python.

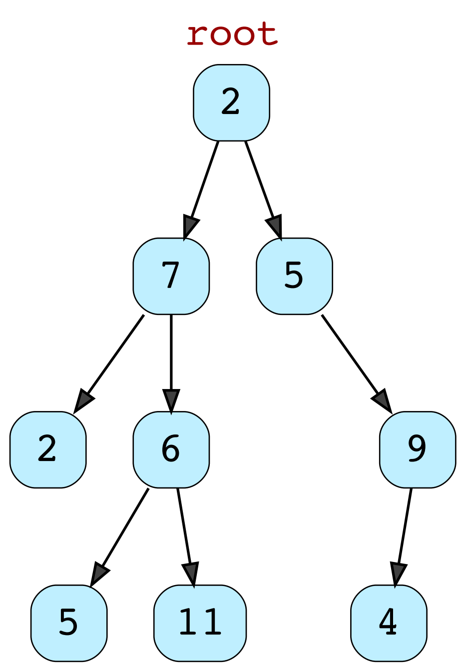
We'll cover the following

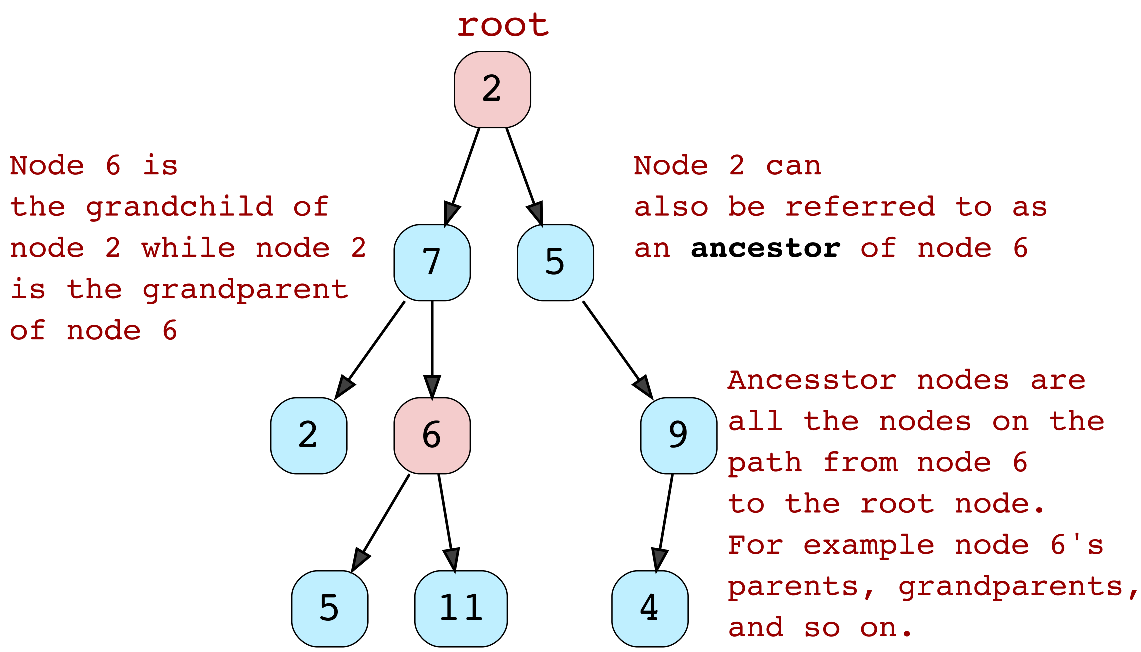
- Depth of a Node
- Height of a Tree
- Types of Binary Trees
  - Complete Binary Tree
  - Full Binary Tree
- Implementation

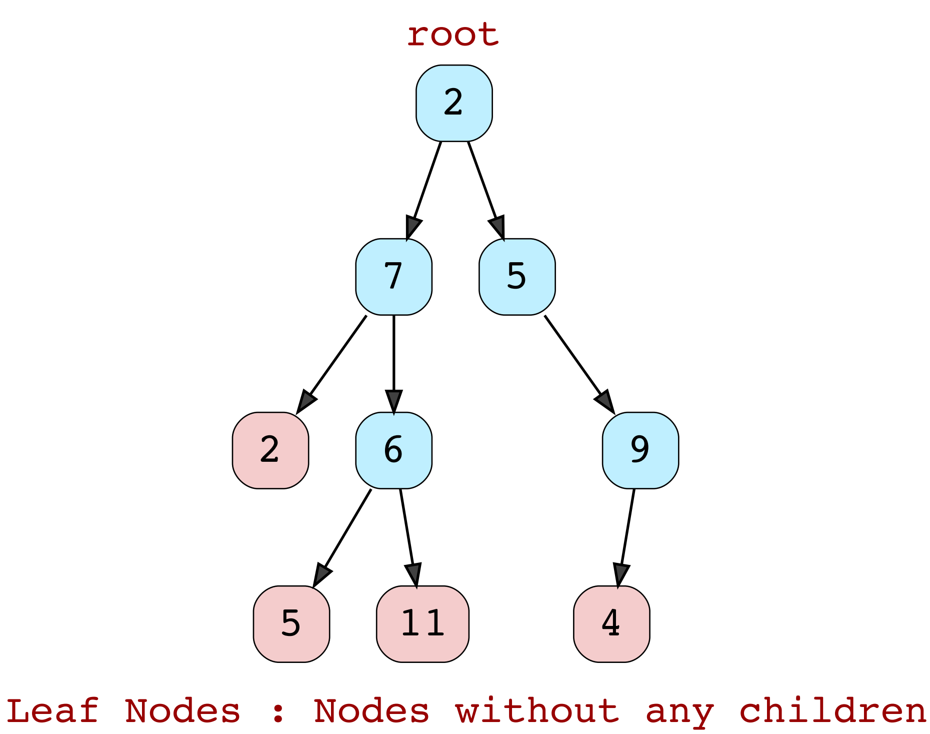
A **binary tree** is a tree data structure in which each node has at most two children, which are referred to as the *left child* and the *right child*. Have a look at an elementary example of a binary tree:



Here is another example of a binary tree that introduces us to other related terminologies:

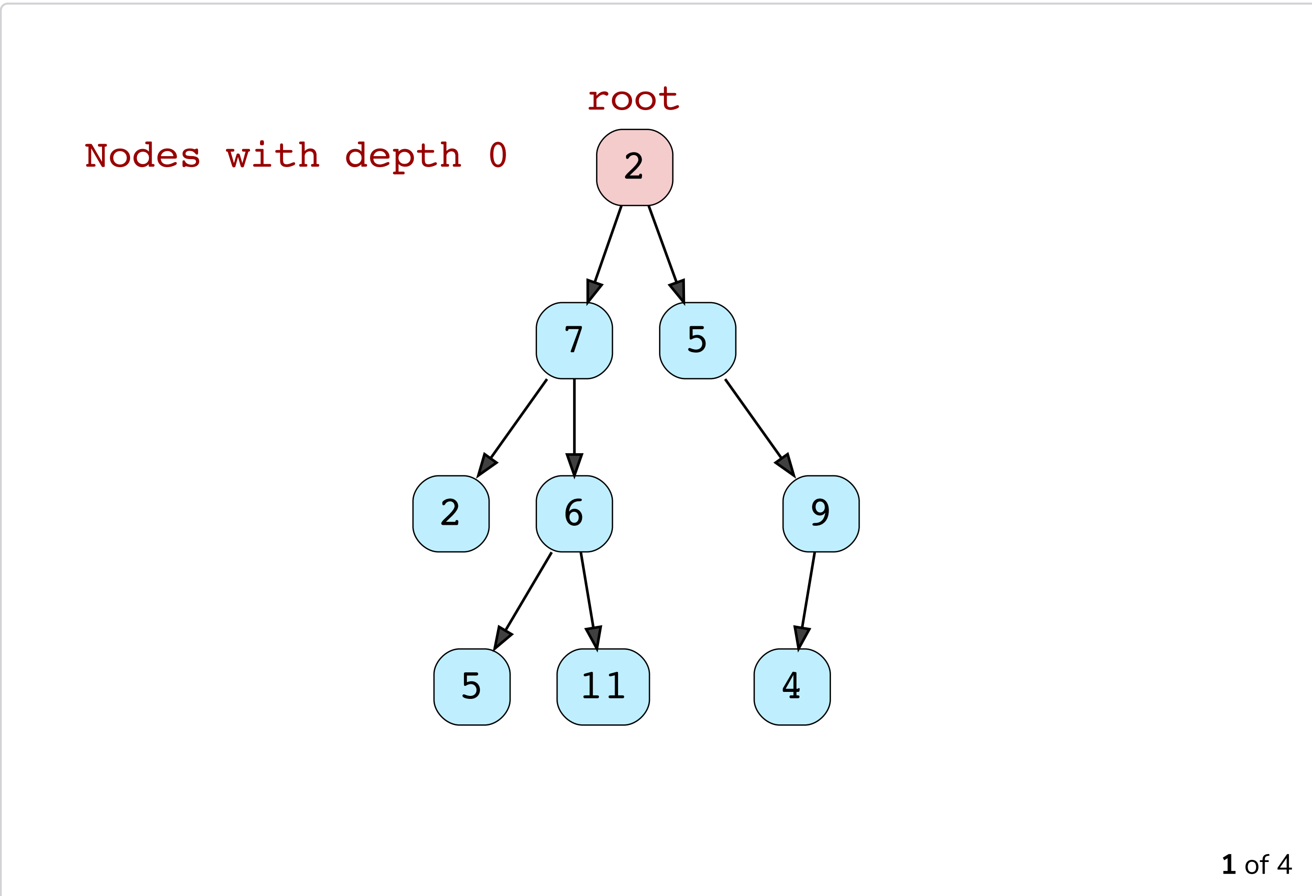


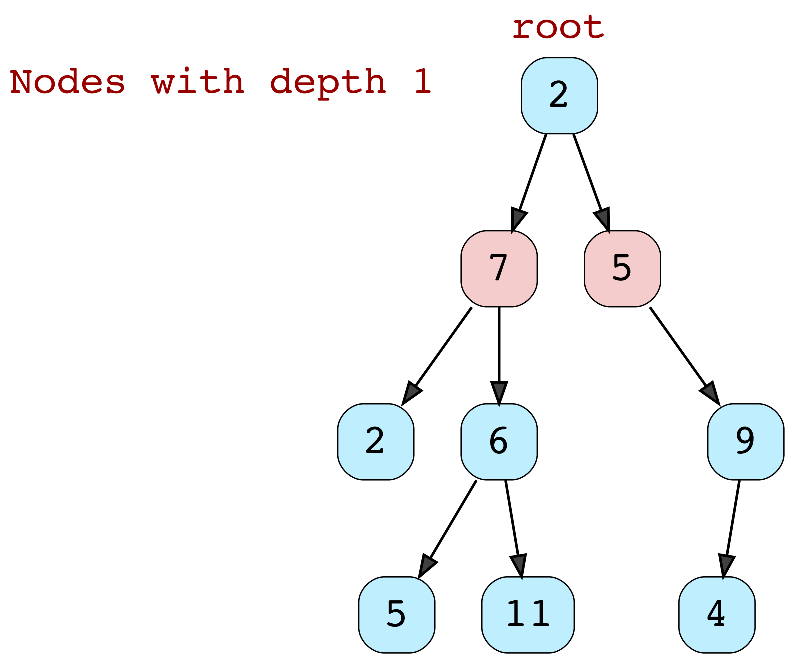

root
2
Node 6 is the grandchild of node 2 while node 2 is the grandparent of node 6
7
5
Node 2 can also be referred to as an **ancestor** of node 6
2
6
9
Ancesstor nodes are all the nodes on the path from node 6 to the root node. For example node 6's parents, grandparents, and so on.
5
11
4


root
2
7
5
2
6
9
5
11
4
Leaf Nodes : Nodes without any children

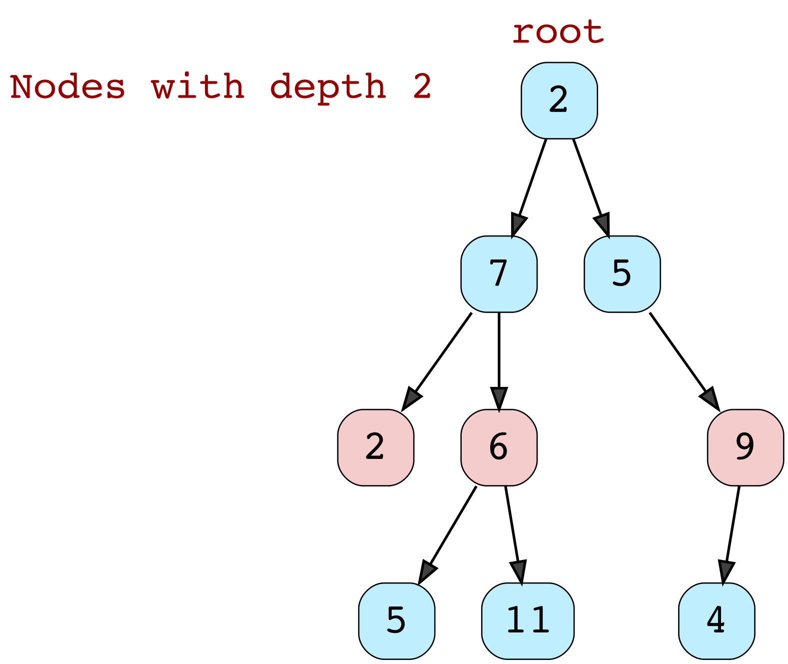
Depth of a Node

The length of the path from a node, *n*, to the root node. The depth of the root node is 0.

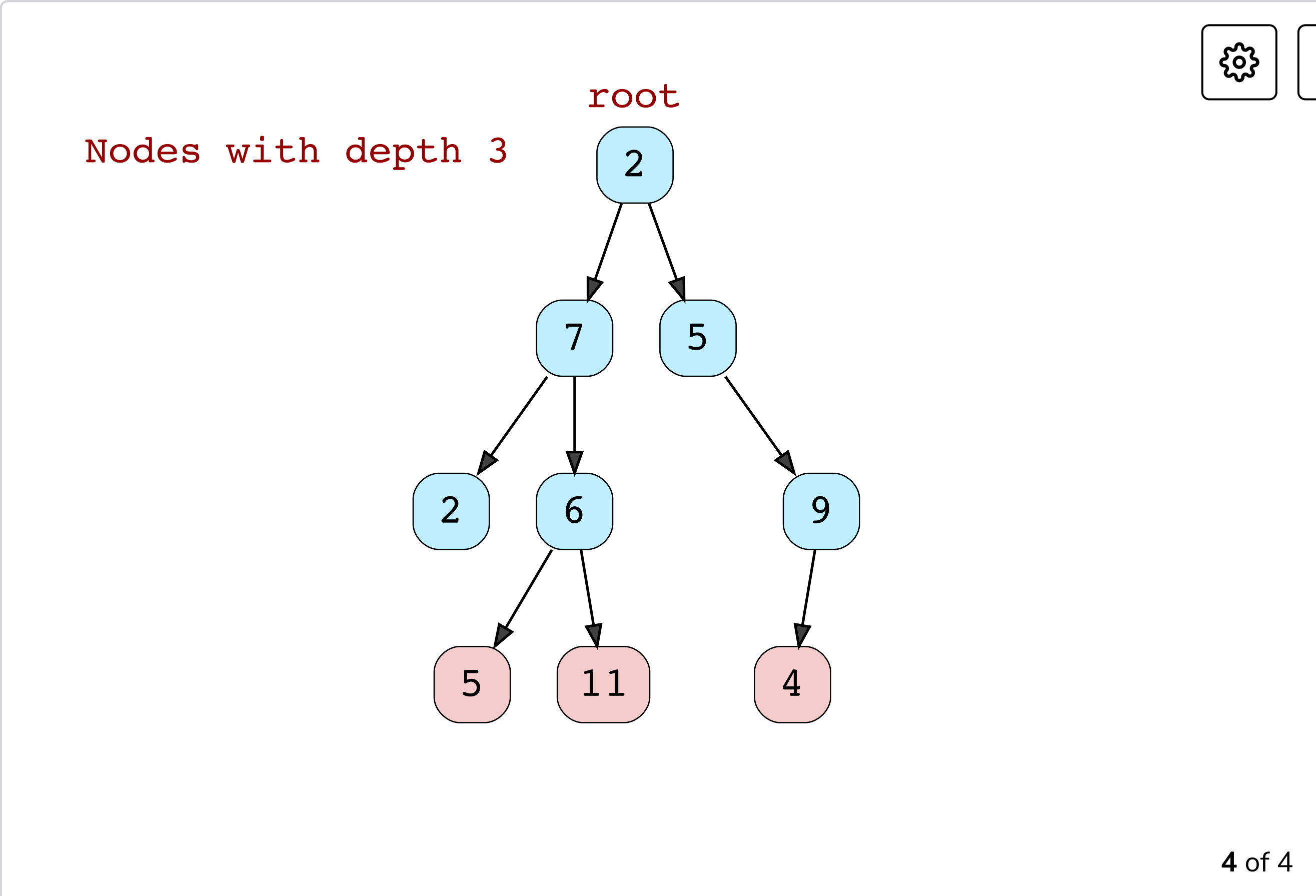



root
Nodes with depth 1
2
7
5
2
6
9
5
11
4

2 of 4

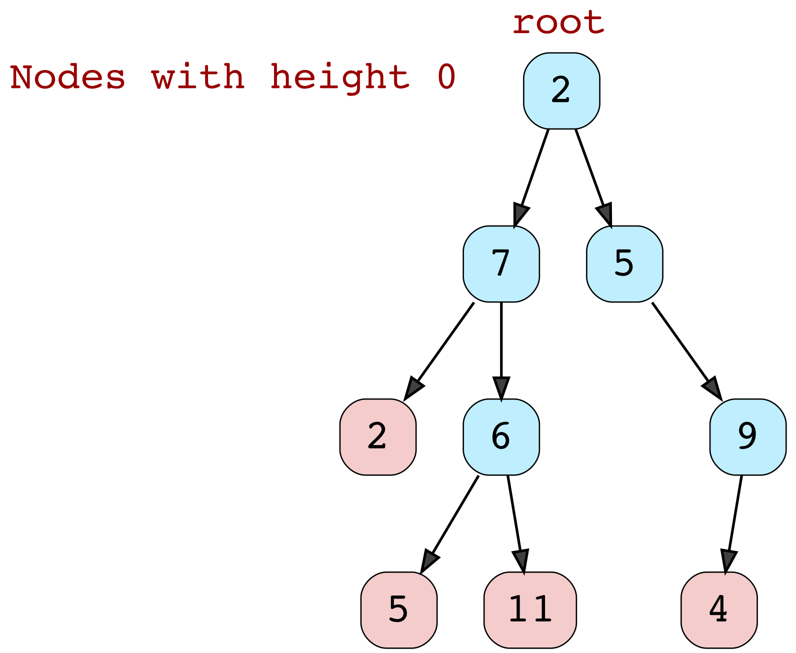

root
Nodes with depth 2
2
7
5
2
6
9
5
11
4

3 of 4

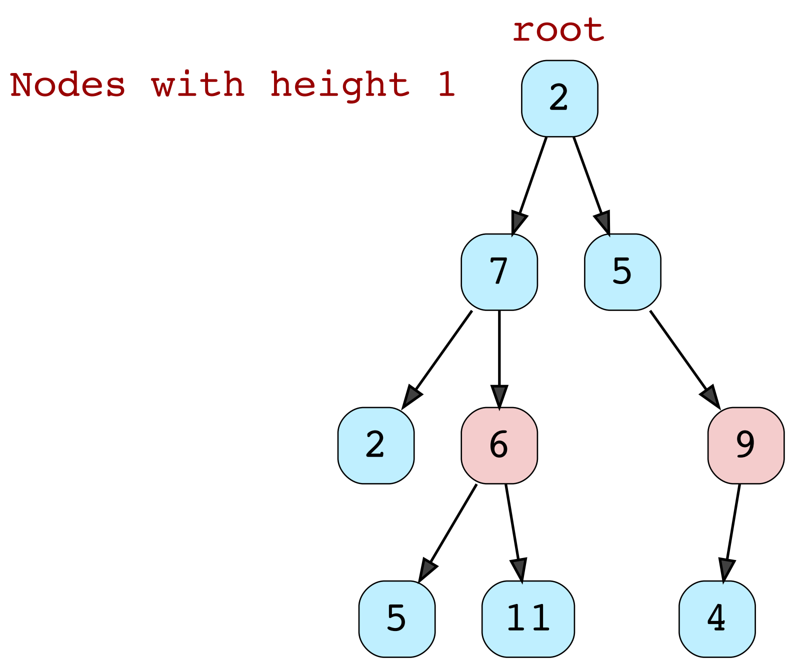


Height of a Tree

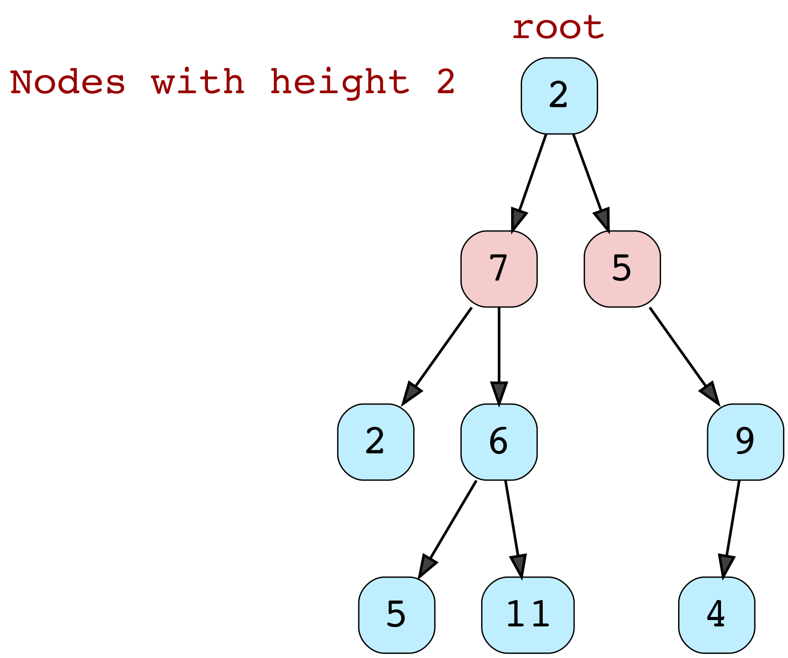
The length of the path from *n* to its deepest descendant. The height of the tree itself is the height of the root node, and the height of leaf nodes is always 0.


root
Nodes with height 0
2
7
5
2
6
9
5
11
4

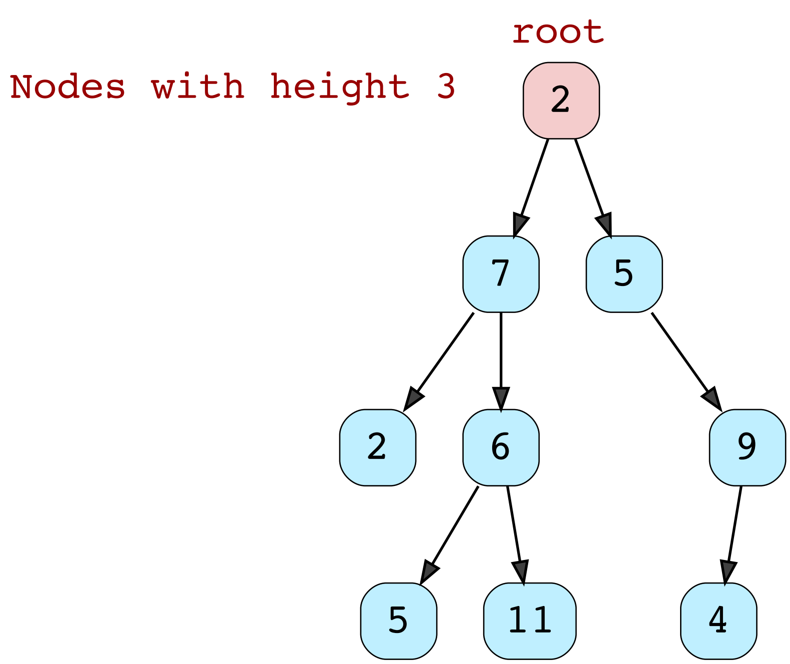
1 of 5


root
Nodes with height 1
2
7
5
2
6
9
5
11
4

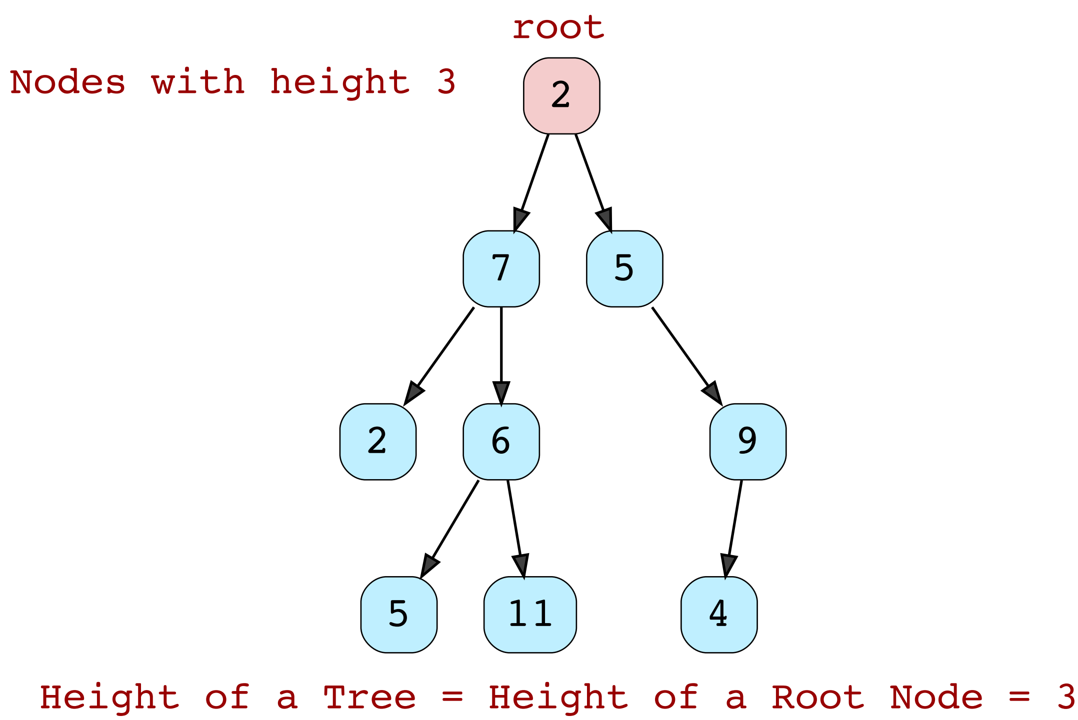
2 of 5


root
Nodes with height 2
2
7
5
2
6
9
5
11
4

3 of 5


root
Nodes with height 3
2
7
5
2
6
9
5
11
4

4 of 5

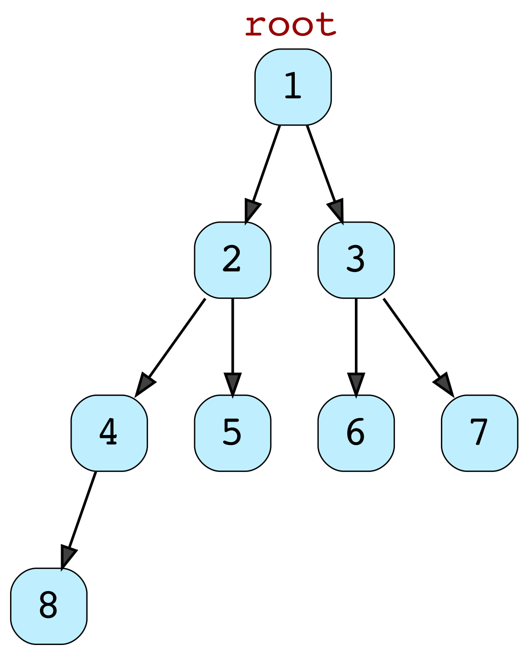


5 of 5

Types of Binary Trees

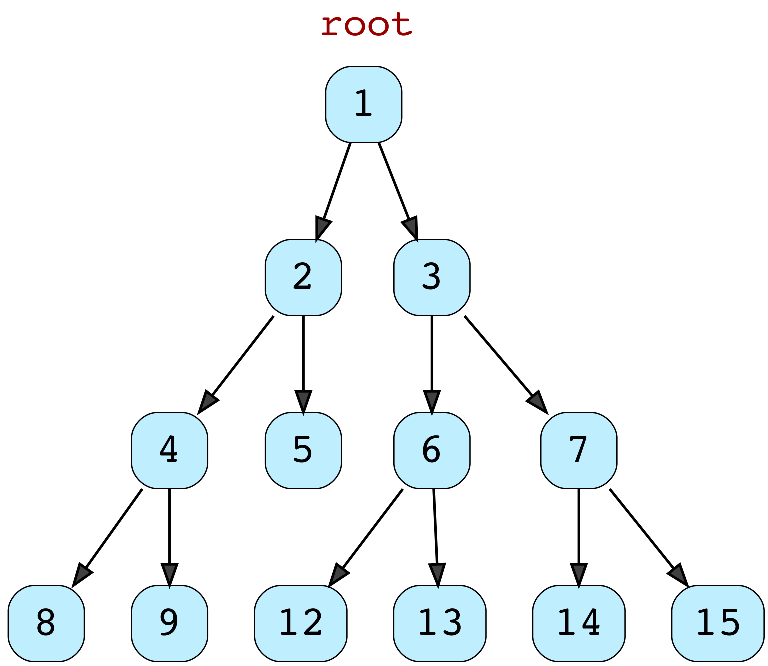
Complete Binary Tree

In a **complete** binary tree, every level *except possibly the last*, is completely filled and all nodes in the last level are as far left as possible.



Full Binary Tree

A **full** binary tree (sometimes referred to as a **proper** or **plane** binary tree) is a tree in which every node has either 0 or 2 children.



Implementation

To implement a binary tree in Python, we will first implement the `Node` class.

```
class Node(object):
  def __init__(self, value):
    self.value = value
    self.left = None
    self.right = None
```

In the code above, we have defined the `Node` class with a "new" style of defining classes in Python. The `Node` class has three attributes:

1. `self.value`
2. `self.left`
3. `self.right`

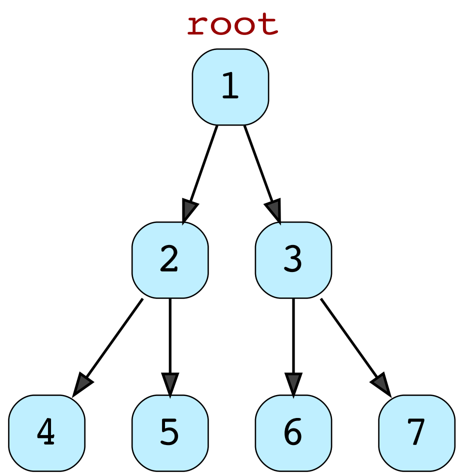
`self.value` will be equal to the `value` passed to the constructor while `self.left` and `self.right` will contain the nodes which will be the left and the right child of this node.

Let's go ahead and implement `BinaryTree` class:

```
1  class Node(object):
2    def __init__(self, value):
3      self.value = value
4      self.left = None
5      self.right = None
6
7  class BinaryTree(object):
8    def __init__(self, root):
9      self.root = Node(root)
10
11
12 tree = BinaryTree(1)
13 tree.root.left = Node(2)
14 tree.root.right = Node(3)
15 tree.root.left.left = Node(4)
16 tree.root.left.right = Node(5)
17 tree.root.right.left = Node(6)
18 tree.root.right.right = Node(7)
```

Succeeded

The `BinaryTree` contains `root`, which is an object of the `Node` class and contains the value passed as `root` in the constructor of `BinaryTree` class. From **lines 12-18**, we construct an object from the `BinaryTree` class and populate the tree by creating nodes. The visual representation of that tree will be as follows:



The implementation of the Binary Tree was pretty straightforward. At this point, we need some way to traverse the tree and visualize through code in Python. In the next lesson, we will go over some of the tree traversal methods.

Back

Solution Review: Buy and Sell Stock

Next

Traversal Algorithms

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/introduction__binary-trees__data-structures-and-algorithms-in-python)

Traversal Algorithms

In this lesson, you will learn how to traverse binary trees using a depth-first search.

We'll cover the following

Tree Traversal

Tree Traversal is the process of visiting (checking or updating) each node in a tree data structure, exactly once. Unlike linked lists or one-dimensional arrays that are canonically traversed in linear order, trees may be traversed in multiple ways. They may be traversed in *depth-first* or *breadth-first* order.

There are three common ways to traverse a tree in depth-first order:

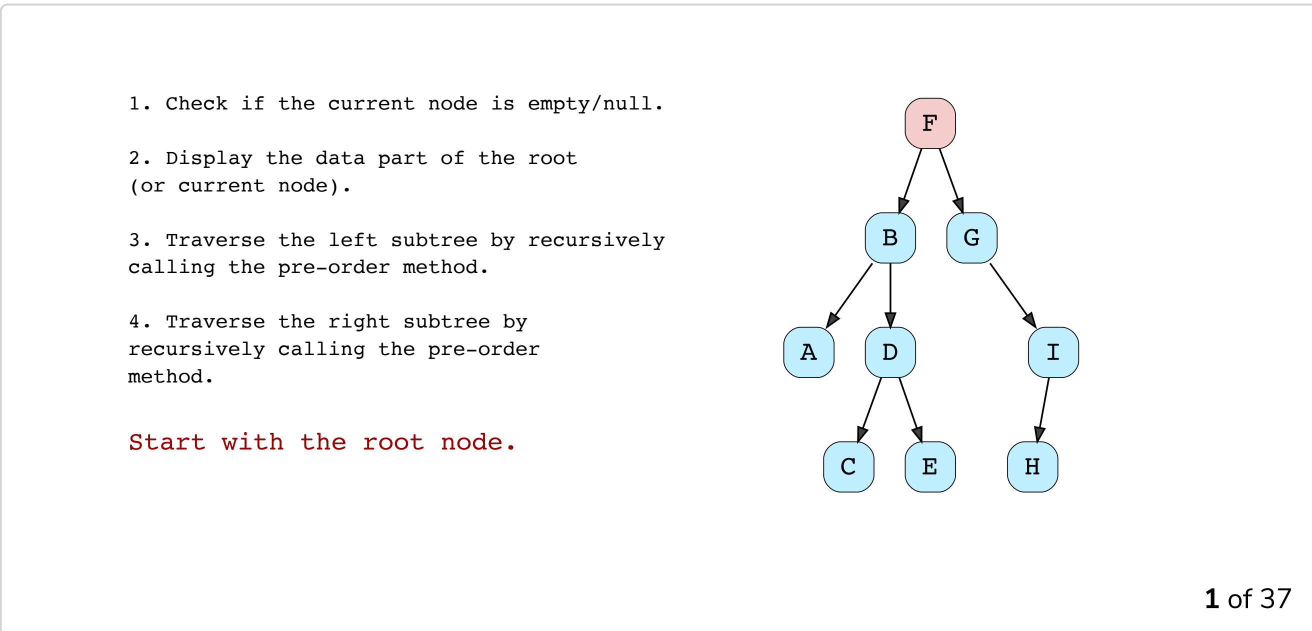
1. In-order
2. Pre-order
3. Post-order

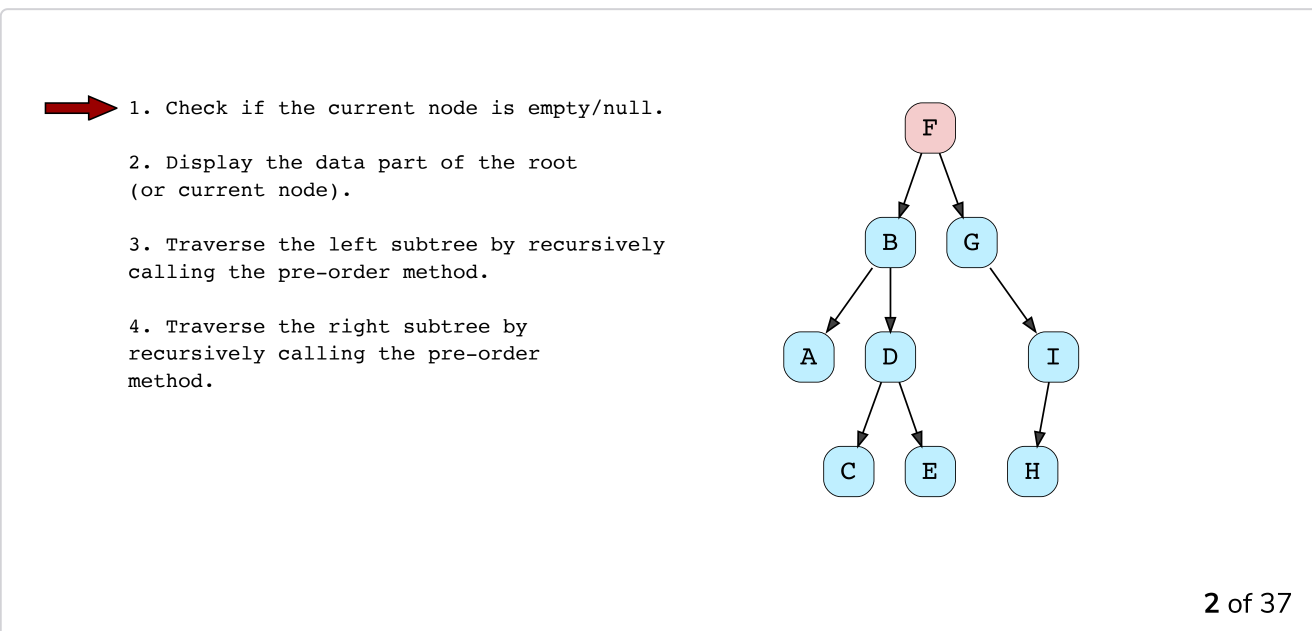
Let's begin with the Pre-order Traversal.

Pre-order Traversal

Here is the algorithm for a pre-order traversal:

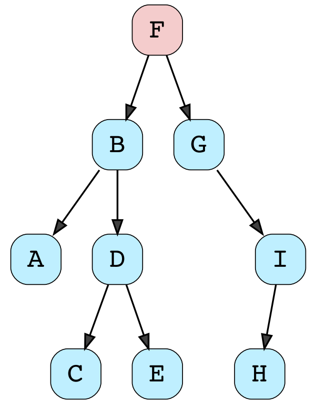
1. Check if the current node is empty/null.
2. Display the data part of the root (or current node).
3. Traverse the left subtree by recursively calling the pre-order method.
4. Traverse the right subtree by recursively calling the pre-order method.





1. Check if the current node is empty/null.
2. Display the data part of the root (or current node).
3. Traverse the left subtree by recursively calling the pre-order method.
4. Traverse the right subtree by recursively calling the pre-order method.

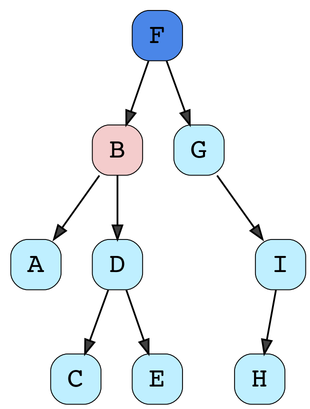
F



3 of 37

1. Check if the current node is empty/null.
2. Display the data part of the root (or current node).
3. Traverse the left subtree by recursively calling the pre-order method.
4. Traverse the right subtree by recursively calling the pre-order method.

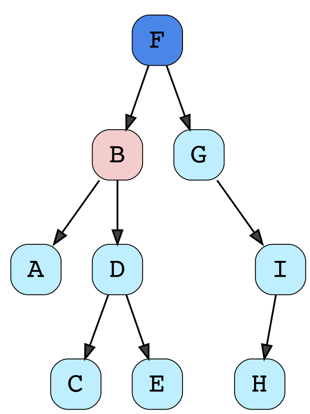
F



4 of 37

1. Check if the current node is empty/null.
2. Display the data part of the root (or current node).
3. Traverse the left subtree by recursively calling the pre-order method.
4. Traverse the right subtree by recursively calling the pre-order method.

F



5 of 37

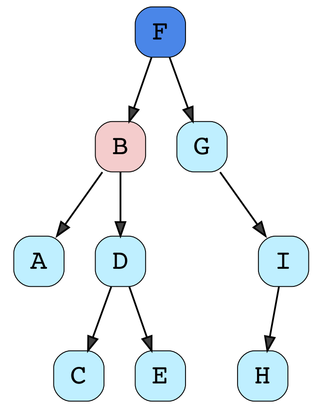
1. Check if the current node is empty/null.

2. Display the data part of the root (or current node).

3. Traverse the left subtree by recursively calling the pre-order method.

4. Traverse the right subtree by recursively calling the pre-order method.

F, B



6 of 37

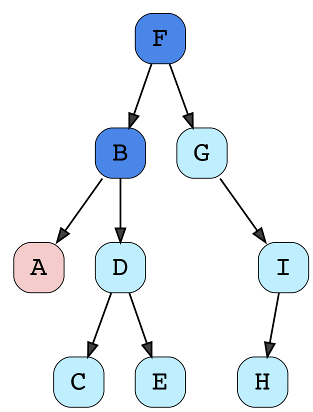
1. Check if the current node is empty/null.

2. Display the data part of the root (or current node).

3. Traverse the left subtree by recursively calling the pre-order method.

4. Traverse the right subtree by recursively calling the pre-order method.

F, B



7 of 37

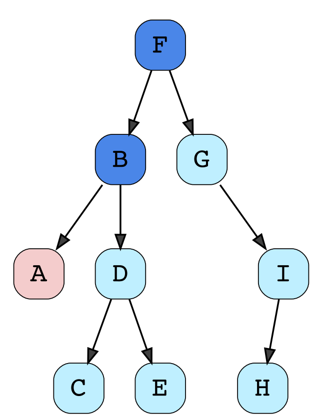
1. Check if the current node is empty/null.

2. Display the data part of the root (or current node).

3. Traverse the left subtree by recursively calling the pre-order method.

4. Traverse the right subtree by recursively calling the pre-order method.

F, B



8 of 37

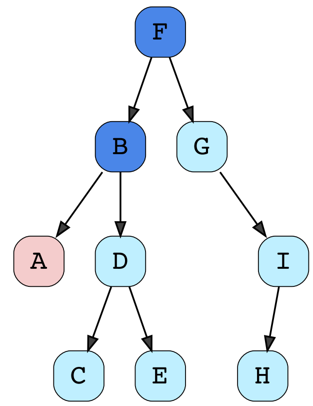
1. Check if the current node is empty/null.

2. Display the data part of the root (or current node).

3. Traverse the left subtree by recursively calling the pre-order method.

4. Traverse the right subtree by recursively calling the pre-order method.

F, B, A



9 of 37

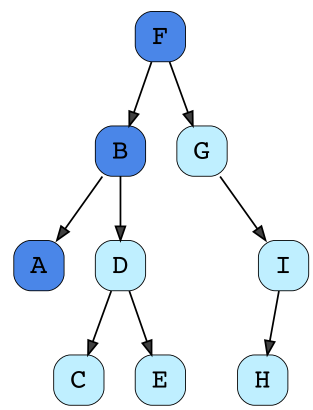
1. Check if the current node is empty/null.

2. Display the data part of the root (or current node).

3. Traverse the left subtree by recursively calling the pre-order method.

4. Traverse the right subtree by recursively calling the pre-order method.

F, B, A



10 of 37

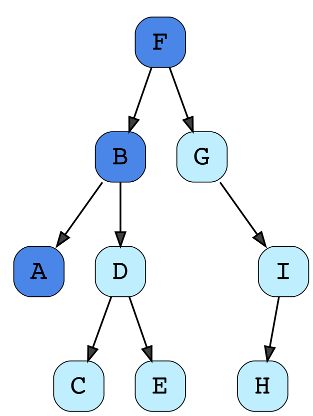
1. Check if the current node is empty/null.

2. Display the data part of the root (or current node).

3. Traverse the left subtree by recursively calling the pre-order method.

4. Traverse the right subtree by recursively calling the pre-order method.

F, B, A



11 of 37

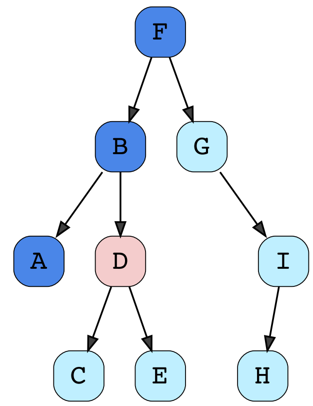
1. Check if the current node is empty/null.

2. Display the data part of the root (or current node).

3. Traverse the left subtree by recursively calling the pre-order method.

4. Traverse the right subtree by recursively calling the pre-order method.

F, B, A



12 of 37

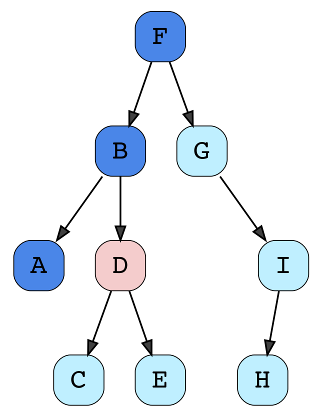
1. Check if the current node is empty/null.

2. Display the data part of the root (or current node).

3. Traverse the left subtree by recursively calling the pre-order method.

4. Traverse the right subtree by recursively calling the pre-order method.

F, B, A



13 of 37

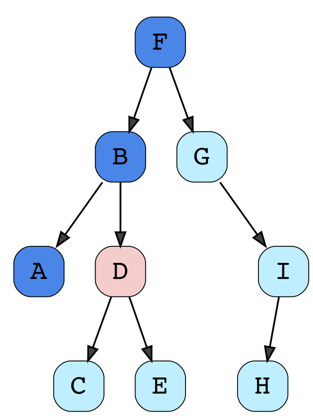
1. Check if the current node is empty/null.

2. Display the data part of the root (or current node).

3. Traverse the left subtree by recursively calling the pre-order method.

4. Traverse the right subtree by recursively calling the pre-order method.

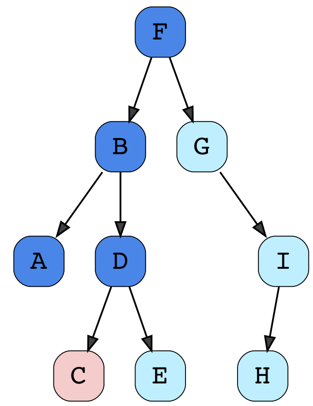
F, B, A, D



14 of 37

1. Check if the current node is empty/null.
2. Display the data part of the root (or current node).
3. Traverse the left subtree by recursively calling the pre-order method.
4. Traverse the right subtree by recursively calling the pre-order method.

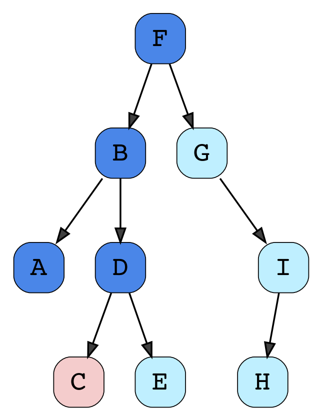
F, B, A, D



15 of 37

1. Check if the current node is empty/null.
2. Display the data part of the root (or current node).
3. Traverse the left subtree by recursively calling the pre-order method.
4. Traverse the right subtree by recursively calling the pre-order method.

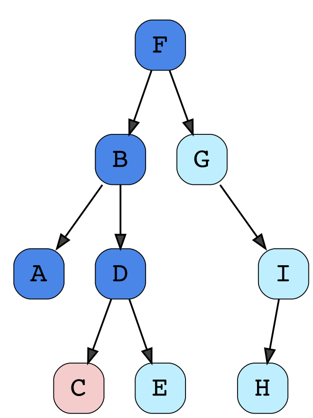
F, B, A, D



16 of 37

1. Check if the current node is empty/null.
2. Display the data part of the root (or current node).
3. Traverse the left subtree by recursively calling the pre-order method.
4. Traverse the right subtree by recursively calling the pre-order method.

F, B, A, D, C



17 of 37

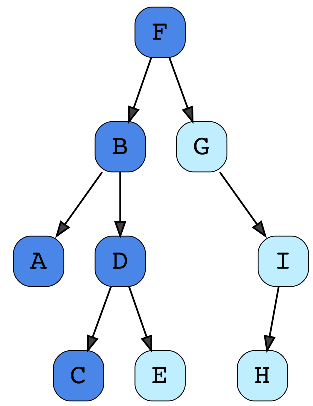
1. Check if the current node is empty/null.

2. Display the data part of the root (or current node).

3. Traverse the left subtree by recursively calling the pre-order method.

4. Traverse the right subtree by recursively calling the pre-order method.

F, B, A, D, C



18 of 37

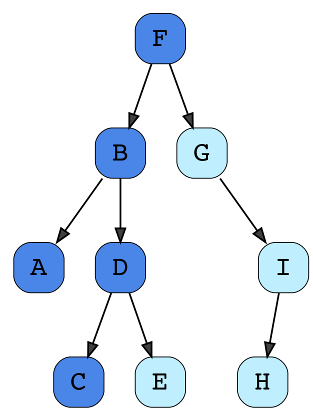
1. Check if the current node is empty/null.

2. Display the data part of the root (or current node).

3. Traverse the left subtree by recursively calling the pre-order method.

4. Traverse the right subtree by recursively calling the pre-order method.

F, B, A, D, C



19 of 37

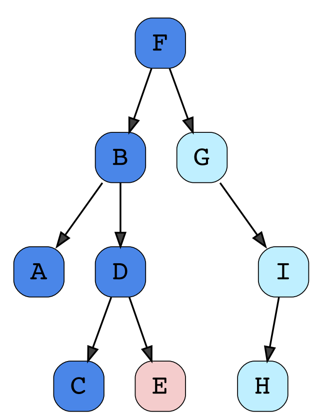
1. Check if the current node is empty/null.

2. Display the data part of the root (or current node).

3. Traverse the left subtree by recursively calling the pre-order method.

4. Traverse the right subtree by recursively calling the pre-order method.

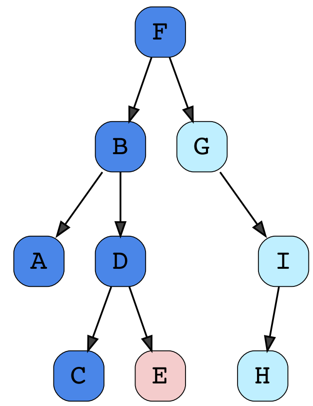
F, B, A, D, C



20 of 37

1. Check if the current node is empty/null.
2. Display the data part of the root (or current node).
3. Traverse the left subtree by recursively calling the pre-order method.
4. Traverse the right subtree by recursively calling the pre-order method.

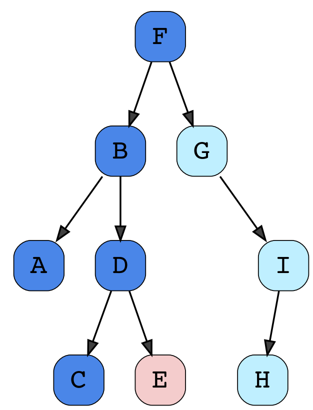
F, B, A, D, C



21 of 37

1. Check if the current node is empty/null.
2. Display the data part of the root (or current node).
3. Traverse the left subtree by recursively calling the pre-order method.
4. Traverse the right subtree by recursively calling the pre-order method.

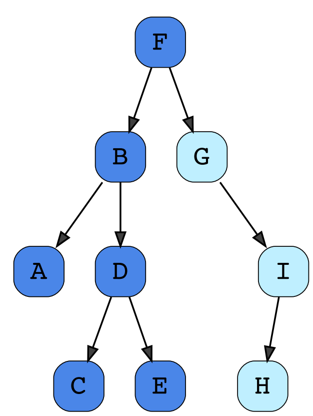
F, B, A, D, C, E



22 of 37

1. Check if the current node is empty/null.
2. Display the data part of the root (or current node).
3. Traverse the left subtree by recursively calling the pre-order method.
4. Traverse the right subtree by recursively calling the pre-order method.

F, B, A, D, C, E



23 of 37

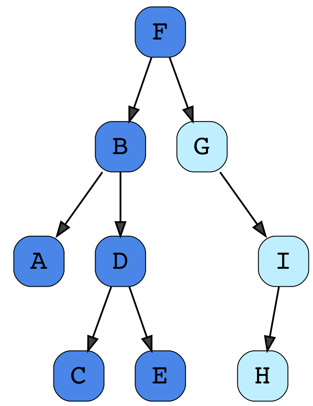
1. Check if the current node is empty/null.

2. Display the data part of the root (or current node).

3. Traverse the left subtree by recursively calling the pre-order method.

4. Traverse the right subtree by recursively calling the pre-order method.

F, B, A, D, C, E



24 of 37

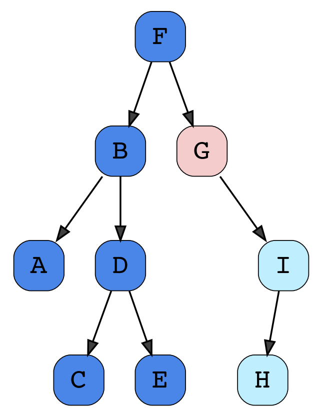
1. Check if the current node is empty/null.

2. Display the data part of the root (or current node).

3. Traverse the left subtree by recursively calling the pre-order method.

4. Traverse the right subtree by recursively calling the pre-order method.

F, B, A, D, C, E



25 of 37

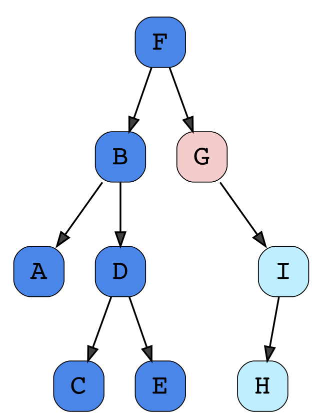
1. Check if the current node is empty/null.

2. Display the data part of the root (or current node).

3. Traverse the left subtree by recursively calling the pre-order method.

4. Traverse the right subtree by recursively calling the pre-order method.

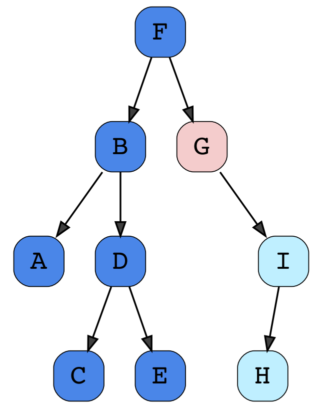
F, B, A, D, C, E



26 of 37

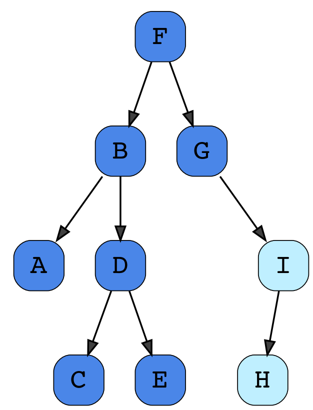
1. Check if the current node is empty/null.
2. Display the data part of the root (or current node).
3. Traverse the left subtree by recursively calling the pre-order method.
4. Traverse the right subtree by recursively calling the pre-order method.

F, B, A, D, C, E, G



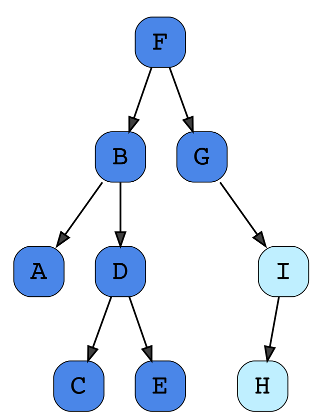
1. Check if the current node is empty/null.
2. Display the data part of the root (or current node).
3. Traverse the left subtree by recursively calling the pre-order method.
4. Traverse the right subtree by recursively calling the pre-order method.

F, B, A, D, C, E, G



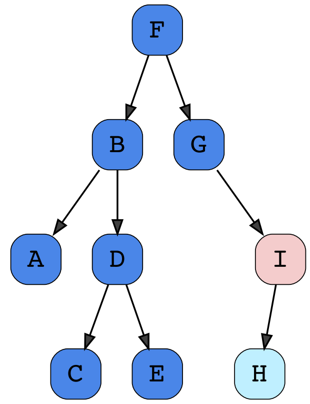
1. Check if the current node is empty/null.
2. Display the data part of the root (or current node).
3. Traverse the left subtree by recursively calling the pre-order method.
4. Traverse the right subtree by recursively calling the pre-order method.

F, B, A, D, C, E, G



1. Check if the current node is empty/null.
2. Display the data part of the root (or current node).
3. Traverse the left subtree by recursively calling the pre-order method.
4. Traverse the right subtree by recursively calling the pre-order method.

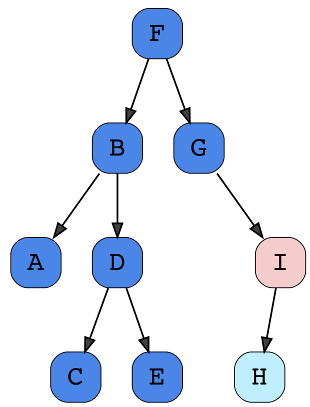
F, B, A, D, C, E, G



30 of 37

1. Check if the current node is empty/null.
2. Display the data part of the root (or current node).
3. Traverse the left subtree by recursively calling the pre-order method.
4. Traverse the right subtree by recursively calling the pre-order method.

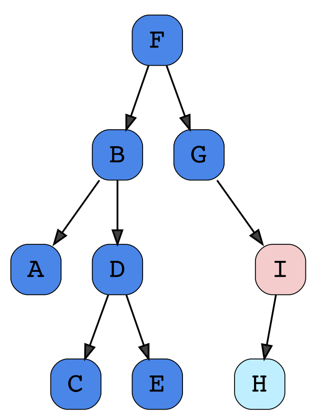
F, B, A, D, C, E, G



31 of 37

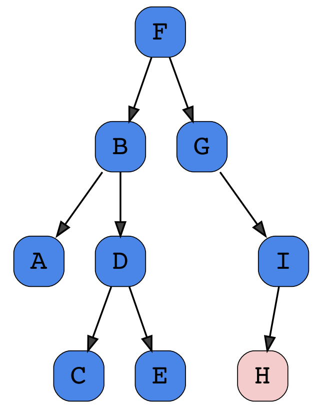
1. Check if the current node is empty/null.
2. Display the data part of the root (or current node).
3. Traverse the left subtree by recursively calling the pre-order method.
4. Traverse the right subtree by recursively calling the pre-order method.

F, B, A, D, C, E, G, I



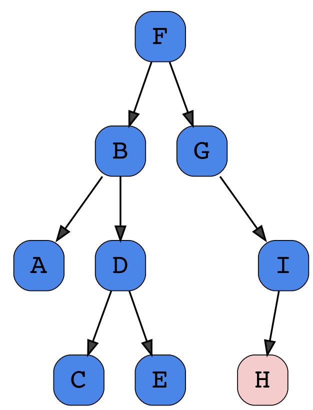
32 of 37

```
1. Check if the current node is empty/null.

2. Display the data part of the root
(or current node).

3. Traverse the left subtree by recursively
calling the pre-order method.

4. Traverse the right subtree by
recursively calling the pre-order
method.

F, B, A, D, C, E, G, I
```



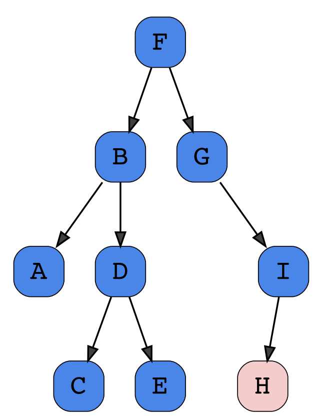
33 of 37

```
1. Check if the current node is empty/null.

2. Display the data part of the root
(or current node).

3. Traverse the left subtree by recursively
calling the pre-order method.

4. Traverse the right subtree by
recursively calling the pre-order
method.

F, B, A, D, C, E, G, I
```



34 of 37

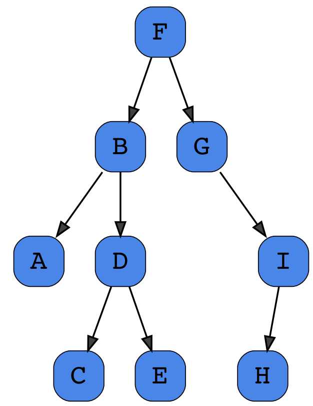
```
1. Check if the current node is empty/null.

2. Display the data part of the root
(or current node).

3. Traverse the left subtree by recursively
calling the pre-order method.

4. Traverse the right subtree by
recursively calling the pre-order
method.

F, B, A, D, C, E, G, I, H
```



35 of 37

1. Check if the current node is empty/null.
2. Display the data part of the root (or current node).
3. Traverse the left subtree by recursively calling the pre-order method.
4. Traverse the right subtree by recursively calling the pre-order method.

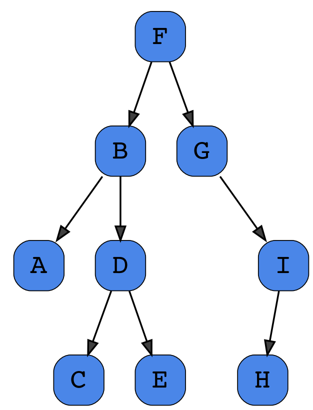
F, B, A, D, C, E, G, I, H



36 of 37

1. Check if the current node is empty/null.
2. Display the data part of the root (or current node).
3. Traverse the left subtree by recursively calling the pre-order method.
4. Traverse the right subtree by recursively calling the pre-order method.

F, B, A, D, C, E, G, I, H



37 of 37

I hope the illustrations have made the algorithm pretty clear. Let's go over its implementation in Python:

```
def preorder_print(self, start, traversal):
  """Root->Left->Right"""
  if start:
    traversal += (str(start.value) + "-")
    traversal = self.preorder_print(start.left, tr
    traversal = self.preorder_print(start.right, t
  return traversal
```

preorder_print(self, start, traversal)

Just as specified in the algorithm, we check if `start` (i.e., the current node) is empty or not. If not, then we append `start.value` to the `traversal` string and recursively call `preorder_print` on `start.left` and `start.right` which are the right and left child of the current node. Finally, we return `traversal` from the method after we have returned from all the recursive calls in case `start` is not `None`. `traversal` is just a string that will concatenate the value of nodes in an order that we visited them.

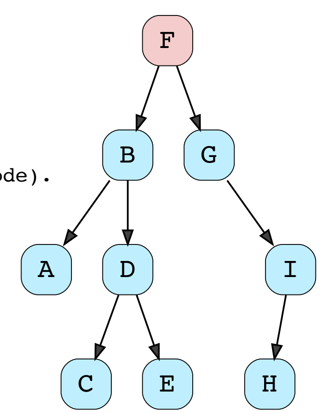
In-order Traversal

Here is the algorithm for an in-order traversal:

1. Check if the current node is empty/null.
2. Traverse the left subtree by recursively calling the in-order method.
3. Display the data part of the root (or current node).
4. Traverse the right subtree by recursively calling the in-order method.

```
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively
calling the in-order method.

3. Display the data part of the root (or current node).

4. Traverse the right subtree by recursively
calling the in-order method.
```

Start with the root node.



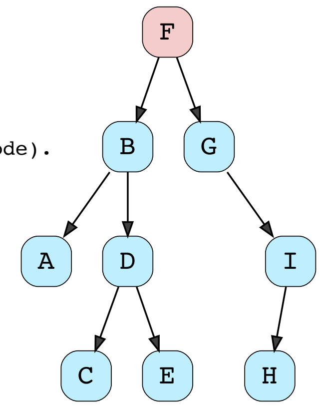
1 of 33

1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively calling the in-order method.

3. Display the data part of the root (or current node).

4. Traverse the right subtree by recursively calling the in-order method.



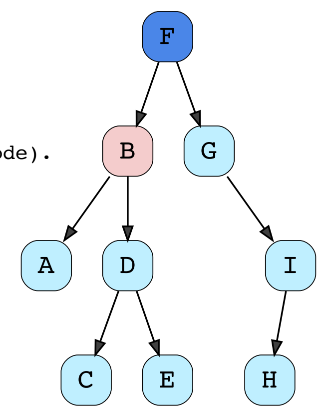
2 of 33

1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively calling the in-order method.

3. Display the data part of the root (or current node).

4. Traverse the right subtree by recursively calling the in-order method.



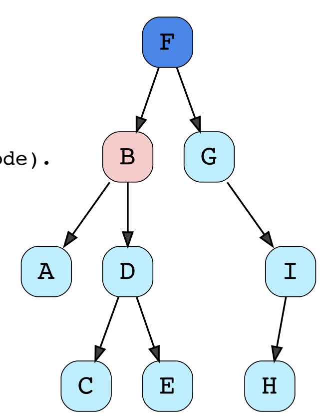
3 of 33

1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively calling the in-order method.

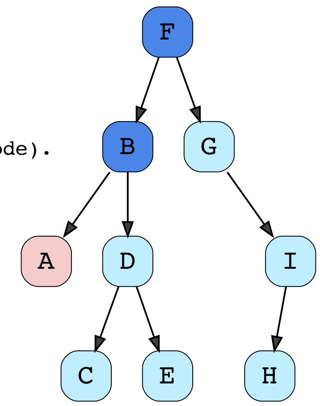
3. Display the data part of the root (or current node).

4. Traverse the right subtree by recursively calling the in-order method.



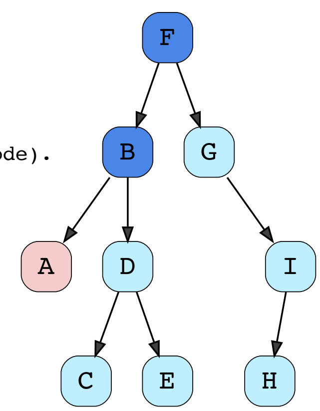
4 of 33

1. Check if the current node is empty/null.
2. Traverse the left subtree by recursively calling the in-order method.
3. Display the data part of the root (or current node).
4. Traverse the right subtree by recursively calling the in-order method.



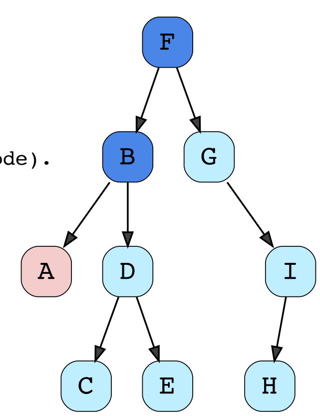
5 of 33

1. Check if the current node is empty/null.
2. Traverse the left subtree by recursively calling the in-order method.
3. Display the data part of the root (or current node).
4. Traverse the right subtree by recursively calling the in-order method.



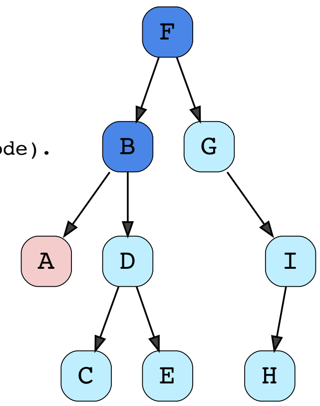
6 of 33

1. Check if the current node is empty/null.
2. Traverse the left subtree by recursively calling the in-order method.
3. Display the data part of the root (or current node).
4. Traverse the right subtree by recursively calling the in-order method.



7 of 33

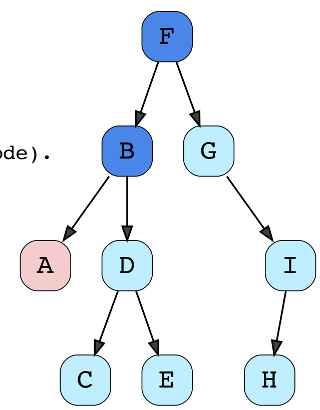
1. Check if the current node is empty/null.
2. Traverse the left subtree by recursively calling the in-order method.
3. Display the data part of the root (or current node).
4. Traverse the right subtree by recursively calling the in-order method.



8 of 33

1. Check if the current node is empty/null.
2. Traverse the left subtree by recursively calling the in-order method.
3. Display the data part of the root (or current node).
4. Traverse the right subtree by recursively calling the in-order method.

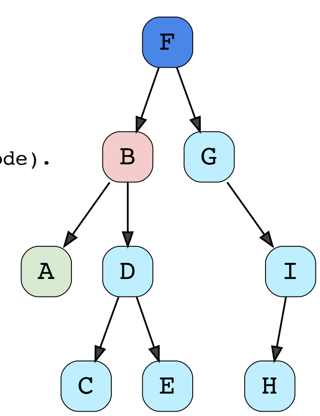
A



9 of 33

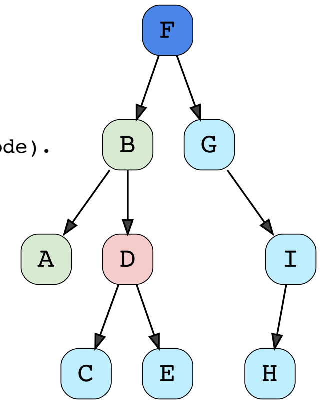
1. Check if the current node is empty/null.
2. Traverse the left subtree by recursively calling the in-order method.
3. Display the data part of the root (or current node).
4. Traverse the right subtree by recursively calling the in-order method.

A, B



10 of 33

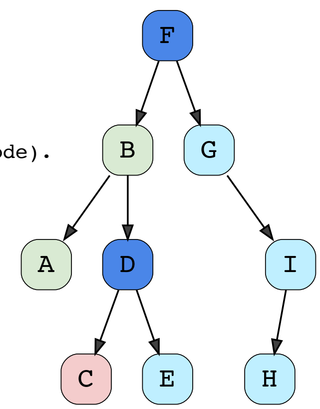
```
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively
calling the in-order method.

3. Display the data part of the root (or current node).

4. Traverse the right subtree by recursively
calling the in-order method.
```



A, B

11 of 33

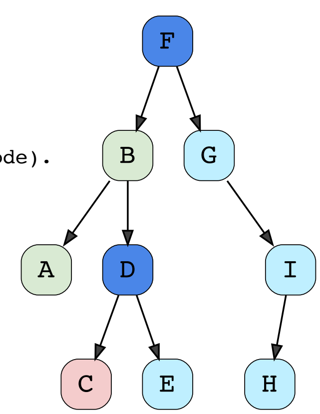
```
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively
calling the in-order method.

3. Display the data part of the root (or current node).

4. Traverse the right subtree by recursively
calling the in-order method.
```



A, B

12 of 33

```
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively
calling the in-order method.

3. Display the data part of the root (or current node).

4. Traverse the right subtree by recursively
calling the in-order method.
```



A, B

13 of 33

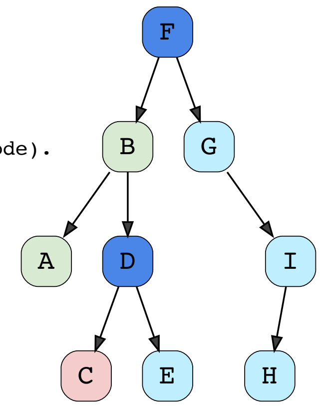
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively calling the in-order method.

3. Display the data part of the root (or current node).

4. Traverse the right subtree by recursively calling the in-order method.

A, B



14 of 33

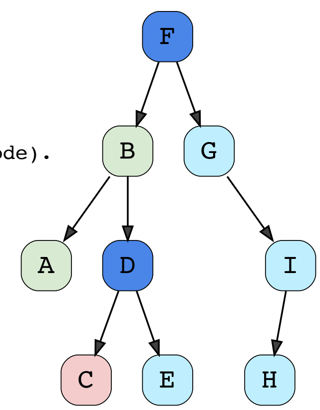
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively calling the in-order method.

3. Display the data part of the root (or current node).

4. Traverse the right subtree by recursively calling the in-order method.

A, B, C



15 of 33

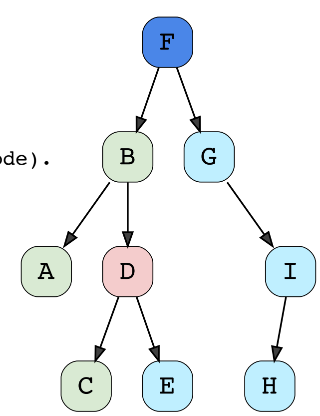
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively calling the in-order method.

3. Display the data part of the root (or current node).

4. Traverse the right subtree by recursively calling the in-order method.

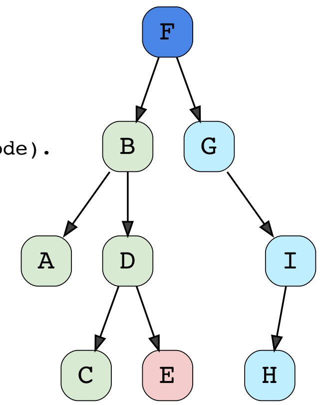
A, B, C, D



16 of 33

```
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively
calling the in-order method.

3. Display the data part of the root (or current node).

4. Traverse the right subtree by recursively
calling the in-order method.
```

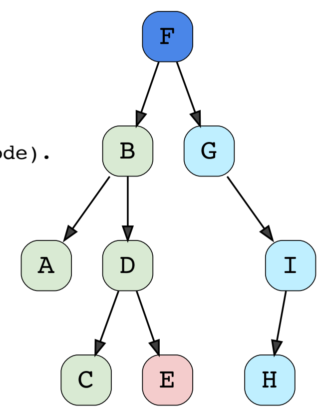
A, B, C, D



17 of 33

```
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively
calling the in-order method.

3. Display the data part of the root (or current node).

4. Traverse the right subtree by recursively
calling the in-order method.
```

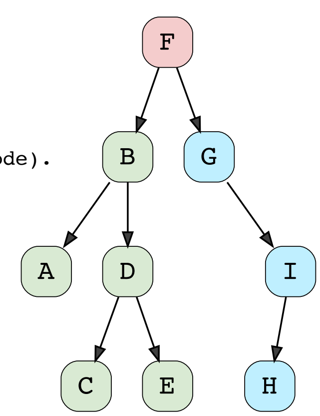
A, B, C, D, E



18 of 33

```
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively
calling the in-order method.

3. Display the data part of the root (or current node).

4. Traverse the right subtree by recursively
calling the in-order method.
```

A, B, C, D, E, F



19 of 33

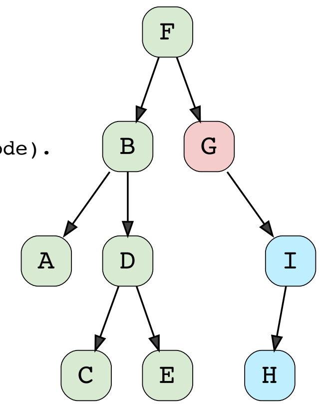
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively calling the in-order method.

3. Display the data part of the root (or current node).

4. Traverse the right subtree by recursively calling the in-order method.

A, B, C, D, E, F



20 of 33

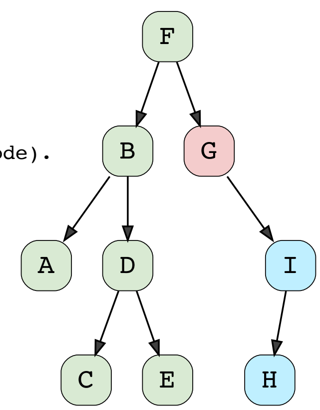
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively calling the in-order method.

3. Display the data part of the root (or current node).

4. Traverse the right subtree by recursively calling the in-order method.

A, B, C, D, E, F



21 of 33

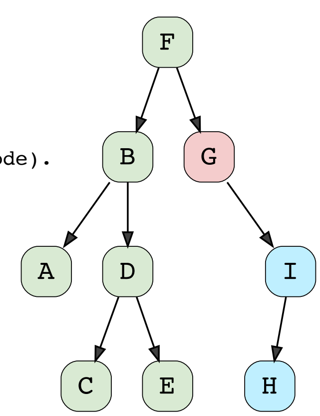
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively calling the in-order method.

3. Display the data part of the root (or current node).

4. Traverse the right subtree by recursively calling the in-order method.

A, B, C, D, E, F



22 of 33

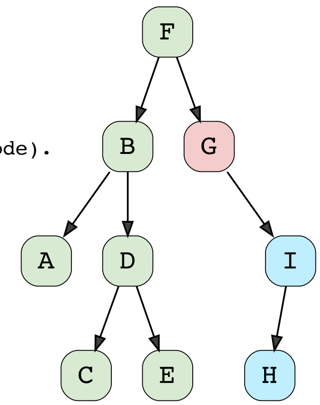
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively calling the in-order method.

3. Display the data part of the root (or current node).

4. Traverse the right subtree by recursively calling the in-order method.

A, B, C, D, E, F, G



23 of 33

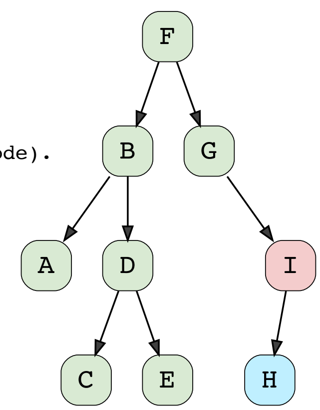
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively calling the in-order method.

3. Display the data part of the root (or current node).

4. Traverse the right subtree by recursively calling the in-order method.

A, B, C, D, E, F, G



24 of 33

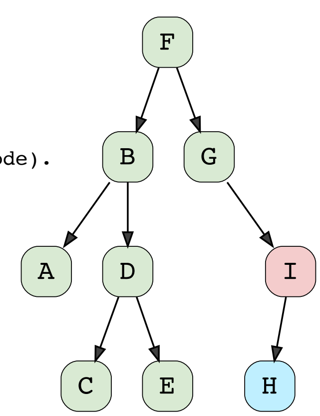
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively calling the in-order method.

3. Display the data part of the root (or current node).

4. Traverse the right subtree by recursively calling the in-order method.

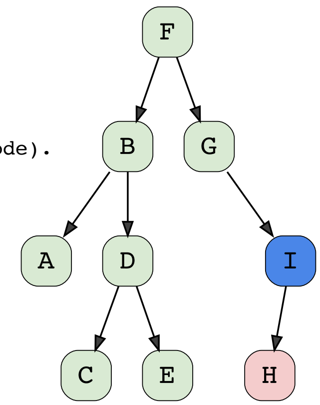
A, B, C, D, E, F, G



25 of 33

1. Check if the current node is empty/null.
2. Traverse the left subtree by recursively calling the in-order method.
3. Display the data part of the root (or current node).
4. Traverse the right subtree by recursively calling the in-order method.

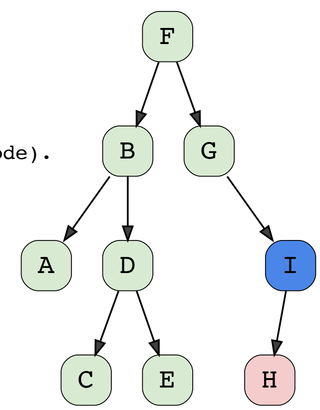
A, B, C, D, E, F, G



26 of 33

1. Check if the current node is empty/null.
2. Traverse the left subtree by recursively calling the in-order method.
3. Display the data part of the root (or current node).
4. Traverse the right subtree by recursively calling the in-order method.

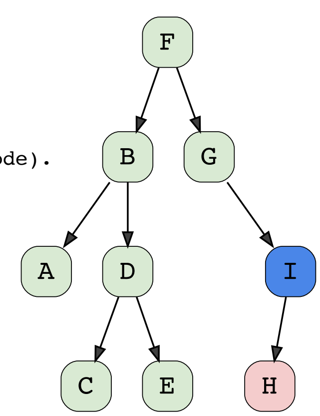
A, B, C, D, E, F, G



27 of 33

1. Check if the current node is empty/null.
2. Traverse the left subtree by recursively calling the in-order method.
3. Display the data part of the root (or current node).
4. Traverse the right subtree by recursively calling the in-order method.

A, B, C, D, E, F, G



28 of 33

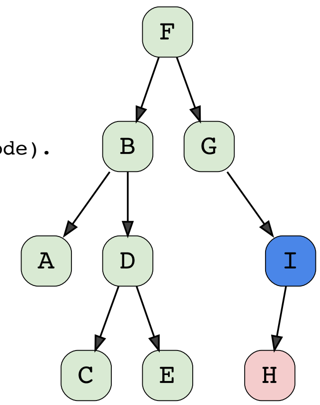
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively calling the in-order method.

3. Display the data part of the root (or current node).

4. Traverse the right subtree by recursively calling the in-order method.

A, B, C, D, E, F, G, H



29 of 33

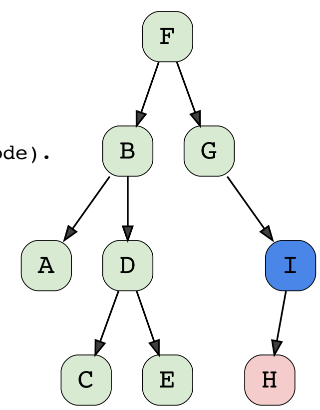
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively calling the in-order method.

3. Display the data part of the root (or current node).

4. Traverse the right subtree by recursively calling the in-order method.

A, B, C, D, E, F, G, H



30 of 33

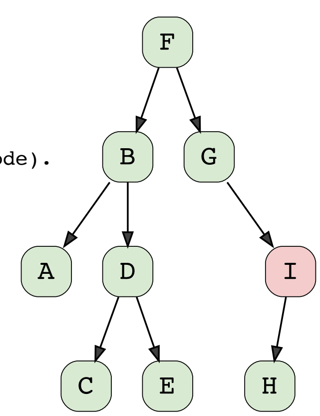
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively calling the in-order method.

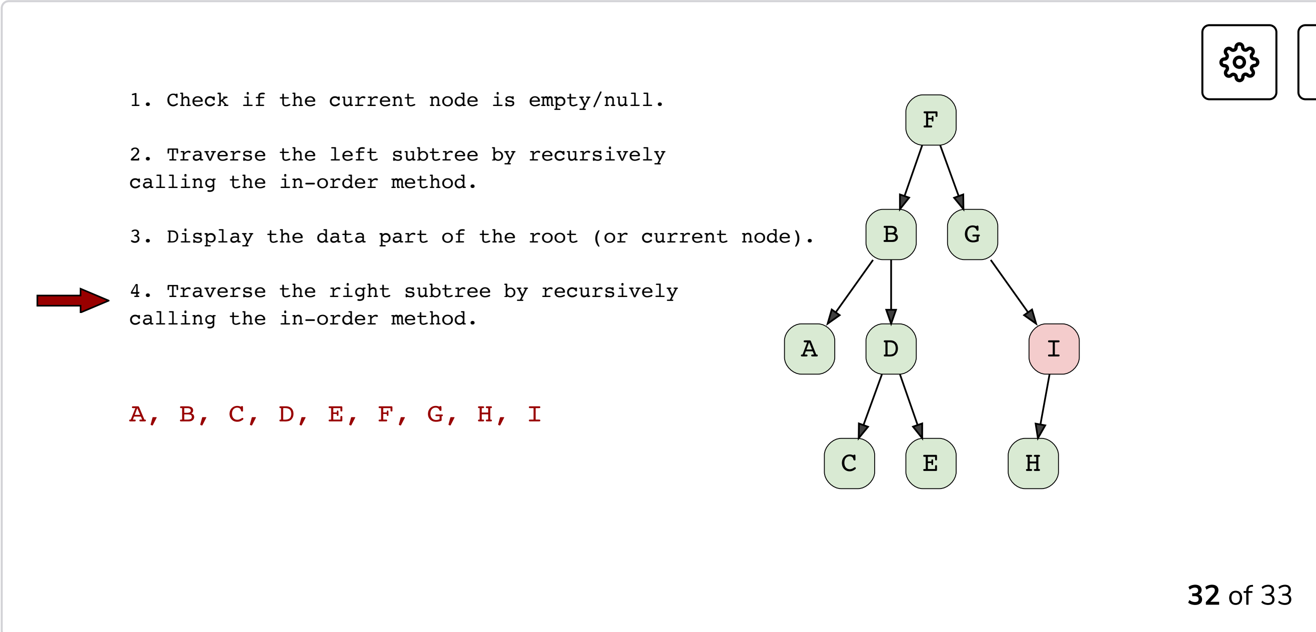
3. Display the data part of the root (or current node).

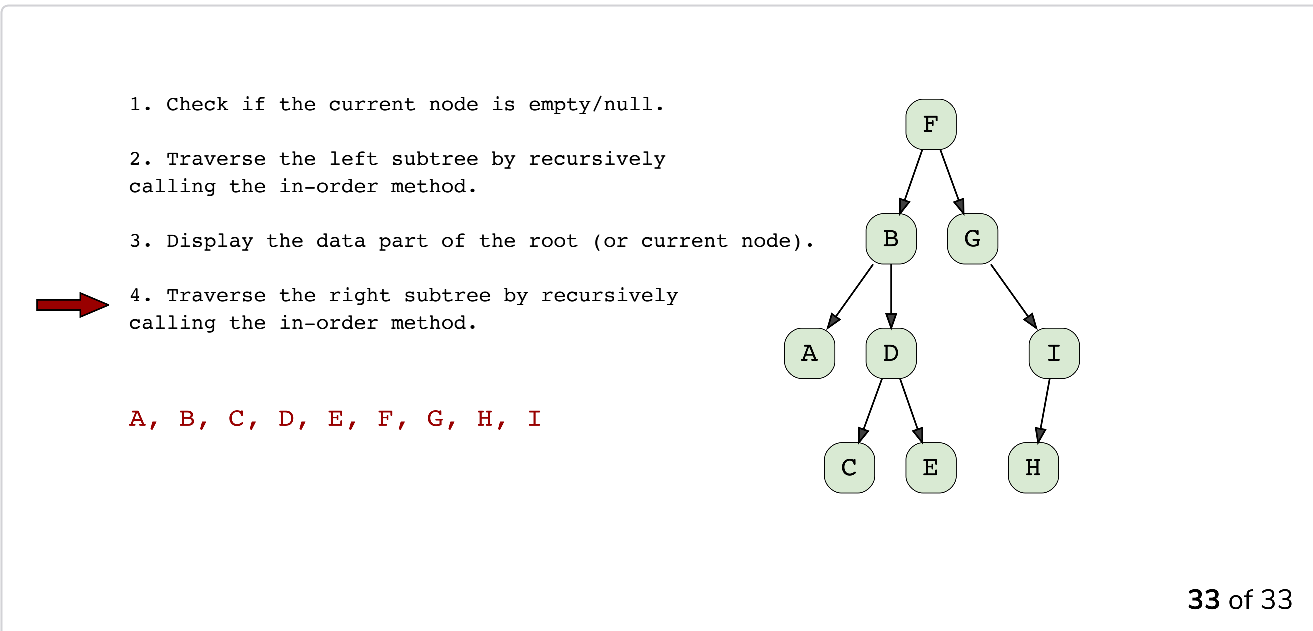
4. Traverse the right subtree by recursively calling the in-order method.

A, B, C, D, E, F, G, H



31 of 33





Now that you are familiar with the algorithm, let's jump to the code in Python:

```
def inorder_print(self, start, traversal):
        """Left->Root->Right"""
        if start:
            traversal = self.inorder_print(start.
            traversal += (str(start.value) + "-")
            traversal = self.inorder_print(start.
        return traversal
```

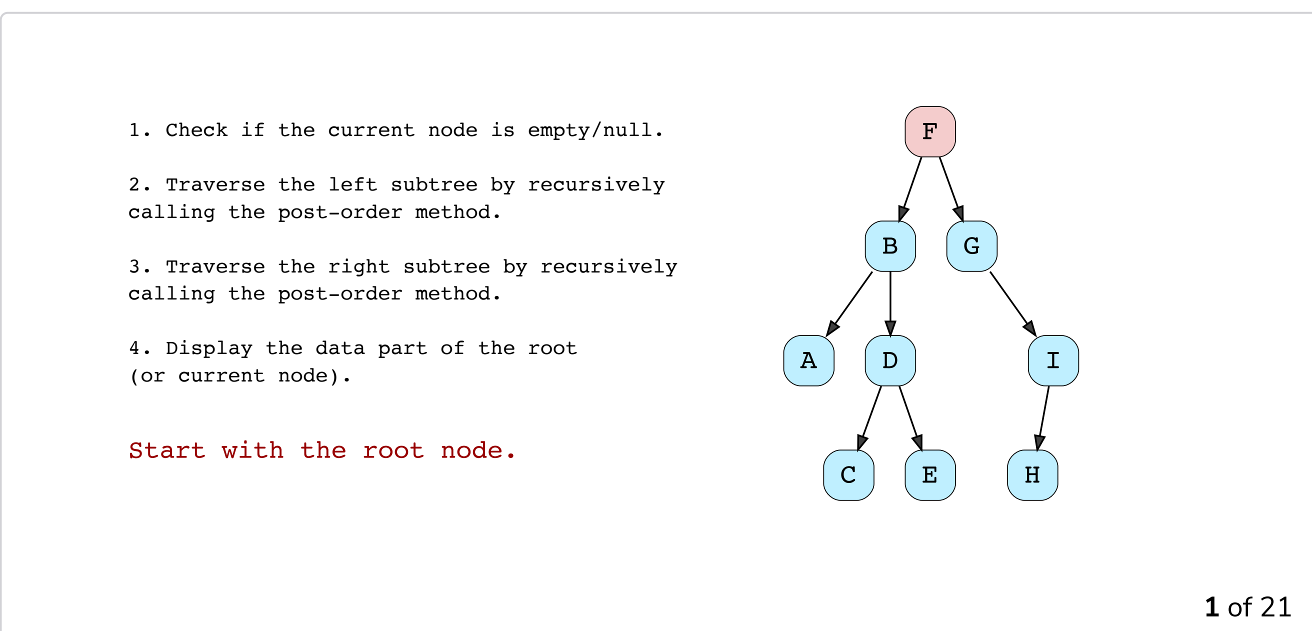
inorder_print(self, start, traversal)

The `inorder_print` is pretty much the same as the `preorder_print` except that the order *Root->Left->Right* from pre-order changes to *Left->Root->Right* in in-order traversal. In order to achieve this order, we just change the order of statements in the if-condition, i.e., we first make a recursive call on the left child and after we are done will all the subsequent calls from **line 4**, we concatenate the value of the current node with `traversal` on **line 5**. Then, we can make a recursive call to right subtree on **line 6**. This will help us keep the order required for the in-order traversal.

Post-order Traversal

At this point, it will be very easy for you to guess the algorithm for post-order traversal. There you go:

1. Check if the current node is empty/null.
2. Traverse the left subtree by recursively calling the post-order method.
3. Traverse the right subtree by recursively calling the post-order method.
4. Display the data part of the root (or current node).

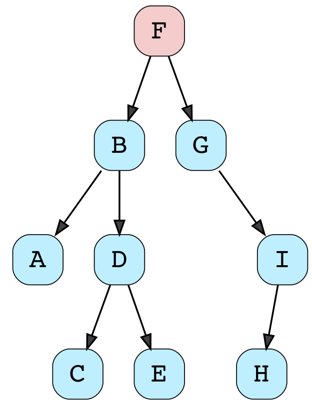


1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively calling the post-order method.

3. Traverse the right subtree by recursively calling the post-order method.

4. Display the data part of the root (or current node).



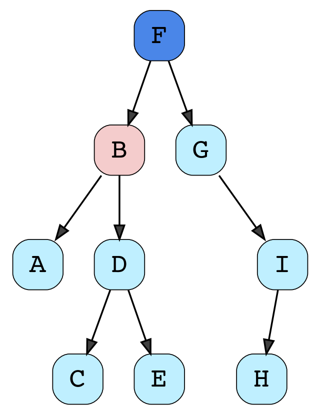
2 of 21

1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively calling the post-order method.

3. Traverse the right subtree by recursively calling the post-order method.

4. Display the data part of the root (or current node).



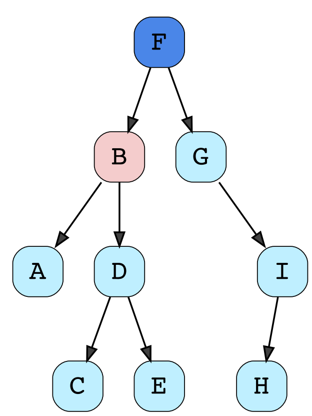
3 of 21

1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively calling the post-order method.

3. Traverse the right subtree by recursively calling the post-order method.

4. Display the data part of the root (or current node).



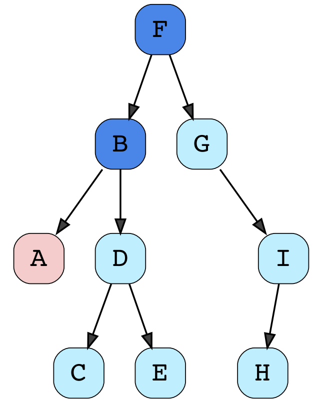
4 of 21

1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively calling the post-order method.

3. Traverse the right subtree by recursively calling the post-order method.

4. Display the data part of the root (or current node).



5 of 21

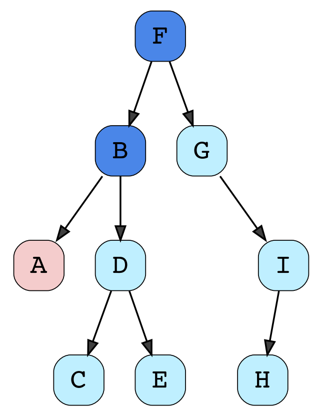
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively calling the post-order method.

3. Traverse the right subtree by recursively calling the post-order method.

4. Display the data part of the root (or current node).

A



6 of 21

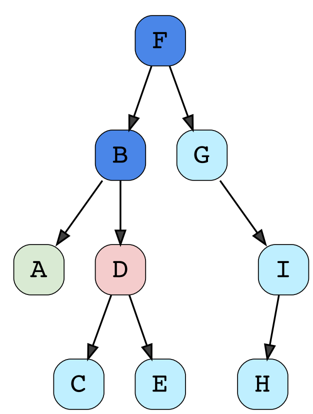
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively calling the post-order method.

3. Traverse the right subtree by recursively calling the post-order method.

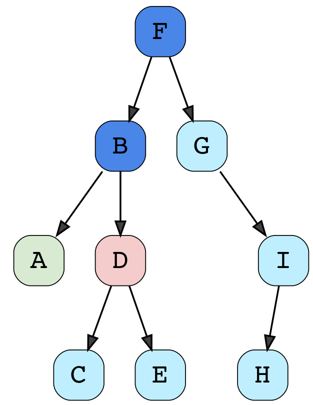
4. Display the data part of the root (or current node).

A



```
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively
calling the post-order method.

3. Traverse the right subtree by recursively
calling the post-order method.

4. Display the data part of the root
(or current node).
```

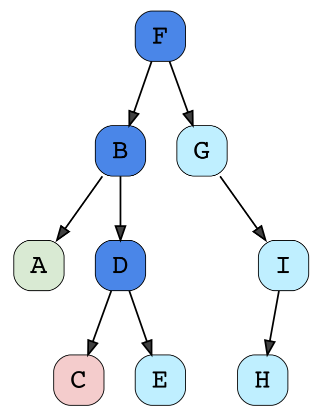
A



8 of 21

```
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively
calling the post-order method.

3. Traverse the right subtree by recursively
calling the post-order method.

4. Display the data part of the root
(or current node).
```

A



9 of 21

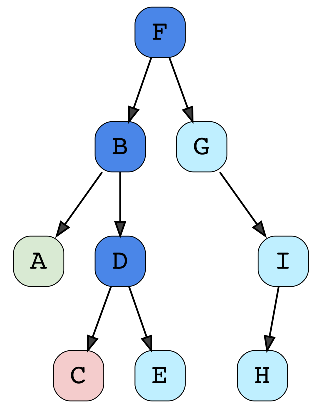
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively calling the post-order method.

3. Traverse the right subtree by recursively calling the post-order method.

4. Display the data part of the root (or current node).

A, C



10 of 21

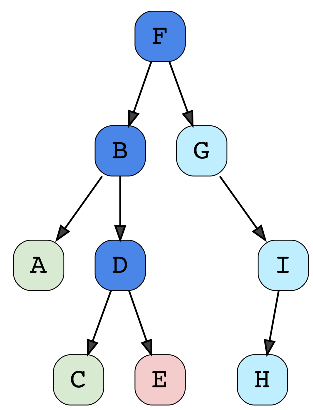
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively calling the post-order method.

3. Traverse the right subtree by recursively calling the post-order method.

4. Display the data part of the root (or current node).

A, C



11 of 21

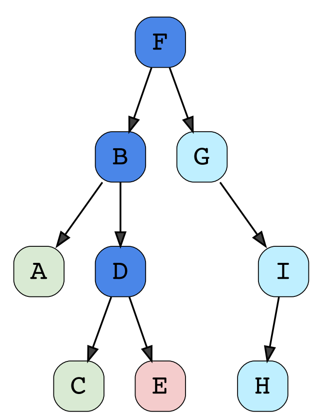
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively calling the post-order method.

3. Traverse the right subtree by recursively calling the post-order method.

4. Display the data part of the root (or current node).

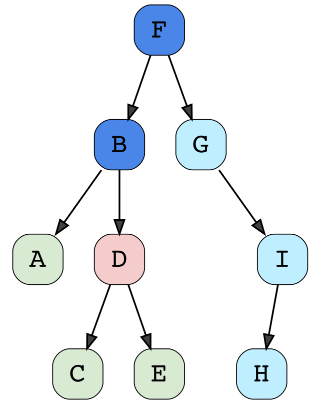
A, C, E



12 of 21

```
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively
calling the post-order method.

3. Traverse the right subtree by recursively
calling the post-order method.

4. Display the data part of the root
(or current node).
```

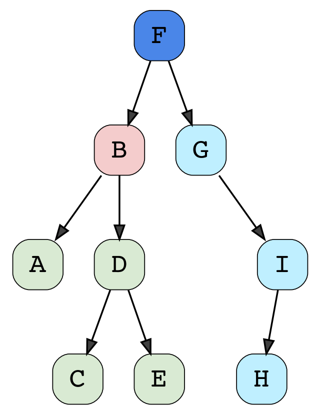
A, C, E, D



13 of 21

```
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively
calling the post-order method.

3. Traverse the right subtree by recursively
calling the post-order method.

4. Display the data part of the root
(or current node).
```

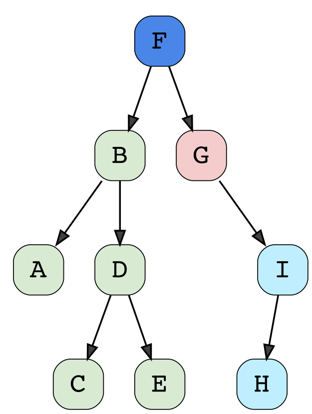
A, C, E, D, B



14 of 21

```
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively
calling the post-order method.

3. Traverse the right subtree by recursively
calling the post-order method.

4. Display the data part of the root
(or current node).
```

A, C, E, D, B



15 of 21

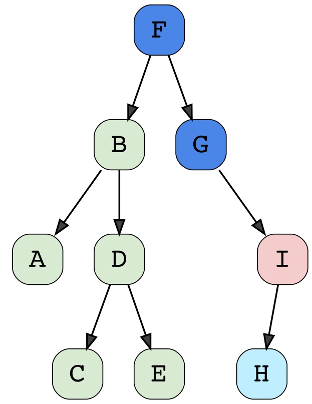
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively calling the post-order method.

3. Traverse the right subtree by recursively calling the post-order method.

4. Display the data part of the root (or current node).

A, C, E, D, B



16 of 21

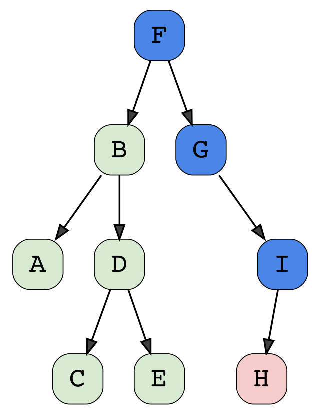
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively calling the post-order method.

3. Traverse the right subtree by recursively calling the post-order method.

4. Display the data part of the root (or current node).

A, C, E, D, B



17 of 21

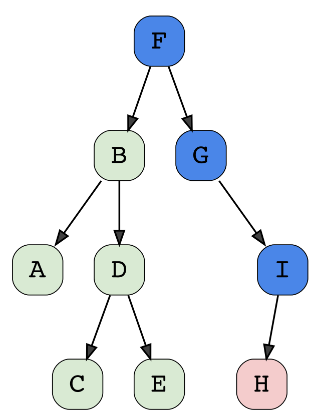
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively calling the post-order method.

3. Traverse the right subtree by recursively calling the post-order method.

4. Display the data part of the root (or current node).

A, C, E, D, B, H



18 of 21

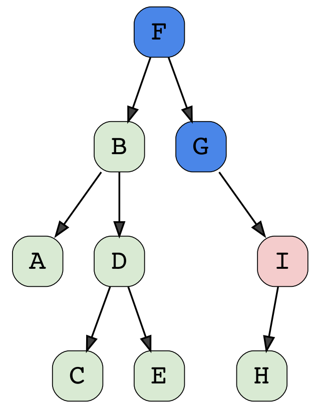
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively calling the post-order method.

3. Traverse the right subtree by recursively calling the post-order method.

4. Display the data part of the root (or current node).

A, C, E, D, B, H, I



19 of 21

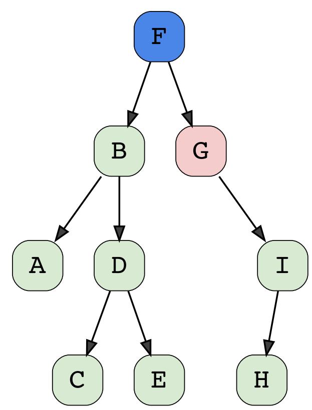
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively calling the post-order method.

3. Traverse the right subtree by recursively calling the post-order method.

4. Display the data part of the root (or current node).

A, C, E, D, B, H, I, G



20 of 21

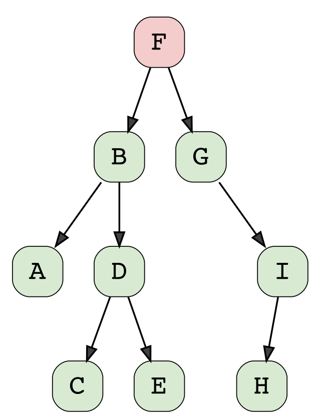
1. Check if the current node is empty/null.

2. Traverse the left subtree by recursively calling the post-order method.

3. Traverse the right subtree by recursively calling the post-order method.

4. Display the data part of the root (or current node).

A, C, E, D, B, H, I, G, F



21 of 21

Here is the implementation of post-order traversal in Python:

```
def·postorder_print(self,·start,·traversal):
··"""Left->Right->Root"""
··if·start:
····traversal·=·self.postorder_print(start.left,·t
····traversal·=·self.postorder_print(start.right,
····traversal·+=·(str(start.value)·+·"-")
··return·traversal

```

postorder_print(self, start, traversal)

Again, we just changed the order of statements. The recursive calls to the left and the right subtree have been placed before concatenating the value of the current node to `traversal`.

Helper Method

Below is the implementation of all the tree traversal methods within the `Binary Tree` class. Additionally, there is a helper method `print_tree(self, traversal_type)` which will invoke the specified method according to `traversal_type`.

```
        if start:
            traversal = self.postorder_print(start
            traversal = self.postorder_print(start
            traversal += (str(start.value) + "-")
        return traversal

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

```
#          /  \      /   \
#         4    5    6    7

# Set up tree:
tree = BinaryTree(1)
tree.root.left = Node(2)
tree.root.right = Node(3)
tree.root.left.left = Node(4)
tree.root.left.right = Node(5)
tree.root.right.left = Node(6)
tree.root.right.right = Node(7)

print(tree.print_tree("preorder"))
print(tree.print_tree("inorder"))
print(tree.print_tree("postorder"))
```

Output 0.19s

```
1-2-4-5-3-6-7-
4-2-5-1-6-3-7-
4-5-2-6-7-3-1-
```

A suggestion is to take out a piece of paper and traverse a sample binary tree yourself according to all the traversal types. Once that is done, you can confirm your results by playing around with the implementation provided above.

Hope you find these depth-first tree traversals useful! See you in the next lesson for level-order traversal which is a kind of breadth-first tree traversal.

Back
Introduction

Next
Level-Order Traversal

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/traversal-algorithms__binary-trees__data-structures-and-algorithms-in-python) educative

Level-Order Traversal

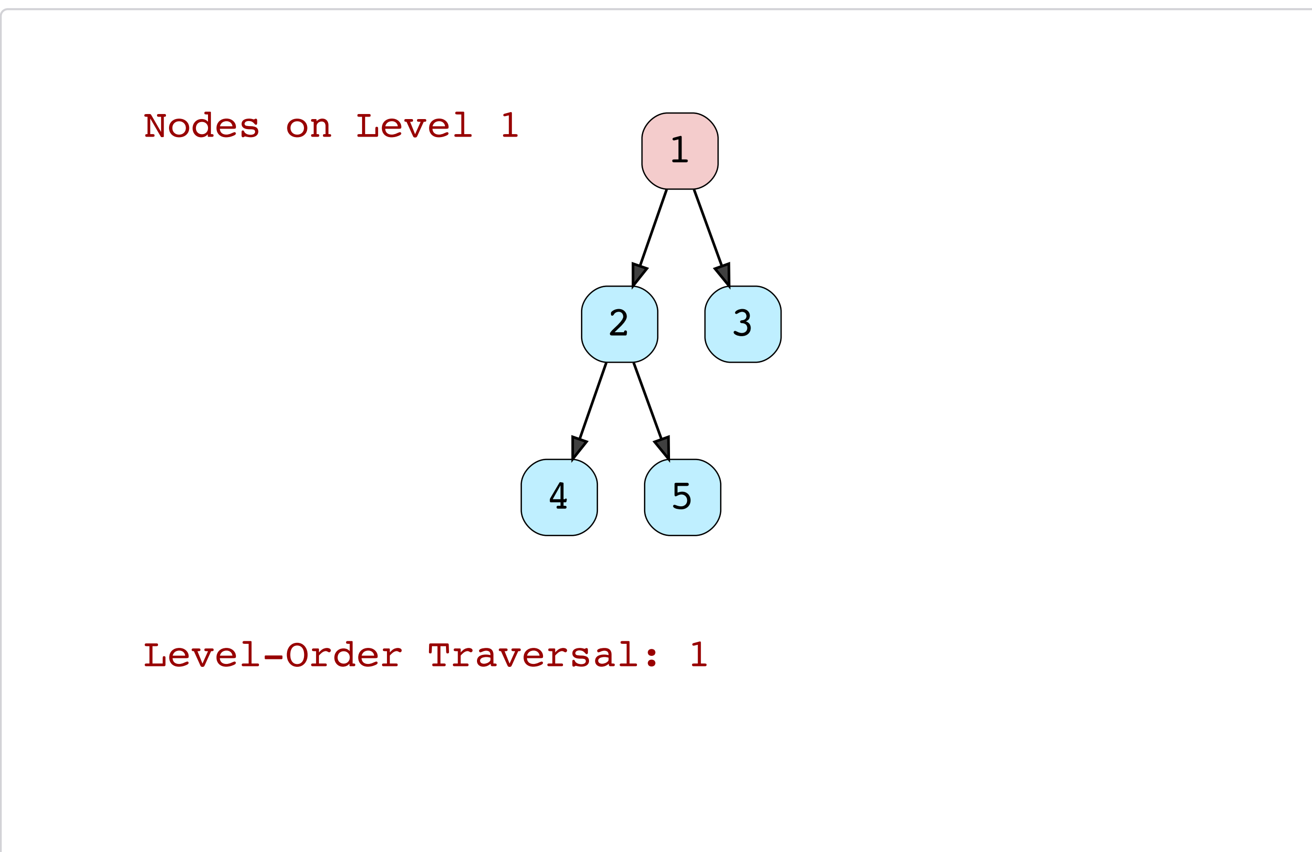
In this lesson, you will learn how to implement level-order traversal of a binary tree in Python.

We'll cover the following

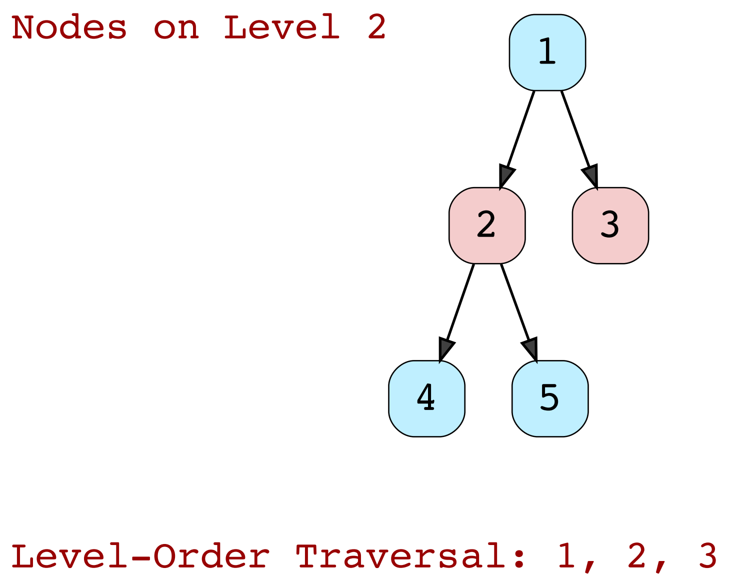
- Algorithm
- Implementation

In this lesson, we go over how to perform a level-order traversal in a binary tree. We then code a solution in Python building upon our binary tree class.

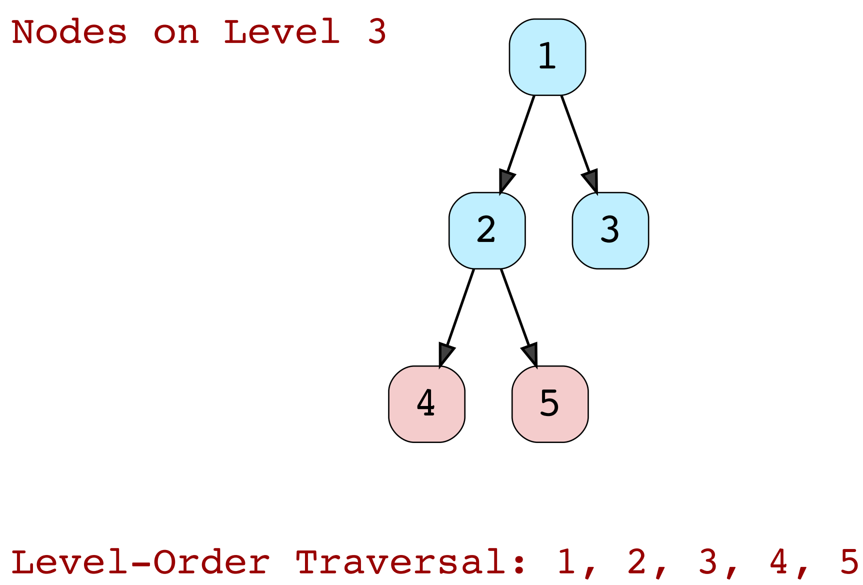
Here is an example of a level-order traversal:



1 of 3



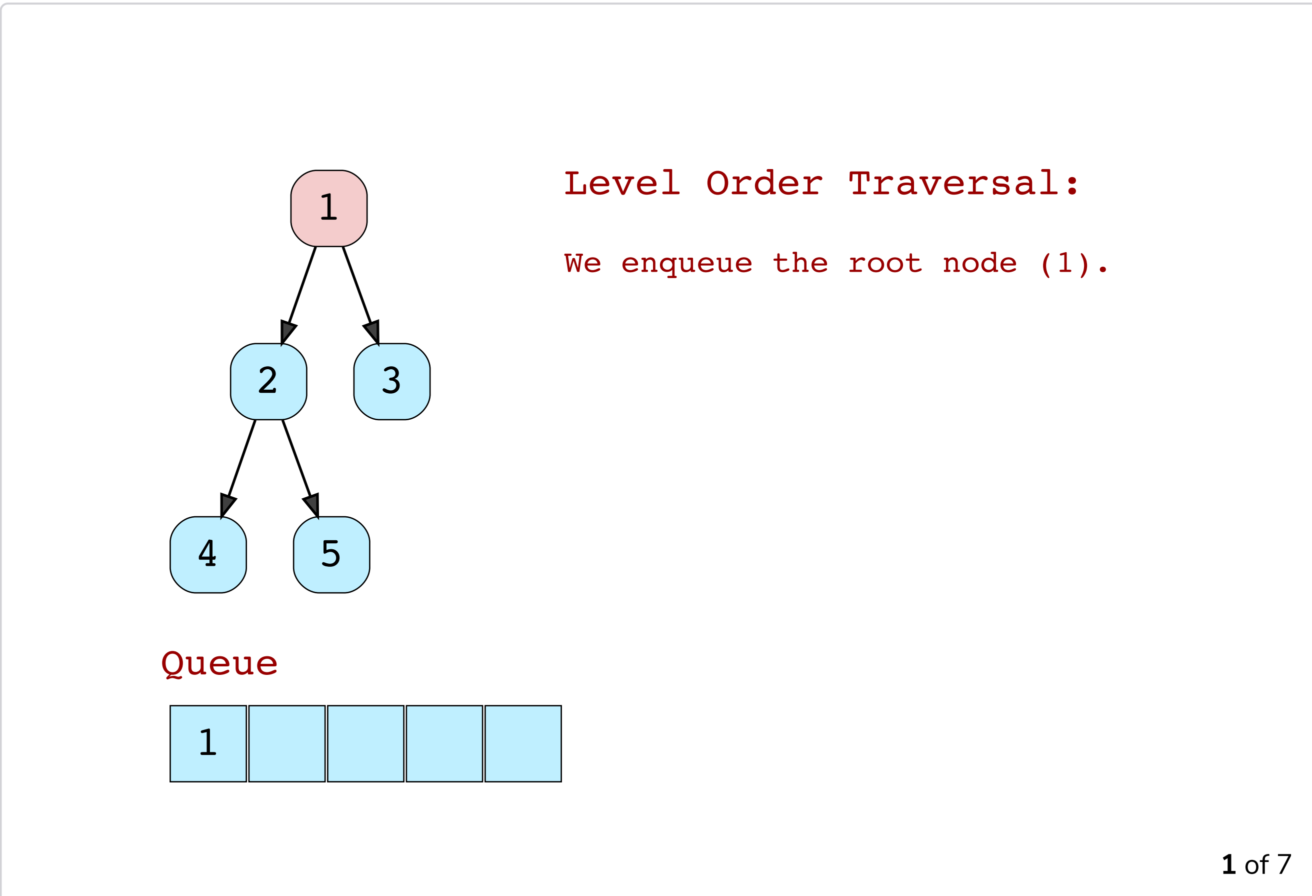
2 of 3

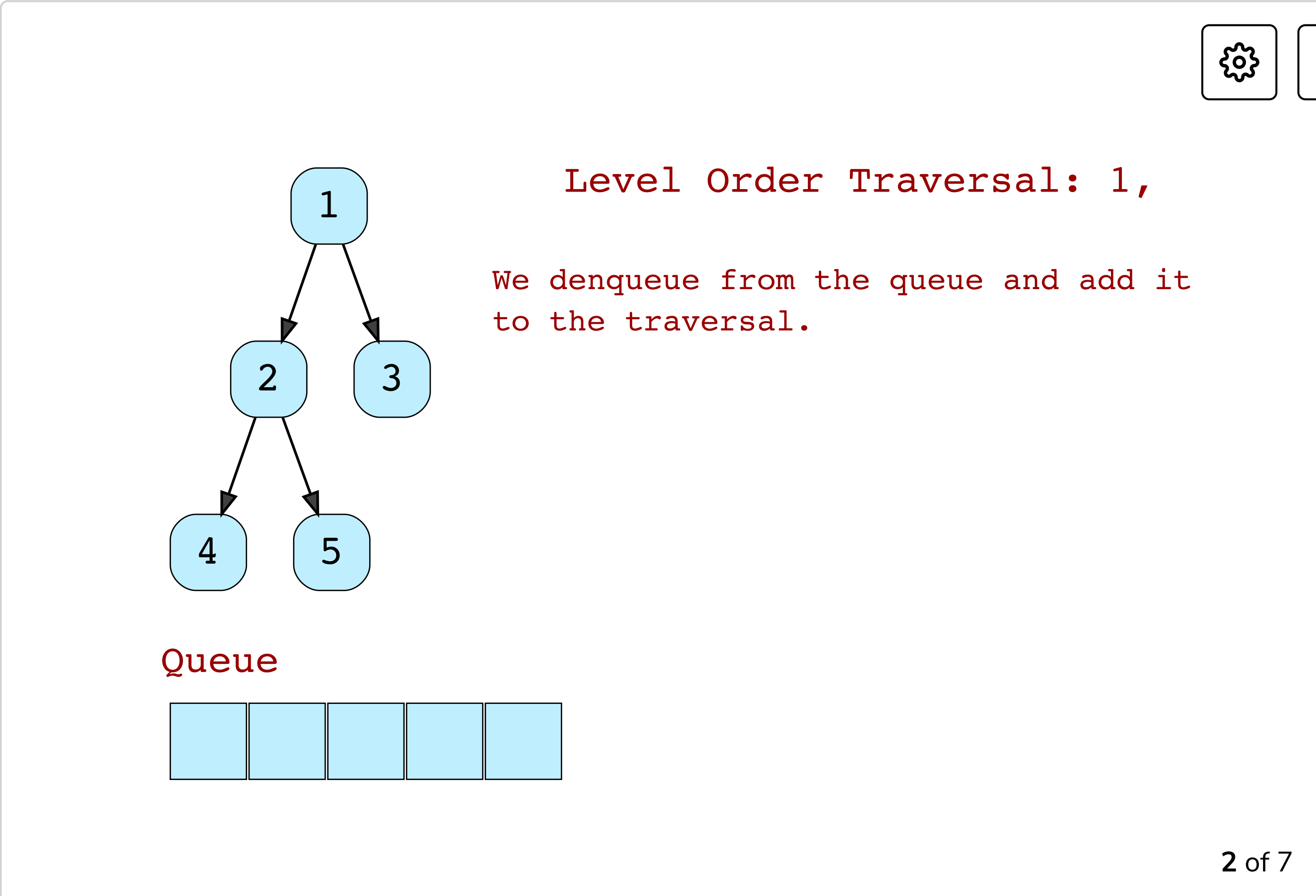


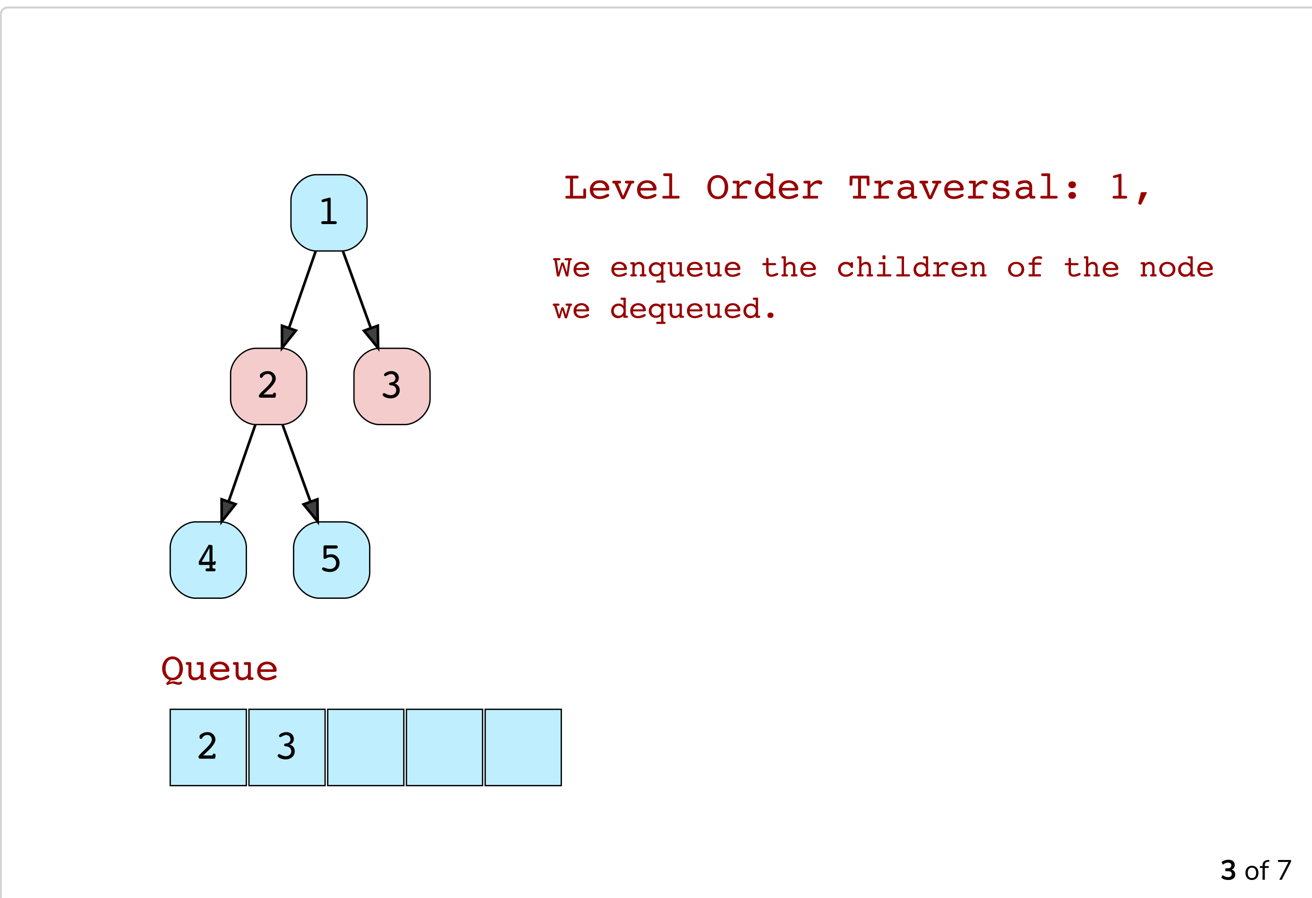
3 of 3

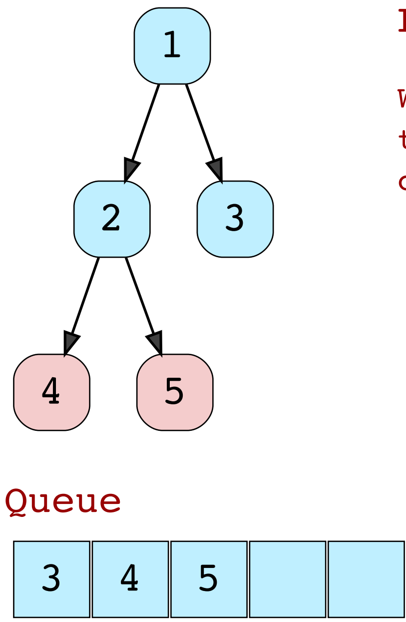
Algorithm

To do a level-order traversal of a binary tree, we require a queue. Have a look at the slides below for the algorithm:




Level Order Traversal: 1,
We denqueue from the queue and add it to the traversal.
1
2
3
4
5
Queue
2 of 7

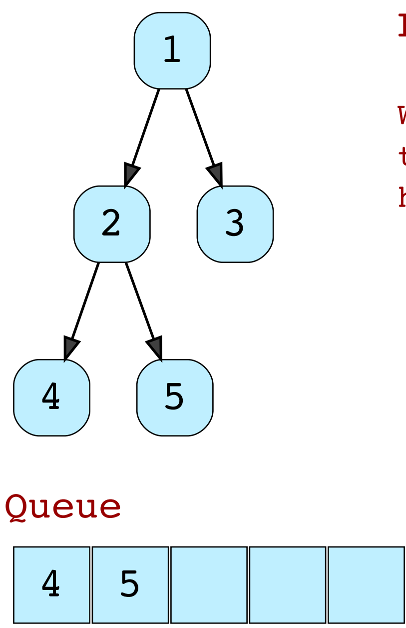

Level Order Traversal: 1,
We enqueue the children of the node we dequeued.
1
2
3
4
5
Queue
2
3
3 of 7



Level Order Traversal: 1,2

We dequeue 2, add it to the traversal and enqueue its children.

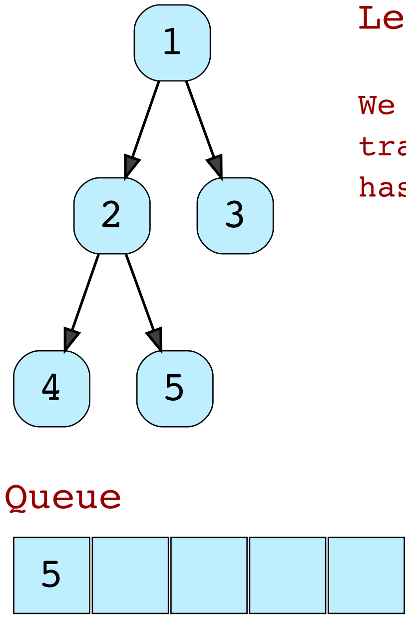
4 of 7



Level Order Traversal: 1,2,3

We dequeue 3, add it to the traversal and enqueue nothing as 3 has no children.

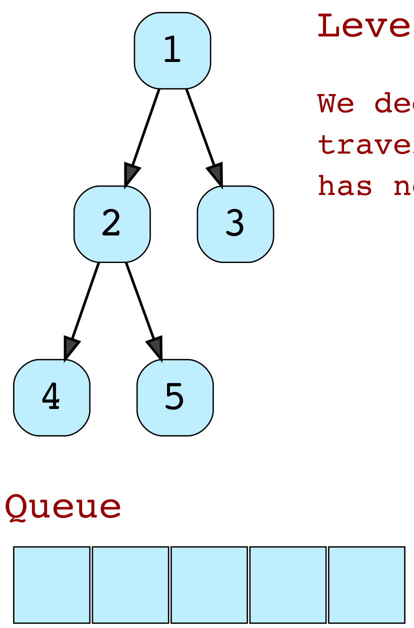
5 of 7



Level Order Traversal: 1,2,3,4

We dequeue 4, add it to the traversal and enqueue nothing as 4 has no children.

6 of 7



Level Order Traversal: 1,2,3,4,5

We dequeue 5, add it to the traversal and enqueue nothing as 5 has no children.

7 of 7

Implementation

Now that you are familiar with the algorithm, let's jump to the implementation in Python. First, we'll need to implement `Queue` so that we can use its object in our solution of level-order traversal.

```
1  class Queue(object):
2    def __init__(self):
3      self.items = []
4
5    def enqueue(self, item):
6      self.items.insert(0, item)
7
8    def dequeue(self):
9      if not self.is_empty():
10       return self.items.pop()
11
12   def is_empty(self):
13     return len(self.items) == 0
14
15   def peek(self):
16     if not self.is_empty():
17       return self.items[-1].value
18
19   def __len__(self):
20     return self.size()
21
22   def size(self):
23     return len(self.items)
```

class Queue

The constructor of the `Queue` class initializes `self.items` to an empty list on **line 3**. This list will store all the elements in the queue. We assume the last element to be the *front* of the queue and the first element to be the *back* of the queue.

To perform the enqueue operation, in the `enqueue` method, we make use of the `insert` method of Python list which will insert `item` on the `0` th index in `self.items` as specified on **line 6**. On the other hand, in the `dequeue` method, we use the `pop` method of Python list to pop out the last element as the queue follows the *First-In, First-Out* property. The method also ensures that the `pop` method is only called if the queue is not empty. To see if a queue is empty or not, the `is_empty` method comes in handy which checks for the length of `self.items` and compares it with `0`. If the length of `self.items` is `0`, `True` is returned, otherwise, `False` is returned.

The `peek` method will return the value of the last element in `self.items` which we assume to be the front of our queue. We have also overridden the `len` method on **line 19** which calls the `size` method on **line 22**. The size method returns the length of `self.items`.

Now that we have successfully implemented the `Queue` class, let's go ahead and implement level-order traversal:

```
def levelorder_print(self, start):
  if start is None:
    return

  queue = Queue()
  queue.enqueue(start)

  traversal = ""
  while len(queue) > 0:
    traversal += str(queue.peek()) + "-"
    node = queue.dequeue()

    if node.left:
      queue.enqueue(node.left)
    if node.right:
      queue.enqueue(node.right)

  return traversal
```

levelorder_print(self, start)

In the code above, first of all, we handle an edge case on **line 2**, i.e., `start` (root node) is `None` or we have an empty tree. In such a case, we return from the `levelorder_print` method.

On **line 5**, we initialize a `Queue` object from the class we just implemented and name it as `queue` to which we enqueue `start` on **line 6** as described in the algorithm. `traversal` is initialized to an empty string on **line 8**. Next, we set up a `while` loop on **line 9** which runs until the length of the queue is greater than `0`. Just as depicted in the algorithm, we append an element using the `peek` method to `traversal` and also concatenate a `-` so that the traversal appears in a format where the visited nodes will be divided by `-`. Once traversal is updated to register the node we visit, we dequeue that node and save it in the variable `node` on **line 11**. From **lines 13-16**, we check for the left and the right children of `node` and enqueue them to `queue` if they exist.

Finally, we return `traversal` on **line 18** which will have all the nodes we visited according to level-order.

In the code widget below, we have added `levelorder_print` to `BinaryTree` class and have also added `"levelorder"` as a `traversal_type` to `print_tree method.`

```
    def levelorder_print(self, start):
        if start is None:
            return

        queue = Queue()
        queue.enqueue(start)

        traversal = ""
        while len(queue) > 0:
            traversal += str(queue.peek()) + "-"
            node = queue.dequeue()

            if node.left:
```

```
                    queue.enqueue(node.left)
                if node.right:
                    queue.enqueue(node.right)

        return traversal


tree = BinaryTree(1)
tree.root.left = Node(2)
tree.root.right = Node(3)
tree.root.left.left = Node(4)
tree.root.left.right = Node(5)

print(tree.print_tree("levelorder"))
```

Output 0.19s

```
1-2-3-4-5-
```

I hope level-order traversal is clear to you! In the next lesson, we will cover reverse level-order traversal. Stay tuned!

Back
Traversal Algorithms

Next
Reverse Level-Order Traversal

Mark as Completed

Report an Issue

Ask a Question
(https://discuss.educative.io/tag/level-order-traversal__binary-trees__data-structures-and-algorithms-in-python)

Reverse Level-Order Traversal

In this lesson, you will learn how to implement reverse level-order traversal of a binary tree in Python.

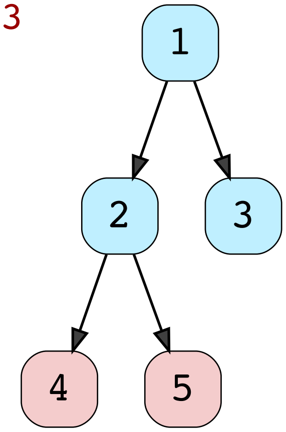
We'll cover the following

- Algorithm
- Implementation

This lesson will be an extension of the previous lesson. In this lesson, we will go over how to perform a reverse level-order traversal in a binary tree. We then code a solution in Python building on our binary tree class.

Below is an example of reverse level-order traversal of a binary tree:

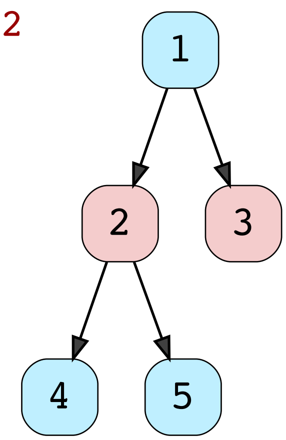
Nodes on Level 3



Reverse Level-Order Traversal: 4, 5

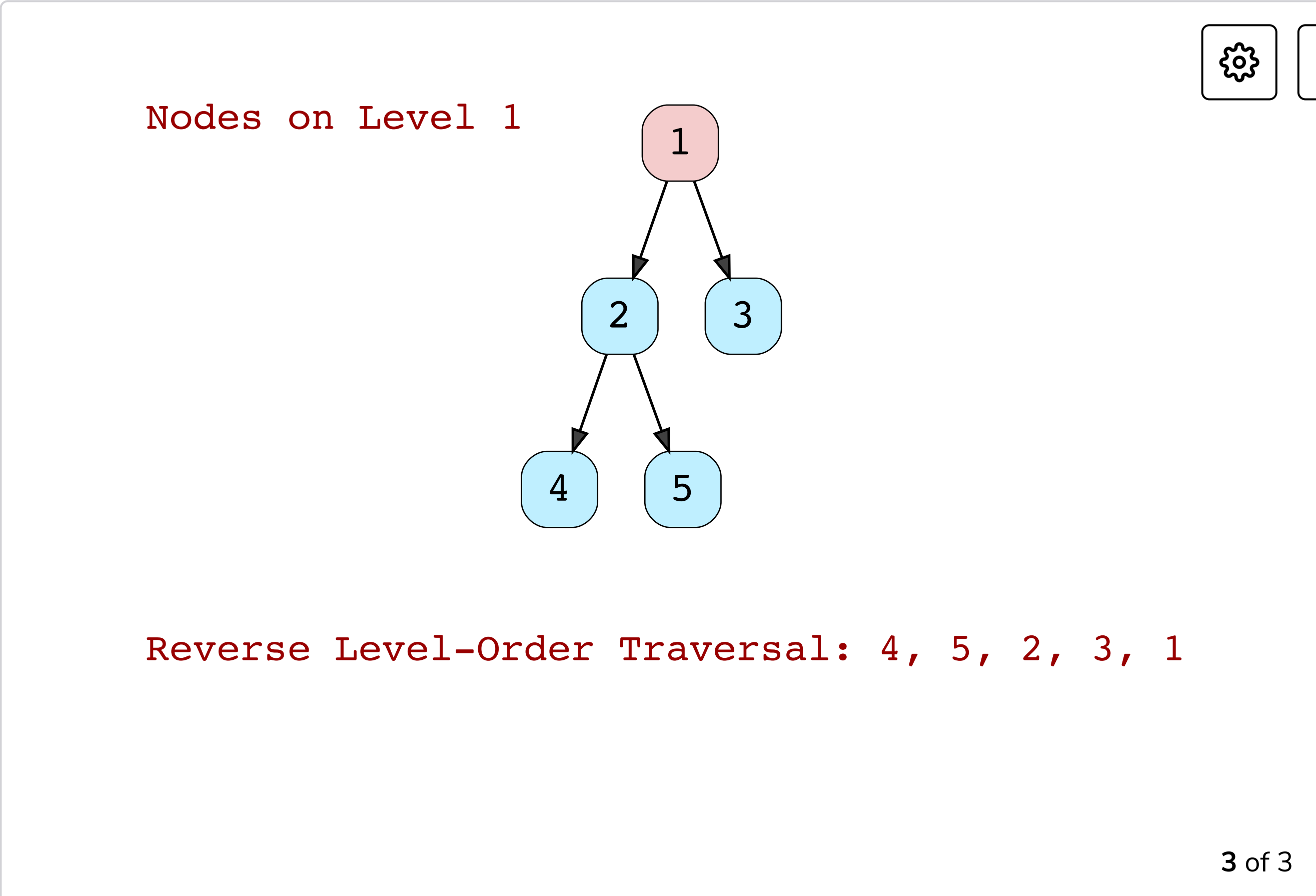
1 of 3

Nodes on Level 2



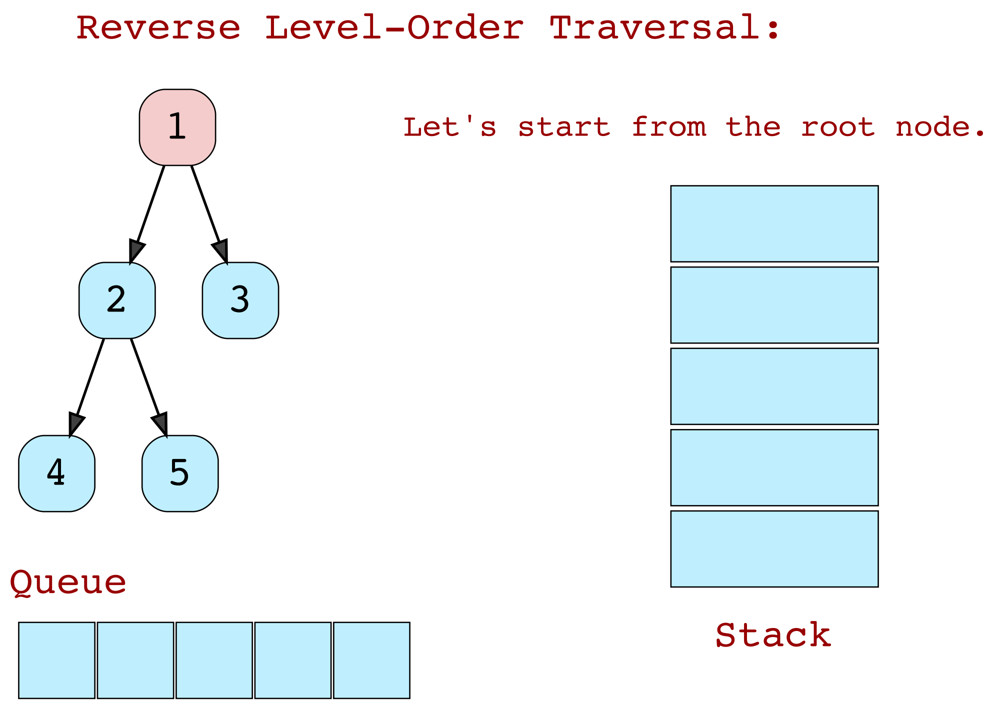
Reverse Level-Order Traversal: 4, 5, 2, 3

2 of 3



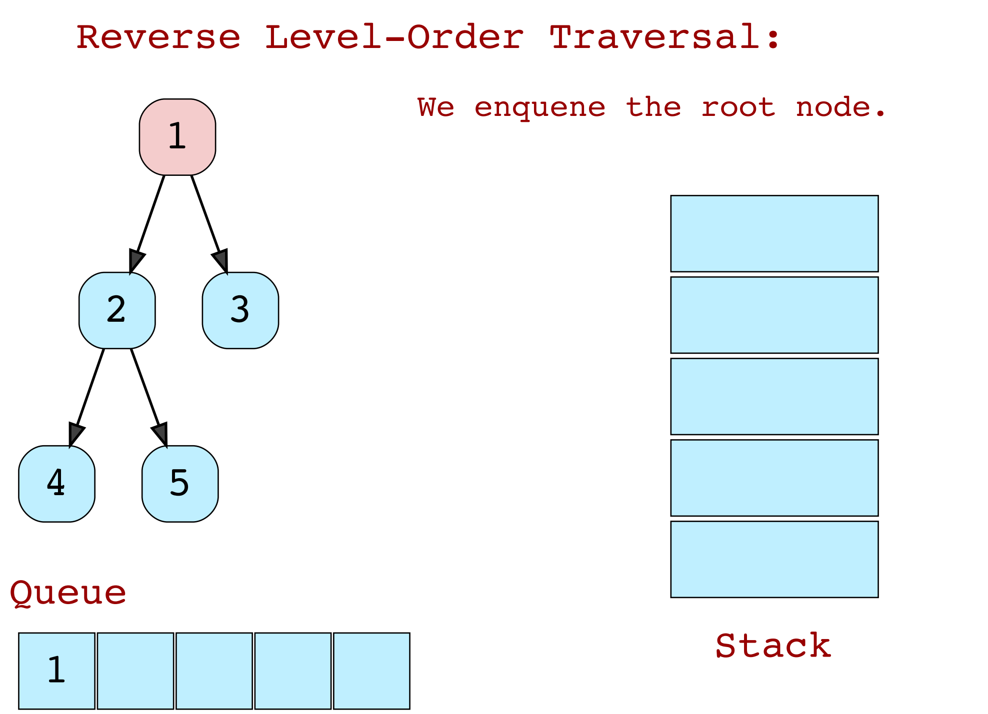
Algorithm

To solve this problem, we'll use a queue again just like we did with level-order traversal, but with a slight tweak; we'll enqueue the right child before the left child. Additionally, we will use a stack. The algorithm starts with enqueuing the root node. As we traverse the tree, we dequeue the nodes from the queue and push them to the stack. After we push a node on to the stack, we check for its children, and if they are present, we enqueue them. This process is repeated until the queue becomes empty. In the end, popping the element from the stack will give us the reverse-order traversal. Let's step through the algorithm using the illustrations below:



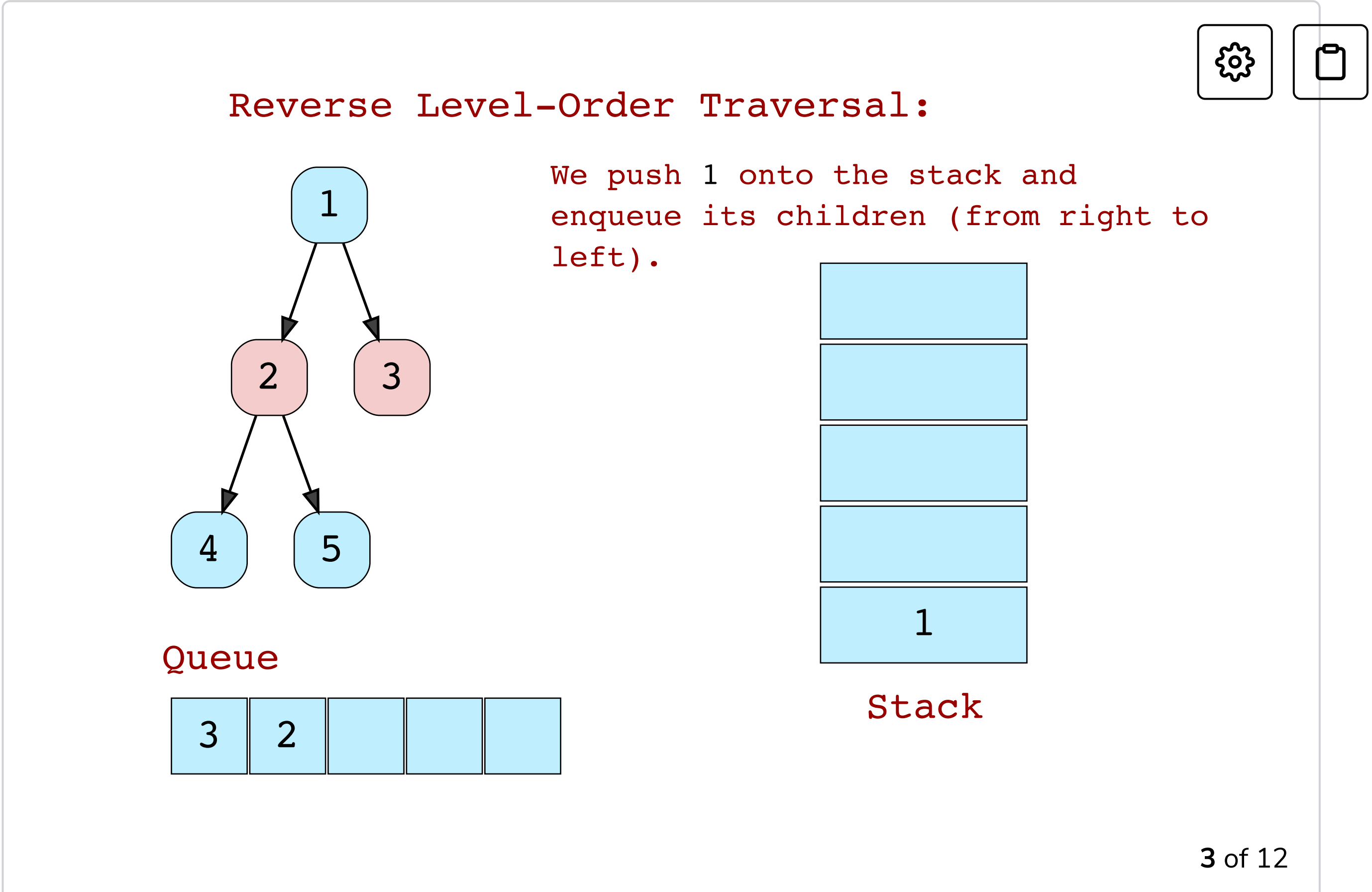
Reverse Level-Order Traversal:
Let's start from the root node.
1
2
3
4
5
Queue
Stack

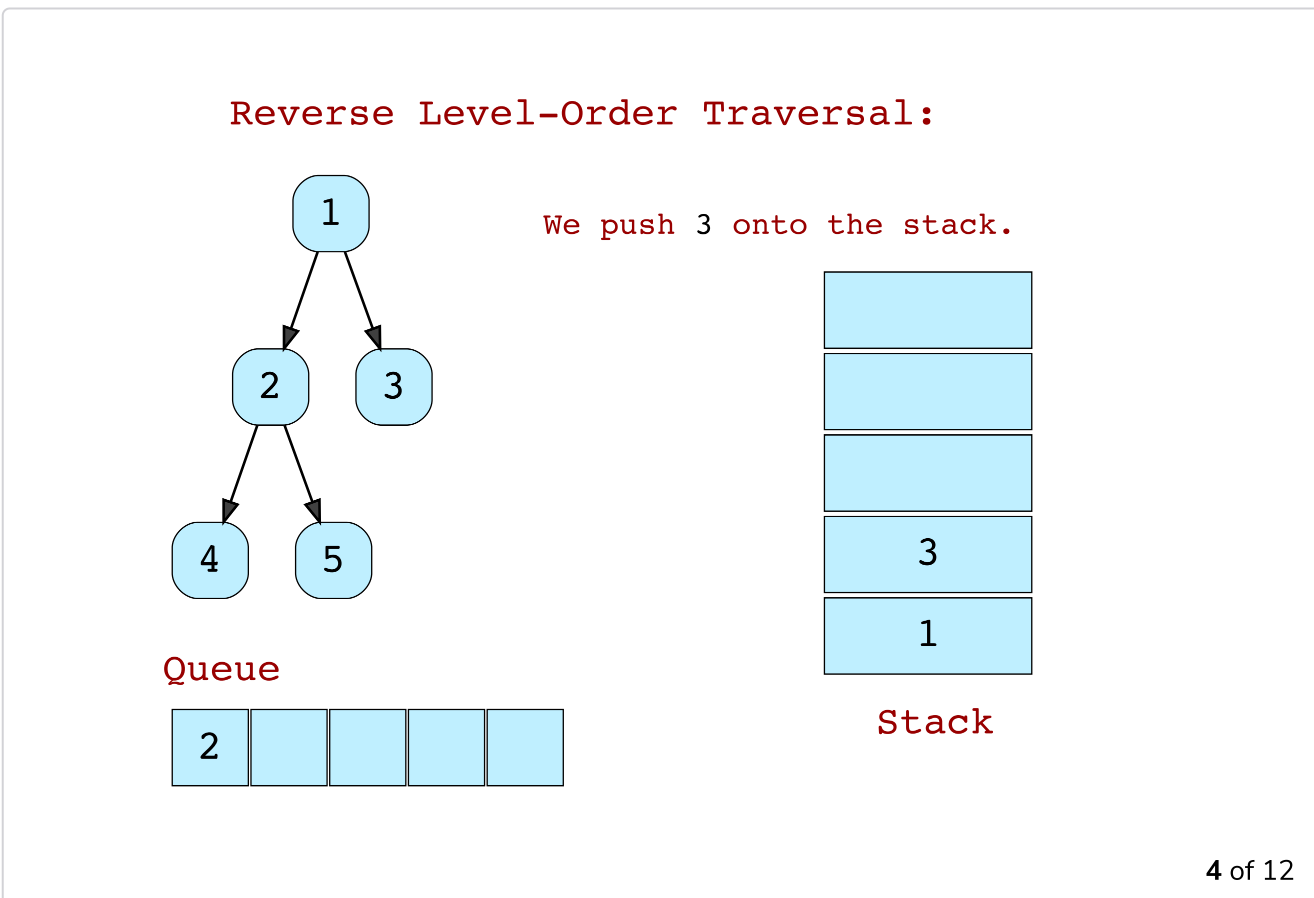
1 of 12

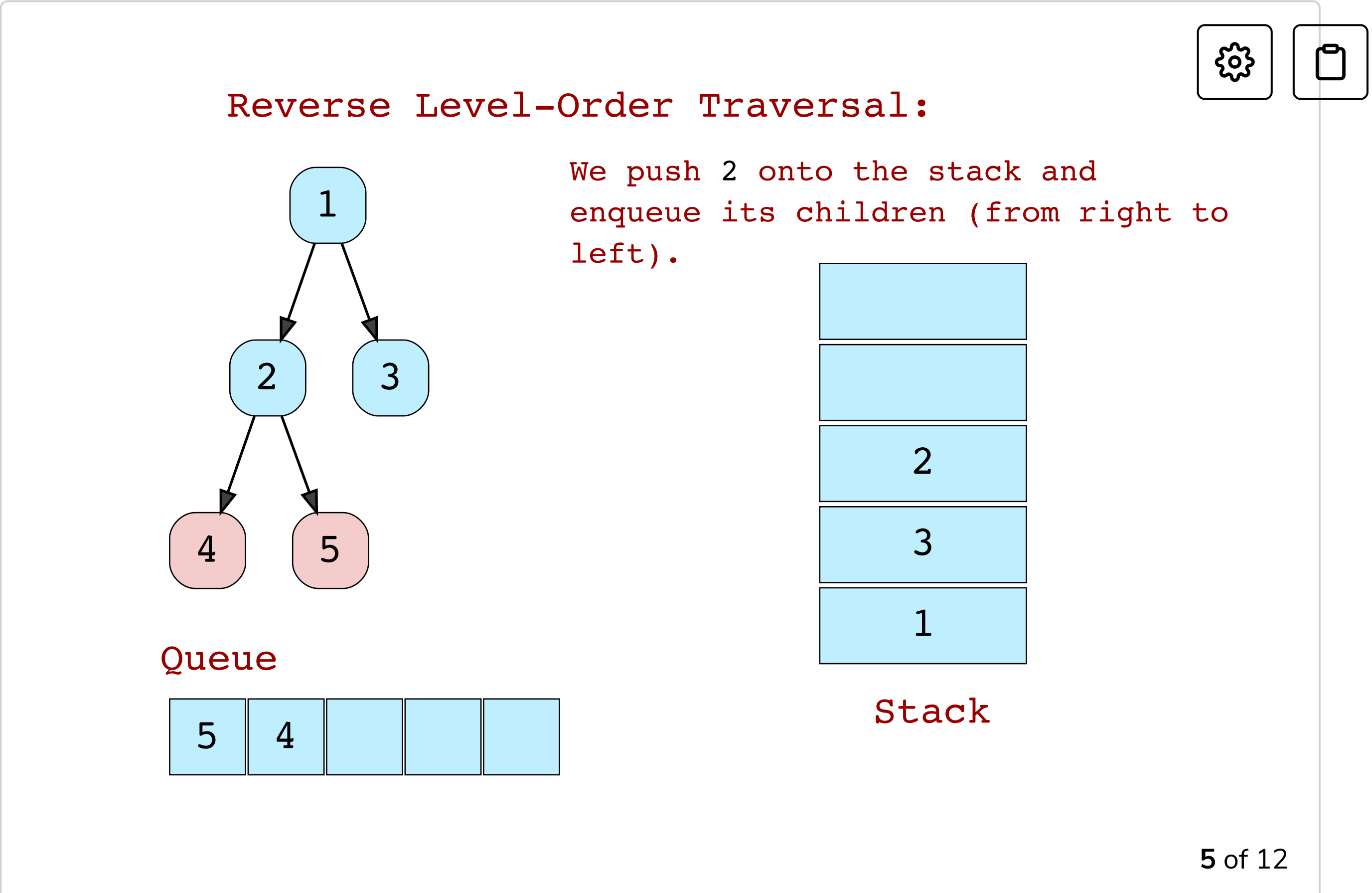


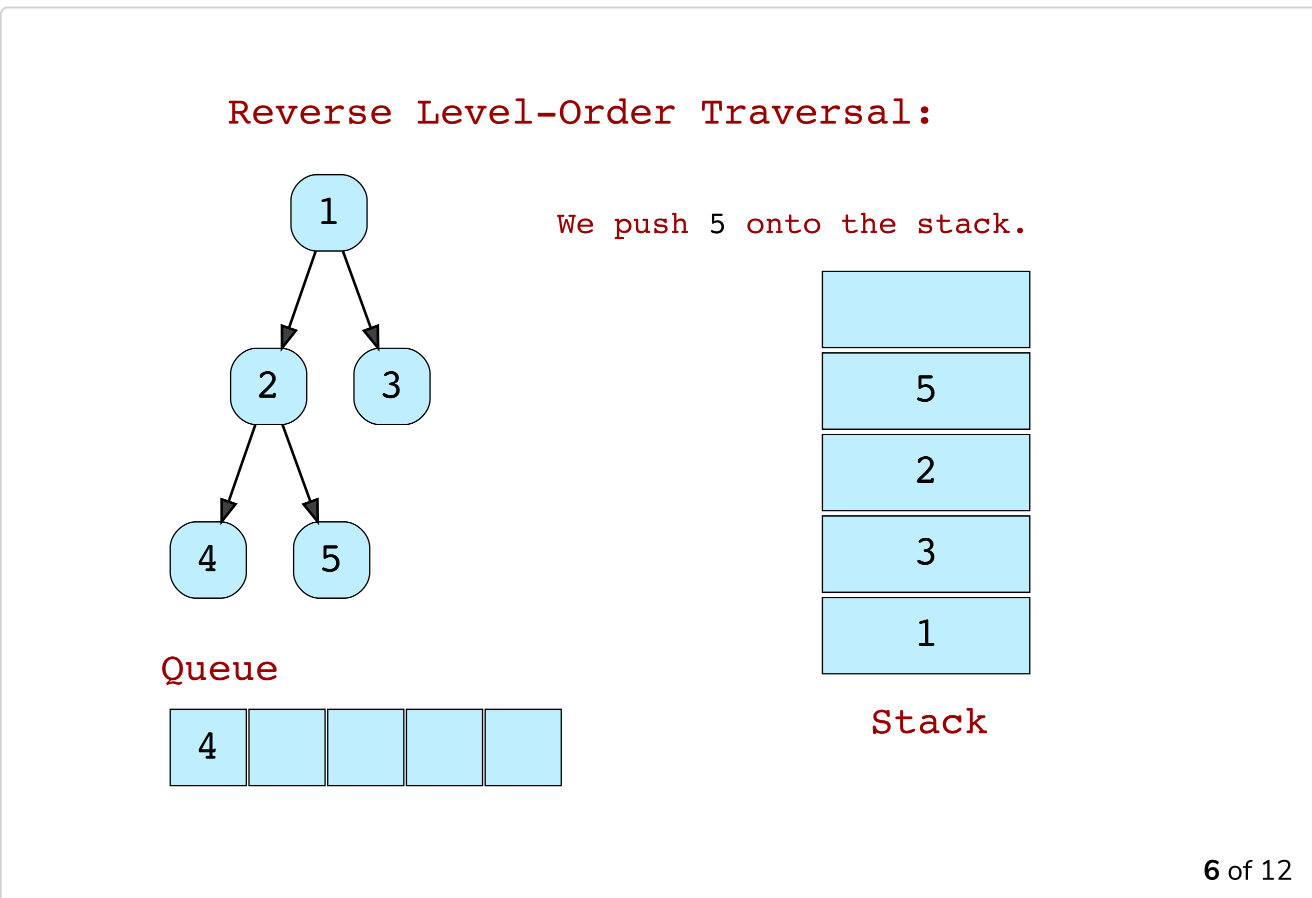
Reverse Level-Order Traversal:
We enquene the root node.
1
2
3
4
5
Queue
1
Stack

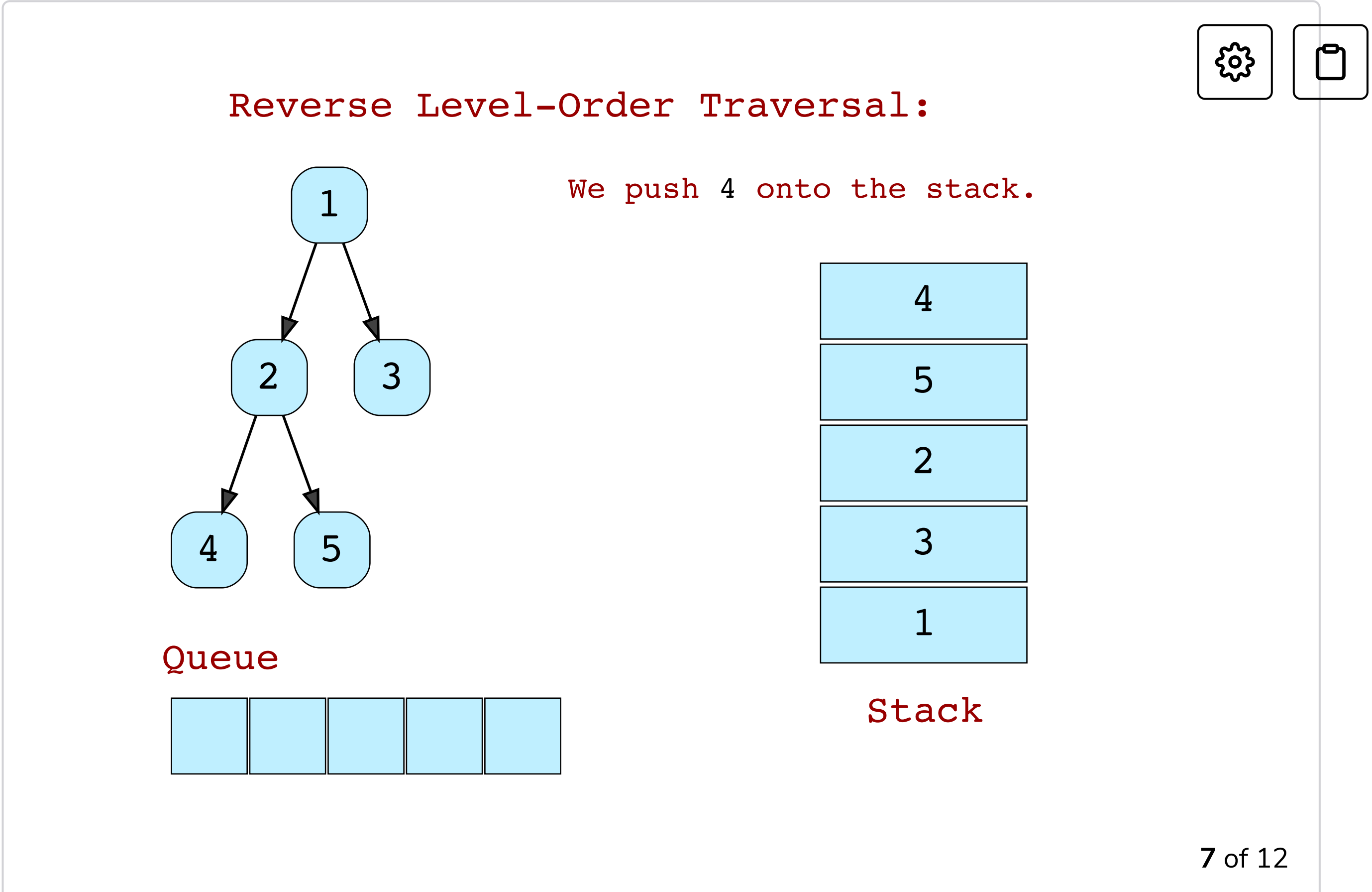
2 of 12

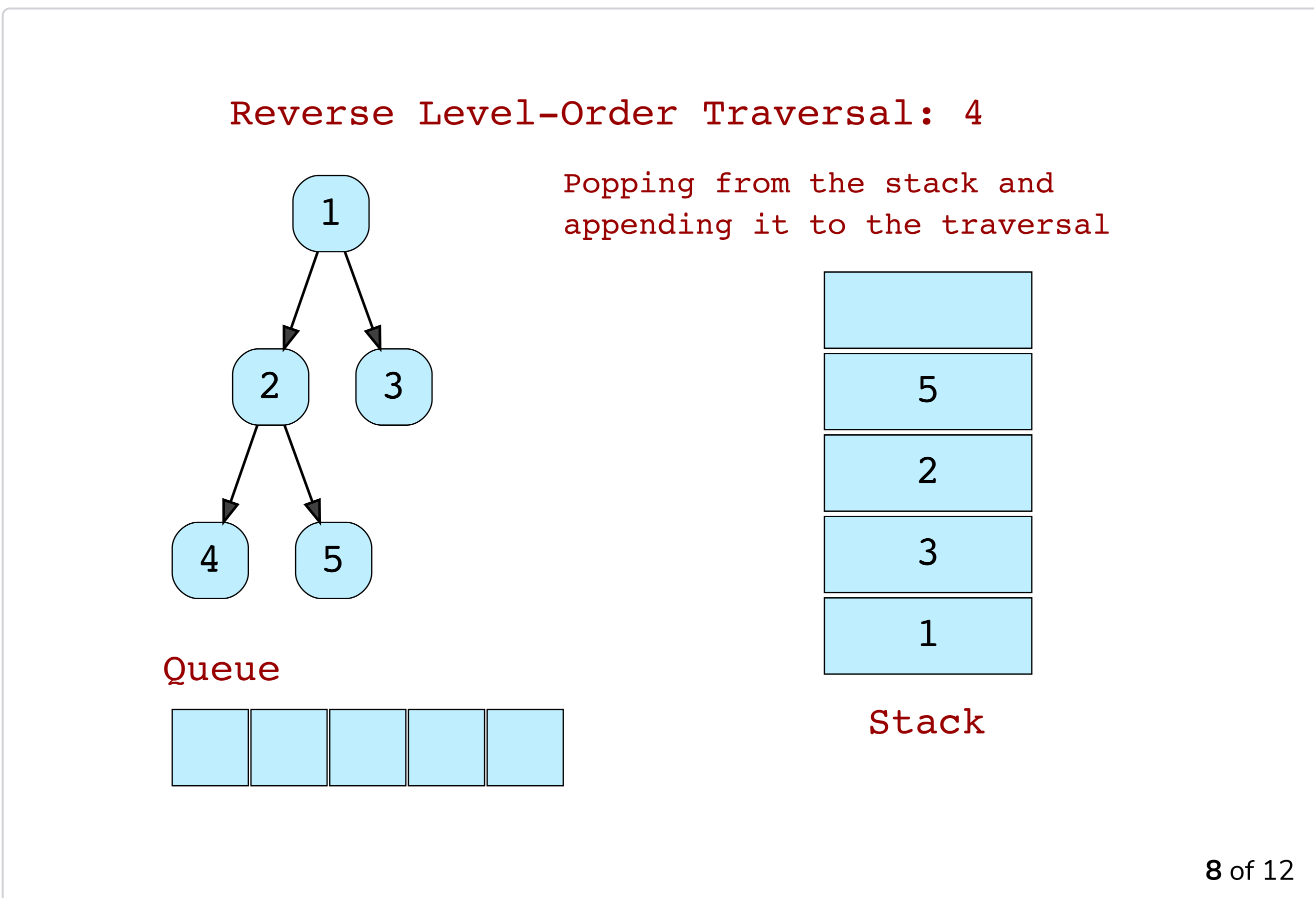

Reverse Level-Order Traversal:
We push 1 onto the stack and enqueue its children (from right to left).
1
2
3
4
5
Queue
3
2
1
Stack
3 of 12

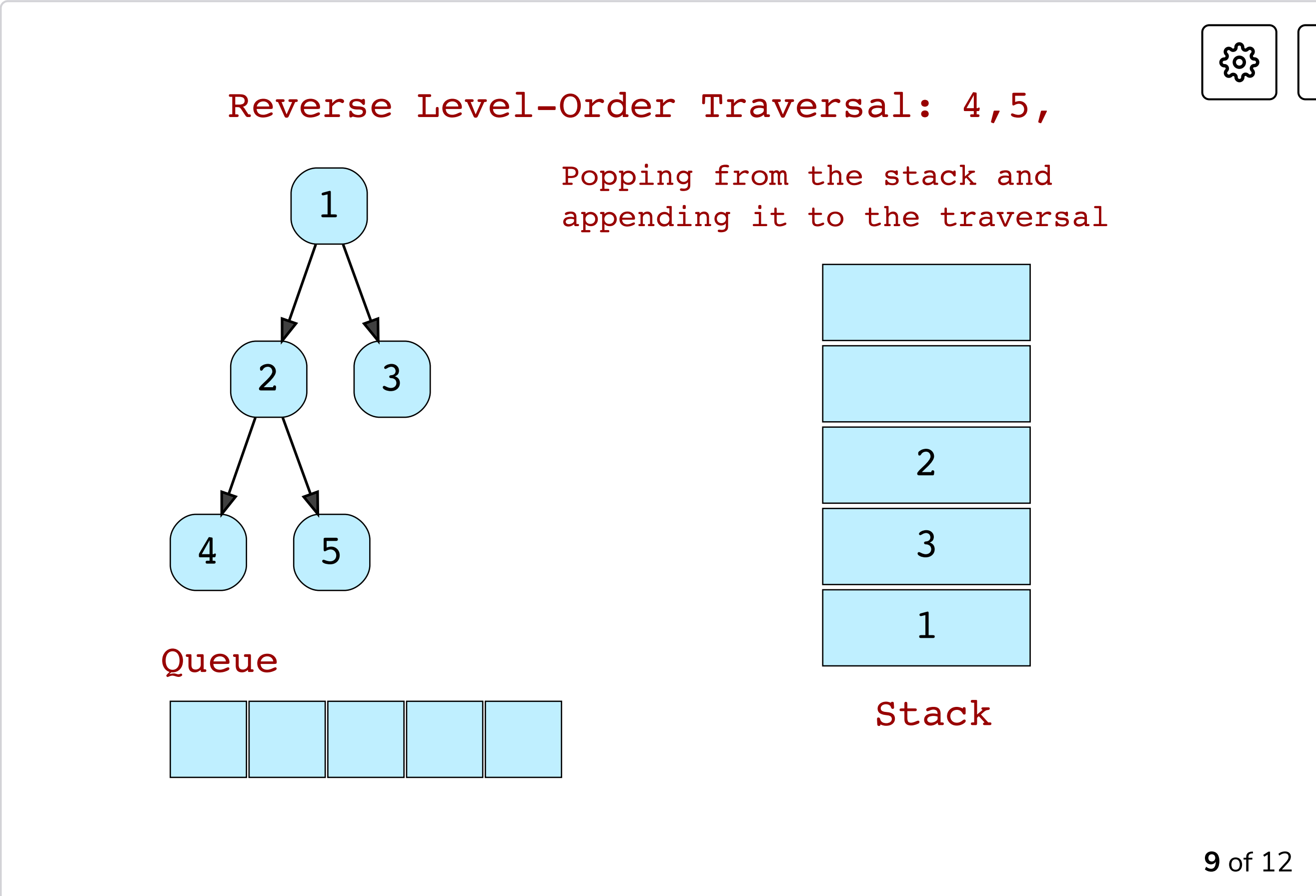

Reverse Level-Order Traversal:
We push 3 onto the stack.
1
2
3
4
5
Queue
2
3
1
Stack
4 of 12

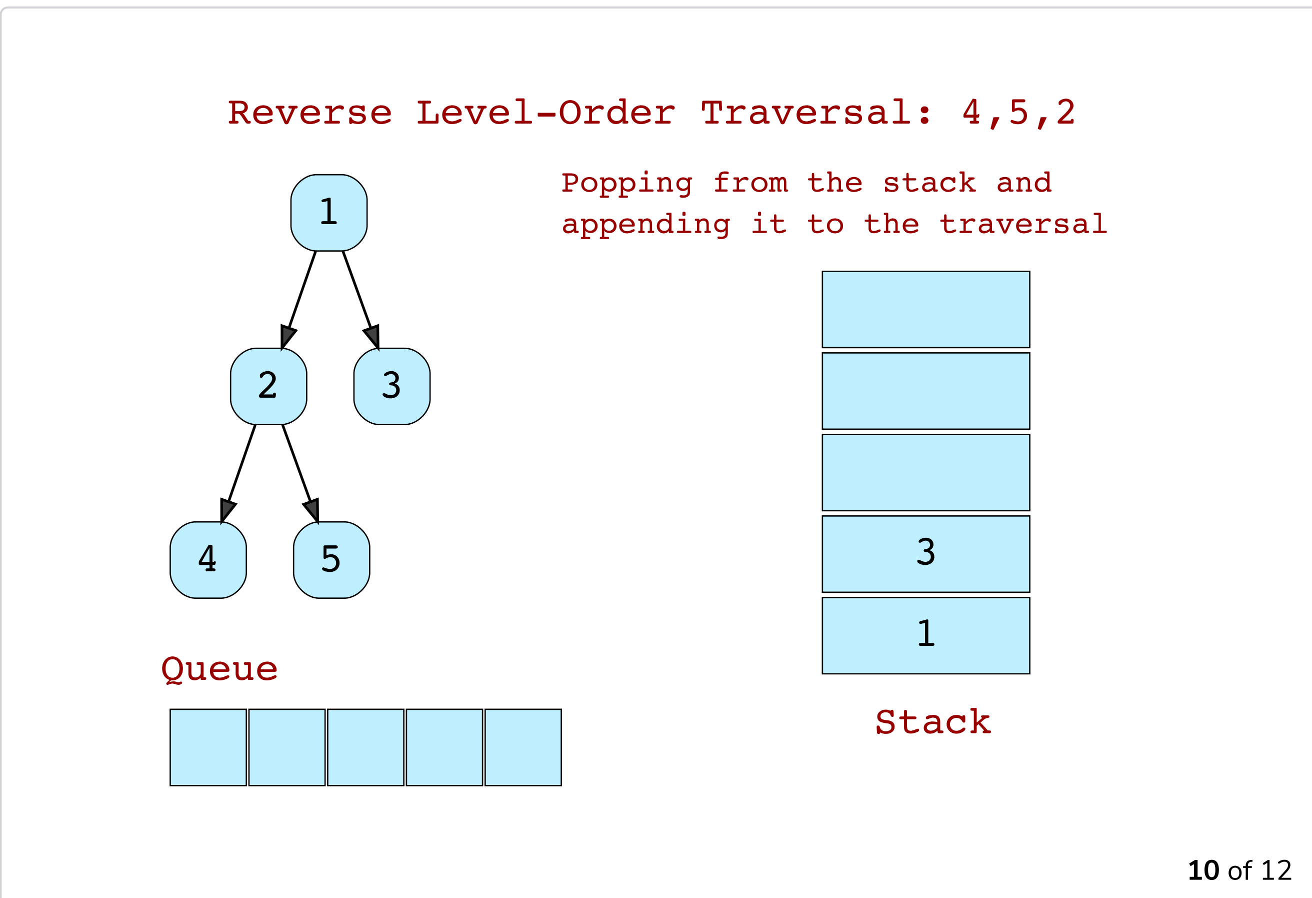

Reverse Level-Order Traversal:
We push 2 onto the stack and enqueue its children (from right to left).
1
2
3
4
5
Queue
5
4
2
3
1
Stack
5 of 12

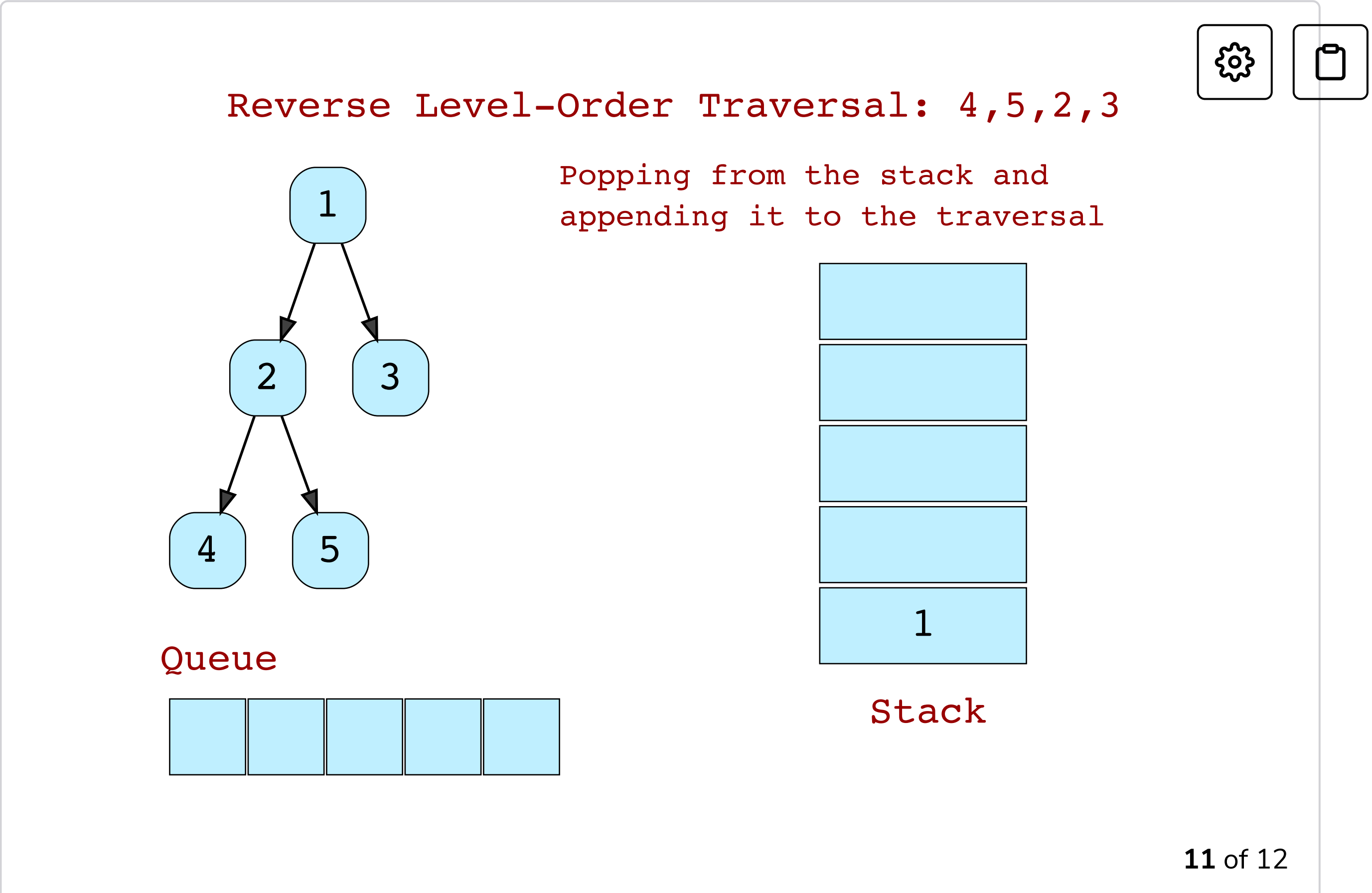

Reverse Level-Order Traversal:
We push 5 onto the stack.
1
2
3
4
5
Queue
4
5
2
3
1
Stack
6 of 12


Reverse Level-Order Traversal:
We push 4 onto the stack.
1
2
3
4
5
Queue
4
5
2
3
1
Stack
7 of 12

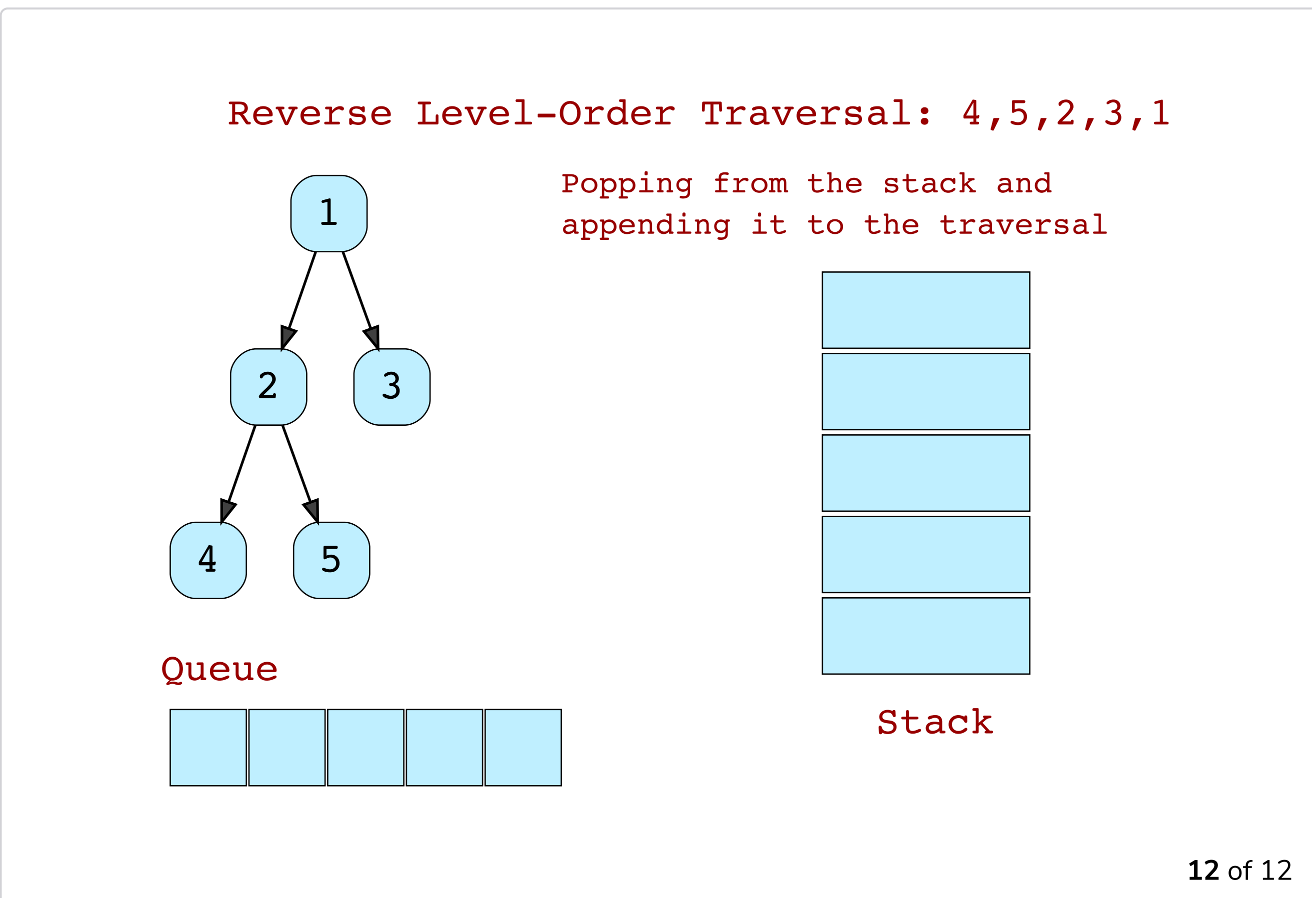

Reverse Level-Order Traversal: 4
Popping from the stack and appending it to the traversal
1
2
3
4
5
Queue
5
2
3
1
Stack
8 of 12


Reverse Level-Order Traversal: 4,5,
Popping from the stack and
appending it to the traversal
1
2
3
4
5
Queue
2
3
1
Stack
9 of 12


Reverse Level-Order Traversal: 4,5,2
Popping from the stack and
appending it to the traversal
1
2
3
4
5
Queue
3
1
Stack
10 of 12



Reverse Level-Order Traversal: 4,5,2,3
Popping from the stack and appending it to the traversal
1
2
3
4
5
Queue
1
Stack
11 of 12



Reverse Level-Order Traversal: 4,5,2,3,1
Popping from the stack and appending it to the traversal
1
2
3
4
5
Queue
Stack
12 of 12

Implementation

Now that we have studied the algorithm, let's jump to the implementation in Python. First, we'll implement `Stack` class:

```
2     def __init__(self):
3         self.items = []
4 
5     def __len__(self):
6         return self.size()
7      
8     def size(self):
9         return len(self.items)
10 
11     def push(self, item):
12         self.items.append(item)
13 
14     def pop(self):  
15         if not self.is_empty():
16             return self.items.pop()
17 
18     def peek(self):
19         if not self.is_empty():
20             return self.items[-1]
21 
22     def is_empty(self):
23         return len(self.items) == 0
24 
25     def __str__(self):
26         s = ""
27         for i in range(len(self.items)):
28             s += str(self.items[i].value) + "-"
29         return s
```

class Stack

In the constructor, we initialize `self.items` to an empty list just like we did with `Queue`. In the `push` method on **line 11**, the built-in `append` method is used to insert elements (`item`) to `self.items`. So whenever we push an element onto the stack, we append that element to `self.items`. The `pop` method on **line 14** first checks whether the stack is empty or not using the `is_empty` method implemented on **line 22**. In the `pop` method, we use `pop` method of Python list to pop out the last element as the stack follows the *First-In, Last-Out* property and the latest element we inserted is at the end of `self.items`. The `is_empty` method on **line 22** checks for the length of `self.items` by comparing it with `0` and returns the boolean value accordingly. On **line 18**, `peek` method is implemented which may or may not be used in the solution of our lesson problem. If the stack is not empty, the last element of `self.items` is returned on **line 20**.

The `size` and `len` method have also been added in a way as to the `Queue` class. Also, we have an `str` method on **line 25** which iterates through `self.items` and concatenates them into a string which is returned from the method.

Now that we are done with the implementation of `Stack` class, let's discuss `reverse_levelorder_print`:

```
def reverse_levelorder_print(self, start):
  if start is None:
    return

  queue = Queue()
  stack = Stack()
  queue.enqueue(start)


  traversal = ""
  while len(queue) > 0:
    node = queue.dequeue()

    stack.push(node)

    if node.right:
      queue.enqueue(node.right)
    if node.left:
      queue.enqueue(node.left)

  while len(stack) > 0:
    node = stack.pop()
    traversal += str(node.value) + "-"
```

```
24
25      return traversal
```

reverse_levelorder_print(self, start)

In the code above, we handle an edge case on **line 2**, i.e., the `start` (root node) is `None` or we would have an empty tree. In such a case, we return from the `reverse_levelorder_print` method.

On **line 5** and **line 6**, we initialize a `Queue` object and a `Stack` object from the class we just implemented. In the next line, we enqueue `start` to `queue` as described in the algorithm. `traversal` is initialized to an empty string on **line 10**. Next, we set up a `while` loop on **line 11** which runs until the length of the queue is greater than `0`. Just as depicted in the algorithm, we dequeue an element from the queue and push it on the stack on **line 12**. From **lines 16-19**, we check for the right and left children of the `node` and enqueue them to `queue` if they exist. At the end of the `while` loop, `stack` will contain all the nodes of the tree. On **line 21**, we are using a `while` loop to pop elements from the stack and concatenate them to `traversal` which is returned from the method on **line 25**.

In the code widget below, we have added `reverse_levelorder_print` to the `BinaryTree` class and have also added `"reverse_levelorder"` as a `traversal_type to print_tree method.`

```
        traversal = ""
        while len(queue) > 0:
            node = queue.dequeue()

            stack.push(node)

            if node.right:
                queue.enqueue(node.right)
            if node.left:
                queue.enqueue(node.left)

```

```
        while len(stack) > 0:
            node = stack.pop()
            traversal += str(node.value) + "-"

        return traversal



tree = BinaryTree(1)
tree.root.left = Node(2)
tree.root.right = Node(3)
tree.root.left.left = Node(4)
tree.root.left.right = Node(5)

print(tree.print_tree("reverse_levelorder"))
```

Output 0.15s

```
4-5-2-3-1-
```

In the next lesson, we will learn how to calculate the height of a binary tree in Python.

Calculating the Height of a Binary Tree

In this lesson, you will learn how to calculate the height of a binary tree.

We'll cover the following

Height of Tree

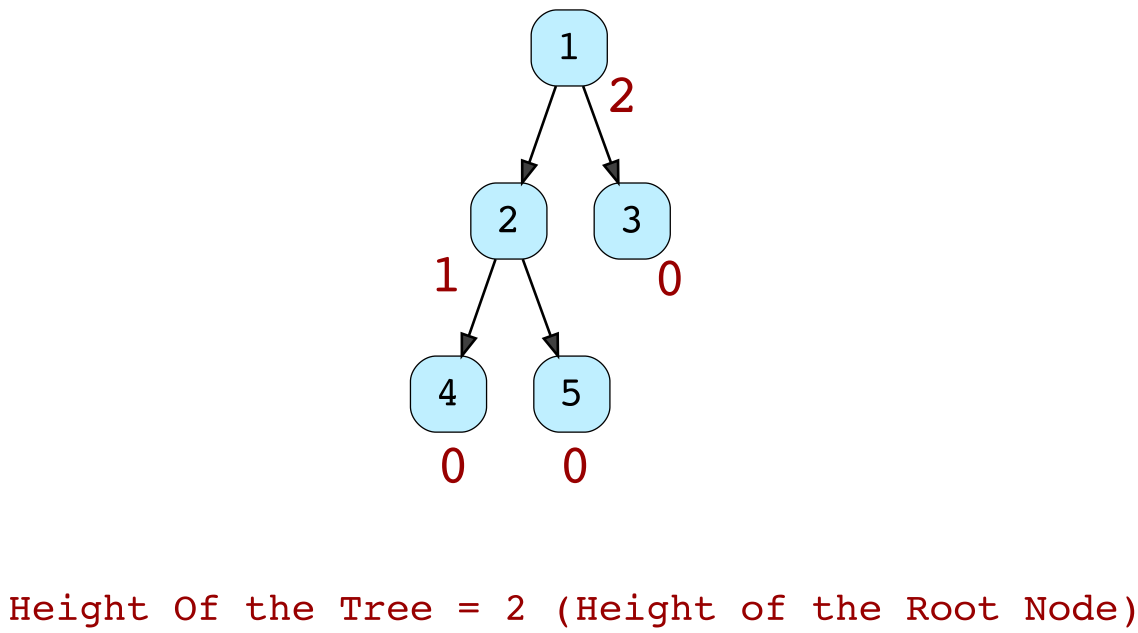
Let's start by defining the height of a tree. The height of a tree is the height of its root node. Now let's see what we mean by the height of a node:

Height of Node

The height of a node is the number of edges on the longest path between that node and a leaf. The height of a leaf node is 0.

Recursively defined, the height of a node is one greater than the max of its right and left children's height.

Below is an example of a binary tree labeled with heights of individual nodes:



Algorithm

In this lesson, we will consider the recursive approach to calculate the height of a tree. The idea is to break down the problem using recursion and traverse through the left and right subtree of a node to calculate the height of that node. Once we get the height of the left and right subtree, we will consider the maximum of the two heights plus one to be the height of the tree.

Implementation

Let’s move to the implementation in Python:

```
def height(self, node):
  if node is None:
    return -1
  left_height = self.height(node.left)
  right_height = self.height(node.right)

```

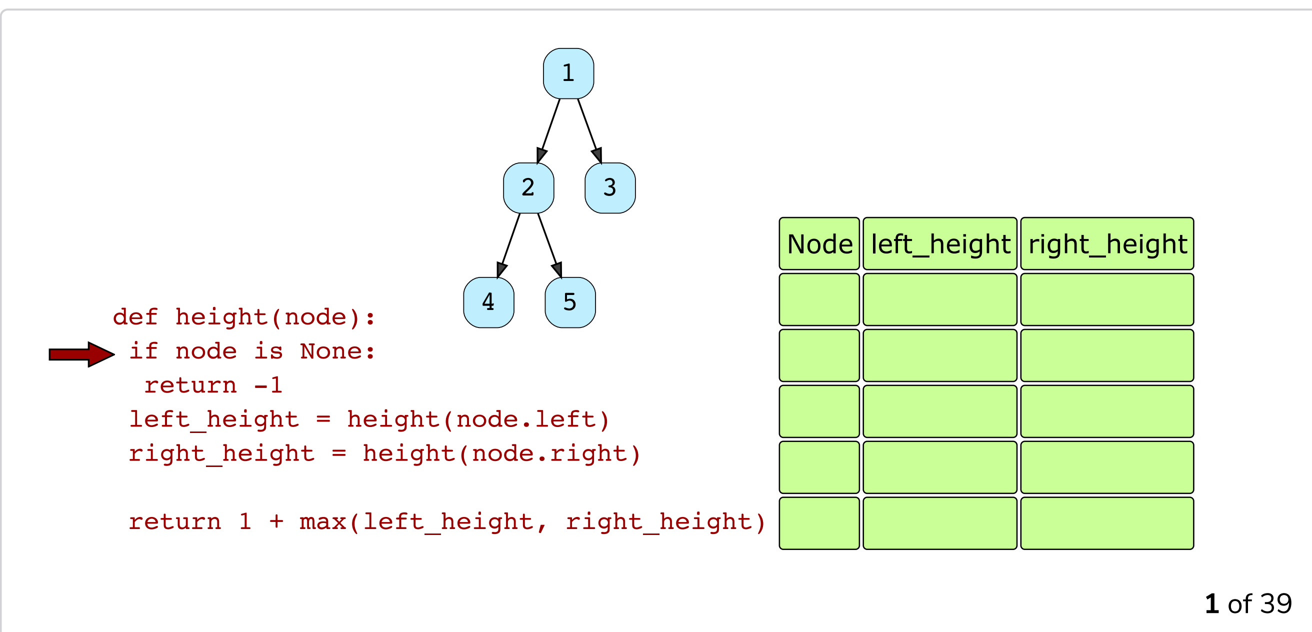
```
7    return 1 + max(left_height, right_height)
```

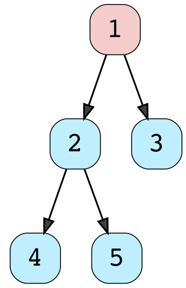
height(self, node)

In the code above, our base case is when `node` equals `None`. If `node` equals `None`, we return `-1` on **line 3** as we have gone past the leaf nodes. Once a leaf node discovers that its left and right children are reporting heights of `-1` each, it will add `1` to `-1` and return `0` as its height.

In **lines 4-5**, we recursively call the `height` method on the left child and the right child. The final height is calculated by adding `1` to the maximum height of the left and right subtree, as height is the longest path between that node and the leaf node. The final height is what is returned from the method.

Let's run this code step by step in the illustration below:

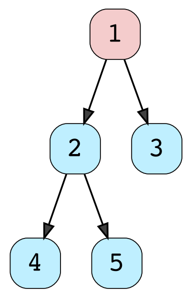





```
def height(node):
 if node is None:
  return -1
 left_height = height(node.left)
 right_height = height(node.right)

 return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1		

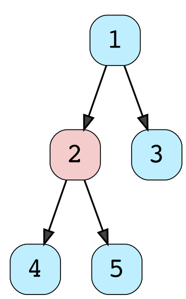
2 of 39




```
def height(node):
 if node is None:
  return -1
 left_height = height(node.left)
 right_height = height(node.right)

 return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1		

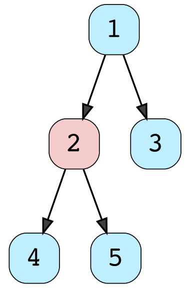
3 of 39




```
def height(node):
 if node is None:
  return -1
 left_height = height(node.left)
 right_height = height(node.right)

 return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1		
2		

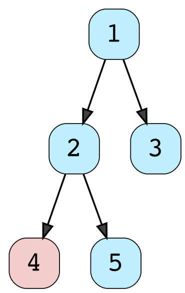
4 of 39




```
def height(node):
 if node is None:
  return -1
 left_height = height(node.left)
 right_height = height(node.right)

 return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1		
2		

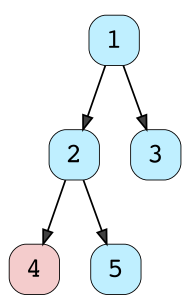
5 of 39




```
def height(node):
 if node is None:
  return -1
 left_height = height(node.left)
 right_height = height(node.right)

 return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1		
2		
4		

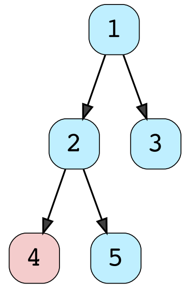
6 of 39




```
def height(node):
 if node is None:
  return -1
 left_height = height(node.left)
 right_height = height(node.right)

 return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1		
2		
4		

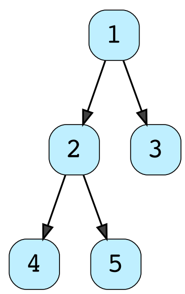
7 of 39




```
def height(node):
 if node is None:
  return -1
 left_height = height(node.left)
 right_height = height(node.right)

 return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1		
2		
4		

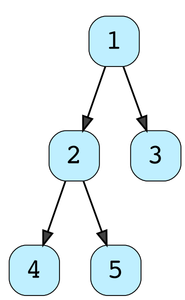
8 of 39




```
def height(node):
 if node is None:
  return -1
 left_height = height(node.left)
 right_height = height(node.right)

 return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1		
2		
4		
None		

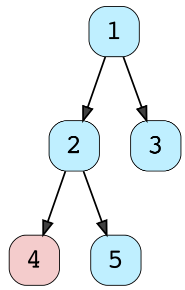
9 of 39




```
def height(node):
 if node is None:
  return -1
 left_height = height(node.left)
 right_height = height(node.right)

 return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1		
2		
4		
None		

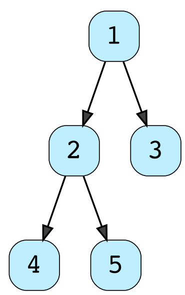
10 of 39



```
def height(node):
    if node is None:
        return -1
    left_height = height(node.left)
    right_height = height(node.right)

    return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1		
2		
4		
None	-1	

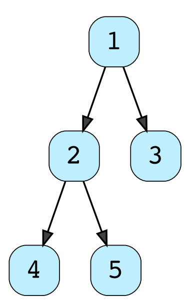
11 of 39



```
def height(node):
    if node is None:
        return -1
    left_height = height(node.left)
    right_height = height(node.right)

    return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1		
2		
4		
None	-1	

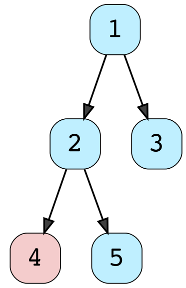
12 of 39



```
def height(node):
    if node is None:
        return -1
    left_height = height(node.left)
    right_height = height(node.right)

    return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1		
2		
4		
None	-1	-1

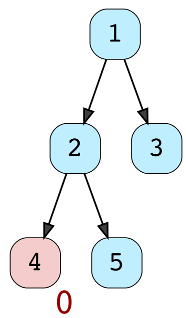
13 of 39




```
def height(node):
 if node is None:
  return -1
 left_height = height(node.left)
 right_height = height(node.right)

 return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1		
2		
4		
None	-1	-1

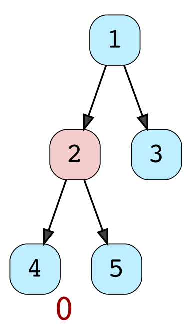
14 of 39




```
def height(node):
 if node is None:
  return -1
 left_height = height(node.left)
 right_height = height(node.right)

 return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1		
2		
4		

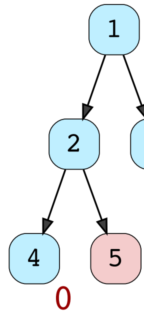
15 of 39




```
def height(node):
 if node is None:
  return -1
 left_height = height(node.left)
 right_height = height(node.right)

 return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1		
2		

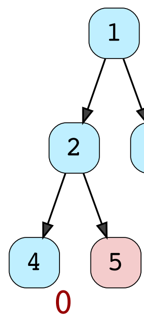
16 of 39




```
def height(node):
 if node is None:
  return -1
 left_height = height(node.left)
 right_height = height(node.right)

 return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1		
2		
5		

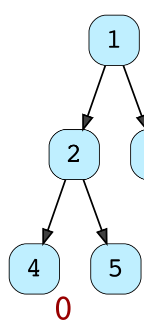
17 of 39




```
def height(node):
 if node is None:
  return -1
 left_height = height(node.left)
 right_height = height(node.right)

 return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1		
2		
5		

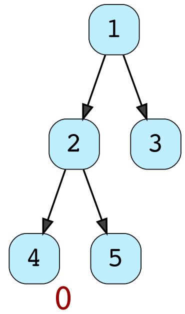
18 of 39




```
def height(node):
 if node is None:
  return -1
 left_height = height(node.left)
 right_height = height(node.right)

 return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1		
2		
5		
None		

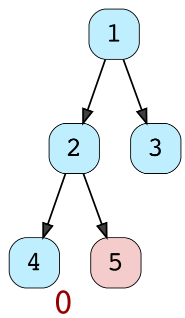
19 of 39




```
def height(node):
 if node is None:
  return -1
 left_height = height(node.left)
 right_height = height(node.right)

 return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1		
2		
5		
None	-1	

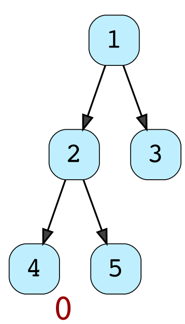
20 of 39




```
def height(node):
 if node is None:
  return -1
 left_height = height(node.left)
 right_height = height(node.right)

 return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1		
2		
5		
None	-1	

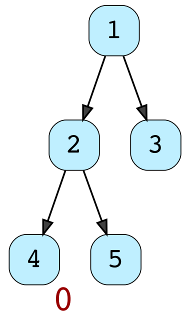
21 of 39




```
def height(node):
 if node is None:
  return -1
 left_height = height(node.left)
 right_height = height(node.right)

 return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1		
2		
5		
None	-1	

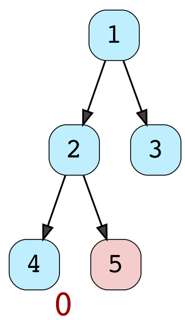
22 of 39




```
def height(node):
 if node is None:
  return -1
 left_height = height(node.left)
 right_height = height(node.right)

 return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1		
2		
5		
None	-1	-1

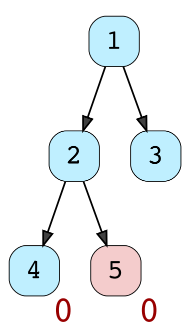
23 of 39




```
def height(node):
 if node is None:
  return -1
 left_height = height(node.left)
 right_height = height(node.right)

 return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1		
2		
5		
None	-1	-1

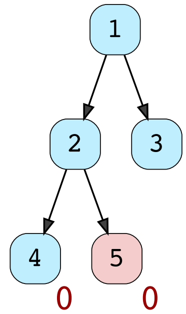
24 of 39




```
def height(node):
 if node is None:
  return -1
 left_height = height(node.left)
 right_height = height(node.right)

 return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1		
2		
5		

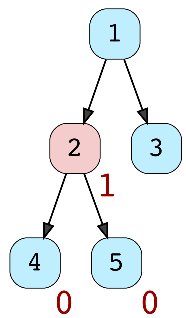
25 of 39



```
def height(node):
 if node is None:
  return -1
 left_height = height(node.left)
 right_height = height(node.right)

 return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1		
2		

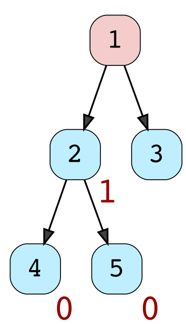
26 of 39



```
def height(node):
 if node is None:
  return -1
 left_height = height(node.left)
 right_height = height(node.right)

 return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1		
2		

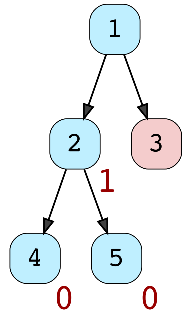
27 of 39



```
def height(node):
 if node is None:
  return -1
 left_height = height(node.left)
 right_height = height(node.right)

 return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1	1	

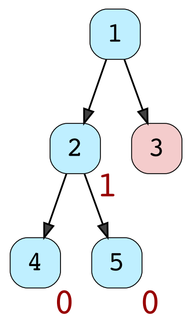
28 of 39




```
def height(node):
 if node is None:
  return -1
 left_height = height(node.left)
 right_height = height(node.right)

 return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1	1	
3		

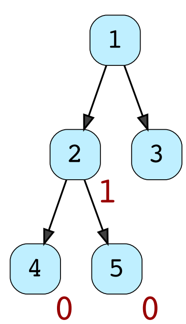
29 of 39




```
def height(node):
 if node is None:
  return -1
 left_height = height(node.left)
 right_height = height(node.right)

 return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1	1	
3		
None		

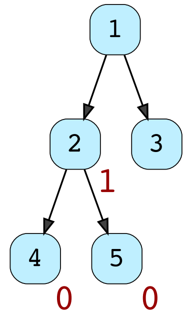
30 of 39




```
def height(node):
 if node is None:
  return -1
 left_height = height(node.left)
 right_height = height(node.right)

 return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1	1	
3		
None		

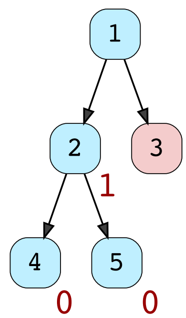
31 of 39




```
def height(node):
  if node is None:
    return -1
  left_height = height(node.left)
  right_height = height(node.right)

  return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1	1	
3		
None	-1	

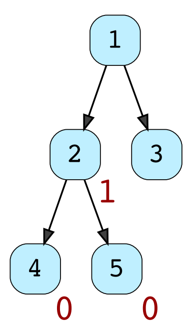
32 of 39




```
def height(node):
  if node is None:
    return -1
  left_height = height(node.left)
  right_height = height(node.right)

  return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1	1	
3		
None	-1	

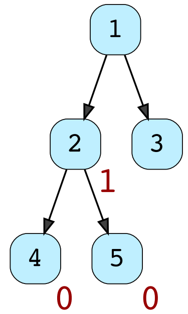
33 of 39




```
def height(node):
  if node is None:
    return -1
  left_height = height(node.left)
  right_height = height(node.right)

  return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1	1	
3		
None	-1	

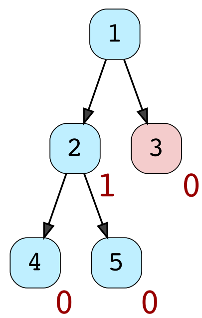
34 of 39




```
def height(node):
 if node is None:
  return -1
 left_height = height(node.left)
 right_height = height(node.right)

 return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1	1	
3		
None	-1	-1

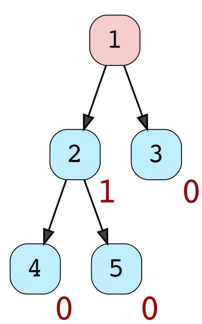
35 of 39




```
def height(node):
 if node is None:
  return -1
 left_height = height(node.left)
 right_height = height(node.right)

 return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1	1	
3		

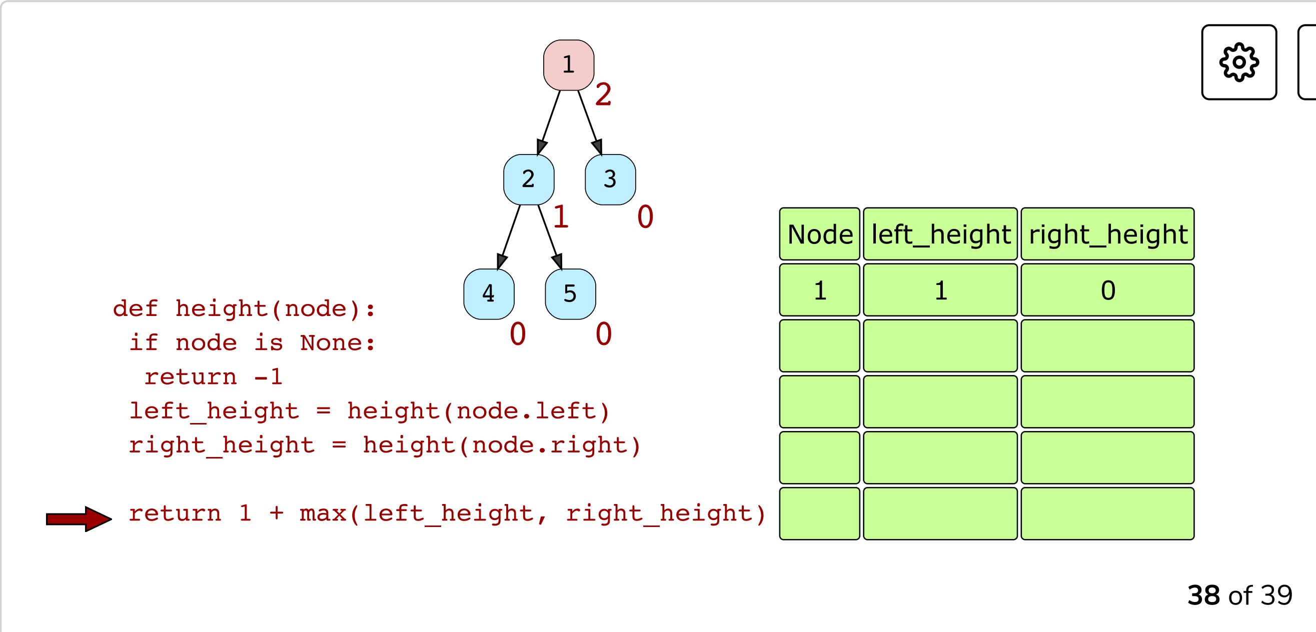
36 of 39

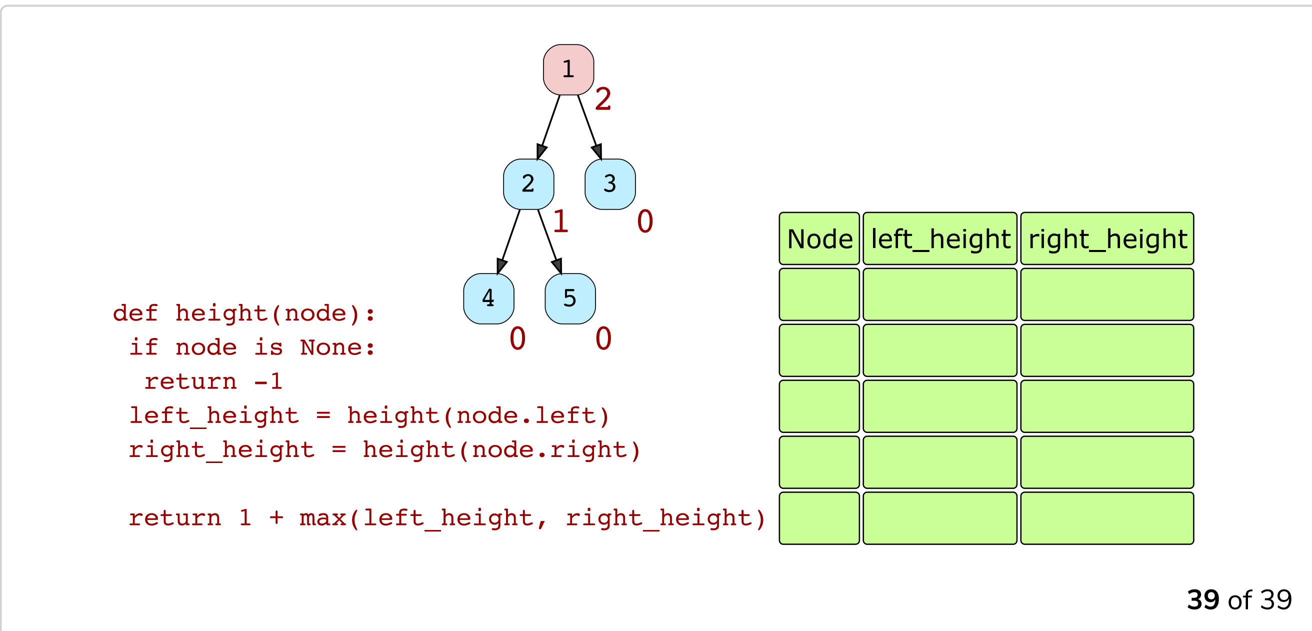



```
def height(node):
 if node is None:
  return -1
 left_height = height(node.left)
 right_height = height(node.right)

 return 1 + max(left_height, right_height)
```

Node	left_height	right_height
1	1	0

37 of 39





I hope the visuals are helpful to understand the algorithm.

In the code widget, the `height` method is made part of the `BinaryTree` Class. Write your test cases to verify the `height` method. A sample test case has been given to you.

```
            traversal = self.postorder_print(start
            traversal = self.postorder_print(start
            traversal += (str(start.value) + "-")
```

```
        return traversal

    def height(self, node):
      if node is None:
        return -1
      left_height = self.height(node.left)
      right_height = self.height(node.right)

      return 1 + max(left_height, right_height)

# Calculate height of binary tree:
#     1
#    / \
#   2   3
#  / \
# 4   5
#
tree = BinaryTree(1)
tree.root.left = Node(2)
tree.root.right = Node(3)
tree.root.left.left = Node(4)
tree.root.left.right = Node(5)

print(tree.height(tree.root))
```

```
Output                                                  0.16s

 2
```

This sums up our content on Binary Trees. Get ready as we have a challenge regarding binary trees waiting for you in the next lesson.

← Back

Reverse Level-Order Traversal

Next →

Exercise: Calculating the Size of a Tree

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/calculating-the-height-of-a-binary-tree__binary-trees__data-structures-and-algorithms-in-python)

Exercise: Calculating the Size of a Tree

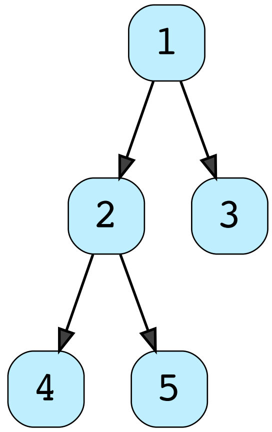
Challenge yourself with an exercise in which you'll have to calculate the size of a binary tree!

We'll cover the following

Problem

The size of the tree is the total number of nodes in a tree. You are required to return the size of a binary tree given the root node of the tree.

Below is an example illustrated for you:



Size Of the Tree = 5

Coding Time!

In the code below, `size_` is a class method of the `BinaryTree` class. You cannot see the rest of the code as it is hidden. As `size_` is a class method, please make sure that you don't change the indentation of the code provided to you. You are required to write your solution under the method prototype and return the size of the tree from the method.

Good luck!

```
    def size_(self, node):
      if node is None:
        return 0
      left_size = self.size_(node.left)
      right_size = self.size_(node.right)
      return 1 + left_size + right_size
```

Show Results Show Console

0.23s

3 of 3 Tests Passed

Result	Input	Expected Output	Actual Output	Reason
✓	size(tree)	5	5	Succeeded
✓	size(tree)	7	7	Succeeded
✓	size(tree)	1	1	Succeeded

Solution Review: Calculating the Size of a Tree

This lesson contains the solution review for the challenge of calculating the size of a binary tree.

We'll cover the following

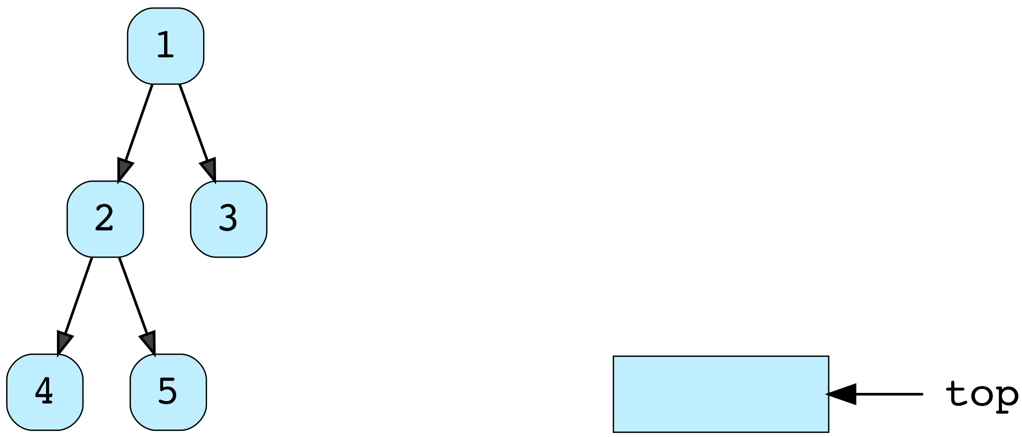
In this lesson, we will review a solution to the problem of determining the size of a binary tree.

The "size" of a binary tree is the total number of nodes present in the binary tree. We will explicitly define this quantity in greater detail and cover a strategy for how one may calculate this quantity in the binary tree data structure we have been building in this chapter.

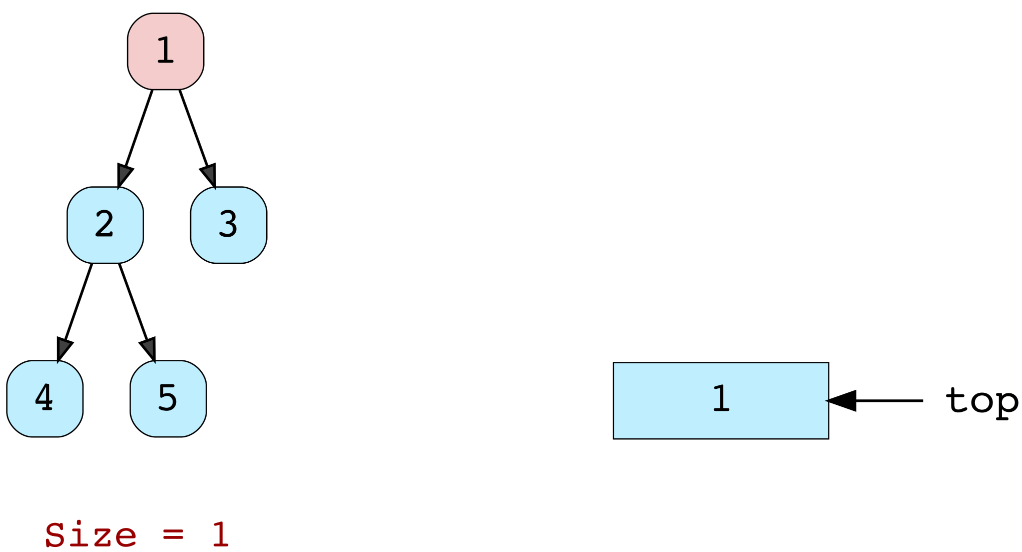
We will discuss an iterative and a recursive approach for solving this challenge.

Iterative Approach

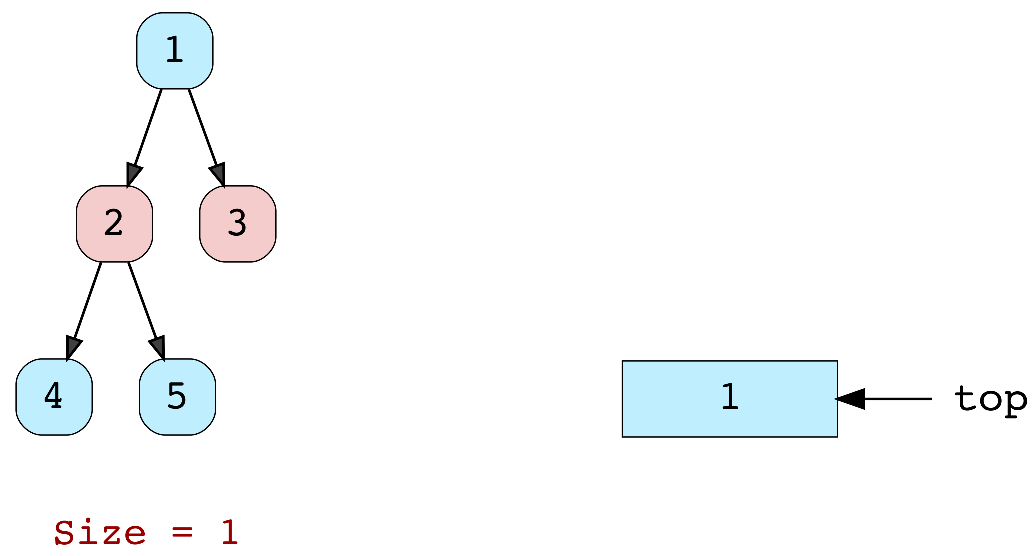
In the iterative solution, we'll make use of a stack on which we can push the starting node and increment size by `1`. Next, we'll pop elements and push their children on to the stack if they have any. For every push, we'll increment size by `1`. When the stack becomes empty, the count for the size will also be final. Have a look at the slides below to check out the algorithm:



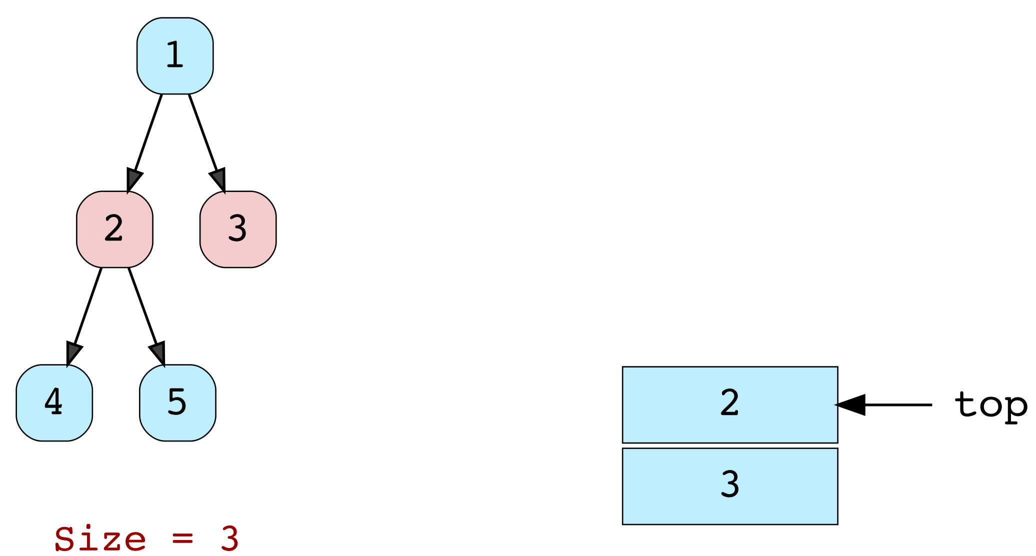
1 of 8



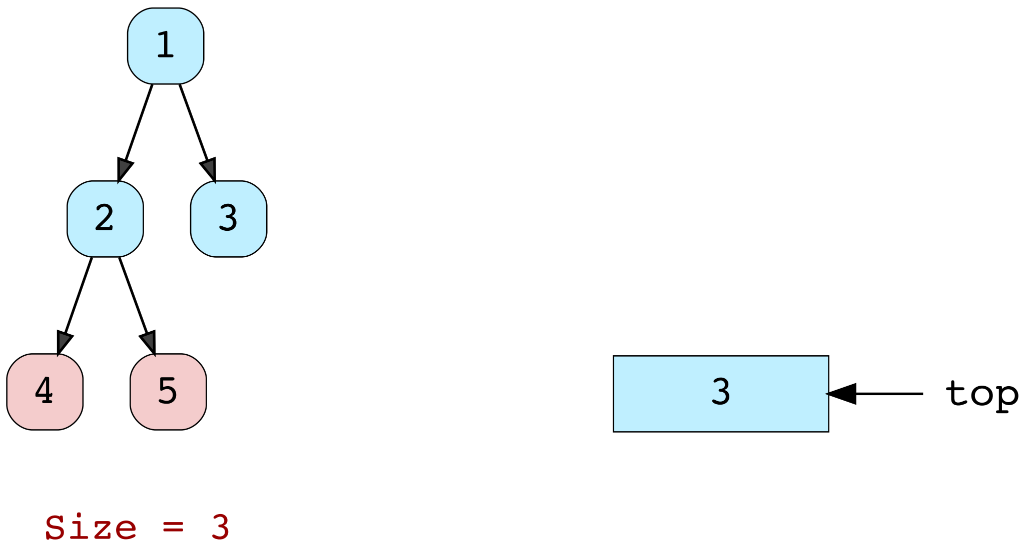
2 of 8



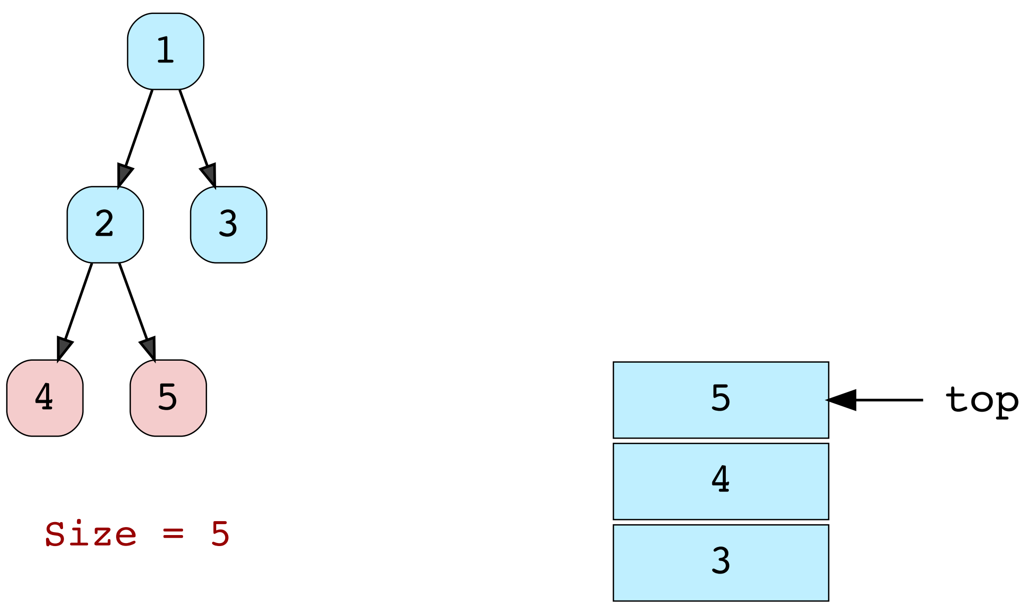
3 of 8



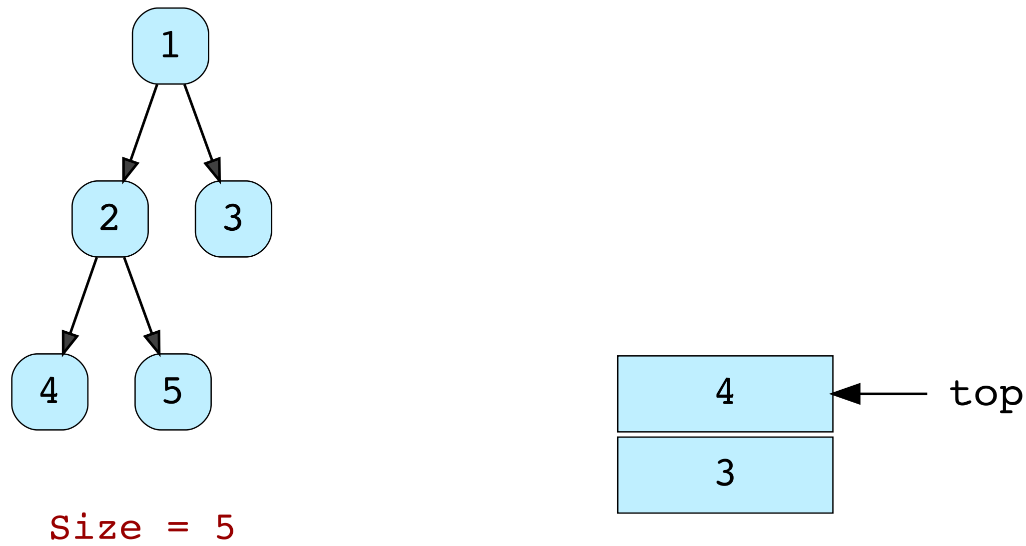
4 of 8



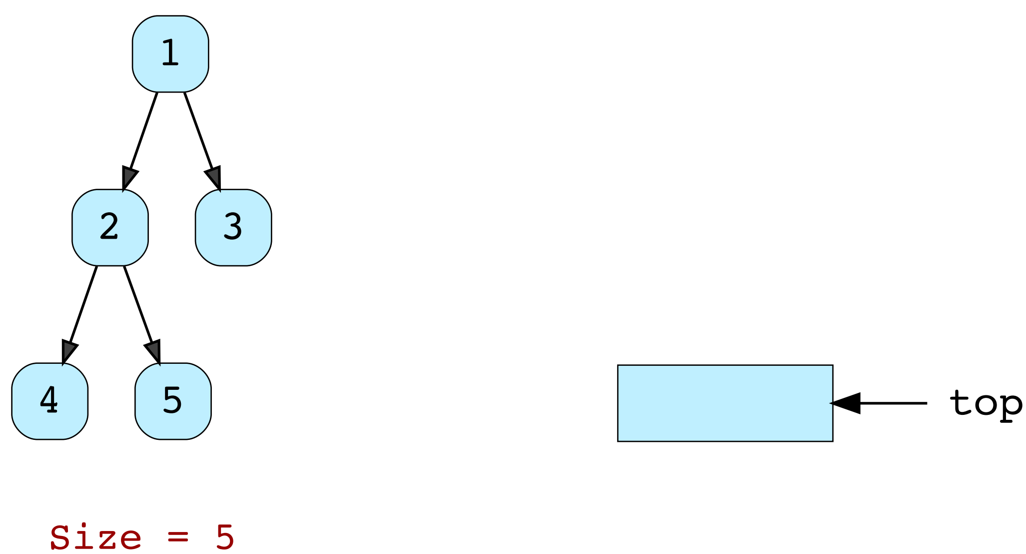
5 of 8



6 of 8



7 of 8



8 of 8

Here is the implementation of the algorithm illustrated above in Python:

```
1 def size(self):
2   if self.root is None:
3       return 0
4
5   stack = Stack()
6   stack.push(self.root)
7   size = 1
8   while stack:
9     node = stack.pop()
10    if node.left:
11      size += 1
12      stack.push(node.left)
13    if node.right:
14      size += 1
15      stack.push(node.right)
16  return size
```

Lines 2-3 contain the edge case which checks for an empty tree and returns `0` in that case. On **line 5**, we declare `stack` to a `Stack` object and push the root node on to the stack on **line 6**. After the push, we initialize `size` to `1` as we have a node present in the stack.

Next, we have a `while` loop which runs as long as `stack` is not empty. On **line 9**, we pop from the stack and store the popped element in `node`. On **lines 10-15**, we check if `node` has a left or right child and push the child on to the stack while also incrementing the `size` by `1`. Finally, when `stack` is empty, the `while` loop terminates, and `size` is returned on **line 16**.

Recursive Approach

We will recursively traverse the nodes and keep track of the count of the nodes visited. Check out the implementation below:

```
def size_(self, node):
  if node is None:
```

```
        return 0
    return 1 + self.size_(node.left) + self.size_(n(
```

Recursively speaking, the size of the tree is the size of the left subtree of the root node + the size of the right subtree of the root node + 1 (for the root node).

The base case is that an empty binary tree has a size of 0 so when `node` becomes `None`, we return `0` as a count. Otherwise, we return `1` plus the count from the recursive call on the left and the right subtree.

Now, this was pretty straightforward. In the code widget below, you can play around with the entire implementation of `BinaryTree` that we have covered so far in this chapter.

```
        size = 1
        while stack:
            node = stack.pop()
            if node.left:
                size += 1
                stack.push(node.left)
            if node.right:
                size += 1
                stack.push(node.right)
        return size

# Calculate size of binary tree:
#     1
#    / \
#   2   3
#  / \
# 4   5
#
tree = BinaryTree(1)
tree.root.left = Node(2)
tree.root.right = Node(3)
tree.root.left.left = Node(4)
tree.root.left.right = Node(5)

print(tree.size())
print(tree.size_(tree.root))
```

259

Output 0.16s

```
5
5
```

I hope you had fun learning about Binary Trees. In the next chapter, we have a different type of binary tree, i.e., the binary search tree. Stay tuned to find out more!

Back
Exercise: Calculating the Size of a Tree

Next
Quiz

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/solution-review-calculating-the-size-of-a-tree__binary-trees__data-structures-and-algorithms-in-python)